# Intro to Computers

**UT LAW 379M**

**Fall 2021**

Lecture Notes

# Three Major Computer Parts


Central Processing Unit (CPU)

Hard Drive

Random Access Memory (RAM)


# First, a Word About Data

- Computers store and process all data as ***BINARY NUMBERS***
- Everything is a binary number.  Games, programs, music, etc
- The binary numbers are ***interpreted*** as letters, music, etc

# Quick Binary Lesson

Base 10 Numbers

| $10^3 = 1000$ | $10^2 = 100$ | $10^1 = 10$ | $10^0 = 1$ |
|---|---|---|---|
| 3 | 0 | 9 | 5 |
| (3X1000) + | (0X100) + | (9X10) + | (5X1) |
| 3000 + | 0 + | 90 + | 5 |
| | | | 3095 |

Base 2 Numbers

| $2^3 = 8$ | $2^2 = 4$ | $2^1 = 2$ | $2^0 = 1$ |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| (1X8) + | (0X4) + | (1X2) + | (0X1) |
| 8 + | 0 + | 2 + | 0 |
| | | | 10 |

# What are These Binary Numbers?

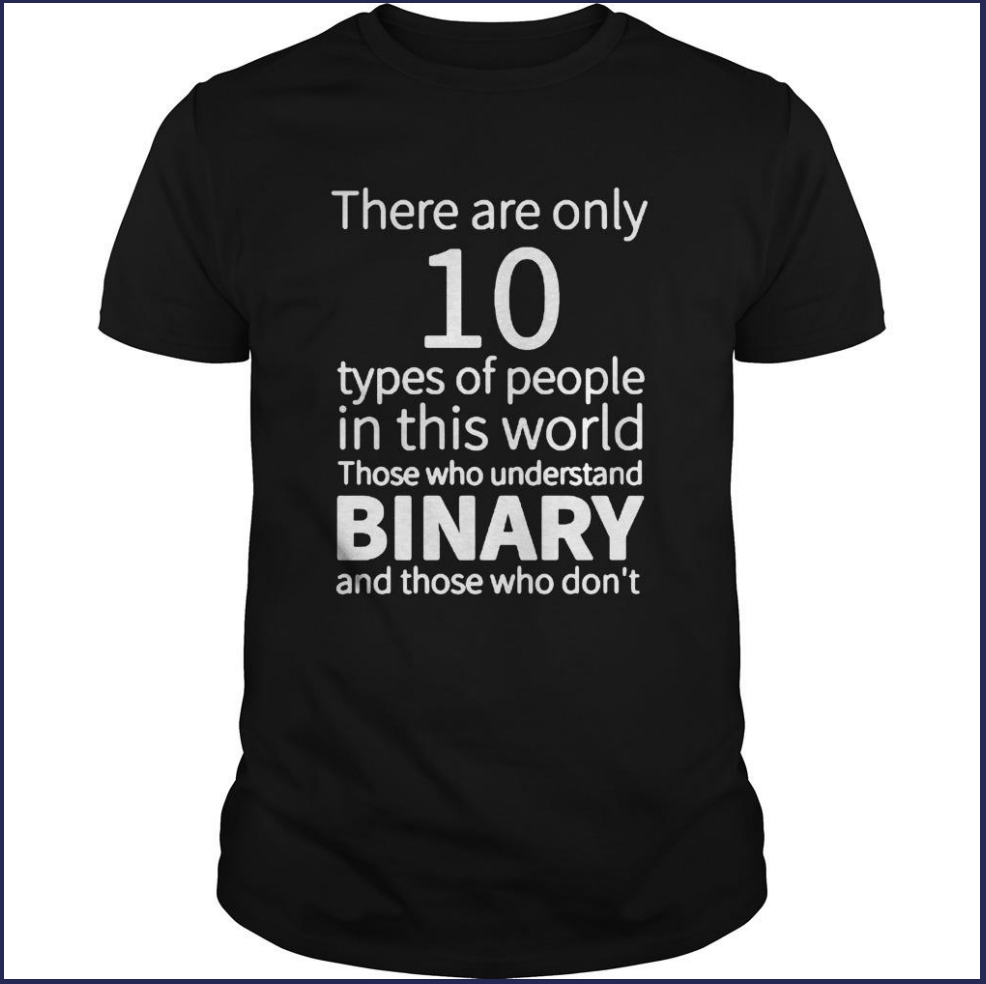
- Binary 1000?
- Binary 0100?
- Binary 0010?
- Binary 0001?
- Binary 0101?
- Binary 1010?
- Binary 1111?

