Blitz-64:
Emulator Reference Manual

Harry H. Porter III

HHPorter3@gmail.com

18 October 2022

This document describes the Blitz-64 emulator.

Table of Contents

Chapter 1: Overview

Emulators and Virtual Machines

In order to be run, a program must be written and compiled for a particular computer. However, if you want to execute the program on a different computer system, there is a problem. The executable program file cannot be run on a computer other than the one it was originally intended to execute on.

To run the program on a different computer, we use a virtual machine. A virtual machine is a piece of software that sits between one computer (the host) and the application program. It provides the illusion to the application program that it is running on the computer it was intended for, although it is actually running on a different computer.

The computer actually being used is called the "**host computer**". The computer that the application program was compiled for is called the "**target computer**".

There are two different kinds of "**virtual machine**" software. We use the term "**interpreter**" for one type and "**emulator**" for the other. We should note that people often use the term "virtual machine" for either or both types.[1]

The distinction depends on whether the target machine is real hardware or not.

---

[1] Emulators were invented first and were called "virtual machines" back in the days of the IBM System/360. Around the time of LISP and the Pascal language, the term "bytecode interpreter" was used. Early interpreters had a terrible reputation for poor performance. The Java language extended the term "virtual machine" (somewhat incorrectly) to mean byte code interpreter, in order, I think, to avoid the connotation of inefficiency and because "virtual" sounded groovy at the time. As for the inefficiency associated with interpreters, several things have happened. First, portability has become much more important and worth the loss of performance. Second, computers have gotten fast enough for people to accept the degradation in speed for many applications. Third, compiler technology has improved, reducing the performance penalty of interpreted code.

Emulator

The idea is that the target computer is a real computer but, for one reason or another, that hardware is unavailable. The programmer wishes to develop code for the target computer, running and debugging it, without using the actual hardware on which it is intended to run. The virtual machine software simulates the real, physical target hardware.

Interpreter

The idea is that programs will always run on the virtual machine software and no real hardware will ever exist. The goal is to facilitate portability. With a new host computer, only the virtual machine software needs to be ported. Then, millions of application programs become runnable all at once. In this case, the design of the target "machine" is tailored to facilitate fast interpretation, easy compiling, and ease of portability. Often, it would be impractical or impossible to implement the target machine in hardware.

As an example of an emulator, imagine that you have obtained copy of a program meant to run on an extinct device (such as Tetris on a GameBoy) and you wish to run it on your laptop. You need an emulator that can emulate the GameBoy as the target machine on your host computer.

Emulators are often used during the development of new computer hardware. The development of the hardware proceeds in parallel to the development of the software. Code is developed and debugged on the emulator before the hardware is fully available and many programmers can be coding using emulators until they can get their hands on physical devices.

The Java Virtual Machine is a good example of an interpreter. There are many Java programs in existence and many Java Virtual Machines installed. You've probably got a Java Virtual Machine on your computer. So you can easily run some Java code you've downloaded from the web on your computer, whether its a Mac, PC, or Linux box. The same goes for Python, although Python seems to have reverted to using the term "interpreter" rather than "virtual machine".

We should also mention "**simulators**". Computer software is digital and bits are either 0 or 1. At least this is the way software views the machine hardware. However,

at a lower level, the hardware is composed of analog components such as transistors.

During the development of new circuitry, “simulators” are used to model the performance of analog devices. Due to the analog nature of the hardware (voltage levels, resistance and capacitance, manufacturing variations, etc.), the model is imperfect. A simulator models the hardware at a deeper level where uncertainty and probabilities exist. Generally speaking, the term “simulate” is used when the modeling is not 100% exact and “emulate” when the model is perfectly exact.

Simulators run much slower than emulators because they are modeling at a greater level of detail. As an example, an emulator might model a register using a 64 bit integer variable and emulate a MOV instruction with an assignment statement. On the other hand, a simulator might model each individual transistor in the register. Each transistor involves several bits of data as well as information about wires and connectivity. To model a MOV instruction might involve the switching of hundreds of transistors, which causes the execution of tens of thousands of instructions in the simulator. So a simulator can easily run 100 times slower than an emulator.

A simulator is usually used to debug the hardware. An emulator is usually used to debug the software.

An emulator will execute all machine instructions exactly the same way as the hardware would. Thus, a running program cannot tell whether it is running directly on real, physical hardware or whether it is running on top of an emulator. Since the emulator will precisely mimic the hardware, every register and every byte of memory will contain exactly the same contents, whether emulated or running on hardware. Every instruction will execute exactly as specified by the Instruction Set Architecture.

However, the may be “holes” in the illusion and there may be several differences between the execution environment provided by emulator and that of physical hardware. For example, there may be a real-time clock and a program might be able to use it to determine whether it is running on an emulator or on hardware. It can use the clock to time the execution of a block of its own code and then, based on the speed of execution, the program can make determine whether it is being emulated or running on hardware.

Also, there may be explicit differences with the I/O devices or other parts of the system that differ between the emulator and physical hardware. The emulator is,

after all, for the purpose of developing software. It may be unnecessary to emulate the full machine in order to fulfill the emulator's function. Put another way, there may be many versions of the hardware, each with slightly different I/O configurations. One device might have MicroSD slot while another device has a USB connector instead. The emulator can be consider just another variation, with its own particular configuration.

The Blitz Emulator: An Introduction

The Blitz emulator is a program named "**blitz**". It is written in "C" and runs on the host computer, e.g., on a Mac laptop. The program emulates a Blitz core, which allows programs that have been written, compiled, and targeted for Blitz hardware to execute on a host machine such as a Mac laptop.

The Blitz emulator is a line-oriented program and is run from a shell command line.

The Blitz emulator is capable of loading a Blitz executable file in to to memory. The executable file should be prepared (i.e., compiled, assembled, linked) beforehand and will exist as a file on the host machine. It will conform to the executable file format, as described in the document "*Blitz-64: Assembler, Linker, and Object File Format*".

The Blitz emulator will emulate the Blitz processor core, including:

- The General Purpose Registers
- The Control and Status Registers (CSRs)
- The Translation LookAside Buffer Registers
- The Main Memory
- Several I/O Devices

The Blitz Emulator is capable of emulating a multi-core processor. In many situations, only a single core system is needed, but in other situations there may be a need to emulate several cores, operating on shared memory.

The emulator can be configured with a configuration file, which contains a number of parameters that describe the target machine. The configuration file is read upon startup of the emulator. The configuration parameters include:

- The number of cores
- The topological arrangement if the target is multi-core
- The amount of shared main memory
- The amount of non-shared memory, private and local to each core
- The number of TLB registers
- Where in target memory the memory-mapped I/O devices are placed
- The starting value for the PC (program counter)
- The behavior of the Blitz DEBUG machine instruction

The emulator runs it either of two modes:

- Auto-go ON
- Auto-go OFF

The "auto-go" mode is determined by a command line flag. Here is an example command line, where "auto-go" is enabled.

```
Shell% blitz -g MyExamplePgm.exe
```

The "Shell%" represents the Unix/Linux shell prompt.

The "auto-go" mode is enabled with the "-g" option. With auto-go, the emulator immediately begins execution.

Without auto-go, the emulator begins in command mode, which gives the user control to execute commands before execution starts.

The emulator is either executing Blitz machine instructions or is in "**command mode**". Command mode is also called "**debugging mode**".

In command mode, the emulator is driven by commands, which the user enters. The user types a command, it is executed, and the emulator then prompts for the next command.

Here are some of the important emulator commands:

Command	Abbreviation	
`help`	`h`	Display a menu of commands
`quit`	`q`	Terminate the emulator
`go`	`g`	Begin instruction execution

`step`	`s`	Execute a single instruction
`regs`	`r`	Display register contents
`info`	`i`	Display additional details about core state
`r`***N***		Update register ***N*** (r1, r2, ...)
`csr`		Update CSR register
`dumpMem`	`dm`	Display main memory
`setmem`		Modify main memory
`dis`		Disassemble contents of memory
`stack`		Display the runtime execution stack
`globals`		Display contents of global variable

We will describe these commands in detail later in this document.

The executable file contains information helpful to the debugger. This information comes from the KPL source code. For example, the executable file contains:

- Variable names (locals, globals, and parameters)
- Information about KPL functions and methods
- Information about KPL statements
- Information about stack frames

As we said above, the emulator is the debugger. When we refer to the "**debugger**", we mean the emulator operating in command mode. In other words, when the emulator is not executing instructions, it is in command mode. The user can use the commands listed above to debug the code.

Generally speaking, the debugging functions of the emulator work closely with the compiler and the information placed in the executable file, in order to present as much information as possible in source-level, KPL terms.

For example, when an error occurs, the emulator will immediately show the source file name and the line number within the source code where execution was at the moment of the error. The user can immediately view the calling stack, to see which functions are active and the values of the local variables, often presented in a form determined by their KPL types.

The emulator reads in all the debugging information when it starts up. This information comes from the executable file and is not placed in the main memory of the target Blitz machine.

When Blitz code is executing on Blitz hardware and not being emulated, the debugging functions will be performed by the "**native debugger**" and not by the emulator running in command mode. The native debugger, which is written in KPL (with some Blitz assembly), is invoked by the operating system when errors occur. The native debugger is not documented here.

Chapter 2: Example Usage

Summary

This chapter walks through a simple example. The following emulator commands will be demonstrated:

`go`	Start execution
`regs`	Display register values
`stack`	Show the runtime stack
`globals`	Show values of global variables
`watch`	Watch for updates to a memory location
`hex`	Convert hex to decimal
`where`	Find out where execution is
`find`	Find the location of a function
`dis`	Disassemble memory
`dumpmem`	Display memory contents
`dumpmem2`	Display memory contents
`step`	Single step execution
`r`***N***	Modify a register
`setmem`	Modify memory
`quit`	Quit the emulator

Compiling and Linking a Program

We start by creating a small KPL program called **MyProgram**. Here is the ".h" **header file** and the ".c" **code file** for our program:

MyProgram.h:

```
header MyProgram
```

```
    uses PrintPackage
    functions
      main ()
  endHeader
```

MyProgram.c:

```
  code MyProgram
    function main ()
      printf ("Hello, world\n")
    endFunction
  endCode
```

These files can be created with your favorite text editor.[2]

Next, we compile this package to produce a ".s" **assembly file**. The KPL compiler is a tool called "**kpl**".

```
Shell% kpl MyProgram -d ../ -o MyProgram.s
```

In this document, "`Shell%`" will be used to represent the Unix/Linux shell prompt. User input is shown **`like this`**.

Our example package uses **PrintPackage**, which uses packages **System** and **HostInterface**. The compiler will need to access the header files for these packages. The "-d" option to the compiler is followed by the directory pathname where these header files are to be found, if not in the current directory. So this assumes that files "**../PrintPackage.h**", "**../System.h**", and "**../HostInterface.h**" all exist.

Next, we invoke the Blitz assembler to produce a ".o" **object file**. The Blitz assembler is a tool called "**asm**":

```
Shell% asm MyProgram.s -o MyProgram.o
```

Next, we must link the object file using the Blitz "**link**" command, as shown next.

(This line has been broken into multiple lines for clarity.)

```
Shell% link MyProgram.o ../runtime.o ../HostInterface.o
                    ../System.o ../PrintPackage.o
```

[2] The TextEdit app for macOS works for me.

```
-o MyProgram.exe -k
```

Our example package uses **PrintPackage**, which uses packages **System** and **HostInterface**. All KPL programs must also be linked with some assembly functions, which come from **runtime.s**.

To keep things simple, we are not using any libraries for this example.

In practice the Unix/Linux "**make**" facility could be used. The compile/assemble/link commands would be collected in a "**makefile**" and the user would simply type "make" to compile, assemble, and link any and all files necessary, according to dependencies and details about which files have been modified recently:

```
Shell% make
```

Typing "make" is a lot quicker and the **make** facility makes sure that everything that needs to be updated will get updated.

We assume that **runtime.s** and the **System**, **HostInterface**, and **PrintPackage** packages have been compiled and assembled previously. Using an appropriate **makefile** will ensure they get recompiled and reassembled if they have been modified and their object files are out of date.

Assuming there were no errors, then the following executable file has been produced.

MyProgram.exe

We are ready to run our example program using the emulator.

Invoking the Emulator

Next, we will invoke the emulator tool from the Unix/Linux shell command line. The emulator prints a few opening lines and then waits for a command.

```
Shell% blitz MyProgram.exe
Reading executable file...
The executable file (MyProgram.exe) was loaded. The _entry address (0x00001885C)
was loaded into the PC.
```

```
=========================================
=====                               =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III      =====
=====         6 August 2019           =====
=====                               =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E>
```

When in debugging mode (i.e., command mode), the emulator prints the prompt

```
E>
```

and and waits for user input. If we type "**q**" (or "**quit**"), then emulator immediately terminates.

If we type "**g**" (or "**go**"), then emulator begins executing instructions and we see output from the program:

```
E> g
Beginning execution...
Hello, world

====================  KPL PROGRAM TERMINATION  ====================
Done!
E>
```

At the prompt, we can type "**r**" (or "**regs**") to display the contents of the registers:

```
E> r
=============================================================
  csr_instr    = 0x0000000000000772
  csr_cycle    = 0x0000000000001656
  csr_timer    = 0x7ffffffffffe9b8
  csr_status   = 0x0000000000000003
     ( ASID: 0x0000,  FlRound: 00 Nearest,
       NV/OF/UF/DZ/NX: 00000,  SingleStep: 0,
       InterruptsEnabled: 1,  KernelMode: 1 )
  csr_stat2    = 0x0000000000000000
  csr_prevpc   = 0x0000000000000000
  csr_cause    = 0x0000000000000000
  csr_bad      = 0x0000000000000000
  csr_addr     = 0x0000000000000000
  csr_ptr      = 0x0000000000000000
=======================  REGISTERS  =======================
  r1        = 0x0000000000000000
```

```
  r2        = 0x0000000000000001     ( decimal: 1 )
  r3        = 0x0000000000001c30     ( decimal: 7216 )
  r4        = 0xffffffffffffffff     ( decimal: -1 )
  r5        = 0x0000000000001e68     ( decimal: 7784 )
  r6        = 0x000000000000000d     ( decimal: 13 )
  r7        = 0x000000000000000d     ( decimal: 13 )
  r8     t  = 0x000000000000000d     ( decimal: 13 )
  r9    s0  = 0x00000000ffffffb8     ( decimal: 268435384 )
  r10   s1  = 0x00000000ffffffb8     ( decimal: 268435384 )
  r11   s2  = 0x0000000000000000
  r12   tp  = 0x0000000005f77078     ( decimal: 100102264 )
  r13   gp  = 0x0000000000010000     ( decimal: 65536 )
  r14   lr  = 0x000000000000bcac     ( decimal: 48300 )
  r15   sp  = 0x00000000fffffc0      ( decimal: 268435392 )
      Instruction time (all cores) = 1906
===============  NEXT INSTRUCTION TO EXECUTE  ===============
  PC = 0x000018968       Address = 0x18968  [ PHYSICAL ]
   Within Function "EmulatorShutdown"   [runtime.s]
      000018968: 19000040       jump       TerminateRuntime          # PC + 0x4
E>
```

In this document, I am editing the computer output a little, but I'm only changing the spacing to make long lines easier to read.

As another example, we can execute the emulator with the "**auto-go**" option, which is enabled with "**-g**" on the command line. We also use "**-nowarn**", which suppresses warnings and unnecessary messages.

We will also modify our program by adding a call to the **EmulatorShutdown** function:

MyProgram.c:

```
code MyProgram
  function main ()
    printf ("Hello, world\n")
    EmulatorShutdown (0)
  endFunction
endCode
```

We can recompile it:

```
Shell% make
```

Now when we execute our KPL program, we see nothing but the output from the program.

```
Shell% blitz MyProgram.exe -g -nowarn
Hello, world
Shell%
```

Debugging: An Example

Next we will modify our example to add a couple of functions. This code has a bug, as we will see.

MyProgram.c:

```
1:   code MyProgram
2:
3:     function main ()
4:       var
5:         i: int = 4
6:         j: int = 5
7:       foo1 (i + j)
8:       printf ("Goodbye\n")
9:       EmulatorShutdown (0)
10:    endFunction
11:
12:    function foo1 (x: int)
13:      foo2 (x)
14:    endFunction
15:
16:    function foo2 (myArg: int)
17:      printf ("myArg = %d\n", myArg)
18:      myArg = 1 / (myArg - 9)
19:    endFunction
20:
21:  endCode
```

In this document, some lines will be highlighted

```
   like this
```

to focus your attention on the most relevant information.

Next, we compile and run this program:[3]

[3] This is an example where long lines were edited to make them easier to read.

```
Shell% make
...
Shell% blitz MyProgram.exe -g -nowarn
myArg = 9
====================
==================== "System: ERROR_ArithmeticException" was thrown but not
                                          caught within thread "Main Thread"
====================

The CATCH STACK is empty


**********  RUNTIME ERROR: An "ARITHMETIC EXCEPTION" has occurred!  **********

    Offending Instruction = 0x0000000000050767

*****  Native debugger is not implemented - EXECUTION TERMINATING  *****

**********  EMULATOR DEBUGGING: Type 'stack' for more info. **********

Execution is stopped at ASSIGN on line 18 in function "foo2"   [MyProgram.c]
      005FAF0E4: 00050767       div        r7,r6,r7
Done!

Entering machine-level debugger...
=========================================
=====                               =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III      =====
=====         6 August 2019           =====
=====                               =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for
                                                    info about commands.
E>
```

This program begins in function **main**, which then calls function **foo1** which then calls function **foo2**. Function **foo2** prints the message “myArg = 9” and then attempts to divide by zero.

This division-by-zero causes an **Arithmetic Exception**, which throws an error. unfortunately, our little program fails to catch this error, so there is a problem. The initial error handling occurs in Blitz and the first line indicates the nature of the error. Then the program gives up and ceases execution.

Next, emulator debugging begins with a message (also highlighted above), telling where in the source code the problem arose:

- On **line 18** in file **MyProgram.c**
- In an **ASSIGNMENT** statement
- Within a function named "**foo2**"

Finally, the debugger prints a prompt and waits for a user command.

Next, let's enter the "**stack**" command. This will give a summary of the calling history for this function.

```
E> stack
   Function/Method                 Execution at...          File
   ============================    ====================     ==============================
   EmulatorDebuggingRequested                               runtime.s
   invokeDebugger                  CALL       line 2312     System.c
   RuntimeErrorArithmeticExceptio  CALL       line 2190     System.c
   _runtimeErrorArithmeticExcepti                           runtime.s
   foo2                            ASSIGN     line 18       MyProgram.c
   foo1                            CALL       line 13       MyProgram.c
   main                            CALL       line 7        MyProgram.c
   _kplEntry                                                MyProgram.c
   _entry                                                   runtime.s

--------------------  EmulatorDebuggingRequested  --------------------
Execution is stopped within Function "EmulatorDebuggingRequested"   [runtime.s, line 0]
            Code Address: 000018954
                   Frame: 00ffffed0 - 00ffffee0,   size = 0x10 (decimal 16)
     offset   0 0x0000... 00ffffed0: 000000000000cc38    codeAddress: int = 52280
I can show you the frames of the callers. How many more frames would
              you like to see (hit ENTER if none)?
```

The stack shows the functions that are active at the time of the error (**main**, **foo1**, and **foo2**) and I have highlighted these lines. We also see two functions that are called upon program startup (**_entry**, and **_kplEntry**) which are still active. After the error arises, four more functions are invoked as part of the error handling sequence, but we can ignore these.

After that, we see a representation of the stack frame at the top of the stack. This is for a function called **EmulatorDebuggingRequested** . This function and the other three functions at the top of the calling stack are not particularly interesting since they happen after the error.

This ends with a request for the user to type a number. The stack only contains 9 functions, but we will type 999 in order to see the entire stack.

```
I can show you the frames of the callers. How many more frames would
              you like to see (hit ENTER if none)? 999
```

```
--------------------  invokeDebugger  --------------------
Execution is stopped at CALL on line 2312 in function "invokeDebugger"   [System.c]
            Code Address: 000006a04
                  Frame: 00ffffee0 - 00ffffef0,   size = 0x10 (decimal 16)
 arg offset  16 0x0010... 00ffffef0: 000000000000cc38    codeAddress: int = 52280

--------------------  RuntimeErrorArithmeticException  --------------------
Execution is stopped at CALL on line 2190 in function "RuntimeErrorArithmeticException"
                                                                    [System.c]
            Code Address: 0000064dc
                  Frame: 00ffffef0 - 00fffff58,   size = 0x68 (decimal 104)
 arg offset 104 0x0068... 00fffff58: 000000000000cc38    codeAddress: int = 52280
 arg offset 112 0x0070... 00fffff60: 0000000000050767    offendingInstr: int = 329575
     offset  80 0x0050... 00fffff40: 0000000000002be8    errorID: String = "System:
ERROR_ArithmeticException"
     offset  88 0x0058... 00fffff48: 0000000000006498    codeAddr_notUsed: int = 25752

--------------------  _runtimeErrorArithmeticException  --------------------
Execution is stopped within Function "_runtimeErrorArithmeticException"
                                                                  [runtime.s, line 0]
            Code Address: 000018774
                  Frame: 00fffff58 - 00fffff70,   size = 0x18 (decimal 24)

--------------------  foo2  --------------------
Execution is stopped at ASSIGN on line 18 in function "foo2"   [MyProgram.c]
            Code Address: 00000cc30
                  Frame: 00fffff70 - 00fffffa0,   size = 0x30 (decimal 48)
 arg offset  48 0x0030... 00fffffa0: 0000000000000009    myArg: int = 9

--------------------  foo1  --------------------
Execution is stopped at CALL on line 13 in function "foo1"   [MyProgram.c]
            Code Address: 00000cb98
                  Frame: 00fffffa0 - 00fffffb0,   size = 0x10 (decimal 16)
 arg offset  16 0x0010... 00fffffb0: 0000000000000009    x: int = 9

--------------------  main  --------------------
Execution is stopped at CALL on line 7 in function "main"   [MyProgram.c]
            Code Address: 00000cb40
                  Frame: 00fffffb0 - 00ffffff0,   size = 0x40 (decimal 64)
     offset  40 0x0028... 00fffffd8: 0000000000000004    i: int = 4
     offset  48 0x0030... 00fffffe0: 0000000000000005    j: int = 5

--------------------  _kplEntry  --------------------
Execution is stopped within Function "_kplEntry"   [MyProgram.c, line 0]
            Code Address: 00000c8a4
                  Frame: 00ffffff0 - 00ffffff8,   size = 0x8 (decimal 8)

--------------------  _entry  --------------------
Execution is stopped within Function "_entry"   [runtime.s, line 0]
            Code Address: 000018888
                  Frame: 00ffffff8 - 010000000,   size = 0x8 (decimal 8)
E>
```

The frames for the three functions of interest are highlighted. We can see that the error occurred in function **foo2** which was called from function **foo1**, which was called from the **main** function.

Now let's look more closely at the frame for the **main** function. Here are the relevant line from above, repeated with different highlighting:

```
--------------------  main  --------------------
Execution is stopped at CALL on line 7 in function "main"   [MyProgram.c]
            Code Address: 00000cb40
                   Frame: 00fffffb0 - 00ffffff0,   size = 0x40 (decimal 64)
     offset  40 0x0028... 00fffffd8: 0000000000000004    i: int = 4
     offset  48 0x0030... 00fffffe0: 0000000000000005    j: int = 5
```

We see that the call (to **foo1**) occurred at line 7 in file **MyProgram.c**. We also see the values of the local variables **i** and **j**. These variables have type integer, and their values are shown in decimal.

To illustrate the capabilities of the debugger to display the values of variables in human-friendly source form, let's create another function **foo3**:

```
function foo3 (myArg1: int, myArg2: bool)
  var
    localVar1: byte
    localVar2: halfword
    localVar3: word
    localVar4: int
    localVar5: double
    localVar6: String
    localVar7: Person
    localVar8: ptr to Person
    localVar9: array [5] of byte
    …
    myArg1 = 123
    myArg2 = true
    localVar1 = 'a'
    localVar2 = 12345
    localVar3 = 100200300
    localVar4 = MAX_64
    localVar5 = 3.141596
    localVar6 = "greetings"
    localVar7 = new Person {f = 57}
    localVar8 = & localVar7
    localVar9 = new array of byte {11,22,33,44,55}
    <<< Execution is stopped here >>>
    …
endFunction
```

Assume that execution is stopped within this function. Here is what we might see with the "stack" command.[4]

[4] This output was edited a little, but only spacing to make long lines easier to read.

```
--------------------  foo3  --------------------
Execution is stopped at RETURN on line 47 in function "foo3"   [MyProgram.c]
            Code Address: 005faf284
                   Frame: 00fffff48 - 00fffffb0,   size = 0x68 (decimal 104)
 arg offset 104 0x0068... 00fffffb0: 000000000000007b    myArg1: int = 123
 arg offset 112 0x0070... 00fffffb8: 01                  myArg2: bool = true
     offset  94 0x005e... 00fffffa6: 61                  localVar1: byte = 'a'
                                                            (decimal 97)
     offset  92 0x005c... 00fffffa4: 3039                localVar2: halfword = 12345
     offset  88 0x0058... 00fffffa0: 05f8ef6c            localVar3: word = 100200300
     offset  24 0x0018... 00fffff60: 7fffffffffffffff    localVar4: int =
                                                            9223372036854775807
     offset  32 0x0020... 00fffff68: 400921fd1569f490    localVar5: doubleFloat =
                                                            3.141596
     offset  40 0x0028... 00fffff70: 0000000000002880    localVar6: String = "greetings"
     offset  48 0x0030... 00fffff78: 0000000005faf298    localVar7: object
     offset  64 0x0040... 00fffff88: 000000000fffff78    localVar8: ptr   ---> to a
                                                            Person object at 0x00fffff78
     offset  72 0x0048... 00fffff90: 0000000500000005    localVar9: array (currentSize =
                                                            5, maxSize = 5)
```

As you can see, the debugger displays values in terms appropriate for the type of the variable. However, the debugger lacks information about the fields in objects or the element types in arrays, which imposes some limits.

Debugging: A Second Example

In this example, we will exhibit the ability to:

- See if a particular statement is executed
- Stop the program at a given place
- Disassemble the assembly code
- Watch for any change to a variable
- Modify the variable

For this example, assume we have a global variable named **myGlob**:

```
var myGlob: int = -1
```

We will be invoking this function:

```
function foo4 (myArg: int) returns int
  myGlob = myArg + 456
  return myGlob * 100
endFunction
```

Let us assume that we want to find out where **myGlob** is being modified. So we start the program, but without the "-g" **auto-go** option:

```
Shell% blitz MyProgram.exe
```

We use the "**globals**" command to list all global variables:

```
E> globals
From package "MyProgram.c"...
   line 48     000002880: ffffffffffffffff    myGlob: int = -1
From package "runtime.s"...
From package "HostInterface.c"...
   line 84     0000023c8: 00000000000023e0    stdin: ptr  -->  ...
   line 85     0000023d0: 00000000000023e8    stdout: ptr  -->  ...
   line 86     0000023d8: 00000000000023f0    stderr: ptr  -->  ...
   line 87     0000023e0: 0000000000000000    stdinFILE: struct
   line 88     0000023e8: 0000000000000001    stdoutFILE: struct
   line 89     0000023f0: 0000000000000002    stderrFILE: struct
   line 61     0000023f8: 0000000000018e88    print: ptr  -->  ...
   line 62     000002400: 0000000000018e40    readString: ptr  -->  ...
   line 90     000002980: 0000000000000000    errno: int = 0
From package "System.c"...
   line 40     000000008: 00                  alreadyInAlloc: bool = false
   line 37     000018f48: 0000000000000000    TheHeapArray: array
...
```

The highlighted lines show where variable **myGlob** is located in memory.

