

**INTRO. TO COMP. ENG.  
CHAPTER I-1  
INTRODUCTION**

**•CHAPTER I**

# **WELCOME TO ECE 2030: Introduction to Computer Engineering\***

**Richard M. Dansereau  
rdanse@pobox.com**

**Copyright © by R.M. Dansereau, 2000-2001**

**\* ELEMENTS OF NOTES AFTER W. KINSNER, UNIVERSITY OF MANITOBA**

# LEARN BASICS BEHIND COMPUTER SYSTEMS

- Hardware architecture and organization
- Digital logic design
  - Switching and logic gate design
- Computer architecture building blocks
  - Adders/subtractors/counters
  - Shift/rotate registers
  - Multiplexers/demultiplexers and encoders/decoders
  - Controllers and sequencers
- Software mapping to hardware
  - Instruction types and assembly languages
  - OS issues, branching, jumping, interrupts, subroutines

## **APPRECIATION OF:**

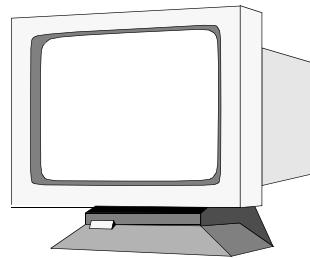
- Computer system architectures and organization

## **KNOWLEDGE OF:**

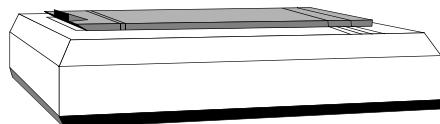
- Taxonomy of computing structures
- Switching and Boolean algebra
- Combinational logic and sequential logic
- Building blocks to computer architectures
- State machines, finite states machines
- Instruction types and addressing modes
- Single and multi-cycle data path units, microcode
- Software execution in hardware for higher level operating systems

**TYPICAL OUTPUTS**

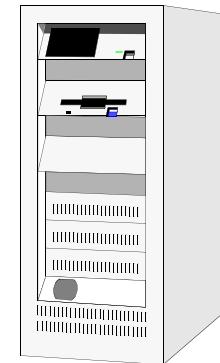
**MONITOR**



**PRINTER**

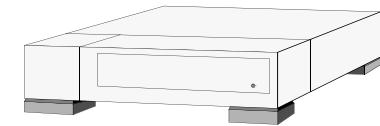


**MAIN COMPUTER**

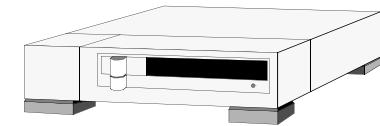


**TYPICAL I/O**

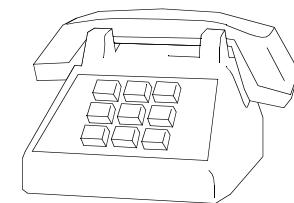
**HARD DRIVE**



**TAPE**



**MODEM**

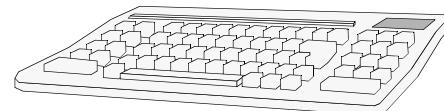


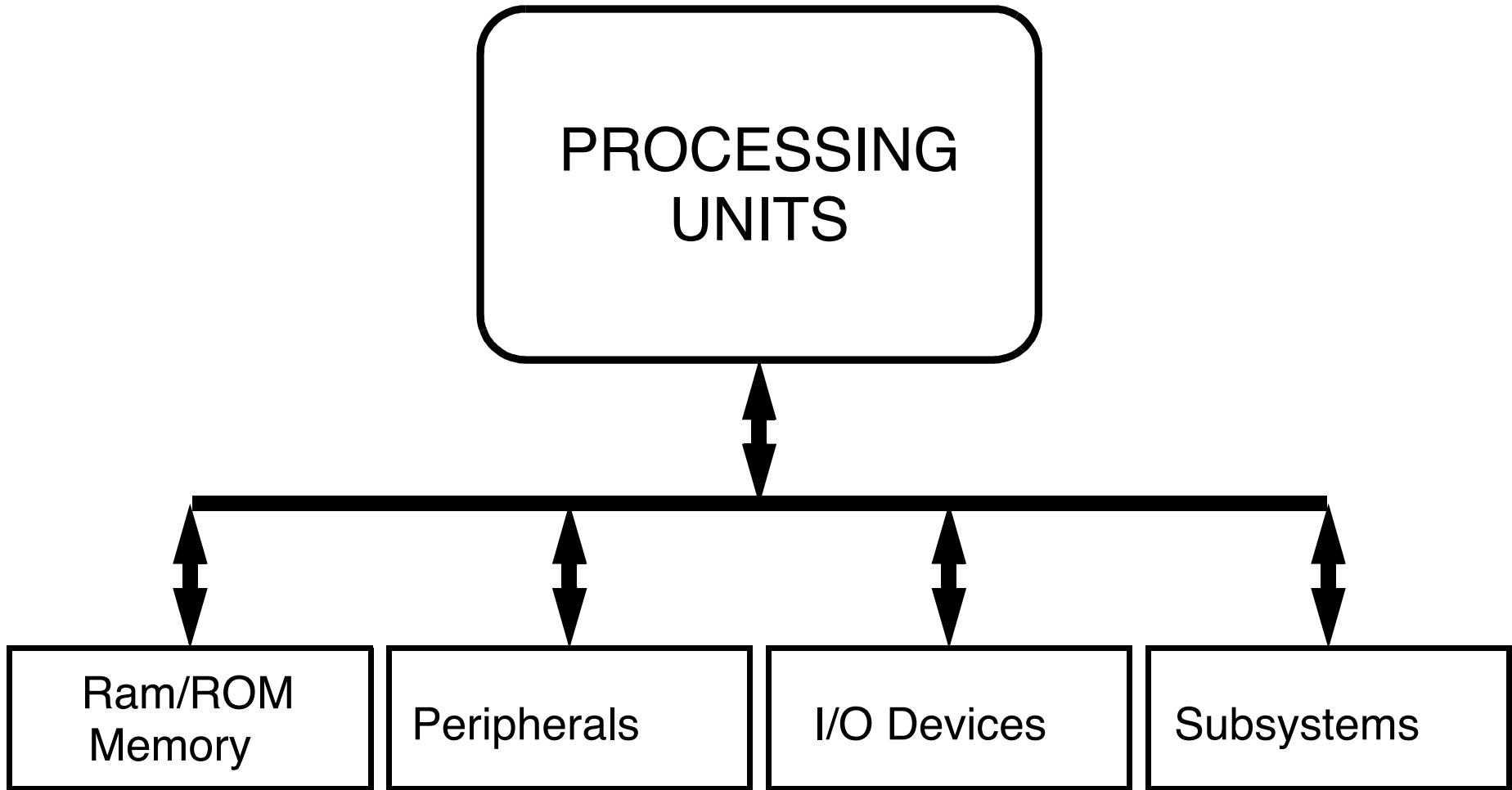
**TYPICAL INPUTS**

**MOUSE**



**KEYBOARD**



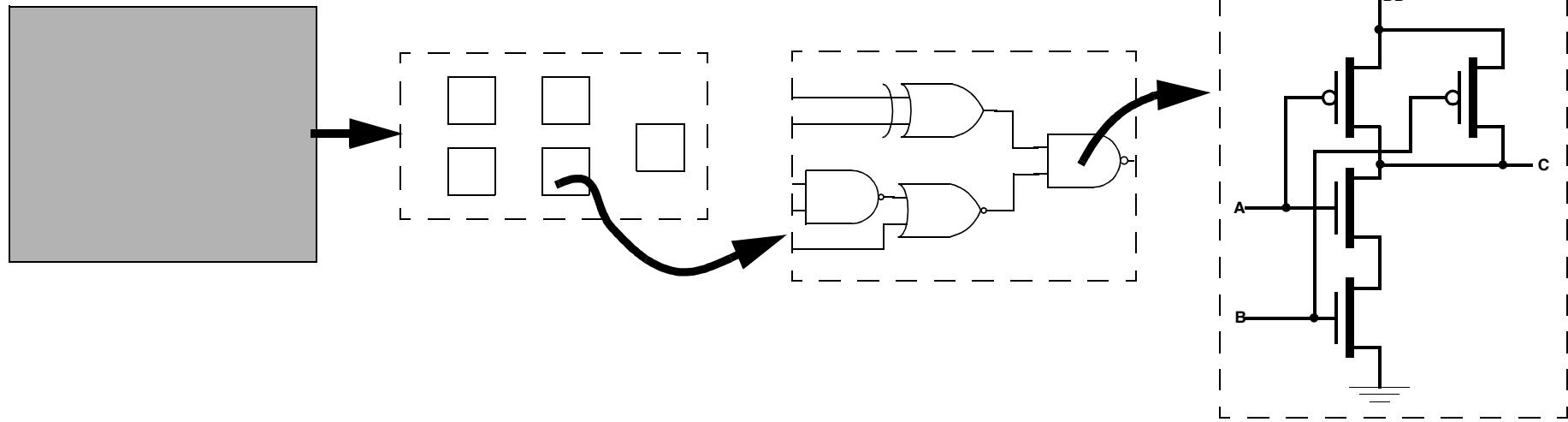


### SYSTEM

### MODULES

### GATES AND FLIP-FLOPS

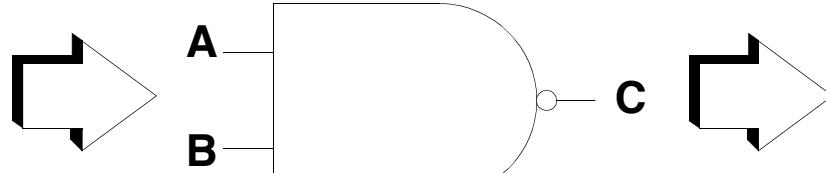
### TRANSISTORS



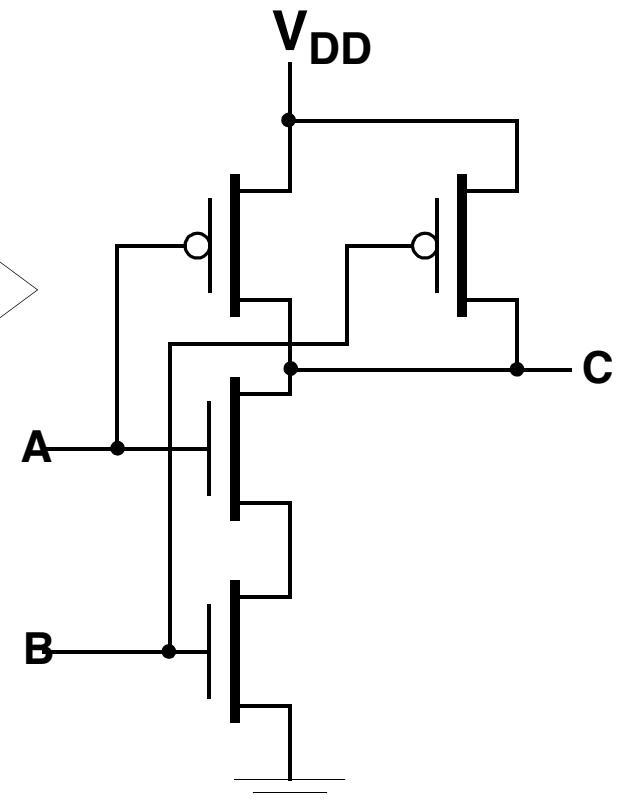
## TRUTH TABLE

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

## LOGIC GATE REPRESENTATION



## TRANSISTOR (SWITCH) IMPLEMENTATION



**SYSTEM LEVEL:**

- Processors, memories, peripherals
- Words, files, records, programs
- HDL, natural language

**REGISTER-TRANSFER LEVEL:**

- Registers, ALUs, buses, CCUs
- Bytes, words, double words
- Block diagrams, state diagrams

**LOGIC LEVEL:**

- Gates, flip-flops
- 1, 0, X (unknown); Strong, weak, Z
- Logic diagrams, boolean equations

**CIRCUIT LEVEL:**

- R, C, L, Diodes, Transistors
- Voltage, current, temperature
- Schematic diagrams, circuit equations

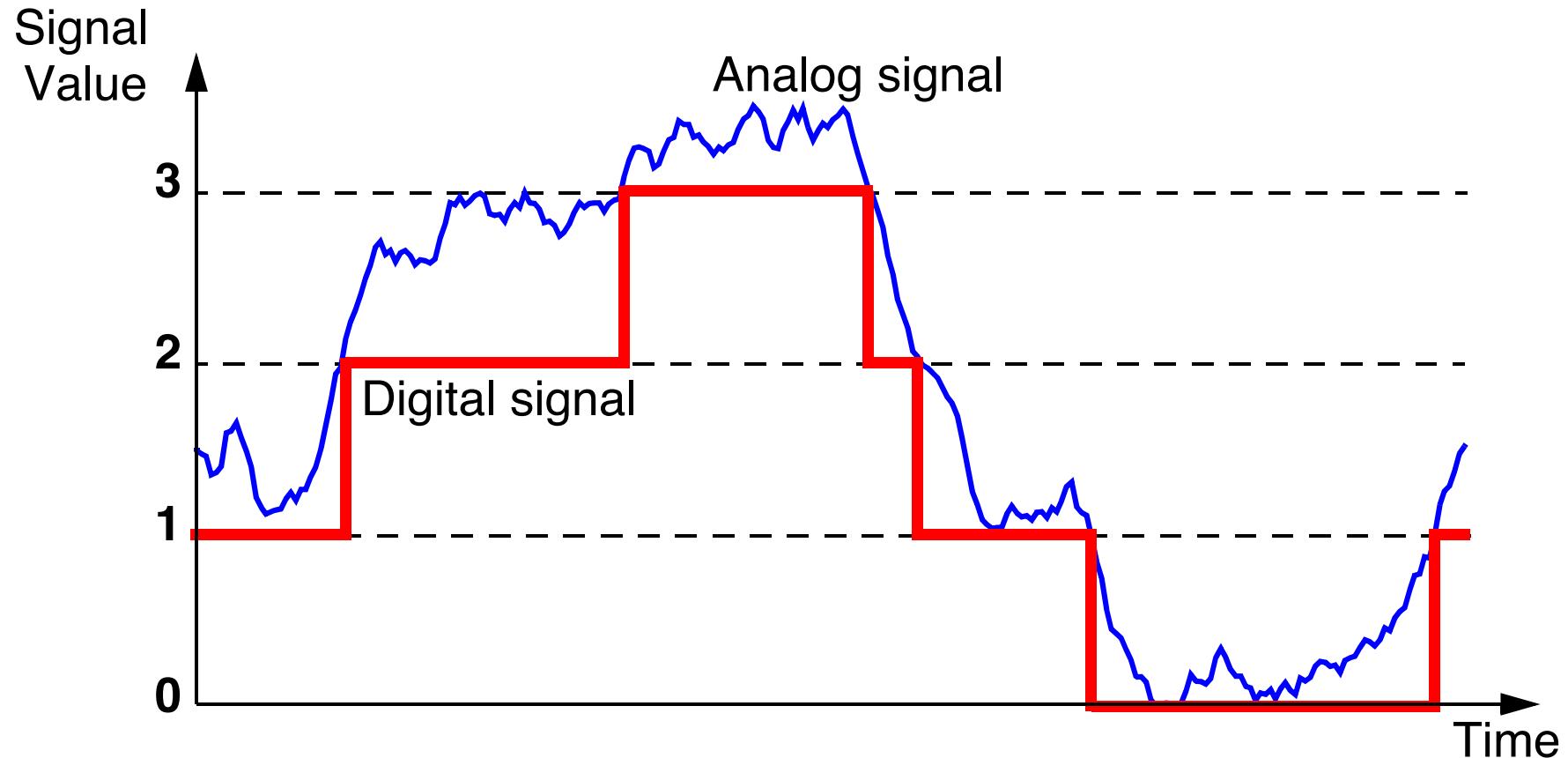
**SILICON LEVEL:**

Elements:	-nPN & PNP transistors, CMOS
Values:	-Voltage, current, temp., fields
Description:	-Device models, interconnects

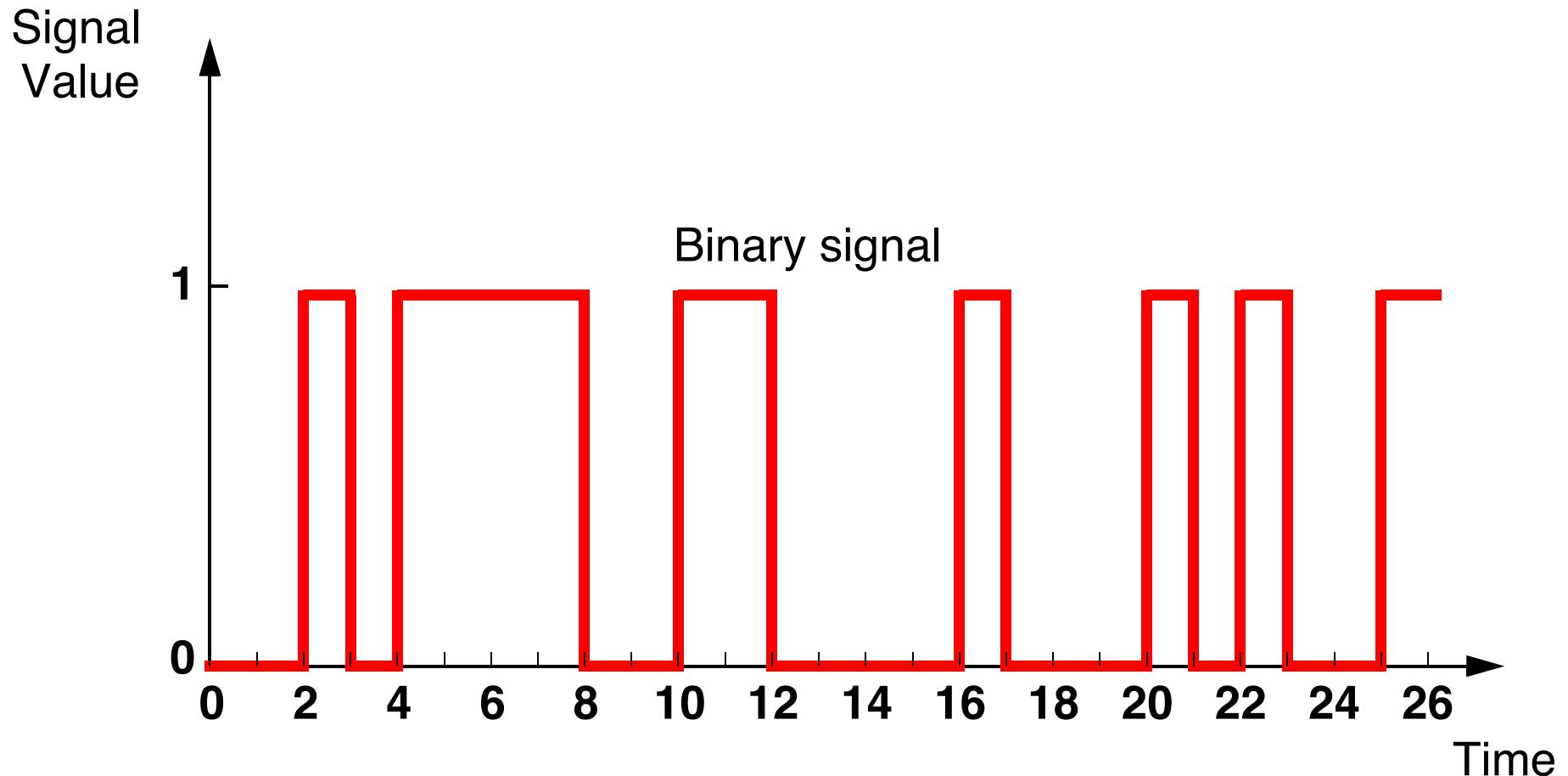
- **Small scale integration (SSI)**
  - ~10 transistors
  - Individual gates, flip-flops
- **Medium scale integration (MSI)**
  - 10-100 transistors
  - Adders, encoders/decoders, multiplexers, shift registers, counters
- **Large scale integration (LSI)**
  - 100-10,000 transistors
  - small memories, ROMs, PLAs, small memories
- **Very-large scale integration (VLSI)**
  - > 10,000 transistors
  - microprocessors, DSP chips, large memories

# ANALOG VS. DIGITAL

## A-TO-D CONVERSION

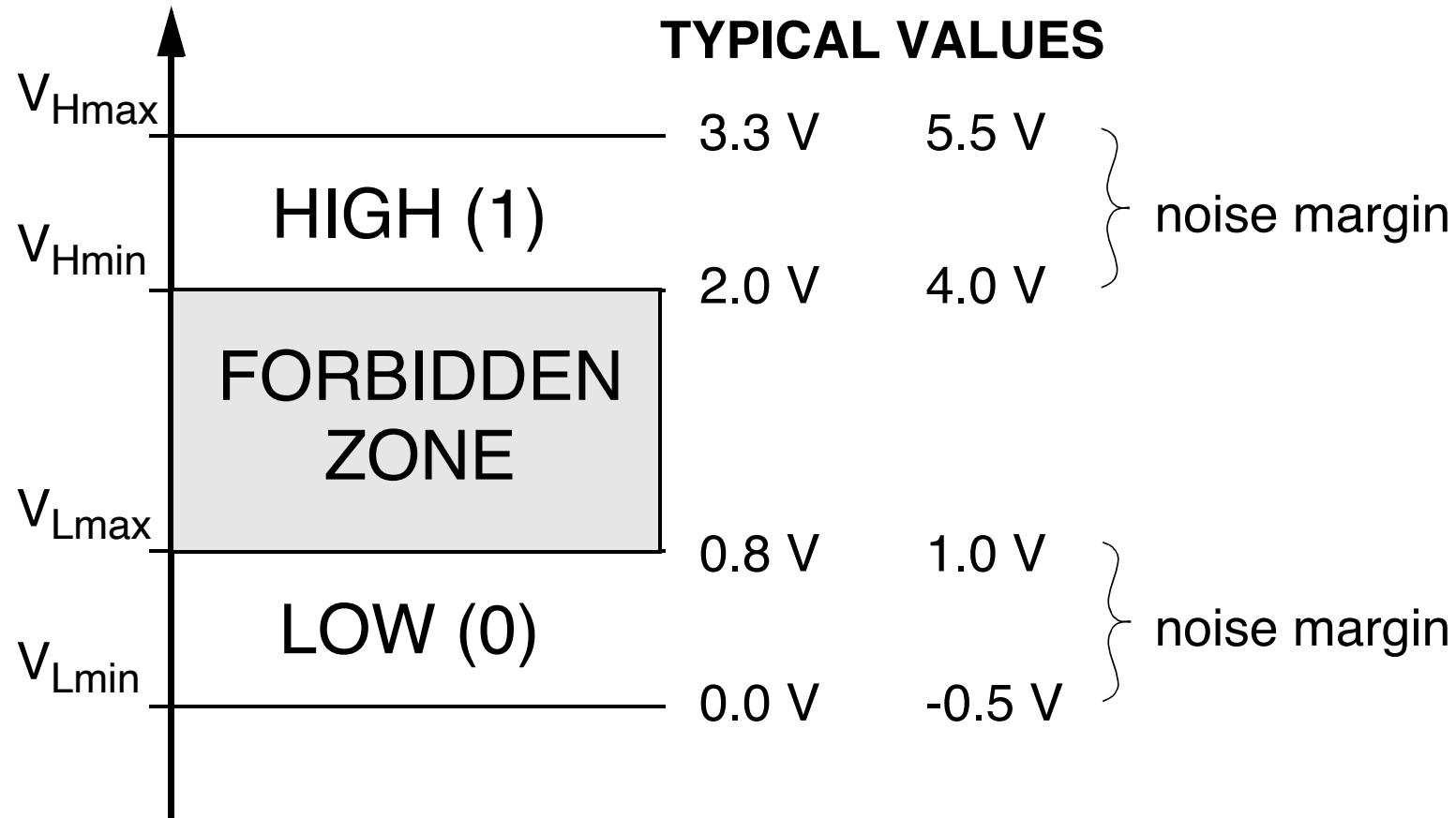


# ANALOG VS. DIGITAL BINARY SIGNAL



# BINARY

## BINARY LOGIC LEVELS



- Binary numbers are base 2 as opposed to base 10 typically used.
- Instead of decimal places such as 1s, 10s, 100s, 1000s, etc., binary uses powers of two to have 1s, 2s, 4s, 8s, 16s, 32s, 64s, etc. places.

Examples:

$$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 4_{10} + 1_{10} = 5_{10}$$

$$10111_2 = (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 23_{10}$$

$$00101111_2 = (1 \times 2^5) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 47_{10}$$

- We will discuss binary numbers and binary arithmetic in a little more depth later.

**INTRO. TO COMP. ENG.  
CHAPTER XV-1  
PROCEDURE CALLS**

**•CHAPTER XV**

# **CHAPTER XV**

## **PROCEDURE CALLS AND SUBROUTINES**

# PROCEDURE CALLS

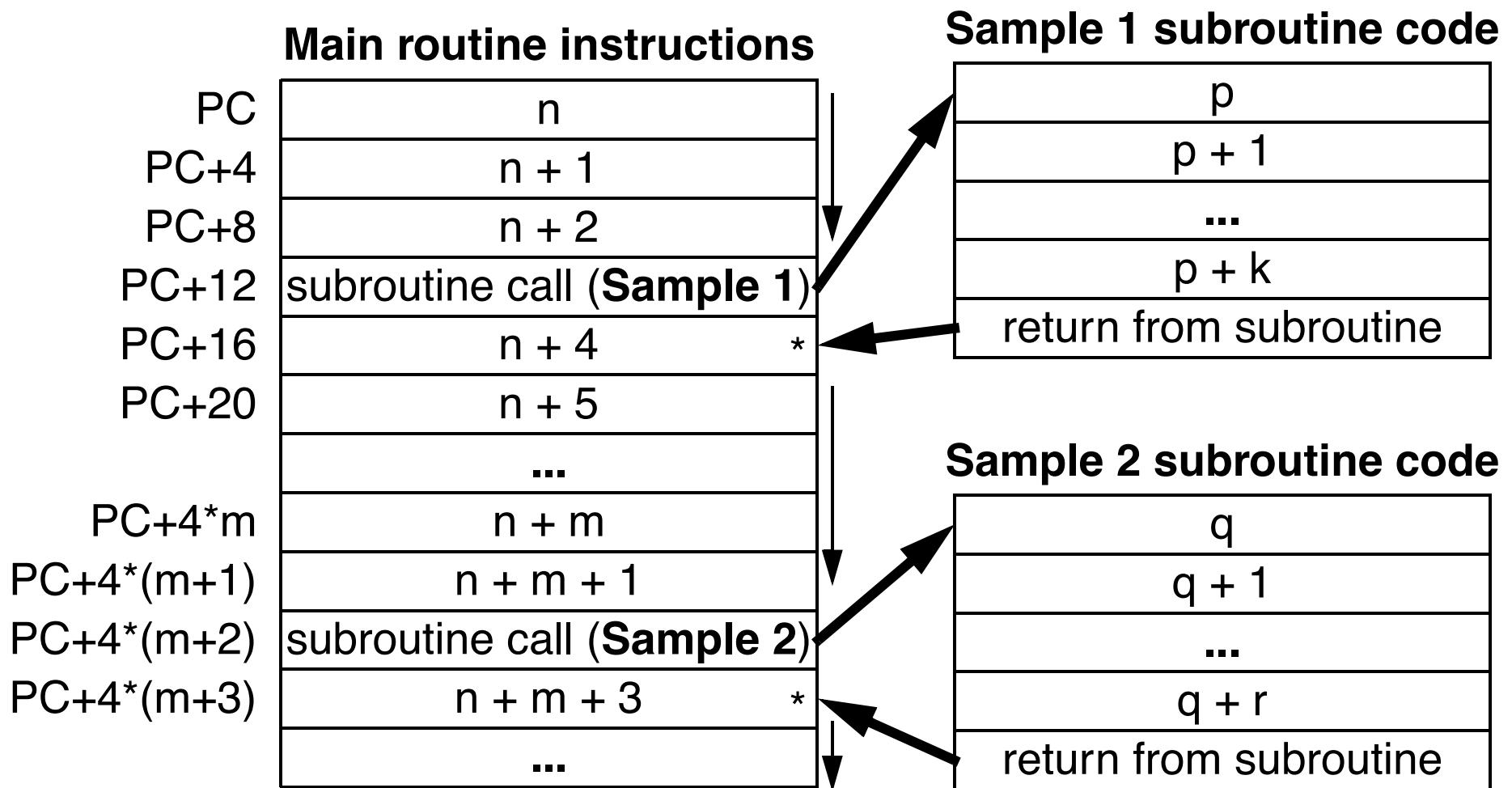
## INTRODUCTION

- Branches and jumps are important program control constructs, but another important extension of program control are **procedure calls**, often referred to as **subroutines**.
- Three basic steps form of a subroutine call
  - Program control is changed
    - **from the current routine**
    - **to the beginning of the subroutine code.**
  - Subroutine code is executed.
  - Program control is changed
    - **from end of subroutine**
    - **to the calling routine immediately\* after subroutine call instruction.**

# PROCEDURE CALLS

## PROGRAM FLOW

- We can illustrate how subroutine calls change program flow as follows.



- How can program flow be changed to a subroutine?
  - **PC = address of 1st instruction of subroutine**
  - And then returned from a subroutine call?
    - **PC = address of instruction after subroutine call instruction**
- The idea is to **save the state of the machine**.
- In the most basic microprocessor, saving the state means to **save the PC in a known location!**
- Some microprocessors also save other registers during a procedure call.
- **MIPS** only saves the **PC** and then restores the **PC** after the subroutine.

## MACHINE STATE

### SAVING STATE TO \$RA

- For MIPS, the primary location for saving the **PC** is in **\$31/\$ra**.
- MIPS uses the instruction **jal <imm>** (jump and link)
  - **jal** is **J-format** type instruction.
  - **Stores the return address in \$ra**, i.e. **\$ra = PC + 4\***.
  - **Performs jump** such as with the **j** instruction.
- At the end of the subroutine, the instruction **jr \$ra** is executed to return to calling routing.
  - This causes the contents of **\$ra** to be put into **PC**
    - i.e. **PC = \$ra** which after the original **jal** instruction is **PC = PC + 4\***.

# MACHINE STATE

## EXAMPLE PROCEDURE CALL

- Below is an example piece of pseudo-code that has been translated in assembly with a main routine and a square root subroutine.

Pseudo-Code

MIPS Assembly

b = 6;  
a = sqrt(b);  
a = a + b;

main routine  
(use \$s0 for a, \$s1 for b)

square root subroutine  
(argument in \$a0, result in \$v0)

```
lwi $s1, 0x06
move $a0, $s1
jal sqrt
move $s0, $v0
add $s0, $s0, $s1
...
jr $ra
```

```
graph TD; A[lwi $s1, 0x06] --> B[move $a0, $s1]; B --> C[jal sqrt]; C --> D[move $s0, $v0]; D --> E[add $s0, $s0, $s1]; E --> F[jr $ra];
```

- Another approach to saving the **PC** is the in the form **jalr \$<dest>, \$<src>** (jump and link register) instruction.
  - **jalr** is roughly an **R-format** type instruction.
  - **Stores the return address in \$<dest>**, i.e.  $\$5 = PC + 4^*$ .
  - **Performs jump** such as with the **jr <\$src>** instruction.
- At the end of the subroutine, to return from the subroutine the following can be executed.
  - **jr \$<dest>** (i.e. **jr \$5**)
- Another option for returning from a subroutine is to execute
  - **jalr \$0, \$5,**
  - or even **jalr \$<new dest>, \$5.**

# MACHINE STATE

## EXAMPLE PROCEDURE CALL

- Another example where jalr is used and the subroutine is completely given.

Pseudo-Code

MIPS Assembly

main routine  
(use \$s0 for a, \$s1 for b)      decrement subroutine  
(argument in \$a0, result in \$v0)

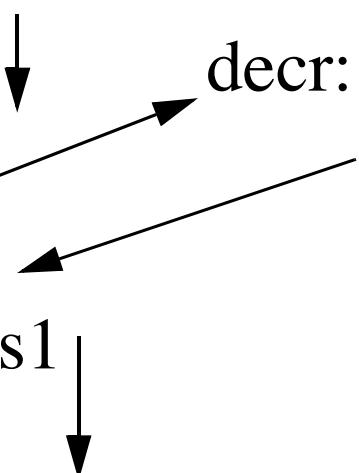
b = 6;

a = decr(b);

a = a + b;

lwi \$s1, 0x06  
move \$a0, \$s1  
jalr \$s7, decr  
move \$s0, \$v0  
add \$s0, \$s0, \$s1  
...

decr: subi \$v0,\$a0,1  
          jr \$s7



# MACHINE STATE

## EXAMPLE PROCEDURE CALL

- A more complicated example could be as follows.

### Pseudo-Code

```
a = 6;  
b = 4;  
c = 10;  
d = func(b,c,a);  
...  
  
int func(x,y,z)  
    return x+y-z;
```

### MIPS Assembly

Main routine  
(use \$s0-3 for a,b,c,d)

```
lwi $s0, 0x06  
lwi $s1, 0x04  
lwi $s2, 0x0A  
move $a0, $s1  
move $a1, $s2  
move $a2, $s0  
jal func  
move $s3, $v0  
...
```

func subroutine  
(arguments in \$a0-2,  
result in \$v0)

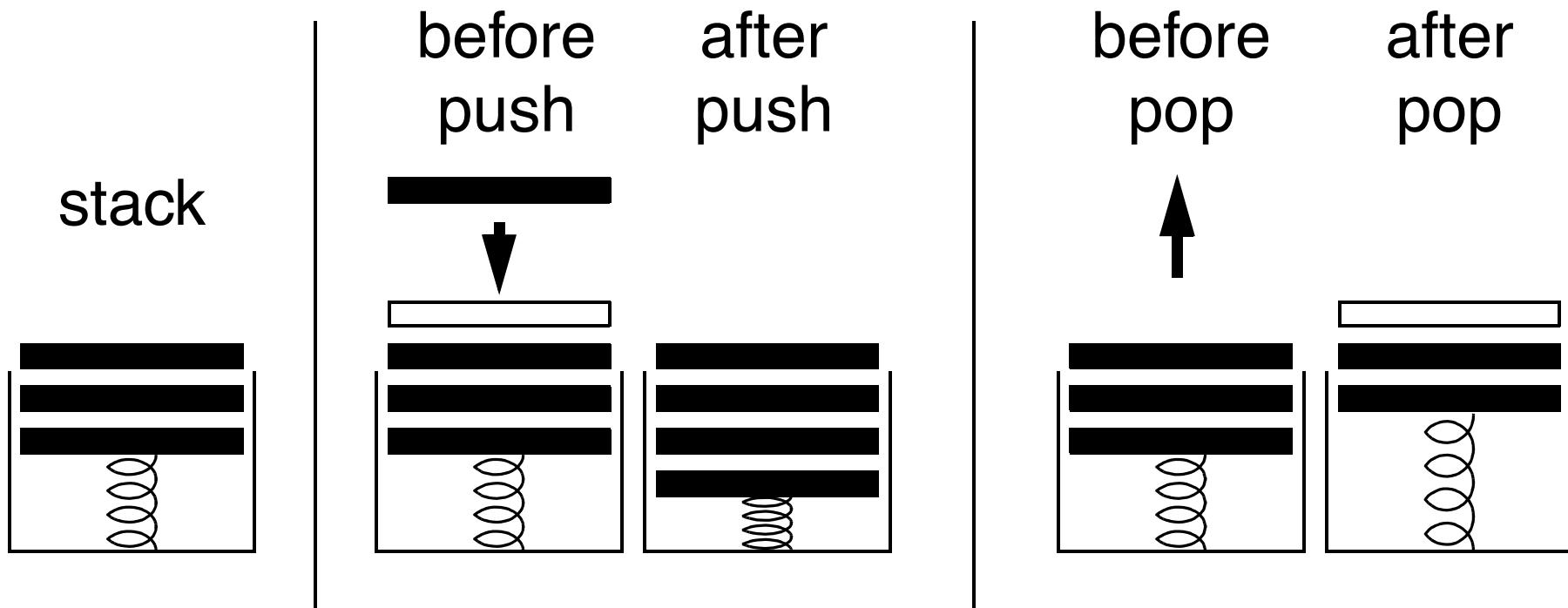
```
func: sub $v0,$a1,$a2  
      add $v0,$a0,$v0  
      jr $ra
```

- Two problems exist with the subroutine approach discussed so far.
- **Problem 1:**
  - What if we want to call a subroutine within a subroutine?
    - Only one **\$ra**, so only one return address is stored with **jal**.
    - If we call a nested subroutine, the return address in **\$ra** is lost.
- **Problem 2:**
  - What if we need many temporary registers within the subroutine?
    - We don't want to lose the contents of registers that the calling function might still need!
- **Solution: Stacks**

# STACKS

## PUSHING AND POPPING

- A stack is a **LIFO** (Last-In, First-Out) data structure.
  - Consider the example of a stack of plates at a cafeteria.



- A plate can be added to the top of the stack, called a **push**.
- A plate can be removed from the top of the stack, called a **pop**.

- Which way should a **stack grow** in memory?
  - It is customary for a stack to **grow from larger** memory addresses **to smaller** memory addresses.
- Use a stack pointer (**SP**) to point to top of stack. This is **\$29/\$sp** on MIPS.
- **push:** To place a new item onto the stack
  - first decrement **SP**,
  - then store item at the new location pointed to by **SP**.
- **pop:** To retrieve an item from the stack
  - first copy item pointed to by **SP** into desired destination,
  - then increment **SP**.
- **Many processors deviate slightly from this, but with the same idea.**

- Following the previous slide, we can think of our memory model as follows if **SP = 0x00FFFFF4** and the bottom of the stack is **0x01000000**.

**push: SP = SP - 4**

**pop: SP = SP + 4**

0x00FFFE0	
0x00FFFE4	
0x00FFFE8	
0x00FFFFC	
0x00FFFF0	
0x77777777	
0x01234567	
0x76543210	
0x45553323	
0x01000000	

**SP →**

- We can see that the stack grows from larger address to smaller address.

- The following instructions perform a **push** of **R15** onto the stack.

```
sub $sp, 0x04  
sw $15, $sp
```

- The following instructions perform a **pop** from the stack into **R15**.

```
lw $15, $sp  
add $sp, 0x04
```

- Many processors actually have the instructions **push** and **pop**, but MIPS removes these to have fewer opcodes (i.e. RISC).

- A **push** on MIPS is performed and illustrated as follows.

Given that

R15=0x77777777

Push of R15 onto stack

```
sub $sp, 0x04  
sw $15, $sp
```

**Before push: R15=0x77777777**

SP →	0x00FFFFFF0	
	0x00FFFFFF4	
	0x00FFFFFF8	0x01234567
	0x00FFFFFFC	0x76543210
	0x01000000	0x45553323

**After push: R15=0x77777777**

SP →	0x00FFFFFF0	
	0x00FFFFFF4	0x77777777
	0x00FFFFFF8	0x01234567
	0x00FFFFFFC	0x76543210
	0x01000000	0x45553323

- A **pop** on MIPS is performed and illustrated as follows.

Pop from stack to R15

```
lw $15, $sp  
add $sp, 0x04
```

Now R15=0x01234567

**SP →**

0x00FFFFFF0  
0x00FFFFFF4  
0x00FFFFFF8  
0x00FFFFFFC  
0x01000000

0x01234567
0x76543210
0x45553323

**Before pop: R15=0x?????????**

**After pop: R15=0x01234567**

**SP →**

0x00FFFFFF0  
0x00FFFFFF4  
0x00FFFFFF8  
0x00FFFFFFC  
0x01000000

0x01234567
0x76543210
0x45553323

- Procedure calls can now be nested since **\$ra** can be saved on the stack.

Main routine

...

...

...

jal func1

...

...

...

func1

sub \$sp, 0x04  
sw \$ra, \$sp

} Save return  
address (push)

...

jal func2

...

lw \$ra, \$sp  
add \$sp, 0x04  
jr \$ra

func2

...

...

jr \$ra

- This example can be thought of in a higher level language as

```
complex Z addcomplex(complex X, complex Y) {  
    Z.real = X.real + Y.real;  
    Z.imaginary = X.imaginary + Y.imaginary;  
    return Z;  
}  
  
complex W funcAadd2B(complex U, complex V) {  
    W = addcomplex(U, V);  
    W = addcomplex(W, V);  
    return W;  
}  
  
main {  
    complex A = 5 + i6, B = 2 + i7, C;  
    C = funcAadd2B(A, B);  
}
```

- Say that we want to write a function **funcAadd2B** that calculates **A+2B** where **A** and **B** are complex numbers.
  - (**\$a0,\$a1**) contains (**real,imaginary**) part of **A**.
  - (**\$a2,\$a3**) contains (**real,imaginary**) part of **B**.
  - (**\$v0,\$v1**) contains (**real,imaginary**) part of answer.
- To make life easier, also design function **addcomplex** that adds two complex numbers **X** and **Y**.
  - (**\$a0,\$a1**) contains (**real,imaginary**) part of **X**.
  - (**\$a2,\$a3**) contains (**real,imaginary**) part of **Y**.
  - (**\$v0,\$v1**) contains (**real,imaginary**) part of answer.

- This example could be implemented as follows in assembly.

Main routine

...  
...  
...  
jal funcAadd2B  
...

funcAadd2B

```
sub $sp, 0x04
sw $ra, $sp
jal addcomplex
move $a0,$v0
move $a1,$v1
jal addcomplex
lw $ra, $sp
add $sp, 0x04
jr $ra
```

addcomplex

```
add $v0,$a0,$a2
add $v1,$a1,$a3
jr $ra
```

**SWITCH DESIGN**  
**CHAPTER II-1**  
**SWITCH DESIGN**

**•CHAPTER II**

# **CHAPTER II**

## **SWITCH NETWORKS AND SWITCH DESIGN**

# Analog vs Digital



**FILM vs DIGITAL**



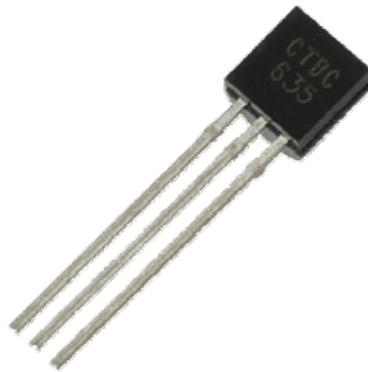
**VS**



# Transistor: Electrical Switch



**Bardeee, Shockley, Brattain (Bell Labs, 1948), Nobel Prize Winners**



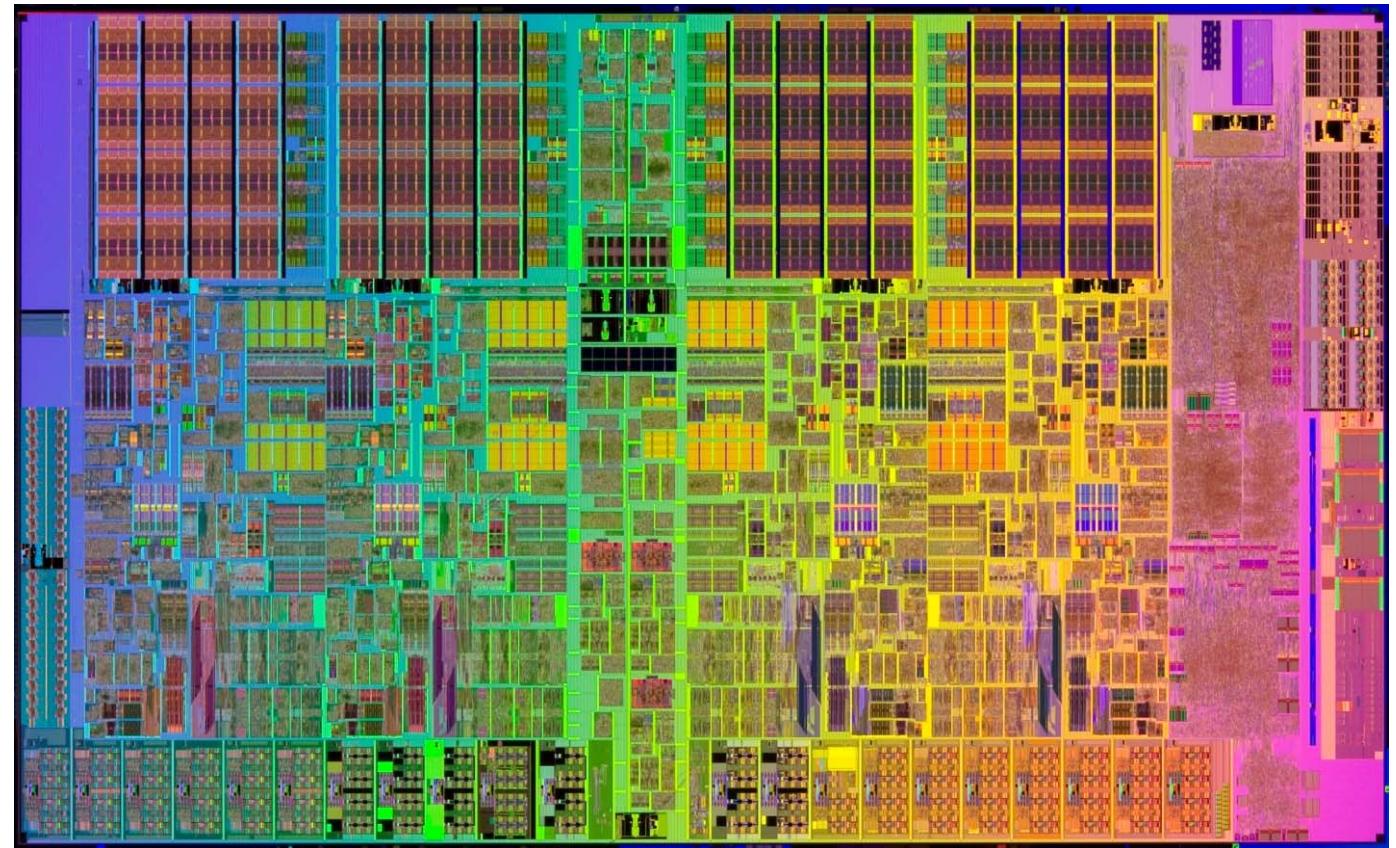
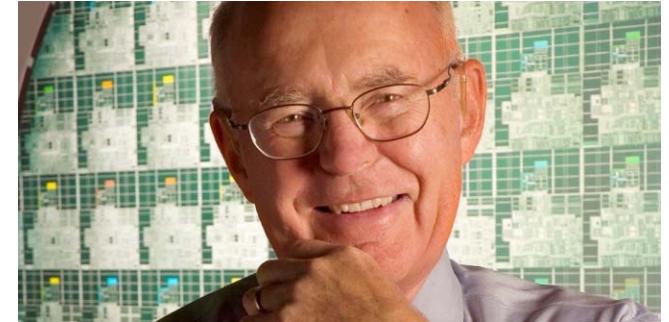
**bipolar transistor  
(single TR)**



**field-effect transistor  
(many TR)**

# Modern Integrated Circuits

- How many transistors do you see?
- How small are they?



- Simplest structure in a computing system is a switch

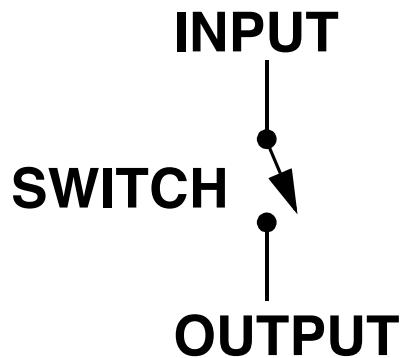
### **IDEAL SWITCH**



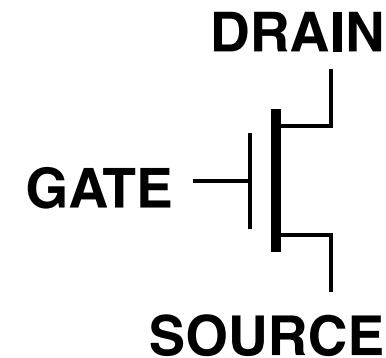
- Path exists between INPUT and OUTPUT if Switch is CLOSED or ON
- Path does not exist between INPUT and OUTPUT if SWITCH is OPEN or OFF

- The idea is to use the series and parallel switch configurations to route signals in a desired fashion.
- Unfortunately, it is difficult to implement an ideal switch as given.
- Complementary Metal Oxide Semiconductor (CMOS) devices give us some interesting components.

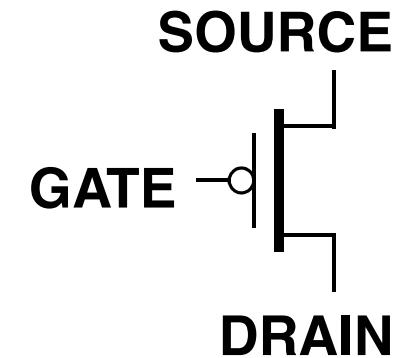
**IDEAL SWITCH**



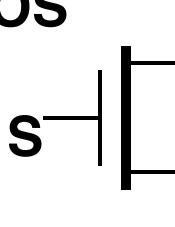
**nMOS transistor**



**pMOS transistor**



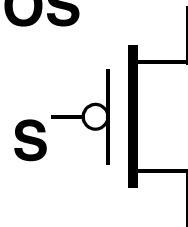
**nMOS**



S	SWITCH
0	OPEN
1	CLOSED

- nMOS when CLOSED
  - Transmits logic level 0 well
  - Transmits logic level 1 poorly

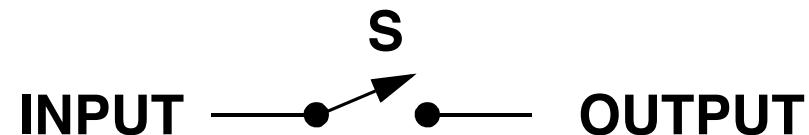
**pMOS**



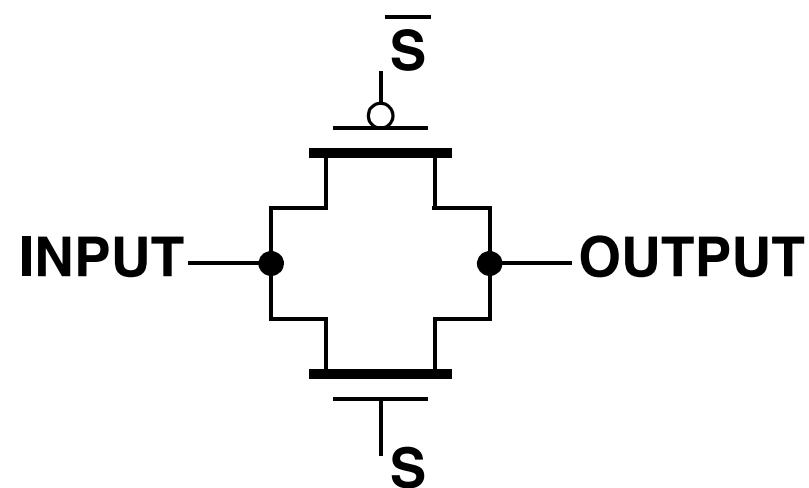
S	SWITCH
0	CLOSED
1	OPEN

- pMOS when CLOSED
  - Transmits logic level 1 well
  - Transmits logic level 0 poorly

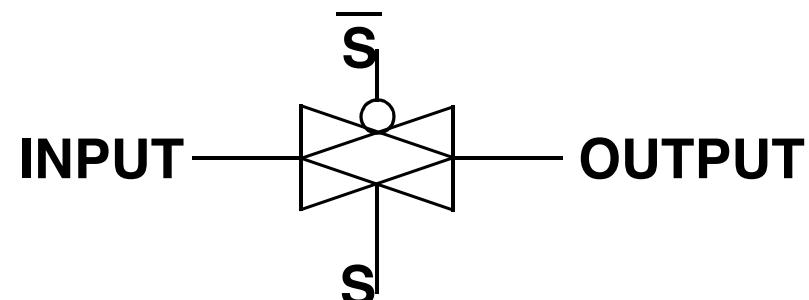
IDEAL SWITCH



CMOS TRANSMISSION GATE  
(SWITCH)

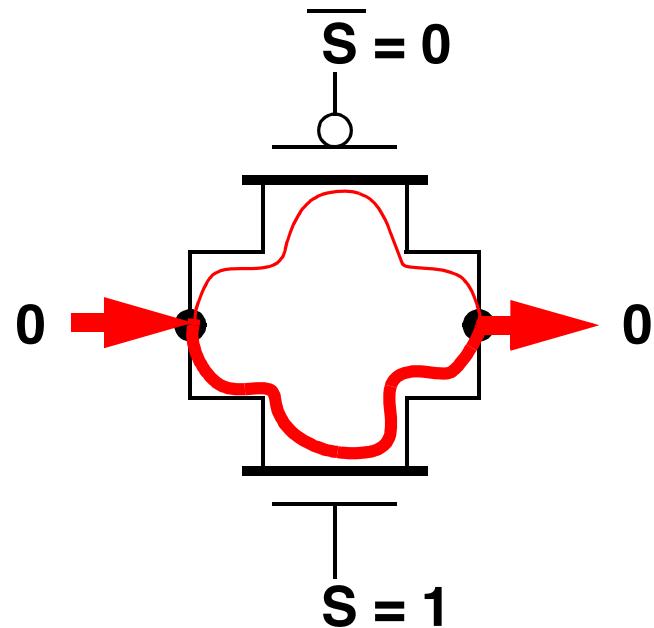


S	nMOS	pMOS	OUTPUT
0	OFF	OFF	Z
1	ON	ON	INPUT

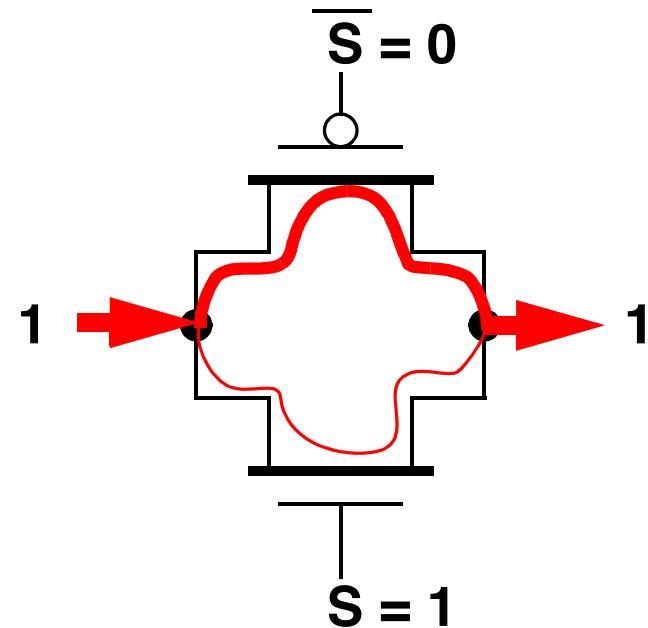


## SPLIT OF CURRENT ACROSS A TRANSMISSION GATE FOR LOGIC-0 AND LOGIC-1 INPUT

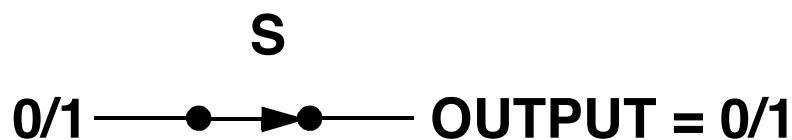
LOGIC-0 AT INPUT



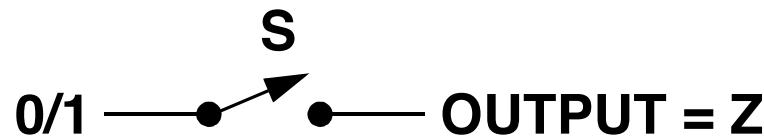
LOGIC-1 AT INPUT



- With switches, we can consider three states for an output:
  - Logic-0
  - Logic-1
  - High Impedance **Z**
- Path exists for Logic-0 and Logic-1 when the switch is CLOSED.

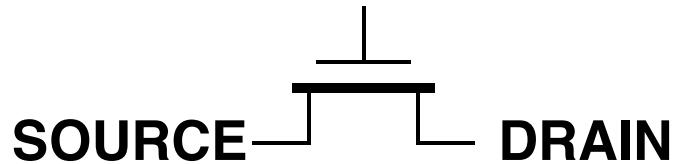


- High impedance is a state where the switch is OPEN.



- Another way of thinking of switches is as follows
  - Path exists for Logic-0 and Logic-1 when the switch is CLOSED, meaning that the **impedance/resistance is small** enough to allow ample flow of current.

**1 = CLOSED**

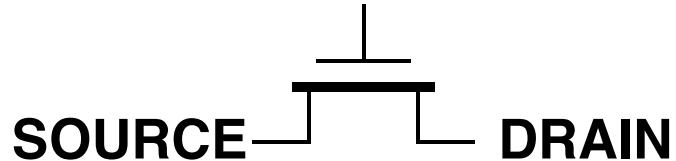


$\ll 10K\Omega$

**SOURCE** **DRAIN**

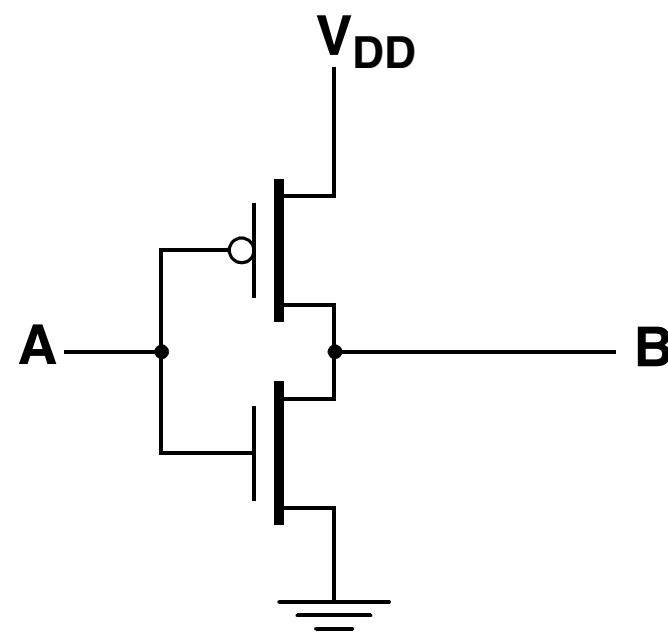
- High impedance is a state where the switch is OPEN, meaning that the **impedance/resistance is very large** allowing nearly no current flow.

**0 = OPEN**



$\gg 100M\Omega$

**SOURCE** **DRAIN**



$$B = \bar{A}$$

PULL-DOWN		PULL-UP	
A	B	A	B
0	Z	0	1
1	0	1	Z

→

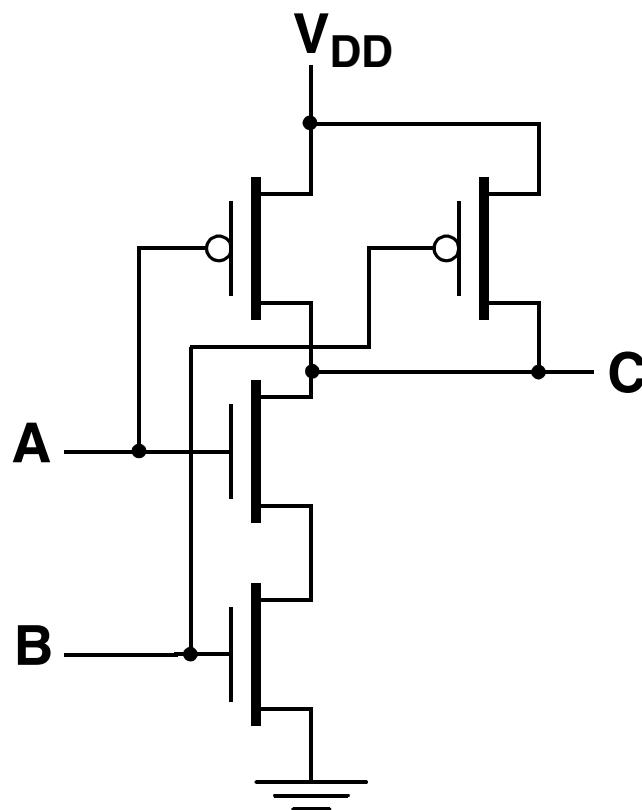
A	B
0	1
1	0

- This network inverts the binary input value.

# SWITCH NETWORKS

## NAND NETWORK

- CMOS
- SWITCH NETWORKS
- HIGH IMPEDANCE Z
- INVERTER



$$C = \overline{AB}$$

PULL-DOWN      PULL-UP

A	B	C	A	B	C
0	0	Z	0	0	1
0	1	Z	0	1	1
1	0	Z	1	0	1
1	1	0	1	1	Z

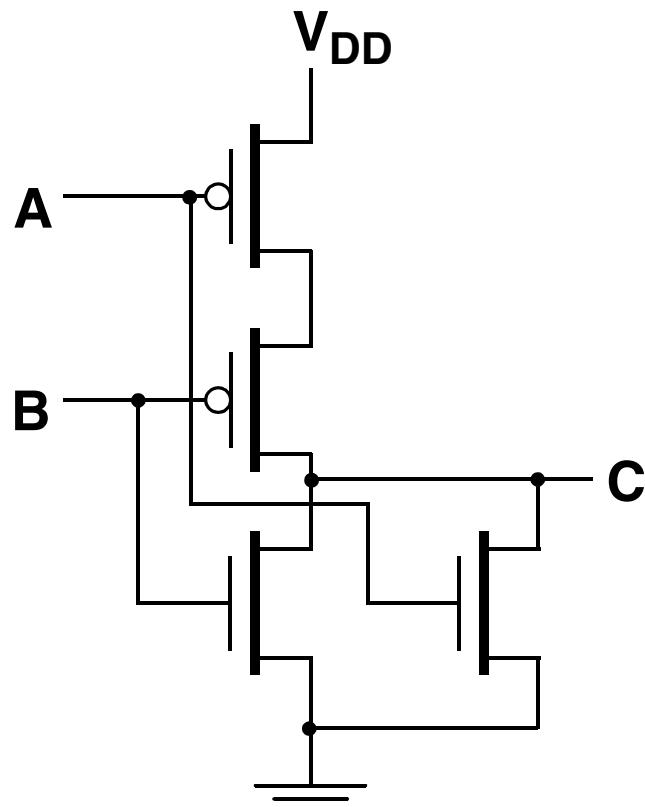


A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

# SWITCH NETWORKS

## NOR NETWORK

- SWITCH NETWORKS
  - HIGH IMPEDANCE Z
  - INVERTER
  - NAND NETWORK



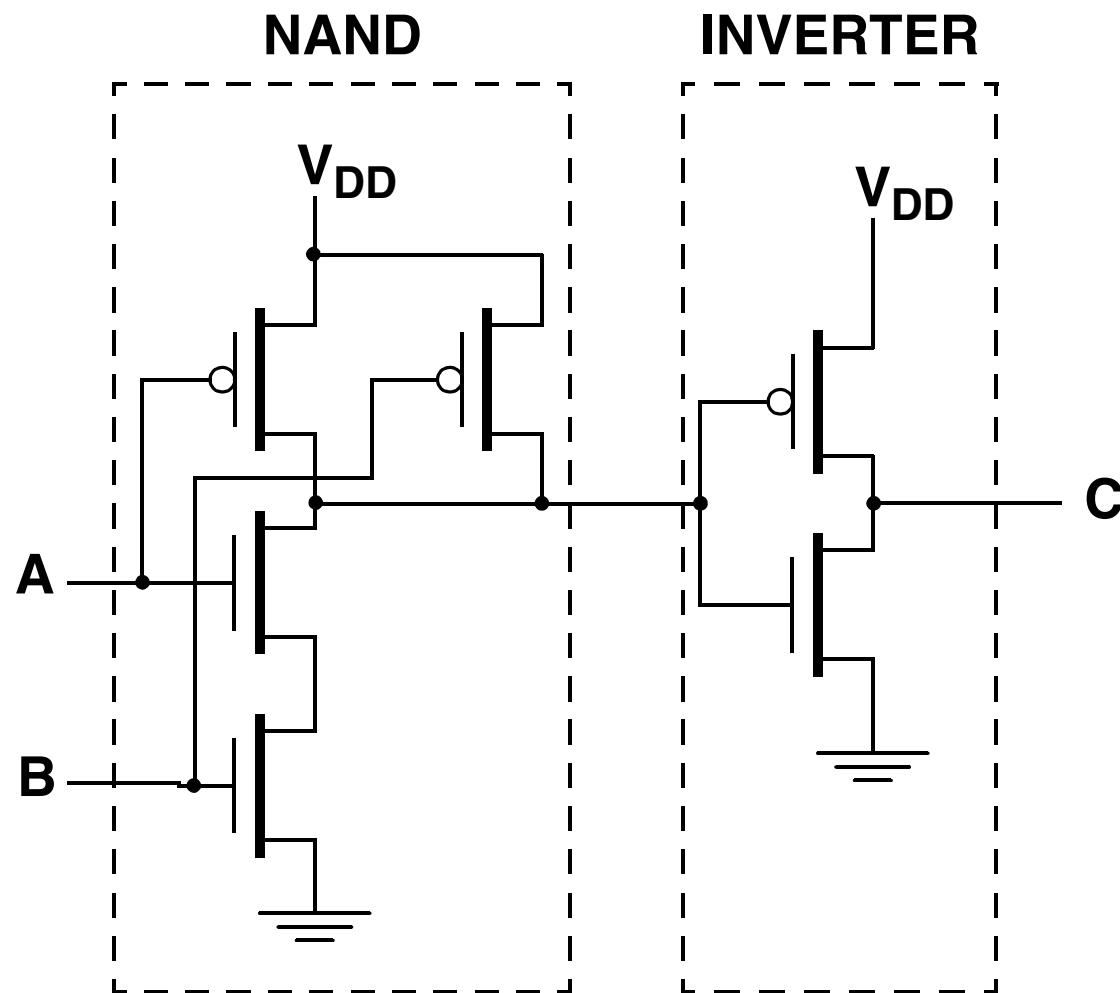
$$C = \overline{A + B}$$

PULL-DOWN      PULL-UP

A	B	C
0	0	Z
0	1	0
1	0	0
1	1	0

A	B	C
0	0	1
0	1	Z
1	0	Z
1	1	Z

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

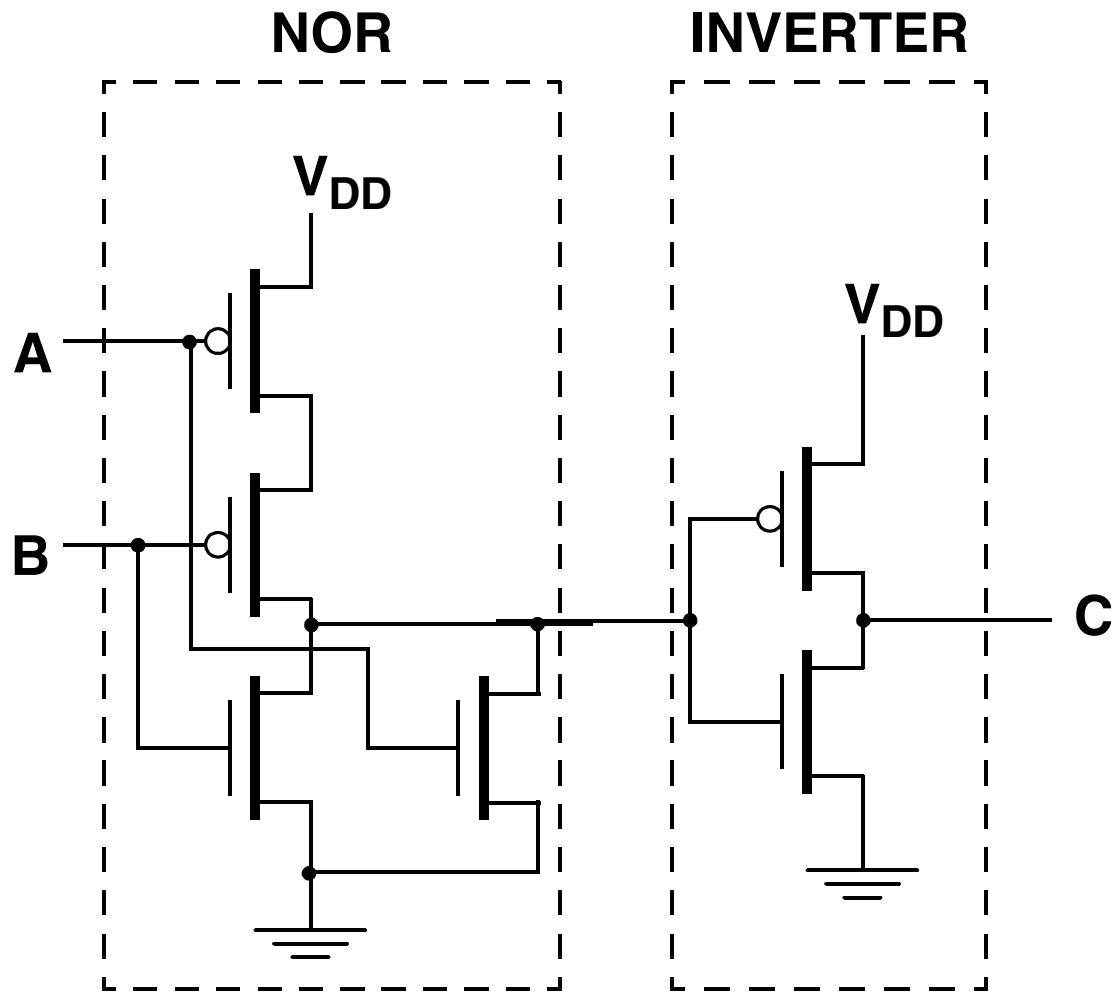


$$C = AB$$

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

# SWITCH NETWORKS OR NETWORK

- SWITCH NETWORKS
  - NAND NETWORK
  - NOR NETWORK
  - AND NETWORK

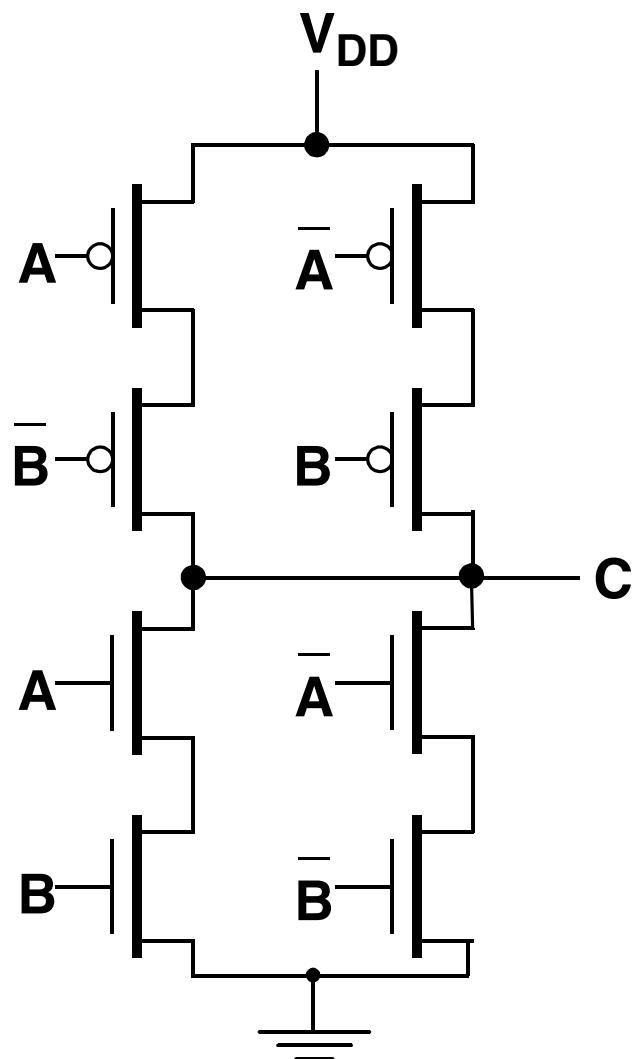


$$C = A + B$$

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

# SWITCH NETWORKS

## XOR NETWORK



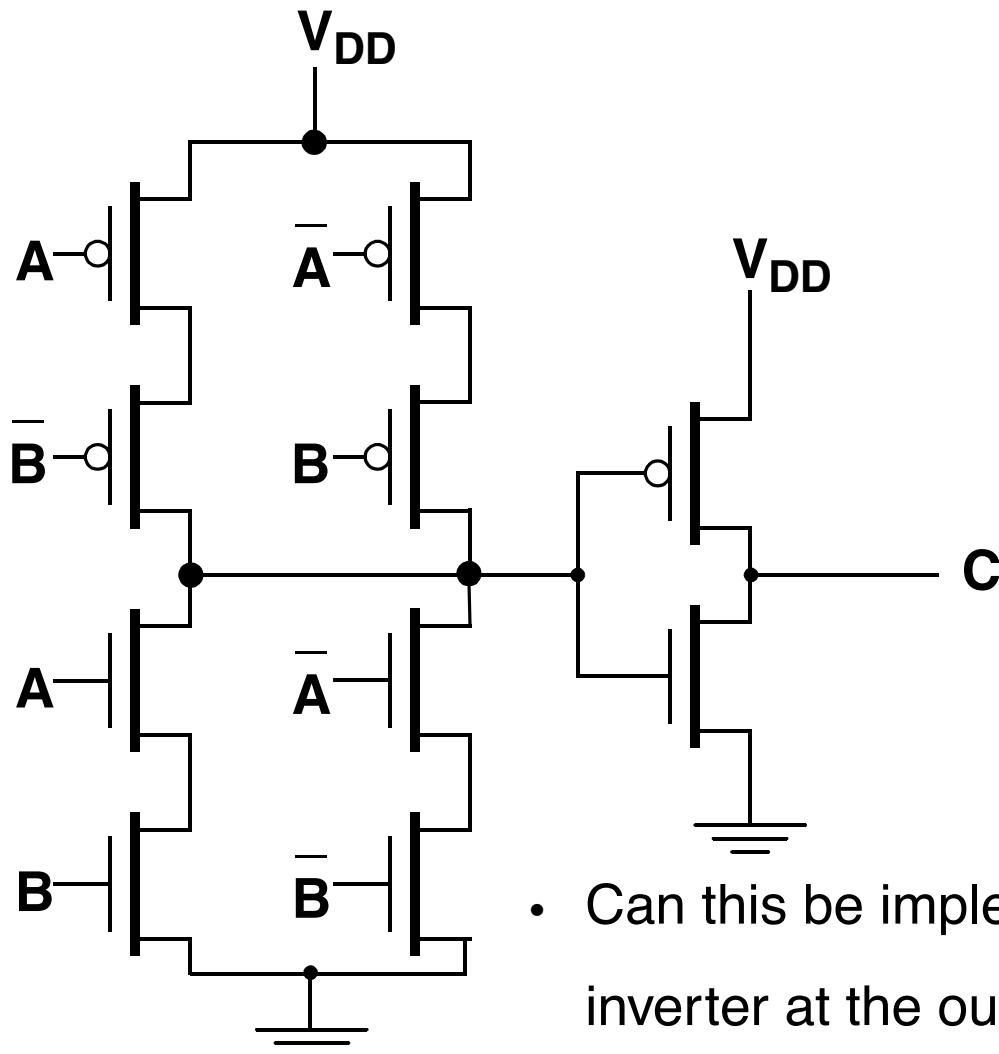
$$C = \bar{A}\bar{B} + \bar{A}B + A\bar{B} + AB$$

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

# SWITCH NETWORKS

## XNOR NETWORK

- SWITCH NETWORKS
  - AND NETWORK
  - OR NETWORK
  - XOR NETWORK



$$C = \overline{\overline{AB} + \overline{AB}}$$

A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

- Can this be implemented without the extra inverter at the output? Answer: Yes!

