POWERS OF 2

- You are required to memorize these.

$2^0 = 1$		$2^7 = 128$
$2^1 = 2$		$2^8 = 256$
$2^2 = 4$		$2^9 = 512$
$2^3 = 8$		$2^{10} = 1024$
$2^4 = 16$		$2^{11} = 2048$
$2^5 = 32$		$2^{12} = 4096$
$2^6 = 64$		$2^{13} = 8192$

DECIMAL HEXADECIMAL BINARY

You are required to memorize these

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

2S COMPLEMENT

- You should have learned the 2s complement system in Discrete Math.
- You will need to use 2s complement soon in assembly language.
- So you might as well review it too.

2'S COMPLEMENT

- Allows both positive and negative numbers
- Works because of a fixed number of bits
- Assume you have only 5 bits to work with
- You can store 00000 to 11111
- Adding a number to its additive identity should give zero so 7 + (-7) = 0

```
 00111 (7)
+11001 (?)
 00000 (0)
```

- 11001 must be -7

DECIMAL TO 2'S COMPLEMENT

- If the number is positive
  - Treat as unsigned binary
- If the number is negative
  - Convert as unsigned
  - Invert all bits and add 1
- Convert 17 and -23 to 6 bit 2's complement

DECIMAL TO 2'S COMPLEMENT EXAMPLE

- Convert 17 and -23 to 6 bit 2's complement

17 (Positive so treat as unsigned binary and done)
17 = 16 + 1 = 010001 (make sure to write the number as six bits)

Answer: 010001

-23 (negative so convert as unsigned then do the following)
23 = 16 + 4 + 2 + 1 = 010111 (six bits is important)
101000 (invert all bits)
101001 (add 1)

Answer: 101001 (How can you verify this is -23?)

SYSTEMS AND NUMBERS

- Systems Classes require
  - Being able to quickly do powers of 2
  - Being able to convert between hex, binary, and decimal digits
  - 2s complement
- There are three quizzes online to help you review.

