Blitz-64

Instruction Set Architecture Reference Manual

ISA Version: 2.1

Harry H. Porter III
Portland State University

HHPorter3@gmail.com

14 December 2023

This document describes the Instruction Set Architecture (ISA) for the Blitz-64 processor core. It documents all the machine instructions as well as the assembly code notation for these instructions.

Available Online: Blitz64.org/Documentation/Blitz64-ISA.pdf

Table of Contents

List of Instructions

Chapter 1: Introduction

What is originality? Undetected plagiarism.
— Dean William R. Inge

Quick Summary

- Blitz-64 introduces a novel 64-bit “Instruction Set Architecture” (ISA).
- The goals of the Blitz-64 project are:
    - — Create a complete hardware / software system
    - — Simple, small, easy to understand
    - — Fully functional and fully modern
    - — Reliability, security, and error handling are emphasized
- This project is open, not proprietary
- Software and documents use dates instead of version numbers

Instruction Set Architectures

An Instruction Set Architecture (ISA) defines, describes, and specifies how a particular computer processor core works. The ISA describes the registers and all the machine instructions. The ISA specifies exactly what each instruction does and how it is encoded into bits.

The ISA forms the **interface between hardware and software**. Hardware engineers design digital circuits to implement a given ISA and software engineers write code (operating systems, compilers, etc.) based on a given ISA specification.

There are a number of Instruction Set Architectures in widespread use, for example:

x86-64 (AMD, Intel)
ARM (ARM Holdings)
SPARC (Sun/Oracle)
RISC-V (Berkeley/open source)

Most of these ISAs are **proprietary** and very complex. The **details are obscured** in lengthy manuals and some details of the ISA are not made public at all. Furthermore, the widely used ISAs have been around for years and their designs carry baggage as a result, e.g., for **backward compatibility**. Since these **legacy designs** were first created, we've learned more about how to design computers. Changes in silicon hardware technology have also had an impact on which design choices are now optimal. The RISC-V project attempts to address the issues of open source and interoperability, and heavily influences Blitz-64.

In this document we define and describe a new ISA called Blitz-64.

Goals and Principles: Personal Statements

The following are the guiding goals of the Blitz-64 architecture.

- Simple, small, modest
- Understandable
- Reliable
- Good error reporting/recovery
- Secure against malware
- No desire to support virtualization / hypervisors (due to security concerns)
- Programmable, pleasing design
- Encourage assembly language and kernel programming and experimentation

All modern processor cores have become far too complex for any single individual to understand. My primary goal is to create a computer that is **simple** enough for one person to understand, yet fully modern and practical.

The way I hope to achieve simplicity is to design the entire system (the ISA, all the system software, and good documentation) alone, myself. The resulting system must, by necessity, be radically simpler than existing computers. A key aspect to making any design simpler is to make it smaller. Size and complexity are strongly correlated. A system designed by one person must be **fairly small and modest** and, as a result, it will necessarily be simpler and easier to understand.

On the hardware side, modern computer cores (ARM, x86-64, etc.) are just too complicated to be understood by any single human being. They are designed by vast teams of specialists; they incorporate legacy designs; their documentation comprises thousand of pages, and they are proprietary and at least partially shrouded in corporate secrecy.

On the software side, modern operating systems contain millions of lines of code written over the course of many decades, by vast numbers of programmers. Much of the code is written in "C", which is notoriously difficult to read, modify, and validate. This is unquestionably true of Apple, Windows, and Linux software. Nobody can fully comprehend a million lines of code; these large chunks of software must remain mysterious black boxes. So instead, programmers today blindly trust and build on top of a gigantic accumulation incompletely understood software. It's remarkable that today's software works as well as it does.

It is easier to use, trust, and rely on systems that we understand. A primary goal of Blitz-64 is to create a complete, modern, and functional computer ISA and collection of system software that is **understandable** by a single person.

Elegance of design is always a laudable goal. Elegance and beauty are correlated with simplicity and size. By keeping the design simple and small, I believe that elegance of design will follow.

As computers are growing more complex and integrated into society, reliability is becoming ever more critical. The more complex a system is, the more difficult it is to verify correctness and repair bugs. Making systems simpler contributes to greater reliability. But beyond simplicity, many small design decisions along the way determine whether **performance execution speed** or **reliability** is preferred and optimized.

To increase reliability, more error checking must be done at runtime. Furthermore, when errors occur, they must be handled with more care, better reporting, and reasonable recovery. Error checking incurs a performance penalty. Modern systems evolved from ancient, slow computers where performance was the critical bottleneck. The legacy systems, upon which the foundation of all modern software is built, often ignored the possibility of program bugs and focused all effort on execution speed. Back in the day, when program size was measured in tens or hundreds of lines of code, this was a reasonable choice.

The dynamic has obviously shifted, changing the tradeoff analysis. Today's computers are really fast. It may now be the case that performance is being hurt by complexity itself. As the size and complexity of software grows, the reliability of individual parts and components becomes ever more critical. (For example, a failure rate of 0.1% for each part might be acceptable for a system with 100 parts, but is totally unacceptable with a million parts.)

I recognize that performance is very, very important, but I reject the "performance at all costs" mentality. One of my goals is to perform **greater runtime error checking** and **improved error recovery**, even at the cost of performance. The radical choice I make is to sacrifice performance for increased reliability, whenever there is a choice.

As an example, the Blitz-64 architecture specifies overflow detection and exception processing on standard arithmetic computations, like the ADD instruction. In the "C" language, an overflow results in no error processing and the program proceeds using incorrect values. In other words, the program fails silently. For any program that has not undergone a thorough numerical analysis (in other words, almost every program), this approach is abominable.

Simplicity also impacts physical reliability. In order to increase the reliability of computer circuits in the face of physical insults (e.g., radiation, temperature extremes, and other environmental problems) simplicity of the ISA has several benefits. First, simpler designs can be implemented with fewer transistors. Given a fixed die size, this allows the individual transistors and wires to be made physically larger. Bigger transistors are more fault tolerant, which increases the circuit's reliability. Second, the small size of an implementation allows more space for redundancy, and duplication is another important approach to fault tolerance. A simple computer with a small footprint can be replicated several times to increase reliability.

Modern computer systems are increasingly susceptible to malware, intrusion, and hacking. In addition to guarding against physical insults, the threats of intentional attack require careful attention in ISA designs.

The approach with current systems seems to be the "whack-a-mole" strategy: when a security hole is uncovered, the hole is patched. Then, wait and repeat. With a gigantic body of legacy software — millions of lines of code, which nobody really understands — the "whack-a-mole" approach seems to be the only viable strategy.

My approach to **increased security** includes creating a smaller, simpler design, improving error detection, and assuming the presence of "black hat" players (bad guys) in all domains, at all times.

The goal of creating software that is secure, reliable, and bug-free is obviously both worthy and elusive. A key approach to making a system secure is to make it reliable and bug-free. So my focus on simplicity and reliability is, implicitly, a focus on security.

In order to verify a computer system, to find and patch security holes, it is necessary to thoroughly review and analyze the system design. With complex ISAs and millions of lines of code, the task of verification is problematic. Simplicity and smallness help a lot.

Another security threat involves embedding spyware or malware within system software. Such software remains present during normal operation and can act as a backdoor for black hat access to private data at any time. Embedded backdoor software can also perform secret surveillance of behavior and activity on the computer, compromising the trust and security of the system.

Spyware can be injected into the system software at many levels. My approach to shutting out spyware and embedded malware involves:

- Designing and implementing all the software from scratch
- Completely reimplementing the boot process
- Banning dynamically alterable firmware
- Securely controlling kernel updates
- Keeping the software small enough that it can be entirely reviewed
- Performing system design and implementation in a sort of clean-room isolation

In particular, **hypervisors** and emulated systems are considered to be a threat to security. It is difficult for kernel software to be certain that it is running on a bare machine, but it is critical to security. For example, a kernel is intended to prevent security leaks, but if that kernel is being emulated or run in a hypervisor context, all the actions of the kernel are subject to surveillance and manipulation.

There is currently a trend toward increased use of hypervisors. Typically, a user wants to own a single computer, but be able to run software developed for the Mac, Windows, Linux, etc. operating systems. The clever approach is to run multiple operating systems on top of hypervisor software. As a result, modern ISAs are designed with an eye to supporting hypervisor-like software, to make the hosted OSes run faster.

The Blitz-64 system takes the opposite approach. While there seems to be little we can do to prevent software from being executed in an emulated environment, the emulation of kernels should be discouraged due to security issues. The Blitz-64 architecture makes no concessions and no special instructions are added to support the emulation of “kernel mode” software. This is an intentional design decision, not an oversight.

A final goal of the Blitz-64 project is to **support programming for fun** and, in particular, to support assembly language and kernel programming.

Programming on “bare metal” is an acquired taste and certainly does not appeal to the mass of average programmers because of the high level of skill and attention to detail it requires. But there may be a small group of highly proficient hobbyists who want this experience.

I feel that modern computers are simply too complex for programming to be fun. Kernel programming is pretty much impossible. I want to create a computer system that is more than a one-off, home-brew computer. My goal is to design a computer that is small and simple, yet roughly as functional as an ARM or x86-64 machine. Basically, I want to create a computer that programmers will enjoy — that I will enjoy programming.

Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name are used. The document history is:

Date	**Author**	
23 May 2018	Harry H. Porter III	*<initial version>*
28 May 2019	Harry H. Porter III	*<document mostly completed>*
24 May 2021	Harry H. Porter III	*<new instructions added>*
18 October 2022	Harry H. Porter III	*<version 2.0 of ISA>*
30 April 2023	Harry H. Porter III	*<changes to **csr_pgtable**>*
12 September 2023	Harry H. Porter III	*<minor changes>*
14 December 2023	Harry H. Porter III	*<version 2.1 of ISA>*

For details, consult the appendix titled "Recent Changes".

In the spirit of the open-source and free software movements, the author grants permission to freely copy and/or modify this document, with the following requirement:

> ***You must not alter this section, except to add to the revision history. You must append your date/name to the revision history.***

Any material lifted should be referenced.

Relevant Software Tools

The primary software tools relevant to this document are:

- The Blitz-64 **virtual machine** — a "C" program called "`blitz`"
- The Blitz-64 **assembler** — a "C" program called "`asm`"
- The Blitz-64 **linker** — a "C" program called "`link`"

For our purposes, the terms "emulator" and "virtual machine" are synonymous.

Tool	**Version Described Here**	**Coding Status**
`blitz`	*< same date as this document >*	Completed
`asm`	*< same date as this document >*	Completed
`link`	*< same date as this document >*	Completed

Instead of version numbers, the Blitz-64 project uses dates to identify versions of both programs and documents. By comparing dates, you can determine whether this document matches the version of the tools you are using or, if not, which is more recent.

Chapter 2: Terminology and Notation

> *If you can't convince them, confuse them.*
> — Harry S Truman

Quick Summary

- "Halfword" = 16 bits = 2 bytes.
- "Word" = 32 bits = 4 bytes.
- "Doubleword" = 64 bits = 8 bytes.
- Main memory is byte addressable.
- Main memory is Big Endian.
- The notation [n:m] is used to identify bits.
- For example, [63:60] means the most significant (MSB) 4 bits in a doubleword.
- We use KiByte, MiByte, GiByte… instead of KByte, MByte, GByte…
- Alignment (e.g., halfword, word, doubleword) is defined.
- Proper alignment for sizes 8, 16, 32, and 64 bits is defined.
- Properly aligned doublewords are at addresses divisible by 8 (ending in bits 000).
- Integers are represented with signed, two's complement values.
- All arithmetic is done using 64 bits.
- Sign-extension enlarges an integer represented in signed two's complement binary.
- Size reduction (e.g., from 64 to 32 bits) may result in an "overflow" error.

Kilo and Mega Prefixes

There has been some confusion in computer science documentation regarding abbreviations for large numbers. For example:

4K = ?
    4,000
    4,096

We use the following prefix notation for large numbers, which is becoming common in the context of computer architecture:

Prefix		Example	Value		
Ki	kibi	KiByte	2^{10}	1,024	~10^3
Mi	mebi	MiByte	2^{20}	1,048,576	~10^6
Gi	gibi	GiByte	2^{30}	1,073,741,824	~10^9
Ti	tebi	TiByte	2^{40}	1,099,511,627,776	~10^{12}
Pi	pebi	PiByte	2^{50}	1,125,899,906,842,624	~10^{15}
Ei	exbi	EiByte	2^{60}	1,152,921,504,606,846,976	~10^{18}

Contrast this to the standard metric prefixes, which we avoid:

Prefix		Example	Value	
K	kilo	KByte	10^3	1,000
M	mega	MByte	10^6	1,000,000
G	giga	GByte	10^9	1,000,000,000
T	tera	TByte	10^{12}	1,000,000,000,000
P	peta	PByte	10^{15}	1,000,000,000,000,000
E	exa	EByte	10^{18}	1,000,000,000,000,000,000

Bits and Bytes

We use the terms "**byte**", "**halfword**", "**word**", and "**doubleword**", to refer to various sizes of binary data.

	number of bytes	number of bits	example value (in hex)
byte	1	8	`A4`
halfword	2	16	`C4F9`
word	4	32	`AB12CD34`
doubleword	8	64	`0123456789ABCDEF`

A single hex digit can be used to represent 4 bits:

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

The 8 bits within a byte are conveniently expressed with two hex digits. For example:

8 bit byte	In Hex
1010 0100	A4

The 32 bits in a word are given with 8 hex digits. For example:

32 bit word	In Hex
1010 1011 0001 0010 1100 1101 0011 0100	AB12CD34

Sometimes we insert spaces or commas to make long hex values more readable.

These examples show different ways of representing the same doubleword:

```
0123456789ABCDEF
0123_4567_89AB_CDEF
0123,4567,89AB,CDEF
0123 4567 89AB CDEF
01234567 89ABCDEF
```

Often we prefix hex values with “0x” to make it clear they are hex values and not decimal:

```
0x1234
```

The bits within an 8-bit byte are numbered from 0 (lower, least significant) to 7 (upper, most significant).

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

The bits within a 16 bit halfword are numbered from 0 to 15.

The bits within a 32 bit word are numbered from 0 to 31.

The bits within a 64 bit doubleword are numbered from 0 to 63.

```
63      56       48       40       32       24       16        8        0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

We use the following notation to represent a range of bits:

Example	Meaning
[7:0]	All bits in a byte
[63:0]	All bits in a doubleword
[31:28]	The upper 4 bits in a word
[5]	The 6th bit from the right end

Main Memory

Main memory is byte addressable.

Main addresses are 36 bits. We generally express addresses in hex. Here are two equivalent notations we use:

```
8_ABCD_1234
0x8ABCD1234
```

Memory can be viewed as a sequence of bytes:

address	data
(in hex)	(in hex)
0_0000_0000	89
0_0000_0001	AB
0_0000_0002	CD
0_0000_0003	EF
0_0000_0004	01
0_0000_0005	23
0_0000_0006	45
0_0000_0007	67
...	...
F_FFFF_FFFC	E0
F_FFFF_FFFD	E1
F_FFFF_FFFE	E2
F_FFFF_FFFF	E3

“Low” memory refers to smaller addresses, closer to 0_0000_0000. “High” addresses are numerically greater.

Big Endian

Blitz-64 is a big endian architecture.

As an example, assume that main memory holds the following bytes:

address (in hex)	data (in hex)
...	...
E_5000_0004	1A
E_5000_0005	2B
E_5000_0006	3C
E_5000_0007	4D
E_5000_0008	5E
E_5000_0009	6F
E_5000_000A	70
E_5000_000B	81
E_5000_000C	92
E_5000_000D	A3
E_5000_000E	B4
E_5000_000F	C5
...	...

In Blitz-64, the registers are 64 bits (8 bytes) wide. There are several LOAD and STORE instructions, which can move either a byte, halfword, word or doubleword between memory and a register.

Consider a LOADB instruction that loads a byte from address 0xE_5000_0004. After execution, the register will contain:

```
0x0000_0000_0000_001A
```

Consider a LOADW instruction which loads a word from address 0xE_5000_0004. After execution, the register will contain:

```
0x0000_0000_1A2B_3C4D
```

Commentary In a little endian architecture, the order of the bytes is changed whenever data is copied from memory to a register or stored from a register into memory. This can be a source of confusion, particularly when humans look at a printout of memory contents.

As an example, consider this memory:

address	**data**
(in hex)	**(in hex)**
...	...
`E_5000_0004`	`1A`
`E_5000_0005`	`2B`
`E_5000_0006`	`3C`
`E_5000_0007`	`4D`
`E_5000_0008`	`5E`
`E_5000_0009`	`6F`
`E_5000_000A`	`70`
`E_5000_000B`	`81`
`E_5000_000C`	`92`
`E_5000_000D`	`A3`
`E_5000_000E`	`B4`
`E_5000_000F`	`C5`
...	...

Memory can be viewed either as a series of bytes, or as a series of larger units, such as words or doublewords.

With a “big endian” computer, this memory is interpreted as:

address	**data**
(in hex)	**(in hex)**
...	...
`E_5000_0004`	`1A2B3C4D`
`E_5000_0008`	`5E6F7081`
`E_5000_000C`	`92A3B4C5`
...	...

With a “little endian” computer, this memory is interpreted as:

address (in hex)	data (in hex)
...	...
`E_5000_0004`	`4D3C2B1A`
`E_5000_0008`	`81706F5E`
`E_5000_000C`	`C5B4A392`
...	...

Big endian architectures are simpler to understand since the bytes are not reordered during loads and stores.

The primary argument for choosing little endian is legacy compatibility. The two approaches are similar in terms of circuit complexity.

Alignment

A “**halfword aligned**” address is an address that is a multiple of 2. The last bit of a halfword-aligned address will always be 0. Likewise, a “**word aligned**” address is a multiple of 4, and ends with the bits 00. And finally, a “**doubleword aligned**” address will be evenly divisible by 8 and will end with bits 000.

A halfword-sized value is said to be “**properly aligned**” if it is stored at a halfword aligned address. Likewise, a word-sized value is properly aligned if it is stored at a word aligned address. And similarly, a doubleword-sized value is properly aligned if it is stored at a doubleword aligned address.

Blitz-64 requires data to be properly aligned for the LOAD and STORE instructions.

Full-sized instructions are 32 bits in length. Compressed instructions are 16 bits in length. All instructions are required to be halfword aligned. The LSBit of the PC is hardwired to 0, so there can be never be an exception when an instruction is fetched. When the PC is loaded — for example during a BRANCH or CALL instruction — the LSBit is simply ignored; no exception will be generated.

Commentary BRANCH and CALL instructions are normally generated by a compiler or assembler, which will always place the target instruction on a properly aligned address. Therefore, there is little possibility that an error will be made.

However, with LOADs and STOREs, the address may come from a programmer computed pointer, which may easily be in error. Rather than silently ignoring the last 1, 2, or 3 bits and loading/storing from an incorrect location, an "Unaligned LOAD/STORE Exception" will be signaled.

Signed Numbers

Integers are represented in binary.

With **unsigned number** representation, only zero and positive integers can be represented. The maximum possible value is determined by the number of bits available and is always 2^N-1, where N is the number of bits.

	Size	
	in bits	**Range of values**
byte	8	0 … 255
halfword	16	0 … 65,535
word	32	0 … 4,294,967,295
doubleword	64	0 … 18,446,744,073,709,551,615 (≈ 2 × 10^{19})

Signed numbers are represented using "**two's complement**" representation. The most significant bit gives the sign (1=negative; 0=zero or positive).

	Size	
	in bits	**Range of values**
byte	8	-128 … 127
halfword	16	-32,768 … 32,767
word	32	-2,147,483,648 … 2,147,483,647
doubleword	64	-9,223,372,036,854,775,808 … 9,223,372,036,854,775,807

To make things simpler, we define the following constants:

Name	Decimal	Hex (64 bits)
MIN_8	-128	`FFFF_FFFF_FFFF_FF80`
MAX_8	127	`0000_0000_0000_007F`
MAX_UNSIGNED_8	255	`0000_0000_0000_00FF`
MIN_16	-32,768	`FFFF_FFFF_FFFF_8000`
MAX_16	32,767	`0000_0000_0000_7FFF`
MAX_UNSIGNED_16	65,535	`0000_0000_0000_FFFF`
MIN_32	-2,147,483,648	`FFFF_FFFF_8000_0000`
MAX_32	2,147,483,647	`0000_0000_7FFF_FFFF`
MAX_UNSIGNED_32	4,294,967,295	`0000_0000_FFFF_FFFF`
MIN_64	-9,223,372,036,854,775,808	`8000_0000_0000_0000`
MAX_64	9,223,372,036,854,775,807	`7FFF_FFFF_FFFF_FFFF`

The Blitz-64 architecture relies entirely on 64 bit signed integers. There is only one type for integers.

Arithmetic on 32 bit quantities is not supported, although there are instructions to enlarge and shrink values between 8, 16, 32, and 64 bits.

Note that the range of signed doublewords is sufficient to represent every *byte*, *halfword*, and *word* value regardless of whether it is *signed* or *unsigned*.

Commentary Signed 64 bit integers seem both necessary and sufficient for computer arithmetic. There seems to be no good reason to include support for “unsigned 64 bit integer” operations.

The range of signed doublewords is adequate for expressing quantities such as an “astronomical unit” in microns, the number of seconds since the big bang, or the world GDP in hundredths of a cent. Unfortunately, the range of 32 bit words is inadequate for many things, such as counting humans, the US federal debt in dollars, the number milliseconds since January 1, 1970 (widely used by computers), or the number of bytes of main memory in typical smartphones. Any programmer who uses 32 bit integers needs to think very, very carefully about overflow conditions.

The use of unsigned data types made sense in the past, when the word sizes were smaller. In some applications, the difference between a maximum value of 127 and 255 (for byte-sized data), or between 32,767 and 65,535 (for 16 bit data) was important and critical, and worth sacrificing the ability to represent negative values.

It is even conceivable that some applications needed numbers between 2,147,483,647 and 4,294,967 295 (for 32-bit data), while at the same time, never needing negative values.

However, it's virtually impossible to imagine an application for which unsigned 64 bit numbers are appropriate. For such an application, the expected values would be expected to exceed 9,223,372,036,854,775,807*, and yet be guaranteed to never exceed 18,446,744,073,709,551,615, and also be guaranteed to never be negative*!

Commentary The cost of using "unsigned" binary numbers is that negative values must be thrown out. Negative numbers are obviously useful and shouldn't be ignored or excluded. Throwing out the negative numbers is a bad, anti-mathematical idea. It's dangerous because we know it causes all sorts of program bugs; it makes the discrepancy between "computer integers" and "mathematical integers" vastly greater; and a proliferation of different datatypes complicates programming.

In Blitz-64, if the programmer wishes to force some number into one of the limited, legacy ranges, he/she can easily write tests such as:

```
    if (x<0 || x>MAX_UNSIGNED_32) …
```

Commentary In any core processor, the speed of addition is critical since addition is involved in:

- Incrementing the PC.
- Performing address calculations in LOAD, STORE, BRANCH, … instructions.
- Implementing the ADD and SUB instructions, for loop control, arrays, etc.

The Blitz-64 architecture does not support arithmetic on integer data of size byte, halfword, or word. How much of a performance penalty does this radical decision incur?

In modern cores, we can assume that addition is implemented with carry lookahead units (CLA), each with 4 inputs. Thus, the carry lookahead tree has a branching factor of 4 and the depth of the tree determines the gate delay for the adder unit. A 16 bit adder will require 2 CLA levels (4 × 4) to add 16 bits. A 32 bit adder will require 3 levels, since 4 × 4 is not enough. However, a 3 level tree will also be sufficient for a 64 bit adder, since 4 × 4 × 4 = 64.

Therefore, **64-bit addition incurs no performance penalty** over 32 bit addition. This holds for subtraction, as well.

Concerning multiplication, the execution time is constrained by the time to add a column of numbers. The setup and sign-adjustment logic incurs a constant delay which does not depend on word size.

For 32 bit multiplication, a set of 32 numbers must be added. For a 64 bit multiplication, a set of 64 numbers must be added. Many of the additions can be done in parallel, and the final result sum can be determined in log time. A set of 32 numbers can be added using a tree of adders of depth 5. A set of 64 numbers can be added using a tree of adders of depth 6. Therefore, the time required **to multiply 64 bit values** will be **no more than 20% greater** than the time required to multiply 32 bit values.

Thus, our (perhaps counterintuitive) **conclusions** are:

- There is no significant performance penalty to pay for performing all arithmetic using 64 bits.

- The simplicity to be gained by eliminating legacy data types (i.e., “unsigned”, “byte”, “halfword”, and “word”) is well worth any small performance cost.

Sign-Extension

A value of one size can be “sign-extended” to a larger size. For example, a 32 bit word can be sign-extended to 64 bits.

The sign-extension operation does not change the integer value of the number.

The sign-extension operation looks at the sign bit (i.e., the most significant bit) of the smaller number. Then, that bit value is replicated as necessary to fill additional bits on the left, most significant end of the smaller value, until it is the required larger size.

For example, sign-extending a 16 bit value to 64 bits will look at bit [15] of the input value. If it is “1”, the number is negative. To sign-extend it to 64 bits, the uppermost 48 bits, i.e., bits [63:16], will be filled with “1”. Otherwise, the uppermost bits will be filled with “0”.

Many Blitz-64 instructions include a 16 bit “immediate” value, which is encoded directly within the instruction. This immediate value is sign-extended to 64 bits before being used.

Size Reduction

Often it is necessary to take a larger value and reduce its size. For example, a register may contain a doubleword value (i.e., 64 bits) and we may want to reduce it to a halfword (i.e., 16 bits).

A size reduction can be performed by simply cutting off (i.e., ignoring, eliminating) the uppermost bits.

If the original value happens to lie within the range representable by the smaller size, then there is no problem. The value remains unchanged by the operation.

If the original value does not lie within the range representable by the smaller size, then the new value will be numerically different. This is considered a form of “overflow”, in the sense that the operation has resulted in a mathematically incorrect result.

Looking at a value, we can easily determine whether a size reduction will result in overflow or not. For example, if we are reducing a 64 bit value to 16 bits, we ask whether the upper 48 bits (i.e., bits [63:16]), which will be discarded, are all equal to the sign bit (i.e., bit [15]) of the new, smaller result. If so, there is no problem. In other words, we ask whether bits [63:15] are either all 0s or all 1s. If the uppermost 49 bits are all equal, there is no problem, but if both 0s and 1as are present, the size reduction operations will cause an overflow error.

Chapter 3: Architectural Summary

A doctor can bury his mistakes but an architect can only advise his clients to plant trees.
— Frank Lloyd Wright

Quick Summary

- Register size: 64 bits.
- Number of general purpose registers: 16.
- Zero register: r0 always reads as zero and acts as a destination for unneeded results.
- All remaining registers (r1, r2, … r15) are general purpose and equally functional.
- Natural data size: 64 bits (i.e., doubleword)
  — Integer overflow is never ignored; an exception is always generated.
  — All arithmetic is done using 64 bit signed integers.
- Floating Point:
  — No separate floating-point regs. General purpose regs are used.
  — Floating point precision: Double only; there is no single precision.
- Main memory is Big Endian.
- Instructions are 32 bits in size.
- Compressed instructions are multiples of 8 bits in size.
- Number of privilege modes: 2 (Kernel and User).
- Number of Control and Status Registers (CSRs): 16.
- Size of Control and Status Registers (CSRs): 64 bits.
- Program-generated addresses: 36 bits.
- Maximum Physical Memory: 16 GiBytes.
- Memory-Mapped Address Range: 16 GiBytes.
- Maximum Virtual Address Space: 32 GiBytes.
- Page size: 16 KiBytes.
- Virtual Memory System: Page tables are supported.

Memory, Addresses, and Memory-Mapped I/O

Installed physical memory up to 16 GiBytes can be addressed without using page tables.

There is an additional 16 GiBytes of physical address space allocated for memory-mapped I/O.

Virtual address spaces use page tables that map program generated addressed into physical addresses. Virtual address spaces can be up to 32 GiBytes in size.

All program-generated addresses are 36 bits. Note that 2^{36} = 64 Gi.

Supporting Larger Main Memory

Using virtual memory and page tables, up to **16 TiBytes of physical memory** is supported.

In the basic configuration, up to 16 GiBytes of physical memory is supported in a simple, uniform, linear address space. This should suffice for many applications. Larger memory sizes can be supported, but these can only be accessed via virtual addresses and the page table mapping.

Virtual memory and the memory mapping scheme are discussed in a later chapter.

The limitation on addresses to 36 bits might seem naïve and overly restrictive, but this is an important design choice and was not made lightly. ISA design involves a trade-off between (1) a large number of registers, (2) a small instruction size, (3) long addresses, and (4) the number of instructions required to load arbitrary addresses. Since you can't have it all, our design decisions involve a compromise on these issues.

Remember that main memory is only one tier in a memory hierarchy ranging from terabytes of solid state stable memory to megabytes of fast cache. Main memory is properly viewed as a staging ground in which programs and data are held, in order to supply the core with grist for computation. It is nothing more than a form of per-core cache between a processing unit and shared data sources. We predict that the bandwidth between main memory and the core/ fast-cache circuitry will remain a performance bottleneck; 16 GiBytes seems more than adequate to keep a single core busy. Since Blitz-64 cores may be deployed in multi-core systems with 100s or 1000s of cores, the per-core limit of 16 GiBytes is properly understood as imposing a limitation on the entire core array measured in terabytes or exabytes.

The Processor State

The entire state of a running Blitz-64 core consists of:

- The general purpose registers (r0, … r15)
- The Program Counter (PC)
- A set of 16 "Control and Status Registers" (CSRs)

(Here we mean the directly visible state of the core, observable by software; additional state, such as related to pipeline stages, cache contents, etc. should not affect software functionality or correctness.)

The Registers

The **general purpose registers** are 64 bits (a doubleword, 8 bytes) in width.

There are **16 registers**.

The registers are named **r0, r1, r2, … r15**.

Register r0 is a special "**zero register**". When read, its value is always 0x0000_0000_0000_0000. Whenever there is an attempt to write to r0, the data is simply discarded.

All other registers are treated identically by the ISA; there is nothing special about any register.

By convention, several registers have special functions and these registers are given alternate names. The assembler will accept either name.

	Alternate Name	Function
r0		Zero
r1		Argument 1 / Return Value
r2		Argument 2
r3		Argument 3
r4		Argument 4
r5		Argument 5
r6		Argument 6
r7		Argument 7
r8	t	Temp register, used by assembler/linker
r9	s0	Work reg (caller-saved)
r10	s1	Work reg (caller-saved)
r11	s2	Work reg (caller-saved)
r12	tp	Thread data pointer
r13	gp	Global data pointer
r14	lr	Link register
r15	sp	Stack pointer

All registers are treated equally by the ISA, with the exception of **r0**. Their special functions arise solely in how the programmer uses them in instructions.

Register Usage Conventions

The registers **r1** … **r7** are used to pass arguments to functions and methods and **r1** is used to return results. Registers r1 … r7 are also used as general working registers to hold local variables and intermediate results within a function or method. The compiler or assembly language programmer is free to use them as desired within functions or methods. If fewer than 7 arguments are passed, then the remaining registers can be used as general work registers in the function/method. If more than 7 arguments are passed, or if any argument is larger than a doubleword, then those arguments will be passed on the stack. If most of the registers are taken up with argument passing and the function/method has immediate need for some temporary work registers, then the function/method may, at its discretion, immediately upon entry, store the less urgently needed arguments in the stack frame, thereby freeing up registers for other uses.

The Blitz-64 calling convention sets aside a fairly large number of registers for argument passing. Each argument must be collected by the calling code and moved into a known, agreed-upon location by the caller's code. Even if the argument were to be placed on the stack, the caller would at least need to move the argument into a register temporarily to do this.

When the compiler is compiling a function, it cannot know whether it is best for the value to be placed in a register or written to the stack. Only the called function can make an informed decision about this. Therefore, the Blitz convention is to place a large number of arguments (up to 7) in registers and let the called function store some of all of them to memory, at its discretion. Ideally, the called function can avoid moving any arguments to memory.

We considered allocating all available registers to carry arguments, but there are rarely functions with more than 7 arguments and it may be convenient for a function to have some registers free upon entry. We can assume a function with more than 7 arguments is big and complex; having three work registers available may allow the function to achieve much of its task without having to spill registers to the stack frame just to have some work registers to work with. Placing all arguments in registers and therefore leaving no work registers available means that some spills must occur immediately upon entry into the function.

Therefore, we allocate three additional registers called **s0**, **s1**, and **s2** (i.e., **r9**, **r10**, and **r11**) as work registers.

The "**temporary register**" (register **t**, i.e., **r8**) is used by the assembler for some synthetic instructions. When describing the synthetic instructions, this document indicates whether and how register t will be used. The use of register t is "clandestine", in the sense that t is not explicitly named in the synthetic instructions. The programmer and compiler are free to use register t in a function/method, as long as they realize that some synthetic instructions may alter t.

The "caller" of a function/method should assume that registers r1…r7, t, and s0…s2 will trashed (i.e., altered or arbitrarily modified) by the "called" function/method. If the contents are important, the caller should save their contents before calling the function/method. In that sense, r1…r7, t, and s0…s2 are said to be "**caller-saved**".

A "**callee-saved**" register is one in which the caller can assume that the called function will not modify the value. Or more accurately, if the called function needs to use a callee-saved register, it will save it first and then restore it before returning.

In some sense, the registers tp, gp, and sp are callee-saved, since the convention states that they are to have the same value upon return that they had before the call.

We considered setting aside some registers as “callee-saved”.[1]

In a program with multiple threads, each thread may have a block of data specific to that thread. The “**thread pointer register**” (register **tp**, i.e., **r12**) points to this block of data, making it easy for the thread to access its private data. Typically, this register does not change and stays constant during the entire life of the thread.[2]

The “**global pointer register**” (register **gp**, i.e., **r13**) points to a block of memory containing static global variables shared by all functions/methods in all threads, making it easy for the code to access these variables with a single LOAD/STORE instruction using a small offset. The 16 bit immediate offset in LOAD/STORE instructions makes it easy to access data within a 4 page (i.e., 64 KiByte) range by using offsets up to ±32 KiBytes.

Typically, the global data will be placed at the beginning of the virtual address space, i.e., at address 0x8_0000_0000. Therefore, register gp will contain 0x8_0000_8000 which is the start of virtual memory, plus 2 pages (i.e., plus 32 KiBytes), allowing access to the first 4 pages of virtual memory. Register gp will remain constant during the execution of the program.

The “**link register**” (register **lr**, i.e., **r14**) is used in function/method invocation. The CALL instruction will store the return address in register “lr” and the RET instruction will jump back to that address. If the function/method is a leaf routine (i.e., if it doesn’t invoke other functions/methods) then the return address can remain in lr until the RET instruction causes the return. Otherwise, the value of lr must be saved somewhere, typically on the stack, and retrieved before the return.

The “**stack pointer register**” or “stack top” (register **sp**, i.e., **r15**) points to the runtime stack. By convention, the stack grows downward from high memory (larger addresses) toward low memory (smaller addresses). By convention, sp will point to the first byte of the stack, i.e., the most significant byte of the doubleword sitting at the top of the stack. By convention, the stack will always grow in multiples of 8. In other words, sp will always contain a doubleword aligned address.

[1] In fact, s0…s2 were originally callee-saved with the “s” standing for “saved”.

[2] In programs which have only a single thread and no need for a thread pointer, this register might instead be used as a callee-saved register. But beware that called functions will likely use this register to locate various parameters; using register tp as a callee-saved register is not practical.

Although floating point instructions are defined, there are no separate **floating point registers**. Instead, floating point data is kept and manipulated in the general purpose registers.

There is a **program counter (PC)** whose size is 36 bits.

Thus, the PC can contain any number within 0x0_0000_0000 ... 0xF_FFFF_FFFF. Any attempt to load the PC with a number outside this range is legal: bits [63:36] will be ignored with no overflow exception signaled.

Commentary Many processor ISAs include a “condition code register.” Such a register usually contains bits such as:

- Sign / Negative Value
- Zero / Equal
- Carry Bit
- Overflow

In such ISAs, there is usually a COMPARE instruction (which will set bits in the status register) and several BRANCH instructions (which will test the status register bits and conditionally jump).

The normal pattern of most code is to execute a COMPARE instruction and, immediately afterward, execute a BRANCH instruction. They go together and effectively perform a single “test-and-jump” operation.

Blitz-64 does not include a “condition code register.” Instead, the BRANCH instructions will perform both the test and the conditional jump. By combining them into a single instruction, greater performance efficiency can be achieved whenever this “test-and-jump” operation must be performed.

Control and Status Registers (CSRs)

The “**Control and Status Registers**” (**CSRs**) are used by the protection and privilege system. The privilege system is used by the OS kernel to protect itself and manage user-level processes. The CSRs are also used for interrupt processing, thread switching, and virtual memory manipulation.

At any moment, the processor will be executing either in “**user mode**” or in “**kernel mode**”. OS kernel code is executed in kernel mode and application programs are executed in user mode.

Each instruction is either “**privileged**” or “**non-privileged**”. When the core is running in user mode, only non-privileged instructions may be executed. When running in kernel mode, all instructions are usable.

Changing the privilege mode is accomplished by writing to a CSR. A single bit in the status register (**csr_status**) determines the current privilege mode.

CSRs can only be read/written when running in kernel mode.

There are 16 CSRs.

Each CSR has a special name and each has a unique function. Reading and/or writing a CSR will have an effect on the processor operation. The CSRs are read and written with just a couple of general-purpose instructions. The instructions to read/write the CSRs are privileged and can only be executed in kernel mode.

In order to understand the user-mode instruction set and to create user-level code, the CSRs can and should be ignored, especially on your first introduction to Blitz-64.

Virtual Memory

Blitz-64 supports virtual memory. For each virtual address space, there will be a page table stored in memory. The page table is organized as a tree of nodes and, at any time, the root of the current page table is pointed to by a control and status register (CSR) named **csr_pgtable**.

Pages in the virtual address space can be marked as

- valid / invalid
- writable
- executable
- copy-on-write
- dirty

Any attempt by user code to access a page in violation of the permissions for that page will cause an exception.

The virtual memory architecture and page tables are described in the chapter titled “Memory, Address Spaces, and the Page Table”.

Chapter 4: Instruction Formats

Quick Summary

- Machine instructions are 32 bits long.
    The 16 registers are encoded in fields of 4 bits.
    Immediate values occupy fields of either 16 or 20 bits.
- There are 4 formats of instructions, called A, B, C, and D.
- Assembly syntax is summarized.
    The destination register is schematically called “RegD”.
    The operand registers are schematically called “Reg1”, “Reg3”, and “Reg3”.
- Compressed instructions will be defined and specified in the future.
    Compressed instructions are variable in length.
    Compressed and full-sized instructions can be distinguished by their opcodes.

Compressed and Full-Sized Instructions

There are two types of instructions:

- Full-sized instructions (32 bits)
- Compressed instructions (variable length)

Each compressed instruction is exactly equivalent in function to a 32 bit full-sized instruction. However, there may be many 32 bit instructions for which there is no equivalent compressed version.

A major performance bottleneck is the time required to fetch instructions from main memory. The entire purpose of compressed instructions is to reduce the size of code.

The full-sized and compressed instructions may be intermixed. There is no “mode” bit to put the processor into “compressed instruction mode”, as there is in some processors.

Commentary Reducing the size of code results in increased processor performance since it allows more instructions to be cached, reducing the time to fetch instructions from main memory, which is often a performance bottleneck.

In a typical hardware implementation, when a compressed instruction is fetched and loaded into the Instruction Register (IR) prior to being executed, the hardware will notice that it is a compressed instruction. At that time, the compressed instruction will immediately be expanded into the equivalent 32 bit instruction. Thereafter, there is no need for any additional hardware logic to support the compressed instruction set.

A sophisticated assembler will automatically generate compressed instructions whenever it can. The idea is that the programmer (or compiler) will create only 32 bit instructions. Upon encountering a 32 bit instruction that can also be coded as a compressed instruction, the assembler will choose the smaller instruction. Such an assembler will relieve programmers (and compilers) from the burden of selecting compressed instructions, although a sophisticated compiler may be able to generate shorter code sequences if it is aware of which instructions can be compressed.

At this time, only the full-sized instructions are defined. The compressed instructions will be defined in the future, based on which full-sized instructions are most widely used.

Opcode Encoding

The first 2 bits in every instruction determine whether or not it is a compressed instruction. All full-sized instructions begin with bits 00.

00 - Full-sized instruction
01 - Compressed instruction
10 - Compressed instruction
11 - Compressed instruction

From here on, we only discuss full-sized instructions.

The instruction opcode is either 1 or 2 bytes. The opcode is in either the first byte or the first two bytes of the instruction, i.e., the most significant byte or bytes.

The first byte of every instruction is called “OP1” and the second byte of the opcode, if present, is called “OP2”.

If the first byte (OP1) is 0x00, then a second opcode byte (OP2) will be used. If the first byte (OP1) is non-zero, then there will be no second byte.

Instruction Fields

We use the following notations to describe the various bit fields in an instruction.

Reg1	4 bits, indicating a source register
Reg2	4 bits, indicating a source register
Reg3	4 bits, indicating a source register
RegD	4 bits, indicating the destination register
immed3	3 bits containing an immediate value
immed6	6 bits containing an immediate value
immed10	10 bits containing an immediate value
immed16	16 bits containing an immediate value
immed20	20 bits containing an immediate value

The registers are encoded in the obvious way:

r0 = 0000
r1 = 0001
…
r15 = 1111

The **immed3** field is used in the CHECKADDR instruction and is interpreted as a code indicating which sort of check to perform.

The **immed6** field is used in the shifting instructions and is interpreted as a positive number, i.e., the number of bits to shift by.

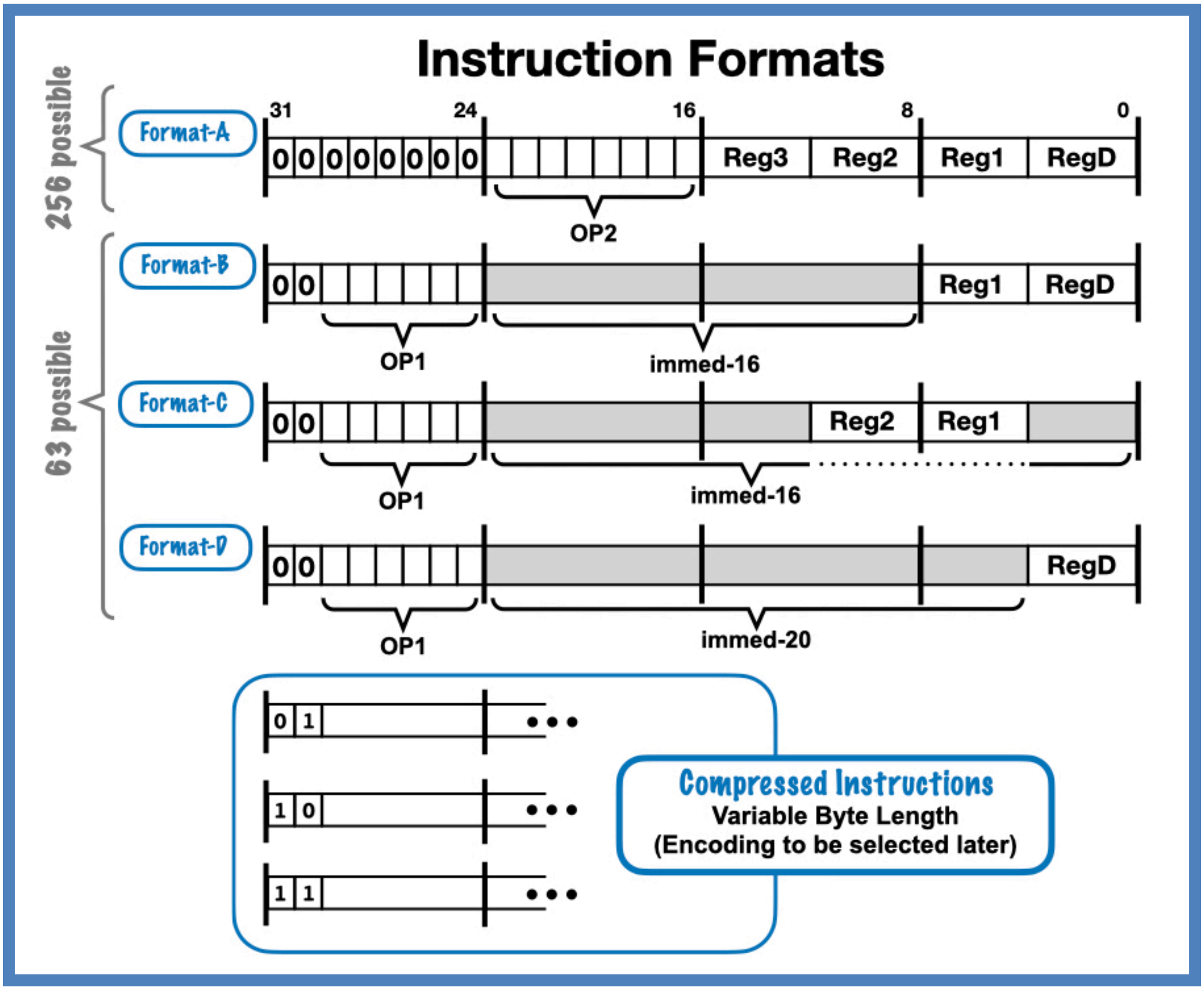
The **immed10** field is only used in the SYSCALL instruction and is interpreted as a positive number.

The **immed16** and **immed20** fields are signed-extended to 64 bits, unless explicitly noted otherwise.

	Smallest Value	**Largest Value**	**Number of Values**
reg	r0	r15	16 = 2^4
immed3	0	7	8 = 2^3
immed6	0	63	64 = 2^6
immed10	0	1,023	1,024 = 2^{10}
immed16	-32,768	+32,767	65,536 = 2^{16} = 64 Ki
immed20	-524,288	+524,287	1,048,576 = 2^{20} = 1 Mi

Instruction Formats

FIGURE: Instruction Formats



When giving the binary patterns for the various instruction formats below, we use the following notation to represent bit fields.

DDDD = RegD
1111 = Reg1
2222 = Reg2
3333 = Reg3
VVVVVVVV = Immediate value
XXXXXXXX = Op-code
00000000 = Zero bits

For some instructions, one or more of the fields may be unused.

Unused fields are ignored. The assembler should fill them with zeros, but they do not affect the core's execution.

For example, the ADD instruction is a Format-A instruction, which has room for 4 register operands. However the ADD instruction only uses 3 registers. The remaining field is unused for ADD.

The shorter immediate values (i.e., **immed3**, **immed6**, and **immed10**) are encoded as 16 bit values with the upper bits being unused and ignored.

Format-A instructions:

Operands:

RegD,Reg1,Reg2,Reg3

Binary Encoding:

```
0000 0000 XXXX XXXX 3333 2222 1111 DDDD
```

Examples:

```
SYSRET              # Return from trap handler
CHECKH  r4          # Ensure r4 is within 16 bits
SEXTW   r4,r6       # r4 ← SignExtend(r6)
ADD     r4,r6,r7    # r4 ← r6+r7
```

Format-B instructions:

Operands:

RegD,Reg1,immed16

Binary Encoding:

```
00XX XXXX VVVV VVVV VVVV VVVV 1111 DDDD
```

Examples:

```
ADDI    r4,r6,1234   # r4 ← r6+1234
LOAD.B  r6,1234(r4)  # r6 ← Mem[1234+r4]
```

Format-C instructions:

Operands:

Reg1,Reg2,immed16

Binary Encoding:

```
00XX XXXX VVVV VVVV VVVV 2222 1111 VVVV
```

Examples:

```
B.LT    r4,r6,loop   # if r4<r6, goto offset(pc)
STORE.B 1234(r4),r6  # Mem[1234+r4] ← r6
```

Format-D instructions:

Operands:

RegD,immed20

Binary Encoding:

```
00XX XXXX VVVV VVVV VVVV VVVV VVVV DDDD
```

Examples:

```
JAL     lr,MyFunc   # call: pc←offset+pc; lr←ret addr
UPPER20 r4,0x3A4B5  # r4 ← (0x3A4B5 << 16)
```

Operand Syntax

In assembly language, the instruction operands are specified in several different ways.

	General Form	Example	
Format-A			
A-0	OP *<no operands>*	sysret	
A-1	OP Reg1	checkb	r1
A-2	OP RegD,Reg1	sextb	r7,r1
A-3	OP RegD,Reg1,Reg2	add	r7,r1,r2
A-4	OP RegD,Reg1,Reg2,Reg3	muladd	r7,r1,r2,r3
A-5	*< No longer used >*		
A-6	*< No longer used >*		
A-7	OP RegD,CSRReg1,Reg2	csrswap	r7,csr1,r2
A-8	OP RegD,CSRReg1	csread	r7,csr1
A-9	OP RegD	getstat	r7
Format-B			
B-1	OP RegD,Reg1,immed16	addi	r7,r1,0x1234
B-2	OP RegD,immed16(Reg1)	load.b	r7,offset(r1)
B-3	OP RegD,Reg1,immed3	checkaddr	r7,r1,5
B-4	OP immed10	syscall	123
B-5	OP RegD,Reg1,immed6	slli	r7,r1,5
B-6	OP CSRReg1,immed16	csrset	csr_status,0x1234
Format-C			
C-1	OP immed16(Reg1),Reg2	store.b	offset(r1),r2
C-2	OP Reg1,Reg2,immed16	b.le	r1,r2,MyLabel
Format-D			
D-1	OP RegD,immed20	jal	lr,MyLabel

Notice that the destination is almost always the first (leftmost) operand. This is easy to remember since this order mimics the order of an assignment statement in a high-level programming language.

Typical assignment statement:

destination = *…expr…* ;

Blitz assembler:

`RegD,` *…other operands…*

For the branching instructions, the operand order mimics an "if" statement.

Typical "if" statement:

if (x <= y) then go to MyLabel

Blitz assembler:

```
B.LE      r1,r2,MyLabel
```

Chapter 5: Instructions

Don't leave the classroom of pain without
gathering wisdom from its instruction.
— Tim Hiller

Machine Instructions versus Synthetic Instructions

A **machine instruction** is implemented in hardware. Each machine instruction has a single numeric opcode and, in assembly code, the opcode is indicated with a symbolic name, such as "ADD" or "SLL".

Synthetic instructions are not implemented in hardware. Instead, each synthetic instruction is processed by the assembler and/or linker and translated into machine instructions.

Each synthetic instruction has a symbolic opcode, such as "LOADD" or "CALL", so the synthetic instructions may be difficult to distinguish when looking at an assembly code program.

Typically, each synthetic instruction is translated into a single machine instruction, but in some cases the translation will be 2, 3, or 4 machine instructions. The processor core does not see or execute synthetic instructions.

An Instruction Set Architecture (ISA) normally defines only machine instructions, because that is all that hardware designers need in a specification of what to implement. However, this document also includes descriptions of synthetic instructions, alongside the machine instructions, making an easy reference for programmers.

In the instruction listings, synthetic instructions are identified by marking them with an asterisk (*) prefixing the symbolic opcode, as in *LOADD or *CALL. This asterisk is only used in this documentation to make it easy to identify the synthetic instructions. The asterisk is not part of the assembly language.

All Instructions - Summary Listing

Arithmetic

ADD	RegD,Reg1,Reg2	
ADDI	RegD,Reg1,immed16	
ADDOK	RegD,Reg1,Reg2	RegD ← (Reg1+Reg2 overflows) ? 0 : 1
ADD3	RegD,Reg1,Reg2,Reg3	RegD ← Reg1+Reg2+Reg3 (unsigned)
SUB	RegD,Reg1,Reg2	
* MUL	RegD,Reg1,Reg2	
MULADD	RegD,Reg1,Reg2,Reg3	RegD ← (Reg1 × Reg2) + Reg3
MULADDU	RegD,Reg1,Reg2,Reg3	RegD ← (Reg1 × Reg2) + Reg3 (unsigned)
DIV	RegD,Reg1,Reg2	
REM	RegD,Reg1,Reg2	
* NEG	RegD,Reg1	
* ABS	RegD,Reg1	

Logical

AND	RegD,Reg1,Reg2	
ANDI	RegD,Reg1,immed16	
OR	RegD,Reg1,Reg2	
ORI	RegD,Reg1,immed16	
XOR	RegD,Reg1,Reg2	
XORI	RegD,Reg1,immed16	
* BITNOT	RegD,Reg1	RedD ← Bitwise NOT (Reg1)
* LOGNOT	RegD,Reg1	RegD ← (Reg1 = 0) ? 1 : 0

Move

* MOV	RegD,Reg1
* MOVI	RegD,immediate-64

Shift

SLL	RegD,Reg1,Reg2	Shift left logical
SLLI	RegD,Reg1,immed6	
SLA	RegD,Reg1,Reg2	Shift left arithmetic
SLAI	RegD,Reg1,immed6	
SRL	RegD,Reg1,Reg2	Shift right logical
SRLI	RegD,Reg1,immed6	
SRA	RegD,Reg1,Reg2	Shift right arithmetic
SRAI	RegD,Reg1,immed6	
ROTR	RegD,Reg1,Reg2	Rotate right (circular)
ROTRI	RegD,Reg1,immed6	

Sign Extension

SEXTB	RegD,Reg1	Sign extend byte to 64 bits
SEXTH	RegD,Reg1	Sign extend 16 bits to 64 bits
SEXTW	RegD,Reg1	Sign extend 32 bits to 64 bits

Range Checking

NULLTEST	Reg1	Trap if Reg1 is contains NULL
CHECKB	Reg1	Trap if Reg1 not within -128 … +127
CHECKH	Reg1	Trap if Reg1 not within -32768 … +32767
CHECKW	Reg1	Trap if Reg1 not within 32 bit range
INDEX0	RegD,Reg1,Reg2,Reg3	Reg1=arrayPtr, Reg2=header, Reg3=index
INDEX1	RegD,Reg1,Reg2,Reg3	. RegD ← Reg1 + 8 + (Reg3 * scale)
INDEX2	RegD,Reg1,Reg2,Reg3	. Reg2 = header = [ArrayMAX || ArrayCURR]
INDEX4	RegD,Reg1,Reg2,Reg3	. Trap if (Reg3 < 0) or (Reg3 ≥ ArrayCURR)
INDEX8	RegD,Reg1,Reg2,Reg3	. or (ArrayMAX = 0)
INDEX16	RegD,Reg1,Reg2,Reg3	.
INDEX24	RegD,Reg1,Reg2,Reg3	.
INDEX32	RegD,Reg1,Reg2,Reg3	.

Byte Reordering

ENDIANH	RegD,Reg1	Reorder bytes in all 4 halfwords
ENDIANW	RegD,Reg1	Reorder bytes in both words
ENDIAND	RegD,Reg1	Reorder bytes in a doubleword

Test and Set a Boolean

TESTEQ	RegD,Reg1,Reg2	RegD ← (Reg1 = Reg2) ? 1 : 0
TESTNE	RegD,Reg1,Reg2	RegD ← (Reg1 ≠ Reg2) ? 1 : 0
TESTLT	RegD,Reg1,Reg2	RegD ← (Reg1 < Reg2) ? 1 : 0
TESTLE	RegD,Reg1,Reg2	RegD ← (Reg1 ≤ Reg2) ? 1 : 0
* TESTGT	RegD,Reg1,Reg2	RegD ← (Reg1 > Reg2) ? 1 : 0
* TESTGE	RegD,Reg1,Reg2	RegD ← (Reg1 ≥ Reg2) ? 1 : 0

TESTEQI	RegD,Reg1,immed16	RegD ← (Reg1 = immed) ? 1 : 0
TESTNEI	RegD,Reg1,immed16	RegD ← (Reg1 ≠ immed) ? 1 : 0
TESTLTI	RegD,Reg1,immed16	RegD ← (Reg1 < immed) ? 1 : 0
TESTLEI	RegD,Reg1,immed16	RegD ← (Reg1 ≤ immed) ? 1 : 0
TESTGTI	RegD,Reg1,immed16	RegD ← (Reg1 > immed) ? 1 : 0
TESTGEI	RegD,Reg1,immed16	RegD ← (Reg1 ≥ immed) ? 1 : 0

* TESTEQZ	RegD,Reg1	RegD ← (Reg1 = 0) ? 1 : 0, i.e., if zero
* TESTNEZ	RegD,Reg1	RegD ← (Reg1 ≠ 0) ? 1 : 0, i.e., if non-zero
* TESTLTZ	RegD,Reg1	RegD ← (Reg1 < 0) ? 1 : 0, i.e., if negative
* TESTLEZ	RegD,Reg1	RegD ← (Reg1 ≤ 0) ? 1 : 0, i.e., if non-positive
* TESTGTZ	RegD,Reg1	RegD ← (Reg1 > 0) ? 1 : 0, i.e., if positive
* TESTGEZ	RegD,Reg1	RegD ← (Reg1 ≥ 0) ? 1 : 0, i.e., if non-negative

Branch - Limited Range

B.EQ	Reg1,Reg2,immed16	Branch if Reg1 = Reg2; Offset is PC-relative
B.NE	Reg1,Reg2,immed16	Branch if Reg1 ≠ Reg2; Offset is PC-relative
B.LT	Reg1,Reg2,immed16	Branch if Reg1 < Reg2; Offset is PC-relative
B.LE	Reg1,Reg2,immed16	Branch if Reg1 ≤ Reg2; Offset is PC-relative

Branch - General

* BEQ	Reg1,Reg2,address	Branch if Reg1 = Reg2
* BNE	Reg1,Reg2,address	Branch if Reg1 ≠ Reg2
* BLT	Reg1,Reg2,address	Branch if Reg1 < Reg2
* BLE	Reg1,Reg2,address	Branch if Reg1 ≤ Reg2
* BGT	Reg1,Reg2,address	Branch if Reg1 > Reg2
* BGE	Reg1,Reg2,address	Branch if Reg1 ≥ Reg2

* BEQI	Reg,value,address	Branch if Reg = immediate value
* BNEI	Reg,value,address	Branch if Reg ≠ immediate value
* BLTI	Reg,value,address	Branch if Reg < immediate value
* BLEI	Reg,value,address	Branch if Reg ≤ immediate value
* BGTI	Reg,value,address	Branch if Reg > immediate value
* BGEI	Reg,value,address	Branch if Reg ≥ immediate value
* BEQZ	Reg,address	Branch if Reg = 0
* BNEZ	Reg,address	Branch if Reg ≠ 0
* BLTZ	Reg,address	Branch if Reg < 0, i.e., if negative
* BLEZ	Reg,address	Branch if Reg ≤ 0, i.e., if not positive
* BGTZ	Reg,address	Branch if Reg > 0, i.e., if positive
* BGEZ	Reg,address	Branch if Reg ≥ 0, i.e., if not negative
* BFALSE	Reg,address	Branch if Reg = 0, i.e., if “false”
* BTRUE	Reg,address	Branch if Reg ≠ 0, i.e., if “true”

Larger Addresses

UPPER20	RegD,immed20	RegD ← (immed<<16)
UPPER16	RegD,Reg1,immed16	RegD ← (immed<<16) + Reg1
SHIFT16	RegD,Reg1,immed16	RegD ← (Reg1 + immed16) << 16
ADDPC	RegD,immed20	RegD ← immed + PC
AUIPC	RegD,immed20	RegD ← (immed<<16) + PC

Jumping - Limited Range

JAL	RegD,immed20	RegD ← return addr; Target ← PC+offset
JALR	RegD,immed16(Reg1)	RegD ← return addr; Target ← offset+Reg1

Call / Jump / Return - General

* CALL	address	Jump to any address; save return addr in “lr”
* CALLR	Reg1	Jump to address; save return addr in “lr”
* RET	<no operands>	Return value is in link register “lr”
* JUMP	address	Jump to any address
* JR	Reg1	Indirect jump, via register

Load - Limited Range

```
  LOAD.B        RegD,immed16(Reg1)     Sign extend 8 bits to 64 bits
  LOAD.H        RegD,immed16(Reg1)     Sign extend 16 bits to 64 bits
  LOAD.W        RegD,immed16(Reg1)     Sign extend 32 bits to 64 bits
  LOAD.D        RegD,immed16(Reg1)
```

Load - General

```
* LOADB         RegD,address
* LOADH         RegD,address
* LOADW         RegD,address
* LOADD         RegD,address

* LOADB         RegD,offset(Reg1)
* LOADH         RegD,offset(Reg1)
* LOADW         RegD,offset(Reg1)
* LOADD         RegD,offset(Reg1)
```

Store - Limited Range

```
  STORE.B       immed16(Reg1),Reg2      Ignore upper 56 bits
  STORE.H       immed16(Reg1),Reg2      Ignore upper 48 bits
  STORE.W       immed16(Reg1),Reg2      Ignore upper 32 bits
  STORE.D       immed16(Reg1),Reg2
```

Store - General

```
* STOREB        address,Reg2
* STOREH        address,Reg2
* STOREW        address,Reg2
* STORED        address,Reg2

