Good luck on the final everyone!
Extra study guide I made here

CS 2110

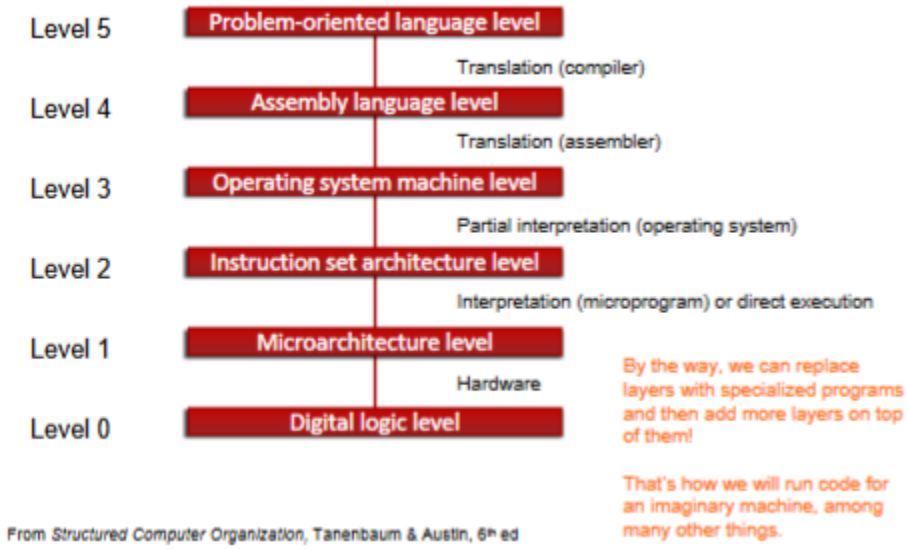
Good luck on the final everyone!
Extra study guide I made here

L01 - Course Intro & Objectives

- Thus far, our learning has been primarily about the "blackbox," we input one thing and something gets outputted but we aren't familiar with the things that occur in between
- 2110 is all about building from the bottom up and understanding what happens inside the blackbox

Big Ideas

- #1: All computers can compute the same kinds of things
  - Ex. Turing machine, small appliances, smartphones, stored program computers, regular expressions, automata theora, formal grammars, etc.
  - Some might be faster or more efficient than others, but they can all do the same thing
  - We call this big idea **Turing-equivalence**
  - Just about everything that we use for computation can be proved capable of solving the same set of problems
- #2: Abstraction: Layers making the electrons work



  - This class will start from the bottom up
- #3: Binary
  - Binary is "better" than decimal for the purpose of building an electronic computer
  - This is because of lots of small physical and economic reasons:
    - It's easier to determine presence/absence of current rather than magnitude
    - Can use lower voltages to distinguish only 0/1 instead of 0/1/…/9, so less power
    - Binary-coded decimal math makes more circuitry than pure binary
- #4: Computers store representations of something outside
  - Computer's can't store the mathematical abstraction we call a "number"
  - Everything in a computer is a finite-sized **representation** of something outside

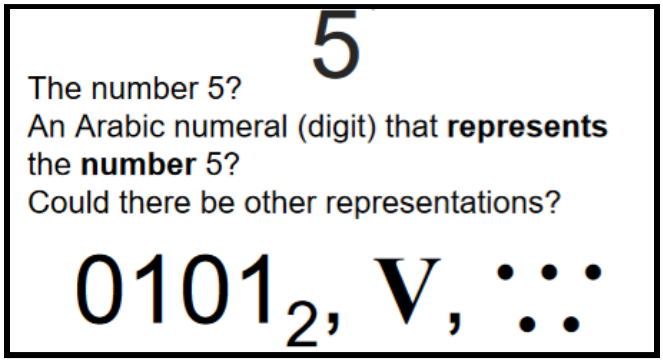
Good luck on the final everyone!
Extra study guide I made here

        - Think of numbers. Number's are an abstract concept in our minds, and the highest number we can think of is whatever we want. In a computer, because our abstraction is made physical, there will always be a bigger number than what the computer can store
    - A bunch of binary digits (bits) is always interpretable as an unsigned whole number. We use that representation often
    - So we can always claim the bigs stored in a computer represent a positive whole number

Good luck on the final everyone!
Extra study guide I made here

L02 - Datatypes 1

- Modern computers effectively store patterns of switches that are set On (1) or Off (0)
- The bits are just bits and we get to choose what we want them to mean; we **choose** representations for numbers, letters, and many other things that are "easy"to work with
- Bits don't have any inherent meaning, the designers choose what they want them to mean



- A **data representation** is a set of values from which a variable, constant, function, or other expression may take its value and includes the meaning of those values. A representation tells the compiler or interpreter how the programmer intends to use it.
    - Ex. The process and result of adding two variables differs greatly according to whether they are integers, floats, or strings
- We say a particular representation is a **data type** if there are operations in the computer that can operate on information that is encoded in that representation
- A **bit** (short for "binary digit") can take exactly two values. We refer to these values as 0 and 1.
- The first computers did decimal arithmetic, primarily because we as humans think in base 10. Historically, these computers were clunky and did not function well. Binary was determined to be the most effective way to store and calculate.
- How many different numbers can be represented by *n* bits?
    - 2^n
- There are 16! different ways to represent the numbers up to 15. That's nearly 20 million different ways to represent them. Here's one of them…
- There's a problem though, this is a terrible way to go about doing this because there's no *pattern*.
- Here's what we should use…

0	0 1 0 0
1	1 0 0 1
2	1 1 0 0
3	0 0 0 1
4	1 0 1 1
5	0 0 1 1
6	1 1 1 0
7	0 1 0 1
8	0 1 1 0
9	0 0 0 0
10	1 1 0 1
11	0 1 1 1
12	1 0 0 0
13	1 1 1 1
14	1 0 1 0
15	0 0 1 0

Good luck on the final everyone!
Extra study guide I made here

0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
10	1 0 1 0
11	1 0 1 1
12	1 1 0 0
13	1 1 0 1
14	1 1 1 0
15	1 1 1 1

- If we're going to be talking about bits, then we're gonna need to talk about the different powers of two.

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256

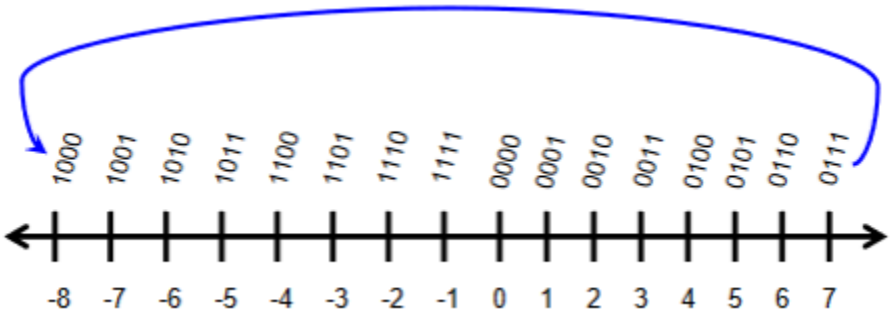
2^9	512
2^{10}	1,024
2^{11}	2,048
2^{12}	4,096
2^{13}	8,192
2^{14}	16,384
2^{15}	32,768
2^{16}	65,536

Unsigned Whole Numbers & Signed Integers (2's Complement)

N = 4	Number Represented			
Binary	Unsigned	Signed Mag	1's Comp	2's Comp
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

Good luck on the final everyone!
Extra study guide I made here

- We're used to dealing with whole numbers as both positive and negative, and working with bits we need a way to represent a negative number. How do we do this?
    - *Signed Magnitude* - Reserve a bit to represent the sign of the number (0 for positive, 1 for negative). The problem with this approach means we can have a -0, which doesn't work.
    - *1's Complement* - To make a negative number, just flip all the bits. We run into our -0 problem again though, so this won't work
    - *2's Complement* - Uses **modular arithmetic** to wrap around the number line so we can avoid the -0 issue.



        - The patterns of the numbers are consistent across any number of bits…
            - The highest positive integer is 0 followed by all 1's
            - The highest negative integer is 1 followed by all 0's
            - -1 is represented as all 1's
        - The range of our number line can be found through the following equation…

$$-2^{n-1} \; to \; 2^{n-1} - 1$$

        - In line with the other approaches, a 1 in the left-most bit represents a negative

2's Complement

- To negate a number, remember:
    - Flip the bits
    - Add one
- Example…
    - 6 = 0110
    - -6 = 1001 + 1 = 1010
- Because you can easily find the negative of a number, we don't even need to know how to subtract two bits, we can simply take the negative and add them together
    - Ex. 8 - 6 becomes 8 + (-6)
- Decimal Conversion from Signed Binary
    - Given a negative number in 2's complement, we can convert it to decimal by doing the following…
        - Flip the bits

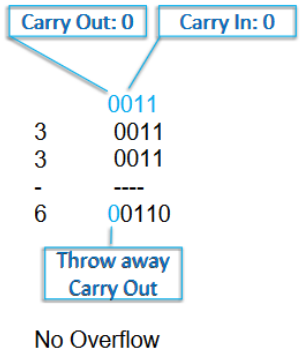
Good luck on the final everyone!
Extra study guide I made here

- Add one
- Convert to decimal
- The number is the negative of that decimal num

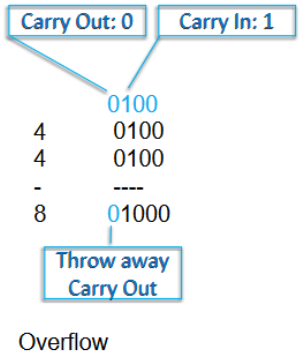
- If it's a positive number, you can simply use your knowledge of the powers of 2 to figure it out

Overflow (2's Complement)

- Because 2's Complement has a range, if we don't have enough bits to represent a number we have an **overflow**.
- To know if we have overflow, we have to look at the *carry in* and *carry out*
- If we add two numbers, we have a problem if…
  - Two positive numbers and we get a carry into the sign bit but no carry out

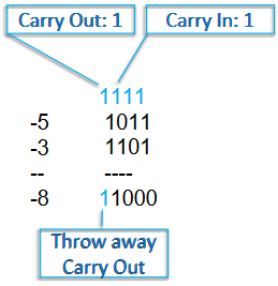


No Overflow

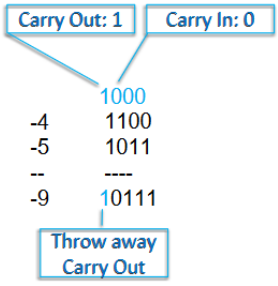


Overflow

  - Two negative numbers and we don't get carries into and out

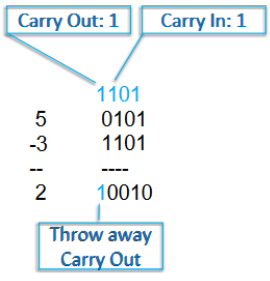


No Overflow

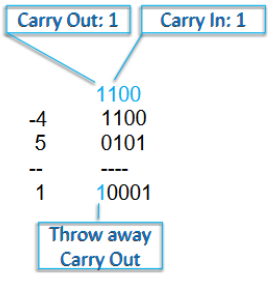


Overflow

- Basically, if carry-in =/= carry-out we have overflow
- If we add a positive and negative number we'll never have a problem…



No Overflow



No Overflow

Good luck on the final everyone!
Extra study guide I made here

Sign Extension

- Suppose we have a four bit number and an eight bit number and we wanted to add them.
- If we have a positive number, we put 0's to the remaining places in the 4-bit number and add them

Add 3 and 64 (0011 and 01000000)

```
  0011          0000 0011
+0100 0000    +0100 0000
 --------      --------
 0100 0011     0100 0011
```

- If we have a negative number, we put 1's to the remaining places in the 4-bit number and add them

Add -3 and 64 (1101 and 01000000)

```
  1101          1111 1101
+0100 0000    +0100 0000
 --------      --------
 0100 1101     10011 1101
```

Shifts in Binary

- In base 10, if we shift a number over, this is the same as multiplying a number by 10
  - Ex. 79 → 790 = 79 * 10
- In base 2, a binary left shift is the same thing, except it multiplies the number by 2 for each shift x << 2 equals to x*4
- If we do a right shift, it's the same thing as dividing by 2 or multiplying by ½.

```
01110011    01010101    00011100    00000001
01110011    01010101    00011100    00000001
--------    --------    --------    --------
11100110    10101010    00111000    00000010
```

A + A is the same as **2 * A** is the same as **A << 1**

Fractional Binary Numbers

```
8   4   2   1 ^ .5   .25  .125  .0625
1   0   1   0   1    1     0     0
```

$1010_{\wedge}1100 = 10.75_{10}$

The result of a Left Shift operation is a multiplication by 2^n , where n is the number of shifted bit positions.
The result of a Right Shift operation is a division by 2^n , where n is the number of shifted bit positions. So for example, A >> 1 = A / 2^1 = A / 2

Good luck on the final everyone!
Extra study guide I made here

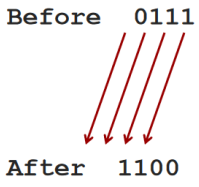
L03 - Datatypes V2 : Electric Boogaloo

Bitwise Operations

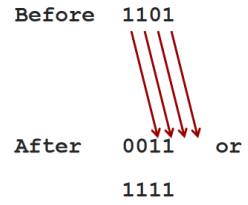
- When we have two strings of bits, we often apply boolean functions to pairs of respective bits in the two strings. These operations are considered “bitwise” operations.

Operation	Example	Truth Table / Notes
AND (&)	0101 0110 0100	A | B | A AND B / A & B / AB 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1
OR (|)	0101 0110 0111	A | B | A OR B / A | B / A + B 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 1
NOT (~) / Complement	0101 1010	A | NOT A / ~A / A’ 0 | 1 1 | 0
XOR (^) / “or but not both”, determines if two bits are different	0101 0110 0011	A | B | A XOR B / A ^ B 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 0

Good luck on the final everyone!
Extra study guide I made here

<table>
<tr>
<td>NAND</td>
<td>
0101
0110
1011
</td>
<td>
<table>
<tr><th>A</th><th>B</th><th>A NAND B
~(A & B)</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
</tr>
<tr>
<td>NOR</td>
<td>
0101
0110
1000
</td>
<td>
<table>
<tr><th>A</th><th>B</th><th>A NOR B
~(A | B)</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
</tr>
<tr>
<td>Left Shift (<<)</td>
<td>
0111 Leftshift 10 7 << 2


</td>
<td>If we run out of space, they fall off the left-side.

If we create holes, we need to replace them with zeroes</td>
</tr>
<tr>
<td>Right Shift (>>)</td>
<td>
1101 Rightshift 10 13 >> 2


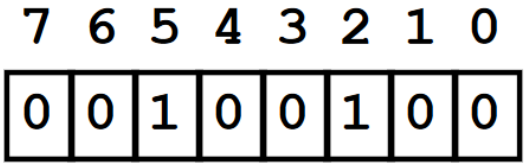
</td>
<td>Two types
Logical Right Shift - Fill in holes with 0's (works with unsigned whole numbers)
Arithmetic Right Shift - Fill in holes with the signed bit (works with 2's Complement #'s)</td>
</tr>
</table>

for non-negative numbers, we get the same result when we use either >> or >>> as shown in the first 11 rows in the output. However, when we divide negative numbers, we need to be sure that we are using the arithmetic right shift >> so that the compiler knows to interpret the dividend as a signed number and yield the correct results. If we use the logical right shift >>> with the intention to divide negative numbers, the compiler will assume those numbers to be unsigned and will just treat the sign bit as a normal bit and thus will not give you the result that you would expect.

Good luck on the final everyone!
Extra study guide I made here

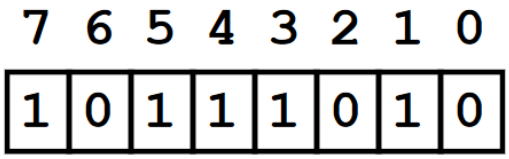
Bit Vectors (3rd data type)

- By convention, the right-most bit is the 0th bit, and every proceeding bit to the left goes up by 1.
- A bit vector is basically an array of bits that we describe as booleans and we can use our binary operations on.



- Suppose we create an empty bit vector…

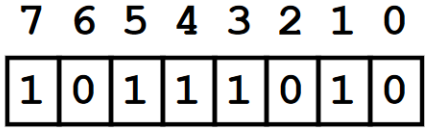
```
w = 0                          // all bits cleared to 0
w = w | 0b10000000             // Doc
w = w | 0b00100000             // Happy
w = w | 0b00010000             // Sneezy
w = w | 0b00001000             // Bashful
w = w | 0b00000010             // Dopey
```



- A **bit mask** is used in conjunction with a binary OR or binary AND to set a value in a bit vector. Ex. $_{0b}$10000000 in the top example.
- This can also go the other way; instead of starting the bit vector with all 0's we can start it with all 1's and work "backwards"

Alternately,

```
w = ~0                         // all bits set to 1
w = w & 0b10111111             // Sleepy
w = w & 0b11111011             // Grumpy
```



Good luck on the final everyone!
Extra study guide I made here

Manipulating Bits

x = x & ~077 Clears last six bits of x to zero! Because 077 is octal, it's 111111 in binary, ~077 is 000000, which clears the last six bits of x to zero.
x = x & ~077 is better than x = x & 0177700 cuz the second not only clears the last 6 bits of x to zero if x is over 16 bits, but the first is independent of word length

- There are a couple of ways we can manipulate the bits by abusing the identities of & and |
  - *CLEAR* :: $wxyz_2$ & 1111_2 == $wxyz_2$, so put a 0 in any bit you want to clear
    - $wxyz_2$ & 1101_2 == $wx0z_2$
  - *SET* :: $wxyz_2$ | 0000_2 == $wxyz_2$, so put a 1 in any bit you want to set
    - $wxyz_2$ | 0100 == $w1yz_2$
  - *TOGGLE* :: $wxyz_2$ ^ 1111_2 = $w'x'y'z'_2$, so put a 1 in any bit you want to toggle
  - *TEST* :: $wxyz_2$ & 0010_2 == $00y0_2$, then we can test $00y0_2$ == 0000_2
- To put a 1 in any bit position *n* in a mask, shift left by *n*
  - 1 << 2 == 0100_2
- To put a 0 in a position in the mask, put a one in that position and complement
  - ~(1 << 2) == 1011_2

Hexadecimal & Octal

- Binary is useful but incredibly tedious to write out
- If we change our base to one that's larger but also a power of 2, we can save a lot of space and work
- Two types…
  - *Octal*

Base 2	000	111	010	100	101
Base 8	0	7	2	4	5

↗ $00111010100101_2 = 07245_8$

  - *Hexadecimal*

Base 2	1000	1111	0011	1100	0001
Base 16	8	F	3	C	1

↗ $1000111100111000001_2 = 8F3C1_{16}$

  - For digits 10 - 15, we can just use the alphabet: A - F
- In order to differentiate the octal from hex from decimal numbers, there are certain conventions that are followed…
  - 456 is *decimal*
  - **0**456 is octal (we use a leading 0 to indicate it's octal)
  - **0x**456 is hexadecimal (we use a leading 0x to indicate it's hexadecimal)

0b1010 is binary (we use a leading 0b to indicate it's binary)

Good luck on the final everyone!
Extra study guide I made here

ASCII (4th data type)

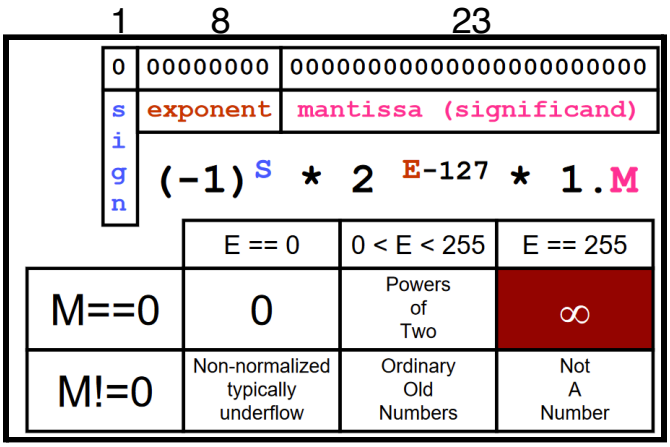
- ASCII are codes for printable characters

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	&#041;	)	73	49	111	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	&#042;	*	74	4A	112	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	&#043;	+	75	4B	113	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	&#044;	,	76	4C	114	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055	&#045;	-	77	4D	115	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

asciichars.com

Floating Point Numbers

- Typical implementations might range from 32 up to 128 bits
- IEEE standard
- Normalized mantissa : in scientific notation, this is the number that is being multiplied by a power of 10. By convention, the first digit cannot be 0.
- Because of this convention, in base 2, the first number in front of the decimal must always be a 1



where *E* is the exponent, *M* is the mantissa, and *S* is the sign bit

Good luck on the final everyone!
Extra study guide I made here

Not a Number (NAN)

- Suppose A is a floating point number set to NaN
- A != B is true when B is another floating point number, NaN, infinity, or *anything*
    - As such, A != A is **true** as well; NaN is never equal to itself
- If A or B is NaN, the following are always *false*
    - A < B, A > B, A == B

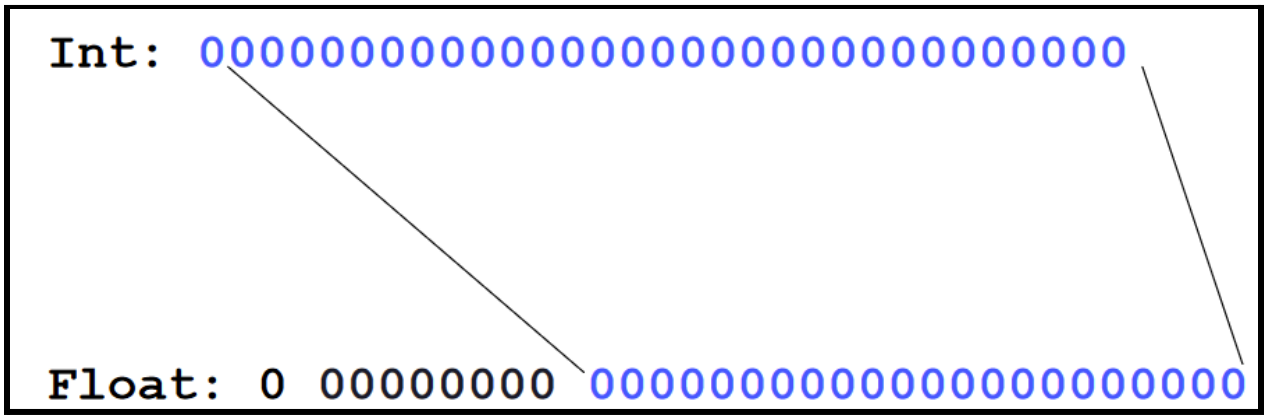
Non-Normalized

- Sometimes, we want to represent *incredibly* small numbers, this is when we used the non-normalized form
- The equation for the non normalized form is…

```
Non-normalized formula: (-1)^S * 0.M * 2^(-126)
note: 0.M and 2^(-126) instead of 1.M and 2^(-127)
```

Conversions

- Ints have 31 significant digits (31 bits for the number, 1 for the sign bit)
- As we've seen above, floats only have 23 bits (or 24 if you consider the implied 1 behind the mantissa)



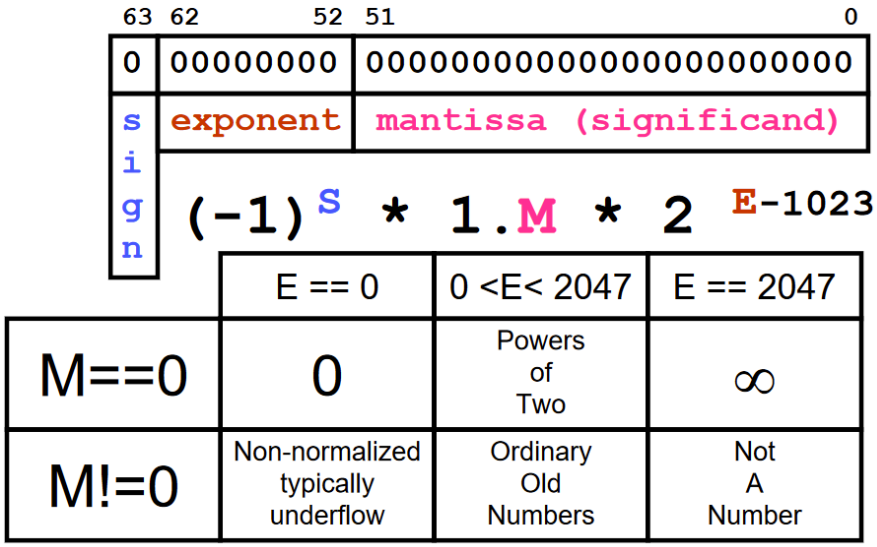
As you can see, we lose some precision when converting from int -> float

- So, if we convert an int to a float then back to an int, our information will be changed. If only floats had more bits…

IEEE Double

- A double has 64-bits including the sign bit, the exponent, and the mantissa.

Good luck on the final everyone!
Extra study guide I made here



	E == 0	0 <E< 2047	E == 2047
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Comparing Floating Point Numbers

- Here are the rules regarding floating point comparisons…
  - If either is NaN, the comparison is defined as "unordered"
  - If either is -0.0, replace with +0.0
  - If the sign bits are different (one is positive, one is negative), the positive one will always be bigger
  - If they are the same, compare the rest of the bits as integers
    - Go left to right and compare the two magnitudes bit by bit until we find one with different values; the number with the 1 bit in that difference slot will be greater

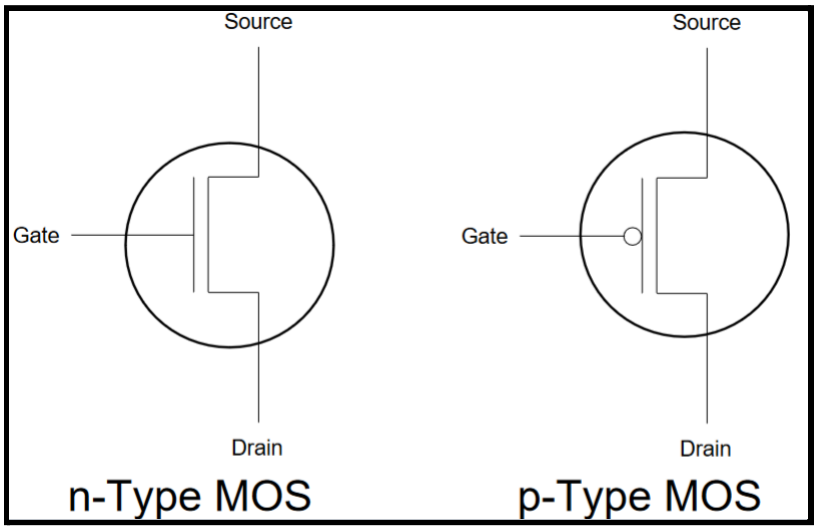
What we need to know

- Given the 6-case chart of FP numbers..
  - Know what each case means and recognize encoded FP numbers that fit each case
  - Know that in the 32-bit form the exponent is biased by 127
  - From an encoded FP number, be able to show its value in the form of $M * 2^E$
  - Compare encoded FP numbers for <, >, ==, !=
- Understand
  - Converting decimal FP to IEEE-754 and vice-versa
  - Precision issues in converting between integer and FP representations

Good luck on the final everyone!
Extra study guide I made here

L04 - Digital Logic

- A computer is essentially a collection of **switches**
- These switches were utilized in **relays**, and made the foundation for modern day computers. However, relays are slow, wear out, big, and noisy.
- To fix this, we came up with *diodes* and *triodes*, which functioned like a switch but with no moving parts. The problem with using diodes and triodes was that they required a vacuum, high power consumption, unreliable, and expensive.
- Eventually, we landed on **transistors**. Two types…



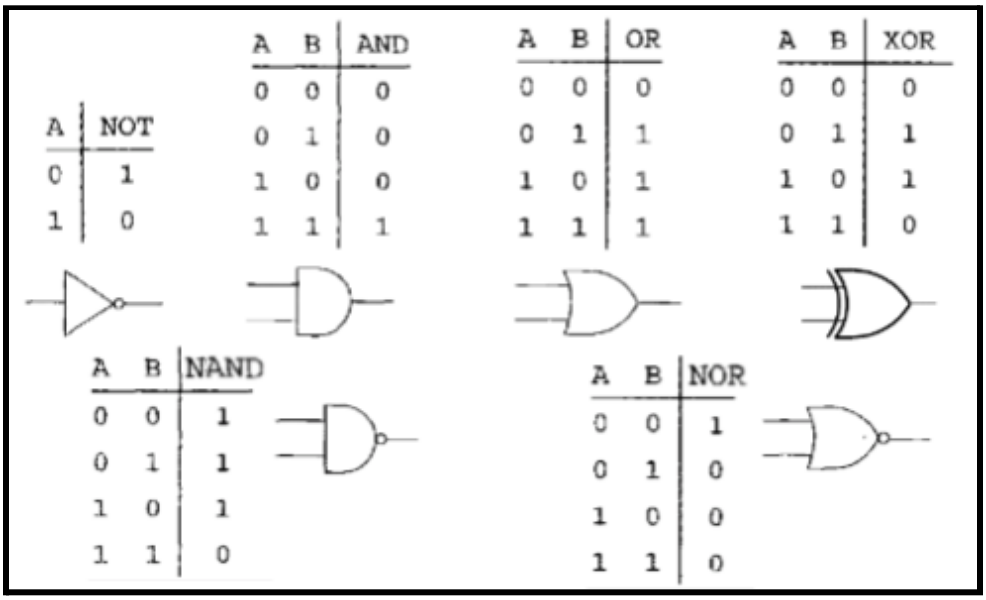
- *N-type*: Normally open; with no volts to the switch, the switch will remain open, meaning it will not connect the source to the drain. With volts, it will "activate" and the gate will be closed to connect the source and the drain
    - Connected to grou**N**d, propagates a strong "1" signal
- *P-type*: Normally closed; with no volts to the switch, the switch will remain closed, meaning it connects the source and the drain. Complement to *n*-type.
    - Connected to **P**ower, propagates a strong "0" signal

A Common Misconception

- Digital wires have **3** states, **not** 2.
    - A wire with some designated voltage (e.g. +2.9) can represent a logical 1
    - A wire with some designated violated (e.g. 0 volts or ground) can represent a logical 0
    - A wire that is not connected to 2.9 volts or ground is said to be **floating** or in a **high impedance state**, and its value can randomly vary from a logical 0 to 1

Good luck on the final everyone!
Extra study guide I made here

Logical Operations with Gates

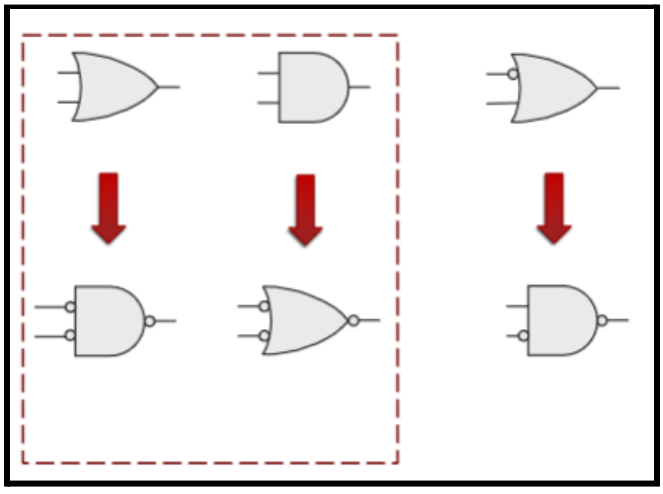


Chips: Building Blocks

- Transistors are big, use a lot of material, use a lot of power and generate a lot of heat, but we need a lot of them. So, we use **chips**, or **integrated circuits**.
- Chips are fast, inexpensive, and easy to work with
- Each chip contains 1 - 6 gates. Complicated designs can require thousands of gates and hundreds of chips

Conte Bubble Theorem

- This is how we apply *DeMorgan's Law* to circuits.

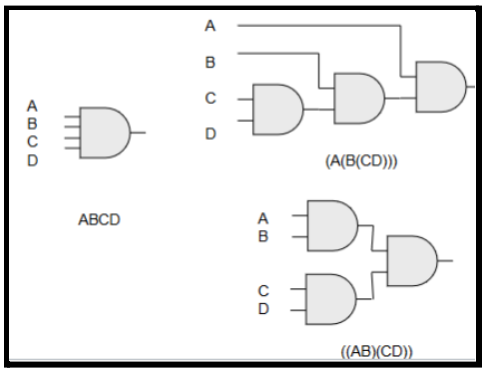


Conte Bubble pushing technique:
1. Change the logic gate (AND to OR and OR to AND).
2. Add bubbles to the inputs and outputs where there were none, and remove the original bubbles.

