

The BLITZ Architecture

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

Abstract

This document provides an overview of the BLITZ architecture and instruction set. The BLITZ microprocessor is designed specifically to support Computer Science education. The architecture is similar to contemporary RISC microprocessor architectures, but is simplified appropriately for student use. The BLITZ processor will normally be implemented by software emulation, although a hardware implementation of the CPU architecture presented here is also possible.

Basic Terminology

In this document, we use the terms “byte”, “halfword”, “word”, and “doubleword” to refer to various sizes of binary data.

	number of bytes =====	number of bits =====	example (in hex) =====
byte	1	8	A4
halfword	2	16	C4F9
word	4	32	AB12CD34
doubleword	8	64	01234567 89ABCDEF

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

7654	3210
====	====
0000	0000

The bits within a 16-bit halfword are numbered from 0 (least significant) to 15 (most significant).

15	12	8	4	0
====	====	====	====	====
0000	0000	0000	0000	

The bits within a 32-bit word are numbered from 0 (least significant) to 31 (most significant).

31	28	24	20	16	12	8	4	0
====	====	====	====	====	====	====	====	====
0000	0000	0000	0000	0000	0000	0000	0000	

The bits within a 64-bit doubleword are numbered from 0 (least significant) to 63 (most significant).

The BLITZ Architecture

63	60	...	20	16	12	8	4	0
====			====	====	====	====	====	====
0000	...		0000	0000	0000	0000	0000	0000

A single hex digit can be used to represent 4 bits (half a byte, sometimes called a “nibble”), as follows.

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

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

8-bit byte	In Hex
=====	=====
1010 0100	A4

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

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

The 64 bits (8 bytes) in a doubleword are conveniently expressed with 16 hex digits.

Other architectures use “quadwords” of 128 bits, but the BLITZ architecture does not provide instructions for manipulating quadword data.

Memory

Main memory is byte addressable. Addresses are 4 bytes (i.e., 32 bits) long, allowing for up to 4 gigabytes to be addressed.

Memory can be viewed either as a sequence of bytes:

The BLITZ Architecture

address (in hex)	data (in hex)
=====	=====
00000000	89
00000001	AB
00000002	CD
00000003	EF
00000004	01
00000005	23
00000006	45
00000007	67
...	...
FFFFFFFC	E0
FFFFFFFD	E1
FFFFFFFE	E2
FFFFFFF	E3

or as a sequence of words:

address (in hex)	data (in hex)
=====	=====
00000000	89ABCDEF
00000004	01234567
...	...
FFFFFFFC	E0E1E2E3

Both are equivalent and show the same memory contents.

Alignment

A “word-aligned address” is an address that is a multiple 4. The last 2 bits of an aligned address will always be zeros.

Word-sized data and doubleword-sized data must always be stored in word-aligned memory locations. Every instruction is one word (i.e., 4 bytes) long. All instructions must be stored at word-aligned addresses.

Any violation of these alignment requirements will result in an error exception. For example, the load and store instructions will cause an error exception if an attempt is made to move a word to or from an address that is not divisible by 4.

Byte-sized data may be stored at any location in memory; there is no alignment requirement.

This architecture uses only word-alignment; there is no “halfword” or “doubleword” alignment” requirement. Henceforth, whenever we say “alignment”, it is understood that we mean “word-alignment”.

Big Endian

The BLITZ architecture is “Big Endian”, meaning that when a word-sized quantity is stored in 4 bytes of memory, the most significant byte is stored in the first byte (i.e., lowest numbered address) and the least significant byte is stored in the last byte (i.e., the byte with the greatest

The BLITZ Architecture

address). In other words, if a word is stored at address x , the most significant byte will be stored in byte x and the least significant byte will be stored in address $x+3$.

Some other (non-BLITZ) computers use “Little Endian” order, in which the least significant byte is stored in the lowest numbered address.

Load Store Architecture

The BLITZ uses a “load-store” architecture. Operations (such as arithmetic and logical functions) can only be performed on data stored in registers. It is necessary to move the data from memory to registers (load it) and back to memory (store it) with separate instructions. The instructions in the following example are not exactly BLITZ instructions, but they give you the idea of load-store architectures. Here, “ x ”, “ y ”, and “ z ” are the names of some memory locations and “ $r1$ ”, “ $r2$ ”, and “ $r3$ ” are the names of registers.

```
load    x,r1
load    y,r2
add     r1,r2,r3
store   r3,z
```

The first instruction moves data from memory into register “ $r1$ ”. The second instructions moves a number into register “ $r2$ ”. The third instruction adds the values in “ $r1$ ” and “ $r2$ ”, placing the result into register “ $r3$ ”. The final instruction moves data from register “ $r3$ ” back to main memory.

BLITZ is a Reduced Instruction Set Computer (RISC) architecture. In a RISC architecture, each instruction is kept very simple, so that instructions may be executed at a fast clock rate. Of course, certain operations are inherently complex, so a couple of instructions may be required in a RISC machine, whereas in another architecture the same operation may require only a single instruction.

Each BLITZ instruction either does a memory access or does a computation. The idea is that each instruction will be executed in one machine (clock) cycle. The goal is to keep each instruction simple so that a faster clock rate can be used, thereby speeding the overall performance of the CPU.

(The BLITZ machine is expected to be emulated in software. When emulated, the RISC architecture may not translate into any increase in speed.)

The BLITZ Register Set

The state of the BLITZ CPU is given by the following registers.

The BLITZ Architecture

User Registers:

=====

r0 - Always zero
r1
r2
...
r13
r14 - Frame Pointer
r15 - Stack Pointer

System Registers:

=====

r0 - Always zero
r1
r2
...
r13
r14 - Frame Pointer
r15 - Stack Pointer

Floating-Point Registers:

=====

f0
f1
f2
...
f15

Miscellaneous Registers:

=====

PC - Program Counter (32 bits)
PTBR - Page Table Base Register (32 bits)
PTLR - Page Table Length Register (32 bits)
SR - Status Register (32 bits-see below)

Status Register (32 bits):

=====

31	28	24	20	16	12	8	4	0

0000	0000	0000	0000	0000	0000	00IS	PZVN	

I = Interrupts Enabled (1=Enabled, 0=Disabled)
S = System Mode (1=System, 0=User)
P = Paging Enabled (1=Enabled, 0=Disabled)
Z = Zero (1=Result was zero)
V = Overflow (1=Overflow occurred)
N = Negative (1=Result was negative)
(All other bits are unused.)

The Registers

Each running process can access 16 general purpose registers and 16 floating-point registers.

Each general purpose register contains one word (32 bits) of data. These registers are numbered from 0 to 15 and are named r0, r1, ..., r15. They typically contain signed integer values.

The BLITZ Architecture

Sometimes the general purpose registers are called the “integer registers” to distinguish them from the floating-point registers.

The floating-point registers each contain two words (64 bits) of data. They are numbered from 0 to 15 and are named f0, f1, ..., f15. Each floating point register contains a double-precision floating point number, stored in the IEEE standard format.

Almost all instructions use at least one of the general purpose integer registers, and some use two or three integer registers. Where ever an integer register is used, any of the 16 registers may be specified. Most integer registers are identical and which register is used is a question for the programmer to determine. However, some registers (in particular, registers r0, r14, and r15) have special uses and functions, as described later.

A subset of instructions (called the “floating-point instructions”) access and modify data in the floating-point registers. These instructions all begin with the letter “f”. For example there are two addition operations. The integer operation is named “add” and the floating-point operation is named “fadd”.

User Registers and System Registers

There are two sets of general purpose registers. Each set has 16 registers. One set is called the “User” register set and the other is called the “System” register set. In each set, the registers are numbered 0 through 15 and named “r0”, “r1”, ... , “r15”.

At any time, the BLITZ processor is executing either in “User Mode” or “System Mode”. When the processor is in User Mode, the User Registers are used; when the processor is in System Mode, the System Registers are used.

The use of two sets of integer registers allows kernel traps (for example, “syscall” system calls) to be executed quickly. The user-level process will execute in User mode and will use the User registers. The kernel will execute in System Mode and will use the System registers. During the “syscall” processing, no register saving is necessary; the processor simply changes the Mode to switch to the other register set.

There is only one set of floating-point registers. This set is shared among between User and System processes.

Register r0

Register r0 is special. Its value is always zero. Any attempt to read from this register will result in zero being returned. Any attempt to store data into this register is perfectly legal; the data will simply be discarded.

Register r14 - The Frame Pointer

By convention, register r14 is used as a “frame pointer”. The frame-pointer register is used in the standard subroutine calling sequence. In conjunction with r15 (the stack pointer), it is used to locate activation records (i.e., “stack frames”) in the activation calling stack.

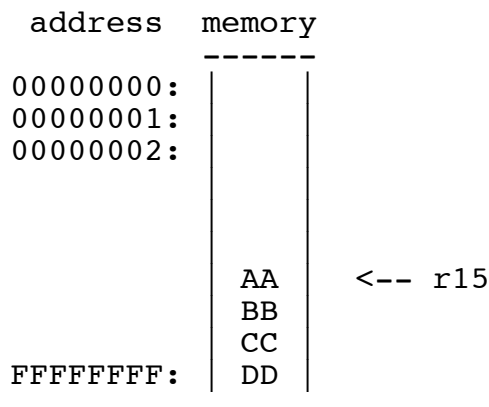
The BLITZ Architecture

There is no direct architectural support for the frame pointer. In other words, no instruction treats r14 any differently than any other register. In fact, any register could be used with no change to the assembler or machine emulator.

Register r15 - The Stack Pointer

By convention, register r15 points to a stack in memory. This stack is used during procedure calls and returns. This stack is also used during interrupt processing. The “push” and “pop” instructions also use the stack. The push and pop instructions will actually work with any register, but synthetic versions of “push” and “pop” make the use of r15 more convenient. (Synthetic instructions are implemented entirely within the assembler and serve as shorthand for other instructions. They are discussed later.)

By convention, the stack grows downward (toward lower-numbered addresses) from high memory. Register r15 points to the item at the “top” of the stack, i.e., r15 points to the lowest numbered byte that contains stack data. Thus, during a “push” operation, the stack-top register will be decremented before the item is moved to memory, whereas during a pop operation, the data will be moved from memory before the stack-top register is incremented.



Normally, the kernel will maintain its own stack, which will be indexed by the stack-top pointer stored in register r15 in the System Register set. Each user-level process will presumably have its own stack, although this is an operating system issue, not a processor feature. Each user-level process will have a set of User Registers, and its stack-top register r15 will index its stack.

Status Register (and Condition Codes)

The Status Register is defined as a 32 bit register; however, only six of the bits are actually used. When the Status Register is stored as part of the syscall / interrupt processing, the unused bits are set to zero. When the Status Register is loaded during a return from interrupt (the “reti” instruction), the unused bits are ignored.

Three of the status bits (Z=Zero, V=Overflow, and N=Negative) are referred to as the “Condition Codes”. These bits are set during certain arithmetic operations to reflect the nature of the computed result. The following instructions modify the condition codes:

add, sub, mul, div, sll, srl, sra, or, and, andn, xor, rem

The BLITZ Architecture

Note that the following synthetic instructions are shorthand for some of these instructions. Therefore, they also modify the condition codes:

```
mov, cmp, neg, not, clr, btst, bset, bclr, btog
```

Of course the “reti” instruction also modifies the condition codes since it reloads the entire Status Register.

The condition code bits are tested with the various branch instructions, which will conditionally jump according to how these bits are set. Thus, the program can test the result of an operation, such as a “cmp” instruction, and can branch accordingly.

I = Interrupts Enabled (1=Enabled, 0=Disabled)

=====

If this bit is set, then interrupts will be processed. If this bit is zero, then interrupts are disabled and incoming interrupts will not be processed; instead they will remain pending until interrupts are once again enabled. At that time, a pending interrupt (if any) will be processed. Only maskable interrupts can be disabled and remain pending. Unmaskable interrupts will always be processed, regardless of the state of this bit, and will never remain pending.

This bit may only be changed by the kernel. That is, this bit may only be changed by privileged instructions, and privileged instructions can only be executed in System Mode.

S = System Mode (1=System, 0=User)

=====

At any time, the BLITZ processor is executing in either System Mode or in User Mode and this bit determines which mode the processor is in. Presumably the operating system kernel will execute in System Mode and all user-level programs will execute in User Mode. All instructions may be executed when in System Mode. In User Mode, some instructions are forbidden. (The forbidden instructions are called “privileged” instructions.) An attempt to execute a privileged instruction while in User Mode will cause a Privileged Instruction exception and will trap into System Mode.

This bit may only be changed by the kernel. That is, it may only be changed by privileged instructions, which may only be executed in System Mode.

P = Paging Enabled (1=Enabled, 0=Disabled)

=====

The BLITZ architecture provides support for virtual memory via a page table with automatic address translation. If this bit is set, then address translation will occur whenever the program accesses main memory. Whenever an instruction accesses memory, the address it provides will be interpreted as a “logical” address (also called a “virtual” address). This logical address will be translated into a physical address by the CPU’s translation hardware before being used. The physical address will then be used to determine which bytes in memory are examined or modified by the instruction. If this bit is zero, then every memory address generated by an instruction will be used as is, with no translation. Presumably, an operating system kernel will run with paging disabled and user-level programs will run with paging enabled.

This bit may only be changed by the kernel. That is, it may only be changed by privileged instructions, which may only be executed in System Mode.

The BLITZ Architecture

Z = Zero (1=Result was zero)
=====

If the result of the arithmetic operation is zero (i.e., all bits are zero), then this bit will be set to 1. Otherwise, it will be cleared to zero.

V = Overflow (1=Overflow occurred)
=====

If the execution of the arithmetic operation caused overflow, then this bit will be set to 1. Otherwise, it will be cleared to zero. Overflow may occur for add, sub, mul, div, rem, and fcmp.

N = Negative (1=Result was negative)
=====

If the result of the arithmetic operation is negative (i.e., the most significant bit is “1”), then this bit will be set. Otherwise, it will be cleared to zero.

