Notes

Thursday January 18th

Overview of the topics

Embedded Systems

- Computers are desigend for a specific purpose
- *Modern embedded systems contain a large number chips or integrated circuits (ICs)*

Types Digital, Analog, and Mixed Signals

- **Digital IC**
  - Microprocessor
- **Analog Ic**
  - Sensors
- **Analog-mixed -signal**
- Contains both digital and analog in a single chip/IC

Embedded vs. general computing systems: Number of Applications

- Embedded:
  - Not end-user programmable, On-time
  - Power, memory & compute limited: i.e. smartwathch, implantable device
- General computing:
  - End-user programmable, Faster is awlays better!
  - Greater resouce! i.e. servers

Internet of Things (IoT)

- **Internet** connected embedded systems
- Interconnection to the internet of devices is important

Cyber-Physical Systes (CPS)

- Cyber system (computer) + physical system (Plant)
- Embedded systems but integration emphasied

Real-Time Systems

- The correctness not only logical and computation but produced at the correct time
- A CORRECT VALUE AT THE WRONG TIME IS A FAULT

Trends in modern embeded systems

- Trending towards: cheaper, powerful, and more connected
- The ECU computing capactiy in BMW i8 is greater than the i3 (developed in 2014 or earlier)
- Rasberry pi 2 had even greater speeds
  - @ 4,744 MIPS (Millions of instructions per second) 1.0 GHz
- Human like intelligence in modern vehicles
- **Super computer introduced on a car in tesla 2019**
  - Full Self Driving Chip
  - Whats inside
    - CPU
    - GPU (1Ghz, 600 GLOPS) (GIGA FLOPS) GIGA FLOPS that is some cool shit man!!!
    - Neural processing units to handle floating points

Performance

- Processing real time from sensors in cars has increased a lot
- Autombile changes described in three step process]
  - **Inform**
    - Drivers wanted a way to meld lifestyle and car
  - **Assist**
    - Assisting drivers
  - **Assume**
    - communicate collaborate and fufill all functions

Efficiency

- Size weight, power, and cost constraints are all decreasing as time progresses

Safety

```
Examples below are examples of things that have outside influences and can have
failure
```

- *Examples
  - Therac 25
    - Computer controlled radiation therapy
  - Arian 5
    - Rocket destroyed in 40 seconds

Security and Privacy

```
Confidentiality issues: Infromation could be stolen
```

- Attack can be done in many ways
  - **Side channel leakage** : Observing the power consumption of the chip to guess the passwords
  - **Hardware or software trojan** : hidden functionality in software or hardware that leaks information to outsider
  - **Physical Attack** : Some dude touches your stuff and does some nasty stuff to it

.

```
Integrity issues: An attacmed can start a failure inside a system
```

- Attack can be done in many ways
  - **Fault Inejection** : Change voltage to corrupt output
  - **Hardware or software trojan** : hidden functionality in software or hardware that leaks information to outsider
  - **Physical Attack** : Tampering mode or chip on your own hardware (i.e. your xbox series x version x installing xtynine)

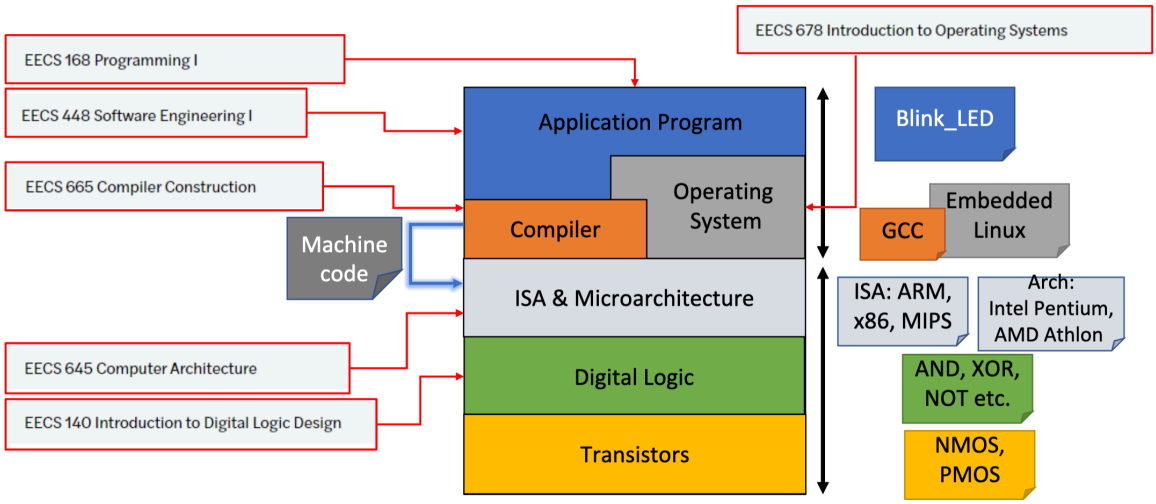
My Garbage Summary

- Limited Function computers in real world
- Requirements: perfromance, efficiency, safety, secrutiy, and privacy

TUESDAY JANUARY 23RD 2023

EMBEDDED SOFTWARE DEVELOPMENT

- **Software Abstraction**
  - Application programs, os, compiler
- **Hardware**
  - Primary Component: microprocessor or microcontroller
  - Abstractions: microacrchitecture and instruction set architecture



  - ISA: Instruction Set Architecture 3

- **Example Abstraction**
  - *Application Software*
    - Word processor, internet, browser, games
  - *System software*
    - *OS* : manages resources to run applications
    - *Compiler* : translates programs to machine readable binary
  - *Hardware*
    - Processor, memory, IO

Instructure Set Architecture

- (ISA)
- Acts as interface between hardware and software
- abstraction of hardware that can be controlled by assemby
- considred a manual for assembly
- specifis;
    - memory org
    - register set
    - instruction set (multiply, add, etc.)
- ARM, x86, MIPS, SPARC, and PowerPC
- *Analogy*
    - ISA of a car describes what the driver needs to do to get the carry out the drivers wishes

Microarchitecture

- Design describes interconnections of microarch elelments
- Implementation of the ISA
- x86-64 : AMD, Intel

Embed Sys Dev Platform

- exe runs on devices with the same arch

Compiler tool chain

- .c/.h -> .i -> .s -> .o -> .exe
- prprocessor :
    - compiler : gcc
        - assembler : as
            - linker : ld, turns machine code into executable code

Cross Compilation

- compile on one system run on another

- hex holds 4x binary

Problems

- Building can be complex
- building manually takes time and effot
- *SO WE USE MAKEFILESSS*

Makefiles

- Make is a tool which controls the generation of executables and other non-source files of a program
- make file automates commands

Thursday January 25th

Programming Languages

- **State if the following statements are true/false:**

1. Instruction Set Architecture (ISA) includes information about available instructions and registers in the hardware
2. The same ISA can be implemented with two different microarchitectures
3. The preprocessing step of the compiler tool-chain generates an assembly program from C/C++

- **Question:**
  - Which one(s) of the following is a cyber-physical system
    I. Robotic arm
    II. Smart watch
    III. GPS
    IV. Autonomous vehicle

3

Answers

1. True

2. True
3. True

- Cyber-physical examples: (i. robotic arm) and (iv. Autonomous vehicle)

High Level

- Feasbility of learning, porting across OS

Low Level

- Lower code overhead for execution
- Bitwise operatoin
- memory management
  - Pointers, dynamic memory allocation
- I/O Operation

Bits vs bites

- b7b6b5b4b43b2b1b0
- b7 Most significant bit (msb)
- b0 Least significant bit (LSB)
- **MSB LSB**
  - 1000 > than 0001

Declaring variables

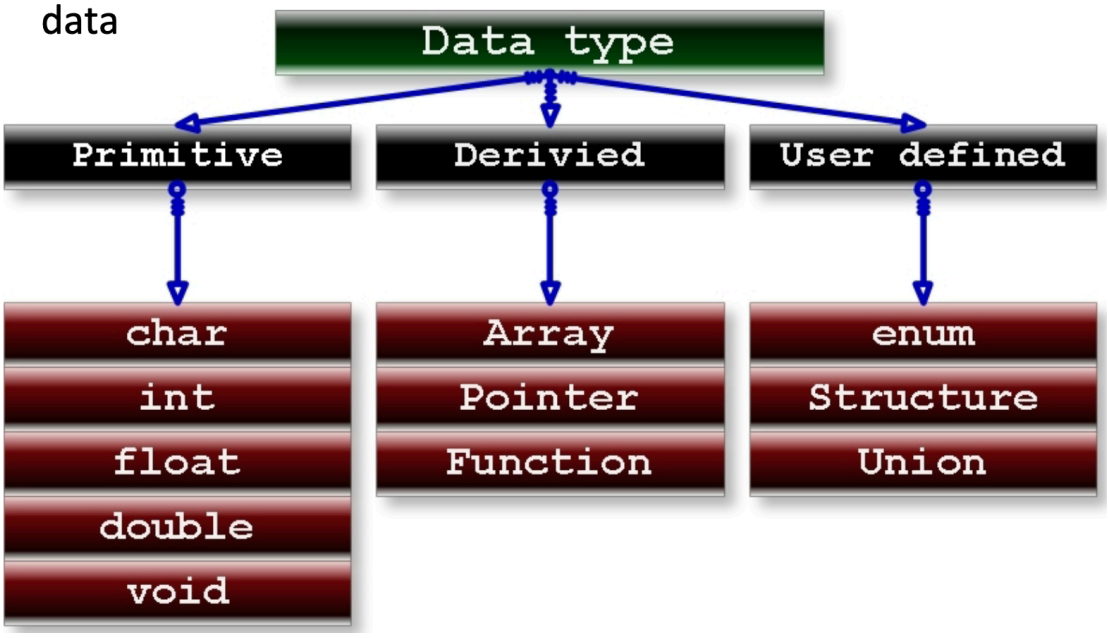
- must have data type and possible declaration of variable
- python is dynamically types while thigns like c/c++ are statically typed

Benefits of static vs dynamically typed

- **Dynamically Typed**
  - flexible and faster to produce
- **Statically Typed**
  - *efficient machine code generation*
  - *low-level control*

- *generates optimized machine code*

data



Differnet variables in computer memory

- Each address location typically hold 8-bit (i.e., 1-byte) of data.
- A 4-byte int value occupies 4 memory locations. A 32-bit system typically uses 32-bit addresses.

Type Modifiers

<type modifier (s)> <data-type> <var name> = < value>;

Increase the size of data types or change their properties

- Short
- Long
- Unsigned
- Signed

Data type	Storage	Range
char	1 Byte	[-128,+127]
unsigned char	1 Byte	[0, 255]
short int	2 Byte	[-32768,+32767]
unsigned short int	2 Byte	[0, 65535]
int	2 or 4 Byte	$[-2^{15}, 2^{15}-1]$ or $[-2^{31}, 2^{31}-1]$
long int	4 or 8 Byte	$[-2^{31}, 2^{31}-1]$ or $[-2^{63}, 2^{63}-1]$
long long int	8 Bytes	$[-2^{63}, 2^{63}-1]$

Example: unsigned long int var = 200;

13

Type Qualifier

- **const** : variable can't be changed
- **volatile** : tells the compiler that the value of the variable may change at any time–without any action being taken by the nearby code (could change by the hardware instead)

Number Systems

- Modern number system : **POSITIONAL SYSTEM**
- hex base 16

Positional System	Base	Allowed Digits
Binary	Base 2	0,1
Octal	Base 8	0,1,2,3,4,5,6,7
Decimal	Base 10	0,1,2,3,4,5,6,7,8,9
Hex	Base ?	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F (A=10, …, F=15)

- How to convert those pesky numbers in case you ever *"find yourself debugging your assembly programs" - Dr. Hoque*

- Binary (base 2)
  - Symbols: 0,1
  - E.g., 1011_2 = **0b1011** = $1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$
  - 0b indicates binary

$$Digit \times Base^{Index}$$

- Hexadecimal (base 16)
  - Symbols: 0,1,…,9,A,B,…,F
  - E.g., 123_{16} = **0x123** = $1x16^2 + 2x16^1 + 3x16^0$
  - 0x indicates hex

Representing Signed numbers

- sign magnitude method
  - reserve a bit to represnet the sign (i.e 001 = +1 and 101 = -1)
- twos compliment
  - invert digits and add one
  - eliminate negative zero and make arithmetic in hardware easier
  - 0b means binary and 0x means hex

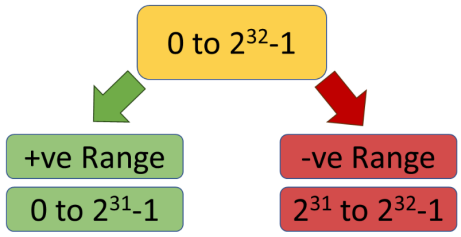
Tuesday January 30th

Computer Language

Review from last lecture: Two's complement

- Solution for representing signed numbers:
  - Using **"two's complement"** representation
- Eliminates negative zero and makes arithmetic in hardware easier
- Total range split into two segments
  - +ve Range: 0 to 2^{31}-1
  - -ve Range: 2^{31} to 2^{32}-1

Data type	Storage	Range
short int	2 Byte	[-32768,+32767]
unsigned short int	2 Byte	[0, 65535]
int	2 or* 4 Byte	[-2^{15}, 2^{15}-1] or [-2^{31}, 2^{31}-1]
long int	4 or* 8 Byte	[-2^{31}, 2^{31}-1] or [-2^{63}, 2^{63}-1]
long long int	8 Bytes	[-2^{63}, 2^{63}-1]



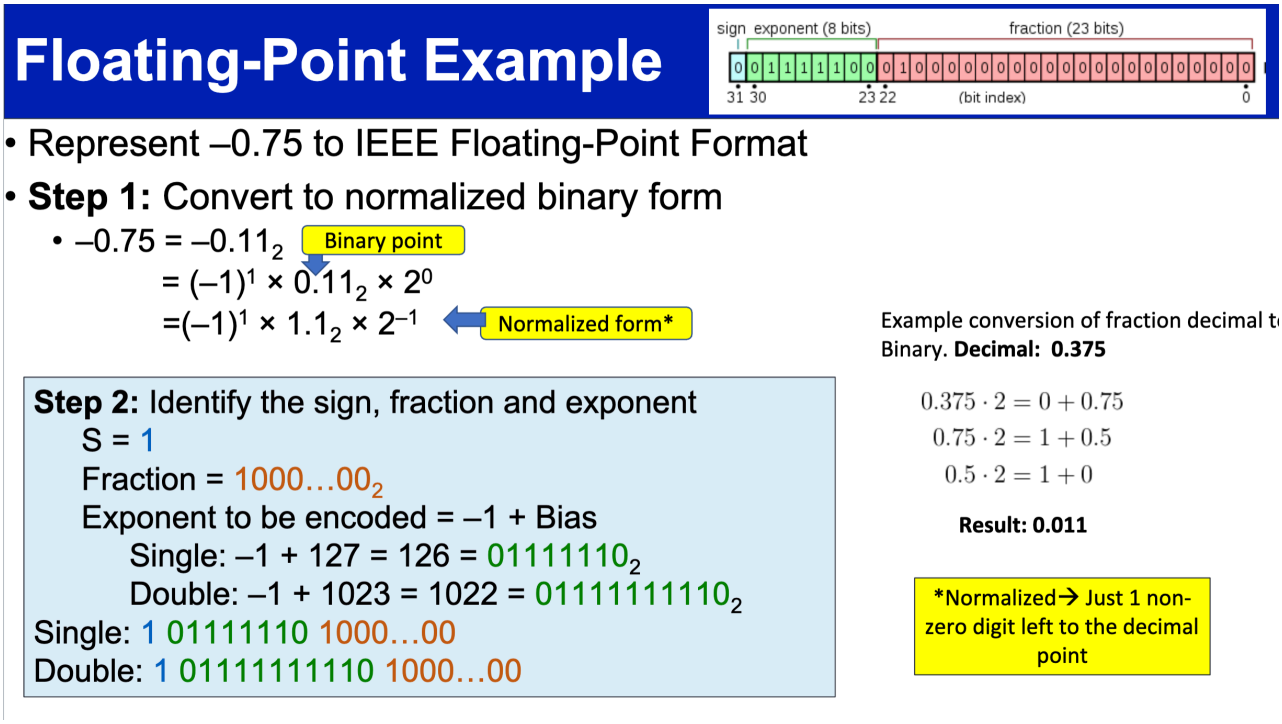
**compiler and processor dependent*

2

Fractional Numbers: Float and Double

- **Float** (*IEEE 754 single precision floating point numbers*)
  - 1-bit sign, 8-bits exponent, 23-bits fraction
  - 6 significant decimal digits of precision
- **Double**
  - 1-bit sign, 11-bits exponent, 52-bits fraction

- 15-17 significant decimal digits of precision



The Language of the Computer

The Software stack

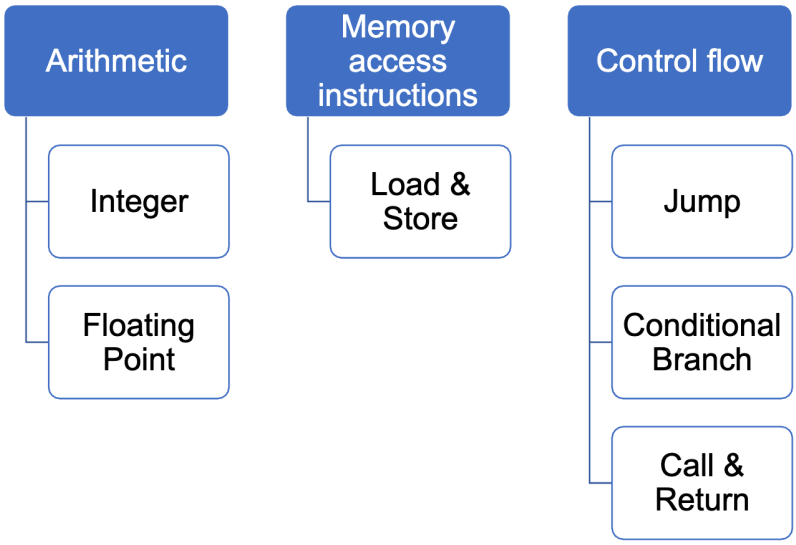
- Application Software
  - word processro, internet browser
- System software
  - OS, compilterç
- Hardware
  - Processor, memory, IO

The Software stack

- MIPS (Microprocessor without Interlocked Pipelined Stages

- widely used by the embedded market (hardware dudes)

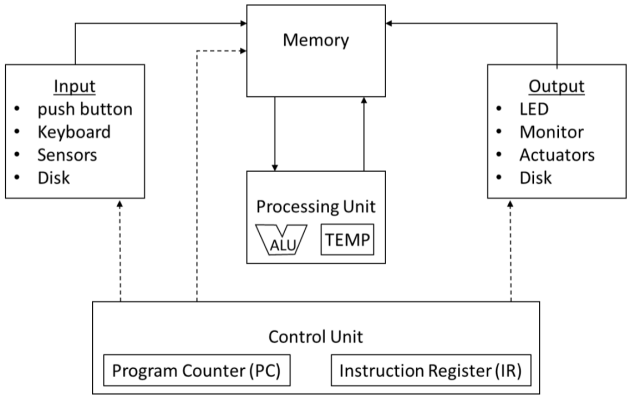
Classes of Instructions in MIPS

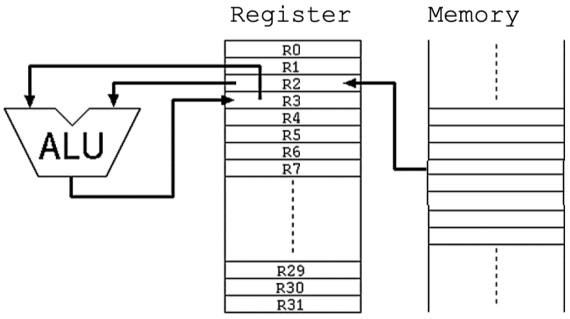


Opcode nad Operand

- **Opcode** : operation that is executed by the CPU (ex: add, sub)
- **Operand** : data or memory location used to execute that operation

Simplified Model of Computer





The Von Neumann computer model

16

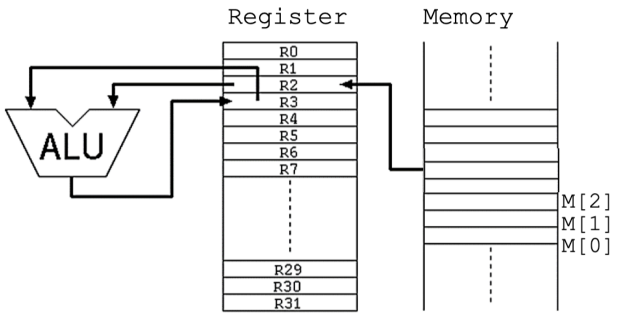
- Register is the TEMP

Registers

- **Registers**
- limited numver of memory location connected to ALU
- MIPS32 (MIPS) has 32 registers (there is also MIPS64)
  - Each of the 32 registers hold 32 bits
  - **C Code**
    - A = B + C;
    - A = B - C;
  - **MIPS Code**
    - add $r1, $r2, $r3
    - sub $r1, $r2, $r3
- different register values hold different usages

Memory

- Your program can have large number of variable and complex data types
  - Ex: Arrays with 100 elements
- 32 or 64 registers are not sufficient
- We use large memory to store this data



- But we cannot perform arithmetic operations directly from memory
  - Use load (lw) to bring data to register from memory

C code:
```
A = B + C;
A = B - C;
```

MIPS code:
```
add M[2], M[1], M[0]
sub M[2], M[1], M[0]
```

C code:
```
A = B + C;
```

MIPS code:
```
lw  $t0, M[1]       //B is in M[1]
lw  $t1, M[0]       //C is in M[0]
add $t1, $t1, $t0   // $t1=A
```

20

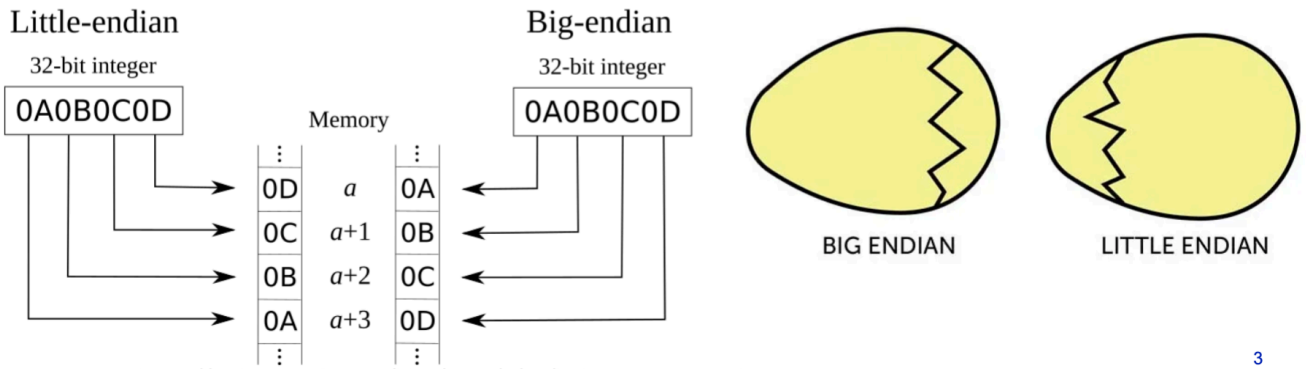
- C code: g= h + A[7]; //first element is A[0]
- MIPS code:
  - lw $t0, 32($s1) //32 because 8 elements
  - add $t0, $s2, $t0
    - 32 because you have to move 32 because each address has 4 bytes
    - this is saying that you are loading the value of the eight element of a to register t0
    - you are adding the two elements to t0
  - sw $t0, 48($s1)
    - This part of code sends/stores the value at A[12]

Thursday Feburary 1st

Language of the Computer

How bytes of a word are organized in memory?

- **Big Endian** : most significant bit is stored at the lowest memory address. Ex: PowerPC
- **Little Endian** : Just reverse of big Endian Ex: Intel , x86 processors, ARM



- Image source: https://agilescientific.com/blog/2017/3/31/little-endian-is-legal

3

- **Practice**

Example:

- C code: `A[1]= y + A[3];`
- Register $s1 has the base address of integer array A. Integer variable y is in $s2.
- Write the MIPS code

Register Data	
Base add of A[] →8000	s1
Value of variable y	s2
	t1
	t2

MIPS code:

```
lw   $t0, 12($s1)    // bring A[3] to reg
add $t0, $s2, $t0 //add A[3] with y
sw   $t0, 4($s1)    //send result to A[1]
```

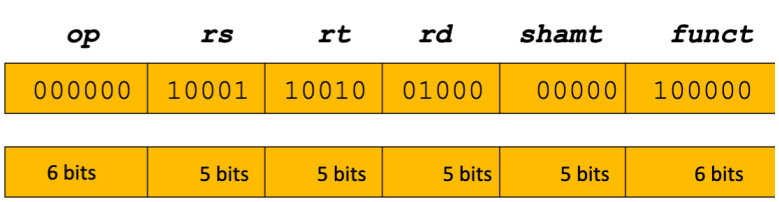
Memory Data	Address
A[0]	8000
	8001
	8002
	8003
A[1]	8004
	8005
	8006
	8007
A[2]	8008
	8009
	8010
	8011
A[3]	8012
	8013
	8014
	8015
	8016

Assembly to binary translation

- Basic Instruction formats
  - R (r type) -> opcode rs rt rd shamt funct
  - I (i type) -> opcode rs rt immediate
  - J (j type) -> opcode address

Machine Language: R-type instruction

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - Registers have numbers: $t0=9, $s1=17, $s2=18



Fields	Meaning
opcode	Partially specifies the instruction. Equal to 0 for all R-format instructions.
$rs	Specify register containing first operand.
$rt	Specify register containing second operand.
$rd	Specify register which will receive the result of computation.
shamt	Amount of shift. Set to 0 in all non-shift instructions.
funct	Combined with opcode exactly specifies the instruction.

6

So far: Instruction formats and encoding

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Data from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	`add $s1,$s2,$s3`
sub	R	0	18	19	17	0	34	`sub $s1,$s2,$s3`
lw	I	35	18	17	100			`lw $s1,100($s2)`
sw	I	43	18	17	100			`sw $s1,100($s2)`
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

11

- MIPS Instruction: addi $s3, $s3, 22 -> (adds 22 to s3)

Logical Operations

- Useful to operate on fields of bits within a word or on individual bits.
  - Set, clear, toggle bits
  - Faster multiplication and division
  - Data encryption, encoding etc.