Good luck on the final everyone!
Extra study guide I made here

Gates with More Than 2 Inputs

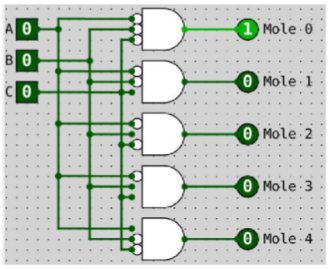
- So far, our boolean logic has consisted of working at 2 boolean values at a time
- We can actually work on more than 2 inputs at a time, we just need to take the inputs two at a time.



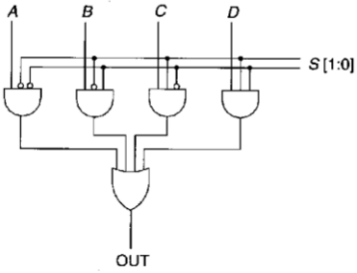
- Which one should we do: the top or bottom one on the right?
  - The bottom one, because the two gates would occur at the same time, meaning it would take 2 time units instead of 3.

Combinational Logic — The same inputs always produce same output

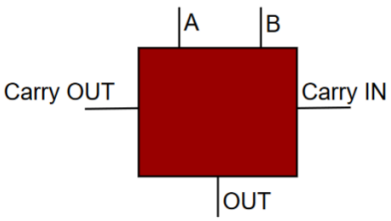
- A combination AND, OR, and NOT gates that are used to build higher-level items
- A truth table uniquely defines combinational logic; if you know the truth table, you know the combination of logic used.
- Decoder
  - A **decoder** decodes an *n* bit binary number, and uses an AND gate and NOT bubbles to decode
    - If there are *n* input bits, there are 2^n outputs



- Multiplexor (MUX) [Input Selector]
  - Say we have *m* signals and we want to select one of them. For this, we use a **multiplexor**, which is essentially a *selector*.
  - We use the *selector bits* to determine which signal we want.
    - If we have 2^n input bits, we need *n* selector bits



- Full Adder
  - Takes an input of bits and outputs their sum.



Good luck on the final everyone!
Extra study guide I made here

Truth Table to Circuits

- Suppose you are given *n* inputs. If you use a decoder of size 2^n, and then use the # of OR gates equal to the number of outputs, you can implement a truth table in circuits.

Boolean Simplification

Basic Boolean algebraic properties

Additive	Multiplicative
$A + B = B + A$	$AB = BA$
$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
$A(B + C) = AB + AC$	

Basic Boolean algebraic identities

Additive	Multiplicative
$A + 0 = A$	$0A = 0$
$A + 1 = 1$	$1A = A$
$A + A = A$	$AA = A$
$A + \overline{A} = 1$	$A\overline{A} = 0$

$\overline{\overline{A}} = A$

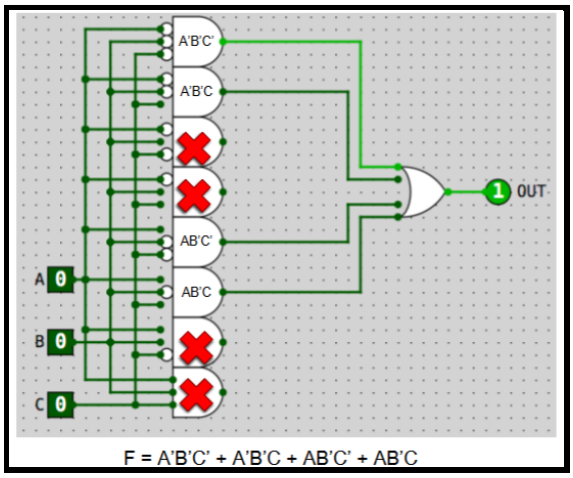
Useful Boolean rules for simplification

$A + AB = A$
$A + \overline{A}B = A + B$
$(A + B)(A + C) = A + BC$

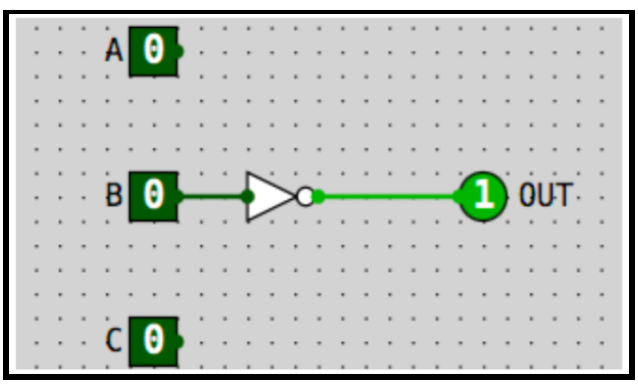
DeMorgan's Laws

$\overline{AB} \rightarrow \overline{A} + \overline{B}$
$\overline{A + B} \rightarrow \overline{A}\,\overline{B}$

- Suppose we have a circuit. We can convert it into a boolean expression. This will be useful to us because in boolean logic, it's easy to determine how to simplify…



- Classic Simplification
  - F = A'B'C' + A'B'C + AB'C' + AB'C
  - F = A'B'(C'+ C)+AB'(C' + C)
  - F = A'B' + AB'
  - F = B'(A' + A)
  - F = B'
  - Huge simplification, right?



Good luck on the final everyone!
Extra study guide I made here

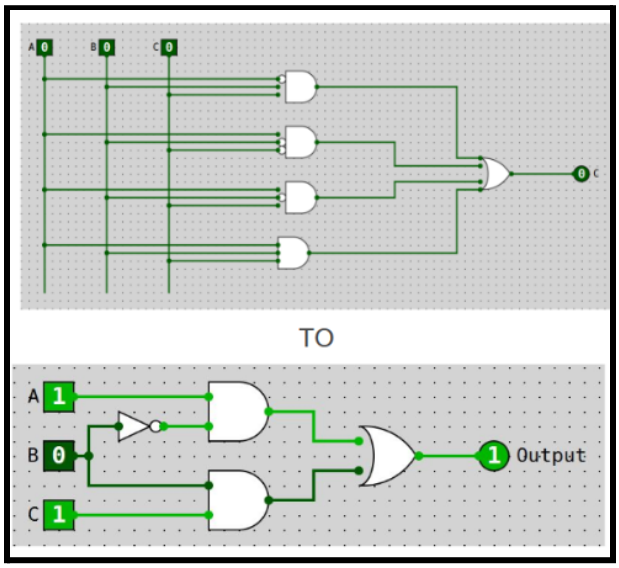
Gray Code Sequence

- The one we need to know →
- Used in the heading

00
01
11
10

Karnaugh Maps

- Method of simplifying boolean expressions by grouping related items
- Results in the simplest sum-of-products expression
- The boxes have to be a power of two because the power they are raised to is the number of variables reduced in the reduction step (i.e. a box of size 2^2 will remove 2 variables)
- K-Maps can go up to many, many variables but they get kinda whacky, but we'll only ever work with k-maps up to 4 variables.
- Steps for making one…



1. Create a K-Map - Distribute variables across the rows and columns using gray code order, then fill in the corresponding entries.

	AB	AB'	A'B'	A'B
C				
C'				

	AB	AB'	A'B'	A'B
C	1	1	0	1
C'	0	1	X	0

2. Make the Groupings

Good luck on the final everyone!
Extra study guide I made here

	AB	AB'	A'B'	A'B
C	1	1	0	1
C'	0	1	X	0

	AB	AB'	A'B'	A'B
C	1	1	0	1
C'	0	1	X	0

Rules of K-map grouping:

1. Groups must be rectangular
2. Groups may wrap around edges!
3. Groups may only contain "1"s or "X"s
4. All "1"s must be contained within at least one group
5. Groups must be as large as possible
6. The size of a group must be a power of 2 – note that you may have a group of size 1

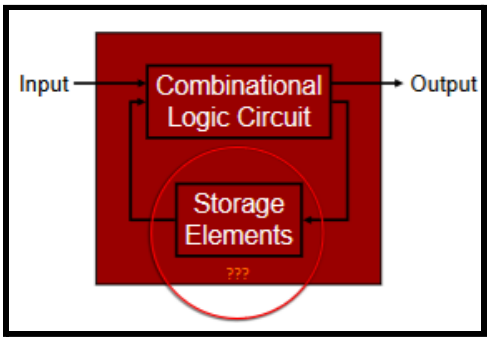
3. Write the Simplified Expression

Good luck on the final everyone!
Extra study guide I made here

L05 - Digital Logic V2 : Electric Boogaloo V2

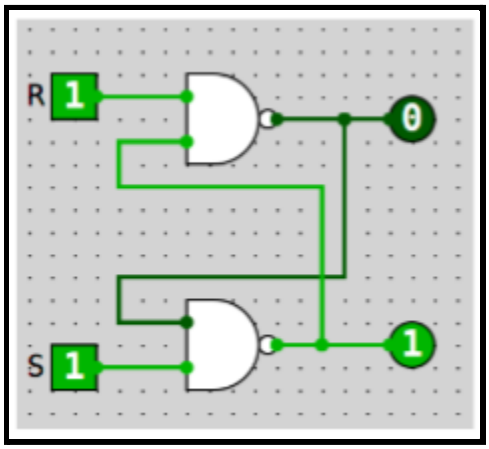
Combinational Logic vs. Sequential Logic

- Combinational
  - Combination of AND, OR, NOT, NAND, and NOR
  - Same inputs always produce same output
  - Analogous to a cheap bike lock (doesn't matter the order I put in the numbers, I could put in the 2nd number before the 1st, but so long as they all get to their number, it's fine)
- Sequential
  - Requires *storage elements* (memory)
  - Output depends on inputs plus **state**
  - Analogous to a RLR combinational lock (you have to put in the right numbers in the right order)
  - Used to build memory & **state machines**



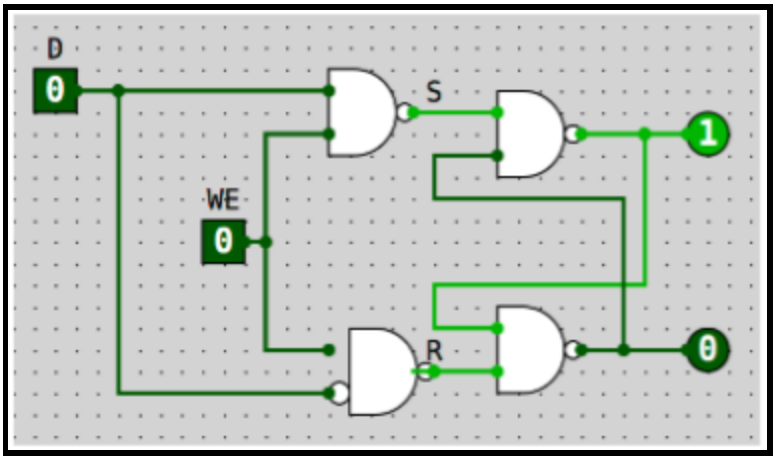
R/S Latch & D Latch

- We can store bits using circuits.
- The R/S latch is the simplest form of this; depending on the input, we can store a 0 or a 1 in the top right bit.

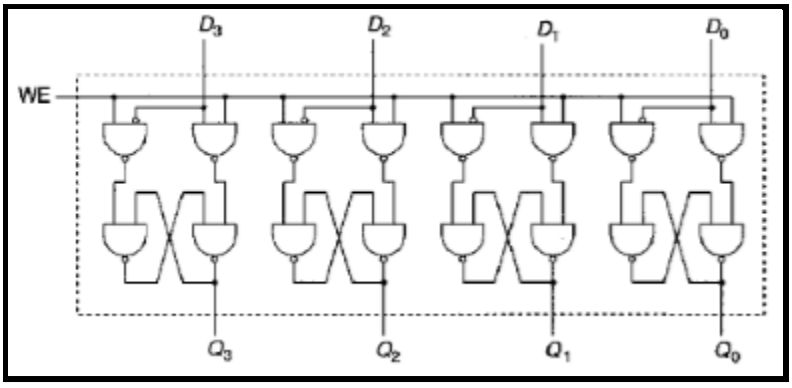


- However, there's a problem with the R/S latch. If both our inputs are 0, the values are invalid. Instead, we can use a D Latch, which alleviates this issue…

Good luck on the final everyone!
Extra study guide I made here



- Where D is our data in and WE is our "record" button (stands for "write enabled"): whether or not we want to store the data we take in
- So now we can store 1 bit! What if we wanna store more? We can, we just string together these latches, but instead of having 1 WE for each input, we have 1 universal WE.



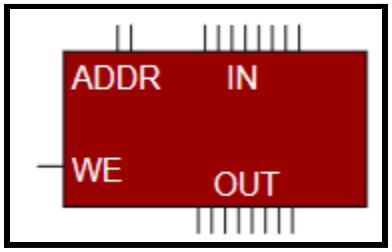
- This is what we call a **register**, which is an array of *n* D latches. (remembers *n* bits)

Level-Triggered Logic

- The RS Latch and the D Latch are both examples of *level-triggered logic*
- **Level-triggered logic** is logic where the output can only change when the enable (WE = write enabled) is 1
- When the enable is 0, output doesn't change

Memory

- **Address Space** : How many addresses are possible?
  - The number of combinations we can make with our address bits
- **Addressability** : How big is each memory location?
  - The number of wires going in and out
- The building blocks of memory…
  - *Registers*
    - Gated D Latch



Good luck on the final everyone!
Extra study guide I made here

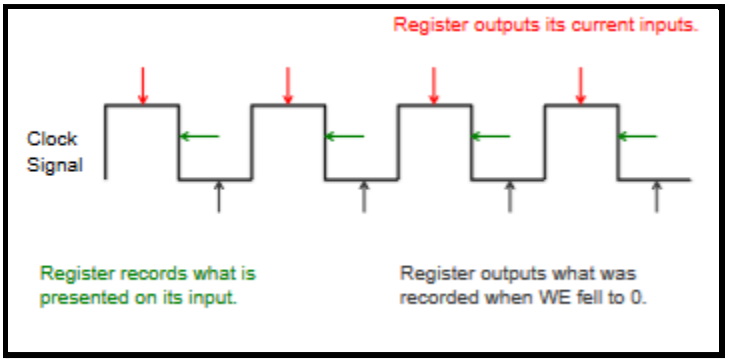
- *Decoder*
- *Multiplexors*

Review

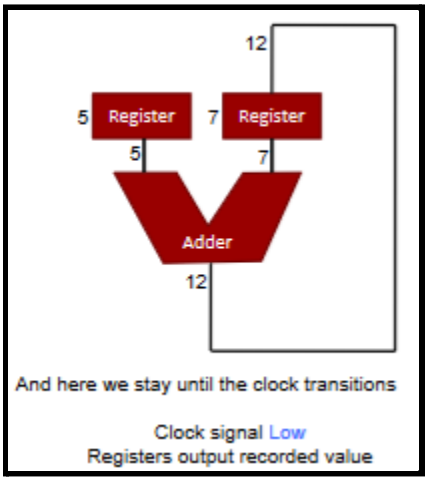
- Combinational Logic Circuits
  - Make decisions; same inputs always produce same output
  - Depends on what is happening **now**
- Sequential Logic Circuits
  - Makes decisions & stores information
  - Output depends on **inputs AND state**
  - Depends on what has **happened in the past** and what is **happening now**

A Problem with our D Latch

- A **clock** makes the job of designing digital circuits easier by synchronizing operations.

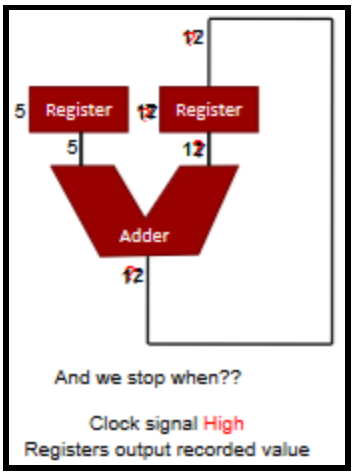


- Using the clock, we can attach it to the WE. Problem solved, right? Not exactly…
- There's an issue when *outputs are used as inputs* to the same circuit element.



- When it's low, the clock functions fine, the 12 isn't stored in the register on the right. However, once the clock changes to high…

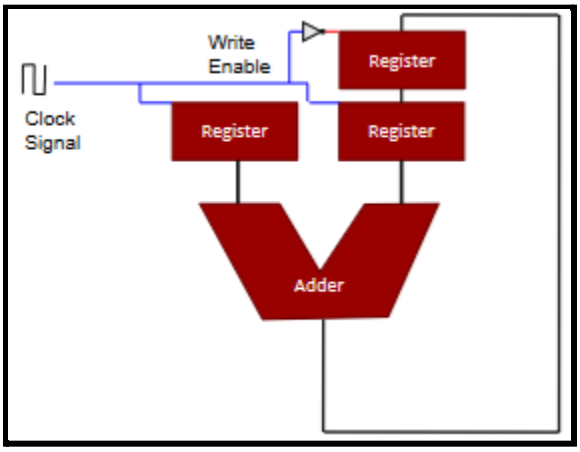
Good luck on the final everyone!
Extra study guide I made here



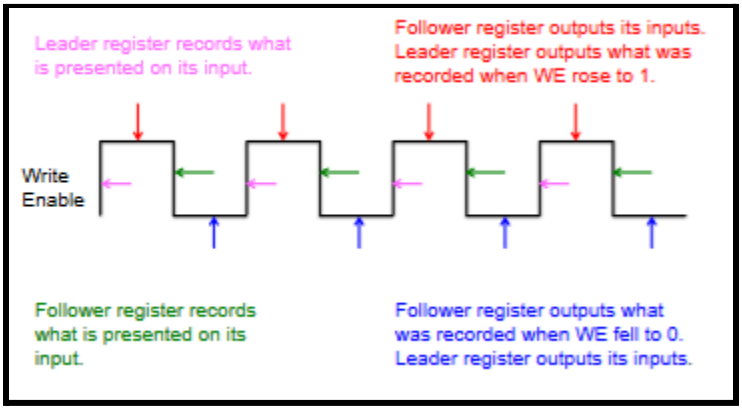
- We have no clue when to stop, it could go on as many times as the registers and adder can work until the clock goes back to low. How can we fix this?

Leader-Follower

- Since the wire is propagating a number, we can't use a transistor to restrict propagation. Instead. Let's use another register…

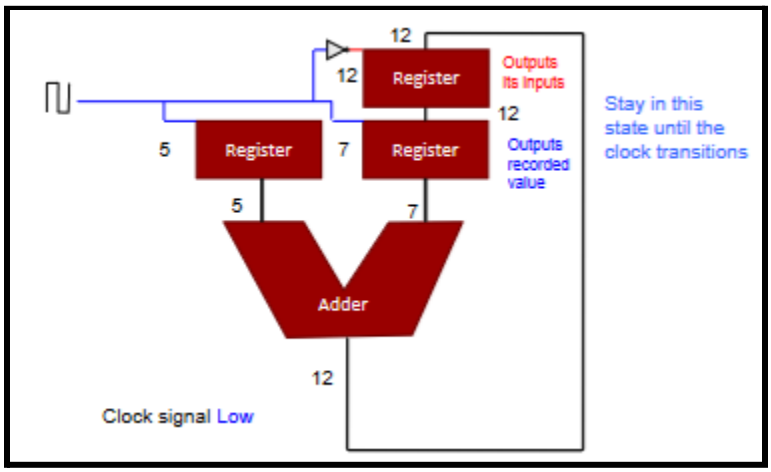


- Using another register, we can take advantage of inversion and the way we store things in registers.

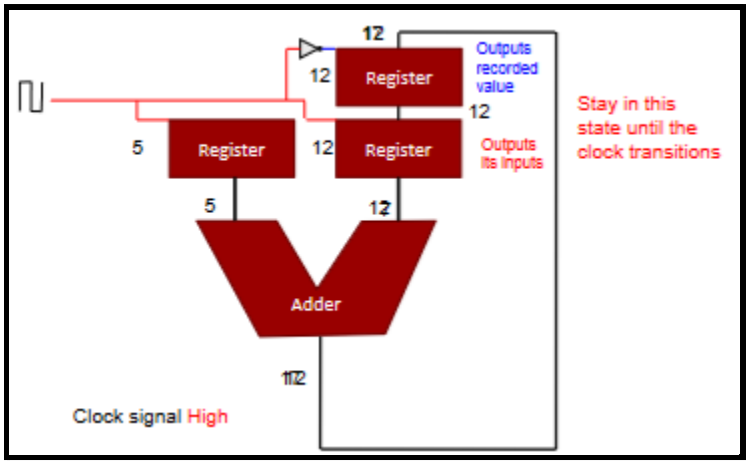


Good luck on the final everyone!
Extra study guide I made here

- Take our previous example. We have a register of 5, an adder, and an original number we want to add to (on the right registers). Now, we will only run this once per clock cycle



- Here, the 12 is stuck between the two registers until the clock signal goes high. Once it goes high…



- The 12 falls through into the 2nd register (what we call the follower register; the top register on the right is the leader), which is then used in the adder to get 17. While the signal is high, the 17 is stuck between the adder and the wire getting to the leader. Once the clock goes back to low, the 17 gets recorded in the leader and we revert back to the image above

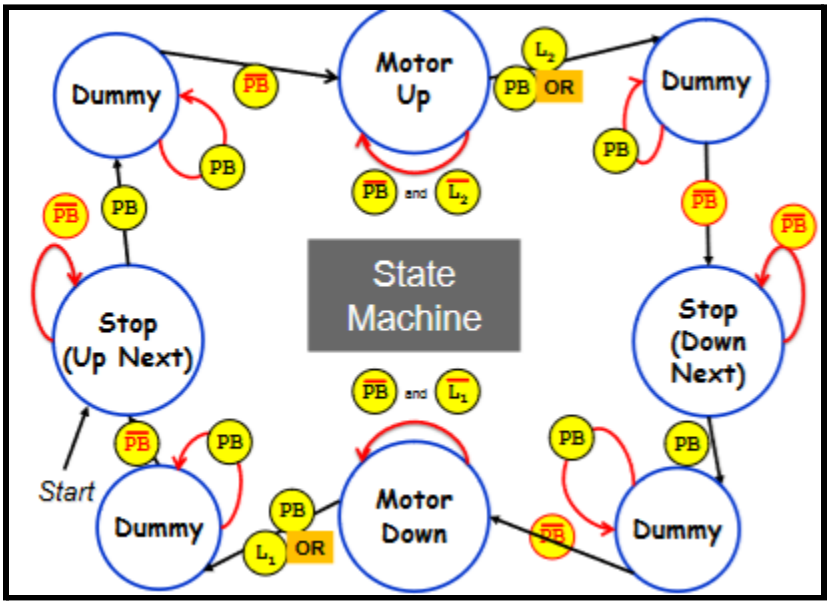
Edge Triggered Logic

- Remember previously that we said the D-Latch was *level-triggered*.
- Well, the above is considered **edge-triggered logic**, where many sequential logic circuits are based on clocks instead of enable bits
- There are two types…
    - *Rising-edge triggered logic*: the output can only change when the clock *rises* from 0 to 1

Good luck on the final everyone!
Extra study guide I made here

- *Falling-edge triggered logic*: the output can only change when the clock *falls* from 1 to 0

Concept of State & State Machine



- With the state machine above, we can convert it into a truth table…

P	L_1	L_2	C_2	C_1	C_0	N_2	N_1	N_0	U	D
0	x	x	0	0	0	0	0	0	0	0
1	x	x	0	0	0	0	0	1	0	0
0	x	x	0	0	1	0	1	0	0	0
1	x	x	0	0	1	0	0	1	0	0
0	x	0	0	1	0	0	1	0	1	0
x	x	1	0	1	0	0	1	1	1	0
1	x	x	0	1	0	0	1	1	1	0
0	x	x	0	1	1	1	0	0	0	0
1	x	x	0	1	1	0	1	1	0	0
0	x	x	1	0	0	1	0	0	0	0
1	x	x	1	0	0	1	0	1	0	0
0	x	x	1	0	1	1	1	0	0	0
1	x	x	1	0	1	1	0	1	0	0

Circuit Inputs | Current state | Next state | Circuit Outputs

P	L_1	L_2	C_2	C_1	C_0	N_2	N_1	N_0	U	D
0	0	x	1	1	0	1	1	0	0	1
1	x	x	1	1	0	1	1	1	0	1
x	1	x	1	1	0	1	1	1	0	1
0	x	x	1	1	1	0	0	0	0	0
1	x	x	1	1	1	1	1	1	0	0

Circuit Inputs | Current state | Next state | Circuit Outputs

- With a truth table from the state machine, we can convert it into a circuit. Take for instance the sum of products equation for N_2…

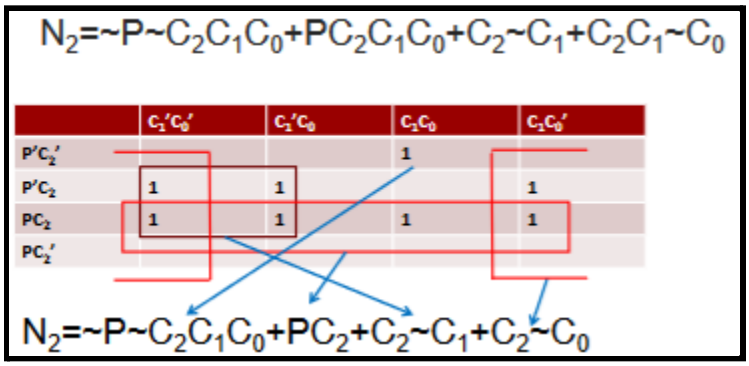
$$
\begin{aligned}
N_2 = \quad & \sim P \sim C_2 C_1 C_0 \\
+ \quad & \sim P C_2 \sim C_1 \sim C_0 \\
+ \quad & P C_2 \sim C_1 \sim C_0 \\
+ \quad & \sim P C_2 \sim C_1 C_0 \\
+ \quad & P C_2 \sim C_1 C_0 \\
+ \quad & \sim P \sim L_1 C_2 C_1 \sim C_0 \\
+ \quad & P C_2 C_1 \sim C_0 \\
+ \quad & L_1 C_2 C_1 \sim C_0 \\
+ \quad & P C_2 C_1 C_0
\end{aligned}
$$

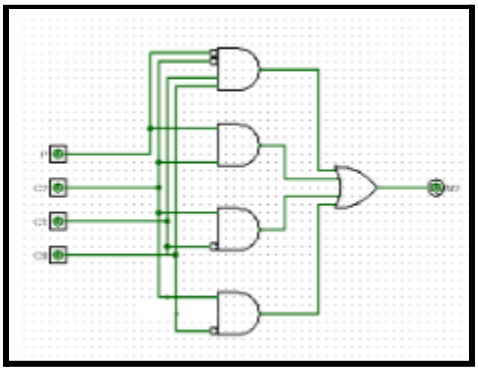
Good luck on the final everyone!
Extra study guide I made here

- In its current form, it wouldn't be wise to go right into building the circuit. Furthermore, since we are working with 5 variables, making a K map is inadvisable (we will only work with KMaps with a max of 4 variables)Using boolean logic, we should try and simplify it…

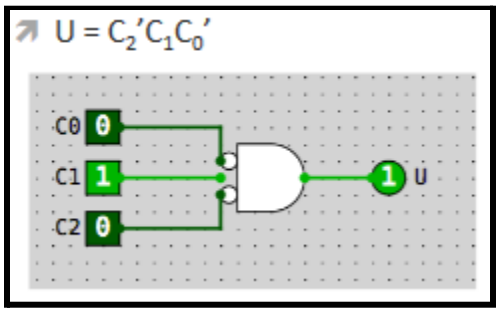
$$N_2 = \sim P \sim C_2 C_1 C_0 + P C_2 C_1 C_0 + C_2 \sim C_1 + C_2 C_1 \sim C_0$$

- Now, we can make a K-Map, since we have less variables…





- Dope, now do that 4 more times for each output. Here is just one more of the output.



- There are two types of state machines…
  - *One Hot*
    - One bit per state
    - Only one bit is on at a time
    - Faster
    - Requires more flip flops
    - State progresses like: 0001, 0010, 0100, 1000

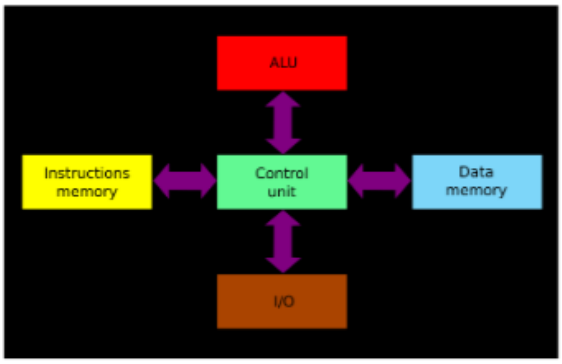
Good luck on the final everyone!
Extra study guide I made here

- *Binary Encoded*
  - Encode state as a binary number
  - Use a decoder to generate a line for each state
  - Slower
  - More complicated
  - States progresses like: 000, 001, 010, 011, 100

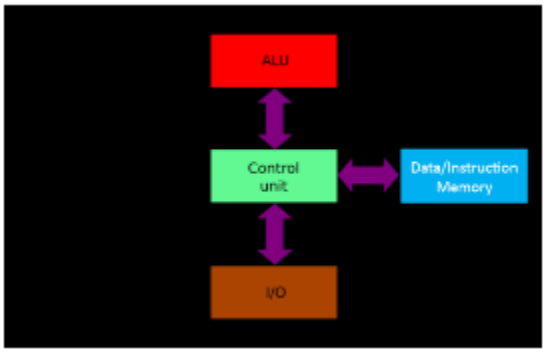
Good luck on the final everyone!
Extra study guide I made here

L06 - Von Neumann Model

- Harvard Model - Data memory and instructions memory are separate



- Von Neumann Model - Data and instructions memory is one block



- Von Neumann's model leads us to treat machine instructions as just another data representation; just like integers, 2's complement integers, ASCII, etc.
- Note that Von Neumann's model presents a risk; if we have one single memory, and someone is malicious enough, they can use instructions to manipulate the data memory. However, this risk was deemed worth it for the benefits we get from it (and we have learned of many ways to mitigate the risk, too)

Introduction to the LC-3 CPU

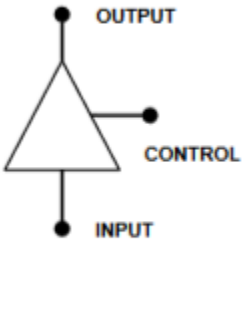
- Address space: $2^{16} = 65536$
- Addressability: 16 bits
- Architecture type: Von Neumann
- General purpose registers: 8
- Instruction size: 16 bits

One Last Digital Logic Component

- There is one last combinational logic circuit we have not yet discussed: **Tri-State Buffer**
- The tri-state buffers are used to help with the **bus**
    - The bus is 16 shared wires in the datapath
    - No transistors or logic, just wires to connect things

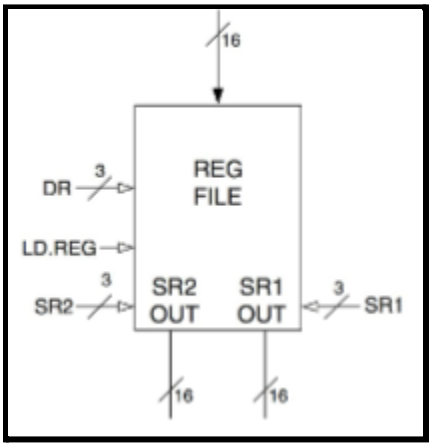
Good luck on the final everyone!
Extra study guide I made here

    - This is what is used to disconnect inputs so the bus can be shared among multiple components
- A triangle by itself is a *buffer*, a triangle with a bubble is an *inverter*
- The **tri-state buffer** is used to…
    - Prevent data path fires
    - Avoid short circuits
    - Disconnect a circuit from a bus so that another circuit can assert a value on the same bus without interference
    - Present a value that is neither 1 or 0 on an output



CONTROL	INPUT	OUTPUT
0	0	Z
0	1	Z
1	0	0
1	1	1

Register File Circuit



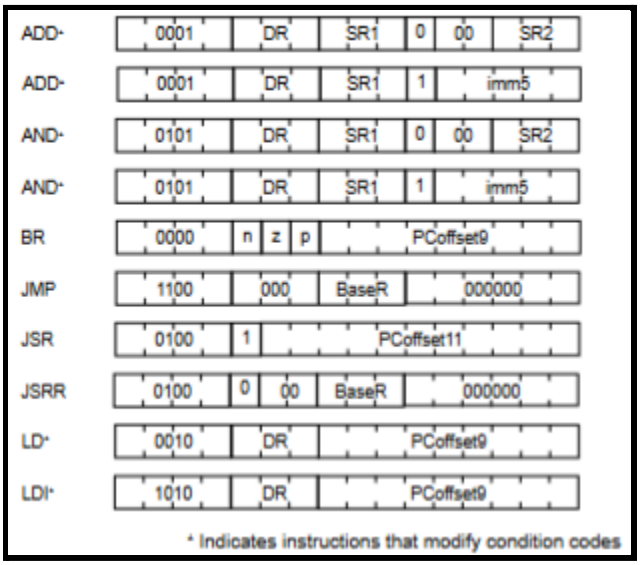
- A small, fast "memory"
    - Address Space: 8 registers
    - 16-bit addressability per reg
- Two outputs
    - Dual-ported memory
    - Can read two regs at same time:
        - SR1 (3 bit address)
        - SR2 (3 bit address)
- One input
    - DR (3 bit address) ["destination register"]
    - LD.REG (write enable)
- Can read two registers and write one register in a signal clock cycle