Remember 8, 4, 2,1 columns.
If there's a one, add the column:

0110 = 4+2 = 6

1001 = 8+1 = 9

# Decode This Shirt



# Binary Sizes

```
09/09/2020  10:50 PM    <DIR>          Links
09/09/2020  10:50 PM    <DIR>          Music
10/04/2020  09:41 AM           397,307 npm-debug.log
09/11/2020  11:28 AM    <DIR>          OneDrive
06/29/2020  01:51 PM                26 osync.log
10/05/2020  11:51 AM    <DIR>          Pictures
08/26/2019  11:10 PM                13 query
09/09/2020  10:50 PM    <DIR>          Saved Games
09/09/2020  10:50 PM    <DIR>          Searches
03/11/2020  08:05 AM         3,298,911 snap_beans1.tgz
08/26/2019  11:10 PM                13 start
05/19/2019  09:57 PM                 0 Sti_Trace.log
08/26/2019  11:10 PM                13 stop
10/06/2020  08:53 AM    <DIR>          Videos
02/10/2020  07:48 PM    <DIR>          VirtualBox VMs
               8 File(s)     61,789,284 bytes
              22 Dir(s)  12,737,953,792 bytes free

C:\Users\seth_>
```

- A one or a zero is a “bit”
- 8 bits is a byte
- Let’s use our command line to see how big files are:
  - Open a terminal on Mac or the cmd shell on Windows
  - Type “ls –l” on Mac or “dir” on Windows

```
sethjn@Paladin:~$ ls -l
total 69160
-rwxrwxrwx 1 sethjn sethjn   737282 Jul 27 13:01 chaum_001.pdf
lrwxrwxrwx 1 sethjn sethjn       30 Aug 26  2019 dev -> /mnt/d/seth_/P
v/
-rw-rw-rw- 1 sethjn sethjn  2488927 Mar 30  2020 files.txt
-rwxrwxrwx 1 sethjn sethjn 32655988 Jul 27 09:58 iso11770_1.pdf
-rwxrwxrwx 1 sethjn sethjn   779655 Jul 27 09:59 iso11770_3.pdf
-rwxrwxrwx 1 sethjn sethjn   186903 Jul 27 22:20 places_caswell.pdf
```

# Binary is Often Represented as Hex

- Binary is Base 2
- For "short hand", binary is often written in Base 16
- This is because there is a 1:1 mapping between 4 bits and 1 Hex number:

| **0000 – 0** | **0100 – 4** | **1000 – 8** | **1100 – C (hex 12)** |
|---|---|---|---|
| 0001 – 1 | 0101 – 5 | 1001 – 9 | 1101 – D (hex 13) |
| 0010 – 2 | 0110 – 6 | 1010 – A (hex 10) | 1110 – E (hex 14) |
| 0011 – 3 | 0111 - 7 | 1011 – B (hex 11) | 1111 – F (hex 15) |

# Convert to Binary/Decimal

- 0xFF
- 0x10
- 0xA3
- 0x8C
- 0xF0
- 0x0F

Please use 8 bits for the binary. Leading 0's are fine

# Convert to Binary/Hex

- 250
- 100
- 128
- 129
- 64
- 32
- 96 (notice, 64+32)