Next, we use the "**watch**" command, which will prompt for a memory address.

```
E> watch
  Execution will halt whenever this address is stored into.
  Enter 0 to display the previous watch address.
  Enter -1 to cancel a previous watch address.
Enter the address in hex: 000002880
Execution will halt whenever address 0x000002880 is stored into.
E>
```

Next, we use the "**go**" command to begin execution:

```
E> g
Beginning execution...

**********  The value 0x000000000001e240 was stored into the 'watched'
                        address (000002880) at instr time = 366  **********
Done!
E>
```

We see the value, but it is in hex. Fortunately, the emulator has commands "**hex**", "**dec**", and "**ascii**" which come in handy to convert values.

```
E> hex
Enter a value in hex: 1e240
     hex: 0x000000000001E240     >120 KiBytes
     decimal: 123456
     ascii: ".......@"
     real: 6.09953683729769333578624768 9e-319
E>
```

We see that the decimal value is **123456**, as expected.

Next, we use the "**where**" command to see where execution is stopped.

```
E> where
Enter an address in hex (or 0 for current PC): << ENTER >>
CURRENT LOCATION OF PC:
  RETURN on line 53 in function "foo4"   [MyProgram.c]
      00000CD80: 04006406    movi     r6,100    # synthetic for XORI r6,r0,0x64
E>
```

Next, assume that we want to stop execution at a particular location in the source code. KPL contains a "**debug**" statement which we can use. We will insert a **debug** statement into out code, as shown next. The **debug** keyword is followed by a string, which becomes useful when we having several debug statements scattered through the program. For this example, we just use the string "here".

Let's add a **debug** statement to our function:

```
function foo4 (myArg: int) returns int
  debug "here"
  myGlob = myArg + 456
  return myGlob * 100
endFunction
```

We recompile the program and re-run it:

```
Shell% make
...
Shell% blitz MyProgram.exe -g
Reading executable file...
The executable file (MyProgram.exe) was loaded. The _entry address (0x00001885C)
was loaded into the PC.
Beginning execution...
```

```
****  A DEBUG machine instruction was executed  *****

Next instruction to execute:
   DEBUG (line 51)
   ----------  ################## here ##################
      00000CD74: 00280000       debug
Done!

Entering machine-level debugger...
=========================================
=====                                   =====
=====  The Blitz-64 Machine Emulator    =====
=====     by Harry H. Porter III        =====
=====         6 August 2019             =====
=====                                   =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E>
```

The program is now stopped. Let's look at the assembly code for this function.

First, we can look in the ".s" assembly code file, **MyProgram.s**. The file might be quite large, but we can search for our string "here" to find the **debug** statement.

Here are the relevant lines from the assembly code file:

```
#
# ===============  FUNCTION foo4  ===============
#
      .align         4
_function_6_foo4:
      .function      "foo4",line=50,framesize=8
      store.d        -8(sp),lr            # Save return addr
      addi           sp,sp,-8             # Allocate frame (8 bytes)
      stored         8(sp),r1             # myArg <-- r1
# Zero out 0 bytes of frame
# VARIABLE INITIALIZATION...
      .stmt          debug,line=51
      .comment       "################## here ##################"
      debug
# ASSIGNMENT STATEMENT...
      .stmt          assign,line=52
      loadd          r7,8(sp)             # myArg
      addi           r7,r7,456
      stored         _GlobalVar_myGlob,r7
# RETURN STATEMENT...
      .stmt          return,line=53
```

```
        movi            r6,100                  # 0x0000000000000064
        mul             r7,r7,r6
        mov             r1,r7
        addi            sp,sp,8
        load.d          lr,-8(sp)
        ret
        .local          8,"myArg",line=50,type="I"
        .endfunction
```

As you can see, the KPL compiler adds some commenting.

In reading the above code, note that the **.function**, **.stmt**, **.comment**, **.local**, and **.endfunction** directives provide supplemental debugging information. This debugging information will be present in the executable file, but will not be loaded into memory during execution. This information is read from the executable file by the emulator and used to facilitate debugging. These directives are how the compiler communicates information to the debugger, so the debugger can display data in human-friendly forms.

The highlighted lines above correspond to the following assignment statement.

```
    myGlob = myArg + 456
```

Next, in the debugger, we use the "find" command to determine at what address the function "foo4" is located. Private functions have names beginning with "**_function**" so we enter this when prompted.

```
E> find
Enter the first few characters of the symbol; all matching will be printed: _func
  Symbol                                Value (hex)  Value (decimal)  Label  Source line number
                                                                                        / filename
  ================================  ===========  ===============  =====  ====================
  _function_137_printClassNameFromDPT         6AF0            27376  LABEL  8567 System.s
  _function_138_printClassNameOfObject        6A24            27172  LABEL  8448 System.s
  _function_139_invokeDebugger                69C0            27072  LABEL  8387 System.s
  _function_140_KPLDefaultFatalErrorFunction  588C            22668  LABEL  6144 System.s
  _function_141_KPLMemoryFree_Version1        33D4            13268  LABEL  1642 System.s
  _function_142_KPLMemoryAlloc_Version1       32C8            13000  LABEL  1524 System.s
  _function_143_KPLMemoryFree_Default         3298            12952  LABEL  1492 System.s
  _function_144_KPLMemoryAlloc_Default        3204            12804  LABEL  1433 System.s
  _function_19_hostDateNext                   BE84            48772  LABEL  2299 HostInterface.s
  _function_20_hostDateSize                   BE5C            48732  LABEL  2277 HostInterface.s
  _function_21_argumentNext                   BCE8            48360  LABEL  2106 HostInterface.s
  _function_22_argumentSize                   BCC4            48324  LABEL  2086 HostInterface.s
  _function_26_LocalPrintString               75D8            30168  LABEL   521 PrintPackage.s
  _function_27_LocalPrintChar                 758C            30092  LABEL   475 PrintPackage.s
  _function_6_foo4                            CD68            52584  LABEL   670 MyProgram.s
  _function_7_foo3                            CBBC            52156  LABEL   496 MyProgram.s
  _function_8_foo2                            CB90            52112  LABEL   471 MyProgram.s
  _function_9_foo1                            CB6C            52076  LABEL   446 MyProgram.s
E>
```

The highlighted line shows that the function **foo4** is located at address 0x00000cd68.

Next, we use the disassemble command "**dis**" to display memory at that location. It displays about a page worth of data, but we show only the first half of it here:

```
E> dis
Enter the beginning address (in hex): cd68
   Function "foo4"   [MyProgram.c]
    _function_6_foo4:
      00000CD68: 22FFFEF8    store.d   -8(sp),lr     # offset = 0xFFF8
      00000CD6C: 01FFF8FF    addi      sp,sp,-8
      00000CD70: 220001F8    store.d   8(sp),r1
   DEBUG (line 51)
   ----------  ################# here #################
      00000CD74: 00280000    debug
   ASSIGN (line 52)
      00000CD78: 1E0008F7    load.d    r7,8(sp)
      00000CD7C: 0101C877    addi      r7,r7,456     # hex = 0x1C8
      00000CD80: 22288700    store.d   10368(r0),r7  # offset = 0x2880
   RETURN (line 53)
      00000CD84: 04006406    movi      r6,100        # synthetic for
                                                            XORI r6,r0,0x64
      00000CD88: 00040677    mul       r7,r7,r6
      00000CD8C: 03000071    mov       r1,r7         # synthetic for ORI _,_,0
      00000CD90: 010008FF    addi      sp,sp,8
      00000CD94: 1EFFF8FE    load.d    lr,-8(sp)     # offset = 0xFFF8
      00000CD98: 1A0000E0    ret                     # synthetic for
                                                            JALR r0,0(lr)
...
E>
```

I have highlighted the statement of interest so you can compare the output from the "**dis**" disassemble command to the original assembly code.

We can display raw memory as a sequence of doublewords, using the "**dumpmem**" command (which can be abbreviated to "**dm**"):

```
E> dm
Enter the starting address in hex: cd68
00000cd68:  00  0x22fffef801fff8ff  2522014657489664255   ".......
00000cd70:  08  0x220001f800280000  2449960361955688448   "....(..
00000cd78:  10  0x1e0008f70101c877  2161737678104676471   .......w
00000cd80:  18  0x2228870004006406  2461365630494860294   "(....d.
00000cd88:  20  0x0004067703000071     1133008128049265   ...w...q
00000cd90:  28  0x010008ff1efff8fe    72067485867702526   ........
00000cd98:  30  0x1a0000e000000000  1873498407058800640   ........
...
```

```
E>
```

We can also display the memory in a more byte-oriented way, using the "**dumpmem2**" command (which can be abbreviated to "**dm2**"):

```
E> dm2
Enter the starting (physical) memory address in hex: cd68
Enter the number of bytes in hex (or 0 to abort): 100
PRIVATE MEMORY:
00000cd68:  22FF FEF8  01FF F8FF  2200 01F8  0028 0000    ".......".....(..
00000cd78:  1E00 08F7  0101 C877  2228 8700  0400 6406    .......w"(....d.
00000cd88:  0004 0677  0300 0071  0100 08FF  1EFF F8FE    ...w...q........
00000cd98:  1A00 00E0  0000 0000  0000 0000  0000 CDB0    ................
...
E>
```

Recall that execution is stopped at the **debug** statement, directly before the assignment statement. Next we use the "**step**" command (which can be abbreviated "**s**") to execute a single machine instruction:

```
E> s
Executing this instruction:
      00000CD78: 1E0008F7       load.d     r7,8(sp)      # offset = 0x8
Instr count = 365
E>
```

Next, let's look see what value was loaded into register **r7**, using the "**regs**" command:

```
E> r
...
  r6        = 0x0000000005f77090     ( decimal: 100102288 )
  r7        = 0x000000000001e078     ( decimal: 123000 )
  r8     t  = 0x0000000005f70000     ( decimal: 100073472 )
...
===============  NEXT INSTRUCTION TO EXECUTE  ===============
  PC = 0x00000CD7C       Address = 0xCD7C  [ PHYSICAL ]
   Within ASSIGN (line 52)
      00000CD7C: 0101C877       addi       r7,r7,456           # hex = 0x1C8
E>
```

Now we can execute another instruction.

```
E> s
Executing this instruction:
      00000CD7C: 0101C877       addi       r7,r7,456           # hex = 0x1C8
Instr count = 366
E>
```

Once again, we might wish to examine the register with the "**regs**" command, but we don't need to illustrate that again.

Next, let's alter the value with the "**r7**" command. (There is one such command for each register: **r1**, **r2**, … **r15**).

```
E> r7
  r7 = 0x000000000001e240     ( decimal: 123456 )
Enter the new value (in hex): 9fbf1
  r7 = 0x000000000009fbf1     ( decimal: 654321 )
E>
```

Next, we execute the next instruction (**stored**).

```
E> s
Executing this instruction:
      00000CD80: 22288700       store.d    10368(r0),r7        # offset = 0x2880
Instr count = 367
E>
```

Next, we verify that the desired value was stored into the memory address for the variable **myGlob**, using both the "**dumpmem**" and "**globals**" commands.

```
E> dm
Enter the starting address in hex: 2880
000002880:  00  0x000000000009fbf1               654321   .......@
...
E> globals
From package "MyProgram.c"...
   line 48     000002880: 000000000009fbf1    myGlob: int = 654321
...
E>
```

Next, let's modify memory directly, using the "**setmem**" command. We will change variable myGlob to a new value.

```
E> setmem
Enter the (physical) memory address in hex of the doubleword to be modified:
2880
***** This address is in private RAM or shared RAM *****
The old value is:
0x000002880: 0x000000000009FBF1
Enter the new value (8 bytes in hex): 36870
0x000002880: 0x0000000000036870
E>
```

Next, we use the "**globals**" command to verify that we have changed the value correctly.

```
E> globals
From package "MyProgram.c"...
   line 48     000002880: 0000000000036870    myGlob: int = 223344
...
E>
```

Finally, we can resume program execution with the "**go**" command (which can be abbreviated "**g**"):

```
E> go
...
```

Or perhaps we will terminate the emulator with the "**quit**" command (which can be abbreviated "**q**"):

```
E> quit
Shell%
```

Chapter 3: Commands

Introduction

In this chapter we describe each command.

Several commands require arguments. The arguments are not entered on the same line. Each command is typed separately, followed by NEWLINE / ENTER / RETURN.

When arguments are required, the command will prompt for them.

In some cases, the argument must be a hex value. A leading "0x" is optional and either upper or lower can be used.

Generally speaking, the commands will do their best to verify that the user has entered legal values.

Several common commands have an abbreviation; either form can be used.

If an invalid command is entered, the emulator will complain and prompt for the next command.

```
E> aBadEntryyy
Unrecognized command.
Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E>
```

In a multi-core system, things can get confusing. At any one time, one of the cores is selected as the "**current core**". This code is the focus of many instructions. For example, the "**regs**" command will display the registers of the current core. The "**cores**" command can be used to change the focus.

The emulator is designed to display its output in a fixed-width font and it assumes that the output window is wide enough to accommodate long line. Within this

document, I have altered the spacing of some very long lines, to make the output more readable.

For example, the following lines from the emulator (which will be discussed later):

```
  4  csr_status   = 0x0000000000000001
                      ( ASID: 0x0000,  FlRound: 00 Nearest,  NV/OF/UF/DZ/NX:
00000,  SingleStep: 0,  InterruptsEnabled: 0,  KernelMode: 1 )

               TLB REGISTER      ASID    Virt Page    Phys Page    W   X   D   V
C
            ==================  ======  ===========  ===========  === === ===
=== ===
```

will be altered to the following:

```
  4  csr_status   = 0x0000000000000001
                      ( ASID: 0x0000,  FlRound: 00 Nearest,
                        NV/OF/UF/DZ/NX: 00000,  SingleStep: 0,
                        InterruptsEnabled: 0,  KernelMode: 1 )

             TLB REGISTER     ASID   Virt Page   Phys Page   W   X   D   V   C
          ================== ====== =========== =========== === === === === ===
```

When the emulator goes into command mode, it will print a welcome message, such as:

```
Shell% blitz MyProgram.exe
Reading executable file...
The executable file (MyProgram.exe) was loaded. The _entry address (0x00001885C)
was loaded into the PC.
=========================================
=====                                 =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III      =====
=====           6 May 2021            =====
=====                                 =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E>
```

In Blitz, program version numbers are not used. Instead, the author and date are used to indicate the version. From the above, you can see which version is documented in this document.

The Commands

q quit

This command immediately terminates the emulator.

If the ROM or SecureStorage has been updated, this command will ask about writing back to the host file before exiting.

The Unix/Linux “exit” code will be 0.

h help

This command produces the following display:

```
================================================================================
This program accepts commands typed into the terminal.  Each command
should be typed without any arguments; the commands will prompt for
arguments when needed.  Case is not significant.  Some abbreviations
are allowed, as shown.  Typing control-C will halt execution.

The available commands are:
  quit      - Terminate this program
  q
  help      - Produce this display
  h
  info      - Display the current state of the core
  i
  regs      - Display a summary of the registers
  r
  r1        - Change the value of register r1
  ...         ...
  r15         Change the value of register r15
  tlb       - Change the value of TLB register
  csr       - Change the value of CSR register
  pc        - Set the Program Counter (PC)
  setmem    - Used to alter memory contents
  ld        - Load 1/2/4/8 bytes from memory or I/O device
```

```
  st       - Store 1/2/4/8 bytes to memory or I/O device
  dumpMem  - Display the contents of memory
  dm
  dumpMem2 - Display the contents of memory (basic format)
  dm2
  dis      - Disassemble several instructions
  d        - Disassemble several instructions from the current location
  stack    - Display stack frames
  stack2   - Display stack frames, asking for thread details
  stackmem - Display top words of stack
  sm
  globals  - Display the global variables
  trans    - See what the MMU would do with a virtual address
  addr     - Enter an address; show it in page/offset format
  addr2    - Enter page and offset format; show as address
  read     - Read a doubleword from memory-mapped I/O region
  write    - Write a doubleword to memory-mapped I/O region
  cores    - Display the status of all cores for this processor
  sel      - Change the currently selected core
  symbols  - Display all symbols
  dinfo    - Display all debugging information
  find     - Find a symbol by spelling
  find2    - Find a symbol by value
  where    - Ask for an address and attempt to locate that in the source code
  go       - Begin or resume BLITZ instruction execution
  g
  step     - Single step; execute one machine-level instruction
  s
  stepn    - Execute N machine-level instructions
  n
  t        - Execute instructions until we encounter a CALL or RETURN
                                                        or EXCEPTION
  watch    - Stop execution when ever an address is stored into
  reset    - Reset the machine state and re-read the a.out file
  rerun    - Do a 'reset', followed by 'go'
  hex      - Convert a user-entered hex number into decimal and ascii
  dec      - Convert a user-entered decimal number into hex and ascii
  ascii    - Convert a user-entered ascii char into hex and decimal
  sim      - Display the current simulation constants; create "emulationParms"
  rom      - Create, manipulate files "emulationROM" and "emulationSecure"
  serial   - Control serial input
  sched    - Modify the multicore timeslice schedule
  startall - Change all STOPPED cores to RUNNING
  start    - Change selected cores to RUNNING
  stopall  - Change all RUNNING cores to STOPPED
  stop     - Change selected cores to STOPPED
  <nl>     - Print some useful info
==========================================================================
```

i info

This command produces a display showing the general state of the machine. An example is shown below.

This include:

General information about the entire system:
- Main memory details
- Memory-mapped I/O device locations
- Number of cores

Information about the "**current**" core:
- The TLB registers
- The CSR registers
- The general purpose registers
- The PC

```
E> i
============================
   Private RAM memory:         0x000000000 ... 0x010000000    256 MiBytes
   Shared RAM memory:          0x010000000 ... 0x010010000    64 KiBytes
   Bootstrap ROM memory:       0x400000000 ... 0x400100000    1 MiByte
   Secure Storage device:      0x400100000 ... 0x400104000    16 KiBytes
   Simple serial device addr: 0x400104000 ... 0x400108000    16 KiBytes
   Multi-core array (columns: 1, rows: 1, planes: 1); total number of cores = 1
   Addressable memory per core = 0x0000000010010000    >256 MiBytes
   Total physical memory       = 0x0000000010010000    >256 MiBytes
Number of Instructions Executed (so far) = 0
              TLB REGISTER      ASID   Virt Page   Phys Page   W   X   D   V   C
            ================== ====== =========== =========== === === === === ===
  TLB[0]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[1]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[2]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[3]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[4]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[5]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[6]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[7]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[8]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[9]:  0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[10]: 0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[11]: 0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[12]: 0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[13]: 0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[14]: 0x0000000000000000 0x0000 0x800000000 0x000000000
  TLB[15]: 0x0000000000000000 0x0000 0x800000000 0x000000000
```

```
=====  CONTROL AND STATUS REGISTERS  =====
  0  csr_version  = 0x0123456700000001     ( CyclesPerMilliSec: 0x01234567,
                                              Version: 0x0001 )
  1  csr_instr    = 0x0000000000000000
  2  csr_cycle    = 0x0000000000000000
  3  csr_timer    = 0x0000000000000000
  4  csr_status   = 0x0000000000000001
                      ( ASID: 0x0000,  FlRound: 00 Nearest,
                        NV/OF/UF/DZ/NX: 00000,  SingleStep: 0,
                        InterruptsEnabled: 0,  KernelMode: 1 )
  5  csr_stat2    = 0x0000000000000000
                      ( ASID: 0x0000,  FlRound: 00 Nearest,
                        NV/OF/UF/DZ/NX: 00000,  SingleStep: 0,
                        InterruptsEnabled: 0,  KernelMode: 0 )
  6  csr_prevpc   = 0x0000000000000000
  7  csr_cause    = 0x0000000000000000
  8  csr_bad      = 0x0000000000000000
  9  csr_addr     = 0x0000000000000000
 10  csr_ptr      = 0x0000000000000000
 11  csr_temp1    = 0x0000000000000000
 12  csr_temp2    = 0x0000000000000000
 13  csr_temp3    = 0x0000000000000000
 14  csr_extra1   = 0x0000000000000000
 15  csr_extra2   = 0x0000000000000000
=====  REGISTERS  =====
  r0  zero  = 0x0000000000000000
  r1        = 0x636f6c64626f6f74     ( decimal: 7165064710573748084 )
  r2        = 0x0000000010000000     ( decimal: 268435456 )
  r3        = 0x0000000010000000     ( decimal: 268435456 )
  r4        = 0x0000000000010000     ( decimal: 65536 )
  r5        = 0x0000000005f77110     ( decimal: 100102416 )
  r6        = 0x0000000000000000
  r7        = 0x0000000000000000
  r8     t  = 0x0000000000000000
  r9    s0  = 0x0000000000000000
  r10   s1  = 0x0000000000000000
  r11   s2  = 0x0000000000000000
  r12   tp  = 0x0000000000000000
  r13   gp  = 0x0000000000000000
  r14   lr  = 0x0000000000000000
  r15   sp  = 0x0000000000000000
=====  PROGRAM COUNTER  =====
  PC = 0x00001885C       Address = 0x1885C  [ PHYSICAL ]
   Within Function "_entry"   [runtime.s]
    _entry:
      00001885C: 15800001       upper16    r1,r0,-32768        # hex = 0x8000
============================
E>
```

Use the **cores** command to switch to a different core.

r regs

This command produces a display showing the registers of the currently selected core. For example:

```
E> r
============================================================
  csr_instr    = 0x0000000000000000
  csr_cycle    = 0x0000000000000000
  csr_timer    = 0x0000000000000000
  csr_status   = 0x0000000000000001
     ( ASID: 0x0000,  FlRound: 00 Nearest,
       NV/OF/UF/DZ/NX: 00000,  SingleStep: 0,
       InterruptsEnabled: 0,  KernelMode: 1 )
  csr_stat2    = 0x0000000000000000
  csr_prevpc   = 0x0000000000000000
  csr_cause    = 0x0000000000000000
  csr_bad      = 0x0000000000000000
  csr_addr     = 0x0000000000000000
  csr_ptr      = 0x0000000000000000
========================  REGISTERS  ========================
  r1         = 0x636f6c64626f6f74     ( decimal: 7165064710573748084 )
  r2         = 0x0000000010000000     ( decimal: 268435456 )
  r3         = 0x0000000010000000     ( decimal: 268435456 )
  r4         = 0x0000000000010000     ( decimal: 65536 )
  r5         = 0x0000000005f77110     ( decimal: 100102416 )
  r6         = 0x0000000000000000
  r7         = 0x0000000000000000
  r8      t  = 0x0000000000000000
  r9     s0  = 0x0000000000000000
  r10    s1  = 0x0000000000000000
  r11    s2  = 0x0000000000000000
  r12    tp  = 0x0000000000000000
  r13    gp  = 0x0000000000000000
  r14    lr  = 0x0000000000000000
  r15    sp  = 0x0000000000000000
      Instruction time (all cores) = 0
===============  NEXT INSTRUCTION TO EXECUTE  ===============
  PC = 0x00001885C       Address = 0x1885C  [ PHYSICAL ]
   Within Function "_entry"   [runtime.s]
    _entry:
      00001885C: 15800001       upper16    r1,r0,-32768        # hex = 0x8000
E>
```

This displays a subset of what the **info** command displays and is more convenient.

r1, r2, … r15

The registers can be modified individually with this command.

This command prints the previous value and asks for a new value.

For example:

```
E> r1
  r1 = 0x636f6c64626f6f74     ( decimal: 7165064710573748084 )
Enter the new value (in hex): 123456
  r1 = 0x0000000000123456     ( decimal: 1193046 )
E>
```

There is no ability to cancel, but with copy-paste you can just enter the previous value.

This instruction applies only to the currently selected core.

tlb

This command allows you to change a particular TLB register to a given value.

Each TLB register is made of several bit fields. This command first displays the current value of the register (a 64 bit value), together with a breakout of the bit fields.

Consult the Instruction Set Architecture manual for details of the fields.

This command first prompts for the number of the register to be modified. In this example, the cores are configured to have 16 TLB registers, each.

After displaying the current value, it prompts for the field individually. Finally, it packs the input fields into a 64 bit value, updates the register, and displays the new value.

For example:

```
E> tlb
Enter TLB number (0..15): 7
             TLB REGISTER      ASID   Virt Page   Phys Page   W   X   D   V   C
           ================== ====== =========== =========== === === === === ===
  TLB[7]:  0x0000000000000000 0x0000 0x800000000 0x000000000
Enter the new value...
Enter the 16 bit ASID: 1234
Enter the 21 bit Page Number: aaa
Enter the 21 bit Physical Page Number: bbb
Want to set WRITABLE bit to 1? y
Want to set EXECUTBLE bit to 1? n
Want to set DIRTY bit to 1? y
Want to set VALID bit to 1? n
Want to set COPY-ON-WRITE bit to 1? y
               TLB REGISTER      ASID   Virt Page   Phys Page   W   X   D   V   C
           ================== ====== =========== =========== === === === === ===
  TLB[7]:  0x123400555002EED5 0x1234 0x802AA8000 0x002EEC000  W       D       C
E>
```

The “Virt Page” and “Phys Page” columns give the starting address of the page. Virtual addresses will always have the upper bit set and the Physical addresses will never have the upper bits set. Both will always have the least significant 14 bits set to 0.

This instruction applies only to the currently selected core.

csr

This command allows the user to modify any CSR register. The old value is printed and the user is prompted to enter a new value.

For example:

```
E> csr
Enter CSR number (0..15): 1
  CSR [1] = 0x000000000000016C
Enter the new value (in hex): 1234
  CSR [1] = 0x0000000000001234
E>
```

CSR register 1 is **CSR_INSTR**, which is read-only. As this example shows, the user can modify registers that the ISA requires to be read-only, although it is unclear why that would be a good thing to do.

This instruction applies only to the currently selected core.

pc

This command displays the current value of the program counter (PC) and prompts the user to enter a new value.

The PC is updated and the new value is displayed.

As the next example shows, the debugger also prints out useful information about where in the program this location is and disassembles the machine instruction at that address.

```
E> pc
  Old PC = 0x00000CD70
Please enter the new value for the program counter (PC) in hex: 00000CD3C
  New PC = 0x00000CD3C
   Within ASSIGN (line 42)
    _Label_41:
      00000CD3C: 1000103C    b.eq   r3,r0,0x1C   # if (r3 == r0) goto _Label_39
E>
```

The additional information may not be available for some addresses. For example:

```
E> pc
  Old PC = 0x00000CD70
Please enter the new value for the program counter (PC) in hex: 120000
  New PC = 0x000120000
      000120000: 00000000
E>
```

At the prompt for a new PC, the user can hit NEWLINE / ENTER / RETURN to cancel the command.

