2

Hardware Architecture and Data Manipulation

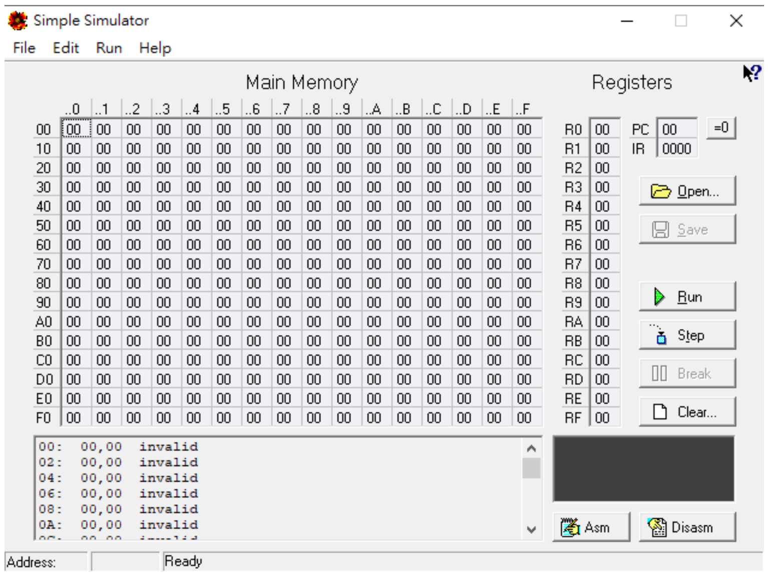
William W.-Y. Hsu
Department of Computer Science and Engineering
Department of Environmental Biology and Fisheries Science
National Taiwan Ocean University

CONTENTS

2.1 Introduction

The author of our textbook has designed a simple machine language for a theoretical computer, which uses the RISC instruction set architecture, discussed in Chapter 2 and Appendix C. This is called a simulator, which mimics the way a computer runs machine language programs. This simulator is very similar to modern computers as it has a *program counter* register, *instruction register*, 16 *general purpose* registers, and 256 bytes of main memory. It also contains an arithmetic & logic unit (ALU) which can perform `ADD`, `AND`, `OR`, `XOR`, and `ROR` operations. The memory of this virtual computer follows the von-neumann architecture, which can store both the machine program and data.

The program that we will be using in this laboratory is a Windows program so you can execute it on any PC. Moreover, we also provide a C source code version that you can compile it for any other target platforms. Upon execution, you will see the GUI of the simulator as shown in Figure 2.1.



	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

R0	00
R1	00
R2	00
R3	00
R4	00
R5	00
R6	00
R7	00
R8	00
R9	00
RA	00
RB	00
RC	00
RD	00
RE	00
RF	00

PC 00
IR 0000

```
00:  00,00  invalid
02:  00,00  invalid
04:  00,00  invalid
06:  00,00  invalid
08:  00,00  invalid
0A:  00,00  invalid
```

FIGURE 2.1
The virtual architecture simulator GUI.

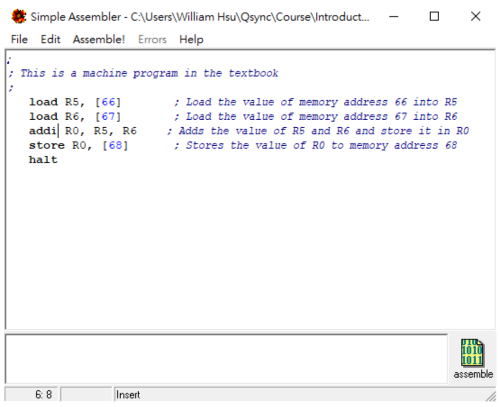
2.2 Simulators

Programs, including assembly language, are for human to understand. A CPU can only execute the instruction code of a machine language, which is rather difficult for a human to write directly. This simulator provides a interface for loading assembly languages and convert it into machine language for the virtual CPU to run. The `Open` button in Figure 2.1 can load an assembly program, usually stored as a `*.asm` file.