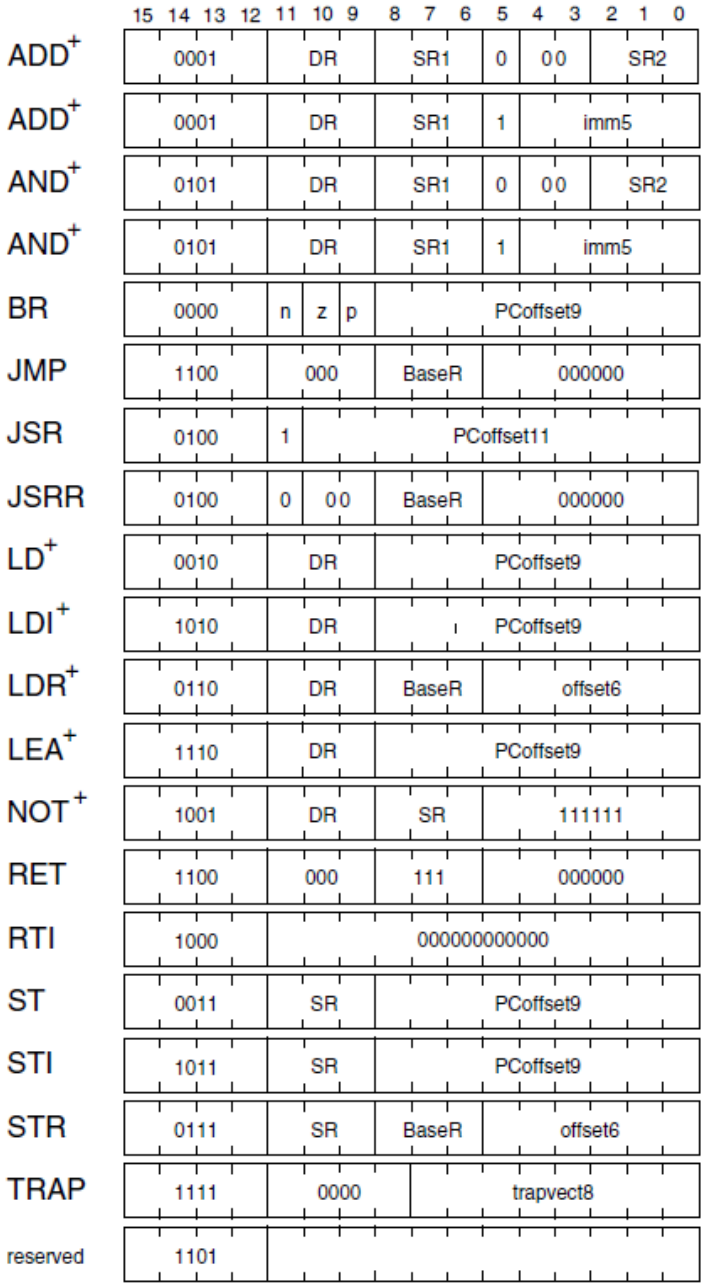
LC-3 ASSEMBLY LANGUAGE

PROGRAMMING

- Machine Language (object code) – The series of binary numbers that the computer can understand directly. May or may not be ready to run for various reasons.
- Assembly Language - A human readable (sort of) way to write code that the computer can easily turn into machine language.
- High Level Language – A MORE human readable (sort of) way to write assembly language.
  - GCC converts C and C++ to assembly then to object code.
  - Java does something a little different (bytecode and JRE)
- Executable Image – The machine language program AFTER it has been finished and is ready to run on the processor.

INSTRUCTIONS

- An instruction has two parts:
  - opcode : The operation being performed.
    - These are divided into three categories: Operate, Data Movement, and Control
    - LC3 has 15 opcodes (ADD, AND, BR, JMP, JSR, JSRR, LD, LDI, LDR, LEA, NOT, RET, RTI, ST, STI, STR, TRAP)
  - operands : The data to use in the operation.
    - This data is read from somewhere as specified by the instruction and the addressing mode
    - LC3 has 5 addressing modes (Immediate, Register, PC-Relative, Indirect, Base+Offset)



See appendix A

OPCODE SUMMARY

- Add - ADD value in a register to a value in a register. Store result in a register.
- Add - ADD value in a register to a number. Store result in a register.
- And - Bitwise AND value in a register to a value in a register. Store result in a register. (The term BITWISE is explained when we discuss AND.)
- And - Bitwise AND value in a register to a number. Store result in a register.
- Not – Invert all of the bits in a register and store the result in a register.
- BR, BRNZP – Conditional branch. Like if statement.
- LD, LDR, LDI – Load register with a value from memory
- ST, STR, STI – Store register value somewhere in memory.
- LEA – Load an address into a register and not the data.
- JMP – Jump execution to some other place in memory.
- JSR, JSRR – Jump to a subroutine similar to a function or method.
- RET – Return from the subroutine
- RTI – Return from interrupt, restores process information.
- TRAP – System call. Like an operating system. Handles input and output.

SIMPLE EXAMPLE PROGRAM

Labels	Opcode	Operands	Comment	Hex Value
	.orig	X3000	;Specify starting memory	3000
	GETC		;R0 <- Keyboard character	F020
	LD	R3, #9	;R3 <- -48	2609
	ADD	R1, R0, R3	;R1 <- R0 + R3	1203
	GETC		;R0 <- Keyboard character	F020
	ADD	R2, R0, R3	;R2 <- R0 + R3	1403
	ADD	R0, R1, R2	; R0 <- R1 + R2	1042
	NOT	R3, R3	; R3 <- ~R3	96FF
	ADD	R3, R3, #1	;R3 <- R3 + 1	16E1
	ADD	R0, R0, R3	;R0 <- R0 + R3	1003
	OUT		;Display <- R0	F021
	HALT		;End of program	F025
	.fill -48			FFD0
	.end		;End of file	

RUNNING A PROGRAM

- Program is loaded into memory at x3000
- Program Counter (PC) is set to x3000
- The instruction pointed to by the PC is loaded into the Instruction Register (IR).
- The PC is incremented at the same time as the loading of the IR.
- The instruction is processed and executed.
- Repeat until halt is reached.
- Note that some instructions change the PC which causes branching.

DATA TYPES AND CONDITION CODES

- An ISA can have different data types.
  - LC3 uses 2's complement numbers (Review of 2s complement below.)
- Condition codes are special flags that get set when certain operations occur.
  - LC3 uses N, P, and Z
    - N – is set if the result of an operation is negative.
    - P – is set if the result of an operation is positive (redundant?).
    - Z – is set if the result of an operation is zero.
  - Other common flags (that are not in the LC3)
    - C – the result of the carry out of an add operation
    - v – set in the even of an overflow condition

REGISTERS

- Eight general purpose registers (R0 – R7).
- Instruction Register (IR)– Holds the instruction in the microprocessor so it can be interpreted.
- Program Counter (PC) – Holds the address of the next program step. Looping is accomplished by changing the value of the PC.
- Memory Address Register (MAR) – When transferring to/from memory, this will hold the address of the memory unit to be written or read.
- Memory Data Register (MDR) – When transferring to memory, this holds the value to save. When transferring from memory, this holds the value read.
- Keyboard Data Register (KBDR) – Holds the ASCII character of the key after it has been typed.
- Keyboard Status Register (KBSR) – Status information that lets the microprocessor know a key has been typed and is ready to be read.
- Display Data Register (DDR) – Holds the ASCII character to display on the screen.
- Display Status Register (DSR) – Status information. Use this to tell the display you are ready to write the character from the DDR to the screen.
- Process State Register (PSR) – Information about the current state of the current program.

