BLITZ
Misc. Technical Notes

Harry Porter
Computer Science Department
Portland State University

BLITZ Floating-Point Architecture – Design Tradeoffs

I considered including special instructions in the BLITZ instruction set to load a floating point register from a constant, in analogy with the "sethi" and "setlo" instructions, which load an integer register. For example, to load an integer register, we use code like this:

```
sethi    0xXXXX,r3
setlo    0xYYYY,r3
```

The idea was to have four instructions, each of which would load 2 bytes into a floating-point register. Code to load a floating-point register would then look like this:

```
fset1    0xWWWW,f3
fset2    0xXXXX,f3
fset3    0xYYYY,f3
fset4    0xZZZZ,f3
```

I ruled such instructions out. Instead, you must use code like this:

```
sethi    0xXXXX,r1
setlo    0xYYYY,r1
fload    [r1],f3
...
.double  123.456
```

The code involving fset1, fset2, fset3, and fset4 is shorter (16 bytes), than the code sequence involving fload (20 bytes). However, I decided to avoid introducing the fset instructions since they complicated the instruction set. The "fload" instruction and the ".double" pseudo-op were necessary anyway, so this solution was simpler.

The most general design philosophy of the BLITZ architecture is that memory is cheap and execution speed is irrelevant, whereas simplicity is of paramount importance.

Several options were also considered with regard to the floating-point compare instruction ("fcmp") and the conditional branching instructions.

One option (which was not chosen) was to introduce several new condition code bits in the status register, to reflect the outcome of the "fcmp" instruction. This would require a separate set of branch instructions to test these new bits. Thus, we would have both a "fbe" and a "be" instruction to branch "if equal." This is obviously more complex and was ruled out.

Another option (also ruled out), was to use the same bits in the Status Register for both integer and floating-point comparisons, but to have a separate set of branch instructions, i.e., to have both "fbe" and "be".

Double-Precision Floating Point Values

Floating-point numbers are represented using the IEEE "Standard for Binary Floating Point Arithmetic" (ANSI / IEEE Std 754-1985). BLITZ supports only double-precision numbers, not single- or quad-precision numbers. We make no claim that the IEEE standard is supported correctly or completely; much of the implementation is simply inherited from the underlying "C" language implementation on which BLITZ is built.

A "double precision" floating-point number is represented with two words (8 bytes):

```
  byte 1      byte 2      byte 3      byte 4
==== ====   ==== ====   ==== ====   ==== ====
SEEE EEEE   EEEE XXXX   XXXX XXXX   XXXX XXXX

  byte 5      byte 6      byte 7      byte 8
==== ====   ==== ====   ==== ====   ==== ====
XXXX XXXX   XXXX XXXX   XXXX XXXX   XXXX XXXX

Where
    S            =  1-bit sign bit
    EEEE...EEEE  =  11-bit exponent field
    XXXX...XXXX  =  52-bit fraction field
```

The sign bit is 0=positive, 1=negative.

The fraction field is 52 bits. With decimal and binary numbers, leading zeros are always insignificant and are often omitted. With a decimal number, the leading non-zero digit can be anything between 1 and 9; with binary numbers, the leading non-zero bit will always be a 1. Therefore, with binary numbers, the leading 1 need not be represented; it may be implicit. In double-precision floating-point numbers, the 52-bits of the fraction field give 53 bits of accuracy. The fractional part (call it F) is thus:

```
F = 1.XXXX...XXXX
```

F is then raised to some power of 2, as given by the exponent field.

The exponent is 11 bits and can therefore be interpreted as an unsigned integer between 0 and 2047. If the exponent field is between 1 and 2046, then a normal number is being represented; if the exponent field is 0 or 2048, then it is a special case as discussed below.

When the exponent field is between 1 and 2046, you should subtract 1023 to obtain the actual exponent. That is, after subtracting 1023, you get a number which we can call M.

To get the number being represented, take F and multiply it by 2, raised to the power M. Then, adjust the sign, according to the S bit.