The Program Counter (PC) Register

There is a 32-bit register called the “PC”, which contains the address of the next instruction to execute. It is modified with instructions that branch or transfer the flow of control.

The Page Table Base Register (PTBR) Register

The Page Table Base Register (“PTBR”) contains the address of the page table. The page table is stored in main memory and this register will always contain a word-aligned address. This register is used whenever the page table is consulted by the CPU’s address translation hardware. In particular, when a virtual address is translated into a physical address, this register will be used to locate the page table in main memory. This only happens when paging is enabled by the “Paging Enabled” bit in the Status Register. If paging is disabled, this register is ignored. This register will contain a physical address, not a virtual address.

This register can only be loaded with the “ldptbr” instruction, and it cannot be examined directly.

The Page Table Length Register (PTLR) Register

The Page Table Length Register (“PTLR”) is used in conjunction with the Page Table Base Register. The page table consists of a number of entries, stored sequentially in memory. Each entry is 4 bytes (one word) long. This register contains the number of bytes in the table and therefore it will always be a multiple of 4.

This register can only be loaded with the “ldptlr” instruction, and it cannot be examined directly.

Instruction Formats

Every BLITZ instruction is 4 bytes long. The first byte of the instruction is the op-code, indicating what the instruction is. The remaining 3 bytes are interpreted differently. Some instructions have no operands, in which case the remaining 3 bytes are ignored. Other

The BLITZ Architecture

instructions take several operands, in which case the operands are encoded into the remaining 3 bytes. There are seven different instruction formats, named Format A, Format B, ... , Format G.

Here are the instruction formats. Each line shows how the 32 bits of an instruction are to be interpreted.

```
Format A:
  xxxx xxxx  ----  ----  ----  ----  ----  ----

Format B:
  xxxx xxxx  cccc  ----  ----  ----  ----  ----

Format C:
  xxxx xxxx  cccc  aaaa  ----  ----  ----  ----

Format D:
  xxxx xxxx  cccc  aaaa  bbbb  ----  ----  ----

Format E:
  xxxx xxxx  cccc  aaaa  vvvv  vvvv  vvvv  vvvv

Format F:
  xxxx xxxx  rrrr  rrrr  rrrr  rrrr  rrrr  rrrr

Format G:
  xxxx xxxx  cccc  ----  vvvv  vvvv  vvvv  vvvv
```

The various fields in the instructions are given by the following legend:

xxxx xxxx	Op code (8 bits)
aaaa	Register A (4 bits)
bbbb	Register B (4 bits)
cccc	Register C (4 bits)
vvvv vvvv vvvv vvvv	Immediate value (16-bits, sign-extended)
rrrr rrrr ... rrrr	Relative displacement (24-bits, sign-extended)
----	(These bits are ignored)

There are 16 general purpose registers and they are encoded in the obvious way in 4-bit fields (shown above as “aaaa”, “bbbb”, and “cccc”) in the instructions:

```
0000 - Register r0
0001 - Register r1
0010 - Register r2
...
1111 - Register r15
```

Many instructions take one or more general purpose registers as operands. These are referred to as Register A, Register B, and Register C. For example, the general format of the multiply instruction is:

```
mul    regA,regB,regC
```

A specific instance of this instruction might be:

```
mul    r7,r8,r13    ! Multiply r7 by r8 and place result in r13.
```

The BLITZ Architecture

In this instruction we have:

```
aaaa = RegA = r7  = 0111
bbbb = RegB = r8  = 1000
cccc = RegC = r13 = 1101
```

The op code for the “mul” instruction (Format D) is 98 (in binary: 0110 0010). Thus, this instruction would be encoded as:

Format D:

```
xxxx xxxx  cccc aaaa  bbbb ----  ---- ----
0110 0010  1101 0111  1000 0000  0000 0000
```

In hex:

```
6    2    D    7    8    0    0    0
```

Some instructions operate on the floating-point registers instead of the general purpose registers. An example is the floating-point multiply instruction:

```
fmul    fregA,fregB,fregC
```

A specific instance of this instruction might be:

```
fmul    f3,f4,f9      ! Multiply f3 by f4 and place result in f9.
```

There are 16 floating-point registers and (like the general purpose registers) they are encoded in the obvious way in 4-bit fields (shown above as “aaaa”, “bbbb”, and “cccc”) in the instructions:

```
0000 - Register f0
0001 - Register f1
0010 - Register f2
...
1111 - Register f15
```

The floating-point instructions (like “fmul”) interpret the 4-bit register fields (aaaa, bbbb, and cccc) as indicating floating-point registers; all other instructions interpret the 4-bit fields as meaning general purpose registers. Thus, the bit pattern

```
0101
```

may mean either “r5” or “f5”, depending on which instruction it is used in.

The op-codes are assigned in such a way that the first 3 bits of the op-code indicate the format of the instruction.

format	op-code range	op-code in binary
=====	=====	=====
A	0-31	000- ----
B	32-63	001- ----
C	64-95	010- ----
D	96-127	011- ----
E	128-159	100- ----
F	160-191	101- ----
G	192-223	110- ----
-	224-255	111- ----

16-Bit Sign-Extended Immediate Values

Many of the instructions allow for an immediate (or “literal”) value to be included directly in the instruction. (In particular, all Format E and Format G instructions allow it.) For example, the subtract instruction may include a value directly:

```
sub    r3,27,r4
```

This instruction subtracts 27 from the value stored in register r3 and stores the result in r4. The data (27) is specified immediately; it is not stored in a register. Instructions such as these include a 16 bit field in the instruction to contain the value. When executed, the 16 bits are interpreted as a signed integer. Thus, the 16 bit value is “sign-extended”. In other words, the most significant bit (bit 15) is duplicated to fill the value out to 32 bits. This means that any value between -32768 and 32767 can be specified literally. If you wish to subtract a number outside this range, you can not use a literal value; you will have to first place the value in a register of its own. (For example, you might use the “set” instruction to accomplish this.)

Note that the logical operations (such as shift left logical “sll” and exclusive-or “xor”) can also take 16-bit immediate values. These values are also treated as signed integers and are sign-extended to 32 bits before being used. Any attempt to use an out-of-range number, as in the following example, will be caught by the assembler and flagged as an error.

```
xor    r3,0xAB12CD34,r4    ERROR!
```

The programmer must be careful in situations like the following. Presumably the programmer is trying to flip a single bit (bit 15).

```
xor    r3,0x8000,r4
```

The assembler will not flag this as an error and will produce the same code as for this instruction:

```
xor    r3,0xFFFF8000,r4
```

One exception to the sign-extension rule is for the “sethi” and “setlo” instructions. These instructions are Format G instructions, so they contain a 16-bit immediate value. However, in these instructions, the value is not sign-extended before use.

24-Bit Relative Displacements

Format F instructions are concerned with branching and jumping. The following instructions have Format F version: the “call” instruction, the “jmp” instruction, and the conditional branch instructions (such as “bne”, “bl”, and “bge”).

Format F instructions contain a 24-bit field, which gives a displacement (also called an “offset”). This displacement is relative to the current location. More precisely, the 24-bit offset is sign-extended to 32-bits. This is then added to the address of the instruction (not the address following the instruction, as in some architectures) to give the target address. The target address is the address to be jumped to. If the branch is taken, the next instruction to be executed will be the instruction stored at the target address.

In the BLITZ architecture, logical (i.e., virtual) address spaces are limited to 16M bytes. (Recall that any byte in a 16M byte address space can be addressed with 24 bits.) Therefore, any byte in

The BLITZ Architecture

the logical address space may be addressed using a 24-bit relative displacement in a Format F instruction located anywhere in that address space. The address computation (adding the current address to the 24-bit displacement) is done using 32-bit arithmetic, but any bits beyond 24 are simply ignored when paging is enabled.

The benefit of using relative displacements instead of absolute addresses is that the code can be relocated (i.e., moved) from one area of memory to another, without having to be modified. As long as the relative separation of the Format F (branch) instruction and the target instruction of the branch remains unchanged, no modification of the branch instruction is necessary.

Consider the following code, for example:

```
loop:    add    r5,5,r5
         add    r3,1,r3
         cmp    r3,17
         ble    loop
```

The branch instruction will contain a displacement of -12, meaning that the target instruction is 12 bytes before the branch instruction. At runtime, this code may be loaded at any address. For example, it might be loaded at addresses starting at 0x65432100:

```
6543 2100    loop:    add    r5,5,r5
6543 2104                add    r3,1,r3
6543 2108                cmp    r3,17
6543 210C                ble    loop
```

When the processor executes the branch instruction, it takes the current value of the program counter (6543210C) and adds the relative offset in the instruction (i.e., -12), giving the address of the instruction labeled “loop”. Obviously, if this code had been loaded at some other address at run-time, it would still work correctly.

To be precise, the 24 bit displacement is signed extended to 32 bits. Then this value is added to the address of the branch instruction. This means that the target of a branch instruction must lie within -8,388,608 to 8,388,607 bytes of the branch instruction. If the machine is operating with paging enabled, then the target address is a virtual address. It will be truncated to 24 bits before being translated into a 32-bit physical address.

(Since the branch instruction must be at an aligned address and since the target address must be aligned, the displacement in a branch instruction should also be divisible by 4. Therefore, the last 2 bits of any 24-bit displacement in a Format F instruction should be zero; if the displacement is not divisible by 4, an Alignment Exception will occur.)

Addressing Memory

In order to fetch or store data to / from memory, the address must first be placed in a register. The “set” instruction is ideal for moving the address into a register. For example, the following code first moves the address of “myVar” into register r1. Then it uses that register to fetch from memory. This code increments the word whose address is “myVar”.

```
set      myVar,r1      ! Move address of myVar into r1
load     [r1],r2        ! Move data from myVar into r2
add      r2,1,r2        ! Increment r2
store    r2,[r1]        ! Store data back, using same address
```

The BLITZ Architecture

A second addressing mode allows an integer offset to be added to a register to compute the effective address. For example, assume “myArr” is the address of the first element of an array of words. We wish to increment the sixth element of this array. We will need an offset of 20, since each element is 4 bytes long.

```
set      myArr,r1
load     [r1+20],r2
add      r2,1,r2
store    r2,[r1+20]
```

A third addressing mode allows us to add the contents of two registers together to compute the effective address. For example, if the offset into the array has been computed and placed in register r3, then we can use the following code to access the desired element.

```
set      myArr,r1
load     [r1+r3],r2
add      r2,1,r2
store    r2,[r1+r3]
```

A common thing to do is to access fields in the activation record (or “frame”). Assume that variable x is a local variable stored at offset -12 in the frame. Then the following code would be used to increment x.

```
load     [r14-12],r2
add      r2,1,r2
store    r2,[r14-12]
```

In this example, we assume that the “standard subroutine calling conventions” have been followed and that “r14” points into the frame stored on the top of the stack. In the standard calling conventions, which are described elsewhere, the frame pointer will point into the middle of the frame, not to the lowest byte in the frame. The local variables will actually be stored in the frame at negative offsets from the frame pointer.

Synthetic Instructions

There are several “synthetic” instructions. Technically, these are not true BLITZ instructions. Instead, they are introduced to make life easier for the programmer. Whenever the assembler sees a “synthetic” instruction, it will translate it into one or two legal BLITZ instructions and will use those instructions instead.

For example, “set” is a synthetic instruction. It will be expanded into two instructions. Consider the following code:

```
set      myArr,r1
load     [r1],r2
```

The “set” instruction will be automatically expanded by the assembler into two instructions, giving:

```
sethi    hi(myArr),r1
setlo    lo(myArr),r1
load     [r1],r2
```

The BLITZ Architecture

Here we are using “lo” and “hi” to indicate the lo-order and hi-order 16 bits of the full 32-bit value given by the symbol “myArr”, although these are not really part of the assembler syntax.

The “sethi” and “setlo” instructions are discussed later, but in short, “sethi” moves 16 bits of data into the register, and the “setlo” instruction brings in the remaining 16 bits of data.

You might consider the following code sequence, which would be more efficient. However, this code sequence is subject to a very subtle bug.

```
sethi    hi(myArr),r1          ! Warning: This code sequence
load     [r1+lo(myArr)],r2     !   may not do what you expect.
```

Recall that the 16-bit immediate value “lo(myArr)” in the load instruction would be sign-extended. The above instruction sequence will work only if the most significant bit of “lo(myArr)” was zero; otherwise it would be sign-extended, causing the address computation to result in a very different address. Since code may be relocated during linking, it is difficult to know whether this condition would be met. Therefore, the “setlo” instruction should always be used when moving 32-bit constant values.

Assembler Comments

The exclamation point (!) is used to begin comments. The comment runs through the end-of-line. As a point of style, every line of assembler code should probably be commented.

When a single comment applies to several instructions, I prefer to use a period as demonstrated below to indicate that a multi-line comment applies to several instructions.

```
set      myVal,r1              ! Increment myVal
load     [r1],r2               ! .  by one
add      r2,1,r2               ! .
store    r2,[r1]               ! .
cmp      r2,47                 ! If myVal >= 47
bge      loop                  ! .  then goto loop
```

Labels and White Space

Each instruction may be preceded by an optional label. This label may then be the target of a “branch” or “call” instruction. Data locations in memory may also be labeled, in which case the label could be used in “load” or “store” instructions.

If a label is present, it must be at the beginning of the line and must be followed by a colon. The instruction op-code is separated from the label by some white-space (usually a single tab character). The instruction op-code is followed by white space (usually a tab or two) and the operands. These can optionally be followed by some white space and a comment.

A label may also appear on a line by itself, in which case it applies to the current location counter, and will therefore be set to the address of the next instruction (notwithstanding things like “.align” pseudo-ops that may change the location counter). Op-codes (both legal BLITZ instructions and pseudo-ops) must be preceded by white space, typically a tab or two.