* STOREB        offset(Reg1),Reg2
* STOREH        offset(Reg1),Reg2
* STOREW        offset(Reg1),Reg2
* STORED        offset(Reg1),Reg2
```

Support for Unaligned Loads and Stores

ALIGNH	RegD,Reg1,Reg2,Reg3
ALIGNW	RegD,Reg1,Reg2,Reg3
ALIGND	RegD,Reg1,Reg2,Reg3

INJECT1H	RegD,Reg1,Reg2,Reg3
INJECT2H	RegD,Reg1,Reg2,Reg3
INJECT1W	RegD,Reg1,Reg2,Reg3
INJECT2W	RegD,Reg1,Reg2,Reg3
INJECT1D	RegD,Reg1,Reg2,Reg3
INJECT2D	RegD,Reg1,Reg2,Reg3

Miscellaneous

SYSCALL	immed10	immed10 selects one of 1,024 syscalls
SYSRET	<no operands>	
* NOP	<no operands>	
ILLEGAL	<no operands>	
SLEEP1	<no operands>	Enter light sleep state
SLEEP2	<no operands>	Enter deep sleep state
RESTART	<no operands>	Same as Power-On-Reset
DEBUG	<no operands>	
BREAKPOINT	<no operands>	
CONTROL	RegD,Reg1,immed16	
CONTROLU	RegD,Reg1,immed16	
CAS	RegD,Reg1,Reg2,Reg3	Compare and Set: If *r1=r2 then *r1←r3
FENCE	<no operands>	
PUSHFRAME	RegD,Reg1,immed16	Save lr, push frame onto stack
POPRET	RegD,Reg1,immed16	Retrieve lr, pop frame, and return

CSR Manipulation

CSRSWAP	RegD,CSRReg1,Reg2	RegD ← CSR; CSR ← Reg2
CSRREAD	RegD,CSRReg1	Reg1 encodes CSR; RegD ← CSR
* CSRWRITE	CSRReg1,Reg2	Reg1 encodes CSR; CSR ← Reg2
CSRSET	CSRReg1,immed16	Set selected bits in CSR
CSRCLR	CSRReg1,immed16	Clear selected bits in CSR
GETSTAT	RegD	RegD ← CSR_STATUS & 0x0000…03f8
PUTSTAT	Reg1	CSR_STATUS [9:3] ← Reg1 [9:3]

Memory Management Unit

TLBCLEAR	<no operands>	Invalidate all TLBs for current ASID
TLBFLUSH	Reg1	Invalidate TLB for virtual address in Reg1
CHECKADDR	RegD,Reg1,immed3	Reg1 = virt addr; RegD ← except. code or 0

Floating Point

FADD	RegD,Reg1,Reg2	RegD ← Reg1 + Reg2
FSUB	RegD,Reg1,Reg2	RegD ← Reg1 - Reg2
FMUL	RegD,Reg1,Reg2	RegD ← Reg1 × Reg2
FDIV	RegD,Reg1,Reg2	RegD ← Reg1 / Reg2
FMIN	RegD,Reg1,Reg2	RegD ← MIN (Reg1, Reg2)
FMAX	RegD,Reg1,Reg2	RegD ← MAX (Reg1, Reg2)
FNEG	RegD,Reg1	RegD ← -Reg1
FABS	RegD,Reg1	RegD ← ABSOLUTE_VALUE (Reg1)
FSQRT	RegD,Reg1	RegD ← SQUARE_ROOT (Reg1)
FEQ	RegD,Reg1,Reg2	RegD ← (Reg1 = Reg2) ? 1 : 0 (float compare)
FLT	RegD,Reg1,Reg2	RegD ← (Reg1 < Reg2) ? 1 : 0 (float compare)
FLE	RegD,Reg1,Reg2	RegD ← (Reg1 ≤ Reg2) ? 1 : 0 (float compare)
* FGT	RegD,Reg1,Reg2	RegD ← (Reg1 > Reg2) ? 1 : 0 (float compare)
* FGE	RegD,Reg1,Reg2	RegD ← (Reg1 ≥ Reg2) ? 1 : 0 (float compare)
FCVTFI	RegD,Reg1	Convert: floating-point ← int
FCVTIF	RegD,Reg1	Convert: int ← floating-point
FMADD	RegD,Reg1,Reg2,Reg3	RegD ← (Reg1 × Reg2) + Reg3
FNMADD	RegD,Reg1,Reg2,Reg3	RegD ← (-(Reg1 × Reg2)) + Reg3
FMSUB	RegD,Reg1,Reg2,Reg3	RegD ← (Reg1 × Reg2) - Reg3
FNMSUB	RegD,Reg1,Reg2,Reg3	RegD ← (-(Reg1 × Reg2)) - Reg3

Machine Instructions, Grouped By Format

Here is a complete list of the Blitz-64 machine instruction set.

The headers give the format that assembly language programmers will use. These are followed by all the instructions that fit the pattern, with example operands and comments, to give a hint at what each instruction does.

Format A-0 *<no operands>*

ILLEGAL		Canonical form of illegal instruction
SYSRET		PC ← csr_prev; csr_status ← csr_stat2
SLEEP1		Enter light sleep state
SLEEP2		Enter deep sleep state
RESTART		Same as Power-On-Reset
DEBUG		
BREAKPOINT		
FENCE		
TLBCLEAR		Invalidate all TLBs for current ASID

Format A-1 *Reg1*

NULLTEST	r1	Trap if reg contains NULL
CHECKB	r1	Trap if reg not within -128 … +127
CHECKH	r1	Trap if reg not within -32768 … +32767
CHECKW	r1	Trap if reg not within 32 bit range
PUTSTAT	r1	CSR_STATUS [9:3] ← Reg1 [9:3]
TLBFLUSH	r1	Invalidate TLB for virtual address in Reg1

Format A-2 *RegD,Reg1*

ENDIANH	r7,r1	Reorder bytes: 76543210 → 67452301
ENDIANW	r7,r1	Reorder bytes: 76543210 → 45670123
ENDIAND	r7,r1	Reorder bytes: 76543210 → 01234567
SEXTB	r7,r1	Sign extend byte to 64 bits
SEXTH	r7,r1	Sign extend 16 bits to 64 bits
SEXTW	r7,r1	Sign extend 32 bits to 64 bits
FNEG	r7,r1	
FABS	r7,r1	
FSQRT	r7,r1	
FCVTFI	r7,r1	Convert: floating-point ← int
FCVTIF	r7,r1	Convert: int ← floating-point

Format A-3	*RegD,Reg1,Reg2*	
`ADD`	`r7,r1,r2`	
`ADDOK`	`r7,r1,r2`	
`SUB`	`r7,r1,r2`	
`DIV`	`r7,r1,r2`	
`REM`	`r7,r1,r2`	
`AND`	`r7,r1,r2`	
`OR`	`r7,r1,r2`	
`XOR`	`r7,r1,r2`	
`SLL`	`r7,r1,r2`	
`SLA`	`r7,r1,r2`	Shift-left-arithmetic; checks for overflow
`SRL`	`r7,r1,r2`	
`SRA`	`r7,r1,r2`	
`ROTR`	`r7,r1,r2`	Rotate right (circular)
`TESTEQ`	`r7,r1,r2`	RegD ← (Reg1 = Reg2) ? 1 : 0
`TESTNE`	`r7,r1,r2`	RegD ← (Reg1 ≠ Reg2) ? 1 : 0
`TESTLT`	`r7,r1,r2`	RegD ← (Reg1 < Reg2) ? 1 : 0
`TESTLE`	`r7,r1,r2`	RegD ← (Reg1 ≤ Reg2) ? 1 : 0
`FADD`	`r7,r1,r2`	
`FSUB`	`r7,r1,r2`	
`FMUL`	`r7,r1,r2`	
`FDIV`	`r7,r1,r2`	
`FMIN`	`r7,r1,r2`	
`FMAX`	`r7,r1,r2`	
`FEQ`	`r7,r1,r2`	RegD ← (Reg1 = Reg2) ? 1 : 0 (float compare)
`FLT`	`r7,r1,r2`	RegD ← (Reg1 < Reg2) ? 1 : 0 (float compare)
`FLE`	`r7,r1,r2`	RegD ← (Reg1 ≤ Reg2) ? 1 : 0 (float compare)

Format A-4	*RegD,Reg1,Reg2,Reg3*	
`ADD3`	`r7,r1,r2,r3`	Reg3 ← Reg1+Reg2+Reg3 (unsigned)
`MULADD`	`r7,r1,r2,r3`	RegD ← (Reg1 × Reg2) + Reg3
`MULADDU`	`r7,r1,r2,r3`	RegD ← (Reg1 × Reg2) + Reg3 (unsigned)
`INDEX0`	`r7,r1,r2,r3`	Reg1=arrayPtr, Reg2=header, Reg3=index
`INDEX1`	`r7,r1,r2,r3`	. RegD ← Reg1 + 8 + (Reg3 * scale)
`INDEX2`	`r7,r1,r2,r3`	. Reg2=header=[ArrayMAX||ArrayCURR]
`INDEX4`	`r7,r1,r2,r3`	. Trap if (Reg3 < 0) or (Reg3 ≥ ArrayCURR)
`INDEX8`	`r7,r1,r2,r3`	. or (ArrayMAX = 0)

`INDEX16`	`r7,r1,r2,r3`	.
`INDEX24`	`r7,r1,r2,r3`	.
`INDEX32`	`r7,r1,r2,r3`	.
`ALIGNH`	`r7,r1,r2,r3`	Reg3 (unaligned addr) gives shift amount
`ALIGNW`	`r7,r1,r2,r3`	Reg3 (unaligned addr) gives shift amount
`ALIGND`	`r7,r1,r2,r3`	Reg3 (unaligned addr) gives shift amount
`INJECT1H`	`r7,r1,r2,r3`	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT2H`	`r7,r1,r2,r3`	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT1W`	`r7,r1,r2,r3`	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT2W`	`r7,r1,r2,r3`	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT1D`	`r7,r1,r2,r3`	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT2D`	`r7,r1,r2,r3`	RegD ← Reg1; inject Reg2 per addr in Reg3
`CAS`	`r7,r1,r2,r3`	Compare and Set: If *r1=r2 then *r1←r3
`FMADD`	`r7,r1,r2,r3`	RegD ← (Reg1 × Reg2) + Reg3
`FNMADD`	`r7,r1,r2,r3`	RegD ← (-(Reg1 × Reg2)) + Reg3
`FMSUB`	`r7,r1,r2,r3`	RegD ← (Reg1 × Reg2) - Reg3
`FNMSUB`	`r7,r1,r2,r3`	RegD ← (-(Reg1 × Reg2)) - Reg3

Format A-5 *Reg1,Reg2*

< No longer used >

Format A-6 *Reg2*

< No longer used >

Format A-7 *RegD,Reg1,Reg2*

`CSRSWAP`	`r7,csr,r2`	Reg1 encodes CSR; RegD ← CSR; CSR ← Reg2

Format A-8 *RegD,Reg1*

`CSRREAD`	`r7,csr`	Reg1 encodes CSR; RegD ← CSR;

Format A-9 *RegD*

`GETSTAT`	`r7`	RegD ← CSR_STATUS & 0x0000…03f8

Format B-1	*RegD,Reg1,immed16*	
ADDI	r7,r1,0x1234	
ANDI	r7,r1,0x1234	
ORI	r7,r1,0x1234	
XORI	r7,r1,0x1234	
TESTEQI	r7,r1,0x1234	RegD ← (Reg1=immed) ? 1 : 0
TESTNEI	r7,r1,0x1234	RegD ← (Reg1≠immed) ? 1 : 0
TESTLTI	r7,r1,0x1234	RegD ← (Reg1<immed) ? 1 : 0
TESTLEI	r7,r1,0x1234	RegD ← (Reg1≤immed) ? 1 : 0
TESTGTI	r7,r1,0x1234	RegD ← (Reg1<immed) ? 1 : 0
TESTGEI	r7,r1,0x1234	RegD ← (Reg1≥ immed) ? 1 : 0
UPPER16	r7,r1,0x1234	RegD ← (immed<<16) + Reg1
SHIFT16	r7,r1,0x1234	RegD ← (Reg1+immed) << 16
CONTROL	r7,r1,0x1234	
CONTROLU	r7,r1,0x1234	
ENTERFUN	sp,sp,32	Push frame onto stack, save **lr** in frame
EXITFUN	sp,sp,32	Retrieve **lr**, pop frame, and return

Format B-2	*RegD,immed16(Reg1)*	
LOAD.B	r7,offset(r1)	Value is sign-extended to 64 bits
LOAD.H	r7,offset(r1)	. May cause unaligned exception
LOAD.W	r7,offset(r1)	. No overflow check on addr calculation
LOAD.D	r7,offset(r1)	
JALR	lr,offset(r1)	RegD ← return addr; Target ← offset+Reg1

Format B-3	*RegD,Reg1,immed3*	
CHECKADDR	r7,r1,5	Reg1 = virt addr; RegD ← except. code or 0

Format B-4	*immed10*	
SYSCALL	123	immed10 selects one of 1,024 syscalls

Format B-5	*RegD,Reg1,immed6*	
SLLI	r7,r1,5	
SLAI	r7,r1,5	Shift-left-arithmetic checks for overflow
SRLI	r7,r1,5	
SRAI	r7,r1,5	
ROTRI	r7,r1,5	Rotate right (circular)

Format B-6	*Reg1,immed16*	
`CSRSET`	`csr,0x1234`	Reg1 encodes CSR; Set selected bits in CSR
`CSRCLR`	`csr,0x1234`	Reg1 encodes CSR; Clear selected bits in CSR

Format C-1	*immed16(Reg1),Reg2*	
`STORE.B`	`offset(r1),r2`	Upper bits in reg are ignored
`STORE.H`	`offset(r1),r2`	. May cause unaligned exception
`STORE.W`	`offset(r1),r2`	. No overflow check on addr calculation
`STORE.D`	`offset(r1),r2`	

Format C-2	*Reg1,Reg2,immed16*	
`B.EQ`	`r1,r2,MyLabel`	Branch if Reg1=Reg2; Offset is PC-relative
`B.NE`	`r1,r2,MyLabel`	Branch if Reg1≠Reg2; Offset is PC-relative
`B.LT`	`r1,r2,MyLabel`	Branch if Reg1<Reg2; Offset is PC-relative
`B.LE`	`r1,r2,MyLabel`	Branch if Reg1≤Reg2; Offset is PC-relative

Format D-1	*RegD,immed20*	
`UPPER20`	`r7,MyLabel`	RegD ← (immed<<16)
`ADDPC`	`r7,MyLabel`	RegD ← immed+PC
`AUIPC`	`r7,MyLabel`	RegD ← (immed<<16) + PC
`JAL`	`lr,MyLabel`	RegD ← return addr ; Target ← PC+immed

The Instruction Set

Next, we list the Blitz-64 instructions, including both machine instructions and synthetic instructions. In this document, synthetic instructions are identified with "*".

```
ADD             RegD,Reg1,Reg2
ADDI            RegD,Reg1,immed16
SUB             RegD,Reg1,Reg2
*MUL            RegD,Reg1,Reg2
DIV             RegD,Reg1,Reg2
REM             RegD,Reg1,Reg2
AND             RegD,Reg1,Reg2
ANDI            RegD,Reg1,immed16
OR              RegD,Reg1,Reg2
ORI             RegD,Reg1,immed16
XOR             RegD,Reg1,Reg2
XORI            RegD,Reg1,immed16
```

May cause an "Arithmetic Exception", "Stack Overflow Exception"

All computations are performed using 64 bit values and the arithmetic instructions are performed with signed, two's complement arithmetic.

The operands are either in Reg1 and Reg2, or in Reg1 and an immediate value embedded in the instruction. The result is placed into RegD.

For the immediate-form instructions, the 16 bit immediate value is signed extended to 64 bits. Thus, any value within the range -32,768 … 32,767 may be used.

It is the assembly programmer's or compiler's responsibility to ensure that the immediate value is within range. If the value is out of range, the assembler will issue an error message. If necessary, the programmer can always use a MOVI instruction to move a larger value into a temp register. (Register "t" is generally used for things like this.)

Overflow is always checked. The goal is to catch all program bugs and failures, and not continue computing with incorrect values, as happens in other systems.

The following instructions will never cause an exception:
AND, ANDI, OR, ORI, XOR, XORI

The following instructions will cause an “Arithmetic Exception” whenever the mathematically correct result is not representable.
ADD, ADDI, SUB, MUL

The following instructions will cause an “Arithmetic Exception” in the case of divide-by-zero or attempt to evaluate MIN_64 / -1:
DIV, REM

The following instructions are candidates for emulation. Any attempt to execute an unimplemented instruction will result in an “Emulation Exception”.
DIV, REM

The MUL instruction is synthetic and is shorthand for:
MULADD RegD,Reg1,Reg2,r0

In the case of ADD and ADDI, if the **sp** register is modified and the new value is less than the **StackLimit** field in **csr_status**, a “Stack Overflow Exception” will occur.

`MULADD`	*RegD,Reg1,Reg2,Reg3*	RegD ← (Reg1 × Reg2) + Reg3
`MULADDU`	*RegD,Reg1,Reg2,Reg3*	RegD ← (Reg1 × Reg2) + Reg3 (unsigned)

MULADD may cause an “Arithmetic Exception”; MULADDU causes no exceptions

These instructions multiply the contents of Reg1 and Reg2, then add the contents of Reg3, and finally place the result in RegD.

In the case of MULADD, the arguments and the result are treated as 64 bit signed integers. If overflow occurs on either the multiplication or addition, it will cause an Arithmetic Exception.

In the case of MULADDU, the arguments and the result are treated as 64 bit unsigned integers. Overflow is ignored and no exception will be raised.

Note that both instructions will produce the same 64-bit result, unless of course the MULADD causes an exception, in which case it fails to produce any result at all.

If Reg1 is r0, then these instructions can be used to perform a simple multiply; signed in the case of MULADD and unsigned in the case of MULADDU.

The MULADD instruction is used to implement the synthetic MUL instruction. The MULADDU instruction is useful for accessing arrays.

Integer Division With Negative Operands

Consider dividing *a* by *n* (that is, *a/n*).

$q \leftarrow a \text{ DIV } n$ # compute quotient
$r \leftarrow a \text{ REM } n$ # compute remainder

The resulting quotient (*q*) and remainder (*r*) must obey these equations:

$a = nq + r$
$|r| < |q|$

With positive operands, this specification is unambiguous. However, there always remains a question about how negative operands are treated. There are several competing definitions which meet the basic division definition given above.

Many languages (C, C++, Java) perform "**truncated division**":

$q \leftarrow trunc(a/n)$
$r \leftarrow a - n\ trunc(a/n)$

which produces these results:

```
 7 DIV  3 =  2          7 REM  3 =  1
-7 DIV  3 = -2         -7 REM  3 = -1
 7 DIV -3 = -2          7 REM -3 =  1
-7 DIV -3 =  2         -7 REM -3 = -1
```

A second reasonable definition is called "**floored division**":

$q = \lfloor a/n \rfloor$
$r = a - n \lfloor a/n \rfloor$

which produces the following results. The dot (•) indicates differences with truncated division.

```
 7 DIV  3 =  2          7 REM  3 =  1
-7 DIV  3 = -3 •       -7 REM  3 =  2 •
 7 DIV -3 = -3 •        7 REM -3 = -2 •
-7 DIV -3 =  2         -7 REM -3 = -1
```

There is also a third definition called “**Euclidean division**”, in which the remainder is never negative. The dot (•) indicates differences with both previous definitions.

```
 7 DIV  3 =  2          7 REM  3 =  1
-7 DIV  3 = -3         -7 REM  3 =  2      same as “floored”
 7 DIV -3 = -2          7 REM -3 =  1      same as “truncated”
-7 DIV -3 =  3 •       -7 REM -3 =  2 •    different from both
```

Which definition is better? The following quote from Wikipedia is pertinent:

> “… Euclidean division is superior to the other ones in terms of regularity and useful mathematical properties, although floored division … is also a good definition. Despite its widespread use, truncated division is shown to be inferior to the other definitions.”
>
> — Daan Leijen, *Division and Modulus for Computer Scientists*

The Blitz-64 spec leaves this decision open as “implementation dependent”.

We chose to name the instruction “REM” and not “MOD” because “MOD” is assumed to mean “Euclidean division”, but this may not be the what the implementation actually performs.

Note that “truncated” and “Euclidean” have identical results as long as the number on top (the “dividend”, which is defined as *a* in the operation a/n) is positive.

Division by a power of 2 (i.e., when the divisor is 1,2,4,8,16,…) is sometimes implemented as a right shift operation. For example, dividing by 4 is implemented with a right shift of two bits. Shifting always works correctly if the dividend is positive. For example, 21 DIV 4 = 21 >> 2 = **10101** >> 2 = **101** = 5.

But note: If the dividend is negative, shifting may not be equivalent to the DIV operation. The result of shifting is always the same as “floored” and “Euclidean” division. However *shifting is not equivalent to "truncated" division*. Truncated division is used in "C" and may be the choice for some Blitz-64 implementations, so care must be taken if the dividend can be negative.

For example with truncated division:
-21 DIV 4 = -5 with reminder -1.
In binary:
-21 >> 2 = ...**11101011** >> 2 = ...**11111010** = -6

<u>Division Overflow Conditions</u>

An attempt to divide by zero will cause an Arithmetic Exception. But there is another possibility for overflow.

Concerning the sizes of the result, note that the following must hold, since $|n| \geq 1$:

$|q| \leq |a|$
$|r| < |a|$

Thus, if the operands (a and n) are 64 bits, then the results (q and r) will almost always fit into 64 bits.

There is exactly one exception in which the result will not fit.

Let MIN_64 represent -2^{63}, which is the most negative number representable in 64 bits. If we divide MIN_64 by -1, the result is $+2^{63}$, which is one greater than the largest positive 64 bit number.

Note that "division overflow" can only occur with negative operands; there is no need to worry if $n>0$ is guaranteed to hold when computing a/n.

This computation will cause an Arithmetic Exception.

<u>Bottom Line</u> Programmers computing a/n should take special care unless the following are certain to hold:

$a \geq 0$
$n > 0$

`*NEG` *RegD,Reg1*

Synthetic

May cause an “Arithmetic Exception”

Register t Usage: Not used; Okay to use as RegD and/or Reg1.

Treating the value as a signed number, this instruction will flip the sign. This instruction is implemented as:

```
    SUB         RegD,r0,Reg1
```

An “Arithmetic Exception” will be signaled for an attempt to negate the most negative number.

`*BITNOT` *RegD,Reg1*

Synthetic

Register t Usage: Not used.

All 64 bits are flipped. This instruction is implemented as:

```
    XORI        RegD,Reg1,-1
```

`*NOP` *<no operands>*

Synthetic

Register t Usage: Not used.

This is a no-op. This instruction is implemented as:

```
    ADDI        r0,r0,0
```

*ABS *RegD,Reg1*

Synthetic

Register t Usage: Not used.

This instruction computes the absolute value. This instruction is implemented as:

```
    MOV        RegD,Reg1
    BGEZ       Reg1,Label
    SUB        RegD,r0,Reg1
  Label:
```

An “Arithmetic Exception” will be signaled for an attempt to compute the absolute value of the most negative number.

*MOV *RegD,Reg1*

Synthetic

Register t Usage: Not used.

This instruction is implemented as:

```
    ORI        RegD,Reg1,0
```

Since OR-ing with a constant of 0 is not commonly done, it is reasonable for a disassembler to render this instruction as MOV.

This instruction may also be implemented as any of these.

```
    ADD        RegD,Reg1,r0
    ADDI       RegD,Reg1,0
    OR         RegD,Reg1,r0
    ORI        RegD,Reg1,0        This instruction is preferred.
    XOR        RegD,Reg1,r0
    XORI       RegD,Reg1,0
```

*MOVI *RegD,immediate*

Synthetic, Variable Length

Register t Usage: Not used; Okay to use as RegD.

An immediate 64 bit value is moved into register RegD.

The implementation of this instruction depends on the value of the immediate operand.

If the value is within 16 bits (i.e., within -32,768 … 32,767):

```
    XORI        RegD,r0,immed16
```

If the value is an address near the MOVI instruction itself, i.e., within 20 bits (-524,288 … +524,287) of the current PC:

```
    ADDPC       RegD,immed20
```

If the value is within 36 bits (e.g., any valid address, within -32Gi … +32Gi-1):

```
    UPPER20     RegD,immed20
    XORI        RegD,RegD,immed16
```

If the value is an address and PC-relative instructions are required:

```
    AUIPC       RegD,immed20
    XORI        RegD,RegD,immed16
```

If the value is within 52 bits:

```
    UPPER20     RegD,immed20
    SHIFT16     RegD,RegD,immed16
    XORI        RegD,RegD,immed16
```

Otherwise, to load an arbitrary 64 bit value:

```
    UPPER16     RegD,r0,immed16
    SHIFT16     RegD,RegD,immed16
    SHIFT16     RegD,RegD,immed16
    XORI        RegD,RegD,immed16
```

Comment: If the immediate value in the XORI instruction is negative, all the upper 48 bits will be 1’s. This will flip any bits previously loaded into the upper 48 bits.

Therefore, the assembler will need to compensate by flipping all the bits in the immediate values used in the UPPER20, UPPER16, and SHIFT16 instructions.

Since these instruction sequences are idiosyncratic and not likely to occur elsewhere, it is reasonable for a disassembler to render them as a MOVI instruction.

`SLL`	*RegD,Reg1,Reg2*	Shift left logical
`SLLI`	*RegD,Reg1,immed6*	
`SLA`	*RegD,Reg1,Reg2*	Shift left arithmetic
`SLAI`	*RegD,Reg1,immed6*	
`SRL`	*RegD,Reg1,Reg2*	Shift right logical
`SRLI`	*RegD,Reg1,immed6*	
`SRA`	*RegD,Reg1,Reg2*	Shift right arithmetic
`SRAI`	*RegD,Reg1,immed6*	
`ROTR`	*RegD,Reg1,Reg2*	Rotate right (circular)
`ROTRI`	*RegD,Reg1,immed6*	

May cause an “Arithmetic Exception”

The 64 bit value in Reg1 is shifted/rotated and the result is placed in RegD. The shift/rotate amount is specified in either Reg2 or as an immediate value.

The logical shifts (SLL, SLLI, SRL, SRLI) will shift 0 bits in, and will discard the bits shifted out.

The Shift Right Arithmetic instructions (SRA, SRAI) are conventional. The sign-bit is duplicated as necessary and shifted in on the most significant (left) end.

However, Blitz-64 also includes Shift Left Arithmetic instructions (SLA, SLAI) which are somewhat unusual. In the case of Shift Left Logical (SLL, SLLI), there is no overflow check; bits are simply shifted out the most-significant end with no consequences. However, in the case of Shift Left Arithmetic (SLA, SLAI), there is an overflow check. For SLA and SLAI, if the bits shifted out do not all agree with the final sign bit, then an “Arithmetic Exception” is signaled. This makes these instructions usable as a way to multiply by a power of 2, which is required to cause an Arithmetic Exception in the case of overflow.

The shift amounts should be between 0 and 63. If an immediate value is provided in SLLI, SLAI, SRLI, and SRAI, only the last 6 bits are examined. The upper bits are ignored and the immed6 value is treated as an unsigned value within the range 0 … 63. The value of 0 results in no shifting and is effectively a "nop".[3]

The following instructions will cause an "Arithmetic Exception" whenever the shift amount (i.e., the value in Reg2) is not within 0 … 63.

SLL, SLA, SRL, SRA

Except as mentioned above, the other instructions never cause exceptions.

For bit rotations (ROTR and ROTRI), if the shift value *N* is larger than 63, it is equivalent in meaning to a rotation of *N* mod 64. For ROTR and ROTRI, only the least significant 6 bits of the shift amount are used; the upper bits are ignored.

Note that there are no rotate-left instructions, since a rotate-left-by-*N* is equivalent to a rotate-right with a shift amount of 64-*N*. Thus, we can use negative numbers to achieve a left-rotation. This works because the rotate instructions ignore all but the least significant 6 bits.

For example, imagine that you wish to rotate left by 5. The number -5 is 0xFFFF_FFFF_FFFF_FFFB. But the ROTR and ROTRI only use the least significant 6 bits. In this example, the least significant 6 bits are 111011, which is interpreted as +59. Rotating right by 59 achieves the same result as rotating left by 5.

Thus, for ROTR, if register *Reg1* contains a negative shift amount of -*N*, the effect will be to rotate to the left by *N* bits.

`SEXTB`	*RegD,Reg1*	Sign extend byte to 64 bits
`SEXTH`	*RegD,Reg1*	Sign extend 16 bits to 64 bits
`SEXTW`	*RegD,Reg1*	Sign extend 32 bits to 64 bits

These instructions sign-extend an 8, 16, or 32 bit value to 64 bits.

For SEXTB, the upper 56 bits [63:8] are all set to the value of bit [7]. Likewise, for SEXTH, the upper 48 bits [63:16] are all set to the value of bit [15]. For SEXTW, bits [63:32] are set to the value of bit [31].

[3] This range restriction is normally enforced by the assembler, so it is never as issue.

`NULLTEST`	*Reg1*	Trap if reg contains NULL

May cause an "Null Address Exception"

This instruction checks to see whether the address in the register is null and signals a "Null Address Exception" if so. More specifically, it signals an exception if and only if bits [35…3] are zero.

Recall that the upper 28 bits, i.e., bits [63…36] of a doubleword are ignored when the value is used as an address. Also, the entire doubleword at address 0 is inaccessible, so the least significant bits are ignored.

`CHECKB`	*Reg1*	Trap if reg not within -128 … +127
`CHECKH`	*Reg1*	Trap if reg not within -32768 … +32767
`CHECKW`	*Reg1*	Trap if reg not within 32 bit range

May cause an "Arithmetic Exception"

These instructions look at the 64 bit signed integer stored in a register and test it. If the value is out of range, an "Arithmetic Exception" will be signaled.

CHECKB will ensure that the value is within the range representable as a signed byte, namely within -128 … 127.

CHECKH will ensure that the value is within the range representable as a signed halfword, namely within -32,768 … 32,767.

CHECKW will ensure that the value is within the range representable as a signed word, namely within -2,147,483,648 … 2,147,483,647.

`ENDIANH`	*RegD,Reg1*	Reorder bytes in all 4 halfwords
`ENDIANW`	*RegD,Reg1*	Reorder bytes in both words
`ENDIAND`	*RegD,Reg1*	Reorder bytes in a doubleword

These instructions are used for transforming data between "big endian" and "little endian" byte ordering.

ENDIANH will swap the bytes in all halfwords in the register:

```
        0x7766_5544_3322_1100  →  0x6677_4455_2233_0011
```

ENDIANW will swap the bytes in both words in the register:

```
        0x7766_5544_3322_1100  →  4455_6677_0011_2233
```

ENDIAND will swap the bytes in a doubleword:

```
        0x7766_5544_3322_1100  →  0x0011_2233_4455_6677
```

Note that ENDIANW can be used to swap the byte order in a word, but the sign bits may not follow. For example, assume that the following 32 bit value from memory is assumed to be stored in little endian order. Note that, as a signed valued, this number is negative.

```
        55 66 77 88
```

We would like to store the correct value in a register. First we load it, using LOADW, giving:

```
        0x 0000_0000_5566_7788
```

Then we execute the ENDIANW instruction, to get:

```
        0x 0000_0000_8877_6655
```

Finally, we must execute the SEXTW instruction to sign extend it, giving:

```
        0x FFFF_FFFF_8877_6655
```

However, if we only need to store the 32 bit word back to memory, the SEXTW is unnecessary, since STOREW ignores the upper 32 bits in the register.

The same issue applies to reversing the byte order of halfwords.

TESTEQ	*RegD,Reg1,Reg2*	RegD ← (Reg1 = Reg2) ? 1 : 0
TESTNE	*RegD,Reg1,Reg2*	RegD ← (Reg1 ≠ Reg2) ? 1 : 0
TESTLT	*RegD,Reg1,Reg2*	RegD ← (Reg1 < Reg2) ? 1 : 0
TESTLE	*RegD,Reg1,Reg2*	RegD ← (Reg1 ≤ Reg2) ? 1 : 0
TESTEQI	*RegD,Reg1,immed16*	RegD ← (Reg1 = immed) ? 1 : 0
TESTNEI	*RegD,Reg1,immed16*	RegD ← (Reg1 ≠ immed) ? 1 : 0
TESTLTI	*RegD,Reg1,immed16*	RegD ← (Reg1 < immed) ? 1 : 0
TESTLEI	*RegD,Reg1,immed16*	RegD ← (Reg1 ≤ immed) ? 1 : 0
TESTGTI	*RegD,Reg1,immed16*	RegD ← (Reg1 > immed) ? 1 : 0
TESTGEI	*RegD,Reg1,immed16*	RegD ← (Reg1 ≥ immed) ? 1 : 0

These instructions compare two values using signed 64 bit arithmetic. The result, a boolean value, is placed in register RegD as either 1 (true) or 0 (false).

For the immediate values, the 16 bit immediate is sign-extended to 64 bits.

It is the assembly programmer's or compiler's responsibility to ensure that the immediate value is within range. If the value is out of range, the assembler will issue an error message. The programmer is always free to use a MOVI instruction using the temporary "t" register if necessary, to deal with a larger immediate value.

*TESTGT	*RegD,Reg1,Reg2*	RegD ← (Reg1 > Reg2) ? 1 : 0
*TESTGE	*RegD,Reg1,Reg2*	RegD ← (Reg1 ≥ Reg2) ? 1 : 0

Synthetic

Register t Usage: Not used; Okay to use as RegD, Reg1 and/or Reg2.

The TESGT instruction is implemented as:

TESTLT *RegD,Reg2,Reg1* Note that Reg1 and Reg2 are reversed

The TESTGE instruction is implemented as:

TESTLE *RegD,Reg2,Reg1* Note that Reg1 and Reg2 are reversed

`*TESTEQZ`	*RegD,Reg1*	RegD ← (Reg1 = 0) ? 1 : 0, i.e., if zero
`*TESTNEZ`	*RegD,Reg1*	RegD ← (Reg1 ≠ 0) ? 1 : 0, i.e., if non-zero
`*TESTLTZ`	*RegD,Reg1*	RegD ← (Reg1 < 0) ? 1 : 0, i.e., if negative
`*TESTLEZ`	*RegD,Reg1*	RegD ← (Reg1 ≤ 0) ? 1 : 0, i.e., if non-positive
`*TESTGTZ`	*RegD,Reg1*	RegD ← (Reg1 > 0) ? 1 : 0, i.e., if positive
`*TESTGEZ`	*RegD,Reg1*	RegD ← (Reg1 ≥ 0) ? 1 : 0, i.e., if non-negative

Synthetic

Register t Usage: Not used; Okay to use as RegD and/or Reg1.

The value in a register is compared with zero and a boolean result (either 0 or 1) is placed in register RegD.

The TESTEQZ instruction is implemented as:

```
   TESTEQ     RegD,Reg1,r0
```

The TESTNEZ instruction is implemented as:

```
   TESTNE     RegD,Reg1,r0
```

The TESTLTZ instruction is implemented as:

```
   TESTLT     RegD,Reg1,r0
```

TheTESTLEZ instruction is implemented as:

```
   TESTLE     RegD,Reg1,r0
```

The TESTGTZ instruction is implemented as:

```
   TESTLT     RegD,r0,Reg1          Note that the registers are reversed
```

The TESTGEZ instruction is implemented as:

```
   TESTLE     RegD,r0,Reg1          Note that the registers are reversed
```

`*LOGNOT`	*RegD,Reg1*	RegD ← (Reg1 = 0) ? 1 : 0

Synthetic

Register t Usage: Not used; Okay to use as RegD and/or Reg1.

The convention is to interpret 0 as “false” and any non-zero value as “true”, with 1 being the desired, canonical value for “true”. The LOGNOT instruction performs a logical “not”. For input 0, it computes 1. For any other input, it computes 0.

The LOGNOT instruction is implemented as:

`TESTEQ` *RegD,r0,Reg1* RegD ← (zero=Reg1) ? 1 : 0

Note that the synthetic instruction:

`*TESTEQZ` *RegD,Reg1*

is implemented as:

`TESTEQ` *RegD,Reg1,r0* RegD ← (Reg1=zero) ? 1 : 0

which is slightly different. This allows a disassembler to differentiate them.

`ADDOK` *RegD,Reg1,Reg2* RegD ← (Reg1+Reg2 overflows) ? 0 : 1

This instruction adds the contents of Reg1 and Reg2 using 64 bit signed arithmetic. If the addition results in overflow, then RegD is set to 0. Otherwise, if the addition proceeds without overflow, RegD is set to 1. The sum is discarded. No exception will be raised.

`ADD3` *RegD,Reg1,Reg2,Reg3* RegD ← Reg1+Reg2+Reg3 (unsigned)

This instruction adds the contents of Reg1, Reg2, and Reg3 using 64 bit arithmetic, placing the result in RegD. Overflow is ignored and no exceptions will be raised.

Commentary Obviously, the ADD3 instruction can be used to add only two unsigned values by using register r0 as the Reg3 argument.

The ADD3 instruction can also be used to subtract two unsigned numbers. Recall that to arithmetically negate a number, we flip the bits and add 1. Thus, the following code sequence can be used to compute "r1 ← r1-r2" while ignoring overflow.

```
BITNOT    r2,r2
MOVI      t,1
ADD3      r1,r1,r2,t
```

The result is identical and this works properly regardless of whether the numbers are viewed as signed or unsigned values.

Unsigned subtraction is not expected to be used much, so we accept the extra instruction overhead. Note that if we have to do a large number of subtractions, the overhead is only one extra instruction (the BITNOT), since we assume that the +1 can be preloaded into a register so the MOVI instruction is not repeated.

Note that with a "carry save adder" the microarchitecture gate delay time is minimal. With a carry save adder, the gate delay for adding three numbers is substantially less than twice the gate delay for adding two numbers.

`INDEX0` *RegD,Reg1,Reg2,Reg3*
`INDEX1` *RegD,Reg1,Reg2,Reg3*
`INDEX2` *RegD,Reg1,Reg2,Reg3*
`INDEX4` *RegD,Reg1,Reg2,Reg3*
`INDEX8` *RegD,Reg1,Reg2,Reg3*
`INDEX16` *RegD,Reg1,Reg2,Reg3*
`INDEX24` *RegD,Reg1,Reg2,Reg3*
`INDEX32` *RegD,Reg1,Reg2,Reg3*

May cause an "Bad Array Index Exception".

This instruction is designed to facilitate array accessing.

To understand these instructions, assume that Reg1 contains a pointer to the array, Reg2 contains the array header, and Reg3 contains the desired array index. There are eight INDEX instructions and each specifies a "scale", which can be 0, 1, 2, 4, 8, 16, 24, or 32. The scale is the size of the array elements, in bytes.

The instruction computes:
    RegD ← Reg1 + 8 + (Reg3 × scale)
If rewritten as follows, we see the address of the desired element is computed:
    RegD ← arrayPtr + 8 + (index × scale)

This computation is performed with unsigned arithmetic and overflow is ignored.

In the KPL programming language every array begins with an 8 byte header, which consists of the MAX array size (bits [63:32]) and the CURRENT size (bits [31:0]). The MAX and CURRENT are unsigned values in the range 0 … 4,294,967,295. We assume that the array header has been preloaded into Reg2.

Each INDEX instruction also performs two tests. The first test is that the index is legal. If ((Reg3 < 0) || (Reg3 ≥ CURRENT)), the instruction causes an "Bad Array

Index Exception". The second test is that the array is initialized. If (MAX = 0), the instruction causes an "Bad Array Index Exception".

The CURRENT size of an array should always be ≤ the MAX array size, but these instructions do not check for that.

Commentary Many programs use arrays and access the elements a lot. An example from KPL is:

```
    myArr [n+1] = myArr [i]
```

Such accesses are prone to program bugs. In the spirit of Blitz-64, the fullest possible error-checking is desired and this sort of check must be performed since the consequences of a program bug can be catastrophic. The purpose of the INDEX__ instructions is to reduce the overhead of this checking.

While these instructions are not limited to checking array index values, it is presumed that the software will respond to the exception with a message such as "Array index out of bounds", which could be confusing if the instruction is being used for another purpose.

Commentary Assume that "myArr" is an array of objects and consider the following KPL statement:

... = myArr [i] . someField

To compile code for this expression, of course instructions to get the address of array "myArr" and the value of the index expression "i" are needed. Let's look at the code after that.

Assume that each array element is 24 bytes in size and that "someField"—the field we are interested in—is a halfword at offset 18. Then the following instructions suffice:

```
    r1 ← ... address of myArr ...
    r3 ← ... index expression ...
    LOADD      r2,0(r1)          Fetch array header
    INDEX24    r7,r1,r2,r3       Compute address & check for errors
    LOADH      ...,18(r7)        Fetch the halfword at offset 18
```

If the access is in a loop, then a clever compiler might be able to pull some of these instructions out of the loop body, resulting in this code:

```
    r1 ← … address of myArr …
    LOADD      r2,0(r1)         Fetch array header
  LOOP:
    …
    r3 ← … index expression …
    INDEX24    r7,r1,r2,r3      Compute address & check for errors
    LOADH      …,18(r7)         Fetch the halfword at offset 18
    …
    JUMP       LOOP
```

In any other RISC computer, it is unlikely that a LOAD instruction would also be able to multiply to perform the scaling, so at least one additional instruction would probably be required, resulting in at least two instructions within the loop. So the overhead of Blitz-64 to provide the bounds checking appears to be *zero instructions*, at least in this example!

If the size of the array elements is not 1, 2, 4, 8 16, 24, or 32, then the INDEX0 instruction can be used in conjunction with the MULADDU instruction. For example, assume the element size is 80:

```
    r1 ← … address of myArr …
    r3 ← … index expression …
    LOADD      r2,0(r1)         Fetch array header
    INDEX0     r7,r1,r2,r3      Check for errors, advance to element 0
    MOVI       r4,80            Size of elements is 80 bytes
    MULADDU    r7,r3,r4,r7      r7 = r7 + (index × scale)
    LOADH      …,18(r7)         Fetch the halfword at offset 18
```

As before, if the access is within a loop, the compiler might pull the loop-invariant instructions out of the loop, yielding:

```
    r1 ← … address of myArr …
    LOADD     r2,0(r1)          Fetch array header
    MOVI      r4,80             Size of elements is 80 bytes
  LOOP:
    …
    r3 ← … index expression …
    INDEX0    r7,r1,r2,r3       Check for errors, advance to element 0
    MULADDU   r7,r3,r4,r7       r7 = r7 + (index × scale)
    LOADH     …,18(r7)          Fetch the halfword at offset 18
    …
    JUMP      LOOP
```

B.EQ	*Reg1,Reg2,immed16*	Branch if Reg1 = Reg2; Offset is PC-relative
B.NE	*Reg1,Reg2,immed16*	Branch if Reg1 ≠ Reg2; Offset is PC-relative
B.LT	*Reg1,Reg2,immed16*	Branch if Reg1 < Reg2; Offset is PC-relative
B.LE	*Reg1,Reg2,immed16*	Branch if Reg1 ≤ Reg2; Offset is PC-relative

May cause a "Null Address Exception"

The values in Reg1 and Reg2 are compared. In the case of LT (less than) and LE (less than or equal), the operand values are treated as signed integers.

If the condition is satisfied, a branch is taken.

To compute the target destination address, the 16 bit immediate value is sign-extended to 64 bits and then added to the value of the PC (i.e., the address of the BRANCH instruction itself).

Overflow is ignored. The upper bits [63:36] of the target address are ignored. The LSBit is set to 0, forcing halfword alignment.

Any attempt to load the PC with zero will cause a "Null Address Exception". Exceptions will only occur if the jump is taken; if the jump is not taken, no exception will occur.

In systems without compressed instructions and in which alignment is required for instructions, there may be an "Unaligned LOAD/STORE Exception" if the target address is not properly aligned. Alternatively, such systems may simply ignore the final bits, rounding the target address down to force alignment.

Commentary For the following instructions, we make a distinction between the opcode names for machine instructions and synthetic instructions, even though their functions are similar.

Machine Instruction	**Synthetic Instruction**
`B.EQ`	`BEQ`
`B.NE`	`BNE`
`B.LT`	`BLT`
`B.LE`	`BLE`
`LOAD.B`	`LOADB`
`LOAD.H`	`LOADH`
`LOAD.W`	`LOADW`
`LOAD.D`	`LOADD`
`STORE.B`	`STOREB`
`STORE.H`	`STOREH`
`STORE.W`	`STOREW`
`STORE.D`	`STORED`

A machine instruction is always a single 32 bit instruction, implemented directly in hardware.

A synthetic instruction may be implemented with 1, 2, or 3 machine instructions, depending on the value of the address operand. The assembler and linker make the decision about which sequence of machine instructions to use.

In most cases, a BRANCH, LOAD, or STORE synthetic instruction will be implemented by a single machine instruction with the corresponding similar name.

To make the distinction between machine instruction and synthetic instruction explicit, we assign different names. But to keep the correspondence obvious and the meaning clear, we use names that differ only by the presence of the period character.

We chose to use a period for the machine instructions on the assumption most assembly programs will contain the synthetic instructions, not the machine variants. The presence of periods (if any) will stand out. Also, programmers are more likely to err by forgetting a period, rather than inserting one, so we chose a naming scheme in which the programmer does not normally use the period character.

`*BEQ`	*Reg1,Reg2,address*	Branch if Reg1 = Reg2
`*BNE`	*Reg1,Reg2,address*	Branch if Reg1 ≠ Reg2
`*BLT`	*Reg1,Reg2,address*	Branch if Reg1 < Reg2
`*BLE`	*Reg1,Reg2,address*	Branch if Reg1 ≤ Reg2
`*BGT`	*Reg1,Reg2,address*	Branch if Reg1 > Reg2
`*BGE`	*Reg1,Reg2,address*	Branch if Reg1 ≥ Reg2

Synthetic, Variable Length, May Overwrite “t” Register, May cause a “Null Address Exception”

Register t Usage: May be modified; Okay to use as Reg1 and/or Reg2.

The values in Reg1 and Reg2 are compared. In the case of LT, LE, GT, and GE, the operand values are treated as signed integers.

If the condition is satisfied, a branch is taken.

In these synthetic instructions, “*address*” may be any absolute or relocatable address, except 0. (Any reference to address 0 always causes a Null Address Exception.)

Typically the programmer or compiler will use a symbolic label to stand for the address, but a hard-coded number can be used, too.

Note that an integer value indicates an absolute address, not a relative address. For example, the following will branch to location 0x0_0000_0008 and not to an instruction located two instructions beyond the branch instruction itself.

```
BEQ        r1,r2,+8
```

This is different behavior from the following machine instruction, which will skip the instruction following the branch instruction.

```
B.EQ       r1,r2,+8
```

In general, the actual target address will be a 36-bit address that is not known until link-time. In such cases, the actual instruction sequence may not be determined until link-time. (In many cases, the assembler will be able to safely produce the final instruction sequence. This happens when the target address is nearby and there are no intervening synthetic, variable length instructions.)

The target of most branch instructions will be within the range of -32,768 … +32,766 from the address of the branch instruction. (Since the target address must be halfword aligned, the high end of the range is 32,766, and not 32,767.)

If the address is within this range, then a single instruction will be used, as follows.

The BEQ instruction is implemented with:

```
B.EQ       Reg1,Reg2,immed16
```

The BNE instruction is implemented with:

```
B.NE       Reg1,Reg2,immed16
```

The BLT instruction is implemented with:

```
B.LT       Reg1,Reg2,immed16
```

The BLE instruction is implemented with:

```
B.LE       Reg1,Reg2,immed16
```

The BGT instruction is implemented by exchanging the registers and changing the test condition:

```
B.LT       Reg2,Reg1,immed16     Note: the test and registers are changed.
```

The BGE instruction is implemented by exchanging the registers and changing the test condition:

```
B.LE       Reg2,Reg1,immed16     Note: the test and registers are changed.
```

If the target is within the range of -524,288 … +524,286, then it can be reached with a JAL instruction. However, the JAL instruction is unconditional. To use it, the assembler/linker must change the sense of the branch (i.e., negate the condition) and use it to branch around the JAL instruction.

The BEQ instruction is implemented with:

```
  B.NE      Reg1,Reg2,+8        Note the test is reversed
  JAL       r0,address
```

The BNE instruction is implemented with:

```
  B.EQ      Reg1,Reg2,+8        Note the test is reversed
  JAL       r0,address
```

The BLT instruction is implemented with:

```
  B.LE      Reg2,Reg1,+8        Note the condition & regs are changed
  JAL       r0,address
```

The BLE instruction is implemented with:

```
  B.LT      Reg2,Reg1,+8        Note the condition & regs are changed
  JAL       r0,address
```

The BGT instruction is implemented with:

```
  B.LE      Reg1,Reg2,+8        Note the test is reversed
  JAL       r0,address
```

The BGE instruction is implemented with:

```
  B.LT      Reg1,Reg2,+8        Note the test is reversed
  JAL       r0,address
```

In the very rare cases where the target address is out of this range, a sequence of 3 instructions must be generated, as follows.

The temp-register “t” is used to build a 36 bit value. In the code below, “*upper-20*” indicates bits [35:16] of the address and “*lower-16*” indicates bits [15:0]. The JALR instruction will sign-extend the immediate *lower-16* value and add it to the register. To compensate, the value used for *upper-20* will have to be adjusted accordingly.

The BEQ instruction is implemented with:

```
  B.NE      Reg1,Reg2,+12       Jump around next 2 statements
  AUIPC     t,upper-20          Execute a long jump if EQ
  JALR      r0,lower-16(t)
```

The BNE instruction is implemented with:

```
  B.EQ      Reg1,Reg2,+12       Note the test is reversed
  AUIPC     t,upper-20
  JALR      r0,lower-16(t)
```

The BLT instruction is implemented with:

```
B.LE      Reg2,Reg1,+12        Note the condition & regs are changed
AUIPC     t,upper-20
JALR      r0,lower-16(t)
```

The BLE instruction is implemented with:

```
B.LT      Reg2,Reg1,+12        Note the condition & regs are changed
AUIPC     t,upper-20
JALR      r0,lower-16(t)
```

The BGT instruction is implemented with:

```
B.LE      Reg1,Reg2,+12        Note the test is reversed
AUIPC     t,upper-20
JALR      r0,lower-16(t)
```

The BGE instruction is implemented with:

```
B.LT      Reg1,Reg2,+12        Note the test is reversed
AUIPC     t,upper-20
JALR      r0,lower-16(t)
```

Commentary Since each instruction is only 32 bits, any operation involving a 36 bit address will necessarily require at least 2 instructions.

The Blitz-64 solution is to break a 36 bit address into two pieces, consisting of the most significant 20 bits and the least significant 16 bits. In some cases, we break a 32 bit value into two equal sized parts, of 16 bits each.

To explain how the assembler/linker produces machine code, we use the following notational abbreviations:

upper-20	The upper 20 bits of a 36 bit value
upper-16	The upper 16 bits of a 32 bit value
lower-16	The lower 16 bits of the value

Generally, the first instruction in the sequence will load the upper 20 bits and the second instruction will add in lower 16 bits.

For example, the following code sequence stores a byte from register "r5" into memory, using a 36 bit absolute address. The temporary register "t" is used to build the memory address.

```
UPPER20    t,upper-20          t = upper 20 bits [35:16]
STORE.B    lower-16(t),r5      address is upper-20 + lower-16
```

Note that the second instruction will sign-extended the lower-16 bit piece and perform an addition.

Therefore, the assembler/linker must be careful when computing the "*upper-20*" and "*lower-16*" pieces from an arbitrary 36-bit value. Because the *lower-16* piece will be sign-extended by the second instruction, the assembler/linker cannot use:

```
upper20 = Value[35:16]        Wrong!
lower16 = Value[15:0]
```

Instead, the assembler/linker must do this:

```
Given:
    Value (a 36-bit quantity)
Compute:
    lower16 = Value[15:0]
    x = Value – SignExtend (lower16)
    upper20 = (x >> 16) [19:0]        i.e., grab upper 20 bits [35:16] from x
```

The *upper-16* value is computed the same way, with the last line modified to:

```
    upper16 = (x >> 16) [15:0]        i.e., grab upper 16 bits [31:16] from x
```

Note that overflow cannot occur either in the subtraction performed by the assembler/linker, or the addition performed by the second instruction in the code sequence (e.g., the STORE.B). This is assuming that the original "Value" is limited to a quantity representable in 36 bits, which is true of all memory addresses and offsets.

`*BEQI`	*Reg,value,address*	Branch if Reg = immediate value
`*BNEI`	*Reg,value,address*	Branch if Reg ≠ immediate value
`*BLTI`	*Reg,value,address*	Branch if Reg < immediate value
`*BLEI`	*Reg,value,address*	Branch if Reg ≤ immediate value
`*BGTI`	*Reg,value,address*	Branch if Reg > immediate value
`*BGEI`	*Reg,value,address*	Branch if Reg ≥ immediate value

Synthetic, Variable Length, Will Overwrite "t" Register, May cause a "Null Address Exception"

Register t Usage: Will be modified; Must not use as Reg.

Since these instructions are synthesized with a MOVI instruction, the *value* can be any 64-bit value. Likewise, the *address* can be any address in memory, since BEQ/BNE/BLT/BLE/BGT/BGE can handle any address.

The BEQI instruction is implemented as:

```
    *MOVI      t,value
    *BEQ       Reg,t,address
```

The BNEI instruction is implemented as:

```
    *MOVI      t,value
    *BNE       Reg,t,address
```

The BLTI instruction is implemented as:

```
    *MOVI      t,value
    *BLT       Reg,t,address
```

The BLEI instruction is implemented as:

```
    *MOVI      t,value
    *BLE       Reg,t,address
```

The BGTI instruction is implemented as:

```
    *MOVI      t,value
    *BGT       Reg,t,address
```

The BGEI instruction is implemented as:

```
    *MOVI      t,value
    *BGE       Reg,t,address
```

`*BEQZ`	*Reg,address*	Branch if Reg = 0
`*BNEZ`	*Reg,address*	Branch if Reg ≠ 0
`*BLTZ`	*Reg,address*	Branch if Reg < 0, i.e., if negative
`*BLEZ`	*Reg,address*	Branch if Reg ≤ 0, i.e., if not positive
`*BGTZ`	*Reg,address*	Branch if Reg > 0, i.e., if positive
`*BGEZ`	*Reg,address*	Branch if Reg ≥ 0, i.e., if not negative

Synthetic, Variable Length, May Overwrite “t” Register, May cause a “Null Address Exception”

Register t Usage: May be modified; Okay to use as Reg.

Since these instructions are synthesized with BEQ/BNE/BLT/BLE/BGT/BGE, the *address* can be any address in memory.

The BEQZ instruction is implemented as:
`*BEQ` *Reg1,r0,address*
The BNEZ instruction is implemented as:
`*BNE` *Reg1,r0,address*
The BLTZ instruction is implemented as:
`*BLT` *Reg1,r0,address*
The BLEZ instruction is implemented as:
`*BLE` *Reg1,r0,address*
The BGTZ instruction is implemented as:
`*BGT` *Reg1,r0,address*
The BGEZ instruction is implemented as:
`*BGE` *Reg1,r0,address*

`*BFALSE`	*Reg,address*	Branch if Reg = 0, i.e., if “false”
`*BTRUE`	*Reg,address*	Branch if Reg ≠ 0, i.e., if “true”

<u>Synthetic, Variable Length, May Overwrite “t” Register, May cause a “Null Address Exception”</u>

<u>Register t Usage:</u> May be modified; Okay to use as Reg.

The BFALSE instruction is implemented as:
`*BEQ` *Reg1,r0,address*
The BTRUE instruction is implemented as:
`*BNE` *Reg1,r0,address*

`UPPER20`	*RegD,immed20*	RegD ← (immed<<16)

The 20 bit immediate value is sign-extended. It is then shifted left by 16 bits. The result is placed into register RegD.

The UPPER20 instruction is useful for building any 36 bit value, which is the size of a memory address. The UPPER20 instruction takes care of the most significant 20 bits. The following instruction (e.g., LOAD or STORE) will typically add in the least significant 16 bits and perform the access.

`UPPER16`	*RegD,Reg1,immed16*	RegD ← (immed<<16) + Reg1

The 16 bit immediate value is sign-extended and then shifted left by 16 bits. This value is added to the value in register Reg1 and the result is placed into register RegD. There is no overflow check.

The UPPER16 instruction is useful for building 32 bit offsets from a register such as the stack pointer “sp”. The UPPER16 instruction takes care of the most significant 16 bits of the offset and the addition of the stack pointer. The following instruction (e.g., LOAD or STORE) will typically add in the least significant 16 bits of the offset and perform the memory access.

Commentary Recall that the ADDI instruction is limited to an immediate value of -32,768 … +32,767:

```
    ADDI        RegD,Reg1,immed16
```

In order to add a larger number, the following code sequence is recommended and will work in all cases:

```
    MOVI        t,value
    ADD         RegD,Reg1,t
```

However, for a 32-bit value (i.e., in the range -2,147,483,648 … +2,147,483,647), notice that the synthetic MOVI will expand to two instructions, giving:

```
    UPPER20     t,upper-20
    XORI        t,t,lower-16
    ADD         RegD,Reg1,t
```

However, you might consider achieving the same effect with this shorter code sequence:

```
    UPPER16     t,Reg1,upper-16
    ADDI        RegD,t,lower-16
```

But beware: The overflow behavior is *not equivalent*! The UPPER16 instruction performs an addition which ignores overflow. The UPPER16 instruction is meant for addresses, so this is reasonable. UPPER16 is not meant for general purpose addition.

`SHIFT16`   *RegD,Reg1,immed16*   RegD ← (Reg1 + immed16) << 16

This instruction combines the immed16 value and the value in register Reg1 and places the computed result in register RegD. The 16 bit immediate value is injected into the lower 16 bits of the value in register Reg1. The value is then shifted left by 16 bits. The result is stored into register RegD.

By inject, we mean the 16 new bits overwrite the original bits [15:0] of the value fetched from Reg1. The "+" in the summary above is a bit misleading. The following is more precise:

RegD ← Reg1[47:16] || immed16 || 0x0000

The immediate value is not sign-extended and there is no overflow check.

This instruction is useful in loading arbitrary 64 bit values into a register. See the discussion for the MOVI instruction to see how this instruction is used.

`ADDPC` *RegD,immed20* RegD ← PC+immed

The 20 bit immediate value gives a PC-relative “target address” which is moved into register RegD.

The immediate value is sign-extended and added to the current value of the PC (the address of the ADDPC instruction, not the following instruction).

Since the PC and the offset are relatively small numbers, overflow is impossible. The PC is a positive number; i.e., it is not sign-extended.

This instruction is used in loading the address of a static variable or function into a register, when that address is within -524,288 … +524,287 of this instruction.

A program can determine its own address with this instruction. For example, xecuting “ADDPC r1,0” will move the address of the ADDPC instruction into register r1.

`AUIPC` *RegD,immed20* RegD ← (immed<<16) + PC

The AUIPC instruction is identical to the UPPER20 instruction, except that the PC is also added in.

In more detail, the 20 bit immediate value is sign-extended. It is then shifted left by 16 bits. This value is added to the current value of the PC (the address of this instruction, not the following instruction) and the result is placed in register RegD.

Since the PC and the offset are relatively small numbers, overflow is impossible. The PC is a positive number; i.e., it is not sign-extended.

The AUIPC instruction is useful for building any PC-relative relocatable address. The AUIPC instruction takes care of the most significant 20 bits. The following instruction (e.g., JALR, LOAD.x, etc.) will then add in the least significant 16 bits and perform the jump, load, etc.

JAL	*RegD,immed20*	RegD ← return addr; Target ← PC+offset

May cause a “Null Address Exception”

The 20 bit immediate value gives a PC-relative “target address". The immediate value is sign-extended and added to the current value of the PC (the address of the JAL instruction, not the following instruction). The address of the instruction following the JAL is stored into RegD, which is typically the link register, “lr”. Finally, the PC is loaded with the target address, causing a jump.

The upper 28 bits [63:36] of the target address are ignored, since addresses are 36 bits. The least significant bit of the address is ignored and 0 is assumed, forcing halfword alignment. There is no overflow check.

This instruction is used to implement the function CALL instruction. The return will be made to the instruction following the JAL, and this address is exactly what this instruction will save in the link register.

This instruction can also be used to implement a PC-relative jump or goto, in which case the zero register “r0” is used as the destination for the link value. Since there is to be no return, there is no reason to save a return address.

Any attempt to load the PC with zero will cause a “Null Address Exception”. Exceptions will only occur if the jump is taken; if the jump is not taken, no exception will occur.

In systems without compressed instructions and in which alignment is required for instructions, there may be an “Unaligned LOAD/STORE Exception” if the target address is not properly aligned. Alternatively, such systems may simply ignore the final bits, rounding the target address down to force alignment.

`JALR`	*RegD,immed16(Reg1)*	RegD ← return addr; Target ← offset+Reg1

May cause a “Null Address Exception”

The 16 bit immediate value is sign-extended and added to the current value of register Reg1, giving a “target address”. The address of the instruction following the JALR is stored into RegD, which is typically the link register, “lr”. Finally, the PC is loaded with the target address, causing a jump.

The upper 28 bits [63:36] of the target address are ignored, since addresses are 36 bits. The least significant bit of the address is ignored and 0 is assumed, forcing halfword alignment. There is no overflow check.

This instruction can be used to implement an indirect jump, via register. It is also used to implement the RETURN instruction. It is also used to implement the CALL instruction when the target address exceeds the 20 bits accommodated by the JAL instruction.

Any attempt to load the PC with zero will cause a “Null Address Exception”. Exceptions will only occur if the jump is taken; if the jump is not taken, no exception will occur.

In systems without compressed instructions and in which alignment is required for instructions, there may be an “Unaligned LOAD/STORE Exception” if the target address is not properly aligned. Alternatively, such systems may simply ignore the final bits, rounding the target address down to force alignment.

`*CALL`	*address*	Jump to address; save return addr in “lr”
`*CALLR`	*Reg1*	Jump to address; save return addr in “lr”

May cause a “Null Address Exception”

Synthetic, Variable Length.

Register t Usage: May be modified (CALL); Okay to use as Reg1.

In the case of the CALL instruction, the target is given by the “*address*” operand, and may be any absolute or relocatable address. Typically the programmer or compiler will use a symbolic label to stand for the address, but a hard-coded number can be used, too.

In the case of CALLR, the target address is in register Reg1.

Since all program-generated addresses are 36 bits, only the lower 36 bits of any target address can affect the effective address. The upper 28 bits [63:36] of any target address are always ignored.

The CALLR instruction is implemented with:

```
    JALR        lr,0(Reg1)
```

In the case of CALL, the *address* will not normally be known until link-time. Consequently, the actual instruction sequence may not be determined until link-time. (However in some cases, the assembler will be able to produce the final instruction sequence.)

If *address* is within the range -524,288 … +524,287 from the CALL instruction (i.e., within the range of a 20-bit offset), then CALL is implemented with:

```
    JAL         lr,address
```

Otherwise (i.e., a full 36 bit relative offset from the PC is needed), then CALL is implemented with:

```
    AUIPC       t,upper-20
    JALR        lr,lower-16(t)
```

If the target *address* is an absolute address in the range -32,768 … +32,767 (i.e., within the lowest 32 GiBytes of the address space 0x0_0000_0000 … 0x0_0000_7FFF or within the highest 32 GiBytes of the address space 0xF_FFFF_8000 … 0xF_FFFF_FFFF), then CALL is implemented with:

```
    JALR        lr,address(r0)
```

Otherwise (i.e., an absolute address is provided and a full 36 bits are required), CALL is implemented with:

```
    UPPER20     t,upper-20
    JALR        lr,lower-16(t)
```

In the above code, "*upper-20*" indicates bits [35:16] of the address and "*lower-16*" indicates bits [15:0]. The JALR instruction will sign-extend the immediate *lower-16* value and add it to the register. To compensate, the value used for *upper-20* will have to be adjusted accordingly.

`*JUMP` *address* Jump to address

Synthetic, Variable Length, May cause a “Null Address Exception”

Register t Usage: May be modified.

See the comments regarding “*address*” for the CALL instruction.

This instruction is implemented exactly like the CALL instruction, except the register r0 is used for the link register. In other words, the return address is discarded, instead of saved.

If *address* is within the range -524,288 … +524,287 from the JUMP instruction (i.e., within the range of a 20-bit offset), then JUMP is implemented with:

```
    JAL        r0,address
```

Otherwise (i.e., a full 36 bit relative offset from the PC is needed), then JUMP is implemented with:

```
    AUIPC      t,upper-20
    JALR       r0,lower-16(t)
```

If the target *address* is an absolute address in the range -32,768 … +32,767, then JUMP is implemented with:

```
    JALR       r0,address(r0)
```

If an absolute addressing requiring 36 bits is given, then JUMP is implemented with:

```
    UPPER20    t,upper-20
    JALR       r0,lower-16(t)
```

In the above code, “*upper-20*” indicates bits [35:16] of the address and “*lower-16*” indicates bits [15:0]. The JALR instruction will sign-extend the immediate *lower-16* value and add it to the register. To compensate, the value used for *upper-20* will have to be adjusted accordingly.

`*JR` *Reg1* Indirect jump, via register

Synthetic, May cause a “Null Address Exception”

Register t Usage: Not used; Okay to use as Reg1.

This instruction jumps to the address contained in the register.

It is implemented with:

```
    JALR        r0,0(Reg1)
```

`*RET` *<no operands>* Return value is in link reg “lr”

Synthetic, May cause a “Null Address Exception”

Register t Usage: Not used.

A jump is made to the address saved in the link register.

This instruction is implemented with:

```
    JALR        r0,0(lr)
```

Commentary The basic approach to function call and return is to store the return address (i.e., the address of the instruction following the CALL instruction) in a register. By convention, one register (named “lr”) is set aside for this purpose.

A “leaf” function is a function that does not call any other functions. For leaf functions, there is no need to save the return address on the stack, since it can remain undisturbed in register “lr” until the function is ready to return. This avoids two (costly) accesses to memory, one to save the return address and one to restore it.

Some functions can pass all arguments and return values in registers and can store all local variables in registers. Such lucky functions can get by without needing to use the stack and can execute without ever accessing memory, which enhances execution speed.

In the case of non-leaf functions, this link register scheme will not work. Instead, the function must explicitly save its return address, and this is normally done on the stack. Thus, the function “entry prologue” will include a STORE instruction to save the return address, while the function “exit epilogue” will include a LOAD instruction directly before the RETURN instruction.

`ENTERFUN`	*RegD,Reg1,immed16*	Save lr, push frame onto stack
`EXITFUN`	*RegD,Reg1,immed16*	Retrieve lr, pop frame, and return

May cause a page-related exception, “Unaligned LOAD/STORE Exception”, “Null Address Exception”, “Arithmetic Exception”, or “Stack Overflow Exception”

The ENTERFUN instruction has the same effect as the following instruction sequence:

```
  STORED    -8(sp),lr
  ADDI      RegD,Reg1,immed16
```

The EXITFUN instruction has the same effect as:

```
  ADDI      RegD,Reg1,immed16
  LOAD      pc,-8(RegD)
```

Of course, the Program Counter (PC) is not a directly addressable register. The above pseudo-code for EXITFUN is merely suggestive. In reality, a doubleword is simply fetched from memory and used as the target address to jump to.

In the case of ENTERFUN, when an exception arises from the store operation, the destination register RegD may or may not be modified. If the addition causes overflow, an Arithmetic Exception will occur. If the instruction modifies the **sp** register and the new value is less than the **StackLimit** field in **csr_status**, a "Stack Overflow Exception will occur. If the addition causes an exception, the store operation may or may not be performed.[4] These are implementation dependencies.

[4] If the instruction raise more than one exception, only one exception will occur, but we do not specify which exception takes priority. This is an implementation dependency for both ENTERFUN and EXITFUN.

In the case of EXITFUN, when an Arithmetic Exception arises from the addition, the load operation may or may not be performed and the PC may or may not be updated. If the load operation causes an exception, the addition may or may not be performed. Whenever an exception occurs, trap processing saves the previous value of the PC in **csr_prevpc**. If the load operation completes without error, but the addition results in an exception, it is indeterminate which value stored in **csr_prevpc**. All these are implementation dependencies.

The memory address is computed by adding -8 to the contents of a register (normally **sp**). Overflow on this addition is ignored. The upper 28 bits [63:36] of the address are ignored and only the lower 36 bits [35:0] are used. The resulting address must be non-zero; any attempt to load or store into address zero will result in a "Null Address Exception" being signaled. The resulting address must be doubleword aligned; if not, a "Unaligned LOAD/STORE Exception" will be signaled.

Both the addition and the memory operation are performed atomically. Assuming there are no exceptions, if one operation is performed the other will also be performed without the possibility of traps occurring between them, i.e., in the middle of the instruction.

Commentary To understand the purpose of ENTERFUN and EXITFUN, consider the following example function, which allocates a frame of 24 bytes on the stack. The return address along with a couple of registers are stored in the frame on entry and then restored before returning.

According to the standard program calling conventions, the return address must always be stored in the top (i.e., highest) doubleword of the frame. Other locations in the frame will be used in different ways, depending on the needs of the particular function.

