

01 - Introduction

CEG 3310/5310 - Computer Organization
Max Gilson

Your Instructor

- Instructor
 - Max Gilson
 - BSEE - Wright State University 2018
 - MSEE - Wright State University 2019
 - 8+ years experience in industry in hardware and software development

Syllabus

- Lecture
- Lab
- Software
- Pilot
- Schedule
- Grading
- Late Assignments

AI Generative Work and Copied Work

- Using generative AI, like ChatGPT, to complete your quizzes, labs, or exams will result in a 0 for the course.
- You must be capable of writing your own code for your assignments without copying and pasting from other sources
- Copying work from others, online resources, or generative AI is an academic integrity violation

How to Get Help for Assignments Steps

Once you need help on an assignment:

1. Ask the TA during lab
2. Ask me during lecture
3. Come to my office hours
4. Ask question in email to TA
 - a. TA will not review your code over email
5. Ask question in email to me
 - a. I will not review your code over email

Note: The TA is taking their own courses too! The TA might not be able to help you the day the assignment is due!

Note: If asking for help 1 or 2 days before the lab is due don't expect an immediate help!

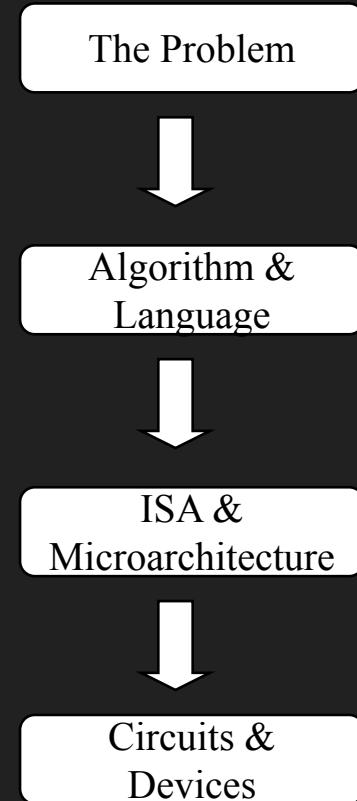
Lab

- No lab for first week
- Lab will start on second week

Why is Computer Organization Important?

- You must understand hardware:
 - To program well
 - To program correctly
- Assembly language is important

How Far Down Can We Go?



Instruction Set Architecture (ISA)

The Problem



Algorithm &
Language



ISA &
Microarchitecture



Circuits & Devices

- Instruction Set Architecture
 - Interface between programs and hardware
 - Examples: IA-32, PowerPC
- Programs get translated from high level language to ISA of computer by a compiler
- Programs translated from assembly to ISA is done by an assembler

Microarchitecture

The Problem



Algorithm &
Language



ISA &
Microarchitecture



Circuits & Devices

- Microarchitecture
 - Transforms the ISA into an implementation
- Each implementation of microarchitecture is an engineering trade-off between cost and performance

Circuits

The Problem



Algorithm &
Language



ISA &
Microarchitecture



Circuits & Devices

- The microarchitecture elements are implemented with simple logic circuits
- There are lots of choices and trade-offs
- Adders, oscillators, etc.

Devices

The Problem



Algorithm &
Language



ISA &
Microarchitecture



Circuits & Devices

- The logic circuits have to be implemented with devices
 - Transistors
 - Resistors
 - Capacitors
- Chemistry gets involved
 - CMOS, NMOS, gallium arsenide, etc.

The Reality of Computer Science

- Binary Encodings
 - Integers, floats, chars, instructions
- How do computers process instructions
 - The stack
- Memory
 - Memory references (pointers)
 - Design constraints (IoT)
- Computers do more than run programs
 - Inputs/Outputs
 - Interrupts
 - Networking
- You must understand your system to optimize performance!

02 - Bits, Data Types and Operations

CEG3310/5310 - Computer Organization
Max Gilson

Binary Encodings

- Is $x^2 \geq 0$?
 - Ints?
 - $50,000^2 = ???$
 - Largest Int32 is 2,147,483,647
 - Floats?
 - Yup!
- Is $4/5 = 4.0/5.0$?
 - Ints?
 - $4/5 = 0$
 - Floats?
 - $4.0/5.0 = 0.8$

Data Types

- How do we represent data in a computer?
 - Numbers
 - Text
 - Images
- Computers are made of transistors
 - Transistors can be ON or OFF or 1 or 0
- We have to turn 1's and 0's into something useful

Why Do Computers Use Binary? (Base 2)

- As humans we use base 10
 - 15, 250, 321, etc.
- Base 10 is hard to implement directly with electronics
- Binary (Base 2) is a lot easier
 - $8_{10} = 0000\ 1000_2$
 - Each digit in base 2 can be represented with 1 or 0
 - Perfect for ON / OFF nature of transistors!

Binary Basics

- Let's try:
 - $1011_2 = ???_{10}$
 - Each N^{th} digit aka bit (from the right) is either $d * 2^{N-1}$
 - Where d = value of digit (either 1 or 0)
 - Then add each digit in base 10
- $1011_2 = (1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0)_{10}$
- $1011_2 = (8 + 0 + 2 + 1)_{10}$
- $1011_2 = 11_{10}$

Binary Basics (Cont.)

- Let's try:
 - $1101\ 1110_2 = ???$
- $(1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0)_{10}$
- $(128 + 64 + 0 + 16 + 8 + 4 + 2 + 0)_{10}$
- $(222)_{10}$
- The range of numbers is:
 - $0 \leq i \leq 2^N - 1$
 - How can we add and subtract?
 - What about negative numbers???

Binary Basics (Cont.)

- Let's try:
 - $205_{10} = ???_2$
1. Find the largest binary digit that goes into the number
 - $128 = 2^7$ is the largest so let's start with 1 in the 8th binary digit
 2. Subtract 128 from the number
 - $205 - 128 = 77$
 3. Repeat steps 1 and 2 subtracting from 77, 13, etc. until you have 0
 - $205_{10} = 1100\ 1101_2$

Adding Binary Numbers

- Add numbers like usual
- If you have to add two 1's, carry the 1
- Let's try:
 - $0011 + 0101$

$$\begin{array}{r} 0011 \\ + 0101 \\ \hline 1000 \end{array}$$

Negative Binary Numbers

- With the previous examples negative numbers are impossible!
- We can make the leading bit aka sign bit either + or -
- Now our range is:
 - $-2^{N-1} + 1 < i < 2^{N-1} - 1$
- Can we add/subtract?
- Now we have +0 and -0

-4	10100
-3	10011
-2	10010
-1	10001
-0	10000
+0	00000
+1	00001
+2	00010
+3	00011
+4	00100

Two's Complement

- Two's Complement is a way of representing positive and negative binary numbers
- Given a binary number i , we can get $-i$ by:
 - Invert all bits in i
 - $0 \rightarrow 1$ and $1 \rightarrow 0$
 - Add 1 in binary

-16	10000
...	...
-3	11101
-2	11110
-1	11111
0	00000
+1	00001
+2	00010
+3	00011
...	...
+15	01111

Two's Complement (Cont.)

- Let's try:
 - 00010
- First invert
 - 11101
- Second add 00001
 - 11110
- If $X = 00010$, then $-X = 11110$

Two's Complement (Cont.)

- Our new range of numbers is:
 - $-2^{N-1} < i < 2^{N-1} - 1$
- Only one representation for 0 (no +0 or -0)
- More efficient use of bits

Adding numbers in 2's Complement

- Overflow can be a problem!
- Let's add 7 and 5:

$$\begin{array}{r} 0111 = 7 \\ + \underline{0101} = 5 \\ \hline 1100 = -4 ??? \end{array}$$

- If adding two positive numbers results in a negative number, an overflow has occurred and is invalid

Adding numbers in 2's Complement

- Overflow can be a problem!
- Let's add 7 and 5:

$$\begin{array}{r} 0111 = 7 \\ + \underline{0101} = 5 \\ \hline 1100 = -4 ??? \end{array}$$

- If adding two positive numbers results in a negative number, an overflow has occurred and is invalid

Hexadecimal Basics

- A 32-bit binary number looks like:
 - 0101 0011 1010 1001 1011 1011 0011 0111
- A simpler way to represent this is:
 - x53 A9 BB 37
- But how are these equal?

Hex Basics (Cont.)

- Each 4 bit binary number corresponds to a hex value
- So:
 - 0111 1101
 - x7D

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

Hex Basics (Cont.)

- To go from hex to decimal
- Use the same method as binary except use $d * 16^{N-1}$
- So: x4F =
 - $(4 * 16^{2-1} + 15 * 16^{1-1}) =$
 - $(4 * 16 + 15 * 1) =$
 - $(64 + 15) = 79$

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

Hex Basics (Cont.)

- To go from decimal to hex
- Use the same method as binary except using base 16
- So: 415 =
 - Highest number that fits into 415 is $1 \cdot 16^{3-1} = 256$
 - Subtract: $415 - 256 = 159$
 - Find highest number for N=2, $9 \cdot 16^{2-1} = 144$
 - Subtract: $159 - 144 = 15$
 - Find highest number for N=1 $15 \cdot 16^{1-1} = 15$
 - Combine all digits in the hex format:
 - x19F

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

Adding Numbers Becomes Easier

- Before:

$$\begin{array}{r} 0101\ 0011\ 1010\ 1001\ 1011\ 1011\ 0011\ 0111 \\ +\ 0011\ 1100\ 0010\ 1111\ 1010\ 0111\ 1001\ 1100 \end{array}$$

- After:

$$\begin{array}{r} x53\ A9\ BB\ 37 \\ +\ x3C\ 2F\ A7\ 9C \end{array}$$

x8F D9 62 D3

Strings

- Strings in C are arrays of characters
- Each character encoded in ASCII format
 - <https://www.asciitable.com/>
 - Must be null terminated “x00”
- A character is just a number
- A number is just bits
- How do we tell the difference between a string and a number?

```
char S[6] = "15213";
```

#49
#53
#50
#49
#51
#00

Dec

x31
x35
x32
x31
x33
x00

Hex

How Do You Know What's In Memory?

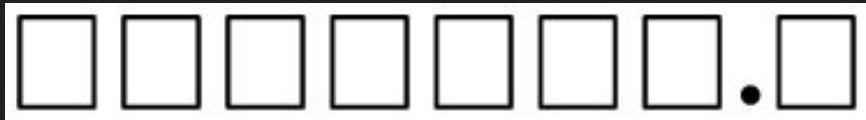
- If memory contains:
 - x4869206D6F6D2100
- Is it two integers?
- A 64-bit float?
- A string?
- You'll never know until:
 - You see the source code
 - You talk to the programmer
 - You reverse engineer it
- Memory registers contain BINARY ONLY

Real Numbers

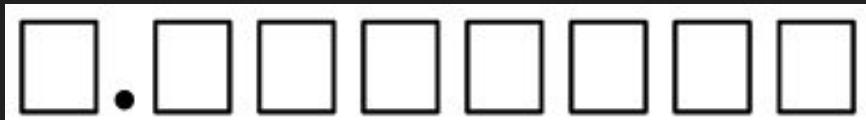
- Most numbers are not integers (0, 1, 25, 534)
 - Max int32 is about +/- 2 billion (Too small!)
- We need decimal precision
- Floating point, or scientific, notation allows for very large or very small numbers with as much precision as needed

Real Numbers (Cont.)

- Let's say we have 8 digits to represent a number:
- Would you prefer:
 - Low precision with large range?



- High precision with small range?



- We can “float” the decimal point to allow for both!

Scientific Notation

- In binary we have 0's and 1's
- In scientific notation we can change the decimal point:

$$425000 (\times 10^0) =$$

$$425 \times 10^3 =$$

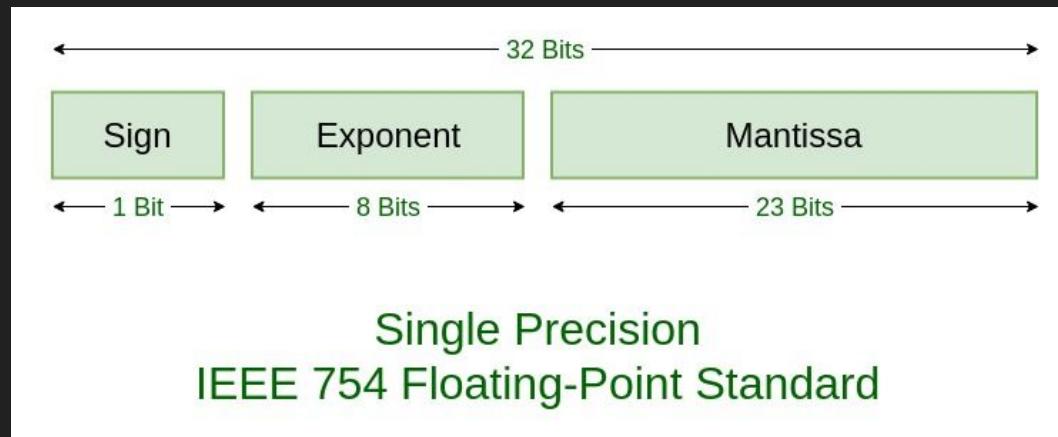
$$42.5 \times 10^4 =$$

$$4.25 \times 10^5$$

- We can use this to our advantage with binary numbers

IEEE-754 32-Bit Floating Point Standard

- IEEE-754 is a standard for floating point numbers
- For a 32 bit floating point number:



- Note: We have to bias the exponent!
- $N = (-1)^s \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 127)}$

IEEE-754 32-Bit Floating Point Standard (Cont.)

Let's try 25.75_{10} in floating point:

1. Convert both sides of decimal to binary:

- $25_{10} = 11001_2$
- $0.75_{10} = 0.11_2$

2. Combine:

- 11001.11_2

3. Move the decimal behind the left-most digit:

- 1.100111_2 (Keep track of how far you moved the decimal! (4 places))

4. Put all the bits on the right hand side of decimal in the 23-bit decimal bits (mantissa)

- $X \text{XXXX XXXX } \underline{1001 \ 1100 \ 0000 \ 0000}$
 $\underline{0000 \ 000}_2$

5. Add the amount of times you moved the decimal to the bias and convert to binary (+ for moving to left - for moving right)

- $127 + 4 = 131_{10}$
- $131_{10} = 1000\ 0011_2$

6. Put this binary number in the 8-bit exponent bits

- $X \underline{1000 \ 0011} \ 1001 \ 1100 \ 0000 \ 0000 \ 0000_2$

7. If number is positive make 1-bit sign bit 0, else make 1

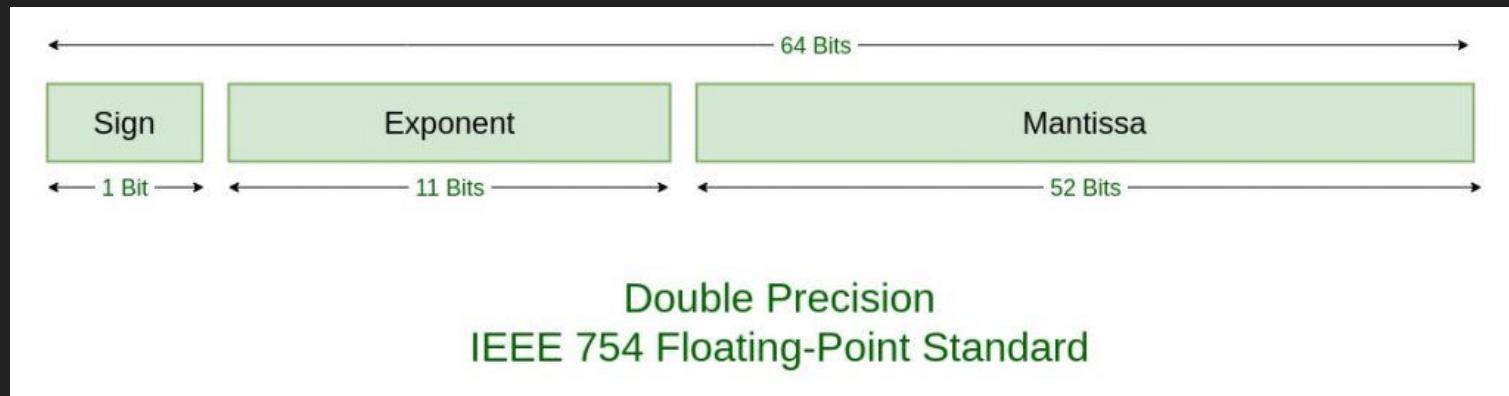
- $\underline{0} \ 1000 \ 0011 \ 1001 \ 1100 \ 0000 \ 0000 \ 0000_2 =$
 25.75_{10}

Struggling With Decimal To 32-Bit FP?

- <https://www.youtube.com/watch?v=8afbTaA-gOQ>

IEEE-754 64-Bit Floating Point Standard

- For a 64-bit floating point number:



- Note: We have to bias the exponent even more!!
- $N = (-1)^s \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 1023)}$

IEEE-754 64-Bit Floating Point Standard (Cont.)

- To convert from decimal to 64-bit floating point:
 - Follow same steps as 32-bit just add larger bias and more bits to mantissa (fraction part)
- To convert from floating point to decimal just use equations:
 - 32-bit: $N = (-1)^s \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 127)}$
 - 64-bit: $N = (-1)^s \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 1023)}$

Floating Point in C

- C has two float data types:
 - float - single precision
 - double - double precision
- Casting (converting between floats and ints)
 - double or float to int:
 - Truncates fraction
 - Rounds down (0.6 float becomes 0 int)
 - Not defined when out of range
 - int to double:
 - Exact conversion, if int has \leq 53 bit word size
 - int to float:
 - Will round according to round mode

Other Data Types

- BCD
- Bit vectors and bit masks
- Logic values (true or false)
- Instructions
- Media
 - Graphics
 - Sounds
 - Etc.

03 - The LC3 Instruction Set Architecture

CEG3310/5310 - Computer Organization
Max Gilson

LC3 Simulator

- Executable can be found here:
 - http://highered.mheducation.com/sites/0072467509/student_view0/lc-3_simulator.html
- In-browser version can be found here:
 - <https://wchargin.com/lc3web/>
- NOTE: You cannot save your work in the “in-browser” version, helpful for testing things but make sure you save your work on your computer somewhere!

LC3 Simulator Sample Code

```
.ORIG x3000 ; code start
```

```
START ; just a label
```

```
LEA R0, HI ; load the address of the "HI"
```

```
PUTS ; print hello world
```

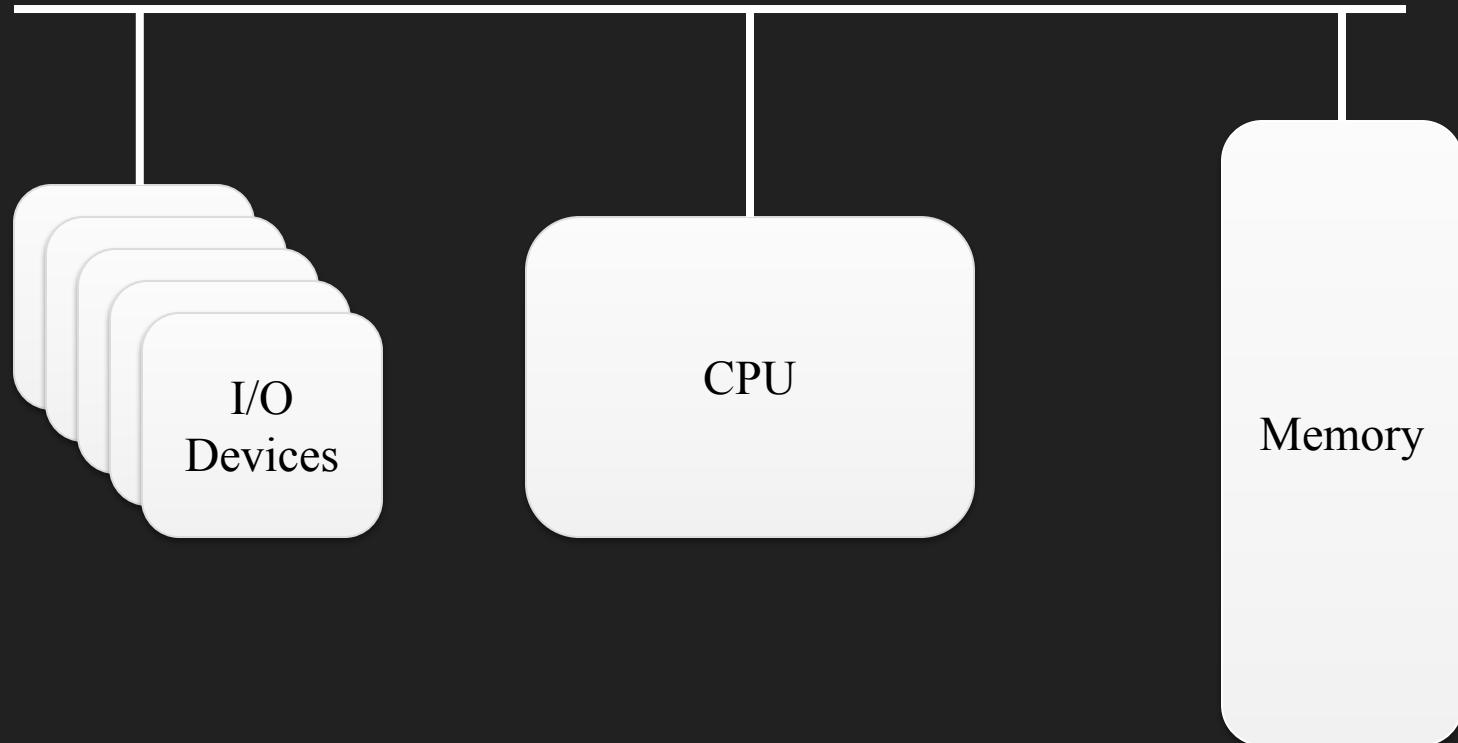
```
BR START ; jump back to start
```

```
HALT
```

```
HI .STRINGZ "Hello World!\n" ; your message
```

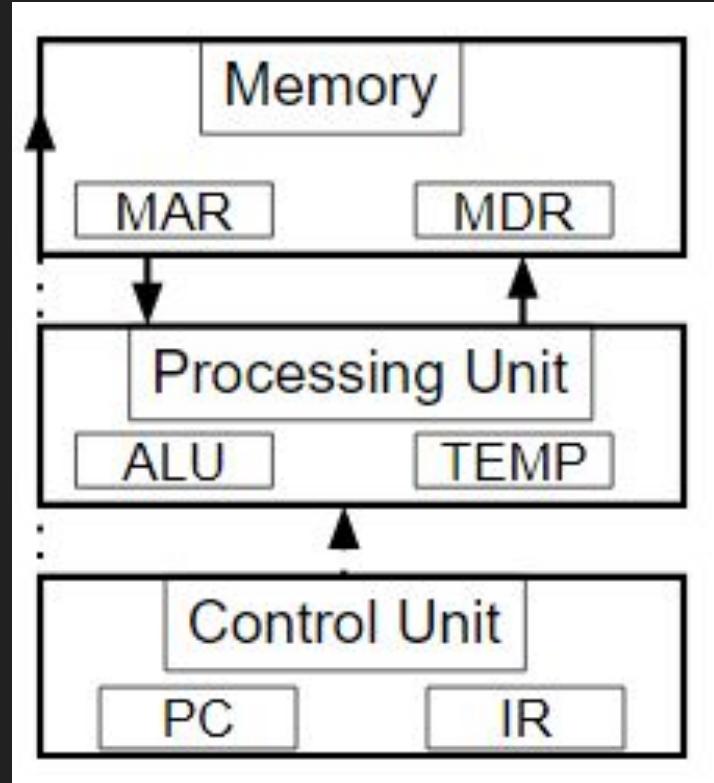
```
.END ; code end
```

Computers In A Nutshell



LC3 CPU Overview

- Memory
 - Holds data and instructions
- Processing Unit
 - Carries out instructions
- Control Unit
 - Sequences and interprets instructions



LC3 Instruction Set Architecture (ISA)

- 16-bit words and word addressable
 - This means each address of memory contains the same amount of memory as 1 word (16-bits)
 - Note: a word is the unit of data used by a specific processor
- All instructions are 16-bit
- All data are 16-bit words
 - 2's complement integer are the only native data type
- Eight 16-bit general purpose registers (GPRs) (R0 - R7)
- Three 1-bit status codes (N, Z, P)

LC3 Instructions (Cont.)

- Instructions:
 - Operate: Manipulate data directly
 - ADD, AND, NOT
 - Data Movement: Move data between memory and registers
 - LD, LDI, LDR, LEA, ST, STI, STR
 - Control: Change the sequence of instruction execution
 - BR, JMP/RET, JSR/JSSR, TRAP, RTI

LC3 Instructions (Cont.)

- Instructions are made of 16-bits
 - 4-bits for opcode
 - 12-bits for operands
- Opcode
 - Specifies the instruction type
- Operands
 - Specifies what the instruction acts on

LC3 Addressing Modes

- Five addressing modes:
 - Operands are located in:
 - Instruction (immediate a.k.a literal)
 - Registers
 - EA encoded in instruction (direct, or PC-relative)
 - Pointer to EA encoded in instruction (indirect)
 - Pointer to EA stored in register (base+offset, a.k.a. relative)
- Note: the effective address (EA) is the memory location of the operand
- Operate instructions only use immediate and register modes
- Data movement instructions use all five modes

LC3 Operate Instructions

- Performs arithmetic operations and manipulates data directly
 - ADD - adds two numbers
 - AND - bitwise boolean and
 - NOT - bitwise boolean not

LC3 Operate Instructions (ADD)

- ADD can have two possible formats:
 - 0001 DR SR1 0 00 SR2
 - 0001 DR SR1 1 IMM5
- 0001 is the opcode the specifies this is an ADD instruction
- DR is the destination register for the result of the addition (3-bits)
- SR1 is the first source register, or the source of the first number in the addition (3-bits)
- SR2 is the second source register, or the source of the second number in the addition (3-bits)
- IMM5 is a 5-bit signed binary number (5-bits)
- This instruction arithmetically adds two numbers
- Assembler Instruction: ADD DR, SR1, SR2
- Assembler Instruction: ADD DR, SR1, IMM5

LC3 Operate Instructions (ADD) (Cont.)

Example:

- If we have an instruction (hex): x1042
- We can convert to binary: 0001 000 001 0 00 010
- We can see:
 - This is an ADD instruction (0001 opcode)
 - The destination of the addition is R0 (000)
 - The first number in the addition is from R1 (001)
 - Two registers are being added (0 00)
 - The second number in the addition is from R2 (010)
- Assembler Instruction: ADD R0, R1, R2

LC3 Operate Instructions (ADD) (Cont.)

Example:

- If we have an instruction (hex): x142A
- We can convert to binary: 0001 010 000 1 01010
- We can see:
 - ADD instruction (0001 opcode)
 - Destination is R2 (010)
 - First number is from R0 (000)
 - 5-bit number is being added (1)
 - Second number is 10_{10} (01010)
- Assembler Instruction: ADD R2, R0, #10

LC3 Operate Instructions (AND)

- AND can have two possible formats:
 - 0101 DR SR1 0 00 SR2
 - 0101 DR SR1 1 IMM5
- 0101 is the opcode
- DR is the destination register (3-bits)
- SR1 is the first source register (3-bits)
- SR2 is the second source register (3-bits)
- IMM5 is a 5-bit signed binary number (5-bits)
- This instruction performs a bitwise AND on two numbers
- Assembler Instruction: AND DR, SR1, SR2
- Assembler Instruction: AND DR, SR1, IMM5

LC3 Operate Instructions (AND) (Cont.)

Example:

- If we have an instruction (hex): x5042
- We can convert to binary: 0101 000 001 0 00 010
- We can see:
 - AND instruction (0101 opcode)
 - Destination of the AND is R0 (000)
 - The first number in the AND is from R1 (001)
 - Two registers are being AND'd (0 00)
 - The second number in the ADD is from R2 (010)
- Assembler Instruction: AND R0, R1, R2

LC3 Operate Instructions (AND) (Cont.)

Example:

- If we have an instruction (hex): x542A
- We can convert to binary: 0101 010 000 1 01010
- We can see:
 - AND instruction (0101 opcode)
 - Destination of the AND is R2 (010)
 - The first number in the AND is from R0 (000)
 - An immediate 5-bit number is being AND'd (1)
 - The second number in the AND is 10_{10} (01010)
- Assembler Instruction: AND R2, R0, #10

LC3 Operate Instructions (NOT)

- NOT has one format:
 - 1001 DR SR1 1 11111
- 1001 is the opcode
- DR is the destination register (3-bits)
- SR1 is the first source register (3-bits)
- This instruction negates all the bits in SR1 and puts the result in DR
- Assembler Instruction: NOT DR, SR1

LC3 Operate Instructions (NOT) (Cont.)

Example:

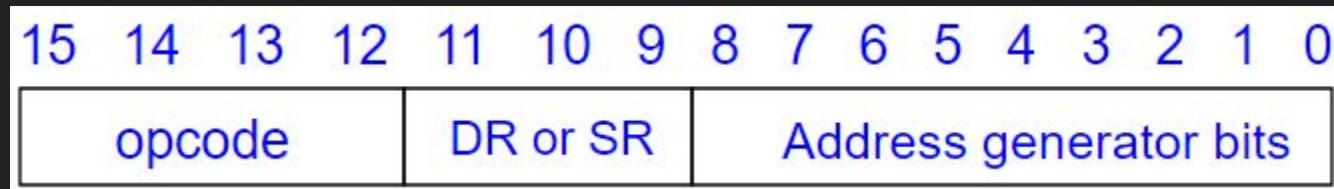
- If we have an instruction (hex): x907F
- We can convert to binary: 1001 000 001 1 11111
- We can see:
 - NOT instruction (1001 opcode)
 - The destination is R0 (000)
 - The number is from R1 (001)
- Assembler Instruction: NOT R0, R1

LC3 Processor Status Register (PSR)

- In the control unit, there is another register the Processor Status Register (PSR)
- Many instructions set the condition codes according to the result of the instruction
 - Z = result was zero
 - N = result was negative
 - P = result was positive

LC3 Data Movement Instructions

- We can move data between memory and registers
 - Register → Memory (Store)
 - Memory → Register (Load)
 - We cannot move data from memory to memory in LC3
 - We will talk about I/O devices later
- LC3 Load/Store Instructions:
 - LD, LDI, LDR, LEA, ST, STI, STR
 - Format:



LC3 Data Movement Instructions (Cont.)

- LD/ST
 - Direct/PC-Relative
- LDI/STI
 - Indirect
- LDR/STR
 - Base+Offset (Relative)
- LEA
 - Immediate
 - Does not access memory!

LC3 Movement Instructions (Load Direct) (LD)

- LD has the format:
 - 0010 DR PCoffset9
- 0010 is the opcode the specifies this is an LD instruction
- DR is the destination register for the result of the load (3-bits)
- PCoffset9 is how far we want to go into memory to retrieve a value (+256/-255 range)
- This instruction retrieves the data in the address from Instruction Address + 1 + PCoffset9
- Assembler Instruction: LD DR, LABEL

LC3 Movement Instructions (Store Direct) (ST)

- ST has the format:
 - 0011 SR PCoffset9
- 0011 is the opcode that specifies this is an ST instruction
- SR is the source register for the data that will be stored into memory (3-bits)
- PCoffset9 is how far we want to go into memory to retrieve a value (+-256 range)
- This instruction retrieves the data in the register SR and puts it in the address: Instruction Address + 1 + PCoffset9
- Assembler Instruction: ST SR, LABEL

LC3 Movement Instructions (Load Indirect) (LDI)

- LDI has the format:
 - 1010 DR PCoffset9
- 1010 is the opcode the specifies this is an LDI instruction
- DR is the destination register for the result of the load (3-bits)
- PCoffset9 is how far we want to go into memory to retrieve an address
- This instruction retrieves an ADDRESS at the address from Instruction Address + 1 + PCoffset9, then loads the data from the ADDRESS we retrieved
- Assembler Instruction: LDI DR, LABEL

LC3 Movement Instructions (Store Indirect) (STI)

- STI has the format:
 - 1011 SR PCoffset9
- 1011 is the opcode the specifies this is an STI instruction
- SR is the source register for the data that will be stored into memory (3-bits)
- PCoffset9 is how far we want to go into memory to retrieve an address
- This instruction retrieves the data in the register SR then retrieves the ADDRESS stored at:
Instruction Address + 1 + PCoffset9, and stores the data in SR into this address
- Assembler Instruction: STI SR, LABEL

LC3 Movement Instructions (Load Base+Index) (LDR)

- LDR has the format:
 - 0110 DR BaseR offset6
- 0110 is the opcode the specifies this is an LDR instruction
- DR is the destination register for the result of the load (3-bits)
- BaseR is the base register that we will add an offset to
- offset6 is how far we want to go into memory to retrieve data, applied as an offset to the value in BaseR
- This instruction loads data at the address BaseR+offset6 into DR
- Assembler Instruction: LDR DR, BaseR, offset6

LC3 Movement Instructions

(Store Base+Offset) (STR)

- STR has the format:
 - 0111 SR BaseR offset6
- 0111 is the opcode that specifies this is an STR instruction
- SR is the source register for the data that will be stored into memory (3-bits)
- BaseR is the base register that we will add an offset to
- offset6 is how far we want to go into memory to retrieve data, applied as an offset to the value in BaseR
- This instruction retrieves the data in the register SR stores it in the address BaseR+offset6
- Assembler Instruction: STR SR, BaseR, offset6

LC3 Movement Instructions

(Load Effective Address) (LEA)

- LEA has the format:
 - 1110 DR PCoffset9
- 1110 is the opcode that specifies this is an LEA instruction
- DR is the destination register for the result of the load (3-bits)
- PCoffset9 is how far we want to go into memory to retrieve an address
- This instruction loads an address into a register
- Assembler Instruction: LEA DR, LABEL

Extra help

- Operate Instructions:
 - <https://www.youtube.com/watch?v=yZChqRqPluI>
- Load Instructions:
 - <https://www.youtube.com/watch?v=cDaPPXyYbH0>
 - <https://www.youtube.com/watch?v=359TeV9UvM8>

LC3 Control Instructions

- Allows us to change the program counter
 - Conditionally or unconditionally
 - Store the original PC (subroutine) or not (goto)
- LC3 Control Instructions:
 - BRx
 - JMP/RET
 - JSR/JSRR
 - TRAP
 - RTI

LC3 Control Instructions

(Jump/GoTo) (JMP)

- JMP has the format:
 - 1100 000 BaseR 00 0000
- 1100 is the opcode the specifies this is an JMP instruction
- BaseR is the register that contains the value of the address we want to jump to (3-bits)
- PCoffset9 is how far we want to go into memory to retrieve an address
- This instruction “jumps” the PC to the address loaded into BaseR
- Assembler Instruction: JMP BaseR

LC3 Control Instructions

(Conditional Branch) (BRx)

- BRx has the format:
 - 0000 N Z P PCoffset9
- 0000 is the opcode the specifies this is an BR instruction
- N Z P are the NZP conditions required to branch to the PC+1+PCoffset9
- PCoffset9 is how far we want to branch into memory
- This instruction “branches” the PC to LABEL if NZP conditions are met
- Assembler Instruction: BRx LABEL
 - where x = n, z, p, nz, np, zp, or nzp

BRx Examples

- If statement:

if(R0 == 2)	x3000	ADD R0, R0, #-2
{	x3001	BRnp #1
R1 = R0 + R1;	x3002	ADD R1, R1, R0
}		

BRx Examples (Cont.)

- Do-while loop

do	x3000	ADD R1, R1, #1
{	x3001	ADD R0, R0, #-1
R1 = R1 + 1;	x3002	BRnp #-3
R0 --;		
}while(R0 != 0);		

BRx Examples (Cont.)

- Do-while loop

while(R0 != 0)	x3000	ADD R2, R0, R0
{	x3001	BRz #3
R1 = R1 + 1;	x3002	ADD R1, R1, #1
R0 --;	x3003	ADD R0, R0, #-1
}	x3004	BRnzp #-4

LC3 Control Instructions

(TRAP)

- TRAP has the format:
 - 1111 0000 trapvect8
- 1111 is the opcode the specifies this is an TRAP instruction
- trapvect8 is location of value to jump to in the vector table
- We will go into more detail later
- This instruction gets an PC address from a vector table and jumps to that address
- Assembler Instruction: TRAP trapvec

LC3 Register Operands Review

- Immediate
 - The number to be in an operation is stored in the instruction
- Register
 - Look at another register to perform an operation

LC3 Memory Addressing Review

- Direct Addressing
 - Operand location can be between approx. 256 locations from PC
- Indirect Addressing
 - Can access memory anywhere from x0000 to xFFFF
 - Memory must be accessed twice to do this
- Relative aka Base+Offset
 - Adds offset to a value in register to get desired address

LC3 Memory Addressing Review (Cont.)

- LD/ST/LDI/STI/LEA
 - $\text{PCoffset9} = \text{DesiredLocation} - \text{InstructionLocation} - 1$
- LDR/STR
 - $\text{offset6} = \text{DesiredLocation} - \text{BaseRegister}$

LC-3 Instruction Cheatsheet

x0	x4	x8	xC
BR _{nzp}	JSR	RTI	JMP
x1	x5	x9	xD
ADD	AND	NOT	X
x2	x6	xA	xE
LD	LDR	LDΣ	LEA
x3	x7	xB	xF
ST	STR	STI	TRAP

control

arithmetic

Loads: register <= memory

Stores: memory <= register

PC : Program counter, address where CPU
gets next instruction from

R0, R1, R2, R3 : general purpose registers

R4 : global variable pointer

R5 : stack frame (start of local variables)

R6 : stack pointer (top of stack)

R7 : subroutine return address

CC : Condition code, either N, Z, or P

16 bits

Arithmetic Instructions:

$$\text{ADD } R0, R1, \#7 \equiv R0 \leftarrow R1 + \#7$$

* constant ranges $[\#-16, 15]$, five bits

$$\text{ADD } R2, R1, R0 \equiv R2 \leftarrow R1 + R0$$

$$\text{AND } R3, R2, \#0 \equiv R3 \leftarrow R2 \text{ AND } \#0$$

$$\text{AND } R1, R2, R3 \equiv R1 \leftarrow R2 \text{ AND } R3$$

* bitwise AND

$$\text{NOT } R0, R1 \equiv R0 \leftarrow \text{NOT } (R1)$$

* invert all bits

Bitwise AND: R0 = x0F0F, R1 = x1234,
AND R0, R0, R1

$$\begin{array}{cccc} x0 & xF & x0 & xF \\ 0000 & 1111 & 0000 & 1111 \\ \text{AND} & \begin{array}{c} x1 \\ 0001 \end{array} & \begin{array}{c} x2 \\ 0010 \end{array} & \begin{array}{c} x3 \\ 0011 \end{array} & \begin{array}{c} x4 \\ 0100 \end{array} \\ \hline & 0000 & 0010 & 0000 & 0100 \end{array}$$

C = A AND B		
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

R0 \leftarrow x0Z04

Load Instructions

$LD R3, \#7 \equiv R3 \leftarrow MEM[PC + 1 + \#7]$

If PC is $x3000$, loads from $MEM[x3000 + 1 + \#7] : R3 \leftarrow MEM[x3008]$

$\Rightarrow PC_{offset}$ is 9 bits, range $[-256, 255]$

$LDI R2, \#3 \equiv R2 \leftarrow MEM[MEM[PC + 1 + \#3]]$

$\Rightarrow PC_{offset}$ is 9 bits. LDI does \geq memory reads.

