The SuperH-3, part 1: Introduction

Raymond Chen

August 5th, 2019

Windows CE supported the Hitachi SuperH-3 and SuperH-4 processors. These were commonly abbreviated SH-3 and SH-4, or just SH3 and SH4, and the architecture series was known as SHx.

I'll cover the SH-3 processor in this series, with some nods to the SH-4 as they arise. But the only binaries I have available for reverse-engineering are SH-3 binaries, so that's where my focus will be.

The SH-3 is the next step in the processor series that started with the SH-1 and SH-2. It was succeeded by the SH-4 as well as the offshoots SH-3e and SH-3-DSP. The SH-4 is probably most famous for being the processor behind the Sega Dreamcast.

As with all the processor retrospective series, I'm going to focus on how Windows CE used the processor in user mode, with particular focus on the instructions you will see in compiled code.

The SH-3 is a 32-bit RISC-style (load/store) processor with fixed-length 16-bit instructions. The small instruction size permits higher code density than its contemporaries, with Hitachi claiming a code size reduction of a third to a half compared to processors with 32-bit instructions. The design was apparently so successful that ARM licensed it for their Thumb instruction set.

The SH-3 can operate in either big-endian or little-endian mode. Windows CE uses it in little-endian mode.

The SH-3 has sixteen general-purpose integer registers, each 32 bits wide, and formally named *r0* through *r15*. They are conventionally used as follows:

Register	Meaning	Preserved?
r0	return value	No
r1		No
r2		No
r3		No
r4	argument 1	No
r5	argument 2	No
r6	argument 3	No
r7	argument 4	No
r8		Yes
r9		Yes
r10		Yes
r11		Yes
r12		Yes
r13		Yes
r14, aka *fp*	frame pointer	Yes
r15, aka *sp*	stack pointer	Yes

We'll learn more about the conventions when we study calling conventions.

There are actually two sets (banks) of the first eight registers (*r0* through *r7*). User-mode code uses only bank 0, but kernel mode can choose whether it uses bank 0 or bank 1. (And when it's using one bank, kernel mode has special instructions available to access the registers from the other bank.)

The SH-3 does not support floating point operations, but the SH-4 does. There are sixteen single-precision floating point registers which are architecturally named *fpr0* through *fpr15*, but which the Microsoft assembler calls *fr0* through *fr15*. They can be paired up to produce eight double-precision floating point registers:

Double-precision register	Single-precision register pair	
dr0	*fr0*	*fr1*
dr2	*fr2*	*fr3*
dr4	*fr4*	*fr5*
dr6	*fr6*	*fr7*
dr8	*fr8*	*fr9*
dr10	*fr10*	*fr11*
dr12	*fr12*	*fr13*
dr14	*fr14*	*fr15*

If you try to perform a floating point operation on an SH-3, it will trap, and the kernel will emulate the instruction. As a result, floating point on an SH-3 is very slow.

Windows NT requires that the stack be kept on a 4-byte boundary. I did not observe any red zone.

There are also some special registers:

Register	Meaning	Preserved?	Notes
pc	program counter	duh	instruction pointer, must be even
gbr	global base register	No	bonus pointer register
sr	status register	No	Flags
mach	multiply and accumulate high	No	For multiply-add operations
macl	multiply and accumulate low	No	For multiply-add operations
pr	procedure register	Yes	Return address

Some calling conventions for the SH-3 say that *mach* and *macl* are preserved, or that *gbr* is reserved, but in Windows CE, they are all scratch.

We'll take a closer look at the status register later.

The architectural names for data sizes are as follows:

- **byte**: 8-bit value
- **word**: 16-bit value
- **longword**: 32-bit value
- **quadword**: 64-bit value

Unaligned memory accesses will fault. We'll look more closely at unaligned memory access later.

The SH-3 has branch delay slots. Ugh, branch delay slots. What's worse is that some branch instructions have branch delay slots and some don't. Yikes! We'll discuss this in more detail when we get to control transfer.

Instructions on the SH-3 are generally written with source on the left and destination on the right. For example,

```
MOV     r1, r2      ; move r1 to r2
```

The SH-3 can potentially retire two instructions per cycle, although internal resource conflicts may prevent that. For example, an `ADD` can execute in parallel with a comparison instruction, but it cannot execute in parallel with a `SUB` instruction. In the case of a resource conflict, only one instruction is retired during that cycle.

After an instruction that modifies flags, the new flags are not available for a cycle, and after a load instruction, the result is not available for two cycles. There are other pipeline hazards, but those are the ones you are likely to encounter. If you try to use the results of a prior instruction too soon, the processor will stall. (Don't forget that the SH-3 is dual-issue, so two cycles can mean up to four instructions.)

Okay, that's enough background. We'll dig in next time by looking at addressing modes.

Raymond Chen

Follow

The SuperH-3, part 2: Addressing modes

Raymond Chen

August 6th, 2019

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

The SuperH-3, part 3: Status flags and miscellaneous instructions

Raymond Chen

August 7th, 2019

Only four of the bits in the status register are available to user-mode:

Bit	Meaning	Notes
M	Modulus	Used by division instructions
Q	Quotient	Used by division instructions
S	Saturate	Used by multiply-add instructions
T	Test	Multi-purpose flag

(There was no official meaning for the names of the registers, so I made up mnemonics for them.)

Aside from the flags used by special-purpose instructions (multiplication and division), there is basically only one flag: *T*. Each instructions decides how it wishes to consume and produce the *T* flag.

```
CLRT        ; T = 0
SETT        ; T = 1

CLRS        ; S = 0
SETS        ; S = 1
```

There are four instructions which directly set or clear two of the bits in the status register. We'll learn more about the *M* and *Q* registers when we study integer division.

```
MOVT    Rn  ; Rn = T (0 or 1)
```

There is also a special instruction to copy the *T* flag into a register. There is no converse instruction, but we'll see later how we could try to synthesize one.

Windows CE requires that the *S* flag be clear at function entry and exit.

Since there wasn't much to be said about flags, I'll use the rest of my time to cover various miscellaneous instructions.

```
MOVA @(disp, PC), r0    ; r0 = PC + disp
```

The *move address* instruction calculates the effective address of `@(disp, PC)` and stores it into *r0*. The displacement can be a multiple of 4 up to 255 × 4 = 1020.

```
SWAP.B Rm, Rn           ; Rn = Rm with bottom two bytes swapped
SWAP.W Rm, Rn           ; Rn = Rm with top and bottom words swapped
XTRCT  Rm, Rn           ; Rn = (Rn << 16) | (Rm >> 16)
```

These instructions are for byte swapping or extracting the middle 32 bits of a 64-bit value.

```
PREF  @Rn               ; prefetch memory at Rn
```

The prefetch instruction has no effect if the memory at *Rn* is inaccessible.

```
TRAPA #imm              ; trap to kernel mode
```

The TRAPA instruction traps to kernel mode. It carries an 8-bit unsigned immediate payload which kernel mode can use to signify anything it wishes.

```
NOP                     ; do nothing
```

Fortunately, the instruction `0000` is invalid, rather than being a nop.

```
STC     GBR, Rn         ; Rn = GBR
LDC     Rn, GBR         ; GBR = Rn
STC     PR, Rn          ; Rn = PR
LDC     Rn, PR          ; PR = Rn
```

These instructions let you move data into and out of the special registers *gbr* and *pr*. We saw *gbr* when we learned about addressing modes. We'll learn about *pr* when we get to control transfer.

Well, that wasn't very exciting yet. Let's start doing math. Next time.

Raymond Chen

Follow

The SuperH-3, part 4: Basic arithmetic

Raymond Chen

August 8th, 2019

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

The SuperH-3, part 5: Multiplication

Raymond Chen

August 9th, 2019

Last time, we looked at simple addition and subtraction. Now let's look at multiplication.