Machine Instruction

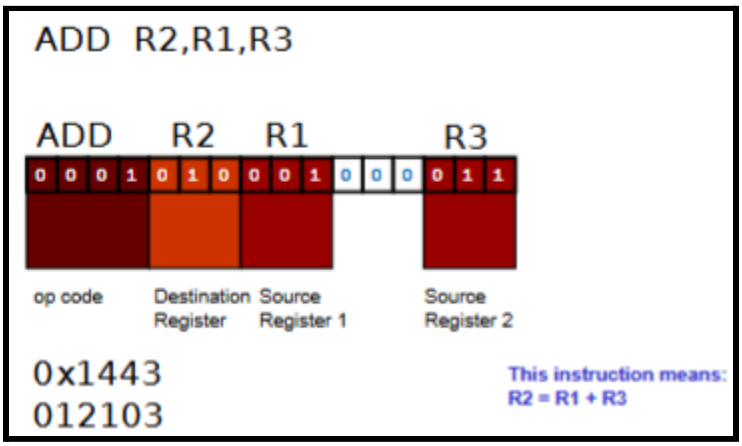
- There are three categories of machine instructions for the LC-3…
    - *Operate (ALU)*
        - ADD, AND, NOT (yes, you can actually do all arithmetic with these 3)
    - *Date Movement (Memory)*

Good luck on the final everyone!
Extra study guide I made here

        - Load: LD, LDR, LDI, LEA
        - Store: ST, STR, STI
    - *Control*
        - BR, JMP, JSR, JSRR, RET, TRI, TRAP



- Here's an example of an instruction…

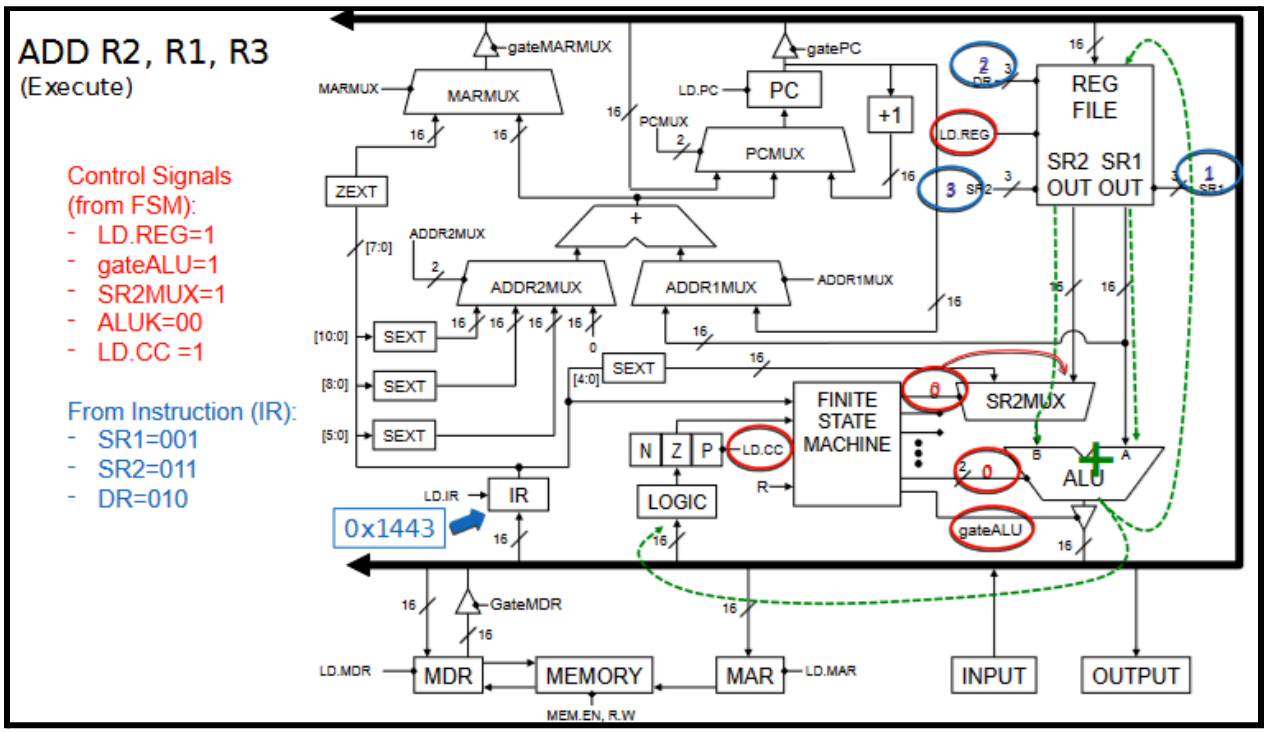


    - The left for bits indicate the operator, called the *op code*. In this case, 0001 is ADD.
    - The next 3 bits indicate the register to store the output of the operation in.
    - The next 3 bits indicate the register to pull the number from, the *source register*
    - Lastly, our the bits all the way on the right indicate
- How do we make the LC-3 do this "ADD" operation?
    - The LC-3 is a glorified state machine (just like the garage state machine), so it has output signals (**control signals**, e.g. UP and DOWN signals in the garage door)
    - The FSM turns on specific *control signals* in the LC-3, which activates certain paths in the datapath

Good luck on the final everyone!
Extra study guide I made here

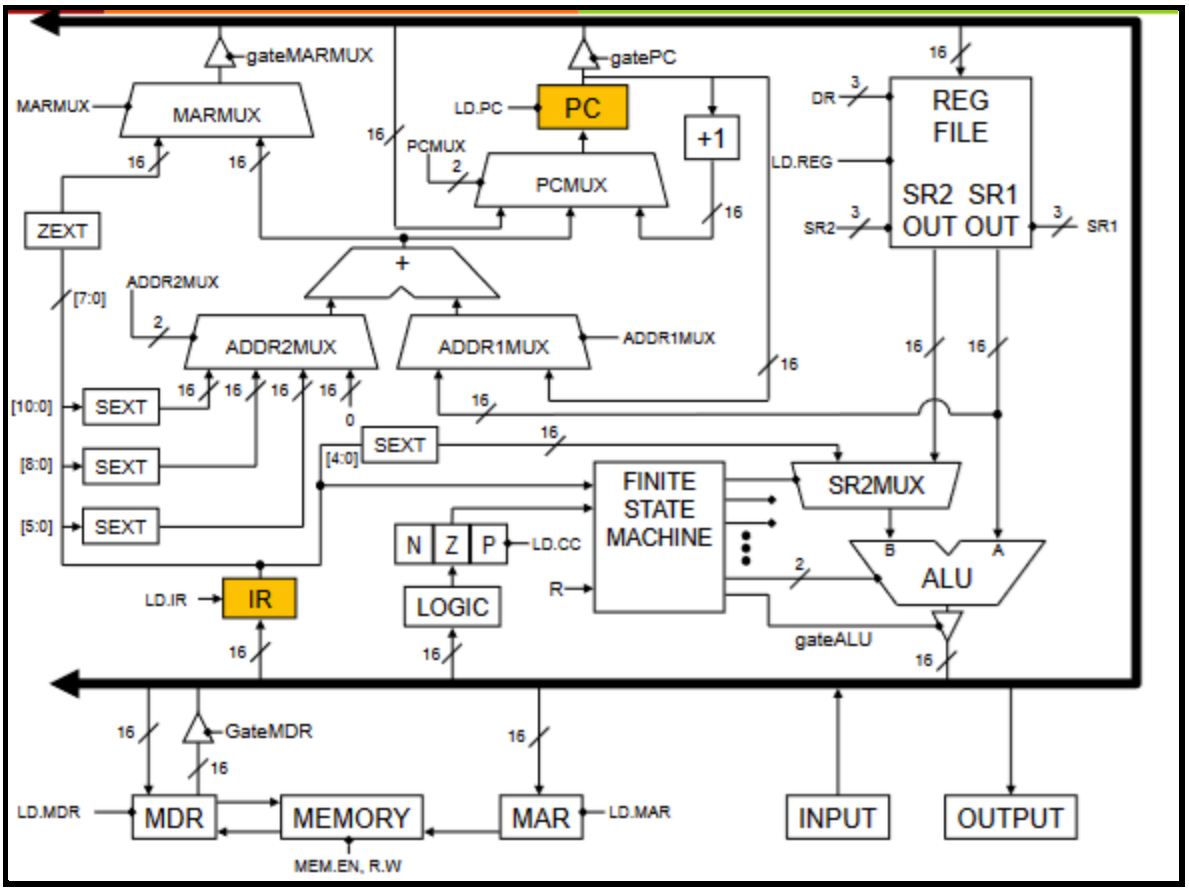
- These control signals can make the datap[ath do things, such as add two numbers (R1 and R3) and store the result in R2

- Example: ADD R2, R1, R3 (which is R2 = R1 + R3)



- How did the FSM get that ADD Instruction?
  - Before the FSM could execute the ADD instruction, it has to fetch the instruction from memory
  - The FSM has a procedure that it executes over-and-over to process instructions
    - FETCH
    - DECODE
    - EXECUTE
  - The FSM fetches and decodes each instruction before it executes it

Good luck on the final everyone!
Extra study guide I made here



PC stands for **Program Counter**, and is a register that holds the address of the next instruction to be executed.

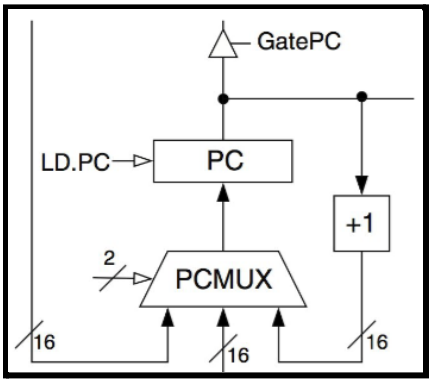
IR stands for the **Instruction Register**, and is a register that holds the instruction currently being executed

Machine Code Program

- A **machine code program** is a series of machine code instructions, where each instruction is a 16-bit data value
- The instructions are stored in memory (usually in sequence; consecutive memory locations)
- Each memory locations has an address, which is a 16-bit unsigned integer

Program Counter Circuit

- The program counter circuit contains multiple elements…
    - PCMUX: Adder to increment PC (+1)
        - (Not shown) Data from bus to PC (bus is above GatePC tri-state buffer)
        - Data from effective address* calculator to PC



Good luck on the final everyone!
Extra study guide I made here

- LD.PC: Write enable for PC reg
- GatePC: Put PC value on bus

FETCH & DECODE

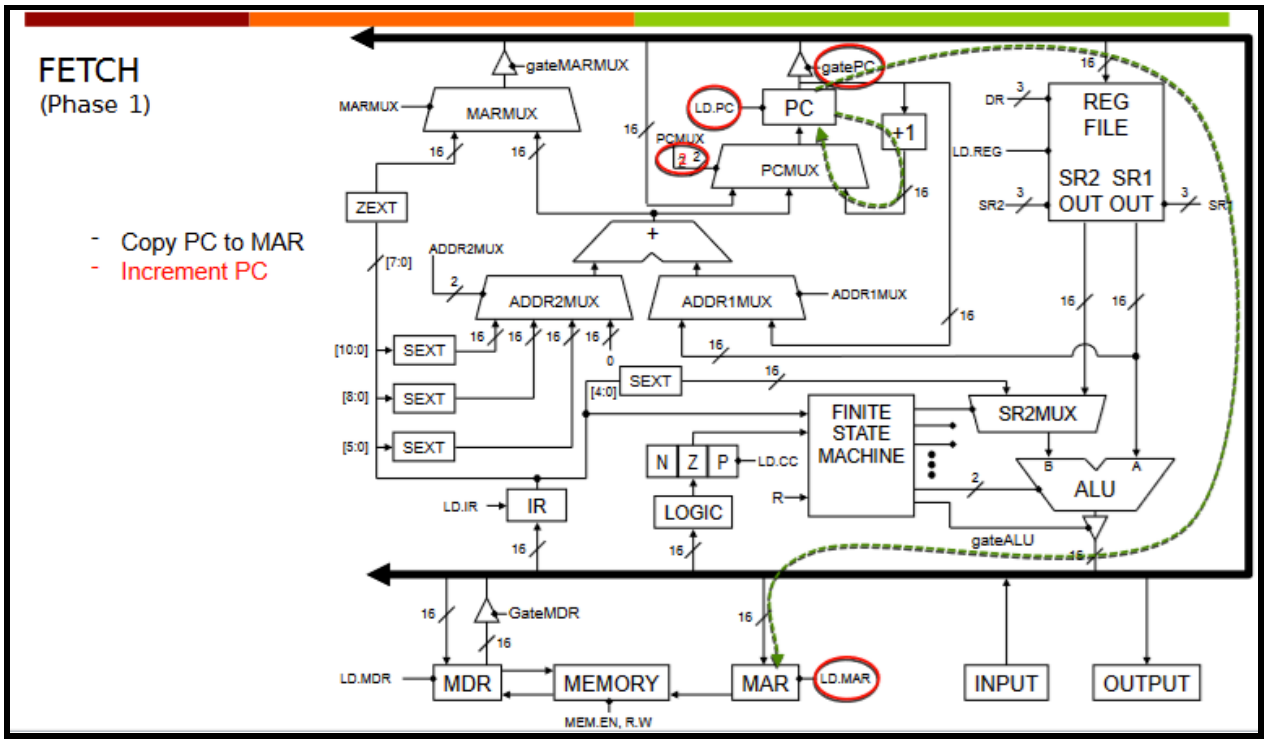
- The ADD machine code instruction lives in memory
- Before we do the add, we have to get the machine code from memory
  - The PC register tells us the memory address where the instruction is located
  - The IR holds the value read from memory (the machine code 16-bit instruction)
- FETCH: Takes 3 clock cycles to read data (the instruction) from memory
- DECODE: Takes 1 clock cycle
  - This is where the FSM generates the control signals for the specific instruction (such as ADD)
- So overall, it takes 4 clock cycles/states to fetch and decode and instruction

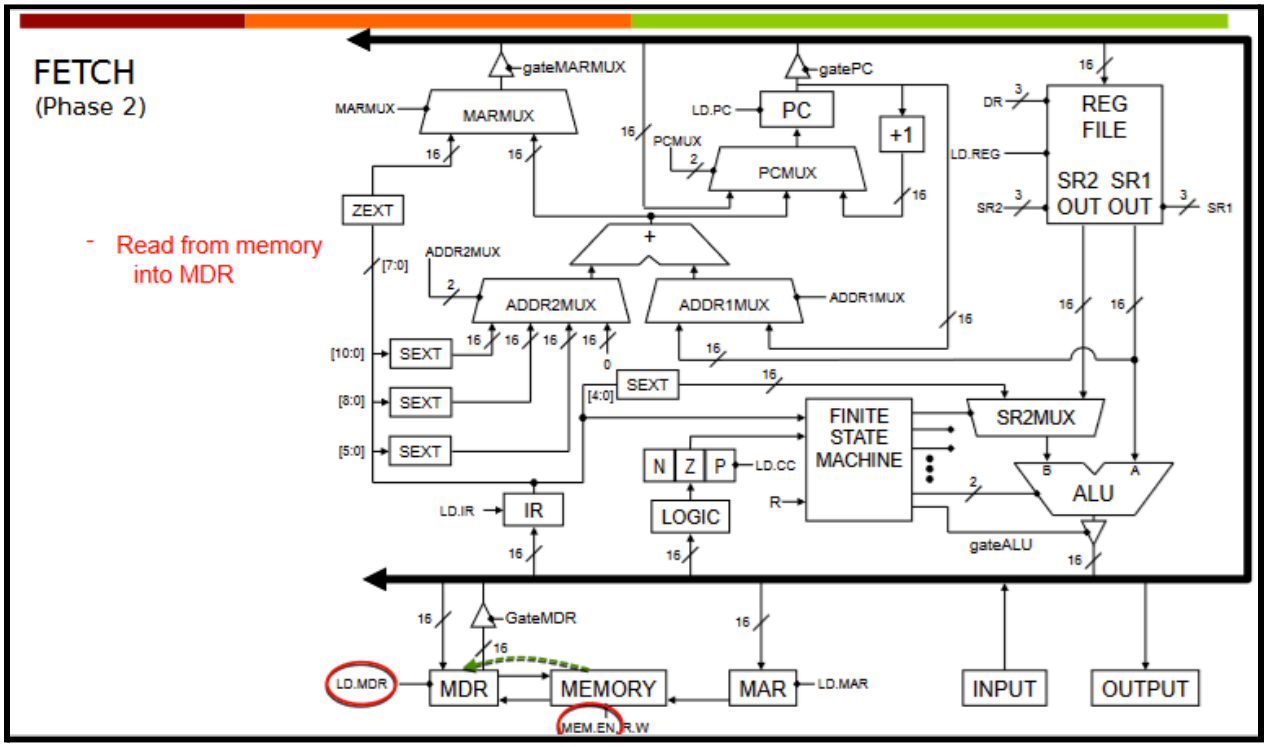
Machine Code Instruction Cycle

- FETCH…
  - Load next instruction (at address stored in PC) from memory into Instruction Register (IR)
    - Copy contents of PC into MAR
    - Send "read" signal to memory
    - Copy contents of MDR into IR
  - Increment PC, so that it points to the next instruction in sequence
    - PC becomes PC + 1

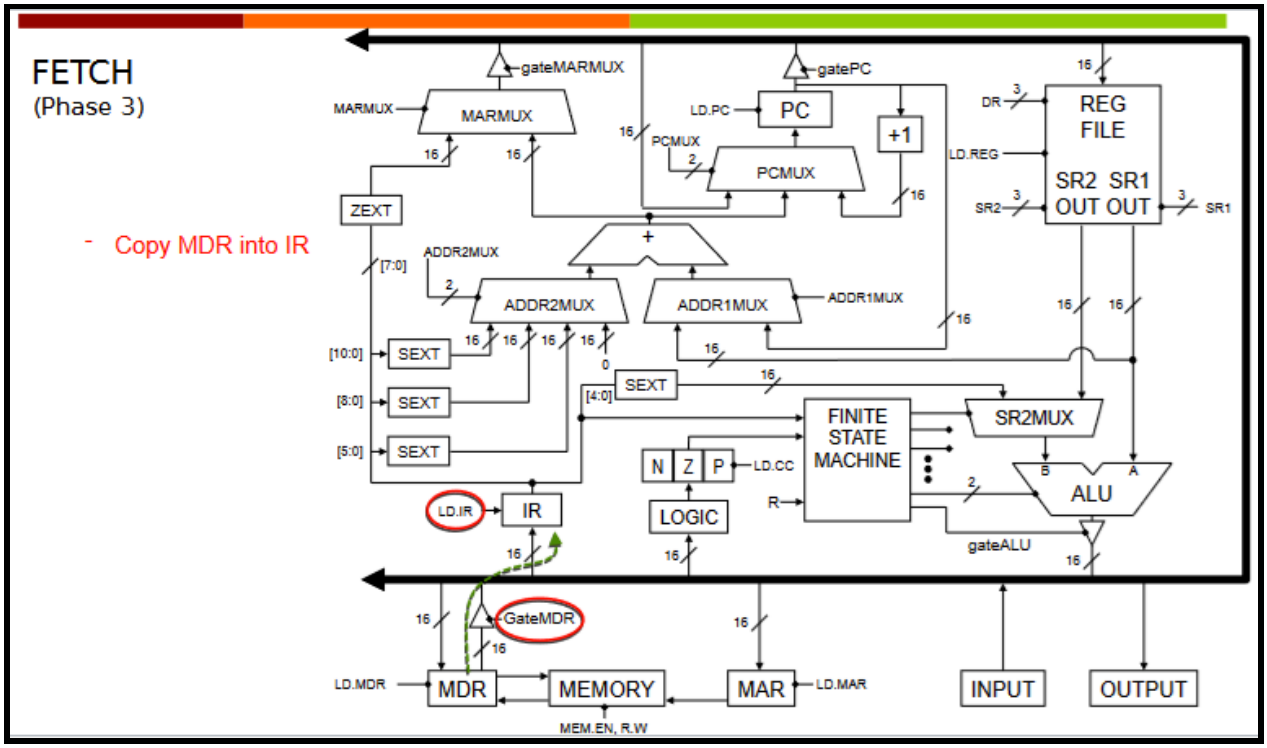
- Fetch
- Decode
- Evaluate Address *[optional]*
- Fetch Operands *[optional]*
- Execute *[optional]*
- Store Result *[optional]*

Good luck on the final everyone!
Extra study guide I made here





Good luck on the final everyone!
Extra study guide I made here



- DECODE
    - We'll get back here…
- EXECUTE
    - Perform the operation, using the source operands
    - Ex. Send operands to ALU and assert ADD control signal
    - Ex. Calculate nothing (e.g., for loads and stores to memory)
    - Every instruction has a different execute path with different control signals
    - Every instruction starts with…
        - FETCH
        - FETCH
        - FETCH
        - DECODE

Good luck on the final everyone!
Extra study guide I made here

Instruction Processing Review

- Instruction bits look just like data bits in memory – it's all a matter of our interpretation
- Three basic kinds of instructions:
  - computational instructions (ADD, AND, NOT)
  - data movement instructions (LD, ST, …)
  - control instructions (JMP, BRnz, …)
- Six basic phases of instruction processing:
  F → D → EA → OP → EX → S
  - not all phases are needed by every instruction
  - phases may take variable number of machine cycles (states)

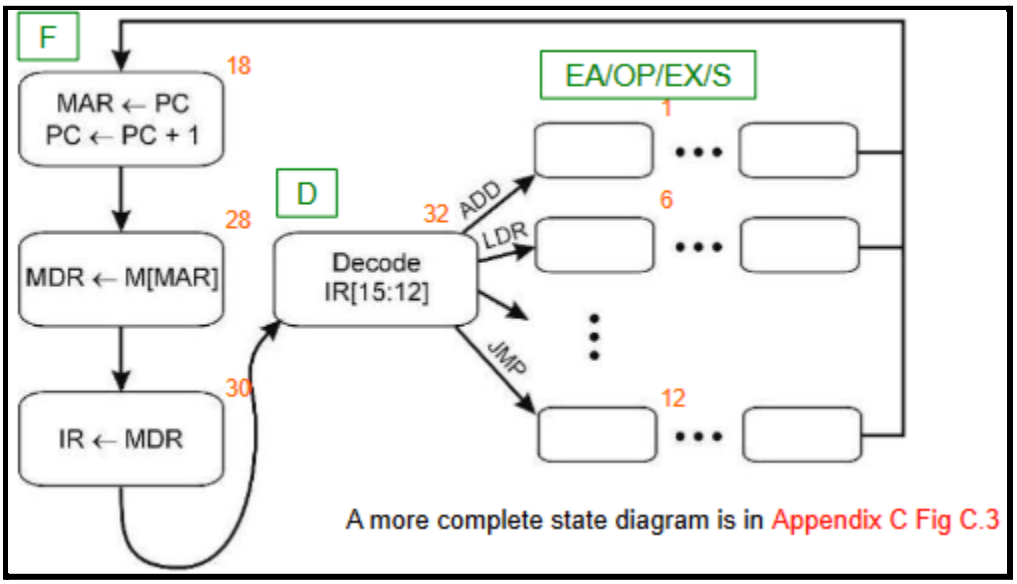
The LC-3 State Machine Review

- LC-3 Finite State Machine
  - Has 64 possible states
  - Orchestrates 42 control signals
    - Multiplexor selectors
      - PCMUX, MARMUX, ADDR2MUX, …
    - Tri-state buffer enables
      - gatePC, gateMARMUX, gateALU,…
    - Register write-enables
      - LD.PC, LD.REG, LD.MAR, LD.CC, …
    - Other control signals
      - ALUK, MEM.EN, R.W, …
- The wires aren't all shown explicitly on the datapath diagram – *to avoid clutter*, but each signal has a name

Good luck on the final everyone!
Extra study guide I made here

What Makes the Fetch-Execute Cycle Happen?

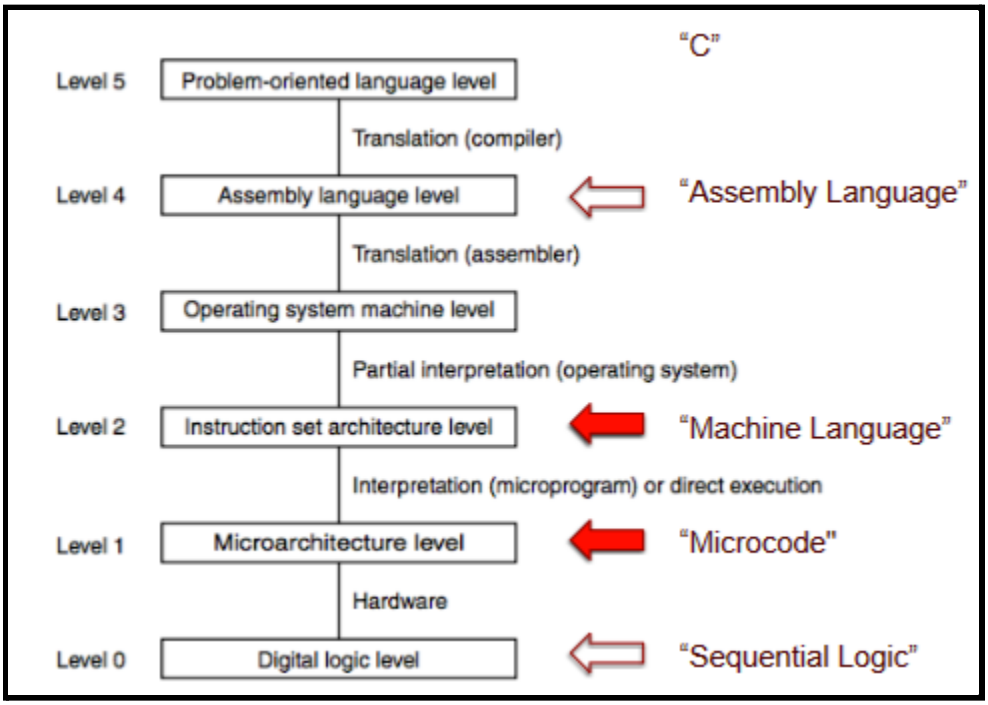
- The FSM, which sequences through the states



Good luck on the final everyone!
Extra study guide I made here

L07 - The LC-3 Part I

Road Map

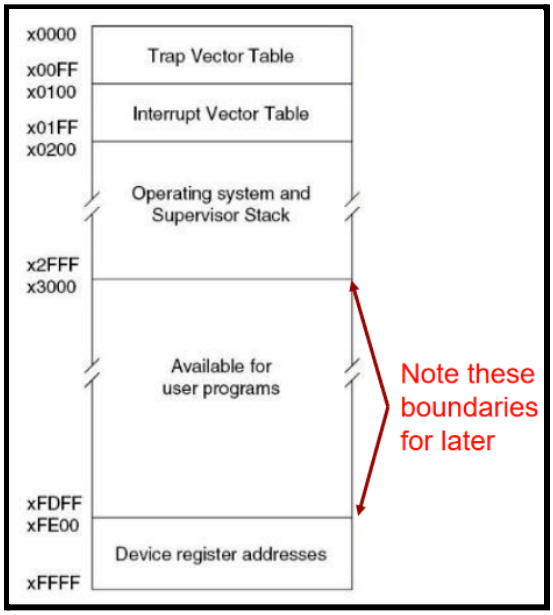


Instruction Set Architecture

- **Instruction set architecture** (ISA) is the view from *outside* the black box, it's the programmers view
- Who uses an ISA?
  - Machine language programmers
  - Assembly language programmers
  - Compiler writers
  - High-level language programmers interested in performance, debugging, etc.
- What is specified?
  - Memory organization
  - Registers
  - Instruction Set
    - Opcodes
    - Data types
    - Addressing modes

LC-3 Memory Organization

- Address space is 16 bits addresses and is written in hexadecimal by convention
  - [0, 65535] or [x0, xFFFF]
- Addressability: 16 bits



Good luck on the final everyone!
Extra study guide I made here

- Word Addressable/Byte Addressable?
    - Strictly word addressable w/ 16 bit words
- Every memory address is the next 16 bit word in memory

LC-3 Registers

- Contains 8 general purpose registers ranging from R0 - R7
- Is the PC considered to be a GPR? => no
- By convention, certain registers have designated purposes

How Many Instructions? (ISA Design Philosophies)

CISC (Complex Instruction Set Computer) aka. Lots of instructions	**RISC** (Reduced Instruction Set Computer) aka. Few instructions
Examples include: X86, DEC, VAX, IBM Z-series	Examples include: ARM, PowerPC, MIPS, LC-3
Do as much as you can in one instruction	Expose as much as you can to the compiler so that it can optimize
Relatively easy to write efficient machine code by hand	Hard to write efficient machine code by hand

LC-3 Machine Code & Assembly Language

- For now, we are focused on the **LC-3 datapath**
    - The circuit implementation, and the state machine
    - The control signals that make each instruction happen
- You will need to understand what each LC-3 machine code instruction does – *atomically, in insolation*
    - What registers or memory locations does it change?
    - Which ALU operation is happening?
    - How does the data flow through each MUX and the bus to make the operation execute with the desired behavior?
- You do NOT need to understand why or how you would use these instructions in a program yet.
    - We'll cover that later, with assembly programming
    - For now, just understand each instruction's behavior – so you can know how to implement it in the datapath and microcode, using control signals

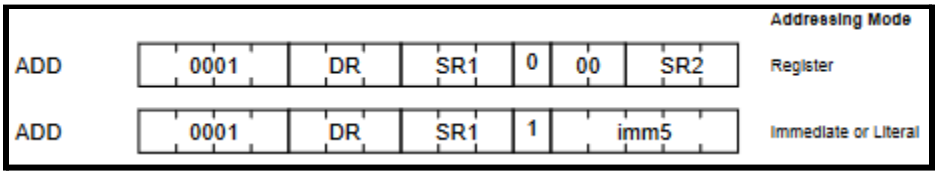
Good luck on the final everyone!
Extra study guide I made here

LC-3 Instructions

Operate (ALU)	Data Movement (Memory)		Control
• ADD • AND • NOT	Load • LD • LDR • LDI • LEA	Store • ST • STR • STI	• BR • JMP • JSR • JSRR • RET • RTI • TRAP

ALU Instructions

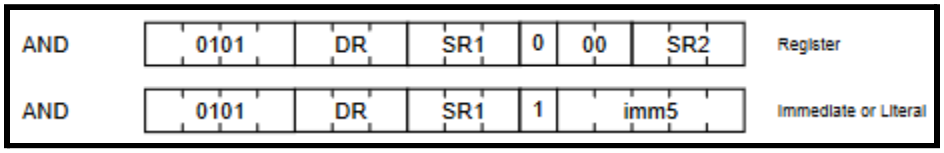
- ADD
  - The ADD instruction has the opcode (0001), the destination register (where to store the result), the source register (the first operand), and then we have two choices based on the **addressing mode**…
    - Register: a 0 after SR1 indicates we will take our second operand from the SR2
    - Immediate: A 1 after SR1 indicates that we will explicitly declare the second operand as part of the instruction



