Blitz-64:
Summary of the Machine Architecture

Harry H. Porter III
Portland State University

HHPorter3@gmail.com

18 October 2022

This document gives an overview of the Instruction Set Architecture (ISA) of the Blitz-64 processor core.

Table of Contents

The Blitz-64 Processor

General Purpose Registers

There are 16 registers. Each register is 64 bits (8 bytes) in size. Register **r0** is fixed at zero. All other registers (**r1** … **r15**) are treated equally and identically by the machine instructions.

By convention some registers have special functions and are given alternative names. Registers **r1** … **r7** are used for argument passing. Register **s0** … **s2** are as work registers. Register **t** is used for temporary results in synthetic instructions. Register **sp** is the stack pointer. Register **lr** is a link register used in function call and return. Register **gp** is a global pointer to shared data. Register **tp** is a thread pointer for thread-specific data.

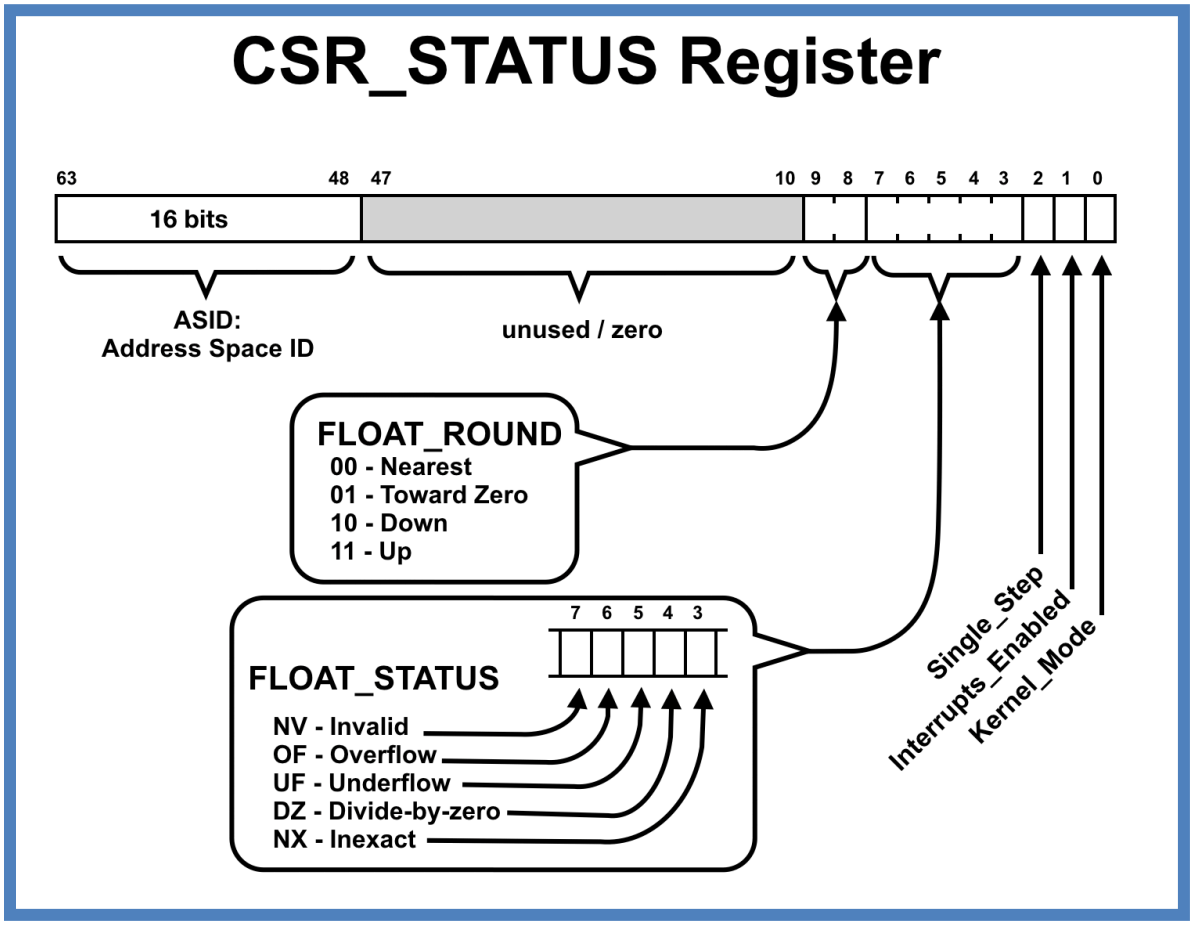
Control and Status Registers

There are 16 **Control and Status Registers** (**CSRs**). Each is 64 bits and each has a special, dedicated functions. Five CSRs are read-only. A particularly important register is **csr_status**, which is often called the "status word". (See the diagram on the next page.)

