An Overview of the
BLITZ Computer Hardware

Harry H. Porter III
Department of Computer Science
Portland State University

Introduction

The BLITZ computer was designed solely to support the development of operating system kernel code by students in a university-level OS course. The BLITZ computer is not a real, physical machine. Instead, the hardware is emulated entirely in software.

This document introduces the BLITZ computer "hardware" and describes it in broad, general terms. Greater detail can be found in the document "The BLITZ Architecture."

The BLITZ machine is modeled loosely on the Sun SPARC architecture. The reader is assumed to be familiar with the main concepts of CPUs and assembly language programming in general, but not with any specific CPU architecture in particular.

Students who use the BLITZ system for OS kernel development will not need to write a single line of assembly code. The necessary assembly routines are provided by the instructor. Nevertheless, students need an understanding of the BLITZ hardware so they can look over and understand the assembly routines.

User Mode and System Mode

At any time, the CPU is executing in either system mode or user mode. Kernel code executes in system mode, while user-level programs execute in user mode.

Any instruction may be executed in system mode, but some instructions are privileged in the sense that they may only be executed in system mode. Examples of privileged instructions include the instructions to alter the page tables and to change the mode itself.

Registers

The CPU is a 32-bit architecture. There are 16 general purpose registers (called r0 through r15) with 32 bits each. In addition, there are 16 floating point registers (called f0 through f15) which can each hold a double precision floating point value. The main types of data are bytes, 32-bit signed integers, and 64-bit floating point values.

There are 2 sets of the integer registers, one for system mode and one for user mode. This reduces the time for context switches between user code and kernel code, since user registers need not be saved.

Additional registers include a program counter (PC); a status registers (SR), and two registers related to the page table.

Instructions

The BLITZ CPU can be programmed in either a high level language called "KPL" or in assembler. Included in the BLITZ tool set are a compiler, an assembler, a linker, and a debugger. All BLITZ tools run on the Unix platform.

The assembly language resembles the Sun SPARC assembly language. Below are some sample instructions. In assembly code, comments follow an exclamation point.

```
add     r2,r3,r4            ! 32-bit addition
mul     r2,ELT_SIZE,r2      ! multiply
sll     r2,1,r2             ! shift left
and     r2,0x000000ff,r2    ! bitwise logical and
cmp     r2,r3               ! compare and set status register
bl      mylabel             ! branch if less than
neg     r2,r3               ! arithmetic negation
bclr    r2,0x00000c01,r3    ! clear selected bits
mov     r2,r4               ! move between regs
mov     123,r5              ! load 16-bit value into reg
set     MyVar,r2            ! load 32-bit value into reg
load    [r2],r7             ! load 32-bits from memory
loadb   [r2],r7             ! load byte from memory
store   r7,[r2]             ! store 32-bits into memory
loadv   [r2],r7             ! load using virtual memory address
call    foo                 ! invoke a function
jmp     mylabel             ! goto
ret                         ! return
reti                        ! return from interrupt
push    r3                  ! push onto stack
pop     r3                  ! pop
ldaddr  MyVar,r2            ! load address of MyVar into reg
syscall r2                  ! system call trap
seti                        ! enable interrupts
```

```
setp                        ! turn on paging
clears                      ! switch to user mode
tset    [r2],r3             ! test and set
readu   r2,r5               ! read from user register set
ldptbr  r2                  ! load page table base register
ftoi    f3,r5               ! floating to integer conversion
fadd    f2,f3,f4            ! floating-point add
fcmp    f2,f3               ! floating-point compare
fsqrt   f2,r3               ! floating-point square root
fload   [r2],f3             ! load from memory to floating reg
nop                         ! do nothing
wait                        ! enter low-power wait state
debug                       ! to facilitate student debugging
```

The above list is not exhaustive. In total, there are 69 different instructions plus 9 "synthetic instructions," giving a total of 78 instructions. (A synthetic instruction is translated by the assembler into one of the existing 69 instructions.) The above list contains only 40 of the 78 instructions.

There are 114 distinct numeric opcodes. This number is greater than 69 because many of the instructions have several forms. For example, the "load" instruction is conceptually one instruction, although it has several forms, with different numeric opcodes.

```
load    [r2+r3],r7
load    [r2+1000],r7
load    [r2],r7
```

The assembler accepts a number of directives (sometimes called "pseudo-ops"). Here are some examples:

```
.ascii   "hello\n"
.byte    ('a'+4)&0x0f
.word    MyLabel+4
.double  -12.34e-56
.align
.skip    1000
.data
.text
.bss
.export  MyLabel
.import  ForeignProc
ELT_SIZE = 120
```

Alignment

Like many computers, the BLITZ requires data to lie on aligned memory addresses. The BLITZ requires 32-bit alignment for word-sized quantities.

