Fun with Toy Computer

Part 2: Programming in Assembly

Assembly provides us with a higher level of abstraction than machine code. Instead of writing machine instructions specifying ALU circuits directly to memory addresses in the RAM, we use labels to represent addresses and mnemonics (keywords) to represent instructions. We then **compile** the program to the actual values that are finally written to the RAM. How the instructions are translated to values and where the values are stored in the RAM now becomes the concern of the **compiler** and not the programmer. In other words, assembly adds a **layer of abstraction** over machine code that allows us to express solutions to problems in a form that is more similar to the way we think.

In this activity we will play around with a simple assembly language designed for *Toy*.

In *Toy* assembly, note:

- *Mnemonics* (keywords) are in lower case.
- *Registers* are marked with the percent symbol: %0 to %f.
- *Labels* are used to indicate addresses in the RAM. Labels must be in lower case, start with a letter, and can then be followed by letters, underscores and numeric digits; labels are followed by a colon.
- *Indirect referencing* of memory addresses is achieved using brackets; e,g. [%0] refers to the value in RAM at the address stored in register %0.
- Binary, octal and hexadecimal values are prefixed by 0b, 0o and 0x respectively; values with no prefixes are treated as denary.
- Some assembly instructions use registers %d to %f for scratch work, so we should avoid using these registers.

The instructions available to us in Toy assembly are shown in the following tables.

Assembly Instructions

Instruction	Example	Effect
`halt`	`halt`	Halts execution of the program.
`not` *d s*	`not %a %b`	Stores bitwise not of value in register *s* to register *d*.
`and` *d s t*	`and %a %b %c`	Stores bitwise and of values stored in registers *s* and *t* in register *d*.
`and` *d s v*	`and %a %b 0x89AB`	Stores bitwise and of value stored in register *s* and value *v* in register *d*.
`or` *d s t*	`or %a %b %c`	Stores bitwise or of values stored in registers *s* and *t* in register *d*.
`or` *d s v*	`or %a %b 0x89AB`	Stores bitwise or of value stored in register *s* and value *v* in register *d*.
`xor` *d s t*	`xor %a %b %c`	Stores bitwise xor of values stored in registers *s* and *t* to register *d*.
`xor` *d s v*	`xor %a %b 0x89AB`	Stores bitwise xor of value stored in register *s* and value *v* to register *d*.
`lsh` *d s t*	`lsh %a %b %c`	Stores value in register *s* left shifted by value stored in register *t* to register *d*.
`lsh` *d s v*	`lsh %a %b 3`	Stores value in register *s* left shifted by value *v* to register *d*.
`rsh` *d s t*	`rsh %a %b %c`	Stores value in register *s* right shifted by value stored in register *t* to register *d*.
`rsh` *d s v*	`rsh %a %b 3`	Stores value in register *s* right shifted by value *v* to register *d*.
`add` *d s t*	`add %a %b %c`	Stores sum of values stored in registers *s* and *t* to register *d*.
`add` *d s v*	`add %a %b 0x89AB`	Stores sum of value stored in register *s* and value *v* to register *d*.
`sub` *d s t*	`sub %a %b %c`	Stores value stored in register *s* minus value stored in register *t* to register *d*.
`sub` *d s v*	`sub %a %b 0x89AB`	Stores value stored in register *s* minus value *v* to register *d*.
`ld` *d l*	`ld %0 loop`	Loads address marked by label *l* to register *d*.
`ld` *d v*	`ld %0 0x89AB`	Loads value *v* to register *d*.
`ld` *d a*	`ld %0 [0xA0]`	Loads value at address *a* to register *d*.
`ld` *d la*	`ld %0 [x]`	Loads value in address marked by label *l* to register *d*.
`ld` *d p*	`ld %0 [%1]`	Loads value at address stored in register *p* to register *d*.
`st` *a s*	`st [0xA0] %0`	Stores value from register *s* to address *a*.
`st` *la d*	`st [x] %0`	Stores value in register *d* to address marked by label *l*.
`st` *p s*	`st [%0] %1`	Stores value in register *s* to address stored in register *p*.
`mv` *d s*	`mv %0 %1`	Moves value in register *s* to register *d*.
`jz` *d a*	`jz %0 0xA0`	Jumps to address *a* if value in register *d* is zero.
`jz` *d l*	`jz %0 end`	Jumps to address marked by label *l* if value in register *d* is zero.
`jp` *d a*	`jp %0 0xA0`	Jumps to address *a* if value in register *d* is positive.
`jp` *d l*	`jp %0 end`	Jumps to address marked by label *l* if value in register *d* is positive.
`jmp` *a*	`jmp 0xA0`	Jumps to address *a*.
`jmp` *l*	`jmp loop`	Jumps to address marked by label *l*.
`call` *d a*	`call %0 0xA0`	Stores current position to register *d* and jumps to address *a*.
`call` *d l*	`call %0 print`	Stores current position to register *d* and jumps to address marked by label *l*.
`ret` *d*	`ret %0`	Jumps to address stored in register *d*.