  - Examples…

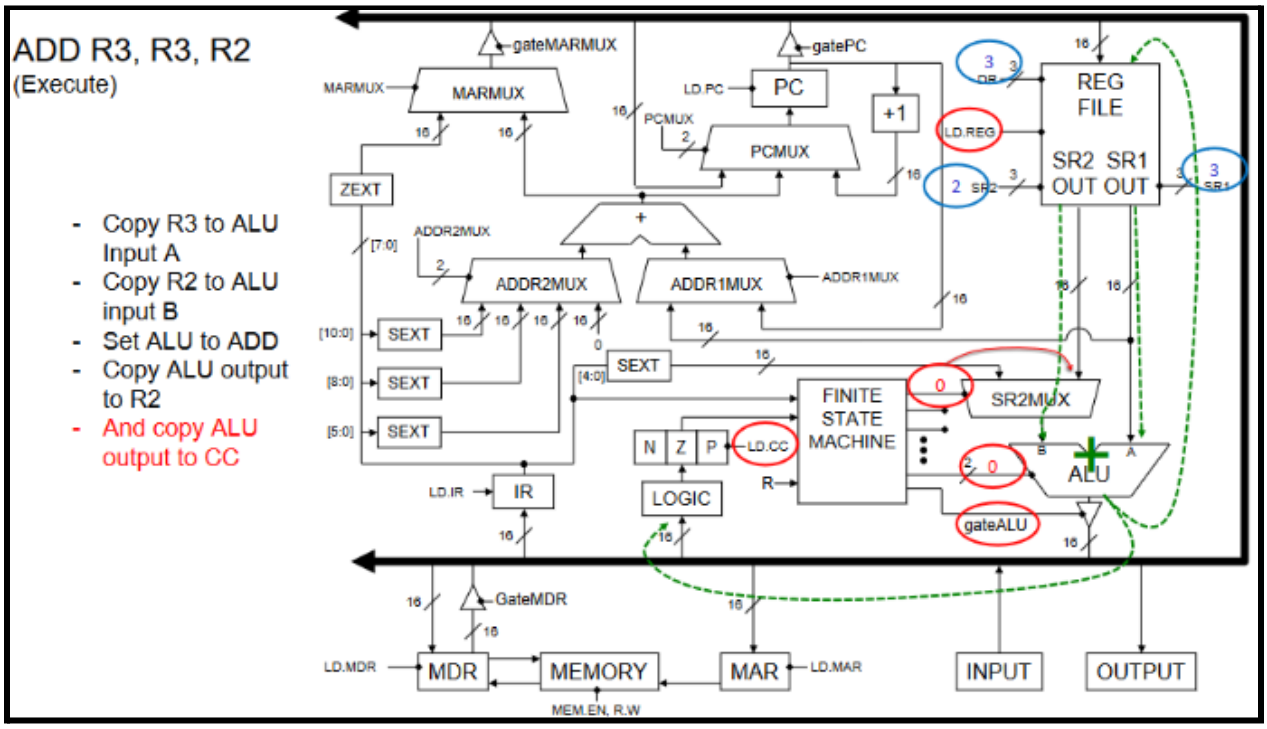
```
ADD  R3,R3,R2
          0001 011 011 000 010     Register
                                   Addressing Mode

ADD  R3,R3,#2
          0001 011 011 1 00010     Literal or Immediate
                                   Addressing Mode
```

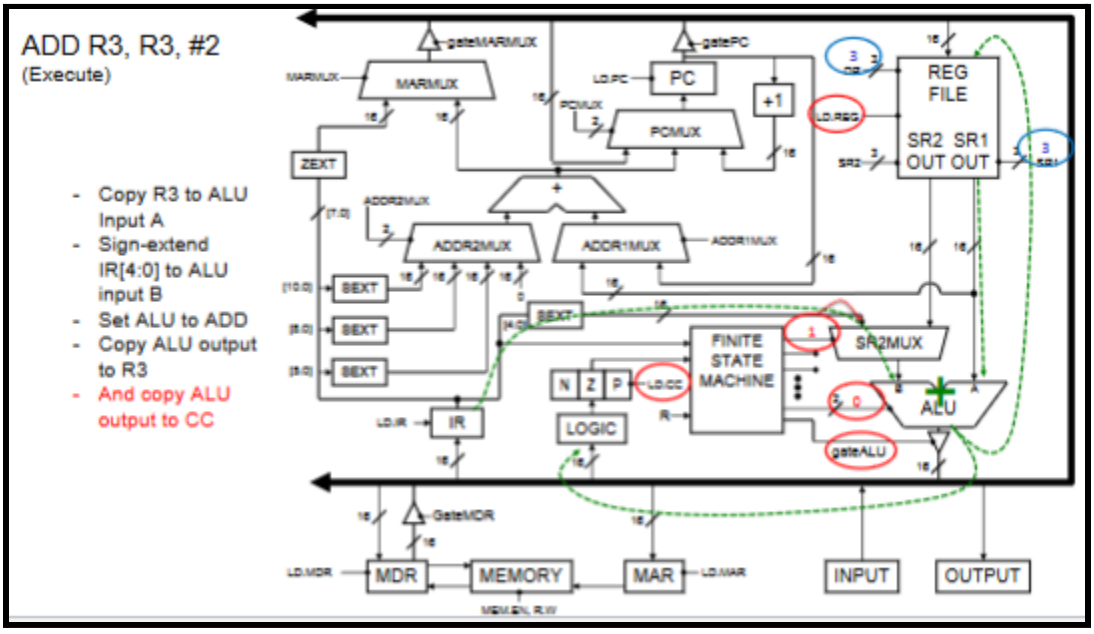
- AND
  - Very similar to ADD. Has it's opcode (0101) along with DR, SR1, and then it branches depending on the addressing mode



Good luck on the final everyone!
Extra study guide I made here

- Here's how these show up in the LC-3…





Data Movement Instructions (Memory)

- The **effective address** describes the memory address we are going to load from or store to
- There's a problem though with trying to pass in a memory-related instruction
  - A machine instruction requires 16 bits
  - A memory address requires 16 bits

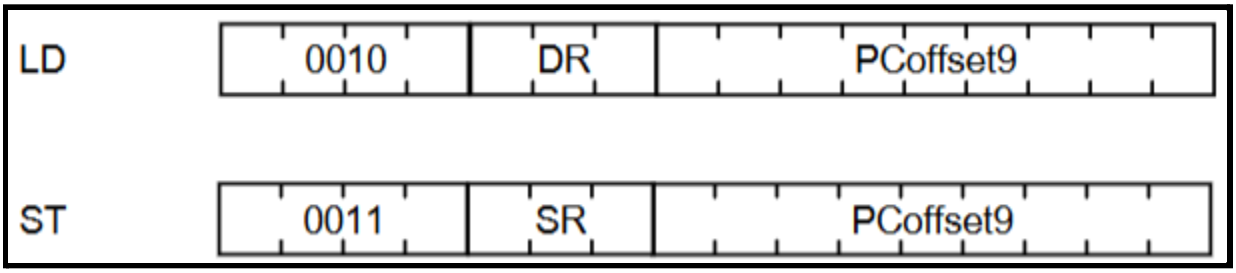
Good luck on the final everyone!
Extra study guide I made here

- We have no more bits to use as an *opcode* to indicate what instruction we want

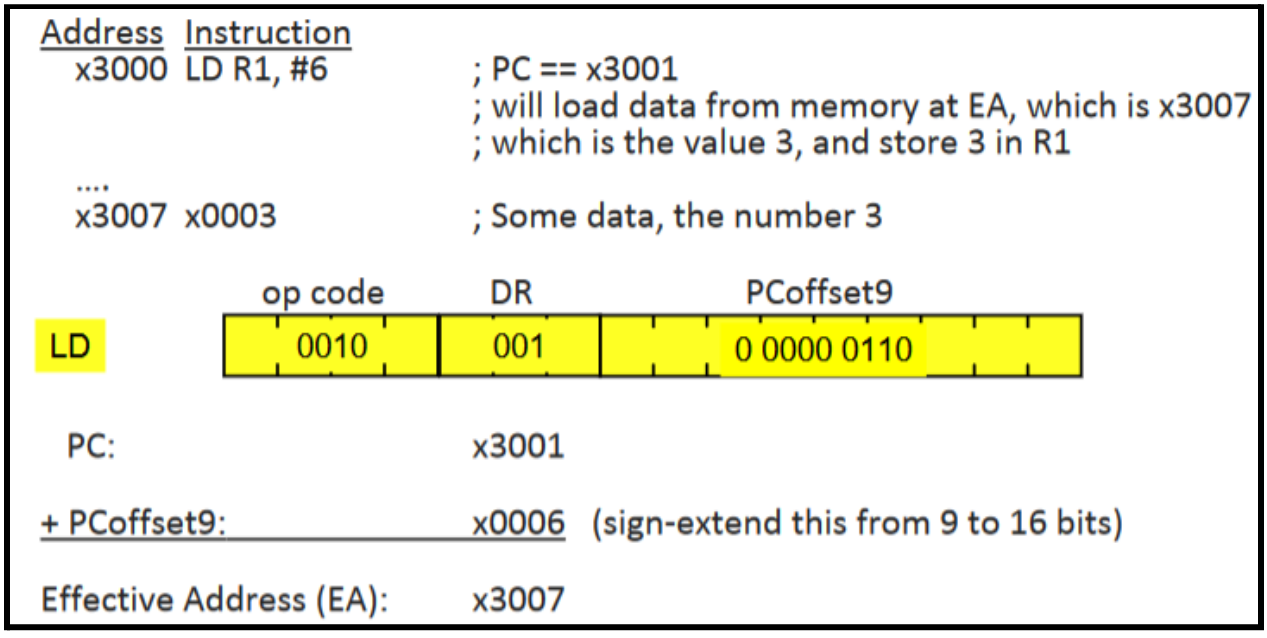
- To handle this, let's go back to the idea of addressing modes brought up when we looked at ADD and AND
- If we add more addressing modes, we can achieve what we want. For the examples here, keep in mind that memory is much like an array, where the size of the array is the address space.

int mem[65536];

  - Immediate or Literal
  - Register
  - *PC-relative*: Take the current address in PC and add an offset to it
    - Ex. mem[PC + offset]
  - *Base* + *Offset*: Stick the memory address in a register, then access the register in the instruction
    - Ex. mem[BaseReg + offset]
  - *Indirect*: :weirdchamp:
    - Ex. mem[mem[PC + offset]]
- Here's an example of PC-Relative addressing…



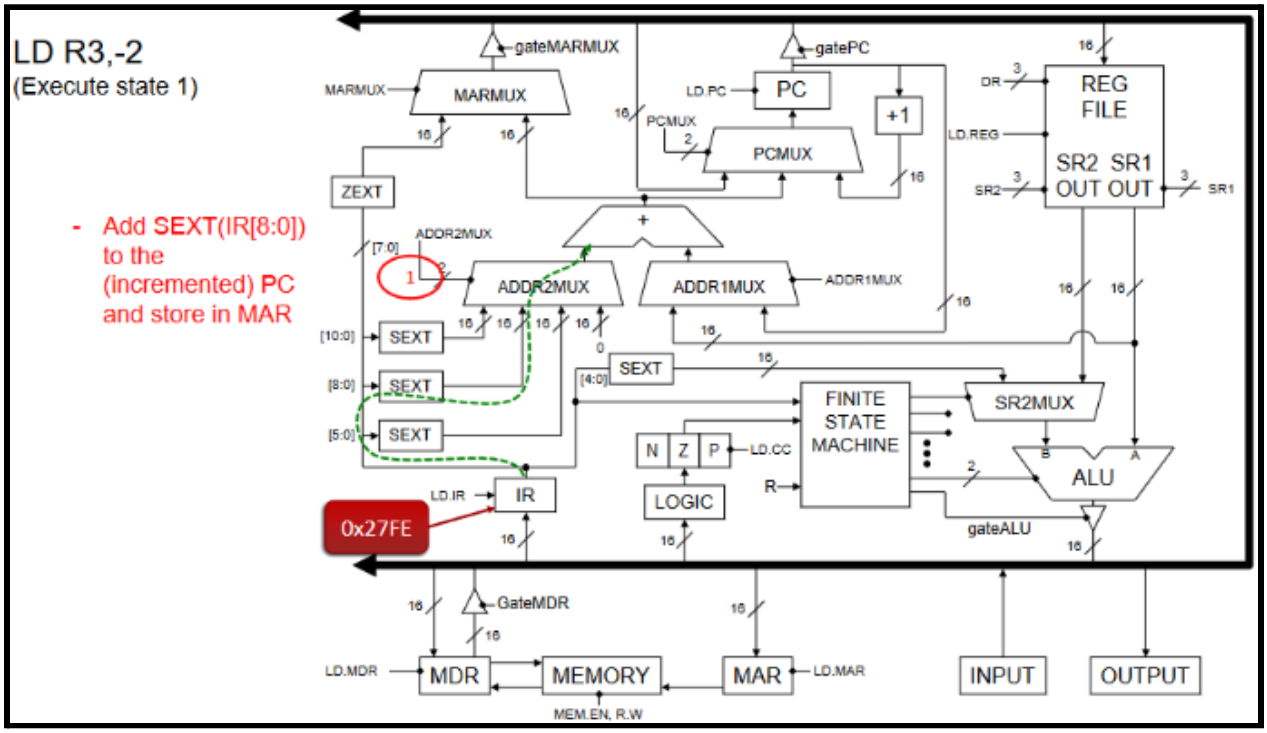
- Another example of PC-relative addressing mode…



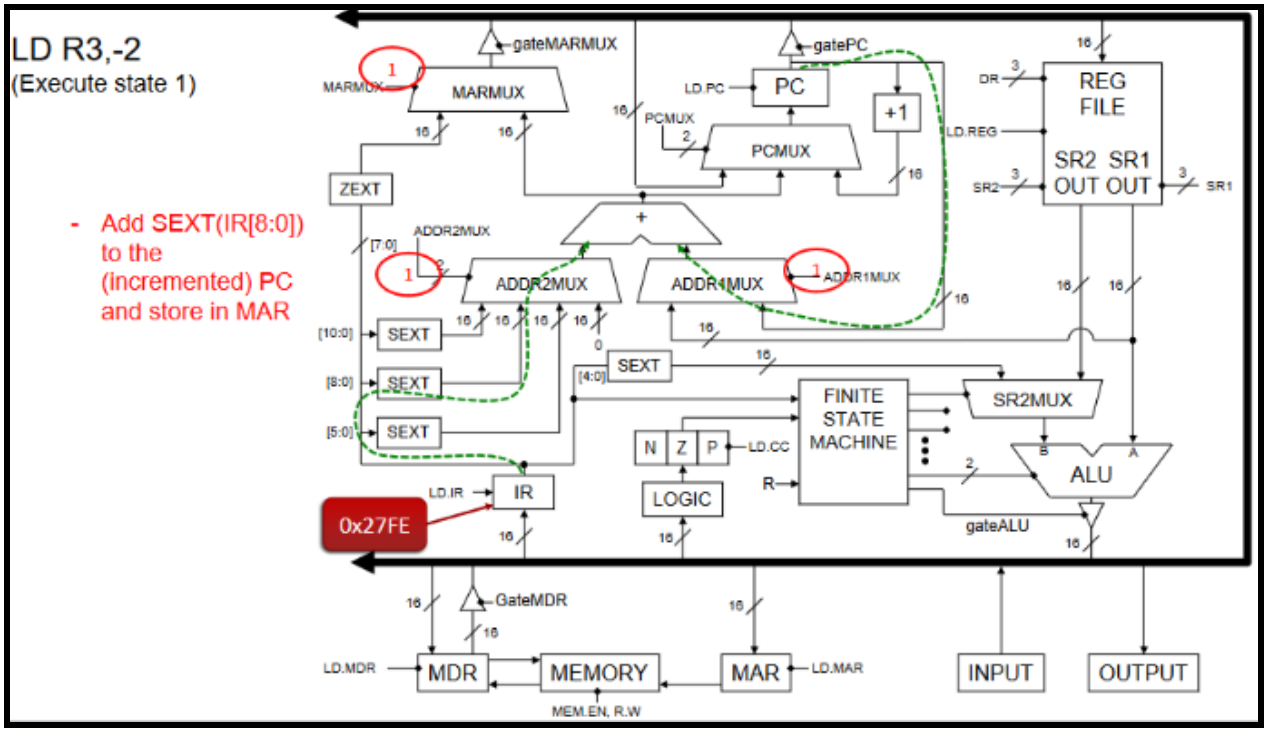
Good luck on the final everyone!
Extra study guide I made here

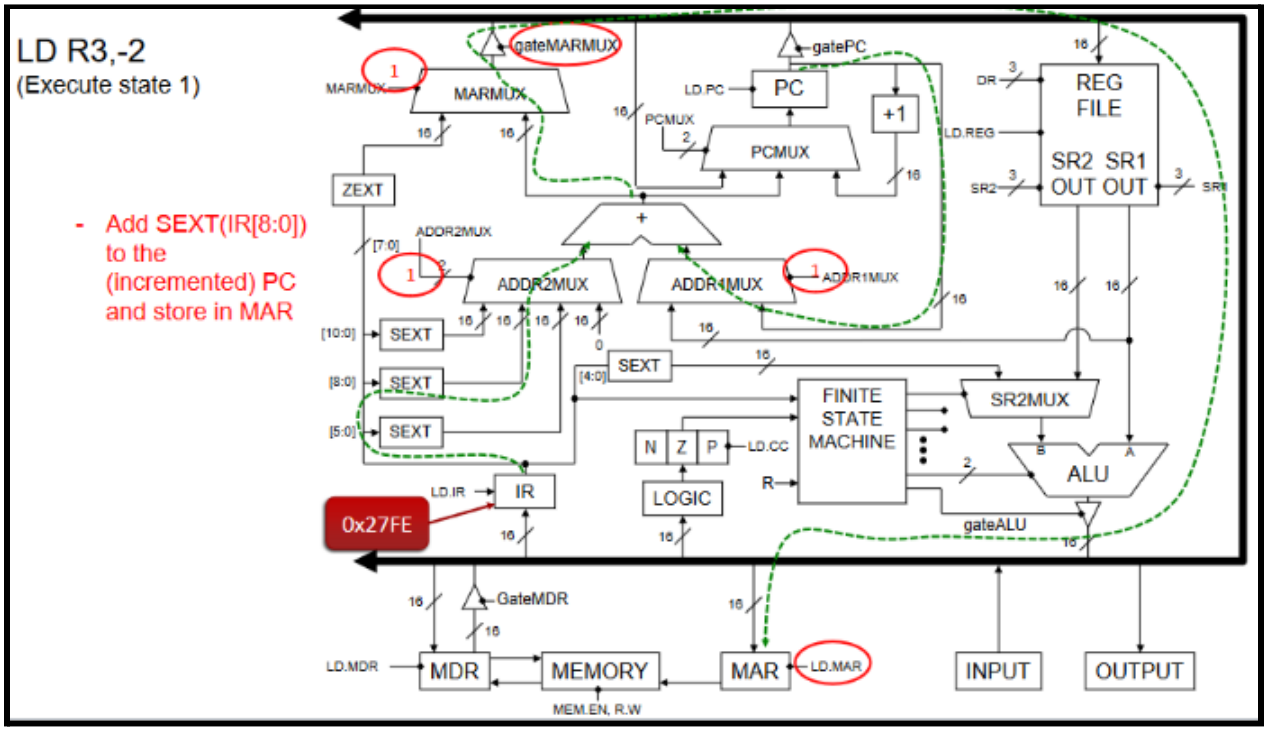
- So effectively, this is telling you to load stuff into register R1. What stuff? Well, whatever is in the *effective address*. Remember: we can't store an entire memory address in a machine instruction, but if we store the address somewhere else, then point to that address, we can circumvent this problem

- Loading from memory takes **three cycles**…
  - Send the address to MAR
  - Read memory[MAR] into MDR
  - Send MDR back to a register
    - Looks a lot like FETCH…
- Here's the path a LD instruction takes in the LC-3…

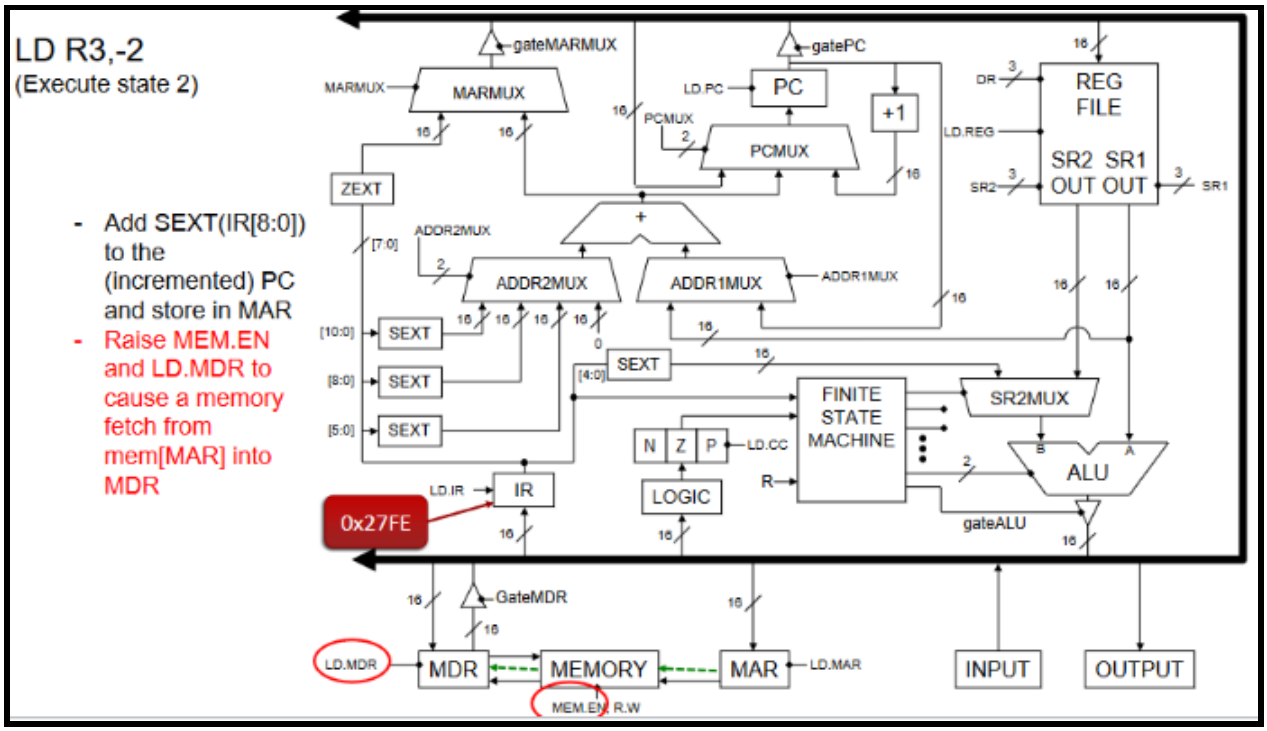


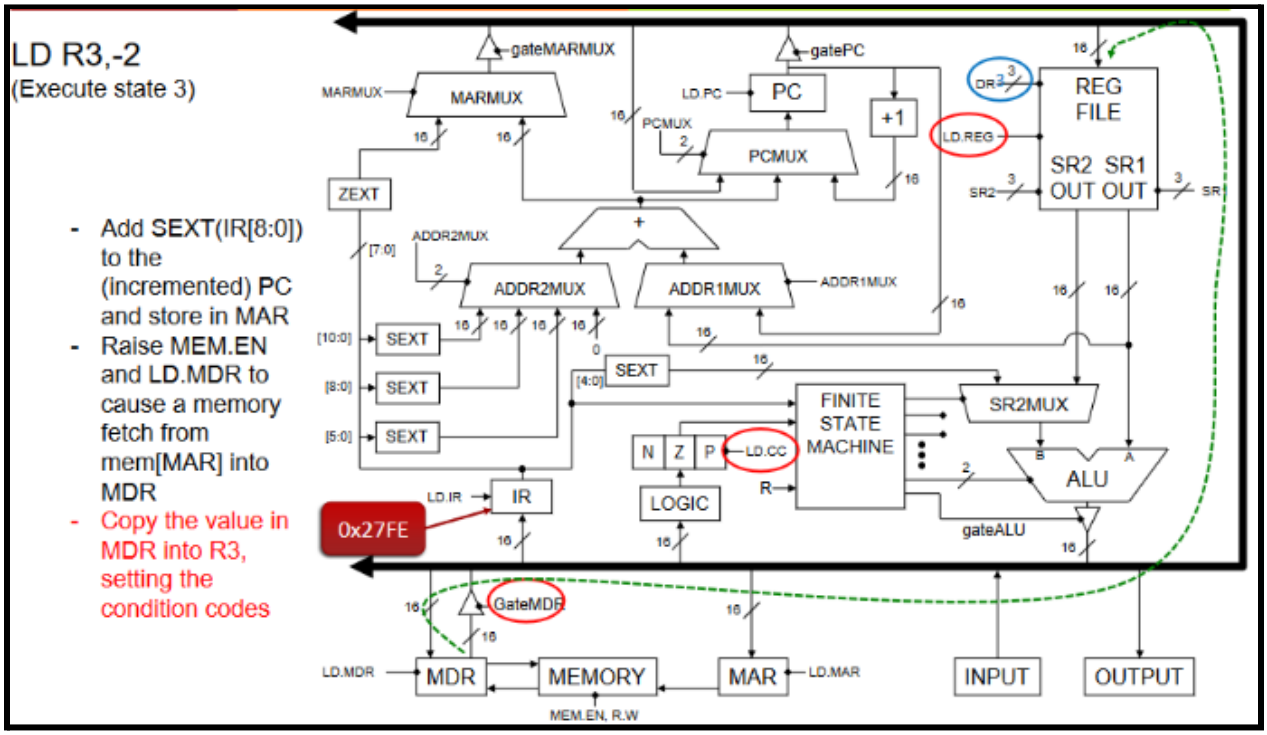
Good luck on the final everyone!
Extra study guide I made here



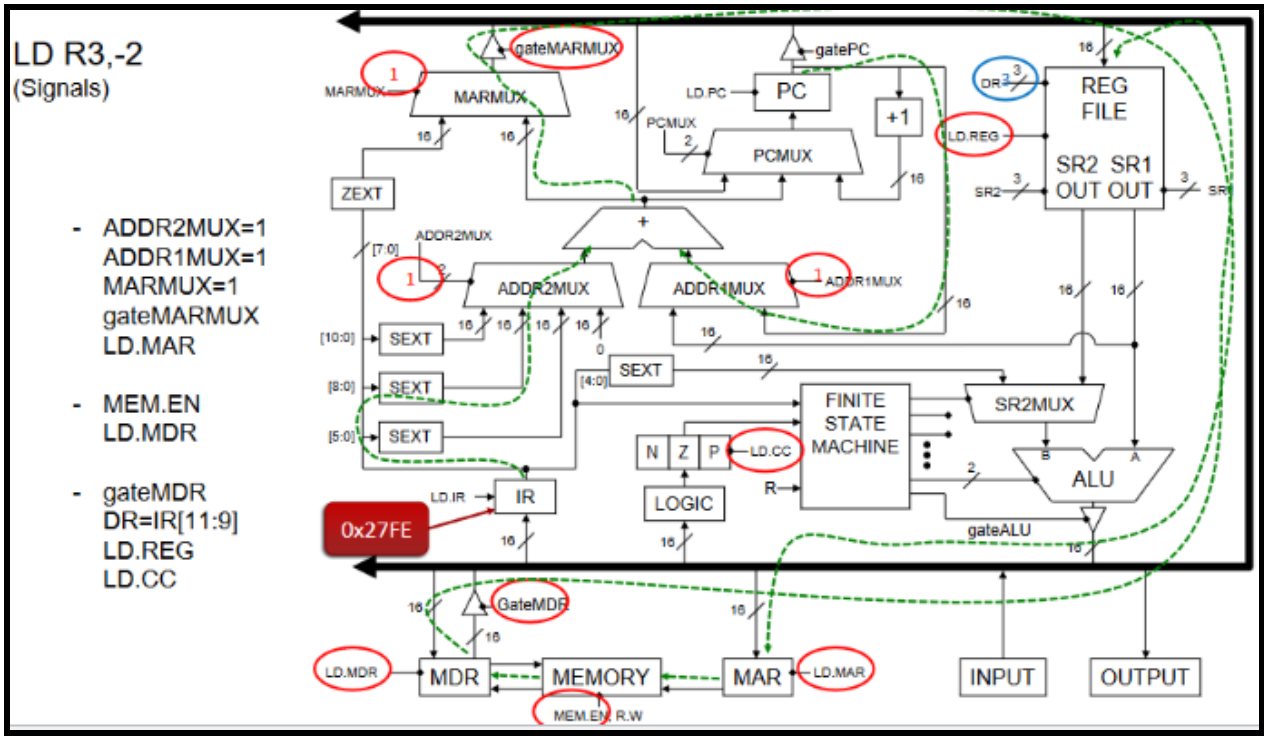


Good luck on the final everyone!
Extra study guide I made here

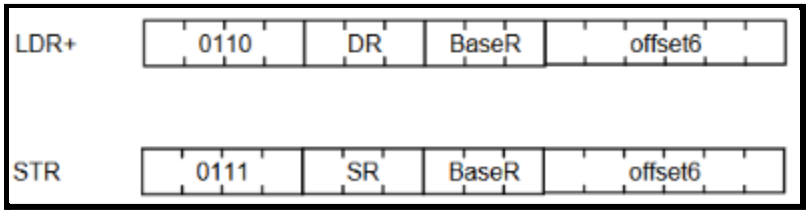




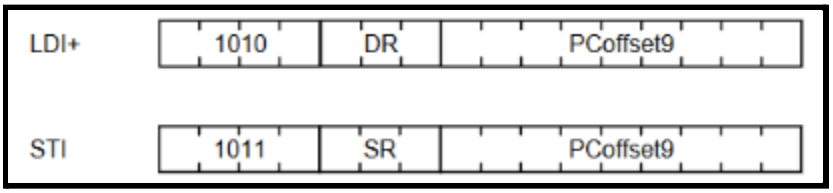
Good luck on the final everyone!
Extra study guide I made here



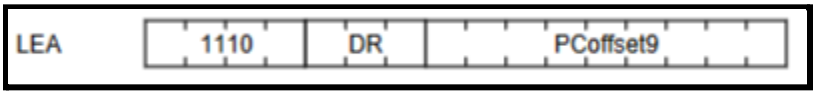
- Here's how the instructions would look with Base + Offset addressing mode…



  - Note that the phase 2 and 3 of the instruction are the exact same, we simply have to change ADDR1MUX so that we change it from getting the PC to the register
- Here's how the instructions would look with indirect addressing mode…



  - This one takes 5 states, because once we have the memory address from MDR we have to go back into MAR to obtain the other memory address
- Here's how the instruction would look with the immediate addressing mode…



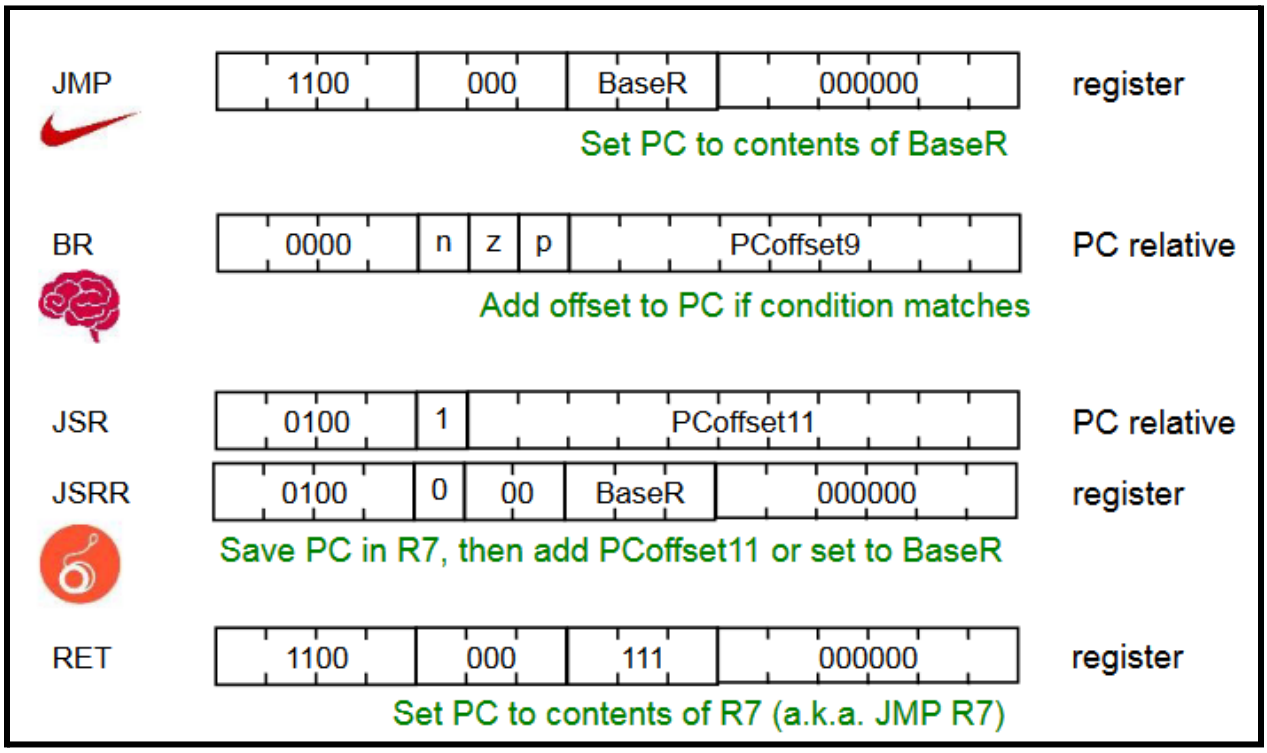
  - This one only takes 1 state.
- Storing to memory also takes **three cycles**…
  - Send the address to MAR

Good luck on the final everyone!
Extra study guide I made here

- Send the data to MDR
- Store MDR into memory[MAR]

Control Instructions (Loops & Conditionals)

- In FETCH, we increment the PC by 1
- If we don't want to execute the next instruction that follows, we need to change this
- There are two types of control instructions…
  - *Jumps* - which are unconditional; they always change PC
  - *Conditionals* - which are conditional; they may or may not change PC
- Let's review these types of control instructions…



- Condition Codes
  - Because of the way 2's-complement was designed, it is never possible to be anything more than negative OR zero OR positive
  - This information can be incredibly useful for determining whether we should execute branch instructions
  - During EXECUTE, the CC register is set. It is *only* set by…
    - ADD, AND, NOT, LD, LDR, LDI
    - ***NOT LEA***

Good luck on the final everyone!
Extra study guide I made here

L08 - Assembly

- Machine code is 16 bits, assembly is simply the human-readable equivalent to that
- Trying to approach a problem and thinking it through only in assembly is a *huge* pain. For the purposes of this course, we are going to think of it in languages we understand, and then do 1-to-1 conversions into assembly.
    - This means it won't be the most efficient, but that doesn't matter; it works!
- Let's take a look at some of the things in assembly…
    - A **label** is sort of like a variable name, except it references a block of code (if you're taking CS 4240, you should be well informed on labels in IR)
        - Think of the label like an *alias* to that memory location. Instead of representing the memory address as it's long string of bits, the label is used to *represent* that address

```
; Do until R1 <= 0
AGAIN ADDR3, R3, R2   ; Summing into R3
      ADDR1, R1, #-1 ; Dec loop counter
      BRPAGAIN
      HALT
```

    - A **comment** is a semicolon followed by the text of your comment
    - In order to denote **number bases** (decimal, binary, hex), use the following prefixes…
        - Decimal = #
        - Binary = b
        - Hex = x

```
Mem Addr   Hex      Dec
3050:  x2207   8711          LD R1, SIX
3051:  x2405   9221          LD R2, NUMBER
3052:  x56E0   22240         AND R3, R3, #0
3053:  x16C2   5826   AGAIN  ADD R3, R3, R2
3054:  x127F   4735          ADD R1, R1, #-1
3055:  x03FD   1021          BRP AGAIN
3056:  xF025   -4059         HALT
3057:  x0000   0      NUMBER  NOP *
3058:  x0006   6      SIX     NOP *
```

Good luck on the final everyone!
Extra study guide I made here

- NOTE: In machine code, we can use PC relative instructions and use an offset contained within our instruction. Well, in assembly, instead of writing out the offset ourselves, we can simply *use a label* that represents that offset, and the assembly compiler will turn that into the PCOffset for us!
    - This fact turns out to be **incredibly** useful when writing in assembly, because all instructions that used PCrelative modes, we can now use a label

Assembler Directives

- `.orig`
  - Where to put the data to be assembled
- `.fill`
  - Initialize one location
- `.blkw n`
  - Set aside n words in memory
- `.stringz "sample"`
  - Initialize 7 locations (sample + 0 word)
- `.end`
  - End of assembly program or block

- Example of .fill

Code		Memory	
	.orig x3000		
A	.fill x3012	X3000:	x3012
B	.fill 21	X3001:	x0015
C	.fill 0	X3002:	x0000

- Example of .blkw
    - Very useful for allocating place for arrays

```
ARR_A       .blkw 1000

ARR_B       .blkw 3
```

Good luck on the final everyone!
Extra study guide I made here

- Example of .stringz
  - Remember that a string is effectively an array of characters, and a character can be represented by an ASCII integer value, so a string is really an array of characters
  - In assembly, you can create a string by filling an array with characters…

```
message.fill 'H'
    .fill 'e'
    .fill 'l'
    .fill 'l'
    .fill 'o'
    .fill 0
```

    - The 0 at the end there is the **null character**, and every string ends with it to denote the fact that the string is complete
  - Assembly let's us replace the integers with a single quoted character. This is convenient, but what if we wanted to write out a lengthy sentence? It'd get pretty annoying really quick.
  - This is where .stringz comes in, it does everything for us…

```
message.stringz "Hello"
```

    - It automatically adds the null character for us, too!

