159.102 Notes

Section 5 : Programming the MASSEY machine

Reminder – The Instruction Set for the MASSEY Machine

Each instruction consists of 16 bits written as 4 hexadecimal digits. The first digit is the **opcode**. The other 3 digits are the **operands**. The list of machine instructions follows.

1RXY **Load** register R with the value XY.
e.g. 1213 means load register R2 with the hexadecimal number 13 (19 in decimal).

20RS **Load** register R with the number in register S.
e.g. 20A7 means load register R10 with the number in register R7.

3RXY **Load** register R with the number in the memory location at address XY.
e.g. 34AB means load register R4 with the contents from memory location AB

4RXY **Store** the number in register R in the memory location at address XY.
e.g. 45B1 means store the contents of register R5 in the memory location B1

5RST **Add** the numbers in registers S and T and load the result into register R.
Note: this is **floating point addition**.
e.g. 534E means add the numbers in R4 and R14 and load the result into R3.

6RST **Add** the numbers in registers S and T and load the result into register R.
Note: this is **integer addition** using 2's complement arithmetic.
e.g. 6823 means add the contents of R2 and R3 and load the result into R8.

70R0 **Negate** register R. (Carry out complement and add 1 – in effect, multiply R with -1)
e.g. 70A0 means multiply the number in R10 by -1.

8R0X **Shift** the number in register R to the **right** X times.
e.g. 8403 means that the number in R4 must be shifted 3 bits to the right.

9R0X **Shift** the number in register R to the **left** X times.
e.g. 9602 means that the number in R6 must be shifted 2 bits to the left.

ARST **AND** the numbers in registers S and T and load the result into register R.
e.g. A045 means AND the numbers in R4 and R5 and load the result into R0.

BRST **OR** the numbers in registers S and T and load the result into register R.
e.g. BC74 means OR the numbers in R7 and R4 and load the result into R12.

CRST **XOR** the numbers in registers S and T and load the result into register R.
e.g. C5F3 means XOR the numbers in R15 and R3 and load the result into R5.

DRXY **Jump** to the instruction at address XY **`if`** the value in register R is equal to the value in register R0.
e.g. D43C means the following: (a) compare the contents of R4 with R0.
(b) if they are equal load 3C into the **program counter**.

E000 **Halt**

A program to do addition

Address	Contents
00	3508
01	3609
02	6056
03	400A
04	E000
05	0000
06	0000
07	0000
08	001B
09	0008
0A	0000

Write the instructions in Assembler code:

Machine code written in words like this is called **assembler code**. It is a direct translation from the machine code but is easier to read. Note that C++ is ***not*** a direct translation from the machine code. This is actually expanded assembler code to make it easier to read. In real assembler code, the first line would be written something like: LD R5, 08

LOAD R5 with the number in memory location 08

LOAD R6 with the number in memory location 09

R0 = R5 + R6

STORE the contents of R0 into memory location 0A

HALT

Working – show the changes in the PC, Instruction Register, Registers and Memory:

Program Counter: 00, 01, 02, 03, 04

Instruction Register: 3508, 3609, 6056, 400A, E000

Registers: R5 = 1B, R6 = 08, R0 = 23

Memory: store 23 in 0A

Extra working: 1B + 8 = 23 (decimal equivalent is 27 + 8 = 35)

What is the equivalent C++ code?

```
int num1, num2, result;
num1 = 27;
num2 = 8;
result = num1 + num2;   // in C++ we use variables in memory – we do not use registers
```

A program to do subtraction

Address	Contents
00	11FC
01	1211
02	7020
03	6012
04	4007
05	E000
06	0000
07	0000

Write the instructions in Assembler code:

Machine code written in words like this is called **assembler code**. It is a direct translation from the machine code but is easier to read. Note that C++ is ***not*** a direct translation from the machine code. This is actually expanded assembler code to make it easier to read. In real assembler code, the first line would be written something like: LD R1, FC

LOAD R1 with FC

LOAD R2 with 11

NEGATE R2 (i.e. R2 changes to -11)

R0 = R1 + R2

STORE the contents of R0 into memory location 07

HALT

Working: show the changes in the PC, Instruction Register, Registers and Memory:

Program Counter: 00, 01, 02, 03, 04, 05

Instruction Register: 11FC, 1211, 7020, 6012, 4007, E000

Registers: R1 = FC, R2 = 11 … -11, R0 = EB

Memory: store EB in 07

Extra working: FC + (-11) = EB

What is the equivalent C++ code?

```
int result;
result = 252 – 17;
```

A program to modify instructions while the program is running

Address	Contents
00	1404
01	1505
02	1066
03	9008
04	1145
05	6201
06	4207
07	0000
08	460A
09	E000
0A	0000
0B	0000

