Senior
Assembler & Simulator
user manual

November 13, 2024

Contents

1 Senior assembler

The Senior assembler is an assembler for the Senior DSP assembly language. The assembler uses three passes. In the first pass all source code, except undefined identifiers and labels, is translated to a hex format. In the second pass all remaining identifiers and labels are translated (if possible) and inserted into the right place in the hex code. In the third pass, hex file output is generated provided there were no errors in the previous passes. The hex file output is a text file for use with the Senior simulator only.
The following sections in this chapter will describe the use of the Senior assembler and the source file structure, the hex file structure and list possible error messages.
The Senior assembler instruction set is described in a separate document.

1.1 Command line syntax

The assembler command line uses the following syntax:

`srasm` *`infile`* [*`outfile`*] [*`0:accfile0.so`*] [*`1:acc1file.so`*] [*`more accelerators`*]

where `srasm` is the name of the Senior assembler program, *`infile`* is the name of the assembler source file and *`outfile`* is the optional name of the output hex file. If no name is given for the output hex file one will be generated using the name of the *`infile`* as the base with the extension `.hex` added. For instance if the *`infile`* is named `test1.asm` the outfile will be named `test1.hex`. The output hex file will always be written in the same directory as the input source file, unless a certain path is specified in the *`outfile`* parameter. For instance, running the command:

```
> srasm test1.asm out/test01
Senior assembler
Assembled successfully into 'out/test01'
```

will generate the file `test01` (without .hex extension) in the `out` directory (provided it exists). Running the command:

```
> srasm src/test1.asm
Senior assembler
Assembled successfully into 'src/test1.hex'
```

will generate the file `test1.hex` in the same directory as the infile `test1.asm`.
`0:accfile0.so`, means that the acceleator in file *`accfile0.so`* is assigned AP=0. *`1:accfile1.so`*, means that the acceleator in file *`accfile1.so`* is assigned AP=1. The same syntax applies to all accelerators. When the assembler finds an unknown instruction, it will look inside each of the accelerator files to find a match. For more information about accelerators and how to use them, read *senior_peripheral_manual.pdf*.

1.2 Assembler source file structure

The assembler source file typically consists of several sections. Usually there is first a section with defines and aliases declaring names of constants, registers and such for later use in the code. There are four memory spaces (`ram0`, `rom0`, `ram1` and `code`) defining where definitions and code eventually end up. Here is an example of a source file:

```
; === main.asm ===
; This is the main program

        .alias  sp      ar3
#define port1   0x0011

        .ram1
        .skip   128
stack

        .align 1024

        .code
#include "output.asm"

main
        set     sp,stack        ;initialize stack pointer
        call    stream_init     ;initialize stream

main_loop
        call    read_header
        call    layer2_decode
        jump    main_loop

        .rom0
table_bitrate
        .dw     0, -1,-1,-1,1, -1,2, 3,  4,  5,  6,  7,  8,  9,  10
table_C
        .scale 2.0
        .dfu    1.33333333333, 1.60000000000, 1.14285714286, 1.77777777777
table_D
        .scale 1.0
        .df     0.06250000000, 0.03125000000, 0.01562500000, 0.00781250000
```

Concerning indentation, label definitions MUST reside at the beginning of a line, hash directives (`#`), dot directives (`.`) and comments MAY start at the beginning of a line and everything else MUST NOT start at the beginning of a line. The various parts of the source code structure is explained in the following sections.

1.2.1 Reserved words

Reserved words are words that can not be used for anything else but their designated purpose. These are typically all the mnemonics (instructions and such), names of registers, accumulators, scale factors, etc, in short everything that is declared in the assembler by default. For instance the word `call` (for the instruction `call`) is reserved and can not be used as an identifier or a label, but for instance the words `_call` or `call4` are not reserved and can be used freely.

1.2.2 Mnemonics

Mnemonics are typically the instruction words and its various options and operators. An assembler source line, with an instruction, always start with the instruction itself followed by its arguments, if any. The exact syntax for all instructions and its options, the instruction set, is described in a separate document.
All mnemonics must be written in the source code using lowercase lettering.

