ECE 4440
ADVANCED DIGITAL DESIGN

Assembly Programming 101 – v2.0

AN OVERVIEW OF ASSEMBLY PROGRAMMING DESIGN FLOW USING THE ECE4435/4440 16-BIT RISC CORE AND THE FPGA PROTOTYPING PLATFORM

~ADAPTED FROM WORK PREVIOUSLY COMPLETED BY WES DUGAN

Mircea Stan
MIRCEA@VIRGINIA.EDU

John Lach
JLACH@VIRGINIA.EDU

Mark Hanson
MAH6S@VIRGINIA.EDU

Arun Thomas
ARUN@VIRGINIA.EDU

Assembly Programming 101

INSTRUCTION SET INTRODUCTION

ASSEMBLY PROGRAMMING:

Assembly programming using the Simple Assembler (sasm) is a very straight forward process. Producing the assembly source is accomplished by using a text editor. The source can be saved as any file name, however, we recommend using an ".asm" extension to distinguish your files.

An assembly program for the Simple Assembler consists of several structures: labels, instructions, operands, values, and data. Labels are essentially line markers in the assembly code. The assembler parses the code for these labels and builds a table that relates a label to its associated memory location or line number. This way, the programmer is not burdened with the task of tracking memory locations of sub routines. Therefore, a conditional or unconditional branch may just reference a label to load the PC with the memory address of the subroutine's first instruction. Labels may take any name, as long as they do not interfere with the assembler's primitives. Labels must be followed immediately by a colon and may or may not have an instruction on the same line. See the assembly source at right for more examples of using labels.

```
# +---------------------------------+
# | Factorial Assembly Example v2.0  |
# | Jan. 21, 2013                    |
# |---------------------------------|
# | ECE 4440 Advanced Digital Design |
# | University of Virginia           |
# +---------------------------------+
#Calculates n!, where n is the value
at the DATA label
#When n=5, final R0(hex)=0078 and
LED=8

RESET:  DATA    2
ISR:    DATA    0
STRT:   LIL     R9 <DATA
        LIH     R9 >DATA
        LD      R0 R9 0
        LIL     R10 0
        LIH     R10 0
        LIL     R11 1
        LIH     R11 0
BASE:   MOV     R15 R0
        SUB     R15 R10
        BZ      ZERO
        MOV     R2 R0
        OR      R2 R10
        BR      LOOP
ZERO:   MOV     R0 R11
        OR      R0 R10
        BR      DONE
LOOP:   SUB     R2 R11
        MOV     R1 R0
        OR      R1 R10
        MOV     R3 R2
        OR      R3 R10
        SUB     R3 R11
        BZ      DONE
        BR      MULT
MULT:   ADD     R0 R1
        SUB     R3 R11
        MOV     R15 R3
        SUB     R15 R11
        BZ      LOOP
        BR      MULT
DONE:   BR      DONE
DATA:   DATA    0x0005
```

The second structure is the instruction. These instructions correspond directly to the instructions that have been defined in the ISA Specification. The assembler merely translates this instruction into an opcode that corresponds to that instruction (i.e. MOV -> 011).

The next structure is the operand. Depending on the instruction, one, two, or three operands may be required. In a typical instruction such as an ADD (i.e ADD R0 R1), the first operand corresponds to the destination register (i.e. R0) and the next operand corresponds to the source operand (i.e. R1). Operands that are already recognized as assembler primitives include (R0 – R15). These operands are translated into the corresponding register addresses (0000 –

1111). The operands are then appended to the opcode to form a complete instruction.

Values or offsets are the next structures that are commonly encountered during the course of an assembly program. Some instructions, such as load immediate low (LIL) use the least significant 8-bits of an instruction word to encode a value. Depending on the instruction, the value may be used in different ways. For example, a LIL instruction will use the value to load the low byte of a register, while a BR will use the value as an offset to add to the incremented program counter. The Simple Assembler will recognize decimal representations of values. For example, the instruction "LIL R0 10" will place the binary equivalent of the decimal value 10 (00001010) into the low byte of register zero. Another feature that has been added to the assembler is the ability to load the low or high byte (least significant or most significant respectively) of a memory address associated with a label. In the above assembly source, the third and fourth instructions display how to load R9 with the address of the second instruction. Please note that the "greater than" or "less than" operator must be used immediately before the label. In other words, no blank space may exist between the operator and the label. Another feature of the assembler is the ability to branch to a label. In the instruction "BR DONE", the label DONE corresponds to the memory address 32.