Real computers, like the Sun SPARC, have additional 16-bit and 64-bit alignment requirements for some instructions. This exemplifies how the BLITZ architecture is designed to capture many details of real CPU architectures, while at the same time being somewhat simpler to avoid many of the technical complexities that can make programming real CPUs quite difficult.

The Stack Pointer

In the BLITZ architecture, register "r15" points to the execution stack, which grows downward from higher memory addresses toward lower addresses. Since there are two sets of the general purpose registers (one for system mode and one for user mode), there is a distinction between a process's system stack and its user stack.

Interrupt Processing

The CPU accepts asynchronous interrupts from the following hardware and external sources:

```
Timer
Disk
Serial Terminal
Power-on Reset
Hardware Fault
```

The CPU also processes program exceptions arising from the following causes:

```
Address Exception
Alignment Exception
Arithmetic Exception
Privileged Instruction
Illegal Instruction
Page Fault (invalid page)
Page Fault (update to read-only page)
Syscall Trap Instruction
```

The timer interrupt is generated regularly at a fixed periodic rate. It provides the necessary "clock tick" needed for a number of OS functions, such as time-slicing.

Some of the interrupts may be masked while others cannot be. For example, the timer interrupt may be masked, which is useful in preventing a context switch while the OS kernel is updating data in a critical section of code.

When an interrupt occurs, the CPU will switch from user mode into system mode, disable interrupts, turn off paging, and push three words onto the system stack. The words pushed onto the stack are:

```
Program Counter
Status Register
Page number causing problem (for page faults only)
```

The lowest 56 bytes of memory contain an interrupt vector, comprised of one 4 byte word per interrupt type. When an interrupt occurs, the interrupt vector is consulted and a jump is made to the interrupt handler routine. There is a separate interrupt handler for each type of interrupt.

All interrupts are precise in the sense that the OS can simply execute a return-from-interrupt instruction ("reti") to return to the user-level program and restart the offending instruction. Since there is no pipelining or super-scalar execution, the OS does not need to do any work to restart interrupted execution after a page fault.

The Page Table

The CPU includes a MMU (Memory Management Unit) to support paging and virtual memory. The page size is 8K bytes. A page table can accommodate up to 2K pages, allowing each virtual address space to be up to 16M bytes. Each virtual address is therefore 24 bits, which are divided into an 11 bit page number and a 13 bit offset.

In a typical student OS project, physical memory is set to 16 MB (2K frames) and a typical virtual address space is 20 pages (160 KB). However, physical memory can be as large as 4G bytes (512K frames).

Each entry in the page table contains:

```
Page Frame Address in Physical Memory (19 bits)
Valid Bit      (0 = page not in memory)
Writable Bit   (0 = page is read-only)
Dirty Bit      (0 = page has not been modified)
Referenced Bit (0 = page has not been accessed)
Unused (9 bits, can be used by the OS as needed)
```

The CPU contains two registers which together describe the current page table:

```
Page Table Base Register (PTBR)
Page Table Length Register (PTLR)
```

Paging is either turned on or is disabled, as determined by a bit in the CPU status register. When paging is enabled, all addresses go through page table translation.

The "Valid Bit" is checked and a page fault is generated whenever a user program attempts to use an address not in its virtual address space. The "Writable Bit" is checked and a page fault occurs whenever a user program tries to modify a read-only page. The MMU will set the "Dirty

Bit" whenever a page is modified and will set the "Referenced Bit" whenever a page is read or modified.

The Disk

The BLITZ emulator simulates a disk. The disk is divided into a number of tracks and sectors. Disk sectors have the same size as page frames, namely 8K bytes. Each track on the disk contains 16 sectors and the number of tracks on the disk is one of the simulation parameters to the emulator.

The disk is simulated with a Unix file. A disk read will therefore get data from the Unix file and disk writes will update the file.

The disk operates asynchronously: the software must start the disk operation and later, after the I/O completes, the CPU will receive an interrupt.