The range of numbers representable with double precision floating-point is:

```
Smallest Number:  2.2250738585072014E-308
Largest Number:   1.7976931348623157E+308
Precision:  about 17 decimal digits of accuracy
```

If the exponent is 2047 (i.e., all 1's), it signals a special case. Examples with exponent equal all ones are:

```
0x7FF00000 00000000 = POSITIVE-INFINITY
0xFFF00000 00000000 = NEGATIVE-INFINITY
0xFFFFFFFF FFFFFFFF = NaN (i.e., "Not-a-Number")
```

Positive and negative infinity can result from division by zero. Not-a-number indicates that an error has occurred, such as the square root of a negative number. The operations (add, multiply, etc.) are defined on these special case values in fairly logical ways. Any operation on a not-a-number value will yield a not-a-number result.

If the exponent is all zeros, it signals a "subnormal" number. These are small numbers, close to zero. They are represented slightly differently than shown in the formula above. These numbers also have a reduced precision. In particular, the implicit leading 1-bit assumed for normal numbers is no longer assumed. For subnormal numbers, we have

```
F = 0.XXXX...XXXX
```

Exceptions During "fload" and "fstore" / Page Faults and Race Conditions

The BLITZ architecture document says that if a page-readonly or page-invalid exception will occur during an instruction, then the instruction will have absolutely no effect and the exception will be processed as if the instruction had no been attempted.

This is not strictly true.

Consider an instruction (such as load) which reads a word from memory. Assume that the instruction fetch causes no problems, but that an exception occurs during the reading of the target word. For example, assume the target word causes a page-invalid exception. The instruction will be cancelled and the registers will be unchanged. The PC will not have been advanced, so it will appear that the instruction has not been attempted. However, one change to the system state may occur.

Recall that whenever a page is touched (i.e., read from) then its page table entry will have its "referenced" bit set. In this example, the CPU will set the referenced bit to 1, indicating that the page was used. Normally, during an instruction flow, the previous few instruction will be on the same page as the problematic instruction, so the referenced bit will already have been set. In such case, there would be no change. But it is possible that the problematic instruction lies in a

page that has not been previously referenced. (Perhaps the flow of control has just crossed a page boundary; this will happen on average every 2048 instructions, so it is a fairly common occurrence.)

There could be a subtle race problem here, if the operating system relies on the referenced bit. Perhaps the OS logic goes something like this, "Try to begin instruction execution. After an exception, bring in the necessary pages and re-start instruction execution. If memory frames are in short supply, then it is ok to page this process's pages out; we'll just get the same fault again later. However, make sure that we are making positive progress on each process. Make sure we executed at least one instruction. If we have not executed any instructions since the last time slice, then do not page this process's pages out. Instead, take frames from another process until this process has executed at least one instruction. Check the referenced bit to see if this process has made progress."

Obviously, the problem is in the last sentence, "Check the referenced bit…" This is an unreliable way to determine if a process has made progress. However, checking the PC is not reliable either, since it may happen to be unchanged, due to a looping process.

Another place that we can have page-table problems is with the "fload" instruction. This instruction reads two words from memory, in addition to the instruction fetch. It may be that the first word is fetched okay, but an exception occurred on the second word. This could occur if the doubleword being fetched happens to straddle a page boundary. As before, the "referenced" bit will be set for the frame containing the first word of the doubleword. The exception will then occur for the second word of the doubleword. Note that the floating-point register will be completely unaffected; i.e., it will not be "half" loaded with just the first word.

A similar problem occurs with storing into memory, with the "fstore." This instruction will store two words, and will mark the page table entries for these two words "dirty." Usually, both words of the doubleword will be within the same page so that the entry will either be marked and the operation completed or will be unmarked and an exception will occur. However, if the doubleword happens to straddle a page boundary and there is a page-invalid or page-readonly fault for the second page, the entry for the first page will be marked dirty, even though no word has been written. In other words, the page table entries are both updated before any writing to memory occurs.