Two-Pass Assembly

- Assembly allows you to reference labels that *haven't been made yet*. How is this possible?
- The assembler does *two*-passes through the code, one to scrape all the potential labels that it can reference, and the other to convert to machine language
- This is kind of different to most other programming languages; it'd be like being able to initialize a variable to a value on line 12, and then declare it on line 13.
- First Pass - Generates the symbol table
  - Scan each line
  - Keep track of the current address (location counter)
    - Increment each instruction by 1
    - Adjust as necessary for pseudo-ops (.fill, .stringz, etc.)
  - For each label, enter it into the symbol table along with the current address (location counter)
  - Stop when .end is encountered

Good luck on the final everyone!
Extra study guide I made here

Symbol	Address
AGAIN	X3053
NUMBER	X3057
SIX	X3058

```
                ;
                ; Program to multiply a number by  six
                ;
                        .orig    x3050
x3050                   LD  R1, SIX
x3051                   LD  R2, NUMBER
x3052                   AND     R3, R3, #0
                ;
                ; The inner loop
                ;
x3053           AGAIN   ADD     R3, R3, R2
x3054                   ADD     R1, R1, #-1
x3055                   BRP     AGAIN
                ;
x3056                   HALT
                ;
x3057           NUMBER  .blkw   1
x3058           SIX     .fill x0006
                ;
                        .end
```

- Second Pass - Generating the ML program
  - Scan each line again
  - Translate each AL instruction into ML
    - Look up symbols in the symbol table
    - Ensure that labels are nore man +256/-255 lines away from PCoffset9 instructions
    - Calculated operand field for the instruction
    - Update the current address (location counter)
  - Fill memory addresses as directed by pseudo-ops
  - Stop when .end is encountered

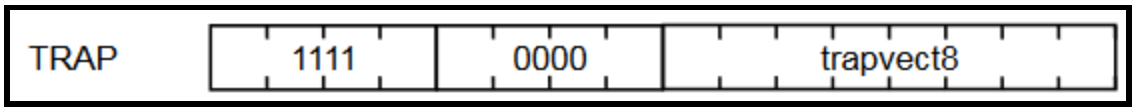
Symbol	Address
AGAIN	X3053
NUMBER	X3057
SIX	X3058

```
x3050   0010 001 0 0000 0111    ; LD R1, SIX
x3051   0010 010 0 0000 0101    ; LD R2, NUMBER
x3052   0101 011 011 1 00000    ; AND R3, R3, #0
x3053   0001 011 011 0 00 010   ; ADD R3, R3, R2
x3054   0001 001 001 1 11111    ; ADD R1, R1, #-1
x3055   0000 001 1 1111 1101    ; BRP AGAIN
x3056   1111 0000 0010 0101     ; HALT
x3057                           ; .blkw 1
x3058   0000 0000 0000 0110     ; .fill x0006
```

Aliases

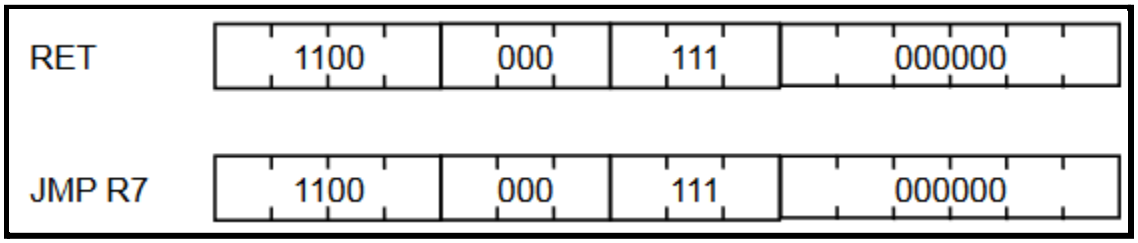
- To make things more readable to humans, certain instructions have *aliases* that we can use in assembly
- TRAP is a good example of this, there are six different aliases for the instruction TRAP…

Good luck on the final everyone!
Extra study guide I made here

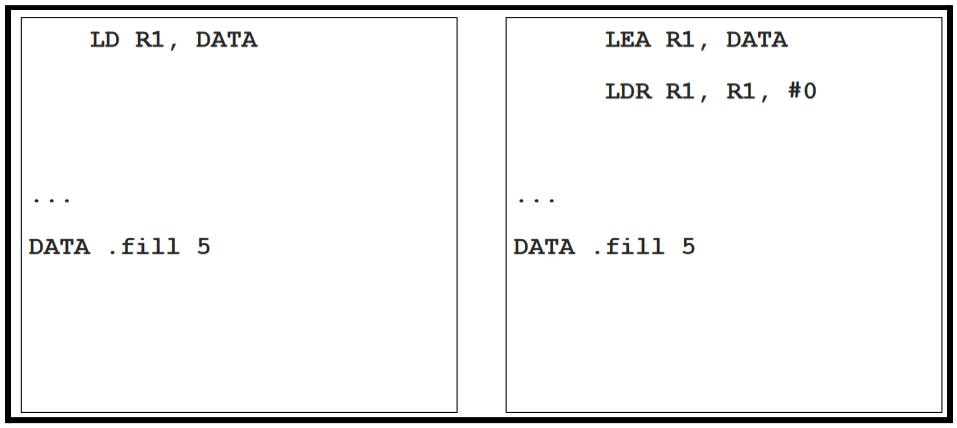


- TRAP x25 ⇒ HALT: Stop the CPU
- TRAP x23 ⇒ IN: input character from keyboard
- TRAP x21 ⇒ OUT: Print character on screen
- TRAP x22 ⇒ PUTS
- TRAP x24 ⇒ PUTSP
- TRAP x20 ⇒ GETC

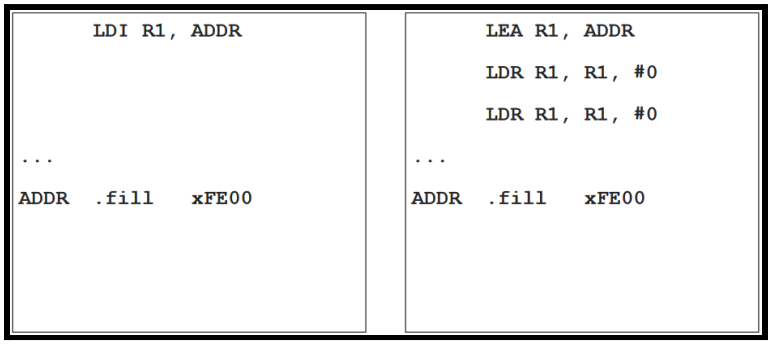
- Another example of this is RET. RET doesn't actually exist, it's just an alias for JMP



- Think of all of our LD and ST instructions (LDR, LDI, STR, STI). Knowing what we know about aliases, we actually don't *really* need all these variations. In fact, all we need is LEA, LDR, and STR.
- Look at the example below for LD. We can actually do the operation for LD with only LEA and LDR.



- Here's another example showing we don't need LDI either…



Good luck on the final everyone!
Extra study guide I made here

L09 - Assembly Programming

- What are all the things we can do with ADD, AND, and NOT?
  - Add
  - And
  - Not
  - Subtract
  - Or
  - Clear a register
  - Copy from one register to another
  - Increment a register by n
- Subtraction

```
;SUB R1, R2, R3 ;instruction does not exist
; how to subtract
NOT R3, R3 ;flip the bits of R3
ADD R3, R3, #1 ;add 1 to R3 ;now R3 is -R3
ADD R1, R2, R3
```

- OR

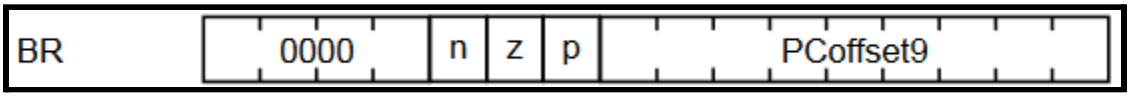
```
;OR R1, R2, R3 ;does not exist: R1=R2 | R3
; how to do OR ;use DeMorgan's Law
NOT R2, R2 ;R2 = ~R2
NOT R3, R3 ;R3 = ~R3
AND R1, R2, R3  ;R1 = ~R2 & ~R3
NOT R1, R1 ;R1 = ~(~R2 & ~R3)
```

- Miscellaneous

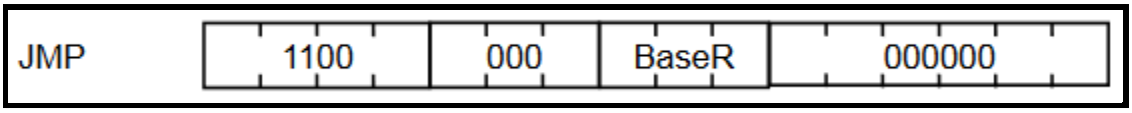
```
;CLR R1 ;e.g. set R1 = 0
AND R1, R1, #0 ;R1 = R1 & 0
;anything and zero is zero


;MOV R1, R2 ;e.g. R1 = R2
ADD R1, R2, #0 ;R1 = R2+0; R1 = R2
```

Good luck on the final everyone!
Extra study guide I made here

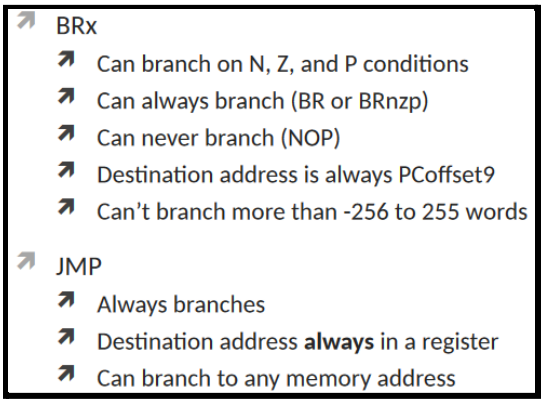
Branch & Jump



- We've seen branches before in previous coding languages. Branches are used for…
  - If then else
  - Loops
- With BR in assembly, we can do many things..
  - If then else
  - For loops
  - While loops
  - Do while loops
  - Conditional branch
  - Unconditional branch (BRNZP or BR)
  - Never branch (NOP)



- Jump is also useful in assembly…
  - Go to
  - Branch long distances
- Although, we will oftentimes use branch over jump for the cases we'll be handling
- Differences



- How to IF using BR
  - First, we need to do an operation to set the CC's (condition codes)
  - Then, BR with the appropriate combination of NZP conditions in the instruction

- "Every condition in assembly can be reduced down to a comparison to 0"
  - *Less Than* <
    - $A < B \Rightarrow A - B < B - B \Rightarrow A - B < 0$
  - *Equal To* =

Good luck on the final everyone!
Extra study guide I made here

- A == B ⇒ A - B == B - B ⇒ A - B == 0

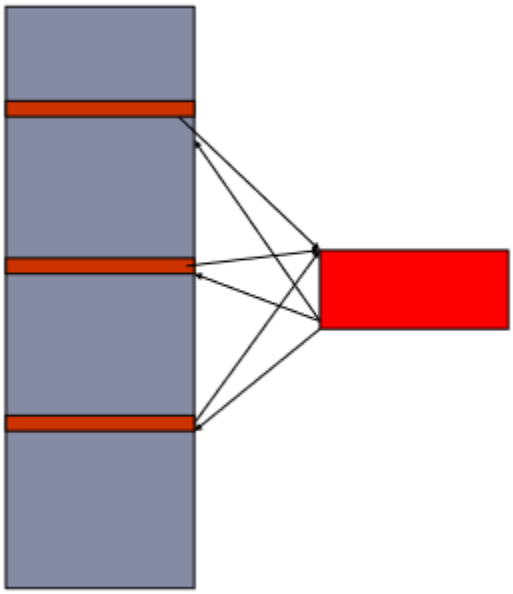
BR (branch conditions)	
<	N
<=	NZ
==	Z
!=	NP
>=	ZP
>	P
Always	NZP (we abbreviate BRnzp as just BR, branch always)
Never	(no condition codes), we use NOP (no operation)

- Usually when we want to start a coding project, we jump right into the code and start writing. In assembly, this doesn't really work. We need to do some steps before writing in assembly to make things easier
- "Act Like a Compiler" Approach to Assembly
    - Write your algorithm in a high-level language
    - Write down where you're going to store your variables (register, static memory, stack offset, etc.)
        - Use comments for registers and stack offsets
        - Use assembler directives to reserve memory
    - Copy your algorithm with ";" at the beginning of each line to create assembly language comments
    - Translate each statement into a stanza of the appropriate machine language instructions
    - Make each stanza independent from each other
    - This will NOT result in efficient code, but it will be *correct*
- Just know how to program assembly lul

Good luck on the final everyone!
Extra study guide I made here

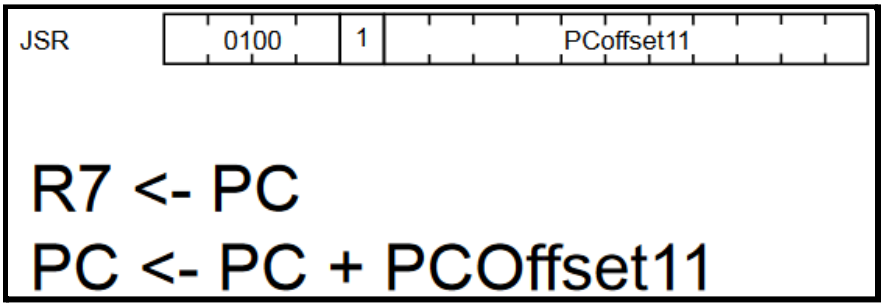
L10 - Subroutines and Stacks

- Since assembly isn't an object oriented language, "methods" are called subroutines, but the following all essentially mean the same thing…
  - Methods
  - Functions
  - Procedures
- But in assembly, we will stick exclusively to **subroutines**
- In essence, subroutines are about reusing code (remember: if you copy the same line of code over and over again, and there's a problem with it, you have to fix it in every occurrence. In a subroutine, you only have to fix it once)

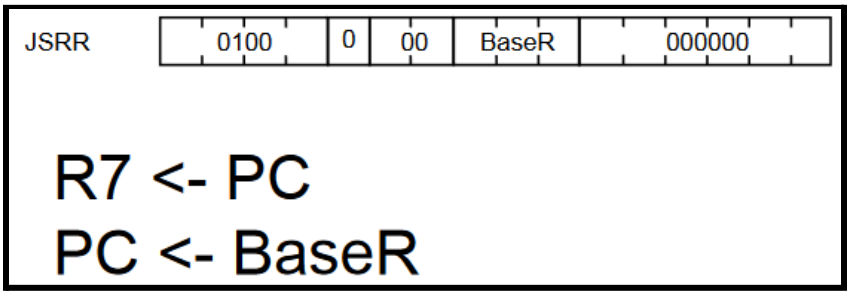


How Subroutines Work with PC

- JSR stands for *jump to subroutine*
- When we need to go to a subroutine, we need to keep track of where we are in the instructions. In order to do this, we need to store the current value of PC somewhere.
- By convention, we store this value into R7



- This is important, because we now effectively have *one less register* to work with
- JSRR does the same thing…



- This should be a little full circle now, because what is our RET instruction? Well, it's an alias for our JMP instruction, but it exclusively jumps to **R7**

Good luck on the final everyone!
Extra study guide I made here

RET	1100	000	111	000000

Short and Long Reach Subroutines

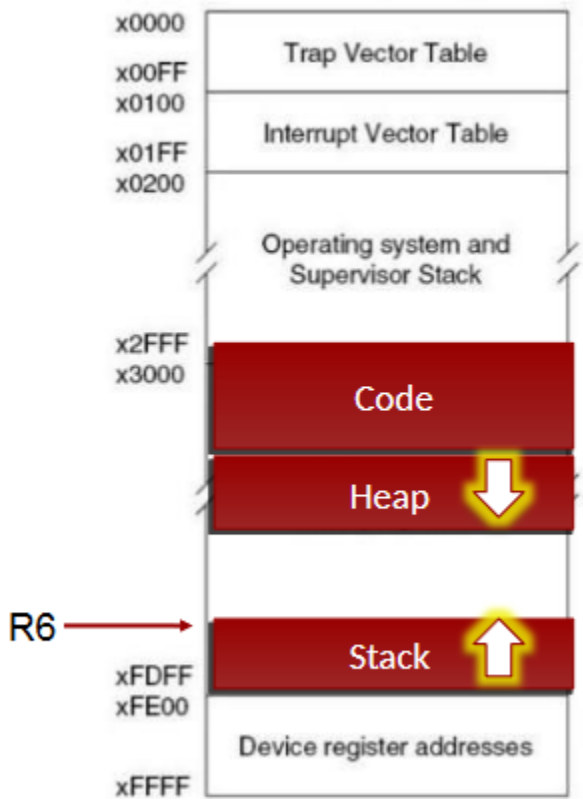
- There are two ways to write a subroutine, array, etc.
- We can either store it very close, close enough for it to fit into the PCOffset slot of the instruction, or we write it very far away and represent the address to it with something closeby that CAN fit in PCOffset

```
    jsr sub



sub   xxx xxx xxx
...
...
...
    ret
```

```
    ld R2, far
    jsrr r2



far   .fill sub
...
...
sub   xxx xxx xxx
    ret
```

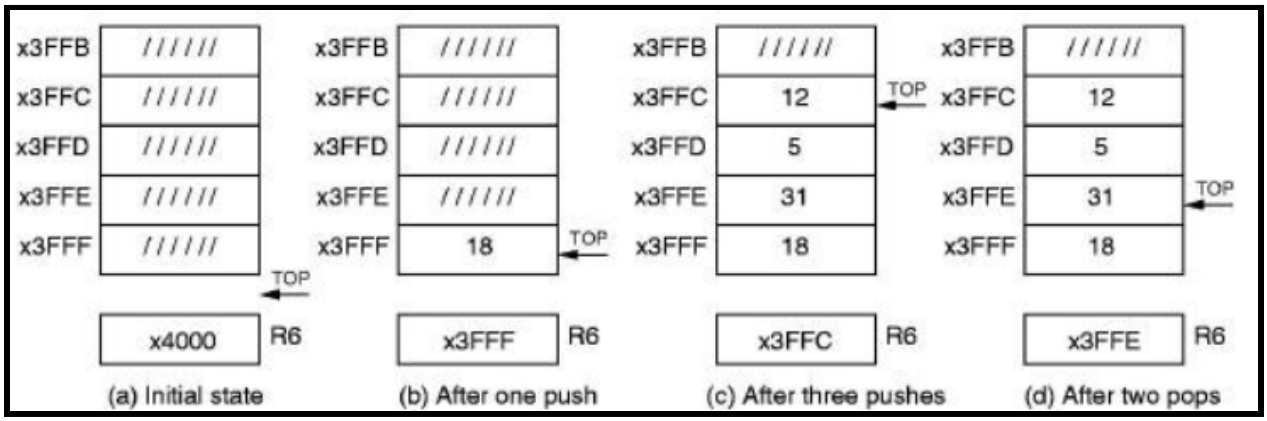
How to Efficiently Write a Subroutine

- When we write a subroutine, there are many problems we need to consider that we normally wouldn't have to consider otherwise
    - Issue #1: What happens if we call another subroutine inside our subroutine?
        - We would normally lose our return address in R7, so we need to find a way to preserve it
    - Issue #2: We only have so many registers.
        - What happens if we want to LD and ST data in a recursive subroutine? The earlier recursive calls would get trampled over.
- So we need to save two things…
    - Old values in registers
    - Our return addresses

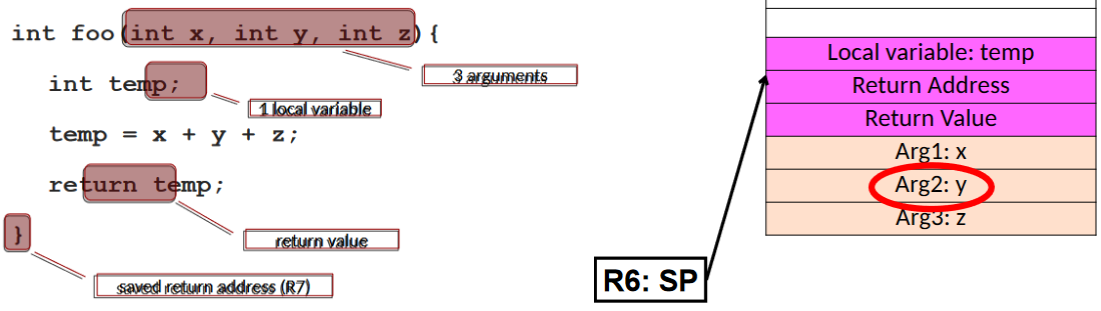


Good luck on the final everyone!
Extra study guide I made here

- To solve this, we can use a **stack** stored in memory!
    - Remember that a stack is a LIFO structure (last-in, first-out)



    - Also note that after we pop, the data is still there. That's because it's *easier* to not have to do this; it's dead data, nothing will be able to reach it.
    - To keep track of this stack, we need something that points to the top of the stack, a *stack pointer*, which by convention is stored in **R6** (yet another register taken from us)
- Stacks have two main functions for us to use: *pushing* and *popping*
    - *Push*: Decrement stack pointer (our stack is growing *down* to *lower* memory addresses; look back at the figure above)
    - *Pop*: Increment stack pointer
        - Read data at current top-of-stack into R0, then increment the stack pointer
- What happens if we run out of space in memory for the stack? We get a **stack overflow**.
- So the stack is going to hold all the information we need about the subroutine, called a **stack frame** (**activation record**). Specifically, the *stack frame* holds the following…
    - Arguments
    - Return values
    - Local variables
    - Saved return address (R7)
    - Saved registers



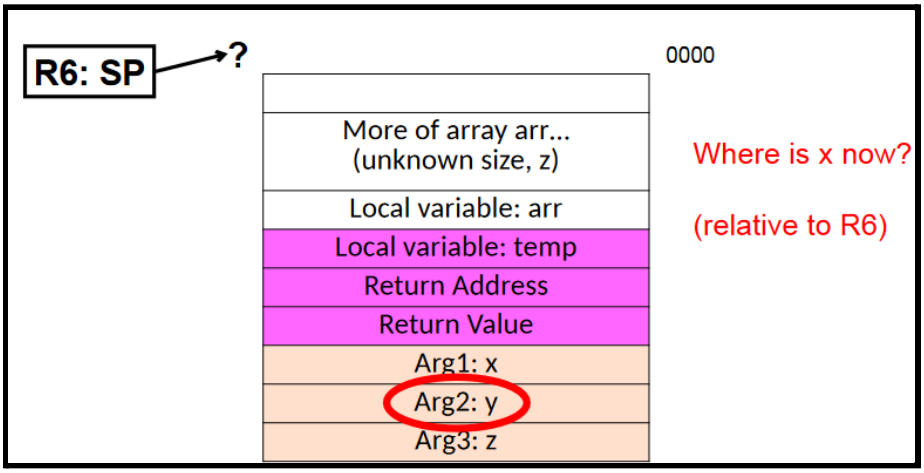
- Above here, we can see an example of using the stack frame.

Good luck on the final everyone!
Extra study guide I made here

Stack Frame & the Issues that Arise

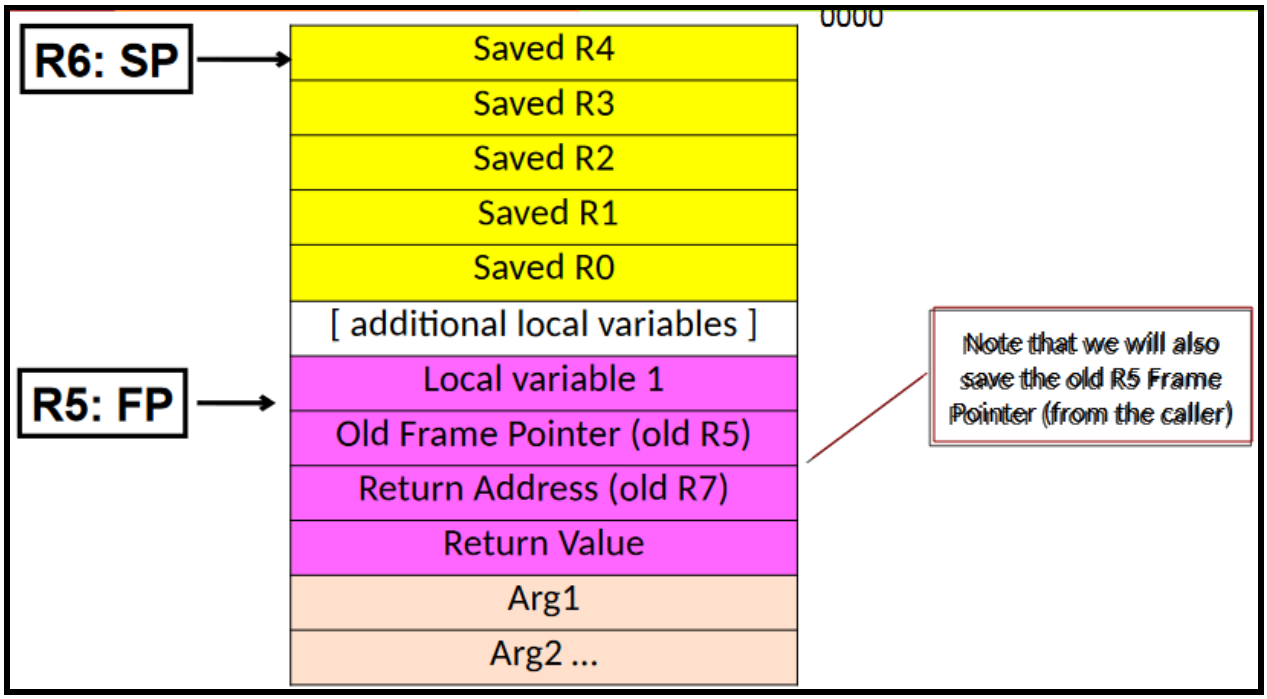
- In the above example, it seems pretty simple to access the arguments, it's just SP + an offset
- But what happens if we *don't know* the offset we need until runtime? See the example below. Notice how the array's dimension is determined by an input…

```
int foo(int x, int y, int z) {

   int temp;

   int arr[z];

         ...

   x = 1;

      ...

   }
```



- The solution to this is using another pointer: a **frame pointer**.
    - The frame pointer will be stored in R5, and it's **always** going to point to the first local variable.
        - If we don't have any local variables, by convention, we leave an empty space such that our frame pointer has somewhere to point to.
    - We say the frame pointer is an *anchor*, because it always points to a predictable location
- So now, we can't use R7, R6, and R5, because they are all reserved for things pertaining to subroutines.

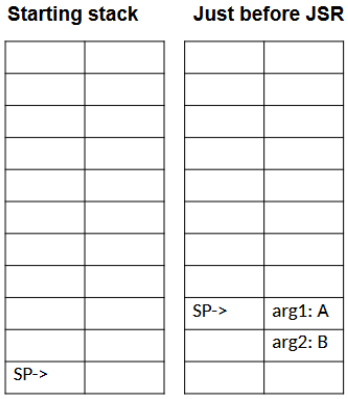
Good luck on the final everyone!
Extra study guide I made here



- The above is **the** stack frame of the LC-3. The way it's formatted is called a **calling convention** (going back to the idea that everything in a computer is decided by us).
- There are infinitely many ways to do a calling convention, but for us, we will follow the format above

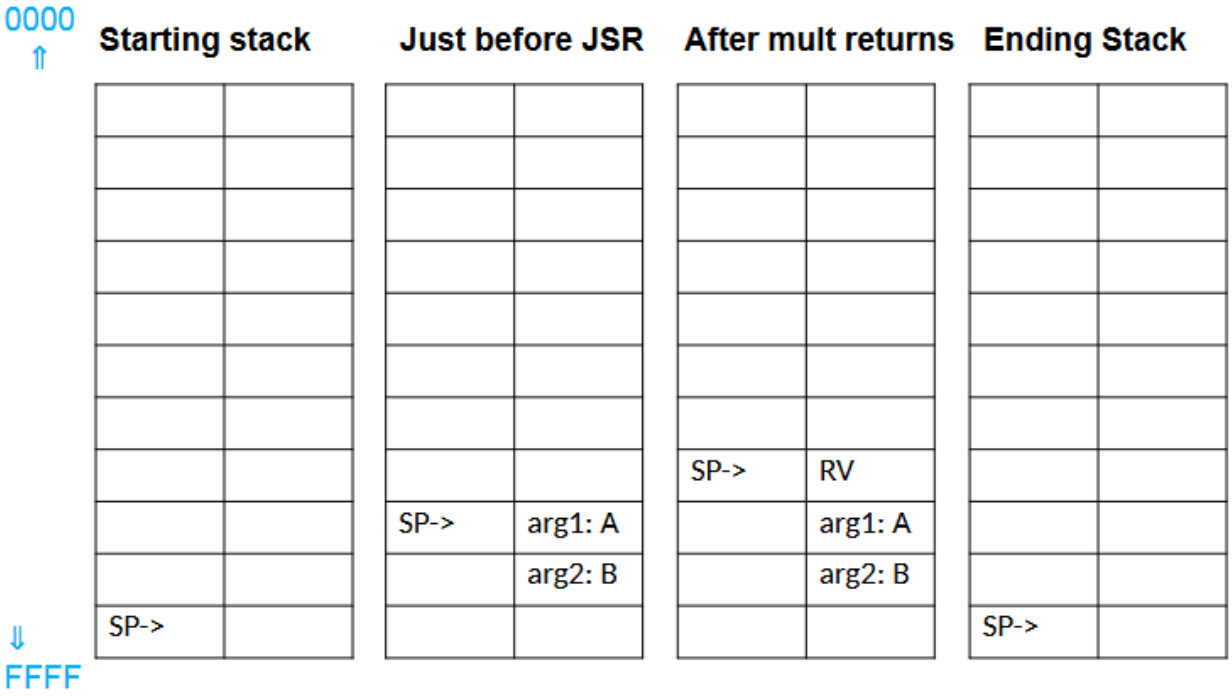
Caller & Callee

- For subroutines, there are two main definitions we need to define…
  - **Caller** - the code that calls the subroutine
    - E.g. m = mult(a, b)
  - **Callee** - the subroutine definition
    - E.g. int mult(a, b) { return a*b }
- Caller
  - When we call a subroutine, the first thing we do is push the arguments (in reverse order)



Good luck on the final everyone!
Extra study guide I made here

- Then we run our JSR instruction.
- When our subroutine returns, first we pop the return value, and then pop the arguments.