# SWITCH NETWORKS

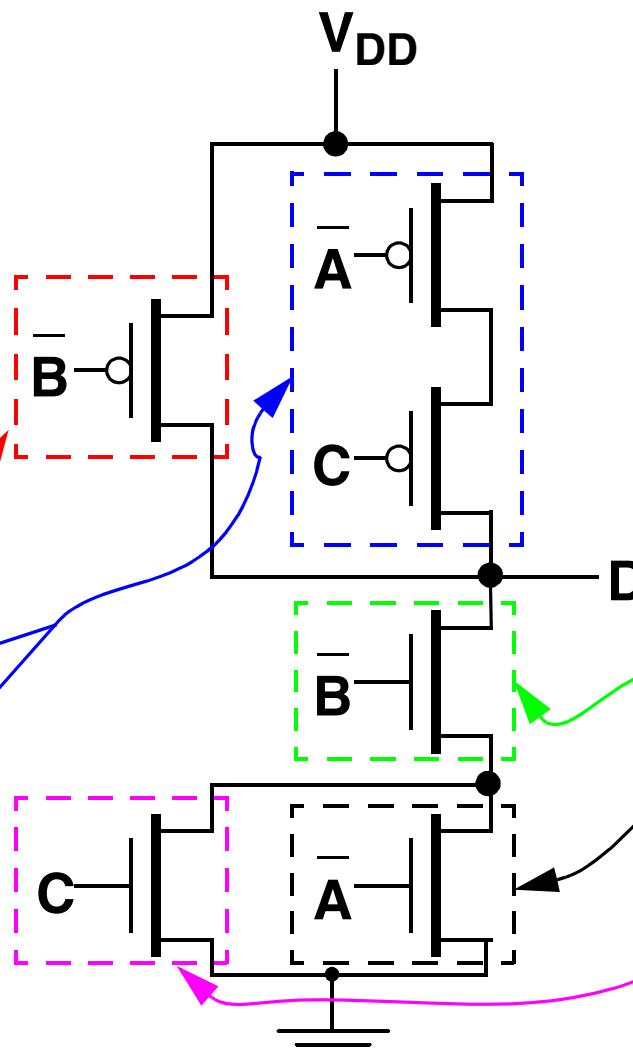
## PULL-UP/PULL-DOWN

- SWITCH NETWORKS
  - OR NETWORK
  - XOR NETWORK
  - XNOR NETWORK

$$D = \overline{AC} + B$$

PULL-UP

A	B	C	D
0	0	0	Z
0	0	1	Z
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	Z
1	1	0	1
1	1	1	1

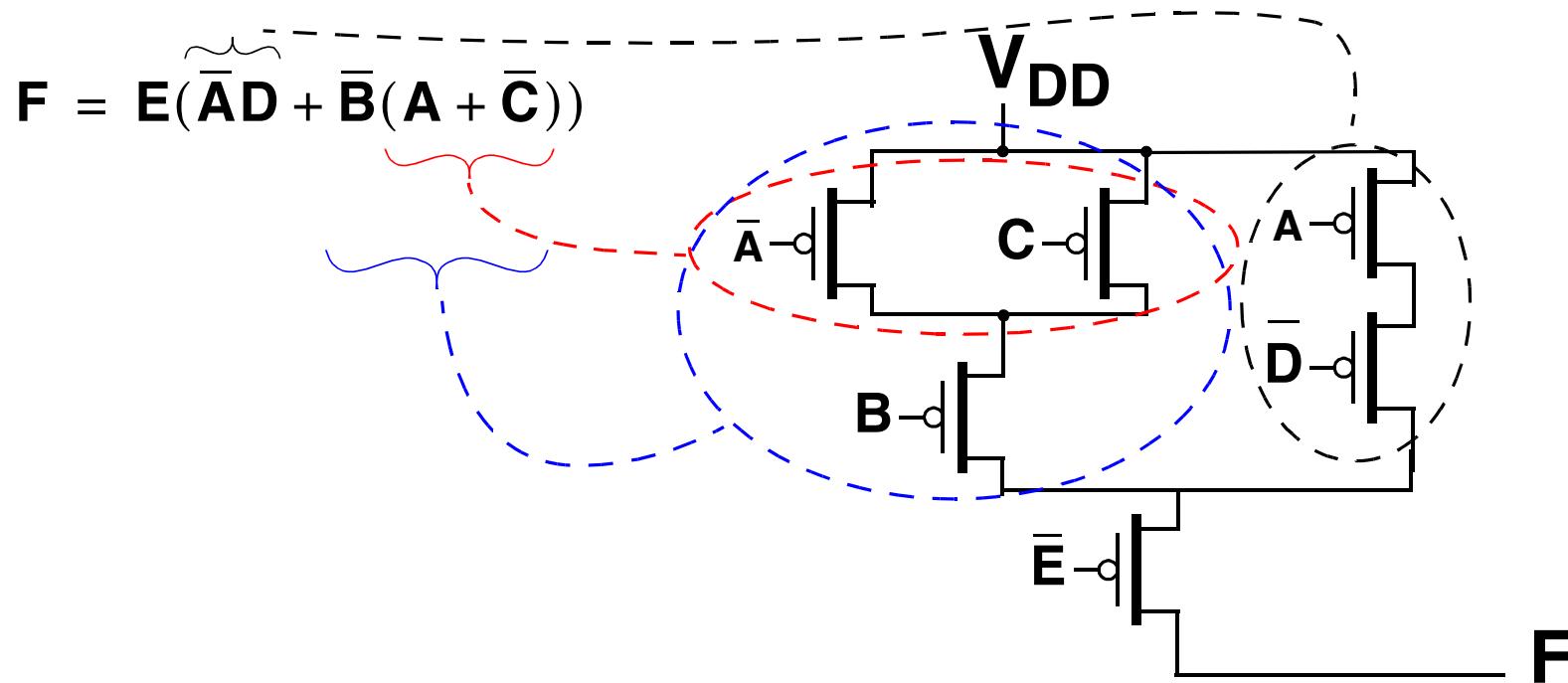


PULL-DOWN

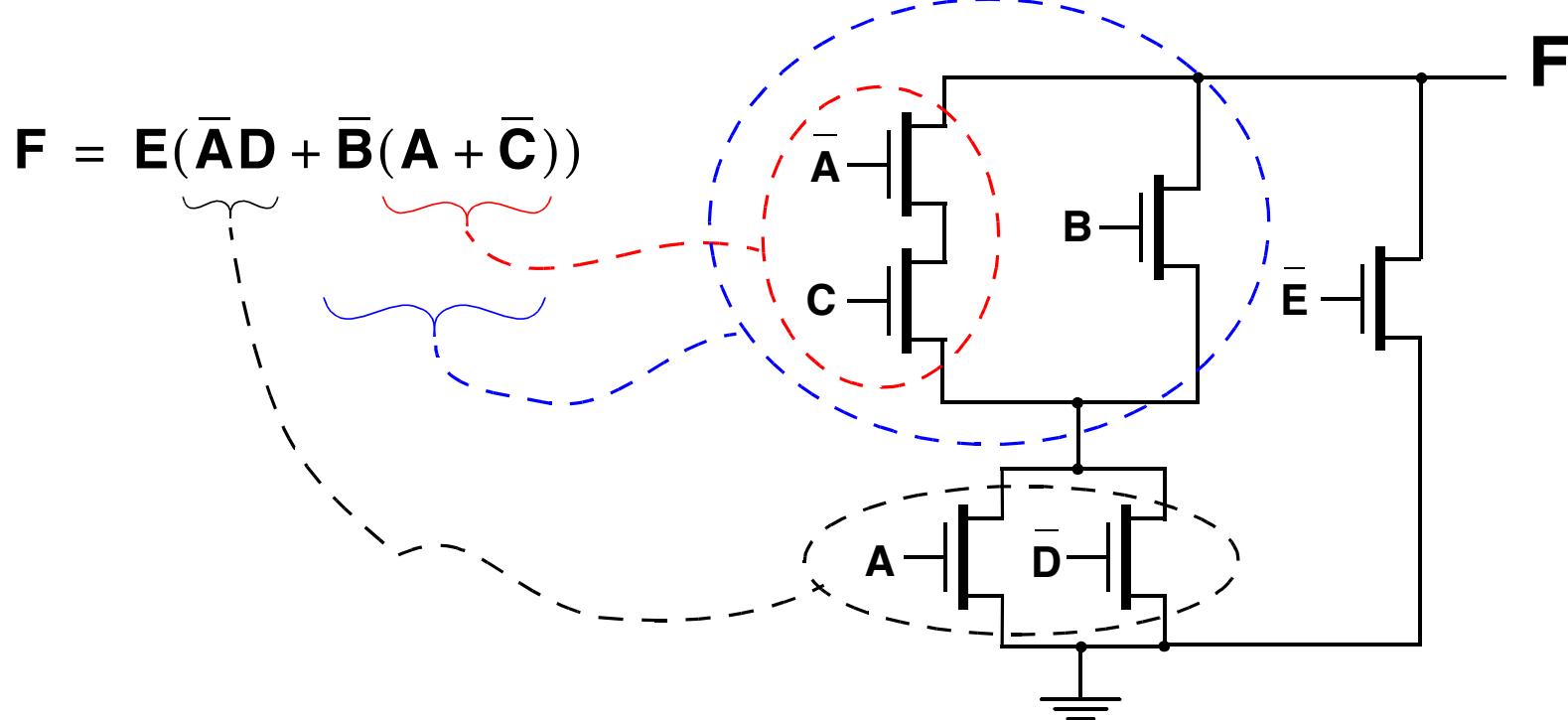
A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	Z
0	1	1	Z
1	0	0	Z
1	0	1	0
1	1	0	Z
1	1	1	Z

- Most Boolean functions can be easily implemented using switches.
- The basic rules are as follows
  - **Pull-up** section of switch network
    - **Use complements** for all literals in expression
    - Use only **pMOS devices**
    - Form **series** network for an **AND** operation
    - Form **parallel** network for an **OR** operation
  - **Pull-down** section of switch network
    - **Use complements** for all literals in expression
    - Use only **nMOS devices**
    - Form **parallel** network for an **AND** operation
    - Form **series** network for an **OR** operation

- To implement the Boolean function given below, the following pull-up network could be designed.

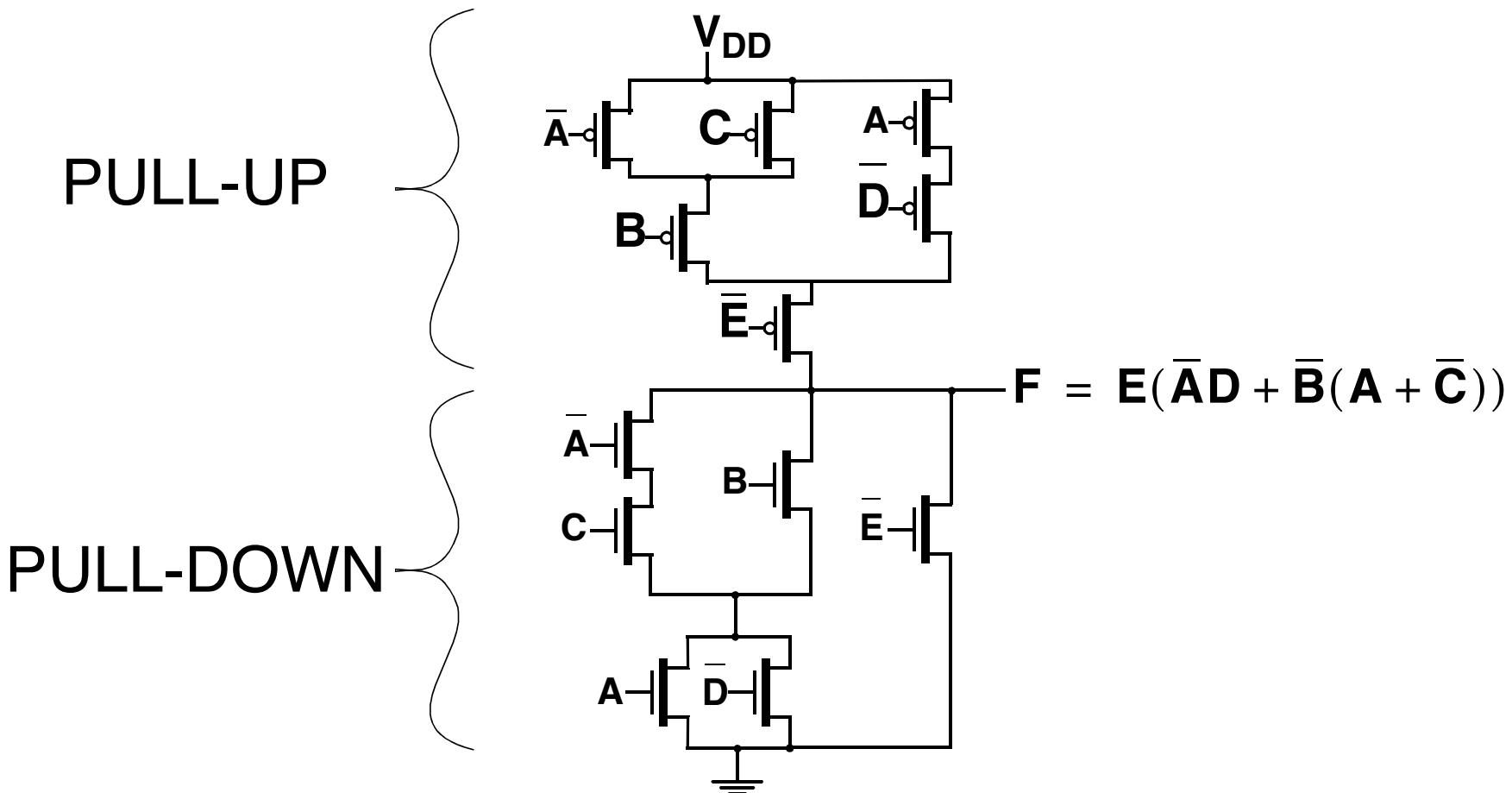


- To complete the switch design, the pull-down section for the Boolean function must also be designed.



- Notice how **AND** and **OR** become **OR** and **AND** circuits, respectively.

- Putting the pull-up and pull-down pieces together gives the following CMOS switch implementation of the Boolean function.



**INTRO. TO COMP. ENG.  
CHAPTER III-1  
BOOLEAN ALGEBRA**

**•CHAPTER III**

# **CHAPTER III**

## **BOOLEAN ALGEBRA**

- Boolean algebra is a form of algebra that deals with single digit binary values and variables.
- Values and variables can indicate some of the following binary pairs of values:
  - ON / OFF
  - TRUE / FALSE
  - HIGH / LOW
  - CLOSED / OPEN
  - 1 / 0

- Three fundamental operators in Boolean algebra
  - **NOT**: unary operator that complements represented as  $\bar{A}$ ,  $A'$ , or  $\sim A$
  - **AND**: binary operator which performs logical multiplication
    - i.e. **A ANDed with B** would be represented as  $AB$  or  $A \cdot B$
  - **OR**: binary operator which performs logical addition
    - i.e. **A ORed with B** would be represented as  $A + B$

**NOT**

A	$\bar{A}$
0	1
1	0

**AND**

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

**OR**

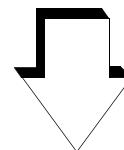
A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

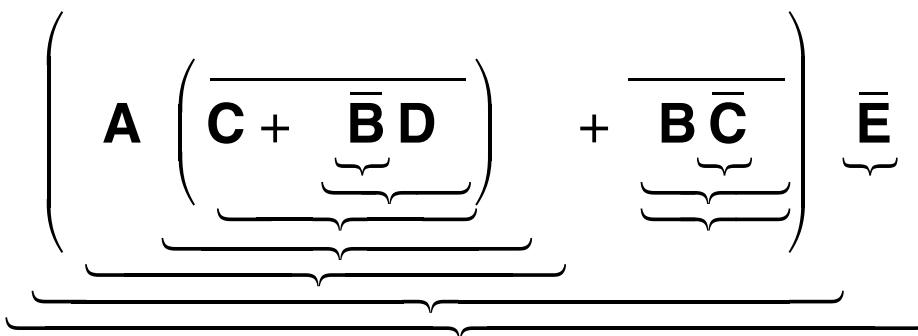
- Below is a table showing all possible Boolean functions  $F_N$  given the two-inputs **A** and **B**.

- Boolean expressions must be evaluated with the following order of operator precedence
  - parentheses
  - NOT
  - AND
  - OR

Example:

$$F = (A(\overline{C + \overline{B}D}) + \overline{B\overline{C}})\overline{E}$$



$$F = \left( A \left( \overline{C + \overline{B}D} \right) + \overline{B\overline{C}} \right ) \overline{E}$$


- Example 1:

Evaluate the following expression when  $A = 1$ ,  $B = 0$ ,  $C = 1$

$$F = C + \bar{C}B + B\bar{A}$$

- Solution

$$F = 1 + \bar{1} \cdot 0 + 0 \cdot \bar{1} = 1 + 0 + 0 = 1$$

- Example 2:

Evaluate the following expression when  $A = 0$ ,  $B = 0$ ,  $C = 1$ ,  $D = 1$

$$F = D(\overline{B\bar{C}A} + \overline{(AB + C)}) + C$$

- Solution

$$F = 1 \cdot (0 \cdot \bar{1} \cdot 0 + \overline{(0 \cdot 0 + 1)}) + 1 = 1 \cdot (0 + \bar{1} + 1) = 1 \cdot 1 = 1$$

# BOOLEAN ALGEBRA

## BASIC IDENTITIES

$$X + 0 = X$$

$$X \cdot 1 = X$$

Identity

$$X + 1 = 1$$

$$X \cdot 0 = 0$$

$$X + X = X$$

$$X \cdot X = X$$

Idempotent Law

$$X + X' = 1$$

$$X \cdot X' = 0$$

Complement

$$(X')' = X$$

Involution Law

$$X + Y = Y + X$$

$$XY = YX$$

Commutativity

$$X + (Y + Z) = (X + Y) + Z$$

$$X(YZ) = (XY)Z$$

Associativity

$$X(Y + Z) = XY + XZ$$

$$X + YZ = (X + Y)(X + Z)$$

Distributivity

$$X + XY = X$$

$$X(X + Y) = X$$

Absorption Law

$$X + X'Y = X + Y$$

$$X(X' + Y) = XY$$

Simplification

$$(X + Y)' = X'Y'$$

$$(XY)' = X' + Y'$$

DeMorgan's Law

$$\begin{aligned} XY + X'Z + YZ \\ = XY + X'Z \end{aligned}$$

$$\begin{aligned} & (X + Y)(X' + Z)(Y + Z) \\ & = (X + Y)(X' + Z) \end{aligned}$$

Consensus Theorem

- **Duality principle:**

- States that a Boolean equation remains valid if we take the dual of the expressions on both sides of the equals sign.
- The dual can be found by interchanging the **AND** and **OR** operators along with also interchanging the **0**'s and **1**'s.
- This is evident with the duals in the basic identities.
  - For instance: DeMorgan's Law can be expressed in two forms

$$(X + Y)' = X'Y' \quad \text{as well as} \quad (XY)' = X' + Y'$$

- Example: Simplify the following expression

$$F = BC + B\bar{C} + BA$$

- Simplification

$$F = B(C + \bar{C}) + BA$$

$$F = B \cdot 1 + BA$$

$$F = B(1 + A)$$

$$F = B$$

- Example: Simplify the following expression

$$F = A + \bar{A}B + \bar{A}\bar{B}C + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}\bar{C}\bar{D}E$$

- Simplification

$$F = A + \bar{A}(B + \bar{B}C + \bar{B}\bar{C}D + \bar{B}\bar{C}\bar{D}E)$$

$$F = A + B + \bar{B}C + \bar{B}\bar{C}D + \bar{B}\bar{C}\bar{D}E$$

$$F = A + B + \bar{B}(C + \bar{C}D + \bar{C}\bar{D}E)$$

$$F = A + B + C + \bar{C}D + \bar{C}\bar{D}E$$

$$F = A + B + C + \bar{C}(D + \bar{D}E)$$

$$F = A + B + C + D + \bar{D}E$$

$$F = A + B + C + D + E$$

- Example: Show that the following equality holds

$$\overline{A(\bar{B}\bar{C} + BC)} = \bar{A} + (\bar{B} + C)(\bar{B} + \bar{C})$$

- Simplification

$$\begin{aligned}\overline{A(\bar{B}\bar{C} + BC)} &= \bar{A} + \overline{(\bar{B}\bar{C} + BC)} \\ &= \bar{A} + (\overline{\bar{B}\bar{C}})(\overline{BC}) \\ &= \bar{A} + (\bar{B} + C)(\bar{B} + \bar{C})\end{aligned}$$

- Boolean expressions can be manipulated into many forms.
- Some standardized forms are required for Boolean expressions to simplify communication of the expressions.
- **Sum-of-products (SOP)**
  - Example:

$$F(A, B, C, D) = AB + \bar{B}C\bar{D} + AD$$

- **Products-of-sums (POS)**
  - Example:

$$F(A, B, C, D) = (A + B)(\bar{B} + C + \bar{D})(A + D)$$

# STANDARD FORMS

## MINTERMS

- The following table gives the minterms for a **three-input** system

A	B	C	$m_0$	$m_1$	$m_2$	$m_3$	$m_4$	$m_5$	$m_6$	$m_7$
			$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}BC$	$\bar{A}B\bar{C}$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	$ABC$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

- **Sum-of-minterms** standard form expresses the Boolean or switching expression in the form of a **sum of products** using **minterms**.
  - For instance, the following Boolean expression using minterms

$$F(A, B, C) = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + A\overline{B}\overline{C} + A\overline{B}C$$

could instead be expressed as

$$F(A, B, C) = m_0 + m_1 + m_4 + m_5$$

or more compactly

$$F(A, B, C) = \sum m(0, 1, 4, 5) = \text{one-set}(0, 1, 4, 5)$$

# STANDARD FORMS

## MAXTERMS

- The following table gives the maxterms for a **three-input** system

	$M_0$	$M_1$	$M_2$	$M_3$	$M_4$	$M_5$	$M_6$	$M_7$
<b>A    B    C</b>	$A + B + C$	$A + \bar{B} + C$	$\bar{A} + B + C$	$\bar{A} + \bar{B} + C$	$A + B + \bar{C}$	$A + \bar{B} + \bar{C}$	$\bar{A} + B + \bar{C}$	$\bar{A} + \bar{B} + \bar{C}$
	0    0    0	0    1    1    1    1    1    1    1	1    0    1    1    1    1    1    1	1    1    0    1    1    1    1    1	1    1    1    0    1    1    1    1	1    1    1    1    0    1    1    1	1    1    1    1    1    0    1    1	1    1    1    1    1    1    0    1
0    0    1	1	0	1	1	1	1	1	1
0    1    0	1	1	0	1	1	1	1	1
0    1    1	1	1	1	0	1	1	1	1
1    0    0	1	1	1	1	0	1	1	1
1    0    1	1	1	1	1	1	0	1	1
1    1    0	1	1	1	1	1	1	0	1
1    1    1	1	1	1	1	1	1	1	0

# STANDARD FORMS

## PRODUCT OF MAXTERMS

- **Product-of-maxterms** standard form expresses the Boolean or switching expression in the form of **product of sums** using **maxterms**.
  - For instance, the following Boolean expression using maxterms

$$F(A, B, C) = (A + B + \bar{C})(\bar{A} + B + C)(\bar{A} + \bar{B} + \bar{C})$$

could instead be expressed as

$$F(A, B, C) = M_1 \cdot M_4 \cdot M_7$$

or more compactly as

$$F(A, B, C) = \prod M(1, 4, 7) = \text{zero-set}(1, 4, 7)$$

- Given an arbitrary Boolean function, such as

$$F(A, B, C) = AB + \bar{B}(\bar{A} + \bar{C})$$

how do we form the canonical form for:

- **sum-of-minterms**
  - Expand the Boolean function into a sum of products. Then take each term with a missing variable **X** and **AND** it with **X + X̄**.
- **product-of-maxterms**
  - Expand the Boolean function into a product of sums. Then take each factor with a missing variable **X** and **OR** it with **XX̄**.

# STANDARD FORMS

## FORMING SUM OF MINTERMS

- Example

$$\begin{aligned}
 F(A, B, C) &= AB + \bar{B}(\bar{A} + \bar{C}) = AB + \bar{A}\bar{B} + \bar{B}\bar{C} \\
 &= AB(C + \bar{C}) + \bar{A}\bar{B}(C + \bar{C}) + (A + \bar{A})\bar{B}\bar{C} \\
 &= \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC + ABC \\
 &= \sum m(0, 1, 4, 6, 7)
 \end{aligned}$$

A	B	C	F
0	0	0	1 ← 0
0	0	1	1 ← 1
0	1	0	0
0	1	1	0
1	0	0	1 ← 4
1	0	1	0
1	1	0	1 ← 6
1	1	1	1 ← 7

Minterms listed as  
1s in Truth Table

# STANDARD FORMS

## FORMING PROD OF MAXTERMS

- Example

$$\begin{aligned}
 F(A, B, C) &= AB + \bar{B}(\bar{A} + \bar{C}) = AB + \bar{A}\bar{B} + \bar{B}\bar{C} \\
 &= (A + \bar{B})(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C}) \quad (\text{using distributivity}) \\
 &= (A + \bar{B} + C\bar{C})(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C}) \\
 &= (A + \bar{B} + C)(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C}) \\
 &= \prod M(2, 3, 5)
 \end{aligned}$$

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0 ← 2
0	1	1	0 ← 3
1	0	0	1
1	0	1	0 ← 5
1	1	0	1
1	1	1	1

Maxterms listed as  
0s in Truth Table

# STANDARD FORMS

## CONVERTING MIN AND MAX

- Converting between sum-of-minterms and product-of-maxterms
  - The two are complementary, as seen by the truth tables.
  - To convert interchange the  $\sum$  and  $\prod$ , then use missing terms.
    - Example: The example from the previous slides

$$F(A, B, C) = \sum m(0, 1, 4, 6, 7)$$

is re-expressed as

$$F(A, B, C) = \prod M(2, 3, 5)$$

where the numbers 2, 3, and 5 were missing from the minterm representation.

# SIMPLIFICATION

## KARNAUGH MAPS

- Often it is desired to simplify a Boolean function. A quick graphical approach is to use Karnaugh maps.

2-variable  
Karnaugh map

A	B	0	1
0	0	0	0
1	0	1	

$$F = AB$$

3-variable  
Karnaugh map

A	BC	00	01	11	10
0	0	0	1	1	0
1	0	1	1	1	1

$$F = AB + C$$

4-variable  
Karnaugh map

CD	AB	00	01	11	10
00	00	0	1	0	0
01	01	0	1	0	0
11	11	1	1	1	1
10	10	0	1	0	0

$$F = AB + \bar{C}D$$

- Notice that the ordering of cells in the map are such that moving from one cell to an adjacent cell only changes one variable.

**2-variable**  
Karnaugh map

	B	0	1
0	A	0	1
1	A	2	3

$\bar{B}$     B

**3-variable**  
Karnaugh map

	BC	$\bar{C}$	C	$\bar{C}$
0	00	01	11	10
1	0	1	3	2

$\bar{B}$     B

**4-variable**  
Karnaugh map

	CD	$\bar{D}$	D	$\bar{D}$
AB	00	01	11	10
$\bar{A}$	00	0	1	3
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

$\bar{A}$     A

$\bar{B}$     B

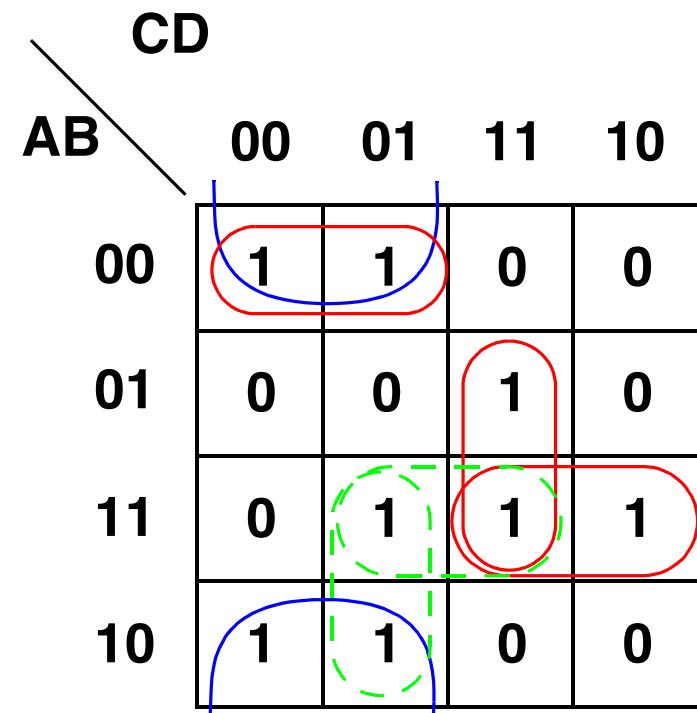
$\bar{C}$     C

- This ordering allows for grouping of minterms/maxterms for simplification.

# SIMPLIFICATION

## IMPLICANTS

- **Implicant**
  - Bubble covering only 1s (size of bubble must be a power of 2).
- **Prime implicant**
  - Bubble that is expanded as big as possible (but increases in size by powers of 2).
- **Essential prime implicant**
  - Bubble that contains a 1 covered only by itself and no other prime implicant bubble.
- **Non-essential prime implicant**
  - A 1 that can be bubbled by more than one prime implicant bubble.



# SIMPLIFICATION

## PROCEDURE FOR SOP

- **Procedure for finding the SOP from a Karnaugh map**
  - Step 1: Form the 2-, 3-, or 4-variable Karnaugh map as appropriate for the Boolean function.
  - Step 2: Identify all essential prime implicants for **1s** in the Karnaugh map
  - Step 3: Identify non-essential prime implicants for **1s** in the Karnaugh map.
  - Step 4: For each essential and one selected non-essential prime implicant from each set, determine the corresponding product term.
  - Step 5: Form a sum-of-products with all product terms from previous step.

# SIMPLIFICATION

## EXAMPLE FOR SOP (1)

- Simplify the following Boolean function

$$F(A, B, C) = \sum m(0, 1, 4, 5) = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + A\bar{B}C$$

- Solution:

		BC	00	01	11	10
		A	0	1	0	0
0	0	1	1	0	0	
	1	1	1	0	0	

zero-set(2, 3, 6, 7)

one-set(0, 1, 4, 5)

- The essential prime implicants are  $\bar{B}$ .
- There are no non-essential prime implicants.
- The sum-of-products solution is  $F = \bar{B}$ .

# SIMPLIFICATION

## EXAMPLE FOR SOP (2)

- Simplify the following Boolean function

$$F(A, B, C) = \sum m(0, 1, 4, 6, 7) = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + AB\bar{C} + ABC$$

- Solution:

		BC	00	01	11	10	
		A	0	1	1	0	0
0	0	1	1	0	0		
	1	1	0	1	1		

*zero-set(2, 3, 5)*  
*one-set(0, 1, 4, 6, 7)*

- The essential prime implicants are  $\bar{A}\bar{B}$  and  $AB$ .
- The non-essential prime implicants are  $\bar{B}\bar{C}$  or  $A\bar{C}$ .
- The sum-of-products solution is

$$F = AB + \bar{A}\bar{B} + \bar{B}\bar{C} \text{ or } F = AB + \bar{A}\bar{B} + A\bar{C}.$$

- **Procedure for finding the SOP from a Karnaugh map**
  - Step 1: Form the 2-, 3-, or 4-variable Karnaugh map as appropriate for the Boolean function.
  - Step 2: Identify all essential prime implicants for **0s** in the Karnaugh map
  - Step 3: Identify non-essential prime implicants for **0s** in the Karnaugh map.
  - Step 4: For each essential and one selected non-essential prime implicant from each set, determine the corresponding sum term.
  - Step 5: Form a product-of-sums with all sum terms from previous step.

# SIMPLIFICATION

## EXAMPLE FOR POS (1)

- Simplify the following Boolean function

$$F(A, B, C) = \prod M(2, 3, 5) = (A + \bar{B} + C)(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})$$

- Solution:

		BC	00	01	11	10
		A	0	1	1	0
0	0	1	1	0	0	
	1	1	0	1	1	

*zero-set(2, 3, 5)  
one-set(0, 1, 4, 6, 7)*

- The essential prime implicants are  $\bar{A} + B + \bar{C}$  and  $A + \bar{B}$ .
- There are no non-essential prime implicants.
- The product-of-sums solution is  $F = (A + \bar{B})(\bar{A} + B + \bar{C})$ .

# SIMPLIFICATION

## EXAMPLE FOR POS (2)

- Simplify the following Boolean function

$$F(A, B, C) = \prod M(0, 1, 5, 7, 8, 9, 15)$$

- Solution:

- The essential prime implicants

*zero-set(0, 1, 5, 7, 8, 9, 15)*

are  $B + C$  and  $\bar{B} + \bar{C} + \bar{D}$ .

*one-set(2, 3, 4, 6, 10, 11, 12, 13, 14)*

- The non-essential prime implicants

can be  $A + \bar{B} + \bar{D}$  or  $A + C + \bar{D}$ .

- The product-of-sums solution can be either

$$F = (B + C)(\bar{B} + \bar{C} + \bar{D})(A + \bar{B} + \bar{D})$$

or

$$F = (B + C)(\bar{B} + \bar{C} + \bar{D})(A + C + \bar{D})$$

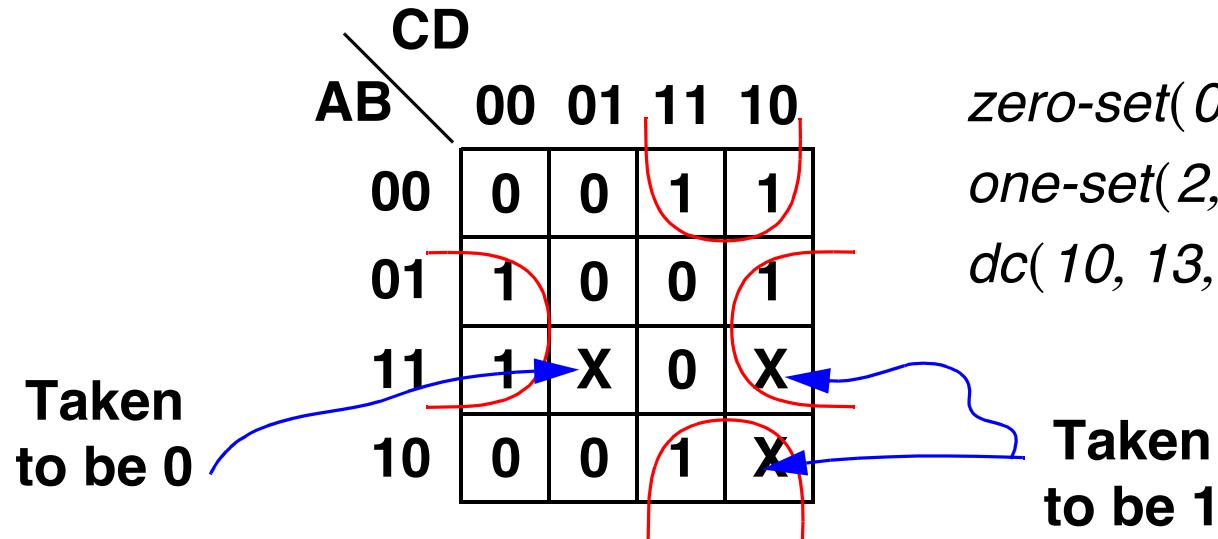
		CD \ AB	00	01	11	10
		AB \ CD	00	01	11	10
A	B	00	0	0	1	1
		01	1	0	0	1
A	B	11	1	1	0	1
		10	0	0	1	1

- Switching expressions are sometimes given as **incomplete**, or with **don't-care conditions**.
  - Having don't-care conditions can simplify Boolean expressions and hence simplify the circuit implementation.
  - Along with the *zero-set( )* and *one-set( )*, we will also have *dc( )*.
  - Don't-cares conditions in Karnaugh maps
    - Don't-cares will be expressed as an “X” or “-” in Karnaugh maps.
    - Don't-cares can be bubbled along with the **1s** or **0s** depending on what is more convenient and help simplify the resulting expressions.

# SIMPLIFICATION

## DON'T-CARE EXAMPLE (1)

- Find the SOP simplification for the following Karnaugh map



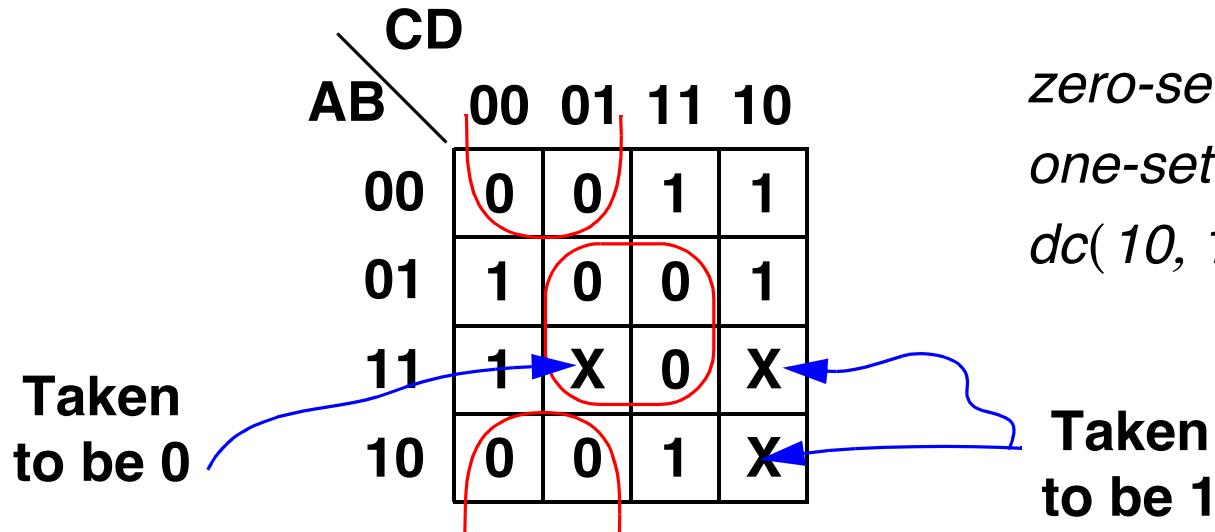
*zero-set(0, 1, 5, 7, 8, 9, 15)  
one-set(2, 3, 4, 6, 11, 12)  
dc( 10, 13, 14)*

- Solution:
  - The essential prime implicants are  $\bar{B}\bar{D}$  and  $\bar{B}C$ .
  - There are no non-essential prime implicants.
  - The sum-of-products solution is  $F = \bar{B}C + \bar{B}\bar{D}$ .

# SIMPLIFICATION

## DON'T-CARE EXAMPLE (2)

- Find the POS simplification for the following Karnaugh map



*zero-set(0, 1, 5, 7, 8, 9, 15)  
one-set(2, 3, 4, 6, 11, 12)  
dc(10, 13, 14)*

- Solution:
  - The essential prime implicants are  $B + C$  and  $\bar{B} + \bar{D}$ .
  - There are no non-essential prime implicants.
  - The product-of-sums solution is  $F = (B + C)(\bar{B} + \bar{D})$ .

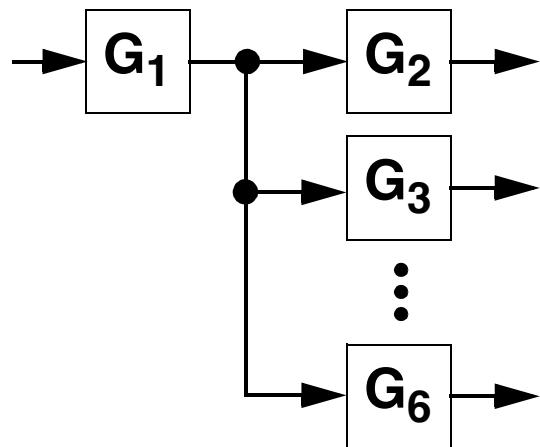
**INTRO. TO COMP. ENG.  
CHAPTER IV-1  
GATE DESIGN**

**•CHAPTER IV**

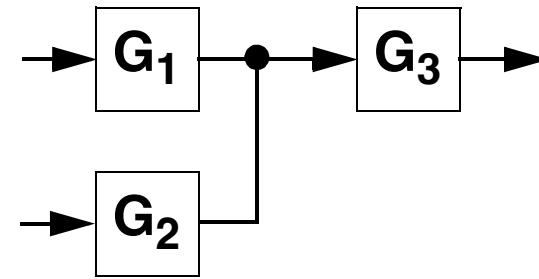
# **CHAPTER IV**

## **GATE DESIGN**

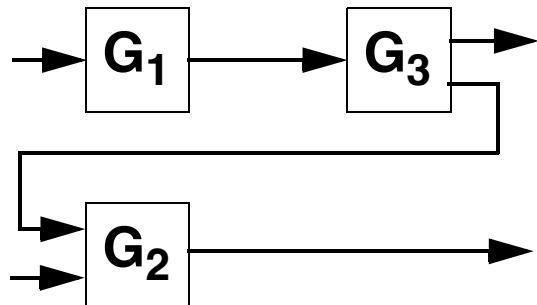
- Gate network consists of
  - Gates
  - External inputs and outputs
  - Connections
- Gate inputs
  - Only one connection to input is allowed (unless tri-state device is used)
    - Connected to constant value (**0** or **1**)
    - Connected to an external input
    - Connected to a gate output
- Gate outputs
  - Output load should not be greater than the fanout factor for the gate and technology being used.



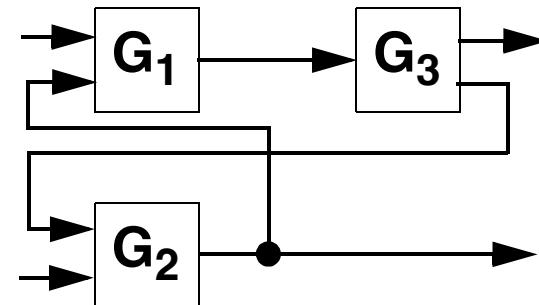
Valid or Invalid?



Valid or Invalid?



Valid or Invalid?

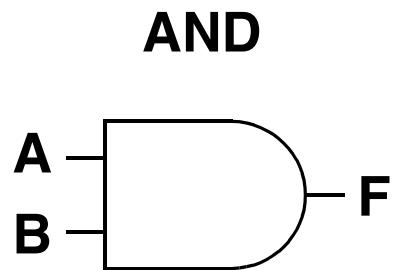


Valid or Invalid?

# LOGIC GATES

## NON-INVERTING OPERATORS

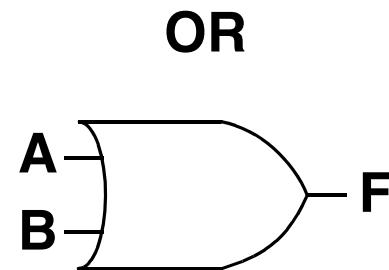
- GATE NETWORKS
  - INTRODUCTION
  - VALID/INVALID NET.



A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

$$F = AB$$

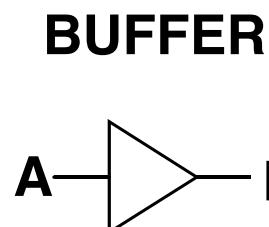
6 transistors



A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

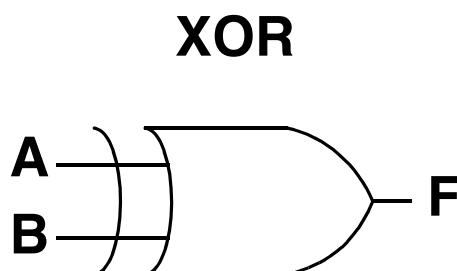
$$F = A + B$$

6 transistors



A	F
0	0
1	1

$$F = A$$



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

$$F = A\bar{B} + \bar{A}B = A \oplus B$$

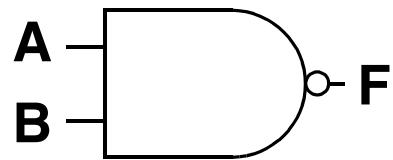
8 transistors

# LOGIC GATES

## INVERTING OPERATORS

- GATE NETWORKS
- LOGIC GATES
- NON-INVERTING OPER.

**NAND**

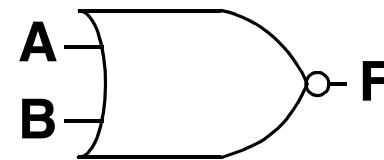


A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

$$F = \overline{AB}$$

4 transistors

**NOR**

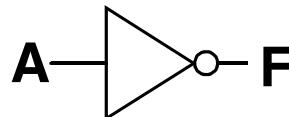


A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

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

4 transistors

**NOT**

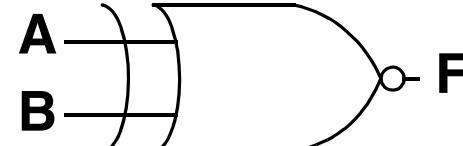


A	F
0	1
1	0

$$F = \overline{A}$$

2 transistors

**XNOR**



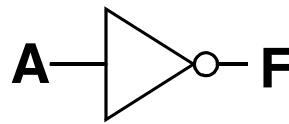
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

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

8 transistors

- Inverters can also be implemented with a **NAND** or with a **NOR** gate.

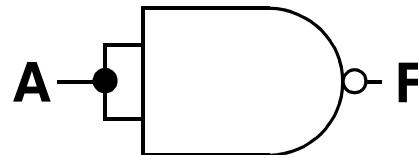
**NOT**



A	F
0	1
1	0

$$F = \bar{A}$$

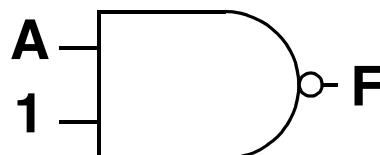
**NAND**



A	A	F
0	0	1
1	1	0

$$F = \overline{AA} = \bar{A}$$

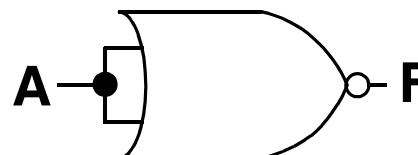
**NAND**



A	1	F
0	1	1
1	1	0

$$F = \overline{A \cdot 1} = \bar{A}$$

**NOR**



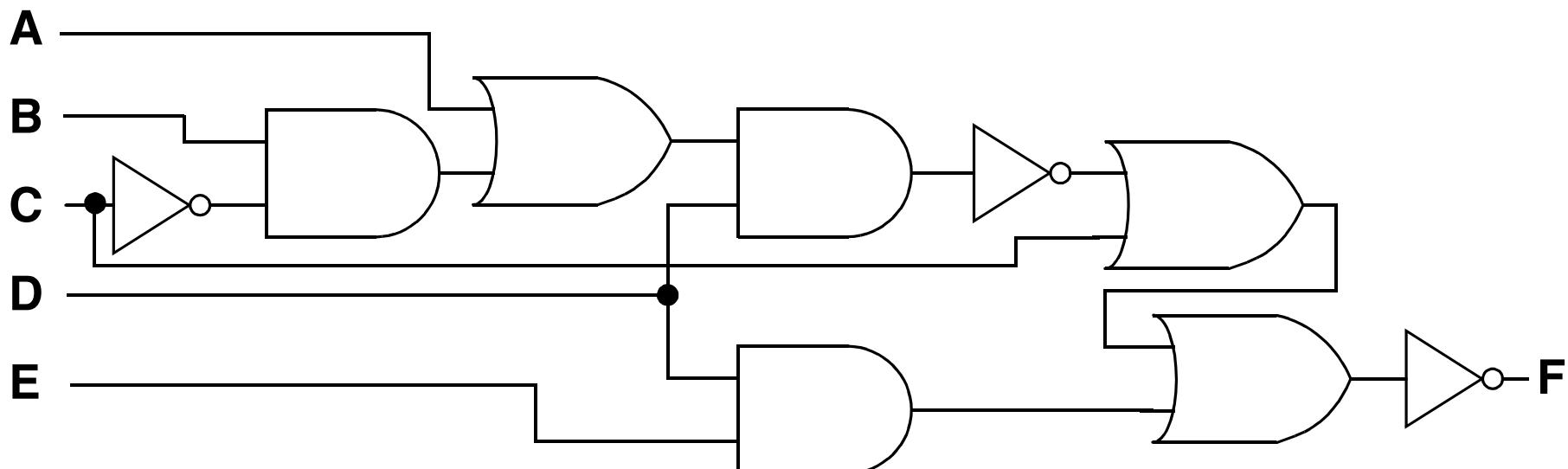
A	A	F
0	0	1
1	1	0

$$F = \overline{A + A} = \bar{A}$$

- Implement the following Boolean function using logic gates

$$F = \overline{((A + B\bar{C})D)} + C + DE$$

- Possible solution:



- $3 \times 6_{AND} + 3 \times 6_{OR} + 3 \times 2_{NOT} = 42$  transistors for CMOS technology.

# LOGIC NETWORKS

## USING SPECIFIC GATES

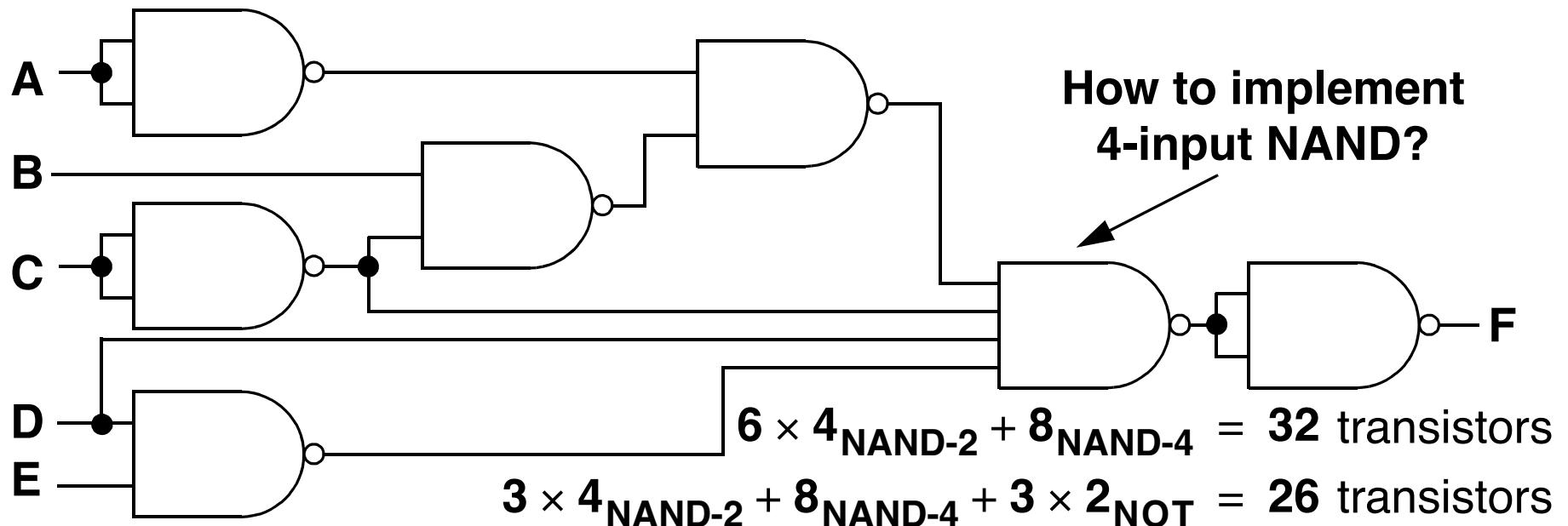
- Because of various implementation reasons, it may be desired to use only specific sorts of logic gates in an implementation.
  - For instance, many CMOS implementations use only **NAND** gates. Some implementations use on **NOR** gates.
  - This can be done in a number of manners. One is to rework the Boolean functions so that only the specific gates desired are used.
  - May reduce the physical number of transistors required if the appropriate types of gates are used.

- Implement the following Boolean function using NAND gates

$$F = \overline{((A + B\bar{C})D)} + C + DE$$

- This Boolean function can be expressed as

$$F = \overline{\overline{ABC}\overline{DC}\overline{DE}} = (((A'(BC'))'')D)(C')(DE)'')$$

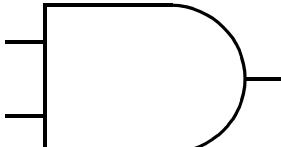
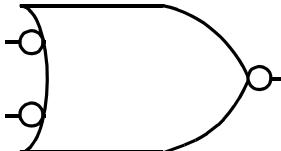
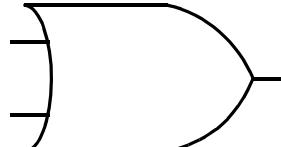
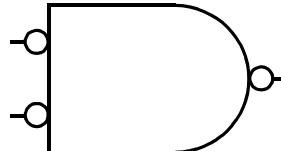
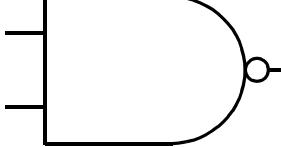
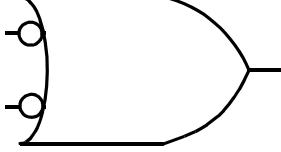
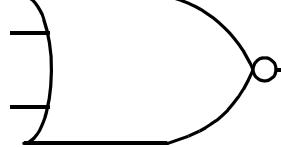
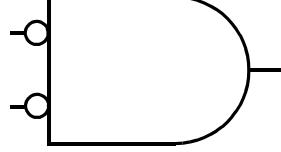


# MIXED LOGIC

## INTRODUCTION

- Mixed logic is one approach that makes it easier to **redesign** a logic network to use **desired types of gates**.
- Mixed logic is also **self-documenting**
  - This means that you can see what the original designer started with and see how the logic network was changed for the implementation.
  - The idea behind mixed logic is to diagram out the logic network from the Boolean equations given, and then make small changes to the logic network to achieve desired results for implementation.

- DeMorgan's Square

AND	OR
 	 
NAND	NOR
 	 

**AND**

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

**OR**

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

**NAND**

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

**NOR**

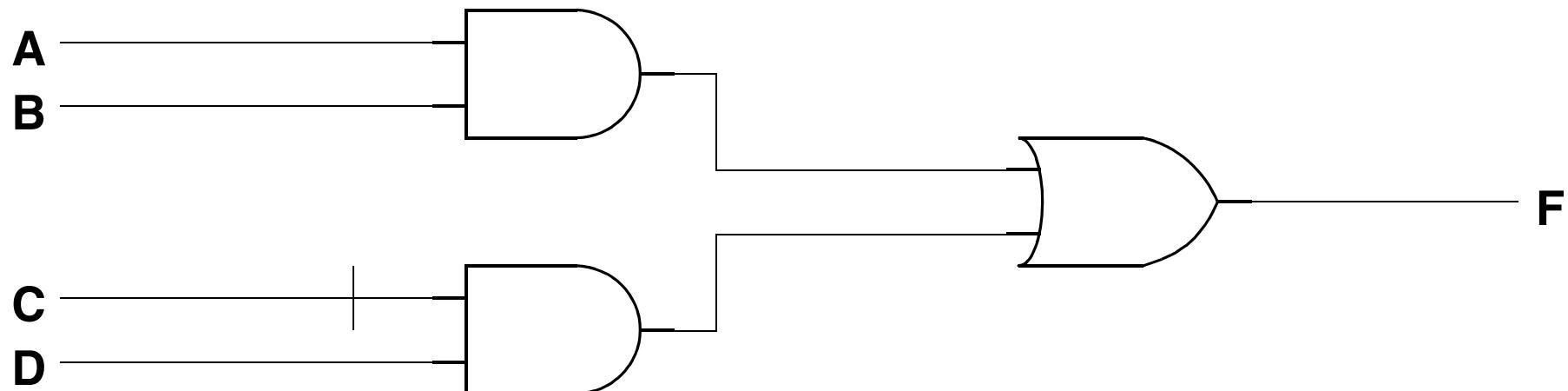
A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

- The procedure for performing mixed logic conversions is as follows:
  - Draw the logic network for the given Boolean equation.
    - Use only **AND** and **OR** gates.
    - Replace all complements with a bar (no bubbles or inverters yet!)
  - Once the initial Boolean equation is drawn with **AND** gates, **OR** gates and bars, the **self-documenting redesign** begins:
    - **Add complement bubbles** and **NOT** gates within the network to appropriately convert logic gates to desired gate sets.
    - The rules in adding complement bubbles and **NOT** gates
      - **All bubbles must cancel each other out**
      - **Exactly one and only one bubble needed on each bar**

- Implement the following Boolean function using NAND gates and then also using NOR gates.

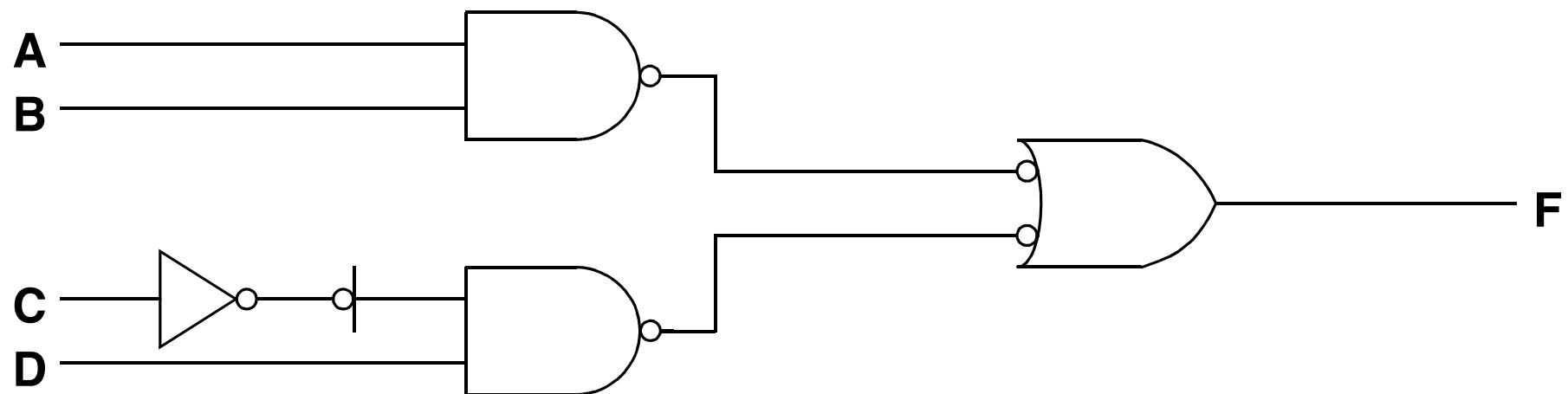
$$F = AB + \bar{C}D$$

- Solution: Start by drawing the logic network for the Boolean function with the complements as bars.



- This completes the information needed to get back original equation.

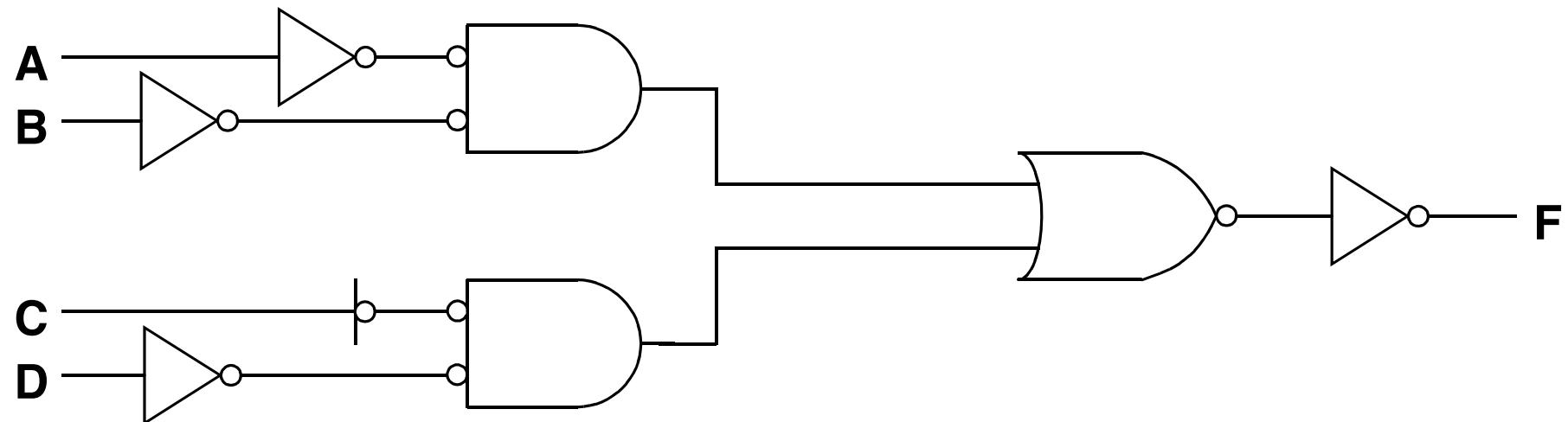
continued... using **NAND** gates



- This logic network now only uses **NAND** gates and inverters.
- This results in the following Boolean function, as obtained previously

$$F(A, B, C, D, E) = \overline{\overline{ABC}}\overline{D}$$

continued... using **NOR** gates



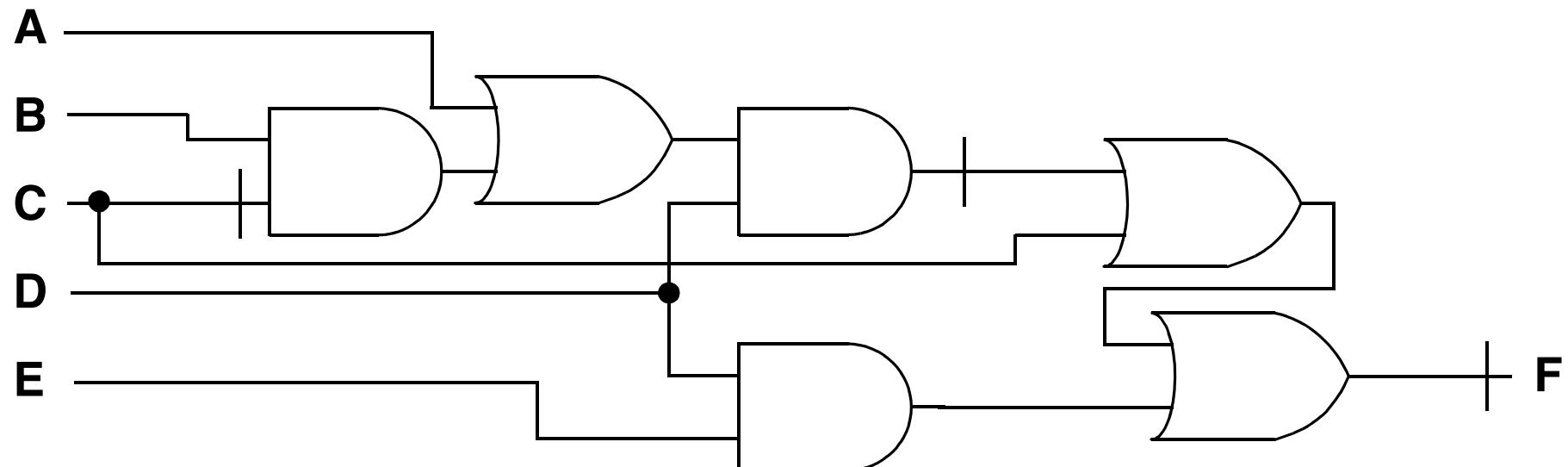
- This logic network now only uses **NOR** gates and inverters.
- This results in the following Boolean function, as obtained previously

$$F(A, B, C, D, E) = \overline{\overline{\overline{A}}} + \overline{\overline{\overline{B}}} + \overline{\overline{\overline{C}}} + \overline{\overline{\overline{D}}}$$

- Implement the following Boolean function using **NAND** gates

$$F = \overline{\overline{(A + B\bar{C})D} + C + DE}$$

- Solution: Start by drawing the logic network for the Boolean function with the complements as bars.

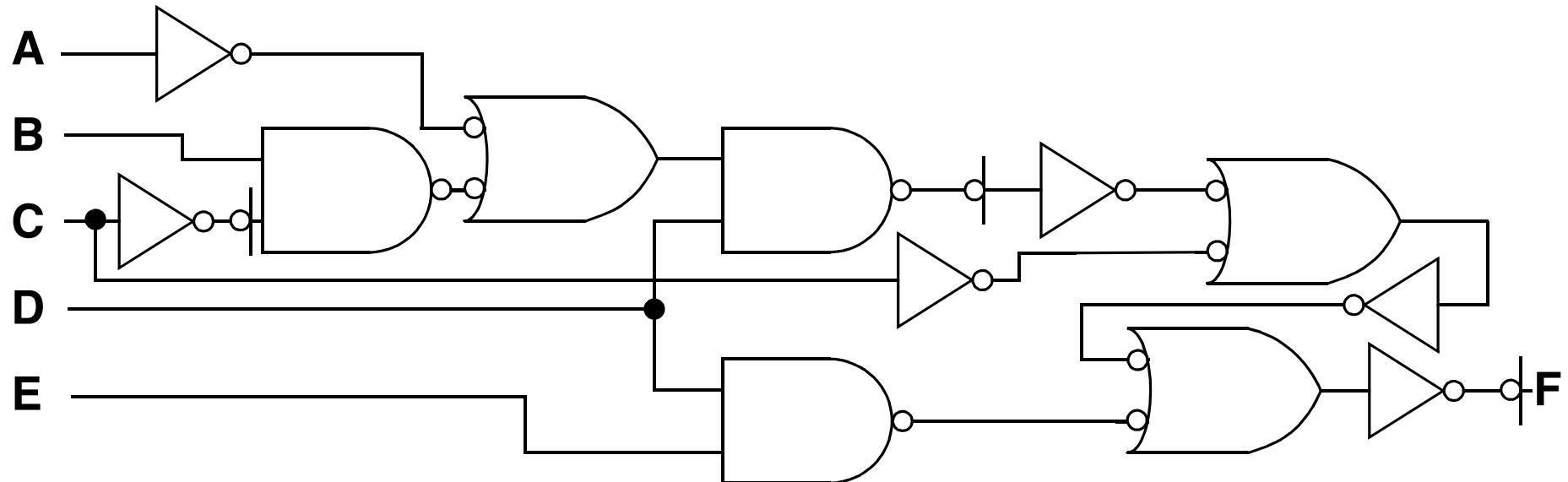


# MIXED LOGIC

## EXAMPLE #2 (2)

- MIXED LOGIC
  - PROCEDURE
  - EXAMPLE #1
  - EXAMPLE #2

continued...



- This logic network now only uses **NAND** gates and inverters.
- This results in the following Boolean function, as obtained previously

$$\overline{F(A, B, C, D, E)} = \overline{\overline{ABC}DC\overline{CDE}} = \overline{\overline{ABC}DC\overline{CDE}}$$

**INTRO. TO COMP. ENG.  
CHAPTER V-1**

**NUMBERS & ARITHMETIC**

**•CHAPTER V**

# **CHAPTER V**

## **NUMBER SYSTEMS AND ARITHMETIC**

- Decimal number expansion

$$73625_{10} = (7 \times 10^4) + (3 \times 10^3) + (6 \times 10^2) + (2 \times 10^1) + (5 \times 10^0)$$

- Binary number representation

$$10110_2 = (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 22_{10}$$

- Hexadecimal number representation

$$\begin{aligned}3E4B8_{16} &= (3 \times 16^4) + (14 \times 16^3) + (4 \times 16^2) + (11 \times 16^1) + (8 \times 16^0) \\&= 255160_{10}\end{aligned}$$

### BINARY <-> HEXADECIMAL

$$0000_2 = 0_{16}$$

$$1000_2 = 8_{16}$$

$$0001_2 = 1_{16}$$

$$1001_2 = 9_{16}$$

$$0010_2 = 2_{16}$$

$$1010_2 = 10 \text{ (A}_{16}\text{)}$$

$$0011_2 = 3_{16}$$

$$1011_2 = 11 \text{ (B}_{16}\text{)}$$

$$0100_2 = 4_{16}$$

$$1100_2 = 12 \text{ (C}_{16}\text{)}$$

$$0101_2 = 5_{16}$$

$$1101_2 = 13 \text{ (D}_{16}\text{)}$$

$$0110_2 = 6_{16}$$

$$1110_2 = 14 \text{ (E}_{16}\text{)}$$

$$0111_2 = 7_{16}$$

$$1111_2 = 15 \text{ (F}_{16}\text{)}$$

### BINARY -> HEXADECIMAL

Group binary by 4 bits from radix point

Examples:

$$0111\ 1011_2 = 7B_{16}$$

$\brace{7} \quad \brace{B}$

$$10\ 1010\ 0110.1100\ 01_2 = 2A6.C4_{16}$$

$\brace{2} \quad \brace{A} \quad \brace{6} \quad \brace{C} \quad \brace{4}$

- Perform radix-2 expansion
  - Multiply each bit in the binary number by 2 to the power of its place. Then sum all of the values to get the decimal value.

Examples:

$$10111_2 = (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 23_{10}$$

$$\begin{aligned}10110.0011_2 &= (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \\&\quad + (0 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4}) \\&= 22.1875_{10}\end{aligned}$$

# NUMBER SYSTEMS

## DECIMAL -> BINARY

- Integer part:

- Modulo division of decimal integer by 2 to get each bit, starting with LSB.

- Fraction part:

- Multiplication decimal fraction by 2 and collect resulting integers, starting with MSB.

Example: Convert  $41.828125_{10}$

$$41 \bmod 2 = 1 \quad \text{LSB}$$

$$20 \bmod 2 = 0$$

$$10 \bmod 2 = 0$$

$$5 \bmod 2 = 1$$

$$2 \bmod 2 = 0$$

$$1 \bmod 2 = 1 \quad \text{MSB}$$

$$0.828125 \times 2 = 1.65625 \quad \text{MSB}$$

$$0.65625 \times 2 = 1.3125$$

$$0.3125 \times 2 = 0.625$$

$$0.625 \times 2 = 1.25$$

$$0.25 \times 2 = 0.5$$

$$0.5 \times 2 = 1.0$$

LSB

Therefore  $41.828125_{10} = 101001.110101_2$

- The range for an  $n$ -bit radix- $r$  unsigned integer is

$$[0, r_{10}^n - 1]$$

- Example: For a 16-bit binary unsigned integer, the range is

$$[0, 2^{16} - 1] = [0, 65535]$$

which has a binary representation of

$$0000\ 0000\ 0000\ 0000 = 0$$

$$0000\ 0000\ 0000\ 0001 = 1$$

$$0000\ 0000\ 0000\ 0010 = 2$$

...

$$1111\ 1111\ 1111\ 1110 = 65534$$

$$1111\ 1111\ 1111\ 1111 = 65535$$

- The range for an  $n$ -bit radix- $r$  signed integer is

$$[-r_{10}^{n-1}, r_{10}^{n-1} - 1]$$

- The most-significant bit is used as a sign bit, where **0** indicates a positive integer and **1** indicates a negative integer.

Example: For a 16-bit binary signed integer, the range is

$$[-2^{16-1}, 2^{16-1} - 1] = [-32768, 32767]$$

- There are a number of different representations for signed integers, each which has its own advantage
  - Signed-magnitude representation:
    - **1010 0001 0110 1111**
  - Signed-1's complement representation:
    - **1101 1110 1001 0000**
  - Signed-2's complement representation:
    - **1101 1110 1001 0001**
- The above examples are all the same number,  $-8559_{10}$ .

- The **signed-magnitude** binary integer representation is just like the **unsigned representation** with the addition of a **sign bit**.
  - For instance, using 8-bits, the number  $-6_{10}$  can be represented as the 7-bit magnitude of  $6_{10}$  using

000 0110

and then the sign bit appended to the MSB to form

1000 0110

- The **1's complement** (diminished radix complement) binary integer representation for an  $n$ -bit integer is defined as

$$(2^n_{10} - 1_{10}) - \text{number}_{10}$$

- In essence, this takes the positive version of the number and flips all of the bits.
  - For instance, using 8-bits, the number  $-6_{10}$  can be represented as the 8-bit positive number  $6_{10}$  using

**0000 0110**

and then each of the bits flipped to form the 1's complement

**1111 1001**

# BINARY NUMBERS

## 2'S COMPLEMENT

- The **2's complement** (radix complement) binary integer representation for an  $n$ -bit integer is defined as

$$2^n_{10} - \text{number}_{10}$$

- In essence, this takes the 1's complement and adds one.
  - For instance, using 8-bits, the number  $-6_{10}$  can be represented as the 8-bit positive number  $6_{10}$  using

**0000 0110**

and then each of the bits flipped to form the 1's complement

**1111 1001**

and then add 1 to form the 2's complement

**1111 1010**

# BINARY NUMBERS

## SIGNED EXAMPLES

- Below are some examples for the signed binary numbers using 6 bits.

