

Assembler, Loader, and Simulator Specification

CPSC / ECE 3710, Fall 2006

1 Introduction

You are required to construct an assembler that turns symbolic instructions written in the *assembly language* (Section 1.1) into machine code. The machine code must be generated in two different formats: the *listing* format (Section 1.1.3) that is meant to be human readable, and the *hex* format that matches what the GXSLOAD program requires (Section 1.1.4). The simulator may be integrated with the assembler itself (need not be a separate tool). It must print the CPU state before and after each instruction. See Section 1.2 for details.

1.1 Assembly Language

1.1.1 General Principles

An assembly language is designed to help the programmer write far more readable – and hence far more reliable – code. Below is an example assembly program and the corresponding machine code generated from it (shown in Section 1.1.3). I’m sure you’ll agree that writing the assembly input is far easier than writing the machine code. The comments on the right-hand side tell us what the code does. Further things pertaining to this program appear in later sections.

```
; An example assembly program for MCR16-bl
; Provided for the sake of illustration

        .org 0xBFFF      ; Defines start of code / data block
L1: .word 0xA0B0        ; The word A0B0 sits here
L2: .word 0xC0D0        ; Here sits the data word C0D0
L3: .word 0x0000        ; and finally 0000

        .org 0x0080      ; Now move to a different origin
L7: .equ *
      movi L7, r1      ; Here, we move L7 (a byte) into r1

        .org 0x0800      ; Move to another origin
L4: .move_word L2, r2    ; Move word L2 into r2 through a series of actual
                        ; machine instructions
L8: add  r1, r2        ; add r1 and r2
      bcs  L6          ; Branch on carry set to L6 (forward branch)
      movi 13, r3      ; move immediate 13 into r3
      movi -13, r3     ; Now move negative 13 (in 2's complement)
      movi 0xF3, r3    ; This also moves negative 13 into r3 (2's complement)

L5: .equ L4            ; equate L5 to L4
      stor r2, r3      ; Store r2 into mem[r3]
      lshi -3, r4      ; This is essentially right-shift of 3 places
      lshi 0x1F, r4    ; This would be directly interpreted as rsh of 15 places
```

```

L6: .equ *           ; Let L6 be the same as the present PC value
    beq L4           ; Branch, generating a negative offset
    .end             ; This serves as a clean end-of-program marker

```

1.1.2 What Must the Thin-slice Assembler Include?

The thin-slice assembler must include the thin-slice instructions and also the assembler directives of `.org`, `.end`, as well as `.word`. It need not support the `.equ` directive, nor the `.move_word` directive.

1.1.3 Listing Output from the Assembler

The example assembly program generates “code” through the following steps:

- The `.org` and the `.word` that follows allows the labels L1 through L3 to be given values as follows:
 - L1 : 0xBFFF
 - L2 : 0xC000
 - L3 : 0xC001
- The assembler’s “focus” shifts to the origin 0x0080
- It sets L7 to 0x0080
- Here, it confronts the task of generating generates the code for `movi`.
- The assembler verifies that L7 indeed fits within a byte. Should L7 be declared forward, the assembler will have to wait for a second pass before it can generate code (and by now if L7 is found to exceed a byte, the assembler must throw an error).

In this case, we generate the code for the instruction `movi L7, r1`

- Now the attention shifts to `ORG 0x0800`
- The directive `.move_word` causes the assembler to generate an “lui, ori” sequence of instructions.
- Negative values are allowed only if supplied in decimal (e.g., -13). Otherwise, the hex value in 2’s complement must be supplied (e.g., 0xF3)
- `lshi` takes sign-magnitude form inputs. Hence 0x1F is a negative shift of 15 places.
- Sign-magnitude is notorious for two representations for zero (e.g., 0x10 and 0x00). Only the latter is required to be supported.

With the above detail, the assembly proceeds as follows.

