
Appendix A: The Basics of Logic Design

ITSC 3181, Introduction to Computer Architecture

<https://passlab.github.io/ITSC3181/>

Department of Computer Science

Yonghong Yan

yyan7@uncc.edu

<https://passlab.github.io/yanyh/>

Appendix A: The Basics of Logic Design

- **Lecture 12**

- A.1 Introduction
- A.2 Gates, Truth Tables, and Logic Equation

- **Lecture 13**

- A.3 Combinational Logic
- ~~A.4 Using a Hardware Description Language~~

- **Lab 7**

- **Lecture 14**

- A.5 Constructing a Basic Arithmetic Logic Unit
- ~~A.6 Faster Addition: Carry Lookahead~~

- **Lecture 15**

- A.7 Clocks
- A.8 Memory Elements: Flip-Flops, Latches, and Registers

- **Lab 8**

- **Lecture 16**

- A.9 Memory Elements: SRAMs and DRAMs

- **Lab 9**

- ~~Lecture 18~~

- ~~A.10 Finite State Machines~~
- ~~A.11 Timing Methodologies~~
- ~~A.12. Field Programmable Devices~~
- A.13 Concluding Remarks
- ~~A.14 Exercises~~

- **Lab 10**

Introduction

- CPU performance factors

- Instruction count

- Determined by ISA and compiler

- CPI and Cycle time

- Determined by CPU hardware

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- A small subset of RISC-V ISA **that can support most high-level programming constructs**

- Memory reference: load and store such as lw, sw

- Arithmetic/logical: add, sub, and, or

- Control transfer: beq, j

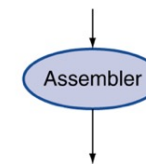
Instruction and Data (1/2)

- Are all numbers stored as binary format in memory
 - It is up to the CPU on how to interpret and do with them
- Each byte/word has its memory address

Name (Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

```

swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
    
```



```

000000001010000100000000000011000
000000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
0000001111100000000000000001000
    
```

2s-Complement Signed Integers

Bit 31 is sign bit

- 1 for negative numbers
- 0 for non-negative numbers

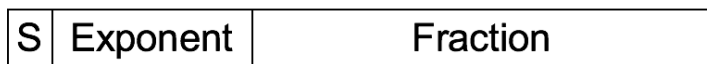
```

000 0000 0000 0000 0000 0000 0000 0000two = 0ten
000 0000 0000 0000 0000 0000 0000 0001two = 1ten
000 0000 0000 0000 0000 0000 0000 0010two = 2ten
...
011 1111 1111 1111 1111 1111 1111 1101two
011 1111 1111 1111 1111 1111 1111 1110two
011 1111 1111 1111 1111 1111 1111 1111two
1000 0000 0000 0000 0000 0000 0000 0000two
1000 0000 0000 0000 0000 0000 0000 0001two
1000 0000 0000 0000 0000 0000 0000 0010two
    
```

IEEE Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits



$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters.

Instruction and Data (2/2)

- Are all numbers stored as binary format in memory
 - It is up to the CPU on how to interpret and do with them
- Each byte/word has its memory address

The screenshot displays a debugger interface with three main panels:

- Text Segment:** A table of assembly instructions with columns for Bkpt, Address, Code, Basic, and Source. The first row is highlighted in yellow.
- Labels:** A table showing labels and their memory addresses.
- Data Segment:** A table showing memory addresses and their corresponding values at various offsets.

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x0040000	0x24080010	addiu \$8,\$0,0x00000010	33: li \$t0, 16 # \$t0 = numb...
<input type="checkbox"/>	0x0040000	0x24090010	addiu \$9,\$0,0x00000010	34: li \$t1, 16 # \$t1 = numb...
<input type="checkbox"/>	0x0040000	0x00008021	addu \$16,\$0,\$0	35: move \$s0, \$zero # \$s0 = row ...
<input type="checkbox"/>	0x0040000	0x00008821	addu \$17,\$0,\$0	36: move \$s1, \$zero # \$s1 = colu...
<input type="checkbox"/>	0x0040001	0x00005021	addu \$10,\$0,\$0	37: move \$t2, \$zero # \$t2 = the ...
<input type="checkbox"/>	0x0040001	0x02090018	mult \$16,\$9	41: loop: mult \$s0, \$t1 # \$s2 = row ...
<input type="checkbox"/>	0x0040001	0x00009012	mflo \$18	42: mflo \$s2 # move multi...
<input type="checkbox"/>	0x0040001	0x02519020	add \$18,\$18,\$17	43: add \$s2, \$s2, \$s1 # \$s2 += col...
<input type="checkbox"/>	0x0040002	0x00129080	sll \$18,\$18,0x00000002	44: sll \$s2, \$s2, 2 # \$s2 *= 4 (...)
<input type="checkbox"/>	0x0040002	0x3c011001	lui \$1,0x00001001	45: sw \$t2, data(\$s2) # store the ...
<input type="checkbox"/>	0x0040002	0x00320821	addu \$1,\$1,\$18	
<input type="checkbox"/>	0x0040002	0xac2a0000	sw \$10,0x00000000(\$1)	

Label	Address
row-major.asm	
loop	0x00400014
data	0x10010000

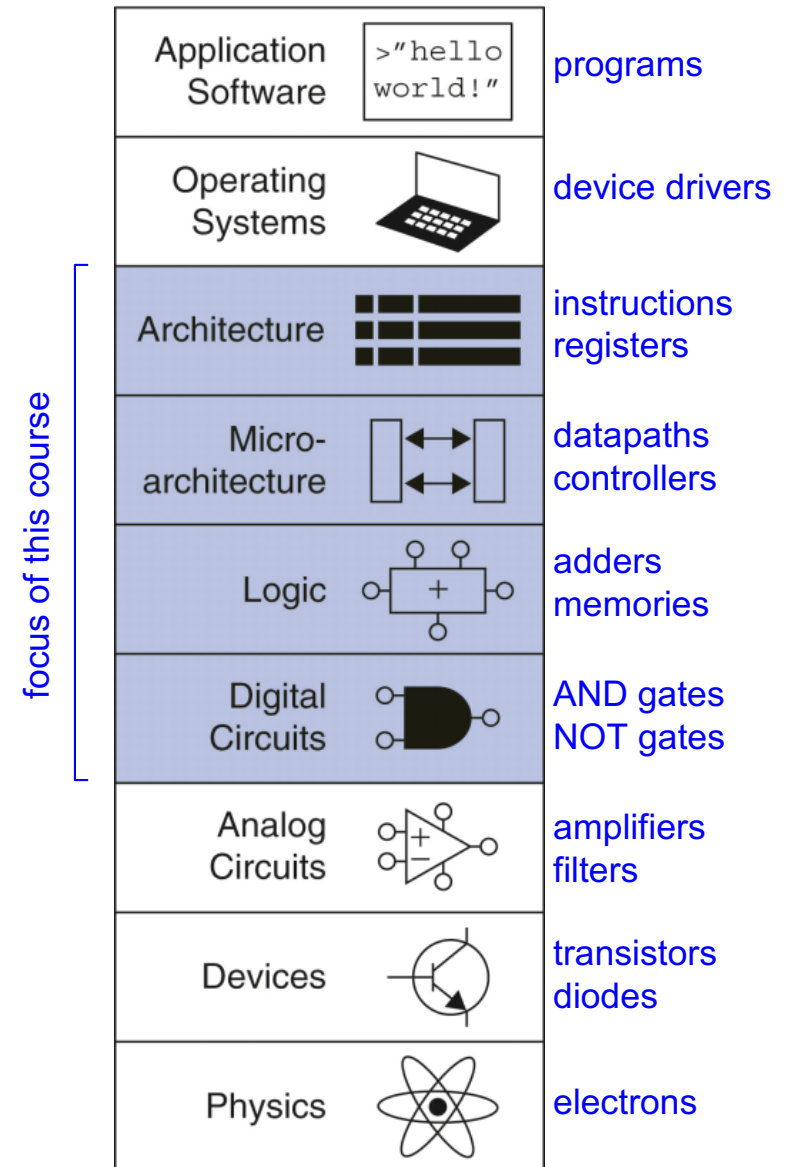
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000

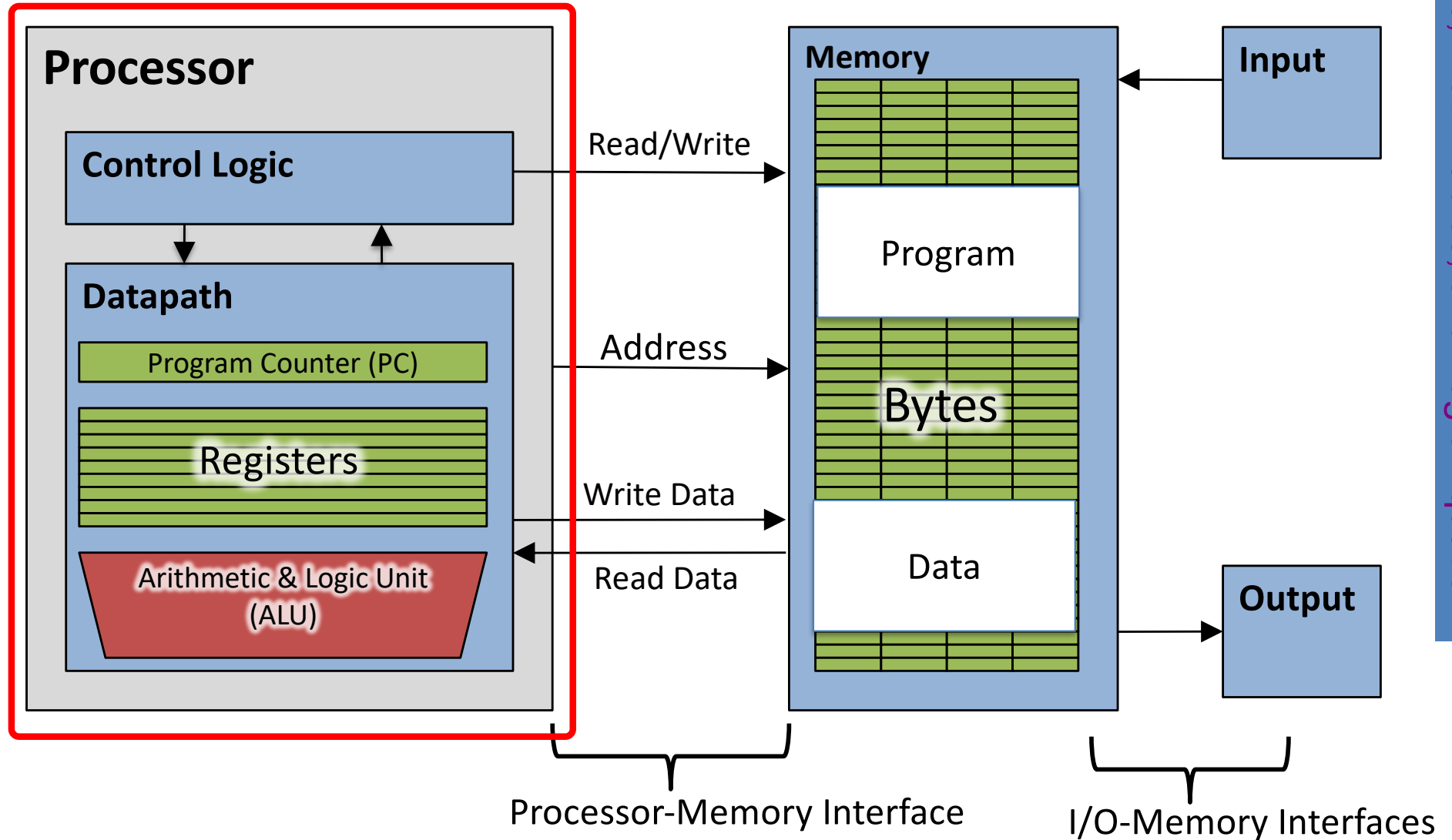
Appendix A and Chapter 4 and 5

- Study how a processor is designed and its implication to software and performance
 - Foundation of CPU design
-
- Bottom-up approach to study
 - Appendix A: logic design
 - Chapter 4: CPU design
 - Chapter 5: Memory design

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

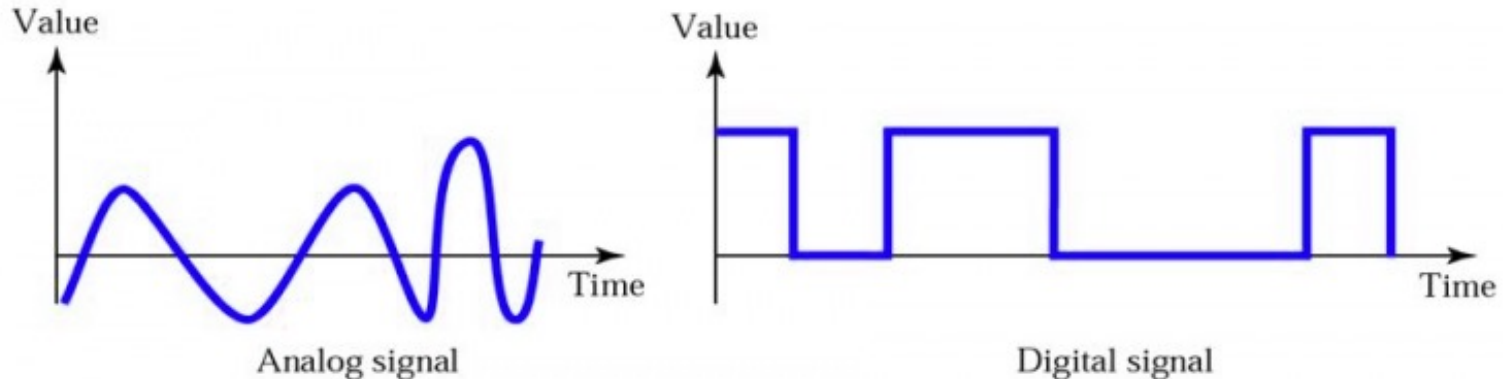


Components of a Computer



Logic Design Basics

- To represent and store data, and to perform operation
 - 0 and 1
 - Start with addition
- Electrics inside a computer are digital



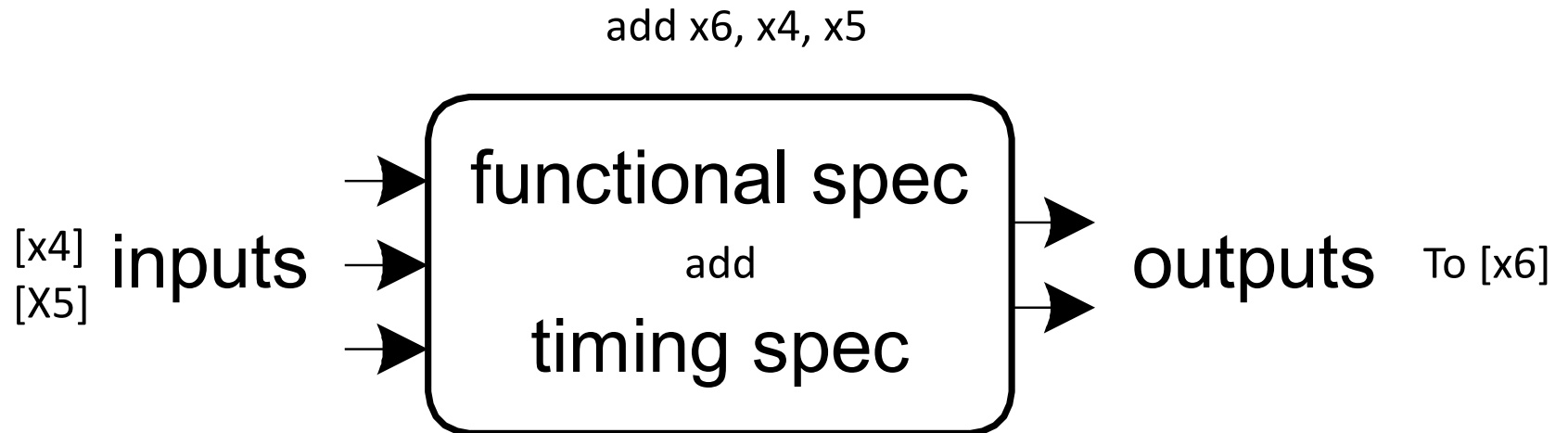
- Information encoded in binary
- Digital circuits use **voltage** levels to represent 1 and 0
 - Low voltage = 0, FALSE, deasserted
 - High voltage = 1, TRUE, asserted
 - One wire per bit
 - Multi-bit data path is encoded via multi-wire buses



Logic Circuit

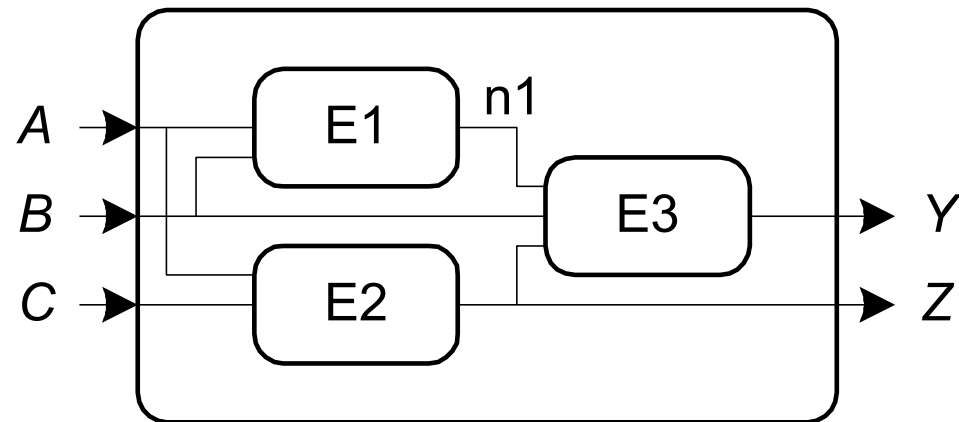
A logic circuit is composed of:

- Inputs
- Outputs
- Functional specification
- Timing specification



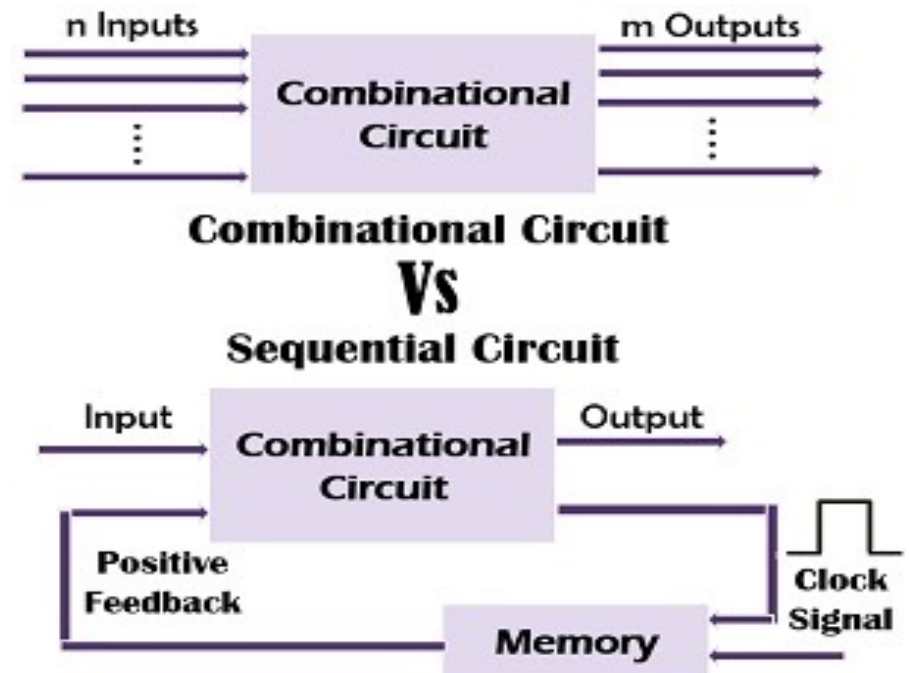
Logic Circuits

- Nodes
 - Inputs: A, B, C
 - Outputs: Y, Z
 - Internal: $n1$
- Circuit elements
 - $E1, E2, E3$

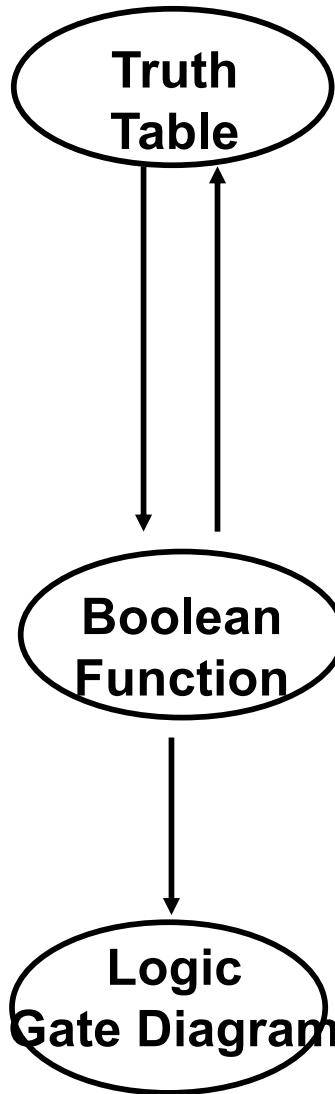


Combinational and Sequential Circuits

- To perform operation and store data
- Combinational circuit, such as adder
 - Operate on data
 - Output is a function of input
- State (sequential) circuit, such as register or memory
 - Store information
 - Outputs determined by previous and current values of inputs

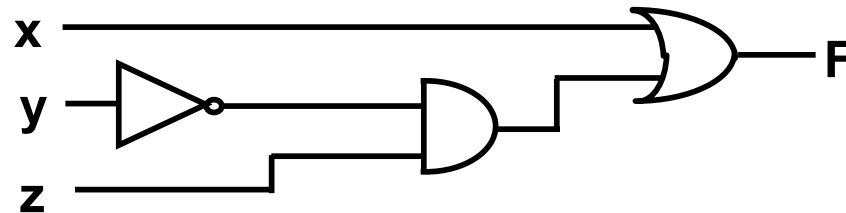


Three Steps of Logic Design in Theory



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = x + y'z$$



Step 1: Truth Table for Binary Logic

- Given N input binary variables, list the output for all the possible inputs
 - N input $\rightarrow 2^n$ number of input combinations of 0 and 1
 - It is digital version of a function, e.g. $D = f(A, B, C)$

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- Computing is a function of binary input, and output is binary

Step 2: Boolean Algebra

- Logic equation to express binary logic function using binary variable, instead of a truth table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

- Three fundamental operators

- OR operator, logic sum, written as +

- $Y = A + B$

- AND operator, logic product, written as * or .

- $Y = A * B$, or AB

- NOT operator, inverse, written as $\sim A$, A' or \overline{A}

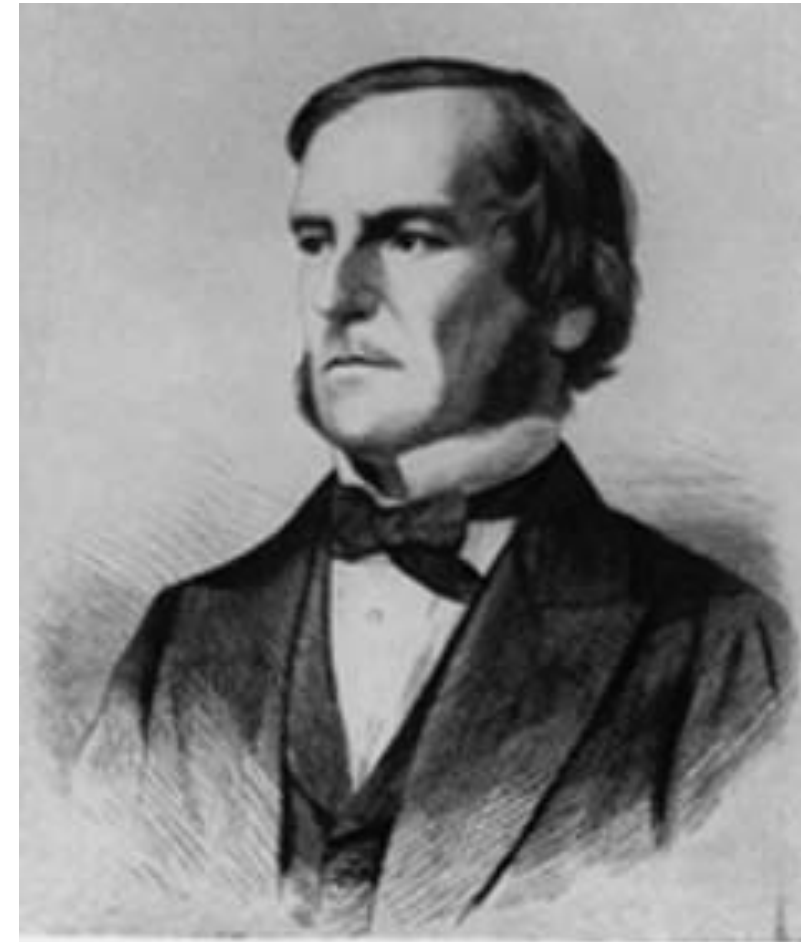
- $\sim\sim A = A$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	$\sim A$
0	1
1	0

George Boole, 1815-1864

- Born to working class parents
- Taught himself mathematics and joined the faculty of Queen's College in Ireland
- Wrote *An Investigation of the Laws of Thought* (1854)
- Introduced binary variables
- Introduced the three fundamental logic operations: AND, OR, and NOT



GEORGE BOOLE

Scanned at the American
Institute of Physics

Laws of Boolean Algebra (1/3)

- Basic operators of Boolean algebra

- AND, *
- OR, +
- NOT, ~

1. Identity Law

- $A + 0 = A$
- $A * 1 = A$

2. Zero and One Laws:

- $A + 1 = 1$
- $A * 0 = 0$

3. Inverse Laws:

- $A + \sim A = 1$
- $A * A^{\sim} = 0$

Laws of Boolean Algebra (2/3)

- Basic operators of Boolean algebra

- AND, *
- OR, +
- NOT, ~

4. Commutative Law

- $A + B = B + A$
- $A * B = B * A$

5. Associative Laws:

- $A + (B + C) = (A + B) + C$
- $A * (B * C) = (A * B) * C$

6. Distributive Laws:

- $A * (B + C) = (A * B) + (A * C)$
- $A + (B * C) = (A + B) * (A + C)$

Laws of Boolean Algebra (3/3)

- Basic operators of Boolean algebra

- AND, *
- OR, +
- NOT, ~

7. DeMorgan's Laws:

- $\sim(A * B) = \sim A + \sim B$
- $\sim(A + B) = \sim A * \sim B$

- Extended

- $(x_1 + x_2 + \dots + x_n)' = x_1' x_2' \dots x_n'$
- $(x_1 x_2 \dots x_n)' = x_1' + x_2' + \dots + x_n'$

- Easy way to remember: each TERM is complemented, AND \rightarrow OR, OR \rightarrow AND

Operators and Laws of Boolean Algebra: Summary

- Basic operators of Boolean algebra

- AND, *
- OR, +
- NOT, ~

1. Identity Law

- $A + 0 = A$
- $A * 1 = A$

2. Zero and One Laws:

- $A + 1 = 1$
- $A * 0 = 0$

3. Inverse Laws:

- $A + \sim A = 1$
- $A * A\sim = 0$

4. Commutative Law

- $A + B = B + A$
- $A * B = B * A$

5. Associative Laws:

- $A + (B + C) = (A + B) + C$
- $A * (B * C) = (A * B) * C$

6. Distributive Laws:

- $A * (B + C) = (A * B) + (A * C)$
- $A + (B * C) = (A + B) * (A + C)$

7. DeMorgan's Laws:

- $\sim(A * B) = \sim A + \sim B$
- $\sim(A + B) = \sim A * \sim B$

- Extended

- $(x_1 + x_2 + \dots + x_n)' = x_1'x_2'\dots x_n'$
- $(x_1x_2 \dots x_n)' = x_1' + x_2' + \dots + x_n'$

Derive Logic Equation from Truth Table 1/2

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- Write down the Boolean equation for each line in the truth table where the output is 1
- Simplify the equation using Boolean Algebra Laws
- $F = A * B * C$
- $E = \sim A * B * C + A * \sim B * C + A * B * \sim C$

Derive Logic Equation from Truth Table 2/2

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- Write down the Boolean equation for each line in the truth table where the output is 0, and do NOT
- Simplify the equation using Boolean Algebra Laws
- $D = \sim(\sim A * \sim B * \sim C) = A + B + C$

More Examples

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

To write down the Boolean expression that describes this truth table (and therefore the system that the truth table describes) we simply write down the Boolean equation for each line in the truth table where the output is 1.

The output for the first line is 0, so we ignore it.

The output for the second line is a 1. The Boolean equation for this line is $\bar{A}.B.\bar{C}$

The output for the third line is a 1. The Boolean equation for this line is $A.\bar{B}.\bar{C}$

The output for the fourth line is 0, so we ignore it.

The output for the fifth line is a 1. The Boolean equation for this line is $\bar{A}.\bar{B}.C$

The output for the sixth line is 0, so we ignore it.

The output for the seventh line is 0, so we ignore it.

The output for the eighth line is a 1. The Boolean equation for this line is $A.B.C$

We can now get the Boolean equation for the whole system simply by getting the equations where the output was 1 and ORing them together. This gives us the output Q:

$$\bar{A}.B.\bar{C} + A.\bar{B}.\bar{C} + \bar{A}.\bar{B}.C + A.B.C = Q$$

<http://theteacher.info/index.php/fundamentals-of-cs/2-logical-operations/topics/2642-deriving-boolean-expressions-from-truth-tables>

Simplifying Boolean Equations

$Y = AB + \sim AB$	<i>Law #:</i> 6		<i>law #:</i>
$= B(A + \sim A)$	3	$Y = A(AB + ABC)$	1,6
$= B(1)$	1	$= A(AB(1 + C))$	2
$= B$		$= A(AB(1))$	1
<ul style="list-style-type: none"> • Basic operators of Boolean algebra <ul style="list-style-type: none"> – AND, * – OR, + – NOT, ~ 	<ol style="list-style-type: none"> 4. Commutative Law <ul style="list-style-type: none"> – $A + B = B + A$ – $A * B = B * A$ 	$= A(AB)$	5
<ol style="list-style-type: none"> 1. Identity Law <ul style="list-style-type: none"> – $A + 0 = A$ – $A * 1 = A$ 	<ol style="list-style-type: none"> 5. Associative Laws: <ul style="list-style-type: none"> – $A + (B + C) = (A + B) + C$ – $A * (B * C) = (A * B) * C$ 	$= (AA)B$	
<ol style="list-style-type: none"> 2. Zero and One Laws: <ul style="list-style-type: none"> – $A + 1 = 1$ – $A * 0 = 0$ 	<ol style="list-style-type: none"> 6. Distributive Laws: <ul style="list-style-type: none"> – $A * (B + C) = (A * B) + (A * C)$ – $A + (B * C) = (A + B) * (A + C)$ 	$= AB$	
<ol style="list-style-type: none"> 3. Inverse Laws: <ul style="list-style-type: none"> – $A + \sim A = 1$ – $A * A\sim = 0$ 	<ol style="list-style-type: none"> 7. DeMorgan's Laws: <ul style="list-style-type: none"> – $\sim(A * B) = \sim A + \sim B$ – $\sim(A + B) = \sim A * \sim B$ 		
	<ul style="list-style-type: none"> • Extended <ul style="list-style-type: none"> – $(x_1 + x_2 + \dots + x_n)' = x_1'x_2'\dots x_n'$ – $(x_1x_2 \dots x_n)' = x_1' + x_2' + \dots + x_n'$ 		

Step 3: Logic Gates

- Implement the Boolean equation in circuits

- Perform logic functions:

- inversion (NOT), AND, OR, NAND, NOR, etc.

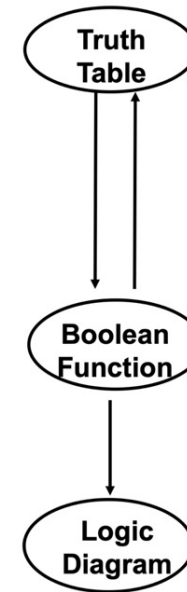
- Single-input:

- NOT gate, buffer

- Two-input:

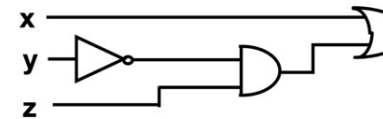
- AND, OR, XOR, NAND, NOR, XNOR

- Multiple-input



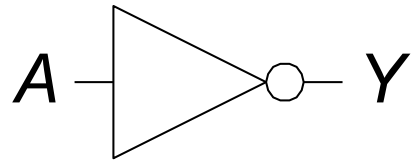
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = x + y'z$$



Single-Input Logic Gates

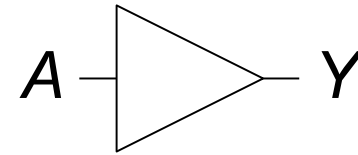
NOT



$$Y = \overline{A}$$

A	Y
0	
1	

BUF

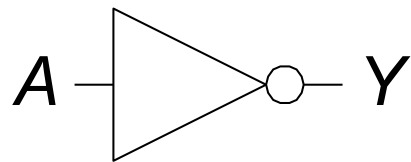


$$Y = A$$

A	Y
0	
1	

Single-Input Logic Gates

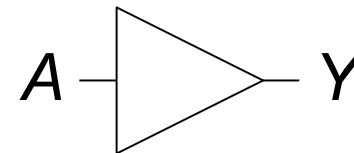
NOT



$$Y = \bar{A}$$

A	Y
0	1
1	0

BUF

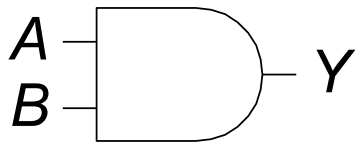


$$Y = A$$

A	Y
0	0
1	1

Two-Input Logic Gates

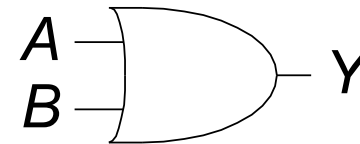
AND



$$Y = AB$$

A	B	Y
0	0	
0	1	
1	0	
1	1	

OR

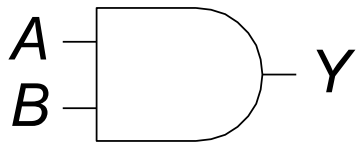


$$Y = A + B$$

A	B	Y
0	0	
0	1	
1	0	
1	1	

Two-Input Logic Gates

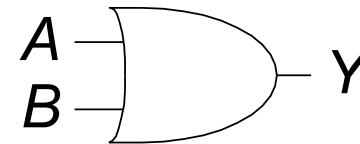
AND



$$Y = AB$$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

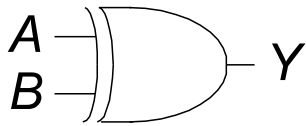


$$Y = A + B$$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

More Two-Input Logic Gates

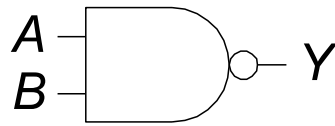
XOR



$$Y = A \oplus B$$

A	B	Y
0	0	
0	1	
1	0	
1	1	

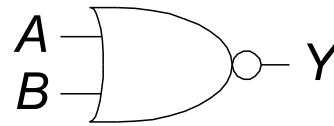
NAND



$$Y = \overline{AB}$$

A	B	Y
0	0	
0	1	
1	0	
1	1	

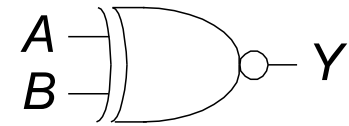
NOR



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

A	B	Y
0	0	
0	1	
1	0	
1	1	

XNOR

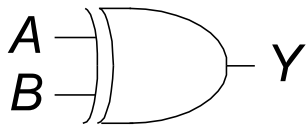


$$Y = \overline{A \oplus B}$$

A	B	Y
0	0	
0	1	
1	0	
1	1	

More Two-Input Logic Gates

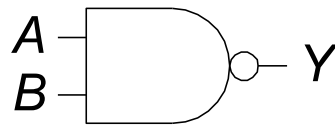
XOR



$$Y = A \oplus B$$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

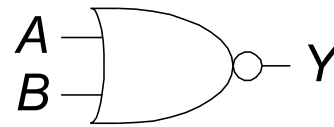
NAND



$$Y = \overline{AB}$$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

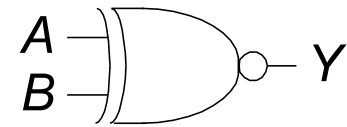
NOR



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

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

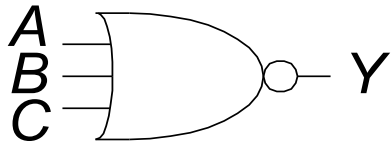


$$Y = \overline{A \oplus B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Multiple-Input Logic Gates

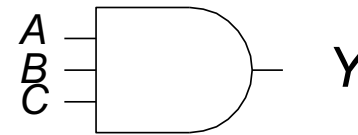
NOR3



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

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

AND3

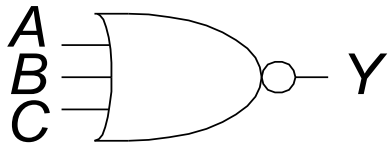


$$Y = ABC$$

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

Multiple-Input Logic Gates

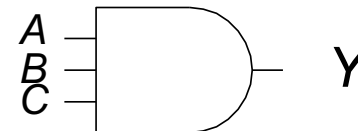
NOR3



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

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

AND3



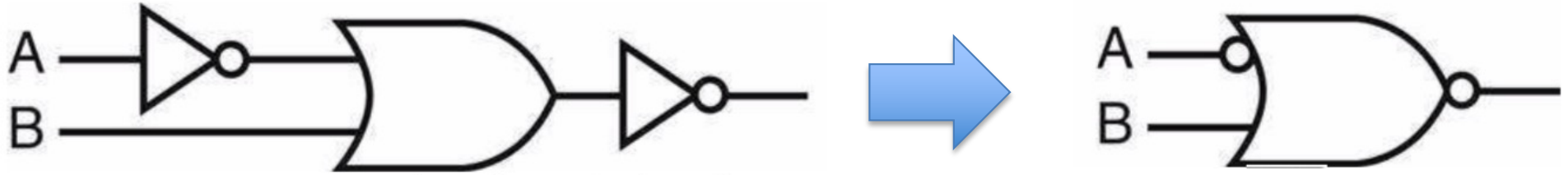
$$Y = ABC$$

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

- Multi-input XOR: Odd parity

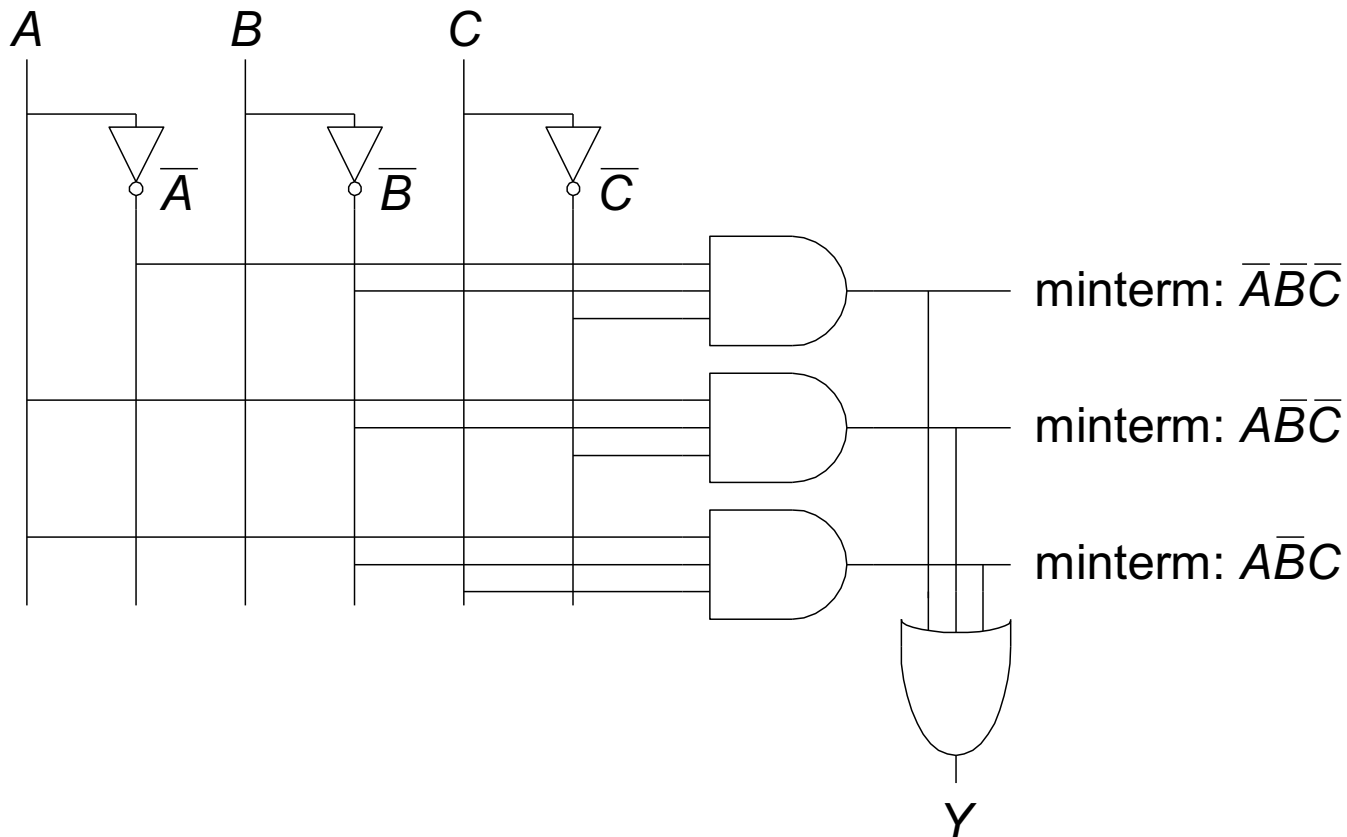
Bubble to Invert (NOT) Inputs or Outputs

$$\overline{\bar{A} + B}$$



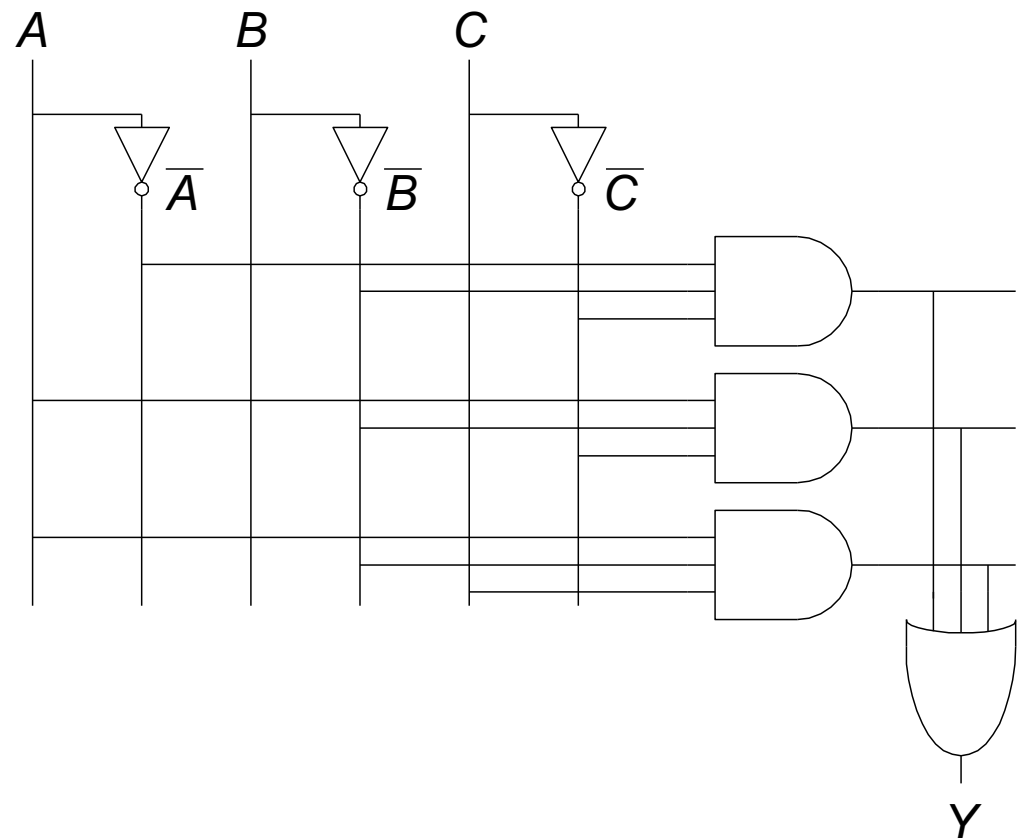
From Logic to Gates

- Two-level logic: ANDs followed by ORs
- Example: $Y = \overline{A}BC + A\overline{B}C + A\overline{B}\overline{C}$



Circuit Schematics Rules

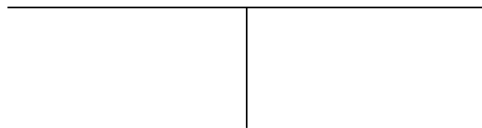
- Inputs on the left (or top)
- Outputs on right (or bottom)
- Gates flow from left to right
- Straight wires are best



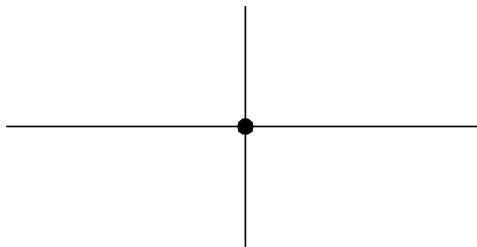
Circuit Schematic Rules (cont.)