Multiplication operations report their results in a pair of 32-bit registers called called *MACH* and *MACL*, which collectively form a 64-bit virtual register known as *MAC* (multiply and accumulate).

We start with the simple multiplication operations.

```
MUL.L   Rm, Rn  ; MACL =           Rm *           Rn, no effect on MACH
MULS.W  Rm, Rn  ; MACL = ( int16_t)Rm * ( int16_t)Rn, no effect on MACH
MULU.W  Rm, Rn  ; MACL = (uint16_t)Rm * (uint16_t)Rn, no effect on MACH
```

The `.W` operations treat the two source operands as 16-bit values, either signed or unsigned, and store the 32-bit result into *MACL*. The `MUL.L` treats the source operands as full 32-bit values, and produces a 32-bit result in *MACL*. (It doesn't matter whether the sources are considered signed or unsigned because the lower 32 bits of the result are the same either way.)

The next instructions produce 64-bit results.

```
DMULS.L Rm, Rn      ; MAC = Rn * Rm,   signed 32x32→64 multiply
DMULU.L Rm, Rn      ; MAC = Rn * Rm, unsigned 32x32→64 multiply

MAC.L   @Rm+, @Rn+  ; MAC += @Rm++ * @Rn++, signed 32x32→64 multiply
MAC.W   @Rm+, @Rn+  ; MAC += @Rm++ * @Rn++, signed 16x16→64 multiply
```

The `MAC.x` instructions are interesting in that they access two memory locations in one instruction. Both *Rm* and *Rn* are treated as addresses, 16-bit or 32-bit values are loaded from those addresses, the loaded values are treated as signed integers, multiplied together, and the result added to the 64-bit accumulator register *MAC*, and finally the registers are incremented by the operand size. The design of the instruction is evidently for performing a dot product of two vectors.

There's an additional wrinkle to the `MAC.x` instructions: If you set the *S* flag, then the operations use saturating addition rather than wraparound addition. For `MAC.L`, the saturation is as a 48-bit value, and the value is sign-extended to a 64-bit value in *MAC*. For `MAC.W`, the saturation is as a 32-bit value, and the bottom bit of *MACH* is set to 1 if an overflow occurred.

In practice, of these multiplication instructions, you will likely see only `MUL.L` in compiler-generated code.

Oh wait, how do you get the answers out of the *MAC* registers? Yeah, there are instructions for that too.

```
CLRMAC              ; MAC = 0

LDS     Rm, MACH    ; MACH = Rm
LDS     Rm, MACL    ; MACL = Rm
LDS.L   @Rm+, MACH  ; MACH = @Rm+
LDS.L   @Rm+, MACL  ; MACL = @Rm+

STS     MACH, Rn    ; Rn = MACH
STS     MACL, Rn    ; Rn = MACL
STS.L   MACH, @-Rn  ; @-Rn = MACH
STS.L   MACL, @-Rn  ; @-Rn = MACL
```

The `CLRMAC` instruction sets *MAC* to zero, which is a good starting point for subsequent `MAC.x` instructions.

The `LDS` instructions move values into the *MAC* registers. You can move a value directly from a register or load it (with post-increment) from memory. Conversely, the `STS` instructions move values out of the *MAC* registers, either into a general-purpose register or into memory.

Next up is integer division, which is going to be interesting.

Raymond Chen

Follow

The SuperH-3, part 6: Division

Raymond Chen

August 12th, 2019

The SH-3 does not have a simple “divide two integers please” instruction. Rather, it has a collection of instructions that let you build a division operation yourself.

```
DIV0U               ; prepare for unsigned division
DIV0S   Rm, Rn      ; prepare for signed division Rn ÷ Rm
DIV1    Rm, Rn      ; generate 1 bit of the quotient Rn ÷ Rm
```

To begin an integer division operation, you execute either a `DIV0U` or `DIV0S` instruction, depending on whether you want a signed or unsigned division.

You then perform a number of `DIV1` instructions equal to the number of bits of quotient you need, mixed in with other instructions are outlined in the programmer’s manual. After running the desired number of iterations, the result is in either the *Rn* register or in the register you accumulated the results into, depending on the specific algorithm you used.

These instructions do not attempt to handle division by zero or division overflow. They will simply generate nonsense results. If preventing division by zero or overflow is important to you, you will have to check for them yourself explicitly.

I’m not going to go into the fine details of how these instructions operate. They use *Rm* and *Rn* to record the state of the division, with three additional bits of state recorded in the *M*, *Q*, and *T* flags.

It’s basically magic.

In practice, you won’t see the compiler generate these instructions anyway. Instead, the compiler is going to do one of the following:

- If dividing by a constant power of 2, use a shift instruction.
- If dividing by a small constant, multiply by $2^{32} \div n$ and extract the high 32 bits of the result.
- Otherwise, call a helper function in the runtime library.

Phew, that was crazy.

Next time, we’ll return to the relative sanity of bitwise logical operations.

Bonus chatter: If you want to see how one compiler implements division on the SH-3 (and if you are okay with being exposed to GPL source code), you can take a look at how GCC implements division in its runtime library.

Raymond Chen

Follow

The SuperH-3, part 7: Bitwise logical operations

Raymond Chen

August 13th, 2019

The SH-3 has a rather basic collection of bitwise logical operations.

```
AND Rm, Rn          ; Rn &= Rm
AND #imm, r0        ; r0 &= unsigned 8-bit immediate

OR  Rm, Rn          ; Rn |= Rm
OR  #imm, r0        ; r0 |= unsigned 8-bit immediate

XOR Rm, Rn          ; Rn ^= Rm
XOR #imm, r0        ; r0 ^= unsigned 8-bit immediate

NOT Rm, Rn          ; Rn = ~Rm
```

Nothing fancy. No *nor* or *nand* or *andnot* or other goofy bitwise operations. Just plain vanilla stuff. Do note that the 8-bit immediate is unsigned here.

There is also an instruction for testing bits wthout modifying anything other than the *T* flag.

```
TST Rm, Rn              ; T = ((Rn & Rm) == 0)
TST #imm, r0            ; T = ((r0 & signed 8-bit immediate) == 0)
```

The *test* instruction performs a bitwise *and* and compares the result with zero. In this case, the 8-bit immediate is signed.

But wait, there's something goofy after all: Load/modify/store instructions!

```
AND.B #imm, @(r0, GBR)  ; @(r0 + gbr) &= 8-bit immediate
OR.B  #imm, @(r0, GBR)  ; @(r0 + gbr) |= 8-bit immediate
XOR.B #imm, @(r0, GBR)  ; @(r0 + gbr) ^= 8-bit immediate
TST.B #imm, @(r0, GBR)  ; T = ((@(r0 + gbr) & 8-bit immediate) == 0)
```

These `.B` versions of the bitwise logical operations operate on a byte in memory indexed by the *r0* and *gbr* registers. Okay, so `TST.B` is not a load/modify/store; it's just a load, but I included it in this group because he wants to be with his friends.

In practice, the Microsoft compiler does not generate these instructions.

Finally, we have this guy, the only truly atomic instruction in the SH-3 instruction set.

```
TAS.B @Rn               ; T = (@Rn == 0), @Rn |= 0x80
```

The *test-and-set* instruction reads a byte from memory, compares it against zero (setting *T* accordingly), and then sets the high bit and writes the result back out. This was clearly designed for building low-level synchronization primitives, but I'm not sure anybody actually uses it.

I say that it is the only truly atomic operation because it holds the data bus locked for the duration of its operation. The load/modify/store instructions we saw above do not lock the bus, so it's possible for a coprocessor to modify the memory out from under the SH-3.

That's it for the logical operations. Next up are the bit shifting operations.

