sf20asm

Standalone assembler
for the sf20 family of
16-bit microprocessors

User Manual

Revision 1.0
21. December 2014

Author: Martin Raubuch

info@racors.com

Revision History

Revision	Date	
1.0	21Dec2014	• First version

Table of contents

1 Overview

1.1 Introduction

The sf20asm is a standalone assembler for the sf20 family of microprocessors. Standalone means that the tool directly generates a code file with absolute instruction and data addresses. An extra linking step is not required. The sources can be contained in multiple files. The assembler has an **include** directive to handle multiple source files.

Syntax of instructions and directives is compatible with the GNU assembler. Translation of sources for use with the GNU assembler is an easy step. Only the scope of symbols (labels, variable names) has to be adapted (local versus external) to enable the linking of the GNU assembler object code output from multiple source files.

1.2 OS support

The sf20asm is a shell tool. Binaries are available for Linux and for Windows with Cygwin software installed.

1.3 Installation

On Linux machines copy the binary **sf20asm** into the /bin directory. On Windows machines with Cygwin copy the binary **sf20asm.exe** into the cygwin/bin directory.

1.4 Feature summery

- Shell based command line tool
- Supports the sf20b and sf20d ISAs
- Dual pass assembler concept with executable code generation (no linking)
- Multiple source/header files are handled with an **include** directive
- Syntax and directives are close to the GNU assembler
- Comprehensive generation of self-explanatory error messages
- Simple, text readable code file format with text, data and symbol records in a single file
- Code files can be read directly by the **dcs** simulator and the **sf20db** command line debugger
- Generates a log file with the generated code and a symbol table

1.5 Invocation

The assembler is invoked by typing **sf20asm** in the command shell followed by a single source file name as argument. For programs with multiple source files the easiest solution is to create a master source file with **include** directives for all other source files and then use the master source file as argument when the assembler is invoked.

The assembler generates two output files, a code file with the same name as the source file but with a .**cod** extension instead of the .**asm** extension of the source file. The second output file is the log file which has the same name as the source file but with a .**log** extension.

1.6 Structure of this manual

Below are brief descriptions of the remaining chapters of this manual:

Source file format, describes the syntax of comments, labels, directives and instructions

Output file formats, describes the format of the generated code and log files

Notes, contains some hints for programmers

2 Source file format

2.1 Overview

Assembler source files have line based syntax. The following line types are distinguished by the first character of the line:

- Lines starting with a ';' character are comment lines
- Lines starting with a white space character and have no further syntax elements (blank lines)
- Lines starting with a 'a'-'z' or 'A'-'Z' letter or a '_' character are global label lines
- All other lines are instruction or directive lines

Comment and blank lines are ignored by the assembler.

Global label lines start with an identifier (see definition further below) in the first character of the line. The identifier is immediately followed by a ':' character. No further syntax elements except a comment following the ';' character are allowed.

Instruction and directive lines have a common format with three main parts from left to right

1. Local label (optional)
2. Instruction or directive mnemonic
3. Operand field (optional)

The parts must be separated by at least one space character. Local labels must start in the first character of a line. If the first character is a space character no label is present and the next non-space character is interpreted as the start of a mnemonic.

Lines of source files can have a maximum length of 256 characters. If the parser detects a ';' character the remainder of the line is treated as comment and is ignored.

2.2 Operand field syntax

The sf20asm assembler implements the operand field syntax as defined in the sf20 ISA reference manuals. Address and data constants of instructions and directives are specified as arithmetic expressions. The operands of arithmetic expressions are symbolic constants (labels, defined symbols) and numbers. A full set of integer operators (arithmetic, logic and shifts) is available.

The next paragraphs define the supported syntax elements.

Single character elements: The following characters are used by the sf20 assembler language as defined in the ISA reference manuals (put in a single string for better readability) ", . : + - * ( )". In addition the following characters are used as arithmetic operators in expressions " / % & | ^ ~ =".

The " + - * " characters are used both as arithmetic operators and in the sf20 assembler language syntax.

Dual character elements: The ">>" and "<<" symbols are used as shift operators in arithmetic expressions.

Numbers: 32-bit signed or unsigned integers, can be specified in decimal format (sequence of the letters 0-9), in hexadecimal format (prefix 0x followed by a sequence of the letters 0-9,a-f,A-F) or in binary format (prefix 0b followed by a sequence of letters 0 and 1).

Identifiers: sequences of letters a-z, A-Z, 0-9, and the '_' characters, must start with a non-digit character, maximum length is 80 characters, identifiers are case sensitive, keywords (see below) cannot be used as identifiers

Strings: sequences of any printable characters and blanks (except for the double quote '"' character) enclosed in double quote '"' characters, CR (carriage return) control characters can be specified with the \n 2-letter sequence, maximum length is 256 characters.