- Wires always connect at a T junction
- A dot where wires cross indicates a connection between the wires
- Wires crossing *without* a dot make no connection

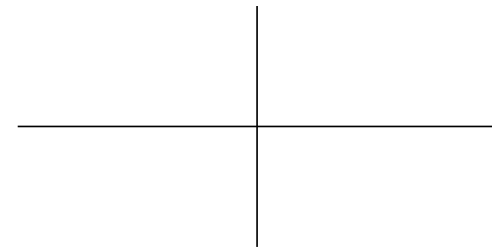
wires connect
at a T junction



wires connect
at a dot



wires crossing
without a dot do
not connect



Equivalent Circuits

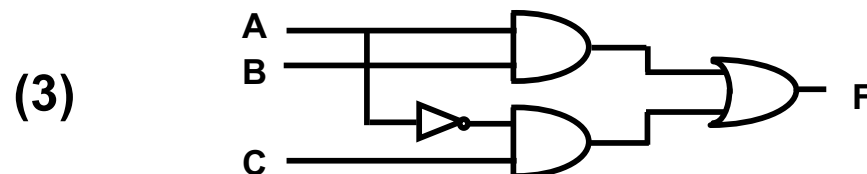
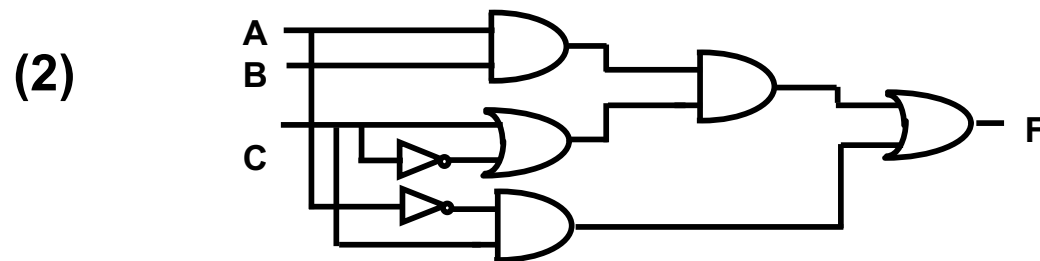
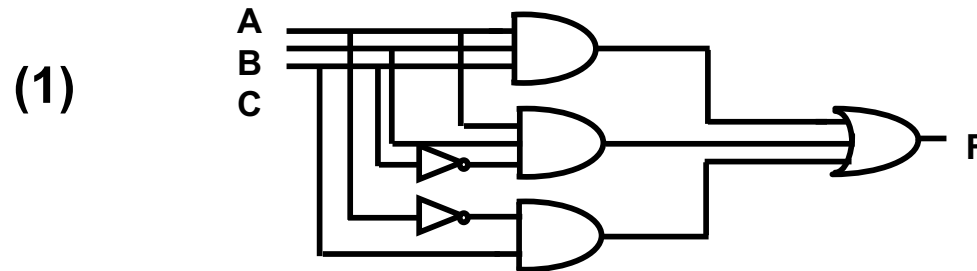
Many different logic diagrams are possible for a given function

$$F = ABC + ABC' + A'C \quad \dots\dots\dots (1)$$

$$= AB(C + C') + A'C \quad \dots\dots\dots (2)$$

$$= AB \cdot 1 + A'C$$

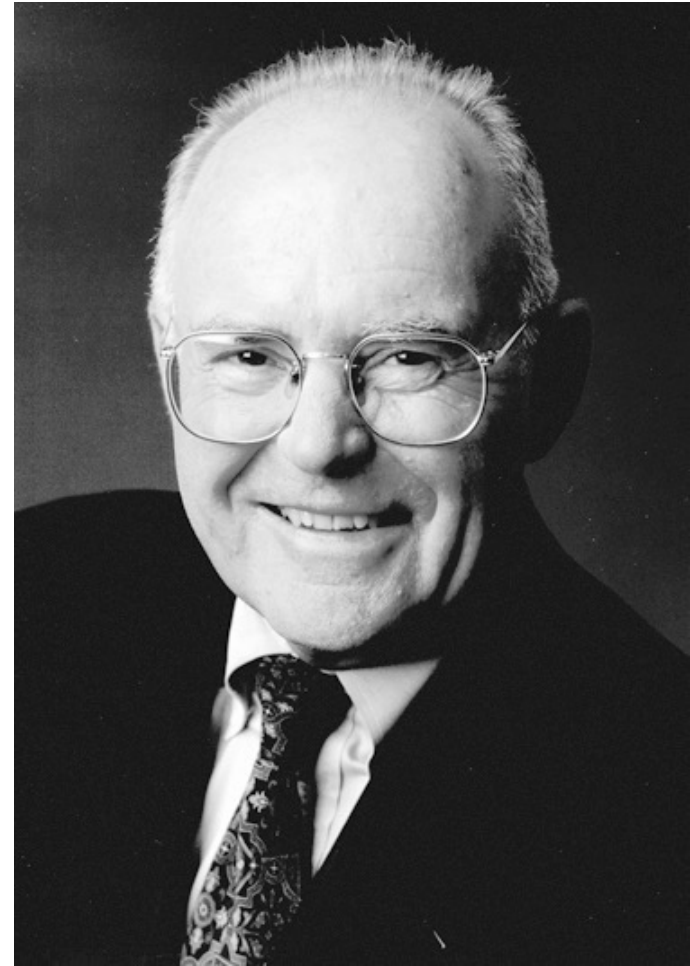
$$= AB + A'C \quad \dots\dots\dots (3)$$



**Simplified function
uses less gates.**

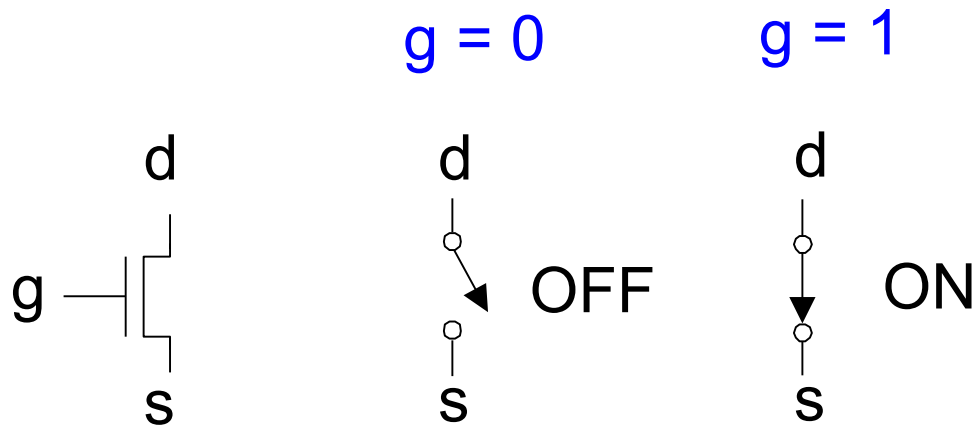
Gates are Implemented Using Transistors

- **Moore's Law:** number of **transistors** on a computer chip doubles every year (observed in 1965)
- Since 1975, transistor counts have doubled every two years.



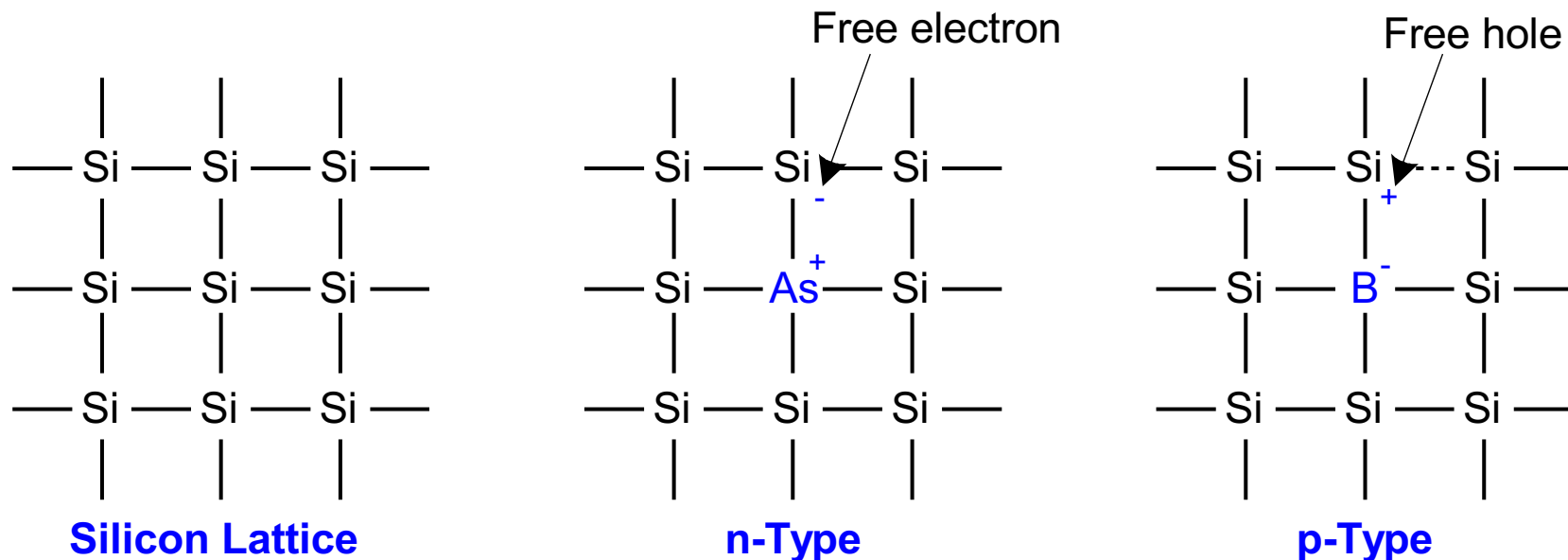
Transistors

- Logic gates built from transistors
- 3-ported voltage-controlled switch
 - 2 ports connected depending on voltage of 3rd
 - d and s are connected (ON) when g is 1



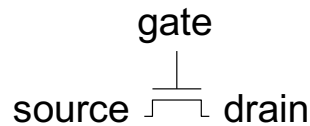
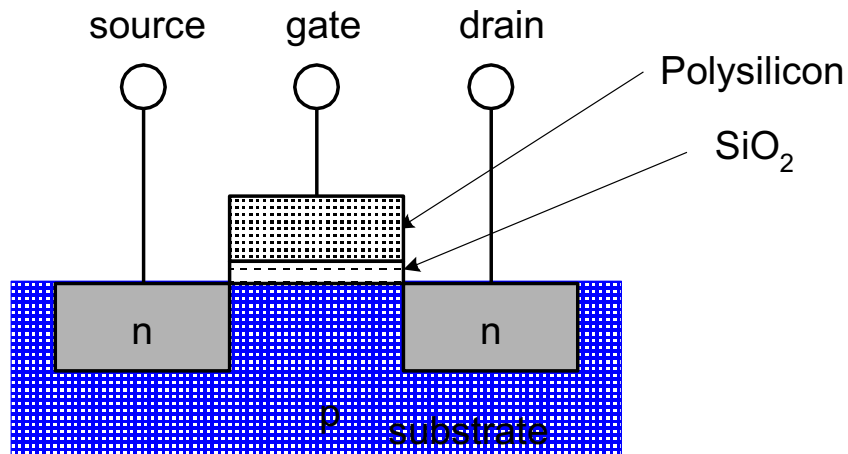
Silicon

- Transistors built from silicon, a semiconductor
- Pure silicon is a poor conductor (no free charges)
- Doped silicon is a good conductor (free charges)
 - n-type (free *negative* charges, electrons)
 - p-type (free *positive* charges, holes)



MOS Transistors

- **Metal oxide silicon (MOS) transistors:**
 - Polysilicon (used to be **metal**) gate
 - **Oxide** (silicon dioxide) insulator
 - Doped **silicon**

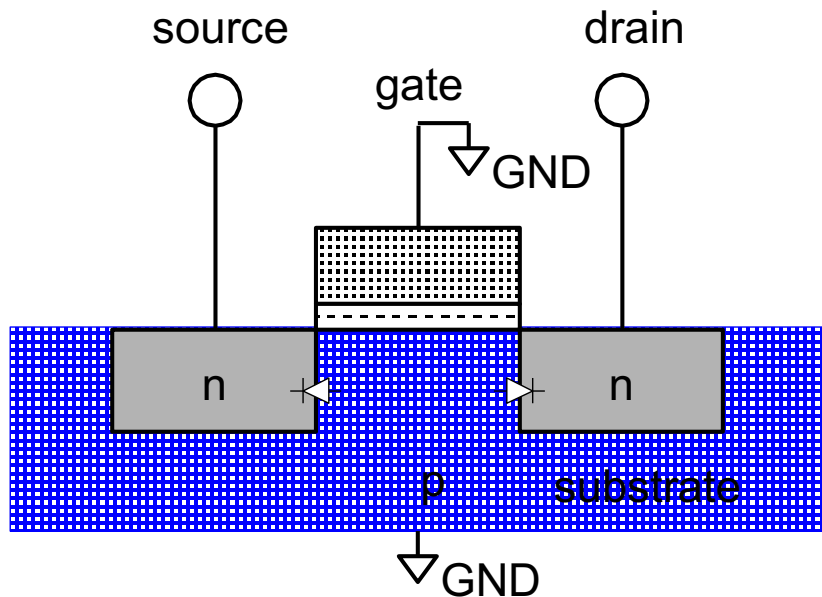


nMOS

Transistors: nMOS

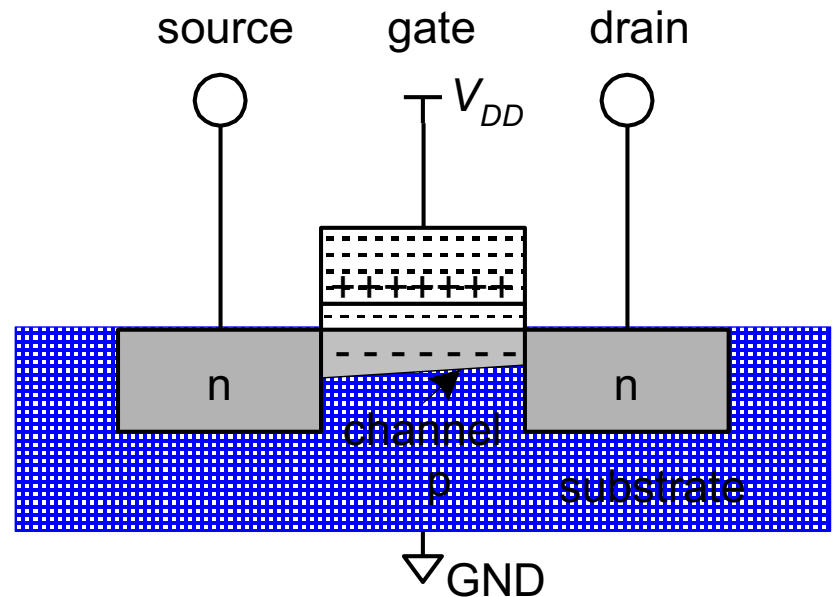
Gate = 0

OFF (no connection between source and drain)



Gate = 1

ON (channel between source and drain)

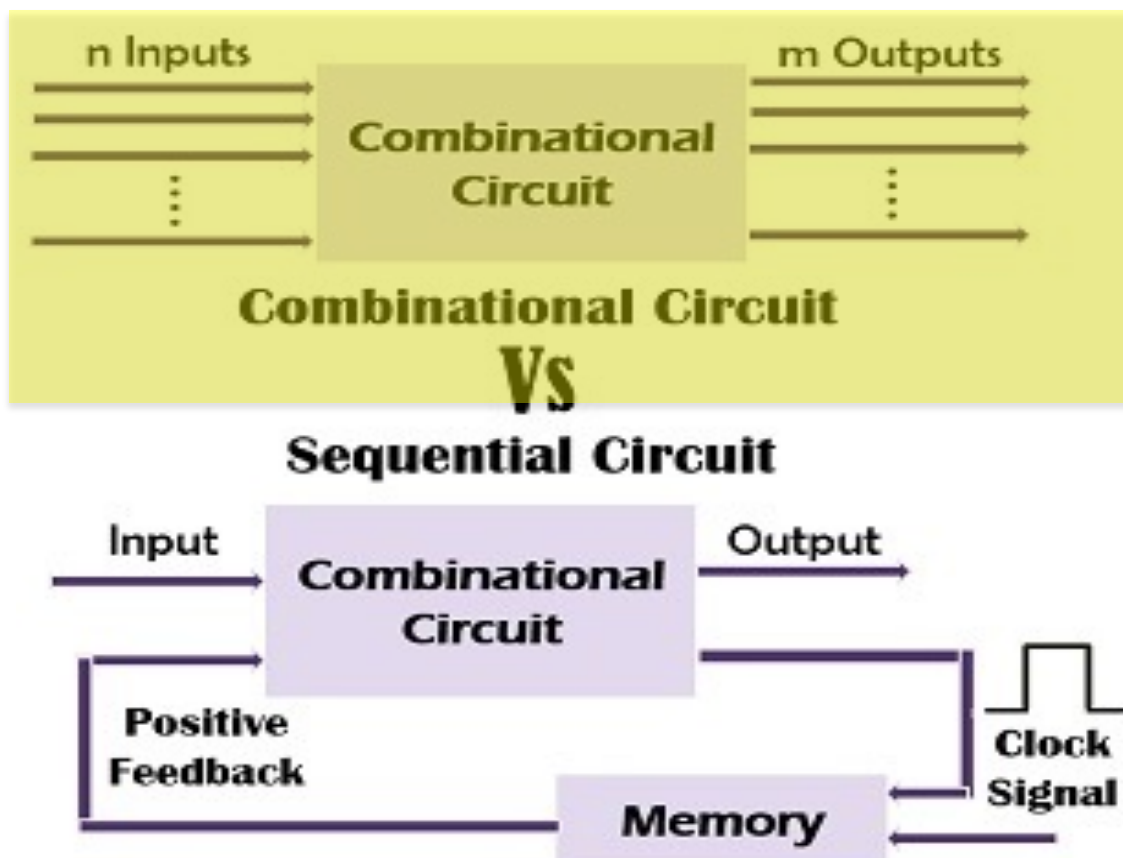


Appendix A: The Basics of Logic Design

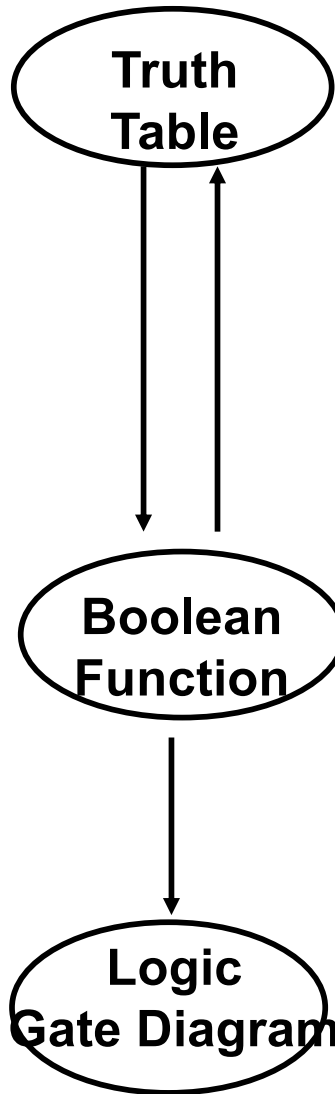
- **Lecture 12**
 - **A.1 Introduction**
 - **A.2 Gates, Truth Tables, and Logic Equation**
- **Lecture 13**
 - **A.3 Combinational Logic**
 - ~~A.4 Using a Hardware Description Language~~
- **Lab 7**
- **Lecture 14**
 - **A.5 Constructing a Basic Arithmetic Logic Unit**
 - ~~A.6 Faster Addition: Carry Lookahead~~
- **Lecture 15**
 - **A.7 Clocks**
 - **A.8 Memory Elements: Flip-Flops, Latches, and Registers**
- **Lab 8**
- **Lecture 16**
 - **A.9 Memory Elements: SRAMs and DRAMs**
- **Lab 9**
- ~~Lecture 17~~
 - ~~A.10 Finite State Machines~~
 - ~~A.11 Timing Methodologies~~
 - ~~A.12. Field Programmable Devices~~
 - **A.13 Concluding Remarks**
 - ~~A.14 Exercises~~
- **Lab 10**

Combinational Logic

- Two-level of logic and PLA
 - Product of Sum and
 - Sum of Product
 - PLA
- ROM
- Don't Care
- Multiplexer
- Decoder

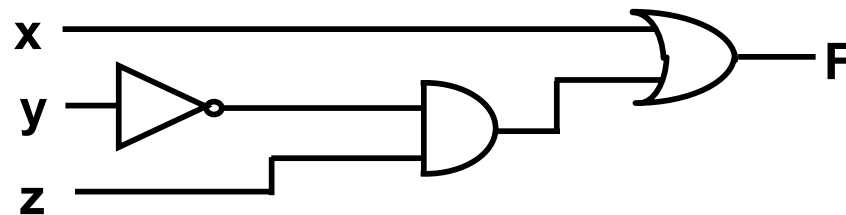


Three Steps of Logic Design in Theory



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = x + y'z$$



Two-Levels of Logic and PLA

- A general approach to derive Boolean function from truth table and then construct logic circuit

Some Definitions

- Complement: variable with a bar or ' over it (NOT)
 A', B', C'
- Literal: variable or its complement
 A, A', B, B', C, C'
- Minterm: **product** that includes all input variables (**AND**)
 $AB'C, A'BC, AB'C'$
 - **0, the minimum, determines the value, so it is called minterm**
- Maxterm: **sum** that includes all input variables (**OR**)
 $(A+B'+C'), (A'+B+C), (A'+B'+C)$
 - **1, the maximum, determines the value, so it is called maxterm**

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
 - Each row has a **minterm**
 - A minterm is a product (AND) of literals
 - Each minterm is TRUE for that row (and only that row)
 - Form function by ORing minterms where the output is TRUE
 - Thus, a sum (OR) of products (AND terms)

A	B	Y	minterm	minterm name
0	0	0	$\overline{A} \overline{B}$	m_0
0	1	1	$\overline{A} B$	m_1
1	0	0	$A \overline{B}$	m_2
1	1	1	$A B$	m_3

$$Y = F(A, B) =$$

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
 - Each row has a **minterm**
 - A minterm is a product (AND) of literals
 - Each minterm is TRUE for that row (and only that row)
 - Form function by ORing minterms where the output is TRUE
 - Thus, a sum (OR) of products (AND terms)

A	B	Y	minterm	minterm name
0	0	0	$\bar{A} \bar{B}$	m_0
0	1	1	$\bar{A} B$	m_1
1	0	0	$A \bar{B}$	m_2
1	1	1	$A B$	m_3

$$Y = F(A, B) = \bar{A}B + AB = \Sigma(1, 3)$$

Product-of-Sums (POS) Form

- All Boolean equations can be written in POS form
 - Each row has a **maxterm**
 - A maxterm is a sum (OR) of literals
 - Each maxterm is FALSE for that row (and only that row)
 - Form function by ANDing the maxterms for which the output is FALSE
 - Thus, a product (AND) of sums (OR terms)

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \bar{B}$	M_1
1	0	0	$\bar{A} + B$	M_2
1	1	1	$\bar{A} + \bar{B}$	— M_3

$$Y = F(A, B) = (A + B)(A + \bar{B}) = \Pi(0, 2)$$

SOP & POS Form

- SOP – sum-of-products

O	C	E	minterm
0	0		$\overline{O} \overline{C}$
0	1		$\overline{O} C$
1	0		$O \overline{C}$
1	1		$O C$

- POS – product-of-sums

O	C	E	maxterm
0	0		$O + C$
0	1		$O + \overline{C}$
1	0		$\overline{O} + C$
1	1		$\overline{O} + \overline{C}$

SOP & POS Form

- SOP – sum-of-products

O	C	E	minterm
0	0	0	$\bar{O} \bar{C}$
0	1	0	$\bar{O} C$
1	0	1	$O \bar{C}$
1	1	0	$O C$

$$E = O\bar{C}$$

$$= \Sigma(2)$$

- POS – product-of-sums

O	C	E	maxterm
0	0	0	$O + C$
0	1	0	$O + \bar{C}$
1	0	1	$\bar{O} + C$
1	1	0	$\bar{O} + \bar{C}$

$$E = (O + C)(O + \bar{C})(\bar{O} + \bar{C})$$

$$= \Pi(0, 1, 3)$$

Sum of Produce (SOP) is easy to use

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- Write down the Boolean equation for each line in the truth table where the output is 1
- $E = \sim A * B * C + A * \sim B * C + A * B * \sim C$ in SOP Form

Sum of Products Example

Inputs		Outputs	
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- Boolean function for D
- $D = A' * B' + A' * B + A * B' + A * B$

Sum of Product \rightarrow Programmable Logic Array (PLA)

$$D = A' * B' * C + A' * B * C + A * B' * C' + A * B * C$$

- Sum of Product representation is two stages of logic
 - Array of AND operations for the minterms
 - Array of OR operations to sum logically up the minterms
- Programmable Logic Array (PLA) to implement
 - Very easy and efficient to implement

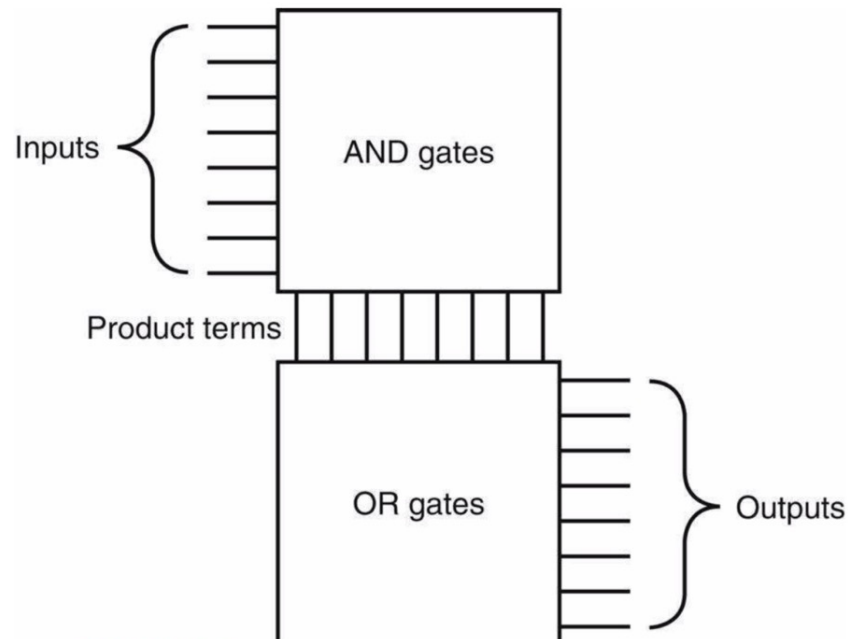
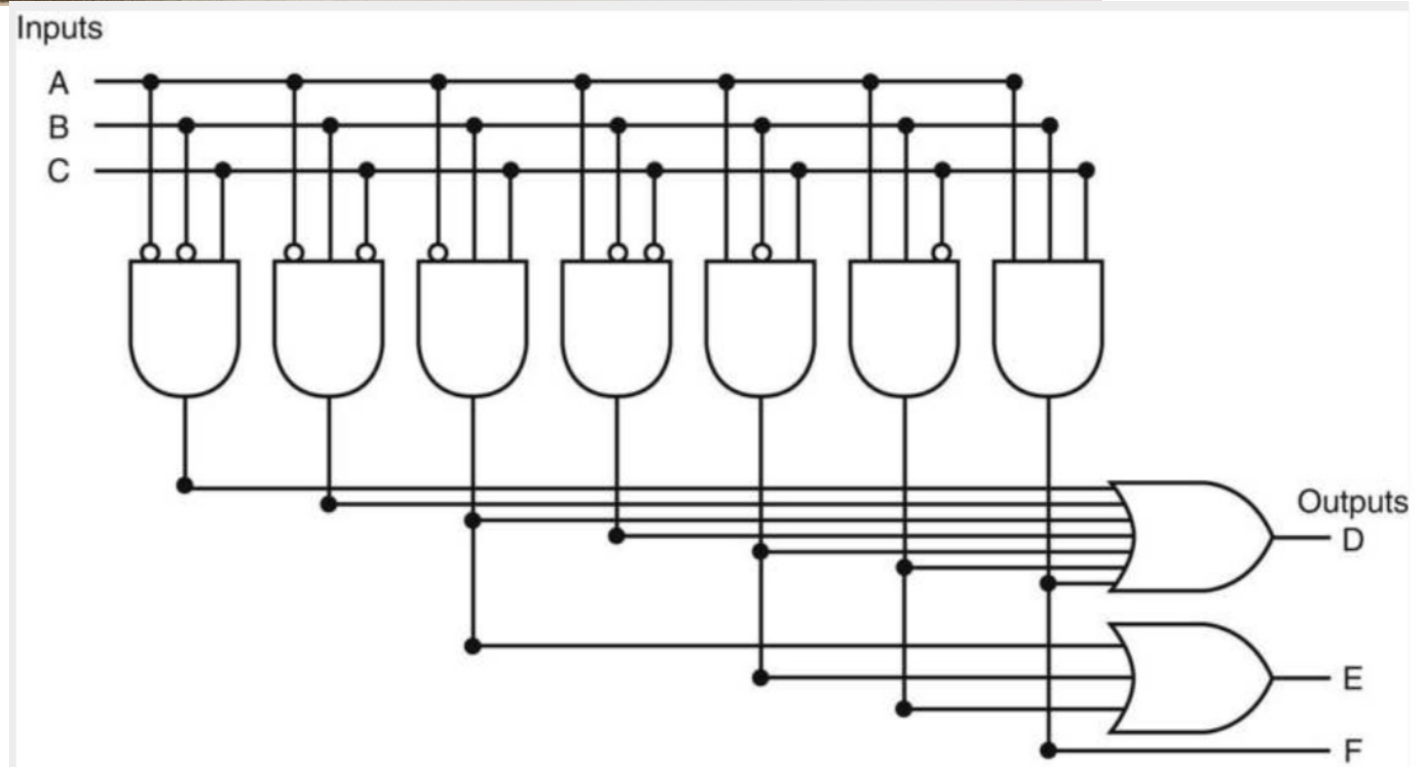
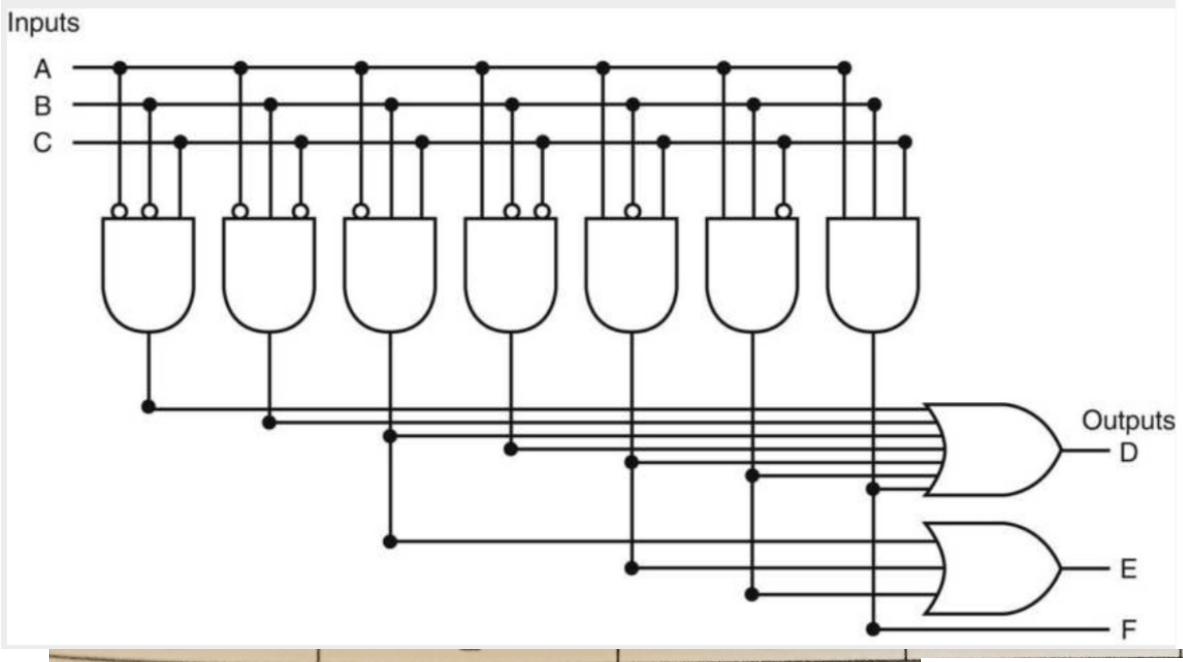


FIGURE A.2.3 The basic form of a PLA consists of

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- PLA impl based on SOP Boolean equation





Outputs	
E	F
0	0
0	0
0	0
1	0
0	0
1	0
1	0
0	1

- Another form to represent the PLA array
 - Use dot to represent AND or OR gate

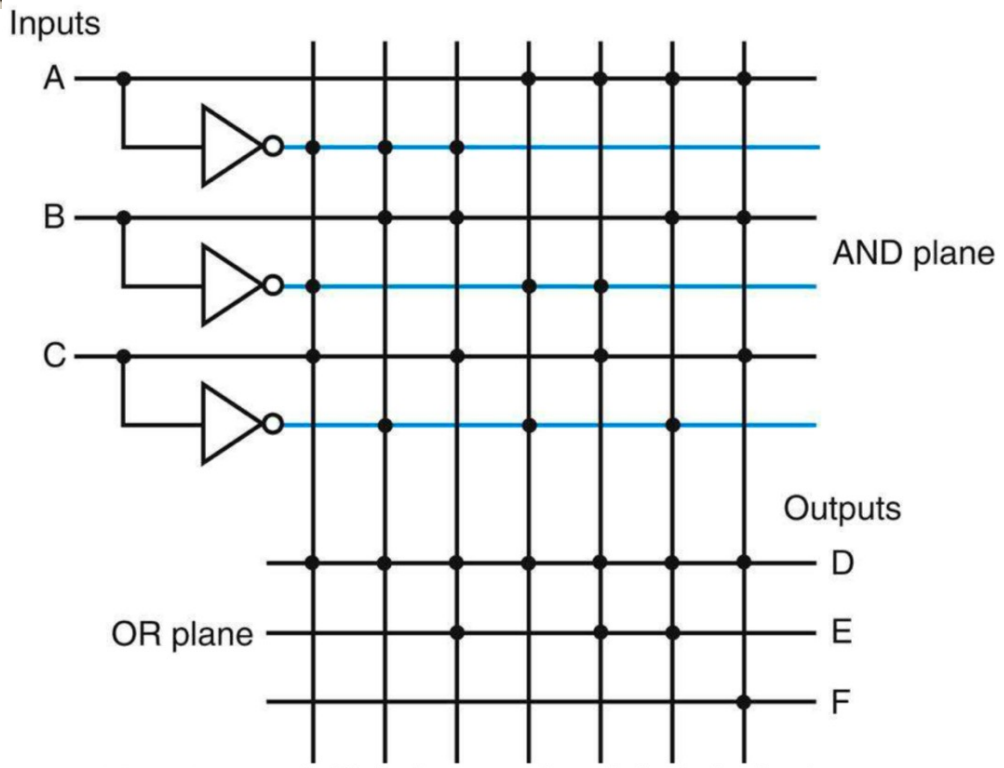


FIGURE A.3.5 A PLA drawn using dots to indicate the components of the product terms and sum terms in the array.

Read-Only-Memory (ROM) Logic for the Truth Table

- Hard-code the truth table in the logic so the output can be read given an input
- The truth table is stored as in memory
 - Address is the input
 - Value is the output
- ROM for the truth table has 3×8 bits for the three outputs

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

PLA vs ROM

- ROM is fully decoded, contain the full output for every possible input
 - Number of entries grows exponentially with regards to the number of inputs
- PLA partially decoded, no need for all the possible input most of the time

Don't Care

- Output don't cares and input don't cares
- Consider a logic function with inputs A , B , and C defined as follows:
 - If A or C is true, then output D is true, whatever the value of B .
 - If A or B is true, then output E is true, whatever the value of C .
 - Output F is true if exactly one of the inputs is true, although we don't care about the value of F , whenever D and E are both true.

Truth Table without Don't Cares

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Truth Table with Don't Cares

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

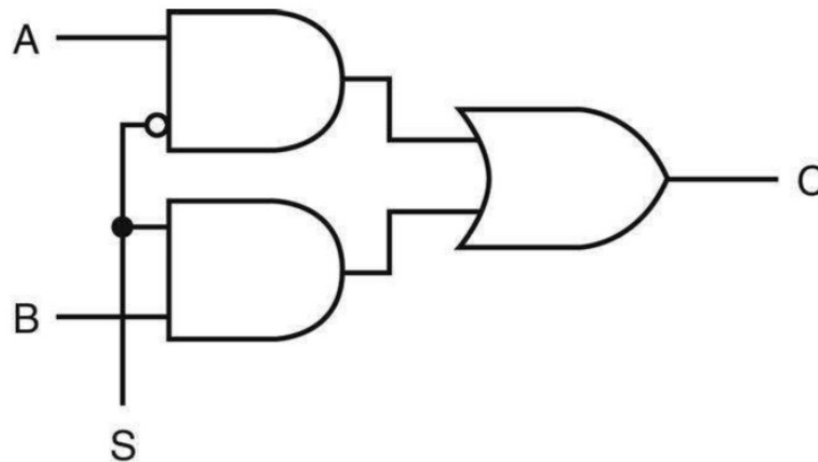
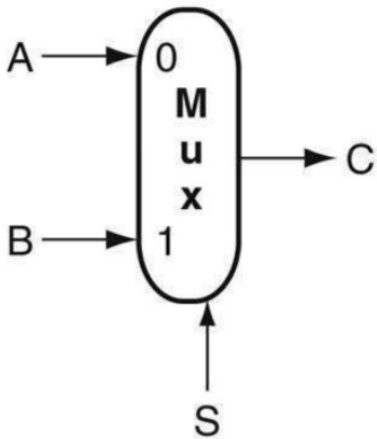
Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

Multiplexors

- Selectors: the output is from one of the two inputs (A and B) according to a control input (S)

- $C = A * S + B * S$

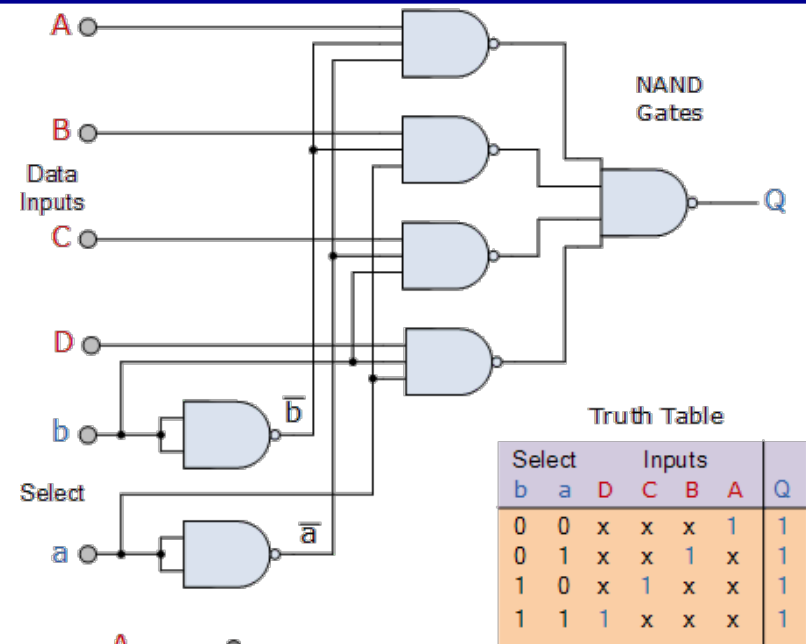
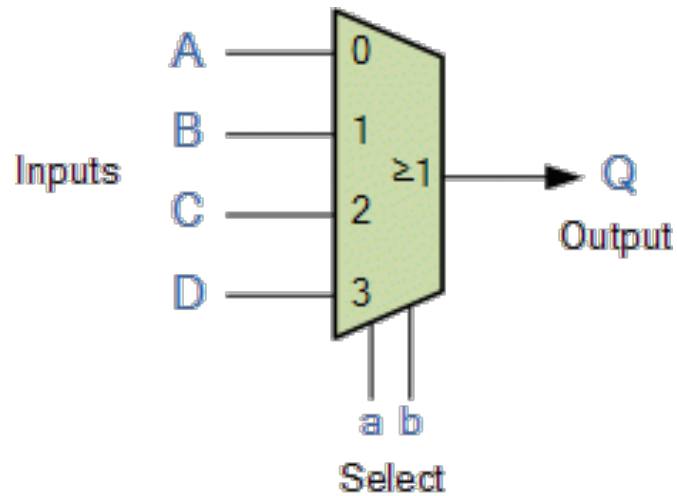
- 1-bit multiplexor



A	B	S	C
0	x	0	0
1	x	0	1
x	0	1	0
x	1	1	1

- Extended to select x-bit width input/output
- Extended to select from n number of inputs
 - Need $\log_2 n$ select bits

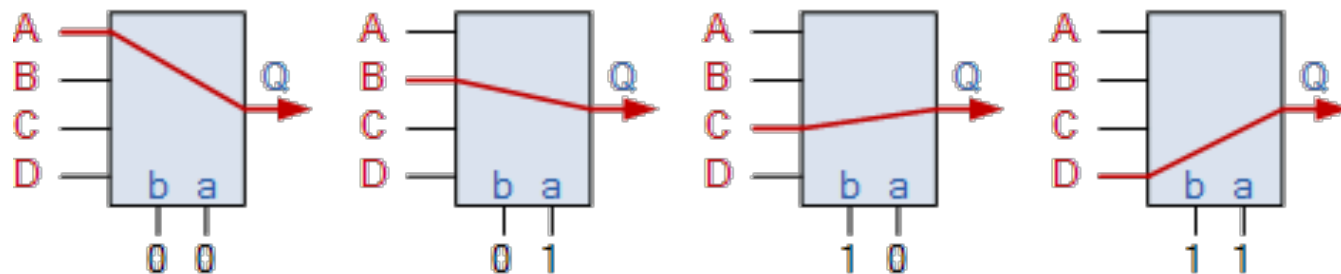
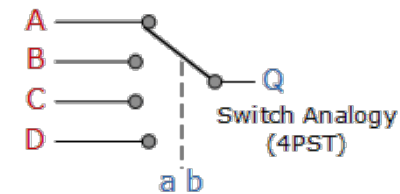
4-to-1 Multiplexor



Truth Table

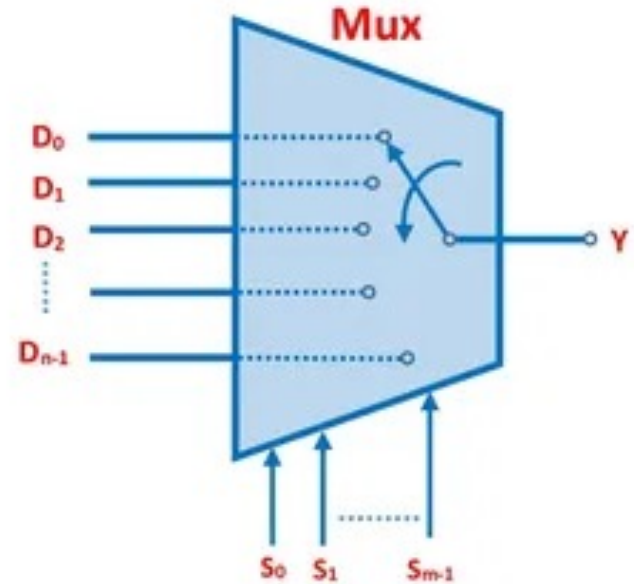
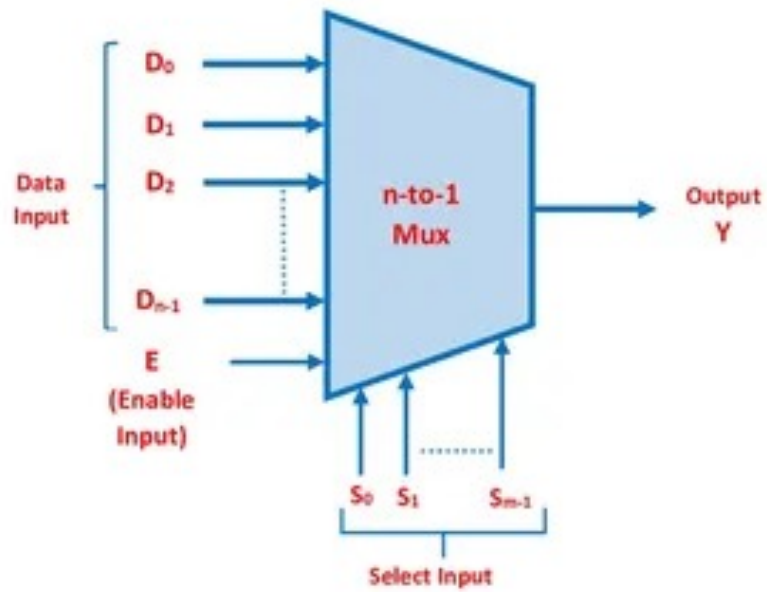
Select	b	a	D	C	B	A	Q
0	0	x	x	x	1	1	1
0	1	x	x	1	x	1	1
1	0	x	1	x	x	1	1
1	1	1	x	x	x	1	1

$$Q = abA + abB + abC + abD$$



https://www.electronics-tutorials.ws/combination/comb_2.html

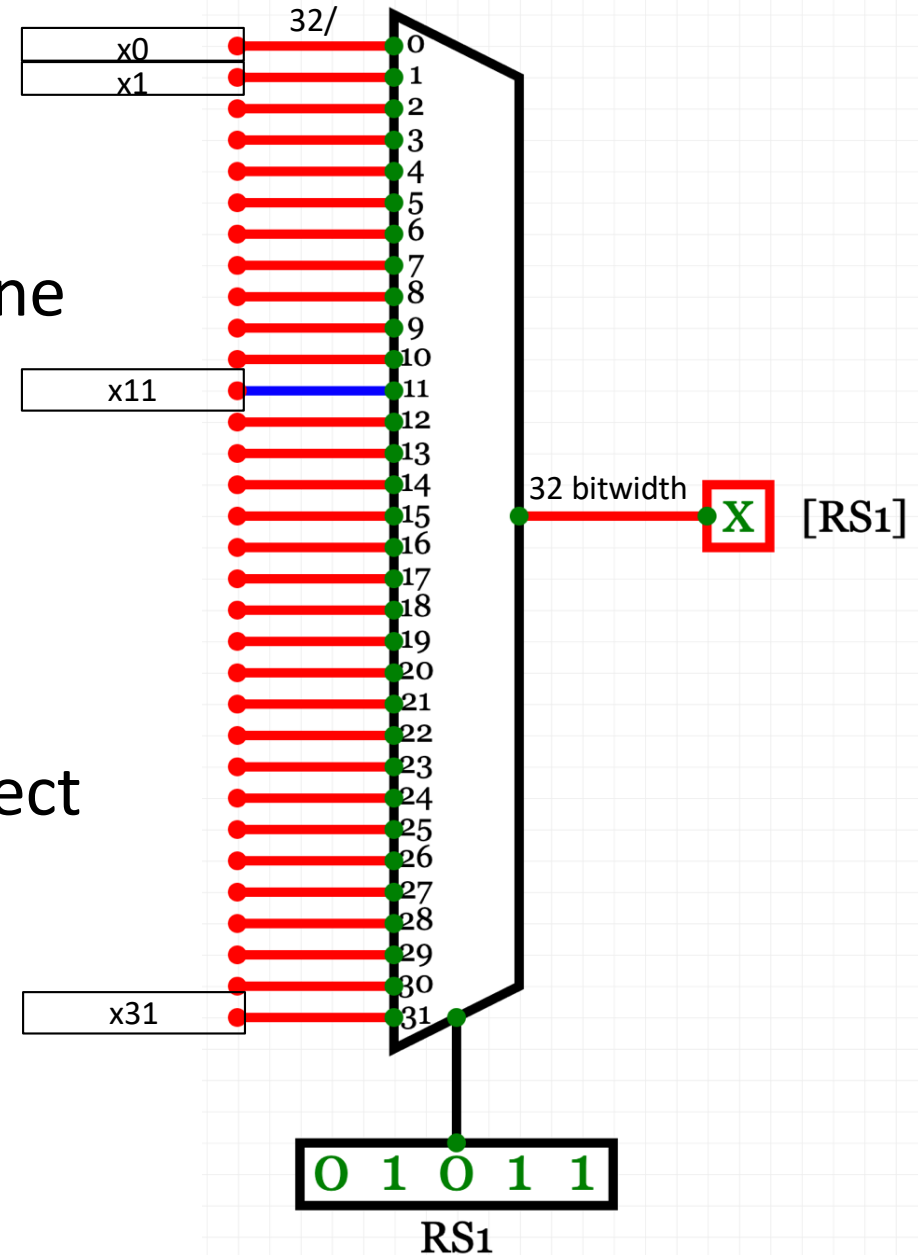
N-to-1 Mux



<https://www.electrical4u.com/multiplexer/>

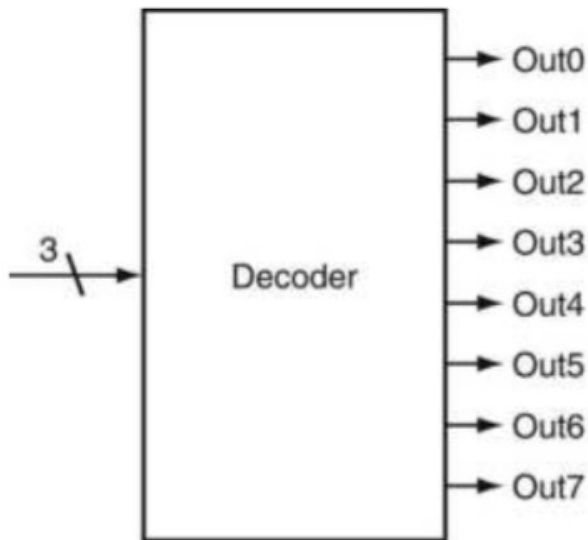
Multiplexors for Selecting Reading from 32 Registers

- Register read: all the 32 registers are being read at the same time
- Only the output of the needed one is selected and sent out
- 32-bit registers
 - Input/output are 32-bit data
- 32 registers, we need 5 bit to select
 - For 32 → 1 selector
- Instruction: add x5, x11, x20



Decoder

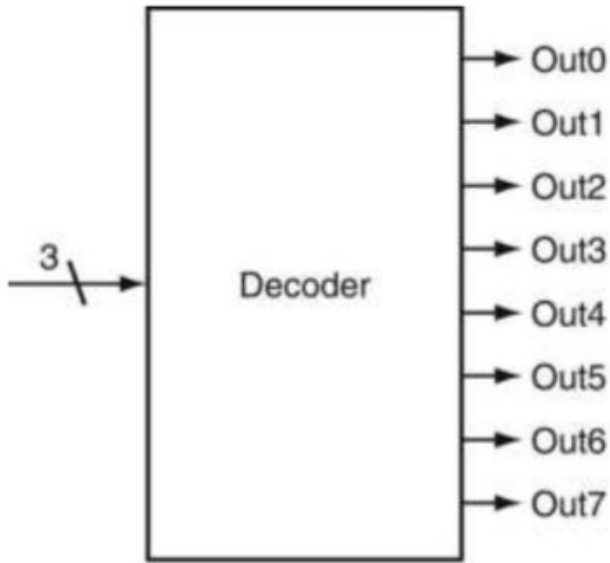
- Translate n-bit input into a single bit that corresponds to the binary value of the n-bit input
- 3-to-8 decoder
 - Inputs are bits of an address
 - Address to enable access to a specific location for that address
 - Access is turned on/off by the single OutX bit.