In the first pass, the symbol table is created and populated as follows:

```

L1 = BFFF
L2 = C000
L3 = C001
L7 = 0080
L4 = 0800
L8 = 0802 -- move_word generates two instructions
L5 = 0800
L6 = 080A

```

After the first pass, the code can be emitted as shown on the left-hand side of the listing:

```

                .org 0xBFFF
BFFF: A0B0   L1: .word 0xA0B0
C000: C0D0   L2: .word 0xC0D0
C001: 0000   L3: .word 0x0000

                .org 0x0080
                L7: .equ *
0080: D180           movi L7, r1

                .org 0x0800
0800: F2C0   L4: .move_word L2, r2 -- expands into LUI, ORI
0801: 2200
0802: 0251   L8: add  r1, r2
0803: C207           bcs  L6           -- displacement is +7
0804: D30D           movi 13, r3
0805: D3F3           movi -13, r3
0806: D3F3           movi 0xF3, r3

                L5: .equ L4
0807: 4243           stor r2, r3
0808: 8413           lshi -3, r4
0809: 841F           lshi 0x1F, r4

                L6: .equ *
080A: C0F6           beq  L4           -- branch offset is -10, which is F6 in 2's-C
                .end

```

1.1.4 Hex Output from the Assembler

The hex MCS format is now described briefly. (Details will follow later.)

```
: <nbytes> <startaddr> <recordtype> <by> <by> ... <by> <checksum>
```

Example (I add _ for clarity - it is NOT allowed below)

```
:10_0170_00_70_71_72_73_74_75_76_77_78_79_7A_7B_7C_7D_7E_7F_07
```

00 = data record type. 01 will be end-of-file record type. Example.

```
:00_0000_01_FF
```

Here, checksum is the sum of all the bytes in binary, chopped to 8 bits, and 2's complemented.

10 + 01 + 70 + 00 + + 7E + 7F, truncated to 8 bits and 2's complemented == 07

Similarly for the end-of-file record

For loading beyond 64K we get the address extension record. Example:

```
:10_0170_00_70_71_72_73_74_75_76_77_78_79_7A_7B_7C_7D_7E_7F_07  
:02_0000_04_0001_F9
```

```
; 0001 is the higher address word  
; for the following records
```

```
:10_0170_00_70_71_72_73_74_75_76_77_78_79_7A_7B_7C_7D_7E_7F_07
```

```
; this record gets loaded at 0x00010170.
```

The MSB stays in effect till the next address extension record.

Exercise: Obtain the MCS records for the assembler listing given in Section 1.1.3.

1.2 Simulator Description

The simulator is totally up to you to develop and endow with features. Minimally, it must read the assembled output (along with data initializations) and then step thru the sequence of instructions. After each instructions, print the relevant quantities (e.g., whatever changed).

1.3 More Formal Assembler Syntax

To recap, the assembler syntax is as follows:

- A comment begins with a “;” and ends at the end of a line
- The syntax is case SENSITIVE.
- Opcodes are required to be in lower case
- Register names are also required to be in lower-case
- Labels are required to be in upper case as are hex numbers
- Spaces and tabs do not count, but end-of-line counts (is significant)

- The syntax only covers the required baseline architecture.
- Extend suitably but keeping same conventions for other instructions

