RAT Assignment #1

Introduction to Architecture and Assembly Language Programming (and a dash of Reverse Engineering)

Learning Objectives:

- ✓ To understand the basics of assembly language programming.
- ✓ To understand the architecture required to support simple assembly language operations.
- ✓ To understand how an assembly language program is assembled and converted into a binary format (machine code) which the processor can execute.
- ✓ To understand how to use an assembly language simulator to analyze an assembly language program.

General Notes:

This lab demonstrates the concept of reverse engineering as part of the presentation of the architecture of the RAT processor and basic assembly language programming. Reverse engineering has its place in computer system development as companies vie for market share. The story at the link below about how Compaq reverse engineered IBM's PC BIOS is a noteworthy footnote in computer engineering history. (https://goo.gl/dKwCrX)

The recent A&E series, *Halt and Catch Fire*, which is available on NetFlix, does a entertaining job of telling the story of reverse engineering the PC Bios in the early 80's. Note that some artistic license is taken with the story.

Assignment: This procedures portion of this experiment has two different procedures.

Procedure – Part 1:

a.) Figure 1 shows Program #1, an example RAT assembly language program; write and assemble this program in the RAT Simulator. Be sure to include a blank line after the last line of your code.

```
.EQU LED_PORT = 0x10                ; port for output
.DSEG
.ORG          0x00

.CSEG
.ORG          0x40                  ; data starts here

main_loop:    MOV     R10,0x05
              MOV     R11,0x64
              ADD     R10,R11
              ADD     R10,0x14
              MOV     R20,R10
              OUT     R20,LED_PORT
              BRN     main_loop       ; endless loop
```

Figure 1: Test Program #1.

b.) Step through the program with the RAT simulator and analyze Program #1 to determine the contents of various registers as this program executes. Complete Table 1 by including the following information after the simulator executes the listed instruction. The completed table should have one line for each instruction in the program. Include a copy of this completed table with your lab report. Note that this table is intended to capture things that change as the code execute and is intended as a walk-through to help you understand program operation during execution.

PC	Instruction	Destination Register Value	C=?	Z=?	Output(ADDR) (OUT instr)
	`MOV R10,0x05`	`R10 = 0x05`			
	`MOV R11,0x64`	`R10 = 0x05`			

Table 1: Program #1 analysis table.

c.) Write an assembly language program that continually performs the following operations: inputs a byte from the port at address 0xCC, stores it in R30, exclusive ORs the value with 0xDE, and then outputs the ORed value to address 0x5A. Your assembly code should look like the style file example in the RAT Assembler manual. This means it should look perfect while including comments, proper naming and spacing conventions, and include a nice header at the top and other banners to delineate sections of your code. Include a printout of this program with your lab report.

Procedure – Part 2:

a.) Reverse engineer the excerpts from the prog_rom.vhd file shown below to determine the assembly instructions implemented by this file. Note that you may disregard any other lines from this VHDL file, as they are all zeros. Also, note that because the code is located in INIT_04 and fills starting from the least significant byte, the address of the first instruction is at 0x40.

See the example of reverse engineering a RAT prog_rom.vhd file at the end of this experiment description. Complete Table 2 for your reverse engineered program (add as many rows as you need) and include the table in your lab report.

```
attribute INIT_04 of ram_1024_x_18 : label is
"0000000000000000000000000000000000820B3EEA1EFA7F057EFA9D017DFC";

attribute INITP_00 of ram_1024_x_18 : label is
"00000000000000000000000000000FB0000000000000000000000000000000";
```

2-bit pairs from INITP_00	2 bytes from INIT_04	Concatenation of left two columns of this table	Assembly Instruction	ProgROM Mem Loc.

Table 2: Sample format for machine code reverse engineering.

b.) Enter the reverse engineered program in the RAT simulator. Make sure this program has proper programming style and include an listing of this program with your lab report. Step through the program with the RAT simulator and perform an analysis of your reverse engineered program to determine the contents of various registers as the program executes and complete Table 3 for each line of code in the program. Include this completed table with your lab report.

PC	Instruction	Destination Reg Value	C=?	Z=?	Output(ADDR) (OUT instr)
	`MOV R10,0x05`	`R10 = 0x05`			
	`MOV R11,0x64`	`R10 = 0x05`			

Table 3: Program #2 analysis table.

Specific Deliverables: *(This assignment will not follow the normal RAT Assignment Report Format)*

1. Procedures: Part 1:
   a. Completed Table 1 for the program.
   b. Typed and properly formatted assembly code for step (c).

2. Procedures: Part 2:
   a. Completed table for reverse-engineered assembly language program.
   b. Typed assembly code for the reverse-engineered assembly language program.
   c. Completed Table 3 for the program showing processor state.