Raymond Chen
Follow

The SuperH-3, part 8: Bit shifting

Raymond Chen

August 14th, 2019

The bit shifting operations are fairly straightforward.

```
; arithmetic (signed) shifts
SHAL Rn         ; Rn <<= 1, T = the bit shifted out
SHAR Rn         ; Rn >>= 1, T = the bit shifted out

; logical (unsigned) shifts
SHLL Rn         ; Rn <<= 1, T = the bit shifted out
SHLR Rn         ; Rn >>= 1, T = the bit shifted out
SHLL2 Rn        ; Rn <<= 2
SHLR2 Rn        ; Rn >>= 2
SHLL8 Rn        ; Rn <<= 8
SHLR8 Rn        ; Rn >>= 8
SHLL16 Rn       ; Rn <<= 16
SHLR16 Rn       ; Rn >>= 16
```

You cannot shift by arbitrary constant amounts. Only certain fixed values are permitted. If you want to shift left by, say, 9, you'll have to construct it from a `SHLL8` and a `SHLL`.

Note also that `SHAL` and `SHLL` are functionally equivalent. But they have different encodings, so the designers burned an opcode for a redundant operation.

There are no "large shift" options for right shifts. You can perform multiple one-bit shifts, or use a variable shift:

```
SHAD Rm, Rn     ; if Rm > 0: Rn <<= (31 & Rm)
                ; if Rm = 0: nop
                ; if Rm < 0: Rn >>= (31 & -Rm), signed

SHLD Rm, Rn     ; if Rm > 0: Rn <<= (31 & Rm)
                ; if Rm = 0: nop
                ; if Rm < 0: Rn >>= (31 & -Rm), unsigned
```

Note that these shift instructions shift both left *and* right, depending on the sign of the shift amount. If you want to shift right by an amount in a register, you therefore need to negate the value, and then shift left.

Finally, we have rotation.

```
ROTL Rn        ; rotate left, T contains carried-out bit
ROTR Rn        ; rotate right, T contains carried-out bit
ROTCL Rn       ; 33-bit rotate through T
ROTCR Rn       ; 33-bit rotate through T
```

The rotation instructions rotate either a 32-bit or 33-bit value by one position. For the 32-bit rotations, the bit that rotated off the end is copied to *T*. For the 33-bit rotations, the *T* flag acts as the 33rd bit.

We saw earlier that there is no `NEGV` instruction. To detect overflow from a negation, you just have to check for the value `0x80000000` directly. Here's the shortest sequence I could come up with:

```
; branch if Rn equals 0x80000000
rotl Rn        ; rotate left one bit
dt   Rn        ; decrement and test for zero
bt   underflow ; Y: underflow occurred
```

The result of the `DT` is zero if the previous value was 1, and the previous value was 1 if the original value was `0x80000000`.

This is a destructive operation, so do it in a scratch register. You should have one available, since it's the source register for the `NEGV` you were checking.

We'll look more at constants next time.

Raymond Chen

Follow

The SuperH-3, part 9: Constants

Raymond Chen

August 15th, 2019

Loading constants on the SH-3 is a bit of a pain. We saw that the `MOV` instruction supports an 8-bit signed immediate, but what if you need to load something outside that range?

The assembler allows you to write this:

```
MOV     #value, Rn      ; load constant into Rn
```

If the value fits in an 8-bit signed immediate, then it uses that. Otherwise, it chooses a PC-relative `MOV.W` or `MOV.L` depending on the size of the value, and it generates the constant into the code at a point it believes that the code is unreachable, such as two instructions after a `bra` or `rts`. If no such point can be found, the assembler raises an error. You can use the `.nopool` directive to prevent constants from being generated at a particular point, or `.pool` to force them to be generated.

If the compiler can generate the constant in two instructions, typically by combining an immediate with a shift, then the compiler will tend to prefer the two-instruction version instead of using a constants pool, especially if it can put the second half of the calculation into an otherwise-wasted branch delay slot. (Yes, we haven't learned about branch delay slots yet. Be patient.)

```
; for -256 ≤ value < 256, multiples of 2
MOV    #value / 2, Rn
SHLL   Rn

; for -512 ≤ value < 512, multiples of 4
MOV    #value / 4, Rn
SHLL2  Rn

; for -65536 ≤ value < 65536, multiples of 256
MOV    #value / 256, Rn
SHLL8  Rn

; for -16777216 ≤ value < 16777216, multiples of 65536
MOV    #value / 65536, Rn
SHLL16 Rn
```

Other instructions that could be useful for building constants are logical right shift and rotate. I'm not going to write them out, though. Use your imagination.

Now, it may seem cumbersome to have to use two instructions to generate a constant, but remember that these instructions are only 16 bits in size, so you can fit two of them in the space of a single MIPS, PowerPC, or Alpha AXP instruction. And if you can schedule the instructions properly, the fact that the SH-3 is dual-issue means that each of the instructions executes in a half-cycle, so the pair of them takes a single cycle, assuming you can schedule another instruction between them.

Next up are the control transfer instructions, and the return of the confusing branch delay slot, but the SH-3 adds more wrinkles to make them even more confusing.

Raymond Chen
Follow

The SuperH-3, part 10: Control transfer

Raymond Chen

August 16th, 2019

Yes, we have once again reached the point where we have to talk about branch delay slots. I will defer to the background information I provided when the issue arose in the discussion of the MIPS R4000. Basically, the branch delay slot is an instruction that occurs in the instruction stream after a branch. That instruction executes even when the branch is taken. (Of course, if the branch is not taken, the instruction executes normally as well.)

On the SH-3, the single-instruction branch delay slot is not sufficient to cover for the pipeline bubble created by a branch. Due to the pipeline structure, two instructions have already been fetched by the time the processor determines whether the branch is taken. The first such instruction goes into the branch delay slot, and the second one is converted to a `nop`. So even if you fill the branch delay slot, you still get a one-cycle stall for the discarded instruction. Therefore, you should prefer to structure branches so that they are normally not taken.

Okay, here we go.

```
BT      label   ; branch if T=1, reach is 256 bytes, squash the delay slot
BT/S    label   ; branch if T=1, reach is 256 bytes

BF      label   ; branch if T=0, reach is 256 bytes, squash the delay slot
BF/S    label   ; branch if T=0, reach is 256 bytes
```

The *branch if true* and *branch if false* test the *T* flag and branch if it is set (true) or clear (false). This particular branch is interesting because you get to choose whether you want the instruction in the delay slot to execute. Note that you already paid for the delay slot, so choosing not to execute it doesn't make things run any faster. The processor just converts the instruction to a `nop` and you waste a cycle.[1]

```
BRA     label   ; branch always, reach is 4KB
BRAF    Rn      ; branch to PC + Rn + 4
JMP     @Rn     ; branch to Rn

BSR     label   ; branch always, reach is 4KB, PR = return address
BSRF    Rn      ; branch to PC + Rn + 4, PR = return address
JSR     @Rn     ; branch to Rn, PR = return address

RTS             ; branch to PR
```

These instructions perform unconditional branches, either to a specific address within 4KB (*branch always*), to an address relative to the current program counter (*branch always far*), or to an address provided by a register (*jump*). The `xSR` instructions branch to a subroutine and record the return address in the special *pr* register. And of course after you branch to a subroutine, you need a way to get back, hence `RTS` *return from subroutine*.

The extra +4 in the `BRAF` and `BSRF` are due to pipelining. By the time the processor determines that the branch needs to be taken, the program counter has already moved ahead two instructions.

The Microsoft compiler doesn't use the `BSR` instruction because the linker is very likely to put the branch target outside the 4KB reach of the `BSR` instruction.

The Microsoft compiler uses the `BRAF` instruction in just one specific scenario (which we'll look at later), and it doesn't appear to use `BSRF` at all. The `BRAF` and `BSRF` instructions appear to be useful for writing position-independent code.

