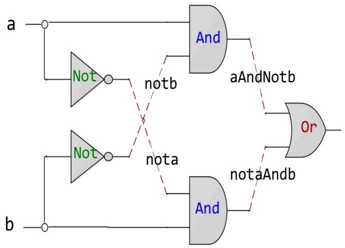
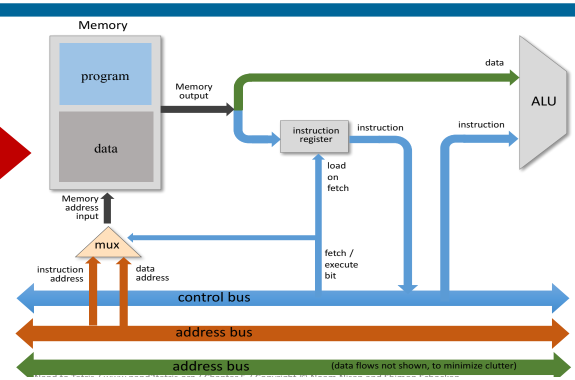
Computer Organization & Assembly Language Programming




```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```




```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 20

Hack Assembly Programming - II

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

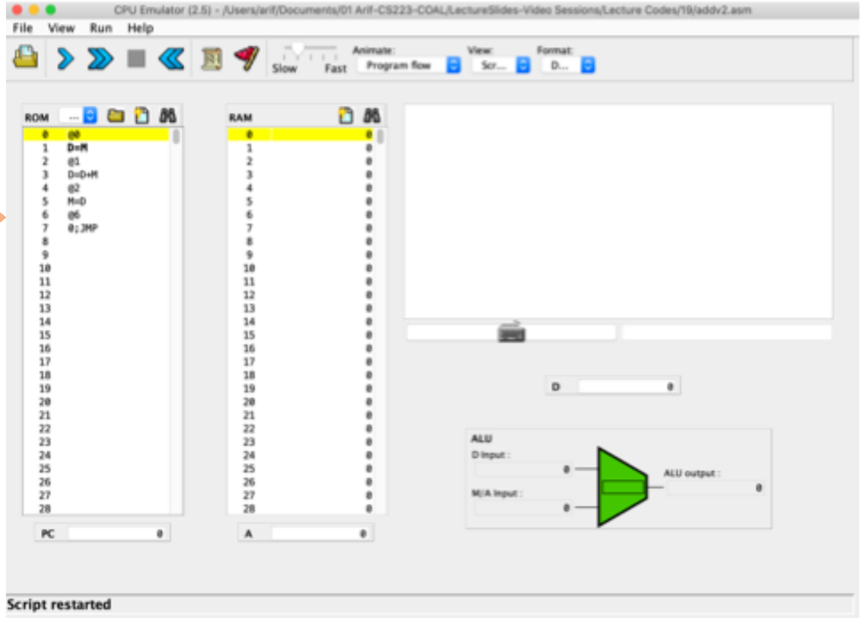
- Recap of Previous Lecture
- Symbols in Hack Assembly Language
  - Built-in Symbols
  - Label Symbols
  - Variable Symbols
- Branching
- Iteration

Recap: CPU Emulator

Hack assembly code

```
    // Program: addv2.asm
    // Computes: RAM[2] = RAM[0] + RAM[1]
    // Usage: put values in RAM[0], RAM[1]

0   @0
1   D=M  // D = RAM[0]

2   @1
3   D=D+M // D = D + RAM[1]

4   @2
5   M=D  // RAM[2] = D

6   @6
7   0;JMP
```

Load



CPU Emulator

- A software tool build in Java
- We can load Hack assembly program into CPU emulator's instruction memory, the CPU emulator translate it into machine language and execute it
- Convenient for debugging and executing symbolic Hack programs in simulation

Symbols in Hack Assembly Language

Symbols in Hack Assembly Language

- Assembly Instructions can refer to memory locations (addresses) using either constants or symbols. Symbols are introduced into Hack assembly programs in the following three ways:
- **Predefined/build-in Symbols:** These are a special subset of RAM addresses that can be referred to by any assembly program using virtual registers, predefined pointers and I/O pointers
- **Label Symbols:** These are user defined symbols, which serve to label destinations of `goto` commands
- **Variable Symbols:** These are also user defined symbols which are assigned unique memory addresses starting at RAM addresses 16 onwards

Pre-Defined / Built-in Symbols

Built-in Symbols: Virtual Registers

To simplify assembly programming, the symbols R0 to R15 are predefined to refer to RAM addresses 0 to 15 respectively

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15

Attention: Hack is case-sensitive! R5 and r5 are different symbols.