Please use 8 bits for the binary. Use 2 digits for hex. Leading 0's are fine

# Interpretation - Letters

- ASCII – The original English language character mapping
- Each letter, symbol, and control character had a code
- ‘a’ – 97 (decimal), 0x61 (hex), 0110 0001 (bin)
- ‘A’ – 65 (decimal), 0x41 (hex), 0100 0001 (bin)
- Newline – 10 (decimal), 0x0a (hex), 0000 1010 (bin)

# ASCII Exercise

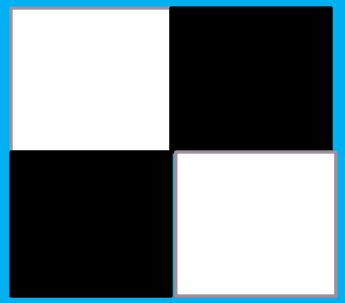
- ASCII table: https://www.ascii-code.com/
- Find the hex codes for “hello world!”
- Find the hext codes for your name
- Include spaces and punctuation!

# Interpretation - Images

- Uncompressed pictures can be 1 number per pixel
- Each number is an index into a table of colors
- 1 bit-per-pixel can do two colors (black and white)
- 8 bpp can do 256 colors
- 32 bpp can do 4,294,967,296 colors

# 2x2 BW Image Example

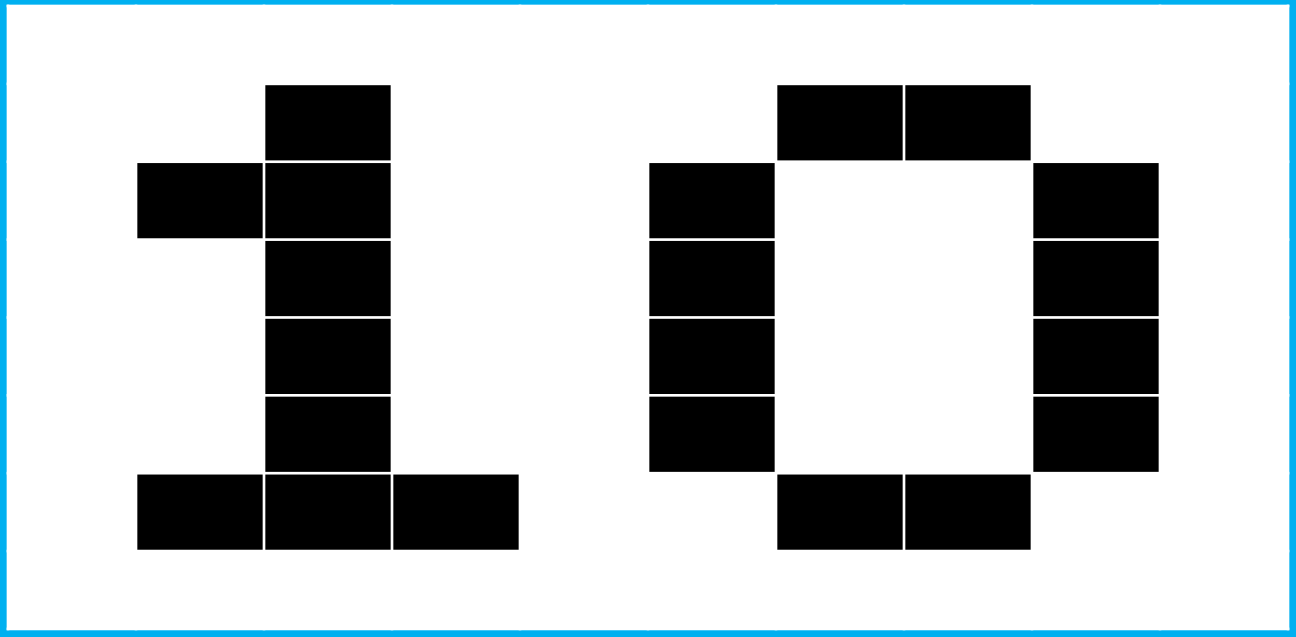
1001



1010

# larger BW Image Example

11111 11111
11011 10011
10011 01101
11011 01101
11011 01101
10001 10011
11111 11111

