SH-5 Common Assembly Language Syntax

Introduction

The purpose of this document is to define a common assembly language subset to be followed by toolchains for the SH-5 architecture.

Scope and aims

The common assembly language subset is a minimal set of conventions to be followed by all SH-5 toolchains. The aim of defining this subset is to ease portability of assembly language code between the various toolchains.

SH-5 toolchains should accept assembly language which follows all of the rules in this document. Individual toolchains are free to adopt their own unique extensions to these rules.

Reference

Please refer to the *SH-5 CPU Core Architecture* manuals (05-CC-1000n) for further information.

Table of Contents

1 General principles

The syntax used by the *SH-5 CPU Core Architecture* manual (05-CC-1000n) shall be followed.

2 Syntax rules

2.1 Instructions

2.1.1 Instruction mnemonics

All instruction mnemonics specified in the *SH-5 CPU Core Architecture* manual (05-CC-1000n) will be accepted.

The spelling of the mnemonics must be identical to that specified in the *SH-5 CPU Core Architecture* manual, however all mnemonics shall be accepted both entirely in upper case and entirely in lower case.

For example, both `LD.L` and `ld.l` shall be accepted.

This rule does not require instructions written in a mixture of upper and lower case to be accepted, however, individual toolchains may accept such a mixture as an extension to this common assembly language subset.

As specified in the *SH-5 CPU Core Architecture* manual, all `PT` and conditional branch instructions have an optional suffix, `/L` or `/U`, to indicate whether the target is likely to be reached. This suffix must be written in the same case as the instruction. Omitting the suffix is equivalent to `/L`.

The syntax of the `PTA` and `PTB` instructions is

```
PTA  label, TRn
PTB  label, TRn
```

where the value of `label` is the value to be loaded into the target register. Direct usage of `PTA` and `PTB` should be discouraged, expect in cases where it is necessary to select either SHmedia or SHcompact explicitly). The pseudo-instruction `PT` (see *Section 2.2.5: Pseudo-ops* on page 7) should be used in preference to `PTA` and `PTB` wherever possible.

2.2 Instruction operands

2.2.1 Register names

All register names specified in the *SH-5 CPU Core Architecture* manual (05-CC-1000n) will be accepted. This includes the architecturally defined control register names: SR, SSR, PSSR, INTEVT, EXPEVT, PEXPEVT, TRA, SPC, PSPC, RESVEC, VBR, TEA, DCR, KCR0, KCR1, CTC, USR.

The spelling of the register names must be identical to that specified in the *SH-5 CPU Core Architecture* manual, however all register names shall be accepted both entirely in upper case and entirely in lower case.

For example., both CR0 and CR0 shall be accepted.

2.2.2 Immediate operands

In SHcompact instructions, immediate operands must be preceded by the `#` symbol.

In SHmedia instructions, immediate operands are not preceded by `#`.

Individual toolchains may choose to accept SHmedia immediate operands preceded by `#` as an extension to this common assembly language subset, but all toolchains must accept SHmedia immediate operands not preceded by `#`.

2.2.3 Scaled operands

SHmedia load, store, prepare-target and cache instructions automatically scale their immediate operands at run-time, for example `LD.L` scales the immediate offset by 4. Similarly, SHcompact load, store and branch instructions automatically scale their immediate operands at run-time.

In both SHmedia and SHcompact assembly language, the assembler should accept the scaled form of the operand, and remove the scaling when encoding the instruction. For example, in the assembly language

```
MOV.L @(8,R4),R0
```

the effective address is R4+8 (not R4 + 32). This is consistent with existing usage for SH-series architectures.

Table 1 indicates the SHmedia instructions that scale their operands and the amount of scaling. *Table 2* indicates the same for SHcompact. For all of these instructions, the assembler should accept the scaled form, and remove the scaling when encoding the instruction. Note that for

SHmedia instruction	Scaling applied to immediate operand
LD.UW, LD.W, ST.W	<< 1
LD.L, ST.L	<< 2
FLD.S, FST.S	<< 2
PTA, PTB	<< 2
LD.Q, ST.Q	<< 3
FLD.D, FST.D	<< 3
FLD.P, FST.P	<< 3
ALLOCO, ICBI, OCBI, OCBP, OCBWB, PREFI	<< 5

Table 1:

SHmedia prepare-target instructions and SHcompact branch instructions, the immediate operand is specified as a label indicating the required destination, rather than as a scaled operand.

SHcompact instruction	Scaling applied to immediate operand
MOV.W @(disp,PC),Rn MOV.W @(disp,Rm),R0 MOV.W @(disp.GBR),R0 MOV.W R0.@(disp,Rn) MOV.W R0,@(disp,GBR)	<< 1
BF, BT, BRA, BSR	<< 1
MOV.L @(disp,PC),Rn MOV.L @(disp,Rm),Rn MOV.L @(disp,GBR),R0 MOVA @(disp,PC),R0 MOV.L Rm,@(disp,Rn) MOV.L R0,@(disp,GBR)	<< 2

Table 2:

2.2.4 SHmedia branch targets

In SH-5 assembly language (both SHmedia and SHcompact), every label reference is implicitly treated as

```
BranchTarget(label)
```

unless the programmer explicitly writes

```
DATALABEL label
```

in which case the reference is treated as

```
DataLabel(label)
```

where

BranchTarget(*x*) = (*value(x) BITOR 1) if shmedia(x) is TRUE,*

= *value(x) if shmedia(x) is FALSE*

DataLabel(*x*) = *value(x)*

value(x), for some symbol *x*, is the symbol's value, as contained in the ELF st_value field,

shmedia(x), for some symbol *x*, is TRUE if the symbol is an SHmedia symbol, or more exactly, if the STO_SH5_ISA32 bit is set in the ELF st_other field.

