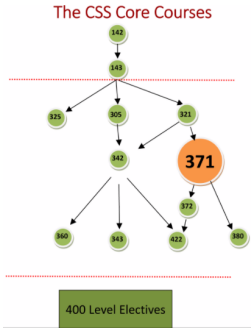
Machine Organization

-Textbook

https://na01.alma.exlibrisgroup.com/view/action/uresolver.do?operation=resolveService&package_service_id=8412825576000145 2&institutionId=1452&customerId=1450

Upcoming Courses



Overview

- A computer is very simple components interconnected
- Ch2, Information stored and processed by a computer is represented internally as strings of 1's and zero's
- Ch3, low and high voltages in a computer represent 1's and zero's
- Ch3, transistors, logic gates, combinational and sequential logic circuits
- Ch4, von Neumann architecture, fetch/execute instruction cycle
- Ch5 + Ch6, programming LC-3 with its own machine code language
- Ch7, programming LC-3 with assembly language
- Ch8, data structures (stacks, queues, strings)
- Ch9, LC-3 Input/Output
- *Ch10, an example - simulation of a calculator*

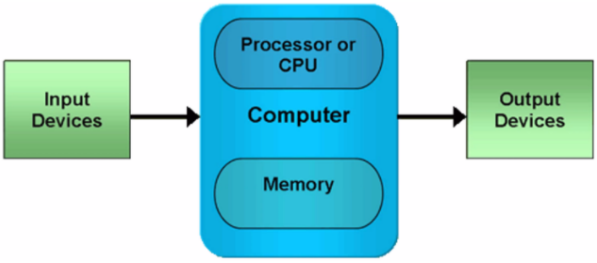
Chapter 1

-What the Computer?

-A device to perform tasks? Amogus?

-This is too abstract cuh, we need to deconstruct it!

-A computer has components



-Memory

-1st, Fastest, Smallest, CPU registers

-2nd, CPU Cache

-RAM

-ROM

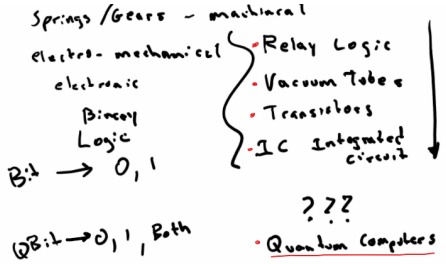
-Textbook def: A systematically interconnected collection of bery simple parts.

-We don't get the function of it though.

- We can further deconstruct the abstraction by examining the circuits inside each subcomponent. For example, the ALU is constructed from digital logic circuits such as **multiplexors** and **adders**.
- Digital logic circuits are constructed from smaller subcircuit components called **logic gates**.
- Logic gates are constructed from **transistors**. A modern CPU is an integrated circuit containing millions of transistors.

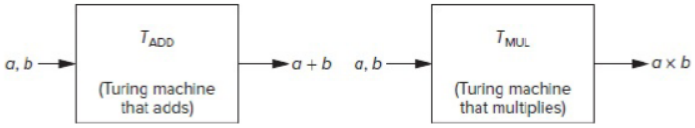
- A computer (and components such as the CPU) can be seen as electronic devices consisting of transistors, which are combined into logic gates, which are combined into digital logic circuits.
- The actual work done in a computer consists of the movement of electrons through an interconnected collection of very simple parts (transistors and circuits).
- We think of a computer as a device for performing high-level tasks. So how are such high-level tasks reduced to a flow of electrons?

-Brief history of computer types

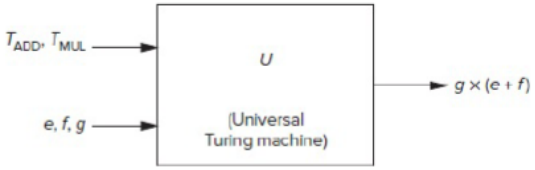


-Universal Computational Device

-Alan Turing defined a Turing machine, which takes inputs and computes them.



-A Universal Turing is one that takes in other Turing machines (algorithms be like) to process compute dif inputs.



Transformations Between Layers of Abstraction

-Problem

-Described in the natural language, what is the problem you want the computer to solve.

-Natural language has lots implied, so we need to design algorithms

-Algorithms

-A step-by-step procedure to solve a problem

-Language/Program

-Java, C, Python, HTML, various assembly languages, etc.

-Instruction Set Architecture (ISA) aka Machine Code, binary

-Translates/Compiles the program to run on the device.

-x86, ARM, etc.

-Microarchitecture

-The high-level hardware/physical implementation of the ISA

-So for x86, different Intel chips have different specs, but can all run x86 instructions.

-Circuits

-Designing logical circuits volta flow cheese.

-Devices

-Different types of circuits and parts of the circuits, way too dank EE major.

Chapter 2: Bits and Data Types

-Bit

-On or off. 1 or 0.

-Computers use voltage for a bit. High or low, pos or neg, presence or absence.

-A wire can represent 1 bit

-Put multiple bits together, boom bit string binary 321 cheese

- Millions of very tiny and fast devices (transistors) control the movement of electrons through the circuits of a computer. Devices react to the presence or absence of voltages in electronic circuits.
- Presence denoted by 1 and Absence denoted by 0.
- 1 wire can represent 1 bit (binary digit) that can be 1 or 0.
- 8 wires represent 8 bits of information that can have 2^8 → 256 different possible values (0 to 255).
- K wires or K bits can have ..... 2^K values.

Chapter 2.2: Integer Data Types

-Unsigned Integers

-No sign, can't differentiate between +/-

-This means with k bits, we can represent 2^k unique integers.

-Ex: $(5)_{10} = (101)_2$

- Weighted positional notation
  - like decimal numbers: "329"
  - "3" is worth 300, because of its position, while "9" is only worth 9

-Signed Integers

-Can represent +/-

-To do so requires us to use a bit to store if it is +/-.

-This means with k bits, we can represent 2^{k-1} magnitudes of integers.

-Ex (5-bit Signed Magnitude Int): $(5)_{10} = (00101)_2$ & $(-5)_{10} = (10101)_2$

-There are multiple ways you can represent +/-, but it's the most significant bit that is a signed bit.

-Fat 4 Representations of Integer Chart

Representation	Value Represented			
	Unsigned	Signed Magnitude	1's Complement	2's Complement
00000	0	0	0	0
00001	1	1	1	1
00010	2	2	2	2
00011	3	3	3	3
00100	4	4	4	4
00101	5	5	5	5
00110	6	6	6	6
00111	7	7	7	7
01000	8	8	8	8
01001	9	9	9	9
01010	10	10	10	10
01011	11	11	11	11
01100	12	12	12	12
01101	13	13	13	13
01110	14	14	14	14
01111	15	15	15	15
10000	16	–0	–15	–16
10001	17	–1	–14	–15
10010	18	–2	–13	–14
10011	19	–3	–12	–13
10100	20	–4	–11	–12
10101	21	–5	–10	–11
10110	22	–6	–9	–10
10111	23	–7	–8	–9
11000	24	–8	–7	–8
11001	25	–9	–6	–7
11010	26	–10	–5	–6
11011	27	–11	–4	–5
11100	28	–12	–3	–4
11101	29	–13	–2	–3
11110	30	–14	–1	–2
11111	31	–15	–0	–1

-1's Complement is letting a negative number be represented by taking the representation of the positive number having the same magnitude, and "flipping" all the Bits. That is, if the original representation had a 0, replace it with a 1; if it originally had a 1, replace it with a 0.

-Ex: +5 is represented as 00101, then –5 as 11010.

-However, operating on ints represented as anything but 2's Complement is cancerous.

Chapter 2.3: 2's Complement Integers

-2's Complement & Arithmetic and Logic Unit (ALU)

-Computers have some sort of ALU, taking in 2 inputs to operate on them.

-Ex Addition (just like decimal but 0 & 1):

```
 00110
 00101
 01011
```

-However, ALU doesn't care about what the input represents, it just computes them.

-2's Complement Integers Representation

REPRESENTATION(value + 1) =
REPRESENTATION(value) + REPRESENTATION(1).

-Came from wanting the ALU to return 0 when adding integer A + (-A) = 0.

-So, the negative representation is determined by the representation of 0 minus the representation of the positive equivalent magnitude.

-Ex: $(5)_{10} = (00101)_2$ & $(-5)_{10} = (11011)_2$ because

$$\begin{array}{r} 00101 \\ +\ 11011 \\ \hline 00000 \end{array} \quad = \quad \begin{array}{r} 00101 \\ -\ 00101 \\ \hline 00000 \end{array}$$

-Doesn't have -0

-Range: -2^{n-1} to $2^{n-1} - 1$

-Shortcut to find -A: Find complement (flip bits) of A, and add 1.

Note: If we know the representation for *A*, a shortcut for figuring out the representation for $-A (A \neq 0)$ is as follows: Flip all the bits of *A* (the official term for "flip" is *complement*), and add 1 to the complement of *A*. The sum of A and the complement of A is 11111. If we then add 00001 to 11111, the final result is 00000. Thus, the representation for *–A* can be easily obtained by adding 1 to the complement of *A*.

```
  011010000
  100101111  (1's comp)
+         1
  100110000
```

-Another shortcut

- To take the two's complement of a number:
  - copy bits from right to left including the first "1"
  - flip remaining bits to the left

```
011010000
(flip)  (copy)
100110000
```

Chapter 2.4: Convert Between Binary and Decimal

-Binary to Decimal calculator be like

Recall that an eight-bit 2's complement integer takes the form

$$b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$$

where each of the bits b_i is either 0 or 1.

1. Examine the leading bit b_7. If it is a 0, the integer is positive, and we can begin evaluating its magnitude. If it is a 1, the integer is negative. In that case, we need to first obtain the 2's complement representation of the positive number having the same magnitude. We do this by flipping all the bits and adding 1.
2. The magnitude is simply
$$\left(b_6 \cdot 2^6\right) + \left(b_5 \cdot 2^5\right) + \left(b_4 \cdot 2^4\right) + \left(b_3 \cdot 2^3\right) + \left(b_2 \cdot 2^2\right) + \left(b_1 \cdot 2^1\right) + \left(b_0 \cdot 2^0\right)$$
In either case, we obtain the decimal magnitude by simply adding the powers of 2 that have coefficients of 1.
3. Finally, if the original number is negative, we affix a minus sign in front. Done!

-Decimal to Binary

Let's summarize the process. If we are given a decimal integer value N, we construct the 2's complement representation as follows:

1. We first obtain the binary representation of the magnitude of N by forming the equation
$$N = b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$
and repeating the following, until the left side of the equation is 0:
   a. If N is odd, the rightmost bit is 1. If N is even, the rightmost bit is 0.
   b. Subtract 1 or 0 (according to whether N is odd or even) from N, remove the least significant term from the right side, and divide both sides of the equation by 2.