The convention is to try to make things line up neatly. For example:

The BLITZ Architecture

```
myLoop:                                ! myLoop:
        set      myVal,r1              ! Increment myVal
        load     [r1],r2              ! . by one
        add      r2,1,r2              ! .
        store    r2,[r1]              ! .
        cmp      r2,47                ! If myVal >= 47
        bge      myLoop              ! . then goto myLoop
        ...
myVal:   .word    0                   ! myVal: counter of things
```

The Program Counter

Every computer processor contains a register called the Program Counter (or “PC”) which contains the address of the next instruction to be executed. In the simplest model of computer operation, the processor repeatedly executes this algorithm:

```
pc := 0;
loop
  fetch next instruction from memory[pc];
  pc := pc + 4;
  execute the instruction:
    Decode the instruction
    Fetch operands from memory or registers
    Perform computation
    Store results back into memory or registers
endLoop
```

Sometimes the instruction will modify the flow of control (a “call” or “branch” instruction). For such instructions, a “result” will be stored into the PC register, causing the next instruction to be fetched from a new address.

In the BLITZ processor, each instruction executes to completion before the next instruction begins. There is no pipelining of instructions. (Not included in this discussion is the processing of exceptions, asynchronous interrupts, or “syscall” traps. When such interrupts occur, the above processing loop is more complex.)

The Location Counter

As the assembler scans and processes each line of code, it keeps track of the current memory location into which each instruction will be placed. To do this, the assembler keeps a variable called the “Location Counter”, which should not be confused with the “Program Counter”. The location counter and the program counter are different.

It is important to understand that the location counter is an assembly-time concept; it does not exist at run time. On the other hand, the program counter is a run-time register (named “PC”) and only has a value when the program is running. When the assembler encounters a branch instruction, it never “takes” the branch; instead it simply moves on to the next instruction in the source file by incrementing the location counter. Later, at runtime, when the CPU encounters a branch instruction, it may take the branch by modifying the PC.

Every time the assembler encounters an instruction, the location counter is advanced by 4 (i.e., by the size of the instruction). Even when a branch instruction is processed, the location counter

The BLITZ Architecture

is always advanced by 4 and the instruction on the line following the branch is then processed. To repeat, a branch is never “taken” by the assembler.

When the assembler encounters a label, it adds a new definition to its symbol table, using the current value of the location counter as the definition of the symbol. When the assembler encounters a branch instruction or a load or store instruction using some label, it looks that label up in the symbol table. It will use the value of that symbol in assembling the branch instruction.

Assemblers make two passes over the source program. In the first pass, the location counter is set to zero, and the source code file is scanned line-by-line. Each time an instruction is encountered, the location counter is advanced by the length of the instruction, but no code is generated in the first pass. Whenever a label is encountered, it is entered into the symbol table, using the current value of the location counter as the symbol’s definition.

In the second pass, the location counter is reset to zero. Then, the source code is examined again line-by-line from beginning to end. This time, machine code is generated and written to the output as each instruction is scanned.

This two-pass approach allows branches and other instructions to use labels that are defined either before or after the instruction. As the second pass begins, all symbols are already defined. As the second pass proceeds, the machine code for each instruction can be determined by plugging in the symbols’ values as necessary.

Branch and call instructions in the BLITZ architecture are “relative”, and not “absolute”. Each branch and call instruction contains a 24 bit field to specify the target address to be jumped to. This 24 bit field does not contain the absolute address of the target instruction. Instead, this field is interpreted as a signed number, which will be added to the current program counter at run-time to give the target address. Although the assembler does not know where the program will ultimately be loaded in memory, it is able to compute the relative distance between a branch instruction and its target address. Thus, it can produce code containing the correct relative displacement.

The assembler can be used to place data and instructions into different segments. The three segments are named “.text”, “.data”, and “.bss”. The assembler maintains a distinct location counter for each segment. Thus, there is a “.data” location counter, a “.text” location counter, and a “.bss” location counter. At any one time, the assembler is “in” only one segment, as determined by the .data, .text, and .bss pseudo-ops. For example, if the last pseudo-op was .text, all generated data following it will be placed in the “.text” segment and the corresponding location counter will be incremented.

Pseudo-Ops (Assembler Directives)

The following sections describe several “pseudo-operations”, which control and direct the assembler in its task. Pseudo-ops are sometimes called “assembler directives”.

Pseudo-ops are included in the assembly language program file, mixed in among the real BLITZ instructions. However, pseudo-ops are not executed at runtime; instead they tell the assembler how to assemble other instructions and what to put into memory before program execution begins. For example, a pseudo-op could be used to initialize a variable with some particular value. The value can be coded using convenient (C-like) notation, freeing the programmer from having to specify the precise bit-pattern used to represent the value.

The BLITZ Architecture

Pseudo-ops begin with a period to make it easy to distinguish them from BLITZ instructions.

Character Data

Here are examples of the “.ascii” pseudo-op:

```
.ascii    "abc"
.ascii    "BLITZ programming is fun!\n"
```

The “.ascii” pseudo-op places N bytes of data in memory, where N is the number of bytes in the character string. A number of escapes (such as “\n”) can be used to place non-printable ASCII codes in string.

The standard “C” string convention is to terminate every string of characters with the NULL character. Note that the .ascii pseudo-op does not place a terminating NULL character in the string. If this is what you want, you must include it explicitly. For example:

```
.ascii    "Hello, world.\n\0"
```

Data

```
.byte     expression
.word     expression
```

These pseudo-ops place data values in memory. In each case, arbitrary expressions may be provided, as long as they can be fully evaluated at assembly-time. These expressions may include many of the operators (such as + and *) available in the C language.

Double Constants

```
.double    value
```

This pseudo-op places an 8 byte (64 bit) double precision floating-point value in memory. Here are some examples, showing the ways in which the value may be specified.

```
.double    123.456
.double    -123.456
.double    +123.456
.double    123.456e10
.double    123.456e-10
.double    123.456e+10
.double    123.456E10
```

Alignment

The “.align” pseudo-op will force the location counter to be word aligned. In other words, one or more bytes of padding may be inserted to round the location counter up to an address divisible by 4.

```
.align
```

The BLITZ Architecture

In the first line of the following example, up to 3 bytes will be inserted (if necessary) to bring the location counter up to a multiple of 4. In the second line, 6 bytes will be allocated. In the next line, 2 additional bytes will be allocated so the next word will be placed in an aligned address.

```
.align
.ascii      "abcdef"
.align
.word       43
.word       "cs30"
.word       -1
.word       0x0123abcd
```

If the location counter is already .aligned, this pseudo-op will insert no padding bytes. Thus, it never hurts to place an .align pseudo-op directly before instructions that require a particular alignment, and it is necessary if the preceding instructions may have left the alignment wrong.

Uninitialized Data Space

```
.skip      N
```

This pseudo-op skips over N bytes, where N is any expression. The assembler and linker will fill these bytes with zeros.

Segment Control

A running process is loaded into main memory before execution begins. In Unix, a running process is divided into four “segments”. These segments have the following names:

```
.data
.text
.bss
.stack
```

Before a program begins execution, initial data will be loaded into the “.data”, “.text”, and “.bss” segments. The assembler is used to specify exactly what will be loaded into the “.data”, “.text”, and “.bss” segments and how long they will be. For example, the “.data” segment might be loaded with 100 bytes giving the initial values of several variables. The “.text” segment might be loaded with 11,000 bytes of machine instructions. Often the “.bss” segment is not used and will have a length of 0 bytes.

After loading, the OS will usually mark the bytes in the “.data” segment read/write, and the bytes in the “.text” segment will be marked “read-only”. Any attempt at run-time to modify a byte in the “.text” segment will result in an exception and the user-level program will be aborted by the OS. Any attempt to access a byte outside of any segment will also result in a similar error.

The “.stack” segment will be marked read/write, and has no fixed size; instead any attempt to access bytes beyond the end of the “.stack” segment will result in new pages being added to that segment.

There are three pseudo-ops that allow control over which segment to place data and instructions into. They are:

The BLITZ Architecture

```
.data
.text
.bss
```

These pseudo-ops determine into which segment the following instructions and/or data will be placed. Each “remains in effect” until a new .data or .text or .bss pseudo-op is encountered.

The .bss segment contains data that will be initialized to zero. Thus, no space will actually be consumed in the executable “a.out” file; the file will merely contain information about which bytes of the virtual address space must be initialized to zero before execution begins. As a consequence, the .bss segment may contain only .align and .skip pseudo-ops. Instructions and other pseudo-ops are not allowed in the .bss segment.

At assembly time, each of the three segments is assumed to start at location zero. Later, when the program is loaded into main memory, an address will be selected. Also, the size of the segment is rounded up to the nearest page boundary. In a machine with 8K byte pages, our “.data” segment with 100 bytes would be rounded up to 8K bytes. The “.text” segment, with 11,000 bytes would be rounded up to 16K bytes. Then an address is selected for each of the four segments and the initial data is loaded into the appropriate memory locations.

Symbols

Symbols may be defined in several ways. The most common way is when an instruction is labeled. Another way is when the location of data in memory is allocated (e.g., with a “.word” pseudo-op) and a label is present.

When the assembler encounters a label, it makes a new entry in its symbol table, associating the symbol with the current value of the location counter. It also makes a note in its symbol table of which segment this address is in. Later, when the several separately-assembled pieces of the program are linked together, each segment will be assigned a specific address. This will require the value of the symbol to be adjusted. Furthermore, every instruction that uses the symbol must be modified to reflect the fact that the symbol is being adjusted. Obviously, there is a lot of information in the “.o” files that indicates which symbols are used as well as where and how they are used, so that the linker can make these adjustments when the various segments are put together.

Symbols may also be defined directly using the “=” pseudo-op. Here is an example:

```
        set      myVal,r1          ! Increment myVal
        load     [r1],r2           ! . by “incrAmt”
        add      r2,incrAmt,r2     ! .
        store    r2,[r1]          ! .
        ...
incrAmt = 3
```

Note that “=” is a pseudo-op (like “.align” or “.text”) although “=” differs in that it is not given an alphabetic name beginning with a period.

In this example, we have introduced a symbol called “incrAmt” and set its value to 3. Such a symbol is an assembly-time constant: it can never change. This symbol is absolute, which means it is not relative to any segment. During linking, when the segments are assigned addresses, no

The BLITZ Architecture

adjustment will be needed to the “add” instruction. The value of 3 will be used. The symbol “incrAmt” may be used any place the constant 3 may be used.

In this example, the operand of the “=” pseudo-op is “3”. In general, the operand can be any expression, as long as it can be evaluated at assembly-time.

External Symbols

```
.export  localSymbol
.import  foreignSymbol
```

It is assumed that “localSymbol” is defined within the “.s” source file that contains the .export pseudo-op. It is assumed that “foreignSymbol” is not defined in the file containing the “.import” pseudo-op but is defined in some other file that is assembled at some other time and linked with this file. The assembler and linker will check that .export and .import are used correctly.

When the assembler processes a “.s” source file and produces a “.o” object file, it will include information in the “.o” file to identify the symbols used and give information about their values (or about their locations, if they are labels). The .export pseudo-op will make “localSymbol” visible to other “.o” files with which this file is linked. The .import pseudo-op will make a symbol called “foreignSymbol” defined in some other file visible and usable within the current file.

Consider this example:

```
                .export    main
                .import    printf
main:           ...
                add        r1,234,r3
                ...
                call        printf
                ...
```

When an imported symbol (such as “printf” in this example) is used in this file, the linker will look for a definition of it in other “.o” files and will modify instructions (such as the “call” instruction) in the executable file to point to it. In this example, a routine called “main” is defined and exported. Thus, this routine can be called from other “.s” files which are assembled separately.

It is a good idea to put an “.align” pseudo-op before the first instruction, in case a previous data item left the location counter on a non-word aligned boundary.

```
                .align
main:           add        r1,234,r3
```

Note that the following is in error since the symbol “main” would be set to point to the “padding” bytes inserted by the “.align”.

```
main:           .align
                add        r1,234,r3
```

Interrupt Processing

In this document, we use the term “interrupt” very generally to refer to an interruption in the normal sequential execution of instructions. We use the terms “exception” and “trap” more specifically, as discussed next.

Asynchronous interrupts may occur at anytime. These are triggered by events outside of the instruction processing sequence of the CPU. For example, a hardware device (such as a disk or the interval timer) may suddenly require attention. The device will send an electrical signal to the processor, which will trigger interrupt processing. This sort of interrupt is referred to as an “asynchronous interrupt” or a “hardware interrupt”.

Synchronous interrupts occur as a result of executing instructions within the CPU. These can be divided into two categories: those that are the result of problems and those that are specifically intended. Interrupts that are the result of problems arising during instruction execution are called “exceptions”. Examples include “Illegal Instruction” and “Alignment Exception”. In the case of an exception, some unexpected event occurred during the execution of an instruction.

The only intentional interrupt is caused by the system call instruction (“syscall”). This sort of an interrupt is often referred to as a “trap”, rather than as an “exception”. It is used by a user-level program to enter (or invoke) the kernel of the operating system.

This table summarizes our terminology and lists all types of interrupts:

```
Interrupt:
  Asynchronous Interrupts (Hardware):
    - PowerOnReset
    - TimerInterrupt
    - DiskInterrupt
    - SerialInterrupt
    - HardwareFault
  Synchronous Interrupts:
    Exceptions:
      - IllegalInstruction
      - Arithmetic
      - Address
      - PageInvalid
      - PageReadOnly
      - PrivilegedInstruction
      - Alignment
      - ExceptionDuringInterrupt
    Trap:
      - Syscall
```

When an interrupt occurs and is serviced, the following actions occur. (If the interrupt is not serviced, it remains pending, as discussed later, and will be serviced at some other time.)

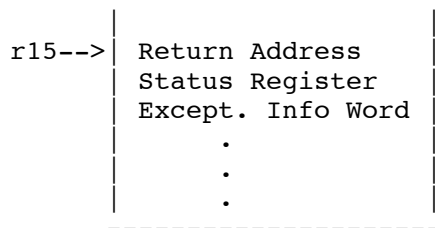
First, the currently executing instruction is finished. In the case of an asynchronous (hardware) interrupt, the current instruction runs to completion and the PC is left pointing at the next instruction. In the case of an exception, the instruction execution is preempted and the PC is left pointing to the offending instruction. Furthermore, in the case of an exception, the offending instruction has no effect; the state of the CPU and memory is not altered. It is as if the instruction had never been attempted. In the case of a “syscall” trap, the PC is left pointing to the instruction following the “syscall”.