Among other things, the CSRs are used to enforce the privilege/protection of kernel code, to control and mask interrupts, and to facilitate fast switching between processes.

The CSRs may only be accessed by privileged instructions. That is, they are only available to code running in Kernel Mode.

There is only one set of general purpose registers and one set of CSRs; they are not copied or shadowed.



Kernel and User Mode

There are two processor modes: **Kernel Mode** and **User Mode**. Some instructions are **privileged**. Privileged instructions may only be executed when running in Kernel Mode. All remaining instructions are non-privileged and may be executed regardless of the current operating mode.

Memory and Address Spaces

Memory is byte addressable and Big Endian. Address spaces are identified by a 16 bit **Address Space Identifier** (**ASID**). The ASID of the currently executing process is contained in the **csr_status** register.

Program-generated addresses are 36 bits in length, allowing for a 64 GByte address space.

Kernel memory and all memory-mapped I/O devices are constrained to reside in the lower 32 GBytes of the address space. The lower 32 GBytes may only be addressed by the operating system, i.e., code running in Kernel Mode.

Application code — code running in User Mode — may only address bytes in the upper 32 GBytes. The upper 32 GBytes is the virtual address space of a process. Of course, these addresses will be mapped into physical memory and I/O addresses via the page table and TLB registers by the **Memory Management Unit (MMU)**.

Page Tables

The Blitz-64 **page size** is 16 KBytes.

Page tables are only **two levels**, not three or four as in other systems. This relatively flat page table fits naturally and makes the entire virtual space accessible.

With page tables, **physical memory up to 16 TBytes** is supported, i.e., using 44 bit physical addresses.

One advantage of a larger page size and flatter page table is that page faults happen less frequently and page table lookup time is reduced when compared to three-level tables and more complex organizations.

Furthermore, a smaller number of **Translation Lookaside Buffer** (**TLB**) registers is needed to capture a process's working set. Together, these effects reduce context switching time. The Address Space ID (ASID) in the status word allows TLB registers to remain valid and usable across multiple context switches.

Page Table Entries (PTEs) can be marked valid, executable, writable, copy-on-write, and dirty. There are several exception types associated with various types of page fault.

Machine Instructions

Machine instructions are 32 bits in size. In addition, the instruction encoding supports compressed instructions of varying sizes.

At this time, the compressed instructions have not been defined. After a larger code body has been created and can be statistically studied, the compressed instruction set will be defined. In order to define the most profitable encoding, we need to determine instruction execution frequencies accurately. The danger in defining the compressed instruction set prematurely is a confirmation bias effect in which the compiler favors generating compressed instructions because they are compressed.