   Each iteration produces the value of one coefficient b_i.
2. If the original decimal number is positive, append a leading 0 sign bit, and you are done.
3. If the original decimal number is negative, append a leading 0 and then form the negative of this 2's complement representation, and then you are done.

1. If the leading bit is one, take two's complement to get a positive number.
2. Add powers of 2 that have "1" in the corresponding bit positions.
3. If the original number was negative, add a minus sign.

$$X = 01101000_{two}$$
$$= 2^6+2^5+2^3 = 64+32+8$$
$$= 104_{ten}$$

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

-Simply, find the max bit you are able to subtract, repeat.

-Decimal to Binary

Decimal to binary The decimal to binary case requires a little more work. Suppose we wanted to convert 0.421 to binary. As we did for integer conversion, we first form the equation

$$0.421 = b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + b_{-4} \times 2^{-4} + ...$$

In the case of converting a decimal integer value to binary, we divided by 2 and assigned a 1 or 0 to the coefficient of 2^0 depending on whether the number on the left of the equal sign is odd or even. Here (i.e., in the case of converting a decimal fraction to binary), we multiply both sides of the equation by 2 and assign a 1 or a 0 to the coefficient of 2^0 depending on whether the left side of the equation is greater than or equal to 1 or whether the left side is less than 1. Do you see why?

Since
$$0.842 = b_{-1} \times 2^0 + b_{-2} \times 2^{-1} + b_{-3} \times 2^{-2} + b_{-4} \times 2^{-3} + ...$$
we assign $b_{-1} = 0$. Continuing,
$$1.684 = b_{-2} \times 2^0 + b_{-3} \times 2^{-1} + b_{-4} \times 2^{-2} + ...$$
so we assign $b_{-2} = 1$ and subtract 1 from both sides of the equation, yielding
$$0.684 = b_{-3} \times 2^{-1} + b_{-4} \times 2^{-2} + ...$$
Multiplying by 2, we get
$$1.368 = b_{-3} \times 2^0 + b_{-4} \times 2^{-1} + ...$$
so we assign $b_{-3} = 1$ and subtract 1 from both sides of the equation, yielding
$$0.368 = b_{-4} \times 2^0 + ...$$
which assigns 0 to b_{-4}. We can continue this process indefinitely, until we are simply too tired to go on, or until the left side = 0, in which case all bits to the right of where we stop are 0s. In our case, stopping with four bits, we have converted 0.421 decimal to 0.0110 in binary.

Chapter 2.5: Operations on Bits - Arithmetic

-Always and Easiest on 2's Complement

-Subtraction and Addition example

-When Subtracting, find 2's complement and add

What is 14 – 9?

```
The decimal value 14 is represented as   01110
The decimal value 9 is represented as    01001

First we form the negative, that is, -9: 10111

Adding 14 to -9, we get                  01110
                                         10111

which results in the value 5.            00101
```

Using our five-bit notation, what is 11 + 3?

```
The decimal value 11 is represented as 01011
The decimal value 3 is represented as  00011
The sum, which is the value 14, is     01110
```

-Addition an int by itself is the same as shifting all bits to the left (if u have one to spare)

-Sign Extension

-If you are doing arithmetic with a negative number represented as by a lower number of bits compared to the other, extend it out with 1s.

-aka, expand by copying the most significant bit.

-Ex: 13 + (-5) = 8 (where -5 is in 6-bit)

-Extending with 0s, wrong

```
 0000000000001101
+       111011
------------------
0000000000111011, that is, +59
```
bruh

-Extending with 1s.

```
  0000000000001101
+ 1111111111111011
------------------
  0000000000001000
```
yay!

-Overflow

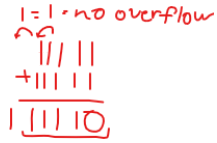
-The operation returns a value too big positively or negatively to represent that it loops around.

-Overflows when

-The signs of both operands are the same, and the sign of sum is different.

-If the carry into the MS bit does not equal the carryout, then we have an overflow.

-Ex: -1 + -1



Chapter 2.6: Operations on Bits - Logical Operations (Actually just 321)

-A Logical Variable (Boolean?)

-1-bit to represent true & false.

-AND, OR, NOT, XOR

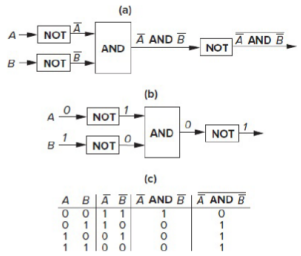
-Refer to 321

- $\overline{A}$ is how the book write not (scuffed complement)

```
 01 1011 0110
 11 0001 1101
--------------
 11 1011 1111   bitwise OR ∨
 01 0001 0100   bitwise AND ∧
 10 1010 1011   bitwise XOR ⊕
```

-DeMorgan's Laws

-Ugly complement cheese



A	B	$\overline{A}$	$\overline{B}$	$\overline{A}$ AND $\overline{B}$	$\overline{A \text{ AND } B}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

$$\overline{\overline{A} \text{ AND } \overline{B}} = A \text{ OR } B$$

-Bit Vector

An *m*-bit pattern where each bit has a logical value (0 or 1) independent of the other bits is called a *bit vector*. It is a convenient mechanism for identifying a property such that some of the bits identify the presence of the property and other bits identify the absence of the property.

-321 and Sets: You represent a set with a bit array. The “1” bit at the n index means the element corresponding to that index is in the set.

-Bit Mask

-Like a photo editing mask, make a bit string with 1s at the places you actually care about.

-Perform AND with the other bit string and only the places with 1s are allowed to be 1/on/let through.

Chapter 2.7: Other Representations

-Floating Point Numbers

-Kinda like estimating by scientific notation

The *floating point* data type solves the problem. Instead of using all the bits to represent the precision of a value, the floating point data type allocates some of the bits to the range of values (i.e., how big or how small) that can be expressed. The rest of the bits (except for the sign bit) are used for precision.

Most ISAs today specify more than one floating point data type. One of them, usually called *float*, consists of 32 bits, allocated as follows:

```
1 bit for the sign (positive or negative)
8 bits for the range (the exponent field)
23 bits for precision (the fraction field)
```

(aka IEEE 754, 32-Bit, Single Precision)

-The Fraction Field / Mantissa requires binary fraction representation (Binary to Decimal Fractional Parts)

-Just like whole numbers, but it’s 2^neg stuff

Number Base as Power	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
Equivalent	16	8	4	2	1	.	1/2	1/4	1/8

2^{-4} 1/16

Binary to decimal The binary to decimal case is straightforward. In a positional notation system, the number

$0.b_{-1}b_{-2}b_{-3}b_{-4}$

shows four bits to the right of the binary point, representing (when the corresponding b_i = 1) the values 0.5, 0.25, 0.125, and 0.0625. To complete the conversion to decimal, we simply add those values where the corresponding b_i = 1. For example, if the fractional part of the binary representation is

.1 0 1 1

we would add 0.5 plus 0.125 plus 0.0625, or 0.6875.

-Ex: $(0.25)_{10} = (0.01)_2$

-Converting from Decimal to Binary Fraction

-Multiply by 2, number in whole (either 1 or 0) is the binary fraction.

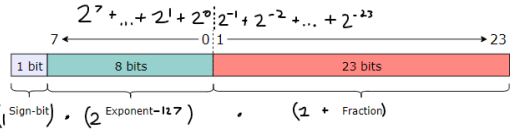
-Repeat multiply by 2 w/ decimal part only.

-Stop when reaching (0.0 * 2).

Decimal Number Multiplication	Result	Number in front of decimal
0.125 x 2	0.25	0
0.25 x 2	0.5	0
0.5 x 2	1.0	1
0.0 x 2	0.0	0

0.125 =.001

-IEEE 32bit formula



$$(-1)^s \times (1 + \textbf{Fraction}) \times 2^{exp-127}$$

-Ex: 1 10000001 01110000000000000000000

$$(-1^1) \cdot (1 + (2^{-2} + 2^{-3} + 2^{-4})) \cdot \left(2^{(2^7+2^0)-127}\right) = -5.75$$

-Normalized Form

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form. This basically puts the radix point after the first non-zero digit. In normalized form, 50 is represented as 5.000×10^1.

-Converting Decimal to Floating Point w/ ex

-Separate whole and decimal parts

85.125 → 85 & 0.125

-Convert the whole part to binary

85 → 1010101

-Convert decimal part to binary fraction

0.125 → 0.001

-Combine “whole.fraction” parts

1010101.001

-If decimal isn’t one from the front, move it up

1010101.001 = 1.010101001 * 2^6 (We moved it 6 spaces up. Notice it’s not 10^6 because binary)

-The decimal part of it is the mantissa (fill up empty bits with 0s)

1.010101001 → **01010100100000000000000**

-Find exponent from how decimal places moved (exp = decMove + 127)

1.010101001 * 2^6 means 127 + 6 = 133

-Turn exponent to binary

133 → **10000101**

-Determine sign bit

+85.125 → **0**

-Done ez: sign, exponent, mantissa

85.125 → **0 10000101 01010100100000000000000**

-Java

-With IEEE, cases like 0.1 + 0.2 = 0.3 can't be represented with float/double

-Therefore, BigDecimal op

-Hexadecimal

-Base 16

-One hex digit represents 4 bits, so it's a shorthand for binary

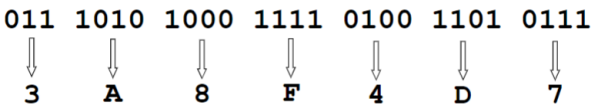
Binary	Hex	Decimal			
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

-Notation

0x… $(\dots)_{16}$

-Converting between Binary & Hexadecimal

-Each 4 bit is one hex digit



Convert the following unsigned binary numbers to hexadecimal.

a) 001 1111 = 0x1F

b) 1 = 0001 = 0x1

-2's Complement, hexadecimal eats it up because it is unsigned.

