




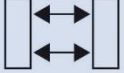
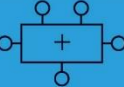
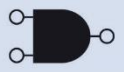
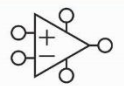


Chapter 5

Digital Design and Computer Architecture, 2nd Edition

David Money Harris and Sarah L. Harris

Chapter 5 :: Topics

- Introduction
- Arithmetic Circuits
- Number Systems
- Sequential Building Blocks
- Memory Arrays
- Logic Arrays

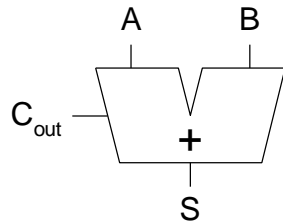
Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Introduction

- **Digital building blocks:**
 - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
 - Hierarchy of simpler components
 - Well-defined interfaces and functions
 - Regular structure easily extends to different sizes
- **Will use these building blocks in Chapter 7 to build microprocessor**

1-Bit Adders

Half Adder

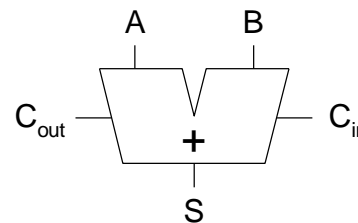


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

$$S =$$

$$C_{out} =$$

Full Adder



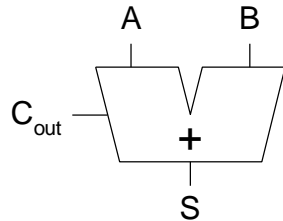
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

$$S =$$

$$C_{out} =$$

1-Bit Adders

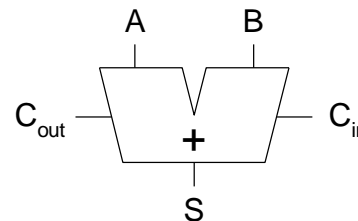
Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\begin{matrix} S \\ C_{out} \end{matrix} =$$

Full Adder

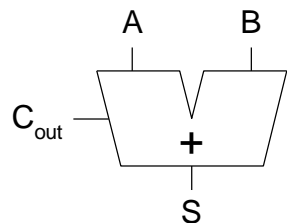


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\begin{matrix} S \\ C_{out} \end{matrix} =$$

1-Bit Adders

Half Adder

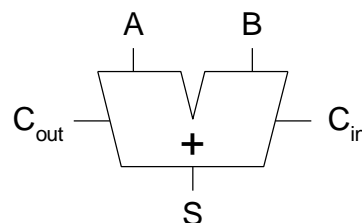


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

Full Adder



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

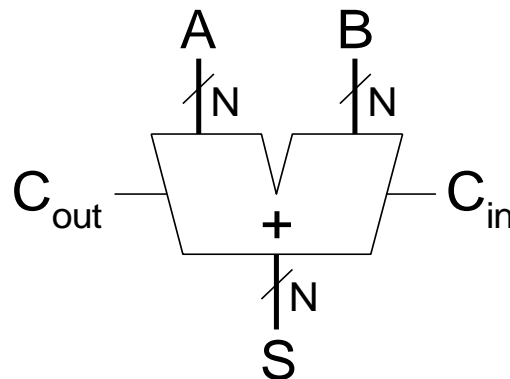
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Multibit Adders (CPAs)

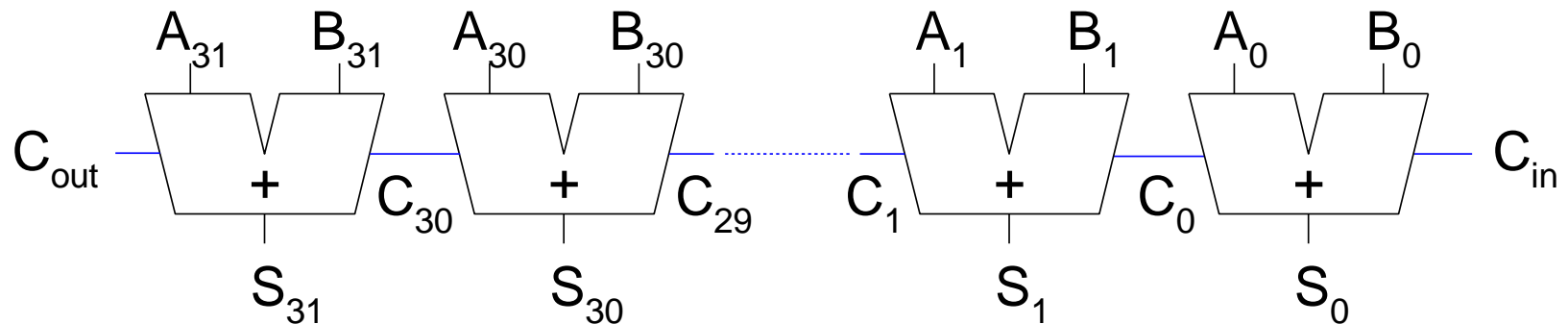
- Types of carry propagate adders (CPAs):
 - Ripple-carry (slow)
 - Carry-lookahead (fast)
 - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



