

Computer Organization

HARI T.S. NARAYANAN

Logistics

- You should complete this course with reasonable design experience and programming experience
- No preferred text book
- You can use all listed books for reference
- Your interaction will make this course fruitful.
- Notes taking is discouraged, pay attention and use handouts
- No phones and no recording
- I will not respond to mail – Talk to me in the class

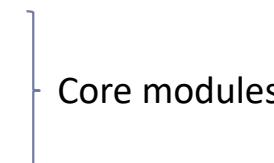
Logistics Continued

- Theory:
 - Quiz 1: 25% (1 Hour, 25 short questions)
 - Quiz 2: 25% (1 Hour, 25 short questions)
 - Final : 50% (2 Hours, 50 short questions)
- Lab:
 - Regular Lab: 30%
 - Final: 50%
 - Final Quiz: 20% (from Theory and Lab)

85-100	S
75-84	A
65-74	B
55-64	C
45-54	D
35-44	E
	F

90-100	S
80-89	A
70-79	B
60-69	C
50-59	D
40-49	E
	F

Content

1. SAP – A simple hypothetical Processor
 2. ARM ISA – First few labs – treat this as my extra classes
 3. Some Introductory material – Data representation
 4. Building blocks of Arithmetic and Logic Unit
 5. Processor Design - Logic, RTL, and Verilog
 6. Superscalar Processor
 7. Memory Hierarchy and Virtual Memory
 8. I/O: Polling, Interrupt, Interrupt Handler, (taught in Lab); DMA, Disk, RAID
 9. Multiprocessor System – Cache Coherence
 10. I/O - USB – likely be skipped
- 
- Core modules

Tools

- *Tcl** - only if you are not taught File I/O
- ARM Emulator (Assembler)
- Logisim (Logic Design)
- Icarus (HDL Verilog Compiler)
- Gtkwave (Wavefile content presenter)

Assumption

- Windows 10, anything else, you are on your own!
- Completed
 - Electronic Devices
 - Electronic Circuits
 - C with File I/O, C-struct, Library (.lib and dll), Memory Allocation, Stack, Debugging, Process life Cycle, Compiler, Linker, and Loader

Pre-Requisite

- Logic Gates
- Data Representation in memory
- Study Assignment 1:
 - Brush up Binary to Decimal, Decimal to Binary conversions
 - Various Representation of Decimal in Binary: Binary encoding, BCD, Signed/Unsigned representations, Floating Point Representation IEEE 754, Octal, Hexadecimal, 1s Complement, 2s Complement, ASCII form,
- Study Assignment 2:
 - Logic Gates and their truth tables
 - Simple logic design using gates
 - DeMorgan's Law of Logic

Recap of some pre-requisites

- Interested in **Decimal System, Binary System, Hexadecimal System**
 - What is the decimal equivalent of **0b110111** (binary)?
 - What is the decimal equivalent of **0X1101**
 - What is the decimal equivalent of **0xFF** (hexadecimal)?
 - What is the binary, hexadecimal equivalent of **255**?

Decimal to Binary & Binary to Decimal

- Decimal to Binary: Repeated division by 2
- Binary to Decimal: Using Positional Weight with 2 as radix
 - ... $2^3, 2^2, 2^1, 2^0 \cdot 2^{-1}, 2^{-2}, 2^{-3}...$
- Exercise
 - Convert 254 to binary
 - Convert 1000 0000 to decimal

Binary Coded Decimal (BCD)

- Each decimal digit represented by 4 bit binary
- No positional weight here
- Exercise:
 1. Represent 255 in BCD
 2. Convert your answer into weighted decimal form
 3. Is your answer same as 255 or is it different?

ASCII & UNICODE representation

- 8-bit coding of numbers, alphabets, and special characters.
- Exercise
 - Represent 255 in ASCII or Unicode
 - Find the Unicode and ASCII representations for INR?*

Use **MS Word Symbols** to find the ASCII or Unicode encodings

Important and Relevant Values

- Number of possible values that you could represent with
 - 2 bits
 - 4 bits
 - 8 bits
 - 10 bits
 - 20 bits
- Is 1 K same as 1000? (1 K is 2^{10} , 1000 is 10^3)
- Similarly Mega and Million differ – handle them with care

Register and Memory

- Registers are storage elements that are faster than cache memory
- Here is an 8-bit Register:

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 LSb
- Memory is represented as follows

$mx4B$

B3	B2	B1	B0		Word Address	Byte Address
					0	0
					1	4
					2	8
					3	12
					.	.
				
					m	4xm

Some Useful Patterns

- Encode all integers (including 0) < 256 in 8 bits
- Encode multiples of 2 (including 0) < 256 in 8 bits
- Encode multiples of 4 (including 0) < 256 in 8 bits
- Decimal Value of: 1000, 1000 0000, 1111, 1111 1111

Important and Relevant Values

- Minimum number of bits required to represent
 - 2 different values
 - 4 different values
 - 8 different values
 - 16 different values
 - 256 different values
- How many bits are needed to represent all the values from 0 to 256?

Maximum Decimal Value

- Largest decimal value that you could represent with
 - 16-bit register
 - 32-bit register
- Represent the following decimals in 8-bit binary: 0, 4, 8, 12, 16, 20, ...
 - What can you say about the least 2 significant bits?

Representing Values with Decimal Point

- Binary notation offers two different representations for numbers with decimal point:
 - Fixed point representation
 - Floating point representations – Two standards
 - 32-bit IEEE 754 standard
 - 64-bit IEEE 754 standard

Fixed & Floating Point Decimal

- Write the following decimal numbers in the following in the 4 slots on the right (decimal point position is fixed)

- 45.0

- | | | | | |
|---|---|---|---|---|
| 4 | 5 | • | 0 | 0 |
|---|---|---|---|---|

- .35

- 955.0

- 0.825

Mantissa

E

- Now write these numbers in the following format:

- | | | | | |
|---|---|---|---|---|
| 4 | • | 5 | 0 | 1 |
|---|---|---|---|---|

Normalized Floating Point Representation

- Writing numbers on a piece of paper is flexible.
- In computer, the number of slots (register size) is fixed
- If we don't use floating notation, we will lose the resolution
- Also, it is important that we standardize this floating point representation

32-Bit Floating Point Representation



To be represented in this format, a number should be in the following normalized form.

$$(+ \text{ or } -) 1.\text{(mantissa)} \times 2^{\text{(exponent)}}$$

⇒ This means

$$\begin{aligned} \Rightarrow 4.6 / 2 &= 2.3 \\ \Rightarrow 2.3 / 2 &= 1.15 \end{aligned}$$

⇒ Hence we get the normalized form and we can write
 $+4.6 \Leftrightarrow 1.15 \times 2^2$

Exponent is not 2!!

It is 129

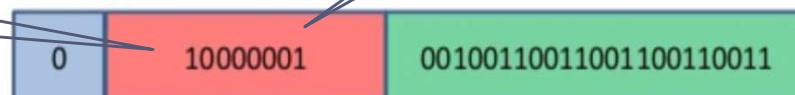
Let us convert the 0.15 to binary

$$\begin{array}{rcl} \Rightarrow 0.15 \times 2 = 0.3 & - & 0 \\ \Rightarrow 0.3 \times 2 = 0.6 & - & 0 \\ \Rightarrow 0.6 \times 2 = 1.2 & - & 1 \quad (\text{i}) \\ \Rightarrow 0.2 \times 2 = 0.4 & - & 0 \\ \Rightarrow 0.4 \times 2 = 0.8 & - & 0 \\ \Rightarrow 0.8 \times 2 = 1.6 & - & 1 \quad (\text{ii}) \end{array}$$

⇒ Now the value from (i) till (ii) will continue to recur and we will keep recurring it till 23 bits are filled.

⇒ Thus the bits obtained are 00100110011001100110011

Hence the bit pattern in the 32 bit format are



$$\Leftrightarrow (4093333)_16$$

Exponent offset is
127

Example

You need to do just the reverse of the above which is very simple.
For example:

Given Binary representation: 11000001101111110.....0

Thus we will break it into three parts as:



We clearly see that the number is negative and the power is $131 - 127 = 4$

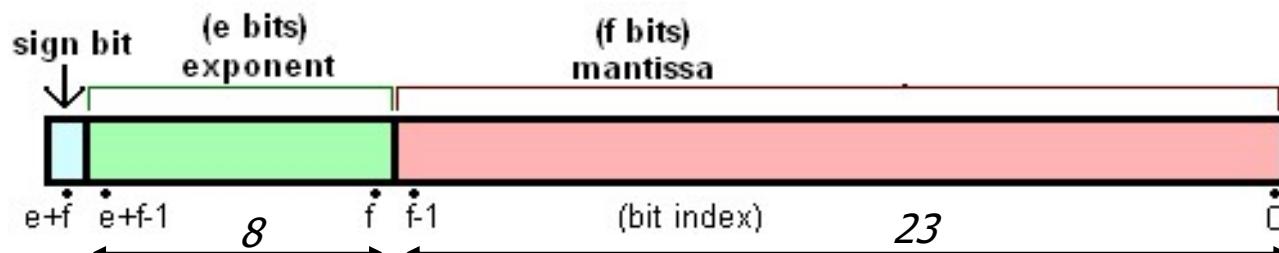
Mantissa is: $2^{-1} \times 0 + 2^{-2} \times 1 + 2^{-3} \times 1 + 2^{-4} \times 1 + 2^{-5} \times 1 + 2^{-6} \times 1 + 2^{-7} \times 1 = 0.4921875$

⇒ The number is -1.4921875×2^4 [note the '1' is added before the 0 in the normal form]

⇒ Which is equal to **-23.875**

ANS: -23.875

IEEE 754 Floating Point Representation



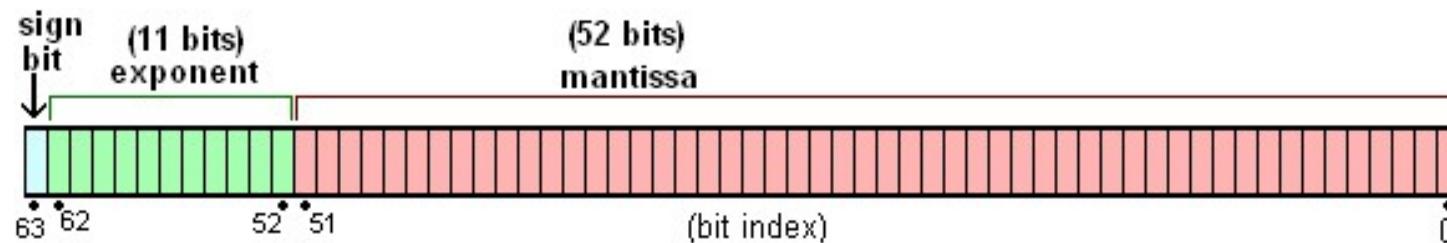
Type	Exponent	Mantissa
Zeros	0	0
Denormalized Numbers	0	non-zero
Normalized Numbers	1 to $2^8 - 2$	any
Infinities	$2^8 - 1$	0
NanS	$2^8 - 1$	non-zero

- Smallest Positive Number
- Largest Positive Number
- Resolution – steps
- **Smallest Negative Number**
- **Largest Negative Number**

Highest and Lowest Values with IEEE 754

- Exponent with sign gives us large range
 - 2^{-126} to 2^{127}
- Mantissa decides the resolution/precision
 - 1.0×2^{-126} to $1.000000\dots01 \times 2^{-126}$
 - 1.0×2^{127} to $1.000000\dots01 \times 2^{127}$

64-bit floating point representation



Double Precision Floating Point Representation

Arithmetic with Floating Point Representation

- Before adding or subtracting make sure the exponents are equal for both operands.
- If not, shift the decimal point of one of the operands to make it equal to the other one

- Example: $0.1 \times 2^4 + 0.01 \times 2^2$ is 0.1001×2^4

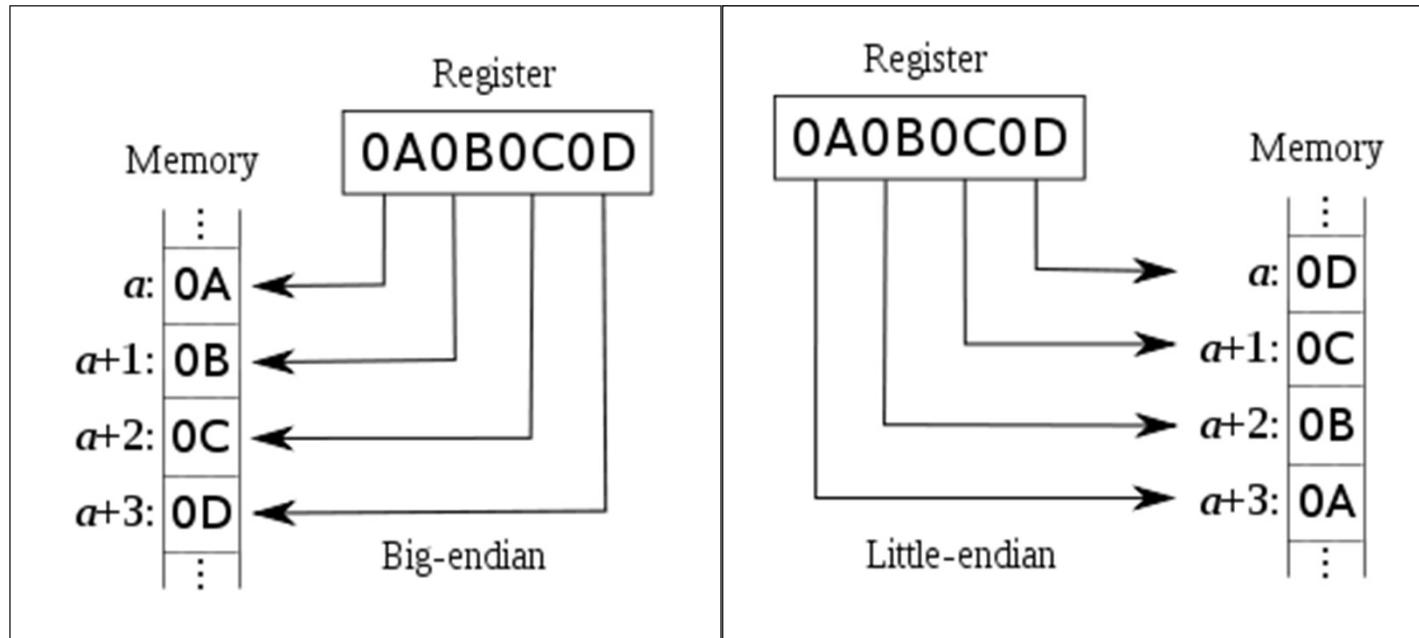
- That is, result is sum of 0.1×2^4 and 0.0001×2^4

} Not IEEE 754

Storing Data in Memory

- Data can be stored in memory from left to right or right to left.
- This is formalized using the following representations:
 - Big-Endian (Network Byte Order)
 - Small(Little)-Endian
- A processor sticks to one. Occasionally a processor might support both representations

Big-endian and Little-endian



Source: <http://en.wikipedia.org/wiki/Endianness>

Exercise: Find out the Endianness of your Windows Laptop?

Alignment

- Alignment normally fixed at
 - 2 Byte boundary
 - 4 Byte boundary
- Floating Alignment
 - Optimizes the storage
 - Makes the CPU logic more complex & slow

Logical Gates

- Logical gates are the basic building blocks of IC (CPU, memory, ...)

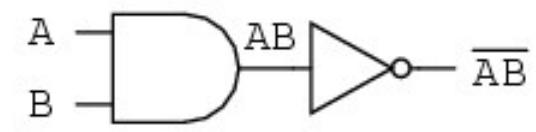
Name	NOT	AND	NAND	OR	NOR	XOR	XNOR																																																																																																
Alg. Expr.	\bar{A}	AB	\overline{AB}	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$																																																																																																
Symbol																																																																																																							
Truth Table	<table border="1"> <tr> <th>A</th> <th>X</th> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </table>	A	X	0	1	1	0	<table border="1"> <tr> <th>B</th> <th>A</th> <th>X</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table border="1"> <tr> <th>B</th> <th>A</th> <th>X</th> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	B	A	X	0	0	1	0	1	1	1	0	1	1	1	0	<table border="1"> <tr> <th>B</th> <th>A</th> <th>X</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	0	<table border="1"> <tr> <th>B</th> <th>A</th> <th>X</th> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	0	<table border="1"> <tr> <th>B</th> <th>A</th> <th>X</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	1	<table border="1"> <tr> <th>B</th> <th>A</th> <th>X</th> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Logical/Boolean Expressions

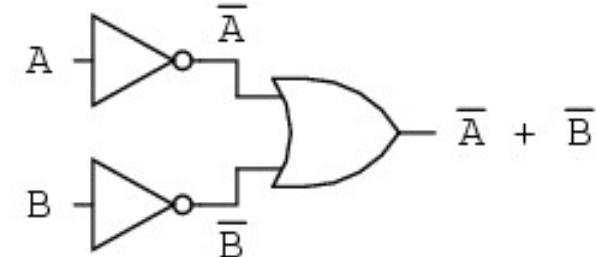
- If A and B are binary/Boolean variables then the following simple Boolean expressions
 - $A \& B$ or AB Read as A AND B
 - $A | B$ or $A + B$ Read as A OR B
 - $!A$ Read as NOT of A; used instead of a line on top
- A more complex expression
 - $(A \& B \& C) + (!A \& !B \& C)$

Logical Expression & Logical (Gates) Diagram

- Logical expressions are functional equivalent of logical diagram
- Logical expressions are optimized for number of gates and delay before their h/w realization (gate representation, ...)
- One of the key concept in this optimization is DeMorgan's law of logical expression:



... is equivalent to ...



$$\overline{AB} = \overline{A} + \overline{B}$$

<https://www.allaboutcircuits.com/textbook/digital/chpt-7/demorgans-theorems/>

De-Morgan's Law - Equivalence

$$\square !(A \text{ or } B) = !A \text{ and } !B \quad (1)$$

$$\square !(A \text{ and } B) = !A \text{ or } !B \quad (2)$$

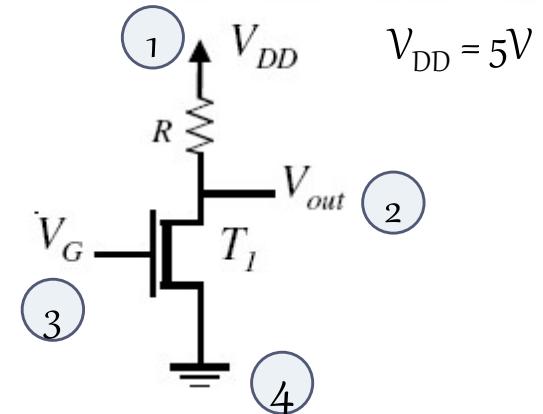
Transistors

- Semi-conductor
- Logical Gates are built with transistors, resistors, & voltage source
- In a Transistor, the output voltage (V_{out}) is V_{DD} (5V) when the input voltage (V_G) is 0V and 0V when the input voltage is V_{DD} – An Inverter!
- It is easy to combine transistors to realize logical functions AND, OR

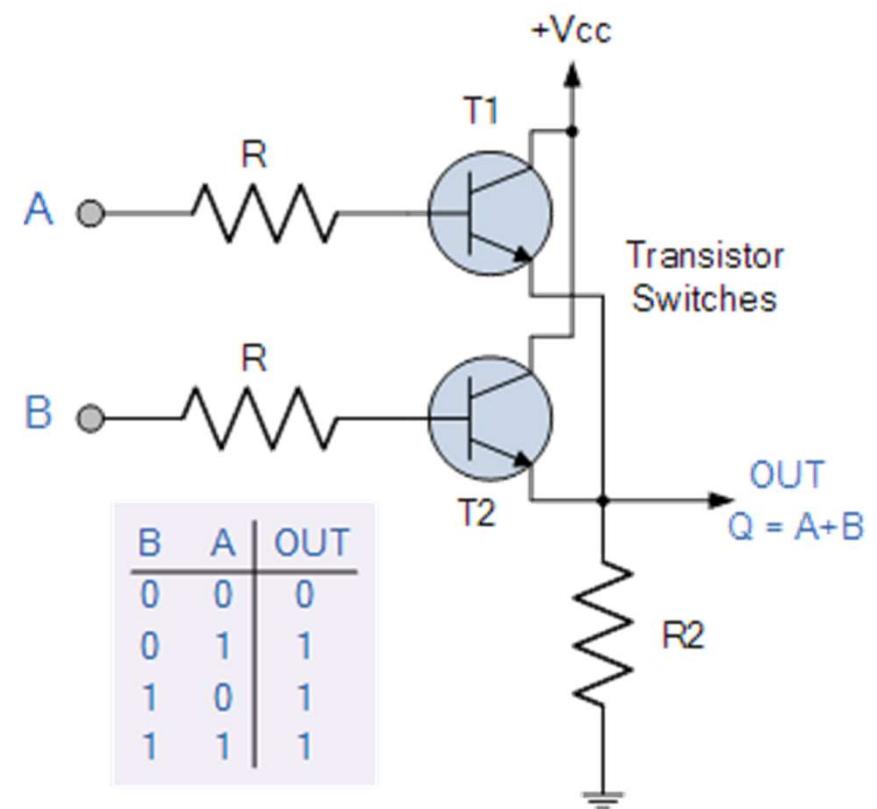
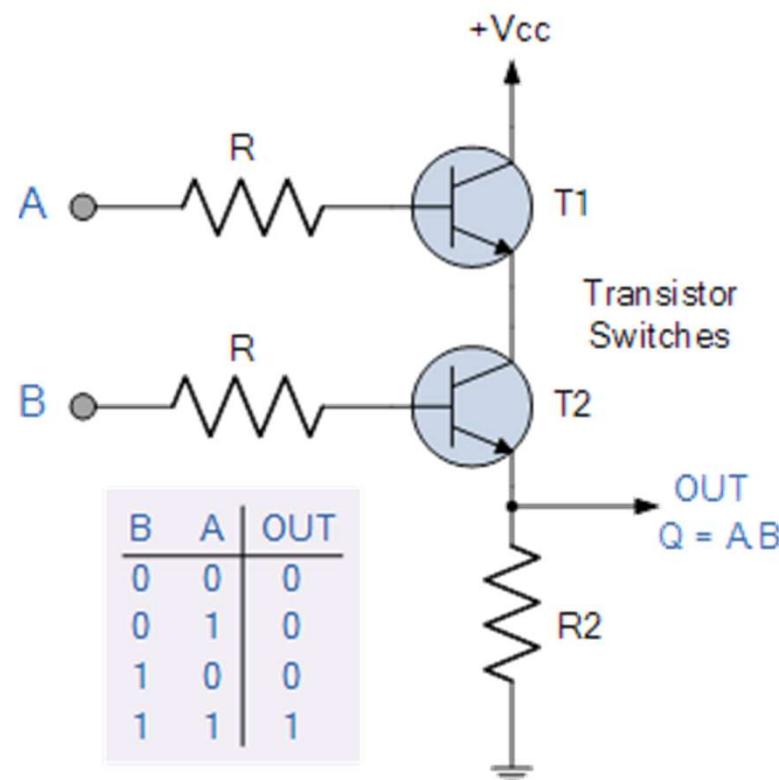
Truth Table

V_G	T_I	V_{out}
0	off	1
1	on	0

Integrated Circuit



Two Input AND & OR with Transistors



Reading Exercise

- Research and learn the following items – relate your laptop configuration for the same
 - 1. Processor (Core, Logical Processors, Thread)
 - 2. Graphics Processor (GPU)
 - 3. Memory (RAM, L₁, L₂, and L₃ Cache)
 - 4. USB Interface
 - 5. Hard Disk or SSD; Review the following HD data sheet!!

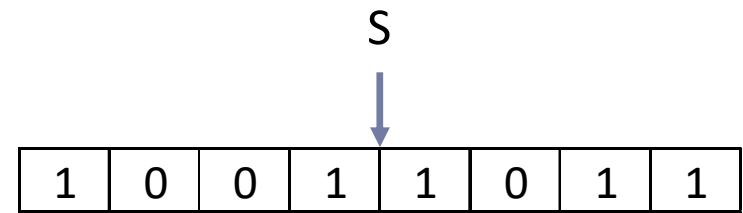
<https://www.seagate.com/www-content/product-content/barracuda-fam/barracuda-new/files/barracuda-2-5-ds1907-1-160gus.pdf>

Semi-Conductor Technologies

- All IC chips (Processor, Memory, GPU, NP, Gates, ASIC, FPGA, etc) are made of semi-conductors (diode and transistors), resistors, and capacitors
- Chip fabrications are supported by different semi-conductor technologies:
 - TTL, CMOS, NMOS, GaAs, ...

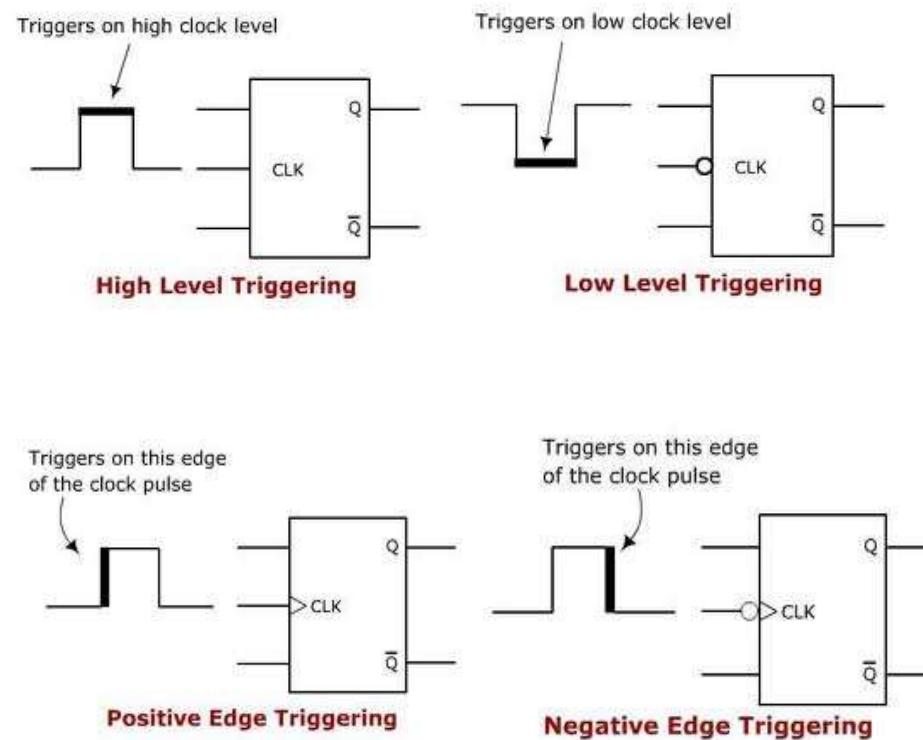
Active and Active Low Signals

- Assume there is a binary counter. We have two options to reset/set this counter either with active high (1) or active low (0) signal (S) :
 - Active high : logic 1 (S)
 - Active low : logic 0 (\bar{S})
- There is a related option – the reset/set can take place at the left edge or level or right edge of the signal

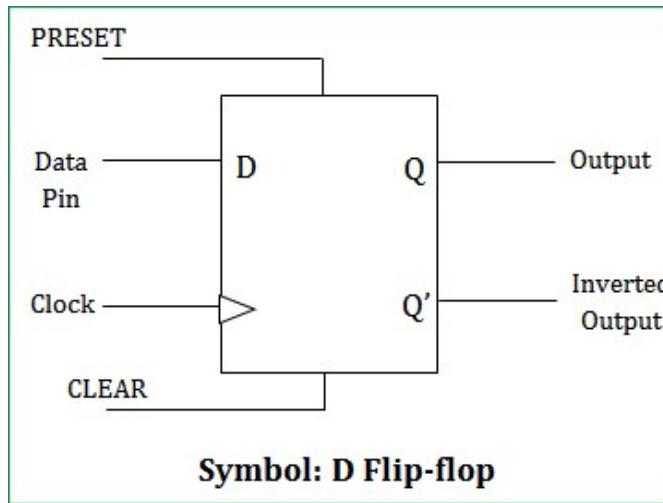


Role of Clock and Triggering

- For output stability, clock pulses are used to trigger changes to counters/registers
- There are 4 different ways clock pulse is used to trigger the change:
 - Level Triggering: Low or High
 - Edge Triggering: Positive or Negative
- Edge triggering is preferable, especially in logical systems with feedback



Toggle (D) Flip-Flop

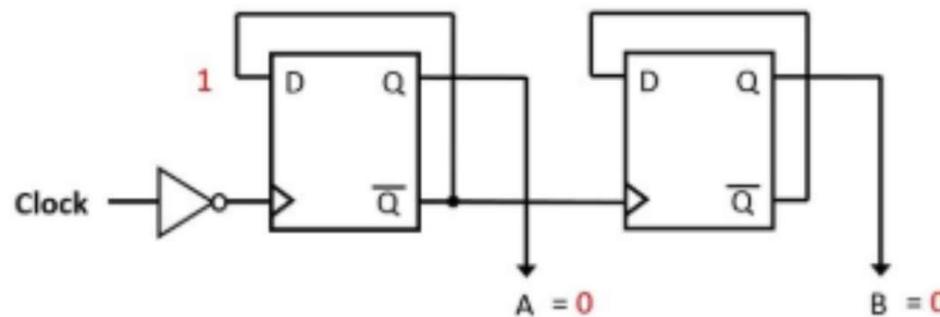


Sequential, Clocked

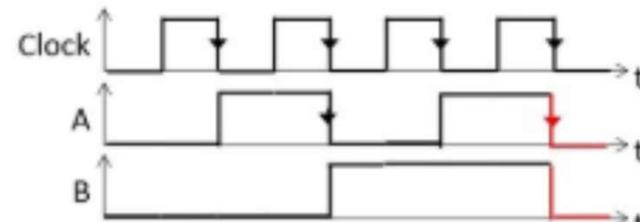
Q and Q' are complementary

Applications: Counter, Register, and Memory

2-bit Up-counter with D Flip-Flop

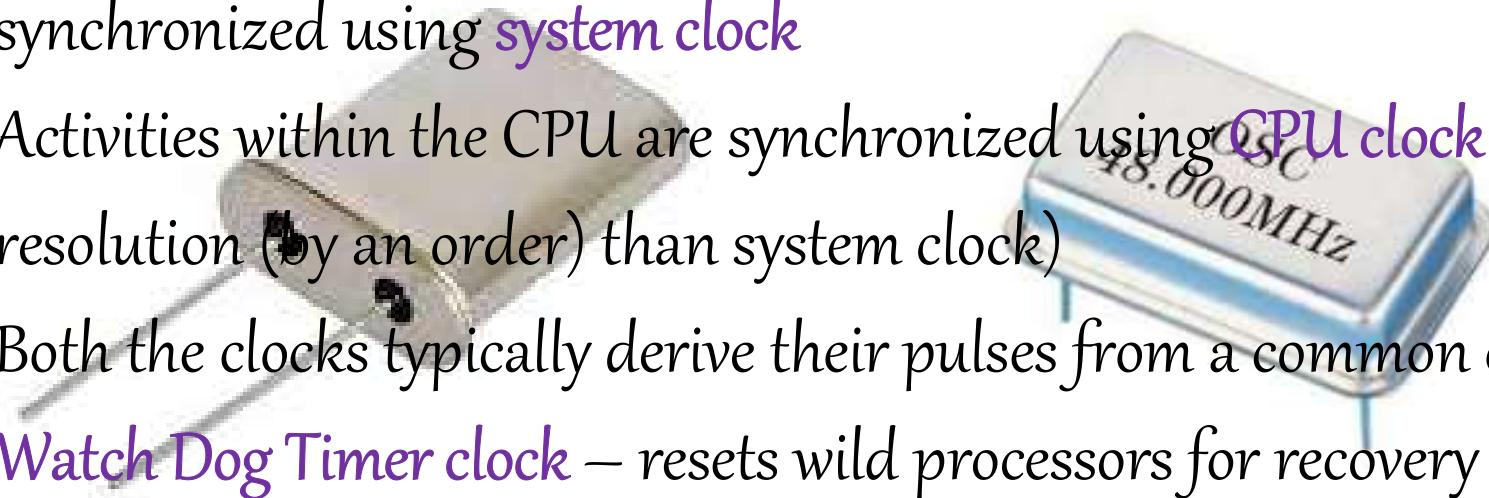


Clock pulse number	B	A
0	0	0
1	0	1
2	1	0
3	1	1
4	0	0



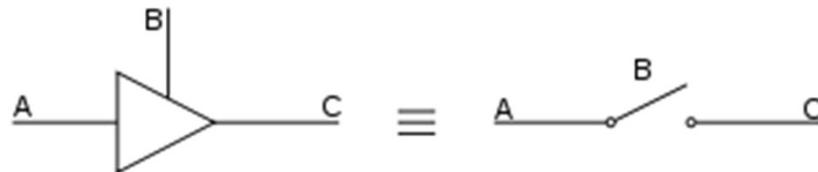
System Clock, CPU Clock, WDT Clock

- In computing system activities among various components are synchronized using **system clock**
- Activities within the CPU are synchronized using **CPU clock** (higher resolution (by an order) than system clock)
- Both the clocks typically derive their pulses from a common oscillator
- **Watch Dog Timer clock** – resets wild processors for recovery
- There are other clocks too besides these two in a computer

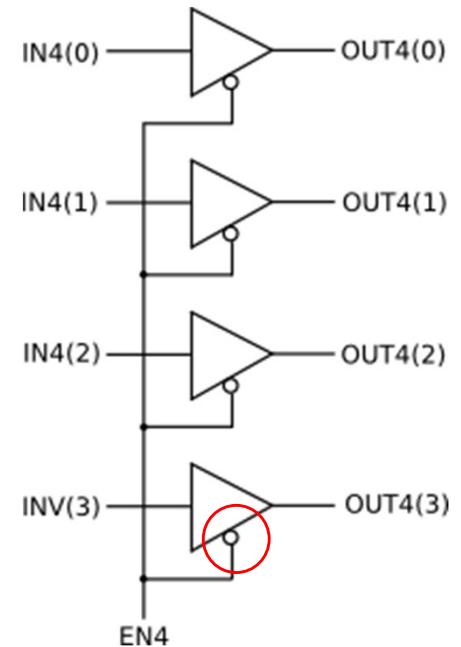


Tri-State Logic

- Tri-states: 0, 1, or high impedance state (Z)
- Useful when multiple registers are sharing the same bus
- Pictorial Notation and Truth table are shown below
- Tri-State both positive and negative logics are possible



INPUT		OUTPUT
A	B	C
0	0	Z
1		
0	1	0
1	1	1



Barrel Shift Register

- Often we need to shift and rotate bits within a register
- Shifting and rotation are slow depending upon the size of the shift
- In a normal register, shifting by 10 positions would cost 10 clock cycles
- Barrel register is designed to achieve shifting within a single or fewer cycles

Microprocessor Components (Recall SAP-1)

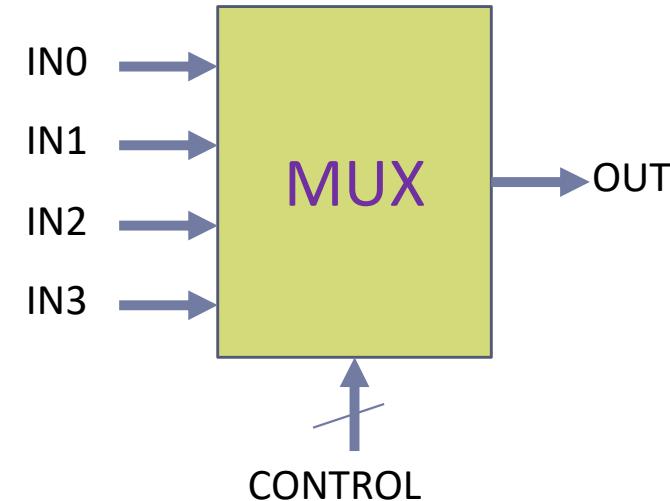
- Functional Blocks
 - Control Unit (CU) – Includes one or more Clocks
 - Arithmetic and Logic Unit (ALU)
 - Memory
 - Input-Output (I/O) devices, Clocks, and Timers
 - Instruction Register, Address Register, Program Counter, Status Register

Microprocessor Components (Recall SAP-1)

- Buses that interconnects these blocks
 - System Bus
 - Data bus
 - Control bus
 - Address bus

Lab 1

1. Install Logisim. Complete Beginners' Tutorial and Sub-circuits provided in the Help → User Guide
2. Using Logisim implement the following:
 - 8-bit AND logic
 - 8-bit OR logic
 - 8-bit NOT logic
 - 8-bit EXOR logic
3. Control every output of Exercise-2 by a 1-bit tristate logic
4. Create simple 4-to-1 multiplexor. This one is due for next lab.

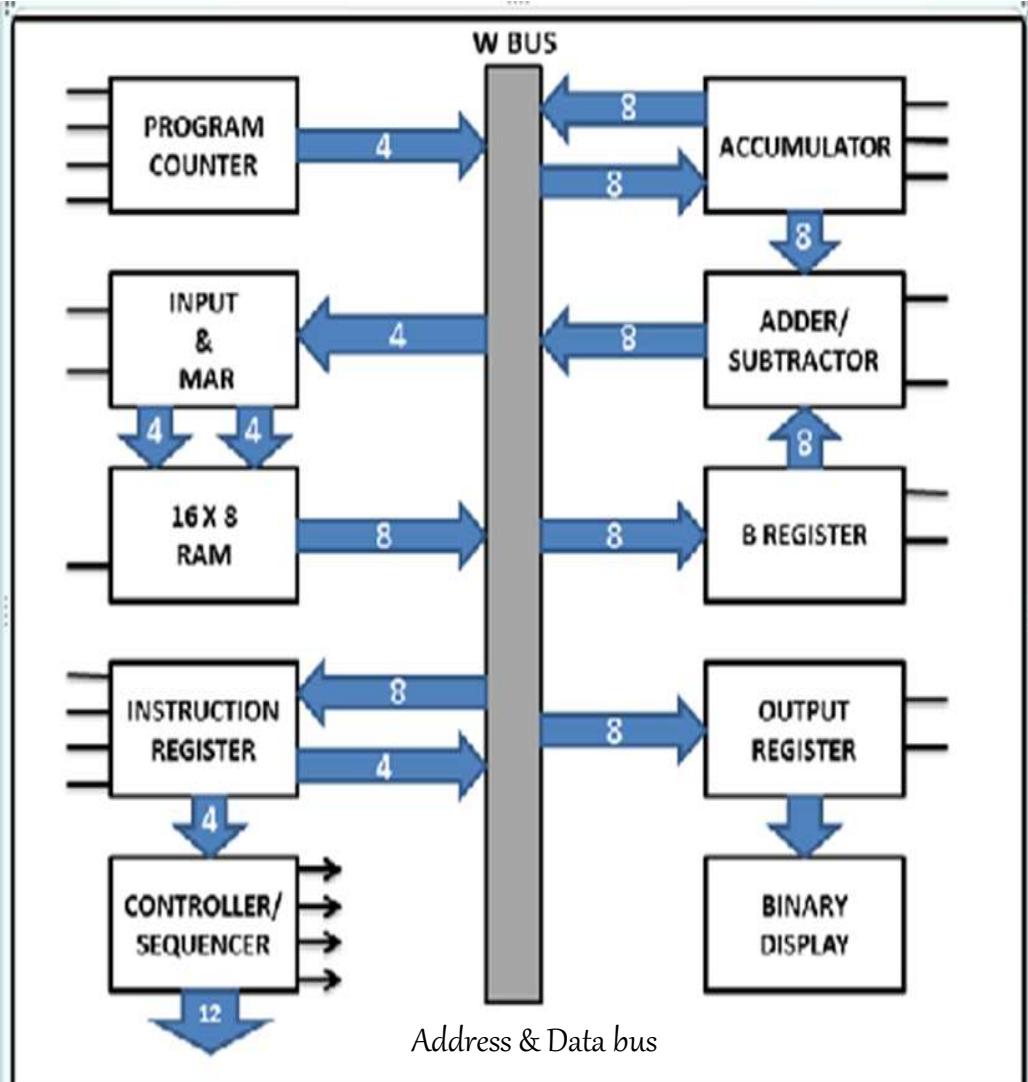


Simple-As-Possible-1 (SAP-1) Processor

FROM MALVINO AND BROWN

Computer Organization

- SAP is a simple hypothetical processor
- SAP uses Princeton Architecture
- Harvard architecture uses separate memories and buses for data and instructions respectively



SAP-1 Instruction Set

- The table in this slide shows Instructions supported by SAP-1 in two different forms:
 - Human friendly assembler mnemonic
 - Machine executable Op Code
 - Higher level languages like C are more human friendlier than assembly code

Mnemonic	Operation	Op code
LDA	Load RAM data into accumulator	0000
ADD	Add RAM data to accumulator	0001
SUB	Subtract RAM data from accumulator	0010
OUT	Load accumulator data into output register	1110
HLT	Stop processing	1111

RAM data specified with RAM Address

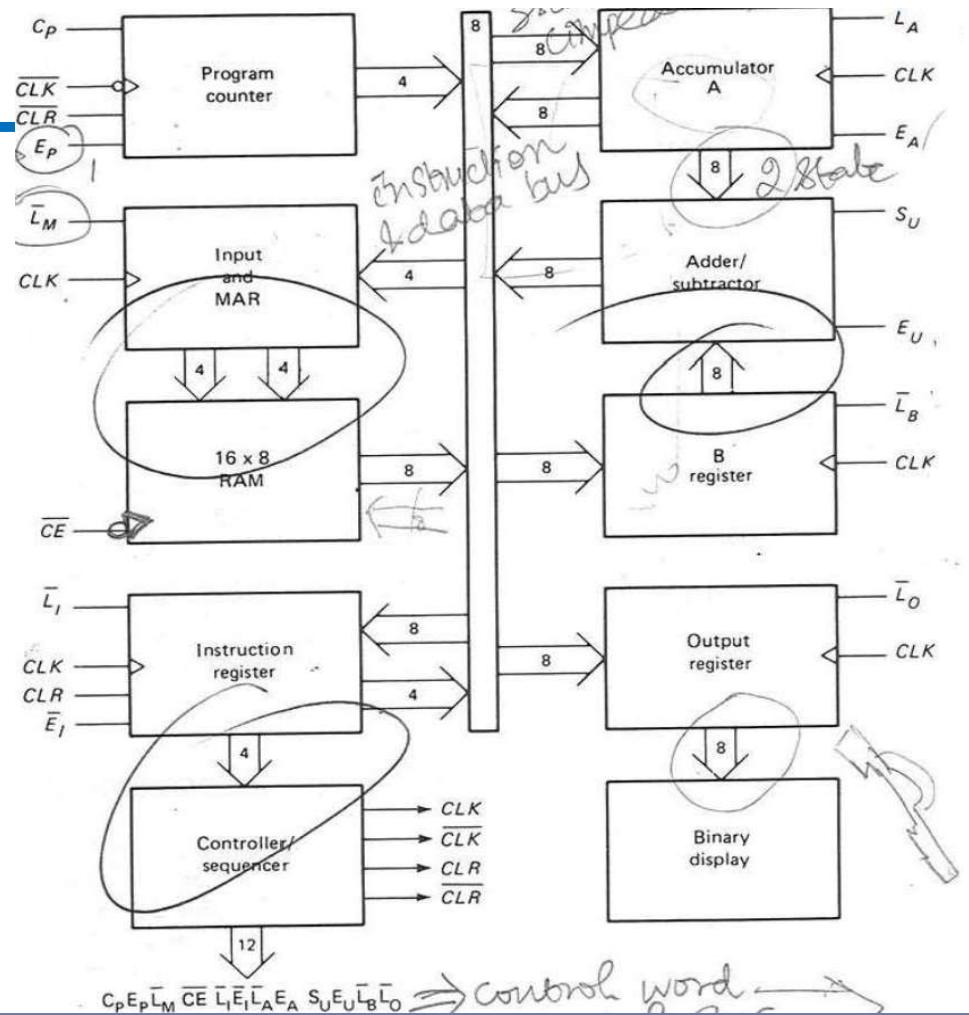
SAP-1 Control

- Control Word Example (without active sign): =

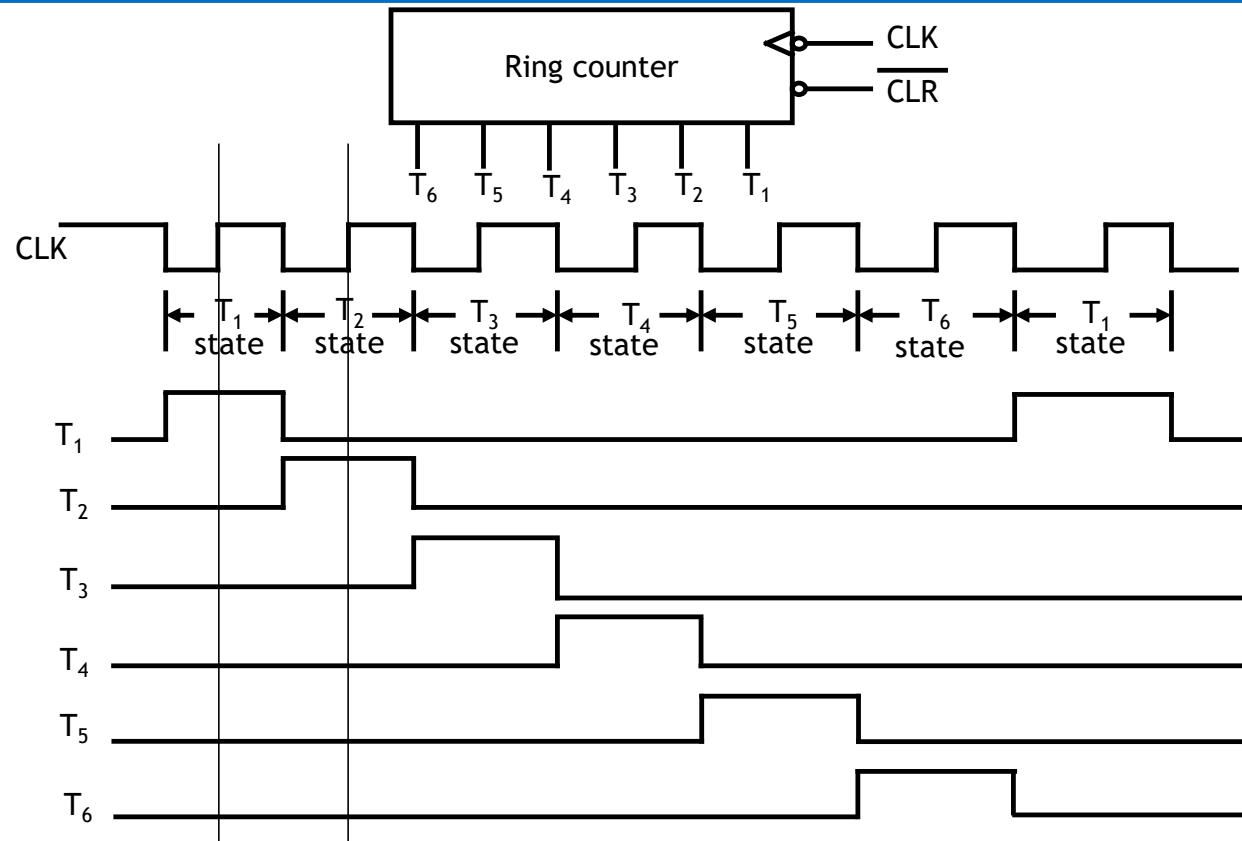
$C_P E_P L_M CE \ L_I E_I L_A E_A S_u E_u L_B L_O$

- Animated SAP Execution

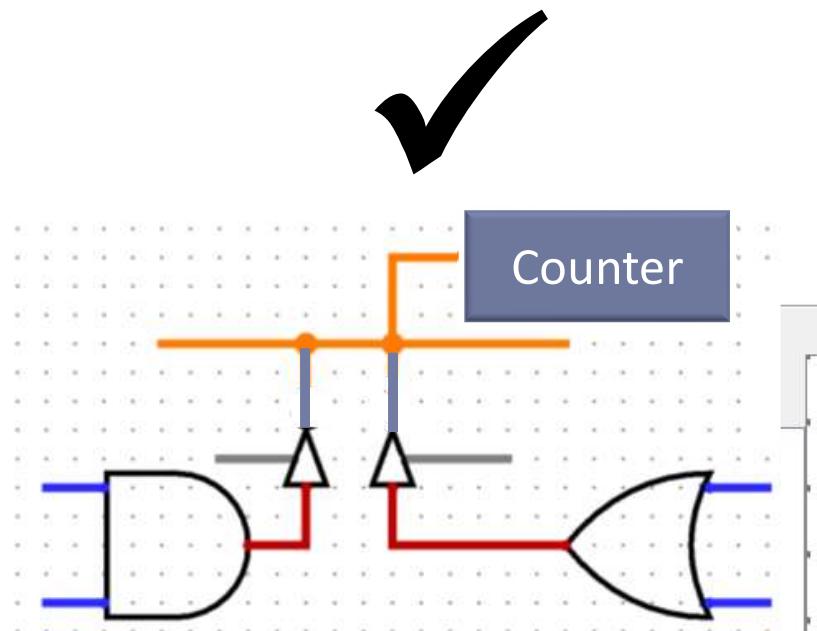
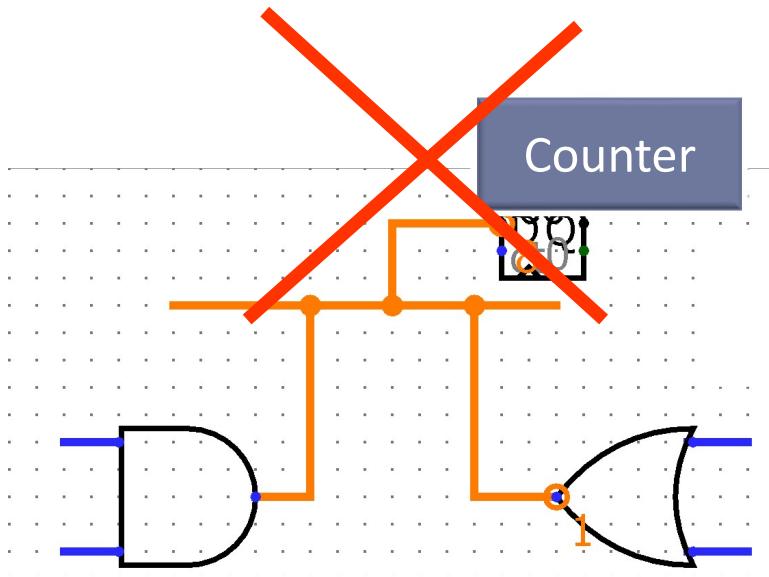
<https://www.youtube.com/watch?v=prpyEFxZCMw>



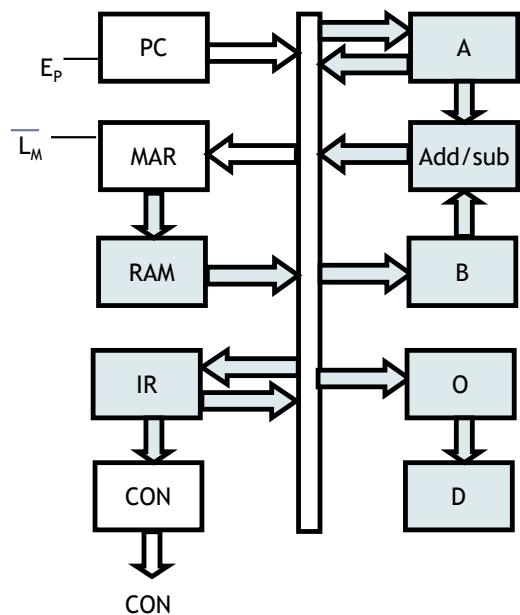
Ring Counter



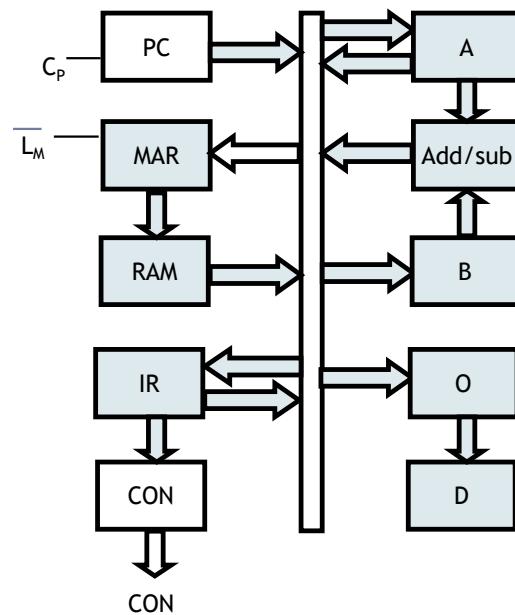
Understanding Bus and Tri-State Logic



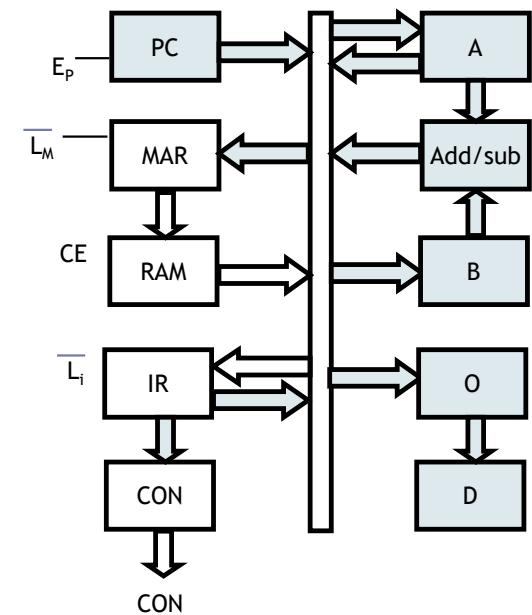
Fetch Cycles



T_1 **Address** State



T_2 **Increment** State



T_3 **Memory** State

SAP-1 Micro-program

Macro	State	CON	Active
LDA	T ₄	0x1A3	$\overline{L_M}, \overline{E_I}$
	T ₅	0x2C3	CE, L _A
	T ₆	0x3E3	None
ADD	T ₄	0x1A3	$\overline{L_M}, \overline{E_I}$
	T ₅	0x2E1	CE, L _B
	T ₆	0x3C7	S _U , E _U , L _A
SUB	T ₄	0x1A3	$\overline{L_M}, \overline{E_I}$
	T ₅	0x2E1	CE, L _B
	T ₆	0x3CF	$\overline{S_U}, E_U, L_A$
OUT	T ₄	0x3?2	E _A , $\overline{L_O}$
	T ₅	0x3E3	None
	T ₆	0x3E3	None

Write a SAP-1 Program

To Evaluate the following expression: $(10+6) - 8$

Address	Content (Mnemonic)	Content (Binary)	Content (Hexa)
0x0	LDA 0x6	0000 0110	0x06
0x1	ADD 0x7	0001 0111	0x17
0x2	SUB 0x8	0010 1000	0x28
0x3	OUT	1110 0000	0xE0
0x4	HLT	1111 0000	0XF0
0x5	XXXX XXXX	0001 0111	0x17
0x6	6	0000 0110	0X06
0x7	10	0000 1010	0X0A
0x8	8	0000 1000	0X08

Write the Microcode for Fetch Cycles T₁, T₂, and T₃

Macro	State	CON	Active Signals
Address State	T ₁	0x5E3	E _P , \bar{L}_M
Increment State	T ₂	0xBE3	C _P
Memory State	T ₃	0x263	CE, \bar{L}_I (Rest of the signals are inactive)

$$\begin{matrix} C_P E_P \bar{L}_M \bar{C_E} & \bar{L}_I \bar{E_I} \bar{L}_A E_A S_u \bar{E_u} \bar{L}_B \bar{L}_O \\ 0101\ 1110\ 0011 \\ 1011\ 1110\ 0011 \\ 0010\ 0110\ 0011 \end{matrix}$$

Exercise

- You are asked to replace SAP's ADD and SUB instructions to Logical AND and OR respectively.
- Which functional blocks of SAP need to be updated to accommodate this change.
- Describe those changes

Exercise

- Assume that this ALU can also do Bitwise AND and OR operations
- Make the necessary changes to CON and ALU (control inputs) to support these new ALU operations
- These two new operations also require two new instructions. Define and these two instructions with necessary changes
- Write the microcode sequence (T_1, \dots, T_6) for these two new instructions
 - Scope of changes:
 - Keep the Memory size, Register Size, the bus size, and Path to bus intact.
 - You can modify, add new opcodes and new signals.