Convert the following unsigned binary numbers to hexadecimal.

a) 001 1111 = 0x1F

b) 1 = 1111 = 0xF

-ASCII

-[ASCII Table - ASCII Character Codes, HTML, Octal, Hex, Decimal]

00 nul	10 dle	20 sp	30 0	40 @	50 P	60 `	70 p	128 Ç	144 É	160 á	176 ▒	192 └	208 ╨	224 α	240 ≡
01 soh	11 dc1	21 !	31 1	41 A	51 Q	61 a	71 q	129 ü	145 æ	161 í	177 ▒	193 ┴	209 ╤	225 ß	241 ±
02 stx	12 dc2	22 "	32 2	42 B	52 R	62 b	72 r	130 é	146 Æ	162 ó	178 ▓	194 ┬	210 ╥	226 Γ	242 ≥
03 etx	13 dc3	23 #	33 3	43 C	53 S	63 c	73 s	131 â	147 ô	163 ú	179 │	195 ├	211 ╙	227 π	243 ≤
04 eot	14 dc4	24 $	34 4	44 D	54 T	64 d	74 t	132 ä	148 ö	164 ñ	180 ┤	196 ─	212 ╘	228 Σ	244 ⌠
05 enq	15 nak	25 %	35 5	45 E	55 U	65 e	75 u	133 à	149 ò	165 Ñ	181 ╡	197 ┼	213 ╒	229 σ	245 ⌡
06 ack	16 syn	26 &	36 6	46 F	56 V	66 f	76 v	134 å	150 û	166 ª	182 ╢	198 ╞	214 ╓	230 µ	246 ÷
07 bel	17 etb	27 '	37 7	47 G	57 W	67 g	77 w	135 ç	151 ù	167 º	183 ╖	199 ╟	215 ╫	231 τ	247 ≈
08 bs	18 can	28 (	38 8	48 H	58 X	68 h	78 x	136 ê	152 ÿ	168 ¿	184 ╕	200 ╚	216 ╪	232 Φ	248 °
09 ht	19 em	29)	39 9	49 I	59 Y	69 i	79 y	137 ë	153 Ö	169 ⌐	185 ╣	201 ╔	217 ┘	233 Θ	249 ·
0a nl	1a sub	2a *	3a :	4a J	5a Z	6a j	7a z	138 è	154 Ü	170 ¬	186 ║	202 ╩	218 ┌	234 Ω	250 ·
0b vt	1b esc	2b +	3b ;	4b K	5b [	6b k	7b {	139 ï	155 ¢	171 ½	187 ╗	203 ╦	219 █	235 δ	251 √
0c np	1c fs	2c ,	3c <	4c L	5c \	6c l	7c |	140 î	156 £	172 ¼	188 ╝	204 ╠	220 ▄	236 ∞	252 n
0d cr	1d gs	2d -	3d =	4d M	5d]	6d m	7d }	141 ì	157 ¥	173 ¡	189 ╜	205 ═	221 ▌	237 φ	253 ²
0e so	1e rs	2e .	3e >	4e N	5e ^	6e n	7e ~	142 Ä	158 ₧	174 «	190 ╛	206 ╬	222 ▐	238 ε	254 ■
0f si	1f us	2f /	3f ?	4f O	5f _	6f o	7f del	143 Å	159 ƒ	175 »	191 ┐	207 ╧	223 ▀	239 ∩	255

-'0' is at 0x30, so that's the offset to get to numbers

-'A' is 0x41 & 'a' is 0x61, so that's the offset between upper and lowercase

-So alphabet order starts at low number and continues to a higher number.

-Other Data Type List

Text strings
- sequence of characters, terminated with NULL (0)
- typically, no hardware support

Image
- array of pixels
  - monochrome: one bit (1/0 = black/white)
  - color: red, green, blue (RGB) components (e.g., 8 bits each)
  - other properties: transparency
- hardware support:
  - typically none in general-purpose processors
  - MMX -- multiple 8-bit operations on 32-bit word

Sound
- sequence of fixed-point numbers

Chapter 3

-Transistors

-Logically, they are like switches. On or off.

-Their state depends on the voltage.

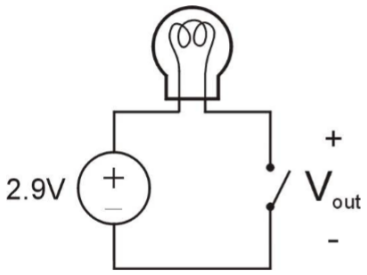
-Combine to create logic gates, which combine creating higher-level structures (Adder, multiplexer, decoder, register, etc.), which combine to create processors

-Modern CPUs have hundreds of millions of transistors

- Microprocessors contain millions of transistors
  - Intel Pentium 4 (2000): 48 million
  - IBM PowerPC 750FX (2002): 38 million
  - IBM/Apple PowerPC G5 (2003): 58 million
- Logically, each transistor acts as a switch
- Combined to implement logic functions
  - AND, OR, NOT
- Combined to build higher-level structures
  - Adder, multiplexer, decoder, register, …
- Combined to build processor
  - LC-3

-Simple Switch Circuit

-Eww volta flow

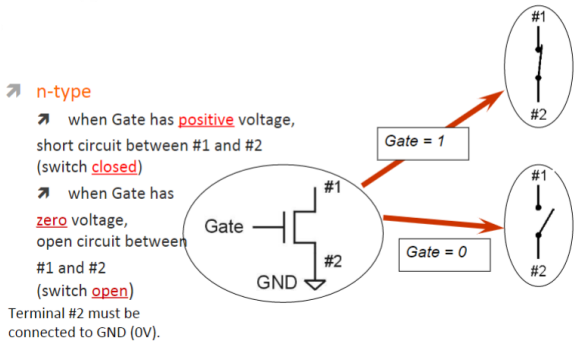


- Switch open:
  - No current through circuit
  - Light is off
  - V_{out} is +2.9V
- Switch closed:
  - Short circuit across switch
  - Current flows
  - Light is on
  - V_{out} is 0V

Switch-based circuits can easily represent two states: on/off, open/closed, voltage/no voltage.

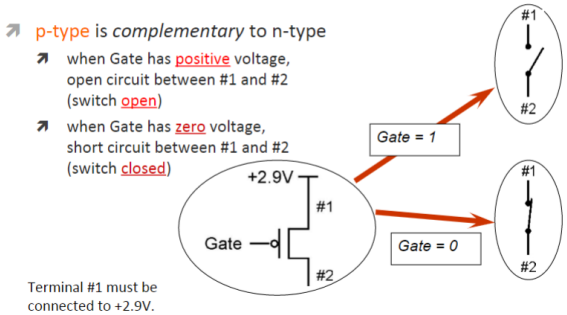
-Metal Oxide Semiconductor (MOS) Transistor

-n-type (+V = short circuit)



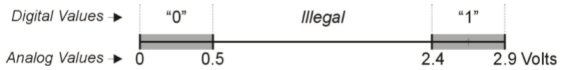
-p-type (+V = open)

-Complement to n-type



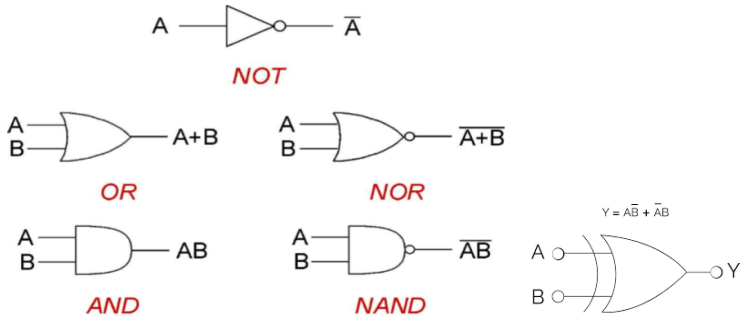
-Voltage can be uncertain so, there's a range that corresponds to each state & an illegal/unused state

- Use switch behavior of MOS transistors to implement logical functions: AND, OR, NOT.
- Digital symbols:
  - recall that we assign a range of analog voltages to each digital (logic) symbol



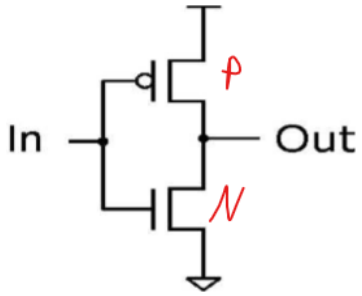
- assignment of voltage ranges depends on electrical properties of transistors being used
  - typical values for "1": +5V, +3.3V, +2.9V
  - from now on we'll use +2.9V

-Logic Gates & Transistors

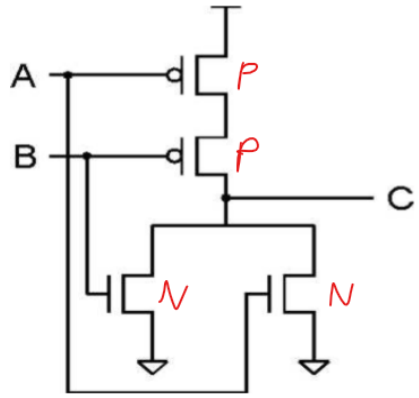


-There are 16 different 2-input logic gates

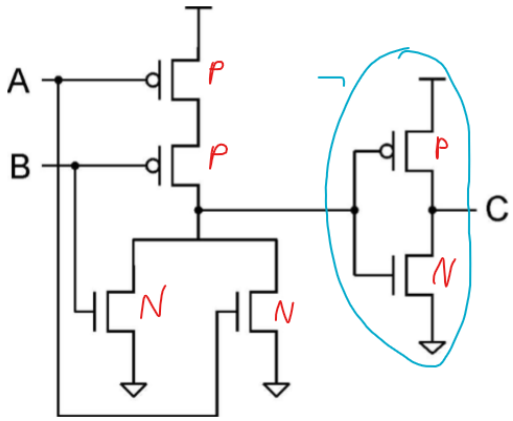
-Inverter (NOT)



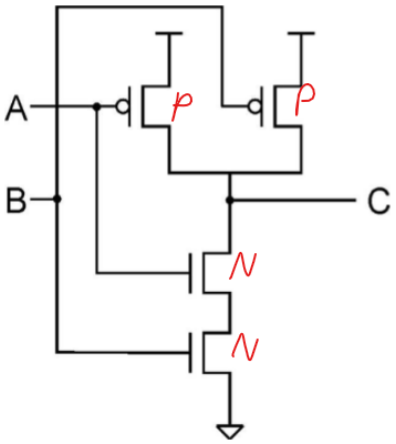
-NOR



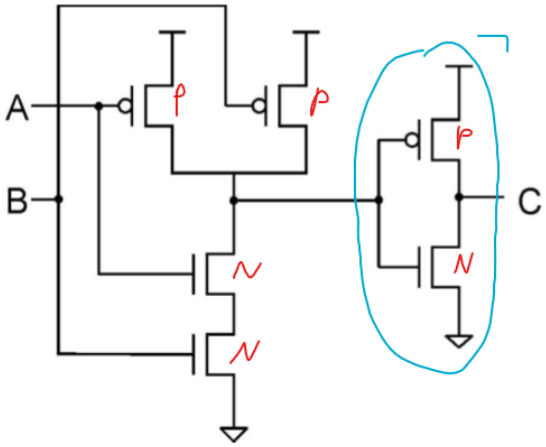
-XOR (NOR & NOT)



-NAND



-AND



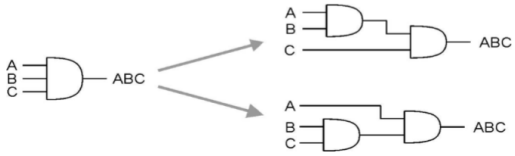
-Actual 321 ptsd

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

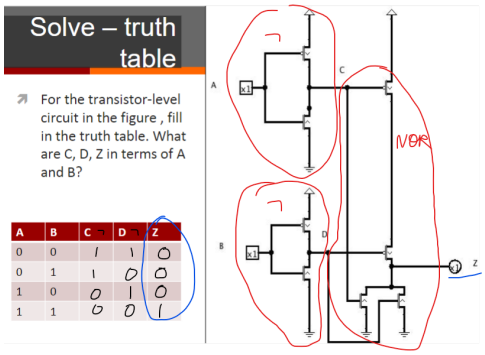
-Multiple Input

-Just combine 2-inputs 4head.