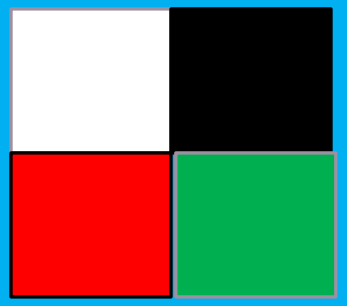


8x10

# 2x2 4 Color Image Example

00 – Black
01 – Red
10 – Green
11 – White

11000110



# Interpretation - Images

- Uncompressed pictures can be 1 number per pixel
- Each number is an index into a table of colors
- 1 bit-per-pixel can do two colors (black and white)
- 8 bpp can do 256 colors
- 32 bpp can do 4,294,967,296 colors

# (Side Note, Compression)

- Images can be compressed with special techniques
- Groups of same-color pixels stored together
- Videos go even further
  - I frame, full picture
  - B frame/p frame, changes from previous picture

# Interpretation - Documents

- Multi-level encoding
- Codes indicate whether text is regular, bold, etc
- All the data is ***rendered*** by the display program

# HTML Example

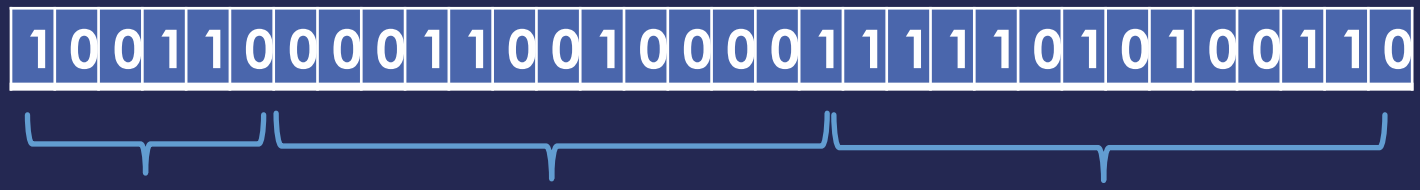
- The entire document, except for media, is text
  - So, there is “interpretation” from binary to text
  - And more “interpretation” from text to codes
- Example:
  - <B>This is bold</B>
  - The <B> says “all text until next </B>” is bold

# Processors and Binary

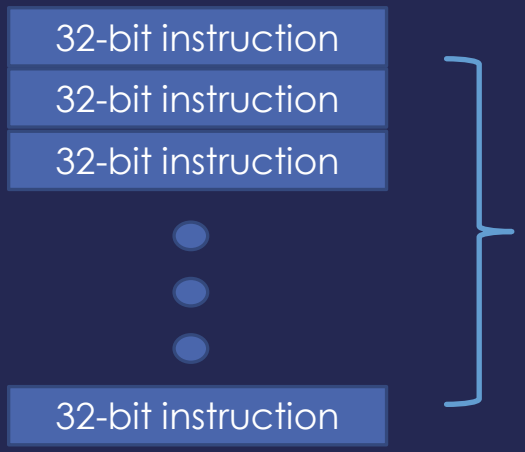

Computer "Instruction" is just a binary number. In older computers, like the 486 shown here, it was 32-bit. In more modern processors it is 64-bit.

# CPU Instructions


The instruction is broken down into pieces. One part is called the "op code" or operation code, it tells the computer what to do. The other parts are parameters. Each instruction "add", "subtract," etc. has its own op code.