Exercise

- Can you optimize the ring counter to operate with 5 cycles instead of 6? This should make SAP faster by 20%!
- Hint: You can overlap what is done in T6 with what is done at T₁. Think, and describe your solution with control words for every SAP-1 Instruction (T₁, ..., T₅)

Exercise

- Explore if this possible:
 - Increase the memory to 32x8 for the same instruction set
 - Instructions appear from address 0 onward – up to address 15
 - Data can appear after instructions, and up to 31.
 - What changes are needed, how will you accomplish this.

Exercise

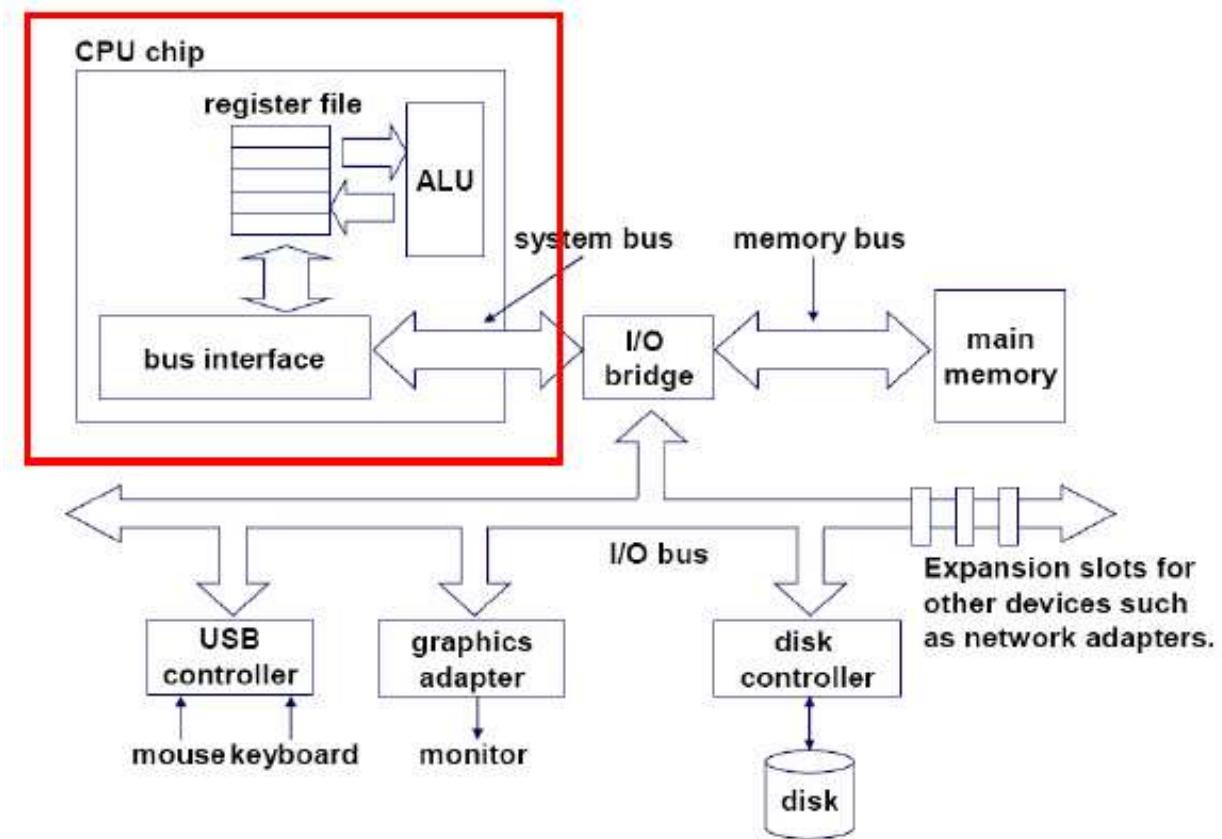
- Sketch and explain the digital circuit design of the 4-bit program counter of SAP.
- The inputs are Cp and Ep. Active Cp increments the counter.
- The output is the counter value with tri-stage logic – controlled by Ep
- For demonstration use LEDs
- Use Logisim to implement and test this.

Exercise

- Sketch and explain the digital circuit design of the 4-bit program counter of SAP.
- The control inputs are C_p and E_p . Active C_p increments the counter
- The output is the counter value controlled (E_p) with tri-stage logic
- Design a 4-bit register. Move the output of the tri-state logic from the counter to this new register with a control input (E_l).

Single-Core Computer

CPU chip also includes some amount of *Cache memory* and a large part of *Control logic*



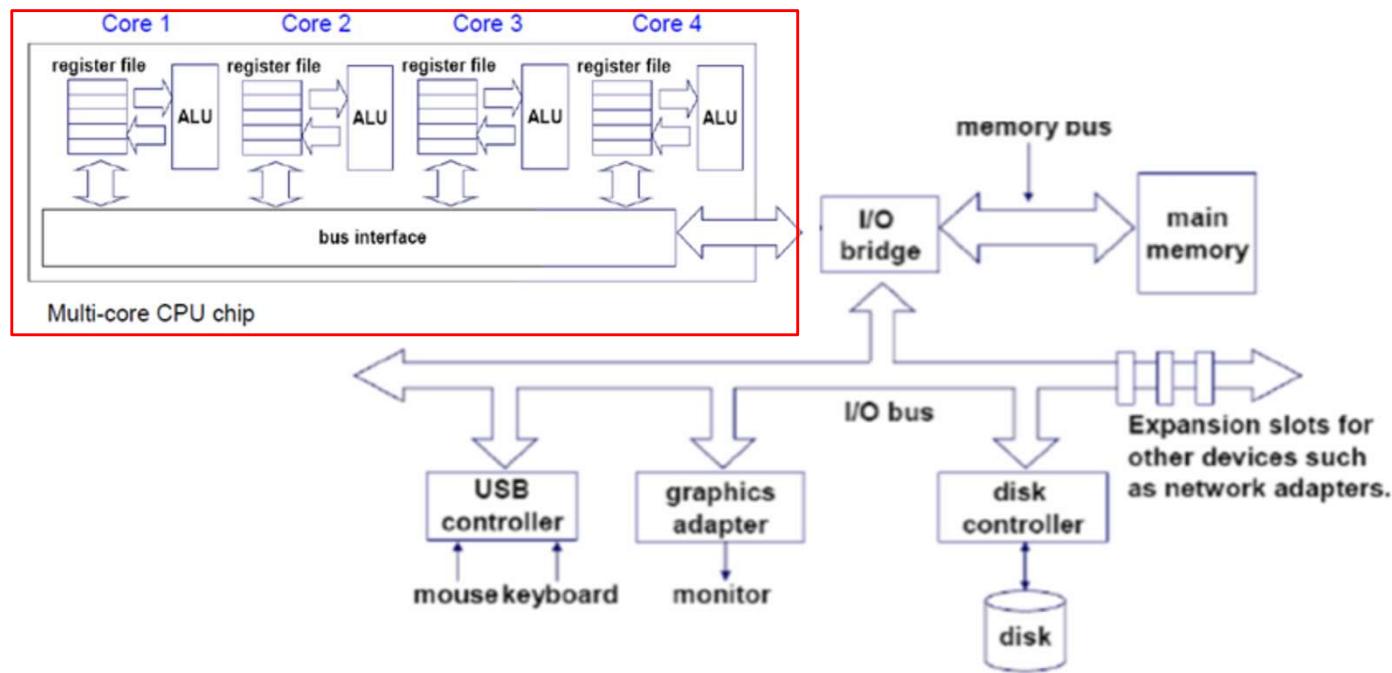
Next 4 slides: Jernej Barbic: Multi-core architectures

Single Core and Multi-Core Processors

- *Multi-Core* processor contains multiple processors (CPU) within a single package unlike *Single Core* processor
- There are, however, lots of similarities at a single CPU level between the two
- We are going to study mostly *single-core CPU* in this course
- All micro-processors use *pipeline concept* to improve instruction throughput these days

Multi-core Processor

- ❑ Multiple processor cores on a single *die*
- ❑ Super Scalar design is another way of increasing the throughput- study later
- ❑ Super scalar can be combined with multi-core



My Laptop Configuration Using CPU-Z

CPU-Z

CPU Caches Mainboard Memory SPD Graphics Bench About

Processor

Name	Intel Core i5 7200U		
Code Name	Kaby Lake-U/Y	Max TDP	15.0 W
Package	Socket 1356 FCBGA		
Technology	14 nm	Core VID	0.883 V



Specification

Intel® Core™ i5-7200U CPU @ 2.50GHz			
Family	6	Model	E
Ext. Family	6	Ext. Model	8E
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3		

Clocks (Core #0)

Core Speed	2493.90 MHz
Multiplier	x 25.0 (4 - 31)
Bus Speed	99.81 MHz
Rated FSB	

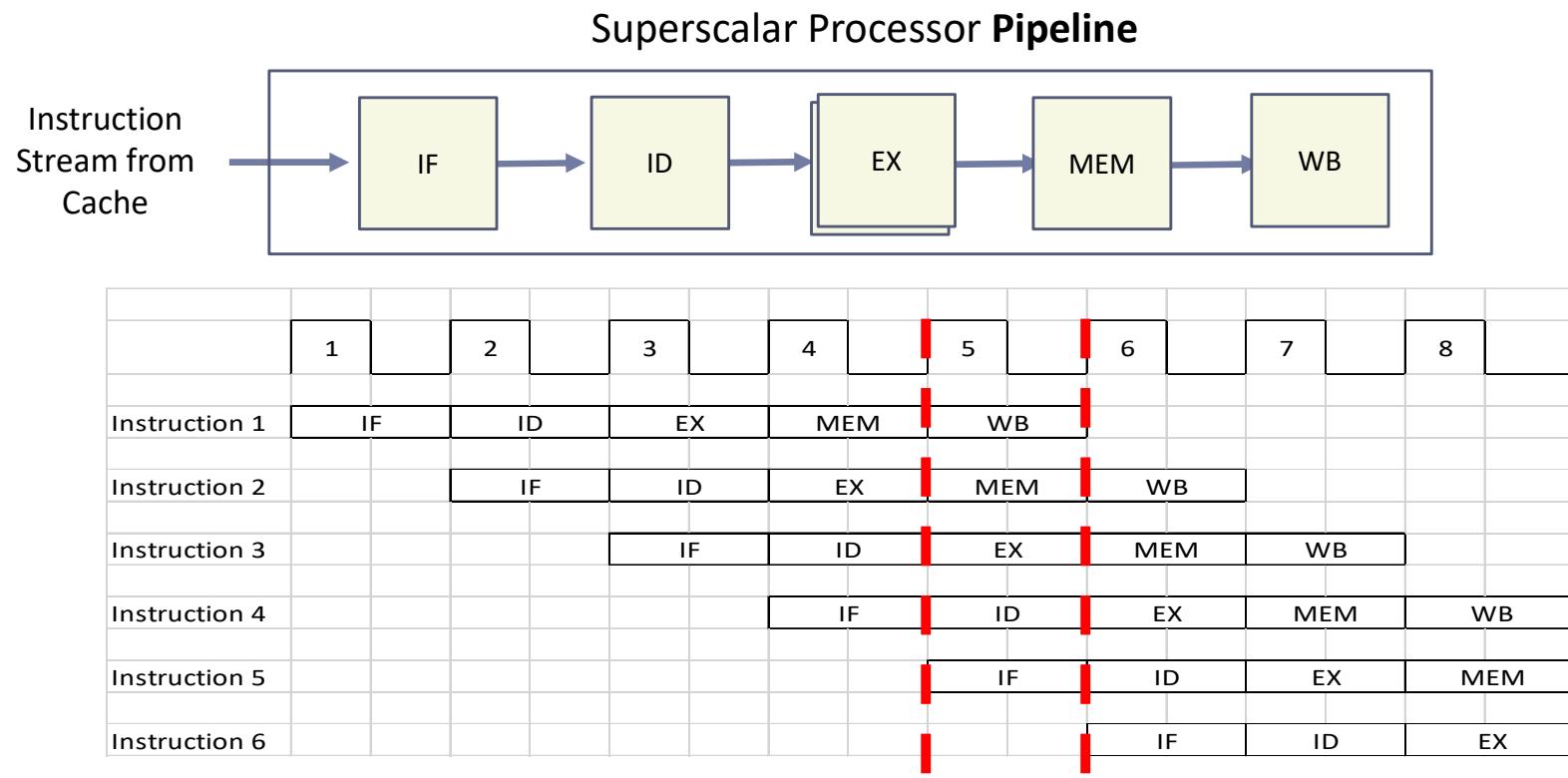
Cache

L1 Data	2 x 32 KBytes	8-way
L1 Inst.	2 x 32 KBytes	8-way
Level 2	2 x 256 KBytes	4-way
Level 3	3 MBytes	12-way

Selection Socket #1 Cores 2 Threads 4

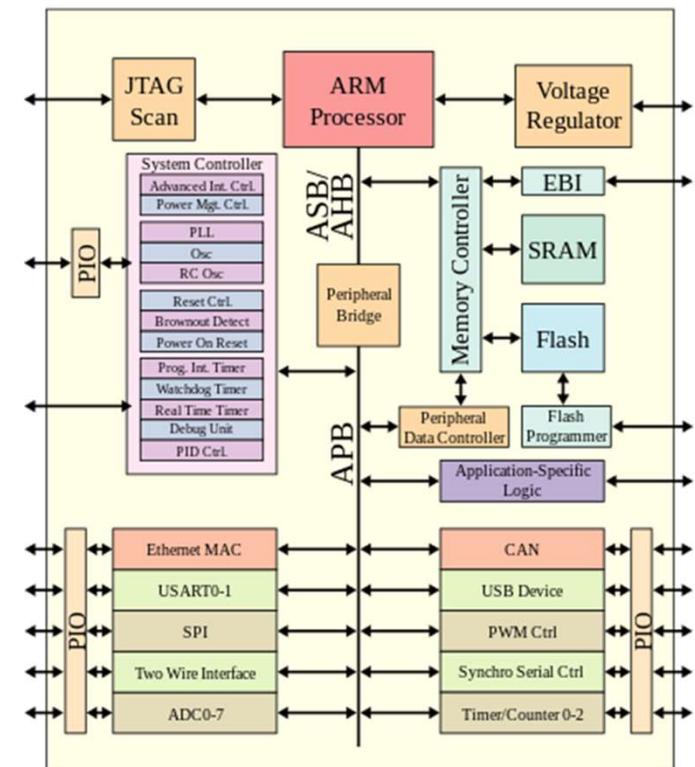
CPU-Z Ver. 1.90.0.x64 Tools Validate Close

Superscalar Processor



System on Chip (SoC)

- A system on a chip (SoC) is an IC that integrates several components of a computing
- Some microcontrollers are designed similarly
- More functions that fall outside of MC are included in SoC – Radio, I/O Interfaces, Graphic processor, Arithmetic Accelerators, etc.
- This allows more compact and power efficient embedded systems to be built



IoT Mote

- A self contained IoT Node
 - With built-in camera
 - Power supply
 - Radio
 - Processor, memory etc.
 - OS with networking stack
 - Other functions



Michigan Micro Mote (M3)

With temperature sensor and custom antenna

Watch this British Movie "***Eye in the Sky***" Circa 2015!

Program and Process

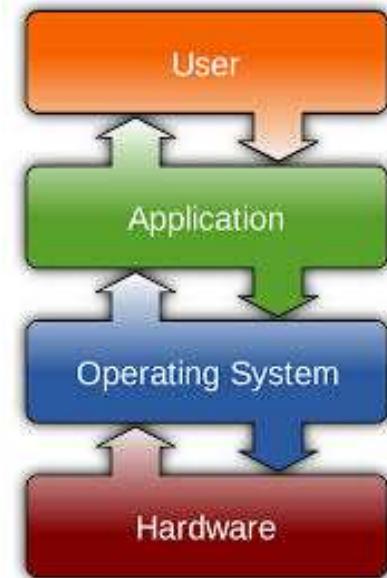
- A *program* becomes a *process* when it is loaded to run
- In non-real time, single core systems, CPU allocation is done at a process level
- CPU can also be allocated to other run time units like *Task* and *thread*.
- *Task* is a light weight process that interwork with other tasks in real time systems

Assembler, Compiler, and Interpreter

- Assembler is a program that translates programs written with assembly mnemonics to machine code
- Assembler translation is simple, it is almost one to one mapping
- Translating Higher level languages to assembly/machine code is lot more challenging. This is achieved by Compiler program
- A C compiler written for a processor, translates C programs optimally to a processor (using the intimate knowledge of the processor design).

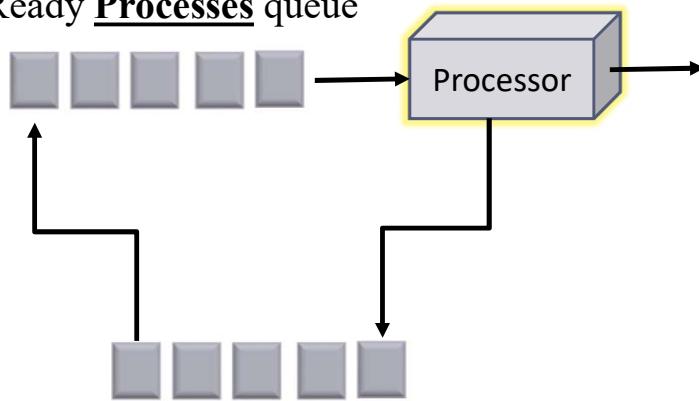
Operating System (OS)

- OS – An abstraction layer to h/w
- OS includes the following functions
 - Scheduler (of processor)
 - Memory Management System (MMS)
 - File System
 - Networking Stack
 - And features like – IPC, Support for Threading, Event Handling, & Power savings, Security



OS Scheduler, Process State, & Process Queues

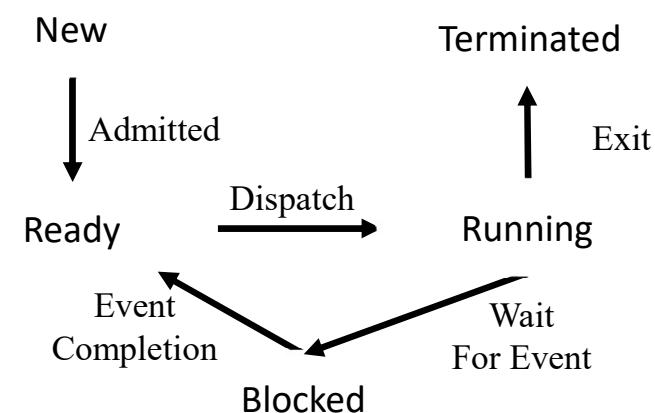
Ready Processes queue



Blocked Processes queue



Process



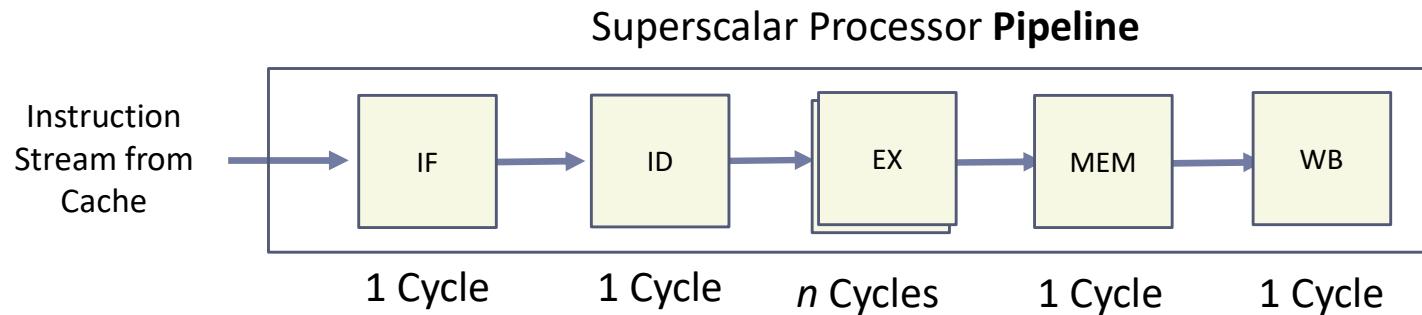
Process State Transition Diagram

Process State

- Scheduler is the OS component that manages process state transition
- Process ID is used in identifying and managing a process

Non-ideal Superscalar processor

- In an ideal superscalar processor each FU takes 1 cycle to complete



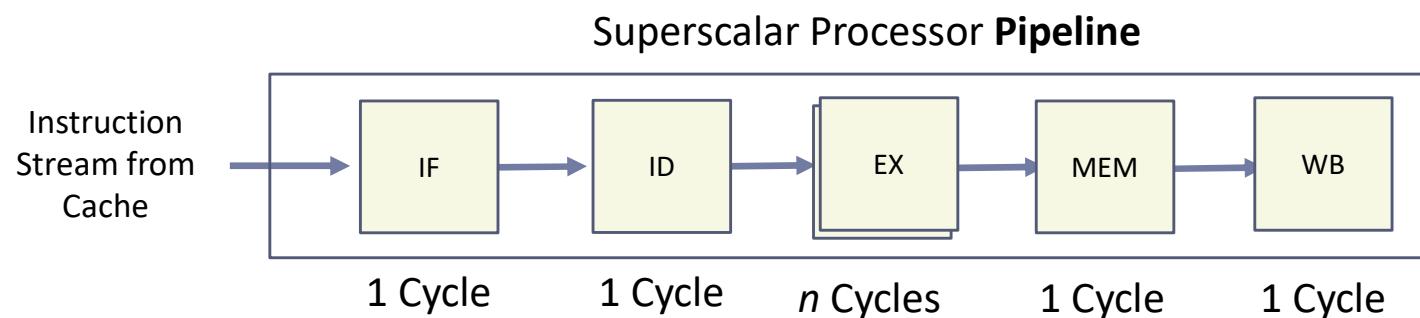
- Threading/interleaving is a solution to reduce the wasted CPU cycles
 - When one thread stalls other thread is scheduled to keep the pipeline busy
 - This is done with extra hardware

Interleaved & Parallel Execution

- *Interleaved execution in a single processor fulfils at least two requirements:*
 - *Active process in focus* is given the CPU, other processes are put in *wait queue*
 - CPU utilization is improved by giving the *processor* to an active *process*
- *Parallel execution is relevant in multi-processor environment*
 - To exploit the presence of larger number of processors
 - To improve both *response time* and *throughput*
 - *Interleaving is also practiced within each processor*

Non-ideal Superscalar Processor

- Another case of non-idealism is the presence of branch instruction
 - Execution of a branch (Goto) instruction can make the pipeline to reset – this wastes some CPU cycles



Superscalar and Threading

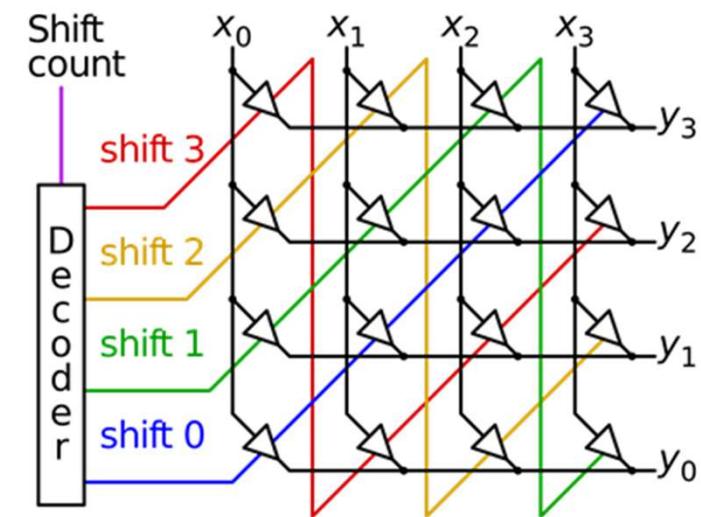
- ❑ Superscalar processor provides instruction level parallel operation
- ❑ Threading enables parallel operation across two or more streams of independent instructions

Reading Exercise

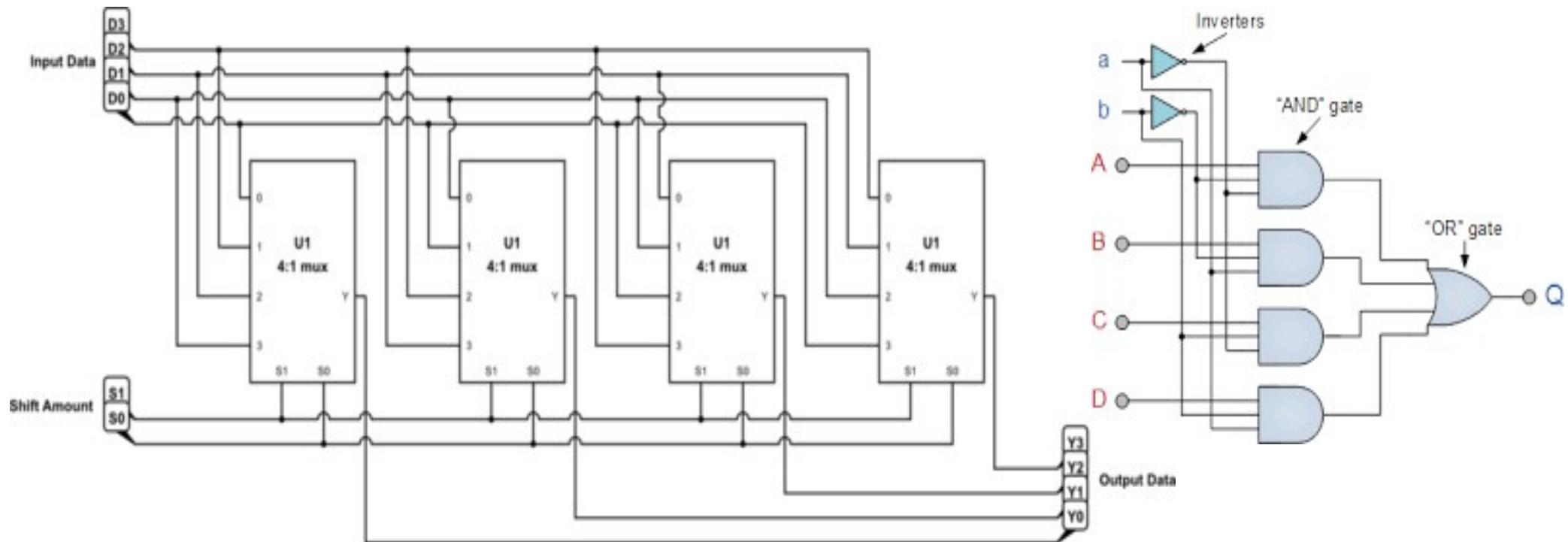
- Learn how *thread operations* are offered and supported by
Posix *Pthread library*

Lab 2 – build with gates and diodes!

- Using Logisim implement the 4-bit barrel shifter with the corresponding 2-bit decoder as shown in the following figure.
- Create a full adder then using this create a 4-bit Ripple Carry Adder using Logisim
- Create a 8-to-1 Multiplexor. Using that implement 3-input AND gate. Then using the same Mux implement 3-input OR gate.



Barrel Shifter with Mux?

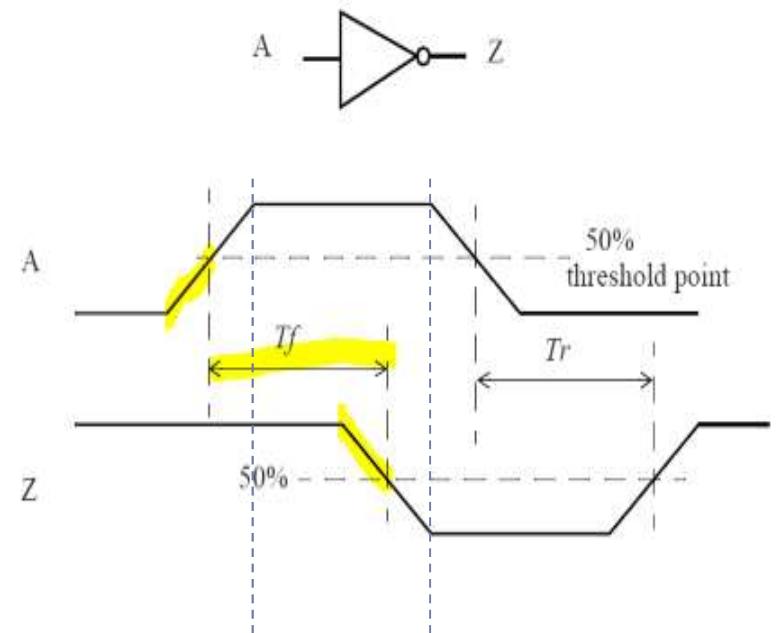


SAP Versus ARM

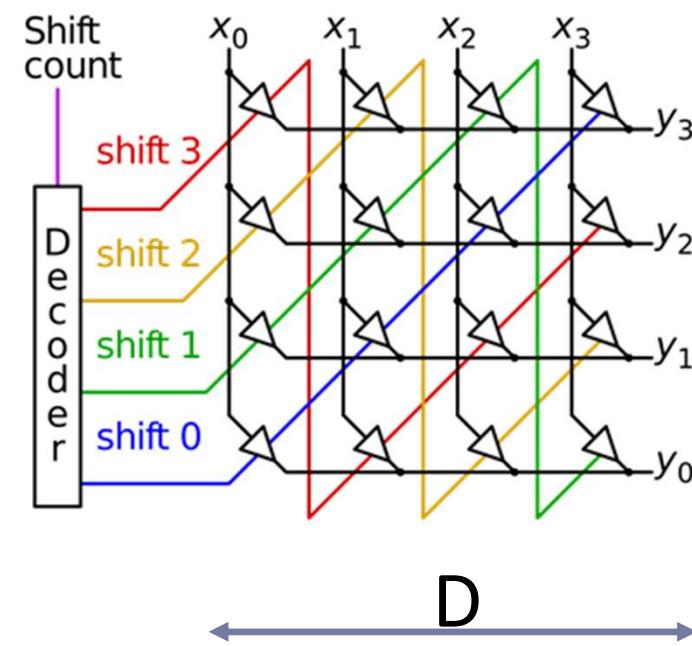
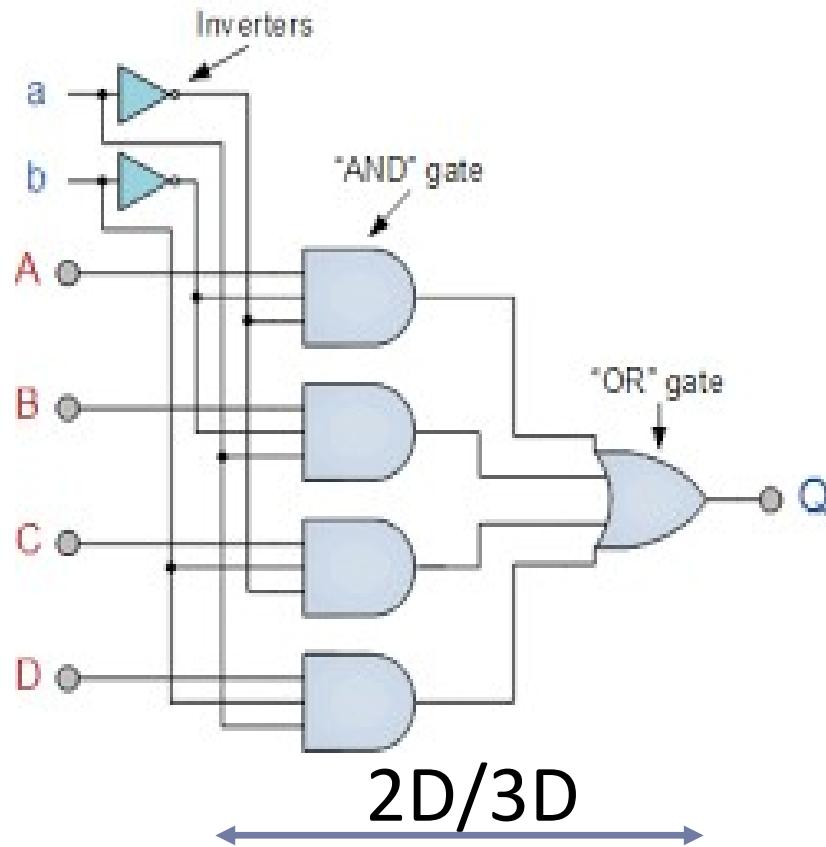
- SAP is a non-superscalar processor
 - Fetch Instruction (Cycles 1 to 3), Decode it (4...), Execute it
 - Then start the next instruction and so on
- ARM is a 3-stage superscalar processor
 - In general, in each cycle, 3 instructions are processed:
Fetch instruction N+2, Decode instruction N-1, Execute
instruction N

Propagation Delay – Logic Gate

- The time taken for the output to change when the input signal changes
- We will assume that this delay to be
 - Independent of the gate logic
 - Technology specific
 - This is one of the factors that decides the pulse duration of the system clock



Propagation Delay for Complex Logic



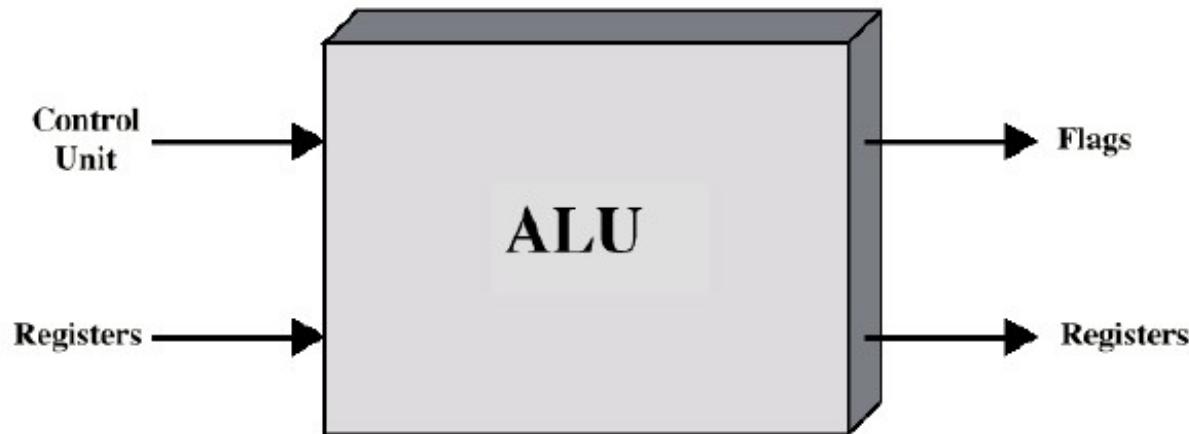
Reference

- <https://mail.google.com/mail/u/1/#inbox/1MfcgxwGDDjqNRxlhTJnnhjvNFxnMdNz?projector=1>

VEMULA LAKSHMI SRINIVAS

Arithmetic and Logic Unit (ALU)

Arithmetic Logic Unit in CPU



Discuss roles of inputs and outputs to ALU

Let assume 32 bit ALU

Focus is more on representation and algorithm rather than computational circuit

Basic Operations Discussed

- Unsigned Arithmetic Operations: +, -, %, **x**
- Logical Operations: AND, OR, NOT, EXOR
- Bitwise Operations: AND, OR, NOT, EXOR
- Signed Arithmetic Operations: +, -, %, **x**
- Floating Point Operations: +, -, %, **x**
- Shift and Rotate operations

ALU

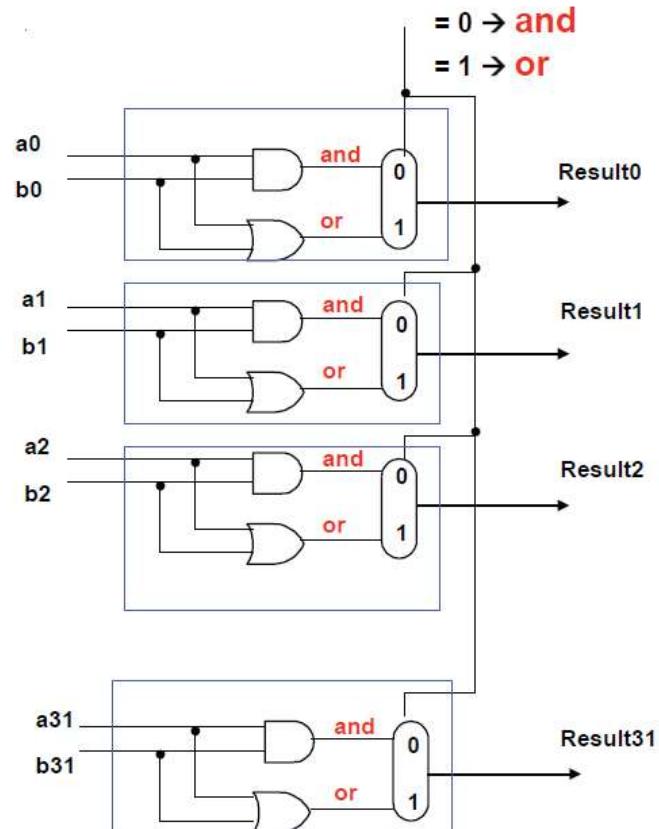
- Arithmetic and Logic Unit (ALU) performs Arithmetic (+, -, \times , %) and logic (AND, OR, EXOR, etc.) functions!!
- The ALU could be a collection of multiple Functional Units (FU)
- The basic FU performs Integer and Logic operations and special purpose ones perform for more complicated operations like Multiplication, Division, floating point operations, etc.
- There can be replicated FUs for parallel execution

Integer Arithmetic & Logical Operations

- Integers could be
 - Unsigned or signed - weighted notation
 - BCD
 - ASCII/Unicode
- Addition forms the basis for other operations like subtraction, multiplication, division, etc.
- Rotate and shift operations are simple in terms of required logical circuit compared to arithmetic operations including addition

Logical and Bit-wise Operations

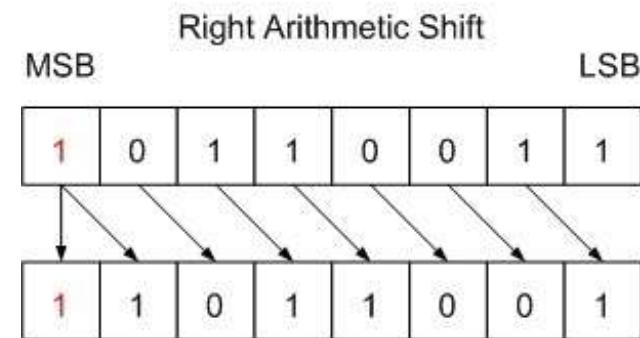
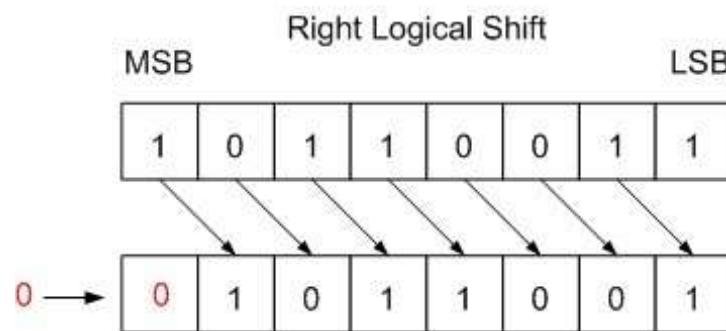
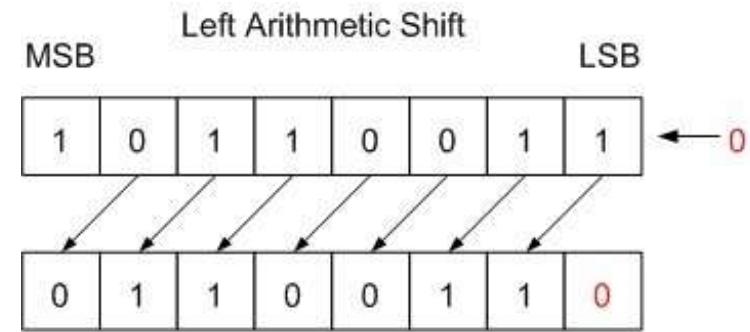
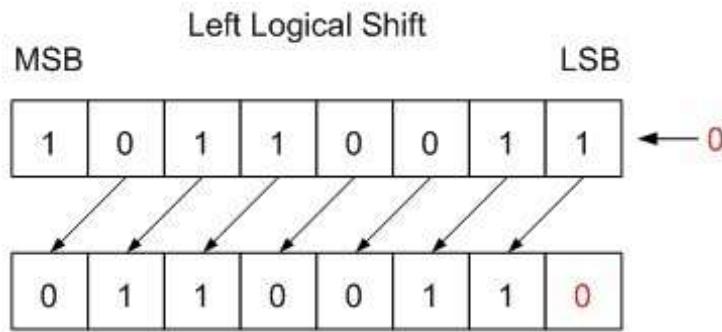
- Bitwise AND, OR, XOR, NOR
 - Implement 8-bit AND and OR operations using Logisim
 - 8 different logical operations; the Control input size?