-It actually cheaper and more optimized to use 2-inputs in irl circuits

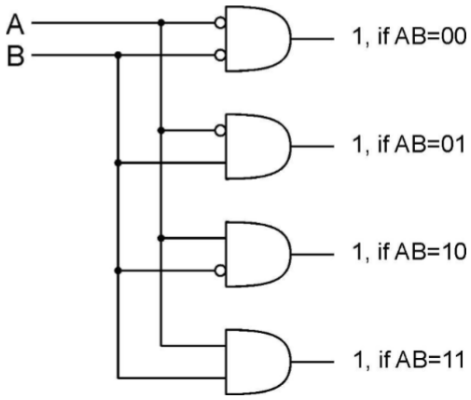


-Ex: Transistor truth table problem



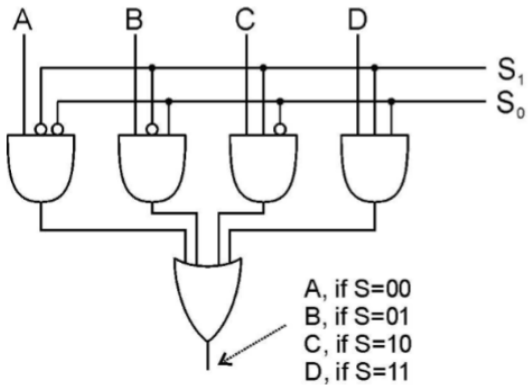
-Decoder

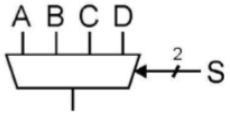
-Only one AND gate will give a T output at a time. Useful for memory address decoding



-Multiplexer (MUX)

-With S1 & S2 input, their value selects which input (A, B, C, D) you want to let through.



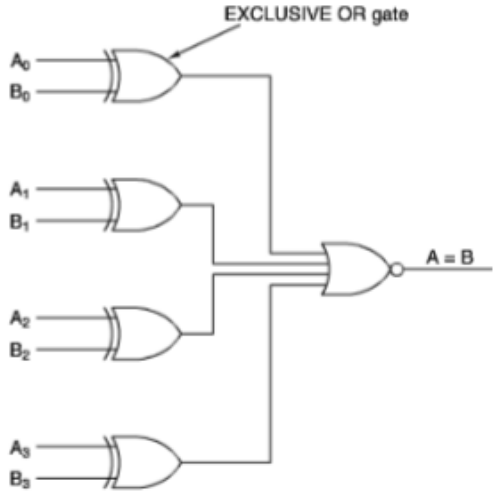


4-to-1 MUX

-Comparator (4-bit)

-A & B are 4-bit inputs. Each bit is split into an XOR gate. If all 4 XOR are true, the AND is true and is the output.

-You can add more XOR for more bits.



-Logical Completeness

-You can implement any truth table with AND and OR.

-Truth Table to Gate-Level

-If we are given a truth table and need to implement it with gates, look at the rows resulting in true.

A	B	C	Z	
0	0	0	1	$\neg A \wedge \neg B \wedge \neg C$
0	0	1	0	
0	1	0	0	
0	1	1	1	$\neg A \wedge B \wedge C$
1	0	0	0	
1	0	1	1	$A \wedge \neg B \wedge C$
1	1	0	1	$A \wedge B \wedge \neg C$
1	1	1	1	$A \wedge B \wedge C$

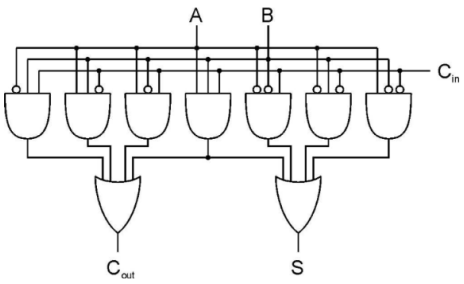
$$(\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C)$$

-There's your circuit (5: 3in-ANDs & 1: 5in-OR for 3 inputs)

-You can simplify doing 321 PTSD bruh (Or Karnaugh Map cheese?)

-Full Adder 1-Bit (2 bits & Carry in, 1 Bit and Carry out)

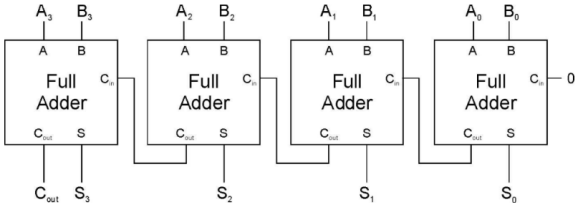
Add two bits and carry-in,
produce one-bit sum and carry-out.



A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

-Chain it, connecting the carry out to the next Adder. Store the output bit to memory.

-4-bit Adder



-Half Adder is this without carrying

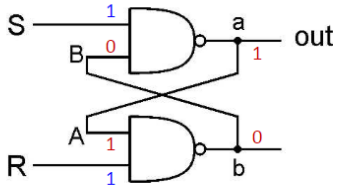
-Combinational vs. Sequential

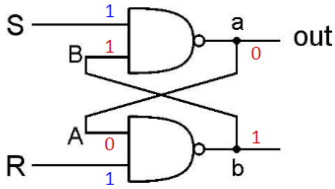
- Combinational Circuit
  - always gives the same output for a given set of inputs
    - ex: adder always generates sum and carry, regardless of previous inputs
- Sequential Circuit
  - stores information
  - output depends on stored information (state) plus input
    - so a given input might produce different outputs, depending on the stored information
  - *example:* ticket counter
    - advances when you push the button
    - output depends on previous state
  - useful for building "memory" elements and "state machines"

-R-S Latch: Simple Storage Element (volatile) (aka R-S flip flop)

-Flip between 1 or 0

- R is used to "reset" or "clear" the element – set it to zero.



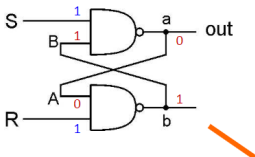


- S is used to "set" the element – set it to one.
- If both R and S are one, out could be either zero or one.
  - "quiescent" state -- holds its previous value
  - note: if a is 1, b is 0, and vice versa

- R = S = 1
  - hold current value in latch
- S = 0, R=1
  - set value to 1
- R = 0, S = 1
  - set value to 0
- R = S = 0
  - both outputs equal one
  - final state determined by electrical properties of gates
  - *Don't do it!*

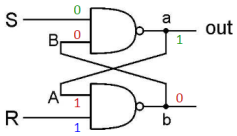
R	S	
0	0	**Illegal**
0	1	**Set Out to 0**
1	0	**Set Out to 1**
1	1	**Remember Out**

-Setting



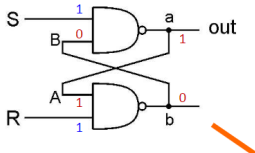
- Suppose we start with output = 0, then change S to zero.

Output changes to one.



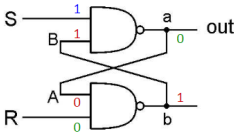
Then set S=1 to "store" value in quiescent state.

-Clearing



- Suppose we start with output = 1, then change R to zero.

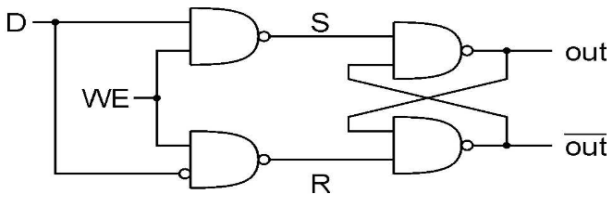
Output changes to zero.



Then set R=1 to "store" value in quiescent state.

-Gated D-Latch (volatile too)

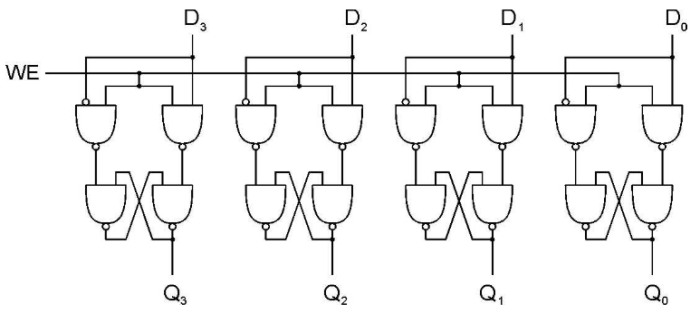
- Two inputs: D (data) and WE (write enable)
  - when WE = 1, latch is set to value of D
    - S = NOT(D), R = D
  - when WE = 0, latch holds previous value
    - S = R = 1



-The better version of R-S Latch, not having an ambiguous state.

-Chain them together, a Register! (Pic of 4-bit)

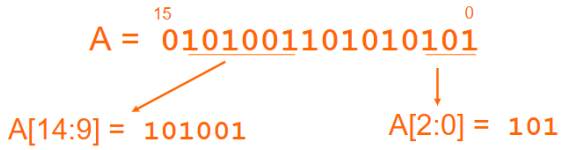
- A register stores a multi-bit value.
  - We use a collection of D-latches, all controlled by a common WE.
  - When WE=1, n-bit value D is written to register.



-Representing Multi-bit Values (A[:] crap)

Number bits from right (0) to left (n-1)

- just a convention -- could be left to right, but must be *consistent*



- Use brackets to denote range:
  D[l:r] denotes bit l to bit r, from *left* to *right*
- May also see A<14:9>,
  especially in hardware block diagrams.

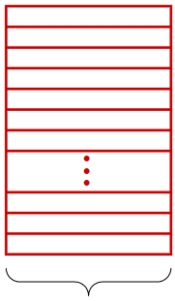
-Very useful for memory address

-Memory

- Now that we know how to store bits, we can build a memory – a logical $k \times m$ array of stored bits.