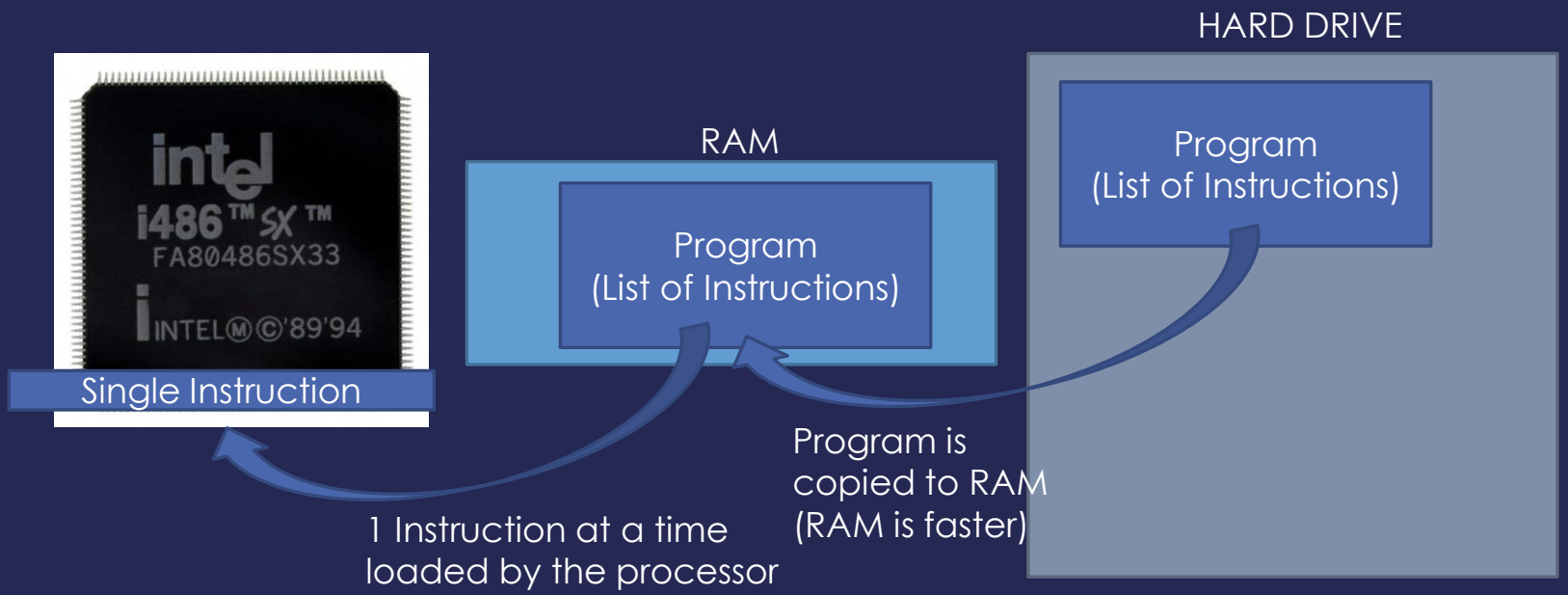


# Program Stored On Disk

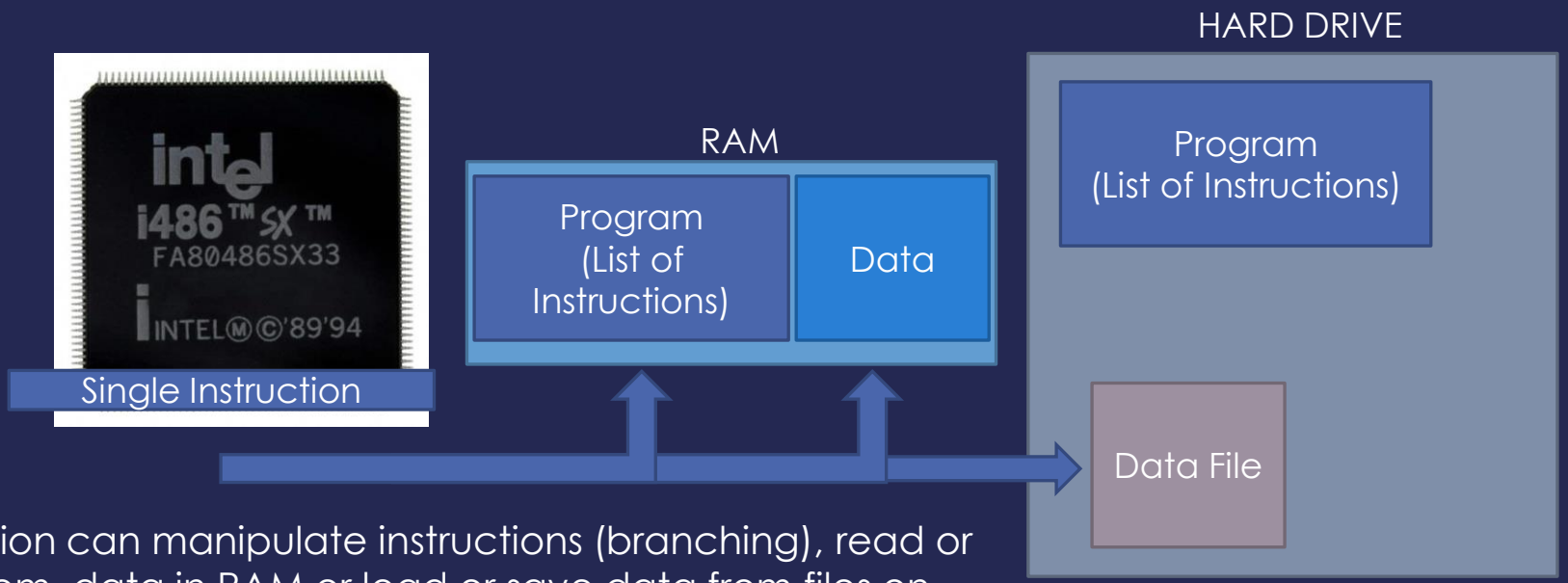



- Program is a list of instructions
- Processor executes one at a time.
- Some instructions are “branches”
- A branch jumps ahead or behind in the list
- Often “conditional” (e.g., if x > 0 then jump)

# Running a Program



# Data Too




Instruction can manipulate instructions (branching), read or write from  data in RAM or load or save data from files on disk. (All are just binary numbers)

# CPU Simulation Example

- YOU are going to be a computer processor
- You will have 8 bit instructions
- add, subtract, multiply, divide
- load, store, jump, jump if not zero

# 8-bit instructions

- First 3 bits are the operation:
  - 000 – load
  - 001 – store
  - 010 – add
  - 011 – subtract
  - 100 – multiply
  - 101 – divide
  - 110 – jump
  - 111 – jump if not zero
- 5 bits for the parameters

# Add/Subtract/Mult/Div

- 3 Parameters (5 bits)
- First, add a number (0) or add a register (1) (1 bit)
- Second, the destination register (2 bits)
- Third, a number or a register (2 bits)

# Registers

- CPU's use mini-storage places called registers
- A register usually just holds a single number
- For our pretend CPU, numbered 0-3 (r0, r1 …)
- 8-bit number storage in each register

# Example Add Number

- Instruction: 01000010
- First 3 bits: 010 – Add
- Next 1 bit: 0 – Add number
- Next 2 bits: 00 – register 0 (r0)
- Next 2 bits: 10 - 2
- “Add r0 and 2 together (store in r0)”

# 2 bits?!

- 2 Bits ranges between 0 and 3
- What if you want to add more than 3?!
- Call add multiple times, use multiply, etc
- Real computers aren't this limited
- But have similar problems

# Visualized


01000010

| R0 | 1000 0000 |
|---|---|
| R1 | |
| R2 | |
| R3 | |

| R0 | 1000 0010 |
|---|---|
| R1 | |
| R2 | |
| R3 | |

# Example Add Register

- Instruction: 01010001
- First 3 bits: 010 – Add
- Next 1 bit: 1 – Add register
- Next 2 bits: 00 – register 0 (r0)
- Next 2 bits: 01 – register 1 (r1)
- “Add r0 and r1 together (store in r0)”