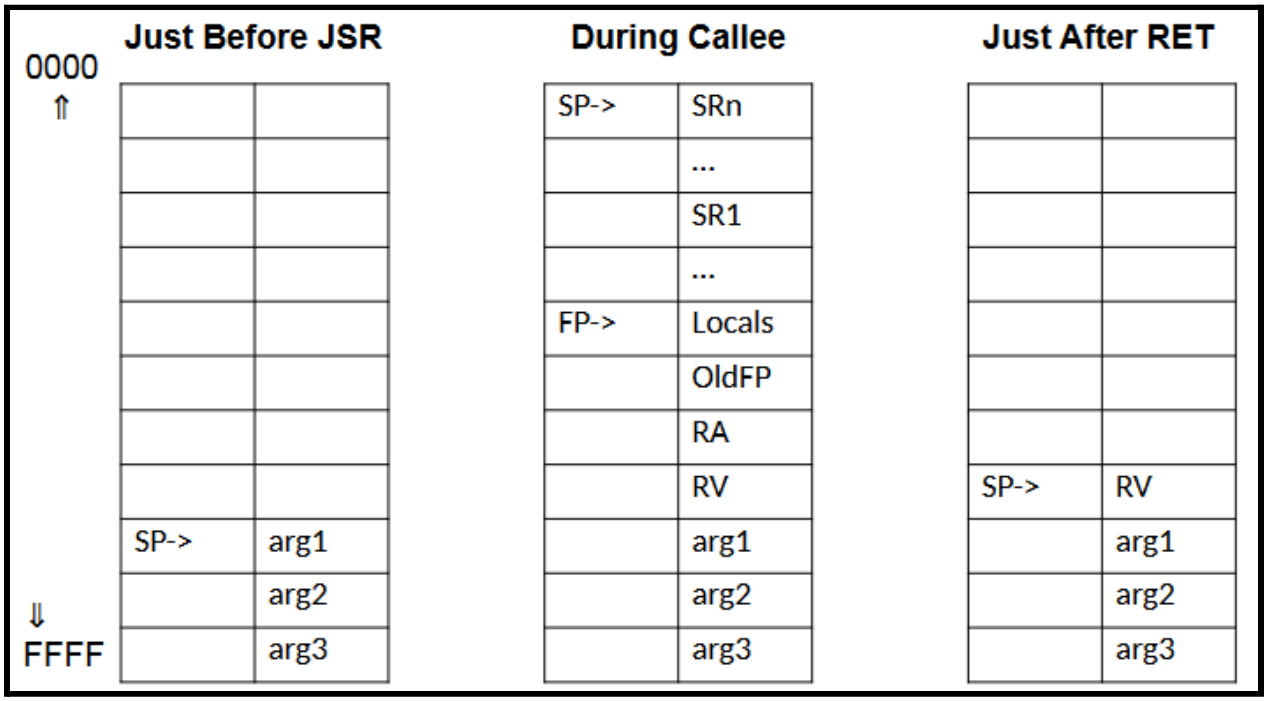


  - Take note of the symmetry: the stack ends up exactly like we started. If this wasn't the case (i.e. we left junk in the stack), we'd eventually get a *stack overflow*
- Here's an example of us manipulating the stack pointer in assembly using instructions…

```
; let's call mult
; M = mult(A,B);
; assume M, A, B, MULT are
;   labels

; push arguments
;   in reverse order
; push B
LD R1, B
ADD R6, R6, #-1
STR R1, R6, #0
; push A
LD R1, A
ADD R6, R6, #-1
STR R1, R6, #0
; call mult
JSR MULT
; after MULT returns

; pop the return value
LDR R1, R6, #0
ADD R6, R6, #1
; save the ret val in M
ST R1, M
; pop the two args, A and B
ADD R6, R6, #2
```

Good luck on the final everyone!
Extra study guide I made here

- Callee
  - When a subroutine is called, we need to create the stack frame. This step is called the **stack buildup**.
    - Save some registers
    - Make room for local variables
  - Then, we can do the work of the subroutine
  - Before the procedure is returned, we need to remove the stack frame. This step is called the **stack teardown**
    - Prepare return value
    - Restore registers
  - Finally, we return
  - Let's go back to those charts to get a visual example. Once again, pay attention to the symmetry…



- Highly advisable to go through the mult example Southern did in class

Good luck on the final everyone!
Extra study guide I made here

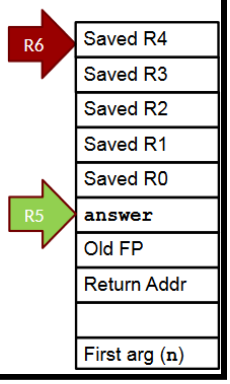
L11 - Recursive Subroutines

- Recursion is really not that much different compared to what we'd normally do for the stack
- Here's the example pertaining to a factorial function…

```
int fact(int n)
{
   int answer;
   if(n <= 0)
      answer = 1;
   else
      answer = n * fact(n-1);
   return answer;
}
```

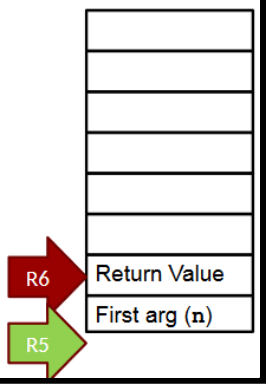
- In the callee, just like we did before in mult, we have to do the stack build up…

```
FACT    ADD  R6, R6, -4   ; Allocate space
        STR  R7, R6, 2 ; Save Ret Addr
        STR  R5, R6, 1 ; Save Old FP
        ADD  R5, R6, 0 ; Copy SP to FP
        ADD  R6, R6, -5   ; Room for 5 regs
        STR  R0, R5, -1
        STR  R1, R5, -2
        STR  R2, R5, -3
        STR  R3, R5, -4
        STR  R4, R5, -5
```



- Similarly, just like we did in mult, we have to do the stack teardown…

```
LDR  R0, R5, 0    ; Ret val = answer
STR  R0, R5, 3
LDR  R4, R5, -5   ; Restore R4
LDR  R3, R5, -4   ; Restore R3
LDR  R2, R5, -3   ; Restore R2
LDR  R1, R5, -2   ; Restore R0
LDR  R0, R5, -1   ; Restore R1
ADD  R6, R5, 0 ; Restore SP
LDR  R5, R6, 1 ; Restore FP
LDR  R7, R6, 2 ; Restore RA
ADD  R6, R6, 3 ; Pop ra,fp,lv1
RET
```



Good luck on the final everyone!
Extra study guide I made here

- So now we have the beginning and end of FACT. Effectively, we've taken care of what it's red…

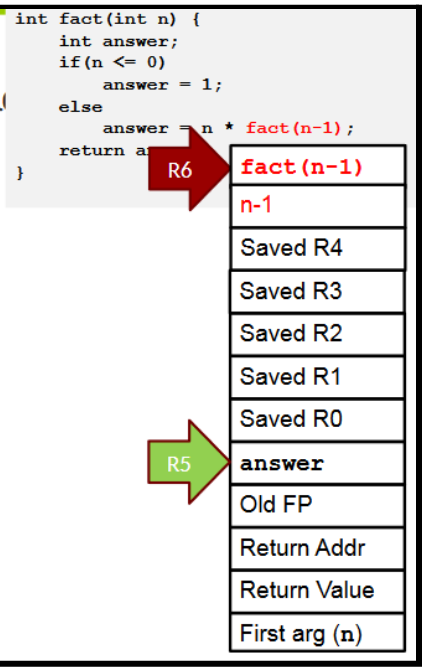
```
int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}
```

- Now we just need to write the method. For simplicity, let's skip to the recursive step.
- It would be pretty difficult to write the line in the else simply, so let's break it down to make it easier…

```
;  answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0);
```

- Now we can tackle this case by case. In order to do the recursive step, we simply do what we would normally do as a caller, and put that inside the callee…

```
;  answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R
IFELSE1 LDR  R0, R5, 4 ; Push n-1
     ADD  R0, R0, -1
     ADD  R6, R6, -1
     STR  R0, R6, 0
     JSR  FACT    ; fact(n-1)
```

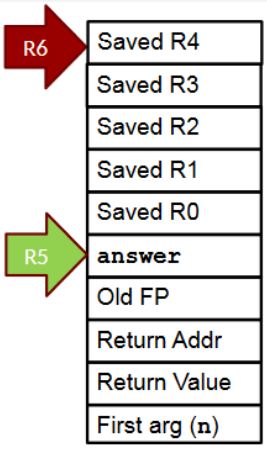


- Notice that the above pushes our first argument (n - 1) and then simply calls itself, FACT.
- So now we've just did our recursive call, and remember that both the caller and the callee have steps that need to be done before and after in order to keep the stack clean
- Just like we would have to do lines of code after we call FACT outside the method, we also need lines of code after our call of FACT inside the method. In this case, we need to get our return value and pop off our argument from the stack…

Good luck on the final everyone!
Extra study guide I made here

```
;  answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R
IFELSE1 LDR  R0, R5, 4 ; Push n-1
     ADD  R0, R0, -1
     ADD  R6, R6, -1
     STR  R0, R6, 0
     JSR  FACT    ; fact(n-1)
     LDR  R0, R6, 0 ; R0 = rv
     ADD  R6, R6, 2 ; Pop rv and arg1
```

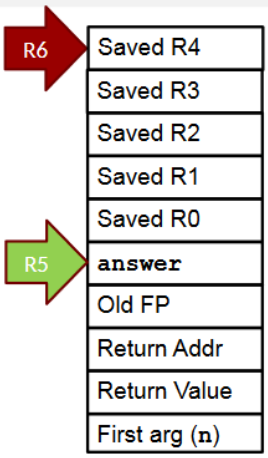
```
int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}
```



- Now we have the return of our recursive call in R0, and so then we just need to call mult(n, R0), store that in answer, and we're done.
- Follow the same steps: treat our callee (FACT) like a caller and do the steps to push the arguments to the stack (in reverse order!) and then call MULT.

```
;  answer = mult(n, R0)
     ADD  R6, R6, -1; Push R0
     STR  R0, R6, 0
     ADD  R6, R6, -1; Push n
     LDR  R0, R5, 4
     STR  R0, R6, 0
     JSR  MULT    ; mult(n, R0)
     LDR  R0, R6, 0 ; answer = rv
     STR  R0, R5, 0 ;
     ADD  R6, R6, 3 ; Pop rv and arg1-2
ENDIF1  NOP
; }

; Tear down stack frame template goes below
```

```
int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}
```

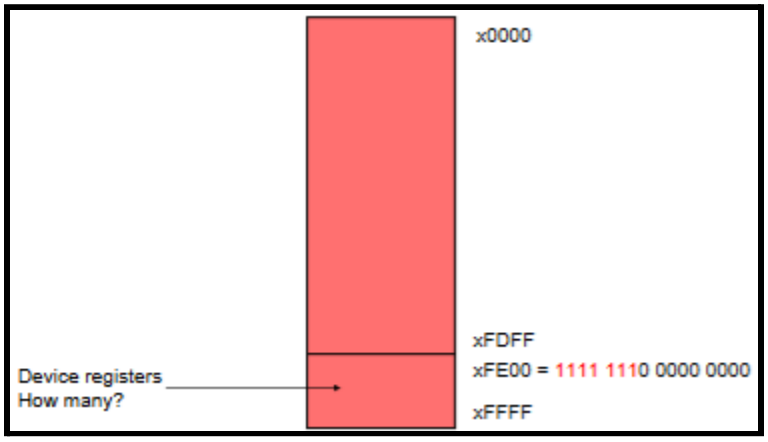


- That's literally it.

Good luck on the final everyone!
Extra study guide I made here

L12 - Input/Output

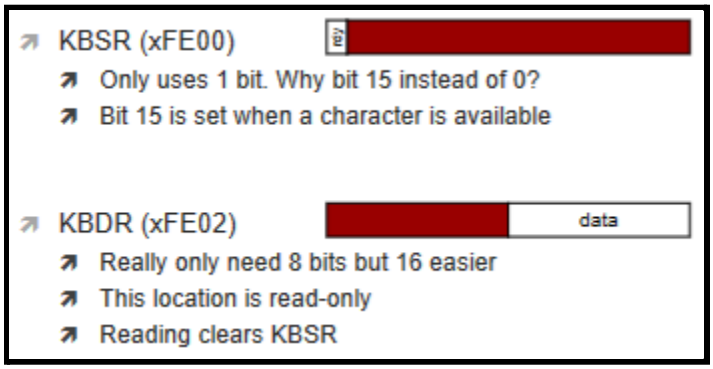
- There are two types of synchronicities…
  - *Asynchronous* - Electronic, mechanical and human speed; speed mismatch
  - *Synchronous* - Processor operation, certain kidneys of high speed I/O
- I/O events generally happen much more slowly than CPU cycles
  - *Synchronous* - data supplied at a fixed, predictable rate
  - *Asynchronous* - data rate less predictable, CPU must *synchronize* with device, so that it doesn't misty data or write to quickly
- Because of the asynchronous nature of I/O, there are two different ways we can approach adding I/O…
  - Special I/O Instructions
    - Need an opcode
    - Need to be general enough to work with devices that haven't been invented yet
    - lame
  - Memory-mapped I/O
    - Steal part of the address space for *device registers*
    - Operations done by reading/writing bits in device registers
    - pog
- Since we'll be focusing more on the memory-mapped I/O, let's take a look at the different device registers that our memory uses…
  - **Data registers**: used for the actual transfer of data (i.e. character code)
  - **Status registers**: Information the device is telling us
  - **Control registers**: Allows us to set changeable device characteristics
- Device registers are often part of the I/O device itself
- Two ways to handle I/O completion…
  - *Interrupt-driven*: Interrupts what the LC-3 is currently doing to go handle the I/O even, and then goes back to what it was doing
  - *Polling*: Continually asks if a character has been typed



Good luck on the final everyone!
Extra study guide I made here

Keyboard Input

- Two keyboard-related device registers that we need to know..
  - *KBSR* - Keyboard Static Register
    - 1 if a new key has been pressed, 0 otherwise
  - *KBDR* - Keyboard Device Register
    - Stores the ASCII value of the most recent key pressed

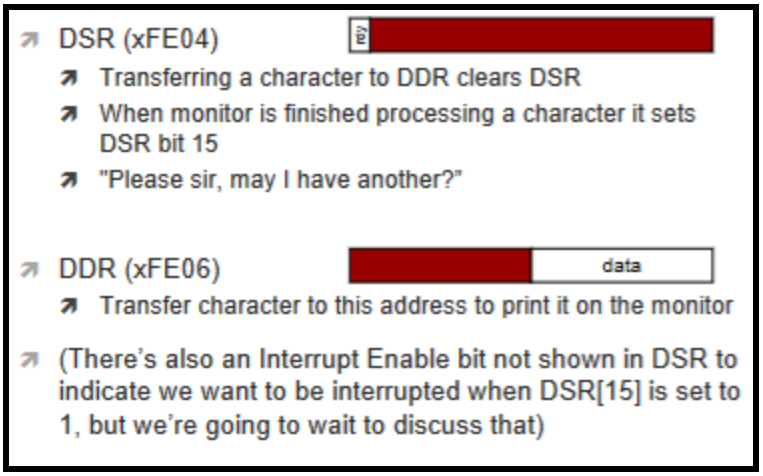



```
; Read characters from the keyboard until CTRL/Z

        .orig   x3000
        ld      r4, term
        lea     r2, buffer ; Initialize buffer pointer
start   ldi     r1, kbsrA  ; See if a char is there
        BRzp    start
        ldi     r0, kbdrA  ; get the character
        str     r0, r2, 0  ; Store it in buffer
        not     r0, r0     ; subtract R4-R0
        add     r0, r0, #1 ;   to check for termination
        add     r0, r0, r4 ;   char stored in R4
        brZ     quit
        add     r2, r2, 1  ; Increment buffer pointer
        br      start      ; Do it again!
quit    halt
term    .fill   x001A      ; CTRL/Z
kbsrA   .fill   xfe00
kbdrA   .fill   xfe02
buffer  .blkw   x0100
        .end
```

Monitor Output

- Two monitor-related device registers we need to know…
  - *DSR*: Display Status Register, indicates if the screen is ready to take another character
    - The screen is really slow compared to the LC-3 (think of the difference between megahertz and gigahertz versus just hertz)
  - *DDR*: Display Device Register, transfers the character in this address to print it on the monitor

Good luck on the final everyone!
Extra study guide I made here

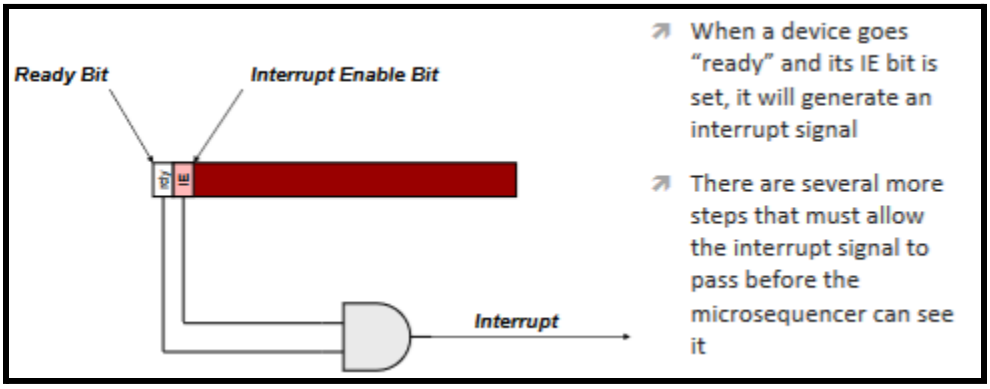



```
; Write the contents of a string to the display

        .orig   x3000
        lea     r2, buffer    ; Initialise buffer ptr
start   ldr     r0, r2, 0     ; Get char into r0
        brz     quit          ; Terminate on null
wait    ldi     r3, dsrA      ; Are we ready?
        brzp    wait
        sti     r0, ddrA      ; Send R0 to monitor
        add     r2, r2, 1     ; Move buffer ptr over 1
        br      start
quit    halt
dsrA    .fill   xfe04
ddrA    .fill   xfe06
buffer  .stringz "Hello, World!"
        .end
```

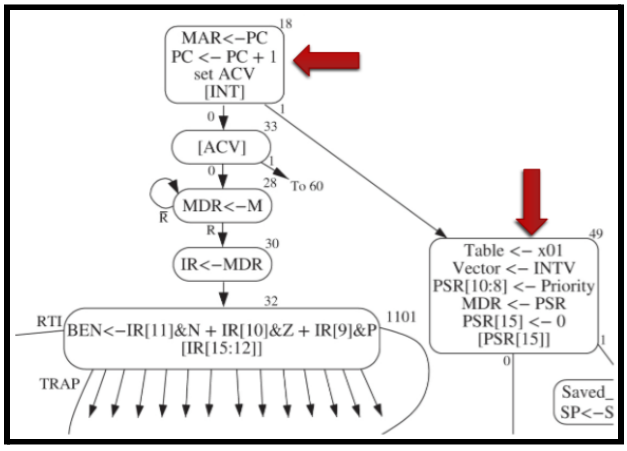
Good luck on the final everyone!
Extra study guide I made here

L13 - Program Discontinuities (Interrupts, TRAPs, and Exceptions)

- Here are the three different types of program discontinuities…
  - *Interrupts* - An I/O device is reporting a completion or an error (e.g. "Read completed")
  - *TRAPs* - The program is calling a privileged operating system subroutine (e.g. "Read a line from a file")
  - *Exceptions* - Something unanticipated has happened
    - Hardware error in the CPU or memory
    - Program error (e.g. illegal opcode, divide by zero)
- Our device status registers also have a reserved *Interrupt Enable Bit* that determines whether or not it can be interrupted



- When do we interrupt? If we interrupt in the middle of an instruction sequence (i.e. between LDR1 and LDR2), then things would get really, really messy. Instead, we can only interrupt before we start fetching…

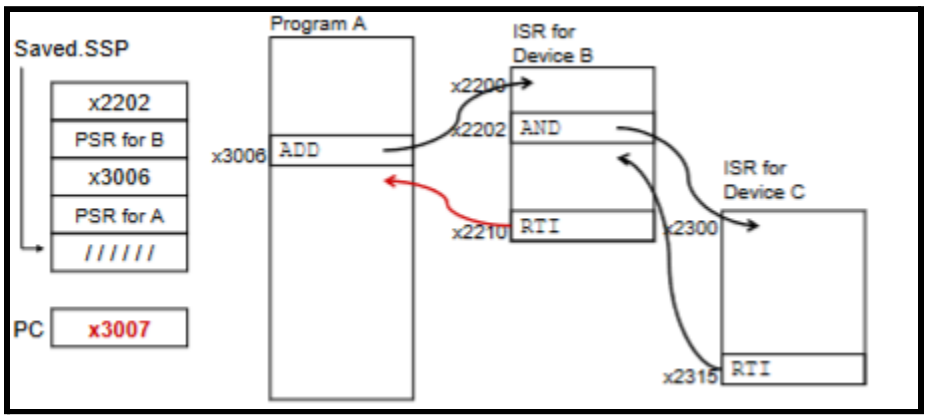


- If we interrupt, where do we save our current state? Well, we can just do what we did for subroutines: use a stack!
- We use something called the supervisor stack, which is reserved by the operating system; normal users are not allowed to access it
- Alright, so we've interrupted and saved our state. How do we get back to our state?

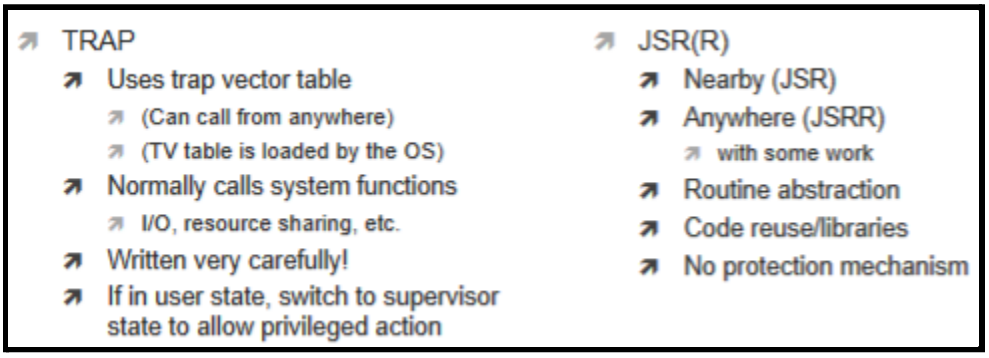
Good luck on the final everyone!
Extra study guide I made here

- It's done by a special instruction: **RTI**
- However, this can only be used in Supervisor Mode (if in User Mode, it causes an *exception*)

- Let's look at an example of what would happen in the case of interrupts…



TRAP

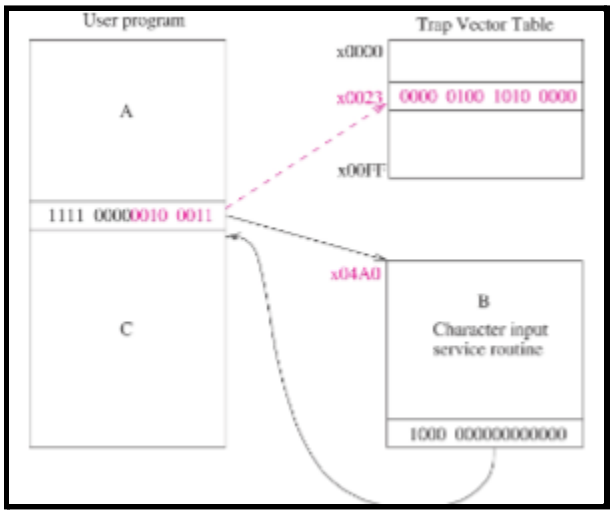


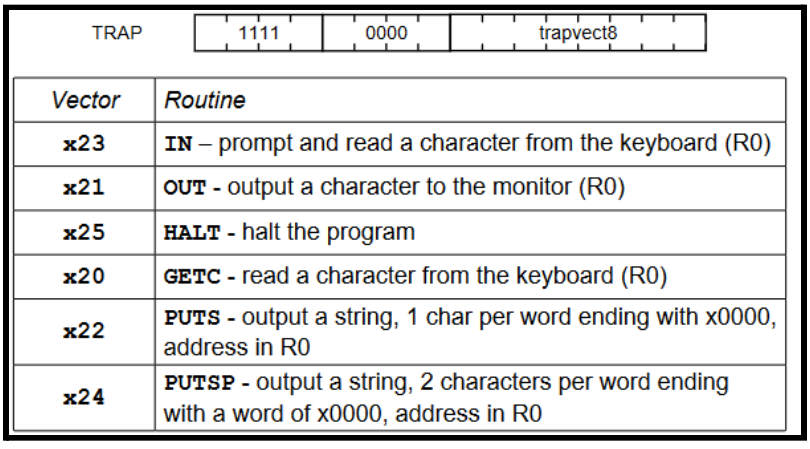
- At the very beginning of memory, from x0000 - x00FF, we store 256 pointers to different service subroutines

Good luck on the final everyone!
Extra study guide I made here

- *1. A set of service routines.*
  - They are part of operating system – service routines start at arbitrary addresses within the OS
    (convention is that system code lives between x0200 and x3000)
  - Supports up to 256 service routines
- *2. Using a table of starting addresses.*
  - Stored at x0000 through x00FF in memory
  - Called System Control Block in some architectures
  - Initialized by the operating system
- *3. TRAP instruction.*
  - Used by program to transfer control to an operating system routine
  - 8-bit trap vector names one of the 256 service routines
  - Saves PSR and PC on via the R6 stack and gains privilege just like Interrupts

- The **trap vector table** is stored at the beginning of memory and stores the addresses to different subroutines utilized by the operating system
  - When these subroutines finish, they use the RTI instruction (remember: this is an operating system subroutine, so we are in superuser mode)
- To the right is an example of what occurs when the trap vector table is used. Notice how we have to first utilize the address of the subroutine in the trap vector table, and then through that address we can go to the subroutine.



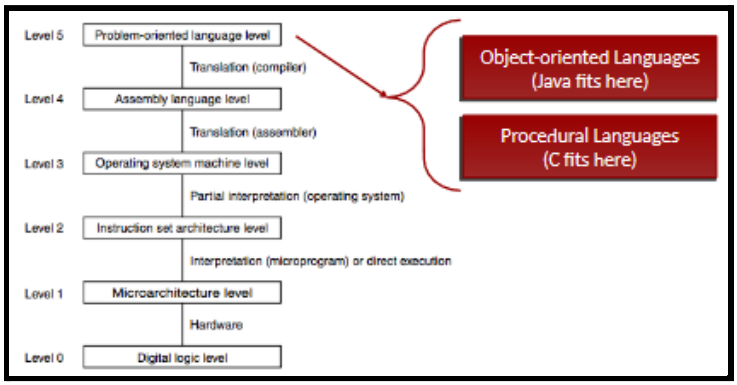


Vector	*Routine*
x23	**IN** – prompt and read a character from the keyboard (R0)
x21	**OUT** - output a character to the monitor (R0)
x25	**HALT** - halt the program
x20	**GETC** - read a character from the keyboard (R0)
x22	**PUTS** - output a string, 1 char per word ending with x0000, address in R0
x24	**PUTSP** - output a string, 2 characters per word ending with a word of x0000, address in R0

Good luck on the final everyone!
Extra study guide I made here

L14 - Intro to C

- Let's look back at the level chart to see where we are now…



- While it may seem like C is very close to Java and Python, don't get confused: *C is more close to assembly than it is Java or Python*
    - For example, Java and Python will throw ArrayIndexOutOfBounds exceptions. C however--much like assembly--will not.
- The best way to think about C: *C is a more readable syntax for assembly*.
- So if C seems so archaic, why do we still use it? Well, because we still use assembly! Since C is, essentially, a better version of assembly, no one writes assembly anymore but rather people write C.
    - Windows, Mac, and Linux were all written in C

Data Types

- C has numerous data types…
    - *Integer Types*
        - [unsigned] char — 8 bits
        - [unsigned] short[int] — 16 bits
        - [unsigned] int — 16/32 bits
        - [unsigned] long [int] — 32/64 bits
        - [unsigned] long long [int] — 64 bits
    - *Floating Point Types*
        - float — 32 bits
        - double — 64 bits
        - long double — 80/128 bits
    - *Aggregate Types*
        - array
        - Struct
        - union
    - Pointers (a special kind of integer)
- You'll notice that the sizes of data types are variable, that's because it depends…

Good luck on the final everyone!
Extra study guide I made here

  - char - exactly 8 bits
  - short int - *at least* 16 bits
  - int - *at least* 16 bits
  - long int - *at least* 32 bits
- To determine the size of the data type, we use the **sizeof** operator
  - *sizeof* is a *compile-time* constant reflecting the number of ***bytes*** held by the data type *or* instance (ex. sizeof(char) = 1)
  - sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)

Booleans

- Historically, C did not have a Boolean data type
- To simplify things, instead of creating an entirely new data type, we can just borrow the integer data type and map it to the following…
  - *False* - any integer that is 0
  - *True* - any non-zero integer
- This concept extends to the concept of chars (which, as we remember, are essentially just numbers)
  - *False* - the NUL character '\0'
  - *True* - any other character
- It even applies to pointers…
  - *False* - the memory address 0x0
  - *True* - any other address

Strings

- Just like in assembly, Strings are just arrays of characters that end in the NUL character (0 or '\0')
  - Ex. in the array char mystr[6], we can hold 6 characters, but only set 5 since we need to reserve 1 for the NUL character
- Strings in C can be initialized in several ways…
  - *Explicit Size*: char mystr[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  - *Non-Explicit Size*: char mystr[] = {'H', 'e', 'l', 'l', 'o', '\0'};
  - *Non-Array:* char mystr[] = "Hello";
    - NOTE: The latter two are **special cases**, in which the C compiler determines the length of the array from the initializer
- Since there are many common functions called for strings, there are a number of library functions for dealing with strings…
  - strlen() - returns the length of the string
  - strcpy() - copies a string from one array to another
  - strdup() - "Returns a pointer to a null-terminated byte string, which is a duplicate of the string pointed to by **`str1`**"

Good luck on the final everyone!
Extra study guide I made here

- Remember: C is simply just a fancier assembly, so much of the functions we do in C have similar instructions in assembly
    - Ex. char s[6] = "hello" is equivalent to **`s .stringz "hello"`** where *s* is the memory address where the string starts
- **Never** use sizeof(s) when you want the string length; this returns the size of the array (or the size of the pointer), not the actual length of the string. Use strlen(s) instead.
    - Keep in mind that we have to include the function in our header…
      `#include <string.h>`

printf() Function

- C doesn't come bundled with a print function, since a lot of the things that use C don't really need them
- However, there is an IO library we can use, specifically we can use the function printf()
- The first argument is a format string, beginning with % and reference arguments that come after the format string, in order
- Example…

%d	Decimal integer (int)
%x	Hex integer (int)
%f	Floating number (float)
%s	String (char * or char [])
%c	Character (char)
%p	Pointer (for debugging)

```
printf("Person: %s  GPA: %f\n", name, gpa);
```

might print

```
Person: Dan  GPA: 2.5
```

The C Preprocessor

- The C preprocessor does two main things…
    - File inclusion: #include
    - Macro expansion: #define
- Let's look at each of them individually…
    - #include
        - Conventionally, we only include files that end in ".h"
        - These are declarations (including function prototypes) and macro definitions but no executable code
        - There are two ways to use #include…
            - *Double Quotes* ("") - the preprocessor looks in the current directory and then the system directories for the file
            - *Angle Brackets*(< >) - looks only in the system directories (e.g. "/usr/include")