Address Space:
number of locations
(usually a power of 2)

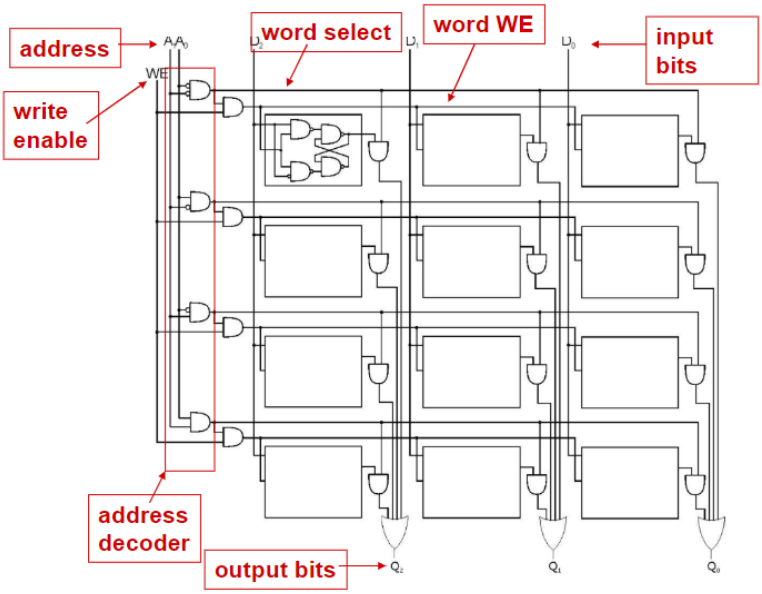
$k = 2^n$ locations



Addressability:
number of bits per location
(e.g., byte-addressable)

m bits

-By using an address, we can change where we place the data (Pic of 2^2 * 3 Memory, two input addresses like x,y coordinates)



-In a CPU, the clock tick would engage the different stages of storing memory. One tick for decoding address, another for triggering WE, etc.

-Other info

- This is a not the way actual memory is implemented.
  - fewer transistors, much more dense, relies on electrical properties
- But the logical structure is very similar.
  - address decoder
  - word select line
  - word write enable
- Two basic kinds of RAM (Random Access Memory)
- Static RAM (SRAM)
  - fast, maintains data as long as power applied
- Dynamic RAM (DRAM)
  - slower but denser, bit storage decays – must be periodically refreshed

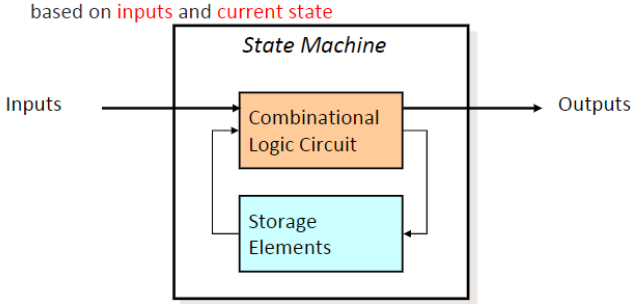
Also, non-volatile memories: ROM, PROM, flash, …

-State Machine

-A system with a finite number of states (ie green, yellow, red traffic lights).

-CPU is one. Logic and Storage, reading inputs and giving output. Logic components like ALU, & Storage like registers.7

- Another type of sequential circuit
  - Combines combinational logic with storage
  - "Remembers" state, and changes output (and state) based on inputs and current state



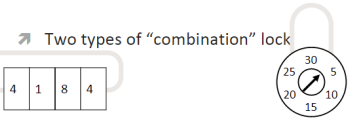
- A description of a system with the following components:

1. A finite number of states
2. A finite number of external inputs
3. A finite number of external outputs
4. An explicit specification of all state transitions
5. An explicit specification of what determines each external output value

- Often described by a state diagram.
  - Inputs trigger state transitions.
  - Outputs are associated with each state (or with each transition).

-Combination vs Sequential

- Two types of "combination" lock

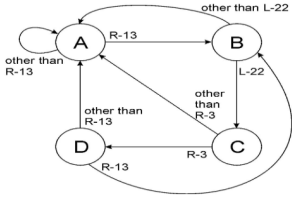


Combinational
Success depends only on the values, not the order in which they are set.

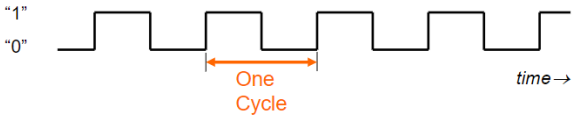
Sequential
Success depends on the sequence of values (e.g, R-13, L-22, R-3).

-State Diagram (Ex sequential)

- Shows states and actions that cause a transition between states.



-The Clock Signal

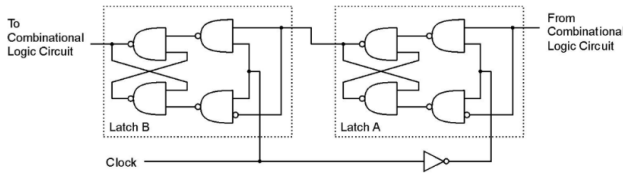


- Frequently, a clock circuit triggers transition from one state to the next.
- At the beginning of each clock cycle, state machine makes a transition, based on the current state and the external inputs.
  - Not always required. In lock example, the input itself triggers a transition.

-Storage: Master-Slave Flipflop

-Depending on the clock's phase, you can switch between reading and writing 2 D-Latches

- A pair of gated D-latches, to isolate *next* state from *current* state.



During 1st phase (clock=1), previously-computed state becomes *current* state and is sent to the logic circuit.

During 2nd phase (clock=0), *next* state, computed by logic circuit, is stored in Latch A.

- Each master-slave flipflop stores one state bit.
- The number of storage elements (flipflops) needed is determined by the number of states (and the representation of each state).
- Examples:
  - Sequential lock
    - Four states – two bits
  - Basketball scoreboard
    - 7 bits for each score, 5 bits for minutes, 6 bits for seconds, 1 bit for possession arrow, 1 bit for half, …

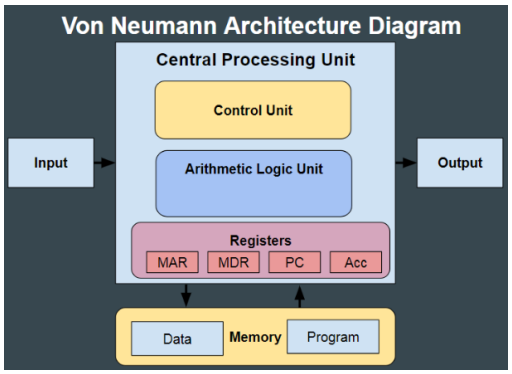
Chapter 4: Von Neumann Model and Machine Instructions

-The Stored Program Computer

-Brief and incomplete history of the computers that started storing programs to execute (instead of being designed to do one program)

- 1943: ENIAC (Electronic Numerical Integrator And Computer)
  - Presper Eckert and John Mauchly -- first general electronic computer.
  - Hard-wired program -- settings of dials and switches.
- 1944: Beginnings of EDVAC (Electronic Discrete Variable Automatic Computer)
  - Binary instead of decimal.
  - among other improvements, includes program stored in memory
- 1945: John von Neumann
  - wrote a report on the stored program concept, known as the *First Draft of a Report on EDVAC*
- The basic structure proposed in the draft became known as the "von Neumann machine" (or model) (or architecture).
  - a *memory*, containing instructions and data
  - a *processing unit*, for performing arithmetic and logical operations
  - a *control unit*, for interpreting instructions

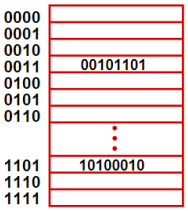
-John Von Neumann made the seminal draft of the CPU, Memory, Control Unit computer, storing programs and being able to execute them.



-Memory

-Address and Operations

- 2^k x m array of stored bits
- Address
  - unique (k-bit) identifier of location
- Contents
  - m-bit value stored in location
- Basic Operations:
- LOAD
  - read a value from a memory location into the MDR
- STORE
  - write a value to a memory location from the MDR



-Interface to Memory

-You need something to manage address and another to manage the data to process (Memory Data Register & Memory Address Register)

- How does the processing unit get data to/from memory?
- MAR: Memory Address Register
- MDR: Memory Data Register

- To LOAD a location (A) from memory into the MDR:
  1. Write the address (A) into the MAR.
  2. Send a"read" signal to the memory.
  3. Read the data from memory into the MDR.
- To STORE a value (X) form the MDR into a location (A) in memory:
  1. Write the data (X) to the MDR.
  2. Write the address (A) into the MAR.
  3. Send a "write" signal to the memory.

-Processing Unit

-ALU, Registers, and

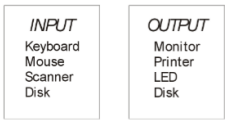
- Functional Units
  - ALU = Arithmetic and Logic Unit
  - could have many functional units. some of them special-purpose (multiply, square root, ...)
  - LC-3 performs ADD, AND, NOT

- Registers
  - Small, temporary storage
  - Operands and results of functional units
  - LC-3 has eight registers (R0, ..., R7), each 16 bits wide
- Word Size
  - number of bits normally processed by ALU in one instruction
  - also width of registers
  - LC-3 is 16 bits

-I/O

-Input/Output uses drivers to interact with the processor in order to manipulate memory

- Devices for getting data into and out of computer memory
- Each device has its own interface, usually a set of registers like the memory's MAR and MDR



  - LC-3 supports keyboard (input) and monitor (output)
  - keyboard: data register (KBDR) and status register (KBSR)
  - monitor: data register (DDR) and status register (DSR)
- Some devices provide both input and output
  - disk, network
- Program that controls access to a device is usually called a *driver*.

-Control Unit

-Instruction Register, Program Counter makes up the Control Unit

- Orchestrates execution of the program
- Instruction Register (IR) contains the *current instruction*.
- Program Counter (PC) contains the *address* of the next instruction to be executed.
- Control unit:
  - reads an instruction from memory
    - the instruction's address is in the PC
  - interprets the instruction, generating signals that tell the other components what to do
    - an instruction may take many *machine cycles* to complete

-Instruction

- The instruction is the fundamental unit of work.
- Specifies two things:
  - *opcode*: operation to be performed
  - *operands*: data/locations to be used for operation
- An instruction is encoded as a sequence of bits. *(Just like data!)*
  - Often, but not always, instructions have a fixed length, such as 16 or 32 bits.
  - Control unit interprets instruction: generates sequence of control signals to carry out operation.
  - Operation is either executed completely, or not at all.
- A computer's instructions and their formats is known as its *Instruction Set Architecture (ISA)*.

-For 16 ibt

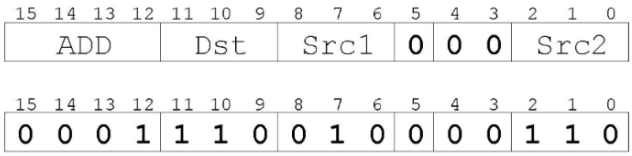
-First 4 bits for opcodes (16 opcodes) (LC-3 ISA has 15 instructions)

-Textbook has LC-3 ISA in appendix

-Intel ISA Windows documentation cancer https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

-Ex: LC-3 ADD instruction

- LC-3 has 16-bit instructions.
  - Each instruction has a four-bit opcode, bits [15:12].
- LC-3 has eight *registers* (R0-R7) for temporary storage.
  - Sources and destination of ADD are registers.

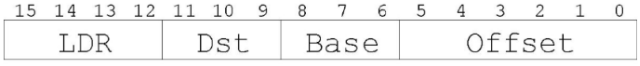


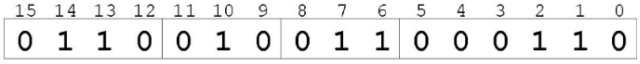
"Add the contents of R2 to the contents of R6, and store the result in R6."