# Visualized


01010001

| | |
|---|---|
| R0 | 1000 0000 |
| R1 | 0000 0010 |
| R2 | |
| R3 | |

| | |
|---|---|
| R0 | 1000 0010 |
| R1 | 0000 0010 |
| R2 | |
| R3 | |

# Store/Load Commands

- 2 Parameters (5 bits)
- Register to store from/load to(2 bits)
- Register with memory address (2 bits)
- Last bit ignored

# Memory Addresses

- Memory locations have addresses
- Each address is just a single number
- Biggest 8-bit number is 255
- With 8 bits, 256 memory addresses (0-255)
- (NOTE: This is RAM, not hard drive)

# Example Load

- Instruction:  00000010
- First 3 bits: 000 (load)
- Next 2 bits: 00 – Load to Register 0 (r0)
- Next 2 bits: 01 – use Register 1 (r1) for memory address
- Last bit is ignored

# Visualized


| | |
|---|---|
| R0 | |
| R1 | 1000 0000 |
| R2 | |
| R3 | |

| | |
|---|---|
| 126 | 1111 1111 |
| 127 | 0000 0000 |
| 128 | 1010 1010 |
| 129 | 0101 0101 |

00000010

| | |
|---|---|
| R0 | 1010 1010 |
| R1 | 1000 0000 |
| R2 | |
| R3 | |

# Where are these instructions?

- Instructions ARE IN MEMORY TOO!!!!
- Jump and Jump if not Zero are for jumping to instructions!

# Jump

- 1 Parameters (5 bits)
- Register with address to jump to
- Last 3 bits ignored

# Jump if not Zero

- 2 Parameter (5 bits)
- First, jump forward or backward (1 bit)
- Second, offset (4 bits)

# Wait! Something's Missing!

- Our Jump If Not Zero didn't specify a register!
- What are we comparing to zero?
- For our system, designate R3 as the check register
- Real systems aren't this limited
- But they use similar tricks

# Jump farther?!

- We only have four bits for the jump
- We can jump forward or back 16 addresses
- What if we need to go farther?
- We can put a jump instruction where we land

# Let's walk through a program!

- Assume that all registers start with 0 in them
- Assume that our first instruction is at address 0
- Assume that 5 numbers were prestored:
  - 250 – 1 (address 250, decimal value 1)
  - 251 – 2
  - 252 – 4
  - 253 – 8
  - 254 - 16

# Instructions I

| MEMORY ADDRESS | INSTRUCTION |
|---|---|
| 0000 0000 | 0100 1111 |
| 0000 0001 | 0100 1110 |
| 0000 0010 | 0101 0011 |
| 0000 0011 | 1001 0011 |
| 0000 0100 | 1001 0011 |
| 0000 0101 | 1000 0010 |
| 0000 0111 | 0110 1101 |

# CHECKPOINT!

- What is the value of reach register by this point?

# Instructions More Readable

| MEMORY ADDRESS | INSTRUCTION |
|---|---|
| 0000 0000 | Add num 3 to r3 |
| 0000 0001 | Add num 2 to r3 |
| 0000 0010 | Add r3 to r0 |
| 0000 0011 | Mul r3 to r0 |
| 0000 0100 | Mul r3 to r0 |
| 0000 0101 | Mul num 2 to r0 |
| 0000 0111 | Sub num 1 from r3 |

# Instructions II

| MEMORY ADDRESS | INSTRUCTION |
|---|---|
| 0000 1000 | 0001 0000 |
| 0000 1001 | 0101 0110 |
| 0000 1010 | 0100 0001 |
| 0000 1011 | 0110 1101 |
| 0000 1100 | 1111 0100 |
| 0000 1101 | 0010 1000 |
| 0000 1111 | 0100 0000 |

# Final Answer?

- What is the value of each register?
- Which value in memory changed?
- What is the purpose of this program?
- What was the purpose of the last instruction?