1.2.3 Identifiers

In the assembler source file context, identifiers are names that refer to something typically defined by the programmer. A label, a constant or the definition of a register. The assembler lines

```
        .alias  sp      ar3
#define port1   0x0011
```

would define the identifiers `sp` and `port1` for reference to the address register `ar3` and the constant `0x0011` respectively, for later use in the code.
Identifiers must use the syntax `[a-zA-Z_][a-zA-Z0-9_]*`. This means they must start with a lowercase or uppercase letter from a to z or the underscore sign (_), then followed by a mix of any number of (0 or more) lowercase or uppercase letters a to z, digits 0 to 9 or underscore signs.
For instance, these are acceptable identifiers: `start1`, `_data_`, `AC9_xy`, and these are not acceptable: `7level`, `-track`, `acer~`.

1.2.4 Labels

Labels are identifiers and must follow the rules as such (see previous section). Label definitions MUST reside at the beginning of a line and MUST NOT be followed by anything else for the remainder of the line, except comments.

1.2.5 Comments

Comments may reside anywhere in the code. A comment is initiated by the semicolon character (`;`) and is active until the end of the line.

1.2.6 Numbers

Numbers can be either integer numbers or floating point numbers. Integer numbers can be entered using the decimal, hexadecimal or binary base. Decimal number are entered just the way they are, with an optional leading minus sign (`-`) if it is a negative number. Hexadecimal numbers are initiated using either `0x` or `$` and binary numbers start with %. For instance, these are decimal numbers: `-17`, `32700`, and these are hexadecimal numbers: `0x3A7F`, `$9001`, and this is a binary number: `%10010110`.
Floating point numbers can be entered in a decimal form only using the syntax `[0-9]+[.][0-9]+`, that is starting with one or several digits followed by a decimal point followed by one or several digits. Negative floating point numbers may be entered with a leading minus sign (`-`). For instance, these are floating point numbers: `3.14159265359`, `-0.33333333333`.

1.2.7 Directives

The assembler makes use of cpp (C pre processor) before processing the assembly source code. Hence cpp must be available in the search path when running the assembler. Cpp makes it possible to use some of its pre processor directives in the assembly code.

#include

The `#include` directive is a cpp directive. It includes another source file exactly as it is in the place of the `#include` directive. This way it is possible to divide a project into several assembly source files and include them wherever needed. The `#include` directive takes a file name within quotation marks as its only argument, like this:

```
#include "init.asm"
```

#define

The `#define` directive defines identifiers for constants. Look at the section Identifiers above for the definition of identifiers. A constant identifier may be defined like this:

```
#define port1   0x0011
```

.code

The `.code` directive is one of four memory space directives. It sets an internal memory space pointer to direct the following code generation to the program memory space. This program memory space is used for actual program content, that is instructions. The directive is valid until another memory space directive occurs.

.ram0

The `.ram0` directive is one of four memory space directives. It sets an internal memory space pointer to direct the following code generation to the first part of data memory 0. This directive is typically used together with the `.skip` directive (see below) to reserve larger sets of RAM memory. For instance the following code:

```
        .ram0
array4
        .skip 128
```

would reserve 128 data words in data memory 0 for access with the label `array4`. The directive is valid until another memory space directive occurs.

.rom0

The `.rom0` directive is one of four memory space directives. It sets an internal memory space pointer to direct the following code generation to the second part of data memory 0. This directive is typically used for ROM table content. It is stored in the hex output file together with the rest of the generated program content and loaded into the simulator at start. Used together with the `.dw`, `.df` or `.dfu` directives it is possible to generate ROM table data, for instance like this:

```
        .rom0
        .dw     0, -1,-1,-1,1, -1,2, 3,  4,  5,  6,  7,  8,  9,  10
        .dfu    1.33333333333, 1.60000000000, 1.14285714286, 1.77777777777
        .df     0.06250000000, -0.03125000000, 0.01562500000, -0.00781250000
```

The directive is valid until another memory space directive occurs.