Write the instructions in Assembler code:

LOAD R4 with 4

LOAD R5 with 5

LOAD R0 with 66

SHIFT R0 LEFT 8 bits (2 hex digits = 8 bits)

LOAD R1 with 45

R2 = R0 + R1

STORE the contents of R2 into memory location 07

R6 = R4 + R5 (this instruction wasn't there when we started)

STORE the contents of R6 into memory location 0A

HALT

Working: show the changes in the PC, Instruction Register, Registers and Memory:

(**Bold** indicates the instruction that has been created by the program while it runs)

Program Counter: 00, 01, 02, 03, 04, 05, 06, 07, 08, 09

Instruction Register: 1404, 1505, 1066, 9008, 1145, 6201, 4207, **6645**, 460A, E000

Registers: R4 = 04, R5 = 05, R0 = 66… 6600, R1 = 45, R2 = 6645, R6 = 09

Memory: **store 6645 in 07**, store 09 in 0A

Extra working: shift 66 left by 2 hex digits = 6600 (all in hex)
6600 + 45 = 6645 (all in hex)

What is the equivalent C++ code?

Memory location 07 started off as 0000 but an instruction (6645) was placed there while the program was running. This is not possible in C++ or other high-level programming languages.

Programming an IF-ELSE statement

Address	Contents
00	1113
01	1212
02	7020
03	6312
04	1480
05	9408
06	A534
07	1000
08	D50B
09	1101
0A	D00C
0B	1102
0C	410E
0D	E000
0E	0000

Write the instructions in Assembler code:

LOAD R1 with 13 (19 in decimal)

LOAD R2 with 12 (18 in decimal)

NEGATE R2 (R2 is now -18)

R3 = R1 + R2

LOAD R4 with 80

SHIFT R4 LEFT 8 bits (2 hex digits = 8 bits) (R4 is now 8000)

R5 = R3 AND R4 (R5 will be zero if R3 is positive)

LOAD R0 with 00

JUMP to memory location 0B if R5 == R0 (i.e. if R3 is positive)

LOAD R1 with 01

JUMP to memory location 0C if R0 == R0 (an unconditional jump)

LOAD R1 with 02 (this instruction is in memory location 0B)

STORE the contents of R1 into memory location 0E (this instruction is in memory location 0B)

HALT

Working – show the changes in the PC, Instruction Register, Registers and Memory:

(**Bold** indicates a JUMP instruction and the destination of the JUMP)

Program Counter: 00, 01, 02, 03, 04, 05, 06, 07, 08, **0B**, 0C, 0D

Instruction Register: 1113, 1212, 7020, 6312, 1480, 9408, A534, 1000, **D50B**, 1102, 410E, E000

Registers: R1 = 13... 02, R2 = 12...-12, R3 = 01, R4 = 80...8000, R5 = 00, R0 = 00

Memory: store 02 in 0E

Extra working: shift 80 left by 2 hex digits = 8000 (all in hex)
0001 AND 8000 = 0000
if (R5 == R0) then set PC to 0B

What is the equivalent C++ code?

```
int num, result;

num = 19;
if (num > 18) {
  result = 2;
} else {
  result = 1;
}
```

Note 1: (num > 18) is true if num – 18 is a positive number. How do we know if a number is positive? It will have a zero in the sign bit.

Note 2: the machine code for the "true section" of the if statement (load R1 with 2) comes AFTER the machine code for the "false section" (load R1 with 1). This may make it confusing to read.

Programming a LOOP

Address	Contents
00	1000
01	1F03
02	1E0A
03	1D80
04	9D08
05	1100
06	4165
07	3165
08	2021
09	7020
0A	622E
0B	A32D
0C	D30E
0D	D011
0E	611F
0F	4165
10	D007
11	4175
12	E000

Some initial comments:

In machine code, all the work takes place in the registers. For this reason, machine code programs often start off by loading useful values into certain registers. When the program is running, these registers remain unchanged and can be used when required at different points in the program. In this particular program, the following registers are loaded at the start:

R0 is set to zero to be used by the JUMP instruction
RF is set to 3 because in this example we are calculating `x = x + 3`
RE is set to ten (0A) because in this example we are checking `if (x < 10)`
RD uses shift to set a mask with a 1 on the left and the rest is zero (to check the sign bit)

In addition, we load memory location 65 (via R1) with a starting value. For this example, I have selected a starting value of zero but you could select any starting value.