Synthetic Instructions

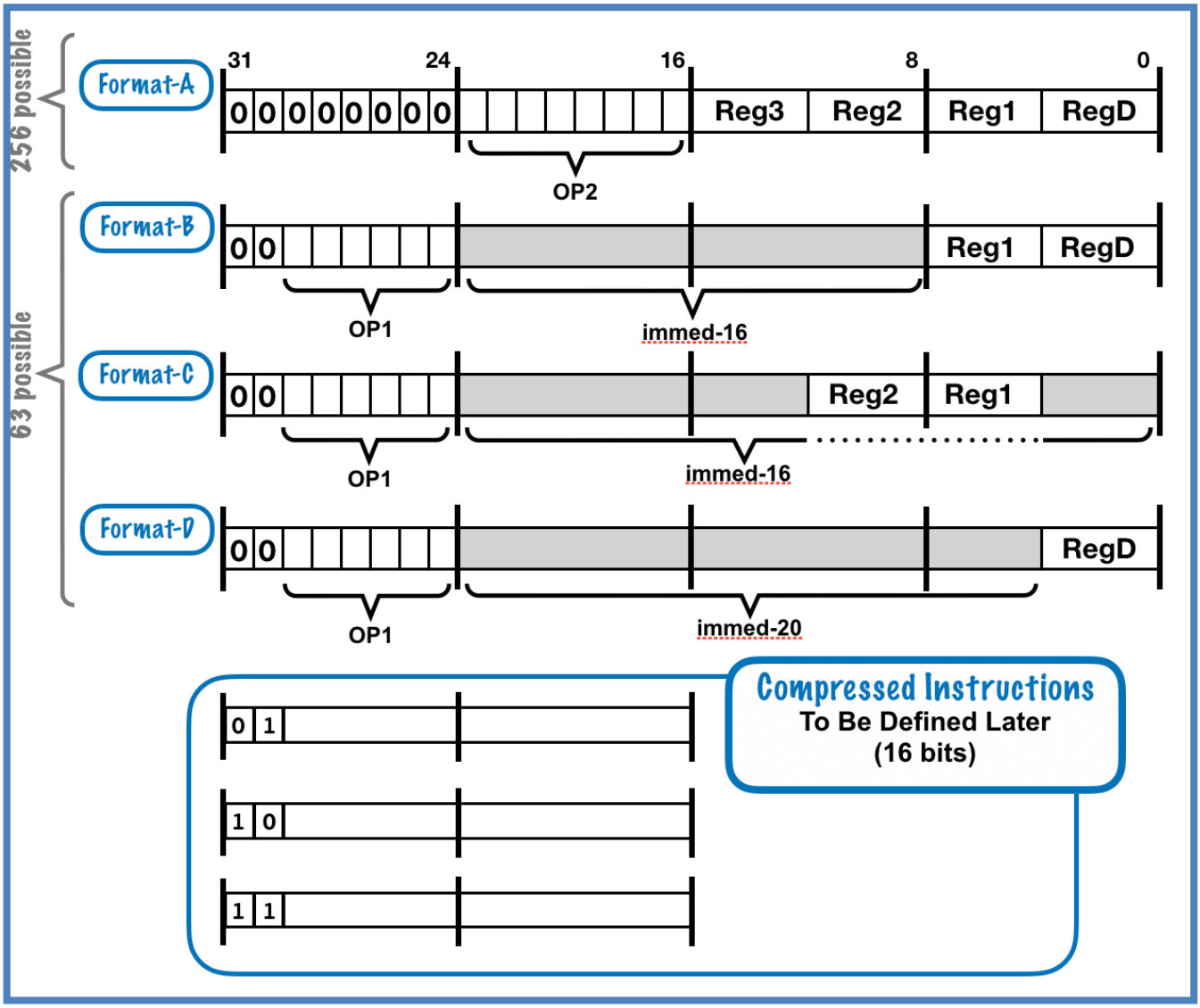
A number of instructions are synthetic, which means they are not implemented in hardware. Instead, synthetic instructions are translated by the assembler into equivalent machine instructions that perform the same function. The distinction between synthetic and machine instructions is invisible to the programmer and compiler. Since the machine instruction set is smaller and significantly simpler, the hardware logic is simplified.

One value of the synthetic instructions is that the programmer / compiler can use full sized addresses and data values (i.e., up to 64 bits), while the machine instructions only allow immediate values that are limited to 16 or 20 bits.

Each synthetic instruction is usually translated into a single machine instruction, although occasionally two or more instructions are required if a particularly large immediate value is involved.

Instruction Formats

There are 4 forms for instructions, termed **Formats A**, **B**, **C**, and **D**. Format A supports only register operands, with up to 4 registers. Formats B and C instructions include a 16 bit immediate value and 2 registers. Format D supports a 20 bit immediate value with only one register. (See the diagram.)



Assembly Code

The assembly code is typical. For example:

```
MyLabel:    addi    r5,r3,0x123    # Compute r5 = r3 + 0x123
```

The destination register is always first — on the left — as in the above example.

Here is another example:

```
Loop_3:                             # REPEAT
            loadd   r1,0(r2)        #   Fetch 8 bytes from memory
            stored  0(r3),r1        #   Store the doubleword
            addi    r2,r2,8         #   Increment pointers
            addi    r3,r3,8         #   .
            addi    r4,r4,-1        #   Decrement counter
            bnez    r4,Loop_3       # UNTIL counter is 0
```

Natural Data Types

Blitz-64 is inherently 64 bit. The primary data type is the 64-bit signed integer and most of the instructions are designed for them. Instructions to support and convert between legacy sizes such as 8, 16, and 32 bits are also provided.