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mylibrary.h"
```

Good luck on the final everyone!
Extra study guide I made here

- NOTE: These lines don't end in a semicolon because they aren't part of the C language. Rather, they are preprocessor directives

- #define
  - Macro processing is just text substitution with very specific rules
  - We're going to need to want to use symbolic names for constants in several places
  - Examples…

```
#define NUL_CHAR '\0'
#define MAXWORDLEN 256
```

  - These symbolic names are *textually* replaced in the source code before it is compiled.

Continuing with our last example, we can write

```
char buf[MAXWORDLEN];
…
while (buf[i] != NUL_CHAR && i < MAXWORDLEN)
   i++;
```

After the C Preprocessor, the code literally becomes

```
char buf[256];
…
while (buf[i] != '\0' && i < 256)
   i++;
```

  - You can also define macros with arguments, but there are some stipulations that we need to account for
  - Suppose we define the following macro expression…

```
#define PRODUCT(a,b) a*b
```

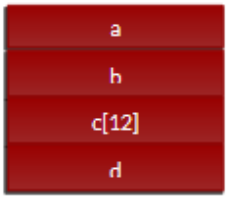
  - If we call `PRODUCT(x+3, y+4)`, this would become x+3*y+4, which is not what we want
  - To account for this, put parentheses around everything in our macro expression…

```
#define PRODUCT(a,b) ((a)*(b))
```

Structs

- Structs are a lot like classes in java, they aggregate data types, contain **no** methods, and all data members are publicly visible

```
struct a {
   int a, b;
   char c[12];
   int d;
};
```



24 bytes – *assuming sizeof(int) is 4 bytes*

Good luck on the final everyone!
Extra study guide I made here

- The struct tag *declares* the type. Let's say we have a struct defined like so…

```
struct car {
   char mfg[30];
   char model[30];
   int year;
};
```

- If we want to create instances of the struct, we need to do the following…

```
struct car mikes_car, joes_car;
```

- You can also declare a struct type and immediately define an instances (or even multiple instances) of the variable

```
struct car {
   char mfg[30];
   char model[30];
   int year;
} mikes_car;

struct car is a data type

mikes_car is a variable of type struct car
```

- Just like in Java, we use the . operator to reference members of a structure, but to place things in them we have to do something funky; we will see why this is the case in a bit

```
printf("%s\n", mikes_car.model);
strcpy(johns_car.mfg, "Chevrolet");
```

Pointers

- Can do the same things you can do with addresses in assembly; powerful and dangerous (no runtime checking, for efficiency)
- Pointers contain memory addresses or NULL (no address)
- In LC-3 Assembly, were we to write…

```
B     .fill     29
BADDR     .fill     B
```

  then in C we would write…

```
int b = 29;
int *baddr = &b;
```

- What's with the asterisk (*) and the ampersand (&)? Well, in C this is how we deal with pointers…
  - *Ampersand* (&) - Means "address of." Indicates that we are getting the address of the variable as opposed to the value stored at that variable
  - *Asterisk* (*) - Means "pointer to." Indicates that a variable is a pointer.

Good luck on the final everyone!
Extra study guide I made here

- NOTE: Both of these change the data types to the special Pointer data type. This means that if you tried to do something like…

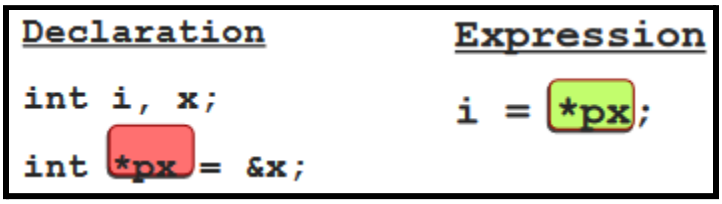
```
int px = &a
```

The right-side of the declaration is of type pointer, but the left side is of type int, which is a data-type mismatch and the compiler will throw an error.

- Suppose we have a pointer variable, and we want to get the value of the spot in memory where the pointer points to. We do this by using the *dereference operator*, which is also an asterisk (*)

```
*baddr = *baddr + 2;
```

- So yes: asterisk has two meanings, it can be used in two different ways…



- In short, if the * is used during declaration, it is being used for data type declaration. If it's any other time, it's being used for dereferencing a pointer.
- NOTE: You can have multiple asterisks to indicate a pointers to other pointers (e.g. int **y is a pointer to a pointer to an int)

Pointers & Arrays

- Suppose we have the following C code…

```
int a[10];

int *p;
```

- Here, *a* is an array of size 10 of type int and *p* is a pointer to int (memory address)
- But a memory address is effectively a pointer, so in assembly when we create an array A, it's sort of like a pointer
- However, we have to be careful about this…

```
// legal
p = a;   // p is the memory address of
         // the start of the array

// illegal
a = p;   // a is a CONSTANT pointer
         // you are not allowed to change its value
```

- Array pointers in C are what we call **constant pointers**, in that you are not allowed to change what they point to. Other pointers are exempt from this, which is why it is legal to change what *p* points to (an normal pointer) but not *a* (a constant pointer)
  - CALLBACK: Remember when I said that you had to do something funky to the string "Chevrolet" into the char array in the struct? This is why. In fact, you are

Good luck on the final everyone!
Extra study guide I made here

not allowed to do *array_name* = *anything* in C, as it will always result in a compiler error

- Since an array of characters is basically a string, one simple way to identify or make a string easily is if you see/write "char *"

Pointer Arithmetic

- Think back to how we handled things in assembly. If we wanted a particular index of an array, we would first get the address of the array (which points to the first index of the array) and then increment that address by an integer to get the address of the index we want
- C allows you to do this same thing: you can add or subtract pointers and integers
  - pointer + int = pointer
  - pointer - int = pointer
- It works conveniently this way with one caveat: the magnitude of the integer is dependent on the datatype size of the pointer. For example…

```
int *p = &i;

p = p + 1;
  is interpreted as
p = p + 1 * sizeof(*p);
```

  - In other words, if *p* is an *int* pointer, then *p + 1* is the address of the **next** *int* - the address of *p* is incremented by the size of an *int*
  - It's important to remember that we are not exclusively working with data that takes only one spot in memory, which is 8 bits or 1 byte. In modern machines, for instance, an integer is 32 bits, or 4 bytes. Bringing this 4 byte requirement back to our LC-3 which has 1 byte memory, that means that if we want to put two integers into memory, we'd need 8 spaces. If our first integer was at x5000, then our next integer would need to be at x5004, because x5000, x5001, x5002, and x5003 are all reserved for our first integer.
  - Conveniently, we actually don't have to write out the "* sizeof(*p)" expression because C does it for us

```
int b[3] = { 9, 12, 13 };
int *p = &b[0];

The value of *p is 9, *(p + 1) is 12,
and *(p + 2) is 13
```

- Since arrays and pointers are sort of the same, it would make sense that we can use things like array notation on pointers. Suppose we have the following…

Good luck on the final everyone!
Extra study guide I made here

```
int a[10];
int *p;
p = a;
```

Then the following are equivalent…

a[5] ⟷ *(a+5)

p[5] ⟷ *(p+5)

&a[5] ⟷ a+5

&p[5] ⟷ p+5

Pointers & Structs

- In the section on structs, it was briefly mentioned that you could declare a variable of type struct immediately after creating your struct…

```
struct car {
   char mfg[30];
   char model[30];
   int year;
} mikes_car;
```

`struct car` is a data type

`mikes_car` is a variable of type `struct car`

- With this idea in mind, we can also create a pointer to a struct immediately after creating a struct…

```
struct myStruct {
        int a, b;
} *p; // p is a pointer to struct myStruct
```

- In fact, this is so common in C that there's actually an operator for this: **->**
- Using this operator, the top line and bottom line of code mean the same thing…

```
(*p).a = (*p).b;
p->a = p->b;
```

  - You might be wondering why this would be so common in C. Well, because it's so common in most other languages you've probably written in. Consider Java: when you pass an object into a method, you're not passing the actual object, you're passing a *reference* to that object (i.e. passing in the memory address to that object), which is what **p* is here

Good luck on the final everyone!
Extra study guide I made here

Functions in C

- The way C handles methods is very similar to how assembly handles methods
- When you call a method, C will push the arguments in reverse order to the stack frame, and when it returns, it takes out the return value and breaks down the rest of the stack

```
int mult(int a, int b) {
        return a*b;
}
```

- When thinking about building up the stack, we aren't actually passing the original numbers onto the stack; we are pushing *copies* of the values onto the stack. This is what C's default is: **pass by value**
- *However*, C does have the capacity to **pass by reference**, and to do so we simply push pointers onto the stack instead of copies of the values. Even though those pointers are still just copie, they still point to the same thing
- Functions: main()
  - Every C program has exactly one function called *main*, which returns an int
    - 0 on success
    - Non-0 on error
  - It is the first function that's invoked when running your program from the command line
  - By convention, is always the first function in the program

```
int main(int argc, char *argv[]) {
        return 0;
}
```

- Functions: void
  - Denotes a function that has no return value, we are all too familiar with this
  - *However*, to denote a function with no arguments, we **also** use void
  - A function with an empty arguments section (no void) is interpreted by C as meaning it takes an unknown amount of arguments

```
void func(void) {
        printf("Hello\n");
}
```

- Functions: Arrays as Arguments
  - To pass an array into a function, we pass a pointer to the first element
  - This includes strings, which are just arrays of chars. As strange as it is, instead of passing in the entire string we simply pass in the address to the first character in the char array

```
char s[10] = "Hello";

void test(char *s) {
        printf("%s\n", s);
}

int main(int argc, char *argv[]) {
        test(s);
        test(&s[0]);
}
```

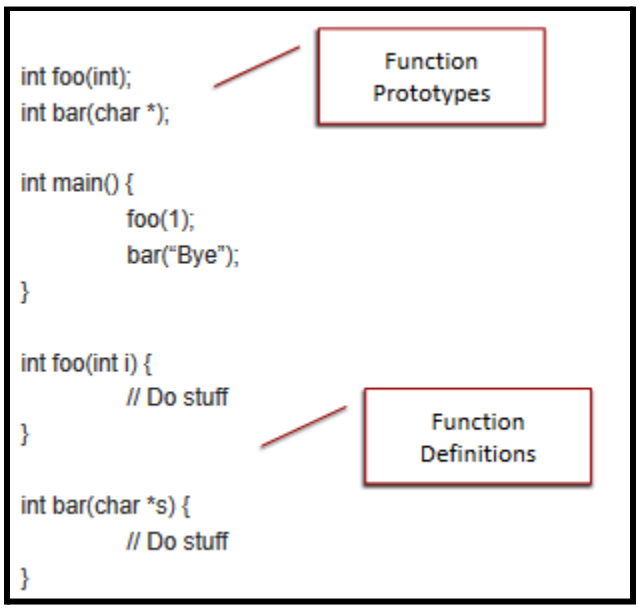
Same meaning

- Functions: Declare *before* call (One-pass Compiler)
  - Remember how in assembly it was possible for us to reference labels that had not yet been declared at our current line of code? This was because our LC-3

Good luck on the final everyone!
Extra study guide I made here

compiler is a *two-pass* compiler, which on the first pass generated a symbol table and on the second pass checked for uses of those symbols from the symbol table.

- C, as a design choice, does not have this. In other words, C is a *one-pass compiler*, meaning you cannot call a function before it is declared. Alright, so then we just make *main()* our last function and put everything else above it, right? Well, this goes against the convention, so no
- To alleviate this, we need to use something called **function prototypes**…



- *Function prototypes* are a way to circumvent this issue by allowing us to declare our functions above the *main()* function so we can call them validly
- A function prototype always looks the same…

```
<return type> funcName(args);
```

where we start it with the return type, the name of the function, the arguments (if any), ending with a semicolon

Command Line Arguments

```
int main(int argc, char *argv[]) {

    return 0;

}
```

- *argc* is the number of arguments (argument count)
- *argv* is an array of pointers to chars i.e. an array of strings (argument values)
- For example…

```
$ ./myprogram cs2110 rocks
```

argc == 3

argv[0] is "./myprogram"

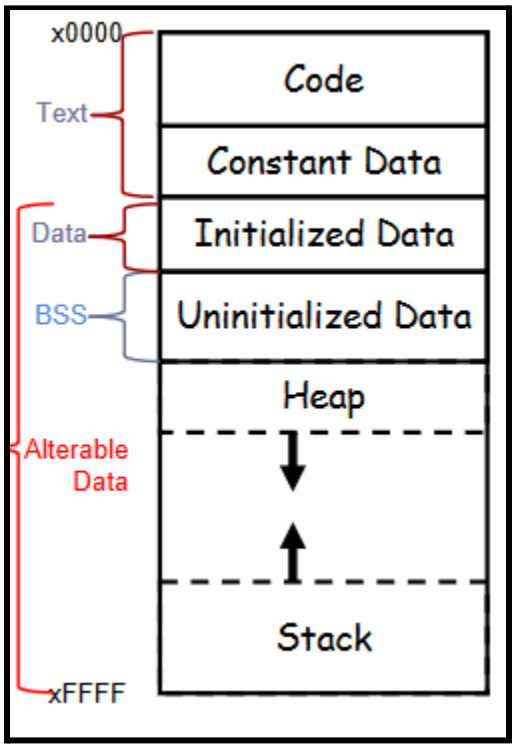
argv[1] is "cs2110"

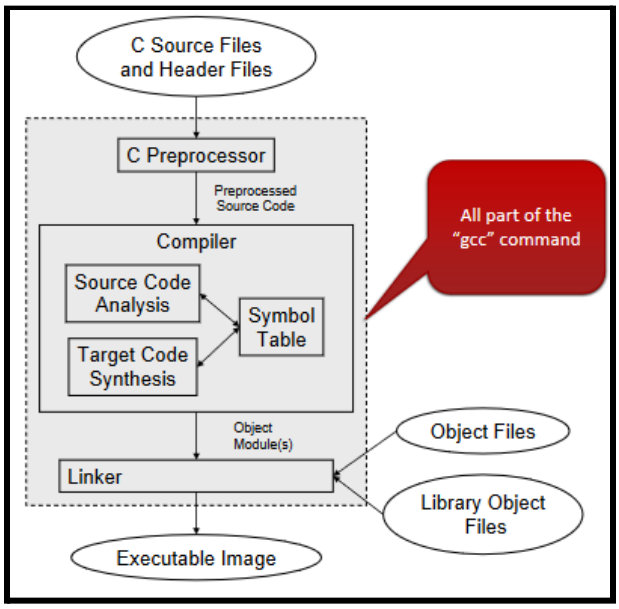
argv[2] is "rocks"

Good luck on the final everyone!
Extra study guide I made here

L15 - Continuing with C

- Thinking back to the memory layout we had on the LC-3, the memory layout in C is very similar…





- C has many different scopes..
  - *File Scope* - Seen by the entire file; declared outside a function definition

<no keyword>	**External definition** – visible to other files – defines storage
static	Visible to no other files
extern	**External reference** – visible to other files – declares reference to an external definition elsewhere

    - For example, let's say we declared *int a[10]* in a C file, and I wanted to use that exact array in another C file. To do this, I would have to use **extern**…

```
file1.c:
        int a[10];                  // external definition
        static struct r *p;  // this file only
        extern float c[100]; // ref to c in file2.c
        …

file2.c:
        extern int a[10];   // ref to arr in file1.c
        static struct r *p; // NOT the p above
        float c[100];       // definition
        …
```

Good luck on the final everyone!
Extra study guide I made here

- NOTE: The linker at the end of the C compiler is what allows for us to do this; the linker looks through the files to see if there is an *int a[10]* it can use
- *Block Scope / Local Scope*: A variable that is only exists inside of a certain block e.g. *i* in a for loop

Storage Classes & Type Qualifiers

- **Storage classes** tells us *where* the data will be stored, **type qualifiers** tell us *who* will be able to see it

Storage Classes

	Outside a function definition	*Inside a function definition*
<none>	**scope:** external definition **storage:** static address	**scope:** within the function **storage:** on the stack
auto	N/A	**scope:** within the function **storage:** on the stack
static	**scope:** within the file only **storage:** static address	**scope:** within the function **storage:** static address
extern	**scope:** external reference **storage:** static address; location determined by file containing the external definition (can't have initializer either)	**scope:** external reference **storage:** static address; location determined by file containing the external definition (can't have initializer either)
register	N/A	**scope:** within the function **storage:** register or stack (hint to compiler; use of & not allowed; seldom used)

same

Storage Class Specifie

register	const
auto	volatile
static	restrict
extern	

Type Qualifiers

Type Qualifiers

- **const** - the value of this variable may not be changed after initialization
- **volatile** - the compiler may not optimize references to this variable (e.g. it's a device register that may change value asynchronously)
  - Ex. I want to run a for loop that counts up to a million and does nothing just to waste time. An optimizer would get rid of the loop because its redundant, but making the counter variable *volatile* tells the compiler I don't want this to be optimized
- **restrict** - for the lifetime of a pointer, only the pointer itself or a value directly derived from it may be used to access the object to which it points; allows for better optimization

What to Know

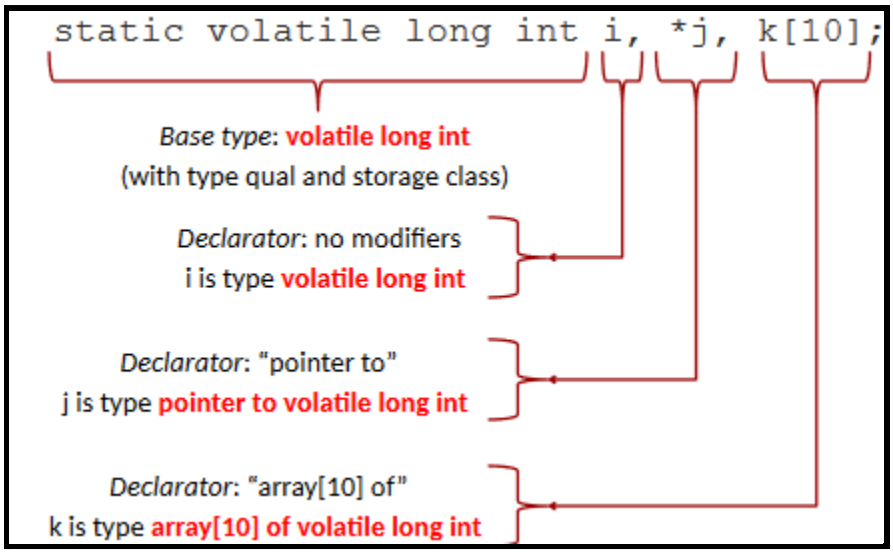
- The two meanings of static…
  - *Inside a function*: Changes the storage location to static memory, either data or BSS segment (scope stays local)

Good luck on the final everyone!
Extra study guide I made here

    - *Outside a function*: static changes the scope (visibility) to be only visible within the file (storage location stays in static memory)
- extern
    - Compiler will not allocate new storage
    - For type checking of the identifier name only
    - Another C file *must* allocate storage by defining that var name/function
- volatile
    - Tells compiler to not optimize variable away

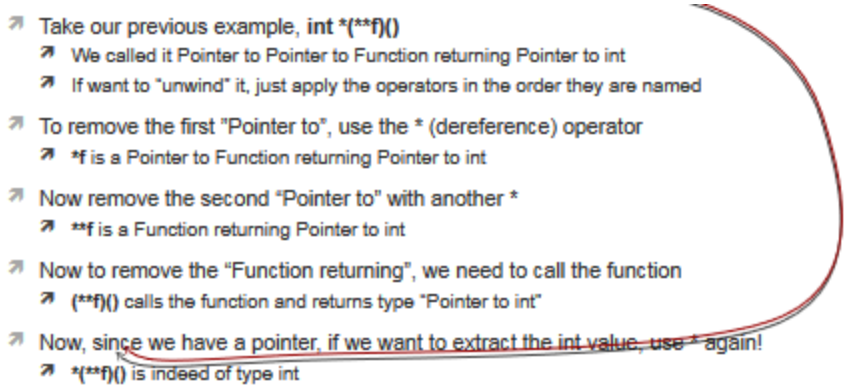
Declarations & Definitions

- A **declaration** in C introduces an identifier and describes its type, be it scalar, array, struct, or function. Have two parts…
    - *Base Type*: the data type of the declarators
    - *Declarators*: A list of the variable names, separated by commas



- A **definition** in C actually instantiates/implements this identifier
- To read declarations…
    - Rule 1: Remember the precedence of the declarators
        - () and [] get processed first
        - * gets processed last
        - Parenthesis change the precedence order (just as in expressions)
    - Rule 2: Read or form the declarations from the *inside out*
    - Ex. int *(**f)() is a pointer to a pointer to a function returning a pointer to an int
- Now that we know how to read declarations, we need to know how to "unwind" data types

Good luck on the final everyone!
Extra study guide I made here



- ye it's just the same thing

Typedef

- **typedef** is a shortcut that allows you to create an alias for a type; it does **not** create a new type

Define an array of 5 struct a named b:

```
struct a b[5];
```

Create a type alias for an array of 5 struct a

```
typedef struct a sa5[5];
```

Now we can use sa5 as a type name:

```
sa5 c;
```

Function Calls

- Functions must be declared before they are used (think back to *function prototypes*)
- Functions are **call by value** as opposed to call by reference, meaning copies of the value are pushed onto the stack frame as opposed to references/pointers to the variables on the stack frame
- Because of this, if we wanted to run a *swap* method on the variables, nothing would happen since we're working with copies

```
void swap(int a, int b)

{
int t;

t = a;

a = b;

b = t;

}
```

```
int x = 42;

int y = 84;

swap(x, y);
```

- To fix this, we need to instead pass in *pointers to* ints. Doing this though, we also need to change our method header, but after doing so we can do what we want

Good luck on the final everyone!
Extra study guide I made here

```
void swap(int *a, int *b)
{
int t;
t = *a;
*a = *b;
*b = t;
}
```

```
int x = 42;
int y = 84;
swap(&x, &y);
```

Arrays

- If we have a global or local array, and we want to compute the size…

```
sizeof(ary) / sizeof(ary[0])
```

- If it's passed as a parameter or in the heap, we cannot use the above method; we'll have to come up with something else
- In assembly, we only have single-dimensional arrays, but in C we can have multi-dimensional arrays, it just gets translated into assembly…

```
short array[1026];
```
would assemble as
```
array    .blkw   1026
```

- Since arrays are always allocated in single, contiguous memory blocks, here is how the following would look…

```
short matrix[10][10][10];
```
would assemble as
```
matrix  .blkw    1000
```

- Just use values in braces "{}"

```
int ib[5] = { 5, 4, 3, 2, 1 };
```

- The compiler will even count them for you

```
int ib[] = { 5, 4, 3, 2, 1, 0 };
```

- Characters initializers are similar

```
char cb[] = { 'x', 'y', 'z' };
```

- But you can arrange for special treatment

```
char cb[] = "hello";
```

- But note this is very different from

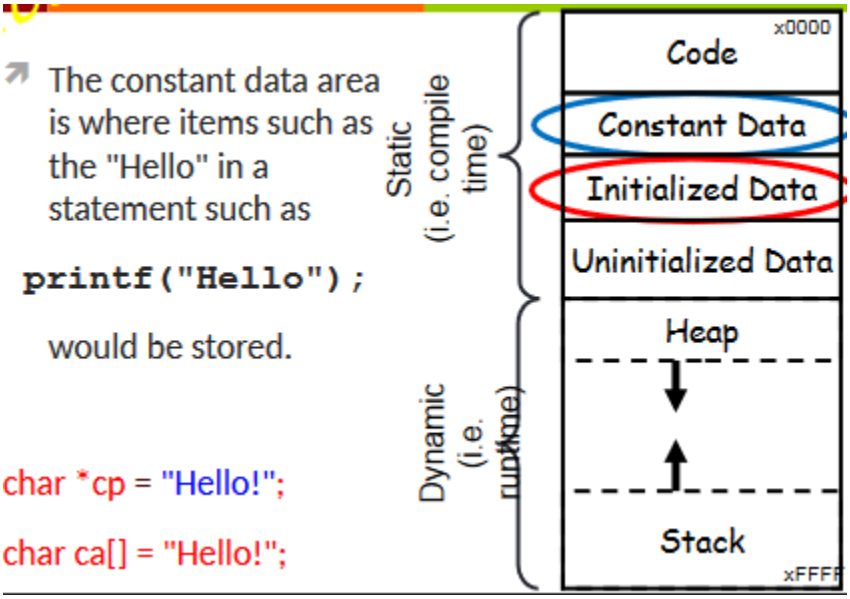
```
char *cb = "hello";
```

- Some important considerations when using *sizeof* on arrays and pointers…

```
char c1[] = "hello";
char *c2 = "hello";
```

Good luck on the final everyone!
Extra study guide I made here

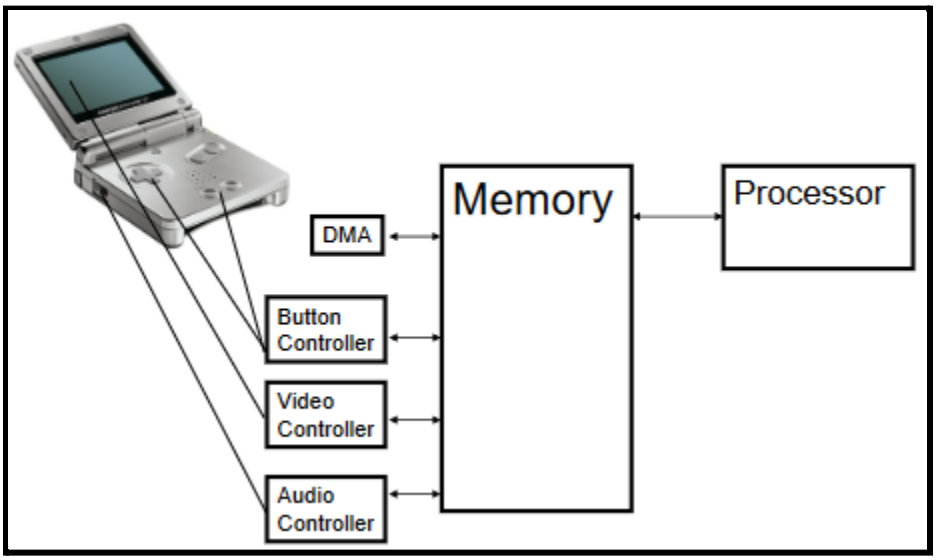
- *sizeof*(c1) = 6, because it's the size of the array including the null terminator
- *sizeof*(c2) = 8, because it's the size of the pointer on your system (this is a system-dependent answer, but most operating machines are 64-bit systems)
- *strlen*(c1) = 5, same as *strlen*(c2)



	Can pointer value be changed?	Can "Hello" be changed?
char *cp = "Hello";	Yes	No
char ca[] = "Hello";	No	Yes

Good luck on the final everyone!
Extra study guide I made here

L16 - Introduction to GBA



- GBA is big slow; runs at 16.78 MHz. For reference, most computers run at 3 GHz
- Datatypes
  - 8 bit (char)
  - 16 bit (short OR short int)
  - 32 bit (int)
  - 64 bits (long int OR long)
  - Has floats/doubles but should be avoided at all costs because they're slow
- 32 bit address space
- 10 buttons on the GBA we have at our disposal…
  - *Start, Select, A, B, Left, Right, Up, Down, Left shoulder, Right shoulder*
- One button register, 1 bit per button; opposite of what you'd expect…
  - 0 pressed
  - 1 not pressed

Pixels on a Screen

- Just like we have *device registers* on the LC-3, the GBA also has a few important registers that we will be utilizing, primarily REG_DISPCTL
- REG_DISPCTL stands for *Display Control Register* and, as you can guess, is in charge of controlling the display
- The first thing we have to do when trying to put pixels on a string is to set the *display mode*
- In order to set the display mode, we need to be able to write to a location in memory; REG_DISPCTL is at 0x0400 0000, which is an integer. However, the number that lives at this register is a short, a 16-bit number. How do we fix this?

Good luck on the final everyone!
Extra study guide I made here

- Remember: this number is supposed to be a memory address that points to a 16-bit number on the other end. Therefore, we need to cast this to a *pointer to short*…

  (short *) (0x0400 0000)

- Alright, so now we have a pointer to short, a memory address. Now, we need to be able to put something into memory with this. For our purposes, the video mode we want is equivalent to the decimal number 1027, which is an int. So we just do…

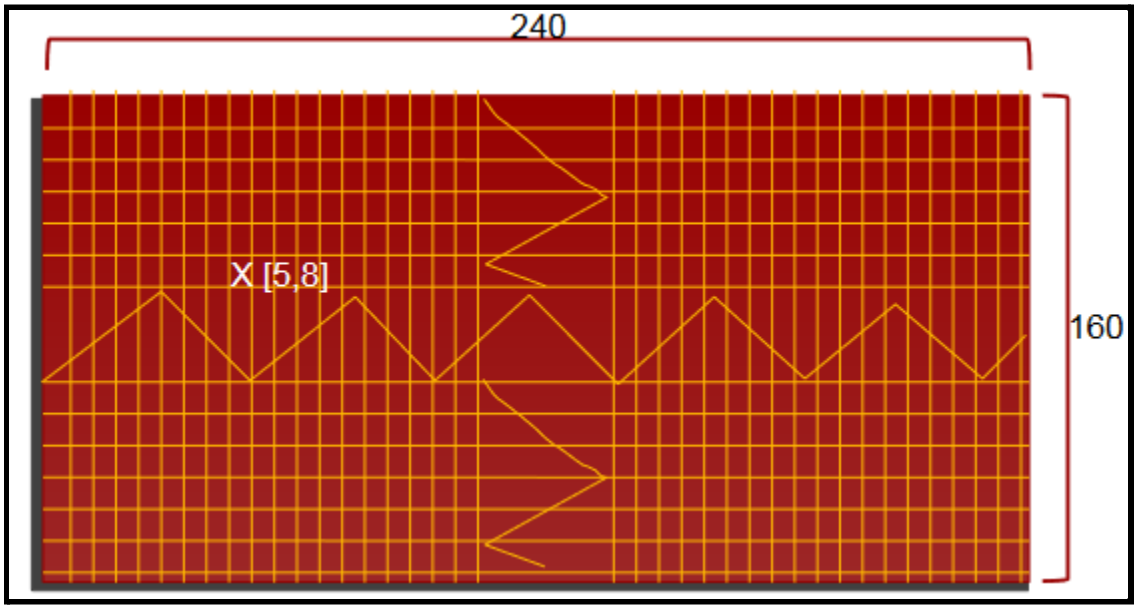
  (short *) (0x0400 0000) = 1027

- Right? ***Nah, :b:***. You're trying to say a pointer to short equals an integer, which is a compiler error because the types don't match. To fix this, all we need to do is *dereference* it (remember that a short is an integer data type, so this is legal)...

  *(short *) (0x0400 0000) = 1027

- To sum it up, we casted our hex number to a pointer to short, so it's now a memory address, and then we dereferenced that memory address so we can put a number at that memory address

- The screen layout for the GBA can most easily be described as a 240x160 2D array of colored pixels…



- Remember though that we don't really have 2-dimensions in memory, so each row consecutively follows the other row in memory. In other words, this 2D array is more accurately described (in memory) as a 1-D array of length 240 * 160
- Therefore, in order to place things into this array, we need to be able to find the coordinate we want to edit in memory. Since this array has a row major order, we can simply use the following equation to get what we want…

  `offset = (row) * (width) + (col)`

- Since width is always 240, this can simplify to…

Good luck on the final everyone!
Extra study guide I made here

offset = (row) * 240 + (col)

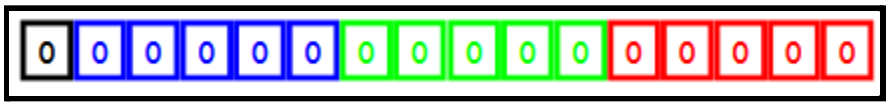
- Once we have our offset, we simply need to use some C magic to be able to place a pixel there…

```
*((unsigned short *)0x6000000 + offset)
```

- And let's not forget that we can treat pointers like arrays…

```
unsigned short *videoBuffer =
   (unsigned short *)0x0600000;
videoBuffer[5*240+8] = 0x7fff;
```

- For pixels on the GBA, we control color using 16-bits with RGB….



Good luck on the final everyone!
Extra study guide I made here

L17 - GBA Programming DMA

- The GBA screen is refreshed at 60 Hz, meaning the screen updates itself 60 times per second, however there are many different phases in this drawing process…
    - For each scanline (160 lines)...
        - Draw scanline (240 pixels)
        - Hblank (68 pixels)
    - Vblank (68 scanlines)

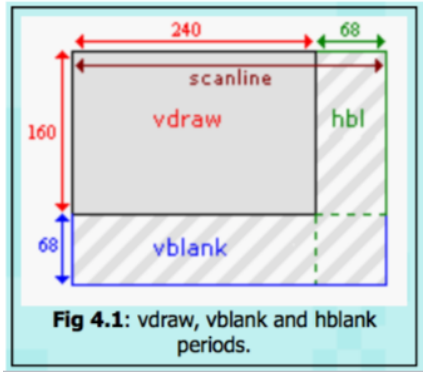


Fig 4.1: vdraw, vblank and hblank periods.

- To avoid tearing, positional data is usually updated while the scanline is at VBlank. This is why most games run at 60 or 30 fps.
- When we say at VBlank, we mean *at* VBlank, not *in* VBlank. If you start drawing when the scanline is at 160 + 30, you're not going to have enough time to draw everything you want. Therefore, we wait to draw new things until the scanline is *exactly* at 160

Buttons on GBA

- The GBA has 10 buttons you can click on…
    - A, B, Start, Select, Right, Left, Up, Down, Right bumper, Left bumper
- They are represented with a *bit vector* where bit 1 is A, bit 2 is B, etc.
- To get the current buttons, since they are in a device register, just like we've been doing with REG_DISPCTL, since it's a hardcoded memory address we can just read from it