Keywords: The processor register names and a number of other words used in the sf20 assembler language are reserved keywords and cannot be used as identifiers. Keywords are not case sensitive. The following is a list of all keywords in alphabetical order: **ae0**, **ae1**, **ae2**, **ae3**, **base**, **cc**, **cs**, **dsp**, **ep**, **ia**, **id**, **lc**, **r0**, **r1**, **r2**, **r3**, **r4**, **r5**, **r6**, **r7**, **r8**, **r9**, **ra**, **rb**, **rc**, **rd**, **re**, **rf**, **sa**, **ta u0**, **u1**, **u2**, **u3**.

Instruction and directive mnemonics are treated separately by the parser and are therefore not keywords. It is however recommended not to use them as identifiers.

Expressions: expressions are a way to specify address and data constants in the operand fields of directives and instructions indirectly or relative instead of using number constants directly, similar as 'defines' are used in C language programs. Operands of expressions can be numbers, global labels, local labels and symbolic constants defined with **.set** directives. Operators and the way they are used are compatible with C-language expressions regarding associativity and precedence. Round brackets '(' ')' can be used to change the order of operations. The following unary and binary operators are supported:

Unary operators: - , +, ~
Binary operators: + , - , * , / , % , & , | , ^ , >> , <<

2.3 Labels

The assembler supports GNU syntax compatible global and local labels. Global labels must follow the syntactical rules for identifiers (see **Operand field syntax** section).

One major difference to the GNU assembler is that the scope of global labels is the entire program and not only the local source file. Therefore global labels must be unique across all source files of a program and don't need to declared 'external' if referenced in a different source file.

Another difference is that global labels must be specified in separate lines. Only local labels can be placed together with an instruction or directive in the same line.

Local labels must start in the first character of a line. They consist of a single digit '0' – '9' character followed by a ':' character. Consequently only 10 different local labels can be specified, but each of them can be used multiple times in the same source file and across an entire program. The scope of a local label is between the next and previous occurrence of the same label.

Local labels are referenced by their number followed by a 'b' (backward) or 'f' (forward) character. E.g. 4b references the next 4: label in backward direction, 7f references the next 7: label in forward direction. Local labels are most commonly used as targets of short distance branch and jump instruction.

2.4 Directives

Directive lines start with an optional local label followed by the directive mnemonic followed by an optional operand field. Directive mnemonics start with a '.' (full stop) character.

For the following syntax descriptions single character syntax elements are enclosed in single quote characters, optional elements are enclosed in square brackets, mandatory elements are enclosed in angular brackets and repetitive elements are enclosed in curly brackets as summarized below:

'single character syntax element'

[optional syntax element]

<mandatory syntax element>

{repetitive syntax element}

The general syntax format of directives is:

[label] <mnemonic> [operand field]

The following paragraphs are detailed descriptions of the sf20asm assembler directives in alphabetical order, directive mnemonics are printed in bold.

[local label] <**.adrofs**> <offset>

Address offset. The specified 16-bit offset (type = expression) is added to the address of generated code records of the current section (test or data). Note that the sf20asm assembler has separate address counters for the text and data sections. The specified offset is added only to code record addresses of the current section.

This directive is typically placed immediately after an **.org** (origin) directive. Purpose is to generate code that can be loaded directly into data or instruction memories that do not start at address zero in the respective processor address space (instruction or data). In most cases the specified offset is negative to compensate the non-zero start address of a memory. The default offsets for both text and data sections are zero.

[local label] <**.ascii**> <string>

Ascii constant data. The text string is converted to a sequence of bytes (one byte per character) and added to the current code section at the current address. The current section address is then incremented by the length of the string.

This directive is equivalent to the **.byte** directive specifying individual characters of a string with their ASCII byte codes, separated by commas. The **.ascii** directive is a more convenient way to define text readable data as constant byte data.

[local label] <**.asciz**> <string>

Ascii constant data for null-terminated strings. The text string is converted to a sequence of bytes (one byte per character) and added to the current code section at the current address. An extra 0x00 byte is appended

at the end of the byte sequence. The current section address is then incremented by the length of the string + 1. This directive is used the same way as the **.ascii** directive but for null-terminated strings.

[local label] <**.byte**> <val1> [{',' valn}]
Constant byte data. The specified constant byte data are added to the current code section at the current address. The current section address is then incremented by the number of specified data items. At least one data value <val1> must be specified, additional data items [val**n**] (**n** = 2 to max 32) can be added separated by commas.

[local label] <**.data**>
Switch to data section. The assembler switches to the data section. Any code generated from this point to the next **.text** directive is written to the data section.