$LDR R0, R6, \#7 \equiv R0 \leftarrow MEM[R6 + \#7]$

\Rightarrow offset is 6 bits, $[-32, 31]$

LDI Example:

address	data	
x3000	LDI R0, x99	L-PL
:	:	
x30A0	xABCD	R0 <= MEM[MEM[x3000+1+x99]]
:	:	R0 <= MEM[MEM[x30A0]]
xABCD	x1234	R0 <= MEM[xABCD]
		R0 <= x1234

LDI R0, x99 does the same as

LD R0, x99
LDR R0, R0, #0

Store Instructions

$ST \ R0, \ #7 \equiv MEM[PC+1+\#7]$

PC offset is 9 bits, range [-256, 255]

$STI \ R1, \ #9 \equiv MEM[MEM[PC+1+\#9]] \leftarrow R1$

PC offset is 9 bits, range [-256, 255]

$STR \ R1, R6, \ #1 \equiv MEM[R6+\#1] \leftarrow R1$

Offset is 6 bits, range is [-32, #31]

Control Instructions:

$\text{BRNZ } \#8 \equiv \text{if CC is N or Z, PC} \leftarrow \text{PC} + 1 + \#8$

$\pm \text{PC offset 9, } [\#-256, 255]$. Usually used with a label.

$\text{JSR } \#127 \equiv R7 \leftarrow \text{PC} + 1$
 $\text{PC} \leftarrow \text{PC} + 1 + \#127$

$\pm \text{PC offset 11, range } [\#-1024, \#1023]$

$\text{RTI} \equiv$ Returns from interrupt. Pops stack twice to backup PSR & PC.
Not super important for test.

TRAP

$\text{TRAP}_{x20} \equiv$ $R7 \leftarrow PC + 1$
 $PC \leftarrow \text{MEM}[x00 + x20]$

Trap offset is 8 bits. Trap vector table goes from
 $x0000$ to $x00FF$.

Built-in TRAPS

$x20$	GETC
$x21$	OUT
$x22$	PUTS
$x23$	IN
$x25$	HALT

04 - LC3 Assembly Language

CEG3310/5310 - Computer Organization
Max Gilson

Reality

- You have to know assembly
- You may never write a program in assembly
- Understanding assembly is needed for:
 - Bugs in high-level languages
 - Efficiency
 - Operating Systems
 - Real-time Applications
 - Reverse Engineering

Assembly Language

- Assembly language makes it possible to write machine language code
 - Each line of assembly language is translated into a single ML instruction (except pseudo-ops)
- An assembler makes it easy
 - Labels for symbolic names of addresses
 - Automatically converts binary/hex/decimal
 - Pseudo-ops (assembler directives)

Assembly Language Instructions

Pseudo-Ops

- Directives to the assembler
 - Not translated into ML instructions
- LC3 Pseudo-Ops:
 - .ORIG address
 - Tells assembler where to locate the program in memory (starting address)
 - .FILL value
 - Store value in the next location in memory
 - .BLKW n
 - Set aside a block of n number of words in memory
 - .STRINGZ string
 - Store the string, one character per word, in memory. Add a word of x0000 after the string.
 - .END
 - Marks the end of the source program (not to be confused with the instruction HALT!)
 - .EXTERNAL
 - The label so indicated is allocated in another module.

Sample Program - Multiply NUMBER by 6

```
;  
; Program to multiply an integer by the number 6  
;  
.ORIG      x3000  
LD    R1, SIX  
LD    R2, NUMBER  
AND   R3, R3, #0    ; clear R3 to hold the product  
;  
; The inner loop  
;  
AGAIN    ADD  R3, R3, R2  
ADD  R1, R1, #-1   ; keep track of iterations  
BRp  AGAIN  
;  
HALT  
;  
NUMBER .BLKW #1  
SIX    .FILL      x0006  
;  
.END
```

The Assembly Process

- Objective
 - Translate the AL (Assembly Language) program into ML (Machine Language)
 - Each AL instruction yields one ML instruction word
 - Interpret pseudo-ops correctly
- Problem
 - An instruction may reference a label
 - If the label hasn't been encountered yet, the assembler can't form the instruction word
- Solution
 - Two-pass assembly

Two-Pass Assembly

- Pass 1 - generate the symbol table
 - Scan each line
 - Keep track of current address
 - Increment by 1 for each instruction
 - Adjust as required for any pseudo-ops (e.g. .FILL or .STRINGZ, etc.)
 - For each label
 - Enter it into the symbol table
 - Allocate to it the current address
 - Stop when .END is encountered

Multiply NUMBER by 6 Example (Pass 1)

Symbol	Address
Again	x3003
Number	x3007
Six	x3008

```
; Program to multiply a number by six
;
; .ORIG x3000
x3000 LD R1, SIX
x3001 LD R2, NUMBER
x3002 AND R3, R3, #0
;
; The inner loop
;
; .AGAIN ADD R3, R3, R2
x3003 ADD R1, R1, #-1
x3004 BRp AGAIN
x3005 ;
;
; HALT
x3006 HALT
;
; NUMBER .BLKW 1
x3007 SIX .FILLx0006
;
; .END
```

Two-Pass Assembly (cont.)

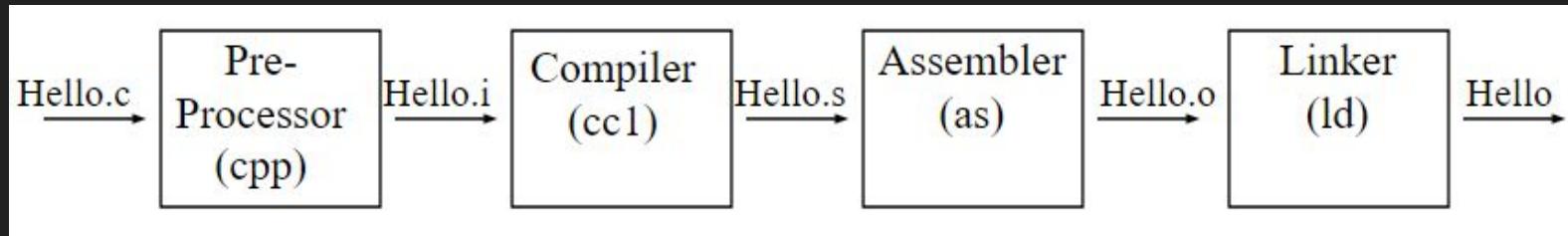
- Pass 2 - generate the Machine Language program
 - Scan each line again
 - Translate each AL instruction into ML
 - Look up symbols in the symbol table instruction
 - Ensure that labels are no more than +256 / -255 lines from instruction
 - Determine operand field for the instruction
 - Fill memory locations as directed by pseudo-ops
 - Stop when .END is encountered

Multiply NUMBER by 6 Example (Pass 2)

Symbol	Address
Again	x3003
Number	x3007
Six	x3008

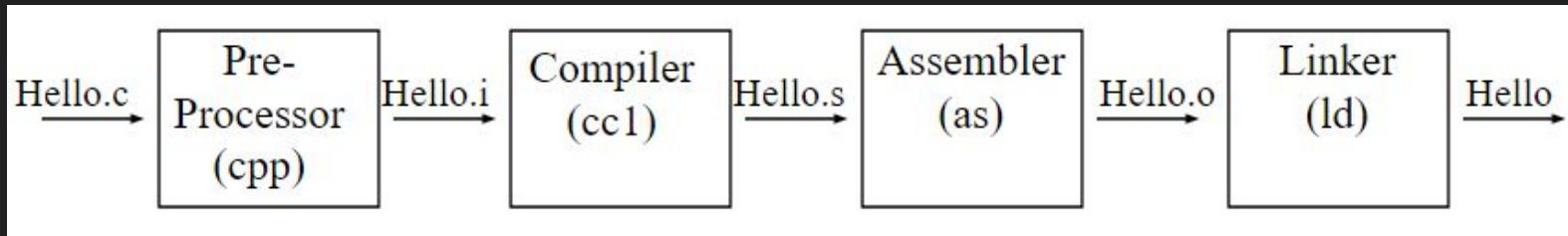
```
x3000  0010 0010 0000 0111 ; LD R1, SIX
x3001  0010 0100 0000 0101 ; LD R2, NUMBER
x3002  0101 0110 1110 0000 ; AND R3, R3, #0
x3003  0001 0110 1100 0010 ; ADD R3, R3, R2
x3004  0001 0010 0111 1111 ; ADD R1, R1, #-1
x3005  0000 0011 1111 1101 ; BRp AGAIN
x3006  1111 0000 0010 0101 ; HALT
x3007  0000 0000 0000 0000 ; .BLKW 1
x3008  0000 0000 0000 0110 ; .FILL x0006
```

From C to Executable



- Preprocessing phase: Modifies original program according to preprocessor directives (these start with the # character). The result is another C program text file (typically with the .i suffix)
 - `#include <stdio.h>`
 - `#define FALSE 0`
- Compilation phase: text file converted from high-level language to assembly.
 - Regardless of the original high-level language, all programs handled identically from this point onward
 - One assembly corresponds one-to-one with machine instructions

From C to Executable (Cont.)



- Assembly phase: assembly text file converted into machine language binary file and packaged into a relocatable object program.
- Linker phase: multiple object programs are merged to result in an executable object file.
 - For example: a standard library function such as `printf` might reside in a separate precompiled object file (like `printf.o`) that exists elsewhere in the system.

Object Files

- Each source file is translated into an object file
 - a list of ML instructions including the symbol table
- A complete program may include several source and/or object files:
 - Source files written in Assembly by the programmer
 - Library files provided by the system (OS or other)
 - Compiled HLL libraries
- The object files must be linked
 - One object file will be the “main”
 - All cross-referenced labels in symbol tables will be resolved

The executable file (the file you can run)

- The executable image (.exe file)
 - this is a file (“image”) of the finalized list of ML instructions, with all symbolic references resolved
 - it is loaded by copying the list into memory, starting at the address specified in the .ORIG directive
 - it is run by copying the starting address to the PC

05 - Subroutines and TRAPs

CEG3310/5310 - Computer Organization
Max Gilson

Trap Instructions

- The TRAP mechanism:
 - A set of trap service routines or TSRs (part of the CPU OS)
 - We have already seen the basic I/O SRs
 - A table of the starting addresses of these service routines
 - Located in a pre-defined block of memory ...
 - ... called the Trap Vector Table or System Control Block
 - In the LC-3: from x0000 to x00FF (only 5 currently in use)
 - The TRAP instruction
 - which loads the starting address of the TSR into the PC
 - Return link
 - from the end of the TSR back to the original program.

LC-3 TRAP Routines

- GETC (TRAP x20)
 - Read a single character from KBD.
 - Write ASCII code to R0[7:0], clear R0[15:8].
- OUT (TRAP x21)
 - Write R0[7:0] to the monitor.
- PUTS (TRAP x22)
 - Write a string to monitor (address of first character of string is in R0).
- IN (TRAP x23)
 - Print a prompt to the monitor and read a single character from KBD.
 - Write ASCII code to R0[7:0], clear R0[15:8], echo character to the monitor.
- HALT (TRAP x25)
 - Print message to monitor & halt execution.
- PUTSP (TRAP x24)
 - Print packed string to monitor (address in R0)

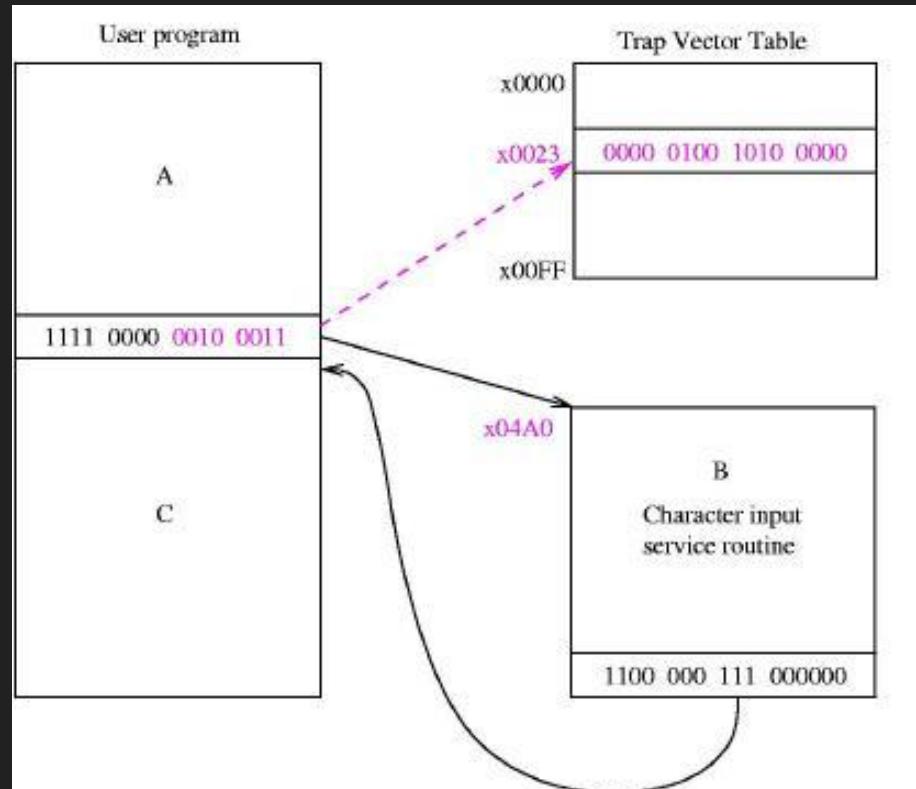
⋮	⋮
x0020	x0400
x0021	x0430
x0022	x0450
x0023	x04A0
x0024	x04E0
x0025	xFD70
⋮	⋮

Trap Instructions

- TRAP: A special instruction
 - A form of subroutine call used to invoke a service routine.
 - Format: 1111 000 trapvector8
 - $R7 \leftarrow (PC)$; the current PC is stored in R7
 - $PC \leftarrow \text{Mem}[\text{Zext}(IR[7:0])]$; the 8-bit trap vector is loaded to the PC
- RET – return instruction
 - The TSR ends with the RET instruction
 - $PC \leftarrow (R7)$; the program now picks up where it left off

Trap Instructions

- Trap Vector Table
 - In LC-3: 8 bits specify one of 256 locations (x0000 to x00FF)
 - The location contains the address of the TRAP service routine.
- TRAP & Interrupts
 - Similar mechanisms
 - A TRAP is an instruction (event internal to a program).
 - An interrupt is external to a program (from an I/O device)
 - Both invoke a supervisor service routine.



Character Output TSR (OUT)

```
01      .ORIG    X0430          ; System call starting address
02      ST R1, SaveR1        ; R1 will be used for polling
03
04      ; Write the character
05      TryWrite   LDI R1, DSR    ; Get status
06      BRzp      TryWrite      ; bit 15 = 1 => display ready
07      STI R0, DDR          ; Write character in R0
08
09      ; Return from TRAP
0A      LD R1, SaveR1        ; Restore registers
0B      RET                  ; Return (actually JMP R7)
0C      DSR     .FILL  xFE04    ; display status register
0D      DDR     .FILL  xFE06    ; display data register
0E      SaveR1   .BLKW    1
0F      .END
```

ALSO

```
11      .ORIG    x0021
12      .FILL    x0430
```

Program Halt TSR (HALT)

```
01      .ORIG XFD70          ; System call starting address
02      ST R0, SaveR0        ; Saves registers affected
03      ST R1, SaveR1        ; by routine
04      ST R7, SaveR7
05
06      ; Print message that machine is halting
07      LD R0, ASCII.NewLine
08      TRAP x21             ; Set cursor to new line
09      LEA R0, Message       ; Get start of message
0A      TRAP x22             ; and write it to monitor
0B      LD R0, ASCII.NewLine
0C      TRAP x21
0D
0E      ; Clear MCR[15] to stop the clock
0F      LDI R1, MCR          ; Load MC register to R1
10      LD R0, MASK          ; MASK = x7FFF (i.e. bit 15 = 0)
11      AND R0, R1, R0        ; Clear bit 15 of copy of MCR
12      STI R0, MCR          ; and load it back to MCR
```

Program Halt TSR (HALT) (Cont.)

```
13 ; Return from the HALT routine
14 ; (how can this ever happen, if the clock is stopped on line 12??)
15 ;
16     LD R7, SaveR7      ; Restores registers
17     LD R1, SaveR1      ; before returning
18     LD R0, SaveR0
19     RET                 ; JMP R7
1A
1B ; constants
1C     ASCIINewLine .FILL x000A
1D     SaveR0 .BLKW 1
1E     SaveR1 .BLKW 1
1F     SaveR7 .BLKW 1
20     Message .STRINGZ "Halting the machine"
21     MCR .FILL xFFFE
22     MASK .FILL x7FFF
23     .END
```

Saving and Restoring Registers

- Protect your values!
 - Any subroutine call may change values currently stored in a register.
- Sometimes the calling program (“caller”) knows what needs to be protected, so it saves the endangered register before calling the subroutine.
 - e.g. if you are using R0 for calculations, but you need to use GETC, you can save R0 to memory, so after you are done with GETC you can recover whatever was in R0
 - This is a “caller save”

Saving and Restoring Registers (Cont.)

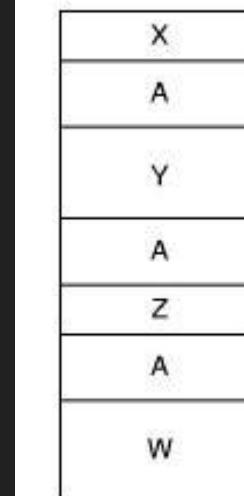
- Other times it will be the called program (“callee”) that knows what registers it will be using to carry out its task.
 - in the HALT routine, R0 and R1 are used as temporary working space to hold addresses, masks, ASCII values, etc., so they are both saved to memory at the start of the routine, and restored from memory before returning to the main program.
 - This is known as “callee save”
- This applies not only to trap service routines but to all subroutine calls, and is the basis of what are called “scope rules” in higher level languages.

Subroutines

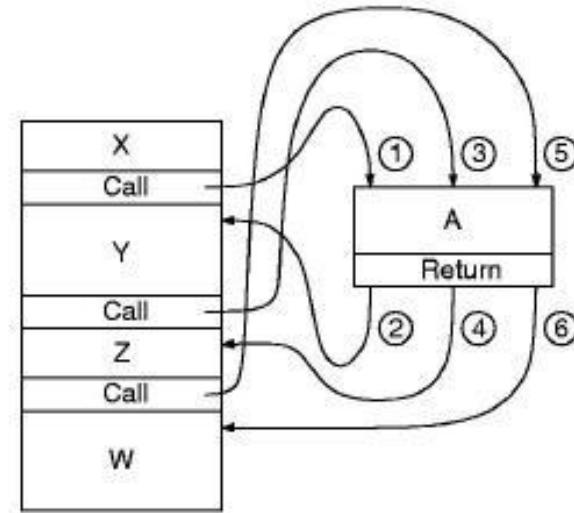
- Used for
 - Frequently executed code segments
 - Library routines
 - Team-developed systems
 - in other words, all the same reasons for using subroutines in higher level languages, where they may be called functions, procedures, methods, etc.
- Requirements:
 - Pass parameters and return values, via registers or memory.
 - Call from any point & return control to the same point.

The Call / Return System

- The figure illustrates the execution of a program comprising code fragments A, W, X, Y and Z.
 - Note that fragment A is repeated several times, and so is well suited for packaging as a subroutine:



(a) Without subroutines



(b) With subroutines

Jump to Subroutine: JSR / JSRR

- JSR: jump to subroutine (PC-Relative)
 - Format: 0010 1 address11
 - $R7 \leftarrow (PC)$ i.e. PC is saved to R7
 - $PC \leftarrow (PC) + \text{Sext(IR[10:0])}$ i.e PC-Relative addressing,
 - using 11 bits => label can be within +1024 / -1023 lines of JSR instruction
- JSRR: jump to subroutine (relative base+offset)
 - Format: 0010 0 BaseR 000000
 - $R7 \leftarrow (PC)$ i.e. PC is saved to R7
 - $PC \leftarrow (\text{BaseR})$ i.e Base+Offset addressing, with offset = 0
 - JSRR can jump anywhere in memory and BaseR can even be changed to point to different subroutines (function pointer)

Subroutine Call Example

; Calling program

Multiply two positive numbers

```
.ORIG x3000
LD R1, num1
LD R2, num2
JSR multi
ST R3, prod
HALT
```

; Input data & result

num1	.FILL x0006
num2	.FILL x0003
prod	.BLKW 1

```
multi    AND R3, R3, #0
        ADD R4, R1, #0
        BRz zero
loop     ADD R3, R2, R3
        ADD R1, R1, # -1
        BRp loop
zero    RET
        END
```

;Notice any undesirable ;side-effects?

06 - The C Programming Language

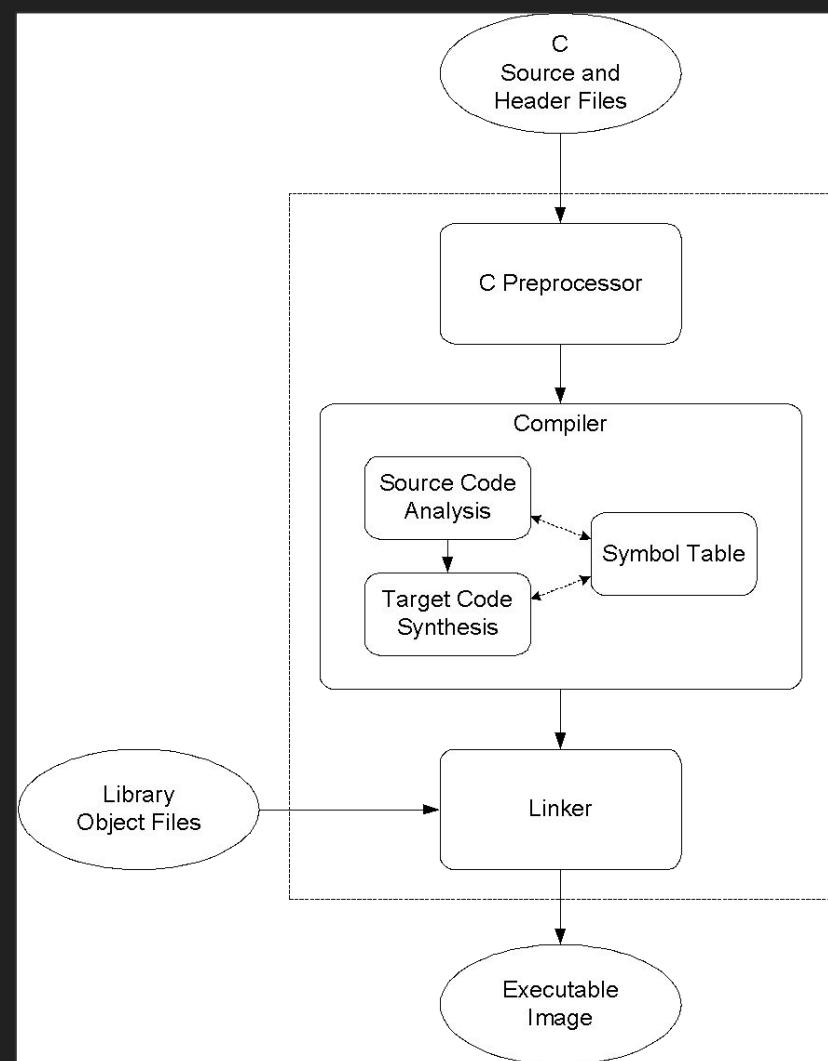
CEG3310/5310 - Computer Organization
Max Gilson

High Level Languages

- Gives symbolic names to values
 - don't need to know which register or memory location
- Provides abstraction of underlying hardware
 - operations do not depend on instruction set
 - example: can write “ $a = b * c$ ”, even though LC-3 doesn't have a multiply instruction
- Provides expressiveness
 - use meaningful symbols that convey meaning
 - simple expressions for common control patterns (if-then-else)
- Enhances code readability
- Safeguards against bugs
 - can enforce rules or conditions at compile-time or run-time
- If it can be specified in C then it MUST be possible to implement in assembly!

Compiling C Code

- Entire mechanism is usually called the “compiler”
- Preprocessor
 - macro substitution
 - conditional compilation
 - “source-level” transformations
 - output is still C
- Compiler
 - generates object file
 - machine instructions
- Linker
 - combine object files (including libraries)
 - into executable image



Compiler

- Source Code Analysis
 - “front end”
 - parses programs to identify its pieces
 - variables, expressions, statements, functions, etc.
 - depends on language (not on target machine)
- Code Generation
 - “back end”
 - generates machine code from analyzed source
 - may optimize machine code to make it run more efficiently
 - very dependent on target machine
- Symbol Table
 - map between symbolic names and items
 - like assembler, but more kinds of information
 - stores information like names and scopes of variables and functions, objects, classes, etc.

LC3 Compiler

- It is possible to convert C code to LC3 assembly
- The tools are decades old and probably won't work
- If you dare:
 - <http://users.ece.utexas.edu/~ryerraballi/ConLC3.html>

A Simple C Program

```
#include <stdio.h>
#define STOP 0

/* Function: main */
/* Description: counts down from user input to STOP */
main()
{
    /* variable declarations */
    int counter;      /* an integer to hold count values */
    int startPoint; /* starting point for countdown */

    printf("Enter a positive number: ");
    scanf("%d", &startPoint);

    /* output count down */
    for (counter=startPoint; counter >= STOP; counter--)
        printf("%d\n", counter);
}
```

- Try it out!
https://www.onlinegdb.com/online_c_compiler

Preprocessor Directives

- `#include <stdio.h>`
 - Before compiling, copy contents of header file (`stdio.h`) into source code.
 - Header files typically contain descriptions of functions and variables needed by the program.
 - no restrictions -- could be any C source code
- `#define STOP 0`
 - Before compiling, replace all instances of the string "STOP" with the string "0"
 - Called a macro
 - Used for values that won't change during execution, but might change if the program is reused. (Must recompile.)
- Every C program must have exactly one function called `main()`.
 - Be careful with what you `#include`!
 - `main()` determines the initial PC.

What about Boolean?

- C programs generally use char or int for boolean.
- 0 = false, any other value = true
- Suppose you have:
 - `int a = 7;`
 - `int b = 0;`
- `(a && b)` will return 0
- `(a || b)` will return a nonzero value (generally 1)

Data type of result

- Addition/Subtraction: If mixed types, smaller type is "promoted" to larger.
 - $x + 4.3$ answer will be float
- Division: If mixed type, the default result is a truncated signed integer
 - For int $x = 5$: $(x / 3 == 1)$ is true! Not 1.6! Not 2! 1!
 - For float $f = 5$: $(f / 3 == 1)$ is false!
 - For int $x = 5$: $((float)x / 3 == 1)$ is false!
- The rules can be overridden by typecasting the operands or result
 - the compiler does this for you automatically to match the destination type
 - $\text{int } i = 2.5 / 3$ is 0
 - $\text{float } f = 2.5 / 3$ is 0.833333 [Note automatic typecasting of 3]
- Without typecasting you are stuck with the limitations of the data type the compiler assigned for the storage/calculation of your intermediate value

Size of Primitive Data Types

Data Type	Size in Bytes	Size in Bits
int	4	32
float	4	32
double	8	64
char	1	8
pointer	8	64

Pointers

```
int i;  
float f;  
int * p;      // p is a pointer to an int  
float * q;    // q is a pointer to a float  
  
p = &i;      // p points to i  
q = &f;      // q points to f  
  
*p = 20;     // dereference p  
*q = 7.5;    // dereference q
```

Arrays

```
#define ARRSIZE 10

int a[ARRSIZE];          // a holds 10 integers
int *p;                  // p is a pointer to an integer
int i;                   // loop counter

for(i=0; i<ARRSIZE; i++) {
    a[i] = 2 * i;
}

p=a;      // The name of the array works like a pointer!
p[3] = 5;  // I can use the pointer like an array!

for(i=0; i<ARRSIZE; i++) {
    printf("a[%d] = %d\n", i, a[i]);
}
```

Strings in C

- C is just a step above assembly language. Strings in C are implemented exactly as they are in assembly: an array of characters ending with zero
 - “Null-terminated string”
- No special type is necessary, just a pointer to char or an array of char (which are really the same thing)

Structs

- The C programming language is not object oriented
 - There are no classes, objects, or methods
- Structs are a way to combine many variables of different types in one place
 - Structs can be thought of as arrays with multiple data types

Structs

```
#include <stdio.h>
#include <string.h>

struct myStructure {
    int a;
    float b;
    char c[30];
};

int main() {
    // Create a structure variable and assign values to it
    struct myStructure s1 = {13, 0.125, "Some text"};

    // Modify values
    s1.a = 30;
    s1.b = -1.25f;
    strcpy(s1.c, "Something else");

    // Print values
    printf("%d %f %s", s1.a, s1.b, s1.c);

    return 0;
}
```

Operations on C “strings”

- `strcmp(str1, str2)`
 - `strncmp`
- `strcpy(dest, src)`
 - `strncpy`
- `strlen(str)`
- `strncat(dest, src)`
- `strcasecmp, strncasecmp`

Functions

- You must create a prototype first (can be done in a .h file)
- Parameters are passed by value “always”
- You can get around this by passing a pointer and dereferencing it in the function

Output with printf

- Variety of I/O functions in C Standard Library.
 - Must include <stdio.h> to use them.
- printf: Can print arbitrary expressions, including formatted variables

```
printf("%d\n", startPoint - counter);
```
- Print multiple expressions with a single statement

```
printf("%d %d\n", counter, startPoint - counter);
```
- Different formatting options:
 - %d decimal integer
 - %x hexadecimal integer
 - %c ASCII character
 - %f floating-point number

Examples of Output

- This code:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43+59);
printf("43 plus 59 in hex is %x.\n", 43+59);
printf("43 plus 59 as a character is %c.\n", 43+59);
```

- Produces this output:

```
43 is a prime number.
43 + 59 in decimal is 102.
43 + 59 in hex is 66.
43 + 59 as a character is f.
```

Input with scanf

- Many of the same formatting characters are available for user input.

```
scanf( "%c" , &nextChar );
```

- reads a single character and stores it in nextChar

```
scanf( "%f" , &radius );
```

- reads a floating point number and stores it in radius

```
scanf( "%d %d" , &length , &width );
```

- reads two decimal integers (separated by whitespace), stores the first one in length and the second in width

- Must use address-of operator (&) for variables being modified.
- We'll revisit pass by reference/value in a future lecture

Operators

- Programmers manipulate variables using the operators provided by the high-level language.
- You need to know what these operators assume
 - Function
 - Precedence & Associativity
 - Data type of result
- You are assumed to know all standard C/C++ operators, including bitwise ops:

Symbol	Operation	Usage
<code>~</code>	bitwise NOT	<code>~x</code>
<code><<</code>	left shift	<code>x << y</code>
<code>>></code>	right shift	<code>x >> y</code>
<code>&</code>	bitwise AND	<code>x & y</code>
<code>^</code>	bitwise XOR	<code>x ^ y</code>

Control Structures

- If it can be done in “C” it must be able to be done in assembly
- Conditionals
 - making a decision about which code to execute, based on evaluated expression
 - If
 - If-else
 - Switch
- Iteration
 - executing code multiple times, ending based on evaluated expression
 - While
 - For
 - do-while

Variable Scope

- Where is the variable accessible?
- All C variables are defined as being in one of two storage classes
 - Automatic storage class (on the stack, uninitialized)
 - Static storage class (in memory, initialized to 0)
- Compiler infers scope from where variable is declared unless specified
 - programmer doesn't have to explicitly state (but can!)
 - automatic int x;
 - static int y;
 - Does not get destroyed when function ends!
- Global: accessed anywhere in program (default static)
 - Global variable is declared outside all blocks
- Local: only accessible in a particular region (default automatic)
 - Variable is local to the block in which it is declared
 - block defined by open and closed braces { }
 - can access variable declared in any "containing" block

Variable Scope Example

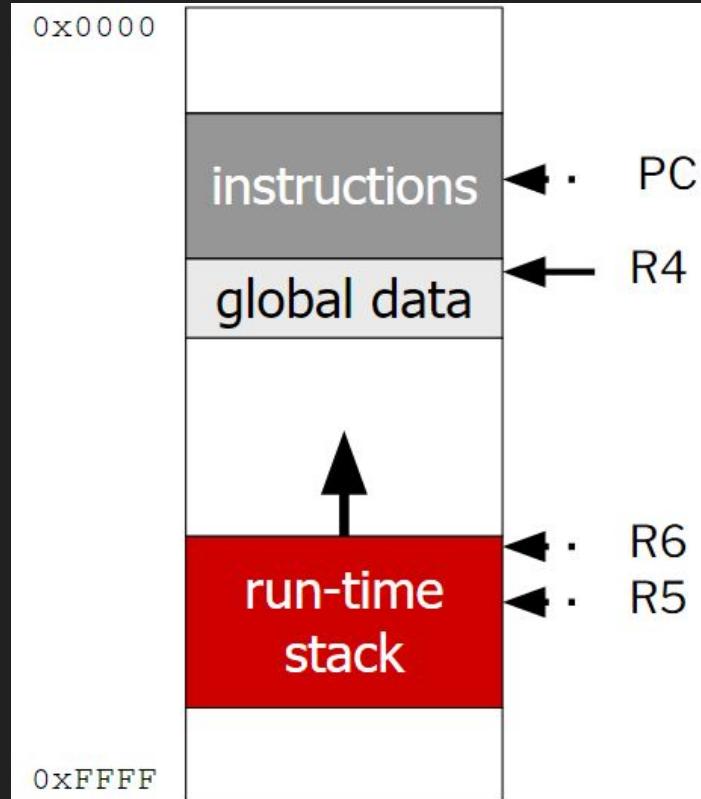
```
#include <stdio.h>
int myNumber0 = 10; // Global variable

main()
{
    int myNumber1 = 15; // Local to main
    printf("%d\n", myNumber0);
    printf("%d\n", myNumber1);
    otherFunction();
}