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

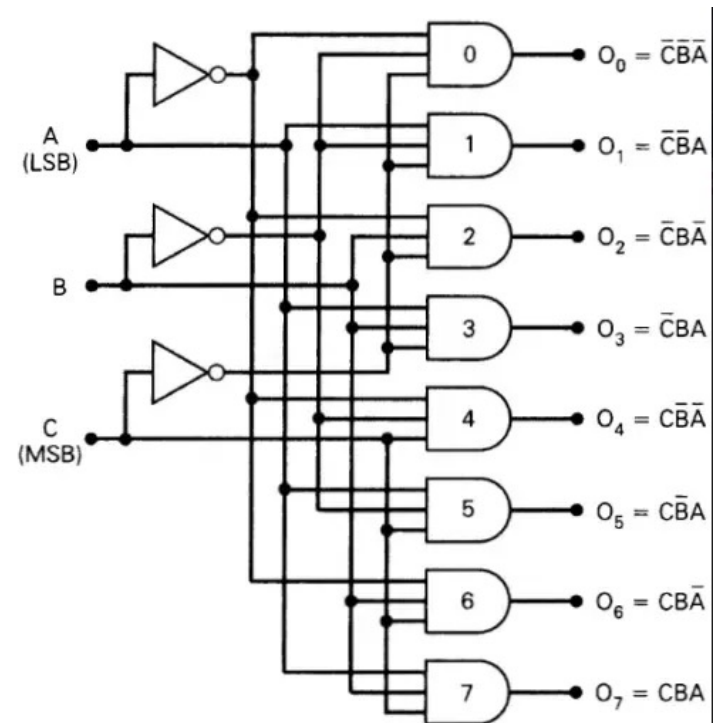


a. A 3-bit decoder

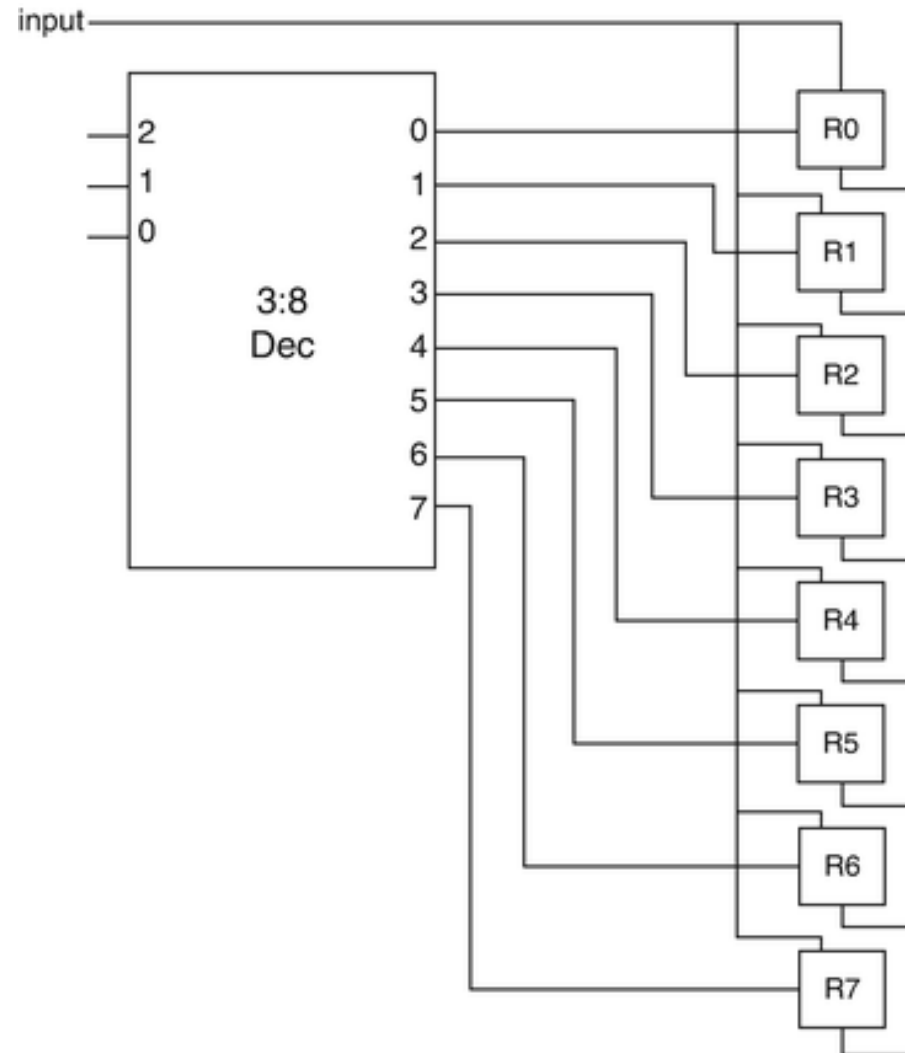
Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

- Boolean equation and logic circuit for 3-to-8 decoder
- A, B, C (or 10, 11, 12) are symbols or bit position of an address

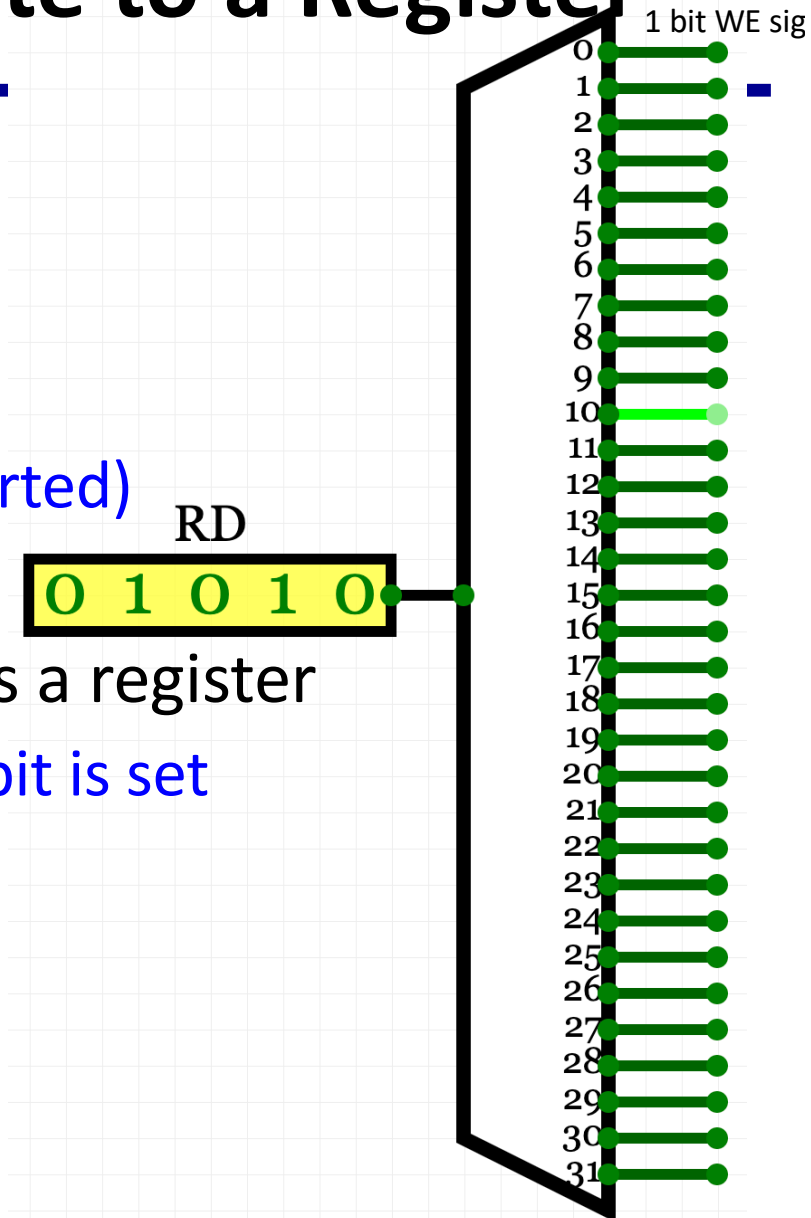


Decoder for Register Write

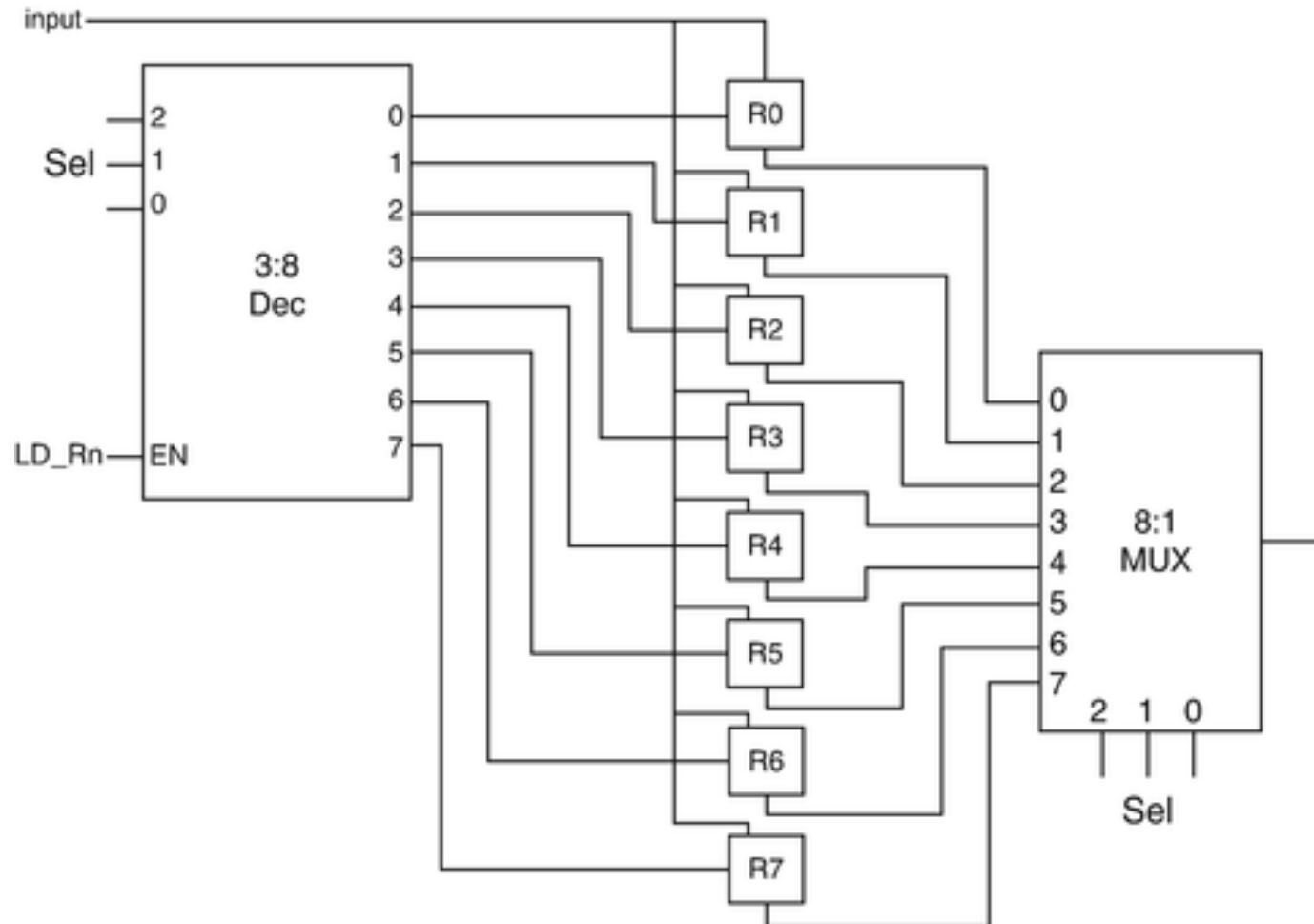


Decoder for Enabling Write to a Register

- Register write: data are sent to all the registers
- Only the selected one is written
 - Write-enable (WE) bit is set to 1 (asserted)
 - We use a decoder for set the WE bit
- 32 registers, we need 5 bit to address a register
 - For each address, the corresponding bit is set
- Instruction: add **r10**, rs1, rs2



Decoder and Mux for Register Write-Read



Lab 7

- **Digital for digital logic design and experiment**
from <https://github.com/hneemann/Digital>
 - A Java software that you download and launch the program

Organization and Do it as Art Work

- 1) Each input and output of a design **MUST** be properly and meaningfully labeled.
- 2) Each component, input and output should be correctly configured in terms of its bitwidth and signal control width.
- 3) Keep wires and components organized and layed out according to the circuit schematics rules, a) Inputs on the left (or top), b) Outputs on right (or bottom), c) gates flow from left to right, d) Straight (not angled) wires are best, e) Wires always connect at a T junction (only 90-degree turn or connection), f) A dot where wires cross indicates a connection between the wires, and g) Wires crossing *without* a dot make no connection. While you may not need to follow all those rules for creating correct and small circuit, it is very important when for creating complex circuit. So, make sure you properly organize the components and wires, make them structured and look good. That will help reduce errors.
- 4) For drawing straight wires using mouse, make one turn per each draw. You should not use one draw to make a connection that needs to have two or more 90-degree turns, which would create angled wires. For those wires, you have to make a wire that has a 90-degree turn, and then connect it with another wire that also make 90 turn, and so on. Try to minimize the turns as much as possible. If two wires have to be crossed, but should not be connected, make sure no dot is marked on where the wire is crossed.

https://passlab.github.io/ITSC3181/notes/Lab_07_IntroMuxDecoder.pdf

Being able to organize complicated things is a skill and ability that can be trained by practice, but hardly a talent.

Appendix A: The Basics of Logic Design

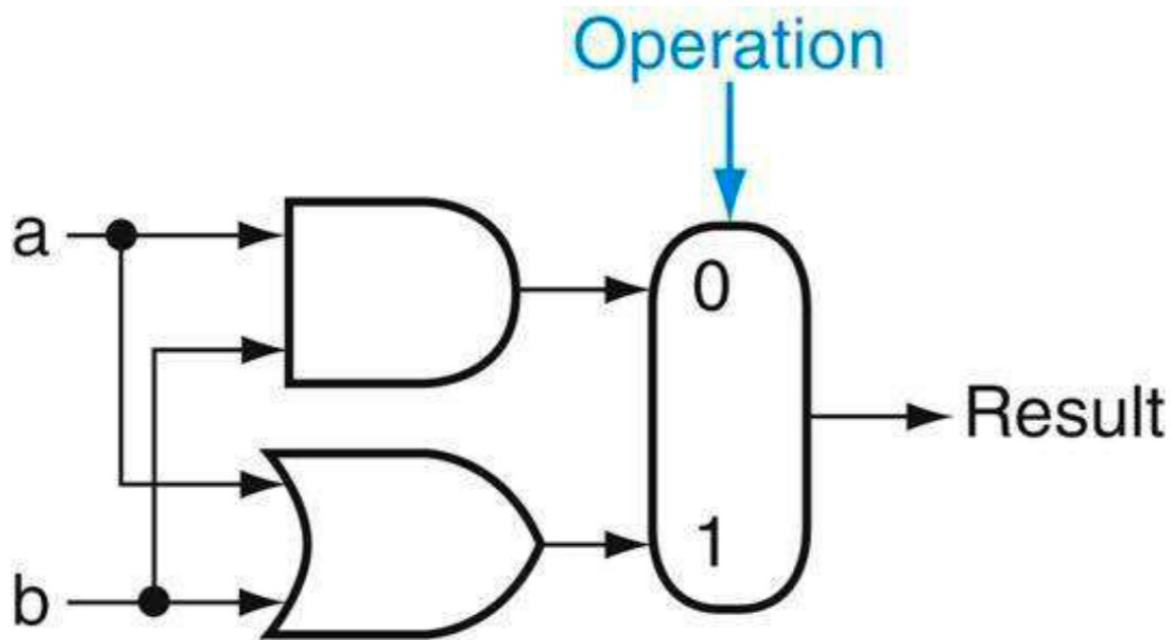
- **Lecture 12**
 - **A.1 Introduction**
 - **A.2 Gates, Truth Tables, and Logic Equation**
- **Lecture 13**
 - **A.3 Combinational Logic**
 - ~~A.4 Using a Hardware Description Language~~
- **Lab 7**
- **Lecture 14**
 - **A.5 Constructing a Basic Arithmetic Logic Unit**
 - ~~A.6 Faster Addition: Carry Lookahead~~
- **Lecture 15**
 - **A.7 Clocks**
 - **A.8 Memory Elements: Flip-Flops, Latches, and Registers**
- **Lab 8**
- **Lecture 16**
 - **A.9 Memory Elements: SRAMs and DRAMs**
- **Lab 9**
- ~~Lecture 17~~
 - ~~A.10 Finite State Machines~~
 - ~~A.11 Timing Methodologies~~
 - ~~A.12. Field Programmable Devices~~
 - **A.13 Concluding Remarks**
 - ~~A.14 Exercises~~
- **Lab 10**

ALU and Bitwidth

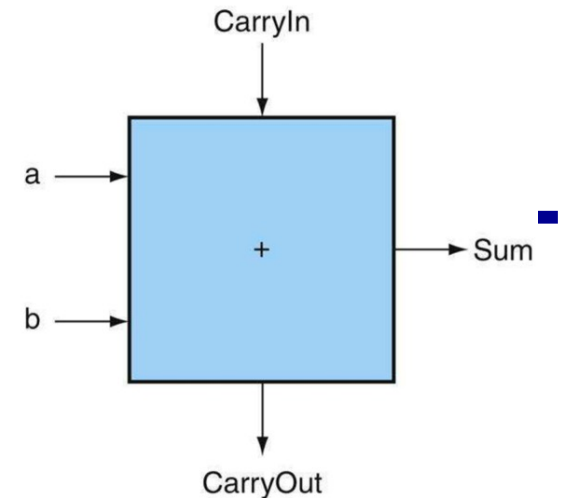
- Arithmetic logic unit (ALU) is the brawn of the computer
 - Perform add, sub, AND, OR, etc.
 - A unit to perform all supported operations
- 64-bit machine (registers are 64-bit wide), we need 64-bit ALU
- 32-bit machine (registers are 32-bit wide), we need 32-bit ALU
 - We will focus 32-bit machine starting from now for the lecture
 - **Choose your bitwidth of your CPU design for your labs 07 – 10**
 - **32-bit, 16-bit, 8-bit**
 - **More bitwidth → more wire/complexity**
 - **8-bit is good choice, 16-bit is nice, 32-bit is challenging**
- Starting with 1-bit ALU first
 - Logic, and then Arithmetic

1-Bit ALU: Logic Unit (AND and OR)

- Logics circuit does AND and OR operations, **operation** is used to select the output of the gate as the result.
 - 0: AND
 - 1: OR



1-Bit ALU adder

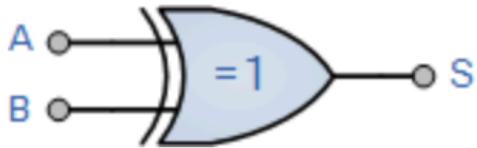


- For add, 1-bit adder
 - 3 Inputs: a, b, and CarryIn
 - 2 Outputs: sum and CarryOut

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{\text{two}}$
0	0	1	0	1	$0 + 0 + 1 = 01_{\text{two}}$
0	1	0	0	1	$0 + 1 + 0 = 01_{\text{two}}$
0	1	1	1	0	$0 + 1 + 1 = 10_{\text{two}}$
1	0	0	0	1	$1 + 0 + 0 = 01_{\text{two}}$
1	0	1	1	0	$1 + 0 + 1 = 10_{\text{two}}$
1	1	0	1	0	$1 + 1 + 0 = 10_{\text{two}}$
1	1	1	1	1	$1 + 1 + 1 = 11_{\text{two}}$

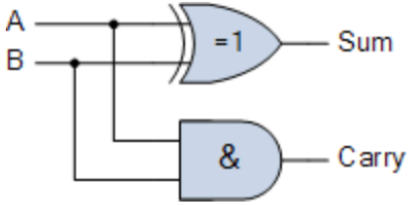
XOR Gate

- It does addition, no CarryIn or CarryOut
 - Sum = A XOR B = $A \oplus B$

Symbol	Truth Table		
 <p>2-input Ex-OR Gate</p>	B	A	S
	0	0	0
	0	1	1
	1	0	1
	1	1	0

1-Bit Half-Adder

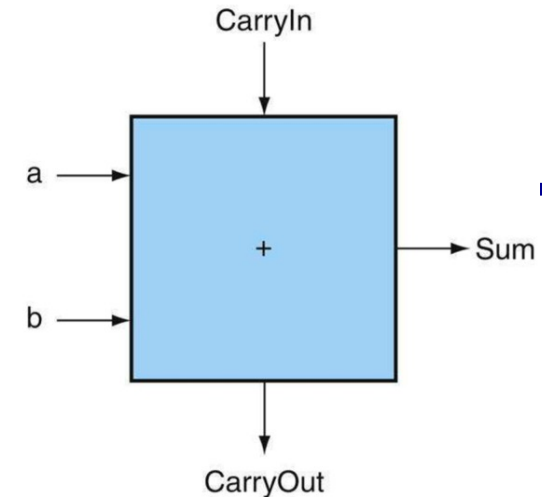
- 1-Bit Half-Adder
 - Two inputs: A and B
 - Two outputs: S and CarryOut

Symbol	Truth Table			
	B	A	SUM	CARRY
	0	0	0	0
	0	1	1	0
	1	0	1	0
	1	1	0	1

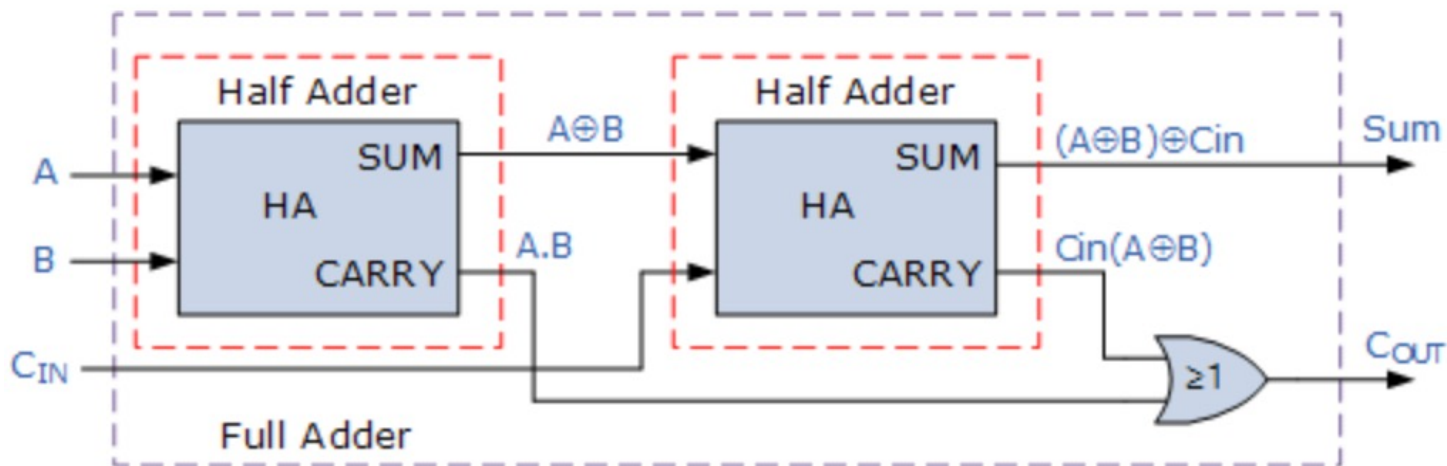
- Sum = A XOR B = $A \oplus B$
- CarryOut = A*B

1-Bit Full Adder

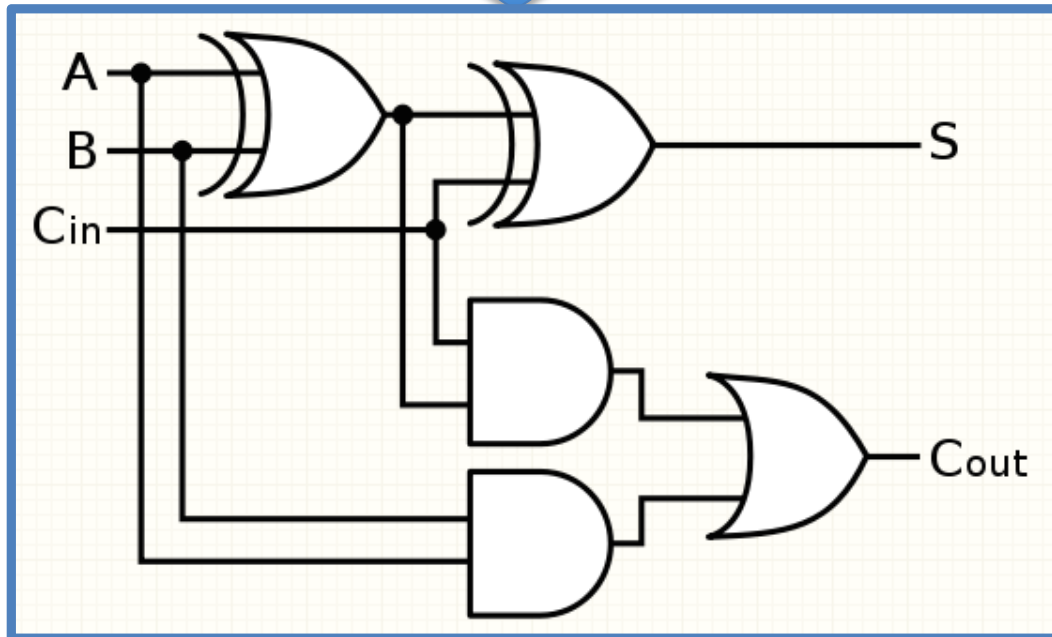
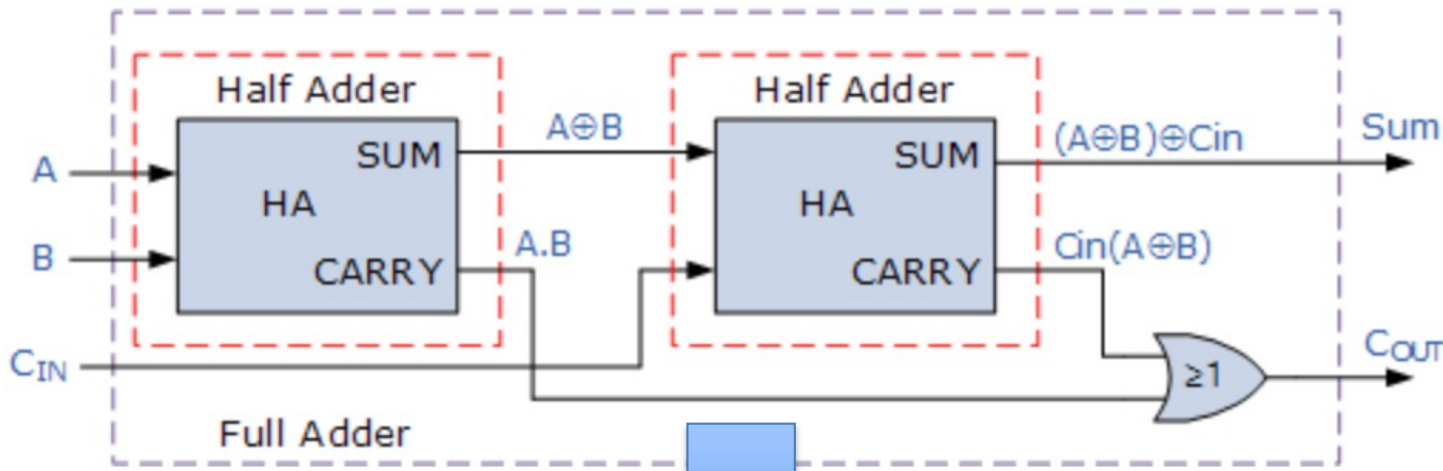
- 1-Bit Full-Adder
 - Three inputs: A, B and CarryIn
 - Two outputs: S and CarryOut



- Combine two half-adder to a full adder



1-Bit Full Adder

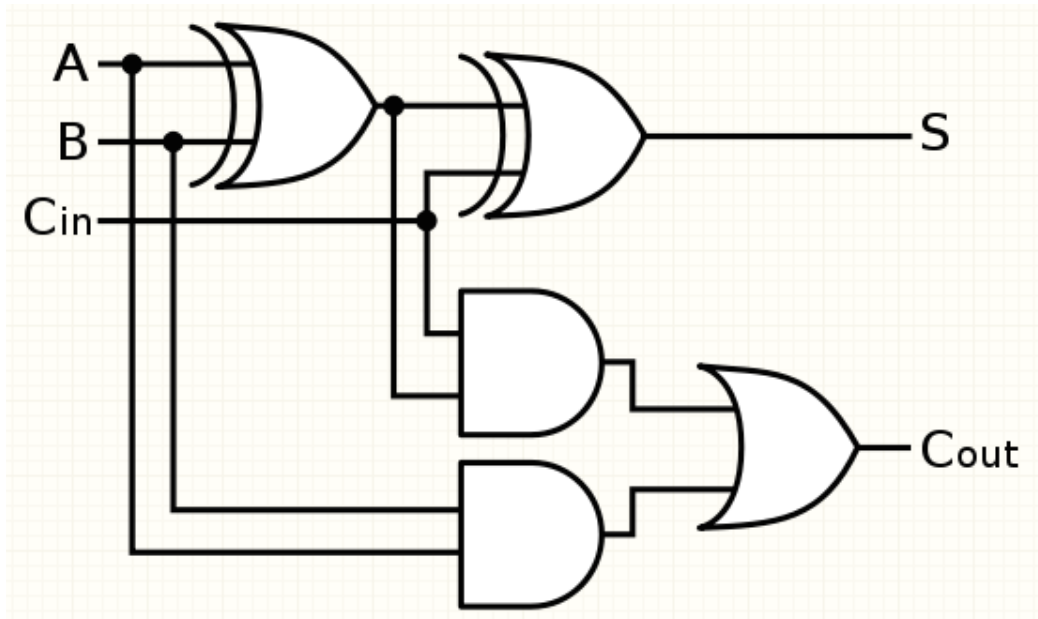


Symbol	
A	B
0	0
0	1
1	1

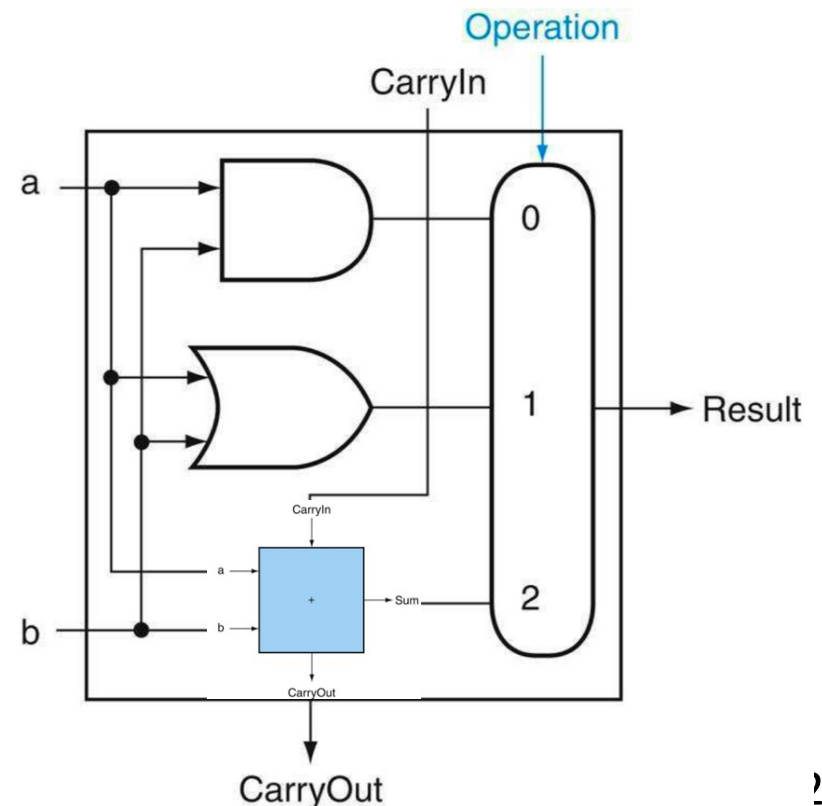
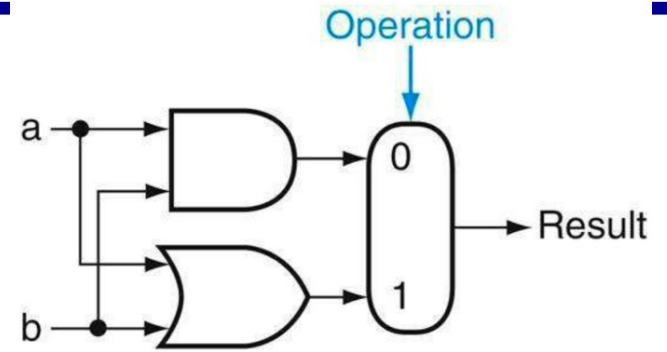
Truth Table				
C-in	B	A	Sum	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

1-Bit ALU that can do **add**, **AND**, and **OR**.

- 1-Bit Full Adder + AND/OR logic unit

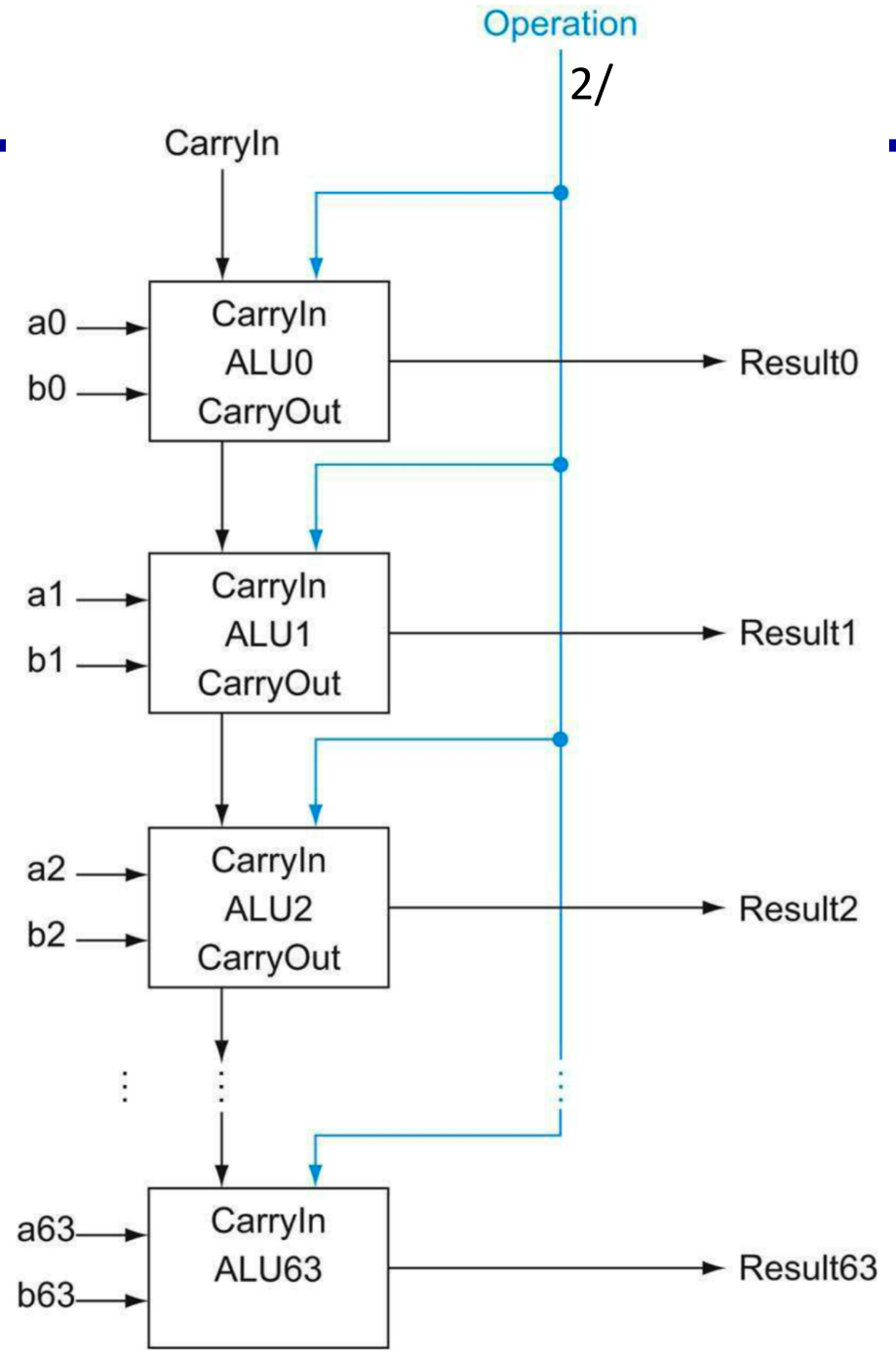


- Operation has 2 bits for the mux
 - **AND: 00**
 - **OR: 01**
 - **ADD: 10**



64-Bit ALU

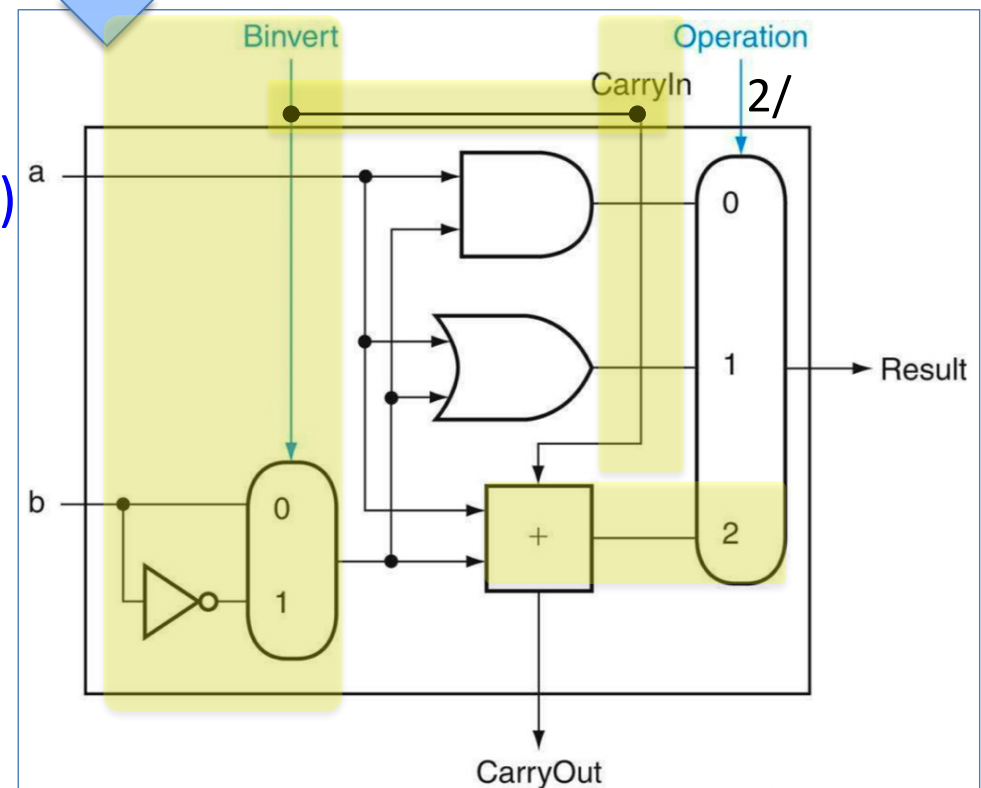
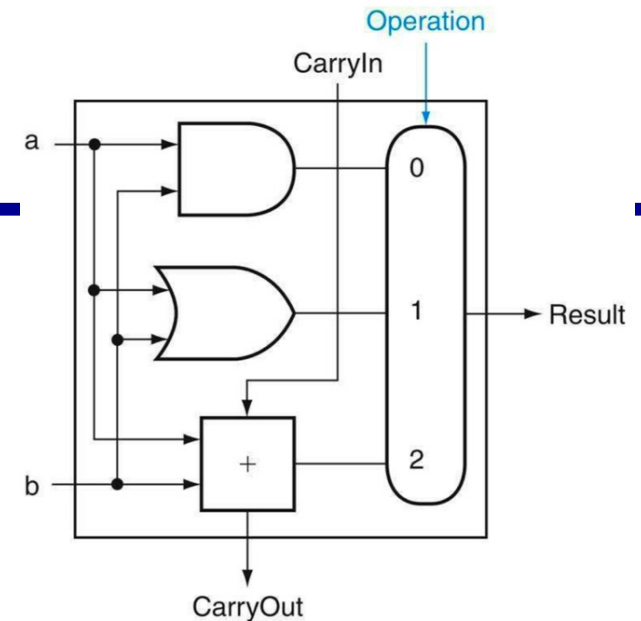
- 64 input bits are split and fed to each 1-bit ALU
- Results/sums of each 1-bit ALU are combined into a single 64-bit double word as the result of the ALU
- **CarryOut goes to CarryIn of the ALU for next bit**
 - **CarryOut of 63-rd bit is the overflow**
- Operation bits go to all adders



Extending ALU to DO

A Little More: sub

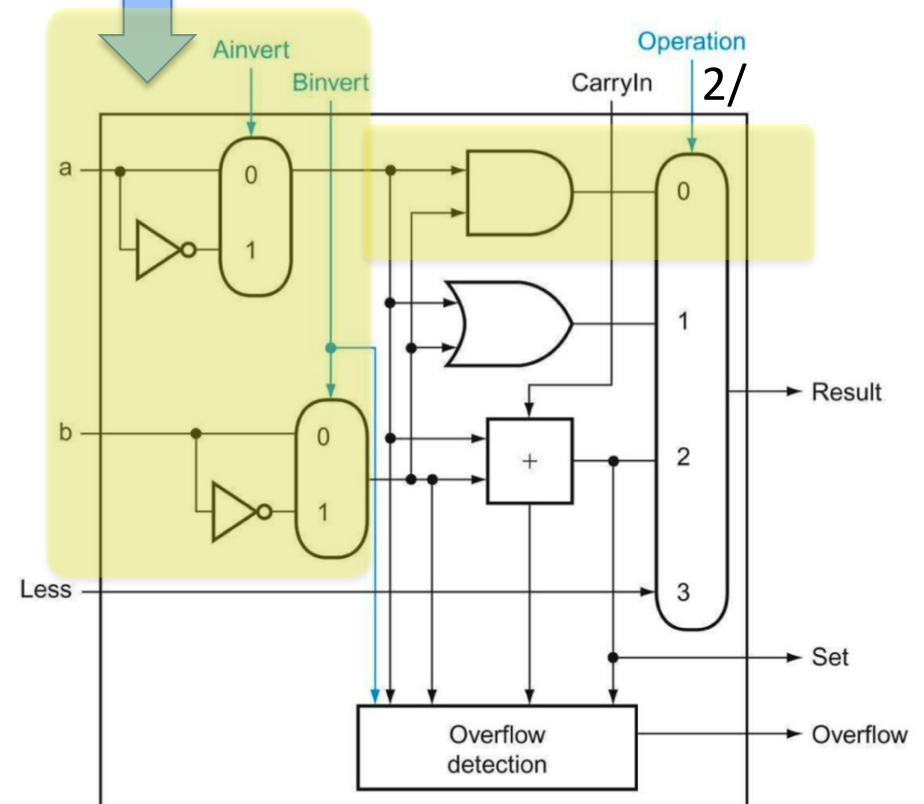
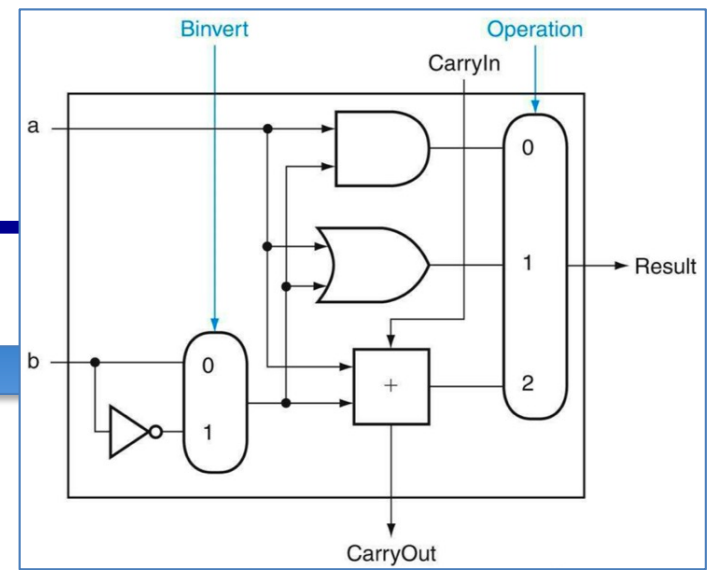
- Starting from the 1-Bit ALU
 - Can do add, AND and OR
- Add subtraction to the ALU
 - $a - b = a + (-b) = a + b' + 1$ since
 - For 2's complement representation, $-b = b' + 1$
 - Thus we just need to revert (NOT) b and “add 1”
 - Use the carryIn for “add 1”
 - A Mux to select b or b'
 - if doing sub
 - Binvert is 1
 - CarryIn is 1
 - Operation is still add (10)