Watch out: Even though the `JMP` and `JSR` instructions use an `@`, there is no memory access going on. I don't know why the mnemonic uses an `@`.

Note that the `BT` and `BF` instructions have a very limited reach. If you need to branch further, you'll have to use a trick like branching to a branch, or reversing the sense of the test to jump over a branch instruction with greater reach.

```
    ; BT toofar

    ; option 1: branch to a branch (trampoline)
    BT      trampoline
...
trampoline:
    BRA      toofar+2
    delay_slot_instruction ; move first instruction of toofar here

    ; option 2: reverse the sense and jump over a branch

    BF      skip
    BRA     toofar+2
    delay_slot_instruction ; move first instruction of toofar here
skip:
```

The SH-3 deals with branch delay slots slightly differently from the MIPS R4000. The SH-3 temporarily disables interrupts between the branch instruction and its delay slot, so you cannot get interrupted in the branch delay slot.

If an exception occurs on the instruction in the branch delay slot, the exception is raised, and assuming the kernel fixes the problem, execution resumes at the branch instruction. This is safe because the branch instructions are all restartable; the only register modification is to *pr*, but none of the `xSR` instructions consume *pr*, so it's okay to re-execute them; you just set *pr* twice to the same value.

Some instructions are disallowed in a branch delay slot.

- Another branch instruction. Because duh.
- A `TRAPA` instruction. Sorry, no system calls in a branch delay slot. If you want to make a system call and return, you'll have to code the system call before the `RTS` and drop a `nop` into the branch delay slot.
- An instruction that uses PC-relative addressing. Because the program counter has already moved to the branch target, so your PC-relative address isn't what you think it is.

The last case is subtle. It means that the branch delay slot cannot contain a load of a value from a PC-relative address, nor can you use `MOVA` to load the address of a PC-relative value. If you need to pass a large constant as a parameter to a function, you'll have to do it ahead of the `JSR` and find something else to put in the delay slot.

If you put a disallowed instruction in a branch delay slot, the processor will raise an *illegal slot instruction* exception.

When it comes time to return from a subroutine, you often have two choices. You can use the `RTS` instruction or an equivalent `JMP @`:

Allowed	Not allowed
`lds.l @r15+, pr` `rts`	`mov.l @r15+, r1` `jmp @r1`

Both sequences are equivalent: They transfer control to the address popped off the stack. They just use a different register to do it. However, Windows requires that you use the first sequence. This is necessary so that function unwinding can be performed by the kernel in the case of an exception.

It's probably in your best interest to use the first version anyway, because it will work well with the return address predictor, should the SuperH ever gain one.

Next time we'll look at atomic operations, more specifically the lack of them.

[1] Technically, you are wasting *another* cycle, because a taken branch already suffers a loss of one cycle for the discarded second prefetched instruction. You're increasing the taken-branch cost from one cycle to two.

Raymond Chen

The SuperH-3, part 11: Atomic operations

Raymond Chen

August 19th, 2019

The SH-3 has a very limited number of read-modify-write operations. To recap:

```
AND.B #imm, @(r0, GBR)  ; @(r0 + gbr) &= 8-bit immediate
OR.B  #imm, @(r0, GBR)  ; @(r0 + gbr) |= 8-bit immediate
XOR.B #imm, @(r0, GBR)  ; @(r0 + gbr) ^= 8-bit immediate
TAS.B @Rn               ; T = (@Rn == 0), @Rn |= 0x80
```

These instructions are "atomic" in the sense that they occur within a single instruction and are hence non-interruptible. Technically, only the last one is truly atomic in the sense that the processor holds the data bus locked for the duration of the instruction.

Let's not quibble about such details. Let's just say we're looking for non-interruptible instructions.

The SH-3 does not support symmetric multiprocessing, so we don't have to worry about competing accesses from other main processors (although there may be competing accesses from coprocessors or hardware devices). But how are we going to build atomic increment, decrement, and exchange out of these guys?

Let's be honest. We can't.

We'll have to fake it.

Windows CE takes a different approach from how Windows 98 created atomic operations on a processor that didn't support them.

On Windows CE, the kernel is in cahoots with the implementations of the interlocked operations. If it discovers that it interrupted a special uninterruptible sequence, it resets the program counter back to the start of the uninterruptible sequence before allowing user mode to resume.[1] In this way, the kernel manufactures multi-instruction uninterruptible sequences.

These sequences have to be carefully written so that they are restartable. This means that they cannot mutate any input parameters, and there are no memory updates until the final instruction in the sequence.

For example, we could try to implement our fake `InterlockedIncrement` like this:

```
; on entry:
;   r4 = address to increment
; on exit:
;   r0 = incremented value

InterlockedIncrement:
    mov.l   @r4, r0     ; load current value    ; (1)
    add     #1, r0      ; increment it          ; (2)
    mov.l   r0, @r4     ; store updated value   ; (3)
    rts                 ; return               ; (4)
```

We load the current value from memory, add 1, store it back, and return. If this sequence is interrupt at any point, the kernel moves the program counter back to the first instruction and restarts the entire operation.

Let's walk through the possible interrupts.

- If interrupted prior to the first instruction, then moving the program counter back to the first instruction has no effect because that's where it already was. So no problems there.

- If interrupted prior to the second instruction, then we will perform the `mov.l @r4, r0` a second time. Since we haven't changed *r4*, this will read the desired memory location. It's a redundant read, but at least it's not harmful.
- If interrupted prior to the third instruction, then we will reload and re-increment the existing value. Again, since we haven't changed *r4*, this will read the correct location.
- If interrupted prior to the fourth instruction, then we're in trouble. We have already written the updated value back to memory, and restarting the operation will increment it a second time! **This code is broken.**

Aha, but we forgot about the branch delay slot of the `rts` instruction, and in fact it's the branch delay slot that provides our escape hatch: Move the final store *into the branch delay slot*.

```
; on entry:
;   r4 = address to increment
; on exit:
;   r0 = incremented value

InterlockedIncrement:
    mov.l   @r4, r0     ; load current value    ; (1)
    add     #1, r0      ; increment it          ; (2)
    rts                 ; return                ; (3)
    mov.l   r0, @r4     ; store updated value   ; (4)
```

Okay, let's run our analysis again.

- If interrupted prior to the first instruction, our analysis from above is still correct.
- If interrupted prior to the second instruction, our analysis from above is still correct.
- If interrupted prior to the third instruction, our analysis from above is still correct.
- An interrupt between the third and fourth instruction is not possible because the processor disables interrupts between a delayed branch instruction and its delay slot. But if an exception occurred (say, because the memory was copy-on-write), we can safely restart the operation because we haven't modified *r4* or the value in memory at *r4*.[2]
- If interrupted after the fourth instruction, then the program counter isn't in our special code region, so the kernel won't restart the sequence.

The branch delay slot saved us!

You never thought you'd see the day when you'd be thankful for a branch delay slot.

The kernel puts these special uninterruptible sequences in a contiguous region of memory. Let's say that it starts each special uninterruptible sequence on a 16-byte boundary. This means that the "special uninterruptible sequence detector" can go something like this:

```
    mov.l   @(usermode_pc), r0          ; see where we're returning to
    mov.l   #start_of_sequences, r1     ; the start of our special sequences
    mov     #length_of_sequences, r2    ; the size in bytes
    sub     r1, r0
    cmp/hs  r0, r2                      ; is it in the magic region?
    bf      fixme                       ; Y: then go fix it
return_to_user_mode:
    ... continue as usual ...

fixme:
    mov     #-15, r2                    ; mask out the bottom 4 bits
    and     r2, r0                      ; to go back to start of special sequence
    add     r1, r0                      ; convert from offset back to address
    bra     return_to_user_mode
    mov.l   r0, @(usermode_pc)          ; update user mode program counter
```

This is not actually how it goes, but it gives you the basic idea. In reality, the special uninterruptible sequences start on 8-byte boundaries, in order to pack them more tightly. Sequences that are longer than 4 instructions need to be arranged so that every 8 bytes is a valid restart point. I just used 16-byte sequences to make the explanation simpler.