This instruction applies only to the currently selected core.

setmem

This command is used to modify memory.

The command first prompts for an address. This address must be doubleword aligned.

Then it prompts for a doubleword value, which is written to memory. For example:

```
E> setmem
Enter the (physical) memory address in hex of
                        the doubleword to be modified: 120000
***** This address is in private RAM or shared RAM *****
The old value is:
0x000120000: 0x0000000000000000
Enter the new value (8 bytes in hex): 1234567890abcdef
0x000120000: 0x1234567890ABCDEF
E>
```

This command can be used to modify either shared main memory or the private main memory of the currently selected core.

This command can be used to write to the "**Boot ROM**" area. The ROM contents are kept in a file called "**emulationROM**". For example:

```
E> setmem
Enter the (physical) memory address in hex of
                        the doubleword to be modified: 400000000
***** This address is in Boot ROM, but you can proceed to store to it *****
The old value is:
0x400000000: 0x17000E4104000E02
Enter the new value (8 bytes in hex): 1111aaaa2222bbbb
0x400000000: 0x1111AAAA2222BBBB
E>
```

If it has been updated, the emulator will alert the user and ask about updating it at the time the emulator quits.

```
E> q
The ROM has been modified. Shall I write it out
                        to the host file ("emulationROM")? n
Shell%
```

This command can also be used to write to the "**Secure Storage Device**" area, which is discussed in the ISA manual. The contents are kept in a file called "**emulationSecure**". For example:

```
E> setmem
Enter the (physical) memory address in hex of
                      the doubleword to be modified: 0x400100000
***** This address is in Secure Storage *****
The old value is:
0x400100000: 0x8888777766665555
Enter the new value (8 bytes in hex): 1111222233334444
0x400100000: 0x1111222233334444
E>
```

If it has been updated, the emulator will alert the user and ask about updating it at the time the emulator quits. For example:

```
E> q
The SecureStorage has been modified. Shall I write it out
                      to the host file ("emulationSecure")? n
Shell%
```

ld

This command is used to read 1, 2, 4, or 8 bytes from the memory system and display it. The command prompts for the address and the number of bytes to read.

See also the "**st**" command, which writes instead of reads.

Address translation is performed (using the TLB registers as described in the Blitz Instruction Set Architecture). The command reminds the user of details such as the current ASID (Address Space Identifier) and Kernel Mode bits, which come from the CSR_STATUS register.

```
E> ld
  NOTE: The core is in kernel mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: cd74
Enter the size of the access (1,2,4, or 8): 4
Your Input Address = 0xCD74  [ PHYSICAL ]
Translated Address = 0xCD74  [ PHYSICAL ]
0x00000cd74: 0x00280000    (decimal: 2621440)
```

```
E>
```

In certain cases, the address translation will fail and would cause an exception if attempted by an executing program. For example, an attempt to access physical memory when running in User Mode will cause a TLB PRIVILEGE EXCEPTION.

This command will deal with exceptional addresses by indicating the problem. For example:

```
E> ld
  NOTE: The core is in user mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: cd74
Enter the size of the access (1,2,4, or 8): 4
Your Input Address = 0xCD74  [ PHYSICAL ]
  ----- This will cause a TLB_PRIVILEGE EXCEPTION
E>
```

As another example, consider an attempt to read a doubleword from an address that is not doubleword aligned:

```
E> ld
  NOTE: The core is in kernel mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: cd74
Enter the size of the access (1,2,4, or 8): 8
Your Input Address = 0xCD74  [ PHYSICAL ]
  ----- This will cause an UNALIGNED_LOAD_STORE EXCEPTION
E>
```

The **ld** command can be used to read from a memory-mapped I/O device. In the next example, we read from the location (0x4_0010_4000) used by the "**Simple Serial Device**" to get a single character of input from the user.

When the LOAD operation is performed, the emulator immediately hangs, waiting for the user to enter something. In this example, the user types the letter "q", followed by NEWLINE / RETURN / ENTER.[5]

```
E> ld
  NOTE: The core is in kernel mode.
```

[5] The NEWLINE is required because we are running in "cooked" mode. Had the emulator been in "raw" mode, pressing the letter alone would be sufficient. However, in "raw" mode, echoing of characters is not done, so we would not see the "q".

```
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: 400104000
Enter the size of the access (1,2,4, or 8): 8
Your Input Address = 0x400104000  [ MEMORY-MAPPED I/O ]
Translated Address = 0x400104000  [ MEMORY-MAPPED I/O ]
q
0x400104000: 0x0000000000000071    (decimal: 113)
E>
```

We see that the command reads the value of 0x71 from the device. We can use the "hex" command to see that this value is 113 in decimal and "q" in ASCII, as expected.

```
E> hex
Enter a value in hex: 71
     hex: 0x0000000000000071
     decimal: 113
     ascii: ".......q"
     real: 5.582941798006085949195227359e-322
E>
```

st

This command is used to write 1, 2, 4, or 8 bytes to the memory system. The command prompts for the address, the number of bytes to write, and the value to be written.

See also the "**ld**" command, which reads instead of writes.

Address translation is performed (using the TLB registers as described in the Blitz Instruction Set Architecture).

```
E> st
  NOTE: The core is in kernel mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: cd74
Enter the size of the access (1,2,4, or 8): 2
     Input Address = 0xCD74  [ PHYSICAL ]
Translated Address = 0xCD74  [ PHYSICAL ]
BEFORE: 0x00000cd74: 0x0028    (decimal: 40)
Enter new value: 1234
AFTER:  0x00000cd74: 0x1234    (decimal: 4660)
E>
```

Like the "**ld**" command, this command will perform address translation and, if an exception would occur, will abort the operation and report the exception.

This command can be used to write to memory-mapped I/O devices. In the next example, we write to the location used by the "**Simple Serial Device**" to write a single character to the output. The command confirms that this is the user's intent, then asks for the value to be written.

```
E> st
  NOTE: The core is in kernel mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: 400104000
Enter the size of the access (1,2,4, or 8): 8
     Input Address = 0x400104000  [ MEMORY-MAPPED I/O ]
Translated Address = 0x400104000  [ MEMORY-MAPPED I/O ]
***** This command appears to access a memory-mapped I/O device.
Do you want to proceed? y
Enter new value: 6b
kE>
```

In this example, we wrote the value 0x6b (decimal 107) which is the ASCII code for the letter "k". On the highlighted line, you can see that the character is inserted into the output stream when the operation is performed.

Note that when the address to be written is within a memory-mapped device's region, this command will not print the before or after values. Why? Because it would require LOADs from the device. For memory-mapped I/O devices, a simple LOAD may elicit some complex operation.

dm dumpmem

This command asks for a starting address. It then displays 30 doublewords of physical memory starting at that address.

The columns of the output are:

address ***offset*** ***value in hex*** ***value in decimal*** ***ASCII interpretation***

For example:

```
E> dm
Enter the starting address in hex: 40
000000040:  00  0x0000000100000001           4294967297   ........
000000048:  08  0x2000000000000000  2305843009213693952    .......
000000050:  10  0x0000000100000001           4294967297   ........
000000058:  18  0x3000000000000000  3458764513820540928   0.......
000000060:  20  0x0000000100000001           4294967297   ........
000000068:  28  0x3100000000000000  3530822107858468864   1.......
000000070:  30  0x0000000600000006          25769803782   ........
000000078:  38  0x46414c53450a0000  5062411376665427968   FALSE...
000000080:  40  0x0000000500000005          21474836485   ........
000000088:  48  0x545255450a000000  6076012602285096960   TRUE....
000000090:  50  0x0000000500000005          21474836485   ........
000000098:  58  0x46414c5345000000  5062411376664772608   FALSE...
0000000a0:  60  0x0000000400000004          17179869188   ........
0000000a8:  68  0x5452554500000000  6076012602117324800   TRUE....
0000000b0:  70  0x0000000200000002           8589934594   ........
0000000b8:  78  0x3078000000000000  3492541511025819648   0x......
0000000c0:  80  0x0000001400000014          85899345940   ........
0000000c8:  88  0x2d39323233333732  3258690996568012594   -9223372
0000000d0:  90  0x3033363835343737  3473179352671467319   03685477
0000000d8:  98  0x3538303800000000  3834868099782279168   5808....
0000000e0:  a0  0x0000000100000001           4294967297   ........
0000000e8:  a8  0x0a00000000000000   720575940379279360   ........
0000000f0:  b0  0x0000000b0000000b          47244640267   ........
0000000f8:  b8  0x2020202020206465  2314885530818471013         de
000000100:  c0  0x633a200000000000  7150062542776172544   c: .....
000000108:  c8  0x0000000200000002           8589934594   ........
000000110:  d0  0x2020000000000000  2314850208468434944     ......
000000118:  d8  0x0000001a0000001a         111669149722   ........
000000120:  e0  0x4144445245535320  4702959031022736160   ADDRESS
000000128:  e8  0x2020202020202020  2314885530818453536
E>
```

This command can print

- Private RAM of the currently selected core
- Shared RAM
- Bootstrap ROM
- Secure Storage

The addresses apply to the currently selected core and address translation is performed. If an exception would occur, this command will abort.

In some cases, the debugger can identify items of memory that appear to be objects. When possible, it identifies the object's class, as in this example:

```
E> dm
Enter the starting address in hex: 0x0000000000018f50
-----  This appears to be the beginning of a Person object at 0x000018f50  -----
000018f50:  00  0x000000000000ce00                52736   ........
000018f58:  08  0x000000000000006f                  111   .......o
000018f60:  10  0x00000000000000de                  222   ........
000018f68:  18  0x0000000000000000                    0
000018f70:  20  0x0000000000000000                    0
...
```

This happens when the first work happens to be the address of a Dispatch Table.[6]

In some cases, this command can identify a pointer to an object and it adds the class name as demonstrated in the first highlighting below.

In other cases, the debugger can identify a likely pointer, in which case it prints out the first 3 doublewords pointed to.

```
E> dm
Enter the starting address in hex: 00fffffa8
00fffffa8:  00  0x0000000000000000                    0
00fffffb0:  08  0x0000000000018f50               102224   .......P
                 ---> to a Person object at 0x000018f50
00fffffb8:  10  0x000000000000cb38                52024   .......8
                 ---> 220011f817002c47 0300001422000740 0300001701000877 ...
00fffffc0:  18  0x0000000000000018                   24   ........
                 ---> 5f00000000000000 0000000500000005 2020203078000000 ...
00fffffc8:  20  0x0000000000000004                    4
00fffffd0:  28  0x0000000000000005                    5
...
```

Actually, the emulator prints everything on one line, which is hard to read in this document, but looks better in a wider window:

```
E> dm
Enter the starting address in hex: 00fffffa8
00fffffa8:  00  0x0000000000000000                    0
00fffffb0:  08  0x0000000000018f50               102224   .......P   ---> to a Person object at 0x000018f50
00fffffb8:  10  0x000000000000cb38                52024   .......8   ---> 220011f817002c47 0300001422000740 0300001701000877 ...
00fffffc0:  18  0x0000000000000018                   24   ........   ---> 5f00000000000000 0000000500000005 2020203078000000 ...
00fffffc8:  20  0x0000000000000004                    4
00fffffd0:  28  0x0000000000000005                    5
...
```

[6] Of course, this could be coincidence. The debugger reads several words of memory, looking at the dispatch table and the class descriptor. It looks for the "magic number" that each class descriptor begins with and, it is matches, then it retrieves the class name.

In some cases, the debugger can identify a pointer to a string, in which case it prints out the string.[7] For example:

```
E> dm
Enter the starting address in hex: FFFFFE0
00fffffe0:  00  0x00000000000028b0                10416   ......(.
                 ---> "Now is the time"
00fffffe8:  08  0x000000000000c8c4                51396   ........
...
```

dm2 dumpmem2

This command can be used to display the contents of memory.

This command prompts for the starting address and the number of bytes. It prints the bytes. Each row prints 16 bytes in hex, followed by an ASCII rendition of the 16 bytes, using a period "." for each byte that is not a printable ASCII character.

For example:

```
E> dm2
Enter the starting (physical) memory address in hex: 4000
Enter the number of bytes in hex (or 0 to abort): 100
PRIVATE MEMORY:
000004000:  0100 0822  0300 0027  1E00 18F1  0300 0072    ..."...'.......r
000004010:  1E00 48F3  1914 A60E  1E00 50F1  0100 60FF    ..H.......P...`.
000004020:  1EFF F8FE  1A00 00E0  22FF FEF8  01FF C8FF    ........".......
000004030:  2200 31F8  2200 42F0  2200 00F0  2200 00F8    ".1.".B."...."...
000004040:  2200 10F0  2200 10F8  2200 20F0  2200 20F8    "..."...". .". .
000004050:  1E00 38F7  1D00 0477  0300 0071  1E00 40F7    ..8....w...q..@.
000004060:  1D00 0477  0300 0072  1900 2D0E  2200 21F8    ...w...r..-."."!.
000004070:  0400 0007  2200 17F8  0300 0017  01FF FF77    ...."..........w
000004080:  2200 17F0  1E00 18F7  2200 27F0  1E00 20F7    "......."."... .
000004090:  1E00 10F6  1200 6760  1E00 20F1  1E00 38F2    ......g`.. ...8.
0000040a0:  1E00 0028  0011 0088  0044 0810  0100 0811    ...(.....D......
0000040b0:  0001 0121  0300 0016  1E00 20F1  1E00 40F2    ...!...... ...@.
0000040c0:  1E00 0028  0011 0088  0044 0810  0100 0811    ...(.....D......
0000040d0:  0001 0121  0300 0017  1B00 0077  000F 0077    ...!.......w...w
0000040e0:  1F00 0760  1E00 20F7  0100 0177  2200 27F0    ...`.. ....w"."."
0000040f0:  19FF F9C0  0100 38FF  1EFF F8FE  1A00 00E0    ......8.........
E>
```

[7] Recall that the type **String** is a pointer to an array of bytes. The debugger looks for a pointer to something that looks like it could be an array of reasonable size and that, if so, would only contain printable ASCII codes.

This command can print

- Private RAM of the currently selected core
- Shared RAM
- Bootstrap ROM
- Secure Storage

Address translation does not occur.

dis

This command displays the contents of memory, interpreting the bytes as machine instructions.

This command prompts for a starting address and then disassembles about a page worth (i.e., 30) of instructions at that address.

For each 32 bit word, it prints

- The address
- The word in hex
- The machine instruction (opcode and arguments)
- Additional information as a comment

For example:

```
E> dis
Enter the beginning address (in hex): 4000
      000004000: 01000822    addi     r2,r2,8
      000004004: 03000027    mov      r7,r2         # synthetic for ORI _,_,0
      000004008: 1E0018F1    load.d   r1,24(sp)     # offset = 0x18
      00000400C: 03000072    mov      r2,r7         # synthetic for ORI _,_,0
      000004010: 1E0048F3    load.d   r3,72(sp)     # offset = 0x48
      000004014: 1914A60E    call     memoryCopy    # PC + 0x14A60
   RETURN (line 798)
    _Label_489:
      000004018: 1E0050F1    load.d   r1,80(sp)     # offset = 0x50
      00000401C: 010060FF    addi     sp,sp,96      # hex = 0x60
      000004020: 1EFFF8FE    load.d   lr,-8(sp)     # offset = 0xFFF8
```

```
      000004024: 1A0000E0    ret                     # synthetic for JALR r0,0(lr)
   Function "overwriteString"   [System.c]
    _P_System_overwriteString:
      000004028: 22FFFEF8    store.d  -8(sp),lr     # offset = 0xFFF8
      00000402C: 01FFC8FF    addi     sp,sp,-56     # hex = 0xFFC8
      000004030: 220031F8    store.d  56(sp),r1     # offset = 0x38
      000004034: 220042F0    store.d  64(sp),r2     # offset = 0x40
      000004038: 220000F0    store.d  0(sp),r0
      00000403C: 220000F8    store.d  8(sp),r0
      000004040: 220010F0    store.d  16(sp),r0     # offset = 0x10
      000004044: 220010F8    store.d  24(sp),r0     # offset = 0x18
      000004048: 220020F0    store.d  32(sp),r0     # offset = 0x20
      00000404C: 220020F8    store.d  40(sp),r0     # offset = 0x28
   CALL (line 821)
      000004050: 1E0038F7    load.d   r7,56(sp)     # offset = 0x38
      000004054: 1D000477    load.w   r7,4(r7)
      000004058: 03000071    mov      r1,r7         # synthetic for ORI _,_,0
      00000405C: 1E0040F7    load.d   r7,64(sp)     # offset = 0x40
      000004060: 1D000477    load.w   r7,4(r7)
      000004064: 03000072    mov      r2,r7         # synthetic for ORI _,_,0
      000004068: 19002D0E    call     _P_System_min # PC + 0x2D0
      00000406C: 220021F8    store.d  40(sp),r1     # offset = 0x28
   FOR_INIT (line 822)
      000004070: 04000007    movi     r7,0          # synthetic for XORI r7,r0,0x0
      000004074: 220017F8    store.d  24(sp),r7     # offset = 0x18
E>
```

As this example shows, the debugger will add labels, source code line numbers, and statement types when it can.

In some cases, such as for CALL and BRANCH instructions, the debugger will display the branch target symbolically. This is occurs in the highlighted line above.

If the section of memory being displayed does not contain instructions, then this command just prints out a series of 32 bit words. Occasionally, the bits may happens to constitute a valid machine instructions and you'll see output like this:

```
E> dis
Enter the beginning address (in hex): 0000000F0
    _StringConst_123:
      0000000F0: 0000000B                          # decimal = 11, ascii = "...."
      0000000F4: 0000000B                          # decimal = 11, ascii = "...."
      0000000F8: 20202020   store.h  8224(r2),r0   # offset = 0x2020
      0000000FC: 20206465   store.h  8293(r6),r4   # offset = 0x2065
      000000100: 633A2000                          # decimal = 1664753664, ascii = "c: ."
      000000104: 00000000
    _StringConst_122:
      000000108: 00000002                          # decimal = 2, ascii = "...."
      00000010C: 00000002                          # decimal = 2, ascii = "...."
      000000110: 20200000   store.h  8192(r0),r0   # offset = 0x2000
...
```

This command will recognize some machine instruction patterns and print them, not as the full machine instruction, but as a synthetic instruction that would have generated them.

For example:

These instructions…		***Will be printed as…***	
`ori`	`RegD,r0,immed`	`movi`	`RegD,immed`
`jalr`	`lr,0(Reg1)`	`callr`	`Reg1`
`jalr`	`r0,0(lr)`	`ret`	
`jal`	`r0,xxxx`	`jump`	`xxxx`
`jal`	`lr,xxxx`	`call`	`xxxx`
`ori`	`RegD,Reg1,0`	`mov`	`RegD,Reg1`
`xori`	`RegD,r0,immed`	`movi`	`RegD,immed`
`sub`	`RegD,r0,Reg2`	`neg`	`RegD,Reg2`
`xori`	`RegD,Reg1,-1`	`bitnot`	`RegD,Reg1`
`testeq`	`RegD,r0,Reg2`	`lognot`	`Regd,Reg2`
`addi`	`r0,r0,0`	`nop`	

d

This command is just like the "**dis**" command excepts that it does not ask for a starting address. Instead, this command just starts where the last "**dis**" or "**d**" command left off.

This command makes it easy to disassemble a lengthy block of code by just typing the "d" command repeatedly.

stack

This command displays the calling stack, which shows which functions are in execution. In the following example, the program has been stopped with a "debug" statement.

```
E> stack
   Function/Method              Execution at...          File
   =======================      ===================      ======================
   foo2                         ASSIGN     line 40       MyProgram.c
   foo1                         CALL       line 31       MyProgram.c
   main                         CALL       line 25       MyProgram.c
   _kplEntry                                             MyProgram.c
   _entry                                                runtime.s

--------------------  foo2  --------------------
Execution is stopped at ASSIGN on line 40 in function "foo2"   [MyProgram.c]
            Code Address: 00000cc4c
                   Frame: 00fffff80 - 00fffffc0,   size = 0x40 (decimal 64)
 arg offset  64 0x0040... 00fffffc0: 0000000000bc6105    myArg: int = 12345605
     offset  40 0x0028... 00fffffa8: 0000000000000457    myVarA: int = 1111
     offset  48 0x0030... 00fffffb0: fffffffffffc747d    myVarB: int = -232323
```

The command first shows the entire stack. The first highlighting shows the functions of interest and we can immediately see that execution is stoped in function **foo2**.

The second highlighting shows some information about the currently executing function: the address in memory of the machine instruction where execution is stopped and the values of the arguments and local variables to this function.

The command then prompts for a number, which is the number of frames the user would like to have printed in detail.

```
I can show you the frames of the callers. How many more frames would you like to
see (hit ENTER if none)?
```

Sometimes, the user will want nothing more, and will hit NEWLINE/ENTER/ RETURN:

But often the user may want to see more of the stack. In this case, the user might also be interested in functions foo1 and main, so the user would ask for 2 more frames.

```
I can show you the frames of the callers. How many more frames would you like to
see (hit ENTER if none)? 2


--------------------  foo1  --------------------
Execution is stopped at CALL on line 31 in function "foo1"   [MyProgram.c]
            Code Address: 00000cb9c
                   Frame: 00fffffc0 - 00fffffd0,   size = 0x10 (decimal 16)
 arg offset  16 0x0010... 00fffffd0: 0000000000bc6105    x: int = 12345605
```

```
--------------------  main  --------------------
Execution is stopped at CALL on line 25 in function "main"   [MyProgram.c]
            Code Address: 00000cb64
                   Frame: 00fffffd0 - 00ffffff0,   size = 0x20 (decimal 32)
     offset   8 0x0008... 00fffffd8: 0000000000bc6100    i: int = 12345600
     offset  16 0x0010... 00fffffe0: 0000000000000005    j: int = 5
I can show you the frames of the callers. How many more frames would you like to
see (hit ENTER if none)?
E>
```

From this, we can see where the functions are called and the values of the parameters and local variables.

In some programs, the stack can be deeper. Rather than counting how many frames to display, the user can type a ridiculously big number like 99 to just display the whole stack. This will include the frames for functions "**_kplEntry**" and "**_entry**", which are called as part of the start-up of any KPL program, but these can be ignored.

This command works roughly as follows:

The address of the next instruction to execute is in **PC**, the program counter register. From this, the debugging information which can be obtained from the executable file, can be used to determine which function this address is within. The debugging information will also contain information about the stack frame. The debugger then looks at the stack frame to find the return address for this function. Then the debugger determines which function this new address is within. Repeating the process, the debugger can then dissect the each frame on the stack.

This system works well, except in the case of a leaf function. A leaf function does not call any other functions and, as such, does not need to save the **Link Register** (**lr**) on the stack. This optimization can save a few instructions, but can also confuse this command. When it detects a leaf function, the **stack** command can use the **lr** register. The command will prompt before proceeding.

The other case when this system fails is when the execution is not within a known function. This can occur for two reasons.

First, the program has taken a bad branch and is not really executing any legal code.

To deal with this, the user can use the "**r**" ("**regs**") command to get the current instruction time. Then, the user can restart the program and use the "**stepn**"

command to get to a point in time shortly before this. Now execution is stopped slightly before the bad jump is to be taken. The "stack" command should work properly.

Second, the program may be executing legitimate code, but the debugging information is missing. The KPL compiler is diligent about including debugging information for all functions and methods. This debugging information is transmitted in the ".s" assembly file, using the following assembler directives:

.function
.endfunction
.stmt
.local

However, there may be functions coded directly in assembler for which these directives have not been added. More likely the problem is that the information in the directives is incorrect. In particular, a mistake in the "**framesize=**..." parameter is easy to make and will always confuse the debugger.

stack2

This command is essentially the same as the "**stack**" command, except that it prompts for the code address and the stack top pointer.

This command is might be useful in a multi-threaded application.

sm stackmem

This command prints the top few bytes of the stack, as a series of 32 bit words. For example:

```
E> sm
   ADDRESS  OFFSET
  00fffed0  0000:  00000000  <--- TOP
  00fffed4  0004:  0000cc38
```

```
  00ffffed8  0008:  00000000
  00ffffedc  000c:  00006a0c
  00ffffee0  0010:  00000000
  00ffffee4  0014:  00000228
  00ffffee8  0018:  00000000
  00ffffeec  001c:  000064e4
  00ffffef0  0020:  00000000
  00ffffef4  0024:  0000cc38
  00ffffef8  0028:  00000000
  00ffffefc  002c:  00050767
  00fffff00  0030:  00000000
  00fffff04  0034:  00000000
  00fffff08  0038:  00000000
  00fffff0c  003c:  00002988
E>
```

globals

This command uses the debugging information and displays the values of all global variables.

For example:

```
E> globals
From package "MyProgram.c"...
   line 91     000002880: 0000000000001e240    myGlob: int = 123456
   line 93     000002888: 48                   myGlobChar: byte = 'H'
                                                    (decimal 72)
   line 92     0000029c8: 00000000000028d0     myGlobVar2: String =
                                                    "this is some text"
From package "runtime.s"...
From package "HostInterface.c"...
   line 84     0000023c8: 00000000000023e0     stdin: ptr  -->
                                                    00000000 00000000 00000000
                                                    00000001 00000000 00000002...
   line 85     0000023d0: 00000000000023e8     stdout: ptr  -->
                                                    00000000 00000001 00000000
                                                    00000002 00000000 00018e88...
   line 86     0000023d8: 00000000000023f0     stderr: ptr  -->
                                                    00000000 00000002 00000000
                                                    00018e88 00000000 00018e40...
   line 87     0000023e0: 0000000000000000     stdinFILE: struct
   line 88     0000023e8: 0000000000000001     stdoutFILE: struct
   line 89     0000023f0: 0000000000000002     stderrFILE: struct
   line 61     0000023f8: 0000000000018e88     print: ptr  -->
                                                    1000101c 01000811 1dfffc12
                                                    14400108 22401188 14400108...
```

```
   line 62     000002400: 0000000000018e40    readString: ptr  -->
                                                   1d000012 1200020c 21000014
                                                   1a0000e0 01000811 14400108...
   line 90     0000029c0: 0000000000000000    errno: int = 0
From package "System.c"...
   line 40     000000008: 00                  alreadyInAlloc: bool = false
   line 37     000018f48: 05f5e10005f5e100    TheHeapArray: array
                                                   (currentSize = 100000000,
                                                    maxSize = 100000000)
   line 38     005f77050: 0000000000018f50    heapRegionStart: ptr  -->
                                                   00000000 00000000 00000000
                                                   00000000 00000000 00000000...
   line 38     005f77058: 0000000000018f50    heapNextPtr: ptr  -->
                                                   00000000 00000000 00000000
                                                   00000000 00000000 00000000...
   line 38     005f77060: 0000000005f77050    heapRegionBeyond: ptr  -->
                                                   00000000 00018f50 00000000
                                                   00018f50 00000000 05f77050...
   line 39     005f77068: 0000000000000000    heapTotalBytesFreed: int = 0
   line 39     005f77070: 0000000000000000    heapTotalAllocation: int = 0
   line 49     005f77078: 0000000000007230    mainThreadData: object
   line 50     005f770b8: 0000000005f770c8    threadPrefs_0: struct
   line 51     005f770c8: 0000000000000000    printPrefs_0: struct
From package "MiscLib.c"...
   line 21     000001c10: 0000000000000000    InputBuffer: array
                                                   (currentSize = 0,
                                                    maxSize = 0)
From package "PrintPackage.c"...
E>
```

This commands goes through each package and prints the global variables from that package. Recall that a "global variable" is one that occurs outside of any function or method and thus exists throughout the execution of the program.