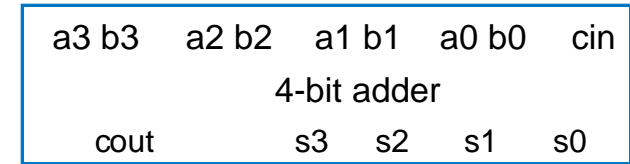
Ripple-Carry Adder Delay

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a full adder

Building a Faster Adder

- Similar to adding by hand, column by column
- Con: Slow
 - Output is not correct until the carries have rippled to the left – *critical path*
 - 4-bit carry-ripple adder has $4 \times 2 = 8$ **gate delays**
- Pro: Small
 - 4-bit carry-ripple adder has just $4 \times 5 = 20$ **gates**

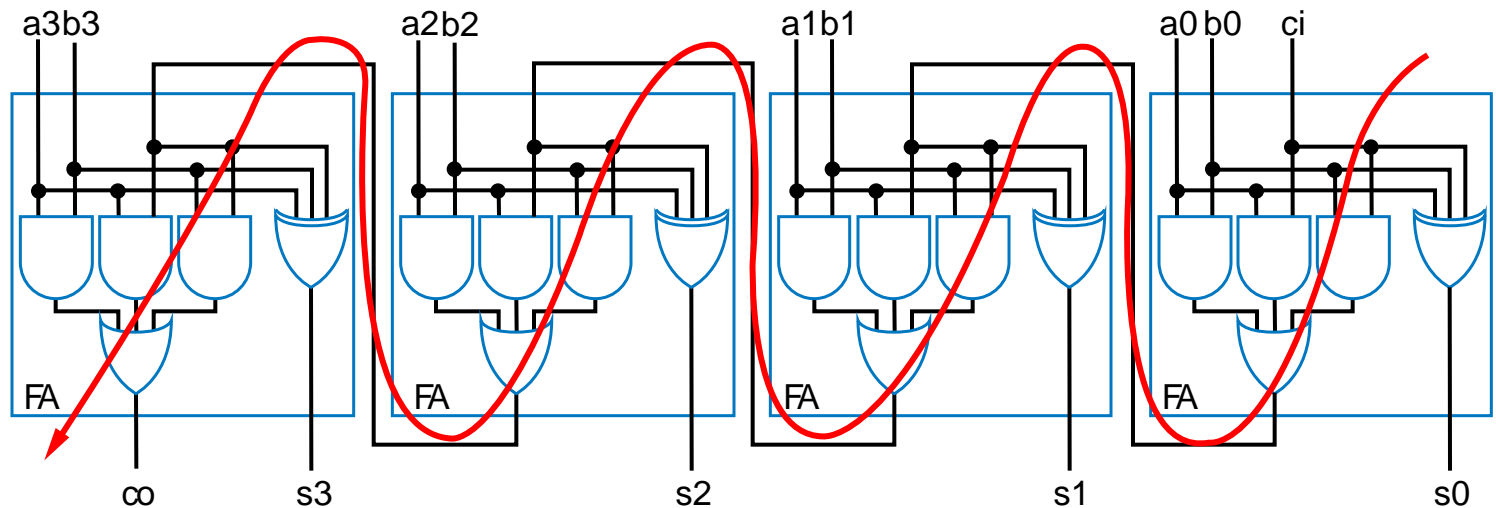


carries: **c3** **c2** **c1** cin

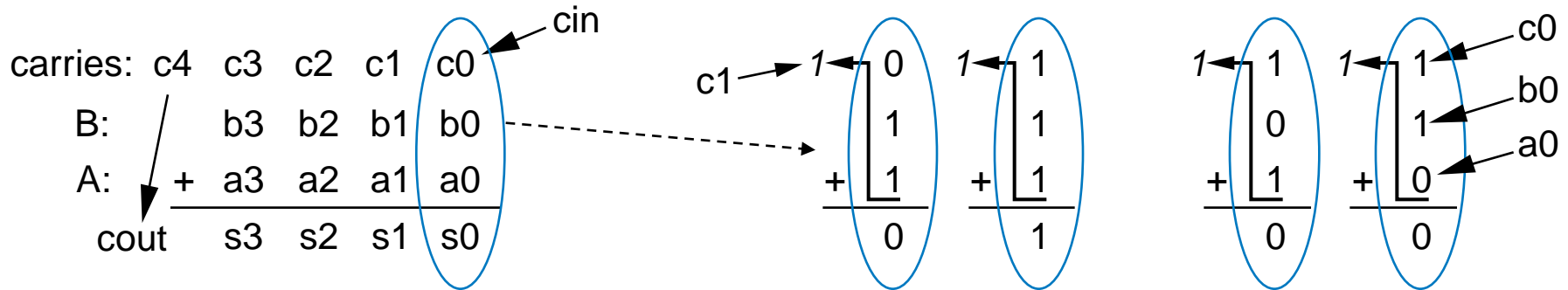
B: b3 b2 b1 b0

A: + a3 a2 a1 a0

cout s3 s2 s1 s0



Efficient Lookahead



$$c1 = a0b0 + (a0 \text{ xor } b0)c0$$

$$c2 = a1b1 + (a1 \text{ xor } b1)c1$$

$$c3 = a2b2 + (a2 \text{ xor } b2)c2$$

$$c4 = a3b3 + (a3 \text{ xor } b3)c3$$

$$c1 = G0 + P0c0$$

$$c2 = G1 + P1c1$$

$$c3 = G2 + P2c2$$

$$c4 = G3 + P3c3$$

if $a0b0 = 1$
then $c1 = 1$
(call this *G: Generate*)

if $a0 \text{ xor } b0 = 1$
then $c1 = 1$ if $c0 = 1$
(call this *P: Propagate*)

Why those names? When $a0b0=1$, we should generate a 1 for $c1$. When $a0 \text{ XOR } b0 = 1$, we should propagate the $c0$ value as the value of $c1$, meaning $c1$ should equal $c0$.

$G_i = a_i b_i$ (generate)

$P_i = a_i \text{ XOR } b_i$ (propagate)

