Announcements

- Program #1A
  - due next Thursday
- Reading
  - Chapters 1 & 2 should be read already
  - Chapter 3 & 4 (by Thursday)

Stored Program Computer

- A Program:
  - Consists of a sequence of operations
  - Is loaded into a computer's memory to execute
- Standard set of allowed operations, called *Instructions*
  - each instruction performs a specific operation
  - Instructions can
    - manipulate memory
    - compute a value
    - read from input or write to output
  - What instruction to execute next:
    - normally it is the next instruction
    - provide control to select alternate next instruction
- Program and data are both stored in memory

Completeness of Instructions

- Need to ensure that the computer can execute any computable function.
  - more instructions may make the machine faster
    - but not more powerful
    - many small building blocks can perform the same task
- Arithmetic and logic operations
  - add, complement (negate)
  - AND and NOT (provides NAND which is enough)
- Data movement instructions
  - Load from memory
  - Store to memory
- Control instructions
  - check values of memory or other locations
  - provide a way to change the next instruction to be executed
- Input/Output Instructions

Parts of a Computer

- Arithmetic Logic Unit (ALU)
  - Performs mathematical and logical operations
- Memory
  - Stores program and data
- Registers
  - Hold temporary values
  - Registers can be dedicated to a specific task or general
  - Special Register, R1, called *Program Counter*, stores address of the next instruction to execute
  - Special Register, R0, always holds value 0
- I/O Devices
  - Store, Display, Input, or Transmit values

Assembly Language vs. Machine Code

- Machine Code
  - Sequences of bits (numbers in binary)
  - Writing Program as numbers is tedious
- Assembly language provides a more symbolic form
  - Instructions are names – that in correspond to their purpose
  - Registers are R0-R15
  - Symbolic labels
    - Replace memory address by name
    - Start and instruction with a name to indicate address
  - Allows pseduo-instructions
    - Data – simply means treat next word as entire value of memory location

What do Instructions look like at the Machine Code Level?

- For Project #1, instructions:
  - one word of memory
  - each word is 32 bits long
  - each word contains three parts:
    - 4 bit Opcode (indicates what to do)
    - 3 register indicators
    - 16 bits for a memory address
      - implies there are at most 65536 words of memory

Purpose	**Opcode**	**Register$_1$**	**Register$_2$**	**Register$_3$**	**Memory**
Size (bits)	**4**	**4**	**4**	**4**	**16**

Project 1a

- machine.h

```
typedef struct {
    unsigned int opCode:4;
    unsigned int r1:4;
    unsigned int r2:4;
    unsigned int r3:4;
    unsigned int address:16;
} instruction;

typedef union {
    instruction insn;
    int number;
} memoryLocation;

#define MEM_SIZE 65536

extern memoryLocation
   memory[MEM_SIZE];
```

- disassemble.c

```
#include "machine.h"

void printInsn(
            instruction insn)
{
}

void disassemble(
         memoryLocation mem[],
         int limit)
{
}
```

Types of Instructions in their Assembly Language Format

- **Load <register$_1$> <memory>**
  - Copies the value stored in the memory location <memory> into the register location <register$_1$> (opcode 1)
- **Load <register$_1$> #<number>**
  - Copies the supplied number #<number> into the register location <register$_1$> (opcode 1)
- **Move <register$_1$> <register$_2$>**
  - Copies the value stored in <register$_2$> into <register$_1$> (opcode 2)
- **Store <register$_1$> <memory>**
  - Copies the value stored in <register$_1$> into memory location <memory> (opcode 3)

Assembly Language Instructions (cont.)

- **Add <register$_1$> <register$_2$> <register$_3$>**
  - Adds the value stored in <register$_1$> to the value stored in <register$_2$> and stores the result into <register$_3$> (opcode 4)
- **Negate <register$_1$>**
  - Negates the value stored in <register$_1$> (i.e. 1 becomes –1), (opcode 6)
- **Input <register$_1$>**
  - Read an integer from standard input, and store the value in <register$_1$> (opcode 10)
- **Output <register$_1$>**
  - Write the integer value stored in <register$_1$> to standard output (opcode 11)

Assembly Language Instructions (cont.)