The first package ("**MyProgram.c**") is repeated below:

```
From package "MyProgram.c"...
   line 91     000002880: 000000000001e240    myGlob: int = 123456
   line 93     000002888: 48                  myGlobChar: byte = 'H'
                                                   (decimal 72)
   line 92     0000029c8: 00000000000028d0    myGlobVar2: String =
                                                   "this is some text"
```

The command shows the line number on which each variable was declared, as well as the name, type, and current value of the variable. Other information includes the address at which the variable is stored as well as the value in hex.

trans

The Blitz core has a **Memory Management Unit** (**MMU**) that is active on every FETCH, LOAD, or STORE. Each access to main memory or to a memory-mapped I/O device is either a FETCH, LOAD, or STORE operation.

The MMU takes a "**program-generated address**" and maps it into a "**physical address**". A program-generated address can be any 36 bit number. A physical address is only 35 bits. In other words, program-generated addresses range over

0x0_0000_0000 … 0xF_FFFF_FFFF (64 GiBytes)

while physical addresses range over

0x0_0000_0000 … 0x7_FFFF_FFFF (32 GiBytes)

Program-generated addresses in the range

0x8_0000_0000 … 0xF_FFFF_FFFF (32 GiBytes)

are within a "**virtual address space**". Virtual address are mapped by the MMU using the TLB registers into a physical address.

The MMU will also check for errors. For example, if the core is in User Mode, only virtual addresses may be used. Also an attempt to STORE into a read-only page would be illegal. If the MMU detects any problems, an exception will be signaled, preventing the illegal access from occurring.

This command is used to see how an address would be translated by the Memory Management Unit (MMU).

Regardless of the outcome, the state of the core will not be altered. No actual access to the physical memory or memory-mapped I/O device will occur. Even if an exception is indicated, no exception is actually signaled.

In our first example, the core is in Kernel Mode and we are asking about fetching an instruction from a valid physical memory address.

This is a valid request and would not result in any exception. This result is highlighted below:

```
E> trans
=====  This command will translate a virtual address into a physical address.
=====  It will see what the MMU will do, making use of the current values in the
=====  TLB registers in the current core. The state of the core will not be
=====  changed.
  NOTE: The core is in kernel mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: 2134
You entered: Address = 0x2134  [ PHYSICAL ]
Want to perform fetch, load, or store? f
Output from MMU...
  Status=OKAY  Address = 0x2134  [ PHYSICAL ]
This address lies within...
    Private RAM,
    Shared RAM, or
    Boot ROM
E>
```

In the next example, the core is in User Mode. An attempt to fetch an instruction from the same address would cause an exception:

```
E> trans
=====  This command will translate a virtual address into a physical address.
=====  It will see what the MMU will do, making use of the current values in the
=====  TLB registers in the current core. The state of the core will not be
=====  changed.
  NOTE: The core is in user mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x0000.
Enter any address in hex: 2134
You entered: Address = 0x2134  [ PHYSICAL ]
Want to perform fetch, load, or store? f
  ----- This will cause a TLB_PRIVILEGE EXCEPTION
E>
```

In the next example, the core is in User Mode and the address in a virtual address. We are asking what would happen if we tried to LOAD a doubleword from memory.

There is a valid mapping in one of the TLB registers, so this access would succeed.

```
E> trans
=====  This command will translate a virtual address into a physical address.
=====  It will see what the MMU will do, making use of the current values in the
=====  TLB registers in the current core. The state of the core will not be
=====  changed.
```

```
  NOTE: The core is in user mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x1200.
Enter any address in hex: 8000140c8
You entered: Address = 0x8000140C8  [ VIRTUAL PAGE = 0x000005  OFFSET = 0x00c8 ]
Want to perform fetch, load, or store? 1
Enter the size of the access (1,2,4, or 8): 8
Output from MMU...
  Status=OKAY  Address = 0x1C0C8  [ PHYSICAL ]
E>
```

The command informs us that we are asking about the doubleword at offset 0x00c8 within page number 5. The MMU will translate this address into physical address 0x1C0C8.

The MMU will turn a virtual address into a physical address, but will not verify that there is actually anything installed at that address.[8] However, the emulator will give us a warning with any attempt to access memory in an area where no installed memory exists. This is illustrated next:

```
E> trans
=====  This command will translate a virtual address into a physical address.
=====  It will see what the MMU will do, making use of the current values in the
=====  TLB registers in the current core. The state of the core will not be
=====  changed.
  NOTE: The core is in kernel mode.
  NOTE: Interrupts are ENABLED.
  NOTE: The current ASID is 0x1200.
Enter any address in hex: 300000000
You entered: Address = 0x300000000  [ PHYSICAL ]
Want to perform fetch, load, or store? 1
Enter the size of the access (1,2,4, or 8): 8

***** Probable Error in the Blitz Code: Within PerformVirtualMapping,
                              the physical address is uninstalled/invalid *****
Output from MMU...
  Status=OKAY  Address = 0x300000000  [ PHYSICAL ]
E>
```

This same warning will appear if an executing program attempts to FETCH, LOAD, or STORE using a address that is within the range of installed physical memory or memory-mapped I/O devices.

[8] It is the responsibility of the OS kernel to use only properly installed physical memory to back virtual memory pages.

addr

Recall that a virtual address is a 35 bit number consisting of a 21 bit virtual page number following by a 14 bit byte offset within the page.

Given an address, breaking it into the 21 bit page number and 14 bit offset is tedious and error prone. This command makes it easy.

This command asks for an address and then does the work. For example:

```
E> addr
Enter an address in hex: 800301234
Address = 0x800301234  [ VIRTUAL    PAGE = 0x0000c0    OFFSET = 0x1234 ]
E>
```

Here's another example, in which the address is not a virtual address:

```
E> addr
Enter an address in hex: 1234
Address = 0x1234  [ PHYSICAL ]
E>
```

addr2

Combining a page number and offset into an address is error prone. This command makes it easy.

This command asks for a page number and offset and then combines them. For example:

```
E> addr2
Enter the 21 bit PAGE NUMBER: 0x0000c0
Enter the 14 bit OFFSET: 0x1234
AS PHYSICAL:
   Address = 0x301234  [ PHYSICAL ]
AS VIRTUAL:
   Address = 0x800301234  [ VIRTUAL    PAGE = 0x0000c0    OFFSET = 0x1234 ]
E>
```

read

In Blitz, all I/O devices are "memory-mapped", which means they are accessed with LOAD and STORE machine instructions.

This command asks for an address and then reads from that location as if a "**load.d**" instruction had been executed. It will then display the value returned by the I/O device.

The exact effect depends on the I/O device involved.

For example, with the **Simple Serial Device**, a LOAD from address 0x400104000 will cause the emulator to wait for the user to enter a single character. In this example, the user enters the letter "k", which is ASCII 0x6b.

```
E> read
Enter the (physical) address in hex: 400104000
Reading from 0x400104000...
k
Value = 0x000000000000006B   (decimal 107)
E>
```

write

In Blitz, all I/O devices are "memory-mapped", which means they are accessed with LOAD and STORE machine instructions.

This command asks for an address and then writes to that location as if a "**store.d**" instruction had been executed.

The exact effect depends on the I/O device involved.

For example, with the **Simple Serial Device**, a STORE to address 0x400104000 will cause the emulator to display a single character:

```
E> write
Enter the (physical) address in hex: 400104000
Enter the value to write, in hex: 6a
About to write value 0x000000000000006A to address 0x400104000...
j
Done.
E>
```

Note that ASCII code of 0x6a is the character “j”. The command ends by printing “\nDone.\n” which explains why the “j” is on a line by itself.

cores

The emulator is capable of emulating a multi-core processor.

The number of cores is determined by the file “**emulationParms**” which is read when the Blitz emulator starts up, or when the **reset** command is used.

In a multi-core systems, the cores are arranged in an array. This array can be 1-dimensional, in which case the cores are placed next to each other in a row and numbered from 0 to M-1, where M is the number of cores.

The cores can also be arranged in a 2-dimensional array. For example, assume the number of rows is N and the number of columns is M. Then there are a total of M × N cores.

Finally, the cores can be arranged in a 3-dimensional array. For example, assume the number of rows is N, the number of columns is M, and the number of planes is P. Then there are a total of M × N × P cores.

The intent of the array arrangement is to accommodate and model the actual physical placement on a silicon chip. While routing is not so much an issue with a dozen or so cores, in the future we can envision 100’s or 1000’s of cores on a single chip, at which time placement becomes important. At present, most chips are flat, but we are beginning to see the stacking of circuits, so a 3-D arrangement may be useful in the future.

Each core has a "**core number**". Let R be the total number of cores, i.e., M × N × P. Each core is assigned a number in the range 0 … R-1. Core 0 is said to be the "**primary core**".

The number of rows, columns, and planes is given in the file **emulatorParms**. For example, the file might contain these lines.

```
CORES_NUMBER_OF_COLS            0x0000000000000002 (decimal: 2)
CORES_NUMBER_OF_ROWS            0x0000000000000003 (decimal: 3)
CORES_NUMBER_OF_PLANES          0x0000000000000004 (decimal: 4)
```

Using the "**sim**" command, you can see these values. To alter them, you must edit the **emulatorParms** file and restart the emulator.

In this example, there are 24 cores (i.e., 2 × 3 × 4), numbered 0, 1, 2, … 23. At any one time there is a "**currently selected**" core.

In addition to the core number, each core has an **X-Y-Z coordinate** as well. In this example **X** ranges over 0 … 1, **Y** ranges over 0 … 2, and **Z** ranges over 0 … 3. Cores can communicate with their neighbors. We refer to directions using this notation[9]:

Direction	**Neighbor's X-Y-Z Coordinate**
west / left	X-1
east / right	X+1
north	Y-1
south	Y+1
up	Z-1
down	Z+1

The "**cores**" command prints out a line for each core, as shown next.

```
E> cores
             core     x,y,z     status  instructions      cycles        PC       interrupts   mode
             ====  ==========  =======  ============  ============  ==========  ==========  ======
Current -->    0   [ 0, 0, 0]  RUNNING         5887         17661  0x00000cd08              kernel <-- Current
               1   [ 1, 0, 0]  stopped            0             0  0x400000000   disabled   kernel
               2   [ 0, 1, 0]  stopped            0             0  0x400000000   disabled   kernel
```

[9] If the cores are linearly arranged, then core 0 (i.e., the core with coordinates X=0, Y=0, Z=0) is the **leftmost** core and core M (i.e., at X=M-1, Y=0, Z=0) is the **rightmost**. If the cores are arranged in a 2D rectangle, then core 0 (i.e., at X=0, Y=0, Z=0) is the most **northwestern** core and the core at X=M-1, Y=N-1, Z=0 is the most **southeastern** core. We reserve the use of "**up**" and "**down**" for 3D arrangements, and avoid these terms for 2D arrangements, since these terms are used for the third dimension. Using "**up**" and "**down**" for "**north**" and "**south**" in 2D arrangements is unambiguous, but it could be confusing.

```
   3   [ 1, 1, 0]  stopped          0            0    0x400000000   disabled   kernel
   4   [ 0, 2, 0]  stopped          0            0    0x400000000   disabled   kernel
   5   [ 1, 2, 0]  stopped          0            0    0x400000000   disabled   kernel
   6   [ 0, 0, 1]  stopped          0            0    0x400000000   disabled   kernel
   7   [ 1, 0, 1]  stopped          0            0    0x400000000   disabled   kernel
   8   [ 0, 1, 1]  stopped          0            0    0x400000000   disabled   kernel
   9   [ 1, 1, 1]  stopped          0            0    0x400000000   disabled   kernel
  10   [ 0, 2, 1]  stopped          0            0    0x400000000   disabled   kernel
  11   [ 1, 2, 1]  stopped          0            0    0x400000000   disabled   kernel
  12   [ 0, 0, 2]  stopped          0            0    0x400000000   disabled   kernel
  13   [ 1, 0, 2]  stopped          0            0    0x400000000   disabled   kernel
  14   [ 0, 1, 2]  stopped          0            0    0x400000000   disabled   kernel
  15   [ 1, 1, 2]  stopped          0            0    0x400000000   disabled   kernel
  16   [ 0, 2, 2]  stopped          0            0    0x400000000   disabled   kernel
  17   [ 1, 2, 2]  stopped          0            0    0x400000000   disabled   kernel
  18   [ 0, 0, 3]  stopped          0            0    0x400000000   disabled   kernel
  19   [ 1, 0, 3]  stopped          0            0    0x400000000   disabled   kernel
  20   [ 0, 1, 3]  stopped          0            0    0x400000000   disabled   kernel
  21   [ 1, 1, 3]  stopped          0            0    0x400000000   disabled   kernel
  22   [ 0, 2, 3]  stopped          0            0    0x400000000   disabled   kernel
  23   [ 1, 2, 3]  stopped          0            0    0x400000000   disabled   kernel
The number of runnable cores is: 1
The current core is: 0
E>
```

We see the **core number**, followed by the **X-Y-Z coordinates** of the core. We also see how many **instructions** have executed on each core. We also see the **Program Counter (PC)** and the status of the **InterruptsEnabled** and **KernelMode** bits in the CSR_STATUS register for each core.

The column marked "**status**" indicates whether the core is running or not. If a SLEEP1 instruction has been executed, the status will be "**sleep-1**" and if a SLEEP2 instruction has been executed, the status will be "**sleep-2**". Otherwise, the status will be "**RUNNING**" or "**stopped**".

We also see which core is the currently selected core.

sel

This command gives the user the opportunity to switch to a different core. Hitting NEWLINE / ENTER / RETURN is the usual response, leaving the currently selected core unchanged. For example:

```
E> sel
The current core is: 0
```

```
Enter the number of the core to make current (0..23) or ENTER for no change: 3
The current core is now: 3
E> cores
              core     x,y,z     status  instructions      cycles        PC       interrupts   mode
              ====  ==========  =======  ============  ============  ==========  ==========  ======
                0   [ 0, 0, 0]  RUNNING         5887         17661    0x00000cd08              kernel
                1   [ 1, 0, 0]  stopped            0             0    0x400000000   disabled   kernel
                2   [ 0, 1, 0]  stopped            0             0    0x400000000   disabled   kernel
Current -->     3   [ 1, 1, 0]  stopped            0             0    0x400000000   disabled   kernel <-- Current
                4   [ 0, 2, 0]  stopped            0             0    0x400000000   disabled   kernel
                5   [ 1, 2, 0]  stopped            0             0    0x400000000   disabled   kernel
                6   [ 0, 0, 1]  stopped            0             0    0x400000000   disabled   kernel
                7   [ 1, 0, 1]  stopped            0             0    0x400000000   disabled   kernel
                8   [ 0, 1, 1]  stopped            0             0    0x400000000   disabled   kernel
                9   [ 1, 1, 1]  stopped            0             0    0x400000000   disabled   kernel
               10   [ 0, 2, 1]  stopped            0             0    0x400000000   disabled   kernel
               11   [ 1, 2, 1]  stopped            0             0    0x400000000   disabled   kernel
               12   [ 0, 0, 2]  stopped            0             0    0x400000000   disabled   kernel
               13   [ 1, 0, 2]  stopped            0             0    0x400000000   disabled   kernel
               14   [ 0, 1, 2]  stopped            0             0    0x400000000   disabled   kernel
               15   [ 1, 1, 2]  stopped            0             0    0x400000000   disabled   kernel
               16   [ 0, 2, 2]  stopped            0             0    0x400000000   disabled   kernel
               17   [ 1, 2, 2]  stopped            0             0    0x400000000   disabled   kernel
               18   [ 0, 0, 3]  stopped            0             0    0x400000000   disabled   kernel
               19   [ 1, 0, 3]  stopped            0             0    0x400000000   disabled   kernel
               20   [ 0, 1, 3]  stopped            0             0    0x400000000   disabled   kernel
               21   [ 1, 1, 3]  stopped            0             0    0x400000000   disabled   kernel
               22   [ 0, 2, 3]  stopped            0             0    0x400000000   disabled   kernel
               23   [ 1, 2, 3]  stopped            0             0    0x400000000   disabled   kernel
The number of runnable cores is: 1
The current core is: 3
E>
```

<\n>

A null command — that is, hitting ENTER/RETURN without anything else — will result in an informative display. For example:

```
E> << ENTER >>
Number of running cores: 1
Currently selected: Core_3
Instruction executed so far: Core_3 = 0
                             total  = 5887
CURRENT LOCATION OF PC:
      400000000: 0300002F     mov     sp,r2         # synthetic for ORI _,_,0
E>
```

sched

When emuatoing a multicore system with more than one runnable core, the emulator — which is single threaded — will execute a few instructions on the

currently selected core. Then change the selected core to the next core that is runnable. The emulator will then execute a few instructions on that core. This process will continue, with each core getting a "timeslice", during which it executes "a few" instructions. The scheduling is strictly round-robin among the cores with status **RUNNING**. Execution will continue this way until either

The user hits control-C
Some core executes a DEBUG instruction
In the case of a limit (as in the **stepn** command), then limit is reached.
Some other error occurs

The question is, what does "a few" instructions mean? The answer is determined by the schedule.

This command allows the user to change the schedule. The command give the users choices. For example:

```
E> sched
    The "go" command will begin execution. The current core will execute "schedule[0]"
    instructions. Then the next core will execute "schedule[1]" instructions. This
    will continue with each core getting a timeslice. After the last entry in the
    "schedule" array is used, we loop back to "schedule[0]".
HERE IS THE CURRENT TIMESLICE SCHEDULE:
  schedule [0] = 329
  schedule [1] = 248
  schedule [2] = 149
  schedule [3] = 195
  schedule [4] = 260
  schedule [5] = 332
  schedule [6] = 147
  schedule [7] = 349
  schedule [8] = 431
  schedule [9] = 299
  schedule [10] = 469
  schedule [11] = 415
  schedule [12] = 128
  schedule [13] = 155
  schedule [14] = 50
  schedule [15] = 440
  schedule [16] = 505
Please choose by number...
   1 - Default schedule
   2 - Perfect interleaving 1-1-1-1...
   3 - Generate a new random schedule
   4 - Generate a new schedule where each timeslice is equal
Select a new schedule or ENTER for no change: 2
HERE IS THE NEW TIMESLICE SCHEDULE:
  schedule [0] = 1
  schedule [1] = 1
  schedule [2] = 1
  schedule [3] = 1
```

```
  schedule [4] = 1
  schedule [5] = 1
  schedule [6] = 1
  schedule [7] = 1
  schedule [8] = 1
  schedule [9] = 1
  schedule [10] = 1
  schedule [11] = 1
  schedule [12] = 1
  schedule [13] = 1
  schedule [14] = 1
  schedule [15] = 1
  schedule [16] = 1
E>
```

In the above example, the user has selected option 2, which is "Perfect Interleaving". This means that each core will execute one instruction. Then the emulator will move to the next core. The cores will effectively operate in lockstep, proceeding at exactly the same speed.

The default time schedule, which was shown first, gives every core between 50 and 505 instructions for its timeslice. There are 17 slots in the schedule with 17 values.[10] The first core to run will get 329 instructions. The second core will get 248 instructions. This will continue. At each timeslice, the emulator will move to the next core and to the next slot in the schedule. When the end of the schedule (i.e., the seventeenth entry) has been reached, the emulator will loop back to the first entry. Assuming that the number of cores is not 17 or a multiple thereof, this will eventually give every core time slices of every one of the 17 sizes. Thus, in the long term, all cores will execute the same speed, although there will be local variations.

This command also allows the user to change the timeslice, giving all processor the same number of instructions. For example:

```
E> sched
    The "go" command will    ...etc...
HERE IS THE CURRENT TIMESLICE SCHEDULE:
  schedule [0] = 1
  schedule [1] = 1
  schedule [2] = 1
    ...etc...
  schedule [14] = 1
  schedule [15] = 1
  schedule [16] = 1
Please choose by number...
   1 - Default schedule
   2 - Perfect interleaving 1-1-1-1...
```

[10] These 17 random values were pre-chosen and are fixed; changing them would require recompiling the emulator itself.

```
   3 - Generate a new random schedule
   4 - Generate a new schedule where each timeslice is equal
Select a new schedule or ENTER for no change: 4
Enter the size of each timeslice: 800
HERE IS THE NEW TIMESLICE SCHEDULE:
  schedule [0] = 800
  schedule [1] = 800
  schedule [2] = 800
    ...etc...
  schedule [14] = 800
  schedule [15] = 800
  schedule [16] = 800
E>
```

In addition, there is an option for mixing things up. For example:

```
E> sched
    The "go" command will    ...etc...
HERE IS THE CURRENT TIMESLICE SCHEDULE:
  schedule [0] = 800
  schedule [1] = 800
  schedule [2] = 800
    ...etc...
  schedule [14] = 800
  schedule [15] = 800
  schedule [16] = 800
Please choose by number...
   1 - Default schedule
   2 - Perfect interleaving 1-1-1-1...
   3 - Generate a new random schedule
   4 - Generate a new schedule where each timeslice is equal
Select a new schedule or ENTER for no change: 3
Enter the maximum timeslice size: 100000
HERE IS THE NEW TIMESLICE SCHEDULE:
  schedule [0] = 31730
  schedule [1] = 78841
  schedule [2] = 42613
  schedule [3] = 44304
  schedule [4] = 33170
  schedule [5] = 17710
  schedule [6] = 97158
  schedule [7] = 29561
  schedule [8] = 70934
  schedule [9] = 93100
  schedule [10] = 80279
  schedule [11] = 51817
  schedule [12] = 95336
  schedule [13] = 99098
  schedule [14] = 7827
  schedule [15] = 13513
  schedule [16] = 29268
E>
```

startall

This command will make all cores runnable. In particular, it will change the status of any core that is **STOPPED** to **RUNNING**. For example:

```
E> startall
Core 1 is now RUNNING
Core 2 is now RUNNING
Core 3 is now RUNNING
Core 4 is now RUNNING
Core 5 is now RUNNING
Core 6 is now RUNNING
Core 7 is now RUNNING
Core 8 is now RUNNING
Core 9 is now RUNNING
Core 10 is now RUNNING
Core 11 is now RUNNING
Core 12 is now RUNNING
Core 13 is now RUNNING
Core 14 is now RUNNING
Core 15 is now RUNNING
Core 16 is now RUNNING
Core 17 is now RUNNING
Core 18 is now RUNNING
Core 19 is now RUNNING
Core 20 is now RUNNING
Core 21 is now RUNNING
Core 22 is now RUNNING
Core 23 is now RUNNING
E>
```

stopall

This command will stop all cores. In particular, it will change the status of any core that is **RUNNING** to **STOPPED**. For example:

```
E> stopall
Core 0 is now STOPPED
Core 1 is now STOPPED
Core 2 is now STOPPED
Core 3 is now STOPPED
Core 4 is now STOPPED
Core 5 is now STOPPED
```

```
Core 6 is now STOPPED
Core 7 is now STOPPED
Core 8 is now STOPPED
Core 9 is now STOPPED
Core 10 is now STOPPED
Core 11 is now STOPPED
Core 12 is now STOPPED
Core 13 is now STOPPED
Core 14 is now STOPPED
Core 15 is now STOPPED
Core 16 is now STOPPED
Core 17 is now STOPPED
Core 18 is now STOPPED
Core 19 is now STOPPED
Core 20 is now STOPPED
Core 21 is now STOPPED
Core 22 is now STOPPED
Core 23 is now STOPPED
E>
```

start

This command can be used to change one or more cores from **STOPPED** to **RUNNING**. It prompts the user to select the core by number, as shown in this example:

```
E> start
This command allows you to resume a core's execution by changing its status to RUNNING.
Enter the number of a core (or ENTER to exit): 5
Core 5 is now RUNNING.
Enter the number of a core (or ENTER to exit): 6
Core 6 is now RUNNING.
Enter the number of a core (or ENTER to exit): 9
Core 9 is now RUNNING.
Enter the number of a core (or ENTER to exit): << ENTER >>
E>
```

To verify the result of this, we can use the cores command to see which cores are now RUNNING. These are highlighted below. This command does not alter which core is “currently selected”, as you can see.

```
E> cores
              core     x,y,z     status  instructions      cycles        PC       interrupts   mode
              ====  ==========  =======  ============  ============  ==========  ==========  ======
                0   [ 0, 0, 0]  stopped         5887         17661    0x00000cd08              kernel
                1   [ 1, 0, 0]  stopped            0             0    0x400000000   disabled   kernel
                2   [ 0, 1, 0]  stopped            0             0    0x400000000   disabled   kernel
Current -->     3   [ 1, 1, 0]  stopped            0             0    0x400000000   disabled   kernel <-- Current
                4   [ 0, 2, 0]  stopped            0             0    0x400000000   disabled   kernel
```

```
                5   [ 1, 2, 0]  RUNNING          0              0    0x400000000   disabled   kernel
                6   [ 0, 0, 1]  RUNNING          0              0    0x400000000   disabled   kernel
                7   [ 1, 0, 1]  stopped          0              0    0x400000000   disabled   kernel
                8   [ 0, 1, 1]  stopped          0              0    0x400000000   disabled   kernel
                9   [ 1, 1, 1]  RUNNING          0              0    0x400000000   disabled   kernel
               10   [ 0, 2, 1]  stopped          0              0    0x400000000   disabled   kernel
               11   [ 1, 2, 1]  stopped          0              0    0x400000000   disabled   kernel
               12   [ 0, 0, 2]  stopped          0              0    0x400000000   disabled   kernel
               13   [ 1, 0, 2]  stopped          0              0    0x400000000   disabled   kernel
               14   [ 0, 1, 2]  stopped          0              0    0x400000000   disabled   kernel
               15   [ 1, 1, 2]  stopped          0              0    0x400000000   disabled   kernel
               16   [ 0, 2, 2]  stopped          0              0    0x400000000   disabled   kernel
               17   [ 1, 2, 2]  stopped          0              0    0x400000000   disabled   kernel
               18   [ 0, 0, 3]  stopped          0              0    0x400000000   disabled   kernel
               19   [ 1, 0, 3]  stopped          0              0    0x400000000   disabled   kernel
               20   [ 0, 1, 3]  stopped          0              0    0x400000000   disabled   kernel
               21   [ 1, 1, 3]  stopped          0              0    0x400000000   disabled   kernel
               22   [ 0, 2, 3]  stopped          0              0    0x400000000   disabled   kernel
               23   [ 1, 2, 3]  stopped          0              0    0x400000000   disabled   kernel
The number of runnable cores is: 3
The current core is: 3
E>
```

stop

This command can be used to change one or more cores from **RUNNING** to **STOPPED**. It prompts the user to select the core by number, as shown in this example. As you can see, an attempt to stop a core that is not running will get a message.

```
E> stop
This command allows you to freeze a core's execution by changing its status to STOPPED.
Enter the number of a core (or ENTER to exit): 6
Core 6 is now STOPPED.
Enter the number of a core (or ENTER to exit): 8
Core 8 is already STOPPED!
Enter the number of a core (or ENTER to exit): 9
Core 9 is now STOPPED.
Enter the number of a core (or ENTER to exit): << ENTER >>
E>
```

symbols

Executable (.exe) files normally contain debugging information. This includes a number of symbols. This command lists all known symbols. It lists all symbols twice. The first list is sorted by numerical value. The second is sorted alphabetically.