Decimal	Signed-magnitude	1's complement	2's complement
0	00 0000	00 0000	00 0000
1	00 0001	00 0001	00 0001
-1	10 0001	11 1110	11 1111
5	00 0101	00 0101	00 0101
-5	10 0101	11 1010	11 1011
12	00 1100	00 1100	00 1100
-12	10 1100	11 0011	11 0100
15	00 1111	00 1111	00 1111
-15	10 1111	11 0000	11 0001
16	01 0000	01 0000	01 0000
-16	11 0000	10 1111	11 0000

- Notice that all representations are the **same for positive numbers!!!!**

- Unsigned binary addition follows the standard rules of addition.
  - Examples

**1111 0100 Carries**

$$\begin{array}{r} 0011\ 1011 \\ +\ 0111\ 1010 \\ \hline 1011\ 0101 \end{array}$$

**0000 0010 Carries**

$$\begin{array}{r} 1011\ 1001 \\ +\ 0100\ 0101 \\ \hline 1111\ 1110 \end{array}$$

**1111 0000 Carries**

$$\begin{array}{r} 1111\ 1001 \\ +\ 0100\ 1000 \\ \hline 1\ 0100\ 0000 \end{array}$$

**1110 0000 0000.0001 Carries**

$$\begin{array}{r} 0101\ 1000\ 1001.1001 \\ +\ 0011\ 0011\ 0100.01 \\ \hline 1000\ 1011\ 1101.1101 \end{array}$$

# BINARY ARITHMETIC

## UNSIGNED SUBTRACTION

- Unsigned binary subtraction follows the standard rules.
  - Examples

**0000 0000 Borrows**

$$\begin{array}{r} 1111\ 1001 \\ - 0100\ 1000 \\ \hline 1011\ 0001 \end{array}$$

**1000 1000 Borrows**

$$\begin{array}{r} 1011\ 1001 \\ - 0100\ 0101 \\ \hline 0111\ 0100 \end{array}$$

**1000 0000 Borrows**

$$\begin{array}{r} 0011\ 1011 \\ - 0111\ 1010 \\ \hline 1100\ 0001 \end{array}$$

**0100 1110 1000.1000 Borrows**

$$\begin{array}{r} 0101\ 1000\ 1001.1001 \\ - 0011\ 0011\ 0100.01 \\ \hline 0010\ 0101\ 0101.0101 \end{array}$$

- **Signed-magnitude**
  - Add magnitudes if signs are the same, give result the sign
  - Subtract magnitudes if signs are different. Absence or presence of an end borrow determines the resulting sign compared to the augend. If negative, then a 2's complement correction must be taken.
- **2's complement**
  - Add the numbers using normal addition rules. Carry out bit is discarded.
- **1's complement**
  - Easiest to convert to 2's complement, perform the addition, and then convert back to 1's complement. This is done as follows:
    - Add 1 to each integer, add the integers, subtract 1 from the result

# BINARY ARITHMETIC

## SIGNED SUBTRACTION

- BINARY ARITHMETIC
  - UNSIGNED ADDITION
  - UNSIGNED SUBTRACT.
  - SIGNED ADDITION

- Typically want to do addition or subtraction of **A** and **B** as follows.

$$\text{SUM} = A + B$$

$$\text{DIFFERENCE} = A - B$$

- If we use **2's complement**, we can make life easy on us since addition and subtraction are done in the same manner: **with addition only!!!**
- A subtraction can be re-represented as follows.

$$\text{SUM} = A + (-B)$$

- Or in general any two numbers can be added as follows.

$$\text{SUM} = (\pm A) + (\pm B)$$

# BINARY ARITHMETIC

## SIGNED MATH EXAMPLE

- Subtraction of signed numbers can best be done with 2's complement.
  - Performed by taking the 2's complement of the subtrahend and then performing addition (including the sign bit).
    - Example:

$$\begin{array}{r} 59 \\ - 122 \\ \hline \end{array} = \begin{array}{r} 0011\ 1011 \\ - 0111\ 1010 \\ \hline \end{array} = \begin{array}{r} 0011\ 1011 \\ + 1000\ 0110 \\ \hline 1100\ 0001 \end{array} = -(0011\ 1111) = -63$$

2's complement

discard carry out

0 0111 1100 Carries

standard addition

2's complement

**INTRO. TO COMP. ENG.  
CHAPTER VI-1**

**COMBINATIONAL LOGIC**

**•CHAPTER VI**

# **CHAPTER VI**

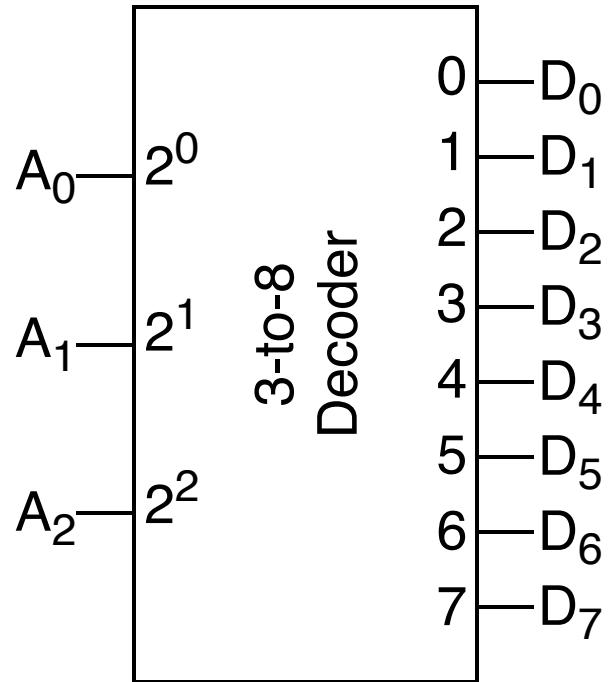
## **COMBINATIONAL LOGIC BUILDING BLOCKS**

- Combinational logic
  - Output at any time is determined completely by the current input.
    - We will later consider circuits where the output is determined by the input and the current state (memory) of the system.
  - In this chapter we will consider some useful building blocks that can be pieced together and used in larger designs. This will include:
    - Multiplexers (selectors) and demultiplexers (distributors)
    - Encoders, priority encoders, decoders
    - Adders (full and half)
    - Parity generators and parity checkers
    - Shifters and rotators
    - Comparators

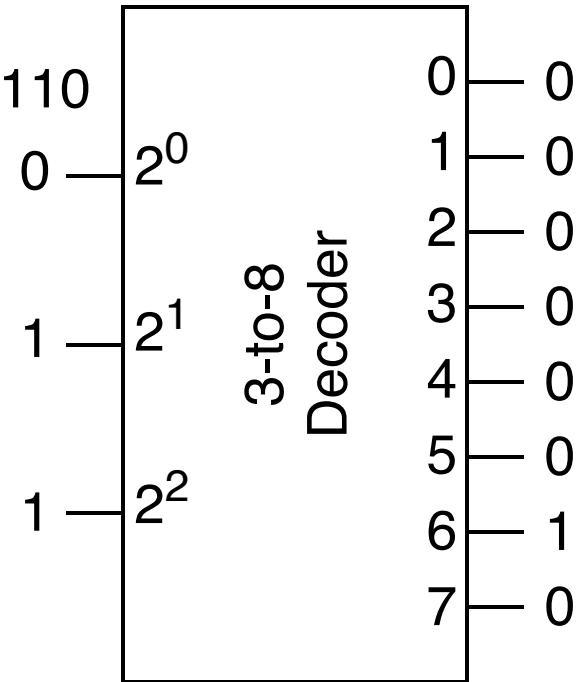
# DECODERS

## BASIC DECODER

- Standard decoder is an  $n$ -to- $m$ -line decoder, where  $m \leq 2^n$ .
  - Example: 3-to-8-line decoder



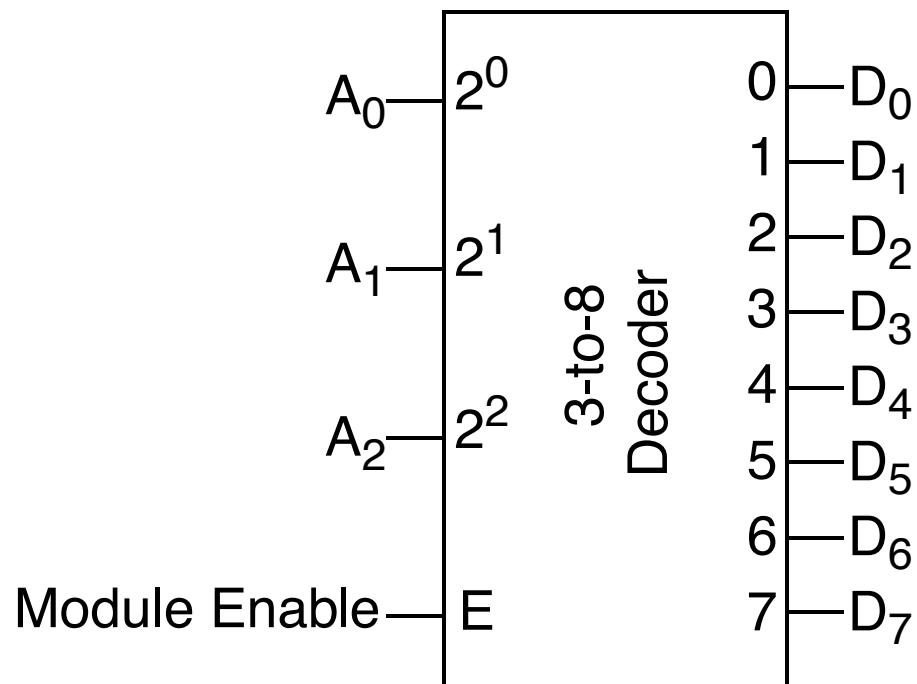
Example:  
Input of 110



- All outputs  $D_m$  are low except for the one corresponding to the binary value of the input  $A_n \dots A_1 A_0$ .

- Often, combinational logic building blocks will also have an enable line that turns on outputs or leaves them off.

3-to-8 Decoder  
with Enable



# DECODERS

## TRUTH TABLES

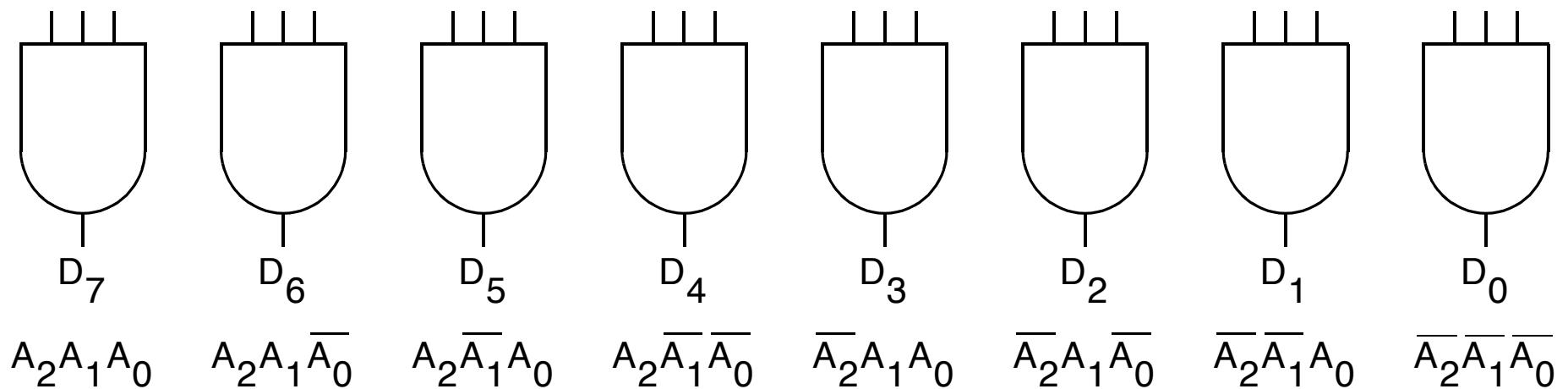
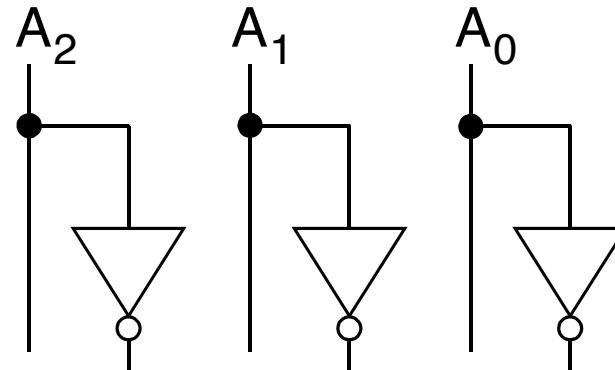
- Truth table for a **3-to-8-line decoder**:

Inputs				Outputs							
A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	E	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
X	X	X	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0
0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	0
1	0	1	1	0	0	1	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

# DECODERS

## IMPLEMENTATION

- How can a decoder be implemented? Fill in the circuit!

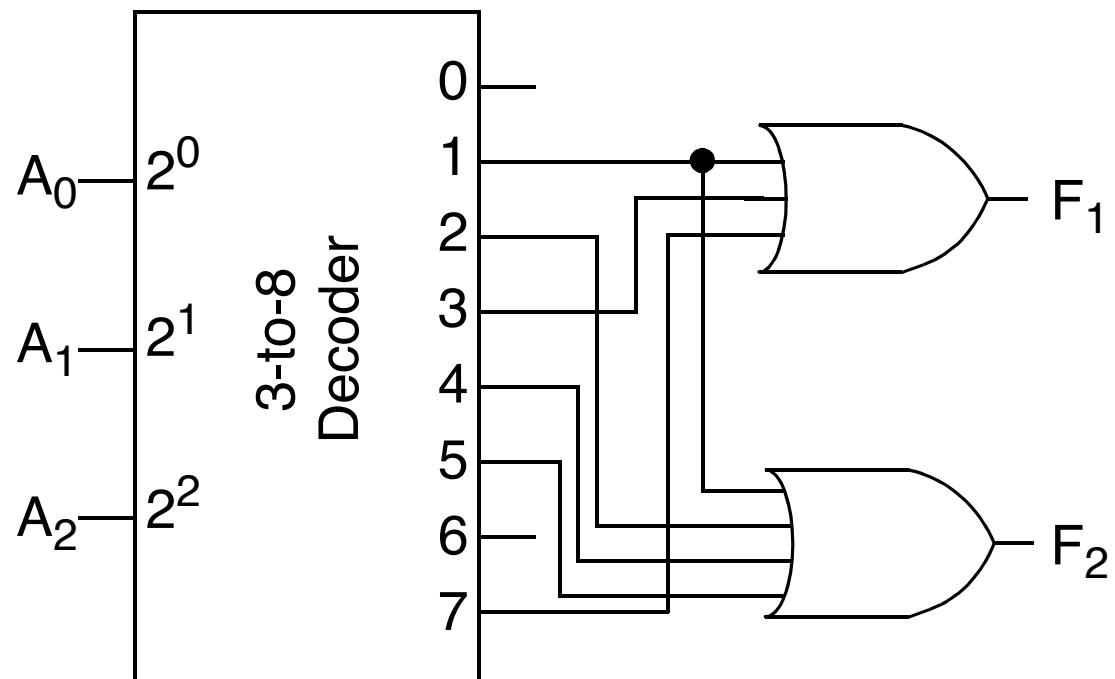


# DECODERS

## DESIGNING WITH DECODERS

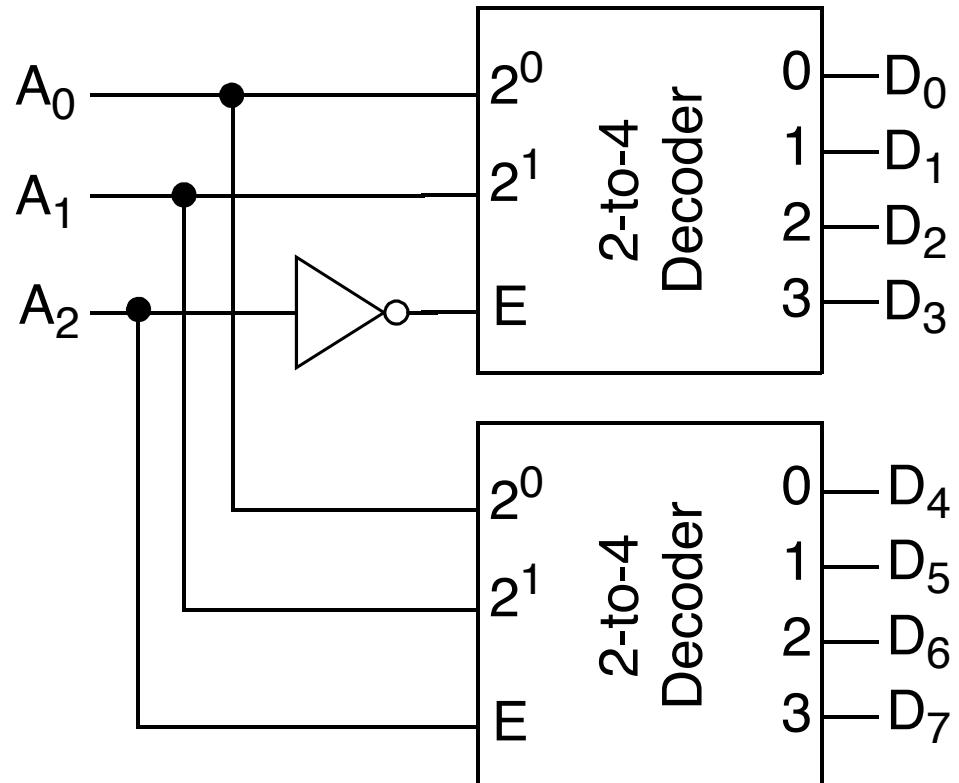
- Any Boolean function can be implemented using a **decoder** and **OR** gates by ORing together the function's **minterms**.

Inputs			Outputs	
A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0



- We can also use multiple decoders to form a larger decoder.

3-to-8 Decoder Implemented  
with two 2-to-4 Decoders



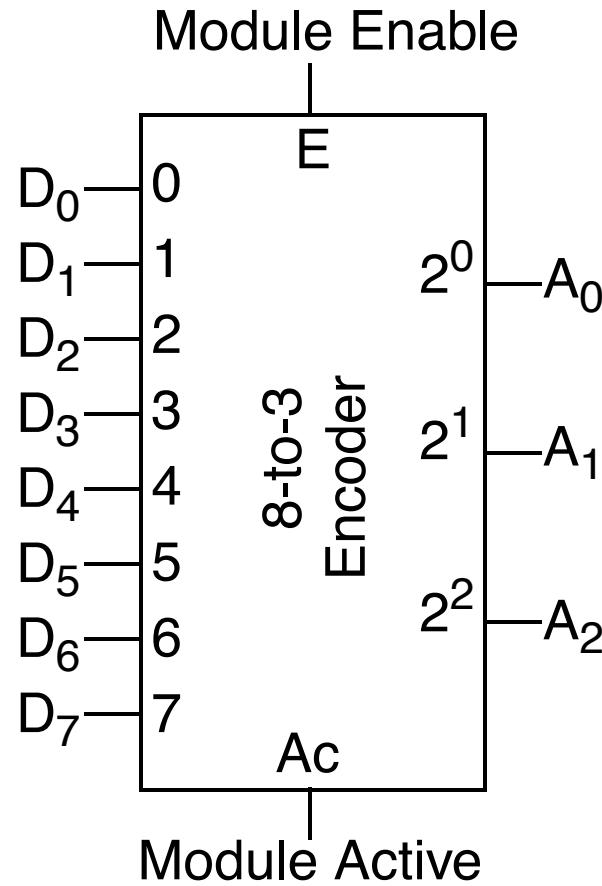
$A_2$  used with enable input to control which decoder will output the 1.

$A_1$  and  $A_0$  used to select which output on specific decoder will output 1.

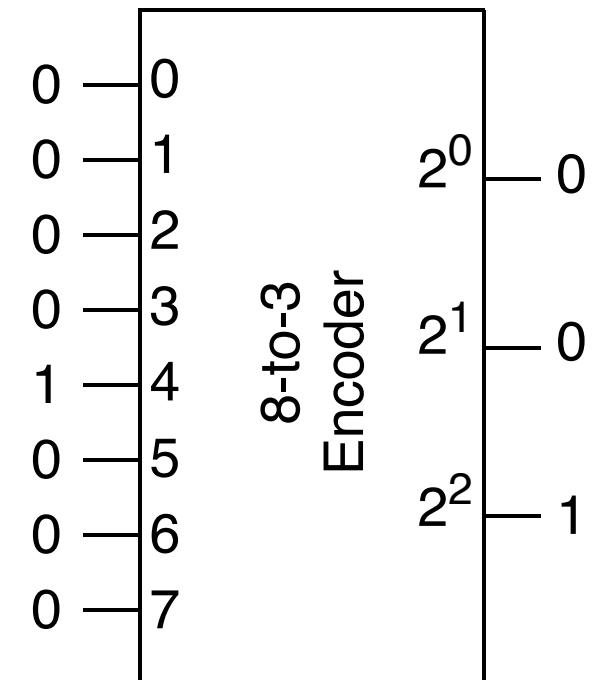
# ENCODERS

## BASIC ENCODER

- Standard binary encoder is an  $m$ -to- $n$ -line encoder, where  $m \leq 2^n$ .
  - Example: 8-to-3-line encoder



Example:  
Input of 00010000



# ENCODERS

## ENCODER TRUTH TABLE

- Truth table for an **8-to-3-line encoder**:

**Inputs**

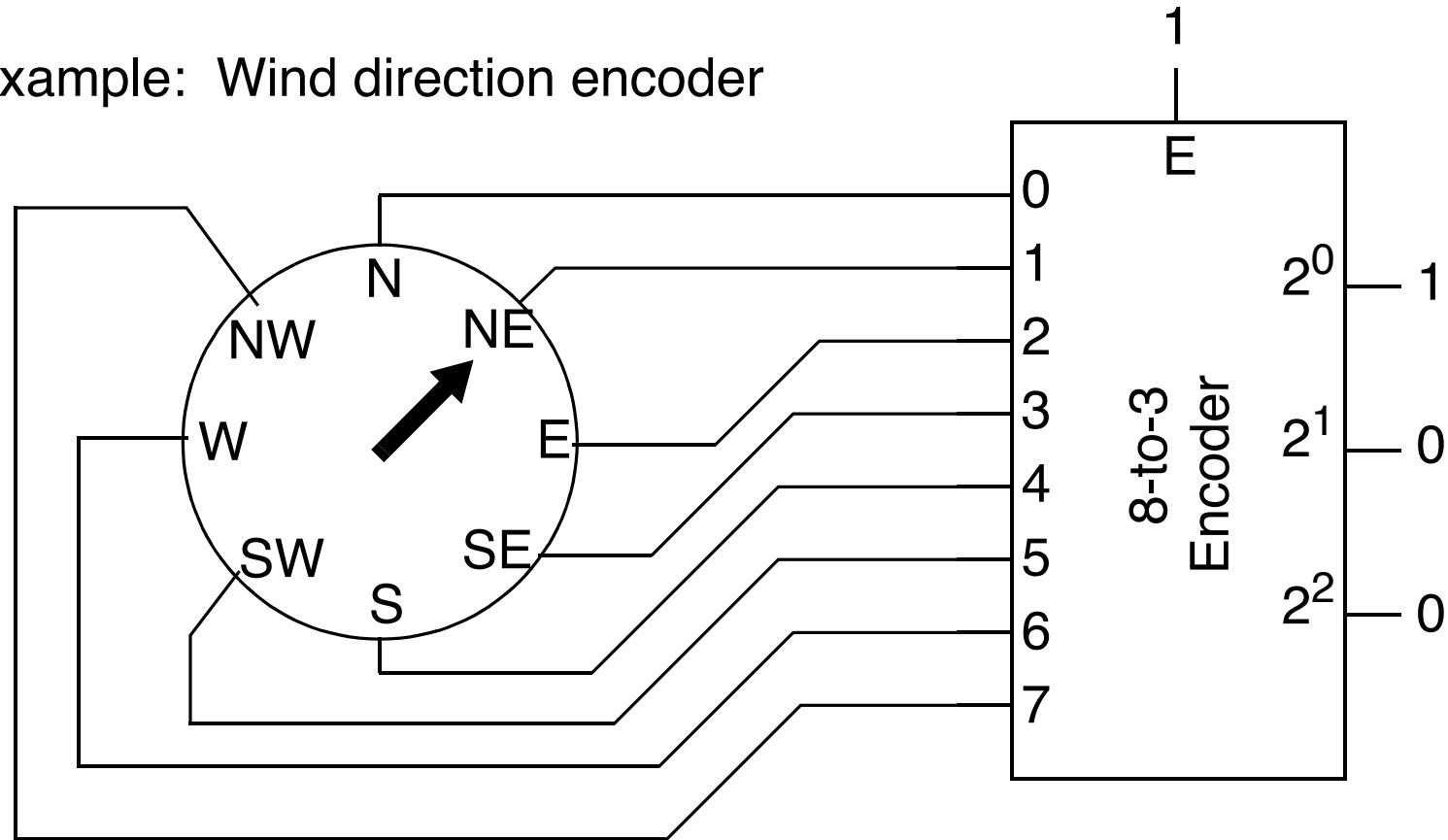
**Outputs**

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

- Assumed that only one input is 1. What happens if more than one is 1?

- Encoders are useful when the occurrence of one of several disjoint events needs to be represented by an integer identifying the event.

Example: Wind direction encoder



pp. 253-254 of Ercegovac, Lang and Moreno, "Introduction to Digital Systems", 1999.

# ENCODERS

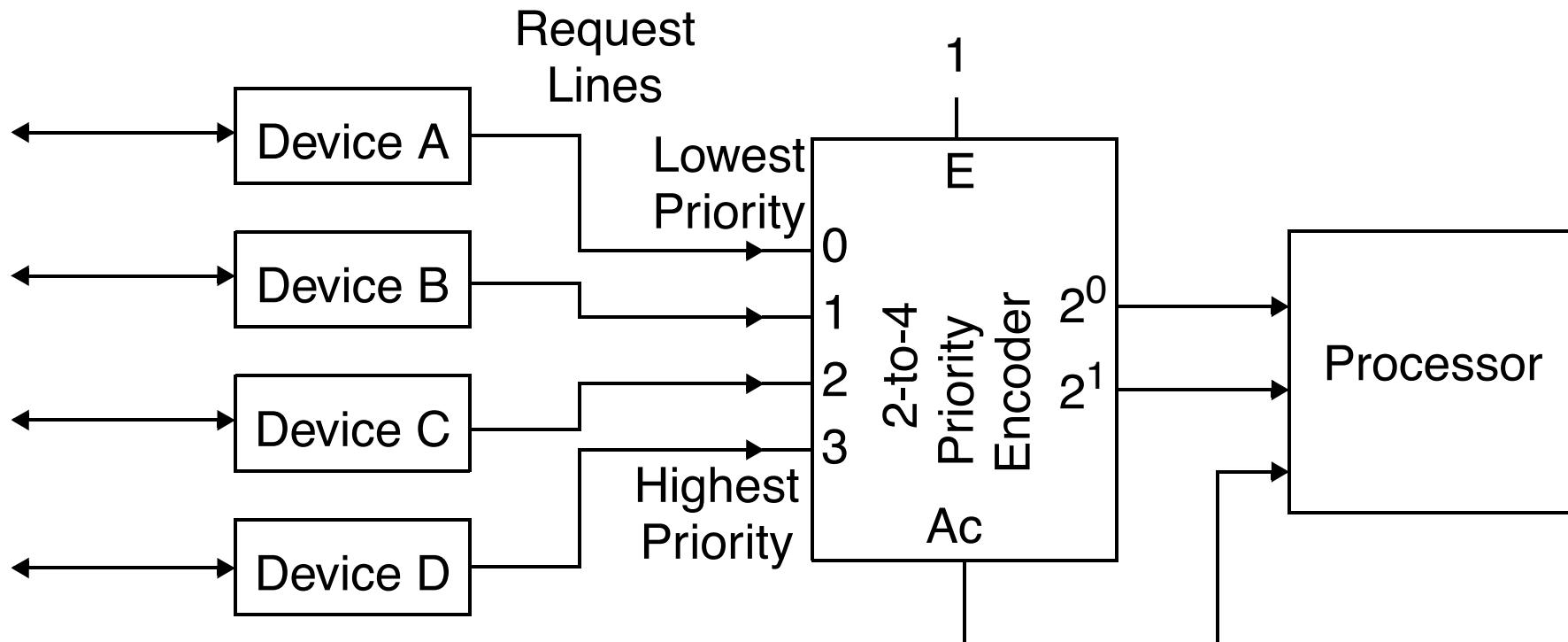
## PRIORITY ENCODERS

- A priority encoder takes the input of 1 with the highest index and translates that index to the output.

Inputs								Outputs		
D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

- Priority encoders are useful when inputs have a predefined priority and we wish to select the input with the highest priority.

Example: Resolving interrupt requests



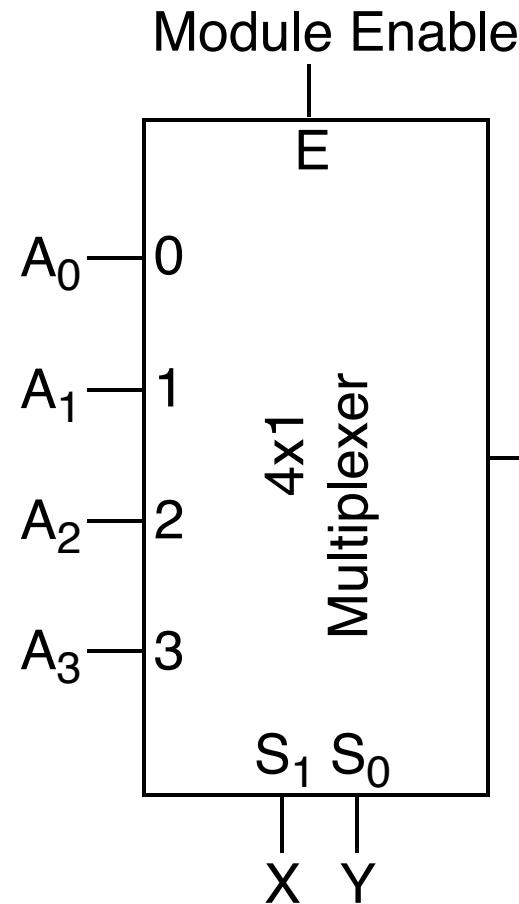
pp. 253-256 of Ercegovac, Lang and Moreno, "Introduction to Digital Systems", 1999.

# MULTIPLEXERS

## BASIC MULTIPLEXER (MUX)

- ENCODERS
  - DESIGN W/ ENCODERS
  - PRIORITY ENCODERS
  - DESIGN W/ P-ENCODERS

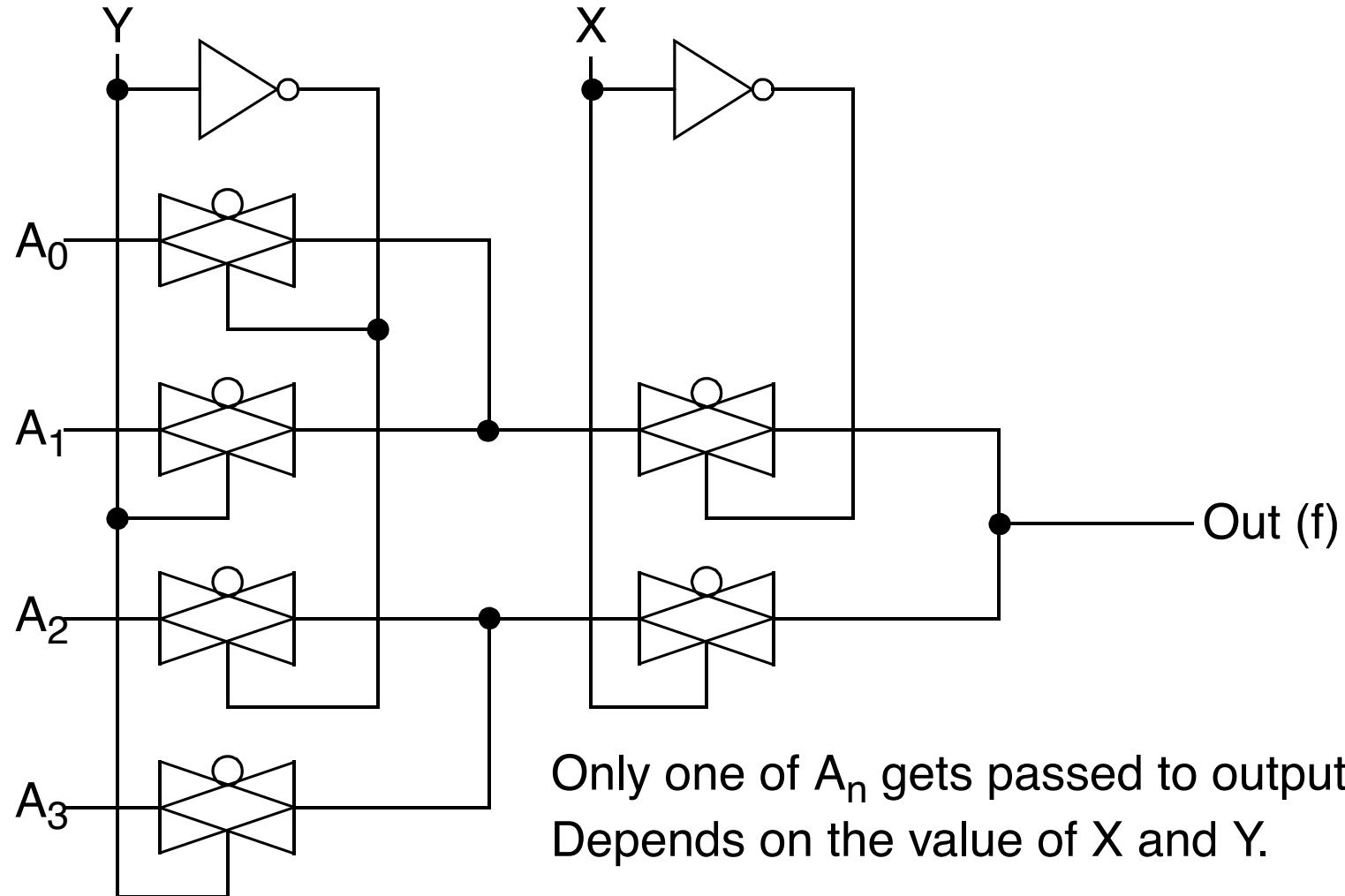
- Selects one of many inputs to be directed to an output.



	Inputs						Output
	X	Y	$A_3$	$A_2$	$A_1$	$A_0$	F
	0	0	X	X	X	0	$A_0=0$
	0	1	X	X	0	X	$A_1=0$
	1	0	X	0	X	X	$A_2=0$
	1	1	0	X	X	X	$A_3=0$
	0	0	X	X	X	1	$A_0=1$
	0	1	X	X	1	X	$A_1=1$
	1	0	X	1	X	X	$A_2=1$
	1	1	1	X	X	X	$A_3=1$

# MULTIPLEXERS USING PASS GATES

- The **4x1 mux** can be implemented with **pass gates** as follows.

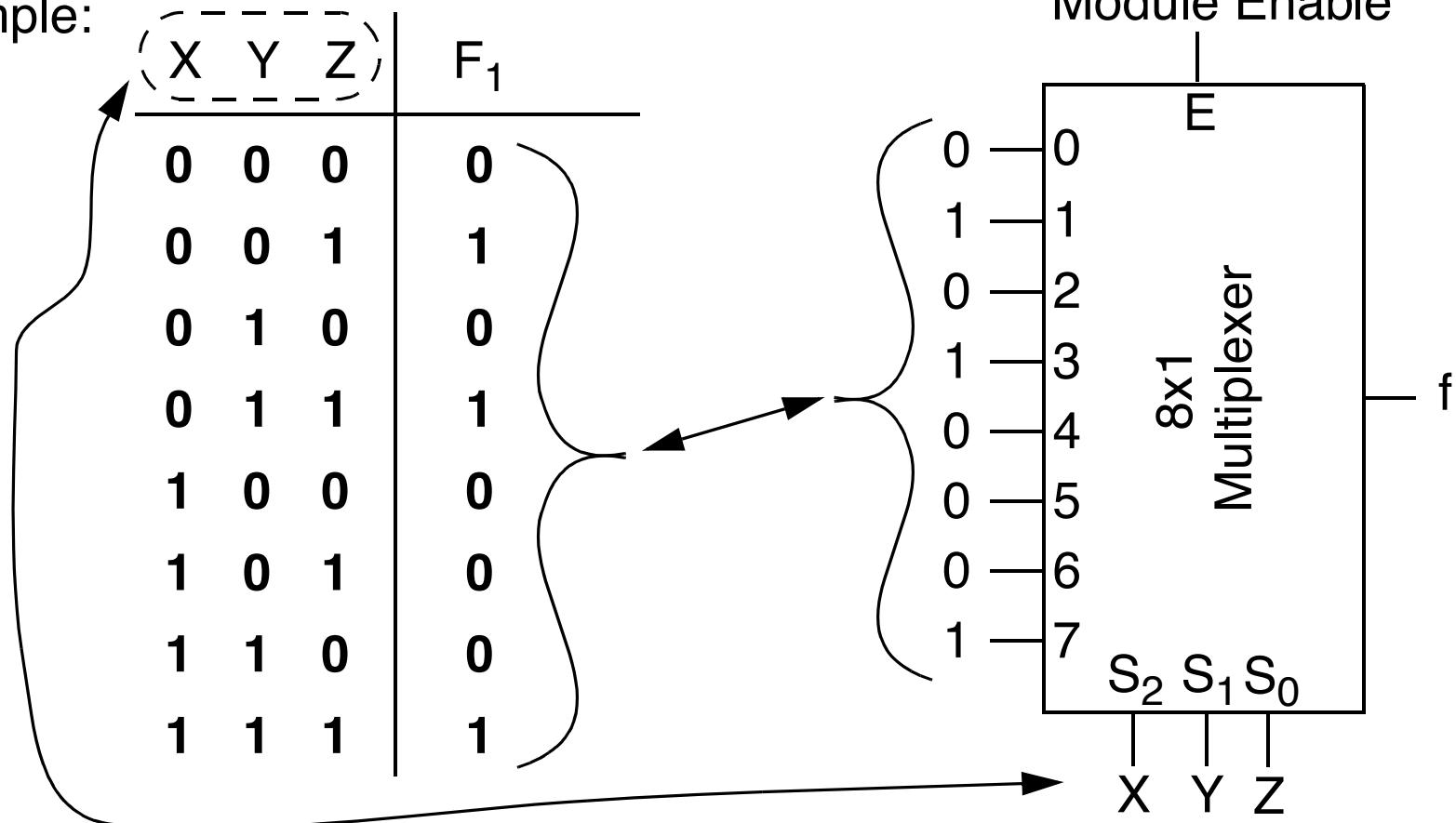


# MULTIPLEXERS

## DESIGN WITH MULTIPLEXERS

- Any Boolean function can be implemented by setting the inputs corresponding to the function and the selectors as the variables.

Example:

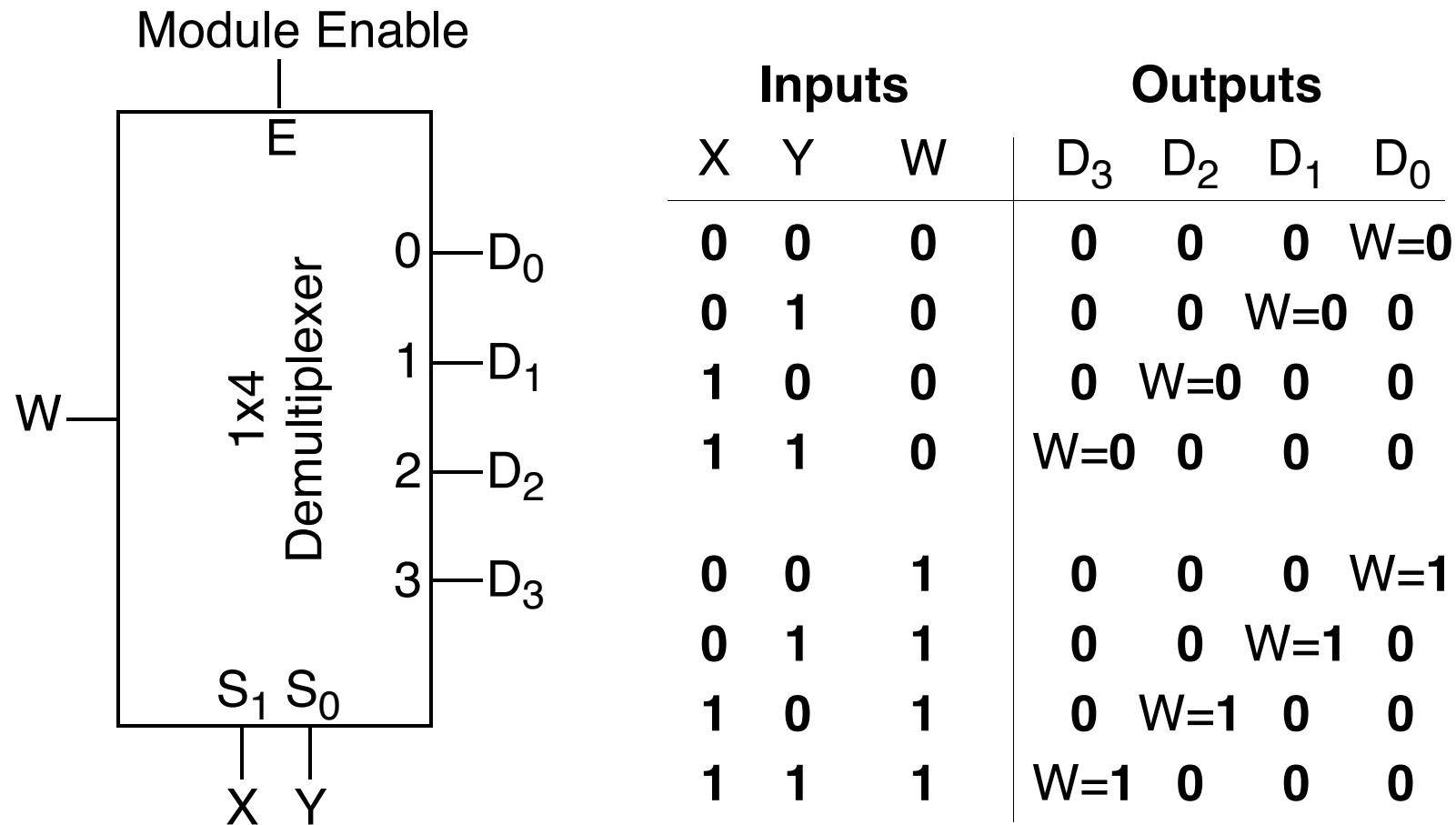


# DEMULITPLEXERS

## BASIC DEMULITPLEXER

- MUXPLEXERS
  - BASIC MUXPLEXER
  - USING PASS GATES
  - DESIGN W/ MUXPLEX.

- Takes one input and selects one of many outputs to direct the input.

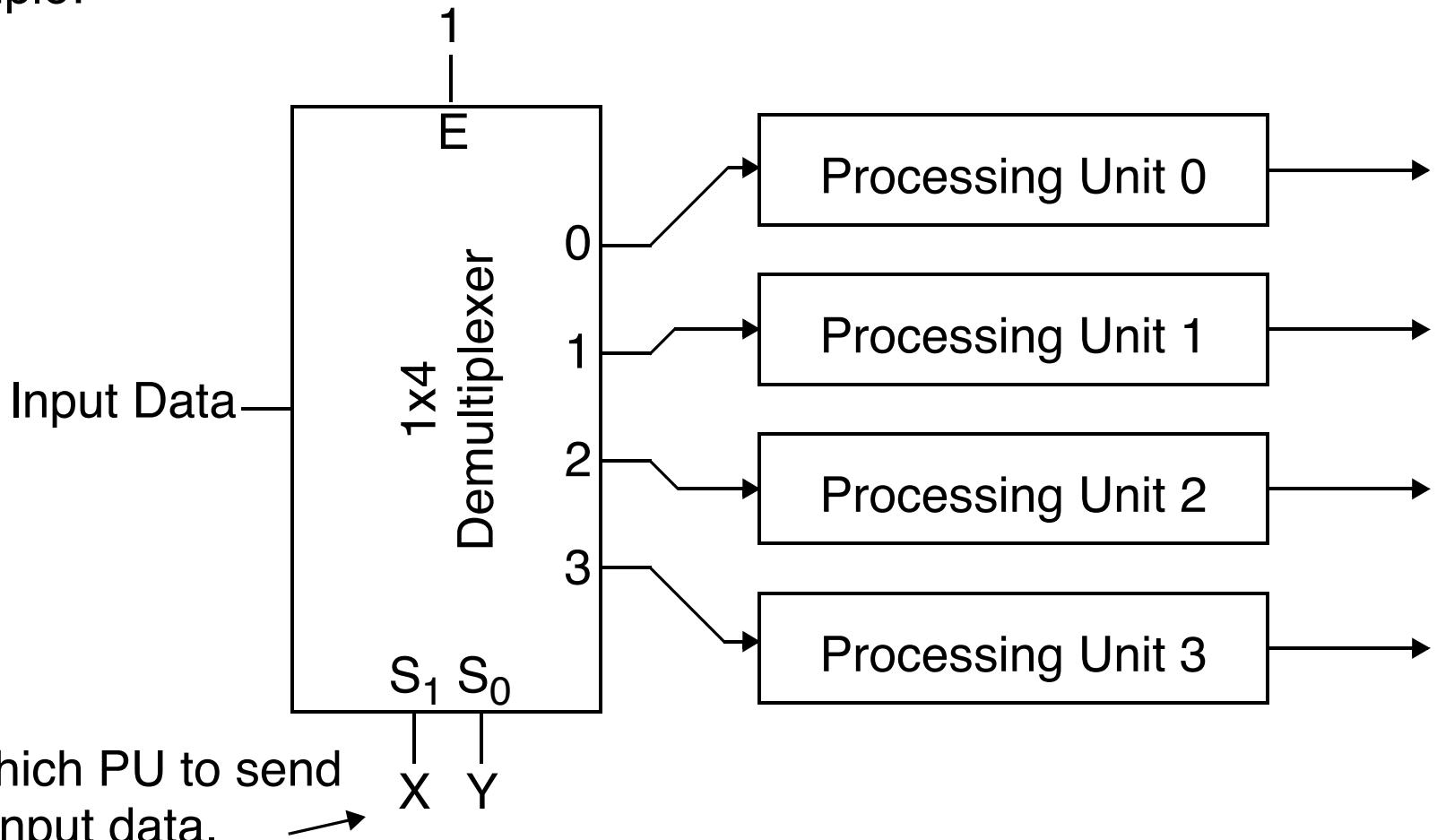


# DEMULITPLEXERS

## DESIGN W/ DEMULITPLEXERS

- A demultiplexer is useful for routing an input to a desired location.

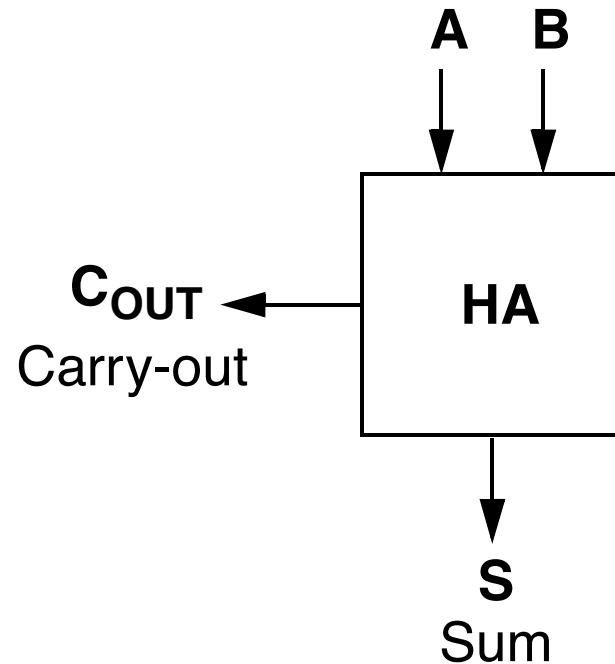
Example:



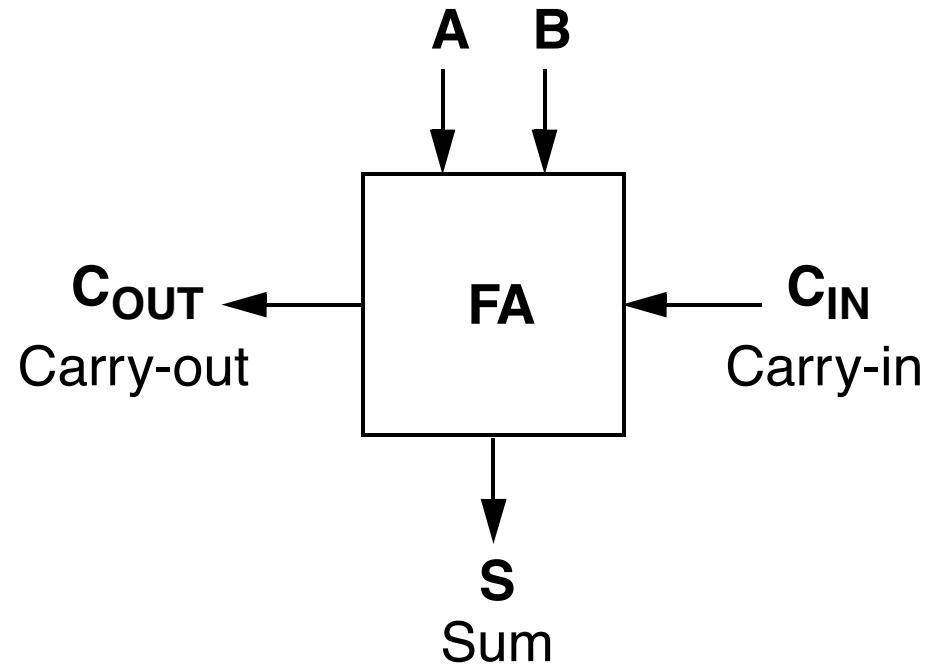
# ADDERS

## HALF- AND FULL-ADDERS

- Two basic building blocks for arithmetic are half- and full-adders as depicted by the block diagrams below.



Half-adder



Full-adder

# ADDERS

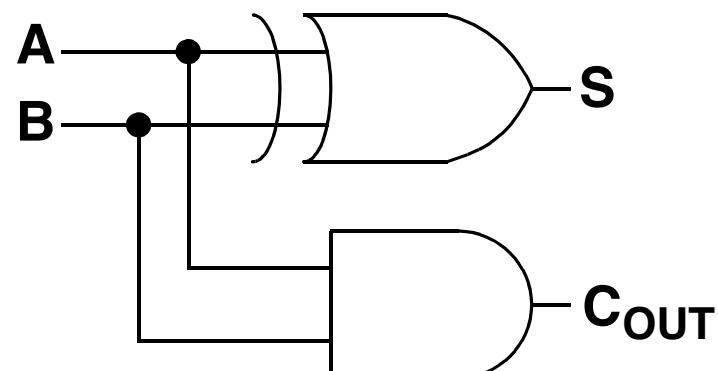
## HALF-ADDER (HA)

- First of all, how do we add?
- 2's complement arithmetic allows us to add numbers normally.

Inputs		Sum	Carry-out
A	B	S	C <sub>OUT</sub>
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$S = \bar{A}B + A\bar{B} = A \oplus B$$

$$C_{OUT} = AB$$



# ADDERS

## FULL-ADDER (FA) (1)

- Half-adder missed a possible carry-in. A full-adder (FA) includes this additional carry-in.

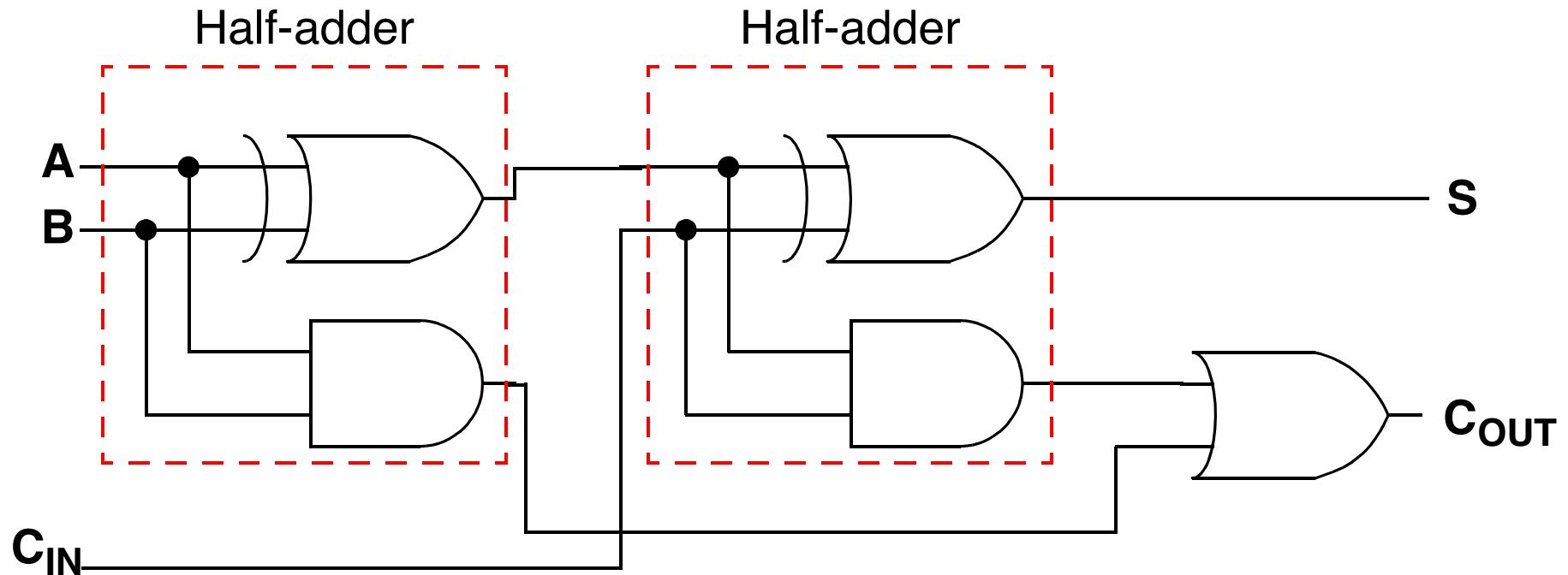
Inputs			Carry-in	Sum	Carry-out
A	B	$C_{IN}$		S	$C_{OUT}$
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

$$S = (A \oplus B) + C_{IN}$$

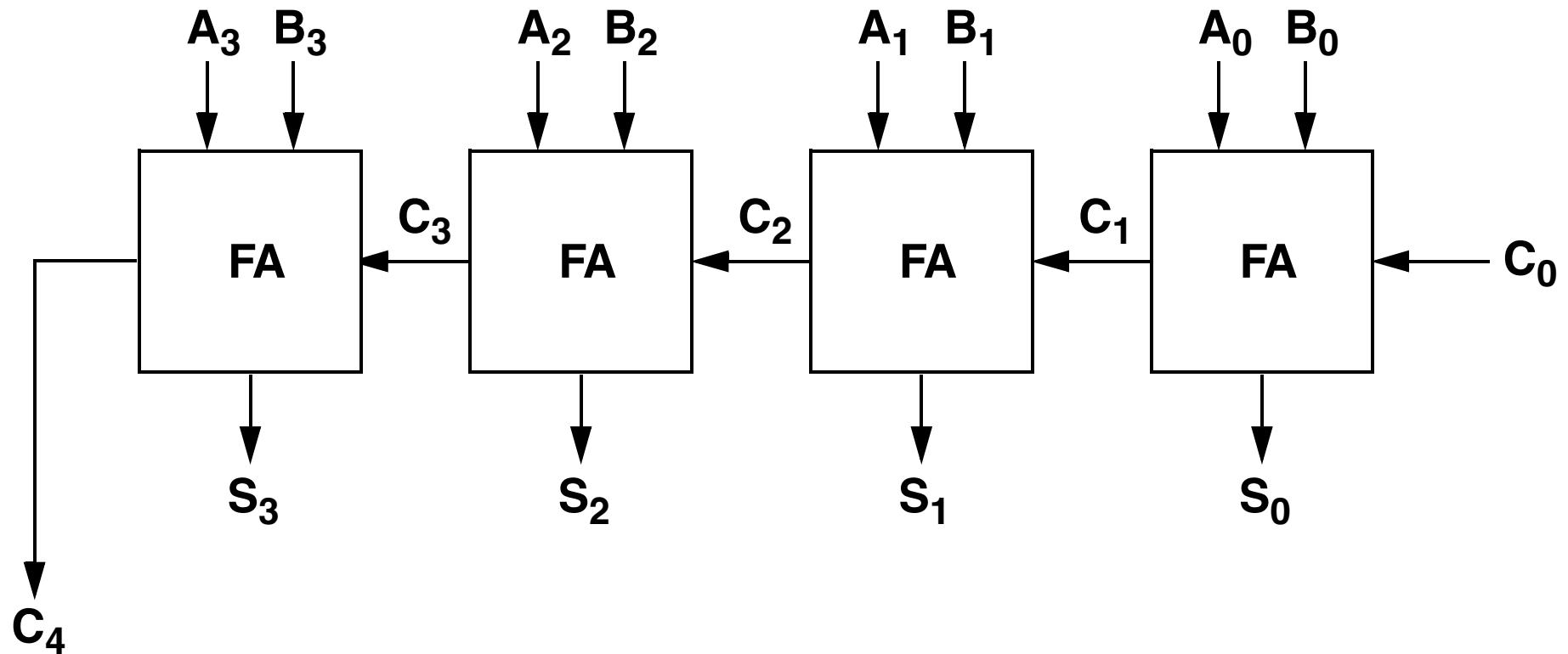
$$C_{OUT} = AB + C_{IN}(A \oplus B)$$

$$S = (A \oplus B) \oplus C_{IN}$$

$$C_{OUT} = AB + C_{IN}(A \oplus B)$$



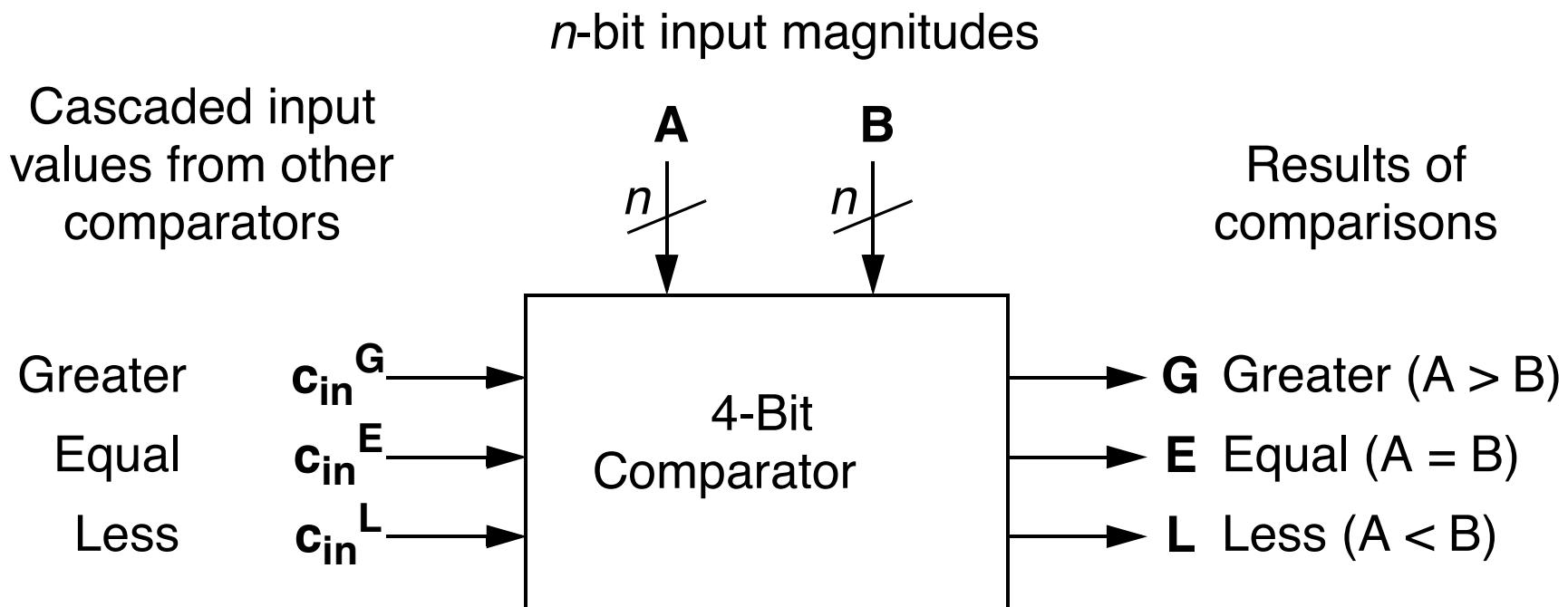
- A 4-bit binary adder can be formed with four full-adders as follows.



# COMPARATORS

## MAGNITUDE COMPARATOR

- Given two  $n$ -bit magnitudes, A and B, a comparator indicates whether
  - $A = B$ ,  $A > B$ , or  $A < B$



- The approach is to use the XNOR function (equivalence) on each of the  $n$ -bits as follows

$$x_i = \mathbf{A}_i \mathbf{B}_i + \overline{\mathbf{A}}_i \overline{\mathbf{B}}_i = \overline{\mathbf{A}_i \oplus \mathbf{B}_i}$$

- The Boolean functions for a 4-bit magnitude comparator is as follows
  - $(\mathbf{A} = \mathbf{B}) = x_3 x_2 x_1 x_0$
  - $(\mathbf{A} > \mathbf{B}) = \mathbf{A}_3 \overline{\mathbf{B}}_3 + x_3 \mathbf{A}_2 \overline{\mathbf{B}}_2 + x_3 x_2 \mathbf{A}_1 \overline{\mathbf{B}}_1 + x_3 x_2 x_1 \mathbf{A}_0 \overline{\mathbf{B}}_0$
  - $(\mathbf{A} < \mathbf{B}) = \overline{\mathbf{A}}_3 \mathbf{B}_3 + x_3 \overline{\mathbf{A}}_2 \mathbf{B}_2 + x_3 x_2 \overline{\mathbf{A}}_1 \mathbf{B}_1 + x_3 x_2 x_1 \overline{\mathbf{A}}_0 \mathbf{B}_0$

Note:  $\mathbf{A}_i \overline{\mathbf{B}}_i$  indicates whether  $\mathbf{A}_i > \mathbf{B}_i$ ,  $\overline{\mathbf{A}}_i \mathbf{B}_i$  indicates whether  $\mathbf{A}_i < \mathbf{B}_i$ , and  $x_i$  indicates whether  $\mathbf{A}_i = \mathbf{B}_i$ .

**INTRO. TO COMP. ENG.  
CHAPTER XII-1**

**SINGLE CYCLE DPU**

**•CHAPTER XII**

# **CHAPTER XII**

## **SINGLE CYCLE DATAPATH UNIT**

**READ SINGLE CYCLE DATAPATH FREE-DOC ON COURSE WEBPAGE**

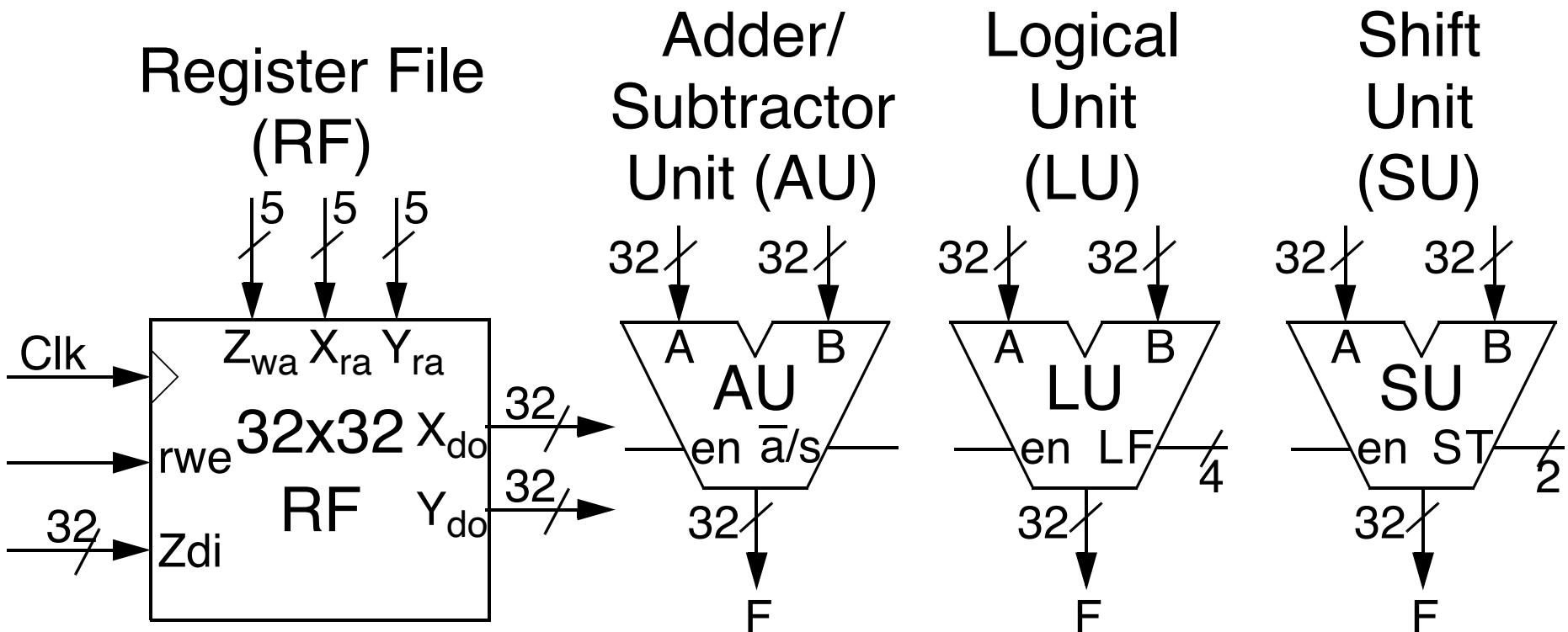
# SINGLE CYCLE DPU

## INTRODUCTION

•SINGLE CYCLE DPU  
-INTRODUCTION

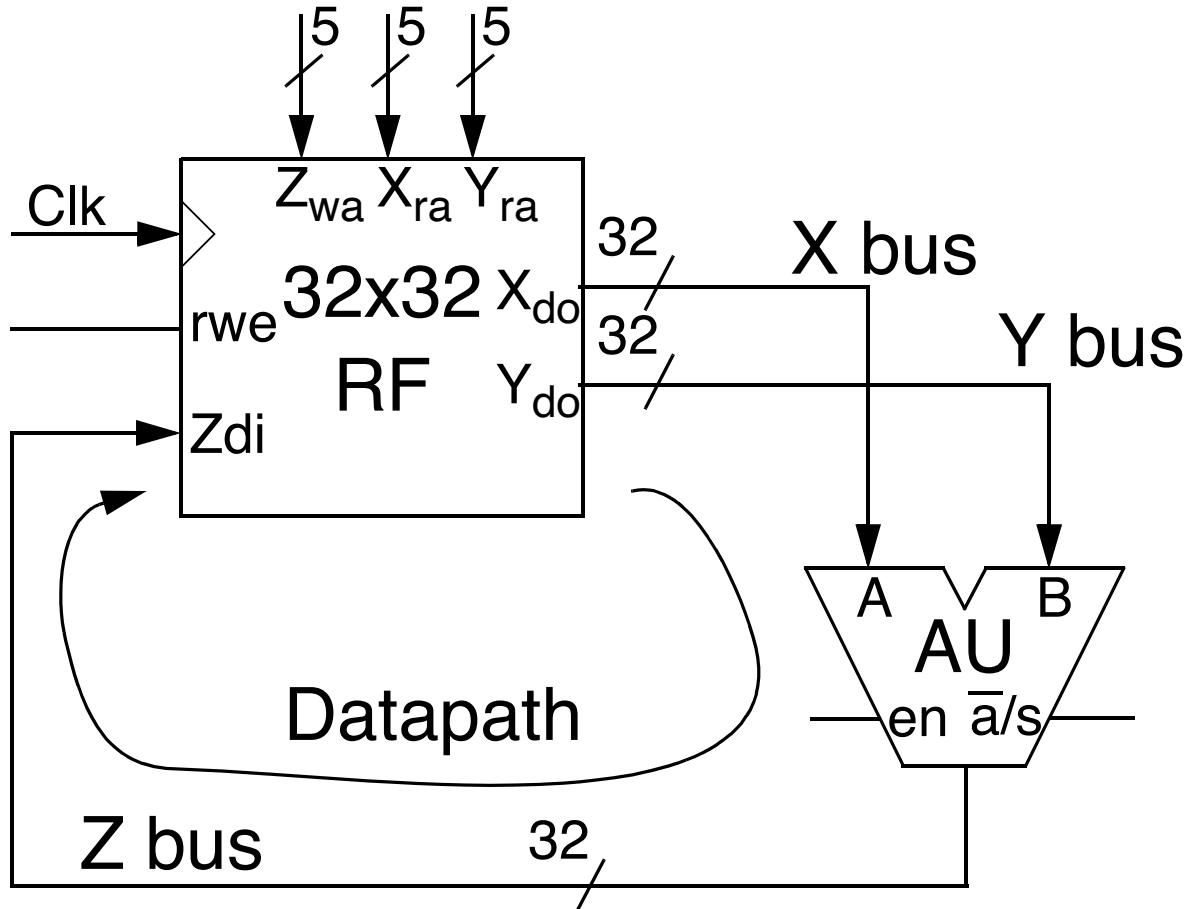
- From the previous chapter, we now have a number of datapath elements such as
  - Register file (**RF**)
  - Adder/subtractor unit (**AU**)
  - Logical unit (**LU**)
  - Shift unit (**SU**)
- The question now is how to take these datapath elements and form a **datapath unit (DPU)**.
- The DPU that we will focus on in this chapter is a basic single cycle DPU using a *triple bus internal architecture*.

- For our examples, we will use the following 32-bit type DPU elements.



- These allow us to design a 32-bit word computer with 32 registers.
- Of course, other word sizes could be used for other designs.

- Below is a simple datapath with a register file and adder/subtractor.



**Important:**  
It only takes 1 clock cycle to add/subtract and store the result.

- This structure is also known as a **triple bus internal DPU architecture**.

- This simple add/subtract machine DPU allows us to add or subtract values in our registers and store the result back into another register.
  - For instance, say that we wanted to **add** the contents of register **R1** with register **R2** and store the result back in register **R3**.

$$R3 = R1 + R2$$

- What control signals are required?
  - $\bar{a}/s = 0$  and  $en = 1$  for **AU**.
  - $X_{ra} = 00001$ ,  $Y_{ra} = 00010$ ,  $Z_{wa} = 00011$ , and  $rwe = 1$  for **RF**.
- These control signals are applied at the beginning of a clock cycle. The signals then propagate forming the sum at the output of the **AU**. At the end of the clock cycle, the sum (on **Z bus**) is clocked into **R3**.

- What if instead we wanted to perform the following operation.

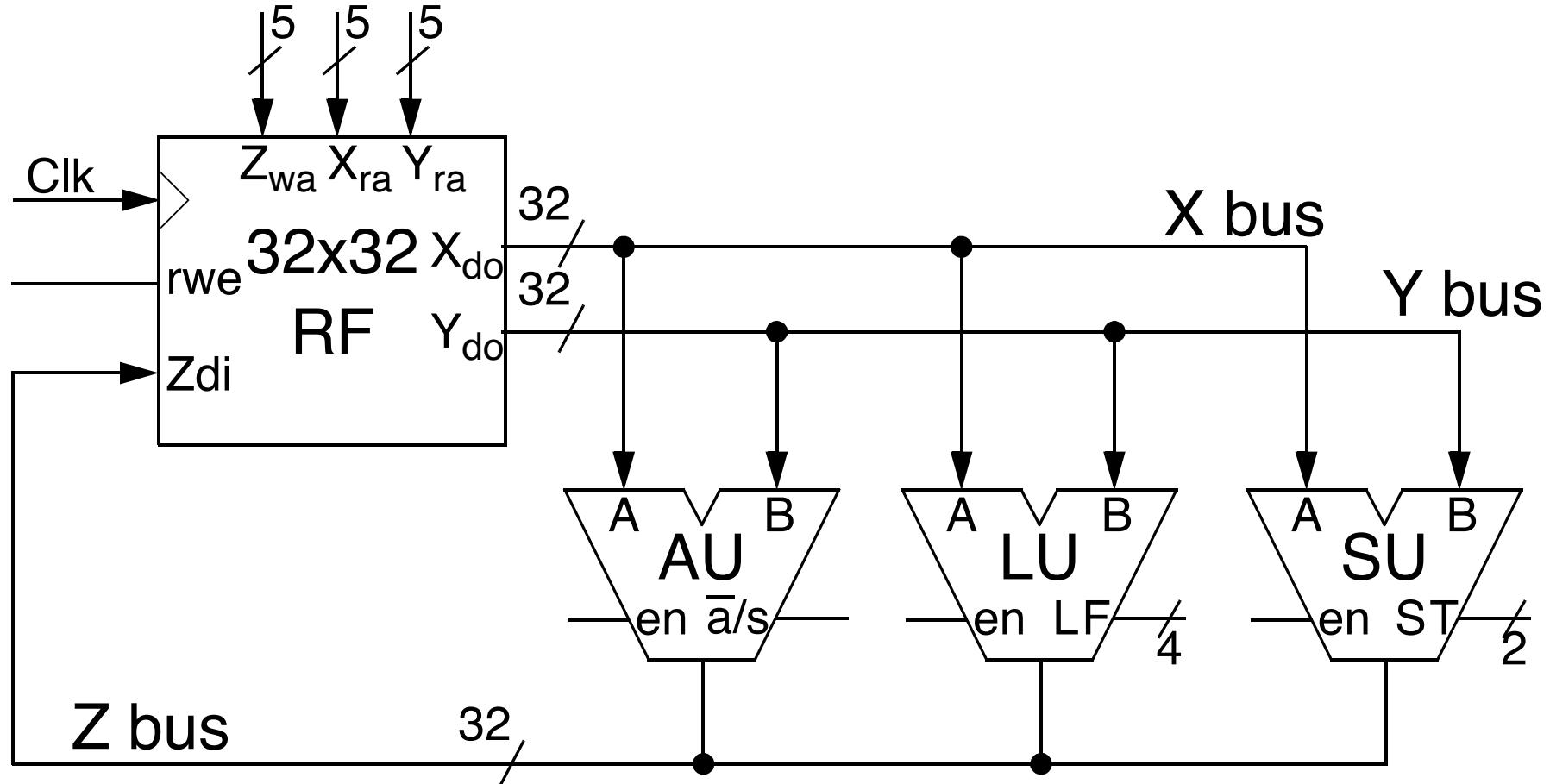
$$\mathbf{R2} = \mathbf{R1} + \mathbf{R2}$$

- The control signals required are
  - $\bar{a}/s = 0$  and  $en = 1$  for AU.
  - $X_{ra} = 00001$ ,  $Y_{ra} = 00010$ ,  $Z_{wa} = 00010$ , and  $rwe = 1$  for RF.
- What is the result of this if the current value of  $\mathbf{R1}=0x00000001$  and  $\mathbf{R2}=0x00000003$ ?
  - The register **R2** would be updated at the end of the clock cycle with the value **0x00000004**.
  - Remember, the current value of **R2** is put on the **X** or **Y bus**, and it is only at the **END** of the clock cycle that the contents of **R2** get changed.