By using the `Open` button, we can load an assembly program. Suppose we load the `textbook.asm` file, with the contents as follows:

```
; This is a machine program in the textbook
  load R5, [0x6C]   ; Load the value of memory address 0x6C into R5
  load R6, [0x6D]   ; Load the value of memory address 0x6D into R6
  addi R0, R5, R6   ; Adds the value of R5 and R6 and store it in R0
  store R0, [0x6E]  ; Stores the value of R0 to memory address 0x6E
  halt
```

we get an editor showing the following assembly code in Figure 2.2.

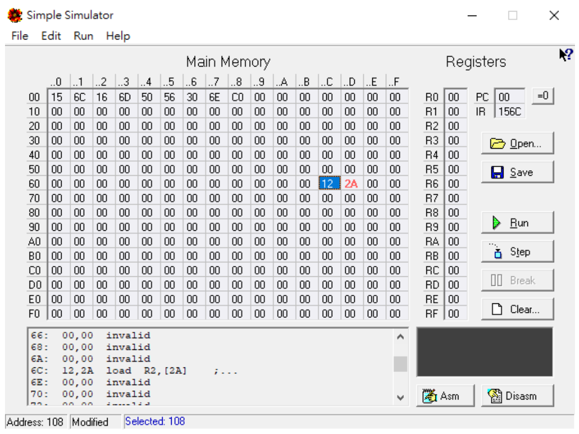

Simple Assembler - C:\Users\William Hsu\Qsync\Course\Introduct...
File Edit Assemble! Errors Help
assemble
6: 8
Insert

```
;
; This is a machine program in the textbook
;
   load R5, [66]        ; Load the value of memory address 66 into R5
   load R6, [67]        ; Load the value of memory address 67 into R6
   addi R0, R5, R6     ; Adds the value of R5 and R6 and store it in R0
   store R0, [68]       ; Stores the value of R0 to memory address 68
   halt
```

FIGURE 2.2
Loading an assembly file into the simulator.

After the program has been loaded into the editor, we can push the `assemble` button to convert the assembly language into machine codes. Shown in Figure 2.3, the machine code has been stored at memory 0x00, and the *program counter* register (PC) is set to 00, which is the beginning of the program. You can verify to see if the assembly code converts to the machine code as listed in the PPT slides of the main course (or Figure 2.2 in the textbook). The machine instruction is stored starting at address 0x00, `156C166D5056306EC0`.

Before we start to execute the program, we need to fill in values for address 0x6C and 0x6D. For example, we can put in the value 0x12 and 0x2A respectively. The simulator has a button `Run`



Main Memory

	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
00	15	6C	16	6D	50	56	30	6E	C0	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	12	2A	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Registers

R0	00
R1	00
R2	00
R3	00
R4	00
R5	00
R6	00
R7	00
R8	00
R9	00
RA	00
RB	00
RC	00
RD	00
RE	00
RF	00

PC 00 =0
IR 156C

```
66:   00,00   invalid
68:   00,00   invalid
6A:   00,00   invalid
6C:   12,2A   load   R2,[2A]     ;...
6E:   00,00   invalid
70:   00,00   invalid
```

Address: 108 Modified Selected: 108

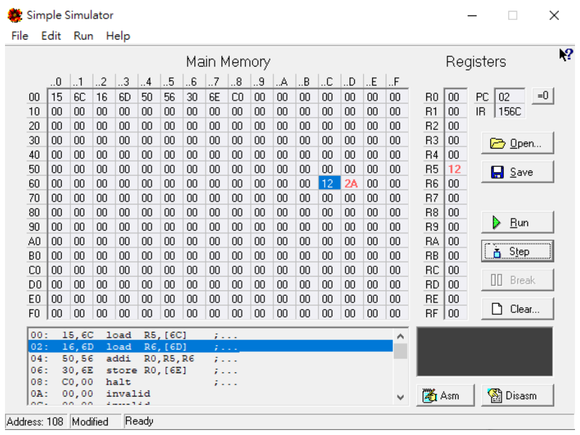
FIGURE 2.3
Converting the assembly file into machine code.

which executes the program from the beginning to the end, and a `Step` functionality to run the program line by line. We will use the `step` button to demonstrate. Starting from PC=0x00, after we press `step` once, we get the results shown in Figure 2.4. The current PC has been increased to PC=0x02, pointing to the next instruction to be fetched, and the *instruction register* IR=0x156C, meaning the instruction that has just been executed. We see that after executing 0x156C, the contents of register R5=0x12, which is copied from the memory address 0x6C.