Note that fstore does not do an atomic store. In other words, memory is not locked between the write of the first half of the doubleword and the write of the second half of the double word. In a multi-processor implementation, it is possible that another process doing a write to the same doubleword will overlap and the value stored will be half of one value and half of the other value, and thus meaningless. It is the compiler's responsibility to protect all fstore instructions with some sort of concurrency control if there is any possibility of concurrent access by multiple processes.

This would be a truly subtle, obscure, and hard-to-trace bug. It would result in no more than an apparent loss of less-significant bits. A value would appear to be approximately correct, but the final 32 bits would be incorrect, resulting in nothing more than a loss of accuracy.

Note that care must be taken in the operating system whenever doubleword values are stored and there is a possibility of concurrency.

Note that we may get into somewhat of a race condition in the OS. The fload and fstore instructions may require as many as 3 pages to be in memory at once: (1) the page containing the instruction, (2) the page containing the first half of the doubleword, and (3) the page containing the second half of the doubleword.

Overflows in Expression Evaluation During Assembly and Linking

Consider the following instruction:

```
sub     0x12345678,r3
```

The assembler can deduce that this value will not fit into 16 bits and will issue a warning. The assembler will use the least significant 16 bits, and will assemble the program as if the following had been coded:

```
sub     0x00005678,r3
```

For instructions requiring a 16-bit sign-extended literal value, the assembler will ensure that when sign-extension from 16 to 32 bits occurs, the value will be unchanged. The assembler will issue a warning (not a fatal error) whenever the value is outside of the range

```
0xffff8000 through 0x00007fff (inclusive)
```

In decimal, this range is

```
-32768 through 32767 (inclusive)
```

During expression evaluation, overflow may occur. Consider this instruction:

```
sub      r3,0x80000004+0x80000005,r6
```

In decimal, these numbers are:

```
-2147483644 and -2147483643
```

so this instruction is equivalent to

```
sub      r3,-2147483644 + -2147483643,r6
```

All computation is performed 32-bit arithmetic and any overflow is ignored. The result of the addition in decimal is:

```
-4294967287
```

This number cannot be represented in 32-bit two's complement. In hex, the result of the addition is

```
...FFFF00000009
```

When truncated to 32 bits, the value becomes

```
0x00000009
```

which is incorrect. Since this new value can be represented with only 16 bits, no error or warning will be issued. It will be as if the programmer had coded the following:

```
sub      r3,0x0009,r6
```

or

```
sub      r3,9,r6
```

Next, consider the following instruction, where "myExternalSymbol" is defined in another file:

```
.import  myExternalSymbol
sub      r3,myExternalSymbol,r6
```

The actual value cannot be known by the assembler, so it is impossible at assembly time to determine whether it will fit into 16 bits or not. When the linker determines the actual value, the linker will issue a warning if the value is not in the range:

```
0xffff8000 through 0x00007fff (inclusive)
```

This is a warning and not a fatal error. The linker will simply use the least significant 16 bits and proceed with that value. If this warning is ignored, the program will almost certainly malfunction.

There may be expressions in which one value is known by the assembler and the other is not. For example:

```
.import  myExternalSymbol
sub      r3,myExternalSymbol + -2147483643,r6
```

Assume that the actual value of "myExternalSymbol", given in some other file, is

```
                  .export  myExternalSymbol
myExternalSymbol  =        -2147483644
```

The linker will perform the addition and the result, which will overflow 32 bits, will be truncated to 32 bits. No error or warning will be issued, since the truncated value can be represented in 16 bits.

Now consider this example:

```
.import  myExternalSymbol2
sub      r3,myExternalSymbol2 + 0x11110000,r6
```

The value 0x11110000 exceeds the 16 bit limit but the assembler will not issue and error or warning since it cannot determine the final value of the expression.

The linker will perform the addition. As before, the linker will use 32 bit arithmetic (ignoring overflow) and will issue a warning if and only if the resulting 32 bit value cannot be represented in 16 bits.