For example, `InterlockedCompareExchange` really went like this:

```
; on entry:
;   r4 = address of value to test
;   r5 = replacement value (if current value matches expected value)
;   r6 = expected value
; on exit:
;   r0 = previous value

InterlockedCompareExchange:
    mov.l   @r4, r0     ; load current value
    cmp/eq  r0, r6      ; is it the expected value?
    bf      nope        ; Nope, just return current value
    mov.l   r5, @r4     ; Store the replacement value
nope:
    rts
    nop
```

There is a second restart point after four instructions, at the `rts`, and it's okay to restart there because the operation is complete. All we're doing is returning to our caller.

This trick for creating restartable multi-instruction sequences was not unique to the SH-3. Windows CE employed it to synthesize pseudo-atomic operations for other processors, too.

One curious side effect of this design for restartable multi-instruction sequences is that you can't debug them! If you try to single-step through these multi-instruction sequences, you'll get stuck on the first instruction: The breakpoint will fire, and the kernel will reset the program counter back to the first instruction.

Next time, we'll look at the Windows CE calling convention.

Bonus chatter: The SH-4A processor added load-locked and store-conditional instructions, bringing it in line with other RISC processors.

```
    MOVLI.L @Rm,r0          ; Load from @Rm, remember lock
    MOVCO.L r0,@Rn          ; Store to @Rn provided lock is still valid
                            ; T = 1 if store succeeded, 0 if failed
```

Bonus chatter 2: What about the TEB? Where does Windows keep per-thread information?

Turn out this is easier than it sounds. The SH-3 doesn't support symmetric multiprocessing, so there is only one processor, which therefore can be executing only one thread at a time. A pointer to the per-thread information is stored at a fixed location, and that pointer is updated at each thread switch.

[1] Fast Mutual Exclusion for Uniprocessors. Brian Bershad, David Redell, and John Ellis, Proceedings of the fifth international conference on Architectural support for programming languages and operating systems, 1992.

[2] Suppose an exception occurs in the delay slot because the memory isn't writable, and the exception handler fixes the problem (by making the memory writable on demand). Resuming execution will rewind the instruction pointer back to the start of the sequence because the memory value may have changed as part of handling the exception.

Raymond Chen

Follow

The SuperH-3, part 12: Calling convention and function prologues/epilogues

Raymond Chen

August 20th, 2019

The calling convention used by Windows CE for the SH-3 processor looks very much like the calling convention for other RISC architectures on Windows.

The short version is that the first four parameters (assuming they are all 32-bit integers) are passed in registers *r4* through *r7*, and the rest go onto the stack after a 16-byte gap. The 16-byte gap is the home space for the register parameters, and even if a function accepts fewer than four parameters, you must still provide a full 16 bytes of home space.

More strictly, the first 16 bytes of parameters are passed in registers *r4* through *r7*. If a parameter is a floating point type, then how it gets passed depends on how the parameter is declared in the function prototype.

- If the floating point type is prototyped as non-variadic, then it goes into the corresponding register *fr4* through *fr7*, and the integer register goes unused.
- If the floating point type is prototyped as variadic, then it stays in the integer register.
- If the function has no prototype, then the floating point type goes into both the floating point register and the integer register.

The reason for this rule is the same as before. Variadic parameters go into integer registers because the callee doesn't know what type they are upon function entry. To make things easier, variadic parameters are always passed in integer registers, so that the callee can just spill them into the home space and treat them all as stack-based parameters. And unprototyped functions pass the floating point values in both floating point and integer registers because it doesn't know whether the function is going to treat them as variadic or non-variadic, so it has to cover both bases.

Unlike the Windows calling convention for the MIPS R4000, the Windows calling convention for the SH-3 does not require 64-bit values to be 8-byte aligned. For example:

```
void f(int a, __int64 b, int c);
```

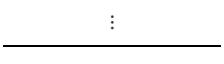
<table>
<tr><th>MIPS</th><th>Contents</th></tr>
<tr><td>a0</td><td>a</td></tr>
<tr><td>a1</td><td>unused</td></tr>
<tr><td>a2</td><td rowspan="2">b</td></tr>
<tr><td>a3</td></tr>
<tr><td>on stack</td><td>c</td></tr>
</table>

<table>
<tr><th>SH-3</th><th>Contents</th></tr>
<tr><td>r4</td><td>a</td></tr>
<tr><td>r5</td><td rowspan="2">b</td></tr>
<tr><td>r6</td></tr>
<tr><td>r7</td><td>c</td></tr>
</table>

On entry to the function, the return address is provided in the *pr* register, and on exit the function's return value is placed in the *r0* register. However, if the function's return value is larger than 32 bits, then a secret first parameter is passed which is a pointer to a buffer to receive the return value. The parameters are caller-clean; the function must return with the stack pointer at the same value it had when control entered.

If the concept of home space offends you, you can think of it as a 16-byte red zone that sits above the stack pointer.

The stack for a typical function looks like this:



param 6	(if function accepts more than 4 parameters)
param 5	(if function accepts more than 4 parameters)
param 4 home space	
param 3 home space	
param 2 home space	
param 1 home space	← stack pointer at function entry
saved registers ⋮	
saved return address	← stack pointer after saving registers
local variables ⋮	
outbound parameters beyond 4 (if any) ⋮	
param 4 home space	
param 3 home space	
param 2 home space	
param 1 home space	← stack pointer after prologue complete

The function typically starts by pushing onto the stack any nonvolatile registers, as well as its return address. This takes advantage of the pre-decrement addressing mode. In practice, the Microsoft C compiler allocates nonvolatile registers starting at *r8* and increasing, and preserves them on the stack in that order, followed by the return address.

In this example, the function has four registers to save, plus the return address.

```
function_start:
    MOV.L   r8, @-r15   ; push r8
    MOV.L   r9, @-r15   ; push r9
    MOV.L   r10, @-r15  ; push r10
    MOV.L   r11, @-r15  ; push r11
    STS.L   pr, @-r15   ; push pr
```

At some point (perhaps not immediately), the function will adjust its stack pointer to create space for its local variables and outbound parameters. If the function has a small stack frame, it can use the immediate form of the `SUB` instruction. Otherwise, it's probably going to load a constant into a register and use that as the input to the two-register form of the `SUB` instruction.

If the function has a large stack frame, it will be difficult to access variables far away from *r15* due to the limited reach of the *register indirect with displacement* addressing mode. To help with this problem, the compiler might park the frame pointer register *r14* in the middle of the frame, or at least close to a frequently-used variable, so that it can reach more local variables in a single instruction.

At the exit of the function, the operations performed in the prologue are reversed: The stack pointer is adjusted to point to the saved return address, and the saved registers are popped off the stack. Finally, the function returns with a `rts`.

```
    LDS.L   @r15+, pr   ; pop pr
    MOV.L   @r15+, r11  ; pop r11
    MOV.L   @r15+, r10  ; pop r10
    MOV.L   @r15+, r9   ; pop r9
    RTS                 ; return
    MOV.L   @r15+, r8   ; pop r8 (in the delay slot)
```

Lightweight leaf functions are those which call no other functions and which can accomplish their task using only volatile registers and the 16 bytes of home space. Such functions may not modify the *pr* register or any nonvolatile registers (which includes the stack pointer).

Next time, we'll look at some code patterns you'll see in the compiler-generated code, y'know, the stuff that goes *inside* the function. We'll start with misaligned data.