The result of second step, executing the instruction IR=0x166D is shown in Figure 2.5. We can see that the value of register R6=0x2A is now copied from the memory address 0x6D.

The third instruction 0x5056, conducts integer addition of the registers R5 and R6 and stores the result in R0=0x3C (see Figure 2.6).

The fourth instruction stores the content of register R0 back into memory address 0x6E (see Figure 2.7), and the last instruction 0xC000 end the program (see Figure 2.8).



Main Memory

	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
00	15	6C	16	6D	50	56	30	6E	C0	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	12	2A	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Registers

R0	00
R1	00
R2	00
R3	00
R4	00
R5	12
R6	00
R7	00
R8	00
R9	00
RA	00
RB	00
RC	00
RD	00
RE	00
RF	00

PC 02
IR 156C

```
00:  15,6C  load  R5,[6C]     ;...
02:  16,6D  load  R6,[6D]     ;...
04:  50,56  addi  R0,R5,R6    ;...
06:  30,6E  store R0,[6E]     ;...
08:  C0,00  halt              ;...
0A:  00,00  invalid
```

Address: 108 Modified Ready

FIGURE 2.4
Executing 0x156C.

2.3 Complete instruction set

The complete instruction set for this virtual architecture is listed in Table.

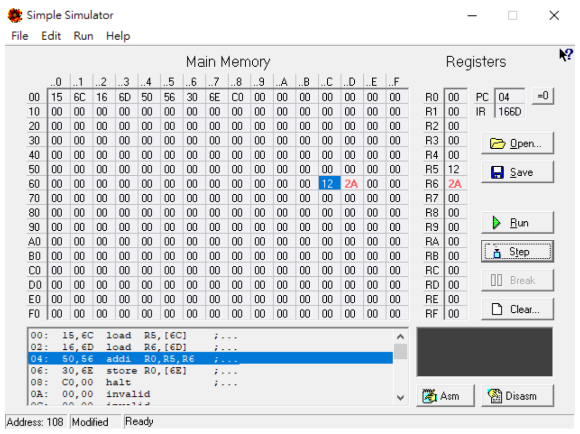
2.4 Simple instruction combinations

The RISC instruction set can be combined to generate complex function. We will demonstrate some of them in this section.

2.4.1 Immediate, direct, and indirect addressing

There are three modes of addressing that can be used in this simulator. Consider the following instructions:

```
load R1, 0x64
load R1, [0x64]
load R1, [R5]
```



Main Memory

	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
00	15	6C	16	6D	50	56	30	6E	C0	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	12	2A	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Registers

R0	00
R1	00
R2	00
R3	00
R4	00
R5	12
R6	2A
R7	00
R8	00
R9	00
RA	00
RB	00
RC	00
RD	00
RE	00
RF	00

PC	04
IR	166D

```
00:  15,6C  load  R5,[6C]     ;...
02:  16,6D  load  R6,[6D]     ;...
04:  50,56  addi  R0,R5,R6    ;...
06:  30,6E  store R0,[6E]     ;...
08:  C0,00  halt              ;...
0A:  00,00  invalid
```

Address: 108 | Modified | Ready

FIGURE 2.5
Executing 0x156D.

The first instruction is *immediate* instructions, which loads the value 0x64 into register R1. The second instruction is direct addressing, which loads the value stored in memory address 0x64 into register R1. The last instruction is indirect addressing, similar to pointer concept in C. It loads the value of the memory address stored in register R5 into register R1. These instructions are similar to the following C code:

```
a = 100;
a = b;      // where b has been assigned the value 100 before
a = *b;     // where b is now register R5
```