# SINGLE CYCLE DPU

## BASIC SINGLE CYCLE DPU

- A more useful single cycle datapath can be as follows.



- This structure is still a **triple bus internal DPU architecture**.

- How does this change the additions we were doing earlier?
  - Say we want to again perform the following addition.

$$R3 = R1 + R2$$

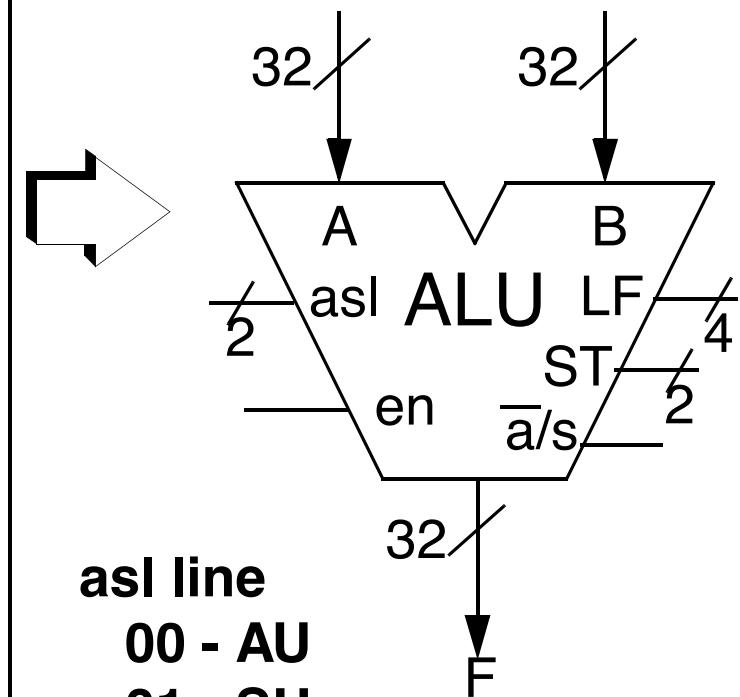
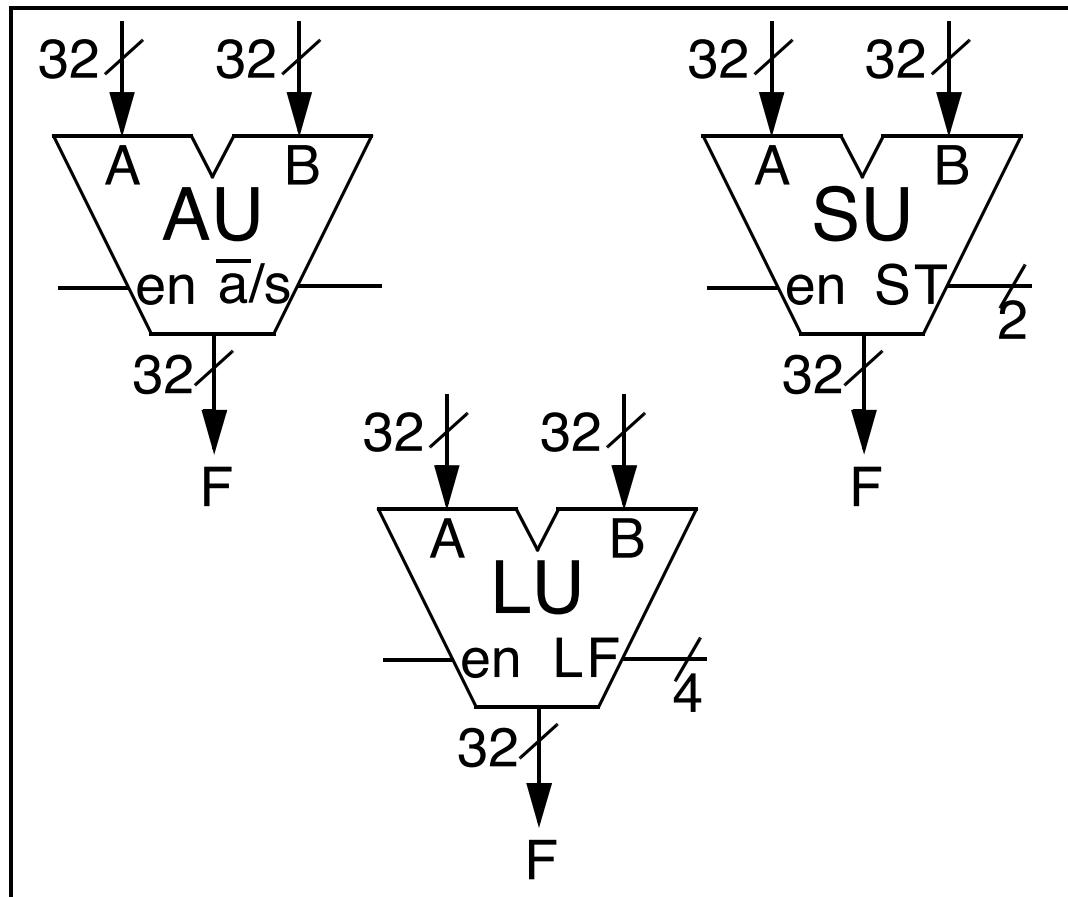
- The control signals we would need are
  - $\bar{a}/s = 0$  and  $en = 1$  for AU.
  - $en = 0$  for LU.
  - $en = 0$  for SU.
  - $X_{ra} = 00001$ ,  $Y_{ra} = 00010$ ,  $Z_{wa} = 00011$ , and  $rwe = 1$  for RF.
- Notice that we use the same control signals as before, but now include signals to disable the LU and SU during this addition clock cycle.

- Another operation we might want to do with this DPU is perform a logical shift of the contents of **R15** by a distance indicated in **R6**.
  - The control signals required are
    - **en = 0** for **AU**.
    - **en = 0** for **LU**.
    - **en = 1** and **ST = 00** for **SU**.
    - **X<sub>ra</sub> = 01111**, **Y<sub>ra</sub> = 00110**, **Z<sub>wa</sub> = 01111**, and **rwe = 1** for **RF**.
  - Notice that this set of control signals disables the **AU** and **LU** while enabling the **SU**.
  - The **SU** is set to do a logical shift with **ST = 00**.
  - The distance of the shift is according to what is in **R6**.
  - The result is stored back in **R15** with **Z<sub>wa</sub> = 01111** and **rwe = 1** for **RF**

# SINGLE CYCLE DPU

## ARITHMETIC LOGIC UNIT

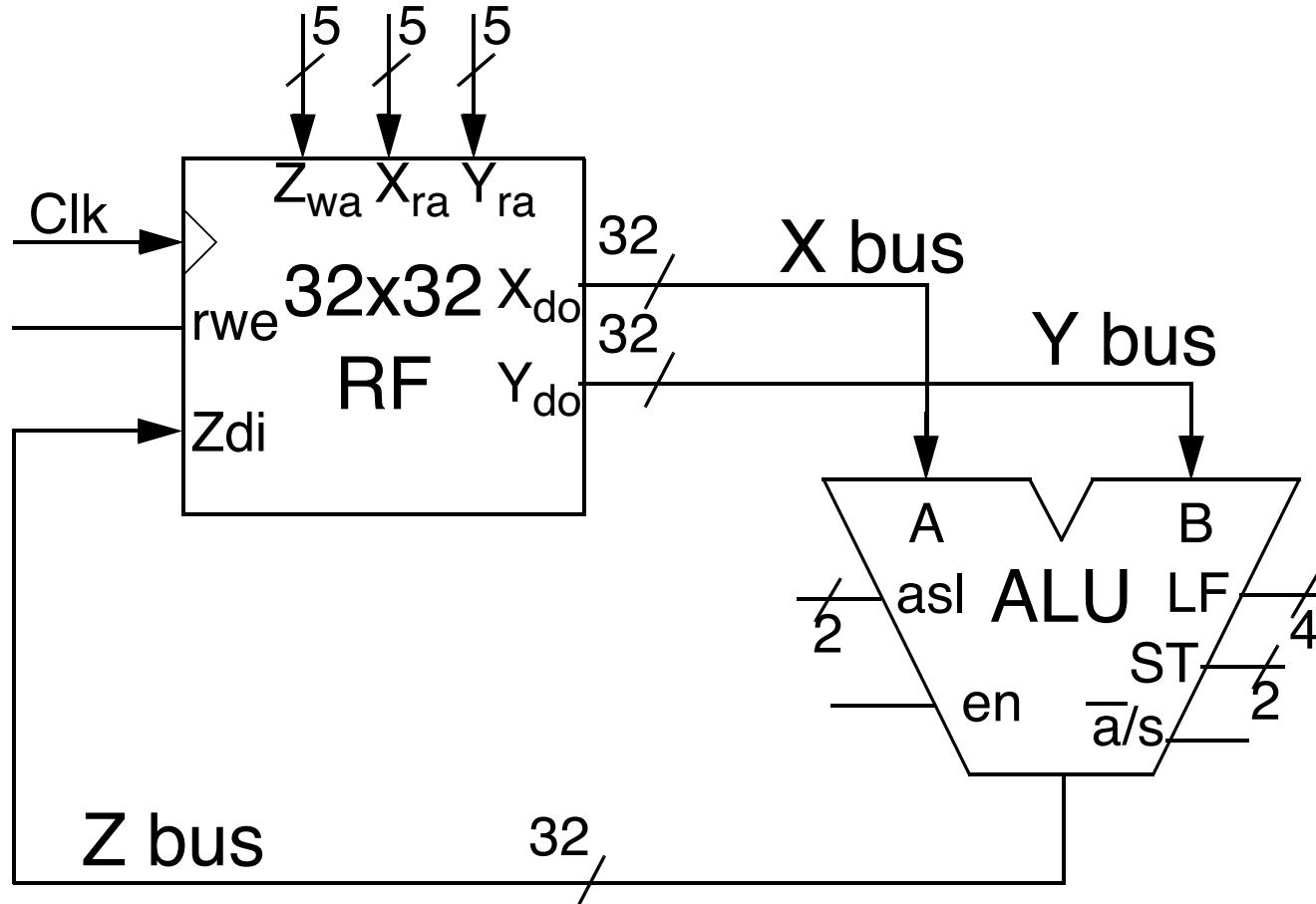
- Since only one of **AU**, **SU**, or **LU** will be active at a time in this architecture, we will combine to form an **arithmetic logic unit** (ALU).



# SINGLE CYCLE DPU

## SINGLE CYCLE DPU W/ALU

- Using our **ALU**, the DPU can be redrawn as follows.

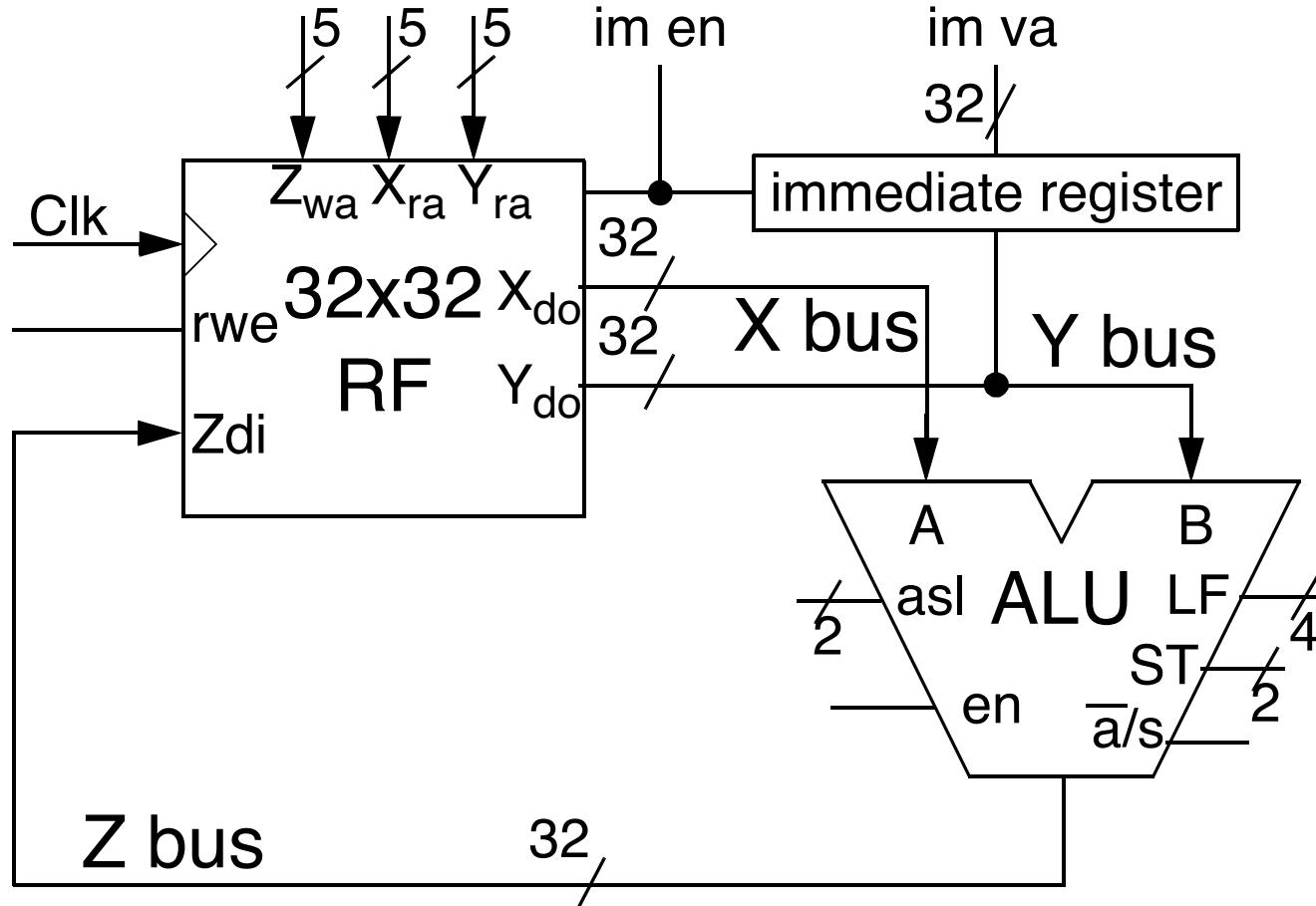


- This structure is still a **triple bus internal DPU architecture**.

# SINGLE CYCLE DPU

## IMMEDIATE REGISTER

- Many designs also include some form of immediate register.



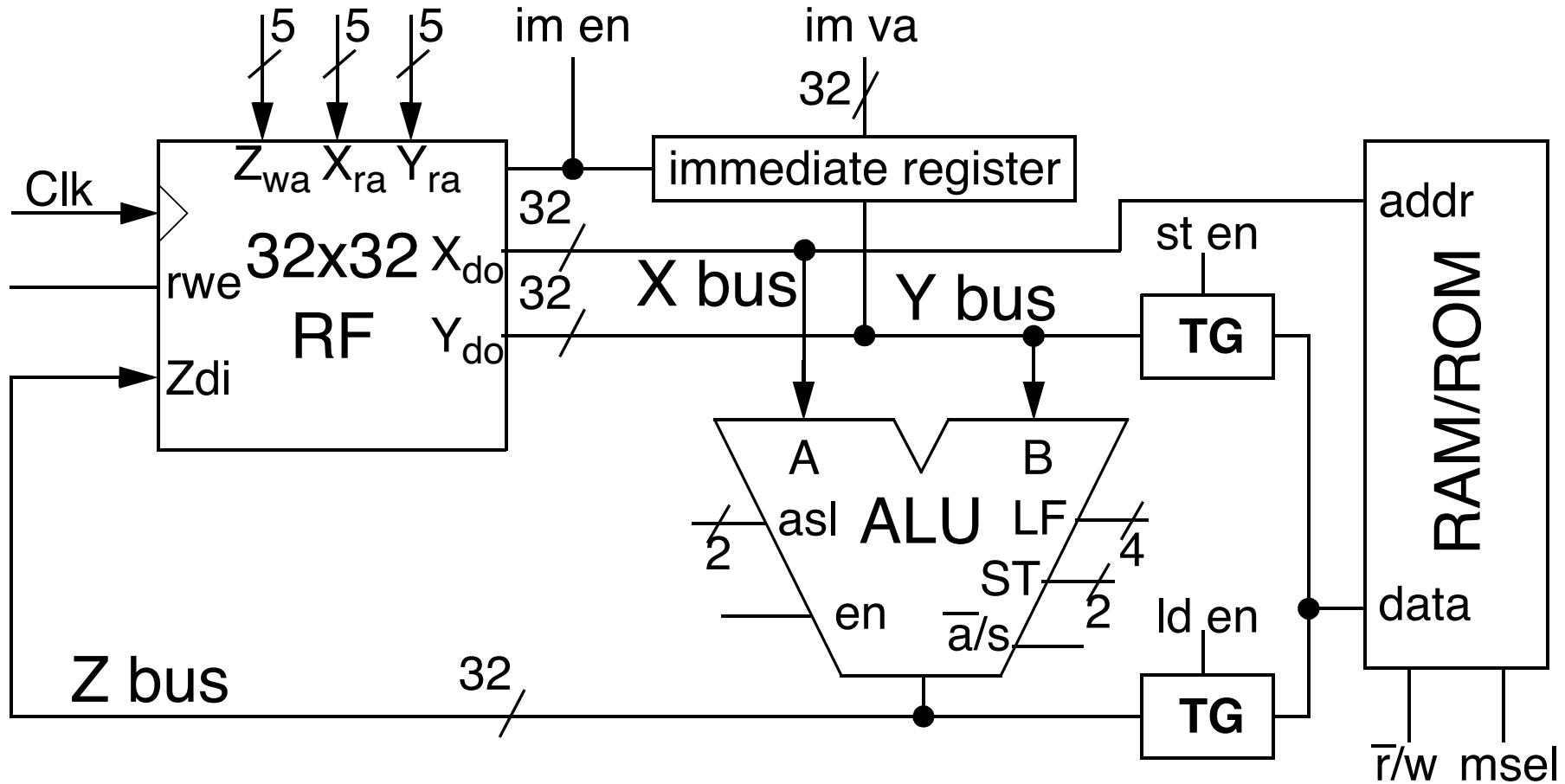
- Allows for operations such as **R28 = R5 + Immediate**.

- The **im\_en** line does two things:
  - When 0, **im\_en** controls
    - **immediate register outputs** to go to **high impedance** so as **NOT** to affect **Y bus**.
    - **register file Y data out** to output corresponding register value.
  - When 1, **im\_en** controls
    - **immediate register** to output register value to **Y bus**.
    - **register file Y data out** to go to **high impedance** so as **NOT** to affect **Y bus**.
- The **im\_va** lines pass a value to the immediate register.

# SINGLE CYCLE DPU

## INCLUDING MEMORY

- 32x32 bits is not sufficient memory for most computers.
- We can include external memory (SRAM, DRAM, etc.) as follows.



# SINGLE CYCLE DPU

## INCLUDING MEMORY

- The included has the following characteristics
  - 32 address lines
  - 32 data lines
  - a read/write line
  - a chip select or memory select line
- Two transmission gates block the bidirectional data lines for the memory.
  - Notice that if **st\_en** is high, then we can potentially write to the memory.
  - Notice that if **ld\_en** is high, then we can potentially read from the memory.

# SINGLE CYCLE DPU

## READING FROM MEMORY

- We wish to be able to read and write from our memory.
- A sample read/load operation can be expressed as follows

$$R4 = M[R7]$$

- This operation uses the value in **R7** as the address to the memory and reads the value at that address in the memory to **R4**.
- What control signals are required?
  - **en** = 0 for **ALU**.
  - **X<sub>ra</sub>** = 00111, **Y<sub>ra</sub>** = XXXXX, **Z<sub>wa</sub>** = 00100, and **rwe** = 1 for **RF**.
  - **st\_en** = 0 and **ld\_en** = 1
  - **~r/w** = r and **msel** = 1

# SINGLE CYCLE DPU

## WRITING TO MEMORY

- A sample write/store operation can be expressed as follows

$$M[R5] = R9$$

- This operation uses the value in **R5** as the address to the memory and write the value in **R9** to that address in the memory.
- What control signals are required?
  - **en = 0** for **ALU**.
  - **X<sub>ra</sub> = 00101**, **Y<sub>ra</sub> = 01001**, **Z<sub>wa</sub> = XXXXX**, and **rwe = 0** for **RF**.
  - **st\_en = 1** and **Id\_en = 0**
  - **~r/w = w** and **msel = 1**

- Microcode in a processor are all of the control signals required to execute an operation for a clock cycle.
- We have actually looked at examples of a microcode operation when we considered various operations such as

$$\mathbf{R3} = \mathbf{R1} + \mathbf{R2}$$

or

$$\mathbf{M[R5]} = \mathbf{R9}$$

- Later we will talk about macrocode which are longer operations consisting of many microcode operation over a number of clock cycles.

**INTRO. TO COMP. ENG.  
CHAPTER XI-1**

**DATAPATH ELEMENTS**

**•CHAPTER XI**

# **CHAPTER XI**

## **DATAPATH ELEMENTS**

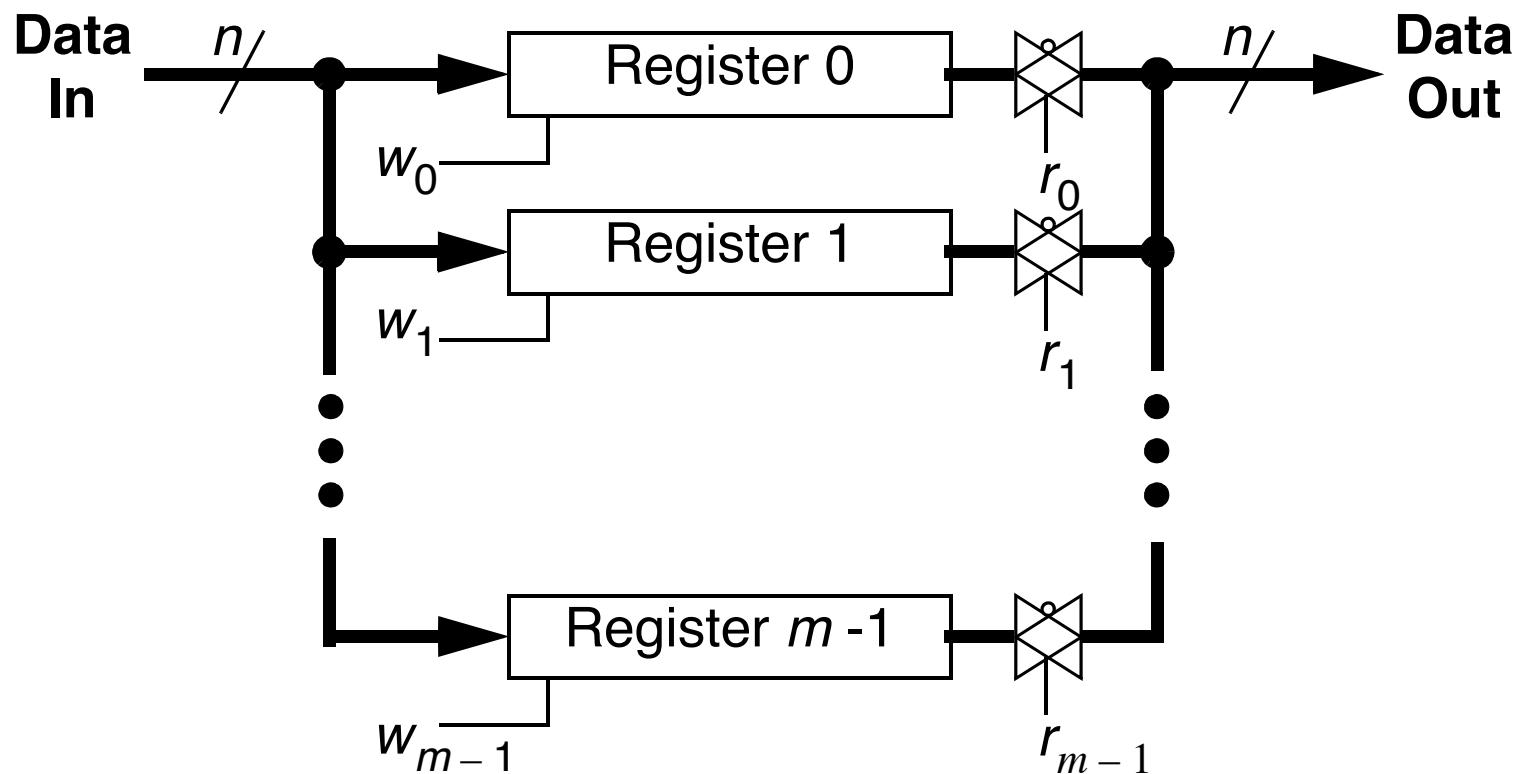
**READ DATAPATH ELEMENTS FREE-DOC ON COURSE WEBPAGE**

# DATAPATH ELEMENTS

## INTRODUCTION

- So far we have discussed many small components and building blocks.
- One final step in our building blocks before we can start to piece together a microprocessor is various datapath elements.
  - We have already discussed portions of these datapath elements in terms of other components and building blocks.
  - We will now consider some of these components and building blocks in ways that will make the design of a microprocessor a little easier in the next chapter.

- A general  $m \times n$  register file with  $m$  registers that are each  $n$ -bits wide is illustrated below.

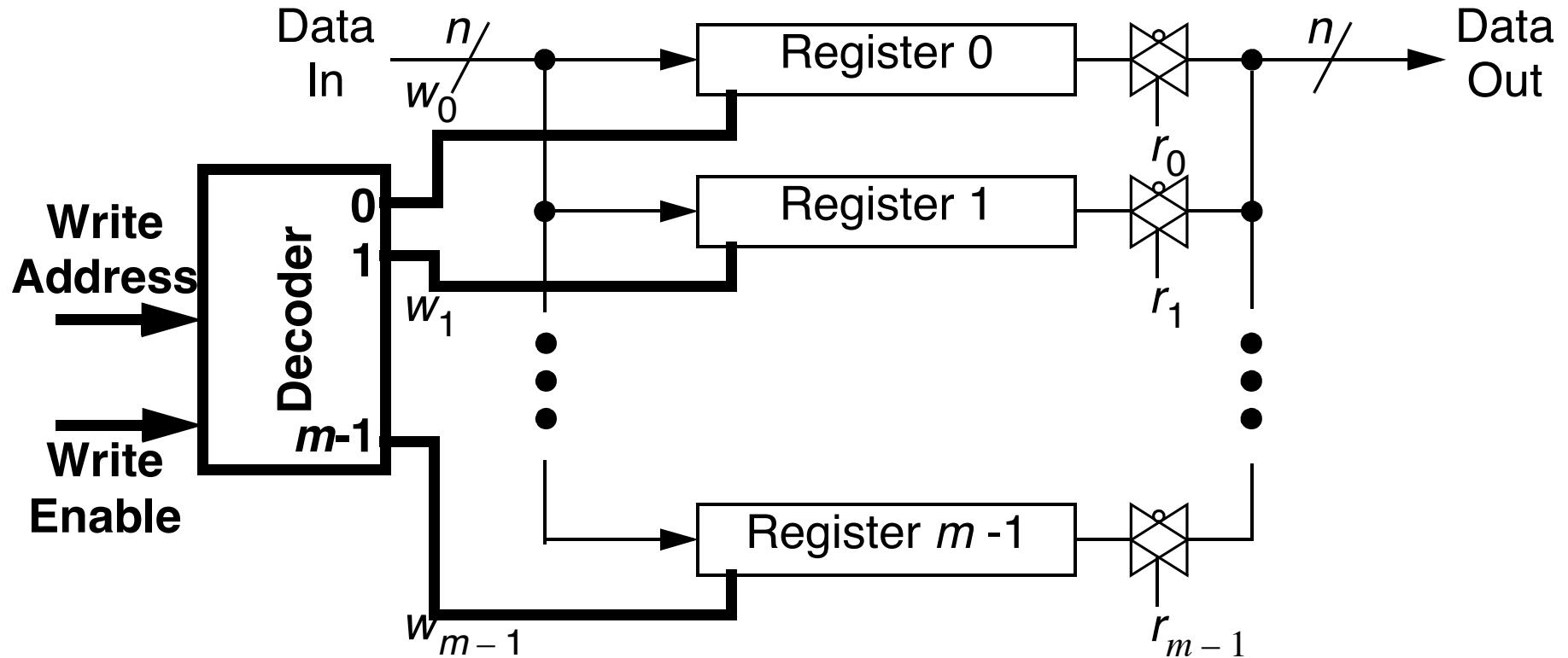


- The  $r_k$  and  $w_j$  signals indicate which register to read/write, respectively.

# REGISTER FILES

## WRITE DECODER

- For writing to a register, we include a write address with decoder.

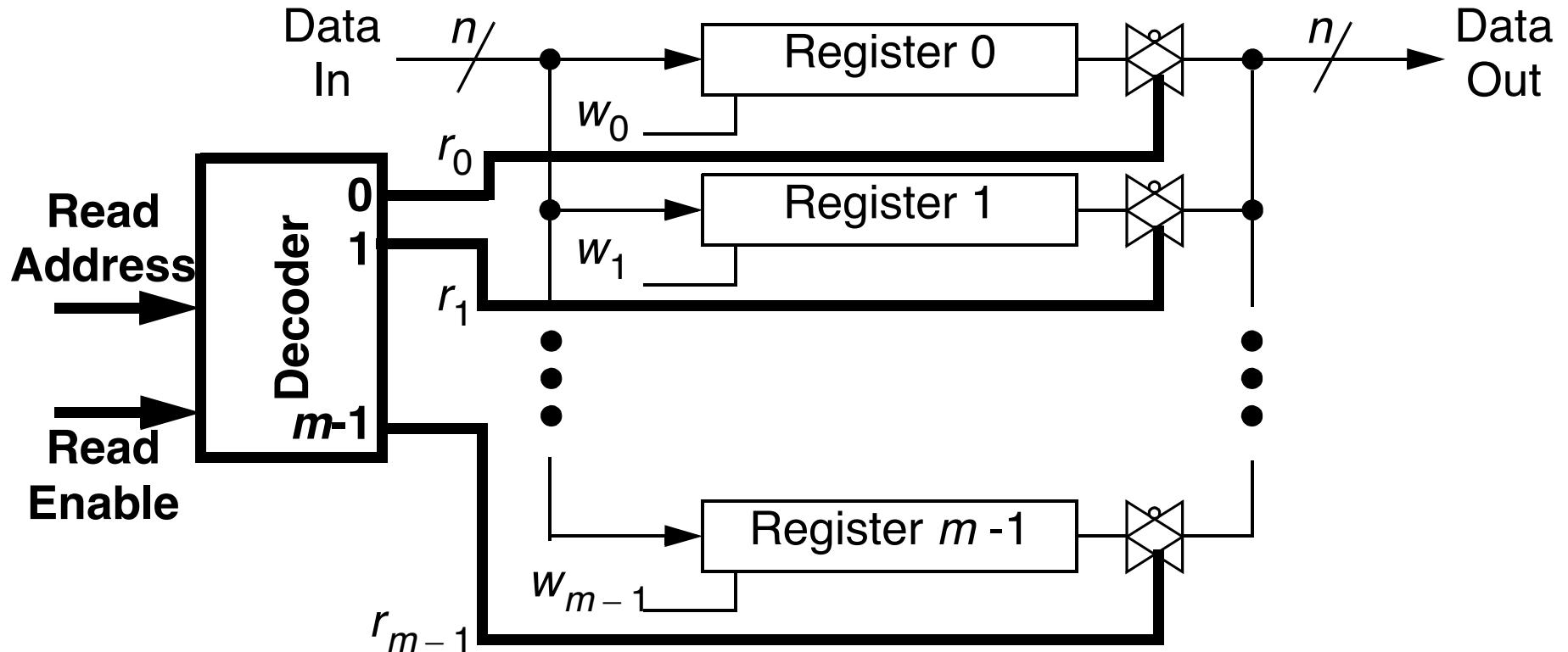


- A given **Write Address** (with **Write Enable = 1**) selects which register, 0 through  $m - 1$ , to store the input from **Data In**.

# REGISTER FILES

## READ DECODER

- For reading from a register, we include a read address with decoder.



- A given **Read Address** (with **Read Enable** = 1) selects which register, 0 through  $m - 1$ , to read from and output to Data Out.
- Could have multiple **data outputs** with multiple **read address** decoders.

# REGISTER FILES

32-BIT WORD, 32 REGISTERS

- For the upcoming datapath designs in the next chapter, we want to have a 32x32 register file with one write input and two read outputs.

$X_{ra}$  - X read address

$Y_{ra}$  - Y read address

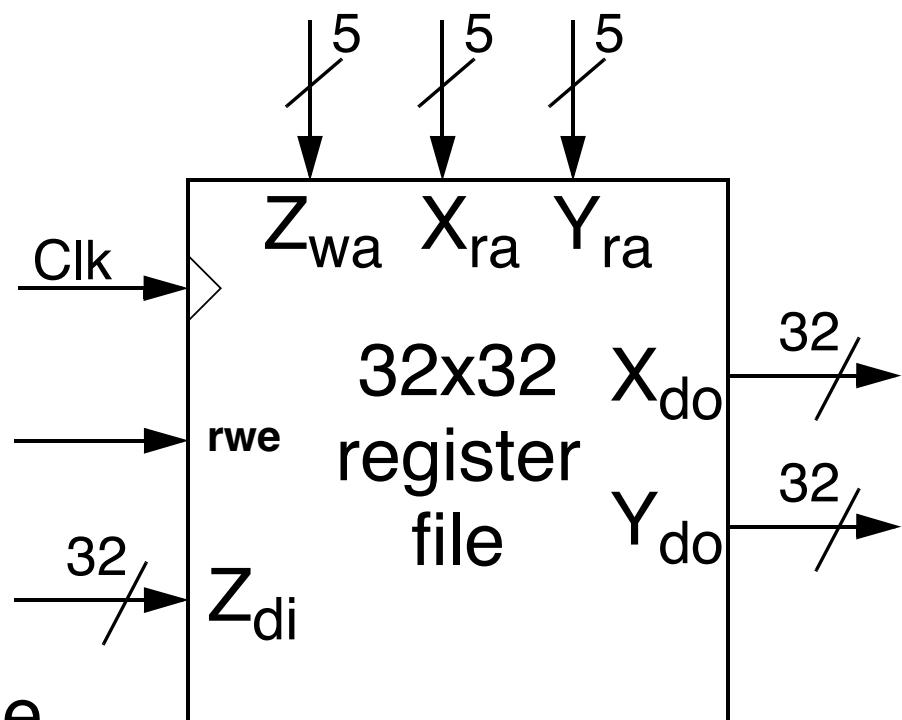
$Z_{wa}$  - Z write address

$X_{do}$  - X data out

$Y_{do}$  - Y data out

$Z_{di}$  - Z data in

rwe - register write enable

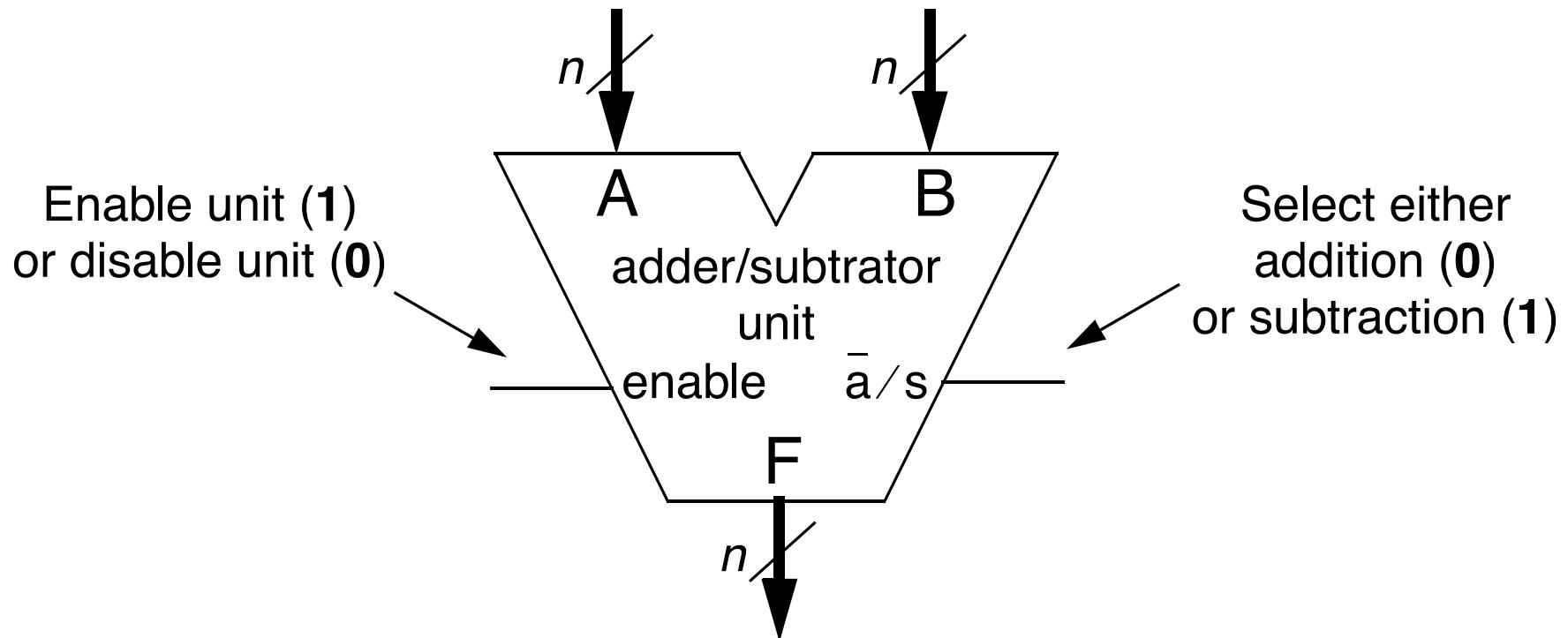


- Note: Two data outputs implemented with two read address decoders.

# ADDER/SUBTRACTOR

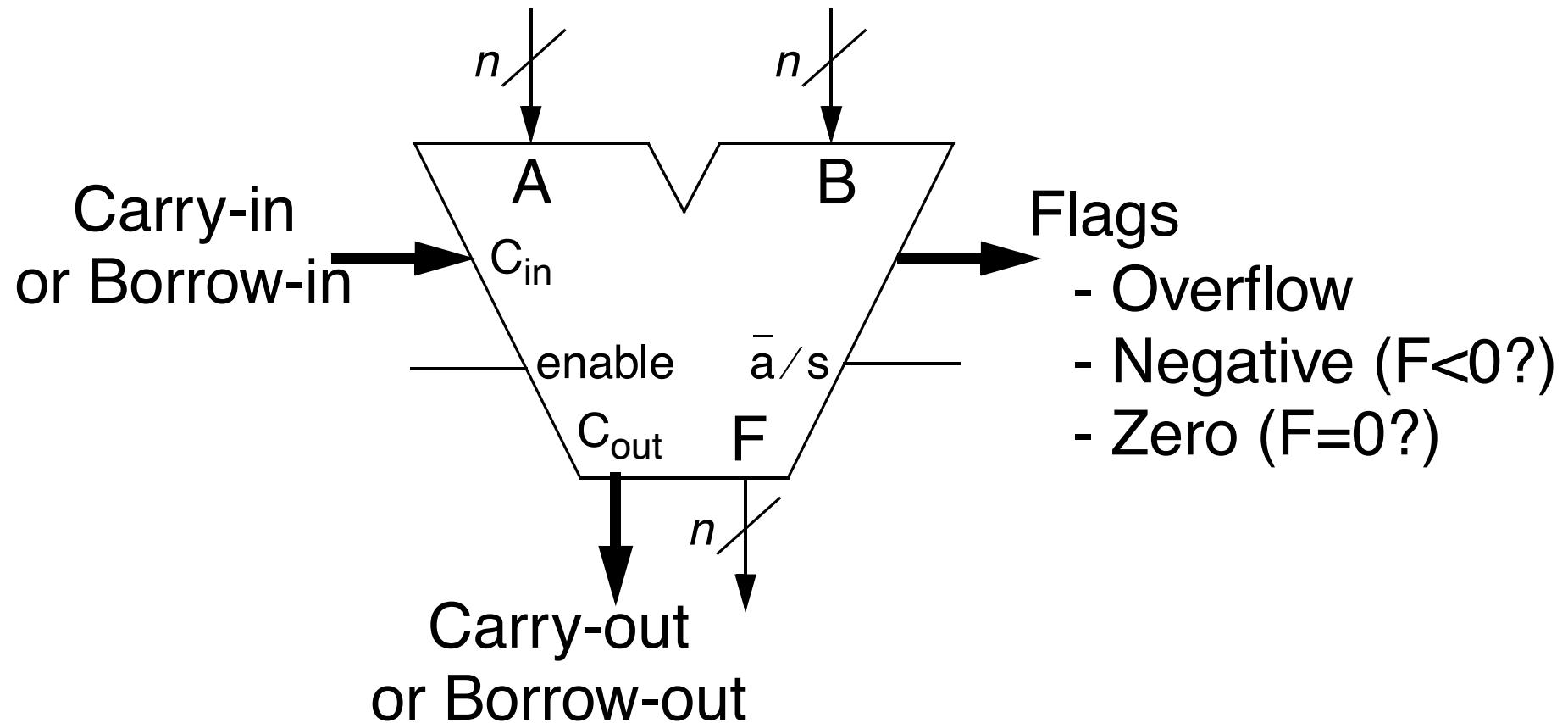
## GENERAL UNIT DIAGRAM

- An  $n$ -bit adder/subtractor unit is often illustrated as follows.



- This unit would have  $n$  full-adders internally.

- Other signals often included with an adder/subtractor are shown below.



- A useful unit would be one that can take two  $n$ -bit inputs and perform some logical operation between each of the bits to get an  $n$ -bit output.
  - For example, given the 8-bit values 0001 1110 and 1001 1000, we might want to find the **bit-wise logical OR**.

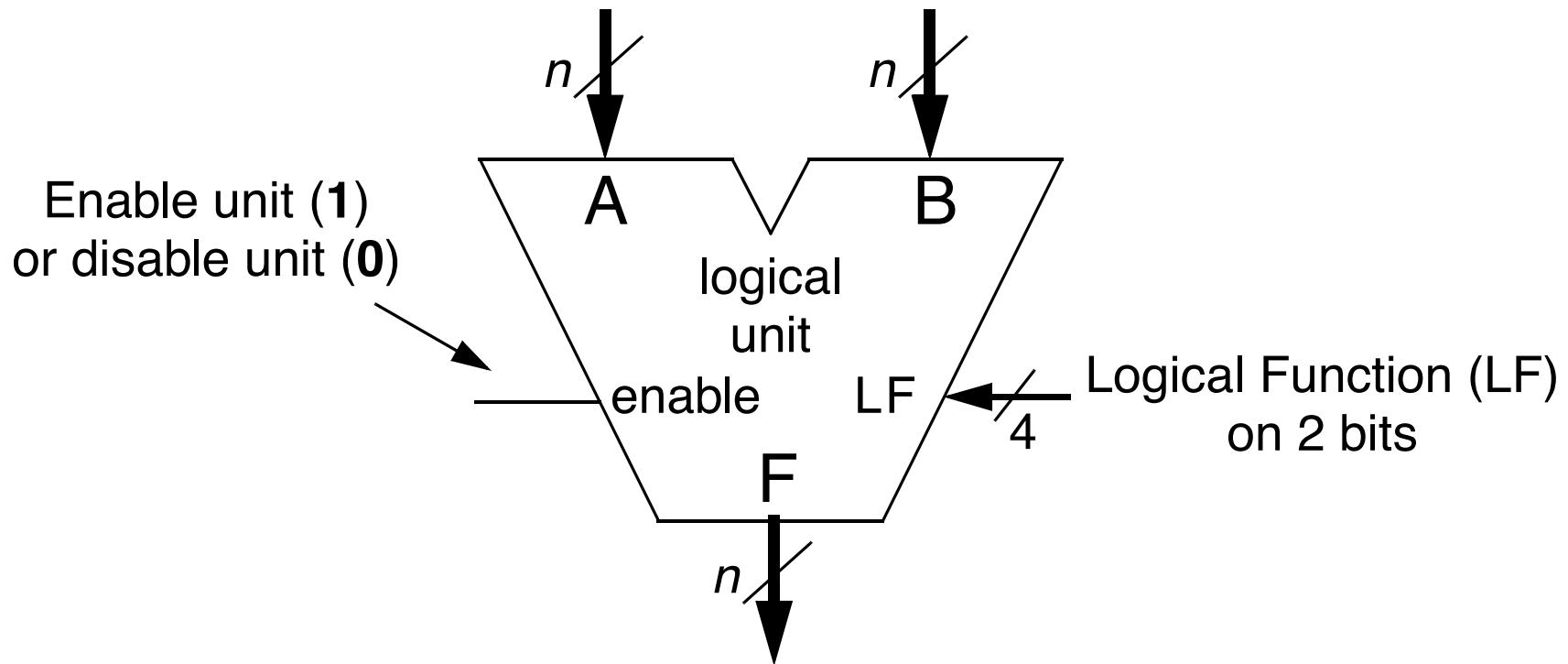
<b>bit-wise</b>	0001 1110
<b>logical OR</b>	1001 1000
1001 1110	

- Or similarly, the **bit-wise logical AND** of the two 8-bit values.

<b>bit-wise</b>	0001 1110
<b>logical AND</b>	1001 1000
0001 1000	

- These types of operations are often used for masking and setting bits.

- Below is a general unit diagram for an  $n$ -bit logical unit.

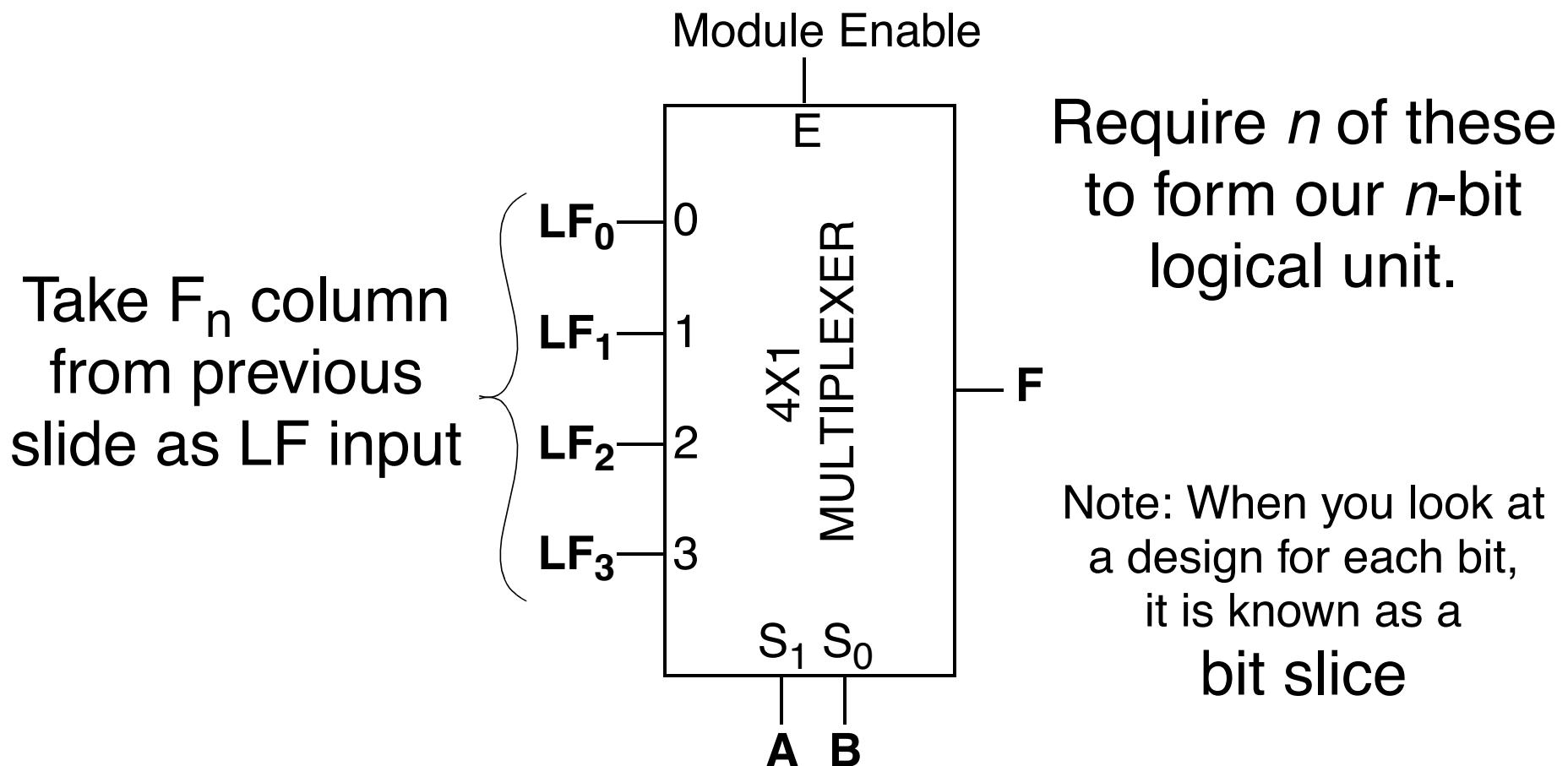


- Logical operations, such as **AND/OR/NOT/NAND/NOR/etc.**, are done for each bit of **A** and **B** to form **F**.

# LOGICAL UNIT

- Recall the possible logic functions for two bits, A and B.
    - We can use the column  $F_n$  as the 4-bit LF input for the logical unit.

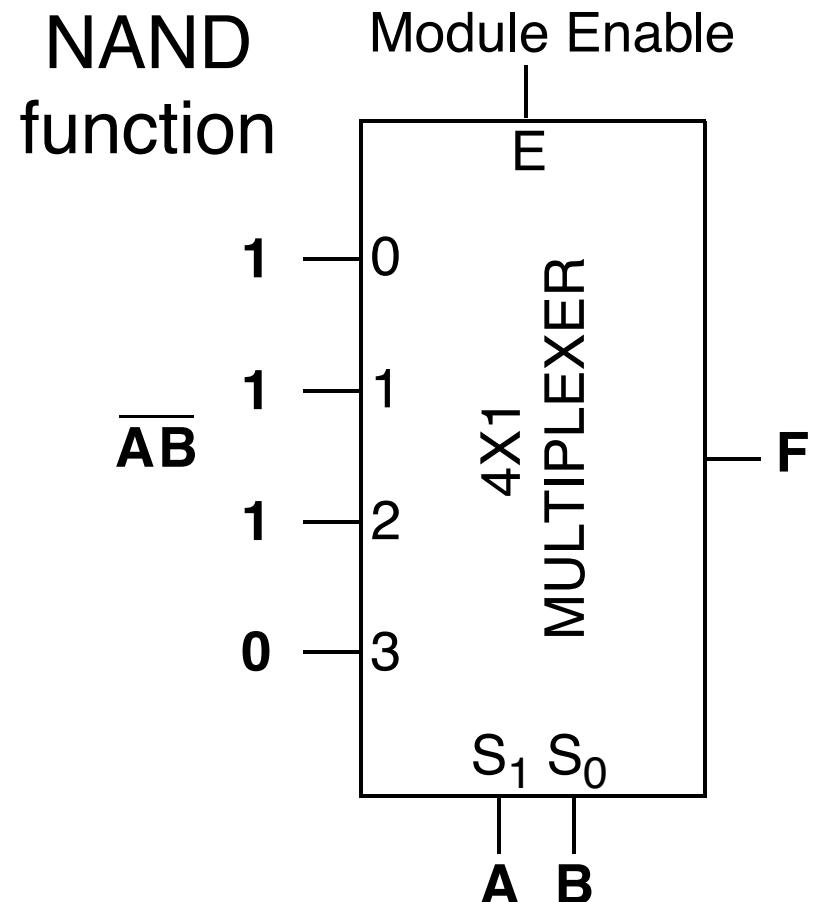
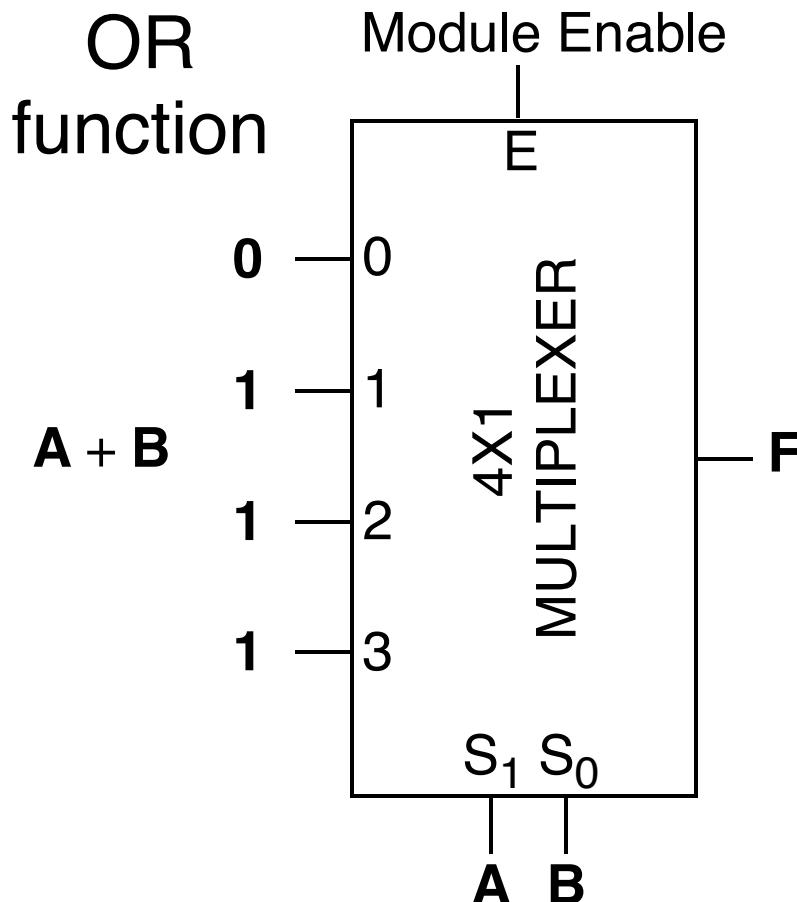
- A number of internal implementations exist for the logical unit.
  - The easiest is to use a **4-to-1 multiplexer** for each bit as follows



# LOGICAL UNIT

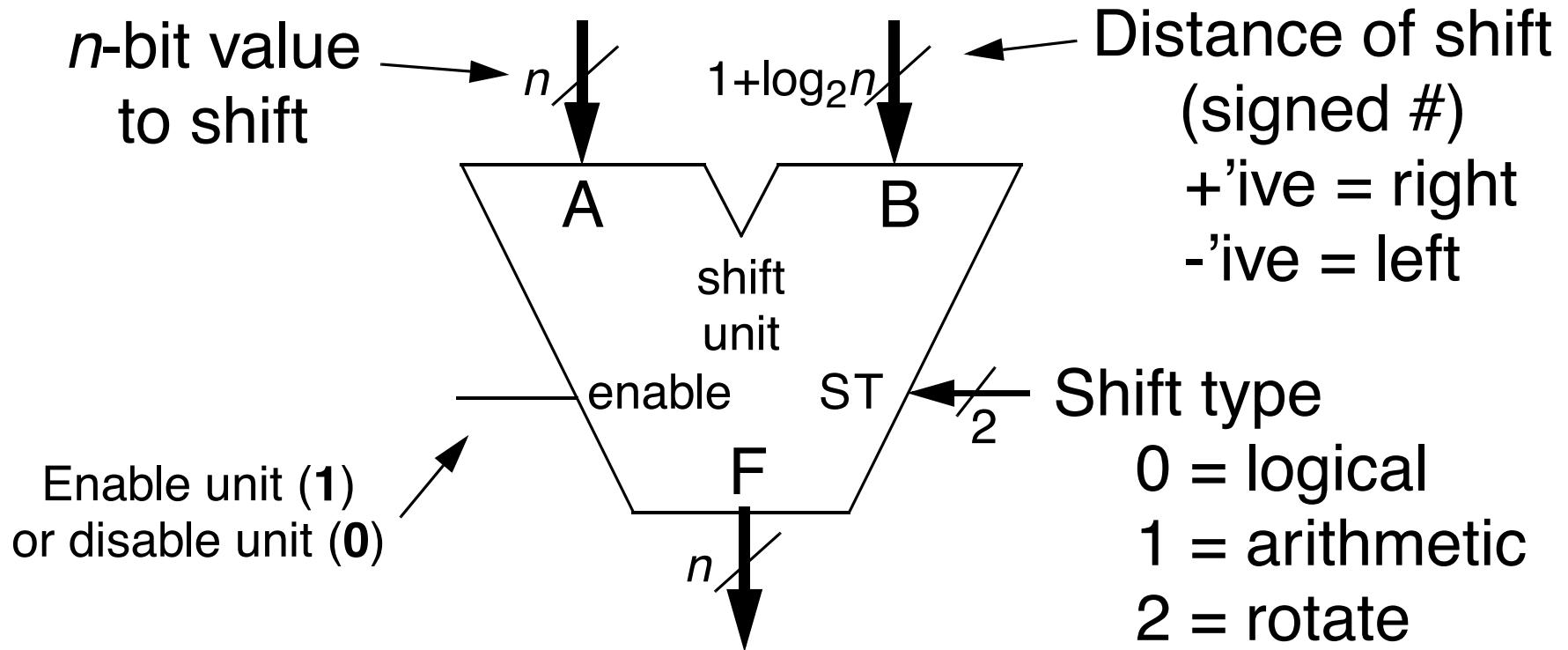
## BIT SLICE IMPLEMENTATION

- The following are example LF inputs for a logical unit bit slice.



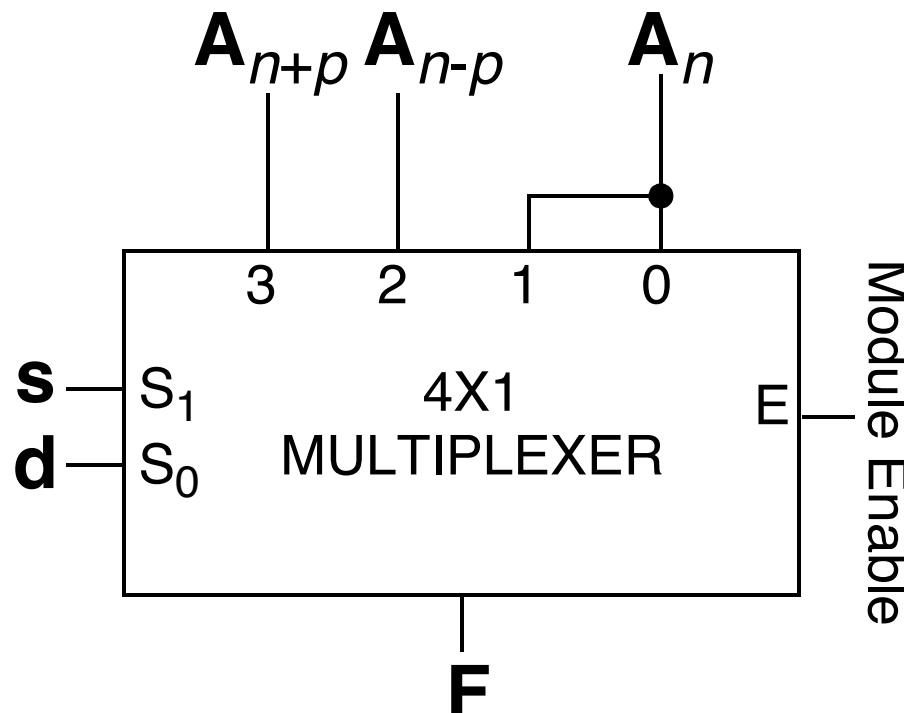
- We have already discussed the bulk about shift units in previous chapters.
- As given in the Free-Doc, there are different types of shift units.
  - Logical shift
  - Arithmetic shift
  - Circular shift (this is just a rotate unit)
- We want to discuss an implementation, the barrel shifter, that will be useful in our single cycle datapath computer we will design next chapter.

- Below is a general unit diagram for an  $n$ -bit shift unit.



- Notice that the  $n$ -bit value **A** will be shifted according to the distance indicated with signed number **B**.

- Previously, we discussed the *p*-shifter but not its implementation.
  - A *p*-shifter shifts the value to the left or right by *p*-bits.
  - A bit slice view of a *p*-shifter for *n*th bit could be as follows.



<b>s</b>	
0 = no shift	
1 = shift	
<b>d</b>	
0 = left	
1 = right	

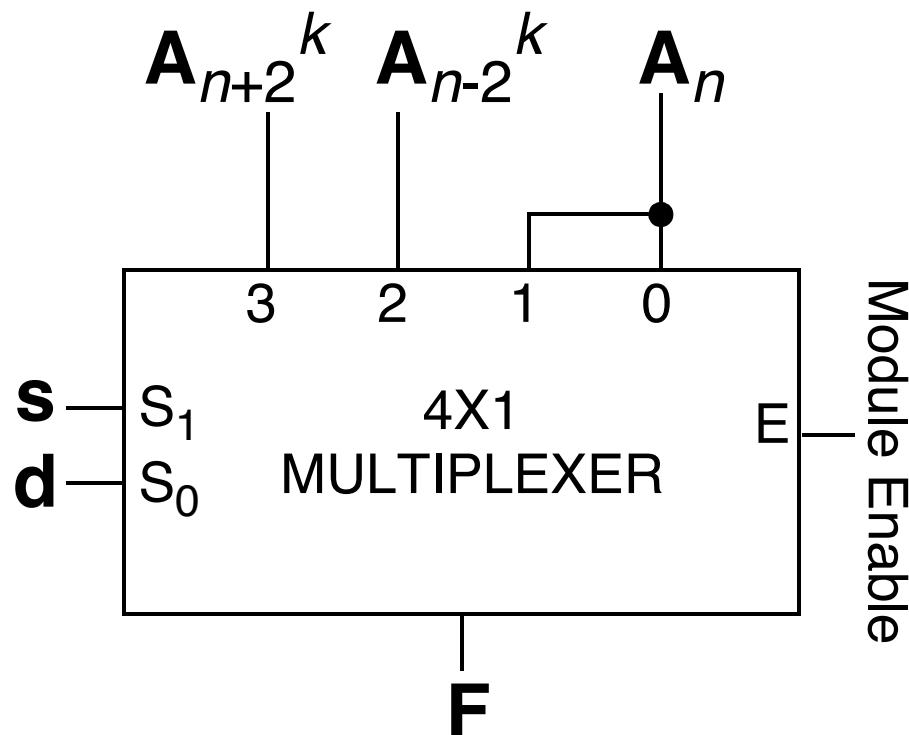
- Notice that this can **ONLY** shift by *p*-bits. It is **hardwired** to shift *p*-bits.

# SHIFT UNIT

$2^k$ -SHIFTER BIT SLICE

- SHIFT UNIT
  - INTRODUCTION
  - GENERAL UNIT DIAGRAM
  - $P$ -SHIFTER BIT SLICE

- A useful type of  $p$ -shifter is when  $p = 2^k$  for some positive integer  $k$ .



<b>s</b>	
0	= no shift
1	= shift
<b>d</b>	
0	= left
1	= right

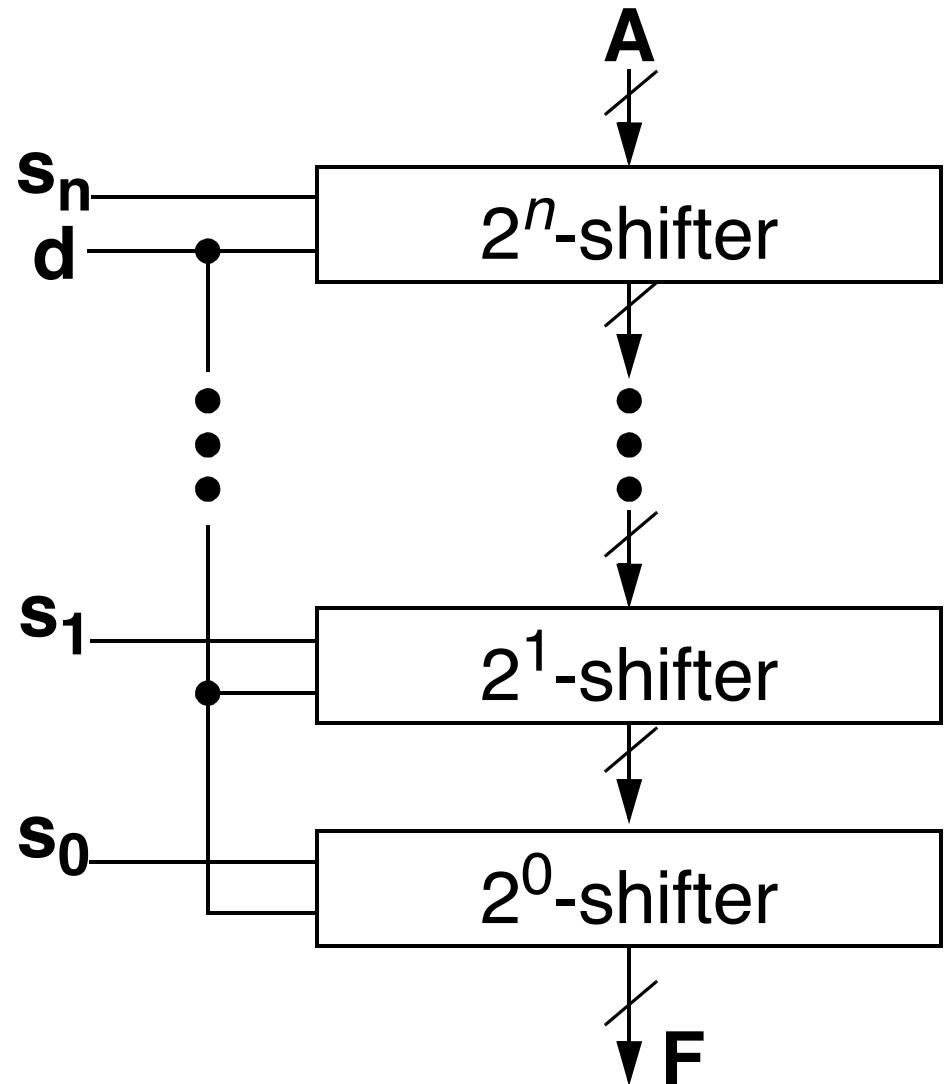
- A  $2^k$ -shifter allows use to build a barrel shifter.

# SHIFT UNIT

## BARREL SHIFTER

- We want to be able to shift a vector by an arbitrary distance instead of hardwired like the  $p$ -shifter and  $2^k$ -shifter.

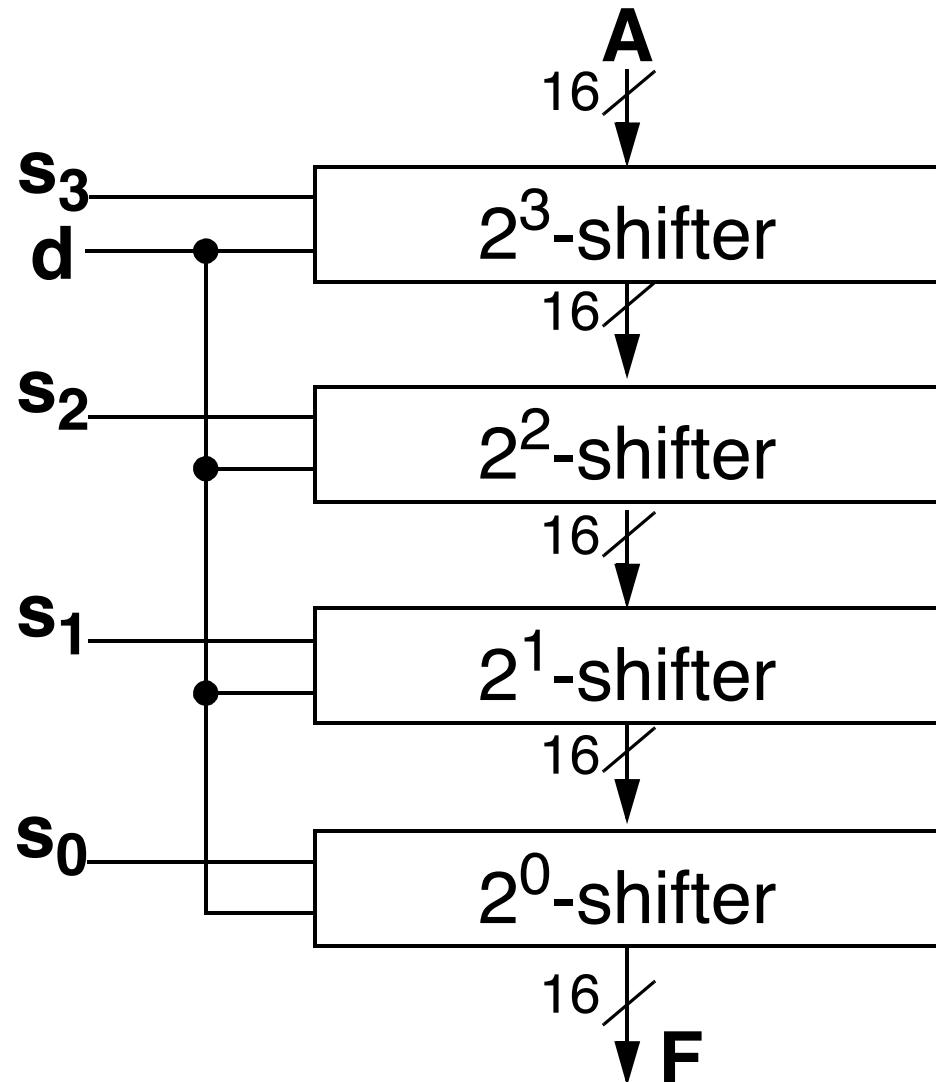
- The top level can shift  $A$  by  $n$  bits, depending on  $s_n$ .
- Subsequent levels can shift result by  $n/2$  bits, depending on their input  $s_q$ .



# SHIFT UNIT

## SAMPLE BARREL SHIFTER

- We will do some examples with the following arbitrary  $n$ -shifter on a 16-bit input.
- Note that this barrel shifter can shift the input by 15 bits in either direction.



# SHIFT UNIT

## BARREL SHIFTER: EXAMPLE #1

- SHIFT UNIT
  - $2^k$ -SHIFTER BIT SLICE
  - BARREL SHIFTER
  - SAMPLE BARREL SHIFTER

- For example, consider the input of

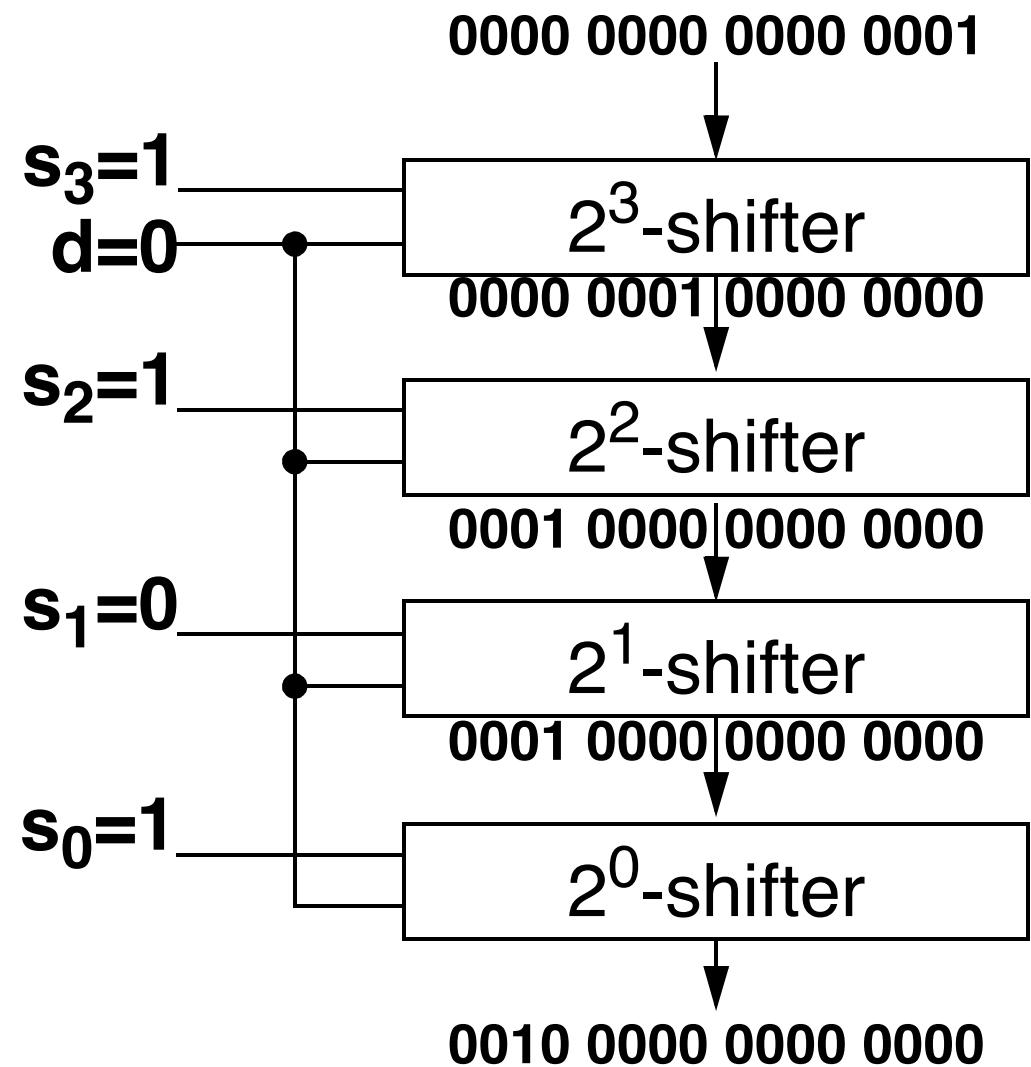
**0000 0000 0000 0001**

- If we want to shift this value to the **left** by 13, we need the input

**d = 0**

**s = ( $s_3 s_2 s_1 s_0$ ) = 1101**

- Note: This example is for a logical shift.