Logical operations	C operators	MIPS
Shift left	<<	sll
Shift right	>>	srl
Bit-by-bit AND	&	and, andi
Bit-by-bit OR	|	or, ori
Bit-by-bit NOT	~	nor

and	and	$s1,$s2,$s3	$s1 = $s2 & $s3
or	or	$s1,$s2,$s3	$s1 = $s2 | $s3
nor	nor	$s1,$s2,$s3	$s1 = ~ ($s2 | $s3)
and immediate	andi	$s1,$s2,100	$s1 = $s2 & 100
or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100
shift left logical	sll	$s1,$s2,10	$s1 = $s2 << 10
shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10

- **Bitwise AND/OR**
  - and $t0,$t1,$t2 #reg t0 = reg t1 & reg t2
  - 1000 and 0111 = 0000
  - 1 and 0 = 0 or 0 and 0 = 0
  - 1 and 1 = 1
- **Bitwise Shfits**
  - sll = left shift (<<)
  - srl = right shift (>>)
  - sll $t2,$s0,4 -> reg t2 = reg s0 shifted 4 bits
branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L
branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L

    jumping to L

Multiplication is with a right shift

this only works with unsigned numbers because the sign will be lost

left shift

- Table below shows **left shift** for multiplication:
  - Applies for **unsigned number**

	Syntax	Binary (8 bit)	Decimal	Hint
	x=7	0000 0111	7	
Example 1	y=x<<1	0000 1110	14	7*2 or 7*2^1
Example 2	z=y<<3	0111 0000	112	14*8 or 7*2^3
Example 3	i=z<<2	1100 0000	192	Bit is lost

- Ex 1: left shifting by 1 bit is equivalent to multiplying by 2
- Ex 2: left shifting by **n** bits is equivalent to multiplying by **2^n**
- Ex 3: If a 1 passes the MSB due to shifting, the bit is lost and the above rule does not work

17

Division is a right shift

```
this only works with unsigned numbers again
```

right shift

- The result of a Right Shift operation is a **division** by 2^n , where n is the number of shifted bit positions.
  - Applies **for unsigned number**
  - **Ignores the remainder**

	Syntax	Binary (4 bit)	Decimal	Hint
	x=7	0111	7	
Example 1	y=x>>1	0011	3	7/2=3.5
Example 2	z=x>>2	0001	1	7/4=1.75
Example 3	i=x>>4	0000	0	7/16=0.45

- Ex 1: right shifting by 1 bit is equivalent to division by 2
- Ex 2: Shifting by n bits is equivalent to division by 2^n

18

Unconditional jump

- To implement the both if and else, we need an unconditional jump

```
j Exit    # go to Exit
```

- Example:

```
if (i==j)          bne $s3, $s4, Else
    h=g+h;         add $s0, $s1, $s2
else               j Exit
    f=g-h;         Else:sub $s0, $s1, $s2
                   Exit:
```

Loops

Loops

- while loop in C:

```
while (save[i]==k)
      i+=1;
```

- Assume: i and k are present in $s3 and $s5, base address of the array is in $s6.
- **Step 1:** Check if save[i]==k.

```
Loop: sll $t1,$s3,2 # reg $t1=4*i
      add  $t1,$t1,$s6    # $t1= address of save[i]
      lw   $t0,0($t1)# reg t0=save[i]
      bne  $t0, $s5, Exit # go to Exit if save[i]!=k
      addi $s3,$s3,1 # i=i+1
      j    Loop      # go to Loop
Exit:
```

Here's a summary of what each instruction does:

1. **`sll $t1, $s3, 2`**: Left shifts the value in register **`$s3`** by 2 bits, effectively multiplying it by 4, and stores the result in **`$t1`**.
2. **`add $t1, $t1, $s6`**: Adds the value in register **`$s6`** to the previously calculated value in **`$t1`**, resulting in the address of **`save[i]`**, and stores the result back in **`$t1`**.
3. **`lw $t0, 0($t1)`**: Loads a 32-bit word from the memory address stored in **`$t1`** (address of **`save[i]`**) into register **`$t0`**.
4. **`bne $t0, $s5, Exit`**: Branches to the **`Exit`** label if the values in registers **`$t0`** and **`$s5`** are not equal, essentially checking if **`save[i]`** is not equal to **`k`**.
5. **`addi $s3, $s3, 1`**: Adds an immediate value of 1 to the value in register **`$s3`**, incrementing the loop variable **`i`**.
6. **`j Loop`**: Unconditionally jumps back to the **`Loop`** label, restarting the loop.
7. **`Exit:`**: Label marking the end of the loop. The code following this label will be executed after the loop completes.

Control Flow

- Set on less than (slt)
- this can be used to compare two registers for greater or less than
- no set greater than because the simpler the hardware the better the clock speed (**lightning fast**)

```
if  $s3 < $s4 then
      $t0 = 1
else
      $t0 = 0
```

slt $t0, $s3, $s4

- comparing constants with variables we use slti
- `slti $t0,$s2,10`

this is the same as saying

- `t0 = 1 if s2 < 10`

Tuesday Feburary 16th Lecture 6

****

Most of the content was from lecture slides 5 so I includd it in the previoous day! Happy assembly!

- check the MIPS cheat sheet and add it here and add the table from the last slide in lecture 5

Tuesday Feburary 16th Lecture 6

Functions

Function call or Procedures

- A function makes code reusable

Memory Map of Running a Program

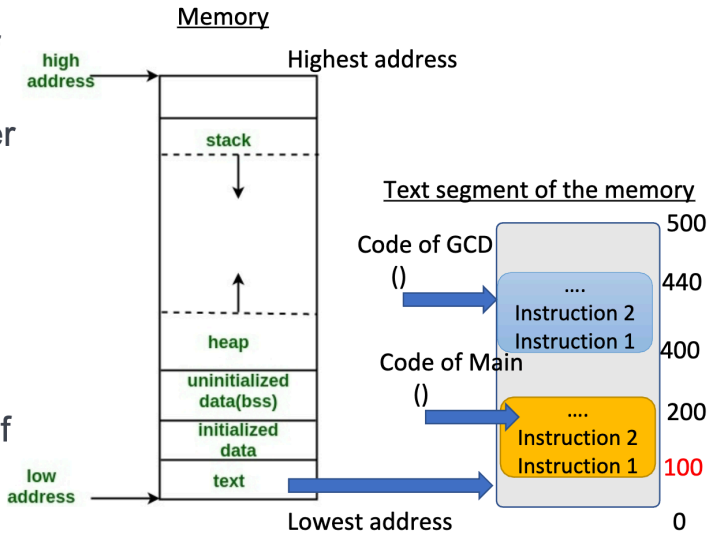
Memory Map of a Running Program

Program Counter (PC):

- A register that holds the address of the instruction to be executed
- Usually, the PC is incremented after fetching an instruction

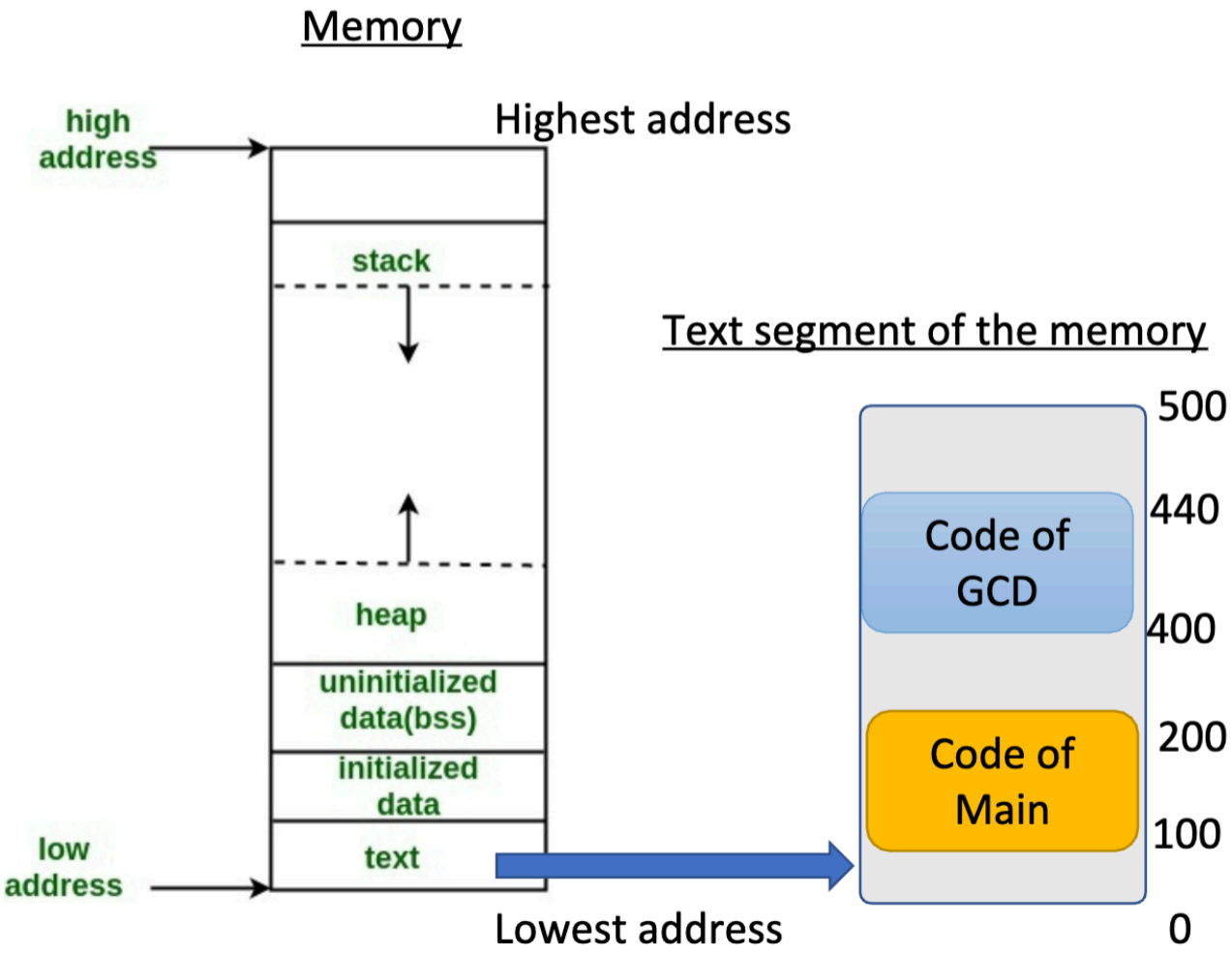
Example:

- Main() starts from address 100
- PC=100
- When instruction 1 is fetched, PC=PC+4=104, which is address of instruction 2



4

1. Static
    - Text/Code segment
    - Intialized data segment
    - Unintialized data segment
2. Stack
3. Heap



- 3

  Program Counter (PC)
  - Register that holds addresses of instructions and incremented after fetching instruction

Caller and Callee

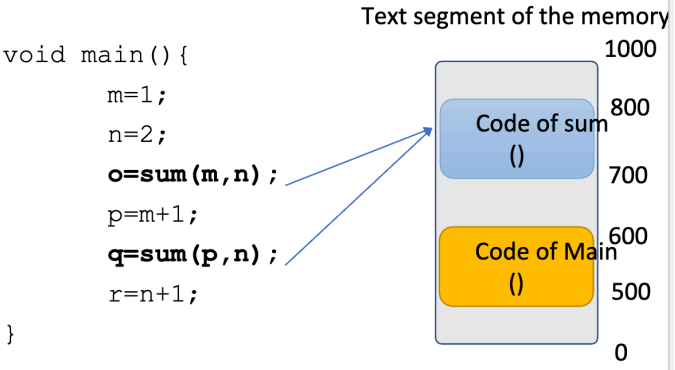
- if main() calls sum()
  - main is caller and sum is callee
  - Caller use the same register set as teh callee

Callign a Fucntion

Calling a function

- When a callee is executed, the flow of execution jumps to a different segment of the memory
  - Performs jump instruction to a memory address
- Main() starts from address 500
- Sum() starts from address 700
- **Questions:**
  1. What instruction to use to execute a jump to a callee function
  2. How can we return back to the caller function?

```
void main(){
        m=1;
        n=2;
        o=sum(m,n);
        p=m+1;
        q=sum(p,n);
        r=n+1;
}

Int sum (int a, int b){
        return a+b;
}
```



Instructions to use to execute a jump

Instruction to use to execute a jump

- **Use jump and link instruction (jal):**
  - Written as: ***jal callee_Address***
- **Link means that the link (address) to go back to the caller is preserved**
  - In return address register $ra
  - Now we can go back to main() after execution of sum() completes
- **What exactly is being stored in $ra ?**
  - The address of next instruction, from where the procedure was called
  - Example: Sum() was called from address 500
  - Thus, $ra=Current PC+4=500+4
  - After that, PC becomes the address of sum()

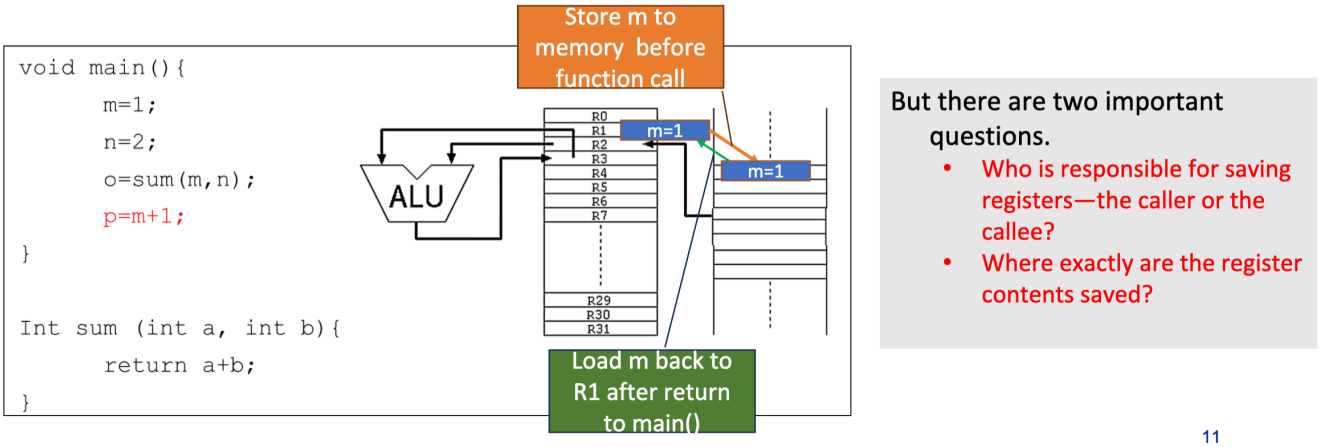
```
void main(){
        m=1;
        n=2;
        o=sum(m,n); ---Address 500
        p=m+1;
}

Int sum (int a, int b){
        return a+b;
}
```

7

Register Management

Register Management during function call

- The CPU has a limited number of registers for use by all functions, and it's possible that several functions will need the same registers.
- We can keep important registers from being overwritten by a function call, by saving them before the function executes, and restoring them after the function completes.



But there are two important questions.

- Who is responsible for saving registers—the caller or the callee?
- Where exactly are the register contents saved?

11

Answerin Questions on the previous slide

- Who is responsible for saving important registers across function calls
  - The **caller** knows which registers are important sand should be saved
  - The **callee** knows what registers it will use and overwrite
- However, inthe rtpical "black box" proramming approach teh caller adn teh callee do not know anything about each others' implementation
  - Difference functions can be written by different people but should stil run regardless when compiled

So how do functins cooperate and share registers when they don't know anything about each other?

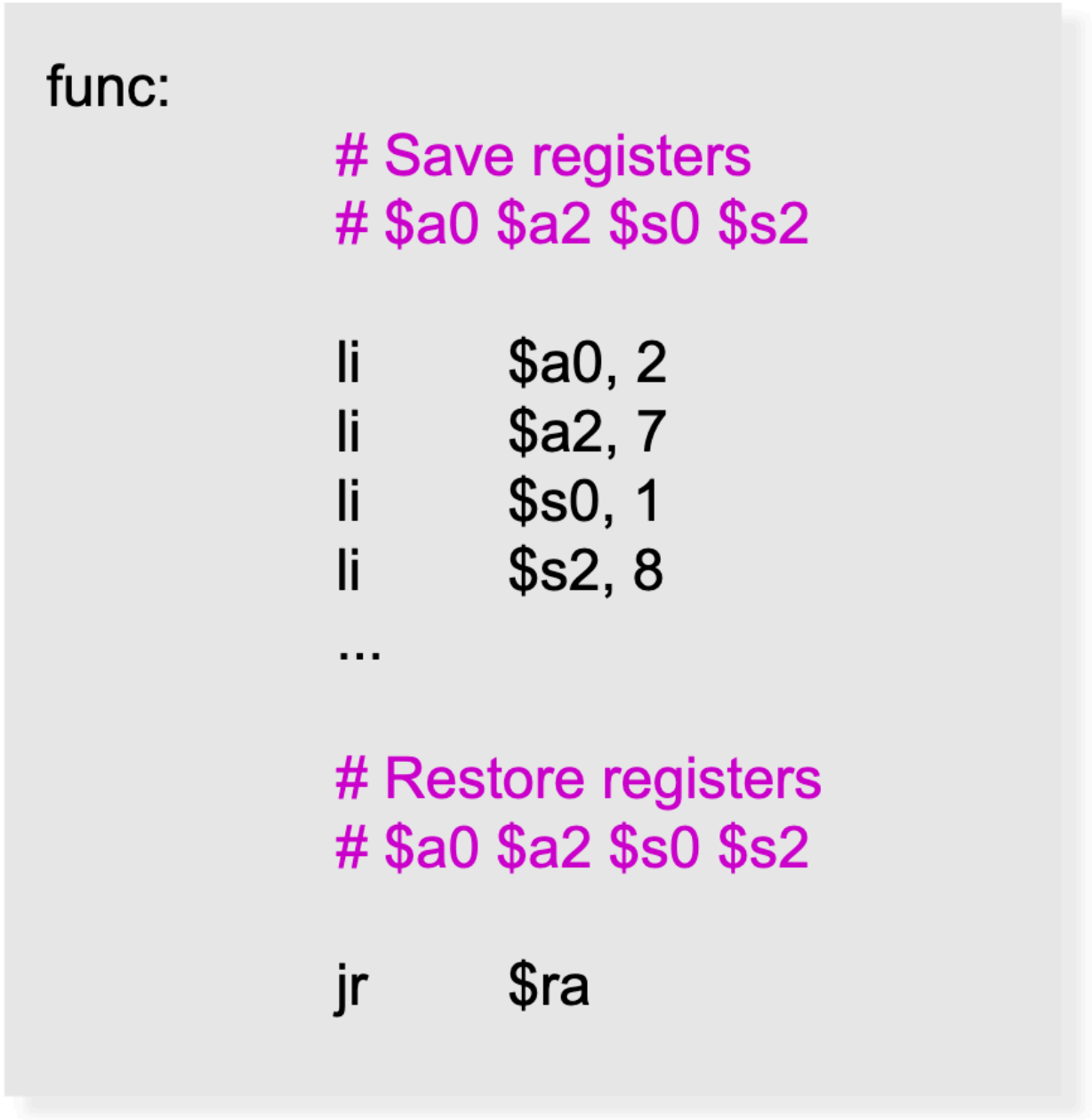
- **Approach 1:**
  - The caller saves registes to ensure they are not written

```
main:   li      $a0, 3
        li      $a1, 1
        li      $s0, 4
        li      $s1, 1

        # Save registers
        # $a0, $a1, $s0, $s1

        jal     func

        # Restore registers
        # $a0, $a1, $s0, $s1

        add     $v0, $a0, $a1
        add     $v1, $s0, $s1
        jr      $ra
```

Note: The li is a pseudo instruction that loads an immediate value into a register.

- 13
- main is trying to preserve a0 a1 s0 s1 by giving it arbitrary values

- **Approach 2:**
  - Callee save all registers



- Best Approach

Best Approach: Caller and Calle work together

- MIPS uses conventions again to split the register spilling chores.
- The *caller* is responsible for saving and restoring any of the following caller-saved registers that it cares about.

  $t0-$t9 $a0-$a3 $v0-$v1

  In other words, the callee may freely modify these registers, under the assumption that the caller already saved them if necessary.

- The *callee* is responsible for saving and restoring any of the following callee-saved registers that it uses. (Remember that $ra is "used" by jal.)

  $s0-$s7 $ra

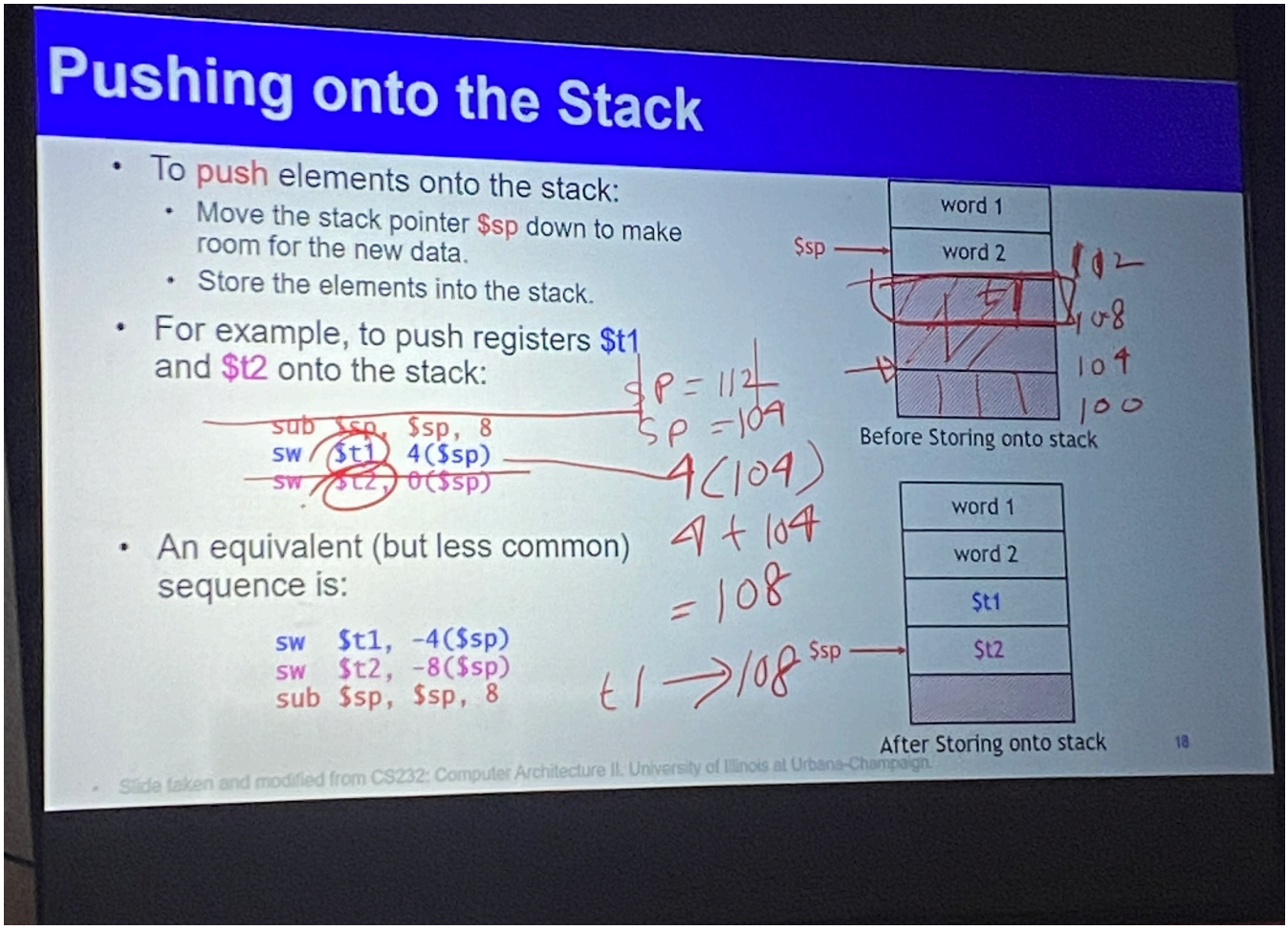
  Thus the caller may assume these registers are not changed by the callee.

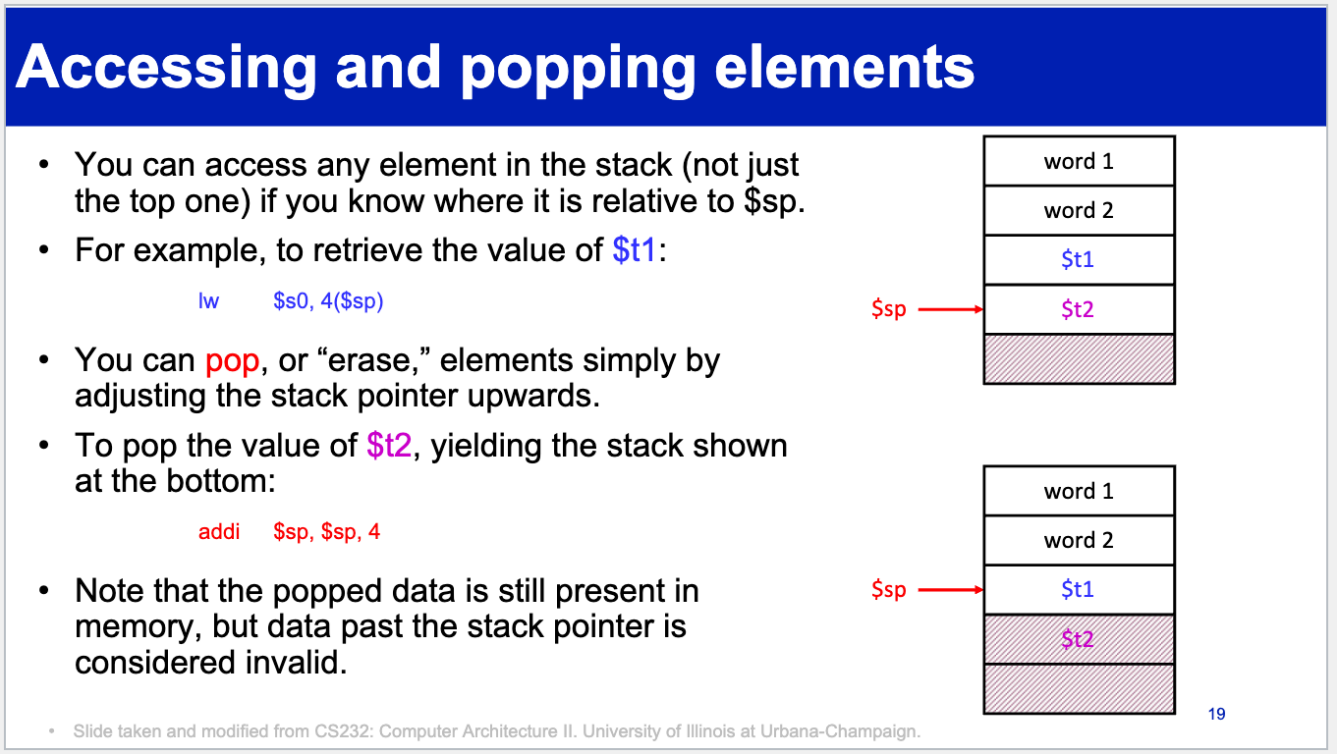
- Be especially careful when writing nested functions, which act as both a caller and a callee!
- $ra is tricky; it is saved by a callee who is also a caller.

15

- Slide taken and modified from CS232: Computer Architecture II. University of Illinois at Urbana-Champaign.