These symbols can be used to denote "virtual registers"

Example: Suppose a programmer wants to write a constant value 7 at RAM[5]

Implementation:

```
// let RAM[5] = 7
@7
D=A
@5
M=D
```

Better Style:

```
// let RAM[5] = 7
@7
D=A
@R5
M=D
```

Built-in Symbols: Predefined Pointers

- The following five symbols are predefined to refer to RAM addresses 0 to 4 respectively
- Note that RAM addresses from 0 to 4 has two labels. For example, address 2 can be referred to using either R2 or ARG
- These symbols will come into play in the implementation of the virtual machine, which will not be used in this part of the course

symbol	value
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

Built-in Symbols: I/O Pointers

- The following two symbols SCREEN and KBD are predefined to refer to RAM addresses 16384 (0x4000) and 24576 (0x6000) respectively
- These are the base addresses of the screen and keyboard memory maps (discussed in detail in Lecture # 18)
- These symbols will come into play, when we will write assembly programs that deals with the screen and keyboard in the next lecture

symbol	value
SCREEN	16384
KBD	24576

Branching

Branching

- Branching is the fundamental ability to tell the computer to evaluate certain Boolean expression and based on the result, decide whether or not the flow of execution should continue the next instruction in sequence or jump to some other location in the code
- All programming languages support various branching mechanisms like `if…else`, `while…`, `for…`, and so on
- In machine language we have only one branching mechanism called `goto`

Branching Example

```
   // Program: ifelsev1.asm
   // Computes: if R0 > 0
                    R1 = 1
                else
                    R1 = 0
// Usage: put a value in RAM[0], run and inspect RAM[1]
0  @R0
1  D=M   //D = RAM[0]

2  @8
3  D;JGT // If R0>0 goto 8

4  @R1
5  M=0
6  @10
7  0;JMP

8  @R1
9  M=1

10  @10
11  0;JMP
```

Branching Example (cont…)

```
@R0
D=M


@8
D;JGT

@R1
M=0
@10
0;JMP

@R1
M=1

@10
0;JMP
```

cryptic code

- If we remove all the comments as well as the line numbers, the code become quite unreadable or cryptic
- It is of course really difficult to understand what this code actually do
- Yet the code will work perfectly fine as expected by the programmer

Branching Example (cont…)

```
@R0
D=M

@8
D;JGT

@R1
M=0
@10
0;JMP

@R1
M=1

@10
0;JMP
```

cryptic code

"Instead of imagining that our main task as programmers is to instruct a computer what to do, let us concentrate rather on explaining to human beings (fellow programmers) what we intend a computer to do."

– Donald Knuth

Important

If our programs are not self documented, we will not be able to fix and extend them

Use of Labels

Branching Example: Understanding Labels

```
   // Program: ifelsev1.asm
   // Computes: if R0 > 0
                    R1 = 1
                else
                    R1 = 0
// Usage: put a value in RAM[0], run and inspect RAM[1]
0  @R0
1  D=M   //D = RAM[0]

2  @8
3  D;JGT // If R0>0 goto 8

4  @R1
5  M=0
6  @10
7  0;JMP

8  @R1
9  M=1

10  @10
11  0;JMP
```

Branching Example: Understanding Labels

```
   // Program: ifelsev2.asm
   // Computes: if R0 > 0
                    R1 = 1
                else
                    R1 = 0
// Usage: put a value in RAM[0], run and inspect RAM[1]
0  @R0
1  D=M   //D = RAM[0]

2  @POSITIVE        //@8
3  D;JGT // If R0>0 goto 8

4  @R1
5  M=0
6  @10
7  0;JMP
   (POSITIVE)
8  @R1
9  M=1

10  @10
11  0;JMP
```