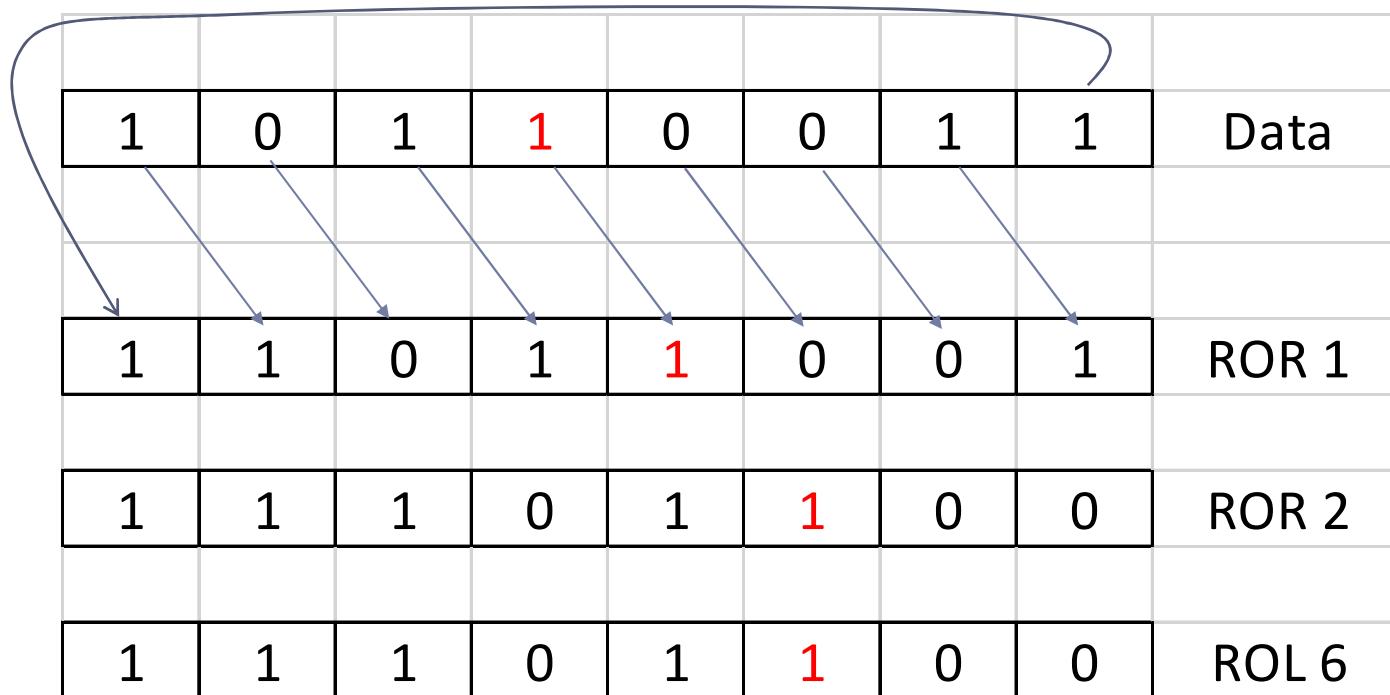
Shift and Rotate Operations

- `rol` => rotate left ($\text{MSb} \rightarrow \text{LSb}$) - Integral Multiplication of 2
- `ror` => rotate right ($\text{LSb} \rightarrow \text{MSb}$) - Integral Division of 2
- `sll` -> shift left logical ($0 \rightarrow \text{LSb}$)
- `srl` -> shift right logical ($0 \rightarrow \text{MSb}$)
- `sra` -> shift right arithmetic (old $\text{MSb} \rightarrow$ new MSb)
- We will study them with ARM assembly program

Shift Operations

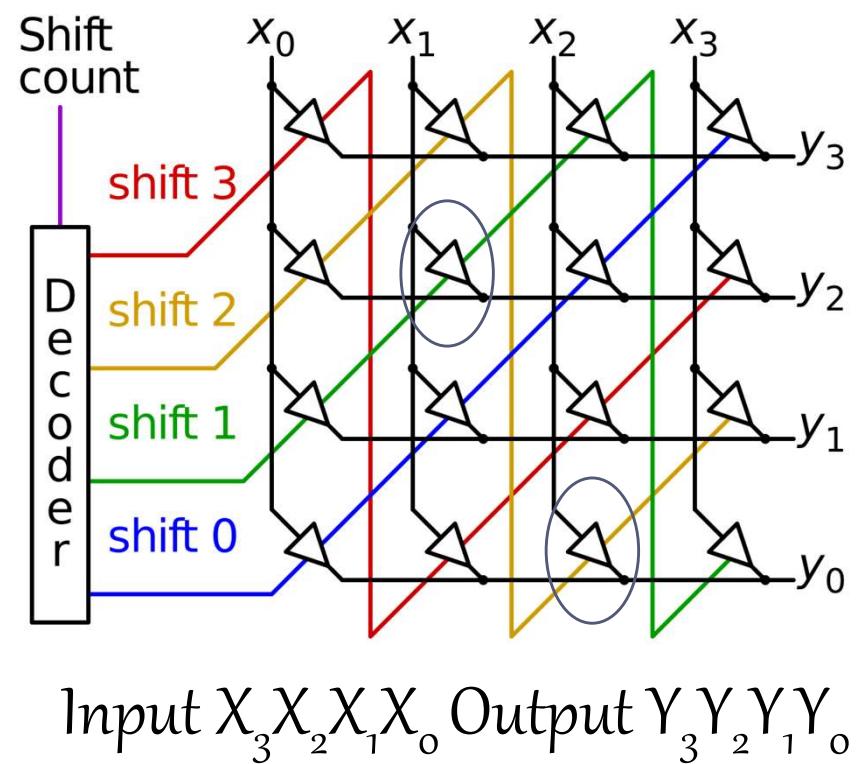


Simple Rotate Operation



Shifter & Barrel Shifter

- Single bit shift/rotate (right/left) in a single register is trivial
- Multi-bit (n) shift delay is $O(n)$ with simple shifter
- Barrel register is a complex shifter/rotator with better performance



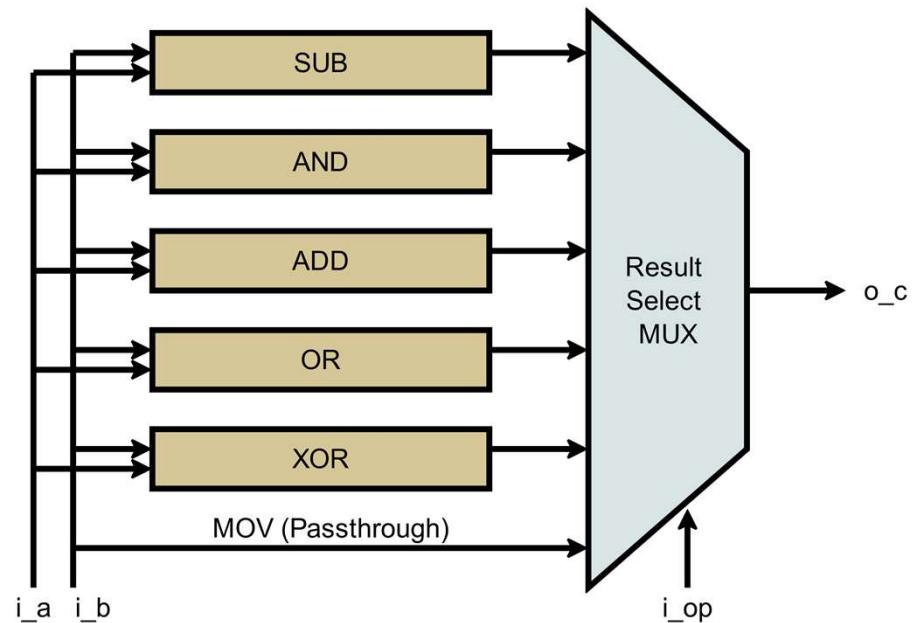
Unsigned Integer Representation

- ❑ Unsigned integers are encoded using weighted binary representation

31	30	29		0		Decimal
0	0	0	...	0		0
0	0	0	...	1		1
1	1	1	...	1		$2^{32} - 1$

Plan

- Realizing ALU for individual logical operation is simple and implementing ALU for multiple logical operations is a bit of challenge but still simple
- Let's look at arithmetic operations starting with unsigned integers



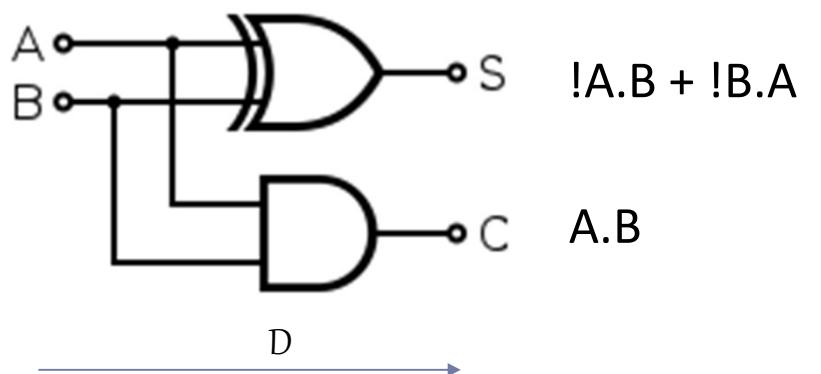
Adders

- Adder logic adds binary numbers
- The inputs are two binary numbers
- The output is sum (S) and carry (C)
- There are a variety of adders with varying complexity
- The primitive ones are *Half adder* and *Full adder*

Half Adder (Source Wikipedia)

- The half adder adds two single binary digits A and B .
- It has two inputs (A, B) and two outputs, sum (S) and carry (C).
- One simple logic circuit and Truth table for half adder is shown below

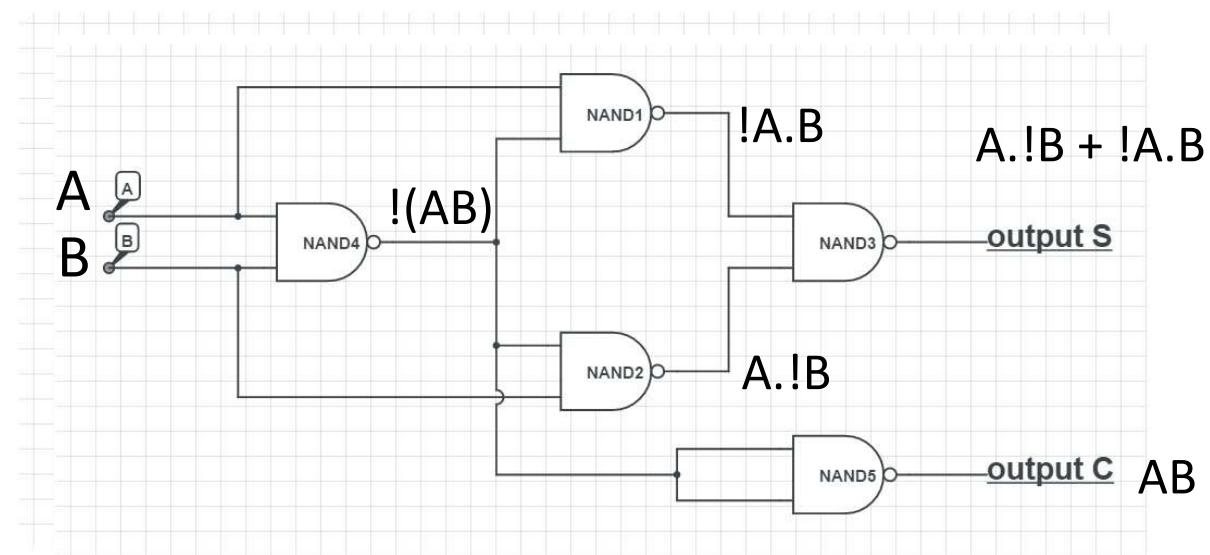
Inputs		Outputs	
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0



Half Adder with Nand gates

- The output is stable after 3 gate delays (3D)

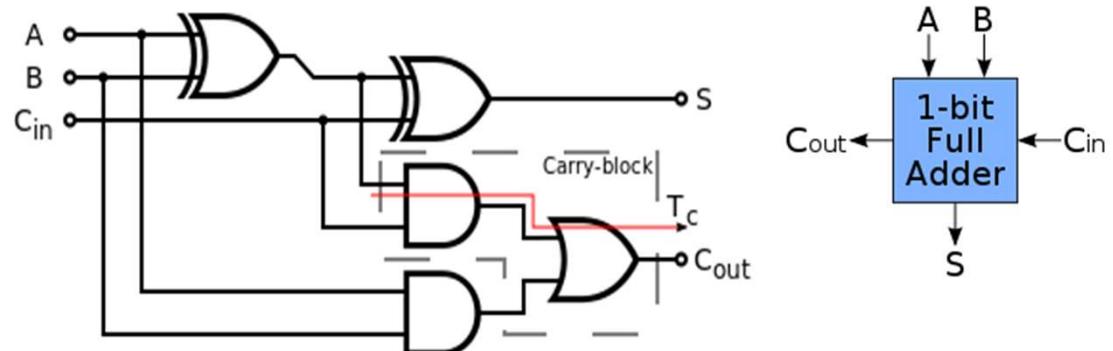
Inputs		Outputs	
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0



Full Adder

- Unlike half-adder, full adder receives 3 inputs. More pragmatic than half-adder
- The third input is the carry from previous addition

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



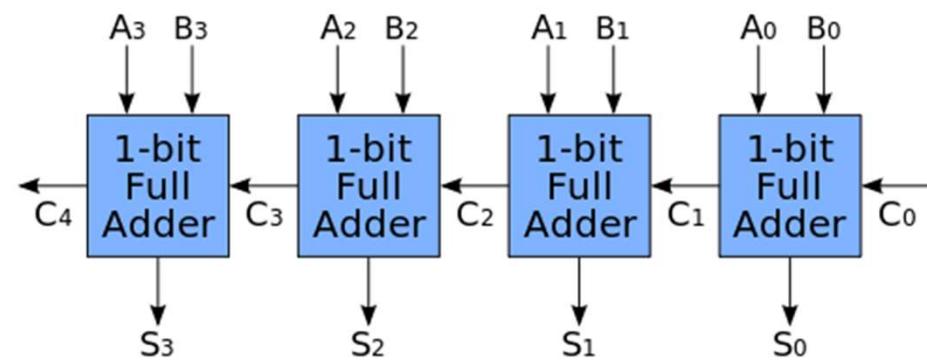
Logic Circuit

Symbolic

The carry is available after 3D, Sum after 2D

Multi-bit Adders ✓

- 4-bit Ripple Carry Adder (RCA) – simple circuitry but slow
- The single stage output is stable after 3 gate delays (D)
- End-to-end delay is $4 \times 3D = 12 D$
- With 16 bits, it is $16 \times 3D = 48 D$



Other Multi-bit Adders

- RCA is slow – $O(n)$ where n is number of bits in data
- Improving the speed of addition is critical to the performance for other arithmetic operations.
- There are other time optimized and more complex multi-bit adders
- *Carry Save Adder & Carry-lookahead Adder* are two such adders

Carry Look Ahead Adder (CLA)

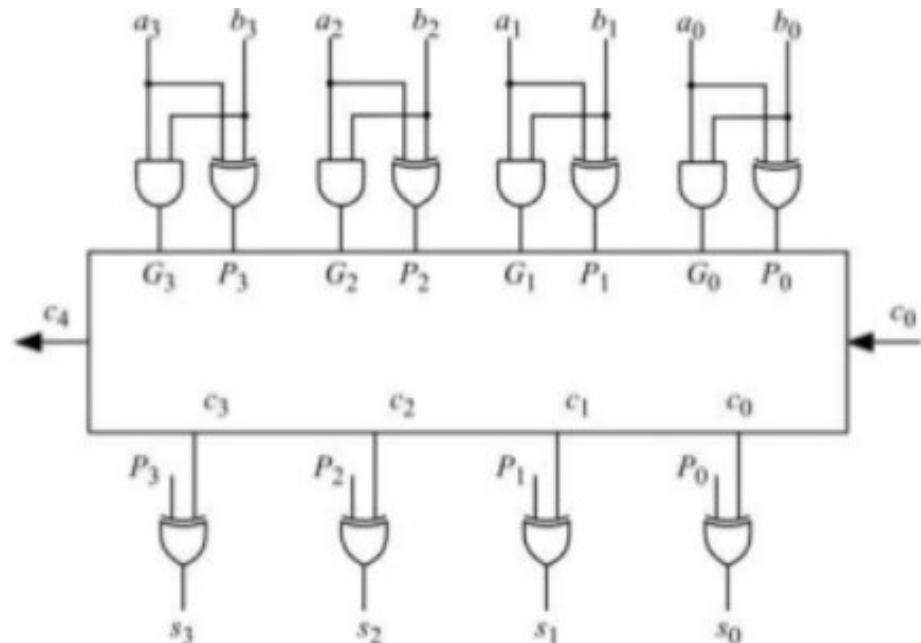
- To find the carry for a stage, we need not wait until the carry from the previous stage arrives (ripple in) in RCA
- We can build a more complex logic to compute all the carries in parallel
- Complexity appears in the form of FAN-IN!
- We can still build faster adder by keeping the FAN-IN to a moderate values

Generated and Propagated Carry

- If $x_i = y_i = 1$ - carry generated regardless of incoming carry - no additional information needed
- If $x_i y_i = 10$ or $x_i y_i = 01$ – then incoming carry is propagated
- Generated carry $G_i = x_i y_i$; Propagated carry $c_i P_i = c_i (x_i \oplus y_i)$
- Most important observation: Both G_i and P_i are functions of X_i and Y_i only!
- Carry output c_{i+1} is given the following express

$$x_i y_i + c_i (x_i \oplus y_i) = G_i + c_i P_i$$

4-bit Carry Look Ahead Adder



$$S_i = (a_i \oplus b_i) \oplus C_i$$

$$G_i = a_i b_i; P_i = (a_i \oplus b_i)$$

$$c_1 = G_0 + P_0 c_0$$

$$c_2 = G_1 + P_1 c_1$$

$$= G_1 + P_1 (G_0 + P_0 c_0)$$

$$= G_1 + P_1 G_0 + P_1 P_0 c_0$$

c_3 and c_4 can be expressed similarly

Delay: Sum = 4D; Carry c_4 = 3D; – how?

$$c_1 = G_0 + c_0 P_0,$$

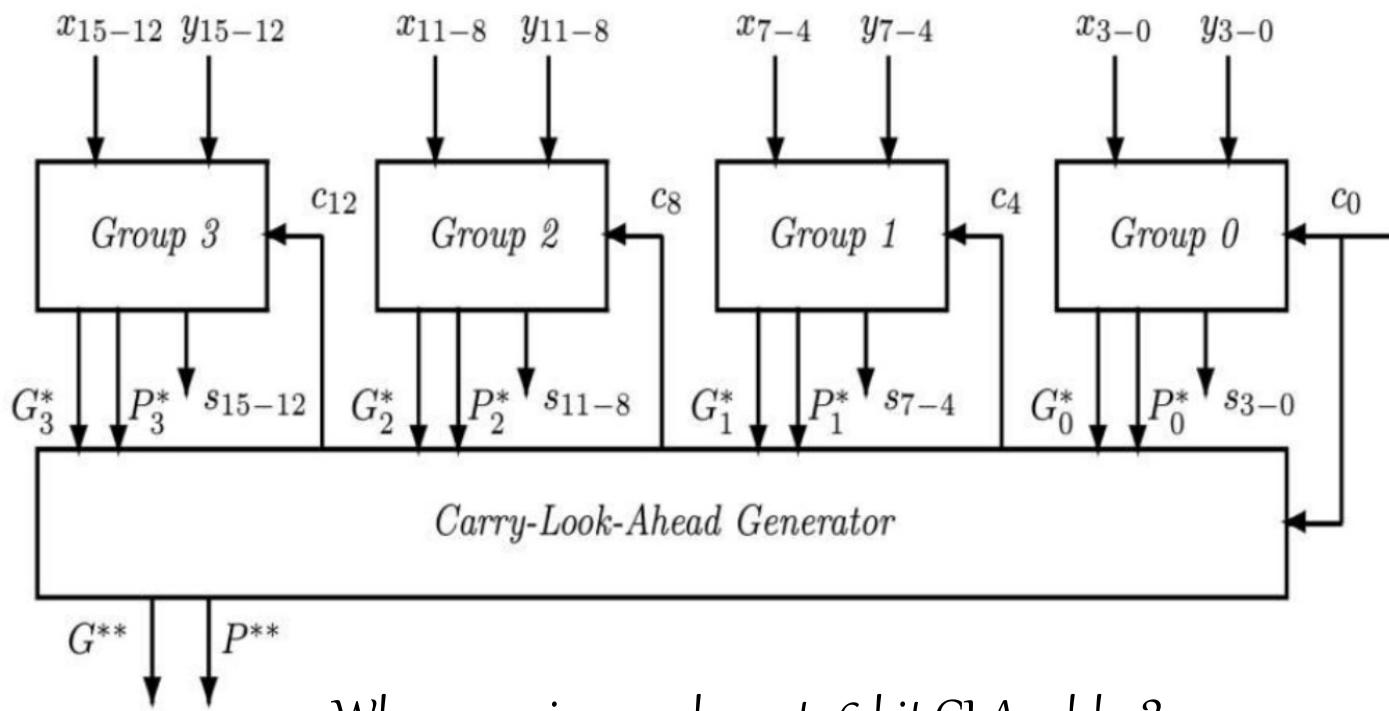
$$c_2 = G_1 + G_0 P_1 + c_0 P_0 P_1,$$

$$c_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2,$$

$$c_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3$$

$$C_4 = c_0 P_{\text{group0}} + G_{\text{group0}}$$

Multi-Group Multi-bit Carry Look Ahead ✓



Why grouping – why not 16 bit CLA adder?

- Delay Computation
- Assumption Co is available
- G_s and P_s are computed in parallel – 1D
- After 2D C_4 out
- After another 2D C_8 out, and so on.
- 10D for 16 bits
- Compare this delay with RCA delay (for 16 bits) - 48D

Carry output for each Group

$$c_4 = G_0^* + c_0 P_0^*,$$

$$c_8 = G_1^* + G_0^* P_1^* + c_0 P_0^* P_1^*,$$

$$c_{12} = G_2^* + G_1^* P_2^* + G_0^* P_1^* P_2^* + c_0 P_0^* P_1^* P_2^*$$

Similarly for the stage of 16 bits

$$C_{16} = c_0 P_{\text{stage0}} + G_{\text{stage0}}$$

$$c_1 = G_0 + c_0 P_0,$$

$$c_2 = G_1 + G_0 P_1 + c_0 P_0 P_1,$$

$$c_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2,$$

$$c_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3$$

P_{stage0} ?

G_{stage0} ?

Status Register – Carry Bit

- ❑ Note, when you add to “large” unsigned integer, the result could produce a carry and carry could be stored in status register
- ❑ Carry is special case of overflow – overflow with respect addition or subtraction of unsigned binary representation

Example

- <https://www.slideshare.net/dragonpradeep/carry-look-ahead-adder>

Signed Integers

- The most significant bit (MSB) is generally used as a sign bit
- When the sign bit changes due to addition, it is could be overflow
- If this bit is 0 then rest of the 31 bits represent a weighted positive integer else a negative integer

Sign	30	29		0
0/1	0	0	...	0

- 2s complement is another notation for signed integer representation

Ones Complement

- *Ones complement of 1 is 0 and ones complement of 0 is 1*
- *Ones complement of an integer is its bit-wise complement*
- Example:
 - 1's complement of 10001 is 01110
 - 1's complement of 111000 is 000111
 - 1's complement of 00000 is 11111
- Ones complement is part of 2's complement representation

Two's Complement Representation

- Two's complement of an integer is 1 added to 1's complement of it
- In Two's complement the MSB represents the sign (positive or negative)
- A positive integer is represented simply by its binary weighted notation
 - Example: 3 is 011 (note: the sign bit is 0 for positive)
- A negative integer is represented by converting its magnitude to 2's complement
 - Example: -3 is 101 (note the sign bit is 1 for negative)
 - 3 bit range is -3 to +3
 - Add +3 (011) to -3 (101)

Exercise

- 3 bit 2's complement encoding - list
- 4 bit 2's complement encoding – list

Lab Exercise

- Build an eight bit adder with two 4-bit CLA

Benefits of 2's Complement

- Computing 2's complement is simple
- Easier arithmetic operations
 - Subtraction is done as addition!
 - Adder is sufficient to do addition and subtraction
 - Example: $3 - 2 = 3 + (-2) = 011 + (-010) = 011 + (110) = 001$
 - Ignore the carry in the above addition

Addition/Subtraction

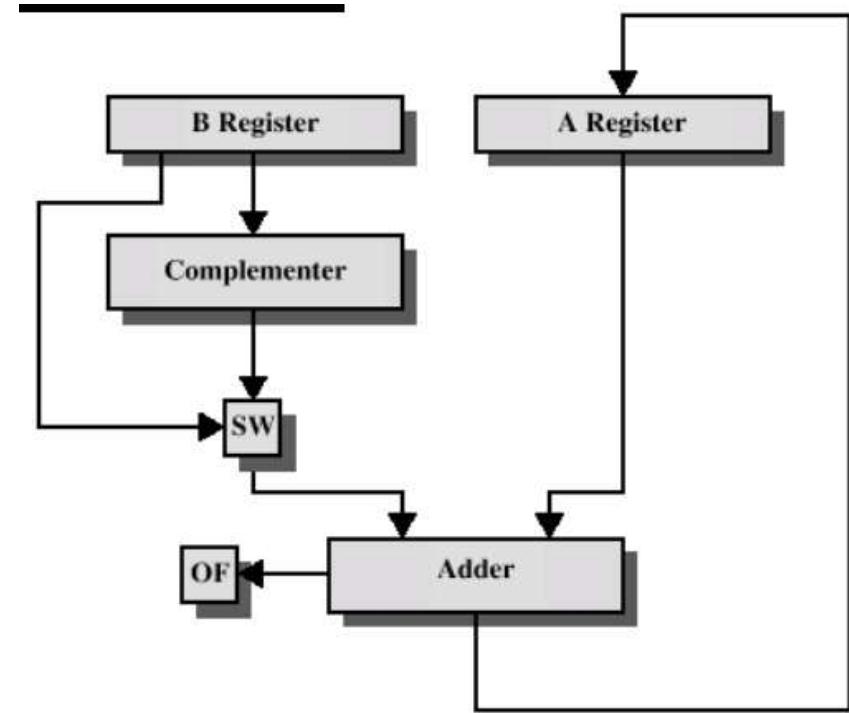
- Addition/subtraction of two integers A and B is symbolically represented as $A + B$ where A and B could be positive negative independently
- The following example summarizes all of those combinations with 2's complement (3 bit numbers)

Overflow Condition

- Example 1 (Positive, Positive): $010\ (2) + 001\ (1) = 011$
- Example 2 (P,P,O): $010\ (2) + 011\ (3) = 101$
- Example 3 (P, Negative): $010\ (2) + 111\ (-1) = 001$
- Example 4 (N, N, O): $100\ (-4) + 110\ (-2) = 010$
- Example 5 (N, N): $110\ (-2) + 111\ (-1) = 101\ (-3)$
- Overflow Conditions:
 1. Two positive numbers yielding a negative result.
 2. Two negative numbers yielding a positive result.

32/64 Bit adder/subtractor

- We can use CLA adder for both addition and subtraction using 2's complement representation for negative integers



OF = overflow bit

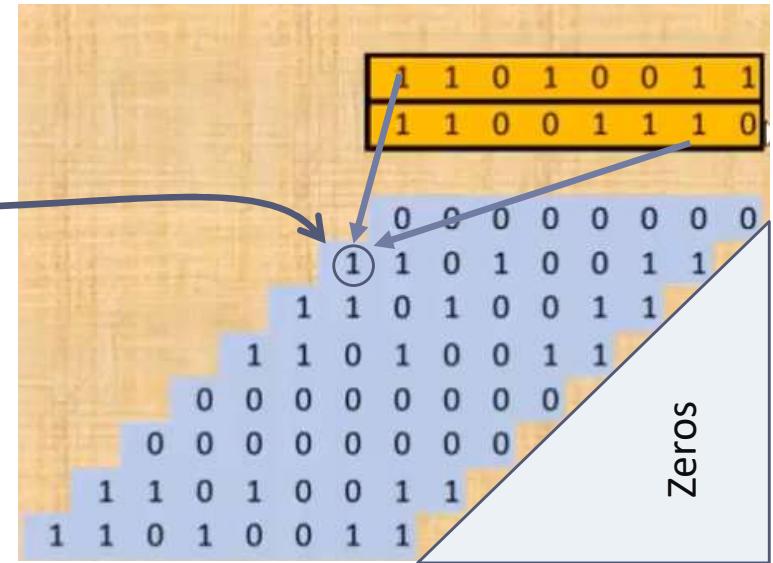
SW = Switch (select addition or subtraction)

Wallace Multiplier

- Motivation: Faster multiplication with parallel operations
- The Wallace tree has three steps:
 1. Multiply (that is – AND) each bit of one of the arguments, by each bit of the other, yielding n^2 *partial products* like conventional multiplication.
 2. Reduce the number of *partial products* to two layers of *full* and *half adders* with multiple iterations
 3. Finally when there are two reduced rows, add them by one of those *multi-group & multi-bit adder*

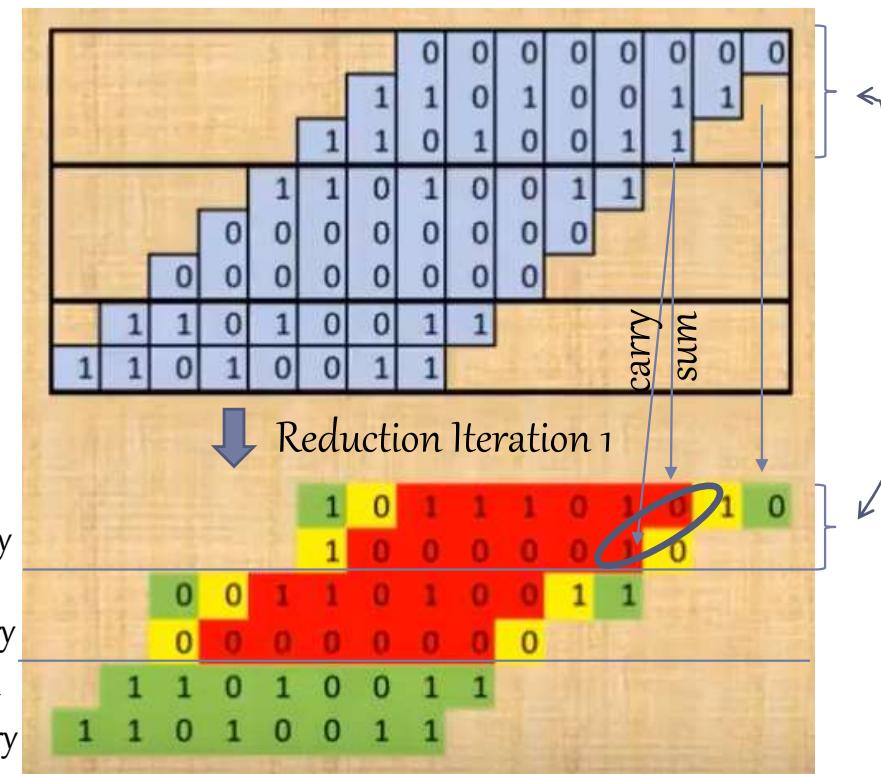
Example – Step 1

- Normal *long hand* multiplication
- Partial products (**bit by bit**) are produced by AND operation – $ox_1, ox_1, ox_0 \dots$ and so on.
- This multiplication is an AND operation
- All partial products are computed in parallel
- More h/w but improved time – D!
- The challenge is in adding these partial products
- All partial products are computed in parallel.
Thus, 1 gate delay to produce the result



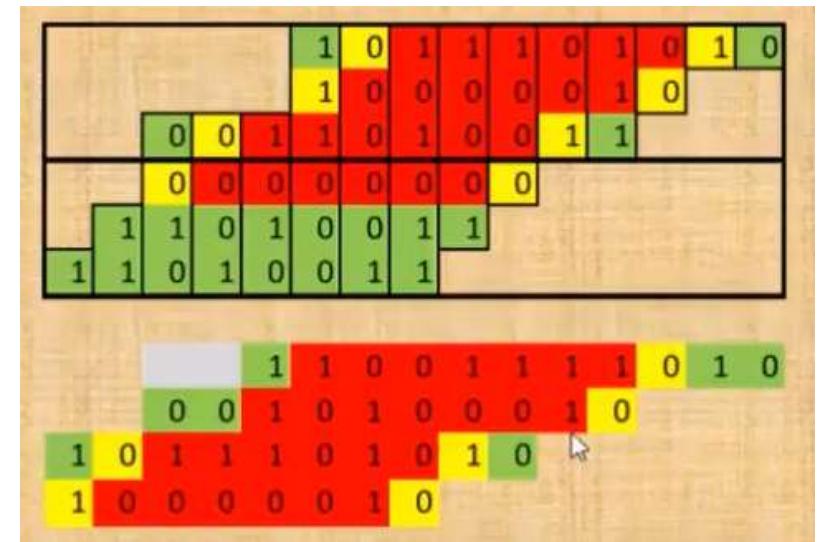
Example – Step 2 (Iterations) ✓

- Columns are added by grouping them 3 rows at a time (iteration)
- The result of this addition is 2 new rows replacing 3 old
- Red: full adder output
- Yellow: half-adder output
- Green: simple forwarding
- Time: Full Adder time of 3D!



Iteration 2

- Repeat the reduction process
- Now we are left with 2 sets of 3 rows
- The result is 2 sets of two rows
- Gray boxes indicate that summation bits have been moved down to the carry-out row



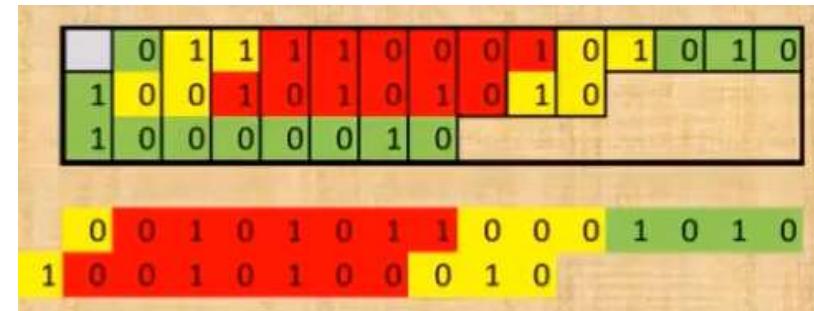
Iteration 3

- Repeat the process
- There is only one set of 3 rows, plus an extra row to carry down
- This set of 3 rows produces 2 new rows
- The resulting 3 rows are iterated to produce the final 2 rows (next slide)

1	0	1	1	1	0	1	0	0	1	1	1	1	0	1	0
1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1
0	1	1	1	1	0	0	0	1	0	1	0	1	0	1	0
1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	0	0	0	0	0	0	1	0	1	0	0	1	0	1	0

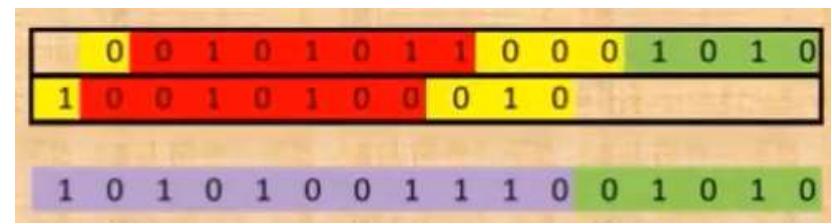
Iteration - 4

- Repeat the process one last time
- Remaining three rows become 2 rows
- In all, these 4 iterations cost 4 full adder delay
- The five least significant bits (LSB) are already computed



Step 3 (Final Addition)

- Final result is computed by adding the last two rows
- In this example, the last 4 LSBs do not need to be added
- The savings from already having 5 bits offsets the delay due to iterations
- Overall, Wallace Tree Multiplication takes nearly the same time as RCA!!



Time Delay

- Step1's time 1D (AND)
- Step2's total time is 4 (iterations) x Full Adder delay – $4 \times 3D$
 - $8 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2$
- Step3's time depends on the adder used – dominates the overall delay
(48D assuming RCA)
- If we use CLA, then step 3 delay is 10D! (Close to 16-bit addition)
- Thus overall delay of $D + 12D + 10D = 23D$

Wallace Multiplier

- Wikipedia
- <https://www.youtube.com/watch?v=4-lPGPoggo>

Lab Exercises for rest of the course

- Week 1
 - ARMSIM Ex 1,2, and 3
- Week 2
 - ARMSIM Ex 4 and 5
- Week 3
 - ARMSIM Ex 6
- Week 4, 5, and 6
 - ARMSIM Ex 14, 15, and 16
- Week 7
 - Verilog Ex 19,20, and 21
 - Verilog Ex 22 is optional
- Week 8
 - ARMSIM Ex 17, 18

Tentative Lab Exam schedule: on 16th of April 9 am to 12:30 pm

Booth's Multiplication

- Decimal multiplication provides some short cuts like the ones below:
 - $234 \times 99 = 234 \times 100 - 234 \times 1$
 - $234 \times 101 = 234 \times 100 + 234 \times 1$
- Booth's algorithm is based on similar approach for binary multiplication

Booth's Algorithm - Principle

- It is a generalization of the above short cut, works with sequence of 1's.
Multiplication with sequence of 0's is already trivially optimized
- Example:

$$001101 \times \textcolor{red}{01110} = 001101 \times \textcolor{blue}{10000} - 001101 \times 000\textcolor{red}{10}$$

$$001101 \times (14) = 001101 \times (16) - 001101 \times (2)$$

The subtraction is done as 2's complement addition

Arithmetic right shift used

6-bit Example ✓

Multiplier (MR): 00110 (14)

Multiplicand (MD): 001101 (13)

$2s$ Complement of MD: 110011

$$14 \times 13 = 182$$

Note: Right shift is Arithmetic

$$001101 \times 00\textcolor{red}{1110}$$

$$= 001101 \times 0\textcolor{blue}{1000}0 - 001101 \times 000\textcolor{red}{010}$$

$$\begin{array}{r} 001101 \end{array} \Xi \begin{array}{r} \textcolor{blue}{0} \textcolor{blue}{1} \textcolor{black}{0} \textcolor{black}{0} \textcolor{red}{1} \textcolor{black}{0} \\ \textbf{A R R S R} \end{array}$$

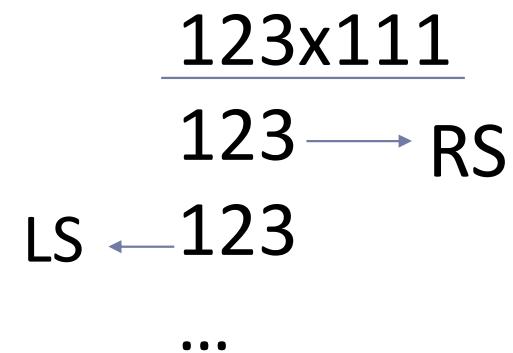
No	Operation	Accumulator								Q1	Q0
		1	0	0	0	0	0	0	0		
1	Initialize									B2	0
2	Right Shift	0	0	0	0	0	0	0	0		
3	Subtract	1	1	0	0	1	1			1	0
	Partial Result	1	1	0	0	1	1	0			
4	Right Shift	1	1	1	0	0	1	1	0		
5	Right Shift	1	1	1	1	0	0	1	1	1	1
6	Right Shift	1	1	1	1	1	0	0	1	1	1
7	Add	0	0	1	1	0	1			0	1
		0	0	1	0	1	1	0	1	1	0
				128	32	16		4	2		

Generalization

BXA

$$b_{3^1} b_{3^0} \dots b_0 X a_{3^1} a_{3^0} \dots a_0$$

$a_i a_{i-1}$	Operation
0	In the middle of string 0. Shift partial product to right (<i>why right shift?</i>)
10	Beginning of string of 1. Subtract B from partial product. Shift partial product to right.
11	In the middle of string 1. Simple shift right of partial product
"01"	End of string of 1. Add B to partial product. Shift partial product to right.



This is equal to
RS, ADD, RS

Example

Assume $n = 7$ bits available. Multiply $B = 22 = (0010110)_2$ by $A = -34 = -(0100010)_2$. First represent both operands and their negation

[Calculator Example](#)

22:	0010110,	-22:	1101010	Multiplicand (B)
34:	0100010,	-34:	1011110	Multiplier (A)

Then carry out the multiplication in the hardware:

Initialization:

- Q with A (Multiplier) and q_{i-1} to 0
- AC with Zero
- B (with Multiplicand)

$$B \times A = -\overline{11110100010100} = -00001011101100 = -748_{10}$$

$q_i q_{i-1}$	Action			q_i	q_{i-1}
		[AC]	[Q]		
1 00	right shift	0000000	0101111	0	
2 10	-B	+ 1101010	1101010	0101111	0
.	.	1101010	0010111	0	
.	right shift	1110101	0010111	1	
.	right shift	1111010	1001011	1	
.	right shift	1111101	0100101	1	
11	right shift	1111110	1010010	1	
01	+B	+ 0010110	0010100	1010010	1
.	.	0010110	0101001	0	
.	right shift	0001010	0101001	0	
10	-B	+ 1101010	1110100	0101001	0
	right shift	1101010	0010100	1	

Do the
following
multiplicati
on using
Booth's
algorithm:
00110100*
01111110

			Multiplier	01110110								118										
			Multiplicand	01010101								85										
			2s Comp	10101011																		
			Product	118x85=								10030										
Step	Qi	Qi-1	Action	AC								Q								Qi	Qi-1	
			Initialize	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0	0	
1	0	0	Right Shift	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0
2	1	0	Subtract Multiplicand	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
				1	0	1	0	1	0	1	1	0	0	1	1	1	1	0	1	1	1	0
			Right Shift	1	1	0	1	0	1	0	1	1	0	0	0	1	1	1	1	0	1	1
3	1	1	Right Shift	1	1	1	0	1	0	1	0	1	1	1	0	0	0	1	1	1	0	1
4	0	1	Add Multiplicand	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1
				0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	1
			Right Shift	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0
5	1	0	Subtract Multiplicand	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
				1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	1	1	0
			Right Shift	1	1	1	0	0	1	0	1	1	0	1	1	1	1	0	0	1	1	1
6	1	1	Right Shift	1	1	1	1	0	0	1	0	0	1	0	1	0	1	1	1	0	0	1
7	1	1	Right Shift	1	1	1	1	1	0	0	0	1	0	1	0	1	1	1	0	0	0	1
			Add Multiplicand	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
				0	1	0	0	1	1	1	0	0	1	0	1	1	1	0	0	1	1	1
			Right Shift/Result	0	0	1	0	0	1	1	1	1	0	0	0	1	0	1	1	1	1	0

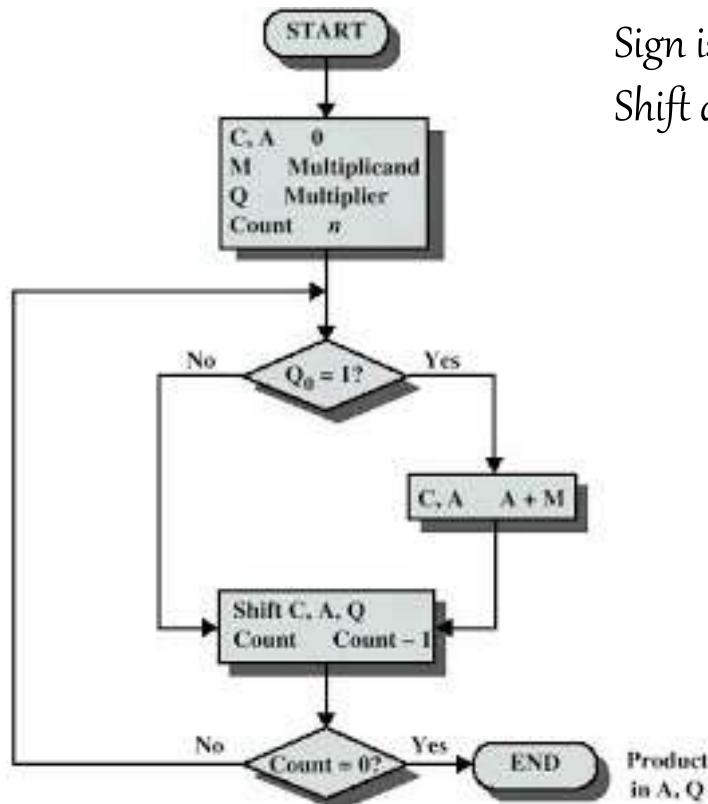
Booth's versus Wallace

- Wallace exploits parallel operations – requires complex circuit
- Booth's simpler (no parallel operation), but requires 8-bit adder (not 16 bit adder)
- Full analysis is not done here!

Booth's Multiplication Algorithm

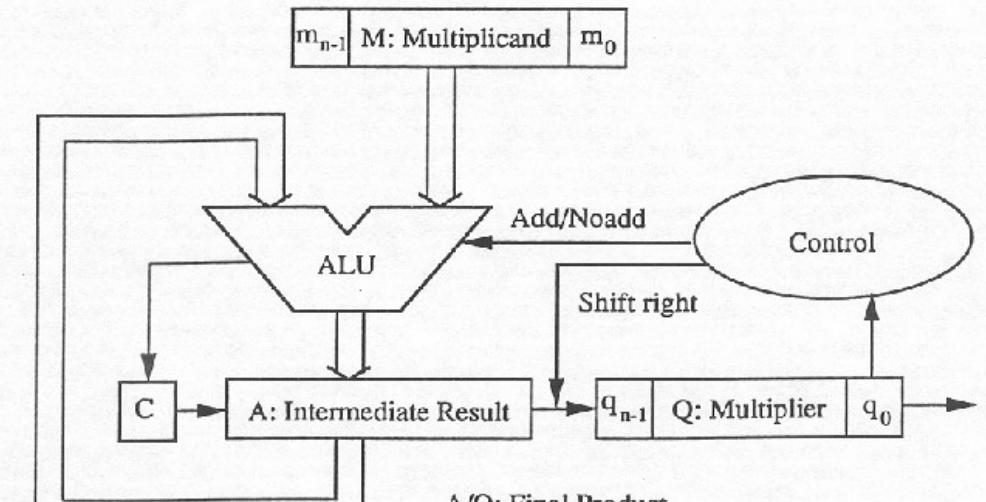
- http://fourier.eng.hmc.edu/e85_0ld/lectures/arithmetic.html/node10.html
- <https://nobelsharanyan.wordpress.com/2015/05/25/booths-algorithm-multiplication-of-two-unsigned-numbers-and-signed-numbers/>

No Frill Multiplication Algorithm



Sign is ignored
Shift does not loose LSB!

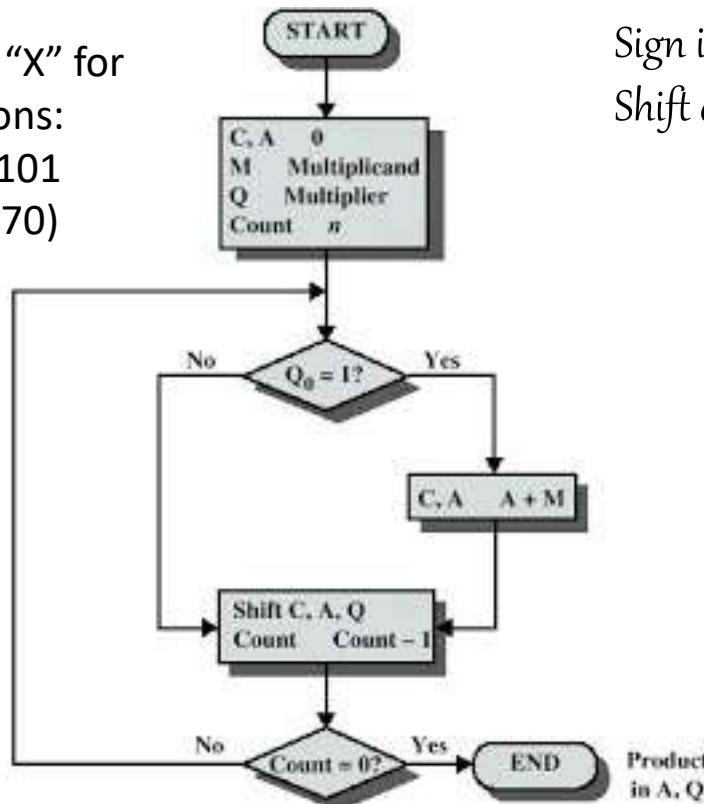
A contains the MSBs of the result
Q at the end contains the LSBs of the result
C contains the carry
Do timing analysis with RCA and CLA



Hardware for Multiplication

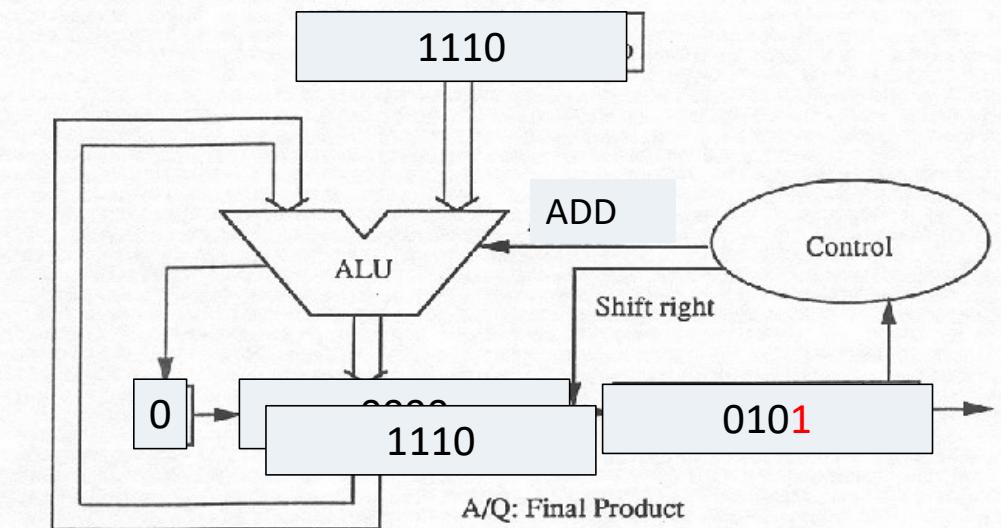
No Frill Multiplication Algorithm

Lets do this “X” for
2 iterations:
 1110×0101
($14 \times 5 = 70$)



Sign is ignored
Shift does not loose LSB!

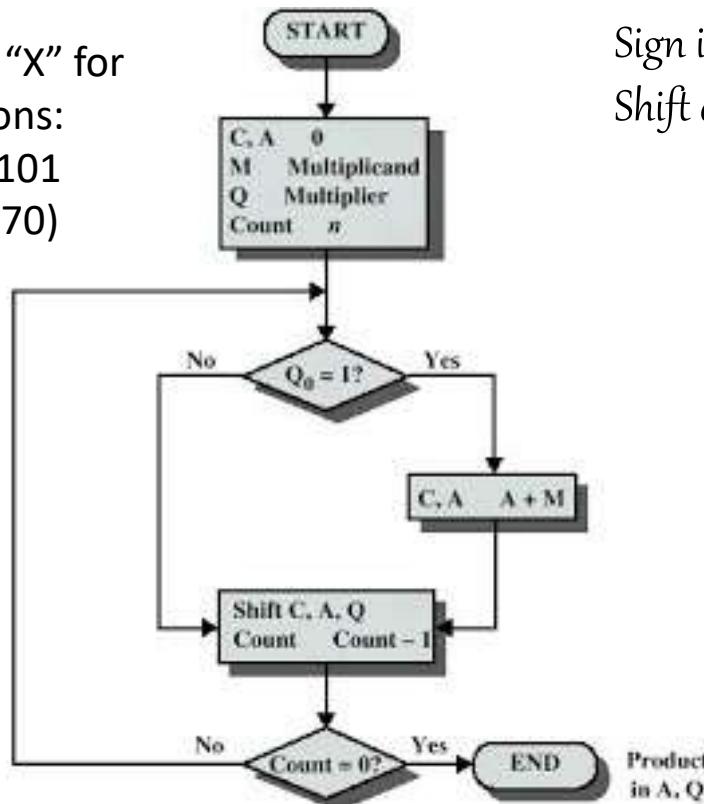
A contains the MSBs of the result
Q at the end contains the LSBs of the result
C contains the carry
Do timing analysis with RCA and CLA



Hardware for Multiplication

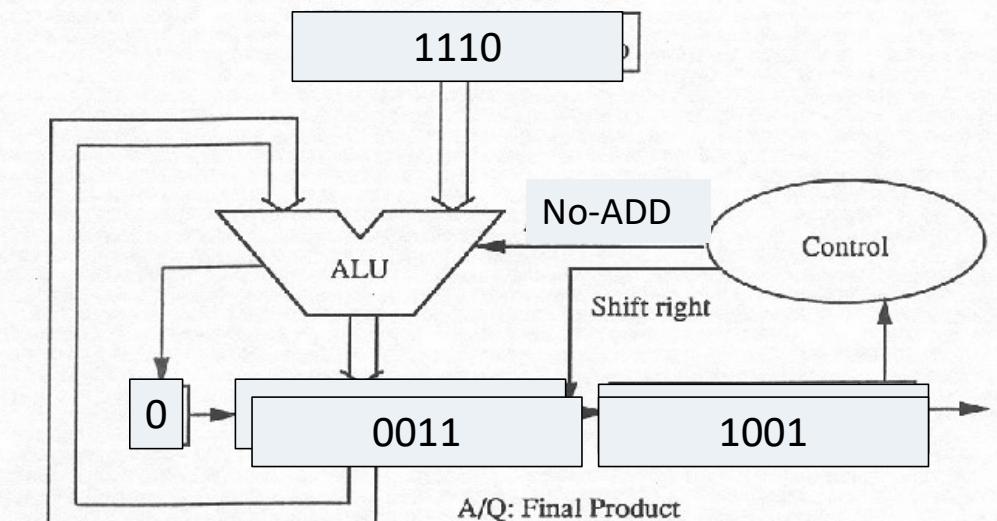
No Frill Multiplication Algorithm

Lets do this “X” for
2 iterations:
 1110×0101
($14 \times 5 = 70$)



Sign is ignored
Shift does not loose LSB!

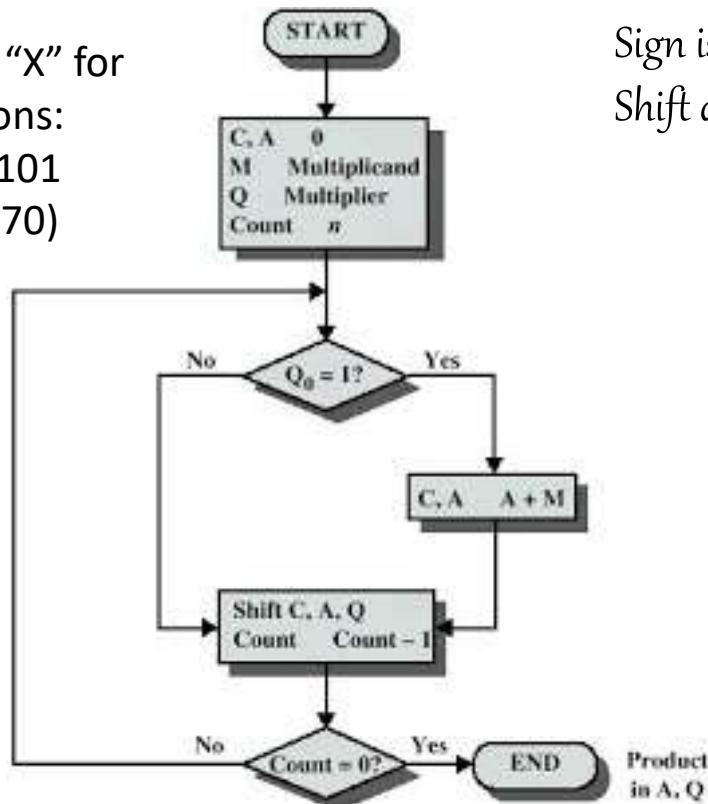
A contains the MSBs of the result
Q at the end contains the LSBs of the result
C contains the carry
Do timing analysis with RCA and CLA



Hardware for Multiplication

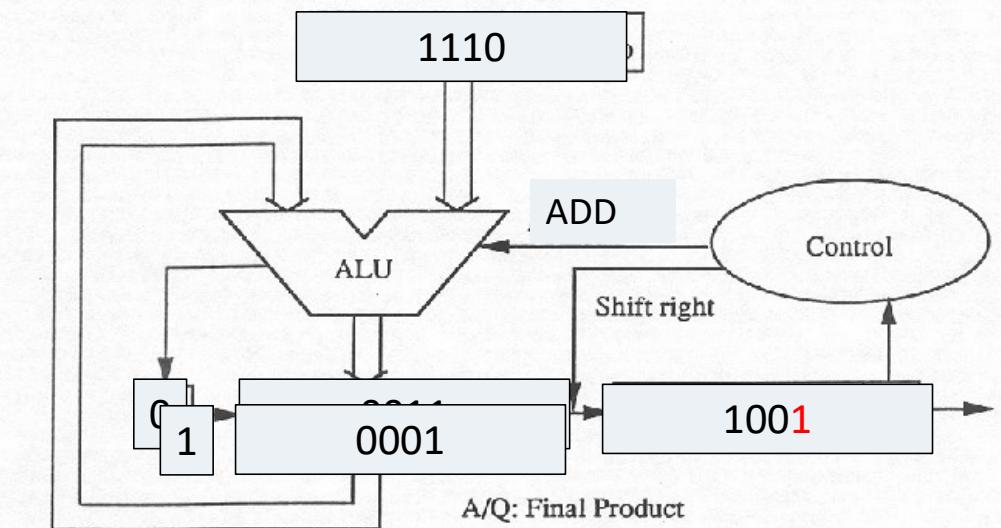
No Frill Multiplication Algorithm

Lets do this “X” for
2 iterations:
 1110×0101
($14 \times 5 = 70$)



Sign is ignored
Shift does not loose LSB!