otherFunction()
{
    int myNumber2 = 20; // Local to otherFunction()
    printf("%d\n", myNumber2);
}
```

Runtime Stack Intro

- Global data section
 - All global variables stored here
 - (actually all static variables)
 - R4 points to beginning (global pointer)
- Run-time stack
 - Used for local variables
 - R6 points to top of stack (stack pointer)
 - R5 points to top frame on stack (frame pointer)
 - New frame for each block (goes away when block exited)
 - Offset = distance from beginning of storage area



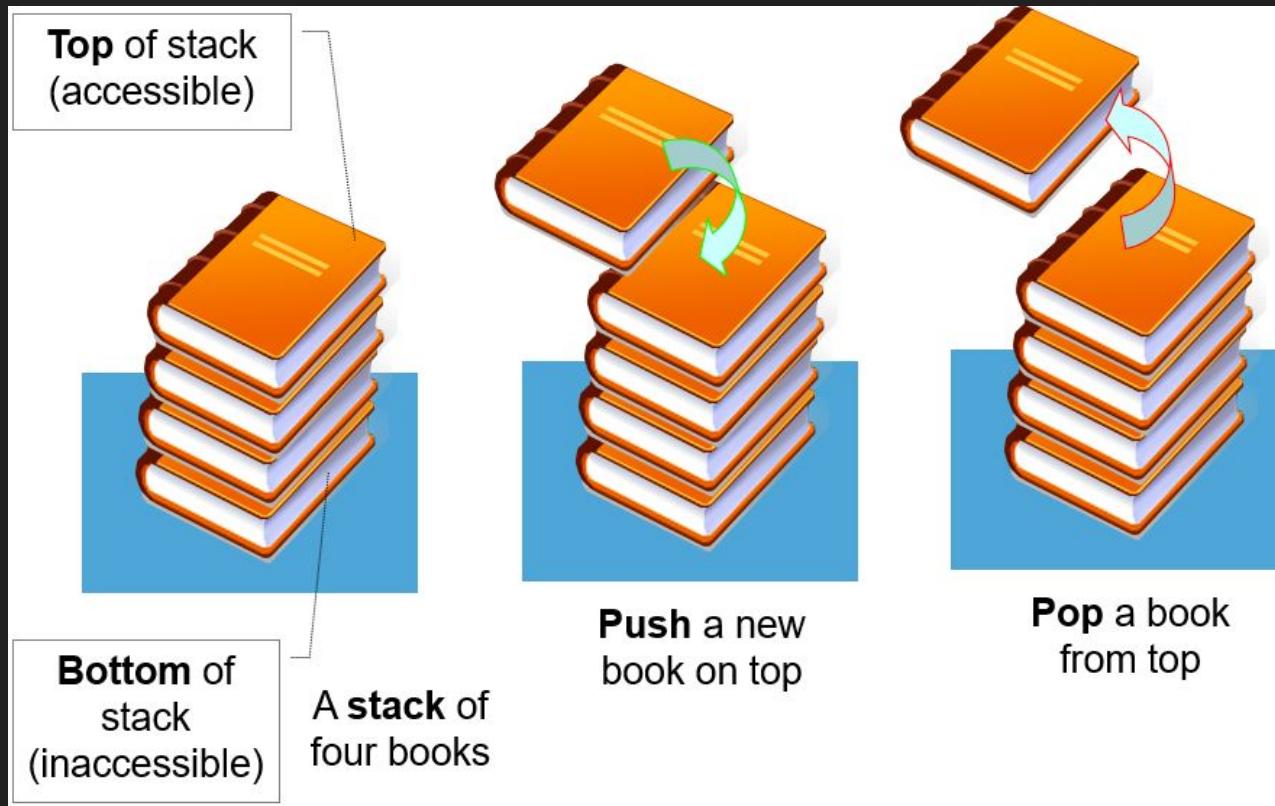
07 - The Runtime Stack

CEG3310/5310 - Computer Organization
Max Gilson

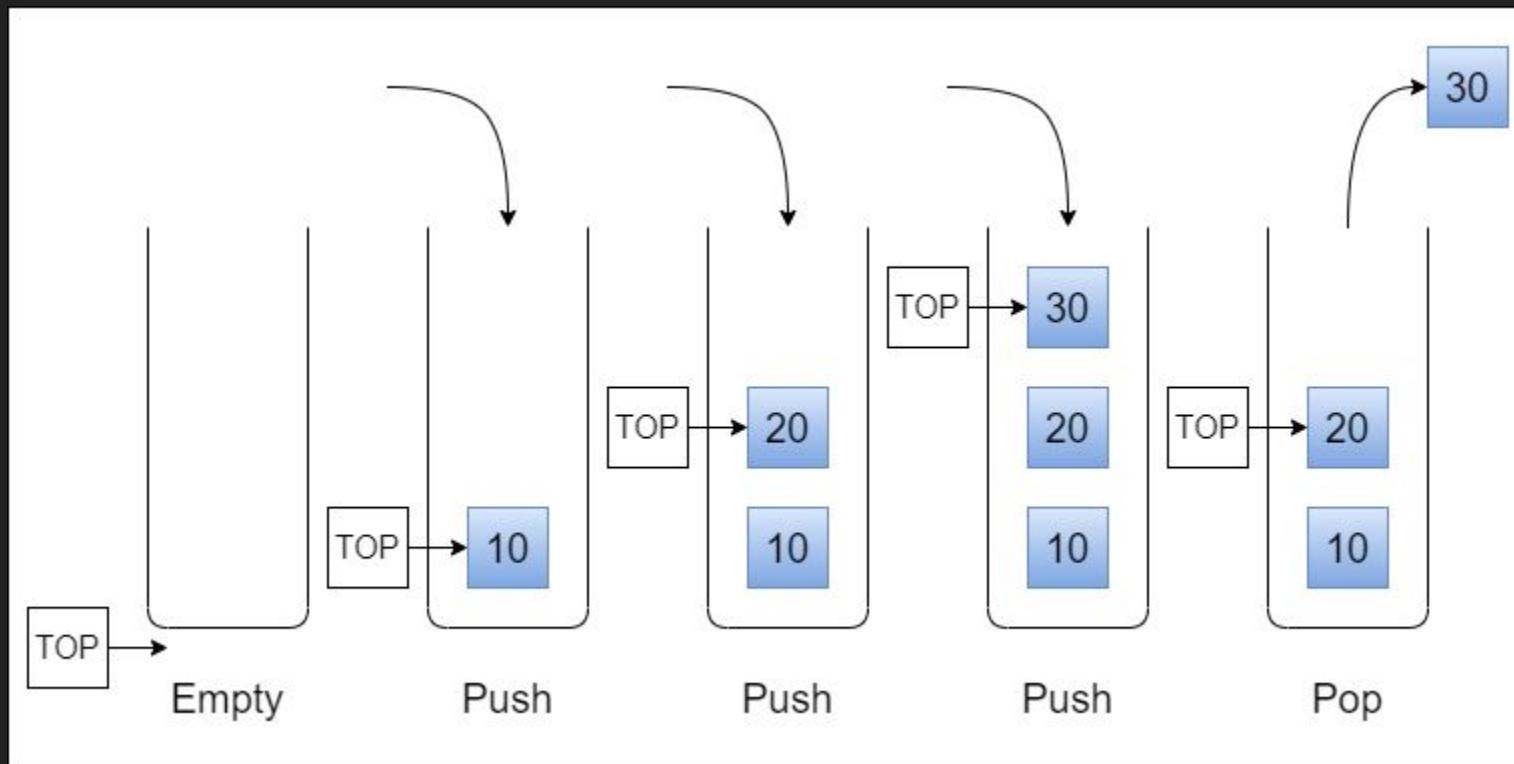
Stacks

- Stacks are a data structure that work on the principle of LIFO (Last In, First Out)
- Operations:
 - Push - Add an element to the stack on the top
 - Pop - Remove an element from the stack off the top
- Errors:
 - Stack Underflow - trying to pop from an empty stack
 - Stack Overflow - trying to push to a full stack
- To implement a stack, all you need is a reference to the top element

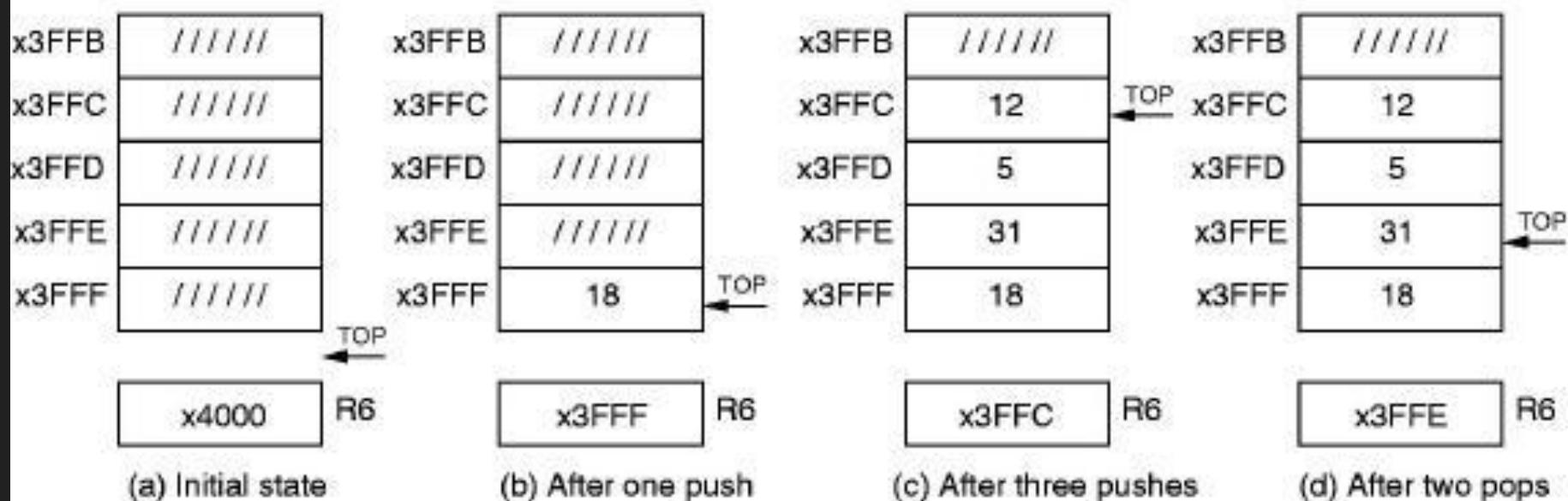
Stacks Visualized



Stacks Visualized (Cont.)



Stacks in Memory



The Runtime Stack

- The Runtime Stack allows gives us a place to store our variables
- There are two components of the runtime stack:
 - Stack Pointer
 - Use R6 for this
 - Keeps track of the TOP of our stack
 - Frame Pointer
 - Use R5 for this
 - Keeps track of the BOTTOM of the current frame

Stack Pointer

- The stack pointer always points to the TOP of your stack
- Below this area in memory you store (in this order):
 - Inputs to called function (if any)
 - Local variables
 - Saved Registers
 - Return Address
 - Return Values
- When you enter into a new function/subroutine, you will move your stack pointer to the top of this new area

Frame Pointer

- The frame pointer always points to the BOTTOM of your stack for a current frame
 - The “bottom” is where your local variables start

Global Variable Pointer

- The global variable pointer always points to the top of your global variables section that follows the last instruction in your main section of instructions
 - For example:

x3000 LD R0, x3003

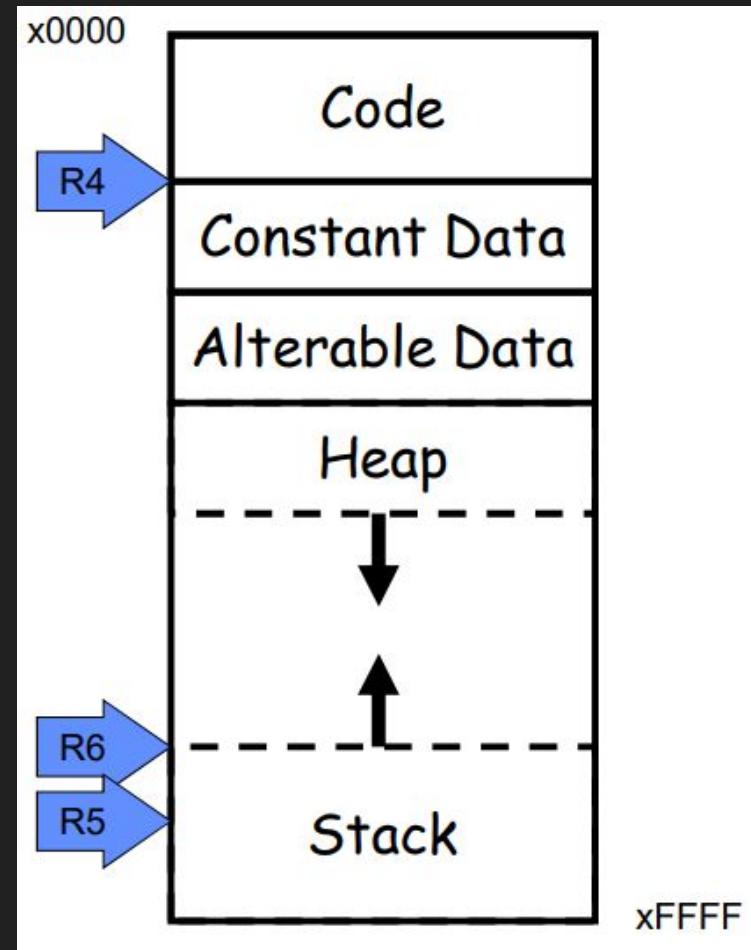
x3001 ADD R1, R0, #5

x3002 HALT

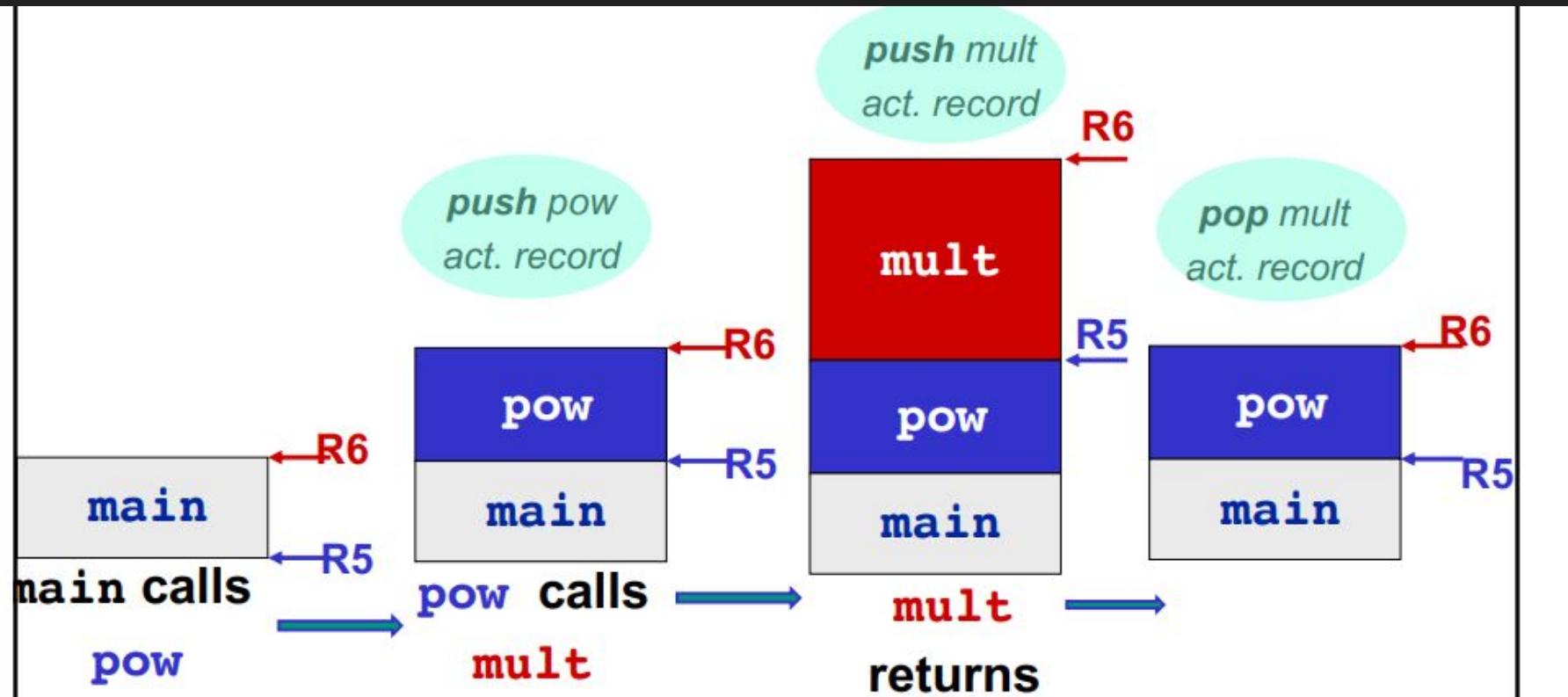
x3003 .FILL x1234 <- R4

C Programs in Memory

- R4
 - points to your constants and globals
- R5
 - points to your current function's local variables
- R6
 - points to the end of the stack



Stack and Frame Pointer Example (pow)



C Functions in The Stack

- Three things must happen to execute a function:
 - Pass inputs to a function (if any)
 - Complete the task
 - Return output from function (if any)
- Functions must be caller independent
 - This means, any function should be able to call another function without issue
- The Activation Record
 - When a function is called it must save/have access to:
 - Inputs from caller function
 - Return Value
 - Return Address
 - Saved Registers
 - Local Variables
 - Inputs to other functions

Stack and Frame Pointer Example (max)

```
int main()
{
    int x, y, val;
    x = 10;
    y = 11;
    val = max(x + 10, y);
    return val;
}

int max(int a, int b)
{
    int result;
    result = a;
    if (b > a)
    {
        result = b;
    }
    return result;
}
```

Start of main()					
Address	Data	Notes	Stack Pointer	Frame Pointer	Frame
x5FF3	x#####				
x5FF4	x#####				
x5FF5	x#####				
x5FF6	x#####				
x5FF7	x#####				
x5FF8	x#####				
x5FF9	x#####				
x5FFA	x#####				
x5FFB	x#####				
x5FFC	x#####				
x5FFD	x#####				
x5FFE	x#####				
x5FFF	x#####				
x6000	x#####	main's return value	<- R6		int main()

Stack and Frame Pointer Example (max)

```
int main()
{
    int x, y, val;
    x = 10;
    y = 11; ←
    val = max(x + 10, y);
    return val;
}

int max(int a, int b)
{
    int result;
    result = a;
    if (b > a)
    {
        result = b;
    }
    return result;
}
```

Before calling max(x + 10, y)					
Address	Data	Notes	Stack Pointer	Frame Pointer	Frame
x5FF3	x#####				
x5FF4	x#####				
x5FF5	x#####				
x5FF6	x#####				
x5FF7	x#####				
x5FF8	x#####				
x5FF9	x#####				
x5FFA	x#####				
x5FFB	x#####	int val	<- R6		
x5FFC	x000B	int y			
x5FFD	x000A	int x			
x5FFE	x#####	previous frame pointer (R5)			
x5FFF	x#####	main's return address (R7)			
x6000	x#####	main's return value			

Stack and Frame Pointer Example (max)

```
int main()
{
    int x, y, val;
    x = 10;
    y = 11;
    val = max(x + 10, y);
    return val;
}

int max(int a, int b)
{ ←
    int result;
    result = a;
    if (b > a)
    {
        result = b;
    }
    return result;
}
```

Start of max					
Address	Data	Notes	Stack Pointer	Frame Pointer	Frame
x5FF3	x#####				
x5FF4	x#####				
x5FF5	x#####				
x5FF6	x#####				
x5FF7	x#####				
x5FF8	x#####				
x5FF9	x000B	int b	<- R6		int max(int a, int b)
x5FFA	x0014	int a			
x5FFB	x#####	int val			
x5FFC	x000B	int y			
x5FFD	x000A	int x			
x5FFE	x#####	previous frame pointer (R5)			int main()
x5FFF	x#####	main's return address (R7)			
x6000	x#####	main's return value			

Stack and Frame Pointer Example (max)

```
int main()
{
    int x, y, val;
    x = 10;
    y = 11;
    val = max(x + 10, y);
    return val;
}

int max(int a, int b)
{
    int result;
    result = a;
    if (b > a)
    {
        result = b;
    } ←
    return result;
}
```

Before returning from max(x + 10, y)					
Address	Data	Notes	Stack Pointer	Frame Pointer	Frame
x5FF3	x0014	int result	<- R6		
x5FF4	x#####	save R1			
x5FF5	x#####	save R0			
x5FF6	x#####	previous frame pointer (R5)		<- R5	
x5FF7	x#####	max's return address (R7)			int max(int a, int b)
x5FF8	x#####	max's return value			
x5FF9	x000B	int b			
x5FFA	x0014	int a			
x5FFB	x0014	int val			
x5FFC	x000B	int y			
x5FFD	x000A	int x			
x5FFE	x#####	previous frame pointer (R5)			int main()
x5FFF	x#####	main's return address (R7)			
x6000	x#####	main's return value			

Stack and Frame Pointer Example (max)

```
int main()
{
    int x, y, val;
    x = 10;
    y = 11;
    val = max(x + 10, y); ←
    return val;
}

int max(int a, int b)
{
    int result;
    result = a;
    if (b > a)
    {
        result = b;
    }
    return result;
}
```

After returning from max(x + 10, y)					
Address	Data	Notes	Stack Pointer	Frame Pointer	Frame
x5FF3	x0014	int result			
x5FF4	x#####	save R1			
x5FF5	x#####	save R0			
x5FF6	x#####	previous frame pointer (R5)			
x5FF7	x#####	max's return address (R7)			int max(int a, int b)
x5FF8	x0014	max's return value	<- R6		
x5FF9	x000B	int b			
x5FFA	x0014	int a			
x5FFB	x0014	int val			
x5FFC	x000B	int y			
x5FFD	x000A	int x			
x5FFE	x#####	previous frame pointer (R5)			
x5FFF	x#####	main's return address (R7)			
x6000	x#####	main's return value			

Stack and Frame Pointer Example (max)

```
int main()
{
    int x, y, val;
    x = 10;
    y = 11;
    val = max(x + 10, y);
    return val; ←
}

int max(int a, int b)
{
    int result;
    result = a;
    if (b > a)
    {
        result = b;
    }
    return result;
}
```

After returning from main					
Address	Data	Notes	Stack Pointer	Frame Pointer	Frame
x5FF3	x0014	int result			int max(int a, int b)
x5FF4	x#####	save R1			
x5FF5	x#####	save R0			
x5FF6	x#####	previous frame pointer (R5)			
x5FF7	x#####	max's return address (R7)			
x5FF8	x0014	max's return value			
x5FF9	x000B	int b			
x5FFA	x0014	int a			
x5FFB	x0014	int val			
x5FFC	x000B	int y			
x5FFD	x000A	int x			
x5FFE	x#####	previous frame pointer (R5)			
x5FFF	x#####	main's return address (R7)			
x6000	x0014	main's return value	<- R6		int main()

Stack and Frame Pointer Example

```
#include <stdio.h>

main()
{
    int x = 3;
    int y = 8;
    int z;

    z = multiply(x, y);

}

int add(int a, int b)
{
    int sum = 0;
    sum = a + b;
    printf("Sum: %d\n", sum);
    return a + b;
}
```

```
int multiply(int c, int d)
{
    int product = 0;
    int count = 0
    while(count < d)
    {
        product = add(c, product);
        count = count + 1;
    }
    printf("Product: %d\n", product);
    return product;
}
```

Writing Good Subroutines

- Generality – can be called with any arguments
 - Passing arguments on the stack does this.
- Transparency – you have to leave the registers like you found them, except R6.
 - Registers must be callee saved.
- Readability – well documented.
- Re-entrant – subroutine can call itself if necessary
 - Store all information relevant to specific execution to non-fixed memory locations
 - The stack!
 - This includes temporary callee storage of register values!

08 - Midterm Study Guide

CEG3310/5310 - Computer Organization
Max Gilson

Midterm

- 14 questions total - 25% of final class grade
- True/False
 - 5 questions
 - 0.5 points each
 - 2.5 points total (2.5% of final class grade)
- Multiple choice
 - 5 questions
 - 0.5 points each
 - 2.5 points total (2.5% of final class grade)
- Short answer
 - 2 questions
 - 4.0 points each
 - 8.0 points total (8.0% of final class grade)
 - Must be graded manually (0 points initially)
- Coding
 - 2 questions
 - 6.0 points each
 - 12.0 points total (12.0% of final class grade)
 - Must be graded manually (0 points initially)

02 - Bits, Data Types and Operations

- How is data represented?
 - Convert between binary/decimal/hex
 - Adding numbers in binary
 - ASCII and strings
- Two's Complement
- Floats

03 - The LC3 Instruction Set Architecture

- Instructions
 - Operate instructions
 - Data movement instructions
 - Control instructions
- Opcodes
- Operands
- PCoffset
- Addressing

04 - LC3 Assembly Language

- Pseudo-Ops
- Labels
- Opcodes
- C code to Executable

05 - Subroutines and TRAPs

- TRAP calls
- TRAP vector table
- TRAP instructions / RET instruction
- Subroutines
- JSR / RET

06 - The C Programming Language

- High level languages
- Compiling C Code
- Pointers
- Arrays
- Function calls
- Scope

07 - The Runtime Stack

- Stacks
- Stack Pointer
- Frame Pointer
- Global Variable Pointer
- Runtime Stack construction

08 - Inputs and Outputs (I/O)

CEG3310/5310 - Computer Organization
Max Gilson

I/O Basics

- Input:
 - transfer data from the outside world to the computer:
 - keyboard, mouse, scanner, bar-code reader, etc.
- Output
 - transfer data from the computer to the outside:
 - monitor, printer, LED display, etc.
- Peripheral: any I/O device, including hard drives / SSDs
- LC-3 supports only a keyboard and a monitor

Device Registers

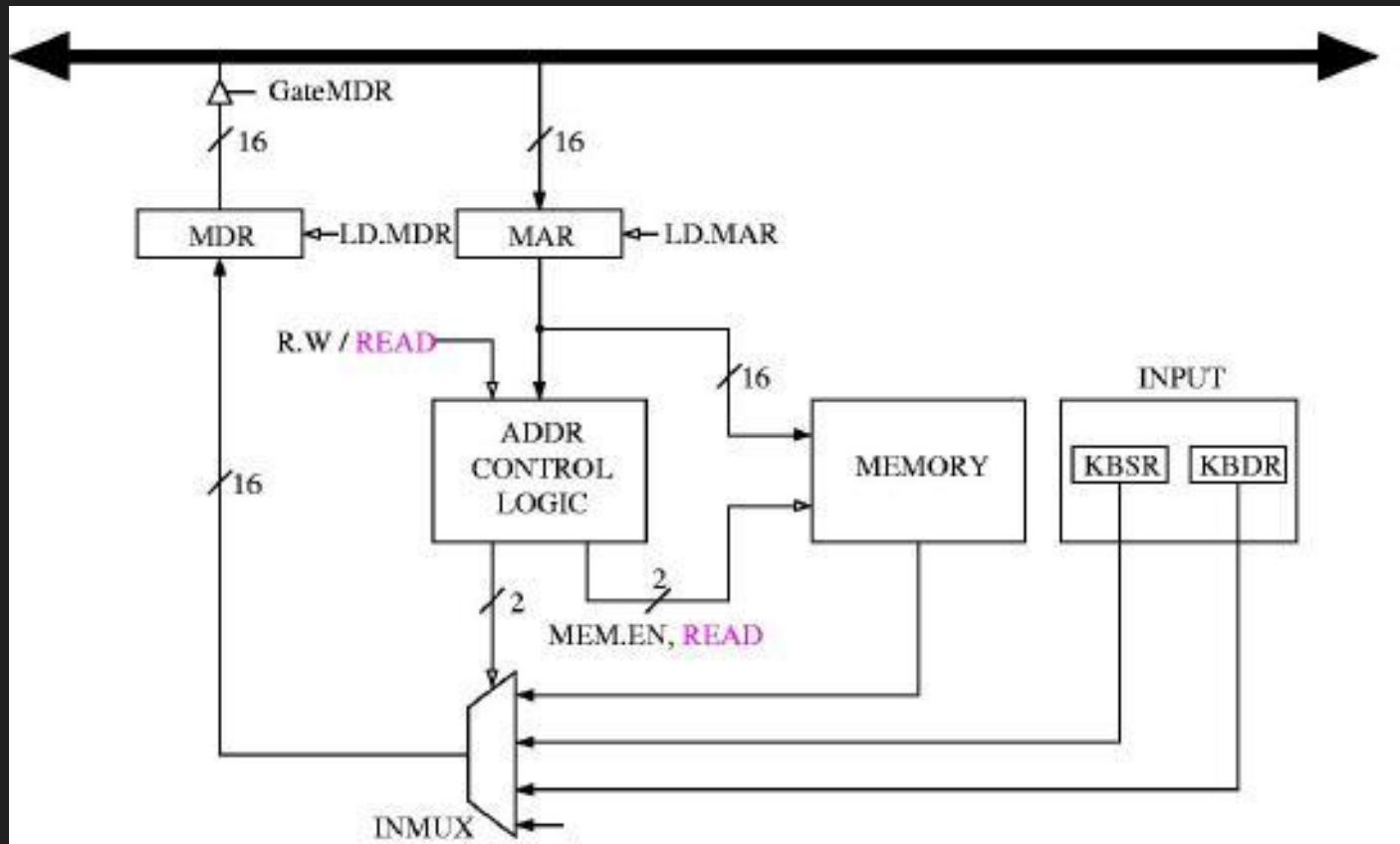
- I/O Interface
 - Through a set of Device Registers:
 - Status register (device is busy/idle/error)
 - Data register (data to be moved to/from device)
 - The device registers have to be read/written by the CPU.
- LC-3
 - KBDR: keyboard data register
 - KBSR: keyboard status register
 - DDR: display data register
 - DSR: display status register

Addressing Device Registers

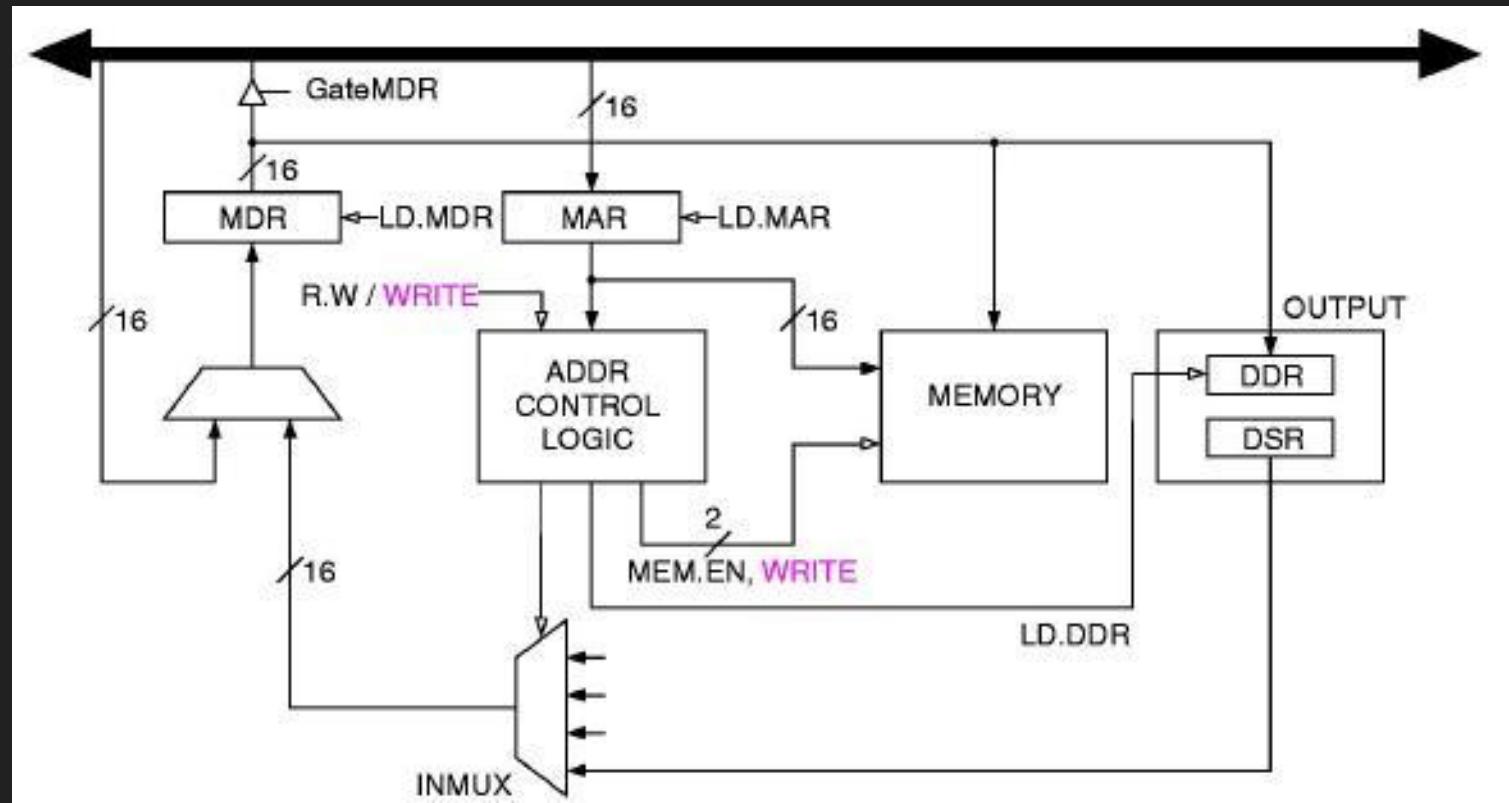
- Special I/O Instructions
 - Read or write to device registers using specialized I/O instructions.
- Memory Mapped I/O
 - Use existing data movement instructions (Load & Store).
 - Map each device register to a memory address (fixed).
 - CPU communicates with the device registers as if they were memory locations.
 - Frame buffers: Large areas of Memory Mapped I/O for video display
- LC-3
 - Uses memory mapped I/O:

xFE00	KBSR	Keyboard Status Register
xFE02	KBDR	Keyboard Data Register
XFE04	DSR	Display Status Register
XFE06	DDR	Display Data Register
XFFE	MCR	Machine Control Register

Memory Mapped Input



Memory Mapped Output



Synchronizing CPU and I/O

- Problem
 - Speed mismatch between CPU and I/O
 - CPU runs at up to 4 GHz, while all I/O is much slower
 - Example: Keyboard input is both slow and irregular
 - We need a protocol to keep CPU & KBD synchronized
 - Two common approaches
- Polling (handshake synchronization)
 - CPU checks the KBD Ready status bit
 - If set, CPU reads the data register and resets the Ready bit
 - Repeat
 - Makes CPU-I/O interaction seem to be synchronous
- Interrupt-driven I/O
 - An external device is allowed to interrupt the CPU and demand attention
 - The CPU attends to the device in an orderly fashion (more later)

Polling v/s Interrupts (Who's driving?)

- Polling: CPU in charge
 - CPU checks the ready bit of status register (as per program instructions).
 - If ($KBSR[15] == 1$) then load KBDR[7:0] to a register.
 - If the I/O device is very slow, CPU is kept busy waiting.
- Interrupt: peripheral in charge
 - Event triggered - when the I/O device is ready, it sets a flag called an interrupt
 - When an interrupt is set, the CPU is forced to an interrupt service routine (ISR) which services the interrupting device
 - There can be different priority levels of interrupt
 - Specialized instructions can mask an interrupt level

Polling Algorithm

- Input (keyboard)
 - The CPU loops checking the Ready bit
 - When bit is set, a character is available
 - CPU loads the character waiting in the keyboard data register
- Output (monitor)
 - CPU loops checking the Ready bit
 - When bit is set, display is ready for next character
 - CPU stores a character in display data register

Polling Details

- Keyboard
 - When key is struck
 - ASCII code of character is written to KBDR[7:0] (least significant byte of data register)
 - KBSR[15] (Ready Bit) is set to 1.
 - Keyboard is locked until CPU reads KBDR.
 - The CPU sees Ready Bit, reads KBDR, and clears the Ready Bit, unlocking the keyboard.
- Monitor
 - When CPU is ready to output a character
 - CPU checks DSR[15] (Ready Bit) until it is set to 1
 - CPU writes character to DDR[7:0]
 - Monitor sets DSR[15] to 0 while it is busy displaying the character, then sets it back to 1 to indicate readiness for next character.

Simple Polling Routines (Input)

```
START LDI R1, A      ;Loop if Ready not set
      BRzp START
      LDI R0, B      ;If set, load char to R0
      BR NEXT_TASK
A     .FILL xFE00    ;Address of KBSR
B     .FILL xFE02    ;Address of KBDR
```

Input a character from keyboard

Simple Polling Routines (Output)

```
START LDI R1, A      ;Loop if Ready not set
      BRzp  START
      STI R0, B      ;If set, send char to DDR
      BR  NEXT_TASK
A     .FILL xFE04    ;Address of DSR
B     .FILL xFE06    ;Address of DDR
```

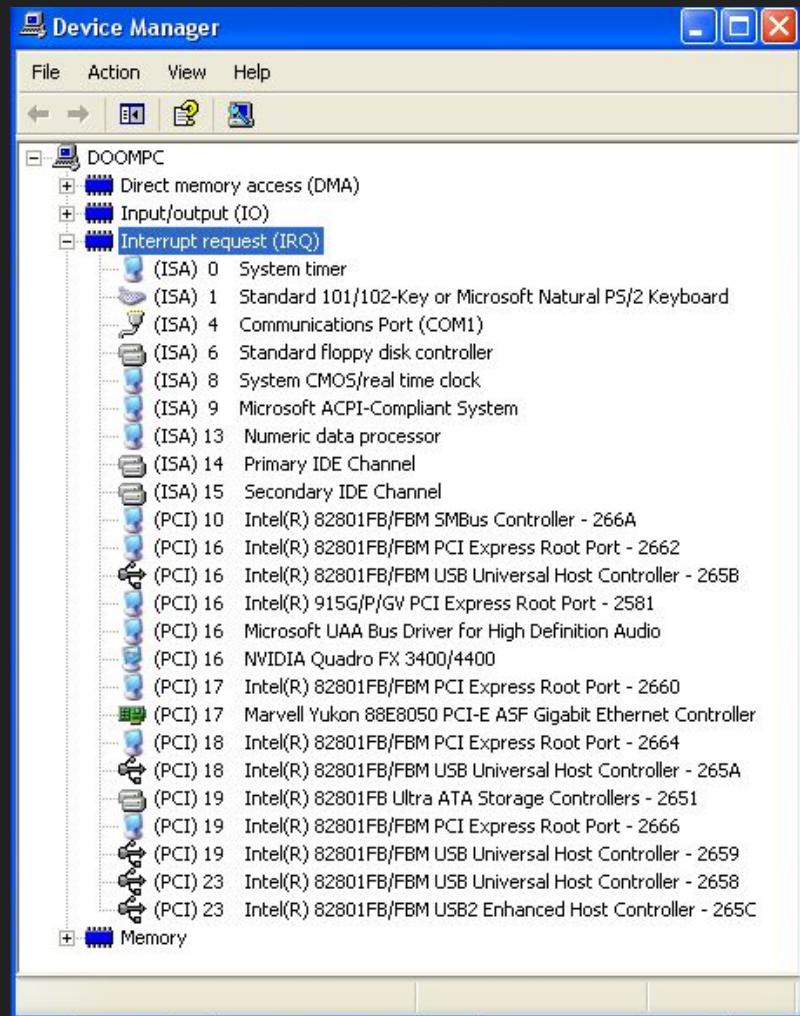
Output a character to the monitor

Interrupts Are Much More Efficient

- Your programs are not stuck waiting for an input
- IO that is infrequent do not waste precious processing power
- IO is only used/checked when it needs to be

Windows PCs

- Windows uses Memory Mapped IO + Interrupts



14 - Interrupts

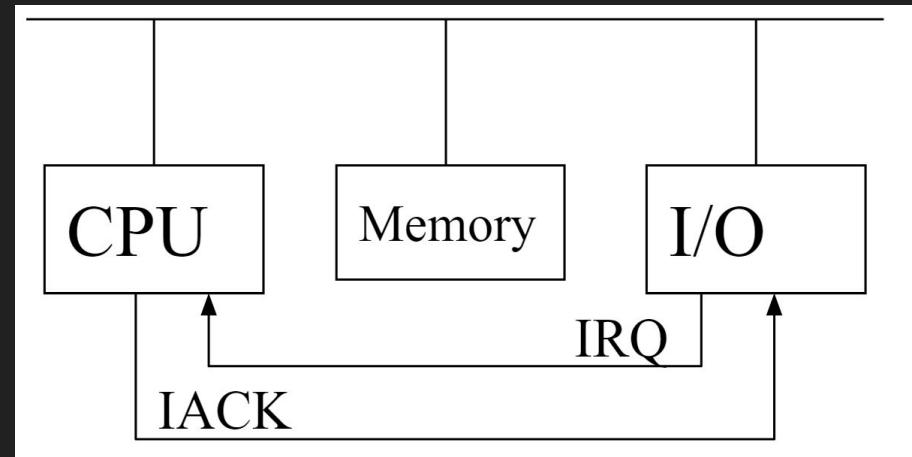
CEG3310/5310 - Computer Organization
Max Gilson

Interrupts

- Interrupts allow your computer to respond to I/O requests (IRQ) immediately, without having to poll
- Recall polling:
 - To get information from the keyboard, your program must loop, constantly checking if the keyboard is ready
- Interrupts:
 - Enabling an interrupt allows your program to jump to an ISR (interrupt service routine) so that the I/O can be dealt with when the I/O device requests
 - Afterwards, seamlessly return to your program

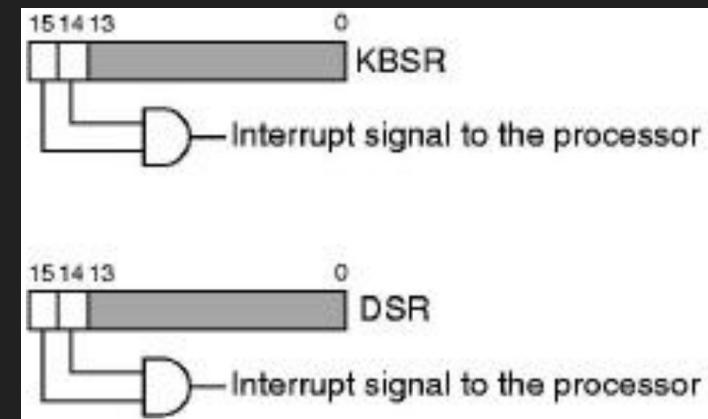
Interrupts (cont.)

- When IRQ (interrupt request) goes active, jump to a special memory location: the ISR (interrupt service routine)
- Activate IACK (interrupt acknowledged) to tell the device that the interrupt is being serviced, so it can stop activating the IRQ line



Enabling Interrupts

- Enabling interrupts on the LC3 can be done through the status registers
- Other computers may have different methods for enabling interrupts
- Recall, the KBSR aka Keyboard Status Register (x_{FE00})
 - Bit 15 of the KBSR indicates the keyboard is ready to send data
 - Bit 14 of the KBSR indicates that we want to enable an interrupt for this device
 - This allows for the CPU to dictate whether or not it wants to accept interrupts



Processing a Single Interrupt

- Device generates an IRQ
- CPU signals IACK – “OK, I’m on it.”
- Switch to Supervisor Mode
 - (admin/operating system mode)
- CPU saves its current state
- Address of the ISR is loaded into the PC
- Continue – process the interrupt
- When finished, return to running program

Interrupts and Program State

- We need to save the PC, the PSR, and all Registers
 - We could require that ISRs save all relevant registers (callee save)
 - The callee would ALWAYS have to save the contents of the PC and PSR
- In most computers these values (and possibly all register contents) are stored on a stack
 - Remember, there might be nested interrupts, so simply saving them to a register or reserved memory location might not work.
 - This is why R6 is used for the stack!

Interrupts on The LC3

- Only one interrupt exists on the LC3, the keyboard interrupt
- The ISR works very similarly to the TRAP service routines
- Interrupt vector table specifies what address the instructions start at for an ISR
 - For the LC3, the keyboard's interrupt vector table entry is at address x0180
 - So if you have data x3500 stored inside address x0180, then when the keyboard interrupt occurs, it will jump to address x3500 to execute instructions

Interrupts on The LC3 (cont.)

- Once you enter the ISR, the old PC and PSR are pushed to the stack (R6)
- The ISR should save all registers, execute instructions in ISR, then restore registers, and return
- The instruction RTI handles popping the PC and PSR and returns the PC to the instruction needed to be executed before the interrupt was requested

LC3 Interrupt Example