The range of a 64 bit signed number is huge. With 64 bit numbers, there is no need to perform arithmetic in the other smaller sizes. There is certainly no need for unsigned numbers, which are error-prone and commit the mathematically dubious practice of ignoring negative values. Instructions for perform arithmetic / logic operations on legacy sizes of 8, 16, and 32 bits are not present in Blitz-64.

To save space in memory, numbers must sometimes be squeezed into smaller spaces. Blitz-64 includes instructions to support squeezing, either with range checking or without, as the application requires.

Data must be aligned in memory with the natural requirements:

Data Type	**Size**	**Alignment Required**
doubleword	64 bits	8 bytes
word	32 bits	4 bytes
halfword	16 bits	even addresses
byte	8 bits	<none>

Support for Legacy Code

The Blitz-64 system is not intended to support C or C++. While there is nothing that prevents these languages from being compiled into Blitz-64 assembler, this was not a design priority.

Any traditional implementation of the C and C++ programming languages assumes optimized support for 32 bit arithmetic. However, **Blitz-64 is strongly 64 bit**, and the emphasis is shifted away from the smaller data sizes of C and C++, such as int, int32_t, uint16_t, …

The KPL Language

KPL stands for **Kernel Programming Language**. KPL is the primary systems programming language for Blitz-64. KPL is used exclusively where C or C++ would be used in a traditional Unix-based operating system.

KPL is similar to C / C++ with support for:

- Classes and object-oriented programming
- Direct use of pointers and memory manipulation
- Separate compilation of large programs
- Linkage with assembly code programs
- Familiar constructs (IF, WHILE, FOR, …)
- Familiar printing with printf format strings

KPL differs from C / C++ in several ways, such as:

- Try-Throw-Catch mechanism
- Greater attention to error detection and reporting
- Simpler syntax
- Cross-package specification checking
- Parameterized classes (as opposed to template copying)
- Range and overrun checking for all arrays

Floating Point

Blitz-64 includes floating point instructions.

There is no separate set of floating point registers. Instead, the general purpose registers are used. This decision was taken in order to reduce context switch times, by reducing the size of a process's state and thus the time required to save that state.

Only double precision floating point is supported; single precision is not supported. It is assumed that number-crunching applications will use specialized processors for floating point computation. But since there is occasionally a need for floating point computation, double precision is included in the Blitz-64 architecture.

Some Blitz-64 cores may choose to implement the floating point instructions directly in hardware. Other implementations may choose not provide hardware support, in which case the instructions will cause an Emulated Instruction Exception, and will be implemented in software by trap handlers.

Error-Handling Philosophy

Blitz-64 seeks to catch all programming errors and we put an emphasis on performing **as much runtime checking as possible**. We understand that execution speed and performance are important, but the Blitz-64 position is that error-checking is neglected in other ISAs.

This is a significant difference between Blitz-64 and other processor architectures.

Modern software is growing exponentially in complexity. The proliferation of subtle bugs is intolerable. Software flaws are difficult to identify and expunge. Blitz-64 adopts a decidedly conservative and cautious approach, adding much more runtime error checking than competing ISAs. It is hoped that this tradeoff will ultimately make the software more reliable and fault-tolerant, as well as making the debugging process faster and more thorough.

For example, by restricting arithmetic to 64 bit signed numbers, we reduce overflow possibilities dramatically since most commonly occurring integer values are handled without issue. And by also **checking all arithmetic operations for overflow**, Blitz-64 deals with errors directly, rather than simply ignoring problems and producing incorrect result values.

It remains to be seen whether the demand for fault-tolerance is vigorous enough to support the Blitz-64 philosophy. The Blitz-64 approach targets mission critical applications — where correctness outweighs speed — over non-critical software, such as gaming and entertainment.

Exception Handling

Blitz-64 defines a number of types of exceptions. **Exceptions** are caused by an error arising during the execution of an instruction. Any exception will invoke “exception handling". There is a single trap hander routine, which will save thread state and dispatch to the appropriate handler code.

In some case, such as page faults, the exception handler will repair the problem and execution will resume where it was interrupted.