Accessing and Popping Elements

- 

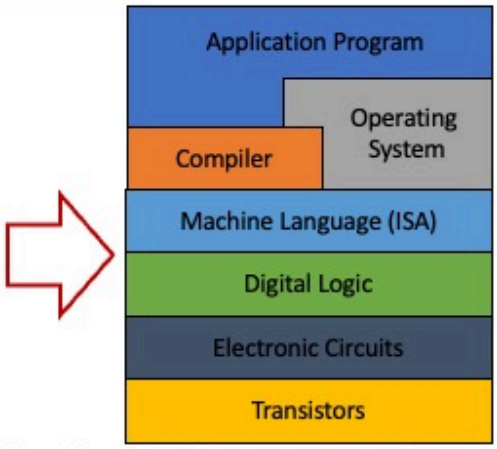


Tuesday Feburary 13th

Introduction to computing systems

I got super lazy here are all the slides LOL this should print nicely for exam

Context



- Chapter 4:Introduction to Computing Systems: From Bits & Gates to C & Beyond, Yale Patt

2

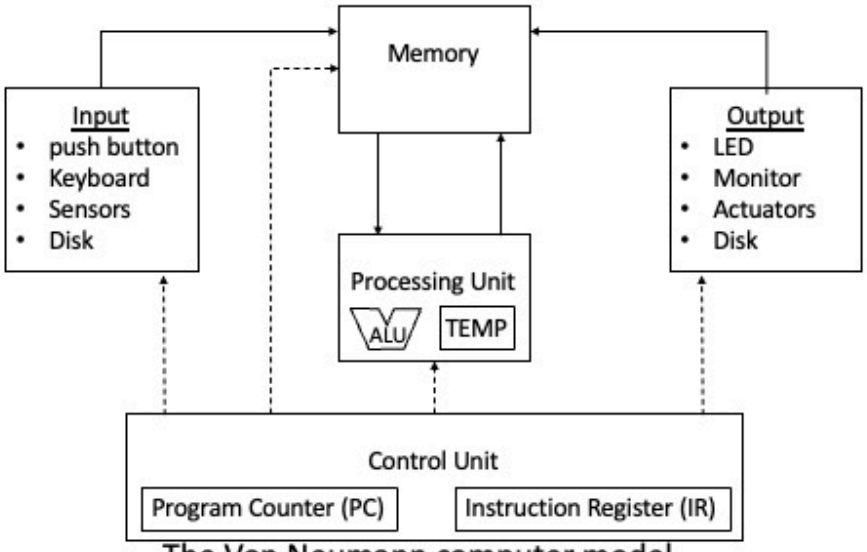
Fundamental model of computer

For preforming a task by a computer we need:

- A program that specifies the task
  - Instructions: smallest piece of work specified in a computer
- Data to operate on
- A computer that carry out the task
  - Von Neumann model: a fundamental model of computer for processing instructions

3

The Von Neumann computer model



The Von Neumann computer model

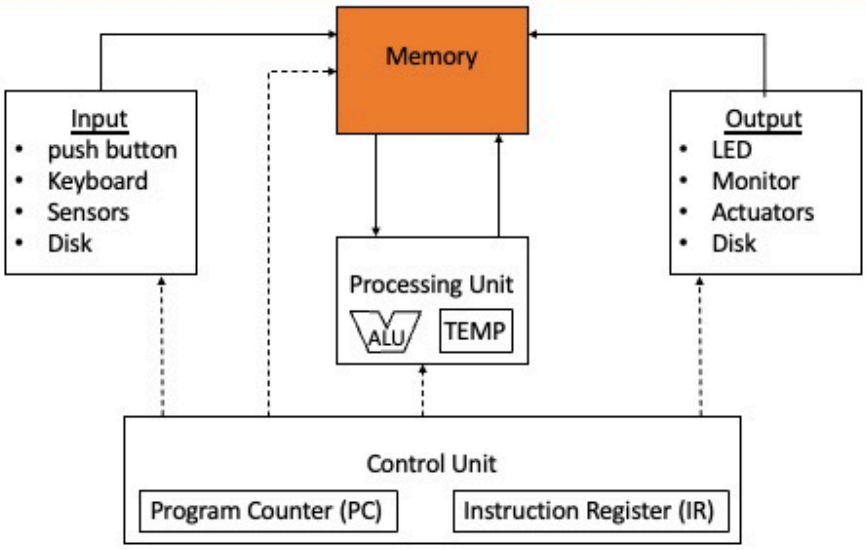
4

Von Neumann model has five parts:

- Memory ← Computer program and data resides here
- Processing Unit ← Processing is carried out here e.g., Add, Sub, compare, …
- Control unit ← Keep track of where we are in the process of running a program
- Input ← Provide the information for processing
- Output ← Store processed data or actuate upon the data

5

Memory



The Von Neumann computer model

6

Memory and Address

The memory contains bits that stores:
- Program (instructions)
- Data

Addressability: How many bits each address stores
- E.g., 8-bit addressable or "byte addressable"
  - Each address holds 8 bit

Address space:
- Total number of addresses of a memory
- Depends on the size of the address→ n bit address can represent 2^n unique memory locations

7

Example Memory

1 byte

Address	data
0x00	0x00
0x01	0xF1
0x02	0x11
...	...
0xFE	0x20
0xFF	0x34

What is the total storage capacity in terms of bits of a memory with:
- 8-bit addresses
- Addressability of 16 bits?

Address space: of 2^8 unique memory locations

Addressability: 8 bits or *byte addressable*

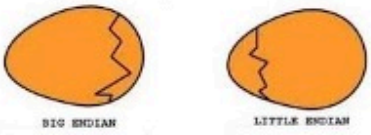
8

How to store multi-byte variables in memory?

- Multi-byte: Int, long etc.
- int var = 0x49345678;

MSB LSB

Do you start eating your boiled egg from big end or little end?



Big Endian
(start by storing **big end** or **MSB**)

Address	data	
0x00	0x49	← MSB
0x01	0x34	
0x02	0x56	
0x03	0x78	← LSB
...	...	

Little Endian
(start by storing **little end** or **LSB**)

Address	data	
0x00	0x78	← LSB
0x01	0x56	
0x02	0x34	
0x03	0x49	← MSB
...	...	

https://www.youtube.com/watch?v=NcaiHcBvDR4&ab_channel=Computerphile

9

Little Endian vs. Big Endian

Does it really matter?

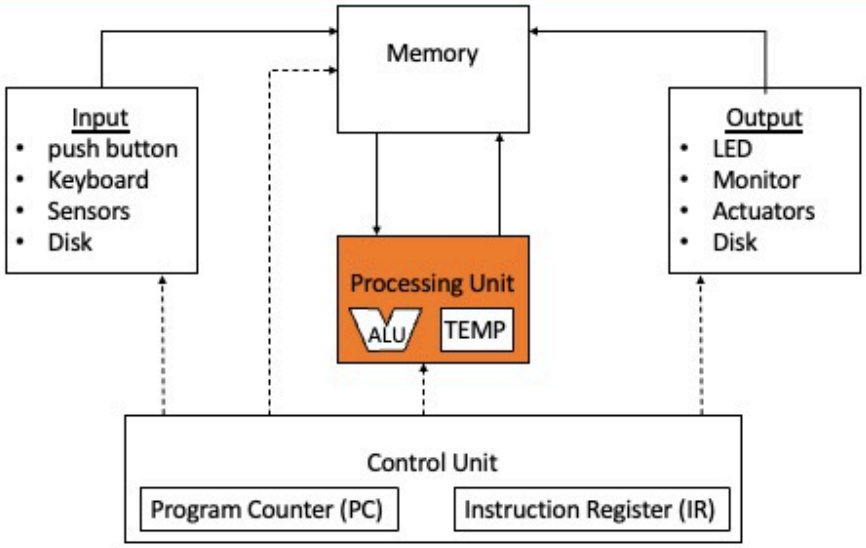
- Only when two systems with different endianness want to share data
  - Older systems were generally big endian
  - Intel x86→ little endian
  - ARM architecture was little-endian before version 3

How different data types are loaded?

Category	Name	Fmt	RV32I Base	
Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm

10

Processing Unit



The Von Neumann computer model

11

Processing Unit

Functional units

- ALU = Arithmetic and Logic Unit
- E.g., ADD, SUB, Not, XOR, Multiply, Shift, Compare, …

Registers

- Small temporary storage
- Operands and results of functional units
- RISC-V and MIPS32 has 32 registers

12

Registers

Registers are fast but small

- Memory is large but slow

Register file

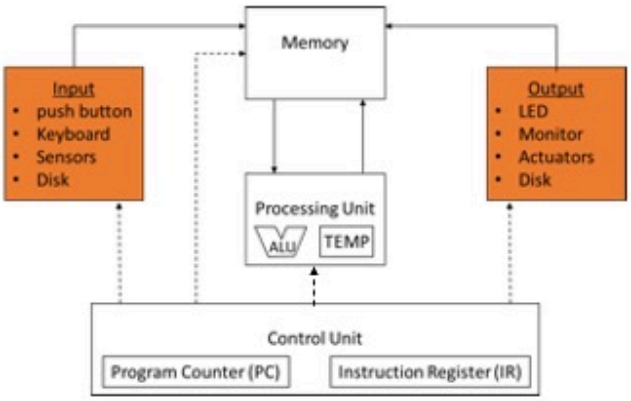
- A structure in CPU that holds all the registers
- r0-r31 in MIPS

NAME	NUMBER
$zero	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$k0-$k1	26-27
$gp	28
$sp	29
$fp	30
$ra	31

MIPS Register Names

13

Input/Output

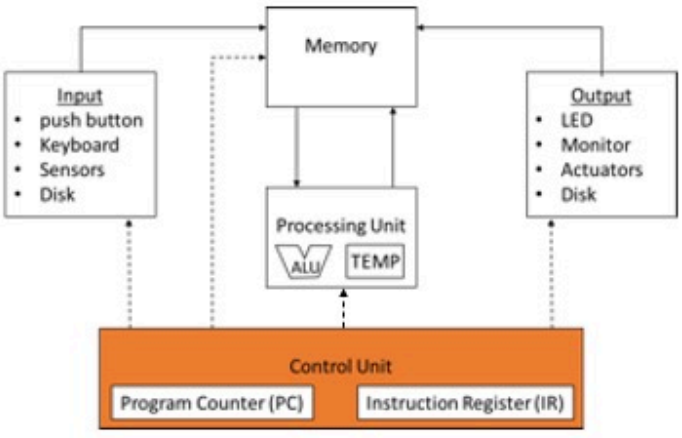


- Input and output devices (i.e., peripherals) are used for getting data into and out of computer memory
- Usually each device has a set of registers to communicate with the CPU

The Von Neumann computer model

14

Control Unit

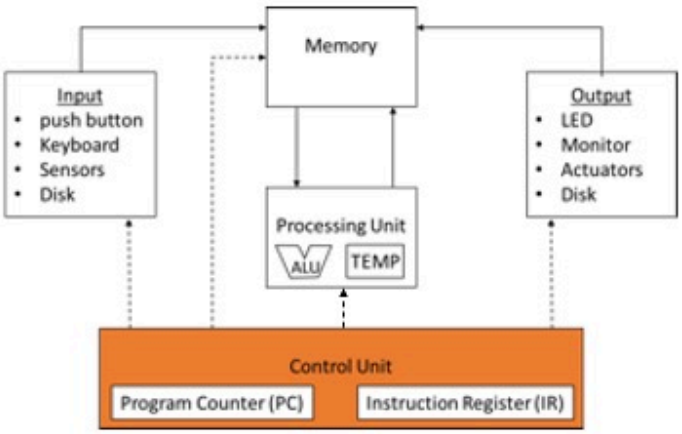


- Control unit orchestrates the execution of each instruction
- Instruction Register (IR): the current instruction being executed on the CPU
- Program Counter (PC): the address of the next instruction to be executed

The Von Neumann computer model

15

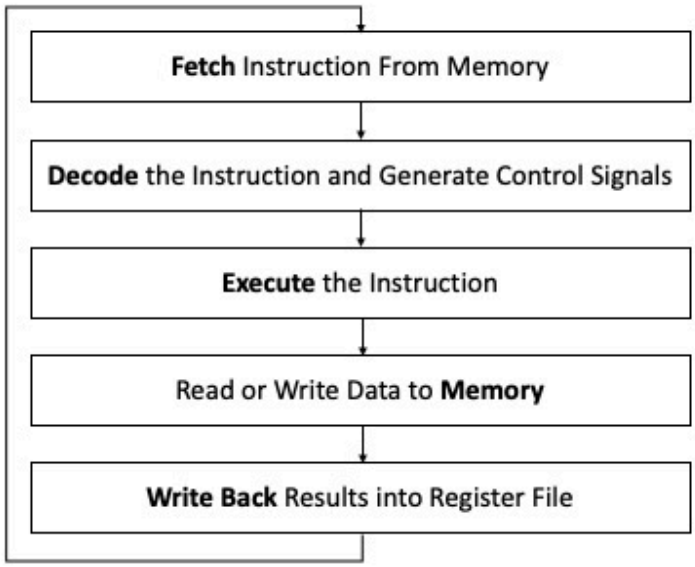
Function of Control Unit



- Read an instruction from memory into Instruction Register (IR)
  - By loading the memory address pointed by PC
- Decodes the instruction and generate control signals that tell other components what to do

16

Instruction Processing



17

Instruction

Smallest unit of execution in a processor

- Carries two pieces of information
  - Opcode: operation to be performed
  - Operands: data to be used for operation
    - Registers, immediate values, memory addresses
- Instructions are sequences of bits
- Instruction Set Architecture (ISA) is the collection of all instructions that a processor supports

18

Instruction Example

Assembly: `addi $sp, $sp, -32` ⇨ Machine code: `0x23bdffe0`

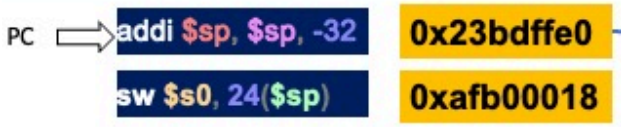
opcode	rs	rt	immed
addi	$sp	$sp	-32
001000	11101	11101	1111111111100000

`sw $s0, 24($sp)` ⇨ `0xafb00018`

opcode	rs	rt	immed
sw	$sp	$s0	24
101011	11101	10000	0000000000011000

19

Instruction in memory



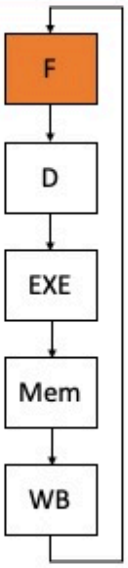
Machine code in Main Memory	
Address	Data / Instructions
0x00	0x23
0x01	0xbd
0x02	0xff
0x03	0xe0
0x04	0xaf
0x05	0xb0
0x06	0x00
0x07	0x18
0x08	....

- Q1: What is the value of PC before any instruction is fetched?
- Q2: What is the endianness of the processor?

20

Instruction Processing: Fetch

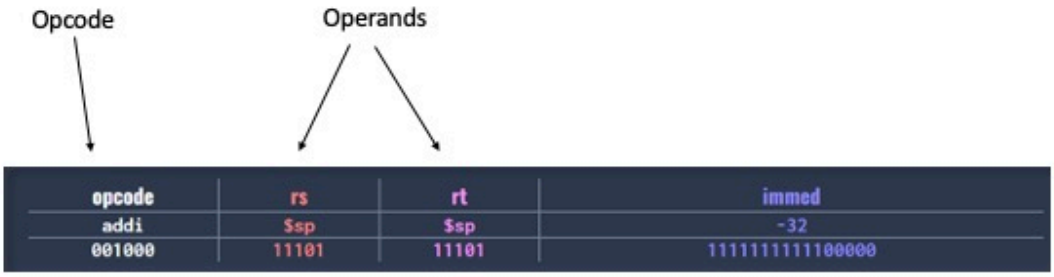
- Read next instruction (pointed by PC) from memory into IR
- Increment PC (so that it points to the next instruction in memory)
- Question: How much should you increment the PC after fetch stage in a processor that is **byte addressable** and the width of each instruction is **8 Bytes**?

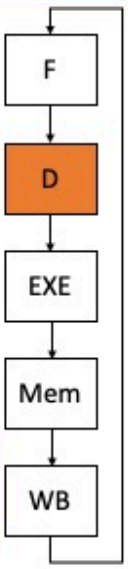


21

Instruction Processing: Decode

1. Identify the opcode and operands
2. Generate control signal to perform the tasks

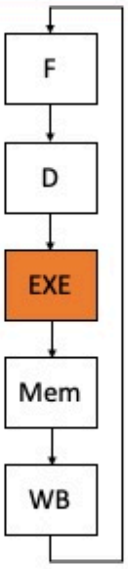




22

Instruction Processing: Execute

- Perform the operation using the source operands
- E.g., addi $sp, $sp, -32



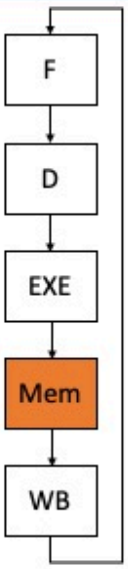
23

Instruction Processing: Memory Operations

- If the instruction is a load or store, then perform loading from or storing to memory

```
sw $s0, 24($sp)
```

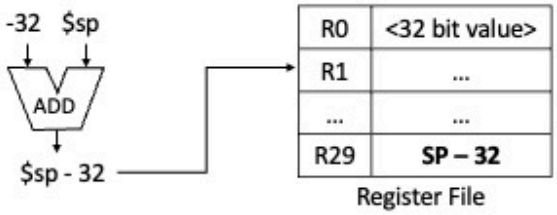
- Load/store also involves ALU operation in EXE stage. What ALU operation is that?



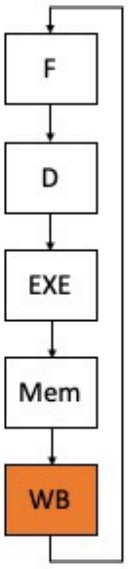
24

Instruction Processing: Write Back

- Update the register file with newly calculated results
- E.g., `addi $sp, $sp, -32`
- Here, $sp register needs update after the addi is performed



NAME	NUMBER
$zero	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$k0-$k1	26-27
$gp	28
$sp	29
$fp	30
$ra	31



25

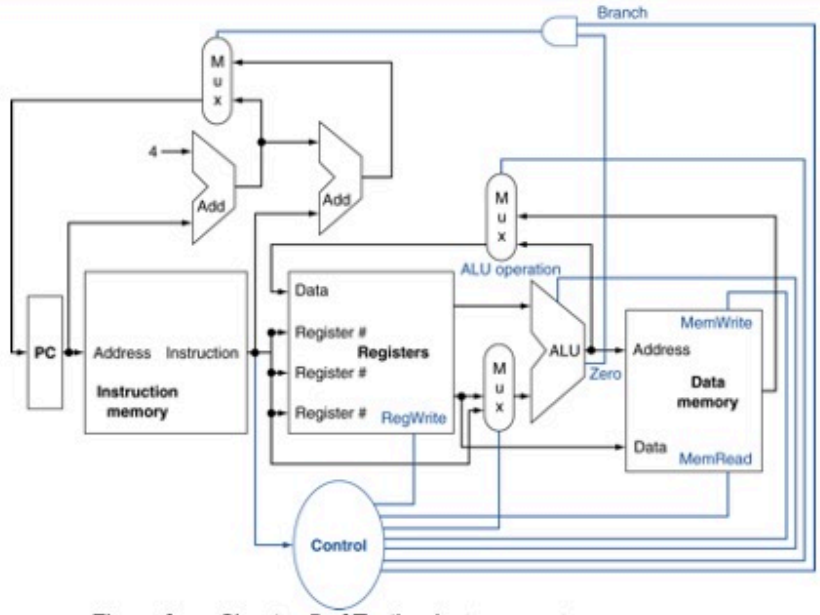
Updating PC for Jump instruction

But we do not always fetch from the next address!

- Control flow instructions (JAL, BEQ, etc.) set the PC value to something other than PC + 4
- E.g., JAL rd, imm_{20}
- Semantics
  - target_addr = PC + imm
  - GPR[rd] ← PC + 4
  - **PC ← target_addr**

26

Illustration of CPU Datapath for MIPS

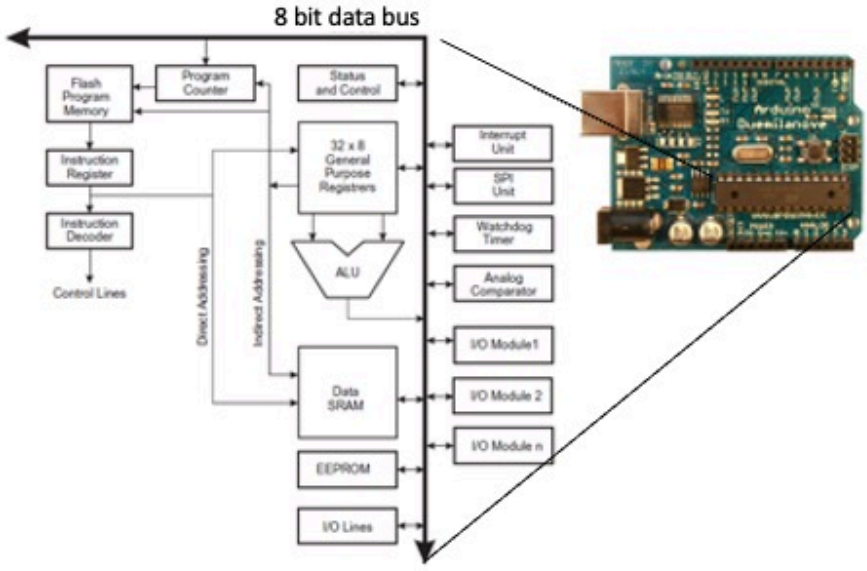


How this architecture operates for:

- Fetch
- Decode
- Execute
- Memory
- Write Back

Figure from Chapter 5 of Textbook: Computer Organization and Design, Revised Printing, Third Edition

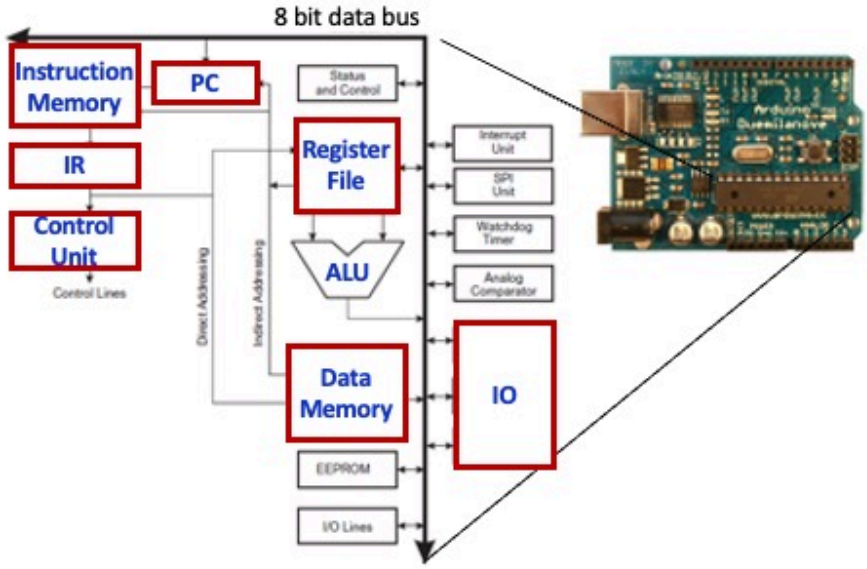
Case Study: Atmel ATmega168



Atmel ATmega168 datasheet

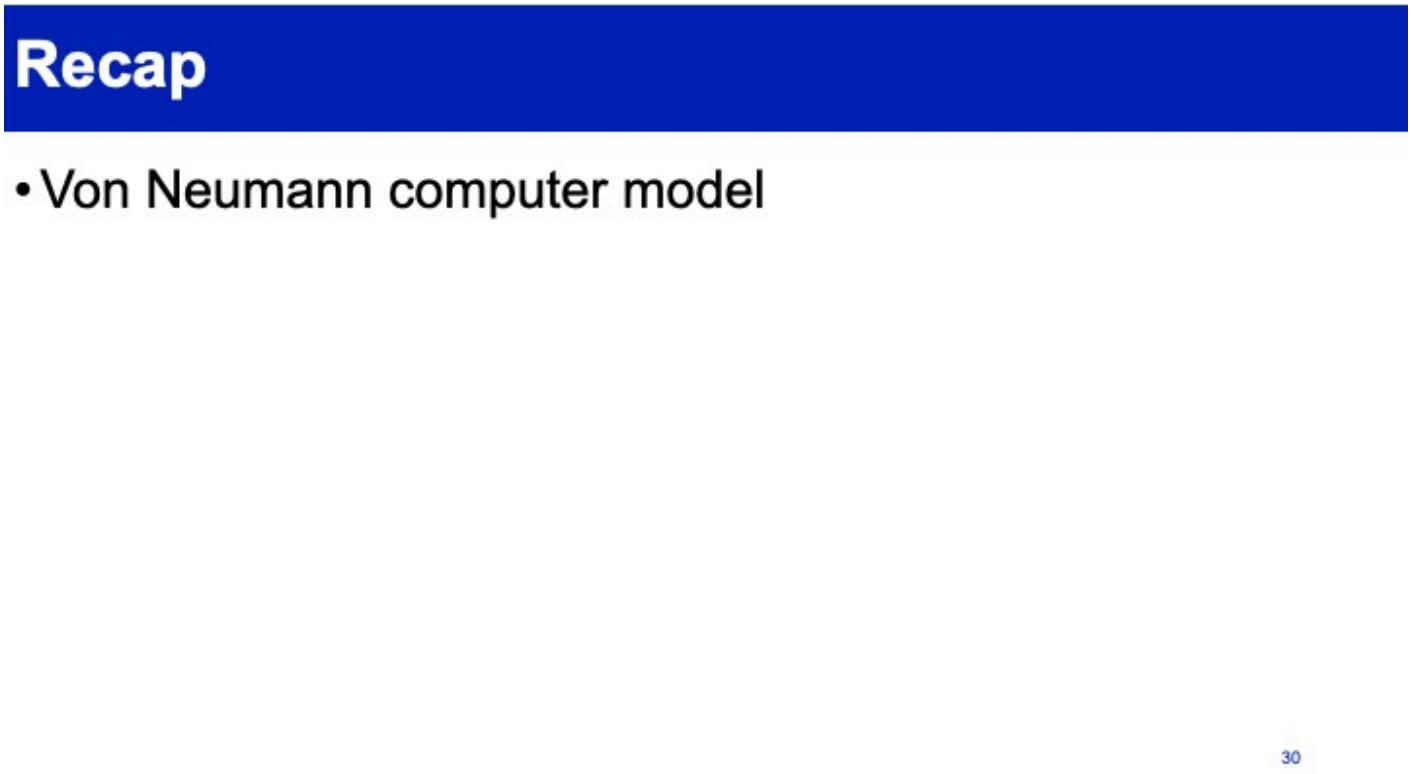
28

Case Study: Atmel ATmega168



Atmel ATmega168 datasheet

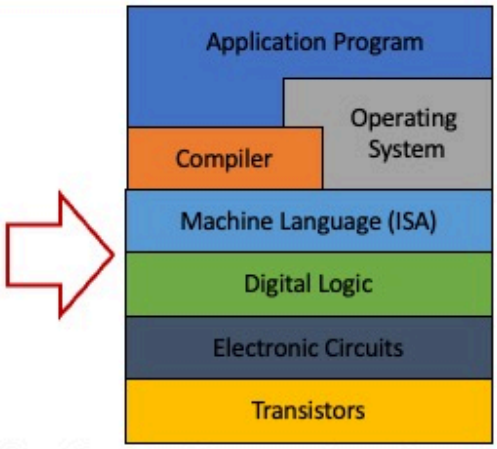
29



Thursday Feburary 15th

Lecture 8 Digital Logic and Design

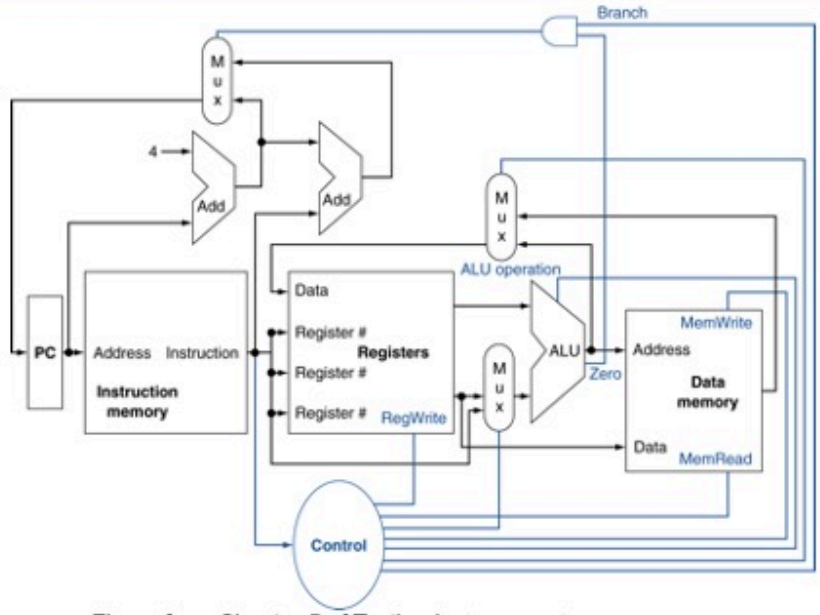
Context



- Chapter 4:Introduction to Computing Systems: From Bits & Gates to C & Beyond, Yale Patt