# Instructions More Readable

| MEMORY ADDRESS | INSTRUCTION |
|---|---|
| 0000 1000 | Load from mem at r0 into r2 |
| 0000 1001 | Add r2 to r1 |
| 0000 1010 | Add num 1 to r0 |
| 0000 1011 | Sub 1 to r3 |
| 0000 1100 | Jump -4 if r3 is not 0 |
| 0000 1101 | Store to mem at r0 from r1 |
| 0000 1111 | Add num 0 to r0 |

# Nop (No Operation)

- The last instruction adds zero to a register
- This doesn't change anything.
- A "nothing" instruction (filler)

# Repurposing instructions

- We assumed memory registers initialized to zero
- In real systems, can't assume this
- Most systems have an op for setting to zero
- We can set to zero using "multiply num 0 to register rx"

# Summary

- This is an OVERLY simplistic example
- Real instructions are larger and more complicated
- But the basic ideas and concepts are the same
- All of our marvelous modern world runs on small instructions

# Creating a Program

- NOBODY WANTS TO WRITE IN PROCESSOR CODES!!!!
- Programs written in Programming Language
- Programming Language is human readable
- However, programming language ***NOT*** machine readable
- Remember, processor ***ONLY*** speaks in basic instructions

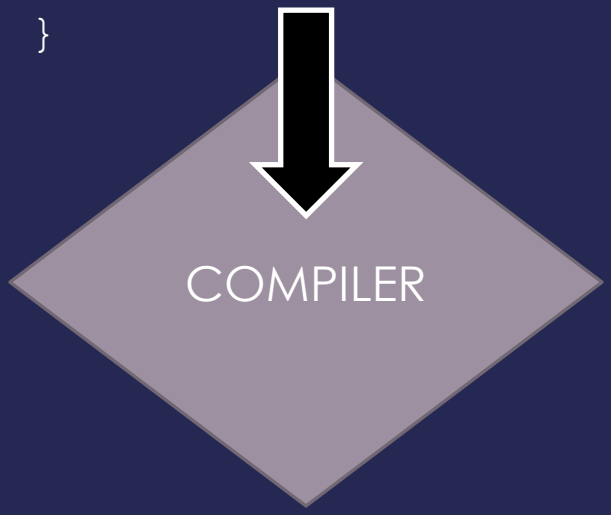
# Compiling

- Some languages can be “complied”
- This means the program is converted to machine readable
- Compiler knows how to translate and organize
- Quite complicated and still subject of research
- Example: C/C++

# C Example

```
#include <stdio>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```





1100101001111010101
1010010101001110101
1010101111101110000
1010101010001111000

# Interpreting

- Some languages can be “interpreted”
- This means the program is “executed” by interpreter
- Usually one instruction at a time
- Interpreter has to be a fully compiled program
- Interpretation is considerably slower
- Example: Python

# Python Example

```
if __name__ == "__main__":
    print("Hello World")
```


print("Hello World")

PYTHON INTERPRETER
(Already Binary Program)

1100101001111010101

# Virtual Machine

- An extreme form of “interpreting”
- Virtual processor with virtual instructions
- Programs are compiled to the virtual instructions
- Virtual instructions executed on the virtual processor
- Virtual processor translates instructions to host machine code
- Example: Java

# Java Example


```
import System;

class HelloWorld {
  static int main(void) {
    System.Out.println("Hello World!");
    return 0;
  }
}
```

Java Virtual Machine

1100101001111010101

COMPILER

1100101001111010101
1010010101001110101
1010101111101110000
1010101010001111000

# Why This is Useful

- Everything in a computer is a number. Programs, Data, etc
- This is great because programs and data can be stored the same way
- Everything can be copied over a network the same way

# Basic Execution