The BLITZ Architecture

Second, an “exception info word” is pushed onto the stack. In the case of a PageInvalid or PageReadOnly exception, the offending virtual memory address will be pushed onto the system stack. In the case of a “syscall” trap, the “system trap number” will be pushed onto the system stack. By pushed onto the system stack, we mean that System Register “r15” (not User Register r15) will be used. For all other interrupt types, the “exception info word” will be all zeros.

Third, the Status Register will be pushed onto the system stack.

Fourth, the PC will be pushed onto the system stack. At this point, the system stack looks like this:



Fifth, the following bits of the Status Register will be changed. This will change the CPU to System Mode, disable subsequent interrupts, and disable address translation via the page table.

```

I := 0
S := 1
P := 0
```

Sixth, the PC will be loaded with the address of one of the Interrupt Vector entries. This has the effect of causing a branch to the appropriate interrupt handler code. The interrupt vector consists of several words stored in low memory, as shown below. In particular, there is one word for each type of interrupt. The table below gives the addresses of the table entries (in hex). (Normally, the OS would have stored into each word of this table a “jump” instruction, which will branch to the first instruction of the interrupt handler for that type of interrupt.) In this step, the PC is simply loaded with the corresponding address from the following table.

Interrupt Vector in Low Memory

Address	Description	Maskable
=====	=====	=====
000000	Power On Reset	No
000004	Timer Interrupt	Yes
000008	Disk Interrupt	Yes
00000C	Serial Interrupt	Yes
000010	Hardware Fault	No
000014	Illegal Instruction	No
000018	Arithmetic Exception	Yes
00001C	Address Exception	No
000020	Page Invalid Exception	No
000024	Page Readonly Exception	No
000028	Privileged Instruction	No
00002C	Alignment Exception	No
000030	Exception During Interrupt	No
000034	Syscall Trap	Yes

The BLITZ Architecture

To repeat what happens, given the type of the interrupt, the corresponding address (from the above table) is moved into the PC. This will cause a transfer of control directly into the table. Normally, each table entry will hold a “jump” instruction; if so, this jump instruction will be executed on the following instruction cycle, causing a second transfer of control to the correct interrupt handler routine.

Some interrupts can be handled simply by the interrupt handler routine and a return to the interrupted process can be made fairly quickly. (For example, a Serial Interrupt may indicate that the user has pressed a character key on the keyboard. It may be adequate to simply query the device and save the character in some sort of a buffer before returning to the interrupted process. The details are operating system dependent.)

The Return From Interrupt instruction (the “reti” instruction) would normally be used to return from an interrupt handler routine to the interrupted process. The “reti” instruction works as follows. First, the “reti” instruction will restore the PC by popping the top value from the from system stack and moving it into the PC, thereby preparing a return in the flow-of-control back to the interrupted process. Then, “reti” will restore the Status Register, by popping the next value from the from system stack into the Status Register. Then, “reti” will pop and discard the next value (the exception info word) from the system stack. After “reti” has completed, instruction execution will then resume in the interrupted process.

Generally, the interrupted process will be a user-level process and so will have been executing in User Mode, although sometimes interrupts will occur while the kernel itself is executing. If a User Mode process was interrupted, none of its registers will have been modified while the handler routine serviced the interrupt, since the interrupt handler executed in System Mode. If the handler routine was written correctly, the interrupted process will be entirely unaffected and unaware that it was interrupted.

For example, if the interrupt was a PageInvalid or PageReadOnly exception, then presumably the necessary page will have been moved into a memory frame during the execution of the interrupt handler. The PC will have been left pointing to the instruction that caused the exception. Upon return, the interrupted instruction will be re-tried and will succeed this time (unless another different interrupt occurs first).

Masking Interrupts

Some interrupt types are said to be “maskable” and the rest are “unmaskable”. When an unmaskable interrupt occurs, it will be serviced as soon as possible, namely on the next instruction cycle. Maskable interrupts may be serviced either immediately or at some later time, when the software is ready to deal with the interrupt. (By “serviced”, we mean that the interrupt processing sequence discussed above will be performed. Three words will be pushed onto the system stack and a branch will be made to the interrupt handling routine.)

The Interrupt Enabled bit (the “I” bit) in the Status Register determines whether a maskable interrupt will be serviced when it occurs. If the “I” bit is set to one, then the interrupt will be serviced immediately, without delay (i.e., within the current or next instruction cycle) and a transfer will be made to the handler routine.

If the interrupt type is unmaskable, then interrupt processing will occur regardless of the state of the “I” bit. The interrupt cannot be masked or disabled. If the interrupt is signaled, the interrupt sequence will occur and a transfer will be made to the corresponding handler.

The BLITZ Architecture

If the interrupt type is maskable, and the “I” was previously cleared to zero, then the interrupt servicing sequence will not occur. No transfer to the handler routine will be made and instruction execution will continue without interruption. Furthermore, the interrupt will remain “pending”. That is, the moment that interrupts are re-enabled (by setting the “I” bit to one), the interrupt servicing sequence will be initiated and a transfer to the interrupt handler will occur.

There may be several interrupts pending, but within a single interrupt type, there will be at most one pending interrupt. For example, there could be a Disk Interrupt and a Timer Interrupt both pending, but there can be at most one Disk Interrupt pending. Of course the disk device may repeatedly request a Disk Interrupt after the first has been fully serviced (and the “I” has been reset to one, re-enabling interrupts) causing the handler to be invoked a second time.

Exception During Interrupt

During interrupt servicing, data is pushed onto the system stack. If system register “r15” is not aligned properly or points to an invalid address, there are going to be problems. In other words, an “Address Exception” or an “Alignment Exception” can occur during the processing of interrupts.

(Note that an exception may occur during the servicing of an exception. To be precise and avoid infinite regress, we should probably avoid saying that “an exception occurs during the servicing of an exception” and just talk about what happens.)

What happens when we have problems while servicing an interrupt? The current interrupt is abandoned and the PC is loaded with the address of the ExceptionDuringInterrupt entry in the low-memory interrupt vector. While processing the original interrupt, the processor may have successfully pushed one or two words onto the system stack before encountering a problem. In any event, no further attempt to push words onto the stack is made. Instead, the branch is immediately made to the ExceptionDuringInterrupt handler.

Presumably, an ExceptionDuringInterrupt indicates a catastrophic software failure of the operating system itself and will be handled by printing a final message before terminating the operating system.

Virtual Memory Address Translation and The Page Table

The Paging Enabled bit in the Status Register (the “P” bit) tells whether virtual memory address translation is turned on or not. If the “P” bit is cleared to zero, then there is no address translation. Every address generated during instruction execution is used “as is” as an index into physical memory. By “physical memory”, we mean the main memory of the processor.

Physical addresses are a full 32 bits. If a physical address is beyond the end of the installed memory, then an Address Exception occurs and interrupt servicing begins (as discussed later). A 32 bit address allows up to 4G bytes to be installed and addressed, although since I/O devices are memory-mapped, several addresses will be unavailable for use. The Interrupt vector lies in low memory and memory-mapped I/O devices will be mapped into the highest addresses in the physical address space.

The BLITZ processor includes a Memory Management Unit (MMU), which is either enabled or disabled, as determined by the “P” bit. When enabled, the MMU provides a simple page-table

The BLITZ Architecture

based translation scheme, which may be used by an operating system to implement virtual memory.

As a program is running and instructions are being executed one after another, “logical” addresses are generated. Every instruction must be fetched from memory, so the execution of each instruction always begins by fetching a 32-bit word from memory. The current value of the PC is used as the logical address of the word to fetch. Many instructions will not access memory again; all of their action will be done using CPU registers only. However, several instructions (such as “load” and “store”) will go to memory a second time. The “load” instruction will fetch from memory and the “store” will write data to memory. One instruction, the “test and set” (“tset”) instruction, will go to memory two more times. The tset instruction will fetch a word and then will store data to that same word. Regardless of which instruction is being executed, the CPU will generate a stream of “logical” addresses that will be used to fetch and store data from/to memory. Logical addresses are 32-bits long.

When address translation is disabled, the “logical” addresses will be used as “physical” addresses directly. There will be no translation: the address will be used as is.

When address translation is enabled, the logical address will be broken into three fields, denoted “xxx”, “ppp”, and “ooo” below. The full 32-bit logical address is shown below, with the bits numbered.

31	28	24	20	16	12	8	4	0
====	====	====	====	====	====	====	====	
xxxx	xxxx	pppp	pppp	pppo	oooo	oooo	oooo	

xxxxxxxxx - high order bits (8 bits)
ppppppppppp - page number (11 bits)
ooooooooooooo - offset (13 bits)

Page Size = 8K bytes
Page Table Entry = 4 bytes (32-bits)
Max Page Table Size = 2K entries = 8K bytes

The high-order 8 bits of a logical address (“xxx”) are ignored and zeros are used. This limits the amount of addressable memory to 16M bytes, i.e., the amount that can be addressed with only 24 bits of address. This limits each logical address space to 16M bytes. Since the upper bits are ignored, it is not an error to address bytes outside the 16M byte limit; instead wrap-around occurs with no fanfare. Note that with a 24-bit offset (as is provided in all Format F instructions), every byte within the logical address space can be addressed.

The logical address space is broken into “pages” and each page is 8K bytes long. Since the logical address space is 16M bytes long, there are 2K (i.e., 2048) pages in each logical address space. The “ppp” field is 11 bits long and is used to select the page. Since each page is 8K bytes long, 13 bits are needed to select the individual byte within the page. This is given by the “ooo” (offset) field in the logical address.

Physical memory is divided into “frames” and each frame is 8K bytes long. Frames always begin on even (8K) boundaries. Each frame can hold exactly one page, but there need not be a simple one-to-one mapping between the pages in a logical address space and the frames in physical memory. (In a simple one-to-one mapping, frame 0 holds page 0, frame 1 holds page 1, frame 2 holds page 2, and so on.) The page table allows the mapping to be more complex.

The BLITZ Architecture

A page table is an array of “page table entries”. Each page table entry is one word (32 bits) long. A page table may have up to 2K entries, but it may have fewer (possibly zero) entries as well. (Note that a page table of the maximum size will exactly fit into a single page/frame.)

A page table is stored in memory, not in special MMU registers. In fact, the MMU has no visible state or registers of its own, beyond the “Page Table Base Register” and the “Page Table Length Register”. At any one time, there is a current page table and this is given by the contents of these two registers. If paging is disabled, these registers and the table they point to are ignored. If paging is enabled, then these two registers define the current page table, which is used during address translation. An operating system may store other page tables in memory, but these are ignored by the hardware. Only the current table counts and it is only used if paging is enabled.

The format of an entry in a page table is given by the following fields:

31	28	24	20	16	12	8	4	0
====	====	====	====	====	====	====	====	====
ffff	ffff	ffff	ffff	fff-	----	----	----	DRWV

ffffffffffffffffffff - frame number (19 bits)
D - Dirty bit (1=updated, 0=not updated)
R - Referenced bit (1=referenced, 0=not referenced)
W - Writable bit (1=writable, 0=read-only)
V - Valid bit (1=valid, 0=not valid)
(other bits) - unused (9 bits)

The frame number serves to address the frame in physical memory. 19 bits can select one out of 512K frames. (Each frame is 8K bytes. This allows addressing any byte in physical memory, since $512K * 8K \text{ bytes} = 4G \text{ bytes}$.)

The Valid bit is checked during translation; it must be set to one. If not, a Page Invalid Exception will occur. If the CPU is trying to store into memory, the Writable bit will also be checked; it must be set to one. If not, a Page Read-Only exception will occur. The Referenced bit will be set to one by the MMU when a page is either queried or updated. The Dirty bit will be set whenever a page is updated.

The remaining 9 bits are not examined or updated. The operating system is free to store information in these bits if desired.

When presented with a logical address, if paging is enabled, the MMU will perform the following functions.

First, the appropriate entry from the current page table will be fetched into an internal (hidden and unnamed) register within the MMU. If the Valid bit is zero, a Page Invalid exception will be signaled and no further processing will be done on this instruction. If this is an attempt to store into memory, the Writable bit will be tested. If it is zero, a Page Read-Only exception will be signaled and no further processing will be done on this instruction. If both bits are OK, the frame number will be concatenated with the offset, to give the following physical address:

31	28	24	20	16	12	8	4	0
====	====	====	====	====	====	====	====	====
ffff	ffff	ffff	ffff	fffo	oooo	oooo	oooo	

If this is an illegal address in physical memory, an Address exception will be signaled and no further processing will be done on this instruction. Otherwise, the data will be either fetched from or stored into memory at this physical address. (In the case of the “tset” instruction, a word

The BLITZ Architecture

will be both fetched and written to the selected address.) The Referenced bit will then be set to one and, if data was stored into memory, the Dirty bit will also be set to one. All other bits will be unchanged. Finally, the page table entry will be written back to memory from the internal MMU register.

The BLITZ architecture is designed to accommodate shared memory multiprocessing, although it will be mostly used as a uniprocessor. With shared memory multiprocessing, several concurrently executing CPUs will share a single physical memory. Multiprocessor interaction and frame locking are discussed next.

Note that when the CPU generates a single memory operation (for example, an instruction fetch) there are actually several reads and writes performed on main memory. A typical memory access will include (1) a read from the page table, (2) a read and/or write to the target frame, and (3) a final update to the page table. Technically, the write back to the page table entry will occur before the actual target location is queried or updated.

In the case of a multiprocessor system, we have the possibility of interleaving of memory accesses from several processors. For example, one processor might conceivably change a page table entry that is being used by another processor.

There are several design choices in how the various memory operations are interleaved in a multi-processor system. Here are several options that were considered in the design of the BLITZ architecture.