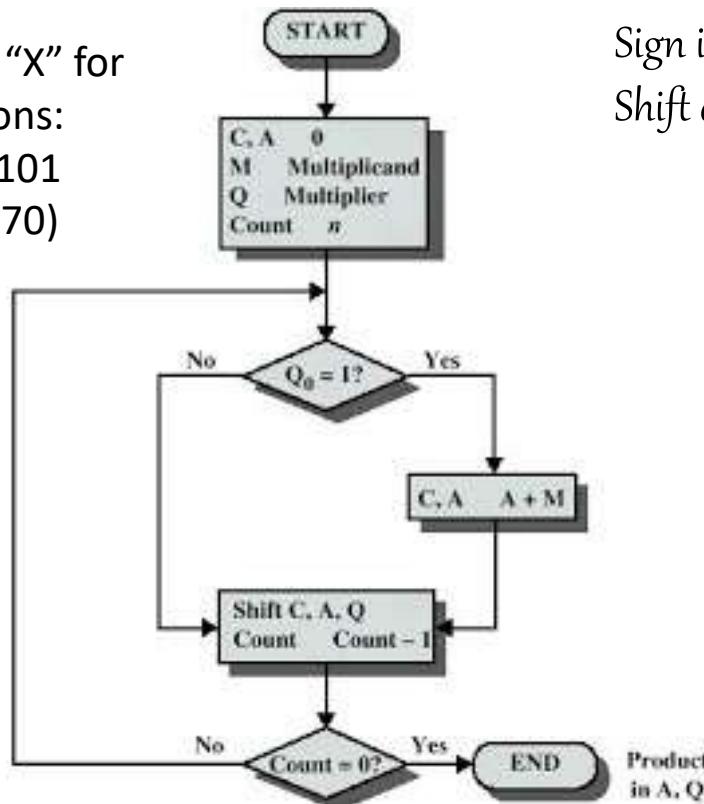
A contains the MSBs of the result
Q at the end contains the LSBs of the result
C contains the carry
Do timing analysis with RCA and CLA



Hardware for Multiplication

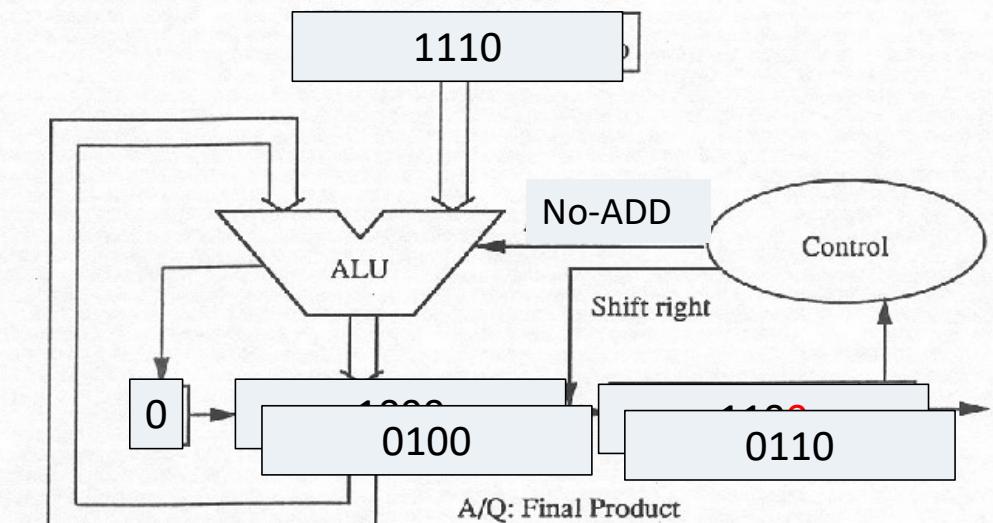
No Frill Multiplication Algorithm

Lets do this "X" for
2 iterations:
 1110×0101
($14 \times 5 = 70$)



Sign is ignored
Shift does not loose LSB!

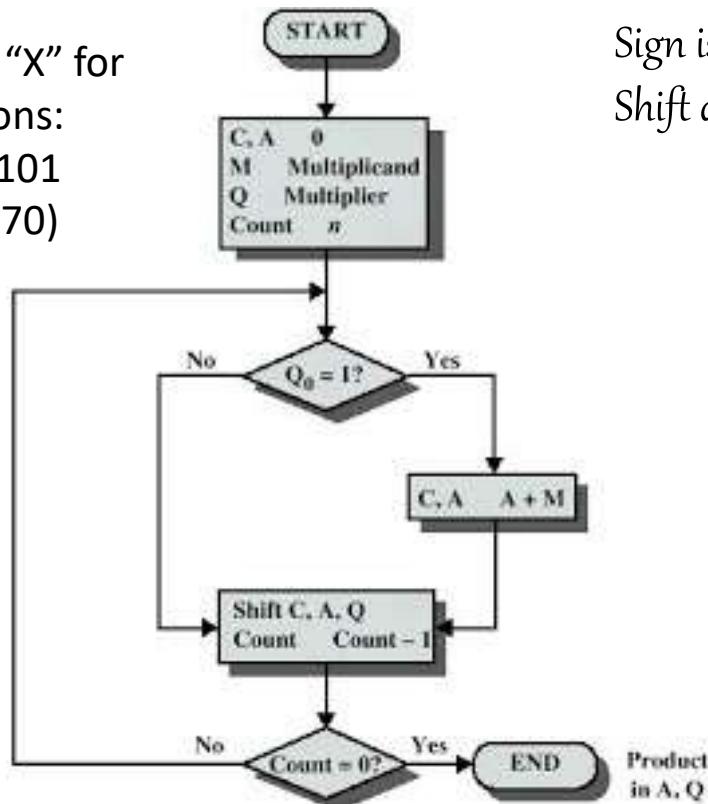
A contains the MSBs of the result
Q at the end contains the LSBs of the result
C contains the carry
Do timing analysis with RCA and CLA



Hardware for Multiplication

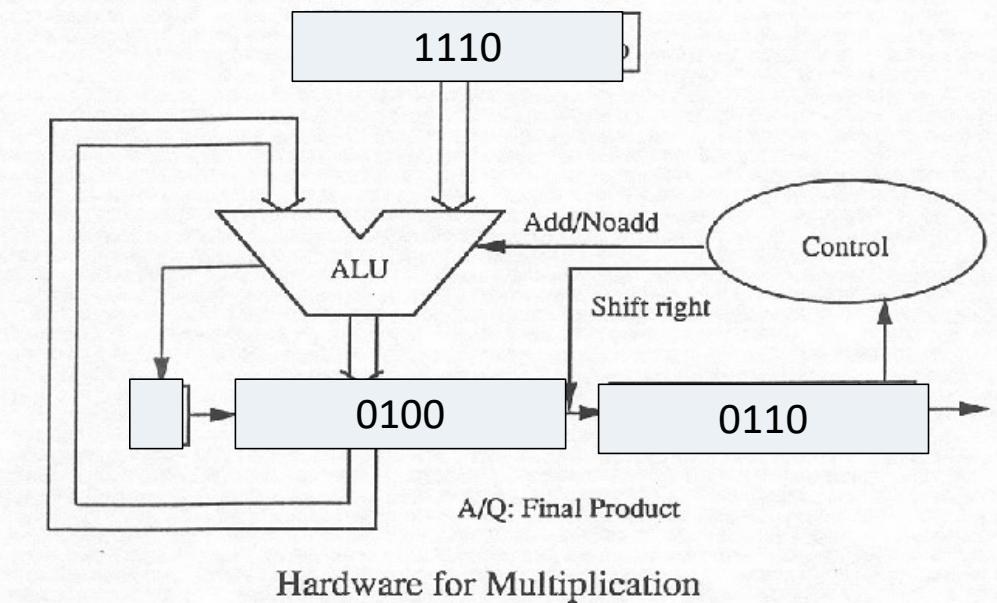
No Frill Multiplication Algorithm

Lets do this “X” for
2 iterations:
 1110×0101
($14 \times 5 = 70$)



Sign is ignored
Shift does not loose LSB!

A contains the MSBs of the result
Q at the end contains the LSBs of the result
C contains the carry
Do timing analysis with RCA and CLA



Division (Restoring)

Algorithm for hardware division (restoring)

Do n times:

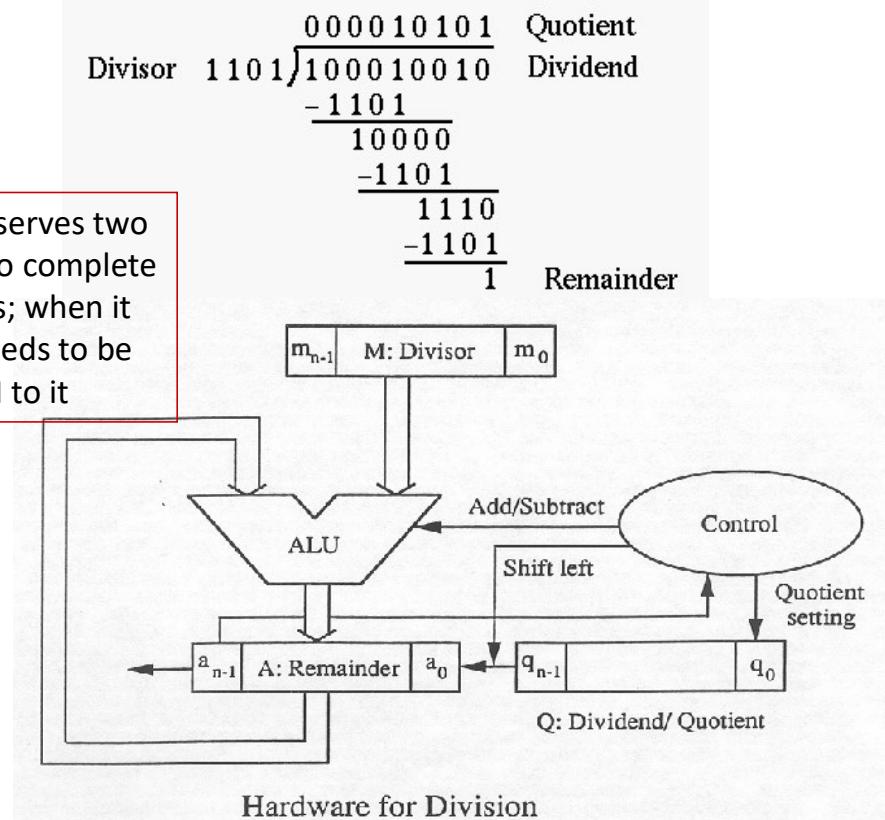
{ left shift A and Q by 1 bit

$$A \leftarrow A - M;$$

$A \leftarrow A - M$ This operation serves two purposes 1) to test and 2) to complete a step in certain iterations; when it serves only as a test, A needs to be restored by adding M to it

if $A < 0$ ($a_{n-1} = 1$), then $q_0 \leftarrow 0$, $A \leftarrow A + M$ (restore)

else $q_0 \leftarrow 1$



Non-Restoring

- In the previous algorithm, we perform either 2 or 3 operations in each iteration.
- This can be replaced by non-restoring algorithm where in ALL iterations we perform only 2 operations (shift with either an addition or a subtraction)

Do n times:

{ left shift A and Q by 1 bit

Subtract
Subtract

if (previous $A \geq 0$) then $A \leftarrow A - M$

else $A \leftarrow A + M$; **Restore**

if (current $A \geq 0$) then $q_0 \leftarrow 1$

else $q_0 \leftarrow 0$

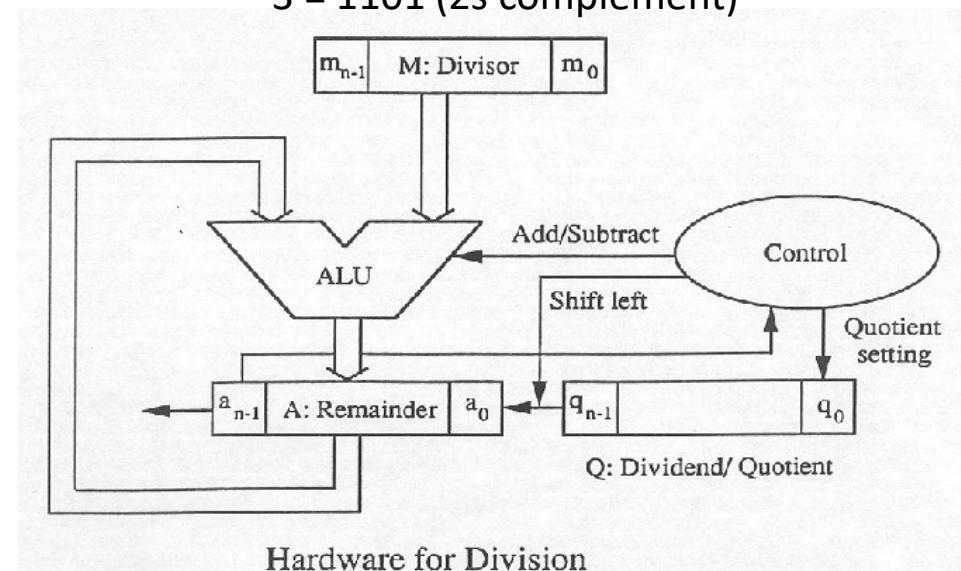
Division (Non-Restoring)

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001	000.	
$A = [A] - [M]$	+	1101		
$A < 0$		1110	0000	
left shift A/Q		1100	000.	
$A = [A] + [M]$	+	0011		
$A < 0$		1111	0000	
left shift A/Q		1110	000.	
$A = [A] + [M]$	+	0011		
$A > 0$		0001	0001	
left shift A/Q		0010	001.	
$A = [A] - [M]$	+	1101		
$A < 0$		1111	0010	
$A = [A] + [M]$	+	0011		
		0010	0010	

Divisor = 0011 (3)

Dividend = 1000 (8)

-3 = 1101 (2s complement)



Floating Point Representation

- There are standards (IEEE 754) for representing floating point numbers
- Reference:
<https://www.slideshare.net/rituranjanshrivastwa/quick-tutorial-on-ieee-754-floating-point-representation>

32-Bit Floating Point Representation



To be represented in this format, a number should be in the following normalized form.
 $(+ \text{ or } -) 1.\text{(mantissa)} \times 2^{\text{(exponent)}}$

\Rightarrow This means

$$\begin{aligned}\Rightarrow 4.6 / 2 &= 2.3 \\ \Rightarrow 2.3 / 2 &= 1.15\end{aligned}$$

\Rightarrow Hence we get the normalized form and we can write
 $+4.6 \Leftrightarrow 1.15 \times 2^2$

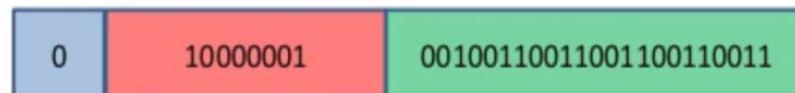
Let us convert the 0.15 to binary

$$\begin{aligned}\Rightarrow 0.15 \times 2 &= 0.3 && 0 \\ \Rightarrow 0.3 \times 2 &= 0.6 && 0 \\ \Rightarrow 0.6 \times 2 &= 1.2 && 1 \quad (\text{i}) \\ \Rightarrow 0.2 \times 2 &= 0.4 && 0 \\ \Rightarrow 0.4 \times 2 &= 0.8 && 0 \\ \Rightarrow 0.8 \times 2 &= 1.6 && 1 \quad (\text{ii})\end{aligned}$$

\Rightarrow Now the value from (i) till (ii) will continue to recur and we will keep recurring it till 23 bits are filled.

\Rightarrow Thus the bits obtained are 00100110011001100110011

Hence the bit pattern in the 32 bit format are



$\Leftrightarrow (4093333)_16$

Example

You need to do just the reverse of the above which is very simple.
For example:

Given Binary representation: 11000001101111110.....0

Thus we will break it into three parts as:



We clearly see that the number is negative and the power is $131 - 127 = 4$

Mantissa is: $2^{-1} \times 0 + 2^{-2} \times 1 + 2^{-3} \times 1 + 2^{-4} \times 1 + 2^{-5} \times 1 + 2^{-6} \times 1 + 2^{-7} \times 1 = 0.4921875$

⇒ The number is -1.4921875×2^4 [note the '1' is added before the 0 in the normal form]

⇒ Which is equal to **-23.875**

ANS: -23.875

Quiz

- Average is close to 12
- Grading is done with slotted range
- Answers and scores will be posted on my cloud storage
- You can prepare your solution in additional sheets but answer must be provided in question paper only
- Write concisely and legibly. Enough space provided to restrict open ended writing.
- Inconsistent and unnecessary statements will be penalized
- Duplicate answers will be penalized.

Common mistakes

- Process state diagram (Q8)
- Tertiary logic – Bus
- Resolution
- Process, Processor, and Scheduler
- Big Endian and Small Endian – Address is must (left to right, top to down will not do it)

FP Representation Double Precision

IEEE Floating Point Representation

s	exponent	mantissa
1 bit	8 bits	23 bits

IEEE Double Precision Floating Point Representation

s	exponent	mantissa
1 bit	11 bits	52 bits

Representation of Special Numbers

Single-Precision	Exponent = 8	Fraction = 23	Value
Normalized Number	1 to 254	Anything	$\pm (1.F)_2 \times 2^{E-127}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-126}$
Zero	0	0	± 0
Infinity	255	0	$\pm \infty$
NaN	255	nonzero	NaN

Double-Precision	Exponent = 11	Fraction = 52	Value
Normalized Number	1 to 2046	Anything	$\pm (1.F)_2 \times 2^{E-1023}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-1022}$
Zero	0	0	± 0
Infinity	2047	0	$\pm \infty$
NaN	2047	nonzero	NaN

<https://www.slideshare.net/prochwanig5/o6-floating-point>

Floating Point Addition/Subtraction

1. Check for zeros
2. Align *significands* (by adjusting exponents)
3. Add or subtract significands
4. Normalize result

- ❖ Consider Adding (Single-Precision Floating-Point):
+ 1.111001000000000000000010₂ × 2⁴
+ 1.1000000000000110000101₂ × 2²
 - ❖ Cannot add significands ... Why?
 - ✧ Because **exponents are not equal**
 - ❖ How to make exponents equal?
 - ✧ **Shift the significand of the lesser exponent right**
 - ✧ Difference between the two exponents = 4 – 2 = 2
 - ✧ So, **shift right** second number by 2 bits and increment exponent
- $$\begin{aligned} & 1.1000000000000110000101_2 \times 2^2 \\ & = 0.0110000000000001100001\ 01_2 \times 2^4 \end{aligned}$$

<https://www.slideshare.net/prochwanig5/o6-floating-point>

Floating Point Multiplication/Division

1. Check for zero
 2. Add/subtract exponents
 3. Multiply/divide significands
 4. Normalize
 5. Round
 6. All intermediate results should be in double length storage
- Source: <https://www.slideshare.net/prochwanig5/06-floating-point>

❖ Consider multiplying:

$$\begin{array}{r} -1.110\ 1000\ 0100\ 0000\ 1010\ 0001_2 \times 2^{-4} \\ \times\ 1.100\ 0000\ 0001\ 0000\ 0000\ 0000_2 \times 2^{-2} \end{array}$$

❖ Unlike addition, we **add the exponents** of the operands

$$\diamond \text{Result exponent value} = (-4) + (-2) = -6$$

❖ Using the biased representation: $E_Z = E_X + E_Y - Bias$

$$\diamond E_X = (-4) + 127 = 123 \text{ (*Bias = 127* for single precision)}$$

$$\diamond E_Y = (-2) + 127 = 125$$

$$\diamond E_Z = 123 + 125 - 127 = 121 \text{ (*value = -6*)}$$

❖ Sign bit of product can be computed independently

❖ Sign bit of product = $\text{Sign}_X \text{ XOR } \text{Sign}_Y = 1$ (**negative**)

Other Operations

- Set Operations; with bit map and logical operations
- Modulo Operation; https://en.wikipedia.org/wiki/Modular_arithmetic
- Matrix Operations
- Vector Operations
- Convolution
- ...

Arithmetic Operations of Information Security

- Information security uses data which are much wider (128, 256, 512, 1/2/4/8 K in size)
- This necessitates arithmetic operations with large data size
- How do we handle arithmetic operations of this magnitude with 32/64 bit processors?
- Example: Simple operation of adding two 256 bit numbers
- Read two binary strings of equal size (128 bits) from a file, add them and append the result to the same input file.

Processor Design – A case study

Instruction Set Architecture (ISA)

- ISA is the low-level software interface to the machine
 - Assembly language of the machine
 - Must translate any programming language into this language
- Examples of ISA: IA-32 (Intel), MIPS, SPARC, Alpha, PARISC, PowerPC, ...
- ISA is broadly classified into 2 types: RISC and CISC

Software

ISA

Hardware

Another Related Classification of ISA

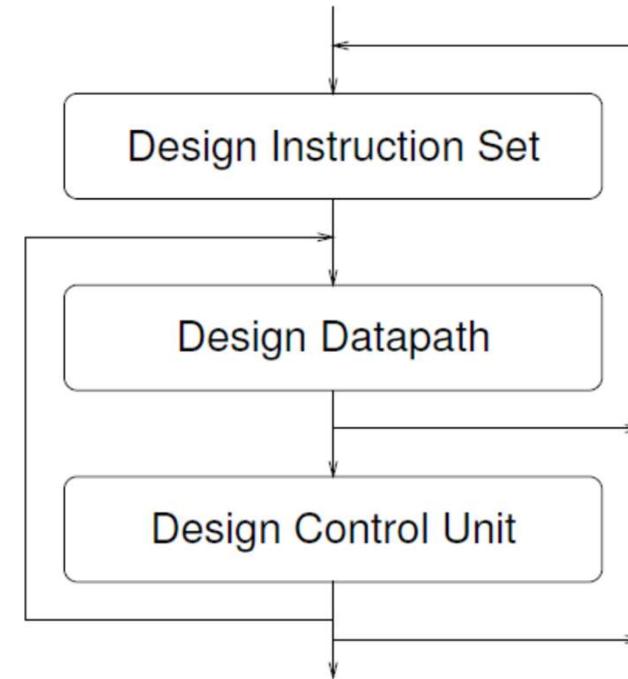
- Register-Register
- Register-Memory

Introduction

- So far we have seen various aspects of a computing system
 - 1. Pre-requisite material
 - 2. A simple hypothetical processor (SAP)
 - 3. Logic design for various operations
 - 4. Instruction Set Architecture (ISA) of a real processor - ARM
- In this module we are going study a processor design
 - Some aspects of this module is already discussed with SAP

Processor Design

- ❑ Processor design is an iterative procedure with three steps.
- ❑ We touched upon these steps with SAP



CISC (Complex Instruction Set Computer)

- CISC family has abundant instns, addressing modes, instrn formats and instrn. sizes.
- The control is micro-programmed with different instructions executing within different no: of cycles. ∴ The control units are complex since they have to deal with large no. of opcodes, addressing modes and formats.
- HLL Support in CISC: Because of large instruction set, there is a large choice for the compiler of any high level language. ∴ designing the optimizing stage of CISC compiler is very difficult.

RISC (Reduced Instruction Set Compute)

- RISC advantages: Each instruction is almost executed at the same pace. This allows instructions to from memory to CPU in a constant stream.
- RISC properties:
 - Single cycle execution of all (or majority of) instructions.
 - Fixed length of instructions.
 - Small no: of instructions (128)
 - Small no: of instruction formats (4)
 - Small no: of addressing modes(4)
 - Memory access by **load, store** only. Rest of the operations are reg-to-reg in CPU.
 - Hard-wired control unit as opposed to microprogrammed
 - Reg-file size in CPU is about 32, larger than in CISC.

Register-Register Instruction Set

- RISC Architecture is dominated by Register-Register instruction type.
- Only limited special instruction types are allowed to access memory:
 - LOAD Memory, Register
 - STORE Register, Memory
- CISC Architecture is dominated by Register-Memory instruction types

Comparison of Reg-Reg with Reg-Memory

Type	Advantage	Disadvantage
Register-register(0,3)	<ol style="list-style-type: none">1. Simple encoding2. <i>fixed-length instruction encoding.</i>3. Simple code generation model.4. <i>Instruction take similar numbers of clocks to execute</i>	<ol style="list-style-type: none">1. <i>Higher instruction count than architectures with memory references in instructions.</i>2. More instructions , lower instruction density leads to larger programs.
Register-memory(1,2)	<ol style="list-style-type: none">1. <i>Data can be accessed without a separate load instruction</i>2. <i>format tends to be easy to encode and yields good density</i>	<ol style="list-style-type: none">1. Encoding a register number and a memory address in each instruction may restrict the number of registers.2. <i>Clocks per instruction vary by operand location.</i>

Yet Another Hypothetical Processor

- We will design a simple hypothetical processor for this module
- This processor is more complex and more powerful than SAP
- Reference: Lecture Notes of Prof. Dan Connors
dconnors@colostate.edu Lecture Notes

Processor Design Parameters

- This is a 16-bit RISC, Reg-Memory processor
- Number of instructions 11
- 4-bit Opcode 12 bits for RAM address
- Special purpose registers 8

Register Set – Iteration 1

1. Program Counter
2. MAR
3. Accumulator
4. Instruction Register
5. Opcode Register
6. Input Register
7. Output Register

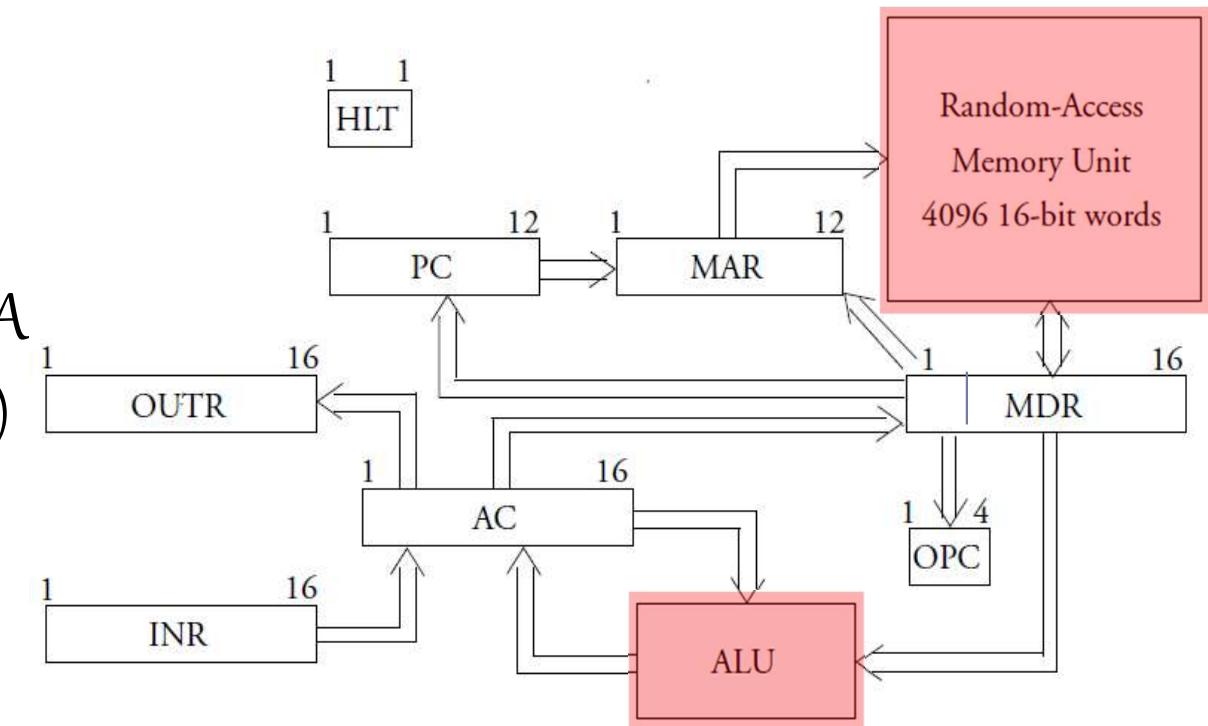
Instruction Set

	<i>Opcode</i>	<i>Binary</i>	<i>Description</i>
Arithmetic Logic	ADD	0000	Add memory word to AC
	AND	0001	AND memory word to AC
	CLA	0010	Clear (set to zero) the accumulator
	CMA	0011	Complement AC
	CIL	0100	Circulate AC left
Memory	LDA	0101	Load memory word into AC
	STA	0110	Store AC into memory word
Jump	JZ	0111	Jump to address if AC zero
I/O	IN	1000	Load INR into AC
	OUT	1001	Store AC into OUTR
Halt	HLT	1010	Halt computer

- This classification helps in data path design.
- Data path is the same for each instruction in every group.
- Data path varies across groups

Registers – Iteration 2

- *accumulator*(AC)
- *program counter*(PC)
- *opcode*(OPC)
- *memory address register*(MA)
- *memory data register*(MDR)
- *input register*(INR)
- *output register*(OUTR)
- *halt register*(HLT)



What Architecture is this - Harvard/Princeton (Von Neumann)?

Expressive Power of ISA

- Subtraction is possible with ADD, CMA and 2's complement – how?
- Rotate right can be realized using CIL Rotate Right 15 is equal to CIL 1
- AND & CMA are enough to express Boolean expressions of any complexity – how?

Timing and Control

- We need to design circuits controlling the combining and movement of data.
- The following Assignment notation is introduced

Notation	Explanation
$AC \leftarrow MDR$	Contents of MDR loaded into AC.
$AC \leftarrow AC + MDR$	Contents of MDR added to AC.
$MDR \leftarrow M$	Contents of memory location MAR loaded into MDR.
$M \leftarrow MDR$	Contents of MDR stored at memory location MAR.
$PC \leftarrow MDR$	Address portion of MDR loaded into PC.
$MAR \leftarrow PC$	Contents of PC loaded into MAR.

Register transfer notation uses these assignment operations as well as timing information to break down a machine-level instruction into *microinstructions* that are executed in successive microcycles.

	<i>Timing</i>	<i>Microinstructions</i>	
timing variable t_j $1 \leq j \leq k$	t_1	MAR \leftarrow PC	
	t_2	MDR \leftarrow M, PC \leftarrow PC+1	Parallel Operations
	t_3	OPC \leftarrow MDR	

Sequential Operations

The microcode for the fetch portion of each instruction.

Data Path design

- Majority of the microinstructions move data from one register to another
- This suggests that the controller's main task is to enable these data transfers (the other task is to enable operations)
- The task of designing the controller is better done by focusing on the register-to-register transfers rather in individual instructions!

Plan of action

1. For each instruction write the micro-instructions using the notation suggested in the next slide
2. Then using these micro-instructions, for each register
 - A. List the transfers from other registers and associated control
 - B. Decide required multiplexor parameter (Size and control lines)
 - C. Decide the control logic to drive this multiplexor

Microinstructions for the execute portion

<i>Control</i>	<i>Microcode</i>	
	CLA	
Active Control	c_{CLA} t_4 $AC \leftarrow 0$	Microinstruction
	CIL	
	c_{CIL} t_4 $AC \leftarrow \text{Shift}(AC)$	
	LDA <Address>	
	c_{LDA} t_4 $\text{MAR} \leftarrow \text{MDR}$	
	c_{LDA} t_5 $\text{MDR} \leftarrow M$	
	c_{LDA} t_6 $AC \leftarrow \text{MDR}$	

No arguments

Microinstructions for the execute portion

<i>Control</i>		<i>Microcode</i>
ADD <Address>		
c_{ADD}	t_4	MAR \leftarrow MDR Address part
c_{ADD}	t_5	MDR \leftarrow M
c_{ADD}	t_6	AC \leftarrow AC + MDR
AND <Address>		
c_{AND}	t_4	MAR \leftarrow MDR Address part
c_{AND}	t_5	MDR \leftarrow M
c_{AND}	t_6	AC \leftarrow AC AND MDR

<i>Control</i>	<i>Microcode</i>
STA <Address>	
c_{STA}	t_4
	MAR \leftarrow MDR
c_{STA}	t_4
	MDR \leftarrow AC
c_{STA}	t_5
	M \leftarrow MDR
CMA	
c_{CMA}	t_4
	AC $\leftarrow \neg$ AC
JZ <Address>	
c_{JZ}	t_4
	if (AC = 0) PC \leftarrow MDR

<i>Control</i>	<i>Microcode</i>
IN	
c_{IN}	$t_4 \mid \text{AC} \leftarrow \text{INR}$
OUT	
c_{OUT}	$t_4 \mid \text{OUTR} \leftarrow \text{AC}$
HLT	
c_{HLT}	$t_4 \mid t_j \leftarrow 0 \text{ for } 1 \leq j \leq k$ Reset

Generating Control Signals

Define ***control variables*** that control the movement of data between registers or combine the contents of two registers and assign the result to another register.

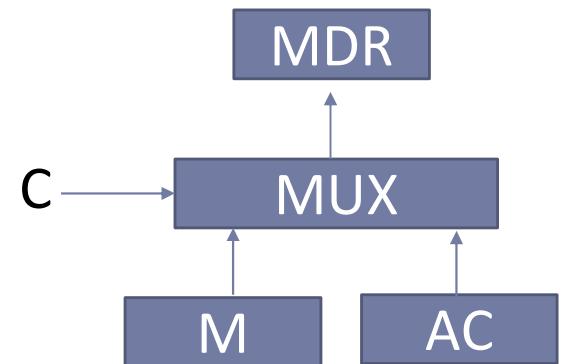
Associate a control variable $L(A,B)$ if a microinstruction results in the movement of data from register B to register A, denoted $A \leftarrow B$.

$L(OTP,MDR) = t_3$: OPC is loaded with the contents of MDR when $t_3 = 1$

Associate a control variable $L(A, B \odot C)$ if a microinstruction results in the combination of the contents of registers B and C with the operation \odot and the assignment of the result to register A, denoted $A \leftarrow B \odot C$.

$L(AC, AC+MDR) = C_{ADD} \wedge t_6$: The contents of AC are added to those of MDR and copied into AC when $C_{ADD} \wedge t_6 = 1$.

Group together all the microinstructions affecting MDR



Derive control variables

$$L(\text{MDR}, \text{M}) = t_2 \vee (c_{\text{ADD}} \vee c_{\text{AND}} \vee c_{\text{LDA}}) \wedge t_5$$

$$L(\text{MDR}, \text{AC}) = c_{\text{STA}} \wedge t_4$$

C is 0 M→ MDR

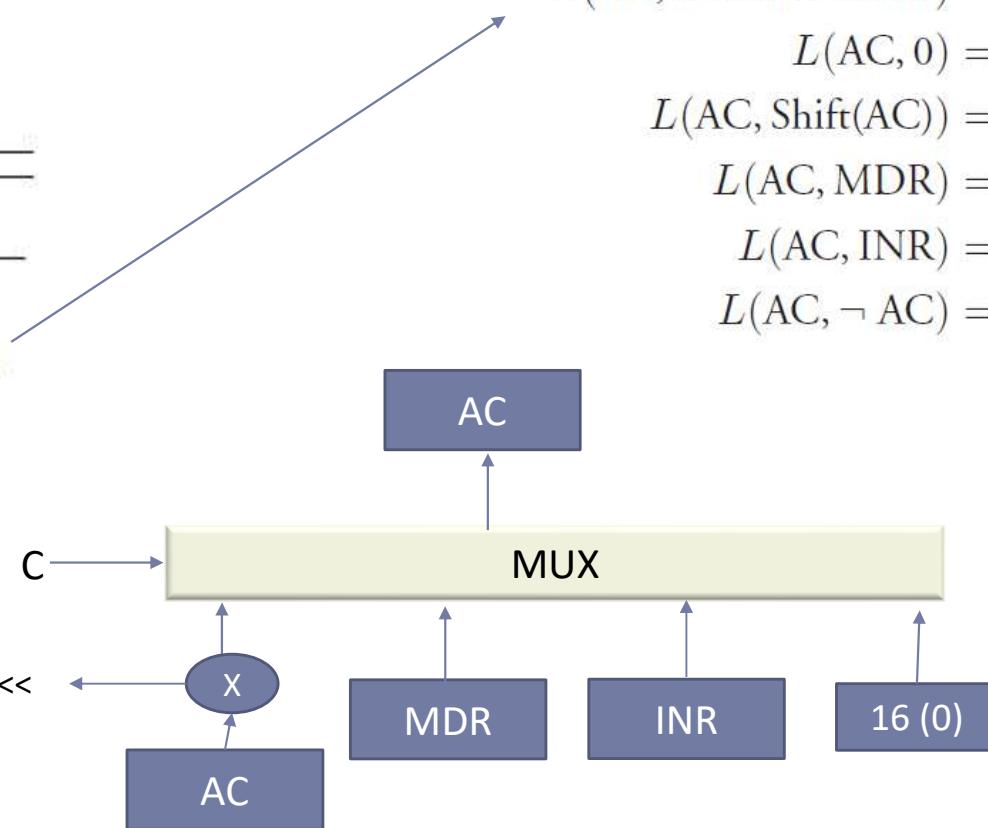
C is 1 AC → MDR

Control of Accumulator

Group together all the microinstructions affecting accumulator

Control	Microcode
	AC
c_{ADD} t_6	$AC \leftarrow AC + MDR$
c_{AND} t_6	$AC \leftarrow AC \text{ AND } MDR$
c_{CLA} t_4	$AC \leftarrow 0$
c_{CIL} t_4	$AC \leftarrow \text{Shift}(AC)$
c_{LDA} t_6	$AC \leftarrow MDR$
c_{CMA} t_4	$AC \leftarrow \neg AC$
c_{IN} t_4	$AC \leftarrow INR$

+, !, &, <<



Derive control variables

$$L(AC, AC + MDR) = c_{ADD} \wedge t_6$$

$$L(AC, AC \text{ AND } MDR) = c_{AND} \wedge t_6$$

$$L(AC, 0) = c_{CLA} \wedge t_4$$

$$L(AC, \text{Shift}(AC)) = c_{CIL} \wedge t_4$$

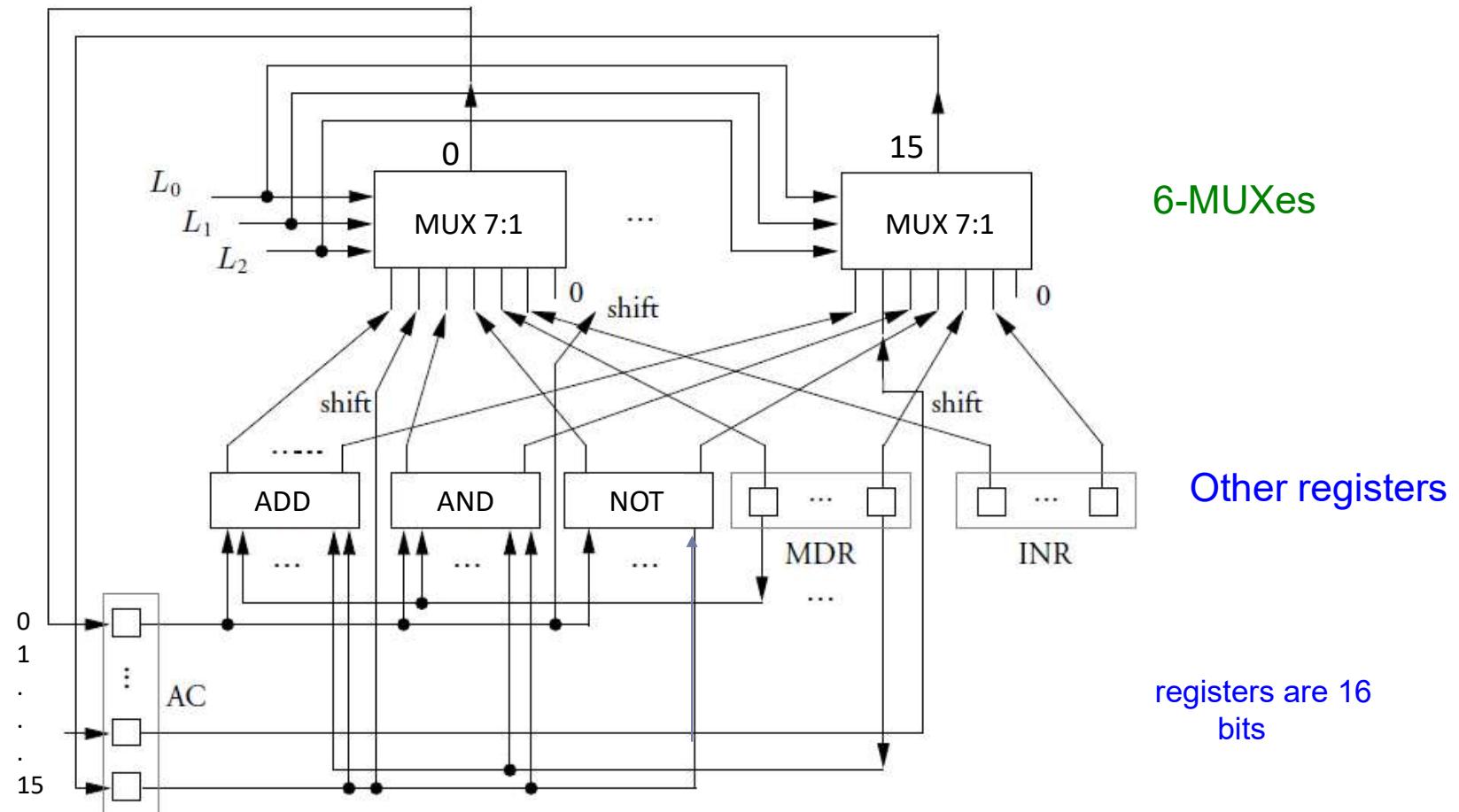
$$L(AC, MDR) = c_{LDA} \wedge t_6$$

$$L(AC, \neg AC) = c_{CMA} \wedge t_4$$

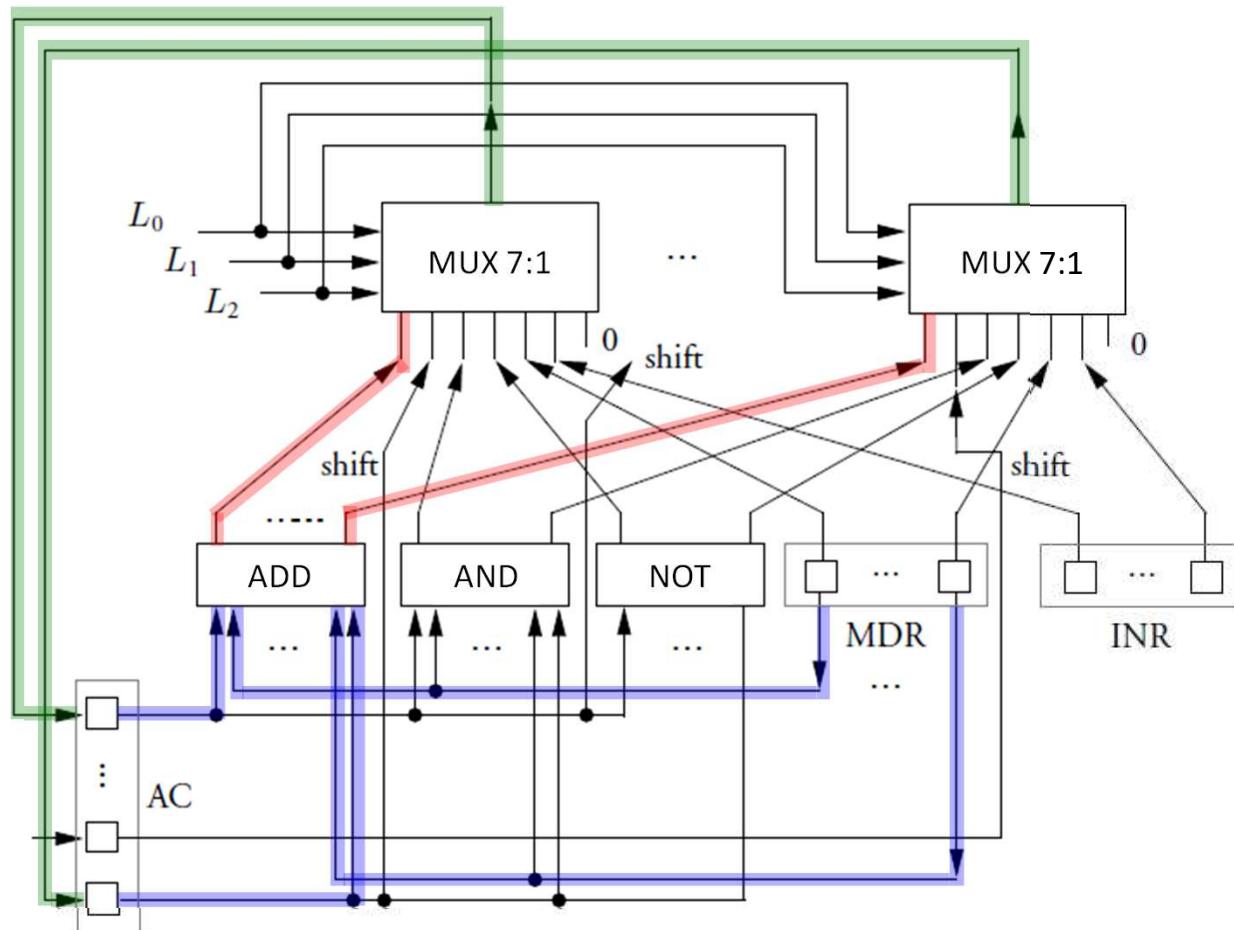
$$L(AC, INR) = c_{IN} \wedge t_4$$

Hardware Implementation

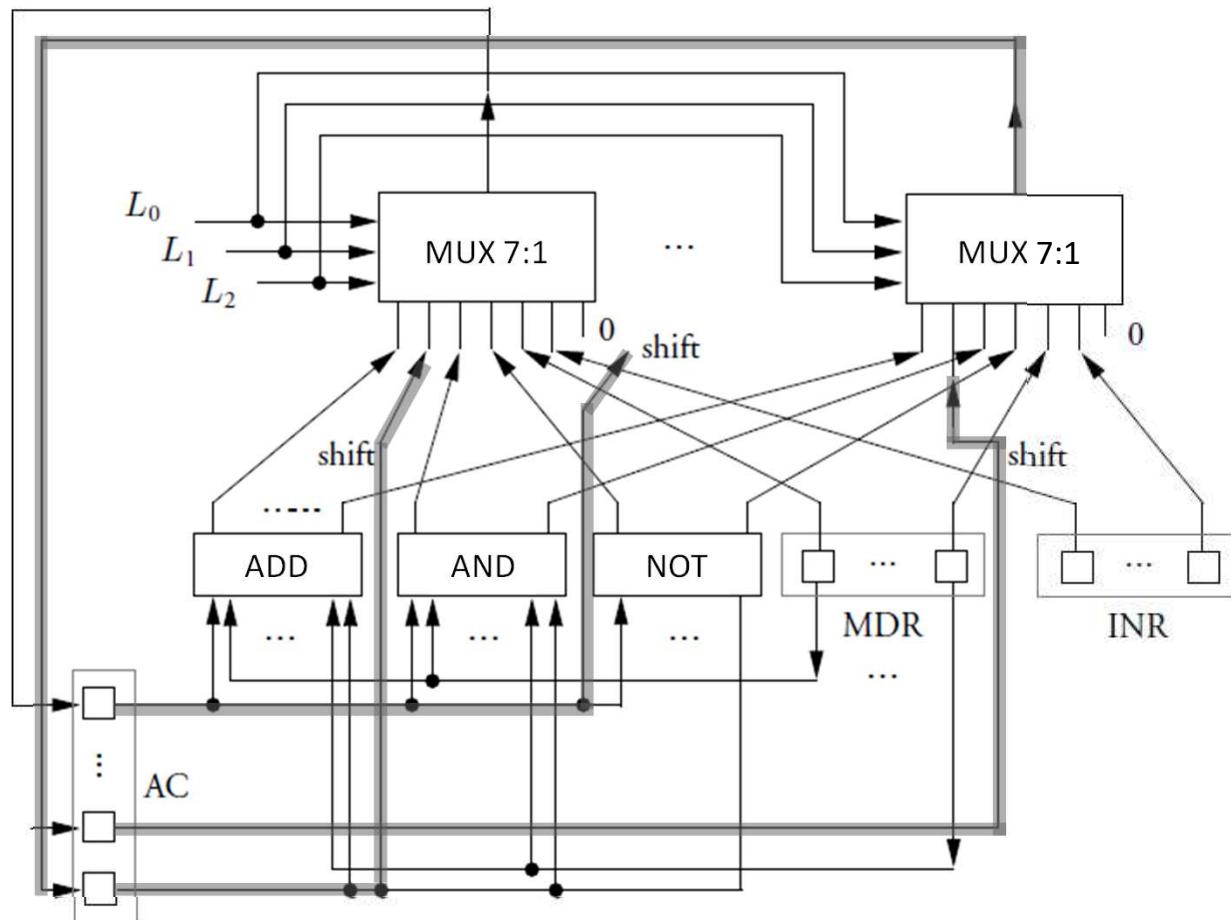
16-bit
ALU



Execution of $AC \leftarrow AC + MDR$



Execution of $AC \leftarrow Shift(AC)$



Group together all the microinstructions affecting MAR

<i>Control</i>	<i>Microcode</i>
	MAR
	t_1
c_{ADD}	t_4
c_{AND}	t_4
c_{LDA}	t_4
c_{STA}	t_4
	$MAR \leftarrow PC$
	$MAR \leftarrow MDR$

Derive control variables

$$L(MAR, PC) = t_1$$

$$L(MAR, MDR) = (c_{ADD} \vee c_{AND} \vee c_{LDA} \vee c_{STA}) \wedge t_4$$

<i>Control</i>	<i>Microcode</i>	
		Group together all the microinstructions affecting a given register
c_{STA}	t_5 $M \leftarrow MDR$	
c_{JZ}	t_2 t_4 $PC \leftarrow PC + 1$ if ($AC = 0$) $PC \leftarrow MDR$	Derive control variables
OPC	t_3 $OPC \leftarrow MDR$	$L(M, MDR) = c_{STA} \wedge t_5$ $L(PC, PC+1) = t_2$ $L(PC, MDR) = (AC = 0) \wedge c_{JZ} \wedge t_4$ $L(OPC, MDR) = t_3$

<i>Control</i>	<i>Microcode</i>	
		Group together all the microinstructions affecting a given register
OUTR c _{OUT}	t ₄ OUTR ← AC	
HLT c _{HLT}	t ₄ t _j ← 0 for 1 ≤ j ≤ k	
$L(\text{OUTR}, \text{AC}) = c_{\text{OUT}} \wedge t_4$		Derive control variables
$L(t_j) = c_{\text{HLT}} \wedge t_4 \text{ for } 1 \leq j \leq 6$		

Lab Exercise

- Design 3 input logic gates (AND, OR, EXOR, NAND, & NOR) with Multiplexor (no gates)
 - EXOR output is 1 only when one of the inputs is 1!
- Design a Register File of 4 Registers (4-bit) with 1 READ port and 1 WRITE port

Super Scalar Processor

More Complex and Realistic Processor

- For similar exercise for a Superscalar architecture with MIPS ISA
- Superscalar processors improve instruction level parallel operation using Pipeline at multiple levels
- ARM Cores support 3-stage pipeline: Fetch, Decode, and Execute

MIPS Instruction Set

MIPS operands

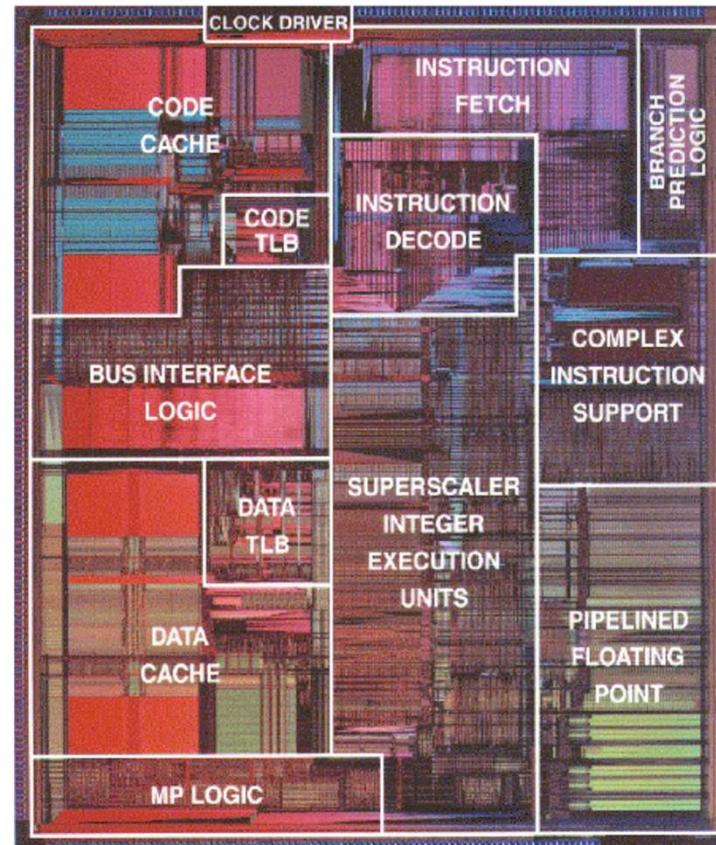
Name	Example	Comments
32 registers	\$s0, \$s1, ..., \$s7 \$t0, \$t1, ..., \$t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers \$s0-\$s7 map to 16-23 and \$t0-\$t7 map to 8-15.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Shift left by constant
	shift right logical	srl \$\$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Shift right by constant
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory

Pentium Die – Circa 1993

- First Superscalar processor with IA-32



Superscalar Processor

- In scalar processor, an instruction is completed before the next instruction is fetched.
- ARM is a simple Superscalar processor where multiple instructions are in various stages (fetch, decode, and execute) of their execution in a full pipeline
- 5-stage pipeline is common among Superscalar processing

Pipeline - Overlap



- Compare
 - 1. Single person serving a full menu to every one in the line.
 - 2. Serving in a pipeline as shown in the right

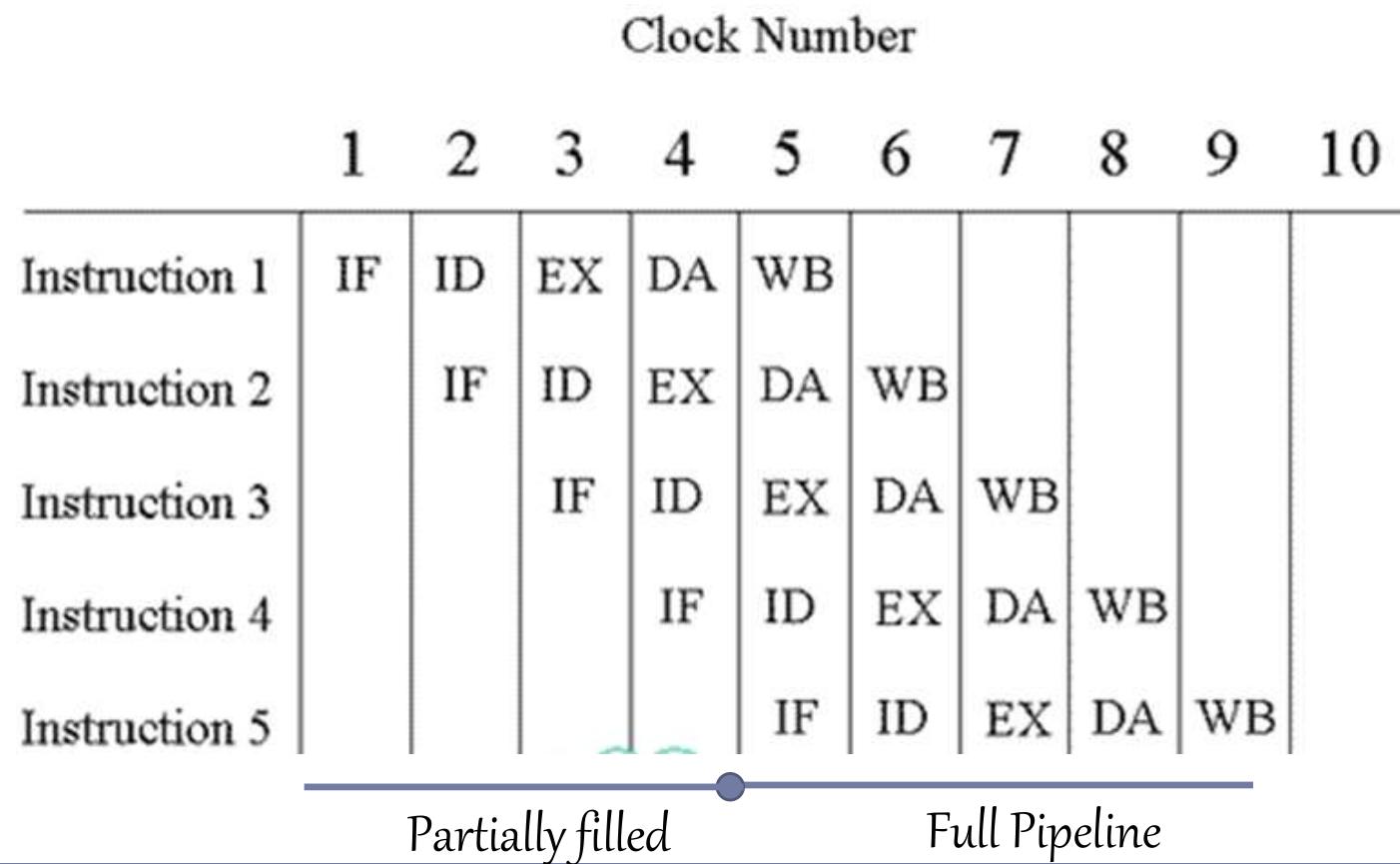


- Use the following parameters for comparison:
 - Time to serve an individual
 - Time to serve 100 people

5-Stage Pipe Line

- Pipeline stages are separated by buffers that stores output of a stage before it is given as input to the succeeding stage.
- Stages of 5-stage pipeline
 - Instruction Fetch (IF)
 - Instruction Decode (ID)
 - Execution (EX)
 - Memory Access/*Data Access* (MEM/DA)
 - Write-back Cycle (WB)

5-Stage Pipeline Operation



Problem 1

- Consider the non-pipelined hypothetical processor. Assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20% and 40% respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipelining?