.ram1

The `.ram1` directive is one of four memory space directives. It sets an internal memory space pointer to direct the following code generation to data memory 1. This directive is typically used together with the `.skip` directive (see below) to reserve larger sets of RAM memory. For instance the following code:

```
        .ram1
table7
        .skip 12000
```

would reserve 12000 data words in data memory 1 for access with the label `table7`. The directive is valid until another memory space directive occurs.

.skip

The `.skip` directive increments the internal memory space pointer for the current memory space by the number of its one and only argument. Skipping, thus reserving, that amount of memory. This directive only makes sense in the `ram0` or the `ram1` memory space.

.align

The `.align` directive takes one positive number as its argument. That argument will adjust the current memory space pointer to the next multiple of the argument number. For instance, if the current memory space pointer is at 1000 and the directive `.align 256` is set, the memory space pointer would be adjusted to 1024, since 1024 is divisible by 256. If using `.align 4` nothing will happen, since 1000 is already divisible by 4.

.alias

The `.alias` directive defines identifiers for already existing definitions of registers, accumulators, other identifiers and such. It takes two comma separated identifier arguments. The first one must not be previously defined while the second one must be previously defined. For instance, the following:

```
        .alias  sp      ar3
```

would define the identifier `sp` to be the very same as the address register `ar3`.

.scale

The `.scale` directive affects the way floating point numbers are stored. The directive takes one positive floating point number as its only argument. The directive will set a range for floating point numbers to be stored. For instance, the following:

```
        .scale 2.0
```

would set the range `[-2.0,2.0[` for signed floating point numbers and the range `[0,2.0[` for unsigned floating point numbers. The directive is valid until another `.scale` directive is set. Any floating point number outside the range will be saturated to within the range, with a given warning. The default `.scale` directive value is `1.0`.

.dw

The `.dw` directive will store a 16 bit integer number at the memory space pointer in the current memory space. The directive takes a comma separated list of signed or unsigned constants as its argument. It is only useful in the `rom0` memory space. It may be used for instance like this:

```
.rom0
.dw     7, -3, data3, $3F12 ; data3 is a previously defined constant
```

.df

The `.df` directive will store a signed floating point number at the memory space pointer in the current memory space. The directive takes a comma separated list of signed floating point constants as its argument. It is only useful in the `rom0` memory space. Each value will be divided by the scale factor (set by the `.scale` directive) and then converted to a binary representation with a sign bit and 15 fractional bits. This 16 bit value is finally stored in memory. If the value does not fit this representation it will be saturated. It may be used for instance like this:

```
.rom0
.scale 2.0
.df     0.0, 0.5, -1.0, 1.0, -2.0, 2.0
```

which will be equivalent with the following code:

```
.rom0
.dw     $0000, $2000, $C000, $4000, $8000, $7FFF
```

Note that the sixth value (`2.0`) has been saturated.

.dfu

The `.dfu` directive is similar to the `.df` directive. The difference is that `.dfu` uses an unsigned representation with 16 fractional bits. It may be used for instance like this:

```
.rom0
.scale 2.0
.dfu    0.0, 0.5, 1.0, 1.5, 2.0
```

which will be equivalent with the following code:

```
.rom0
.dw     $0000, $4000, $8000, $C000, $FFFF
```

Note that the fifth value (`2.0`) has been saturated.

1.3 Special I/O ports

There are four I/O ports that have special meaning for the assembler and simulator. They are the following:
These I/O ports are either in or out only, to be used with the instructions `in` or `out` respectively.

Using the instruction:

```
in      r14,$0010
```

Address	I/O	Description
$0010	in	Read data from the external file "IOS0010"
$0011	out	Write data to the external file "IOS0011"
$0012	out	Stop simulation and exit simulator
$0013	out	Stop simulation, same as pressing CTRL-c

will read the next data from the file "`IOS0010`" to general register `r14`. The file "`IOS0010`" must be a plain text file with 16 bit hexadecimal values, one value per line. It may look something like this:

```
0087
7a71
bb27
005e
308d
006d
0022
```

Using the instruction:

```
        out     $0011,r21
```

will write data from the general register `r21` to the file "`IOS0011`". The format of the file "`IOS0011`" is the same as for the file "`IOS0010`" above.