Referring to a label (→ @POSITIVE)

declaring a label (→ (POSITIVE))

- These are user-defined symbols, which serve to label destinations of goto commands
- Declared by `(xxx)` directive
- So @xxx refer to the instruction number following the declaration
- A label can be declared only once and can be referred to any number of times and any-where in the assembly program, even before the line in which it is declared
- The name of a user defined symbol can be any sequence of alphabets, digits, underscore, dot, dollar sign and a colon. However, the name must not begin with a digit
- The naming convention is to use uppercase alphabets for labels and lower case alphabets for variables

Branching Example : Understanding Labels

```
   // Program: ifelsev2.asm
   // Computes: if R0 > 0
                    R1 = 1
                else
                    R1 = 0
// Usage: put a value in RAM[0], run and inspect RAM[1]
0  @R0
1  D=M   //D = RAM[0]

2  @POSITIVE       //@8
3  D;JGT // If R0>0 goto 8

4  @R1
5  M=0
6  @END            //@10
7  0;JMP
   (POSITIVE)
8  @R1
9  M=1
   (END)
10  @END           //@10
11  0;JMP
```

Referring to a label (@END, line 6)

declaring a label ((END))

Referring to a label (@END, line 10)

- These are user-defined symbols, which serve to label destinations of goto commands
- Declared by `(xxx)` directive
- So @xxx refer to the instruction number following the declaration
- A label can be declared only once and can be referred to any number of times and any-where in the assembly program, even before the line in which it is declared
- The name of a user defined symbol can be any sequence of alphabets, digits, underscore, dot, dollar sign and a colon. However, the name must not begin with a digit
- The naming convention is to use uppercase alphabets for labels and lower case alphabets for variables

Branching Example : Resolving Labels

```
   // Program: ifelsev2.asm
   // Computes: if R0 > 0
                    R1 = 1
                else
                    R1 = 0
// Usage: put a value in RAM[0], run and inspect RAM[1]
0  @R0
1  D=M   //D = RAM[0]

2  @POSITIVE       //@8
3  D;JGT // If R0>0 goto 8

4  @R1
5  M=0
6  @END           //@10
7  0;JMP
   (POSITIVE)
8  @R1
9  M=1
   (END)
10 @END          //@10
11 0;JMP
```

resolving labels →

ROM

0	@0
1	D=M
2	@8 // @POSITIVE
3	D;JGT
4	@1
5	M=0
6	@10 // @END
7	0;JMP
8	@1
9	M=1
10	@10 // @END
11	0;JMP
12	
13	
14	
15	
32767	

Label resolution rules:

- Label declarations are not translated, are ignored, so generate no code and are called pseudo-commands
- Each reference to a label is translated, i.e., replaced with a reference to the instruction number following that label's declaration

Running an Assembly Program in CPU Emulator

Use of Variables

Variables

- Variable is an abstraction of a container, that has a name and a value
- You can say that it is a named memory location
- In high level languages we also have a type associated with a variable, but in Hack machine/assembly language, we have only 16 bit values of a variable
- So in Hack assembly language, a variable is user-defined symbol **xxx** appearing in the program that is not predefined and is not defined elsewhere using the **(xxx)** directive. It is assigned a unique memory address by the assembler, starting at RAM address 16 (0x0010)

Variables: Example

```
//Program: swap.asm
//flips the values of RAM[0] and RAM[1]
//temp = R1
// R1 = R0
//R0 = temp
// temp = R1
@R1
D=M
@temp
M=D
// R1 = R0
@R0
D=M
@R1
M=D
// R0 = temp
@temp
D=M
@R0
M=D
(END)
@END
0;JMP
```


@temp:

- Since this is the first occurrence of the symbol **`temp`**, not declared as a label elsewhere using **`(temp)`**, so this qualifies it to be a variable
- The assembler will map it to some available memory register, starting at RAM address 16 (0x0010)
- So from this point onwards, each occurrence of **`@temp`** in the program will be translated into **`@16`**