-Src1 is where in the register is the first number
-Src2 is the 2nd number
-Dst is the destination to write.
-Ex here overwrites Src2 with the calculation

-Ex: LC-3 LDR (load memory to CPU)

- Load instruction -- reads data from memory
- Base + offset mode:
  - add offset to base register -- result is memory address
  - load from memory address into destination register



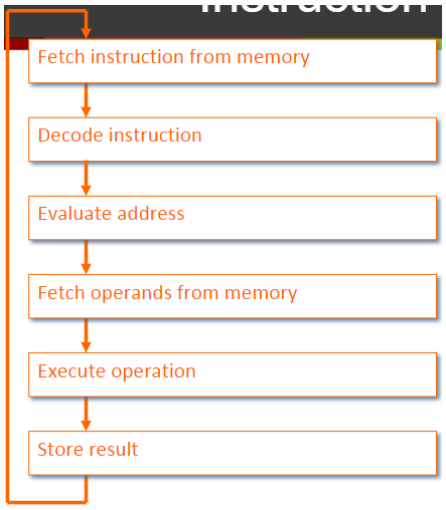


"Add the value 6 to the contents of R3 to form a memory address. Load the contents of that memory location to R2."

-This uses Base+Offset memory retrieving mode

-Instruction Processing

-The cycle that allows the computer memory to change by processor



-The cycle is implemented by writing machine code

-Fetch the instruction from memory

- 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.
- Then increment PC, so that it points to the next instruction in sequence.
  - PC becomes PC+1.

-Decode the instruction

- First identify the opcode.
  - In LC-3, this is always the first four bits of instruction.
  - A 4-to-16 decoder asserts a control line corresponding to the desired opcode.
- Depending on the opcode, identify other operands from the remaining bits.
  - Example:
    - for LDR, last six bits is offset
    - for ADD, last three bits is source operand #2

-A bit mask singles out operand or opcode

-Evaluate the address

-Only when you have instructions that use addresses (opposite of execute)

- For instructions that require memory access, compute address used for access.
- Examples:
  - add offset to base register (as in LDR)
  - add offset to PC
  - add offset to zero

-Fetch operands

- Obtain source operands needed to perform operation.
- Examples:
  - load data from memory (LDR)
  - read data from register file (ADD)

-Execute

-Only when you need to calculate something. (opposite of evaluate address)

- Perform the operation, using the source operands.
- Examples:
  - send operands to ALU and assert ADD signal
  - do nothing (e.g., for loads and stores)

-Store results

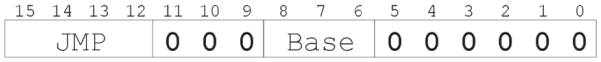
- Write results to destination. (register or memory)
- Examples:
  - result of ADD is placed in destination register
  - result of memory load is placed in destination register
  - for store instruction, data is stored to memory
    - write address to MAR, data to MDR
    - assert WRITE signal to memory

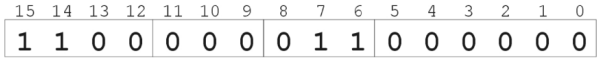
-We can skip the ones we don't what to do.

- In the FETCH phase, we increment the Program Counter by 1.
- What if we don't want to always execute the instruction that follows this one?
  - examples: loop, if-then, function call
- Need special instructions that change the contents of the PC.
- These are called *control instructions*.
  - jumps are unconditional -- they always change the PC
  - branches are conditional -- they change the PC only if some condition is true (e.g., the result of an ADD is zero)

-LC-3 JMP (jump) instruction

- Set the PC to the value contained in a register. This becomes the address of the next instruction to fetch.





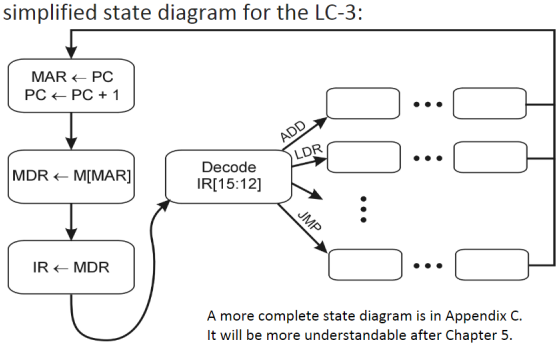
"Load the contents of R3 into the PC."

-Summary

- Instructions look just like data -- its all interpretation.
- Three basic kinds of instructions:
  - computational instructions (ADD, AND, …)
  - data movement instructions (LD, ST, …)
  - control instructions (JMP, BRnz, …)
- Six basic phases of instruction processing:
- $F \rightarrow D \rightarrow EA \rightarrow OP \rightarrow EX \rightarrow S$
  - not all phases are needed by every instruction
  - phases may take variable number of machine cycles (machine cycle = CPU hz)

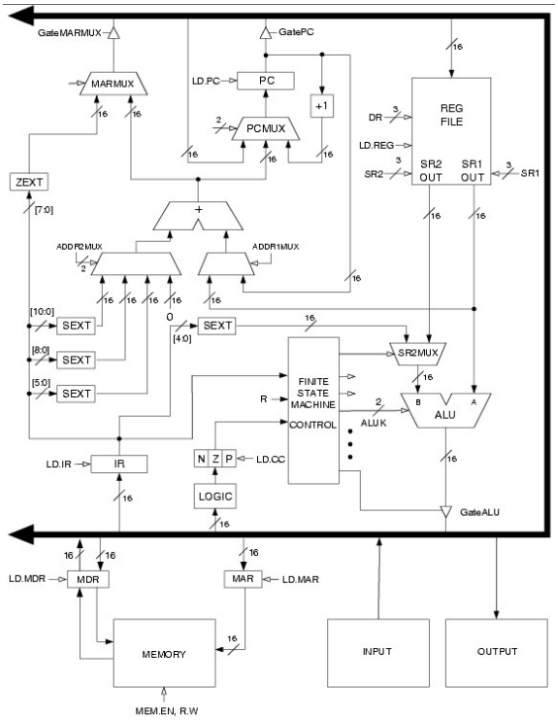
-LC-3 Control Unit State Diagram

- The control unit is a state machine. Here is part of a simplified state diagram for the LC-3:



A more complete state diagram is in Appendix C.
It will be more understandable after Chapter 5.

-LC-3 Data Path



Filled arrow
= info to be processed.
Unfilled arrow
= control signal.

-The bold black line is the Bus

Chapter 5: LC-3 ISA

-Wat the ISA?

- ISA = All of the *programmer-visible* components and operations of the computer
  - memory organization
    - address space -- how may locations can be addressed?
    - addressability -- how many bits per location?
  - register set
    - how many? what size? how are they used?
  - instruction set
    - opcodes
    - data types
    - addressing modes
- ISA provides all information needed for someone that wants to write a program in machine language (or translate from a high-level language to machine language).

-LC-3 Memory& Register

- Memory
  - address space: 2^{16} locations (16-bit addresses)
  - addressability: 16 bits
- Registers
  - temporary storage, accessed in a single machine cycle
    - accessing memory generally takes longer than a single cycle
  - eight general-purpose registers: R0 - R7
    - each 16 bits wide
    - how many bits to uniquely identify a register?
  - other registers
    - not directly addressable, but used by (and affected by) instructions
    - PC (program counter), condition codes

-LC-3 ISA

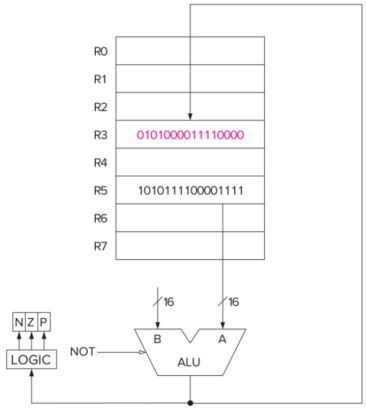
- Opcodes
  - 15 opcodes
  - *Operate* instructions: ADD, AND, NOT
  - *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
  - *Control* instructions: BR, JSR/JSRR, JMP, RTI, TRAP
  - some opcodes set/clear *condition codes*, based on result:
    - N = negative, Z = zero, P = positive (> 0)
- Data Types
  - 16-bit 2's complement integer
- Addressing Modes
  - How is the location of an operand specified?
  - non-memory addresses: *immediate*, *register*
  - memory addresses: *PC-relative*, *indirect*, *base+offset*

- Only three operations: ADD, AND, NOT
- Source and destination operands are registers
  - These instructions *do not* reference memory.
  - ADD and AND can use "immediate" mode, where one operand is hard-wired into the instruction.
- Will show dataflow diagram with each instruction.
  - illustrates *when* and *where* data moves to accomplish the desired operation

-LC-3 NOT

-1001 is the opcode, Dst is the destination on the register, Src is the source of the register, 111111 is unused.

-Ex: NOT R3 R5 (1001 R3 R5 111111)



(NOT sets the logic bit. In this case, it'll be 100 because the result is negative)

-LC-3 Workflow

-Memory (storage space for your program starting at x3000) and Registry (storage for values)

-Fat LC-3 instructions

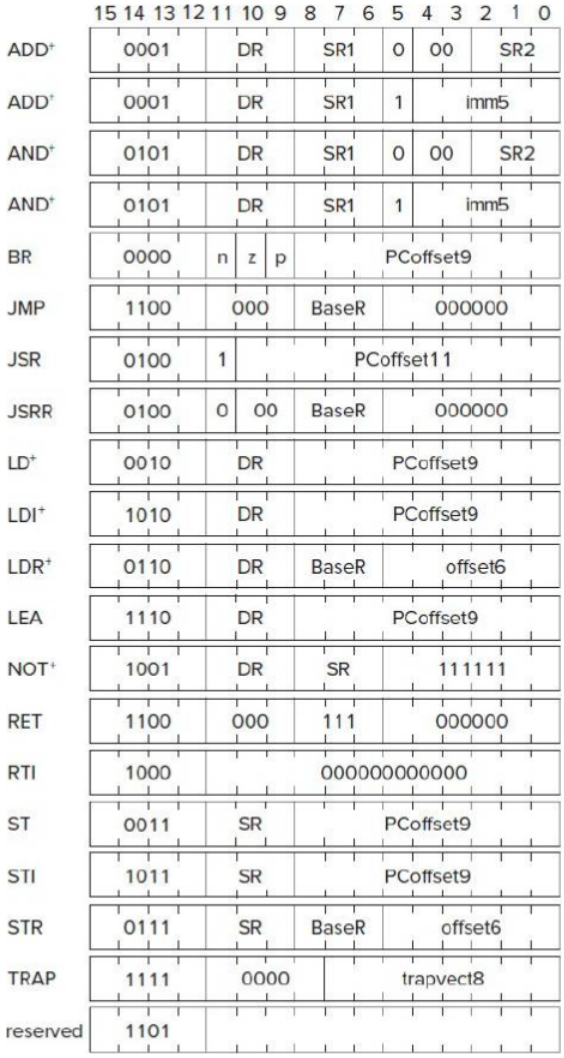


Table A.3 Trap Service Routines

Trap Vector	Assembler Name	Description
x20	GETC	Read a single character from the keyboard. The character is not echoed onto the console. Its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x21	OUT	Write a character in R0[7:0] to the console display.
x22	PUTS	Write a string of ASCII characters to the console display. The characters are contained in consecutive memory locations, one character per memory location, starting with the address specified in R0. Writing terminates with the occurrence of x0000 in a memory location.
x23	IN	Print a prompt on the screen and read a single character from the keyboard. The character is echoed onto the console monitor, and its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x24	PUTSP	Write a string of ASCII characters to the console. The characters are contained in consecutive memory locations, two characters per memory location, starting with the address specified in R0. The ASCII code contained in bits [7:0] of a memory location is written to the console first. Then the ASCII code contained in bits [15:8] of that memory location is written to the console. (A character string consisting of an odd number of characters to be written will have x00 in bits [15:8] of the memory location containing the last character to be written.) Writing terminates with the occurrence of x0000 in a memory location.
x25	HALT	Halt execution and print a message on the console.