OPERATE INSTRUCTIONS

- On LC3 all operate instructions use register or immediate addressing mode.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Add

0	0	0	1	DR	SR1	0	0	0	SR2
0	0	0	1	DR	SR1	1	Immediate		

And

0	1	0	1	DR	SR1	0	0	0	SR2
0	1	0	1	DR	SR1	1	Immediate		

Not

1	0	0	1	DR	SR	1	1	1	1	1	1

BITWISE

- AND & NOT are bitwise operations
- Bitwise means it applies the operations you expect but applies the operation on each bit of a binary number.
- Bitwise operators ARE NOT logical. They DO NOT use Boolean argument.
- Bitwise AND uses single ampersand (&)
- Bitwise NOT uses the tilde (~)

AND EXAMPLE

- Assume four bit unsigned numbers.
- Each pair of bits from the two different values have the AND operator applied.
- For bitwise operations, 1 is true and 0 is false.
- And means they both have to be true for the output to be true

```
  1001          0011
 &1100         &0101
```

AND EXAMPLE

- Assume four-bit unsigned numbers
- Each pair of bits from the two different values have the AND operator applied.
- For bitwise operations 1 is true and 0 is false.
- And means they both have to be true for the result to be true

```
 1001 (9)          0101 (5)
&1100 (12)        &0011 (3)
 1000 (8)          0001 (1)
```

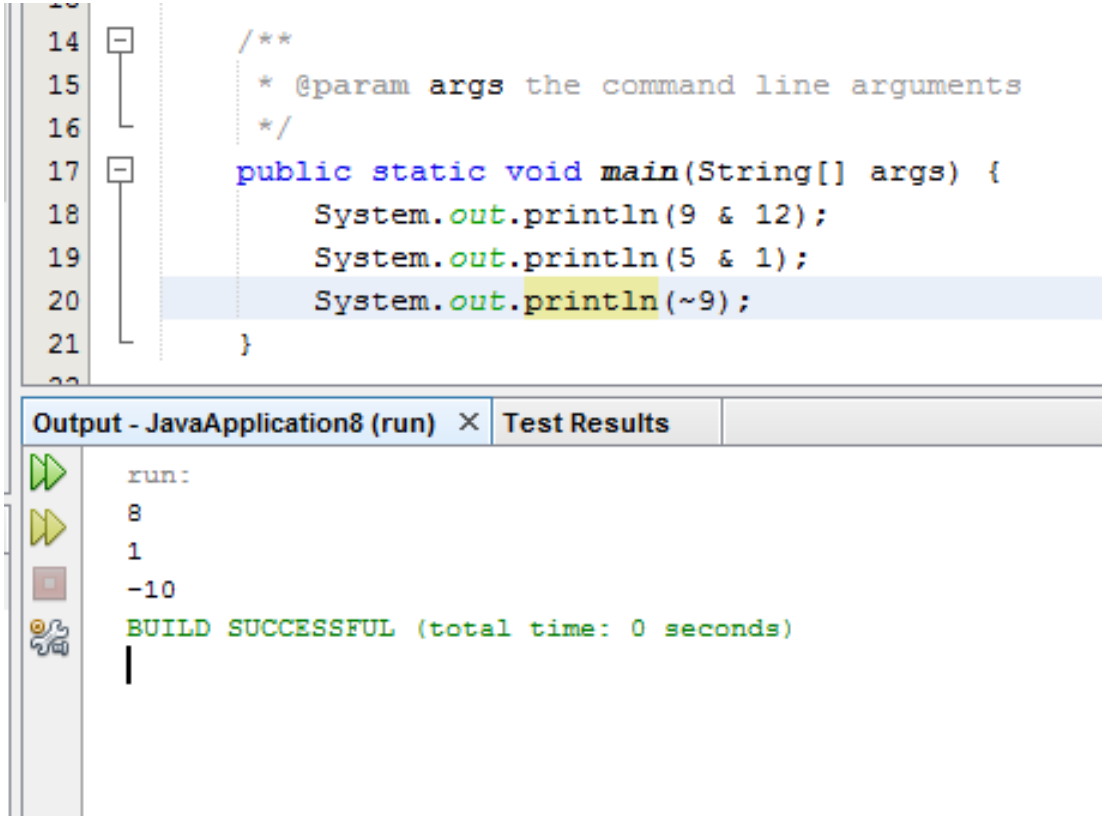
Try it in java:

```
System.out.println(9 & 12);
System.out.println(5 & 1);
```