```
  myFunc:
          addi      sp,sp,-24    # Function Prologue
          store.d   0(sp),r1     # .
          store.d   8(sp),r2     # .
          store.d   16(sp),lr    # .
          ...
          load.d    r1,0(sp)     # Return Sequence
          load.d    r2,8(sp)     # .
          load.d    lr,16(sp)    # .
          addi      sp,sp,24     # .
          ret                    # .
```

We can rewrite the above code as follows:[5]

```
myFunc:
          store.d   -8(sp),lr    # Function Prologue
          addi      sp,sp,-24    # .
          store.d   0(sp),r1     # .
          store.d   8(sp),r2     # .
          ...
          load.d    r1,0(sp)     # Return Sequence
          load.d    r2,8(sp)     # .
          addi      sp,sp,24     # .
          load.d    lr,-8(sp)    # .
          ret                    # .
```

The highlighted code above can be replaced by the ENTERFUN and EXITFUN instructions to give:

[5] Note that the re-written version saves register **lr** at **-8(sp)**, beyond and outside the stack (i.e., "above" the "top" of the stack using a negative offset to **sp**). In certain situations, this may be risky. If interrupts are possible and the interrupting trap handler saves things on the stack, disaster will result if the trap occurs between saving **lr** and decrementing **sp**, since the saved **lr** will be overwritten by the trap handler. With ENTERFUN and EXITFUN, interrupts cannot occur in the middle between these two operations, so this problem is avoided. However, in certain situations, the KPL compiler will nevertheless generate code that saves values using negative offsets from **sp**.

```
myFunc:
           enterfun  sp,sp,-24    # Function Prologue
           store.d   0(sp),r1     # .
           store.d   8(sp),r2     # .
           ...
           load.d    r1,0(sp)     # Return Sequence
           load.d    r2,8(sp)     # .
           exitfun   sp,sp,24     # .
```

Normally, register **sp** will be used as both Reg1 and RegD in both ENTERFUN and EXITFUN. It would be uncommon for any other register to be used.

If the frame size exceeds what is representable in 16 bits, then we cannot use ENTERFUN or EXITFUN or ADDI. In that case, the compiler will need to fall back on a more general code sequence.

Commentary The ENTERFUN and EXITFUN instructions are unusual in the following ways:

Two registers are implied — the **sp** and **lr** registers — in the load and store portions of the ENTERFUN and EXITFUN functionality. In all other Blitz-64 instructions, if a general purpose register is used, it will be specified explicitly in the instruction. Furthermore, wherever a register is specified, any general purpose register can be used. In this way, the **sp** and **lr** registers now have a special use that is not shared by the other registers.

The ENTERFUN and EXITFUN instructions each perform two very different operations. The instruction performs both an addition to a register and a memory operation. In all other Blitz-64 instructions, each instruction is limited to a single, simple operation. For the most part, the Blitz-64 architecture is RISC, but these instructions can reasonably be termed CISC instructions.

```
LOAD.B      RegD,immed16(Reg1)
LOAD.H      RegD,immed16(Reg1)
LOAD.W      RegD,immed16(Reg1)
LOAD.D      RegD,immed16(Reg1)

STORE.B     immed16(Reg1),Reg2
STORE.H     immed16(Reg1),Reg2
STORE.W     immed16(Reg1),Reg2
STORE.D     immed16(Reg1),Reg2
```

May cause a page-related exception, “Unaligned LOAD/STORE Exception”, or “Null Address Exception”

The LOAD instructions transfer a byte/halfword/word/doubleword from main memory to the destination register RegD.

In the case of a LOAD of less than 64 bits, the value will be signed-extended to 64 bits.

The STORE instructions transfer a byte/halfword/word/doubleword from register Reg2 to main memory.

In the case of a STORE of less than 64 bits, the upper bits of the register will be ignored. There will not be a “Arithmetic Exception” signaled.

The address is computed by sign-extending the 16 bit immediate value to 64 bits and adding it to the contents of register Reg1. Overflow on this addition is ignored. The upper 28 bits [63:36] of the address are ignored and only the lower 36 bits [35:0] are used.

The resulting address must be non-zero; any attempt to load or store into address zero will result in a “Null Address Exception” being signaled.

The resulting address must be properly aligned for the size being transferred; if not, a “Unaligned LOAD/STORE Exception” will be signaled.

Commentary The large size of the 16 bit immediate offset in the LOAD and STORE instructions provides a lot of flexibility in addressing.

To access values stored on the stack using LOAD and STORE instructions, the stack pointer register "sp" can be used as Reg1. The stack grows downward toward low memory. Therefore, positive offsets from "sp" can be used to access any data within the top 32KiBytes of the stack.

Many language compilers maintain a "frame pointer" as well as a "stack pointer". The frame pointer (often called "fp") is used in the implementation of stack frames for storing local variables in a function invocation. The "fp" register will typically be initialized to point to the boundary with the previous stack frame. That is, "fp" will point to the top of the current frame and the bottom of the previous frame. As always, the "sp" register points to the stack top, accommodating dynamically-sized stack frames and ad hoc pushing and popping onto the stack.

In this approach, negative offsets from "fp" will access variables in the current (top) frame and positive offsets will access variable in the previous frames, such as function arguments. Given the range of the immed16 offset, this method will accommodate stack frames of up to 32 KiBytes in size. For stack frames larger than this, an additional instruction may be required for some variables. Thus, in almost all cases, stack frame variables will be accessible with a single instruction.

The Blitz-64 calling conventions do not include a frame pointer register. In other words, there is no register named "fp". However, for functions that need a frame pointer, the compiler can choose any register to use. Often, there will be no separate thread data area, so register "tp" is normally the logical choice to use as a frame pointer. By convention, the compiler will use "tp" as a frame pointer. To prevent confusion, we intentionally do not give the "tp" a second name, such as "fp".

Data within the lower 32 KiBytes of main memory (addresses 0x0_0000_0000… 0x0_0000_7FFF, i.e., decimal 0…32,767) can be conveniently accessed by using the zero register "r0" for Reg1.

Due to the fact that the upper bits [63:36] of addresses are ignored, data within the upper 32 KiBytes of addressable memory (i.e., addresses 0xF_FFFF_8000… 0xF_FFFF_FFFF) is also accessible with negative addresses (i.e., decimal -32,768 … -1). This is achieved with a negative immed16 value.

Assuming the global data pointer register "gp" has been initialized, a range of 64 KiBytes of static variable data can be accessed with a single LOAD or STORE instruction. This can be achieved by initializing the global data pointer "gp" to point to the center of the block of 64 GiBytes of global data. The first half of the global data will be accessed with a negative offset and the upper half will be accessed with a positive offset.

Assuming that some register points to an “object” (in the sense of object-oriented programming), a single LOAD or STORE instruction can be used to access any field within the object, as long as the object is not larger than 32 KiBytes.

For data located at any other address, an additional instruction will be required. See the UPPER20 and UPPER16 instructions.

Concerning Atomicity of LOAD and STORE Instructions The following machine instructions are atomic:

```
LOAD.B    LOAD.H    LOAD.W    LOAD.D
STORE.B   STORE.H   STORE.W   STORE.D
CAS
```

This means the memory operation occurs as a single, uninterruptible unit. Conflicting memory operations which touch the same or overlapping memory locations will be serialized, which means one operation will be executed to completion entirely before the other operation begins execution.

This assumes that LOADs and STOREs are aligned; if they are not aligned, then an “Unaligned LOAD/STORE Exception” will occur and atomicity becomes a software issue, and only if the instruction is to be emulated.

Within the private memory of a single core, unaligned operations may be atomic; this is implementation dependent. However, when performed on a shared memory address in the presence of multiple processors, the programmer must be careful when using unaligned operations. Consider two more-or-less simultaneous attempts to STORE different values into a single doubleword that happens to cross a boundary and involves two cache lines. Even if operations involving a single cache line are atomic, an operation involving two cache lines is not normally atomic.

We chose the granularity of atomicity to be 64 bits on the assumption that all memory busses and transfer paths to memory and caches will be at least 64 bits in width, or at least all bus transactions will accommodate 64 bits. Thus, the atomicity of LOADs and STOREs will “come for free”. Perhaps the system busses will transfer data in larger units, such as cache lines of 128 bytes, but this should never be relied upon.

Relying on the atomicity of memory operations is somewhat error prone. To guard against race-related bugs, all potentially shared data should be protected by software locks. However, for the efficient implementation of “mutex” locks, the atomicity of LOAD.x, STORE.x, and CAS (compare-and-set) is critical.

`*LOADB`	*RegD,address*	Where *address* is any value
`*LOADH`	*RegD,address*	
`*LOADW`	*RegD,address*	
`*LOADD`	*RegD,address*	
`*LOADB`	*RegD,offset(Reg1)*	Where *offset* is any value
`*LOADH`	*RegD,offset(Reg1)*	
`*LOADW`	*RegD,offset(Reg1)*	
`*LOADD`	*RegD,offset(Reg1)*	

<u>Synthetic, Variable Length, may cause a page-related exception, “Unaligned LOAD/STORE Exception”, or “Null Address Exception”</u>

<u>Register t Usage:</u> Not used; Okay to use as RegD or Reg1.

<u>Register Note:</u> RegD and Reg1 must be different !

In these synthetic instructions, “*address*” may be any absolute or relocatable address. Typically the programmer or compiler will use a symbolic label to stand for the address, but a hard-coded number can be used, too. The “*offset*” operand may be any absolute value.

Both *address* and *offset* are limited to 36 bits. Since all program-generated addresses are 36 bits, only the lower 36 bits of any *address* or *offset* can affect the effective address. The upper 28 bits [63:36] of any address are always ignored.

Normally, the *address* will not be known until link-time. The *offset* may also be unknown until link-time. In such cases, the actual instruction sequence may not be determined until link-time. (However in some cases, the assembler will be able to produce the final instruction sequence.)

In the following, we describe the sequence of machine instructions produced for a LOADB instruction. The LOADH, LOADW, and LOADD instructions are handled analogously.

Consider an instruction of the form:

```
    LOADB      RegD,address
```

If the value of *address* is within the range of -32,768 … +32,767, then use:

```
    LOAD.B     RegD,address(r0)
```

If *address* is any other value, then use:

```
    UPPER20    RegD,upper-20
    LOAD.B     RegD,lower-16(RegD)
```

The above code works for any absolute address. Normally the linker will convert (i.e., "resolve") all addresses to absolute numbers. But if PC-relative addressing is demanded, the following sequence must be used, regardless of the size of the offset:

```
    AUIPC      RegD,upper-20
    LOAD.B     RegD,lower-16(RegD)
```

Consider an instruction of the form:

```
    LOADB      RegD,offset(Reg1)
```

If *offset* is within the range of -32,768 … +32,767, then use:

```
    LOAD.B     RegD,offset(Reg1)
```

If *offset* is within 32 bits (i.e., within the range -2,147,483,648 … +2,147,483,647), then use:

```
    UPPER16    RegD,Reg1,upper-16
    LOAD.B     RegD,lower-16(RegD)
```

If *offset* is a value even larger than 32 bits, then use the following. (An offset larger than 32 bits would be quite rare, so this sequence won't be needed often.)

```
    UPPER20    RegD,upper-20
    ADD        RegD,RegD,Reg1
    LOAD.B     RegD,lower-16(RegD)
```

(This sequence contains an ADD instruction; can this cause an "Arithmetic Exception"? The UPPER20 instruction is only capable of loading a 36 bit value.

Assuming that Reg1 contains a legal address (i.e., a value limited to 36 bits) the ADD instruction cannot cause an Arithmetic Exception.)

In the above code, “*upper-16*” indicates the upper 16 bits [31:16] of the value, “*upper-20*” indicates the upper 20 bits [35:16] of the value, and “*lower-16*” indicates bits [15:0]. The LOAD instruction will sign-extend the immediate *lower-16* value and add it to the register. To compensate, the value used for *upper*-16 and/or upper-*20* will have to be adjusted accordingly.

To understand why *RegD* and *Reg1* must be different, consider the following when the offset is large:

```
   LOADB      r5,offset(r5)
```

Here is the code generated; obviously it will not work correctly.

```
   UPPER20    r5,upper-20         Error: Original value of r5 is lost.
   ADD        r5,r5,r5
   LOAD.B     r5,lower-16(r5)
```

```
*STOREB       address,Reg2              Where address is any value
*STOREH       address,Reg2
*STOREW       address,Reg2
*STORED       address,Reg2

*STOREB       offset(Reg1),Reg2         Where offset is any value
*STOREH       offset(Reg1),Reg2
*STOREW       offset(Reg1),Reg2
*STORED       offset(Reg1),Reg2
```

Synthetic, Variable Length, May cause a page-related exception, “Unaligned LOAD/STORE Exception”, or “Null Address Exception”

Register t Usage: May be modified; Must not use as Reg1 and/or Reg2 !

Concerning “*address*” and “*offset*”, see the comments under the LOAD instructions.

As with the LOAD instructions, we describe how the synthetic instruction can be implemented with 1, 2, or 3 instructions.

In some cases we need a register in which to build a large address. In the case of the LOAD instructions, we used the destination register RegD, since it was to be overwritten anyway. In the case of the STORE instructions, this will not work. Instead, we use the temporary register “t”.

Consider an instruction of the form:

```
    STOREB      address,Reg2
```

If the value of *address* is within the range of -32,768 … +32,767, then use:

```
    STORE.B     address(r0),Reg2
```

If *address* is any other value, then use:

```
    UPPER20     t,upper-20
    STORE.B     lower-16(t),Reg2
```

If PC-relative addressing is demanded, then use:

```
    AUIPC       t,upper-20
    STORE.B     lower-16(t),Reg2
```

Consider an instruction of the form:

```
    STOREB      offset(Reg1),Reg2
```

If *offset* is within the range of -32,768 … +32,767, then use:

```
    STORE.B     offset(Reg1),Reg2
```

If *offset* is within 32 bits (i.e., within the range -2,147,483,648 … +2,147,483,647), then use:

```
    UPPER16     t,Reg1,upper-16
    STORE.B     lower-16(t),Reg2
```

If *offset* is a value even larger than 32 bits, then use:

```
    UPPER20     t,upper-20
    ADD         t,t,Reg1
    STORE.B     lower-16(t),Reg2
```

`CAS` *RegD,Reg1,Reg2,Reg3* Compare and Set

May cause a page-related exception, or “Null Address Exception”

Register Reg1 contains the address of a doubleword, Reg2 contains the expected “old” value, and Reg3 contains the “new”value.

This operation will load a doubleword from memory. If the value is equal to the expected old value (in Reg2), then memory will be updated by storing the new value (in Reg3) into memory and 1 will be moved into RegD. If not equal, memory will not be updated and 0 will be moved into RegD.

More precisely, this instruction does the following as one atomic operation:

```
if *Reg1 == Reg2
  *Reg1 ← Reg3
  RegD ← true
else
  RegD ← false
endIf
```

The address in Reg1 is forced to be doubleword aligned by ignoring the final 3 bits. Thus, an “Unaligned LOAD/STORE Exception” cannot occur.

This instruction is not normally used on memory-mapped I/O devices. This instruction is implementation dependent if performed on a memory-mapped I/O address and may not work as expected.

Commentary The compare-and-set (CAS) instruction is used for concurrency control to allow synchronization between multiple threads which may be running on different cores accessing shared memory.

Consider implementing a mutex lock, which will be represented as a doubleword with 0=unlocked and 1=locked. If it is currently locked, the following code will spin in a tight loop continually executing the CAS instruction to check it.

<u>To acquire the lock:</u>

```
        movi     r1,…addressOfLock…
        movi     r3,1
  loop:
        cas      r7,r1,r0,r3
        beqz     r7,loop
```

<u>To release the lock:</u>

```
        movi     r1,…addressOfLock…
        store.d  0(r1),r0
```

Next, consider the case where we have several cores, each executing a thread with the goal of selecting one of the cores as a "leader". For example, we might want all cores to agree on which core will act as "master", with the others acting as "slaves".

Assume that each core has a unique ID number and the goal is for these concurrent processes to select exactly one core. We will assume there is a shared memory location (which we will name *Leader*) which will be used for the election. The *Leader* variable is assumed to be a doubleword initialized to zero.

Here is the code that each core will execute in order to chose their leader:

```
        movi     r1,…address of Leader…
        movi     r3,…my core ID…
        cas      r7,r1,r0,r3
        bnez     r7,WeWon
  WeLost:
        load.d   …,0(r1)       Load the ID of the leader
```

<u>Commentary</u> To use the CAS instruction, the KPL programming language has a built-in function with this usage:

function cas (p: **ptr to int**, old: **int**, new: **int**) **returns bool**

Such a function could be implemented in assembly code as:

```
  casFunct:
        cas    r1,r1,r2,r3
        ret
```

However, the compiler will recognize this as a predefined function and insert the CAS instruction inline.

FENCE *<no operands>*

<u>No exceptions; Not privileged</u>

You might imagine that the processor fetches instructions in order and executes them in sequence; this is the programming model that programmers and compilers naturally adopt.

However, to increase performance, modern processors will sometimes execute instructions out of sequence. This is acceptable as long as the effect is identical to executing them in the sequence they appear.

In reordering instructions, the processor implicitly assumes that there are no other processors. However, this may not be true and the results can be incorrect in a multiprocessor system with shared memory. Programmers assume the operations in their code are executed in the sequence written but, with concurrent algorithms, violations of this assumption can cause race bugs.

The FENCE instruction is used to ensure critical instructions complete before other instructions begin. Therefore, FENCE constrains and limits out-of-order execution and may introduce delays and pipeline bubbles.

The FENCE instruction affects instructions that read or write to memory. This includes:

> LOAD.B, LOAD.H, LOAD.W, LOAD.D
> STORE.B, STORE.H, STORE.W, STORE.D
> CAS, TLBCLEAR, TLBFLUSH, CHECKADDR

The FENCE instruction requires that any of the above instructions that appear before the FENCE instruction will be completed before the FENCE instruction. It requires that any of the above instructions that appear after the FENCE instruction will not be started until after the FENCE instruction.

To say this another way, let's call all memory-related instructions that appear in the instruction stream before the FENCE as *X* and all memory-related instructions that appear after the FENCE as *Y*. The FENCE instruction says, “Finish executing all *X* instructions before starting the execution of any *Y* instructions.”

Commentary In some ISAs, a distinction is made between operations that read memory and operations that write memory. Also a distinction can be made between operations that affect memory and operations that affect I/O. RISC-V is an example.

We chose to keep it simple and provide a single FENCE instruction for all cases.

Commentary To use the FENCE instruction, the KPL programming language has a built-in function which takes no arguments and returns no result:

```
function fence ()
```

Such a function could be implemented in assembly code as:

```
fenceFunct:
      fence
      ret
```

However, the compiler will recognize this as a predefined function and insert the FENCE instruction inline.

Like an out-of-order processor, compilers also reorder instructions.

In fact, the compiler can make major changes to improve performance, for example, by keeping variables in registers and delaying writes to memory for long periods of time. While these optimizations improve performance, they also open the door for race bugs in concurrent programs. Thus, there must be a way to instruct the compiler when to limit optimizations and execute the operations in the order specified.

In addition to inserting a FENCE instruction, the KPL compiler will also recognize the use of the predefined “fence ()” function as a signal to avoid the sorts of reordering and register caching that could confuse and break concurrent code.

The Blitz approach may seem like a blunt force tool and it is certainly the case that other ISAs and languages provide a finer-grained level of control. However, we chose the simple approach of Blitz for two reasons.

First, since it is simpler, we may avoid a programmer's failure to fully and adequately constrain memory operations that could occur with a finer level of control. In other words, a more complicated approach opens the way for the programmer to use it incorrectly. We feel that race bugs are so problematic that anything we can do is worthwhile.

Second, the sections of code that will be affected by a particular use of "fence" will probably be quite small. We do not expect many variables to be "in play" and subject to movement around a particular use of "fence" beyond the variables that we are intending to constrain, so the only optimizations that are eliminated are the ones we need to eliminate. The overly blunt "fence" of Blitz will not cause many unwanted, unintended inefficiencies. (There's probably an entire PhD thesis to be had in sorting this issue out.)

Although this is never a good justification, the Blitz/KPL approach may be easier to implement than the approach of declaring certain variables to be "volatile".

`ALIGNH`	*RegD,Reg1,Reg2,Reg3*	Reg3 (unaligned addr) gives shift amount
`ALIGNW`	*RegD,Reg1,Reg2,Reg3*	Reg3 (unaligned addr) gives shift amount
`ALIGND`	*RegD,Reg1,Reg2,Reg3*	Reg3 (unaligned addr) gives shift amount

The ALIGN instructions support the emulation of memory LOADs in which the target address is not properly aligned.

The high-order portion of the data comes from Reg1, the low-order portion of the data comes from Reg2. The least significant bits in Reg3 tell how to shift/combine the portions. The result is sign-extended and placed in RegD.

The ALIGN instructions use only the least signifiant bits of Reg3, so the (possibly unaligned) target address itself can be used. These bits are the final bits of the address and tell how "misaligned" the address is, i.e., how much shifting must be done.

<u>Commentary</u> Let's look at the operation of the ALIGN instructions in more detail.

<u>ALIGNW</u>

For the ALIGNW instruction, the goal is to select four bytes, which may be aligned in any of four ways.

Assume the memory contains the following sequence of bytes, where xx represents one byte. Word alignment boundaries are indicated with a period, so words AA BB CC DD and EE FF GG HH are properly aligned. The (possibly unaligned) target address will fall into one of the four cases shown.

```
    …  xx xx xx xx.AA BB CC DD.EE FF GG HH.xx xx xx xx  …
0                  AA BB CC DD
1                     BB CC DD EE
2                        CC DD EE FF
3                           DD EE FF GG
```

The LOADW instruction only loads properly aligned words. In order to retrieve a (possibly misaligned) word such as CC DD EE FF, we will LOAD two consecutive words. Regardless of alignment, this guarantees that we will get all the bytes.

Assume that Reg1 and Reg2 have been loaded using the LOADW instruction from two consecutive words in the memory that contain the desired word.

```
Reg1   xx xx xx xx AA BB CC DD
Reg2   xx xx xx xx EE FF GG HH
```

The result placed in RegD will be determined by the least significant two bits in Reg3, i.e., the final bits of the address. The resulting word will be sign-extended to fill RegD.

Reg3	**Result in RegD**
00	`ss ss ss ss AA BB CC DD`
01	`ss ss ss ss BB CC DD EE`
10	`ss ss ss ss CC DD EE FF`
11	`ss ss ss ss DD EE FF GG`

ALIGNH

For the ALIGNH instruction, the goal is to select two bytes, which may be aligned in any of two ways.

Assume the memory contains the following sequence of bytes, where xx represents one byte. Halfword alignment boundaries are indicated with a period, so words AA BB and CC DD are properly aligned. The (possibly unaligned) target address will fall into one of the two cases shown.

```
          …  xx.xx xx.AA BB.CC DD.xx xx.xx  …
   0                    AA BB
   1                       BB CC
```

The LOADH instruction only loads properly aligned halfwords. In order to retrieve a (possibly misaligned) halfword such as BB CC, we will LOAD two consecutive halfwords. Regardless of alignment, this guarantees that we will get all the bytes.

Assume that Reg1 and Reg2 have been loaded using the LOADH instruction from two consecutive halfwords in the memory that contain the desired halfword.

```
   Reg1    xx xx xx xx xx xx AA BB
   Reg2    xx xx xx xx xx xx CC DD
```

The result placed in RegD will be determined by the least significant bit in Reg3, i.e., the final bit of the address. The resulting halfword will be sign-extended to fill the RegD.

Reg3	**Result in RegD**
0	ss ss ss ss ss ss AA BB
1	ss ss ss ss ss ss BB CC

ALIGND

For the ALIGND instruction, the goal is to select eight bytes, which may be aligned as follows. The period indicates properly aligned doubleword boundaries. The (possibly unaligned) target address will fall into one of the eight cases shown.

```
        … .AA BB CC DD EE FF GG HH.II JJ KK LL MM NN OO PP. …
   0       AA BB CC DD EE FF GG HH
```

```
1         BB CC DD EE FF GG HH II
2            CC DD EE FF GG HH II JJ
3               DD EE FF GG HH II JJ KK
4                  EE FF GG HH II JJ KK LL
5                     FF GG HH II JJ KK LL MM
6                        GG HH II JJ KK LL MM NN
7                           HH II JJ KK LL MM NN OO
```

The LOADD instruction only loads properly aligned doublewords. In order to retrieve a (possibly misaligned) doubleword, we will LOAD two consecutive doublewords.

Assume that Reg1 and Reg2 have been loaded using the LOADD instruction from two consecutive doublewords in the memory that contain the desired halfword.

Reg1	AA BB CC DD EE FF GG HH
Reg2	II JJ KK LL MM NN OO PP

The result placed in RegD will be determined by the least significant three bits in Reg3, i.e., the final bits of the address.

Reg3	**Result in RegD**
000	AA BB CC DD EE FF GG HH
001	BB CC DD EE FF GG HH II
010	CC DD EE FF GG HH II JJ
011	DD EE FF GG HH II JJ KK
100	EE FF GG HH II JJ KK LL
101	FF GG HH II JJ KK LL MM
110	GG HH II JJ KK LL MM NN
111	HH II JJ KK LL MM NN OO

Code Examples

Here is an example of how to use the ALIGNW instruction. Assume that we wish to load a word into register RegD from the (possibly unaligned) target address contained in register "RegAddr". This code requires two additional registers, represented as RegLo and RegAlign.

Registers "RegD" and "RegLo" will be used to contain two consecutive words from memory, which will contain the target 4 bytes somewhere within them. First, we must modify the address to force alignment, to avoid a “Unaligned LOAD/STORE Exception”, placing the rounded-down version of the address in register “RegAlign”. Then we load two consecutive words from memory. Finally, we use the ALIGNW instruction to compute the desired result.

```
    ANDI     RegAlign,RegAddr,0xFFFC
    LOADW    RegD,0(RegAlign)
    LOADW    RegLo,4(RegAlign)
    ALIGNW   RegD,RegD,RegLo,RegAddr
```

[Note that, in the case when the address happens to be correctly aligned, the second LOAD instruction is unnecessary. Also note that if the target word happens to be the last word in a page, the second LOAD will retrieve data on a different page than the first LOAD. In rare cases, this second page could have different permissions or be an unallocated page, causing an exception to occur. This exception is extraneous and should be avoided since it could cause the program to fail. To avoid this, the programmer could add an extra 8 bytes to the end of the data, which will guarantee that an extraneous unnecessary LOAD will not cause problems. Or the programmer could use the approach described next.]

Here is a variation which avoids an unnecessary LOADW in the case where the address happens to be correctly aligned.

```
    ANDI     RegAlign,RegAddr,0xFFFC
    LOADW    RegD,0(RegAlign)
    ANDI     t,RegAddr,0x03
    BEQZ     t,EndLabel
    LOADW    RegLo,4(RegAlign)
    ALIGNW   RegD,RegD,RegLo,RegAddr
  EndLabel:
```

To determine which of these sequences is superior will require a performance analysis and depend on the relative costs of LOADW versus the ANDI/BEQZ/ ALIGNW instructions.

Without the ALIGN instructions, the alternative to loading data from arbitrary, unaligned addresses is to load individual bytes, one by one. For example, to load a word, we could use a code sequence like this. For loading an unaligned doubleword, the code sequence will be twice as long.

```
    LOADB    r1,0(RegAddr)      # Get MSByte and sign bits
    LOADB    r2,1(RegAddr)      # Get byte 2
    ANDI     r2,r2,0x0ff        # .
    SLL      r1,r1,8            # Shift byte 2 into result
    ORI      r1,r1,r2           # .
    LOADB    r2,2(RegAddr)      # Get byte 3
    ANDI     r2,r2,0x0ff        # .
    SLL      r1,r1,8            # Shift byte 3 into result
    ORI      r1,r1,r2           # .
    LOADB    r2,3(RegAddr)      # Get LSByte
    ANDI     r2,r2,0x0ff        # .
    SLL      r1,r1,8            # Shift LSByte into result
    ORI      r1,r1,r2           # .
```

The hardware implementation of the ALIGN instructions is fairly simple and small. The hardware will require some shifting of bits (no gates), several multiplexors to select which result to use, and the circuitry to sign-extend either a halfword or a word (which might already be present anyway).

The benefit of the ALIGN instructions depends on how much unaligned data we expect to encounter. The KPL language always places all variables, objects, and fields on aligned boundaries, so there will be almost no unaligned data in KPL, unless the programmer decides to do it explicitly with pointers. Occasionally, we will encounter unaligned data from files read in, or data received over the Internet.

`INJECT1H`	*RegD,Reg1,Reg2,Reg3*	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT2H`	*RegD,Reg1,Reg2,Reg3*	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT1W`	*RegD,Reg1,Reg2,Reg3*	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT2W`	*RegD,Reg1,Reg2,Reg3*	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT1D`	*RegD,Reg1,Reg2,Reg3*	RegD ← Reg1; inject Reg2 per addr in Reg3
`INJECT2D`	*RegD,Reg1,Reg2,Reg3*	RegD ← Reg1; inject Reg2 per addr in Reg3

The INJECT instructions support the emulation of memory STOREs in which the destination address is not properly aligned.

The contents of Reg1 are copied to RegD with no shifting. However, some bytes from Reg2 may be injected into the copied data. By “injected”, we mean a byte from Reg2 will replace a byte being copied from Reg1 to RegD.

The value in Reg3 controls which bytes from Reg2 are injected into the data and where in the data they are injected. The least significant bits in Reg3 tell how to shift/combine the values in Reg1 and Reg2 to produce the value stored in RegD.

The INJECT instructions use only the least signifiant bits of Reg3, so the (possibly unaligned) destination address itself can be used. These bits are the final bits of the address and tell how "misaligned" the address is, i.e., how much shifting must be done.

In the case of INJECT1H and INJECT2H, only the least significant bit of Reg3 is used. In the case of INJECT1W and INJECT2W, the least significant 2 bits of Reg3 are used. In the case of INJECT1D and INJECT2D, the least significant 3 bits of Reg3 are used. The remaining bits in Reg3 are ignored.

Details of the INJECT Instructions

Let's look at the operation of the INJECT instructions in more detail, starting with a code sequence to show how they can be used. This example deals with doubleword-sized data; the code for halfword or word data would be virtually identical.

To store a doubleword "source value" into an unaligned memory address, the code must first LOAD two aligned doublewords from memory, then use the source doubleword to modify (i.e., "inject") some bytes into each of these doublewords, then issue two STORE instructions to store the updated doublewords back into memory.

```
# Assume the unaligned address is in r4
# Assume the source data to be stored is in r7
ANDI       r5,r4,0xFFF8  # Compute an aligned address
LOAD.D     r1,0(r5)      # Read two doublewords from memory
LOAD.D     r2,8(r5)      # .
INJECT1D   r1,r1,r7,r4   # Inject bytes into lefthand dword
INJECT2D   r2,r2,r7,r4   # Inject bytes into righthand dword
STORE.D    0(r5),r1      # Store two dwords back to memory
STORE.D    8(r5),r2      # .
```

(It's possible that the address happens to be aligned and, by adding a test and branch, some LOADs and STOREs could be avoided. This optimization is not shown.)

INJECT1D and INJECT2D

The two INJECTD instructions will inject eight bytes in one of eight ways, depending on the unaligned address, as shown next.

Assume that the most significant doubleword fetched from memory is

```
xx xx xx xx xx xx xx xx
```

Assume the least significant doubleword fetched from memory is

```
yy yy yy yy yy yy yy yy
```

Assume that the source value to be stored is

```
AA BB CC DD EE FF GG HH
```

The source doubleword will need to be injected into these two doublewords in one of these 8 ways, where "`..`" means that the byte is unchanged.

```
      xx xx xx xx xx xx xx xx    yy yy yy yy yy yy yy yy

 0    AA BB CC DD EE FF GG HH    .. .. .. .. .. .. .. ..
 1    .. AA BB CC DD EE FF GG    HH .. .. .. .. .. .. ..
 2    .. .. AA BB CC DD EE FF    GG HH .. .. .. .. .. ..
 3    .. .. .. AA BB CC DD EE    FF GG HH .. .. .. .. ..
 4    .. .. .. .. AA BB CC DD    EE FF GG HH .. .. .. ..
 5    .. .. .. .. .. AA BB CC    DD EE FF GG HH .. .. ..
 6    .. .. .. .. .. .. AA BB    CC DD EE FF GG HH .. ..
 7    .. .. .. .. .. .. .. AA    BB CC DD EE FF GG HH ..
```

INJECT1D will perform the injection shown above on the left and INJECT2D will perform the injection shown above on the right.

To be more precise, assume that Reg1 and Reg2 contain the following bytes.

```
Reg1    X1 X2 X3 X4 X5 X6 X7 X8
Reg2    AA BB CC DD EE FF GG HH
```

INJECT1D will move the following values into RegD, based on the least significant 3 bits in Reg3, i.e., the final bits of the unaligned address.

Reg3	Result in RegD
000	AA BB CC DD EE FF GG HH
001	X1 AA BB CC DD EE FF GG
010	X1 X2 AA BB CC DD EE FF
011	X1 X2 X3 AA BB CC DD EE
100	X1 X2 X3 X4 AA BB CC DD
101	X1 X2 X3 X4 X5 AA BB CC
110	X1 X2 X3 X4 X5 X6 AA BB
111	X1 X2 X3 X4 X5 X6 X7 AA

Assume that Reg1 and Reg2 contain the following bytes.

Reg1	Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
Reg2	AA BB CC DD EE FF GG HH

INJECT2D will move the following values into RegD, based on the least significant 3 bits in Reg3, i.e., the final bits of the unaligned address.

Reg3	Result in RegD
000	Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
001	HH Y2 Y3 Y4 Y5 Y6 Y7 Y8
010	GG HH Y3 Y4 Y5 Y6 Y7 Y8
011	FF GG HH Y4 Y5 Y6 Y7 Y8
100	EE FF GG HH Y5 Y6 Y7 Y8
101	DD EE FF GG HH Y6 Y7 Y8
110	CC DD EE FF GG HH Y7 Y8
111	BB CC DD EE FF GG HH Y8

INJECT1W and INJECT2W

When it comes to storing a word into an unaligned address in memory, we make the assumption that it will be implemented in terms of aligned *word* LOADs and STOREs, not *doubleword* LOADs and STOREs.

The two INJECTW instructions will inject four bytes in one of four ways, depending on the unaligned address, as shown next.

Assume that the most significant word fetched from memory is

```
     xx xx xx xx
```

Assume the least significant word fetched from memory is

```
     yy yy yy yy
```

Assume that the source value to be stored is

```
     AA BB CC DD
```

The source word will need to be injected into these two words in one of these 4 ways, where “..” means that the byte is unchanged.

```
       xx xx xx xx    yy yy yy yy

  0    AA BB CC DD    .. .. .. ..
  1    .. AA BB CC    DD .. .. ..
  2    .. .. AA BB    CC DD .. ..
  3    .. .. .. AA    BB CC DD ..
```

Assume that Reg1 and Reg2 contain the following bytes.

```
  Reg1   X1 X2 X3 X4 X5 X6 X7 X8
  Reg2   ss ss ss ss AA BB CC DD
```

where “ss” represents sign-extension bytes that will be ignored.

INJECT1W will move the following values into RegD, based on the least significant 2 bits in Reg3, i.e., the final bits of the unaligned address.

Reg3	Result in RegD
00	X1 X2 X3 X4 AA BB CC DD
01	X1 X2 X3 X4 X5 AA BB CC
10	X1 X2 X3 X4 X5 X6 AA BB
11	X1 X2 X3 X4 X5 X6 X7 AA

Assume that Reg1 and Reg2 contain the following bytes.

```
  Reg1   Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
  Reg2   ss ss ss ss AA BB CC DD
```

INJECT2W will move the following values into RegD, based on the least significant 2 bits in Reg3, i.e., the final bits of the unaligned address.

Reg3	**Result in RegD**
00	`Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8`
01	`Y1 Y2 Y3 Y4 DD Y6 Y7 Y8`
10	`Y1 Y2 Y3 Y4 CC DD Y7 Y8`
11	`Y1 Y2 Y3 Y4 BB CC DD Y8`

INJECT1H and INJECT2H

When it comes to storing a halfword into an unaligned address in memory, we make the assumption that it will be implemented in terms of aligned *halfword* LOADs and STOREs, not *word* or *doubleword* LOADs and STOREs.

The two INJECTH instructions will inject two bytes in one of two ways, depending on the unaligned address, as shown next.

Assume that the most significant halfword fetched from memory is

```
xx xx
```

Assume the least significant halfword fetched from memory is

```
yy yy
```

Assume that the source value to be stored is

```
AA BB
```

The source halfword will need to be injected into these two hafwords in one of these 2 ways, where "`..`" means that the byte is unchanged.

	`xx xx`	`yy yy`
0	`AA BB`	`.. ..`
1	`.. AA`	`BB ..`

Assume that Reg1 and Reg2 contain the following bytes.

Reg1	`X1 X2 X3 X4 X5 X6 X6 X7`
Reg2	`ss ss ss ss ss ss AA BB`

where, "`ss`" is represents sign-extension bytes, which will be ignored.

INJECT1H will move the following values into RegD, based on the least significant bit in Reg3, i.e., the final bit of the unaligned address.

Reg3	**Result in RegD**
0	`X1 X2 X3 X4 X5 X6 AA BB`
1	`X1 X2 X3 X4 X5 X6 X7 AA`

Assume that Reg1 and Reg2 contain the following bytes.

Reg1	`Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8`
Reg2	`ss ss ss ss ss ss AA BB`

INJECT2H will move the following values into RegD, based on the least significant bit in Reg3, i.e., the final bit of the unaligned address.

Reg3	**Result in RegD**
0	`Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8`
1	`Y1 Y2 Y3 Y4 Y5 Y6 BB Y8`

Commentary The assumption made by the INJECTD instructions is that unaligned doubleword operations will be emulated by using aligned doubleword operations. Underlying this approach is the implicit assumption that the implementation naturally supports doubleword memory operations.

However, the implementation may actually support only word-sized operations directly and implement doubleword LOADs and STOREs by breaking each instruction into two memory operations. A LOAD.D instruction will result in two memory reads and a STORE.D instruction will result in two memory writes. Thus, the approach outlined in the code example above (with 2 LOAD.D and 2 STORE.D instructions) will actually result in 4 memory reads and 4 memory writes.

The INJECTD operations will continue to work, but a better solution might be desirable. Note that a doubleword, no matter how it is aligned, can only touch 3 words. To store a doubleword no matter how it is aligned, one only needs to read at most two words and write at most three words.

We also assume that unaligned word operations will be emulated with aligned word operations. However, the implementation may not naturally support word-sized LOADs and STOREs and may actually implement them as doubleword memory operations. This would mean that, for a word sized STORE instruction, the implementation will be reading, injecting, and storing beneath the level of software, similarly to what we are discussing doing in software.

To store 4 bytes with a machine whose natural unit of transfer is 8 bytes, we only require at most 1 read and 1 write in 5/8 of the cases and 2 reads and 2 writes in the remaining 3/8 of the cases. This averages to 1.4 reads and 1.4 writes per operation.

However, naïvely using the scheme suggested in the code snippet earlier would result in duplicate effort and severely impact performance. There are two LOAD.W instructions, resulting in 2 doubleword memory reads, and there are two STORE.W instructions, resulting in 2 memory reads and 2 memory writes. This comes to 4 reads and 2 writes per operation, much worse than the optimal solution.

A better approach would be to emulate an unaligned word operation using aligned doubleword operations. The INJECTW instructions are not designed for this.

Likewise, in the case of halfword data, we have the same issues.

`ILLEGAL` *<no operands>*

Will cause “Illegal Instruction Exception”

This is the canonical illegal instruction. Both OP1 and OP2 values are 0x00. Often, uninstalled main memory will be read as containing all zeros. Thus, when fetched, an instruction of 0x0000_0000 will be interpreted as an illegal instruction, preventing the execution of uninstalled memory.

Uninstalled main memory may also be read as all 1 bits. An instruction 0xFFFF_FFFF will be interpreted as a pair of compressed instructions. For this reason, the pattern 0xFFFF will be interpreted as the compressed form of an illegal instruction.

This instruction will cause an “Illegal Instruction Exception”.

SYSRET	*<no operands>*	

Privileged

This instruction is used to return from a trap handler. It performs the following operations:

PC = **csr_prevpc**
csr_status = **csr_stat2**

In other architectures, this instruction is often named “RETI”.

Commentary Interrupts are disabled by the hardware whenever trap handlers are invoked and they generally remain disabled throughout the trap handler code. Interrupts should always be disabled at the time of the SYSRET at the end of the trap handler. Here’s why.

Note that the SYSRET instruction uses the CSRs — **csr_stat2** and **csr_prevpc** in particular. If interrupts happen to be enabled at the time the SYSRET instruction is executed, there is a possibility that an interrupt might occur directly before the SYSRET instruction.

Trap handlers save the state of the general purpose registers, but the state of the CSRs is not saved. As part of all trap invocation, the hardware will overwrite **csr_stat2** and **csr_prevpc** and their previous contents will be lost. Thus, if an interrupt might occur directly before a SYSRET, upon return after the interrupt processing, the SYSRET could not possibly function correctly.

Thus, it is always a kernel bug to execute a SYSRET with interrupts enabled. However, the hardware does nothing to enforce this.

SLEEP1	*<no operands>*	Enable interrupts; enter light sleep state
SLEEP2	*<no operands>*	Enable interrupts; enter deep sleep state

Privileged

These are the “wait” instructions, which put the core to sleep until the next interrupt occurs. There are two levels of sleep state. “Light sleep” is intended to make wake-up faster. “Deep sleep” is intended to be a power saving state, which may require more time and effort to recover from.

In both cases, the following are preserved:

- General Purpose Registers
- CSR registers, PC
- Main Memory

The sleep state ends when an interrupt occurs. During the sleep state, the Program Counter (PC) points to the instruction following the SLEEP instruction so, after the interrupt trap handler returns, instruction execution will resume with the instruction following the SLEEP instruction.

These instructions will enable interrupts before entering the sleep state.

These instructions should only be executed with interrupts disabled, for the following reason. We only want to sleep when there are no runnable threads and the only way to know that is to check first, before going to sleep. But an interrupt might occur at any time (including directly before the SLEEP instruction is executed) and this may cause some new thread to become runnable. To prevent going to sleep when runnable threads exist, the software should disable interrupts, check to make sure it is safe to sleep, then execute the SLEEP instruction. If interrupts have become pending, then the sleep state will end immediately and the interrupt trap handler will be invoked.

In some implementations, there will be no difference between “light” and “deep” sleeping. Thus, the instructions may function identically. A valid implementation is to act as a sort of no-op, doing nothing more than enabling interrupts. In more complex implementations, SLEEP1 and SLEEP2 may differ as follows: In the light sleep state, the clock continues, **csr_cycle** is constantly incremented, and timer interrupts occur. In the deep sleep state, **csr_cycle** is not incremented and therefore timer interrupts do not occur; effectively, the clock is turned off.

In the emulator, SLEEP2 will cause an immediate halt to emulation. If the emulator is executing in auto-go mode (command line option -g), the emulator will terminate and the value in register r1 will be returned as the Unix/Linux exit code (where 0=ok/no error). This is useful for KPL programs that are to be run under Unix/Linux.

`RESTART` *<no operands>* Same as Power-On-Reset

Privileged

The purpose of this instruction is to cause a full reboot of the system. This instruction will have the same effect as cycling the power on the processor, namely:

The following two registers will be set to their initial values:

csr_inst ← 0x0000_0000_0000_0000
csr_cycle ← 0x0000_0000_0000_0000

csr_status ← 0x0000_0000_0000_0001
Program Counter (PC) ← 0x4_0000_0000

This means that the following conditions will be true:

Kernel Mode: Enabled
Interrupts: Disabled

The PC is set to the first word of the memory-mapped I/O area, which is where the "**Boot ROM**" is located.

Any pending interrupts are cleared. All memory-mapped I/O devices are sent a "reset" signal and will go into their initial states. In particular, the Secure Storage will be reset and will be writable.

All other other programmer-visible state of the core (i.e., the general purpose registers and all other CSRs) will have undefined values.

In a multi-core processor, this instruction will affect all cores. The execution of this instruction by any one core will instantly kill all cores, which will all be restarted.

```
DEBUG           <no operands>
BREAKPOINT      <no operands>
```

Will cause "Debug/Breakpoint Exception"

These instructions are used by the debugger. Each instruction will cause an exception and there is a corresponding exception type for each:

Debug Exception
Breakpoint Exception

It is intended that the DEBUG instruction will be inserted into code by the programmer. When executed, the resulting exception will be used to invoke and start up the debugger. This will allow the user to begin debugging his/her code.

It is intended that the BREAKPOINT instruction will be inserted by a debugging tool into the code being debugged.

Typically, the user of the debugger will command the debugger to insert a breakpoint a some point in the code being debugged. The debugger will replace the instruction at the the target address with a BREAKPOINT instruction. Then later, after execution is resumed and execution reaches the target address, the BREAKPOINT instruction will be encountered. The resulting exception allows the debugger to regain control. Typically, the debugger will save the instruction that was replaced and, when the BREAKPOINT is removed, the instruction will be restored.

These two instructions are almost identical, except (1) they each cause a different exception, and (2) the value stored in **csr_prevpc** at the time of the exception is different.[6]

These instructions are both Format-A instructions. The opcode occupies the first 16 bits of the instruction and the remaining 16 bits contain space for register fields. However, since no registers are used, these 16 bits are unused and shall be ignored

[6] DEBUG stores the address of the following instruction, while BREAKPOINT stores the address of the BREAKPOINT instruction itself.

by the ISA. Bits [15:0] in the instruction are therefore available for use by a debugger, to store additional information.[7]

If the Blitz-64 architecture is being emulated on a virtual machine, the DEBUG and BREAKPOINT instructions may be unimplemented. In other words, they will not cause exceptions. Instead, when encountered, they will be used by a debugger that is built in to the emulator itself.

These instructions are primarily expected to be used to debug code running in user mode. However, they might also to debug code running in kernel mode.

`SYSCALL` *immed10*

Will cause "SYSCALL Exception"

The SYSCALL instruction is used by user mode code to invoke one of the 1,024 system calls.

This instruction causes a "SYSCALL Exception". More precisely, this instruction will perform the following actions:

csr_stat2 = **csr_status**
csr_prevpc = PC
csr_cause = immed10 × 8

It will then initiate trap processing by performing these actions.

csr_status[KERNEL_MODE] = 1
csr_status[INTERRUPTS_ENABLED] = 0
csr_status[SINGLE_STEP] = 0
PC = **csr_trapvec** (the address of the global trap handler)

The immediate value gives a number in the range 0 … 1,023. This number is shifted left by 3 bits (i.e., multiplied by 8) and passed to the trap handler for use in

[7] For example, when the debugger sets a break point, it will replace some instruction by a BREAKPOINT instruction. The debugger will need to remember which instruction was removed so that when the break point is encountered it can replace the BREAKPOINT instruction with the saved instruction. There may be a number of break points set and the debugger might use the 16 bits as an index into some record-keeping table it maintains.

dispatching to the correct kernel function. The immediate value is not sign-extended.

The PC value copied to **csr_prevpc** is the address of the instruction after the SYSCALL instruction, not the SYSCALL itself.

Note that this instruction is normally executed in user mode. Whether execution of the SYSCALL instruction in kernel mode is a bug or not is a software design decision; however, the instruction will still function as described.

```
CONTROL     RegD,Reg1,immed16
CONTROLU    RegD,Reg1,immed16
```

CONTROL: Privileged; CONTROLU: Not privileged

The definition of these instructions is left unspecified here and is completely implementation dependent.

The idea is that an implementation of the Blitz-64 architecture is free to use these instructions whenever it is necessary to supplement the instruction set with instructions not included in the ISA specification.

A specific implementation may need to add a large number of instructions to the ISA. The immediate value is available to act as a sort of additional op-code. The idea is that different values of immed16 will invoke different behaviors.

These instructions may or may not access registers Reg1 and RegD. They may also access other registers not directly mentioned; everything is left to the specific implementation designers.

We really want all User Mode programs to be fully portable between Blitz-64 implementations. To this end, we make a distinction between the CONTROL and CONTROLU instructions.

Implementation-dependent behavior really ought to be encapsulated within the Kernel. Otherwise, a User Mode program that used the instruction would be tied to a specific implementation. To support this, the CONTROL instruction is a privileged instruction and an Illegal Instruction Exception will be raised if this instruction is executed in User Mode.

On the other hand, some operations need to be usable in User Mode. For this, we provide the CONTROLU instruction, which may be executed in User Mode. Note that any program using a CONTROLU instruction will be implementation dependent and may have completely unexpected results when executed on a different Blitz-64 processor.

The implementation is free to define whether these instructions might raise other exceptions. If the CONTROL or CONTROLU instruction is used incorrectly (for example, with an undefined immed16 value), the implementation really ought to raise an Illegal Instruction Exception.

Commentary The memory-mapped I/O regions in the Blitz-64 architecture are designed so they can be selectively mapped into the virtual address spaces of user processes. However, the CONTROL/CONTROLU mechanism does not have this flexibility.

Whether additional functionality is added to the Blitz-64 architecture using CONTROL/CONTROLU or by adding a new memory-mapped I/O region is an engineering decision left to implementors.

Example Uses for CONTROL and CONTROLU

How might a Blitz-64 implementation use the CONTROL instruction? Let's look at several examples.

Digital I/O Pins Imagine that the Blitz-64 chip has a number of digital I/O pins. This might occur for a Blitz-64 processor used in an Arduino-like setting.

For such a system, the CONTROL instruction will be defined to write values to OUTPUT pins based on the contents of register Reg1. The instruction will also be used to read in values on INPUT pins to register RegD. Each operation will both read and the write the I/O pins simultaneously.

If this is the only use of the CONTROL instruction, the immed16 value can be ignored:

```
    controlu   r7,r1,0    # Read inputs and change outputs
```

In this case, the COLTROLU instruction was used, which means that user-mode programs can control the digital pins directly, without needing kernel intervention.

A slightly different implementation might be to separate the input and output operations, using the immed16 value to distinguish between “read” and “write” operations:

```
DIGITAL_READ:    .equ    0x0001
DIGITAL_WRITE:   .equ    0x0002
```

To read and write the digital I/O pins, the operations would look like this:

```
    controlu   r7,r0,DIGITAL_READ    # sample the inputs
    controlu   r0,r1,DIGITAL_WRITE   # update the outputs
```

LED Control LEDs are helpful for single-board computers. For example, as the system boots, a green LEDs might turn on. If error conditions arise, the core can turn on a red LED to signal that it is unhappy. In order to drive such LEDs, each chip will need a couple of output pins dedicated to these LEDs.

Such “status LEDs” are cheap and ought to be included in every single-board computer.

In such a system design, the CONTROL instruction could be used to control the status LEDs.[8]

```
RED_LED_ON:          .equ       0x0004
RED_LED_OFF:         .equ       0x0008
GREEN_LED_ON:        .equ       0x0010
GREEN_LED_OFF:       .equ       0x0018
BLUE_LED_ON:         .equ       0x0020
BLUE_LED_OFF:        .equ       0x0028
```

To manipulate specific LEDs, the core can execute an instruction such as:

```
    control    r0,r0,GREEN_LED_ON | RED_LED_OFF
```

[8] Note that we are defining the immediate values so that there is no overlap with the values used to control the digital pins, so both could be used within the same system.

Imagine a system with a large number of Blitz-64 cores, perhaps with hundreds of processor blades mounted in racks, comprising a giant multi-processor system. LEDS for each board (or processor or core) could be very helpful in detecting and understanding faults.

Flushing Caches The Blitz-64 instruction set includes a single instruction (namely FENCE) that has the side-effect of flushing caches, if they exist. However the FENCE instruction may be too course-grained for some system designs. In order to add the ability to flush individual caches separate, the CONTROL instruction could be employed.

If, for example, the implementation needed several different "flush" operations, then some bits in the immed16 field could be defined to indicate which cache flush operation is intended. Perhaps the implementation has several caches:

```
FLUSH_CACHE_L1_I:    .equ      0x0001
FLUSH_CACHE_L1_D:    .equ      0x0002
FLUSH_CACHE_L2:      .equ      0x0004
FLUSH_CACHE_L3:      .equ      0x0008
```

Then, to perform a cache-flush operation, the OS kernel might execute an instruction such as:

```
    control       r0,r0,FLUSH_CACHE_L1_D
```

Flushing the caches does not involve registers, so r0 is specified for both source and destination.

Encryption Support A common but time-consuming operation is to encode and decode encrypted messages. A related function is computing message digests. The algorithms are computationally intensive but it is desirable to perform these operations quickly.

The document *"Blitz-64: Memory-Mapped I/O Devices"* discusses using the DMA memory-mapped I/O device to support SHA-256 and AES, but in some implementations it may make sense to implement other, similar algorithms in hardware, using the CONTROL instruction to access this special-purpose hardware.

Typically, such algorithms involve the simultaneous manipulation of a number of variables. For example, in the SHA-2 algorithm there are 8 variables, named a, b, c, …,

h. During one iteration, all 8 variables are used as inputs to compute 8 new values for the next iteration.

Of course the function computed in each iteration can be done with existing Blitz-64 instructions, but it will take quite a few instructions for each iteration of the loop. To support such an algorithm, the idea is that the entire loop body will be implemented using a single new "instruction". Imagine that an implementation decides to add a single instruction which will perform the entire loop body computation in one step.

Such a proposed new instruction will need 8 inputs and 8 outputs. Consider SHA-256 which is a specific example of the SHA-2 class of algorithms. It uses eight variables, each of 32 bits. We can pack these into four registers. In this hypothetical implementation design, we will assign variables a, b, c, …, h to registers r1, r2, r3, and r4. The new, hypothetical CONTROL instruction, which we are suggesting here, will ignore the RegD and Reg1 specifiers in the instruction and will always operate on registers r1-r4.

There is no reason that such encryption operations can't be done in User Mode, so for these operations, it makes sense to use the CONTROLU instruction, instead of the CONTROL instruction, which must be executed in Kernel Mode.

Additional Floating Point Operations The Blitz ISA only requires support for double precision floats. It might be desirable to provide support for single precision or quad precision floats in some systems. Likewise, there might be special-purpose numerical engines (e.g., neural net or graphic engines). This might be accommodated with CONTROL instructions.

Accessing the Micro-architecture Another possibility is that the CONTROL instruction would be defined to access or modify internal core state. For example, the CONTROL instruction might be used to read pipeline registers that are otherwise invisible to the ISA.

Flexibility Given that immediate field has 16 bits, the CONTROL instruction framework can be employed to add many unique instructions and behaviors. The immed16 field can be considered as a sort of secondary opcode and a large number (up to 65,536) of additional implementation-dependent instructions can be added to any core using the CONTROL instruction framework.

`TLBCLEAR`	*<no operands>*	Invalidate all TLBs for current ASID
`TLBFLUSH`	*Reg1*	Invalidate TLB for virtual address in Reg1

Privileged

The TLBCLEAR instruction invalidates all TLB registers that apply to the Address Space ID (ASID) in **csr_pgtable**. While the format of the TLB registers is implementation dependent, it is assumed that each TLB register will contain a "valid" bit. This instruction will clear this bit for all registers with ASIDs matching the current ASID.

The TLBCLEAR should be executed after any change to the page table for a specific address space. This will force all subsequent FETCHes and LOAD, STORE, and CAS instructions to this virtual address space to trigger a walk of the new page table.

The TLBFLUSH instruction expects Reg1 to contain a virtual address. If the TLB contains a register with a matching virtual address and an ASID matching the current ASID (in **csr_pgtable**), then the "valid" bit for that TLB register will be cleared to 0. The address in Reg1 need not be page aligned; only bits [35:14] are used.

If the address is a physical address (i.e., bit [35] is 0) or does not match any TLB register, TLBFLUSH does nothing. If the system does not contain TLBs, these instructions do nothing.

Note

In a multi-core system, the TLBCLEAR and TLBFLUSH instructions affect only the TLB registers on the core executing the instruction. There is a potential a problem when one virtual address space is shared across multiple cores and the kernel running on one core wishes to alter the address space and will execute one of these instructions to eliminate out-of-date information in the TLB registers. A change in the current specification is contemplated.

CHECKADDR *RegD,Reg1,immed3* Reg1 = virt addr; RegD ← except. code or 0

Privileged

The CHECKADDR instruction requires an immediate value, which should be one of the following values.

immed3	Access Type
0	LOAD.B
1	LOAD.H
2	LOAD.W
3	LOAD.D
4	STORE.B
5	STORE.H
6	STORE.W
7	STORE.D

(Only the least significant 3 bits of the immediate 16 bit value are used; bits [15:3] are ignored.)

Register Reg1 will contain an address, which may be physical or virtual. The CHECKADDR instruction determines what would happen if an instruction of the indicated type were to be executed using that address. CHECKADDR will store the following code in register RegD.

Result	Outcome
0	Success; no exception
1	Null Address Exception
2	Unaligned LOAD/STORE Exception
3	Page Illegal Address Exception
4	Page Table Exception
5	Page Invalid Exception
6	Page Write Exception
7	Page Copy-On-Write Exception
8	Page First Dirty Exception

The hypothetical access is assumed to be performed in USER MODE, not KERNEL MODE.

From time to time, a system call will be handled by kernel code. The user code will pass a virtual address to the kernel. For example, the kernel may wish to retrieve argument data from the user's virtual address space or move result data into the user's virtual address space.

Assuming **csr_pgtable** has been previously set, the kernel can simply use normal LOAD and STORE instructions with virtual addresses to retrieve data from or store data into the virtual address space.

Of course the kernel cannot trust any address provided by user code. Executing a LOAD or STORE instruction might cause an exception.

The CHECKADDR instruction is provided so that the kernel can check (before the LOAD or STORE operation is attempted) whether such an access would result in an exception.

More Detail

If CHECKADDR indicates one of the following exceptions, the address is in error. If passed from user code to the kernel, the kernel should not attempt to use the address in a LOAD or STORE operation.

Null Address Exception
Unaligned LOAD/STORE Exception
Page Illegal Address Exception
Page Write Exception

The following exception will probably never to occur, since we can assume that the kernel has, at some earlier time, set **csr_pgtable** correctly:

Page Table Exception

The following exception types may occur in correct user mode code. Normally, they would be serviced and the instruction re-tried. When CHECKADDR indicates this sort of exception, some additional work may be required of the kernel before it can perform the access to the virtual address space.

Page Copy-On-Write Exception
Page First Dirty Exception

The following exception could be caused by a bad address, i.e., pointing to some region of the virtual address space that is unallocated. Alternately, it might also point to a region of the address space that is "allocate-on-demand", and thus be perfectly legitimate.

Page Invalid Exception

The following exception type cannot be returned by the CHECKADDR instruction:

Page Fetch Exception

It may be that a hardware fault occurs during the CHECKADDR instruction (or would occur during a LOAD or STORE operation). The CHECKADDR instruction does not report such a situation; instead the Hardware Fault Exception will simply occur.

If the implementation uses TLB registers, the CHECKADDR instruction may alter them.

`CSRSWAP`	*RegD,CSRReg1,Reg2*	RegD ← CSR; CSR ← Reg2
`CSRREAD`	*RegD,CSRReg1*	Reg1 encodes CSR; RegD ← CSR
`CSRSET`	*CSRReg1,immed16*	Set selected bits in CSR
`CSRCLR`	*CSRReg1,immed16*	Clear selected bits in CSR

Privileged

These instructions each access one of the 16 CSR registers. The identity of the CSR is encoded using 4 bits in the Reg1 field.

The CSRSWAP instruction performs both a read and a write operation. If RegD and Reg2 indicate the same register, the value in that register is swapped with the value in the CSR register.

The CSRREAD instruction reads a CSR and moves it into a general purpose register.

For CSRSET and CSRCLR, the immediate field is sign-extended and forms a 64 bit mask. Wherever there is a 1 bit in the mask, the corresponding bit in the CSR is either set to 1 or cleared to 0.

It is the assembly programmer’s or compiler’s responsibility to ensure that the immediate value is within the range -32,768 … 32,767 (0x8000 … 0x7FFF). If the value is out of range, the assembler will issue an error message. This is not of great concern, since none of the CSRs have individual bit fields, except in the least significant bits.

(But note that if the immediate value is negative, the assembler will not issue an error and the mask will include 1 bits in the upper 48 bits, which may not be what is intended. For example, CSRSET csrReg,0x8000 will set all bits in the register except the least significant 15 bits, while CSRCLR csrReg,-4 will clear all bits except the least significant 2 bits.)

`*CSRWRITE`	*CSRReg1,Reg2*	Reg1 encodes CSR; CSR ← Reg2

Synthetic, Privileged

Register t Usage: Not used; Okay to use as Reg2.

The CSRWRITE instruction is implemented as.