For fatal problems, the exception handler will return to the interrupted process, but will **throw an error** in the KPL language. A fault-tolerant application program will **catch** and the error and take corrective action. For simpler programs which do not provide code to catch the error, the exception will immediately invoke debugging.

The Blitz-64 ISA was designed in such a way that the location at which an exception occurs is always captured. The system always **reports the exact nature of the error and the exact location at which it occurred**, leading to faster and easier debugging. The error reporting is in source code file and line number form.

All arithmetic instructions are checked for overflow and any overflow problem arising during execution will cause an **Arithmetic Exception**. It is simply unacceptable to continue any computation silently with incorrect results, and Blitz-64 strives to avoid this.

The “load” and “store” instructions access memory and these instructions have alignment requirements. Normally, all KPL code will meet the requirements, but for those occasions where the alignment requirement is not met, a **Load / Store Alignment Exception** will be signaled. The handler code can be invoked to complete the operation and Blitz-64 includes several instructions designed to speed this operation.

The “null” value is commonly used in pointer operations. As every programmer knows, a common program mistake is to dereference a null pointer. With Blitz-64, addresses are always checked in hardware and a **Null Address Exception** will be signaled to let the programmer know exactly what has happened and where it happened.

Any attempt to execute invalid instructions will result in an **Invalid Instruction Exception**. Furthermore, any attempt by User Mode code to execute privileged Kernel Mode instructions will result in this same exception.

There are several exception types to support debugging, namely the **Debug Exception**, the **Breakpoint Exception**, and the **Singlestep Exception**.

Some instructions (typically the floating point instructions) may not be implemented in a particular core, for example, to save chip real estate or reduce processor complexity. When such instructions are encountered at runtime, an **Emulated Instruction Exception** will be signaled, allowing OS handler code to emulate the missing operation. The executing program code will be none-the-wiser.

Certain exceptions must never occur when the processor is executing handler code, or else infinite regress occurs. Thus, an exception will be promoted to a **Kernel Mode Exception** if it occurs within handler code.

A processor core may perform some internal error checking. When a violation is detected, a **Hardware Fault Exception** will be signaled, allowing failsafe procedures to be initiated.

A number of exception types are associated with supporting page tables and virtual address spaces. If a user process tries to access kernel memory, a **Page Illegal Address Exception** is signaled. If the page table register is incorrect, a **Page Table Exception** is signaled. If a page is missing or invalid, a **Page Invalid Exception** is signaled. If a write to a page not marked writable is attempted, a **Page Write Exception** is signaled. If an instruction is fetched from a page not marked executable, a **Page Fetch Exception** is signaled. If a write to a non-dirty page is attempted, a **Page First Dirty Exception** is signaled. If a write to a non-dirty page marked copy-on-write is attempted, a **Page Copy-On-Write Exception** is signaled.

System Call and Return

The “syscall” instruction is treated as an exception. A jump is made to the trap handler, which dispatches to the appropriate handler function code. The syscall instruction contains a 10 bit immediate value, essentially allowing for 1,024 distinct syscall instructions. So for the 1,024 most common syscalls, the dispatching is optimized to be particularly fast. The corresponding system call “return” instruction is designed so that, for short operations not requiring a context-switch, the return is simple and fast.

Asynchronous Interrupts

There are several sources for external interrupts and each is handled in a manner similar to synchronous exceptions. Interrupts coming from outside the instruction stream are said to be “asynchronous”, since their timing is unrelated to the instruction currently in execution. Examples include **Timer Interrupts**, **I/O Interrupts**, **DMA Controller Interrupts**, and **Communication Interrupts** from tightly coupled cores in a multi-processor array.

Asynchronous interrupts invoke trap handling and exception processing in the same way as synchronous exceptions caused by error conditions occurring during normal instruction execution.

Instruction Categories

Blitz-64 includes 123 unique machine instructions and 60 synthetic instructions. These are fully described in the Instruction Set Architecture (ISA) document.