Raymond Chen

Follow

The SuperH-3, part 13: Misaligned data, and converting between signed vs unsigned values

Raymond Chen

August 21st, 2019

When going through compiler-generated assembly language, there are some patterns you'll see over and over again. Note that the code you see may not look exactly like this due to compiler instruction scheduling. In particular, the sequences for misaligned memory access may bring additional registers into play in order to avoid register dependencies.

First, is the unsigned memory access. Bytes and words loaded from memory are sign-extended by default. If you want to load an unsigned value, you need to perform an explicit zero-extension.

```
; load unsigned byte from address in r0
MOV.B   @r0, r1         ; loads sign-extended byte
EXTU.B  r1, r1          ; zero-extend the byte to a longword

; load unsigned word from address in r0
MOV.W   @r0, r1         ; loads sign-extended word
EXTU.W  r1, r1          ; zero-extend the word to a longword
```

Next up is misaligned data. The SH-3 does not support unaligned memory access. Not only that, but the kernel doesn't even emulate unaligned memory access. If you access memory from a misaligned address, you take an access violation and your process crashes. So don't mess up!

There are no special instructions for accessing misaligned data. You are on your own to take individual bytes and combine them into the desired final value, or to take the starting value and decompose it into bytes.

```
; store 16-bit value in r1 to possibly unaligned address in r0
; destroys r1
;                           r1      @r0
;                         xxxxAABB  xx xx
MOV.B   r1, @r0         ; xxxxAABB  BB xx
SHLR8   r1              ; 00xxxxAA  BB xx
MOV.B   r1, @(1, r0)    ; 00xxxxAA  BB AA

; store 32-bit value in r1 to possibly unaligned address in r0
; destroys r1
;                           r1      @r0
;                         AABBCCDD  xx xx xx xx
MOV.B   r1, @r0         ; AABBCCDD  DD xx xx xx
SHLR8   r1              ; 00AABBCC  DD xx xx xx
MOV.B   r1, @(1, r0)    ; 00AABBCC  DD CC xx xx
SHLR8   r1              ; 0000AABB  DD CC xx xx
MOV.B   r5, @(2, r0)    ; 0000AABB  DD CC BB xx
SHLR8   r1              ; 000000AA  DD CC BB xx
MOV.B   r1, @(3, r0)    ; 000000AA  DD CC BB AA

; read 16-bit value from possibly unaligned address in r0
;                           r1      r2        @r0
;                         xxxxxxxx  xxxxxxxx  BB AA
MOV.B   @(1, r0), r1    ; SSSSSSAA  xxxxxxxx
SHLL8   r1              ; SSSSAA00  xxxxxxxx
MOV.B   @r0, r2         ; SSSSAA00  SSSSSSBB
EXTU.B  r2, r2          ; SSSSAA00  000000BB
OR      r1, r2          ; SSSSAA00  SSSSAABB
                        ; r2 contains signed 16-bit value
EXTU.W  r2, r2          ; SSSSAA00  0000AABB
                        ; r2 contains unsigned 16-bit value

; read 32-bit value from possibly unaligned address in r0
;                           r1      r2        @r0
;                         xxxxxxxx  xxxxxxxx  DD CC BB AA
MOV.B   @(3, r0), r1    ; SSSSSSAA  xxxxxxxx
SHLL8   r1              ; SSSSAA00  xxxxxxxx
MOV.B   @(2, r0), r2    ; SSSSAA00  SSSSSSBB
EXTU.B  r2, r2          ; SSSSAA00  000000BB
OR      r2, r1          ; SSSSAABB  000000BB
SHLL8   r1              ; SSAABB00  000000BB
MOV.B   @(1, r0), r2    ; SSAABB00  SSSSSSCC
EXTU.B  r2, r2          ; SSAABB00  000000CC
OR      r2, r1          ; SSAABBCC  000000CC
SHLL8   r1              ; AABBCC00  000000CC
MOV.B   @r0, r2         ; AABBCC00  SSSSSSDD
EXTU.B  r2, r2          ; AABBCC00  000000DD
OR      r1, r2          ; AABBCC00  AABBCCDD
```

Less often, you will see code that sign-extends a 32-bit value to a 64-bit value.

```
; sign-extend 32-bit value in r0 to 64-bit value in r1:r0
MOV     r0, r1          ; copy value to r1
SHLL    r1              ; T contains high bit of value
SUBC    r1, r1          ; if T=0, then r1 = 00000000
                        ; if T=1, then r1 = FFFFFFFF
```

If you happen to have the value 0 lying around in a register, you could accomplish the task in two instructions:

```
; sign-extend 32-bit value in r0 to 64-bit value in r1:r0
; assumes that r2 already contains the value zero
CMP/GT  r0, r2          ; T = (0 > r0)
                        ; in other words, T=0 if r0 is positive or zero
                        ;                 T=1 if r0 is negative
SUBC    r1, r1          ; if T=0, then r1 = 00000000
                        ; if T=1, then r1 = FFFFFFFF
```

That is just code golf on my part. I haven't seen the compiler use this trick, or the next one.

```
; sign-extend 32-bit value in r0 to 64-bit value in r1:r0
; preserves flags
ROTCL   r0              ; rotate r0 left, copying high bit into T
                        ; and saving old T in low bit of r0
SUBC    r1, r1          ; if T=0, then r1 = 00000000, T stays 0
                        ; if T=1, then r1 = FFFFFFFF, T stays 1
ROTCR   r0              ; rotate r0 right to restore original value
                        ; and recover original value of T
```

In general, you'll see that SH-3 assembly code is somewhat verbose, even more so because compiler technology back in this time period was not as advanced as it is today, but you have to realize that each of these instructions is only half the size of the instructions of its RISC-style contemporaries, so even though you plowed through 2000 instructions, that's only 4KB of code.

Okay, next time, we're returning to reality and looking at function call patterns.

Raymond Chen

Follow

The SuperH-3, part 14: Patterns for function calls

Raymond Chen

August 22nd, 2019

Function calls on the SH-3 are rather cumbersome. The BSR instruction has a reach of only 4KB, which makes it impractical for compiler-generated code because the compiler doesn't know where the linker is going to put the function it's calling. In practice, all function calls in compiler-generated code are performed with the `JSR` instruction, which calls a function whose address is given by a register.

The typical case of a direct function call goes like this:

```
MOV.L   r3, @(16, r15)          ; parameter 5 passed on the stack
MOV     r8, r7                  ; parameter 4 copied from another register
MOV     #20, r6                 ; parameter 3 is address of local variable
ADD     r15, r6                 ; r6 = r15 + 20
MOV     #8, r5                  ; parameter 2 is calculated in place
MOV.L   #function, r0           ; r0 = function to call
JSR     @r0                     ; call the function
MOV     @(24,r15), r4           ; parameter 1 copied from the stack
                                ; (in the branch delay slot)
```

We load the function address into some register. The compiler usually uses one of the non-parameter scratch registers for this purpose, *r0* through *r3*. Note that we wrote this as a 32-bit immediate, but that is a pseudo-instruction which the assembler converts to a PC-relative load, with a constant embedded in the code segment.

```
; You write
MOV.L   #function_address, r0   ; r0 = function to call

; Assembler produces
MOV.L   @(n, PC), r0            ; r0 = function to call

... around n+4 bytes later ...
.data.l function_address        ; constant stored in code segment
```

The notation used by the Microsoft SH-3 assembler is that the name of a label is treated as its address. You don't need to say `offset` like you do in the Microsoft 80386 assembler.

We also prepare the parameters for the call. As we noted when we discussed the calling convention, the first four parameters go in registers *r4* through *r7*, and the rest go on the stack.