Using the instruction:

```
        out     $0012,r0
```

will stop the simulation and exit the simulator. This is useful when using external scripts and makefiles for running several simulations sequentially without human intervention. The contents of the source register (`r0`) will be ignored.

Using the instruction:

```
        out     $0013,r0
```

will stop the simulation and return to the simulator prompt, thus remaining in the simulator environment. This is useful when there is a need to inspect processor status (registers, flags, memories, etc) at a certain point in the program after running at full speed. Think of it as a break point. The contents of the source register (`r0`) will be ignored.

1.4 Hex file structure

1.5 Error messages

The Senior Assembler will provide you with an error message if something has gone wrong. Table 1.1 will give you a description of the problem.

Error	Description
Too many arguments	Arguments 4 to 11 are used for accelerators, more arguments than that is not allowed.
Usage: srasm ¡infile¿ [outfile] ...	Wrong arguments given.
Unable to open '<*filename*>'	Could not open the specified input file.
Unable to create <*filename*>	Could not open the specified output file.
Wrong number of registers.	Wrong number of registers given to an accelerator instruction.
No OP code	Assembler did not find an accelerator with the specified accelerator instruction name.
Dot directive df saturating <*value*> (scale=<*value*>)	Stored .df value too large for memory and becomes saturated.
Dot directive dfu saturating <*value*> (scale=<*value*>)	Stored .dfu value too large or too small for memory and becomes saturated.
Use one argument only for dot directive 'skip'. Ignoring extra arguments	.skip takes only one argument.
Ignoring dot directive 'skip' in code segment	Skip only makes sense in the **ram0** or the **ram1** memory space.
Negative argument for dot directive 'skip'	.skip only allows positive values in its argument.
Use one argument only for dot directive 'align'. Ignoring extra arguments	.align takes only one argument.
Ignoring dot directive 'align' in code segment	.align is not usable in code space.
Negative argument for dot directive 'align'	.align only accepts positive values.
Second argument for define is not an integer	Wrong argument to **#define** statement.
8 bit mode not allowed with bit reversed addressing.	Cannot combine 8 bit reading/writing with bit reversed
Illegal addressing mode for stack operation	Cannot manually use stack pointer for addressing in mo
Illegal addressing mode for dm0[/dm1]	Trying to use illegal addressing mode.
Illegal register extension	Trying to use an illegal extension mode.
Illegal addressing mode	Trying to use an addressing mode which is not allowed
Illegal repeat count	Repeat count need to be a positive number.
Repeat label out of range	The repeat label needs to be within 0 to 127 addresses
Ambiguous use of code space	

Table 1.1: Senior Assembler error messages

2 Senior simulator

The Senior simulator is a simulator for the Senior DSP. Although the Senior DSP has a pipelined architecture the simulator operates from a programmers point of view, so that each instruction is seemingly executed immediately, though all cycle events are kept track of and executed accordingly. In other words, the Senior simulator is not a fully true pipeline cycle simulator.

2.1 Command line syntax

The simulator command line uses the following syntax:

`srsim` [`-r`] [`-c` *`config_file`*] [`-p` *`pm_size`*] [`-0` *`dm0_size`*] [`-1` *`dm1_size`*] *`program`*

where `srsim` is the name of the Senior simulator program and *`program`* is the filename of the program to simulate, the file produced as output from the Senior assembler. All other arguments are optional and work as follows:

Option	Description	Maximum value	Default value
-r	run program directly when loaded	---	---
-c	specify project configuration file	---	---
-p	set size for program memory	65536	65536
-0	set size for data memory 0	65536	65536
-1	set size for data memory 1	65536	65536