Solution to Problem 1

Take a 100 line program
40 ALU instructions
20 Branch instructions
40 Memory instructions

Non-pipe lined processor:

clock cycle = 1 ns

Execution time (in cycles)

$$\begin{aligned} &= 40 \times 4 + 20 \times 4 + 40 \times 5 \\ &= 160 + 80 + 200 = 440 \text{ cycles} \end{aligned}$$

Execution time (in secs) = 440 ns

5-Stage Pipelined processor

Clock cycle is 1.2 ns

First instruction takes 6 clock cycle

After that an instruction is completed for every cycle
To complete rest of the 99 instruction it takes 99 cycles

The total execution time for pipelined process

$$= 99 + 6 \text{ cycles} = 105 \text{ cc} = 105 \times 1.2 = 105 + 1.2 = 126 \text{ ns}$$

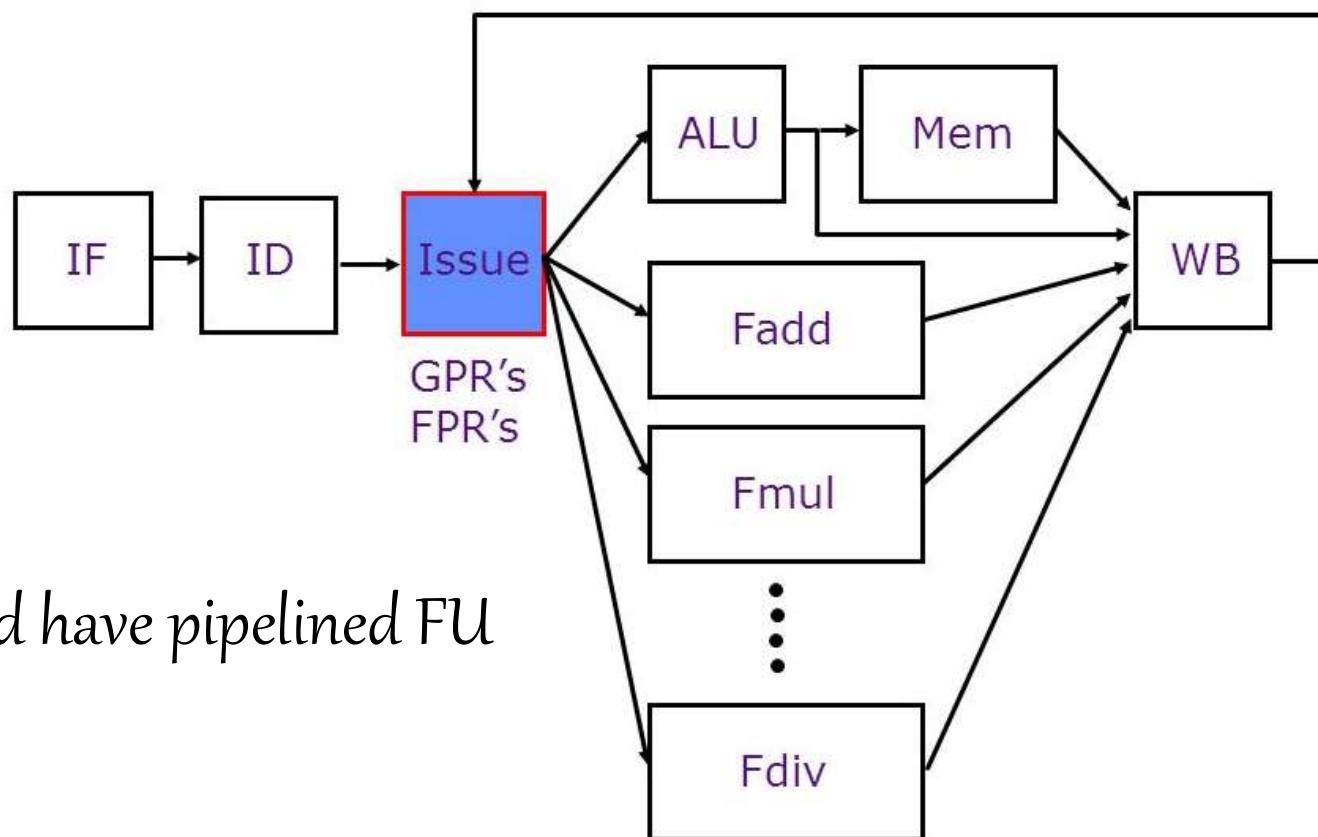
Speed up = $440/126 = 3.52$;

Speed up if we ignore pipeline latency

$$= (440 \times 1) / (100 \times 1.2) = 4.4 / 1.2 = 3.66$$

Speed up an approach 5 - why 5?

Pipeline with Multiple ALUs



Test Score and Lab Assignments

- Overall average 13.67 (11.71)
 - COE Average 14.51 (11.88)
 - CED Average 13.60 (12.17)
-
- Lab 1: Embest (19th)
 - Lab 2: Caching (26th)
 - Lab 3: Verilog (2nd April)
 - Lab 4: Verilog & Misc (9th)
 - Lab 5 : Exam - 16th (April 16th – 10 to 1 pm)
 - Your laptop should include all the tools that I gave and my notes.
 - Fully charged.
 - WiFi and Ethernet adapter disabled

Pipeline Hazards

- There are occasions when the next instruction in the pipeline cannot execute in the following cycle.
- This is referred to as **Pipeline Hazard**
- The pipeline hazard is classified into 3 types
 - **Structural Hazard**
 - **Data Hazard**
 - **Control Hazard**

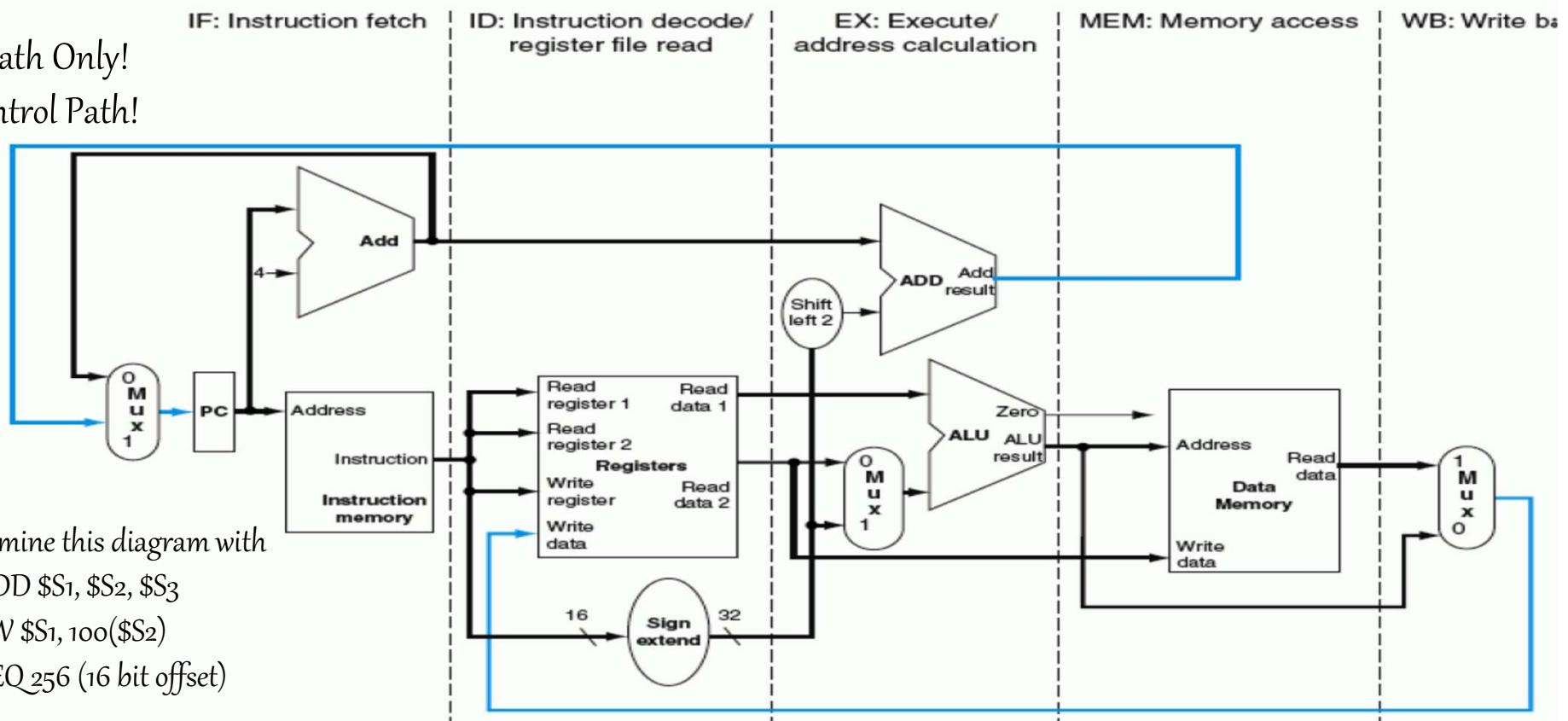
Effect of Hazard on Throughput

- Hazards break the pipeline flow
- Pipeline needs to be emptied and refilled or pipeline need to be stalled
- Both will reduce the instruction throughput
- There are solutions to improve the throughput by working around hazards – Compiler implements these workarounds

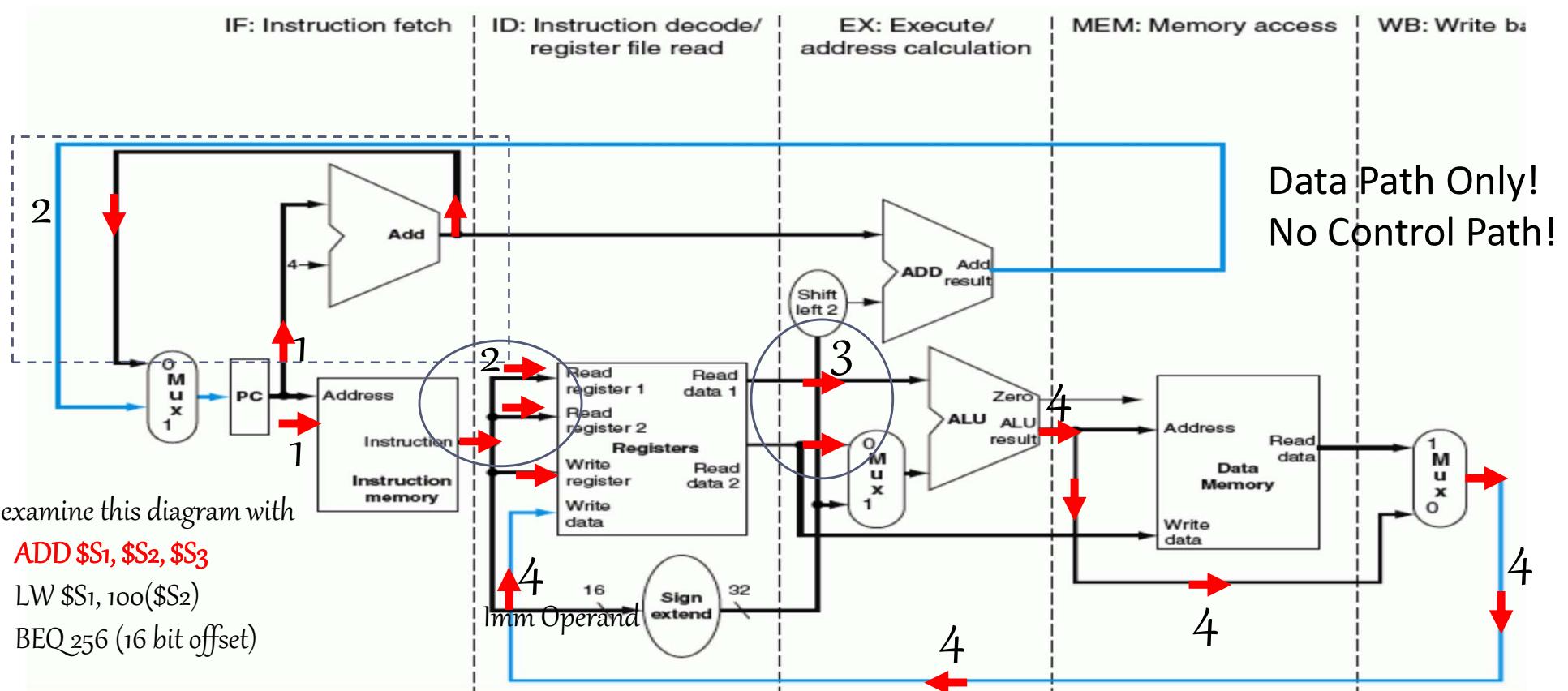
5-Stage Pipeline with a Single ALU

Data Path Only!

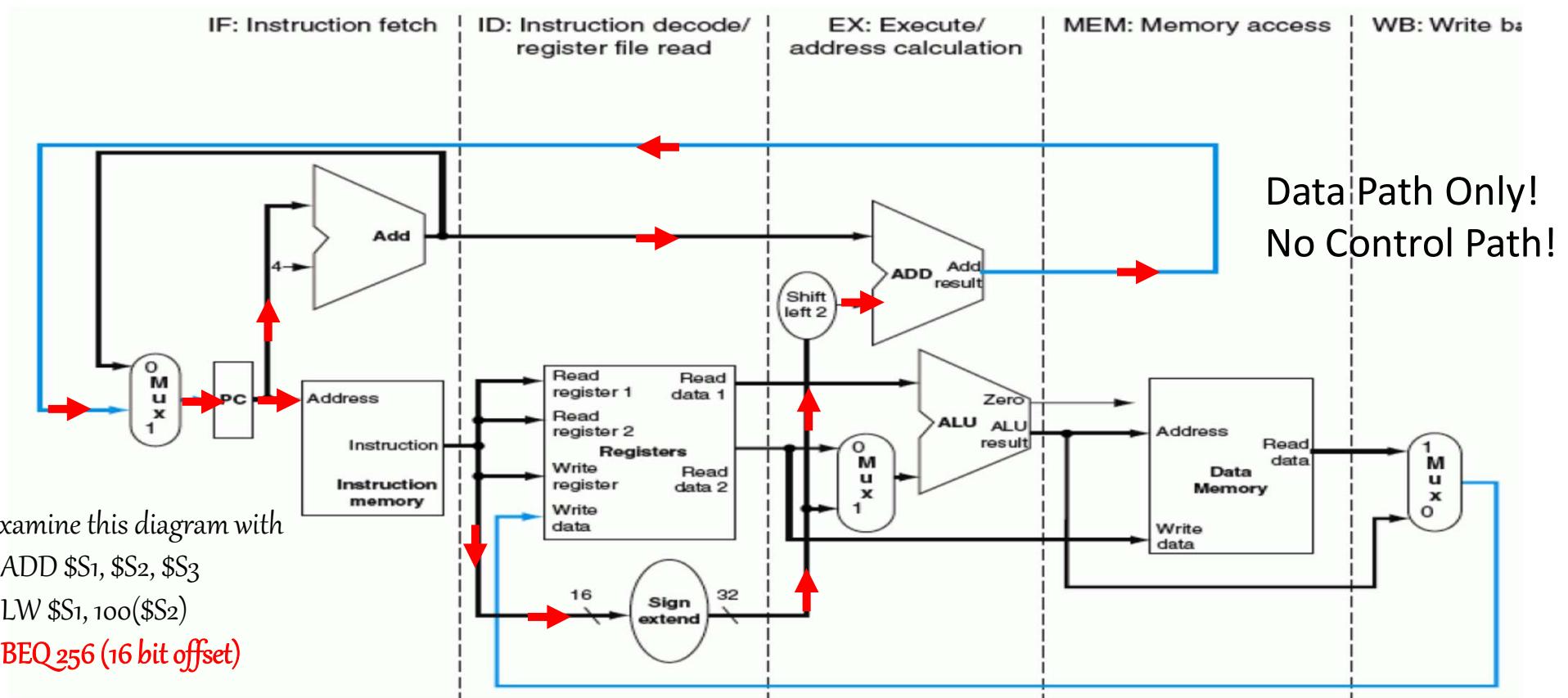
No Control Path!



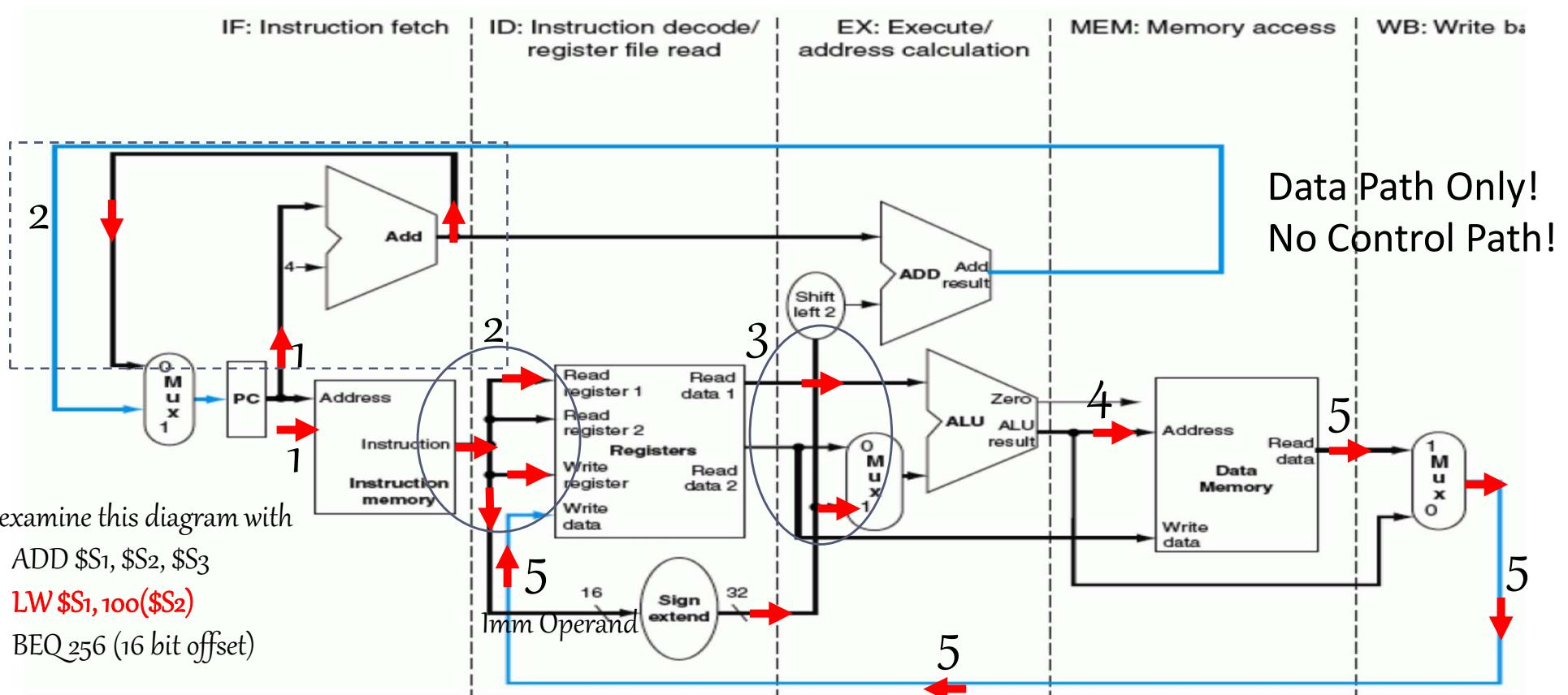
Data Path of ADD



Datapath for BEQ – Number the Arrows



Data Path LW



Structural Hazard

- Structural hazard is caused by limited number of ALUs (FU) in execution stage
- If there is only one floating point multiplier functional unit then the 2nd of the two consecutive multiplication instructions will be stalled until the first multiplication is completed (it may take several cycles!)

Data Hazard

- Data dependence is the root cause of Data hazard.
- An instruction may wait for the previous MUL instruction to complete if it is using the result of that multiplication

Hazards and Stalls

- Instructions are stalled due to hazard
- Example:
- Stalled cycles go waste and consequently average execution time of an instruction becomes more than 1 cycle
- Stalling reduces the throughput – reduces the pipeline effect
- This is the reason that Superscalar architecture is not suited for CISC - instruction size variation, frequent memory access

Control Hazard

- Control Hazard is due to branching logic in programs
- Execution of branch instruction may change the control flow and make the following instruction (in pipeline) unnecessary
- Instruction to be executed need to be fetched from different part of the memory space and the pipeline need to restart
- More than one pipeline instruction may need to be pre-empted (not even delayed)!

Role of Compiler with Hazards

- Compilers attempt to eliminate hazards to improve the performance
- For instance, a compiler can change the issue order of instructions to eliminate data and control hazards.
 - Other instructions can be scheduled ahead between 2 instructions with data hazard
- Effect of control hazard is reduced by branch prediction methods

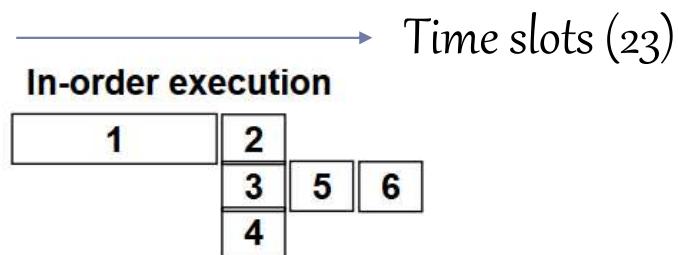
Out Of Order Execution (skip to next module)

- Look ahead in a window of instructions and find instructions that are *ready to execute*
 - Don't depend on data from previous instructions still not executed
 - Resources are available
- Out-of-order execution
 - Start instruction execution before execution of a previous instructions
- Advantages:
 - Help exploit Instruction Level Parallelism (ILP)
 - Help cover latencies (e.g., L1 data cache miss, divide)

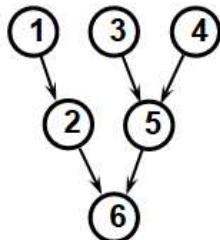
Example of Out of Order Execution

- Example: assume that a divide operation takes 20 cycles.

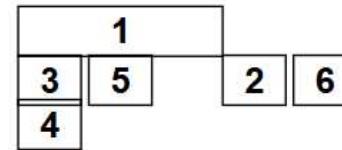
```
(1) r1 ← r4 / r7  
(2) r8 ← r1 + r2  
(3) r5 ← r5 + 1  
(4) r6 ← r6 - r3  
(5) r4 ← r5 + r6  
(6) r7 ← r8 * r4
```



Data Flow Graph



Out-of-order execution

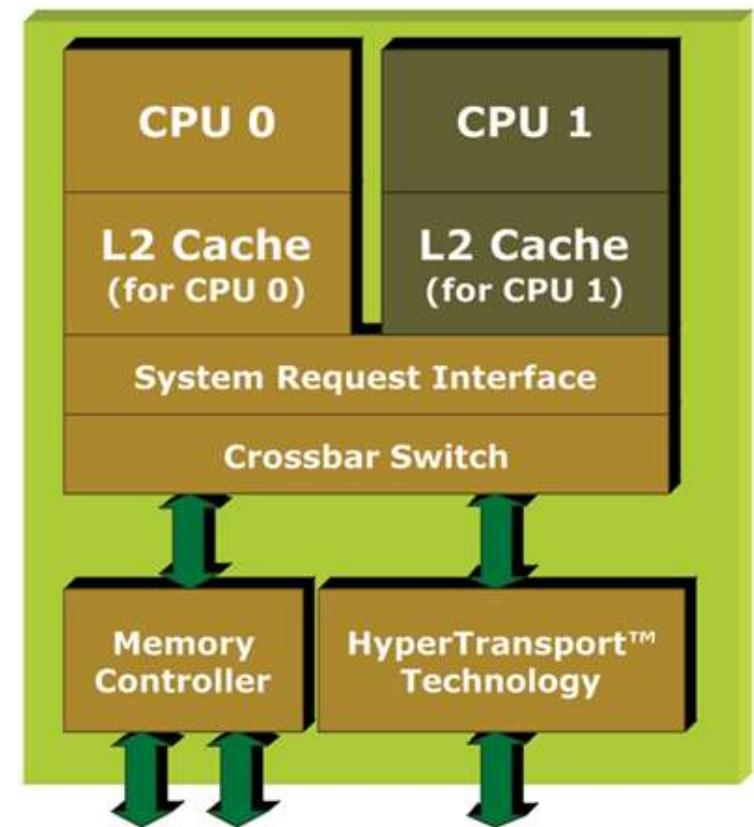


Multi-processor System

- There are architectures where there are multiple processors with tight or loose coupling or no coupling
- Multi-core architecture is a type of multi-processor architecture where there are more than 1 processor within a single processor chip

AMD Athlon 64 X2 Dual-Core Processor

- ❑ Athlon supports two buses –
Memory bus, and
HyperTransport (I/O Bus)

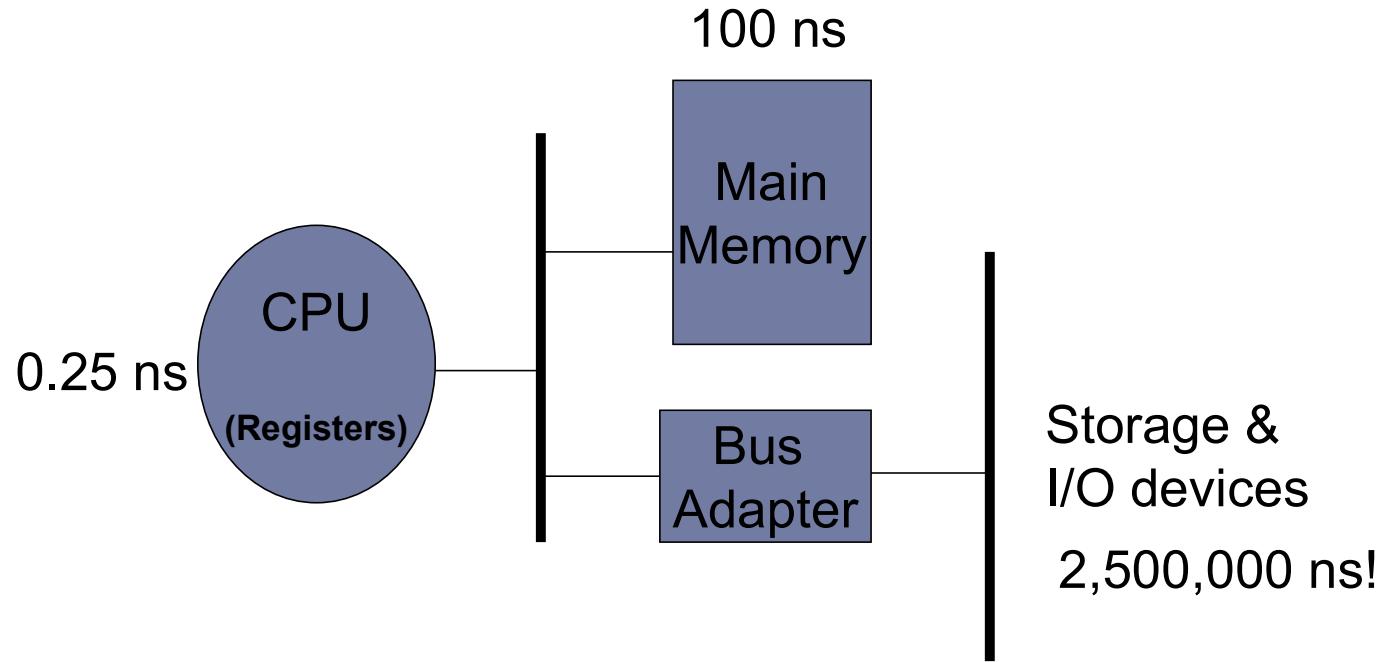


Multi-processor Systems

- This topic is covered in a separate module!!

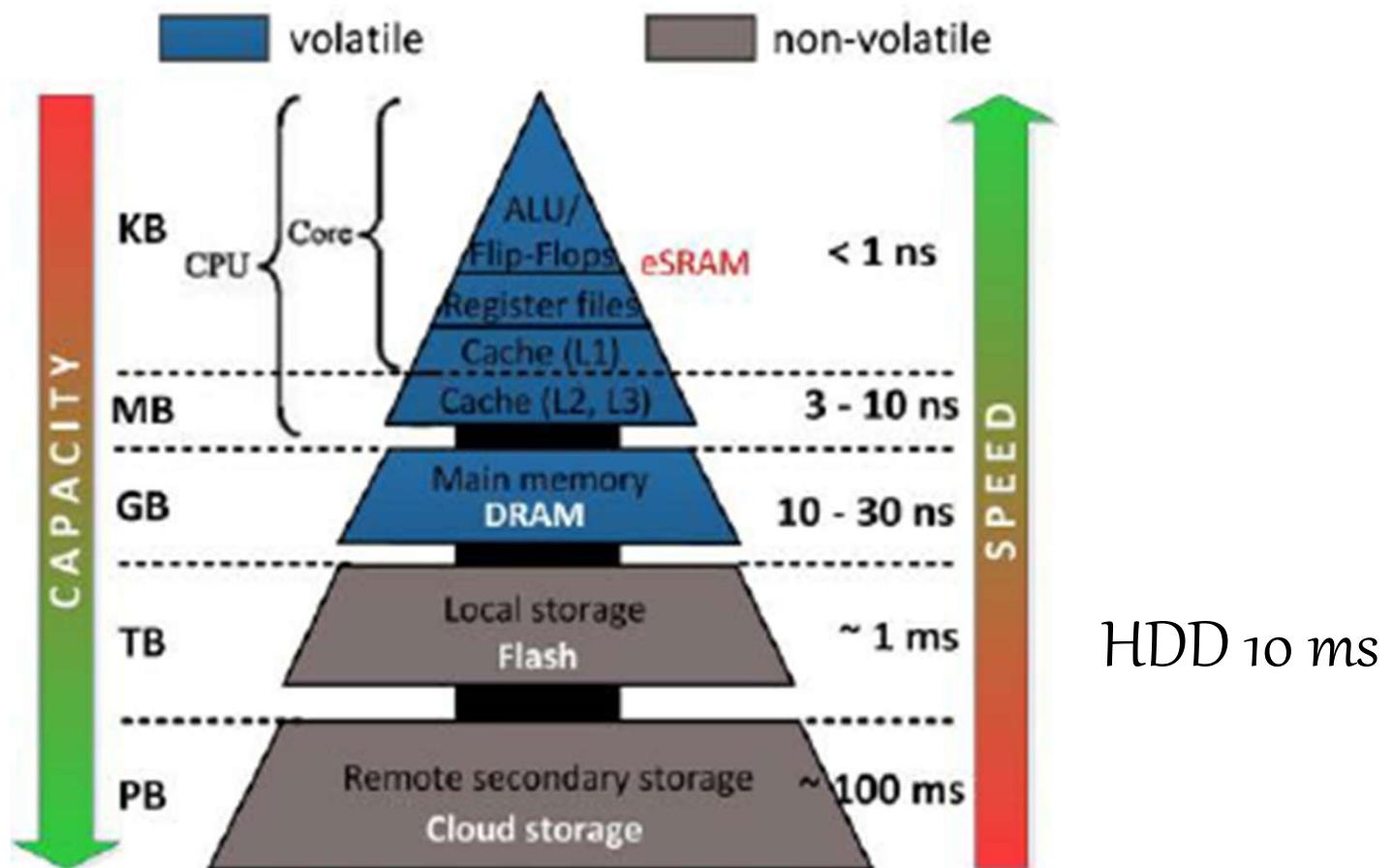
Memory Hierarchy

What is Memory Hierarchy and Why?

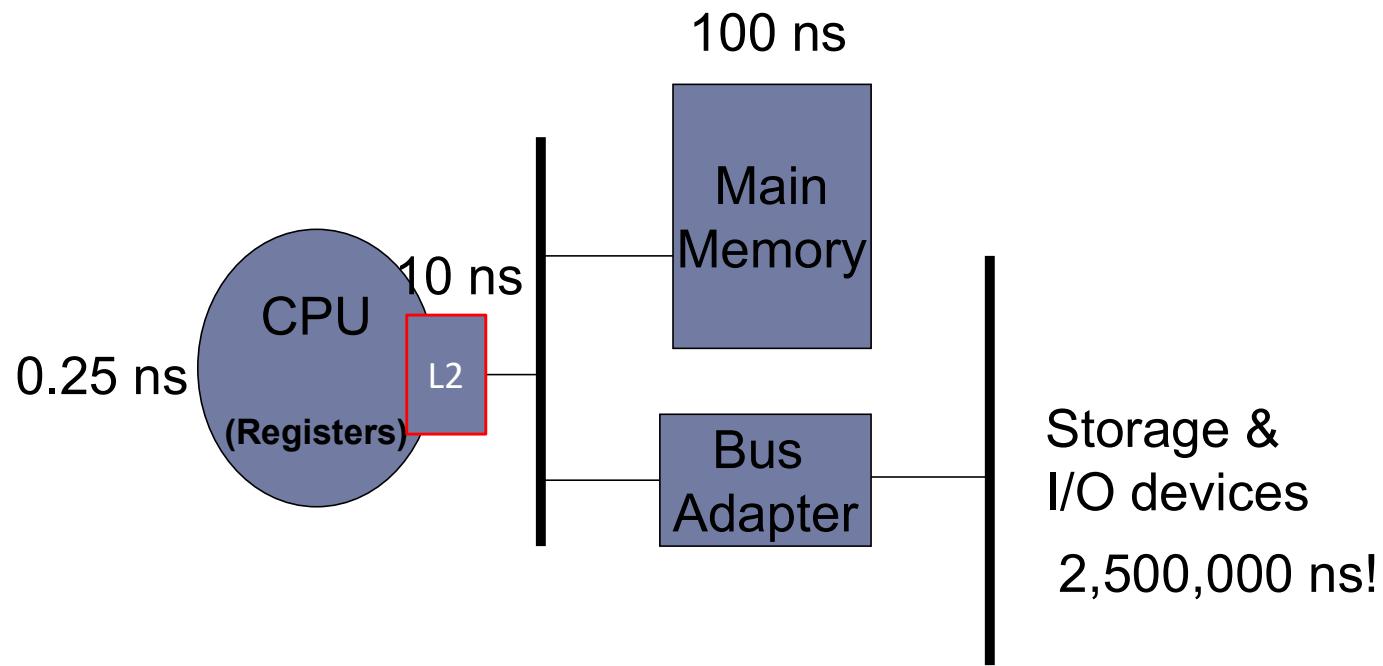


Execute a program of 100 instruction to understand the bottle neck problem. Assuming Blocking between Memory and Storage

Memory Hierarchy & Cache



What is Memory Hierarchy and Why?



Execute a program of 100 instruction to understand How L1 cache helps

Flash and Disk

- Reviewed in another module!

Locality Properties

Spatial Locality property: When data at address X is accessed, there is a high probability that data at address $X+1$ will be accessed in the near future.

Note: Blocking is productive only with these properties!

Temporal Locality: When data at address X is accessed, there is a high probability that the same data will be accessed in the near future.

This means when data X is read, read all the data items which are nearby. This is called **blocking**. Blocking reduces the access **overhead** and improves memory **throughput**.

Registers

- Registers are within the processor chips, thus they operate at the highest speed (of the processor)
- Registers are classified into two types
 - Dedicated registers:
 - Accumulator, PC, Status Register, IR, etc
 - General purpose registers

Cache (1)

- Cache is a smaller, faster, and expensive memory (Compared to main memory)
- Improves the throughput/latency of slower memory next to it in the memory hierarchy by using locality properties
- Caching improves overall performance by Blocking reads and delaying the writes to slower memory

Cache (2)

- There are two cache memories L₁ and L₂ between CPU and main memory.
- L₁ is built into CPU chip
- L₂ is an SRAM chip(s).
- Both Data and Instructions are cached either in the same cache or in different
- Reference: http://cs.mwsu.edu/~terry/courses/5133/lectures/Chapter_2.html

Cache and Main Memory Technologies

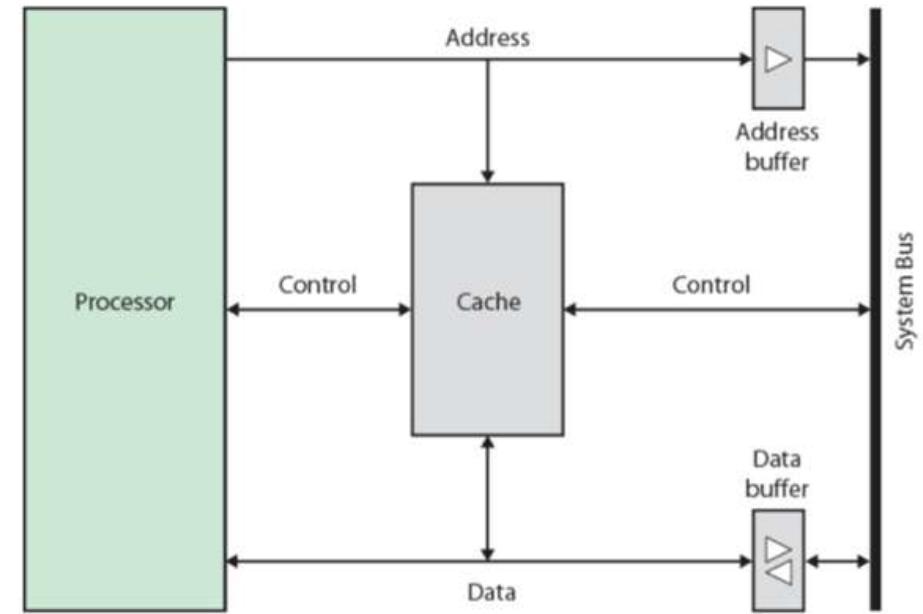
- Main memory is made of DRAM
- DRAM uses capacitors for memory
- This allows larger capacity for main memory, however at the cost of access speed
- Cache memory is made of SRAM
- SRAM uses transistors for memory
- This allows faster memory, however at the cost of capacity

Cache Operation

- Cache Hit: CPU finds the required data item (or instruction) in the cache.
- Cache Miss: CPU does not find the required item in the cache.
 - CPU Stalled (is this same stalling?)
 - Hardware loads the entire block that contains the required data item from the memory into cache.
 - CPU continues to execute once the cache loaded.

Hit and Miss

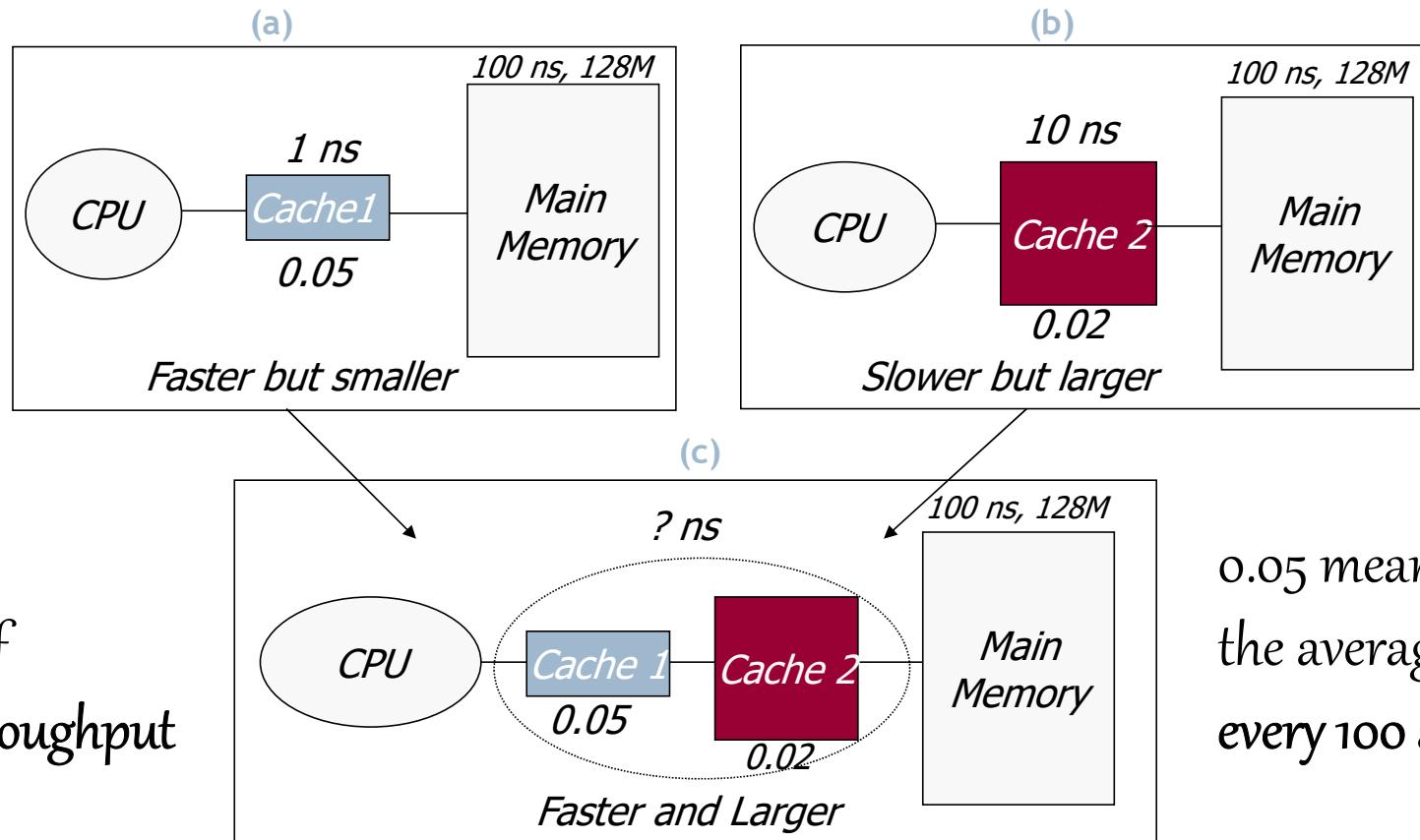
- Hit: Hit Rate & Hit Time
- Miss: Miss Rate & Miss Penalty
- Hit Time \ll Miss Penalty
- Miss Penalty is typically measured in CPU cycles.
- In general, rates are statistical averages!



Hit and Miss time are absolute values

Reducing Miss Penalty - Multilevel caches (1)

Assumption:
One mem
Access per
Instruction!