```
.ORIG x3000
    LD R6, STACK_BASE
    LD R0, ISR_SET
    STI R0, KEYBOARD
TOP      LD R0, ASCII_0
        AND R1, R1, #0
        ADD R1, R1, #10
PRINT_NUM   ADD R0, R0, #1
        OUT
        ADD R1, R1, #-1
        BRp PRINT_NUM
        LD R0, NEWLINE
        OUT
        BRnzp TOP

NEWLINE .FILL #10
ASCII_0  .FILL #47
STACK_BASE .FILL x3050
KEYBOARD .FILL xFE00
ISR_SET   .FILL x4000
.END
```

```
.ORIG x0180
    .FILL x3500
.END

.ORIG x3500
    STR R0, R6, #-1
    STR R1, R6, #-2
    STR R2, R6, #-3
    STR R3, R6, #-4
    STR R4, R6, #-5
    STR R5, R6, #-6
    STR R7, R6, #-7

    GETC
    LEA R0, MESSAGE
    PUTS
    LDR R0, R6, #-1
    LDR R1, R6, #-2
    LDR R2, R6, #-3
    LDR R3, R6, #-4
    LDR R4, R6, #-5
    LDR R5, R6, #-6
    LDR R7, R6, #-7

    RTI
MESSAGE .STRINGz "\nYou pressed a button!\n"
STACK_BASE2 .FILL x3150
.END
```

The Supervisor State

- Recall, the PSR that stores the NZP bits
- The PSR also contains valuable info such as priority and supervisor status
- Bit 15: 0 for supervisor privileges, 1 for standard user use
- Bits 10 - 8: 000 to 111 for interrupt priority 0 to 7
- Bits 2 - 0: ### for NZP bits, ex. 100 for N, 010, Z, 001, P
- So for a PSR, in user mode, priority 3, and positive result looks like:
 - PSR = 1000 0011 0000 0001

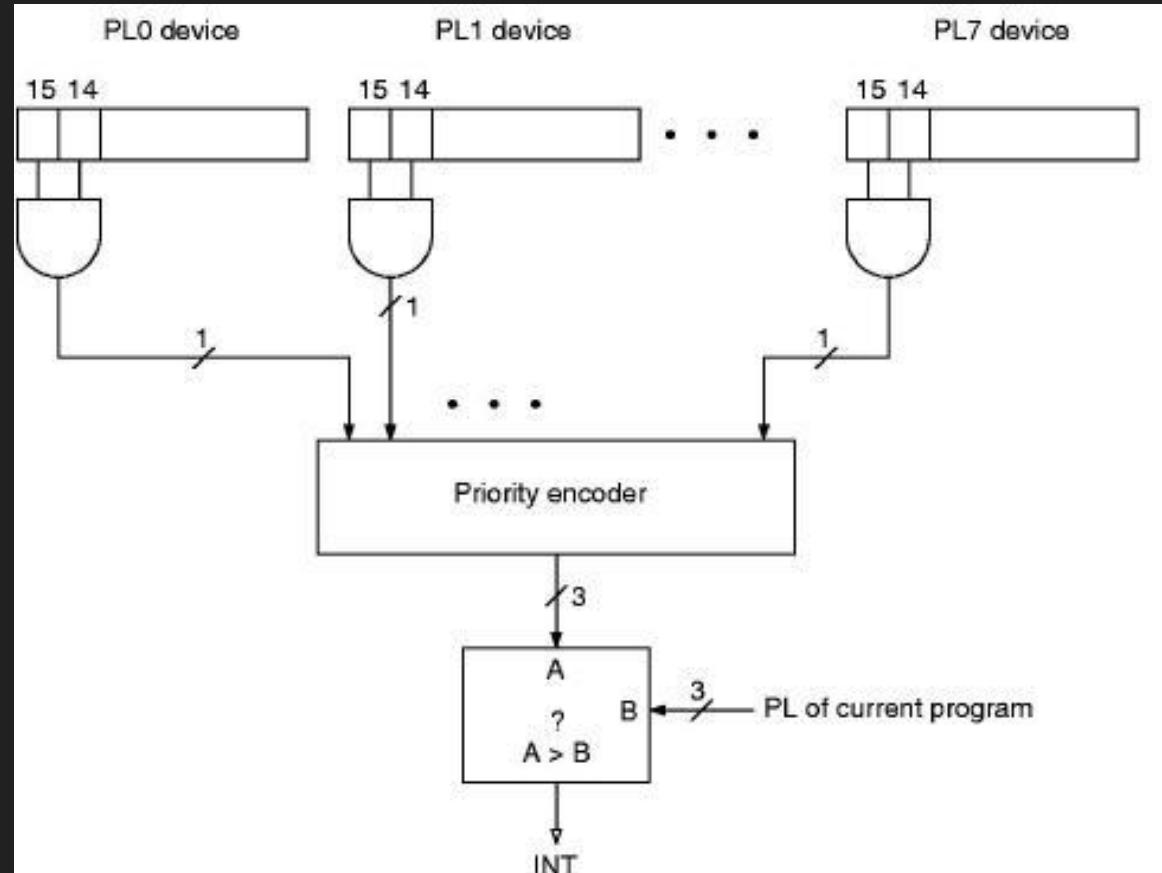
The Supervisor State (cont.)

- The Memory Protection Register (MPR) dictates what areas of memory are allowed to be accessed when in user mode
 - The MPR is not currently implemented on the LC3, if it was, this is how it would function
- This restricts a user's program from accessing special locations in memory
- Supervisor mode bypasses this
- When in supervisor mode (when an interrupt gets triggered) special protected code can be executed/read/written
- This protected code is for memory mapped I/O, OS instructions, ISRs, etc.

Interrupt Priorities

- Since interrupts can occur at any time and interrupt your program, it would be helpful to prevent an ISR from interrupting something critical
- The keyboard ISR is priority level 4
- If you run something critical, like a program to control a nuclear power plant, this should be a high priority level, like priority 7
 - Now, the keyboard will not interrupt your critical program

Interrupt Priorities



The Supervisor Stack

- When an interrupt is triggered, the supervisor stack is loaded into R6
- Two things get pushed onto the stack
 - PC
 - PSR
- This is really important because a second interrupt could be triggered while we are processing an interrupt!
- Since the PC and PSR are saved to the stack, they will not get lost
- Now that we understand the supervisor mode and priority level, saving the PSR to the supervisor stack now makes more sense

Other Uses of Interrupts

- Interrupts are also used for:
 - Errors (divide by zero, etc.)
 - TRAPs
 - Operating system events (quanta for multitasking, etc.)
 - User generated events (Ctrl-C, Ctrl-Z, Ctrl-Alt-Del, etc.)
 - ...and more.

Exceptions

- Exceptions are handled the same way as interrupts
- The LC3 currently has 2 exceptions:
 - privilege mode violation (calling RTI as a user)
 - illegal opcode (executing an instruction with opcode 1101, ex. xD000)
- Exception handling is not currently implemented in the LC3, if it was it would function like this:
 - When an exception occurs go to exception vector table
 - Find address stored in exception vector table that corresponds to the exception
 - Execute the code starting from address retrieved
 - Return from exception

Lab 6 Display Integer Example

- How can we display a positive hex value from x0000 to x7FFF in decimal to the user?
- The highest decimal digit that this hex value can be is the 10000's digit
 - First, subtract 10,000 from hex value
 - If value is negative, then we know the hex value is <10,000, don't print anything for this decimal place
 - If value is positive, keep subtracting 10,000 keeping track of number of times a subtracting occurs until 0 or negative
 - Print the counter and continue to the next digit
 - Repeat these steps for digits 1,000, 100, 10, 1
- Example:
 - x0F14 = 3860
 - 10,000's subtraction count = 0 (don't print)
 - 1,000's subtraction count = 3 (print)
 - 100's subtraction count = 8 (print)
 - 10's subtraction count = 6 (print)
 - 1's subtraction count = 0 (always print this digit)

12 - Recursion

CEG3310/5310 - Computer Organization
Max Gilson

Recursion

- A recursive function is a function that completes a task by calling itself
- This is similar to iterative code and can be used interchangeably
- Recursion can be more/less efficient and depends on the problem being solved

Recursion Concepts

- Base case
 - The problem that we know the answer to
 - When no more recursive calls are needed
 - Every recursive function may have a base case or else it would go on forever
- General case
 - The problem that we need to perform another recursive call for
 - This general case will happen every time except for the base case

Recursion Example

- Let's say we want to calculate the factorial of a number:
 - $6! = 6 * 5 * 4 * 3 * 2 * 1$
- How can we do this recursively?
 - Base Case: 1
 - We know the last number in the factorial is 1 so that's perfect for a base case
 - General Case: $\text{result} = x * \text{factorial}(x-1)$
 - If $x = 6$ we can follow the steps:

Recursion Example (cont.)

- To calculate $6!$ we must do these steps:
- $\text{result} = n * \text{factorial}(n - 1)$ <- General Case
 - $\text{result} = 6 * \text{factorial}(6 - 1)$
 - $\text{factorial}(6-1) = 5 * \text{factorial}(5-1)$
 - $\text{factorial}(5-1) = 4 * \text{factorial}(4-1)$
 - $\text{factorial}(4-1) = 3 * \text{factorial}(3-1)$
 - $\text{factorial}(3-1) = 2 * \text{factorial}(2-1)$
 - $\text{factorial}(2-1) = 1$ <- Base Case

Recursion Example (cont.)

- Now plug in all the values:
- $\text{result} = n * \text{factorial}(n - 1)$ <- General Case
 - $\text{result} = 6 * \text{factorial}(6 - 1) = 6 * 5 * 4 * 3 * 2 * 1 = 720$
 - $\text{factorial}(6-1) = 5 * \text{factorial}(5-1) = 5 * 3 * 2 * 1$
 - $\text{factorial}(5-1) = 4 * \text{factorial}(4-1) = 4 * 3 * 2 * 1$
 - $\text{factorial}(4-1) = 3 * \text{factorial}(3-1) = 3 * 2 * 1$
 - $\text{factorial}(3-1) = 2 * \text{factorial}(2-1) = 2 * 1$
 - $\text{factorial}(2-1) = 1$ <- Base Case

Recursion Example (cont.)

- C Code:

```
int factorial(int x)
{
    if(x == 1)
    {
        return x;
    }else
    {
        return x*factorial(x-1);
    }
}
```

Recursion Example 2

- Let's say we want to print numbers 0 to n-1 to the user
- We can call this function countUp(n)
- How can we do this recursively?
 - Base Case: 0
 - General Case: print(countUp(n-1)); return n;
 - If x = 5 we can follow the steps:

Recursion Example 2 (cont.)

- To print these values from 0 to 5 we must:
- `print(countUp(n-1))` <- General Case
`return n;`
 - `print(countUp(5-1))`
 - `print(countUp(4-1))`
 - `print(countUp(3-1))`
 - `print(countUp(2-1))`
 - `print(countUp(1-1))` <- Base case ($n = 0$)
 - `countUp(0): return 0;`

Recursion Example 2 (cont.)

- Now plug in all the values:
- ```
print(countUp(n-1)) <- General Case
return n;
```

  - `print(countUp(5-1)); Console = "01234"`
  - `print(countUp(4-1)); Console = "0123"`
  - `print(countUp(3-1)); Console = "012"`
  - `print(countUp(2-1)); Console = "01"`
  - `print(countUp(1-1)); Console = "0"`
    - `countUp(0): return 0;`

# Recursion Example 2 (cont.)

- C Code:

```
int countUp(int x)
{
 if(x == 0)
 {
 return x;
 }else
 {
 printf("%d", countUp(x-1));
 return x;
 }
}
```

# The Base Case is Really Important

- Imagine if you miss/forget the base case
- Your function will keep calling more functions, infinitely
- This can crash your program and cause a variety of issues
  - Stack Overflow

# How Does Recursion Look in Assembly?

- Implement recursive functions as subroutines
- The subroutines are implemented as you would with any other subroutine:
  - Initialize a proper runtime stack
  - Pass inputs
  - Check for base case
    - Compute base case
  - Compute general case
  - Return outputs

# Recursion in Memory

# Recursion in Memory (cont.)

# 06 - Digital Logic

CEG3310/5310 - Computer Organization  
Max Gilson

# The Power of Electrons

- How can we make electrons perform calculations?
- To answer this we need to know two things:
  - What fundamental operations can we make electricity (electrons) do?
  - How can we combine these fundamental operations to do useful things?

# Operations on Bits

- An operation can be represented by a truth table
- Three fundamental operations are AND ( $ab$ ), OR ( $a+b$ ), and NOT ( $a'$ ):

| $a$ | $b$ | $a b$ |
|-----|-----|-------|
| 0   | 0   | 0     |
| 0   | 1   | 0     |
| 1   | 0   | 0     |
| 1   | 1   | 1     |

| $a$ | $b$ | $a + b$ |
|-----|-----|---------|
| 0   | 0   | 0       |
| 0   | 1   | 1       |
| 1   | 0   | 1       |
| 1   | 1   | 1       |

| $a$ | $a'$ |
|-----|------|
| 0   | 1    |
| 1   | 0    |

# Boolean Algebra

- Simple operations:
  - Suppose  $a = 0, b = 1$
  - $a + b = ?$
  - $ab = ?$
  - $(a + ((ab)')) = ?$ 
    - We need DeMorgan's Law

# DeMorgan's Law

$$(a + b)' = a'b'$$

$$(ab)' = a' + b'$$

# More Boolean Algebra

- Simple operations:
  - Suppose  $a = 0, b = 1$
  - $(a' + b)'$
  - $a' + (ab)'$
  - $(a + b')(a + b)'$
- What are the values?
- Can we make our own truth table?

# Bit Vector Operations

- Computers often represent data using bit vectors
  - Bit vectors are just arrays of bits like we've been dealing with already
- We can apply Boolean operations to entire bit vectors at once:

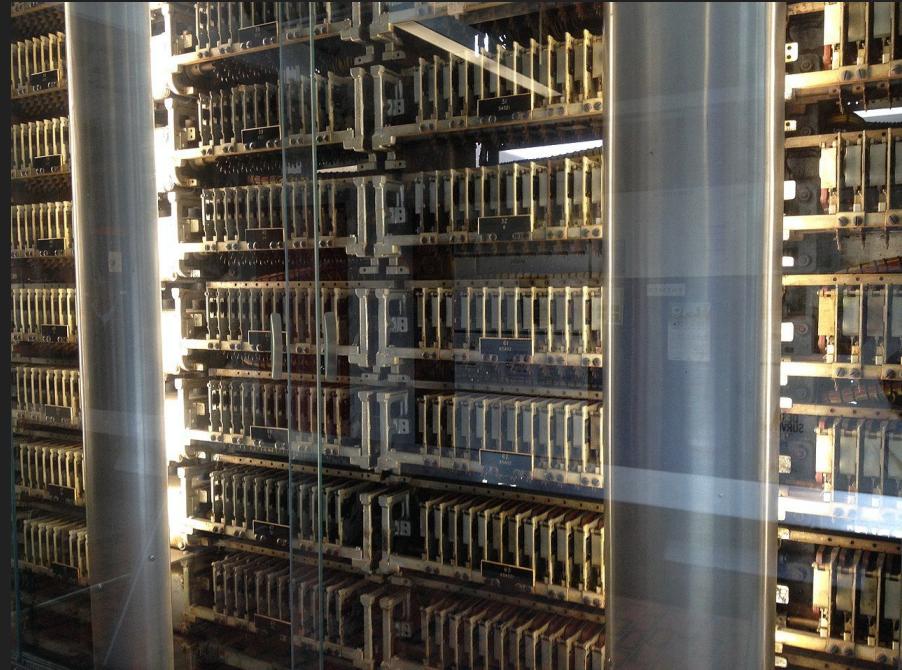
10100101  
AND 01101011  
00100001

# How Can Electrons Do This?

- All of computing is made up of a few very basic operations like AND, OR, and NOT
- These simple digital operations, applied to an appropriate binary representation can be combined to perform complex tasks such as
  - Addition
  - Multiplication
  - Fortnite®

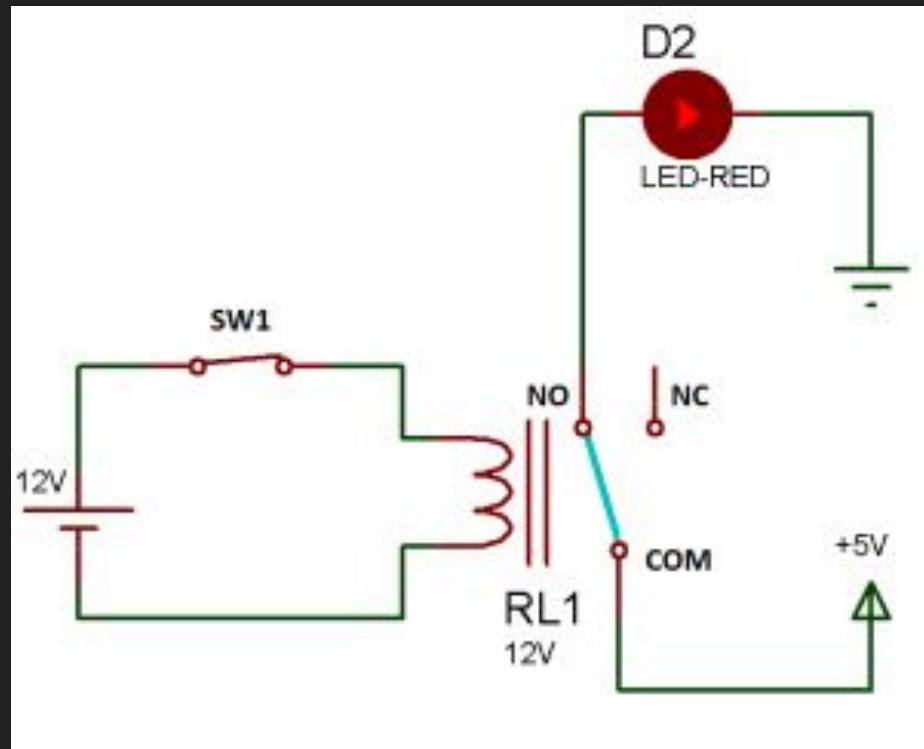
# The First\* Computers Used Relays

- Harvard Mark I
- 3 additions or subtractions per second
- Multiplication took 6 seconds
- Division took 15.3 seconds
- \*The “first” computer was invented in 1833 and was completely mechanical



# The First Computers Used Relays (Cont.)

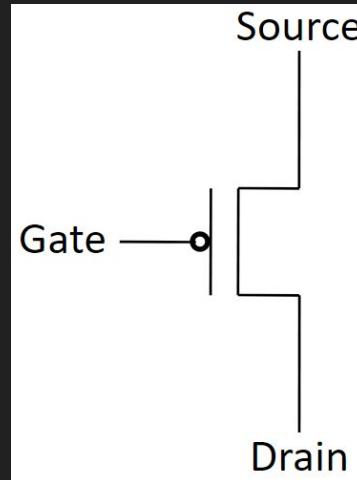
- Relays are very slow compared to vacuum tubes, and transistors
- Clunky and consume a lot of power relatively



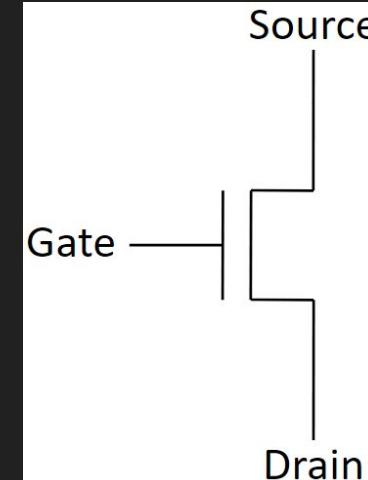
# MOS Transistors

- Metal-Oxide Semiconductor = MOS
  - P-type: gate = 0 volts  $\Rightarrow$  closed switch (current flows)
  - N-type: gate = 2.9 volts  $\Rightarrow$  closed switch

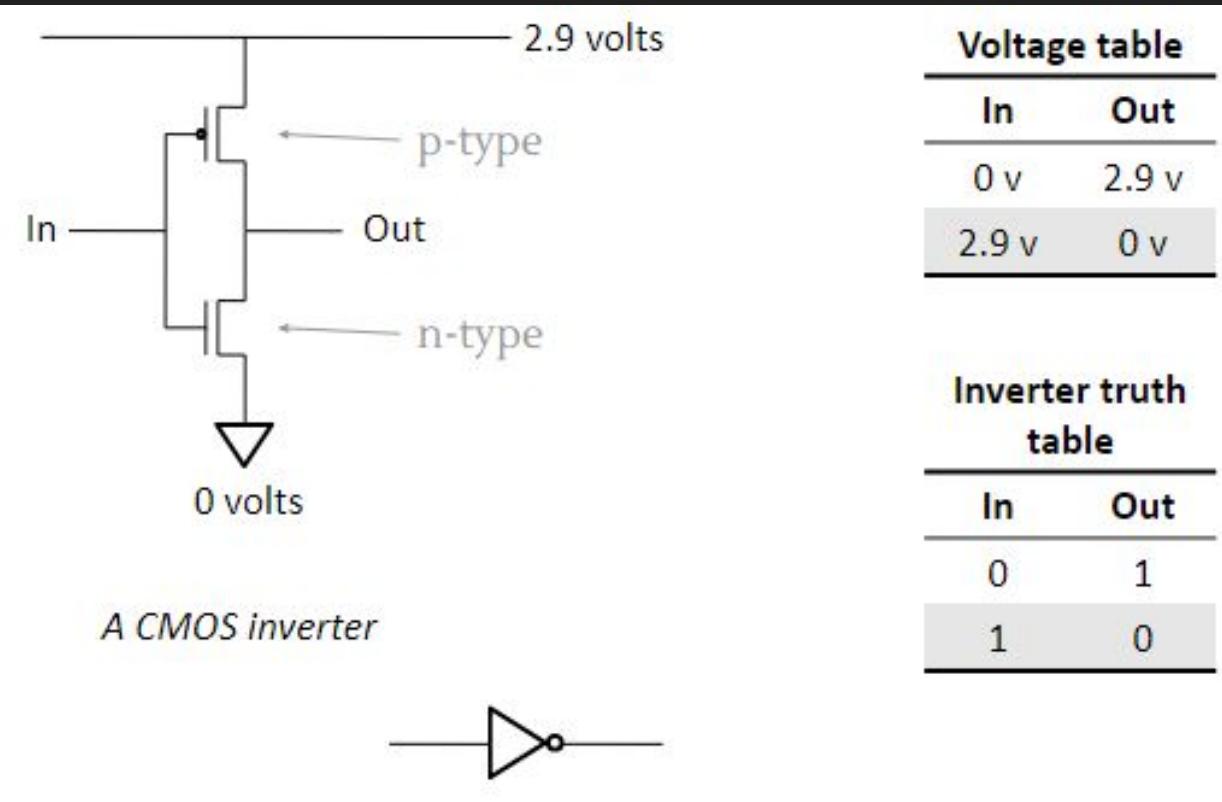
P-type:



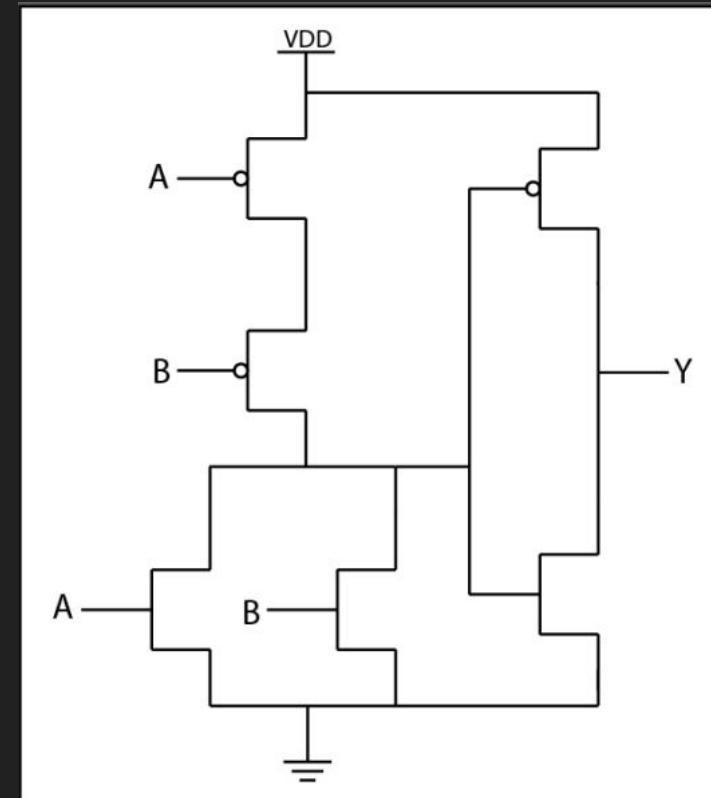
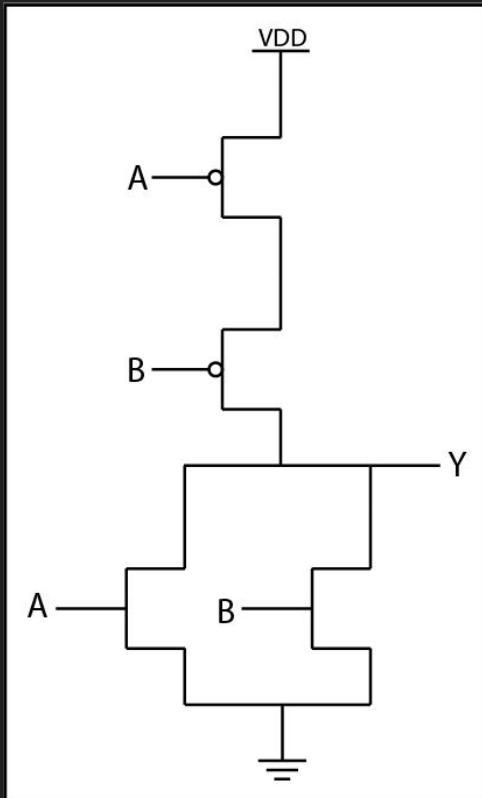
N-type:



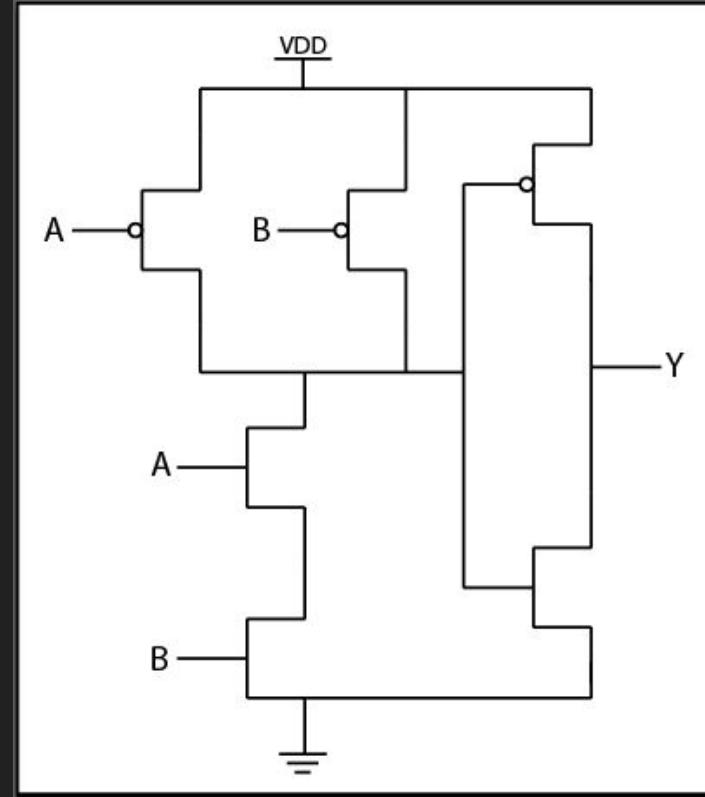
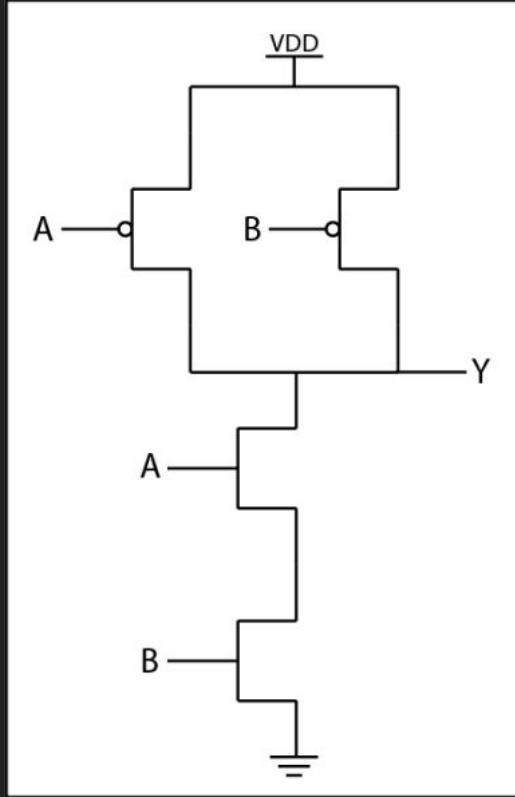
# Combining Transistors: Logic Gates



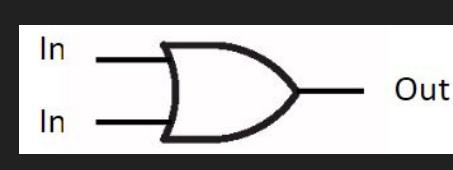
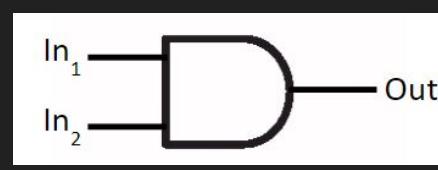
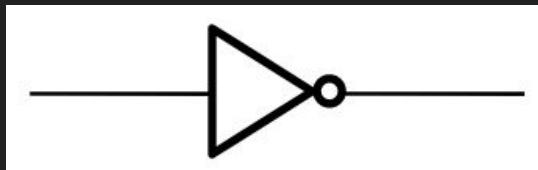
# Combining Transistors: Logic Gates (NOR/OR)



# Combining Transistors: Logic Gates (NAND/AND)



# Represented As Gates



**NOT**

| <b>In</b> | <b>Out</b> |
|-----------|------------|
| 0         | 1          |
| 1         | 0          |

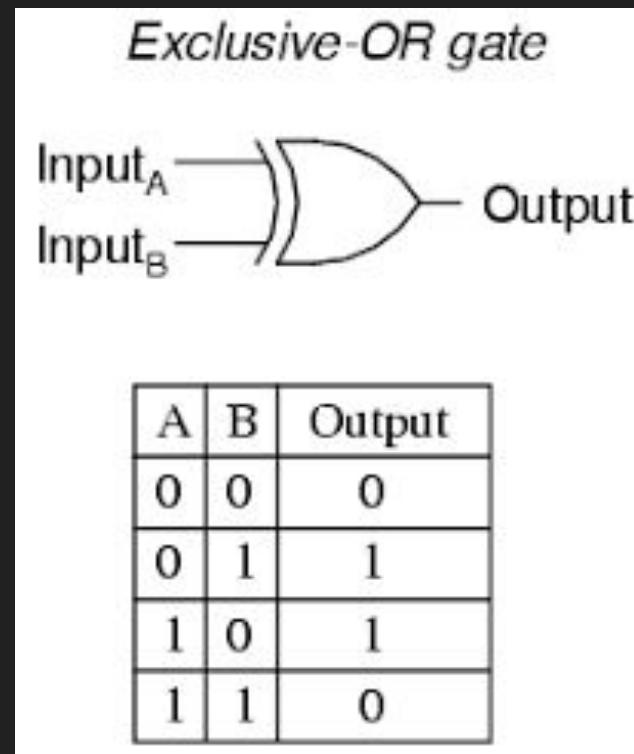
**AND**

| $In_1$ | $In_2$ | Out |
|--------|--------|-----|
| 0      | 0      | 0   |
| 0      | 1      | 0   |
| 1      | 0      | 0   |
| 1      | 1      | 1   |

**OR**

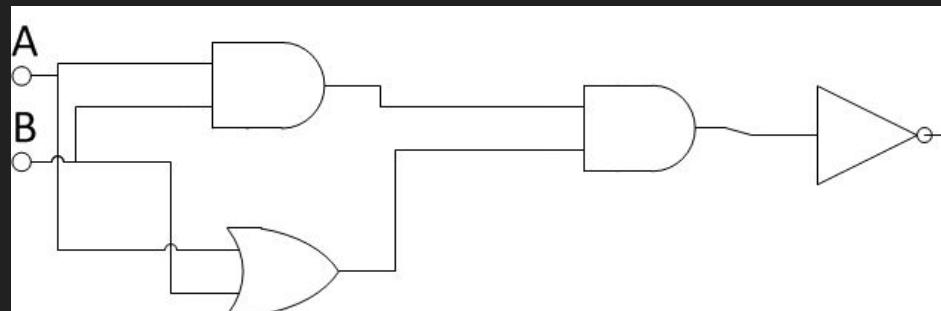
| $In_1$ | $In_2$ | Out |
|--------|--------|-----|
| 0      | 0      | 0   |
| 0      | 1      | 1   |
| 1      | 0      | 1   |
| 1      | 1      | 1   |

# Represented As Gates (Cont.)



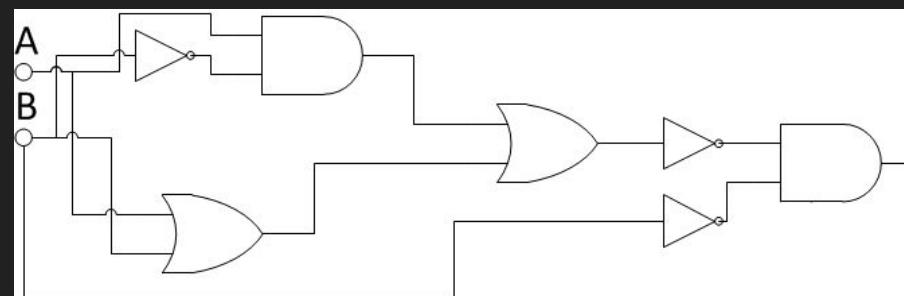
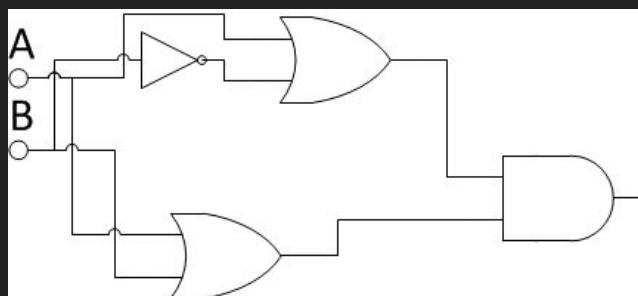
# Gates and Boolean Logic

- Draw gate diagrams for these expressions:
  - $(a+b)'$
  - $a(a+b)+(a'+b)$
  - $(a+b)(ab')(a')$
- Write a boolean expression for this gate diagram:



# Gates and Boolean Logic

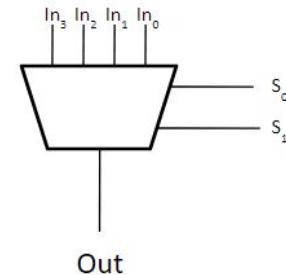
- Write a boolean expression for this gate diagram:



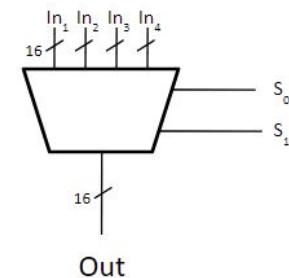
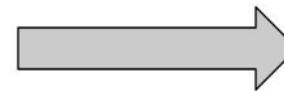
# Multiplexers

| S1 | S0 | Out             |
|----|----|-----------------|
| 0  | 0  | In <sub>0</sub> |
| 0  | 1  | In <sub>1</sub> |
| 1  | 0  | In <sub>2</sub> |
| 1  | 1  | In <sub>3</sub> |

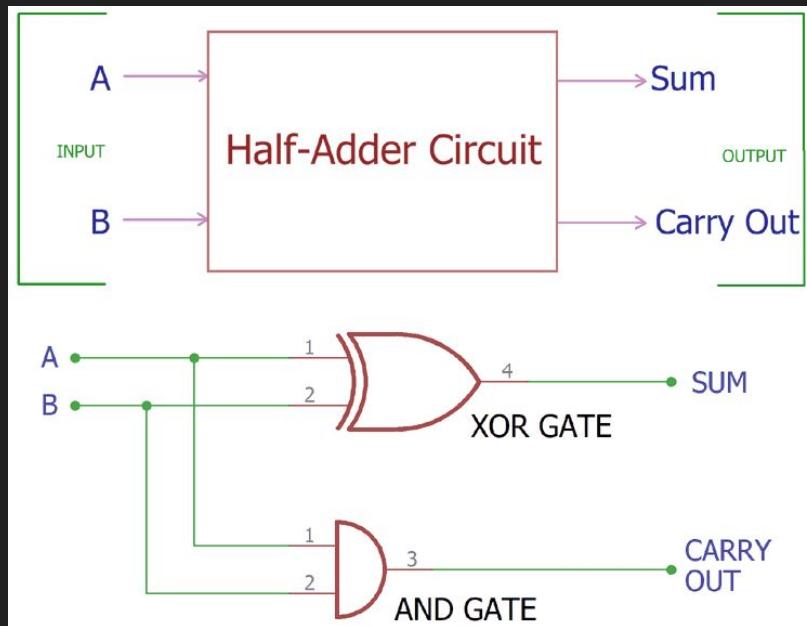
A four-input MUX



*With 16 of these,  
we can build a  
16-bit four input  
MUX*

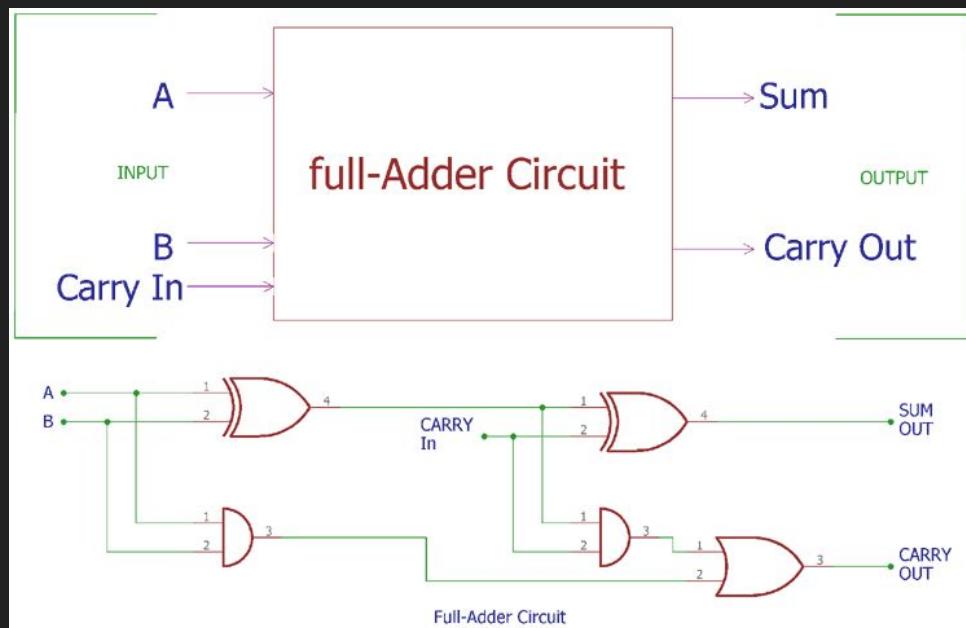


# Half Adders



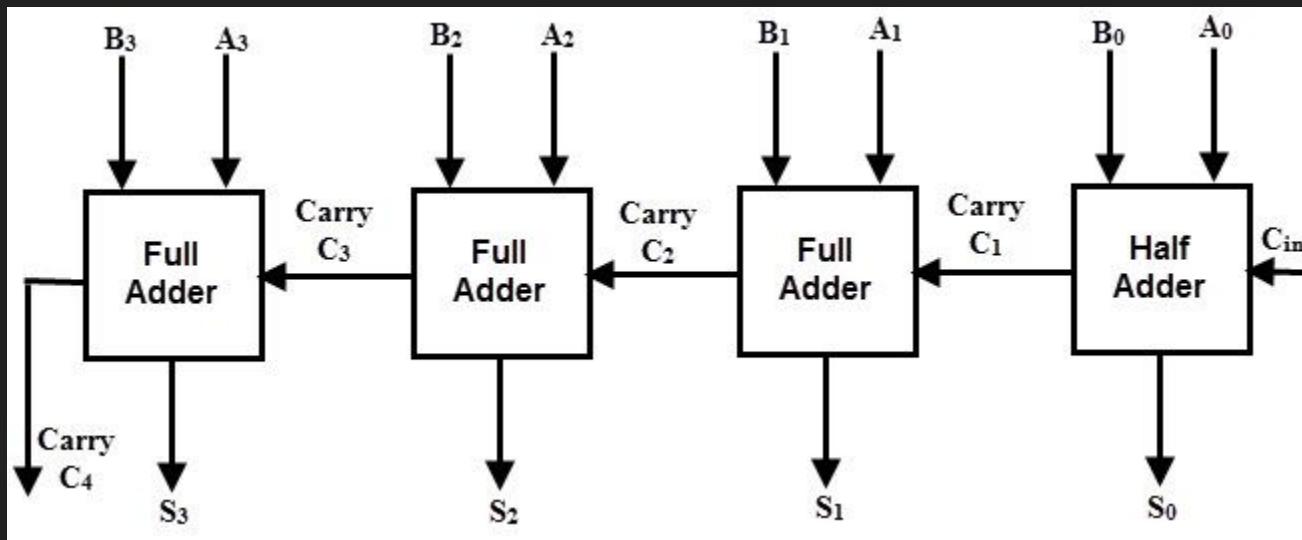
| Truth Table |   |        |       |
|-------------|---|--------|-------|
| Input       |   | Output |       |
| A           | B | Sum    | Carry |
| 0           | 0 | 0      | 0     |
| 0           | 1 | 1      | 0     |
| 1           | 0 | 1      | 0     |
| 1           | 1 | 0      | 1     |

# Full Adders



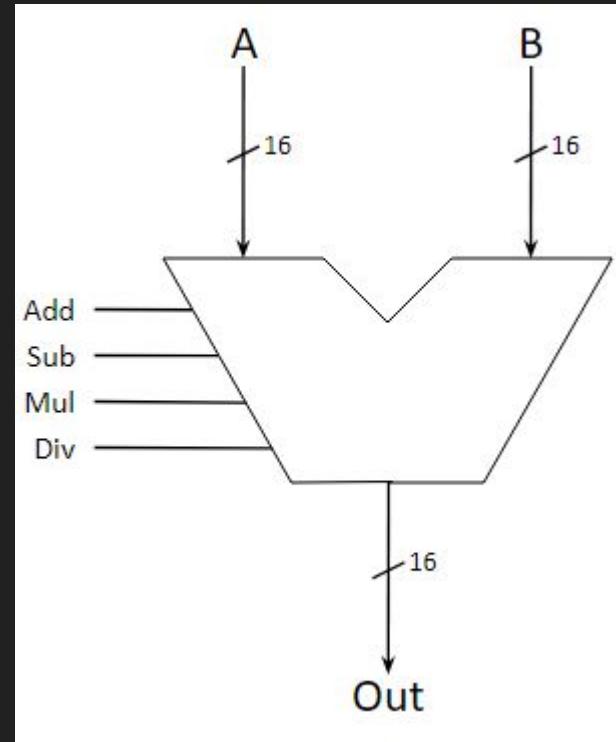
| Input |   |     | Output |       |
|-------|---|-----|--------|-------|
| A     | B | Cin | Sum    | Carry |
| 0     | 0 | 0   | 0      | 0     |
| 0     | 0 | 1   | 1      | 0     |
| 0     | 1 | 0   | 1      | 0     |
| 0     | 1 | 1   | 0      | 1     |
| 1     | 0 | 0   | 1      | 0     |
| 1     | 0 | 1   | 0      | 1     |
| 1     | 1 | 0   | 0      | 1     |
| 1     | 1 | 1   | 1      | 1     |

# Combined Full Adders



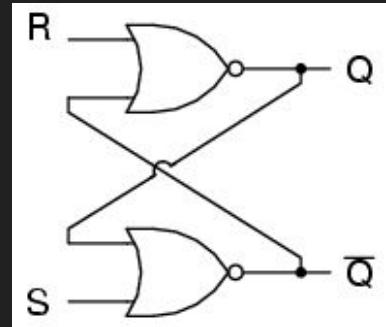
# Arithmetic Logic Unit (ALU)

- Gives us addition, subtraction, multiplication, and division capabilities
  - What are we missing on the LC3?
- This device has no memory and only calculates arithmetic operations



# Memory Basics (SR Latch)

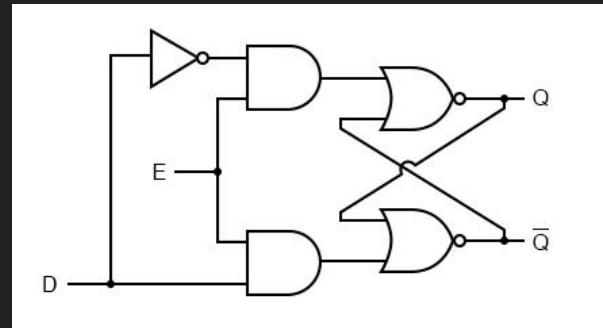
- The SR Latch gives us extremely basic memory capabilities
- How does this give us memory?
- First Set = 1, Q = 1
- Next when Set = 0, Q = 1
- This latches onto the Set value, even when it goes back to 0
- There is one problem, S=1, and R=1 is invalid!



| Set | Reset | Q              | Q'             |
|-----|-------|----------------|----------------|
| 0   | 0     | Previous Value | Previous Value |
| 0   | 1     | 0              | 1              |
| 1   | 0     | 1              | 0              |
| 1   | 1     | 0              | 0              |

# Memory Basics (Gated D Latch)

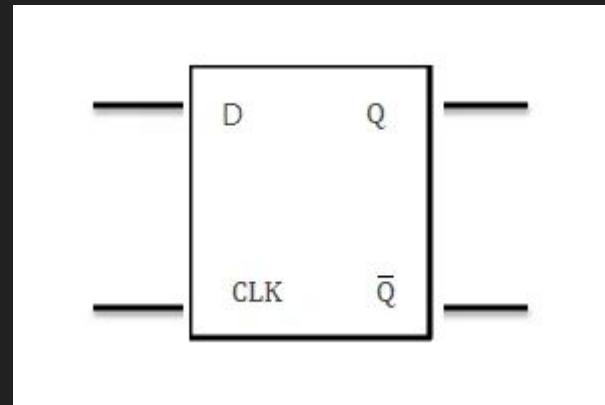
- Gated allows us to either enable or disable changing the output
- There are no more invalid inputs!



| Enable | D | Q              | $\bar{Q}$      |
|--------|---|----------------|----------------|
| 0      | 0 | Previous Value | Previous Value |
| 0      | 1 | Previous Value | Previous Value |
| 1      | 0 | 0              | 1              |
| 1      | 1 | 1              | 0              |

# Memory Basics (D Flip Flop)

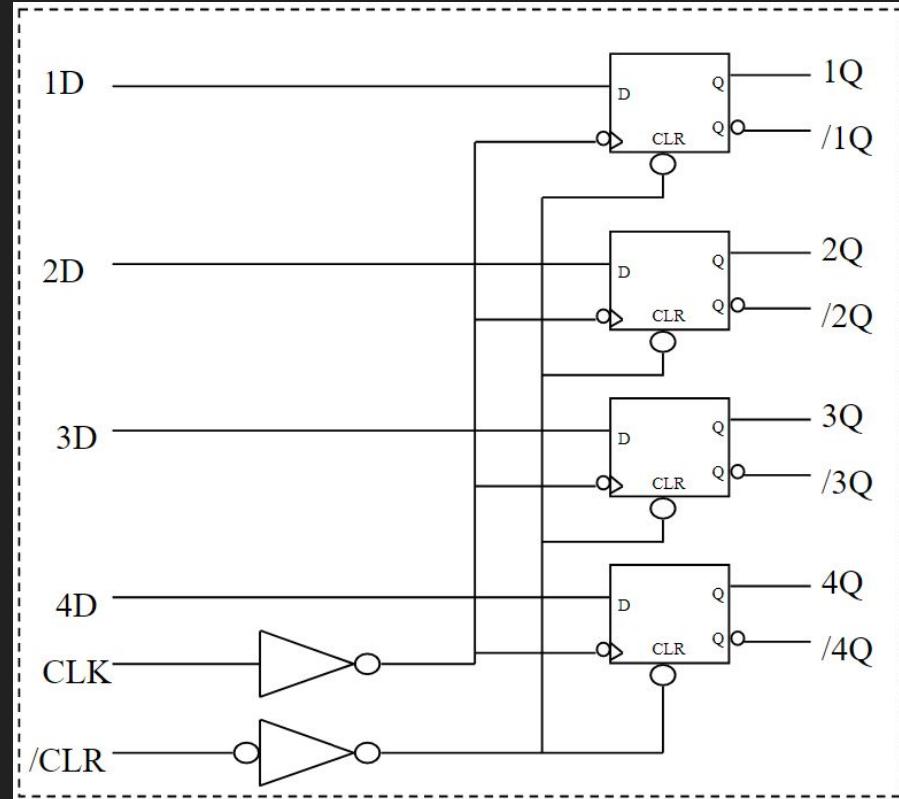
- Flip Flops are different than latches
- Flip Flops are edge triggered
  - D can only change value at a clock edge, NOT whenever  $CLK = 1$
- This allows for use to store our values until the next clock cycle is ready to use them
- There are many processes in computers that can be done in 1 clock cycle, the flip flop let's us synchronize them to this clock



| CLK  | D | Q              | $Q'$           |
|------|---|----------------|----------------|
| Low  | 0 | Previous Value | Previous Value |
| Low  | 1 | Previous Value | Previous Value |
| High | 0 | 0              | 1              |
| High | 1 | 1              | 0              |

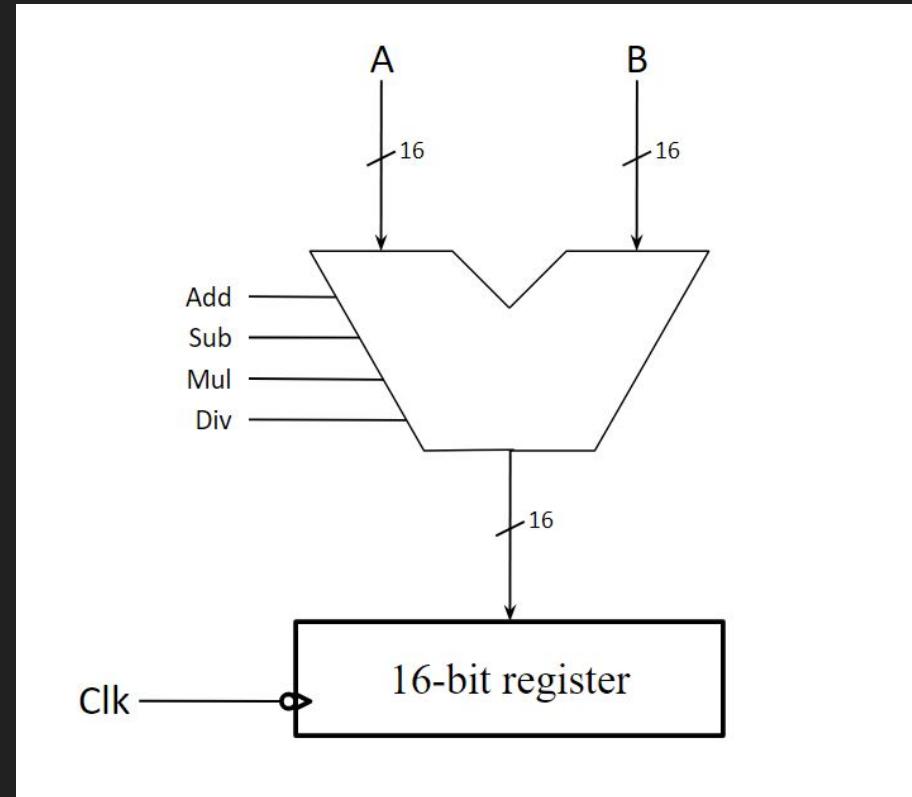
# Registers

- Flip flops let us store things in registers
- This allows us to perform various operations in 1 clock cycle
- Then use the valid data at the next clock cycle



# Registers + ALU

- Once the ALU is able to output a result, it can get stored in a register
- This acts as a buffer
- The output is now waiting to be used once at the next clock cycle



# Practice Problems (NOT REQUIRED)

- 3.3, 3.9, 3.14, 3.29, 3.31, 3.25

# 07 - The Von Neumann Model

CEG3310/5310 - Computer Organization  
Max Gilson

# How to Build a Computer

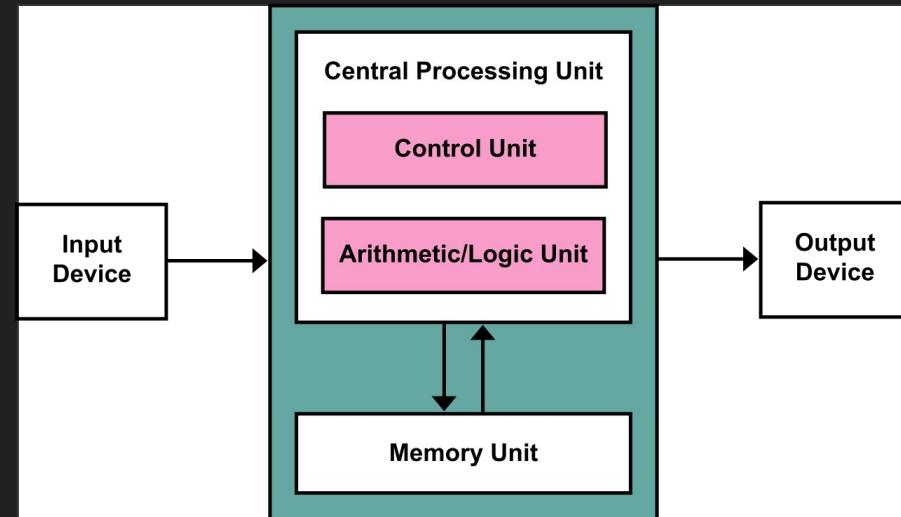
- To get a task done by a computer, we need two things:
  - A program that specifies what must be done to complete the task
  - Hardware that is capable of completing those tasks as specified
- Low-level design of computer components is “digital design”
  - This was discussed on the last set of slides
- The basic computer model was proposed by John von Neumann in 1946

# What is The Von Neumann Model

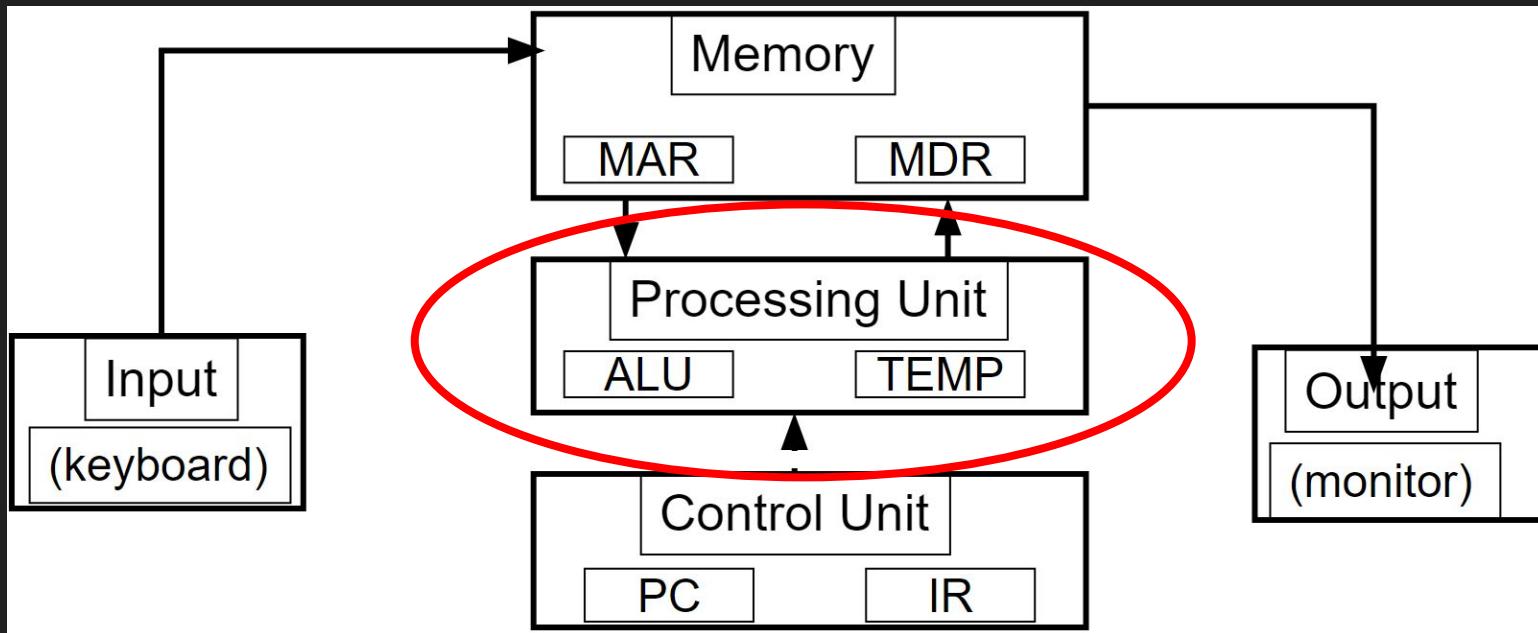
- A conceptual computer architecture
- One of the most fundamental and widely used models for computer architecture
- Outlines the core components necessary for a digital computer
- Other models do exist, i.e. Harvard Architecture

# The Von Neumann Model

- Memory
  - holds both data and instructions
- Processing Unit
  - carries out the instructions
- Control Unit
  - sequences and interprets instructions
- Input
  - external information into the memory
- Output
  - produces results for the user



# The Von Neumann Model - The Processing Unit

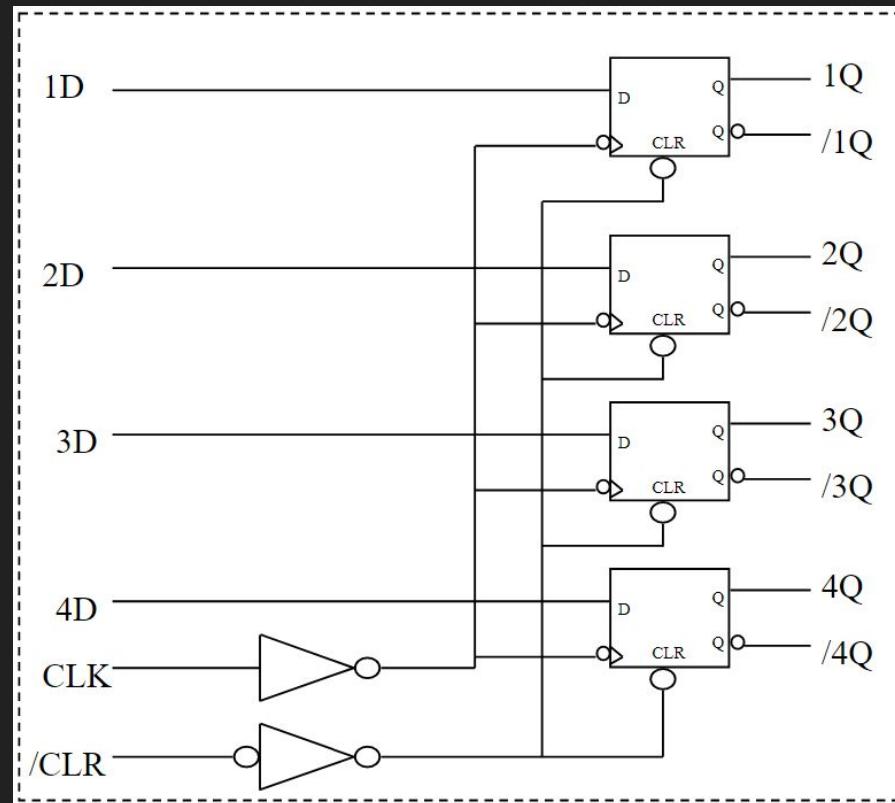


# The Processing Unit

- Processing Unit - CPU
  - Can consist of many units, each specializing in one complex function.
  - At a minimum, has Arithmetic & Logic Unit (ALU) and temporary storage (registers).
  - The number of bits a basic Processing Unit operation can handle is called the WORD SIZE of the machine.
  - Arithmetic & Logic Unit - ALU
    - Performs basic operations: add, subtract, and, not, etc.
    - Generally operates on whole words of data.
    - Some can also operate on subsets of words (eg. single bits or bytes).
  - Registers
    - Fast “on-board” storage for a small number of words.
    - Invaluable for intermediate data storage while processing.
    - Close to the ALU (much faster access than RAM).
    - General Purpose Registers (GPRs) are available to the programmer.

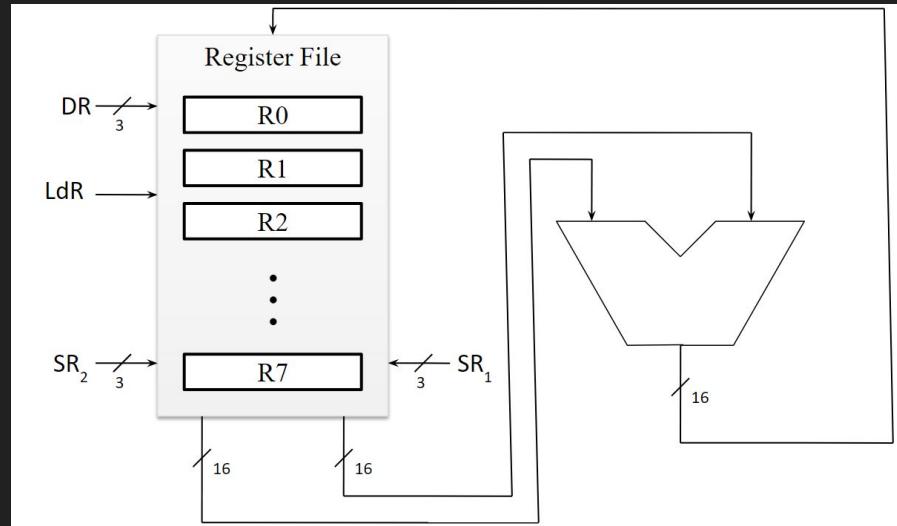
# General Purpose Registers (GPR)

- Small temporary storage
  - This is not like RAM or a drive
- Gives us immediate access to numbers we wish to manipulate
- In the LC3 we have GPRs R0-R7, each are 16-bits wide

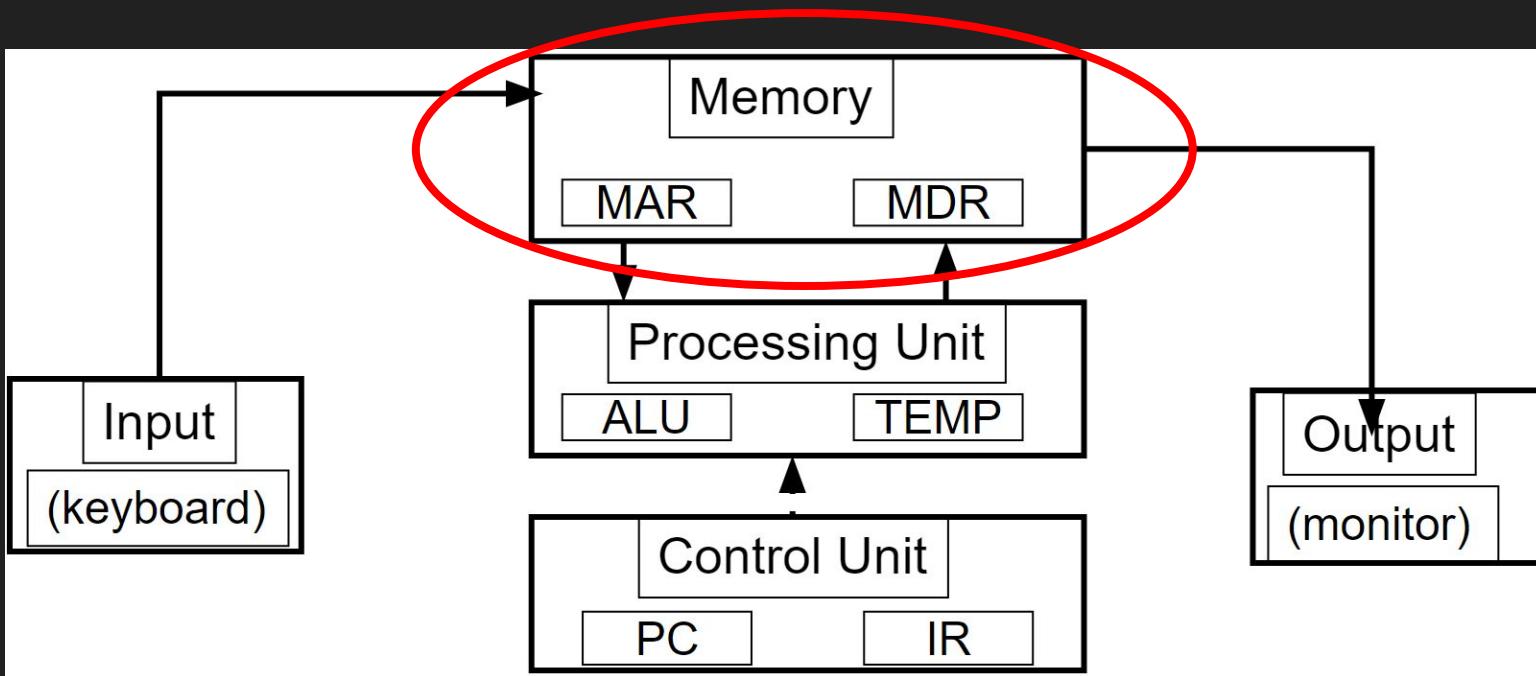


# Central Processing Unit (CPU) Overview

- Pass data loaded into registers to the ALU to perform computations
- Notice:
  - The results from the operation are sent back into the registers!
  - Whenever we use an operate instruction, we have to specify the Destination Register (DR)
  - This is why!



# The Von Neumann Model - Memory



# Memory

- Random Access Memory (RAM)
  - We are not talking about external storage yet, i.e. hard drives, SSD's, flash memory, cloud, etc.
- Memory is a large number of addressable fixed-sized locations
  - Each location has an address and contents
  - Address: bit pattern that uniquely identifies a memory location
  - Contents/Data: bit pattern stored at a given address.
- Address Space
  - $n$  bits allow the addressing of  $2^n$  memory locations.
- Example: 24 bits can address  $2^{24} = 16,777,216$  locations (i.e. 16M locations)... but what SIZE is each location?

# The Von Neumann Architecture - Memory

- We will access memory via a standard interface
  - The Memory Address Register (MAR) sets up the decoder circuitry in the memory
  - The Memory Data Register (MDR) hold contents to/from memory (bidirectional)
  - The Write enable signal indicates which direction information should flow.
- Memory Read
  - Read: the contents of the specified address will be written to the Memory Data Register.
  - If: Write enable is not applied in any given clock tick
  - Then:  $MDR \leftarrow \text{Memory}[(\text{MAR})]$
- Memory Write
  - Write: the value to be stored is first written to the Memory Data Register, then the Write Enable is asserted, and the contents of the MDR are written to the specified address.
  - If: a Write enable is applied in any given clock tick
  - Then:  $\text{Memory}[(\text{MAR})] \leftarrow \text{MDR}$

# Data In Memory

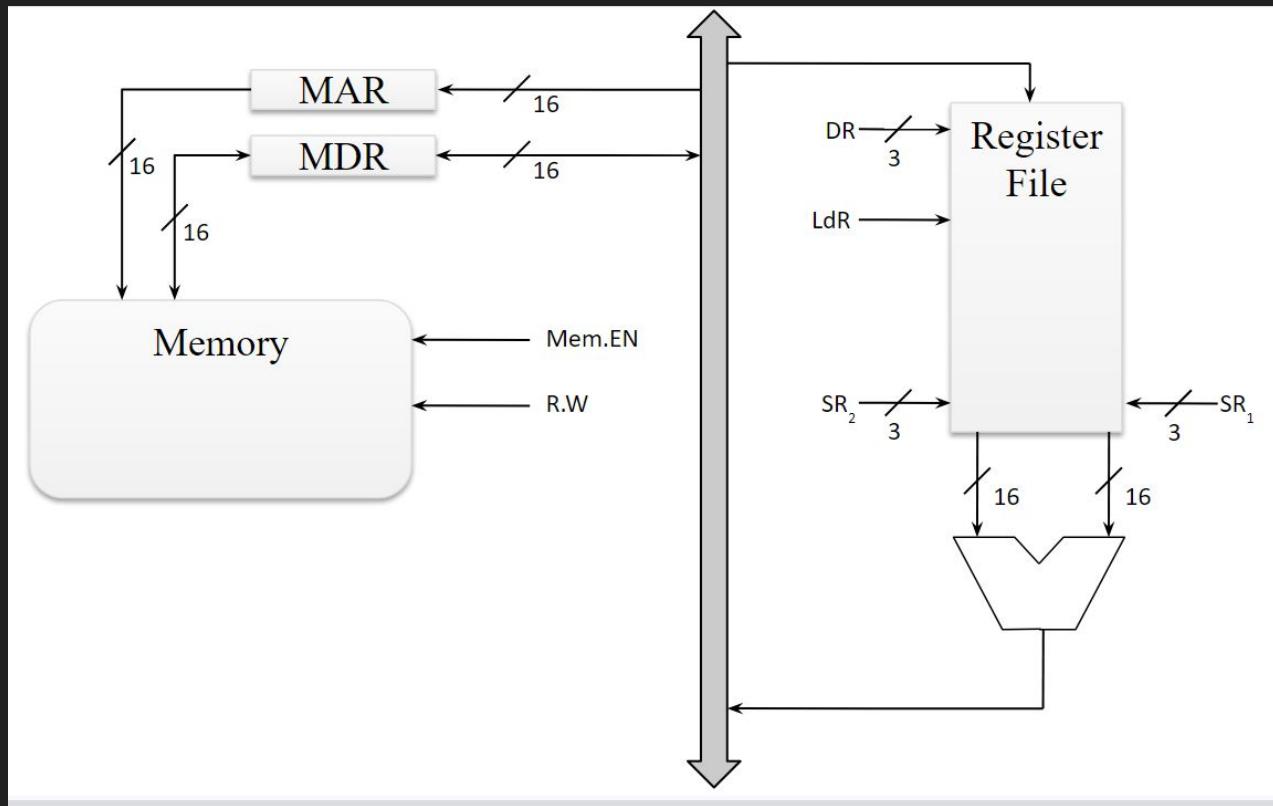
| Address | Value |
|---------|-------|
| x1000   | x0030 |
| x1001   | x0300 |
| x1002   | xC2FE |
| x1003   | x1F00 |
| x1004   | x1000 |
| x1005   | x3F3F |

Is this value...

- A positive integer?
- A negative integer?
- A float?
- An address?
- Data?
- Part of a program (a machine instruction)?

*How can we tell?*

# The LC3 So Far...



# Machine Words

- Machine Has “Word Size”
  - Nominal size of integer-valued data
    - Including addresses
  - The LC3 word size is 2 bytes or 16 bits
  - Most current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - High-end systems are 64 bits (8 bytes)
    - Potentially address  $\approx 1.8 \times 10^{19}$  bytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

## Sizes of C Objects (in Bytes)

| C Data Type     | Typical 32-bit |
|-----------------|----------------|
| – int           | 2              |
| – long int      | 4              |
| – char          | 1              |
| – short         | 2              |
| – float         | 4              |
| – double        | 8              |
| – long double   | 8 to 16        |
| – char *        | 4              |
| – Any other ptr | 4              |

# Byte Addressable Concepts

- If we can only store one byte per address, how can we create variables/data that take up more than 1 byte?
- We can order each byte in memory one after the other
- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address

# Big Endian vs Little Endian Example

- Given a variable  $x$  that takes up 4-bytes
- Where  $x$  in byte form looks like: 0x01234567
- The address for  $x$  in our program starts at 0x100

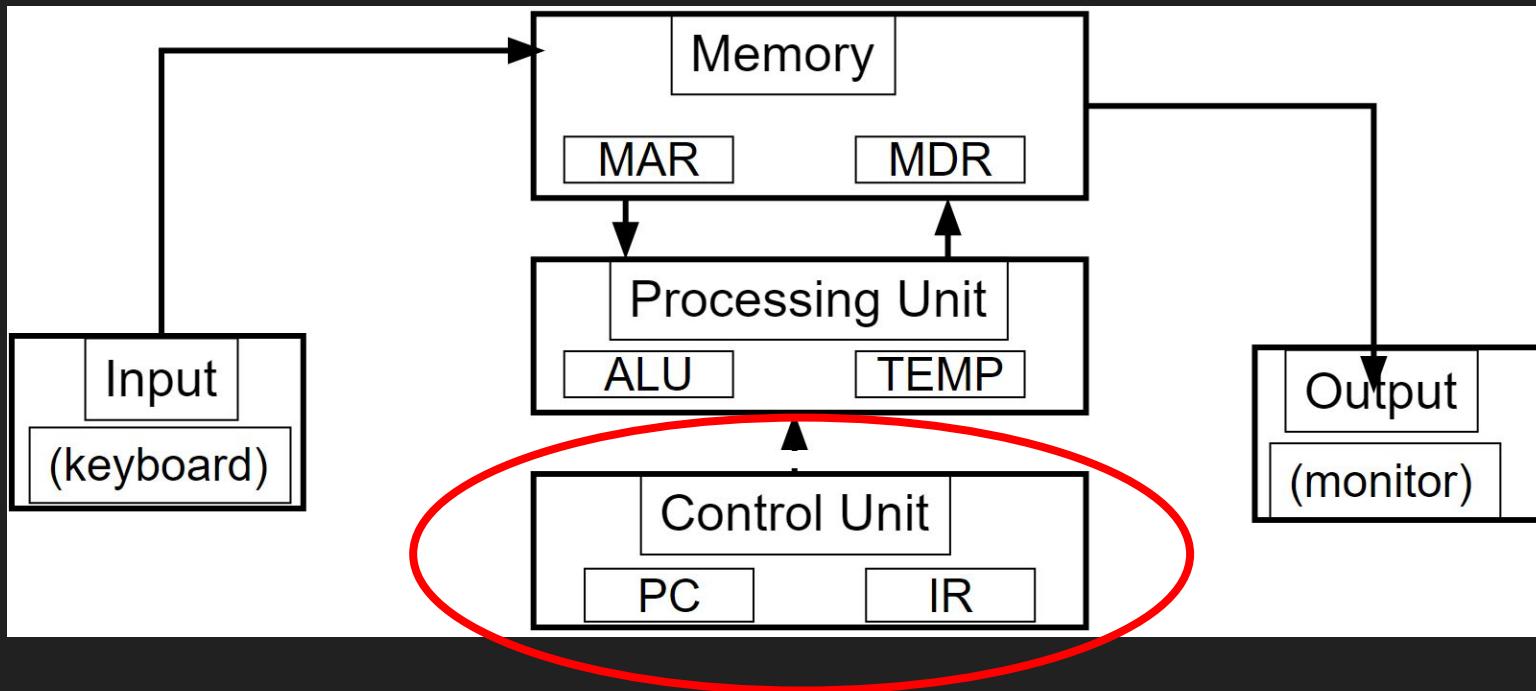
BigEndian:

| Address | Data |
|---------|------|
| 0x0FF   | —    |
| 0x100   | 0x01 |
| 0x101   | 0x23 |
| 0x102   | 0x45 |
| 0x103   | 0x67 |
| 0x104   | —    |

LittleEndian:

| Address | Data |
|---------|------|
| 0x0FF   | —    |
| 0x100   | 0x67 |
| 0x101   | 0x45 |
| 0x102   | 0x23 |
| 0x103   | 0x01 |
| 0x104   | —    |

# The Von Neumann Model - Control Unit



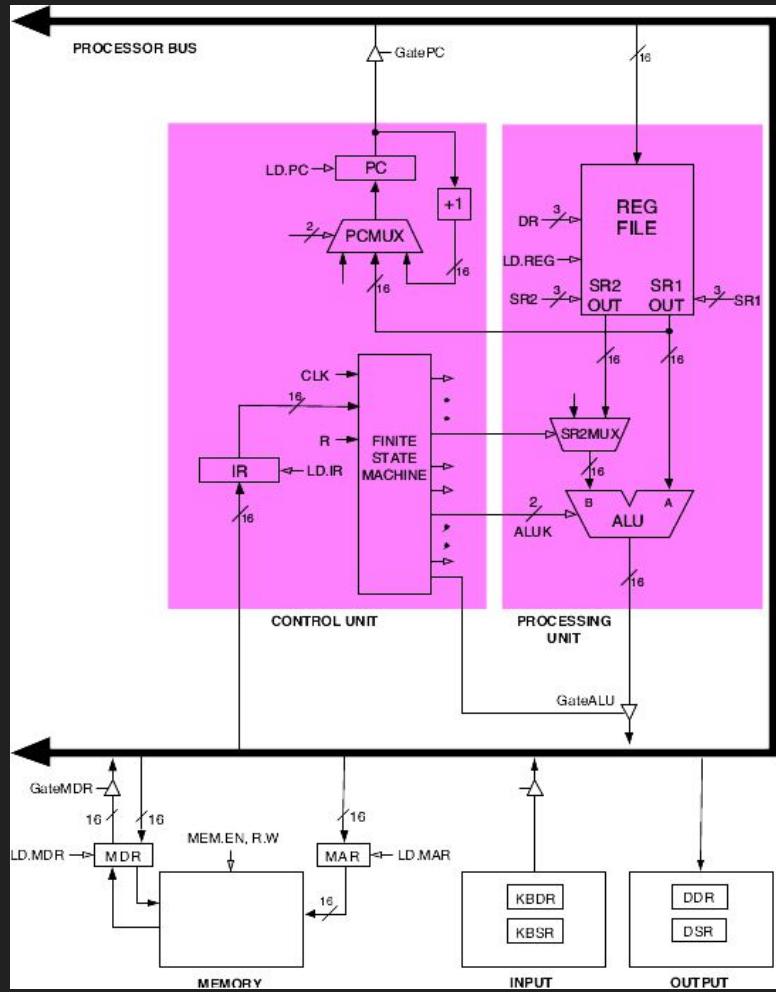
# Control Unit

- The control unit coordinates all actions needed to execute the instruction
  - Fetches, then decodes each instruction
  - Sets up inputs for memory, processing, and I/O
  - Communicates with memory via the Program Counter (PC) and Instruction Register (IR)

# Control Unit (Cont.)

- Program Counter (PC)
  - Pointer to the *next* instruction
  - Holds the address of the next instruction to be fetched
- Instruction Register (IR)
  - Holds the instruction currently being executed
- Input/Output (I/O)
  - Also known as peripherals, they are external to the CPU
  - Keyboard, mouse, printer, displays, networking

# The LC3 - A Von Neumann Machine



# The Instruction Cycle

- The Control Unit orchestrates the complete execution of each instruction
  - At its heart is a Finite State Machine that sets up the state of the logic circuits according to each instruction.
  - This process is governed by the system clock - the FSM goes through one transition (“machine cycle”) for each tick of the clock.

# Clock Cycles

- One clock cycle is often (not always) the time it takes for the machine to perform a simple instruction
- The clock cycle is broken down into sub-cycles for each phase
- These sub cycles make up the phases in the Instruction Cycle

# Faster Clock = More Performance?

- Most modern CPUs will process multiple instructions per clock cycle
  - It depends on the instructions executed and the CPU itself
  - Conducting benchmark tests give a more realistic view of the performance of a CPU
- AMD Ryzen 5 3600XT
  - @ 3.4GHz clock speed
  - 6 cores / 12 threads
- AMD Ryzen 5 5600X
  - @ 3.4GHz clock speed
  - 6 cores / 12 threads
  - Outperforms 3600XT in benchmark tests

See: <https://www.youtube.com/watch?v=8QOoQWvrQ-Y>

# The Instruction Cycle (Cont.)

- The instruction cycle has 6 phases:
  - Fetch
  - Decode
  - Evaluate address
  - Fetch operands
  - Execute
  - Store results

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

# Decode

- 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 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

# Evaluate Address

- For instructions that require memory access, compute address used for access.
  - Examples:
    - LDR R0, R1, #2
      - If:  $R1 = x3010$
      - Then:  $R1 + 2 = x3012$  is computed