For instance, running the command:

```
> srsim -p 4096 test1.hex
Senior simulator
Allocated 4096 words for program memory
Allocated 65536 words for data memory 0
Allocated 65536 words for data memory 1
Read 924 words to program memory
sim>
```

will set program memory size to 4096, load the program `test1.hex` to program memory and wait for command input at the `sim>` prompt. OBSERVE, all sizes for program memory and data memories must be large enough to hold its designated content or there will be a memory access violation. If the size of an data or program memory is specified both by a configuration file and by the command line, any of the values may be used.
When using the option `-r`, simulation may be aborted by pressing *`CTRL-c`* which will also exit the simulator.
Note that the simulator should be executed from the same directory as the assembler was executed from due to non absolute filenames in hex file.

2.2 Configuration file

Below is an example of the optional xml-formatted project configuration file.

```
<configuration>
  <constraints>
    <pm>65536</pm>
    <dm0>256</dm0>
    <dm1>512</dm1>
  </constraints>
  <iomapping>
    <map adr="15">acc1</map>
  </iomapping>
  <accelerators>
    <acc1>./accelerators/accelerator.so</acc1>
  </accelerators>
  <io_per_mapping>
    <map adr="1">timer</map>
  </io_per_mapping>
  <peripherals>
    <timer>./sim/peripherals/timer.so</timer>
  </peripherals>
</configuration>
```

The accelerator, called ”acc1” (may be any name), is located in file ”./accelerators/accelerator.so” and attached to the General I/O at address 15. The timer peripheral is located in file ”./sim/peripherals/time and is attached to the General I/O at address 1.

2.3 Simulator commands

The simulator has a few ”in house” commands (explained below) for use when simulating. It is always possible to repeat the last command by pressing *`CTRL-d`*.

2.3.1 h/? — print help

Print a list of available commands. It should look something like this:

```
sim> h
Senior simulator (V2020-TSEA26-01)
----------------------------------------
p               print registers
r <n>           run <n> cycles
n               run next instruction
d0 [n]          dump dm0 from address [n]
d1 [n]          dump dm1 from address [n]
g               go, until CTRL-C
e               toggle event
l               list program around pc
al              toggle automatic program listing around pc when program stops
b <addr>        set breakpoint at address
```

```
b <file> <line> set breakpoint at a source code line
h/?             display this help
q               quit
----------------------------------------
sim>
```

2.3.2 p — print registers

Print the contents of all 32 general registers, all 32 special registers, all 4 accumulators, the settings of all flags, the value of the program counter, and the cycle counter. It should look something like this:

```
sim> p
r:
0000 0040 0000 0000 0000w0000 0000 0000
0000 0000 0000 0000 0000w0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
sr:
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 3F12r0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
acr:
0000000000 0000000000 0000000000 0000000000
flags:
MV:0 MS:0 MN:0 MZ:0 AV:0 AC:1 AN:0 AZ:0
pc:     sp:   cc:
0003    0       3
   test1.asm:14:        set     r7, #$3F7A
sim>
```

Lastly, the next source code line to be executed may be printed. Printed is the source code file name, line number and then the actual source code line. The simulator is not aware of the source code file location, so for this to work the source code file has to be in the same directory as from where simulator is started.
After each and one of the general and special registers there may be one of the characters `r` or `w` printed in either lowercase or uppercase. A lowercase `r` or `w` signifies this register is scheduled for a reading or writing, respectively, at a later cycle. An uppercase `R` or `W` signifies this register is to be read or written, respectively, at the current cycle.

2.3.3 l — list program

List the source code program around the current program counter address. Some addresses before and some after the current program counter will be printed. The current source code line to be executed is notified with an arrow, `->`. It should look something like this:

```
sim> l
        set     r5,2

        lsr     r1,r8,12
        lsr     r2,r8,10
```

```
        lsr     r3,r8,9
        lsr     r4,r8,6
        lsr     r6,r8,4
        set     r8,1
 ->     and     r1,15
        and     r2,3
        and     r3,1
        and     r4,3
        and     r6,3
        st0     (ar2++),r1      ; bitrate
        cmp     3,r4            ; mode_single_channel
        move.eq r5,r8           ; channels (1 or 2)
sim>
```

The simulator is not aware of the source code file location, so for this to work the source code file has to be in the same directory as from where the simulator is started.