Extending ALU to DO

A Little More: NOR

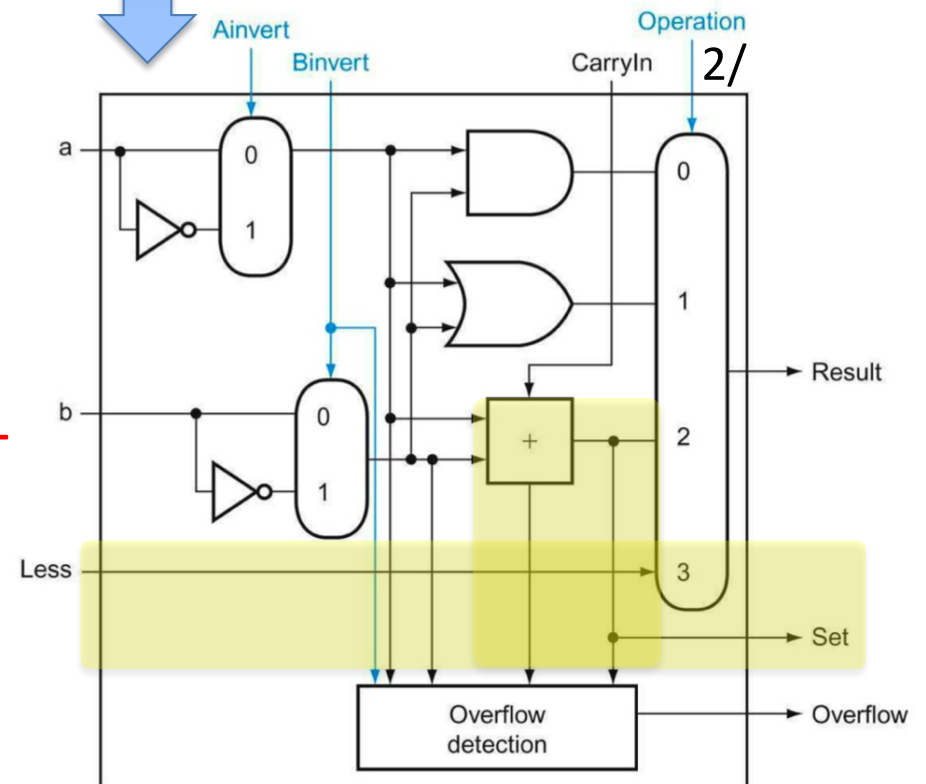
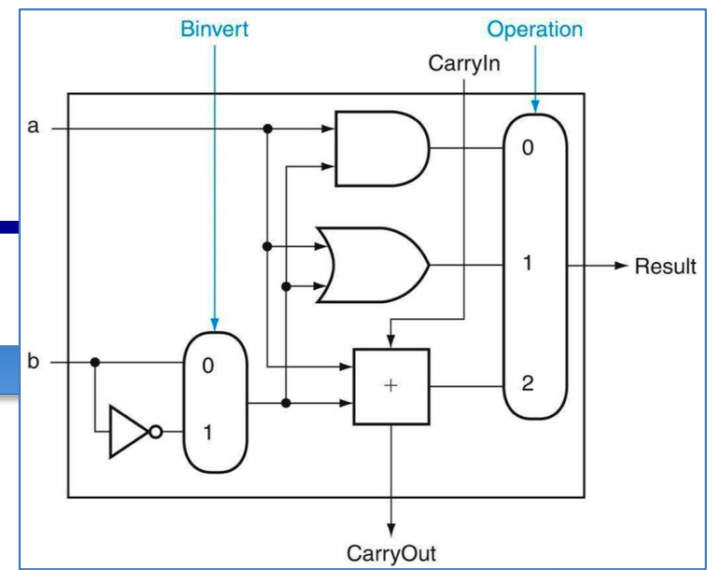
- Current 1-Bit ALU
 - Can do add, sub, AND and OR
- Add NOR operation
 - $(a + b)' = a' * b'$
 - Just need to add NOT for a, and a mux to select a or a'
- For doing NOR
 - Operation is AND (00)
 - Ainvert is 1
 - Binvert is 1



Extending ALU to DO

A Little More: SLT (1/2)

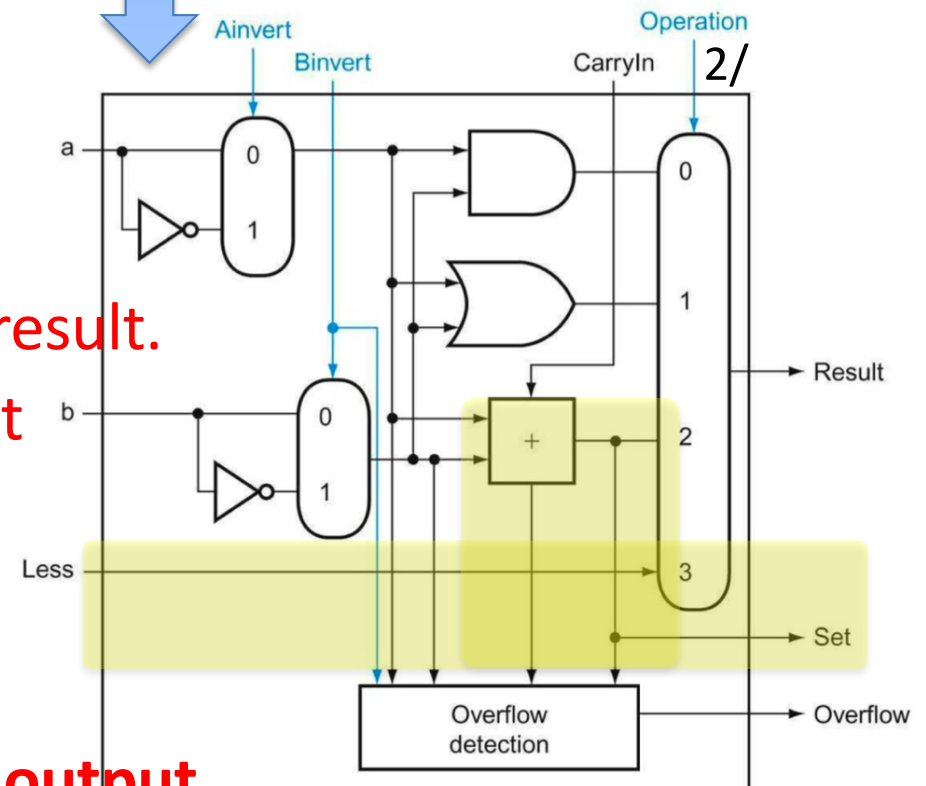
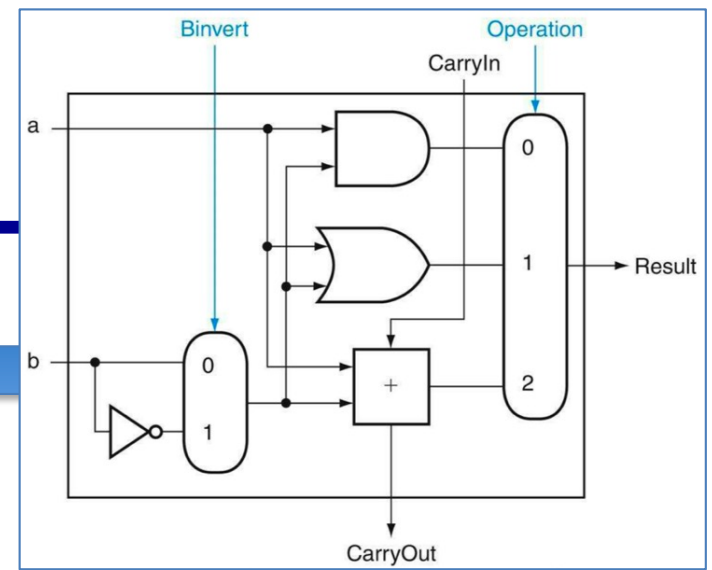
- Current 1-Bit ALU
 - Can do add, sub, AND, OR and NOR
- slt (set less than) instruction
 - `slt rd, rs1, rs2`
 - if ($[rs1] < [rs2]$) $[rd] = 1$
else $[rd] = 0$
 - `slti rd, rs1, #immediate`
 - if ($[rs1] < \#immediate$) $[rd] = 1$
else $[rd] = 0$
 - For the ALU, a is $[rs1]$, b is $[rs2]$
and result is $[rd]$



Extending ALU to DO

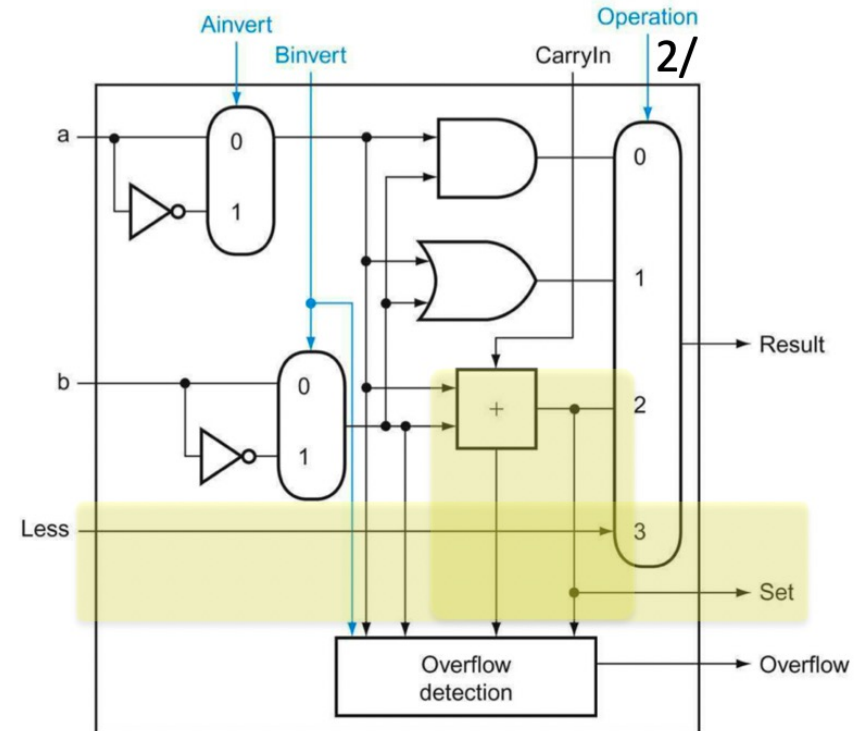
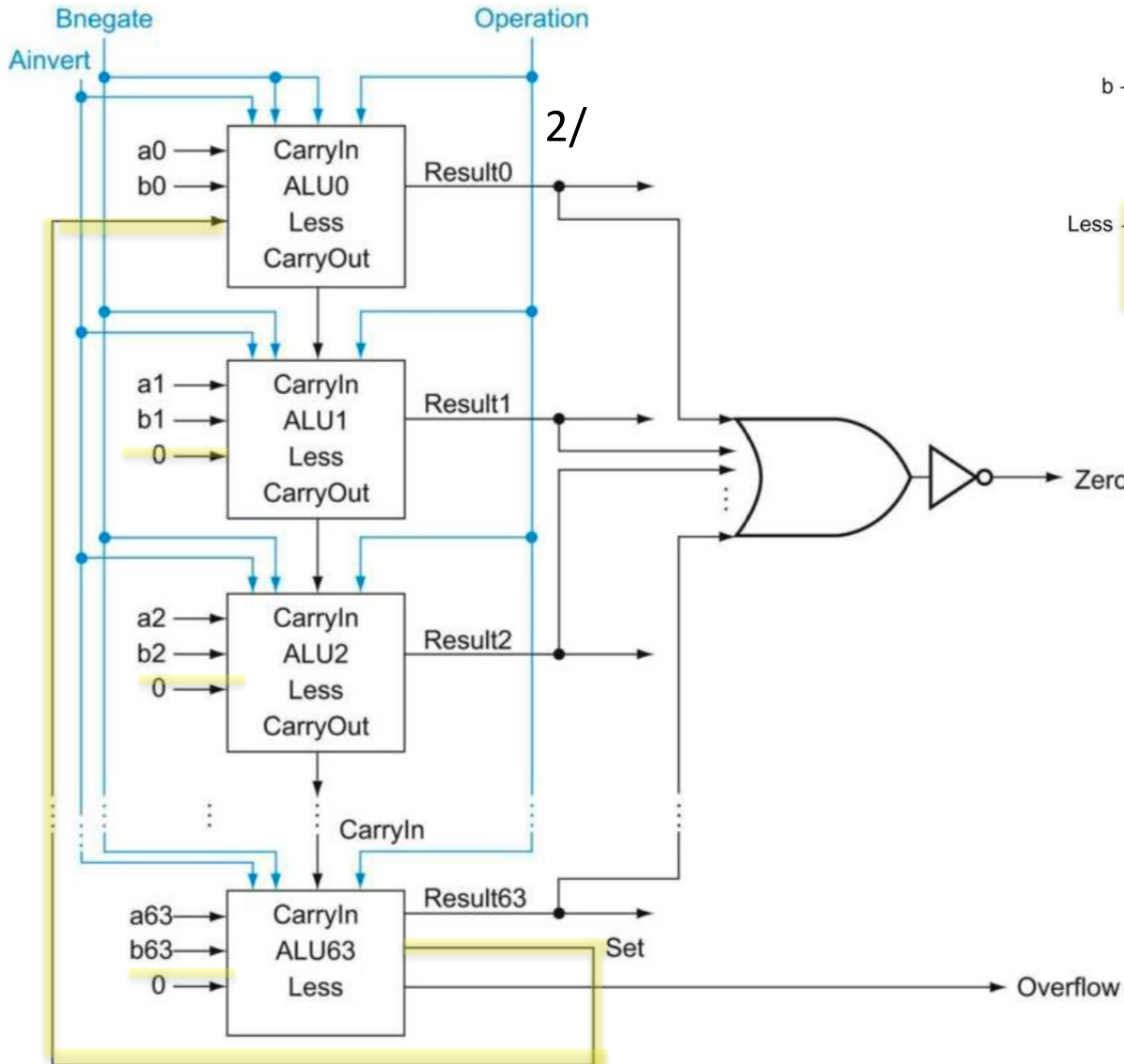
A Little More: SLT (2/2)

- Current 1-Bit ALU
 - Can do add, sub, AND, OR and NOR
- slt (set less than) instruction
 - Implementation:
 - **$(a < b) == (a-b) < 0$**
 - Thus slt is to perform sub and then check the sign bit of the result.
 - A set bit from the adder output is used to pass through the sign bit from MSB to LSB
 - **SLT has its own operation code (11) and less goes to the output**



SLT in 64-Bit ALU

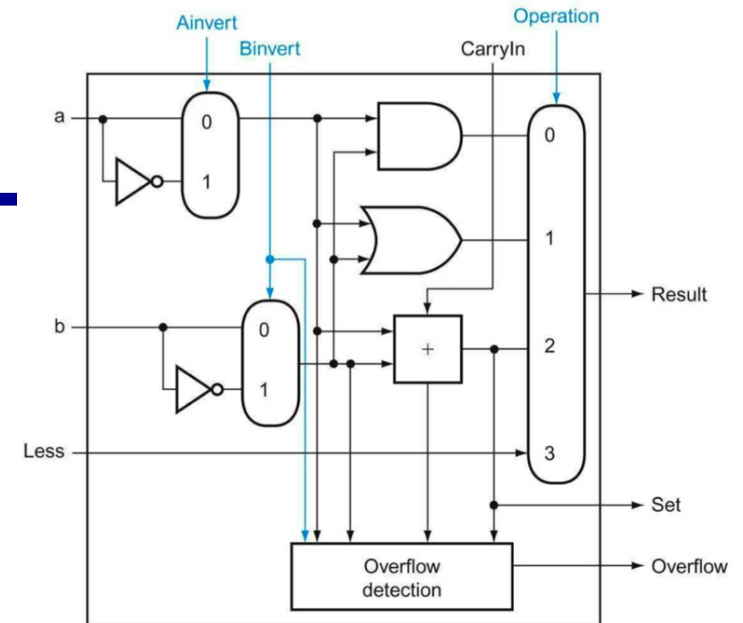
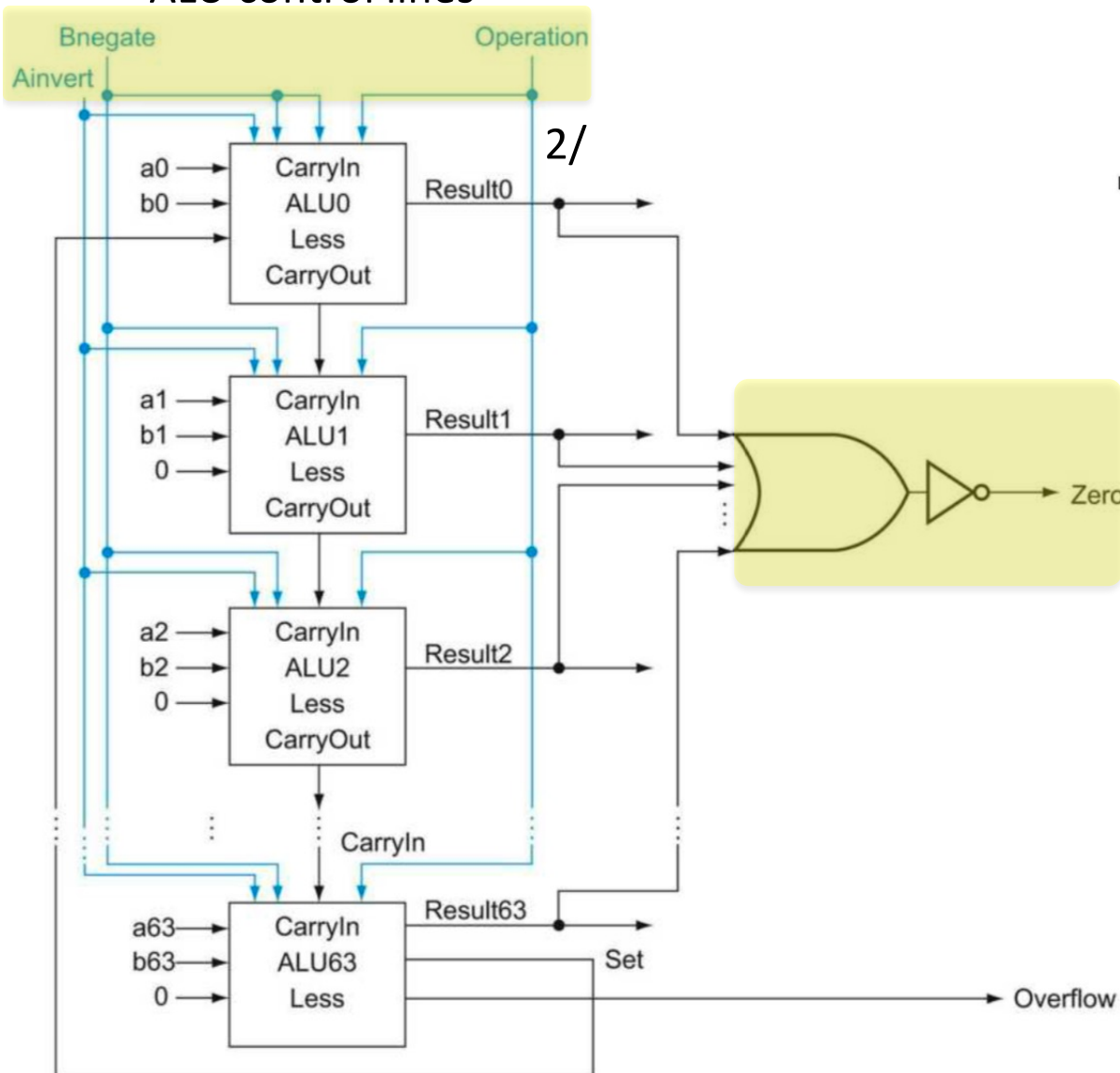
ALU control lines



- For SLT (set less than)
 - Operation is 11
 - MSB set, bit 63, which is the sign bit goes to LSB as the result
 - If negative, set of 63 is 1, thus [rd] = 1
 - All others are 0

Check zero in 64-Bit ALU

ALU control lines

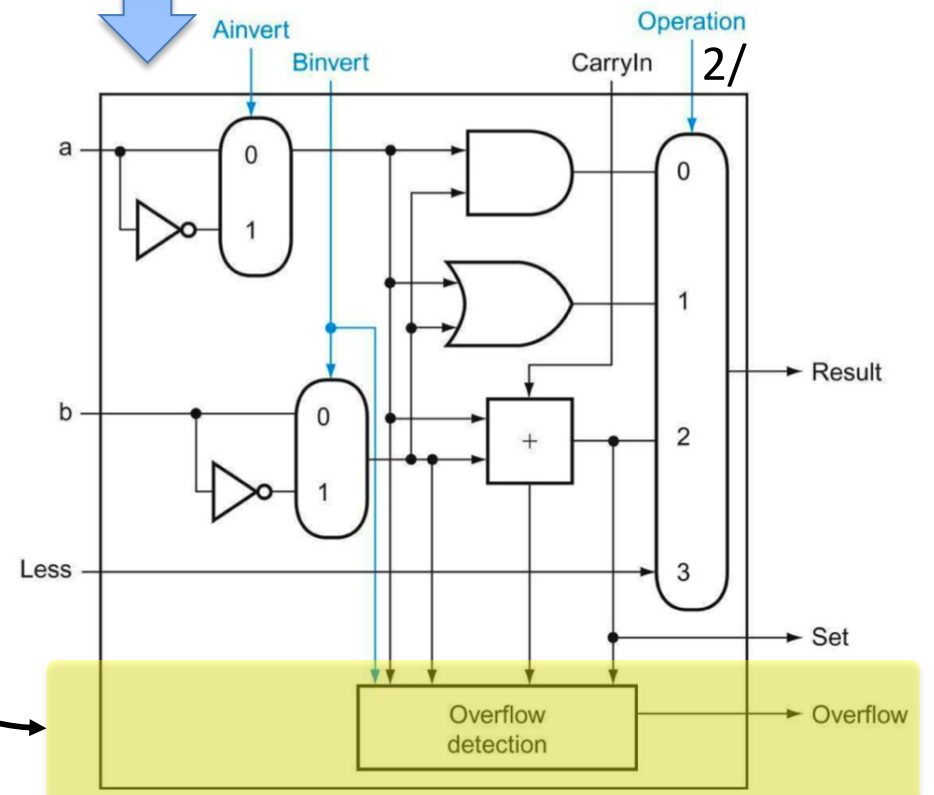
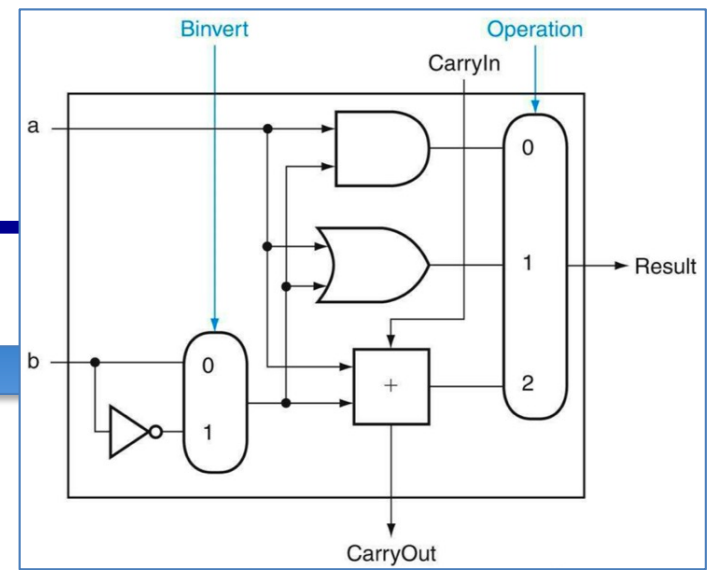


- Check zero of any operations
 - NOR of all result bit
- Usage, e.g.
 - For beq rs1, rs2, label instruction
 - Which do [rs1] – [rs2] first, and check whether result is zero or not

Extending ALU to DO

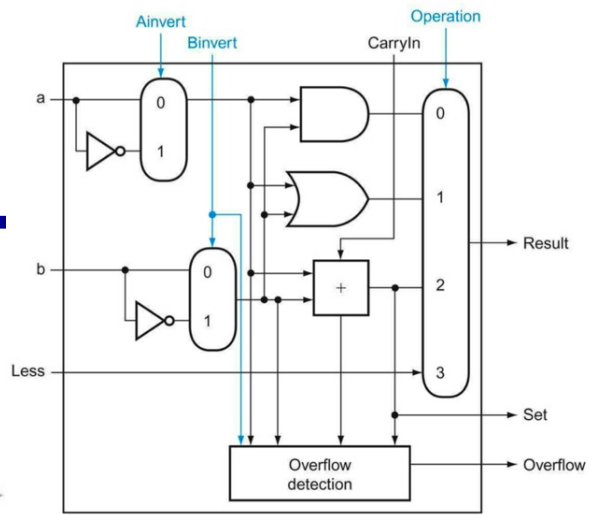
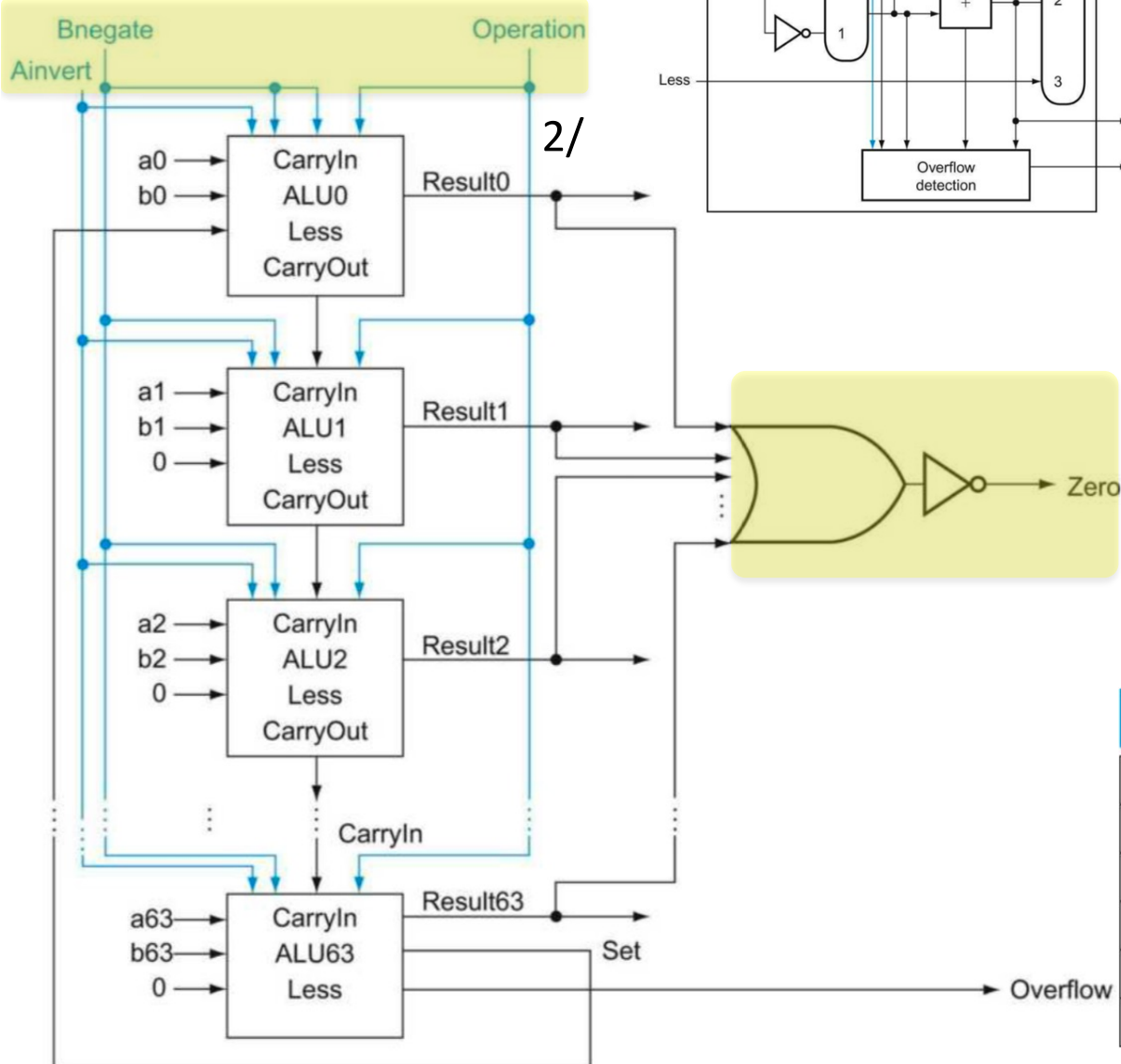
A Little More: overflow

- Current 1-Bit ALU
 - Can do add, sub, AND, OR, NOR and SLT
- Overflow detection is optional
 - Can be just CarryOut
- **We will NOT do this in the lab**



64-Bit ALU

ALU control lines



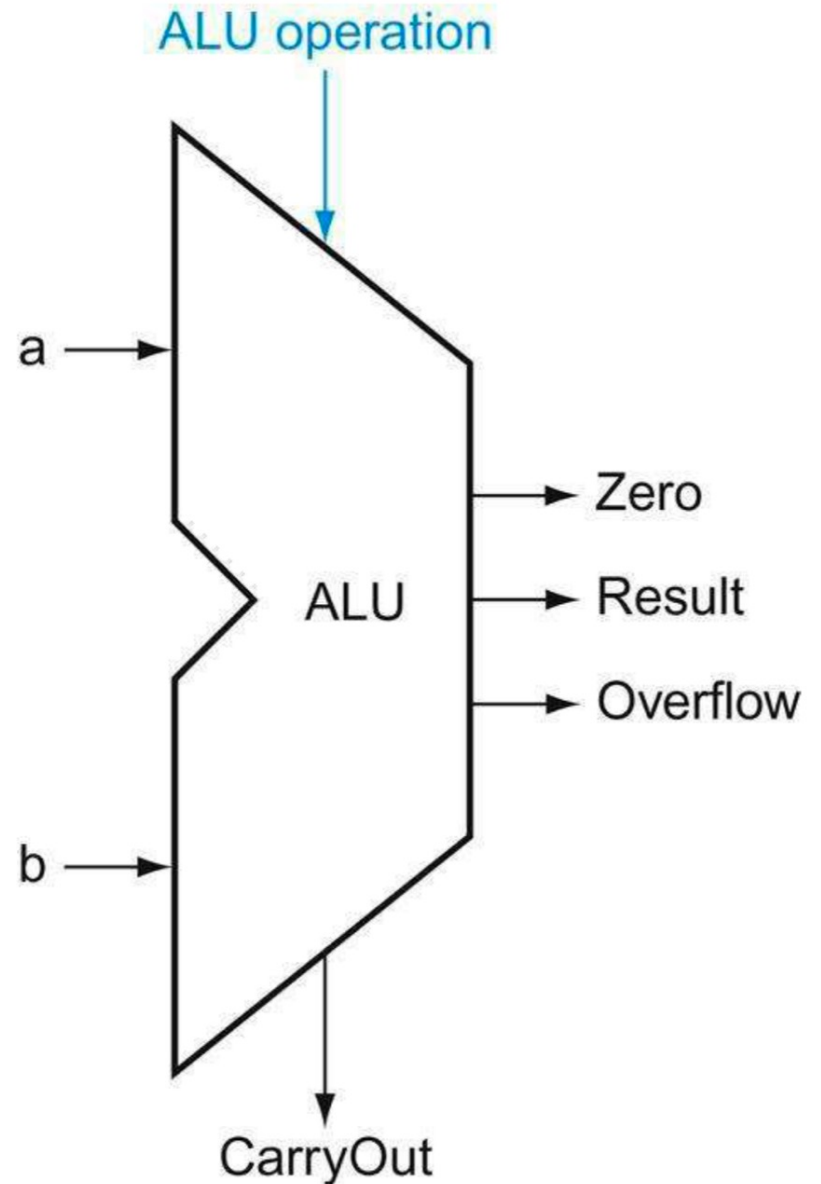
- **Operation:**
 - **AND: 00**
 - **OR: 01**
 - **ADD: 10**
 - **SUB: 10 (same as ADD)**
 - **SLT: 11**
 - **NOR: 00 (same as AND)**
- **4 bits ALU controls:**
 - **Ainvert: for NOR**
 - **Bnegate: for sub and slt**
 - **2-bit operation**

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

ALU Symbol

- 4-bit ALU operation
 - Ainvert, Bnegate, 2-bit operation

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR



Examples using Hardware Description Language: Verilog for Half-adder

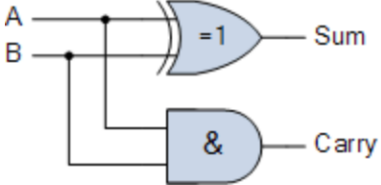
- HDL **describe** the behavior of the logic

- **logic synthesis** converts to gates

- Half-adder

- $\text{Sum} = A \text{ XOR } B = A \oplus B$

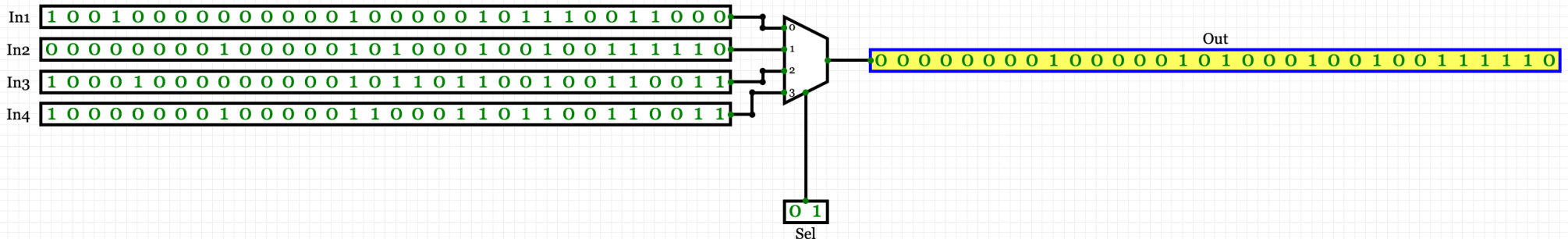
- $\text{CarryOut} = A * B$

Symbol		Truth Table			
		B	A	SUM	CARRY
		0	0	0	0
		0	1	1	0
		1	0	1	0
		1	1	0	1

```
module half_adder (A,B,Sum,Carry);  
    input A,B; //two 1-bit inputs  
    output Sum, Carry; //two 1-bit outputs  
    assign Sum = A ^ B; //sum is A xor B  
    assign Carry = A & B; //Carry is A and B  
endmodule
```

Verilog for 4-to-1 Mux

- 4-to-1 Mux (32-bit in the picture, 64-bit in the code)



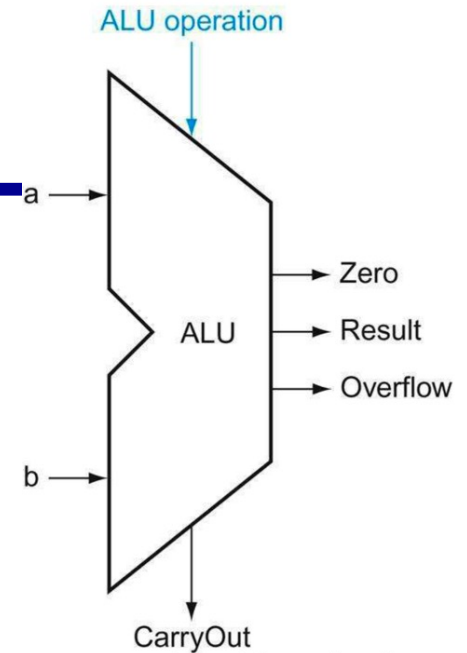
- always @(list of signals that cause reevaluation)
 - Re-evaluate the assignment if any of the sensitive list changes

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
    input [63:0] In1, In2, In3, In4; //four 64-bit inputs
    input [1:0] Sel; //selector signal
    output reg [63:0] Out; //64-bit output
    always @(In1, In2, In3, In4, Sel)
    case (Sel) // a 4->1 multiplexor
        0: Out <= In1;
        1: Out <= In2;
        2: Out <= In3;
        default: Out <= In4;
    endcase
endmodule
```

Verilog for ALU

- HDL **describe** the behavior of the logic
 - **logic synthesis** converts to gates

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
  input [3:0] ALUctl;
  input [63:0] A,B;
  output reg [63:0] ALUOut;
  output Zero;
  assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
  always @(ALUctl, A, B) begin //reevaluate if these change
    case (ALUctl)
      0: ALUOut <= A & B;
      1: ALUOut <= A | B;
      2: ALUOut <= A + B;
      6: ALUOut <= A - B;
      7: ALUOut <= A < B ? 1 : 0;
      12: ALUOut <= ~(A | B); // result is nor
      default: ALUOut <= 0;
    endcase
  end
endmodule
```



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

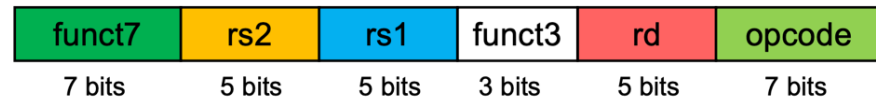
Verilog for a Complete ALU for RISC-V

- 6-bit function code derived from the instruction decoding
 - Pick the needed bits from, e.g. opcode/func3/func7
- Use function code to assign the 2-bit ALUop code
 - E.g. for Load and store instruction, **add** ALUop is used to signal the ALU to perform **add operation of base and offset**