    - BRx #-3
      - If:  $PC + 1 = x3006$
      - Then:  $PC + (-3) = x3003$  is computed

# Fetch Operands

- Obtain source operands needed to perform operation.
- Operands can come from Registers or RAM, or be embedded in the instruction itself.
- The Effective Address (EA) determined in the previous step may be used to obtain an operand from memory.
  - Examples
    - load data from memory (LDR)
    - read data from register file (ADD)

# Execute

- 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 Result

- 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

# Return to Fetch

- This cycle repeats, going back to the fetch step
- The fetch step will get the next instruction and the remaining steps will be carried out
- This cycle allows for instructions to be fully executed one right after the other

# Changing the Sequence of Instructions

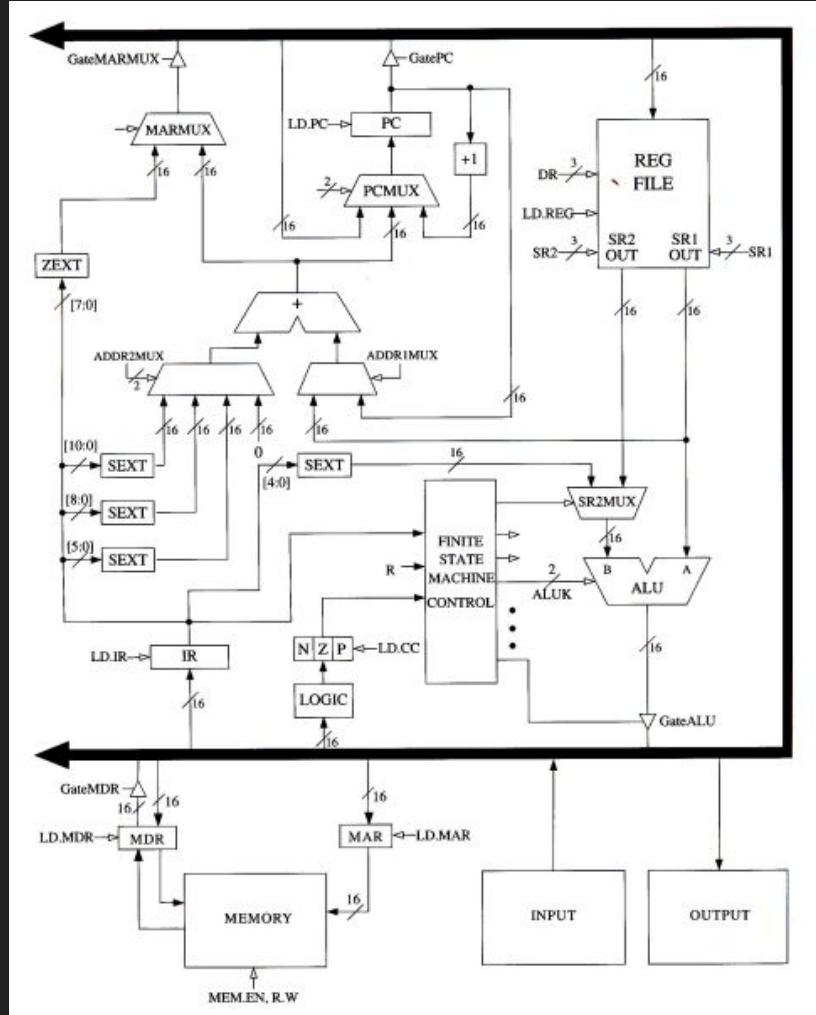
- In the FETCH phase, we increment the Program Counter by 1.
- What if we don't want to always execute the instruction that follows this one?
  - Jumps
  - Branches

# 07a - LC3 Data Path and Control Signals

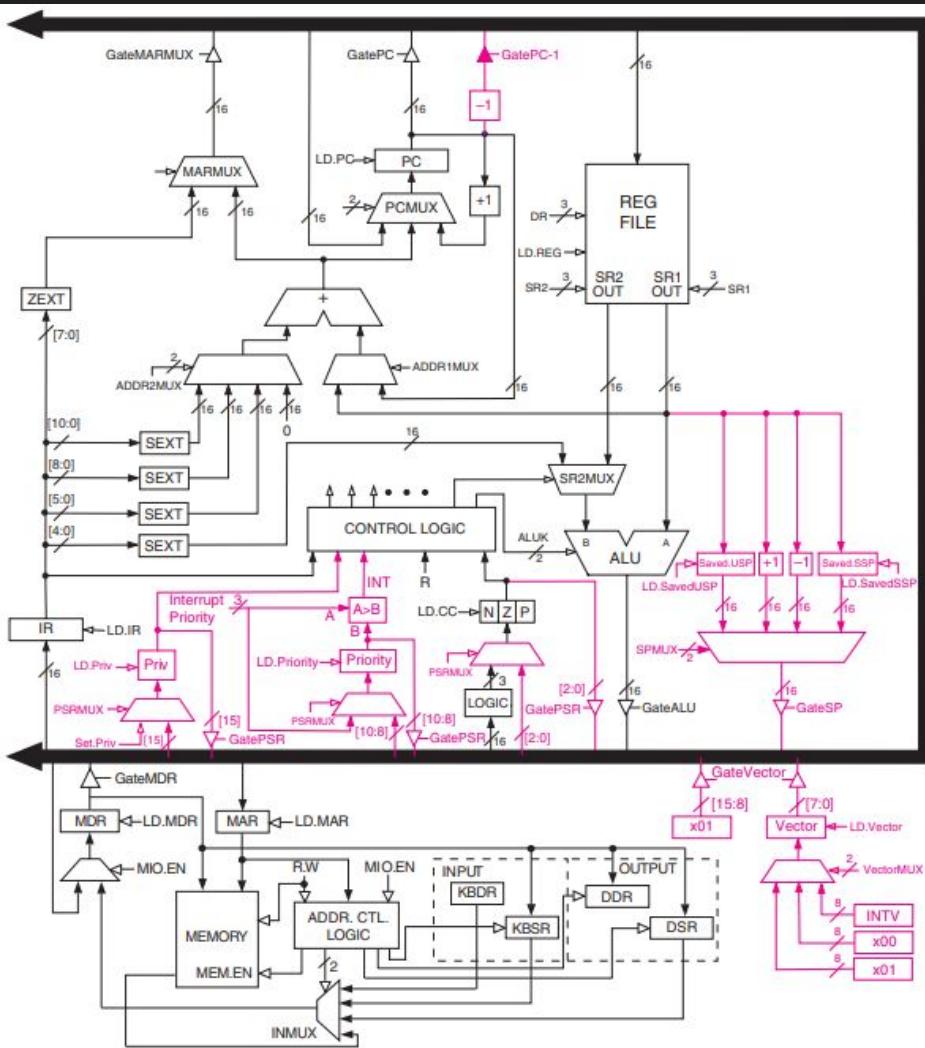
CEG3310/5310 - Computer Organization  
Max Gilson

# LC3 Hardware and Control Signals

- Looks complicated!!!
- We will break it down piece by piece

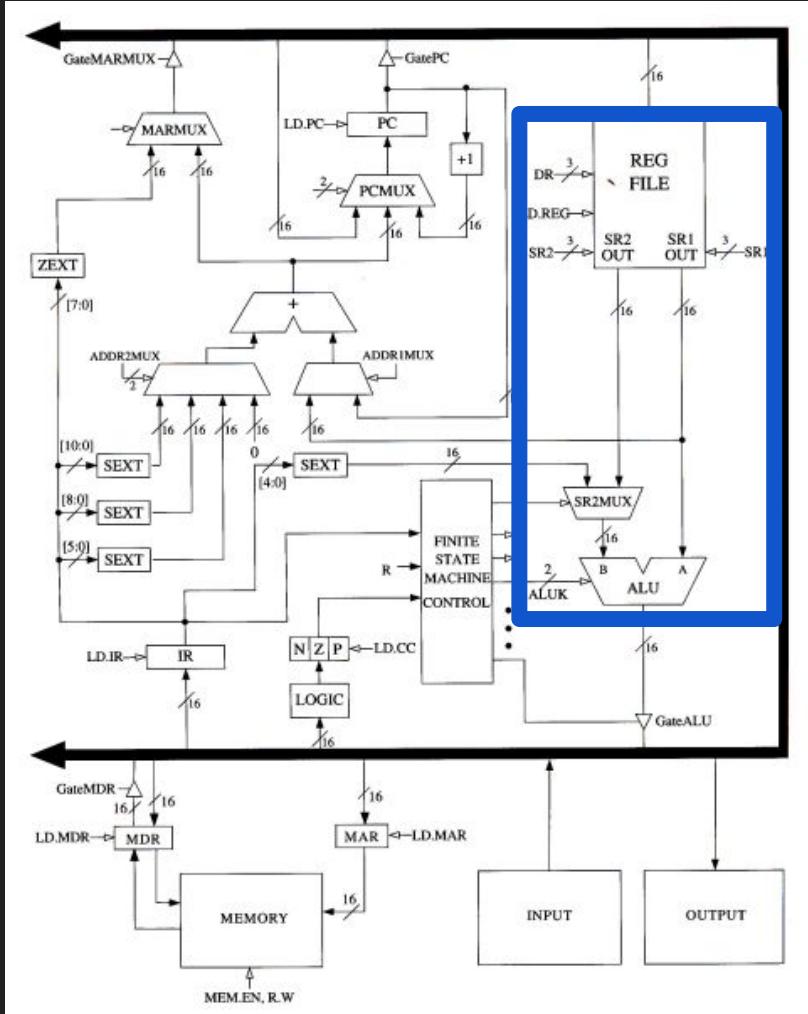


# LC3 Hardware and Control Signals Even More Complex



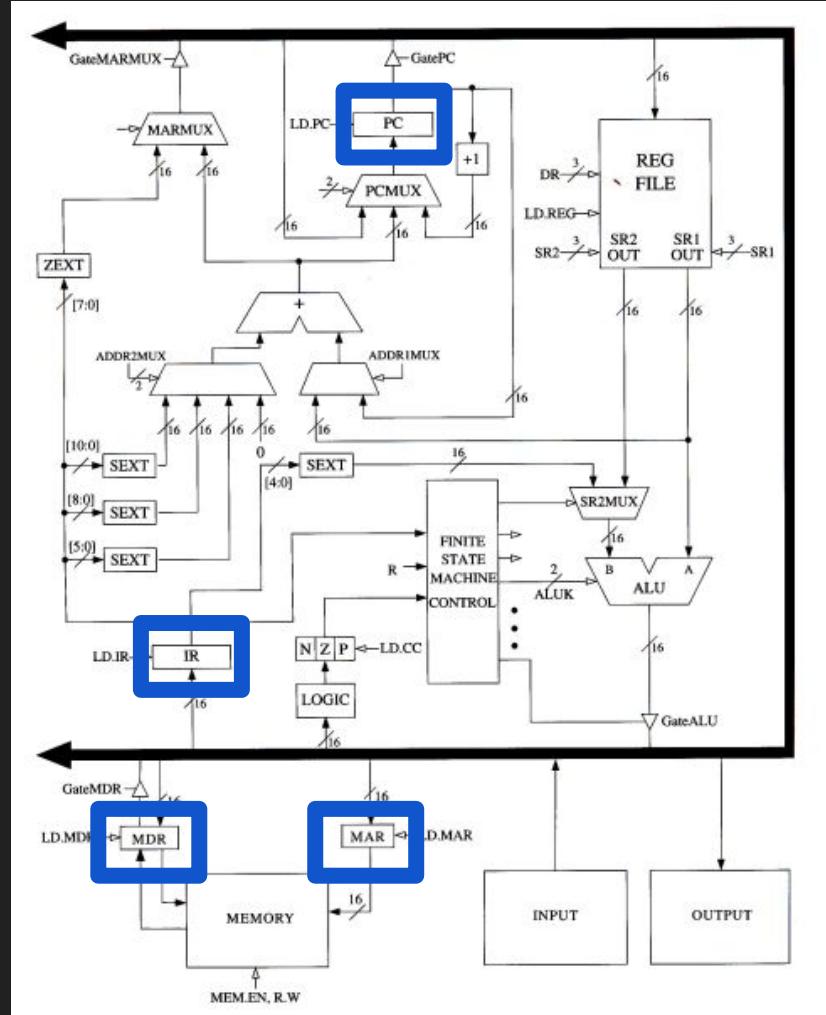
# LC3 Hardware and Control Signals (Cont.)

- Arithmetic Logic Unit (ALU)
  - Performs ADD, AND, NOT operations
- Register File (REG FILE)
  - Provides/stores the data from the registers for the operations
- SR2MUX
  - Selects between immediate or register value for instruction



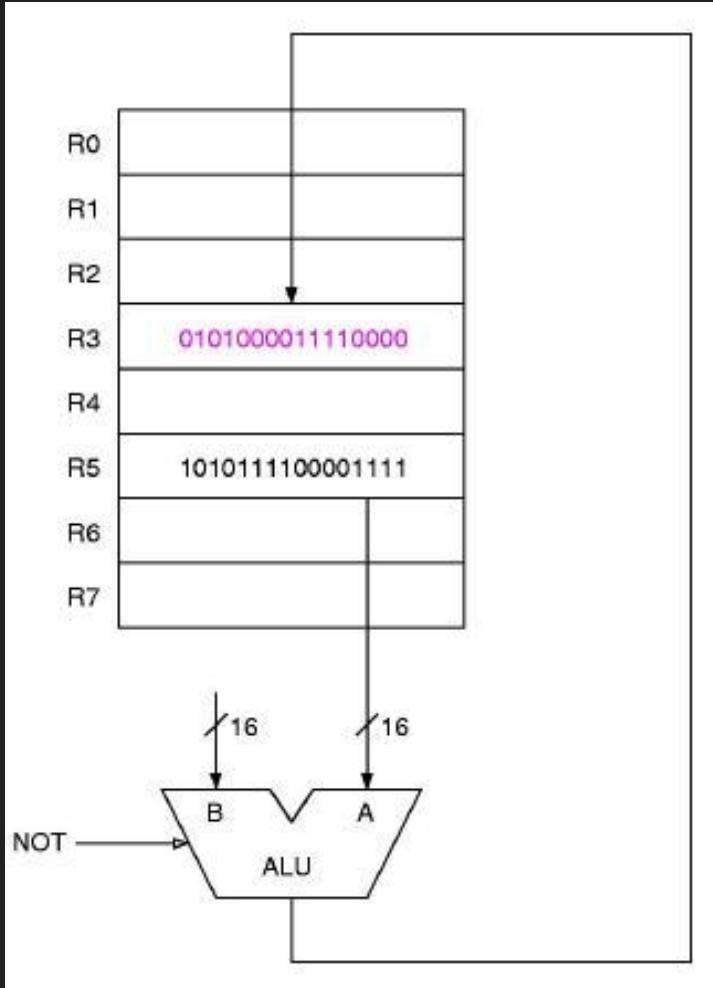
# LC3 Hardware and Control Signals (Cont.)

- Program Counter (PC)
  - Keeps track of what address our program is running
- Instruction Register (IR)
  - Holds the current instruction
- Memory Address Register (MAR)
  - Stores the address that we want to access from memory
- Memory Data Register (MDR)
  - Stores the data we read from memory or the data we want to write to memory



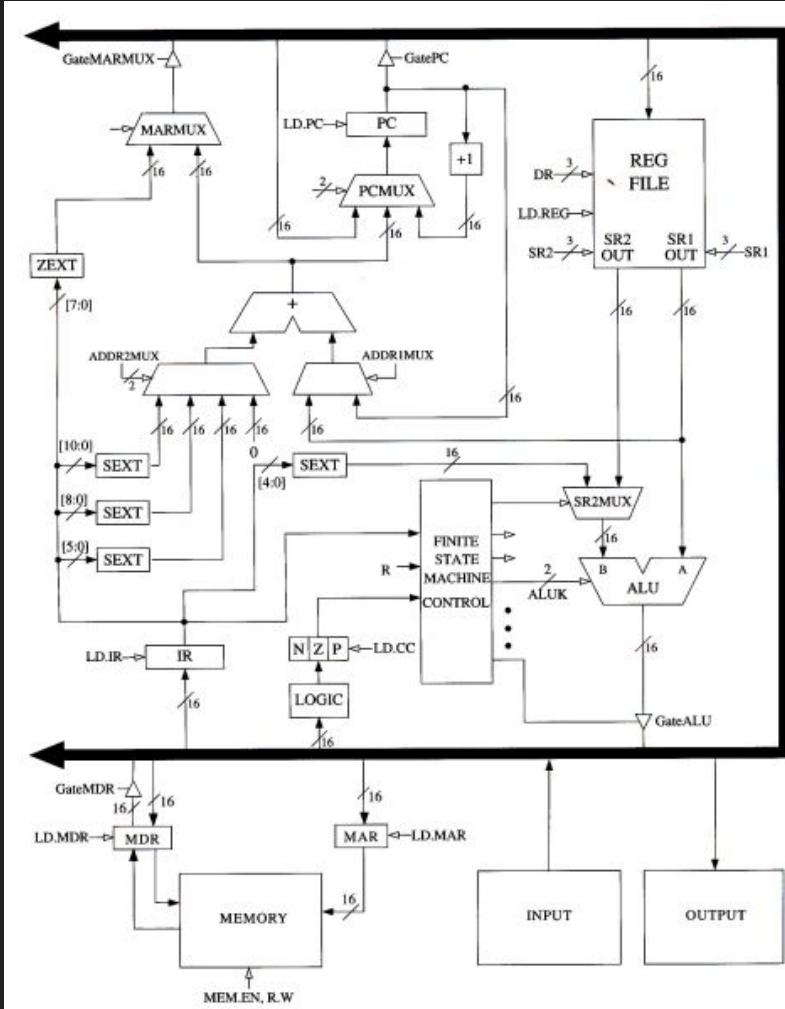
# LC3 Hardware and Control Signals NOT

- Unary Operation
  - Only uses one operand
- Assembler Instruction
  - NOT DR, SR
- Encoding
  - 1001 DR SR 111111
- Example
  - NOT R3, R5



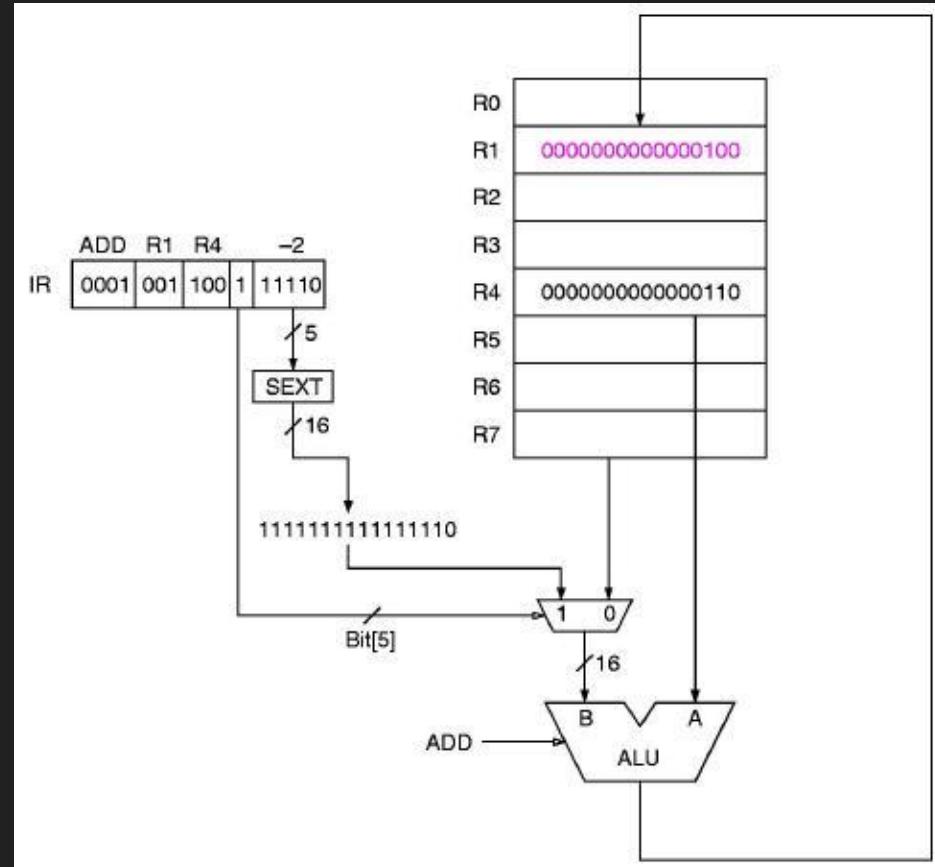
# LC3 Hardware and Control Signals NOT (Cont.)

- Control signal path for NOT R3, R5
  1. SR1 = 101
  2. ALUK = 10 (NOT)
  3. GateALU = 1
  4. DR = 011
  5. LD.REG = 1
  6. LD.CC = 1
- In English
  1. R5 is selected as SR1
  2. ALU is told to compute NOT operation
  3. GateALU allows the result of the NOT to be written onto main bus
  4. R3 is selected as DR
  5. LD.REG loads result into the DR
  6. LD.CC enables the N, Z, P bits



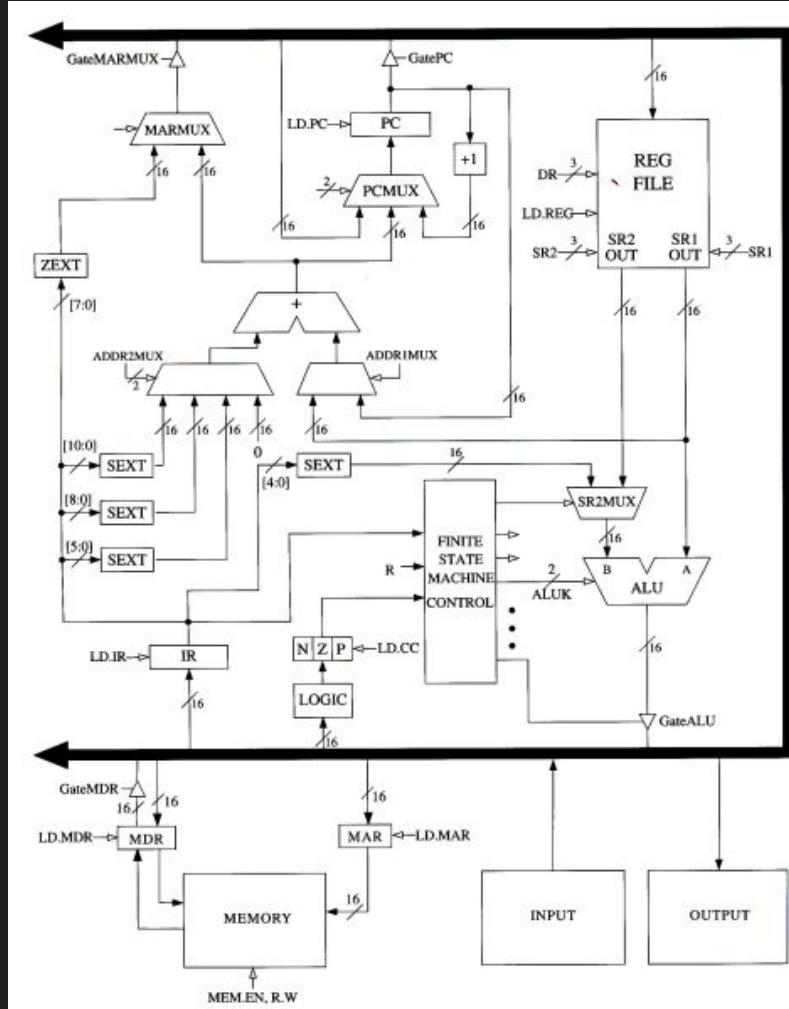
# LC3 Hardware and Control Signals ADD

- Binary Operation
  - Uses two operands
- Assembler Instruction
  - ADD DR, SR1, SR2
  - ADD DR, SR1, IMM5
- Encoding
  - 0001 DR SR1 0 00 SR2
  - 0001 DR SR1 1 IMM5
- Examples
  - ADD R1, R4, R5
  - ADD R1, R4, #-2
- NOTE: This is Two's Complement Addition!
- NOTE: SEXT sign extends the immediate number



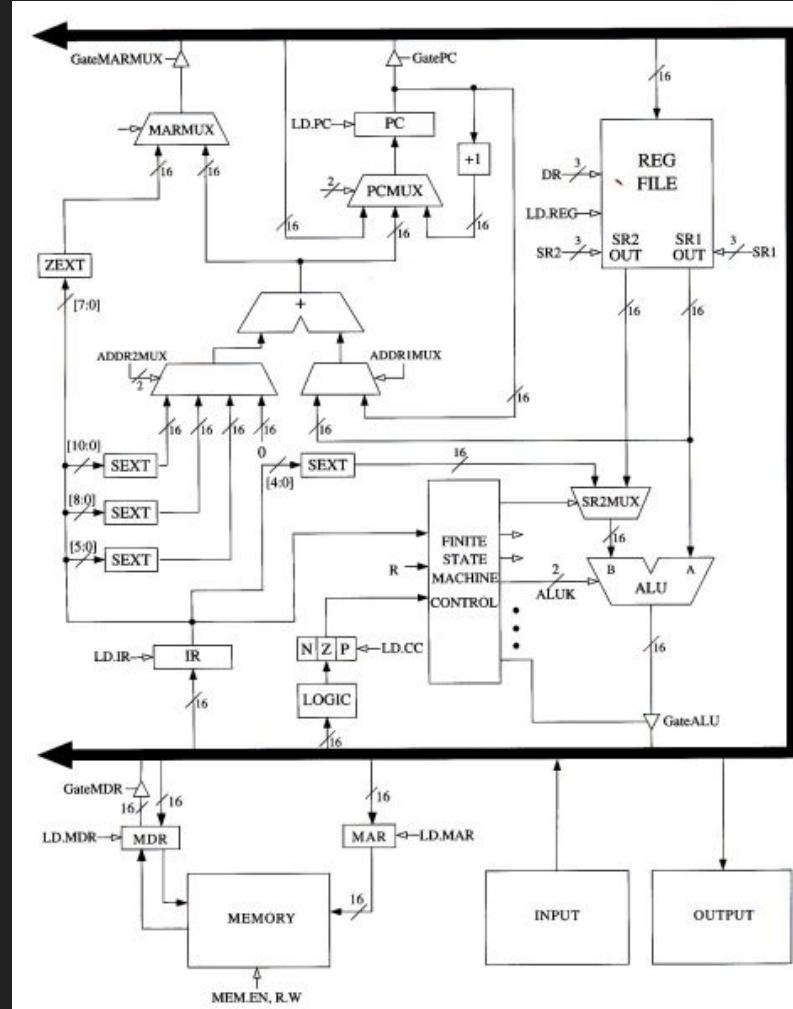
# LC3 Hardware and Control Signals ADD (Cont.)

- Control signal path for ADD R1, R4, -2
  - SR1 = 100
  - SR2MUX = 0
  - ALUK = 00 (ADD)
  - GateALU = 1
  - DR = 001
  - LD.REG = 1
  - LD.CC = 1
- In English
  - R4 is selected as SR1
  - SR2MUX selects the immediate number as second operand (-2)
  - ALU is told to compute ADD operation
  - GateALU allows the result of the ADD to be written onto main bus
  - R1 is selected as DR
  - LD.REG loads result into the DR
  - LD.CC enables the N, Z, P bits



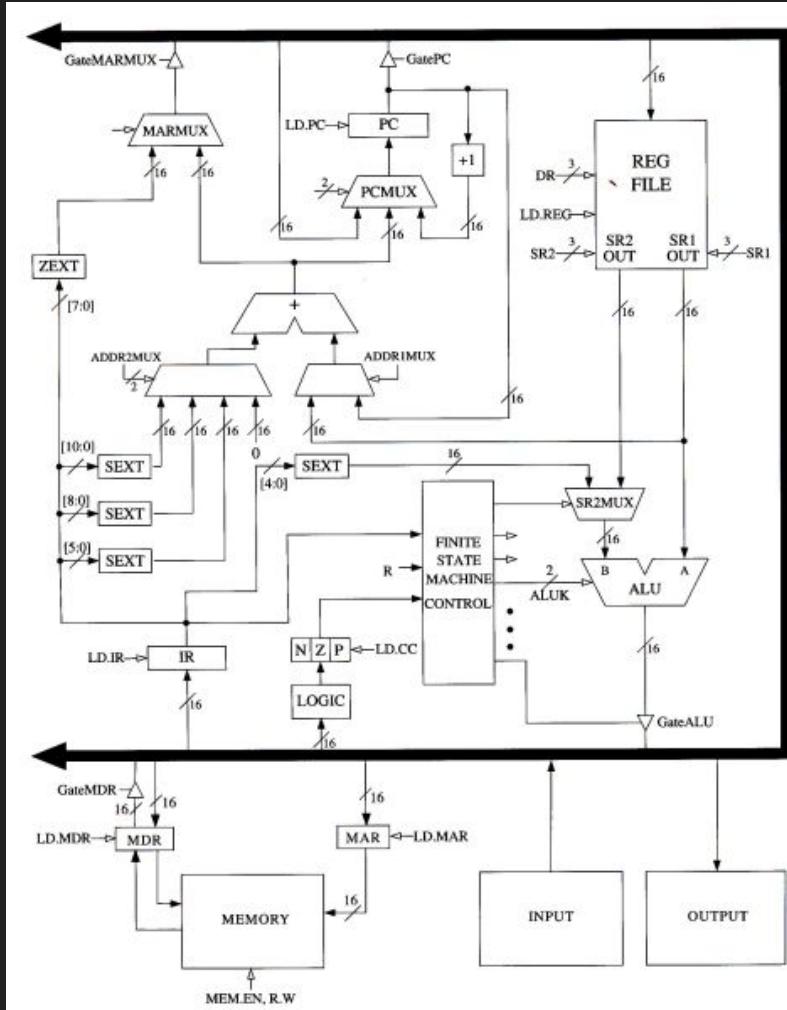
# LC3 Hardware and Control Signals ADD (Cont.)

- Control signal path for ADD R1, R4, R5
  - SR1 = 100
  - SR2 = 101
  - SR2MUX = 1
  - ALUK = 00 (ADD)
  - GateALU = 1
  - DR = 001
  - LD.REG = 1
  - LD.CC = 1
- In English
  - R4 is selected as SR1
  - R5 is selected as SR2
  - SR2MUX selects the second register as second operand (R5)
  - ALU is told to compute ADD operation
  - GateALU allows the result of the ADD to be written onto main bus
  - R1 is selected as DR
  - LD.REG loads result into the DR
  - LD.CC enables the N, Z, P bits



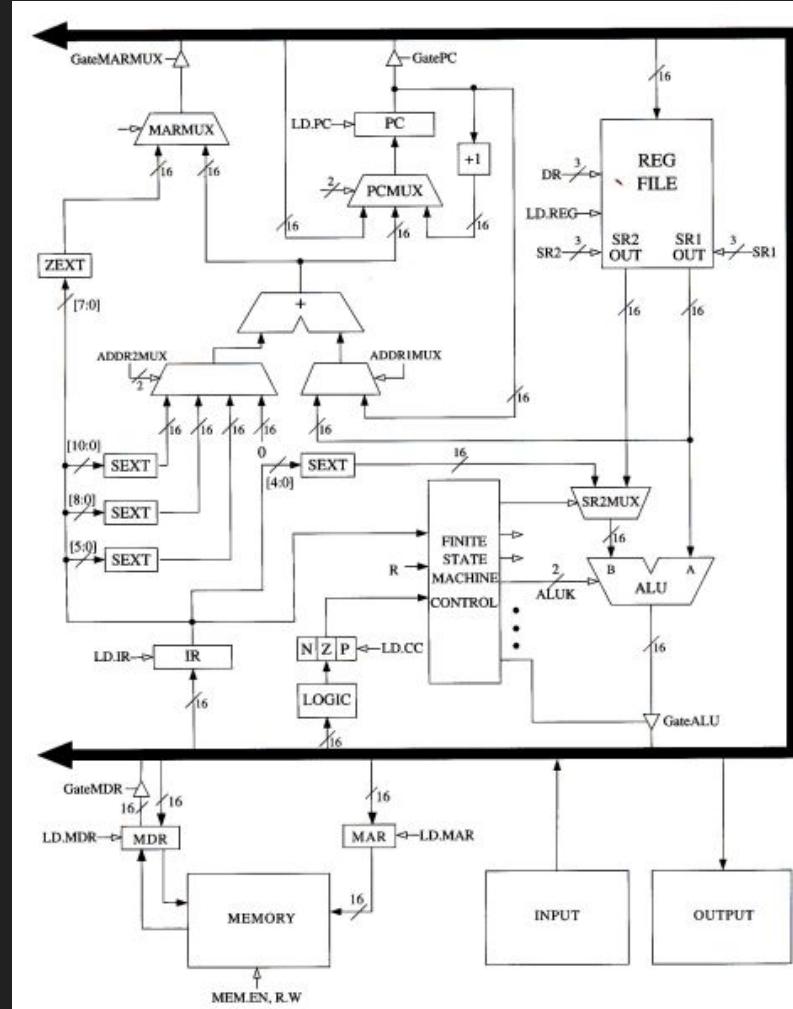
# LC3 Hardware and Control Signals AND (Cont.)

- Control signal path for AND R3, R2, #0
  - SR1 = 010
  - SR2MUX = 0
  - ALUK = 01 (AND)
  - GateALU = 1
  - DR = 011
  - LD.REG = 1
  - LD.CC = 1
- In English
  - R2 is selected as SR1
  - SR2MUX selects the immediate number as second operand (0)
  - ALU is told to compute AND operation
  - GateALU allows the result of the AND to be written onto main bus
  - R3 is selected as DR
  - LD.REG loads result into the DR
  - LD.CC enables the N, Z, P bits



# LC3 Hardware and Control Signals AND (Cont.)

- Control signal path for AND R2, R3, R6
  - SR1 = 011
  - SR2 = 110
  - SR2MUX = 1
  - ALUK = 01 (AND)
  - GateALU = 1
  - DR = 010
  - LD.REG = 1
  - LD.CC = 1
- In English
  - R3 is selected as SR1
  - R6 is selected as SR2
  - SR2MUX selects the second register as second operand (R6)
  - ALU is told to compute AND operation
  - GateALU allows the result of the AND to be written onto main bus
  - R2 is selected as DR
  - LD.REG loads result into the DR
  - LD.CC enables the N, Z, P bits

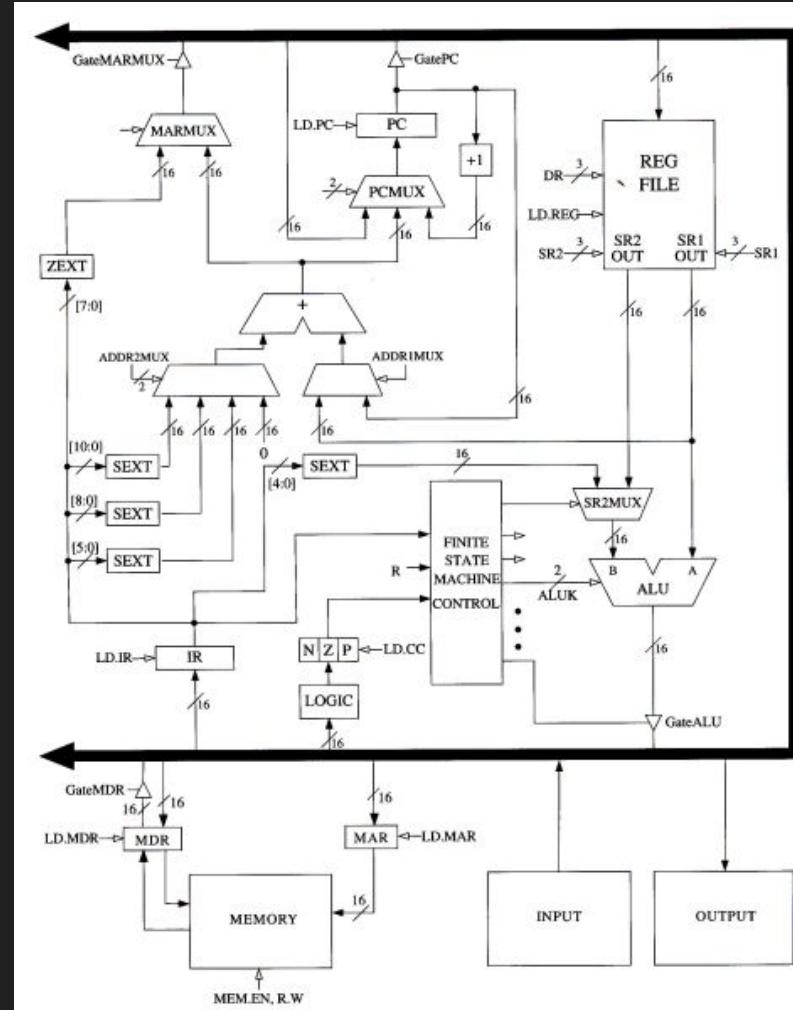


# That is just a nutshell...

- Since we now know much more about control signals we will have to look at all 6 steps of the instruction cycle:
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result
- Let's start with:
  - ADD R1, R4, R5

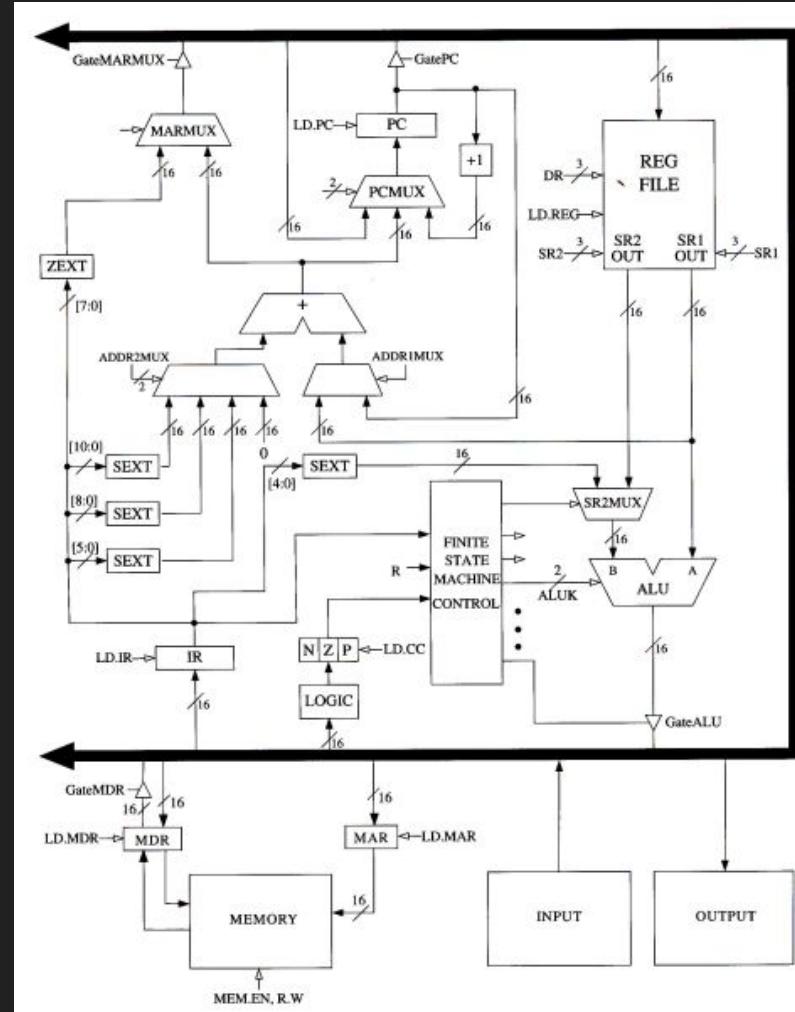
# LC3 Complete Instruction Cycle

- First we have to learn how to fetch a new instruction:
  1. GatePC = 1
  2. LD.MAR = 1
  3. MEM.EN = 1
  4. R.W = 1
  5. LD.MDR = 1
  6. GateMDR = 1
  7. LD.IR = 1
  8. LD.PC = 1
- In English:
  1. Allow Program Counter (PC) to be written to bus
  2. Copy PC into Memory Address Register (MAR)
  3. Enable Memory
  4. Enable Reading from memory
  5. Read data from MAR into Memory Data Register (MDR)
  6. Write MDR to the bus
  7. Write MDR into the Instruction Register (IR)
  8. Increment the PC by +1



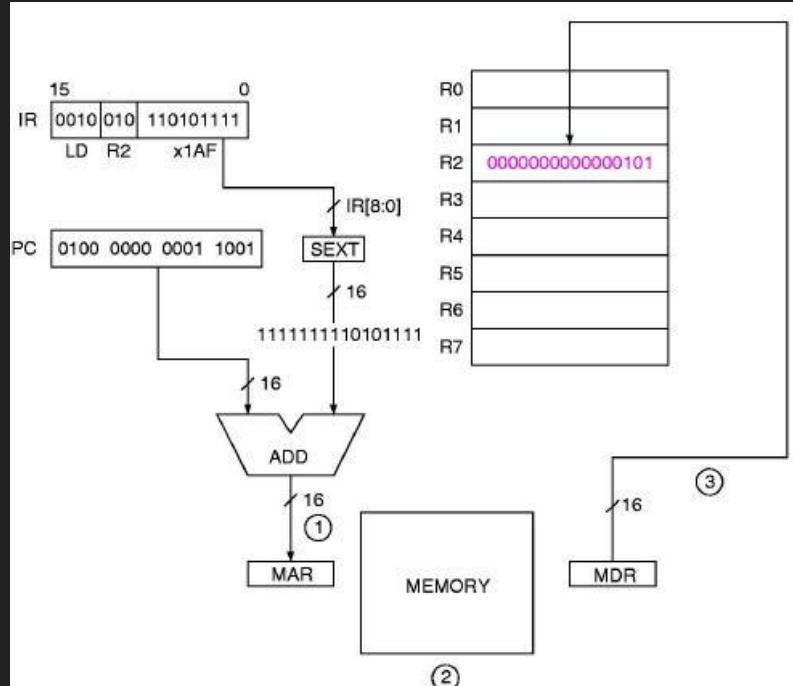
# LC3 Complete Instruction Cycle (Cont.)

- Now that we've fetched the instruction the ADD control signals can now be released:
  - SR1 = 100
  - SR2 = 101
  - SR2MUX = 1
  - ALUK = 00 (ADD)
  - GateALU = 1
  - DR = 001
  - LD.REG = 1
  - LD.CC = 1
- Hint: This was already covered on a previous slide



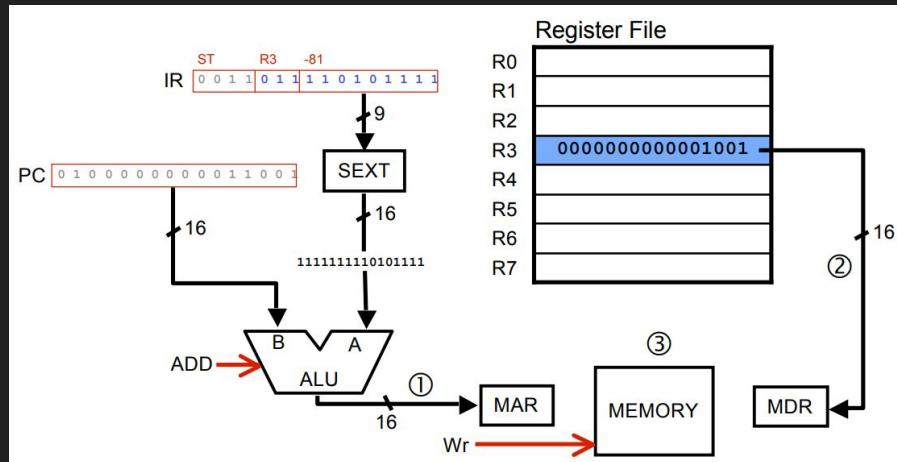
# LC3 Movement Instructions (Load Direct) (LD) (Cont.)

- Example:
  - LD R2, x1AF
  - Loads whatever value is in PC+1+x1AF into R2
  - NOTE: This does not use the same ALU that can ADD two registers!



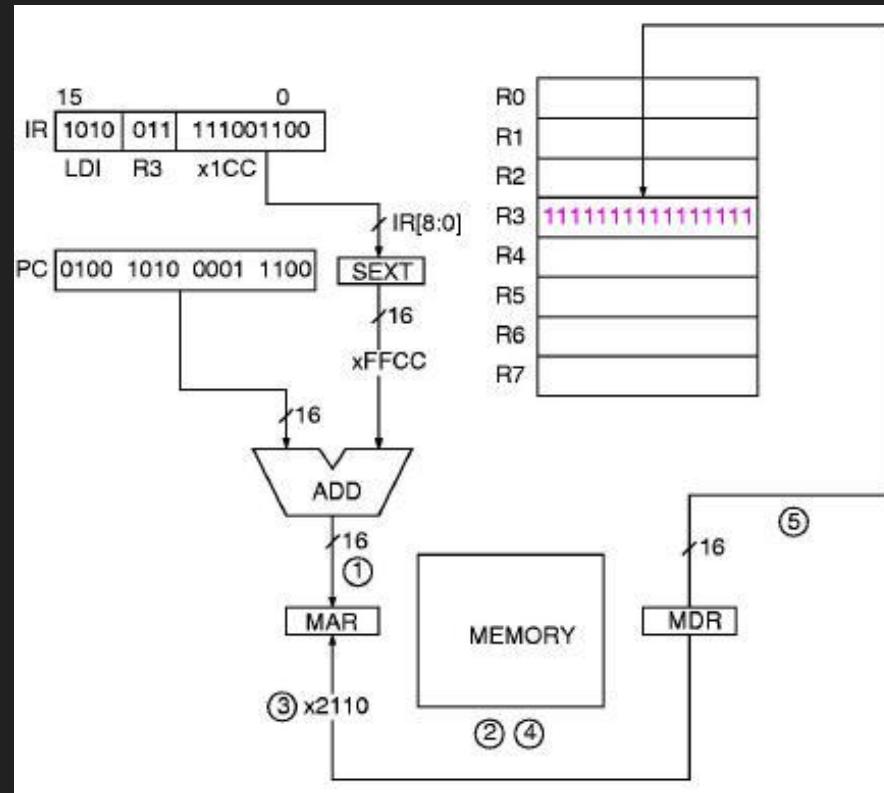
# LC3 Movement Instructions (Store Direct) (ST) (Cont.)

- Example:
  - ST R3, x1AF
  - Stores whatever value is in R3 into PC+1+x1AF



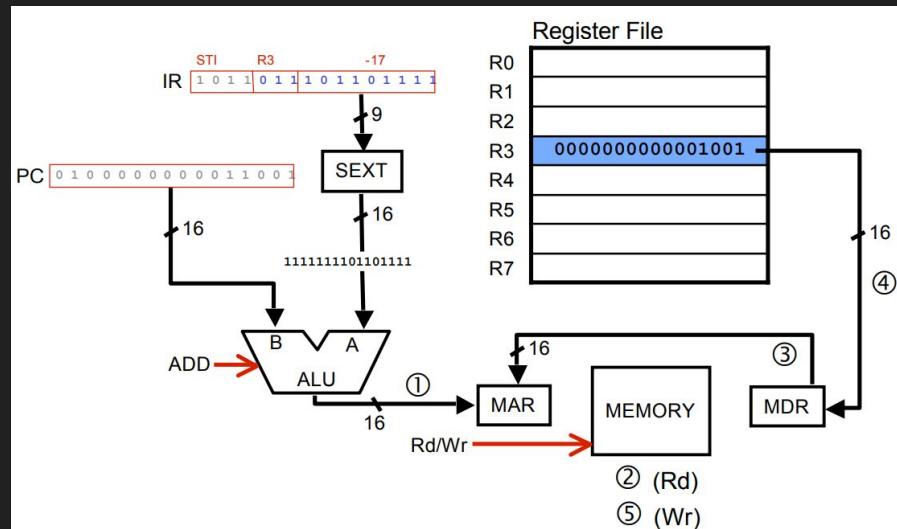
# LC3 Movement Instructions (Load Indirect) (LDI) (Cont.)

- Example:
  - LDI R3, x1CC
  - Retrieves an address stored in PC+1+x1CC
  - Retrieves data at that address
  - Loads that data into R3



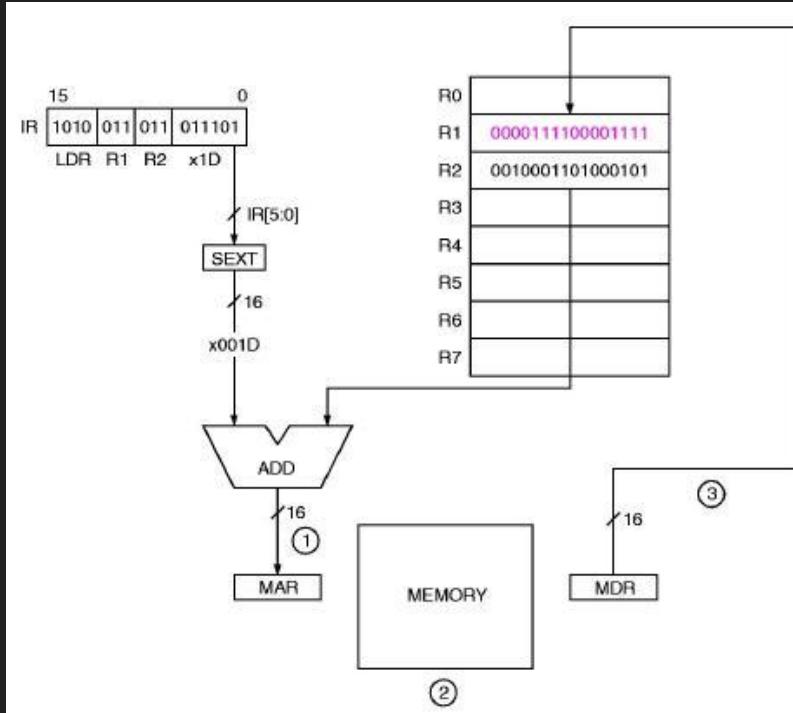
# LC3 Movement Instructions (Store Indirect) (STI) (Cont.)

- Example:
  - STI R3, x16F
  - Retrieves an address stored in PC+1+x16F
  - Stores data from R3 at that address



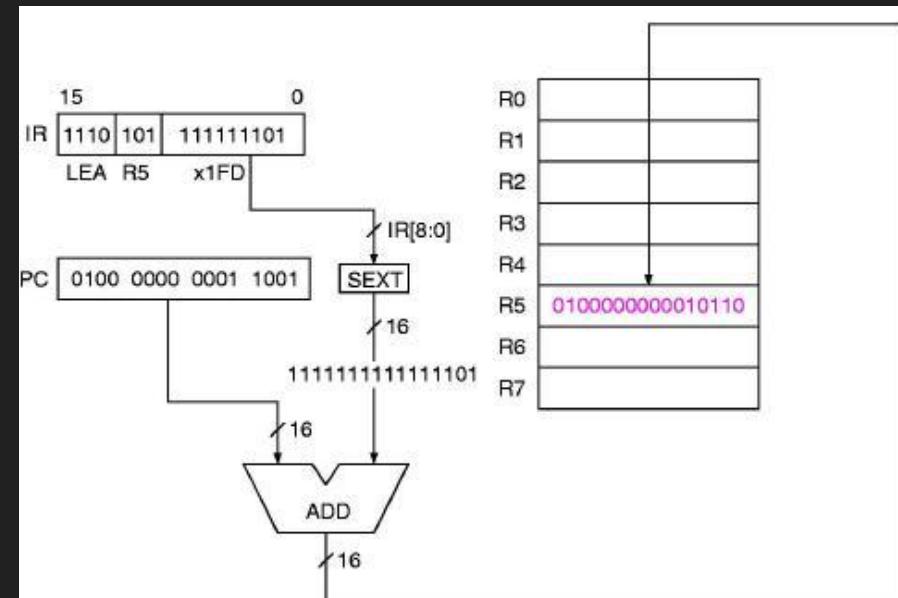
# LC3 Movement Instructions (Load Base+Offset) (LDI) (Cont.)

- Example:
  - LDR R1, R2, x1D
  - Adds x1D to value in R2
  - Retrieves data at location: x1D+R2
  - Loads that data into R1



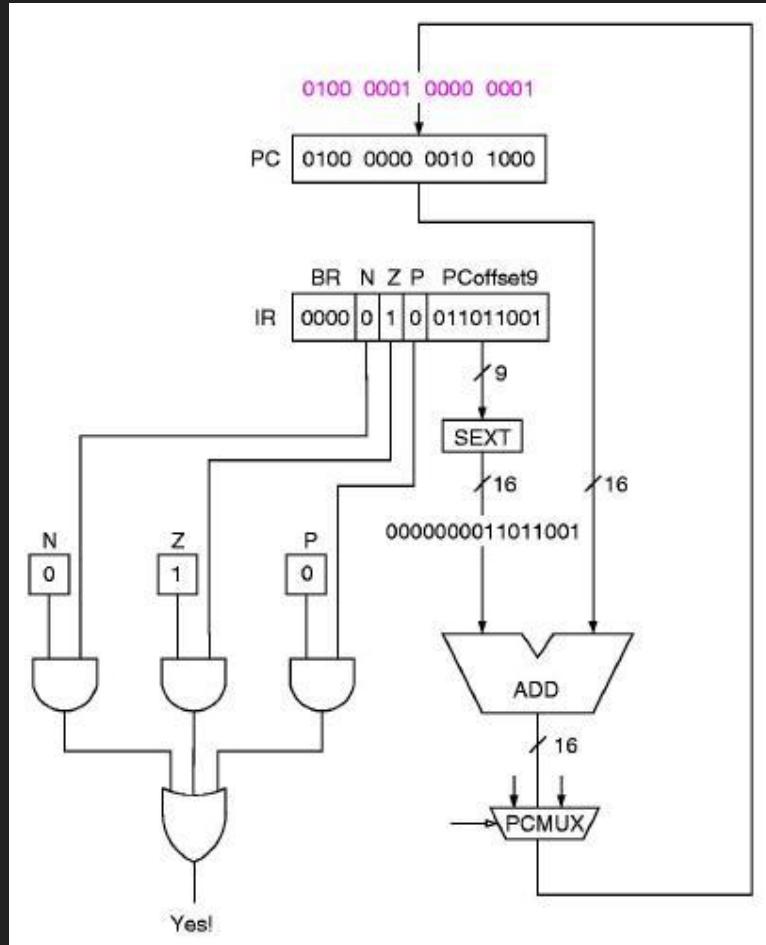
# LC3 Movement Instructions (Load Effective Address) (LEA) (Cont.)

- Example:
  - LEA R5, #-3
  - Loads the address at PC+1-3 into R5



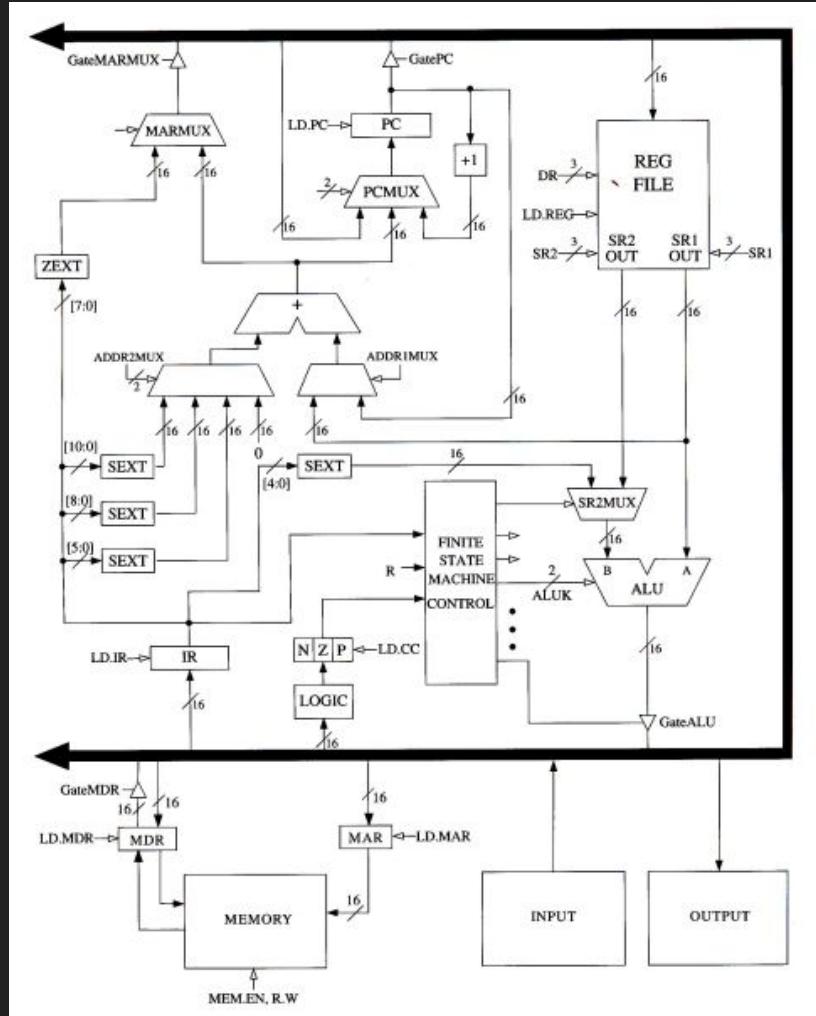
# LC3 Control Instructions (Conditional Branch) (BRX)

- Example:
  - BRz x0D9
  - Changes to PC to  $PC+1+x0D9$  if the previous operation resulted in a Z (zero)



# LC3 Control Instructions (Uncond. Branch) (JMP)

- Example:
  - JMP R5
  - Change PC to value in R5
  - We don't have a simplified drawing for this
  - Can we figure it out?



# Extra help

- Operate Instructions:
  - <https://www.youtube.com/watch?v=yZChqRqPluI>
- Control Signals:
  - <https://www.youtube.com/watch?v=6hhNNj67w1E>
- Load Instructions:
  - <https://www.youtube.com/watch?v=cDaPPXyYbH0>
  - <https://www.youtube.com/watch?v=359TeV9UvM8>

# 13 - Memory and Cache

CEG3310/5310 - Computer Organization  
Max Gilson

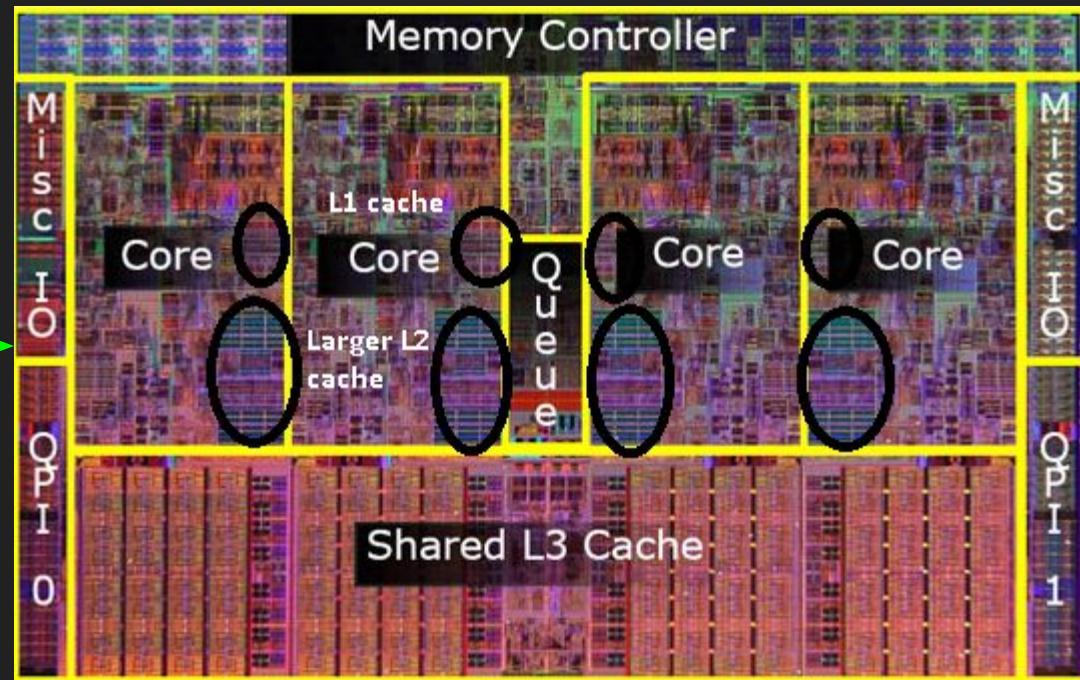
# Cache Memory

- Caches are memory storage devices that are inside the CPU chip itself
- Caches hold data that the CPU will most likely need to access from memory
- Caches are much faster than accessing RAM, but they contain very little data compared to RAM
- Caches sit in between the CPU and Memory

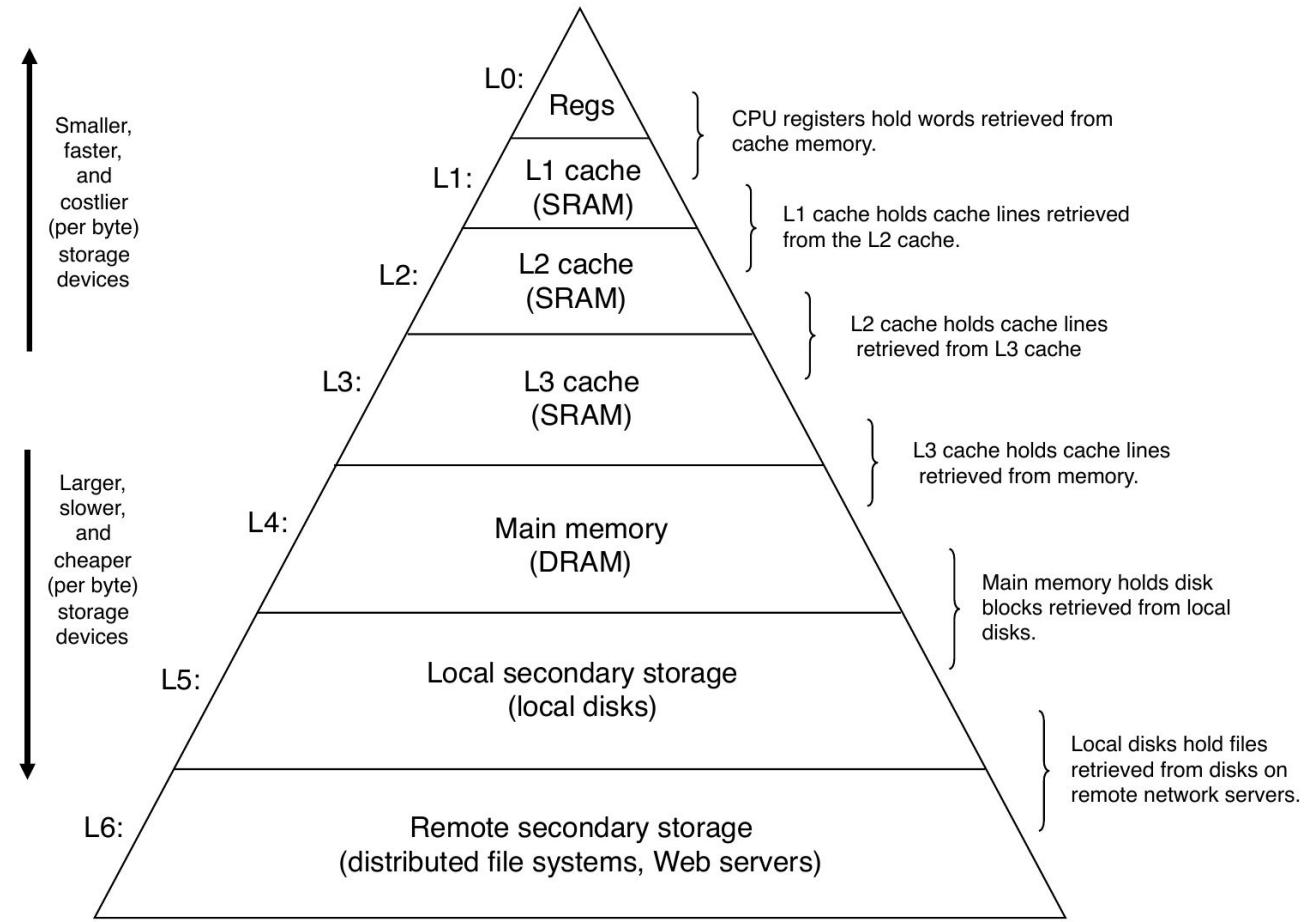
# Types of Caches

- In most modern CPUs there are 3 levels of cache
  - Level 1 Cache (L1) (100+ KBytes)
    - Fastest cache to access
    - Smallest memory capacity
  - Level 2 Cache (L2) (1 to 3+ MBytes)
    - Slower than L1 cache
    - More memory capacity than L1
  - Level 3 Cache (L3) (10 to 64+ MBytes)
    - Slowest cache
    - Most memory capacity
    - Typically shared between multiple CPU cores (shared cache)

# Where Are Caches on Modern CPUs?



# The Great Memory Hierarchy



# Locality of Reference

- Temporal Locality
  - Recently accessed data/instructions are likely to be accessed again
    - Example: In a loop, some of the variables/data will be accessed every time the loop runs
- Spatial Locality
  - Data that exists in nearby memory addresses are likely to be accessed in the future
    - Example: Instructions get accessed one instruction after the other (except for subroutines)
    - Example: Arrays may get accessed one element after the other
- In both of these cases, the cache memory becomes incredibly useful because we can access memory very quickly

# Caches Exploiting Temporal and Spatial Locality

- Temporal
  - When something is accessed from memory, it's also saved onto the cache
  - Then when the CPU needs to access it again, it'll go to the cache instead of the slow RAM
- Spatial
  - When something is accessed from memory, the next few memory locations are saved to the cache also
  - Then if the CPU needs to access the next item in memory, it'll go to the cache
  - The number of memory words that are accessed are called the cache line size or cache block size

# Fully-Associative Cache Mapping

- A fully-associative cache is the most simple caching algorithm
  - Benefit: No chance of thrashing (we'll talk about this)
  - Drawback: Expensive (more hardware to implement) and slow
- As an example, let's assume we are dealing with 2 byte memory addresses and 2 byte words (like the LC3)
  - i.e. Address: x43F6
  - holds the data: x0001
- Let's assume that we will use the 3 MSD's as the *tag* and the last LSD as the *word number*
  - In this case x43F is the tag and x6 is the word number

## Fully-Associative Cache Mapping (cont.)

- So far our cache looks like:

# Fully-Associative Cache Mapping

- When the address x43F6 is accessed from memory, it will bring it's nearby memory locations into the cache to fill up the cache block
- When we access x43F6 (which contains x0001) let's look what happens to the cache
- Let's also assume that nearby locations:
  - x43F5 contains xFFFF
  - x43F7 contains x1AA3

# Fully-Associative Cache Mapping (cont.)

- After accessing x43F6 (containing x0001), remember, x43F is the tag and x6 is the word number:

|            | Block (Word Numbers) |        |        |        |        |       |       |       |        |        |        |        |        |        |        |        |
|------------|----------------------|--------|--------|--------|--------|-------|-------|-------|--------|--------|--------|--------|--------|--------|--------|--------|
| Tag Number | 0                    | 1      | 2      | 3      | 4      | 5     | 6     | 7     | 8      | 9      | A      | B      | C      | D      | E      | F      |
| x43F       | x#####               | x##### | x##### | x##### | x##### | xFFFF | x0001 | x1AA3 | x##### |
|            |                      |        |        |        |        |       |       |       |        |        |        |        |        |        |        |        |
|            |                      |        |        |        |        |       |       |       |        |        |        |        |        |        |        |        |

- NOTE: x##### represents whatever data was in the memory addresses nearby

# Fully-Associative Cache Mapping

- Now if we access x53AF (containing x1111) we can add that too, and keep x43F# data from the previous access
- Let's assume that:
  - x53AE contains x0123
  - x53B0 contains x9999

# Fully-Associative Cache Mapping (cont.)

- After accessing x53AF (containing x1111):

|            | Block (Word Numbers) |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |
|------------|----------------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| Tag Number | 0                    | 1      | 2      | 3      | 4      | 5      | 6      | 7      | 8      | 9      | A      | B      | C      | D      | E      | F      |
| x43F       | x#####               | x##### | x##### | x##### | x##### | xFFFF  | x0001  | x1AA3  | x##### |
| x53A       | x#####               | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x0123  |
|            |                      |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |

- NOTICE: Even though x53B0 (containing x9999) was close to x53AF, it does not have the same tag number so it does not get pulled into the cache for easy access

# Fully-Associative Cache Mapping

- This helps incredibly, since now if we need any memory that is nearby  $x43F\#$  then we have it in the cache which we can access much faster than RAM
- This makes our programs run a lot faster since the data we need is conveniently stored in the cache
- What happens when the cache is full?

# Fully-Associative Cache Mapping (cont.)

- Let's say our cache is full:

|            | Block (Word Numbers) |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |
|------------|----------------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| Tag Number | 0                    | 1      | 2      | 3      | 4      | 5      | 6      | 7      | 8      | 9      | A      | B      | C      | D      | E      | F      |
| x43F       | x#####               | x##### | x##### | x##### | x##### | xFFFF  | x0001  | x1AA3  | x##### |
| x53A       | x#####               | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x1111  |
| x###       | x#####               | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### | x##### |

- What should we replace when we access a location in memory that is not already in the cache?
  - Replacement Algorithms

# Cache Replacement Algorithms

- If you need to add something to a full cache, you can remove old data from the cache with a replacement algorithm:
  - FIFO: the first thing that was added to the cache gets replaced (*first in -> first out*)
    - Problem: operating system data may be the first thing loaded!
  - LRU: the *least recently used* block in the cache gets replaced
  - LFU: the *least frequently used* block in the cache gets replaced
  - Random: randomly select a block in the cache to be replaced
    - Efficient for cost and power consumption (less hardware to implement this)

# Cache Replacement Algorithms (cont.)

- Replacement algorithms include engineering trade-offs
  - Random
    - low cost to implement, because most computers have random number generators already
    - May not have the best performance in real world
  - Least Recently Used
    - Very useful because intuitively, the stuff in the cache that hasn't been used in a while gets removed
    - Costs more and requires more components to implement (timers, counters, comparators, for ea. tag)

# Other Types of Caches

- Other types of cache formats/architectures:
  - Direct Mapped Cache
  - Set Associative Cache
- There are performance/cost trade-offs for incorporating FA, DM, or SA cache designs

# Cache Writing

- So far, we've discussed reading memory locations into the cache
- What if we want to write something back? Should it be written to the cache or to memory?
  - Write through cache:
    - write to both cache and main memory. Cache and memory are always consistent
  - Write back cache:
    - write only to cache and set a “dirty bit”. When the block gets replaced from the cache, write it out to memory.

# DMA and Cache

- Direct Memory Access (DMA) is when I/O devices can independently read/write memory without involving the CPU