2.3.4 r — run a number of cycles

Run a specified number of cycles. This command takes one argument, an integer number, specifying how many cycles to run. After running the specified number of cycles the `p` command, print registers, is executed. It should look something like this:

```
sim> r 4
Simulating 4 cycle(s):[100..104[
r:
0040 000aW0000w0000w0001w0002 0a04 0000
0001 0001 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
sr:
0000 0000 0004 007e 0000 0000 0000 0000
0000 0000 0000w0001 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
acr:
0000000000 0000000000 0000000000 0000000000
flags:
MV:0 MS:0 MN:0 MZ:0 AV:0 AC:0 AN:0 AZ:0
pc:     sp:   cc:
00c2    1       104
   test1.asm:79:  and     r6,3
sim>
```

2.3.5 d0 — dump data memory 0

Dump (print) 128 address contents, starting from a specified address, from data memory 0. This command takes one optional argument, an integer number, specifying from what address to start dumping. If no argument is given, dumping will continue from after the last address previously dumped. It should look something like this:

```
sim> d0 33368
8258: 9592 8276 8276 9592 b8e3 e707 18f9 471d
8260: 6a6e 7d8a 7d8a 6a6e 471d 18f9 e707 b8e3
8268: 9592 8276 83d6 a129 d4e1 12c8 4c40 73b6
8270: 7fd9 6dca 41ce 0648 c946 9930 8163 877b
8278: aa0a e0e6 1f1a 55f6 7885 7e9d 66d0 36ba
8280: f9b8 be32 9236 8027 8c4a b3c0 ed38 2b1f
8288: 5ed7 7c2a 8583 aecc f374 3c57 70e3 7f62
8290: 62f2 2528 dad8 9d0e 809e 8f1d c3a9 0c8c
8298: 5134 7a7d 7a7d 5134 0c8c c3a9 8f1d 809e
82a0: 9d0e dad8 2528 62f2 7f62 70e3 3c57 f374
82a8: aecc 8583 877b be32 12c8 5ed7 7fd9 66d0
82b0: 1f1a c946 8c4a 83d6 b3c0 0648 55f6 7e9d
82b8: 6dca 2b1f d4e1 9236 8163 aa0a f9b8 4c40
82c0: 7c2a 73b6 36ba e0e6 9930 8027 a129 ed38
82c8: 41ce 7885 89be cf04 30fc 7642 7642 30fc
82d0: cf04 89be 89be cf04 30fc 7642 7642 30fc
sim>
```

2.3.6 d1 — dump data memory 1

Dump (print) 128 address contents, starting from a specified address, from data memory 1. This command works in the same way as the command `d0`, but from data memory 1. See above.

2.3.7 g — run until CTRL-C pressed

Run continously until *CTRL*-`c` is pressed. Having run the command and then pressed *CTRL*-`c` it should look something like this:

```
sim> g
Running...(break with CTRL-C)...
Execution aborted at PC:626, CYCLE:3892040
sim>
```

It will also stop on a breakpoint if set. See information in Section **??** on how to set the breakpoint. For example:

```
sim> b 4
sim> g
Running...(break with CTRL-C)...
Breakpoint at PC = 0004
sim>
```

2.3.8 e — toggle event

Toggle event messaging on or off. Some events during simulation like saturation from calculations or delay slot jump activation, may be notified by event messaging. It can be useful when stepping through a program but may be annoying when running at full speed. Toggling event messaging should look something like this:

```
sim> e
Event is on
```

```
sim> e
Event is off
sim>
```

Event messaging is off by default.

2.3.9 b — set breakpoint

Set a breakpoint to a give address. When the program counter reaches the breakpoint address the simulation will be halted. Useful for running command with g and exit when reaching a particular part of the code. See example in Section 2.3.7.

2.3.10 q — quit

Quit and exit the simulator.

2.4 Error messages