# SHIFT UNIT

## BARREL SHIFTER: EXAMPLE #2

- SHIFT UNIT
  - BARREL SHIFTER
  - SAMPLE BARREL SHIFTER
  - BARREL SHIFTER EX. #1

- As another example,  
consider the input of

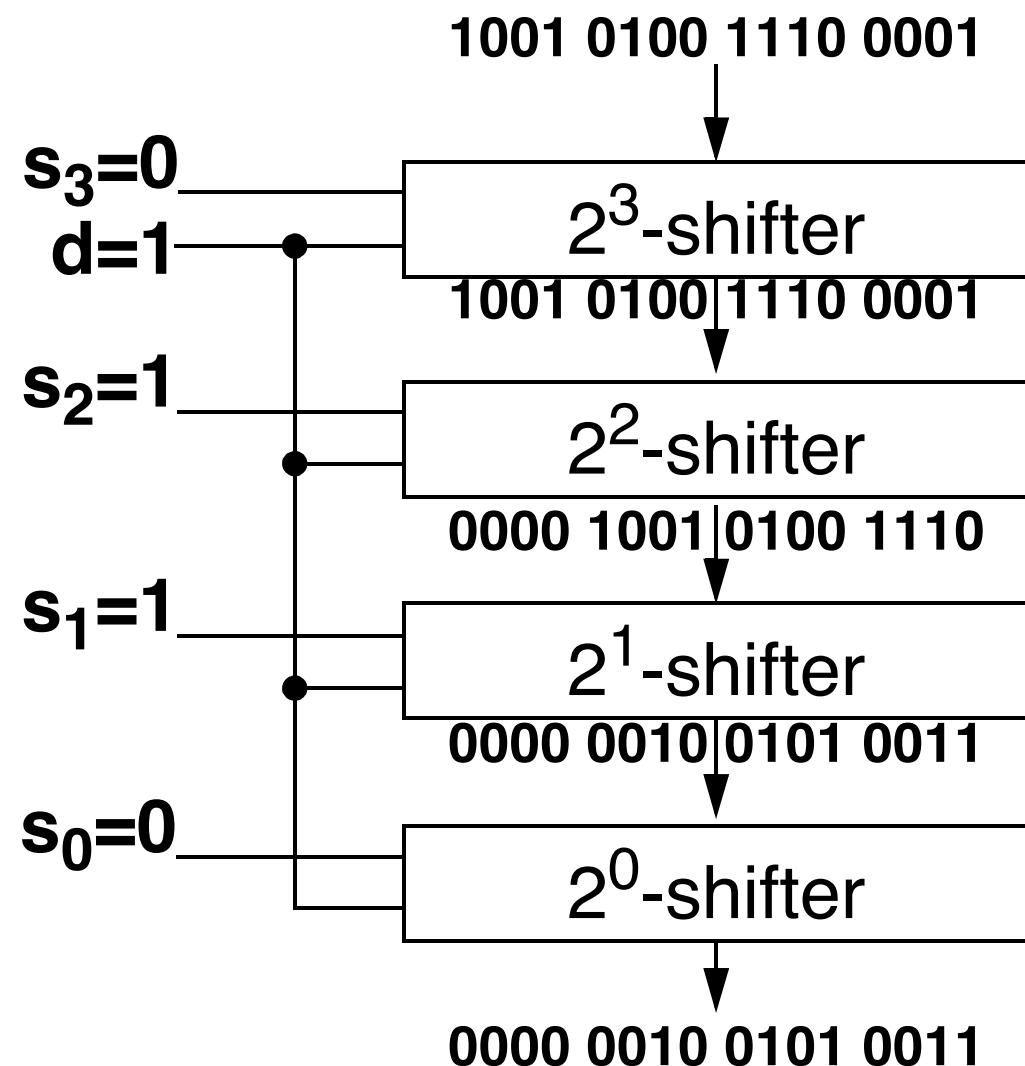
**1001 0100 1110 0001**

- If we want to shift this value  
to the **right** by 6, we need  
the input

**d = 1**

**s = (s<sub>3</sub>s<sub>2</sub>s<sub>1</sub>s<sub>0</sub>) = 0110**

- Note: This example is for a  
logical shift.



**INTRO. TO COMP. ENG.  
CHAPTER XII-1**

**SINGLE CYCLE DPU**

**•CHAPTER XII**

# **CHAPTER XII**

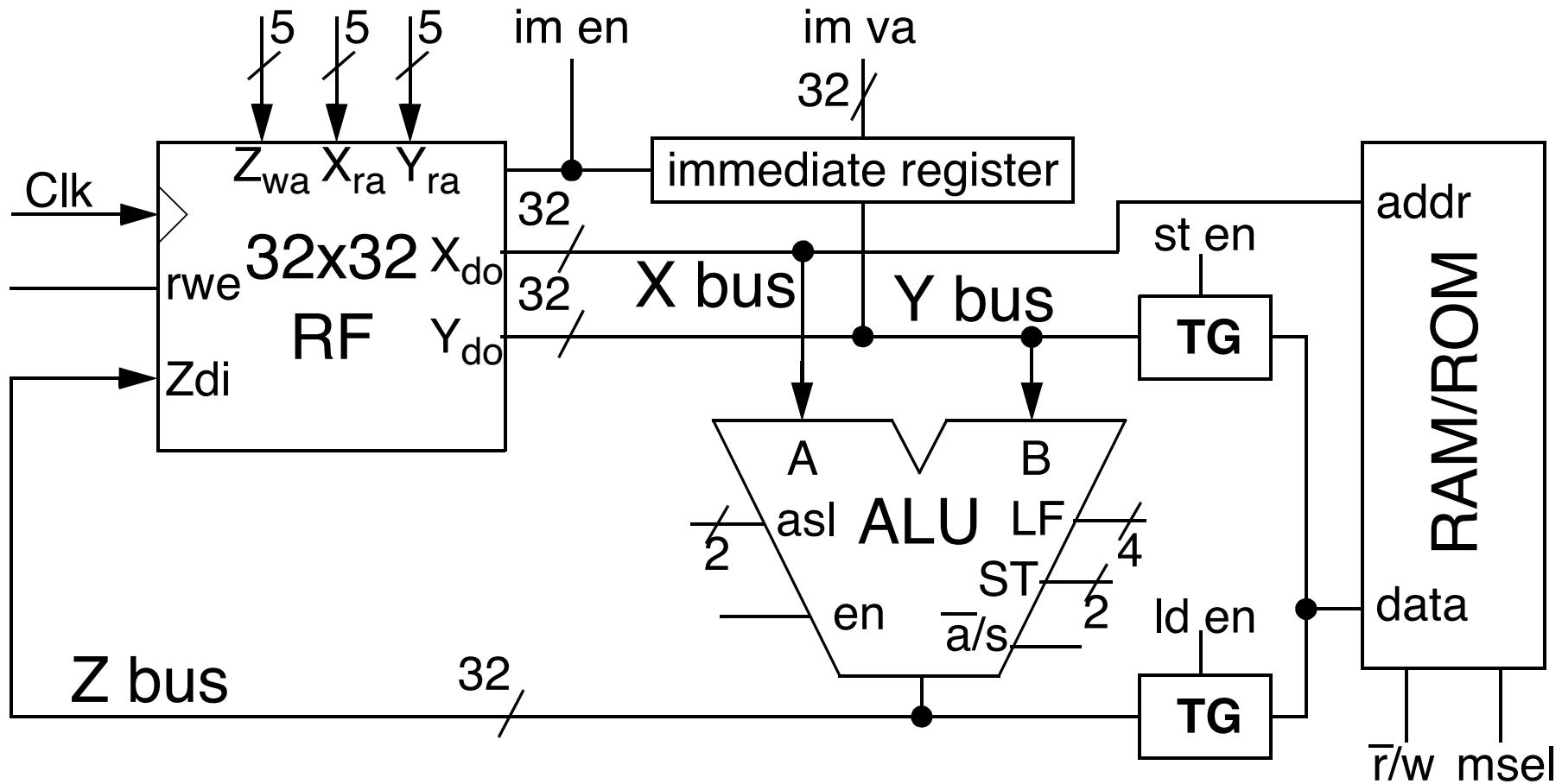
## **SINGLE CYCLE DATAPATH UNIT**

**READ SINGLE CYCLE DATAPATH FREE-DOC ON COURSE WEBPAGE**

# SINGLE CYCLE DPU

## INCLUDING MEMORY

- 32x32 bits is not sufficient memory for most computers.
- We can include external memory (SRAM, DRAM, etc.) as follows.



# REGISTER FILES

32-BIT WORD, 32 REGISTERS

- For the upcoming datapath designs in the next chapter, we want to have a 32x32 register file with one write input and two read outputs.

$X_{ra}$  - X read address

$Y_{ra}$  - Y read address

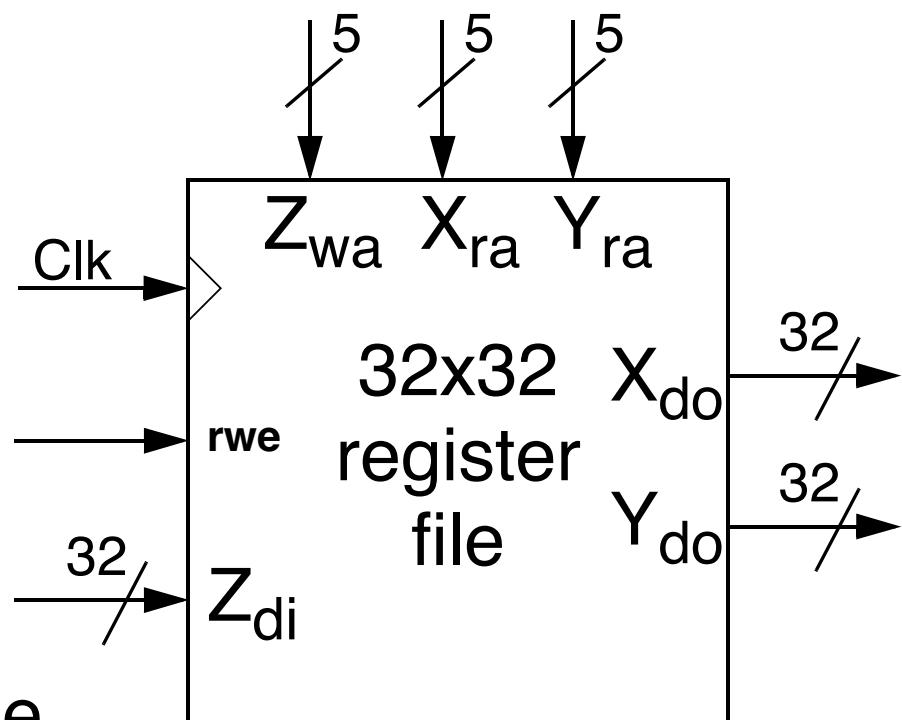
$Z_{wa}$  - Z write address

$X_{do}$  - X data out

$Y_{do}$  - Y data out

$Z_{di}$  - Z data in

rwe - register write enable

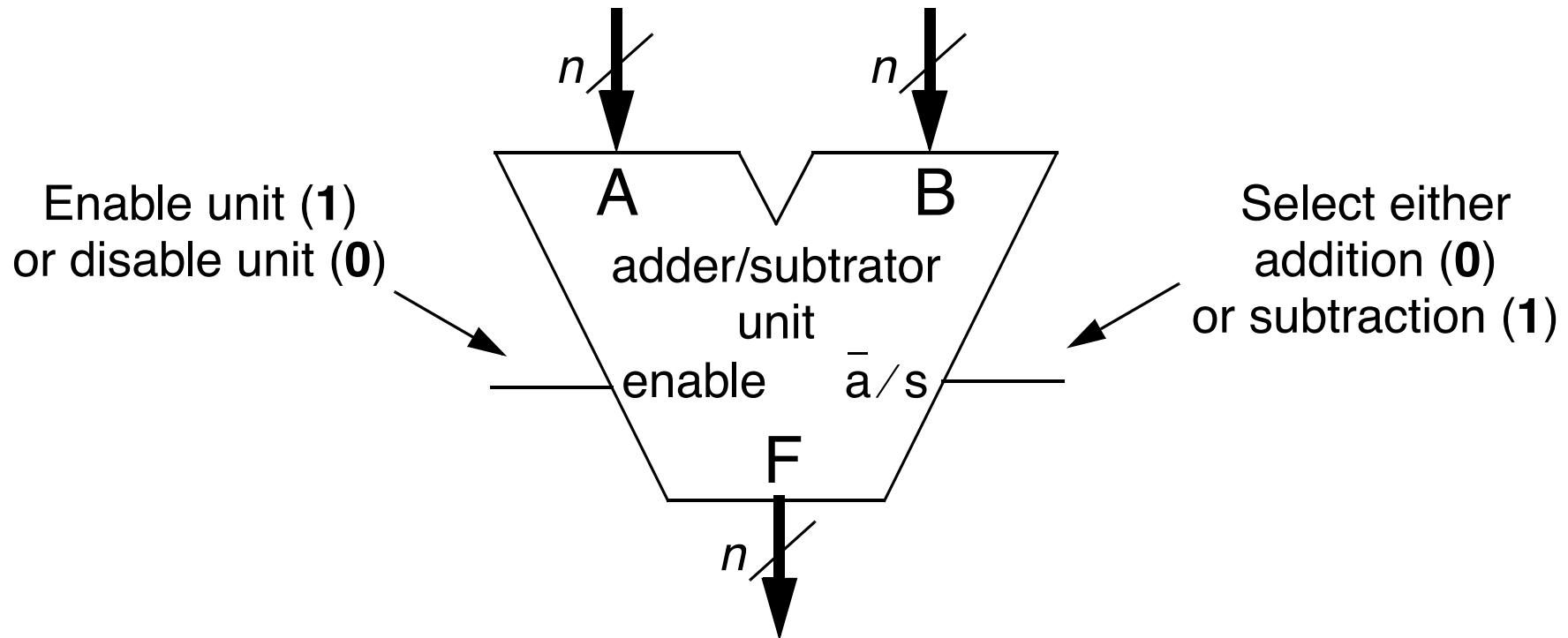


- Note: Two data outputs implemented with two read address decoders.

# ADDER/SUBTRACTOR

## GENERAL UNIT DIAGRAM

- An  $n$ -bit adder/subtractor unit is often illustrated as follows.



- This unit would have  $n$  full-adders internally.

- A useful unit would be one that can take two  $n$ -bit inputs and perform some logical operation between each of the bits to get an  $n$ -bit output.
  - For example, given the 8-bit values 0001 1110 and 1001 1000, we might want to find the **bit-wise logical OR**.

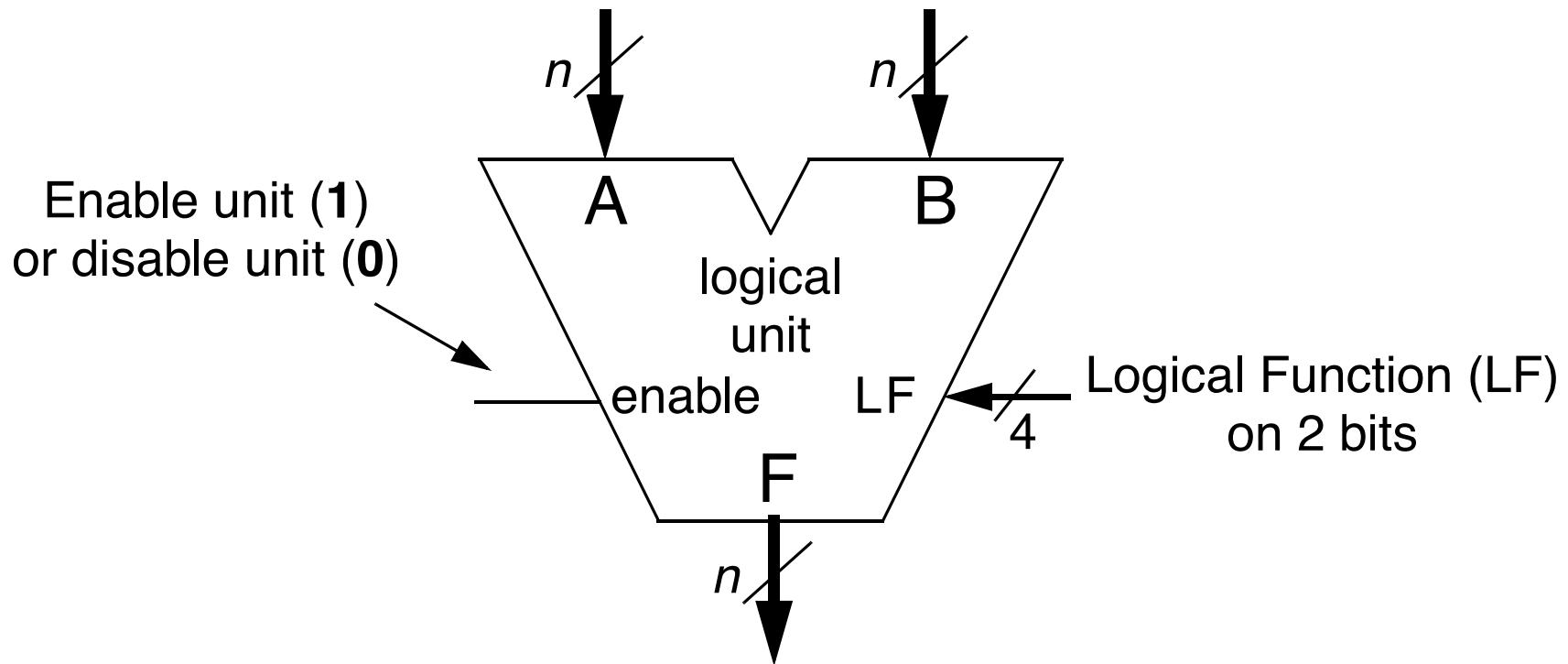
<b>bit-wise</b>	0001 1110
<b>logical OR</b>	1001 1000
1001 1110	

- Or similarly, the **bit-wise logical AND** of the two 8-bit values.

<b>bit-wise</b>	0001 1110
<b>logical AND</b>	1001 1000
0001 1000	

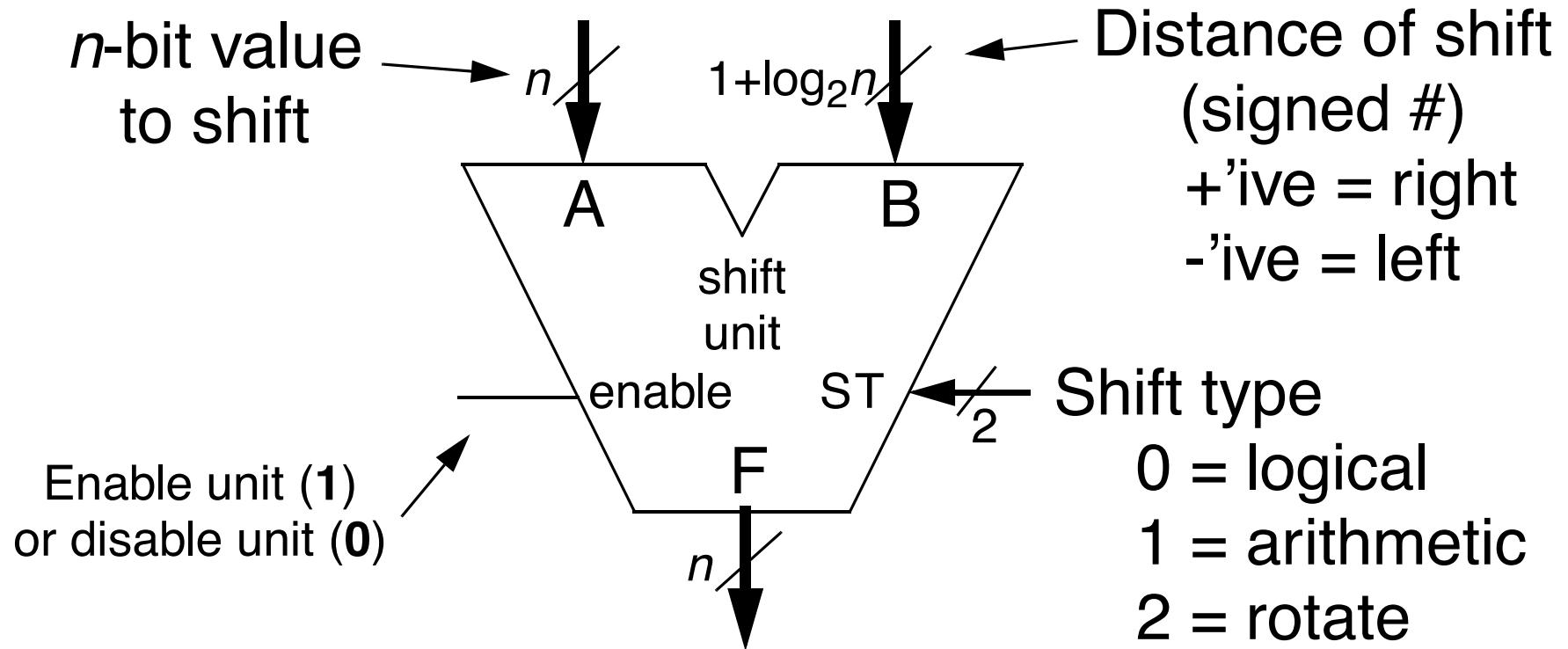
- These types of operations are often used for masking and setting bits.

- Below is a general unit diagram for an  $n$ -bit logical unit.



- Logical operations, such as **AND/OR/NOT/NAND/NOR/etc.**, are done for each bit of **A** and **B** to form **F**.

- Below is a general unit diagram for an  $n$ -bit shift unit.

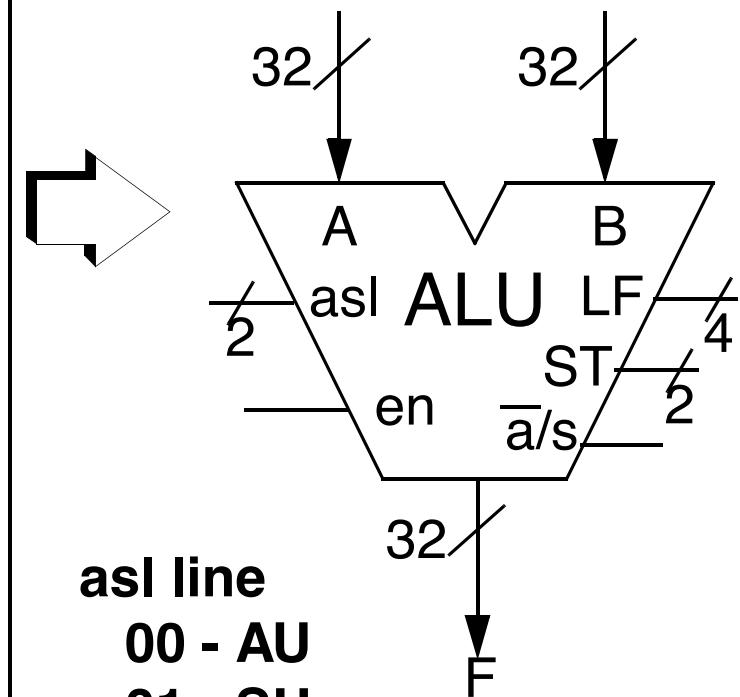
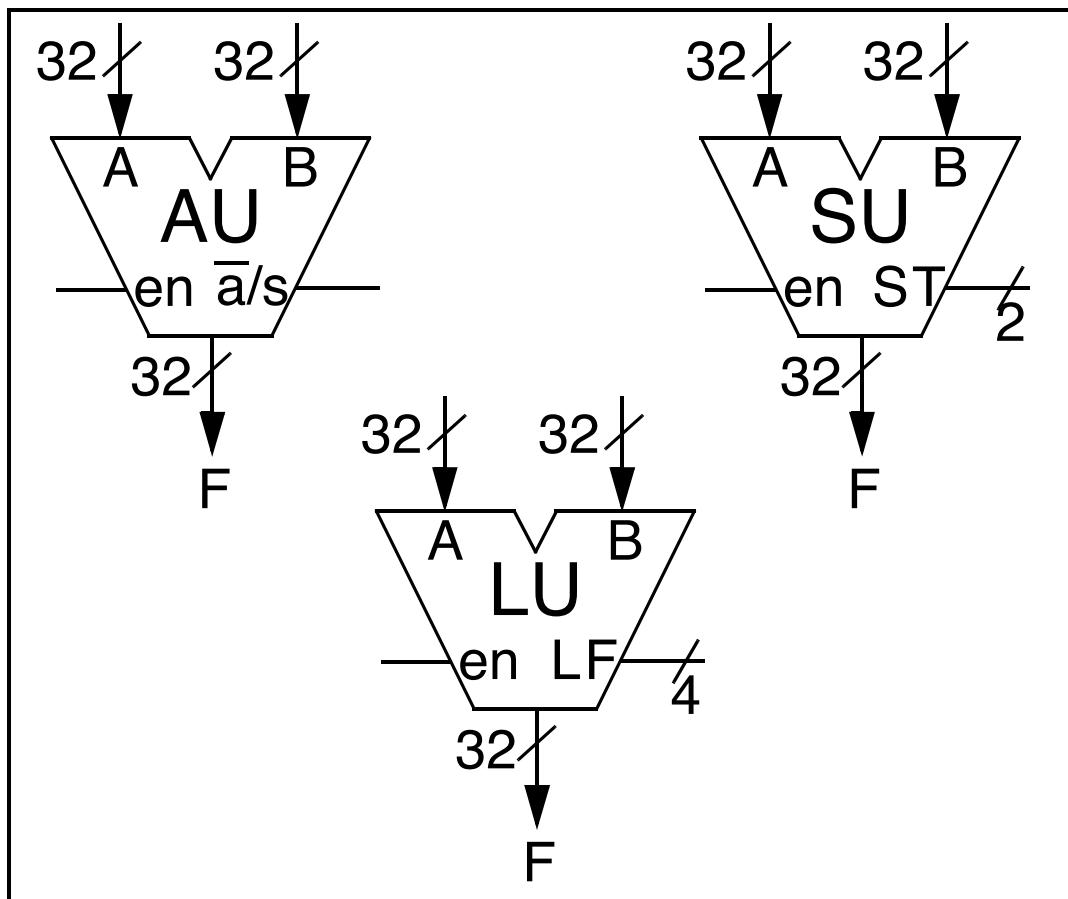


- Notice that the  $n$ -bit value **A** will be shifted according to the distance indicated with signed number **B**.

# SINGLE CYCLE DPU

## ARITHMETIC LOGIC UNIT

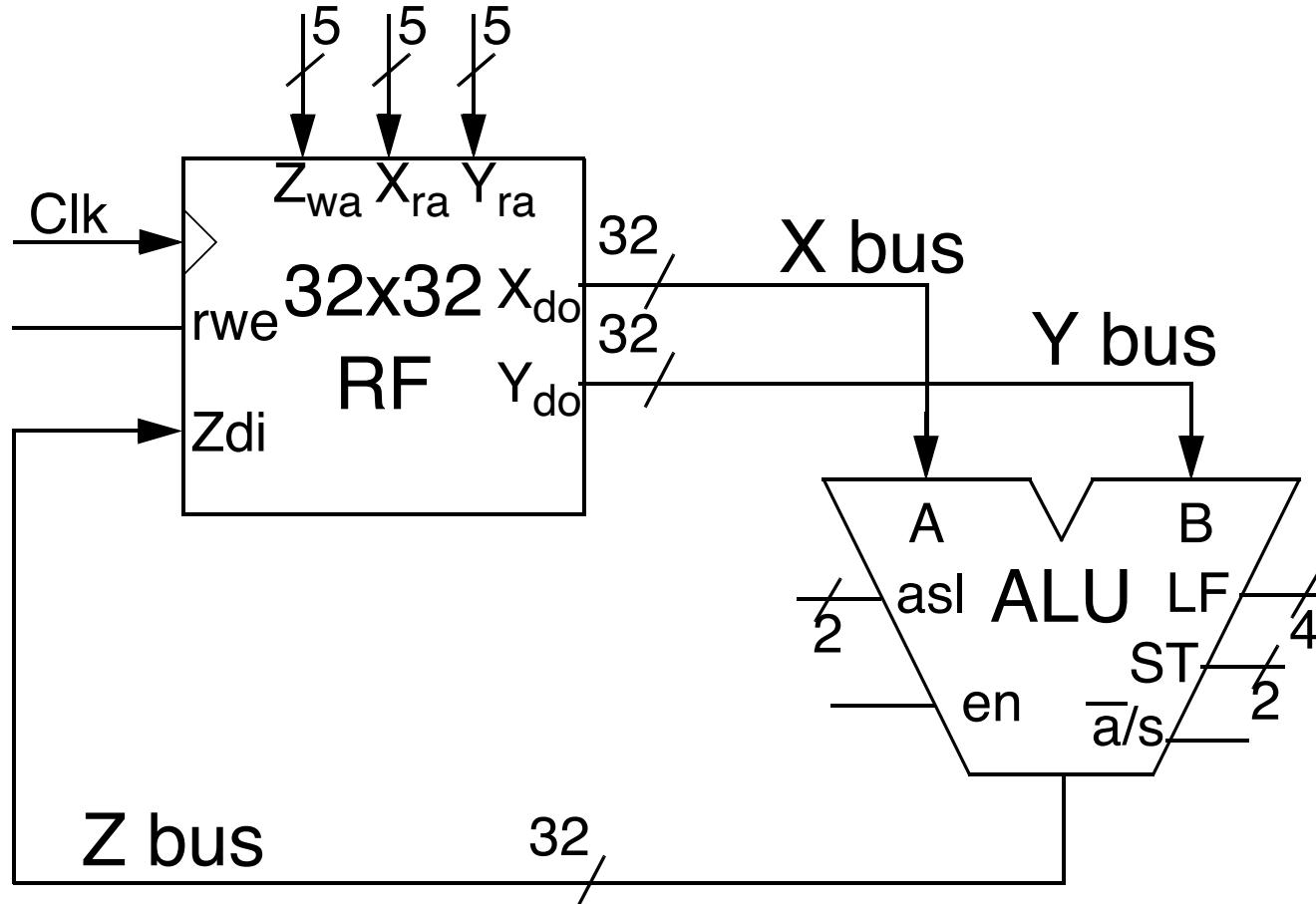
- Since only one of **AU**, **SU**, or **LU** will be active at a time in this architecture, we will combine to form an **arithmetic logic unit** (ALU).



# SINGLE CYCLE DPU

## SINGLE CYCLE DPU W/ALU

- Using our **ALU**, the DPU can be redrawn as follows.



- This structure is still a **triple bus internal DPU architecture**.

- The **im\_en** line does two things:
  - When 0, **im\_en** controls
    - **immediate register outputs** to go to **high impedance** so as **NOT** to affect **Y bus**.
    - **register file Y data out** to output corresponding register value.
  - When 1, **im\_en** controls
    - **immediate register** to output register value to **Y bus**.
    - **register file Y data out** to go to **high impedance** so as **NOT** to affect **Y bus**.
- The **im\_va** lines pass a value to the immediate register.

- Microcode in a processor are all of the control signals required to execute an operation for a clock cycle.
- We have actually looked at examples of a microcode operation when we considered various operations such as

$$\mathbf{R3} = \mathbf{R1} + \mathbf{R2}$$

or

$$\mathbf{M[R5]} = \mathbf{R9}$$

- Later we will talk about macrocode which are longer operations consisting of many microcode operation over a number of clock cycles.

# SINGLE CYCLE DPU

## READING FROM MEMORY

- We wish to be able to read and write from our memory.
- A sample read/load operation can be expressed as follows

$$R4 = M[R7]$$

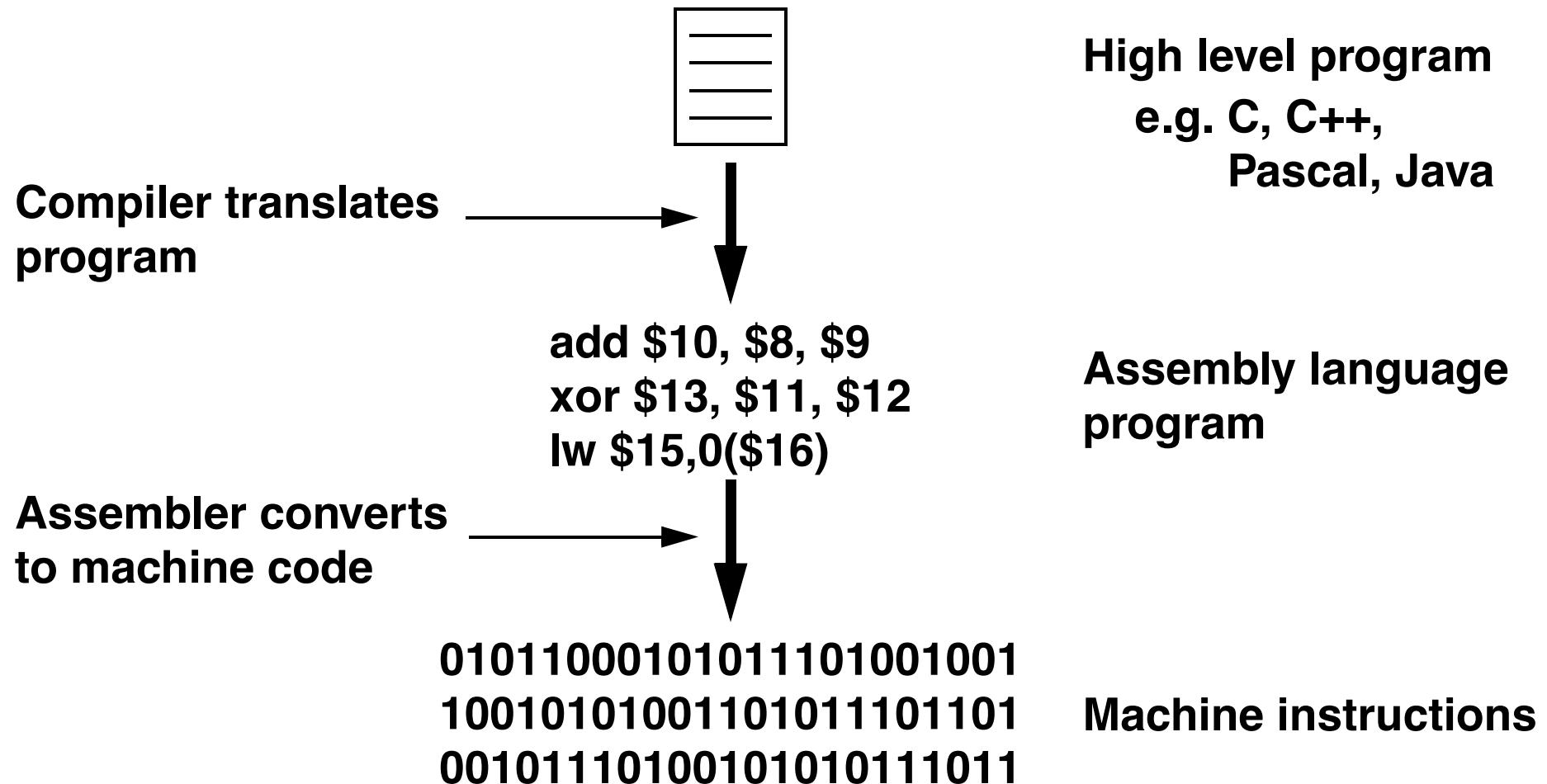
- This operation uses the value in **R7** as the address to the memory and reads the value at that address in the memory to **R4**.
- What control signals are required?
  - **en** = 0 for **ALU**.
  - **X<sub>ra</sub>** = 00111, **Y<sub>ra</sub>** = XXXXX, **Z<sub>wa</sub>** = 00100, and **rwe** = 1 for **RF**.
  - **st\_en** = 0 and **ld\_en** = 1
  - **~r/w** = r and **msel** = 1

- A sample write/store operation can be expressed as follows

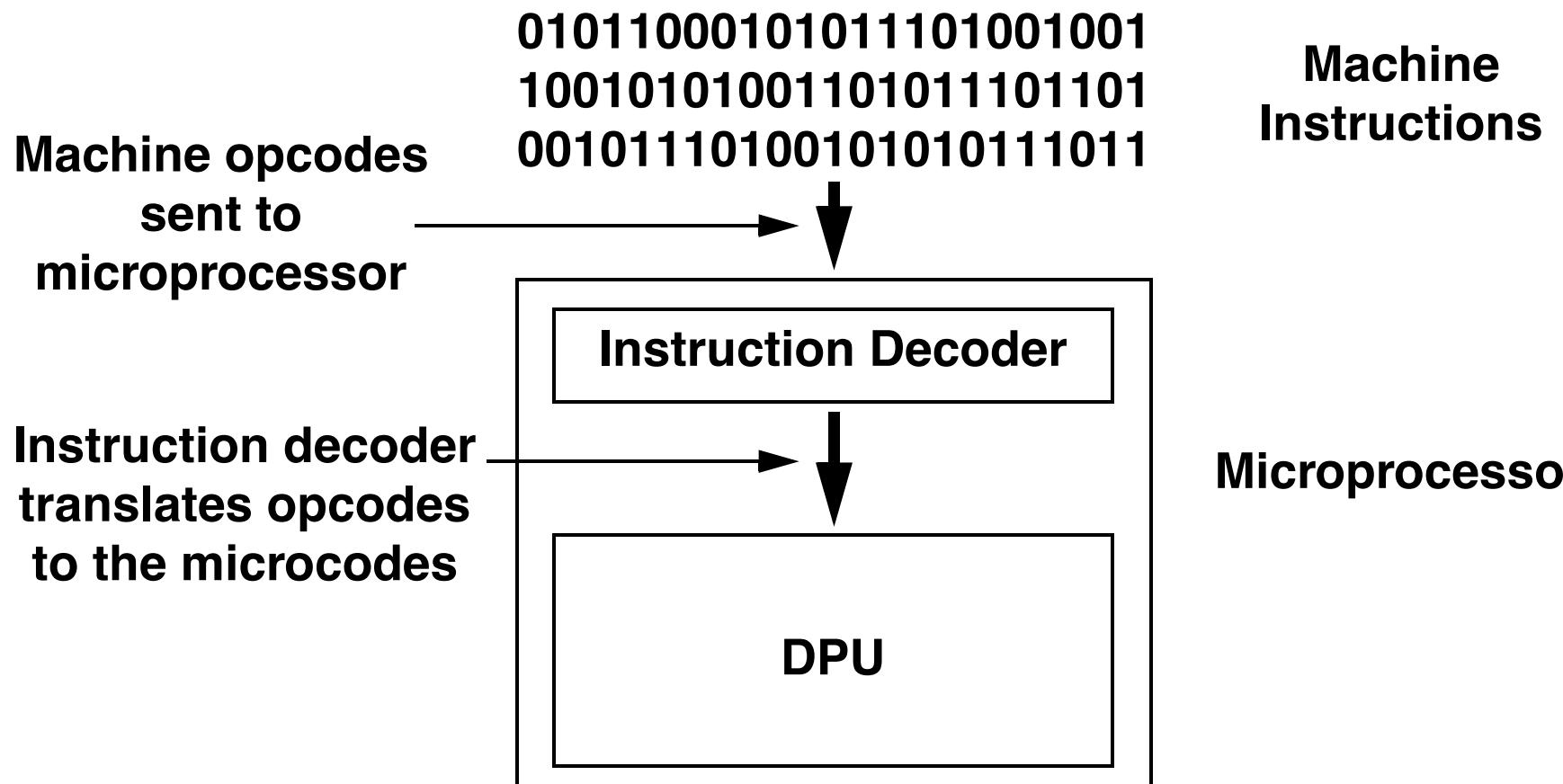
$$M[R5] = R9$$

- This operation uses the value in **R5** as the address to the memory and write the value in **R9** to that address in the memory.
- What control signals are required?
  - **en = 0** for **ALU**.
  - **X<sub>ra</sub> = 00101**, **Y<sub>ra</sub> = 01001**, **Z<sub>wa</sub> = XXXXX**, and **rwe = 0** for **RF**.
  - **st\_en = 1** and **Id\_en = 0**
  - **~r/w = w** and **msel = 1**

- Below is the process for translating a program to machine opcodes.



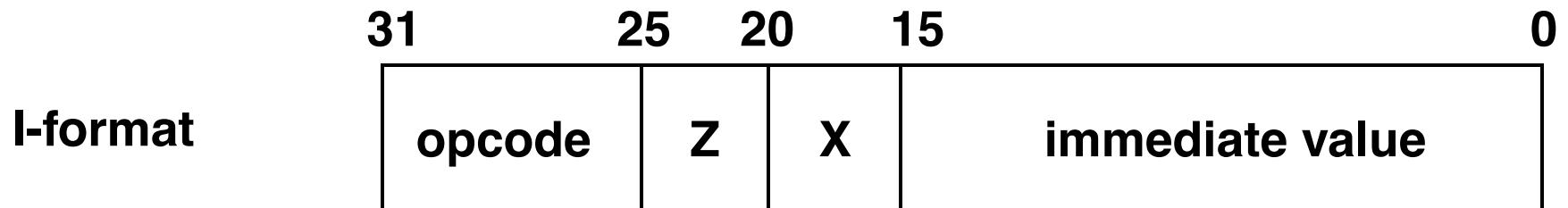
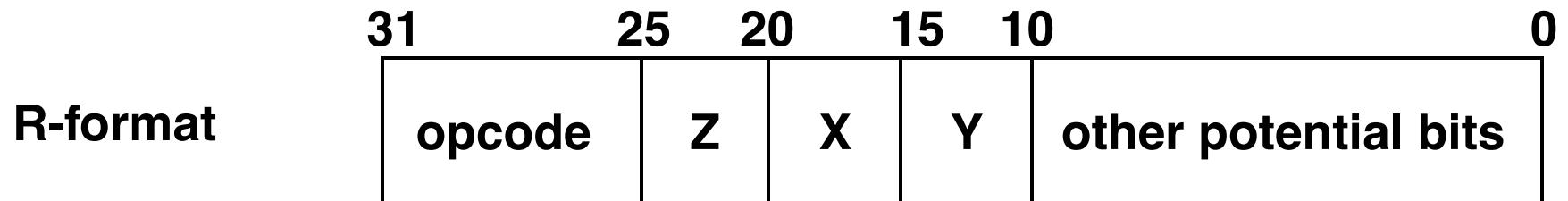
- Once the opcodes are given to the microprocessor, it translates the opcode instructions to the microcodes operations we discussed.



# INSTRUCTIONS

## INSTRUCTION FORMATS

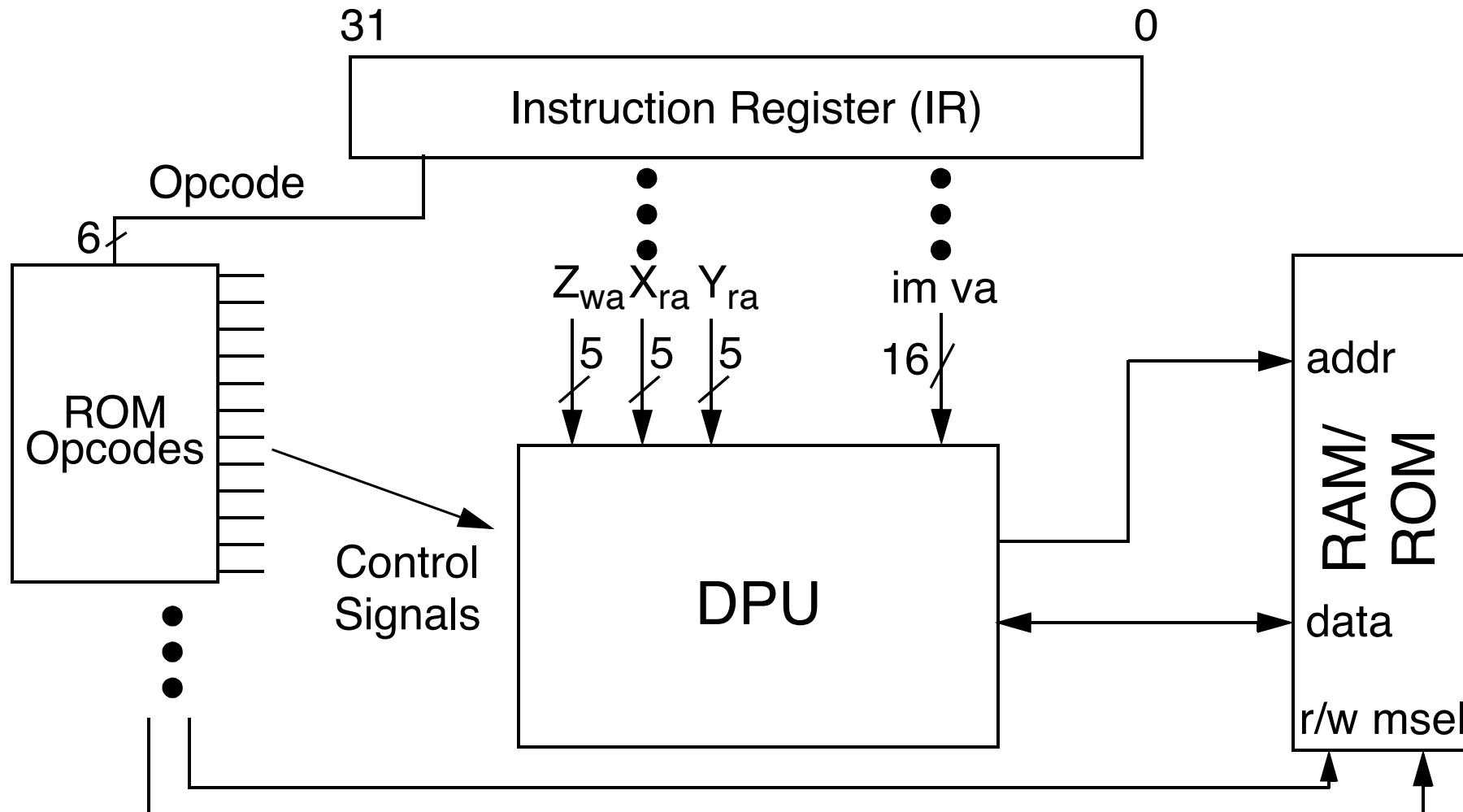
- While instructions can come in many different shapes and forms, we will consider the following 32-bit instruction formats to loosely follow the MIPS R3000/4000 format.



# INSTRUCTIONS

## INSTRUCTION REGISTER

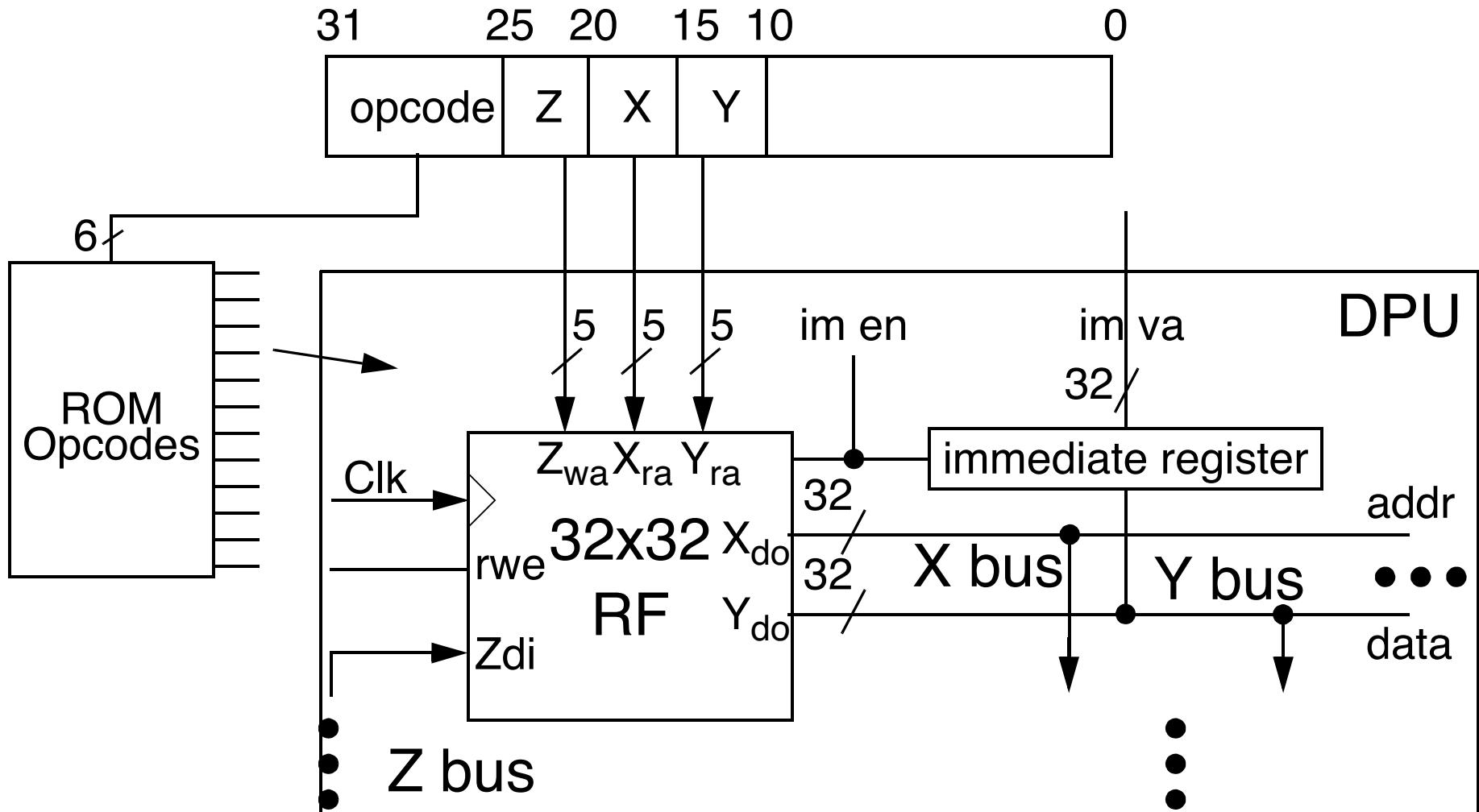
- Use a general instruction register that can act as R- or I-Format.



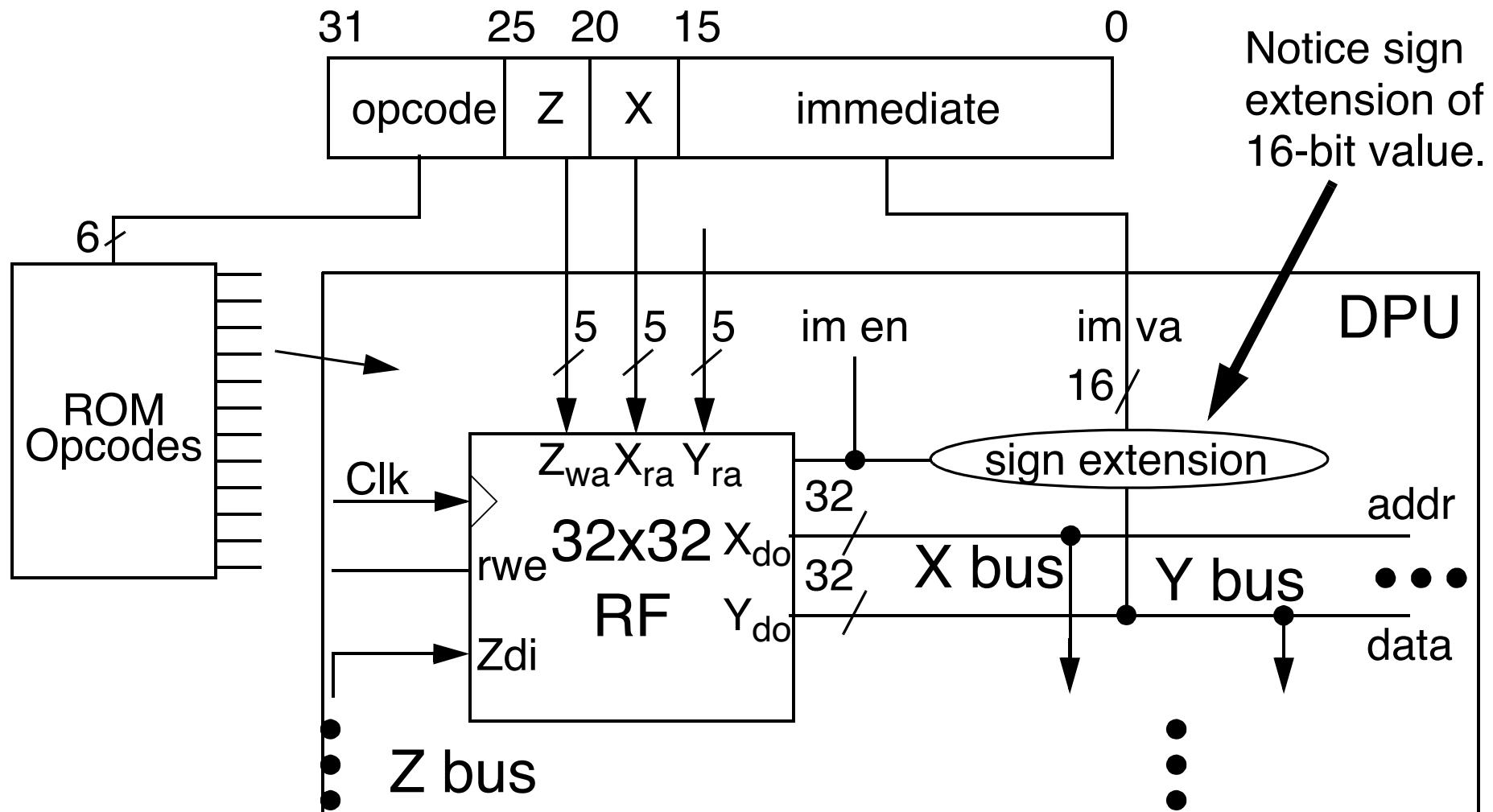
# INSTRUCTIONS

## R-FORMAT W/ DPU

- If we have an R-format instruction, we link the bits as follows.



- If we have an I-format instruction, we link the bits as follows.



**INTRO. TO COMP. ENG.  
CHAPTER IX-1**

**REGISTER BLOCKS**

**•CHAPTER IX**

# **CHAPTER IX**

## **REGISTER BLOCKS COUNTERS, SHIFT, AND ROTATE REGISTERS**

**READ PAGES 249-275 FROM MANO AND KIME**

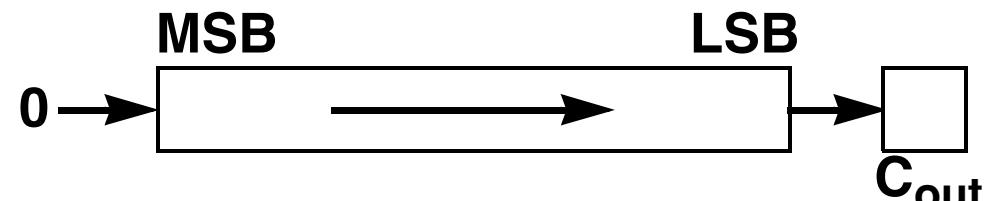
# REGISTER BLOCKS

## INTRODUCTION

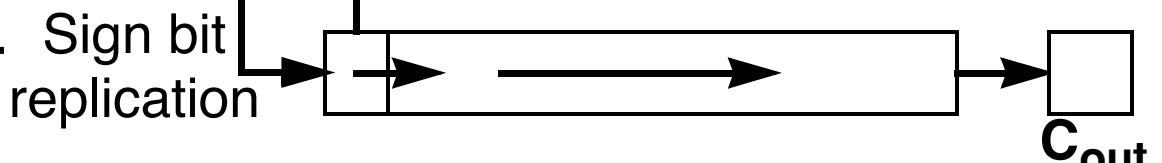
- Like combinational building blocks, we can also develop some simple building blocks using registers. These include:
  - Shift registers
  - Rotate registers
  - Counters
- Implementations of these components can use state machines, but, it is often easier to think of them without the complication of a state machine.

- Logical shift registers take the bits stored and move them up a significant bit or down a significant bit.

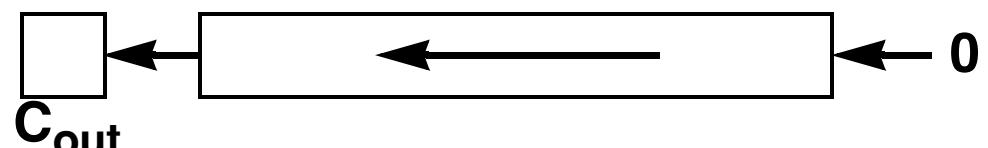
**LOGICAL SHIFT RIGHT  
(LSR)**



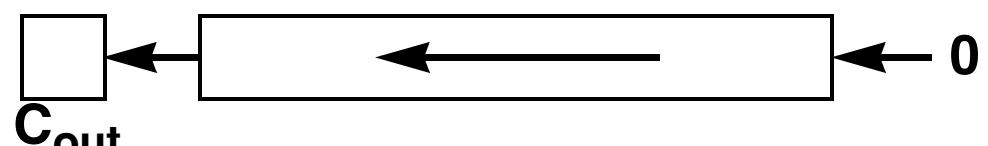
**ARITHMETIC SHIFT RIGHT  
(ASR)**



**LOGICAL SHIFT LEFT  
(LSL)**

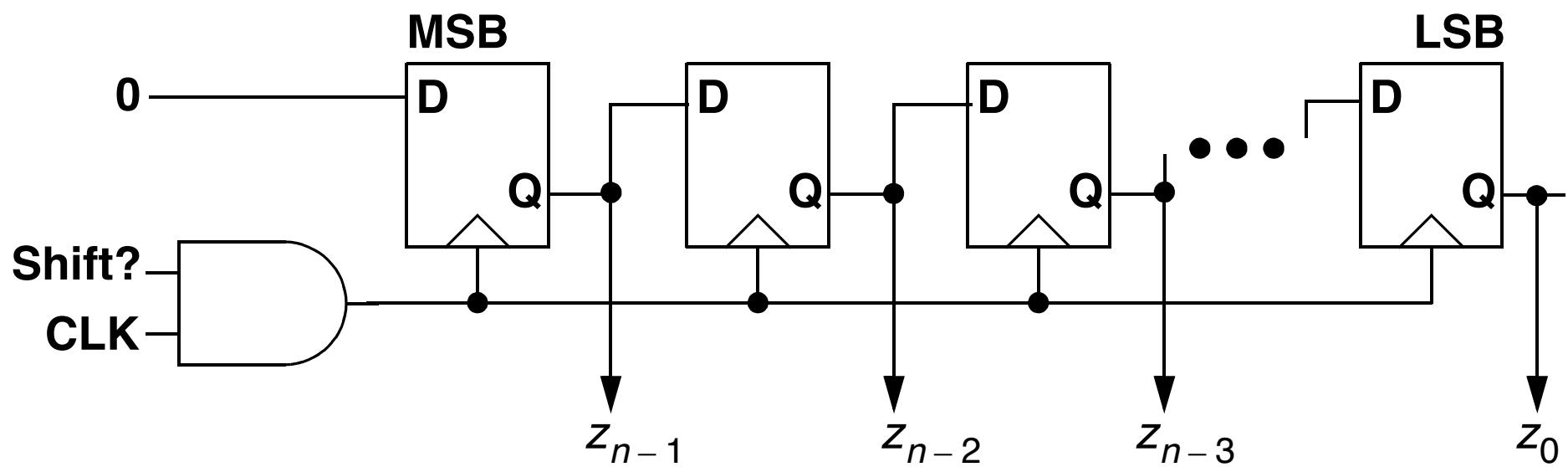


**ARITHMETIC SHIFT LEFT  
(ASL)**

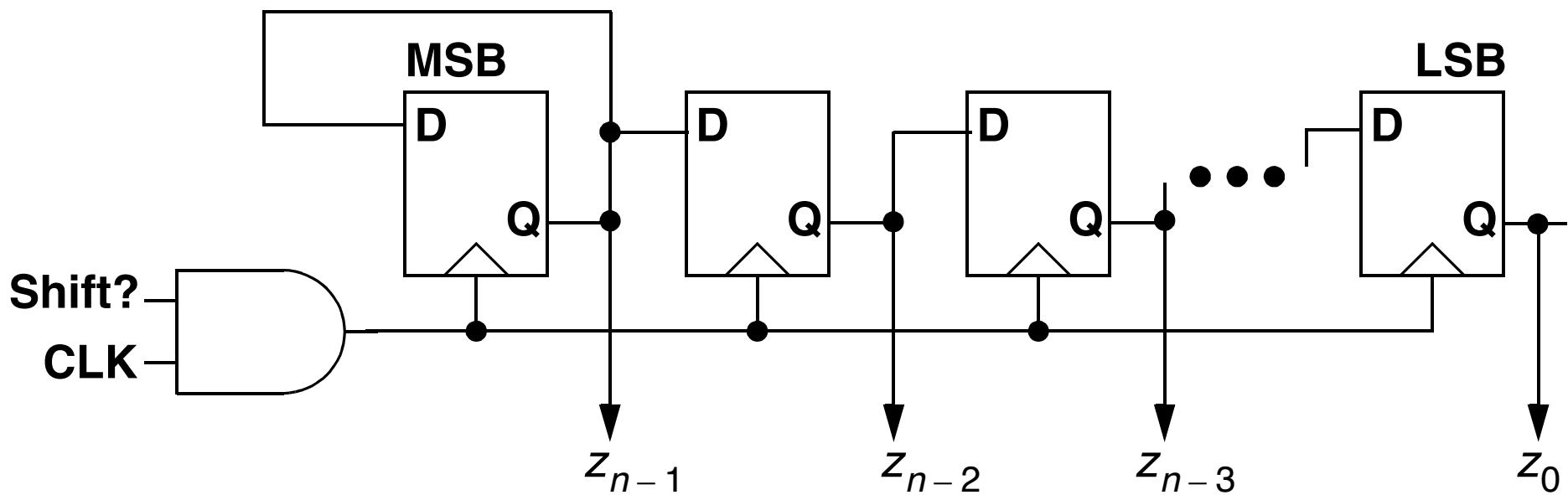


- Notice that logical and arithmetic shift lefts are the same.

- A simple implementation of a logical right shift register might look like the following.



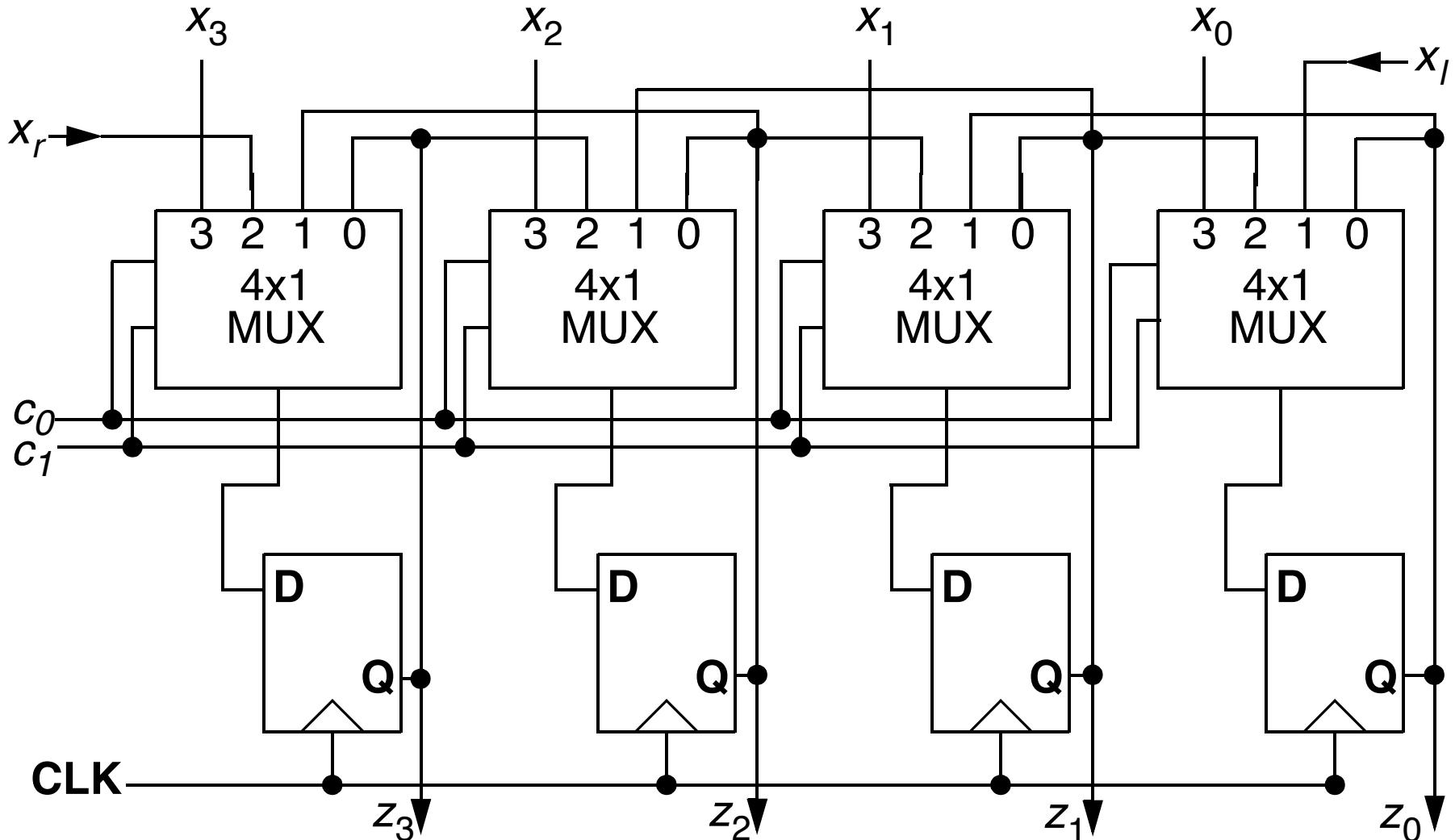
- An arithmetic right shift register might look like the following.



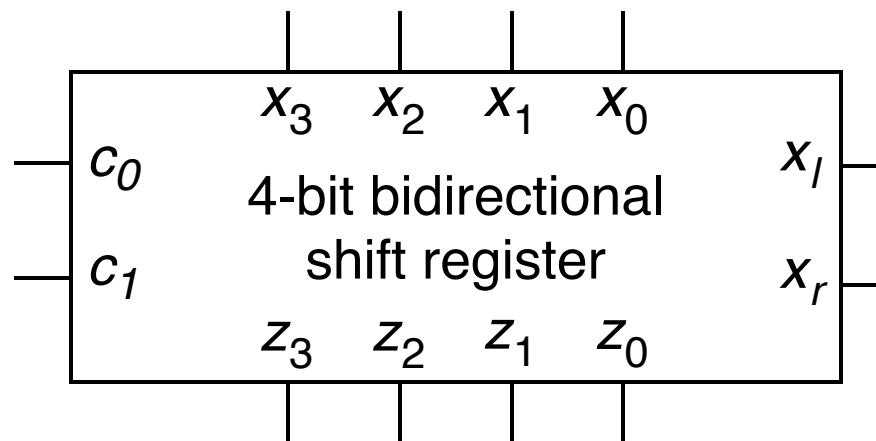
# SHIFT REGISTERS

## 4-BIT BIDIRECTIONAL

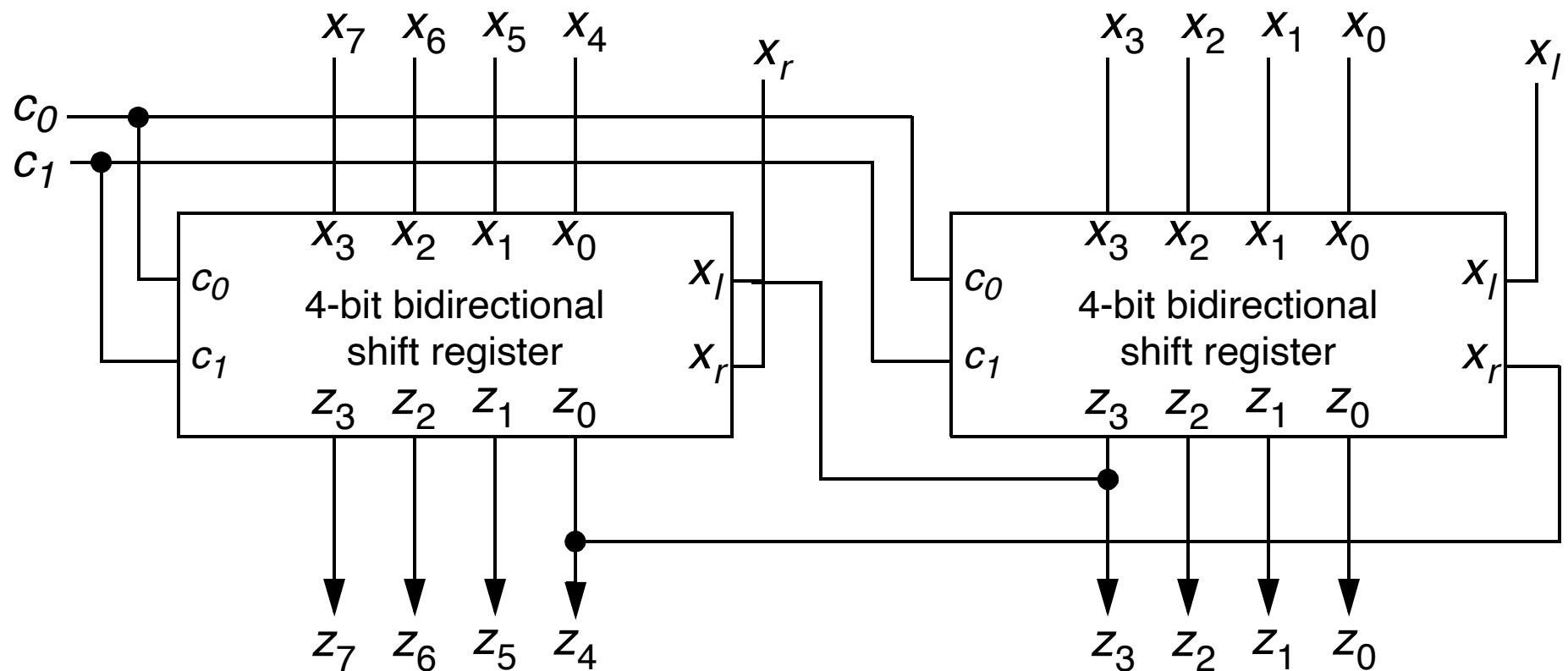
- The following is a 4-bit bidirectional shift register with parallel load.



- Cascading of shift registers can also be done if the discarded bit is used to shift into another shift register module.
- For instance, the 4-bit bidirectional shift register previously presented can be easily cascaded using the
  - $x_r$  (right shift data input) and
  - $x_l$  (left shift data input)

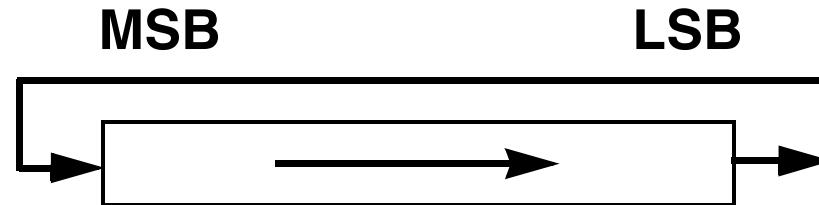


- For example, an 8-bit bidirectional shift register with parallel load can be formed as follows.

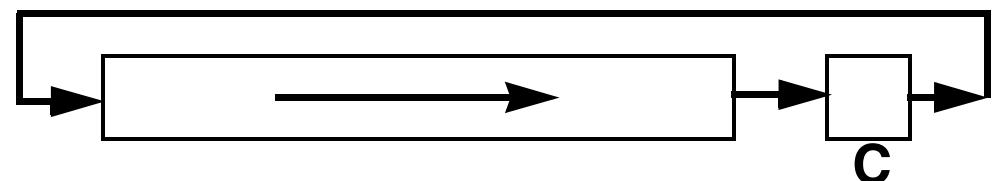


- A rotate register is the same as a logical shift register except that the discarded bit is fed back into the empty space from the shift.

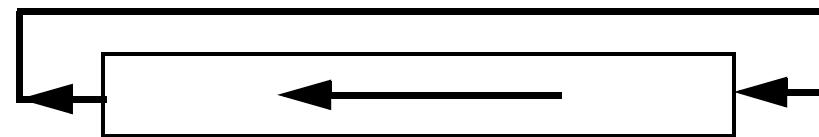
**ROTATE RIGHT**



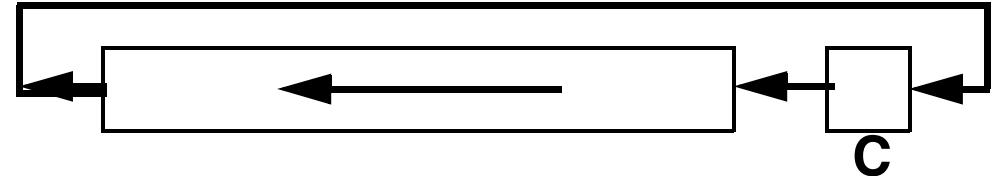
**ROTATE RIGHT WITH CARRY**



**ROTATE LEFT**

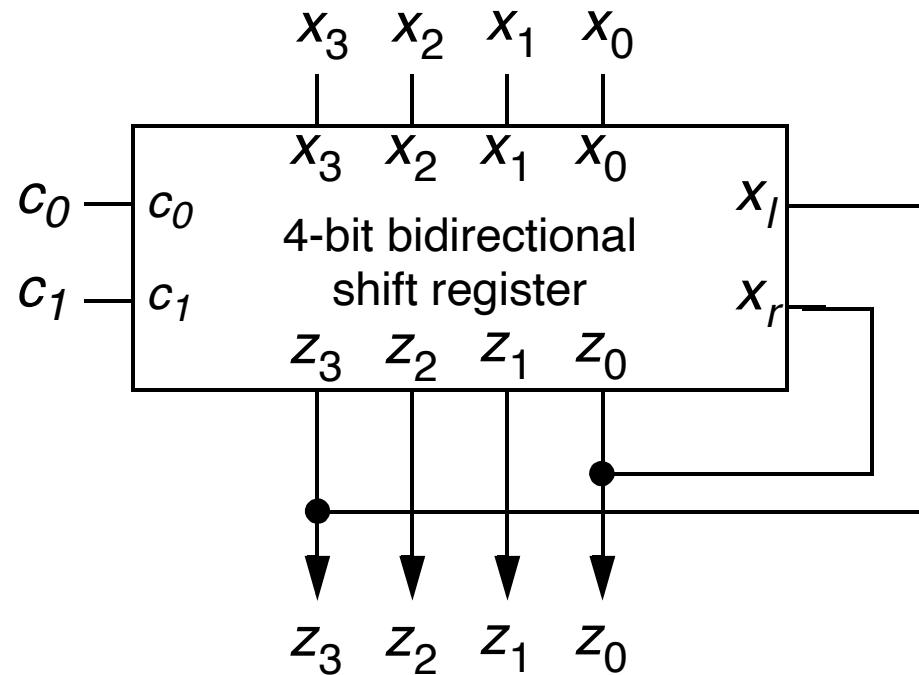


**ROTATE LEFT WITH CARRY**



# ROTATE REGISTERS USING SHIFT REGISTERS

- Rotate registers can actually be implemented using shift registers that have serial data inputs (such as the 4-bit bidirectional shift register discussed).
- For example, a 4-bit rotate register can be formed as follows.