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
  input [1:0] ALUOp;
  input [5:0] FuncCode;
  output [3:0] reg ALUCtl;
  always case (FuncCode)
    32: ALUOp<=2; // add
    34: ALUOp<=6; // subtract
    36: ALUOp<=0; // and
    37: ALUOp<=1; // or
    39: ALUOp<=12; // nor
    42: ALUOp<=7; // slt
    default: ALUOp<=15; // should not happen
  endcase
endmodule
```

R-Format Encoding Example 2

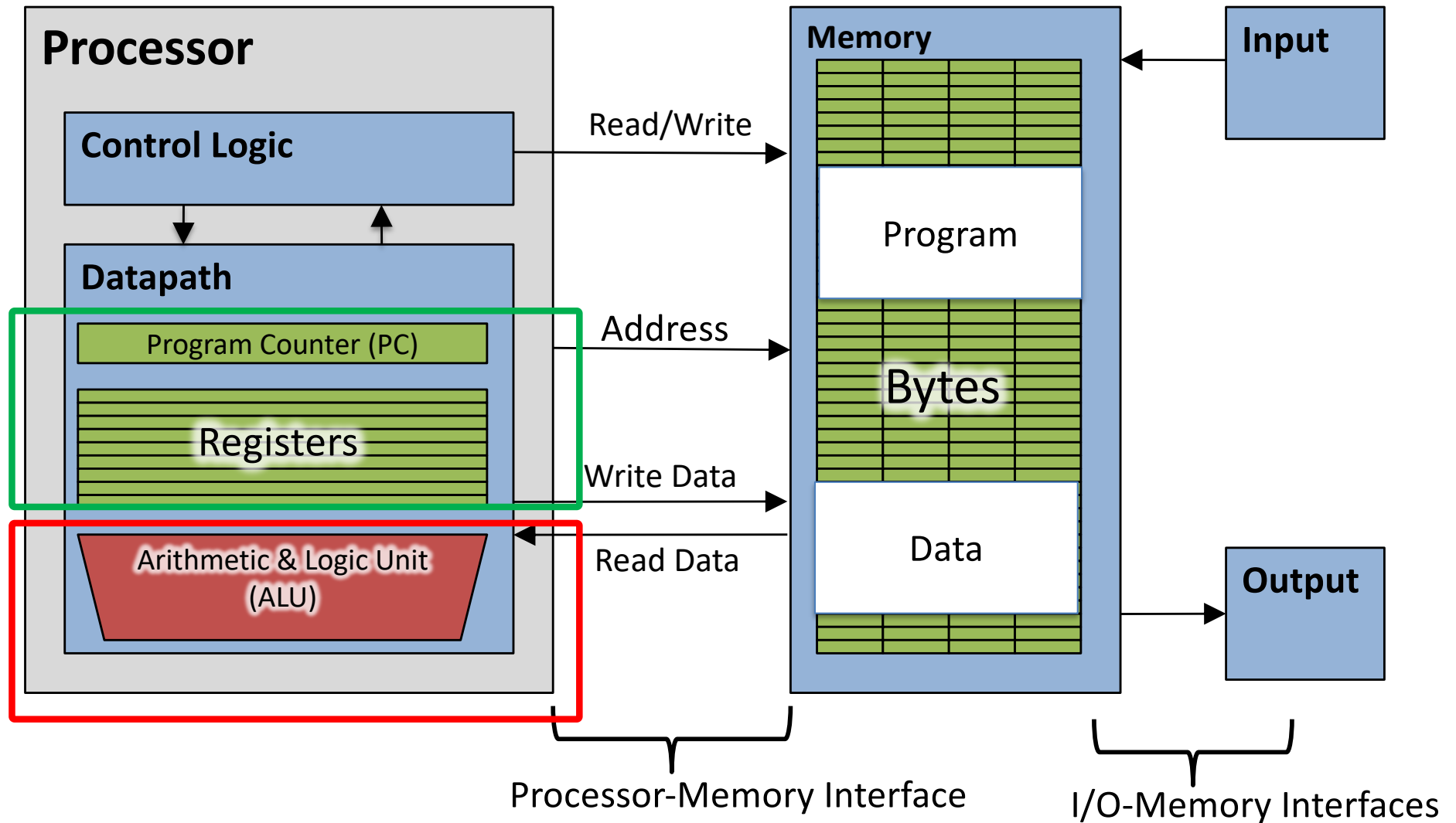
add x6, x10, x6 (add rd, rs1, rs2)



Appendix A: The Basics of Logic Design

- **Lecture 12**
 - **A.1 Introduction**
 - **A.2 Gates, Truth Tables, and Logic Equation**
- **Lecture 13**
 - **A.3 Combinational Logic**
 - ~~A.4 Using a Hardware Description Language~~
- **Lab 7**
- **Lecture 14**
 - **A.5 Constructing a Basic Arithmetic Logic Unit**
 - ~~A.6 Faster Addition: Carry Lookahead~~
- **Lecture 15**
 - **A.7 Clocks**
 - **A.8 Memory Elements: Flip-Flops, Latches, and Registers**
- **Lab 8**
- **Lecture 16**
 - **A.9 Memory Elements: SRAMs and DRAMs**
- **Lab 9**
- ~~Lecture 17~~
 - ~~A.10 Finite State Machines~~
 - ~~A.11 Timing Methodologies~~
 - ~~A.12. Field Programmable Devices~~
 - **A.13 Concluding Remarks**
 - ~~A.14 Exercises~~
- **Lab 10**

Components of a Computer

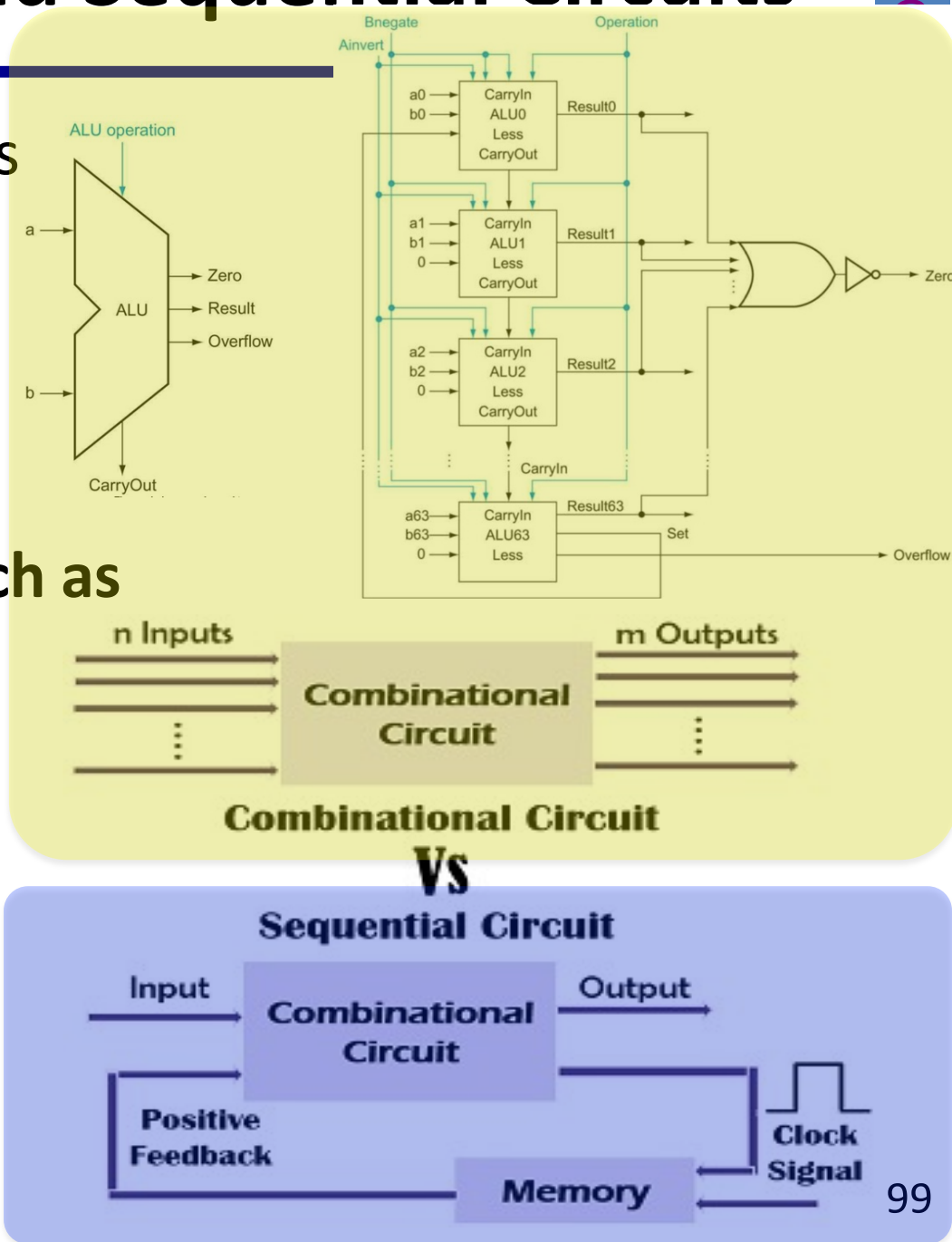


Appendix A: logic design
Chapter 4: CPU design

Chapter 5: Memory design

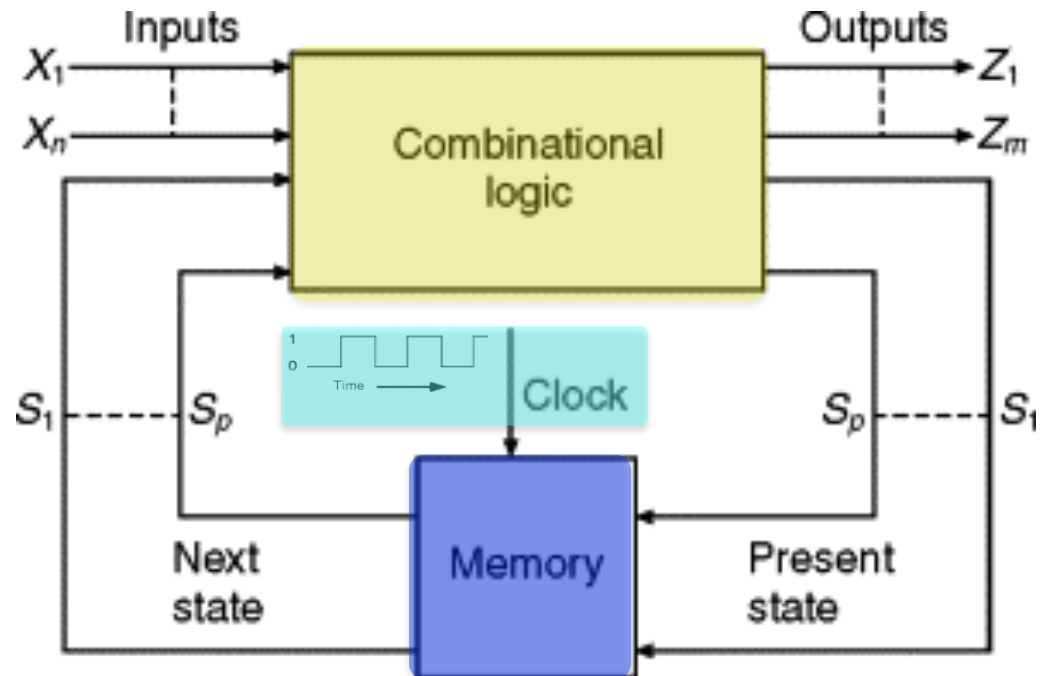
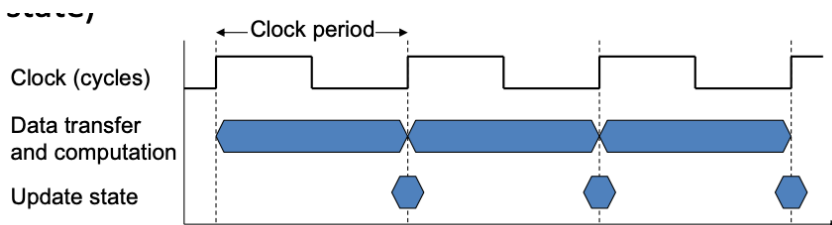
Combinational and Sequential Circuits

- Combinational circuit, such as mux, decoder, ALU
 - Operate on data
 - Output is a function of input
- State (sequential) circuit, such as register or memory
 - Store information
 - Outputs determined by previous and current values of inputs
 - e.g. previous inputs are stored



Sequential Circuits

- Mostly consists of combinational logic + Memory
- Memory is used to store state
 - Update memory according to both input and previous state
- Combinational logic to drive output
 - It uses both input and internal state to drive the output
- Controlled by clock
 - Update at specific time

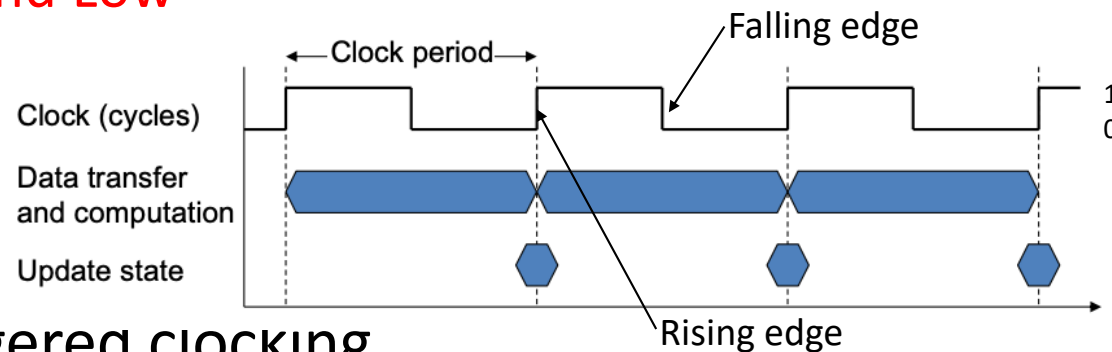


$$\begin{aligned} \text{CPU Time}(s) &= \# \text{ CPU Clock Cycles} \times \text{Clock Cycle Time } (s) \\ &= \frac{\# \text{ CPU Clock Cycles}}{\text{Clock Rate } (Hz)} \end{aligned}$$

Clocks



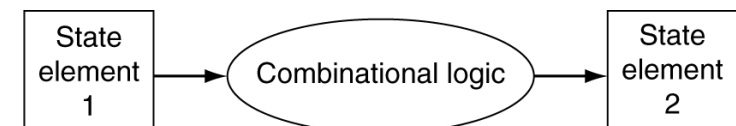
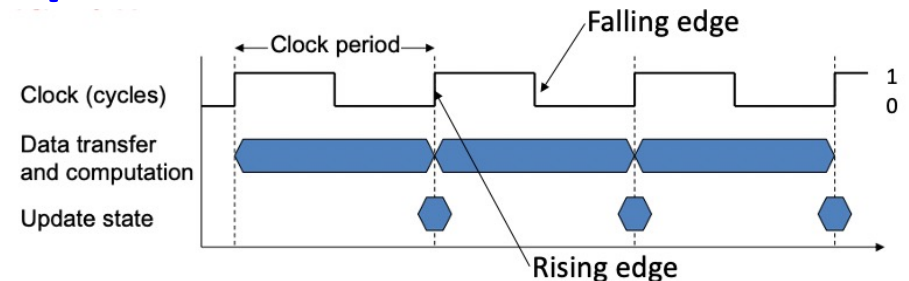
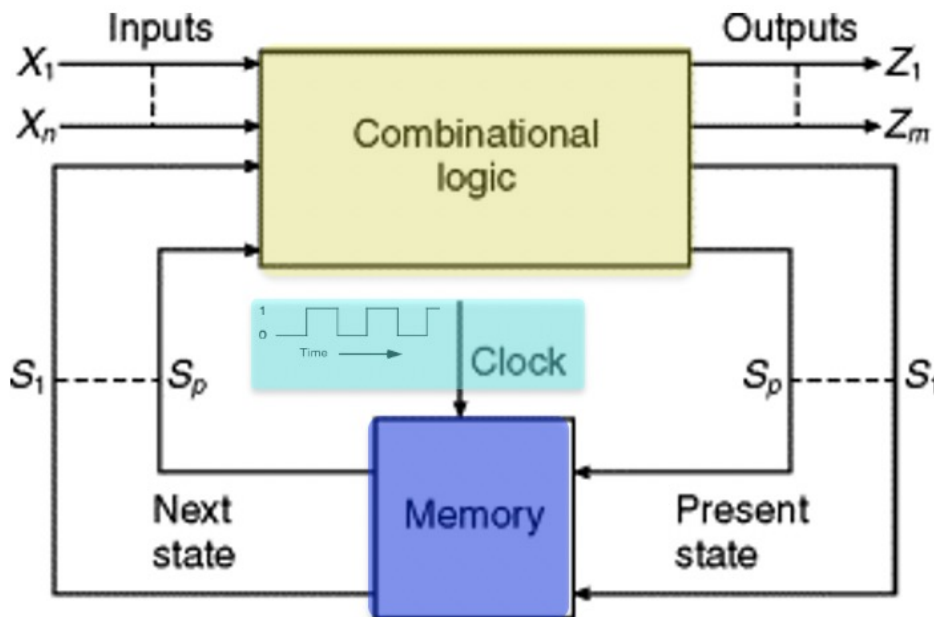
- Are needed in sequential logic to decide **when** an element that contains state (memory) should be updated (written or stored).
 - It is a real clock, but in different form
 - E.g. update the memory every 125 ns, not anytime one wants. But since it is very fast, not a big deal.
 - Cycle time, or clock period (inverse of clock frequency)
 - High and Low



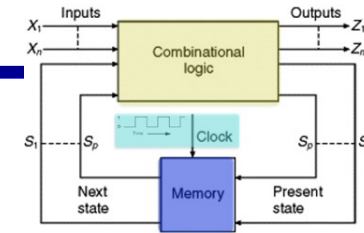
- Edge-triggered clocking
 - State change ($0 \rightarrow 1$ or $1 \rightarrow 0$) on a clock edge
 - Rising edge and falling edge
 - Active edge causes stage change
 - Could be just rising or falling, or both
 - State change in the rising edge in the above figure

Clocking Methodology

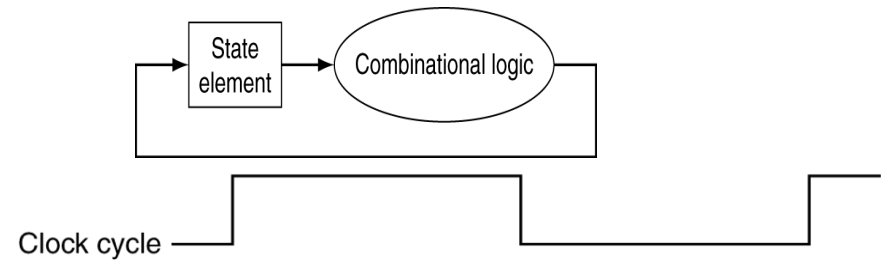
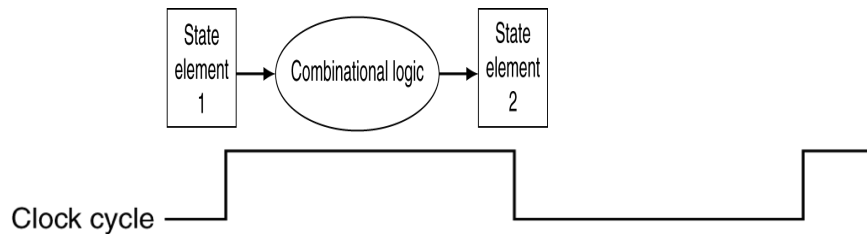
- Synchronous systems
 - Clock and input **MUST** be synchronized to make sure update is stabilized.
- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Input from state elements (memory), output to state element
 - **Longest delay determines clock period**



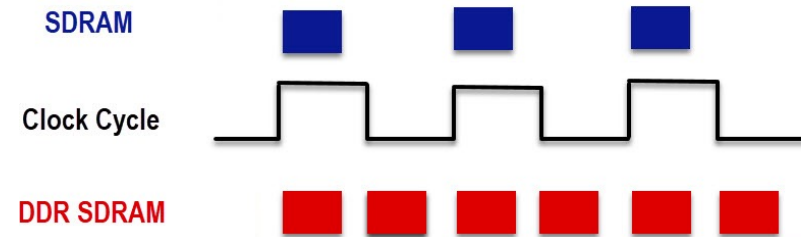
Read and Write in the Same Cycle



- Both rising and falling edge are active
 - Update twice per cycle



- E.g. Double data rate (DDR) memory



- Register files work in this way as well
 - Read and write to a register file in the same cycle
 - Read and write a register (**x6**) in the same cycle
 - Write to a register (**x6**) and then read it (**x6**) in the same cycle for two instructions

add x6, x4, x6

add x7, x6, x8

Memory Element

- ALU is used for doing computation

- Memory: to store information

- State: information at a particular time

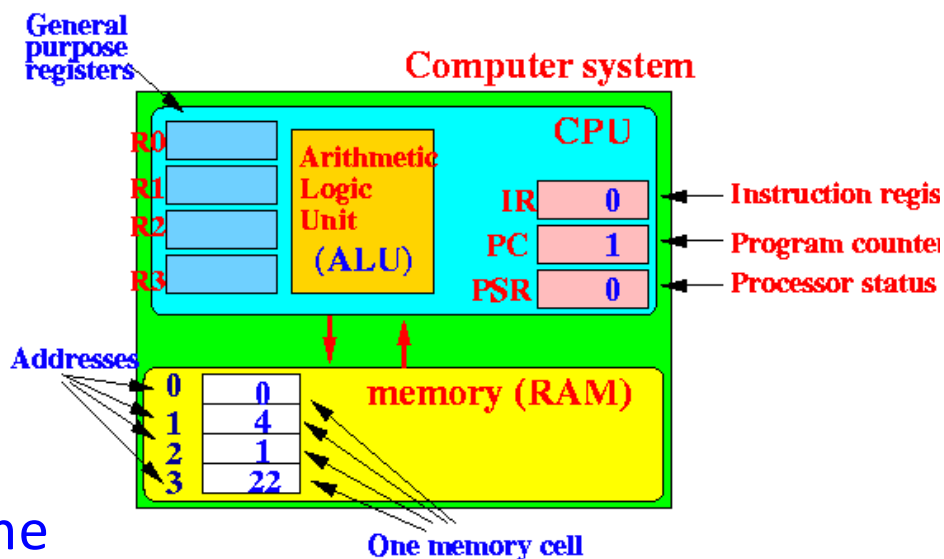
- Registers, cache and main memory are all “memory”

- Different type of technologies, e.g. SRAM and DRAM, detailed in Chapter 5

- Memory element in circuit

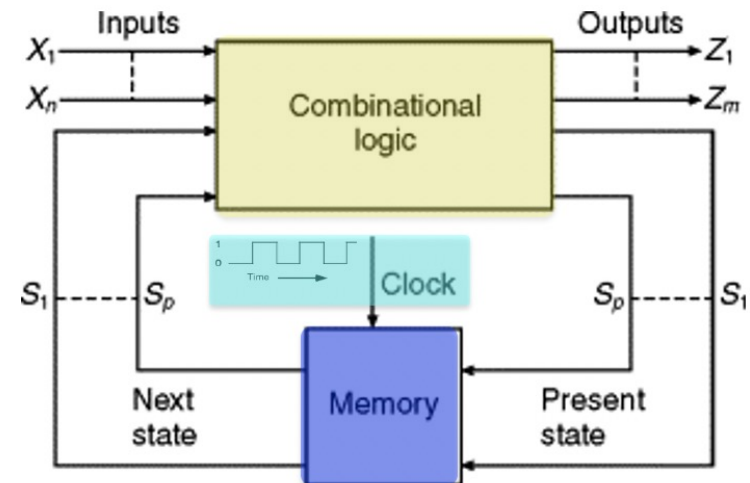
- The output from any memory element depends both on the inputs and on the value that has been stored inside the memory element.

- All logic blocks containing a memory element contain state are sequential, e.g. registers, cache and main memory



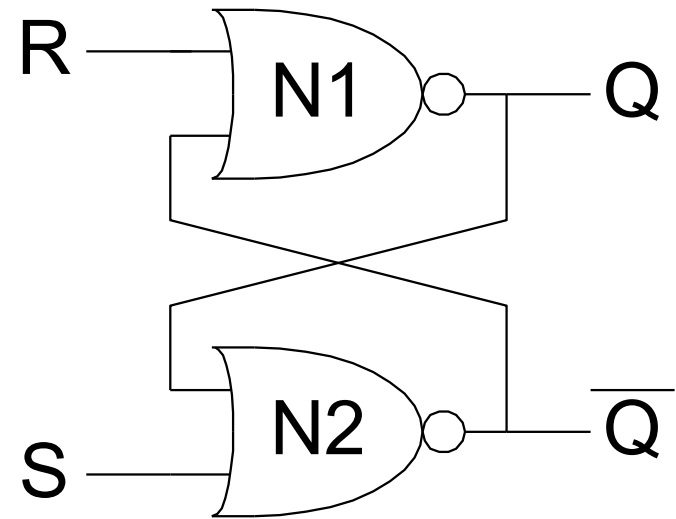
Some Terms about Memory

- Outputs of sequential logic depend on current *and* prior input values – it has **memory**.
- Some definitions:
 - **State**: all the information about a circuit necessary to explain its future behavior
 - **Latches and flip-flops: state elements (circuits) that store one bit of state**
 - **Sequential circuits are synchronous combinational logic followed by a bank of flip-flops (memory) controlled by clock**
- **We start studying how to design logic to store a single-bit**
 - **Then extend to create logics for storing multiple bits**
 - **1-bit ALU → 32/64-bit ALU**



SR (Set/Reset) Latch

- SR Latch
 - Cross-coupled structure

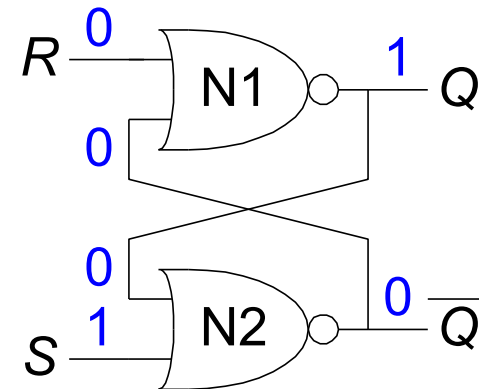


- Consider the four possible cases of the 2 inputs:
 - $S = 1, R = 0$
 - $S = 0, R = 1$
 - $S = 0, R = 0$
 - $S = 1, R = 1$

SR Latch Analysis

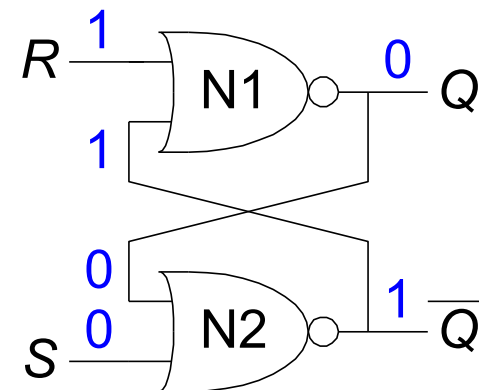
– $S = 1, R = 0$:

then $Q = 1$ and $\bar{Q} = 0$



– $S = 0, R = 1$:

then $Q = 0$ and $\bar{Q} = 1$

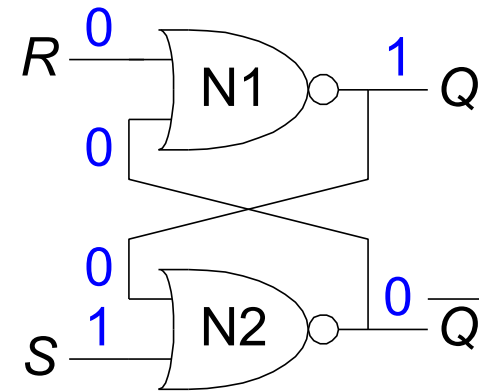


SR Latch Analysis

– $S = 1, R = 0$:

then $Q = 1$ and $\bar{Q} = 0$

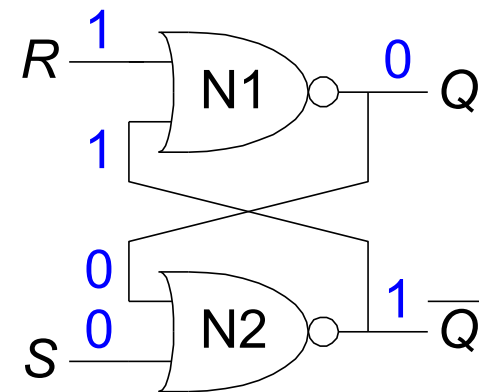
Set the output to 1



– $S = 0, R = 1$:

then $Q = 0$ and $\bar{Q} = 1$

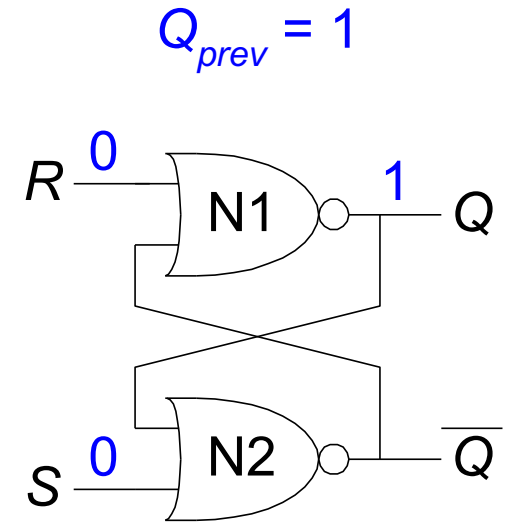
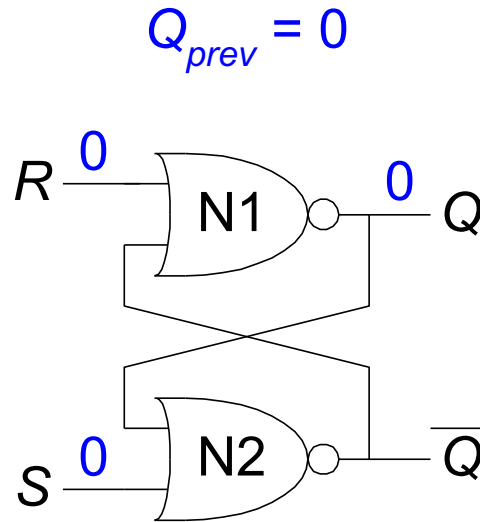
Reset the output to 0



SR Latch Analysis

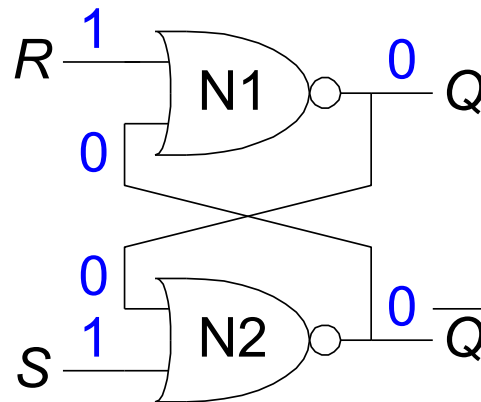
– $S = 0, R = 0$:

then $Q = Q_{prev}$



– $S = 1, R = 1$:

then $Q = 0, \bar{Q} = 0$

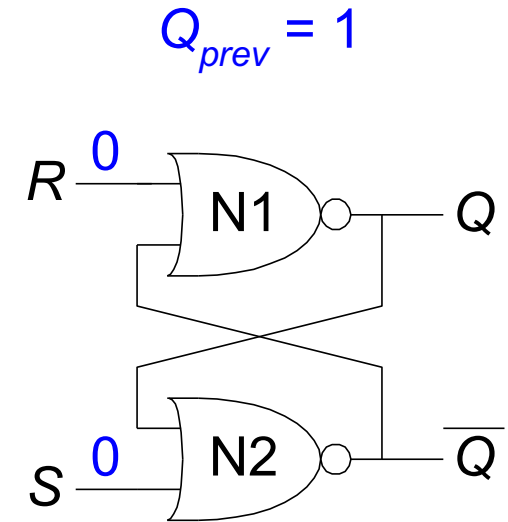
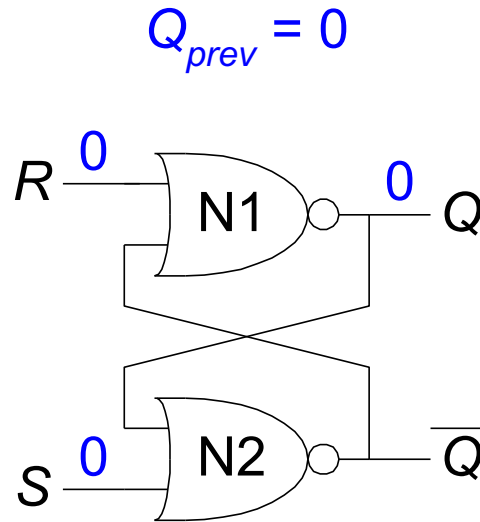


SR Latch Analysis

– $S = 0, R = 0$:

then $Q = Q_{prev}$

Memory!

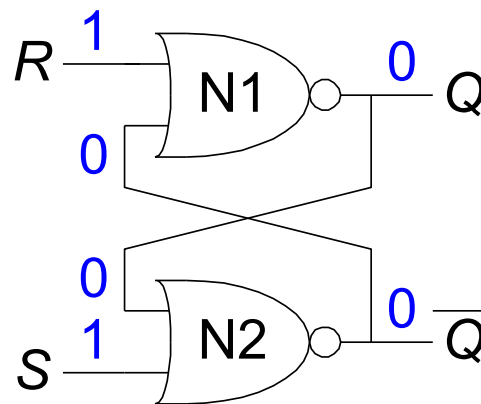


– $S = 1, R = 1$:

then $Q = 0, \bar{Q} = 0$

Invalid State

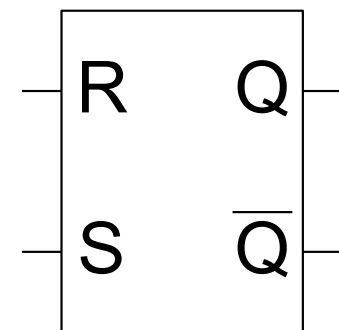
$Q \neq \text{NOT } \bar{Q}$



SR Latch Symbol

- SR stands for Set/Reset Latch
 - Stores one bit of state (Q)
- Control what value is being stored with S , R inputs
 - **Set:** Make the output 1
($S = 1, R = 0, Q = 1$)
 - **Reset:** Make the output 0
($S = 0, R = 1, Q = 0$)
- **Must do something to avoid invalid state (when $S = R = 1$)**

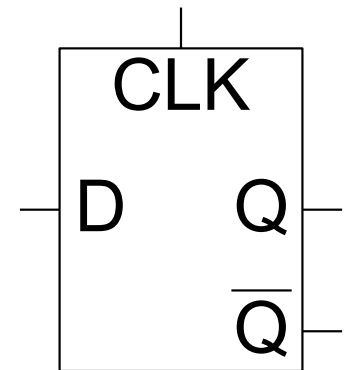
SR Latch
Symbol



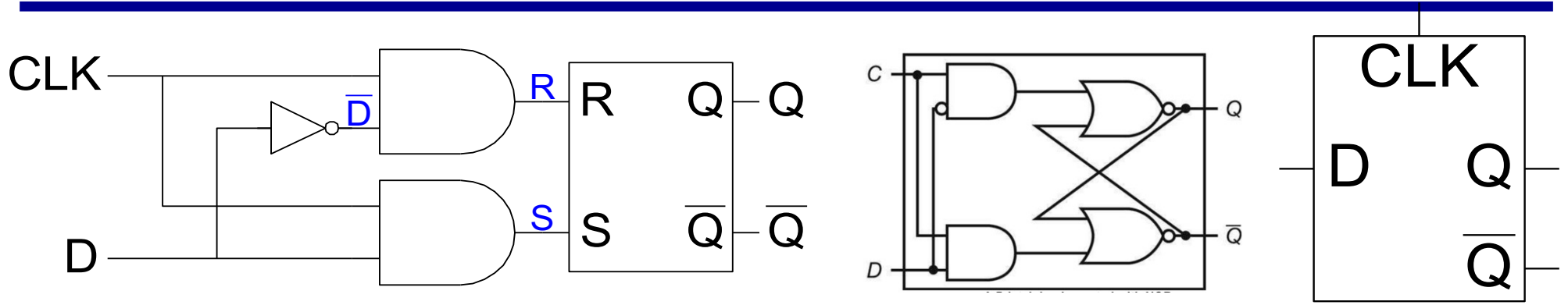
D Latch

- Two inputs: CLK , D
 - CLK : controls *when* the output changes
 - D (the data input): controls *what* the output changes to
- Function
 - When $CLK = 1$, ($0 \rightarrow 1$, rising edge)
 D passes through to Q (*transparent*)
 - When $CLK = 0$,
 Q holds its previous value (*opaque*)
- Avoids invalid case when
 $Q \neq \text{NOT } \bar{Q}$

D Latch
Symbol

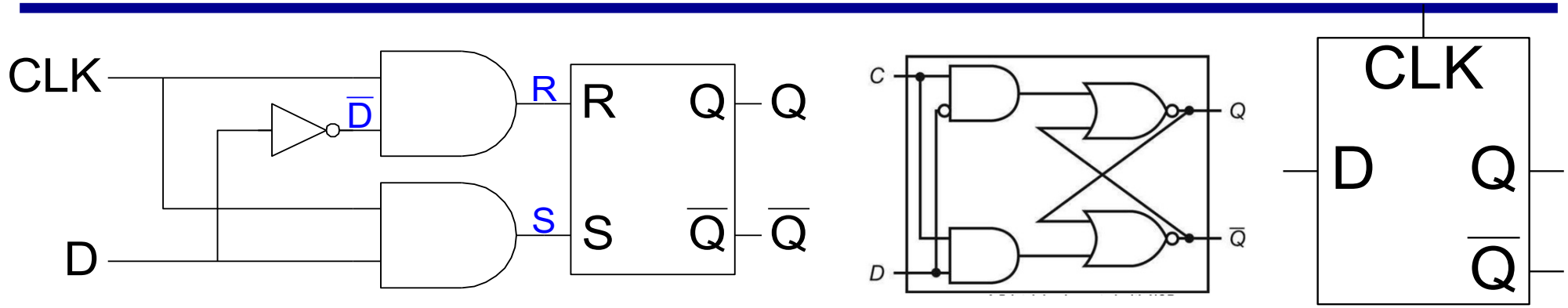


D Latch Internal Circuit



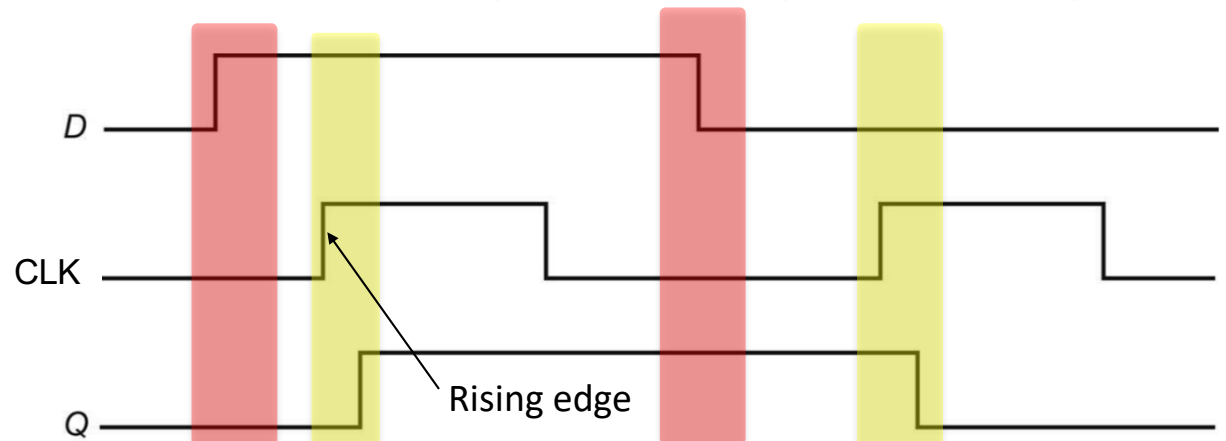
<i>CLK</i>	<i>D</i>	\overline{D}	<i>S</i>	<i>R</i>	<i>Q</i>	\overline{Q}
0	X					
1	0					
1	1					

D Latch Internal Circuit



CLK	D	\overline{D}	S	R	Q	\overline{Q}
0	X	\overline{X}	0	0	Q_{prev}	\overline{Q}_{prev}
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Q does not change even D is changed since CLK is not on the rising edge (0→1).

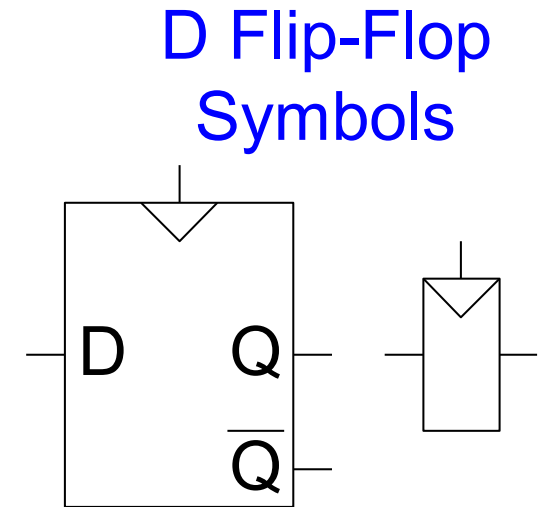


When the clock, CLK, is asserted (rising edge, i.e. 0→1), the latch is open and the Q output immediately assumes the value of the D input.

FIGURE A.8.3 Operation of a D latch, assuming the output is initially deasserted.

D Flip-Flop

- **Inputs:** CLK , D
- **Function**
 - Samples D on falling edge of CLK
 - **When CLK falls from 1 to 0, D passes through to Q**
 - Otherwise, Q holds its previous value
 - **Q changes only on falling edge of CLK**
- Called *edge-triggered*
- Activated on the clock edge



D Flip-Flop: Falling Edge Triggered

- Two back-to-back latches (L1 and L2) controlled by complementary clocks

- When $CLK = 1$

- L1 is transparent
- L2 is opaque
- D passes through to N1

- When $CLK = 0$

- L2 is transparent
- L1 is opaque
- N1 passes through to Q

- Thus, on the falling edge of the clock (when CLK falls from $1 \rightarrow 0$)

- D passes through to Q

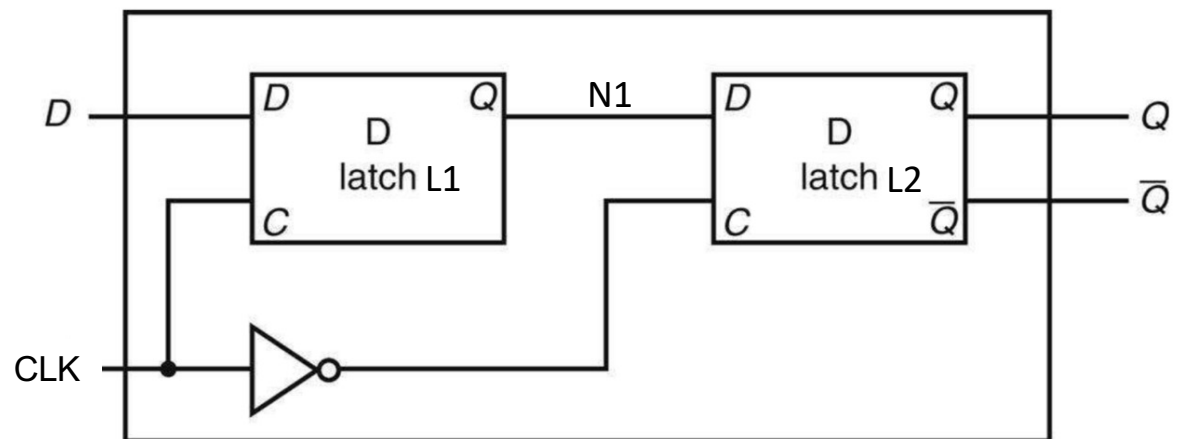


FIGURE A.8.4 A D flip-flop with a falling-edge trigger.

D Flip-Flop: Falling Edge Triggered

- Two back-to-back latches (L1 and L2) controlled by complementary clocks

- When $CLK = 1$

- L1 is transparent
- L2 is opaque
- D passes through to N1

- When $CLK = 0$

- L2 is transparent
- L1 is opaque
- N1 passes through to Q

- Thus, on the falling edge of the clock (when CLK falls from 0 \rightarrow 1)

- D passes through to Q

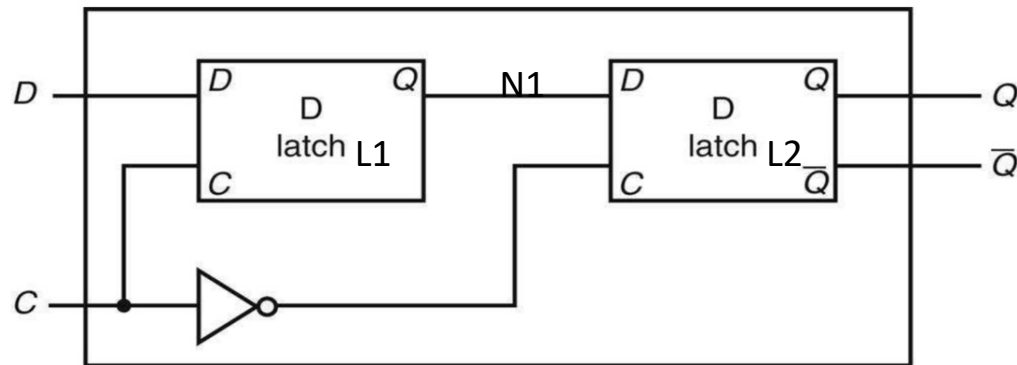


FIGURE A.8.4 A D flip-flop with a falling-edge trigger.

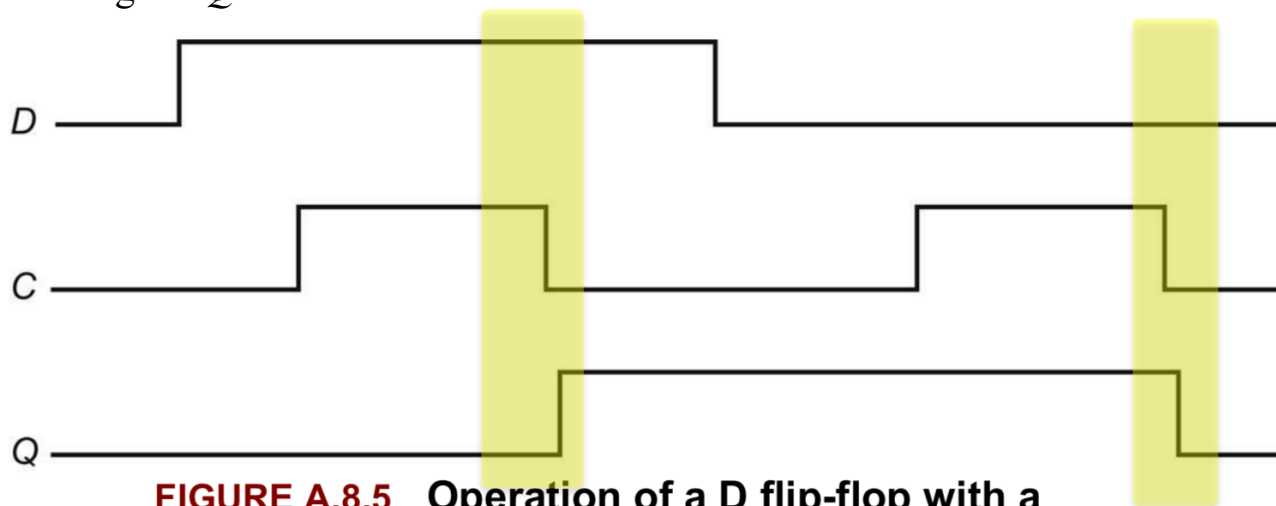
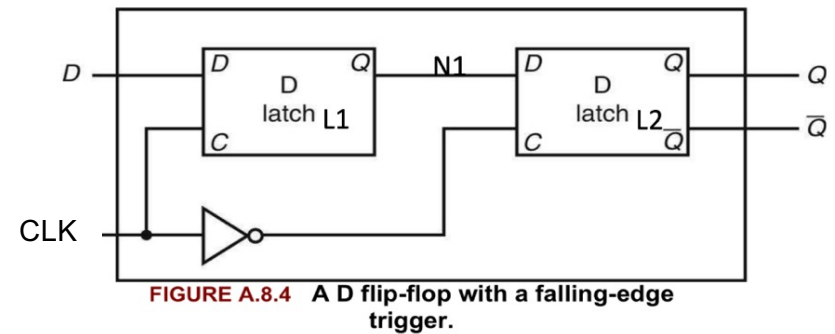
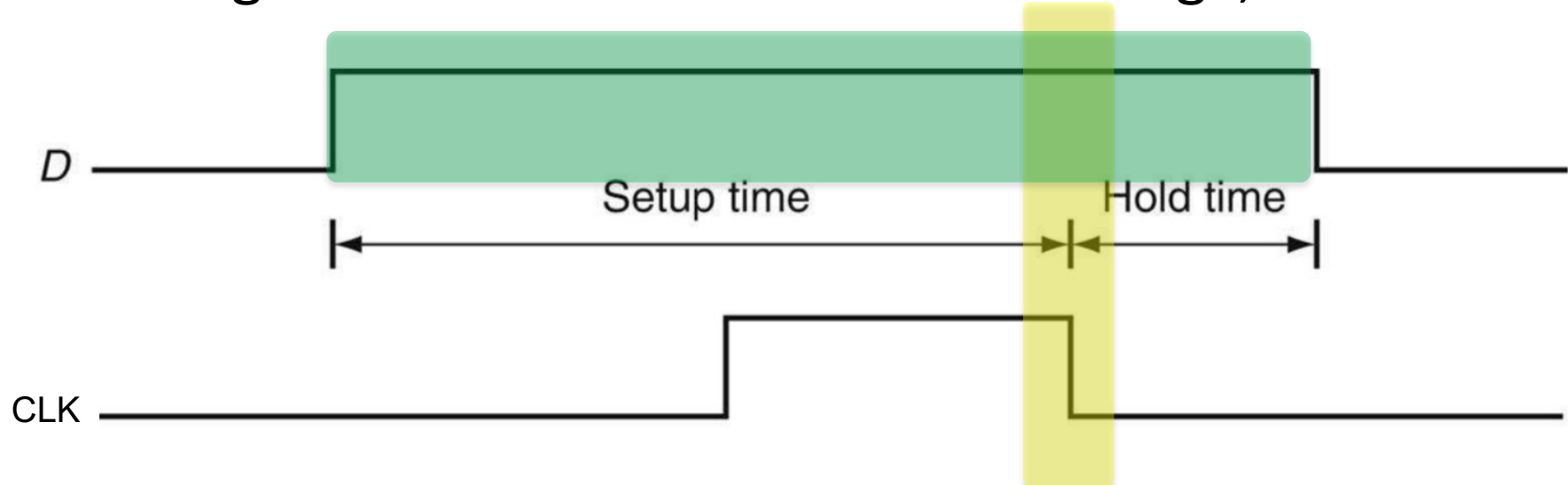


FIGURE A.8.5 Operation of a D flip-flop with a falling-edge trigger, assuming the output is initially deasserted.

Timing



- For a falling-edge triggered D flip-flop
 - The input must be stable for a period of time before the clock edge, as well as after the clock edge, for the latches to sample
- **Setup time:** the minimum time that the input must be valid before the clock edge;
- **Hold time:** the minimum time during which it must be valid after the clock edge
- Thus the inputs to any flip-flop (or anything built using flip-flops) must be valid during a window that begins at time t_{setup} before the clock edge and ends at t_{hold} after the clock edge,



D Flip-Flop: Rising Edge Triggered

- Two back-to-back latches (L1 and L2) controlled by complementary clocks

- When $CLK = 0$

- L1 is transparent
- L2 is opaque
- D passes through to N1

- When $CLK = 1$

- L2 is transparent
- L1 is opaque
- N1 passes through to Q

- Thus, on the rising edge of the clock (when CLK rises from $0 \rightarrow 1$)

- D passes through to Q

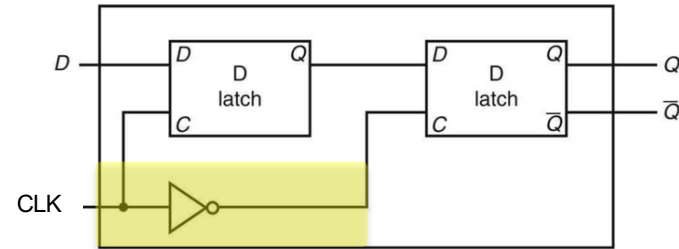
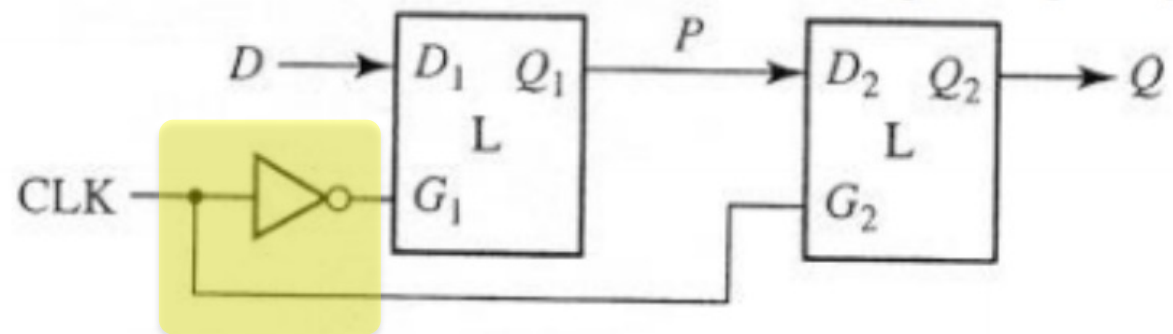


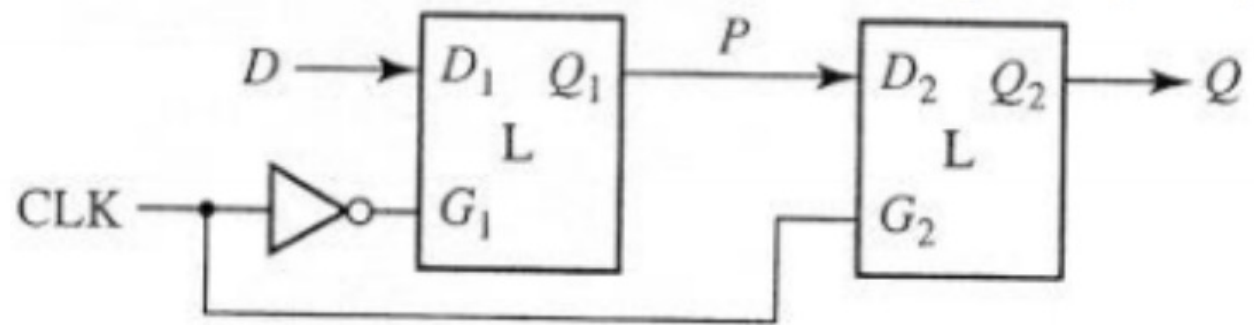
FIGURE A.8.4 A D flip-flop with a falling-edge trigger.



D Flip-Flop, Rising Edge Triggered

- Verilog description:

```
module DFF(clock, D, Q, Qbar);  
  input clock, D;  
  output reg Q;  
  output Qbar;  
  assign Qbar = ~ Q;  
  always @(posedge clock)  
    Q = D;  
endmodule
```



- **always@(posedge clock):** at the positive/rising edge of the clock
- **always@(negedge clock):** at the negative/falling edge of the clock
- They are used to describe sequential Logic, or Registers.

Registers in a Processors

- General purpose registers

- 32 32-/64-bit registers

- Instruction register

- Store the current instruction word

- Program counter (PC)

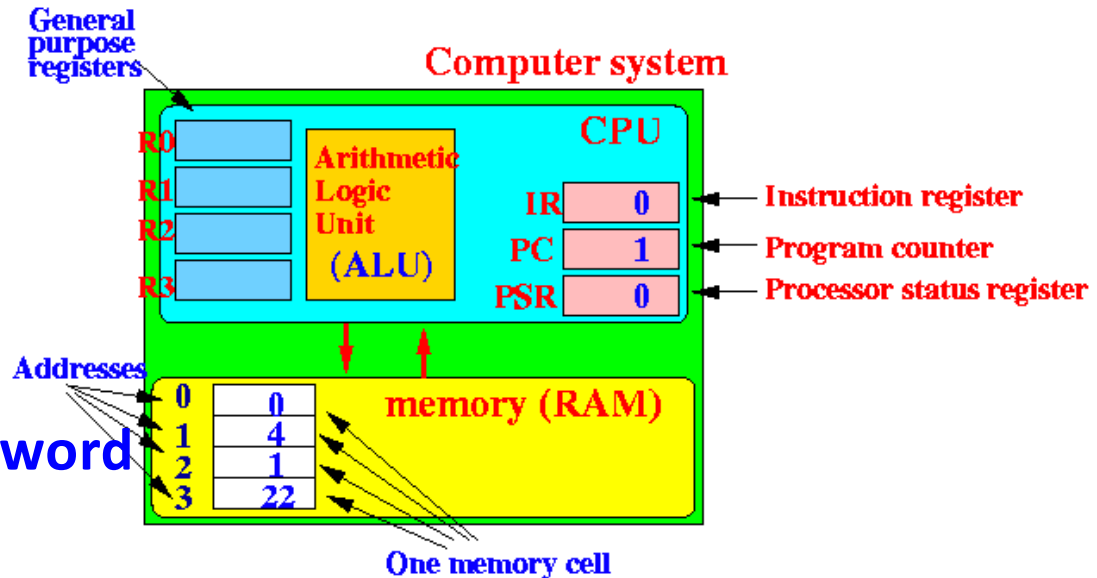
- Store the address of the current instruction

- Status registers

- Others: page table register, etc.

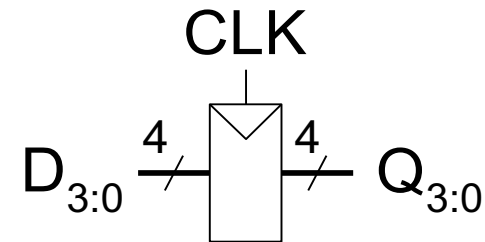
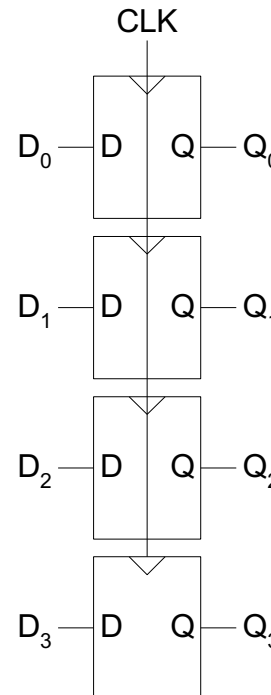
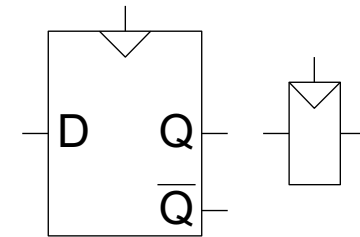
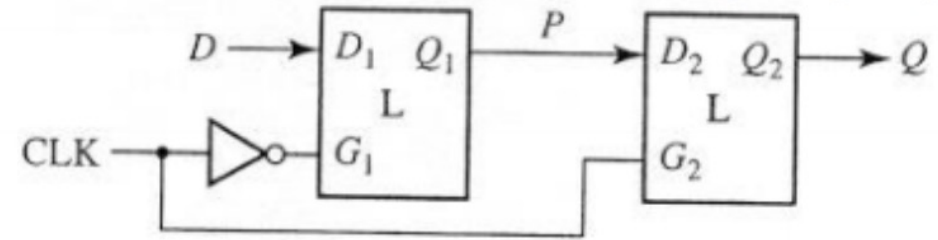
- A 32- or 64-bit register: a kind of memory

- Created with 32 or 64 D-flip-flops and combinational logics to read and write
- Can be read from and written to



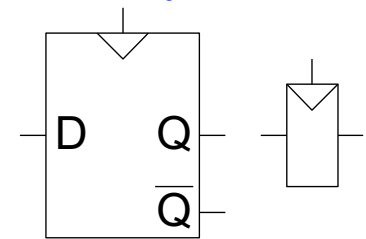
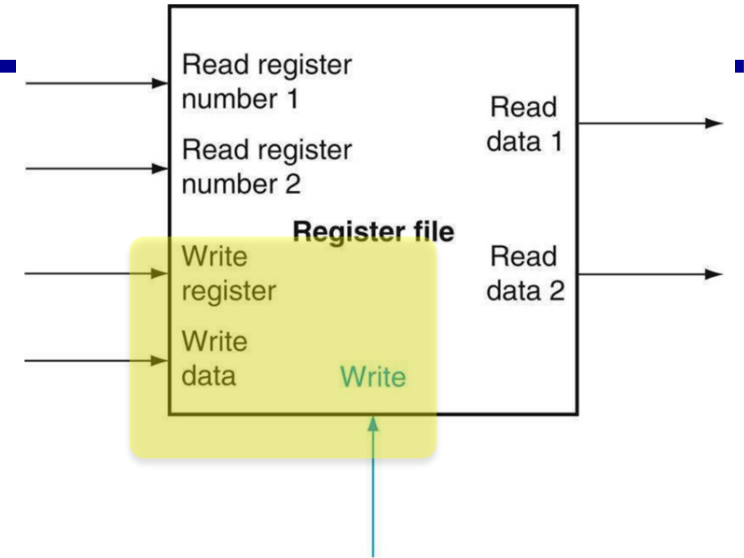
1-Bit and 4-bit Register

- A D Flip-flop is a 1-bit register
 - D is write input port
 - Q is read output port
 - Clk is the input control to write
- A 4-bit register
 - Needs 4 D flip-flops
- Same for a 32- or 64-bit register



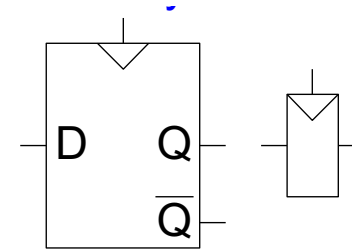
A Register File

- Has multiple registers
 - E.g. 32 32-bit registers
 - Numbered from 0 to 31
 - Needs 5 bit to address each
- **Read: (does not change state)**
 - Input: register number
 - Output: data of the register
- **We want to read (at least) two registers the same time**
 - add x7, x6, x5 (two register reads and one register write)
 - Two read ports
- **Write: (change state)**
 - 3 Inputs:
 - Register number
 - Data to be written to the register
 - Write signal (1-bit): to tell register that this is a write (clock signal)
 - No output
- A D flip-flops:
 - Read anytime
 - Write on the rising edge of the clock (write signal)



Design a Register File with 32 32-bit Registers from Digital

- Design a single 32-bit register
 - A 32 bitwidth D flip-flop
 - D is “write data” input
 - Q is “read data” output
 - Clk is the “Write” input, or called WE (Write-Enable)



Sequential Elements

Test Bench

Misc

PROPERTIES

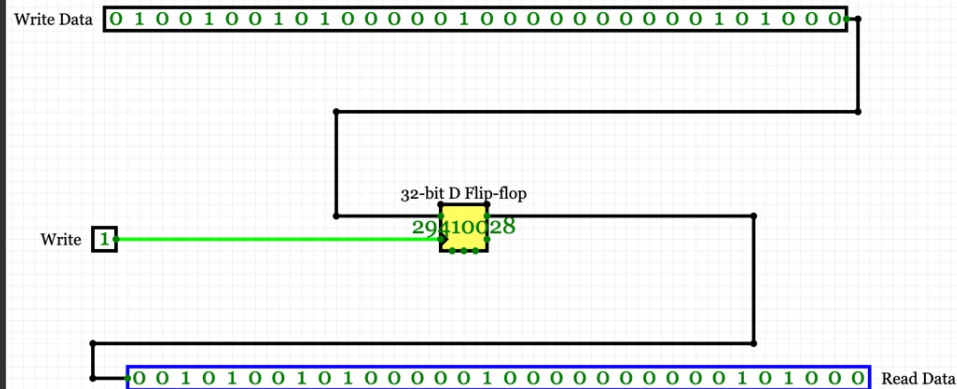
DFLIPFLOP

BitWidth: 32

Delay: 100

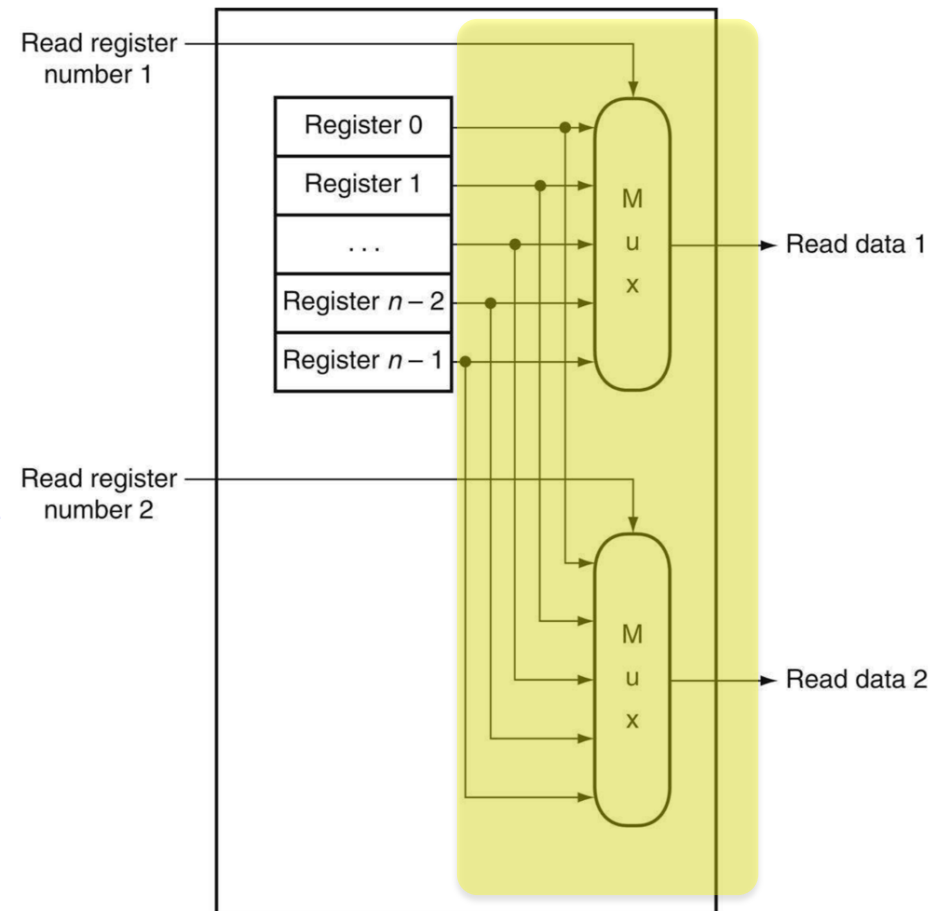
Label: 32-bit D Flip-flop

Label Direction: UP



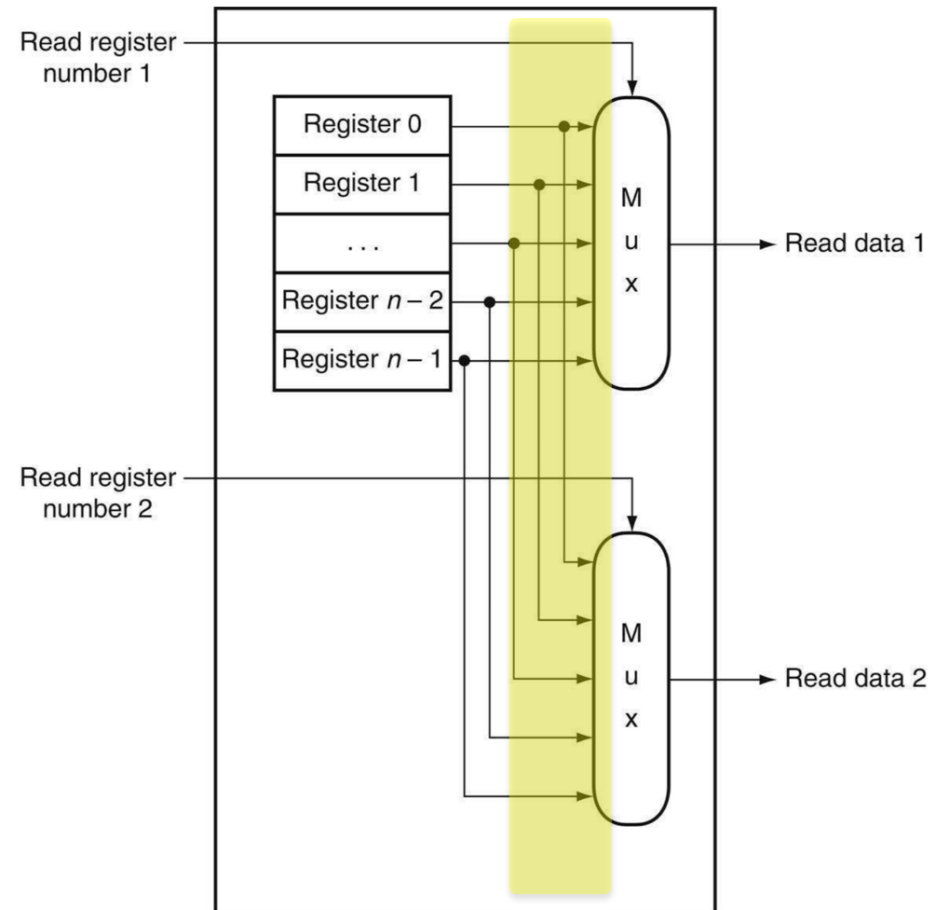
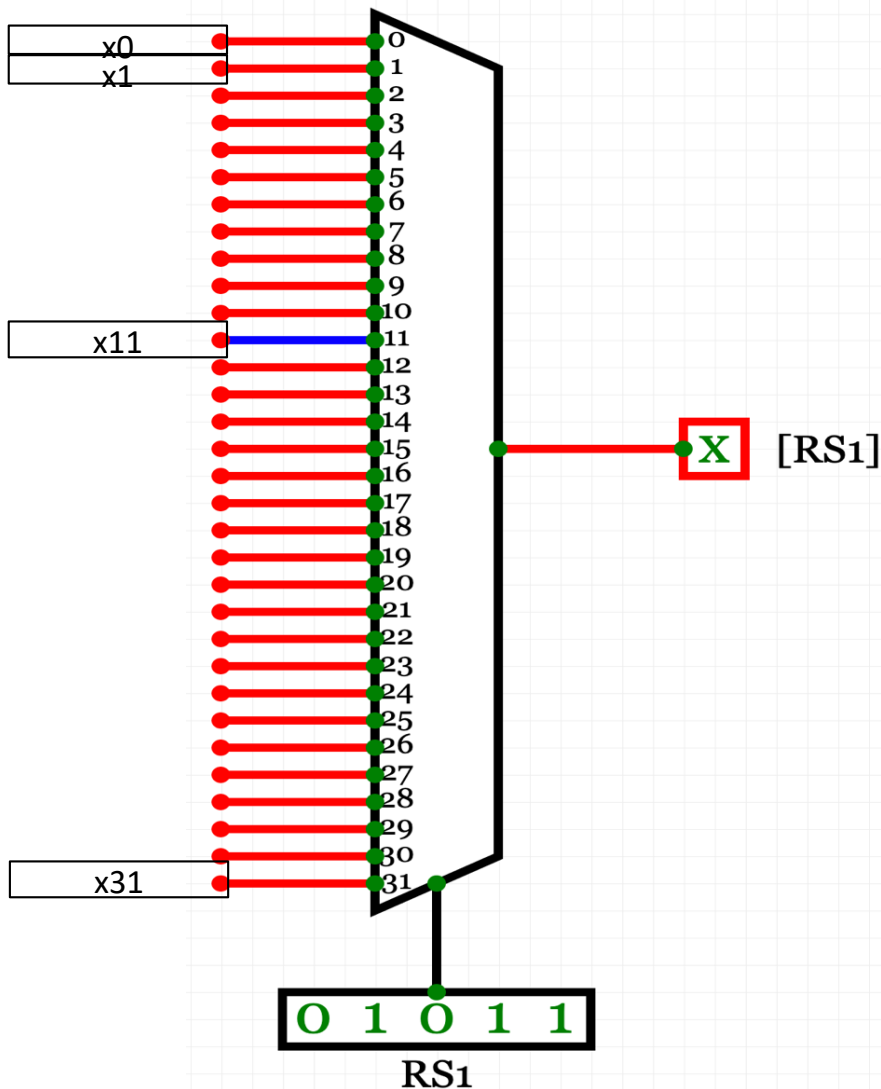
Design the Read Port of the Register File

- Stack up 32 32-bit registers and label them from x0 to x31
 - Leave some space between registers for wires
- Each Read Port:
 - Since the output (Q of D flip-flop) of each register is always available, to read from one register is just to select which output to become the output of the register file.
 - **Select one from 32: use a Mux**
 - 5-bit selection (register number)
 - Two read ports since for most instructions, there are two source operands, e.g.
 - **add x7, x6, x5**
 - **sd x6, 0(x5)**
 - **beq x1, x2, label**



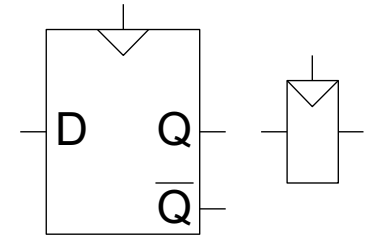
Use the 32-1 Mux You Designed Before

- Need two Mux's, one for each read port



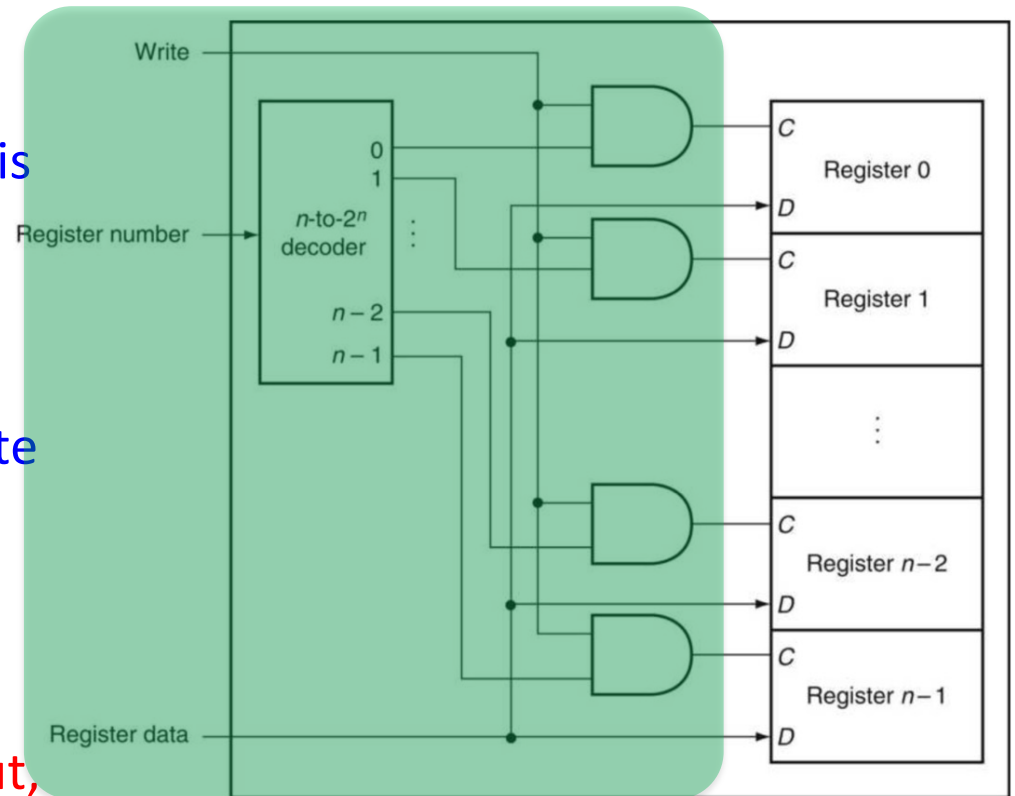
Design the Write Port of the Register File (1/2)

- Three inputs:
 - **Register number to be written to**
 - **Data to be written to the register: which is D**
 - **Write signal (1-bit): to tell that this is a write: which is clock**
- Design:
 - Write data sent to all registers (write data wired to D of all).
 - Use writer register number to turn on the write signal (1-bit clock) of the target register
 - A decoder to set one of 32 output,
 - and then AND with the write signal
 - and then send the output of AND to the clock input of each register



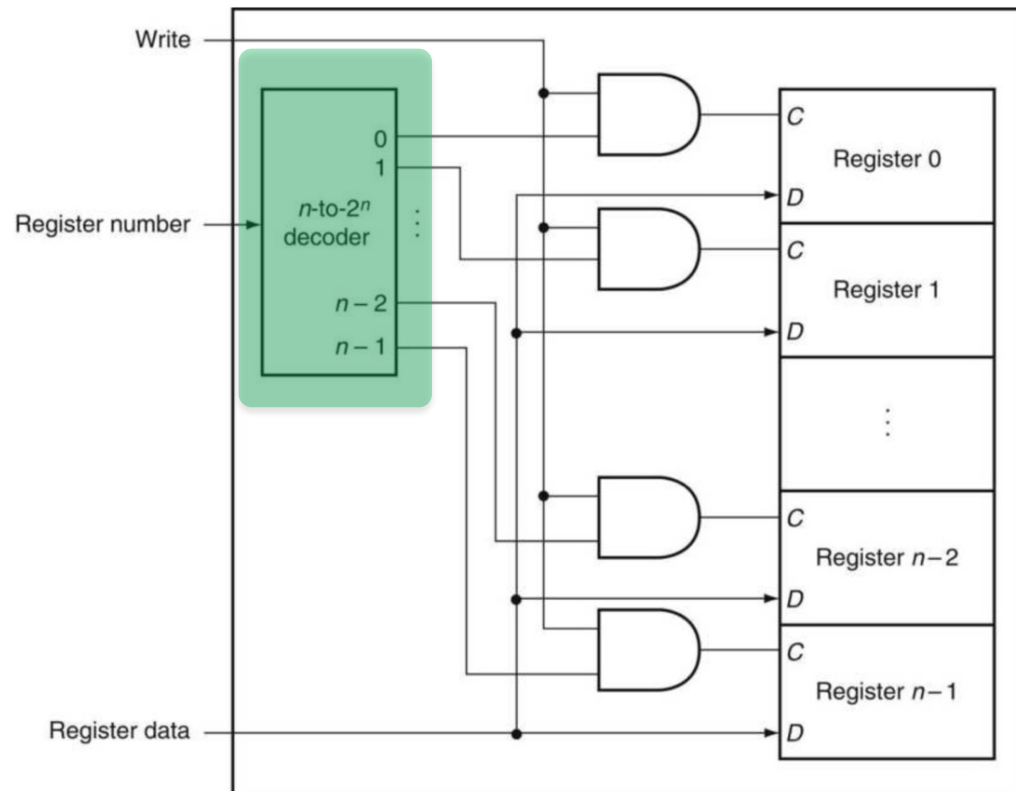
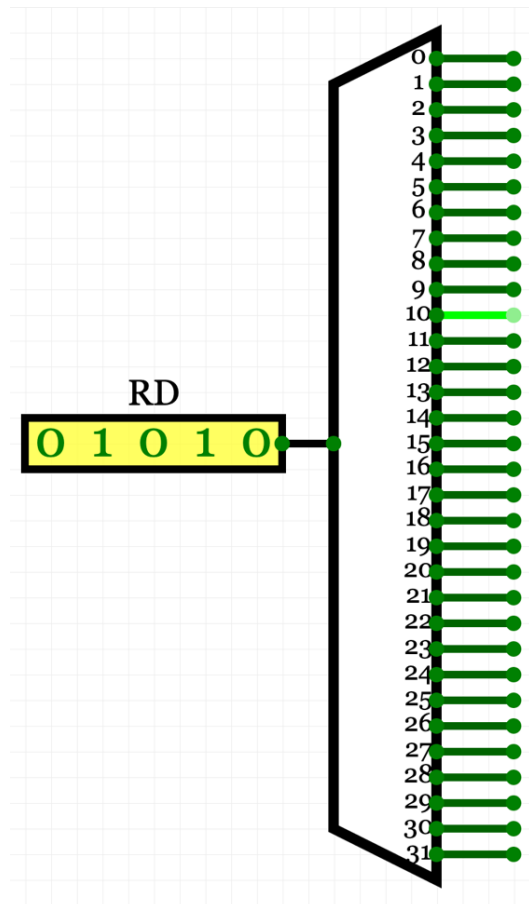
Design the Write Port of the Register File (2/2)

- Three inputs:
 - Write register number
 - Data to be written to the register: **which is D**
 - write signal (1-bit): to tell that this is a write: **which is clock**
- Design:
 - Write data sent to all registers (write data wired to D of all).
 - Use writer register number to turn on the write signal (1-bit clock) of the target register
 - A decoder to set one of 32 output,
 - and then AND with the write signal
 - and then send the output of AND to the clock input of each register

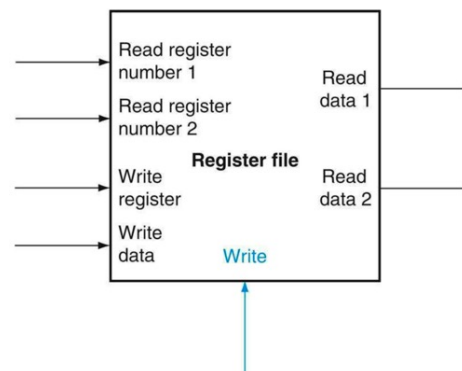
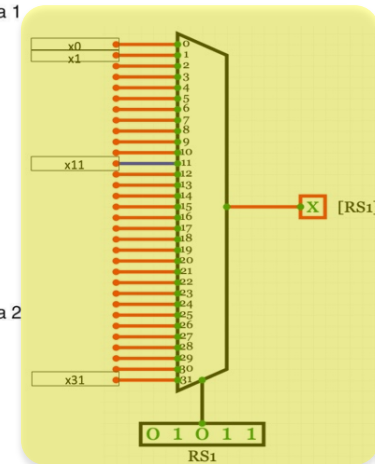
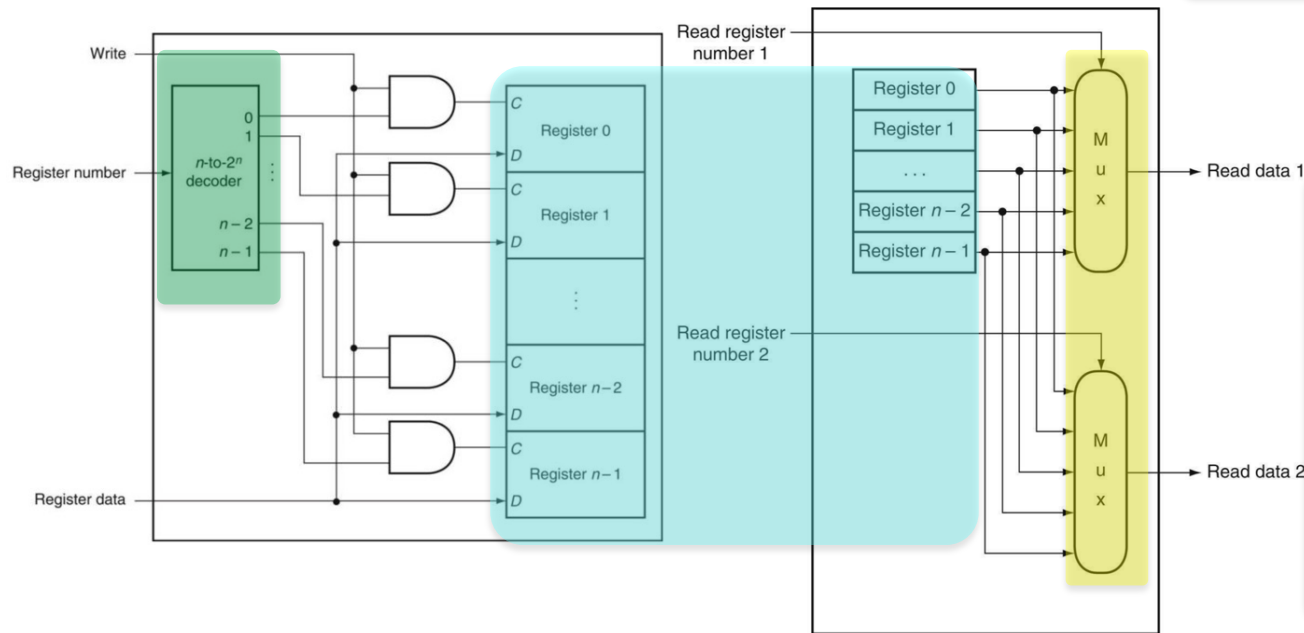
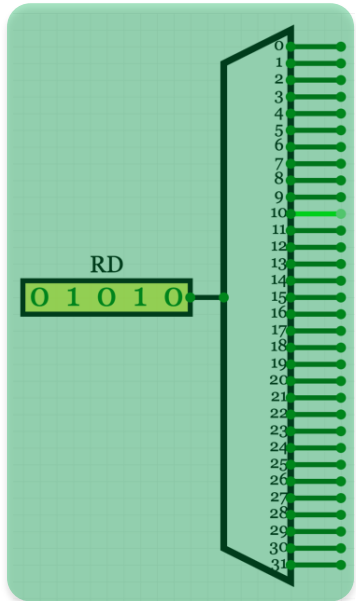
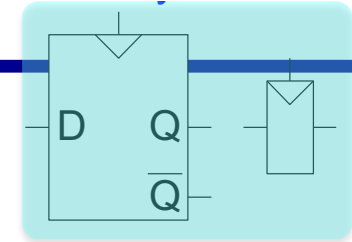


Use the 32-Output Decoder You Designed Before

- Add AND gate for each register for write

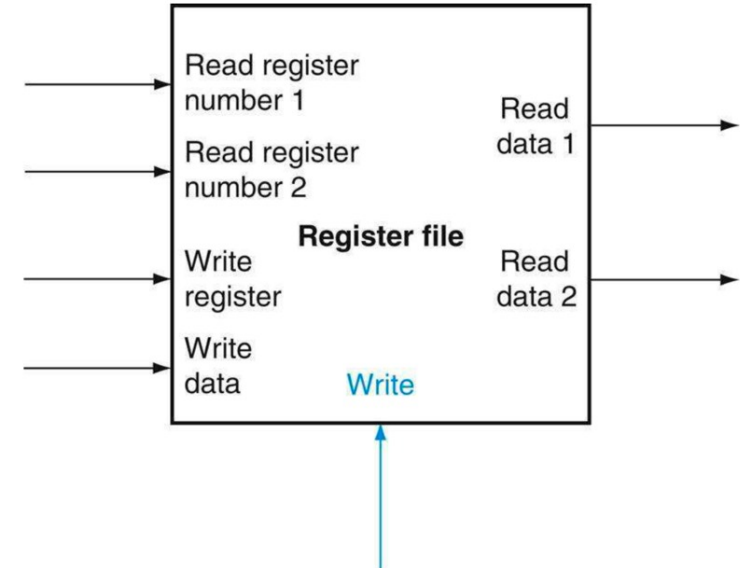


Lab 09: Design a Register File with 32 32-bit (16- or 8-bit) Registers (1/2)



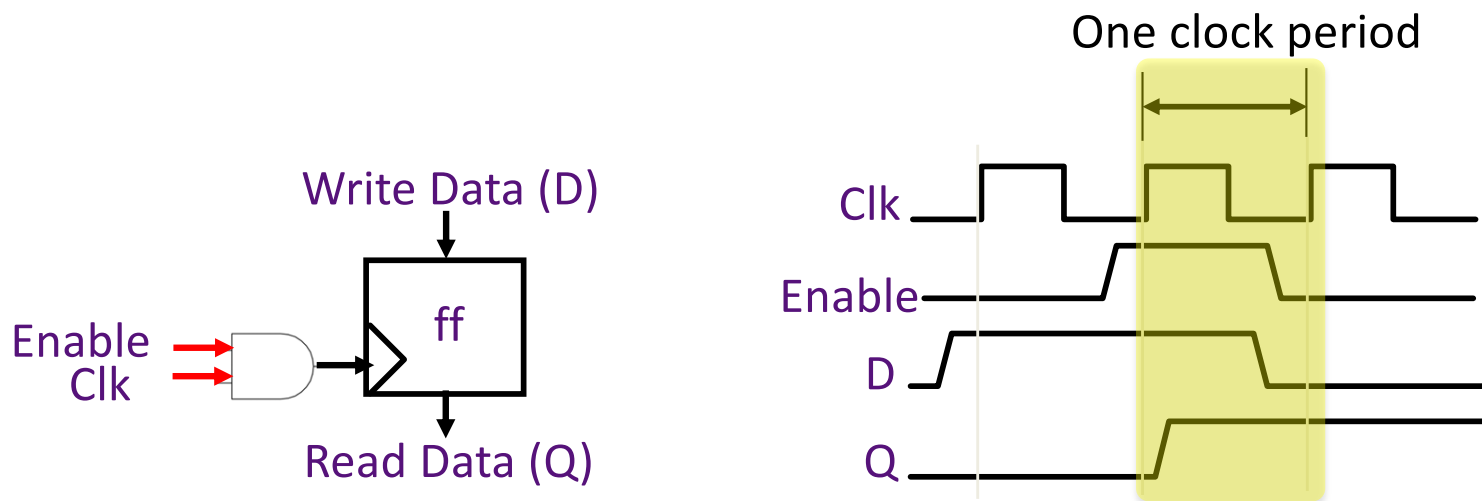
Lab 09: Design a Register File with 32 32-bit (16- or 8-bit) Registers (2/2)

1. Design 1 32/16/8-bit register
 - Use the system provided D flip-flop
 - You can design your own D flip-flop and use yours
2. Stack up 32 32/16/8-bit registers
 - Leave room between registers for wires
3. Design the two read ports
 - Use the Mux you designed before
4. Design the write inputs
 - Use the decoder you designed before
5. Label input/output correctly, wire neatly.



Rising Edge Triggered Register Can be Written and Read in the Same Clock Cycle

- D is written to D flip-flop (register) at the rising edge, and data is available in the same cycle for read

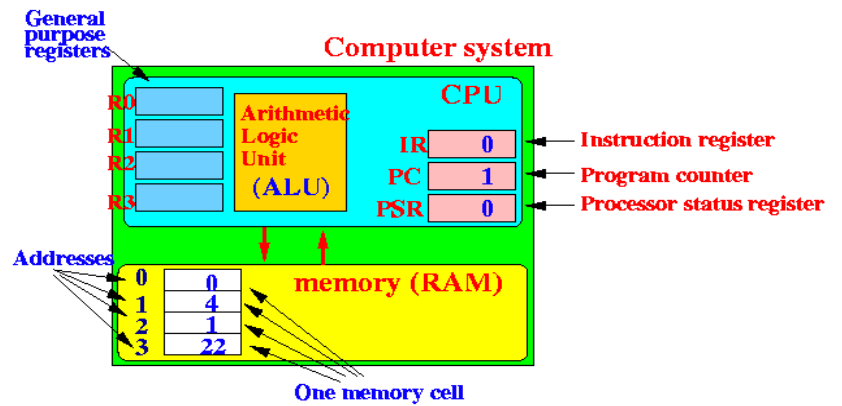


Appendix A: The Basics of Logic Design

- **Lecture 12**
 - **A.1 Introduction**
 - **A.2 Gates, Truth Tables, and Logic Equation**
- **Lecture 13**
 - **A.3 Combinational Logic**
 - ~~A.4 Using a Hardware Description Language~~
- **Lab 7**
- **Lecture 14**
 - **A.5 Constructing a Basic Arithmetic Logic Unit**
 - ~~A.6 Faster Addition: Carry Lookahead~~
- **Lecture 15**
 - **A.7 Clocks**
 - **A.8 Memory Elements: Flip-Flops, Latches, and Registers**
- **Lab 8**
- **Lecture 16**
 - **A.9 Memory Elements: SRAMs and DRAMs**
- **Lab 9**
- ~~Lecture 17~~
 - ~~A.10 Finite State Machines~~
 - ~~A.11 Timing Methodologies~~
 - ~~A.12. Field Programmable Devices~~
 - **A.13 Concluding Remarks**
 - ~~A.14 Exercises~~
- **Lab 10**

A Typical Single-Cycle Processor

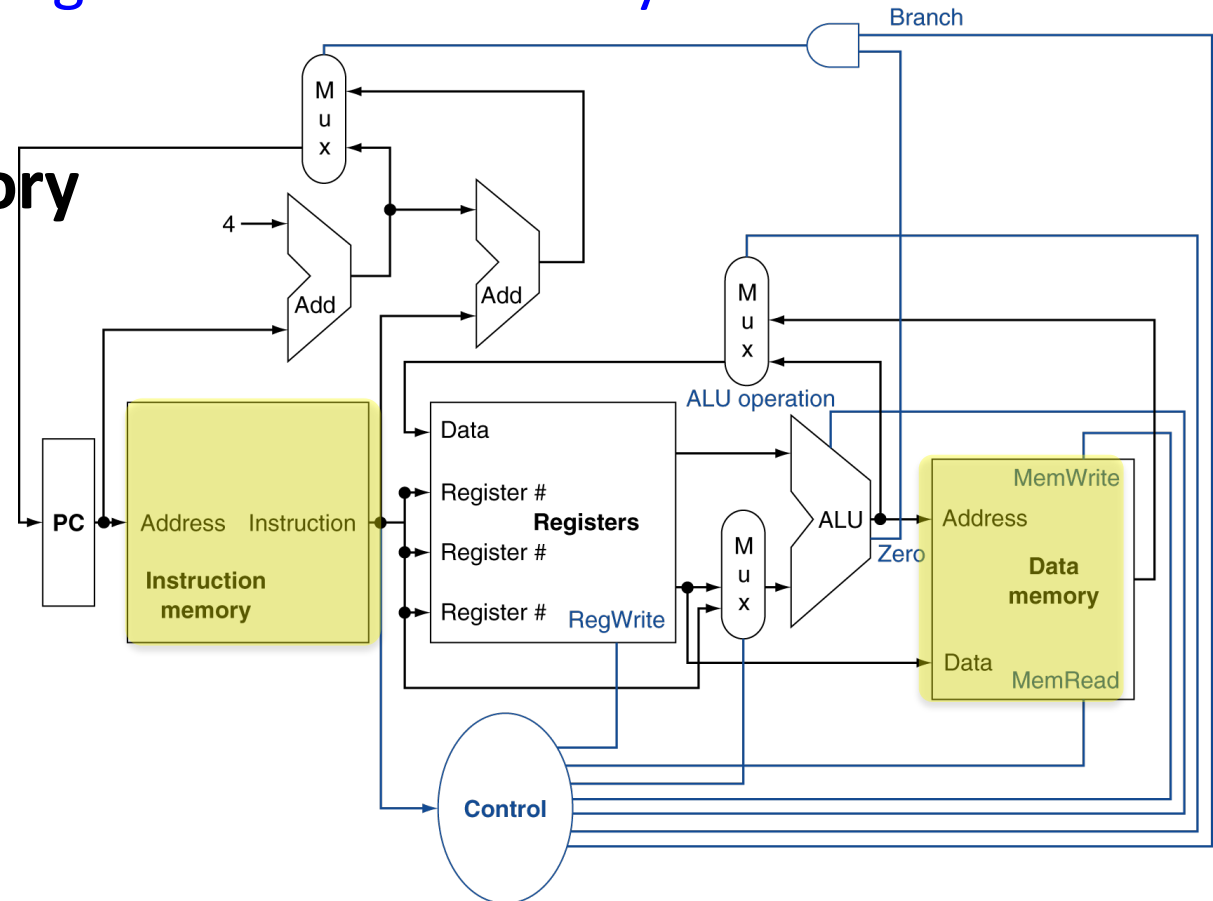
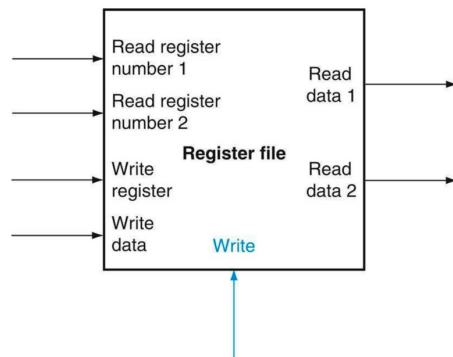
- We learned
 - how to design ALU, adder, gate, Mux, registers, etc
 - how to program, both high-level and assembly



- **Random Access Memory**

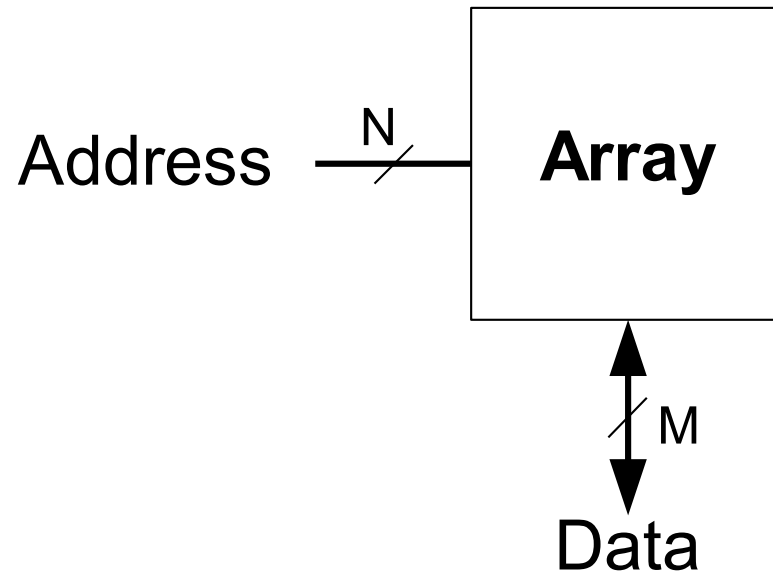
- Thinking of extending a register file to have larger capacity

- **32 64-bit registers**



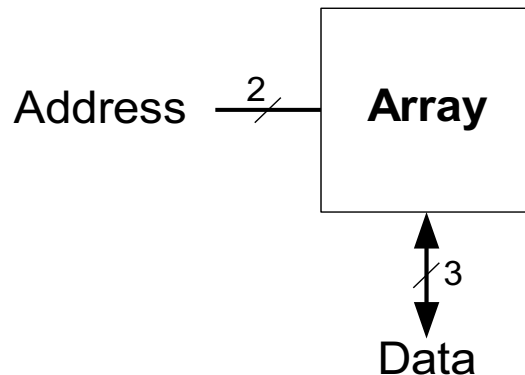
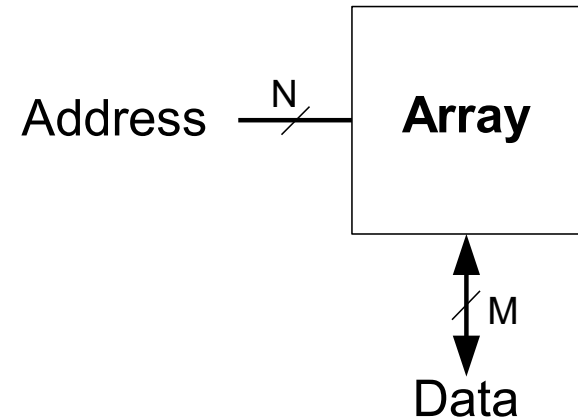
Memory Arrays

- Efficiently store large amounts of data
- 3 common types:
 - **Dynamic random access memory (DRAM)**
 - *Static random access memory (SRAM)*
 - Read only memory (ROM)
- M -bit data value read/ written at each unique N -bit address



Memory Arrays

- 2-dimensional array of bit cells
 - Consider it as a bigger register file
- Each bit cell stores one bit
- N address bits and M data bits:
 - 2^N rows and M columns
 - **Depth:** number of rows (number of words)
 - **Width:** number of columns (size of word)
 - **Array size:** depth \times width = $2^N \times M$



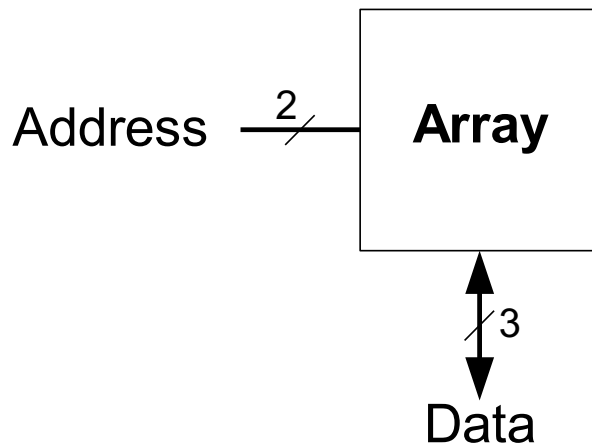
Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

↑
↓
 depth

←
→
 width

Memory Array Example

- $2^2 \times 3$ -bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

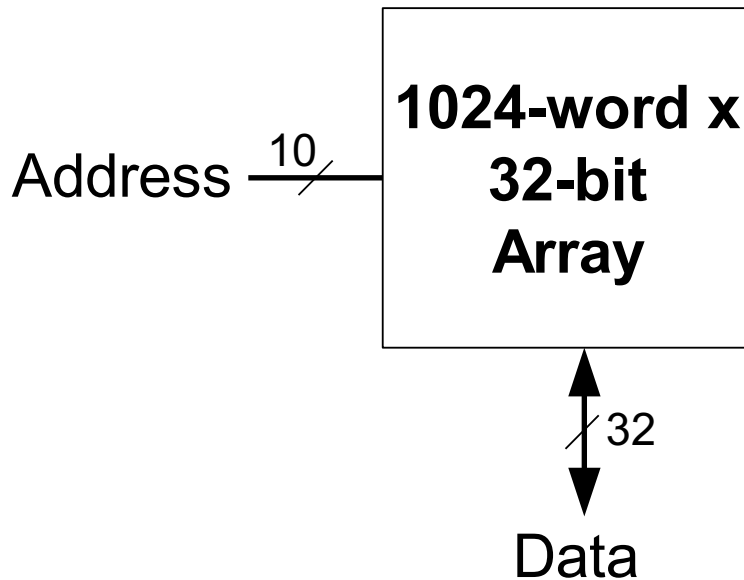


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

width

depth

Memory Arrays