Roughly speaking, the instructions can be grouped into the following categories:

- Arithmetic
- Logic and shifting
- Sign extension and range checking
- Byte / Endian reordering
- Testing (with boolean result)
- Test and branch
- Data manipulation for large values

- Call, return, switch, and method dispatch
- Memory load and store
- Support for unaligned load and store
- System call and return
- CSR register manipulation
- TLB register flushing
- Debug / breakpoint
- Sleep / shutdown / power control
- Floating point computations
- Methodology to accommodate unspecified / non-standard instructions

Existing Tools

The following software tools are completed at this time:

- Assembler
- Linker
- Library creation tool
- KPL Compiler
- Blitz-64 Processor Virtual Machine (Emulator)

The assembler is a full-function assembler, with expression evaluation and a number of assembler directives (i.e., pseudo-ops). Transforming synthetic instructions into machine code sequences is done by both the assembler and the linker.

The assignment to memory addresses can affect the translation from synthetic instructions to machine instructions. Furthermore, the translation from synthetic instruction to machine instruction can affect the address assignments. The assembler and linker work together with complex algorithms to find the best translation.

The assembler handles the full assembly language. The KPL compiler compiles the full KPL language. And the virtual machine emulates the full Blitz-64 ISA. In fact, the virtual machine emulates a multiprocessor with an arbitrary number of cores.

The KPL compiler is written in C++ and the remaining tools are written in C. Both the assembler and the compiler have also been ported to KPL.

Workload and Environmental Expectations

The following seem to be distinct, non-overlapping market niches:

- Small, embedded systems — such as the AVR chip and Arduino platform
- Systems running Linux/Unix — such as the ARM, x-86, and Risc-V

The Blitz-64 architecture is targeted at the space between these extremes.

The core design is targeted at application loads in which each core will typically host 100 to 1000 simultaneous threads, with perhaps 100 virtual address spaces. Address spaces are expected to range from very small, up to 1 or 2 gigabytes. Although the maximum virtual address space is 32 GiBytes, we expect applications larger than a few gigabytes to be broken into multiple cooperating processes, for a variety of reasons.

Multiple cores — perhaps on a single chip — are expected to be arrayed in two or three dimensional arrays. Multiprocessor arrays of smaller, simple cores are expected to be more widely deployed in the future, in support of complex parallel applications.

Blitz-64 is intended to be a cleaner core design which can be implemented with a smaller silicon footprint. Our thinking is that a smaller core allows more cores to be placed on a single die, thus increasing the overall computational horsepower within a single silicon package.

Simplicity, Reliability, and Usability

Smaller, simpler designs are more easily understood. For applications requiring high reliability, a complex computer system presents a challenge to implementation, testing, verification, and certification. Blitz-64 is intended to be simple enough to be understood by mortals, yet powerful and functional enough to be deployed effectively for serious engineering applications. We also feel there is a need for a system of moderate complexity for educational purposes, and hope that Blitz-64 can also fill this void.

Documentation

Please consult the following for more detail:

- *Blitz-64: ISA Quick Reference Card* (6 pages)
- *Blitz-64: Instruction Set Architecture Reference Manual* (340 pages)
- *Blitz-64: Assembler, Linker, and Object File Format* (289 pages)
- *An Introduction to KPL: A Kernel Programming Language* (207 pages)
- *KPL Syntax* (17 pages)

These are available online at: **Blitz64.org**

Open Source

The Blitz-64 design is open and free to use without license. The software is also free and open.

Hardware Core - System Verilog IP

There is at least one implementation of the Blitz-64 processor core — The Weedman Core — written in System Verilog. For details, contact HDL Express at:

www.hdlexpress.com

Ongoing Work

The Blitz-64 project is active and ongoing. Future work includes these themes:

- Porting the software toolchain to KPL
- Designing and creating the Blitz OS
- Increasing and improving the documentation
- Refinement of the hardware IP cores

Document Revision

Document History

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

Date	Author
20 February 2020	Harry H. Porter III <document created>
6 March 2020	Harry H. Porter III
18 October 2022	Harry H. Porter III <current version>

Permission to Copy

About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s. The computer has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. Porter also designed and constructed the BLITZ System, a collection of software designed to support a university-level course on Operating Systems. Using the software, students implement a small, but complete, time-sliced, VM-based operating system kernel. Porter has habit of designing and implementing programming languages, the most recent being a language specifically targeted at kernel implementation.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.

Porter lives in Portland, Oregon. When not trying to figure out how his computer works, he skis, hikes, travels, and spends time with his children building things.

Professor Porter's website: www.cs.pdx.edu/~harry