NOT EXAMPLE

- NOT is a complement. ALL bits are inverted.
- Assuming 4-bit unsigned numbers.
- ~1001 (~9) becomes 0110 (6)
- What about Java? What would the following print?

```
System.out.println(~9);
```


14 /**
15 * @param args the command line arguments
16 */
17 public static void main(String[] args) {
18 System.out.println(9 & 12);
19 System.out.println(5 & 1);
20 System.out.println(~9);
21 }
Output - JavaApplication8 (run)
Test Results
run:
8
1
-10
BUILD SUCCESSFUL (total time: 0 seconds)

OPERATE INSTRUCTIONS

- With only ADD, AND, NOT…

  R1 = 5, R2 = 3, R3 = 0

  Perform the following (write the code in hex)

R1 <- R2 – R1	(Subtraction)
R1 <- R2 | R1	(Bitwise or)
R1 <- R2	(Transfer)
R1 <- 0	(Clear)

DATA MOVEMENT INSTRUCTIONS

- Moving data from
  - Register to memory
  - Memory to register
  - (Register to register is covered under operate instructions)
- Addressing modes
  - Immediate
  - PC-Relative
  - Indirect
  - Base + offset

PC RELATIVE

- Uses the current value of the program counter
- Adds a value specified in the instruction.
  - LD – Load value from memory and place it into a register.
  - ST – Stores register value in memory.
- 4-bit opcode, 3-bit register, 9-bit value
- Value is 9-bit 2's compliment (Range?)
- Effective address – The ACTUAL address of the operand. This is the value after all calculations and redirections have been made.

PC RELATIVE

- LD – Load value from memory and place it into a register.
- ST – Save value from register to memory

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			PC Offset 2's comp value added to PC								

0	0	1	0	0	1	0	0	1	0	1	0	1	1	1	1
LD				R2			x0AF								

- Load what is stored in memory location PC + x0AF into R2
- Assume this LD is stored at x3000, what is the effective address?

PC RELATIVE

- LD – Load value from memory and place it into a register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			PC Offset 2's comp value added to PC								

0	0	1	0	0	1	0	0	1	0	1	0	1	1	1	1
LD				R2			x0AF								

- Assume this LD is stored at x3000, what is the effective address? x30B0
- Remember the PC is incremented after loading the instruction.
- Assume x0035 is stored at memory location x30B0

PC RELATIVE – CALCULATE THE PC OFFSET

- ST – Store register value to memory location.
- Where in memory do you want to store the data?
- What is the PC when this instruction is executed. Note that the PC will ALWAYS be this instruction's address in memory + 1?
- The PC Offset (the thing stored in the instruction) is the difference.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			PC Offset 2's comp value added to PC								

0	0	1	1	0	1	0	0	1	0	1	0	1	1	1	1
ST				R2			x0AF								

PC RELATIVE – CALCULATE THE PC OFFSET EXAMPLE

- We want the program to the right to store R0 into memory location 300B
- The ST is at address 3002 therefore the PC will be 3003.
- The PC Offset is

  Storage Address – PC.
- PC Offset = 300B – 3003 = 8
- Store the number 8 as a 9-bit 2s complement number in the PC Offset field.

Address	Value
3000	AND R0, R0, #0
3001	ADD R0, R0, #5
3002	***ST R0, _____***
3003	HALT
3006	
3007	
3008	
3009	
300A	
300B	

Opcode				SR			PC Offset								
0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0
ST				R0			8								

EXAMPLE: ADD 3 NUMBERS

- Add three numbers.
  - Start program at x3000
  - Values 5,7,and 17 are stored at locations x3080, x3081, and x3082.
  - The sum will be stored in x3083
  - Only use PC-Relative addressing
    - LD (0010)
    - ST (0011)
  - Will also need add (0001) (Which add?)

FORMATS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LD (0010)				DR			2's comp value added to PC								
ST (0011)				SR			2's comp value added to PC								
Add (0001)				DR			SR1			0	0	0	SR2		
And(0101)				DR			SR			1	2's comp value				

- How can we add three numbers stored in memory?
- How many registers will we need? Minimize this as much as possible.
- I recommend zeroing registers before use.

R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	

3080	5
3081	8
3082	17
3083	?

How could we get the sum of these three numbers stored in memory location x3083?