[local label] <**.include**> <string>
Include source file. The file specified by the string is included as source file. The parsing of source lines immediately switches to the included file. Because the scope of global labels is the entire program the order in which files are included is relevant. Nested include files are not supported.

[local label] <**.isa**> <base|dsp>
Select processor ISA (Instruction Set Architecture). The ISA selection determines the supported instruction set. This **sf20asm** version supports the sf20b (base) and sf20d (dsp) ISAs.

[local label] <**.long**> <val1> [{',' valn}]
Constant long data. The specified constant long (32-bit) data words are added to the current code section at the current address. The current section address is then incremented by the number of specified data items multiplied by 4. At least one data value <val1> must be specified, additional data items [val**n**] (**n** = 2 to max 32) can be added, separated by commas.

[local label] <**.org**> <address>
Origin. The address counter of the current section (text or data) is set to the specified byte address. The address is a 16-bit value for both data and text sections. When the **sf20asm** assembler is invoked the address counters of both text and data sections are set to zero.

[local label] <**.p2align**> <exponent>
Power of 2 align. The address counter of the current section (text or data) is set to the next address that is aligned to the specified power of two.

[local label] <**.set**> <name> ',' <value>
Define constant. A constant symbol with the specified name (identifier) and specified value (32-bit expression) is added to the symbol list. The symbol can be used as operand in expressions.

[local label] <**.short**> <val1> [{',' valn}]
Constant short data. The specified constant short (16-bit) data words are added to the current code section at the current address. The current section address is then incremented by the number of specified data items multiplied by 2. At least one data value <val1> must be specified, additional data items [val**n**] (**n** = 2 to max 32) can be added, separated by commas.

[local label] <**.text**>
Switch to text section. The assembler switches to the text section. Any code generated from this point to the next **.data** directive is written to the text section.

2.5 Instructions

Syntax

The **sf20asm** assembler is fully compatible with the assembler syntax described in the sf20 ISA reference manuals. Details are not specified here but can be looked up in the ISA reference manual of the respective processor. Operand field syntax is checked for correctness and error messages are generated in case of syntax errors.

Semantic

The assembler checks a number of semantic rules and generates error messages in case of errors. The most important rules and associated error messages are:

Illegal addressing mode: the operand field may have correct syntax but the addressing mode derived from this syntax is not available for the specified instruction

Range exceeded: Address and data constants contained in the operand fields of instructions usually have a limited legal range of signed and/or unsigned values. If a specified expression exceeds this range an error message is generated pointing at the type of address or data constant.

Forward References

The **sf20asm** assembler allows forward references. Labels and symbols can be used in expressions before they are defined. The assembler handles this with a dual pass concept that calculates defined values for all labels and symbols in the first pass and then uses these values in the second pass.

Pseudo addressing modes

To improve the readability of source code the **sf20asm** assembler supports a number of pseudo addressing modes. These are additional operand field formats that are not defined by the native instruction sets of the sf20 processors. A pseudo addressing mode omits certain components of a native addressing mode (e.g. a register) and fills in a default value for this component.

Example is the triadic addressing mode **`Rs0,Rs1,Rd`**. In many cases these instructions are used with identical second source register and destination register. A pseudo addressing mode is available where the second source register is omitted. The assembler fills in the correct code for the second source register.

One disadvantage that should be considered when using pseudo addressing modes is the fact that disassembled code, e.g. in simulation listings or debugger traces always shows the native addressing mode and operand field syntax, which with the use of a pseudo addressing mode looks different from the source code instruction. This may be confusing especially for programmers that are not very familiar with the native instruction set of the processor.

The following table lists the pseudo addressing modes supported by the **sf20asm** assembler. For the acronyms used refer to the ISA reference manuals of the sf20 processors.

Instructions	Native addressing mode	Pseudo addressing mode	Omitted component(s)	Default value(s) inserted

`addt,subf`
`addc,subc`
`mult,mlhu,mlhs`
`addsd,subsd`
`andb,iorb,xorb`
`btst,btcl,bttg`
`shlz,shru,shlf`
`shrs`
	`Rs0,Rs1,Rd`	`Rs0,Rd`	`Rs1`	`Rd`

`negt,absl,invt`
`clzr,sxbt,sxsh`
`adcf,sbcf`
`clipd,clshd,clubd`
	`Rs,Rd`	`Rd`	`Rs`	`Rd`

`shlz,shru,shlf` `shrs`	`SHC4,Rs1,Rd`	`SHC4,Rd`	`Rs1`	`Rd`

btst,btcl,bttg	**BTI4,Rs1,Rd**	**BIT4,Rd**	**Rs1**	**Rd**
ldbt,ldsh	**(DO8$_S$,An),Rd**	**(An),Rd**	**DO8$_S$**	**0**
stbt,stsh	**Rs,(DO8$_S$,An)**	**Rs,(An)**	**DO8$_S$**	**0**
brxx	**IO10$_S$,S**	**IO10$_S$**	**S**	**direction (*)**

* If the speculation flag of a conditional branch is omitted, the assembler fills in S=1 for backward branches and S=0 for forward branches.