```
long int a = A[8]; A[4] = b;
```

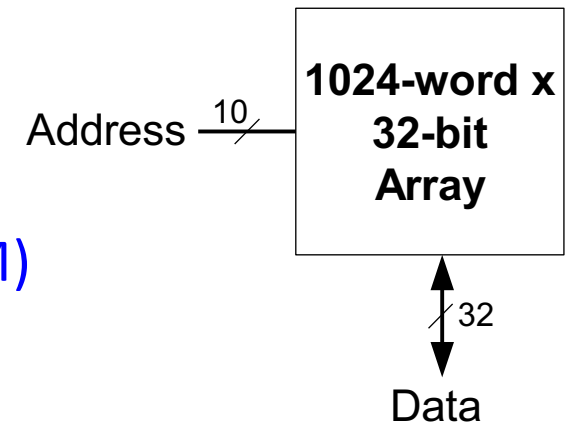
Instructions that access memory:

- Load: e.g. ld x5 64(s0)
 - Address: $64 + [s0]$
- Store: e.g. sd x6, 32(s0)
 - Address: $32 + [s0]$

- 10-bit address: $2^{10} = 1024$ word
- Each word is 32-bit
- Total: $1024 * 32$ bits = 32K bits = 4K bytes

Random-Access Memory (RAM)

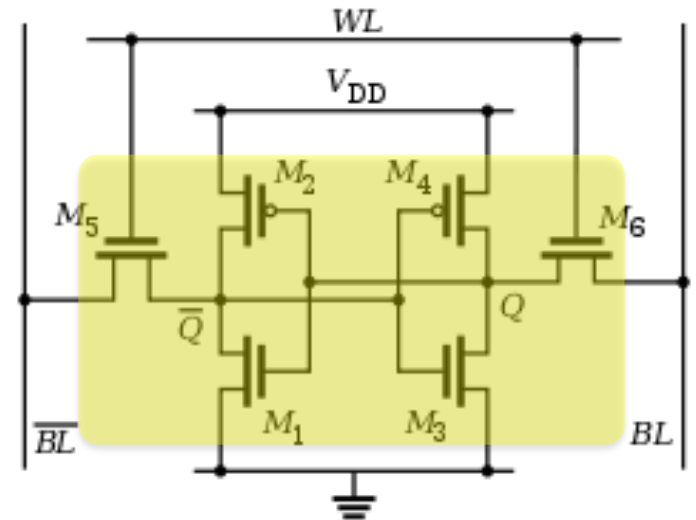
- Historically called **random access memory (RAM)** because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)
- Volatile:
 - loses its data when power off
 - Read and written quickly
 - Main memory in your computer is RAM (DRAM)
- **Two types:**
 - **DRAM (Dynamic random access memory)**, main memory of computer
 - **SRAM (Static random access memory)**
- Differ in how they store data:
 - DRAM uses a capacitor
 - SRAM uses cross-coupled inverters



SRAM and DRAM Technology Differences

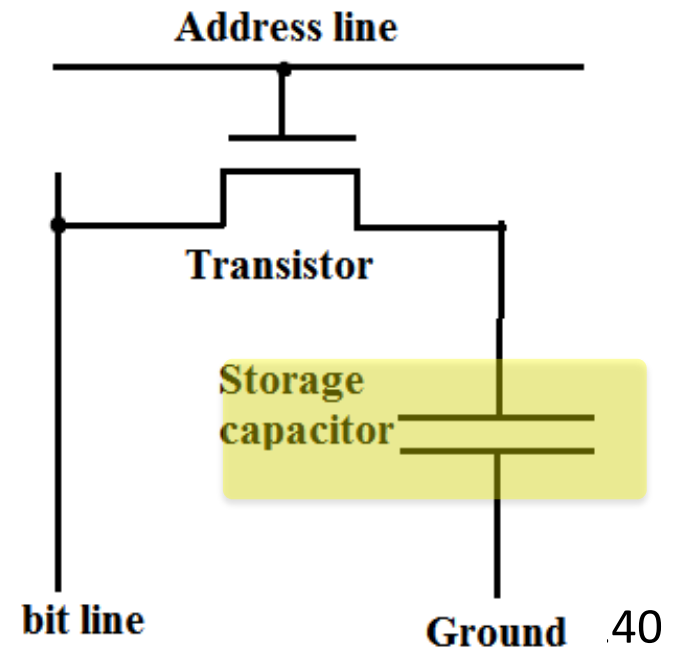
- **Static RAM (SRAM)**

- Each cell stores a bit with a six-transistor circuit, a flip-flop
- Retains value indefinitely, as long as it is kept powered.
- Relatively insensitive to disturbances such as electrical noise.
- Faster and more expensive than DRAM.



- **Dynamic RAM (DRAM)**

- Data stored as a charge in a capacitor
- Single transistor used to access the charge
- Dynamic: need to be “refreshed” regularly, every 10-100 ms.
- Sensitive to disturbances.
- Slower and cheaper than SRAM.



SRAM (Static Random Access Memory)

- Data is stored statically in flip-flop
 - As long as it is powered, data (high/low voltage) will be stored.
 - RAM (random access): fixed access time to any datum
 - **Not like spinning disk.**
 - Same as register
- A 2M x 16 SRAM module: 2 M ($2 * 2^{20}$) 16-bit entries
 - Address line has 21 bits $\rightarrow 2^{21}$, which is $2 * 2^{20} = 2M$ rows
 - Each row has 16 bits, thus total $2M * 16 = 32M$ bits = $32 * 2^{20}$ bits = 4MByte)

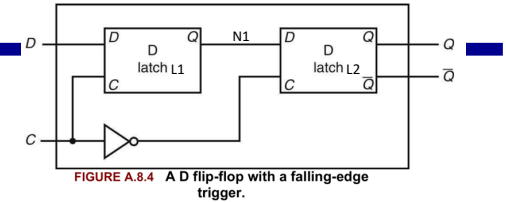
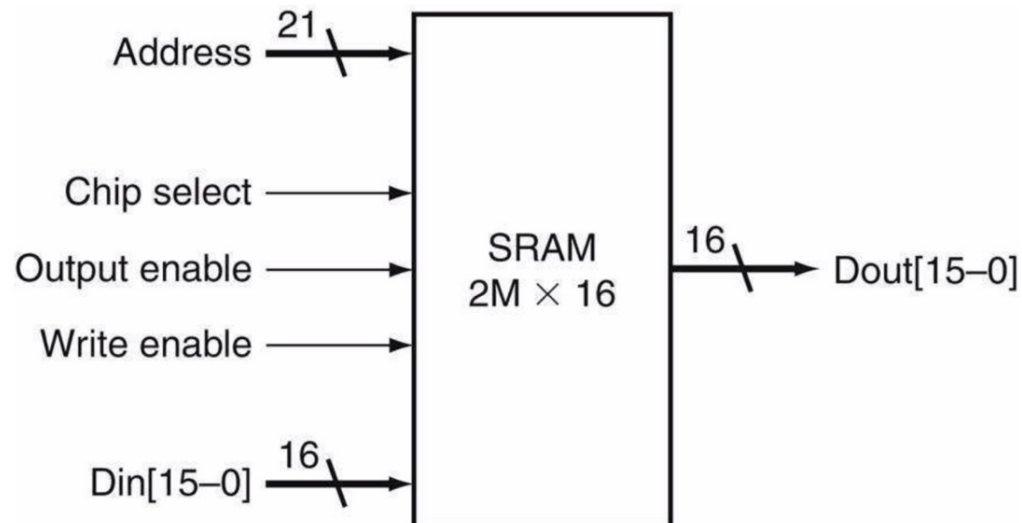
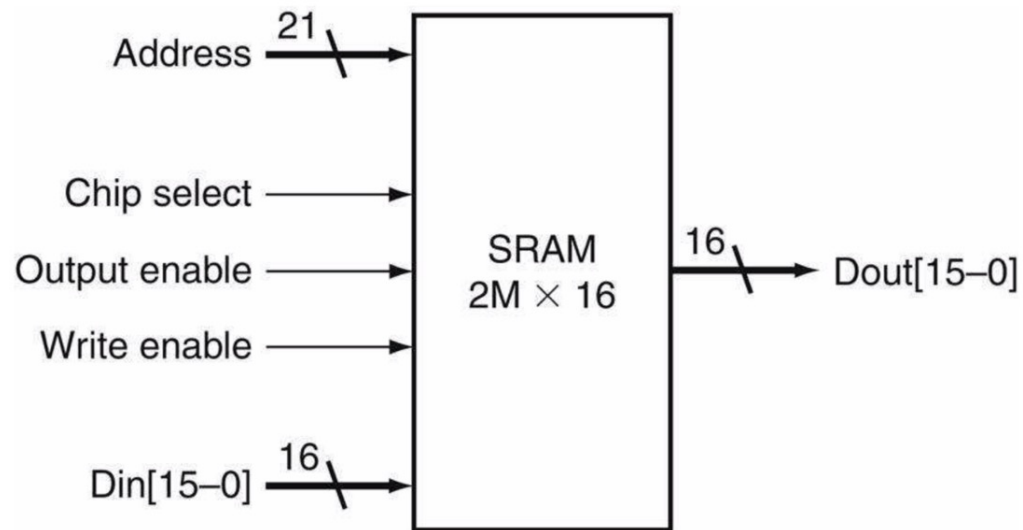


FIGURE A.8.4 A D flip-flop with a falling-edge trigger.



Read of SRAM (Static Random Access Memory)

- A 2M x 16 SRAM module: 2 M ($2 * 2^{20}$) 16-bit entries
 - Address line has 21 bits $\rightarrow 2^{21}$, which is $2 * 2^{20} = 2M$ rows
 - Each row has 16 bits, thus total $2M * 16 = 32M$ bits = $32 * 2^{20}$ bits = 4MByte)
- **Read**
 - **Input:**
 - **Chip select**
 - **Output enable**
 - **Address**
 - **Output:**
 - **Dout**
- **Read access time (latency):** time to initiate read to when data is available on Dout
 - **2-4 ns**



Write of SRAM (Static Random Access Memory)

- A 2M x 16 SRAM module: 2 M ($2 * 2^{20}$) 16-bit entries
 - Address line has 21 bits $\rightarrow 2^{21}$, which is $2 * 2^{20} = 2M$ rows
 - Each row has 16 bits, thus total $2M * 16 = 32M$ bits = $32 * 2^{20}$ bits = 4MByte)

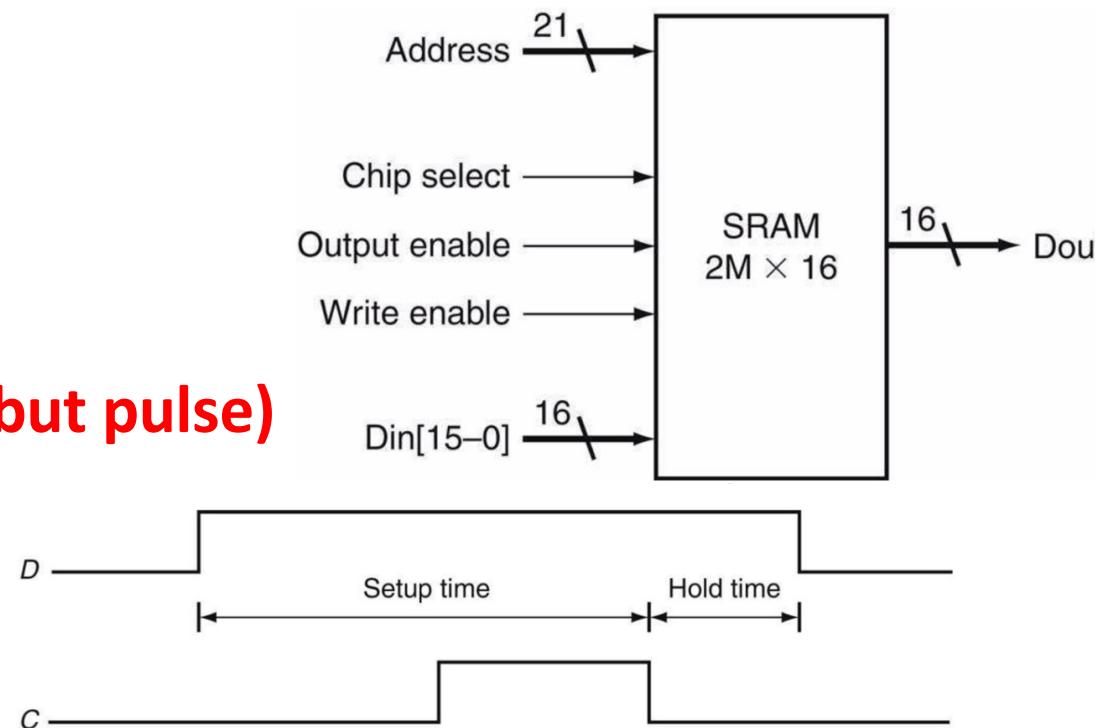
- **Write**

- **Input:**

- **Chip select**
- **Write enable (not clock, but pulse)**
- **Address**
- **Din**

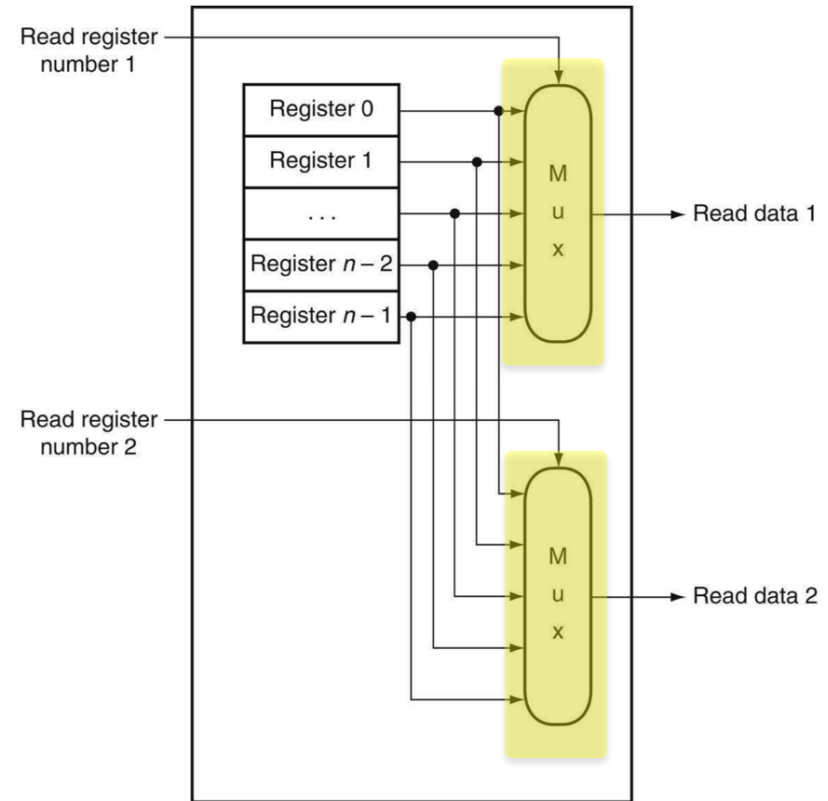
- **Output: (not output)**

- **Write time: setup time, hold time and pulse width**



Memory Read Implementation

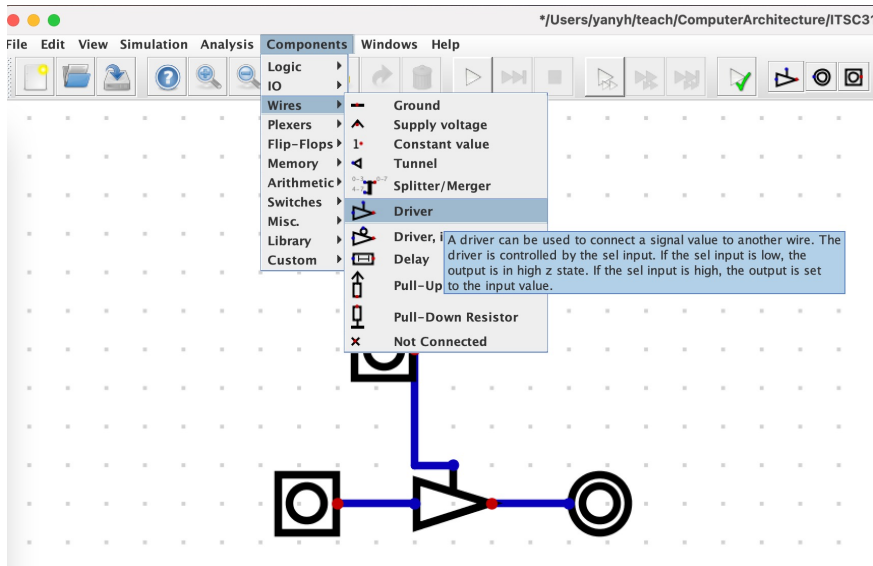
- Recall register read:
 - A 32-1 Mux to select one of the register
 - 32 registers
- Not practical for select from one of the large amount of memory word using the regular **centralized** Mux
 - 64k x 1 memory array needs a 64k-1 mux
- Memory use tristate buffer to create a mux
 - “Mux” is **distributed** to the memory cell



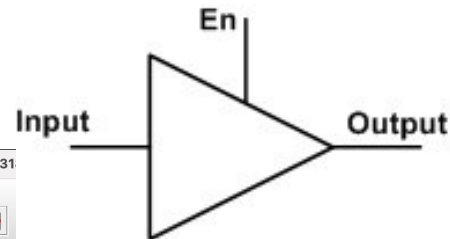
Tristate Buffer

- Tristate buffer, or Three-state buffer
 - 1: asserted
 - 0: deasserted
 - **Hi-Z state: to allow other control the output**

- **“Driver” in Digital**



Symbol

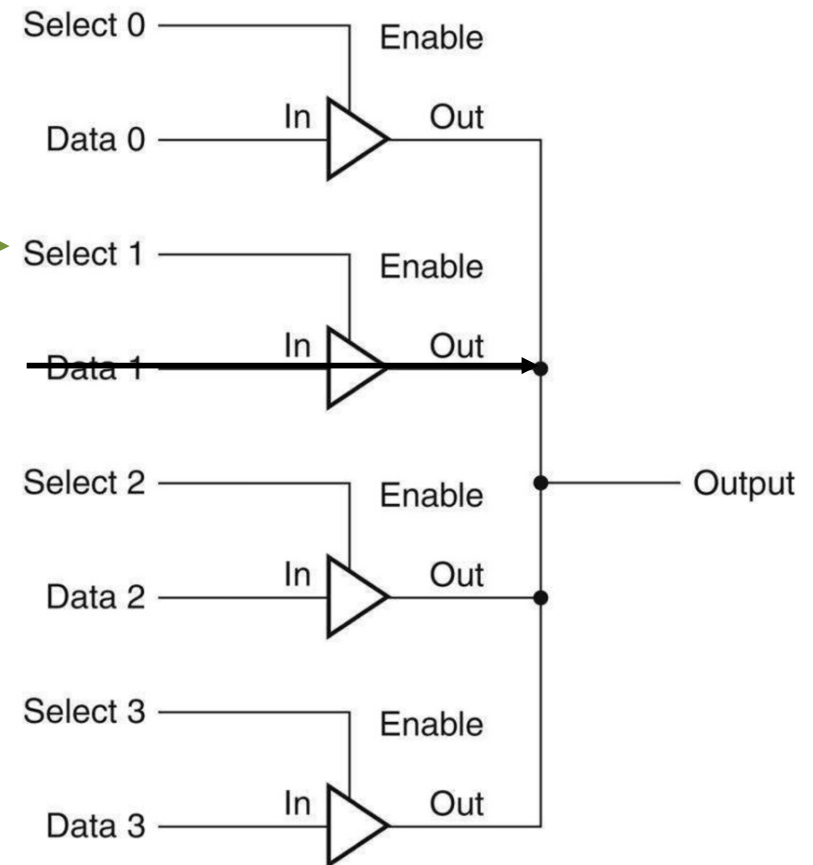
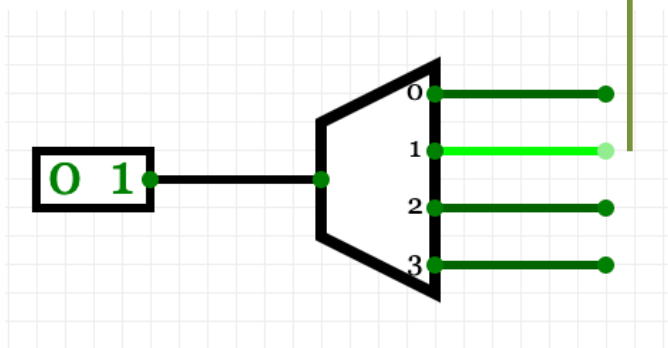


Truth Table

En	Input	Output
0	X	Hi-Z
1	0	0
1	1	1

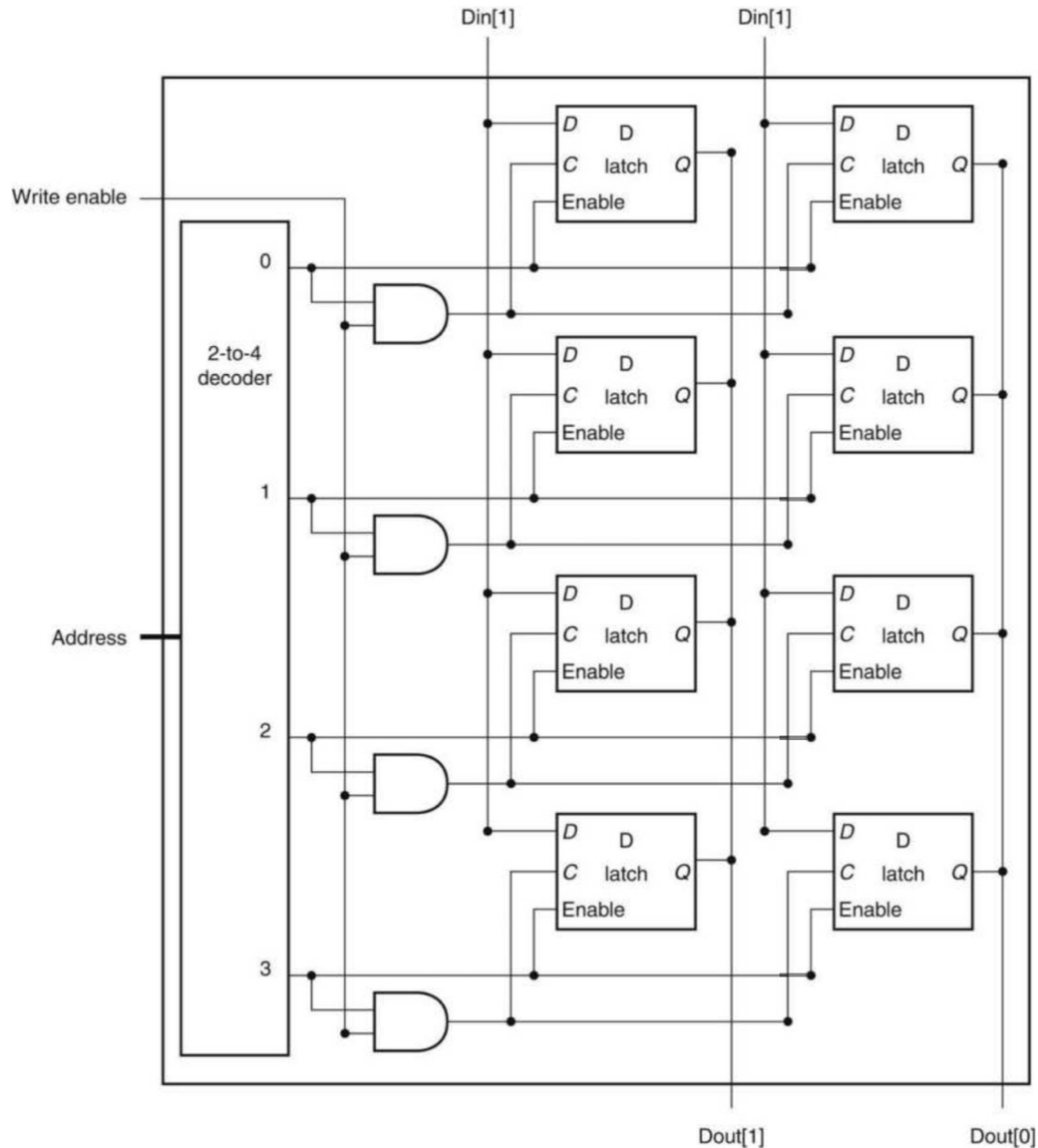
4-1 Distributed Mux Using Tristate Buffer

- Select bit is the output of a decoder whose input is the address
- Select bit to select one of the data from memory to drive the output line (become the output)
 - No need Mux to select
- Create a distributed Mux



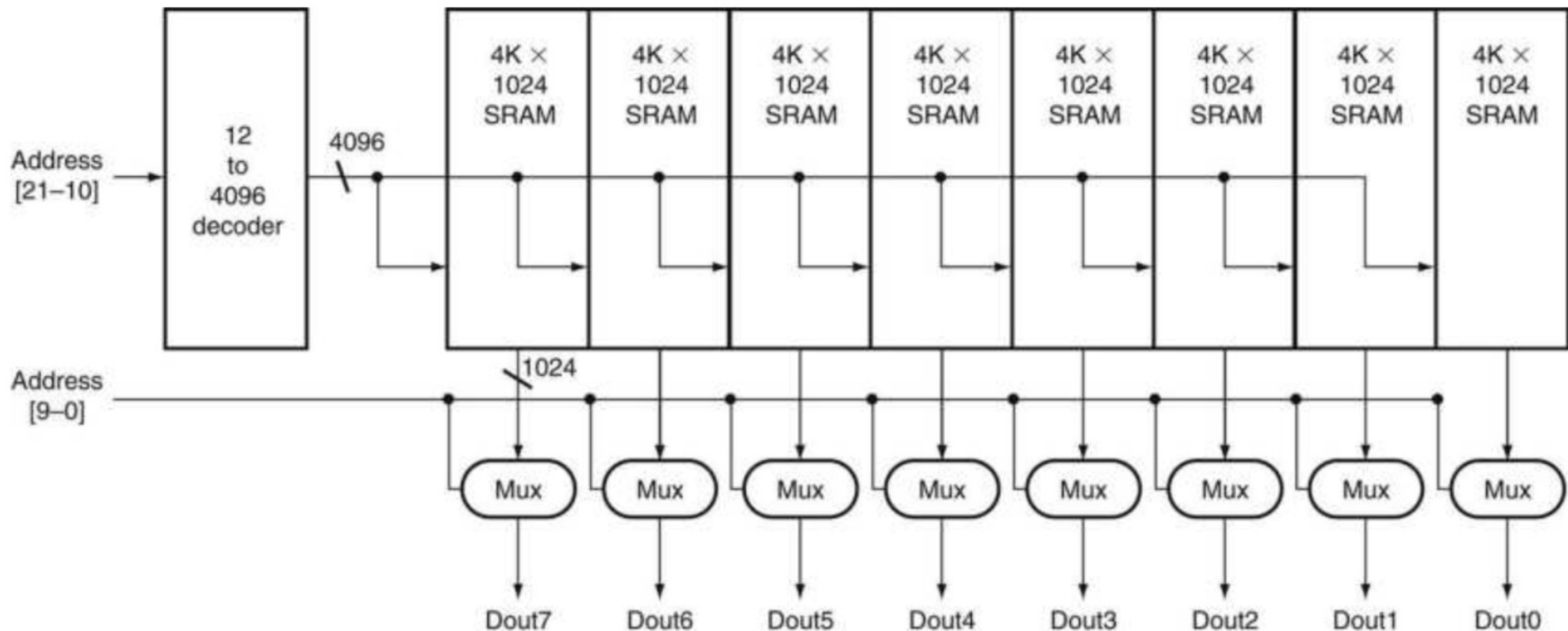
Another Design for Not Using Centralized Large Mux

- 4x2



Two-level of Decoding

- 4M x 8 SRAM: 4K rows and each has 8 bits
- First decoder: select 8 1024-bit-wide data from 4K
- Second: 8 Muxes, each to select 1 bit from each 1024-bit-wide data
- Two steps: Increase read latency



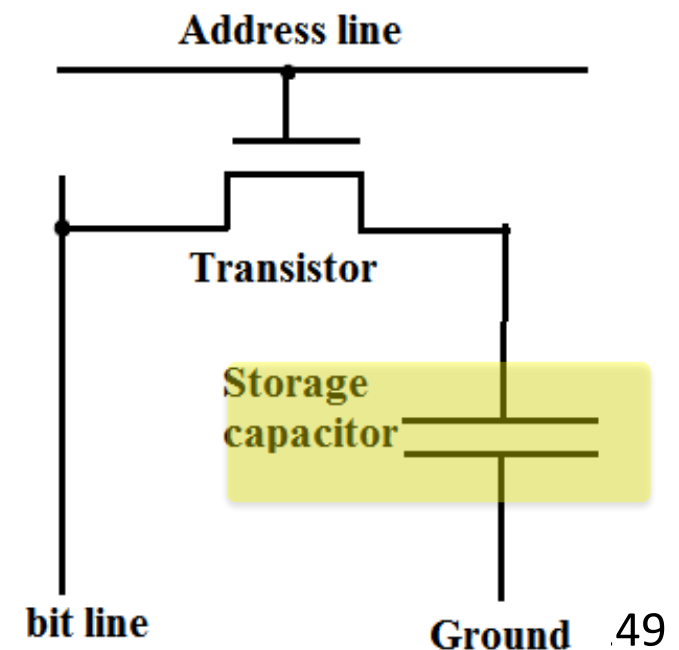
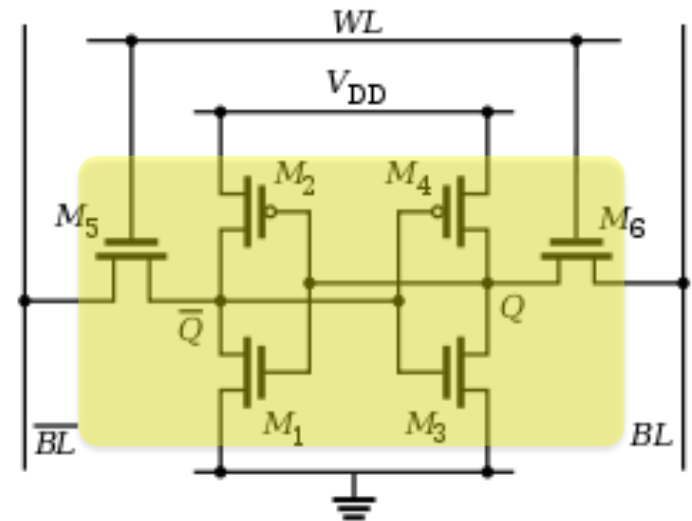
DRAM (Dynamic Random Access Memory)

- **Static RAM (SRAM)**

- Each cell stores a bit with a six-transistor circuit, a flip-flop
- Faster and more expensive than DRAM.

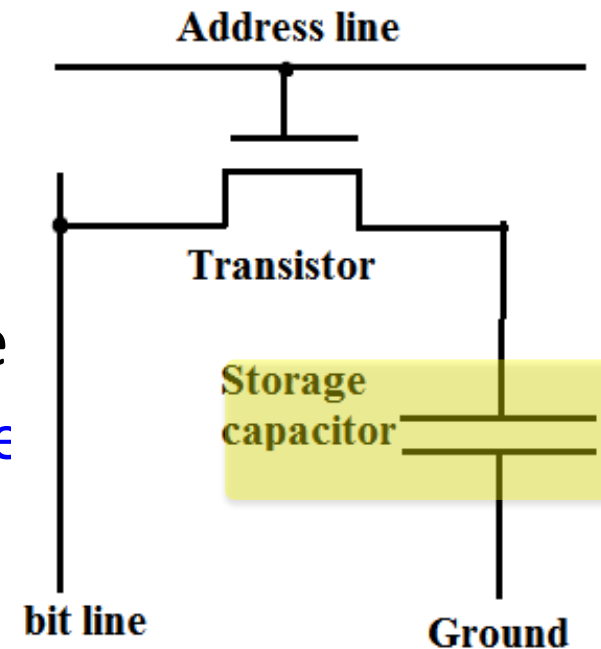
- **Dynamic RAM (DRAM)**

- Data stored as a charge in a capacitor
- Single transistor used to access the charge
- **Dynamic: need to be “refreshed” regularly, every 10-100 ms.**
- Sensitive to disturbances.
- Slower and cheaper than SRAM.



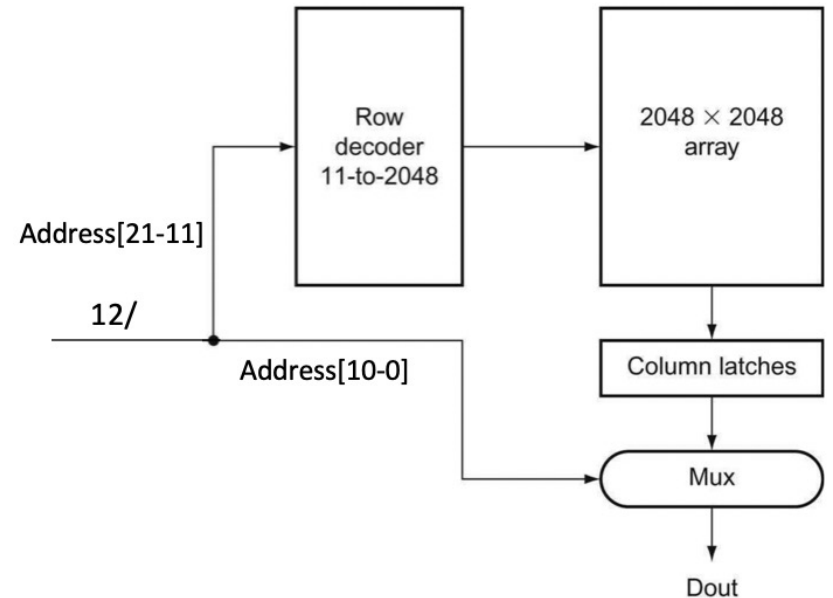
To Refresh a DRAM Cell

- Refresh: Read its content and write it back
 - Every several milliseconds
- Refreshing conflicts with normal read/write
 - Two-level decoding → refresh one row a time
 - Refresh consume 1% - 2% of active cycles
 - 98% - 99% cycles for normal read and write



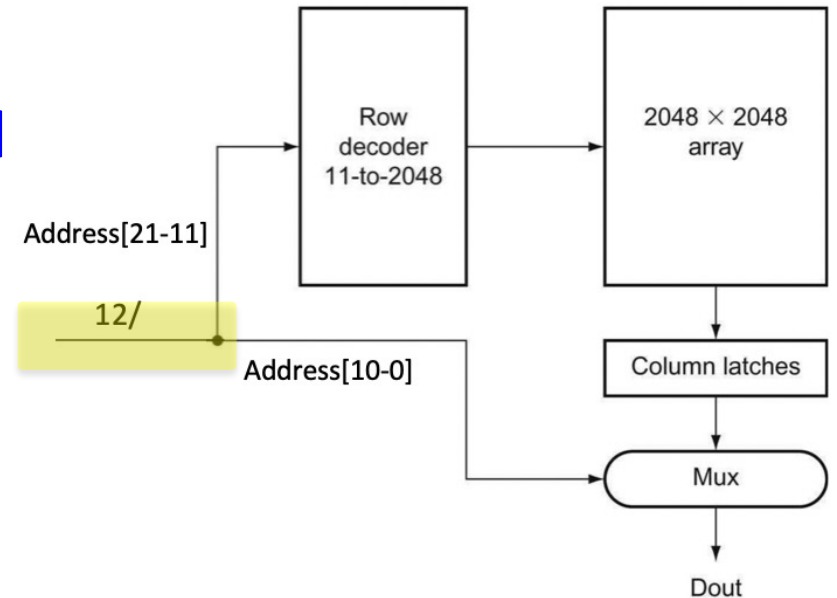
Two-Level Decoder for DRAM

- Row access
 - Select one of a number of rows and activate words of the row
 - Store words in the column latches
- Column access
 - Select data from column latches
- 4M x 1 DRAM built with a 2048 x 2048 Array
 - 22-bit address line
 - Address[21-11]: Row address
 - Select a row and latch 2048 bits in the column latches
 - Address[10-0]: Column address
 - Select one bit from the 2048 latches



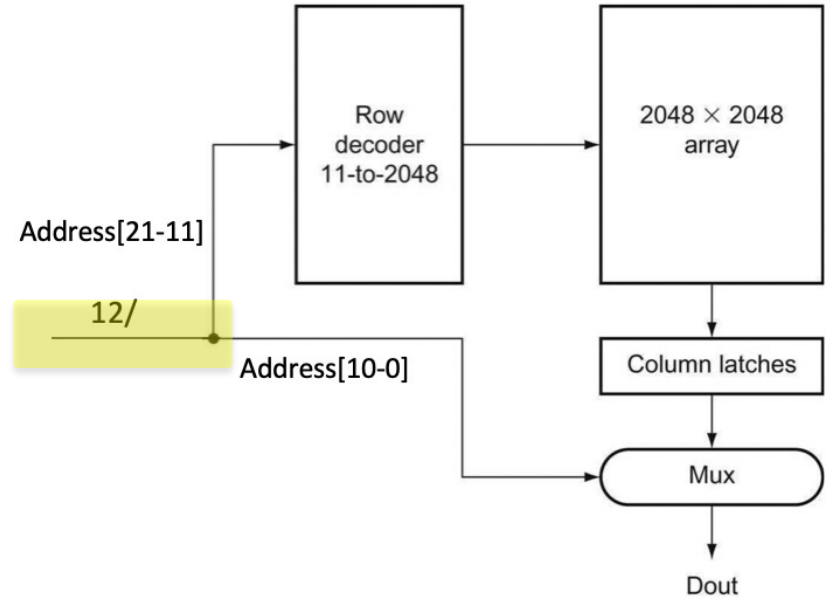
Two-Level Decoder for DRAM

- Row access
 - Select one of a number of rows and activate words of the row
 - Store words in the column latches
- Column access
 - Select data from column latches
- Use the same address wire for row address and column address
 - RAS (Row Access Strobe) and CAS (Column Access Strobe) are used to signal DRAM either a row or column address is being supplied
- Refreshing:
 - Reading data into column latch and write them back

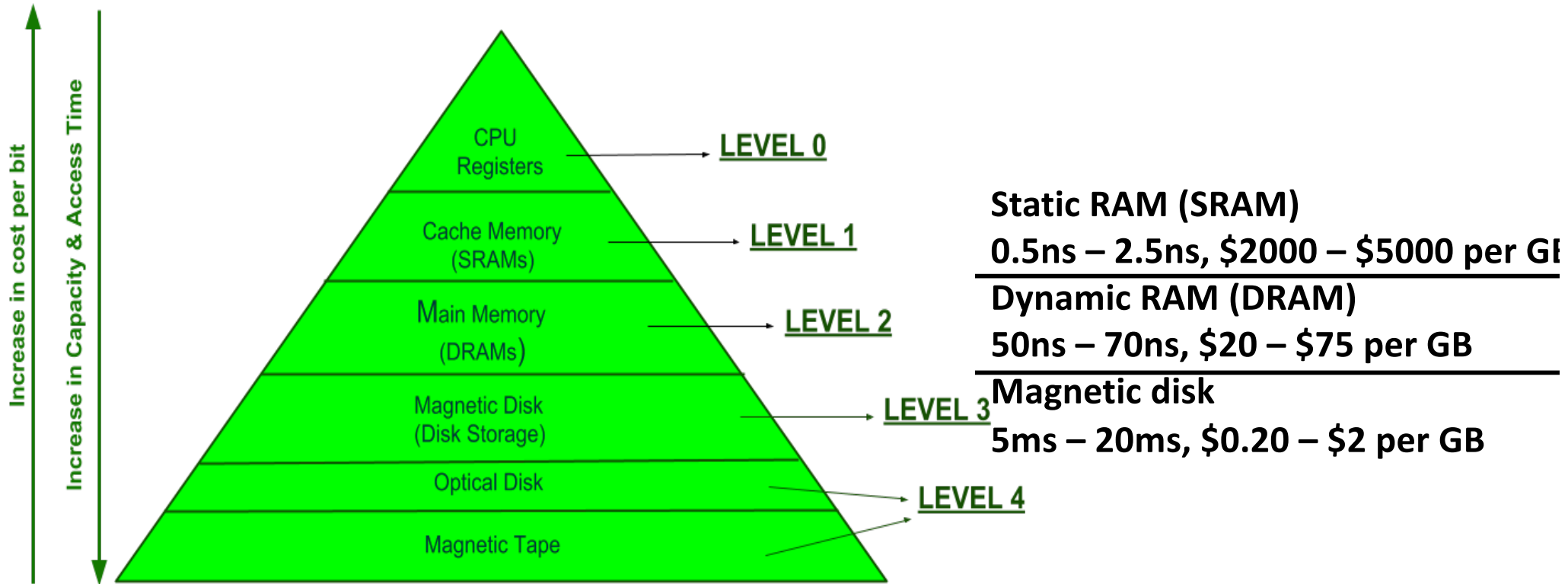


Two-Level Decoder for DRAM

- Row access
 - Select one of a number of rows and activate words of the row
 - Store words in the column latches
- Column access
 - Select data from column latches
- DRAM Access Time
 - Because of two-level decoding and internal circuitry,
 - 5-10 times slower than SRAM access time, e.g. 45-65 ns

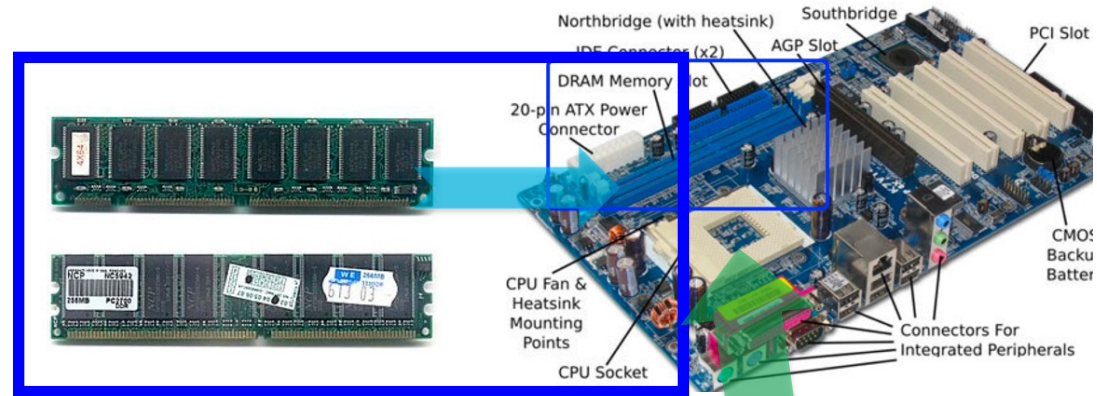
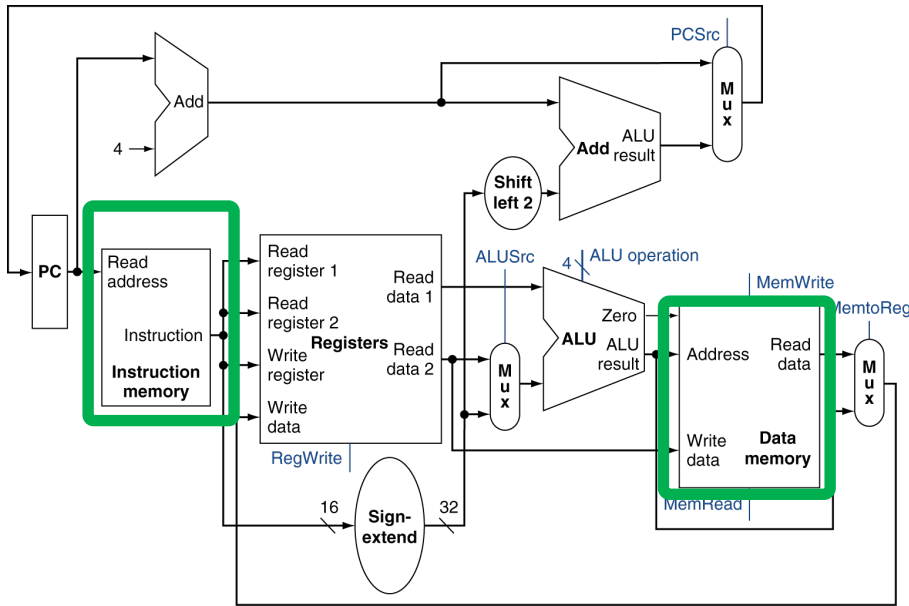


Memory Hierarchy of Computer in Real

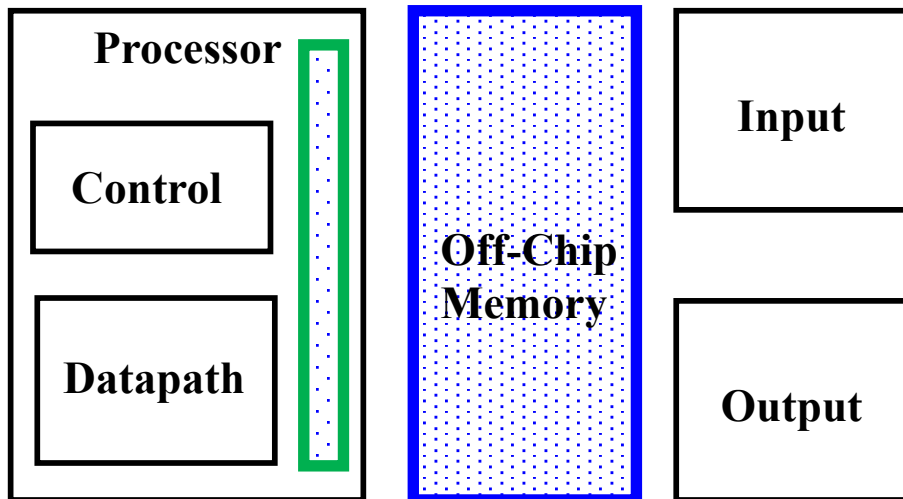
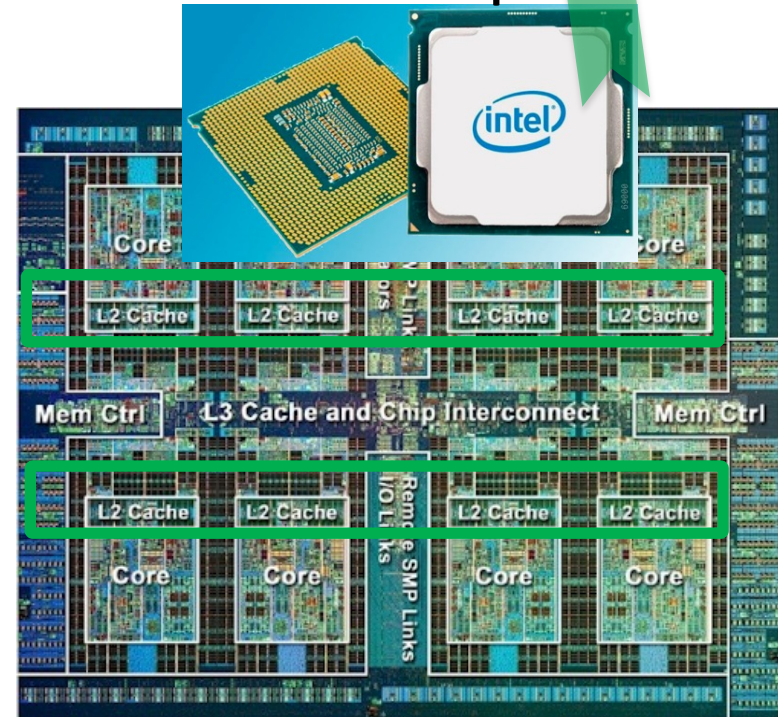


MEMORY HIERARCHY DESIGN

Green Boxes: Cache, on-chip, SRAM, fast, small, expensive
Blue Boxes: Main memory, off-chip, DRAM, slower, large not expensive



CPU is The chip.



Lecture Ends Here

Appendix A: The Basics of Logic Design

- **Lecture 12**
 - **A.1 Introduction**
 - **A.2 Gates, Truth Tables, and Logic Equation**
- **Lecture 13**
 - **A.3 Combinational Logic**
 - ~~A.4 Using a Hardware Description Language~~