First, an architecture might specify that the memory accesses are done independently, with no coordination or synchronization. The only thing that would be done atomically is the read-write pair performed by the “tset” instruction. Unfortunately, with this design option, it is possible that a page table entry may not be updated correctly. For example, assume that some page table entry has its Dirty bit clear. Assume two processors both fetch the same entry, more or less simultaneously. Next, one processor updates the entry first, writing back an entry in which the Dirty bit has been changed to one. Then, the second processor writes back its version of the entry, in which the Dirty bit is still clear. The problem is that the update to the Dirty bit is lost. This is clearly unacceptable.

Second, an architecture might specify that all memory accesses (from (1) the initial read from the page table, through (2) the read and/or write to the target frame, and to (3) the final update to the page table) will be executed atomically, without interference from other processors. This could be implemented by locking each of the page frames involved. Up to two frames may be involved in a single CPU-initiated memory operation: the frame containing the page table and the target frame containing the data to be fetched and/or updated. If the architecture specifies that a processor will first get a lock on the frame containing the page table entry and then will try to get a lock on the frame containing the data before releasing the first lock, there is the possibility of deadlock. Two processors could each hold a lock on their first page while waiting to acquire a lock held by the other processor. Another serious problem with this design option is that for every single memory operation, the MMU holds a lock on a frame containing a page table entry longer than absolutely necessary, and this could slow performance whenever two processors try to run in the same logical address space.

In the BLITZ architecture, a lock is associated with each frame in memory. The lock on a frame may be acquired and released by the MMU during its operation. To perform a memory access, the MMU will first obtain a lock on the frame containing the page table entry, waiting for other processors to release locks on this frame, if necessary. The MMU will then fetch the page table entry, update it, determine if any exceptions will be signaled, write the entry back, and release the lock on the frame. Only after the lock is released, will the MMU perform the memory

The BLITZ Architecture

operation to the target address. Then, if necessary, a lock will be acquired on the second frame (containing the target address) with the CPU waiting again, if necessary. Finally, the memory operation will be performed and the second lock released. If the operation is to fetch a word or a fetch a byte, or update a word, the second lock (on the target frame) may be unnecessary if all transfers to and from the memory are in units of 32-bit words. If the memory operation is the “tset” instruction, a second lock on the target frame must be acquired. If the memory only moves data in units of words, then an update to a single byte (e.g., the “storeb” instruction) will require a lock on the target frame as well, since an entire word must be fetched, the selected portion of the word updated, and the entire word written back to memory. Without using locks, it is possible that a second processor simultaneously updating a byte in the very same word will cause one of the updates to be lost, which is clearly unacceptable.

Note that the “fload” and “fstore” instructions read and write a doubleword (8 bytes) at a time. The “fload” instruction fetches a doubleword from memory and places it in a floating-point register. The “fstore” instruction copies a doubleword from a floating-point register to memory. In both instructions, the doubleword is moved in two separate steps, which each move a word (4 bytes). There is no locking between the two word-length operations. In a multi-processor system, there may be interleaving of these operations. As an example, consider two processors simultaneously attempting to store different doublewords into the very same address; it is possible that the words are stored in such a way that the final two-word result gets its first word from one processor and its second word from the other processor, due to interleaving of memory operations.

Exceptions Common to All Instructions

Note that all instructions may cause the following exceptions:

Page Invalid
Address Exception

When an instruction is fetched, a word from memory must be read to obtain the instruction. If Paging is enabled, then the processor will first consult the Page Table to determine which frame in memory contains the data. If the page containing the instruction is not currently in memory, a page fault will occur, resulting in a Page Invalid exception.

Regardless of whether or not paging is enabled, a physical address will be produced before the instruction word is fetched. If this physical address points to a word that is not in the memory supplied on the specific machine (for instance, the memory address exceeds the amount of installed memory), then an Address Exception will occur.

Since these two exceptions may occur for every instruction, they are not listed separately under each instruction. However, if either of these exceptions could also be caused by attempts to fetch or store operands, that exception is listed explicitly.

Also, any of the asynchronous (hardware) interrupts or the ExceptionDuringInterrupt may occur during any instruction, so these are not listed individually for each instruction.

Alignment Exceptions

Every BLITZ instruction is 32-bits long and must be stored in a word-aligned address in memory.

The BLITZ Architecture

There are several ways that different architectures may enforce such an alignment restriction. The design options are: (1) Check for alignment on every instruction fetch and have Alignment Exceptions always possible. (2) Check for alignment whenever the PC is loaded (e.g., for the “jmp”, “call”, “bxx”, and “reti” instructions) and have an Alignment Exception only for these instructions. (3) Never check for alignment; simply ignore the last 2 bits whenever the PC is loaded or used.

Option (1) requires the overhead of checking the PC on every instruction fetch. This is often wasted effort: if the previous instruction did not cause an Alignment Exception, then the next sequentially fetched instruction cannot possibly cause an exception. Option (3) requires the least overhead: since nothing is ever checked, an Alignment Exception will never occur.

The BLITZ architecture uses option (2). Any instruction that loads the PC will cause an Alignment Exception if the value being loaded into the PC is not divisible by 4. Due to a bug in some user program, it is possible that an attempt will be made to branch to an address that is not the address of a valid instruction sequence. If the bad address happens to be unaligned, it will cause an Alignment Exception and the kernel can then abort the program. (Of course it is possible that an erroneous target of a branching instructions happens to be aligned properly, in which case instruction execution will continue. What happens next depends on what data happens to lie at the target address.)

Several instructions query or modify data in memory. Some instructions move bytes and other instructions move words. Whenever a word is moved to or from memory, the address must be aligned; if it is not, these instructions will cause an Alignment Exception.

The following instructions may cause Alignment Exceptions:

```
load    (Format E)
loadv   (Format E)
store   (Format E)
storev  (Format E)
call
jmp
bxx
push
pop
reti
ret
tset
readu   (Format E)
writeu  (Format E)
ldptbr
ldptlr
fload   (Format E)
fstore  (Format E)
```

PowerOnReset and HardwareFault

When first powered on, the BLITZ CPU begins with a PowerOnReset interrupt. At this time, volatile memory may or may not contain meaningful values, and the code that is branched to as a result of the PowerOnReset should lie in non-volatile memory. Typically, this will be “boot” code stored in read-only memory (ROM).

The BLITZ Architecture

When the BLITZ emulator is used, the memory will be preloaded with a program (e.g., the OS kernel). When execution is initiated (e.g., with the “go” command), the PowerOnReset interrupt processing will occur.

During normal CPU execution, the BLITZ hardware may continually perform a sequence of error and consistency checks in parallel with normal instruction execution. Normally, the software will be unaware of these checks. However, if a problem is detected by this error-checking hardware, it will be signaled by the generation of a HardwareFault interrupt. The interrupt handler might, for example, attempt a graceful shutdown of the operating system and display a message that a hardware fault has been detected.

In both the PowerOnReset and HardwareFault interrupts, we cannot assume that the system stack register has been properly initialized. An attempt to push onto this stack might result in an ExceptionDuringInterrupt. Thus, for both PowerOnReset and HardwareFault, nothing will be pushed onto the stack and the stack pointer (“r15”) will be ignored. This makes any return to the interrupted process impossible. (Of course, with a PowerOnReset, there will not even be an interrupted process.)

Also, in the case of both PowerOnReset and HardwareFault interrupts, any other pending interrupts are ignored and cancelled. This is done to prevent the code invoked by the interrupt from being immediately interrupted by another interrupt, which is a distinct possibility in the case of the HardwareFault. Of course another interrupt may occur just after the HardwareFault handler begins execution, thereby confounding the shutdown sequence. To prevent this, the expectation is that the electrical signal that generates the PowerOnReset and HardwareFaults will be sent simultaneously as a “reset” signal to all peripheral devices capable of generating interrupts.

Synthetic Instructions

Several instructions documented below are called “synthetic” instructions. These instructions are not actually part of the BLITZ architecture. Instead, they are simple abbreviations for other BLITZ instructions. Normal, non-synthetic BLITZ instructions are referred to as “true” instructions.

Synthetic instructions are processed entirely by the Assembler. The assembler recognizes and accepts the synthetic instructions. The assembler produces a similar true BLITZ instruction, which performs the desired operation. Synthetic instructions are provided merely to make the programming task easier for assembly language programmers. This approach is intended to reduce the size and complexity of the BLITZ processor design and implementation.

For some instructions, there is a synthetic version in addition to one or more true BLITZ forms. The example, the “load word” instruction has the following two “true” formats:

```
load    [regA+regB],regC    ! Format D
load    [regA+data16],regC  ! Format E
```

In addition, the following synthetic form is accepted by the assembler:

```
load    [regA],regC
```

For this instruction, the assembler will produce a Format D instruction, using register “r0” (which always contains zero) for the missing regB. In other words, the assembler will output the following instruction.

The BLITZ Architecture

```
load      [regA+r0],regC
```

For other instructions, there is only a synthetic form and no “true” form. For example, the “move into register” instruction is translated by the assembler into an “or” instruction, using register “r0” as the second operand. (Since “r0” contains zeros, and since “x OR zero” is always just x, this simply moves the value unchanged.) The “move into register” instruction has the following two forms:

Synthetic instruction:	Assembled identically to:
=====	=====
mov regA,regC	or regA,r0,regC
mov data16,regC	or r0,data16,regC

One synthetic instruction (namely “set”) is assembled into two true BLITZ instructions, namely “sethi” and “setlo”. All other synthetic instructions are assembled into a single true BLITZ instruction.

Although synthetic instructions are not true BLITZ instructions, they are described in this document for convenience. BLITZ processors and emulators are completely ignorant of synthetic instructions; they see only true instructions.

Software tools (primarily the debugger) will occasionally disassemble memory contents, working in the reverse direction of an assembler by taking a 32-bit word, interpreting it as a BLITZ instruction, and attempting to produce a human-readable form. One consequence of using synthetic instructions is that a debugger will show only “true” instructions, not the synthetic instructions originally coded by the programmer.

The BLITZ Instruction Set: Notation

The next sections describe the instructions available in the BLITZ architecture.

Each section describes one, or several related, instructions. In the discussion, we use a number of abbreviations:

```
regA
regB
regC
data16
data24
```

The terms “regA”, “regB”, and “regC” stand for any of the general purpose registers, namely registers “r0”, “r1”, ... , “r15”.

In general, the destination register or the register that is modified is denoted by “regC” and the data movement tends to be from left to right. For example, in this instruction:

```
add      regA,regB,regC
```

the values in the first two registers are added and moved into the last register. This instruction is given in general form; examples of this instruction include the following:

The BLITZ Architecture

```
add    r5,r9,r2
add    r7,r2,r0
```

Instructions in formats E and G contain a 16-bit field for an immediate value. Instructions in format F contain a 24-bit field for immediate data. These are denoted in the instruction descriptions with “data16” and “data24”. In most cases, the data in the fields will be sign-extended, but there are exceptions. Consult the specific instruction descriptions.

When a field is sign-extended, its most significant bit is copied into the high-order bits to fill it out to a full 32 bits. This means that the immediate fields can hold a signed value. In the case of “data16”, the value must lie between $-32,768$ and $+32,767$. In the case of “data24”, the value must lie between $-8,388,608$ and $+8,388,607$.

In some cases, the value is used as is, after being sign-extended. In this case, the data is considered to be “absolute” (i.e., will not change, even if the program and data are relocated to another area in memory).

In other cases, the immediate value is first added to a register before being used. For some instructions, this will be a general purpose register. For other instructions, the value will be signed-extended and added to the current value of the PC register, giving a “relative” address in memory, since the actual value used will change if the program and data are relocated to another area in memory.

Note that many of the instructions do not query or modify memory, while a few of the instructions will query or modify memory. The instructions that access memory all contain brackets around those operands that specify the address to be queried or modified. For example, the following instruction will store into memory; the address portion of the instruction is in brackets.

```
store    r7,[r4+24]
```

If you see brackets, you know the instruction will read or update memory; if there are no brackets, memory will not be used to execute the instruction. The only exception to this rule is the short forms of the “push” and “pop” instructions.

```
push     r7           ! Modifies memory
pop      r3           ! Queries memory at location [r14]
```

The BLITZ Architecture

Add

```
add    regA,regB,regC
add    regA,data16,regC
```

The contents of register regB or the immediate data value is added to the contents of regA and the result is placed in register regC. The immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between -32768 and 32767 . The 32-bit result of binary addition is the same regardless of whether the data is interpreted as signed or unsigned. The condition codes are set based on signed arithmetic. If the mathematically correct result of the addition is not representable as a 32-bit two's complement integer, the overflow bit in the Status Register will be set.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 96 (hex 60)

Format E: 128 (hex 80)

Possible Exceptions:

None

Subtract

```
sub    regA,regB,regC
sub    regA,data16,regC
```

The contents of register regB or the immediate data value is subtracted from the contents of regA and the result is placed in register regC. The immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between -32768 and 32767 . The 32-bit result of binary subtraction is the same regardless of whether the data is interpreted as signed or unsigned. The condition codes are set based on signed arithmetic. If the mathematically correct result of the subtraction is not representable as a 32-bit two's complement integer, the overflow bit in the Status Register will be set.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 97 (hex 61)

Format E: 129 (hex 81)

Possible Exceptions:

None

Multiply

```
mul    regA,regB,regC
mul    regA,data16,regC
```

The contents of register regB or the immediate data value is multiplied by the contents of regA and the result is placed in register regC. The immediate value is encoded as a sign-extended 16-

The BLITZ Architecture

bit value; therefore it must lie between -32768 and 32767 . All values are treated as signed quantities. The 32-bit result will be stored in regC. (In general, the multiplication of two 32-bit signed numbers will result in a 64-bit quantity, however this instruction only retains 32 bits of result.) The condition codes will be set according to the result of the computation. In particular, if the result of the multiplication is not representable as a 32-bit two's complement integer, the overflow bit in the Status Register will be set.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 98 (hex 62)

Format E: 130 (hex 82)

Possible Exceptions:

None

Divide

```
div    regA, regB, regC
div    regA, data16, regC
```

The contents of register regB or the immediate data value is divided into the contents of regA and the result is placed in register regC. The immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between -32768 and 32767 . All values are treated as signed quantities. The condition codes will be set according to the result of the computation.