2

Illustration of CPU Datapath for MIPS

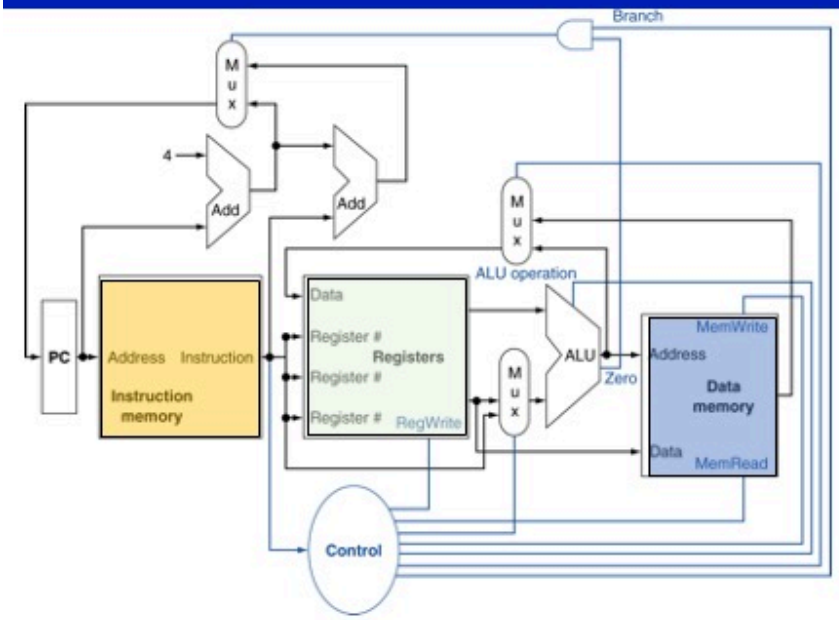


How this architecture operates for:

- Fetch
- Decode
- Execute
- Memory
- Write Back

Figure from Chapter 5 of Textbook: Computer Organization and Design, Revised Printing, Third Edition

Initial State



Register		
	$t0	0
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

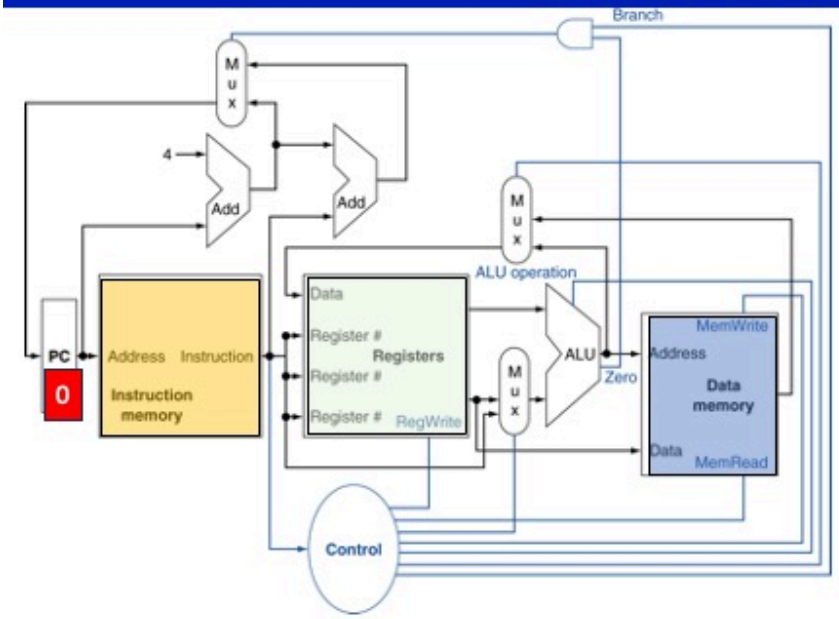
Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

Data Memory	Address	Data
	40	0
	44	0
	48	0

4

Fetch

F → D → EXE → Mem → WB

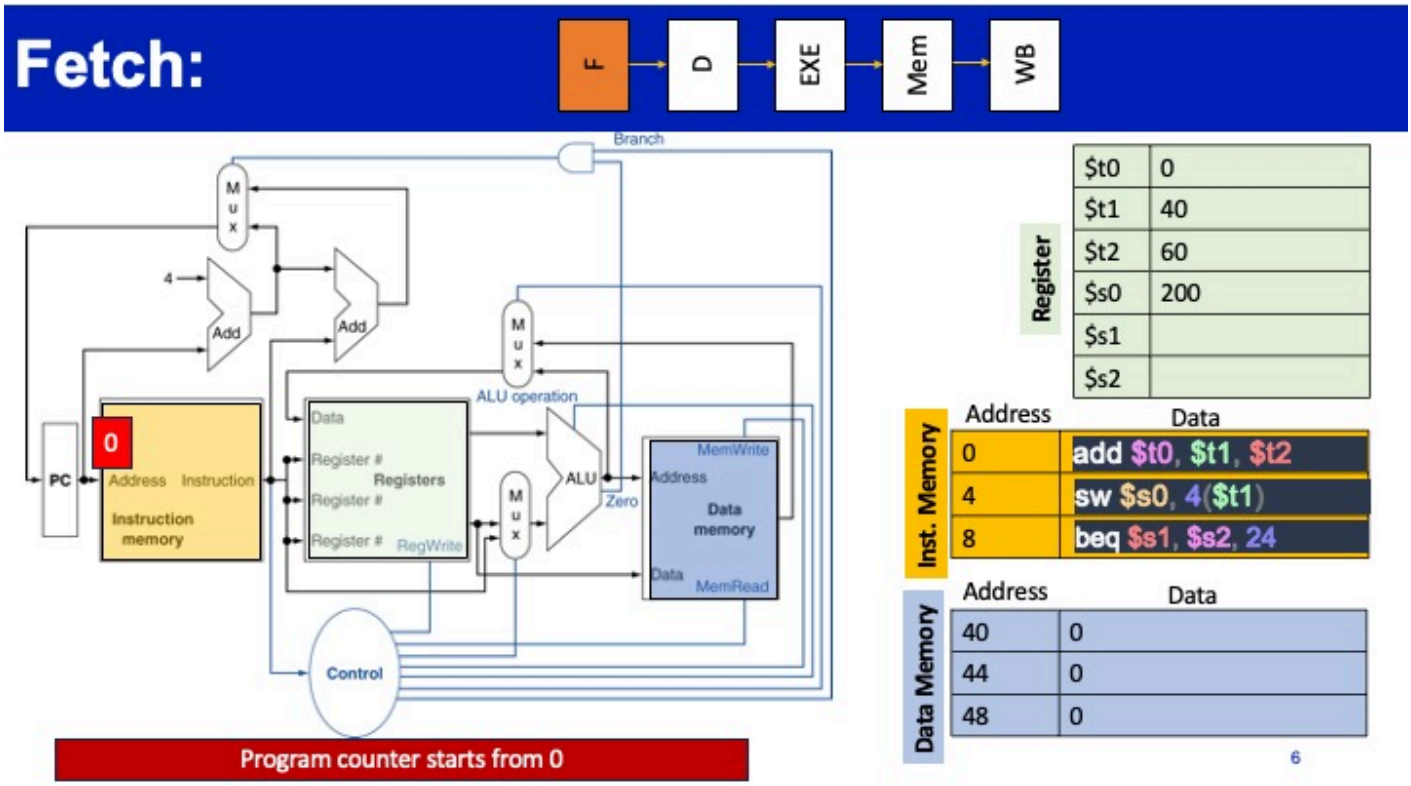


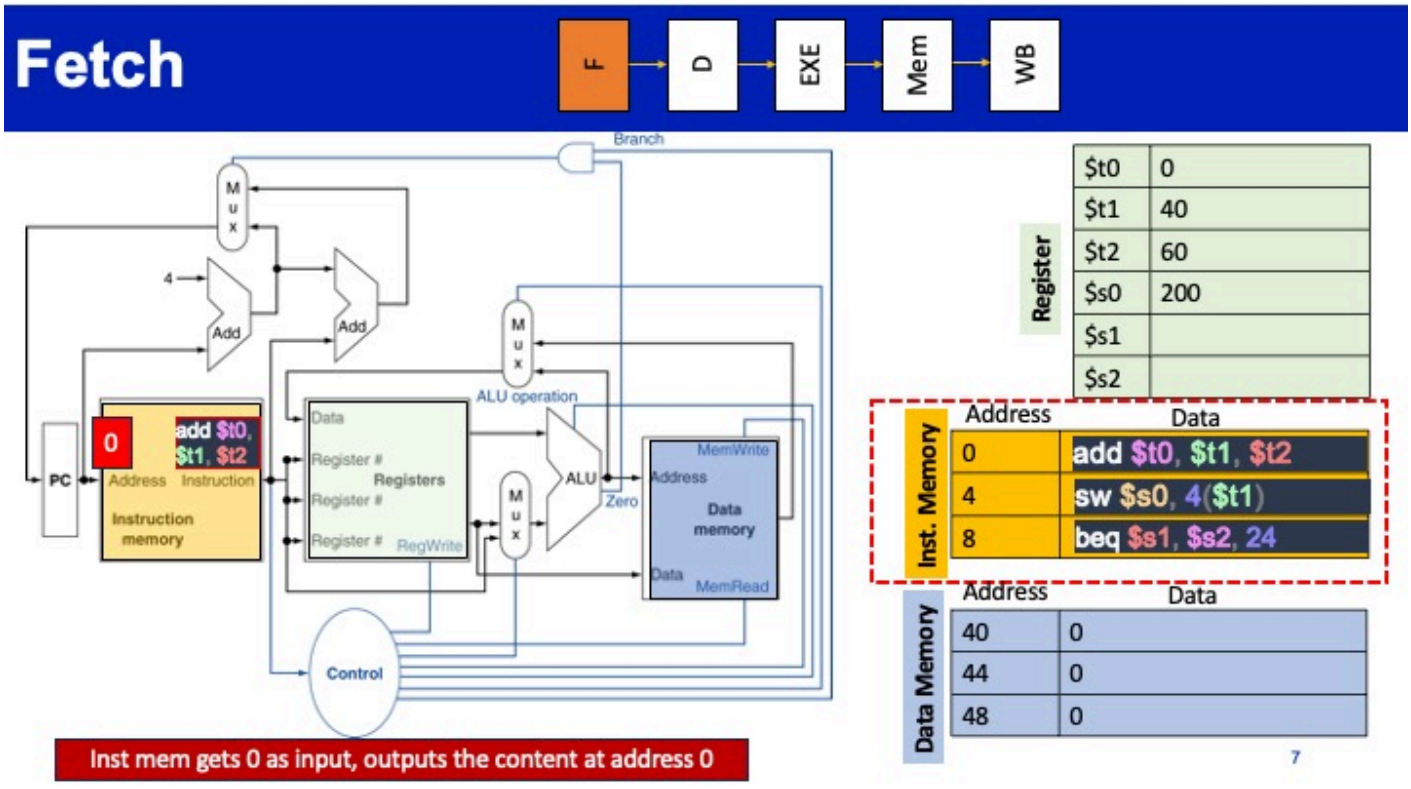
Register		
	$t0	0
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

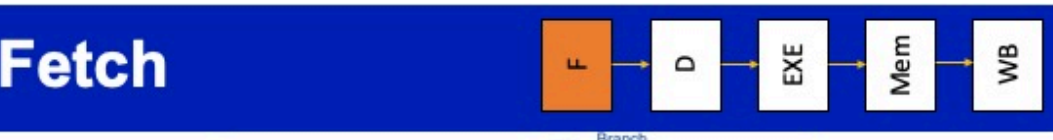
Data Memory	Address	Data
	40	0
	44	0
	48	0

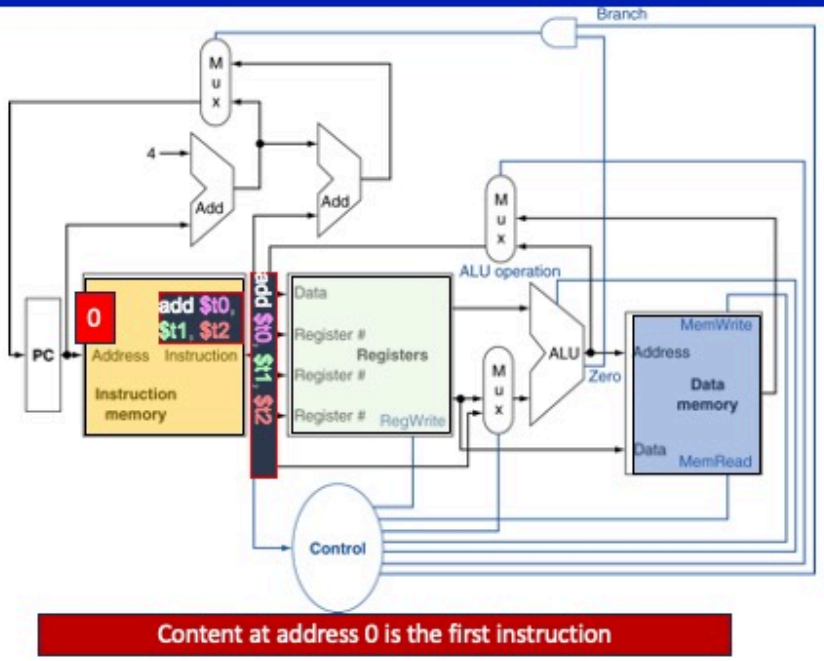
5


Fetch:
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
M u x
ALU operation
0
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control
Program counter starts from 0
Register
$t0 0
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
6


Fetch
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
M u x
ALU operation
0
add $t0, $t1, $t2
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control
Inst mem gets 0 as input, outputs the content at address 0
Register
$t0 0
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
7

Fetch





Content at address 0 is the first instruction

Register		
	$t0	0
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

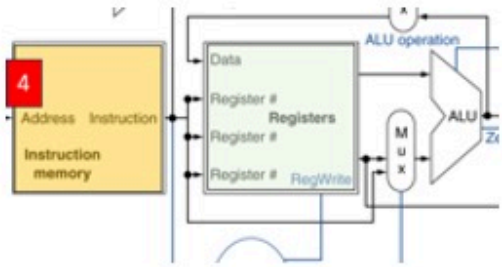
Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

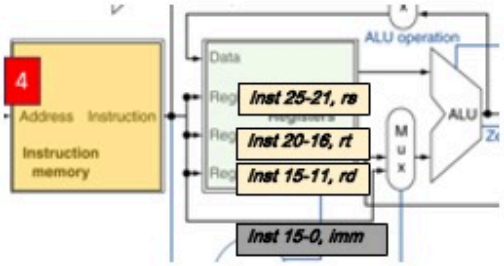
Data Memory	Address	Data
	40	0
	44	0
	48	0

8

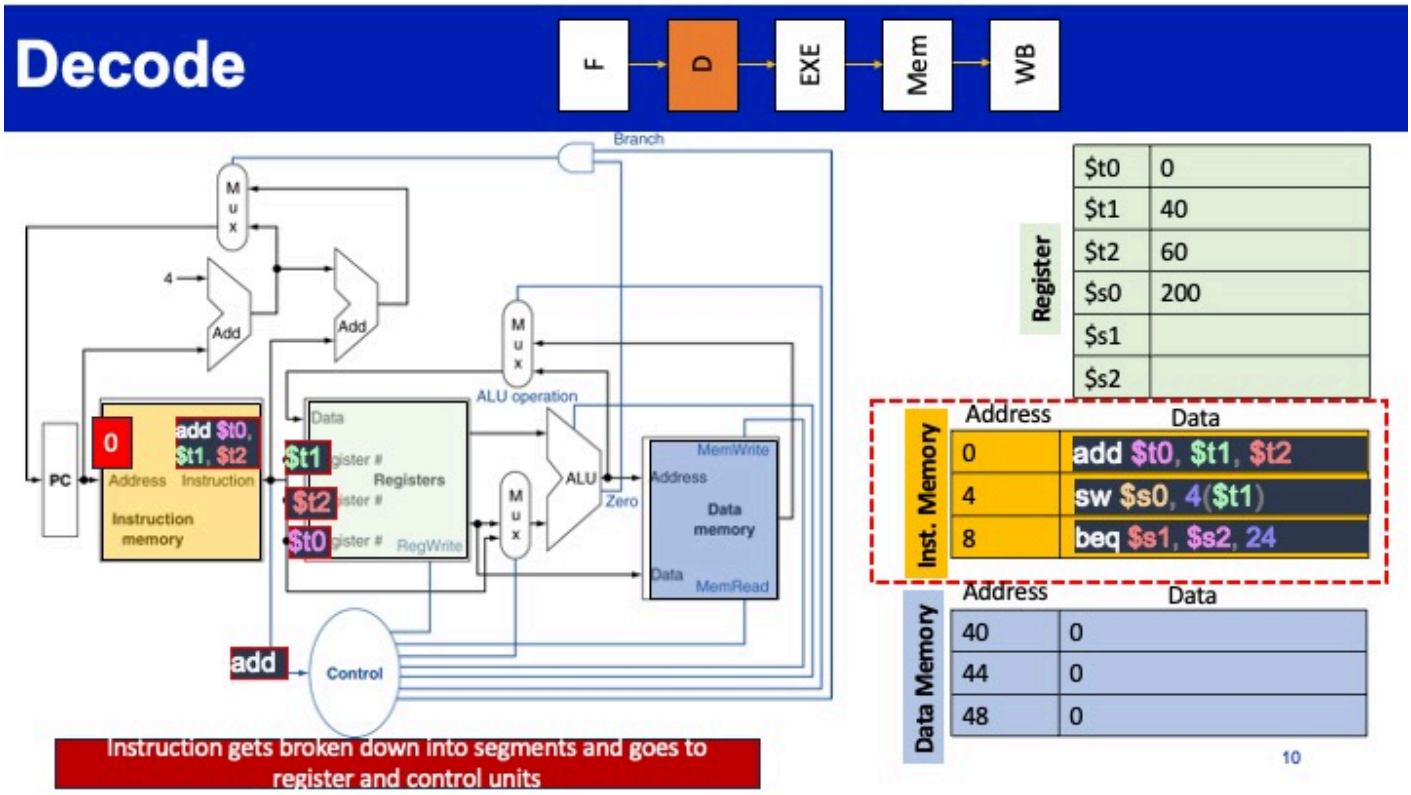
Decode for Add

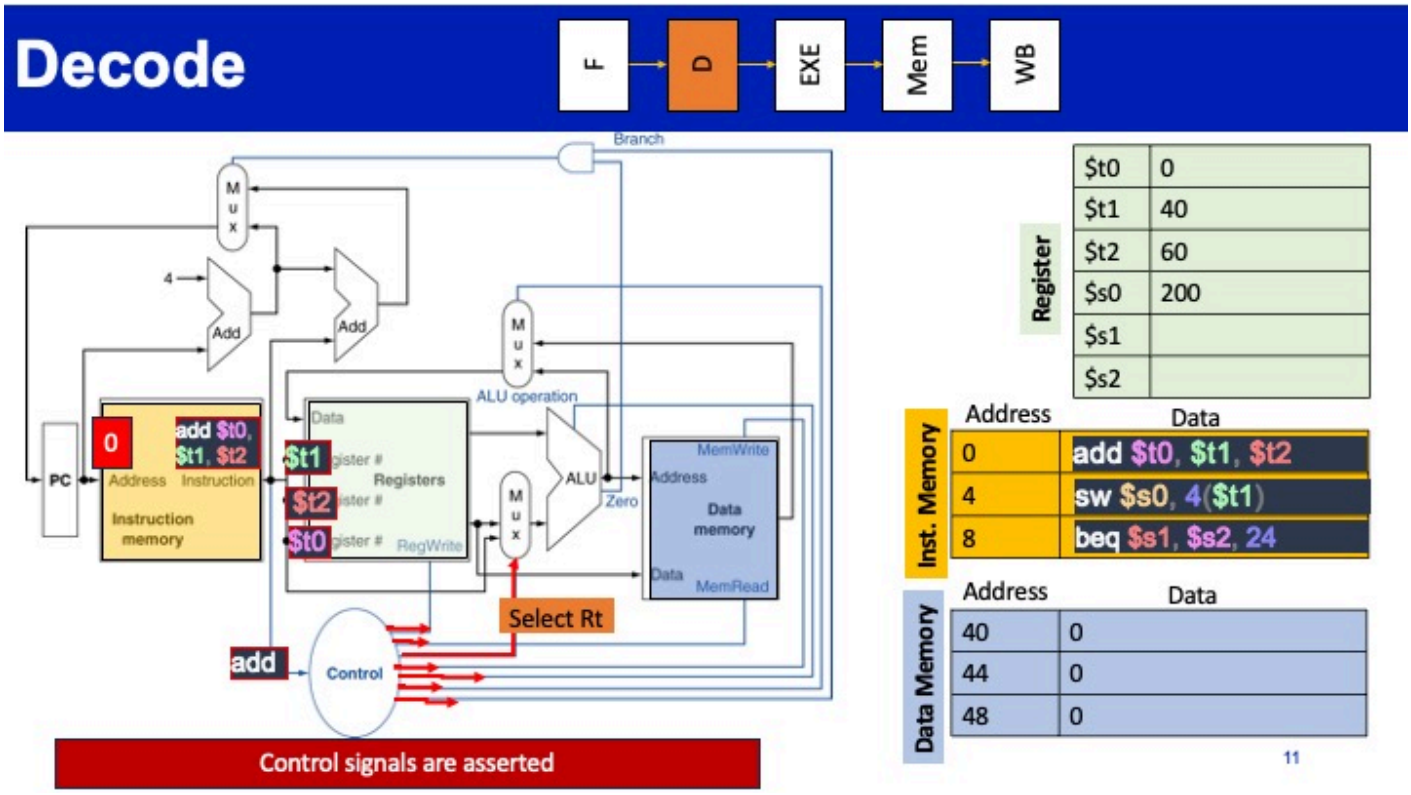
R-Type:
- add rd, rs, rt
- sub rd, rs, rt
- and rd, rs, rt
- or rd, rs, rt
- slt rd, rs, rt