3 Output file formats

3.1 Code file

The generated code file has a simple, text readable format and contains the code of both text and data sections as well as a list of the global symbols. Each line contains one code record which can be a text record, a data record or a symbol record. The first character of the line determines the record type and is a ‘T’ for text records, a ‘D’ for data records and a ‘S’ for symbol records. The record type character is followed by a ‘-‘ character and then by a 32-bit byte address in an 8-digit hexadecimal format. The address field is followed by a single space character as separator. From there on the format is different for text and data records on one side and symbol records on the other side.

For data and text records the next field is the record length printed as 2-digit decimal number and followed by two space characters. The remainder of the line consists of a number of byte fields equal to the record length. The byte fields are printed as 2-digit hexadecimal numbers separated with a space character.

Text records contain a 32-bit value for each 20-bit instruction code word in little endian byte order. The lower 16 bits of 20-bit opcodes are in byte 0 and 1, bits[19:16] are in bits[3:0] of byte 2. Byte 3 and bits[7:4] of byte 2 are always zero.

For symbol records the next field after the blank character following the address field is a single character symbol type specifier which is a ‘T’ for a text section label, a ‘D’ for a data section label and a ‘C’ for a constant. The symbol type specifier is followed by a single space character as separator. The last field is the symbol name printed as variable length text string.

3.2 Log file

The generated log file is intended as a debugging aid for software, simulation models and for the assembler itself. All source files that are read during an assembler run are echoed into the log file. Each line starts with a 13 character code field generated by the assembler followed by a copy of the source file line. Following the echoed source file(s) is a list of all global and local labels. At the end of the file is a statistics summery with the sizes of text and data sections in bytes and the total number of symbols.

The assembler generated field in each line of the echoed source files starts with a section marker character which is a ‘D’ for data sections and a ‘T’ for text sections. The next component is the current 16-bit instruction address or 16-bit data address printed as four hexadecimal digits. In data sections this is a byte address and in text sections it is a word address (points to a 20-bit opcode). In data sections the printed address is always aligned on a 16-bit boundary which means that the LSB is zero. It represents the 16-bit word address of the following code field.

In the data section the code field consists of an 16-bit word printed as 4-digits hex number. Bytes within 16-bit words for which no code has been generated are printed as two ‘-‘ characters. If the code generated from a source line does not fit into a single code field additional lines are printed with only an assembler generated code field and no echoed source line following. Opcodes in text sections are always full 20-bit words printed as 5-digit hex number. In data section it depends on the address counter and on the number of code bytes generated how many lines are printed. The address counter in a data section does not have to be aligned on a 16-bit boundary. The ‘—‘ printed bytes indicate the actual position of the address counter.

4 Notes

4.1 Text and data section addresses

The **sf20asm** assembler has separate address counters for the test and data sections. Both are set to zero when the assembler is invoked. The address counter of the current section is updated by **.org** directives, **.p2align** directives, by constant data **(.byte**, **.short**, **.long**, **.ascii** and **.asciz** directives) and by instructions. The other address counter (of the non current section) is not affected.

Code is always generated for the current section. If instruction lines occur in the source code within a data section then the code is written into the data section with no warning. If constant data lines occur within a text section the code is written into the text section with no warning. While this can be useful in some rare cases it is not what programmers want in most cases. So care must be taken to make sure all code is generated for the intended section.

Data addresses are specified as 16-bit byte addresses within a 64kBytes address space. Instruction addresses are 20-bit word addresses within a 64k x 20-bit address space. Because they select 20-bit words and not bytes they cannot be misaligned.

Data addresses don't need to be aligned on the addressed data type and therefore are not checked for alignment by the assembler. However support for misaligned 16-bit, accesses is not implemented in the sf20 processor cores. If required it must be implemented in the bus glue logic. Programmers must either make sure that data addresses are always aligned on the addressed data word size or that the target system supports misaligned accesses.

4.2 Known Problems

Some operand field syntax errors cause the assembler to hang up after printing an error message. The assembler must then be terminated with a Ctr-C character in the shell window.

When using multiple **.org** directives for the same section care must be taken that the generated code address windows do not overlap. The assembler does not check if code for the same address location is generated multiple times.