Efficient Lookahead

$$c1 = G0 + P0c0$$



$$c2 = G1 + P1c1$$

$$c2 = G1 + P1(G0 + P0c0)$$

$$c2 = G1 + P1G0 + P1P0c0$$



$$c3 = G2 + P2c2$$

$$c3 = G2 + P2(G1 + P1G0 + P1P0c0)$$

$$c3 = G2 + P2G1 + P2P1G0 + P2P1P0c0$$

$$c4 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0$$

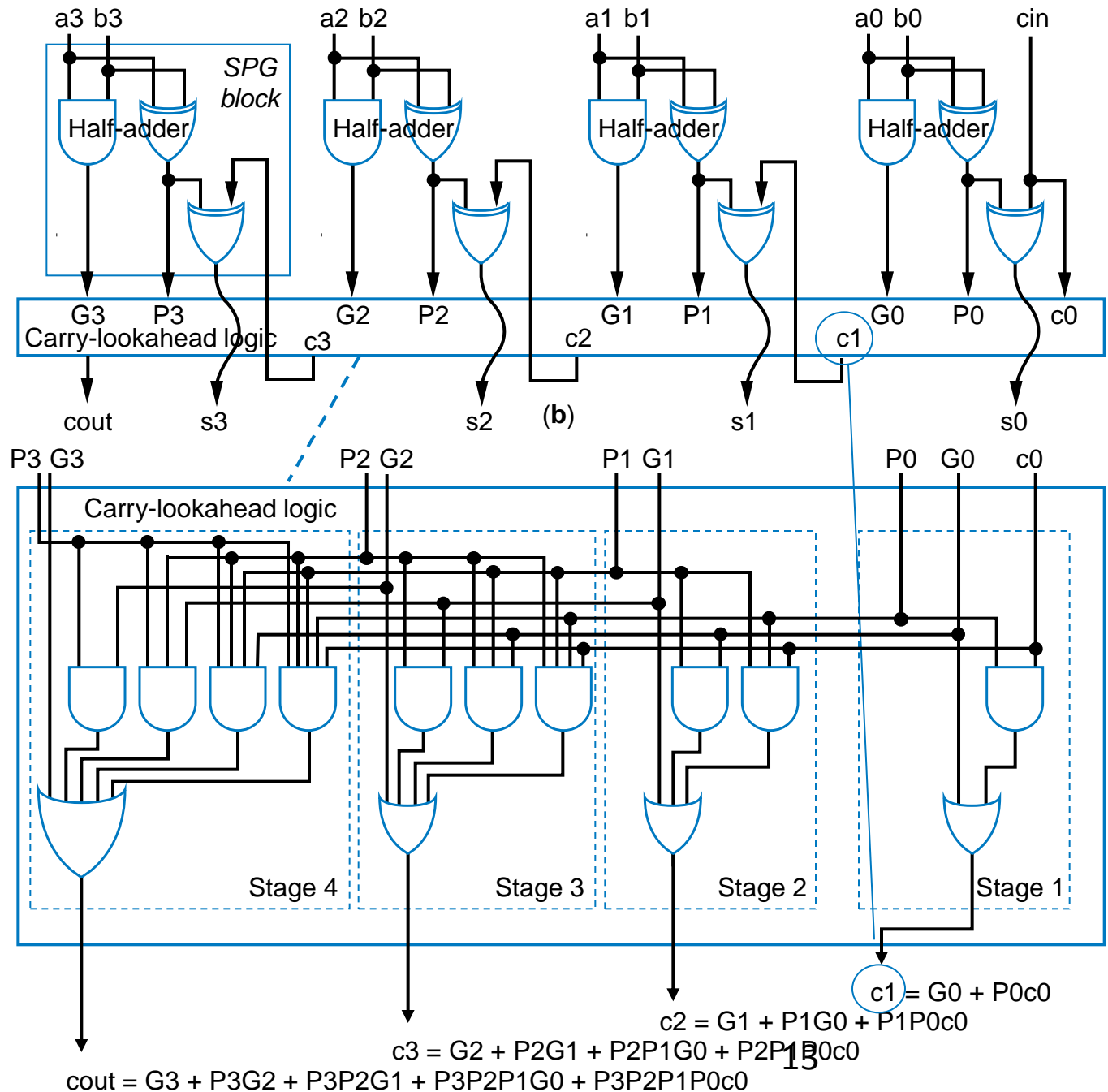
$G_i = a_i b_i$ (generate)

$P_i = a_i \text{ XOR } b_i$ (propagate)