```
        CSRSWAP         r0,CSRReg1,Reg2
```

`GETSTAT`	*RegD*	RegD ← CSR_STATUS & 0x00000000000003f8
`PUTSTAT`	*Reg1*	CSR_STATUS [9:3] ← Reg1 [9:3]

These instructions read and write the portions of the CSR_STATUS register that are visible to User Mode code.

GETSTAT will only return bits that should be visible to User Mode code; all other bits will be masked and returned as 0. PUTSTAT will only modify bits that are modifiable by User Mode code.

The bits that can be read and written are the FLOAT_ROUND bits (i.e., [9:8]) and the FLOAT_STATUS bits (i.e., [7:3]).

Even though these instructions access a CSR register, they are not privileged.

`FADD`	*RegD,Reg1,Reg2*	RegD ← Reg1 + Reg2
`FSUB`	*RegD,Reg1,Reg2*	RegD ← Reg1 - Reg2
`FMUL`	*RegD,Reg1,Reg2*	RegD ← Reg1 × Reg2
`FDIV`	*RegD,Reg1,Reg2*	RegD ← Reg1 / Reg2
`FMIN`	*RegD,Reg1,Reg2*	RegD ← MIN (Reg1, Reg2)
`FMAX`	*RegD,Reg1,Reg2*	RegD ← MAX (Reg1, Reg2)
`FNEG`	*RegD,Reg1*	RegD ← -Reg1
`FABS`	*RegD,Reg1*	RegD ← ABSOLUTE_VALUE (Reg1)
`FSQRT`	*RegD,Reg1*	RegD ← SQUARE_ROOT (Reg1)
`FEQ`	*RegD,Reg1,Reg2*	RegD ← (Reg1 = Reg2) ? 1 : 0 (float compare)
`FLT`	*RegD,Reg1,Reg2*	RegD ← (Reg1 < Reg2) ? 1 : 0 (float compare)
`FLE`	*RegD,Reg1,Reg2*	RegD ← (Reg1 ≤ Reg2) ? 1 : 0 (float compare)
`FCVTFI`	*RegD,Reg1*	Convert: floating-point ← int
`FCVTIF`	*RegD,Reg1*	Convert: int ← floating-point
`FMADD`	*RegD,Reg1,Reg2,Reg3*	RegD ← (Reg1 × Reg2) + Reg3
`FNMADD`	*RegD,Reg1,Reg2,Reg3*	RegD ← (-(Reg1 × Reg2)) + Reg3
`FMSUB`	*RegD,Reg1,Reg2,Reg3*	RegD ← (Reg1 × Reg2) - Reg3
`FNMSUB`	*RegD,Reg1,Reg2,Reg3*	RegD ← (-(Reg1 × Reg2)) - Reg3

May cause an "Emulated Instruction Exception"

The comments above describe the computations performed by these instructions.

All arithmetic is performed in double precision floating point, per the IEEE 754 standard. The FLOAT_STATUS bits in **csr_status** are set as required.

With Blitz, all rounding is "to nearest, with ties to even". The FLOAT_ROUND bits in **csr_status** are ignored.[9]

Note that there are no FMOV, FLOAD, or FSTORE instructions. The instructions MOV, LOADx, and STOREx will work fine.

The test performed by FEQ is not the same as BEQ, due to facts like "+0.0 = -0.0" and "NaN ≠ NaN". Programmers should note that equality testing of floating point values is especially risky, due to rounding errors.

[9] At least in this version of the Blitz ISA; if the need should ever arise, this decision could be revisited.

The conversion instructions (FCVTFI and FCVTIF) are discussed below.

The floating point instructions are candidates for emulation. Any attempt to execute an unimplemented instruction will result in an “Emulation Exception”.

Commentary The IEEE 754 specification requires single precision floating point to be implemented whenever double precision is implemented. Blitz-64 does not implement single precision floating point, as a conscious design decision. Therefore Blitz-64 clearly does not conform to the IEEE 754 spec.

That said, Blitz-64 “respects” and “follows" the IEEE 754 floating point specification.

IEEE 754 is a complex specification and floating point math is a can of wriggling worms. The Blitz-64 architecture intends and attempts to conform precisely and accurately to the IEEE spec.

To be honest, floating point is a bit out of my primary research expertise and I’d really appreciate your help. If you see violations or other issues, you are encouraged to speak up and email me.

FCVTIF The FCVTIF instruction converts a double precision floating point number into a 64 bit signed integer with about the same value.

If the value to be converted is **NaN**, the instruction will set the **NV-Invalid** flag in the **CSR_STATUS** register. The integer result will be “0”.

If the value is **+inf**, the result will be 0x7FFF_FFFF_FFFF_FFFF and the **OF-Overflow** and **NX-Inexact** flags will be set. If the value is **-inf**, the result will be 0x8000_0000_0000_0000 and the **OF-Overflow** and **NX-Inexact** flags will be set.

Concerning overflow, here are the values around the largest signed integer (0x7FFF_FFFF_FFFF_FFFF = 9,223,372,036,854,775,807) that can be represented exactly with double precision floats:

+9,223,372,036,854,774,784.0
+9,223,372,036,854,775,808.0
+9,223,372,036,854,777,856.0

If the floating value to be converted is greater than +9,223,372,036,854,775,807.0 (including **+inf**) then the **OF-Overflow** and **NX-Inexact** bits will be set and the result will be +9,223,372,036,854,775,807 (i.e., 0x7FFF_FFFF_FFFF_FFFF).

If the floating value is more negative than -9,223,372,036,854,775,808.0 (including **-inf**) then the **OF-Overflow** and **NX-Inexact** bits will be set and -9,223,372,036,854,775,808 (i.e., 0x8000_0000_0000_0000) will be used.

If the floating point value is not an integer (i.e., if it has non-zero digits to the right of the decimal point, as in 4.5) then the **NX-Inexact** bit will be set and the value will be rounded to the nearest integer, with ties to even.

The **UF-Underflow** and **DZ-Divide-by-zero** bits in **FLOAT_STATUS** will be unchanged.

<u>FCVTFI</u> The FCVTFI instruction converts a 64 bit signed integer into a double precision floating point number with about the same value.

All integers within the following range can be represented exactly in double precision floating point:

> -9,007,199,254,740,992 … +9,007,199,254,740,992

In hex, this range is:

> 0xFFE0_0000_0000_0000 … 0x0020_0000_0000_0000

Some integers outside this range can be represented exactly, but most cannot be. If the integer cannot be represented exactly, then the value will be rounded to the nearest integer that can be represented, with ties to even, and the **NX-Inexact** flag in **FLOAT_STATUS** will be set.

There are no error or overflow conditions, so no other bits in **FLOAT_STATUS** will be affected.

`*FGT`	*RegD,Reg1,Reg2*	RegD ← (Reg1 > Reg2) ? 1 : 0 (float compare)
`*FGE`	*RegD,Reg1,Reg2*	RegD ← (Reg1 ≥ Reg2) ? 1 : 0 (float compare)

<u>Synthetic</u>

<u>Register t Usage:</u> Not used; Okay to use as RegD, Reg1, and/or Reg2.

The FGT instruction is implemented as:
`FLT` *RegD,Reg2,Reg1* Note the reversal of the registers
The FGE instruction is implemented as:
`FLE` *RegD,Reg2,Reg1* Note the reversal of the registers

<u>Commentary</u> Blitz-64 includes floating point instructions because there are applications which require this functionality. We include only double precision because (1) it fits the 64 bit size of the registers and (2) because it provides more precision than single precision. Presumably, double precision can be substituted in applications that require single precision, but not vice versa. If there is only one precision (to keep the architecture simple), it seems that double precision is a better choice.

However, we make no great effort to design an architecture for high-performance floating point computation. Applications that do lots of floating point calculations and are dependent on floating point performance benefit from the sort of vector architecture and parallelism that are commonplace in special purpose hardware, like graphics coprocessors, neural net accelerators, etc. This is really where floating point calculations should be done, not in a general purpose core.

<u>Commentary</u> Whenever one or both of the arguments of FEQ, FLT, FLE, FGT, or FGE is not-a-number (NaN), the result is false. This comes from the IEEE 754 spec.

By true and false, we mean that either 1 or 0 is placed in the result register.

Whenever one or both of the arguments to FEQ is not-a-number (NaN), the result is false (0). Surprisingly, this means that when asking "NaN == NaN?", that answer is "no"!

The definitions of "not equal" is "NOT(equal)". Consequently, this means that when asking "NaN ≠ NaN?", that answer is "yes"!

This happens regardless of the exact bit patterns used to represent NaN.

We have not included an "FNE" instruction, but we could easily have included a synthetic instruction, which would be translated to:

```
FEQ        RegD,Reg2,Reg1      Store 0 if not equal
TESTEQZ    RegD,RegD           Change 0 to 1
```

But note that a test is typically followed by a branch, as in:

```
FNE        RegD,Reg2,Reg1
BNEZ       RegD,Label          Branch if not equal, i.e., if RegD==1
```

This would expand to:

```
FEQ        RegD,Reg2,Reg1
TESTEQZ    RegD,RegD
BNEZ       RegD,Label
```

By not including an FNE instruction, we encourage the compiler writer to generate the following superior code sequence:

```
FEQ        RegD,Reg2,Reg1
BEQZ       RegD,Label          Instead, branch test is reversed
```

Instruction Opcodes

This list is provisional and subject to change in future versions.

The Blitz-64 instruction encoding allows up to 256 Format-A instructions (including ILLEGAL) and up to 63 instructions in other formats.

Max Number of Format-A Instructions	**256**
Max Number of non-Format-A Instructions	**63**
Range of possible OP2 values	**0 … 255**
Range of possible OP1 values	**1 … 63**

Currently there are…

Number of Format-A Instructions	**80**
Number of non-Format-A Instructions	**42**
Total Number of Machine Instructions	**120**
Current range of OP2 values[10]	**0 … 79**
Current range of OP1 values	**1 … 42**

Commentary The process of decoding machine opcodes involves circuitry that will transform the OP1 and OP2 fields (i.e., bits [31:16]) into a collection of control signals.

The opcode assignment given here is done without any attempt to make instruction decoding via combinational circuitry easier. Instead, we assume that all opcodes are decoded using lookup tables.

The number of OP1 values is less than 64 and the number of OP2 values is also less than 64. Therefore, decoding the can be done with two lookup tables, each with no more than 64 entries, along with a multiplexor to differentiate Format A instructions (with OP1=00000000) from the others.

In all “Format A” instructions, the value of OP1 is 0x00; to avoid clutter in the list below, OP1 is not shown.

[10] There may be some gaps in the numbering, as a result of previously deleted instructions. These counts count the ILLEGAL instruction.

Illegal

OP1		OP2			
hex	dec	hex	dec	format	
		00	0	A	ILLEGAL

Arithmetic

OP1		OP2			
hex	dec	hex	dec	format	
		01	1	A	ADD
01	1			B	ADDI
		02	2	A	ADDOK
		03	3	A	ADD3
		04	4	A	SUB
		05	5	A	MULADD
		06	6	A	MULADDU
		07	7	A	DIV
		08	8	A	REM

Logical

OP1		OP2			
hex	dec	hex	dec	format	
		09	9	A	AND
02	2			B	ANDI
		0A	10	A	OR
03	3			B	ORI
		0B	11	A	XOR
04	4			B	XORI

Shift

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
		0C	12	A	SLL
05	5			B	SLLI
		0D	13	A	SLA
06	6			B	SLAI
		0E	14	A	SRL
07	7			B	SRLI
		0F	15	A	SRA
08	8			B	SRAI
		10	16	A	ROTR
09	9			B	ROTRI

Sign Extension

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
		11	17	A	SEXTB
		12	18	A	SEXTH
		13	19	A	SEXTW

Range Checking

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
		14	20	A	NULLTREST
		15	21	A	CHECKB
		16	22	A	CHECKH
		17	23	A	CHECKW
		18	24	A	INDEX0
		19	25	A	INDEX1
		1A	26	A	INDEX2
		1B	27	A	INDEX4
		1C	28	A	INDEX8
		1D	29	A	INDEX16
		1E	30	A	INDEX24
		1F	31	A	INDEX32

Byte Reordering

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
		20	32	A	ENDIANH
		21	33	A	ENDIANW
		22	34	A	ENDIAND

Test and Set a Boolean

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
		23	35	A	TESTEQ
		24	36	A	TESTNE
		25	37	A	TESTLT
		26	38	A	TESTLE
0A	10			B	TESTEQI
0B	11			B	TESTNEI
0C	12			B	TESTLTI
0D	13			B	TESTLEI
0E	14			B	TESTGTI
0F	15			B	TESTGEI

Branch

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
10	16			C	B.EQ
11	17			C	B.NE
12	18			C	B.LT
13	19			C	B.LE

Larger Addresses

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
14	20			D	UPPER20
15	21			B	UPPER16
16	22			B	SHIFT16
17	23			D	ADDPC
18	24			D	AUIPC

Jump

OP1		OP2			
hex	dec	hex	dec	format	
19	25			D	JAL
1A	26			B	JALR

Load & Store

OP1		OP2			
hex	dec	hex	dec	format	
1B	27			B	LOAD.B
1C	28			B	LOAD.H
1D	29			B	LOAD.W
1E	30			B	LOAD.D
1F	31			C	STORE.B
20	32			C	STORE.H
21	33			C	STORE.W
22	34			C	STORE.D

Align

OP1		OP2			
hex	dec	hex	dec	format	
		27	39	A	ALIGNH
		28	40	A	ALIGNW
		29	41	A	ALIGND
		2A	42	A	INJECT1H
		2B	43	A	INJECT2H
		2C	44	A	INJECT1W
		2D	45	A	INJECT2W
		2E	46	A	INJECT1D
		2F	47	A	INJECT2D

Miscellaneous

OP1		OP2				
hex	dec	hex	dec	format		
23	35			B	SYSCALL	
		30	48	A	SYSRET	
		31	49	A	SLEEP1	
		32	50	A	SLEEP2	
		33	51	A	RESTART	
		34	52	A	DEBUG	
		35	53	A	BREAKPOINT	
24	36			B	CONTROL	
25	37			B	CONTROLU	
		36	54	A	CAS	
		37	55	A	FENCE	
29	41			B	ENTERFUN	*(Note that OP1 is out of order)*
2A	42			B	EXITFUN	

CSR Manipulation

OP1		OP2			
hex	dec	hex	dec	format	
		38	56	A	CSRSWAP
		39	57	A	CSRREAD
26	38			B	CSRSET
27	39			B	CSRCLR
		3A	58	A	GETSTAT
		3B	59	A	PUTSTAT

Memory Management Unit

OP1		OP2			
hex	dec	hex	dec	format	
		3C	60	A	TLBCLEAR
		3D	61	A	TLBFLUSH
28	40			B	CHECKADDR

Floating Point

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
		3E	62	A	FADD
		3F	63	A	FSUB
		40	64	A	FMUL
		41	65	A	FDIV
		42	66	A	FMIN
		43	67	A	FMAX
		44	68	A	FNEG
		45	69	A	FABS
		46	70	A	FSQRT
		47	71	A	FEQ
		48	72	A	FLT
		49	73	A	FLE
		4A	74	A	FCVTFI
		4B	75	A	FCVTIF
		4C	76	A	FMADD
		4D	77	A	FNMADD
		4E	78	A	FMSUB
		4F	79	A	FNMSUB

Unused Opcodes

OP1		OP2			
hex	**dec**	**hex**	**dec**	**format**	
2B	43			-	(Next unused OP1)
		50	80	A	(Next unused OP2)

Miscellaneous Remarks

Commentary: The COPY and CLEAR Instructions

It is useful to be able to copy bytes quickly, or to clear large blocks of memory to zero. For example, an operating system must be careful to initialize newly allocated memory pages, in order to prevent data leakage from one address space into another. The OS will also need the ability to copy pages quickly, whenever the "copy-on-write" technique is used. Plus, there just seems to be a lot of data copying, no matter how much programmers try to eliminate it.

We considered adding the following instructions, but did not.

```
COPY  Reg1,Reg2,Reg3              This is not a Blitz-64 instruction!
      while Reg3>0 repeat:
          *(Reg1++) ← *(Reg2++)   [ 8 bytes ]
          Reg3--

CLEAR  Reg1,Reg2,Reg3             This is not a Blitz-64 instruction!
      while Reg3>0 repeat:
          *(Reg1++) ← Reg2    [ 8 bytes ]
          Reg3--
```

There are several problems with these instructions. First, these instructions require many clock cycles to execute and this doesn't fit within the RISC philosophy. In particular, a lengthy operation will effectively disable interrupts for a long time.

Second, these instructions access memory. When executed in user mode, it is possible there could be a virtual memory fault (i.e., one of the page-related exceptions). There is no clean way to handle this situation.

Finally, there is the possibility that the counter register, Reg3, is inordinately large due to a bug, and this will effectively bring the core to a stop as the instruction takes forever to execute.

Instead, we opt for coding these operations as functions, which solves all these problems.

In many systems, a Direct Memory Access (DMA) controller will be present as an I/O device. If present, the DMA controller can be employed to perform the "copy" and “clear” memory operation at a higher speed. If a DMA controller with this capability is present, then it makes sense to add a system call to access this functionality. The user-level functions will handle boundary cases and then use the system call to invoke the DMA controller to do the bulk of the work.

Chapter 6: Privileged Instructions and Kernel Mode

Quick Summary

- There are two privilege modes: kernel and user.
- Privileged instructions may only be executed in kernel mode.
- There are 16 Control and Status Registers (CSRs).

Privileged Instructions

At all times, the processor is executing in one of two possible modes:

- Kernel Mode
- User Mode

The current mode is determined by a single bit within the status register **csr_status**. Upon power-on-reset, the processor begins execution in kernel mode.

Some instructions are privileged instructions; these may only be executed when running in kernel mode. Any attempt to execute a privileged instruction when running in user mode will signal a “Illegal Instruction Exception”.

Control and Status Registers

There are 16 **Control and Status Registers** (CSRs):

0	r/o	**csr_version**	Version of the BLITZ-64 architecture ISA
1	r/o	**csr_prod**	Product Info
2	r/o	**csr_core**	Core number
3	r/o	**csr_instr**	Instruction counter (Reset upon power-on-reset)
4	r/o	**csr_cycle**	Cycle counter (Reset upon power-on-reset)
5	r/w	**csr_timer**	Time until next interrupt, in cycles
6	r/w	**csr_status**	System status register
7	r/w	**csr_stat2**	Used during trap invocation and return
8	r/w	**csr_trapvec**	Pointer to trap handler code
9	r/w	**csr_pgtable**	Pointer to page table root node
10	r/w	**csr_prevpc**	Previous PC (for trap handler)
11	r/w	**csr_cause**	A code indicating which trap just happened
12	r/w	**csr_bad**	Offending instruction
13	r/w	**csr_addr**	Bad Address
14	r/w	**csr_ptr**	Used during trap invocation and return
15	r/w	**csr_temp**	Temp work register

The following instructions are used to access the CSRs:

CSRREAD	Retrieve data from a CSR
CSRWRITE	Move data into a CSR
CSRSWAP	Simultaneous read and write to/from a CSR
CSRSET	Set selected bits to 1
CSRCLR	Clear selected bits to 0

These instructions reference a CSR, which is encoded using 4 bits in the Reg1 register field within the instruction:

0000 = **csr_version**
0001 = **csr_prod**
…
1111 = **csr_temp**

These instructions are all privileged, which means that they cannot be used by user mode code. Thus, the CSRs are hidden and inaccessible from user programs.

Some CSRs, as marked above, are read-only. Any attempt to store data into these registers is legal, but the data will simply be discarded.

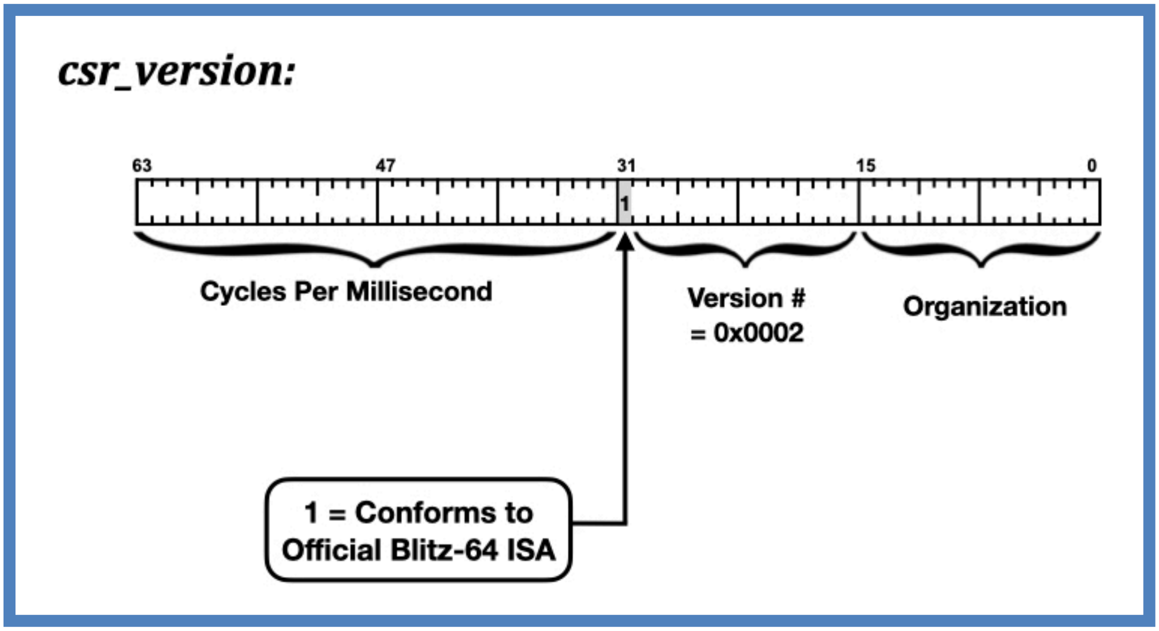
Next, we discuss the function of each CSR.

Each CSR is a 64 bit register. All 64 bits can be set and cleared like any normal register, with these exceptions:

— Unused bits in **csr_status** are always zeros; they cannot be altered.
— The registers **csr_version**, **csr_intsr**, **csr_cycle**, **csr_prod**, and **csr_core** are read-only.

csr_version

This CSR is read-only. Its value is fixed and will never change. Any attempt to update this CSR is ignored.



The uppermost 32 bits [63:32] indicate the number of cycles per millisecond that the core normally runs at. This number need not be perfectly accurate; the actual processor speed may be more or less with under- and over-clocking. (This value might be used, for example, to control the flashing of LEDs at a rate appropriate for humans or for initializing the default time-slice size.)

Bit [31] indicates whether this core fully conforms to an official Blitz-64 ISA specification. If the core meets all the requirements given in an officially sanctioned Blitz-64 specification—either this one or some future specification—this bit will be

1 and bits [30:16] will contain a version number indicating exactly which specification it conforms to.

Bits [30:16] of this CSR contain a version number of the architecture. The version documented here is 0x0002. It is intended that version numbers will be incremented sequentially as changes are made to the official Blitz-64 ISA.

A core can be said to be “in full conformance” (i.e., “compliance”) to this version of the Blitz-64 specification if and only if all instructions, registers, and behaviors documented here are implemented exactly as described.

However, the inclusion of additional, novel instructions is acceptable and is not cause for bit [31] to be zero. If an opcode that is defined as an “illegal instruction” in this document is assigned to a newly created instruction, then it will not affect bit [31].

If the architecture of a core fails to meet any official Blitz-64 ISA specification, bit [31] must be zero.[11] We presume that if a Blitz core fails to fully conform, then it will at least implement “a lot of” the Blitz specification. In particular, we assume the version number (bits [30:16]) will still contain the version number of the Blitz-64 ISA that is most closely implemented by the core, such as 0x0001.

Bits [15:0] is the “implementor/organization” field and contains a value which identifies a specific implementor (e.g., a person, group, or corporation). These numbers are to be assigned centrally and are not to be created independently. The current assignment is:

0x0000	All other implementors / organizations
0x0001	Harry Porter
0x0002	HDL Express

Values 0x0003 … 0xFFFF are reserved for future assignment; do not use them.

NOTE: The **csr_prod** register (“product info”) is intended to be defined by a specific implementor/organization. Software recognizing a particular implementor/

[11] For example, if the MULADD instruction causes an Emulated Instruction Exception, bit [31] must be 0. As another example, if FDIV causes an Emulated Instruction Exception, it is not cause for bit [31] to be 0. But is FDIV is implemented but fails to round ties to even, then the entire core fails to conform and bit [31] must be 0.

organization may wish to examine **csr_prod** to gather more info about the specific capabilities of the core.

csr_prod

This CSR is read-only. Its value is fixed and will never change. Any attempt to update this CSR is ignored.

The definition of this register is left up to the specific implementation. The implementor /organization is identified in **csr_version**, which should be consulted before any attempt to decipher the contents of this register.

The intent is that a particular implementor or organization may create several implementations of the Blitz-64 architecture. Each might be considered a unique "product". This register is intended to contain a number that identifies the product. The implementor is free to define certain bits in this register to indicate the presence or absence of certain features. For example, certain bits might indicate whether some special instruction is available, or whether the core is optimized for "low power" or for "high performance".

It may also be the case that each part has a serial number hardwired into it and a part's serial number can be obtained by reading this register. For example, the upper 16 bits might contain a product number and the lower 48 bits might contain a serial number.[12]

csr_core

This CSR is read-only. Its value is fixed and will never change. Any attempt to update this CSR is ignored. This register is divided into these fields:

bits [63:48]	Number of columns (0 ... 65536)
bits [47:32]	Number of rows (0 ... 65536)
bits [31:16]	Number of planes (0 ... 65536)
bits [15:0]	Number of this core (0 ... 65536)

[12] Of course current technology may make it impractical to hardwire unique serial numbers directly into each core. Thus, an implementation may choose to reveal serial number information in other ways, such as through a memory-mapped I/O device created for this purpose, or a number written into Secure Storage and therefore not updatable.

In a multi-core processor, the lower 16 bits [15:0] of this CSR give the core number and will be within 0 … 65536. Core number 0 is considered the "primary core".

In a multi-core system, the cores can be arranged either linearly, in a 2 dimensional array, or in a 3 dimensional array. The upper 48 bits describe the arrangement of the cores.

In a 1D arrangement, the number of rows and the number of planes will be 1. In a 2D arrangement, the number of planes will be 1.

In a uni-core processor, this register will contain 0x0001_0001_0001_0000. In a multi-core system with ***NNNN*** cores in which the cores are not organized as an array, this register will contain:

0x***NNNN***_0001_0001_***KKKK***

where ***KKKK*** is the number of a particular core (within 0…***NNNN***-1).

csr_instr

This CSR is set to zero upon power-on-reset. It is incremented by one for every instruction executed. It can be used for performance measurement.

This CSR is read-only; any attempt to update this CSR is ignored.

csr_cycle

This CSR is set to zero upon power-on-reset. It is incremented by one for every clock cycle. It can be used for performance measurement.

(Assuming the processor runs at 10 gigahertz, the **csr_cycle** will run for about 30 years before overflowing. Thus, a problem will only arise in a processor core which runs non-stop for decades. The workaround is to reboot every decade.)

This CSR is read-only; any attempt to update this CSR is ignored.

csr_timer

The processor has a built-in timer. This timer is used by the kernel to implement time-slicing. When the timer expires (i.e., when the time-slice ends), a “Timer Interrupt” will be signaled.

This CSR controls when the timer will cause the interrupt. This register is decremented on every clock cycle, with no check for overflow. A “Timer Interrupt” will be signaled when this CSR goes negative. The trap handler should reset this CSR before reenabling interrupts to avoid an infinite chain of timer interrupts.

If no interrupt is wanted, the value MAX_64 (0x7FFF_FFFF_FFFF_FFFF) can be used. The maximum time interval is measured in decades, but normally it will be reset after every time-slice, e.g., every millisecond.

Commentary The timer is specified in terms of clock cycles, rather than real-time, since it is easier to implement. A real-time clock may exist, but it will be implemented as a separate I/O device, probably with a separate power supply and independent frequency generator, so that it continues to operate even when the processor is powered down and can measure time accurately, regardless of which frequency the processor is clocked.

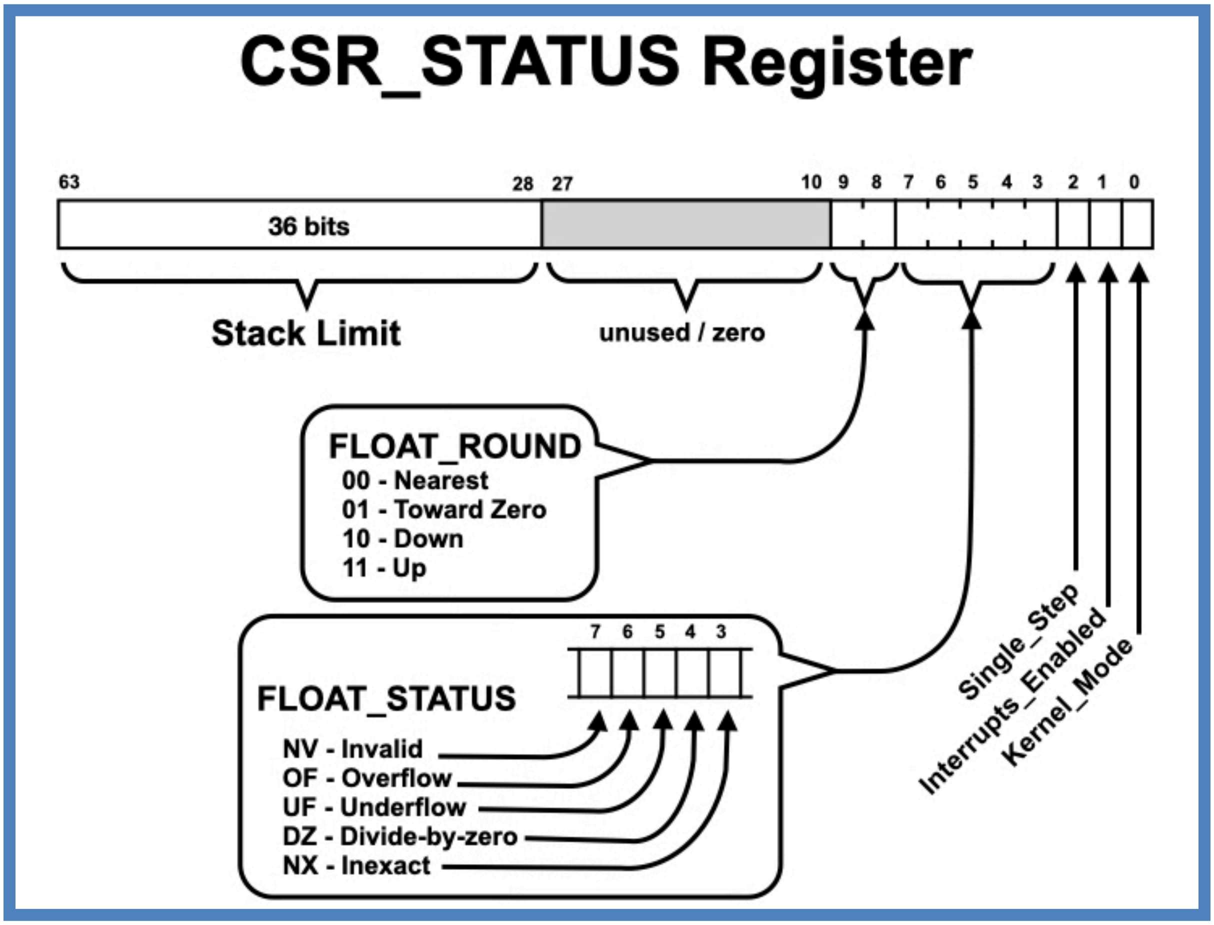
The purpose of the cycles-per-millisecond field of **csr_version** is so that a real-time clock can be approximated from the processor cycle frequency, if no real-time clock is present.

csr_status

This CSR is broken into the following fields:

[63:28]	36	StackLimit
[27:10]	18	*< unused / zero >*
[9:8]	2	FLOAT_ROUND — the rounding mode for float operations
[7:3]	5	FLOAT_STATUS — the error status of recent float operations
[2]	1	SINGLE_STEP (1=enabled, 0=disabled)
[1]	1	INTERRUPTS_ENABLED (1=enabled, 0=disabled)
[0]	1	KERNEL_MODE (1=Kernel Mode, 0=User Mode)

FIGURE: CSR_STATUS Register



There are 2 bits (FLOAT_ROUND) which should be set by software to control which rounding mode to be used when the need arises during floating point computations:

00	RN — Round to nearest. Tie goes to value with 0 in LSB.
01	RZ — Round toward zero, i.e., truncate.
10	RD — Round down, i.e., round toward -inf.
11	RU — Round up, i.e., round toward +inf.

There are 4 bits (FLOAT_STATUS) which are set by hardware to reflect recent floating point computations. Here is the meaning of these bits:

bit	
3	NX — Inexact results were produced
4	DZ — Divide by zero has occurred
5	UF — Underflow has occurred[13]
6	OF — Overflow has occurred
7	NV — Invalid operation has been attempted

A value of 00000 indicates that no problems have occurred. These bits are "sticky", once set to 1 they remain set until explicitly cleared by software.

The floating bits (FLOAT_ROUND and FLOAT_STATUS) must be saved and restored upon any context switch. This is why they are in the status register.

In general , non-privileged instructions are not allowed to read or write the CSR registers. However, the GETSTAT and PUTSTAT instructions (which are not privileged) can be used to read and write bits [9:3]. This allows user programs to make use of the FLOAT_ROUND and FLOAT_STATUS bits.

The INTERRUPTS_ENABLED bit determines whether an interrupt will cause an immediate trap or not. If set to 1, any interrupt will cause trap processing to occur after the current instruction completes execution. If the bit is 0, the signaling of an interrupt will not cause trap handling. Instead, that interrupt type will become pending. The interrupt will remain pending until the bit is set to 1, at which time trap processing will occur.

[13] The "underflow" (UF) bit is set when the result of an operation is both a subnormal number (including zero) and the result is inexact.

The KERNEL_MODE bit determines whether the processor is in kernel mode or user mode. The mode determines whether privileged instructions can be executed and determines whether addresses below 0x8_0000_0000 (i.e., physical memory and memory mapped I/O) can be used directly.

Code running in user mode, but with interrupts disabled, is particularly risky since an infinite loop would freeze the system: even timer interrupts would not stop the looping. The kernel will normally not run user mode code with interrupts disabled.

The SINGLE_STEP bit is used to invoke a trap handler after each instruction. This functionality is used by a debugger to single-step the code being debugged.

Bits [63:28] contain a 36 bit address called the "**StackLimit**". The **StackLimit** is the smallest address which is allowed; attempting to grow the stack beyond the value (i.e., below, with smaller addresses) will cause a Stack Overflow Exception. This is discussed more fully in the section on the "Stack Overflow Exception".

Bits [27:10] are unused. Attempts to set these bits are ignored and reads always return zeros.[14]

csr_stat2

The **csr_status** CSR must be saved and restored at context switches and this is the function of **csr_stat2**. During the hardware phase of trap invocation for interrupts and exceptions, the status register **csr_status** is copied to this CSR. This allows the software trap handler to return to the interrupted code at some later time.

This process is discussed more fully later, when trap processing is described.

csr_trapvec

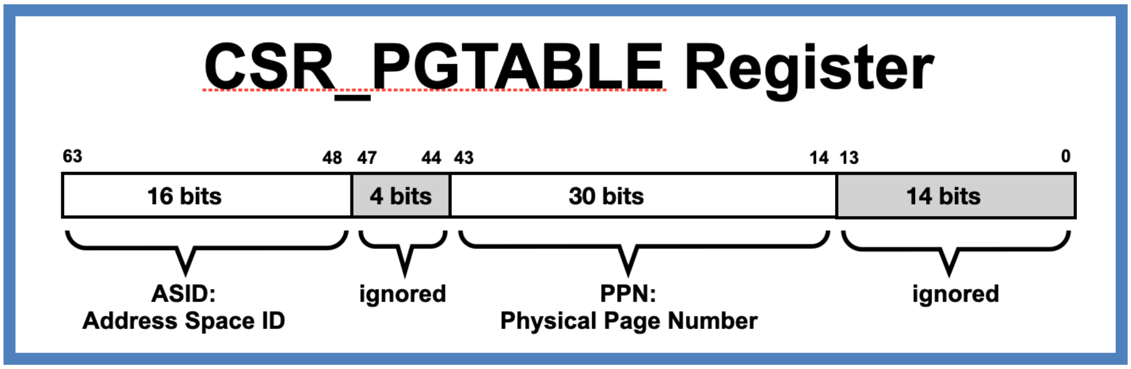
This CSR will contain the address of the function that will handle traps. When a trap (i.e., an interrupt or exception) occurs, the program counter will be loaded with the address in **csr_trapvec** as part of trap handling. Execution of the trap handler will then begin with the next instruction FETCH.

[14] If, in the future, a new version of the architecture defines some meaning for an unused bit, then we must ask whether the bit will be visible to User Mode programs or not. If visible to User Mode, then it would make sense to redefine the GETSTAT and PUTSTAT instructions to include the newly defined bit, along with the FLOAT_ROUND and FLOAT_STATUS bits.

Only 35 bits — i.e., bits [34:0] — of **csr_trapvec** will be used, giving an address in the kernel address space; the upper bits will be ignored. If **csr_trapvec** is not a valid address — for example 0, which would normally cause a Null Address Exception — the results are undefined.

csr_pgtable

This CSR will point to the current page table. That is, **csr_pgtable** will contain the address of the root node of the page table tree. In addition, this register will contain the address space's ID (ASID).



The address of the root of the page table tree is a 44 bit address in the physical memory space. The root of the page table must be page aligned. A page-aligned address will contain zeros in the lower 14 bits. As such, only bits [43:14] are used, constituting the Physical Page Number (PPN) of the page containing the root node. The offset bits [13:0] are ignored and zeros are assumed. This results in a page-aligned address in the 16 TiByte physical address space.

The Address Space Identifier (ASID) is a 16 bit number which identifies the virtual address space. Each virtual address space should have a unique ASID, so there is a one-to-one correspondence between virtual address spaces and ASIDs.

The ASID from **csr_pgtable** will be used by the Memory Management Unit (MMU) during address translation during any LOAD, STORE, or FETCH. It will be matched against the ASID value stored in the Translation Lookaside Buffer (TLB) registers.[15]

csr_prevpc

This CSR is used to save the value of the Program Counter (PC) during trap processing. During the hardware phase of trap invocation for interrupts and exceptions, this CSR is set to point to either the instruction causing the exception or the following instruction (depending on exactly which interrupt or exception has occurred). This allows the software trap handler to return to the interrupted code at some later time.

This process is discussed more fully later, when trap processing is described.

The PC is only 36 bits, yet **csr_prevpc** is a 64 bit register. When the PC is copied to this register, the upper 28 bits will be set to zero. When **csr_prevpc** is copied to the PC, the upper 28 bits will be ignored.

csr_cause

This CSR is set by the hardware during trap processing to a code to indicate which exception/interrupt caused the trap.

csr_bad

This CSR is set by the hardware during the trap processing to contain the instruction that caused the exception.

When an instruction is copied to **csr_bad**, the upper bits will be set to zero.

[15] It is assumed that the Translation Lookaside Buffer (TLB) will cache page table entries in an associative memory, in order to reduce the number of page table lookups. Each TLB entry will be keyed on both ASID and virtual page number. Presumably, each virtual address space will be assigned and associated with a unique ASID.

The purpose of the ASID is to make sure that code running in one virtual address space will only use TLB entries that are associated with the correct virtual address space. If, for some reason, the TLB registers are not implemented (meaning every FETCH, LOAD, and STORE requires a page table lookup), the ASID becomes unnecessary and will be ignored.

csr_addr

This CSR is set by the hardware during trap processing of certain exceptions to contain additional information about the exception. For example, in the case of page-related exceptions, this CSR is set to the program-generated address causing the problem.

When the hardware stores an address in **csr_addr** during an exception, the upper 28 bits will be set to zero.

csr_ptr

This CSR is intended to be used by the kernel to store a pointer to a "thread control block" while a user mode program is running. During trap handling and/or context switches the kernel must save user mode state, including registers, and this CSR is intended to contain a pointer to where in memory that saving should be done.

This CSR is neither set nor queried by the hardware, except by the CSR instructions.

csr_temp

This CSR is available for use as needed by trap handler and/or kernel code. It is neither set nor queried by the hardware, except by the CSR instructions.

Chapter 7: Exceptions, Interrupts, and Trap Handling

Quick Summary

- There are two kinds of traps: exceptions and interrupts.
  - Exceptions are synchronous and include the “syscall” trap.
  - Interrupts are asynchronous and come from I/O devices.
- All trap processing invokes a single Global Trap Handler.
- The trap cause code can be used as an index into a jump table.
- A single timer is specified, which will signal a “Timer Interrupt”.
- Up to 1,024 different SYSCALLs are supported in the jump table.
- A novel approach for null pointer exceptions is used.

Traps, Exceptions, and Interrupts

There are two sources of “**traps**”, namely exceptions and interrupts.

- **Trap**
  - **Exception** — synchronous, caused by an instruction
  - **Interrupt** — asynchronous, caused by an external source

An “**exception**” is caused by and related to a specific instruction. In that sense, exceptions are **synchronous**.

An “**interrupt**” is caused by the arrival of a signal from an external source. Interrupts are **asynchronous**, which means their timing is unrelated to instruction execution. Interrupts can occur at any time during execution.

Exceptions and interrupts are said to be **signaled** or **raised**. (We use the terms “signaled” and “raised” synonymously.)

In response to an exception or interrupt, the hardware will invoke a **trap handler**.

Exceptions

An exception is raised by the execution of an instruction and is therefore directly related to one particular instruction.

In some cases, exceptions are caused by a problem in the instruction which prevents it from executing. In other cases, the instruction requires kernel attention and will be re-executed after the kernel handles the exception. A **system call** (raised by the "**syscall**" instruction) is considered to be a type of exception.

When an exception occurs, trap handling will always be invoked directly after the instruction. Trap handling for exceptions will never be delayed, become pending, or be ignored.

Interrupts

An interrupt is signaled when an external source sends a hardware signal to the processor. The device is requesting attention. An interrupt has nothing to do with the instruction currently being executed.

When an interrupt is signaled, the interrupt becomes **pending**. Trap processing will occur at some future time. The interrupt remains pending until trap handling occurs. The interrupt may be handled immediately after the completion of the current instruction, or it may be postponed until later. Either way, the interrupt will remain pending until trap handling is invoked.

Once handled, the interrupt will cease being pending.

Interrupts are masked by the INTERRUPTS_ENABLED bit in the status word **csr_status**. If 0, then any interrupt that occurs will remain pending until interrupts are once again enabled. Trap processing will not occur until this time.

Exceptions may not be masked. Any exception will cause trap processing immediately after the current instruction (i.e., the instruction which caused the exception) has finished execution. In that sense, we might say that exceptions remain pending only a very short time, until the current instruction is completed.

Interrupt handling is only masked by the INTERRUPTS_ENABLED bit in the **csr_status** register. Unlike other ISAs, Blitz-64 has no additional masking mechanisms.[16]

There are several different types of interrupt (e.g., Timer Interrupt, Serial Device Interrupt, etc.). If interrupts of two or more types are pending then, when trap processing occurs, the one with the higher priority will be selected and the others will remain pending during the execution of the trap handler. At some later time, the other interrupts will be processed.

If there are multiple interrupts of the same type signaled before trap handling occurs, then they are combined. In other words, only one trap of each interrupt type can be pending at once. If an interrupt of type X is signaled and is still pending at the time a second interrupt of the same type X is signaled, the second interrupt is combined with the first interrupt, which means it is effectively ignored and lost.

Trap Handlers

When an interrupt or exception occurs, a trap handler will be invoked. An interrupt may remain pending for some time, but eventually it will be handled. An exception will be handled immediately, before the next instruction is executed.

Trap handling has two components: the **hardware component** and the **software component**. When the trap handling is invoked, the processor will perform several simple actions. These actions will occur between instructions. In other words, the previous instruction will complete processing, and then the hardware component will execute.

Basically, the hardware phase will save some processor state, clear the INTERRUPTS_ENABLED bit in **csr_status**, and set the Program Counter (PC) to point to the trap handler.

After the hardware component has completed, instruction processing will resume, with the first instruction in a kernel function known as a “**trap handler**”.

[16] However, the Platform-Level Interrupt Controller (PLIC) can be used to mask interrupts from various selected devices to various selected cores.

Here are the **types of trap**:

Exceptions

SYSCALL (multiple types, determined by immed10 in SYSCALL instruction)
Arithmetic Exception (integer overflow, divide-by-zero, …)
Unaligned LOAD/STORE Exception
Null Address Exception
Illegal Instruction Exception (including privileged instruction violation)
Page Illegal Address Exception (attempt to access kernel space)
Page Table Exception (bad **csr_pgtable**)
Page Invalid Exception (either index page or PTE)
Page Write Exception
Page Fetch Exception
Page Copy-On-Write Exception
Page First Dirty Exception
Debug Exception
Breakpoint Exception
Singlestep Exception
Emulated Instruction Exception
Hardware Fault Exception
Bad Array Index Exception
Stack Overflow Exception

Interrupts

Timer Interrupt
DMA Complete Interrupt
Platform Level Interrupt Controller (PLIC) Interrupt
… Interrupt details are implementation dependent …

When a trap occurs, a transfer of control is made to the address in **csr_trapvec**. In other words, the value in **csr_trapvec** is copied into the Program Counter (PC) as part of the trap processing.

Thus, there is a single trap handler for all trap types, which we call the "**global trap handler**". This function is responsible for determining the nature of the trap and jumping to individual trap handlers to finish the handling of the trap.

Generally speaking, we expect there will be several **individual trap handler functions**, one for each type of trap. Trap handlers will typically end with a SYSRET instruction, which will be used to resume execution in the interrupted code sequence.

Upon trap handling, the hardware will cause a jump to the global trap handler by loading the PC with the address of the the global trap handler, i.e., the contents of **csr_trapvec**. Presumably, this CSR has been previously loaded with the address of the global trap handler.

It is intended that the global trap handler will begin by saving additional state of the interrupted process (such as the general purpose registers) and then jump, via a **trap dispatch table**, to the desired individual trap handler.[17]

This dispatch vector is entirely in software. This global trap handler, which may be written in assembler, will perform an indirect jump through the jump table. The individual target trap handlers will typically be written in high-level KPL.

The trap dispatch table will contain one entry for each type of trap and each entry is 8 bytes. With 8 bytes, there is enough room for two instructions, so, if necessary, each entry can contain a long absolute jump (UPPER20+JALR). However, many trap handler functions may be close and reachable with a single instruction. Although all table entries are 8 bytes, some will be padded with unused bytes.

There are approximately 1100 types of traps, since there are 1024 different syscall traps. So the trap vector will consume about 8,800 bytes.

At the time a trap is handled, the hardware will set the **csr_cause** register to a code indicating which sort of trap is being handled. Each code is divisible by 8, which makes trap dispatching simpler.

[17] Of course, the global trap handler may be written in an entirely different style, avoiding the jump table altogether. Instead, such a global trap handler would contain special case testing to determine which particular trap has occurred.

Code (decimal)	Code (hex)	Trap Type
0	0000	Syscall 0
8	0008	Syscall 1
16	0010	Syscall 2
…	…	…
8184	1FF8	Syscall 1023
8192	2000	Arithmetic Exception
8200	2008	Unaligned LOAD/STORE
8208	2010	Null Address Exception
8216	2018	Illegal Instruction, privilege violation
8224	2020	Page Illegal Address Exception
8232	2028	Page Table Exception
8240	2030	Page Invalid Exception
8248	2038	Page Write Exception
8256	2040	Page Fetch Exception
8264	2048	Page Copy-On-Write Exception
8272	2050	Page First Dirty Exception
8280	2058	Debug Exception
8288	2060	Breakpoint Exception
8296	2068	Singlestep Exception
8304	2070	Emulated Instruction Exception
8312	2078	Hardware Fault Exception
8320	2080	Bad Array Index Exception
8328	2088	Stack Overflow Exception
8336	2090	Timer Interrupt
8344	2098	DMA Complete
8352	20A0	PLIC Platform Level Interrupt Controller

… Codes for asynchronous interrupts are implementation dependent …

Note that bit 13 (8192=0x2000) of **csr_cause** indicates whether this is a syscall or not.

Cause codes are zero-extended to 64 bits whenever the hardware writes them into a CSR register.

Interrupt Processing

Interrupt processing occurs between instructions.

If the previous instruction happens to have caused an exception, then that exception will be processed first. Processing an exception will always have the effect of disabling interrupts. Then, since interrupts are disabled, any pending interrupt will remain pending until after interrupts are re-enabled, at which time the interrupt processing will occur. But assuming there is no exception, the interrupt will be processed.

If multiple interrupts are signaled, then one will be selected. The order of preference among the different interrupt types is implementation dependent.

The previous instruction will complete execution before an interrupt is processed. If there are any future, partially-executed instructions in the pipeline, they will be cancelled and will have no effect.

When an interrupt is processed, the register **csr_prevpc** will be set to the address of the next unexecuted instruction, which is just the value of **pc** (the Program Counter). The registers **csr_bad** and **csr_addr** will be set to zero.

> **csr_prevpc** = the address of the next instruction
> **csr_bad** = 0
> **csr_addr** = 0

We next discuss the individual types of asynchronous interrupts.

Timer Interrupt

The register **csr_timer** is decremented on every clock cycle. If it is negative and interrupts are enabled, then a "Timer Interrupt" will be signaled.

If interrupts are disabled, this interrupt will not be signaled. When interrupts are once again enabled, this interrupt will occur if and only if **csr_timer** is still negative at that time.[18]

Presumably the trap handler that deals with timer interrupts will reset **csr_timer** in preparation for the next time-slice. But before it can do this, there will several cycles in which the **csr_timer** remains negative. However, the negative value of **csr_timer** will not cause an interrupt since interrupts are disabled during the handler, thus avoiding an infinite cascade of interrupts.

If another interrupt is signaled simultaneously to a Timer Interrupt, then the determination of which interrupt is handled first is implementation dependent.

If another interrupt has priority and is handled before the Timer Interrupt, then the Timer Interrupt does not remain pending; instead it will be re-signaled when interrupts are once again enabled if and only if **csr_timer** is still negative.[19]

DMA Complete Interrupt

When the DMA Controller device completes an operation, it will signal this interrupt.

If interrupts are disabled, this interrupt will remain pending.

[18] Previously we said that an interrupt, once signaled, remains pending until interrupts are again enabled. What if a Timer Interrupt becomes pending while interrupts are disabled, i.e., during the trap handler for some other trap? What if that trap handler subsequently resets **csr_timer**, in an attempt to reschedule the Timer Interrupt? Shall the Timer Interrupt, once raised, remain pending or shall it be checked anew during the execution of each instruction?

Here we specify that the core will check **csr_timer** for each instruction. It will not remain pending.

[19] The idea is that the other interrupt's handler may invoke the scheduler and cause a different thread to be scheduled, thereby terminating the previous thread's time-slice. Such a thread-switch would naturally cause **csr_timer** to be reset to a new value. Allowing an earlier Timer Interrupt to remain pending and to occur later once interrupts are re-enabled would effectively terminate the new thread's time-slice the moment it starts, before it has a chance to do anything.

PLIC Complete Interrupt

Various I/O devices will interrupt the core from time to time. In a multi-core system, we need to have exactly one core service each interrupt. The duty of the Platform-Level Interrupt Controller (PLIC) is to channel each interrupt to a single core and moderate between multiple cores that are each willing and able to service interrupts.

When the PLIC wishes to interrupt a core, the interrupt will be raised. If interrupts are enabled, a trap will occur. The core will then communicate directly with the PLIC — which is a memory-mapped device — to determine which device in interrupting and to claim the interrupt.

If interrupts are disabled, this interrupt will become pending and trap processing will not occur until interrupts are re-enabled. However, if some other core claims the interrupt in question, the PLIC may cause the pending interrupt to disappear so that trap processing will not interrupt the core once it re-enables interrupts.

The details of the Platform-Level Interrupt Controller (PLIC) are discussed elsewhere.

Additional Devices

There may be additional devices that interrupt. Details are implementation dependent.

Description of Exceptions

SYSCALL Exception

This exception is caused by the execution of the SYSCALL instruction.

The SYSCALL machine instruction is used in the implementation of a "**system call**" to a kernel function. By convention, the standard calling conventions are used, which means that all arguments and returned results are passed in registers r1, r2, … There will be many system calls and the global trap handler must be able to quickly dispatch to the correct individual trap handler (i.e., to the desired kernel system function).

The SYSCALL instruction takes a 10 bit immediate value which is interpreted as an integer in the range 0 … 1,023. This integer is used in dispatching to the individual syscall trap handlers. The integer is multiplied by 8 (since each dispatch table entry is 8 bytes) and is placed in **csr_cause**.

Otherwise, exception processing occurs just like other exceptions.

The Blitz-64 design facilitates fast dispatching for the most commonly used system call functions. If there are more than 1,024 system calls, one of the code numbers (e.g., the last code number) can be used to implement a second level of dispatching for functions that are not commonly used and not performance-critical.

This exception will set…

csr_prevpc = the address of the instruction following the SYSCALL
csr_bad = the syscall instruction
csr_addr = 0

Commentary Here are the programming conventions that will be followed by user mode code making a system call to the kernel:

- There can be up to 6 arguments. Arguments are passed in registers r1 … r6. (Recall that the normal calling conventions allow up to 7 arguments in registers.) Any additional argument data must be passed in user memory, placing a pointer to the memory area (i.e., a virtual address) in r1 … r6.
- A value will be returned in register r1. Zero will be returned for system calls that have no meaningful return value. If additional return data is required, the

kernel will place it in memory at the virtual address supplied by one of the arguments.

- Upon return, registers r2 … r7, and r8 (t) will be zero.
- Registers r9 … r15 (i.e., s0, s1, s2, tp, gp, lr, and sp) will be unchanged.

Arithmetic Exception

This exception can be caused by the following operations:

Integer arithmetic :	ADD, ADDI, SUB, MULADD, DIV, REM
Shift operations:	SLA, SLAI, SRA, SRL, SLL
Size checking:	CHECKB, CHECKH, CHECKW

The instruction may have modified RegD but the result computed is not "mathematically correct."

This exception will set…
csr_prevpc = the address of the offending instruction
csr_bad = the offending instruction
csr_addr = 0

Unaligned LOAD/STORE Exception

This exception can be caused by the following instructions:

LOAD.H, LOAD.W, LOAD.D
STORE.H, STORE.W, STORE.D

This exception will be signaled whenever the program-generated address is not properly aligned. The instruction may have modified RegD but any value stored is incorrect.

This exception will set…
csr_prevpc = the address of the offending instruction
csr_bad = the offending LOAD or STORE instruction
csr_addr = the program-generated address

Emulation of Unaligned LOAD and STORE Instructions

Perhaps unaligned data will be simply banned by fiat. Whenever it might occur, we shall make it the responsibility of compiler/programmer to use properly aligned operations to read and write data to/from memory. Note that the descriptions earlier in this document for the machine instructions LOAD.H, LOAD.W, LOAD.D, STORE.H, STORE.W, and STORE.D included the requirement that the addresses must be properly aligned. With this approach, this restriction is enforced.

If this approach is to be taken, then, whenever this exception occurs, it indicates a program error. The trap handler will probably just terminate the offending process.

However, the Blitz-64 ISA is designed to support another more complex approach, in which the compiler/programmer is relieved of the responsibility to always use aligned addresses. With this approach, the compiler/programmer is free to use LOAD and STORE instructions with addresses that are not properly aligned.

Next we describe this approach.

It is assumed that the majority of LOAD and STORE operations will be properly aligned, but unaligned data will occasionally occur and the compiler/programmer will not take special action to check alignment.

Instead, the Unaligned LOAD/STORE Exception is designed to allow a trap handler to intervene and deal with unaligned data addresses by completing the operation (using only aligned LOAD and STORE instructions) and returning to the interrupted code. To the programmer of the original code, it will simply appear that the LOAD and STORE instructions work just fine with improperly aligned addresses.

In reality, the unaligned LOAD or STORE will invoke a handler that will run in Kernel Mode and will ultimately return to the interrupted user code.

The ALIGN and INJECT instructions are specifically designed to be used in such a trap handler to support unaligned LOAD or STORE operations.

Generally speaking, the trap handler for the Unaligned LOAD/STORE Exception will need to make two accesses to memory. For example, to load a doubleword from 8 bytes that spans two properly aligned doublewords, the handler will need to load two aligned doubleword and extract the initial bytes from the first and the final bytes from the second. (Example code sequences are discussed in detail elsewhere in this document.)

When an unaligned LOAD or STORE operation also causes a page-related exception, the Unaligned LOAD/STORE Exception will have priority, Thus, the trap handler code for Unaligned LOAD/STORE Exception will be invoked.

Once invoked, the trap handler will then go on to perform two aligned operations. One or both of these may cause a page-related exception.[20] Such exceptions could either be fatal to the LOAD/STORE or repairable. For example, if the operation is a STORE into a page that is not writable, then it is fatal; the "Page Write Exception" needs to happen and the STORE aborted. On the other hand, a "Page Copy-On-Write Exception" should be transparent; it may require the kernel to copy a page, but the STORE operation should complete with no consequence to the user code.

There are two approaches the kernel programmer can take.

In the first approach, the trap handler code can use the CHECKADDR instruction before accessing data in the user's virtual address space. If a page-related exception would occur, the handler can directly invoke whatever kernel functions are required. Once complete and the handler code is assured that access to the memory is safe, it can proceed.

But if CHECKADDR indicates that a fatal page-related exception would occur, the trap handler for the Unaligned LOAD/STORE Exception will need abandon the operation and end by simulating the page-related exception.[21] In any case, interrupts remain disabled throughout the handler.

In the second approach, interrupts are reenabled. The handler code for the Unaligned LOAD/STORE Exception becomes interruptible. Then it simply performs the necessary memory operations without first checking, possibly causing a page-

[20] The first memory access might not cause an exception while the second access — falling within a different page — might cause an exception.

[21] An exception can be "simulated" or "faked" as follows. The kernel code must set **csr_stat2**, **csr_prevpc**, **csr_cause**, **csr_bad**, and **csr_addr** as they would be if the faked exception had actually occurred. Then it must jump to the start of the global trap handler, i.e., to the address in **csr_trapvec**. From then on, the trap handler will deal with the exception just as it normally does with all exceptions, without knowing that it was tricked.

related exception. If such a page-related exception occurs, it will be handled (since interrupts are enabled) by the appropriate handler. Assuming the exception is not fatal, there will be a return to the interrupted handler and the handler will then run to completion.[22]

If the page-related exception is fatal, then the user thread must be terminated. The kernel will need to detect that the problem occurred during the emulation of a LOAD or STORE so that it can report it correctly. The real error location is within the user code at the instruction that caused the Unaligned LOAD/STORE Exception, not within the trap handler that actually caused the fatal exception.

Concerning Atomicity

Note that the emulation of unaligned LOAD and STORE operations differs in an important way from aligned operations. Aligned LOAD and STORE instructions are guaranteed to be atomic, which means that the entire operation is either executed or not. With regard to other unrelated memory operations, the LOAD/STORE instruction either occurs before or occurs after the other operation. There is no interleaving.

With a single-core processor, it might seem like there is no risk concerning atomicity. If there is only one core that can be issuing LOADs and STOREs to memory, you might assume an unaligned memory operation is effectively atomic, because the trap routine that emulates unaligned operations runs with interrupts disabled. Nothing can interrupt the core between the first memory operation and the second memory operation and there is no second core capable of touching memory.

However, the kernel programmer must think carefully. What if the second memory operation entails a page-related exception? The thread might get rescheduled and other threads might run while the trap handler is waiting for a page to be read in from disk. Or, what if there is an I/O device that is also accessing memory? For example, a DMA controller might be copying a block of memory at the same time the unaligned LOAD/STORE is being emulated.

[22] Is it guaranteed that there will not be additional page-related exceptions? Might it be necessary to pin the pages to prevent an infinite chain of page-related exceptions? These are the challenges that make kernel hacking fun!

Practically speaking, the problems associated with non-atomic memory operations should be dealt with in other ways. Typically, the kernel will lock shared data, thus enforcing the constraint that only one "customer" is allowed to touch the shared memory at any time. The use of locks can enforce data consistency. However, within the code to implement the locks themselves, it may be necessary to use atomic operations, so care must be taken.

Null Address Exception

This exception can be caused by the following instructions:

Store to memory :	STORE.B, STORE.H, STORE.W, STORE.D
Read from memory:	LOAD.B, LOAD.H, LOAD.W, LOAD.D
Jumping:	JAL, JALR, B.EQ. B.NE, B.LE, B.LT
Other:	NULLTEST, CAS

This exception will set…

csr_prevpc = the address of the offending instruction
csr_bad = the offending instruction
csr_addr = 0

Any attempt to use an address in the range 0…7 as the target for a LOAD, STORE, jumping, NULLTEST, or CAS instruction will cause a "Null Address Exception".

A LOAD.X instruction may have modified RegD but the value is "undefined." A STORE.X instruction may or may not have modified the first 8 bytes of physical memory (i.e., the doubleword at address 0); this behavior is undefined.

If any jumping instruction sets the PC to zero, there will be no effect on the PC because the trap handling will immediately overwrite the PC. The PC value saved into **csr_prevpc** will be the address of the jumping instruction itself.

Note that the machine instructions for jumping are used in the implementation of a number of synthetic instructions, including CALL, CALLR, JR, RET, and the branch instructions.

Commentary "Null" pointers are widely used in programming. Ideally programs are either bug-free or will always test pointers before use, so that there will never be any attempt to dereference a null pointer. But alas, we live in a difference universe.

The Blitz manifesto dictates that we should try to catch and handle every error. Catching null pointer dereferencing must be done. We considered requiring the compiler to implicitly insert a test for every pointer dereference. In fact, we took this approach in Blitz-32 and came to appreciate the testing enormously. However, there was a huge performance hit: a test and branch for every pointer use. We also considered adding a new instruction, whose sole purpose is to cause an exception if necessary. But the overhead of even a single instruction is too much.

The novel approach we are introducing with Blitz-64 is an unusual innovation.

In code such as:

```
  i = *p
```

the compiler will produce a single instruction, such as:

```
  LOADD      r1,0(r2)
```

With our approach, no additional test is necessary.

In other situations, an offset from the pointer is needed, as in:

```
  i = p.field
```

For this, the compiler might produce an instruction sequence such as:

```
  NULLTEST   r2
  LOADD      r1,48(r2)
```

With Blitz-64, the 8 bytes at location 0 are forever inaccessible, unused, and wasted.

Any attempt to read or write location zero will cause a “Null Address Exception”. If the programmer ever attempts to use a NULL pointer, the program will undergo controlled exception handling. The payoff is that every use of a pointer will be checked in hardware, in parallel to other execution. Thus, we expect no performance penalty.

This particular sort of error is unique and important enough to be handled specially. The programmer deserves an error message that says “NULL pointer used” instead of the legacy incantation “segmentation fault”.

We should note that one approach taken in traditional OSes is to reserve an entire virtual page. Page number zero in the user’s address space is not mapped, and any attempt to read/write to it will cause a virtual memory exception. Our approach uses a separate type for this error; it is not piggybacked onto a more general error condition.

We also want to catch null pointer use in kernel code.

In traditional systems, null pointer use within the kernel can be trapped by executing the kernel with virtual memory mapping turned on. Page zero in the kernel's virtual address mapping would be marked as invalid. If the kernel's page table maps all of physical memory to virtual memory in a one-to-one manner, then, marking page zero as invalid will waste an entire page of physical memory. The Blitz-64 approach wastes only 8 bytes.

The Blitz approach seeks to improve the performance of kernel code by avoiding the necessity of mapping the kernel address space. Thus, Blitz kernel code can potentially execute faster than in other ISAs, since address mapping is not used for kernel code. However, since the kernel's address space is not mapped into a virtual address space, catching null pointers by using a mapping in which the page at virtual address 0x0_0000_0000 is invalid is not possible in Blitz-64. But the Null Address Exception works better anyway.

With a 1 GiByte physical memory, the sacrifice of 8 bytes is an insignificant, trivial overhead.

The memory at location 0 can, in fact, be read and written. Physical page zero can be mapped into a virtual page, and the first byte of this page can be read/written with a non-zero virtual address, effectively accessing byte 0. Recall that the first pages of physical memory are intended to contain the kernel's static data. So mapping these into a virtual address space for use by user-level code would be foolish and risky.

Illegal Instruction Exception (including Privilege Violations)

Any attempt to execute an instruction with an undefined OPCODE will cause this exception. Any attempt to execute a privileged instruction while running in user mode will also cause this instruction. The instruction named ILLEGAL will cause this exception.

The privileged instructions are:

SYSRET
SLEEP1
SLEEP2
CONTROL
CSRSWAP
CSRREAD
CSRWRITE (a synthetic form of CSRSWAP)
CSRSET
CSRCLR
TLBCLEAR
TLBFLUSH
CHECKADDR

This exception will set…
csr_prevpc = the address of the offending instruction
csr_bad = the offending instruction
csr_addr = 0

Page-Related Exceptions

Page Illegal Address Exception	*Attempt to access kernel space in User Mode*
Page Table Exception	*Bad **csr_pgtable***
Page Invalid Exception	*VALID bit = 0 either at level 1 or 2*
Page Write Exception	*Write to a page which is not marked writable*
Page Fetch Exception	*Fetch from a page not marked executable*
Page Copy-on-Write Exception	*Page is not dirty and marked copy-on-write*
Page First_Dirty Exception	*Writing to page which is not marked dirty*

The page-related exceptions occur when the Memory Management Unit (MMU) has a problem translating a virtual address to a physical address. They are discussed more fully later.

A page-related exception can occur during the FETCH phase of execution, whenever an instruction is read from memory. It can also occur during LOAD and STORE instructions. These situations are the only ones that can cause a page-related exception.

Page-related exceptions may be caused by code running in either user mode or kernel mode.

For code running in user mode, all program-generated addresses (whether FETCH, LOAD, or STORE) should be in the upper, virtual address range. In other words, all program-generated addresses should have bit [35] set to 1. Any problem with an address will cause one of the page-related exceptions.

For code running in kernel mode, addresses in the physical address region (that is, within the lower 32 GiBytes, i.e., addresses with bit [35] cleared to 0) will never cause page-related exceptions. Addresses in upper virtual address range will cause the exact same page-related exceptions they would cause if executed in user mode.

Thus, address translation and page-related exceptions work the same for both user code and kernel code, with one difference: Program-generated addresses in the physical address region are perfectly okay for kernel code, but will cause a Page Illegal Address Exception in user mode.

The **csr_prevpc** is set to the address of the instruction causing the problem. The **csr_addr** is set to the program-generated address that caused the exception. (In the case of a problem with fetching, both CSRs will contain the same value.[23])

The general purpose registers will be unchanged. The assumption is that, in many cases, the kernel trap handler will fix the memory problem and execution will be resumed, starting with the faulting instruction which will be re-executed.

These exceptions will set…

- **csr_prevpc** = the address of the offending instruction
- **csr_bad** = 0
- **csr_addr** = the program-generated address causing the problem

Debug Exception

This exception is used in debugging programs.

This exception is caused by the execution of the DEBUG instruction. If instruction execution is resumed, it will occur after the DEBUG instruction.

[23] For example, if a SYSRET instruction loads a bad value into the PC, then **car_prevpc** will contain that bad address, not the address of the SYSRET instruction.

The DEBUG instruction uses only the OP1 and OP2 fields. The remaining 16 bits of the instruction [15:0] are left undefined and may be used by software to store additional information.

This exception will set…

csr_prevpc = the address of the instruction after the DEBUG instruction
csr_bad = the offending instruction, i.e., the DEBUG instruction itself
csr_addr = 0

Breakpoint Exception

This exception is used in debugging programs.

This exception is caused by the execution of the BREAKPOINT instruction. If instruction execution is resumed, it will occur by attempting to re-execute the instruction.

It is assumed that a BREAKPOINT instruction replaces some other valid instruction. After the breakpoint is reached, the BREAKPOINT instruction will be removed and the original instruction will be restored. After execution is resumed, the restored instruction will be executed.

The BREAKPOINT instruction uses only the OP1 and OP2 fields. The remaining 16 bits of the instruction [15:0] are left undefined and may be used by software to store additional information.

This exception will set…

csr_prevpc = the address of the BREAKPOINT instruction
csr_bad = the offending instruction, i.e., the BREAKPOINT instruction itself
csr_addr = 0

Singlestep Exception

The purpose of this exception is to allow debugging software to single-step execution, that is, to execute a single instruction of the target program and then regain control.

Whenever single-stepping is turned on (i.e., when the Singlestep bit in **csr_status** is set to 1) then a Singlestep Exception will be signaled following the execution of any instruction, as long as interrupts were enabled during the instruction execution and no other exceptions were signaled.

This exception is described more fully in a later section.

This exception will set…

csr_prevpc = the address of the next instruction to be executed
csr_bad = the instruction just executed
csr_addr = 0

The **csr_prevpc** is set to the address of the next instruction to execute after the instruction that caused the exception. For example, if a jump instruction causes the exception, **csr_prevpc** will be set to the jump target address.

Emulated Instruction Exception

This exception is caused by an attempt to execute a machine instruction which is defined but not implemented. The hardware will set **csr_bad** to the instruction that is not implemented, as it was fetched from memory. The **csr_prevpc** will be set to the address of the instruction following the unimplemented instruction.

The following instructions are candidates for emulation. The algorithms are complex and will require complex hardware. It may be preferable to perform these operations in software, especially on smaller, simpler implementations of the Blitz-64 architecture.

DIV, REM
All floating point instructions

Emulated instructions may be used in either user mode or kernel mode.

This exception will set…

csr_prevpc = the address of the following instruction
csr_bad = the offending instruction
csr_addr = 0

Hardware Fault Exception

Some implementations of the Blitz-64 ISA will include circuitry that detects errors. For example, a core might include circuitry for:

- Additional error checking bits for register contents
- Additional error checking bits for main memory data
- Additional error checking bits for bus data
- Duplication of ALU circuitry, to catch errors

When circuitry such as the above detects that an error has occurred, a Hardware Fault Exception will be triggered.

Presumably a single hardware error will directly affect only the thread in execution. That thread can no longer be considered reliable and correct. Presumably, the kernel will never ignore a hardware fault. Instead the kernel might take actions such as:

Log the error
Notify the affected thread
Abort the thread
Restart the affected thread

This exception will set…
csr_prevpc = the address of the offending instruction
csr_bad = the offending instruction
csr_addr = 0

A hardware error may be "transient" in which case it was a one-time event and there will be no further malfunctions in the core. Or it may be an ongoing problem and the same error will be detected again in the future, whenever a similar operation is performed. The error may also be the result of a physical insult, such as the power supply falling below specifications. In such cases, we might encounter an increasing number of hardware faults with total failure being imminent.

In some cases, the "**error detection and correction**" (EDC) codes will be used. Such codes are capable of not only detecting that a bit has erroneously flipped in value, but also of determining which bit is in error, thus allowing the bit to be corrected.

Commentary In some ISAs, hardware faults are treated like asynchronous interrupts. However in Blitz-64, a hardware fault causes an *exception,* not an *interrupt*.

Interrupts can and are masked at various times. However, hardware faults should never be masked. Also, exceptions are tied to a particular instruction; this may be needed for hardware faults, since the kernel may need to identify which thread was executing at the moment the fault was detected so that the kernel can abort the thread.[24]

In the case when a hardware fault is persistent and recurs repeatedly, what will happen? A second Hardware Fault Exception will occur on the heels of the first Hardware Fault Exception, before the trap handler has completed. Since the exception handler is unable to complete before getting restarted, presumably the core will go into an infinite loop and freeze up.[25]

Bad Array Index Exception

This exception can be caused by the following instructions:

INDEX0, INDEX1, INDEX2, INDEX4,
INDEX8, INDEX16, INDEX24, INDEX32

The purpose of these instructions is to verify that an array index is legal and within range and cause this exception if there is a problem. Presumably the software will react to this exception by printing a message to the effect that there was an array index error.

This exception will set…
csr_prevpc = the address of the INDEX__ instruction
csr_bad = the INDEX__ instruction
csr_addr = 0

Commentary There is no "Illegal Address" exception.

All upper bits [63:36] in a program-generated address (i.e., above the normal 36 bits in every address) are ignored.

If a program-generated address is in the upper half of the 36 bit address range, the address goes through the Memory Management Unit (MMU) which performs virtual-

[24] This would be appropriate for a one-time, non-recurring fault which only impacts a single user-mode thread.

[25] We considered a design in which hardware exceptions could be masked in an attempt to prevent such looping. However, the added complexity doesn't seem worth its cost.

to-physical address translation. If there is a problem with the address, one of the page-related exceptions will be generated.

If the address is in the lower half of the 36 bit address range, the address is a physical address and will be used without translation.

Regardless of whether any address translation was performed, the full 35 bit physical address is sent to the main memory and the memory-mapped I/O devices “as is”.

Physical address violations (i.e., attempts to access an uninstalled address in the physical memory region) are not checked and the consequences are undefined. If an attempt is made to access uninstalled memory, then writes are likely to be ignored and reads are undefined, and likely to return garbage values. In any case, no exception will occur. It is the kernel’s responsibility to access only installed memory and defined memory-mapped I/O addresses.

Stack Overflow Exception

The Stack Overflow Exception can be caused by the following instructions:

ADD, ADDI, ENTERFUN

The exception is caused when one of these instructions attempts to store a value into the **sp** register which is less than the **StackLimit** field in **csr_status**.

If this exception occurs, the instruction will not modify the **sp** register.

This exception will set…
csr_prevpc = the address of the offending instruction
csr_bad = the offending instruction
csr_addr = 0

Philosophy Behind Checking the Stack Limit

In Blitz-64, the stack grows downward from high memory to smaller addresses. The purpose of the **StackLimit** field is to catch errors where the stack grows too much and exceeds the space allocated for it.

In one common arrangement, the OS will place the heap and the stack in the same, shared region of memory. The heap will grow up from the lower addresses and the stack will grow downward from higher addresses. When they meet, memory is exhausted. The stack limit check makes it possible to catch this error.

In multithreaded programs, each thread will require its own stack. Typically, the space for each stack will be preallocated and each stack will be limited to a region and incapable of growing further. In fact, each stack may be located within a single object that is allocated on the heap. Thus, a comparison against the heap limit pointer will not work.

In a virtual memory system, one typical practice is to allocate each thread's stack "very" far apart in the address space, thus hoping to avoid the possibility of a stack out-growing its region. But "hope" is not a viable strategy for systems requiring high-reliability.[26]

In a smaller embedded application, there might be no virtual memory and the heap and the stack will occupy a not-huge region of memory. A stack-heap collision is possible.

A stack can outgrow its allocated region either because the region was just too small to begin with or because an algorithm recurses too deeply. In any case, when a stack outgrows its region without triggering an error, it typically causes the overwriting of some other, random memory locations. Often, the stack will subsequently shrink after doing this damage and the problem will, in some sense, disappear. The overwritten memory will cause a failure at a later time in a some completely unrelated section of code or — what's worse — the program's output will simply be incorrect. Errors from stack overflow are insidious since the site of the failure is usually unrelated to the function causing the problem.

The Blitz-64 philosophy is to try to catch all errors, whenever reasonably possible. To be sure, checking the **sp** register against a limit requires a little more hardware,

[26] In this approach, a buffer (also called a "sentinel") page is allocated for each stack. This page will be marked as invalid or unwritable. The idea is that if the stack grows into this page, a page fault will occur and the kernel can take appropriate action. To be 100% reliable, this technique would place a requirement that writes to the stack must not be spread too far apart. Otherwise, a really large stack allocation might fail to cause the expected page fault. This technique works with high reliability for stacks in virtual space but can be problematic for stacks allocated dynamically in the heap or in systems that do not use virtual memory.

although not much. The overhead in execution speed is expected to be negligible. We recognize this, but believe the benefit exceeds the cost.

When checking the value to be stored in the **sp** register, the core will only look at the lower 36 bits; the upper bits of **sp** are ignored. Both **sp** and the **StackLimit** field are treated as unsigned 36 bit values and the exception occurs when **sp < StackLimit**.

If no limit enforcement is desired, a value of 0 can be stored in the **StackLimit** field in **csr_status**. This will prevent the exception from occurring.

Details on Stack Limit Checking

In typical code, the stack is only used for stack frames. A stack frame is allocated at the beginning of every function or method and the frame is removed from the stack when the function or method returns.

The "prologue" code at the beginning of a function or method will allocate a frame by adjusting the **sp** register using one of these instructions:

```
ADD
ADDI
ENTERFUN
```

As such, it is only necessary to check for a stack overflow condition when these instructions are executed.

The Blitz-64 specification requires that the condition is checked whenever one of these instructions modifies the **sp** register. It would seem that checking at other times might be reasonable, but this is not allowed.[27]

It may be that the OS kernel will respond to a Stack Overflow Exception by enlarging the stack region, adjusting the **StackLimit** within **csr_status**, and then re-executing the instruction that caused the exception. For this reason, we want to ensure that the

[27] We considered an alternative ISA design in which the limit check was performed on the STORE.x instructions. The idea was to check any STORE.x instruction that uses the **sp** register, comparing the target memory address against the **StackLimit**. However, the compiler can generates code to address data on the stack in which the final STORE.x instruction uses another register besides **sp**. Thus the design adopted is expected to be more reliable.

exception is only possible under known conditions, that is, under the conditions described here.

The ADD, ADDI, and ENTERFUN instructions can be used to add a (negative) value to **sp**, triggering the exception. It is conceivable that the addition operation will trigger both a Stack Overflow Exception as well as an Arithmetic Exception. Adjusting **sp** should never result in an overflow, so the Arithmetic Exception will take priority and the Stack Overflow Exception will be lost.

In the case of ENTERFUN, the instruction normally stores the return address (from the **lr** register) into memory. If the instruction causes a Stack Overflow Exception, this memory operation must not occur. If there are exceptions related to the store operation (such as page-related exceptions), the Stack Overflow Exception takes priority and the other exceptions are ignored and lost.

Occasionally, code will store values above the stack top (i.e., using negative offsets from **sp**). This is frowned upon, since this might store data beyond the **StackLimit**. The stack limit is not checked when data is stored, so this error would not be caught.

The Singlestep Exception

Programmers may want to debug user mode programs with a "debugger" which will allow them to examine variables and execute instructions in a controlled manner. The single-stepping facility is designed for use by such a debugger.[28]

The status register **csr_status** contains a single bit named SINGLE_STEP. When set to 1, there will be a "Singlestep Exception" signaled immediately after the completion of the next instruction. When cleared to 0, no such exception will be signaled.

[28] This approach described here requires the execution of privileged instructions, such as modifying the SINGLE_STEP bit in **csr_status**. We make no assumptions about whether the debugger is running as a separate process, or running within the virtual address space of the target program as a separate thread, or whether the debugger code is entirely integrated with the kernel code. If the debugger is not integrated with the kernel code, then the debugger process will need to make specific requests of the kernel to perform the privileged operations and take the actions discussed in this section.

The Singlestep Exception will only occur if the previous instruction was executed with interrupts enabled and no other exceptions were generated by the instruction.[29]

The purpose of this exception is to allow a debugger to execute a single instruction of the target code and then regain control immediately afterward.

In order for the debugger to execute a single-step operation, the debugger will execute a SYSRET instruction in which the new value of **csr_status** has SINGLE_STEP = 1. A "return" (which you can think of as a "jump") will be effected to the code sequence being debugged and a single instruction will be executed.

After that instruction, a Singlestep Exception will be signaled. The Singlestep Exception has a priority below all other exceptions. If the instruction causes another exception, the Singlestep Exception will not occur and will be effectively ignored and lost.

Assuming that no other exceptions occurred for that instruction, the Singlestep trap handler will be invoked and the debugger will regain control.

If the target instruction caused another exception, then the Singlestep Exception will not occur. It is assumed that the debugger will regain control through the trap handler for whatever other exception occurred.[30]

Normally, interrupts will be disabled at the time the SYSRET instruction is executed, so the SYSRET will not itself cause a Singlestep Exception. (However, if a SYSRET is executed with interrupts disabled — a buggy scenario — a Singlestep Exception will occur.)

[29] More precisely, a Singlestep Exception may only occur after an instruction for which interrupts were enabled *directly prior to* instruction execution. This distinction makes a difference for a couple of privileged instructions which may alter the INTERRUPTS_ENABLED bit.

[30] If the other exception was due to a programming error (e.g., a Null Address or Arithmetic Exception), then the kernel should deliver this information to the debugger, along with the address of the instruction causing the exception, so the debugger can report it. But the other exception might not indicate a program error. For example, some STORE instruction might cause a Page First Dirty Exception, which would be handled by the kernel by updating the in-memory page table. The kernel can then return to the user code, which will naturally re-attempt to execute the instruction. On the second execution, the singlestep exception will finally happen, and the debugger can be notified.

What about the presence of interrupts occurring around the time of a single-step operation?

The trap handler for the Singlestep Exception will run with interrupts disabled. The timing of an incoming asynchronous interrupt determines whether it will be handled before the Singlestep Exception handler runs or whether it must wait until after the Singlestep Exception handler completes.

Interrupts are disabled before and through the execution of the SYSRET instruction. An interrupt may have been signaled, but the interrupt will remain pending until the SYSRET instruction is executed.

Immediately after the SYSRET instruction is executed and interrupts are re-enabled, a pending interrupt X may exist. Interrupt processing effectively occurs *between* the execution of instructions, not *during* them. After the SYSRET, but before the next instruction, the hardware will initiate trap processing to invoke the trap handler for interrupt X. The Singlestep Exception will not occur, since no instructions were executed in user mode before code in the interrupt handler runs.

Presumably, all interrupt handlers will save **csr_status** and, upon completion of the trap handler, the handler will restore it (with its own SYSRET instruction). This time — assuming no more interrupts are pending — a single instruction will be executed and the Singlestep Exception will finally occur.

Because the Singlestep Exception cannot occur when interrupts are disabled, it is impossible to single-step though trap handers, using the Singlestep Exception mechanism.

A Singlestep Exception will never occur immediately after a SYSCALL instruction, since the Singlestep Exception is overridden by the SYSCALL Exception. This makes it moderately tricky to perform single-stepping at a SYSCALL. A Singlestep Exception may occur directly before the SYSCALL, but the next opportunity will not be until after the instruction following the SYSCALL completes execution. Of course, the trap handler for SYSCALL may be aware of the presence of a debugger and the single-stepping activity.

There is also an issue with emulated instructions. If an instruction (e.g., FMUL) causes an Emulation Exception, it cannot also cause a Singlestep Exception. Thus, the FMUL instruction will invoke a trap handler which will return to the instruction following the FMUL, call it X. The Singlestep Exception will occur after the execution of instruction X. There will be no Singlestep Exception associated with the FMUL instruction.[31]

Instructions that cause page-related exceptions should not present a single-stepping problem. Typically, an instruction (e.g., LOAD or STORE) will cause a page-related exception. After the trap handler deals with the problem, the instruction will be re-tried. The Singlestep Exception will then occur after the second attempt succeeds with no exception.

Value of Saved PC

During trap handling, the hardware will begin by saving the program counter (PC) in **csr_prevpc**. This allows the trap handler software to locate the instruction causing the trap and, in many cases, to resume execution of the interrupted code upon completion of the trap handler function.

For the following trap types, the address of the instruction causing the trap is saved.

In the case of some exceptions, the SYSRET instruction at the end of the trap handler will resume execution by attempting to re-execute the offending instruction again. In the case of other exceptions, the instruction has a fatal problem that requires debugging. In either case, pointing to the offending instruction makes sense.

[31] Perhaps the Emulated Instruction Exception will check to see if the interrupted code is actively being debugged. If so, the handler can end by “faking” a Singlestep Exception, rather than simply returning to instruction X. We discuss “faking” an exception elsewhere in this document.

csr_prevpc points to offending instruction:

Exceptions

- Arithmetic Exception
- Unaligned LOAD/STORE Exception[32]
- Null Address Exception
- Illegal Instruction Exception (including privilege violation)
- Page Illegal Address Exception
- Page Table Exception
- Page Invalid Exception
- Page Write Exception
- Page Fetch Exception
- Page Copy-On-Write Exception
- Page First Dirty Exception
- Breakpoint Exception
- Hardware Fault Exception
- Bad Array Index Exception
- Stack Overflow Exception

Interrupts

- *… all interrupt types …*

For the following trap types, the address to be saved in **csr_prevpc** will be the next instruction to execute. It is assumed that the previous instruction executed to completion and re-executing that instruction would be in error. The SYSRET instruction will resume by executing the following instruction.

csr_prevpc points to the following instruction:

Exceptions

- SYSCALL
- Debug Exception
- Singlestep Exception
- Emulated Instruction Exception

[32] For an Unaligned LOAD/STORE Exception, **csr_prevpc** will point to the LOAD or STORE instruction. If this exception is to be treated as an error, then pointing at the instruction causing the problem makes sense. But if this exception is handled by emulating the operation, then the emulation handler will need to increment PC so that, on execution of SYSRET, the same instruction will not be re-executed, causing an infinite chain of exceptions.

Traps Related to Instruction Fetching

The following instructions can modify the Program Counter (PC)[33]:

B.EQ
B.NE
B.LT
B.LE
JAL
JALR
SYSRET

Keep in mind that a number of synthetic instructions (such as JUMP, JR, CALL, RET, BEQ, …) are translated into one of the above instructions.

Furthermore, trap processing will modify the PC by copying **csr_trapvec** into the PC. Sequential program execution also modifies the PC by incrementing it.

Whenever an instruction is fetched, one of the following exceptions may arise:

Null Address Exception
Page Illegal Address Exception
Page Table Exception
Page Invalid Exception
Page Fetch Exception

Since compressed instructions may be as short as a single byte, there is no alignment requirement for instructions. Therefore, the Unaligned LOAD/STORE Exception cannot occur.[34]

The Null Address Exception will occur at the time the PC is loaded, by the jumping instruction. The "offending instruction" is the jumping instruction itself.

The remaining exceptions (that is, the page-related exceptions) occur when the memory operation to fetch the instruction is performed. As such, the "offending instruction" is the instruction being fetched. No information about where the jump "came from" is captured.

[33] We ignore power-on-reset and the RESTART instruction, since exceptions do not occur then.

[34] In earlier versions of the ISA, there was an alignment requirement on instructions.

In some cases, the Null Address Exception may not be detected until the instruction fetch occurs. For example, if a trap occurs at a time when **csr_trapvec** happens to be null, the problem is really that someone failed to load **csr_trapvec**. In such a case, there is no identifiable "offending instruction". As another example, a user-mode program might take a branch to some random address in a page that is invalid.

In such cases, **csr_bad** and **csr_addr** may not be set to the values that were mandated in the above description of the Null Address Exception.

As another example, a CALL instruction (i.e., JALR) could load the PC with a problematic PC that results in an exception during the fetch. Normally, when exceptions occur, the offending instruction will be completely aborted and have no effect. However, in the case of the JALR, the return address may be saved (e.g., register LR will be modified) before the exception is discovered.[35]

Trap Priority and Simultaneous Exceptions

The occurrence of an interrupt or exception will invoke hardware trap processing, which initiates the execution of a software trap handler. Conceptually, hardware trap invocation occurs *between* the execution of instructions; it is not done concurrently with instruction execution (at least as far as functionality observable by software).

More precisely, interrupts are checked for *before* each instruction is executed and, if triggered, hardware trap invocation occurs *prior* to the instruction execution. On the other hand, exceptions are checked for *during* the execution of instructions and, if triggered, hardware trap invocation occurs *after* instruction execution is terminated.

If an interrupt is pending before an instruction X begins execution, then the hardware interrupt processing will occur immediately. This will cause a change in the flow of control and the next instruction to execute will be the first instruction of the interrupt handler. Instruction X will be delayed and will not be executed until the interrupt handler completes and ends by executing a SYSRET instruction.

[35] If the kernel repairs the problem and re-executes the instruction, there will be no harm done by moving the return address into the LR register a second time.

On the other hand, if the interrupt arrives a little later, then instruction X will execute. If instruction X causes an exception, then that exception will cause trap handling. The next instruction after X will be the first instruction of the trap handler for that exception. The interrupt will not invoke a trap handler and the interrupt will remain pending.

As a result of the exception and trap processing, the hardware will clear the INTERRUPTS_ENABLED bit in **csr_status** to disable interrupts. Therefore, the handler for the interrupt will not run until after the trap handler for the exception completes.

At some later time, the trap handler for the exception will end by executing a SYSRET instruction. At this time, interrupts will become re-enabled. As a result, the trap handler for the interrupt will be invoked immediately after the SYSRET instruction and before any instruction in the original, interrupted code sequence is executed.

Commentary Conceptually, interrupt processing occurs before the execution of an instruction, and exception processing occurs during and after the processing of an instruction.

It may seem that our model somehow gives priority to exceptions over interrupts, but this is not necessarily accurate. In fact, any core will check for and accept interrupts at only certain moments in execution. During other times, instruction execution will occur.

Machine instructions atomic, in the sense that instructions either execute completely or do not execute at all. In other words, the instructions following an instruction causing an exception are not executed at all. The Blitz-64 architecture requires that any partial or incomplete instruction execution (for example, instructions further ahead in the pipeline) shall not be visible to the programmer.

The Blitz-64 model does not preclude a pipelined implementation in which interrupts are accepted and processed with alacrity. For example, a pipelined implementation might accept and process interrupts immediately, even though there are several instructions at varying stages of the pipeline. When an interrupt arrives, trap processing would be begin immediately, which will force the emptying of whatever is in the pipeline and an immediate switchover to the trap processing sequence. However, any and all partially executed instructions must be abandoned and any possible effects must be avoided or undone.

On the other hand, an implementation may delay an interrupt for several cycles, in order to allow all instructions currently in the pipeline to complete execution. The key constraint imposed by the ISA is that the interrupt processing must occur discretely between two instructions. The instruction before the trap handling will complete fully[36] and all instructions after the interrupt must not begin execution.

Our model gives only a semantics for interrupt acceptance and processing. Any Blitz-64 implementation must have the the same behavioral result as a simple, non-pipelined implementation in which instructions are executed serially, one after the other, and interrupt processing occurs between two instructions.

It is possible that some instruction will cause more than one exception. Only one exception will be signaled. All other other exceptions for that instruction will be ignored and forgotten.

For example, consider a LOADD instruction attempting to load from address 1. Both the Unaligned LOAD/STORE Exception and the Null Address Exception apply. For such a conflict, the Null Address Exception shall occur and the Unaligned LOAD/STORE Exception shall be ignored.[37]

[36] This includes exception processing associated with the instructions. If the previous instruction causes an exception, then trap handling for that exception will occur. Trap handling for the interrupt will be delayed and the interrupt will remain pending.

[37] In this example, the decision about precedence is more-or-less arbitrary. While an Unaligned LOAD/STORE Exception might invoke emulation, a Null Address Exception always indicates a program bug. For this reason, we chose to give the Null Address Exception priority. However, this distinction only affects error reporting and should not affect correct code.

Here is a summary of how multiple simultaneous exceptions are handled:

Highest priority → Hardware Fault Exception

Illegal Instruction Exception (including privilege violation)
Debug Exception
Breakpoint Exception
Syscall
Arithmetic Exception
Emulated Instruction Exception
Bad Array Index Exception
Stack Overflow Exception

Null Address Exception
Unaligned LOAD/STORE Exception

Page Illegal Address Exception
Page Table Exception
Page Invalid Exception
Page Write Exception
Page Fetch Exception
Page Copy-On-Write Exception
Page First Dirty Exception

Lowest priority → Singlestep Exception

The following rules apply when there are multiple exceptions and interrupts.

- If an instruction causes more than one exception, then only one exception will be chosen for trap processing. All other exceptions will be lost.

- Exceptions have priority over interrupts. If an exception is signaled, any and all pending interrupts will remain pending and the exception will be chosen for trap processing.

- If a “Hardware Fault Exception” is signaled, it will have the next highest priority. Any and all exceptions of lower priority will be ignored and lost.

- A “Null Address Exception” overrides an “Unaligned LOAD/STORE Exception” and all Page-Related exceptions.

- An “Unaligned LOAD/STORE Exception” overrides all Page-Related exceptions.

- The following exceptions are all mutually exclusive and cannot co-occur with other exceptions types:[38]

  Illegal Instruction Exception
  Debug Exception
  Breakpoint Exception
  Syscall
  Arithmetic Exception
  Emulated Instruction Exception
  Bad Array Index Exception
  Stack Overflow Exception

- A “Page Illegal Address Exception” overrides all other Page-Related exceptions.

- The following Page-Related exceptions are mutually exclusive; at most only one of these can occur.

  Page Table Exception
  Page Invalid Exception
  Page Write Exception
  Page Fetch Exception
  Page Copy-On-Write Exception
  Page First Dirty Exception

- During an instruction FETCH a Page-Related exception can occur. However, if such an exception occurs, then the instruction is not fetched and instruction execution is not begun. Any other exception that the instruction might have caused never happens.

- A “Singlestep Exception” has a priority below all other exceptions. If, for example, the debugger is single-stepping some code and an ADD instruction causes an “Arithmetic Exception”, the “Arithmetic Exception” will be chosen.

[38] Except “Hardware Fault Exception”, which has higher priority, and “Singlestep Exception”, which has lower priority.

- Interrupts have the lowest priority and will only be handled if there are no exceptions.

- If there are several interrupts pending, then one will be chosen for processing. That interrupt will cause trap processing and the others will remain pending.

Commentary The logic behind choosing one exception and ignoring all others is this.

A Hardware Fault Exception has the highest priority since it precludes correct operation of at least some code. The hardware fault may be a transient, one-time event, in which case the kernel might get by with only terminating a single thread; the kernel itself might be able to continue. For example, a Hardware Exception occurring in user mode code might be dealt with by simply terminating the affected process; it is not necessarily a cause for crashing the kernel. However, a Hardware Exception occurring in when in Kernel Mode is more serious (compromising the kernel itself) and is probably a good reason for a kernel crash / reboot.

A Page Illegal Address Exception and a Page Table Exception conflict would occur if user code attempts to access kernel memory at the same that the **csr_pgtable** register is uninitialized. However, it is impossible to FETCH an instruction from user space without a page table, so this conflict could only arise when the kernel jumps to user code with both **csr_pgtable** and the PC being invalid. Thus, the Page Illegal Address Exception is given higher priority. A Page Illegal Address Exception cannot co-occur with any other Page-Related exception.

The decision about precedence between the Null Address Exception and the Unaligned LOAD/STORE Exception is more-or-less arbitrary. While Unaligned LOAD/STORE Exceptions might sometimes be legitimate (invoking emulation), a Null Address Exception always indicates a program bug. For this reason, we chose to give the Null Address Exception priority.

Pending Interrupts

Once an interrupt is signaled by a device, it becomes "pending" and remains pending until it is accepted for trap processing.[39] At the time it is accepted, the program counter (PC) is set to **csr_trapvec**, the status word is saved in **csr_stat2**, an interrupt code is loaded into **csr_cause**, and the first instruction of the trap handler will be executed next.

If the previous instruction caused an exception, any interrupt that occurs will remain pending during the execution of the trap handler for the exception. The only information that must be kept is the identity of the interrupt type, i.e., the fact that an interrupt is pending.[40]

With this approach, an interrupt can only be serviced after the execution of an instruction which causes no exception. Note that the SYSRET will not cause an exception. Thus, immediately after the trap handler returns, a pending interrupt will be handled, before the next instruction is executed.[41]

Any instruction which causes an exception will invoke a trap handler and during the entire execution of the handler, interrupts will be disabled. If an interrupt arrives early enough — that is, before the excepting instruction begins execution — the interrupt will be handled before the exception's trap handler runs. On the other hand, if the interrupt happens to be signaled a little bit later, it will miss its window of opportunity and will be delayed until after the exception's trap handler completes and re-enables interrupts.[42]

[39] Some pending interrupts can disappear, such as an interrupt from the Platform-Level Interrupt Controller (PLIC) that has been claimed by another core.

[40] In the case of a conflict between an exception and interrupt occurring simultaneously, we considered a design in which the exception is made pending and the interrupt is handled. However, an exception involves more information (such as the values of the Program Counter, **csr_cause**, **csr_bad**, and **csr_addr**). Instead of keeping this info, the exception is handled and the interrupt remains pending.

[41] Well, a SYSRET might theoretically cause an exception if it is executing in a virtual address space and a page-related exception arises for the FETCH. But kernel code should never be placed in virtual pages that are not pinned and exception-proof, and probably not even then, so this is not expected to occur.

[42] This is a simplification. A real-world kernel will sometimes reenable interrupts before returning to the interrupted code sequence. The point is that the pending interrupt will occur at the moment interrupts are re-enabled.

To summarize, with the design of Blitz-64, an interrupt occurring during an exception-causing instruction is effectively treated as if it came a little later and simply missed its window of opportunity. It will get serviced immediately after the trap handler finishes and reenables interrupts, which is what must happen anyway if the interrupt had arrived one instruction later.

Commentary We considered a design in which a trap handler for one exception or interrupt could, itself, be interrupted. For example, a scheme with multiple levels of interrupt execution might provide more responsiveness for some interrupt types. Such a design might be necessary to handle interrupts that always require a immediate, super-fast response.

Such was the case in the old days, when the CPUs were relatively slow and they had to manage a rotating disk drive directly. The CPU was interrupted when the disk platter rotated into position and data was ready to be transferred. The CPU needed to pay attention quickly and, if it failed to, the disk would continue rotating and the opportunity for data transfer would be lost. A similar situation arose with communication links, where an incoming message had to be moved into memory as the bits arrive, or else the message was lost.

We've come a long way and interrupting devices generally have their own controllers. Device-specific controllers handle most of the time-critical operations of peripheral hardware and allow the primary core to merely transfer data and high-level logical commands back and forth to the controller. Nowadays, the interrupt primarily serves the function of letting the core know that the some peripheral operation is complete and the core is now free to make use of the results. In other words, interrupts now serve primarily to send information to the core, allowing the core to take action when it is ready, not to demand the core perform some time-critical action.

Blitz-64 is designed to be a general purpose operating system core, not a microcontroller. Accommodating interrupts during trap handling would require the addition of another level of complexity, perhaps necessitating a third mode or a trap priority scheme or a mechanism for selectively masking interrupts. This is just too complicated. The simplicity and limitations of the Blitz-64 interrupt mechanism are an intentional and logical manifestation of the overall project objectives.

That said, it is critical that all trap handlers keep interrupts disabled for as short a time as possible. The expectation is that any trap handler unable to return quickly will do something — such as simply signal a semaphore in order to wake up another thread or immediately go to sleep, waiting on some condition. In any case, the handler will effectively branch into to the scheduler to resume execution of user threads as quickly as possible, thereby re-enabling interrupts in short order.

Delegation to User Mode Error Handlers

Typically, when exceptions occur in user code (such as "Illegal Instruction Exception"), the kernel will abort the process without further ado. However, there is the possibility that some exceptions will be handled a little differently by invoking a "**user mode error handler**".

There is no support for this **user mode exception delegation** in the hardware; the delegation is handled entirely by the kernel software. When an exception occurs in user mode code, the kernel will get control through the trap handling mechanisms discussed above. The kernel may then, at its option, notify the user process in some way or another. This notification is entirely a software operation since there is no special hardware involved.

We specifically use the term "error handler" rather than "trap/exception/interrupt handler", since the mechanisms are quite different.

The following types of exception are candidates for software delegation to user mode code, when they arise in that code:

- Hardware Fault Exception
- Arithmetic Exception
- Illegal Instruction Exception
- Null Address Exception
- Bad Array Index Exception
- Unaligned LOAD/STORE Exception
- Privilege Exception
- Bad Array Index Exception
- Stack Overflow Exception
- Debug Exception
- Breakpoint Exception
- Singlestep Exception

- Page-related exceptions, under certain conditions such as:
    - — Attempt to access an unallocated page
    - — Attempt to STORE to a read-only page
    - — Attempt to FETCH from a non-executable page

After the exception occurs and the corresponding trap handler is invoked, it will see that the processor was running in user mode when the exception occurred. It will then return to the user code but will modify the PC to cause a forced jump to the user error handler’s address.

The kernel may maintain flags associated with each address space so the kernel mode trap handler can optionally either (1) abort the process, or (2) pass control to the user mode error handler.

Then, the user code will presumably invoke the “throw-error” sequence in the KPL language. But if nothing else, the user code can simply abort the offending thread. In any case, this mechanism makes possible the creation of user mode programs that can address their own bugs, including the support of debugging and error reporting, and potentially fault tolerance and error recovery. Support for error reporting and/or debugging will likely be included in the shared core function library, as so will be available to all processes at no extra cost.

Trap Processing and Handler Startup

When a trap is processed, the hardware will take a number of simple, fixed actions. These actions are performed between the execution of the previous machine instruction and the first instruction of the global trap handler.

Interrupts are either “**enabled**” or “**disabled**” and this is controlled by the INTERRUPTS_ENABLED bit in the status register **csr_status**.

When an interrupt is signaled, it remains pending until the hardware invokes trap handling. Interrupts may not be masked, other than by the INTERRUPTS_ENABLED bit in the **csr_status**. Pending interrupts remain pending while interrupts are disabled. Once interrupts are re-enabled, one interrupt will be selected and the corresponding trap handler is invoked.

Exceptions cannot be masked: If an instruction causes an exception, then hardware trap invocation will be immediately performed.

When a trap handling is invoked, the hardware will perform the following actions:

Invocation (Hardware Phase):

```
csr_prevpc ← PC
PC ← csr_trapvec
csr_stat2 ← csr_status
csr_status [INTERRUPTS_ENABLED] ← 0
csr_status [KERNEL_MODE] ← 1
csr_status [SINGLE_STEP] ← 0
csr_cause ← <trap code>
csr_bad ← <additional trap info; e.g., the offending instruction>
csr_addr ← <additional trap info; e.g., the virtual address>
```

The PC is copied to **csr_prevpc** so that it can be saved by the trap handler software so that after the trap handler finishes, a return can be made to the interrupted code.

The PC is loaded with the value in **csr_trapvec**, the address of the global trap handler.

The **csr_status** is copied to **csr_stat2**. If a return is made to the interrupted code, the status register will need to be restored.

Interrupts are disabled and the mode is switched to "kernel mode". The global trap handler begins with interrupts disabled, since it needs to perform operations (such as saving the general purpose registers and some CSRs) which cannot be interrupted. Some individual trap handlers may choose to re-enable interrupts. However, the final sequence of returning to the interrupted code (ending with a SYSRET instruction) must be performed with interrupts disabled.

Single-stepping is turned off. Since the global trap handler runs with interrupts disabled, this is not strictly necessary, but is a convenience for those individual trap handlers which will re-enable interrupts during their execution.

A code indicating the type or nature of the trap is written to **csr_cause**. The global trap handler is expected to use this information to dispatch to the individual trap handlers.

In the case of several exceptions, additional information is written to **csr_bad** and **csr_addr**. For example, during a page-related exception, the program-generated virtual address causing the problem in written to **csr_addr**.

Saving State During Thread Switching

When trap processing occurs as a result of an interrupt, an executing thread will be interrupted. Generally speaking, the trap handler will return to that thread after trap handling is complete. As such, the state of the general purpose registers must be saved and restored so that the interrupted thread is unaffected by the trap handler.

In some cases such as a timer interrupt, the registers associated with the thread will be saved for a longer period of time, and the return is made to a different thread. The interrupted thread will be delayed for quite some time while other threads are run.

Next, we discuss how the kernel software is expected to use the Blitz-64 hardware. This discussion motivates and explains the Blitz-64 ISA; it should not be confused with a description of any specific kernel code.

It is assumed that each thread will have an area of memory, which we call a **Thread Control Block (TCB)**, that will contain important information about the thread and that will be used to save the general purpose registers and other state during an interruption.

[In the discussion of page tables later in this document, we will see that the root page will be half used, leaving 8,192 bytes of unused and available space. This area is an ideal place in which to store information about the processes, perhaps including one or more Thread Control Blocks.]

When a thread is interrupted, the first thing that must be done is to save the registers and these will be saved in the Thread Control Block (TCB). We expect **csr_ptr** to be used to point to the Thread Control Block of the running thread.

For any thread running with interrupts enabled — whether in user mode or kernel mode — we assume that the following will contain valid state information:

The State of the Currently Running Thread:

csr_status — The status register
csr_ptr — A pointer to the TCB
csr_pgtable — A pointer to the root of the page table
csr_trapvec — Address of the trap handler to be used for traps
… all general purpose registers …

In this discussion, we make no assumptions about the other CSR registers. They are not assumed to contain state information and may be used as work registers by the trap handler. (But of course, this discussion is hypothetical. Kernel programmers may elect to use other CSRs as they see fit.)

Some threads will never use virtual memory, so **csr_pgtable** is not needed for them. Such threads — **kernel threads** — are not associated with any particular address space and have no use for a page table.

Other threads (which we call **user threads**) will have an associated virtual address space. They run in user mode and all addresses are translated from the virtual address space to the physical address space, with the assistance of the page table.

The Translation Lookaside Buffer (TLB) registers — if they exist — serve as a cache of Page Table Entries (PTEs). If there is no TLB, the page table must be walked on every access to memory. The TLB registers dramatically reduce the need to access the page table. For a running user mode thread, the TLB registers contain the **working set**, i.e., information about pages that have been recently used and can be expected to be needed in the near future.

Each TLB is tagged with an Address Space Identifier (ASID). At context switches, there is no need to flush the TLB since the ASIDs will function to distinguish the TLB registers associated with one process's address space from the registers for another process's address space. Only when pages tables are modified or deleted is there any need to flush the TLB.

When a context switch occurs and a new process begins execution, it is likely that the existing TLB registers used by the previous process will gradually be evicted and replaced with the working set of the newly executing process. As the new process begins execution, several walks of the page table will be necessary until the new process's working set has been loaded into the TLB.

However, if the TLB is large enough and there are not too many addresses spaces, it is possible that the working sets of several processes can all coexist in the TLB. In such a scenario, at context switches, evictions will be rare and the walking of the page tables will be reduced.

Consider what happens when a user thread makes a system call. The thread will be running in user mode before the SYSCALL and then will be running trap handler code in kernel mode. Since it is the same thread and since the TLB registers continue to contain important values, we will refer to this as a "user thread running in kernel mode". But regardless of what we call it, any code running in kernel mode will need to use its own stack. It must be careful not to rely on the correctness of any user mode register values and be careful to restore any user mode registers that were used. Of course information must not be allowed to not leak from the kernel back to the user mode code.

There are several cases to be considered:

<u>For user threads...</u>
- The thread performs a SYSCALL
- The thread causes an exception
- The thread is suspended by an interrupt

<u>For kernel threads...</u>
- The thread performs a SYSCALL
- The thread causes an exception
- The thread is suspended by an interrupt

<u>Upon completion of the trap handler...</u>
- The interrupted thread is resumed.
- The interrupted thread is suspended and another thread is scheduled.

Global Trap Handler — Dispatching and Return

Once the core has completed the hardware phase of trap invocation, the first instruction of the Global Trap Handler will be fetched and executed.

In this section, we sketch out algorithms and code sequences for how the Global Trap Handler might be coded. The goal is to give some idea about how the various architectural features of the Blitz-64 architecture might be used. We also want to get a rough idea of how many machine instructions are involved.

This discussion is speculative and kernel programmers may take a different approach.

We assume the Global Trap Handler is written in assembler and will invoke functions written in the KPL language. We look at handling syscalls (where the arguments are passed in registers) and we look at all other traps (where we assume that all user registers must be saved and restored before the SYSRET). We assume that interrupts will remain disabled for the duration of all handler code and each handler will terminate by returning to the interrupted code.[43]

First, let's look at a possible algorithm for the Global Trap Handler, which is invoked after any trap. Its duty is to save the state of the interrupted process and then dispatch to function that will handle the particular trap encountered.

Global Trap Handler - Algorithm

```
    // We assume the following have just been set by the hardware:
    //      csr_stat2, csr_prevpc, csr_cause, csr_bad, csr_addr
    // We also assume:
    //      csr_ptr points to the Thread Control Block (TCB)
    //      csr_pgtable points to a page table
    //      csr_trapvec points to this Global Trap Handler

    Save general purpose registers in the TCB:
        Swap csr_ptr with register r7.
        Save r12-r15 (i.e., "tp", "gp", "lr" and "sp")  in the TCB.
                (About 5 instructions)

    Determine if this is a SYSCALL Exception.
                (About 3 instructions, using only r12-r15)
```

[43] Realistically, a kernel will invoke the scheduler at timer interrupts, if not during other traps as well. So interrupts will be enabled at some point before return to the interrupted code, but this is beyond the scope of the discussion here.

If this is a SYSCALL Exception…

Prepare to execute a kernel-mode function written in high-level KPL:

Load r15 (the kernel stack pointer “sp”) from the TCB.
Load r12-r13 (i.e, “tp”, “gp”) from the TCB.
Copy r7 (the ptr to the TCB) back into **csr_ptr**.
(About 4 instructions)

Dispatch to the individual SYSCALL handlers:

Dispatch on **csr_cause**, i.e., jump through the trap vector to a KPL function.
(About 4 instructions, using register “t”, including the indirect jump)

Upon entry to the KPL function to handle the SYSCALL…

- Registers r1 … r6 contain arguments to the system function.
- Register r7 contains a pointer to the TCB.
- Registers r8-r11 (“t, s0, s1, s2”) are work registers.
- Registers r1 will contain a return value to the user code.
- Registers r2-r11 should be zeroed before return.
- User-mode registers r12-r15 have been saved in the TCB.
- CSRs **csr_stat2**, **csr_prevpc**, **csr_ptr**, **csr_pgtable**, **csr_trapvec** will remain unchanged throughout the handler function.

If this is NOT a SYSCALL Exception…

Save the user mode registers:

Save r1-r6, r8-r11 in the TCB.
Read **csr_ptr** (i.e., previous value of r7) into a reg.
Store it in the TCB.
(About 12 instructions)

Prepare to execute kernel-mode functions written in high-level KPL:

Load r15 (the kernel stack pointer “sp”) from the TCB.
Load r12-r13 (i.e, “tp”, “gp”) from the TCB.
Copy r7 (the ptr to the TCB) back into **csr_ptr**.
(About 4 instructions)

```
Prepare the arguments to the individual trap handler.
    r1 ← csr_cause
    r2 ← csr_addr
    r3 ← csr_bad
    r4 ← csr_prevpc
    r5 ← csr_stat2
    // r6 ← <nothing>
    // r7 ← addr of TCB from above
        (About 5 instructions)

Dispatch to the individual trap handlers:
    Using r1 (csr_cause), jump through the trap vector to a KPL function.
        (About 3 instructions, using register "t", including indirect jump)

Upon entry to the KPL function to handle the interrupt / exception…
    • Registers r1 … r7 contain arguments, see above.
    • Register r8 ("t")  can be trashed.
    • Registers r1-r15 should be restored from the TCB before return.
    • CSRs csr_stat2, csr_prevpc, csr_ptr, csr_pgtable, csr_trapvec
         will remain unchanged throughout the handler function
```

So we may be looking at about 16 instructions for dispatching to a SYSCALL function and about 32 instructions to dispatch to any other trap handler.

After completely dealing with the trap, the individual KPL handler routines will not return. Instead, they will call one of two assembler functions. In either case, this is effectively a jump, since these routines do not return.

The KPL functions for handling SYSCALL traps will invoke a function named "SyscallHandlerReturn". The KPL function for handling all other exceptions and interrupts will invoke a function called "TrapHandlerReturn".

The **SyscallHandlerReturn** function is passed a value which it leaves in register r1 before executing the SYSRET. All other caller-saved regs should be zeroed or restored to prevent information leakage from the kernel.

Here are the final steps of a SYSCALL trap handler, which must be coded in assembly language:

Syscall Handler Return - Algorithm

```
// At this point, we assume…
//     r1 contains the value to be returned to the user mode code.
//     CSRs csr_stat2, csr_prevpc, csr_ptr, csr_pgtable, csr_trapvec
//         have remained unchanged throughout the handler function.
//     csr_ptr still points to the Thread Control Block (TCB).
//     csr_prevpc still contains the PC.
//     csr_stat2 still contains the status register.
//     csr_pgtable and csr_trapvec are unchanged.
//     User-mode registers r12-r15 have been saved in the TCB.

r7 ← 0.
Swap r7 with csr_ptr.
// r7 now points to the TCB.
// csr_ptr now contains 0.

// We assume that nothing of value remains on the kernel stack,
//     so we can avoid saving the value of “sp”.
// We assume that the kernel “tp” and “gp” registers never change,
//     so we can avoid saving their values.
// Register r14 (“lr”) is meaningless, so we don’t need to save it.

// Values of user registers “tp, gp, lr, sp” were saved at the time of the trap.
Fetch the saved registers from the TCB and move into r12…r15.

Set registers r2-r6,r8-r11 to 0    // To prevent info leakage from the kernel.
Swap csr_ptr with r7.
Execute the SYSRET instruction, which will…
    PC ← csr_prevpc.
    csr_status ← csr_stat2.

(About 17 instructions)
```

The **TrapHandlerReturn** function is passed no args. All the registers are restored before the SYSRET is executed. The TrapReturn function will take the following actions, performed in assembly code:

Trap Handler Return - Algorithm

```
// On entry, we assume…
//      csr_ptr still points to the Thread Control Block (TCB).
//      csr_prevpc still contains the PC.
//      csr_stat2 still contains the status register.

r7 ← csr_ptr

// We assume that nothing of value remains on the kernel stack,
//      so we can avoid saving the value of “sp”.
// We assume that the kernel “tp” and “gp” registers never change,
//      so we can avoid saving their values.
// We assume their values of sp, tp, and gp were placed in the TCB
//      when it was initialized.
// Register r14 (“lr”) is meaningless, so we don’t need to save it.

Fetch the “saved r7” from the TCB and move it into csr_ptr.
Fetch the saved registers from the TCB and move into r1…r6, r8…r15.

Swap csr_ptr with r7.
Execute the SYSRET instruction, which will…
    PC ← csr_prevpc.
    csr_status ← csr_stat2.