-imm5: 5-bit immediate number. If you need values greater than that, repeatedly ADD to a register or something.
-PCoffset9: 9-bit offset to the current address the PC is at. (If going backwards, negative offset, sign extend)
-If instruction in memory is at x3000, treat it as x3001
-This is because LC-3 fetches the instruction which increments PC before executing the instruction
-Also, storing program into memory at x3000 doesn't count as a line (0011 000 000000000)
-The "+" indicates nzp update.

Chapter 7: Assembly Language

-Literally just coding Machine Code that's more human readable

Computers like ones and zeros…

```
0001110010000110
```

Humans like symbols…

```
ADD  R6,R2,R6  ; increment index reg.
```

Assembler is a program that turns symbols into machine instructions.

- **ISA-specific:**
  close correspondence between symbols and instruction set
  - **mnemonics for opcodes**
  - **labels for memory locations**
- **additional operations for allocating storage and initializing data**

-Ex1:

```
;
; Program to multiply a number by the constant 6
;
        .ORIG  x3050
        LD     R1, SIX
        LD     R2, NUMBER
        AND    R3, R3, #0     ; Clear R3.  It will
                              ; contain the product.
; The inner loop
;
AGAIN   ADD    R3, R3, R2
        ADD    R1, R1, #-1    ; R1 keeps track of
        BRp    AGAIN          ; the iteration.
;
        HALT
;
NUMBER  .BLKW  1
SIX     .FILL  x0006
;
        .END
```

-LC-3 Assembly Syntax

Each line of a program is one of the following:

- an instruction
- an assembler directive (or pseudo-op)
- a comment

Whitespace (between symbols) and case are ignored.
Comments (beginning with ";") are also ignored.

An instruction has the following format:

```
LABEL OPCODE OPERANDS ; COMMENTS
```

optional *mandatory*

Opcodes

- reserved symbols that correspond to LC-3 instructions
- listed in Appendix A
  - ex: `ADD, AND, LD, LDR, ...`

Operands

- registers -- specified by Rn, where n is the register number
- numbers -- indicated by # (decimal) or x (hex)
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format
  - ex:

```
ADD R1,R1,R3
ADD R1,R1,#3
LD  R6,NUMBER
BRz LOOP
```

Label

- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
  - ex:

```
LOOP ADD R1,R1,#-1
     BRp LOOP
```

Comment

- anything after a semicolon is a comment
- ignored by assembler
- used by humans to document/understand programs
- tips for useful comments:
  - avoid restating the obvious, as "decrement R1"
  - provide additional insight, as in "accumulate product in R6"
  - use comments to separate pieces of program

-Operands still has the limitations of the operations, so you imm5.
-A use for Labels is replacing PCoffset, allowing you to break back to the label.
-In between HALT and .END, we can use variables to point to memory
-DON'T USE R0!!! Some commands uses R0 to process data (ie input).

-LC-3 Assembly Directives

Assembler Directives

Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but "opcode" starts with dot

Opcode	*Operand*	*Meaning*
.ORIG	address	starting address of program
.END		end of program
.BLKW	n	allocate n words of storage
.FILL	n	allocate one word, initialize with value n
.STRINGZ	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Trap Codes

LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them.

Code	*Equivalent*	*Description*
HALT	TRAP x25	Halt execution and print message to console.
IN	TRAP x23	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	TRAP x21	Write one character (in R0[7:0]) to console.
GETC	TRAP x20	Read one character from keyboard. Character stored in R0[7:0].
PUTS	TRAP x22	Write null-terminated string to console. Address of string is in R0.

-.FILL, .BLKW, .STRINGZ automatically allocate memory for use.
-You don't have to put anything into the allocated memory on declaration.
-You can edit these memory by ST.

-Style (& character reading program)
-4 Columns (Label, Opcode, Operand, Comment) OP cheese

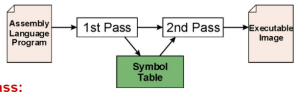
Use the following style guidelines to improve the readability and understandability of your programs:

1. Provide a program header, with author's name, date, etc., and purpose of program.
2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
3. Use comments to explain what each register does.
4. Give explanatory comment for most instructions.
5. Use meaningful symbolic names.
   - Mixed upper and lower case for readability.
   - ASCIItoBinary, InputRoutine, SaveR1
6. Provide comments between program sections.
7. Each line must fit on the page -- no wraparound or truncations.
   - Long statements split in aesthetically pleasing manner.

```
;
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
; Initialization
;
        .ORIG   x3000
        AND     R2, R2, #0      ; R2 is counter, initially 0
        LD      R3, PTR         ; R3 is pointer to characters
        GETC                    ; R0 gets character input
        LDR     R1, R3, #0      ; R1 gets first character
;
; Test character for end of file
;
TEST    ADD     R4, R1, #-4     ; Test for EOT (ASCII x04)
        BRz     OUTPUT          ; If done, prepare the output
;
; Test character for match.  If a match, increment count.
;
        NOT     R1, R1
        ADD     R1, R1, R0  ; If match, R1 = xFFFF
        NOT     R1, R1      ; If match, R1 = x0000
        BRnp    GETCHAR     ; If no match, do not increment
        ADD     R2, R2, #1
;
; Get next character from file.
;
GETCHAR ADD     R3, R3, #1  ; Point to next character.
        LDR     R1, R3, #0  ; R1 gets next char to test
        BRnzp   TEST
;                                               ;
; Output the count.                             ; Storage for pointer and ASCII template
;                                               ;
OUTPUT  LD      R0, ASCII   ; Load the ASCII template       ASCII   .FILL   x0030
        ADD     R0, R0, R2  ; Covert binary count to ASCII  PTR     .FILL   x4000
        OUT                 ; ASCII code in R0 is displayed.        .END
        HALT                ; Halt machine
```

-Assembly Process

Convert assembly language file (.asm) into an executable file (.obj) for the LC-3 simulator.



First Pass:

- scan program file
- find all labels and calculate the corresponding addresses; this is called the *symbol table*

Second Pass:

- convert instructions to machine language, using information from symbol table

First Pass: Constructing the Symbol Table

1. Find the `.ORIG` statement, which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      - NOTE: If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated.
3. Stop when `.END` statement is reached.

Second Pass: Generating Machine Language

For each executable assembly language statement, generate the corresponding machine language instruction.

- If operand is a label, look up the address from the symbol table.

Potential problems:

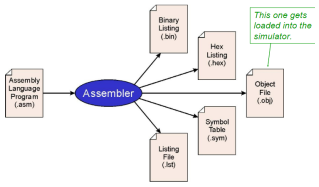
- Improper number or type of arguments
  - ex:

```
NOT  R1,#7
ADD  R1,R2
ADD  R3,R3,NUMBER
```

- Immediate argument too large
  - ex: `ADD R1,R2,#1023`
- Address (associated with label) more than 256 from instruction
  - can't use PC-relative addressing mode

LC-3 Assembler

Using "assemble" (Unix) or LC3Edit (Windows), generates several different output files.



-The assembler uses the Location Counter (LC) to allocate memory for the variable of the program.
-.END is hw the LC knows when to stop
-Labels, .FILL, .BLKW, .STRINGZ allocates the names to the addresses.

Chapter 8: I/O

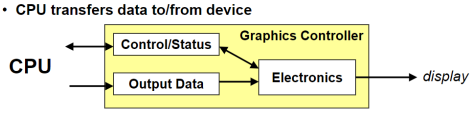
-We have register and memory.
-But where does data in memory come from?
-And how does data get out of the system so that humans can use it?
-Categorized by
-Behavior
-Input, Output, Storage
-Data Rate
-Speed
-I/O Controller
-There are other registers not used by the programmer / for the system to interact with I/O

Control/Status Registers

- CPU tells device what to do -- write to control register
- CPU checks whether task is done -- read status register

Data Registers

- CPU transfers data to/from device



Device electronics

- performs actual operation
  - pixels to screen, bits to/from disk, characters from keyboard

-I/O has different ways to implement by ISA & different systems.

Programming Interface

How are device registers identified?
- Memory-mapped vs. special instructions

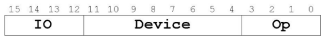
How is timing of transfer managed?
- Asynchronous vs. synchronous

Who controls transfer?
- CPU (polling) vs. device (interrupts)

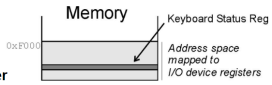
-LC3 doesn't have a special instruction for I/O.
-TRAP in LC3 is an interrupt.

-Memory-Mapped vs I/O Instructions

Instructions
- designate opcode(s) for I/O
- register and operation encoded in instruction



Memory-mapped
- assign a memory address to each device register
- use data movement instructions (LD/ST) for control and data transfer



-Async Vs Sync

Transfer Timing

I/O events generally happen much slower than CPU cycles.

Synchronous
- data supplied at a fixed, predictable rate
- CPU reads/writes every X cycles

Asynchronous
- data rate less predictable
- CPU must ***synchronize*** with device, so that it doesn't miss data or write too quickly

-Transfer Control

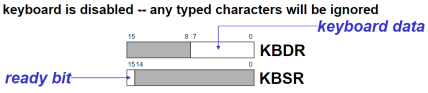
Polling
- CPU keeps checking status register until ***new data*** arrives OR ***device ready*** for next data
- "Are we there yet? Are we there yet? Are we there yet?"

Interrupts
- Device sends a special signal to CPU when ***new data*** arrives OR ***device ready*** for next data
- CPU can be performing other tasks instead of polling device.
- "Wake me when we get there."

-LC3 I/O

Location	*I/O Register*	*Function*
xFE00	Keyboard Status Reg (KBSR)	Bit [15] is one when keyboard has received a new character.
xFE02	Keyboard Data Reg (KBDR)	Bits [7:0] contain the last character typed on keyboard.
xFE04	Display Status Register (DSR)	Bit [15] is one when device ready to display another char on screen.
xFE06	Display Data Register (DDR)	Character written to bits [7:0] will be displayed on screen.