The final structure that will be encountered during the course of an assembly program is the DATA structure. This assembler-recognized primitive will load the hexadecimal 2 byte word into the program's corresponding memory location. In the program above, the memory location 0x0000 is loaded with the value 2 which is the start of the program. When the processor is reset the PC will be loaded by the value 2 and the execution of the program will proceed from there. Also, the memory location 33 is loaded with the value 5 which is used by the program as input. Note that the DATA primitive may be preceded by a label. Also make sure that the program's execution does not perceive a DATA's memory location as an instruction. An easy way to prevent this occurance is to jump over these DATA locations or put all data at the end of the program.

The assembler also provides a means of commenting your assembly source. The pound directive (#) will allow you to add comments to the source code. Any text on a line following the pound character will be ignored by the assembler. Any blank lines will also be ignored by the assembler. A few other imporant notes about this assembler are:

- sasm_mod is case sensitive
- All directives and labels must have white-space before and after them.
- sasm_mod has VERY poor error checking. You have to make sure you are getting what you would expect. A misspelled instruction or label will result in incorrect code.

ASSEMBLING PROGRAMS:

The assembly program (sasm_mod.exe) can be downloaded from the Toolkit webpage and executed from a DOS command line on any PC. You must also download 'isafile_mod' from the Toolkit webpage into the same directory as the executable.

You are now ready to assemble files. The input file can be named anything that you want, but we recommend the convention of appending '.asm' on the end of the input file-name so you know that it is an assembly file. The output from sasm is sent to stdout. It can easily be redirected to a file, which is what you will want to do most of the time. The output file must be a single word with the file extension ".hex". If you do not follow this convention, your design will not load onto the FPGA platform. The command to assemble a file will look similar to the following command:

```
sasm_mod foo.asm > foo.hex
```

The file 'foo.hex' will be in a format that can be used without modification on the FPGA platform. The resultant output is in the Intel Hexadecimal format. Any modifications you make to the program after assembly will not be reflected in the file's checksum information and therefore will not load onto the FPGA platform. Any changes made to the assembly source requires you to reassemble that source.

Another feature of the sasm assembler is that it can output a file that can be easily imported into your simulation memory block for simulations. Therefore, you will no longer have to code your programs in binary for your simulations. This will enable you to do more complex simulations and to try the testbenches we provide. Simply use the -b option after sasm and write to a ".bin" file:

```
Sasm_mod –b foo.asm > foo.bin
```

or the –t option and write to a table format:

```
sasm_mod –t foo.asm > foo.tab
```

INTEL HEX FORMAT:

The output of the assembler is in Intel Hexadecimal format. This format is a commonly used format that is used for producing ROM files of all types. We recommend checking the output of the program to ensure that your source has produced the result you wish for execution onto the processor platoform. To check your output will require that you know how to decipher the hex output. First off, as its name implies, the hex format is written as a strings of hexadecimal characters. The specifics of the format are covered in the following document: http://www.interlog.com/~speff/usefulinfo/Hexfrmt.pdf. Here are a few details about what you need to know to debug your output. An example hex file is included below:

```
:10008000AF5F67F0602703E0322CFA92007780C361
:1000900089001C6B7EA7CA9200FE10D2AA00477D81
:0B00A00080FA92006F3600C3A00076CB
:00000001FF
```

- The first character (:) indicates the start of a record.
- The next two characters indicate the record length (10h in this case).
- The next four characters give the load address (0080h in this case).
- The next two characters indicate the record type (see below).
- Then we have the actual data.
- The last two characters are a checksum (sum of all bytes + checksum = 00).
- The last line of the file is special, and will always look like that above.

Record types:

- 00 - Data record
- 01 - End of file record
- 02 - Extended segment address record
- 03 - Start segment address record
- 04 - Extended linear address record
- 05 - Start linear address record