You can only do one of the following:

- -Move memory to register
- -Move register to memory
- -Add two registers and store in a register

R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	

3000	
3001	
3002	
3003	
3004	
3005	
3006	
3007	
3008	

3080	5
3081	8
3082	11
3083	?

Clear R0
Load R1 with first number.
Add R0 to R1 and store in R0
Load R1 with second number.
Add R0 to R1 and store in R0
Load R1 with third number.
Add R0 to R1 and store in R0
Store R0 to memory.

HOMEWORK 1 & 2

- USE THE DOCUMENTS ON ASULEARN FOR DETAILS ABOUT THESE ASSIGNMENTS

INDIRECT ADDRESSING

- LDI – Load Indirect – Memory to Register
- STI – Store Indirect – Register to Memory
  - LDI and STI are very similar to LD and ST.
  - Memory calculation for operand is that same, however, the address calculated is NOT actually the operand.
  - The address calculated is the address of the operand. (TRICKY)
  - Indirect has one extra step and needs an extra memory access to get the operand.
- You need to learn how LDI and STI work.
- **YOU WILL NOT USE LDI OR STI IN YOUR PROGRAMS.**

LOAD INDIRECT

- LDI – Load value from memory and use it as the address of the actual data to move into a register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			2's comp value added to PC								

1	0	1	0	0	1	0	1	1	0	1	0	1	1	1	1
LDI				R2			x1AF								

- MAR <- PC + x1AF
- MDR <- MEM[MAR]
- MAR <- MDR
- R2 <- MEM[MAR]

STORE INDIRECT

- STI – Save value from register to address specified in memory location.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			2's comp value added to PC								

1	0	1	1	0	1	0	1	1	0	1	0	1	1	1	1
STI				R2			x1AF								

- MAR <- PC + x1AF
- MAR <- MEM[MAR] (Can this be done in one cycle?)
- MDR <- MEM[MAR]

Address	Value
3000	A403
3001	14A1
3002	B402
3003	FF25
3004	300A
3005	300C
3006	001A
3007	00B2
3008	C133
3009	2F25
300A	0005
300B	11E2
300C	0200

LDI R2, #3 (1010 0100 0000 0011)

The address is PC + 3 = 3001 + 3 = 3004

What is in address 3004?

Address	Value	
3000	A403	LDI R2, #3
3001	14A1	The address is PC + 3 = 3001 + 3 = 3004
3002	B402	What is in address 3004? 300A
3003	FF25	300A is the Effective Address
3004	300A	The operand is stored in 300A
3005	300C	What is in address 300A?
3006	001A	
3007	00B2	
3008	C133	
3009	2F25	
300A	0005	
300B	11E2	
300C	0200	

Address	Value
3000	A403
3001	14A1
3002	B402
3003	FF25
3004	300A
3005	300C
3006	001A
3007	00B2
3008	C133
3009	2F25
300A	0005
300B	11E2
300C	0200

R2	5

LDI R2, #3

The address is PC + 3 = 3001 + 3 = 3004

What is in address 3004? 300A

300A is the Effective Address.

The operand is stored in 300A.

What is in address 300A? 5

R2 will be loaded with 5.

Address	Value
3000	A403
3001	14A1
3002	B402
3003	FF25
3004	300A
3005	300C
3006	001A
3007	00B2
3008	C133
3009	2F25
300A	0005
300B	11E2
300C	0200

R2	6

ADD R2, R2, #1 (0001 0100 1010 0001

R2 <- R2 + 1

Address	Value	
3000	A403	
3001	14A1	
3002	B402	STI R2, #2 (1011 0100 0000 0010)
3003	FF25	Store R2 somewhere.
3004	300A	What is the address from the instruction?
3005	300C	
3006	001A	What is the effective address?
3007	00B2	
3008	C133	What gets stored where?
3009	2F25	
300A	0005	
300B	11E2	
300C	0200	

R2	6

Address	Value
3000	A403
3001	14A1
3002	B402
3003	FF25
3004	300A
3005	300C
3006	001A
3007	00B2
3008	C133
3009	2F25
300A	0005
300B	11E2
300C	0006

R2	6

STI R2, #2 (1011 0100 0000 0010)

Store R2 somewhere.

What is the address from the instruction?

3003 + 2 = 3005

What is the effective address?

300C

What gets stored where?