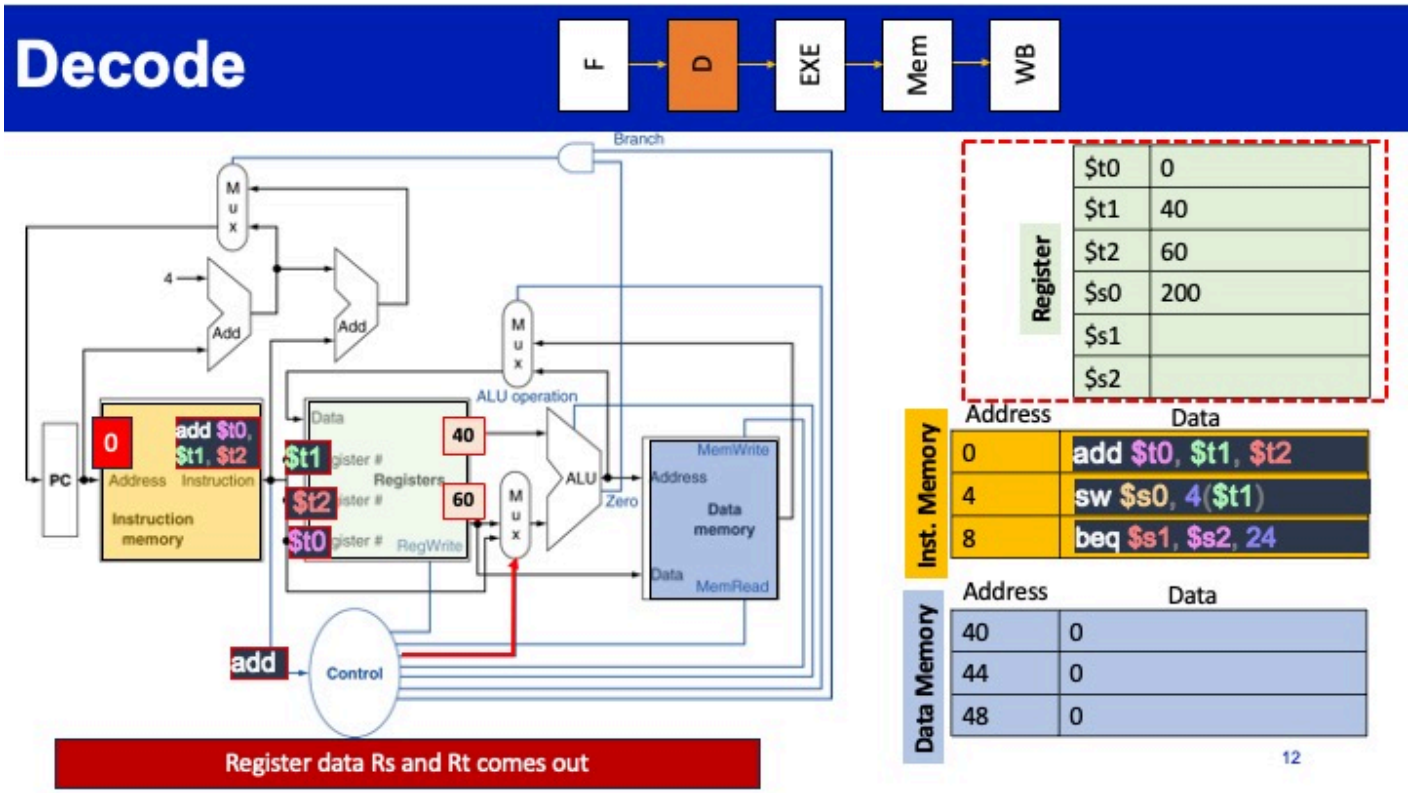


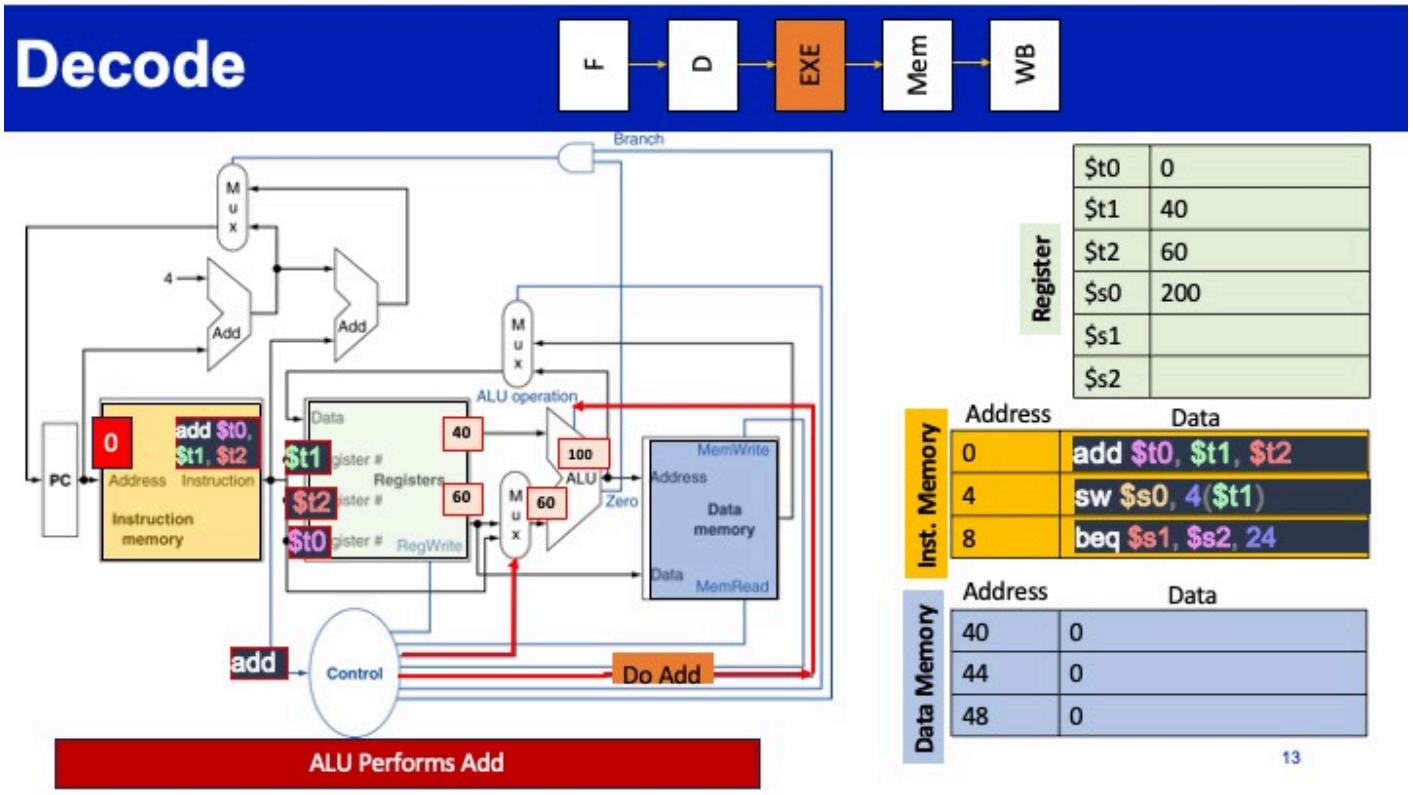


9

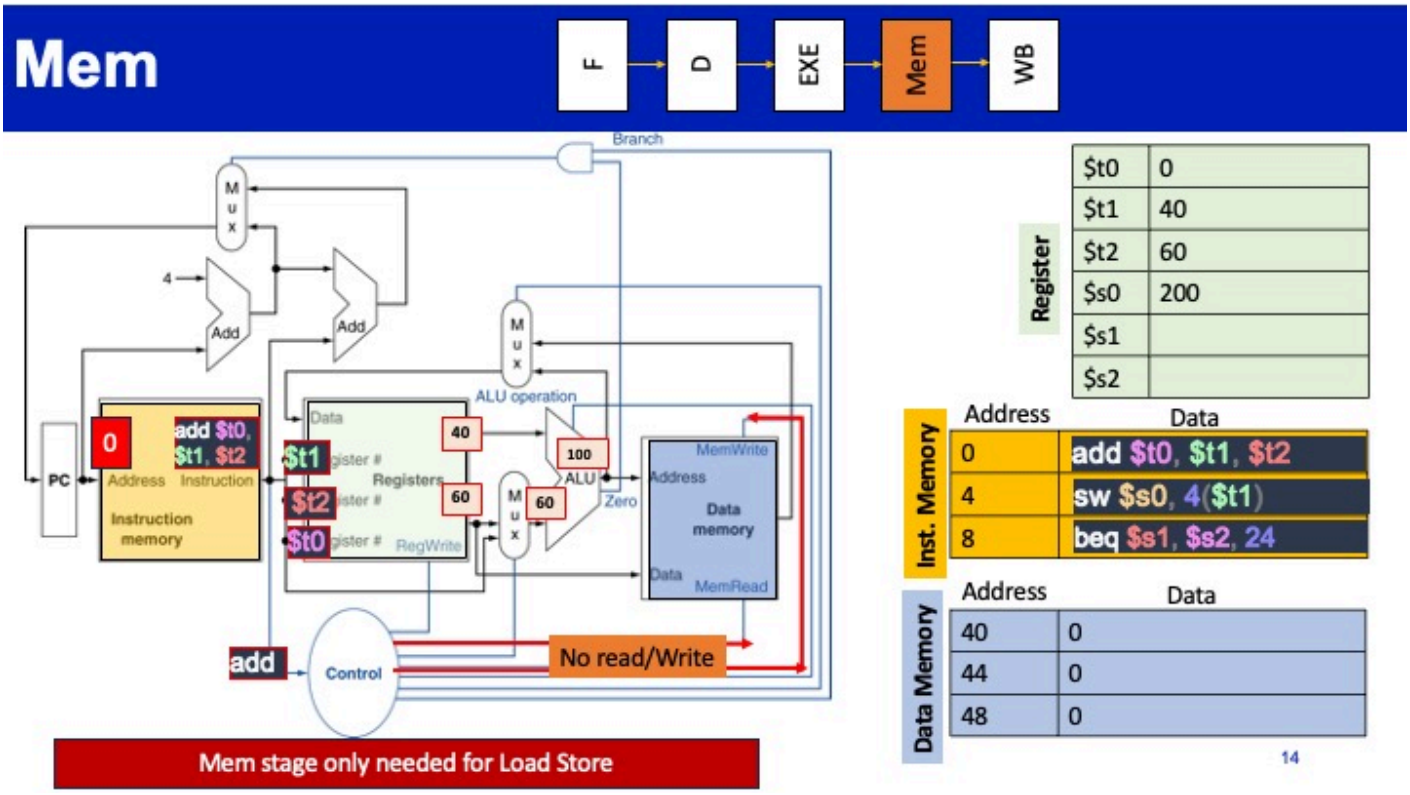

Decode
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
ALU operation
PC
0
add $t0, $t1, $t2
Address
Instruction
Instruction memory
Data
$t1
$t2
$t0
Registers
RegWrite
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
add
Control
Register
$t0 0
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
Instruction gets broken down into segments and goes to register and control units
10


Decode
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
ALU operation
PC
0
add $t0, $t1, $t2
Address
Instruction
Instruction memory
Data
$t1
$t2
$t0
Registers
RegWrite
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Select Rt
add
Control
Register
$t0 0
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
Control signals are asserted
11


Decode
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
M u x
ALU operation
PC
0
add $t0, $t1, $t2
Address
Instruction
Instruction memory
Data
$t1
$t2
$t0
Registers
RegWrite
40
60
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
add
Control
Register data Rs and Rt comes out
Register
$t0 0
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
12


Decode
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
M u x
ALU operation
PC
0
add $t0, $t1, $t2
Address
Instruction
Instruction memory
Data
$t1
$t2
$t0
Registers
RegWrite
40
60
100
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
add
Control
Do Add
ALU Performs Add
Register
$t0 0
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
13

Mem



Mem stage only needed for Load Store

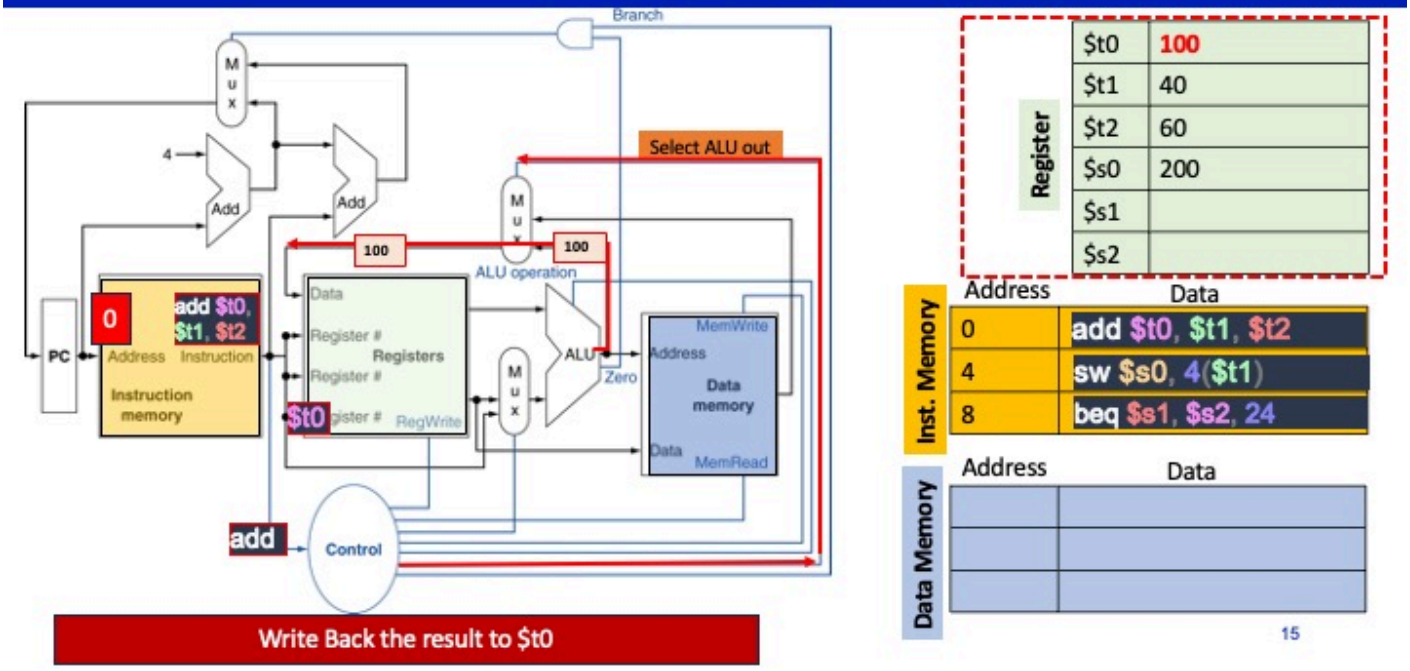
Register		
	$t0	0
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

Data Memory	Address	Data
	40	0
	44	0
	48	0

14

Write Back



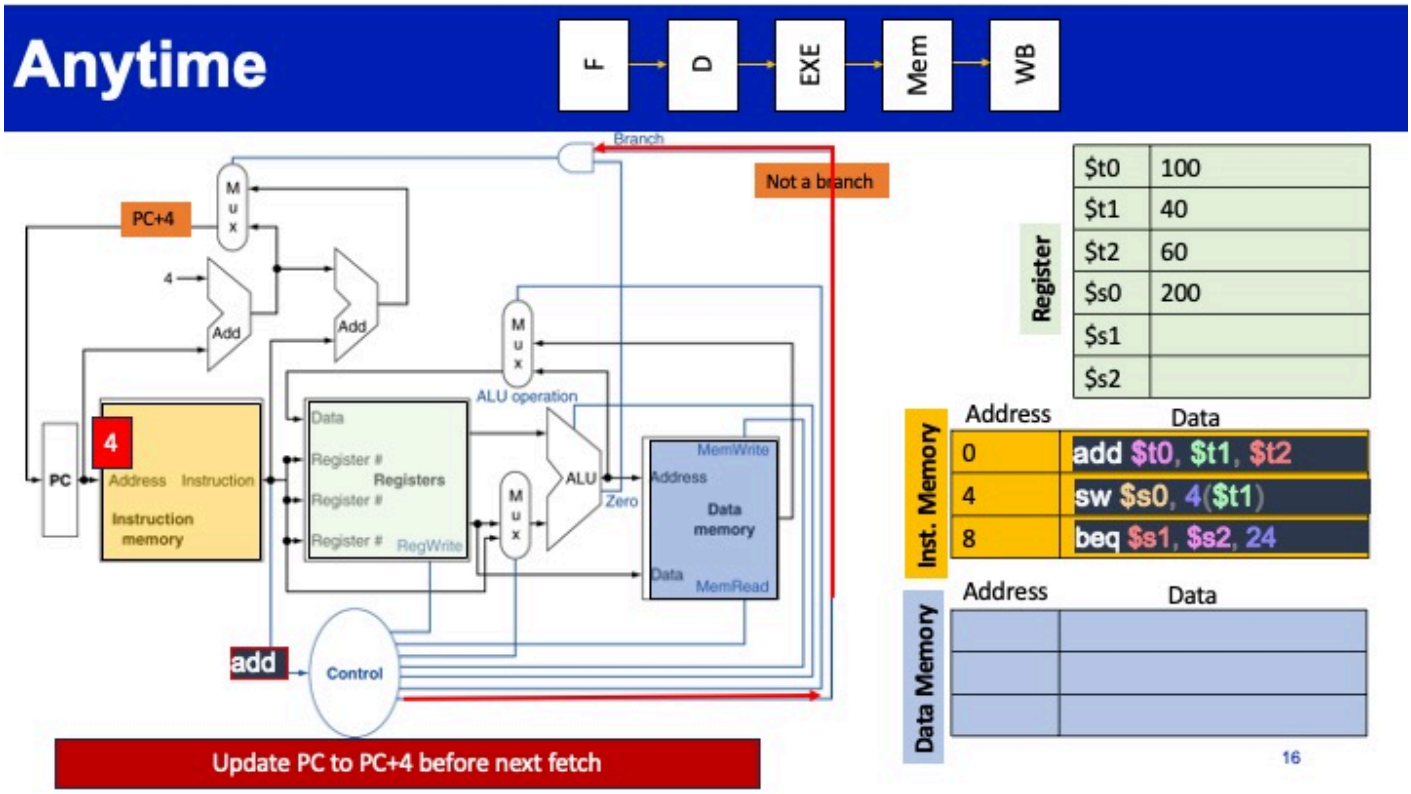
Write Back the result to $t0

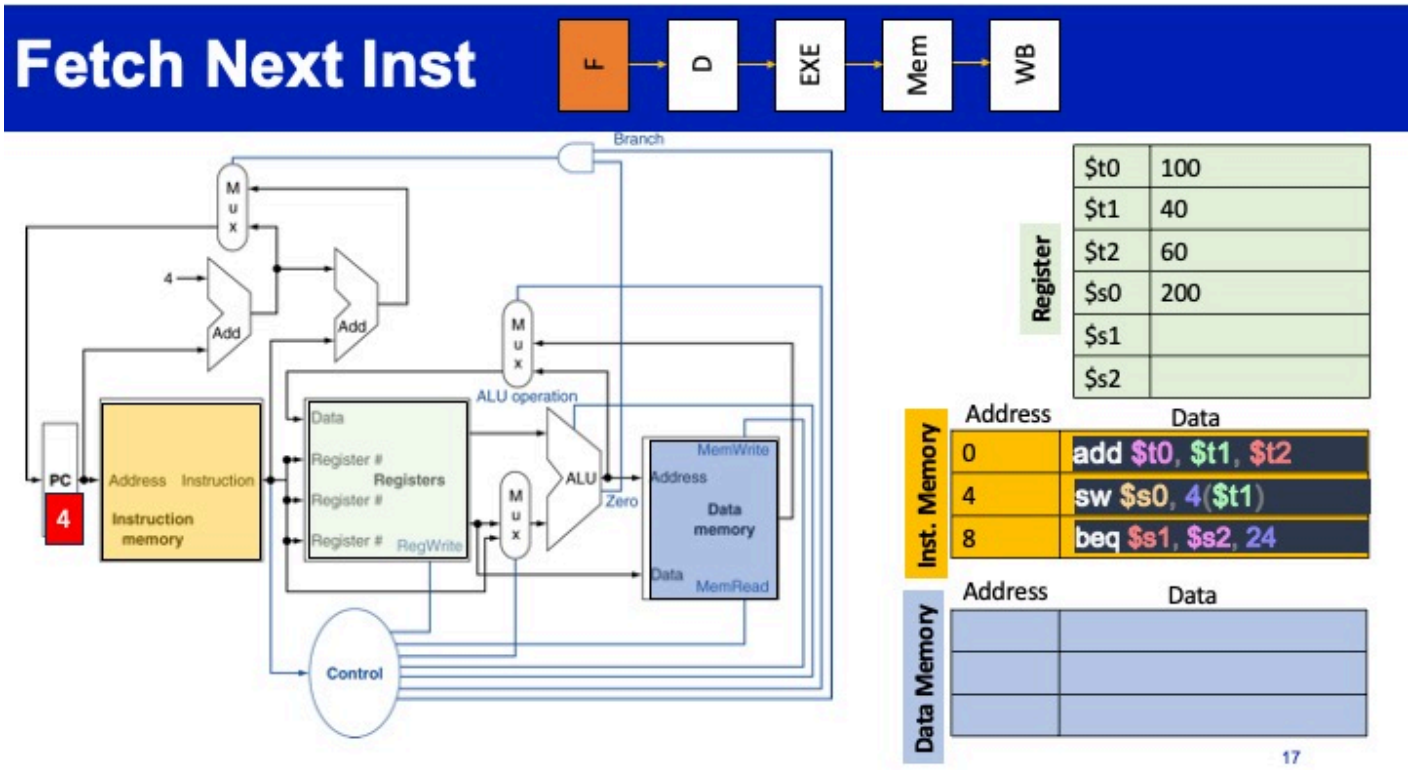
Register		
	$t0	100
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

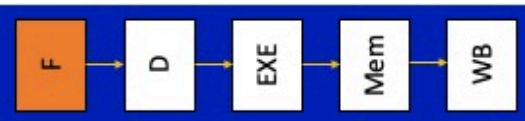
Data Memory	Address	Data

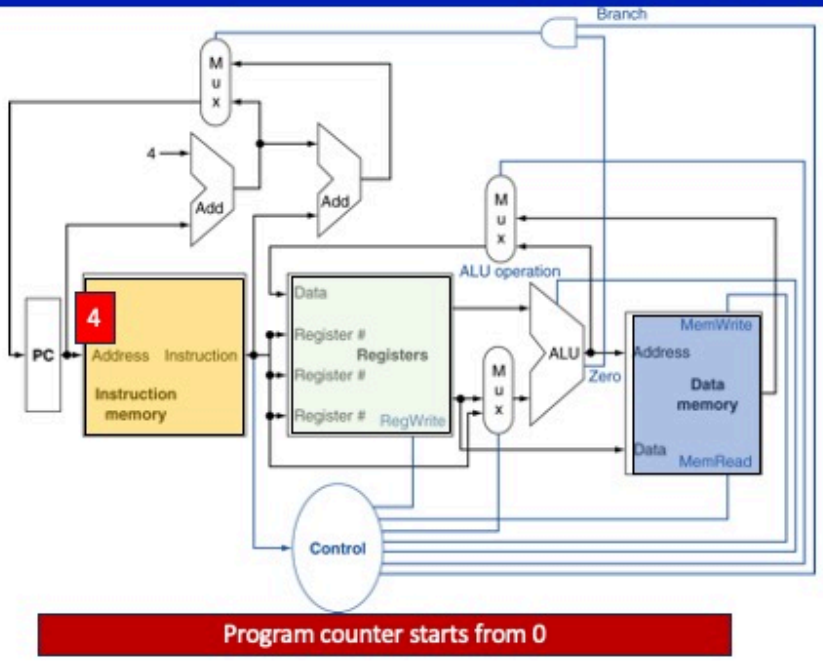
15


Anytime
F
D
EXE
Mem
WB
Branch
Not a branch
PC+4
M u x
4
Add
Add
ALU operation
PC
4
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
add
Control
Update PC to PC+4 before next fetch
Register
$t0 100
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
16


Fetch Next Inst
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
ALU operation
PC
4
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control
Register
$t0 100
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
17

Fetch:





Program counter starts from 0

Register		
	$t0	100
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

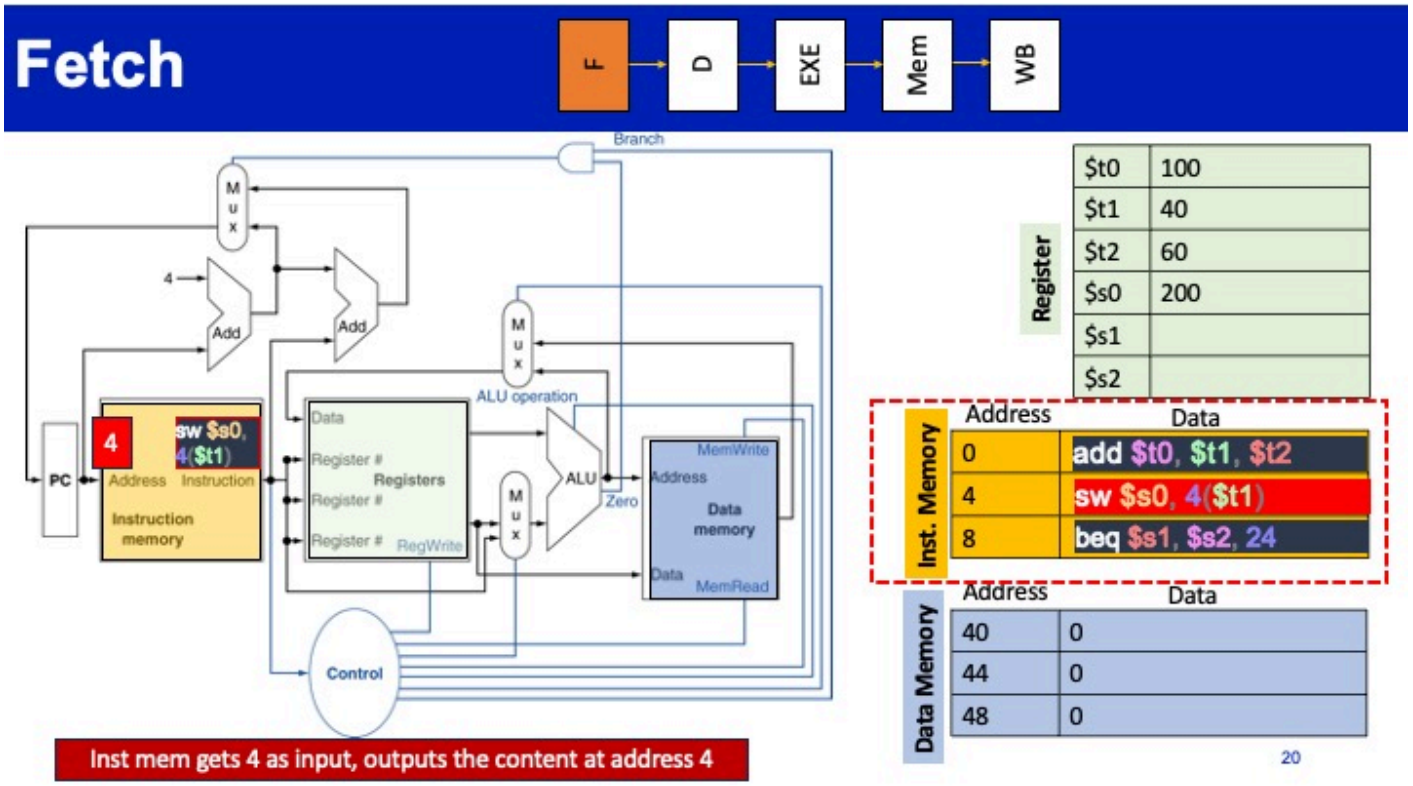
Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

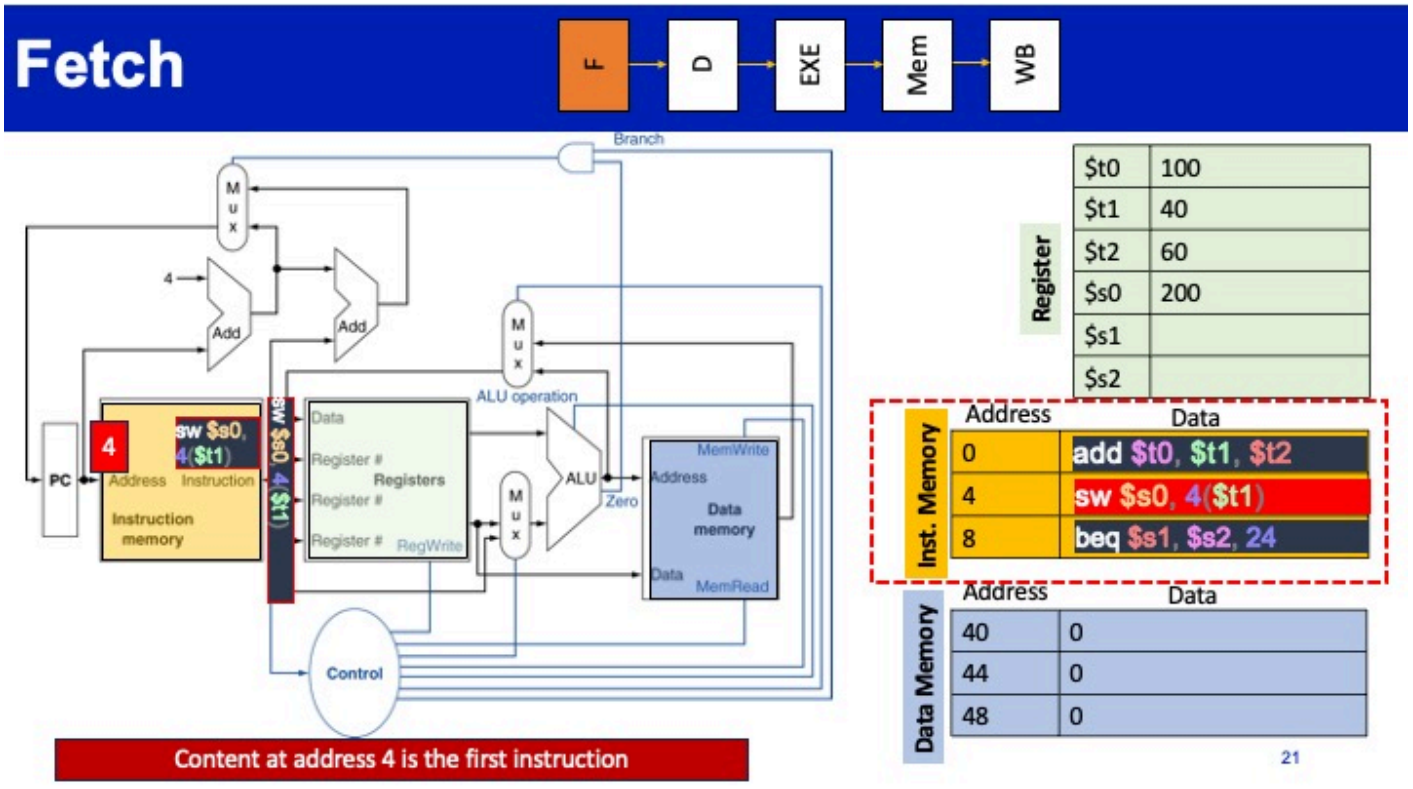
Data Memory	Address	Data
	40	0
	44	0
	48	0