When a character is typed:
- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the "ready bit" (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored



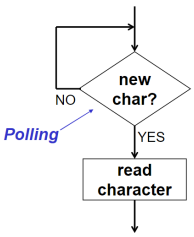
When KBDR is read:
- KBSR[15] is set to zero
- keyboard is enabled

Asynchronous devices
- synchronized through status registers

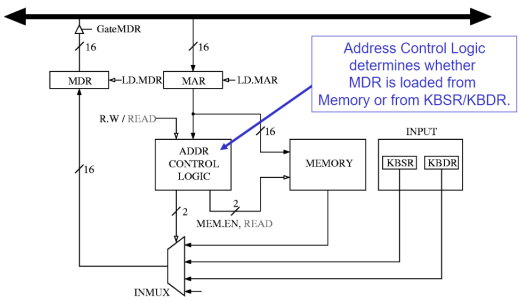
Polling and Interrupts
- the details of interrupts will be discussed in Chapter 10

-Memory mapped
-It uses 2 memory location for each because of timing.
-The KBDR is set first, then KBSR is set. Same for display.
-Reading from KBDR, KBSR is set back to 0.

-Basic Input Routine




```
POLL    LDI  R0, KBSRPtr
        BRzp POLL
        LDI  R0, KBDRPtr
        ...
KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

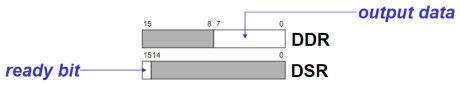


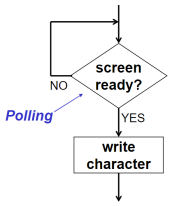
-This is basically TRAP GETC, but polling keeps looping and waste computational power.

-Same idea with output.

When Monitor is ready to display another character:

- **the "ready bit" (DSR[15]) is set to one**

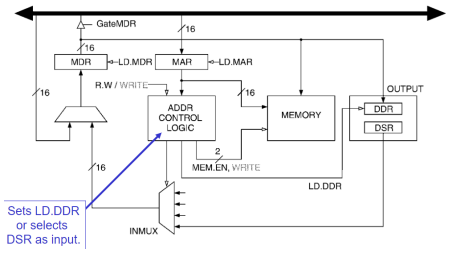





```
POLL    LDI  R1, DSRPtr
        BRzp POLL
        STI  R0, DDRPtr
        ...
DSRPtr .FILL xFE04
DDRPtr .FILL xFE06
```

When data is written to Display Data Register:

- **DSR[15] is set to zero**
- **character in DDR[7:0] is displayed**
- **any other character data written to DDR is ignored (while DSR[15] is zero)**



-Put both in and out together: routine to echo input to screen.

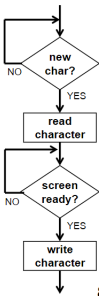
Usually, input character is also printed to screen.

- **User gets feedback on character typed and knows its ok to type the next character.**

```
POLL1   LDI  R0, KBSRPtr
        BRzp POLL1
        LDI  R0, KBDRPtr
POLL2   LDI  R1, DSRPtr
        BRzp POLL2
        STI  R0, DDRPtr

        ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
DSRPtr  .FILL xFE04
DDRPtr  .FILL xFE06
```



-Interrupt Driven I/O

-Polling waste cycles.

External device can:

(1) Force currently executing program to stop;

(2) Have the processor satisfy the device's needs; and

(3) Resume the stopped program as if nothing happened.

Why?

- **Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.**
- **Example: Process previous input while collecting current input. (See Example 8.1 in text.)**

-An interrupt mechanism has to tell the CPU when an interrupt happens (the CPU then tells OS, which then tells the program)

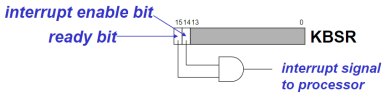
-LC3 has an "interrupt enable" bit in KBSR.

To implement an interrupt mechanism, we need:

- **A way for the I/O device to signal the CPU that an interesting event has occurred.**
- **A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.**

Generating Signal

- **Software sets "interrupt enable" bit in device register.**
- **When ready bit is set and IE bit is set, interrupt is signaled.**



-Priority

-Interrupts can have priority.

Every instruction executes at a stated level of urgency.

LC-3: 8 priority levels (PL0-PL7)

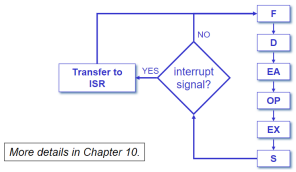
- **Example:**
  - **Payroll program runs at PL0.**
  - **Nuclear power correction program runs at PL6.**
- **It's OK for PL6 device to interrupt PL0 program, but not the other way around.**

Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.

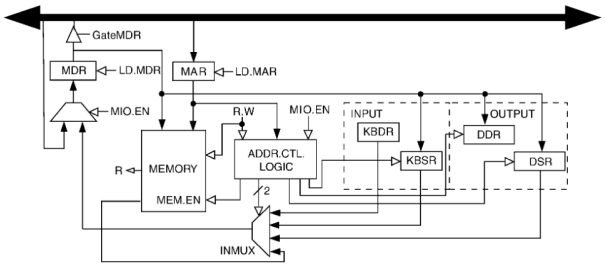
-Testing for Interrupt Signal

-Between STORE and FETCH. So you can't interrupt an instruction.

CPU looks at signal between STORE and FETCH phases.
If not set, continues with next instruction.
If set, transfers control to interrupt service routine.



-LC3 actually has Memory Mapped Interrupt Driven I/O



TRAP Routines and Subroutines LC3

-System Calls

-In LC3, this is TRAP command

Certain operations require specialized knowledge and protection:

- **specific knowledge of I/O device registers and the sequence of operations needed to use them**
- **I/O resources shared among multiple users/programs; a mistake could affect lots of other users!**

Not every programmer knows (or wants to know) this level of detail

Provide *service routines* or *system calls* (part of operating system) to safely and conveniently perform low-level, privileged operations

1. User program invokes system call.
2. Operating system code performs operation.
3. Returns control to user program.

-LC3 TRAP Mechanism

1. A set of service routines.

- **part of operating system -- routines start at arbitrary addresses** (convention is that system code is below x3000)
- **up to 256 routines**

2. Table of starting addresses.

- **stored at x0000 through x00FF in memory**
- **called System Control Block in some architectures**

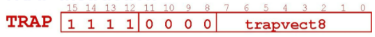
3. TRAP instruction.

- **used by program to transfer control to operating system**
- **8-bit trap vector names one of the 256 service routines**

4. A linkage back to the user program.

- **want execution to resume immediately after the TRAP instruction**

TRAP Instruction



Trap vector

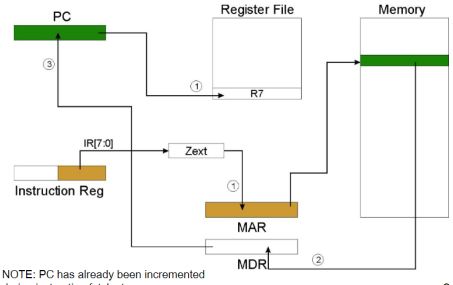
- **identifies which system call to invoke**
- **8-bit index into table of service routine addresses**
  - **in LC-3, this table is stored in memory at 0x0000 – 0x00FF**
  - **8-bit trap vector is zero-extended into 16-bit memory address**

Where to go

- **lookup starting address from table; place in PC**

How to get back

- **save address of next instruction (current PC) in R7**



vector	*symbol*	*routine*
x20	GETC	read a single character (no echo)
x21	OUT	output a character to the monitor
x22	PUTS	write a string to the console
x23	IN	print prompt to console, read and echo character from keyboard
x25	HALT	halt the program

-RET (aka Return & “JMP R7”)

-PC is temp stored in R7, RET is JMP to address in R7. (So both R0 and R7 is used for system calls)

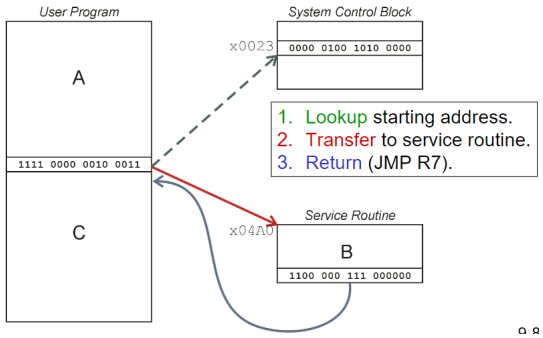
How do we transfer control back to instruction following the TRAP?

We saved old PC in R7.

- **JMP R7 gets us back to the user program at the right spot.**
- **LC-3 assembly language lets us use RET (return) in place of “JMP R7”.**

Must make sure that service routine does not change R7, or we won’t know where to return.

-TRAP Operation



-Subroutine

-Aka Function/Method

A subroutine is a program fragment that:

- **lives in user space**
- **performs a well-defined task**
- **is invoked (called) by another user program**
- **returns control to the calling program when finished**

Like a service routine, but not part of the OS

- **not concerned with protecting hardware resources**
- **no special privilege required**

Reasons for subroutines:

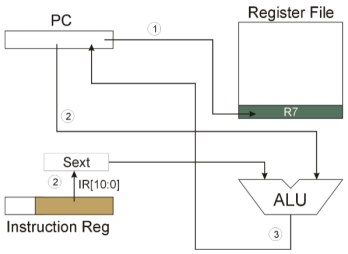
- **reuse useful (and debugged!) code without having to keep typing it in**
- **divide task among multiple programmers**
- **use vendor-supplied *library* of useful routines**

-JSR Instruction

-JSR can be used to hep to a subroutine, BR to the PCoffset while saving current PC to R7. RET will end the subroutine.

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.

- **saving the return address is called "linking"**
- **target address is PC-relative** (PC + Sext(IR[10:0]))
- **bit 11 specifies addressing mode**
  - **if =1, PC-relative: target address = PC + Sext(IR[10:0])**
  - **if =0, register: target address = contents of register IR[8:6]**



NOTE: PC has already been incremented during instruction fetch stage.

-JSRR Instruction

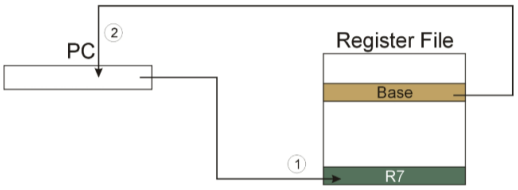
-JSR but you can specify where you want to save current PC.

-Useful for subroutine calling another subroutine, since JSR twice overrides R7.

Just like JSR, except Register addressing mode.

- **target address is Base Register**
- **bit 11 specifies addressing mode**

What important feature does JSRR provide that JSR does not?



NOTE: PC has already been incremented during instruction fetch stage.