The Backus-Naur Form (BNF) syntax for our CR-16 assembly language is given below.

```

<assembly program> ::= <line>*

<line> ::= <equ> | <org> | <word> | <moveword> | <instruction>

<equ> ::= <label> <equkwd> <number-or-star>

<equkwd> ::= .equ

<number-or-star> ::= <number> | * ; '*' is used to denote the current PC.
                        ; 'LABEL: equ *' is quite commonly used

<org> ::= <orgkwd> <number>

<orgkwd> ::= .org

<word> ::= [ <label> ] <wordkwd> <number> -- optional label shown by [...]

<wordkwd> ::= .word

<moveword> ::= [ <label> ] .move_word <operand>, <regopnd>

<label> ::= <labelstring> :
                ; suggestion: use block capitals or capitalized for labels
                ; suggestion: keep distinct from opcode mnemonics etc

<labelstring> ::= [A-Z] [A-Z|0-9]*

<number> ::= <hex> | <bin> | <dec>

<hex> ::= <hexheader> <hexstring>
<bin> ::= <binheader> <binstring>
<dec> ::= <decheader> <decstring>

<hexheader> ::= 0x -- signs not allowed
<hexstring> ::= [0-9|A-F] [0-9|A-F]*

<binheader> ::= 0b -- signs not allowed
<binstring> ::= [0|1] [0|1]*

```

```

<decheader> ::= [ + | - ]      -- optional sign
<decstring> ::= [0-9] [0-9]*

<instruction> ::= [ <label> ] <opcode> <operands>

<opcode> ::=
add | addi | sub | subi | cmp | cmpi | and | andi | or      | ori | xor | xori |
mov | movi | lsh | lshi | lui | load | stor| jal

-- the branch and jump mmenonics come from Table-1

beq | bne | bge | bcs | bcc | bhi | bls | blo | bhs | bgt | ble | bfs |
bfc | blt | buc

jeq | jne | jge | jcs | jcc | jhi | jls | jlo | jhs | jgt | jle | jfs |
jfc | jlt | juc

<operands> ::= <operand>* -- depends on instruction; if more than one,
                        -- separate by comma

<operand> ::= <regopnd> | <immopnd>

<regopnd> ::= r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7
            ; well, we can have upto 16 regs
            ; you are required to implement r0 thru r7

            ; you may also implement r8, r9, rA, rB, rC, rD, rE, and rF
            ; thereby implementing all 16
            ; ... or do something else with these reg field designators
            ; ... e.g. to ‘‘trigger’’ a graphics space write, it’s quite
            ; ... possible that you can store into ‘‘register r8’’ a value
            ; which provides the base address for further graphics transfers.

<immopnd> ::= <labelstring>
            ; must be .word or .equ defined
            | <number>
            ; must be of right size

```

Other conventions:

- Operands can be numbers or labels
- Negative or positive signs can be attached only to decimals
- Else provide in 2’s-complement binary or hex

- All instruction operands (whether provided thru numbers or labels) **must fit** the field width available – else, assembler must generate error message.

2 Uploading the DRAM contents into a file

An update: The TAs suggested as simpler method for uploading:

- Use GXSLOAD, and enter the low and high address to dump (e.g., 0-0xFFFF)
- Then drag the folder icon into a Windows folder

My older instructions:

- Uploading : The command was found with great difficulty at web location <http://www.cs.ualberta.ca/~amaral/courses/329/labs/misc.html> as other locations gave erroneous syntax for command. The syntax itself is `xsload -fpga pet.bit -ram dump.hex -upload 0 127`
BTW <http://www.cs.ualberta.ca/~amaral/courses/329/labs> has lots of useful stuff.
 - Give it ANY bit file (for it to know what chip we are talking to).
 - Then give the start (0) and length (127) ; must be even number of bytes
 - File dump.hex will be created.
 - FPGA program will be destroyed!
- Downloading : GXSLOAD

Exercise: Write the following programs and hand-assemble them:

1. The program in the CPU manual – in Section 3
2. Now here are the details of the “find the largest” program you have to write.

Given three numbers located in locations L1, L2, and L3, write a program to find which of the numbers is the largest. Write the address of this location (say L2) into location L4.

Hint: Consider the specific declarations that begin as follows

```

.org 0xF000 ; for example
L1: .word 0xA0A0 ; first number
L2: .word 0xAAA0A ; second number
L3: .word 0x0A0A ; third number
L4: .word 0x0000 ; will be clobbered by L2 = 0xF001 when program finishes
ENTER: <here goes your code>
...
.end

```