The MIPS R4000, part 2: 32-bit integer calculations

devblogs.microsoft.com/oldnewthing/20180403-00

April 3, 2018

Raymond Chen

The MIPS R4000 has the usual collection of arithmetic operations, but the mnemonics are confusingly-named. The general notation for arithmetic operations is

```
OP      destination, source1, source2
```

with the destination register on the left and the source register or registers on the right.

Okay, here goes. We start with addition and subtraction.

```
ADD     rd, rs, rt      ; rd = rs + rt, trap on overflow
ADDU    rd, rs, rt      ; rd = rs + rt, no trap on overflow
SUB     rd, rs, rt      ; rd = rs - rt, trap on overflow
SUBU    rd, rs, rt      ; rd = rs - rt, no trap on overflow
```

The `ADD` and `SUB` instructions perform addition and subtraction and raise a trap if a signed overflow occurs. The `ADDU` and `SUBU` instructions do the same thing, but without the overflow trap. The `U` suffix officially means “unsigned”, but this is confusing because the addition can be performed on both signed and unsigned values, thanks to twos complement. The real issue is whether an overflow trap is raised.

There are also versions of the addition instructions that accept a 16-bit signed immediate as a second addend:

```
ADDI    rd, rs, imm16   ; rd = rs + (int16_t)imm16, trap on overflow
ADDIU   rd, rs, imm16   ; rd = rs + (int16_t)imm16, no trap on overflow
```

Note that the `U` is double-confusing here, because even though the `U` officially stands for “unsigned”, the immediate value is treated as signed, and the addition is suitable for both signed and unsigned values.

There are no corresponding `SUBI` or `SUBIU` instructions, but they can be synthesized:

```
ADDI   rd, rs, -imm16   ; SUBI   rd, rs, imm16
ADDIU  rd, rs, -imm16   ; SUBIU  rd, rs, imm16
```

(Of course, this doesn't work if the value you want to subtract is −32768, but hey, it mostly works.)

The next group of instructions is the bitwise operations. These never trap.[1]

```
AND     rd, rs, rt      ; rd = rs & rt
ANDI    rd, rs, imm16   ; rd = rs & (uint16_t)imm16
OR      rd, rs, rt      ; rd = rs | rt
ORI     rd, rs, imm16   ; rd = rs | (uint16_t)imm16
XOR     rd, rs, rt      ; rd = rs ^ rt
XORI    rd, rs, imm16   ; rd = rs ^ (uint16_t)imm16
NOR     rd, rs, rt      ; rd = ~(rs | rt)
```

Note the inconsistency: The addition instructions treat the immediate as a signed 16-bit value (and sign-extend it to a 32-bit value), but the bitwise logical operations treat it as an unsigned 16-bit value (and zero-extend it to a 32-bit value). Stay alert!

The last group of instructions for today is the shift instructions. These also never trap.

```
SLL     rd, rs, imm5    ; rd = rs <<  imm5
SLLV    rd, rs, rt      ; rd = rs <<  (rt % 32)
SRL     rd, rs, imm5    ; rd = rs >>U imm5
SRLV    rd, rs, rt      ; rd = rs >>U (rt % 32)
SRA     rd, rs, imm5    ; rd = rs >>  imm5
SRAV    rd, rs, rt      ; rd = rs >>  (rt % 32)
```

The mnemonics stand for "shift left logical", "shift right logical" and "shift right arithmetic". The `V` suffix stands for "variable", and indicates that the shift amount comes from a register rather than an immediate.

Yup, that's another inconsistency. Following the pattern of the addition and bitwise logical groups, these instructions should have been named `SLL` for shifting by an amount specified by a register and `SLLI` for shifting by an amount specified by an immediate. Go figure.

There are no built-in sign-extension or zero-extension instructions. You can get zero-extension in one instruction by explicitly masking out the upper bytes:

```
; zero extend byte to word
ANDI    rd, rs, 0xFF    ; rd = ( uint8_t)rs

; zero extend halfword to word
ANDI    rd, rs, 0xFFFF  ; rd = (uint16_t)rs
```

Sign extension requires two instructions.

```
; sign extend byte to word
SLL     rd, rs, 24      ; rd = rs << 24
SRA     rd, rd, 24      ; rd = (int32_t)rd >> 24

; sign extend halfword to word
SLL     rd, rs, 16      ; rd = rs << 16
SRA     rd, rd, 16      ; rd = (int32_t)rd >> 16
```

And I'm going to mention these instructions here because I can't find a good place to put them:

```
SYSCALL imm20           ; system call
BREAK   imm20           ; breakpoint
```

Both instructions trap into the kernel. The system call instruction is intended to be used to make operation system calls; the breakpoint instruction is intended to be used for software breakpoints. Both instructions carry a 20-bit immediate payload that can be used for whatever purpose the operating system chooses.

Here are some more instructions you can synthesize from the official instructions:

```
SUB     rd, zero, rs    ; NEG     rd, rs
SUBU    rd, zero, rs    ; NEGU    rd, rs
ADDU    rd, zero, rs    ; MOVE    rd, rs
OR      rd, zero, rs    ; MOVE    rd, rs
NOR     rd, zero, rs    ; NOT     rd, rs
SLL     zero, zero, 0   ; NOP
SLL     zero, zero, 1   ; SSNOP
```

There are many possible ways of synthesizing a `MOVE` instruction, but in order to be able to unwind exceptions, Windows NT requires that register motion in the prologue or epilogue of a function must take one of the two forms given above.

Similarly, there are many ways of performing a `NOP`. Basically, any non-trapping 32-bit computation that targets the *zero* register is functionally a nop, but the two above are treated specially by the processor.

- `NOP` = `SLL zero, zero, 0` is special-cased by the processor as a nop that can be optimized out entirely. Use it when you need to pad out some code for space.
- `SSNOP` = `SLL zero, zero, 1` is special-cased by the processor as a nop that must be issued, and it will not be simultaneously issued with any other instruction. Use it when you need to pad out some code for time. (The `SS` stands for "super-scalar".)

The encoding of `SLL zero, zero, 0` happens to be `0x00000000`, which I'm sure is not a coincidence. I'm not convinced that it's a good idea, though. I would have chosen `0x00000000` to be the encoding of a breakpoint or invalid instruction.

Okay, those are the 32-bit computation instructions. Next time, we'll look at multiplication, division, and the temperamental *HI* and *LO* registers.

[1] Alas, there is no `NORI` instruction. You think I'm joking, but I'm not. Be patient.

Raymond Chen

Follow