Write the instructions in Assembler code:

LOAD R0 with zero

LOAD RF with 3 (this program calculates x = x + 3)

LOAD RE with A (this program checks if (x < 10)

LOAD RD with 80

SHIFT RD LEFT 8 bits (2 hex digits = 8 bits) (RD is now 8000)

LOAD R1 with zero

STORE R1 in memory location 65 (start value for x)

(end of initialisation of registers and memory locations)

(the next line is at location 07 and is the start of the test for the loop)

LOAD R1 with the contents of memory location 65

LOAD R2 with R1

NEGATE R2 (i.e. R2 is now -R1)

R2 = R2 + RE (i.e. R2 = -R1 + RE) (in other words R2 = ten – R1)

R3 = R2 AND RD (if R2 is positive then the result in R3 will be zero)

JUMP to location 0E if R3 == R0 (jump if ten – R1 is positive, i.e. if R1 is less than ten)

JUMP to location 11 if R0 == R0 (always true, i.e. an unconditional jump)

(the next line is at location 0E and is the start of the body of the loop)

R1 = R1 + RF (x = x + 3)

STORE R1 in memory location 65 (update the value of variable x)

JUMP to location 07 if R0 == R0 (always jump back to the start of the test for the loop)

(the next line is at location 11 and is the first line after the loop)

STORE R1 in memory location 75 (equivalent to y = x;)

HALT

What is the equivalent C++ code?

```
int x, y;
x = ???;  // select any number
while (x < 10) {
  x = x + 3;
}
y = x;
```

Note that (in the machine code) variable x is represented by memory location 65 and variable y is represented by memory location 75.

Optimisation of machine code

Optimisation is a technique designed to increase the speed and performance of machine code. The basic approach is to look for redundant machine code instructions and remove them from the program. If you took great care when designing and creating your machine code, then optimisation would not be needed. However, we often create machine code one section at a time and it is easy to overlook where instructions become redundant. Let us look at optimisation in the loop program above. I will use the assembler code and not the machine code, just because the assembler code is easier to read.

In this program, the first instruction for the test of the loop is this:
LOAD R1 with the contents of memory location 65

The last line of the body of the loop (before it jumps back to repeat the test) is this:
STORE R1 in memory location 65 (update the value of variable x)

When the program is running, each time the loop ends, it jumps back to the test. This means that these two lines are actually executed in this order:
STORE R1 in memory location 65 (update the value of variable x)
LOAD R1 with the contents of memory location 65

The LOAD instruction is redundant as R1 already contains the value that is in memory location 65. We can thus remove the LOAD instruction from the program – one less instruction to be executed.

The main use of optimisation is inside compilers. Compilers (a) translate your high-level instructions into machine code, and then (b) optimise the machine code. This means that your final executable program (in machine code) will be as efficient as possible.

Comparing machine code with high level languages

All numbers are in binary (or hexadecimal). Machine code does not use decimal numbers.

There are no if-statements and loops. Everything is done with JUMP.

Simple tests (e.g. x < 10) are complicated to calculate in machine code.

There is a C++ equivalent of JUMP called goto. Do not use goto in C++ programs. Jumping is used at the machine code level and makes high level programs difficult to read (and easy to make a mistake).

Machine code often uses instructions such as SHIFT and AND.

Variables are never used directly in machine code. E.g. y = x; can only be done by loading x into a register and then storing the register into y.

Machine code is generally more complicated to use, harder to read and you are more likely to make mistakes (and then it is more difficult to fix the errors).

Machine code can change instructions while the program is running. This is because data and instructions are actually the same thing (they are all binary numbers). High level programs usually cannot do this because they separate the instructions from the data. Some people argue that to have "true" artificial intelligence we need programs that can change their instructions while they are running. Is this a good idea…?

If written by an expert, machine code will usually be more efficient (and thus faster to execute) than high level code.

Definitions

High level language – a programming language designed to overcome the problems of programming in machine code. Usually by making instructions more readable and easier to use. Examples of high-level languages include C++, C, C#, Java and Python.

Machine code – instructions in binary using the instruction set for a particular computer. Usually stored in .exe files (i.e. the code can be immediately executed).

Optimisation of machine code – instructions removing redundant machine code instructions to make a program more efficient because (a) they take up less memory (there are fewer instructions) and (b) the program runs faster (there are less instructions to execute.

Assembler code – a low level programming language in which instructions usually correspond directly to the machine code instructions. Usually stored in .asm files.

Assembler – a program that translates assembler code into machine code.

Compiler – a program that translates high level code into machine code and then optimises the machine code.