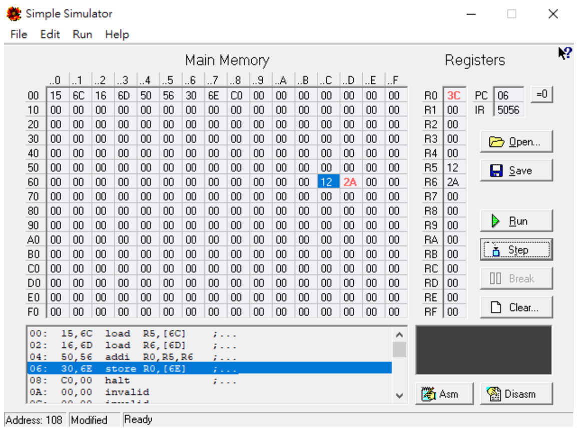
2.4.2 Branching

Consider the following C code:

```
if( a == 10 )   // Conditional branch test
  ;             // then construct
else
  ;             // else construct
```

This code can be implemented using the jmpEQ instruction.

```
; A branching structure
      load R0, 0x0A        ; Loads the value 10 into R0
                           ; Let R1 be some value
      jmpEQ R1=R0, A       ; Jumps to label A if R1 = R0
```



Main Memory

	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
00	15	6C	16	6D	50	56	30	6E	C0	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	12	2A	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Registers

R0	3C
R1	00
R2	00
R3	00
R4	00
R5	12
R6	2A
R7	00
R8	00
R9	00
RA	00
RB	00
RC	00
RD	00
RE	00
RF	00

PC 06 =0
IR 5056

```
00:  15,6C  load  R5,[6C]    ;...
02:  16,6D  load  R6,[6D]    ;...
04:  50,56  addi  R0,R5,R6   ;...
06:  30,6E  store R0,[6E]    ;...
08:  C0,00  halt             ;...
0A:  00,00  invalid
```

Address: 108 Modified Ready

FIGURE 2.6
Executing 0x5056.

```
        ....                   ; 'else' part of code
        jmp Exit               ; Unconditionally jump to label Exit
A:      ....                   ; 'then' part of code
        ....
Exit:                          ; Outside the if-then-else structure
        halt
```

2.4.3 Looping

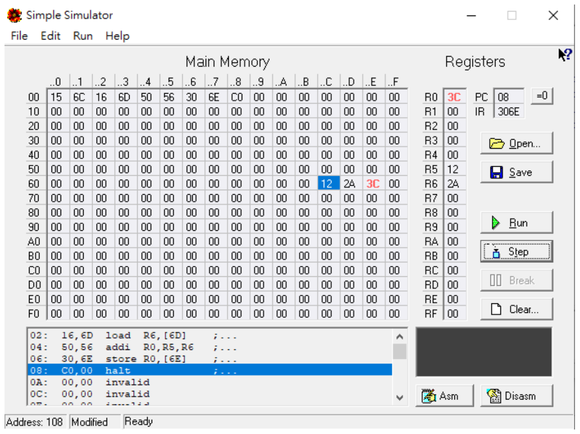
Consider the following `C` code:

```
int a;

for( a = 0; a < 100; a++);
```

We can convert the above program into a assembly code using `jmpLE` instructions to mimic looping. Using `R5` to represent the variable `a`, we use it as the control variable to detect wether we should continue looping or not. In additions, we have to convert constants within the `C` program and store it somewhere. Here we put the value 100 (or 0x64 in hexadecimal) in `R0` for `jmpLE` to compare. Also, we need the value 1 for increment, so we store the value into `R1`.

```
;  A looping code
        load R0, 0x64          ; Loads the value 100 into R0
```



	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
00	15	6C	16	6D	50	56	30	6E	C0	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	12	2A	3C	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

R0	3C
R1	00
R2	00
R3	00
R4	00
R5	12
R6	2A
R7	00
R8	00
R9	00
RA	00
RB	00
RC	00
RD	00
RE	00
RF	00

PC 08 =0
IR 306E

```
02:  16,6D  load  R6,[6D]    ;...
04:  50,56  addi  R0,R5,R6   ;...
06:  30,6E  store R0,[6E]    ;...
08:  C0,00  halt             ;...
0A:  00,00  invalid
0C:  00,00  invalid
```