The “div” and “rem” instructions are passed two integers (“a” and “b”). They divide a by b to get a quotient (“q”) and remainder (“r”). The “div” instruction moves “q” into regC and the “rem” instruction moves “r” into regC.

Integer division is defined such that the following is always true:

$$a = b * q + r$$

Furthermore, the remainder follows the mathematical definition of the “modulo” operator, namely that the remainder will have the same sign as b and that

$$0 \leq \text{abs}(r) < \text{abs}(b)$$

Another way to look at this is that the quotient is the real quotient, rounded down to the nearest integer.

For example:

a	b	q	r	a = b * q + r	a/b	rounded
==	==	==	==	=====	=====	=====
7	3	2	1	7 = 3 * 2 + 1	2.3	2
-7	3	-3	2	-7 = 3 * -3 + 2	-2.3	-3
7	-3	-3	-2	7 = -3 * -3 + -2	-2.3	-3
-7	-3	2	-1	-7 = -3 * 2 + -1	2.3	2

With this definition of “q” and “r”, problems can and will occur in exactly two situations.

The BLITZ Architecture

The first problem is if “b” is zero (i.e., an attempt to divide by zero). There are no integers “q” and “r” that will satisfy the equations listed above. In this case, an Arithmetic Exception will be raised.

The second problem occurs with the following values:

```
a = -2147483648
b = -1
```

The mathematically correct answer is:

```
q = +2147483648
r = 0
```

Unfortunately, this value of q is not representable as a 32-bit two’s complement integer. In this case, the following values will be used and the overflow bit in the Status Register will be set.

```
q = -2147483648
r = 0
```

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 99 (hex 63)

Format E: 131 (hex 83)

Possible Exceptions:

Arithmetic Exception

Remainder

```
rem      regA,regB,regC
rem      regA,data16,regC
```

The contents of register regB or the immediate data value is divided into the contents of regA and the remainder is placed in register regC. The immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between –32768 and 32767. All values are treated as signed quantities. The condition codes will be set according to the result of the computation

See the comments under the “div” instruction.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 115 (hex 73)

Format E: 149 (hex 95)

Possible Exceptions:

Arithmetic Exception

Shift Left Logical

```
sll    regA, regB, regC
sll    regA, data16, regC
```

The contents of register regA is shifted left by N bits and the result is placed in regC. N is a number from 0 through 31 and may be in register regB or may be specified literally. All bits except the least significant 5 bits are ignored, giving a number between 0 and 31. If N is zero, the data is not shifted. Zeros will be shifted in from the right. The condition codes will be set according to the result of the computation.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 100 (hex 64)

Format E: 132 (hex 84)

Possible Exceptions:

None

Shift Right Logical

```
srl    regA, regB, regC
srl    regA, data16, regC
```

The contents of register regA is shifted right by N bits and the result is placed in regC. N is a number from 0 through 31 and may be in register regB or may be specified literally. All bits except the least significant 5 bits are ignored, giving a number between 0 and 31. If N is zero, the data is not shifted. Zeros will be shifted in from the left. The condition codes will be set according to the result of the computation.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 101 (hex 65)

Format E: 133 (hex 85)

Possible Exceptions:

None

Shift Right Arithmetic

```
sra    regA, regB, regC
sra    regA, data16, regC
```

The contents of register regA is shifted right by N bits and the result is placed in regC. N is a number from 0 through 31 and may be in register regB or may be specified literally. All bits except the least significant 5 bits are ignored, giving a number between 0 and 31. If N is zero, the data is not shifted. The sign bit will be duplicated and shifted in from the left, thus preserving the sign of the value and implementing a signed divide by a power of 2. The condition codes will be set according to the result of the computation.

The BLITZ Architecture

Condition Codes:
 Modified
Privileged:
 No
Opcodes:
 Format D: 102 (hex 66)
 Format E: 134 (hex 86)
Possible Exceptions:
 None

Or

```
or      regA,regB,regC
or      regA,data16,regC
```

The contents of register regB or the immediate data value is logically ORed with the contents of regA and the result is placed in register regC. Note that the immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between -32768 and 32767 . The condition codes will be set according to the result of the computation.

Condition Codes:
 Modified
Privileged:
 No
Opcodes:
 Format D: 103 (hex 67)
 Format E: 135 (hex 87)
Possible Exceptions:
 None

And

```
and      regA,regB,regC
and      regA,data16,regC
```

The contents of register regB or the immediate data value is logically ANDed with the contents of regA and the result is placed in register regC. Note that the immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between -32768 and 32767 . The condition codes will be set according to the result of the computation.

Condition Codes:
 Modified
Privileged:
 No
Opcodes:
 Format D: 104 (hex 68)
 Format E: 136 (hex 88)
Possible Exceptions:
 None

The BLITZ Architecture

And-Not

```
andn    regA,regB,regC
andn    regA,data16,regC
```

The contents of register regB or the immediate data value is first logically NEGATED. The resulting 32-bit value is then logically ANDed with the contents of regA and the result is placed in register regC. Note that the immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between -32768 and 32767. The condition codes will be set according to the result of the computation.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 105 (hex 69)

Format E: 137 (hex 89)

Possible Exceptions:

None

Exclusive-Or

```
xor     regA,regB,regC
xor     regA,data16,regC
```

The contents of register regB or the immediate data value is logically EXCLUSIVE-ORed with the contents of regA and the result is placed in register regC. Note that the immediate value is encoded as a sign-extended 16-bit value; therefore it must lie between -32768 and 32767. The condition codes will be set according to the result of the computation.

Condition Codes:

Modified

Privileged:

No

Opcodes:

Format D: 106 (hex 6A)

Format E: 138 (hex 8A)

Possible Exceptions:

None

Compare

Synthetic instruction:	Assembled identically to:
=====	=====
cmp regA,regB	sub regA,regB,r0
cmp regA,data16	sub regA,data16,r0

In the first form, the contents of regA are compared to the contents of register regB. In the second form, a 16-bit immediate value is sign extended to 32 bits. Register regA is compared with this value.

The BLITZ Architecture

This instruction will normally be following by a conditional branch instruction. In the following example code, the branch to “label” would be taken if the value in register r4 is less than 100.

```
cmp    r4,100
bl     label
```

Condition Codes:

Modified to reflect the comparison

Privileged:

No

Opcodes:

(see the "sub" instruction)

Possible Exceptions:

None

Arithmetic Negation

Synthetic instruction:

=====

```
neg    regB,regC
neg    regC
```

Assembled identically to:

=====

```
sub    r0,regB,regC
sub    r0,regC,regC
```

This instruction negates a word, using 2's complement, 32-bit arithmetic. In the first form, the value in register regB is negated and is moved into register regC. In the second form, the value in register regC is negated and returned to the same register. In either case, the condition codes are set to reflect the result.

Note that an attempt to negate the most negative value (0x80000000) will result in the same value, rather than the mathematically correct answer. (Due to the nature of 2's complement arithmetic, there is always one more negative integer than there are positive integers; thus one integer, when negated, cannot be represented in the same number of bits.)

Condition Codes:

Modified

Privileged:

No

Opcodes:

(see the "sub" instruction)

Possible Exceptions:

None

Logical Negation

Synthetic instruction:

=====

```
not    regA,regC
not    regC
```

Assembled identically to:

=====

```
xor    regA,0xFFFF,regC
xor    regC,0xFFFF,regC
```

This instruction logically negates a 32-bit word, flipping each bit. In the first form, the value in register regA is negated and is moved into register regC. In the second form, the value in register regC is negated and returned to the same register. In either case, the condition codes are set to reflect the result.

The BLITZ Architecture

Condition Codes:

Modified

Privileged:

No

Opcodes:

(see the "xor" instruction)

Possible Exceptions:

None

Clear

Synthetic instruction:

=====

clr regC

Assembled identically to:

=====

or r0,r0,regC

This instruction moves 32 bits of zeros into register regC.

Condition Codes:

Modified (Z:=1, N:=0, V:=0, C:=0)

Privileged:

No

Opcodes:

(see the "or" instruction)

Possible Exceptions:

None

Bit Test

Synthetic instruction:

=====

btst regA,regB

btst data16,regB

Assembled identically to:

=====

and regA,regB,r0

and regB,data16,r0

This instruction tests bits in register regB to see if any are set to “1” and sets the condition codes accordingly. In the first form, register regA should contain a “mask” with a “1” in every bit position of interest. In the second form, the instruction contains 16-bits of immediate data and these comprise the mask.

After the test, a “be” branch will be taken if ALL the bits selected by the mask were zero, while a “bne” branch will be taken if ANY of the bits selected by the mask were one.

Note that in the second form, the immediate value will be sign-extended before being used.

Condition Codes:

Modified

Privileged:

No

Opcodes:

(see the "and" instruction)

Possible Exceptions:

None

The BLITZ Architecture

Bit Set

Synthetic instruction:	Assembled identically to:
=====	=====
bset regB,regC	or regC,regB,regC
bset data16,regC	or regC,data16,regC

This instruction sets selected bits in register regC. In the first form, register regB should contain a “mask” with a “1” in every bit position of interest. In the second form, the instruction contains 16-bits of immediate data and these comprise the mask. All bits in register regC that were selected by the mask will be set to “1”, while the other bits in register regC will remain unchanged.

Note that in the second form, the immediate value will be sign-extended before being used.

Condition Codes:

Modified

Privileged:

No

Opcodes:

(see the "or" instruction)

Possible Exceptions:

None

Bit Clear

Synthetic instruction:	Assembled identically to:
=====	=====
bclr regB,regC	andn regC,regB,regC
bclr data16,regC	andn regC,data16,regC

This instruction clears selected bits in register regC. In the first form, register regB should contain a “mask” with a “1” in every bit position of interest. In the second form, the instruction contains 16-bits of immediate data and these comprise the mask. All bits in register regC that were selected by the mask will be cleared to “0”, while the other bits in register regC will remain unchanged.

Note that in the second form, the immediate value will be sign-extended before being used.

Condition Codes:

Modified

Privileged:

No

Opcodes:

(see the "andn" instruction)

Possible Exceptions:

None

Bit Toggle

Synthetic instruction:	Assembled identically to:
=====	=====
btog regB,regC	xor regC,regB,regC
btog data16,regC	xor regC,data16,regC

This instruction flips selected bits in register regC. In the first form, register regB should contain a “mask” with a “1” in every bit position of interest. In the second form, the instruction contains 16-bits of immediate data and these comprise the mask. All bits in register regC that were selected by the mask will be flipped from “0” to “1” or vice versa, while the other bits in register regC will remain unchanged.

Note that in the second form, the immediate value will be sign-extended before being used.

Condition Codes:

Modified

Privileged:

No

Opcodes:

(see the "xor" instruction)

Possible Exceptions:

None

Move Into Register

Synthetic instruction:	Assembled identically to:
=====	=====
mov regA,regC	or regA,r0,regC
mov data16,regC	or r0,data16,regC

In the first form, the contents of regA are moved into register regC. In the second form, a 16-bit immediate value is sign extended to 32 bits and moved into register regC.

Condition Codes:

Modified to reflect the value moved

Privileged:

No

Opcodes:

(see the "or" instruction)

Possible Exceptions:

None

Load Word

```
load        [regA+regB],regC
load        [regA+data16],regC
```

Synthetic instruction:	Assembled identically to:
=====	=====
load [regA],regC	load [regA+r0],regC

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give

The BLITZ Architecture

an address. In either case, the address must be word aligned, or an alignment exception will occur. A 32-bit (4 byte) word is moved from the memory location given by the address into register regC.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 107 (hex 6B)

Format E: 139 (hex 8B)

Possible Exceptions:

Address Exception

Page Invalid

Alignment

Load Byte

```
loadb    [regA+regB],regC
loadb    [regA+data16],regC
```

Synthetic instruction:

=====

```
loadb    [regA],regC
```

Assembled identically to:

=====

```
loadb    [regA+r0],regC
```

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. A single byte is moved from the memory location given by the address into the least significant bits of register regC. The high-order 24-bits of the register are cleared to zero.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 108 (hex 6C)

Format E: 140 (hex 8C)

Possible Exceptions:

Address Exception

Page Invalid

Load Word Virtual

```
loadv    [regA+regB],regC
loadv    [regA+data16],regC
```

Synthetic instruction:

=====

```
loadv    [regA],regC
```

Assembled identically to:

=====

```
loadv    [regA+r0],regC
```

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. In either case, the address must be word aligned, or an alignment exception will

The BLITZ Architecture

occur. A 32-bit (4 byte) word is moved from the memory location given by the address into register regC.

The operation of this instruction differs from the “load” instruction only when the Page Table is disabled. When the Page Table is disabled, the “load” instruction will generate an address which will be used as a physical address to access physical memory directly; no address translation will occur. However, the “loadv” instruction will perform address translation, even though the Page Table is disabled. The instruction is intended to be used by the kernel to fetch arguments and other data from the logical address space of a user-level program. It will cause a Page Invalid exception if there are problems in accessing the virtual address space.

Condition Codes:

Not affected

Privileged:

Yes

Opcodes:

Format D: 109 (hex 6D)

Format E: 141 (hex 8D)

Possible Exceptions:

Address Exception

Page Invalid

Alignment

Privileged Instruction

Load Byte Virtual

```
loadbv    [regA+regB],regC
loadbv    [regA+data16],regC
```

Synthetic instruction:

=====

```
loadbv    [regA],regC
```

Assembled identically to:

=====

```
loadbv    [regA+r0],regC
```

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. A single byte is moved from the memory location given by the address into the least significant bits of register regC. The high-order 24-bits of the register are cleared to zero.

The operation of this instruction differs from the “loadb” instruction only when the Page Table is disabled. When the Page Table is disabled, the “loadb” instruction will generate an address which will be used as a physical address to access physical memory directly; no address translation will occur. However, the “loadbv” instruction will perform address translation, even though the Page Table is disabled. The instruction is intended to be used by the kernel to fetch arguments and other data from the logical address space of a user-level program. It will cause a Page Invalid exception if there are problems in accessing the virtual address space.