```
#define BUTTONS *(volatile
                  unsigned int *)0x4000130

#define KEY_DOWN_NOW(key)
                  (~(BUTTONS) & key)
```

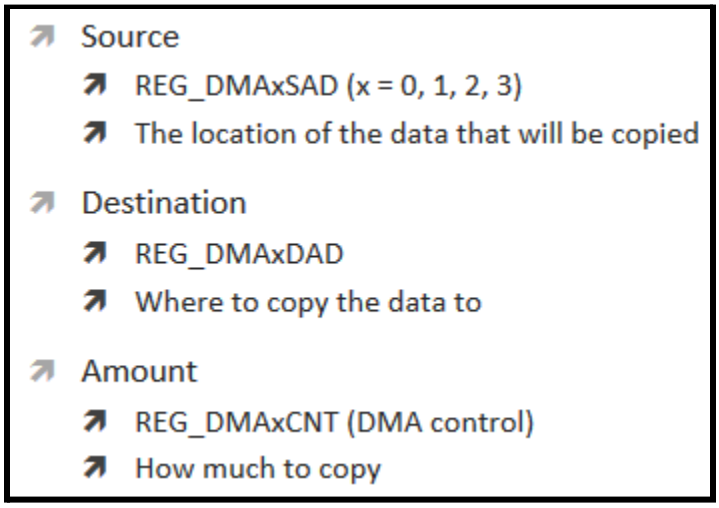
DMA

- As it turns out, even if we wait to start drawing until VBlank is 160, we still get tearing if the object we are trying to update is above a certain size. This is because trying to change things with arrays in a loop is generally pretty slow
- To alleviate this, the GBA uses something called **DMA** (**Direct Memory Access**)
- DMA is a separate CPU next to the ARM processor whose only job is to copy memory from one location to another *really* fast in hardware, as opposed to software like we've been doing with the loops so far

Good luck on the final everyone!
Extra study guide I made here

- In fact, DMA is 10x faster than doing it through the loops in software, and as such it removes the tearing on moving large objects on the screen

- DMA has 3 channels, but we will only be using *Channel 3*, which has the lowest priority and is used for general purpose copies, like loading tiles or bitmaps into memory



- Loop Template for your game

```
Initialize

While true
    Previous state = Current state

    Check buttons
    Calculate new values for Current state

    Wait for Vblank

    Undraw using Previous state
    Draw using Current state
```

```
Set Gamestate to INITPLAY

While true
    Previous state = Current state

    if Gamestate == PLAY
        Check buttons
        Calculate new values for Current state
    Else if Gamestate == INITPLAY
        Initialize the Current state

    Wait for Vblank

    If Gamestate == PLAY
        Undraw using Previous state
        Draw using Current state
    Else if Gamestate == INITPLAY
        Gamestate = PLAY
```

- Lastly, for our HW we've been given a font data that we can use to write words on screen as fontdata_6x8[]

Good luck on the final everyone!
Extra study guide I made here

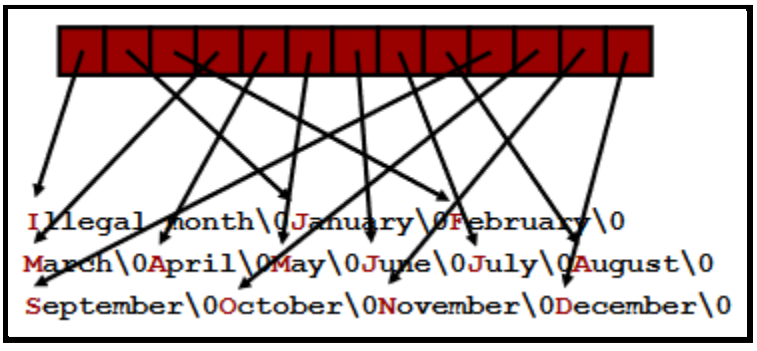
L18 - Continuing with C

Arrays of Pointers - Marginally Indexed Array

```
char *month_name(int n){

   static char *name[] = {
      "Illegal month",
      "January", "February", "March",
      "April", "May", "June",
      "July", "August", "September",
      "October", "November", "December"
   };

   return (n<1 || n>12)? name[0]: name[n];

}
```

- Some things to note about this function…
    - The return type is an array of pointers to char, and it uses a local variable that is also an array of pointers to chars. As we know, pointer to char is the same thing as String
    - Our local variable is static. This was a design choice; remember that things that are static in local functions are stored in the static section of memory, which is a persistent section. This means that instead of every time the function gets ran it creates an entirely new local variable array on the stack, it just puts one array on the stack
        - An array of pointers to char is an array of memory addresses that point to the first character in the string in memory which are null terminated

```
Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0
```

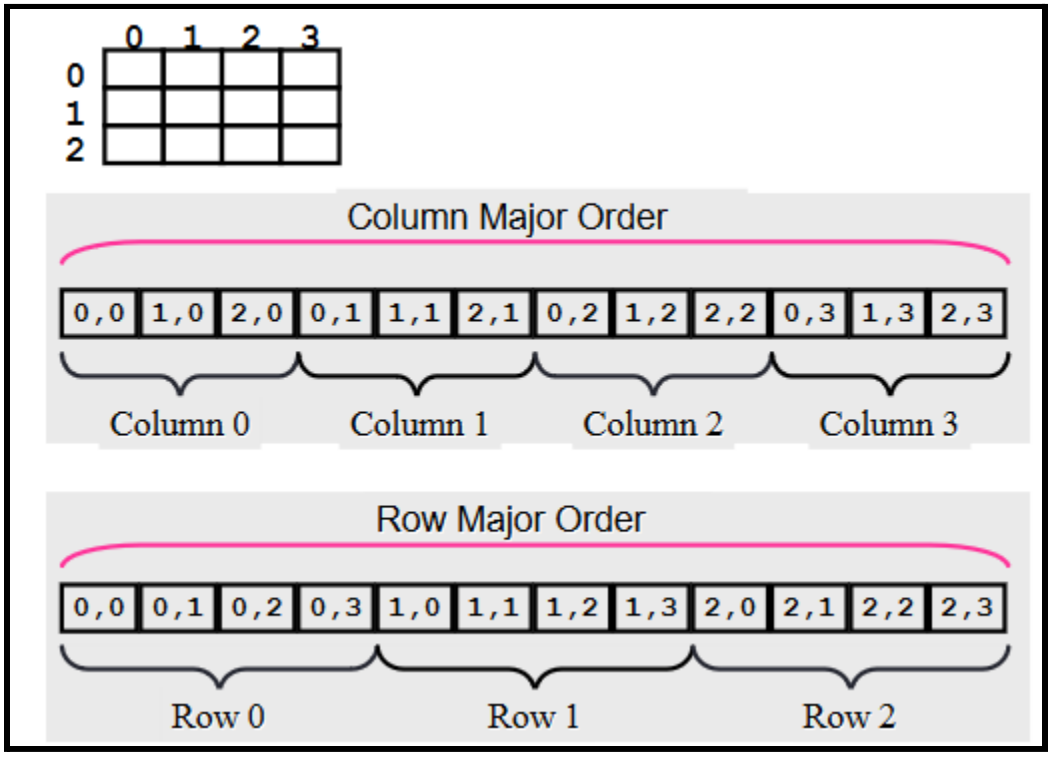


Good luck on the final everyone!
Extra study guide I made here

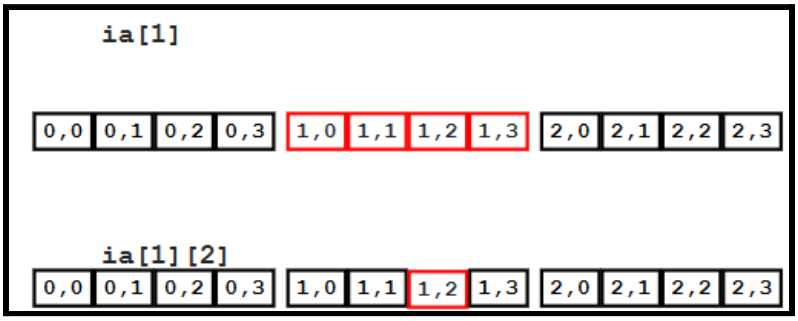
    - We call this an **marginally indexed array**, where each index points to a location in memory. Note that this is *not* a 2D array.
- Suppose we have a 2D array of characters of size 5x5, and we want to store 5 words in this. What if our 5 words are shorter than 5 letters? (Waste memory slots on empty spaces) Longer than 5 letters? It should be clear now why doing things with *marginally indexed arrays* is sometimes more beneficial

Multi-Dimensional Arrays

- As we know, array sizes are defined at compile time, and so when we create an array we need to know the number of rows and columns beforehand
- As we know from the GBA chapter, we can't really store 2D arrays in memory the way we think about them, we have to "slim them down" to 1D arrays
- There are two days we can slim them down: *Column Major Order* and *Row Major Order*



- The designers of C chose to use *Row Major Order*. This allows us to easily visualize what we are looking for…



Good luck on the final everyone!
Extra study guide I made here

- Suppose we have the following line…

```
int a[5][7]
```

- Assuming `sizeof(int) = 4`, what would it return if I were to ask for the size of…
  - `a[3][4] = 4` (asking for the size of an element in an array of array of ints, 4)
  - `a[3] = 28` (asking for the size of an array of 7 ints, so 7 * 4)
  - `a = 140` (asking for an array of 5 * 7 ints, so 5 * 7 * 4)

One Dimensional Array

```
int ia[6];
```

Address of beginning of array:

```
ia ≡ &ia[0]
```

Two Dimensional Array

```
int ia[3][6];
```

Address of beginning of array:

```
ia ≡ &ia[0][0]
```

also

Address of row 0:

```
ia[0] ≡ &ia[0][0]
```

Address of row 1:

```
ia[1] ≡ &ia[1][0]
```

Address of row 2:

```
ia[2] ≡ &ia[2][0]
```

- So, given a row and column index, how do we calculate the location?

```
row_index * sizeof(row) + column_index * sizeof(arr_type)
```

- Suppose we have the method with the following header…

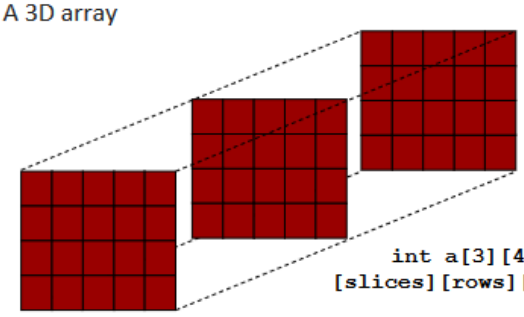
```
void tester(int arr[][4][5], int len)
```

- The above is actually legal because C doesn't have bounds checking, and if you look above at the calculating for finding a location in a 2D array, you'll notice that we don't actually need the number of rows in our calculation. Here's another formula to be more explicit…

```
offset = i * columns + j
```

- In general, we **never** need the first dimension when we declare an array

A 3D array



offset = (i * rows * columns) + (j * columns) + k
a[i][j][k] == *(*(*(a + i) + j) + k)

Good luck on the final everyone!
Extra study guide I made here

L19 - More Structs

- Recall that a *struct* is sort of like a class in Java but it only holds data and doesn't have methods
- The struct tag *is* optional, but it's bad practice to not name them
- Structs are in a **separately-scoped namespace** from variables, i.e. a struct variable can have the same name as a struct tag w/o causing confusion

```
struct [ <optional tag> ] [ {
   <type declaration>;
   <type declaration>;
   ...
} ] [ <optional variable list> ];
```

- Remember that when working with structs, you can't handle arrays the same way you have historically in other languages…

```
struct mystruct_tag {
      int myint;
      char mychar;
      char mystr[20];
} ms;
ms.mystr = "foo";
```

  - The above is illegal because you are trying to assign an array to an array
- You can also initialize a struct in the variable list…

```
struct mystruct_tag {
            int myint;
            char mychar;
            char mystr[20];
      };

struct mystruct_tag ms = {42,'b',"Boo!"};
```

- You can copy a struct to another struct too…

```
struct s {
   int i;
   char c;
} s1, s2;

[insert data to s1]
     s1 = s2;
[check data of s2]
```

- In the above snippet, the data in s2 will be the exact same data as the stuff from s1

Good luck on the final everyone!
Extra study guide I made here

Structs in Memory

- Suppose we have the following struct…

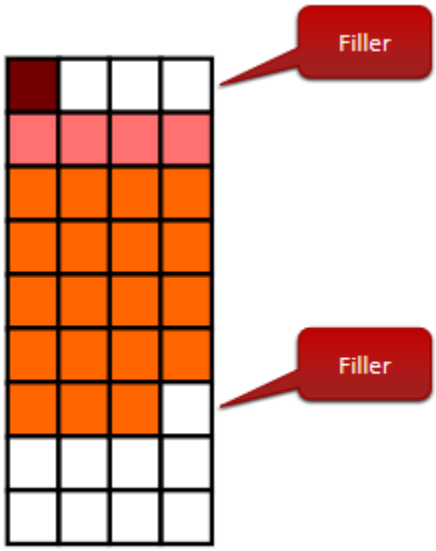
```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
```

- How many bytes of memory does this take up (remember: chars take 1 byte and ints take 4) => 1 + 4 + 19 = 24
- However, just because the sum of its parts take 24 bytes, that does not always necessarily mean it will be 24 bytes in memory. For instance, this struct will actually take **28** bytes

```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
```

Structs are usually filled out to meet the most stringent alignment of its members

sizeof(mystruct) = 28



- The compiler keeps track of the offsets of each member in a table for each struct…

member	offset
mychar	0
myint	4
mystr	8 (Sum of sizes of all previous elements incl filler)

- So if mychar lives at 1000, then myint lives at 1004, and mystr lives at 1008
- What if we want to get the address of a particular char in mystr?
- Well, we know the struct starts at 1000, it's offset by 8, and then lets say we want the 4th element, that's 1000 + 8 + 4 = 1012

Good luck on the final everyone!
Extra study guide I made here

```
static struct {
        int n;
        char m[3];
        double p;
} s[12];
```

If *s* is stored at memory address 0x0e3c, where is *s[5].m[2]* stored?

A. 0xe91

B. 0xe92

C. 0xe96

D. 0xea0

- Member offsets
  n: 0, m: 4, p: 8
- Struct size
  16
- Offset from s to &s[5]
  5*sizeof(s[0]) = 80 = 0x50
- Offset from &s[5] to s[5].m
  4 = 0x4
- Offset to from s[5].m to &s[5]m[2]
  2*sizeof(char) = 2 = 0x2
- Address
  0x0e3c+0x50+0x4+0x2 = 0xe92

Structures may

- be copied or assigned
- have their address taken with &
- have their members accessed
- be passed as arguments to functions
- be returned from functions

Structures may not

- be compared

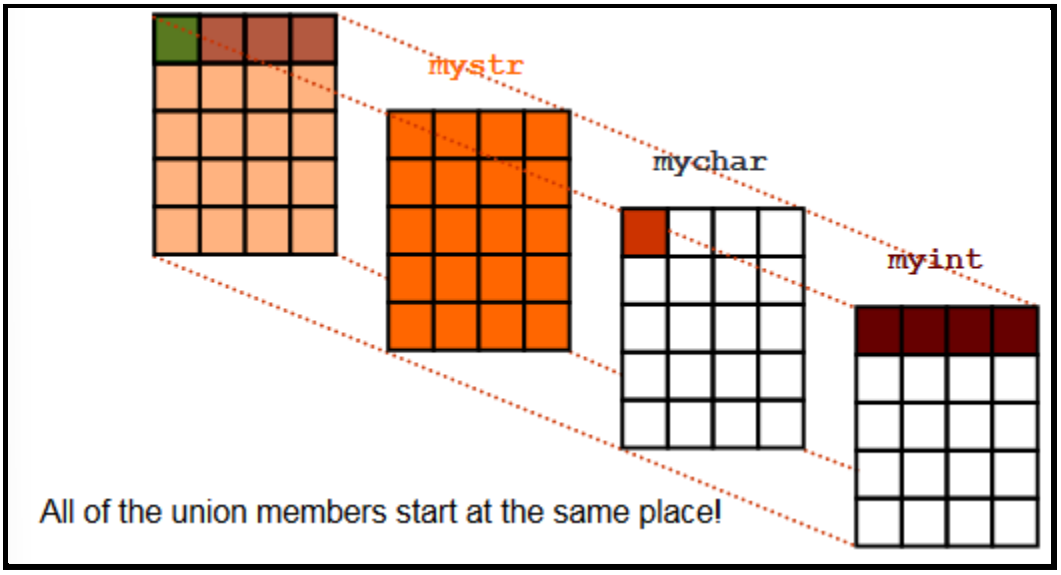
Good luck on the final everyone!
Extra study guide I made here

L20 - Continuing with C (cont.)

Unions

```
union {
  int myint;
  char mychar;
  char mystr[20];
} myun;
```

- **Unions** look like structs and accessing them is the same as astruct, however all members have an offset of zero
- That is, you can only store one member at a time, so you only need to have enough space in memory to accommodate the largest member

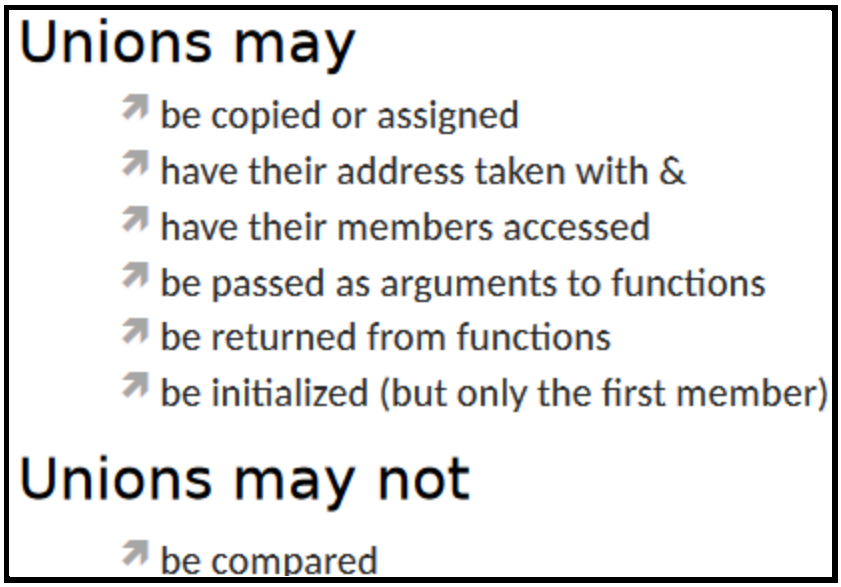


- So…
  - &myun.myint == &myun.mychar == &myun.mystr[0]
  - sizeof(myun) is the size of the largest member
- Why would we ever want something like this?
  - Suppose we have a database of athletes, and let's say they can only be a football player, baseball player, or basketball player. We could create a struct called "player" and define it as so =>

```
struct player {
   char name[20];
   char jerseynum[4];
   char team[20];
   int player_type;
   union sport {
   struct football {...} footbstats;
   struct baseball {...} basebstats;
   struct basketball {...} baskbstats;
   } thesport;
} theplayer;
```

- Unions are also useful for implementing **polymorphism** found in object-oriented languages

Good luck on the final everyone!
Extra study guide I made here



Function Pointers

- **Function pointers** are just like any other pointer, in that they store the address to something, but in this case they only hold the address to a *function*

```
int fi(void);        /* Function that returns an int  */
int *fpi(void);      /* Function that returns a pointer
                      * to an int                      */
int (*pfi)(void);    /* pfi is a pointer to a function
                      * returning int!  */
```

- Since we can treat these function pointers like variables, we can actually pass them into other functions as variables; this is incredibly powerful for trying to do things like *sorting*
- In fact, the standard C library provides for us a quicksorting algorithm for arrays…

```
qsort(3)

NAME
       qsort - sort an array

ANSI_SYNOPSIS
       #include <stdlib.h>
       void qsort(void * base, size_t nmemb, size_t size,
           int (* compar)(const void *, const void *) );
```

- Note that when we actually use a function pointer, we don't call it like a function; we simply pass the name of the function…

```
qsort(a, MAX, sizeof(int),
                        compar_ints);
```

Good luck on the final everyone!
Extra study guide I made here

L21 - Dynamic Allocation

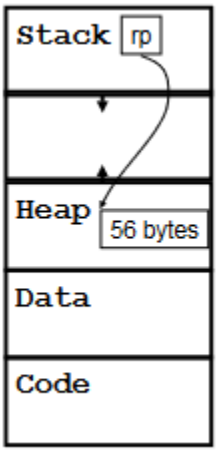
- Since we have a static number of memory, it'd be convenient for programs that require a large amount of memory to be able to free up memory whilst the inner-workings of the program are running
- C gives us 2 libraries to help with this…
    - malloc()
    - free()

Malloc

- Malloc is a function that takes an integer for the number of bytes it needs to allocate on the heap, and returns an address on the stack that points to the starting address of those bytes on the heap

```
struct r {
   char name[40];
   double gpa;
   struct r *next;
};
struct r *rp;
rp = malloc(sizeof(struct r));
if (rp == NULL){
   /* Handle Error! */
}
```

Options for handling error
- Abort
- Ask again
- Save user data
- Ask for less
- Free up something



- However, since it's entirely possible for memory to not have enough space, it is possible for it to return null. Because of this, you have to check if it is null…

```
rp = malloc(sizeof struct r);

// Code foolishly inserted here!

if(rp == NULL)
```

```
if((rp = malloc(sizeof struct r)) == NULL){
   /* Handle Error Here */
}
```

```
if( !(rp = malloc(sizeof struct r)) ){
   /* Handle Error Here */
}
```

Good luck on the final everyone!
Extra study guide I made here

- malloc() returns a point to at least bytes as we requested, but since the method header for malloc is…

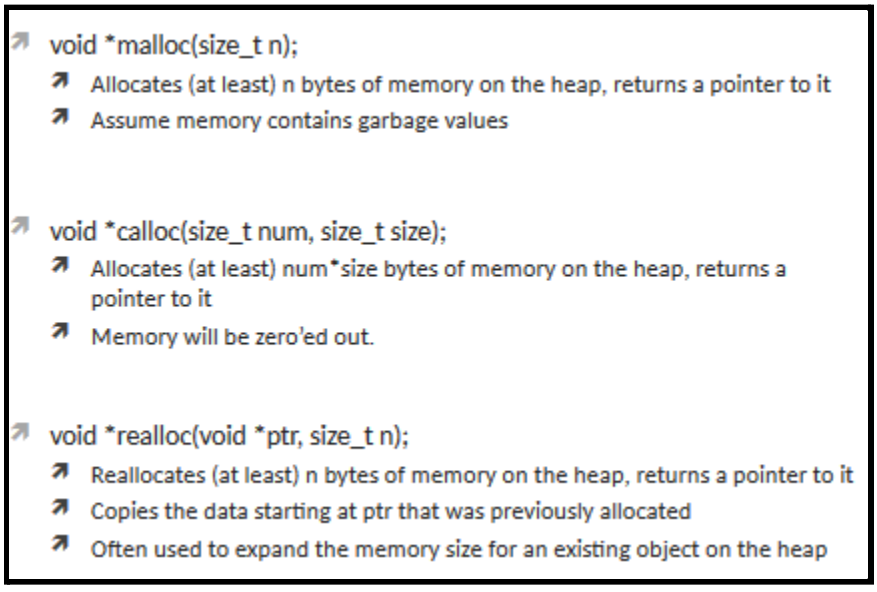
  `void *malloc(unsigned long)`

  it returns what C calls a "pointer to void," a generic pointer
- Because of this, you should cast the pointer into the correct type so that type-checking will work for you again, otherwise the compiler will silently cast the pointer for you without checking

Free

- Once you are done with a chunk of storage, you can use free() to make it available for reuse (C ***doesn't*** do this for you; remember: no garbage collection)
- From our previous example, free(rp) would return the memory back to the heap for re-use
- Note that once you use free() on a pointer, *you must **not** use the value in rp nor dereference the memory it points to*
- This is because once you've freed it, there's no guarantee that *rp will be what you want
- So, once we call rp…
  - The variable rp still exists
  - Points to garbage data
  - However, you *can* assign a new value to rp

Other Functions



Handling Persistent Data

- Suppose we have a function that returns a local variable. So far, without malloc, we have two ways this can be done…

Good luck on the final everyone!
Extra study guide I made here

```
char *foo(void)
{
   char ca[10];
   return ca;
}
```

- This is immediately problematic because once the function returns, the pointer is pointing to something on the stack, which as we know from the stack teardown, this is now pointing to garbage
- What if we make it static?

```
char *foo(void)
{
   static char ca[10];
   return ca;
}
```

- This is better, but it doesn't handle the case where we want to have multiple char arrays stored in memory. To solve this, we can use malloc…

```
char *foo(void)
{
   char *ca = malloc(...);
   /* error checking but no free */
   return ca;
}
```

- A **memory leak** occurs when the programmer loses track of memory allocated by malloc or other functions that call malloc. The below function causes a memory leak…

```
void foo(void)
{
char *ca = malloc(...);
/* no free */
return;
}
```

- As such, we need to understand which functions call malloc and be able to handle the way they manage memory

Good luck on the final everyone!
Extra study guide I made here

- Calloc()

```
void *calloc(size_t num, size_t size);
```

  - Finds space for *num* new allocations of size *size* and initialize the space to zero (0)
- Realloc()

```
ptr = realloc(ptr, num_bytes);
```

  - Finds space for new allocation by copying original data into new space, freeing the old space, then returning a pointer to the new space
  - Because of the way realloc() works, it may return…
    - the same pointer
    - a different pointer
    - NULL
  - Since realloc has the potential to return null, you should **never** set the same pointer and use the same pointer in a realloc call

```
cp = realloc(cp, n);
```

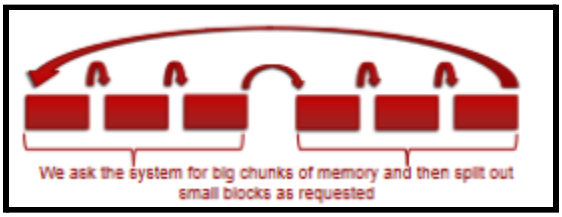
    - If realloc returns null, our data on the heap is still there, but now we can no longer get to it. This is a *memory leak*
  - To circumvent this, we need to use a temporary pointer…

```
void *tmp;

if((tmp = realloc(cp,...)) == NULL)
{
   /* realloc error */
}
```

  - Some edge cases regarding realloc…
    - realloc(NULL, n) = malloc(n)
    - realloc(cp, 0) = free(cp) [only works on some compilers]

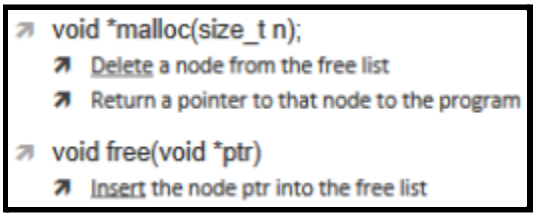
Malloc Implementation

- The K&R implementation uses a *Circularly Linked List* to track free memory, are it uses something called a **free list**, which contains memory available *to be malloc'ed*
  - As such, once memory is malloc'ed, we don't track it anymore until it's free'd

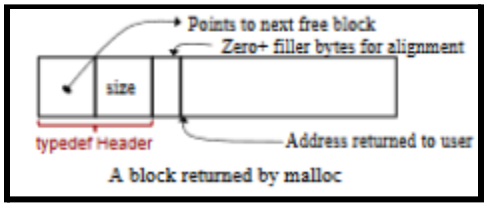


Good luck on the final everyone!
Extra study guide I made here

- Based on their implementation, the following functions affect the linked list as such…



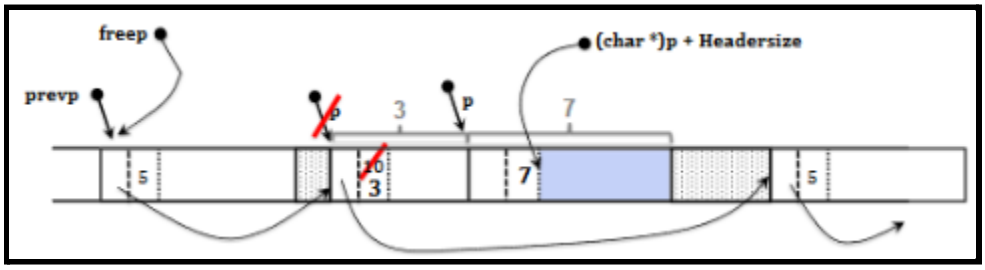
- So, each node in the free list is available memory and it looks like the following…



- Some of the design choices K&R took…
  - Linked List is ordered by memory address (in the heap)
  - We could also order the linked list from largest to smallest size
  - Size in K&R malloc is the number of *units*, **not** the number of bytes. A block is sizeof(Header) rounded up to the strictest alignment boundary. In this implementation, a block is **16** bytes
- The size of the header is accounted for in the size variable. For instance, if the header indicates our block has a size of 3 units, we really have 2 units + the header, which is 1 unit

Slicing Blocks

- If we don't have the perfect size we need in the free list, we can still make space by slicing an already existing block to a block size that can accomodate our malloc call
- Suppose we have a block of size 10 in the free list and we are mallocing something of size 7. When iterating through the free list to look for space, it looks for a block size of equal or greater size
- If it finds a block of greater size, it will *slice* the existing block into two blocks…



- We can also do this because remember: the header is accounted for in the size. Even though we really have 8 units available in this block of 10 units, the math still works out
- Once we split the block, we set *p* to point to the beginning of the first block. But we want to return to the user a pointer to the first memory address of available space, not the memory address of the header

Good luck on the final everyone!
Extra study guide I made here

- Because of this, we need to return **`(Header *)((char *)p - Headersize)`**
    - NOTE: We cast p to a pointer to char so we can ensure that the pointer arithmetic done on *p* is done through bytes, and not offset by the sizeof(p)

Good luck on the final everyone!
Extra study guide I made here

L22 - Miscellaneous Topics

Opening Files in C

- In the C environment, there are three files opened for you on pre-defined "streams," typically connected to your keyboard/display…
  - stdin - Standard input
  - stdout - Standard output - printf(...) defaults to stdout
  - stderr - Standard error output - use fprintf(stderr,...)
- Any other file must be opened and closed by calling a Standard IO routine…

```
FILE *infile;
if( (infile = fopen("f.txt","r")) == NULL){
      // Handle error
}

FILE *infile;
if( !(infile = fopen("f.txt","r"))){
      // Handle error
}
```

```
size_t fread(void *ptr,       // Read size*nmemb bytes
             size_t size,     // from stream
             size_t nmemb,
             FILE *stream );

size_t fwrite(const void *ptr, // Write size*nmemb bytes
              size_t size,     // to stream
              size_t nmemb,
              FILE *stream);
```

- C uses something called **buffered output**; when we call printf, we are really adding a string to a buffer that will eventually output to the screen
- However, if our code happens to segfault, it's entirely possible our buffered output won't come out
- Why? Well, in the LC-3 the instruction to print something out on the screen is a TRAP instruction, which is very expensive. Ideally, we'd want to run TRAP instructions and little as possible
- If we want to, we can circumvent this by using *fflush*…

```
printf("Checkpoint 1");
fflush(stdout);
```

- For error output, it is **not** buffered; it uses a special function called *fprintf*…

```
fprintf(stderr, "Checkpoint 1");
```

Inline Functions

- Think back to calling functions in LC-3 assembly. First we have to build up the stack, do the function, then tear it down. This is pretty wasteful
- C allows us to define a function as *inline*, which does essentially the same thing a macro does; it tells the compiler to do a sort of copy and paste of the function

Good luck on the final everyone!
Extra study guide I made here

```
inline int myfunc(int a, int b) {

    return a+b;

}
```

Good luck on the final everyone!
Extra study guide I made here

L23 - Signals

- Typically in a computer, multiple processes are happening at the same time
- Sort of, the computer works on one process at a time, halts it, moves to another process, then halts that and so on
- The way it's able to manipulate these processes is with **signals**, which tells a processes that some event occurs
- When the process receives the signal, it can do one of three things…
    - Ignore it
    - Perform the default operation
    - Catch the signal (perform a user defined operation)
- Every process has a *process ID*, and the signals also have