DATALABEL is an operator recognized by the assembler. It may also be spelled entirely in lower case, that is, **datalabel**. As for instruction mnemonics and register names, an implementation is allowed to accept mixed case, for example, **DataLabel**, as an extension to the common assembly syntax subset.

Relocations need to take into account whether a label reference is a **BranchTarget** reference or a **DataLabel** reference, for example

```
.long DATALABEL label
```

can be relocated using R_SH_DIR32, but for

```
.long label + addend
```

the reference to **label** is treated as **BranchTarget**(**label**), and needs a relocation for **BranchTarget**(S)+A, where S is **label** and A is **addend**.

Note: It is the reference that specifies whether a label is to be used as a branch target or as the address of a piece of data, so a single label definition can be used as a branch target in one reference, and as a data label in another reference.

Examples

1 Literal pool entry to be used as a branch target.

```
.long label
```

The reference to **label** will be treated as a branch target. If **label** is an SHmedia symbol then the bottom bit of the value inserted will be set.

*Note: The behavior of the **.long** directive is to append a 32-bit value to the current section. This is only an example: this document places no requirement upon the spelling of the name of such a directive.*

2 Longword containing the address of the instruction at **label** (for example, debug information).

```
.long DATALABEL label
```

The reference to **label** will not be treated as a branch target due to the use of the **DATALABEL** operator, so the value inserted will be the address labelled by **label**.

3 Calling a function from SHcompact code.

This will work whether **function** is an SHcompact function or an SHmedia function, as the reference to **function** in the literal pool is treated as a branch target.

```
    MOV.L litpool,R0
    JSR @R0
    ...
litpool:
    .long function
```

4 Absolute call of a function from SHmedia code.

This will work whether **function** is an SHcompact function or an SHmedia function, as the reference to **function** used in the **SHORI** instruction will be treated as a branch target and will have the bottom bit set if **function** is an SHmedia function. **JSR** is a pseudo-op described in *Section 2.2.5*.

```
    MOVI (function >> 16) & 65535,R0
    SHORI function & 65535, R0
    PTABS R0,TR0
    JSR TR0
```

Note: This document does not specify the syntax of hexadecimal constants, so the decimal constant 65535 rather than hexadecimal FFFF has been used in this example.

5 Relative call of a function from SHmedia code.

This will work whether **function** is an SHcompact function or an SHmedia function, as the reference to **function** used in the **MOVI** instruction will be treated as a branch target. Note the use of the **datalabel** operator to ensure that the reference to **lab** is not treated as a branch target.

```
   MOVI (function - DATALABEL lab), R0
lab:
   PTREL R0,TR0
   JSR TR0
```

6 Relative call of a nearby function from SHmedia code.

PT is a pseudo-op described in *Section 2.2.5*. This will work whether **function** is an SHcompact function or an SHmedia function, as the **PT** pseudo-op will expand to **PTA** if **function** is an SHmedia function, or **PTB** if **function** is an SHcompact function.

```
   PT function,TR0
   JSR TR0
```

2.2.5 Pseudo-ops

The following pseudo-operations shall be supported, and will be expanded to an appropriate sequence of instructions by all toolchains. The register R25 is reserved by the ABI for use in such expansions.

PT label,TRa — This pseudo-op will be expanded into either a **PTA** or a **PTB** instruction, depending on whether the instruction at **label** is an SHmedia or an SHcompact instruction.

MOV Rm,Rd — In SHmedia code this pseudo-op will be expanded into the most efficient instruction that performs a register copy operation on the target processor. For the SH-5, this is **ORI Rm,0,Rd.**

JMP TRb — In SHmedia code this pseudo-op will be expanded into the most efficient instruction that performs an unconditional branch on the target processor. For the SH-5, this is **BLINK TRb,R63**.

JSR TRb — In SHmedia code this pseudo-op will be expanded into the most efficient instruction that performs a subroutine call on the target processor. For the SH-5, this is **BLINK TRb,R18.**

RTS TRb — In SHmedia code this pseudo-op will be expanded into the most efficient instruction that performs a subroutine return on the target processor. For the SH-5, this is **BLINK TRb,R63.**

2.3 Expressions

2.3.1 Symbols

Toolchains should accept symbol names that consist of an alphabetic or underscore character, followed by a sequence of alphanumeric or underscore characters.

2.3.2 Program counter ($)

The symbol **$** is a special symbol representing the program counter of the current instruction or directive.

References to **$** are treated just as references to labels, i.e. as **BranchTarget**(**$**), unless the **DATALABEL** operator is used explicitly. This means that in SHmedia code, bit 0 of **$** will always be set. The following are synonymous:

```
          MOVI lab-$,R40

tmplab:   MOVI lab-tmplab,R40
```

Examples

1 This is a (rather contrived) example of performing a position-independent function call from SHcompact code. It will work whether **function** is an SHmedia function or an SHcompact function:

```
   MOVA litpool,R1
   MOV.L @R1,R0
   ADD R1,R0
   BSRF R0
   ...
litpool:
   .long function - DATALABEL $
```

2.3.3 Operators

The following expression operators shall be accepted:

```
+ - << >> ~ & | * /
```

They shall have the same meaning and operator precedence as is defined for the C language.

SuperH, Inc.

Issued by the SuperH Documentation Group on behalf of SuperH, Inc.

 is a registered trademark of SuperH, Inc.

SuperH, Inc.
San Jose, U.S.A. - Bristol, United Kingdom - Tokyo, Japan

www.superh.com