The MIPS R4000, part 5: Memory access (aligned)

devblogs.microsoft.com/oldnewthing/20180406-00

April 6, 2018

Raymond Chen

The MIPS R4000 has one addressing mode: Register indirect with displacement.

```
LW      rd, disp16(rs)  ; rd = *( int32_t*)(rs + disp16)
LH      rd, disp16(rs)  ; rd = *( int16_t*)(rs + disp16)
LHU     rd, disp16(rs)  ; rd = *(uint16_t*)(rs + disp16)
LB      rd, disp16(rs)  ; rd = *(  int8_t*)(rs + disp16)
LBU     rd, disp16(rs)  ; rd = *( uint8_t*)(rs + disp16)
```

The load instructions load an aligned word, halfword, or byte from the address specified by adding the 16-bit signed displacement to the source register (known as the "base register").[1] By convention, the displacement can be omitted, in which case it is taken to be zero.

The plain versions of these instructions sign-extend to a 32-bit value; the `U` versions zero-extend.

There are corresponding aligned store instructions.

```
SW      rs, disp16(rd)  ; *( int32_t*)(rd + disp16) = (int32_t)rs
SH      rs, disp16(rd)  ; *( int16_t*)(rd + disp16) = (int16_t)rs
SB      rs, disp16(rd)  ; *(  int8_t*)(rd + disp16) = ( int8_t)rs
```

In all cases, if the effective address turns out not to be suitably aligned, an alignment fault occurs. Windows NT handles the alignment fault by loading the value using the unaligned memory access instructions (which we'll see next time), and then resuming execution. The overhead of the emulation swamps the cost of having done it correctly in the first place, so if you know that the address may be unaligned, then you are far better off using the unaligned memory access instructions instead of having the kernel fix it up for you.

The assembler emulates absolute addressing with the help of the *at* assembler temporary register. For example, the pseudo-instruction

```
LW      rd, global_variable
```

loads an aligned word from a global variable.

Let *A* be the address of the global variable, and let

- `YYYY = (int16_t)(A & 0xFFFF)` and
- `XXXX = (A - YYYY) >> 16`

Then the assembler generates the following two instructions:

```
LUI     at, XXXX
LW      rd, YYYY(at)
```

Note that if the bottom 16 bits of the address are greater than `0x8000`, then that results in a negative value for `YYYY`, and `XXXX` will be one greater than the upper 16 bits of the address.

Another pseudo-instruction is

```
LW      rd, imm32(rs)
```

You may want to do this if indexing a global array. A straightforward implementation of the pseudo-instruction would be

```
LUI     at, XXXX        ; load high part
ADDIU   at, at, YYYY    ; add in the low part
ADDU    at, at, rs      ; add in the byte offset
LW      rd, (at)        ; load the word
```

but this can be shortened by an instruction by merging the fixed offset `YYYY` into the displacement of the effective address calculation in the `LW`. The result is

```
LUI     at, XXXX
ADDU    at, at, rs
LW      rd, YYYY(at)
```

While the assembler emulation is convenient, it may not be the most efficient. If you are accessing the global variable more than once, or if you are accessing more than one variable within the same 64KB region, you can share the `LUI` instruction among them.

For example, suppose `global1` and `global2` reside in the same 64KB block of memory.

```
; lazy version of global2 = global1 + 1
LW      r1, global1
ADDIU   r1, r1, 1
SW      r1, global2
```

This expands to

```
LUI     at, XXXX
LW      r1, YYYY(at)
ADDIU   r1, r1, 1
LUI     at, XXXX
SW      r1, ZZZZ(at)
```

You can factor out the `XXXX` into a register that you reuse for the entire section of code.

```
; sneakier version of global2 = global1 + 1
LUI     r2, XXXX
LW      r1, YYYY(r2)
ADDIU   r1, r1, 1
SW      r1, ZZZZ(r2)
; can keep using r2 to access other variables in the block
```

In theory, you could even store constants in your data segment, but since loading a 32-bit constant takes only two instructions at most, you probably won't bother.

Next time, we'll look at unaligned access.

[1] In earlier versions of the MIPS architecture, there was a *load delay slot*: The value retrieved by a load instruction was not available until two instructions later.

We saw last time that the MIPS architecture supports forwarding of arithmetic computations. Why can't it forward memory access?

The memory stage comes after the execute stage. This means that the result of a memory load in the memory stage cannot be forwarded into the execute stage of the next instruction, because the memory stage of the first instructions takes place at the same time as the execute stage of the second instruction. The earliest the result of the load can be consumed is two instructions later.

That means that in the sequence

```
LW      r1, (r2)        ; load word from r2 into r1
ADDIU   r3, r1, 1       ; r3 = r1 + 1
```

the `ADDIU` instruction operated on the *old* value of *r1*,[2] not the value that was loaded from memory. If you want to add 1 to the value loaded from memory, you need to insert some other instruction in the load delay slot:

```
LW      r1, (r2)        ; load word from r2 into r1
NOP                     ; load delay slot
ADDIU   r3, r1, 1       ; r3 = r1 + 1
```

The MIPS III architecture removed the load delay slot. On the R4000, if you try to access the value of a register immediately after loading it, the processor stalls until the value becomes ready. Sure, the stall is bad, but it's better than running ahead with the wrong value!

[2] This is true only if no hardware interrupt occurred. If an interrupt occurred, then the load would complete during the kernel transition, and then when the kernel resumed execution, the `ADDIU` would operate on the loaded value after all. Therefore, the value of the

destination register of a load instruction should be treated as garbage until the load delay clears.

Raymond Chen

Follow