In practice, the parameters will be prepared in whatever order the compiler finds convenient, and they will be interleaved with the code that prepares the function address (and with each other) in order to improve scheduling.

The final instruction for setting up the parameters can go into the branch delay slot, provided it does not use a PC-relative addressing mode.

```
MOV.L   #function, r0           ; r0 = function to call
MOV.L   @(24, r15), r5          ; r5 = local variable
JSR     @r0                     ; call the function
MOV.L   #large_constant, r4     ; r4 = some large constant
^^^^^ ILLSLOT EXCEPTION         ; (in the branch delay slot)
```

The `MOV.L #large_constant, r4` will be encoded by the assembler as a PC-relative load, which is illegal in a branch delay slot. Fortunately, the assembler will not let you do this:

```
error A151: Can't compute PC displacement in a delay slot
```

To fix this, you'll have to move the PC-relative load out of the delay slot, preferably by swapping it with some instruction that it is not dependent upon.

```
    MOV.L   #function, r0           ; r0 = function to call
    MOV.L   #large_constant, r4     ; r4 = some large constant
    JSR     @r0                     ; call the function
    MOV.L   @(24, r15), r5          ; r5 = local variable
                                    ; (in the branch delay slot)
```

Calling a function through a global variable function pointer (such as through the import address table, in the case of a function that was declared as `__declspec(import)`) involves two memory accesses, one to get the address of the global variable, and another to get the code pointer.

```
    MOV.L   #variable, r0            ; r0 = variable that holds the fptr
    MOV.L   @r0, r0                 ; r0 = the address to call
    JSR     @r0                     ; call it
```

Here and in the subsequent examples, I've removed the parameter-loading instructions.

Calling a virtual function means getting the function address from the object's vtable.

```
    MOV     r8, r4                  ; r4 = "this" for function call
    MOV.L   @r4, r0                 ; load vtable pointer into r0
    MOV.L   @(n, r0), r0            ; load function pointer from vtable into r0
    JSR     @r0                     ; call it
```

And calling a naïvely-imported function means calling a stub.

```
    MOV.L   #stub_address, r0       ; r0 = pointer to stub function
    JSR     @r0                     ; call it

    ...
stub:
    MOV.L   #__imp__Function, r0    ; r0 = pointer to IAT entry
    MOV.L   @r0, r0                 ; r0 = the address to call
    JMP     @r0                     ; and jump there
    NOP                             ; (branch delay slot)
    .data.l __imp__Function         ; address of IAT entry
                                    ; (constant for first MOV.L instruction)
```

Our last common pattern for today is the dense switch statement.

```
    switch (value) {
    case 1: ...
    case 2: ...
    case 3: ...
    case 4: ...
    case 5: ...
    default: ...
    }

        ADD     #-1,r4              ; bias by lowest valid value
        MOV     #4,r3               ; is it in the range of our jump table?
        CMP/HI  r3,r4
        BT      default             ; N: go to default case
        MOV.L   #jump_table, r2     ; get address of jump table
        MOV     r4,r0               ; prepare for indexed addressing
        MOV.B   @(r0,r2),r0         ; r0 = instruction offset for case
        NOP                         ; (we'll see more about this nop later)
        BRAF    r0                  ; jump to appropriate handler
        NOP                         ; (nothing in the branch delay slot)

    ...
jump_table:
        .data.b 0x0
        .data.b 0x1a
        .data.b 0x2c
        .data.b 0x42
        .data.b 0x78
```

The code first subtracts the lowest non-default case value, producing an index so that all the interesting cases are in the range 0 to *n* for some *n*. If the value is not in that range, then we jump to the `default:`. Otherwise, we use the index as an index into a jump table of bytes, and use a `BRAF` instruction to perform a relative jump.

If there is a case label more than 127 bytes away from the `BRAF`, then the jump table expands to contain word offsets, and the index needs to be doubled before being looked up.

```
ADD     #-1,r4              ; bias by lowest valid value
MOV     #4,r3               ; is it in the range of our jump table?
CMP/HI  r3,r4
BT      default             ; N: go to default case
MOV.L   #jump_table, r2     ; get address of jump table
MOV     r4,r0               ; prepare for indexed addressing
ADD     r0,r0               ; convert byte offset to word offset
MOV.W   @(r0,r2),r0         ; r0 = instruction offset for case
BRAF    r0                  ; jump to appropriate handler
NOP                         ; (nothing in the branch delay slot)
```

We double the index by adding it to itself (`add r0, r0`). This is where the extra `NOP` from the previous case comes into play. The compiler leaves a `NOP` in its code generation so it can choose the size of the jump table later without having to go back and recalculate all its offsets.

In theory the compiler could have emitted the jump table directly into the code rather than dropping just the address of the jump table, which then needs to be indirected through in order to access the actual jump table. That has its drawbacks though: You have a potentially large jump table in your code, which pushes the jump targets further away and makes it more likely you're going to need a bigger table. And having the possibility of a variable-sized table means that the calculation of jump offsets requires multiple passes until all the consequences have stabilized. It's easier for the compiler to just generate a pointer to a jump table and figure out the jump table later.

I guess in theory if there is more than 64KB of code in the `switch` statement, the jump table might have to contain longword offsets, and the `NOP` becomes a `SLL2` to scale the index up so it can access a longword array. I've never seen a function so large that this became an issue, though.

Next time, we'll wrap up this whirlwind tour of the SH-3 processor by walking through some actual code.

Raymond Chen

Follow

The SuperH-3, part 15: Code walkthough

Raymond Chen

August 23rd, 2019

Once again, we wrap up our processor retrospective series by walking through a simple function from the C runtime library.

```
extern FILE _iob[];

int fclose(FILE *stream)
{
    int result = EOF;

    if (stream->_flag & _IOSTRG) {
        stream->_flag = 0;
    } else {
        int index = stream - _iob;
        _lock_str(index);
        result = _fclose_lk(stream);
        _unlock_str(index);
    }

    return result;
}
```

Here's the corresponding disassembly.

```
; int fclose(FILE *stream)
; {
        mov.l   r8,@-r15        ; push r8
        mov.l   r9,@-r15        ; push r9
        mov.l   r10,@-r15       ; push r10
        sts.l   pr,@-r15        ; save return address

        add     #-16,r15        ; allocate space for outbound calls
```

We start by saving the nonvolatile registers that we are going to be using as local variables in this function. Next, we allocate space on the stack to act as home space for our outbound calls. Most function start this way.

```
        mov     r4,r9           ; r9 = stream
```

This function enregisters the `stream` parameter, so save it from the volatile *r4* register into a non-volatile register *r9*. Other register variables are going to be *r10* for `result` and *r8* for `index`.

```
;   int result = EOF;
;
;   if (stream->_flag & _IOSTRG) {

        mov.l   @(12,r9),r3     ; r3 = stream->_flag
        mov     #64,r2          ; r2 = _IOSTRG
        and     r2,r3           ; r3 = stream->_flag & _IOSTRG
        tst     r3,r3           ; is it zero?
        bt/s    isfile          ; Y: so it's a file
        mov     #-1,r10         ; Set r10 = EOF
```

To test the flag, we load the value into a register (*r3*), load the constant `0x40` into another register so we can `AND` them together and test the result. The `TST` instruction implicitly tests against zero, so a *branch if true* means *branch if zero*. If the result is indeed zero, then we branch to the string handling case, but not before setting `r10` to `-1`, which initializes the `result` variable.

```
;       stream->_flag = 0;
;   }

        mov     #0,r3           ; prepare to store zero
        bra     done            ; and we're done
        mov.l   r3,@(12,r9)     ; stream->_flag = 0
                                ; (in the branch delay slot)
```