18

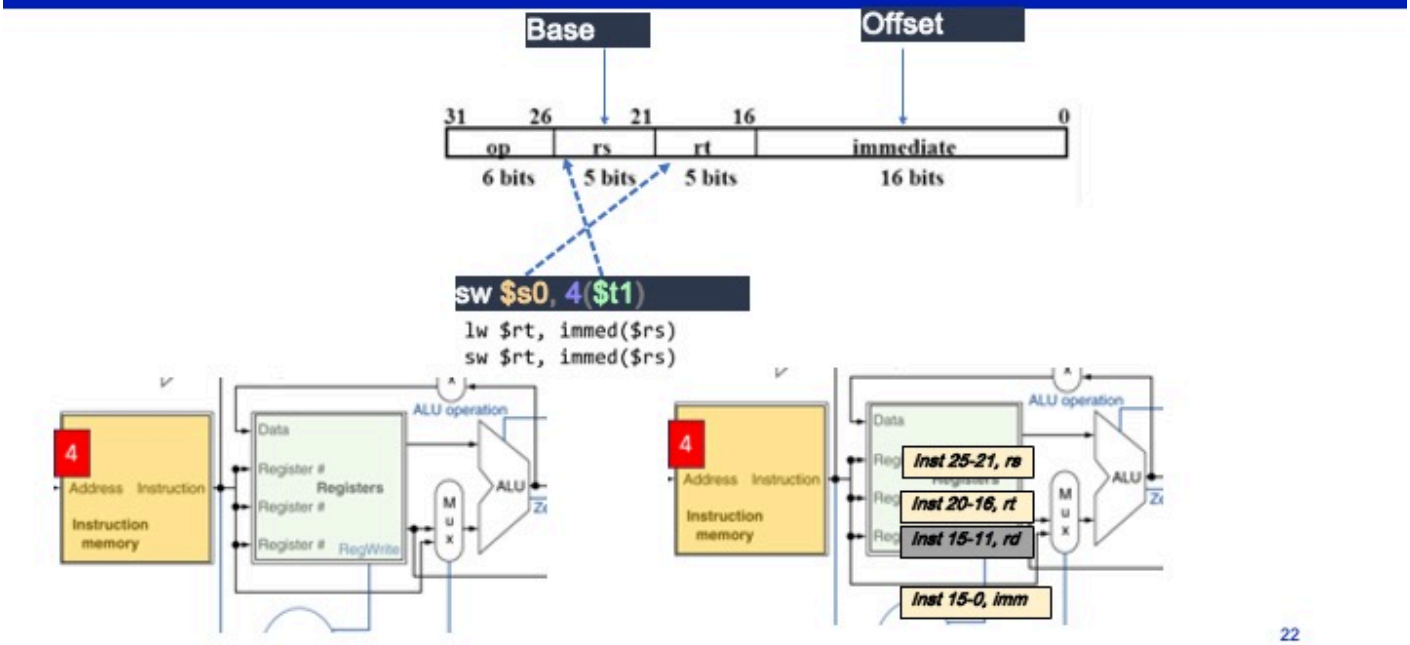
Execution process of Second Instruction

19


Fetch
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
M u x
ALU operation
PC
4
sw $s0, 4($t1)
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control
Register
$t0 100
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
Inst mem gets 4 as input, outputs the content at address 4
20

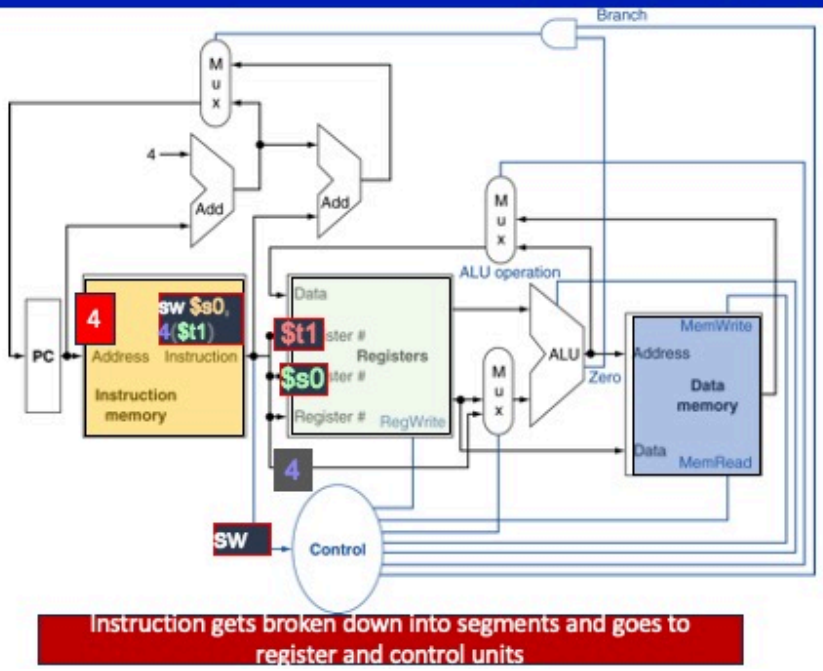

Fetch
F
D
EXE
Mem
WB
Branch
M u x
4
Add
Add
M u x
ALU operation
PC
4
sw $s0, 4($t1)
Address
Instruction
Instruction memory
sw $s0 4 $t1
Data
Register #
Registers
Register #
Register #
RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control
Register
$t0 100
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
Address
Data
40 0
44 0
48 0
Content at address 4 is the first instruction
21

Decode for SW/LW



22

Decode



Instruction gets broken down into segments and goes to register and control units

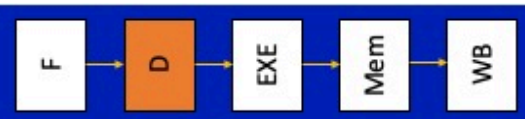
Register		
	$t0	100
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

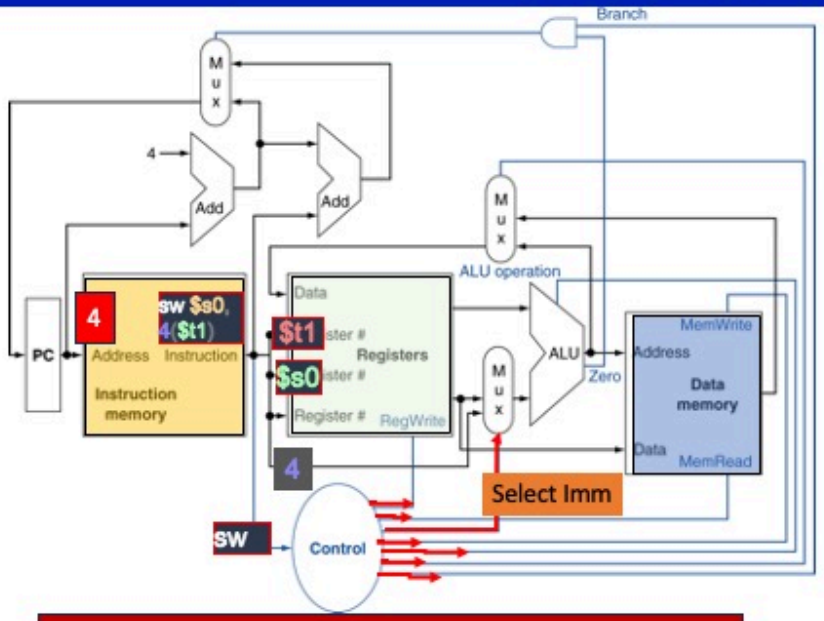
Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

Data Memory	Address	Data
	40	0
	44	0
	48	0

23

Decode





Control signals are asserted

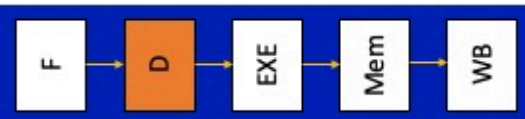
Register		
	$t0	100
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

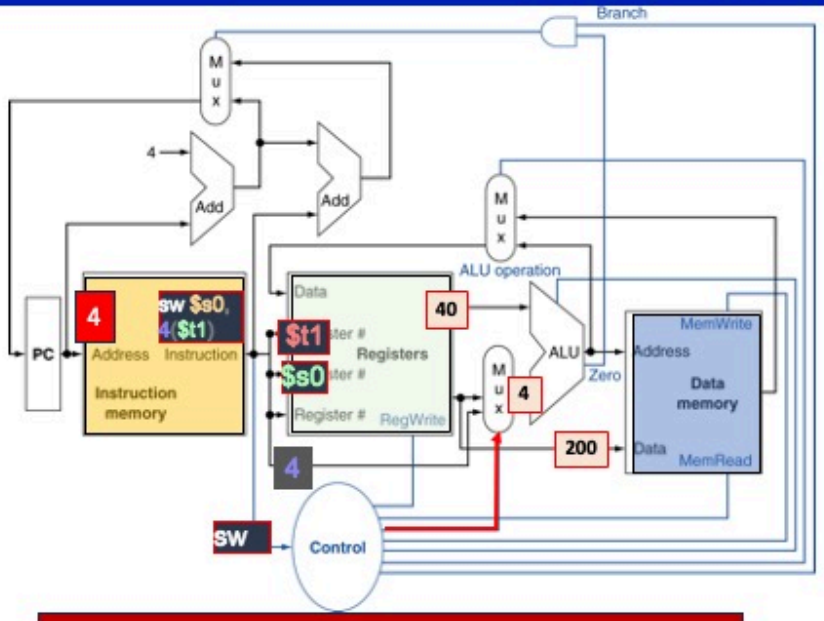
Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

Data Memory	Address	Data
	40	0
	44	0
	48	0

24

Decode





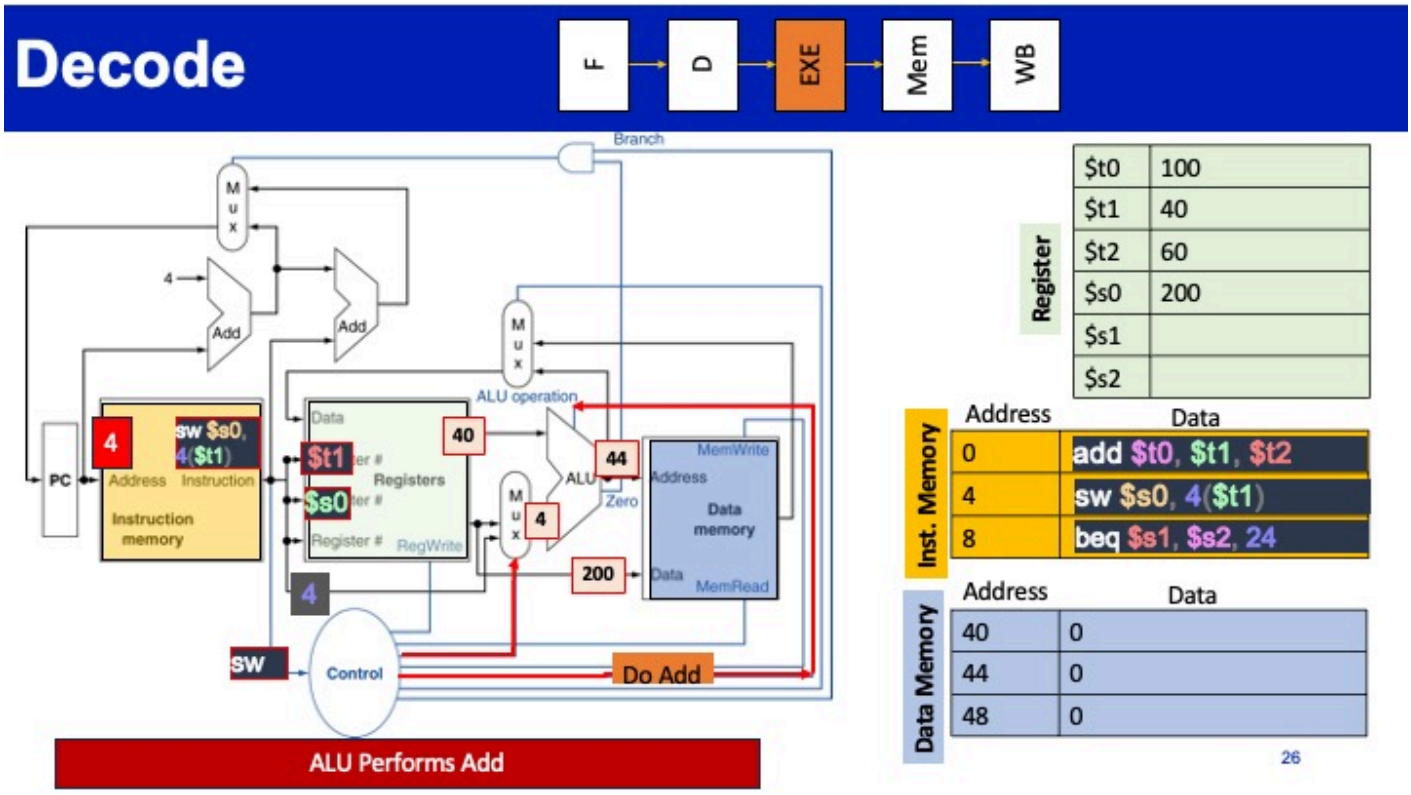
Register data comes out

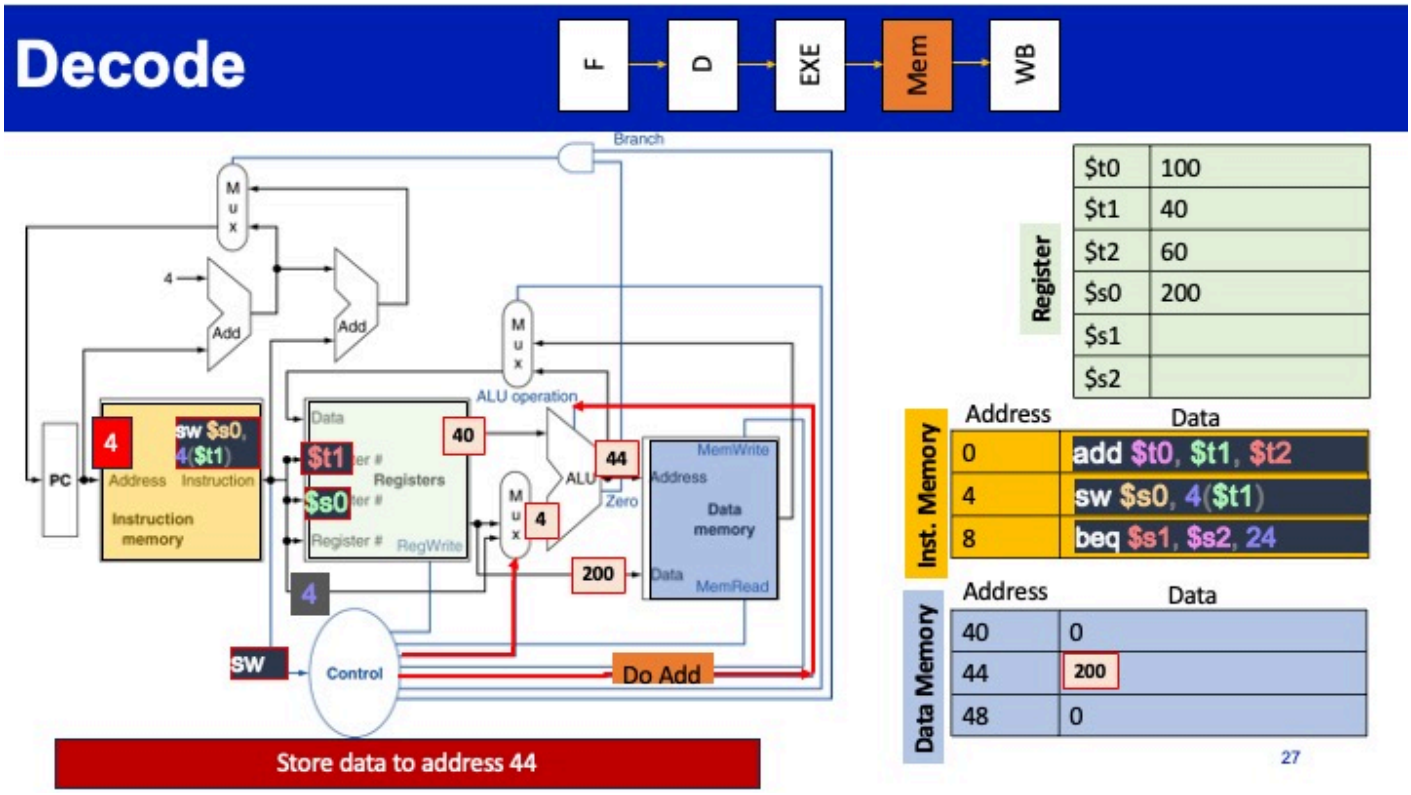
Register		
	$t0	100
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

Inst. Memory	Address	Data
	0	add $t0, $t1, $t2
	4	sw $s0, 4($t1)
	8	beq $s1, $s2, 24

Data Memory	Address	Data
	40	0
	44	0
	48	0

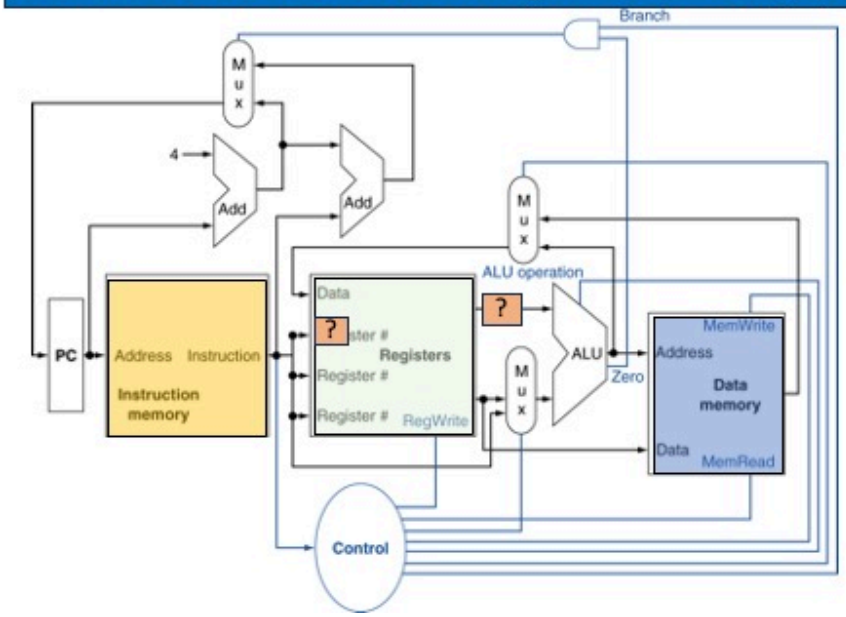
25


Decode
F
D
EXE
Mem
WB
Branch
M u x
Add
4
ALU operation
sw $s0, 4($t1)
Address
Instruction
Instruction memory
PC
Data
$t1
$s0
Registers
Register #
RegWrite
40
ALU
44
Zero
MemWrite
Data memory
MemRead
200
SW
Control
Do Add
ALU Performs Add
Register
$t0 100
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
40 0
44 0
48 0
26


Decode
F
D
EXE
Mem
WB
Branch
M u x
Add
4
ALU operation
sw $s0, 4($t1)
Address
Instruction
Instruction memory
PC
Data
$t1
$s0
Registers
Register #
RegWrite
40
ALU
44
Zero
MemWrite
Data memory
MemRead
200
SW
Control
Do Add
Store data to address 44
Register
$t0 100
$t1 40
$t2 60
$s0 200
$s1
$s2
Inst. Memory
Address
Data
0 add $t0, $t1, $t2
4 sw $s0, 4($t1)
8 beq $s1, $s2, 24
Data Memory
40 0
44 200
48 0
27

Example Question

Calculate the values in the box after all stages of instruction sw $s0, 4($t1) are done?

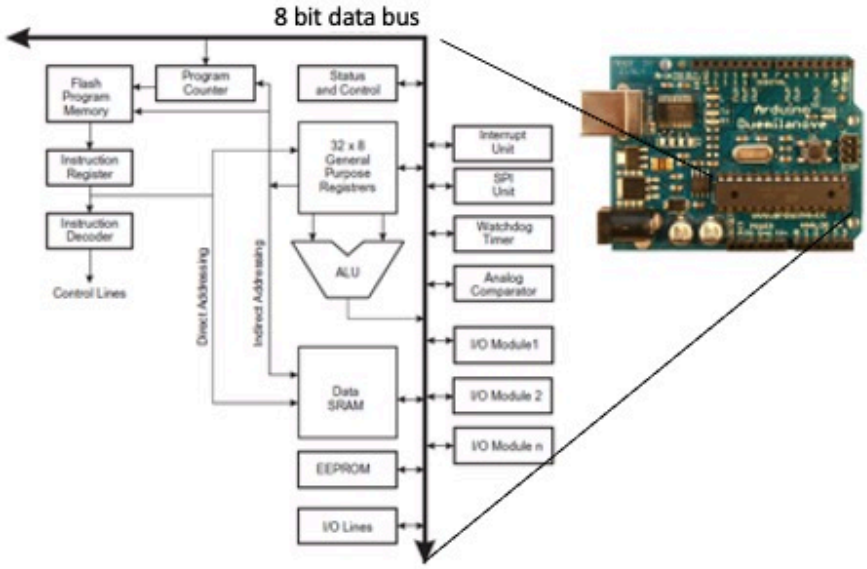


Register		
	$t0	100
	$t1	40
	$t2	60
	$s0	200
	$s1	
	$s2	

Data Memory	Address	Data
	40	0
	44	?
	48	0

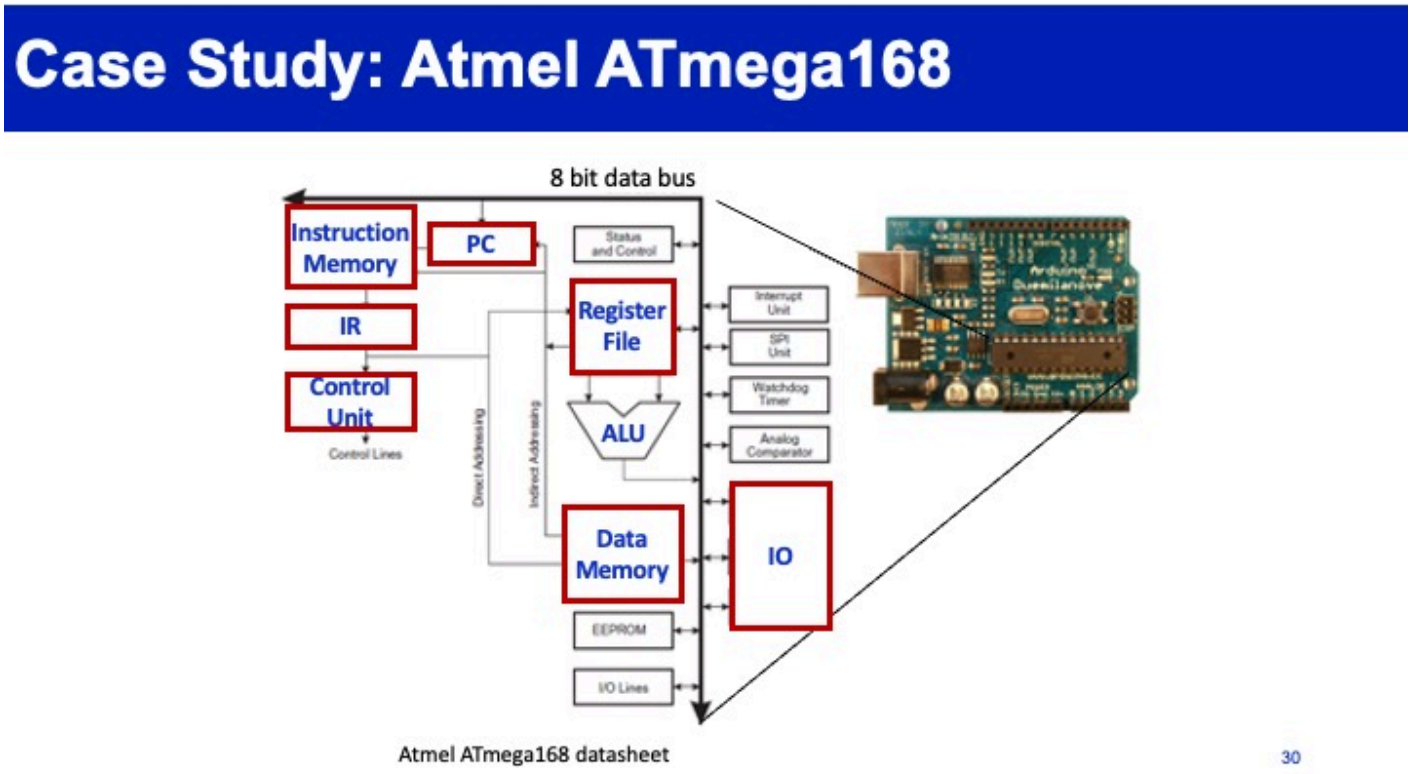
28

Case Study: Atmel ATmega168



Atmel ATmega168 datasheet

29



p

Tuesday Feburary 20th

Memory Technologies and Hierarchy

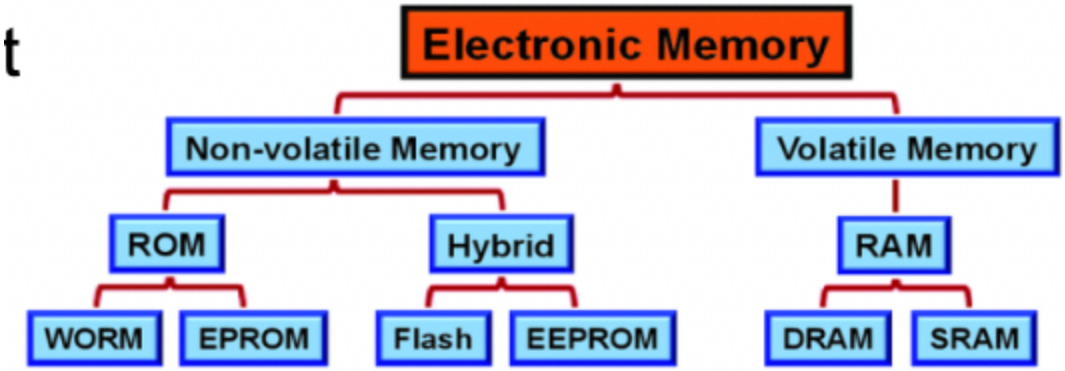
```
Store --> Memory
Load <--- Memory
```