The instructions **and**, **or**, **xor**, **lsh**, **rsh**, **add** and **sub** also work with only two arguments. For example, **and** `%a %b` means the same thing as **and** `%a %a %b`.

Special Instructions

Special	Example	Effect
`.main`	`.main`	Indicates the start point of the program.
`.word`	`.word`	Adds an empty register to the memory.
`.data`	`.data 1, 0xab12, 0b101`	Stores data to memory.
`.ascii`	`.ascii "Hello, world!"`	Stores ascii values of characters (then zero) to memory.
`.line`	`.line`	Outputs a new line.
`.char` *s*	`.char %0`	Outputs value in register *s* as an ascii character.
`.bin` *s*	`.bin %0`	Outputs value in register *s* as binary.
`.oct` *s*	`.oct %0`	Outputs value in register *s* as octal.
`.den` *s*	`.den %0`	Outputs value in register *s* as denary.
`.hex` *s*	`.hex %0`	Outputs value in register *s* as hexadecimal.
`.rand` *d*	`.random %0`	Stores random value to register *d*.
`.input` *d*	`.input %0`	Stores input value to register *d*.

Example

Recall the machine code programming example for which we wanted to write a program that accepts two numbers from the user and then outputs their sum. This time, let's solve the problem using assembly, and extend the program to have prompts for the user.

```
    prompt: .ascii "Input "  ; define some strings to use
         a: .ascii "A: "
         b: .ascii "B: "
       sum: .ascii "A + B = "

     print: ld %b [%0]        ; define a procedure to print strings
            jz %b done_print ; expects string address in %0
            .char %b          ; uses %b to store each character
            add %0 1          ; increments the value in %0
            jmp print
done_print: ret %a            ; sets PC to address in %a
```

```
.main                 ; program start
ld %0 prompt          ; loads address labeled prompt to %0
call %a print         ; stores PC to %a, sets PC to print
ld %0 a               ; repeat to print string at label a
call %a print
.input %1             ; get first number from user
ld %0 prompt          ; print string at label prompt
call %a print
ld %0 b               ; print string at label b
call %a print
.input %2             ; get second number from user
add %1 %2             ; add the values (result in %1)
ld %0 sum             ; print string at label sum
call %a print
.den %1               ; output the sum in denary
.line                 ; output a new line
halt
```

An example run of this program gives us the following (with user input in bold).

```
Input A: 10
Input B: 20
A + B = 30
```

Notice that, unlike when we were programming in machine code, we no longer need to keep track of memory addresses (instead we use labels) or ALU circuit addresses (instead we use mnemonics like *add*).