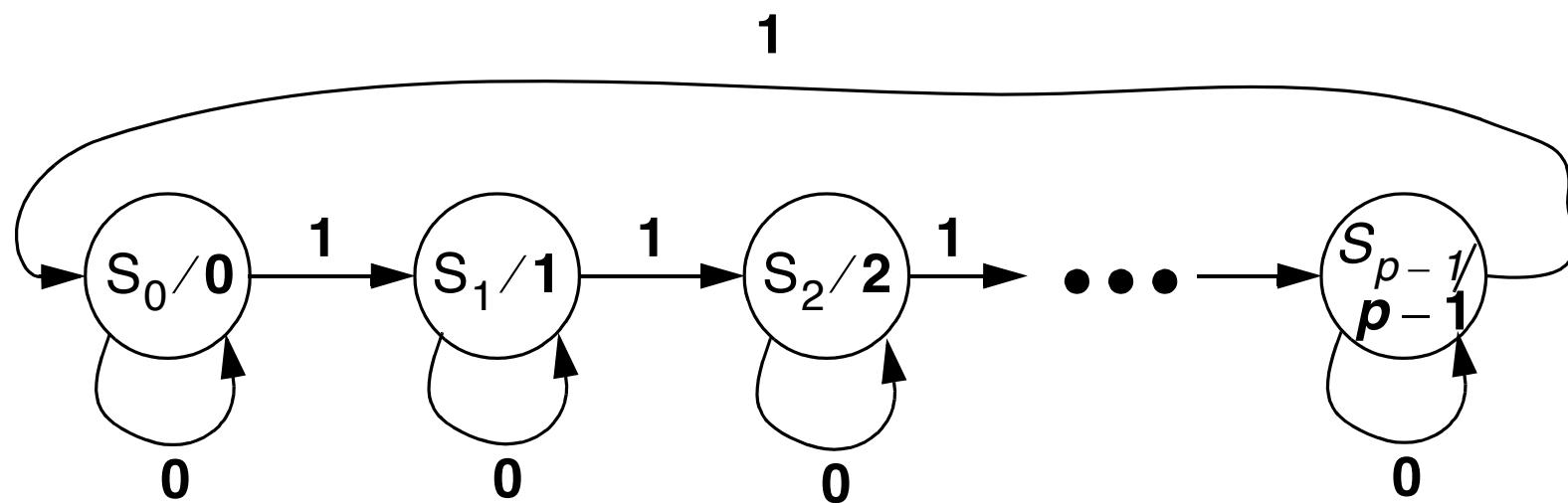


- A counter is a register that on each clock pulse counts up or down, usually in binary.
- Types of counters
  - ripple counters
  - synchronous counters
  - binary counters
  - BCD counters
  - Gray-code counters
  - Ring counters (a 1 moves in a ring from one flip-flop to the next)
  - up/down counters (ability to increment or decrement)
  - counters with a parallel load (load in starting value with parallel input)

- A modulo- $p$  counter is defined by the following equation.

$$S(t + 1) = (S(t) + x) \bmod p$$

- The state diagram for the modulo- $p$  counter is as follows.

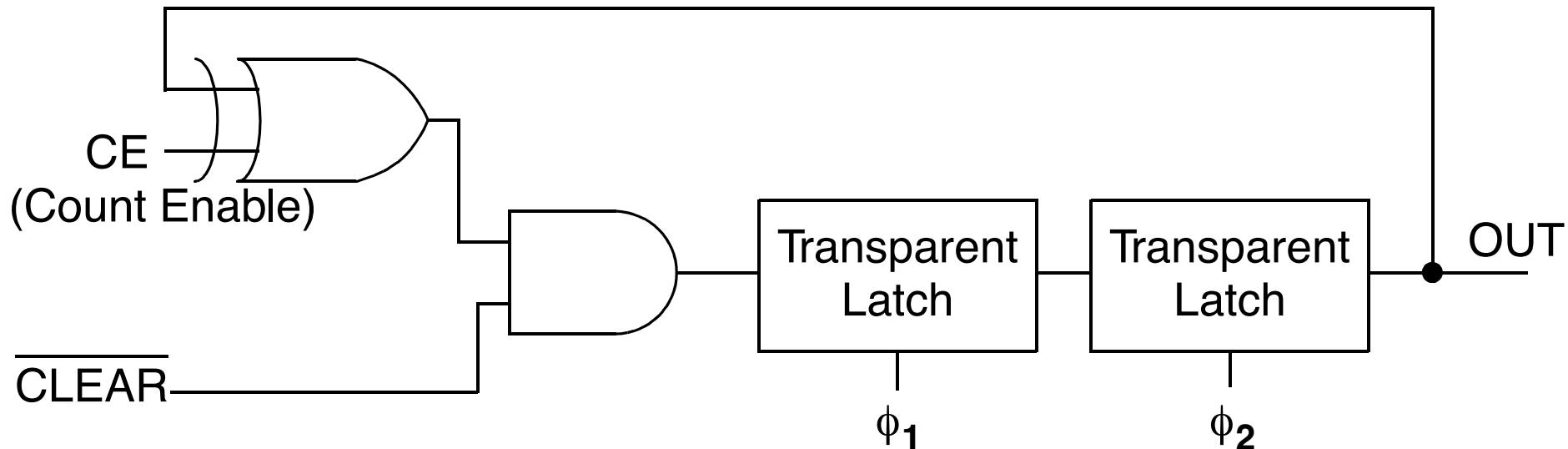


- An  $n$ -bit binary counter consists of  $n$  flip-flops and can count in binary from 0 through  $2^n - 1$ .
  - This can be formed with a modulo- $p$  counter where  $p = 2^n$ .
  - Two main categories exist for counters:
    - Ripple counters
      - One flip-flop transition serves to trigger other flip-flops.
      - The clock pulse is usually only sent to the first flip-flop.
      - This requires a memory cell that can complement its value.
      - The JK flip-flop would be one approach (we have not studied this!)
    - Synchronous counters
      - Change of state is determined from the present state.
      - Clock pulse sent to all flip-flops.

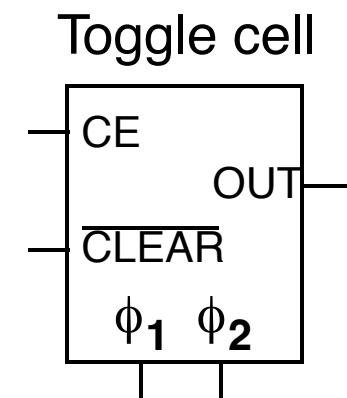
# COUNTERS

## TOGGLE CELL

- A toggle cell will be useful for implementing counters.



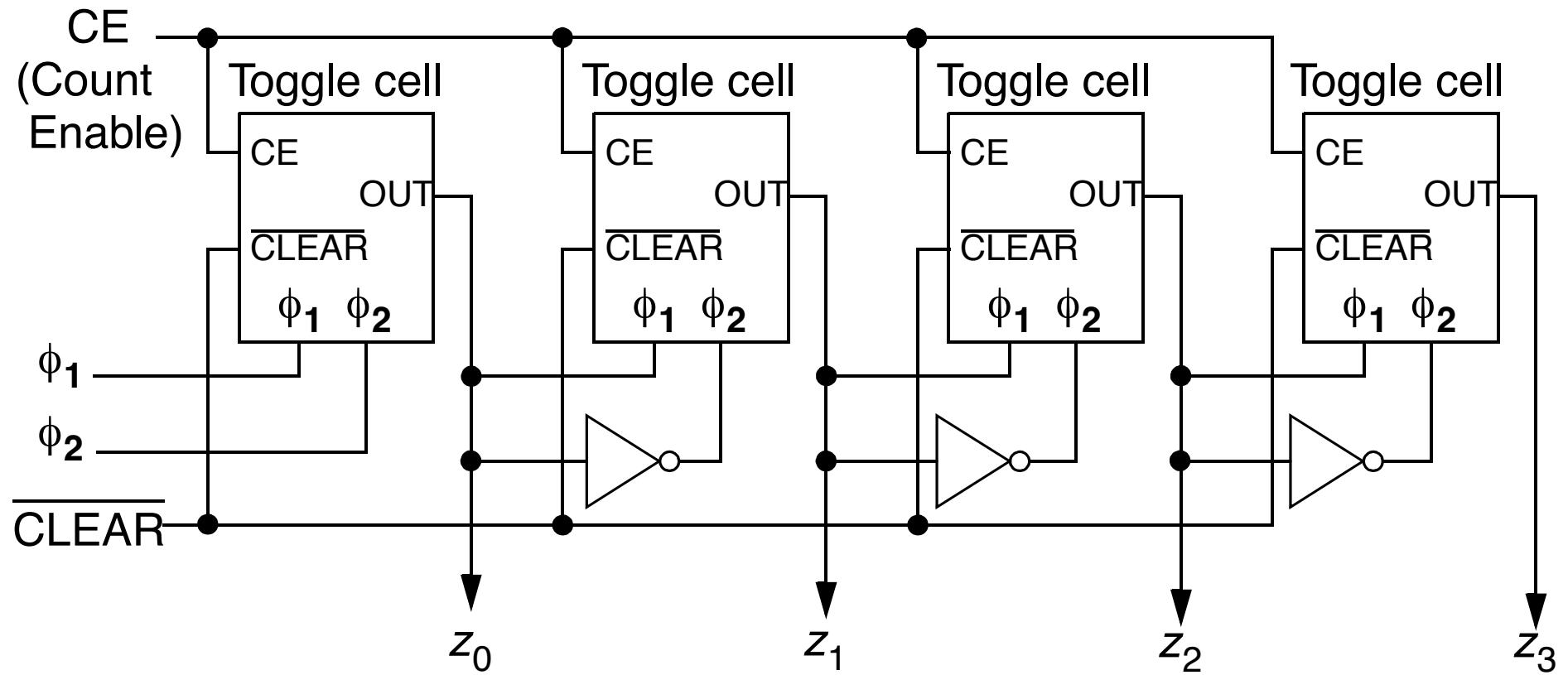
Present Latch Value	CE	CLEAR	Next Latch Value	OUT
X	X	0	0	?
0	0	1	0	0
1	0	1	1	1
0	1	1	1	0
1	1	1	0	1



# COUNTERS

## RIPPLE COUNTER

- The toggle cell can be used as follows to form a ripple counter.

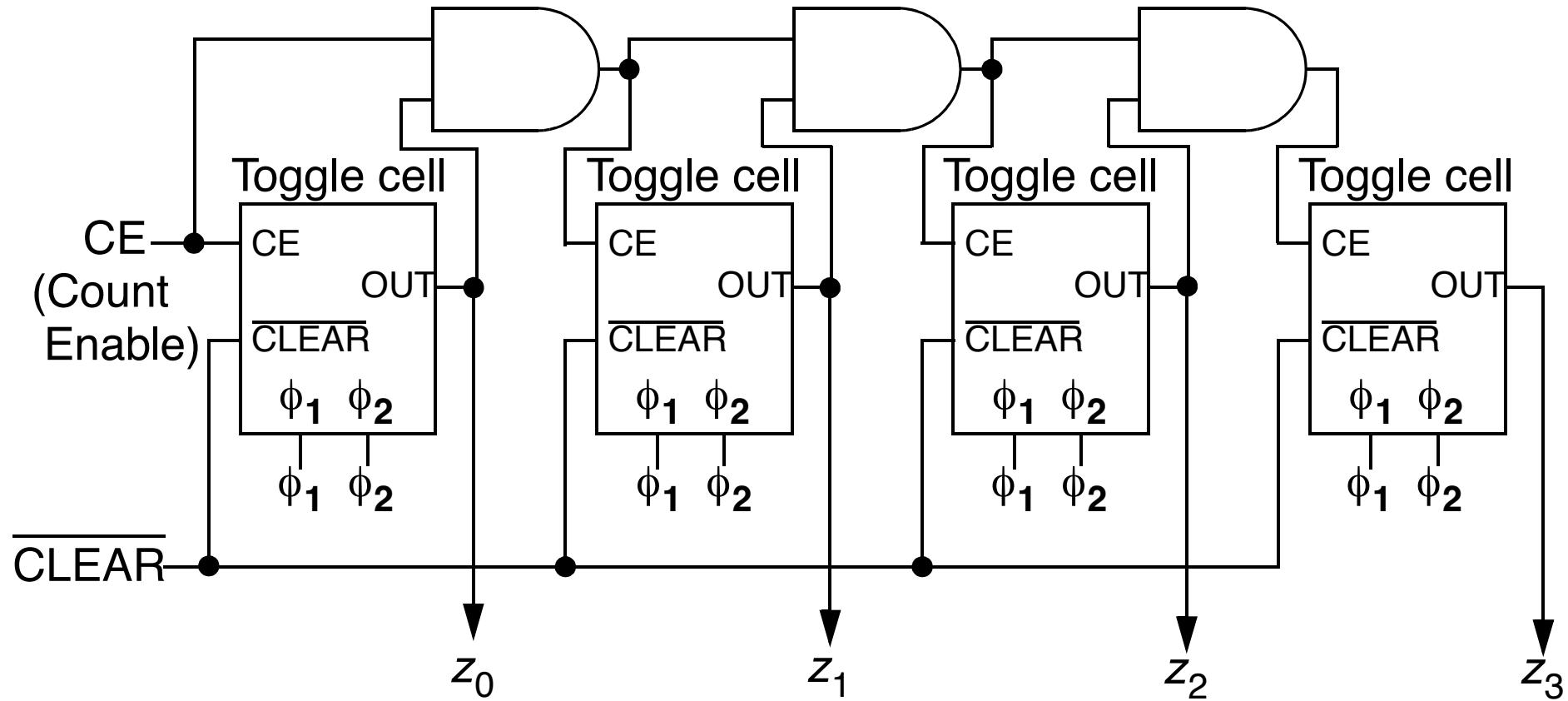


- Notice that the previous toggle cell is connected to the clock input of the next cell. This causes the bits to ripple through the counter.

# COUNTERS

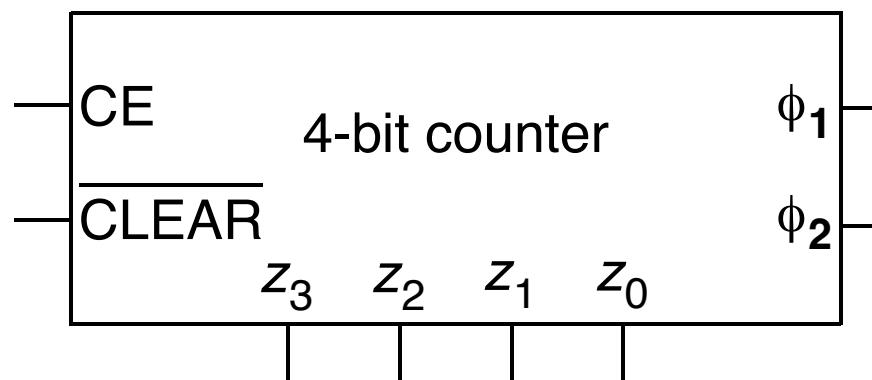
## SYNCHRONOUS COUNTER

- Below is an example 4-bit synchronous counter using toggle cells.

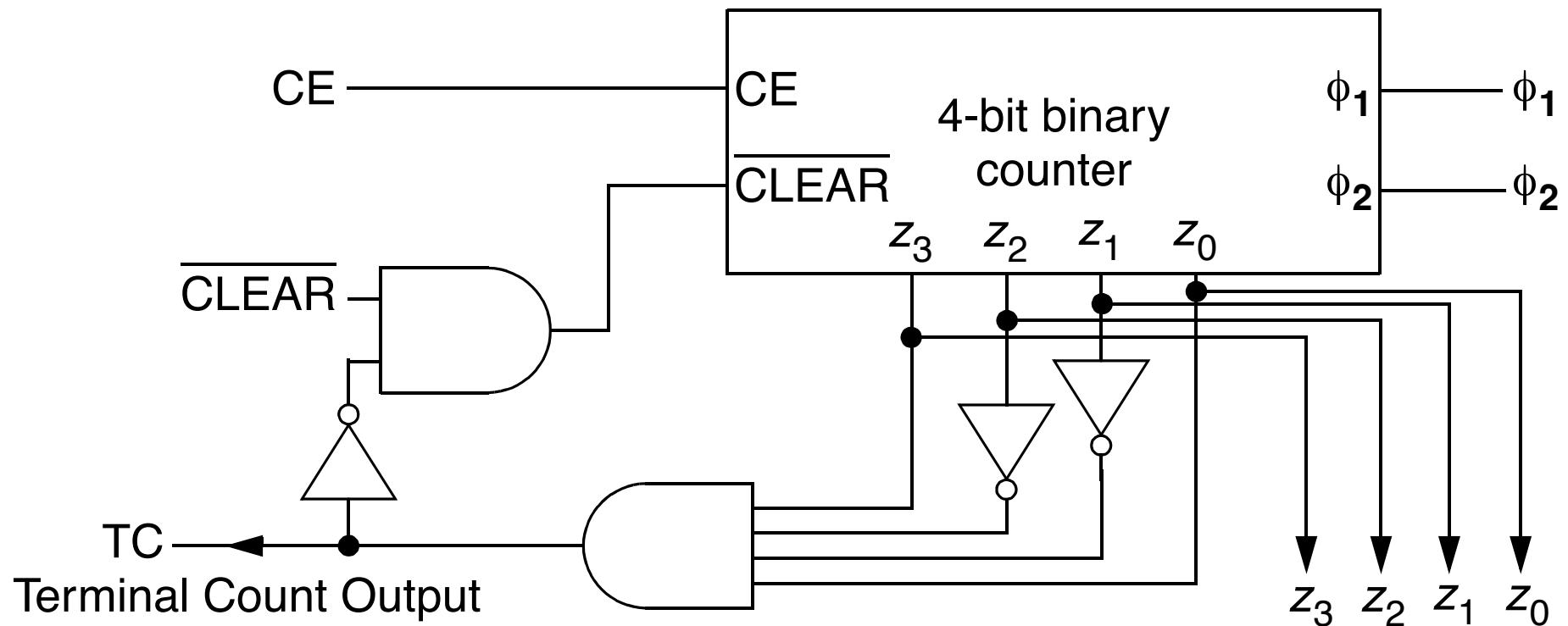


- Notice that clock is sent to all toggle cells.
- A simplified form is in Figure 5-11, pp. 269 of Mano & Kime.

- Notice that the counters developed so far can count from 0 to  $2^n - 1$  for  $n$  toggle cells.
  - Therefore, for module- $p$  counting, the  $p$  is currently limited to  $2^n$ .
  - How about if we wish  $p$  to be a non-power of 2?
    - Need to build what can be referred to as a divide by counter.
    - Given the following counter block, a general modulo- $p$  counter can be constructed by clearing the counter after the desired maximum value.

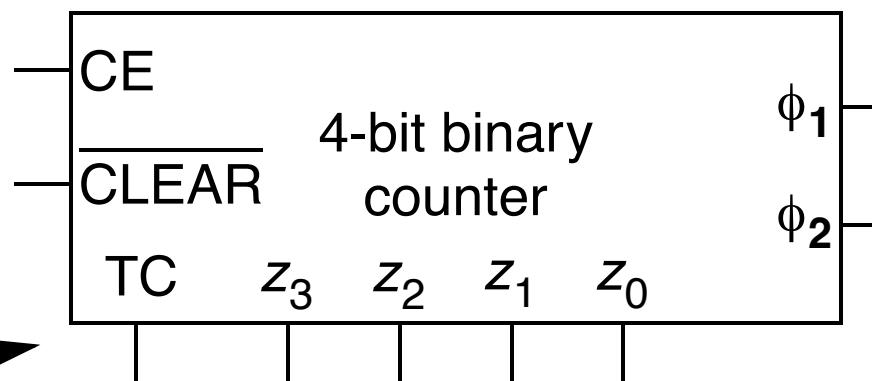


- To illustrate general modulo- $p$  counters, consider the following implementation of a single digit decimal counter using BCD.



- Notice that the counter is cleared after a value of 9 (**1001**).

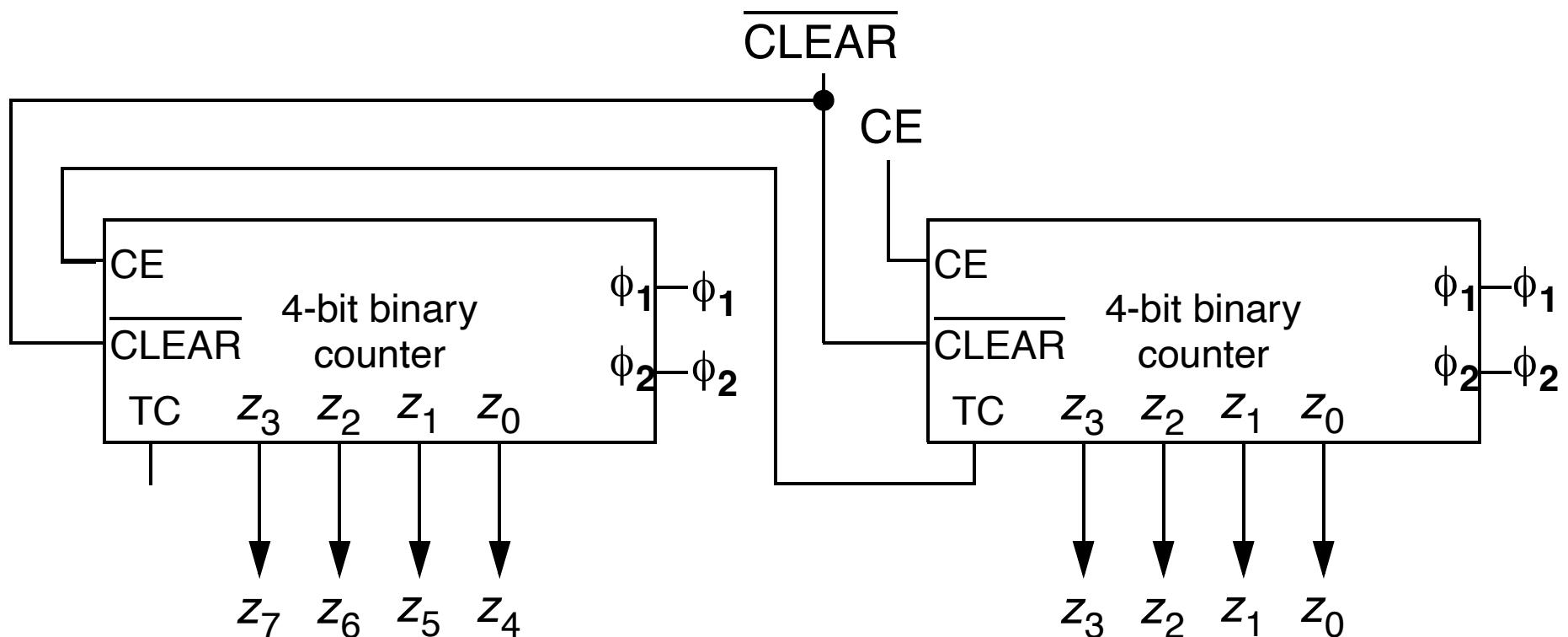
- The previous BCD counter was built by deriving a terminal count (TC) output signal.
- A terminal count output signal for any counter can be useful, so, we will be included in general block diagram for a binary counter.



Notice TC output

- In this 4-bit binary counter example,  $TC=1$  only when the output is **1111**.

- With a terminal count output (TC), counters can be easily cascaded together to form larger counters.
  - For instance, an 8-bit binary counter can be formed as follows.



**INTRO. TO COMP. ENG.  
CHAPTER VIII-1**

**FINITE STATE MACHINES**

**•CHAPTER VIII**

# **CHAPTER VIII**

# **FINITE STATE MACHINES (FSM)**

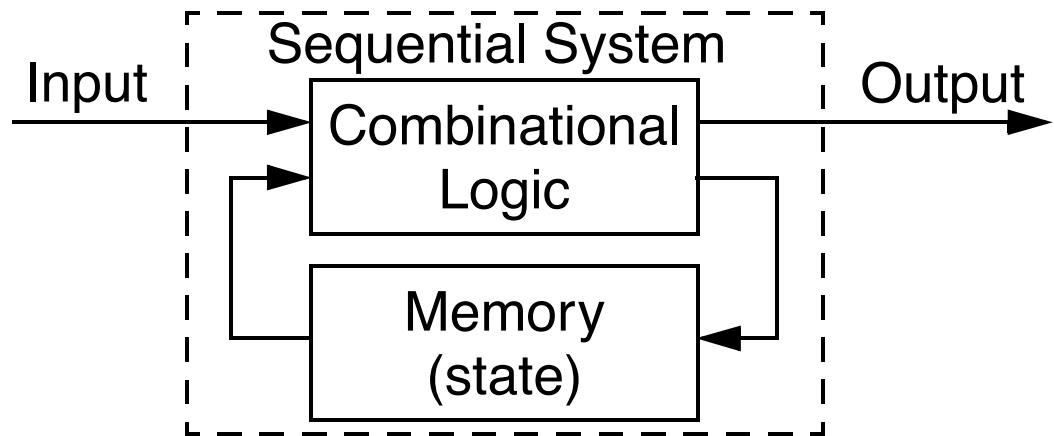
# STATE MACHINES

## INTRODUCTION

- From the previous chapter we can make simple memory elements.
  - Latches as well as latches with control signals
  - Flip-flops
  - Registers
- The goal now is to use the memory elements to hold the running state of the machine.
  - The state of the machine can be used to perform sequential operations.
  - This chapter will discuss how to represent the state of the machine for design and communication purposes.

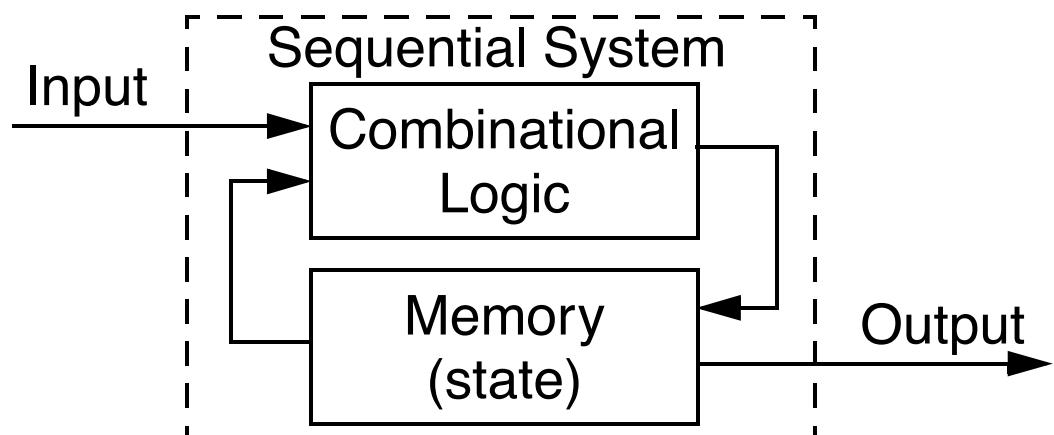
- **Mealy machine**

- Sequential system where output depends on current input and state.



- **Moore machine**

- Sequential system where output depends only on current state.

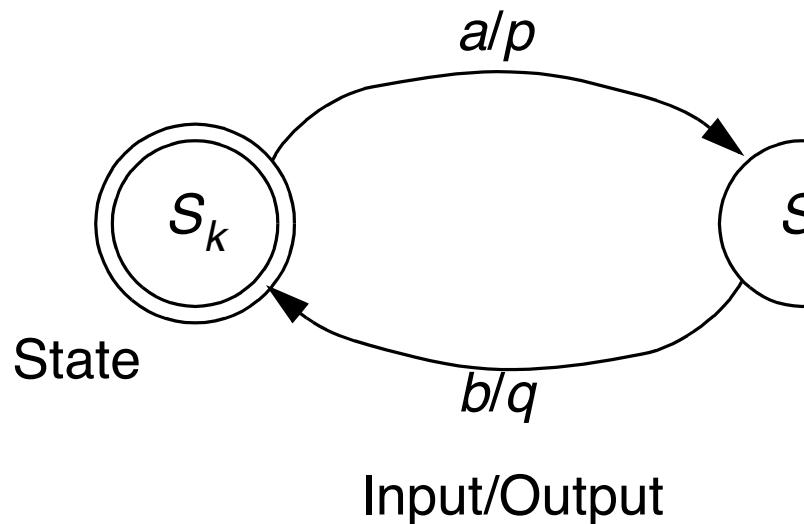


- **Synchronous sequential system**
  - Behaviour depends on the inputs and outputs at discrete instants of time.
  - Flip-flops, registers, and latches that are enabled/controlled with a signal derived from clock form a synchronous sequential system.
- **Asynchronous sequential system**
  - Behaviour depends on inputs at any instant of time.
  - Latches without control signals behave in an asynchronous manner.
- The state machines discussed in this chapter will be synchronous sequential systems (i.e. controlled by a clock)
  - This allows us to form timed Boolean functions such as
    - $\mathbf{N}(t) = \mathbf{D}_A(t+1)$  where  $\mathbf{N}$  is the next state of a D flip-flop  $\mathbf{D}_A$ .

# STATE DIAGRAMS

## ELEMENTS OF DIAGRAMS

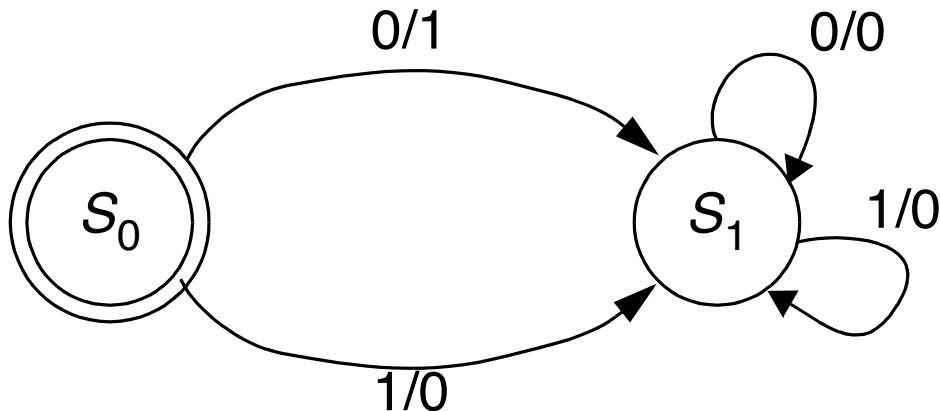
- A state diagram represents a finite state machine (FSM) and contains
  - **Circles:** represent the machine states
    - Labelled with a binary encoded number or  $S_k$  reflecting state.
  - **Directed arcs:** represent the transitions between states
    - Labelled with input/output for that state transition.



Input:  $x(t) \in \{a, b\}$   
Output:  $z(t) \in \{p, q\}$   
State:  $s(t) \in \{S_k, S_j\}$   
Initial state:  $s(0) = S_k$

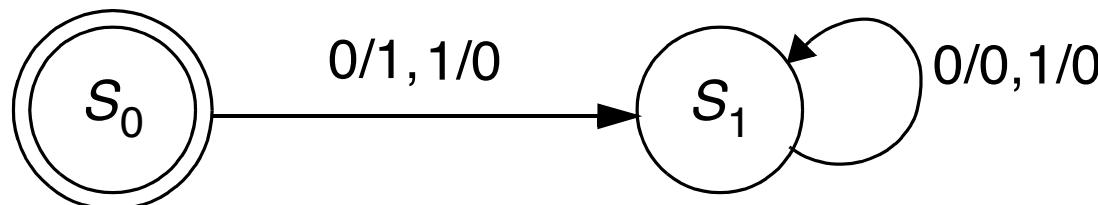
- Some restrictions that are placed on the state diagrams:
  - FSM can only be in **one state at a time!**
    - Therefore, only in one state, or one circle, at a time.
  - State transitions are followed only **on clock cycles**. (synchronous!)
- Mealy machines and Moore machines can be labelled differently.
  - **Mealy machine:** Since output depends on state and inputs:
    - Label directed arcs with **input/output** for that state transition.
  - **Moore machine:** Since output depends only on state:
    - Label directed arcs with **input** for that state transition.
    - Label state circles with  $S_k$ /**output**.

- The following is a simple example. What does this state machine do?



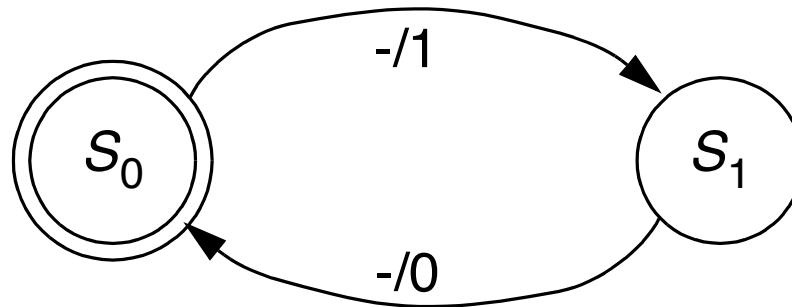
Input:  $x(t) \in \{0, 1\}$   
Output:  $z(t) \in \{0, 1\}$   
State:  $s(t) \in \{S_0, S_1\}$   
Initial state:  $s(0) = S_0$

- Here is a simplified way of forming the above state machine.

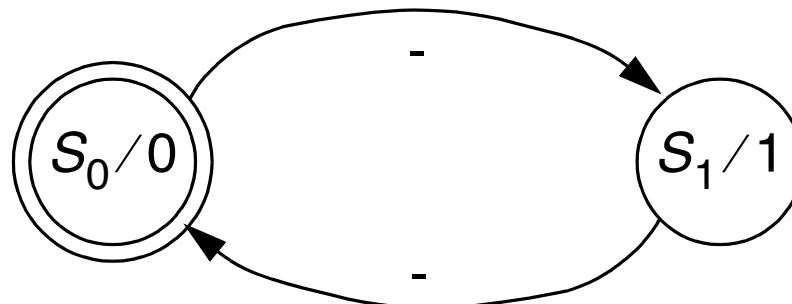


- An input of 0 or 1 causes the transition with output 1 and 0, respectively.

- Consider the simple bit flipper looked at in previous chapter. How would a state diagram be formed?
- Below is one possible way of drawing the state diagram for the bit flipper.



- Since the bit flipper is a Moore machine, the state diagram can also be



# STATE DIAGRAMS

## PATTERN DETECT EXAMPLE

- Suppose we want a sequential system that has the following behaviour

Input:  $x(t) \in \{0, 1\}$

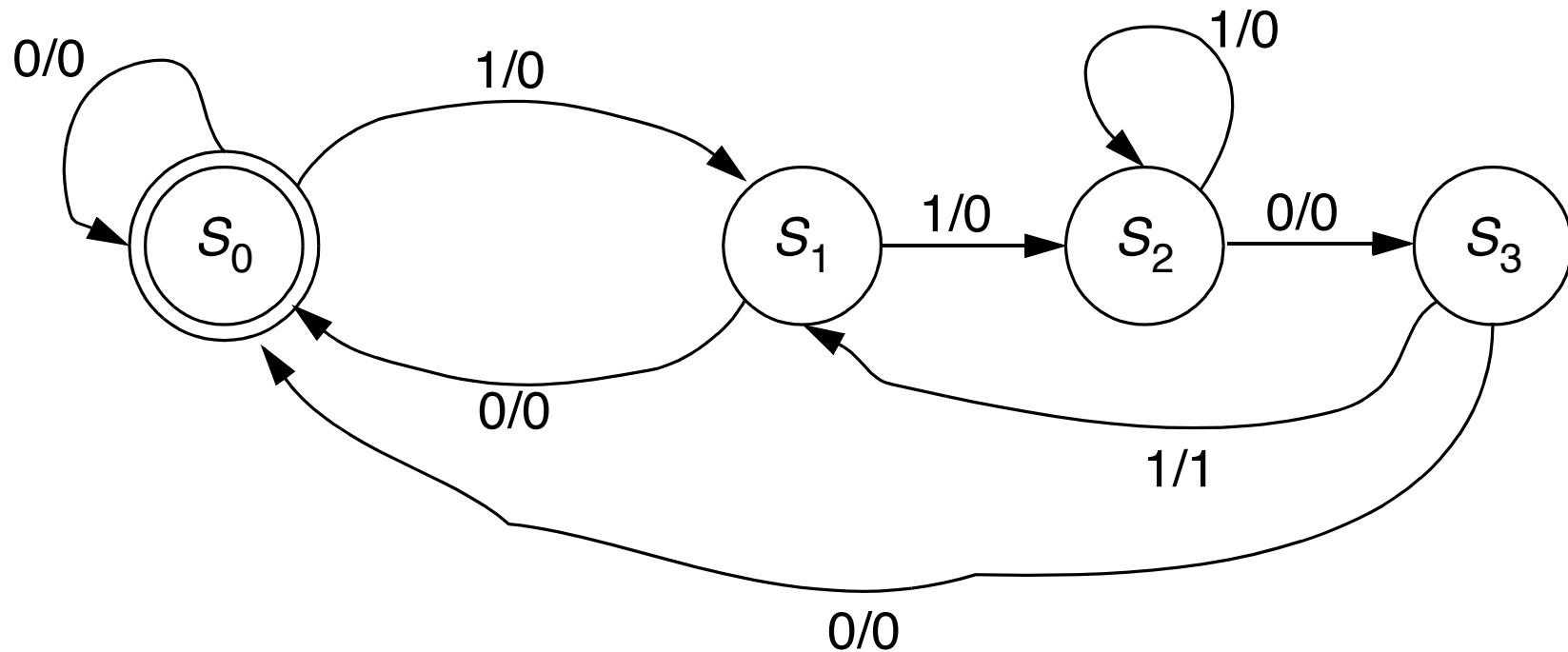
Output:  $z(t) \in \{0, 1\}$

Function: 
$$z(t) = \begin{cases} 1 & \text{if } x(t-3, t) = \mathbf{1101} \\ 0 & \text{otherwise} \end{cases}$$

- Effectively, the system should output a **1** when the last set of four inputs have been **1101**.
- For instance, the following output  $z(t)$  is obtained for the input  $x(t)$

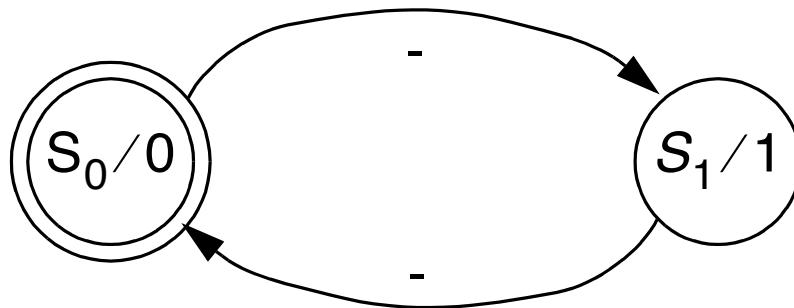
$t$	0123456789...
$x(t)$	100100100100 <b>110101101101001101001</b>
$z(t)$	???00000000000 <b>100001001000001000</b>

- The following state diagram gives the behaviour of the desired 1101 pattern detector.
  - Consider  $S_0$  to be the initial state,  $S_1$  when first symbol detected (1),  $S_2$  when subpattern 11 detected, and  $S_3$  when subpattern 110 detected.



- State tables also express a systems behaviour and consists of
  - Present state
    - The present state of the system, typically given in binary encoded form or with  $S_k$ . So, a state of  $S_5$  in our state diagram with 10 states would be represented as 0101 since we require 4 bits.
  - Inputs
    - Whatever external inputs used to cause the state transitions.
  - Next state
    - The next state, generally in binary encoded form.
  - Outputs
    - Whatever outputs, other then the state, for the system. Note that there would be no outputs in a Moore machine.

- Consider again the bit flipper example with state diagram



- The state table for this state diagram would be

Present State	Input	Next State	Output
$S_0$ or 0	-	1	-
$S_1$ or 1	-	0	-

- From a state diagram, a state table is fairly easy to obtain.
  - Determine the number of states in the state diagram.
  - If there are  $m$  states and  $n$  1-bit inputs, then there will be  $m2^n$  rows in the state table.
    - Example: If there are 3 states and 2 1-bit inputs, each state will have  $2^2 = 4$  possible inputs, for a total of  $3*4=12$  rows.
  - Write out for each state, the  $2^n$  possible input rows.
  - For each state/input pair, follow the directed arc in the state diagram to determine the next state and the output.

# STATE TABLES

## PATTERN DETECT EXAMPLE

- If we consider the pattern detection example previously discussed, the following would be the state table.

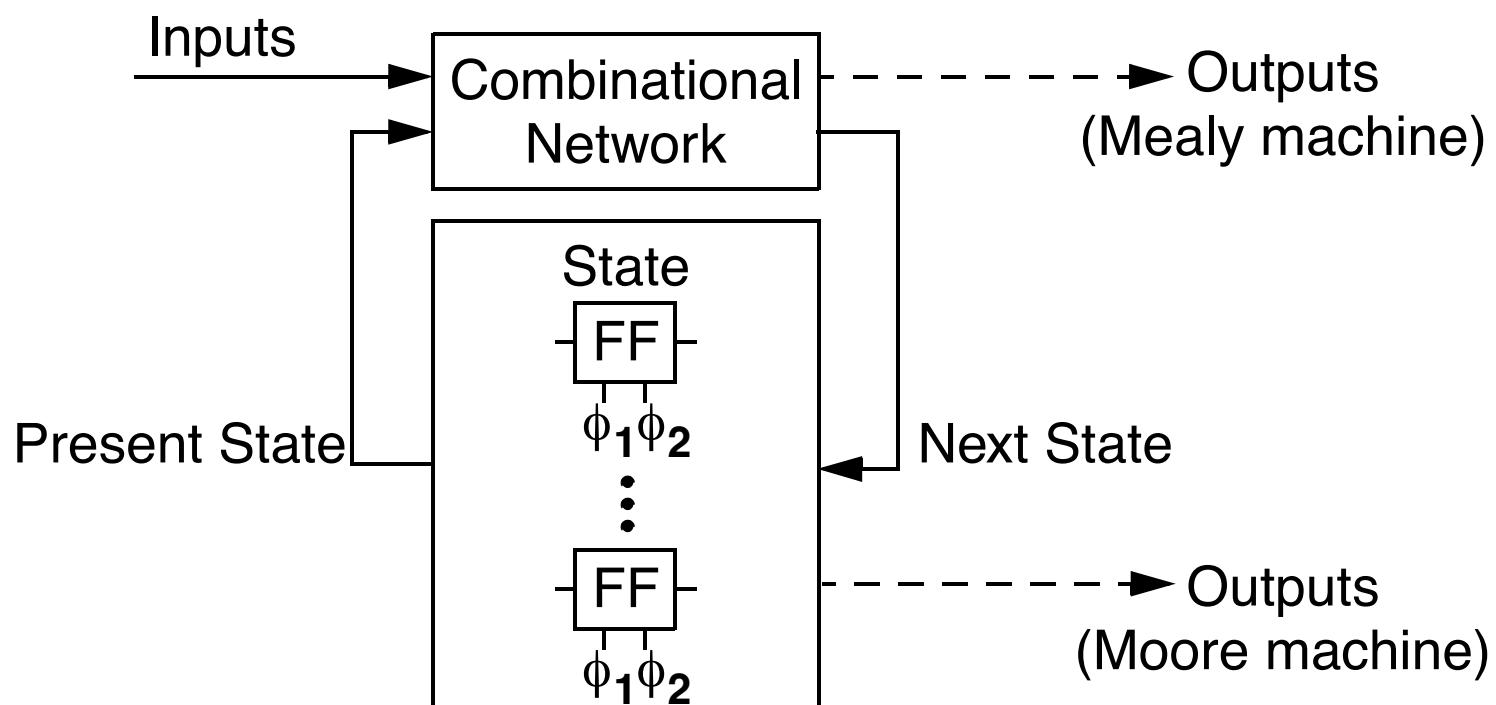
Present State			Input <b>X</b>			Output <b>Z</b>
	<b>P<sub>1</sub></b>	<b>P<sub>0</sub></b>		<b>N<sub>1</sub></b>	<b>N<sub>0</sub></b>	
$S_0$ or 0 0	0	0	0	$S_0$ or 0 0	0 0	0
$S_0$ or 0 0	0	0	1	$S_1$ or 0 1	0 1	0
$S_1$ or 0 1	0	1	0	$S_0$ or 0 0	0 0	0
$S_1$ or 0 1	0	1	1	$S_2$ or 1 0	1 0	0
$S_2$ or 1 0	1	0	0	$S_3$ or 1 1	1 1	0
$S_2$ or 1 0	1	0	1	$S_2$ or 1 0	1 0	0
$S_3$ or 1 1	1	1	0	$S_0$ or 0 0	0 0	0
$S_3$ or 1 1	1	1	1	$S_1$ or 0 1	0 1	1

# STATE TABLES

## TRANSLATE TO DIAGRAM

- If given a state table, the state diagram can be developed as follows.
  - Determine the number of states in the table and draw a state circle corresponding to each one.
    - Label the circle with the state name for a Mealy machine.
    - Label the circle with the state name/output for a Moore machine.
  - For each row in the table, identify the present state circle and draw a directed arc to the next state circle.
    - Label the arc with the input/output pair for a Mealy machine.
    - Label the arc with the input for a Moore machine.

- With the descriptions of a FSM as a state diagram and a state table, the next question is how to develop a sequential circuit, or logic diagram from the FSM.
- Effectively, we wish to form a circuit as follows.



- The procedure for developing a logic circuit from a state table is the same as with a regular truth table.
  - Generate Boolean functions for
    - each external outputs using external inputs and present state bits
    - each next state bit using external inputs and present state bits
  - Use Boolean algebra, Karnaugh maps, etc. as normal to simplify.
  - Draw a register for each state bit.
  - Draw logic diagram components connecting external outputs to external inputs and outputs of state bit registers (which have the present state).
  - Draw logic diagram components connecting inputs of state bits (for next state) to the external inputs and outputs of state bit registers (which have the present state).

- Following the procedure outlined, Boolean functions for the pattern detector state table can be formed using Karnaugh maps as follows.

		$P_1 P_0$	00	01	11	10
		X	0	0	1	0
			0	1	0	0
0	0		0	0	1	0
1	1		1	1	0	0

$N_1$

		$P_1 P_0$	00	01	11	10
		X	0	0	1	0
			0	1	0	0
0	0		0	0	1	0
1	1		1	0	1	0

$N_0$

		$P_1 P_0$	00	01	11	10
		X	0	0	0	0
			0	0	0	0
0	0		0	0	0	0
1	1		0	0	1	0

$Z$

$$N_1 = X\bar{P}_1 + \bar{X}P_1P_0$$

$$N_0 = \bar{X}\bar{P}_1P_0 + X\bar{P}_1P_0 + X\bar{P}_1\bar{P}_0 = \bar{X}\bar{P}_1P_0 + X(\bar{P}_1 \oplus P_0)$$

$$Z = X\bar{P}_1P_0$$

- Notice that the previous Boolean functions can also be expressed with time as follows.

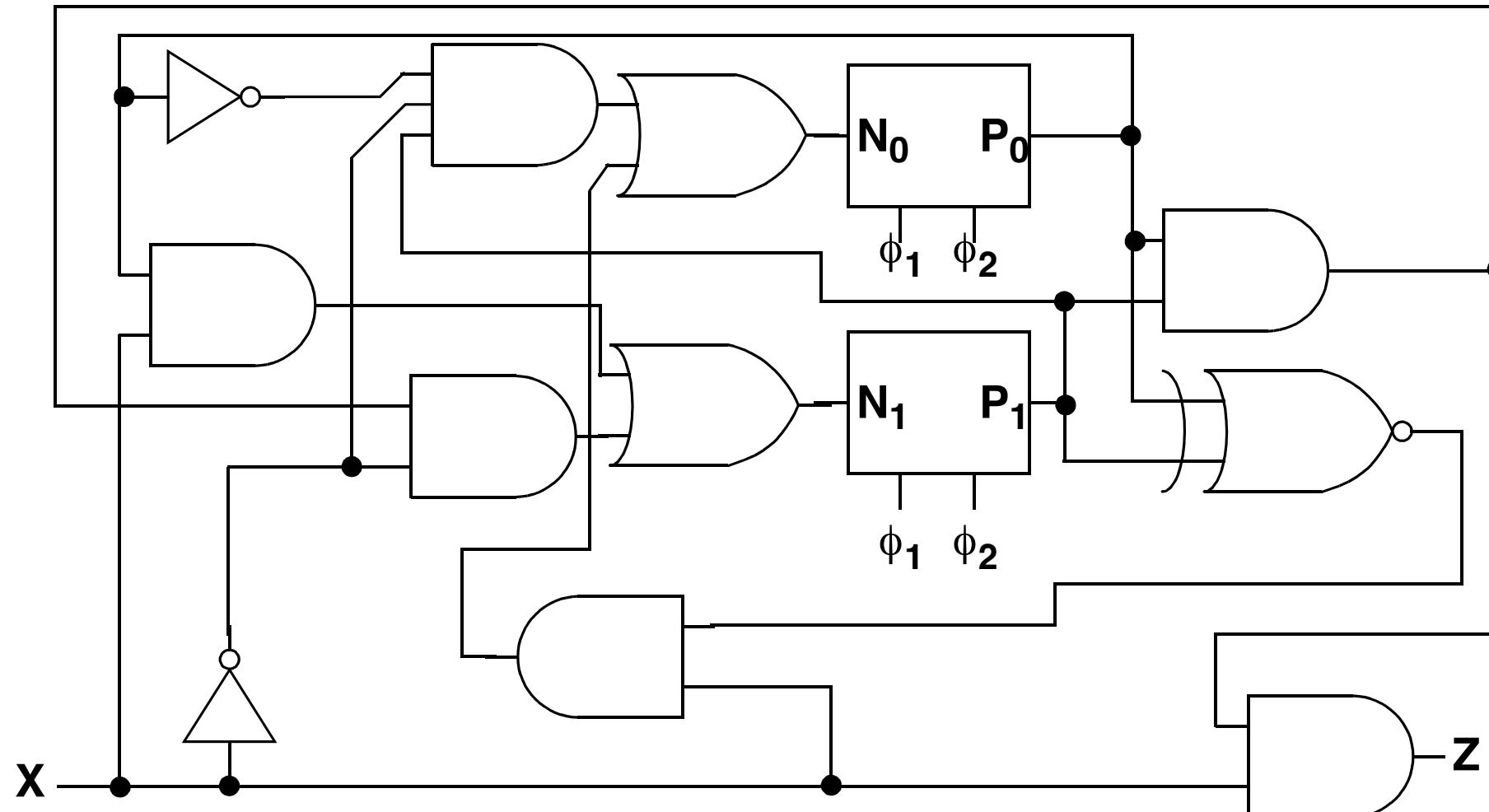
$$N_1(t) = P_1(t+1) = X(t) \cdot \overline{P_1(t)} + \overline{X(t)} \cdot P_1(t) \cdot P_0(t)$$

$$\begin{aligned} N_0(t) = P_0(t+1) &= \overline{X(t)} \cdot \overline{P_1(t)} \cdot P_0(t) + X(t) \cdot P_1(t) \cdot P_0(t) \\ &\quad + X(t) \cdot \overline{P_1(t)} \cdot \overline{P_0(t)} \\ &= \overline{X(t)} \cdot \overline{P_1(t)} \cdot P_0(t) + X(t) \cdot \overline{P_1(t) \oplus P_0(t)} \end{aligned}$$

$$Z(t) = X \cdot P_1(t) \cdot P_0(t)$$

- An important thing to note in these equations is the relation between the present states P and the next states N.

- The following logic circuit implements the pattern detect example.



# FSM EXAMPLES

## EXAMPLE #1

- Consider the following system description.
  - A sequential system has
    - One input = {**a**, **b**, **c**}
    - One output = {**p**, **q**}
  - Output is
    - **q** when input sequence has even # of **a**'s and odd # of **b**'s
    - **p** otherwise

# FSM EXAMPLES

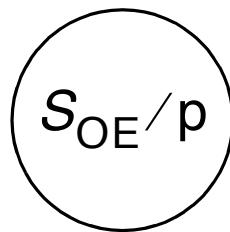
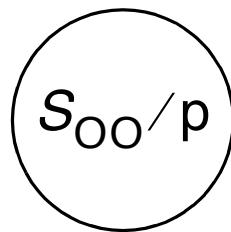
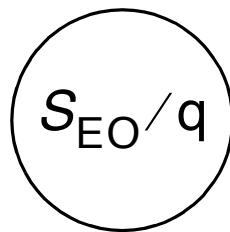
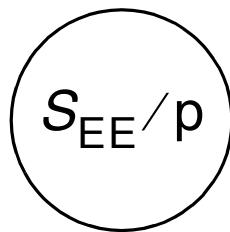
## EXAMPLE #1

- We can begin forming a state machine for the system description by reviewing the possible states. In addition, assign each state a state name.
  - $S_{EE}$ : even # of **a**'s and even # of **b**'s / output is **p**
  - $S_{EO}$ : even # of **a**'s and odd # of **b**'s / output is **q**
  - $S_{OO}$ : odd # of **a**'s and odd # of **b**'s / output is **p**
  - $S_{OE}$ : odd # of **a**'s and even # of **b**'s / output is **p**
- Note that this machine can be a Moore machine. So, we can associate the output with each state.

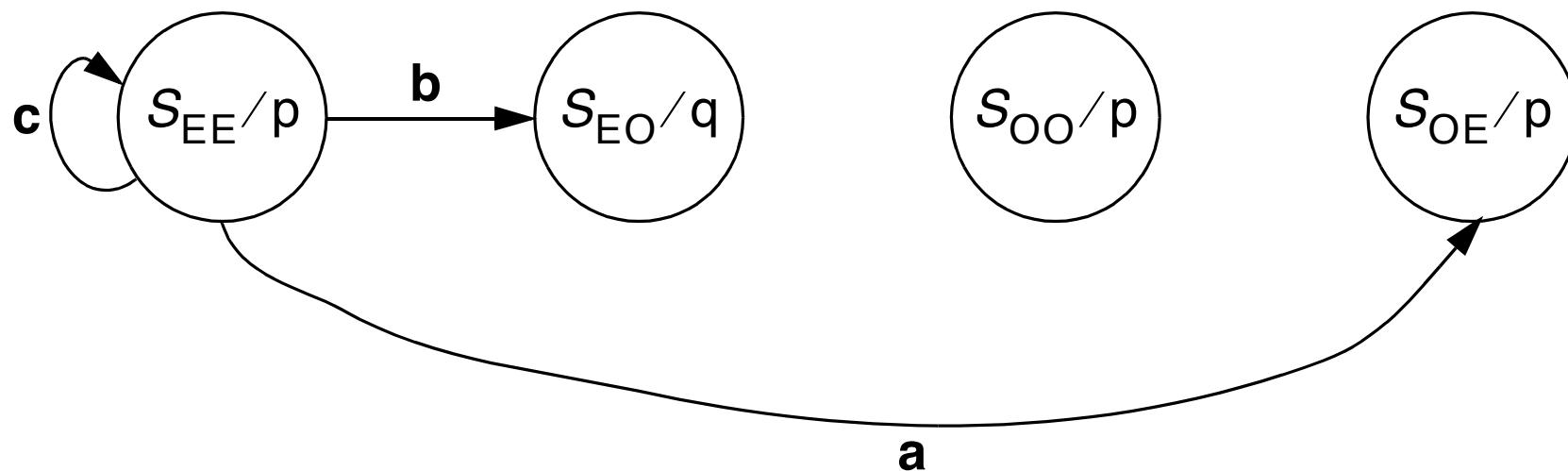
# FSM EXAMPLES

## EXAMPLE #1

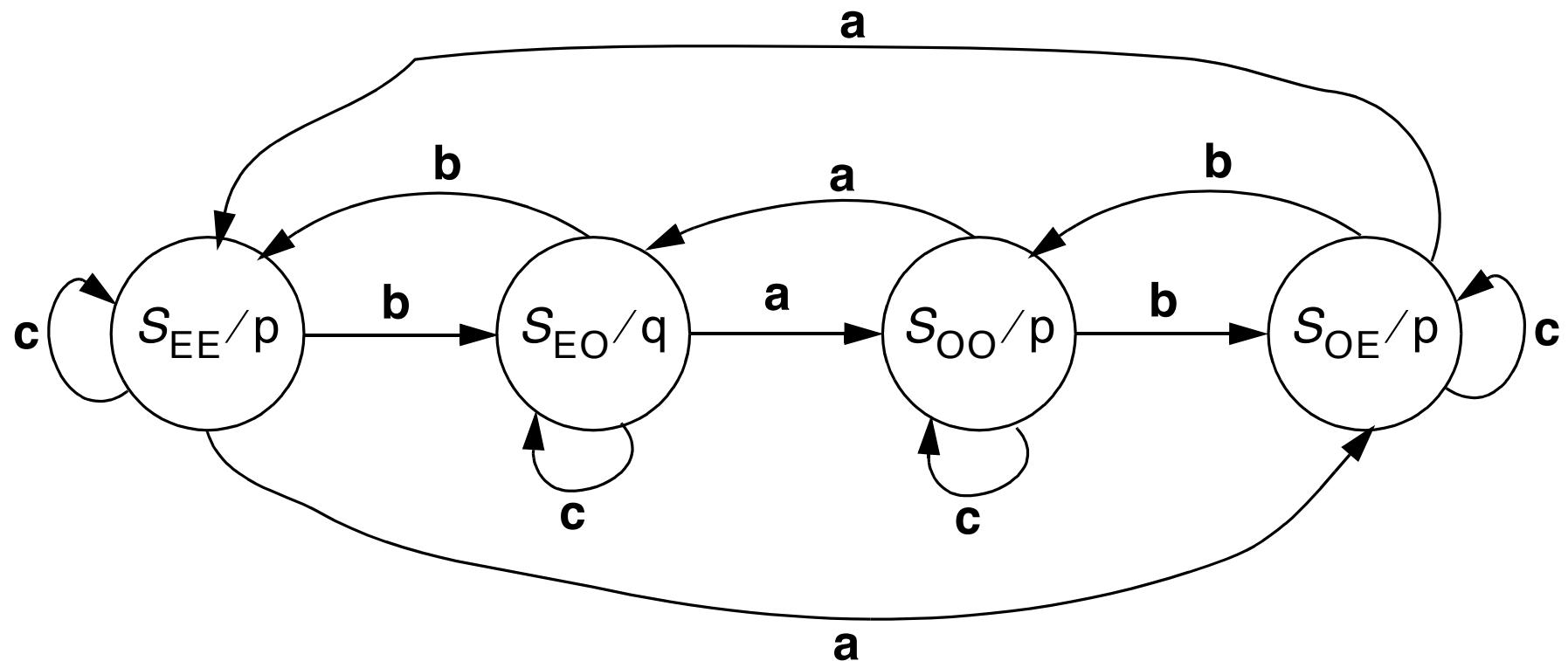
- Now draw a circle with each state.



- Finally, for each state, consider the effect for each possible input.
  - For instance, starting with state  $S_{EE}$ , the next state for the three input **a**, **b**, and **c** are determined as follows.



- Finishing the state diagram, the following is obtained.



- A state table can also be formed for this state diagram as follows.
  - First, assign a binary number to each state
    - $S_{EE} = 00$ ,  $S_{EO} = 01$ ,  $S_{OO} = 10$ ,  $S_{OE} = 11$
  - Assign a binary number to each input
    - $a = 00$ ,  $b = 01$ ,  $c = 10$
  - Assign a binary number to each output
    - $p = 0$ ,  $q = 1$
  - Then for each state, find the next state for each input. In this case there are three possible input values, so, three possible state transitions from each state.
  - The state table on the following slide shows the results for this example.

# FSM EXAMPLES

## EXAMPLE #1

Present State $P_1 \ P_0$	Input $X$	Next State $N_1 \ N_0$	Output $Z$
$S_{EE} = 0 \ 0$	$a = 00$	$S_{OE} = 1 \ 1$	$p = 0$
$S_{EE} = 0 \ 0$	$b = 01$	$S_{EO} = 0 \ 1$	$p = 0$
$S_{EE} = 0 \ 0$	$c = 10$	$S_{EE} = 0 \ 0$	$p = 0$
$S_{EO} = 0 \ 1$	$a = 00$	$S_{OO} = 1 \ 0$	$q = 1$
$S_{EO} = 0 \ 1$	$b = 01$	$S_{EE} = 0 \ 0$	$q = 1$
$S_{EO} = 0 \ 1$	$c = 10$	$S_{EO} = 0 \ 1$	$q = 1$
$S_{OO} = 1 \ 0$	$a = 00$	$S_{EO} = 0 \ 1$	$p = 0$
$S_{OO} = 1 \ 0$	$b = 01$	$S_{OE} = 1 \ 1$	$p = 0$
$S_{OO} = 1 \ 0$	$c = 10$	$S_{OO} = 1 \ 0$	$p = 0$
$S_{OE} = 1 \ 1$	$a = 00$	$S_{EE} = 0 \ 0$	$p = 0$
$S_{OE} = 1 \ 1$	$b = 01$	$S_{OO} = 1 \ 0$	$p = 0$
$S_{OE} = 1 \ 1$	$c = 10$	$S_{OE} = 1 \ 1$	$p = 0$

- The Boolean function for the output can be determined from a Karnaugh map as follows.
  - Note that an input of **11** is not possible since we only have three inputs that we have assigned to **00**, **01**, and **10**. We can therefore use don't cares for this possible input.

		$P_1 P_0$	00	01	11	10	
		$X_1 X_0$	00	0	1	0	0
		01	0	1	0	0	
		11	X	X	X	X	
		10	0	1	0	0	

$$Z = \overline{P_1} P_0$$

- The Boolean function for the next state bit can also be determined from Karnaugh maps as follows.

		P <sub>1</sub> P <sub>0</sub>	00	01	11	10
		X <sub>1</sub> X <sub>0</sub>	00	01	11	10
X <sub>1</sub> X <sub>0</sub>	P <sub>1</sub> P <sub>0</sub>	00	1	1	0	0
		01	0	0	1	1
11	X	X	X	X		
10	0	0	1	1		

$$N_1 = \overline{P_1 \oplus X_1 \oplus X_0}$$

		P <sub>1</sub> P <sub>0</sub>	00	01	11	10
		X <sub>1</sub> X <sub>0</sub>	00	01	11	10
X <sub>1</sub> X <sub>0</sub>	P <sub>1</sub> P <sub>0</sub>	00	1	0	0	1
		01	1	0	0	1
11	X	X	X	X		
10	0	1	1	0		

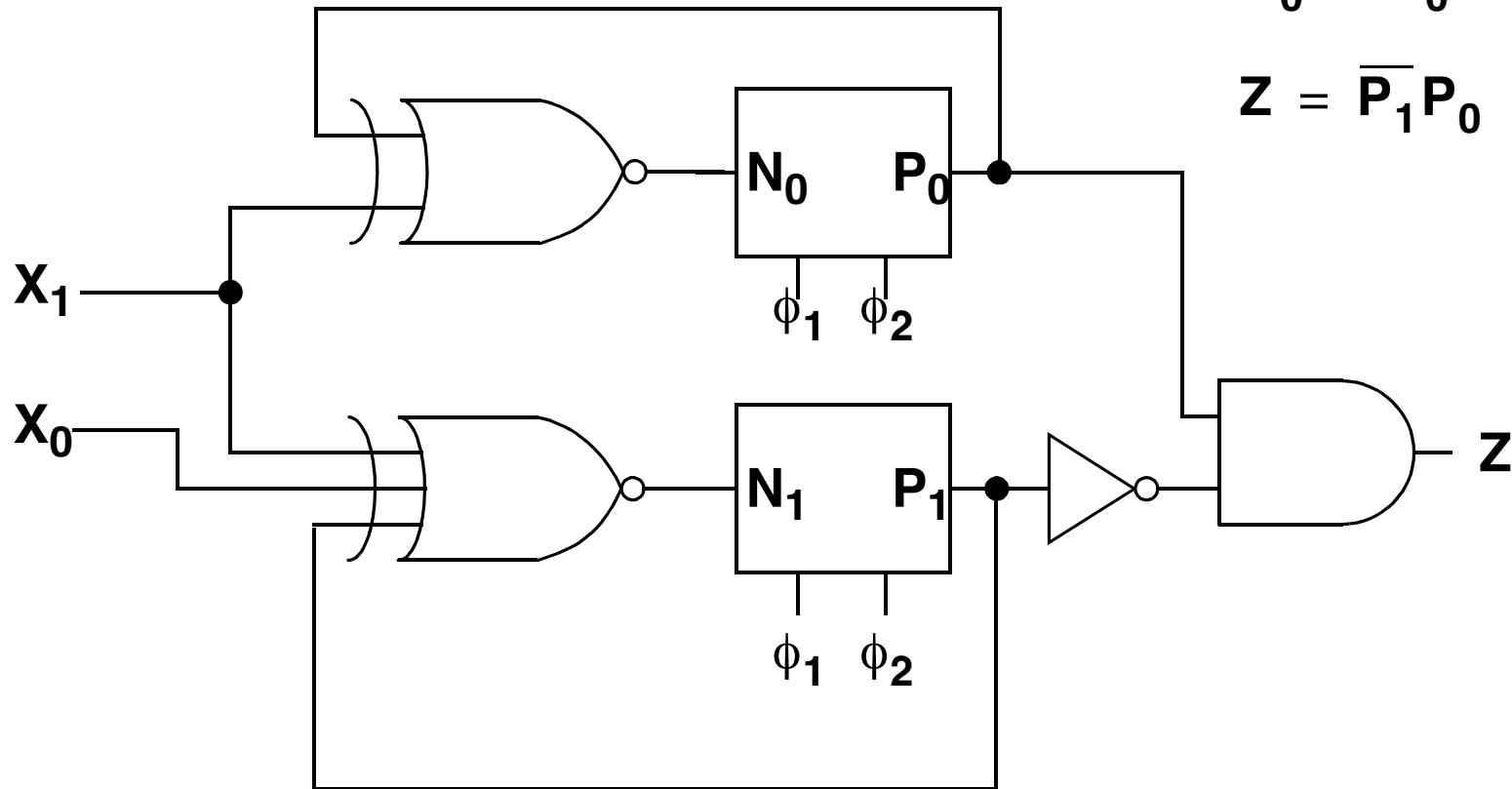
$$N_0 = P_0 X_1 + \overline{P_0} \overline{X_1} = \overline{P_0 \oplus X_1}$$

- The following logic circuit can be made with these Boolean functions.

$$N_1 = \overline{P_1 \oplus X_1 \oplus X_0}$$

$$N_0 = \overline{P_0 \oplus X_1}$$

$$Z = \overline{P_1} P_0$$



- A sequential circuit is defined by the following Boolean functions with input  $X$ , present states  $P_0$ ,  $P_1$ , and  $P_2$ , and next states  $N_0$ ,  $N_1$ , and  $N_2$ .
  - $N_2 = X(P_1 \oplus P_0) + \overline{X}(\overline{P_1 \oplus P_0})$
  - $N_1 = P_2$
  - $N_0 = P_1$
  - $Z = XP_1P_2$
- Derive the state table.
- Derive the state diagram.

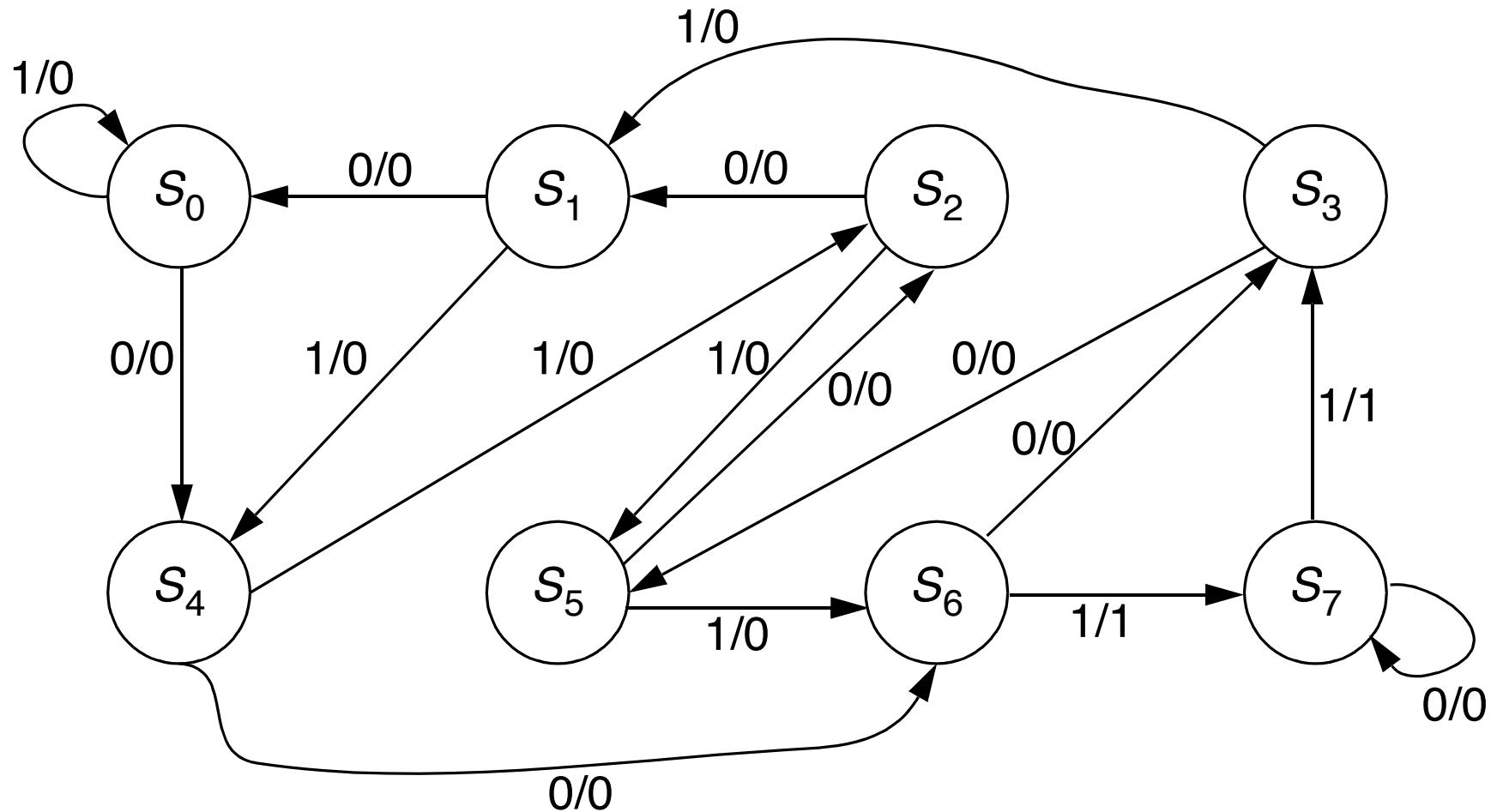
# FSM EXAMPLES

## EXAMPLE #2

- The state table is formed as follows.

Present State			Input	Next State			Output
$P_2$	$P_1$	$P_0$	$X$	$N_2$	$N_1$	$N_0$	$Z$
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	0	1	0
0	1	0	1	1	0	1	0
0	1	1	0	1	0	1	0
0	1	1	1	0	0	1	0
1	0	0	0	1	1	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	0
1	0	1	1	1	1	0	0
1	1	0	0	0	1	1	0
1	1	0	1	1	1	1	1
1	1	1	0	1	1	1	0
1	1	1	1	0	1	1	1

- The state diagram can be drawn as follows.



**INTRO. TO COMP. ENG.  
CHAPTER VII-1  
SEQUENTIAL SYSTEMS**

**•CHAPTER VII**

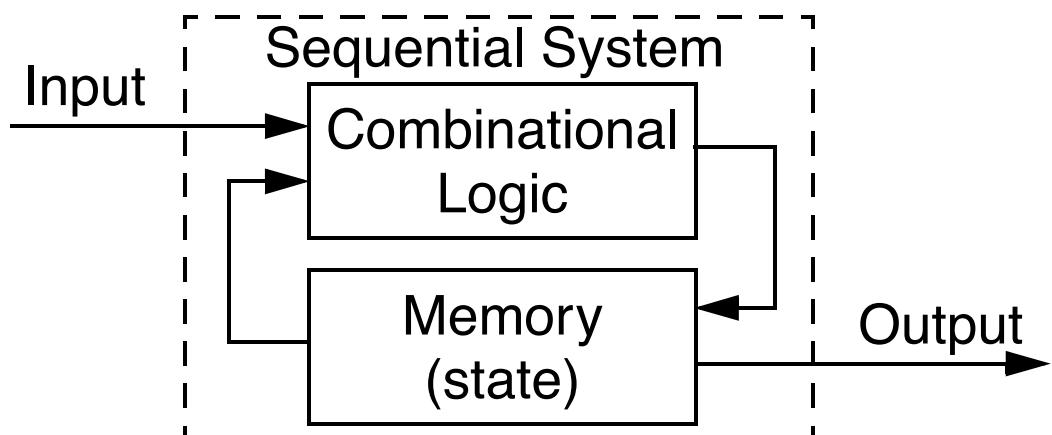
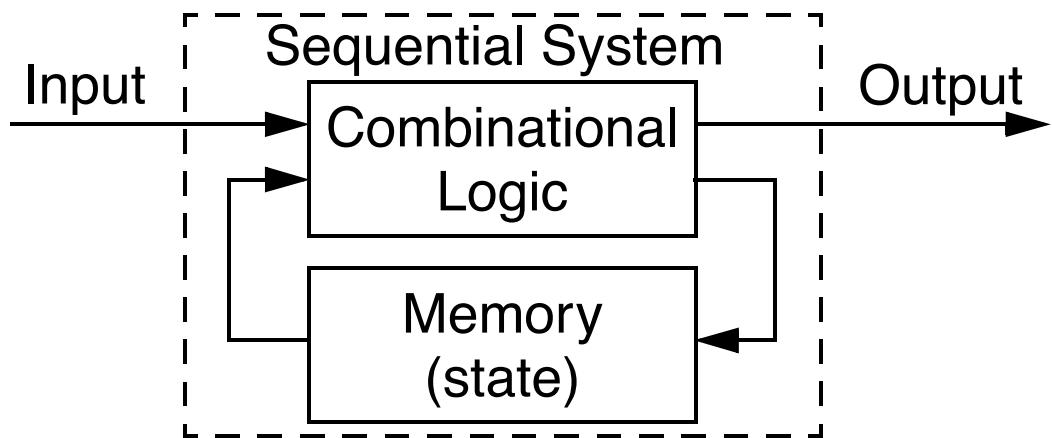
# **CHAPTER VII**

## **SEQUENTIAL SYSTEMS - LATCHES & REGISTERS**

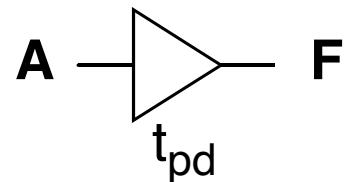
- So far...
  - So far we have dealt only with combinational logic where the output is formed from the current input.
  - Sequential systems
    - Sequential systems extend the idea of combinational logic by including a **system state**, or in other words **memory**, to our system.
    - This allows our system to perform operations that build on past operations in a *sequential* manner (*i.e.* one after another).
    - Timing diagrams will be needed to analyze the operation of many sequential systems.