For example:

```
E> symbols
Symbols (ordered numerically):
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===============  =====  ============================
  KPL_Compiler_Path_Testing_Symbol               0                0          114 runtime.s
  _GlobalVar_alreadyInAlloc                      8                8  LABEL   175 System.s
  OFFSET_OF_memAllocFun                         10               16         1784 runtime.s
  _StringConst_136                              10               16  LABEL   182 System.s
  OFFSET_OF_memFreeFun                          18               24         1814 runtime.s
  _StringConst_135                              20               32  LABEL   187 System.s
     ...etc...
  Serial_String_Addr                     400104018       17180934168         2473 runtime.s
  Serial_String_In_Len                   400104020       17180934176         2474 runtime.s
  Serial_String_Out_Len                  400104028       17180934184         2475 runtime.s
Symbols (ordered alphabetically):
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===============  =====  ============================
  BadNumberString                          18D08             101640  LABEL  1951 runtime.s
  BadNumberString_1                        18D10             101648  LABEL  1954 runtime.s
  BadNumberString_x                        18D48             101704  LABEL  1956 runtime.s
  BadTimerHandler                          187FC             100348  LABEL   842 runtime.s
     ...etc...
  stEq_loopD                               18B78             101240  LABEL  1373 runtime.s
  stEq_loopD_x                             18B98             101272  LABEL  1381 runtime.s
  stEq_retF                                18BC8             101320  LABEL  1403 runtime.s
  stEq_retT                                18BB8             101304  LABEL  1396 runtime.s
  startupMessage                           18898             100504  LABEL   917 runtime.s
  strEqual                                 18B54             101204  LABEL  1357 runtime.s
E>
```

For programs compiled with the KPL compiler which include packages like **System** and **PrintPackage**, there are a large number of symbols, making this command less useful.

dinfo

This command displays all the debugging information from the executable file.

Note: This command produces a flood of information and is not very useful.

This command was used in debugging the emulator.

find

In order to find the value of a symbol, this command can be used. It prompts the user to enter the symbol, or at least the first few characters of the symbol. It prints all symbols that begin with the same characters. For example:

```
E> find
Enter the first few characters of the symbol; all matching will be printed: Pri
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===================  =====  ============================
  PrintBool                              18EE8                102120  LABEL  2392 runtime.s
  PrintBoolStr                           18F00                102144  LABEL  2400 runtime.s
  PrintCSRPrevPCAndHalt                  18198                 98712  LABEL   515 runtime.s
  PrintFalse                             18EFC                102140  LABEL  2398 runtime.s
  PrintRuntimeError                      18850                100432  LABEL   881 runtime.s
E>
```

Case is significant.

Helpful Trick: To get a list of all private functions, recall that the KPL compiler attaches a prefix to the names of all private functions. So enter "_fun" as the search pattern:

```
E> find
Enter the first few characters of the symbol; all matching will be printed: _fun
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===================  =====  ============================
  _function_10_RandomNumber               CF14                 53012  LABEL   856 MyProgram.s
  _function_11_foo4                       CEE4                 52964  LABEL   830 MyProgram.s
  _function_12_foo3                       CD28                 52520  LABEL   650 MyProgram.s
  _function_137_printClassNameFromDPT     6B68                 27496  LABEL  8567 System.s
  _function_138_printClassNameOfObject    6A9C                 27292  LABEL  8448 System.s
  _function_139_invokeDebugger            6A38                 27192  LABEL  8387 System.s
  _function_13_foo2                       CC68                 52328  LABEL   549 MyProgram.s
  _function_140_KPLDefaultFatalErrorFunction  5904             22788  LABEL  6144 System.s
  _function_141_KPLMemoryFree_Version1    344C                 13388  LABEL  1642 System.s
  _function_142_KPLMemoryAlloc_Version1   3340                 13120  LABEL  1524 System.s
  _function_143_KPLMemoryFree_Default     3310                 13072  LABEL  1492 System.s
  _function_144_KPLMemoryAlloc_Default    327C                 12924  LABEL  1433 System.s
  _function_14_foo1                       CC44                 52292  LABEL   524 MyProgram.s
  _function_19_hostDateNext               BEFC                 48892  LABEL  2299 HostInterface.s
  _function_20_hostDateSize               BED4                 48852  LABEL  2277 HostInterface.s
  _function_21_argumentNext               BD60                 48480  LABEL  2106 HostInterface.s
  _function_22_argumentSize               BD3C                 48444  LABEL  2086 HostInterface.s
  _function_26_LocalPrintString           7650                 30288  LABEL   521 PrintPackage.s
  _function_27_LocalPrintChar             7604                 30212  LABEL   475 PrintPackage.s
E>
```

This same trick can be used to find where all the private globals are located:

```
E> find
Enter the first few characters of the symbol; all matching will be printed: _Glob
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===================  =====  ============================
  _GlobalVar_TheHeapArray                 18F48                102216  LABEL   867 System.s
  _GlobalVar_alreadyInAlloc                  8                     8  LABEL   175 System.s
  _GlobalVar_heapNextPtr               5F77058             100102232  LABEL   873 System.s
  _GlobalVar_heapRegionBeyond          5F77060             100102240  LABEL   876 System.s
  _GlobalVar_heapRegionStart           5F77050             100102224  LABEL   870 System.s
  _GlobalVar_heapTotalAllocation       5F77070             100102256  LABEL   882 System.s
  _GlobalVar_heapTotalBytesFreed       5F77068             100102248  LABEL   879 System.s
  _GlobalVar_mainThreadData            5F77078             100102264  LABEL   885 System.s
  _GlobalVar_myGlob                       2880                 10368  LABEL   203 MyProgram.s
  _GlobalVar_myGlobChar                   2890                 10384  LABEL   209 MyProgram.s
  _GlobalVar_myGlobVar2                   29F8                 10744  LABEL   266 MyProgram.s
  _GlobalVar_print                        23F8                  9208  LABEL    93 HostInterface.s
  _GlobalVar_printPrefs_0              5F770C8             100102344  LABEL   891 System.s
  _GlobalVar_randomSeed                   2888                 10376  LABEL   206 MyProgram.s
  _GlobalVar_readString                   2400                  9216  LABEL    96 HostInterface.s
  _GlobalVar_threadPrefs_0             5F770B8             100102328  LABEL   888 System.s
E>
```

To get a list of all public things[11] in a package, use a search string that begins with "_P_" followed by the package name:

```
E> find
Enter the first few characters of the symbol; all matching will be printed: _P_Mis
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===============  =====  ============================
  _P_MiscLib_AppendIntToString            C760                51040  LABEL  1061 MiscLib.s
  _P_MiscLib_GetInputLine                 C500                50432  LABEL   773 MiscLib.s
  _P_MiscLib_GetInt                       C474                50292  LABEL   684 MiscLib.s
  _P_MiscLib_GetOneChar                   C3EC                50156  LABEL   609 MiscLib.s
  _P_MiscLib_GetYesNo                     C2C8                49864  LABEL   438 MiscLib.s
  _P_MiscLib_Indent                       C7F8                51192  LABEL  1141 MiscLib.s
  _P_MiscLib_InputBuffer                  1C10                 7184  LABEL   311 MiscLib.s
  _P_MiscLib_PadTo                        C86C                51308  LABEL  1205 MiscLib.s
E>
```

find2

This command allows you to look up a symbol given its value. For example:

```
E> find2
Enter a value in hex: 18f00
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===============  =====  ============================
  PrintBoolStr                          18F00               102144  LABEL  2400 runtime.s
E> find2
Enter a value in hex: 18f01
***** There is no symbol with that value. (The next largest value is 0x18f10 (decimal 102160). *****
  Symbol                          Value (hex)       Value (decimal)  Label  Source line number / filename
  ==============================  ================  ===============  =====  ============================
  _TrueMsg                              18F10               102160  LABEL  2405 runtime.s
E>
```

This command was useful in debugging the emulator.

where

This command allows you to see where the currently selected core is stopped. To get the current location, just respond to the prompt with ENTER.

```
E> where
Enter an address in hex (or 0 for current PC): << ENTER >>
CURRENT LOCATION OF PC:
  ASSIGN on line 76 in function "foo2"   [MyProgram.c]
```

[11] This includes public functions, public global variables, public classes, etc.

```
      00000CD08: 1E0040F7       load.d     r7,64(sp)           # offset = 0x40
E>
```

If the address is within a function coded in KPL, the debugger will indicate the type of statement (In this example, it's an assignment statement) and function name ("foo2"), as well as the source filename and line number where that statement is located (line 76 within "MyProgram.c").

You can also enter a specific address to learn in which function it resides. For example:

```
E> where
Enter an address in hex (or 0 for current PC): c000
  ASSIGN on line 671 in function "hostDate"   [HostInterface.c]
      00000C000: 01FFFF77       addi       r7,r7,-1
E>
```

If the address is not in anything that the debugger knows about, it will just give the contents of memory at that location. For example:

```
E> where
Enter an address in hex (or 0 for current PC): 100000
      000100000: 00000000
E>
```

At other times you might ask for the address of some variable. Here, we see the name of the variable and an indication of its value in hex, decimal, and ASCII.

```
E> where
Enter an address in hex (or 0 for current PC): 18F10
   Within Function "PrintBool"   [runtime.s]
    _TrueMsg:
      000018F10: 54525545             # decimal = 1414681925, ascii = "TRUE"
E>
```

g go

This command will start execution. The emulator will begin executing machine instructions and this will continue until

- Command-C is pressed

- A DEBUG or BREAKPOINT instruction is executed
- A SLEEP1 or SLEEP2 instruction is executed
- An error occurs or a warning is printed
- A "watched" location is stored into

After one of these events occurs, the emulator will re-enter command mode and you can use the various commands to examine the state. The SLEEP2 instruction will terminate the emulator itself.

If there are multiple cores, then time slicing will occur, and each core will have a turn executing, until one of the above events happens on any running core. After execution halts, the core that was last running will become the "currently selected" core.

If "auto-go" is enabled (i.e., the "**-g**" command line option was used), then execution will commence immediately upon startup of the emulator or execution of the "**rerun**" command, as if the "**go**" command had been entered.

s step

The "**step**" command is similar to the "**go**" command, except that only one instruction will be executed.

This command will print the machine instruction before it is executed. Here is an example. After two instructions are executed (see highlighting below), the user hits ENTER to see where in the program execution is located (see highlighting below).

```
E> s
Executing this instruction:
      00000CC58: 220000F0       store.d    0(sp),r0
Instruction executed so far: Core_0 = 2812
                             total  = 2812
E> s
Executing this instruction:
      00000CC5C: 1E0010F1       load.d     r1,16(sp)           # offset = 0x10
Instruction executed so far: Core_0 = 2813
                             total  = 2813
E> << ENTER >>
Number of running cores: 1
Currently selected: Core_0
```

```
Instruction executed so far: Core_0 = 2813
                             total  = 2813
CURRENT LOCATION OF PC:
  CALL on line 65 in function "foo1"   [MyProgram.c]
      00000CC60: 1900010E       call       _function_13_foo2          # PC + 0x10
E>
```

n stepn

This command prompts for a number and then executes that number of statements. Either "**stepn**" or the abbreviation "**n**" can be used. For example:

```
E> stepn
Enter the number of instructions to execute (in decimal): 100
Beginning execution...
Done!
E> n
Enter the number of instructions to execute (in decimal): 23
Beginning execution...
Done!
E> << ENTER >>
Number of running cores: 1
Currently selected: Core_0
Instruction executed so far: Core_0 = 123
                             total  = 123
CURRENT LOCATION OF PC:
    _CheckVersion_P_System_:
      000002D28: 22FFFEF8       store.d    -8(sp),lr           # offset = 0xFFF8
E>
```

If multiple cores are runnable and the number entered is large enough, timeslicing will occur and all cores will make progress.

Execution may halt prematurely, under the same circumstances that would halt execution for the "**go**" command. This includes errors and instructions like DEBUG.

t

The "**t**" command is similar to the "**go**" command in that it begins execution. Here are the differences:

Execution will halt after…
- A CALL instruction is executed.
- A RETURN instruction is executed.
- A SYSRET instruction is executed.
- A SYSCALL instruction is executed[12]
- An exception occurs.

Only the currently selected core will run.

Here is an example:

```
E> t
   Within Function "foo1"   [MyProgram.c]
    _function_12_foo1:
      00000CC2C: 22FFFEF8       store.d    -8(sp),lr           # offset = 0xFFF8
Instr count = 2808
E> t
   Within Function "foo2"   [MyProgram.c]
    _function_11_foo2:
      00000CC50: 22FFFEF8       store.d    -8(sp),lr           # offset = 0xFFF8
Instr count = 2814
E> t
  RETURN on line 65 in function "foo2"   [MyProgram.c]
      00000CC78: 1A0000E0       ret                 # synthetic for JALR r0,0(lr)
Instr count = 2825
E> t
  RETURN on line 47 in function "foo1"   [MyProgram.c]
      00000CC4C: 1A0000E0       ret                 # synthetic for JALR r0,0(lr)
Instr count = 2828
E> t
```

This instruction allows the programmer to jump through the execution of the program at a much faster rate than with the "**step**" command.

[12] The SYSCALL instruction causes an exception, and exceptions cause a halt.

Debugging Trick: How to Back Up Execution

Assume you want to back up the program's execution. That is, you want to execute in reverse, undoing the execution of one or more instructions, until you get to some particular point in the past. How can this be done?

The trick is to restart the program from the beginning and execute instructions up to the point in time you want.

It is helpful that commands like "**t**" displays the instruction count. The instruction count is the number of machine instructions that have been executed so far and we can use this number to identify when in the past we want to reach.

To get to a point in the past, we can simply issue a "**reset**" command and then use the "**stepn**" command to get to the desired point.[13]

For example, looking back at the previous example, let's assume we have gone a little too far. We wish we had stopped right before the return from "**foo2**" and so we could have looked at the local variables of "**foo2**" directly before the return. We see that the return from "**foo2**" happened at 2825; this is our target.[14]

KPL always ends a function with 3 instructions (restore the stack, restore register **lr**, and the RET instruction).

So compute 2825 - 3 to get 2822.

First we issue the "**reset**" command, which restarts the emulator.

```
E> reset
Resetting all processor state and re-reading file "MyProgram.exe"...
Reading executable file...
The executable file (MyProgram.exe) was loaded. The _entry address (0x00001885C)
was loaded into register r6.
E>
```

[13] This assumes that only one core is running. If you are emulating a multicore system and timeslicing is occurring, this must be taken into account. The "**t**" command disables timeslicing while the "**stepn**" command does not, so the instruction counts may be different.

[14] The "t" command stops *after* the CALL or RET instruction is executed. So 2825 is the count after the RET.

Then we issue an "**stepn**" command (abbreviation: "**n**"), with the computed count.

```
E> stepn
Enter the number of instructions to execute (in decimal): 2822
Beginning execution...
MyProgram running...
Done!
E>
```

Then we hit ENTER to see where we are: the return within "**foo2**". We see the ADDI instruction, which adjusts to the stack pointer, is next so this confirms that we are were we want to be.

```
E> << ENTER >>
Number of running cores: 1
Currently selected: Core_0
Instruction executed so far: Core_0 = 2822
                             total  = 2822
CURRENT LOCATION OF PC:
  RETURN on line 65 in function "foo2"   [MyProgram.c]
      00000CC70: 010008FF       addi        sp,sp,8
E>
```

Then the "**stack**" command can be used to view the local variables in "**foo2**." In this function, there is only one, named "**myArg**".

```
E> stack
   Function/Method                Execution at...        File
   ============================   ====================   ============================
   foo2                           RETURN     line 65     MyProgram.c
   foo1                           CALL       line 46     MyProgram.c
   main                           CALL       line 37     MyProgram.c
   _kplEntry                                             MyProgram.c
   _entry                                                runtime.s

--------------------  foo2  --------------------
Execution is stopped at RETURN on line 65 in function "foo2"   [MyProgram.c]
            Code Address: 00000cc6c
                   Frame: 00fffff98 - 00fffffa0,   size = 0x8 (decimal 8)
 arg offset   8 0x0008... 00fffffa0: 0000000000000000    myArg: int = 0
I can show you the frames of the callers. How many more frames would you like to
see (hit ENTER if none)? << ENTER >>
E>
```

Finally we might use the "**step**" command to execute the final 3 instructions of the function:

```
E> s
Executing this instruction:
```

```
      00000CC70: 010008FF       addi       sp,sp,8
Instruction executed so far: Core_0 = 2823
                             total  = 2823
E> s
Executing this instruction:
      00000CC74: 1EFFF8FE       load.d     lr,-8(sp)           # offset = 0xFFF8
Instruction executed so far: Core_0 = 2824
                             total  = 2824
E> s
Executing this instruction:
      00000CC78: 1A0000E0       ret                 # synthetic for JALR r0,0(lr)
Instruction executed so far: Core_0 = 2825
                             total  = 2825
E>
```

watch

The "watch" command is used to stop execution whenever a particular address in memory is stored into.

In this example, we first use the "**globals**" command to learn the address of a variable called "**myGlob**". The address is highlighted below:

```
E> globals
From package "MyProgram.c"...
   line 100    000002880: ffffffffffffffff  myGlob: int = -1
   line 115    000002888: 1234567890abcdef  randomSeed: int = 1311768467294899695
   line 102    000002890: 48                myGlobChar: byte = 'H' (decimal 72)
   line 5      0000029d0: 0000000000000000  MyGlobal: int = 0
   line 101    0000029d8: 0000000000000000  myGlobVar2: String = null
From package "runtime.s"...
From package "HostInterface.c"...
   line 84     0000023c8: 00000000000023e0  stdin: ptr  --> 00000000 00000000 00000000
                                                            00000001 00000000 00000002...
   line 85     0000023d0: 00000000000023e8  stdout: ptr --> 00000000 00000001 00000000
                                                            00000002 00000000 00018e88...
  ... etc ...
E>
```

Next, we use the "**watch**" command and provide this address:

```
E> watch
```

```
  Execution will halt whenever this address is stored into.
  Enter 0 to display the previous watch address.
  Enter -1 to cancel a previous watch address.
Enter the address in hex: 000002880
Execution will halt whenever address 0x000002880 is stored into.
E>
```

Then we issue the "**go**" command to begin execution. The 2796-th instruction to be executed was a STORE into this location. We also see the value that was stored.

```
E> g
Beginning execution...
MyProgram running...

**********  The value 0x000000000001e240 was stored into the 'watched'
                address (000002880) at instr time = 2796  **********
Done!
E>
```

We can then use the "**hex**" command to interpret this value as a decimal number, a series of ASCII codes, and as a double precision floating point number. The variable "**myGlob**" had type **int** as we saw earlier, so we can focus on the decimal value.

```
E> hex
Enter a value in hex: 0x000000000001e240
     hex: 0x000000000001E240   >120 KiBytes
     decimal: 123456
     ascii: ".......@"
     real: 6.0995368372976933335786247689e-319
E>
```

To determine were in the source code this occurred, we just hit ENTER. Since execution is stopped, about to execute the instruction following the STORE, we are likely stopped on the line just after the update to the variable:

```
E> << ENTER >>
Number of running cores: 1
Currently selected: Core_0
Instruction executed so far: Core_0 = 2796
                             total  = 2796
CURRENT LOCATION OF PC:
  RETURN on line 108 in function "foo4"   [MyProgram.c]
      00000CE54: 04006406       movi       r6,100        # synthetic for XORI r6,r0,0x64
E>
```

This command will allow only one "watched" location at a time.[15] The watch remains in effect until cancelled.

The user can hit ENTER to see which address is being watched and -1 to cancel a watch.

```
E> watch
  Execution will halt whenever this address is stored into.
  Enter 0 to display the previous watch address.
  Enter -1 to cancel a previous watch address.
Enter the address in hex: << ENTER >>
Execution will halt whenever address 0x000002880 is stored into.
E> watch
  Execution will halt whenever this address is stored into.
  Enter 0 to display the previous watch address.
  Enter -1 to cancel a previous watch address.
Enter the address in hex: -1
From now on, no address is being watched.
E> watch
  Execution will halt whenever this address is stored into.
  Enter 0 to display the previous watch address.
  Enter -1 to cancel a previous watch address.
Enter the address in hex: << ENTER >>
There is no current watch address.
E>
```

The watched address is the address of a byte and need not be aligned. Any STORE — whether byte, halfword, word, or doubleword — that includes the watched byte will trigger a halt.

reset

This command will reset the emulator as if the user had executed a "**quit**" command and then restarted the emulator.

The "**reset**" command will:

- If the BootROM has been updated,

[15] We can certainly imagine a debugger that allows multiple locations to be watched simultaneously, but in practice, one at a time is adequate. In fact, the "**watch**" command is used quite rarely.

ask about updating the "**emulationROM**" file
- If the SecureStorage has been updated,
  ask about updating the "**emulationSecure**" file
- Re-read the "**emulationParms**" file
- Reset the ROM from the "**emulationROM**" file
- Reset the SecureStorage from the "**emulationSecure**" file
- Reset all processors the their initial state
  Clear all registers, clear private and shared memory
- Reset the instruction counters and cycle clocks
- Reset I/O devices
- Re-read the executable file and the debugging information
- Reset the multicore timeslice scheduling

This command will always leave the emulator in "command mode". It will not start execution.

rerun

The "**rerun**" command is equivalent to executing a "**reset**" command to completely reset the processor state, followed by the "**go**" command to start execution from beginning.

In the following example, we load and begin execution of a program. The program prints a message (see highlighting) and then stops after encountering a DEBUG instruction.

```
Shell% blitz MyProgram.exe -g -nowarn
MyProgram running...

****  A DEBUG machine instruction was executed  *****

Next instruction to execute:
   DEBUG (line 38)
   ----------  ################# here #################
      00000CBDC: 00280000       debug

Entering machine-level debugger...
=========================================
=====                                 =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III      =====
```

```
=====           6 May 2021            =====
=====                                 =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E>
```

In a separate window (not shown) we edit and recompile our program. We changed the message. Then we use the "**rerun**" command to reset the processor state and re-run the program. We can see that the modified program is now executed and the message is now different:

```
E> rerun
MyProgram running... HELLO WORLD!

****  A DEBUG machine instruction was executed  *****

Next instruction to execute:
   DEBUG (line 38)
   ----------  ################# here #################
      00000CBEC: 00280000       debug
E>
```

The above example illustrates the use of this command in a common approach to debugging:

Write and run a new program.
REPEAT
    See a problem.
        [Use debugging commands to explore the state after an error is reported.]
    Edit the program.
    Recompile the program.
    Re-run the program (use the "**rerun**" command).
UNTIL program works[16]

[16] The actual expression is (programWorks | sleepy | hungry). A REPEAT-UNTIL is more appropriate than a WHILE, since the loop will always iterate at least once.

hex
dec
ascii

These three commands exist for convenience and have nothing to do with debugging or emulation. It is just handy to be able to convert from hex into other forms.

The "**hex**" command asks for a value to be entered in hex. The "**dec**" command asks for a value to be entered in decimal. The "**ASCII**" command asks for a single character to be entered.

All commands do the same thing. The input is converted to a 64 bit value, which is then displayed in hex, decimal, ascii, and as a double precision value.

A hex value can be entered in upper or lowercase and the "0x" prefix is optional. Also, a negative sign is allowed, as in the highlighted line:

```
E> hex
Enter a value in hex: 1e240
     hex: 0x000000000001E240   >120 KiBytes
     decimal: 123456
     ascii: ".......@"
     real: 6.0995368372976933335786247689e-319
E> hex
Enter a value in hex: 0x000000000001E240
     hex: 0x000000000001E240   >120 KiBytes
     decimal: 123456
     ascii: ".......@"
     real: 6.0995368372976933335786247689e-319
E> hex
Enter a value in hex: -abc
     hex: 0xFFFFFFFFFFFFF544   (-0xabc)
     decimal: -2748
     ascii: ".......D"
     real: nan
E>
```

The "**dec**" command is mostly useful for converting decimal into hex. The decimal value can be entered in upper or lowercase and the "0x" prefix is optional. Also, a negative sign is allowed:

```
E> dec
Enter a value in decimal: -1234
     hex: 0xFFFFFFFFFFFFFB2E   (-0x4d2)
     decimal: -1234
```

```
     ascii: "........"
     real: nan
E> dec
Enter a value in decimal: 123456789
     hex: 0x00000000075BCD15   >117 MiBytes
     decimal: 123456789
     ascii: ".....[.."
     real: 6.09957581907715021013858322le-316
E>
```

The "**ascii**" command is mostly useful for determining the ASCII code for a given character. You must enter a line containing a exactly one character from the ASCII character set.

```
E> ascii
Enter a single character followed by a newline/return: k
     hex: 0x000000000000006B
     decimal: 107
     ascii: ".......k"
     real: 5.28650241050133802268928608 4e-322
E>
```

parms

In order to startup, the emulator needs some basic information about the Blitz-64 processor to be emulated. For example, it needs to know how much memory the system will have, how many cores, and so on.

This information is normally kept in a file named "**emulationParms**". If this file exists upon startup (or at a "**reset**" or "**rerun**" command), it will be read and the values of the parameters will be gotten from the file. If the file does not exist, then default values will be used.

The "**parms**" command is used to

- Display the current values of the "emulation parameters", and
- Create a new "**emulationParms**" file.

The command begins by displaying the current values. In the following example, there was no "**emulationParms**" file upon start up and these are the default values.