Example: Resolving Variables

```
//Program: swap.asm
//flips the values of RAM[0] and RAM[1]
//temp = R1
// R1 = R0
//R0 = temp
 @R1
 D=M
 @temp
 M=D    // temp = R1
 @R0
 D=M
 @R1
 M=D // R1 = R0
 @temp
 D=M
 @R0
 M=D    // R0 = temp
(END)
 @END
 0;JMP
```

Symbolic variable

Symbolic label

resolving symbols

Symbol resolution rules:

- A reference to a symbol that has no corresponding label declaration is treated as a reference to a variable
- Variables are allocated to the RAM from address 16 onward (say n), and the generated code is @n
- Here we have only one variable, so that is allocated RAM address 16. If there are more they will be allocated address 17, 18, and so on

In other words: variables are allocated to `RAM[16]` onward.

ROM

0	@1
1	D=M
2	@16 // @temp
3	M=D
4	@0
5	D=M
6	@1
7	M=D
8	@16 // @temp
9	D=M
10	@0
11	M=D
12	@12
13	0;JMP
14	
15	
32767	

Implications of Using Symbols

```
//Program: swap.asm
//flips the values of RAM[0] and RAM[1]
//temp = R1
// R1 = R0
//R0 = temp
 @R1
 D=M
 @temp
 M=D   // temp = R1
 @R0
 D=M
 @R1
 M=D // R1 = R0
 @temp
 D=M
 @R0
 M=D   // R0 = temp
(END)
 @END
 0;JMP
```

Symbolic variable

Symbolic label

resolving symbols

ROM

0	@1
1	D=M
2	@16 // @temp
3	M=D
4	@0
5	D=M
6	@1
7	M=D
8	@16 // @temp
9	D=M
10	@0
11	M=D
12	@12
13	0;JMP

Implications:

Symbolic code is easy to read and debug

The program has become Relocatable Code:

- You can take this program and load it into memory, not necessarily to address zero, as long as you remember the base address of memory where this program is loaded
- This is very important when several such programs are loaded and running inside the memory

Running an Assembly Program in CPU Emulator

Iteration

Interactive Processing Example

Pseudo Code:

```
// Computes RAM[1] = 1 + 2 + 3 … + n
  n = R0
  i = 1
  sum = 0
LOOP:
  if i > n goto STOP
  sum = sum + i
  i = i + 1
  goto LOOP
STOP:
  R1 = sum
```

Assembly Code:

```
// Computes RAM[1] = 1 + 2 + 3 … + n
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
  @R0
  D=M
  @n
  M=D   // n = R0
  @i
  M=1   //i = 1
  @sum
  M=0   //sum = 0

  .  .  .  .
```

Variables are allocated to consecutive RAM locations from address 16 onwards

ROM

0	@0
1	D=M
2	@16 // @n
3	M=D
4	@17 // @i
5	M=1
6	@18 // @sum
7	M=0
8	. . .
9	
10	
11	
12	
13	
14	
15	
32767	

Interactive Processing Example

```
// Computes RAM[1] = 1 + 2 + 3 … + n
   n = R0
   i = 1
   sum = 0
LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

```
// Computes RAM[1] = 1 + 2 + 3 … + n
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
  @R0
   D=M
   @n
   M=D     // n = R0
   @i
   M=1   //i = 1
   @sum
   M=0   //sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT     //if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D            // sum = sum + i
   @i
   M=M+1          // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D           // RAM[1] = sum
(END)
   @END
   0;JMP
```

Things To Do

- You all must have a very clear understanding of built-in symbols, labels, variables, branching and iteration
- Download all the assembly program from the course bitbucket repository, make changes to them and execute them in the CPU Emulator
- Run the programs, one instruction at a time, do the working in your head or on a piece of paper, while executing the programs one instruction at a time
- Interested students should try to write down max.asm program that computes the maximum out of two numbers

Coming to office hours does NOT mean you are academically week!