The BLITZ Architecture

Condition Codes:

Not affected

Privileged:

Yes

Opcodes:

Format D: 110 (hex 6E)

Format E: 142 (hex 8E)

Possible Exceptions:

Address Exception

Page Invalid

Privileged Instruction

Store Word

```
store    regC,[regA+regB]
store    regC,[regA+data16]
```

Synthetic instruction:

=====

```
store    regC,[regA]
```

Assembled identically to:

=====

```
store    regC,[regA+r0]
```

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. In either case, the address must be word aligned, or an alignment exception will occur. A 32-bit (4 byte) word is moved from register regC to the memory location given by the address. A Page Read-only exception will occur if the Page Table is enabled and the page containing the address is marked read-only.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 111 (hex 6F)

Format E: 143 (hex 8F)

Possible Exceptions:

Address Exception

Page Invalid

Page Read-only

Alignment

Store Byte

```
storeb   regC,[regA+regB]
storeb   regC,[regA+data16]
```

Synthetic instruction:

=====

```
storeb   regC,[regA]
```

Assembled identically to:

=====

```
storeb   regC,[regA+r0]
```

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. A single byte is moved from register regC to the memory location given by the

The BLITZ Architecture

address. A Page Read-only exception will occur if the Page Table is enabled and the page containing the address is marked read-only.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 112 (hex 70)

Format E: 144 (hex 90)

Possible Exceptions:

Address Exception

Page Invalid

Page Read-only

Store Word Virtual

```
storev    regC,[regA+regB]
storev    regC,[regA+data16]
```

Synthetic instruction:

=====

```
storev    regC,[regA]
```

Assembled identically to:

=====

```
storev    regC,[regA+r0]
```

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. In either case, the address must be word aligned, or an alignment exception will occur. A 32-bit (4 byte) word is moved from register regC to the memory location given by the address.

The operation of this instruction differs from the “store” instruction only when the Page Table is disabled. When the Page Table is disabled, the “store” instruction will generate an address which will be used as a physical address to access physical memory directly; no address translation will occur. However, the “storev” instruction will perform address translation, even though the Page Table is disabled. The instruction is intended to be used by the kernel to store results and other data into the logical address space of a user-level program. It will cause a Page Invalid exception or a Page Read-only exception if there are problems in accessing the virtual address space.

Condition Codes:

Not affected

Privileged:

Yes

Opcodes:

Format D: 113 (hex 71)

Format E: 145 (hex 91)

Possible Exceptions:

Address Exception

Page Invalid

Page Read-only

Alignment

Privileged Instruction

Store Byte Virtual

```
storebv  regC,[regA+regB]
storebv  regC,[regA+data16]
```

Synthetic instruction:	Assembled identically to:
=====	=====
storebv regC,[regA]	storebv regC,[regA+r0]

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. A single byte is moved from register regC to the memory location given by the address.

The operation of this instruction differs from the “storeb” instruction only when the Page Table is disabled. When the Page Table is disabled, the “storeb” instruction will generate an address which will be used as a physical address to access physical memory directly; no address translation will occur. However, the “storebv” instruction will perform address translation, even though the Page Table is disabled. The instruction is intended to be used by the kernel to store results and other data into the logical address space of a user-level program. It will cause a Page Invalid exception or a Page Read-only exception if there are problems in accessing the virtual address space.

Condition Codes:

Not affected

Privileged:

Yes

Opcodes:

Format D: 114 (hex 72)

Format E: 146 (hex 92)

Possible Exceptions:

Address Exception

Page Invalid

Page Read-only

Privileged Instruction

Call

```
call     regA+regC
call     data24
```

Synthetic instruction:	Assembled identically to:
=====	=====
call regA	call regA+r0

In the first form, the contents of regA and regB are added together to give a target address. In the second form, a 24-bit immediate value is sign extended and added to the address of the “call” instruction itself to give a target address.

This instruction is used to invoke a subroutine. First, this instruction pushes the address of the instruction following the “call” instruction onto the stack. Then the “call” instruction moves the target address into the program counter (PC). Therefore, the next instruction to be executed will be the first instruction of the subroutine.

The BLITZ Architecture

The target address should be word-aligned; if it is not divisible by 4, this instruction will cause an Alignment Exception.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format C: 64 (hex 40)

Format F: 160 (hex A0)

Possible Exceptions:

Alignment

Address Exception

Page Invalid

Page Read-only

Jmp

jmp regA+regC

jmp data24

Synthetic instruction:

=====

jmp regA

Assembled identically to:

=====

jmp regA+r0

In the first form, the contents of regA and regB are added together to give a target address. In the second form, a 24-bit immediate value is sign extended and added to the address of the “jmp” instruction itself to give a target address.

This instruction is the unconditional “goto” instruction. This instruction moves the target address into the program counter (PC), causing control to transfer to this address.

The target address should be word-aligned; if it contains a value not divisible by 4, this instruction will cause an Alignment Exception.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format C: 65 (hex 41)

Format F: 161 (hex A1)

Possible Exceptions:

Alignment

Return

ret

This instruction is used to return from a subroutine call. (We assume that, when the routine was originally called, the “call” instruction pushed the return address onto the stack. This was the address of the instruction following the “call” instruction.) The “ret” instruction will pop the top of the stack into the PC, thereby effecting a transfer of control back to the calling routine.

The BLITZ Architecture

The stack pointer should be word-aligned. If register r15 contains a value not divisible by 4, this instruction will cause an Alignment Exception. All instructions must be word aligned. If the address popped into PC is not divisible by 4, it will cause an Alignment Exception.

Condition Codes:

Not affected

Privileged:

No

Opcode:

Format A: 9 (hex 09)

Possible Exceptions:

Alignment

Address Exception

Page Invalid

Page Read-only

Conditional Branching

bxx regA+regC
bxx data24

Synthetic instruction:

=====

bxx regA

Assembled identically to:

=====

bxx regA+r0

There are 36 different branch instructions (such as “be”, “bne”, “bl”, “ble”, and so on). The above forms are schematic: there is no “bxx” instruction.

In the first form, the contents of regA and regB are added together to give a target address. In the second form, a 24-bit immediate value is sign extended and added to the address of the branch instruction itself to give a target address.

Each branch instruction is a conditional “goto”. The instruction first tests the condition. (Each instruction tests for a different condition. See the table below.) If the condition is true, the instruction moves the target address into the program counter (PC), causing control to transfer to the target address. If the condition is false, the PC is incremented as normal, causing the next sequential instruction to be executed.

All instructions must be word aligned. If register r15 contains a value not divisible by 4, this instruction will cause an Alignment Exception. The Alignment Exception will only occur if the branch is taken.

In the following table, I, S, P, Z, V, and N represent the values of the various bits in the status register.

The BLITZ Architecture

opcode	condition tested	meaning
=====	=====	=====
be	Z	equal to, equal to zero
bne	NOT Z	not equal to, not equal to zero
bl	N XOR V	less than, negative
ble	Z OR (N XOR V)	less than or equal to, negative or zero
bg	NOT (Z OR (N XOR V))	greater than, positive
bge	NOT (N XOR V)	greater than or equal to, positive or zero
bvs	V	overflow occurred
bvc	NOT V	overflow did not occur
bns	N	negative result
bnc	NOT N	non-negative result
bss	S	in system mode
bsc	NOT S	in user mode
bis	I	interrupts enabled
bic	NOT I	interrupts disabled
bps	P	page table enabled
bpc	NOT P	page table disabled

Condition Codes:

Not changed

Privileged:

No

Opcodes:

Format C: 66-83 (hex 42-53)

Format F: 162-179 (hex A2-B3)

Possible Exceptions:

Alignment

Push

push regC, [--regA]

Synthetic instruction:

=====

push regC

Assembled identically to:

=====

push regC, [--r15]

This instruction first decrements the contents of register regA by 4. Then, it moves the contents of register regC into the memory word addressed by the new value of regA.

The push and pop instructions are used to support stacks that grow downward in memory, from higher addresses toward low memory. Register regA is the “stack top” pointer; it always points to the last item pushed onto the stack. The push instruction decrements the stack top pointer before writing a word to memory, while the pop instruction retrieves a word from memory before incrementing the stack top pointer.

By convention, register r15 will point to a stack used during procedure call and return, although nothing prevents other registers from being used simultaneously to address other stacks. To support the use of r15 as a stack register, the assembler provides a synthetic version of push and pop that specifies r15 implicitly.

Note that the push and pop instructions have utility beyond just stack operations. These instructions may also be used to walk through an array of words, while simultaneously adjusting

The BLITZ Architecture

a pointer to the next word. The push instruction can be used to both store a word of data and decrement the pointer in one operation, working downwards through the array (towards low memory). The pop instruction can be used to fetch a word of data and increment a pointer in one operation, working upwards through the array (towards high memory).

The stack should always be word-aligned. If register regA is not word aligned, an Alignment Exception will be raised.

Condition Codes:

Not affected

Privileged:

No

Opcode:

Format C: 84 (hex 54)

Possible Exceptions:

Address Exception

Page Invalid

Page Read-only

Alignment

Pop

pop [regA++], regC

Synthetic instruction:

=====

pop regC

Assembled identically to:

=====

pop [r15++], regC

This instruction moves the contents of the word pointed to by register regA into register regC. It then increments register regA by 4.

See the comments about stacks under the pop instruction.

The stack should always be word-aligned. If register regA is not word aligned, an Alignment Exception will be raised.

Condition Codes:

Not affected

Privileged:

No

Opcode:

Format C: 85 (hex 55)

Possible Exceptions:

Address Exception

Page Invalid

Alignment

Setlo

setlo data16, regC

This instruction moves the 16 bits of immediate data into the low-order (least significant) 16 bits of register regC. The high-order 16 bits of the register are unchanged.

The BLITZ Architecture

Condition Codes:
 Not affected
Privileged:
 No
Opcode:
 Format G: 193 (hex C1)
Possible Exceptions:
 None

Sethi

```
sethi    data16,regC
```

This instruction moves the 16 bits of immediate data into the high-order (most significant) 16 bits of register regC. The low-order 16 bits of the register are unchanged.

Note that the assembler expects a 32-bit value in this instruction. It will use only the hi-order 16 bits and will ignore the lo-order bits. For example, the instruction

```
sethi    0x12340000,r5
```

will load the data 0x1234 into the register as desired, while the instruction

```
sethi    0x1234,r5
```

will load only zeros, which is probably not intended. This approach allows the programmer to use external symbols, as in:

```
sethi    myExtSym,r5  
setlo    myExtSym,r5
```

Condition Codes: Not affected
Privileged:
 No
Opcode:
 Format G: 192 (hex C0)
Possible Exceptions:
 None

Set

Synthetic instruction:	Assembled identically to:
=====	=====
set data32,regC	sethi HI(data32),regC
	setlo LO(data32),regC

This instruction moves a 32-bit immediate value into register regC. It is a synthetic instruction which expands into two BLITZ instructions. Here, we use “HI(data32)” and “LO(data32)” to mean the high-order 16 bits, and the low-order 16 bits, of the immediate 32 bit value given. No sign extension is performed.

The BLITZ Architecture

Condition Codes:
 Not affected
Privileged:
 No
Opcodes:
 (see the "sethi" and "setlo" instructions)
Possible Exceptions:
 None

Load Address

```
ldaddr    data16,regC
```

This instruction is provided with a 16 bit immediate value. This value is sign-extended and added to the contents of the PC. Specifically, it is added to the PC before it is incremented; i.e., it is added to the address of the ldaddr instruction itself. The result is placed in register regC.

Condition Codes:
 Not affected
Privileged:
 No
Opcode:
 Format G: 194 (hex C2)
Possible Exceptions:
 None

System Call

```
syscall   regC+data16
```

Synthetic instruction:	Assembled identically to:
=====	=====
syscall regC	syscall regC+0
syscall data16	syscall r0+data16

The immediate 16 bit value is sign-extended to 32 bits and is then added to the contents of register regC. This value is called the “system trap number”. A “syscall” trap then occurs, as follows:

First, a copy is made of the current status register and the current PC (after being incremented to point to the instruction following the “syscall” instruction)

Then, the following bits in the status register are changed:

```
I := 0 (Interrupts are disabled)
S := 1 (Mode is changed to System)
P := 0 (Paging is turned off)
```

Then, the system trap number is pushed onto the system stack. (Since the Mode has been changed to System Mode, all pushes will be onto the system stack, pointed to by system register r15.) Then the old value of the status register is pushed onto the stack. Then the old value of the PC (after being incremented to point to the instruction following the “syscall” instruction) is pushed onto the system stack.

The BLITZ Architecture

Finally, a branch into the interrupt vector in low memory is made. In other words, the PC is set to point to the “syscall” entry in the interrupt vector. Presumably, this word will contain a “jump” instruction and execution will branch to the “syscall” interrupt handler.

If problems arise during the “syscall” sequence, then an `ExceptionDuringInterrupt` will occur. (In other words, even if an `Address Exception` or `Alignment Exception` occurs while pushing information onto the system stack, these exceptions will not be signaled.)

Condition Codes:
 Not affected
Privileged:
 No
Opcode:
 Format G: 195 (hex C3)
Possible Exceptions:
 Syscall
 ExceptionDuringInterrupt

No-op

`nop`

This instruction does nothing beyond taking up a little time.

Condition Codes:
 Not affected
Privileged:
 No
Opcode:
 Format A: 0 (hex 00)
Possible Exceptions:
 None

Wait

`wait`

This instruction causes the CPU to suspend instruction execution. The CPU will then remain dormant in a low-power state, waiting for interrupts. If and when an interrupt occurs, execution will resume according to the normal exception handling sequence.