MEM[300C] <- 6

BASE REGISTER + OFFSET

- Add an immediate offset to some register.
- Easier to calculate for humans.
- LDR and STR
- Often a register will be dedicated being the memory base.
- You need to learn how LDR and STR work.
- DON'T USE LDR OR STR IN YOUR PROGRAMS.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OpCode				DR or SR			Base R			2's comp value					

0	1	1	0	0	1	0	1	1	0	0	0	1	1	0	1
LDR				R2			R6			#13					

LEA AND IMMEDIATE MODE

- Load Effective Address (LEA) – Puts an address into a register instead of data.
- Immediate because it doesn't access memory.
- Add 8-bit 2's complement immediate (part of the instruction) data to PC and store in register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OpCode				DR			2's comp value								

1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	0
LEA				R5			#-2								

ADDRESSING MODES SUMMARY

- Immediate – Data in instruction.
- Register – Data is in register.
- PC-Relative – Add immediate value to PC to get address of data.
- Indirect – PC-Relative gives address of the address of the data and NOT the data directly.
- Base + offset – Immediate value is added to an address from a register.

BRANCHING

- Branch (BR) – Changes execution order if conditions are met.
- Detects Negative, Positive, and Zero
- Works in conjunction with one of the following instructions.

  ADD, AND, NOT, LD, LDI, LDR, LEA
- If one of the above instructions results in a negative, positive, or zero value being written to a register, the appropriate flag (N, P, Z) will be set.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OpCode				N	Z	P	2's comp value								

0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	0
BR				N		P	#-2								

WHAT DOES THIS DO?

AND R0, R0, #0 (The # lets you know it is immediate)

AND R1, R1, #0

ADD R0, R0, #6

ADD R1, R1, R0

ADD R0, R0, #-1

BRP #-3

HALT (What is stored in R1?)

VIDEO ON IF AND THE ABOVE PROGRAM

I AM GOING TO GO THROUGH THIS. TAKE NOTES!!!

LOOPING AND BRANCHING

```
if (R2 > R1)          {
        R2 <- R2 + R1
}
```

We have to translate if(R2 > R1) into a BRNZP.

How? Subtraction.

In assembly language, it is easier to think of this as:

"IF THE CONDITION IS FALSE, JUMP OVER THE BLOCK OF CODE"

USING SUBTRACTION TO SET FLAGS

if(R2 > R1) {R2 <- R2 + R1}

Subtract R2 from R1. (R1 – R2)

What is the result if R2 < R1? N, Z, or P?

What is the result if R2 = R1? N, Z, or P

What is the result if R2 > R1? N, Z, or P?

If R2 < R1 (P) or R2 = R1 (Z) skip the add code.

```
NOT   R3, R2                  (Invert R2 and put in R3)
ADD   R3, R3, #1   (Add 1 to R3, R3 is now –R2)
ADD   R3, R1, R3   (Add R3 to R1. Same as R1 – R2)
BR_ _ _ #____                 (Jump over add if R2 <= R1)
ADD R2, R2, R1
```

RESULTS

if(R2 > R1) {R2 <- R2 + R1}

Subtract R2 from R1. (R1 – R2)

What is the result if R2 < R1? N, Z, or P?

What is the result if R2 = R1? N, Z, or P

What is the result if R2 > R1? N, Z, or P?

If R2 < R1 (P) or R2 = R1 (Z)skip the add.

```
NOT  R3, R2                (Invert R2 and put in R3)
ADD  R3, R3, #1 (Add 1 to R3, R3 is now –R2)
ADD  R3, R1, R3 (Add R3 to R1. Same as R1 – R2)
BRPZ #1                    (Jump over add if R2 > R1)
ADD R2, R2, R1
```

IF R0 > R1

- This is how the flags will be set for R0 – R1
  - if R0 is bigger the P flag will be set.
  - if R0 is smaller the N flag will be set.
  - if they are equal, the Z flag will be set

```
if (R0 > R1) {
        block1
}
```

  - We want to DO block1 if R0 > R1, which means we want to BRANCH to the code AFTER block1 when R0 <= R1
  - So we use BRNZ (N in case R0 is smaller and Z in case they are the same)

IF R0 < R1

- This is how the flags will be set for R0 – R1
  - if R0 is bigger the P flag will be set.
  - if R0 is smaller the N flag will be set.
  - if they are equal, the Z flag will be set

```
if (R0 < R1) {
        block1
}
```

  - We want to DO block1 if R0 < R1, which means we want to BRANCH to the code AFTER block1 when R0 >= R1
  - So we use BRZP (Z in case R0 is bigger and Z in case they are the same)

IF R0 == R1

- This is how the flags will be set for R0 – R1
  - if R0 is bigger the P flag will be set.
  - if R0 is smaller the N flag will be set.
  - if they are equal, the Z flag will be set

```
if (R0 == R1) {
        block1
}
```