The disk is operated by moving control words to and from several registers in the disk device controller. For example, to start a disk read, the OS must (1) move the desired sector number into one device register, (2) move the target memory address to another device register, (3) move the number of sectors to be read into a third device register, and finally (4) move the "read" command into a fourth device register. After the interrupt signals completion of the disk read, the OS can examine a device status register to see whether the operation completed without error.

The device registers are mapped into high memory addresses. To load a device register, the OS simply stores a value into a fixed, predetermined memory location. This makes it possible to control the disk completely within the high-level language (i.e., from KPL code); no assembly programming is needed.

The transfer of a block of data bytes occurs through direct memory access (DMA).

The BLITZ emulator simulates disk I/O delays accurately, modeling rotational delays, seek times, settle times, transfer times, current rotational angle and current head position. The simulation constants can be modified and random variations are also modeled. The emulator also simulates randomly occurring transient read and write errors.

The Serial Terminal Device

The BLITZ computer also simulates a serial terminal device. The emulator models a Universal Asynchronous Receive Transmit (UART) chip, such as would be used to interface to a modem or dumb terminal.

The serial terminal interface allows the CPU to send and receive individual ASCII characters asynchronously. Only one character may be sent at a time and each character is received one at a time. The serial device is "full duplex," which means that the send and receive channels are independent, so the device can be sending and receiving simultaneously.

To send a character, the OS software must load a device register with the character to transmit. After the transmission is complete and the device is ready for another character, the CPU will receive an interrupt.

To receive a character, the OS must wait until a character appears on the input. When a character appears, the CPU will receive an interrupt. The character can be retrieved by reading a device register. All device registers are memory-mapped to predefined addresses in high memory.

The emulator simulates the serial device by sending characters directly to the Unix "stdout," which generally means that when the BLITZ OS kernel sends a character, it will appear on the student's screen. Likewise, when the student types a character, the emulator will arrange for the BLITZ CPU to be interrupted to indicate that a byte is ready to be received from the UART interface.

For automated testing of student OS code, the serial I/O may also be redirected. If serial input is coming from a file, the emulator simulates a reasonable human typing speed by providing the interrupts associated with incoming characters at a predetermined, rather slow rate.

Nondeterminism and Debugging

The BLITZ emulator simulates an number of probabilistic events using a pseudo-random number generator. For example, the likelihood of a transient disk error and a slight random variation in the arrival of timer interrupts are simulated using random numbers.

Programs that are sensitive to external events and behave slightly differently on each run can be difficult to debug, since a bug may be timing dependent and not easily reproducible. Operating systems certainly fall into this category.

To facilitate testing, the emulator takes, as a command line option, an initial seed for its random number generator. By supplying the same random number seed, students can rerun their programs and always get the exact same output. When supplying a different random number seed, the same program may produce a different output, due to race conditions, timing dependencies, etc.

What is Missing from the BLITZ Hardware

The following features are found in real CPU architectures, but have not been included in the BLITZ hardware architecture. These features add complexity to the job of creating an operating system, yet their inclusion would not substantially change the nature of the OS or contribute to a deeper understanding of the relevant concepts and issues. Including these features would make the students' job of creating a kernel more difficult, without giving the students anything more than an appreciation of how difficult and complex a real OS can be.

- Pipelined instruction execution
- Superscalar or out-of-order instruction execution
- Instruction or data caches
- Interrupts that save internal (hidden) CPU registers
- Translation lookaside buffers (TLBs)
- Graphical interface devices, such as a bitmapped screen, keyboard, and mouse
- Other I/O devices, including multiple disks, CD-ROMs, etc.
- Complexities of real disk drives and device drivers
- Networking capability
- Multiple interrupt levels (BLITZ supports only "enabled" and "disabled")
- Register windows, such as on the Sun SPARC
- Other data types, such as half-word integers and single or quad precision floating-point
- Additional alignment restrictions
- More detailed instruction sets, with more opcodes
- Support for multiple CPUs

One might reasonably ask why the BLITZ architecture does not include Translation Lookaside Buffers (TLBs) instead of page tables, since TLBs are a great idea, yet they impact the OS design significantly. My answer is that implementing virtual memories is hard enough with page tables and the student OS project would be made overly complex by the inclusion of TLBs.

In retrospect, the inclusion of floating point in the BLITZ architecture seems unnecessary, since an OS will not normally perform any floating-point computation.

Networking capability is something that might be useful and may be added in the future. However, in a single term or semester course, very few students will have time to add something like TCP/IP support to the operating system kernels they construct.