The SuperH-3, part 4: Basic arithmetic

devblogs.microsoft.com/oldnewthing/20190808-00

August 8, 2019

Raymond Chen

Okay, we're ready to do some arithmetic. Due to the limited instruction encoding space, there isn't room for any three-operand instructions.[1] All of the arithmetic instructions are two-operand, where the second source operand also acts as the destination.

```
ADD     Rm, Rn      ; Rn += Rm    , no effect on T
ADD     #imm, Rn    ; Rn += imm   , no effect on T
ADDC    Rm, Rn      ; Rn += Rm + T, T receives carry
ADDV    Rm, Rn      ; Rn += Rm    , T receives signed overflow
```

The `ADD` instructions add two values and put the result in the second register. You can add two registers together, or you can add a signed 8-bit immediate to the destination register.

The `ADDC` instruction treats the *T* flag as a carry flag: It is added to the sum, and it receives the carry of the result.

The `ADDV` instruction treats the *T* flag as an overflow flag: It reports whether a signed overflow occurred.

Okay, subtraction is going to look really similar now.

```
SUB     Rm, Rn      ; Rn -= Rm    , no effect on T
SUB     #imm, Rn    ; Rn -= imm   , no effect on T
SUBC    Rm, Rn      ; Rn -= Rm + T, T receives borrow
SUBV    Rm, Rn      ; Rn -= Rm    , T receives signed underflow
```

Basically the same as addition, except you're now subtracting. The SH-3 treats *T* as a borrow flag in the case of `SUBC`, whereas for `SUBV` it reports whether a signed underflow occurred.

Arithmetic negation is up next.

```
NEG     Rm, Rn      ; Rn = -Rm    , no effect on T
NEGC    Rm, Rn      ; Rn = -Rm - T, T receives borrow
```

There is no `NEGV`, but overflow occurs only if the value is `0x80000000`, so I guess you could test for that value specifically.

There is a special instruction for for decrementing a register:

```
DT      Rn           ; Rn = Rn - 1, T  = (Rn == 0)
```

The *decrement and test* instruction decrements a register and compares the result against zero. This is presumably for counted loops.

Next come the comparison instructions.

```
CMP/EQ #imm, r0     ; T = (r0 == signed 8-bit immediate)
CMP/EQ Rm, Rn       ; T = (Rn == Rm)
CMP/HS Rm, Rn       ; T = (Rn ≥ Rm), unsigned comparison
CMP/GE Rm, Rn       ; T = (Rn ≥ Rm),   signed comparison
CMP/HI Rm, Rn       ; T = (Rn > Rm), unsigned comparison
CMP/GT Rm, Rn       ; T = (Rn > Rm),   signed comparison
CMP/PZ Rn           ; T = (Rn ≥ 0),    signed comparison
CMP/PL Rn           ; T = (Rn > 0),    signed comparison
CMP/STR Rm, Rn      ; T = 1 iff any corresponding bytes are equal
```

These instructions set the *T* flag according to a particular comparison. Note that the comparison is backward! For example, `CMP/GE r1, r2` does not check whether *r1* ≥ *r2*; rather, it checks whether *r2* ≥ *r1*. *This takes a lot of getting used to.*

You have the special ability to compare *r0* for equality with a signed 8-bit immediate. Otherwise, you can compare two registers against each other, or a register against zero.

The special `CMP/STR` compares two registers to determine whether any of the four component bytes are equal. It's clear from the mnemonic that the intended purpose is to search for a null terminator in a string. You set *Rn* to zero and then do a `CMP/STR` against every longword in the string until it says, "Hey, I found a zero byte!" and then you can study that longword to see where the zero byte is.

The processor documentation doesn't explain why they chose the names for the mnemonics, but I can guess.

Condition	Meaning
`EQ`	equal
`HS`	high or same
`GE`	greater or equal
`HI`	high
`GT`	greater than
`PZ`	plus or zero

`PL`	plus
`STR`	string

It took me a while to come up with a plausible explanation for `HS`.

Exercise 1: Synthesize the `SETT` and `CLRT` instructions.

Exercise 2: Perform the opposite of the `MOVT` instruction: Set the *T* register to 0 if a register is zero, or 1 if the register is nonzero.

The last arithmetic instructions are the extension instructions.

```
EXTS.B Rm, Rn       ; sign extend byte in Rm to Rn
EXTS.W Rm, Rn       ; sign extend word in Rm to Rn
EXTU.B Rm, Rn       ; zero extend byte in Rm to Rn
EXTU.W Rm, Rn       ; zero extend word in Rm to Rn
```

That's it for the basic arithmetic instructions. We'll start looking at the more complicated arithmetic instructions next time, starting with multiplication.

[1] Well, okay, you can have three-operand instructions if some of them are hard-coded! But that's not what I mean. I mean three-operand instructions where the programmer can choose all three of the operands.

Raymond Chen

Follow