- **Branch <register$_1$> <register$_2$> <memory>**
  - Stores the current value of R1 (program counter) into <register$_1$>, Sets the value of R1 (program counter) to <register$_2$> plus the value of the <memory> field. (opcode 7)

- **Bnn <register$_1$> <memory>**
  - If the value stored in <register$_1$> is non-negative (i.e. >= 0), change the program counter (next instruction to execute) to execute the instruction stored in <memory> next (opcode 8)

- **Halt**
  - Stops the execution of the program (opcode 5)

Steps Involved in Executing a Program

```
while (not done)  {
    Read the instruction from memory
    figure out what the instruction is (this is called decode)
    read the effective address from memory
        # only required if the instruction is indirect
    perform the appropriate operation
    update the Program Counter
    update memory or registers
}
```

Simple Program

- Program:

main:	Load R3 #24	0x1310 0018
	Add R3 R0 R3	0x4303 0000
	Store R3 Val	0x3300 0008
	Negate R3	0x6300 0000
	Bnn R3 main	0x8300 0000
	Bnn R0 Label	0x8000 0007
	Output R3	0xA300 0000
Label:	Halt	0x5000 0000
Val:	Data 0	0x0000 0000

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0000
09	
0A	
0B	
0C	

Register

Program Counter →

Register	Value
00	0x0000 0000
01	0x0000 0000
02	0x0000 0000
03	0x0000 0000
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0000
09	
0A	
0B	
0C	

Register

Program Counter →

Register	Value
00	0x0000 0000
01	0x0000 0001
02	0x0000 0000
03	0x0000 0018
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Load R3 #24

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0000
09	
0A	
0B	
0C	

Register

Program Counter →

Register	Value
00	0x0000 0000
01	0x0000 0002
02	0x0000 0000
03	0x0000 0018
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Add R3 R0 R3

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0018
09	
0A	
0B	
0C	

Register

Program Counter →

Register	Value
00	0x0000 0000
01	0x0000 0003
02	0x0000 0000
03	0x0000 0018
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Store R3 8

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0018
09	
0A	
0B	
0C	

Register

Program Counter →

Register	Value
00	0x0000 0000
01	0x0000 0004
02	0x0000 0000
03	0xFFF FFE8
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Negate R3

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0018
09	
0A	
0B	
0C	

Register

Program Counter →

Register	Value
00	0x0000 0000
01	0x0000 0005
02	0x0000 0000
03	0xFFF FFE8
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Bnn R3 0

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0018
09	
0A	
0B	
0C	

Register

Program Counter →

Register	Value
00	0x0000 0000
01	0x0000 0007
02	0x0000 0000
03	0xFFF FFE8
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Bnn R0 0007

Program Execution

Memory

Location	Value
00	0x1310 0018
01	0x4303 0000
02	0x3300 0008
03	0x6300 0000
04	0x8300 0000
05	0x8000 0007
06	0xA300 0000
07	0x5000 0000
08	0x0000 0018
09	
0A	
0B	
0C	

Program Counter →

Program Stops!

Halt

Register

Register	Value
00	0x0000 0000
01	0x0000 0008
02	0x0000 0000
03	0xFFF FFE8
04	0x0000 0000
05	0x0000 0000
06	0x0000 0000
07	0x0000 0000
08	0x0000 0000
09	0x0000 0000
0A	0x0000 0000
0B	0x0000 0000
0C	0x0000 0000
0D	0x0000 0000
0E	0x0000 0000
0F	0x0000 0000

Another Program

```
main:   Input R3
        Load R2 #1
        Negate R2
        Add R2 R3 R3
loop:   Input R4
        Add R4 R5 R5
        Add R2 R3 R3
        Bnn R3 loop
        Output R5
Label:  Halt
Val:    Data 0
```

- What does this do?