  - This differs from Memory Mapped I/O (LC3) where the CPU has to manipulate control signals to read/write I/O to memory
- If we have a write-back cache, DMA output devices must check the cache before attempting to read memory
  - DMA outputs can trigger a cache flush, forcing memory to be updated before reading
- If there is a value in a cache that was copied from memory, and a DMA input device modifies the memory afterwards, we now have *stale data* in the cache
  - Caches blocks can have a “valid bit” which will be triggered whenever a DMA input changes memory, which forces the CPU to pull the fresh data into the cache

# Measuring Cache Performance

- Let's assume without a cache accessing memory requires 10 clock cycles
- For a simple cache:
  - $t_{avg} = h*C + (1-h)*M$
  - $C$  = time to access cache = 1 clock cycle
  - $M$  = miss penalty (cycles to load an entire cache block) = 17
  - $h$  = hit rate = 90%
  - $t_{avg} = 0.9*1 + (1-0.9)*17 = 2.6$  cycles per access
- As we can see, using a cache is a lot faster than the 10 clock cycles needed to read from RAM

# Measuring Cache Performance (cont.)

- For multi-level cache:
  - $t_{avg} = h_1 C_1 + (1-h_1)h_2 C_2 + (1-h_1)(1-h_2)M$
  - $h_1$  = hit rate in primary cache
  - $h_2$  = hit rate in secondary cache
  - $C_1$  = time to access primary cache
  - $C_2$  = time to access secondary cache
  - $M$  = miss penalty (time to load an entire cache block from main memory)
- $t_{avg} = 0.25*1+(1-0.25)*0.50*1+(1-0.25)(1-0.50)*17=7$  cycles per access
- Even with really poor hit rates, a 2 level cache outperforms DRAM needing 10 cycles per access

# Memory Interleaving

- The electrical design of DRAM requires capacitors that need to be recharged by a voltage in order to maintain memory
- Problem: CPU cannot read from memory when this recharge is occurring, this slows everything down
- Solution: Create multiple banks of memory so you can recharge a previous bank and read/write to the next, this allows for really fast communication between CPU and Memory

# Memory Interleaving (Cont.)

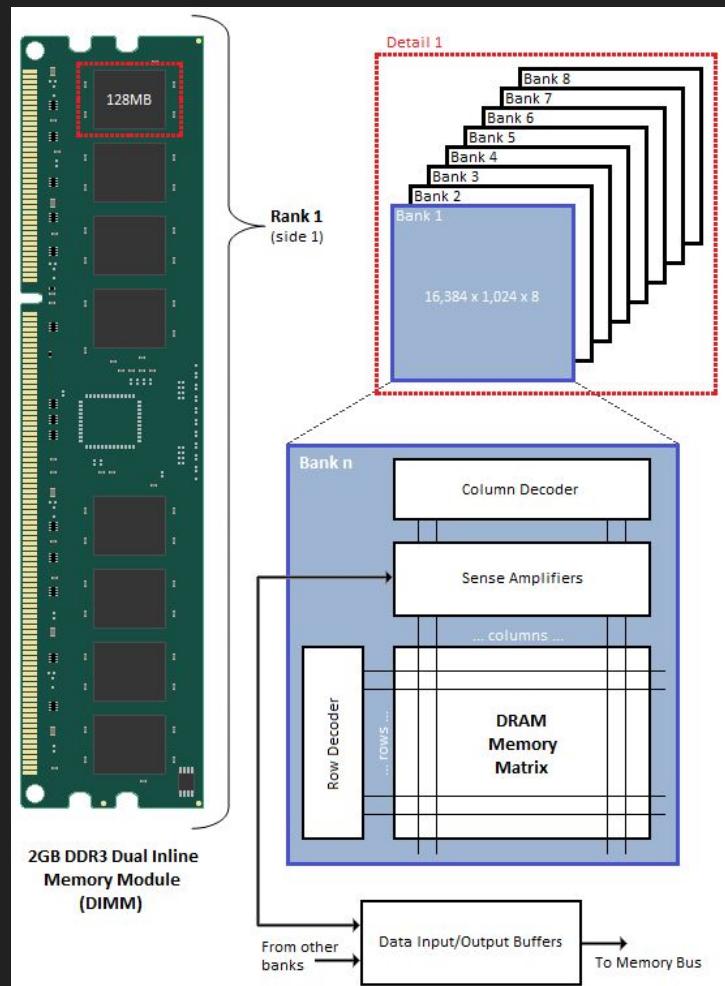
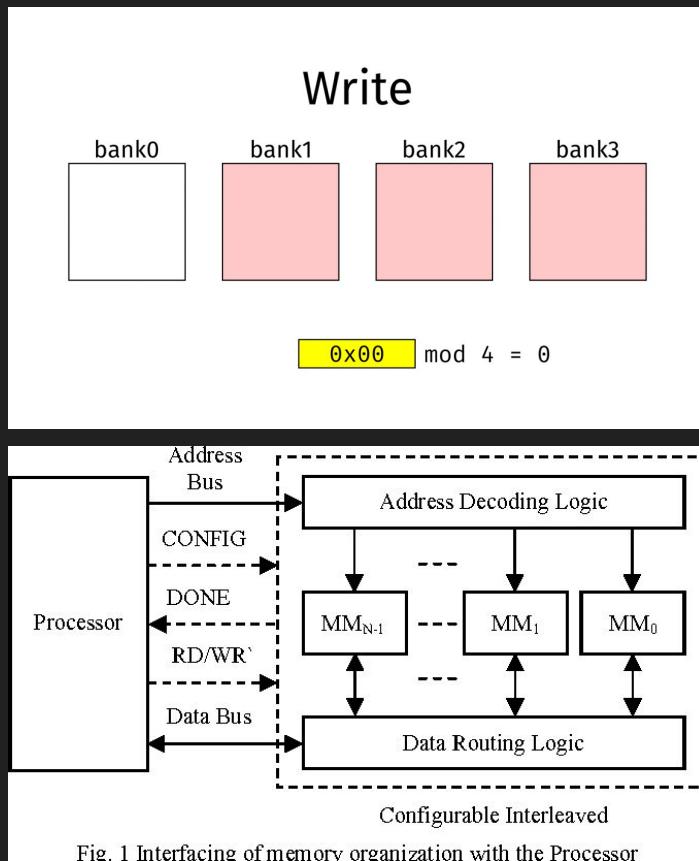


Fig. 1 Interfacing of memory organization with the Processor

# Virtual Memory

- Modern 64-bit systems are able to reference  $2^{64}$  memory locations
  - Technically, x86-64 allows for  $2^{48}$  virtual memory locations and  $2^{52}$  physical memory locations
- Consider a system that can reference  $2^{64}$  memory locations, that's 18,446,744 terrabytes or 18,446,744,073 gigabytes of byte-addressable memory
- Have you ever seen a computer with 18,446,744 TB of RAM?

# Virtual Memory (cont.)

- Since we can reference all these memory locations, but we can't make physical memory that large we have wasted potential
  - Hence, virtual memory!
- Memory that is not being used can be stored onto the hard drive/local disk/ssd and can be identified with a “memory address” even though it's not stored in RAM
- Your local storage acts as a *cache* in a way

# Working Set

- Not all of a program needs to be in memory while you are executing it:
  - Error handling routines are not called very often.
  - A menu screen generally only happens at the start of the game... why keep that code easily available?
- Working set is the memory that is consumed at any moment by a program while it is running.
  - Includes stack, allocated memory, active instructions, etc.

# Paging and Thrashing

- As you can imagine, having valuable memory stored outside of RAM can have its downsides
  - Recall the memory hierarchy!
- Paging
  - Sending/receiving memory between secondary storage and RAM
  - The blocks of memory passed between your local storage and RAM are called *pages*, when a page exists in RAM your program can access it very quickly
- Thrashing
  - When you spend more time paging in your program than actually executing useful code

# Page Faults

- Your operating system will manage various programs that are running and controls which programs are currently stored in RAM versus in virtual memory
  - Have you ever had a program minimized for a long time, only to open it again and it take forever to load?
- Page faults
  - Your program tries to access memory that is not currently in RAM but is in virtual memory
  - The operating system must then save a page that is in RAM to virtual memory (to make some room)
  - The operating system must then retrieve the desired page from virtual memory and put it in RAM

# Improving Your Virtual Memory

- To improve the performance of virtual memory you can try a few things:
  - Run fewer programs
    - The more programs you run, the more (potentially) that has to be sent/received between virtual memory and RAM
  - Writing programs that use memory more efficiently
    - Programs that are very memory inefficient will cause your system to run slow due to page faults or thrashing
  - Add RAM to the system
    - More RAM gives you more room in physical memory to run programs
  - Increasing the swap size
    - Increase the size of each page

# The Reality of Memory

- Memory Matters:
  - Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
  - Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
  - Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

# 15 - x86 Instruction Set Architecture

CEG3310/5310 - Computer Organization  
Max Gilson

# “Modern” Computing

- Modern CPUs have roots in x86 architecture
- Dates back to the 8086 CPU (1978)

## INTEL 8086 AND CORE i7-8086K

A 40 YEAR COMPARISON



Copyright © 2018 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

# x86 Family of Processors

- CISC Architecture
  - Complex Instruction Set Computer
    - Meaning, single instructions can carry out complex and multistage tasks
  - Supports both 32 and 64 bit operation
- Register Memory Machines
  - Operands can be in either registers or memory
- We will focus on 32 bit operations

# Modes of Operation

- x86 ISA allows for multiple modes of operation
  - Protected mode
  - Real-address mode
  - System management mode
  - Virtual-8086 mode
- We will focus on protected mode

# 32-Bit Operation

- The basic (default) sizes of the following are all 32 bits:
  - Registers
  - Data
  - Memory Addresses
- 32 bit byte addressable memory addresses implies up to 4GB of addressable memory (remember when we talked about virtual memory)

# x86 Registers

- General Purpose Registers are available
  - These are similar to R0 - R7 in LC3
- Some registers have special purposes
- Assembly language for x86 processors will reference these registers by name

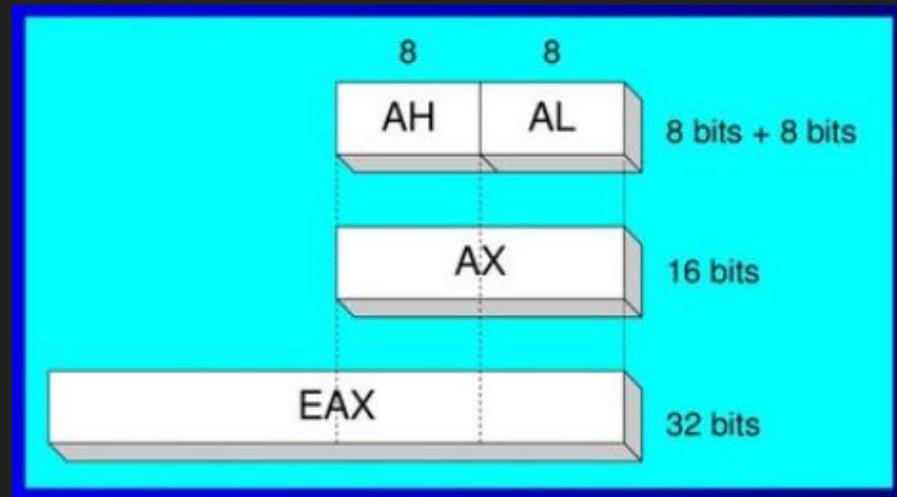
# General Purpose Registers

- 32-bit GPRs
  - EAX - EDX
  - EBP - EDI
- 16-bit Segment Registers
  - CS - GS
- 32-bit Status/System Registers
  - EFLAGS
  - EIP

|        |     |
|--------|-----|
| EAX    | EBP |
| EBX    | ESP |
| ECX    | ESI |
| EDX    | EDI |
| CS     | ES  |
| SS     | FS  |
| DS     | GS  |
| EFLAGS |     |
|        | EIP |

# Accessing GPRs

- 32-bit, 16-bit, 8-bit, segments can be accessed
- Applies to:
  - EAX
  - EBX
  - ECX
  - EDX



| 32-bit | 16-bit | 8-bit (high) | 8-bit (low) |
|--------|--------|--------------|-------------|
| EAX    | AX     | AH           | AL          |
| EBX    | BX     | BH           | BL          |
| ECX    | CX     | CH           | CL          |
| EDX    | DX     | DH           | DL          |

# Accessing GPRs

- Some registers only have either 32-bit or 16-bit segments:
  - ESI
  - EDI
  - EBP
  - ESP

| 32-bit | 16-bit |
|--------|--------|
| ESI    | SI     |
| EDI    | DI     |
| EBP    | BP     |
| ESP    | SP     |

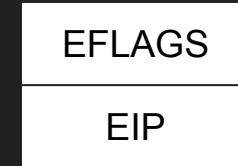
# Specialized Register Uses

- GPRs
  - EAX - mathematical results
  - ECX - loop counter
  - ESP - stack pointer
  - ESI - memory indexing (source)
  - EDI - memory indexing (destination)
  - EBP - frame pointer
- Segment Registers (pointers)
  - CS - code segment
  - DS - data segment
  - SS - stack segment
  - ES/FS/GS - additional segments

|     |     |
|-----|-----|
| EAX | EBP |
| EBX | ESP |
| ECX | ESI |
| EDX | EDI |
| CS  | ES  |
| SS  | FS  |
| DS  | GS  |

# Specialized Register Uses

- Status/System Registers
  - EIP - instruction pointer (like PC)
  - EFLAGS - status and control flags



# Status Flags

- Carry
  - Unsigned arithmetic out of range
- Overflow
  - Signed arithmetic out of range
- Sign
  - Negative result (N)
- Zero
  - Zero result (Z)
- Auxiliary Carry
  - Carry from bit 3 to bit 4 (carry from first 4 bits to second 4 bits)
- Parity
  - If sum of bits in least significant byte is even

# More Registers!

- Eight 80-bit floating point data registers
  - ST(0) - ST(7)
  - Arranged in a stack
  - Used for all float arithmetic
- Eight 64-bit MMX registers
  - MMX stand for *nothing*
  - Used for processing multiple data points with one instruction/register
  - Single-Instruction Multiple-Data (SIMD) operations
- Eight 128-bit XMM registers
  - SIMD

# Overview (x86 32-bit + x86 64-bit)

|       |       |       |       |       |       |       |       |        |       |       |       |       |        |        |      |      |      |      |      |     |       |      |      |  |
|-------|-------|-------|-------|-------|-------|-------|-------|--------|-------|-------|-------|-------|--------|--------|------|------|------|------|------|-----|-------|------|------|--|
| ZMM0  | YMM0  | XMM0  | ZMM1  | YMM1  | XMM1  | ST(0) | MM0   | ST(1)  | MM1   | ALAH  | AXEAX | RAX   | R8B    | R8W    | R8D  | R8   | R12B | R12W | R12D | R12 | MSW   | CR0  | CR4  |  |
| ZMM2  | YMM2  | XMM2  | ZMM3  | YMM3  | XMM3  | ST(2) | MM2   | ST(3)  | MM3   | BLBH  | BXEBX | RBX   | R9B    | R9W    | R9D  | R9   | R13B | R13W | R13D | R13 | CR1   | CR5  |      |  |
| ZMM4  | YMM4  | XMM4  | ZMM5  | YMM5  | XMM5  | ST(4) | MM4   | ST(5)  | MM5   | CLCH  | CXECX | RCX   | R10B   | R10W   | R10D | R10  | R14B | R14W | R14D | R14 | CR2   | CR6  |      |  |
| ZMM6  | YMM6  | XMM6  | ZMM7  | YMM7  | XMM7  | ST(6) | MM6   | ST(7)  | MM7   | DLDH  | DXEDX | RDX   | R11B   | R11W   | R11D | R11  | R15B | R15W | R15D | R15 | CR3   | CR7  |      |  |
| ZMM8  | YMM8  | XMM8  | ZMM9  | YMM9  | XMM9  |       |       |        |       | BPL   | BPEBP | RPB   | DIL    | DI     | EDI  | RDI  |      | IP   | EIP  | RIP | MXCSR | CR8  |      |  |
| ZMM10 | YMM10 | XMM10 | ZMM11 | YMM11 | XMM11 | CW    | FP_IP | FP_DP  | FP_CS | SIL   | SI    | ESI   | RSI    | SPL    | SP   | ESP  | RSP  |      |      |     |       | CR9  |      |  |
| ZMM12 | YMM12 | XMM12 | ZMM13 | YMM13 | XMM13 | SW    |       |        |       |       |       |       |        |        |      |      |      |      |      |     |       | CR10 |      |  |
| ZMM14 | YMM14 | XMM14 | ZMM15 | YMM15 | XMM15 | TW    |       |        |       |       |       |       |        |        |      |      |      |      |      |     |       | CR11 |      |  |
| ZMM16 | ZMM17 | ZMM18 | ZMM19 | ZMM20 | ZMM21 | ZMM22 | ZMM23 | FP_DS  |       |       |       |       |        |        |      |      |      |      |      |     |       | CR12 |      |  |
| ZMM24 | ZMM25 | ZMM26 | ZMM27 | ZMM28 | ZMM29 | ZMM30 | ZMM31 | FP_OPC | FP_DP | FP_IP | CS    | SS    | DS     | GDTR   | IDTR | DR0  | DR6  |      |      |     |       | CR13 |      |  |
|       |       |       |       |       |       |       |       |        | ES    | FS    | GS    | TR    | LDTR   | DR1    | DR7  |      |      |      |      |     |       | CR14 |      |  |
|       |       |       |       |       |       |       |       |        |       |       |       | FLAGS | EFLAGS | RFLAGS | DR2  | DR8  |      |      |      |     |       |      | CR15 |  |
|       |       |       |       |       |       |       |       |        |       |       |       |       |        | DR3    | DR9  |      |      |      |      |     |       |      |      |  |
|       |       |       |       |       |       |       |       |        |       |       |       |       |        | DR4    | DR10 | DR12 | DR14 |      |      |     |       |      |      |  |
|       |       |       |       |       |       |       |       |        |       |       |       |       |        | DR5    | DR11 | DR13 | DR15 |      |      |     |       |      |      |  |

■ 8-bit register    ■ 32-bit register    ■ 64-bit register    ■ 80-bit register    ■ 128-bit register    ■ 256-bit register  
■ 16-bit register    ■ 32-bit register    ■ 64-bit register    ■ 128-bit register    ■ 512-bit register

# Differences between 32 bit and 64 bit

- 64-bit x86 has a number of differences in register usage, stack use, function calling conventions, etc.
- 64-bit CPUs are backwards compatible with 32-bit conventions and ISA
- We will learn 32-bit x86 first, and move on to 64-bit later

# x86 Assembly Syntaxes

- Although there is just one machine code format, there are THREE different assembly languages
  - Intel syntax, dominant for MS-DOS and Windows
    - Used with Visual Studio, and MASM
  - NASM syntax, a variation of Intel syntax
  - AT&T syntax, dominant for UNIX/Linux
    - Used with gcc and gas

# The Confusion Begins (*or continues*)

- There are differences between AT&T and Intel assembly:

|                     | AT&T                                                                                                                                                                                | Intel                                                                                                                                                                                                        |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Parameter order     | Source before the destination<br><code>eax ← 5 is mov \$5, %eax</code>                                                                                                              | Destination before source<br><code>eax ← 5 is mov eax, 5</code>                                                                                                                                              |
| Parameter Size      | Mnemonics are suffixed with a letter indicating the size of the operands (e.g., "q" for qword, "l" for long (dword), "w" for word, and "b" for byte)<br><code>addl \$4, %esp</code> | Derived from the name of the register that is used (e.g., <i>rax</i> , <i>eax</i> , <i>ax</i> , <i>a</i> imply <i>q,l,w,b</i> in that order)<br><code>add esp, 4</code>                                      |
| Immediate values    | Prefixed with a "\$", and registers must be prefixed with a "%"                                                                                                                     | The assembler automatically detects the type of symbols; i.e., if they are registers, constants or something else.                                                                                           |
| Effective addresses | General syntax <i>DISP(BASE,INDEX,SCALE)</i> Example:<br><code>movl arr1(%ebx,%ecx,4), %eax</code>                                                                                  | Use variables, and need to be in square brackets; additionally, size keywords like <i>byte</i> , <i>word</i> , or <i>dword</i> have to be used. Example:<br><code>mov eax, dword [ebx + ecx*4 + arr1]</code> |

# NASM Syntax

- First we will cover NASM syntax
  - NASM syntax is a variation of Intel syntax that is compatible with the Netwide Assembler (NASM)
- AT&T syntax is different than Intel, so we will discuss each style separately
  - To quote Reddit:  
*“AT&T syntax is an affront to humanity, and should be cast into the flames of Mount Doom.”*
- All the following slides cover NASM syntax unless otherwise noted

# Byte Addressable Memory

- Each byte has an address
- When pushing a longword (32 bits) onto the stack, you must subtract ***four*** from the stack pointer
  - It's up to the programmer to do this well
- Because of memory interleaving, memory accesses are faster when *properly aligned*
  - Words should have an address that is a multiple of 2, longwords should have multiples of 4, etc.

# What is a word?

- On a 32-bit x86 machine, the word size is 32 bits.
- This is, paradoxically, called a *longword*.
- The terms used to describe sizes in the x86 architecture are:
  - byte: 8 bits
  - word: 2 bytes (16 bits)
  - dword/long: 4 bytes (stands for "double word") (32 bits)
  - qword: 8 bytes (stands for "quad word") (64 bits)
- So why does word = 32 bits AND 16 bits?
- When x86 was first introduced 16 bit was the actual word size, nowadays to maintain compatibility with everything x86 related, word in the architecture is 16 bits, even though the actual word size may be 32 or 64 bits

# Linux vs Windows

- Programming assembly on Windows is vastly different than programming x86 assembly on Linux
  - Most tutorials and examples are written for Linux
- Linux uses system calls to do simple things like display to the monitor
- Windows requires using external functions to display to the monitor

# Hello World on Linux x86 32 bit