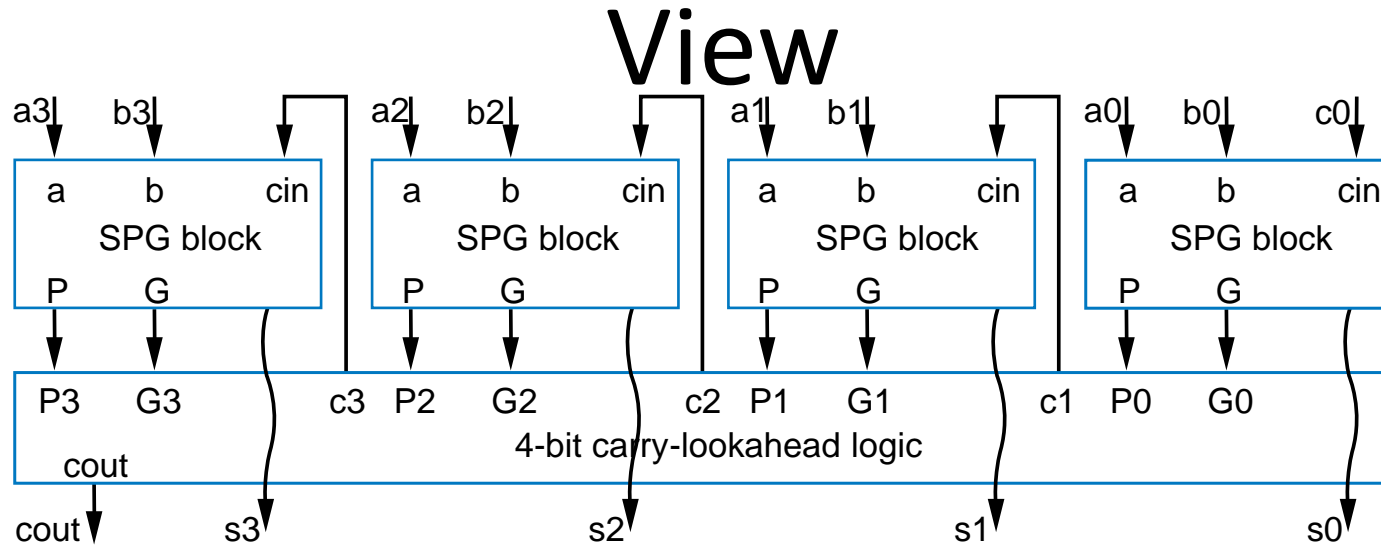
G_i/P_i function of inputs only

CLA

- Each stage:
 - HA for G and P
 - Another XOR for s
 - Call SPG block
- Create carry-lookahead logic from equations
- More efficient than naïve scheme, at expense of one extra gate delay



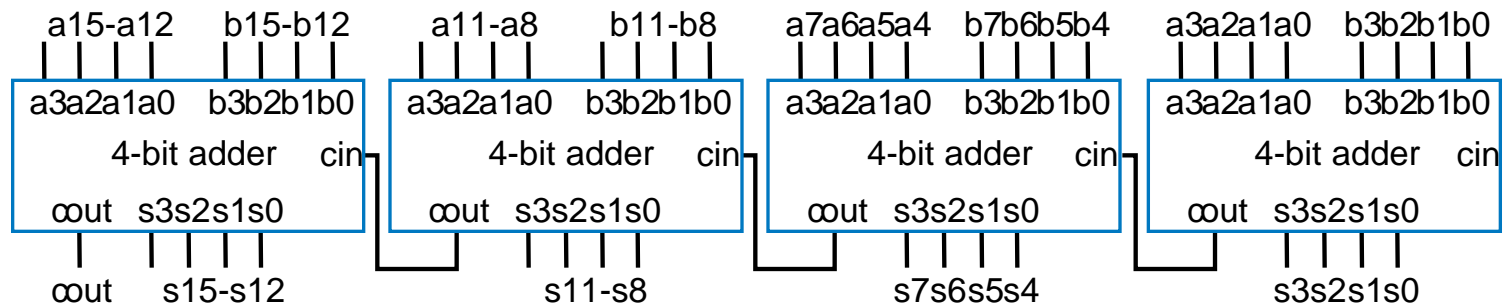
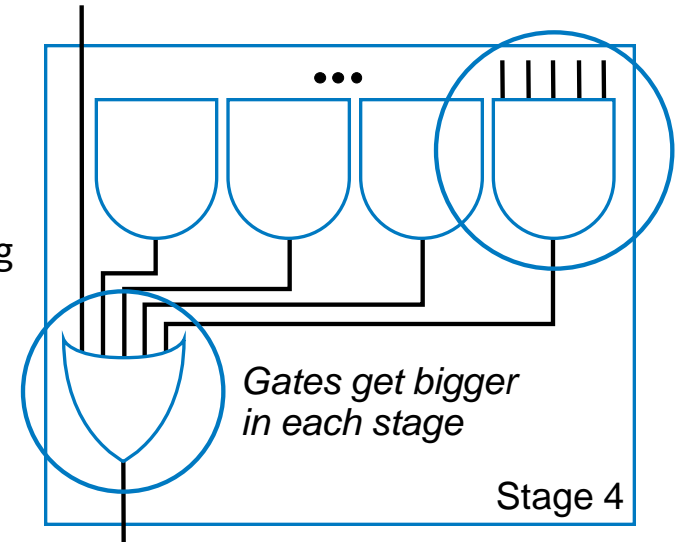
Carry-Lookahead Adder – High-Level