Talk about
lower bound of
instruction throughput

0.05 means 5 misses on
the average for
every 100 accesses

Cache Performance - Program Execution Time

- CPU Clock Cycle
- Cycle Time
- IC – Instruction Count
- Program Execution Time (Simple model)
= (Useful CPU Cycles + Idle CPU Cycles) x Cycle Time

Stalled CPU Cycles

Stalled CPU Cycles

= Number of Cache Misses \times Miss Penalty

= $(IC \times \frac{\text{Memory Access}}{\text{Instruction}} \times \text{Miss Rate}) \times \text{Miss Penalty}$

Instruction

IC – Instruction Count in the program under consideration

Recall that Unit of Miss Penalty is in CPU cycles

Exercise

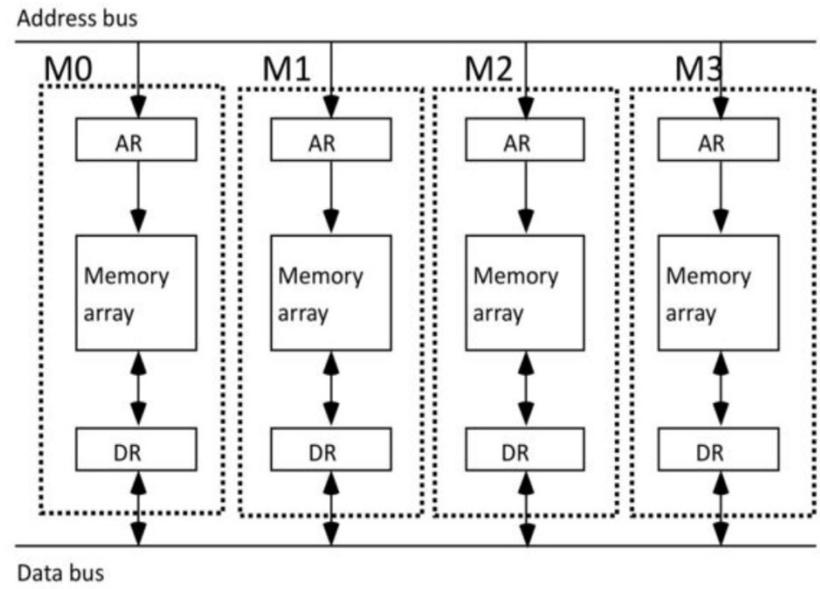
- Total execution time of a program includes two components: Useful cycles and Stalled cycles. Compute the useful cycles for a program with 1 million instructions. Use the following data: Average Memory access per instruction is 1.05; Hit Time is 2 cycles; Miss Rate 1 out of 1000 access to memory; Miss Penalty of 100 cycles

Memory Access Time

- Component of Memory Access Time (a simple model):
 - Write Address to Address Bus
 - Wait for data
 - Data written to Data Bus
 - Copy data
- 
- Bulk of the time

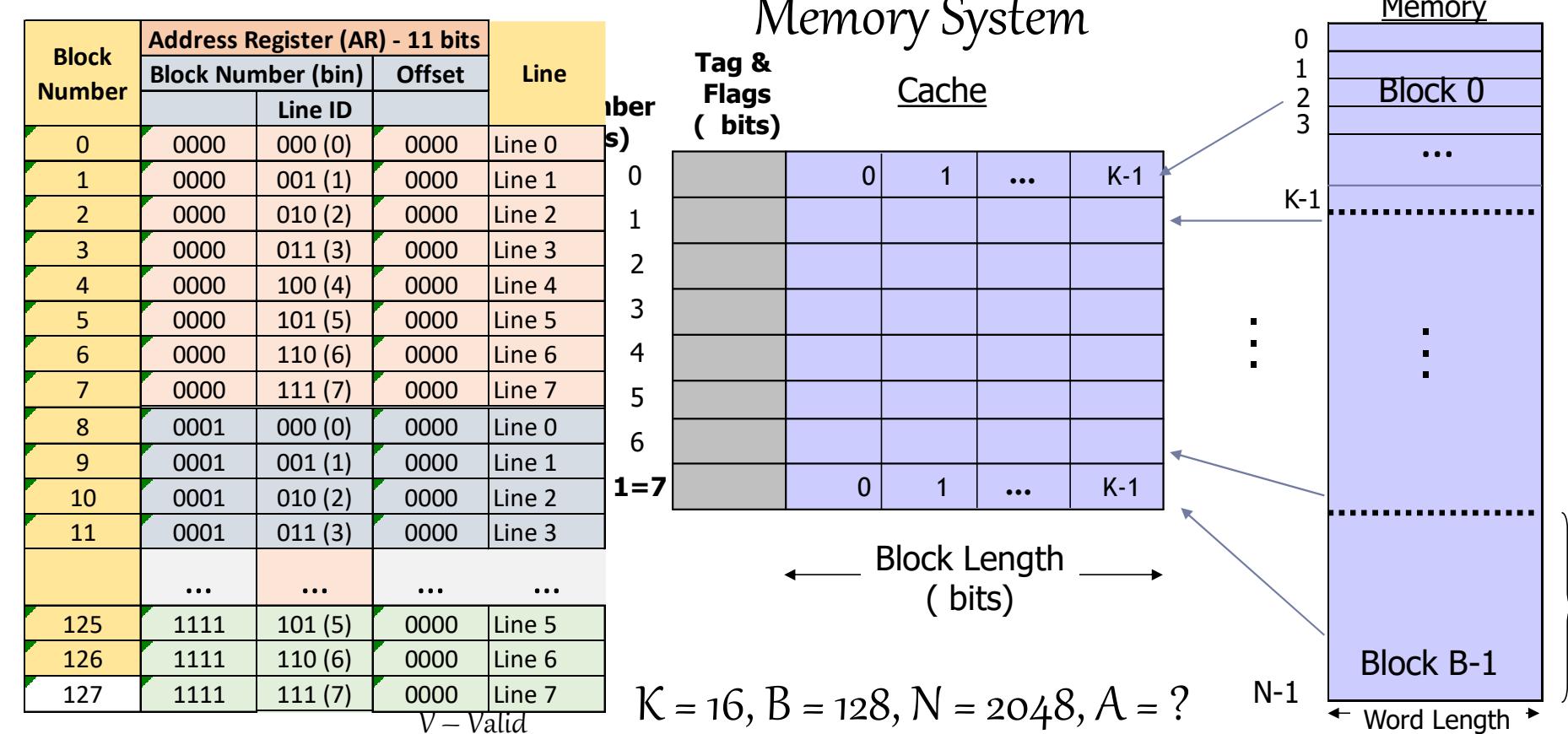
Cache Memory Operation - Assumptions

- Cache memory is faster and smaller in capacity in relation to ...
- Block size of cache and next level are the same
- Every access to next level is in terms of block
- Block size is decided by memory interleaving and bus interface



Memory with 4-way Interleaving

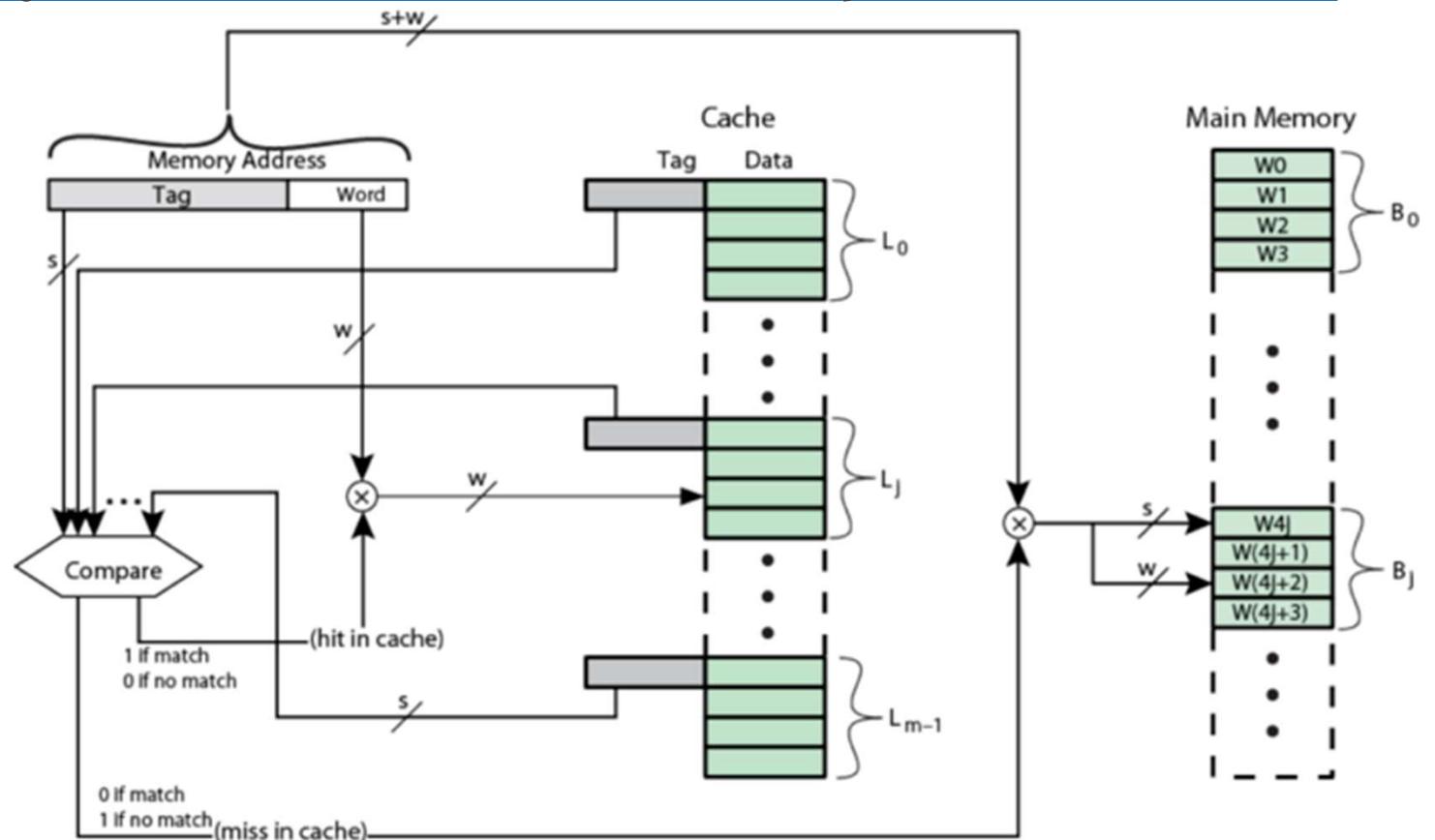
(Unified) Cache Operation



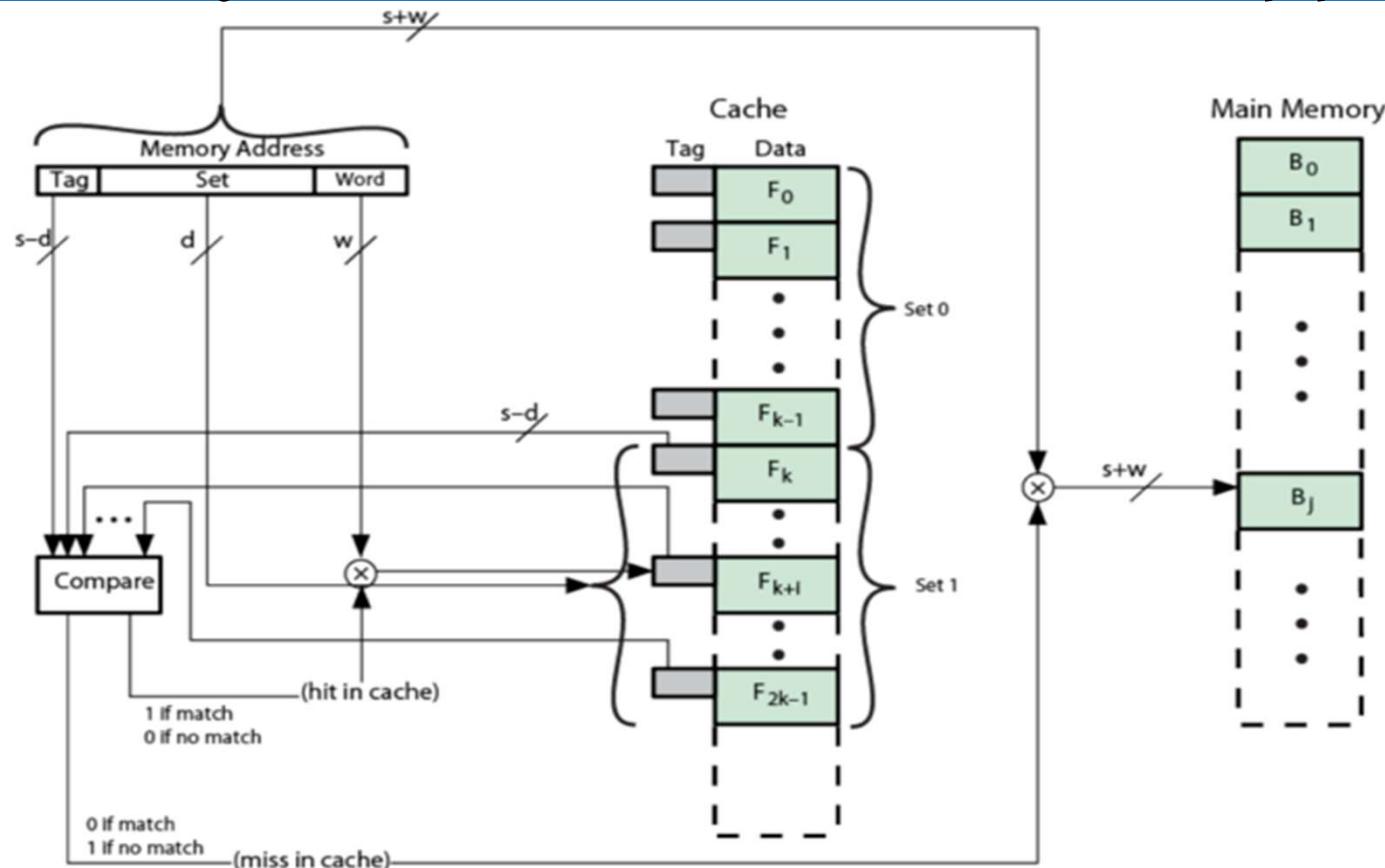
Issues of Direct Mapping

- Direct mapping is simple, but could lead to *thrashing*.
- *Thrashing* results when two or more blocks of the program with strong locality relation are mapped to the same line in the cache
- This mapping causes unusually higher values of cache misses.
- *Fully associative* mapping addresses this issue.

Fully Associative Cache Operation



m -Way Set Associative Cache Mapping



Mapping Function

- Direct
 - Line value in address uniquely points to a line in cache. 1 tag Comparison
- Set Associative
 - Line value in address points to a set of lines in cache (typically 2/4/8, so 2/4/8 tag comparisons). This is known as 2/4/8 way Set Associative.
- Fully Associative
 - Line value is always 0. This means Line points to all the lines in cache (all tag Comparisons)
 - Uses Content Addressable Memory (CAM) for comparison.
 - Needs non-trivial replacement algorithm

Elements of Cache Design

- There are a number of design options and features with cache:
 - Cache Size – Improved performance with larger cache?
 - Block Size – Larger block or smaller block – which is good?
 - Mapping Function – Direct/Fully Associative/Set Associative?
 - Replacement Algorithm – Random, LRU, Less Frequently Used, FIFO, ...
 - Write Policy – Write Through or Write Back?
 - Write Miss – Write Allocate/No Write Allocate
 - Number of caches – L₁, L₂, L₃, L₄, ...?
 - Split versus Unified/Mixed Cache

Mapping Function Comparison

Cache Type	Hit Ratio	Search Speed
Direct Mapping	Poor	Good
Fully Associative	Good	Poor/Complex h/w
N-Way Set Associative	Moderate	Moderate

Replacement Algorithm

- Least Recently Used (LRU)
- First In First Out (FIFO)
- Least Frequently Used (LFU)
- Randomly chosen one
 - Why Randomly chosen one could be as good as anything else?
 - Overhead is one reason
 - Replace block, say LRU, what is the guarantee that the next access is NOT for it?

Write Policy

- ❑ There are two options in Writing data to cache

1. Write back

- ❑ Information written only to cache.
- ❑ Content of the cache is written to the main memory only when this cache block is replaced or the program terminates.
- ❑ Write requirement is identified by “dirty bit” flag associated with the cache block.

2. Write-through

- ❑ Information written to cache and the memory

Comparing Write Options

Write-back

- Complicates cache coherency problem
- Low memory access overhead
- Better cache access time than write-through
- Requires higher memory bandwidth if blocking

- *Write-through* does not have any advantage for a single processor system due to absence of cache coherency

Write Through

- Simplifies cache coherency problem
- High memory access overhead
- If cache is blocking, then higher access time
- Requires Lower memory bandwidth

Write Miss

- Two options for handling Write Miss
 - 1. Write-Allocate
 - 2. No Write-Allocate

Number of Caches

- Level 1 - Primary Cache (CPU)
- Level 2 - Secondary Cache (SRAM)
- Level 3 - Cache (Cheaper SRAM)

Split versus Unified/Mixed Cache

- Misses per 1000 access with various Cache size
- Unified or Split
- Any recommendation based on the table data?

Size (KB)	Instruction Cache	Data Cache	Unified Cache
8	8.16	44	63
16	3.82	40.9	51
32	1.36	38.4	43.3
64	0.61	36.9	39.4
128	0.3	35.3	36.2
256	0.02	32.6	32.9

Improving Cache Performance

- Total execution time of a program:

$$= \text{Used Cycles} + \text{stalled cycles}$$

$$\text{Used Cycles} = IC \times (\text{Memory Access/instruction}) \times \text{Hit Time per access}$$

$$\text{Stalled Cycles} = IC \times (\text{Memory Access /instruction}) \times \text{Miss Rate} \times \text{Miss Penalty}$$

- Total Execution time of a program can be improved by

- Reducing Miss Penalty

- Improving Hit Time

- Reducing Miss Rate (Number of Misses)

- Misses are given either as *misses/1000 instructions* or *misses/memory-access AKA miss rate*.

Average Access Time with Cache

Average Access Time = Hit Time + Miss Rate X Penalty

Based on single access.

Multi-level Cache

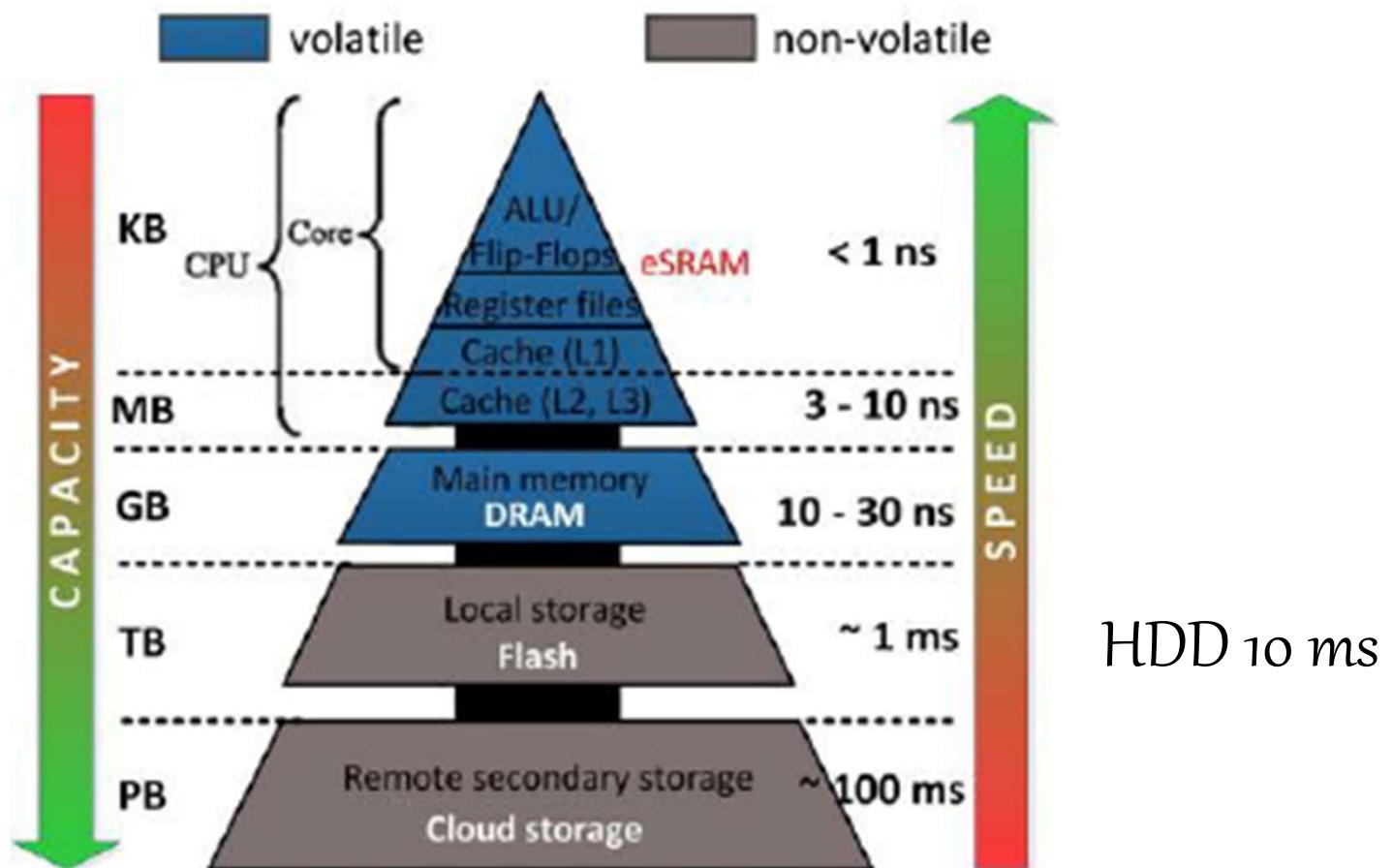
Average Access Time = Hit Time_{L1} + Miss Rate_{L1} X Penalty_{L1}

Penalty_{L1} = Hit Time_{L2} + Miss Rate_{L2} X Penalty_{L2}

Reducing Cache Miss Penalty

1. Multilevel caches
2. Critical word first & early restart
3. Victim Cache

Memory Hierarchy & Cache



Virtual Memory and Paging

- Interface between CPU, main and secondary memory.
- Cache is more of h/w, whereas VM is a s/w implementation
- This interface is managed by OS in a manner, completely transparent to the user.
- Definition:
 - Virtual memory is a hierarchical storage system of at least 2 levels managed by the OS to appear to a user as a single, large directly addressable main memory.
 - Virtual memory – Logical
 - Main memory – Physical
 - Both virtual and physical address – 32 bits – in all our examples

Virtual Memory Operation

Clip si

VIRTUAL MEMORY

Made possible by swapping pages in/out of memory

Program execution: only a portion of the program in memory at any given moment

Requires cooperation between:

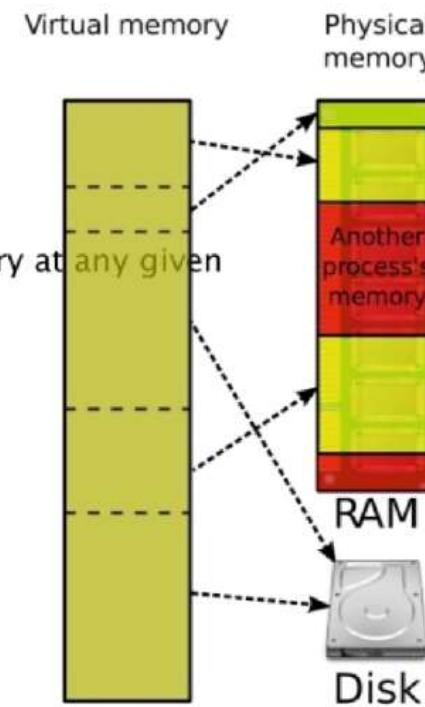
- Memory Manager: tracks each page or segment
- Processor hardware: issues the interrupt and resolves the virtual address

Advantages

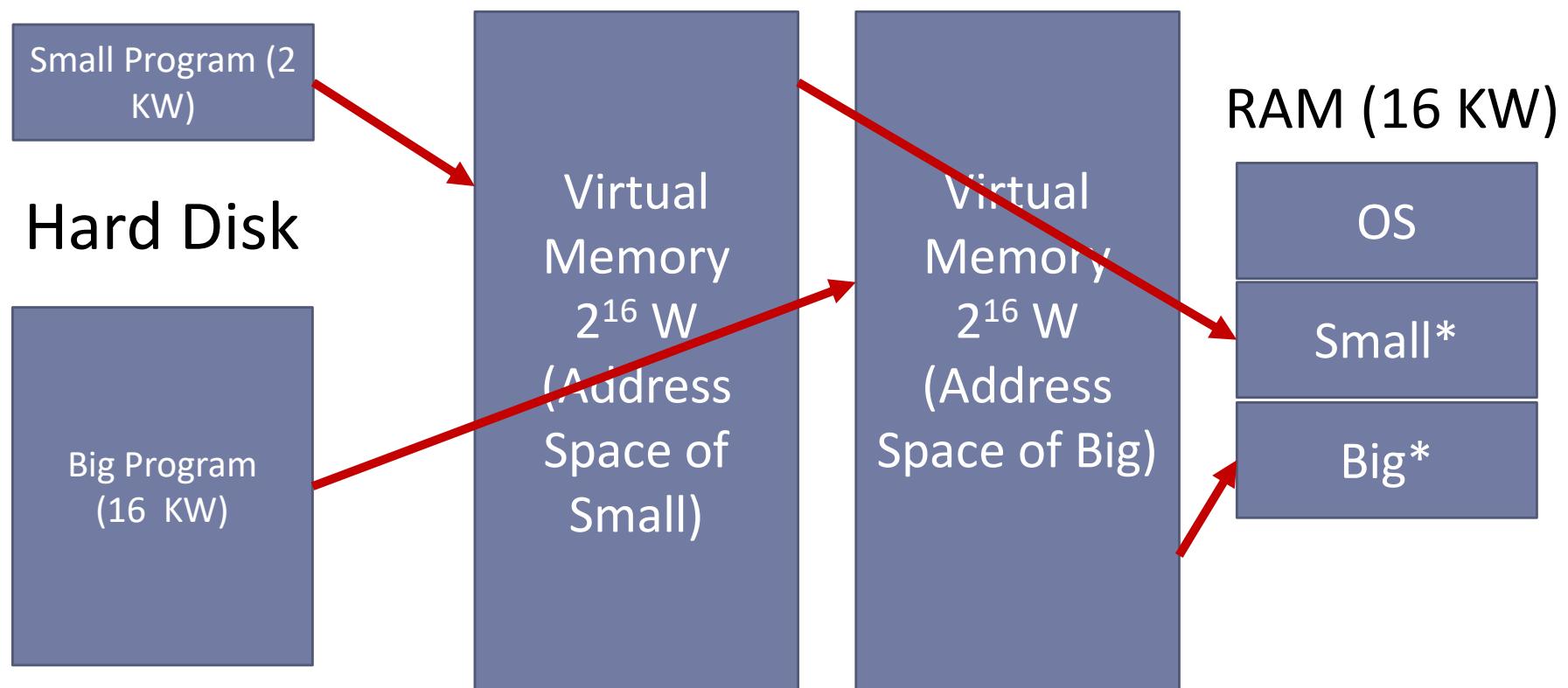
- Job size: not restricted to size of main memory
- More efficient memory use
- Unlimited amount of multiprogramming possible
- Code and data sharing allowed
- Dynamic linking of program segments facilitated

Disadvantages

- Higher processor hardware costs
- More overhead: handling paging interrupts
- Increased software complexity: prevent thrashing



Address Space of a Program in 32-bit Computer



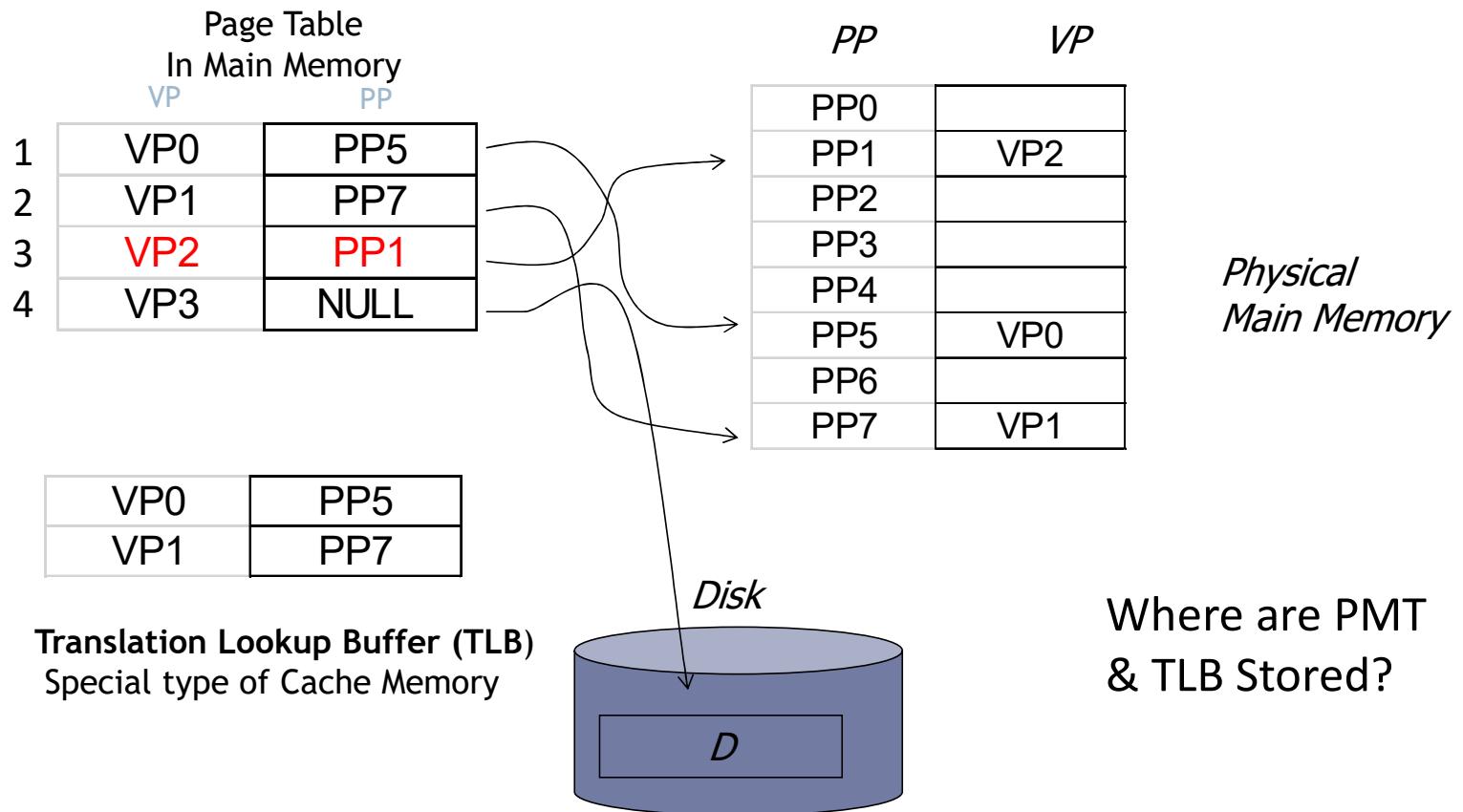
Virtual memory & paging (contd..)

- Virtual memory is divided into a number of pages.
- Typical page size is few Kbytes.
- If the page size is $1\text{K}\text{W} = 2^{10}$ Words, the 10 bits refer to the item's address within the page.
This is the offset/displacement.
- The upper 6 bits contain the virtual page address.
- The starting address of the page is called the page base address.
- The offset is the distance between page base and data item within the page being accessed.
- The offset of a word is same for both virtual and physical address.
- The base address of page is translated from **virtual** to **physical**.

Virtual Memory Mapping Operation - Associative

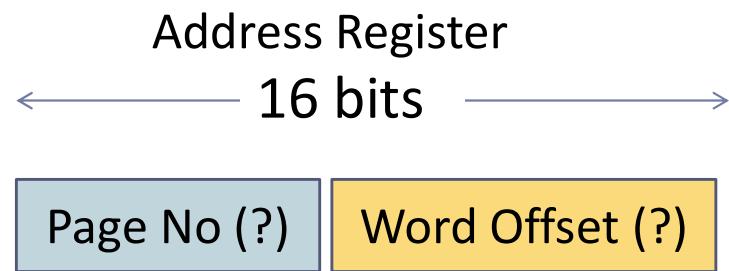
Page map
table is new. There
is no need for this
in cache!

Size of PMT
TLB is cached part
of Page Map Table

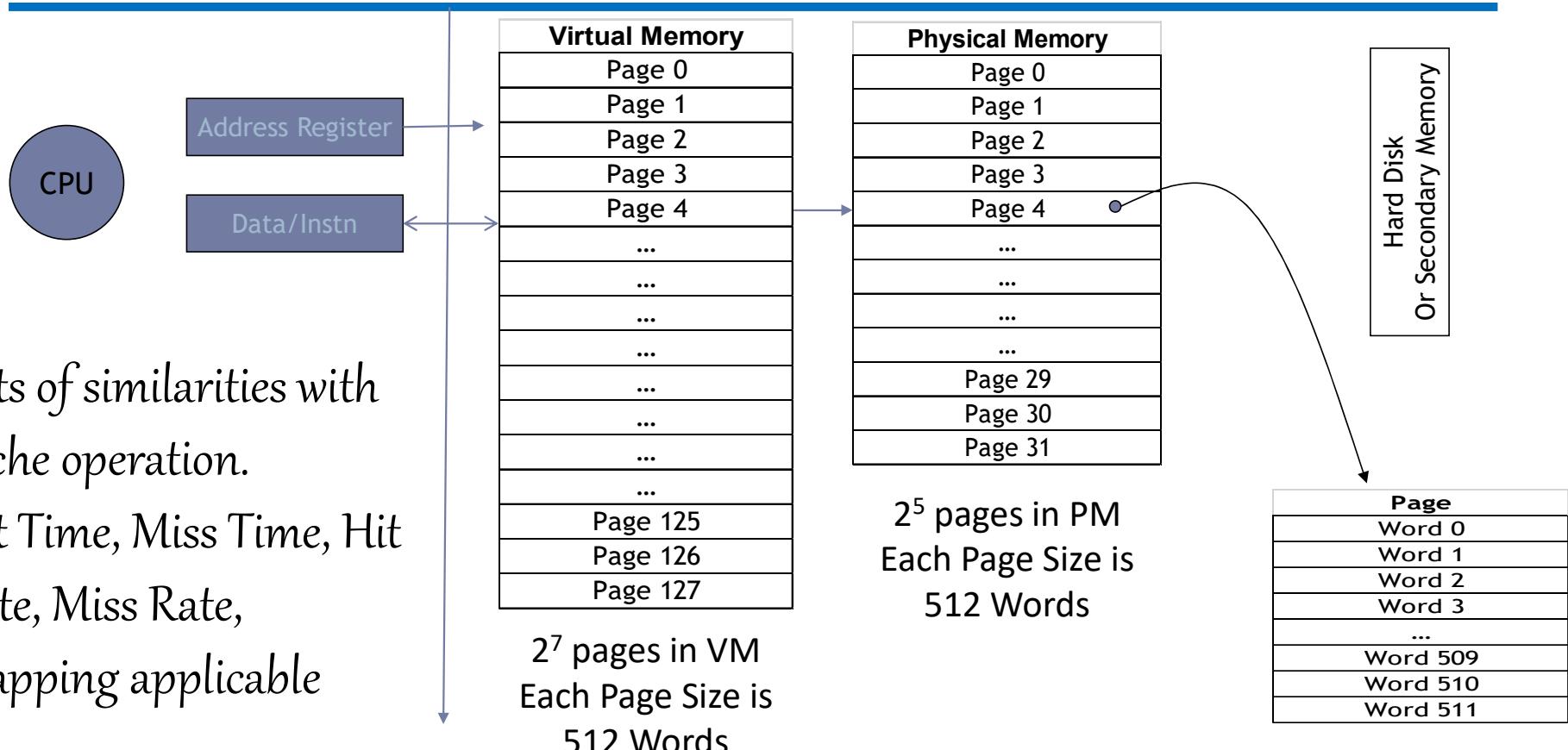


Combining Virtual Memory and Cache

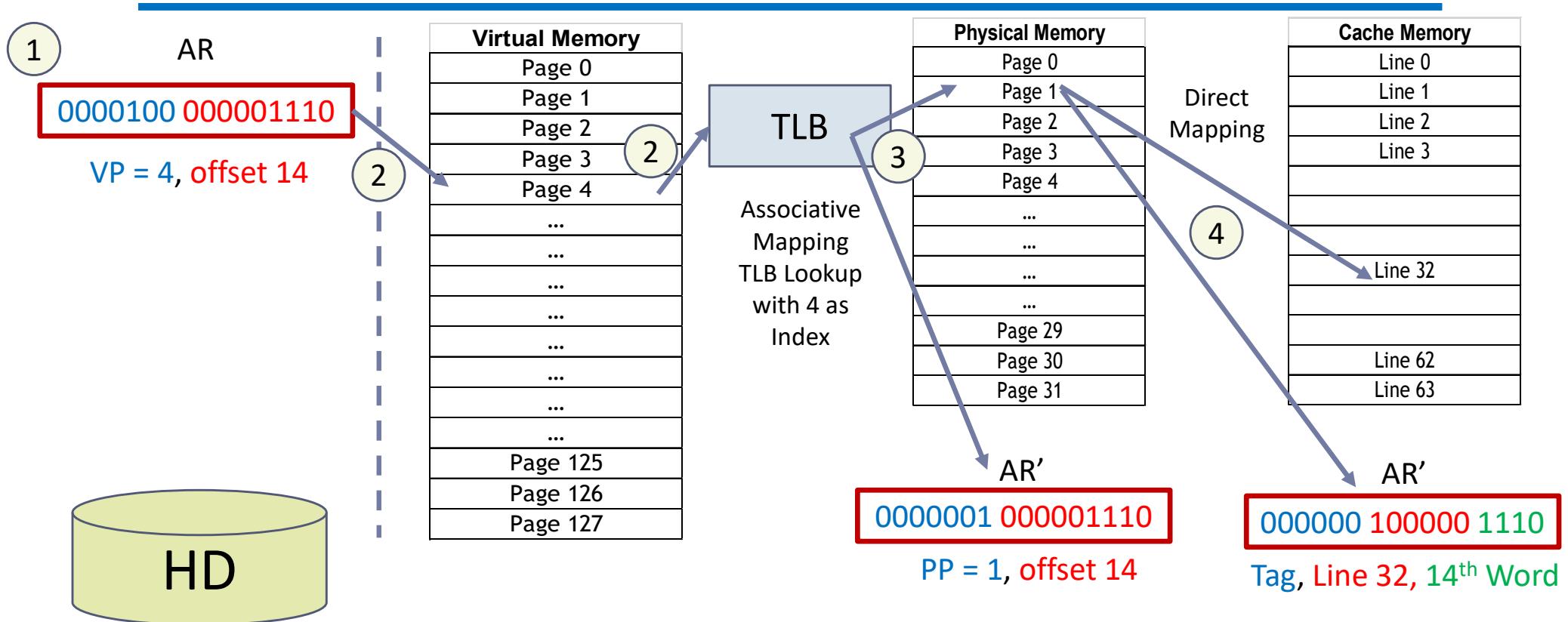
- Address Register Size = 16 bits
- Virtual Memory Size = 2^{16} words (W)
- Associative Mapped to Main Memory
- Page Size = 512 W
- Number of Virtual Pages = ?
- Main Memory Size = 16K W
- Number of Physical Pages = ?
- Cache Size = 1 K W
- Line Size = 16 W
- Cache is Direct Mapped



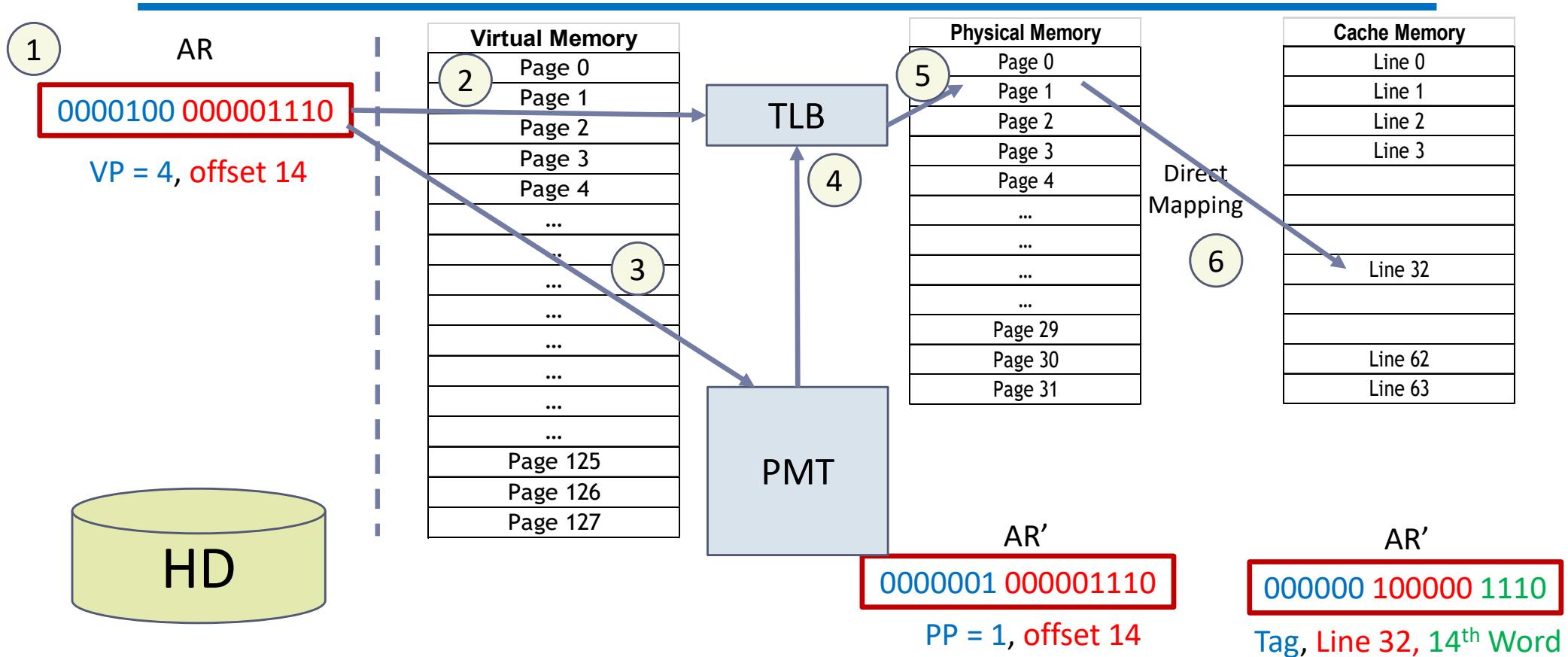
Virtual Memory System – Operation for one program



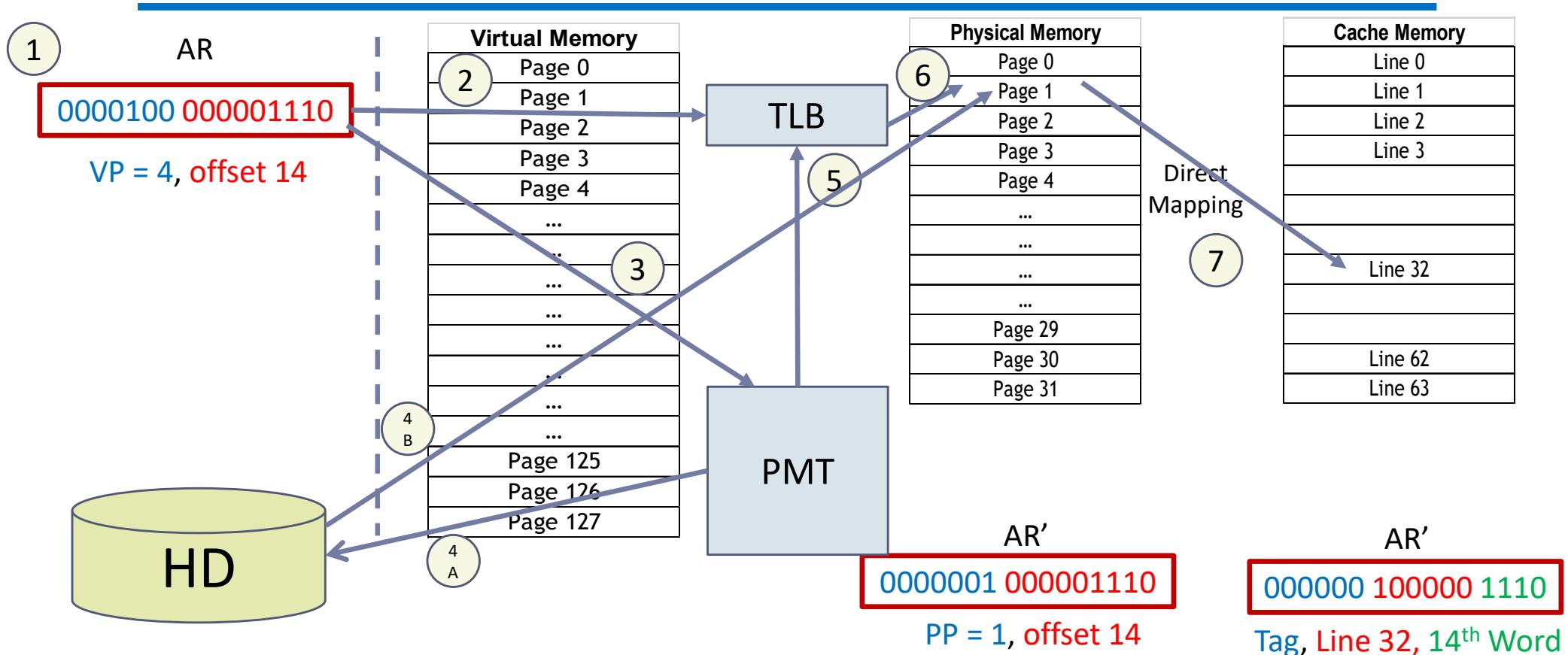
Case 1 - TLB Points to Requested Page in PM*



Case 2 - PMT Points to Requested Page in PM*



Case 3 - Requested Page Not in PM



Page Design Considerations

- HD Sector size also influences page size
 - Page size is either a factor or multiple of Sector size
- Smaller page size: Pros
 - In this case, page transfer will be faster, saving on time
 - There will be less memory space wasted in incomplete pages (when the info. is not an integral no. of pages). – Internal fragmentation
- Smaller page size: Cons
 - Page size may not be big enough to make use of locality
 - This will increase the number of misses – consequently increase the average access time

First Level Cache versus Virtual Memory

	CM	VM
Block/Page Size	16-128 B	4K-64K B
Hit Time	1-3 CC	50-150 CC
Miss Penalty	8-150 CC	1-10 M CC
Miss Rate	0.1-10%	.00001 to 0.001%

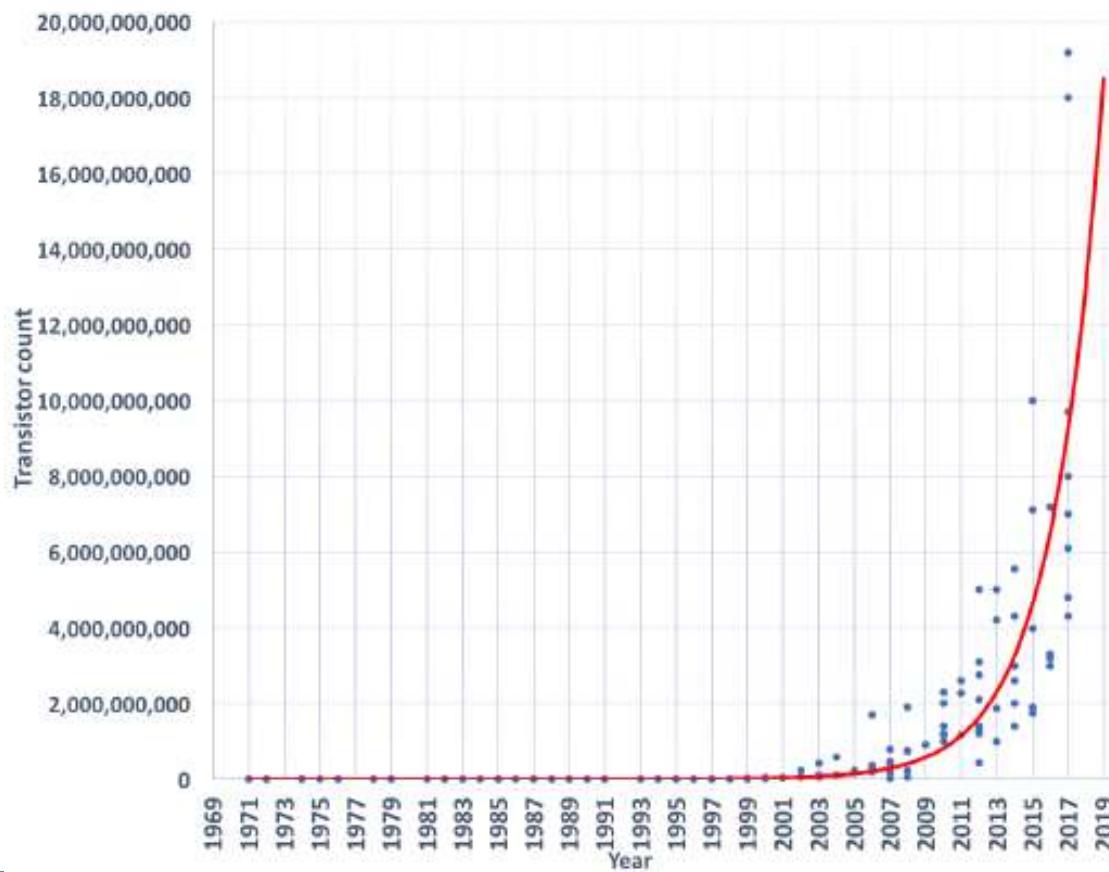
Note: All are average or typical values

Hardware Description Languages (HDL)

Exercise

- Watch the Youtube NPTEL videos by Prof. Indranil Sen Gupta: First 3 videos will do, starting from
 - <https://www.youtube.com/watch?v=lXjNLK7GC70>
 - Verilog: <http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik%2014/14%20Verilog%20Testbenches.pdf>

Moore's Empirical Law



Hardware Description Languages

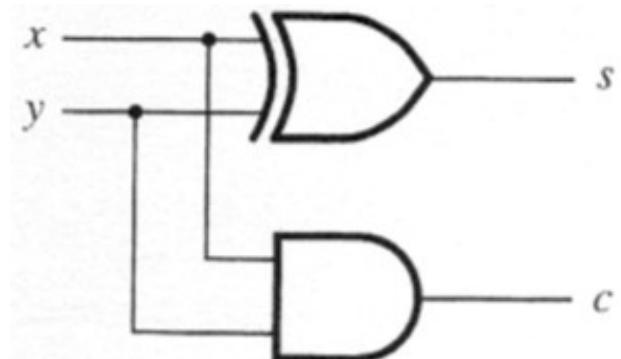
- Before HDLs were introduced, designing PCBs and ICs were manual intensive and thus error prone
- Moore's law and consequent complexity in hardware compelled the innovation of design and development tools
- A key tool in the H/W development tool chain is Hardware Description Language.
- It enabled considerable amount of automation in Chip design, development, and testing

Two Popular HDLs

- Verilog
- VHDL
- There are a number of other tools besides these two

HDL versus C

- Verilog is H/W description language – C is developed for expressing Computations.
- Verilog expresses logical blocks that work in parallel – C generally, describes sequentially executing blocks.

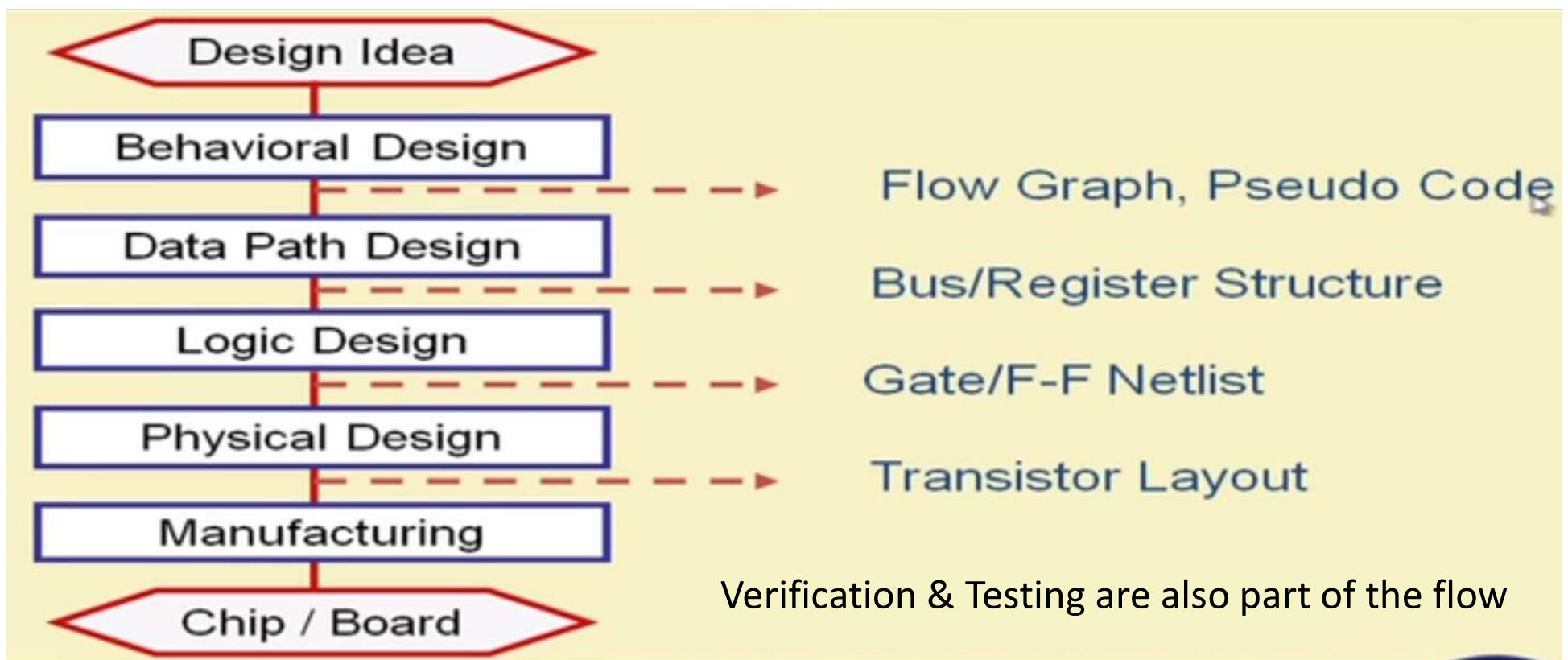


Can you code 2 functions in C such that these 2 functions are always executed in parallel every time the program is executed?