The command asks whether a file containing the defaults should be created (see highlighted):

```
E> parms
=========================  Emulation Parameters  ==============================
  PRIVATE_MEMORY_SIZE              0x0000000040000000 (decimal: 1073741824)
  SHARED_MEMORY_SIZE               0x0000000040000000 (decimal: 1073741824)
  NUMBER_OF_TLB_REGS               0x0000000000000010 (decimal: 16)
  VALUE_OF_CSR_VERSION             0x000249F000000001 (decimal: 644245094400001)
  INITIAL_VALUE_OF_PC              0x0000000400000000 (decimal: 17179869184)
  CORES_NUMBER_OF_COLS             0x0000000000000001 (decimal: 1)
  CORES_NUMBER_OF_ROWS             0x0000000000000001 (decimal: 1)
  CORES_NUMBER_OF_PLANES           0x0000000000000001 (decimal: 1)
  BOOT_ROM_START_ADDR              0x0000000400000000 (decimal: 17179869184)
  BOOT_ROM_NUMBER_OF_PAGES         0x0000000000000040 (decimal: 64)
  SECURE_STORAGE_START_ADDR        0x0000000400100000 (decimal: 17180917760)
  SECURE_STORAGE_NUMBER_OF_PAGES   0x0000000000000001 (decimal: 1)
  SIMPLE_SERIAL_START_ADDR         0x0000000400104000 (decimal: 17180934144)
  HOST_DEVICE_START_ADDR           0x0000000400108000 (decimal: 17180950528)
  DEBUG_INVOKES_EMULATOR           0x0000000000000001 (decimal: 1)
  START_ALL_CORES                  0x0000000000000000 (decimal: 0)
  IN_RAW_IGNORE_CONTROL_C          0x0000000000000000 (decimal: 0)
  TRANSLATE_INPUT_CR_TO_NL         0x0000000000000001 (decimal: 1)
================================================================================

=== ABOUT THE EMULATION PARAMETERS...
===
=== The emulation parameters are read in from the file "emulationParms", if it exists
=== when the emulator starts up.  If the file does not exist at startup, defaults
=== are assumed.  You may edit the "emulationParms" file to change the values. To re-read
=== an updated "emulationParms" file, either restart the emulator or use the “reset"
=== command.

The file "emulationParms" does not seem to exist and the above values are the defaults.

Would you like me to write these values out, creating a new file? y
The "emulationParms" file has been written out.
E>
```

In this case, the user answered “yes” and a text file named “emulationParms” was created. Here are the contents of this file:

```
# Blitz-64 Emulation Parameters
#
# This file is read by the Blitz-64 emulator when it starts up and after a
# "reset" command.  This file is used to initialize various values that
# will be used by the emulator.
#
# This file was produced by the emulator with the "parms" command.  It may
# be edited to change any or all values.
#
# Each line has variable name followed by an integer value.  A value may
# be specified in either decimal (e.g., "1234") or hex (e.g., "0x1234abcd56780000").
# Values may be left out if desired, in which case a default will be used.
#
#
```

```
PRIVATE_MEMORY_SIZE              0x0000000040000000
SHARED_MEMORY_SIZE               0x0000000040000000
NUMBER_OF_TLB_REGS               0x0000000000000010
VALUE_OF_CSR_VERSION             0x000249F000000001
INITIAL_VALUE_OF_PC              0x0000000400000000
CORES_NUMBER_OF_COLS             0x0000000000000001
CORES_NUMBER_OF_ROWS             0x0000000000000001
CORES_NUMBER_OF_PLANES           0x0000000000000001
BOOT_ROM_START_ADDR              0x0000000400000000
BOOT_ROM_NUMBER_OF_PAGES         0x0000000000000040
SECURE_STORAGE_START_ADDR        0x0000000400100000
SECURE_STORAGE_NUMBER_OF_PAGES   0x0000000000000001
SIMPLE_SERIAL_START_ADDR         0x0000000400104000
HOST_DEVICE_START_ADDR           0x0000000400108000
DEBUG_INVOKES_EMULATOR           0x0000000000000001
START_ALL_CORES                  0x0000000000000000
IN_RAW_IGNORE_CONTROL_C          0x0000000000000000
TRANSLATE_INPUT_CR_TO_NL         0x0000000000000001
```

To modify the parameters, the user can edit this file with some text editor and modify the hex values directly. Upon restarting the emulator, the "**parms**" command can be used to verify the new values. The highlighted area show some changes that have been made.

```
E> parms
=========================  Emulation Parameters  ==============================
  PRIVATE_MEMORY_SIZE              0x0000000080000000 (decimal: 2147483648)
  SHARED_MEMORY_SIZE               0x0000000004000000 (decimal: 67108864)
  NUMBER_OF_TLB_REGS               0x0000000000000010 (decimal: 16)
  VALUE_OF_CSR_VERSION             0x000249F000000001 (decimal: 644245094400001)
  INITIAL_VALUE_OF_PC              0x0000000400000000 (decimal: 17179869184)
  CORES_NUMBER_OF_COLS             0x0000000000000002 (decimal: 2)
  CORES_NUMBER_OF_ROWS             0x0000000000000003 (decimal: 3)
  CORES_NUMBER_OF_PLANES           0x0000000000000004 (decimal: 4)
  BOOT_ROM_START_ADDR              0x0000000400000000 (decimal: 17179869184)
  BOOT_ROM_NUMBER_OF_PAGES         0x0000000000000040 (decimal: 64)
  SECURE_STORAGE_START_ADDR        0x0000000400100000 (decimal: 17180917760)
  SECURE_STORAGE_NUMBER_OF_PAGES   0x0000000000000001 (decimal: 1)
  SIMPLE_SERIAL_START_ADDR         0x0000000400104000 (decimal: 17180934144)
  HOST_DEVICE_START_ADDR           0x0000000400108000 (decimal: 17180950528)
  DEBUG_INVOKES_EMULATOR           0x0000000000000001 (decimal: 1)
  START_ALL_CORES                  0x0000000000000000 (decimal: 0)
  IN_RAW_IGNORE_CONTROL_C          0x0000000000000000 (decimal: 0)
  TRANSLATE_INPUT_CR_TO_NL         0x0000000000000001 (decimal: 1)
================================================================================

=== ABOUT THE EMULATION PARAMETERS...
===
=== The emulation parameters are read in from the file "emulationParms", if it exists
=== when the emulator starts up.  If the file does not exist at startup, defaults
=== are assumed.  You may edit the "emulationParms" file to change the values. To re-read
=== an updated "emulationParms" file, either restart the emulator or use the "reset"
=== command.
```

```
The file "emulationParms" already exists.

Would you like me to write these values out, overwriting the existing file? n
E>
```

If the "**emulationParms**" file contains errors when it is read in, the emulator will complain with an error message and ignore all the parameters.

```
E> reset
Resetting all processor state and re-reading file "MyProgram.exe"...

*****  ERROR in "emulationParms" file: CORES_NUMBER_OF_COLS is not 0..1023!
*****  ERROR in "emulationParms" file: An attempt to set non-existent
                    value "BOOT_ROM_START_ADDRxx"!
*****  ERROR in "emulationParms" file: The line beginning
                   "SECURE_STORAGE_NUMBER_OF_PAGES" has no value!
*****  ERROR in "emulationParms" file: All values in the file
                    have been ignored. Use the "parms" command.
E>
```

rom

Upon power-up every Blitz core begins by executing a boot program which is located in read-only memory (ROM). This area of memory is called the "**BootROM**". The Program Counter (PC) is initialized to the first address in the BootROM and execution begins with the first instruction being fetched from the first bytes of the BootROM area. This program is called the "**BootROM program**".

The BootROM code is a short program that is responsible for getting the processor going. Tasks might include running a Power-On-Test (POST) to determine if the core is functional and determining how much memory is installed and resetting the I/O devices. The main task is often to read in something (such as a kernel) from storage (i.e., or disk or flash memory) and end execution by transferring to the storage-based program it read in. The BootROM code might also provide a few critical functions that can be used later. Examples include some form of very basic character-based output which can be used for error messages when all else fails. The BootROM might use these functions itself to print out diagnostic messages during the boot process.

Historical Comments

In non-Blitz computers, the program equivalent to our BootROM code is called "**BIOS**", but we don't use this term for Blitz. The term "BIOS" implies a set of concrete specifications and behavior that applies primarily to Windows/PC machines.

BIOS ends its execution by reading a single 512 byte block, called the "**Master Boot Record**" (MBR), from disk/flash storage and jumping to that. 512 is not a lot of space for a program, so a multi-step boot process was required. It works like this: BIOS reads in a 512 byte block of memory and jumps to it. The code in the 512 byte block is called the "Boot Loader-First Stage". This program then reads in a larger program called the "Boot Loader-Second Stage".

The first stage was necessarily a very small program, limited to 512 bytes, so it is not able to parse and understand disk partitions or a complex file system. Therefore, the first stage simply reads a fixed, predetermined area of disk/flash storage. For example, if the MBR (master boot record) is block 0, the first stage code might read blocks 1-31 from storage and jump to them. The second stage is then capable of looking at the file system and understanding directories and so on. If the storage device is partitioned, it will need to understand that as well. So the second stage is a more sophisticated program capable of searching the file system and locating the file containing the desired OS kernel. The second stage loads the kernel and jumps to it. It is the second stage that determines which kernel (of there are several available) will be chosen to execute.

Since the blocks containing the second stage must be at fixed locations (such as contiguous blocks 1-31 in our example), the second stage essentially lives outside any partitions of file systems. The boot process also lives outside of any security protection provided by the OS kernel, so it is a point of vulnerability. The boot process must be absolutely secure, protected, reliable, and impenetrable to malware, or else the kernel it loads can be compromised.17

Over time, we've seen…

- An increase in the number of file systems
- An increase in the number of OS kernels
- An increase in the variety of hardware
- An increase in the security threats and malware attacks

[17] In 2021, we really ought to say "*will* be compromised" instead of "*can* be compromised".

In order to accommodate the choices, there has been an increase in the complexity of the boot process, leading to boot processes with names like GNU Gand Unified Bootloader (GRUB), OpenFirmware, Firmware Interface (EFI), Unified Extensible Firmware Interface (UEFI).

The complexity increase has lead to a modern boot loader that requires drivers and has a shell with multiple commands. This is beginning to look like a single-user OS. With this level of complexity, comes the need for new version, software patches, fixes. To hide all this from the typical user, the updating process may be automated.

A Blitz implementation contains a section of memory called the BootROM. The starting address and size of this region is fixed and unchangeable. It is specified by the following emulation parameters, so it can be changed by updating the "**emulationParms**" file:

Parameter Name	***Typical Value***
BOOT_ROM_START_ADDR	0x4_0000_0000
BOOT_ROM_NUMBER_OF_PAGES	64 pages (1 MiByte)

The file "emulationROM" contains bytes. In this example, the file size would be exactly 1 MiByte. Upon startup or the "**reset**" command, the bytes are loaded into the processor ROM.

When emulating a multi-core processor, all cores share the same ROM. Whether there is a separate copy of the ROM for each core or whether there is only one ROM is invisible to the code.

The ROM cannot be altered by the Blitz machine instructions since the ROM is read-only.

However, the user of the emulator can modify the ROM. In the following example, the "**dumpMem2**" command is used to examine the first few bytes of the ROM area. The first 8 bytes are highlighted.

```
E> dm2
Enter the starting (physical) memory address in hex: 400000000
Enter the number of bytes in hex (or 0 to abort): 30
BOOTSTRAP ROM MEMORY:
400000000:  0300 002F  1400 010D  0400 00DD  1A00 0060    .../...........`
400000010:  002B 00F7  0500 3077  0700 3077  1100 0078    .+....0w..0w...x
400000020:  1A00 0060  1700 0241  0400 1D02  1900 074E    ...`...A.......N
```

We can also disassemble the bytes in the ROM area with the "**dis**" command:

```
E> dis
Enter the beginning address (in hex): 400000000
      400000000: 0300002F       mov        sp,r2         # synthetic for ORI _,_,0
      400000004: 1400010D       upper20    gp,0x10       # decimal = 16
      400000008: 040000DD       xori       gp,gp,0
      40000000C: 1A000060       jalr       r0,0(r6)
  ... etc ...
```

Then, we can use the "**setmem**" command to alter memory. Here, we modify the first 8 bytes:

```
E> setmem
Enter the (physical) memory address in hex of the doubleword to be modified: 400000000
***** This address is in Boot ROM, but you can proceed to store to it *****
The old value is:
0x400000000: 0x0300002F1400010D
Enter the new value (8 bytes in hex): 1111222233334444
0x400000000: 0x1111222233334444
E>
```

We can use the "dumpMem2" command again to see the change:

```
E> dm2
Enter the starting (physical) memory address in hex: 400000000
Enter the number of bytes in hex (or 0 to abort): 30
BOOTSTRAP ROM MEMORY:
400000000:  1111 2222  3333 4444  0400 00DD  1A00 0060    .""33DD.......`
400000010:  002B 00F7  0500 3077  0700 3077  1100 0078    .+....0w..0w...x
400000020:  1A00 0060  1700 0241  0400 1D02  1900 074E    ...`...A.......N
E>
```

At some later time, we will terminate the emulator with the "**quit**"command. If the ROM has been changed, the emulator will ask whether the new ROM contents should be made permanent by writing it to the "**emulationROM**" file.

```
E> q
The ROM has been modified. Shall I write it out to the host file ("emulationROM")? y
The "emulationROM" file has been updated.
Shell>
```

There is also a "**rom**" command which gives the user a chance to write out the ROM contents immediately.

```
E> rom

=== ABOUT THE READ-ONLY MEMORY (ROM)...
===
=== This emulator supports only a single ROM memory; All cores shared this ROM.
=== The data for the ROM comes from a file called "emulationROM", if it exists
```

```
=== when the emulator starts up. If this file does not exist at startup, the ROM
=== is initialized to zeros. To re-read the contents of an updated "emulationROM"
=== file, either restart the emulator or use the "reset" command.

The file "emulationROM" already exists.

Would you like me to write out the current ROM contents, overwriting the existing file? y
```

This command continues with a similar question about the Secure Storage Device:

```
=== ABOUT THE SECURE STORAGE DEVICE...
===
=== This emulator supports only a single Secure Storage device; it is shared by all cores.
=== The data for the the Secure Storage device comes from a file called "emulationSecure",
=== if it exists when the emulator starts up. If this file does not exist at startup, the
=== Secure Storage device is initialized to zeros. To re-read the contents of an updated
=== "emulationSecure" file, either restart the emulator or use the "reset" command.

The file "emulationSecure" already exists.

Would you like me to write out the current Secure Storage contents, overwriting the existing file? n
E>
```

The "**Secure Storage Device**" is very much like a ROM except that it can be updated.

The data for the Secure Storage Device is kept in a file called "**emulationSecure**" From the point of view of the emulator, it functions very much like the ROM.

The Secure Storage Device is described in:

> *"Blitz-64: Instruction Set Architecture Reference Manual"*

If an attempt is made to store into the Secure Storage Device after it is locked, the emulator will print an error message and halt execution:

```
***** Probable Error in the Blitz Code: Attempt to STORE to
            Secure Storage, but it is locked! (The STORE was ignored.) *****

Entering machine-level debugger...
=========================================
=====                               =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III      =====
=====           6 May 2021            =====
=====                               =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E>
```

How To Update BootROM

Here is how to store a "BootLoader" program into the BootROM:

- Write the program.
  (either hand-coded assembler or large KPL program)
- Compile, Assemble, and Link to produce a **.exe** executable file.
- Run the emulator, without "auto-go" (without "**-g**" on command line).
  (This will load the program into the ROM area.)
- Use the "**rom**" command to write to the "**emulationROM**" file.
- Quit the emulator.

A Blitz BootLoader will serve roughly the same function as a firmware program like BIOS in traditional computers. It can be either a small, hand-coded assembly language program or a larger KPL program.[18]

The assembly program should start with a line such as:[19]

```
.begin   kernel,startaddr=0x400000000,gp=undefined
```

The next step is to assemble and link it:

```
Shell% asm myBoot.s -o myBoot.o
Shell% link myBoot.o -o myBoot.exe -k
```

Then, the emulator is started with this executable:

```
Shell% blitz myBoot.exe
Reading executable file...
The executable file (myBoot.exe) was loaded. The _entry address (0x400000000)
was loaded into register r6.
=========================================
=====                                 =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III      =====
```

[18] At this time, all BootLoader programs are stand-alone, hand-coded assembly programs.

[19] The "**gp=undefined**" might left off, or replaced with something specific like "**gp=0x400008000**". If left off, then the default value of 0x000010000 is assumed since "**kernel**" is present. If not undefined, then the code had better initialize the **gp** register as the first statement.

```
=====           6 May 2021            =====
=====                                 =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E>
```

This will load the executable into the ROM area. Next we use the "**rom**" command to write ROM data to the "**emulationROM**" file.

We answer the question about writing the ROM with "yes" and the question about writing the SecureStorage with "no".[20]

```
E> rom

=== ABOUT THE READ-ONLY MEMORY (ROM)...
===
=== This emulator supports only a single ROM memory; All cores shared this ROM.
=== The data for the ROM comes from a file called "emulationROM", if it exists
=== when the emulator starts up. If this file does not exist at startup, the ROM
=== is initialized to zeros. To re-read the contents of an updated "emulationROM"
=== file, either restart the emulator or use the "reset" command.

The file "emulationROM" already exists.

Would you like me to write out the current ROM contents, overwriting the existing file? y
The "emulationROM" file has been updated.

=== ABOUT THE SECURE STORAGE DEVICE...
===
=== This emulator supports only a single Secure Storage device; it is shared by all cores.
=== The data for the the Secure Storage device comes from a file called "emulationSecure",
=== if it exists when the emulator starts up. If this file does not exist at startup, the
=== Secure Storage device is initialized to zeros. To re-read the contents of an updated
=== "emulationSecure" file, either restart the emulator or use the "reset" command.

The file "emulationSecure" already exists.

Would you like me to write out the current Secure Storage contents, overwriting the existing file? n
E> q
Shell%
```

Finally, we issue the "**quit**" command.

The next time the emulator is started up, the new contents of the ROM will be present.

[20] The second answer doesn't matter. Since the Secure Storage has not been modified, writing it won't hurt.

Upon startup, the emulator will first load register r6 with the "**EntryPoint**" from the executable **.exe** file. Then it will load the PC with the first address of the BootRom (i.e., 0x4_0000_0000) and begin executing instructions.

For most KPL programs, we can use a very simple BootLoader program that simply jumps to the value in register **r6**. Here is such a program:

```
      .begin     kernel,startaddr=0x400000000,gp=undefined
_entry:
      .export    _entry
      jr         r6          # Jump to address in r6 (i.e., to "_entry")
```

Upon startup or "**reset**", the emulator will read an executable **.exe** file into memory and load the registers as follows:

r1	0x636F6C64626F6F74 (This is ASCII for "`coldboot`")
r2	The size of the private memory in bytes
r3	The starting address of shared memory
r4	The size of shared memory in bytes
r5	The highest address loaded
r6	The value of "**EntryPoint**" (from the **.exe** file)

The emulator will then set PC to the first address of the BootROM and begin executing the BootLoader program, which will immediately jump to the "**_entry**" label in the program.

serial

The "serial" command can be used to switch between "cooked" and "raw" mode. The command starts by printing an explanation, then ends by asking if the user wishes to switch mode.

```
E> serial
================================================================================
From time to time a running Blitz program may read characters from the "Serial
I/O" device, which is intended to model a "terminal" interface.  The character
data to be supplied to the running Blitz program will come from either a file
(which is specified using the "-i filename" command line option when the
```

emulator is started) or from the interactive user-interface which you are apparently using now.

With this second option, you may enter characters on "stdin" at any time during the emulation of a running Blitz program. These characters will be supplied to the running Blitz program (via the emulated Serial I/O device). If the emulator seems to hang, it may be because the emulator is waiting for you to type additional characters to supply to the running Blitz code. (It may also be because the Blitz program has gotten into an infinite loop.)

At any time you may always hit control-C to suspend instruction execution and re-enter the emulator command interface.

Normally an operating system will process user input by echoing characters on the screen, buffering entire lines, and processing special characters like backspaces, etc. The OS then delivers input, one full line at a time, to running programs. This is called "cooked" input. But for some programs cooked input is inadequate so these program use "raw mode". In raw mode, each character is delivered as-is immediately after the key is pressed, with no buffering and without the normal echoing and processing of special characters.

The Blitz emulator runs in either "raw mode" or "cooked mode". Simple Blitz applications are usually designed to be run in cooked mode, while more complex programs (e.g., OS kernels, editors, and anything that handles control-C or arrow keys) may be designed to operate in raw mode.

In cooked mode, the host OS will suspend the emulator until you enter a complete line of data and hit ENTER. This allows you to use the Backspace/DELETE key, without requiring the Blitz program to deal with corrections.

In raw mode, the normal echoing of keystrokes by the host OS is turned off. A good Blitz program should echo all characters, so you *should* see each keystroke echoed properly. But of course your Blitz program may not be working properly. It may fail to echo characters because it has a bug. Also, the running Blitz program may not handle backspaces, newlines, CRs, etc., exactly as you and your terminal expect. It may be helpful to recall \n=Control-J, \r=Control-M, and Backspace=Control-H. On some terminals, the ENTER key is \r, while many programs expect to use \n for END-OF-LINE.

Note that if a Blitz program expects to run in raw mode, but is run in cooked mode, you will see all the characters echoed, resulting in a second, identical line. A Blitz program meant to run on hardware will probably want to echo all character data, so a duplication of input will occur if the program is emulated in cooked mode.

The mode only affects typed input to be delivered to the running Blitz program; typed input to the emulator itself is always in cooked mode.

For the Simple Serial device, the raw/cooked distinction applies only to single character and string input. The device also supports the input of integers in decimal and hex, but this always occurs in cooked mode.

The default is cooked mode; raw mode is selected with the "-raw" command line option. The mode may also be changed with this command.

```
Input for the Serial I/O device will come from..... "stdin"
The current input mode is.......................... "cooked"
================================================================================
Do you want to change to "raw" mode? y
The terminal is in "raw" mode.
E>
```

Chapter 4: Errors and Warnings

Problems During Emulation

Things could go wrong during emulation. Here we discuss different kinds of errors.

Fatal Error

Some error conditions are considered "fatal" and cause an error message to be displayed, followed by an immediate termination of the emulator. If this happens, you might see a message:

```
***** Blitz Emulator Error: <details> *****
```

The host file system could return an error or an attempt to allocate memory might fail. Here are two examples:

```
***** Blitz Emulator Error: Error from fseek for executable file *****

***** Blitz Emulator Error: Calloc failed - insufficient memory
                                            available - Shared RAM *****
```

Fatal error messages are sent to **stderr**, not **stdout**. After this, the emulator will try to clean up and exit gracefully, with a Unix/Linux/POSIX exit code of 1.

In some cases, the error message will be preceded by additional messages giving more information about the problem.

Command Line Errors

Any error on the command line will result in a message and immediate termination. For example:

```
***** Blitz Emulator Error: Options -raw and -i are incompatible;
                                                Use -h for help display. *****
```

The "-nowarn" Command Line Option

If the emulator is run with the "-nowarn" option, certain warning and informational messages will be suppressed.

Here we see a typical run of the emulator. The highlighted material is displayed by the emulator during startup, before execution begins.

```
Shell% blitz -g MyProgram.exe
***** WARNING: The file "emulationParms" was not found. *****
Reading executable file...
The executable file (MyProgram.exe) was loaded. The _entry address (0x00001885C)
was loaded into register r6.
Beginning execution...
😀😀😀😀😀  MyProgram running... HELLO WORLD!  😀😀😀😀😀
Shell%
```

With the use of "-nowarn", the highlighted material is suppressed:

```
Shell% blitz -g MyProgram.exe -nowarn
😀😀😀😀😀  MyProgram running... HELLO WORLD!  😀😀😀😀😀
Shell%
```

The "**-nowarn**" option will suppress some informational messages and some execution warnings, but the handling of "fatal" errors is not changed.

Execution Errors

During execution of Blitz code, certain conditions are considered errors or at least probable errors. These will result in a message and an immediate halt to execution. The message will have this format:

```
***** Probable Error in the Blitz Code: <details> *****
```

An example follows. There may be additional information displayed (see highlighted material):

```
Shell% blitz -g -nowarn MyProgram.exe

***** Probable Error in the Blitz Code: Within PerformVirtualMapping,
                                                    the physical address is uninstalled/invalid *****
Address = 0x300000000  [ PHYSICAL ]

***** Probable Error in the Blitz Code: Attempt to read from uninstalled address; zero returned *****
Done!

Entering machine-level debugger...
=========================================
=====                               =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III    =====
=====           6 May 2021          =====
=====                               =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about commands.
E> g
```

In general, the user can resume execution after such a message with the "**go**" command. But the user really ought to use other debugging commands to understand and fix the problem.

Some execution errors will be ignored if the "**-nowarn**" command line option is present. An example is an attempt to execute an illegal instruction. Without "-nowarn", this will suspend execution, as shown here:

```
Shell% blitz -g MyProgram.exe
Reading executable file...
The executable file (MyProgram.exe) was loaded. The _entry address (0x00001885C) was loaded into
register r6.
Beginning execution...

***** Probable Error in the Blitz Code: Illegal instruction - Suspending execution;
                                                    'g' will allow exception to proceed *****
***** PROBLEM INSTRUCTION:    Within Function "main"   [MyProgram.c]
      00000C4B0: FFFFFFFF              # decimal = -1, ascii = "...."
Done!

Entering machine-level debugger...
=========================================
=====                               =====
=====  The Blitz-64 Machine Emulator  =====
=====     by Harry H. Porter III    =====
=====           6 May 2021          =====
=====                               =====
=========================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about commands.
```

```
E>
```

However, with the "**-nowarn**" option, emulation will not halt.

According to the Blitz ISA, iIllegal instructions cause an "**Illegal Instruction Exception**" and cause exception processing to begin. The kernel executes in Kernel Mode; an illegal instruction in the kernel represents a real problem and the emulator's debugger functionality may come in handy. However, User Mode programs can be expected to occasionally execute illegal instructions from time-to-time. The Blitz OS kernel code should handle all User Mode exceptions so—assuming the kernel can handle exceptions correctly—there is no reason to halt emulation.

Other errors will cause a message and halt execution regardless of the "**-nowarn**" option.

For example, any attempt to read from uninstalled memory represents a kernel error and will halt execution. This error should never happen in User Mode code, since the kernel will presumably map all virtual pages into valid physical memory pages frames.

Execution error messages are sent to **stderr**, not **stdout**, in case **stdout** is being redirected to a file.

Program Logic Errors

The emulator contains number of internal consistency checks. If the emulator detects a problem, you might see this message:

```
***** PROGRAM LOGIC ERROR IN BLITZ EMULATOR: <details> *****
```

For example:

```
***** PROGRAM LOGIC ERROR IN BLITZ EMULATOR: mod->moduleNumber != modNum *****
```

Such messages are sent to **stderr**, not **stdout**. After this, the emulator will exit immediately, with Unix/Linux/POSIX exit code of 1, without attempting to update or close files.

DIV / REM Implementation Dependencies

When the DIV and REM instructions involve a negative operands, there can be a different result, depending on what sort of division is implemented. For example:

	x	y	DIV	REM	(y * DIV) + REM == x
Euclidean:	-7	-3	3	2	-3 * 3 + 2 = -7
Truncated:	-7	-3	2	-1	-3 * 2 + -1 = -7

The Blitz ISA specification leaves the choice open, as an "implementation dependency".

If an attempt is made to execute a DIV or REM instruction where the outcome is "implementation dependent", the emulator will print a message and execution will halt. For example:

```
***** Probable Error in the Blitz Code: During a REM instruction, an implementation
      dependency was encountered. The result depends on whether "truncated division" or
      "Euclidean division" is implemented. Truncated assumed. Okay to proceed with 'g'. *****
```

If the "**-nowarn**" option is present, the message will not be printed and execution will not halt.