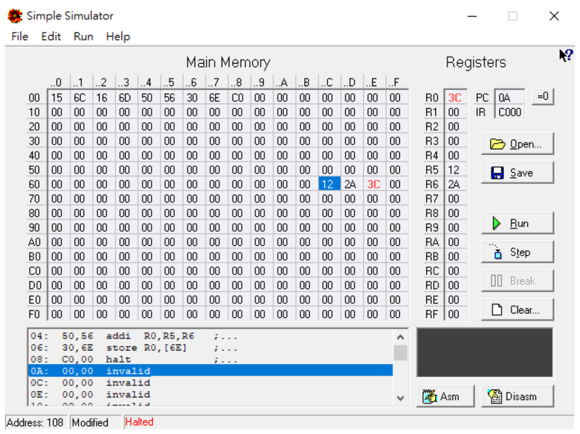
Address: 108 Modified Ready

FIGURE 2.7
Executing 0x306E.

```
        load R1, 0x0          ; Loads the value 0x0 into R1
        load R2, 0x1          ; Loads the value 0x1 into R2
Loop:   addi R1, R1, R2       ; Add: R1 = R1 + R2
        ....                  ; Within loop instructions
        jmpLE R1<=R0, Loop    ; If R1 <= R0, then branch to Loop
        ....                  ; Exits loop
        halt                  ;
```

2.5 Lab Questions

- Write a program to compute the **XOR** value of address 0x6C, 0x6D, and store the results in address 0x6E.
- Write a program to swap the values of memory location of 0xA0 and 0xB0.
- Fill in the values 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA into the memory location starting from 0x50. Write a assembly program and add the values within the memory locations 0x50 to 0x59 and store it into memory location 0x40.



	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
00	15	6C	16	6D	50	56	30	6E	C0	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	12	2A	3C	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

R0	3C
R1	00
R2	00
R3	00
R4	00
R5	12
R6	2A
R7	00
R8	00
R9	00
RA	00
RB	00
RC	00
RD	00
RE	00
RF	00

PC	0A
IR	C000

```
04:   50,56  addi  R0,R5,R6   ;...
06:   30,6E  store R0,[6E]    ;...
08:   C0,00  halt             ;...
0A:   00,00  invalid
0C:   00,00  invalid
0E:   00,00  invalid
```

Address: 108 Modified Halted

FIGURE 2.8
Executing 0xC000.

- Write a program to add the values of 1 to `x`, where $1 \leq x \leq 20$. (`x` can be stored in some register)
- For the program in Figure 2.2, instead of adding the value of memory address 0x6C and 0x6D, i.e., 0x6C + 0x6D, change the program with the available instructions to do subtraction, i.e., 0x6D - 0x6D.
- (Hard) Write a program to perform bubble sort within the memory locations of 0xC0 to 0xCF. (Requires using Op-code **D** and **E**)

2.6 Lab Report

Your lab report is due before the next class.

Machine instruction			
Op-code	Operand	Assembly Instruction	Operation
1	RXY	`load R, [XY]`	Load R with the content from the memory cell at address XY
2	RXY	`load R, XY`	Load R with the bit pattern XY
3	RXY	`store R, [XY]`	Store the content of R into the memory cell at address XY
4	0RS	`move S, R`	Move content of R into S
5	RST	`addi R, S, T`	Add S and T and put the result in R (R, S, and T are in two's complement integer notation)
6	RST	`addf R, S, T`	Add S and T and put the result in R (R, S, and T are in floating-point notation)
7	RST	`or R, S, T`	OR the bit patterns in S and T and put the result in R
8	RST	`and R, S, T`	AND the bit patterns in S and T and put the result in R
9	RST	`xor R, S, T`	XOR the bit patterns in S and T and put the result in R
A	R0X	`ror R, X`	Circularly rotate the bit pattern in R one bit to the right X times
B	RXY	`jmpEQ R=R0, XY`	Start decoding the instruction located at address XY if the bit pattern in R is equal to the bit pattern in register 0
C	000	`halt`	Halt execution
D	0RS	`load R, [S]`	Load R with the content from the memory cell whose address is in S
E	0RS	`store R, [S]`	Store the content of R into the memory cell whose address is in S
F	RXY	`jmpLE R<=R0, XY`	Start decoding the instruction located at address XY if the bit pattern in R is less than or equal to the bit pattern in register 0
		`jmp XY`	Unconditional jump to address XY
		`db XY`	Creates a static space at address XY filled with data

TABLE 2.1
The instruction set of the simulator.