The SuperH-3, part 2: Addressing modes

devblogs.microsoft.com/oldnewthing/20190806-17

August 6, 2019

Raymond Chen

The SH-3 supports a large number of addressing modes, which is somewhat unusual for a RISC processor.

When I write *operand size*, I mean 1 for byte access, 2 for word access, and 4 for longword access.

The mnemonic for many of the addressing modes is that `@x` uses *x* as the address, and `@(x, y)` first adds *x* and *y*, and then uses the sum to form the address.

Immediate: The value is a constant.

```
MOV     #imm, Rn    ; Copy sign-extended byte constant to Rn
```

There is obviously no "store" version of this instruction.

Register direct: The value is taken directly from or stored directly to a register.

```
MOV     Rm, Rn      ; Copy Rm to Rn
```

Register indirect: The value is read from or written to memory whose address is provided by a register.

```
MOV.B   Rm, @Rn     ; Store byte in Rm to address in Rn
MOV.W   Rm, @Rn     ; Store word in Rm to address in Rn
MOV.L   Rm, @Rn     ; Store longword in Rm to address in Rn

MOV.B   @Rm, Rn     ; Load and sign-extend byte from address in Rm to Rn
MOV.W   @Rm, Rn     ; Load and sign-extend word from address in Rm to Rn
MOV.L   @Rm, Rn     ; Load longword from address in Rm to Rn
```

Register indirect with post-increment: The value is read from memory whose address is provided by a register, and then the register is increased by the operand size.

```
MOV.B   @Rm+, Rn    ; Load and sign-extend byte from address in Rm to Rn,
                    ; then increment Rm by 1

MOV.W   @Rm+, Rn    ; Load and sign-extend word from address in Rm to Rn,
                    ; then increment Rm by 2

MOV.L   @Rm+, Rn    ; Load longword from address in Rm to Rn,
                    ; then increment Rm by 4
```

Post-increment is supported only by load instructions. You cannot post-increment a store.

This instruction is used primarily to pop values from the stack.

Register indirect with pre-decrement: The register is decreased by the operand size, and then the decremented value provides the address.

```
MOV.B   Rm, @-Rn    ; Decrement Rn by 1,
                    ; then store byte in Rm to address in Rn

MOV.W   Rm, @-Rn    ; Decrement Rn by 2,
                    ; then store word in Rm to address in Rn

MOV.L   Rm, @-Rn    ; Decrement Rn by 2,
                    ; then store longword in Rm to address in Rn
```

Pre-decrement is supported only by store instructions. You cannot pre-decrement a load.

This instruction is used primarily to push values to the stack.

Register indirect with displacement: A small unsigned constant is added to the register, and the result is the address to be accessed. The constant must be a multiple of the operand size, up to 15 times the operand size. In other words, for byte access, the offset is an integer from 0 to 15; for word access, the offset is an even integer from 0 to 30; and for longword access, the offset is a multiple of four from 0 to 60.

```
MOV.B   @(disp, Rm), r0 ; Load sign-extended byte from (disp + Rm) to r0
MOV.W   @(disp, Rm), r0 ; Load sign-extended word from (disp + Rm) to r0
MOV.L   @(disp, Rm), Rn ; Load longword from (disp + Rm) to Rn

MOV.B   r0, @(disp, Rn) ; Store byte in r0 to address (disp + Rn)
MOV.W   r0, @(disp, Rn) ; Store word in r0 to address (disp + Rn)
MOV.L   Rm, @(disp, Rn) ; Store longword in Rm to address (disp + Rn)
```

Note that if you are accessing a byte or word, then the value must be stored from or loaded into the *r0* register. If you are accessing a longword, then any register can be the target.

Indexed register indirect: The value in *r0* is added to the value of the other register, and the result is the address to be accessed. Note that the first register is always *r0*.

```
    MOV.B   @(r0, Rm), Rn   ; Load sign-extended byte from (r0 + Rm) to Rn
    MOV.W   @(r0, Rm), Rn   ; Load sign-extended word from (r0 + Rm) to Rn
    MOV.L   @(r0, Rm), Rn   ; Load longword from (r0 + Rm) to Rn

    MOV.B   Rm, @(r0, Rn)   ; Store byte in Rm to address (r0 + Rn)
    MOV.W   Rm, @(r0, Rn)   ; Store word in Rm to address (r0 + Rn)
    MOV.L   Rm, @(r0, Rn)   ; Store longword in Rm to address (r0 + Rn)
```

GBR indirect with displacement: A small unsigned constant is added to the *gbr* register, and the result is the address to be accessed. The constant must be a multiple of the operand size, up to 255 times the operand size. In practice, the operand size is 4 (longword), so the reach is 1KB.

```
    MOV.B   @(disp, GBR), r0 ; Load sign-extended byte from (disp + GBR) to r0
    MOV.W   @(disp, GBR), r0 ; Load sign-extended word from (disp + GBR) to r0
    MOV.L   @(disp, GBR), r0 ; Load longword from (disp + GBR) to r0

    MOV.B   r0, @(disp, GBR) ; Store byte in r0 to address (disp + GBR)
    MOV.W   r0, @(disp, GBR) ; Store word in r0 to address (disp + GBR)
    MOV.L   r0, @(disp, GBR) ; Store longword in r0 to address (disp + GBR)
```

The value must be stored from or loaded into the *r0* register.

PC-relative with displacement: A small unsigned constant is added to the *pc* register, and then 4 is added, and the result is the address to be accessed. The constant must be a multiple of the operand size, up to 255 times the operand size. In practice, the operand size is usually 4 (longword), in which case the reach is 1KB.

```
;   Note: No byte version
    MOV.W   @(disp, PC), Rn ; Load sign-extended word from (disp + PC + 4) to Rn
    MOV.L   @(disp, PC), Rn ; Load longword from ((disp + PC + 4) & ~3) to Rn
```

If the operand is a longword, then the bottom two bits of the result are forced to zero before using it to access memory. If this weren't done, then it wouldn't be possible to access 32-bit PC-relative data from instructions at addresses that are not exact multiples of 4!

There is no instruction for loading large constants into registers. Instead, you put the constants in the code segment and use a PC-relative load to load them into a register. Since the reach is only 1KB, you need to break up your functions into 1KB chunks in order to inject constants.

The disassembler is kind enough to perform the calculation of the effective address for you. It even reads the memory for you, if it can.

Why does the instruction add 4? That's an artifact of the pipelining. By the time the processor has gotten around to executing the instruction, the program counter has moved ahead two instructions. This also means that if you execute this instruction in a branch delay slot, you're

in for a nasty surprise, because it's going to use the branch destination as the value of PC! (To avoid the nasty surprise, the SH-4 made this instruction outright illegal in a branch delay slot.)

Although the SH-3 has a lot of addressing modes, none of them provides a significant reach; the furthest you can reach is 1KB. This is an unfortunate consequence of the 16-bit instruction size. There simply isn't room in the instruction to put a large displacement.

In practice, you spend a lot of time calculating offsets so you can use the indexed register indirect addressing mode. What makes it even worse is that the indexed register indirect addressing mode must use *r0* as a base register, which means that the *r0* register becomes a bottleneck because a lot of things need to pass through it.

Note also that there is no absolute addressing mode. The PC-relative addressing mode doesn't have a large reach, so you can't use it to access variables in your data segment. Accessing global variables is typically done in two steps: First, load the address of the global variable from a constants pool near your function, and then dereference that address to access the global variable itself.

Okay, enough with the memory addressing modes. Next time, we'll look at the flags register.

Raymond Chen

Follow