Floating Point Dependencies

According to the Blitz ISA, floating point instructions (such as FADD, FMUL, …) can either be implemented or cause an "Emulated Instruction Exception". This is configurable with the "-fp" command line option.

If "**-fp**" is absent, then every floating point instruction will cause an Emulated Instruction Exception.

If "**-fp**" is present, then the emulator will execute the instruction.

The FMADD, FNMADD, FMSUB, and FNMSUB instructions impose some tricky issues regarding the proper setting of the **overflow**, **underflow**, **inexact**, and **invalid** flags. As of this date, I have not implemented this. Any attempt to executed on of these

instructions will issue an "unimplemented code" message and halt execution, unless "**-nowarn**" is present. If "**-nowarn**" is present, execution will continue without interruption or a message.

Tight Infinite Loops

The following is an example of a tight infinite loop:

```
LoopLabel:
                jump        LoopLabel
```

The emulator will detect such an instruction and immediately halt with a message like this:

```
***** Probable Error in the Blitz Code: A TIGHT INFINITE LOOP WAS DETECTED! *****
***** The jump-to-self instruction:
   LoopLabel:
      00000A248: 19000000       jump       LoopLabel           # PC + 0x0
```

If (for some reason) you really want an infinite loop to be executed, you can code this instead:

```
LoopLabel:
                nop
                jump        LoopLabel
```

The emulator will not recognize this as special and will merrily spin. In that case, you can use control-C to halt emulation and regain control.

To debug code using the emulator, it is common to place DEBUG instructions within your code. Whenever a DEBUG instruction is encountered, the emulator goes into command mode and you can begin debugging.

However, when running kernel code, you may want DEBUG and BREAKPOINT instructions to function normally and cause exceptions (per the Blitz-64 ISA) and not halt emulation. To do this, you would run the emulator with the **-nodebug** option or set the **DEBUG_INVOKES_EMULATOR** parameter to false.

In such situations, you can use a tight infinite loop check described here to suspend emulation and regain control.

Chapter 5: Miscellaneous Instructions

The SLEEP1 Instruction

According to the Blitz Instruction Set Architecture (ISA), the SLEEP1 instruction will place the core in a "sleep state" and suspend instruction execution until an interrupt is received.

At this time, the emulator does not implement any sources of external interrupts. At some future time, we expect inter-core interrupts at a min

With the emulator, the execution of a SLEEP1 instruction will cause execution of that core to halt and the status to be changed from RUNNING to SLEEP-1.

If the emulator is configured with only one core or there are no other RUNNING cores, all execution will halt and debugging commands will be accepted. If there are other RUNNING cores, execution will continue with other cores.

The following message will be printed, unless option "**-nowarn**" was present on the command line:

```
***** Core N executed a SLEEP1 instruction and has been halted! *****
```

The SLEEP2 Instruction

Like SLEEP1, the SLEEP2 instruction will place a core in a "sleep state" and suspend instruction execution until an interrupt is received. In a hardware implementation, the power consumption in the two sleep states may be different, with SLEEP2 being a deeper, lower power version.

With the emulator, the SLEEP2 instruction will terminate the execution of the current core and change its status from RUNNING to SLEEP-2.

But there is more. SLEEP2 is used to implement the "**EmulatorShutdown**" function. If the emulator is running with "**auto-go**", then the emulator itself will terminate. It will return a Unix/Linux/POSIX return code using the value in register **r1**.

For example:

```
Shell% blitz MyProgram.exe -g -nowarn
😀😀😀😀😀  MyProgram running... HELLO WORLD!  😀😀😀😀😀
Shell% echo $?
123
Shell%
```

If the emulator is not running with "**auto-go**", then a message will be printed and the execution of all cores will be halted. Other cores will keep their status of RUNNING, but the emulator will begin accept debugging commands. For example:

```
Shell% blitz MyProgram.exe
  ... startup messages ...
E> g
Beginning execution...
😀😀😀😀😀  MyProgram running... HELLO WORLD!  😀😀😀😀😀
Emulation stopped by SLEEP2 instruction; EXIT CODE: r1 = 123!
***** Core 0 executed a SLEEP2 instruction and has been halted! *****
Done!
E>
```

The DEBUG Instruction

According to the Blitz-64 Instruction Set Architecture (ISA), the DEBUG instruction shall cause a "**Debug Exception**".

The emulator will either process the DEBUG instruction according to the ISA or will halt execution and enter the emulator's debugging command mode. This is controlled by a parameter in the "**emulationParms**" file:

```
DEBUG_INVOKES_EMULATOR          0x0000000000000001
```

If the value is 1 (i.e., true), the emulator will halt and enter debugging mode. If 0 (i.e., false) the emulator will execute the DEBUG instruction according to the ISA and execution will not be halted.

The default value is 1: the DEBUG instruction will halt execution.

The DEBUG instruction is conveniently inserted at any point in a KPL program with the **debug** statement. The **debug** statement can optionally be followed by a string:

```
function main ()
    ... etc...
  debug
    ... etc...
  debug "This is a message"
```

When the emulator hits the first DEBUG instruction it displays the source line number where the debug statement occurred, the address in memory, and a message designed to catch the eye. See highlighting:

```
Shell% blitz MyProgram.exe -nowarn
      ... etc...
****  A DEBUG machine instruction was executed  *****

Next instruction to execute:
   DEBUG (line 21)
   ----------  ################# DEBUG #################
      00000C510: 00280000       debug
E>
```

Next, we will continue execution with the "**go**" command. When the emulator hits the second DEBUG instruction, it displays the informative message as well:

```
E> g
      ... etc...
****  A DEBUG machine instruction was executed  *****

Next instruction to execute:
   DEBUG (line 34)
   ----------  ################ This is a message ################
    _Label_44:
      00000C5FC: 00280000       debug
E>
```

Imagine that you wish to look at the assembly code produced by the KPL compiler for a particular KPL statement. You will need to take a look at the file produced by the KPL compiler, which is called "**MyProgram.s**" in the case.

Such a file is typically very long. It can be difficult to locate the lines of interest.

However, KPL's debug statement makes this fairly easy.

Here is a section of **MyProgram.s** produced by the KPL compiler. The highlighted lines came from the first **debug** statement.

```
#   Argument fileID <-- _P_HostInterface_stdout  (8 bytes)
        loadd         r1,_P_HostInterface_stdout
#   Call function 'f_print_end'
        call          _P_PrintPackage_f_print_end
        .stmt         debug,line=21
        .comment      "################# DEBUG #################"
        debug
# FOR STATEMENT...
        .stmt         for_init,line=22
#   Calculate and save the FOR-LOOP starting value
        .stmt         for_init,line=22
        movi          r7,0
        stored        64(sp),r7       # _temp_45
```

Here is the code for the second **debug** statement.

```
#   i = i + 1
        loadd         r7,72(sp)       # i
        addi          r7,r7,1
        stored        72(sp),r7       # i
        jump          _Label_41
# END FOR
_Label_44:
        .stmt         debug,line=34
        .comment      "################# This is a message #################"
        debug
# CALL STATEMENT...
        .stmt         call,line=37
#   Argument exitCode <-- 123  (8 bytes)
        movi          r1,123          # 0x000000000000007b
```

In each case, you see the DEBUG instruction, which causes the emulator to halt execution. You also see two debugging statements (**.stmt** and **.comment**). The debugging information goes into the executable **.exe** file and not into memory at runtime. The emulator will use that debugging information at the time the DEBUG instruction is encountered.

All the "###"s are added for the purpose of making the line stand out to the human eye when scanning a long file or when executing a program containing a lot of DEBUG instructions.

There is also another aspect to the handling of the DEBUG instruction. Often, the running Blitz code will detect an error or exception. Generally speaking, the KPL error handling will throw an "error"n, allowing the running KPL application to catch the error and deal with it. However, if the error is not caught, the debugger must be invoked. In this case, code in the **System** package will execute a DEBUG statement. However, the location of the error is not within the System package. In this case there is a bit of coordination between the code in System and the emulator.

When encountering a DEBUG statement, the emulator checks to see if a particular DEBUG instruction was executed. In particular, a function called **EmulatorDebuggingRequested** (a hand-coded assembly function within **runtime.s**) is looked for. This function will leave the address where the error occurred in register **r1**. The emulator will retrieve this value and display a message, as if the error had occurred at given location. This will give the user more appropriate error reporting.

This is illustrated in the next example.

Here, an **Arithmetic Exception** occurs in a running KPL program. From the highlighted material, we can see what happened and where in the source code this happened. Also, noticing that it was a DIV (divide) instruction, we can guess that the problem was divide-by-zero, although it could be overflow.[21]

```
E> g
====================
==================== "System: ERROR_ArithmeticException" was thrown but not
====================                                caught within thread "Main Thread"
====================

The CATCH STACK is empty


**********  RUNTIME ERROR: An "ARITHMETIC EXCEPTION" has occurred!  **********

    Offending Instruction = 0x0000000000050767

*****  Native debugger is not implemented - EXECUTION TERMINATING  *****

**********  EMULATOR DEBUGGING: Type 'stack' for more info. **********

Execution is stopped at ASSIGN on line 29 in function "main"   [MyProgram.c]
      00000C560: 00050767       div        r7,r6,r7
E>
```

[21] Dividing the most negative 64 bit integer by -1 results in a positive integer that cannot be represented as a 64-bit signed int.

While this gives the programmer a good error message, we can see that execution is really stopped within the **EmulatorDebuggingRequested** function:

```
E> where
Enter an address in hex (or 0 for current PC):
CURRENT LOCATION OF PC:
   Within Function "EmulatorDebuggingRequested"   [runtime.s]
      000018958: 010010FF       addi       sp,sp,16            # hex = 0x10
E>
```

By executing the **stack** command, we see that there was some error handling done before the DEBUG instruction caused execution to halt and the emulator's debugging functions to be invoked. The error actually occurred in the function which is located just below the top four error handling functions.

```
E> stack
   Function/Method                 Execution at...         File
   ==============================  ====================    ==============================
   EmulatorDebuggingRequested                              runtime.s
   invokeDebugger                  CALL       line 2312    System.c
   RuntimeErrorArithmeticExceptio  CALL       line 2190    System.c
   _runtimeErrorArithmeticExcepti                          runtime.s
   main                            ASSIGN     line 29      MyProgram.c
   _kplEntry                                               MyProgram.c
   _entry                                                  runtime.s
      ... etc...
```

Attempting to resume execution at this point is fruitless:

```
E> g

====================  KPL PROGRAM TERMINATION  ====================
E> g

==================== The KPL program has terminated; you may not continue. ====================
E> where
Enter an address in hex (or 0 for current PC):
CURRENT LOCATION OF PC:
   Within Function "TerminateRuntime"   [runtime.s]
      000018978: 19FFFF40       jump       TerminateRuntime            # PC - 0xC (PC + 0xFFFFFFF4)
E>
```

In the future, we anticipate that a native debugger (i.e., a debugger written in KPL and executing Blitz code) will be invoked instead of relying on the emulator's debugging functionality.

The BREAKPOINT Instruction

Like the DEBUG instruction, the operation of the BREAKPOINT instruction is controlled by the "**DEBUG_INVOKES_EMULATOR**" parameter from the "**emulationParms**" file.

```
DEBUG_INVOKES_EMULATOR          0x0000000000000001
```

The default value is 1=true.

If the value is 1=true, execution will halt. There is no special processing or handling of BREAKPOINT and there is no message. This behavior is used by the KPL runtime system to simply stop execution.

If the value is 0=false, execution will continue, and a **Breakpoint Exception**, as specified by the Blitz ISA.

The CONTROL and CONTROLU Instructions

For a description of the **CONTROL** and **CONTROLU** instructions, consult:

> *"Blitz-64: Instruction Set Architecture Reference Manual"*

The definition, operation, and functionality of these instructions is "implementation dependent". The instruction essentially allows the code to provide a 64 bit value and, after something happens, to receive a 64 bit result. A specific implementation of the Blitz architecture can use these instructions to perform operations not covered by the ISA, or these instructions can simply be considered invalid, illegal operations.

The KPL language provides a way to execute these instructions with two built-in, predefined functions:

```
CPUControl (arg: int, opcode) returns int
CPUControlUserMode (arg: int, opcode) returns int
```

The "*opcode"* must be a value within 0 … 65,535. The CONTROL instruction is privileged so it can only be executed by kernel code, while the CONTROLU instruction can be executed by User Mode code as well as code running in Kernel Mode.

These instructions are unimplemented by the emulator. If the emulator encounters one of these instructions during execution, it will halt and alert the user. For example, this KPL statement

```
i = CPUControl (1234, 57)
```

will result in this during execution:

```
Shell% blitz MyProgram.exe -g -nowarn
  ... etc ...
****  A CONTROL (KernelMode) machine instruction was executed  *****

      Immed-16 control code: 0x0039 (decimal 57)
      Value in source register: 0x00000000000004D2 (decimal 1234)
      Enter a value in hex: 0x11223344
      Do you want this instruction to cause an Illegal Instruction Exception? n
  ... etc ...
```

Execution will resume.

In the future, to model specific hardware, it is reasonable to imagine that the emulator will be modified and that these instructions will perform some hitherto unknown operations.

Chapter 6: Memory-Mapped I/O Devices

Introduction

A Blitz computer will have several I/O devices and different implementations of the Blitz architecture may have difference devices.

The emulator provides the following; others may be added in the future.

BootROM
SecureStorage
SimpleSerial
HostInterface

These are discussed below.

The BootROM Area

The emulator provides an area of memory that is read-only. The location and size of the ROM is typically:

Starting Address	0x4_0000_0000
Size in Bytes	0x10_0000 (1 MiByte)
Size in 16 KiByte Pages	0x40 (decimal 64)

However, this can be be adjusted with the following parameters by updating the "**emulationParms**" file, shown here with their default values:

```
BOOT_ROM_START_ADDR             0x0000000400000000
BOOT_ROM_NUMBER_OF_PAGES        0x0000000000000040
```

Upon startup or a "**reset**" command, the emulator will initialize the ROM area with bytes it reads from the file named: "**emulationROM**", or zeros if the file does not exist.

Presumably, programs will never attempt to STORE to the ROM area. If an emulated program tries to STORE into the ROM area, the emulator will print a message and suspend instruction execution. The STORE will not occur.

Real Blitz hardware would probably just ignore any STORE attempts into this address range.

The user may update the ROM. The executable **.exe** program loaded upon startup may include addresses within the ROM. It is not an error and the bytes will overwrite the initial contents. The user may also use commands such as "**setmem**" to update bytes within the ROM area.

Upon exiting the emulator, if the ROM area has been altered from the initial contents (either by user commands or executed instructions), the emulator will ask whether the user wants to write the new ROM contents to the file "**emulationROM**".

```
E> q
The ROM has been modified. Shall I write it out to the host file ("emulationROM")? y
The "emulationROM" file has been updated.
Shell%
```

The SecureStorage Area

The Secure Storage area functions much like the BootROM area, with the following exceptions.

The data is kept in a file called "**emulationSecure**".

The location of the Secure Storage is typically:

Starting Address	0x4_0010_0000
Size in Bytes	0x4000 (16 KiBytes)
Size in 16 KiByte Pages	1

The location of the Secure Storage can be changed by modifying the following emulation parameters, shown here with their default values:

```
SECURE_STORAGE_START_ADDR          0x0000000400100000
SECURE_STORAGE_NUMBER_OF_PAGES     0x0000000000000001
```

Upon startup or a "**reset**" command, the emulator will initialize the Secure Storage area with bytes it reads from the file named: "**emulationSecure**", or zeros if the file does not exist.

The Secure Storage area is either "**locked**" or "**unlocked**", as described in the Blitz ISA. While unlocked, data can be stored and the new values will be retained, just as in normal memory.

Storing into byte 0 of the region will cause the Secure Storage to become locked. After that, it functions like ROM. Any attempt to STORE into the area is ignored. Normal programs would be unlikely to do this and the emulator will catch such errors and halt execution.

Like the BootROM, the user can update the area using the "**setmem**" command. However, the executable **.exe** file cannot store into the Secure Storage area.

Like the BootROM, if there have been changes to the Secure Storage area, either from user commands or executed code, the emulator will ask whether the user wants to write the new ROM contents to the file "**emulationSecure**".

```
E> q
The SecureStorage has been modified. Shall I write it out to
                                                    the host file ("emulationSecure")? y
The "emulationSecure" file has been updated.
Shell%
```

The SimpleSerial Device

The "**Simple Serial**" device is intended to provide a simple way for Blitz program to communicate with the user. It is not intended to model real hardware.

The location and size of the Simple Serial device is:

Starting Address	0x4_0010_4000
Size in Bytes	0x4000 (16 KiBytes)
Size in 16 KiByte Pages	1

This size is always 16 KiBytes but, the location can be be adjusted with the following parameter by updating the "**emulationParms**" file, shown here with the default value:

```
SIMPLE_SERIAL_START_ADDR        0x0000000400104000
```

The Simple Serial device is documented in:

"Blitz-64: Instruction Set Architecture Reference Manual"

The following two parameters in "emulationParms" apply to the Simple Serial device. Here they are with their default values:

```
IN_RAW_IGNORE_CONTROL_C         0x0000000000000000 (decimal: 0)
TRANSLATE_INPUT_CR_TO_NL        0x0000000000000001 (decimal: 1)
```

The "**IN_RAW_IGNORE_CONTROL_C**" parameter must be 0 (i.e., false) or 1 (i.e., true). This parameter is only used for input that is coming from **stdin** when the emulator is in "**raw**" mode. (The "**serial**" command prints out a description of these modes.)

Normally, whenever control-C is hit, it will interrupt program execution. Execution will halt and the user will be able to enter debugging commands. However, some Blitz programs (e.g., the Blitz OS) may wish to see the control-C directly and, for example, allow a Blitz shell to interpret it in the same way Linux/Unix shells do. So the emulator must not stop execution.

By setting the "**IN_RAW_IGNORE_CONTROL_C**" parameter to 1 (true), any control-C being pressed by the user will be forwarded to the running Blitz core. The Blitz program will not be interrupted.

The "**TRANSLATE_INPUT_CR_TO_NL**" parameter must be 0 (i.e., false) or 1 (i.e., true). This parameter is only used for when the emulator is in "**raw**" mode; it has no effect when running in "**cooked**" mode.

Some host systems (like Apple macOS) will treat the ENTER key as \r, and not \n.[22] So, in raw mode, macOS delivers the '\r' character (ASCII 0x0d) and not '\n' (ASCII 0x0a), which will confuse a Blitz program using the Unix/Linux/POSIX convention. When this parameter is set to true, the emulator will translate every '\r' it sees into '\n'.

The HostInterface Device

The location and size of the HostInterface device is:

Starting Address	0x4_0010_8000
Size in Bytes	0x4000 (16 KiBytes)
Size in 16 KiByte Pages	1

This size is always 16 KiBytes but, the location can be be adjusted with the following parameter by updating the "**emulationParms**" file, shown here with the default value:

```
HOST_DEVICE_START_ADDR          0x0000000400108000
```

This is a memory-mapped I/O device that facilitates communication with the host OS running this emulator.

In particular, this "device" allows a running Blitz program to:

- Perform file operations (fopen, fgetc, …)
- Retrieve the command line arguments
- Determine the date and time

Consult the **HostInterface package**. The functionality of the HostInterface device is encapsulated in a number of useful functions, such as:

```
hostArgs () returns String
hostDate () returns String
fopen    (filename: String, mode: String) returns ptr to FILE
fclose   (fileID: ptr to FILE)
remove   (filename: String) returns int
feof     (fileID: ptr to FILE) returns bool
```

[22] On my Mac, the key is actually labeled RETURN, not ENTER.

```
fgetc    (fileID: ptr to FILE) returns int
fputc    (ch: int, fileID: ptr to FILE) returns int
ungetc   (ch: int, fileID: ptr to FILE)
perror   (str: String)
fread1   (buffPtr: ptr to void, byteCount: int, fileID: ptr to FILE) returns int
fwrite1  (buffPtr: ptr to void, byteCount: int, fileID: ptr to FILE) returns int
fseek    (fileID: ptr to FILE, offset: int, whence: int)
ftell    (fileID: ptr to FILE) returns int
fputs    (src: String, fileID: ptr to FILE) returns bool
```

Commentary This device is obviously not intended to model any real I/O device. Instead, it is used to allow KPL application code to be developed. In particular, the Blitz assembler and KPL compiler were re-coded in KPL in anticipation of there existing a Blitz OS in the future. This was done to verify that the Blitz toolchain was robust and up to the task of developing the Blitz OS. In order to verify that these KPL programs were working exactly identically to the host C/C++ versions, it was necessary to run the full verification suites against them. The functionality provided by the HostInterface device is essentially the minimum required to enable the KPL versions of the assembler and compiler to function.

Using the HostInterface device, Blitz code running within this emulator can communicate with the underlying host operating system (e.g., Unix). This allows Blitz code to do things like read files and perform system calls (e.g., to get the time of day).

This is done by creating a phony device called the "**HostInterface Device**" which is memory-mapped like all Blitz I/O devices. LOADs and STOREs to this device will cause the emulator to communicate with the hist OS. Arguments to a host system call can be transferred by the Blitz code by STOREing to this "device". Results from the host OS can be retrieved by the Blitz code by LOADing from this "device".

Here are the key addresses within the HostInterface device. Although these are often called "I/O registers" they are locations in the Memory-Mapped I/O region and not the sort of registers found in a CPU core.

Register name	*offset*	*size (bytes)*	
ARG_SIZE	0	8	read only
ARG_CHAR	8	8	read only
DATE_SIZE	16	8	read only
DATE_CHAR	24	8	read only
FUN_CODE	32	8	write only
ARG_1	40	8	write only

ARG_2	48	8	write only
ARG_3	56	8	write only
ARG_4	64	8	write only
Ret_Val	72	8	read only
DO_IT	80	8	read only
LONG_STR	80	1024	write only

Here is how they are used:

ARG_SIZE

Command line arguments to the emulator are given on the emulator command line with the "**-args**" option. The **-args** is followed by a **argumentString**. For example:

```
Shell% blitz MyProgram.exe -g -args "-aaa -bbb -ccc"
```

Read this register to find the length in bytes of **argumentString**. In this example, the length of the string "`-aaa -bbb -ccc`" is 14 bytes. Reading this register also resets the current position for **ARG_CHAR** to the beginning of the string.

ARG_CHAR

Read this register to fetch the next byte in **argumentString**. Each byte is returned as an **int** (0..255) since this register is 8 bytes. In our example, to obtain all the bytes of the argumentsString, we would read this register 14 times. For any read beyond the end of the string, zeros will be returned.

DATE_SIZE
DATE_CHAR

Reading the **DATE_SIZE** register will cause the time and date to be obtained from the host and stored for subsequent retrieval by the Blitz code. The time and date will be in the form shown by this example:

```
"Sat May 22 09:44:18 2021\n"
```

A read to the **DATE_SIZE** register will (1) obtain and store the time and date from the host, (2) reset the current position to the beginning of the string, and (3) return the number of characters in the string.

To retrieve the characters, the Blitz code can read the **DATE_CHAR** register repeatedly, once for each character in the string. Each successive read will return each successive character. After each read, the current position is advanced. For any read beyond the end of the string, zeros will be returned.

FUN_CODE

In order to make a function call to the host operating system—that is, to perform a host operation—write to this register, storing a **functionCode** in it. This **functionCode** is a small integer that will identify which host operation is to be performed. Then write to the **ARG_***N* registers to store arguments to the call. Finally, to actually make the call, read from register **DO_IT**.

ARG_1
ARG_2
ARG_3
ARG_4

These registers are used to pass argument values to a host system call. Store the argument value in them before making the call with **DO_IT**. There is accommodation for up to 4 arguments.

The **LONG_STR** register also functions as an argument register in the same way and should be written before an **fopen** operation.

Ret_Val

If the host operation returns a value, it will be placed in this register. Read this register to retrieve it. Here are the operations that return a value:

operation	returned value
fopen	fileNumber
feof	boolean
fgetc	character
fread1	count read
fwrite1	count written

DO_IT

This register is used to perform the host operation. The act of reading this register will cause a host OS operation to occur. The operation will be determined by **FUN_CODE**, **ARG_1** ... **ARG_4**. Any retuned value will be placed in **Ret_Val**.

The value returned from reading this register will be an error code with the standard Unix/Linux/POSIX meanings, i.e., that value of "**errno**" from the host. Here are the most common codes:

Value	Name	Meaning
0	OK	No error
1	EPERM	Operation not permitted
2	ENOENT	No such file or directory
5	EIO	Input/output error
9	EBADF	Bad file descriptor
12	ENOMEM	Cannot allocate memory
13	EACCES	Permission denied
14	EFAULT	Bad address (invalid address in attempting to use an argument of a call)
17	EEXIST	File already exists
22	EINVAL	Invalid argument
24	EMFILE	Too many open files
63	ENAMETOOLONG	File name too long. A component of a path name exceeded 255 characters.

Note: **DO_IT** always returns a value, and it is often "0=OK". In Unix/Linux/POSIX, the errno value is often unchanged if there is no error. This requires the programmer to set it to zero ahead of time, but this is not required here.

Notes:[23]

EINVAL	Includes "bad whence" (fseek), "bad mode" (fopen)
EBADF	Not an open file, or not open for writing
EACCES	The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of pathname, or the file did not exist yet and write access to the parent directory is not allowed.
ENAMETOOLONG	A component of a path name exceeded 255 characters.

Note: The **HostInterface** package provides a function "**perror**" which prints the typical Unix/Linux/POSIX messages, at least for the error codes listed above.

[23] From the Linux/POSIX online documentation.

LONG_STR

This "register" consists of a sequence of 1,024 bytes.[24] A filename can be stored into these bytes. The **fopen** operation will use this name.

Here are the host operation that can be performed. For more info, consult the Unix/ Linux/POSIX documentation.

FUN_CODE	Operation	Arguments
1	fopen	**arg1**: ptr to KPL array of byte **arg2**: mode (see below) **returns**: fileNumber
2	fclose	**arg1**: fileNumber
3	feof	**arg1**: fileNumber **returns**: bool 1=true=EOF
4	fgetc	**arg1**: fileNumber **returns**: char (0 … 255, -1 = EOF)
5	ungetc	**arg1**: int **arg2**: fileNumber
6	fgets	**arg1**: address **arg2**: fileNumber
7	fread1	**arg1**: address **arg2**: byteCount **arg3**: fileNumber **returns**: bytecount successfully read
8	fwrite1	**arg1**: address **arg2**: byteCount **arg3**: fileNumber **returns**: bytecount successfully written
9	fseek	**arg1**: fileNumber **arg2**: offset **arg3**: whence 1=SEEK_SET, 2=SEEK_CUR, 3=SEEK_END
10	fgetc	**arg1**: fileNumber **returns**: int

Note: For the **mode** argument to **fopen**:

"r" File must exist; position at beginning; Reading only.

"r+" File must exist; position at beginning; Both reading & writing allowed.

[24] See the emulator constant "HOST_DEVICE_BUFFER_SIZE".