Data Storage Technologies/Approaches

- Flip-Flops
  - Very Fast
  - Very Expensive - 46 transistors for a D flip Flop
- Static RAM (SRAM)
  - Fast
  - expensive - 6 transistors per bit

- Dynamic RAM (DRAM)
  - Slow, destructive read
  - Cheap - one transistor + one capacitor per bit
- Flash
  - Very slow, non-volant
  - cheap

t



- ---

Types of Memories

SSD: $0.75 /GB, HDD: $0.1-0.2/GB

DRAM: $20-25/GB

Type	Size	Latency	Cost/bit
Register	< 1KB	< 1ns	$$$$
On-chip SRAM	8KB-6MB	< 2ns	$$$
Off-chip SRAM	1Mb – 16Mb	< 10ns	$$
DRAM	64MB – 1TB	< 100ns	$
Disk (SSD, HDD)	40GB – 1PB	< 20ms	< $1/GB

Array Organization of Memories

- Components
  - A memory Array
  - Address decoder
  - I/O circutry
- M-bit data can be stored/loaded in/from at each unique N-bit address
  - Addressability: M bits
  - Address space: 2^N unique adresses

Memory Array Organization

- **Row Decoder**

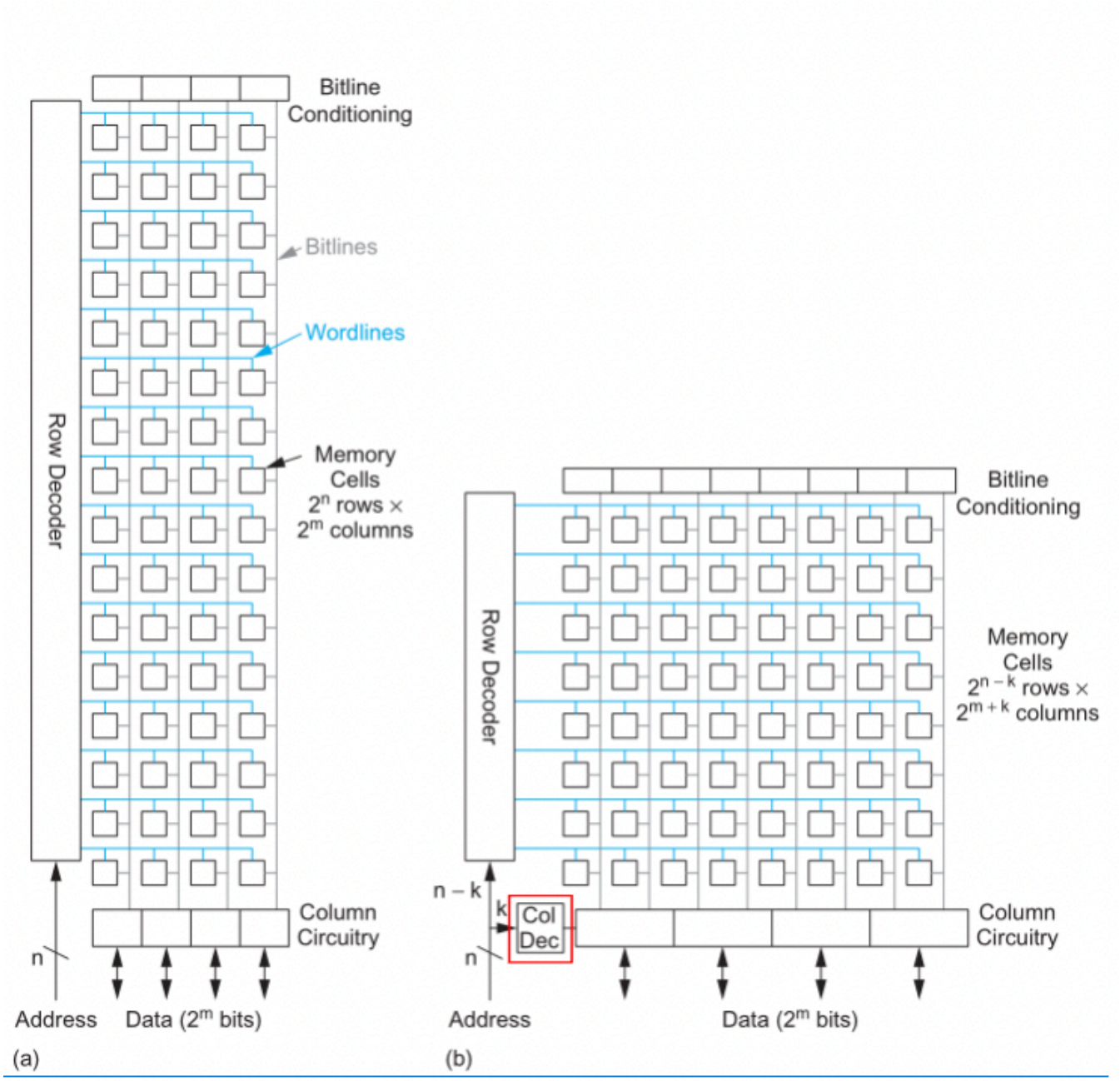
SSD: $0.75 /GB, HDD: $0.1-0.2/GB

DRAM: $20-25/GB

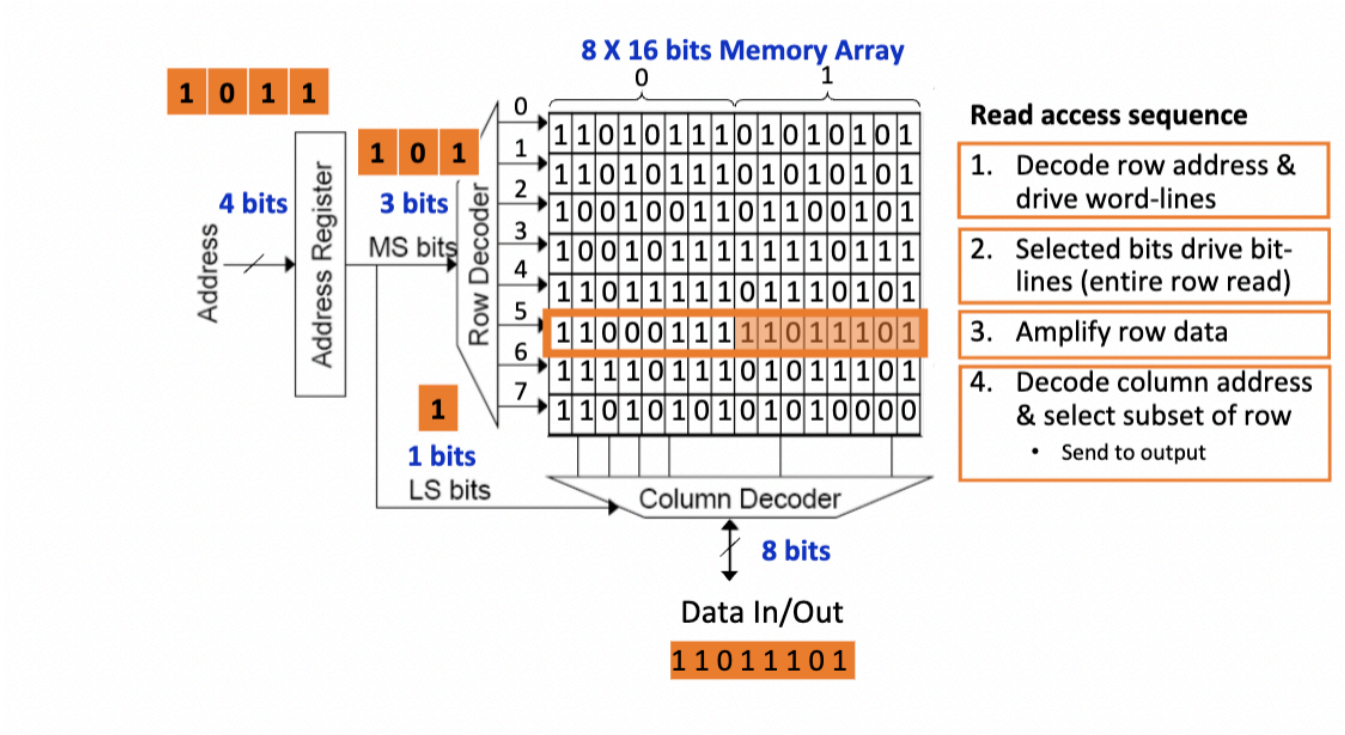
Type	Size	Latency	Cost/bit
Register	< 1KB	< 1ns	$$$$
On-chip SRAM	8KB-6MB	< 2ns	$$$
Off-chip SRAM	1Mb – 16Mb	< 10ns	$$
DRAM	64MB – 1TB	< 100ns	$
Disk (SSD, HDD)	40GB – 1PB	< 20ms	< $1/GB

Memory Array Folding

- The row decoder uses the address to activate one of the rows by asserting the wordline
- Array is often folded into fewer rows of more columns.Column decode is need to choose between multiple data lines


Bitline Conditioning
Bitlines
Wordlines
Row Decoder
Memory Cells
2^n rows × 2^m columns
Column Circuitry
n
Address
Data (2^m bits)
(a)
Bitline Conditioning
Row Decoder
Memory Cells
2^{n-k} rows × 2^{m+k} columns
n – k
k
Col Dec
n
Column Circuitry
Address
Data (2^m bits)
(b)

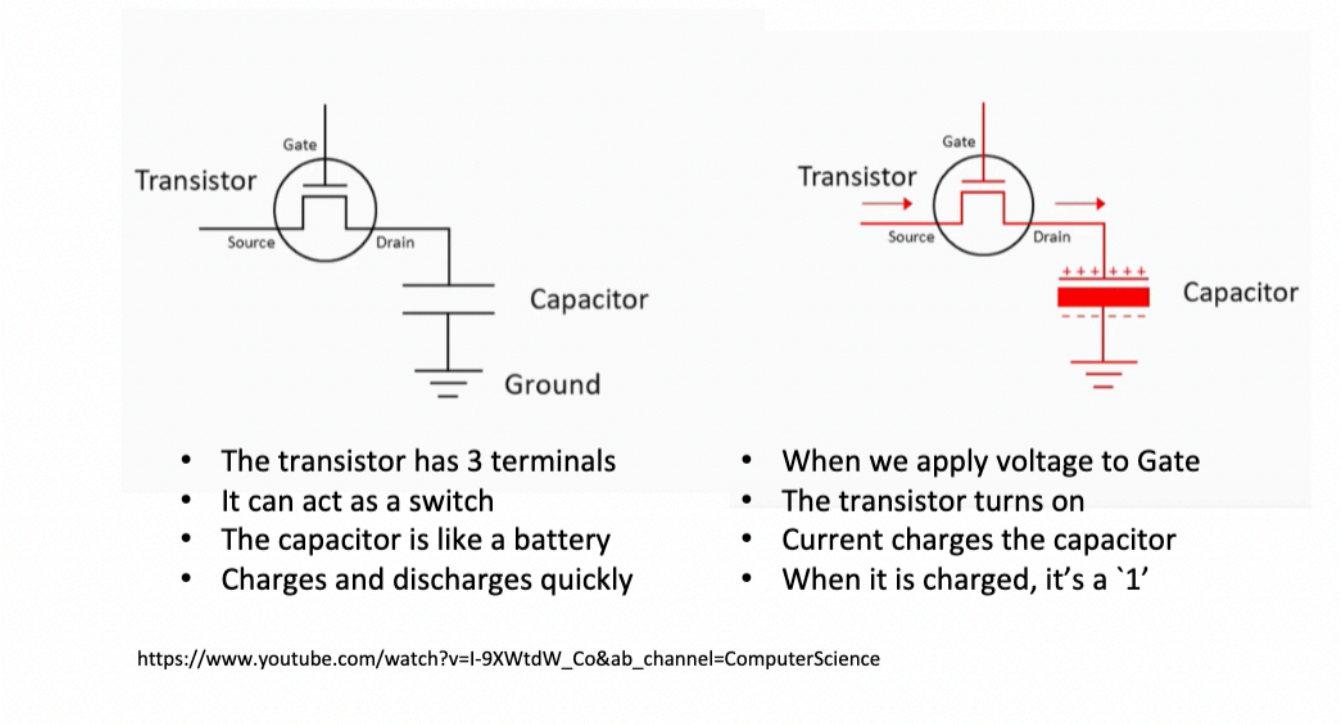
Memory Read Example



DRAM Memory Technology

- Dynamic random-access memory
- Capacitor cahrge state indicates tored value
  - charged 1
  - discharded 0
  - 1 capacitor
  - 1 access transistor
- Capacitor Leaks
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed

Workign of the memory cell

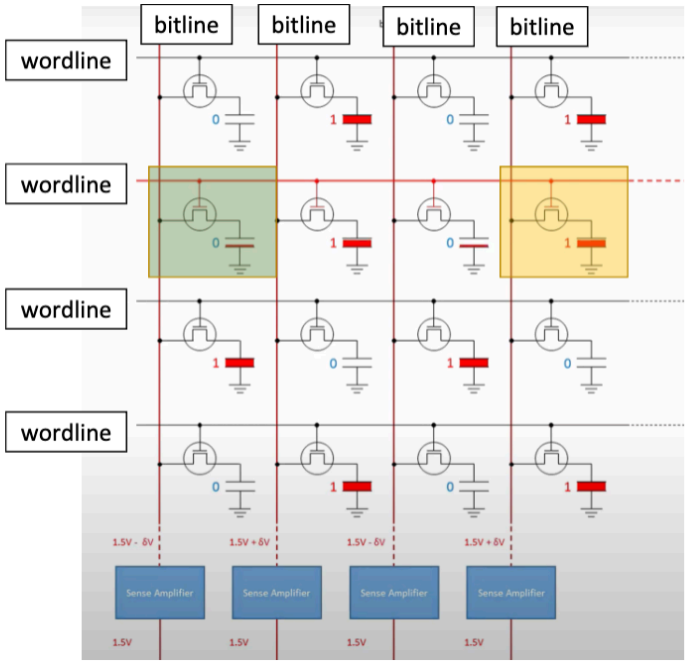


Read DRAM

- Bitlines are precharged
- Precharge voltage→ vdd/2
  - All bitlines in a row
- Apply Vdd to wordline
  - Only 2nd wordline

- If capacitor is already charged
  - Charge will move out from cap
  - Bitline voltage will increase
  - Sense Amp will detect '1'

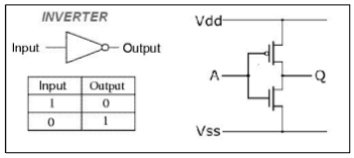
- If capacitor is empty
  - Charge will move into Cap
  - Bitline voltage will reduce
  - Sense Amp will detect '0'

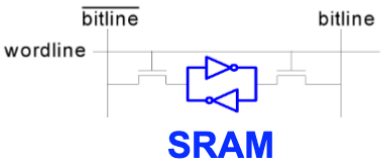


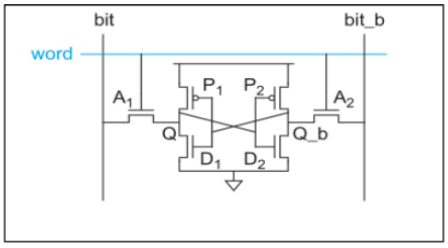
- Read process was desctructive
- It charged/discharged some Cpa
- we must write back after read
  - The correct data is in SA
- Write form SA to impacted row

SRAM Memory Technology

- Static random access memory
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the “cell”
  - 4 transistors for storage
  - 2 transistors for access







19

Thursday Feburary 22nd

Memory Hierarchy and Address Space

MROM/EPROM/EEPROM

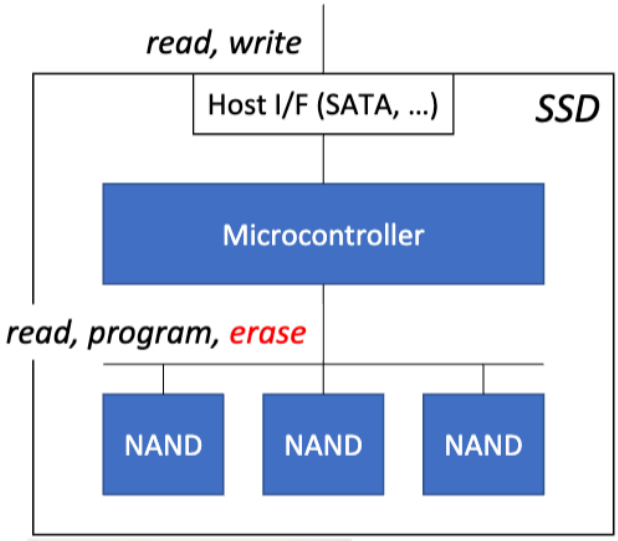
- Mask ROM (MROM)
  - Non programmable read only memory
  - its cheap
- EPROM
  - Erase by using UV light
  - Erasable programmable read-only memory
- EEPROM
  - Electrically erasable programmable read-only memory
  - Can be reprogrammed many times 8 Can be erased at byte granularity
- All are often used t ostore small read-only code
  - i.e. Boot loader

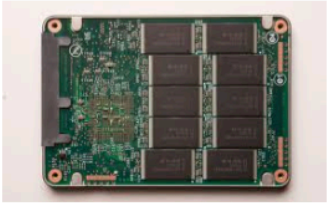
Flash Memory

```
Two types of flash memory
```

- NOR Flash (e.g. S70GL02GT Cypress device)
  - ReadReads at byte granularity
  - Fast random reads (~120ns)
  - slow writes (~520ms to erase 120KB)
  - Lesser density
  - Primarily used as code storage
- NAND Flash (S34ML04G2 Cypress device)
  - Page addressable (e.g., 512 Bytes)
  - Slow random reads (~30us)
  - fast writes (~3.5ms to erase 120KB)
  - Higher density
  - Primarily used as data storage 4

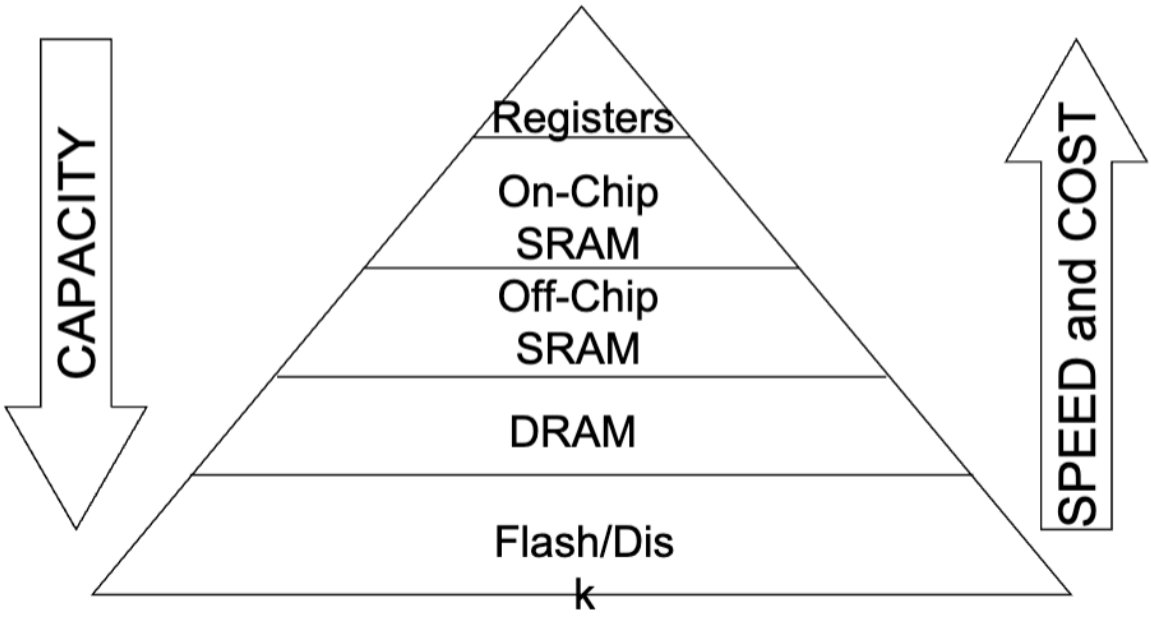
Solid-State Disk (SSD) / eMMC





- Flash Translation Layer (FTL)
  - S/W running on the controller
  - Provides disk abstraction
- No seek time
  - No mechanical part
- High capacity
  - SSD: ~1000s GB
  - eMMC: ~10s GB
- High bandwidth
  - SSD: ~1000s MB/s
  - eMMC: ~100s MB/s

Memory Hierarchy



-

Analogy

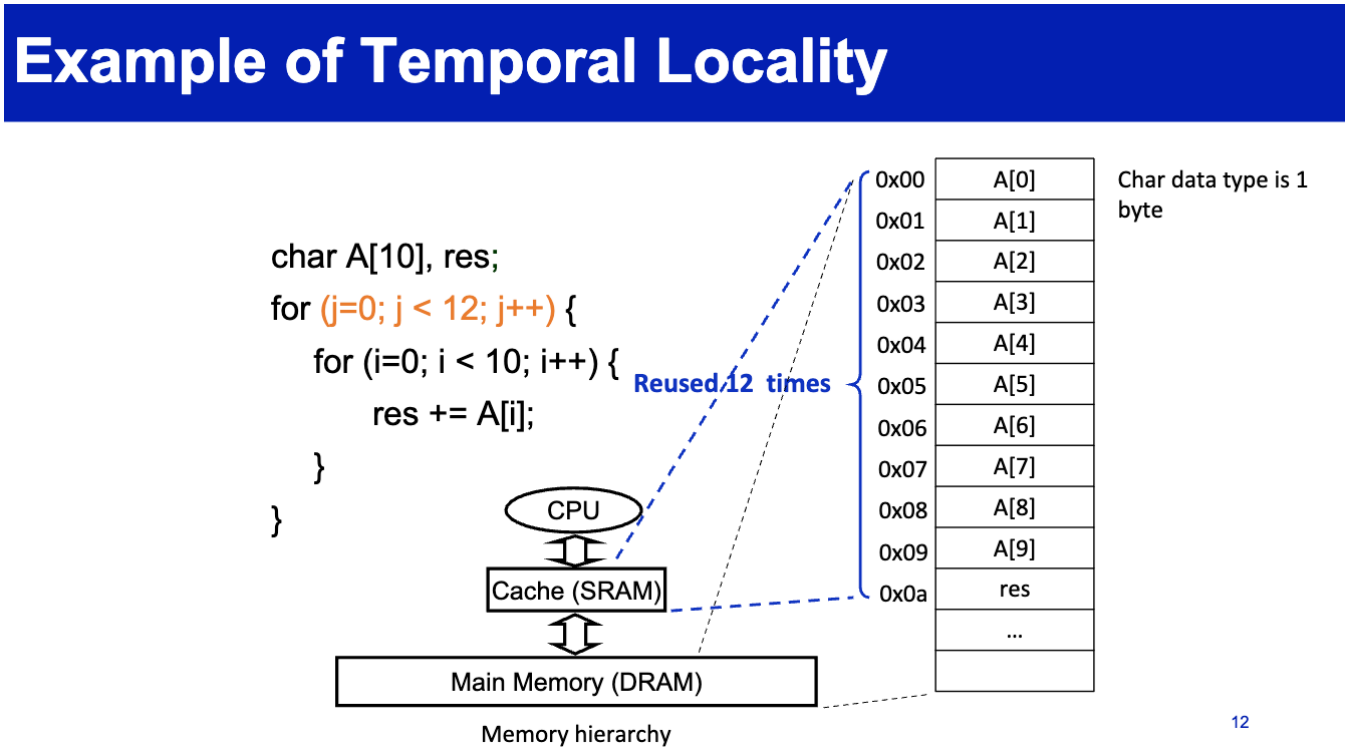
- the memory hierarch is similar to a library
- this is used to spped up and keep costs low
  - hands = Register 1 book
  - Table = SRAM Up to 4 books
  - Book Trolly = DRAM Up to 8 books
  - Bookshelf = SSD ALL Books

Why to implement the memory subsystem hierarchy?