H/W Design Terminology

- Design Objectives – Optimum Area/Power/Speed
- Design Process/Flow
- Structural versus Behavioural design
- Combinational and Sequential circuits design
 - <https://www.geeksforgeeks.org/difference-between-combinational-and-sequential-circuit/>

Simplistic View of H/W Design Flow



Behavioural Design

- Specify the functionality of the design
- Other functional notations include
 - Boolean Expression, Truth Table, and FSM
- HDL specification is synthesized into more detailed specification for h/w realization

Structural Design

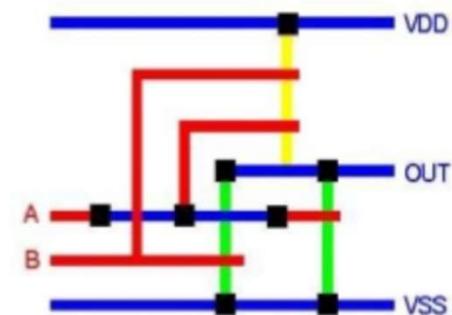
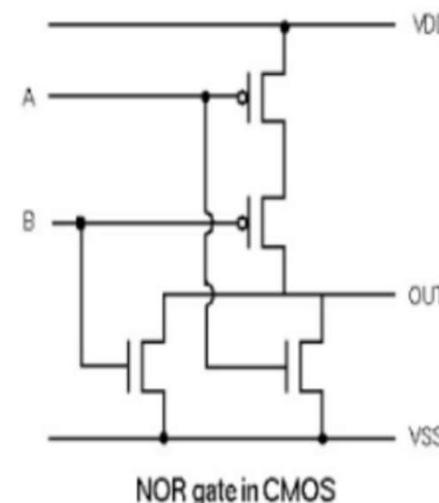
- A netlist is a directed graph, where the vertices are components and the edges are interconnections
- A netlist specification is also referred to as structural design
- A netlist can be specified at various levels, where the components may be functional modules, gates, or transistors
- The data path design we did for ALU in earlier module is a structural design at functional module (RTL) level

Logic Design

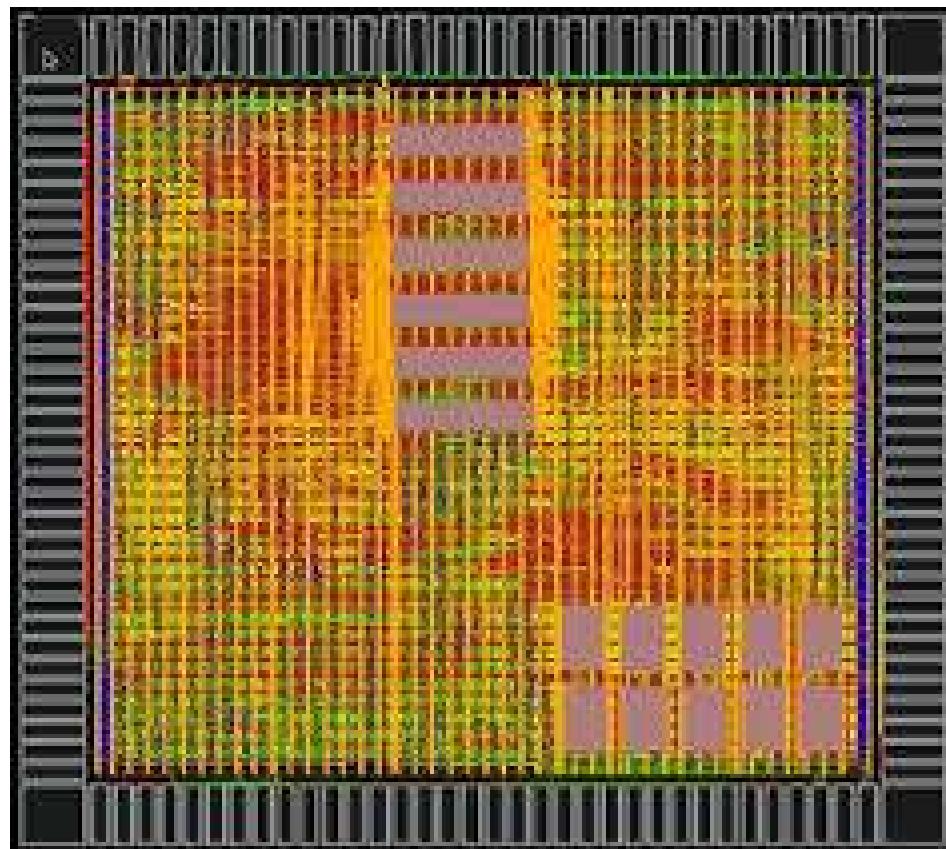
- Generate a Netlist of gates/flip-flops or standard cells
- A standard cell is a pre-designed module (like gates, flip-flops, mux, etc) at the layout level – Logisim type
- Logic Optimization is included: Number of gates, gate levels (delay), and dynamic power

Physical Design and Manufacturing

- Generate the final layout that can be sent for fab
- The layout contains a large number of regular geometric shapes



An Example of Final Layout



Installing and testing Icarus Verilog

- ❑ *Icarus Verilog* is a Verilog simulation and synthesis tool.
- ❑ Iverilog operates as a compiler, compiling source code written in Verilog into an intermediate form called *vvp assembly*.
- ❑ This intermediate form is executed by the “*vvp*” command.
- ❑ For synthesis, the compiler generates netlists in the desired format.

https://www.swarthmore.edu/NatSci/mzuckerl/e15_f2014/iverilog.html

<http://iverilog.icarus.com/>

Verilog Behavioural Specification

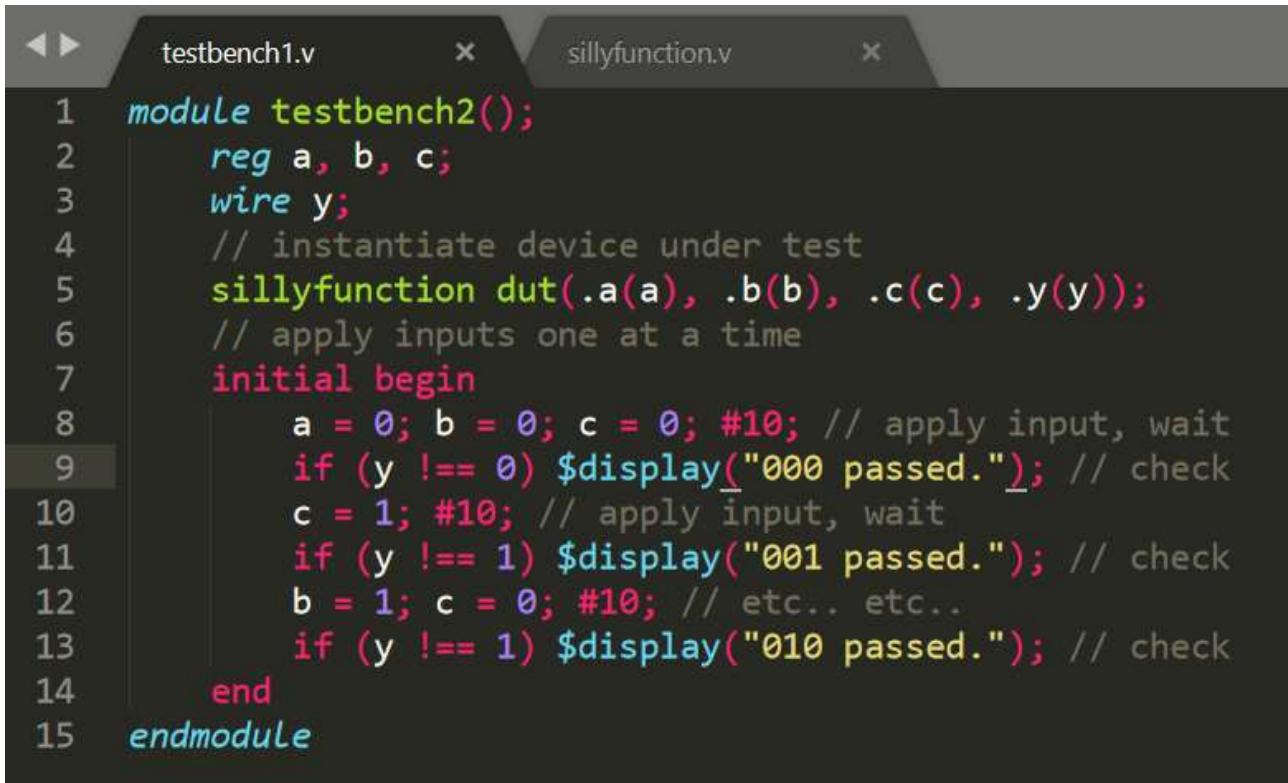
- ❑ Hello World type example:



```
testbench1.v      sillyfunction.v
1  module sillyfunction(input a, b, c, output y);
2  |   assign y = ~b & ~c | a & ~b;
3  endmodule
4
```

- ❑ Testbench Reference:
- ❑ [http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/infsec/system-security-group-dam/education/Digitaltechnik 14/14 Verilog Testbenches.pdf](http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/infsec/system-security-group-dam/education/Digitaltechnik%2014/14%20Verilog%20Testbenches.pdf)

Test Bench for Sillyfunction



```
testbench1.v      sillyfunction.v
1  module testbench2();
2      reg a, b, c;
3      wire y;
4      // instantiate device under test
5      sillyfunction dut(.a(a), .b(b), .c(c), .y(y));
6      // apply inputs one at a time
7      initial begin
8          a = 0; b = 0; c = 0; #10; // apply input, wait
9          if (y !== 0) $display("000 passed."); // check
10         c = 1; #10; // apply input, wait
11         if (y !== 1) $display("001 passed."); // check
12         b = 1; c = 0; #10; // etc.. etc..
13         if (y !== 1) $display("010 passed."); // check
14     end
15 endmodule
```

Testing Sillyfunction

- Using Icarus to build behavioural model with testbench
 - [http://iverilog.wikia.com/wiki/Getting Started](http://iverilog.wikia.com/wiki/Getting_Started)

```
C:\iverilog\bin>copy testbench1.v sillytest.v
                  1 file(s) copied.

C:\iverilog\bin>iverilog -o silly.vvp sillyfunction.v sillytest.v

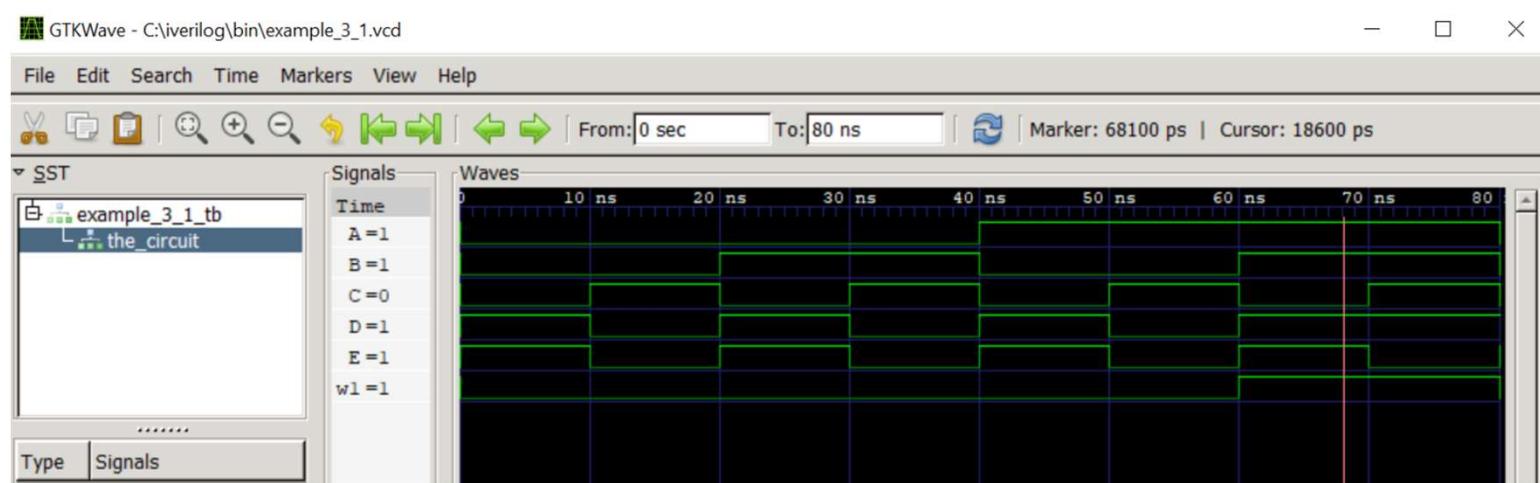
C:\iverilog\bin>vvp silly.vvp
000 passed.
001 passed.
010 passed.

C:\iverilog\bin>
```

Building with Testbench & Simulation

- Look at the examples for Simulation with gtkwave
 - <https://www.swarthmore.edu/NatSci/mzucker/e15f2014/iverilog.html>

and G1(w_1, A, B);
not G2(E, C);
Or G3(D, w_1, E);



Sublime Text Editor with Verilog Plugin

- Text Editor Download:
 - <https://www.sublimetext.com/3>
- Verilog Plugin Installation:
 - <https://packagecontrol.io/installation>

Verilog Overview

- Verilog primitives allow one to specify h/w (FPGA, ASIC, etc) behaviour design
- This design is synthesized using Verilog compiler to produce structural designs
- Verilog also allows one to specify test bench for behavioural design (non-synthesizable)
- Both concurrent and sequential execution could be specified

Learning Verilog from Code

- Verilog `wire` primitive in ideal case can take a value of 1 or 0
- Example of an AND gate specified using `Wire`

```
1  wire and_temp;
2  assign and_temp = input_1 & input_2;
```

- The 2nd line is read as "The signal `and_temp` gets `input_1` AND-ed with `input_2`."

Always Primitive

- In Verilog code below, `input_1` and `input_2` are in what is called a *sensitivity list*.
- The *sensitivity list* is a list of all of the signals that will cause the *Always Block* to execute.
- In the example above, a change on either `input_1` or `input_2` will cause the *Always Block* to execute.

```
1  always @ (input_1 or input_2)
2    begin
3      and_gate = input_1 & input_2;
4    end
```

Module Primitive

- Modules are the basic unit of Verilog models
 - Functional Description
 - Unambiguously describes module's operation
Functional, i.e., without timing information
 - Input, Output and Bidirectional ports for interfaces
 - May include instantiations of other modules
 - Allows building of hierarchy

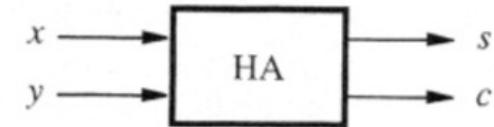
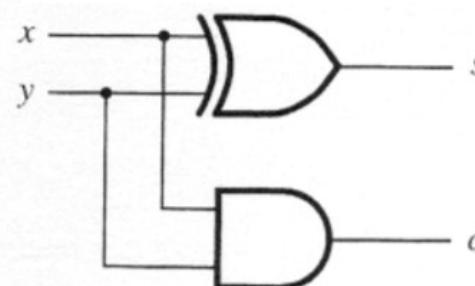
Half Adder

$$\begin{array}{r} x \\ + y \\ \hline c \ s \end{array} \quad \begin{array}{c} 0 \quad 0 \\ + 0 \quad + 1 \\ \hline 0 \ 0 \quad 0 \ 1 \end{array} \quad \begin{array}{c} 1 \quad 0 \\ + 0 \quad + 1 \\ \hline 0 \ 1 \quad 1 \ 0 \end{array}$$

Carry Sum

(a) The four possible cases

x	y	Carry	Sum
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Module Syntax

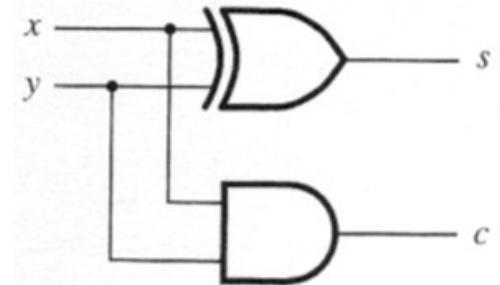
- Module declaration
 - `module ADD_HALF (s,c,x,y);`
 - Parameter list is I/O Ports
- Port declaration
 - Can be input, output or inout (bidirectional)
 - `output s,c;`
 - `input x,y;`

Gates Primitive

- Gates and interconnection
 - $\text{xor } G_1(s,x,y);$
 - $\text{and } G_2(c,x,y);$
- Verilog gate level primitive
 - Gate name
 - Internal (local) name: Instance name
 - Parameter list: Output port, input port, input port...

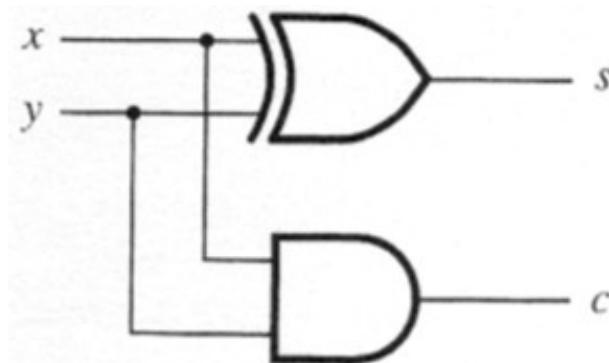
Verilog

- Verilog is a Hardware Description Language (HDL).
- HDL is a critical component of a tool chain that enables the development of all kinds of ASICs and FPGAs
- Using Verilog we can describe interconnected complex **gate level modules**



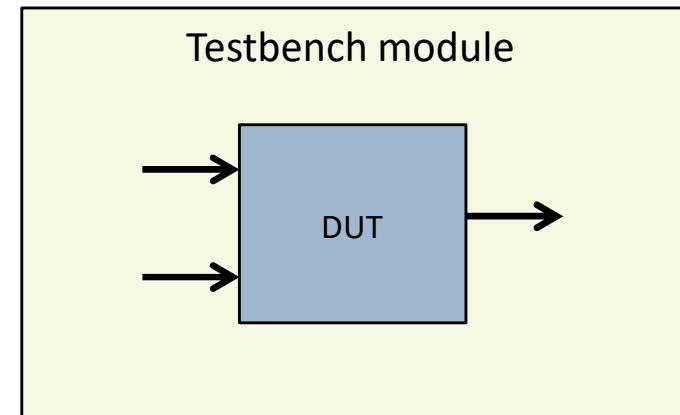
Module Description in Verilog – Example 1

```
module half_adder(s, c, x, y);
    output s, c;
    input x, y;
    xor sum(s, x, y);
    and carry(c, x, y);
endmodule //half_adder
```

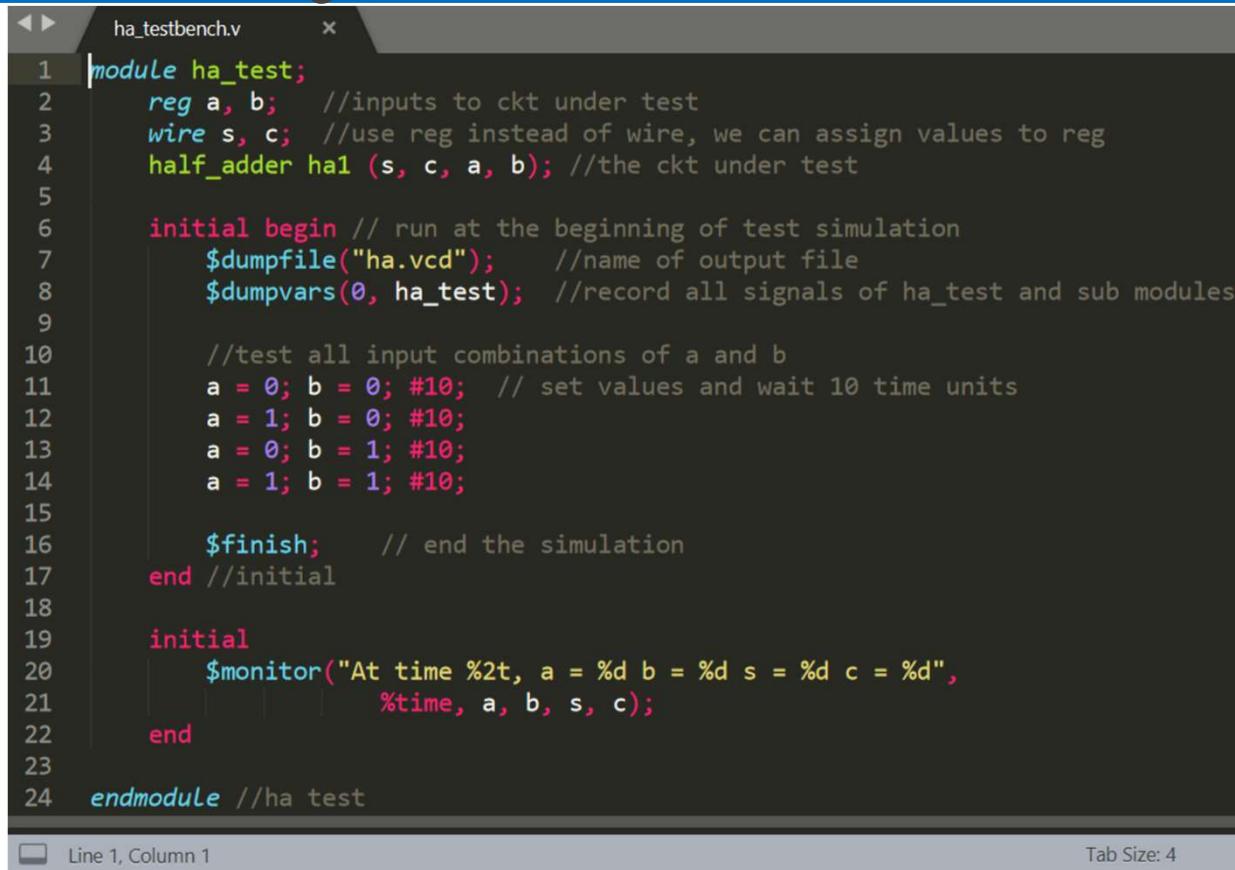


Verilog Testbench

- Testbench is a type of Verilog module that enables the testing of the designed module
- Testbench module includes the MUT (Module Under Test)



Half_adder Testbench



A screenshot of a text editor displaying Verilog code for a half-adder testbench. The code is contained in a file named 'ha_testbench.v'. The code defines a module 'ha_test' with two input registers 'a' and 'b', one output wire 's', and one output register 'c'. It uses a half-adder module 'half_adder' with inputs 's' and 'c' and outputs 'a' and 'b'. The testbench includes an initial block that runs at the beginning of simulation. This block sets up four input combinations for 'a' and 'b': (0,0), (0,1), (1,0), and (1,1). For each combination, it waits 10 time units and then calls the \$finish command to end the simulation. There is also a monitor block that logs the state of the signals 'a', 'b', 's', and 'c' at each time step.

```
1 module ha_test;
2     reg a, b; //inputs to ckt under test
3     wire s, c; //use reg instead of wire, we can assign values to reg
4     half_adder ha1 (s, c, a, b); //the ckt under test
5
6     initial begin // run at the beginning of test simulation
7         $dumpfile("ha.vcd"); //name of output file
8         $dumpvars(0, ha_test); //record all signals of ha_test and sub modules
9
10    //test all input combinations of a and b
11    a = 0; b = 0; #10; // set values and wait 10 time units
12    a = 1; b = 0; #10;
13    a = 0; b = 1; #10;
14    a = 1; b = 1; #10;
15
16    $finish; // end the simulation
17 end //initial
18
19 initial
20     $monitor("At time %2t, a = %d b = %d s = %d c = %d",
21             %time, a, b, s, c);
22 end
23
24 endmodule //ha test
```

Line 1, Column 1 Tab Size: 4

Build and Run Half_Adder with Testbench

```
cmd - Shortcut - gtkwave ha.vcd

C:\Intel\iverilog\bin>iverilog -o half_adder.vvp half_adder.v ha_testbench.v

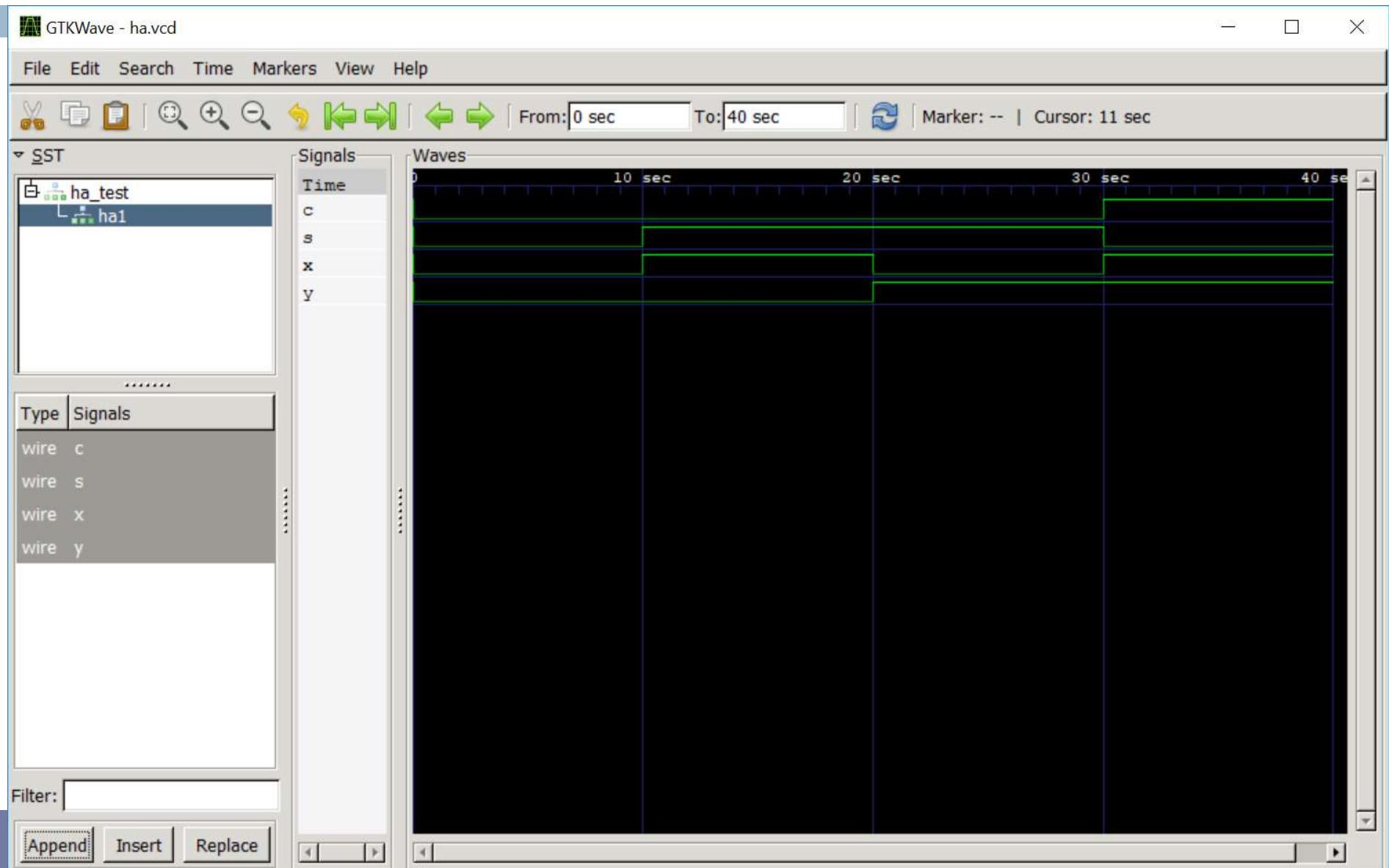
C:\Intel\iverilog\bin>vvp half_adder.vvp
VCD info: dumpfile ha.vcd opened for output.
At time 0, a = 0 b = 0 s = 0 c = 0
At time 10, a = 1 b = 0 s = 1 c = 0
At time 20, a = 0 b = 1 s = 1 c = 0
At time 30, a = 1 b = 1 s = 0 c = 1

C:\Intel\iverilog\bin>copy ha.vcd ../gtkwave/bin/ha.vcd
The syntax of the command is incorrect.

C:\Intel\iverilog\bin>copy ha.vcd c:\Intel\iverilog\gtkwave\bin\ha.vcd
Overwrite c:\Intel\iverilog\gtkwave\bin\ha.vcd? (Yes/No/All): Yes
1 file(s) copied.

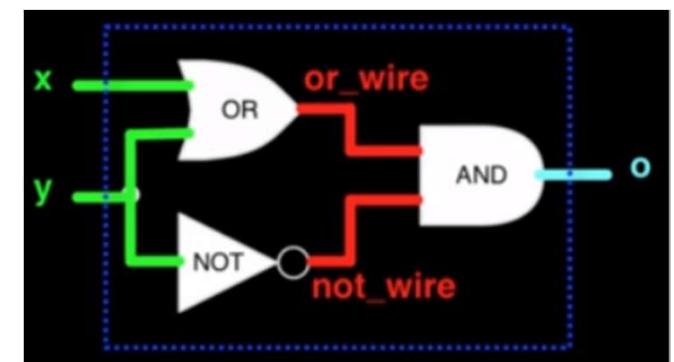
C:\Intel\iverilog\bin>cd ..\gtkwave\bin

C:\Intel\iverilog\gtkwave\bin>gtkwave ha.vcd
```



Module Description in Verilog – Example 2

```
test_testbench.v      test.v
1 module test(out, x, y);
2     input x, y;
3     output out;
4
5     or or1(wire1, x, y);
6     not not1(wire2, y);
7     and and1(out, wire1, wire2);
8
9 endmodule //test |
```



Build and Test Example 2

```
cmd - Shortcut - gtkwave test1.vcd
C:\Intel\iverilog\bin>iverilog -o test.vvp test.v test_testbench.v

C:\Intel\iverilog\bin>vvp test.vvp
VCD info: dumpfile test1.vcd opened for output.
At time 0, a = 0 b = 0 out = 0
At time 10, a = 1 b = 0 out = 1
At time 20, a = 0 b = 1 out = 0
At time 30, a = 1 b = 1 out = 0

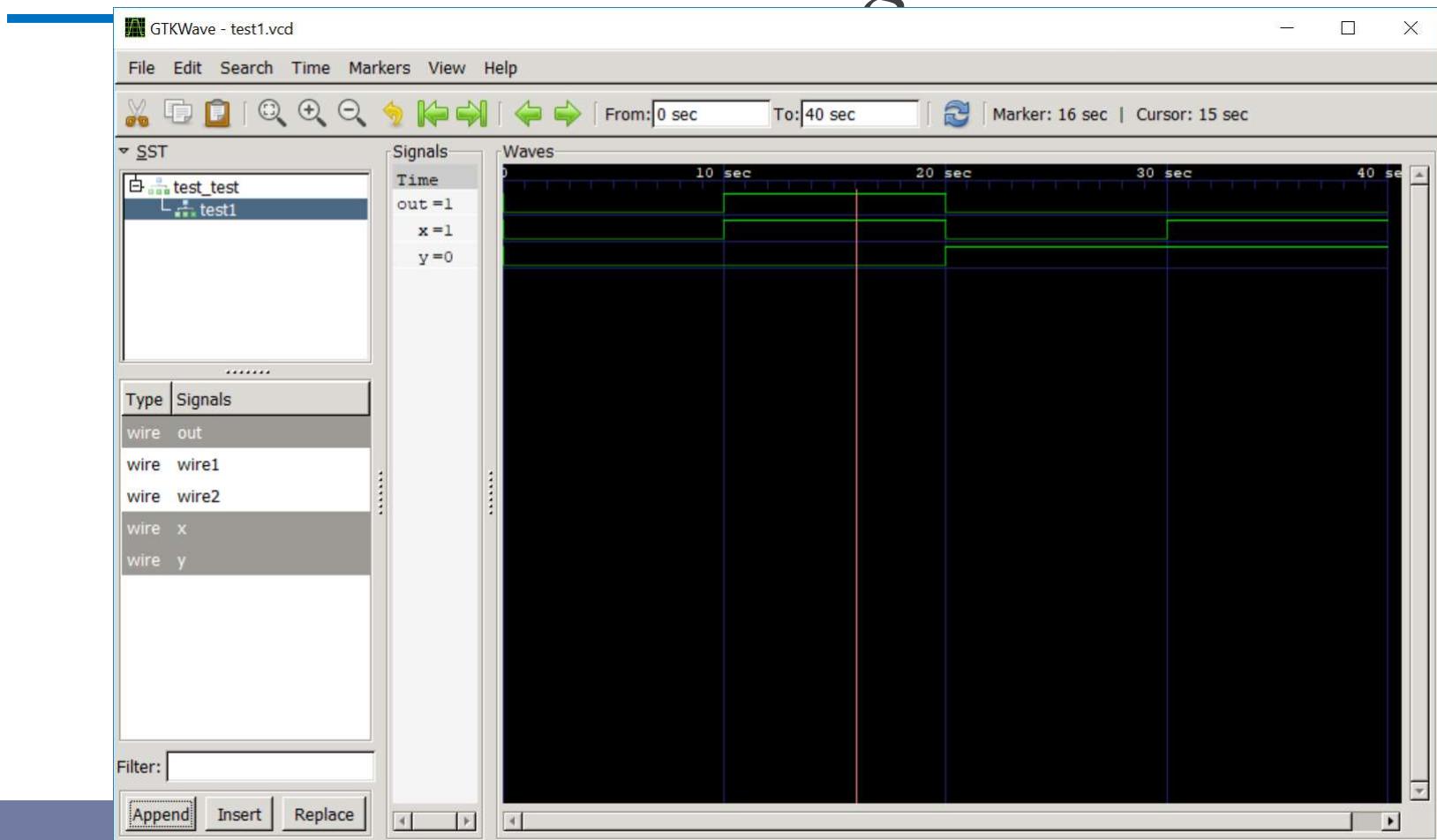
C:\Intel\iverilog\bin>copy test1.vcd C:\Intel\iverilog\gtkwave\bin\test1.vcd
1 file(s) copied.

C:\Intel\iverilog\bin>cd ../

C:\Intel\iverilog>cd gtkwave/bin

C:\Intel\iverilog\gtkwave\bin>gtkwave test1.vcd
GTKWave Analyzer v3.3.71 (w)1999-2016 BSI
```

Timing



Input and Output, Hard Disk, and RAID

Disk Storage

- Disk storage slower than Memory.
- Disk offers better volume at a lower unit price.
- Nicely fit into the Virtual Memory concept.
- Critical piece of the performance puzzle.

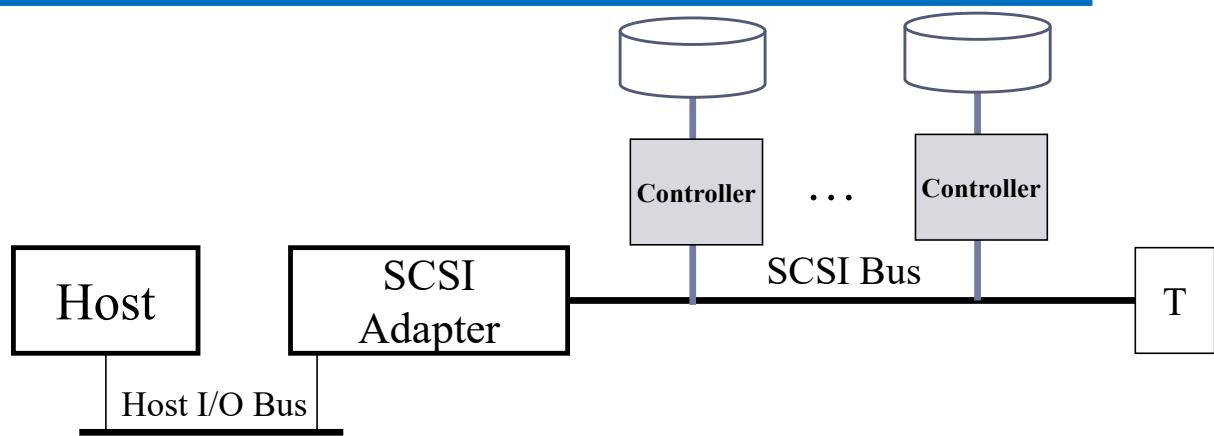


Courtesy:

www.shortcourses.com/choosing/storage/06.htm

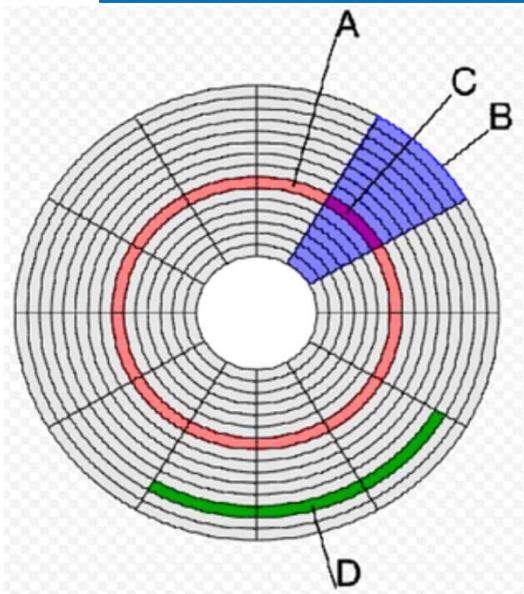
Disk System

- Disk pack
- Cache & Disk
- Embedded Controller
- SCSI Host Adapter
- OS (kernel/driver)
- User Program

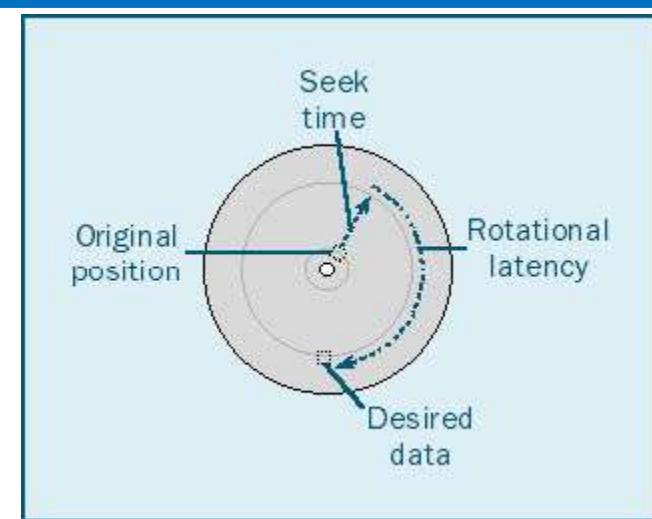


- Controller can take order to retrieve/store data
- Controller uses its buffer to store data
- Controllers can work in parallel
- Controller is an embedded processor

Magnetic Disk



Disk Pack



A : Track

B : Geometrical Sector

C: Track Sector

D: Cluster

$\text{Disk Access Time} = \text{Rotational Delay} + \text{Seek Time} + \text{Transfer Time}$

Components of Disk Access Time

- ❑ Seek Time
- ❑ Rotational Latency
- ❑ Internal Transfer Time
- ❑ Other Delays
- ❑ That is,

Avg Access Time = Avg Seek Time + Avg Rotational Delay + Transfer Time +
Other overhead

Problem

- Seek Time = 5 ms/100 tracks
- Transfer rate 40 MB/sec
- Sector Size = 2 KB

RPM = 10000 RPM

Other Delays = 0.1 ms

Average Access Time

$$\begin{aligned} &= \text{Average Seek Time (5ms)} + \\ &\quad \text{Average Rotational Delay (time for 1/2 revolution)} + \\ &\quad \text{Transfer Time } (2048/(40 \times 10^6)) + \\ &\quad \text{Other overhead (0.1 ms)} \\ &= 5 + 10^3 \times 60 / (2 \times 10000) + 2048 \times 10^3 / (40 \times 10^6) + 0.1 \text{ ms} \\ &= 5 + 3 + 51.2 / 10^4 + 0.1 \text{ ms} = 5 + 3 + 0.0512 + 0.1 = 8.1512 \text{ ms} \end{aligned}$$

RAID

- RAID uses a number of disks instead of one to improve
 - Throughput
 - Reliability
 - Data Availability
- RAID stands for Redundant Array of Individual Disks
- There are a number of configurations offering different performance features
- These configurations are identified by their number: Example: RAID 1 offers reliability

Terminologies

- Reliability and data Availability
- Standby versus Load Balancing
- Striping
- Shadowing/Mirroring
- Parity
- Error Correction Encoding

Reliability - Terminologies

- *Availability* is a measure of the *reliability* of a system & it is a statistical measure
- It is treated like *probability* for computation and it is expressed using the following two statistical terms:

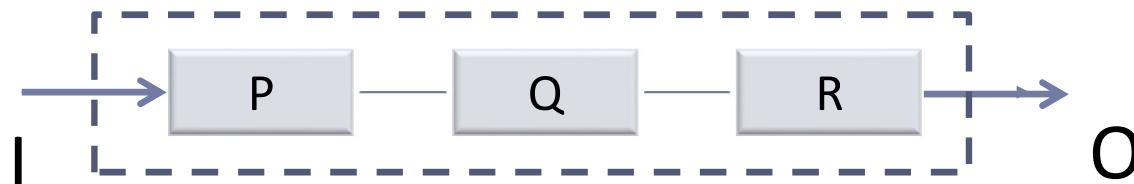
Mean Time Between Failure (MTBF) – Ex: 297 days

Mean Time To Recover (MTTR): Ex: 3 days

$$R = \text{MTBF}/(\text{MTBF}+\text{MTTR}) \text{ & } R \leq 1$$

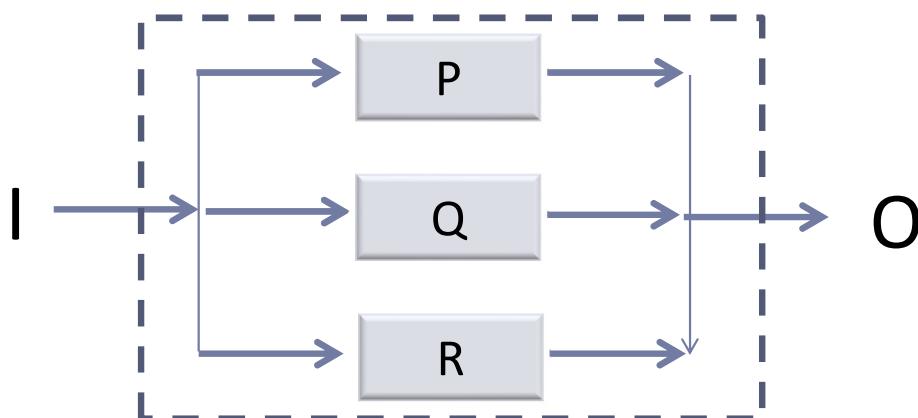
$$R = 297/(297+3) = 0.99$$

Reliability of Complex Systems



$$R_s = R_P \times R_Q \times R_R$$

Cascaded Components/Systems



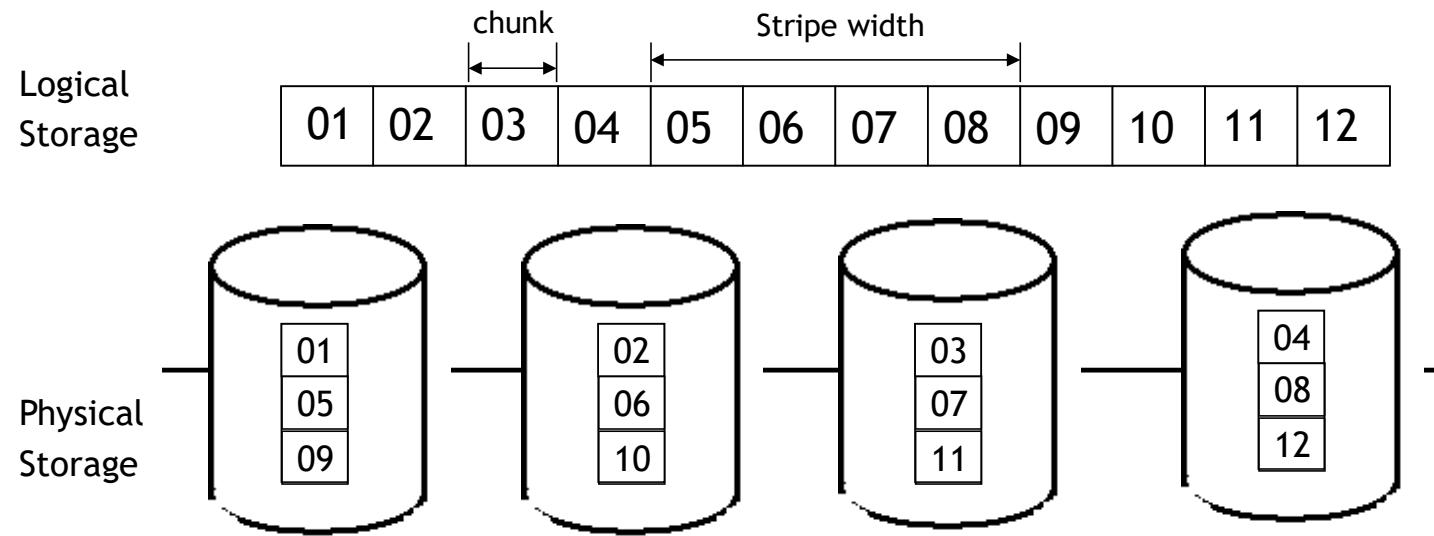
$$R_s = 1 - (1-R_P) \times (1-R_Q) \times (1-R_R)$$

Standby Components/Systems

Fault Tolerant Computing

- Hardware Redundancy
 - Load Balancing with N systems
 - Standby - M Standby for N ($M \ll N$)

RAID-0 Striping



n number of independent disks

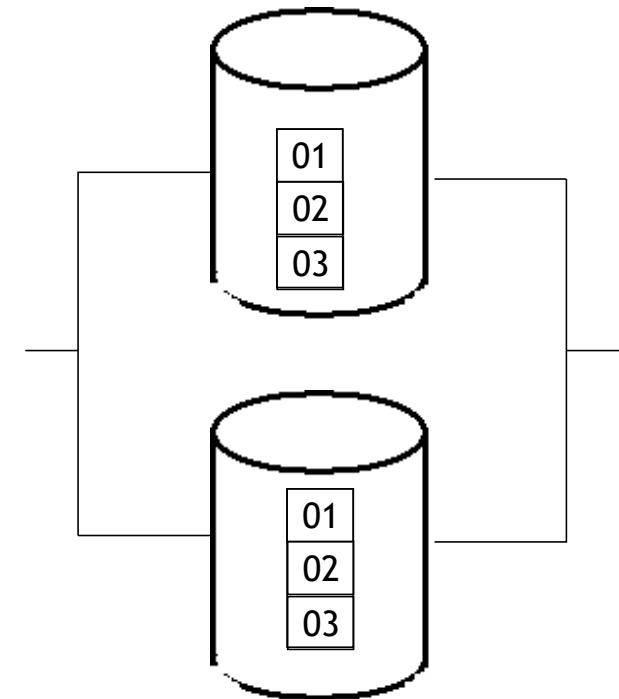
Typical Stripe size values: 16, 32, 64, 128, 256K

RAID-0 Performance

- Throughput: best case - nearly $n \times$ single disk value
- Utilization: worst case – nearly $(1/n) \times$ single disk value
- Space Efficiency is 1
- Data Reliability: $(r)^n$, where r is the reliability of a disk, ($r \leq 1$).
- Sequential Access: Fast
- Random Access: Multithreaded Random Access offers better performance.
- When $r = 0.8$ and $n = 2$, reliability is 0.64

RAID-1 Mirroring

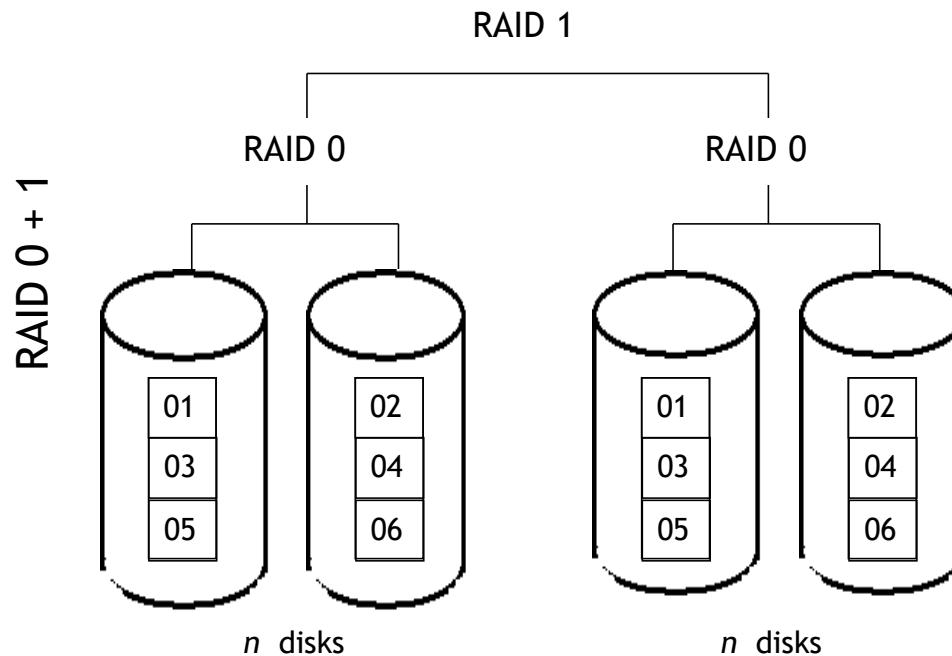
- Issue Addressed: RAID-0's Reliability Problem.
- Shadowing or mirroring is used.
- Data is not lost when a disk goes down.
- One or more disks can be used to mirror primary disk.
- Writes are posted to primary and shadowing disks.
- Read from any of them.



RAID-1 Performance

- Reliability is improved with mirroring:
 - $(1 - (1-r)(1-r))$
- Example: when r is 0.8, the reliability of RAID-1 is .96.
- Writes are more complex – must be committed to primary and all shadowing disks.
- Writes much slower due to atomicity requirement.
- Expensive – due to 1-to-1 redundancy.
- Throughput: Same as single disk value
- Utilization: $\frac{1}{2}$ of single disk value; Space efficiency = $\frac{1}{2}$ (assuming 1 to 1 mirror)

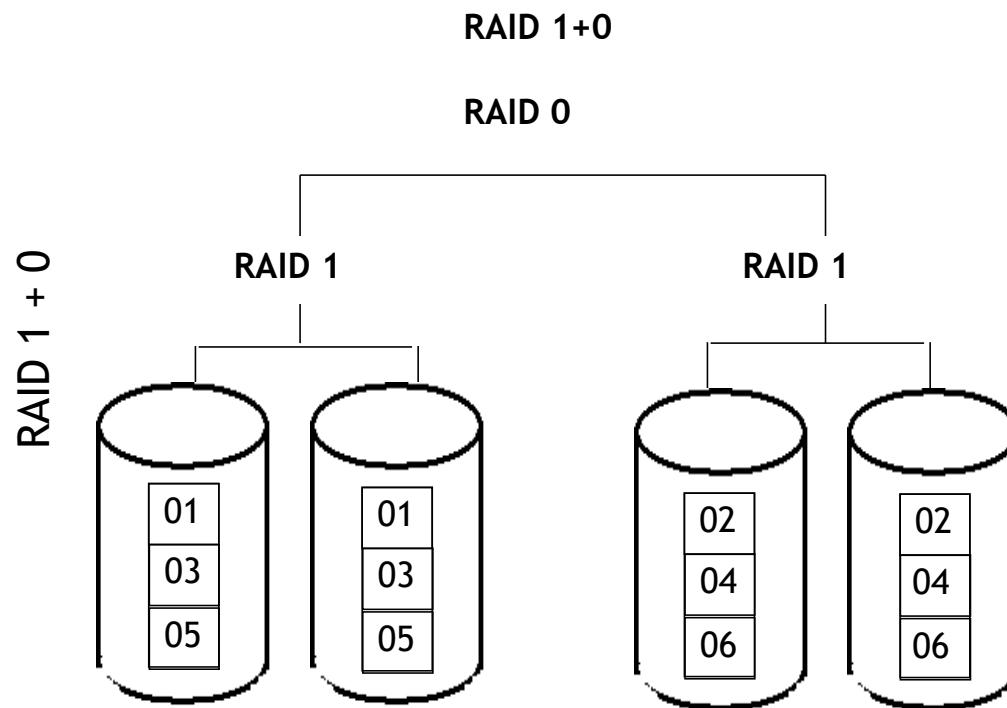
RAID 0+1 -Striping & Mirroring



Performance RAID-0+1

- Let the Reliability of a RAID 0 sub-tree be R' :
 - Then the reliability of RAID 1 tree = $1 - (1-R')(1-R')$
- Reliability R' is:
 - $R' = r^2$ (reliability of a single disk is r):
- Throughput is same as RAID-0, however with $2 \times n$ disks
- Utilization is lower than RAID-0 due to mirroring
- “Write” is marginally slower due to atomicity
- When $r = 0.9$, $R' = 0.81$, and $R = 1 - (0.19)^2 = .96$

RAID 1+0 - Mirroring & Striping



Performance RAID-1+0

- Let the Reliability of a RAID 1 sub-tree be R' :
 - Then the reliability of RAID 0 tree = $(R')^2$
- Reliability R' is:
 - $R' = 1 - (1-r)^2$ (reliability of a single disk is r):
- Throughput is same as RAID-0, however with $2 \times n$ disks
- Utilization is lower than RAID-0 due to mirroring
- “Write” is marginally slower due to its atomicity
- When $r = 0.9$, $R' = 0.99$, and $R = (0.99)^2 = .98$

RAID-2 Hamming Code Arrays

- Low commercial interest due to complex nature of Hamming code computation.
- However, Hamming code is used to make RAM more robust
- The next few slides present Hamming code for robust data

Hamming Error Correcting Codes

- Hamming Error Correcting code is used in RAM - not likely in networking or in disk
- k parity bits are added to an n -bit data word, forming a new word of $n + k$ bits.
- The bit positions are numbered in sequence from 1 to $n + k$.
- Those positions numbered with powers of 2 are used for the parity bits.
- The remaining bits are the data bits.