(About 19 instructions)
```

A typical trap handler will perform its work and complete, returning to the interrupted code without reenabling interrupts. However, in many cases the handler will be unable to return immediately and will need to block the thread. In such cases, the trap handler might perform a “wait” operation on a semaphore, or sleep on a lock, or simple invoke the scheduler directly. Often, an interrupt handler will need to wake up a process to service the interrupt. This could be done by performing a “signal” operation on a semaphore.

The algorithms sketched above are provisional. For example, it may be the case that the kernel will make use of additional CSRs (e.g., **csr_temp**) for storing registers, rather than saving the register in the TCB. This may improve trap invocation by replacing memory STORES and LOADs with CSR_SWAP instructions, at the cost of requiring these registers to be saved before re-enabling interrupts. Or, alternatively, **csr_stat2** and **csr_prevpc** may be saved to the TCB immediately within the global trap handler, removing the need to save/restore them when enabling interrupts.

It will often be the case that after a trap has occurred, but before a SYSRET has been executed, the kernel code will need to re-enable interrupts.

The above algorithms assume the interrupted thread was running in user mode. The global trap handler saves the user mode registers in the TCB in an area reserved for the user mode registers. However, if the interrupted thread is running in kernel mode, there could be issues. If, for example, the interrupted thread is a user thread currently running in kernel mode (for example, in the middle of servicing a SYSCALL) and it has reenabled interrupts, then an interrupt or exception will be catastrophic. The TCB register save area already contains the value of user mode registers. The global trap handler (as coded above) will blindly overwrite those, resulting in disaster when, at some future time, the thread tries to return to user mode.

The above algorithms are only intended to give you an idea of how the hardware could be used. Your kernel-hacking skills will be needed to figure out how to actually use the Blitz-64 ISA.[44]

[44] Another issue is the FENCE instruction, which may be needed.

Chapter 8: Memory, Address Spaces, and Page Tables

Quick Summary

- Program-generated address range: 64 GiBytes (36 bit addresses)
  - — Maximum Virtual Address Space: 32 GiBytes (35 bits)
  - — Max size of physical memory: 16 GiBytes (34 bits)
  - — Memory-mapped I/O region: 16 GiBytes (34 bits)
- The current page table's address is in **csr_pgtable**.
- Page table architecture:
  - — Page size: 16 KiBytes.
  - — Page offset (to access a byte within a page): 14 bits.
  - — Page table entry (PTE): 8 bytes.
  - — Each page holds 2Ki entries.

    2Ki x 8 bytes = 16 KiBytes.

    11 bits to index into a page (recall 2^{11} = 2,048).
  - — Page table has two levels.
  - — Virtual addresses are 35 bits.

    VPN[1]: 10 bits. VPN[2]: 11 bits. Offset: 14 bits.
  - — With a two level table…

    Only half of the top-level page is used.

    1Ki x 2Ki = 2Mi pages per address space.
  - — Maximum size of virtual address space:

    2Mi pages x 16 KiBytes/page= 32 GiBytes.
  - — Maximum size of supported, mappable memory:

    2^{44} = 16 TiBytes.
- The address translation cache (TLB registers): invisible to the ISA.
  - — Each TLB entry is tagged with an Address Space ID (ASID).
  - — ASID is 16 bits; maximum number of address spaces: 65,536.
  - — ASID of current process is in **csr_pgtable** register.

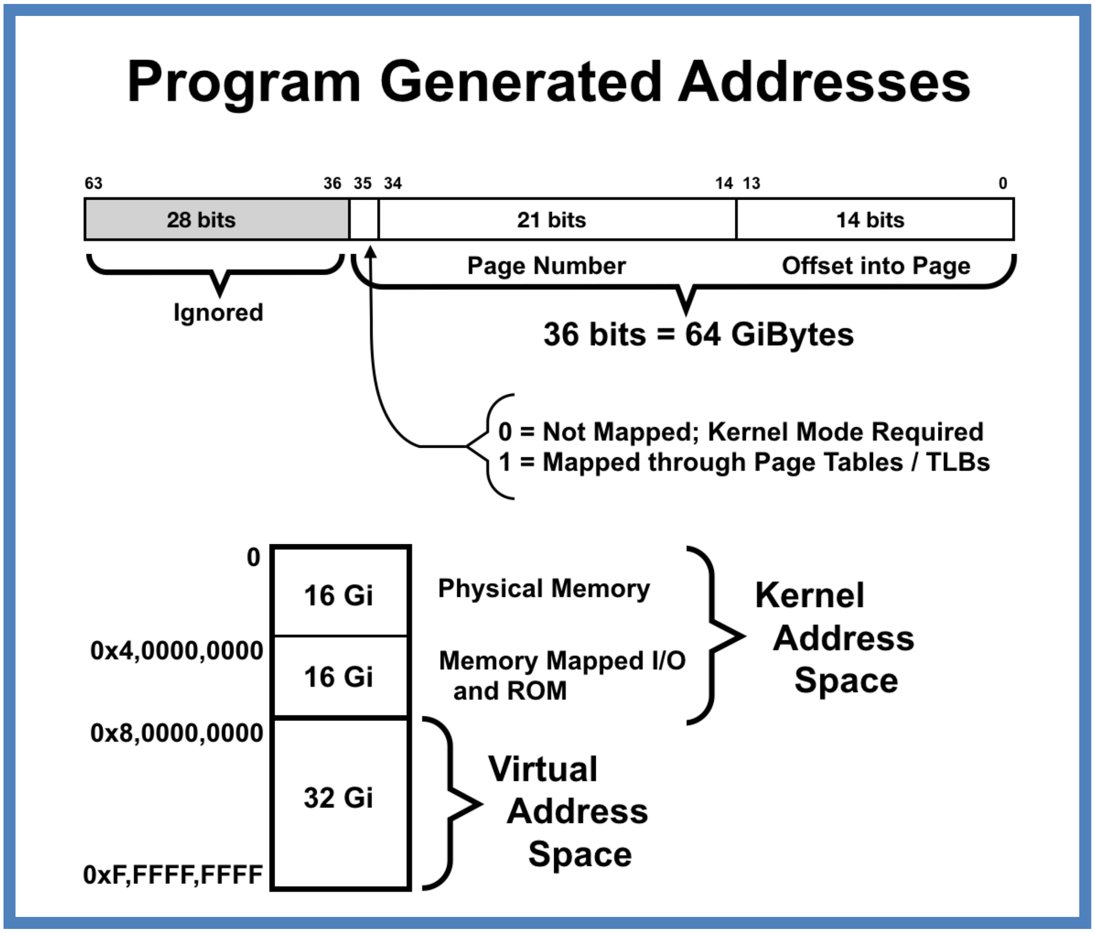
Memory Organization

All program-generated addresses are 36 bits. This allows a program to address up to 64 GiBytes.

This total 64 GiByte address space is divided into the following ranges:

size	
16 GiBytes	**Physical memory**
16 GiBytes	**Memory mapped I/O devices**
32 GiBytes	**Virtual address space**

FIGURE: Program Generated Addresses



Any address in the lower 32 GiBytes is said to be a **physical address**. Any address in the upper 32 GiBytes is said to be a **virtual address**.

Blitz-64 instructions can generate both physical addresses and virtual addresses. However, the processor core will only generate physical address for use in accessing the installed main memory and I/O devices.

The first 32 GiBytes (i.e., physical memory and memory-mapped I/O) is the kernel's address space and can only be accessed in kernel mode. User mode code cannot use addresses in this range. Any attempt by user mode code to use a physical address will cause a "Page Illegal Address Exception".

Of course, the kernel is free to map virtual addresses to physical addresses via the page table scheme. This allows user mode programs to access physical memory and memory-mapped I/O devices.

The upper 32 GiBytes can be accessed regardless of the privilege mode. Any address in this range is a virtual address and memory mapping will be performed to convert the address into a physical address in the lower 32 GiBytes.

Bit 35 of the address determines whether the access is allowed only in kernel mode or whether it will be mapped as a virtual address.

0 = Kernel access only; no memory mapping
1 = The address is virtual; memory mapping always performed

As mentioned above, any attempt to LOAD, STORE, or FETCH instructions using an address in the lower 32 GiBytes while executing in user mode will cause an exception. But any attempt to LOAD, STORE, or FETCH instructions using an address in the lower 32 GiBytes will be allowed when running in kernel mode, and the program-generated address will be used "as is". All bytes in the lower 32 GiBytes are considered to have full read/write/fetch privileges and no checking is performed.

A program-generated address is considered to be "virtual" and will be mapped to a physical address if and only if the address is within upper half of the address range. In other words, any address with bit 35 set to 1 (i.e., within the range 0x8_0000_0000 … 0xF_FFFF_FFFF) will be mapped.

Any attempt to LOAD, STORE, or FETCH instructions using an address in the upper 32 GiBytes will be processed by the **Memory Management Unit (MMU)**. The MMU will translate a virtual address into a physical address. The mode is irrelevant for this range.

Tasks, Address Spaces, and the User Mode Viewpoint

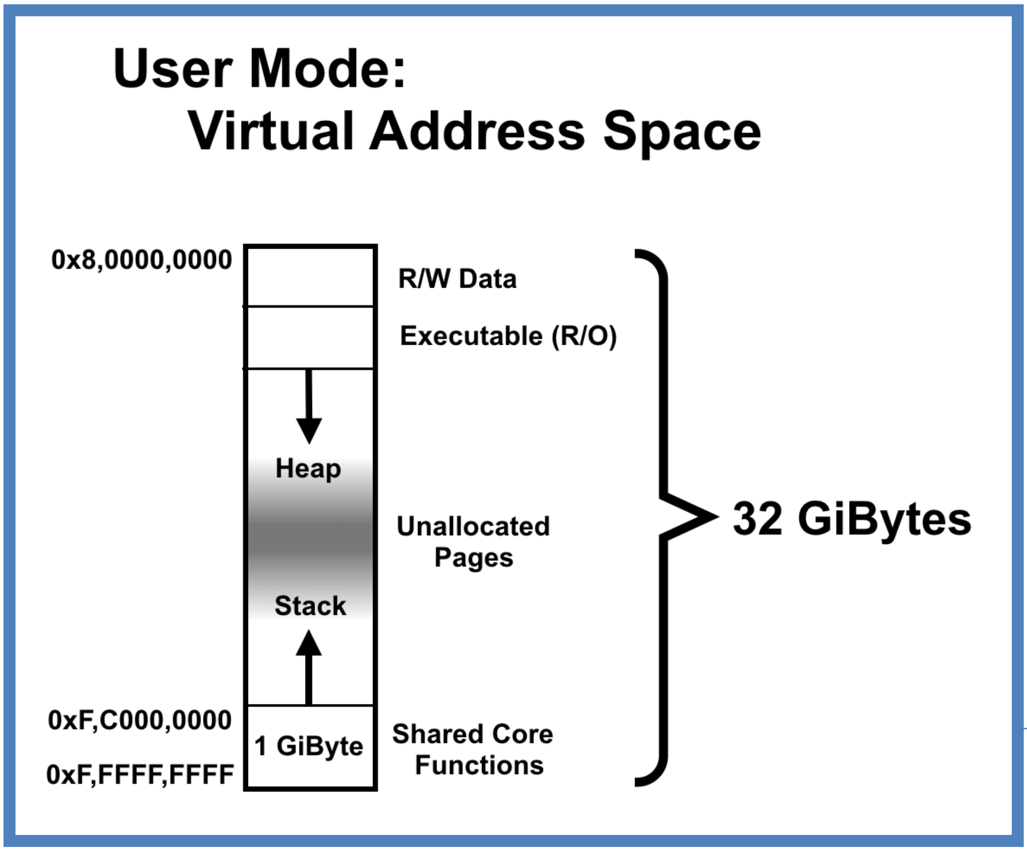
A user mode program in execution (i.e., a running program) is called a "**task**". A task consists of a virtual address space and one or more threads. (The term "process" is often defined as a task with exactly one thread. The "task" concept is more general and useful.)

A user mode program runs within a "**virtual address space**". Each byte has an address and a virtual address space appears to behave very similarly to a chunk of physical main memory. Generally speaking, each byte of the virtual address space will be implemented (i.e., "backed") by a byte of physical main memory.

Virtual address spaces are, of course, subtly different from physical memory. For one thing, virtual addresses are mapped into physical addresses in such a way that the user program has no way of determining which physical addresses are being used. Also, each virtual address space is independent; a user program has no way to access bytes in the kernel space or in other address spaces.

Here is one way a kernel might organize a virtual address space for user tasks, although this particular organization is not mandated by the ISA.

FIGURE: User Mode Virtual Address Space



Every virtual address space is broken into a number of pages.

In Blitz-64, the **page size is 16 KiBytes**. Each page starts on a 16 KiByte boundary: pages are always properly aligned. Since 2^{14} = 16,384 = 16 Ki, the last 14 bits [13:0] of the page address will always be 00000000000000.

Each page of the virtual address space will be either:

- Allocated
- Not allocated

Typically, most of the pages in the address space will be unallocated. Any attempt by the user program to access an unallocated page will cause an exception. Typically the user program will be aborted; that is random memory accesses to unallocated areas normally cause the kernel to terminate the program. However, it may also be the case that the kernel throws (i.e.., signals or forwards) a user mode error.

(There may also some pages in the virtual address space that are not allocated until there is a demand for them. For example, as the stack grows, pages will be allocated as necessary. However, this is transparent to the user program. When an attempt is made by a user program to access such a page, the kernel will quietly allocate a new page and retry the instruction. Such dynamic allocation is solely a kernel function.)

From the viewpoint of the running user mode program, each allocated page will have certain privileges. Every page is:

- Writable or not
- Executable or not

Therefore, the following combinations are allowed:

- Unallocated
- Read-only
- Read/write
- Read/executable
- Read/write/executable

Every allocated page is readable; there is not a separate privilege status for this. Pages containing executable code are always readable.

Every thread within a task will see the exact same address space. Each page will have the same privileges, regardless of which thread within the task is accessing it.[45]

Any attempt by a thread to LOAD from a page that is not allocated will cause an exception. Any attempt by a thread to STORE to a page that is not allocated or not writable will cause an exception. Any attempt to FETCH instructions from a page that is not allocated or not executable will cause an exception.

Presumably, the kernel will treat such accesses as a program error.

Note that here we are talking about the viewpoint of the user mode program. There are cases in which such attempts will cause exceptions but the kernel will take actions, change the status of pages, and restart the user program. The instruction will then execute and the user program will be unaware that there was ever an exception.

For example, imagine a situation where a page is allocated but is not currently resident in memory. Instead, the page has been written out to disk (i.e., backing store). Any attempt to access that page (FETCH, LOAD, or STORE) will cause an exception. The kernel will respond by reading the page's contents from disk into physical memory and resuming execution of the user program. Another example is when a page is marked "copy-on-write"; any attempt to STORE to the page will cause an exception; then the kernel will copy the page and the user program will be resumed.

In some cases, a page may be shared by two different address spaces. The page will be backed by a single page of physical memory. Thus, a page can be mapped into two (or more) address spaces. All tasks will see the same contents of the page. A WRITE by any task can be observed by a READ or FETCH performed in any other task. A single page of physical memory may be mapped into pages in multiple virtual address spaces at either the same or different virtual addresses. A single physical page may have the same or different permissions in the different address spaces.

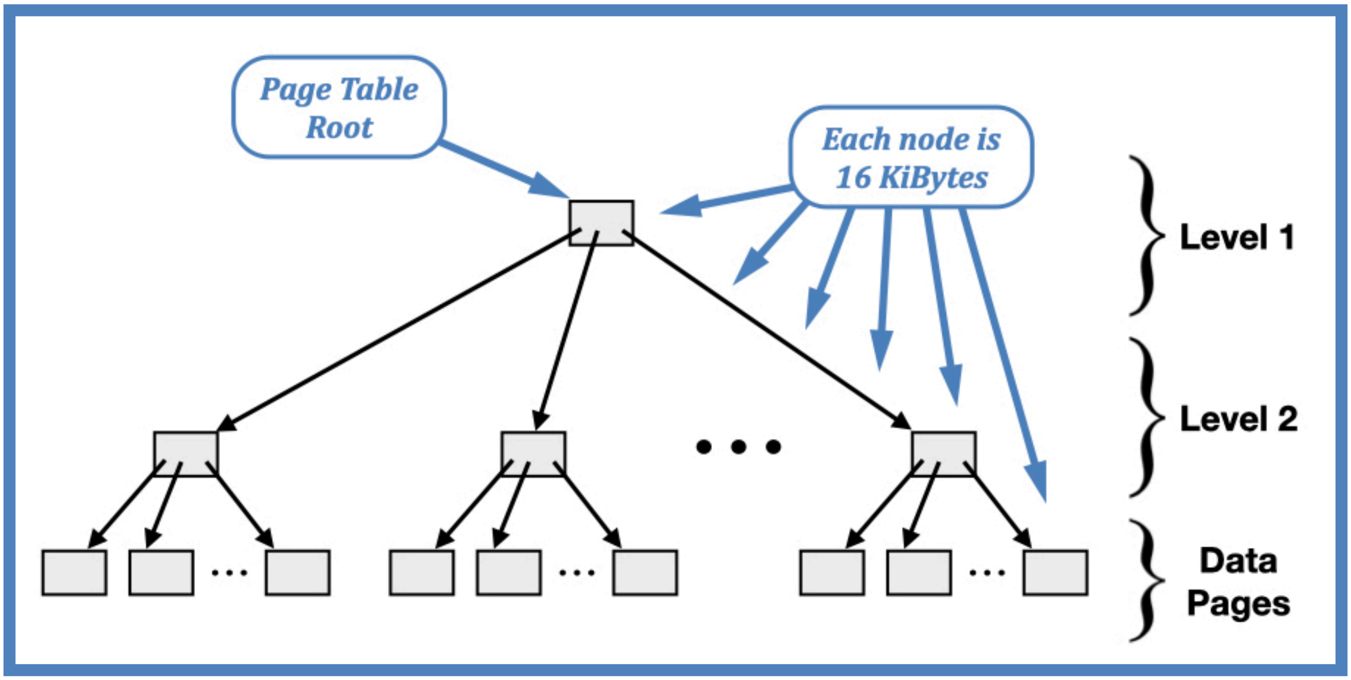
[45] It may be desirable for two threads to share most of an address space, but have some differences. For example, the pages of the stack might need to be mapped into different physical pages. In order to achieve this, the kernel must create a separate address space for each thread and mark all pages except the stack pages as "shared" in both address spaces. However, since address spaces can be up to 32 GiBytes, the preferred solution is to use a single address space and place the stacks in separate, non-overlapping memory regions.

In some cases, a virtual page may be mapped, not onto physical memory, but onto a location in the memory-mapped I/O region. In such a case, when the user mode program WRITEs to an address in the page (i.e., executes a STORE instruction), the data will be sent to the I/O device. When the user mode program READs from an address in the page (i.e., executes a LOAD instruction), data will be transferred from the I/O device.

Page Tables

For each address space, the kernel will create, build, and update a page table. Blitz-64 uses a two level page table.

Diagram: "Page Table Tree"



Each page table **index node** and each **data page** is stored in a single 16 KiByte page. Both index and data pages must at page-aligned addresses.

A page table tree consists of a root node and up to 1,024 second level index nodes. Each second level node can point to up to 2,048 data pages.

There can be up to 2,097,152 data pages in a virtual address space. Since each data page is 16 KiBytes, this exactly accommodates the largest virtual address space, which is 32 GiBytes.[46]

Each page table entry is 8 bytes.[47]

The **root node** contains 1,024 entries pointing to second level nodes. A page can accommodate up to 2,048 entries, but only the first half of the page is used. The second half of the root page is not used.

[46] Notice that

2^{21} = 1,024 × 2,048 = 2,097,152

and

2^{35} = 2 Mi × 16 Ki = 32 Gi

[47] Notice that

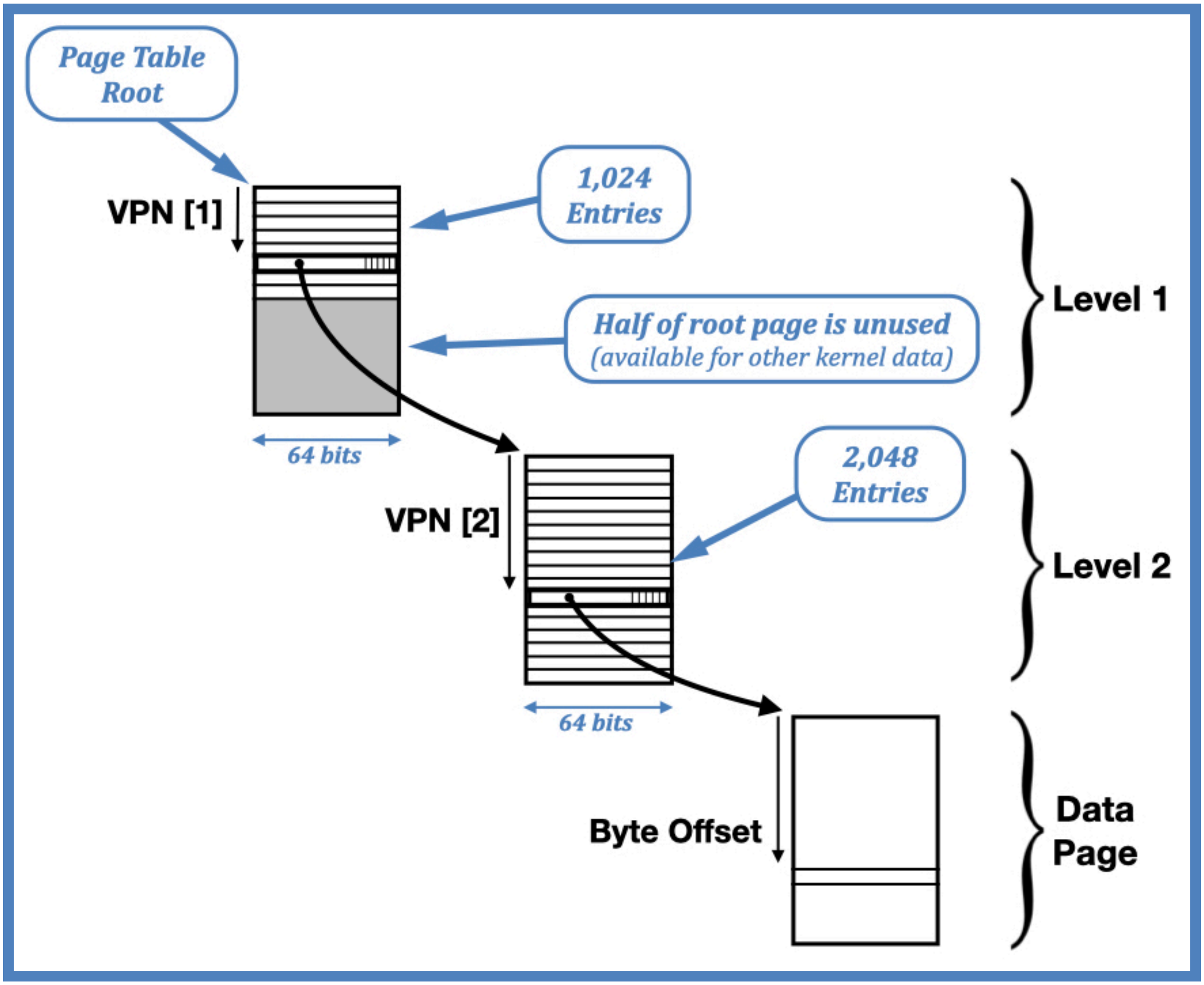
8 × 2,048 = 16,384

2^{11} = 2,048

So up to 2,048 PTE's will fit into a single page. And to address any one of 2,048 entries, 11 bits are required.

Each second level node contains 2,048 entries.

Diagram: "Page Table Detail"



The smallest address space would require a single root node and 2 nodes at the second level (one for low memory and one for high memory).[48] A page table for the largest address space will require 1,024+1 nodes in the page table tree.[49]

[48] Such a minimal page table will accommodate a virtual address space up to 64 MiBytes (i.e., 4,096 pages). This would easily include enough space for entries for the 32 pages required for the shared core functions, as described elsewhere.

[49] The maximum virtual address space has a size of 32 GiBytes (i.e., 2,097,152 pages). The page table for such an address space requires 1,024 + 1 pages (i.e., 16.02 MiBytes).

The memory overhead for page tables is roughly 8 bytes per 16 KiByte data page, in other words, the ratio of page table memory to data memory is less than 1 :: 2,000.

Each index page in the page table is stored in a single 16 KiByte page. Each index page is organized as an array of "**page table entries**" (PTEs). Each page table entry will be 8 bytes in length. Thus, a single page can contain 2,048 PTEs.

As mentioned, the top-level (root) node of the page table will only contain 1,024 PTEs so only the first half of the page is used. The kernel is free to use the second half of the page to store additional information about the address space, the task, and/or the threads running in that address space.

Conceptually, every time a virtual memory address is accessed, the page table will be walked to locate the data page and translate the virtual page number into a physical page number.

However, actually walking the page table to retrieve the page table entry (PTE) requires two additional memory accesses for every "real" memory access. This would impose an intolerable performance penalty.

Instead, it is assumed that the processor will cache recently used page table entries in order to avoid accessing the index pages for most memory operations. To improve performance, we assume Page Table Entries (PTEs) from the page table are cached in a set of special purpose registers designed for the purpose. This cache is called the **Translation Lookaside Buffer (TLB)**.

For the most part, the TLB is invisible to the kernel programmer. The caching is transparent and the result is exactly the same as if no TLBs are implemented.

Just as with main memory caches, the TLB registers are loaded automatically by the hardware, with no special attention required of the software. The presence or absence of a TLB cache will not change the correctness or functionality of the software, only its performance.

Whenever a memory operation (LOAD, STORE, or FETCH) is attempted using a virtual address, the page table must be consulted — at least in theory. If a matching entry is cached in a TLB register, then that can be used instead and the hardware can avoid the two additional memory operations that would be necessary to access the page table.

However, from time to time, the page tables will be modified by the kernel software. This may invalidate the information previously cached in the TLB registers. It is crucial that any PTEs cached in the TLB registers contain only valid and current information.The Blitz-64 ISA provides instructions to flush (i.e., modify or invalidate) selected TLB registers.[50]

An OS kernel will implement a number of virtual address spaces, with one address space for each task. Associated with every address space is a page table. At any one time, a core is executing code within one address space, so there is always a "current page table."

The current page table is pointed to by the Control and Status Register named **csr_pgtable**. More precisely, **csr_pgtable** contains the address of the root index page of the page table.[51]

When the kernel switches from one task to another (i.e., from one address space to a new address space), it will modify **csr_pgtable** to point to the page table of the new address space.

Given that there are many address spaces and many page tables, it is crucial that the cached page table entries (PTEs) in the TLB for the old address space not be confused with PTEs for the new address space.

To facilitate this, each address space is assigned a unique number called the **Address Space Identifier (ASID)**. This is a 16 bit value, accommodating up to 65,536 different address spaces. It is assumed that each PTE cached in a TLB register will be marked with the ASID of the address space to which it belongs.

The ASID of the currently executing task is kept in the **csr_pgtable** register.

The idea is that whenever the TLB is consulted to see if there is a cached PTE, the ASID is checked. Each TLB register will contain an ASID, along with the cached

[50] These instructions are named TLBFLUSH and TLBCLEAR.

[51] The **csr_pgtable** register contains a 44 bit page-aligned address within the physical main memory, i.e., anywhere within the 16 TiByte address space. Addressing memory within the lower 16 GiBytes can be done directly and easily when running in Kernel Mode, so it is likely that most OSes will choose to place all page table nodes within the first 16 GiBytes of main memory. Of course, page table nodes can be placed elsewhere, but in order to accommodate this, the kernel itself will need to set up and use a second virtual address space, solely for accessing such page table nodes.

mapping information. If there is an entry with a matching ASID, then the cached PTE can be used and the core can avoid accessing the index pages altogether. But if the ASID doesn't match, the cached value applies to a different address space and the cached PTE cannot be used.

If there is no cached PTE in the TLB registers, then the core is forced to access the page table in main memory. This will require the core to perform two additional LOADs to read from the index pages in order to retrieve the desired PTE. But once a PTE is retrieved from the in-memory page table, that PTE will be cached in the TLB. This means that future accesses will hit the TLB cache and all page table accesses for any addresses within the same data page can be avoided in the future. Whenever a PTE cached (i.e., the PTE is written to the TLB), the TLB register will be marked with the ASID of the current task, which is the ASID currently stored in **csr_pgtable**.

A General Overview of TLBs

To accommodate virtual memory, program-generated "**virtual addresses**" are translated into "**physical addresses**" in hardware by address translation hardware. This circuitry is called the **Memory Management Unit (MMU)** and it uses a page-table (stored in memory) to perform the translation from virtual to physical addresses.

To make address translation fast enough for virtual memory to be feasible, page table entries must be cached in an "address translation cache". Such a cache is traditionally called a "**Translation Lookaside Buffer**", or "**TLB**".

The TLB will contain a small number of page table entries. When a FETCH, LOAD or STORE to memory occurs, the address translation hardware (the MMU) will check the address translation cache (the TLB). If the TLB contains a matching entry, the address translation hardware will use it to generate a physical address immediately, which is much faster because the core can avoid going to main memory to read the page table tree to locate the data page.

The TLB is organized as an associative memory, keyed on **virtual page number (VPN)**. If a TLB entry is present, then the entry will contain the physical page number. The MMU will then concatenate the physical page number to the offset within the page to build a physical address. It will then proceed directly to performing the FETCH, LOAD, or STORE operation.

However, if the entry is absent, the address translation process must go to memory to locate and fetch the appropriate entry from the page table. Some TLB entry will be evicted and the new entry will be placed in the TLB. The address translation will then proceed.

From the Memory Management Unit's perspective, the in-memory page table is considered to be read-only. Thus, the values stored in the TLB never need to be updated by the hardware whenever a FETCH, LOAD, or STORE occurs.

When a memory access is attempted but the TLB contains no matching entry, the MMU will need to cache a new entry in the TLB. To make room, it must "evict" some existing entry. Since the TLB contains only copies of entries from the in-memory page table, the MMU has no need to update the in-memory page table.

However, from time-to-time, the kernel will modify the address space and update the page table. When this happens, the cached entries in the TLB can become out of date. To handle this, the hardware must include instructions that can be used to invalidate some or all entries in the TLB. The simplest approach is to include an instruction that will invalidate all TLB entries. A more targeted approach is to include an instruction that can be used to invalidate selected pages, possibly also including information about which address space is affected.

Megapages

Due to the large (16 KiByte) page size used in Blitz-64, the overhead of the page table is less than with the typical 4 KiByte page size of traditional architectures. Roughly speaking, a page that is 4 times as large could cut the number of page table walks by a factor of 4 and reduce the number of TLB registers by the same amount. The large page size also allows each page walk to require only two memory LOADs, instead of three, which are needed for a system with three level page tables.

We considered defining a "**megapage**" as a chunk of memory of size 32 MiBytes. This is exactly 2,048 pages in size. The idea is that a single entry in the root page of the page table would point directly to the megapage, instead of pointing to a second-level node in the page table.

With megapages, we would be able to accommodate very large address spaces with almost no page table overhead. The maximal address space with 32 GiBytes would require only a single root page table node.

It should be noted that with megapages, a single PTE will suffice for a very large amount of memory. Without megapages, many more PTE entries may be required to support the same algorithm. Thus, support for megapages reduces the number of PTEs, therefore reducing access to the in-memory page table and contention for TLB registers.

If a process's working set is not too large and changes slowly, then occasional page table walks and TLB loads will not be a great overhead, and a small number of TLB registers will support good performance. We believe that a system with (say) a dozen 16 KiByte pages should be adequate to cover the working set of many typical programs and thus provide good performance.

Of course, support for megapages will reduce the number of page table accesses and could be critical in programs that would otherwise have a lot of page table accesses.

As a general principle, complex algorithms with complex behavior tend to exhibit complex (i.e., seemingly random) memory patterns. In other words, modern programs bounce around a lot. For processes like this, there can be much more pressure on the TLB.

Megapages may become necessary to enable acceptable performance for complex algorithms that consume a lot of memory, exhibit little locality of reference, and bounce all around memory quickly.

At this time, Blitz-64 does not support megapages, but this decision may be revisited. An unused bit in the PTE entry might be defined to flag megapages. The PTE entry will either point to a second level page table node, or to a megapage, as determined by this bit.

<u>Why Only Two Levels?</u>

Most computer architectures use 3 or 4 level page tables. Blitz-64 was carefully designed to use only 2 levels. This mandated a limit on the maximum address space size; is it worth this cost?

For programs that exhibit very good locality of reference — that is, that have very small working sets — the TLBs will work well and the in-memory page tables will need to be consulted rarely. So it will matter less if the table is one level deeper: the additional memory operation will not often be needed.

However, we expect modern programs to often be more complex, such as object-oriented programs which bounce around large heaps or complex algorithms that exhibit minimal locality. Each time such an algorithms follows a pointers to a new area, a new pages is touched and another page table lookup is required. In the extreme, each and every memory reference could be a TLB miss and require a page table lookup. Going from 3 to 2 levels implies turning an operation that requires 4 memory references into one requiring only 3 memory operations, a huge performance gain.

While such completely degenerate programs may be rare, the general idea remains. Page table lookups are very, very costly and reducing each lookup from 3 memory accesses to 2 accesses will improve overall performance, although we cannot yet say by how much.

Another consideration is time-slicing. Every time the kernel's scheduler is invoked, a new thread is selected and initiated. The execution of the new thread may have the effect of entirely flushing the TLB, which means that every time the scheduler runs (i.e., every time-slice), the entire TLB will need to be reloaded. This means a page table lookup is required for each TLB register.

This cost can potentially be large. Of course, there are techniques to mitigate this problem, such as distributing the threads across cores in such a way that the scheduler will likely choose to run a thread that lives in the same address space as the thread previously scheduled.

A final issue is that address spaces will sometimes be changed and modified. With message passing kernels, we expect to see large amounts of data passed by the manipulation of page tables. For example, to pass data from one task to another, a page may be deallocated from one address space and mapped into another address space.

Each time such an operation is done, it may be necessary to flush the entire TLB, this requiring a reloading the TLB registers, with multiple page table lookups. Again, there may be techniques to reduce this cost. For example, Blitz-64 provides an instruction (TLBFLUSH) to clear a single entry cached in the TLB.

Although a 2 level page table is conceptually simpler than a 3 level table, the performance around virtual memory must be the overriding concern.

Some architectures have a flexible design. For example, the RISC-V can accommodate 2, 3, or 4 level page tables. Accommodating multiple depths introduces quite a bit of complexity into an ISA.

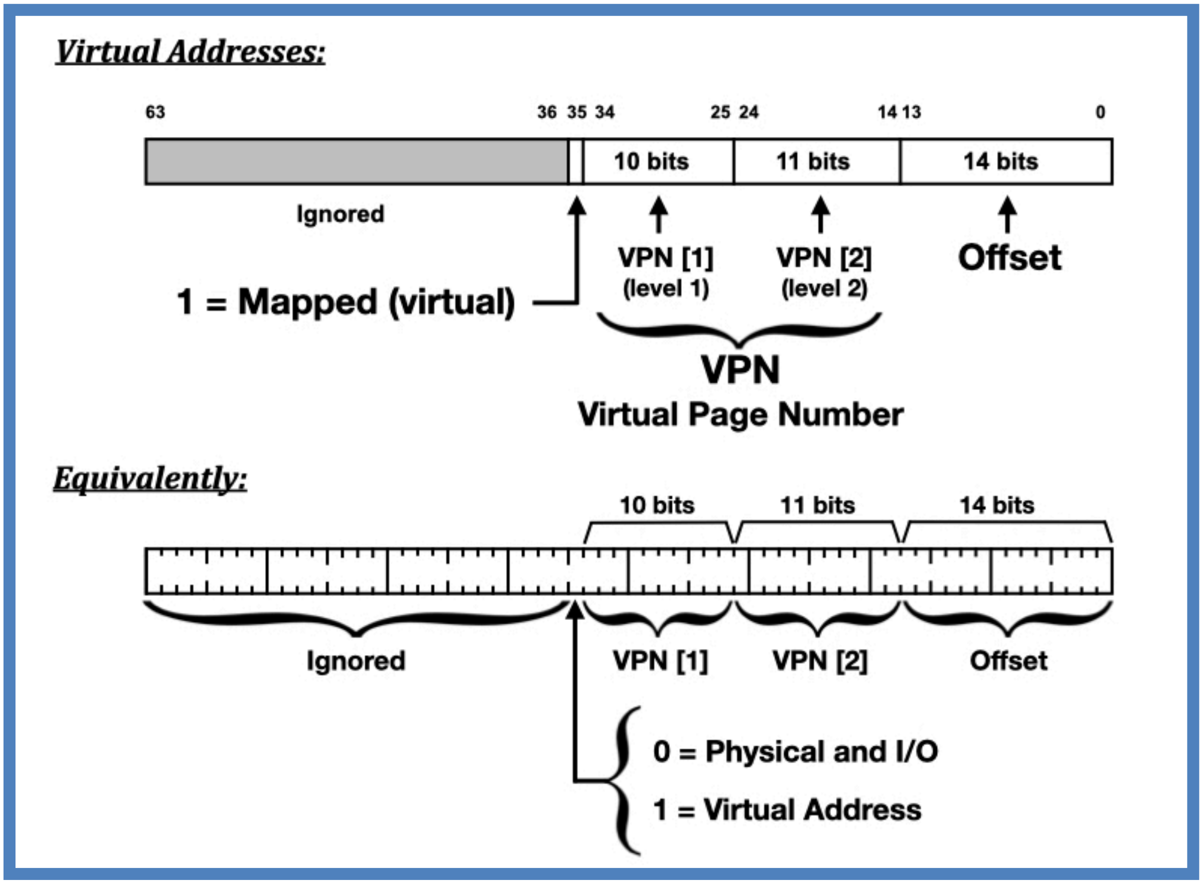
Finally, we note that 2 level page tables seem adequate, so there is no reason for a 3 level table. Of course this is closely tied to the decision to limit the maximum size of a virtual address space to 32 GiBytes.

Will the address space limitation prove to be a problem in practice? Are 3 levels clearly a better approach? Time will tell.

Virtual Addresses

Program-generated addresses are 36 bits, as shown in this diagram[52]:

FIGURE: “Virtual Address”



The upper bits [63:36] are always ignored. This means that any address outside of the basic 64 GiByte range is mapped into the lower 64 GiByte area.

The most significant bit [35] selects for virtual/kernel mode. If the bit is 1 (virtual address space), then memory mapping (i.e., address translation) will occur. If the bit

[52] Here is the same information, expressed differently:

bits	size	field
[35]	1	mapped or unmapped
[34:14]	21	VPN: virtual page number
[34:25]	10	VPN[1]: First level
[24:14]	11	VPN[2]: Second level
[13:0]	14	byte offset (14 bits)

is 0 (kernel/physical space), then mapping does not occur and the address is used, as is.

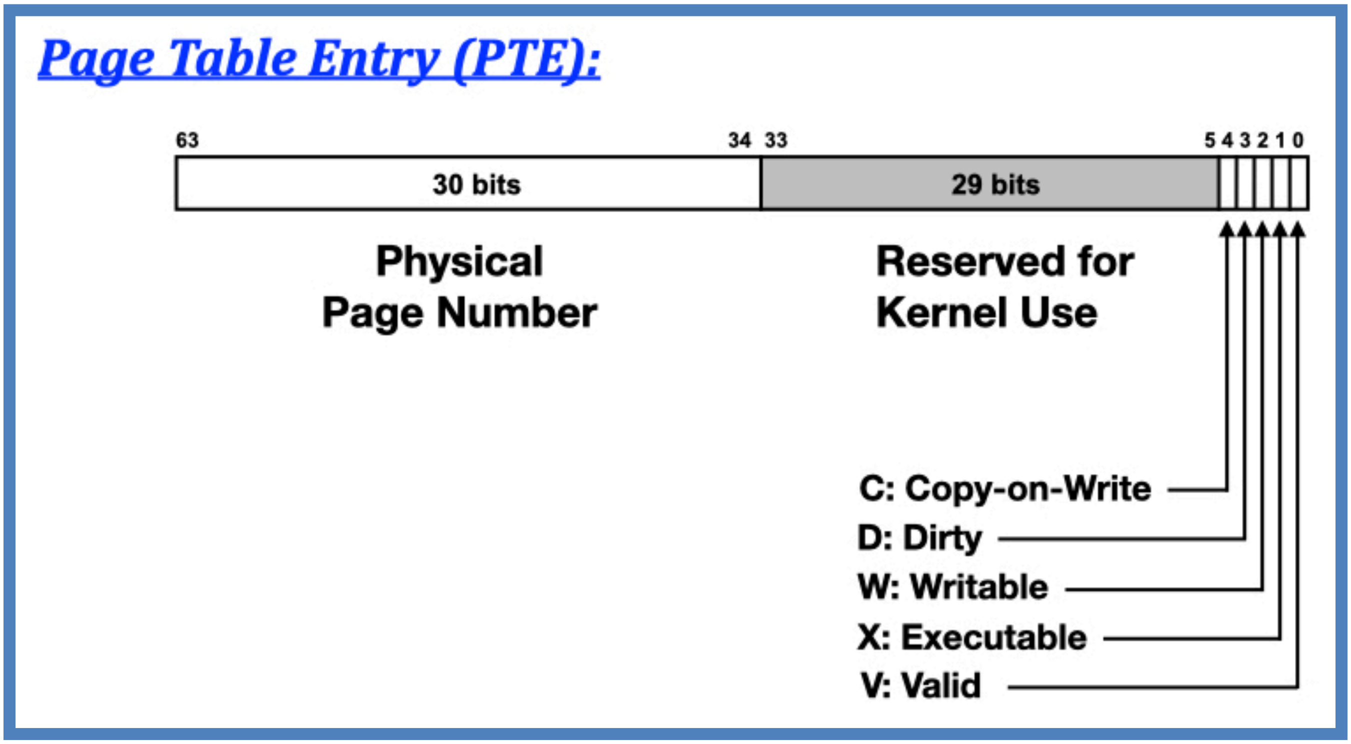
Memory is broken into pages. The **page size** is 16 KiBytes. To access a byte within a page, the **page offset** is 14 bits.

Within a virtual address, 21 bits, namely bits [34:14], indicate the **Virtual Page Number** (VPN). This is further broken into fields VPN[1] and VPN[2], which are used to index into the two-level page table tree.

Page Table Entries

Each entry in the page table is called a **Page Table Entry** (PTE). Each entry is **8 bytes** and has this format:

FIGURE: "Page Table Entry"



Here the same information:

bits	width	
[63:34]	30	Physical Page Number[53]
[33:5]	29	*< unused, available for kernel use >*
[4]	1	C bit (1 = Copy-on-write)
[3]	1	D bit (1 = Dirty)
[2]	1	W bit (1 = Writable)
[1]	1	X bit (1 = Executable)
[0]	1	V bit (1 = Valid)

Commentary We do not include a "Referenced Bit" as is done in some systems. The purpose of the Referenced Bit is to allow software to implement a least-recently-used algorithm (or more likely, an approximation thereto), in order to select which pages are candidates for paging out to backing store. However, updating and maintaining such a least recently used bit requires the MMU to write PTEs back to the page table. In Blitz-64, the MMU only reads from the page table.

When an executing program attempts to access memory, it will generate a 36-bit "program-generated address" and will use it to:

- LOAD
- STORE
- FETCH (i.e., read an instruction for execution)

[53] With 30 bits of physical page number and 14 bits of offset, this allows addressing up to 16 TiBytes of physical memory, since

2^{44} = 17,592,186,044,416

However, in the basic implementation, physical addresses are limited to 35 bits and only 32 GiBytes can be addressed, since

2^{35} = 34,359,738,368

This range accommodates 16 GiBytes of physical memory followed by 16 GiBytes of memory-mapped I/O. As such, only the lower 35-14=21 bits of the Physical Page Number will be non-zero, i.e., bits [54:34]. The upper 9 bits [63:55] should be zero and will be ignored on systems that don't exceed 32 GiBytes of installed physical main memory.

In systems accommodating more main memory than this, the upper 9 bits of a Physical Page Number can be non-zero. However, program-generated addresses are still limited to 36 bits and virtual addresses are still limited to 35 bits. This limits every virtual address space to 32 GiBytes.

The Memory Management Unit (MMU) sits between the core and main memory. The program-generated address will be put through the MMU along with the type of access required (LOAD/STORE/FETCH) and the current privilege mode (kernel or user).

The MMU will either generate an exception or will translate the address into a physical address. (If a TLB is implemented, the translation may be performed using the TLB registers.)

MMU: Basic Operation

We next describe the operation of the Memory Management Unit (MMU). We begin by describing the MMU as if there is no TLB cache and each memory access requires a page table lookup.

The MMU starts with a virtual address and the type of operation requested (LOAD, STORE, or FETCH). It may generate any one of these exceptions:

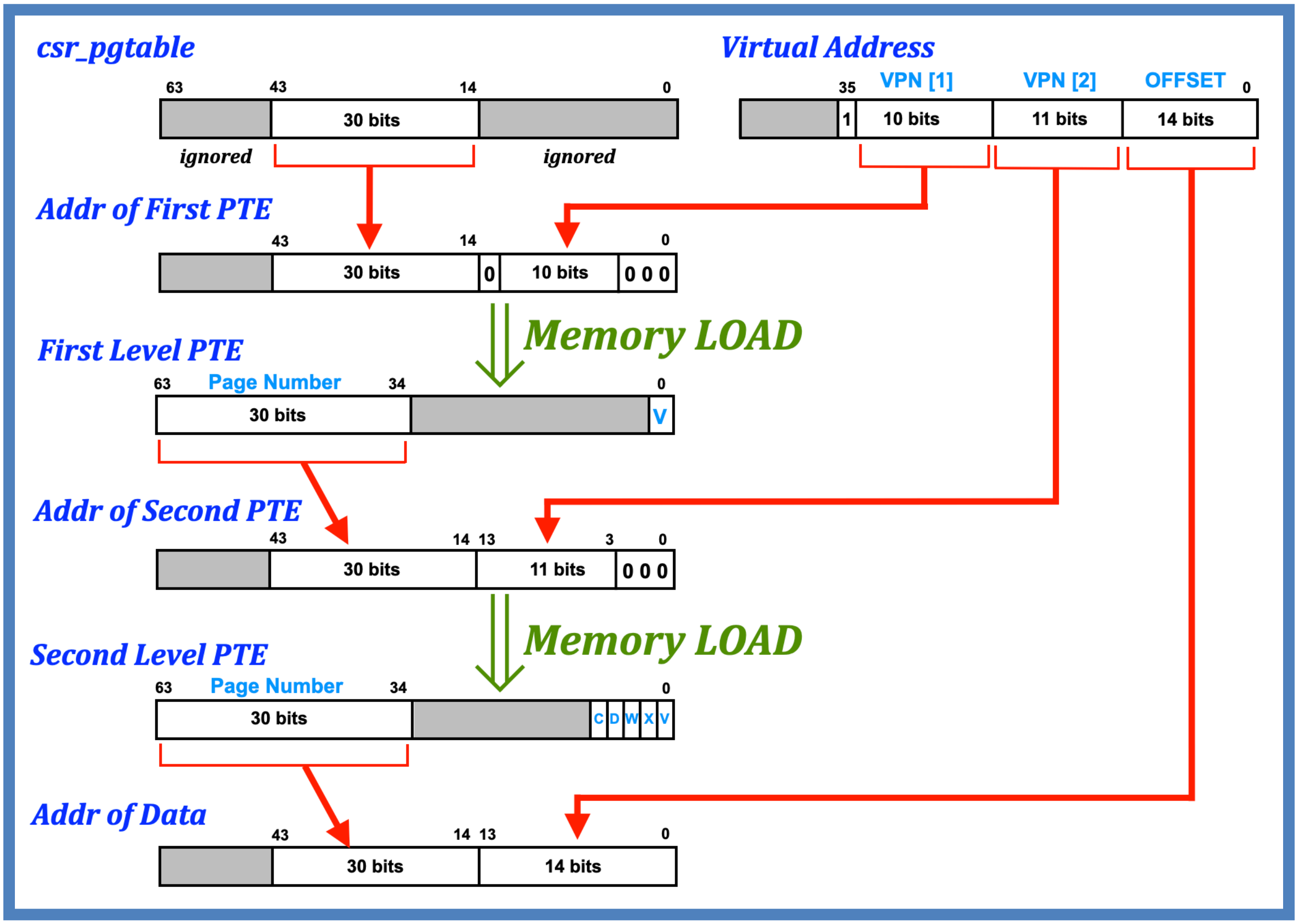
- Unaligned LOAD/STORE Exception
- Page Illegal Address Exception
- Page Table Exception
- Page Invalid Exception
- Page Fetch Exception
- Page Write Exception
- Page Copy-On-Write Exception
- Page First Dirty Exception

If the operation is LOAD.H, LOAD.W, LOAD.D, STORE.H, STORE.W, or STORE.D and the address is not aligned properly, then an “Unaligned LOAD/STORE Exception” will be triggered. Whenever an exception is triggered, the instruction execution is aborted and a trap occurs.

If the core is currently running in user mode and the address has bit [35] = 0, then we have an illegal attempt to access a physical memory or memory-mapped I/O address. A “Page Illegal Address Exception” will be triggered.

Otherwise, the MMU will walk the page table in order to obtain the address of the data. This involves first reading a page table entry (PTE) from the root page and then reading a PTE from the second level index page, as shown in the following diagram.

<u>FIGURE: “Mapping”</u>



First, the **csr_pgtable** register is used to obtain the address of the root index page.

If **csr_pgtable** contains 0, a "Page Table Exception" will be signaled.[54] But assuming **csr_pgtable** is not 0, the address it contains will used to locate and read an entry from the root index page.[55]

The VPN[1] field of the virtual address will be used to select a PTE within the root page and that PTE will be read from physical memory.

Within the root page, each entry will use only the following two fields. (The other fields will be ignored by the MMU and will presumably be zero.)

Physical Page Number
Valid Bit

The VALID (V) bit will be 1 if the PTE points to a second level index page, and 0 if not. If the MMU encounters a 0 valid bit, it will trigger a "Page Invalid Exception".

In the second step, the MMU will extract the physical page number of the second level index page and will use the VPN[2] field to select a PTE within the second level page.[56] That PTE will be read from memory.

If the VALID (V) bit of the second PTE is 0, a "Page Invalid Exception" will be triggered.

The address of the data page will be extracted from this PTE and a physical address will be constructed using the offset field from the virtual address.

[54] Presumably this is a kernel bug; a virtual address should not be generated unless the kernel has already created a page table and set **csr_pgtable** to point to it. The test for null only checks bits [43:14], i.e., the Physical Page Number (PPN) of the root node.

[55] Within **csr_pgtable**, the upper 20 bits (which include the ASID) will be ignored, to form an address of 44 bits, i.e., an address within the 16 TiByte physical memory area. The lower 14 bits will also be ignored and zeros will be used. This forces the address of the root page to be page-aligned, regardless of what **csr_pgtable** contains.

[56] More precisely, 30 bits are extracted from bit positions [63:34] in the "Physical Page Number" in the top level PTE. These upper 30 bits are used to construct the address of the second level PTE. The 10 bits of the VPN[1] field are shifted 3 bits to give a doubleword aligned offset. This is extended to 14 bits [13:0] by adding a zero for bit [13]. Together, the 30 bit page number [43:14] and the offset [13:0] give a 44 bit, doubleword aligned address in physical memory, which will contain the second level PTE.

If the requested operation is LOAD, then the operation will read from memory with no further ado.

If the requested operation is FETCH and the EXECUTE (X) bit is 1, the MMU will read from memory. If the EXECUTE (X) bit is 0, a “Page Fetch Exception” will be triggered.

Finally, if the requested operation is STORE then the W, C, and D bits will be checked, as follows:

W: Writable	**C: Copy-on-write**	**D: Dirty**	
0	…	…	Page Write Exception
1	0	0	Page First Dirty Exception
1	0	1	The operation is performed
1	1	0	Page Copy-On-Write Exception
1	1	1	The operation is performed

We can explain this as follows:

If the page is not writable (W=0) and the user tries to write into it, then it is a user error and the kernel will need to deal with the error (Page Write Exception). When the page is first written (D=0), it may be necessary for the kernel to update the page table in memory to indicate that if the page is to be evicted, it must first be saved to the backing storage (Page First Dirty Exception). Otherwise if the page has already been marked dirty, the operation can be performed without kernel involvement.

If the page is shared using copy-on-write (C=1), then upon the first write (D=0) it is necessary for the kernel to make a copy of that page (Page Copy-On-Write Exception). After that, the kernel can mark the page as having been copied by setting the D bit. Otherwise (D=1), the page has already been copied, so the operation can proceed without kernel involvement.

We can summarize the MMU interface as follows:

Memory Management Unit (MMU)

Input:

The current mode (Kernel or User)

The 36 bit program-generated address:

MSBit: Bit [35]
0=kernel region, i.e., unmapped
1=virtual region, i.e., mapped

VPN: Bits [34:14] page number (21 bits)

Offset: Bits [13:0] offset into page (14 bits)

Is this a FETCH attempt? (1 bit)

Is this a STORE attempt? (1 bit)

Alignment requirement:

- Doubleword
- Word
- Halfword
- None

csr_pgtable

- ASID (Address Space ID)
- Address of the page table root node

Output:

Status:

- Null Address Exception (Address < 8)
- Unaligned LOAD/STORE Exception (Address not properly aligned)
- Page Illegal Address Exception (User mode to access kernel space)
- Page Table Exception (Bad **csr_pgtable**)
- Page Invalid Exception (Either index page or PTE has V=0)
- Page Write Exception (Attempt to write an unwritable page)
- Page Fetch Exception (Attempt to execute an un-executable page)
- Page Copy-On-Write Exception (Attempt to write a copy-on-write page)
- Page First Dirty Exception (Write to a previous unmodified page)
- All okay

The physical address (35 bits)

(If there is an exception, we don't care about the address returned.)

The precedence of the exceptions is:

- Null Address Exception ← highest
- Unaligned LOAD/STORE Exception
- Page Illegal Address Exception
- Page Table Exception
- Page Invalid Exception
- Page Fetch Exception
- Page Write Exception
- Page Copy-On-Write Exception
- Page First Dirty Exception

What if we have several violations at once? For example, what if there is an alignment violation, and the address falls in kernel space while running in user mode, and the **csr_pgtable** register is null? We've got 3 things wrong. In such a case, only the Unaligned LOAD/STORE Exception will be signaled. The Page Table Exception and Page Illegal Address Exception will be ignored.

The last 6 exceptions are mutually exclusive and these errors cannot arise simultaneously.[57]

Here are some example scenarios:

Conflict: Null Address and Unaligned LOAD/STORE
Example: A LOADD instruction attempting to load from address 0x0_0000_0001 while running in kernel mode.
Result: Null Address Exception

Conflict: Null Address and Page Illegal Address
Example: A LOADD instruction attempting to load from address 0x0_0000_0000 while running in user mode.
Result: Null Address Exception

Conflict: Unaligned LOAD/STORE and Page Illegal Address

[57] If there is a Page Table Exception, then there is no Page Table Entry (PTE), so the remaining 5 exceptions cannot occur. If the PTE is invalid, then the flags (Copy-on-write, Dirty, Writable, Executable) do not exist, so the remaining 4 exceptions cannot occur. If there is a Fetch Exception, the operation is a FETCH and not a STORE, so the remaining 3 exceptions cannot occur. Finally, if one of the last there exceptions is signalled, then the operation must have been a STORE and the outcome — which was described above — can result in at most one of the final 3 exceptions.

Example: A LOADD instruction attempting to load from address 0x0_0000_1111 while running in user mode.
Result: Unaligned LOAD/STORE[58]

NOTE: The following exceptions are not suppressed when running in Kernel Mode:

Page Fetch Exception
Page Write Exception
Page Copy-On-Write Exception
Page First Dirty Exception

TLB: Translation Lookaside Buffer

Previously, we described the MMU as if there is no TLB cache, but there would almost certainly be a **Translation Lookaside Buffer (TLB)**.

Although the presence of a TLB is theoretically optional, in practice each core will have its own set of TLB registers to reduce accesses to memory that would otherwise be needed to fetch Page Table Entries (PTEs) from the in-memory page table.

The discussion below is intended to give you an idea of how a TLB would work.

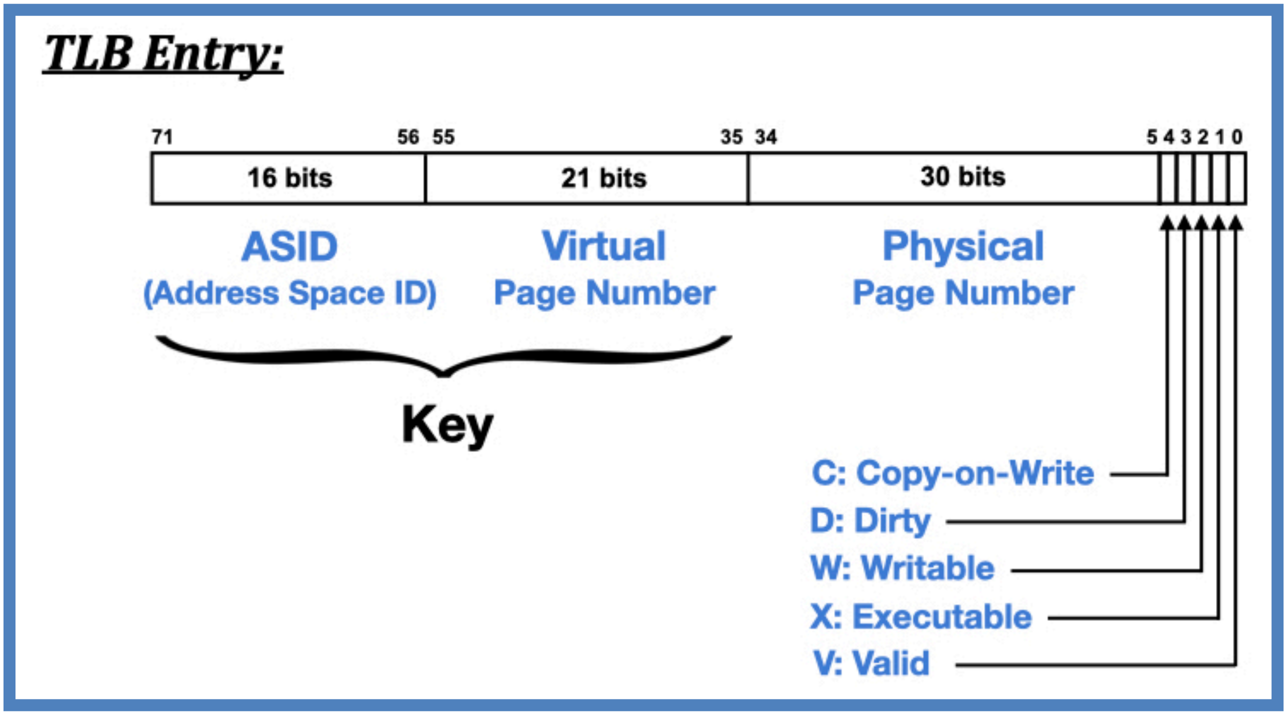
Each core contains a private set of **TLB registers**. These registers constitute the TLB cache and are not directly accessible by software.

The number of TLB registers associated with one core is implementation dependent. For example, there might be 128 TLB registers.

[58] This decision is arbitrary; it is hard to say which exception is more applicable in this case.

Each TLB register contains a **TLB entry**, which has the fields shown in the following diagram[59]:

FIGURE: "TLB Entry"



The TLB is a set-associative memory, keyed on "ASID || VirtPageNumber", i.e., the most significant 37 bits of the TLB entries.

Whenever a LOAD, STORE, or FETCH operation occurs, the MMU will first consult the TLB cache to see if it contains a matching PTE. If so, the MMU uses that and avoids reading from the in-memory page table.

The MMU uses the Address Space Identifier (ASID) from **csr_pgtable** and the virtual page number from from virtual address, to construct a "search key". The TLB cache is an associative memory and this key is used to retrieve a TLB entry with a matching key.

[59] This layout is merely a suggestion and implementations may lay out the TLB entry differently.

If a matching TLB entry is found, then it is used. This is called a **cache hit**.[60] A physical address is constructed and the bits (Copy-On-Write, Dirty, Writable, Executable, and Valid) are used as described earlier. Either an exception is signaled or the memory operation is performed.

However, if no matching entry is found, the MMU will then access the in-memory page table. This was described above. In addition, the MMU will construct an TLB entry and add it to the cache. Since the TLB cache is a fixed small size, this means that an existing entry must be evicted.

The TLB cache — at least as we are describing it here — will implement the least-recently-used algorithm in hardware.

To do this, the TLB will operate as a stack. In other words, the TLB registers are organized as a stack of memory registers, with TLB register 0 at the top of the stack and TLB register 127 at the bottom (assuming 128 registers in the cache).

Entry 0, at the top of the stack, will be **the most recently used** entry. The entry at the bottom of the stack (e.g., entry 127) is the **least-recently-used** entry, and will be the entry that gets evicted (i.e., discarded) when a new entry is pushed onto the stack top.

When there is a **cache miss**, the entire TLB register array will be shifted down. The last entry (e.g.., entry 127) will be discarded. The newly constructed entry will be added as entry 0. In other words, the new entry is pushed onto the top of the stack.

Furthermore, in order to maintain the proper order, any time a cache hit occurs, the matching entry must be removed from its place in the stack and moved to the top of the stack. All entries from the previous top, down to the matching entry, are shifted one position down, making room for the matching entry to be moved into the top position.

The TLB entries contain a VALID bit and, upon power-on-reset all VALID bits are set to 0. Whenever the VALID bit is 0 (indicating the entry is invalid), the entry is ignored. A cache hit can only return a valid entry.

[60] The TLB registers (which are a cache of page table entries) should not be confused with memory caches, such as L1, L2, and L3 which are a cache of main memory data. A hit in the cache of page table entries has nothing to do with a hit in the L1, L2, or L3 memory caches, although both can be said to be "cache hits".

There is an important difference between the VALID bit in a PTE in the in-memory page table and the VALID bit in a TLB entry. For the page table, an invalid entry means the page is not mapped into the virtual address space. Any attempt to access that page will require the kernel to determine whether a page should be allocated or whether the thread should be aborted. For the TLB entries, an invalid entry just means that the TLB register is not in use. When there is a cache hit for the TLB, the returned entry will always be valid.

Whenever the MMU performs a walk of the in-memory page table and retrieves an invalid PTE, it will signal a "Page Invalid Exception". The MMU will not update the TLB cache.

Whenever the kernel updates a PTE in the in-memory page table, we have a situation where the cached TLB entry becomes out-of-date. To deal with this, the kernel must invalidate the old TLB entry by setting its VALID bit to 0.

This is the purpose of the **TLBFLUSH instruction**. This instruction will mark any matching TLB entry as invalid[61]. Later, when the same virtual address is used, the MMU will get a cache miss and will respond by reading the PTE from the in-memory page table and adding to the TLB cache.[62]

From time to time, the kernel may wish to make major changes to an in-memory page table. Perhaps the virtual address space is deleted altogether, or perhaps a large number of PTEs are modified. In such a case, it will be necessary for the kernel to invalidate all the cached entries for a given address space. This is accomplished with the **TLBCLEAR instruction**. This instruction simply marks as invalid all TLB entries that have an ASID that matches the ASID from the **csr_pgtable** register.

[61] There can be at most one matching entry; the cache should never contain more than one valid entry with the same key.

[62] Ideally, when a new entry must be added to the TLB stack (evicting an existing entry from the cache), we'd like to reuse an entry that was previously marked invalid. So, whenever a cache miss occurs and a new entry is to be pushed onto the TLB stack top, instead of shifting all entries down and discarding the last entry, the shifting should only occur above the first invalid entry, thereby evicting and discarding the invalid entry.

An alternative approach to the TLBFLUSH instruction would be to move the invalidated entry to the bottom of the TLB stack, so that whenever an entry must be evicted in the future, the invalid entry will be discarded. However, this is impractical for the TLBCLEAR operation, which must invalidate a number of entries all at once.

When a LOAD, STORE, or FETCH occurs, the MMU will check the TLB. If the TLB contains a matching entry, then that PTE will be used and a walk of the page table is avoided. Whenever a TLB entry is successfully retrieved, it's also possible that the **csr_pgtable** register happens to contain zero, which would normally cause a Page Table Exception. Whether or not an exception will occur in this situation is specifically left unspecified and implementation dependent.[63]

It is also possible that the TLB cache becomes "out of synch" with the page table. For example, this could happen if the kernel failed to flush the TLB after switching to a new task and reloading **csr_pgtable**. In such a case, the TLB would return a PTE that is completely different from the PTE in the in-memory page table. Or perhaps a walk of the page table pointed to by **csr_pgtable** might encounter a missing index page and cause a Page Invalid Exception, while the TLB returns a perfectly serviceable PTE.

Obviously, in such cases, the contents of the TLB will be used. No walk of the in-memory page table will occur and such discrepancies would never be detected.

To summarize: If a TLB cache is implemented and a cached entry in the TLB provides a different result than a walk of the page table would provide, the in-memory page table is ignored and the TLB result prevails.

Comments

Kernel Access to User's Virtual Memory

Note that the kernel always has access to a user's virtual address space. This is convenient for a trap handler that implements a system call. For example, a user process may pass pointers to memory buffers through the syscall to the kernel code. When servicing the syscall, the kernel can simply LOAD and STORE from the virtual

[63] Normally, the **csr_pgtable** register will be zero only after power-on-reset until initialization is complete. From then on, only valid pointers would ever be stored into the register. Null would never be stored, so this situation is unlikely to occur in practice.

addresses that were provided. However, since the user mode process has passed in virtual addresses, the kernel must go through memory mapping.

In the Blitz-64 design, this is accomplished easily and naturally. The kernel simply uses the virtual address pointer as is. Virtual memory mapping occurs regardless of the current privilege mode.

Of course, the user code may pass illegal pointers to the kernel. The kernel really ought to check any virtual address before using it, to see what sort of an exception might be triggered if the access is attempted. The CHECKADDR instruction is provided for exactly this purpose.

As another example, a debugger process may wish to write to a user page that is otherwise not writable. This would be necessary when the debugger writes a BREAKPOINT instruction into a page of code that is marked executable, but not writable. Clearly, this must involve some sort of kernel involvement.

Perhaps the page in question is simply mapped into the debugger space as writable. But if the action is to be done directly by the kernel, it must temporarily change the page to writable. In more detail, the kernel must change the PTE entry in the page table to writable and execute the TLBFLUSH instruction in case the PTE was previously cached. Then the kernel can issue the STORE, change the PTE back to not-writable, and re-execute the TLBFLUSH instruction.

(Why did we not just specify in the Blitz-64 ISA something like “write permission checking is disabled whenever the core is running in kernel mode”? Because the core might be using separate data and instruction caches and an inconsistency might arise. Scenarios like this are explored in detail in a later section concerning caches.)

Regarding the 16 KiByte Page Size

Several page sizes were considered in the design of Blitz-64. Traditionally, pages have been 4 KiBytes; the selection of 16 KiBytes is somewhat radical.

Here are some arguments in favor of the larger page size:

- TLB entries need only be loaded ~¼ times as often, compared to systems with page size of 4 KiBytes.[64]

[64] Assuming good locality-of-reference, of course.

- Fewer TLB registers are needed. A TLB cache with ¼ the size will cover the same amount of virtual address space.

- A two level page table becomes feasible. In other systems with a smaller page size, a page table of at least three levels is required.

- With a two level page table, the hardware response to a TLB cache miss is substantially faster, requiring 2 memory LOADs, versus 3 LOADs.

- Whenever a page-related exception occurs, the kernel must search the in-memory page table. The algorithm to search the page table will be faster with a two level table than a three-level table, but the difference is minimal.

- For large processes, the page table takes ~¼ the space, since it only needs ¼ as many PTEs. Furthermore, initializing such a page table will be 4 times faster.

- Internal fragmentation (i.e., lost space inside the last page at the end of sections) is not much of a problem. Assuming 3 sections per process (i.e., **.data**, **.text,** and stack), 8 KiBytes on average lost per page, and 200 processes, we only lose 0.5% of 1 GiByte.)

- The total size of page tables is small. (Assuming 3 pages per process × 16 KiBytes per page × 200 processes = 1% of 1 GiByte.)

- Every process with data plus code size under 32 MiBytes requires only 3 pages (48 KiBytes) for its page table.[65]

On the other hand, there are some drawbacks to a larger page size:

- Each virtual address space will consume (i.e., waste) more physical memory with a larger page size, than with a smaller size. Let’s assume that most programs have three sections/segments (**.code** and **.data** in low memory and stack in high memory). On average, half of the last page in these segments will be wasted. So, about 3 × 16,384 × ½ = 24,576 bytes of virtual memory will be unnecessarily added to each address space. There is also waste in the second level index-pages, the amount of which is dependent on the size of the virtual

[65] One second-level index page covering the bottom of the virtual address space (where the **.data** and **.text** sections reside) will cover 2,048 × 16,385 = 32 MiBytes. Along with a page to cover high memory and the root index page, we have a total of 3 pages in the page table.

address space. If we assume that, on average, half a page is wasted in both high and low memory, we have 2 × 16,384 × ½ = 16,384. Thus, we estimate the total waste to be 40,960 bytes per process. In a system with (say) 200 processes, this will waste about 8,000,000 bytes. In a system with 2 GiBytes of main memory, this is less than 0.5%.

- Copying / initializing data pages each time a virtual address space is created will require more time. Assume two data sections (.text and .data) in low memory, with half of the last page being unused. We must initialize 2 × 2,048 bytes for 4 KiByte pages and 2 × 8,192 bytes for 16 KiByte pages. Assuming we need to initialize /clear the entire page for the stack section, we must initialize 4,096 bytes for 4 KiByte pages and 16,384 bytes for 16 KiByte pages. In summary, for each new address space, we will need to initialize 8,192 versus 32,768 bytes, which is 4 times as many bytes for the larger page size. To create a new address space, we are presumably reading in several kilobytes of code and data from a file. It is unclear whether the cost of zeroing an additional 24,576 bytes is significant.

- Many address spaces will be very small. Creating a mostly empty page table requires more space and more time with a larger page size, compared to a smaller page size. Roughly speaking, we can say a larger page size will waste as much as 4 times as much memory with additional, unused page table entries[66]. Since we need to initialize these unused PTEs, up to 4 times as much time will be required to setup the virtual address space.

Examples of System Memory Requirements

To give a feel for potential Blitz-64 usage, we show some example virtual address spaces.

Minimal process (112 KiBytes)

1 page for code and constants = 16 KiBytes (~1K lines of code)
1 page for data = 16 KiBytes
1 page for stack = 16 KiBytes
3 pages for page table = 48 KiBytes
1 page for kernel data and stack = 16 KiBytes
⇒ 7 pages (working set cannot exceed 3 TLB entries)

[66] In other words, many entries in the lowest level index pages will be unused and will need to be initialized to “invalid”.

Small process (512 KiBytes)
16 pages for code and constants = 256 KiBytes (~10K lines of code)
8 pages for data = 128 KiBytes
4 pages for stack = 64 KiBytes
3 pages for page table = 48 KiBytes
1 page for kernel data and stack = 16 KiBytes
⇒ 32 pages

Large process (48 MiBytes)
2048 pages for code and constants = 32 MiBytes (~1M lines of code)
1024 pages for data = 16 MiBytes
16 threads @ 4 pages per stack = 1 MiBytes
4 pages for page table = 64 KiBytes
1 page for kernel data and stack = 16 KiBytes
⇒ ~3200 pages

Mega process (400 MiBytes)
16Ki pages for code and constants = 256 MiBytes (~10M lines of code)
8Ki pages for data = 128 MiBytes
128 threads @ 4 pages per stack = 8 MiBytes
16 pages for page table = 256 KiBytes (< 0.1%)
1 page for kernel data and stack = 16 KiBytes
⇒ ~25,000 pages

Examples of installed physical memory
1 MiByte
= 64 pages (0 megapages)
Accommodates 8 minimal processes.
512 MiBytes
= 32Ki pages (or 16 megapages).
Accommodates 1,000 small processes.
Accommodates 200 small, 8 large processes.
4 GiBytes
= 256 Ki pages (or 128 megapages).
Accommodates 5 mega, 20 large, 2,000 small processes.

Shared Core Functions

The Blitz-64 design is tuned to support a set of globally shared functions. The idea is that a collection of functions is so widely used by user programs that it makes sense to make these functions available to all tasks in a uniform way, as a sort of universal, shared library.

These are called the "**shared core functions**" and the pages containing the functions will be mapped into every virtual address space, whether or not the functions are used. The maximum virtual address space is so large that a small number of pages set aside for the shared core functions does not make a significant difference.

In this discussion, we describe how *functions* are shared. We can also place heavily used *methods* into the shared core function library. Thus, commonly used classes need not be included in every executable program.

It is envisioned that there will be several hundred shared functions/methods. As an example, imagine that 2,500 functions are placed in the shared core library, with each consuming and average of 200 bytes each. This number of functions can be accommodated with 32 pages, consuming 512 KiBytes of the virtual address space.

The shared core functions will be placed in the very uppermost pages of the address space, which end at address 0xF_FFFF_FFFF. In our example, setting aside 32 pages out of 2,097,152 possible pages in the address space has no significant cost.

The pages of the shared core functions will be marked "read/execute" so that they can be freely shared by all virtual address spaces. Since the pages are shared and assumed to be memory-resident at all times anyway, there is essentially no overhead for individual tasks.

Regardless of how many pages are consumed by the shared core functions, when loading a program to be executed, the kernel will typically initialize the stack pointer to somewhere below the shared core function area.

To facilitate linking and the dynamic connection between separately compiled programs and the shared library functions, there will be a "dispatch table" (i.e., a branch or jump table) which will consist of one entry per shared core function.

Each entry of the dispatch table is 8 bytes. Each entry will contain a JUMP to the first instruction of the function. If the function itself is located close enough (i.e., within 32 pages, or 512 KiBytes), a single JAL instruction will suffice. Otherwise, the JUMP will require two instructions, i.e., 8 bytes.

The dispatch table will occupy the last page of the address space. If the number of functions exceeds 2,048, the dispatch table will occupy the last two pages.

First page of dispatch table:	
Number of entries:	2,048
Entry 0:	F_FFFF_C000
Entry 2047:	F_FFFF_FFF8
Additional page (if necessary):	
Number of entries:	2,048
Entry 2,048:	F_FFFF_8000
Entry 4,095:	F_FFFF_BFF8

To call (i.e., invoke) a shared core function, user code can contain a CALL instruction to the dispatch table address. From there, the JUMP instruction will take execution to the first instruction of the function.

The purpose of using a dispatch table is to allow simple linking between arbitrary user mode programs and the shared core function library. Each shared core function is assigned an offset in the dispatch table which will never change. There will be a KPL header file declaring each of the shared core functions as an “external” function. This allows the KPL compiler to perform type checking on the function invocations. A simple assembly file will equate each function name with the address of the corresponding dispatch table entry. The user mode programs do not need to know the exact location or size of the functions themselves, and these can be changed without needing to recompile user mode programs. Functions can be modified and new functions can be added to the shared core function library without requiring user mode programs to be recompiled.

To invoke a share core function, the code will first “call” to an address in the dispatch table and, from there, a JUMP instruction will take execution to the first instruction of the function.

Notice that a CALL to a shared core function can always be implemented with a single JALR instruction. (This is because, using register “r0” and a negative offset, each of the 4,096 entries in the dispatch table can be reached with a single instruction.) From there, a single instruction will take execution to the function itself. (This is because the JALR instruction contains a 20 bit offset and can jump -524,288 … +524,287 bytes relative to the PC.)

Thus, the overhead for invoking a shared core function is a single instruction![67]

[67] Since some function invocations might be far away and require two instructions, we can actually say the overhead is *at most* one instruction for shared core functions.

Chapter 9: Multi-Core Issues

Quick Summary

- Cores may have **private memory** and/or **shared memory**.
- LOAD and STORE instructions are always atomic.
- Cores are assumed to have caches, which may contain out-of-date values.
- Blitz-64 uses a **relaxed memory model**.
    - — STOREs do not propagate instantly to all caches in the system.
    - — STOREs do not necessarily propagate in the same order to other cores.
- The **FENCE instruction** is provided to force change propagation.
    - — Blitz-64 uses a single, coarse-grained FENCE semantics for all purposes.
- Programmers must use **locks** to ensure correctness and protect shared data.
- Lock ACQUIRE and RELEASE must use the FENCE instruction.
- A FENCE on one core may require cache lines on other cores to be invalidated.
- A modification to an address space on one core may invalidate remote TLBs.

Private and Shared Memory

A multi-core system in which all cores share a common block of main memory is called a **Shared Memory Multiprocessor (SMP)**.[68] In a typical multi-core system, there is a single large block of physical memory and this block is mapped to the same location in all cores, thus making it fully and symmetrically shared.

The Blitz-64 architecture supports both private and shared memory.

A multi-core system may have only shared memory and have no private memory. This is expected to be a common design choice, mirroring other SMP computers. The shared physical memory will be located at the same physical addresses in all cores.

[68] Other authors use SMP to stand for “Symmetric Multiprocessing”, where all cores are identical and connected symmetrically.

On the other hand, a multi-core system may have only private memory and no shared memory. In this design, each core will have its own block of private memory.

Finally, a Blitz-64 system may have a mix of both private and shared memory. In such a design, the private memory might be devoted to containing the kernel code and the code of the share core functions. The benefit of this is that any cache loads from this region will be entirely local and therefore faster.[69]

Private memory, if present, will always start at address 0. The cores need not have the same amount of private memory, although we normally expect all cores in a given computer to have the same amount.

Shared memory, if it exists, will always follow private memory. The shared memory will be located at the same address in all cores. If no cores have private memory, the shared physical memory will be placed at address 0.

LOAD / STORE Atomicity

In Blitz-64, all LOAD and STORE operations are required to be properly aligned. This means that the data involved can never cross a cache line[70]. Since the data in question will reside entirely within a single cache line,

> ***Every LOAD and STORE instruction is atomic.***

One core (call it "A") may STORE a value and another core (call it "B") may LOAD or FETCH from the same address. By "atomic", we mean that if another core looks at the same address, it will either see the data as it was before the modification or as it is

[69] If all private memories contain the identical data, then these addresses can be mapped into virtual space with no problem. For example, the shared core functions would typically be mapped into the highest pages of the virtual address spaces. When a cache fault occurs and the data must be loaded, the virtual address will be mapped into a physical address within private memory. It will not matter which core is executing, since all private memories contain identical data.

[70] In this section, the terms "cache line" and "cache block" are used synonymously. Elsewhere, "cache block" is used to refer only to the data in a cache line, while a "cache line" includes address key and control bits as well as the block of data.

after the modification. But it will never retrieve a value that is partly modified and partly unmodified.[71]

If instead, some data value is not aligned, it might possibly cross a cache line boundary. In other words, the value could reside partly in one cache line and partly in the following cache line. An update to the data value by core A will need to modify both cache lines. Now consider what might happen at core B. It may be that one of the cache lines is present in the cache of B, but the other line is not present.
Of course the updates will not be instantaneous. If core B looks at the data in a given cache line, it will either see the data before a change or after the modification.

As an example, assume that some data item crosses a cache boundary. When core A updates the data value, the modifications to the cache lines must propagate to other cores.

Blitz-64 allows for a relaxed memory model, which means these changes may propagate at different speeds. Thus it is possible that core B will see part of the data as it was before the modification and the other part of the data as it was after the modification. Thus, core B could effectively retrieve a value that was never actually stored by any core!

Of course some shared data is sometimes very large and must necessarily lie in multiple cache lines.

To control synchronization, we assume the kernel uses locks and respects the **locking protocol**[72]. Before accessing any shared data, we assume the kernel has acquired the appropriate lock, giving the core exclusive use of that data.

But even though locks are used, we still have the problem of cache lines being out-of-date, which we now turn our attention to in the following sections.
core.

[71] We are implicitly mandating that cache lines must be at least 8 bytes long.

[72] The **locking protocol** requires that a lock is **acquired** before the shared data is read or modified and the lock is **released** afterwards. The code between the acquire and the release operations is called a **critical section** and the shared data is only accessed within a critical section, i.e., when the relevant lock is held.

A Relaxed Memory-Model

In a **Shared Memory Multiprocessor (SMP)** model, several cores share physical memory. In this section, we'll ignore private memory, since the issues to be discussed do not arise.

In the basic, simplest model of memory, every memory location has exactly one value — or at least behaves in a functionally equivalent way. Any STORE to some location (say ***X***) will become immediately visible to all cores. Any subsequent LOAD or FETCH from address ***X*** by any core can only return the most recent value, and never any prior value. If one core updates two locations one after the other (say ***X*** first, followed by a STORE to ***Y***), then all cores must observe those updates in that order.

In short, such a system behaves as if there are no caches. Every address is stored in only one location and every memory operation is executed in linear order, one after the other. But of course caches — which complicate things — are necessary for performance.

To improve efficiency, the same cache block will often be held in multiple caches at different cores. To deal with this, many common cache protocols (such as MSI, MESI, MOESI, …) are designed to preserve the invariants discussed above, allowing the programmer to ignore cacheing, without risking incorrect results. In other words, the caching is transparent. While caches will affect performance, they will not affect functionality or results.

But there is a cost to making cacheing fully transparent[73]. To address this, Blitz-64 adopts a **relaxed memory model**.

> ***<u>The Blitz-64 Memory Model:</u> Blitz-64 accommodates a relaxed memory model. Local caches are assumed to exist and the results of updates to memory (i.e., STOREs) are not assumed to propagate to other cores instantly or in the order actually executed.***

The relaxed memory model of Blitz-64 does not assume a memory in which each address contains exactly one value at any instant or that all cores see exactly the same value. This is different from systems with transparent caching, which require a

[73] Generally speaking, this cost is in additional bus traffic imposed by the cache coherence protocol and the additional overhead of the snooping required of the individual caches.

cache protocol that guarantees that it appears to all cores that, at every instant, every memory location has exactly one value.

Instead, Blitz-64 assumes that cores have local caches which are not transparent. Although any update to a memory address by one core will *eventually* be seen by other cores, cache propagations take non-zero time. Some cores may still see the old values while other cores are already seeing the new value.

Since cache propagation is not instant, some cache lines may contain older values and still remain valid in the Blitz-64 memory model. As a result, one core may fetch data that is old, out-of-date, and seemingly made in an inconsistent order with respect to other cores and other memory locations.

Like all systems, Blitz-64 requires the cache protocol to implement a **coherent memory model**.

By “coherent”, we mean that a sequence of writes of different values by one core to any single location must be observed by all cores as happening in the same order, i.e., in the order they were actually performed. Basically, if any core sees the new value in location X, it can never subsequently retrieve an old value from that location.

The behavior of LOADs, STOREs, and FETCHes made exclusively by one core must always respect the order in which they appear, at least to that core. By “respect”, we mean that any reordering performed by the compiler or an out-of-order core must be transparent and will never be observable by that core.

However, since the core doing the reordering may not fully understand the interdependencies of the data, the reordering may be visible at other cores.

Consequently, an additional mechanism — the FENCE instruction — is required to prevent a core from reordering certain operations and to constrain the order in which changes are propagated to other cores.

With the Blitz-64 **relaxed memory model**, if multiple addresses are involved, it is possible and acceptable that the updates made by one core (call it A) do not appear to another core (B) as being made in the same order. For example, if core A writes a new value to address X followed by writing a new value to address Y, some other core (such as B) might see the new value when reading from Y but subsequently see the old value when reading from X!

Of course, the software really ought to use locks and the FENCE instruction (as discussed below) to prevent such confusing scenarios.

FENCE and Memory Synchronization

An **out-of-order** core (which is sometimes called a **superscalar** core) may execute the instructions in a slightly different order than they actually appear in the instruction stream. This dynamic rescheduling of instructions is done to improve performance and more efficiently utilize the hardware's circuits and functional units. The execution of the reordered instructions must be transparent and the results indistinguishable. Reordering by the core is allowable only when there are no data dependencies between the reordered instructions.

For example, consider this instruction sequence:

```
movi   r7,0x1234567890
div    r1,r2,r3
addi   r1,r1,r7
```

Since there are no registers used in common by both the MOVI and DIV instructions, the order of these two instructions doesn't matter. The core is free to begin the DIV instruction first. In fact, this is probably a good idea since DIV will take longer than MOVI to complete. But the ADD cannot begin until both MOVI and DIV have completed.

However, the core may not fully understand all data dependencies, especially in the presence of concurrent algorithms and multiple cores.

On a single core system, **the FENCE instruction** can be used to make sure that an out-of-order core (which might be reordering instructions for the sake of performance) does not violate some ordering and/or synchronization requirements that must be respected, but which cannot be inferred from a myopic analysis of the instruction stream.

> ***The FENCE instruction requires that any instruction that occurs before the FENCE is completed before beginning the execution of any instruction after the FENCE.***

FENCE instructions can be used to limit and restrict any instruction reordering an out-of-order core or compiler might otherwise attempt.[74]

For **multi-core systems**, the FENCE requirement is expanded to include inter-core interactions. The FENCE instruction affects the instructions on the core that executes the FENCE, but the FENCE instruction is required to do more in multi-core systems.

To address the problem of some data being old and out-of-date in the caches of other cores, the following additional requirement is added:

> ***The FENCE instruction requires all memory updates performed by the core to be fully propagated to other cores before execution proceeds.***

In other words, after the FENCE instruction is completed, it must be impossible for any other core to see an old, out-of-date value for any memory location that was updated by the core before that core executed the FENCE instruction. All STORE operations must be fully performed and all old, out-of-date values must be purged from all caches before the FENCE instruction can be retired.[75]

Note that we specifically do not place the following requirement on the FENCE: "All LOADs and FETCHes that occur after the FENCE must retrieve the most recent, up-to-date value."[76] Such a requirement would impose an additional burden on the implementation. Assuming that locks are used rigorously and properly to protect shared data in a multi-core system, the requirement to propagate STOREs is sufficient to ensure the shared data is always accessed exclusively.

[74] We also assume that a "**fence statement**" is available to the compiler. This is true in the KPL programming language. The presence of a fence statement in the KPL code will restrict the reorderings that the compiler might consider, as well as insert a FENCE instruction.

[75] For example, a write buffer must be emptied and STORES must be propagated before execution proceeds.

[76] In other words, a core executing a FENCE instruction must propagate all prior STOREs it made to all other cores. But, any STOREs made on other cores, need not be propagated to the core executing the FENCE instruction.

<u>Commentary</u> The specification of FENCE, as given above, imply that the following are true:

(1) ***All memory operations that appear before the FENCE are truly completed before the FENCE.***

(2) ***All memory operations that appear after the FENCE are truly not started until after the FENCE.***

These requirements impact changes to shared memory and the timing of when those changes become visible to other cores.

<u>Transparent Cache Protocol</u>

Imagine a very simple multi-core system in which private caches do not exist: there is only main memory.[77] This is a perfectly reasonable implementation for smaller systems where simplicity is favored over performance. Every time a FETCH, LOAD, or STORE is executed, the operation is performed directly on the main memory. In such a simple system, it is not possible for an address to simultaneously contain two values, since there is only one location in which the value can be stored. In such a system, the problem of out-of-date data is impossible. The requirement that "all previous memory operations become visible to all cores" happens implicitly with every STORE operation.

Some computer designs include private, per-core local caches, but the cache protocol will be designed in such a way that all updates to data are immediately propagated to other caches. Such an "instant propagation" protocol will guarantee that every memory address will appear to have exactly one value. In this design approach, cacheing is entirely transparent — data is never out-of-date — and its

[77] A **memory-side cache** (often called the **L3 cache**) is a cache which sits between main memory and the bus that connects to the private caches. All accesses to main memory must pass though the L3 cache. In our discussion of cache synchronization, we are focussing on private, per-core caches and we ignore memory-side caches. While the L3 cache can contain a different value than main memory — implying that one value is out-of-date — all cores will always see the same value, and that will be the value in the L3 cache, which is the most current value.

presence does not affect the result. Aside from performance impacts, the behavior is identical to the simpler design without any private caches at all.[78]

Linearizability

A **strictly linearizable cache protocol** is defined as follows. Although the actual order in which operations occur is not fully constrained, there must exist a total order for all LOADs and STOREs.[79] The results of a strictly linearizable protocol are the same as if all operations had been performed sequentially, one after the other in that order, on a system without any cacheing.

The “no cache” and the “instant propagation” protocols described in previous paragraphs are strictly linearizable. However the strictly linearizable protocol allows added flexibility in the ordering of two operations that are performed by different cores. Another way to say this is that cache propagations can be delayed, as long as the outcome is guaranteed to be the identical to the result on a no-cache system in which the cores run at variable and indeterminate speeds.

While a linearizable cache protocol is ideal, there is a cost to making cacheing entirely transparent.

So in some modern systems — including Blitz-64 — strict linearizability is sacrificed. Updates to the cache are not required to propagate immediately. The cache at some core can continue to use out-of-date data at the same time that the core is seeing and using updated values in other cache lines.

[78] Imagine a program that goes through memory byte-by-byte writing to sequential bytes, one-by-one. Each STORE instruction updates a single byte within some cache line. For each STORE, it is necessary to notify all other caches to make certain they invalidate any copy of this cache line they hold in their private caches. This overhead can impose a huge burden on bus traffic.

[79] Of course this total order must respect the order in which the LOADs and STOREs on any one core are performed.

The relaxed memory model of Blitz-64 allows the same cache line in the caches at two different cores to contain different data.[80] In short, the cache protocol can allow some core to continue to see an old, out-of-date value for some time, while other cores are seeing the new, updated value. This allows for improved performance, but opens the door to confusion when two cores are accessing the same address in a shared address space.[81]

Locking Example

To perform synchronization and concurrency control, all shared data really ought to be protected by locks. Program correctness cannot be guaranteed without proper locking.

> ***To accommodate the Blitz-64 memory model, a FENCE instruction should be used within the locking functions.***

The FENCE is used to make sure that all operations which are to be done after a lock is acquired are truly not begun until after the lock has been properly acquired. FENCE is also used before a lock is released to ensure that all instructions that should be executed in the critical section (i.e., before the release) are truly completed before the lock is freed.

This is required to prevent an out-of-order processor or the relaxed cache protocol from violating the locking protocol that programmers depend on.

[80] In Blitz-64, the following restriction holds: It must be clear which value is most recent. We do not allow two cores to STORE into a single location where neither STORE has precedence. While changes may not propagate immediately, the values retrieved *from a single location* are required to be sequentially ordered. Assume that one core stores 6 while another core stores 7 into the same location. Then either the 6 is stored first (in which case it is never possible for any core to read 7 followed by reading 6) or 7 is stored first (in which case it is never possible for any core to read 6 followed by reading 7).

[81] It would get even more problematic if the cache protocol allowed the updates to be propagated in an unconstrained, arbitrary order. It would then become possible that one core can see changes made in an order that is different from the order in which the other core actually made them. But as mentioned, this is not allowed in the Blitz-64 memory model.

As an example, consider the following code:

```
Acquire Lock[82]
    Wait for lock to become 0
    lock ← 1
    FENCE

Critical Section
    Access shared variable X

Release Lock
    FENCE
    lock ← 0
```

Let’s assume some shared data (which we will call ***X***) is protected by a lock, represented by variable ***lock***. We assume the usual locking convention that any core wishing to examine or modify ***X*** must first acquire the lock.

Imagine that core A grabs the lock and updates ***X***. While the lock is held, the value of ***X*** will pass through some “inconsistent states”, but before the lock is released, the core will set ***X*** to a “consistent state”, ready for other cores to see and use.

So what happens after core A releases the lock? It is perfectly legitimate for some other core (call it B) to acquire the lock and then retrieve the value of ***X***. But core B must not see an out-of-date value; it must see the final value of ***X*** and not some earlier, inconsistent state.

Updates to ***lock*** and updates to the shared variable ***X*** must propagate and become visible to all cores in a timely, controlled, and correct way. The FENCE instruction must be used to guarantee this.

What would happen without the FENCE instructions? The locking will not work properly, given the relaxed memory model of Blitz-64. Core A (which we assume initially holds the lock) will update ***X*** before releasing the lock and updating ***lock***. Unfortunately, core B might observe the update to ***lock*** before the update to ***X***. Such a scenario could allow core B to see a premature, inconsistent state of ***X***. This defeats the idea of locking and would be a total disaster.

[82] The “wait” and the “set lock” operations must be done together atomically, but those details are ignored here.

The FENCE instruction requires that all changes by one core must be propagated to all cores at the time of the FENCE.

In the code above, any core (such as A) acquiring the lock must set ***lock*** to 1 before ***X*** can be accessed. The FENCE instruction in the acquire code guarantees that all other cores will see the lock as being set — and must therefore be outside of their critical sections — before core A can proceed to access ***X***.

At the end of its critical section, core A updates ***X*** and then executes a FENCE before updating and releasing the lock. The FENCE in the release code guarantees that all changes to ***X*** will be propagated to other cores before core A can proceed to the instruction after the FENCE which then releases the lock. All changes to ***X*** must be delivered to other cores[83] before the code can begin to release the lock. Therefore, no other core can observe an older value of ***X*** after observing the updated (released) value of ***lock***.

In conclusion, if FENCE is used within the code to acquire and release locks as shown above, and if locks are always used to protect all shared data, then concurrently accessed data will be properly protected and behave as expected. The results obtained will be consistent with a linearizable memory model and the fact that the memory model is relaxed will become invisible.

More Discussion / Implementation

Clearly, the FENCE instruction must prevent any reordering of instructions on the core executing the FENCE. More precisely, memory operations (LOAD, STORE, FETCH) that appear before the FENCE instruction must not be moved past the FENCE instruction and memory operations that occur after the FENCE must not be initiated before the FENCE. In other words, the core executing the FENCE must not reorder instructions around the FENCE instruction.[84]

[83] Or, at least any old values must be made invisible and inaccessible.

[84] In theory, any data pre-fetched into the core's pipeline must be flushed whenever a FENCE is encountered since that might be out-of-date. But we can avoid clearing the instruction prefetch buffer as long as the prefetch buffer is cleared whenever the core issues a STORE that invalidates any line in the core's i-cache.

We must also ensure that any STORE appearing before a FENCE will be propagated to other cores so that no other LOAD or FETCH *on any core* after the FENCE is encountered can possibly retrieve an earlier, out-of-date value.

On the other hand, there is no possibility of another core prematurely getting a value STOREd after the FENCE, since the STORE instruction is not permitted to begin until after the FENCE.

We must also ensure that any LOAD or FETCH that occurs after a FENCE must retrieve the most up-to-date copy of any data. Obviously, there is no danger of a LOAD or FETCH that occurs before the FENCE getting a value that was STORED any time after the FENCE.

Both may require invalidating any cache lines that could possibly contain old, out-of-date data.

To illustrate, imagine a cache line that is held both in the cache of core A and in the cache of some other cores. Imagine that this line is updated by A before a FENCE instruction. (For example, the shared data might be a lock which is getting set, followed by a FENCE instruction before the critical section is entered.) The FENCE must cause any cache line that A updated to be either updated or invalidated in all other private caches that happen to contain that same line, since the other cores' caches might previously have contained different (i.e., older) values. (For example, they might have previously seen the lock as "unset" , but they must now see the lock as being "set".) After that, the only value observable by any core will be the value held in A's cache, which is necessarily up-to-date. This new value may or may not also appear in other caches — depending on whether the FENCE is implemented by invalidating the other cores' caches or by updating them with the new value — but no older value will be present in any cache.

The implementation gets more complex if it allows the possibility that a cache line updated by core A can subsequently migrate to another cache before the FENCE *without invalidating some other, older copies in other cores*. Failure to invalidate the contents in these other cores would violate the requirements of FENCE.

For example, there may be several variables coincidentally occupying the same cache line. Assume that, after A updates the cache line, some other core (call it B) grabs the line for some unrelated usage, thereby invalidating A's copy. Now B has the most recent copy and A no longer contains the line. On top of this, imagine that some other core (C) may happen to contain an older version of this very same cache line.

The cache line at C is old and out-of-date, but since no FENCE has occurred, this old cache line has not been invalidated. C is just looking at an older copy of the data. (Perhaps the cache line contains several variables, each protected by different locks. Since the locking protocols are respected, the data that C is seeing is not, itself, out-of-date at all. Only the data elsewhere on the line is out-of-date.)

Now assume A issues a FENCE operation. It required that this old, out-of-date cache line in core C must be invalidated, even though this cache line is no longer present in A.

In order to implement the Blitz-64 requirements correctly, it seems necessary to do one of the following:

(1) Invalidate any and all cache lines that could possibly be out-of-date in any core, whenever a FENCE instruction is executed on any core.

(2) Never migrate an updated cache line from one core to another without updating or invalidating any and all other copies. More precisely, if other private caches may contain older, out-of-date copies of the cache line, these same cache lines must either be updated or invalidated at the time the current, most recent value of the cache line is migrated from one core to another.

Option (2) is preferable in terms of performance.

Self-Modifying Code

Another issue that can arise with caches is the fact that cores often have separate caches for instructions and data. The so-called **i-cache** holds instructions and the **d-cache** holds data.

A program that does **instruction modification** alters its own code during execution. In other words, the code will STORE into some memory location and then FETCH from that same location at some later time, possibly on the very next instruction.

The kernel regularly writes data to pages that will be subsequently executed, so we must consider it to be self-modifying.[85]. It is not entirely uncommon for a user mode process to modify its own code while it executes, although this is often frowned upon.[86] In some cases, user mode code might even be prohibited from instruction modification.[87]

In any case, we could now have a situation in which data from the same address is held simultaneously in two different caches (the **i-cache** and the **d-cache**). Consequently, there is a need to synchronize these caches from time to time.

The FENCE instruction also guarantees that any writes to the d-cache will be propagated to the i-cache. After a FENCE, the i-cache must never hold out-of-date values. The FENCE instruction must also flush and invalidate any instruction that

[85] One example is the "exec" syscall, in which the kernel loads an executable file into memory, treating it as data. Later, the instructions that were loaded will be executed.

Another example is dynamically loaded I/O drivers. A kernel may download code from the internet to deal with some new device and, after moving this code into memory, invoke this code. Since no kernel rebuild or reboot is required, this is an example of self-modifying code.

A kernel capable of downloading kernel patches and dynamically applying them — while very risky from a security viewpoint — is a third example.

[86] One example involves the implementation of dynamically loaded library functions. The idea is that the CALL to such a function is actually directed at a dynamic loader; upon the first invocation, the function is loaded and the site of the CALL instruction is overwritten so that subsequent invocations go directly to the now-resident function.

Another example involves just-in-time compilation. The idea is that the original code is expressed in some high-level form; upon the first invocation, a resident compiler is called to generate machine code, which will then be executed. Both the code and the compiler might live and run within a single user mode address space.

Another more esoteric example might involve some research-oriented simulation program that uses a genetic algorithm to evolve code via natural selection, treating the address space as a sandbox environment. I'm sure there are other ideas I don't know.

[87] In the case of malware, a virus might enter the virtual address space of some innocent process as data and subsequently get executed. If the user process has some critical security clearance, then the malware code can do its dirty-work. This is an excellent reason to forbid security-critical processes from containing pages that are both writable and executable or ever adding "writable" privileges to any page.

was previously fetched and sitting in the pipeline (or some prefetch buffer) awaiting execution.

In a multi-core system, any modifications by the core executing the FENCE must be propagated to the i-caches — as well as the d-caches — of all other cores.[88] This includes instructions sitting in the pipelines of other cores.[89]

Within a single core, this could be implemented by having the i-cache constantly snooping. Whenever the CORE executes a STORE to a line that happens to be held in the i-cache, that line must be invalidated. Whenever an invalidate or update comes in from another core, the i-cache must respond, just as the d-cache must. Finally, whenever the i-cache must FETCH a new line, it must look in the d-cache as well as the L2 cache.

Invalidating Data in the Pipeline

At any moment in time, a core will contain a number of instructions which are in various stages of decoding and execution within the instruction pipeline. In particular, a core will normally contain an **instruction prefetch buffer** which contains a number of instructions which have been fetched from the i-cache but whose execution has not yet begun, as well as instructions that are in various stages of execution within the **execution pipeline**.

The instructions in the prefetch buffer may or may not ultimately get executed, depending on branch instructions and conditional execution, but they have been FETCHed from main memory, so they are effectively cached.

In this section, we talk more generally about all instruction and data bytes anywhere within the **instruction pipeline** of a core. The instruction prefetch buffer is one

[88] Since the i-cache cannot be written to, propagation in the reverse direction is not an issue.

[89] It is a far-fetched example, but consider a scenario where core A modifies an instruction in memory and executes a FENCE. Assume that core B has already fetched the previous value of this same instruction and it is already in core B's pipeline awaiting execution. A correct multi-core system must make sure that the old instruction is not executed after the FENCE has completed. A design in which the FENCE instruction simply delays enough cycles for the pipelines on all other cores to be completely exhausted is adequate to handle this, as long as the i-caches are also flushed as discussed.

example of data effectively pre-cached in a core's pipeline. A core may also speculatively prefetch data from the d-cache. In either case, the core's pipeline may contain a value that is effectively cached and may differ from values for that same byte that have been modified by other cores and held in other caches.

Of course, it is unacceptable to execute an instruction on core B after core A has modified that instruction and a FENCE instruction has been executed. Therefore when discussing the FENCE instruction, we must consider data present in pipelines.

Given that the instruction prefetch buffer has a finite size, we can assume that any data within the pipeline will be consumed within the execution of a small number of instructions. For example, let us assume that the prefetch buffer can hold 10 instructions and the execution pipeline can hold up to 5 instructions in various stages of execution. Then after the execution of 15 instructions, any old, out-of-date data in the pipeline will necessarily be consumed and subsequent instructions and data must be fetched from the i-cache and d-cache.

We expect the execution of the FENCE instruction to be fairly common; after all, the kernel must use it on every lock "acquire" and "release" operation and it's reasonable to assume that a minimum of four FENCEs are required on every core at every timer interrupt (i.e., every "tick").[90]

In the simplest design approach, the cache protocol would clear every core's pipeline for each and every FENCE operation on any core. It's probably unacceptable to introduce a 15 instruction delay into all cores whenever any single core issues a FENCE, but definitely not out of the question for simpler systems.

In another approach, the FENCE instruction would be implemented by simply making the core that executed the FENCE wait. The idea is that all processing on the core must be suspended until all inconsistencies involving that core's caches have had a chance to propagate. This gives any changes made by the core enough time to propagate to other caches so that no cache can contain an old, out-of-date values. Also, the core will need to wait until the other cores have had a chance to exhaust

[90] For example, the previously executing process must acquire a lock before changing its status from RUNNING to READY; the scheduler releases that lock and acquires the lock of another process in order to change its status from READY to RUNNING; and finally the new process must release that lock.

and consume all the pre-fetched data in their pipelines. So one core must wait for 15 instructions.[91]

If the implementation assumes that all data in a core's prefetch buffer is always present in its i-cache, a third, more sophisticated implementation of FENCE would be to empty the prefetch buffer only whenever any line in the core's i-cache is invalidated. That is, whenever a FENCE instruction from another core forces any line in a core's i-cache to be invalidated, then that core must also unconditionally empty its entire prefetch buffer.

An even more complex implementation[92] might involve keeping track of which cache lines are represented in the core's prefetch buffer. The idea is that the prefetch buffer would only be cleared when one of those particular lines is invalidated.[93]

Summary

We risk correctness unless we empty the caches and the execution pipeline of any core whose local, private caches might possibly contain an out-of-date data at the time of a FENCE.

We stipulate the following:

> ***A FENCE instruction on any core must invalidate out-of-date data kept in the private caches (both d-caches and i-caches) of all other cores. If data is invalidated in a core's private cache, and there is any possibility that the same data is also sitting in the core's execution pipeline, execution must stop and the pipeline must be cleared.***

[91] More precisely, the core must first wait until all cache inconsistencies are eliminated, then it must wait additional time (such as 15 instructions) giving the pipelines on other cores a chance to finish, so that no out-of-date data can possibly exist anywhere in any other core.

[92] Since code modification is fairly rare, the approach described in this paragraph seems like overkill.

[93] An even more targeted implementation would involve adding hardware to remember, for each byte in the pipeline, from exactly which cache line it originated. Then we could use this to limit pipeline flushing even further. The idea is that a "FENCE bubble" would be inserted into the pipeline of every core whenever a FENCE is executed on any core. If the pipeline contains a prefetched byte before that FENCE bubble which came from a cache line that has since been invalidated, we would only need to flush the pipeline from that point on, forcing a reload of the byte and its the new value. However, this sounds overly expensive and complicated.

More precisely, we are referring to any instructions (FETCHed from the i-cache) and any data (LOADed from the d-cache) present in the pipeline at any stage of incomplete, non-retired execution. This includes the prefetch pipeline as well as any data speculatively prefetched. The invalidated bytes and everything behind them in the pipeline must be removed. Instructions and data prefetched and sitting in front of the invalidated data is allowed to remain in the pipeline.

The simplest implementation is to clear every core's pipeline every time there is any FENCE operation. However, this could result in an unacceptable slowdown, since there might be a lot of unnecessary pipeline flushing.

We described a more reasonable implementation, which is to clear the entire pipeline but only whenever any line in the d-cache or i-cache is invalidated as a result of a FENCE operation on a remote core. If no line in the private caches is invalidated, there is no reason to clear the pipeline.[94]

Out-of-Date TLB Registers

The **Translation Lookaside Buffer (TLB)** registers are effectively a cache of data retrieved from memory, namely page table entries (PTEs) that have been cached in the TLB to improve performance. As such, the TLB may become out-of-date whenever changes are made to the in-memory page table.

> ***The FENCE operation is not required or expected to affect the TLB registers.***

Instead, the instructions — TLBFLUSH and TLBCLEAR — are used to invalidate TLB entries.

[94] It is crucial to note that this assumes that any byte in the pipeline that came from the core's local d- or i-caches must still be resident in the local cache and therefore subject to potential invalidation by some other core's FENCE operation. Fortunately, this requirement is easily met. If instructions and data in the pipeline always come from the i-cache and d-cache *and lines in these caches are evicted based on the least-recently-used algorithm*, then this requirement will be met. Since the pipeline is not too large, the relevant lines can not yet have been evicted.

> ***At this time, the Blitz-64 ISA only mandates that the TLBFLUSH and TLBCLEAR instructions only affect the TLB registers on the current core. However, this is still under consideration and the ISA may be modified.***

The alternative is to require a TLBFLUSH or TLBCLEAR operation to affect the TLB registers on all cores.

Consider the following scenario. Two user mode threads are executing simultaneously using a single, shared address space. Assume each thread is executing on a different core. Imagine that one thread requests a kernel operation that causes a change to the address space. As part of the operation, the kernel will naturally issue a TLBFLUSH or TLBCLEAR operation to get rid of old, out-of-date Page Table Entries (PTEs) cached in the TLB. But what about the TLB resisters on the other core? They must also be invalidated! Perhaps the first core must interrupt all other cores to request they execute TLBFLUSH/TLBCLEAR operations. But the other cores are most likely not using the affected address space. Such interruptions would be very common but will result in nothing but wasted time on all other cores!

Unfortunately, the obvious solution — to put all threads operating within a given address space on the same core — tends to defeat the very purpose of having more than one core.

To deal with multiple threads executing within the same virtual address space, another approach might be to create a unique address space for each thread. These address spaces will simply share all pages, although they each have a unique address space identifier (ASID). But this doesn’t really solve the problem: when one thread modifies the address space, the other threads (running on other cores) must still be notified so the same modifications can be made to their address spaces.

Chapter 10: Power-On-Reset

Quick Summary

- After power-on, certain registers will be initialized before execution begins.
- Execution begins with the **BootLoader** program in the **Boot ROM Area**.

Power-On-Reset

A “**power-on-reset**” occurs whenever:

- The processor core is first powered up
- The RESET button (if one exists) is pressed
- The RESTART machine instruction is executed

Before the first instruction is executed, hardware will set the following registers to these initial values:

csr_instr ← 0x0000_0000_0000_0000

csr_cycle ← 0x0000_0000_0000_0000

csr_status ← 0x0000_0000_0000_0001

Program Counter (PC) ← 0x4_0000_0000

With this value for **csr_status**, we have:

Kernel Mode:	**Enabled**
Interrupts:	**Disabled**
StackLimit:	**0** (i.e., any value of **sp** is okay)

The PC is set to the first word of the **Boot ROM Area**, which is the memory-mapped I/O area where the **BootLoader** program is stored.

All memory-mapped I/O devices will be sent a **reset signal** and will go into their initial states. Any pending interrupts will be cleared at power-on-reset.

If the core contains a **Secure Storage Area**, the **Secure Storage Limit Register** will be set to 0.

All other other programmer-visible state of the core (i.e., the general purpose registers and all other CSRs) will have undefined values.[95]

[95] Of course the read-only CSRs will have their expected values.

Chapter 11: Memory-Mapped I/O

Quick Summary

- Each I/O device is allocated one or more pages.
- The memory-mapped I/O pages are located in a dedicated region of addresses.
- The memory-mapped I/O region is 16 GiBytes (1 Mi Pages).
- The memory-mapped I/O region begins at address 0x4_0000_0000.
- Code running in kernel mode has full access to the memory-mapped I/O region.
- The I/O pages may optionally be mapped into virtual address spaces.
- The Boot ROM Area is treated as a memory-mapped I/O region.

Overview

The Blitz-64 architecture does not contain instructions that are dedicated to input or output.

Instead, all I/O devices are **memory-mapped**, which means they are accessed using LOAD and STORE instructions. Instructions can also be FETCHed from memory-mapped I/O regions. For example, instructions are fetched from the Boot ROM Area.

Each device is assigned to, and located within, one or more pages. In other words, the starting address for a device's address range will be page-aligned and the amount of address space the device consumes will be a multiple of the page size, which is 16 KiBytes.

In the layout of the memory-mapped I/O region, the various I/O devices will be ordered and laid out sequentially, one after the other. They will not overlap and different devices will be on different pages.[96]

The exact layout of the memory-mapped I/O regions is implementation-dependent.

Allocating the memory-mapped I/O address space in units of pages is mandated for the following reason: it allows the kernel to use address translation to map the pages into various virtual address spaces. At runtime, the Memory Management Unit will use page tables to map a LOAD or STORE from a virtual address to the physical address of the device. Thus, the kernel can use the paging mechanism to make an individual memory-mapped I/O device available to one address space, but hidden and invisible to all other address spaces.

In most cases, the device driver for a particular device will run as a user-mode program. The pages for the device being managed are mapped into the address space of the driver program. This approach frees the kernel from the overhead of dealing with many devices. More importantly, it allows device drivers to be dynamically loaded, started, and stopped in a safe fashion. If a device driver is buggy or contains malicious code, the damage is limited to the device in question; it cannot modify other devices or corrupt kernel memory. Moving most device drivers out of the kernel is critical for security, as well as flexibility.

Nevertheless, a few devices will undoubtedly be managed directly by the kernel. The pages for such a device would not be mapped into any virtual address space and the kernel code would address the pages directly.

Like normal memory pages, I/O pages that are mapped into virtual spaces may have any combination of permissions.

By not mapping a memory-mapped I/O page into a virtual address space, the kernel prevents user mode code from accessing the device. If a page is mapped, then it will be readable. In addition, the kernel may mark the page as writable and/or executable. Normally, the pages for I/O devices would be marked writable (allowing the code to update/alter/command the device) but not executable.

[96] It is allowed for implementation to place unused gaps between the regions, if this is convenient. If there is an expectation that a region will grow in subsequent implementations or that some devices may be implemented optionally in different versions, then those pages should be pre-allocated, set aside, and documented as such.

In this document, we do not fully specify the nature of all I/O devices available on a Blitz-64 system. In fact, different implementations will have different devices. In other words, which devices are present and how they function will vary between implementations.

Each implementation must specify:

- Which I/O devices are present
- Where each device is located in the physical address space
- How many pages are allocated to each device
- Exactly how the device functions and how it is used

Below is an example placement of memory-mapped I/O devices. (This happens to be the default memory map for the devices implemented by the Blitz-64 emulator.)

		Size		
Device	**Starting Addr**	**Hex**	**Bytes**	**Pages**
Boot ROM Area	4_0000_0000	10_0000	1 MiBytes	64
Secure Storage Area	4_0010_0000	10_0000	1 MiBytes	64
PLIC	4_0020_0000	4000	16 KiBytes	1
UART	4_0020_4000	4000	16 KiBytes	1
DISK	4_0020_8000	4000	16 KiBytes	1
DMA Device	4_0020_c000	4000	16 KiBytes	1
Host Device	4_0021_0000	4000	16 KiBytes	1

The mapping of I/O devices is implementation dependent and will vary. For example, an implementation might also include mappings for devices such as:

- Digital I/O Pins
- MicroSD Card
- HDMI
- USB
- WiFi
- Adjacent Core Links
- Lock Controller

Typical I/O Devices

The following devices are common and expected to be implemented on all or almost all Blitz-64 systems. For greater detail on the devices mentioned in this section, consult the document

> *"Blitz-64: Memory-Mapped I/O Devices"*

The **Boot ROM Area** contains the code that will be executed on startup. Upon power-on-reset, the program counter (PC) is set to the first address in this area. This area is populated with preset memory bytes (i.e., ROM) which cannot be altered.

The **Secure Storage Area** would be present in any complex system and contains additional code used during the bootstrapping process. It contains firmware, which means that the bytes are retained when the power is off, but which are also updatable (e.g., EEPROM).

The **Platform-Level Interrupt Controller** (PLIC) device manages interrupts coming other devices (such as the UART or DISK devices) and channels interrupts to one or more cores.

The **Universal Asynchronous Receive Transmit** (UART) device allows individual 8-bit characters to be sent and received over a serial channel.

This **DISK device** implements some form of long-term stable storage, such as a disk or flash drive.

The **Direct Memory Access** (DMA) controller is capable of the following tasks:

- Move a large block of memory
- Zero a large block of memory
- Perform secure hashing (using SHA-256)
- Perform AES encryption and decryption

The **Host Device** is implemented only in the Blitz-64 emulator. It provides a sort of “back-door” to the underlying Unix/POSIX system running the emulator. This “device" would not be included in a physical implementation. The Host Device provides the following functionality:

- Ability to access command line arguments
- Ability to access the date and time
- Ability to perform file I/O (**fopen**, **fgetc**, **ungetc**, **fputs**, **fseek**, ...)

Boot ROM Area

This is the location of the “**BootLoader**”, the initial program which is executed when the core is powered up. As read-only-memory, all LOAD and FETCH operations function the same as any memory, but STORE operations are ignored.

The BootLoader program and the exact contents of this area are implementation dependent. The BootLoader program will be tailored to the system containing the Blitz-64 core.

The starting address for this region is mandatory and fixed. The number of bytes actually implemented may be less than the 1 MiByte range — this is implementation dependent — although setting aside the entire region as specified above is strongly recommended.

In a multi-core system, there may be one Boot ROM Area shared by all cores or each core may have its own separate Boot ROM Area. This is implementation dependent.

The intended use of this area and various considerations are discussed in the chapter “The Secure Boot Sequence”. That chapter explains why the Boot ROM Area should be implemented with ROM[97] and why this area should not implemented with flash memory, which can be updated.[98]

Secure Storage Area

The Secure Storage Area is used to make certain the boot process is secure and that malicious code cannot gain a foothold during the boot process.

The Secure Storage Area should be implemented if secure booting is wanted. However, simpler systems may leave this area unimplemented.

With secure booting, the boot process is a two-step process. The first stage is the **Low Level Boot Loader (LLBL)**, which is the program residing in Boot ROM. The LLBL will invoke the **Second Stage Boot Loader (SSBL)**, which is a program residing in flash memory, and more particularly in the Secure Storage area described here.

It is the SSBL that will actually access the file system, locate the OS kernel's executable image file, load it, and jump to the kernel's entry point. The SSBL is considered to be "firmware", since it can be updated.

The LLBL is only concerned with managing security and the Secure Storage Area, including updates to the Secure Storage Area and the SSBL.

The Secure Storage functions much like main memory: It is a large chunk of byte-addressable memory that can be read and written, and instructions can be fetched from this area, as well. The number of bytes actually implemented may be less than

[97] Read-Only Memory (ROM) is memory whose contents are fixed and cannot be altered.

[98] As a practical manufacturing concern, the Boot ROM Area might be implemented with flash-type memory that can be loaded when the system is manufactured. The key point is that once written and released into the field, the Boot ROM Area should not be modified by instruction execution or any programmatic behavior of the system.

For example, the Boot ROM Area might be writable using an electrical connection that is made during the manufacturing process. But once the product is delivered, the Boot ROM Area must not be modifiable without physical contact to directly manipulate the device. The inviolability of the Boot ROM Area is crucial for system security and integrity. However, the measures described in this ISA do not attempt to protect a Blitz-64 system against direct physical meddling.

the 1 MiByte range — this is implementation dependent — although the range itself is mandatory and fixed.

In terms of reading data and fetching instructions, it functions exactly like other memory: there are no special restrictions and any byte can be retrieved.

In terms of writing, there is a crucial difference with main memory. The Secure Storage area has **a lock capability** which allows bytes to be "write-protected". The bytes can be written until they become locked and after that, writes are ignored. From that time on, the memory functions like ROM.

The write protection is controlled by a special register called the "**Secure Storage Limit Register**". This is a 64 bit register containing an address. Any byte located below the Secure Storage Limit is locked and cannot be updated. Any byte whose address is greater than or equal to the Secure Storage Limit may be updated freely and without restriction.

The Secure Storage Limit Register is set to zero on startup. That is, a power-on-reset will initialize the Secure Storage Limit Register to 0x0000_0000_0000_0000. A value of zero implies that the entire Secure Storage area is unlocked. Any byte can be written.

The Secure Storage Limit register is mapped to the first doubleword of the Secure Storage Area. That is, the first 8 bytes of the Secure Storage Area are special in that any write to them is a write to the Secure Storage Limit Register. The remaining bytes of the Secure Storage Area function as described above. The Secure Storage Limit Register is readable and can be obtained at any time by reading the first 8 bytes of the Secure Storage Area.[99]

Note that since the Secure Storage Limit Register is at the beginning of the Secure Storage area, if any of the Secure Storage area is locked, then the Secure Storage Limit Register itself will be locked. The first write to the Secure Storage Limit

[99] The Secure Storage Limit register is a 36-bit doubleword aligned address. Any other bits outside of [35:3] shall be ignored for the purpose of imposing write-protection. Presumably, all 64 bits will be written into the first 8 bytes of flash and all bits will be retrievable, but this is not required.

register will lock some or all of the Secure Storage area, but will surely lock the Secure Storage Limit Register itself.[100][101][102]

In a multi-core system, there may be one Secure Storage area shared by all cores or each core may have its own separate Secure Storage area. This is implementation dependent.

The intended use of this area and various considerations are discussed in the chapter "The Secure Boot Sequence".

[100] A write to the register with an address below the start of the Secure Storage area would be possible, but would accomplish nothing, so we ignore such a thing.

[101] We envision that the Stack Limit Register will be implemented as follows. The core will contain a separate register, which is not a Command and Status Register (CSR) and which is not directly accessible. Whenever a STORE to the Secure Storage Area is attempted, the core will check the target address against this register and inhibit the operation if it violates the constraint. Otherwise the STORE will function like any memory STORE. Additionally, if a STORE is attempted in which the target address happens to be the first address of the Secure Storage Area and the constraint is not violated, the Stack Limit Register will also be updated, in additional to the normal writing of the data to the Secure Storage Area itself. With no loss of generality, the Stack Limit Register can be limited to 35 bits.

[102] A specific implementation may limit the Secure Storage Limit Register to holding only certain values. For example, the register may be required to be (say) page aligned. Such a restriction is an implementation dependency. If a write is made to such register, then the actual value stored will be rounded up to the next legal value. In this example, the value will be rounded up to the next page boundary. Such a limitation might be useful to accommodate the nature of the non-volatile storage being used.

Chapter 12: The Secure Boot Sequence

Quick Summary

- After power-on, certain registers will be initialized before execution begins.
- Execution begins with the **BootLoader** program in the **Boot ROM Area**.
- Details of the BootLoader program are implementation dependent.
- Security issues around the boot process are discussed.

The BootLoader Program

It is assumed that a program (called the **BootLoader**) has been pre-installed in the Boot ROM Area.

Upon start-up (i.e., a "power-on-reset"), instructions will be fetched from the Boot ROM Area, beginning with the instruction stored in the first word of the Boot ROM Area. Thus the entry point of the BootLoader after a power-on-reset is its first word, located at address 0x4_0000_0000.

A "**warm reboot**" (also called a "**soft reset**") occurs when the kernel branches into the BootLoader directly, with the intent to reboot exactly as if a power-on-reset had occurred. This branch is made to a second entry point, in case there are subtle distinctions between cold and warm booting.

After a kernel crash (e.g., a "Hardware Exception"), the trap handler may end by branching directly to the BootLoader. In this case, a third entry point is used. This is called the "**kernel-crash**" entry point.

BootLoader Entry Points

Power-on-reset entry point:	0x4_0000_0000
Warm-reboot entry point:	0x4_0000_0008
Kernel-crash entry point:	0x4_0000_0010

In all cases, the behavior of the BootLoader will be almost identical.[103]

Commentary Since instructions can be fetched from the Memory Mapped I/O area, there is no need to remap the physical address space, as is done in some systems. Also, there is no particular reason to make the BootLoader code relocatable.[104]

We don't specify the exact behavior of the BootLoader program here, but perhaps it will begin by probing the physical memory to determine the size of installed physical main memory.

[103] To support kernel development, the code at the "kernel-crash entry point" might be specially tailored for debugging.

[104] Relocatable code is code that will run correctly regardless of where it is placed in memory. This is done by making all addresses PC-relative. For the purposes of debugging, it may be useful to make the BootLoader relocatable. During debugging of the BootLoader itself, it might be convenient to place the BootLoader in a writable area of main memory, to accommodate breakpoints, and so forth.

Probing Memory to Find Its Size

The size of installed physical memory is assumed to be a power of two, e.g., 256 MiBytes. Thus, it will take about 34 probes to determine the installed memory size:

2^{11} = 2 Ki
2^{12} = 4 Ki
2^{13} = 8 Ki
2^{14} = 16 Ki
…
2^{44} = 16 Ti

To probe a memory doubleword, the BootLoader should:

- Read the previous existing value and save it.
- Write 0xFFFF_FFFF_FFFF_FFFF.
- Read the value back & check that it is unchanged.
- Write 0x0000_0000_0000_0000.
- Read the value back & check that it is unchanged.
- Restore the previous value.[105]

By convention, the BootLoader is free to use the uppermost 1 MiByte of physical memory for its R/W data. The kernel image will be loaded into low physical memory so there should be no overlap.

[105] The reason for saving the pre-existing value is that the BootLoader may be invoked after a kernel crash and the pre-existing memory contents may be important. For example, the BootLoader may be passed a pointer to an area of memory where the kernel has stored information about the crash. This may include register state, as well as other data. The BootLoader may be tasked with displaying this info before rebooting. The BootLoader might also need to pass this information on to the reincarnated kernel after the reboot. The re-incarnated kernel may enter a "kernel debugging mode" in which the previous contents of the memory can be queried. In either case, the BootLoader must preserve the pre-existing memory contents.

Before loading the kernel, the BootLoader may perform other operations, such as:

- Check and verify that all physical memory bytes function correctly.
- Initialize various memory-mapped I/O devices.
- Turn on LEDs, e.g., to indicate the core is booting.
- Print messages on the display or serial UART line.
- Allow for interactive use, debugging, selection of kernel source, etc.

The BootLoader will determine on which device the kernel image is stored (e.g., on internal flash memory or on an external microSD card) and read the kernel into main memory.

Presumably, the kernel will be loaded starting at location 0x0_0000_0008.[106] Note that there is no privilege checking for physical memory: All pages have FETCH, READ, and EXECUTE permission.

By convention, the kernel will contain the following entry points:

Power-on-reset entry point:	0x0_0000_0008
Warm-reboot entry point:	0x0_0000_0010
Kernel-crash entry point:	0x0_0000_0018

The kernel (as stored in an executable file to be loaded), will contain additional information, generally including:

- size
- entry point[107]
- error checking code

The BootLoader will check to make sure the kernel was loaded correctly and the computed error checking code matches the expected value. Note that the error checking mentioned here (in which the expected code value is read in from an external source along with the kernel image) is only useful in guarding against accidental, non-malicious errors, such as data corruption due to transient electrical noise. Any malicious user who can corrupt the kernel image will also update the error-checking code mentioned here.

[106] Recall that the first 8 bytes of memory are reserved and never used. Any attempt to access the first 8 bytes will result in a Null Address Exception.

[107] This should be equal to the Power-on-reset entry point, 0x0_0000_0008.

Finally, the BootLoader will complete by branching to the kernel's entry point.

We have just described a straightforward booting chain, which doesn't involve multiple boot phases. While nothing precludes a complex boot chain, the approach described above will be adequate for many systems.

The BootLoader should not contain functions that are used by the kernel. The reasons for this are (1) Different implementations will have different BootLoader programs. Depending on the BootLoader code would tie the kernel to a specific implementation. (2) There may be performance issues. The BootLoader is located in the Boot ROM, which is in the memory-mapped I/O area. Thus, the ROM is functioning as a sort of I/O device and may not operate as quickly as main memory.[108]

There is one situation in which it may be acceptable for the kernel to invoke functions residing within the BootLoader code. When the kernel fails catastrophically (e.g., a Hardware Exception occurs), the BootLoader I/O functions might be used to print error messages. The BootLoader may have a specific interface for use during kernel errors. The BootLoader may contain a primitive user-interface to allow some state to be recovered from the crashed kernel. For example, a branch to the kernel-crash entry point may assume that the registers contain certain values, such as:

- Numeric crash code, indicating the nature of the crash
- Pointer to area of memory containing additional data
- Size of memory area

These values could be passed as parameters to the re-incarnated kernel, for use in debugging and crash reporting.

A warm reboot (i.e., soft reset) occurs whenever a program branches back to the BootLoader, i.e., to address 0x4_0000_0008. Before doing so, the core must be in kernel mode and have interrupts disabled.

[108] For example, instructions fetched from the ROM might not be cacheable in the i-cache. There is no particular need to make the BootLoader run quickly since its performance will almost always be limited by the time required to read the kernel image from an external device. Therefore, any code within the BootLoader may not execute at a speed acceptable for kernel performance.

Note that the power-on sequence may result in some I/O devices receiving a “reset” signal. Such a reset signal is not assured during a soft reset. The kernel should contain code during its initialization phase, to query and reset all I/O devices, in order to avoid complications during a soft-reset. Upon soft-reset, neither the BootLoader nor the kernel can assume that the I/O devices have received their proper reset signals.

For this reason, it is usually preferable to execute the RESTART instruction, rather than branch to the “Warm Reset Entry Point”.

The BootLoader is free to pass information to the Kernel. The BootLoader can do this by initializing some variables in the global static data area of the Kernel, which is at the beginning of memory, or the BootLoader can pass parameters directly in registers.

For example, the BootLoader would normally pass the size of installed memory, information about the hardware configuration that the kernel will find itself running in, and possibly information about the current state of various I/O devices or information about a previous kernel crash.

Contrast with Traditional Booting

In some computers, code in the BIOS will read in the Master Boot Record (MBR) and then jump to code in the sector just loaded. In other approaches, the firmware itself will be capable of understanding the file system and will ignore the code within the MBR.

In some systems, there is a “boot chain”, in which there is a sequence of programs executed one after the other, until finally the full kernel is loaded and executed. For example, the BIOS reads in the MBR; the code in the MBR reads in another (second level) boot loader from the disk; and then the second level boot loader reads in the kernel.

In any case, the first step must necessarily involve executing code stored in some form of non-volatile memory and that code must be capable of understanding, controlling, and reading from any device from which the system can boot. This is true of Blitz-64 as well.

The Blitz-64 architecture does not mandate whether there shall be a complex, multi-phase boot chain, or whether a simple program burned into on-chip ROM will do all the work of loading and starting the kernel.

Security Issues Around Booting

We know that the Master Boot Record (MBR) in traditional systems was a point of vulnerability and a potential target of malware. If the MBR becomes corrupted by malware, it can open the door for a corrupted version of the kernel to be loaded.

However, the entire boot chain including the code in the ROM — whether it is the BIOS of a traditional system or the BootLoader code in Blitz-64 — is also very critical, perhaps even more critical than the MBR. If the BootLoader program has been maliciously tampered with, then nothing that executes afterward can be trusted.

Because it is the first code that executes, the BootLoader is therefore the most trusted piece of software in the computer system, more trusted than the kernel itself. The kernel can trust that the BootLoader will “do the right thing” when executed. But the BootLoader must be very cautious about trusting any kernel code or functionality.

For security reasons, the BootLoader should avoid communication with other entities. (For example, it is very risky to receive instructions or commands over the internet.) If the BootLoader must communicate, the security and integrity of the communication and identity of the other parties must be carefully and securely verified. Otherwise the bad guys can impersonate legitimate sources and can send commands that exploit weaknesses in the BootLoader.

In all ISAs, careful thought must given to guarding against malware. In systems using the Blitz-64 architecture, the BootLoader must remain secure at all times. Putting the BootLoader in firmware[109] — as opposed to ROM — must be done with utmost

[109] ROM means Read-Only Memory: once data is placed the memory, it cannot be modified. Once written, it cannot be altered. By “firmware”, we mean a non-volatile memory device that will retain its data even when power is turned off, but that can be altered or re-programmed. Flash memory is an example.

attention to security, since it may inadvertently create a pathway for the kernel to be compromised.

To emphasize that the memory containing the BootLoader should be implemented with ROM and not some form of updatable memory, the memory-mapped I/O region is named the **Boot ROM Area**.[110]

The key to avoiding firmware is to keep the BootLoader small, simple, and correct. Adding a bunch of code to the BootLoader is not a good idea, since it creates a need for some mechanism to patch the code.

Of course placing some of the BootLoader code in firmware, which can be updated under program control, is very convenient because bugs can be fixed and support for new devices and greater functionality can be added.

The cost of placing the BootLoader entirely in ROM is **programming discipline**: Any bugs with a ROM-based BootLoader cannot be fixed, so the BootLoader must work correctly and be bug-free.

If, instead, the BootLoader is implemented as firmware, then once the kernel is compromised — even once, for a very short time — security on the entire system is lost forever. Using firmware necessarily increases the security risk. During any malicious and successful attack on the kernel, we must assume the malware has updated the BootLoader program, replacing it with a malicious version which will do the bidding of the malware upon every future power-on-reset. Thereafter, the boot process is forever compromised and no future kernel can be trusted.

However if the BootLoader is placed in unalterable ROM, then malware cannot persist beyond a power-on-reset. If a kernel is found to have security bugs and a security breach occurs, then of course it is a bad thing and perhaps the kernel code is forever compromised. But a repaired kernel can be created and distributed to repair the security flaw. And the BootLoader can be relied upon to load the new, corrected kernel correctly.

The BootLoader must not contain secret data. The BootLoader code is fully visible to the kernel and may become visible to arbitrary programs, through bugs, malware, or oversight. It should be assumed that every byte of the BootLoader is in the public

[110] This document cannot control how the Blitz-64 architecture is implemented and does not explicitly prohibit the **Boot ROM Area** from being implemented in updatable, non-volatile memory.

domain, and, in the spirit of open software, it is even encouraged. Any idea of keeping the BootLoader code confidential as a security measure is misguided.

As we should all remember, “security through obscurity”, is not security at all.

For security purposes, the BootLoader may validate the kernel after loading it into memory. For example, the BootLoader may compute a secure hash of the kernel image and compare it to a known value. This ensures that the kernel image is what is expected and the executable file containing the kernel has not been modified in any way.

However, the question is: Where is this “known value” to be kept? There are several possible answers:

- The user is required to type the expected secure hash value in to the BootLoader. This is the most secure, but requires the most effort by the user. Possibly appropriate for military-level security.

- The BootLoader displays the secure hash value and asks the user to verify its correctness before branching to the kernel. This is not reliable, since users will tend to ignore such messages and “accept without reading”.

- The BootLoader keeps the expected value in some form of stable, nonvolatile storage. While most convenient to the user, this nonvolatile storage becomes a critical component which must be protected. If there is any possibility it can be altered by anything but the BootLoader, system security will be compromised.

Various approaches to BootLoader / Kernel security and verification, which make use of the “Secure Storage” area, are discussed later in this chapter.

Simple Systems

In some embedded systems, there may be no OS at all; perhaps all code will be “set in stone” and not updatable as well. This would be particularly desirable for systems that must be impervious to malware. In such cases, the entire code base might reside in flash memory or even in the ROM itself.[111]

ROM-Only Systems

In the simplest **ROM-only system**, there will be no kernel and all code will be burned into ROM. This might be appropriate for a very low cost, mass-produced microcontroller.

It might also be appropriate for military weapons systems and other critical embedded applications, in which extreme efforts must be taken to prevent any and all cyber-attacks.

In a ROM-only system, all code resides within the Boot ROM Area and there is never any branch to other areas of memory. The main memory area (i.e., bytes within the first 16 GiBytes of the physical address space) will only be used for storing variables and data.

Flash-Based Systems

In a **Flash-based system** design, the ROM-based code is solely devoted to loading a program into flash memory. Subsequently, on every power-on-reset, the flash-based code will execute.

This accommodates a model similar to that used for the **Arduino**. The on-board loader code is permanently fixed in the ROM and the various application programs are placed in the Secure Storage Area, which is implemented with flash memory.[112]

If the loader program in ROM detects a working connection to a host computer at power-on-reset, it can download and overwrite the flash with a new program.

[111] A system with no flash might also be appropriate for simple, low-cost systems where the additional cost and complexity of an OS is not worth it. But the real benefit is that maintenance costs are eliminated. It is ridiculous to have to deal with firmware upgrades for (say) headphones. And if there is no possibility of updates, there is no possibility of breaking the system with an update, which has become an increasing plague upon us.

[112] By “flash memory” we mean any form of updatable, nonvolatile memory.

Otherwise, the loader program in ROM will branch directly to the program last stored in flash. This is basically the Arduino model.[113]

Single-Stage Bootstrap Systems

In another system design, booting the kernel will be a single-stage process.

The Boot ROM Area will contain a BootLoader program. This BootLoader will locate the kernel on some other device and will load it directly. The Secure Storage will not be used.

This approach might be appropriate for a **Single Board Computer (SBC)**, which will always boot from a microSD card.

This sort of design might also be appropriate for an embedded application such as an automobile, airframe, or weapon system, where the system must be entirely isolated from the Internet, in order to prevent any possibility of cyberattack. A multitasking kernel is needed to control various complex and interacting functions. However, because of the complexity, bugs and modifications to all parts of the code must be accommodated. So all code — i.e., the kernel and the filesystem — are placed on a microSD card. If upgrades and/or bug fixes are required, the microSD card is simply removed and replaced.

Multi-Stage Boot Processes

In modern operating systems, the kernel is updated periodically and is distributed to the users. We assume the kernel originates from a **trusted source** but is transmitted over **insecure communication channels** to end-users. The communication and bootstrapping must be done in such a way that the kernel cannot be compromised by bad guys at any step of the process.

We assume that both the communication channels and the files on the user's local file system may be intentional hacked, corrupted, and modified in an attempt to

[113] In a related development model, the program resides in flash memory, but updates to the flash are performed by plugging in a microSD card, rather than through a communication channel. At power-on-reset, the ROM-based code will detect that a card is present and will then update the flash from data on the card.

install and boot a malicious version of the kernel, and we must protect against this. We also assume that new devices (requiring new device drivers) are periodically added and that the kernel must boot using driver code that will be written and distributed in the future.

To achieve these goals, booting the OS kernel will be a multi-stage process.

The BootLoader code in the Boot ROM Area is called the **Low Level BootLoader (LLBL)**. The BootLoader code will locate and pass control to the **Second Stage BootLoader (SSBL)**. The Second Stage BootLoader resides in non-volatile storage which, in Blitz-64, is called the "Secure Storage Area". The Second Stage BootLoader will locate the OS Kernel, load it in to memory, and pass control to it.

The idea is to keep the Low Level BootLoader as simple as possible and place all complex functionality in the Second Stage BootLoader.

A "power-on-reset" occurs whenever the system is initially powered up. If the system contains a RESET button, then pressing this button will also initiate a power-on-reset. This can also be triggered by the execution of the RESTART machine instruction.

A power-on-reset will have this effect:

> The PC will be loaded with 0x4_0000_0000.
> The **csr_status** register will be initialized (kernel mode, interrupts disabled).
> Any pending interrupts will be cleared.
> The Secure Storage Limit register will be set to 0.

The BootLoader is a high-security, high-trust system component. If it contains bugs or security vulnerabilities, these might be exploited to load a compromised OS kernel. Furthermore, since the BootLoader is in ROM (not flash), it cannot be fixed or updated. Any flaws it contains will be with the device forever. Putting unnecessary or complex functionality into the BootLoader is risky and ill-advised.

It is assumed that the Boot ROM and the core(s) are bundled together and will often be on the same silicon. The Boot ROM is therefore the obvious place to put constant and unchanging information about the system's design and configuration. This would include information about main memory sizes and details about memory-mapped I/O devices.[114]

Commentary

It is extremely difficult for a core CPU to distinguish between private and shared memory. From the point of view of a single core, a byte of memory functions the same regardless of whether or not it is accessible by another core.

A logical thing to place in the Boot ROM is information about the memory system. In particular, the starting address of the shared memory is critical. The BootLoader can determine how much memory is installed by the use of STORE-LOAD cycles to determine whether there is functional memory at a given address. However, having additional information in the BootLoader might make this process smoother.

Another critical piece of information is the location and size of the Secure Storage area. The Secure Storage is another memory-mapped I/O device that is used for an additional level of security in a multi-level boot chain.

As mentioned above, a BootLoader can simply go find something that looks like an OS kernel — perhaps on some removable microSD card or disk drive — load it, and jump to it.

But in order to implement any level of security for the boot process, something more is required. With Blitz-64, this is supplied by the Secure Storage device. The BootLoader program will access Secure Storage to implement the secure booting protocol.

[114] Although the BootLoader can determine how much memory is installed by the use of STORE-LOAD cycles to determine whether or not there is functional memory at a given address, having this information in the BootLoader might make the process smoother. Furthermore, it is extremely difficult for a core to distinguish between private and shared memory. From the point of view of a single core, a byte of memory functions the same regardless of whether or not it is accessible by other cores. Therefore, it might make sense to place information like the starting address of the shared memory in the Boot ROM.

In a single-stage boot process with no security[115], a minimal BootLoader will only need to perform the following tasks:

- Perform basic machine start-up and error-checking
- Locate the kernel image
- Read the kernel image from an external source
- Load the kernel image into memory
- Jump to the kernel entry point
- Optionally, the BootLoader might pass data about previous kernel crashes to the new kernel

However, the following operations require substantial amounts of code:

- Interface with complex I/O devices, where the kernel might be located
- Understand complex file formats, in which the kernel might be stored
- Implement various cryptographic techniques to verify kernel integrity
- Provide a facility to securely update firmware
- Implement a user interface
- Deal with complex I/O devices for user interaction (e.g., USB, HDMI, Bluetooth)
- Provide a debugging facility to deal with kernel crashes

For this reason, a Second Stage BootLoader (SSBL) is anticipated. Presumably, the Second Stage BootLoader will be firmware, meaning that it will be stored in the Secure Storage area. In this case, the Low Level BootLoader (LLBL) in Boot ROM will:

- Perform basic machine start-up and error-checking
- Pass machine-specific parameters to the SSBL
- Jump to the SSBL entry point
- Manage firmware updates in a secure way[116]

[115] This is appropriate for a device that is (1) not-connected to the Internet, (2) not expected to have software updates, and (3) not mission-critical. Think: dishwasher, refrigerator.

[116] This might also include dealing with firmware corruption and/or firmware rollbacks.

The ISA pre-allocates are block of 1 MiByte for the Boot ROM area and 1 MiByte for the Secure Storage are. The amount of memory actually installed is implementation dependent.[117]

The Secure Storage Area

Next, we describe how Secure Storage is intended to be used.

Upon power-on-reset, the Secure Storage area is assumed to contain the Second Stage Boot Loader (SSBL) program in the lower portion and unused bytes in the upper portion. The Secure Storage is initially unlocked.

The Secure Storage Area

The Secure Storage area works as follows.

The Secure Storage area is implemented as a block of non-volatile memory (i.e., flash memory) which is mapped into a memory-mapped I/O region.

In addition, there is a Secure Storage Limit Register, which is mapped into the first doubleword of the Secure Storage area.

The Secure Storage memory has two states: “locked” and “unlocked”. In the locked state, the memory can only be read, while in the unlocked state, it can be modified.

More precisely, all Secure Storage bytes below the current value of the Limit Register are in the locked state and cannot be modified. All bytes above the Limit Register are unlocked and can be freely read and modified.

Upon power-on-reset, the Secure Storage Limit Register is initialized to zero, which puts all the Secure Storage bytes in the unlocked state. A subsequent STORE into the Secure Storage Limit Register will switch the Secure Storage to the locked state. More precisely, a write to the Limit Register will make the first portion of the Secure

[117] Two MiBytes is a tiny fraction (1/8,192) of the available memory-mapped I/O address space. We aim to keep the BootLoader as small and simple as possible, so this size should be adequate. But setting aside larger regions for the ROM or Secure Storage presents no conceptual issue.

Storage locked. The exact value written to the Limit Register determines how many bytes are to be locked and how many are to remain unlocked.

Since the Limit Register itself occupies the very first bytes in the Secure Storage area, once it is written to, the Limit Register itself will also be in the locked region, preventing any further changes in which portion of the Secure Storage is locked and which is unlocked.

Since the Limit Register is no longer modifiable, the Secure Storage area will remain locked as long as the device is powered up.[118]

The Low Level Boot Loader program will verify the Second Stage Boot Loader program is correct. If everything looks good, it will proceed to lock the lower portion of the Secure Storage area and branch to the Second Stage Boot Loader, leaving the upper portion of the Secure Storage area updatable.

Since the Low Level Boot Loader is in ROM, it must be reliable and cannot be repaired or replaced. Therefore, it really should not interface with I/O devices. The I/O devices connected to a processor may vary from system to system. Furthermore, I/O devices change over time and require updates to software drivers.

The Second Stage Boot Loader (SSBL) is expected to be a large and complex piece of software.

It will, among other things, validate the OS Kernel. It must check to make sure the OS Kernel has not been tampered with or altered by malware. Thus, it must securely maintain and protect the secure hash keys[119] of the various kernel versions that it knows about. If there are issues, it must interact with the user, e.g., to install new kernel versions, or roll-back to earlier kernel versions.

To perform its duties, the Second Stage Boot Loader (SSBL) will need to contain information about:

[118] More precisely, the Limit Register itself will remain locked after any write to the register with any value greater than or equal to 0x4_0010_0008 until a power-on-reset signal is received.

[119] To quote Wikipedia, "A cryptographic hash function … is a mathematical algorithm that maps data of an arbitrary size (often called the 'message') to a bit array of a fixed size (the 'hash value', 'hash', or 'message digest'). It is a one-way function, that is, a function for which it is practically infeasible to invert or reverse the computation. Blitz-64 primarily uses the SHA-256 secure hash function, which produces a 256 bit (32 byte) key.

- The devices where a kernel might be stored.
- The file systems on those devices.
- The user interface devices and interfaces.

Periodically, updates to the Second Stage Boot Loader code will be required, for these reasons:

- A new version of the OS Kernel is distributed with a new secure hash key.
- A new file system has been implemented and the SSBL must interface to it.
- A new device has been implemented and the SSBL must interface to it.
- Changes are made to the user interface used by the SSBL.
- A bug in the SSBL must be repaired.

Periodically, messages must be sent to the firmware. These messages must be acted on before the Secure Storage area is locked, or else they can have no lasting effect. However, since the Low Level Boot Loader will lock secure storage before it branches to the Second Stage Boot Loader and since the Low Level Boot Loader will never access any I/O devices, the messages must be passed in the Secure Storage area itself.

Every update to the Second Stage Boot Loader will follow these steps:

- While running the OS kernel, some app will move a newly received message into Secure Storage at an address above the current Secure Storage Limit.
- A power-on-reset is required, reseting the Secure Storage Limit register and making the entire Secure Storage area updatable.
- The Low Level Boot Loader in Boot ROM will run, before any other code runs.
- The Low Level Boot Loader will see the message previously stored in the Secure Storage area and will process it.
- Using public-private (asymmetric) encryption, the Low Level Boot Loader will verify that the message is from a trusted authority. Using a secure hash function (such as SHA-256), it will verify that the message has not been tampered with.
- The Low Level Boot Loader will update the Secure Storage area as directed by the message. This could be in the form of replacing the Second Stage Boot Loader code, or by adding new secure hash keys for new versions of the OS kernel.
- The Low Level Boot Loader remove the message from the Secure Storage area.

- The Low Level Boot Loader will lock Secure Storage by writing to the Secure Storage Limit register.
- Finally, the Low Level Boot Loader branch to the Second Stage Boot Loader.

When the next power-on-reset occurs, the Low Level Boot Loader will see that there is no new message. It will then lock Secure Storage and branch to the Second Stage Boot Loader program.

It is likely the Secure Storage and the core will be integrated and located on the same chip. Nevertheless, they are separate modules. The Secure Storage Limit register can be located either within the core module or within the Secure Storage module. The important thing, of course, is that the register and the write-protection circuitry must be located "on the side of the Secure Storage", by which we mean there must be no pathways to write the Secure Storage that do not first go through the limit register and the write-protection mechanism.

As an example of a potential vulnerability, imagine that the DMA controller or some other device that updates memory is able to write to the Secure Storage area without first going through the write-protection circuitry. This would provide a way to circumvent this critical security mechanism.

Memory-mapped I/O devices are not involved in the cache system. Caching occurs only for physical main memory, which lies below address 0x4_0000_0000. While it might be tempting to allow the Secure Storage memory to participate in caching, this is disallowed, since it might introduce subtle security vulnerabilities.[120]

Verifying the Kernel Code

The boot process will load a kernel image into main memory. Before branching to it, the boot process must verify that the image it has just loaded is the real, correct image. We must ensure that the kernel image has not been corrupted or altered by malicious software.

[120] Although we cannot see how this could happen, perhaps the cache contents could become outdated, allowing the core to fetch incorrect data from a location in the Secure Storage area that is assumed to have been updated, locked, and guaranteed to be correct. Or perhaps a delayed write-back/write-through from the cache to the Secure Storage area could delay the updating of the Secure Storage Limit register , thereby leaving the Secure Storage area vulnerable to malicious updates for a short window of time.

In order to achieve this, the BootLoader will compute a secure message digest and compare it with a known, expected value.[121]

Secure Message Digests

A **secure message digest** (also called a **secure hash**) is a short, fixed size binary value which is computed from all the bytes in a much longer string of bytes. There are a number of different secure hash algorithms. For example, with the SHA-256 algorithm, the message digest is a 256 bit value.

The secure hash algorithm is designed in such a way that any change in the long byte sequence will alter the digest value (with extremely high probability). Furthermore, given a particular digest value, it is extremely difficult to create a string that will hash to that digest value.

An example usage would be to make sure a kernel executable image has not been modified by a malicious actor or cyberattack. If the kernel image is scanned and a digest value is computed that matches a stored "expected value", then (with extremely high probability) this kernel image must be exactly the one and only same byte string that was used to produce the expected value in the first place. The secure hash system allows us to be sure the kernel has not been modified.

In Blitz-64, we prefer and recommend the **SHA-256 secure message digest algorithm**.

The BootLoader can easily compute a secure hash of any potential kernel image it has loaded, but the question is: Where does the BootLoader get the "expected value" with which the computed value must be compared?

In other words, the BootLoader needs a secure, non-volatile storage in which to store expected hash values. Moreover, to prevent malware from changing the expected value and then substituting a modified kernel, the expected value must be stored in a place that cannot be modified by any software other than the BootLoader.

[121] In the case of a single-stage boot chain, the code that loads the kernel executable will be resident in the Boot ROM Area. In the case of a multi-stage process, the code will be in firmware, i.e., in the Secure Storage area. For this discussion, it doesn't matter and we will just talk about the BootLoader program, regardless of whether it is in Boot ROM or is Second Stage Boot Loader (SSBL) code.

This place is the Secure Storage area.

The Secure Storage area need only be large enough (e.g., 32 bytes in the case of SHA-256) to store a single secure message digest value, although the Secure Storage area is expected to be much larger. To accommodate the loading of several different versions of the kernel, several secure message digest values would need to be stored. The idea is that the BootLoader would accept any kernel executable if its secure message digest matches any of the stored values.

The expected secure message digest of the kernel will be placed in the Secure Storage area. The BootLoader, as part of its functionality, must always lock the Secure Storage before branching to any non-BootLoader code, so that no other software can possibly modify the expected message digest values stored in the Secure Storage area.

On typical power-on-resets and soft-resets, the BootLoader will simply compute the message digest for the kernel executable, retrieve the expected message digest from Secure Storage, and compare them to verify that the kernel being loaded has not been corrupted.

Normally, the version of the kernel to be loaded will be the same version as last time, so there is rarely a need to store a new expected value. But occasionally the user will need to install a new version of the kernel. In that case, the computed secure digest will not match the stored expected value. We need a way to update the stored expected value.

In one approach, the BootLoader might require the user to manually type in the expected secure hash value. The BootLoader will then store the new value in the non-volatile Secure Storage memory area before it locks it. With every new version of the kernel, the user must type in a secure hash value to validate the kernel version.

Of course this secure hash value for the kernel is a cryptographic key which must be securely validated and protected from alteration or spoofing, to prevent the user from seeing a false key.[122]

[122] If this is too onerous, the BootLoader might simply alert the user that the kernel image has changed and ask the user whether this is intended. If the user agrees, the hash value just computed for the new version will be written to Secure Storage and saved as a new “expected value”.

In order for the BootLoader to be able to store a new key, a hard restart is required, i.e., a power-on-reset signal must be generated. This will always and definitely cause the BootLoader code from the Boot ROM Area to be executed, with no intervening software possible.

However, requiring intervention by users when it comes to verification of a new kernel's message digest is risky. Instead, we need a way to completely automate the updating of kernel versions.

Secure Distribution of New Message Digests

Presumably new versions of the kernel software are distributed by a single trusted source and the goal is to prevent any bad actor from impersonating the trusted source. We must make sure the BootLoader never, ever boots to a compromised version of the kernel.

Here we describe an approach to distributing secure hash keys (i.e., secure message digests) using a **public-private key encrypted communication** channel.

In the public-private encryption technique, there are two keys. One key (the **private key**) is used to encrypt the message and the other key (the **public key**) can be used to decrypt the message.

The BootLoader software will have the public key hardcoded directly into it. The public key is not a secret. The private key will be kept remotely, by the organization authorized update the kernel. For example, the private key will be held by the trusted company that creates and distributes new, authorized versions of the kernel.

In the public-private key system, the communication is both kept private and protected from corruption.[123] We must protect against spoofing: we must be certain that the message came from the sender it claims to have come from. Public-private key systems do this, since they guarantee the message has not be altered and that it comes only from the organization holding the other (private) key.

From time to time the trusted authority will communicate with the existing kernel instructing is to install a new version of the kernel. The existing, old version of the kernel will download the new executable file and store it on the boot device. This

[123] In this case, we care only about protection from corruption; privacy is not required.

communication will also contain a special message to be delivered to the BootLoader.

The message to the BootLoader will command it to boot to a new version of the kernel image. The message will consist of two items: (1) the name/filename/version number of the new kernel, and (2) the corresponding secure message digest for that version. The sole purpose of the message is to instruct the BootLoader to update the Secure Storage area to save a new secure hash key for the new version of the kernel code.

The message to the BootLoader will be encrypted using the private key. Only the authorized and trusted organization can create a valid encrypted message in this way. The message will be decrypted by the BootLoader using the public key.[124]

The BootLoader must receive the message and process it after a power-on-reset, since only at that time will the Secure Storage area be unlocked. Thus, the BootLoader will look for an incoming message every time it runs after a power-on-reset. If an incoming message is found, it will be processed.

The message can be communicated to the BootLoader in several ways. One approach is to place the message in a file with a fixed, well-known name, such as "BootLoaderNewKey". Upon power-on-reset, the BootLoader will read from this file. This might be appropriate for a single-stage boot sequence, since the BootLoader code — which is in the Boot ROM Area — will already be capable of understanding the file system and the device on which it is stored, since it will be capable of reading the kernel executable file.

Another approach is for the message to be stored directly in the Secure Storage area. Of course, it can just be stored directly above the current value of the Secure Storage Limit Register. Recall, that writing is always allowed to memory addresses above the limit, but the memory below the limit is in the locked state and cannot be altered, even by the kernel.

[124] In order to reduce the likelihood of the private key being discovered by a bad actor who is viewing the message traffic, the quantity of data encrypted by the private key should always be kept to a minimum. The private key held by the trusted organization for the purpose of validating new kernel versions should only be used for this purpose and the encoded data should be kept as concise and non-redundant at possible, with all repeated, formulaic, or boilerplate information eliminated.

The process of updating a kernel involves (1) downloading the new kernel executable file and the associated encrypted message for the BootLoader; (2) placing the message in a place where the BootLoader will find it; and (3) performing a power-on-reset by executing a RESTART command.

The BootLoader does the rest.

The result is that — without obtaining the private key of the organization trusted with distributing new kernel images — it is impossible to boot into a corrupted or compromised kernel. Only "official" kernels will boot.[125]

[125] For added security, perhaps the Low Level Boot Loader (LLBL) code running in the Boot ROM Area should begin by immediately verifying that the Secure Storage Limit Register is zero (as expected) to verify that a power-on-reset has truly just occurred. It is perhaps conceivable that if not, malicious code running simultaneously might be able to interfere somehow.

Appendix 1: Blitz Assembly Language

Assembling and Linking

This appendix contains a brief introduction to **assembly language** concepts, as applied to Blitz-64.

Assembly language is a sort of primitive programming language, in which the programmer writes instructions that can be directly executed by the processor core. Each computer architecture has its own assembly language. Programming convenience, portability, and maintainability are crucially important. Although all these are absent with assembly language, assembly language programming necessary for anyone close to the hardware.

This appendix discusses one assembler tool in particular; the Blitz-64 assembler. This tool exists in two identical versions. One version is written in C (and runs on a POSIX-based host) and the other version is written in KPL (and runs on a Blitz-64 computer).[126]

The basic idea is that each machine instruction can be written symbolically instead of written in binary. An assembler tool translates the symbolic assembly code into binary machine code. For example, the assembler translates an **assembler instruction** such as:

```
addi     r2,r4,100          # end = start + size
```

into the following 32 bit **machine instruction**:

```
0x01006424
```

[126] There may be other assemblers; for details, consult the documentation for the assembler tool you are using.

An assembly program is a text file with one instruction per line, making it possible to write machine code in a human-readable, symbolic form. Machine code specified in binary or hex is just too error-prone for humans to create, and an assembly language is an improvement over pure machine code.

In addition to the **assembler** tool, the **linker** tool must be used, in a step called "linking".

Roughly speaking, the purpose of the assembler is to:

- Check the assembly source file for errors, to make sure all the instruction names are spelled correctly.
- Determine whether the required operands are present and correctly specified.
- Compose the machine instructions, at least in most cases.

And roughly speaking, the purpose of the linker is to:

- Determine where in memory to place the machine instructions and data.
- Evaluate expressions that depend on memory locations.
- Determine which machine instructions will be used, in cases where the assembler can not do it.

The assembler translates each assembler source file into an "**object file**". One or more object files are then combined by the linker to produce an "**executable file**". Often the executable file is called the "**a.out**" file, since that is the name commonly given to the executable file. At **runtime**, the OS kernel loads the executable file into memory and begins execution.

Assembler Syntax

Each line of the assembly program contains a single instruction. Each line contains the following fields:

Label — optional
Opcode
Operands — zero or more
Comment — optional

The syntax for a line is this (where brackets indicate optional material):

[*label* **:**]   [*opcode* [*operands*]]   [**#** *comment*]

For example:

```
myLabel:  xor   r2,r4,r6      # An example instruction
```

A label consists of an identifier symbol, followed by a colon (":"). The label may be on a line alone, or it may prefix an instruction or pseudo-op. The label associates the symbol with the address of whatever follows.

Label symbols are user-defined identifiers and may not have the same spelling as instructions, register names, or pseudo-ops. Symbols may contain the underscore character, e.g., "MyLab_43" and "_entry". A leading underscore is only meaningful by convention; the assembler doesn't care whether identifiers begin with underscore. Identifiers may not begin with a digit, but they may contain digits.

The register names, the CSR names, and the opcodes are all in lowercase. For registers with two names (e.g., "sp" = "r15"), either name may be used.

Values specified in decimal are written as a sequence of digits, e.g., "1234". Values coded in hex are written with the prefix "0x", e.g., "0x1234". Floating point constants (e.g., "0.5", "123e-9") can be used, but only in the ".float" pseudo-op.

Comments begin with the "#" character and run through the end of the line.

Tabs are typically used between labels, opcodes, operands, and comments, but spaces may also be used.

```
     t    t    t    t    t    t    t    t    t    t
# Here is an example:
     addi       r2,r4,0x3B7F          # Add decimal 15,231
     csrread    r3,csr_status
MyLab_43:
     load.b     r5,123(sp)
     ble        r5,r2,Exit_Label
```

The names of the machine instructions have been given earlier in this document and the order and meaning of the operands have been specified, so they will not be repeated here.

Assembly code is case sensitive.

Whenever the operand is an immediate value (e.g., “immed16” or “address” in the earlier chapters), the programmer may specify a value in hex or decimal, a symbol, or (more generally) an expression using constants, symbols, and the usual operators, such as +, -, <<, &, … For example the following instruction:

```
        ADDI        RegD,Reg1,immed16
```

might be used like this:

```
        addi        r7,sp,MyLabel+(3*len)
```

Pseudo-Ops

In addition to lines containing machine instructions, the assembly code file will contain lines containing pseudo-ops. A “**pseudo-op**” is an **assembler directive** which gives guidance to the assembler/linker about how to assemble instructions.

While a line containing a pseudo-op looks like a machine instruction, it is not. To emphasize the distinction, all pseudo-ops begin with a period.

Here are the pseudo-ops:

`.byte`	*<integer expr>*	Place a byte in memory
`.halfword`	*<integer expr>*	Place a halfword in memory
`.word`	*<integer expr>*	Place a word in memory
`.doubleword`	*<integer expr>*	Place a doubleword in memory
`.float`	*<floating value>*	Place a 64 bit floating point in memory
`.string`	*<string>*	Place a sequence of bytes in memory
`.skip`	*<integer expr>*	Skip N bytes, filling with zeros
`.align`	*2/4/8/16/32/page*	Insert 0x00 bytes to achieve alignment
`.equ`	*<integer expr>*	Equate symbol to an integer value
`.export`	*<symbol>*	Make this symbol available to other files
`.import`	*<symbol>*	Expect symbol to be defined in other file
`.begin`	*<parameters>*	Start filling a new chunk of memory

Each pseudo-op is written on a line by itself, in the same format as a machine instruction. Here are some examples:

```
x:    .byte        123              # Byte containing value
c:    .halfword    0x04d2           # Decimal: 1234
d:    .word        0x000BC614E      # Decimal: 12345678
e:    .doubleword  0x12-100         # 0xffff_ffff_ffff_ffae
f:    .float       -123.456e-10     # Double precision
str:  .string      “Hello\n”        # No terminating \0
arr:  .skip        400              # Array of 400 bytes
      .align       8                # Insert padding bytes
```

The **.byte**, **.halfword**, **.word**, and **.doubleword** pseudo-ops are used to allocate 1, 2, 4, and 8 bytes (respectively). The initial value to be placed in the memory (before execution begins) is given by an expression, which may include values given in decimal or hex. The expression appearing in the operand field may also employ the usual operators. The expression will be evaluated and the value will be computed at “assembly time” (i.e., by the assembler and linker) and not at “run-time”.

If a label precedes a pseudo-op or instruction, that symbol will be associated with the address of the thing that follows. (More precisely, the symbol will be associated with the address of the first byte of the thing that follows.) The label may appear on the same line or on the preceding line.

For example, this

```
  myVar:   .doubleword      0x0123456789abcdef
```

is equivalent to this:

```
  myVar:
           .doubleword      0x0123456789abcdef
```

The **.float** pseudo-op is used to allocate 8 bytes and fill it with the IEEE representation of a double-precision floating point number. The operand should be a floating point constant. Expressions are not supported.

The **.string** pseudo-op is used to place ASCII data in memory. The usual escapes (\n, \0, \t, etc.) can be used, as well as specific hex codes. For example, the byte 0x3f is written as \x3f, where "x" means "hex". The string is not null-terminated, but the null character can be included in two ways, e.g.,

```
  str:  .string          “Bye\0”
```

and:

```
  str:  .string          “Bye”
        .byte            0
```

The **.skip** pseudo-op causes the assembler to skip over a number of bytes, without filling these bytes in with initial values. The bytes are guaranteed to be filled with zeros before execution begins. If a label precedes the .skip pseudo-op, then that symbol is associated with the address of the first byte in the block of bytes allocated by the .skip pseudo-op.

The **.align** pseudo-op is used to insert padding bytes to force the next following thing to be aligned. In the following example, the string may end on an improperly aligned address; the .align pseudo-op will insert as many bytes as necessary to guarantee that the variable "x" is properly aligned.

```
  str:  .string          “hello”
        .align           8
  x:    .doubleword      0x0123456789abcdef
```

The padding bytes inserted by .align are guaranteed to be zero-filled. The operand for .align may be 2, 4, 8, 16, or 32. In addition, the keyword “page” may be used as the operand. Including “.align page" will add padding bytes as necessary to round up to the next page aligned address, i.e., to an address that is a multiple of 16,384 (i.e., a multiple of 16 KiBytes and in which the least significant 14 bits are zeros).

Symbols

The **.equ** pseudo-op should always be preceded by a label. The purpose of .equ is to define a symbol and give it a specific value. The value is given by an expression, which is evaluated at the time of assembly and linking, not at runtime. For example:

```
start:   .string      “hello”
end:
len:     .equ         end-start
```

A symbol is defined by its appearance as a label on some line in an assembly source file. Symbols may be used before they are defined. In other words, the line defining a symbol may appear later in the assembly source file than a line in which the symbol is used as an operand.

A symbol may also be defined in one file and used in another file, although the .export and .import pseudo-ops must be used. As a result, the actual value of a symbol may not be known by the assembler. Therefore, some expressions cannot be evaluated until the linker tool is executed.

The **.export** pseudo-op is used to make a symbol defined in this file available for use in other assembly source files. Symbols are, by default, local to the current assembly source file and must be exported if they are to be used in other files. The operand should be a single symbol. For example:

```
myVar:   .doubleword  1234
         .export      myVar
myConst: .equ         100
         .export      myConst
```

The **.import** pseudo-op is used to make a symbol that is defined in another file available for use in this file. For example:

```
.import      myVar
loadd        r3,myVar
.import      myConst
addi         r3,r3,myConst
```

A symbol must either be defined or imported (but not both). If a symbol is neither defined nor imported, the assembler will flag it as an error. Every symbol that is imported in one file must be exported in exactly one other file; if not, the linker will issue an error message.

Every symbol either has an "**absolute value**" or a "**relative value**". For example, "myConst" in the above example has the absolute value of 100. An absolute value is not dependent on where in memory the linker places things.

A relative symbol is a memory address and is dependent on where the linker places code and data. In the example above, "myVar" is a relative symbol. The values of relative symbols are not computed until the linker assigns memory locations to code and data.

For some instructions, the actual binary machine code cannot be determined by the assembler. This will happen whenever the instruction contains an immediate value for which the programmer has provided an expression containing a relative symbol. Since the value of the symbol cannot be known until link-time, only the linker has enough information to complete the assembly of the instruction.

For instructions using absolute symbols, the assembler will be able to complete the assembly of instructions whenever the symbol is used in the same file in which it was defined. However, when the symbol is defined in one file and used in another file, the linker will be required to fill in the values and complete the instructions.

In the case of synthetic instructions, the assembler will sometimes be able to choose the final machine code and complete the assembly. But in other cases, the synthetic instruction may translate into one, two, three, or even four machine instructions, depending on the actual value of the operand. Since the value of the operand may not be known until link time, it will be up to the linker to determine which sequence of machine instructions will be used to implement a given synthetic instruction.

[The assembler and linker work together to produce the final code. In cases where a synthetic instruction may be turned into several instructions and the assembler must pass the problem to the linker, the assembler will make the initial assumption that a single machine instruction will suffice and will allocate a slot of size 4 bytes. Once the final value of all symbols is known, the linker will determine whether one instruction (i.e., 4 bytes) turns out to be adequate. If 4 bytes as inadequate, the linker will expand the slot (and the segment containing the slot) by another 4 bytes to accommodate a second machine instruction. Once again, the linker will determine whether there is enough room. This process will be repeated, enlarging all synthetic instruction slots until each is large enough to contain the machine instructions needed to handle the value. This is an example of a "relaxation" algorithm. Since slots are only enlarged and never reduced in size, this process will eventually terminate. In the worst case — highly improbable — each slot will be enlarged to its maximum size, which is enough to accommodate any possible value. In the vast majority of cases, the slot size will be just large enough to accommodate the smallest synthetic instruction sequence, and no larger.]

Segments and Linking

The linker will place code and data into pages of memory. Each page of virtual address space will be marked either executable or not, and each page will be marked either writable or not. All pages are readable, so this is not an issue. This was described in an earlier section when virtual memory and page tables were discussed.

Each assembly code source file consists of a sequence of "**segments**". Each segment consists of a sequence of instructions. The segments are listed one-after-the-other in the source code file. Thus, every line in the source file will belong to exactly one segment.[127]

An assembly source file will typically contain only one segment, or just a couple of segments. For example a given assembly source file may contain one segment of instructions (which will go into pages marked "executable" but not "writable") and one segment of data (which will go into pages marked "writable" but not "executable").

[127] The term "section" is sometimes used instead of "segment".

The term "segment", as used here, is a purely software concept used only by the assembler and linker; at runtime there is no such thing as a segment. (Other computer systems have used the term "segment" differently, e.g., for regions of memory supported by various hardware features.)

The purpose of the "**.begin**" pseudo-op is delineate segments.

Below is a small, artificial example, representing a single assembly source code file containing three segments:

```
        .begin          executable
entry:  loadd           r1,myVar
        addi            r1,r1,300
        stored          myVar,r1
        ret

        .begin          writable
myVar:  .doubleword     12345
other:  .doubleword     200

        .begin
str:    .string         "Hello"
        .byte           0
        xor             r1,r2,r3
```

Each segment must start with a .begin pseudo-op. A segment runs from a .begin pseudo-op until just before the next .begin pseudo-op, or until the end-of-file. Every instruction and every other pseudo-op will be located in exactly one segment, based on where it is placed.

There is no requirement that an "executable" segment contains only machine instructions; it may contain data as well. There is no requirement that a "writable" segment contains only data; it may contain machine instructions as well.

In this example, the third segment is marked with neither executable nor writable. It contains a string and an XOR instruction. This segment is read-only (i.e., not writable and not executable) so the XOR instruction cannot be executed.

Segments are not given names and the line containing .begin must not contain a label. Any label directly preceding a .begin pseudo-op will be associated with an address in the previous segment.

The .begin pseudo-op has an operand field that can contain a number of comma-separated parameters.

```
    .begin     parameter , parameter , parameter , parameter
```

For example:

```
    .begin     startaddr=0x8000a0000,executable,writable
```

The following parameters are indicated by a keyword, which is either present or absent.

```
    kernel
    executable
    writable
    zerofilled
```

The programmer may also include a “startaddr=” parameter:

```
    startaddr=integer
```

The programmer may also include a “gp=” parameter:

```
    gp=integer
```

The job of the linker is to determine where in memory to place the segments. More specifically, the input to the linker will be a number of object files, each containing a number of segments.

For programs that will go into a virtual address space, these segments will ultimately be placed into memory pages. One constraint is that two segments with different executable/writable attributes may not be placed in the same page. Another constraint is that segments may not overlap. The linker will attempt to group similar segments together and pack them as close as possible in order to reduce the number of pages in the final memory image.

Normally, the linker will be free to choose the location of a segment. However, the programmer may demand that the linker place a segment at a given memory address. This is the purpose of the "startaddr=" parameter, which gives the starting address of the segment as an absolute value. This parameter forces the linker to place a segment at a particular location in memory.

If there is no starting address given for a segment, the linker is free to place the segment where it best fits. By default, the linker will place segments in the virtual address region, which starts at 0x8_0000_0000. The linker will more-or-less place segments one after another, filling up the virtual address space from 0x8_0000_0000 on up, within the previously mentioned constraints.

However, the presence of the "kernel" keyword will force the linker to place the segment in the lower, physical region of address space. Segments with this keyword will be placed in low memory, starting with 0x0_0000_0000 and going up.

The "zerofilled" keyword is used to indicate that a segment will contain only zeros. Thus, only the following are allowable in a "zerofilled" segment:

```
        .byte         0
        .halfword     0
        .word         0
        .doubleword   0
        .float        0.0
        .skip         <any>
        .align        <any>
        .equ          <any>
        .import       <any>
        .export       <any>
```

The data in zerofilled segments is not present in the object and executable files, since the pages can be created and initialized at the time the executable file is loaded into memory. Zerofilled segments are useful for large data structures (such as gigantic arrays, spaces for heaps, and so on), since these data structures would waste a large amount of space in the object and executable files if all bytes were actually present. For example:

```
         .begin    startaddr=0x9_0000_0000,writable,zerofilled
MyHeap:  .skip     0x1_0000_0000      # 4 GiBytes
```

The assembler will round each segment up in size to a multiple of 8 bytes, by adding 0 to 7 bytes of 0x00, as necessary. The linker will place each segment on an aligned 8 byte address.

The Global Pointer Register, gp

Several of the synthetic instructions specify that an operand can be an "address". Examples include:

BEQ	Reg1,Reg2,address
LOADB	Reg1,address
CALL	address

In the course of generating code, the assembler and linker must be able to translate memory addresses into the forms required by the machine instructions. For example, consider this line from an assembly source file:

```
        loadb    r1,MyVar
```

Assuming the address of MyVar is within 0 … 0x0_0000_7fff, the above instruction can be assembled like this:

```
        load.b   r1,0x7fff(r0)
```

For user programs running in a virtual address space, the assumption is that the global pointer register (gp) will contain the value 0x8_0000_8000 at runtime, and this register can make addressing certain locations in memory particularly easy.

(The gp register will be initialized either by the kernel during thread-creation or within the first couple of instructions at thread-startup, as part of the thread initialization prologue. If initialized within the thread prologue, the MOVI instruction is safe to use for this purpose although it is synthetic. The assembler may use gp whenever it synthesizes a MOVI instruction and the value in question is within range, allowing the MOVI to be translated into a single ADDI instruction. However, the assembler will specifically avoid using gp whenever the destination register is gp itself.)

When generating code, the assembler/linker will make use of the assumed value of gp. For example, if MyVar is located within the first 4 pages of the virtual address space (i.e., the first 64 KiBytes of the virtual address space, 0x8_0000_0000 … 0x8_000_FFFF), then the assembler/linker can generate an instruction which uses an offset from register gp. For example, if MyVar is located at 0x8_0000_8056, it can be assembled like:

```
        load.b   r1,0x0056(gp)
```

Positive offsets will be used for addresses above 0x8_0000_8000 and negative offsets will be used for addresses below that:

`8_0000_0000 … 8_0000_7fff`	negative offset 8000 … ffff from gp
`8_0000_8000 … 8_0000_ffff`	positive offset 0 … 7fff from gp[128]

The assembler/linker can deal with arbitrary addresses but addresses outside this range might require additional instructions or the use of the temp register "t". Therefore, the programmer is encouraged to place commonly used variables at the bottom of the virtual address space, in the first 64 KiBytes. The typical practice would be to place all static, non-stack data at the bottom of the virtual address space, with the code segments in pages following the data pages.

The above comments about register gp primarily concern LOAD and STORE instructions which are used to access data in static, fixed memory locations. Other instructions (e.g., JUMP, BRANCH, CALL) are using addresses as jump targets. For them, PC-relative addressing is more common and useful. However, the gp-relative addressing mechanism is still present and gp-relative jumps can be generated whenever the target address is in low memory. For example, it might make sense to place jump tables in low-memory, so the code can easily branch to various entries.

Kernel code will not be running in a virtual address space, so things are different. All addresses will be located in the physical memory region.

For kernel code, the "gp" register is assumed to be initialized to 0x0_0001_0000 (i.e., 64 KiByte).

This means that any address in the first 6 pages (i.e., the first 96 KiBytes of memory, 0 … 0x0_0001_7fff) can be accessed with a single instruction.

[128] More precisely, non-negative offsets. For address 0x8_0000_8000 an offset of 0 is used.

Addresses within the first 32 KiBytes (0x0 ... 0x7FFF) are easily accessible using offsets from “r0”. The next 64 KiBytes (0x8000 ... 0x1_7FFF) can easily be accessed from register “gp”.

`0_0000_0000 … 0_0000_7fff`	offset 0…7fff from “r0”
`0_0000_8000 … 0_0000_ffff`	negative offset 8000…ffff from “gp”
`0_0001_0000 … 0_0001_7fff`	positive offset 0…7fff from “gp”

Commentary It is recommended that the kernel place all frequently accessed, global, static data in low memory.

If “gp” has been properly initialized, bytes within the first 6 pages (96 KiBytes) are addressable with a single LOAD or STORE instruction, since they can be addressed with a 16 bit immediate offset from register r0 or gp. The need to use two instructions is avoided for the most frequently accessed kernel variables.

Thus, the most critical data should be placed within the first 6 pages (96 KiBytes). The data region can then be followed by the code at successively larger memory addresses. Placing the code after the data (rather than before the data) means that accesses to the most frequently accessed data can be done with a single instruction.

If the “kernel” keyword is present in the .begin pseudo-op, the default assumption made by the assembler and linker is that register gp will contain the value 0x0_0001_0000. If the “kernel” keyword is not present, the assumption is that gp contains 0x8_0000_8000.

The programmer can override the default assumption with the “gp=” parameter.

The “value” associated with a “startaddr=” or “gp=” parameter must be an absolute value that can be calculated immediately by the assembler. Normally, every “startaddr=” or “gp=” value will be a simple hex constant.

User mode code should never be accessing any address below 0x8_0000_0000 and the assembler/linker may issue warnings for any LOAD, STORE, BRANCH, JUMP, or CALL instruction that uses such an address in a code segment that is not marked “kernel”.

The programmer can also specify “gp=undefined” in the .begin pseudo-op, which will entirely prevent the assembler/linker from using register “gp” in any synthesized instructions. This would be useful for code in which the gp register (i.e., r13) is used for an entirely different purpose.

Appendix 2: Implementation Details

Every implementation of the Blitz-64 architecture must provide documentation to elaborate on ISA details that are "implementation dependent" or "undefined" in this document.

Such an implementation document must answer the following questions.

- What values are used for **csr_version** and **csr_prod**? What values are obtained when these registers are read?

- How many cores are implemented and what is their arrangement? Is the organization 1-D, 2-D, or 3-D? What are the dimensions[129] of the array of cores?

- Which machine instructions are unimplemented and require emulation?
  DIV, REM
  Floating point instructions

- If the DIV and REM instructions are implemented, do they perform "truncated", "floored", or "Euclidean" division when the operands are negative?

- How are the CONTROL and CONTROLU instructions defined? (Perhaps they are unused and always causes an Illegal Instruction Exception.)

- Concerning the ENTERFUN instruction, if an Arithmetic Exception occurs for the addition but there is no exception for the store operation, will the store to memory be executed? If there is an exception related to the store operation but the addition does not overflow, will the update to the destination register occur? If there is an Arithmetic Exception and an exception related to the store operation, which exception will be signaled?

[129] That is, what are M, N, and P in the array addresses [0,0,0] … [M-1,N-1,P-1]?

- Concerning the EXITFUN instruction, if an Arithmetic Exception occurs for the addition but there is no exception for the load operation, will the program counter be updated, that is, which value will be saved in **csr_prevpc** ? If there is an exception related to the load operation but the addition does not overflow, will the update to the destination register occur? If there is an Arithmetic Exception and an exception related to the load operation, which exception will be signaled?

- Are there any additional instructions or changes to the Blitz-64 ISA?

- Does the core contain TLB registers? How many?

- Concerning memory…
  How much private memory is available to each core?
  Is shared memory is present? How much? Starting address?

- What memory caching is implemented?
  What are the details?
  Is the cache write-through or not?

- Which memory-mapped I/O devices are implemented?

- Concerning each memory-mapped I/O device…
  What is its starting address?
  How many pages does it occupy?
  Exactly how does it function?

- Concerning asynchronous interrupts…
  What are the possible interrupt types?
  What causes each interrupt to occur?
  What value is stored in **csr_cause** for each type?

- Concerning the Boot ROM area, what does it contain? In particular, what is the assembly source file that produced it, showing all the bytes?

- Are there any other changes to the Blitz-64 specification?

Example: The Emulator

What values are used for **csr_version** and **csr_prod**? What values are obtained when these registers are read?

The emulator is configurable; the values can be set as part of the "emulation parameters". The default for **csr_version** is 0x0002_49F0_8002_0001, i.e., conforms to spec = 1; version number = 0x0002; implementor = 0x0001 ("Harry Porter"); and a value of 0x0002_49F0 (decimal 150,000) for clock cycles per millisecond, indicating 150MHz operation. The default for **csr_prod** is 0x0000_0000_0000_0000.

How many cores are implemented and what is their arrangement? Is the organization 1-D, 2-D, or 3-D? What are the dimensions of the array of cores?

The emulator is configurable; the number of cores and their arrangement are included in the "emulation parameters". The default is 6 core, but by default, only core 0 is "running", while the others are "stopped".

Which machine instructions are unimplemented and require emulation?

DIV, REM
Floating point instructions

The DIV, and REM instructions are implemented, but the "-nodiv" command line option can be used to force an Emulated Instruction Exception. The floating point instructions can either cause an Emulated Instruction Exception, or will be executed directly. This is configurable with the "-fp" command line option.

If the DIV and REM instructions are implemented, do they perform "truncated", "floored", or "Euclidean" division when the operands are negative?

The operations of "truncated" and "Euclidean" division only differ when the top value (the dividend) is negative and the remainder is non-zero. Whenever an operation is attempted where this condition holds, the emulator will signal a user error. Thus, the results of the emulator are consistent with both "truncated" and "Euclidean" division, and signals an error if ever the user attempts an operation where the results would differ.

How are the CONTROL and CONTROLU instructions defined? (Perhaps they are unused and always causes an Illegal Instruction Exception.)

The following operations are implemented for the CONTROLU instruction, using the corresponding values in the immediate field.

DIGITAL_READ	0
DIGITAL_WRITE	1
HALT	2
SERIAL_STAT	3
SERIAL_RECV	4
SERIAL_SEND	5
ENABLE_KERNEL	6
SET_STATUS	7
TLB_DEBUG	8

These codes and operations are designed to match the codes used in the MicroBlitz hardware implementation. These operations are useful in testing and exercising the hardware implementation and are provided in the emulator so that the emulator can be used to execute code meant to be run on the hardware. Such code is meant to test and verify the correctness of the hardware.

The DIGITAL_READ operation is not implemented. DIGITAL_WRITE will print the value in register Reg1 in hex. HALT will halt execution. SERIAL_STAT will move 0x0000_0000_0000_0003 into RegD. This is a value that indicates that the serial device is ready to receive and ready to transmit. SERIAL_RECV will get a byte from the user and store it in RegD. SERIAL_SEND will print the byte in Reg1 on the output. ENABLE_KERNEL will move the current value of **csr_status** to RegD and will then set **csr_status** to 0x0000_0000_0000_0001. SET_STATUS will set **csr_status** to 0x0000_0000_0000_0001. TLB_DEBUG will retrieve the number of a TLB register from Reg1. It will move a representation of that TLB register into RegD. The TLB registers are larger than 64 bits, so this operation will store the following in RegD:

ASID[63:48] || VirtAddr[47:27] || PhysAddr[26:5] || C || D || W || X || V

This captures all the information except the upper bits of the physical address, which are lost.

The behavior of all other CONTROL and CONTROLU operations is left to the user. When encountered, the emulator will halt and display the immed16 value and the contents of register Reg1. Then, the emulator will ask the user whether or not the instruction should cause an Illegal Instruction Exception. If "no", then the emulator will prompt for a value to be entered, which is placed in register RegD. Then, execution is resumed.

Concerning the ENTERFUN instruction, if an Arithmetic Exception occurs for the addition but there is no exception for the store operation, will the store to memory be executed? If there is an exception related to the store operation but the addition does not overflow, will the update to the destination register occur? If there is an Arithmetic Exception and an exception related to the store operation, which exception will be signaled?

If an Arithmetic Exception occurs, it will take precedence; the store operation will not be attempted and the destination register will not be modified. If the addition is okay, but there is an exception related to the store operation, then neither the update to the destination register nor the store operation will be performed.

Concerning the EXITFUN instruction, if an Arithmetic Exception occurs for the addition but there is no exception for the load operation, will the program counter be updated, that is, which value will be saved in **csr_prevpc** ? If there is an exception related to the load operation but the addition does not overflow, will the update to the destination register occur? If there is an Arithmetic Exception and an exception related to the load operation, which exception will be signaled?

If an Arithmetic Exception occurs for the addition, it will take precedence; the load operation will not be attempted. If the addition is okay, then the destination register will be updated, regardless of whether or not there is an exception related to the load operation. If there is an exception (regardless of whether it is from the addition or the load operation), **csr_prevpc** will be set to the address of the EXITFUN instruction.

Are there any additional instructions or changes to the Blitz-64 ISA?

No.

Does the core contain TLB registers? How many?

The emulator is configurable; this is one of the “emulation parameters”. The default number is 16. If desired, the value 0 can be used to run with no TLB registers.

Concerning memory…

How much private memory is available to each core?

1 GiByte.

Is shared memory is present? How much? Starting address?

Shared memory is 1 GiByte, starting at address 0x0000_0000_4000_0000.

These values are the defaults; the emulator is configurable.

What memory caching is implemented?

What are the details?

Is the cache write-through or not?

The emulator does not implement memory caching.

Which memory-mapped I/O devices are implemented?

Boot ROM	
Secure Storage	
DMA Controller	Memory moves, SHA-256, AES-256
Host Device	For passing commands through to the host OS)
UART	Serial communication
DISK	Non-volatile storage device
PLIC	Platform-Level Interrupt Controller
Digital I/O Pins	

There is only one “Secure Storage” device, which is shared between all cores. Ideally, each core would have its own Secure Storage device.

Concerning each memory-mapped I/O device…

What is its starting address?

How many pages does it occupy?

Exactly how does it function?

Details are given in the document “Blitz-64: Memory-Mapped I/O Devices”.

Concerning asynchronous interrupts…

What are the possible interrupt types?
What causes each interrupt to occur?
What value is stored in **csr_cause** for each type?

The following devices cause interrupts:

Timer Interrupt
DMA Controller
Platform-Level Interrupt Controller (PLIC)

Details are specified elsewhere in this document and in “Blitz-64: Memory-Mapped I/O Devices”.

Concerning the Boot ROM area, what does it contain? In particular, what is the assembly source file that produced it, showing all the bytes?

The value stored in the Boot ROM is configurable. It is read in by the emulator from a secondary file named “emulationROM”. A simple version of the BootLoader comes from **boot0.s**; it assumes the emulator has already read in an executable file into memory and simply jumps to it.

Are there any other changes to the Blitz-64 specification?

No.

Appendix 3: Recent Changes

This appendix documents changes to the Blitz-64 architecture and this document over time.

28 May 2019

A new exception called "Null Address Exception" is added.

The exceptions numbers are changed and shifted to make room for the new Null Address Exception.

The relevant instructions were altered. They will now signal this exception when appropriate.

15 June 2019

Modification to instructions SLL, SLA, SRL, SRA. These instructions will now cause an Arithmetic Exception if the value in the register (i.e., the shift amount) is not within 0 … 63.

15 June 2019

The CHECKA instruction is added.

A new exception called "Bad Array Index Exception" is added.

The exceptions numbers are changed and shifted to make room for the new Bad Array Index Exception.

23 July 2019

In the section describing the Arithmetic Exception, there was an error in the list of which instructions can cause the exception. It is changed as follows:

Arithmetic Exception

This exception can be caused by the following operations:

Integer arithmetic :	ADD, ADDI, SUB, MUL, DIV, REM
Shift operations:	SLA, SLAI, SRA, ~~SRAI~~, SRL, SLL
Size checking:	CHECKB, CHECKH, CHECKW

3 August 2019

In the discussion of priorities in the case of multiple, simultaneous exceptions, a mention of the “Bad Array Index” exception was added. This exception cannot co-occur with any Page-related exceptions, since it can only be caused by the INDEX__ instructions, which don’t access memory.

Additional discussion of the FCVTFI and FCVTIF instructions was added. A commentary section titled “Overflow for FCVTFI” was added.

Minor changes and rewordings were added and some typos were corrected.

16 August 2019

Added a commentary about floating comparisons with NaN. Explained why FNE is not included in the instruction set.

18 August 2019

Added clarification about “unused/zero” bits in the **csr_status** register. They cannot be modified.

Added clarification about **csr_prevpc**, which is a 64 bit register. When the PC is copied to this register, the upper 28 bits will be set to zero. When this register is copied to the PC, the upper 28 bits will be ignored.

Added: Cause codes are zero-extended to 64 bits whenever the hardware writes them into a CSR register.

21 August 2019

The version number in the **csr_version** register is specified to be 0x0001.

22 August 2019

Concerning the TLB registers, this sentence was added: "All bits of each register, including bit [5] which marked as "unused," can be read and written by the TLBREAD and TLBWRITE instructions."

6 September 2019

Updates were made to the Simple Serial device concerning how UTF-8 is handled. [The Simple Serial device was later replaced by the UART device.]

The ADD3, CONTROL, and CONTROLU instructions were added.

8 November 2019

The DZ "Divide-by-zero" bit was added to CSR_STATUS and the CSR_STATUS was altered to squeeze it in. In CSR_STATUS register, the DZ bit was inserted as bit 4 and bits 4-8 were shifted to 5-9. (The FCLASS instruction was changed, but a subsequent change eliminated the FCLASS instruction.)

11 November 2019

Added a commentary describing a hypothetical **csr_limit** register (which would be to watch for stack overflow) and why it was not included.

7 October 2020

Footnotes were added discussing error conditions for FCVTIF and FCVTFI. This behavior needs to be reviewed and is subject to change / correction / improvement.

18 April 2021

Previously, the **Null Address Exception** was defined to occur if the address of 0 was used. It has been redefined to include any address within 0…7, i.e., the last 3 bits are now to be ignored in the check. The reason for this change is that some array operations (e.g., **arraySize**) look at the current size at offset 4 without ever reading offset 0. Without this change, use of a null pointer fails to cause an exception.

10 May 2021

The **csr_core** register was created to replace **csr_extra2**. The **csr_extra1** register was renamed **csr_extra**.

24 May 2021

The GETSTAT, PUTSTAT instructions were added. The FCLASS instruction was eliminated.

2 June 2021

The following instructions were added:

INDEX0, …, INDEX32, MULADD, MULADDU

MUL was changed to synthetic. Previously, MUL was optional; MULADD and MULADDU are now mandatory, never emulated. The CHECKA instruction was eliminated, since the INDEX__ instructions are superior.

The format of register **csr_version** was changed.

Register **csr_temp1** was renamed **csr_temp**. Registers **csr_temp2** and **csr_temp3** were renamed **csr_resv1** and **csr_resv2**. Register **csr_extra** was renamed **csr_prod** which was also defined and described.

The DMA Controller memory-mapped I/O device and corresponding interrupt was added.

The Secure Storage memory-mapped I/O device was changed. The allocated space was enlarged and the semantics of locking was changed. Discussion of the boot process was improved.

10 July 2021

Discussion of interrupt priority was improved.

The RESTART instruction was added.

22 October 2021

The CAS and FENCE instructions were added.

LOADs and STOREs are defined to be atomic if they are aligned. (Previously, atomicity was guaranteed only for byte and halfword sizes.)

15 November 2021

The NULLTEST instruction was added.

18 October 2022

There are a number of changes to the ISA. The ISA documented here is now called version 2.0.

The version number in **csr_version** is incremented to 2.

Registers **csr_trapvec** and **csr_pgtable** replace **csr_resv1** and **csr_resv2**. The trap processing will now load **PC** from **csr_trapvec**. (Previously the trap handler was at fixed address 0x0_0001_8000.)

Page tables are introduced and the TLB organization is completely changed. TLB registers are now an optional cache of page table entries.

The following instructions and exceptions have been eliminated:

TLBREAD
TLBWRITE
TLBPUSH
TLBSET
TLBCLR
TLBDELETE
TLBCHECK

TLB Miss Exception
TLB Write Exception
TLB Copy-on-write Exception
TLB Execute Exception
TLB Privilege Exception

The following instructions and exceptions have been added:

TLBCLEAR

TLBFLUSH
CHECKADDR

Page Illegal Address Exception
Page Table Exception
Page Invalid Exception
Page Write Exception
Page Fetch Exception
Page Copy-On-Write Exception
Page First Dirty Exception

The Control and Status Registers (CSRs) were reordered and renumbered.

The opcodes for the machine instructions have been renumbered.

The chapter “Power-On-Reset and the Boot Sequence” was created and added.

The chapter “Memory-Mapped I/O” was rewritten.

9 February 2023

The ASID was moved from **csr_status** to the upper bits of **csr_pgtable**. The Physical Page Number (PPN) field of **csr_pgtable** was extended from 20 to 30 bits, allowing the root node of the page table to be located anywhere in the 16 TiByte physical memory space. The diagram of the virtual-to-physical mapping was corrected to show 30 bit PPNs, instead of 20 bit PPNs.

17 February 2023

The CHECKADDR instruction is modified to return a code number, instead of a trap cause code. Trap cause codes might be renumbered in the future and this reduces dependencies. Formerly, the CAS instruction was named “compare-and-swap”; it has been renamed to “compare-and-set”.

23 April 2023

The ENTERFUN and EXITFUN instructions have been added.

30 April 2023

The **StackLimit** field is added to **csr_status**. The Stack Overflow Exception is added.

6 October 2023

The Simple Serial Device was eliminated. The PLIC, UART0, and DISK0 peripheral memory-mapped I/O devices were added.

4 December 2023

Reorganized the discussion of Memory-Mapped I/O. Moved much material to separate document: *“Blitz-64: Memory-Mapped I/O Devices”*

10 December 2023

The “Kernel Exception” was eliminated and exceptions were renumbered. The CONTROL and CONTROLU instructions were modified.

Acronym List

ASID	Address Space ID
CISC	Complex Instruction Set Computer
CLA	Carry Lookahead Adder
CSR	Control and Status Register
DMA	Direct Memory Access
DZ	Divide by zero (within FLOAT_STATUS within **csr_status**)
EDC	Error Detection and Correction
EOF	End of File
ISA	Instruction Set Architecture
KPL	Kernel Program Language
LLBL	Low Level BootLoader program
LSB	Least Significant (rightmost) Bit or Byte
MBR	Master Boot Record
MMU	Memory Management Unit
MSB	Most Significant (leftmost) Bit or Byte
MTE	Matching TLB Entry
NV	Invalid operation (within FLOAT_STATUS within **csr_status**)
NX	Inexact (within FLOAT_STATUS within **csr_status**)
OF	Overflow (within FLOAT_STATUS within **csr_status**)
OS	Operating System
PC	Program Counter
PLIC	Platform-Level Interrupt Controller
PPN	Physical Page Number
PTE	Page Table Entry
PWM	Pulse Width Modulation
RAM	Random Access Memory (i.e., “main memory”)
RD	Round Down (within FLOAT_ROUND within **csr_status**)
RISC	Reduced Instruction Set Computer
RN	Round Up (within FLOAT_ROUND within **csr_status**)
r/o	Read Only
r/w	Read / Write
ROM	Read-Only Memory
RU	Round Up (within FLOAT_ROUND within **csr_status**)
RZ	Round toward Zero (within FLOAT_ROUND within **csr_status**)
SBC	Single Board Computer
SSBL	Second Stage BootLoader program
SMP	Shared Memory Multiprocessor

TCB	Thread Control Block
TLB	Translation Lookaside Buffer (i.e., the page table cache)
UART	Universal Asynchronous Receive Transmit
UF	Underflow (within FLOAT_STATUS within **csr_status**)
VPN	Virtual Page Number