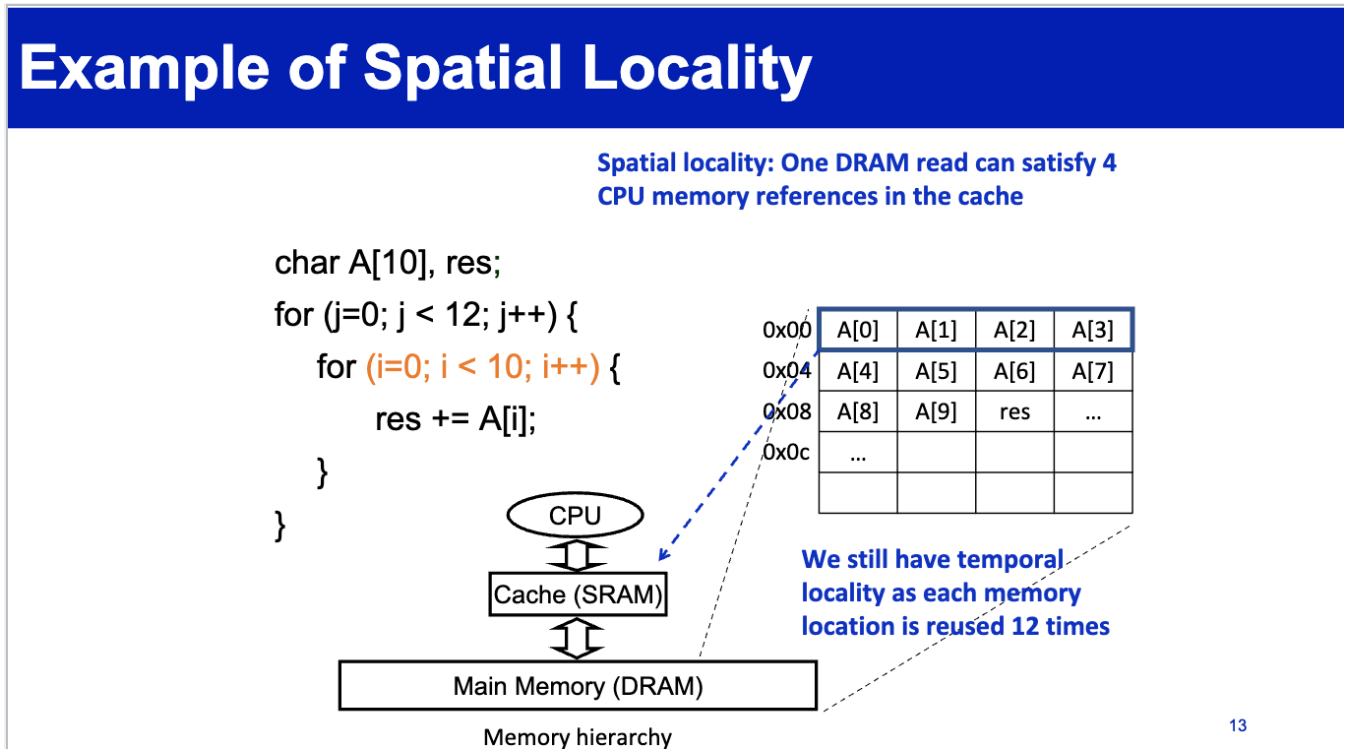
```
Locality of Reference
```

- Temporarl locality
  - reference same meory location many times (close together, in time)
- Spatial Locality
  - Reference near neighbors around the same time

Temporal Locality



Spatial Locality



EXAMPLE QUESTION

Do we need to have a memory hierarchy if CPU is slower than our slowest memory technology? E.g., if we have a CPU with clock frequency of 1 HZ and a non-volatile memory with access latency of 0.1 sec. Explain a little bit

Answer:

No we do not need memory hieracrhcy because the CPU

Program Address Space

Memory Layout of C Programs

```
Typical memory representaion of a C program consists of thre broad regions
```

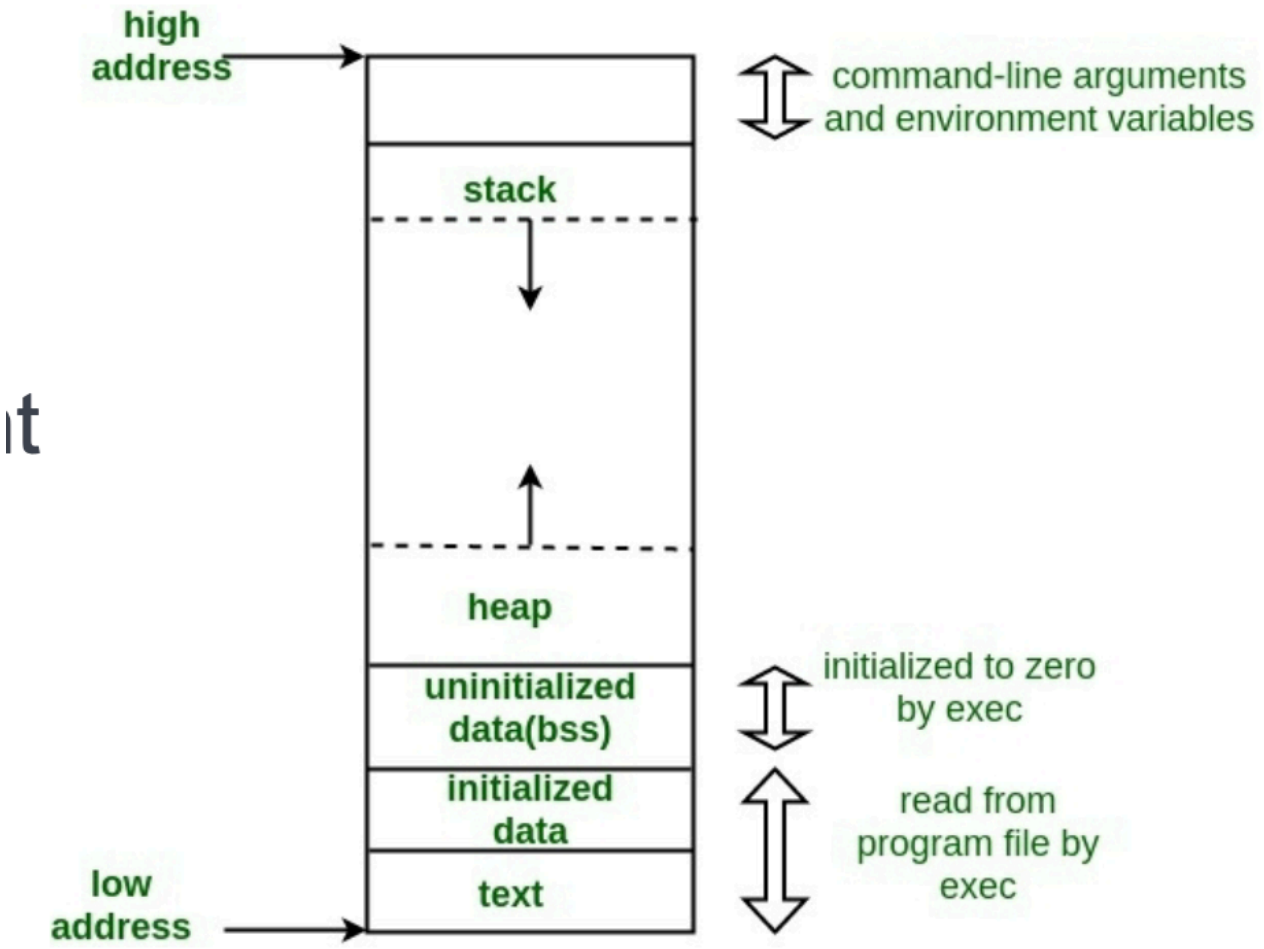
1. Static Data
    - fixed size, read only, exists for entier program
    - text/code segment
    - intialized data segment
    - unitialized data segment

Text segment

- containes executable instructions placed below heap or stack in order to prevent overflows
- Often read only

2. Stack
    - variable size
    - grows when alling function
3. Heap

- variable size grows when allocated in program



-heap/

Untialized Data Segment

8 contains all global cariables and satci variables that are intialized to zero or do not have explicit intialization source code

Static Vars

- Static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

```c
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

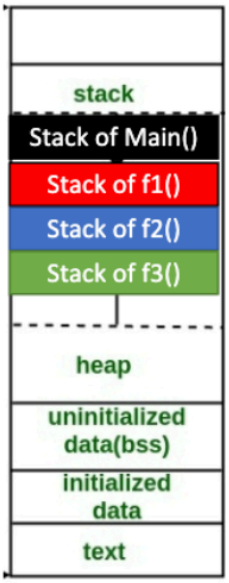
Output: 1, 2

```c
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output: 1, 1

Stack Example

this example the bottom of the stack is on the top because that is where the word is




```
int main(){
        //…
        f1();
        return 0;
}

int f1(){
        //…
        f2();
}

Void f2(){
        //…
        f3();
}

Void f3(){
        //…
}
```

Sample entry sequence

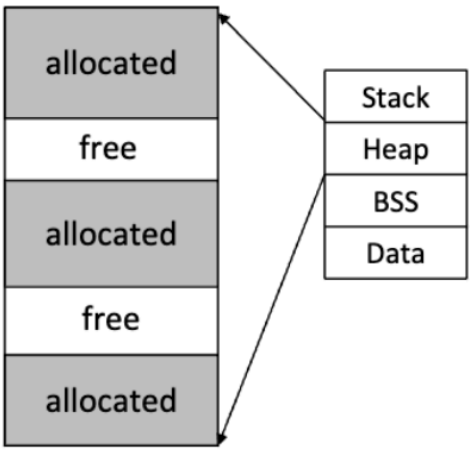
```
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
```

Corresponding Exit sequenc

```
lw ra, 0(sp)
lw a0, 4(sp)
addi sp, sp, 8
```

Heap Example

- segment where dynamic memory allocation usually takes place.
- Reserved at compile time
- Allocated/freed at runtime
  - malloc()
  - free()




```
// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));
```

Not recommended to use for critical embedded applications (e.g., automotive)

Data Memory (Data Segment)

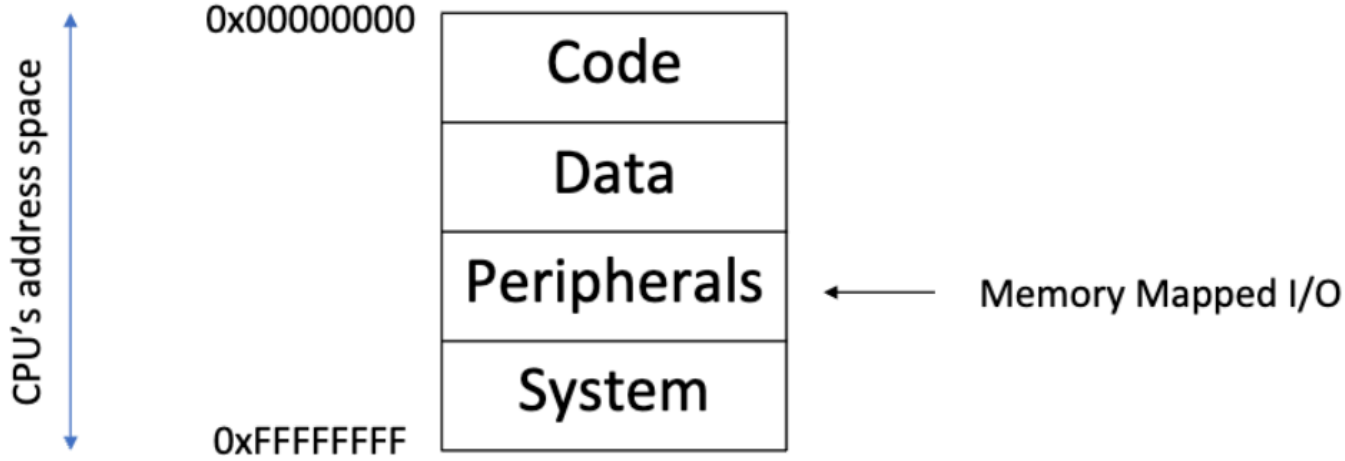
- **Stack**: temporary data like local variables
- **Heap**: dynamically allocated data
  - `malloc` (similar to `new` in C++)
- **Data**: non-zero initialized global and static data
- **BSS** (Block Started by Symbol): zero initialized and uninitialized global and static data

```
B int globA;
D int globB = 1;
  int main () {
S   int varA;
S   int varB = 10;
B   static int varC = 0;
D   static int varE = 1;
S   char *varD;
H   varD = (char*)malloc(8);
    varA = varB + varC;
    return varA;
  }
```

Memory Map

`CPU's view ofthe physical hardware`

- Segregated with multiple regions
- diff memory type for each region
- each platform may have diff mappings



**Tuesday March 5th **

Lectuer 11 I/O Interface Part 1 (NOT ON MIDTERM 1)

Interfaces

- Two types of interfaces
- **Serial**
  - USB, RS-232 VGA cable, SPI, SATA connections
  - Pros: few pins and wires, high scalability, low power consumption
  - cons: lower bandwidth for the same clock speed
- **Parallel**
  - Parallel ATA (advanced tech attachemnet), SCSI (small computer system inteface), PCI (perihperal component interface (this connects ram and graphics card to the board)
  - Pros: higher bandwith
  - Cons : more pins and wires, low scalability, high power consumption

Transfer Types

- transferType.png

Synchronous vs. Asynchronous

Synchronous Transmission

```
TX ---Data----> RX
TX ---Clock----> RX
```

- **synchronous transmission**
  - requires ocmmon shared clock
  - higher throughput communicaiton
  - low scalability
  - parallel interfaces are usually sync

Asynchronous Transmission

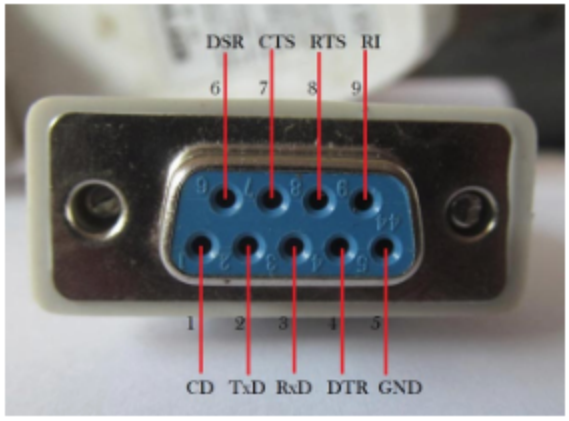
```
TX ---Data----> RX
```

- No shared clokc
- Asynchronous start/stop
- Self clocekd, based on transmitted and reciever

Serial Communication Standard: RS-232

- A 1 is represented by –3 to –25 V
- A 0 is represented by +3 to +25 V
  - Low voltage denotes 1
  - large voltage range makes it less susceptible to noise, interferenace and degredation

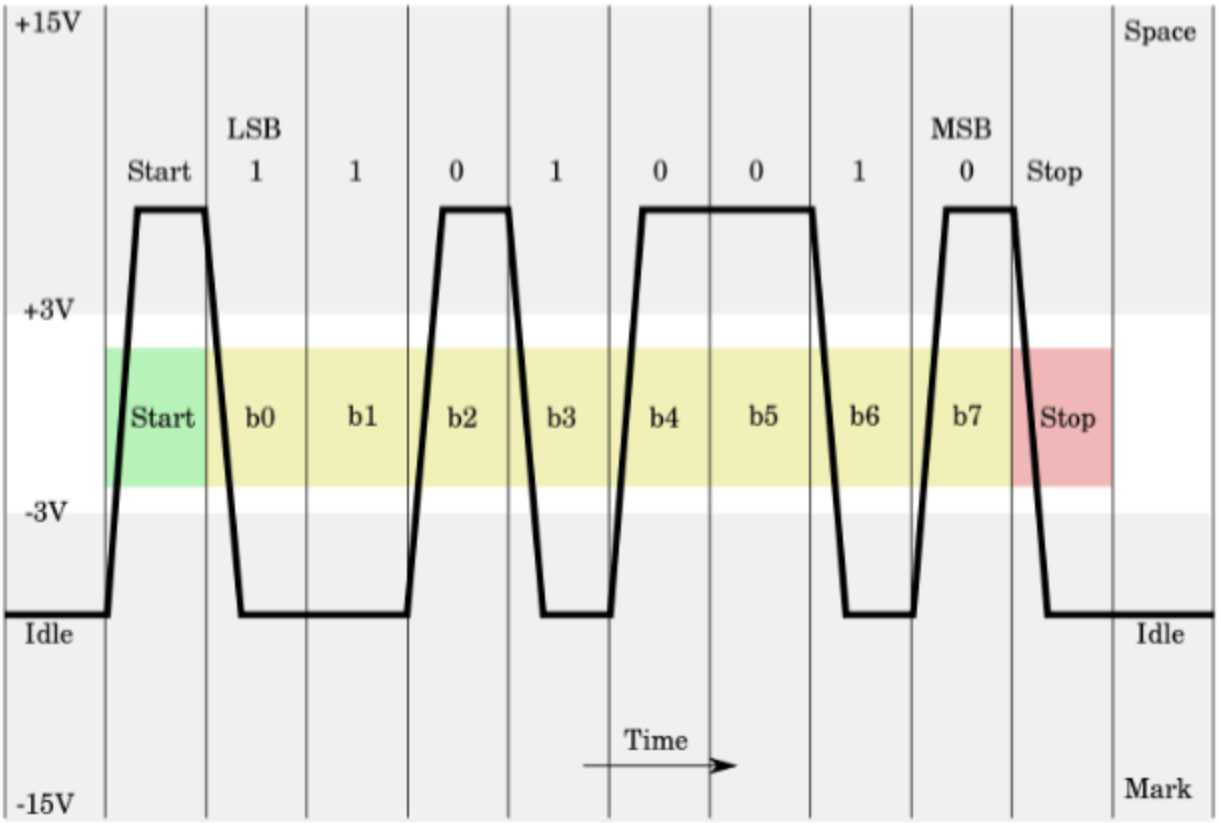
source



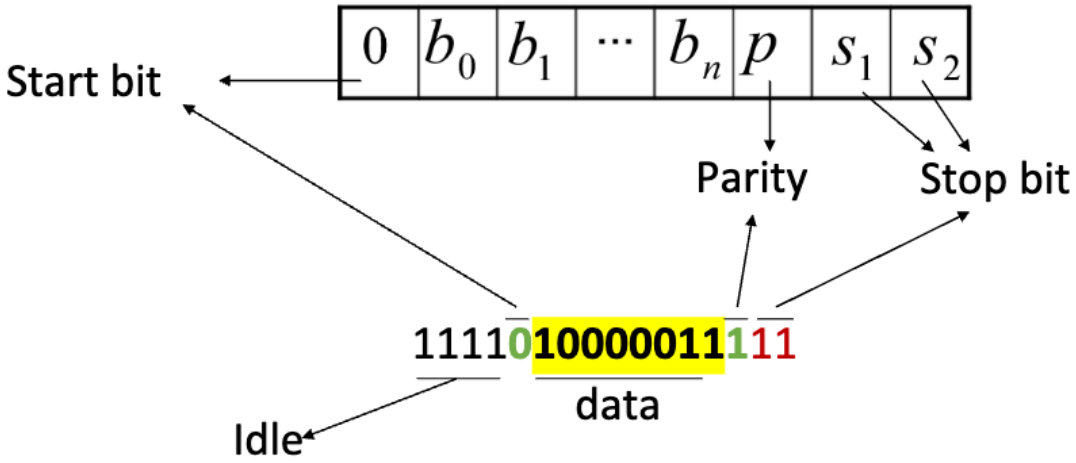
-

RS232

- The idle state of the RS232 lines is logic 1 (-12V)
- To signal a start condition the line is set logic 0 (+12V) for 1 bit period (please note high voltage means 0).
- Cause a 1 to 0 transition indicates valid data is coming 11



RS232 Frame Format



Parity Bit

- used for deteing errors in bits
- Two types: ‘even’ , ‘odd’

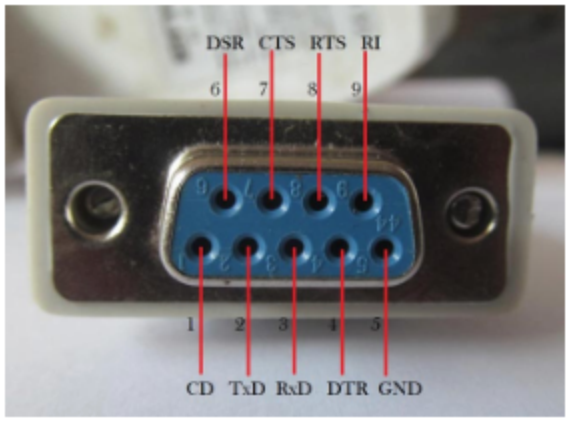
- Even Parity: if total number of 1's are even the parity bit value is set to 0. for odd parity its opposite

7 bits of data	(count of 1-bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010

-
- the number of ones has to be either even or odd than the last bit will be 1 for even adn 0 for odd if they don't match then there is an error
- the first 0 indicates the start bit then the next 8-bits forms the transmitting frame

RS-232 Pins

source



-

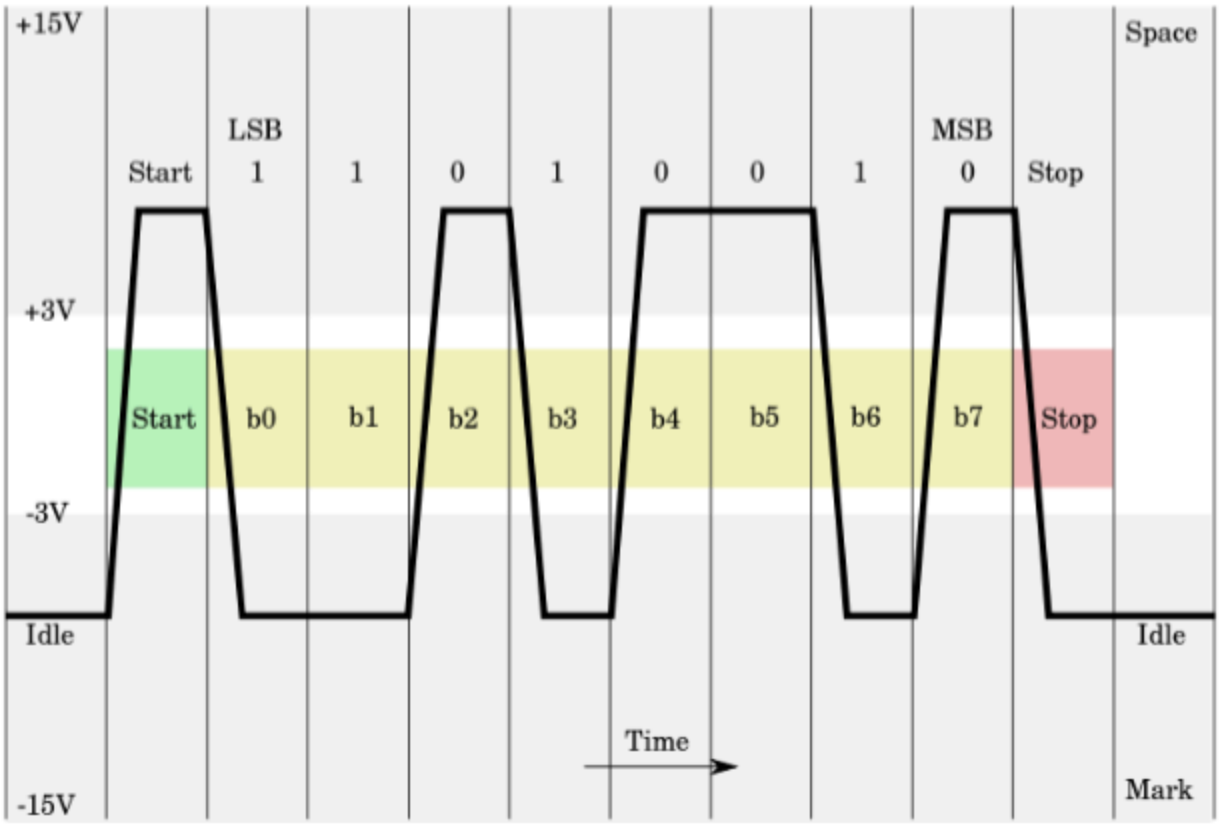
Universial Asynchronous Reciever/Transmitter (UART)

- Convert parallel content of an 8-bit register t oa bit sequence ready to be transmitted ofver a serioal port
- processor pins -> UART -> volt conversion -> RS232

System bus diagram in the slides (slide 17)

UART Speed (Baudrate)

```
Both sender adn reciever must use agreed upon transmisison speed (baudrate)
```



- For a baud rate of 2400 (2400 bps) the frequency is 2400Hz and the bit period is 1/2400 or 416.6us.
- This is the information that a receiver uses to recover the bits from the data stream.

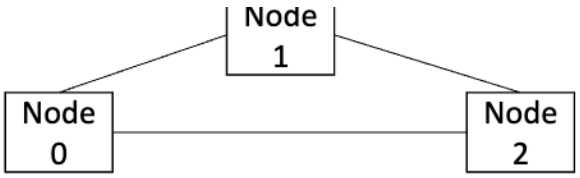
Question

- Suppose you are sending data over a UART channel at a baud rate of 115200 bps. How long does it take to send a single 8-bit character over the channel?
- Assume 2 stop bit, 1 parity bit, and 1 start bit 19
- ANSWER : 1/115200bps * (8 + 2 + 1 + 1)

Type of Network

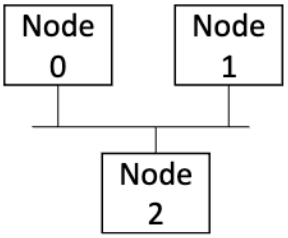
Point-to-point

- 1:1 communication
- No sharing of connection



Bus

- Shared among multiple devices
- Need an arbitration mechanism
- Master
  - An entity who initiates the data transfer
- Slave
  - An entity who cooperates with the master

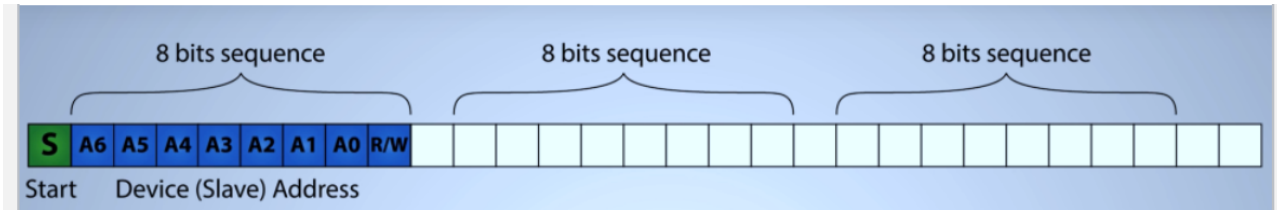


- 20

Example of Bus based communication I2C Bus

- 2C is serial and synchronous communication protocol.
- Synchronous -> Uses clock for syncing the data
- Serial -> data line

I2C Bus Data Protocol

- Data is transferred in sequences of 8 bit
  - bgeings with a start bit
- After start bit, the master provides address of r/w bit
  - It's the address of the slave device
  - R/W denotes if the master wants to read or write
  - 

- After address and R/W bits are sent, Slave returns and ACK bit (acknowledgement)
- If the ACK is 0 slave could not receive the data

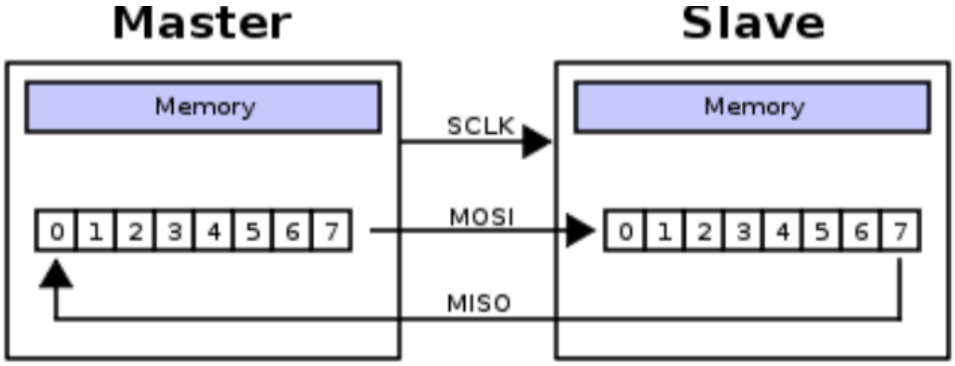
- If the Master wants to point to an internal register of the slave device, a second address can be sent
- **start and stop singal**
  - high-to-low SCL is hgh defines a START condition
  - low-to-hgih SCL is high defines STOP

SPI

```
Serial Peripheral Interface
Synchronous, serial communication protocol
```

- Uses 4 lines, full-duplex, over 10Mbps
- Single master, multi-slave
- No start/stop bits
- Good for fast, short distance communication, e.g., connecting

SPI Protocol

- Master shifts out to MOSI (Master Out Slave In) and shifts in from MISO (Master In Slave Out)
- Slave shifts in from MOSI and shifts out to MISO
- 

https://en.wikipedia.org/wiki/Serial_Peripheral_Interface 29