  - We want to DO block1 if R0 == R1, which means we want to BRANCH to the code AFTER block1 when R0 >R1 or R0 < R1
  - So we use BRNP (P in case R0 is bigger and N in case R0 is smaller)

IF R0 != R1

- This is how the flags will be set for R0 – R1
  - if R0 is bigger the P flag will be set.
  - if R0 is smaller the N flag will be set.
  - if they are equal, the Z flag will be set

```
if (R0 != R1) {
        block1
}
```

  - We want to DO block1 if R0 != R1, which means we want to BRANCH to the code AFTER block1 when R0 == R1
  - So we use BRZ (Z means they were the same)

IF R0 >= R1

- This is how the flags will be set for R0 – R1
  - if R0 is bigger the P flag will be set.
  - if R0 is smaller the N flag will be set.
  - if they are equal, the Z flag will be set

```
if (R0 >= R1) {
        block1
}
```

  - We want to DO block1 if R0 >= R1, which means we want to BRANCH to the code AFTER block1 when R0 < R1
  - So we use BRN (N means R0 was smaller)

IF R0 <= R1

- This is how the flags will be set for R0 – R1
  - if R0 is bigger the P flag will be set.
  - if R0 is smaller the N flag will be set.
  - if they are equal, the Z flag will be set

```
if (R0 <= R1) {
        block1
}
```

  - We want to DO block1 if R0 <= R1, which means we want to BRANCH to the code AFTER block1 when R0 > R1
  - So we use BRP (P means R0 was bigger)

IF - ELSE

IF CONDITION FALSE BRANCH TO BLOCK 2

BLOCK1

UNCONDITIONAL BRANCH (BRNZP or BR) OVER BLOCK2 SINCE BLOCK 1 WAS RUN

BLOCK2

```
if (R1 > R2)
{
        R1 = R1 – R2  //Block 1
}
else
{
        R1 = R2 – R1  //Block 2
}
```

IF - ELSE

IF CONDITION FALSE BRANCH TO BLOCK 2

BLOCK1

UNCONDITIONAL BRANCH OVER BLOCK2 SINCE BLOCK 1 WAS RUN

BLOCK2

```
if (R1 > R2)
{
        R1 = R1 – R2  //Block 1
}
else
{
        R1 = R2 – R1  //Block 2
}
```

BLOCK 1 (R1 <- R1 – R2)

NOT R2, R2
ADD R2, R2, #1
ADD R1, R2, R1

BLOCK 2(R1 <- R2 – R1)

NOT R1, R1
ADD R1, R1, #1
ADD R1, R2, R1

IF - ELSE

IF CONDITION FALSE BRANCH TO BLOCK 2

BLOCK1

UNCONDITIONAL BRANCH OVER BLOCK2 SINCE BLOCK 1 WAS RUN

BLOCK2

High Level Language (Java or C)		Assembly

if (R1 > R2)
 	
{
R1 = R1 – R2 //Block 1
}
else
{
R1 = R2 – R1 //Block 2
}
	If R1 < R2
	or R1 = R2
	skip to
	BLOCK2
		NOT R3, R2
		ADD R3, R3, #1
		ADD R3, R1, R3
		BRNZ #4

	BLOCK1	NOT R2, R2
		ADD R2, R2, #1
		ADD R1, R2, R1

	skip BLOCK2	BRNZP #3

	BLOCK2	NOT R1, R1
		ADD R1, R1, #1
		ADD R1, R2, R1

CONSTANTS AND PRINTING

- Suppose you wanted to print the value in R0 but only if it was the character 'A'.
- We would want to subtract the ASCII code for 'A' from R0. If the result of the subtraction is zero, we know the character is an 'A' and we can print it.
- But how do we store the ASCII code for 'A'

STORING CONSTANTS

- We can simply store the constant as part of our program.
- We must be careful NOT to put the constant where it might get executed.
- A safe place is AFTER the HALT instruction.

```
F025   ;The program stops here
0041   ;The ASCII code for A
```

VARIABLE DATA MUST BE KEPT OUT OF THE INSTRUCTION AREA

- We can simply store the constant as part of our program.
- We must be careful NOT to put the constant where it might get executed.
- This is a problem because the LC3 will attempt to executethe 0041 thinking it is a branch.

```
0041   ;The ASCII code for A

F025   ;The program stops here
```

PRINT CHARACTER IN R0 IF IT IS 'A'

```
LD R1, ________ ; Load ASCII 'A' into R1
NOT R1, R1      ; Convert to negative step 1
ADD R1, R1, #1  ; Convert to negative step 2
ADD R1, R0, R1  ; R1 <- R0 - 'A'
BRNP ________   ; Branch if not zero
Print character; Trap for print (F021)
HALT             ; Trap for halt (F025)
0041             ; ASCII 'A'
```

What are the offsets needed?