Questions:

1. Briefly describe why the prog_rom.vhd is tedious to interpret.

2. Why is it necessary to have assembly language programming when there are much more elegant languages such as C, Java, etc. available?

3. Describe a system other than the examples given in the link above in which reverse engineering could or has led to a significant business advantage?

Example of Reverse Engineering

Please note that this program will not assemble on the latest version of the RAT assembler as it starts at address 0x00. The assembler requires address 0x00 to remain unused. The .ORG value in this program would need to be moved to a higher value, e.g. 0x10, for this program to assemble without error. All other aspects of this example are correct.

1.) Consider the following RAT assembly language program:

```
.EQU LED_PORT = 0x10                ; port for output
.DSEG
.ORG          0x00
.CSEG
.ORG          0x00                  ; data starts here

main_loop:    MOV     R10,0x05
              MOV     R11,0x64
              ADD     R10,R11
              ADD     R10,0x14
              MOV     R20,R10
              OUT     R20,LED_PORT
              BRN     main loop      ; endless loop
```

2.) This program produces the following lines in the prog_rom.vhd file. To reverse engineering the hex values listed in the associated prog_rom.vhd file, extract the contents of the two populated rows from prog_rom.vhd. The underlining shows the bytes of interest.

```
attribute INIT_00 of ram_1024_x_18 : label is
"000000000000000000000000000000000000008000541054518A142A586B646A05";

attribute INITP_00 of ram_1024_x_18 : label is
"000000000000000000000000000000000000000000000000000000000000000C8F";
```

3.) Extract the lower 28 bytes of INIT_00 and divide them into two-byte pairs. Notice that there are seven 2-byte pairs which are non-zero and that our original assembly program had seven instructions:

Two-byte pair	equivalent program address
0x6A05	0
0x6B64	1
0x2A58	2
0x8A14	3
0x5451	4
0x5410	5
0x8000	6
0x0000	7

4.) Extract the four non-zero bytes of INITP_00, break into binary equivalent values, and then break these into 2-bit pairs. Note that you will only want the lower seven 2-bit pairs as the original assembly program had seven instructions:

```
0C8F = 0000        1100        1000        1111
     = 00 00       11 00       10 00       11 11
     = 0x00 0x00   0x03 0x00   0x02 0x00    0x03 0x03
```

5.) Concatenate each 2-bit pair (MSBs) with each 2-byte pair from step #3 to create the 18-bit instruction. This 18-bit field is the machine code representation of one assembly language instruction. For example, the right most 2-bit pair of 11 combines with the right-most 2-byte pair of 6A05 to create the 18-bit machine code for the equivalent assembly instruction. You can now recreate the assembly instructions from the 18-bit binary equivalent of each instruction by comparing the opcodes and field codes of this instruction to the instruction format in the RAT Assembler Manual.

2-bit pairs from INITP_00	2 bytes from INIT_00	Concatenation of left two columns of this table	Assembly Instruction	ProgRom Mem Loc.
11	0x6A05	11 0110 1010 0000 0101	MOV R10,0X05	0
11	0x6B64	11 0110 1011 0110 0100	MOV R11,0X64	1
00	0x2A58	00 0010 1010 0101 1000	ADD R10,R11	2
10	0x8A14	10 1000 1010 0001 0100	ADD R10,0X14	3
00	0x5451	00 0101 0100 0101 0001	MOV R20,R10	4
11	0x5410	11 0101 0100 0001 0000	OUT R20,LED_PORT	5
00	0x8000	00 1000 0000 0000 0000	BRN main_loop	6
00	0x0000	00 0000 0000 0000 0000	NOP	7

Recap of INIT and INITP Organization:

The key to interpreting locations in both the INIT and INITP sections are the characters per instruction or instructions per character ratios. Regardless of whether INIT or INITP, each row contains 64 HEX characters.

When looking at INIT, the low 16-bits of each instruction can be represented by four hex characters (four bits each), so there is a 4 character to 1 instruction ration. This means that each row of INIT represents 16 instructions (64 hex characters / 4 hex characters per instruction = 16 hex characters). The other thing to note regarding the INIT section is that if you start at zero and count to 16, you will have counted from 0x00 to 0x0F. This means that the second row in INIT represents instructions that start at .ORG 0x10 (16 decimal). Try it in the assembler - set .CSEG to 0x10 and see where your code lands. It will start in INIT_01.

When looking at INITP, note that it is only used to represent the upper two bits of each instruction. This means that each hex value in INITP represents two instructions (2 instructions * 2 bits/instruction = 4 bits), giving a two instructions per hex character ratio. Every row in INIT, which represents 16 instructions, is matched with 8 characters in INITP. Try this in the assembler - set .CSEG to 0x10 and see where the bytes in INITP land. The bytes will be eight characters over in the INITP_00 row.