Of course, the computer only understands instructions as words in the RAM. Thus, the above assembly code needs to be **compiled** to these instructions. When we compile the above program, we get the following equivalent machine language program.

```
PC: 20
00: 0049 ;  'I'
01: 006e ;  'n'
02: 0070 ;  'p'
03: 0075 ;  'u'
04: 0074 ;  't'
05: 0020 ;  ' '
06: 0000 ;  null
07: 0041 ;  'A'
08: 003a ;  ':'
09: 0020 ;  ' '
0a: 0000 ;  null
0b: 0042 ;  'B'
0c: 003a ;  ':'
0d: 0020 ;  ' '
0e: 0000 ;  null
0f: 0041 ;  'A'
10: 0020 ;  ' '
11: 002b ;  '+'
12: 0020 ;  ' '
13: 0042 ;  'B'
14: 0020 ;  ' '
15: 003d ;  '='
16: 0020 ;  ' '
17: 0000 ;  null
18: ab00 ;  R[b] <- M[R[0]]
19: cb1f ;  if (R[b] == 0) PC <- 1f
1a: 9bf5 ;  M[f5] <- R[b]
1b: 7e01 ;  R[e] <- 1
1c: 100e ;  R[0] <- R[0] + R[e]
1d: 7f18 ;  R[f] <- 18
1e: ef00 ;  PC <- R[f]
1f: ea00 ;  PC <- R[a]
20: 7000 ;  R[0] <- 0
21: fa18 ;  R[a] <- PC; PC <- 18
22: 7007 ;  R[0] <- 7
23: fa18 ;  R[a] <- PC; PC <- 18
24: 81f0 ;  R[1] <- M[f0]
25: 7000 ;  R[0] <- 0
26: fa18 ;  R[a] <- PC; PC <- 18
27: 700b ;  R[0] <- b
28: fa18 ;  R[a] <- PC; PC <- 18
29: 82f0 ;  R[2] <- M[f0]
2a: 1112 ;  R[1] <- R[1] + R[2]
2b: 700f ;  R[0] <- f
2c: fa18 ;  R[a] <- PC; PC <- 18
2d: 91f4 ;  M[f4] <- R[1]
2e: 90f6 ;  M[f6] <- R[0]
2f: 0000 ;  halt
```

Compare this machine language program to the assembly program in terms of readability and keeping track of memory and instruction addresses. The way the compiler remembers which address each label represents is through a **lookup table**. For this program, the compiler sets up the following lookup table.

Label	Address
prompt	00
a	07
b	0b
sum	0f
print	18
done_print	1f

Problem 1

Notice that toy assembly allows us to load constants to registers. For example, the following program loads constant byte 0xAB to register %a and then outputs the value. When we compile this and look at the generated machine code, we see that the move instruction makes use of the *load address* machine operation (even though we do not interpret 0xAB as an address in this context).

load_byte.asm

```
.main
  ld %a 0xab
  .hex %a
  .line
halt
```

Address	Data	
00	7aab	ld %a 0xab
01	9af3	.hex %a
02	90f6	.line
03	0000	halt

However, notice what happens when we try to store a *two byte word* to a register. This time, the compiled machine code is more complicated.

load_word.asm

```
.main
  ld %a 0xcdef
  .hex %a
  .line
halt
```

Address	Data	
00	7ecd	?
01	7f08	?
02	5eef	?
03	7fef	?
04	1eef	?
05	7a00	?
06	1aae	?
07	9af3	.hex %a
08	90f6	.line
09	0000	halt

Explain why we can't simply use a single load address machine instruction in this case, and outline how storing a value greater than 0xff to a register has been implemented.

Problem 2

Notice that toy assembly has an instruction **or** that loads the *bitwise disjunction* of the values in two registers to a third register. For example, the following program (i) loads 0xC to register %0, (ii) loads 0x5 to register %1, (iii) loads the bitwise disjunction of these two elements to register %a, and finally (iv) outputs this result in binary and a new line to the output.

disjunction.asm

```
.main
  ld %0 0xc
  ld %1 0x5
  or %a %0 %1
  .bin %a
  .line
halt
```

However, notice that toy computer does not have a machine instruction for *bitwise disjunction* in its ALU! When we compile the above assembly, the following machine code is generated.

Address	Data	
00	700c	ld %0 0xc
01	7105	ld %1 0x5
02	3e01	?
03	4f01	?
04	4aef	?
05	9af1	.bin %a
06	90f6	.line
07	0000	halt

Identify the machine instructions used to implement the bitwise disjunction operation, and explain how the bitwise or operation has been implemented.