“w”	Create file or truncate to zero length; Writing only.
“w+”	Create file or truncate to zero length; Both reading & writing allowed.
“a”	Create file if necessary, otherwise position at file end; Writing only.
“a+”	Create file if necessary, otherwise position at file end; Both reading & writing. (Note with "a+": Check host differences on initial position for reading.)

Note: The **fread1** and **fwrite1** operations access a range of bytes in memory. This is the buffer that holds data that is read or written. This buffer may be in virtual memory and will undergo the usual TLB memory mapping. However, accessing the buffer could cause an exception. These operations will perform TLB mapping, but if a TLB exception occurs, the data transfer will be cut short. These operations are named **fread1** and **fwrite1** (instead of **fread** and **fwrite**) to emphasize this difference from the host operations.

Other Devices

At this date, these are the only memory-mapped I/O devices implemented by the emulator.

Chapter 7: Porting and Host Issues

Command Line Options

The Blitz emulator is a Unix/Linux/POSIX tool run from the command line. Here are the command line options:

filename
The input executable file, which will be loaded into memory. This is optional; if missing, nothing will be loaded and main memory will contain zeros.

-h
Print help info about the command line options. Ignore other options and exit.

-g
This is the "auto-go" option. Automatically begin emulation of the executable program immediately, bypassing the command line interface.

-i filename
File to get serial input from. If missing, **stdin** will be used.

-o filename
File to send serial output to. If missing, **stdout** will be used.

-raw
Places the serial input device in "raw" mode; the default is "cooked" mode. In cooked mode, keystrokes are echoed, backspaces are processed, etc. by the host, relieving the Blitz code is relieved from this task. In raw mode, the running BLITZ code must echo keystrokes, process backspaces, etc.

-nowarn
By default, some potential program execution problems will be flagged and instruction execution will be suspended. This option suppresses this, as well as several informational messages the emulator would otherwise display.

-fp
By default, all floating point instructions will cause an Emulation Exception. This option (fp = "floating present") will cause these instructions to be executed.

-nodebug
Normally the DEBUG instruction will cause a halt to emulation and the emulation debugger will be invoked. With this option, the instruction will cause a Debug Exception and emulation will not be suspended. (This can also be accomplished with the DEBUG_INVOKES_EMULATOR=0 emulation parameter. If both "**-nodebug**" and DEBUG_INVOKES_EMULATOR=1 are present, the command option prevails: the instruction will cause an Exception and emulation will continue.) The BREAKPOINT instruction is treated the same way.

-startall
By default in a multicore Blitz system, only core 0 will run. The remaining cores will be stopped. This option causes all cores to be placed in RUNNING mode. The default is determined by the setting of the START_ALL_CORES emulation parameter. If both "**-startall**" and START_ALL_CORES=0 are present, the command line option prevails: All cores will be RUNNING.

-args string
This is used to pass command line arguments to the running program. For example:

```
blitz MyProgram.exe -g —args "-stack -xxx -o myFile.o"
```

Development on Apple macOS

The Blitz emulator was developed and runs under Apple macOS, which is a POSIX-compliant implementation of Unix. The following well-known tools were used:

gcc
make
shells: **csh**, **sh**
TextEdit
Terminal

All editing was done with Apple's **TextEdit**, which is quite simple and well-designed. Apple's **Terminal** app is used to run the shell[25]. I happen to use the **csh** shell, but the testing suite use scripts that begin with:

```
#!/bin/sh
```

In addition, the following Blitz tools are used:

asm
link
kpl

A few other Blitz tools are occasionally helpful: **hexdump**, **dumpobj**, and **hexify**.

The emulator is compiled with these options:

```
gcc -g -std=c99 -Wall -O2 …
```

This invokes the **clang** compiler[26]. The Xcode IDE (integrated development environment) was not used.[27]

The emulator consists of the following files:

File	Lines of code
`CheckHostCompatibility.c`	937
`BlitzSupport.c`	1,023
`BlitzSupport2.c`	508
`blitz.c`	14,676
Total	17,144

The following **#include** files are used:

---

[25] Here are some of the Terminal settings I use: Profiles>>Advanced>>Declare terminal as xterm-256color,Delete sends Control-H,; Text encoding UTF-8; Profiles>>Text>>Menlo Regular14.

[26] More precisely **clang** version 12.0.0 (clang-1200.0.32.29) with target x86_64-apple-darwin20.4.0, as of this date.

[27] I believe the **clang/gcc** toolchain is distributed as part of Xcode, but may also be downloaded separately.

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>
#include <math.h>
#include <signal.h>
#include <time.h>
#include <fenv.h>
```

The **makefile** will execute the following commands:

```
gcc -g -std=c99 -Wall     -DBLITZ_HOST_IS_LITTLE_ENDIAN          \
                -DWithoutOpt CheckHostCompatibility.c -S         \
                -o CheckHostCompatibility1.s
gcc -g -std=c99 -Wall -O2 -DBLITZ_HOST_IS_LITTLE_ENDIAN          \
                -DWithOpt CheckHostCompatibility.c -S            \
                -o CheckHostCompatibility2.s
gcc -g -std=c99 -Wall -O2 -DBLITZ_HOST_IS_LITTLE_ENDIAN blitz.c -S
gcc blitz.s CheckHostCompatibility1.s CheckHostCompatibility2.s   \
                -lm -o blitz
```

CheckHostCompatibility.c contains a function that is called upon startup to verify the host computer will perform as expected. The emulator is compiled with "-O2" optimization, which will, at compile-time, modify or eliminate many of the operations performed in **CheckHostCompatibility**. Therefore, **CheckHostCompatibility.c** is compiled both with and without -O2 optimization and invoked both ways, in order to catch any problems.

The files **BlitzSupport.c** and **BlitzSupport2.c** are incorporated with **#include**[28]. These files are used in other Blitz tools, including the KPL compiler. The file **BlitzSupport.c** contains material that is only for C programs while **BlitzSupport2.c** contains material that is used in both C and C++ programs.

Host Compatibility: Porting to Windows, Linux

[28] Convention is flouted by using ".c" instead of ".h" as an extension.

The Blitz-64 tools have not (yet!) been ported to Windows or Linux, but I think this should be reasonably straightforward.

Here are the areas of concern:

BigEndian / LittleEndian

Blitz is “Big Endian”, and the tools run on the Mac, which is a x86-64 “Little Endian" architecture. Linux often runs on ARM, which is also Little Endian, so ARM-based hosts should not present a problem, but see the section on “Floating Point Endianness on ARM”.

Porting to a Big Endian host, may require changes. The code contains macros to swap bytes, so *in theory* no changes will be required. In any case, the test suites should uncover any and all problems.

Floating Point Endianness on ARM

The ARM processor stores double precision floating point numbers differently than X86-64. The two 32-bit words are stored in Little Endian order, but in ARM, the most significant word is first, while in x86-64, the least significant word is first.

This will require changes in porting the Blitz tools to an ARM-based computer. The test suites should uncover any and all problems.

Apparently Apple is planning to use ARM cores in its laptops, which will necessitate dealing with this.

C / C++ Compiler

The emulator was compiled with C99 using **clang**. The **-Wall** option was used and no issues are flagged. The emulator should compile using a different compiler, without too much difficulty.

POSIX on Linux

The emulator was developed on Apple macOS, which is POSIX-compliant and thus can correctly be called a “Unix” system.

When porting the Blitz tools to Linux, I don't foresee a lot of problems since most Linux systems are either POSIX-certified, or at least POSIX-compliant. There is no use of threads and most of the interface to the host OS is pretty standard.

The big issue with Linux is likely to be that many Linux machines use ARM processors. See the section above on "Floating Point Endianness on ARM".

POSIX on Windows

There are POSIX packages for Windows, such as **Cygwin** and **Windows Subsystem for Linux (WSL)**. To port BLITZ-64 to Windows, one of these will be needed. This should be workable, but porting may present some surprises.

Host Interface

The interface with the host OS can be broken into these areas:

Argument processing (**argc**, **argv**)
Memory allocation (**calloc**)
Termination (**exit**)
File services (**fopen**, **fgetc**, **fflush**, **perror**, …)
Time services (**ctime**)
Double precision (**feclearexcept**, **fetestexcept**)
Control-C handling (**signal**, **SIGINT**)
Raw/Cooked input (**system**, **stty**)

The last items are most likely to cause porting issues.

Integer Division

The **CheckHostCompatibility** function will verify that the host integer division operation implements "truncated division". For a host processor where this is not the case, changes will be needed.

Floating Point Rounding

Presumably the host implements the IEEE-754 standard, with the default rounding mode being "round-to-nearest-with-ties-to-even".

However, if the host processor performs rounding or sets the floating point exception flags (NV-invalid, NX-inexact, OF-overflow, UF-underflow, DZ-divide-by-zero) in an unexpected way, this should be detected when running the KPL "execution" test suite or the "NumberTest" KPL program.

Chapter 8: BlitzHEX1, BlitzHEX2, and Hexify

Quick Summary

- Two file formats are described: **BlitzHEX1** and **BlitzHEX2**.
- The content of these file types is simple:
  - — Load address
  - — Entry point
  - — Bytecount
  - — The bytes to be loaded
- The information content of the two formats is identical.
  - — A **BlitzHEX2** (**.bhex2**) file is ASCII and human-readable.
  - — A **BlitzHEX1** (**.bhex1**) file is binary and about half the size.
- The file can be loaded and executed as-is.
- This format can be used for memory dumps (image files).
- The **hexify** tool is described.
  - — Input: An object (**.o**) or executable (**.exe**) file.
  - — Output: Either format can be produced.
- These file formats are intended to support hardware development.
- The emulator recognizes these file formats.

Introduction

Normally, a program will be assembled to produce an object (**.o**) file. Then, one or more object files will be linked to produce an executable (**.exe**) file. The executable file can be loaded in to memory and executed, either by the emulator or a running Blitz OS.

If this is what you are doing, then you can skip this chapter. However, if you are developing hardware, it may be necessary to work directly with the data bytes. The

formats of object (.o) and executable (.exe) files are complex. The formats present here are much simpler and the **hexify** tool is designed to simplify your life.

To capture a memory image, a simple file format is introduced here. In general, memory image files are called "BlitzHEX" files, but there are two specific file formats introduced here, called "**BlitzHEX1**" and "**BlitzHEX2**". Both file formats carry exactly the same information and you can use whichever format is most convenient.

Files in the **BlitzHEX1** format are given an extension of **.bhex1** and files in the **BlitzHEX2** format are given an extension of **.bhex2**.

In a **BlitzHEX1** file, all values are in binary. Thus, a 64 bit doubleword requires 8 bytes in the file.

In a **BlitzHEX2** file, all values are expressed in hex with ASCII characters and each is placed on a separate line. Thus, a 64 bit doubleword requires 17 bytes in the file (16 hex characters, plus a \n NEWLINE character).

A **BlitzHEX2** file is human readable, while a **BlitzHEX1** is not so easily read. Although a **BlitzHEX2** file is a text file that can be easily viewed, it contains nothing but hex values, so it may not be easily understood.

> **BlitzHEX1** — Benefit: The file size is smaller, about half the size.
> **BlitzHEX2** — Benefit: The file is a human-readable text file.

Each file contains the following information:

- Load address
- Entry point
- Byte count
- Data Checksum
- Header Checksum
- Data bytes

Both a **BlitzHEX1** and **BlitzHEX2** file begin with a "magic number".

The first 8 bytes of a **BlitzHEX1** file will be:

In Binary	***In ASCII***
`426c747a48455831`	`BltzHEX1`

The first 8 bytes of a **BlitzHEX2** file will be:

In Binary	***In ASCII***
`426c747a48455832`	`BltzHEX2`

Any program accepting a BlitzHEX file will first read the first 8 bytes to determine which format the remainder of the file is in.

A BlitzHEX file can be viewed as a program that is to be executed. The idea is that the **data bytes** should be loaded into memory at the **load address**. The **byte count** tells how many **data bytes** are present in the file. The program can be executed by jumping to the **entry address**. The **checksums** can be used (if desired) to make sure that no data corruption has occurred.

The file can also be viewed as a **memory dump** (or "**image**") of memory, possible captured after some program crashed. The **load address** and the **byte count** describe which region of memory has been captured. The **data bytes** contain the data copied from memory. The **entry address** is not needed and a value of -1 indicates that the **entry address** is missing.

There are two ways to create a BlitzHEX file.

In the first approach, the user will create a simple assembly program. This program must be stand-alone, in the sense that it does not need to be linked. Then, the **hexify** tool is used. The assembler produces an object (.o) file and the **hexify** tool can read such a file. The tool will then produce the BlitzHEX file as its output. Of course the **hexify** tool will verify that the object (.o) file meets certain requirements and does not need to be linked.

The second approach accommodates larger and more complex programs. The program may consist of a number of modules — both compiled KPL packages and hand-coded assembly files — which are assembled and then linked to produce an executable. After linking, the **hexify** tool is used. The linker produces an executable (.exe) file and the **hexify** tool can read such a file. The **hexify** tool will then produce the BlitzHEX file as its output.

Normally, the emulator will be used with the name of an executable (.exe) file on the command line, but the emulator will accept files in the **BlitzHEX1** and **BlitzHEX2** format as well. For example:

```
Shell% blitz MyExamplePgm.bhex1 -g
Reading executable file...
Executable file is a "BlitzHEX1" format file...
  Load Address: 0x000001208
  Entry Point:  0x000001210
  Byte Count:   0x000000220
Beginning execution...
```

The emulator will load the data bytes into memory. If the region of memory is private memory and there are multiple cores, then a copy of the data will be loaded into the private memory of each core. If the region of memory is shared memory, then the data bytes will of course be available to all cores. If the region is in the ROM Memory-Mapped I/O pages, then the data bytes will be loaded in to the ROM. All cores share the same ROM data. The **EntryPoint** will be stored in the Program Counter (PC) of each core.

In this chapter, we specify the file formats more precisely and discuss the **hexify** tool.

BlitzHEX1 File Format

This is a binary file with an extension of "**.bhex1**".

The file consists of a sequence of doublewords, where each doubleword is given in binary, with 8 bytes. The file contains:

bytes	value	description
8	**426c747a48455831**	"BltzHEX1" in ASCII
8	**2a2a2a2a2a2a2a2a**	"********" in ASCII
8	***LoadAddress***	where in memory to place data (multiple of 8)
8	***EntryPoint***	where to begin execution (multiple of 8) -1 if **EntryPoint** is missing
8	***ByteCount***	N = number of data bytes (multiple of 8)
8	***DataChecksum***	Logical XOR of all data doublewords
8	***HeaderChecksum***	Logical XOR of **LoadAddress**, **EntryPoint**, **ByteCount**, and **DataChecksum**
8	**2a2a2a2a2a2a2a2a**	"********" in ASCII
N	***Data***	N bytes; N will be a multiple of 8

8 **2a2a2a2a2a2a2a2a** "********" in ASCII

A **separator** consists of 8 bytes giving the ASCII codes for "********", that is, the value 0x2a2a2a2a2a2a2a2a. The file contains three separators which serve to make sure there are no errors in the interpretation of the file.

The **LoadAddress** and the **EntryPoint** must be valid Blitz addresses, which means a 36-bit value, i.e., within

0x0000_0000_0000_0000 … 0x0000_000F_FFFF_FFFF

Furthermore, they must be doubleword aligned and must not be zero, since any attempt to access memory bytes below 0x0_0000_0008 will cause a Null Address Exception.

The **ByteCount** must be a number evenly divisible by 8.[29] It is considered an error if **LoadAddress** + **ByteCount** exceeds the maximum address.[30]

The **EntryPoint** must line within the memory region described by **LoadAddress** and **ByteCount**.

An **EntryPoint** value of -1 is also allowed. This value means default/missing/not-applicable.

The **DataChecksum** is a 64 bit value that is produced by Exclusive-ORing (XOR) all of the data doublewords together.

The **HeaderChecksum** is a 64 bit value that is produced by Exclusive-ORing (XOR) the following doublewords together:

LoadAddress
EntryPOint
ByteCount
DataChecksum

[29] If the desired number of bytes is not an even multiple of 8, the **ByteCount** must be rounded up to the next multiple and zeros must be added to the data bytes to bring the data up to an integral number of doublewords. Likewise, if the desired starting address is not doubleword aligned, **LoadAddress** must be adjusted accordingly. In practice, these exceptions do not arise.

[30] More precisely, **LoadAddress** + **ByteCount** must be 0x0000_0010_0000_0000 or less.

The **Data** region of the file consists of an integral number of doublewords, of the size indicated by **ByteCount**.

Here is an example of a file in **BlitzHEX1** format:

```
Shell% hexdump example.bhex1
000000000:  426C 747A  4845 5831  2A2A 2A2A  2A2A 2A2A    BltzHEX1********
000000010:  0000 0000  0000 0008  0000 0000  0000 0010    ................
000000020:  0000 0000  0000 0020  9A70 214B  2266 11DD    ....... .p!K"f..
000000030:  9A70 214B  2266 11E5  2A2A 2A2A  2A2A 2A2A    .p!K"f..********
000000040:  1122 3344  5566 7788  8877 6655  4433 2211    ."3DUfw..wfUD3".
000000050:  1111 2222  3333 4444  1234 5678  0000 0000    .."33DD.4Vx....
000000060:  2A2A 2A2A  2A2A 2A2A                         ********
Shell%
```

BlitzHEX2 File Format

A file in the **BlitzHEX2** format is an ASCII text file with an extension of "**.bhex2**".

The file consists of a sequence of lines, where each line contains ASCII characters followed by a NEWLINE. Like any text file, the file can be printed.

Here is an example of a **BlitzHEX2** file, originating from the same source as the **BlitzHEX1** example above and containing the same information:

```
BltzHEX2
********
0000000000000008
0000000000000010
0000000000000020
9a70214b226611dd
9a70214b226611e5
********
1122334455667788
8877665544332211
1111222233334444
1234567800000000
********
```

In a **BlitzHEX2** file, each line contains only ASCII characters. Each line is terminated with a NEWLINE[31]. Each line — other that the first line and the three "********" separators — contains exactly 16 hex characters and specifies a 64-bit doubleword value. The file must contain no spaces or tabs.

Regardless of whether the file is in **BlitzHEX1** or **BlitzHEX2** format, the exact same information is conveyed. Files in **BlitzHEX1** format are smaller — about half the size — and files in **BlitzHEX2** format are easier for the human to read.

Below we show this file in another way to see the actual bytes, but you can see that printing it as a text file (as above) is clearly preferable.

```
Shell% hexdump example.bhex2
000000000:  426C 747A  4845 5832  0A2A 2A2A  2A2A 2A2A    BltzHEX2.*******
000000010:  2A0A 3030  3030 3030  3030 3030  3030 3030    *.00000000000000
000000020:  3038 0A30  3030 3030  3030 3030  3030 3030    08.0000000000000
000000030:  3031 300A  3030 3030  3030 3030  3030 3030    010.000000000000
000000040:  3030 3230  0A39 6137  3032 3134  6232 3236    0020.9a70214b226
000000050:  3631 3164  640A 3961  3730 3231  3462 3232    611dd.9a70214b22
000000060:  3636 3131  6535 0A2A  2A2A 2A2A  2A2A 2A0A    6611e5.********.
000000070:  3131 3232  3333 3434  3535 3636  3737 3838    1122334455667788
000000080:  0A38 3837  3736 3635  3534 3433  3332 3231    .887766554433221
000000090:  310A 3131  3131 3232  3232 3333  3333 3434    1.11112222333344
0000000a0:  3434 0A31  3233 3435  3637 3830  3030 3030    44.1234567800000
0000000b0:  3030 300A  2A2A 2A2A  2A2A 2A2A  0A          000.********.
Shell%
```

To be more precise, every **BlitzHEX2** file has the following format:

- "`BltzHEX2`" — 8 characters (ASCII: `42_6c_74_7a_48_45_58_32`)
- "`********`" — Separator
- ***LoadAddress*** — 16 characters, 0 … 0000000FFFFFFFFF (multiple of 8)
- ***Entrypoint*** — 16 characters, 0 … 0000000FFFFFFFFF (multiple of 8)
  FFFFFFFFFFFFFFFF if **EntryPoint** is missing
- ***ByteCount*** — 16 chars, N = number of data bytes (multiple of 8)
- ***DataChecksum*** — 16 chars, Logical XOR of all data doublewords
- ***HeaderChecksum*** — 16 chars, Logical XOR of **LoadAddress**, **EntryPoint**, **ByteCount**, and **DataChecksum**
- "`********`" — Separator
- ***Data*** — Each line contains 16 hex chars, i.e., a doubleword

[31] Normally, the END-OF-LINE will be \n, but the **hexify** tool allows the END-OF-LINE to be \r, \n\r, or \r\n instead.

- "********" — Separator

The fields are discussed in the section describing the **BlitzHEX1** format. The same comments apply to files in the **BlitzHEX2** format.

The hex characters may be lower or upper case.

Hexify

The tool **hexify** is a simple program that can be used to convert the format of a file. The input file can be in either of these formats:

- Object file (.o)
- Executable file (.exe)

The output from the tool will be in one of these formats:

- **BlitzHEX1** format
- **BlitzHEX2** format
- "Hex" format
- System Verilog statements

The **hexify** tool determines the format of the input file by first reading its "magic number". The format of the output file is determined by a command line option.

The input comes either from stdin or from a file which is named on the command line.

Exactly one of the following command line options must appear:

option	***output file format***
`-bhex1`	**BlitzHEX1** format
`-bhex2`	**BlitzHEX2** format
`-hex`	"Hex" format
`-sv`	System Verilog statements

The output will go to **stdout**.

The **hexify** program performs error checking on the input and will print warnings or error messages. It will also print general information about the file

Here is an example usage:

```
Shell% hexify baby.o -bhex2 > example.bhex2
The input is a ".o" file...
  ISA Version:          1
  .o Version:           1
  Number of Segments:   1
  Number of Symbols:    3
  Source Filename:      baby.s
  Line number:          7
  Initial Length:       0x00000001c
  Status:               Kernel Mode,  executable,  writable,  not zero-filled
  Reg gp value:         <undefined>
  Lowest Address:       0x000000008
  Highest Address:      0x000000027
  Size in bytes:        0x000000020 (decimal 32)
  Entrypoint:           0x000000010
Computing checksums...
  Checksum 8  = 0x08
  Checksum 64 = 0x9a70214b226611dd
Producing BlitzHEX output...
Shell%
```

Here is a summary of the command line options:

`filename`
The input will come from the named file. If a file is not given on the command line, the input will come from **stdin**. Only one input file is allowed and it should be in "**.o**" or "**.exe**" format.

`-hex`
Produce output file in "HEX" file format.

`-bhex1`
Produce output file in **BlitzHEX1** format (Binary).

`-bhex2`
Produce output file in **BlitzHEX2** format (ASCII).

`-sv`
Produce output file as System Verilog statements.

`-silent`
Suppress informational output.

`-nowarn`
Suppress warning output.

`-bigok`

For really large files, keep going and do not abort.

-r
End each output line with **\r**. The default is **\n**.

-nr
End each output line with **\n\r**. The default is **\n**.

-rn
End each output line with **\r\n**. The default is **\n**.

-h
Print help info, ignore other options, and terminate.

All warnings, errors, and informational output are directed to **stderr**. If there is an error, the program will terminate with an exit code of 1.

The **-silent** option causes the tool to suppress all informational output. For example:

```
Shell% hexify baby.o -bhex2 -silent > example.bhex2
Shell%
```

The **-nowarn** option causes the tool to suppress all warnings.

If the **ByteCount** is really large, then **hexify** will abort with an error. The **-bigok** option will suppress this check. (The limit is set at 10,000,000 bytes.)

Blitz follows the Unix/Linux/POSIX conventions and lines in a text file end with the \n character (0x0a). However, for FPGA development, it may be necessary to an OS with a different convention. By default, **hexify** uses a "\n" for END-OF-LINE. This can be switched by specifying **-r**, **-rn**, or **-nr** on the command line.

The **-h** option is provided for forgetful users[32] and will print out a short summary of which options do what.

Input Requirements

Object (.o) and executable (.exe) files normally contain debugging information. The **hexify** tool ignores this information.

[32] Me.

If the input is an object (.o) file, then **hexify** checks to make sure the file contains a single segment, i.e., that the source file contained only one **.begin** statement. It also makes sure it contains no relocatable data, which would require the linker's involvement.

Output Form: System Verilog

The -sv command line option can be used to create output that can be included in a System Verilog program.

For example, consider the following file:

```
   .begin        kernel, startaddr=0x000001200, gp=undefined
_entry:
   .export       _entry
   add           r1,r2,r3
   jump          _entry
   .doubleword   0x0011223344556677
   .doubleword   0x8899aabbccddeeff
   .doubleword   0x1111222233334444
   .string       "hello world"
```

We can convert it into System Verilog statements as follows:

```
Shell% asm romExample.s
Shell% hexify romExample.o -silent -sv
      // The following statements were generated by the "hexify" tool
      //         from "romExample.o" on Mon May 17 18:26:03 2021

      36'h000001200: data_val = 64'h00010321_19ffffc0;
      36'h000001208: data_val = 64'h00112233_44556677;
      36'h000001210: data_val = 64'h8899aabb_ccddeeff;
      36'h000001218: data_val = 64'h11112222_33334444;
      36'h000001220: data_val = 64'h68656c6c_6f20776f;
      36'h000001228: data_val = 64'h726c6400_00000000;
Shell%
```

Output Form: HEX File Format

The **hexify** tool can also produce its output is a format we shall call "**HEX format**", using the **-hex** command line option.

Using the same file as above, here is an example:

```
Shell% hexify romExample.o -silent -hex
00000000
00000030

00000000
00001200

00010321
19ffffc0
00112233
44556677
8899aabb
ccddeeff
11112222
33334444
68656c6c
6f20776f
726c6400
00000000

da
Shell%
```

Note: This format is not the "**Intel HEX**" format.

Note: The emulator cannot read data in this format.

Note: This format will be discontinued and removed. At this time, it is only used by the boot loader **MBBooter.s** running on my FPGA.

This file is ASCII and contains only hex digits and NEWLINE characters. Each doubleword is expressed as two 32 bit words on separate lines.

The first doubleword is the **ByteCount**. The second doubleword is the **LoadAddress**. There is no **EntryPoint**; execution is assumed to begin with the first byte, so **EntryPoint** and **LoadAddress** are equal.

The final line contains a single byte, which is the **Checksum** of all the data bytes. The **Checksum** is used to make sure the transfer completes correctly and **MBBooter** will complain if there is a mismatch with the value it computes from the data bytes.

About This Document

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 is used. The document history is:

Date	Author
3 May 2021	Harry H. Porter III <document created>
17 June 2021	Harry H. Porter III
18 October 2022	Harry H. Porter III <current version>

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.

Corrections and Errors

Please contact the author if you find…

- Inaccurate information that you can correct
- Incomplete information that you can fill in
- Confusing text that needs to be reworded

Thanks!

Recent Changes

This appendix documents recent changes to the Blitz-64 emulator and this document.

3-23 May 2021

This document was created.

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.cecs.pdx.edu/~harry