If we have a string, then we set `_flag` to 0 by loading the constant zero into a register and storing it. Then we jump to the common exit code.

```
;   } else {
;       int index = stream - _iob;

isfile:
        mov.l   @(42,pc),r2 ; #0x10004080 ; load constant address of _iob
        mov     r9,r8           ; r8 = stream
        mov     #-5,r3          ; prepare to shift right 5 places
        sub     r2,r8           ; r8 = stream - _iob (byte offset)
        shad    r3,r8           ; index = stream - _iob (element offset)
```

The `FILE` structure is a convenient 32 bytes in size, so the byte offset can be converted to an element offset by a simple shift. There is no right-shift-by-5 instruction, so we have to do a variable shift. There is no right-shift-by-variable instruction, so we instead do a left shift by the negative, because the left-shift instruction `SHAD` can shift both left *or* right, depending on the sign of the shift amount.

```
;       _lock_str(index);

        mov.l   @(36,pc),r3 ; #0x10001040 ; address of _lock_str
        jsr     @r3         ; call it
        mov     r8,r4       ; copy parameter from r8 = index
```

To call the `_lock_str` function, we put the `index` parameter in *r4* (in the delay slot), load up the address of the function, and then call it.

```
;       result = _fclose_lk(stream);

        mov.l   @(36,pc),r3 ; #0x10002130 ; address of _fclose_lk
        jsr     @r3         ; call it
        mov     r9,r4       ; copy parameter from r9 = stream
```

And another function call. Note that the displacement for the `@(36,pc)` is the same offset as the previous one, yet it loads a different value. That's because *pc* has changed!

```
;       _unlock_str(index);

        mov.l   @(32,pc),r3 ; #0x100010c8 ; address of _unlock_str
        mov     r8,r4       ; copy parameter from r8 = index
        jsr     @r3         ; call it
        mov     r0,r10      ; save return value of _fclose_lk into result
```

And then call `_unlock_str`. This time, we also have to save the return value from `_fclose_lk` so we can return it from the function.

```
;   }
;   return result;
; }

done:
        add     #16,r15     ; clean the stack
        mov     r10,r0      ; put return value into r0 register
        lds.l   @r15+,pr    ; pop return address
        mov.l   @r15+,r10   ; pop r10
        mov.l   @r15+,r9    ; pop r9
        rts                 ; return to caller
        mov.l   @r15+,r8    ; pop r8
```

And we reach the function exit. We put the return value in the *r0* register, because that's what the calling convention dictates. And we undo the stack operations we performed in the function prologue: Clean the stack and pop off the registers.

But wait, we're not done yet. We have those constants in the code segment that we need to generate.

```
        .data.l     _iob
        .data.l     _lock_str
        .data.l     _fclose_lk
        .data.l     _unlock_str
```

When you look at the disassembly, these data bytes are going to be disassembled as if they were code, because the disassembler doesn't know that they're actually data. You just have to understand that nonsense instructions after an unconditional branch are likely to be data.

Bonus chatter: Here's my attempt to hand-optimize the assembly.

First observation is that enregistering a variable that is used only once costs the same as spilling it. If you spill it, you write it to memory once and load it from memory once. If you enregister it, you write the original register to memory once, and restore it from memory once. Either way, you perform one read and one write. This means that the `stream` variable may as well be spilled.

Second observation is that there is really only one interesting live variable across each of the calls. Either we are saving the index, or saving the result. So we can use the same register to hold both.

And the third observation is that the compiler didn't take advantage of the free home space.

```
        mov.l   r8,@(12,r15)    ; save r8 in parameter 4 home space
        sts.l   pr,@(8,r15)     ; save pr in parameter 3 home space
        mov.l   r4,@(4,r15)     ; save stream in parameter 2 home space
```

I have 16 bytes of free memory, so I use it instead of pushing values onto the stack. I used 12 bytes of my home space, so need to allocate 12 bytes of stack to get myself back up to 16 bytes of home space for the outbound function calls. I'll interleave that with the next sequence of instructions to try to avoid a load stall.

```
        mov.l   @(12,r4),r3     ; r3 = stream->_flag
        add     #-12,r15        ; allocate space for outbound calls
        mov     #64,r2          ; r2 = _IOSTRG
        and     r2,r3           ; r3 = stream->_flag & _IOSTRG
        tst     r3,r3           ; is it zero?
        mov     #-1,r0          ; return value is EOF (if it's a string)
        bf      isstring        ; N: so it's a string
```

The code to test the flag hasn't really changed, but I moved the stack pointer adjustment into this sequence to avoid the stall that occurs when we try to use *r3* too soon after loading it from memory. This delay of the stack pointer adjustment is legal because we are allowed to advance instructions into the prologue provided they are not jumps and do not modify nonvolatile registers.

There is a stall between the `TST` and the `BF` because we are consuming flags immediately after generating them, so I slip a `MOV` instruction in there. The value is used only if the branch is taken, but it does no harm in the fallthrough case, and we may as well try it, since it's a free instruction due to the stall.

```
;       int index = stream - _iob;
;       _lock_str(index);

        mov.l   #_iob,r2        ; r2 = address of _iob
        mov     r4,r8           ; r8 = stream
        mov.l   #_lock_str,r0   ; address of _lock_str
        mov     #-5,r3          ; prepare to shift right 5 places
        sub     r2,r8           ; r8 = stream - _iob (byte offset)
        shad    r3,r8           ; index = stream - _iob (element offset)
        jsr     @r0             ; call _lock_str
        mov     r8,r4           ; copy parameter from r8 = index
```

The code to calculate the index hasn't really changed, but I interleave it with the preparation to call `_lock_str` to avoid a load stall.

```
;       result = _fclose_lk(stream);

        mov.l   #_fclose_lk,r3  ; address of _fclose_lk
        jsr     @r3             ; call it
        mov     @(20,r15),r4    ; parameter 1 is the stream
```

This is the same as before, except we load the stream from memory because we didn't dedicate a register to it. This does mean that if the `_fclose_lk` function tries to access its parameter within its first two instructions, it will suffer a load stall. (Normally, we'd have to count four instructions, but there is a one-cycle pipeline bubble on a taken branch, so that sucks up

two of the instructions.) However, `_fclose_lk` is almost certainly going to have at least one register variable, so those first two instructions are going to be occupied by spilling *r8* and *pr*. The earliest it is likely to access *r4* is its third instruction, so we're safe.

```
;       _unlock_str(index);

        mov.l   #_unlock_str,r3 ; address of _unlock_str
        mov     r8,r4           ; copy parameter from r8 = index
        jsr     @r3             ; call it
        mov     r0,r8           ; save return value of _fclose_lk into r8
```

The trick here is that the `result` variable becomes live at the same moment that `index` becomes dead, so we can use the same register *r8* for both of them. After the function returns, we put the saved value back into *r0* so we can return it.

```
        bra     done            ; to common exit code
        mov     r8,r0           ; put result back into r0 so we can return it
```

After `_unlock_str` returns, we go to our common exit code, with the desired return value in *r0*.

```
;   int result = EOF;
;   stream->_flag = 0;

isstring:
        mov      #0,r1          ; value to store into stream->_flag
        mov      r1,@(12,r4)    ; stream->_flag = 0
                                ; r0 is already -1
```

In the string case, we just zero out the `_flag` and return `-1`, which we preloaded into *r0* prior to the branch into this code path. Then we fall through to the common exit code.

```
done:
        lds.l   @(20,r15),pr    ; recover return address
        add     #12,r15         ; clean the stack
        rts                     ; return to caller
        mov.l   @(12,r15),r8    ; restore r8
```

And we're done. Our epilogue code is rather brief because we already put the desired return value in the *r0* register, and because we didn't have a lot of saved registers to restore. I put the `add` after the `lds.l` because I'm going to stall on the load delay, so I may as well get a free instruction out of it.

Raymond Chen

Follow