Problem 3

We encountered the Fibonacci sequence in the machine language problem set. As a reminder, the Fibonacci sequence is generated as follows.

- The first term is $a_0 = 0$.
- The second term is $a_1 = 1$.
- Subsequent terms are given by $a_n = a_{n-2} + a_{n-1}$.

Thus, the sequence contains the terms: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

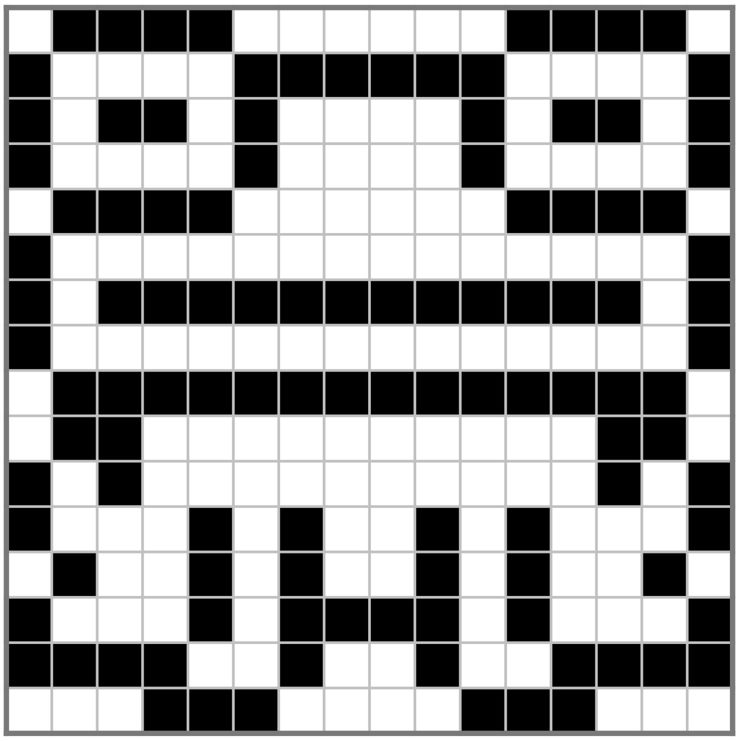
Write an assembly program that inputs a number from the user and then outputs that many terms of the Fibonacci sequence. Include prompts to help the user.

Problem 4

Write an assembly program that inputs two numbers from the user and then outputs the product of the two numbers. Include prompts to help the user.

Problem 5

Write a program that outputs the following sprite to the screen.

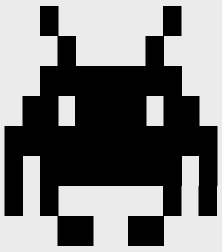


Problem 6 (Challenge)

(a) Write a program that inputs values from the user until the user inputs zero, and then displays the image that corresponds to the values. You can make assumptions, such as the user will input a maximum of 32 pattern values.

An example run (with user input in bold) follows.

```
Input image data:
0x0810
0x0420
0x0ff0
0x1bd8
0x3ffc
0x2ff4
0x2814
0x0660
0
Image:
```



(b) Write a program that inputs values from the user until the user inputs zero, treats the values as machine code instructions, and then runs the program input by the user. You can make assumptions, such as the user will input a maximum of 32 instructions.

For example, recall that the machine code example given part 1 output the sum of two values input by the user, is:

```
PC: 00   ; set the address for the first instruction
00: 80f0 ; load user input to register 0
01: 81f0 ; load user input to register 1
02: 1201 ; load the sum of R[0] and R[1] to register 2
03: 92f4 ; output the value in register 2
04: 90f5 ; output a new line
05: 0000 ; halt the program
```

An example run of the assembly program for this problem (with user input in bold) follows.

```
Input a program:
0x80f0
0x81f0
0x1201
0x92f4
0x90f5
0x0000
Program running:
10
15
25
```