- **Lab 7**
- **Lecture 14**
 - **A.5 Constructing a Basic Arithmetic Logic Unit**
 - ~~A.6 Faster Addition: Carry Lookahead~~
- **Lecture 15**
 - **A.7 Clocks**
 - **A.8 Memory Elements: Flip-Flops, Latches, and Registers**
- **Lab 8**
- **Lecture 16**
 - **A.9 Memory Elements: SRAMs and DRAMs**
- **Lab 9**
- ~~Lecture 17~~
 - **A.10 Finite-State Machines**
 - ~~A.11 Timing Methodologies~~
 - ~~A.12. Field Programmable Devices~~
 - **A.13 Concluding Remarks**
 - ~~A.14 Exercises~~
- **Lab 10**

Combinational and Sequential Circuits

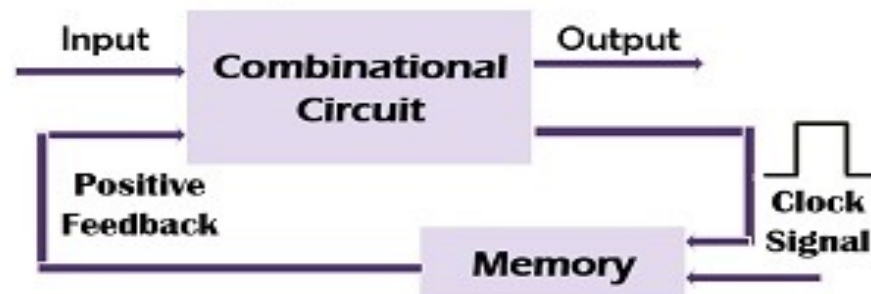
- Combinational circuit, such as mux, decoder, ALU
 - Operate on data
 - Output is a function of input
- State (sequential) circuit, such as register or memory
 - Store information
 - Outputs determined by previous and current values of inputs



Combinational Circuit

Vs

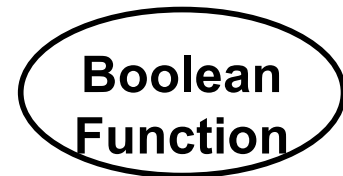
Sequential Circuit



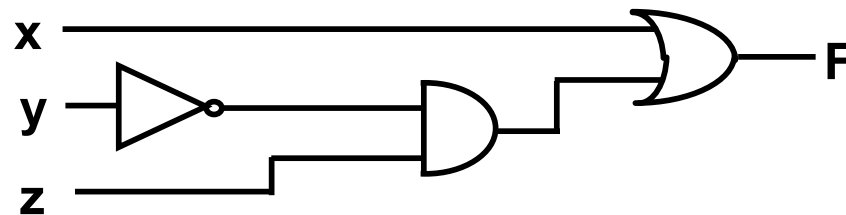
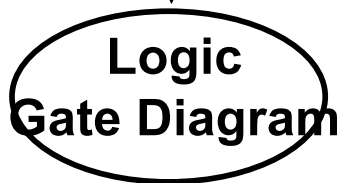
Three Steps of Logic Design in Theory, Mostly for Combinational Circuit, so far



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

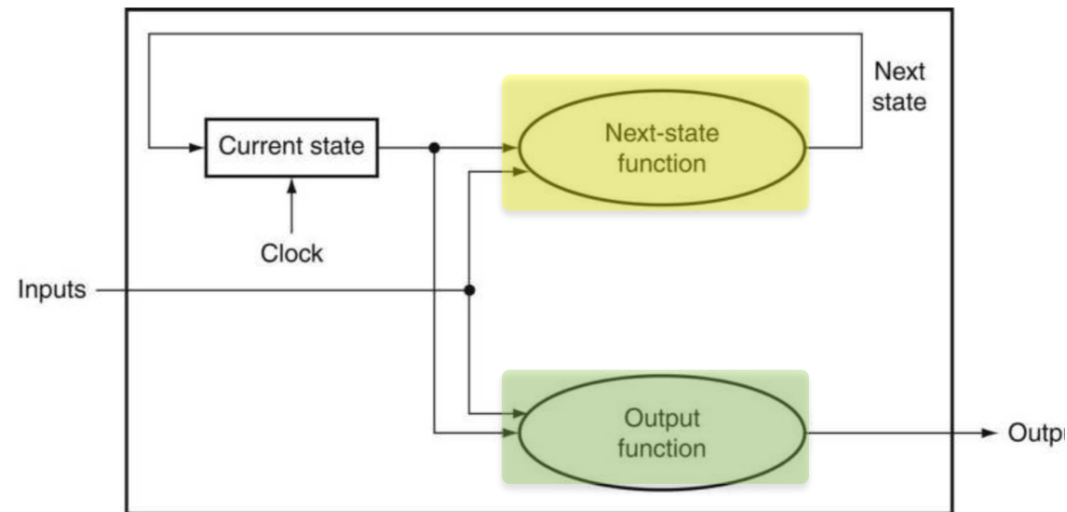
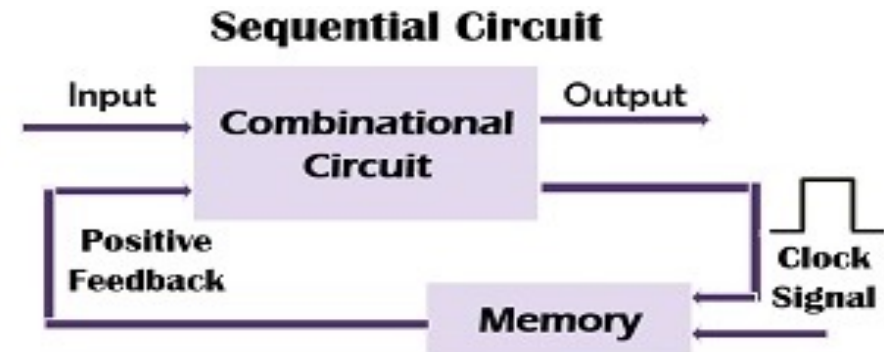


$$F = x + y'z$$



Sequential System is described as a Finite-state Machine (FSM)

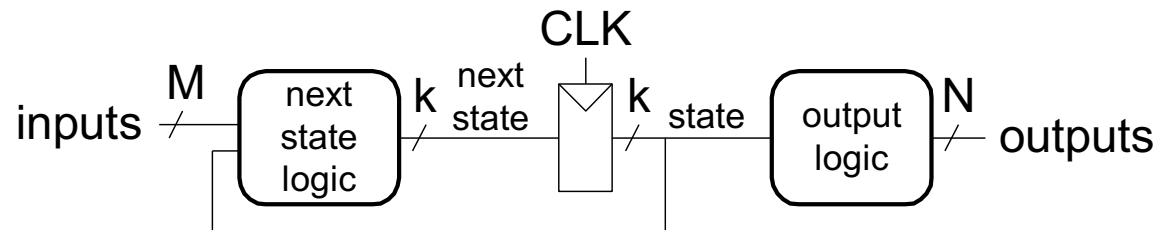
- Input, output and states
- State: Given n bits of storage (memory)
 - 2^n state
- Next-state function
 - Combinational logic, given inputs and current states, determines the next state of the system
- Output function
 - Produce outputs from current state and inputs



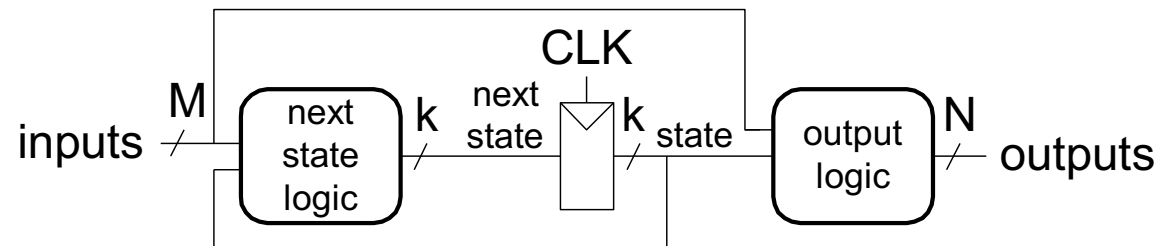
Finite State Machines (FSMs)

- Two types of finite state machines differ in output logic:
 - **Moore FSM: outputs depend only on current state**
 - **Mealy FSM: outputs depend on current state and inputs**
 - These two types are equivalent in capabilities, can convert from one to the other
 - We are only going to deal with the Moore machine.

Moore FSM

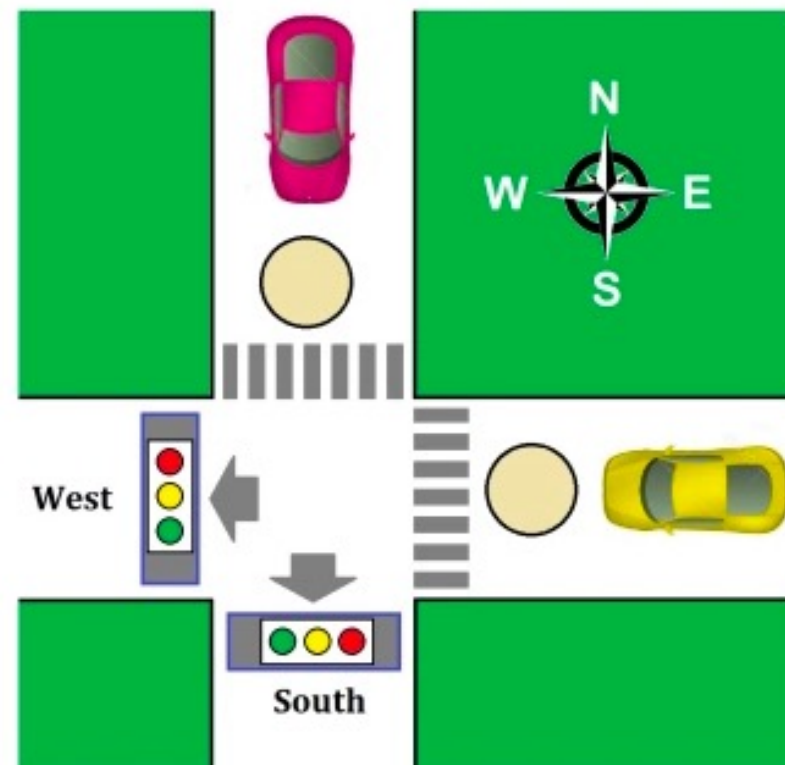


Mealy FSM



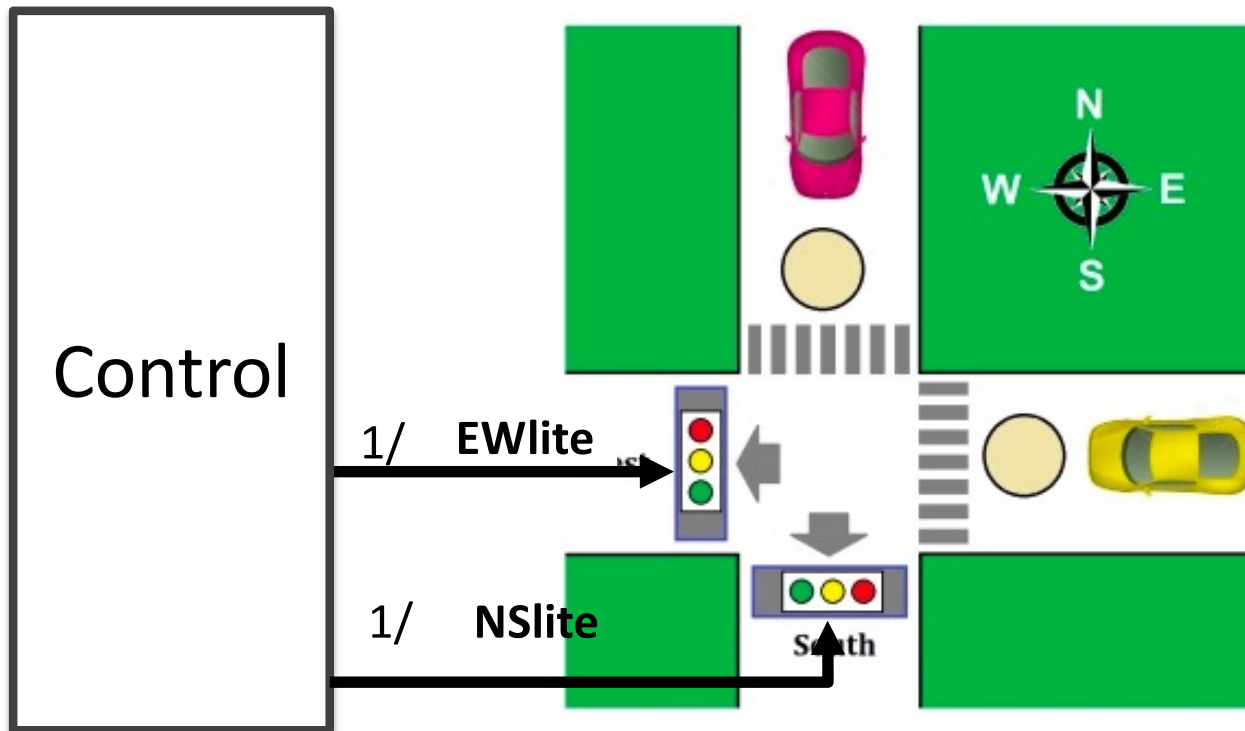
Intelligent Traffic Controller

- We want to use a finite state machine to control the traffic lights at an intersection of a north-south route and an east-west route
 - We consider only the green and red lights
 - We want the lights to change no faster than 30 seconds in each direction
 - So we use a 0.033 Hz clock



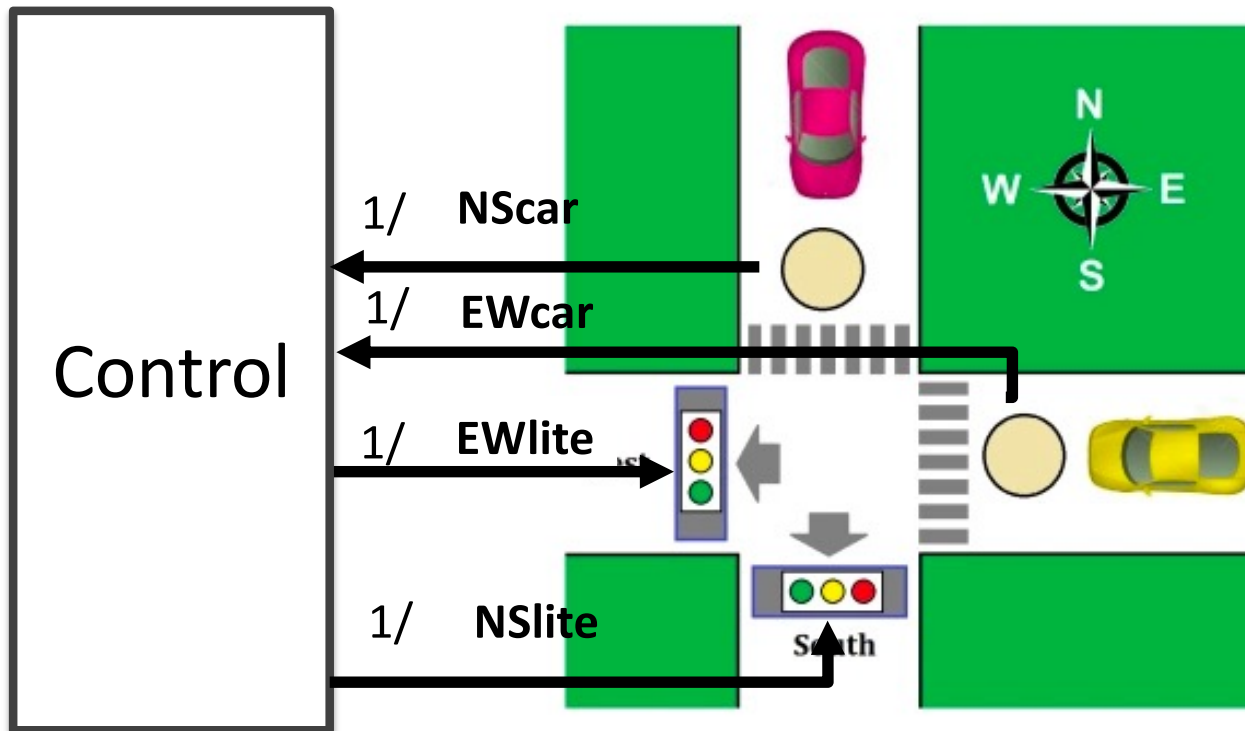
Intelligent Traffic Controller

- There are two output signals:
 - NSlite: When the signal is asserted, the light on the north-south route is green; otherwise, it should be red
 - EWlite: When the signal is asserted, the light on the east-west route is green; otherwise, it should be red



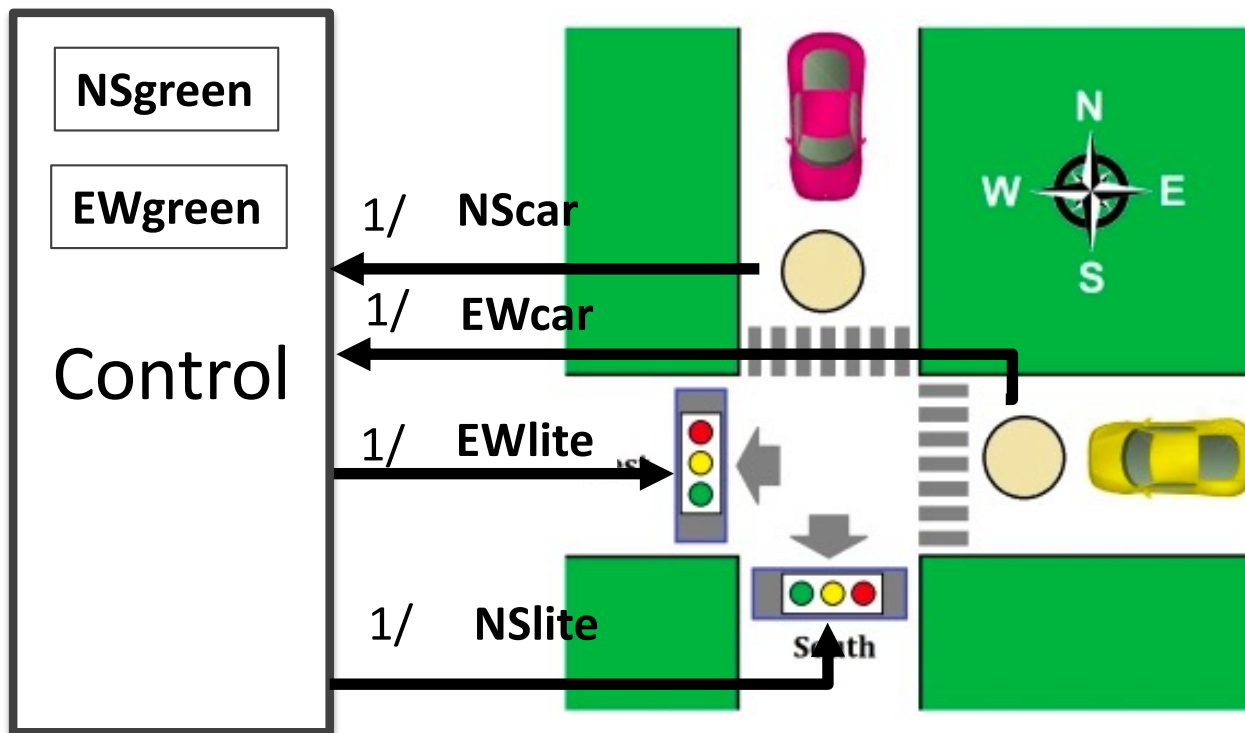
Intelligent Traffic Controller

- There are two inputs
 - NScar: Indicates that there is at least one car that is over the detectors placed in the roadbed in the north-south road
 - EWcar: Indicates that there is at least one car that is over the detectors placed in the roadbed in the east-west road



Intelligent Traffic Controller

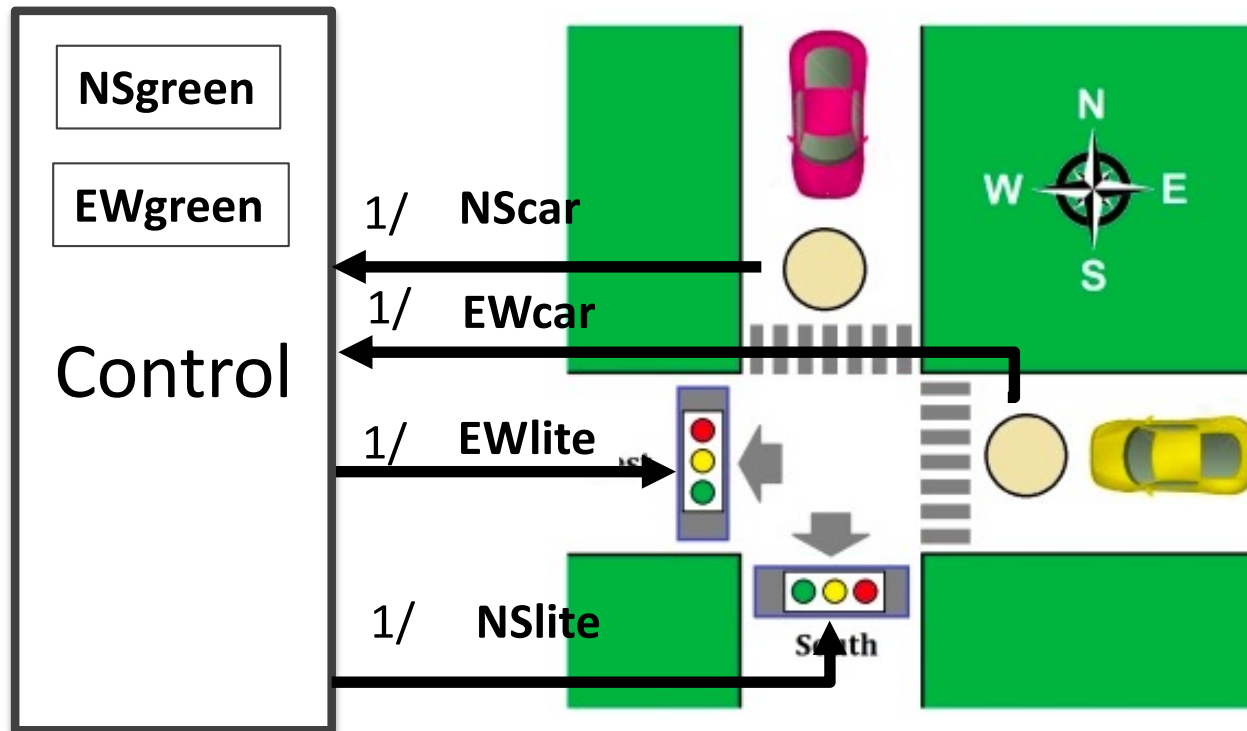
- Here we need two states
 - *NSgreen*: The traffic light is green in the north-south direction
 - *EWgreen*: The traffic light is green in the east-west direction



Intelligent Traffic Controller: Traffic Lights

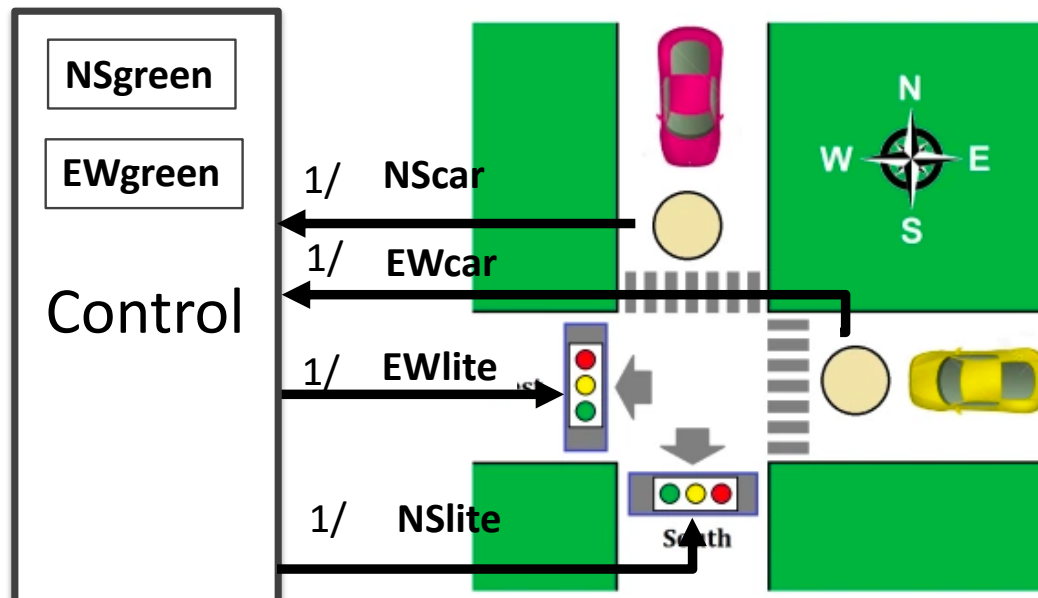
Change only When They Need To

- The traffic lights should only change from one direction to the other only if there is a car waiting in the other direction
 - Otherwise, the light should continue to show green in the same direction



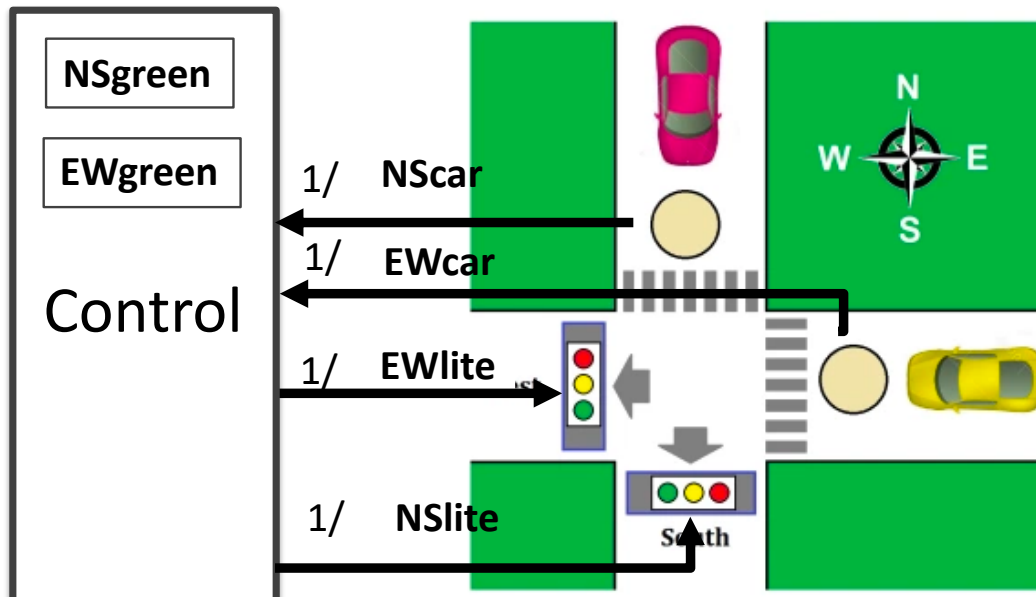
Next State Function

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen



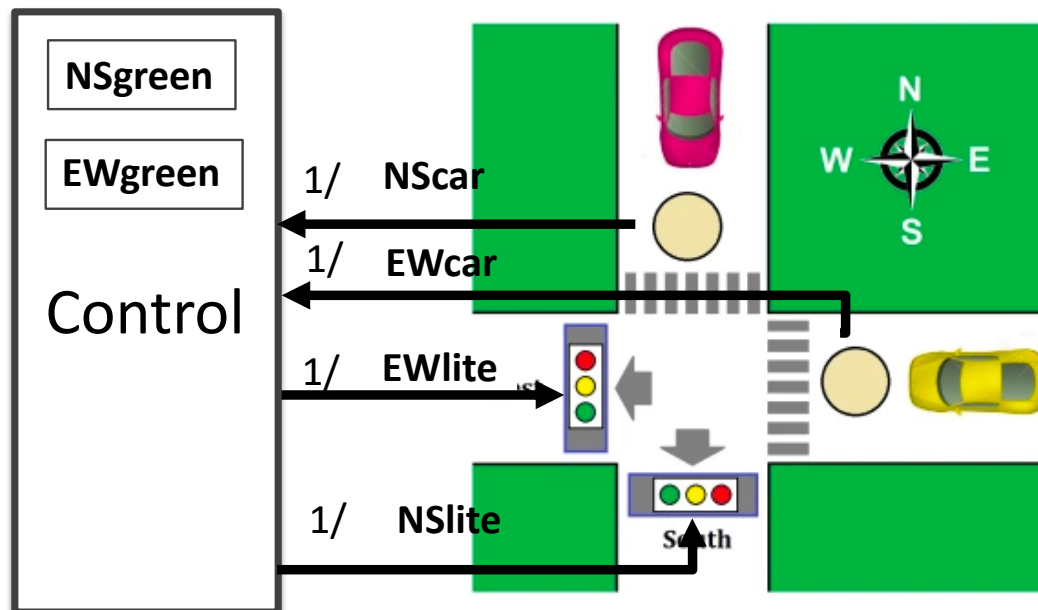
Next State Function: If cars are in both directions, alternate

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen



Output Function

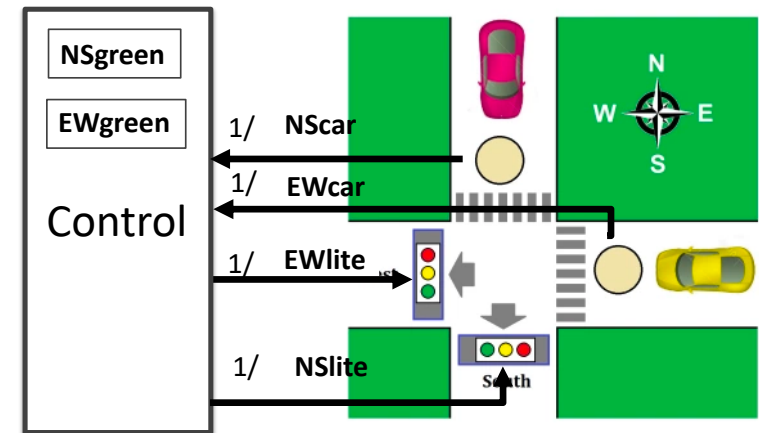
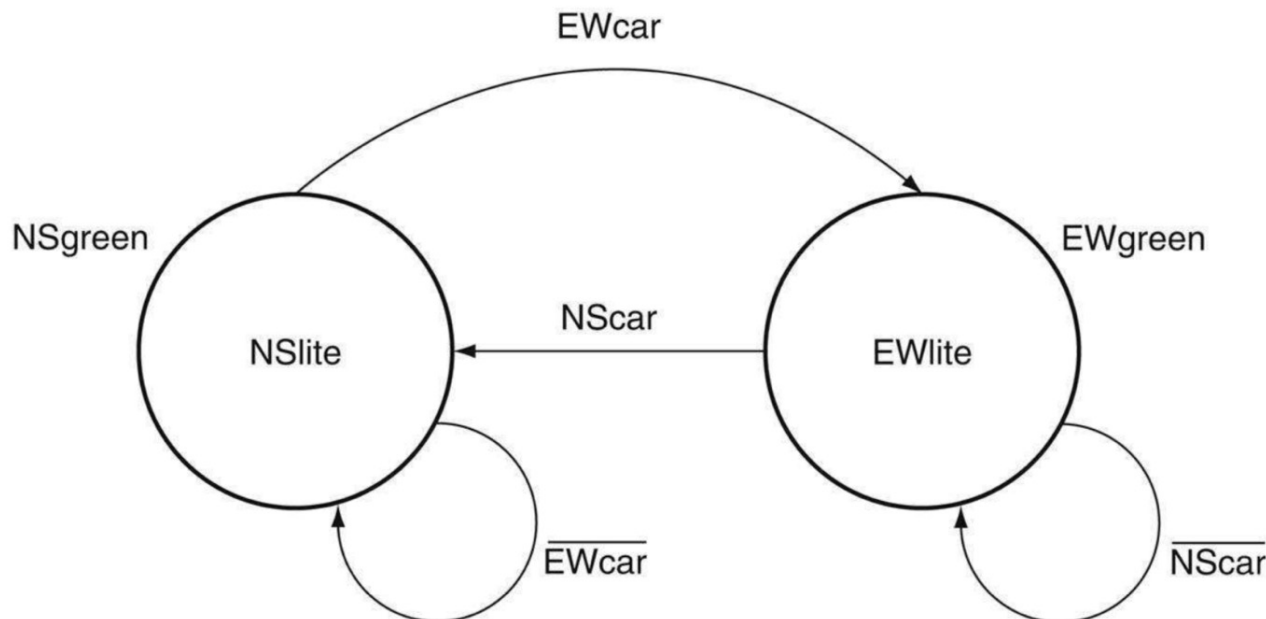
Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1



Graphical Representation

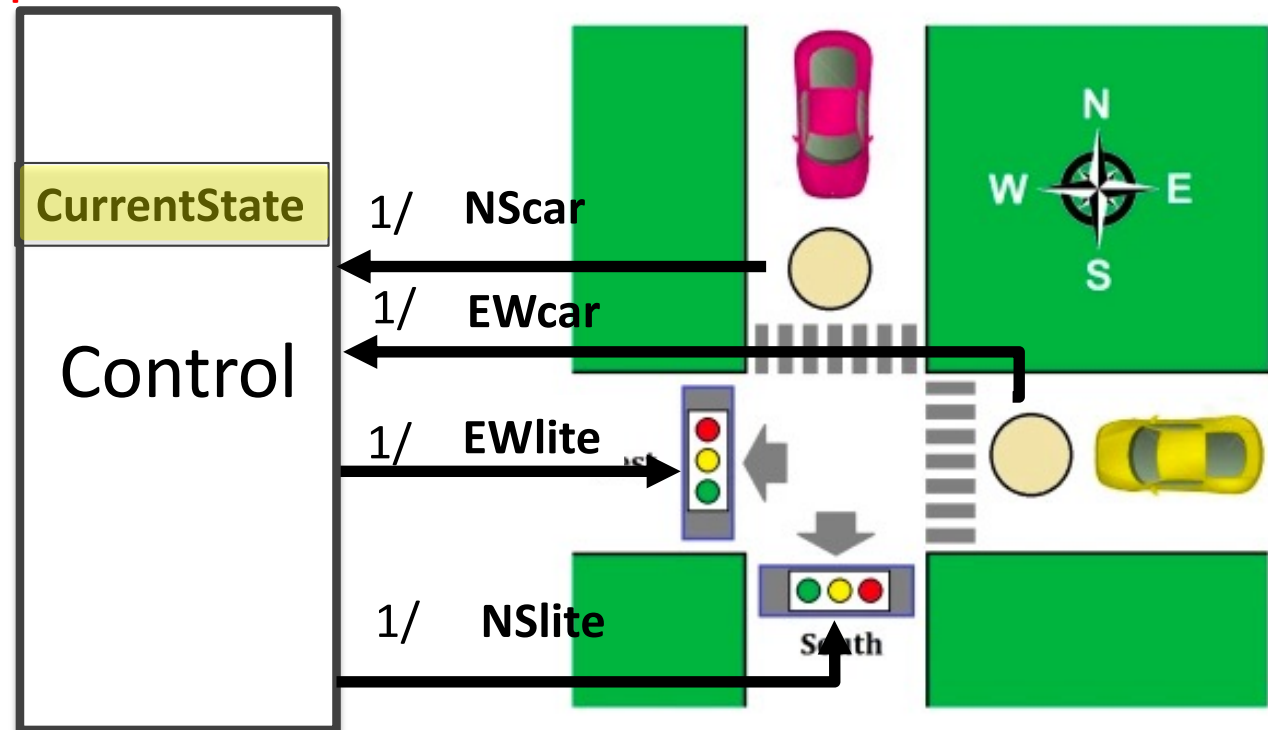
- Node: state
 - Inside: a list of output that are active for the state
- Directed Arch: next-state func
 - Labels: input

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen



Implementation: State Assignment

- We need to assign state numbers to the states
 - Only two states: assign 0 to NSgreen and 1 to EWgreen
 - Therefore we only need 1 bit in the state register
 - CurrentState:
 - 0: NSGreen
 - 1: EWgreen



Combinational Logic for Next State Function

Current state	Inputs		Next state
	NScar	EWcar	
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

Combinational Logic for Output Function

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

– CurrentState:

- 0: NSGreen
- 1: EWgreen

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

$$\text{NSlite} = \overline{\text{CurrentState}}$$

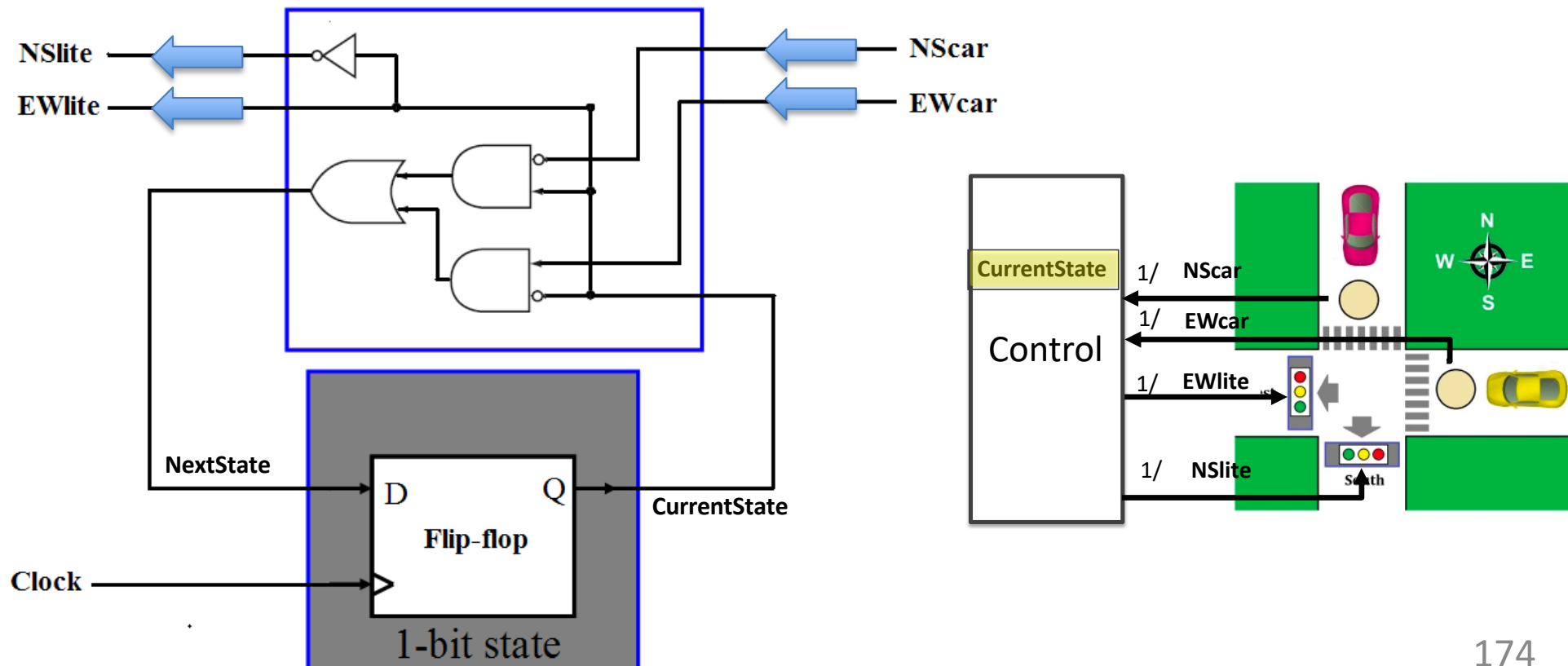
$$\text{EWlite} = \text{CurrentState}$$

Implementing Intelligent Traffic Controller

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$



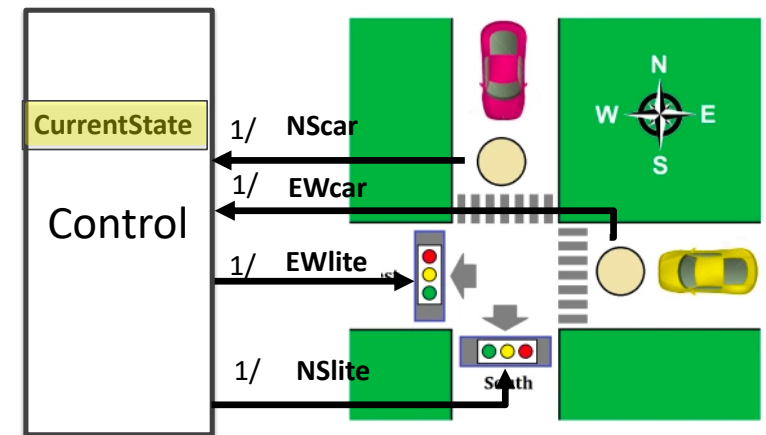
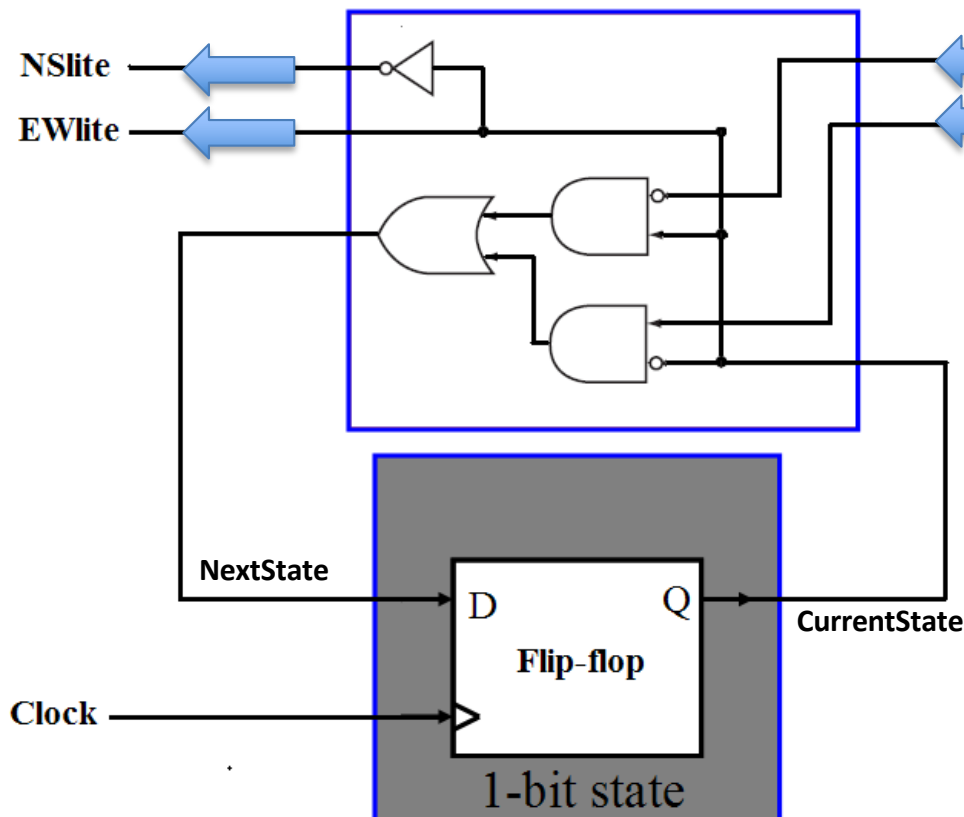
Implementing Intelligent Traffic Controller

- The state is updated at the edge of the clock cycle
- The next state is computed once every clock.

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$



Implementing the Design using Verilog

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);
    input EWCar, NSCar,clock;
    output EWLite,NSLite;
    reg state;

    initial state=0; //set initial state

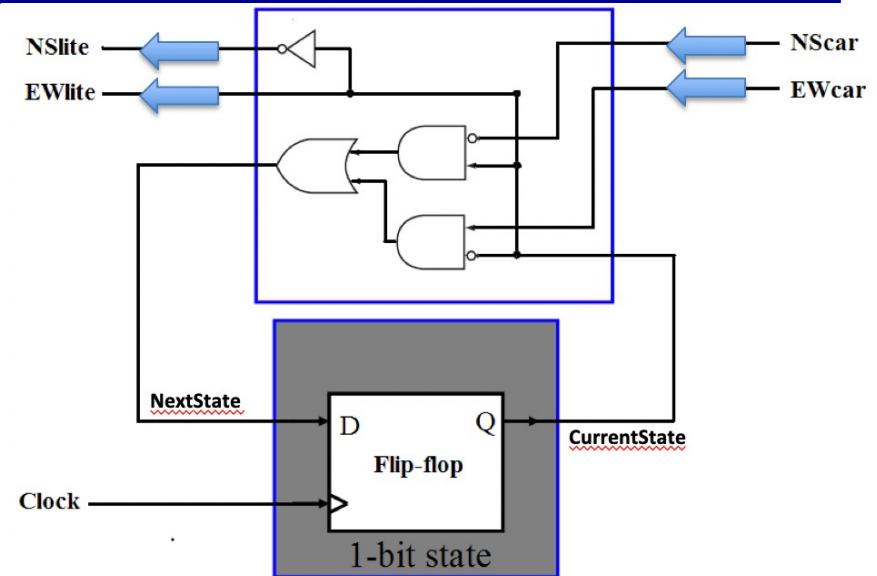
    //following two assignments set the output, which is based
    only on the state variable
    assign NSLite = ~ state; //NSLite on if state = 0;
    assign EWLite = state; //EWLite on if state = 1

    always @(posedge clock) // all state updates on a positive
    clock edge
        case (state)
            0: state = EWCar; //change state only if EWCar
            1: state = ~ NSCar; // change state only if NSCar

        endcase
endmodule
```


FSM

- More complicated FSM
 - More states → more flip-flops
 - More inputs and output
 - Inputs/outputs are more than 1 bit
 - More complicated state transition
 - E.g. extension to support Green/Red/Yellow light in Exercise A.41
- FSM is used to control processor execution
 - Chapter 4 and 5
 - Appendix C: Mapping Control to Hardware



Five Steps to Build a Finite State Machine

- There are no set procedures and diagrams. Application dependent
- Step 1: Identify inputs, outputs and states
- **Step 2: State diagram and state table**
 - **Choose a state to be the starting state when power is turned on the first time**
 - **Draw a state diagram by a graph with regards to input/outputs and state transition**
 - **List in the tables for state transition and for output → Boolean function for each tables → next-state function and output function**
- Step 3: State assignment
 - **Assign a unique binary number to each state**
 - **Rewrite the state table using the assigned number for each state**
- Step 4: Combinational logic for next state function and output function
- Step 5: Logic Implementation

CircuitVerse-Related Slides

Lab 07

- <https://circuitverse.org/>