HEX PRINT CHARACTER IN R0 IF IT IS 'A'

```
3000
2206    LD R1, 6          ; Load ASCII 'A' into R1
9240    NOT R1, R1        ; Convert to negative step 1
1261    ADD R1, R1, #1    ; Convert to negative step 2
1201    ADD R1, R0, R1    ; R1 <- R0 - 'A'
0A01    BRNP 1            ; Branch if not zero
F021    Print character;  Trap for print (F021)
F025    HALT              ; Trap for halt (F025)
0041                      ; ASCII 'A'
```

PRINTING A SINGLE CHARACTER

- Printing is a system subroutine.
- Printing is provided by the operating system.
- System subroutines are accessed with traps.
- The "print a character" subroutine/trap is called **OUT**.
- It has a trap vector of 0x21 so you will sometimes see it shown as TRAP 0x21.
- The hex will always be F021.
- It will print the ASCII code stored in R0.
- The value MUST be stored in R0. No other register will work.

```
OUT R1      ;DOES NOT WORK. WONT COMPILE.
OUT         ;Will print the character whose ASCII is in R0.
```

JUMPING AROUND

- JSR – Jump to Subroutine or Jump Save Return.
  - Return address (PC) saved in R7.
  - 11-Bit PC relative addressing.
- JSRR – Jump to Subroutine
  - Return address (PC) saved in R7.
  - Base register addressing.
- JMP – Jump without saving return.
  - Base register addressing.
- RET – Special form of JMP
  - Base register is always register R7.
- BRNZP or BR – Unconditional branch. BR and BRNZP both will always branch.
  - 9-Bit PC relative addressing.

TRAP

- GETC - Get a character from keyboard (F020).
  - No prompt is printed.
  - Execution stops and waits for a key to be pressed.
  - Store the ASCII code in R0.
- OUT - Send a single character to output (F021).
  - ASCII character must be stored in R0.
- PUTS - Print null terminated string to the output (F022).
  - ADDRESS of string must be stored in R0.
- IN - Get character from keyboard (F023).
  - Print prompt "Input a character>"
  - Stop and wait for a key to be pressed
  - Store the ASCII code in R0.
  - Prints the character to the display.
- PUTSP – Print null terminated string to the output (F024)
  - Same as PUTS but assumes two characters per word.
  - Address of string must be stored in R0.
- HALT – Stop execution of the processor (F025).

PRINTING STRINGS

HEX File	Assembly	
3000		;Where the program starts
E002	LEA R0, #2	;R0 <- Address of string
F022	PUTS	;DISPLAY <- String starting at R0
F025	HALT	;Stop the program
0048	H	;The string one character at a time
0065	e	
006C	l	
006C	l	
006F	o	
0000	/0	;NULL termination

- Note the use of LEA and PUTS to print strings.
- Use LD and OUT to print a single character.
- Use LEA and PUTS is required to print strings.
- **The string MUST be null terminated!!**

PRINTING STRINGS VS PRINTING A SINGLE CHARACTER

- Printing strings
  - Use LEA to load R0 with the ADDRESS of the first character
  - Use PUTS to print the string
- Printing a character
  - Use LD to load R0 with the ASCII CODE of the character.
  - Use OUT to print the character.

ADDRESSING MODES SUMMARY

- Immediate – Data is part of the instruction.
- Register – Data is in a register.
- PC-Relative – Add immediate value to PC to get address of data.
- Indirect – PC-Relative calculation gives address of the address of the data and NOT the data directly.
- Base + offset – Immediate value is added to an address from a register.

OPCODE SUMMARY

- Add - ADD value in a register to a value in a register. Store result in a register.
- Add - ADD value in a register to a number. Store result in a register.
- And - Bitwise AND value in a register to a value in a register. Store result in a register.
- And - Bitwise AND value in a register to a number. Store result in a register.
- Not – Invert all the bits in a register and store the result in a register.
- BR, BRNZP – Conditional branch. Like if statement.
- LD, LDR, ***LDI**** – Load register with a value from memory
- ST, STR, ***STI**** – Store register value somewhere in memory.
- LEA – Load an address into a register and not the data.
- ***JMP**** – Jump execution to some other place in memory.
- JSR, ***JSRR**** – Jump to a subroutine similar to a function or method.
- RET – Return from the subroutine
- ***RTI**** – Return from interrupt, restores process information.
- TRAP – System call. Like an operating system. Handles input and output.

Don't use instructions marked with * in programs in this class.