Condition Codes:
 Not affected
Privileged:
 Yes
Opcode:
 Format A: 1 (hex 01)
Possible Exceptions:
 Privileged Instruction

Debug

debug

This instruction is only used in conjunction with a virtual machine BLITZ instruction emulator. When running on a real machine, this instruction is equivalent to the “nop” instruction and has no effect. When running on an emulator, this instruction will cause the emulation to be interrupted and a debugger to be invoked.

Condition Codes:

Not affected

Privileged:

No

Opcode:

Format A: 2 (hex 02)

Possible Exceptions:

None

Debug2

debug2

This instruction is only used in conjunction with a virtual machine BLITZ instruction emulator. When running on a real machine, this instruction is equivalent to the “nop” instruction and has no effect.

This instruction performs an “upcall” to the emulator, much as a syscall instruction performs an upcall from a user process to the kernel. This instruction provides kernel code a quick and easy way to print debugging messages from anywhere, including interrupt handlers, without the overhead required of interfacing with an I/O device. This instruction will take a single clock cycle, like other instructions, and execution will continue after this instruction, unless some error occurs.

Register “r1” must contain a function code. Additional arguments may be passed in other registers; details will depend on the function code in r1. The following functions are implemented.

Register r1 = 1: PrintInt. Register “r2” will contain an integer. The emulator will immediately print this value on the terminal.

Register r1 = 2: Print. Register “r2” will contain a pointer to a string of characters and register r3 will contain a count. The emulator will immediately print this string on the terminal.

Register r1 = 3: PrintChar. Register “r2” will contain a character. The emulator will immediately print this character on the terminal.

Register r1 = 4: PrintDouble. Register “f0” will contain a double. The emulator will immediately print this value on the terminal.

Register r1 = 5: PrintBool. Register “r2” should contain either 0x00000000 or 0x00000001. The emulator will immediately print this value as either FALSE or TRUE.

The BLITZ Architecture

Register r1 = 6: PrintHex. Register “r2” will contain an integer. The emulator will immediately print this value in hex, e.g., 0x000012AB.

Condition Codes:

Not affected

Privileged:

No

Opcode:

Format A: 10 (hex 0A)

Possible Exceptions:

None

Clear Interrupt Flag

cleari

This instruction disables interrupt servicing by clearing the “I” bit in the Status Register to zero. In other words, any maskable interrupt occurring after this instruction is executed will not be serviced, but will remain pending. The servicing of unmaskable interrupts will be unaffected by this instruction.

Condition Codes:

Not affected

Privileged:

Yes

Opcode:

Format A: 3 (hex 03)

Possible Exceptions:

Privileged Instruction

Set Interrupt Flag

seti

This instruction enables interrupt servicing by setting the “I” bit in the Status Register to one. If a maskable interrupt is pending when this instruction is executed, interrupt servicing will occur immediately after this instruction completes.

Condition Codes:

Not affected

Privileged:

Yes

Opcode:

Format A: 4 (hex 04)

Possible Exceptions:

Privileged Instruction

Clear Paging Flag

clearp

This instruction disables paging (i.e., virtual memory address translation) by clearing the “P” bit in the Status Register to zero.

Condition Codes:

Not affected

Privileged:

Yes

Opcode:

Format A: 5 (hex 05)

Possible Exceptions:

Privileged Instruction

Set Paging Flag

setp

This instruction enables paging (i.e., virtual memory address translation) by setting the “P” bit in the Status Register to one.

Condition Codes:

Not affected

Privileged:

Yes

Opcode:

Format A: 6 (hex 06)

Possible Exceptions:

Privileged Instruction

Clear System Mode Flag

clears

This instruction returns to User Mode by clearing the “S” bit in the Status Register to zero. Another way to return to User Mode is via the “reti” (return from interrupt) instruction.

(Note that there is no corresponding “sets” instruction; the only way to move from User Mode to System Mode is via an interrupt (e.g., a “syscall” trap). Any instruction to change the System Mode bit ought to be a privileged instruction; but there is no point in a “sets” instruction since any attempt to set the bit would result in a PrivilegedInstruction exception unless the bit was already set.)

The BLITZ Architecture

Condition Codes:
 Not affected
Privileged:
 Yes
Opcode:
 Format A: 7 (hex 07)
Possible Exceptions:
 Privileged Instruction

Return From Interrupt

`reti`

This instruction can be used after an interrupt handler has finished its work. First, the “reti” instruction pops the saved PC off of the stack. (Since “reti” is a privileged instruction, all popping will be done from the system stack, not the user stack.) Then the saved Status Register is popped from the stack. Finally, an additional word (the “exception info word”) is popped and discarded. The PC and Status Register are then loaded from the popped values, which will cause a return to the interrupted process. Reloading the Status Register may change the processor from System Mode to User Mode and may also re-enable Interrupts and re-enable Paging.

Condition Codes:
 Reloaded
Privileged:
 Yes
Opcode:
 Format A: 8 (hex 08)
Possible Exceptions:
 Alignment
 Privileged Instruction
 Address Exception

Test and Set

`tset [regA],regC`

Register regA contains the address of a word in memory. This instruction moves the contents of this word into register regC. Then, it writes the value 0x00000001 into the same word in memory. This is done atomically; there will be no interrupts or exceptions between the reading of memory and the subsequent writing to memory.

This instruction may be used as a synchronization primitive in multiprocessor architectures. In multiprocessor implementations of the BLITZ architecture, no other processors are allowed to query or alter the value of the word of memory during the operation of this instruction. This instruction may also be used for synchronization control in a single processor architecture with multiple threads.

The address in register regA should be word-aligned. If register regA is not word aligned, an Alignment Exception will be raised.

The BLITZ Architecture

Condition Codes:
 Not affected
Privileged:
 No
Opcode:
 Format C: 88 (hex 58)
Possible Exceptions:
 Address Exception
 Page Invalid
 Page Read-only
 Alignment

Read From User Register

```
readu    regC,regA
readu    regC,[regA+data16]
```

Synthetic instruction:	Assembled identically to:
=====	=====
readu regC,[regA]	readu regC,[regA+0]

This instruction is used to retrieve a 32-bit word from a User Register (regC) while running in System Mode. This instruction is privileged. Normally, when running in System Mode, all references to registers will query or modify the System Registers. In the case of this instruction, however, regC refers to a User Register. (Register regA still refers to a System Register.) This instruction might be used, for example, by the kernel to obtain arguments to a system call directly from the user-level process's registers.

In the first form, the word is moved from the User Register regC into System Register regA.

In the second form, the immediate 16-bit value is sign-extended to 32 bits and added to the contents of System Register regA to give an address. The word retrieved from the User Register is moved into this location in memory. In this form, the address should be word-aligned; if not, an Alignment Exception will be raised.

Condition Codes:
 Not affected
Privileged:
 Yes
Opcodes:
 Format C: 86 (hex 56)
 Format E: 147 (hex 93)
Possible Exceptions:
 Privileged Instruction
Possible Exceptions caused by Format E (regC,[regA+data16]) only:
 Alignment
 Address Exception
 Page Invalid
 Page Read-only

Write To User Register

```
writeu    regA,regC
writeu    [regA+data16],regC
```

Synthetic instruction:	Assembled identically to:
=====	=====
writeu [regA],regC	writeu [regA+0],regC

This instruction is used to store a 32-bit word into a User Register (regC) while running in System Mode. This instruction is privileged. Normally, when running in System Mode, all references to registers will query or modify the System Registers. In the case of this instruction, however, regC refers to a User Register. (Register regA still refers to a System Register.) This instruction might be used, for example, by the kernel to store results after a system call directly into the user-level process's registers.

In the first form, a 32-bit word is moved from System Register regA to the User Register regC.

In the second form, the immediate 16-bit value is sign-extended to 32 bits and added to the contents of System Register regA to give an address. A 32-bit word is fetched from this address and is moved into User Register regC. In this form, the address should be word-aligned; if not, an Alignment Exception will be raised.

Condition Codes:

Not affected

Privileged:

Yes

Opcodes:

Format C: 87 (hex 57)

Format E: 148 (hex 94)

Possible Exceptions:

Privileged Instruction

Possible Exceptions caused by Format E ([regA+data16],regC) only:

Alignment

Address Exception

Page Invalid

Load Page Table Base Register

```
ldptbr    regC
```

This instruction moves the contents of register regC into the Page Table Base Register. The Page Table must always be located on a word-aligned boundary; thus, the contents of register regC should be divisible by 4. If the contents of register regC are not divisible by 4, an Alignment Exception will occur.

Condition Codes:

Not affected

Privileged:

Yes

Opcode:

Format B: 32 (hex 20)

Possible Exceptions:

Alignment

Privileged Instruction

Load Page Table Length Register

ldptlr regC

This instruction moves the contents of register regC into the Page Table Length Register. The Page Table Length Register gives the size of the Page Table in bytes. Each Page Table entry is 4 bytes long; thus, the Page Table is always be an even multiple of 4 bytes in length. The contents of register regC should be divisible by 4. If not, an Alignment Exception will occur..

Condition Codes:

Not affected

Privileged:

Yes

Opcode:

Format B: 33 (hex 21)

Possible Exceptions:

Alignment

Privileged Instruction

Floating-Point to Integer Conversion

ftoi fregA, regC

The floating-point value in floating-point register fregA is converted into an integer value and the result is placed in integer register regC. The condition codes are not modified.

The conversion from a floating-point to an integer is done by truncating the value to the nearest integer toward zero. Many double-precision values exceed the range of integers representable with 32-bit integers; in this case the largest or smallest integer value will be used. These rules can be summarized with these examples:

4.9	-->	4
-4.9	-->	-4
9e99	-->	2,147,483,647
-9e99	-->	-2,147,483,648

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format C: 89 (hex 59)

Possible Exceptions:

None

Integer to Floating-Point Conversion

itof regA, fregC

The integer value in integer register regA is converted into a floating-point value (with the same numerical value) and the result is placed in floating-point register fregC. The condition codes are not modified.

Note that every integer representable in 32-bits can be represented with exact precision with a double-precision floating-point value. Therefore, this instruction will always produce an exactly correct conversion, and never an approximately correct result.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format C: 90 (hex 5A)

Possible Exceptions:

None

Floating-Point Add

fadd fregA, fregB, fregC

The contents of the register fregB is added to the contents of fregA and the result is placed in register fregC. All registers are floating-point registers. The condition codes are not modified.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 116 (hex 74)

Possible Exceptions:

None

Floating-Point Subtract

fsub fregA, fregB, fregC

The contents of the register fregB is subtracted from the contents of fregA and the result is placed in register fregC. All registers are floating-point registers. The condition codes are not modified.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 117 (hex 75)

Possible Exceptions:

None

Floating-Point Multiply

fmul fregA, fregB, fregC

The contents of the register fregB is multiplied by the contents of fregA and the result is placed in register fregC. All registers are floating-point registers. The condition codes are not modified.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 118 (hex 76)

Possible Exceptions:

None

Floating-Point Divide

fdiv fregA, fregB, fregC

The contents of the register fregB is divided into the contents of fregA and the result is placed in register fregC. All registers are floating-point registers. The condition codes are not modified. In the case of divide-by-zero, the stored result will be NAN, PLUS_INF, or NEG_INF, as appropriate.

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 119 (hex 77)

Possible Exceptions:

None

Floating-Point Compare

fcmp fregA, fregC

The contents of the register fregA is compared to the contents of fregC. Both registers are floating-point registers. Neither register is modified, but the condition codes are set to reflect the relationship between the values. Normally, this instruction would be immediately followed by a conditional branch instruction.

This instruction will set Z=1 if the values in fregA and fregC are equal. It will set N=1 if the value in fregA is less than the value in fregC. It will set V=1 if either fregA or fregC contains NAN (the “not-a-number” value).

The BLITZ Architecture

Condition Codes:
 Modified
Privileged:
 No
Opcodes:
 Format C: 91 (hex 5B)
Possible Exceptions:
 None

Floating-Point Square Root

fsqrt fregA, fregC

This instruction computes the square root of the value in register fregA and stores the result in fregC. Both registers are floating-point registers.

Condition Codes:
 affected
Privileged:
 No
Opcodes:
 Format C: 92 (hex 5C)
Possible Exceptions:
 None

Floating-Point Negation

fneg fregA, fregC

This instruction computes the negative of the value in register fregA and stores the result in fregC. Both registers are floating-point registers.

Condition Codes:
 Not affected
Privileged:
 No
Opcodes:
 Format C: 93 (hex 5D)
Possible Exceptions:
 None

Floating-Point Absolute Value

fabs fregA, fregC

This instruction computes the absolute value of the value in register fregA and stores the result in fregC. Both registers are floating-point registers.

The BLITZ Architecture

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format C: 94 (hex 5E)

Possible Exceptions:

None

Floating-Point Load

```
fload    [regA+regB],fregC
fload    [regA+data16],fregC
```

Synthetic instruction:	Assembled identically to:
=====	=====
fload [regA],fregC	fload [regA+r0],fregC

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. In either case, the address must be word aligned, or an alignment exception will occur.

A 64-bit (8 byte) value is moved from the memory location given by the address into floating-point register fregC. (Registers regA and regB are integer registers, while fregC is a floating-point register.)

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 120 (hex 78)

Format E: 150 (hex 96)

Possible Exceptions:

Address Exception

Page Invalid

Alignment

Floating-Point Store

```
fstore   fregC,[regA+regB]
fstore   fregC,[regA+data16]
```

Synthetic instruction:	Assembled identically to:
=====	=====
fstore fregC,[regA]	fstore fregC,[regA+r0]

In the first form, the contents of regA and regB are added together to give an address. In the second form, a 16-bit immediate value is sign extended and added to the contents of regA to give an address. In either case, the address must be word aligned, or an alignment exception will occur.

The BLITZ Architecture

A 64-bit (8 byte) value is moved from register fregC to the memory location given by the address. A Page Read-only exception will occur if the Page Table is enabled and the page containing the address is marked read-only (Registers regA and regB are integer registers, while fregC is a floating-point register.)

Condition Codes:

Not affected

Privileged:

No

Opcodes:

Format D: 121 (hex 79)

Format E: 151 (hex 97)

Possible Exceptions:

Address Exception

Page Invalid

Alignment