An Illustration

Data bits ($n=8$) 11000100 with 4-bit (k) parity

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
	P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0

The 4 parity bits P_1 through P_8 are in positions 1, 2, 4, and 8, respectively. The 8 bits of the data word are in the remaining positions. Each parity bit is calculated as follows:

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 6, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

XOR of Data
bits only

Data Written to Memory is: 0011 1001 0100

Locating Error

When the 12 bits are read from memory, they are checked again for errors. The parity of the word is checked over the same groups of bits, including their parity bits. The four check bits are evaluated as follows:

$$\left. \begin{array}{l} C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11) \\ C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11) \\ C_4 = \text{XOR of bits } (4, 5, 6, 7, 12) \\ C_8 = \text{XOR of bits } (8, 9, 10, 11, 12) \end{array} \right\} \begin{matrix} \text{XOR of data \& Parity} \\ \text{bits} \end{matrix}$$

$$C_8 C_4 C_2 C_1 = N$$

If $N \neq 0$ then value of N points to the position of the error

Position of either data or parity bit

Questions for Generalization

1. How to compute parity bit values?
2. How to choose the positions for parity bits?
3. How to locate and correct an error
4. What is the relationship between C_i and P_i ?
5. How to choose k ?
 - What is the value of k for $n = 1, 2, 3, 4, 5, 6$, and 7 ?

Correcting Error

Bit position	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	1	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	0	0	0	1	0	1	0	0	Error in bit 5

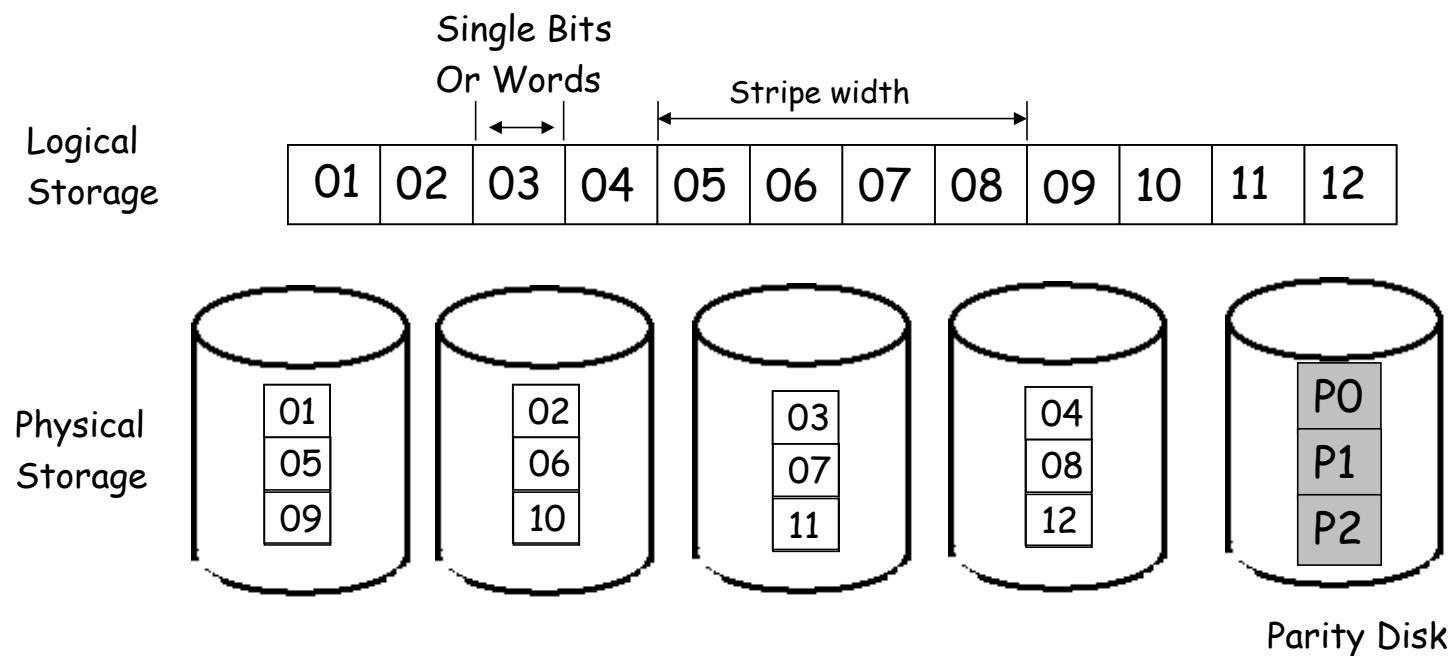
Good enough for multi-bit errors!

Hamming code extended to find and correct burst errors

K for 1 bit data

- Note: number of Cs equal to k
- The condition to be met is the number of Cs should be able to represent all the values from 0 to $k + n$; where $n = 1$
- If we choose $k = 1$ we have only one C
- This one C is NOT sufficient to code 0, 1 and 2
- k needs to be 2. This k can code (0 to $k + n$) – 0, 1, 2, 3
- In general, 2^k should be just greater than $k + n + 1$
- For $n = 2 \& 3$, k is 3

RAID-3 Byte Striping with Parity bit



RAID-3 Operation

- Based on the principle of reversible form of parity computation.
- Where Parity $P = C_0 \oplus C_1 \oplus \dots C_{n-1} \oplus C_n$
- Missing Stripe $C_m = P \oplus C_0 \oplus C_1 \oplus \dots C_{m-1} \oplus C_{m+1} \oplus \dots C_{n-1} \oplus C_n$

RAID-3 Performance

- RAID-1's 1-to-1 redundancy issue is addressed by 1-for-n Parity disk. Less expensive than RAID-1
- Rest of the performance is similar to RAID-0.
- This can withstand the failure of one of its disks.
- Reliability = all the disks are working + exactly one failed

$$= r^n + n_{c_1} r^{n-1} \cdot (1-r)$$

- When $r = 0.9$ and $n = 5$

$$= 0.9^5 + 5 \times 0.9^4 \times (1 - 0.9)$$

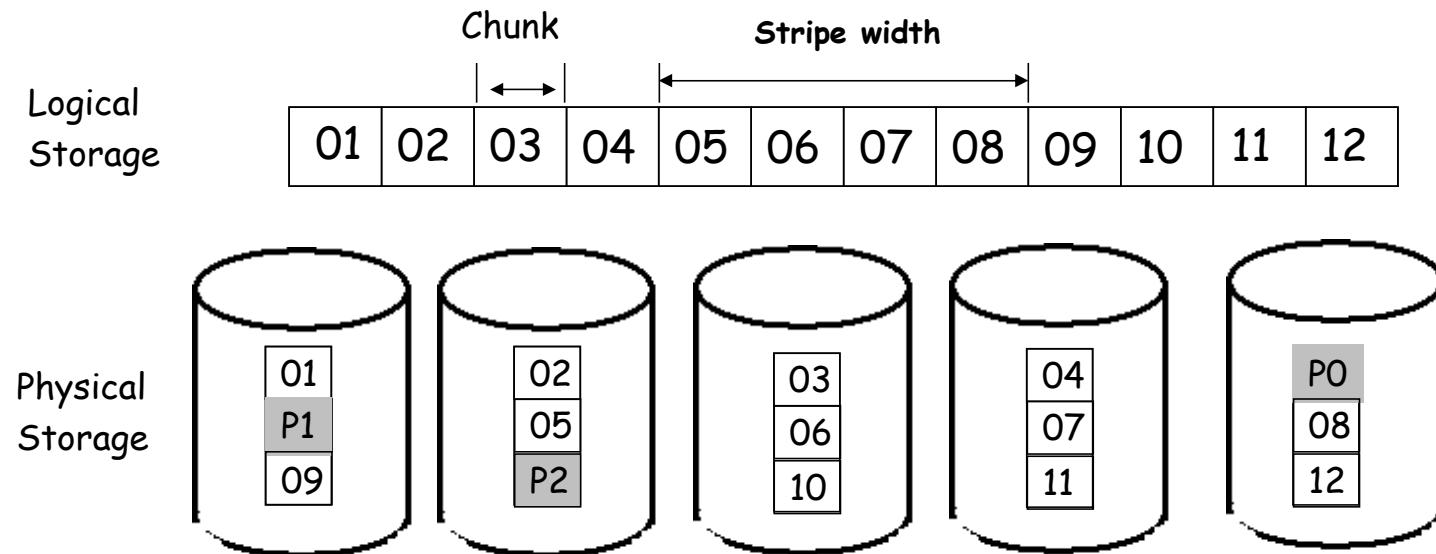
$$= 0.6 + 5 \times 0.66 \times 0.1$$

$$= 0.6 + 0.33 = .93$$

RAID-4 Performance

- Similar to RAID-3, but supports larger chunks.
- Performance measures are similar to RAID-3.

RAID-5 (Distributed Parity)



RAID-5 Performance

- ❑ In RAID-3 and RAID-4, Parity Disk is a bottleneck for Write operations.
- ❑ This issue is addressed in RAID-5.

Multiprocessor-Architecture

Multiprocessor System

- Multiprocessor systems are developed for solving complex problems like weather forecast.
- Multiprocessor system includes cooperating multiple processors
- The cooperation is enabled either by having *shared memory* (tight coupling) *or message passing* (loose coupling).

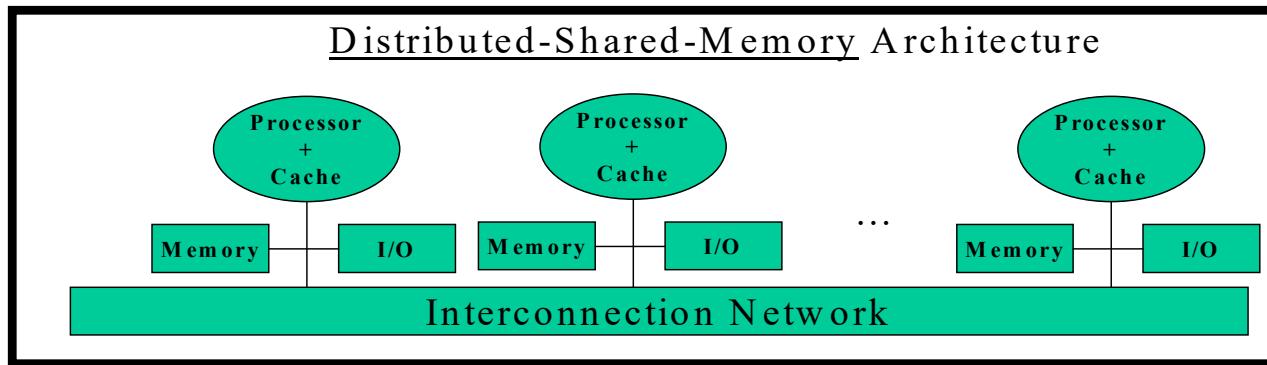
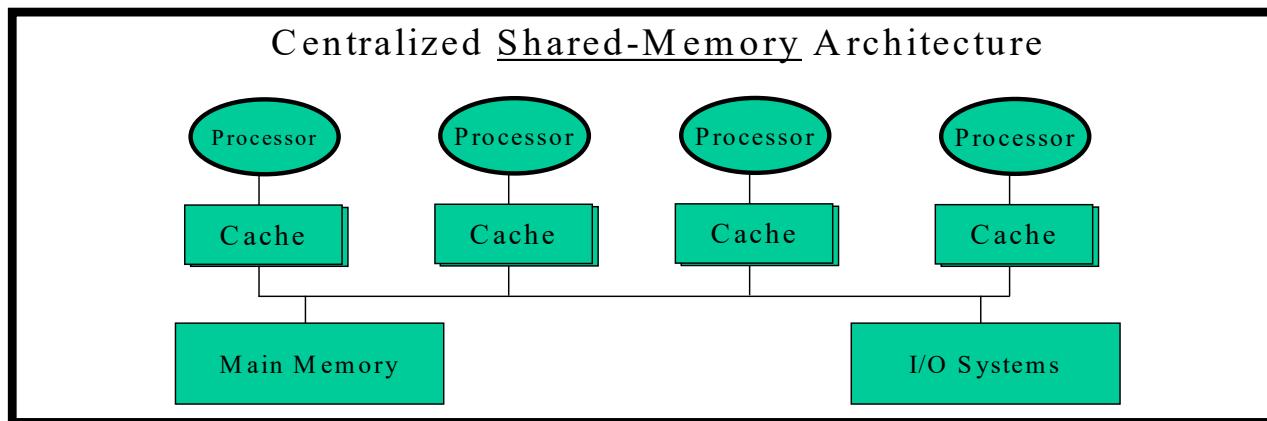
Flynn's Multiprocessor Taxonomy

- *Flynn's taxonomy* of Single and Multi-processor systems:
 - Single Instruction, Single Data stream (**SISD**)
 - Single processor
 - Single Instruction, Multiple Data streams (**SIMD**)
 - Data Level Parallelism (DLP)
 - Multiple Instruction, Single Data stream (**MISD**)*
 - Reliable systems
 - Multiple Instruction, Multiple Data streams (**MIMD**)
 - Multicore Superscalar processors

Classification of MIMD Architectures

- Shared Memory:
 - Centralized shared memory architecture OR Symmetric (shared-memory) Multiprocessors (SMP) OR Uniform Memory Access (UMA).
 - Distributed shared memory architecture OR Non-Uniform Memory Access (NUMA) architecture.
- Message Passing:
 - Multiprocessor Systems based on messaging
 - Slower but cleaner modular compared to Shared memory design

Two types of Shared Memory architectures



Cache Coherence Problem

- Cache coherence is a problem in multi-processor system with shared data

Time	Event	Cache	Cache	Memory
		Contents for	Contents for	contents for
	CPU A	CPU B	Location X	
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 in X	0	1	0

Coherency and Consistency

- Often processors share data in a closely coupled systems like MIMD
- *Coherency* is special type of *consistency*
- *Coherency* defines the consistency of reads and writes to the same memory location while the term *consistency* defines the behavior of reads and writes with respect to two or more different memory locations

Solution

- There are two solutions:
 - **Snooping**: READ/WRITE request over memory bus is monitored by cache controller of each processor. Solution uses this monitored information
 - **Directory based**: A directory (centralized/distributed) maintains the status of each block. Solution (Controller) uses this directory along with additional messaging

Snooping Protocol

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Invalidation of Snooping Protocol

Processor Activity	Bus Activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory contents for Location X
CPU A reads X (block)	Cache Miss for X	0		0
CPU B reads X (block)	Cache Miss for X	0	0	0
A Writes 1 to X	Invalidation for X	1		0
B reads X	Cache Miss for X	1	1	1

Write Broadcast of Snooping Protocol

Processor Activity	Bus Activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory contents for Location X
				0
CPU A reads X (block)	Cache Miss for X	0		0
CPU B reads X (block)	Cache Miss for X	0	0	0
A Writes 1 to X	Write Broadcast for X	1	1	1
B reads X		1	1	1

Lot more to CC

- Look for this in Computer Architecture!

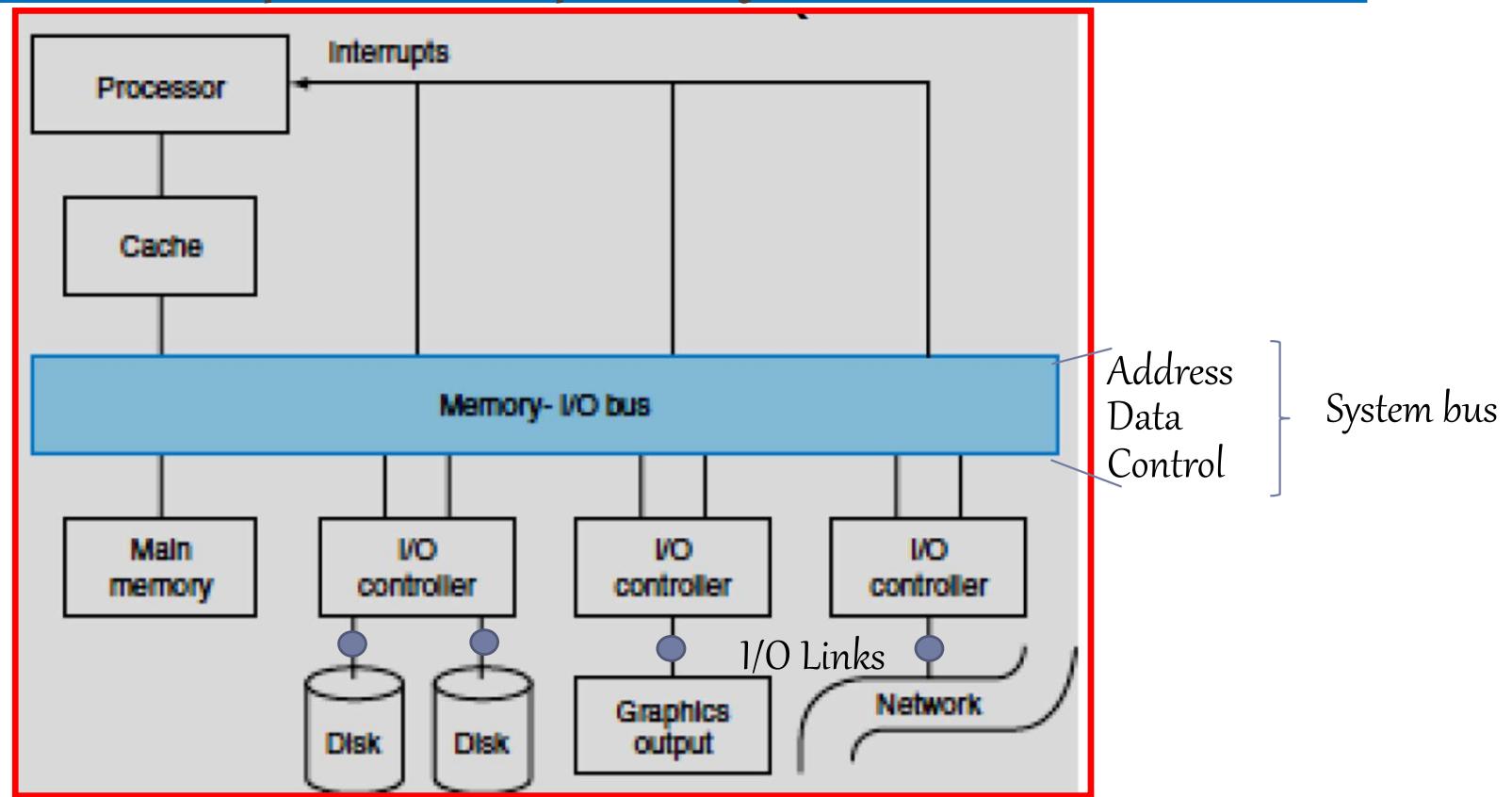
I/O Overview

Synchronous and Asynchronous Bus

- Synchronous – Signal and a Common Clock line
 - Suits on-board I/O, limited number of devices with similar speed
 - Example: system bus
- Asynchronous – No clock or self clocking
 - Variety of devices with range of speeds
 - Handshake adds to overhead
 - Example: Ethernet bus

Input Output System

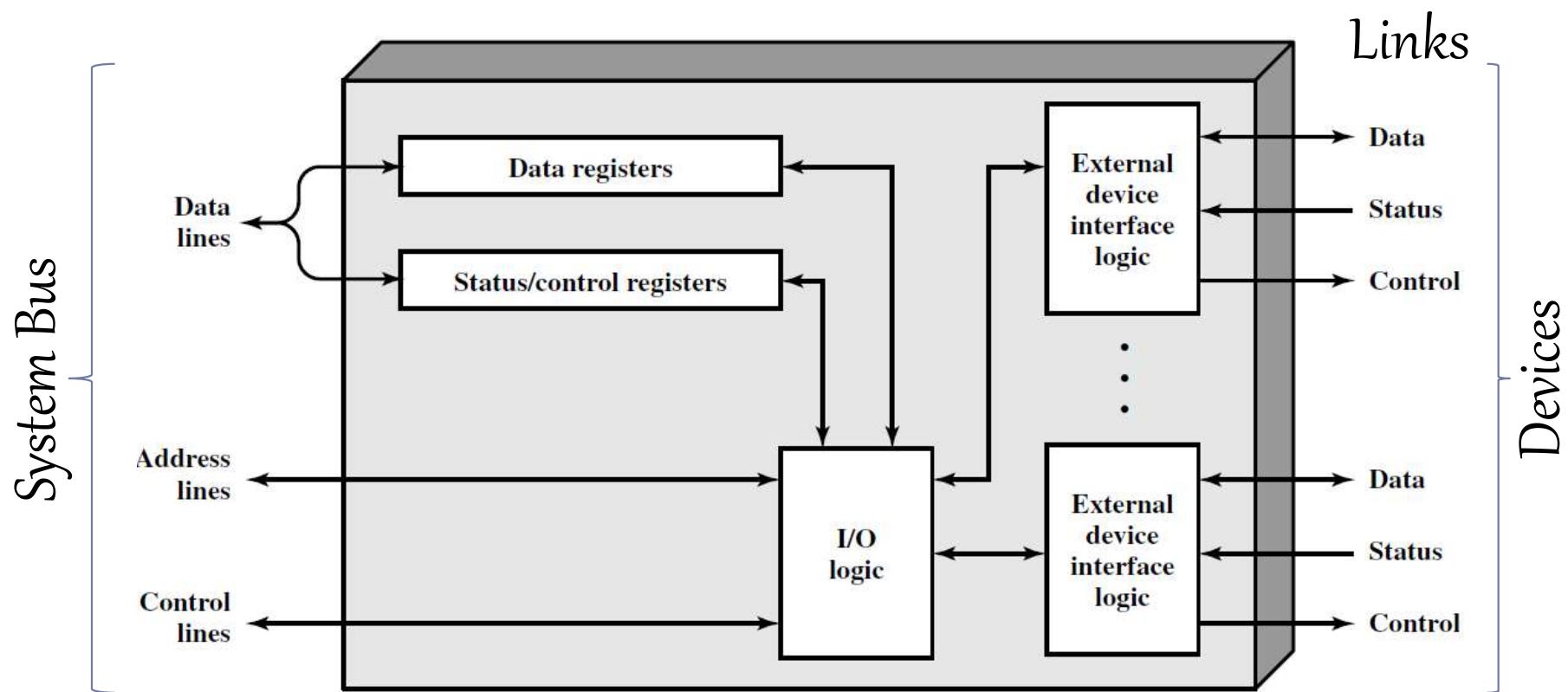
In general System bus is a
Synchronous
bus



Why I/O Module/Controller/Hub?

- I/O module bridges I/O devices to Processor and Memory
 - by
 - 1. Enabling a wide variety of peripherals to work with
 - 2. Supporting devices with various data transfer rates
 - 3. Supporting devices with different data formats

I/O Controller



Transfer of data from an external device

1. The processor interrogates the I/O module to check the status of the attached device.
2. The I/O module returns the device status.
3. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
4. The I/O module obtains a unit of data (e.g., 8 or 16 bits) from the external device.
5. The data are transferred from the I/O module to the processor.

I/O Command

- Processor issues *I/O Command* to I/O module
- *I/O commands* are different from *I/O instructions* but related
- I/O Command includes
 - I/O Module address, I/O Device address, and Command

I/O Command Types

- A command could be one of the 4 types:
 - Control (Seek, Rewind, etc.)
 - Test (Check Status, etc.)
 - Read
 - Write

I/O Address Options

- Registers in an I/O module is addressed either
 - As part of memory's address space – Memory mapped I/O
 - Range of memory address points to registers in I/O module
 - Independent address space – Isolated I/O
 - A control line to select registers in I/O module

Memory Mapped I/O Example

- “DATAIN” is the address of the input buffer associated with the keyboard.

Move DATAIN, R0

Note: Instruction corresponds to I/O Command here!

- Reads the data from DATAIN and stores that into processor register R0;

I/O Types

- Device \leftrightarrow Processor \leftrightarrow Memory
 - Programmed I/O
 - Interrupt driven I/O
- Device \leftrightarrow Memory (Direct Memory Access – DMA)

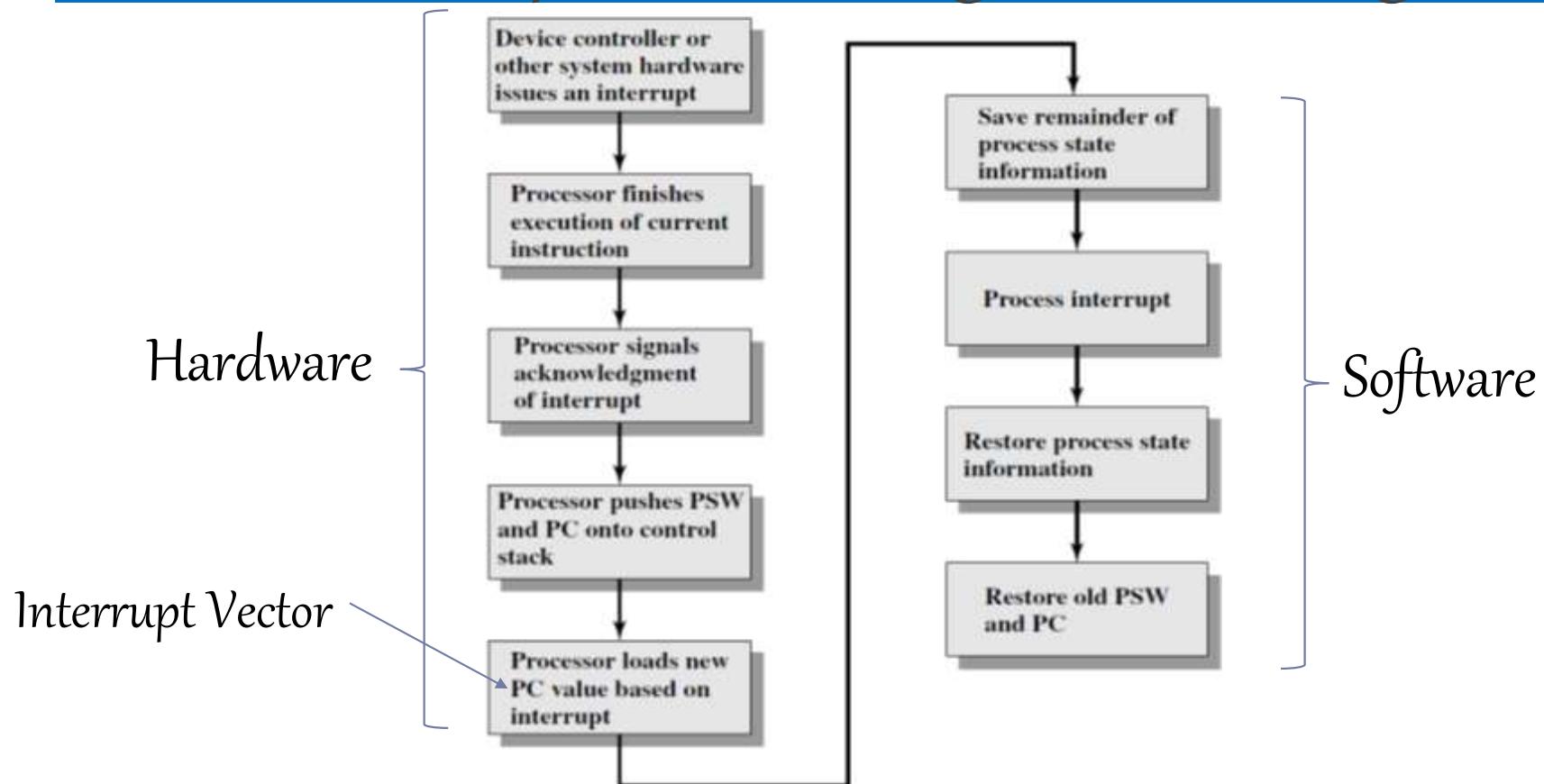
Programmed I/O

- In programmed I/O, processor slows itself down to the speed of the I/O device
- Processor initiates the I/O and actively stays with it by *polling* until I/O is completed
- Programmed I/O is easy to implement but it leads to poor performance (Under utilized CPU and Low Program Throughput)

Interrupt Driven I/O

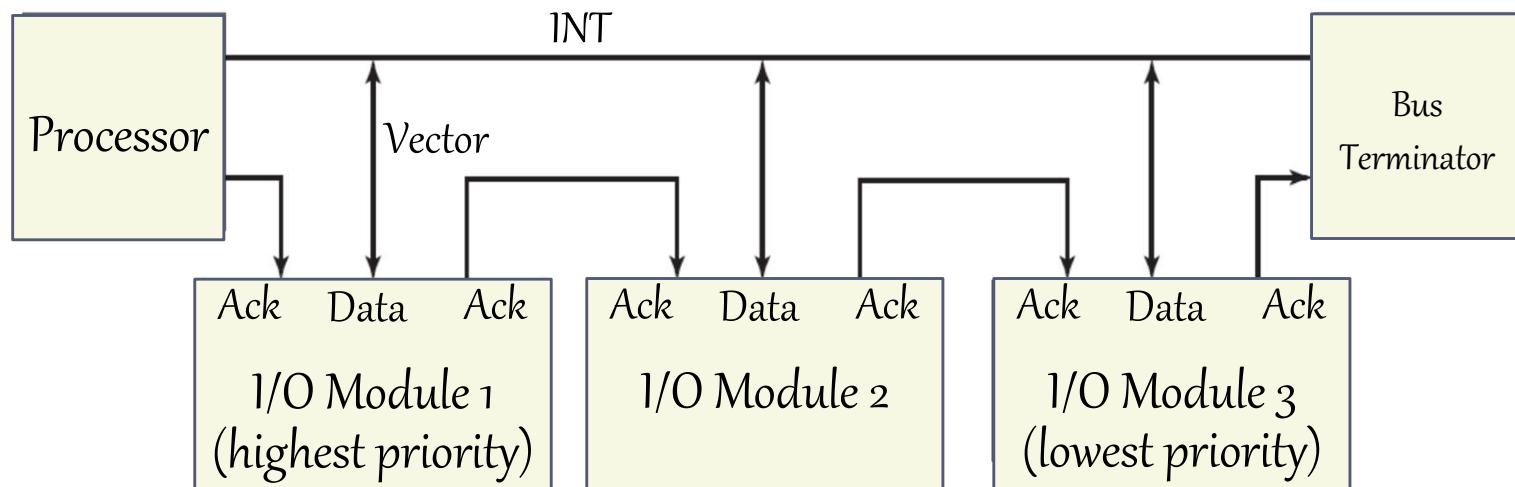
- Interrupt driven I/O is more complex and more efficient than Programmed I/O
- When an I/O is ready to be processed by the processor, processor is interrupted
- Processor can handle the I/O and then go back to what it was doing – thus processor time used more efficiently than polling

Interrupt Processing & handling



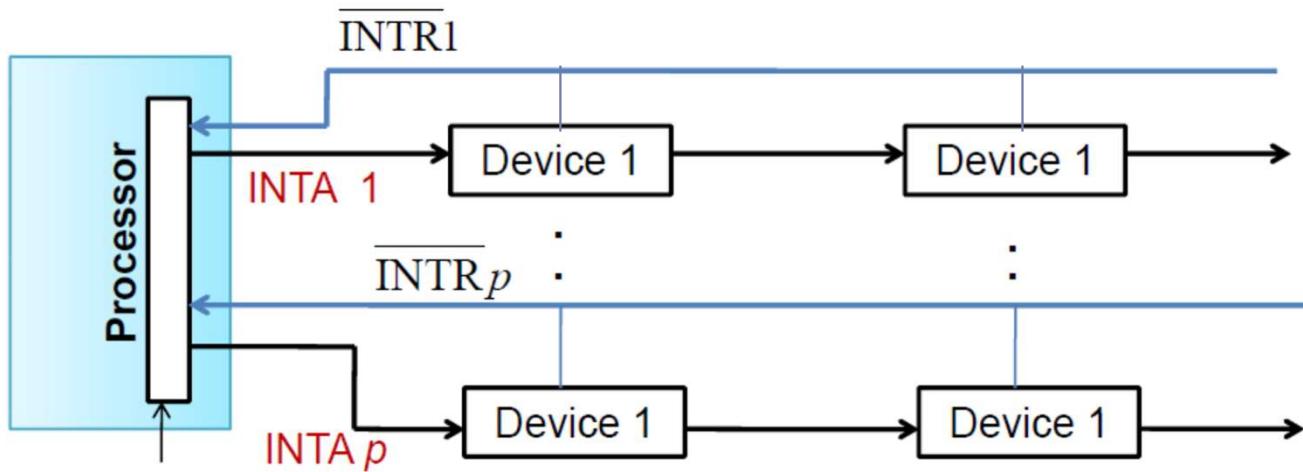
Daisy Chain

- Priority and Interrupt Vectoring are easier – Starvation is a problem
- Faster than polling (still slow due to daisy chain)
- Single interrupt line makes the design to be simple



Solution for Starvation

- Combining *Daisy chaining* and Interrupt nesting to form *priority group*
- Each *group* has different priority levels and within each group devices are connected in *daisy chain* way



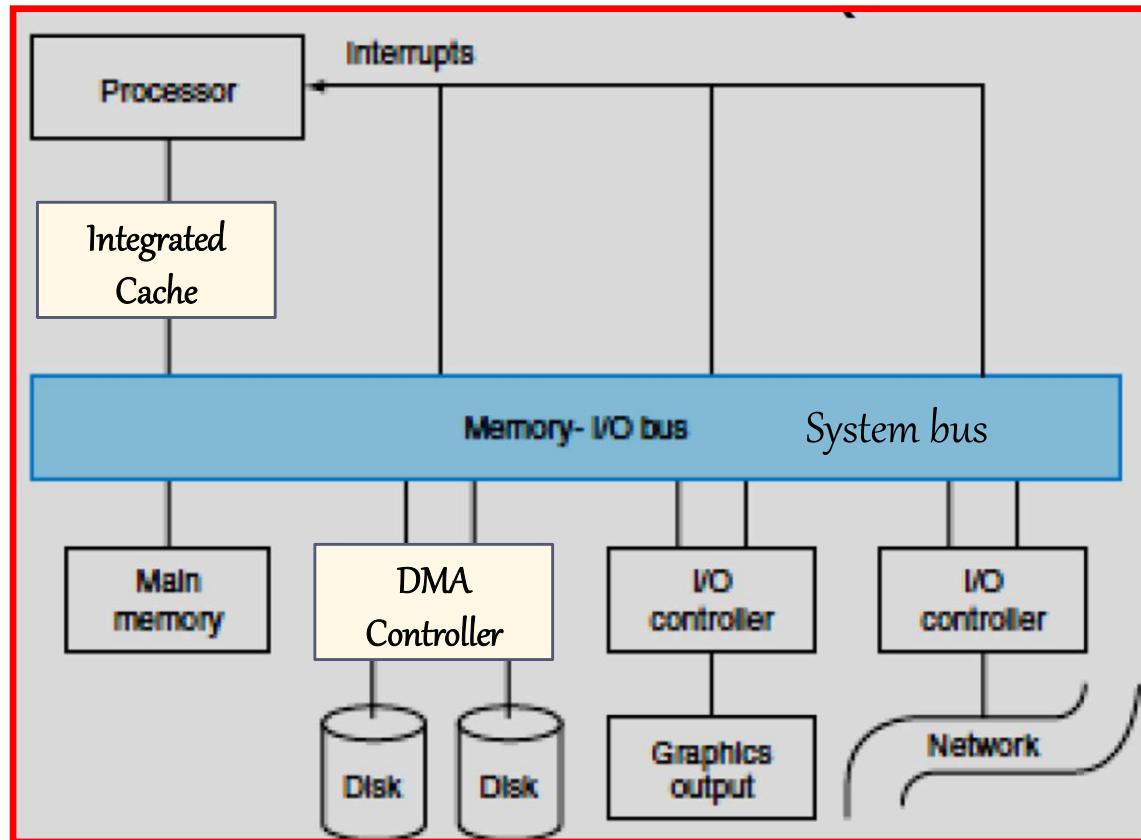
Direct Memory Access - Motivation

- Both Interrupt-driven I/O and Programmed I/O require the active intervention of the processor to transfer data between memory and an I/O Module
- Thus, both these forms of I/O suffer from two inherent drawbacks:
 1. I/O transfer rate is limited by the speed with which the processor can test and service a device.
 2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer
- The transfer rate overhead make both the options inadequate for large volume I/O

DMA Controller

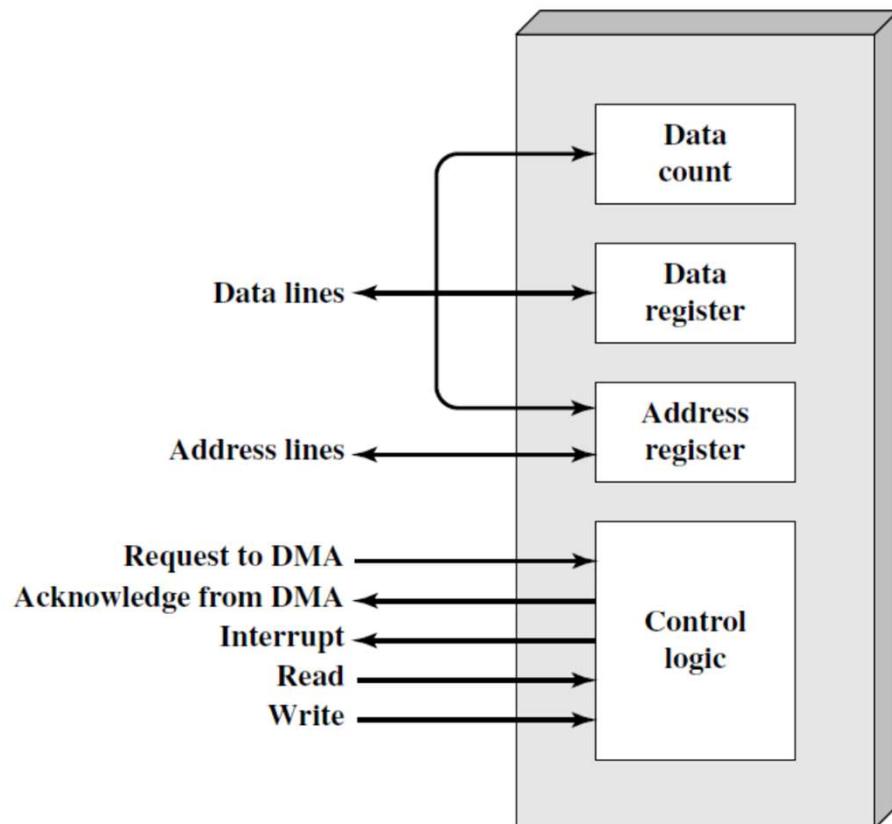
- DMA uses special I/O Controller – DMA Controller
- DMA controller mimic processor for data transfer between memory and I/O device
- DMA controller can acquire bus, use it for transfer, and release when done

DMA with Integrated Cache



DMA Transfer

- Processor can delegate certain transfer tasks to DMA controller
- This delegation request includes the range, addresses at source and destination for transfer
- DMA controller uses Interrupt to indicate the completion of transfer
- Processor is free to do other tasks when DMA transfer is in progress



Some Standard I/O Interfaces

REFERENCES:

1. <HTTP://WWW.PEARSONITCERTIFICATION.COM/ARTICLES/ARTICLE.ASPX?P=1681059>
 2. <HTTPS://WWW.GEOFFKNAGGE.COM/UNI/ELEC101/ESSAY.SHTML#CH7>
 3. <HTTPS://WWW.YOUTUBE.COM/WATCH?V=F7NLCAAL3YU>
 4. HTTPS://WWW.KEIL.COM/PACK/DOC/MW/USB/HTML/_U_S_B_ENDPOINTS.HTML
 5. <HTTP://WWW.USB.ORG>
-

Universal Serial Bus (USB)

- Universal serial bus is used to connect a variety of devices to processor
- USB includes power, thus enabling easier connection to passive devices like flash memory (pen drive)
- KBD, Mouse, and other peripherals are connected to computer using USB port

USB Versions

- There are three major versions of USB standards
 - USB 1.0 & 1.1 (12 Mbps)
 - USB 2.0 (Hi-Speed USB – 480 Mbps)
 - USB 3.0 (SuperSpeed USB 5 Gbs)
 - USB 3.1 (SuperSpeed+ 10 Gbs)
- Backward compatibility is maintained
- USB offers multiple connector types



USB-A



Micro-B



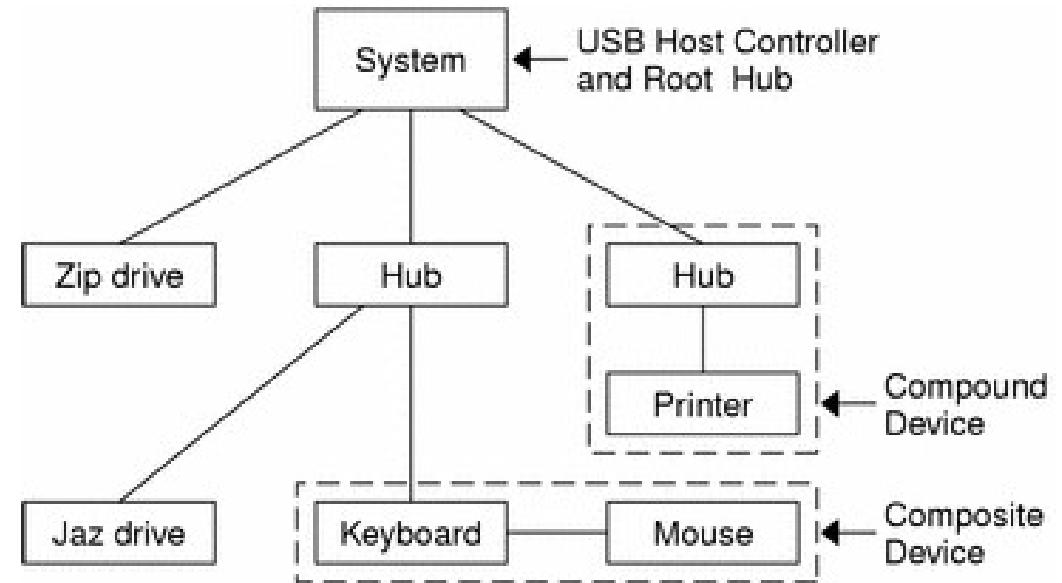
Mini-B



USB-B

USB – Architecture

- USB operates in Master-Slave mode
- Example: Computer is Master, KBD is slave
- Communication between Master and Slave is Asynchronous Serial Protocol



USB Features

- A maximum of 127 peripherals to a single USB host controller.
- Length of individual USB cable can reach up to 5 meters without a hub and 40 meters with hub.
- USB acts as *plug and play* device.
- *Hot swap*
- Devices can be powered through USB

Operation (Summary)

- Host sends a request when a device connects to it.
- Device responds with registered Vendor and Product ID
- Host loads the driver of the registered ID values.
- After this, the device is given a PID. This ID is used in all future exchanges
- Endpoint refer to data transfer port within a device.
- An Endpoint can Send or Receive data, cannot do both
- A device can support 16 end points at the same time. End point 0 is used for Receiving Control signals.

Data Transfer Modes

- There are 4 different modes of data transfers:
 1. **Interrupt:** used for devices which transfer little amount of data but need fast response (E.g. mouse, keyboard)
 2. **Bulk:** used for devices which receive big packet of data (E.g. printer)
 3. **Isochronous:** used for devices which requires streaming process (E.g. speaker, webcam)
 4. **Control:** short, simple commands to the device, and a status response.

Operation (1)

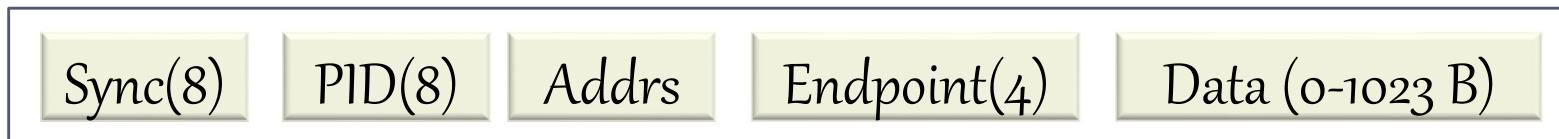
- When a computer is powered up or when a USB hub is connected to a computer, computing system will query and request from devices connected to the hub, the information on how much bandwidth is needed.
- Enumeration process will then occur where each device is assigned with a unique address.
- After that, the system will determine what kind of mode the USB devices wish to transfer data.

Operation (2)

- After all connected devices are enumerated, the computer system will take care of the overall bandwidth and allocate it to different devices according to their transfer mode.
- Most of the bandwidth will be used for interrupt and isochronous transfer to ensure their requests are guaranteed.
- Once 90% of bandwidth is taken, the computer will refuse any other transfer from these two modes.
- Bulk or control transfer (if available) will then take up the remaining bandwidth amount, which is up to 10%.

Message Format

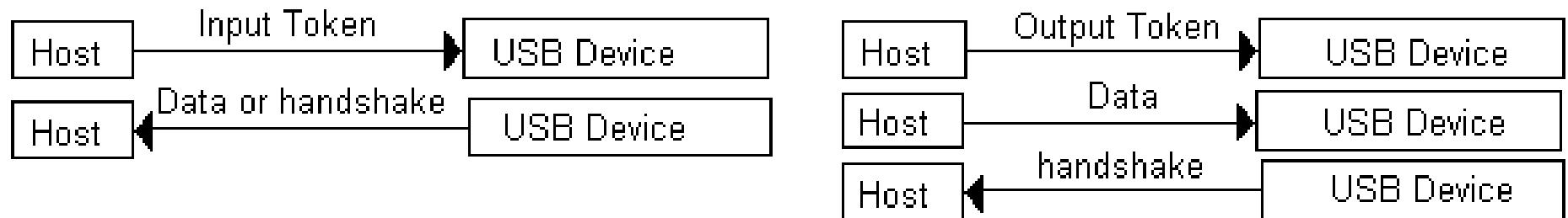
- ❑ Host & device communicate using packet exchange
- ❑ PID – Packet Type Identifier – identifies the mode
- ❑ Address + Endpoint – Device address
- ❑ Error check with variable size CRC



Types of Packet

- Token Packets : These are used to query the device and are issued by the host.
- Data Packets : To Exchange data in either direction
- Handshake Packets : To Send ACK, NACK, and STALL
- Start-of-Frame Packets : The USB host controls the processing of data in 1ms units called frames.

Data Transaction – Bulk Transfer

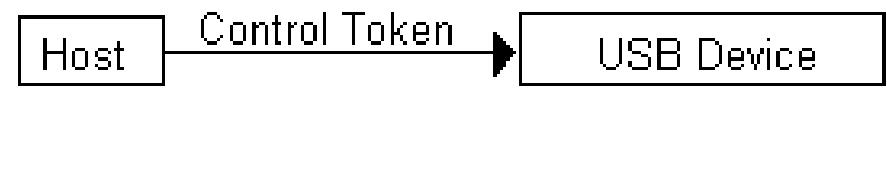


Input and Output Bulk Transaction

Data Transaction – Control Transactions



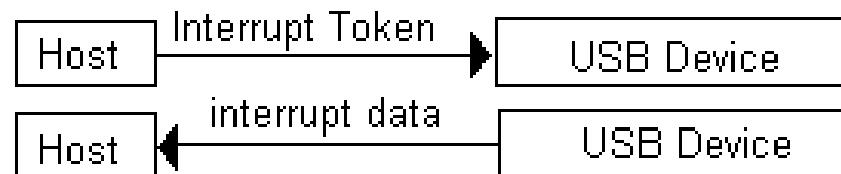
(a) Successful Control Command



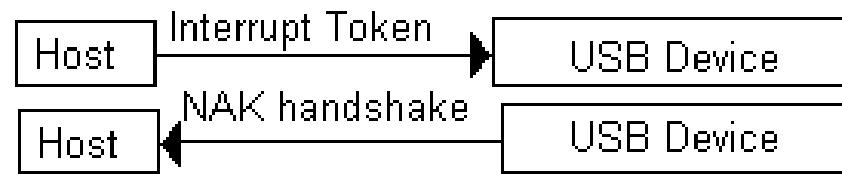
(b) Unsuccessful Control Command

: Communcation sequence for Control transactions

Data Transaction – Interrupt Transactions



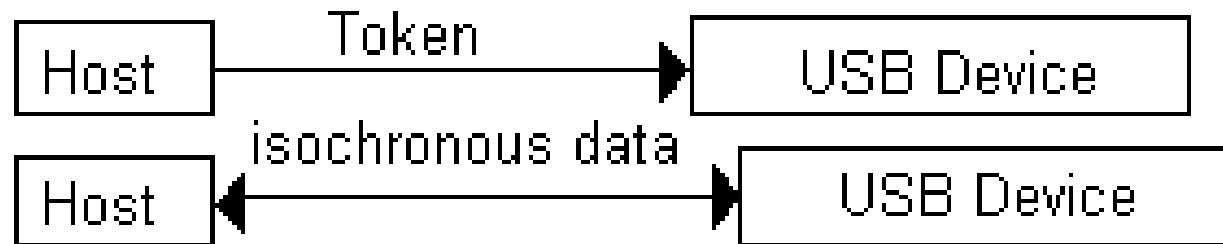
(a) Interrupt Pending



(b) No interrupt data waiting

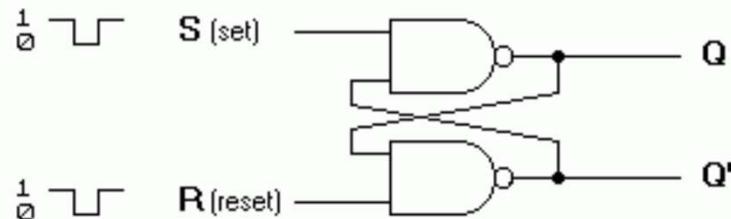
Communication sequence for Interrupt transactions

Data Transaction— Isochronous Transactions



Communication sequence for Isochronous transactions

Race Condition



(a) Logic diagram

S R	Q Q'
1 0	0 1
1 1	0 1
0 1	1 0
1 1	1 0
0 0	1 1

(after S=1, R=0)

(b) Truth table