```
global _start ; global must be defined for the linker

section .text ; Defines the area for assembly instructions

_start:
 mov eax, 0x4 ; prepare the write syscall
 mov ebx, 0x1 ; use stdout
 mov ecx, message ; use the message for the write buffer
 mov edx, message_length ; supply the length
 int 0x80 ; invoke write syscall

 mov eax, 0x1 ; prepare the exit syscall
 mov ebx, 0x0 ; provide error code of 0x0 (no error)
 int 0x80 ; invoke exit syscall

section .data ; Define bytes (db) for "Hello World!(0xA)(0x0)" 0xA is the newline, 0x0 is the null terminator
message: db 'Hello World!', 0xA, 0x0 ; message_length is equal (equ) to length of message
message_length: equ $-message
```

# Hello World on Windows x86 32 bit

```
global _main ; global must be defined for the linker
extern _printf ; On Windows, we must use printf for simplicity

section .text ; Defines the area for assembly instructions

_main: ; main starts here
 push message ; Push message onto the stack (esp gets subtracted by 4 bytes (32 bits))
 call _printf ; Call the printf function
 add esp, 4 ; Pop message off the stack (esp gets added by 4 bytes (32 bits))
 ret ; Return

message: ; Define bytes (db) for "Hello World!(0xA)(0x0)" 0xA is the newline, 0x0 is the null terminator
 db 'Hello World!', 0xA, 0x0
```

# Assembling and Executing x86 on Windows

- Download NASM:
  - <https://www.nasm.us/>
  - NASM will convert your assembly code into an object file
- Download MingGW:
  - <https://sourceforge.net/projects/mingw/>
  - MingGW is a suite of tools, most importantly it contains gcc which is a set of more tools, notably, a compiler
  - gcc will compile our object file into an executable

# Assembling and Executing x86 on Windows (cont.)

- Using the command line:
  - Use NASM:
    - `nasm -felf (your-program-name).asm`
  - Use gcc:
    - `gcc -o helloWorld (your-program-name).o`
  - Run the program:
    - `helloWorld`

# Multiplication on Windows x86 32 bit

```
; Define main function (for linker)
global _main

; Use external functions for simplicity
extern _printf
extern _scanf
extern _exit

; Variables
section .data
 ; Create two empty locations in memory, each is 4 bytes for an integer
 int1: times 4 db 0
 int2: times 4 db 0
 ; These are strings, each ends with a null terminator (0) and some have newlines (10)
 ; Format_in uses the format for scanf, a very standard c function
 format_in: db '%d', 0
 ; Format_in uses the format for printf, a very standard c function
 format_out: db 'The result is: %d', 10, 0
 hello: db 'This program will multiply 2 numbers. Awesome, huh?', 10, 10, 0
 message: db 'Type a number: ', 0

; Code
section .text
_main:
 ; Push address of "hello" and print
 push hello
 call _printf
 ; The address of hello was 4 bytes, so we can now remove 4 bytes from the stack
 add esp, 4

 ; Print the first integer's message
 push message
 call _printf
 add esp, 4

 ; Read the first integer from the keyboard (arguments are reversed in asm)
 push int1
 push format_in
 call _scanf
 add esp, 8 ; We pushed 2 arguments so we must pop 8 bytes

 ; Print the second integer's message
 push message
 call _printf
 add esp, 4

 ; Read the second integer from the keyboard (arguments are reversed in asm)
 push int2
 push format_in
 call _scanf
 add esp, 8

 ; To perform multiplication we need to load both operands into eax and ebx
 mov eax, [int1]
 mov ebx, [int2]

 ; Mul multiplies eax with a specified register
 ; eax is always the first operand, ebx is the second operand
 ; The result may be 64-bit, in that case edx:eax are register pairs:
 ; edx contains the high 32 bits
 ; eax contains the low 32 bits
 mul ebx

 ; Print the result of the multiplication
 push eax
 push format_out
 call _printf
 add esp, 8

 ; Exit with code 0 (no error)
 push 0
 call _exit
 add esp, 4
ret
```

# Comparing Execution Times between Python, C, Assembly

Python

Programming: 56s

Running: 54.4s

C/C++

Programming: 1m19s

Running: 5.3s

x86 ASM

Programming: 2m20s

Running: 5.3s

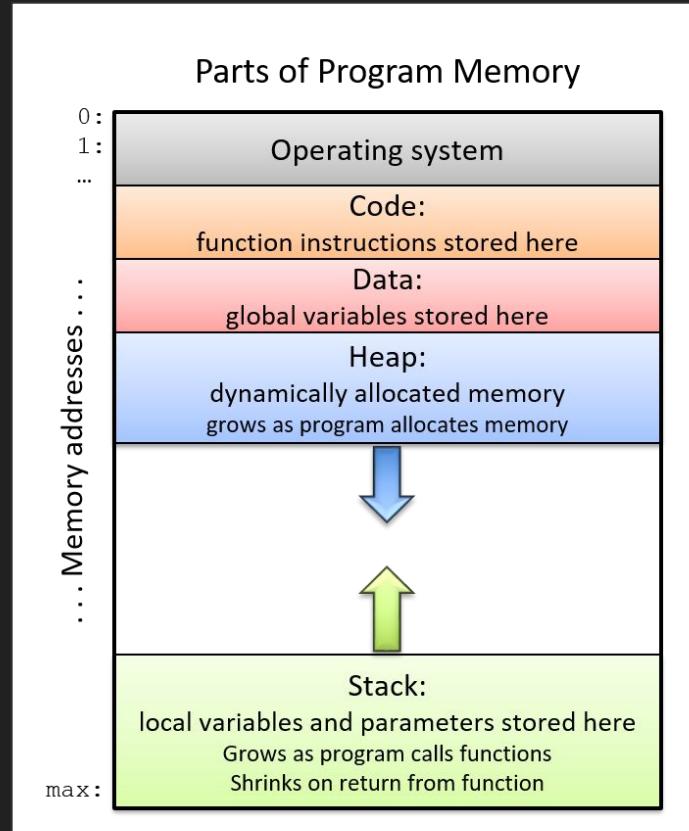
- <https://www.youtube.com/watch?v=3PclJKd1PKU>

# Core Components for x86 Assembly

- Heap
- Stack
- Registers
- Instructions

# Heap

- Used for dynamically allocating memory for your program
  - Malloc, calloc, global/static variables
- Dynamically allocated memory is required when your program does not know until runtime how much memory it needs
  - Example: you read a list of book titles from a text file
    - Your program should expect any reasonable number of book titles or even 0
    - You do not know the number of book titles you will read from the file
    - The memory must be dynamically allocated in order to be used in your program



# Registers

- General Purpose Registers (GPRs)
  - EAX, EBX, ECX, EDX, ESI, EDI
- Reserved:
  - EBP, ESP, EIP

# Stack

- In x86 we have push and pop instructions to simplify working with the stack
- Pushing stores data to the next available location on the stack while automatically decrementing the stack pointer
- Popping loads data from the top of the stack while automatically incrementing the stack pointer
- Just like LC3 each function has its own stack frame
  - EBP: frame pointer
  - ESP: stack pointer

# x86 Simplest Stack Example

```
#include <stdio.h>
#include <stdlib.h>

void func(int x)
{
 int a = 0;
 int b = x;
}

int main()
{
 func(10);
}
```

| Address    | Data | Note                  | Pointer |
|------------|------|-----------------------|---------|
| x5FFF FFF1 | x0A  | int b                 | <- ESP  |
| x5FFF FFF2 | x00  |                       |         |
| x5FFF FFF3 | x00  |                       |         |
| x5FFF FFF4 | x00  |                       |         |
| x5FFF FFF5 | x00  | int a                 |         |
| x5FFF FFF6 | x00  |                       |         |
| x5FFF FFF7 | x00  |                       |         |
| x5FFF FFF8 | x00  |                       |         |
| x5FFF FFF9 | x##  | func's return address | <- EBP  |
| x5FFF FFFA | x##  |                       |         |
| x5FFF FFFB | x##  |                       |         |
| x5FFF FFFC | x##  |                       |         |
| x5FFF FFFD | x0A  | int x                 |         |
| x5FFF FFFE | x00  |                       |         |
| x5FFF FFFF | x00  |                       |         |
| x6000 0000 | x00  |                       |         |

# Instructions

- In x86, instructions can only accept two arguments per operation
- Instructions are written in this format:
  - operation argument, argument
  - Typically, the first argument is the destination of the operation

# Mov Instruction

- The mov instruction takes two arguments:
  - mov dest, source
  - The source is moved into the destination
- If you want to copy a register into another register use:
  - mov eax, ebx
- If you want to load something into EAX using the frame pointer EBP you must use square brackets:
  - mov eax, [ebp - 0x8]
  - If using the previous example, this instruction loads the value of int b into of eax
  - The value inside of the square brackets must be an address, the square brackets mov the value stored at memory address ebp-0x8 into ebp

# Add/Sub Instruction

- If we only have two arguments how can we add 2 numbers and put it into a destination?
  - Hint: the first argument is also the destination
- To add two registers together, you can do:
  - add eax, ebx
  - The result of the addition is stored in eax
- To add a constant to a register, you can do:
  - add eax, 5
  - add eax, 0x5
- To add a register to a value on the stack, you can do:
  - add [ebp - 0x4], eax
  - The result of the addition is stored @ memory location ebp - 0x4
- Using the instruction sub we can subtract, using the same format:
  - sub eax, 5
  - eax = eax - 5
  - If eax is 18 (x12) before the subtraction, it will be 13 (xD) afterwards

# Push/Pop Instruction

- Push and pop decrement and increment the stack pointer automatically
- To push something onto the stack:
  - `push eax`
  - This pushes whatever value is inside of `eax` to the stack and also decrements the stack pointer by 4 (4 bytes required to store `eax`)
- To pop something from the stack:
  - `pop ebx`
  - This pops whatever value is on the top of the stack and puts the value into `ebx` and increments the stack by 4

# Lea Instruction

- To load the address of a memory location into a register the lea address can be used
- If you have a variable called var0 that stores a 32-bit integer, you can use:
  - `lea eax, var0`
  - Now, eax contains whatever the address is of var0

# Cmp Instruction

- To make a comparison between two values using subtraction, use `cmp`
- The `cmp` instruction will set the status flag bits inside of the EFLAGS register
  - `cmp eax, 3`
  - If `eax = 1`, the flag set will be less than 0 since  $1 - 3 = -2$
  - If `eax = 3`, the flag set will be 0 since  $3 - 3 = 0$
  - If `eax = 6`, the flag set will be greater than 0 since  $6 - 3 = 3$
- Later, we can use these flags to perform conditional jumps, just like the conditional branches in LC3

# Cmp Instruction

- To make a comparison between two values using subtraction, use `cmp`
- The `cmp` instruction will set the status flag bits inside of the EFLAGS register
  - `cmp eax, 3`
  - If `eax = 1`, the flag set will be less than 0 since  $1 - 3 = -2$
  - If `eax = 3`, the flag set will be 0 since  $3 - 3 = 0$
  - If `eax = 6`, the flag set will be greater than 0 since  $6 - 3 = 3$
- Later, we can use these flags to perform conditional jumps, just like the conditional branches in LC3

# Jmp Instruction

- After making a comparison, we can jump to an address based on that comparison
- There are many ways we can jump but here's a few:
  - jmp address (jump to address unconditionally)
  - jl address (jump to address if less than)
  - jz address (jump to address if zero)
  - jg address (jump to address if greater than)

# 16 - Final Exam Study Guide

CEG3310/5310 - Computer Organization  
Max Gilson

# Final Exam

- True/False
  - 10 questions
  - 0.35 points each
  - 3.5 points total (3.5% of final class grade)
- Multiple choice
  - 10 questions
  - 0.7 points each
  - 7 points total (7% of final class grade)
- Short answer
  - 5 questions
  - 2.1 points each
  - 10.5 points total (10.5% of final class grade)
- Coding
  - 2 questions
  - 7 points each
  - 14 points total (14% of final class grade)
  - Must be graded manually (0 points initially)

# Bits, Data Types and Operations (02)

- How is data represented?
  - Convert between binary/decimal/hex
  - Adding numbers in binary
- Two's Complement
- Floats

# LC3 ISA (03)

- Instructions
  - Operate instructions
  - Data movement instructions
  - Control instructions
- Opcodes
- Operands
- PCoffset
- Addressing

# LC3 Assembly (04)

- Pseudo-Ops
- Labels
- Opcodes
- C code to Executable

# Subroutines/Traps (09)

- TRAP calls
- TRAP vector table
- TRAP instructions / RET instruction
- Subroutines
- JSR / RET

# The C Programming Language (10)

- High level languages
- Compiling C Code
- Pointers
- Arrays
- Function calls
- Scope

# Runtime Stack (11)

- Stacks
- Stack Pointer
- Frame Pointer
- Global Variable Pointer
- Runtime Stack construction

# I/O (08)

- I/O
- Memory Mapped I/O
- Polling
- Interrupts

# Interrupts (14)

- Interrupts
- ISR
- RTI
- Priority

# Recursion (12)

- Recursion
- Base case
- General case
- Runtime stack in recursion

# Digital Logic (06)

- Logical operations
- Boolean Algebra
- Truth Tables
- Bit Vector Operations
- Transistors
- Gates and Devices
- Memory
- ALU

# Von Neumann Model (07)

- The Von Neumann Model
- Processing Unit
- Memory
- Control Unit
- Instruction Cycle

# LC3 Control Signals (07a)

- ALU
- Registers
- Other components
- Control signal paths
  - Operate
  - Data movement
  - Control
- Instruction cycle

# Memory and Cache (13)

- Caches
- L1, L2, L3
- Memory Hierarchy
- Fully Associative Cache
- Cache replacement algorithms
- Virtual memory