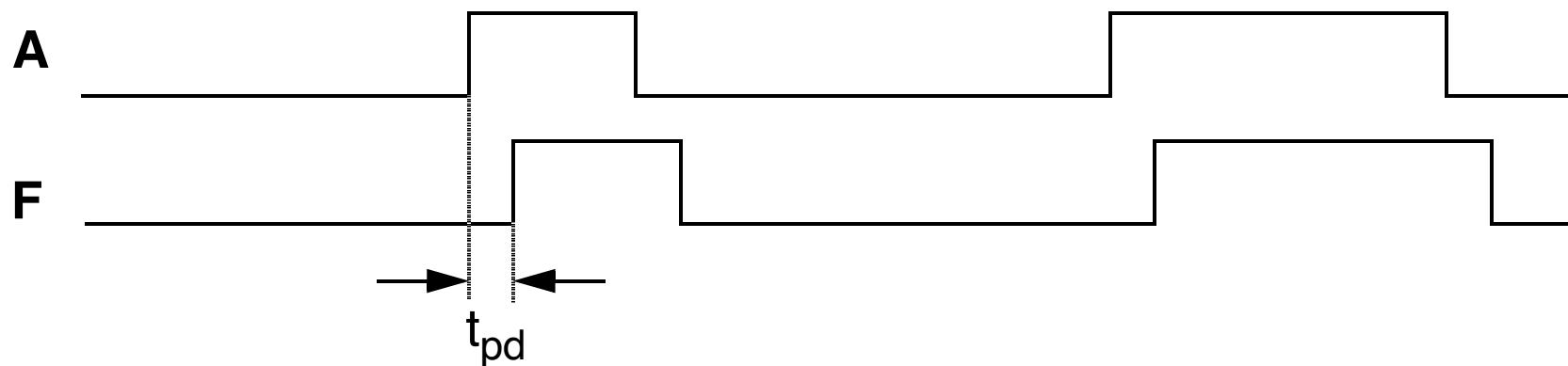
- Mealy machine
  - Sequential system where output depends on current input and state.
- Moore machine
  - Sequential system where output depends only on current state.



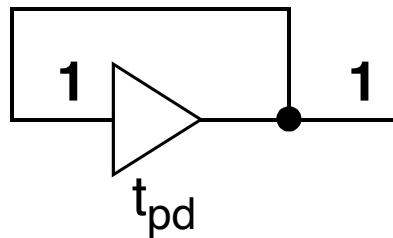
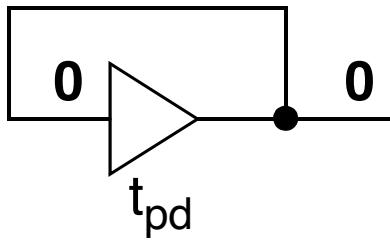
- Since there are propagation delays in real components, this time delay can be used to store information.
- For instance, the following buffer has a propagation delay of  $t_{pd}$ .



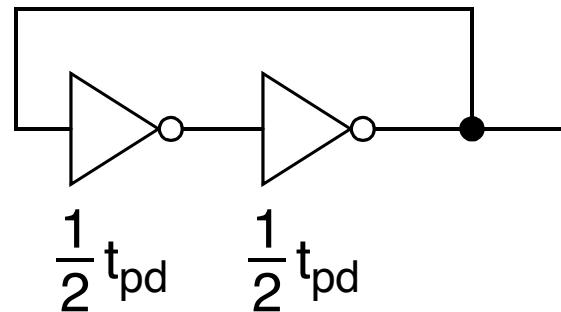
Timing Diagram



- If we wish to store data for an indefinite period of time, then a feedback loop can be used to maintain the bit.



Can also use two inverters!

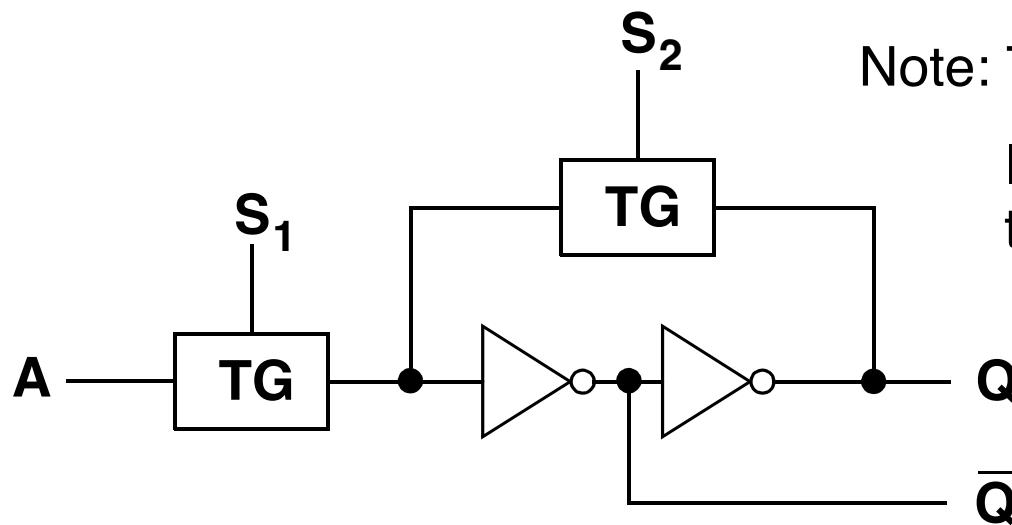


- How do we get the bit in there?

# STORING BITS

## LOADING A BIT

- To store a bit, we need a way of loading an input bit into the structure and making/breaking the connection in the feedback loop.
- One way of breaking connections is to use transmission gates.



Note: The latch is level-sensitive.

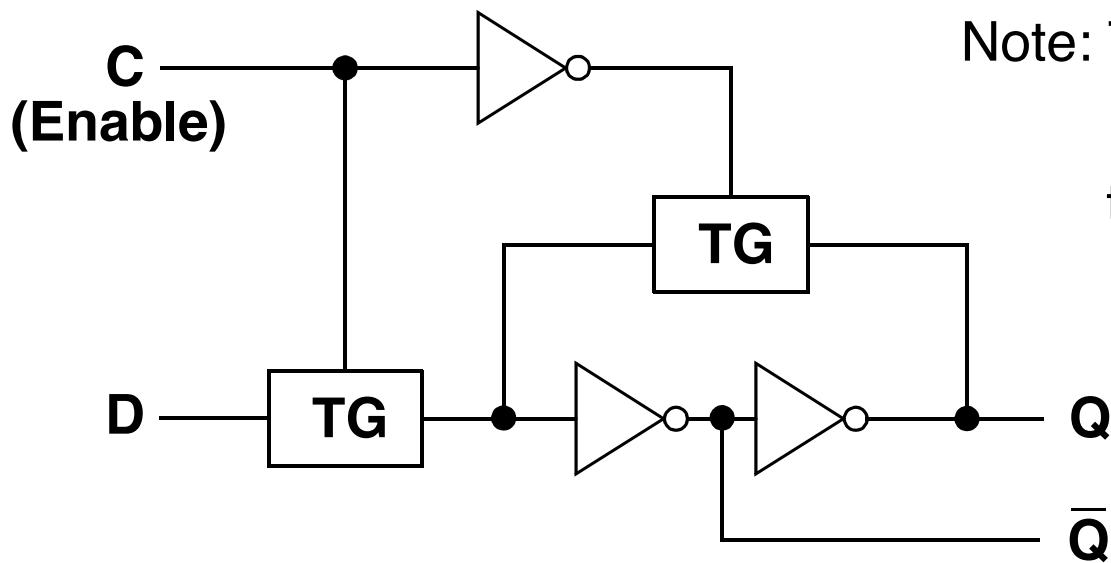
If  $A$  changes while  $S_1 = 1$ , then  $Q$  will change as well.

- $A$  gets temporarily stored in the inverters when  $S_1 = 1$  and  $S_2 = 0$ . Then setting  $S_1 = 0$  and  $S_2 = 1$ ,  $A$  gets held in the feedback loop.

# LATCHES

## D LATCH (WITH TG)

- The previous example is a data latch (**D latch**) if both  $S_1$  and  $S_2$  are controlled by a single line **C** as follows.



Note: The latch is level-sensitive.  
If **D** changes while **C** = 1,  
then **Q** will change as well.

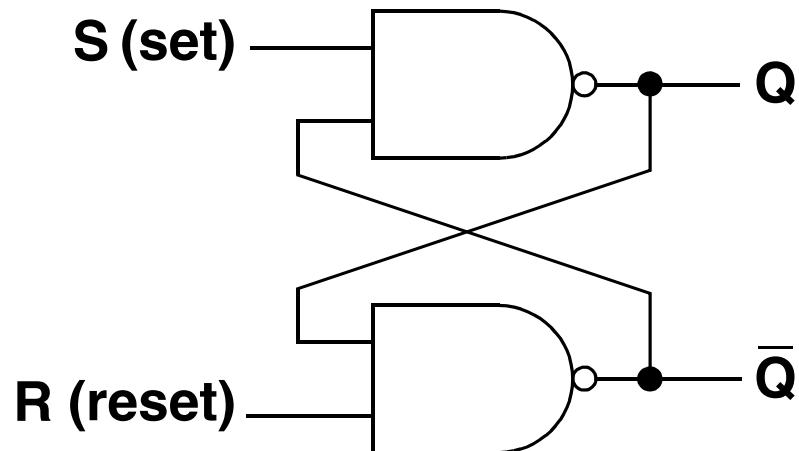
- The control line **C** might be derived from the **clock signal**, or a signal from the **controller/sequencer** in the microprocessor.

# LATCHES

## $\overline{S}\overline{R}$ LATCH (NAND GATES)

- LATCHES
  - D LATCH (WITH TG)
  - NAND PRIMITIVES
  - CONSTRUCTING A LATCH

- NAND gates can also be used to create a latch, this time an  $\overline{S}\overline{R}$  latch.



S	R	Q	$\bar{Q}$
1	0	0	1
1	1	0	1 (after $S = 1, R = 0$ )
0	1	1	0
1	1	1	0 (after $S = 0, R = 1$ )
0	0	1	1

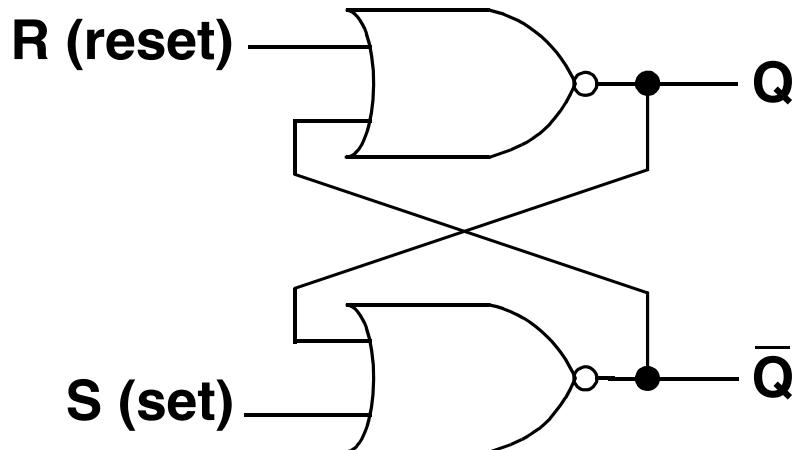
Recall:		
A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0

- Notice that this latch is level-sensitive.

# LATCHES

## SR LATCH (NOR GATES)

- The SR latch also uses feedback to “store” a bit.



S	R	Q	$\bar{Q}$	
1	0	1	0	
0	0	1	0	(after S = 1, R = 0)
0	1	0	1	
0	0	0	1	(after S = 0, R = 1)
1	1	0	0	

Recall:

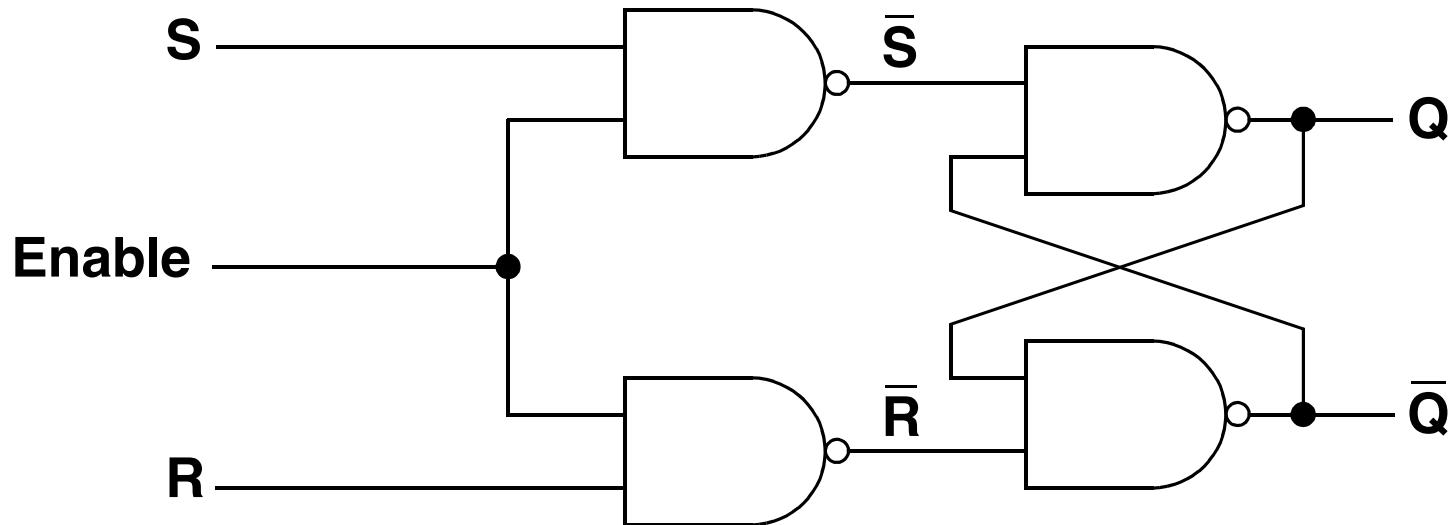
A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

- Notice that this latch is level-sensitive.

# LATCHES

## SR LATCH WITH CONTROL

- A control line can be added to the  $\bar{S}\bar{R}$  latch as follows forming an SR latch

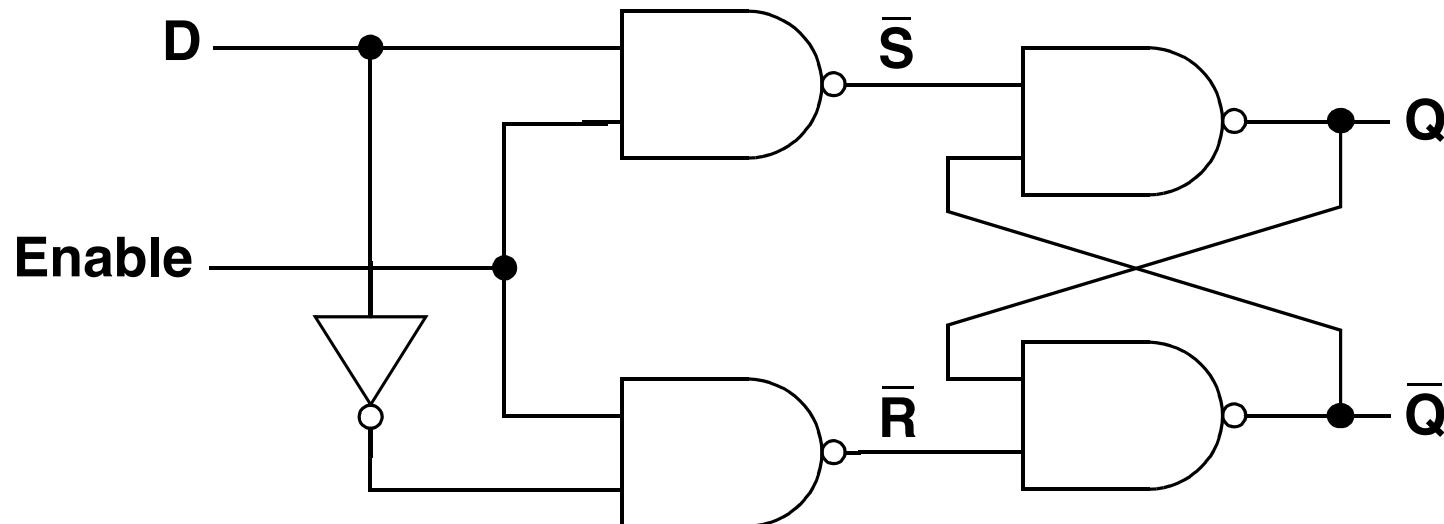


- This control line makes it possible to decide when the inputs **S** and **R** are allowed to change the state of the latch.

# LATCHES

## D LATCH (WITH SR LATCH)

- A D latch can be implemented using what is effectively the SR latch with a control line as follows.

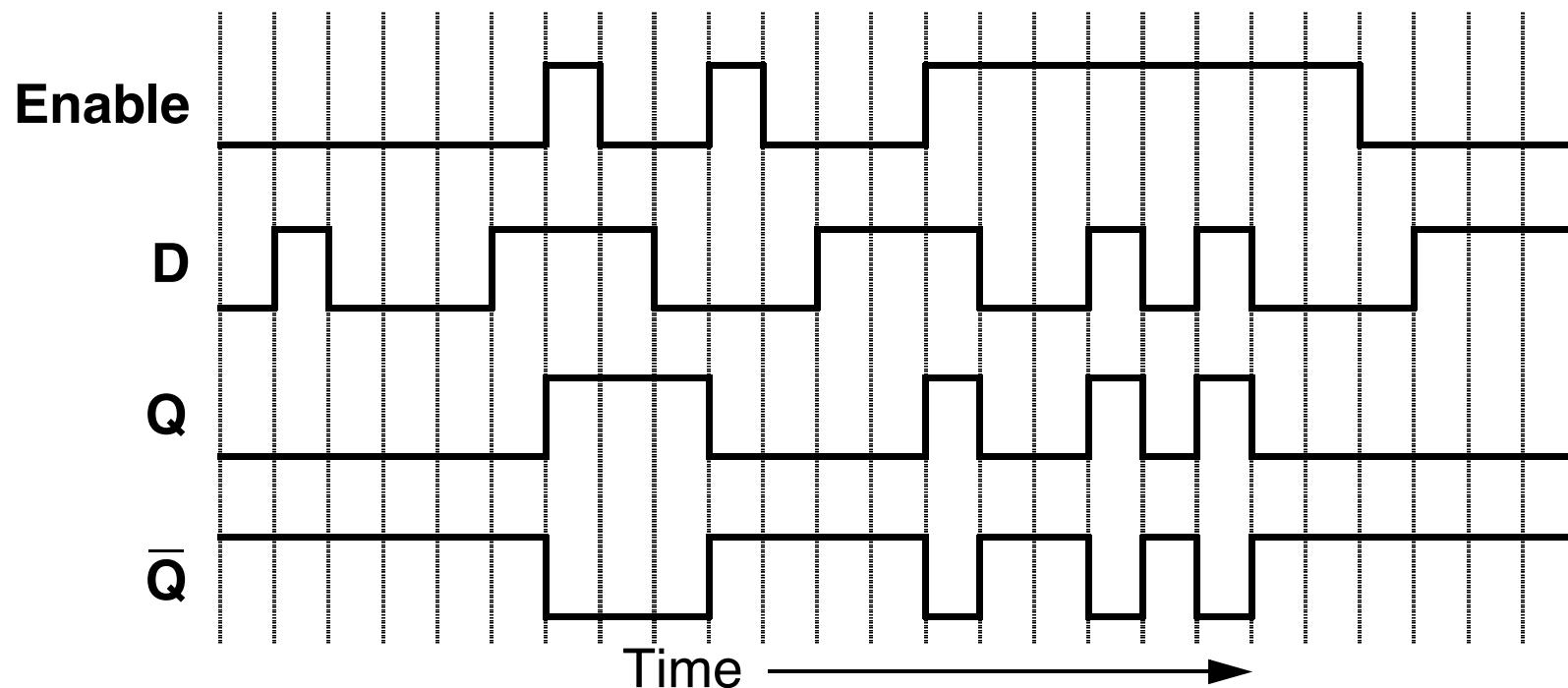


- Note that as long as  $C = 1$ , that the latch will change according to the value of  $D$ .

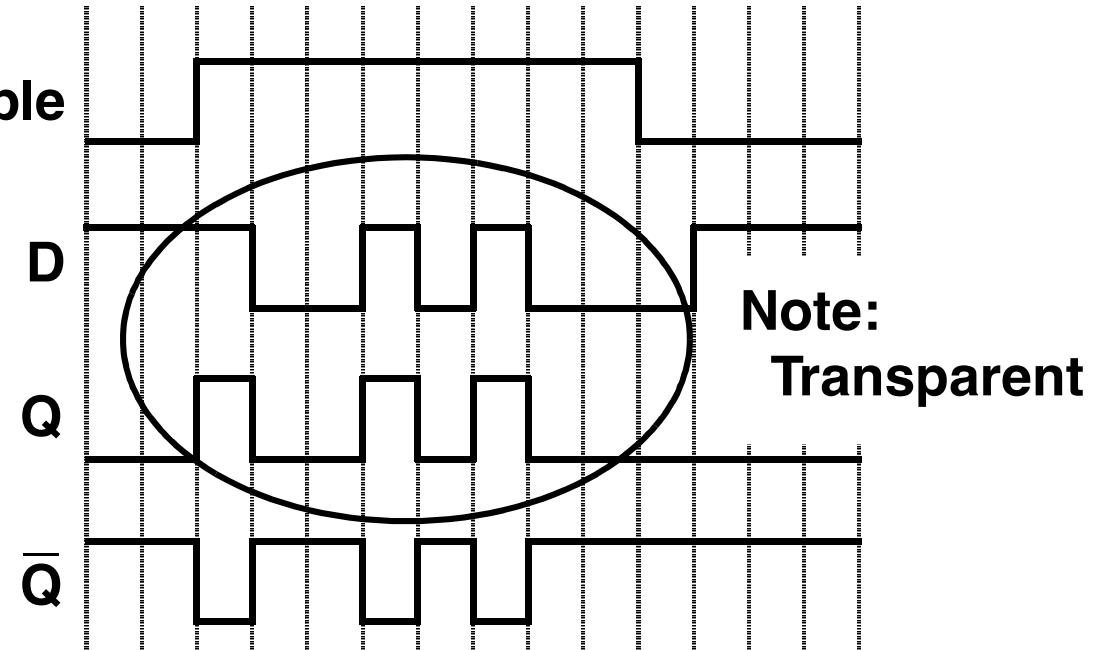
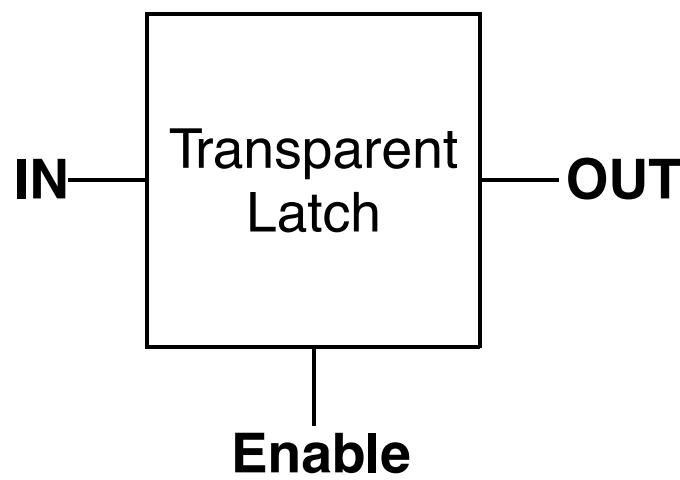
# LATCHES

## TIMING DIAGRAMS

- Timing diagrams allow you to see how a sequential system changes with time using different inputs.
- For instance, a timing diagram for a **D latch** might look like the following.



- Latches like the D latch are termed “**transparent**” or **level-sensitive**.
  - This is because, when enabled, the **output follows the input**.

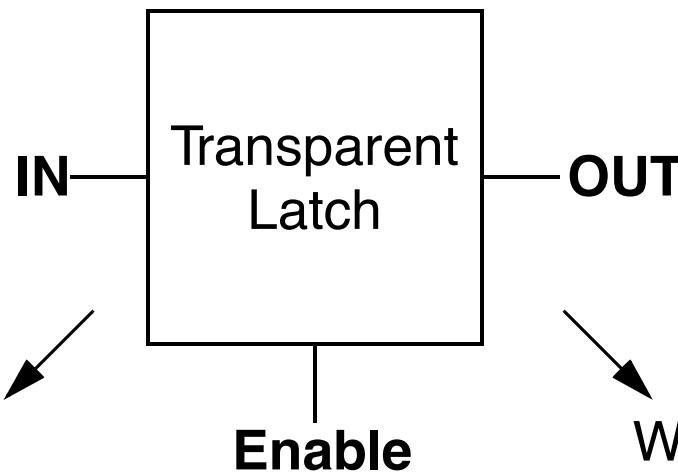


# LATCHES

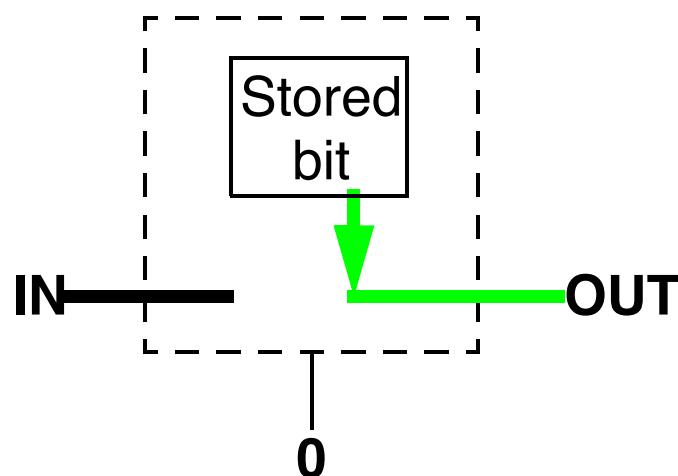
## TRANSPARENCY (2)

- LATCHES
  - D LATCH
  - TIMING DIAGRAMS
  - TRANSPARENCY

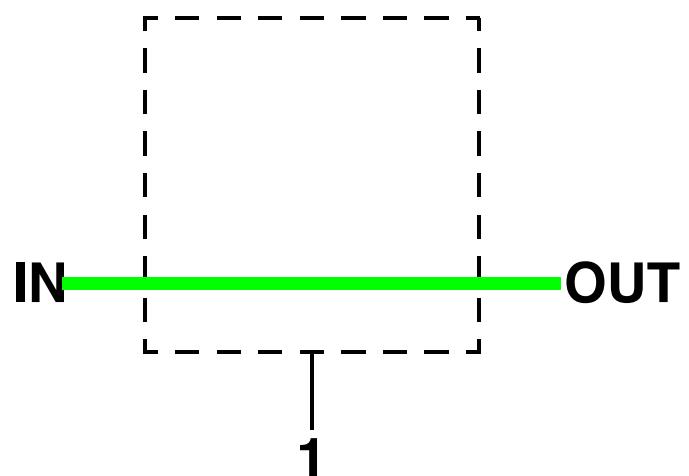
- The following behaviour is observed for Enable = 0 and Enable = 1.



When **Enable = 0**,  
input disconnected and  
stored bit outputted.



When **Enable = 1**,  
latch acts like wire.

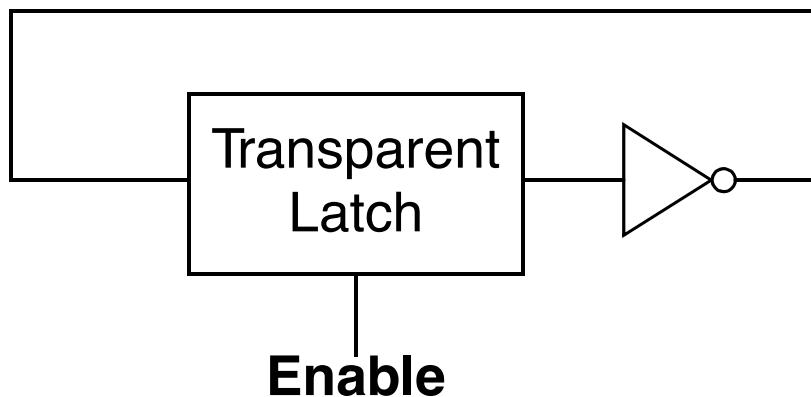


# LATCH EXAMPLE

## PROBLEMS W/ TRANSPARENCY

- LATCHES
  - D LATCH
  - TIMING DIAGRAMS
  - TRANSPARENCY

- A problem with latches is that they are level-sensitive.
  - A momentary change of input changes the value passed out of the latch.
- This is a problem if the input of a latch depends on the output of the same latch.
- Example: Design a system that flips a stored bit whenever **Enable** goes high. An inexperienced engineer might design the following.



How will this design behave?

Will the bit flip once when the **Enable** signal goes high?

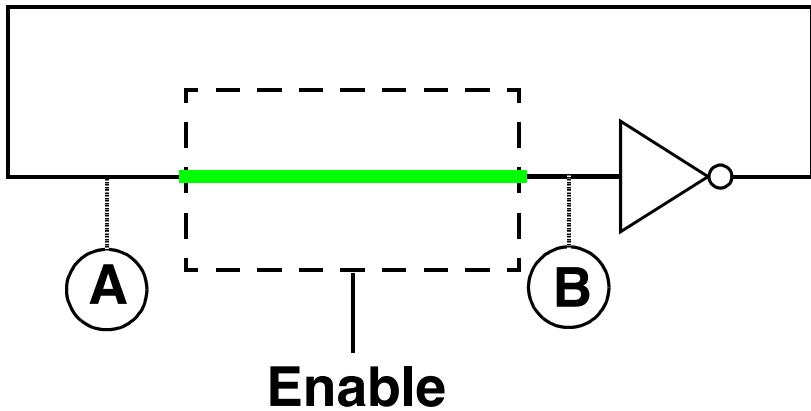
Answer: The output will follow the input, which happens to keep changing.

# LATCH EXAMPLE

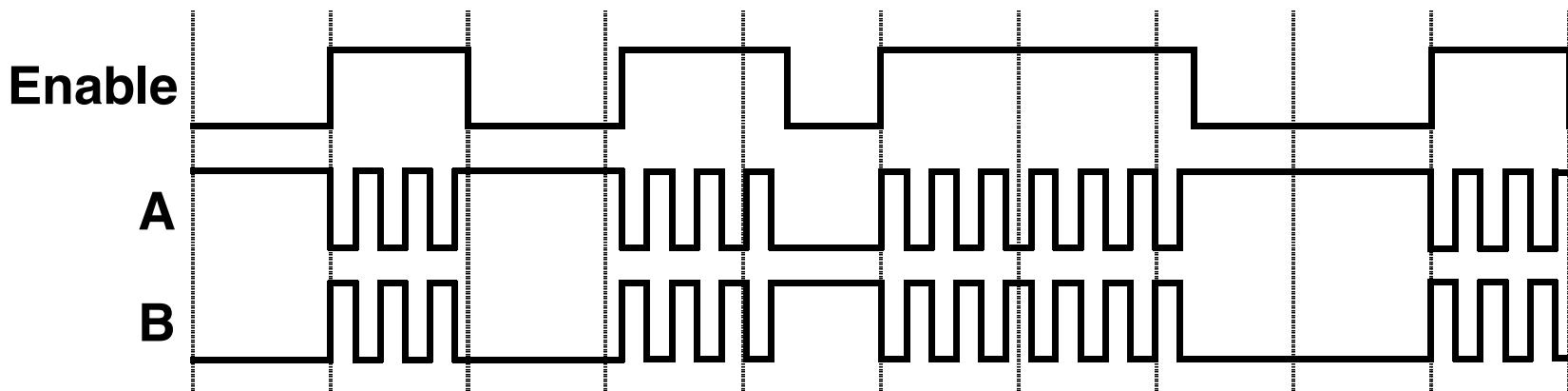
## PROBLEMS W/ TRANSPARENCY

- LATCHES
- LATCH EXAMPLE
- PROB W/TRANSPARENCY

- Let's analyze the timing behaviour of this "poor" design.



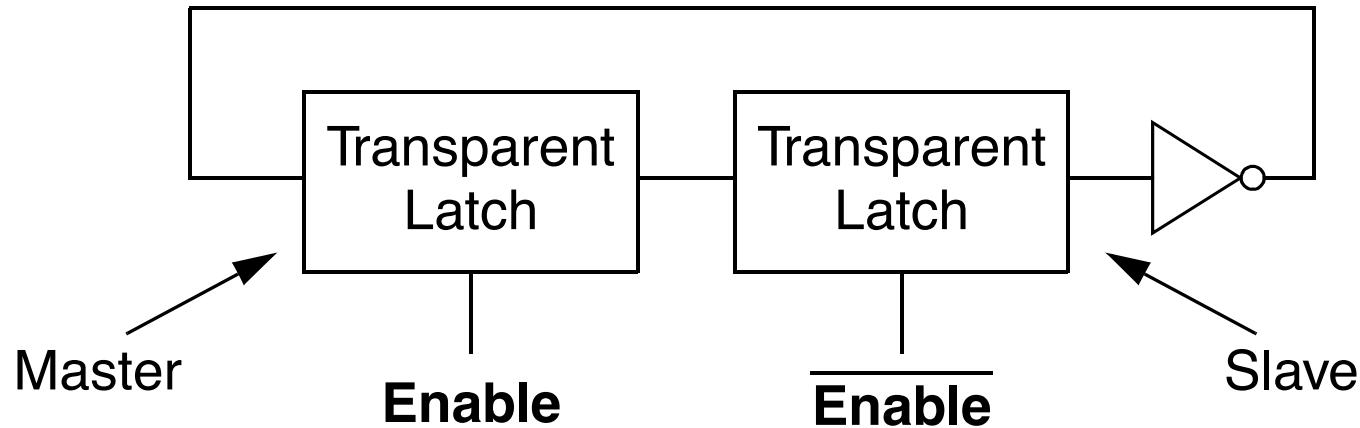
- Notice that instead of the desired bit flip when **Enable=1**, that the input oscillates. This is because the output depends directly on the input since **A** and **B** appear to be connected by a wire.



# LATCH EXAMPLE

## ELIMINATING TRANSPARENCY

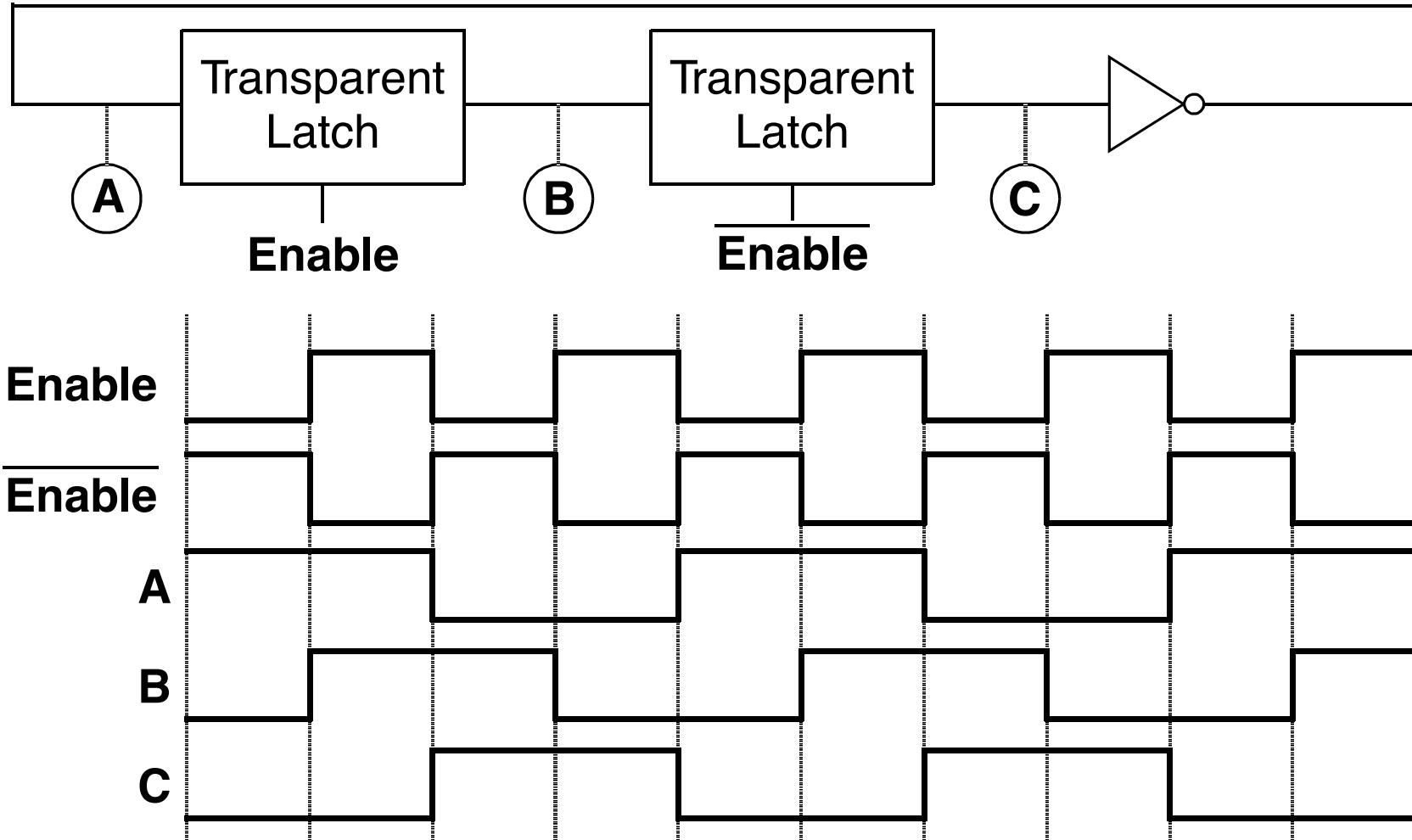
- The problem with transparent, level-sensitive latches can be fixed by splitting the input and output so that they are independent.
  - New solution: Consider the following improved design that flips a stored bit whenever **Enable** goes high. This design now uses a master and a slave transparent latches to separate the input from the output.



# LATCH EXAMPLE

## TIMING DIAGRAM

- Let's analyze the timing behaviour of this improved design.

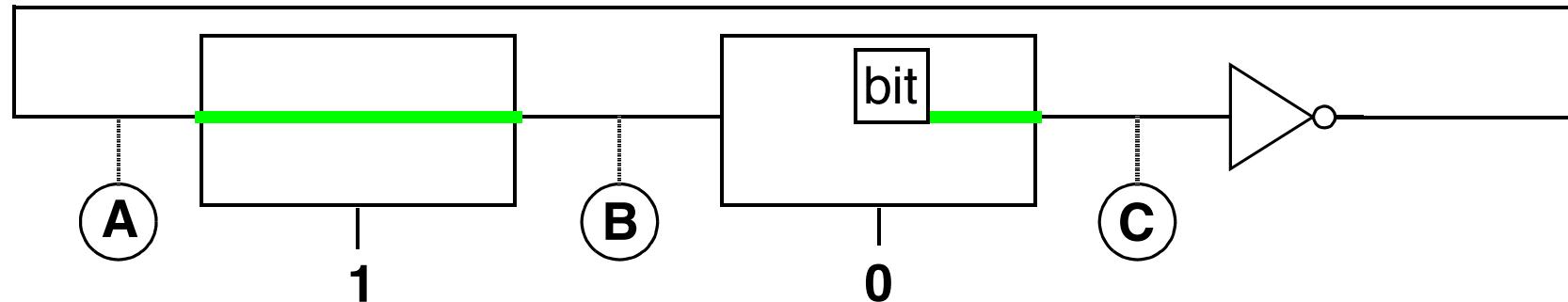


# LATCH EXAMPLE

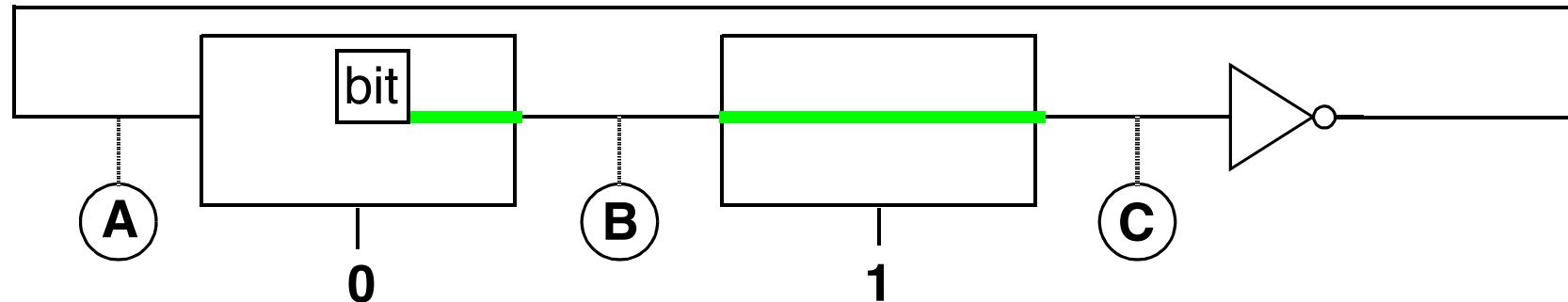
## LATCH BEHAVIOUR

- The behaviour of the master and the slave transparent latches can be thought of as follows.

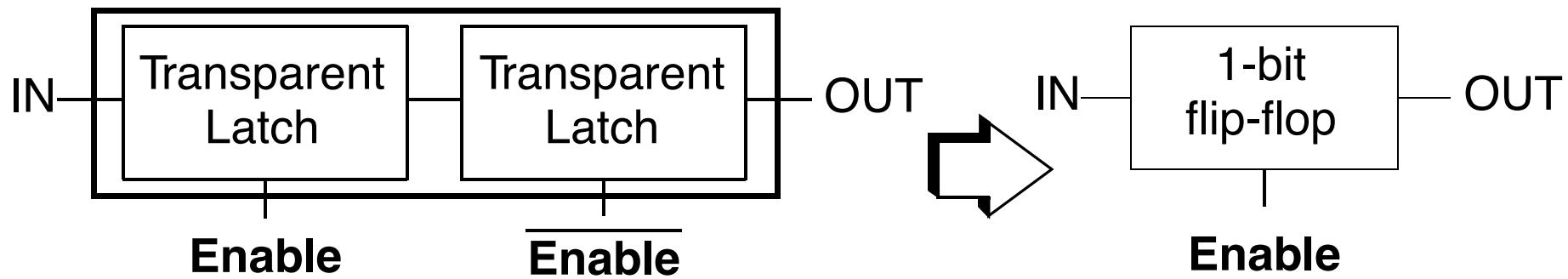
**Enable = 1**



**Enable = 0**

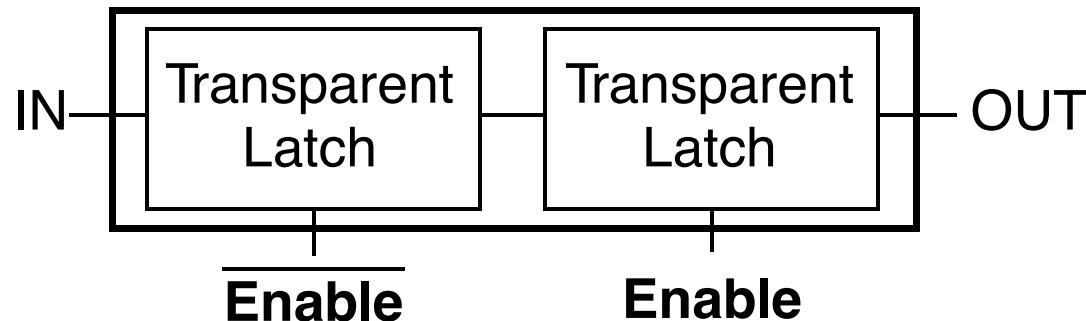


- A flip-flop is a single bit storage unit with two stages (master/slave):
  - First stage, or master, to accept input (flip)
  - Second stage, or slave, to give output as received by the first stage (flop)

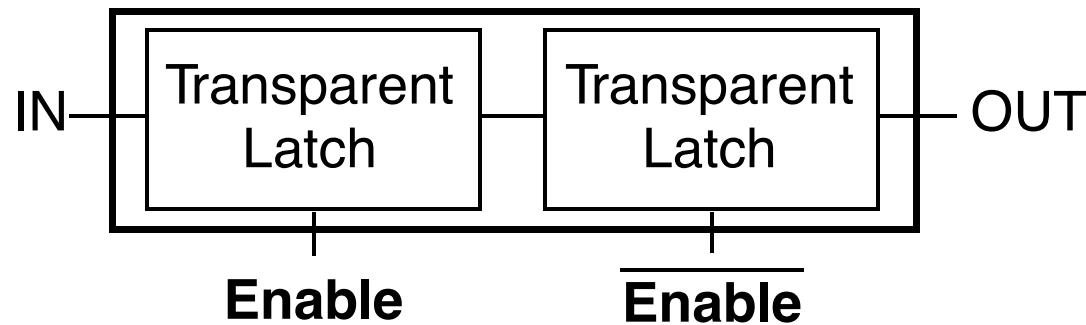


- A number of different types of flip-flops exist such as the SR,  $\bar{S}\bar{R}$ , D, and JK flip-flops. You may wish to review Chapter 4 regarding these types.

- A common and useful type of flip-flop are edge triggered flip-flops.
  - Positive edge triggered flip-flops



- Negative edge triggered flip-flops

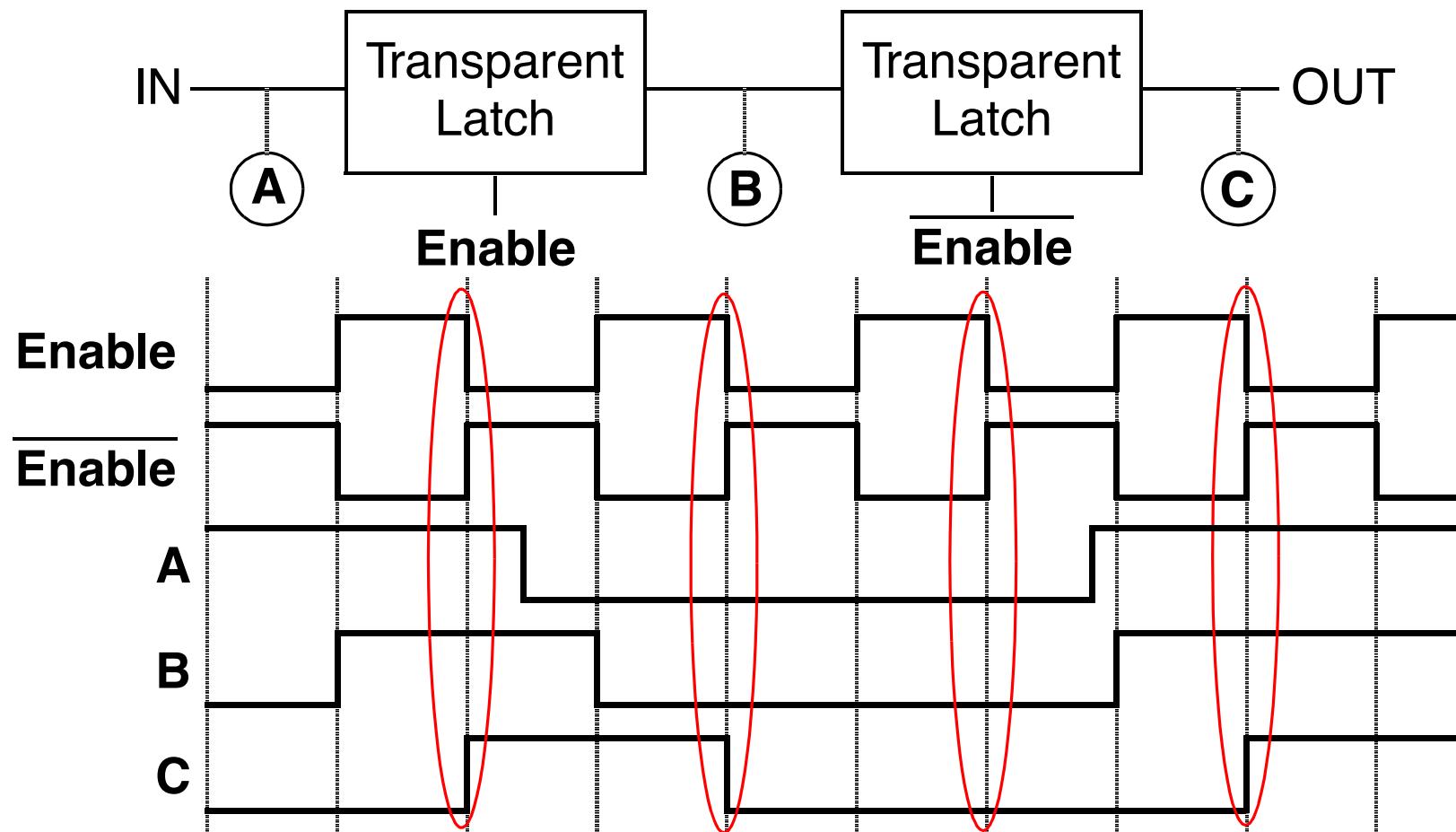


# FLIP-FLOPS

NEGATIVE EDGE TRIGGERED

- LATCH EXAMPLE
- FLIP-FLOPS
- SINGLE BIT STORAGE
- EDGE TRIGGERED

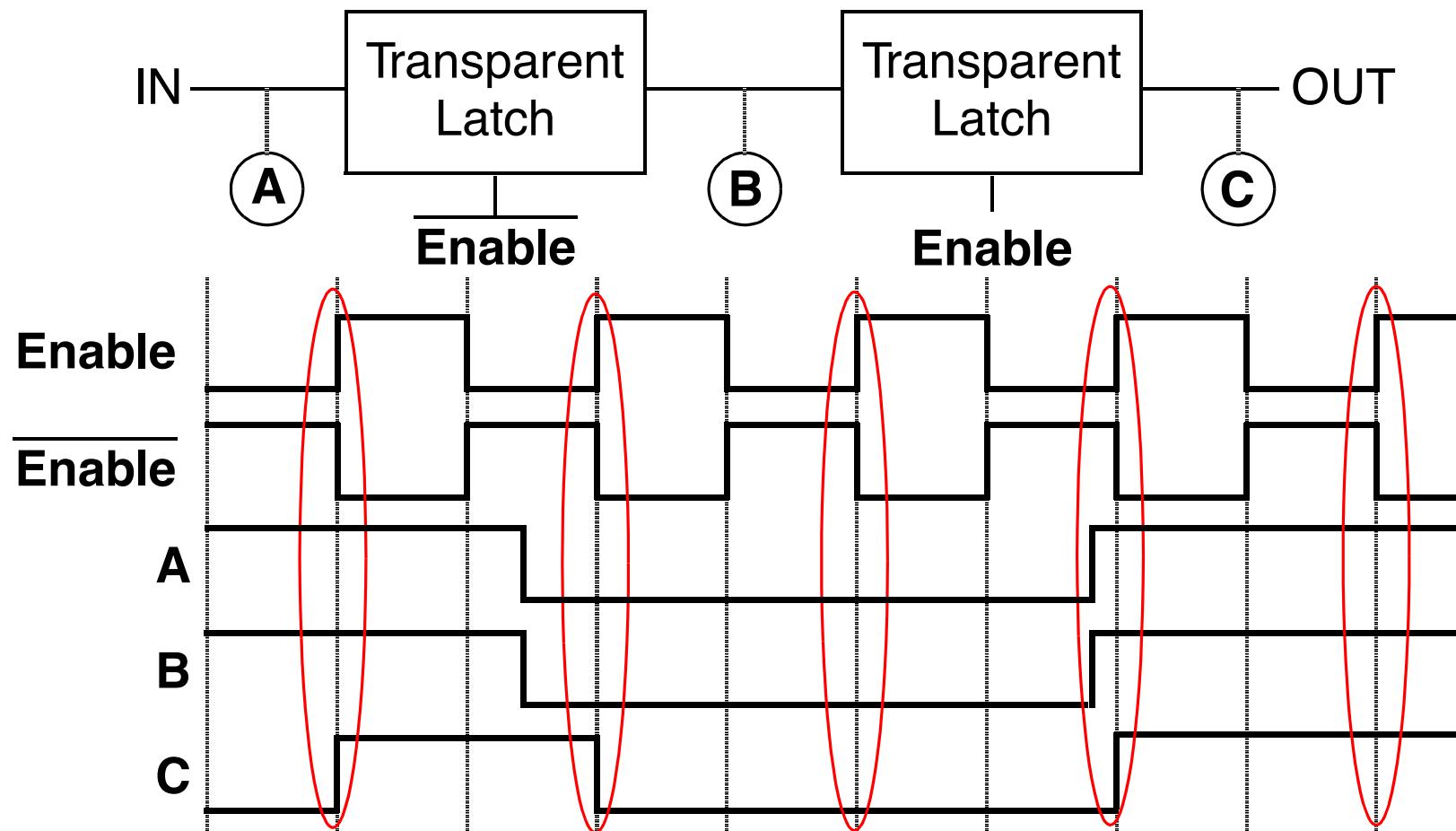
- The output C, which is also the bit stored, appears to change on the negative edge of the Enable transitions.



# FLIP-FLOPS

## POSITIVE EDGE TRIGGERED

- The output C, which is also the bit stored, appears to change on the positive edge of the Enable transitions.

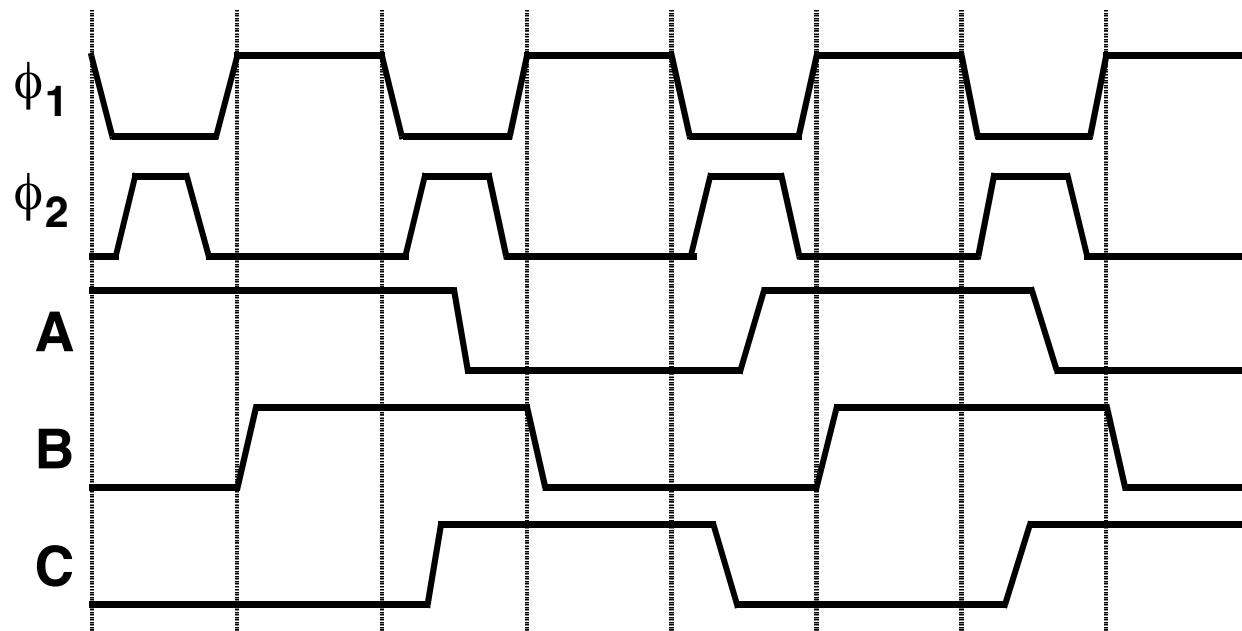


# FLIP-FLOPS

NON-IDEAL W/ DUAL-PHASE

- FLIP-FLOPS
  - EDGE TRIGGERED
  - NEG. EDGE TRIGGERED
  - POS. EDGE TRIGGERED

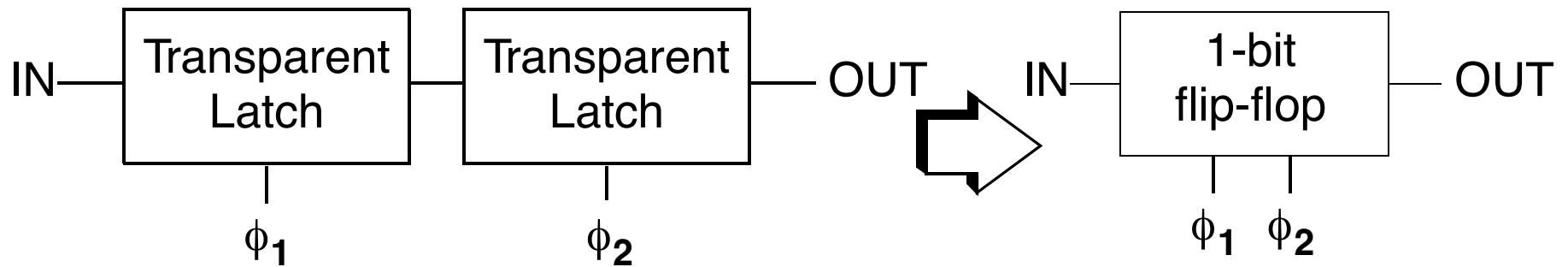
- The previous timing diagrams are in an ideal case. In reality, an implementation with delays might have the following timing diagram.



Propagation  
delays shift  
the outputs and  
slew transitions

- Notice that **Enable/Enable** are replaced with  $\phi_1/\phi_2$ , which are **non-overlapping** phases (normally generated from a dual-phase clock).

- Why use non-overlapping, dual-phase signals for the latch enable?
  - What happens if the latch enable input flip simultaneously?
  - How about if propagation delays cause one latch to change enable state slightly before the other?
  - The goal is to ensure that the **master latch has latched the input before the slave** latch tries takes this bit from the master.



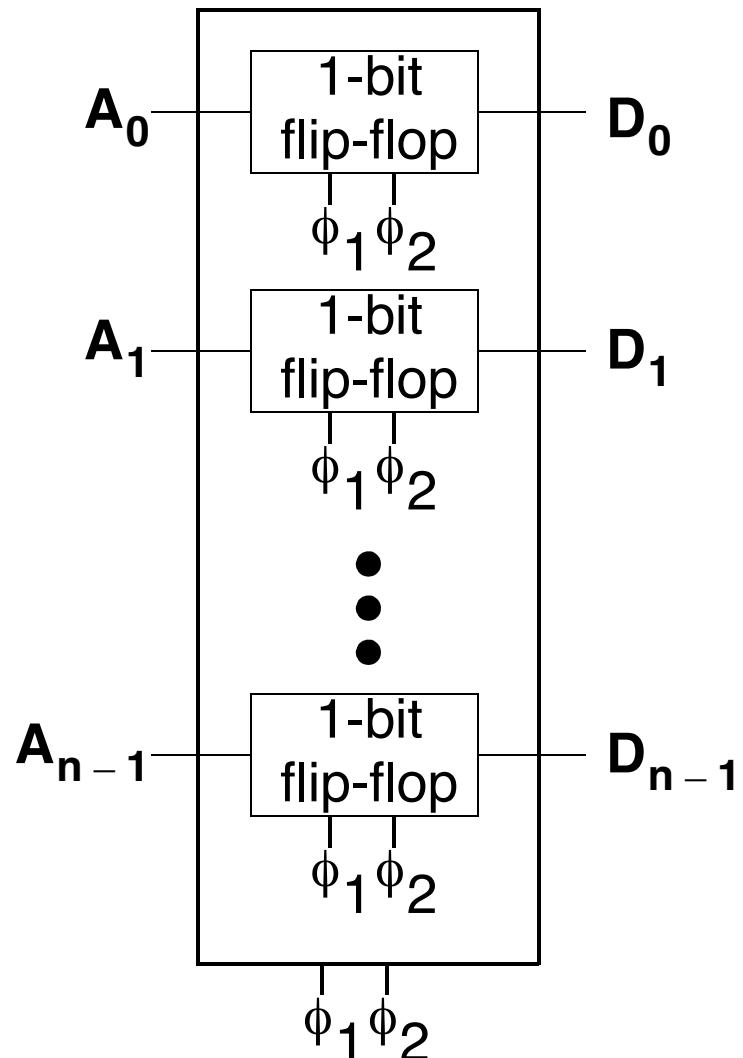
- If the master has not latched, the slave sees the input transparently!!!
- A non-overlapping, dual-phase enable solves this problem.

# REGISTERS

## REGISTERS FROM FLIP-FLOPS

- FLIP-FLOPS
  - POS. EDGE TRIGGERED
  - NON-IDEAL W/DUAL- $\phi$
  - DUAL-PHASE ENABLE

- In essence, a flip-flop is a 1-bit register.
- An n-bit register can be formed by groupign n flip-flops together.

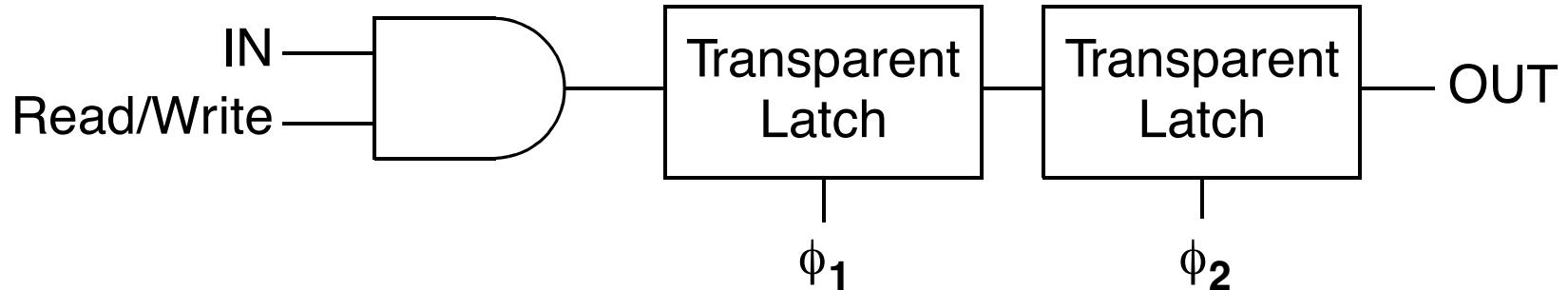


# REGISTERS

## READ/WRITE CONTROL (1)

- FLIP-FLOPS
- REGISTERS
- REGISTERS F/FLIP-FLOPS

- When a clock is used, such as the non-overlapping, dual-phase clock  $\phi_1$  and  $\phi_2$ , we want control over when a new value is written into a register (instead of writing a new value every clock cycle).
  - A read/write control is therefore required.
  - One “poor” design might be as follows for a 1-bit register.



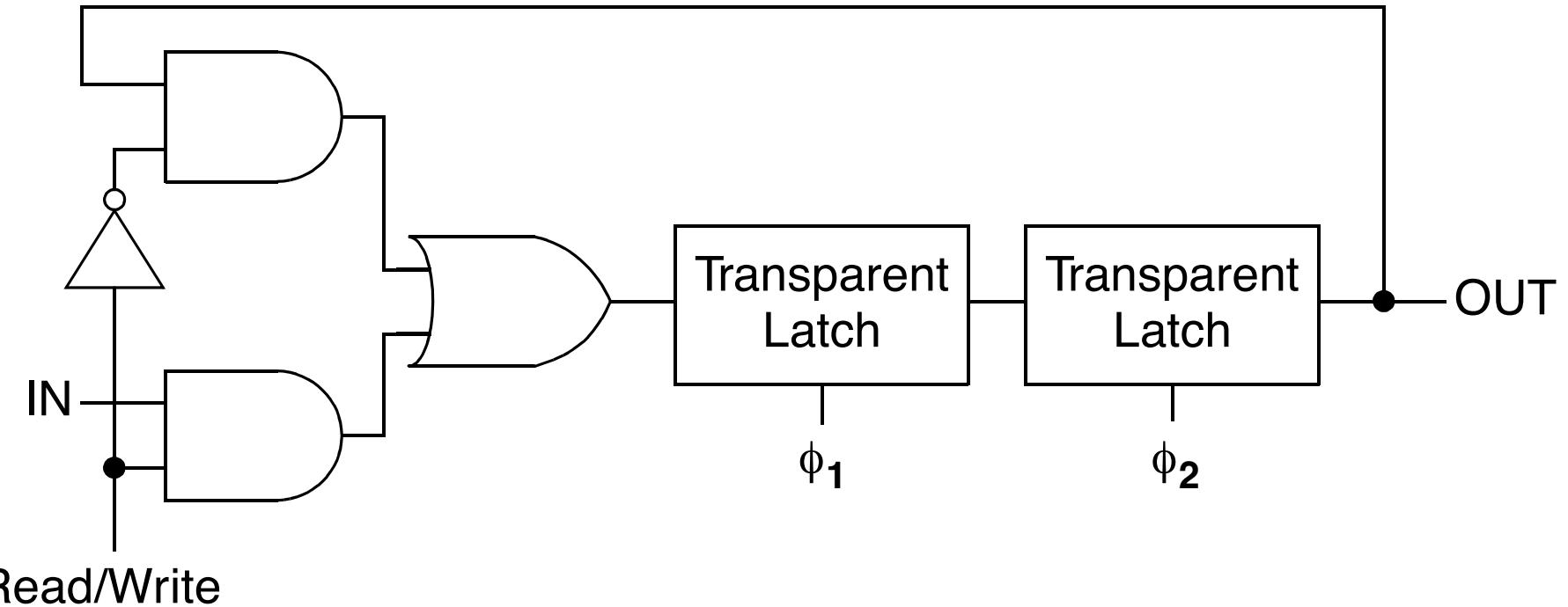
- What is the problem with this design?

# REGISTERS

## READ/WRITE CONTROL (2)

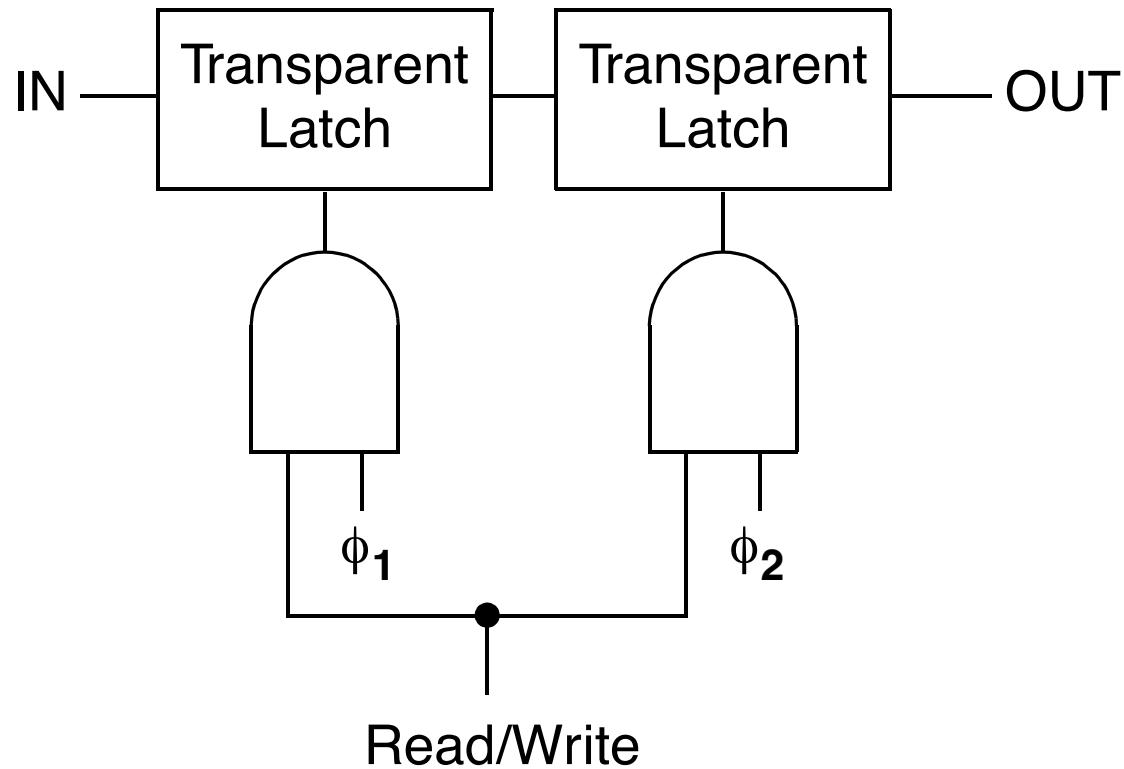
- FLIP-FLOPS
- REGISTERS
- REGISTERS F/FLIP-FLOPS
- READ/WRITE CONTROL

- A better design might be as follows



- When Read/Write = 0, the output is feed back into the master latch.
- When Read/Write = 1, the input is feed into the master latch.

- A different design approach might be as follows



- What problems might exist with this design?
  - One issue might be that both latch enables are 0 when R/W = 0.

**INTRO. TO COMP. ENG.  
CHAPTER XIII-1  
ISA**

**•CHAPTER XIII**

# **CHAPTER XIII**

## **INSTRUCTION SET ARCHITECTURE (ISA)**

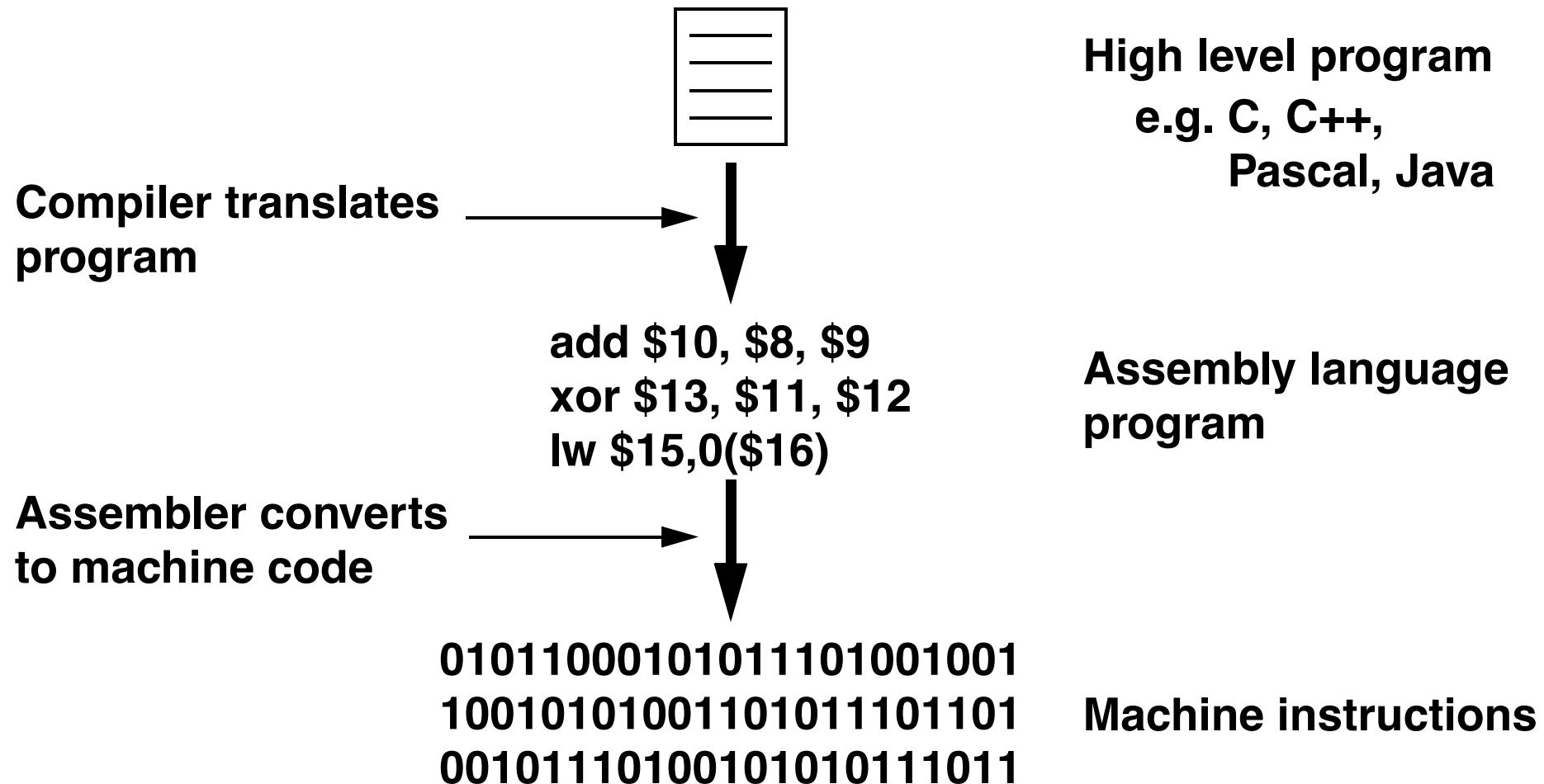
**READ INSTRUCTIONS FREE-DOC ON COURSE WEBPAGE**

# **ISA**

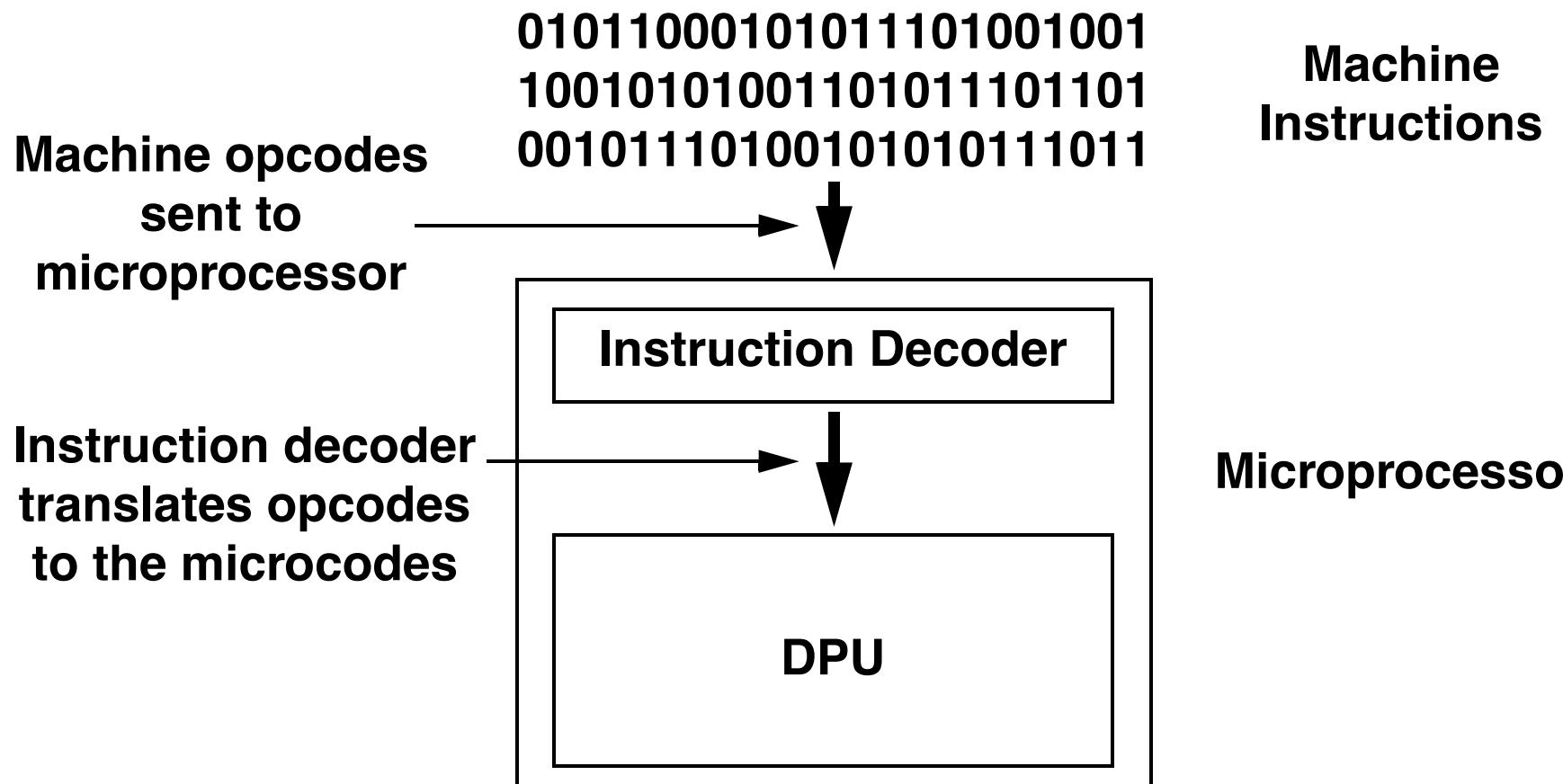
## **INTRODUCTION**

- We have now considered the beginnings of the internal architecture of a computer.
  - With this, we considered microcode operations for performing simple data routing and calculations in one clock cycle.
- As a programmer, we don't want to interface with the microprocessor and manually send each and every control signal as is done with microcode.
  - We would prefer to abstract the instruction sent to the microprocessor.
  - Let the microprocessor designer handle the decoding of the abstracted instruction into the microcode control operations.
- Start to define an assembly language! MIPS R3000/4000!

- Below is the process for translating a program to machine opcodes.



- Once the opcodes are given to the microprocessor, it translates the opcode instructions to the microcodes operations we discussed.



# MIPS ASSEMBLY

## MIPS REGISTER NAMES

- ISA
- PROGRAM PATH
  - TRANSLATING CODE
  - EXECUTING CODE

- For MIPS assembly, many registers have alternate names or specific uses.

Register	Name(s)	Use
0	\$zero	always zero (0x00000000)
1		reserved for assembler
2-3	\$v0-\$v1	results and expression evaluation
4-7	\$a0-\$a3	arguments
8-15	\$t0-\$t7	temporary values
16-23	\$s0-\$s7	saved values
24-25	\$t8-\$t9	temporary values
26-27		reserved for operating system
28	\$gp	global pointer
29	\$sp	stack pointer
30	\$fp	frame pointer
31	\$ra	return address

- Need to consider an assembly language example. We will use the MIPS R3000/4000 assembly so that you can refer to the Instruction free-doc.
- MIPS R3000/4000 assembly instruction format:
  - The *majority* of MIPS instructions have the following assembly language instruction format.
  - **<inst mnemonic> <destination>, <source 1>, <source 2>**
  - You can see that this instruction format fits the register transfer level notation discussed with the single cycle DPU

**R18 = R12 + R15**

destination                          source 1                  source 2

- Register format (R-format) instructions
  - Many MIPS instructions have the following format for register to register type binary operations.
    - **<instr> \$<write register>, \$<read register 1>, \$<read register 2>**
  - An example of this is
    - **add \$10, \$8, \$9**
  - This is the same as with our register transfer level operation
    - **R10 = R8 + R9**

# MIPS ASSEMBLY

## REGISTER INSTRUCTIONS

- Below is the basic list of register format MIPS instructions.

Instruction	Interpretation
<b>add \$10, \$8, \$9</b>	<b>R10 = R8 + R9</b>
<b>sub \$10, \$8, \$9</b>	<b>R10 = R8 - R9</b>
<b>and \$10, \$8, \$9</b>	<b>R10 = R8 and R9</b>
<b>or \$10, \$8, \$9</b>	<b>R10 = R8 or R9</b>
<b>xor \$10, \$8, \$9</b>	<b>R10 = R8 xor R9</b>
<b>sa \$10, \$8, \$9 (shift arithmetic)</b>	<b>Shift R8 by R9 and store in R10</b>
<b>sl \$10, \$8, \$9 (shift logical)</b>	<b>Shift R8 by R9 and store in R10</b>
<b>rot \$10, \$8, \$9 (rotate)</b>	<b>Rotate R8 by R9 and store in R10</b>
<b>lw \$10, 0(\$8)</b>	<b>R10 = M[0+R8]</b>
<b>sw \$10, 0(\$8)</b>	<b>M[0+R8] = R10</b>

- Immediate format (I-format) instructions
  - Many MIPS instructions have the following format for register to register type binary operations.
    - **<instr> \$<write register>, \$<read register>, <immediate value>**
  - An example of this is
    - **addi \$10, \$8, 4**
  - This is the same as with our register transfer level operation
    - **R10 = R8 + 4**

Note: No \$ for last argument

Again, no \$ for immediate value

Note: Include "i" to indicate an immediate value is used.

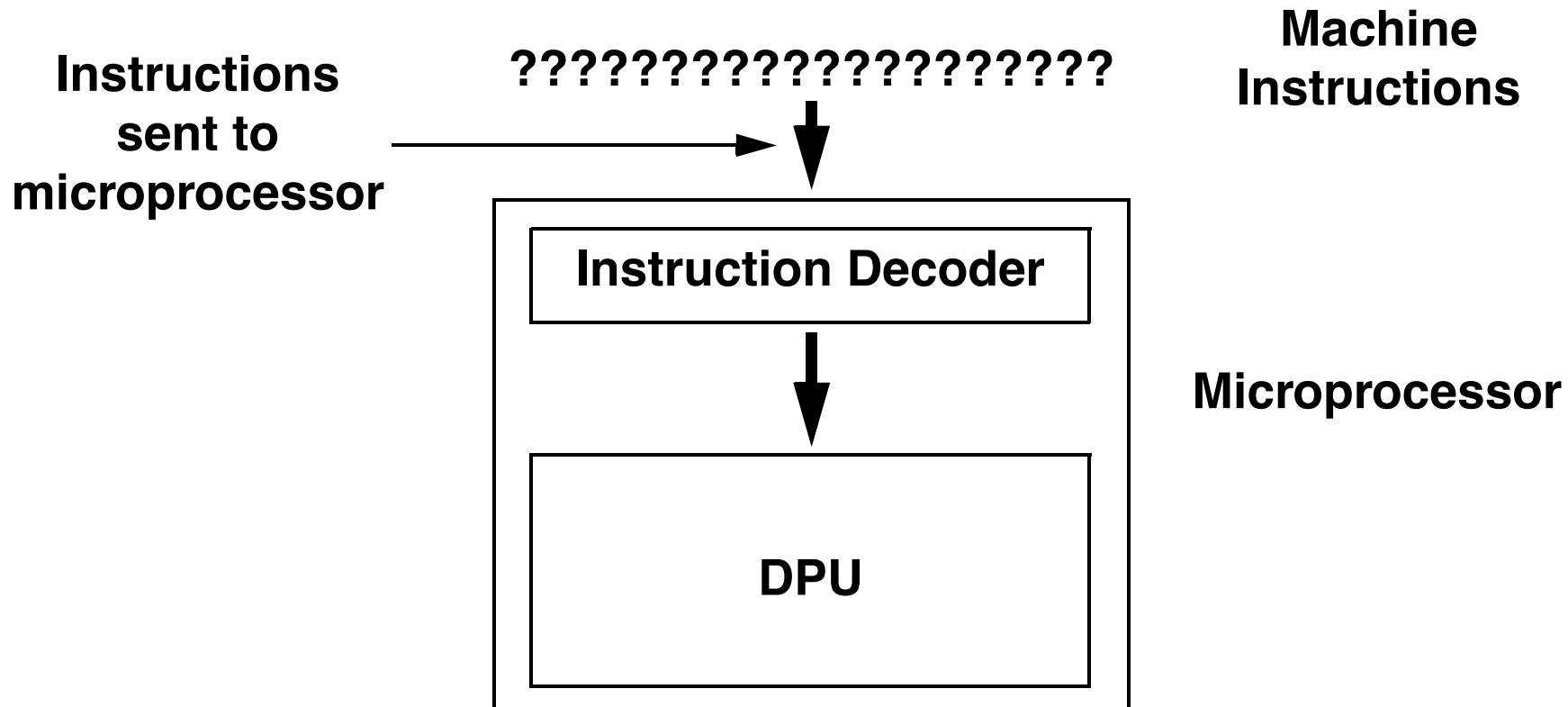
# MIPS ASSEMBLY

## IMMEDIATE INSTRUCTIONS

- Below is the basic list of immediate format MIPS instructions.

Instruction	Interpretation
<b>addi \$10, \$8, 4</b>	<b>R10 = R8 + 4</b>
<b>subi \$10, \$8, 4</b>	<b>R10 = R8 - 4</b>
<b>andi \$10, \$8, 4</b>	<b>R10 = R8 and 4</b>
<b>ori \$10, \$8, 4</b>	<b>R10 = R8 or 4</b>
<b>xori \$10, \$8, 4</b>	<b>R10 = R8 xor 4</b>
<b>sai \$10, \$8, 4 (shift arithmetic)</b>	<b>Shift R8 by 4 and store in R10</b>
<b>sli \$10, \$8, 4 (shift logical)</b>	<b>Shift R8 by 4 and store in R10</b>
<b>roti \$10, \$8, 4 (rotate)</b>	<b>Rotate R8 by 4 and store in R10</b>
<b>lw \$10, 4(\$0)</b>	<b>R10 = M[4+R0]</b>
<b>sw \$10, 4(\$0)</b>	<b>M[4+R0] = R10</b>

- How should the assembly be translated to machine code?



- Have to consider what control signals the DPU requires!
- How do we abstract from the DPU's requirements?

- First important part of a machine instruction is known as the operational codes (opcodes).
  - An **opcode** indicates what **major operation** to perform.
    - Example major operations:  
add, subtract, AND, OR, NOT, XOR, shift
    - Once all major operations are identified for a processor design, **assign binary codes** to each of the operation.
      - For example, say that we want to design a machine that can perform 40 different types of major operations.
      - Then we would require at least 6 bits to represent all of the opcodes.

- Some example opcodes used in the MIPS processors are as follows.

Instruction	Assigned Opcode Value
<b>add \$10, \$8, \$9</b>	<b>100000</b>
<b>sub \$10, \$8, \$9</b>	<b>100010</b>
<b>and \$10, \$8, \$9</b>	<b>100100</b>
<b>or \$10, \$8, \$9</b>	<b>100101</b>
<b>lw \$10, 0(\$8)</b>	<b>100011</b>
<b>sw \$10, 0(\$8)</b>	<b>101011</b>
<b>addi \$10, \$8, 4</b>	<b>001000</b>
<b>nop (no operation)</b>	<b>000000</b>

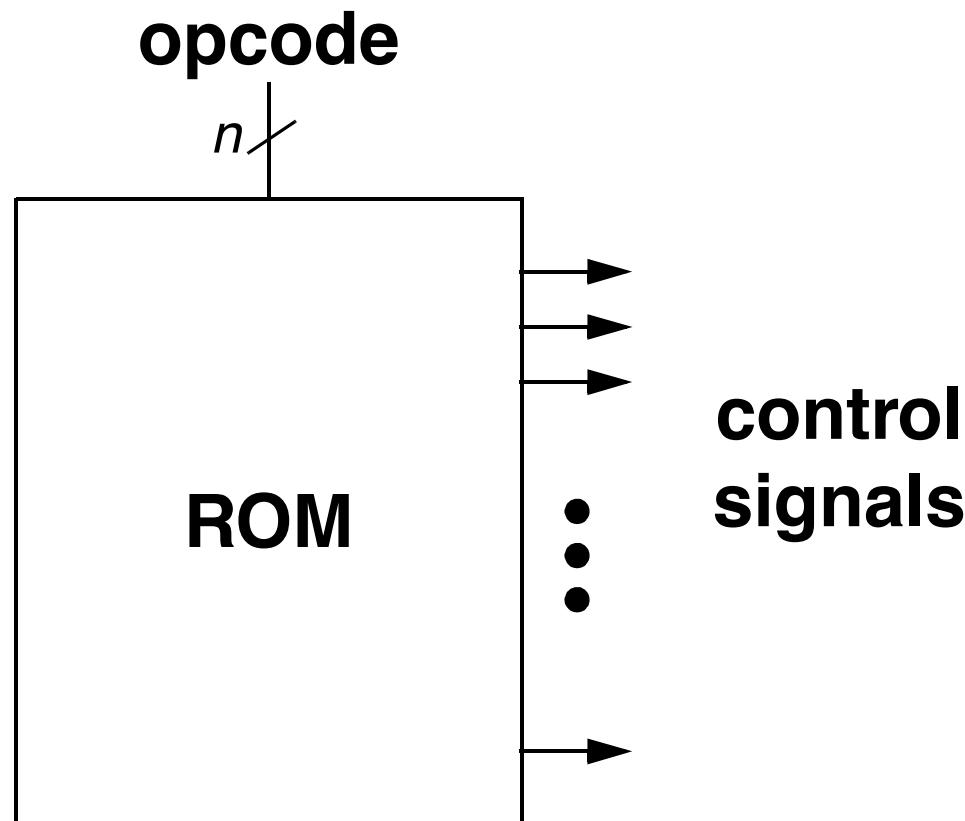
- Note: Different opcodes for **add** and **addi**. Why?

- Once you have assigned opcodes to all of your major functions, now need to decode the opcodes to the appropriate controller signals.
  - i.e. we no longer want to control the DPU manually.
- Recall that we used the following DPU signals when performing

$$R10 = R8 + R9$$

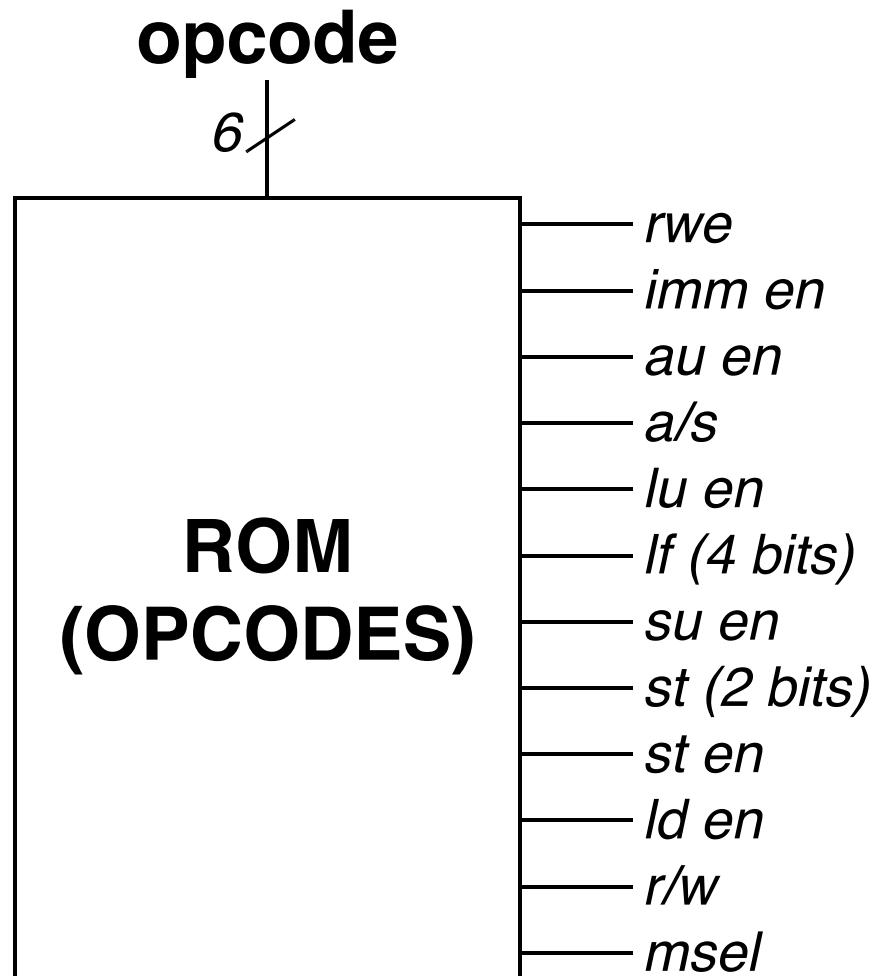
- $\bar{a}/s = 0$  and  $en = 1$  for AU.
- $en = 0$  for LU,  $en = 0$  for SU.
- $st\_en = 0$ ,  $ld\_en = 0$ ,  $r/w = X$ ,  $msel = 0$ .
- $X_{ra} = 01000$ ,  $Y_{ra} = 01001$ ,  $Z_{wa} = 01010$ , and  $rwe = 1$  for RF.
- Note: We will pass  $X_{ra}$ ,  $Y_{ra}$ , and  $Z_{wa}$  from the outside.
- Refer to Table 3 in Instruction free-doc for other examples.

- In general, these control signals can be burned into a ROM.

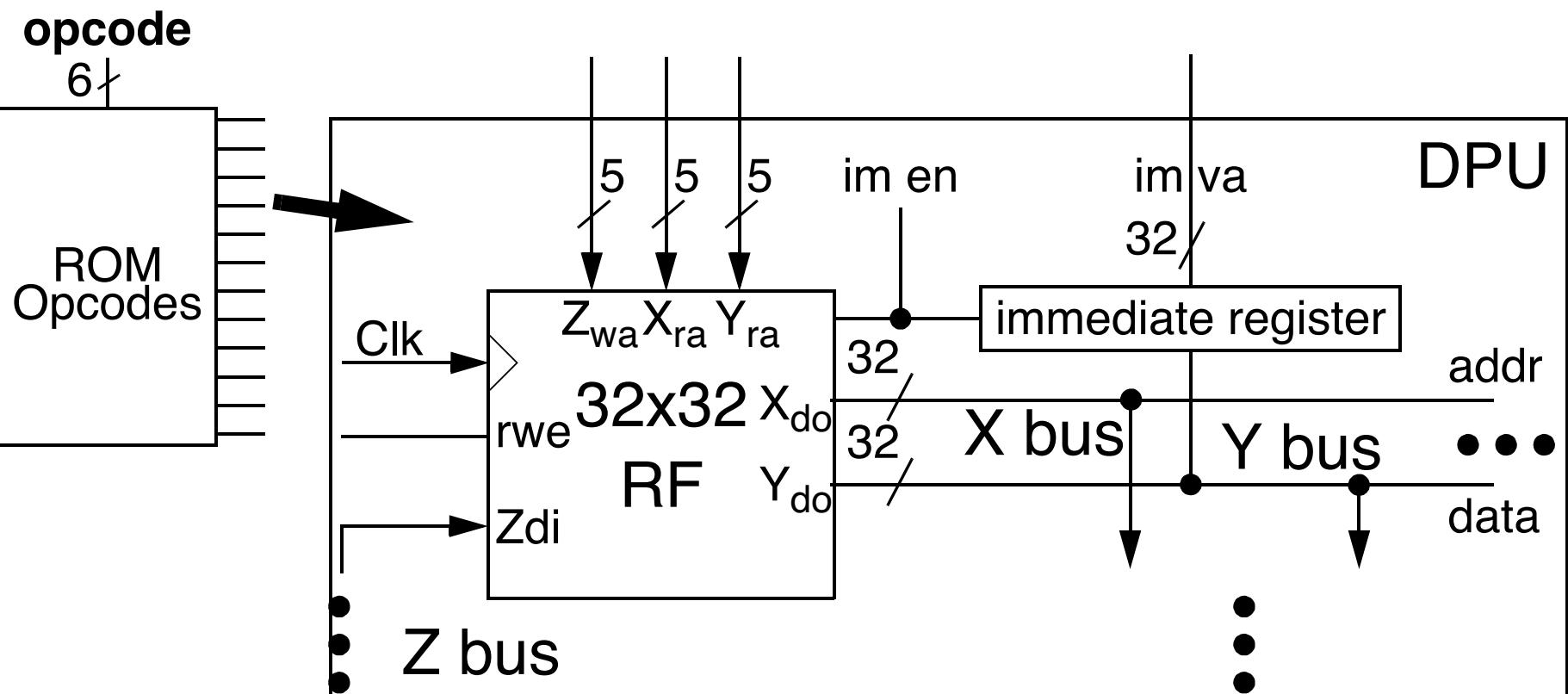


- Each opcode has its own set of general control signals for the DPU.

- For our DPU, the control signals are as follows.

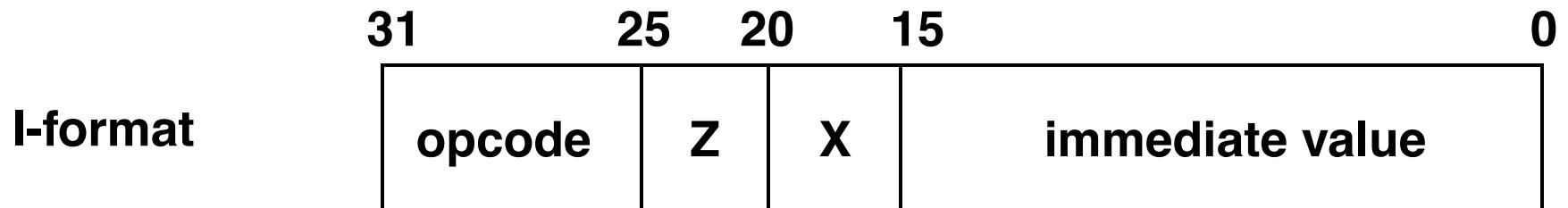
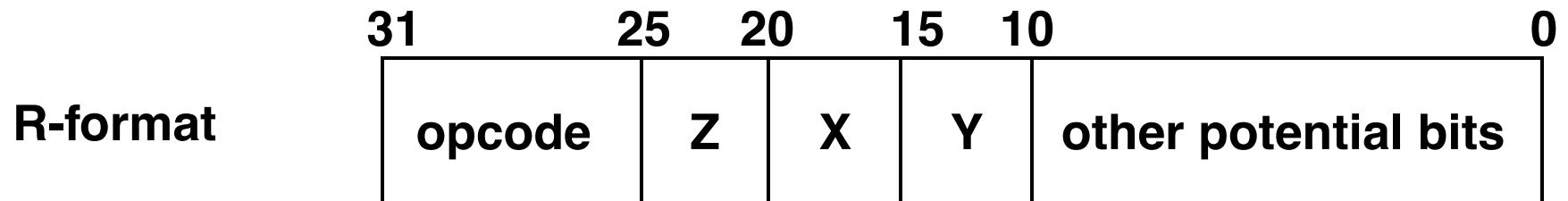


- Now, an input opcode will send appropriate control signals to the DPU for that major operation.



- Notice that we still need register addresses and the immediate value.

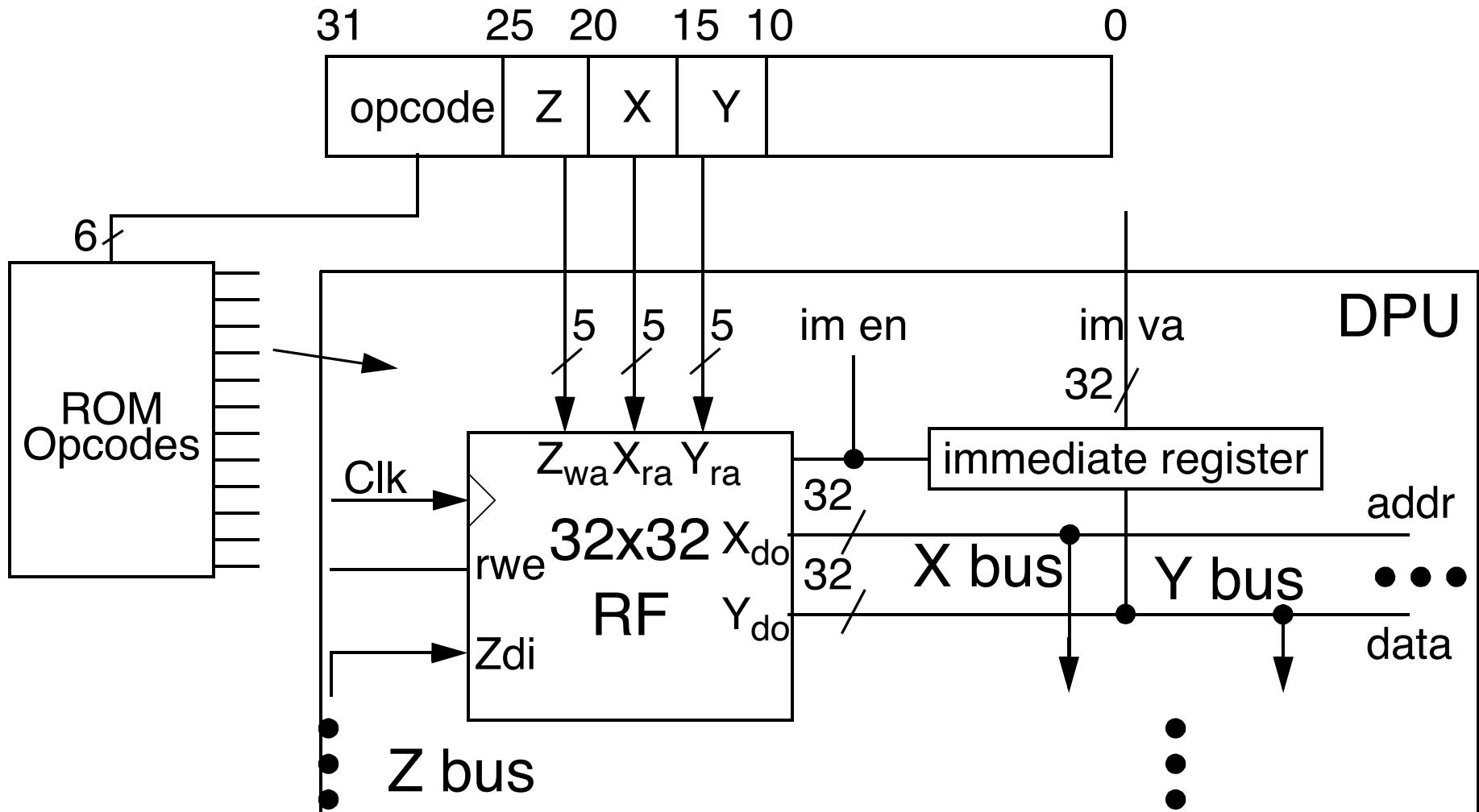
- While instructions can come in many different shapes and forms, we will consider the following 32-bit instruction formats to loosely follow the MIPS R3000/4000 format.



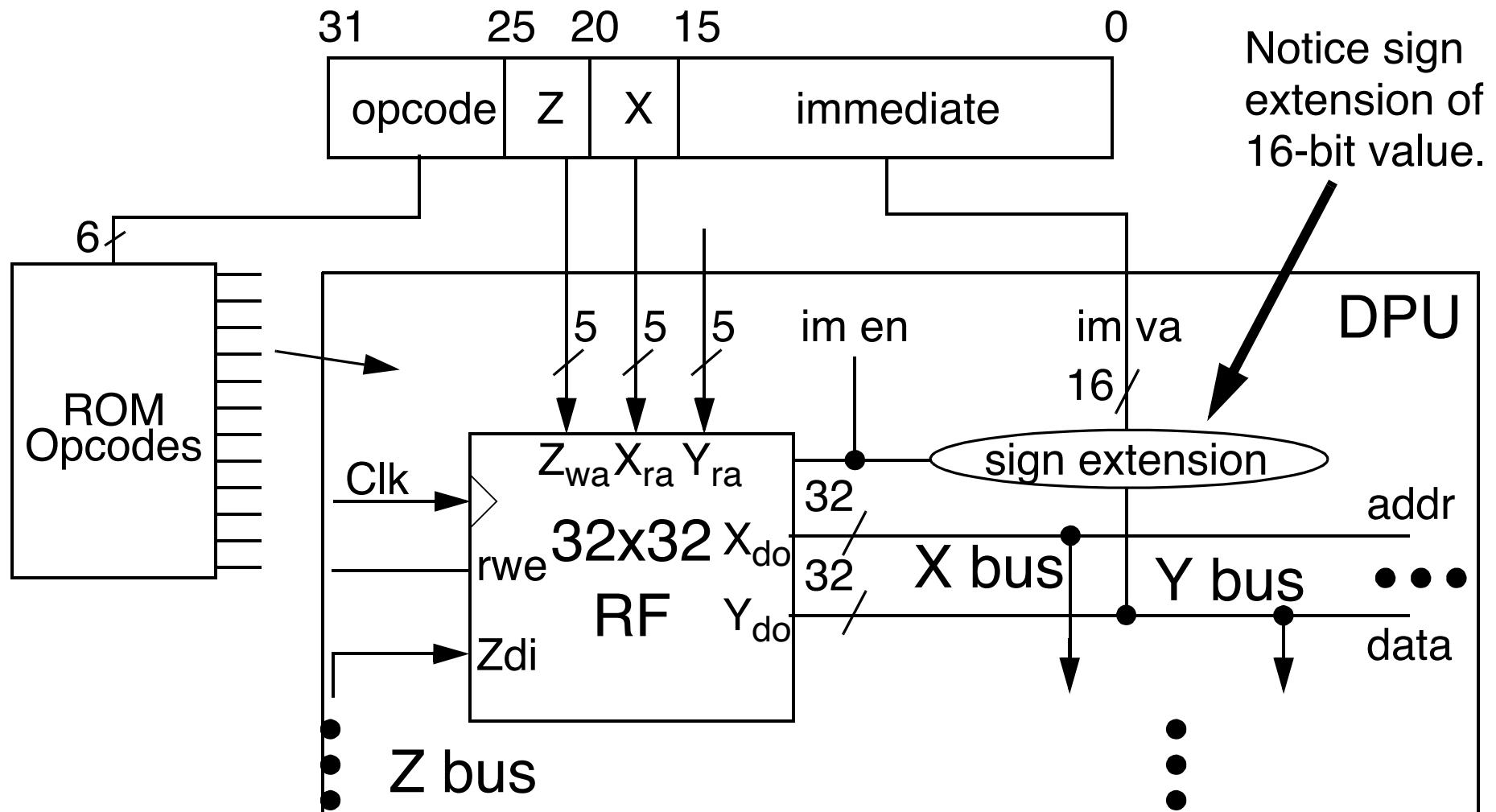
# INSTRUCTIONS

## R-FORMAT W/ DPU

- If we have an R-format instruction, we link the bits as follows.



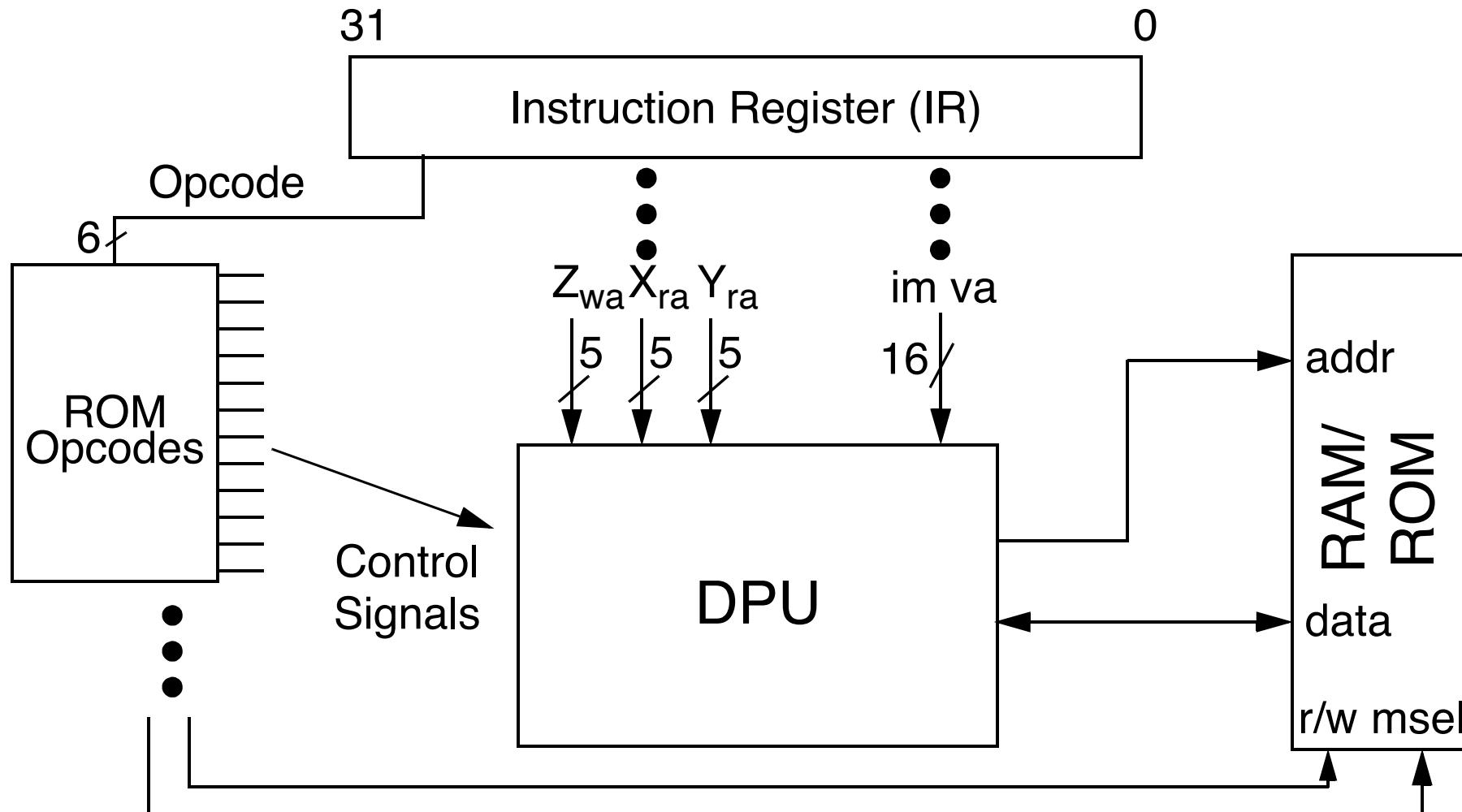
- If we have an I-format instruction, we link the bits as follows.



# INSTRUCTIONS

## INSTRUCTION REGISTER

- Use a general instruction register that can act as R- or I-Format.



**INTRO. TO COMP. ENG.  
CHAPTER XIV-1  
PROGRAM CONTROL**

**•CHAPTER XIV**

## **CHAPTER XIV**

# **PROGRAM CONTROL, JUMPING, AND BRANCHING**

**READ BRANCHING FREE-DOC ON COURSE WEBPAGE**

# PROGRAM CONTROL

## INTRODUCTION

- So far we have discussed how the instruction set architecture for a machine can be designed.
- Another important aspect is how to control the flow of a program execution.
  - What order should instructions be executed?
  - Are there times when we need to change the order of instruction execution?
  - How do we handle changes of the program flow and decide when to change the program flow?

# PROGRAM CONTROL

## PROGRAM COUNTER (PC)

- How should a program or list of instructions be executed?

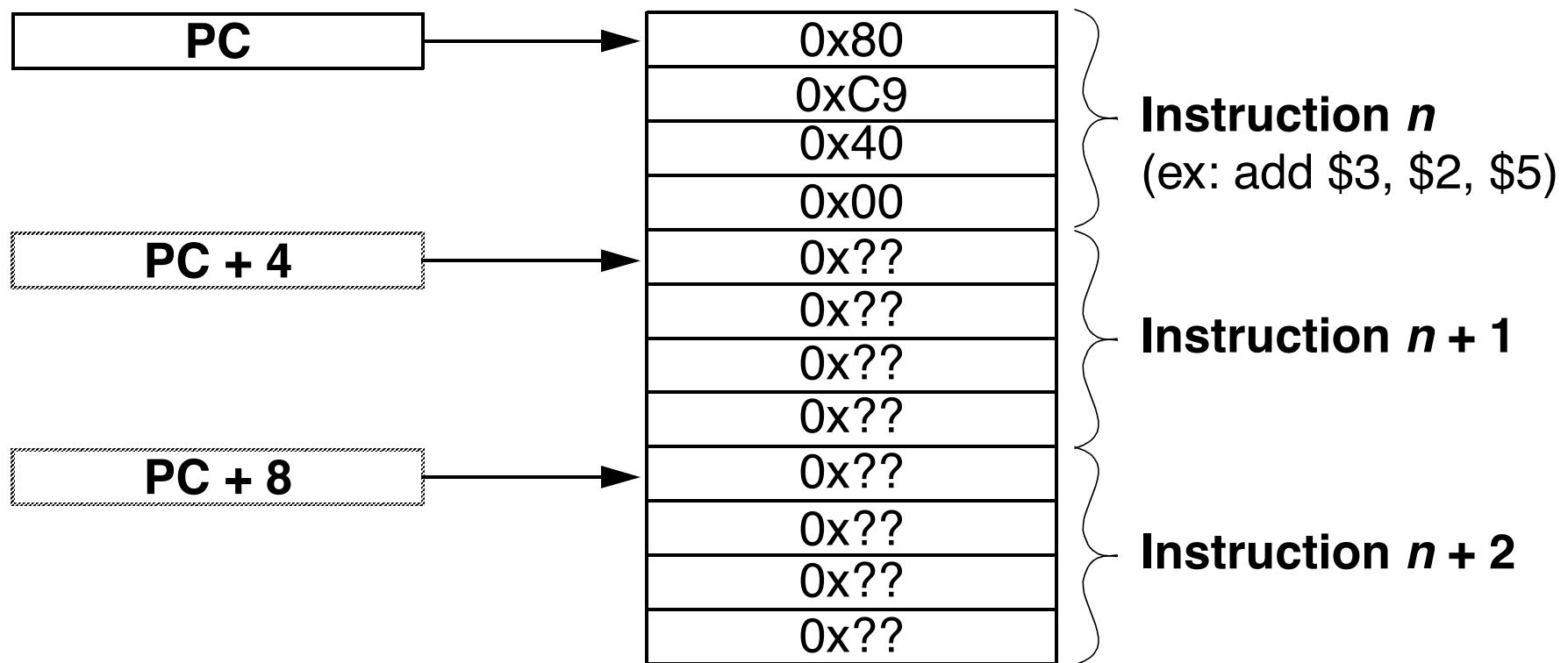
- The most obvious choice is to execute the 32-bit instruction words in sequential order.

PC	→	1st	lw \$2, 0x00001004(\$0)
		2nd	addi \$15, \$2, 0x00201003
		3rd	xor \$13, \$15, \$2
		4th	add \$3 \$13 \$2
		5th	sai \$18, \$3, 0x00000004
		6th	sw \$18, 0x00001003(\$0)
		...	...

Standard  
Order of  
Execution

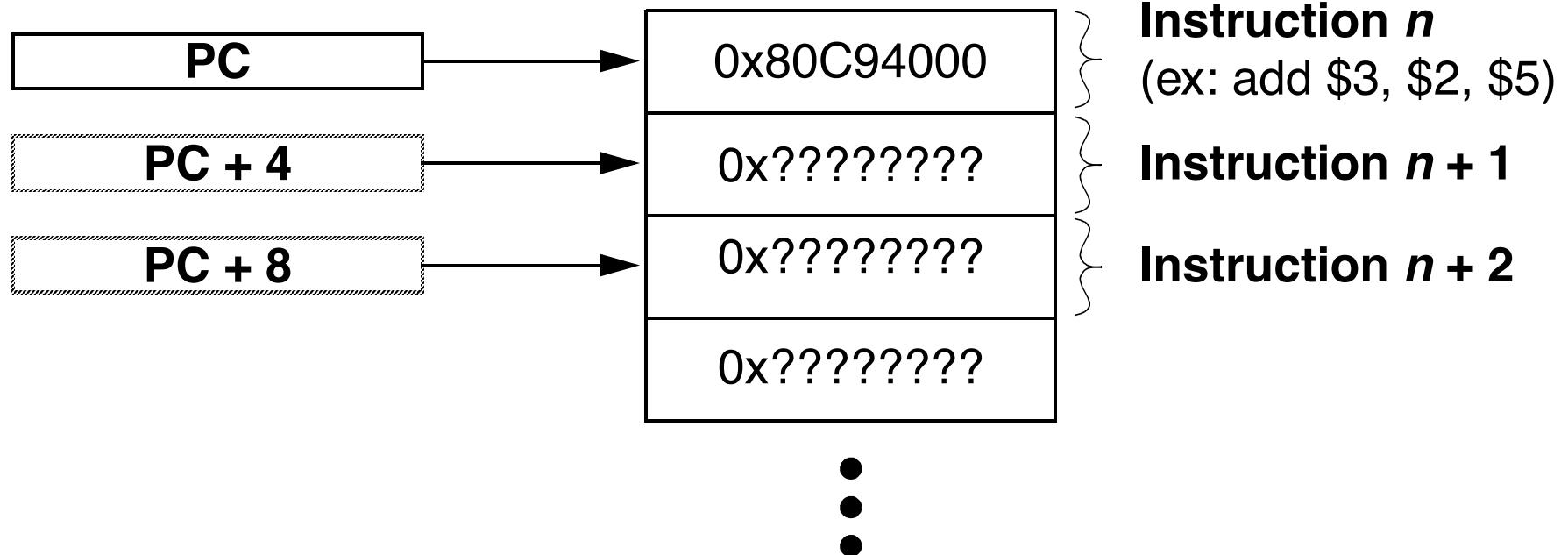
- Would be useful to have a pointer to the next instruction.
- We will call this the program counter (**PC**).

- We can consider the program counter as pointing into memory at the next instruction to be executed.



- Instructions are 32-bits (4 bytes), so add 1 to get next instruction.

- To make the memory map representation a little more compact, we will make each address location 32-bits with the **PC** incremented by 4..



**Instruction *n***  
(ex: add \$3, \$2, \$5)

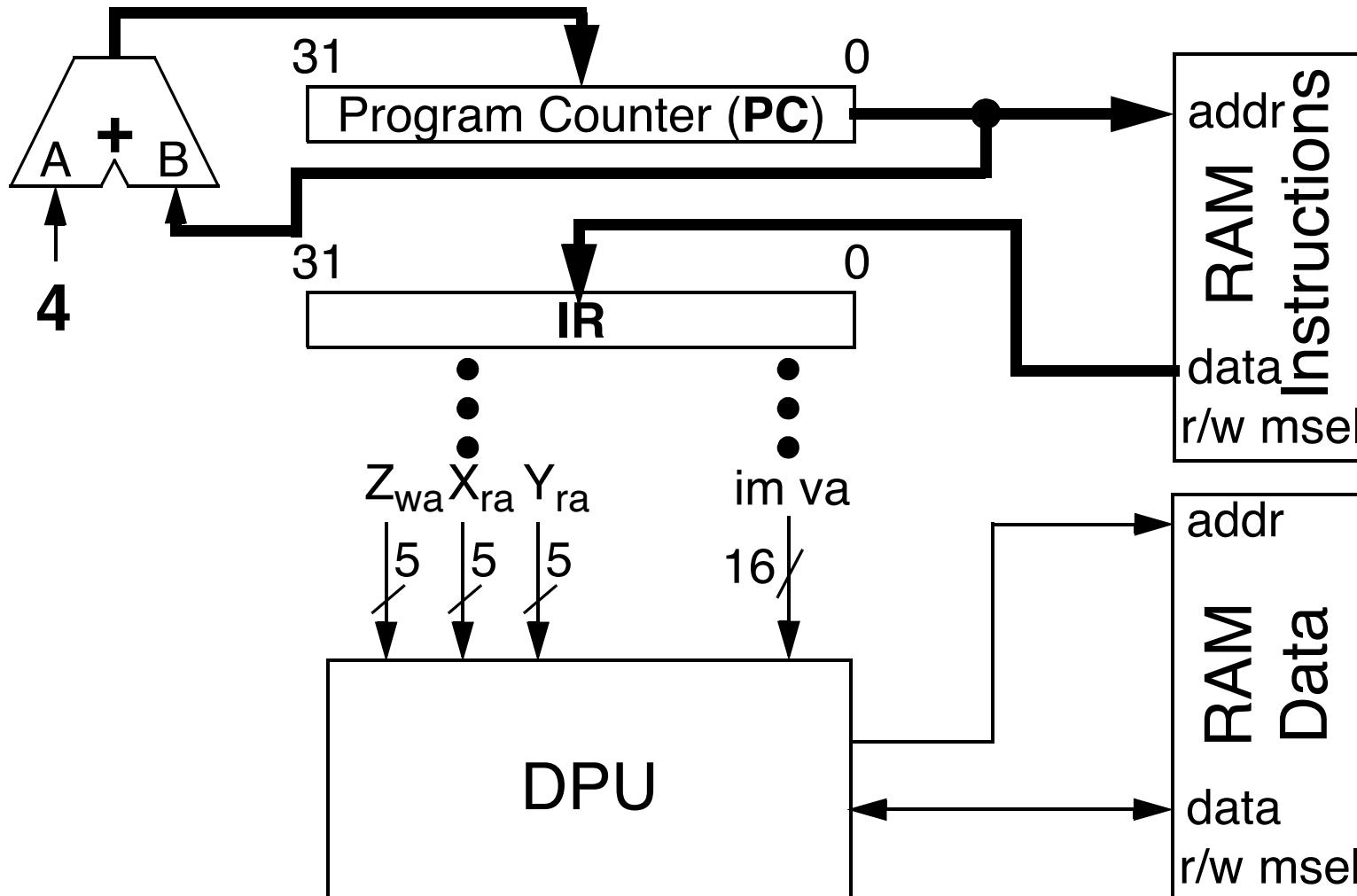
**Instruction *n + 1***

**Instruction *n + 2***

# PROGRAM CONTROL

## PC IN SINGLE CYCLE DPU

- The **PC register** can be added as follows to our single cycle DPU.



- **At the beginning of the clock cycle**
  - Current contents of **IR** used and decoded as the current instruction.
  - **PC** addresses the instruction memory to fetch the next instruction.
  - The next instruction is output from the instruction memory and applied to the input of the **IR**, though, not loaded until the end of the clock cycle.
  - **PC + 4** is calculated and applied to the **PC**, though, not loaded until the end of the clock cycle. A **+4** is used so that the next 32-bit (4-byte) word is addressed which is the next instruction to be addressed.
- **At the end of the clock cycle.**
  - The next instruction is clocked into the **IR**.
  - The address for the following instruction is clocked into the **PC**.

# PROGRAM CONTROL

## CHANGING PROGRAM FLOW

- While executing instructions in **sequential order** is a good **default mode**, it is desirable to be able to **change the program flow**.
  - Two main classifications for deviation from sequential order are
    - **absolute** versus **relative** instruction addressing
    - **conditional** versus **unconditional** branching/jumping
  - and
- The MIPS R3000/4000 uses only
  - **unconditional absolute instruction addressing** and
  - **conditional relative instruction addressing**

- **Absolute instruction addressing**, generally known as **jumping**.
  - A **specific address**, or **absolute address**, is given where the next instruction is located.
    - **PC = address**
  - This allows execution of any instruction in memory.
  - Jumps are good if you have a piece of code that will not be relocated to another location in memory.
    - For instance, ROM BIOS code that never moves.
    - Main interrupt service routines that will always be located in a set instruction memory location.
  - Different MIPS instructions will use byte or word addressing such that
    - **PC = byte\_address** or **PC = (word\_address<<2)**

- **Relative instruction addressing**, generally known as **branching**.
  - An **offset to the current address** is given and the next instruction address is calculated, in general, as **PC = PC + byte\_offset**.
  - For MIPS, and many other processors, since **PC** has already been updated to **PC + 4** when loading in the current instruction, it is actually calculated as
    - **PC = PC + inst\_size + inst\_offset = PC + 4 + (word\_offset << 2)**
  - Note that the offset can be **positive** or **negative**.
  - Useful since a program can therefore be loaded anywhere in the instruction memory and still function correctly.
    - Move a program around in memory, and it can still branch within itself since the branching is relative to the current **PC** value.

- For **unconditional** program control instructions
  - The absolute **jump** or relative **branch** is **ALWAYS performed** when that instruction is encountered.
- For **conditional** program control instructions
  - A **condition** is first tested.
    - If the result is **true**, then the branch/jump is taken.
      - **PC = byte\_address** or **PC = (word\_address<<2)** for a jump or
      - **PC = PC + 4 + (word\_offset<<2)** for a branch.
    - If the result is **false**, then the branch/jump is NOT taken and program execution continues
      - ie. **PC = PC + 4**.

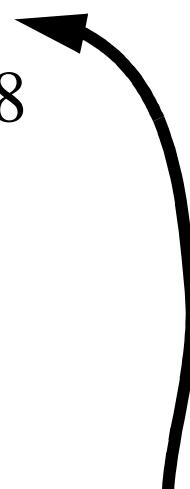
- The first form of program control is the **absolute jump** is as follows
  - **jr <register>**
  - The **jr** instruction changes **PC** to the value contained in the **register**.
  - For example, if **R10** contains **0x00004400** then after executing the following **jr** instruction, the next instruction executed is the **add**.

PC → 0x00001000 0x00001004 ... next PC → 0x00004400 0x00004404 ...	jr \$10 sub \$15, \$2, \$8 ... add \$3 \$13 \$2 ... ...
---	--

- We can also have an **immediate** form of the **jump instruction**
  - **j <instruction address>**
  - The **j** instruction changes **PC** to the given **instruction address**.
  - For example, with the following **j** instruction, the next instruction executed is the **add**.

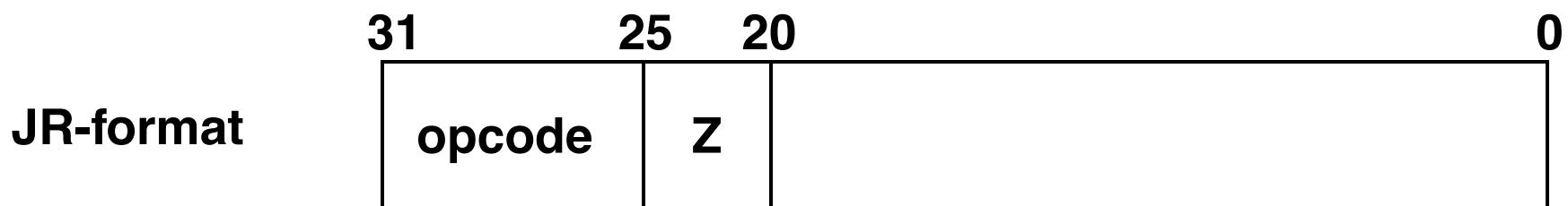
PC → 0x00001000  
0x00001004  
...  
next PC → 0x00004400  
0x00004404  
...

j 0x00004400  
sub \$15, \$2, \$8  
...  
add \$3 \$13 \$2  
...  
...



Note: assembler will convert to 0x0001100 so that  $0x0001100 \ll 2 = 0x00004400$ .

- Both jump instructions have **one implied destination**, the **PC**, and one source, either a **register** or an **immediate value**.
- We therefore need some new instruction formats.
  - The **jr** instruction can essentially use the **R-format**, but need the **jr** opcode route **Z<sub>wa</sub>** to **Y<sub>ra</sub>** and route **Y bus** to the **PC** so that the address in the register is loaded in the **PC**.

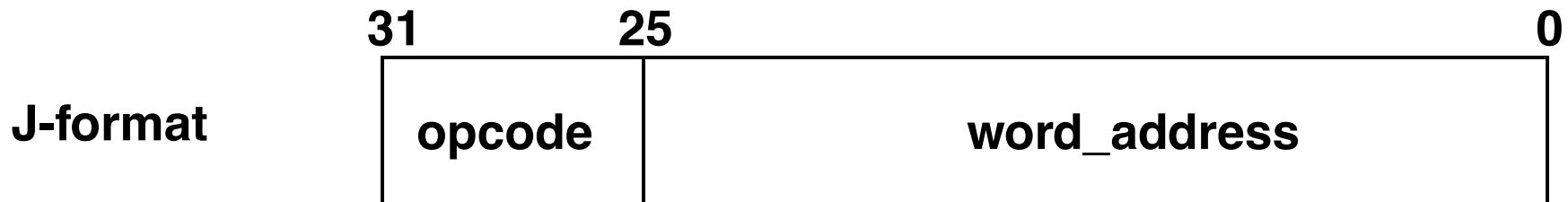


- The jump can go anywhere in memory using the 32-bit register value.

# JUMPING

## J-FORMAT

- The **j** instruction only needs the opcode and the immediate address for the new value of the **PC**.
  - Unfortunately, the **PC** is **32 bits** and using a **6-bit opcode**, this **leaves only 26 bits** in our 32-bit instruction.



- If we assume the immediate address is for **4 byte words**, then our **26-bits** can effectively address **28-bit bytes**.
- Update **PC** with **PC[27:0] = (word\_address<<2)** leaving **PC[31:28]** unchanged. Therefore, cannot jump anywhere in memory, but almost.

## JUMPING ASSIGNED OPCODES

- As seen, the MIPS R3000/4000 has two basic forms of a jump or absolute instruction addressing.

Instruction	Assigned Opcode	Interpretation
j 0x00004028	000010	<b>The next instruction fetched is at address 0x00004028.</b> <b>Restriction: address is a 26-bit address to 4 byte words.</b>
jr \$10	000011	<b>The next instruction fetched is at the 32-bit address stored in R10</b>

- As mentioned, branching uses an offset from the current instruction to determine the next instruction.
- For the MIPS, the only branching is with conditional branches.
  - Conditional branches typically compare two items, such as two registers, to test a condition.
  - This comparison is usually done by simply subtracting one number from the other and setting the appropriate **N**, **V**, **C**, **Z** flags.
  - ie. for MIPS
    - **<branch mnemonic> <register 1> <register 2> <branch offset>**
    - Here, the calculation **<register 1> - <register 2>** is performed with the flags **N**, **V**, **C**, **Z** set accordingly (the subtraction result is not stored).

# BRANCHING

## BRANCH TYPES

- Below is a list of some possible branch types (many of these do not exist for the MIPS R3000/4000).

Common Mnemonics	Branch Type	Flags
<b>beq</b>	<b>Branch if equal</b>	<b>Z = 1</b>
<b>bne or bnq</b>	<b>Branch if not equal</b>	<b>Z = 0</b>
<b>bpl</b>	<b>Branch if positive</b>	<b>N = 0</b>
<b>bmi</b>	<b>Branch if negative</b>	<b>N = 1</b>
<b>bcc</b>	<b>Branch on carry clear</b>	<b>C = 0</b>
<b>bcs</b>	<b>Branch on carry set</b>	<b>C = 1</b>
<b>bvc</b>	<b>Branch on overflow clear</b>	<b>V = 0</b>
<b>bvs</b>	<b>Branch on overflow set</b>	<b>V = 1</b>

# BRANCHING

## BRANCH TYPES

- continued...

Common Mnemonics	Branch Type	Flags
blt	Branch on less than	$N \oplus V$
ble	Branch on less than or equal	$Z + (N \oplus V)$
bge	Branch on greater than or equal	$\overline{N \oplus V}$
bgt	Branch on greater	$\overline{Z} + (N \oplus V)$
bra	Branch always	No flags needed
bsr	Branch to subroutine	No flags needed

- One MIPS instruction is the **branch if equal (beq)** instruction that checks if the contents of two registers are equal and branches if they are equal.
- For example, consider the following code

PC → start: beq \$1, \$2, skip  
(false) next PC → sub \$15, \$2, \$8  
R1 != R2  
...  
...

(true) next PC → skip: add \$3 \$13 \$2  
R1 = R2  
...  
...

- Notice that the branch is taken if **\$1 = \$2**.

# BRANCHING

## BRANCH IF NOT EQUAL

- Another MIPS instruction is the **branch if not equal (bne)** instruction that checks if two registers are **NOT** equal.
- For example, consider the following code

PC → start: bne \$1, \$2, skip  
(false) next PC → sub \$15, \$2, \$8  
R1 = R2  
...

(true) next PC → skip: add \$3 \$13 \$2  
R1 != R2  
...  
...

- Notice that the branch is taken if **\$1 != \$2**.

# BRANCHING

## ASSIGNED OPCODES

- BRANCHING
  - BRANCH TYPES
  - BRANCH IF EQUAL
  - BRANCH IF NOT EQUAL

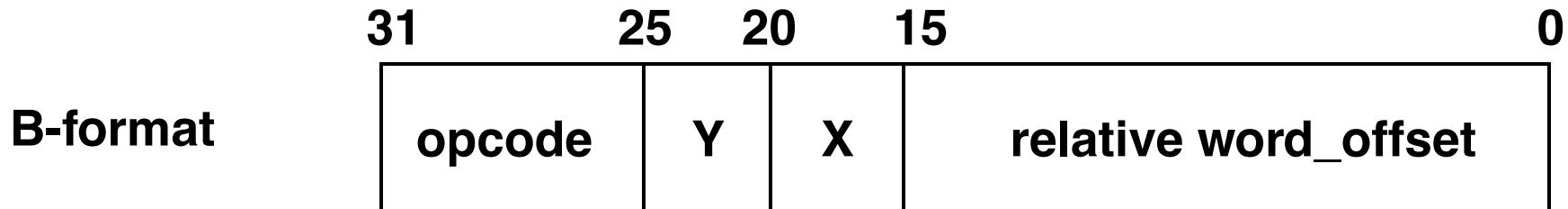
- As seen, the MIPS R3000/4000 has two basic forms of a jump or absolute instruction addressing.

Instruction	Opcode	Interpretation
<b>beq \$10, \$8, label</b>	<b>000100</b>	If contents of R10 is equal to contents of R8, next instruction that is fetched is the instruction labeled “label”. Otherwise, the next instruction fetched is after the beq.
<b>bne \$10, \$8, label</b>	<b>000101</b>	If contents of R10 is not equal to contents of R8, next instruction that is fetched is the instruction labeled “label”. Otherwise, the next instruction fetched is after the bne.

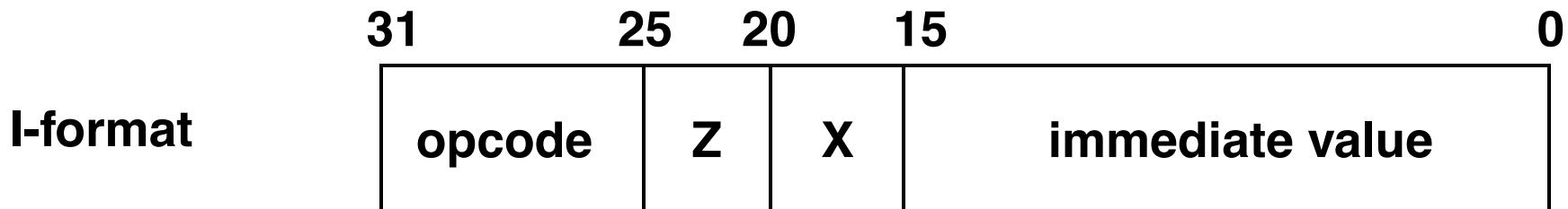
# BRANCHING

## INSTRUCTION FORMAT

- Branching requires **two sources** for comparison and the **relative offset**.



- This **B-format** is effectively the same as the **I-format**.



- We can likely make the instruction decoder simpler if we take the **B-format** to be the same as the **I-format**.
  - This might take a bit of extra decoding elsewhere in our DPU.

- Consider the following pseudo-code for a **loop**.

Pseudo-Code

```
a = 0
do
    ...
    a = a + 1
while ( a != 5 )
...
```

MIPS Assembly

```
add $26, $0, $0
add $14, $0, 0x05
loop: ...
...
add $26, $26, 0x01
bne $26, $14, loop
...
```

Register Transfer  
Notation

```
R26 = 0
R14 = 5
R26 = R26 + 1
PC = PC + 4
+word_offset<<2
```

- Notice how a conditional branch is used for the while loop.

- Consider the following pseudo-code for an **if-then-else** statement.

Pseudo-Code

```
if ( x != y ) then
    ...
    ...
else
    ...
    ...
endif
```

MIPS Assembly

(assume x in \$5, y in \$6)

```
beq $5, $6, else
...
...
j endif
else: ...
...
endif: ...
```

- Notice use of **beq** for **if-then-else** and **j** at end of **if-then**.

- Problem with previous slide is that we cannot relocate assembly code because of **j** instruction. Therefore, change assembly as follows.

### Pseudo-Code

```
if ( x != y ) then
    ...
    ...
else
    ...
    ...
endif
```

### MIPS Assembly (assume x in \$5, y in \$6)

```
beq $5, $6, else
...
...
beq $0, $0, endif
else: ...
...
endif: ...
```

- Another example is given below. Note: \$0 contains 0x00000000.

### Pseudo-Code

```
if (num<0) then
    num = -num
end
if (temperature>=25) then
    activity = "swim"
else
    activity = "cycle"
endif
```

### MIPS Assembly (almost)

```
lwi $15, num
bge $15, $0, endif0
sub $15, $0, $15
swi $15, num
endif0: lwi $15, temperature
       blt $15, 0x0019, else25
swim: ...
       j endif25
else25: ...
cycle: ...
endif25: ...
```

# BRANCHES ON MIPS

## GENERAL COMPARISONS

- The MIPS processor does not include all of the branches listed in the branch types table. The assembly makes *synthetic instructions* available.
- It actually only has **beq** and **bne** as built-in instructions.
- To perform branches such as **blt**, **ble**, **bgt**, and **bge**, MIPS uses another instruction, **slt** or **slti**, in combination with **beq** or **bne**.

Instruction	Opcode	Interpretation
<b>slt \$10, \$8, \$9</b>	<b>101010</b>	<b>If contents of \$8 &lt; contents of \$9, then \$10 = 0x01, else \$10 = 0x00.</b>
<b>slti \$10,\$8, 4</b>	<b>001010</b>	<b>If contents of \$8 &lt; 4, then \$10= 0x01, else \$10 = 0x00.</b>

# BRANCHES ON MIPS

## SLT AND SLTI

- How can **blt**, **ble**, **bgt**, and **bge** effectively be performed using **slt** and **slti**?

Desired Instruction	Meaning	Equivalent slt Condition	MIPS Instructions
<b>blt \$10, \$8, loop</b>	Branch to loop if $\$10 < \$8$	Branch to loop if $\$10 < \$8$	<code>slt \$5, \$10, \$8 bne \$5, \$0, loop</code>
<b>bge \$10, \$8, loop</b>	Branch to loop if $\$10 \geq \$8$	Branch to loop if NOT ( $\$10 < \$8$ )	<code>slt \$5, \$10, \$8 beq \$5, \$0, loop</code>
<b>bgt \$10, \$8, loop</b>	Branch to loop if $\$10 > \$8$	Branch to loop if $\$8 < \$10$	<code>slt \$5, \$8, \$10 bne \$5, \$0, loop</code>
<b>ble \$10, \$8, loop</b>	Branch to loop if $\$10 \leq \$8$	Branch to loop if NOT ( $\$8 < \$10$ )	<code>slt \$5, \$8, \$10 beq \$5, \$0, loop</code>

- Note: **\$0** contains **0x00000000**.

- Therefore, our previous example would actually be assembled as

MIPS Assembly (almost)

lwi \$15, num

~~bge \$15, \$0, endif0~~

sub \$15, \$0, \$15

swi \$15, num

endif0: lwi \$15, temperature

~~blt \$15, 0x0019, else25~~

swim: ...

j endif25

else25: ...

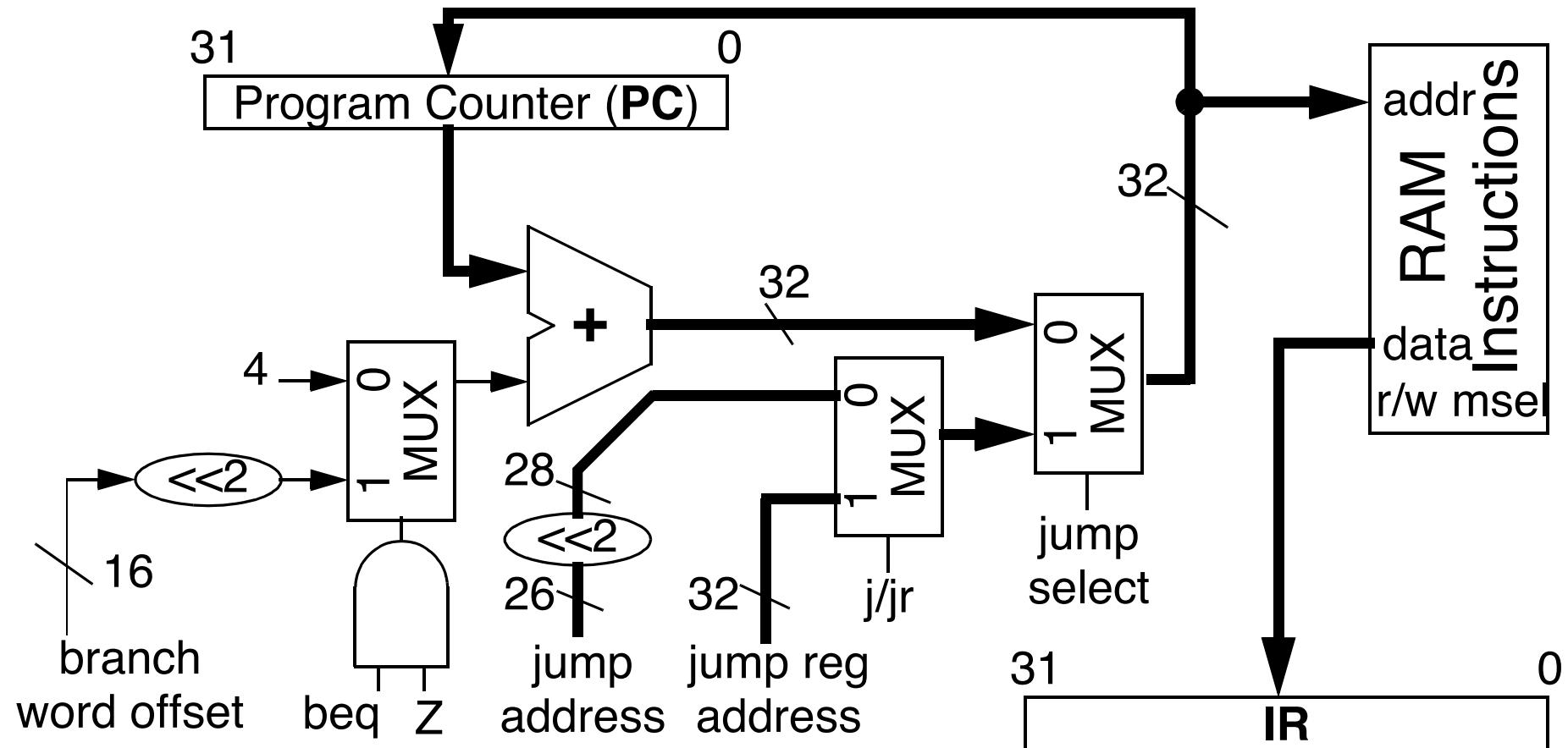
cycle: ...

endif25: ...

{ slt \$5, \$15, \$0  
    beq \$5, \$0, endif0

{ slti \$5, \$15, 0x0019  
    bne \$5, \$0, else25

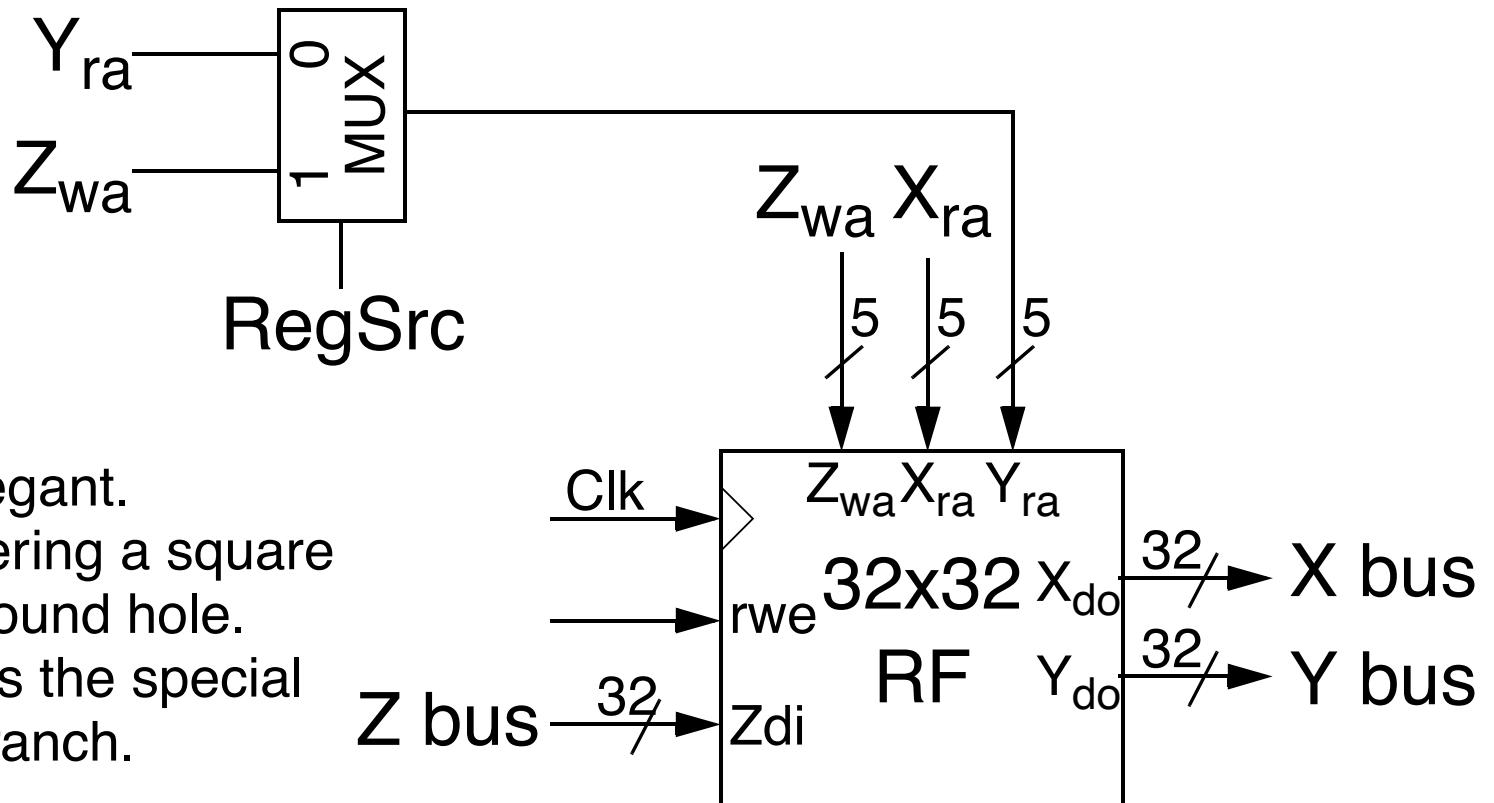
- Of course, modifications are needed to the DPU\* to allow updating the program counter appropriately with these branches and jumps.



# SINGLE CYCLE DPU

## MODIFICATIONS TO DPU

- Note that branch instructions have **two sources** and an **immediate value**.
- Differs from I-format with **one destination**, **one source**, and an **immediate value**.



- To calculate jump target consider the following instruction
  - **j 0x00400040**
- The encoding of the jump would be

J-format	31	25	0
	opcode	instruction word address	
	0000 10	00 0001 0000 0000 0000 0001 0000	

which gives an instruction encoding of **0x08100010** and not 0x08400040.

- Why? Because we need to encode with word addresses such that
  - **$0x00100010 \ll 2 = 0x00400040$**
  - This gives the preferred 28-bits over 26-bits.
- Hence, **PC[27:0] = (word\_address << 2)**

- Now consider the following instruction when **R8=0x00400040**,
  - **jr \$8**
- For this instruction, since the register **R8** is already 32-bits, we do not need to perform any shifting of the contents of **R8**.
- Hence, **PC = R8**, which is effectively **PC = 0x00400040** in the case.
- The encoding of this instruction will look like

JR-format	31	25	20	0
	opcode 0000 11	Z 01 000	X XXXX XXXX XXXX XXXX XXXX	

- This gives an instruction encoding of **0x0D000000** (for **X=0**).

# TARGET CALC.

## BRANCH TARGET CALC.

- For branch target calculation, consider the following code fragment.

0x00001000

beq \$1, \$2, skip

0x00001004

sub \$15, \$2, \$8

0x00001008

...

...

0x00004400    skip:    add \$3 \$13 \$2

- What is the value of the label **skip**? **skip = 0x00004400**
- We do not want to encode **skip** directly. We need **word offset!!**
  - **word offset = (0x00004400 - (0x00001000 + 0x04)) >> 2 = 0xCFF**

- Using the word offset calculated on the previous slide we can verify that this is the correct word offset since

$$\begin{aligned} \text{PC} &= \text{PC} + 4 + (\text{word offset} \ll 2) \\ &= 0x00001000 + 0x04 + (0x0CFF \ll 2) \\ &= 0x00001000 + 0x04 + 0x000033FC = 0x00004400 \end{aligned}$$

- Therefore, the instruction encoding for the branch **beq \$1, \$2, skip** is

	31	25	20	15	0
I-format	opcode	Z	X	word offset	
	0001 00	00 001	0 0010	0000 1100 1111 1111	

- This gives an instruction encoding of **0x10220CFF**.