- Fast – only **4 gate delays**
 - Each stage has SPG block with 2 gate levels
 - Carry-lookahead logic quickly computes the carry from the propagate and generate bits using 2 gate levels inside
- Reasonable number of gates – 4-bit adder has only **26 gates**
- 4-bit adder comparison (gate delays, gates)
 - Carry-ripple: (8, 20)
 - Two-level: (2, 500)
 - CLA: (4, 26)
 - **Nice compromise**

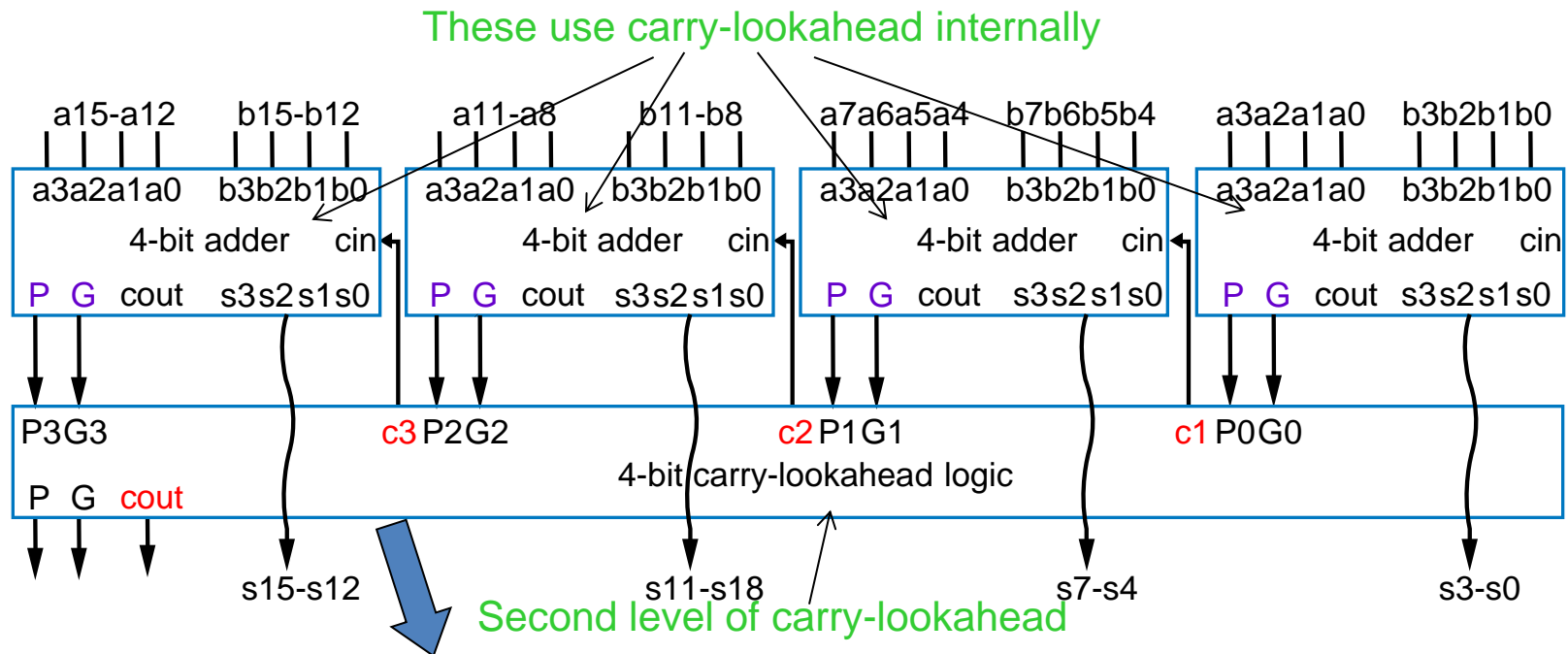
Carry-Lookahead Adder – 32-bit?

- Problem: Gates get bigger in each stage
 - 4th stage has 5-input gates
 - 32nd stage would have 33-input gates
 - Too many inputs for one gate
 - Would require building from smaller gates, meaning more levels (slower), more gates (bigger)
- One solution: Connect 4-bit CLA adders in carry-ripple manner
 - Ex: 16-bit adder: $4 + 4 + 4 + 4 = 16$ gate delays.
 - Can we do better?



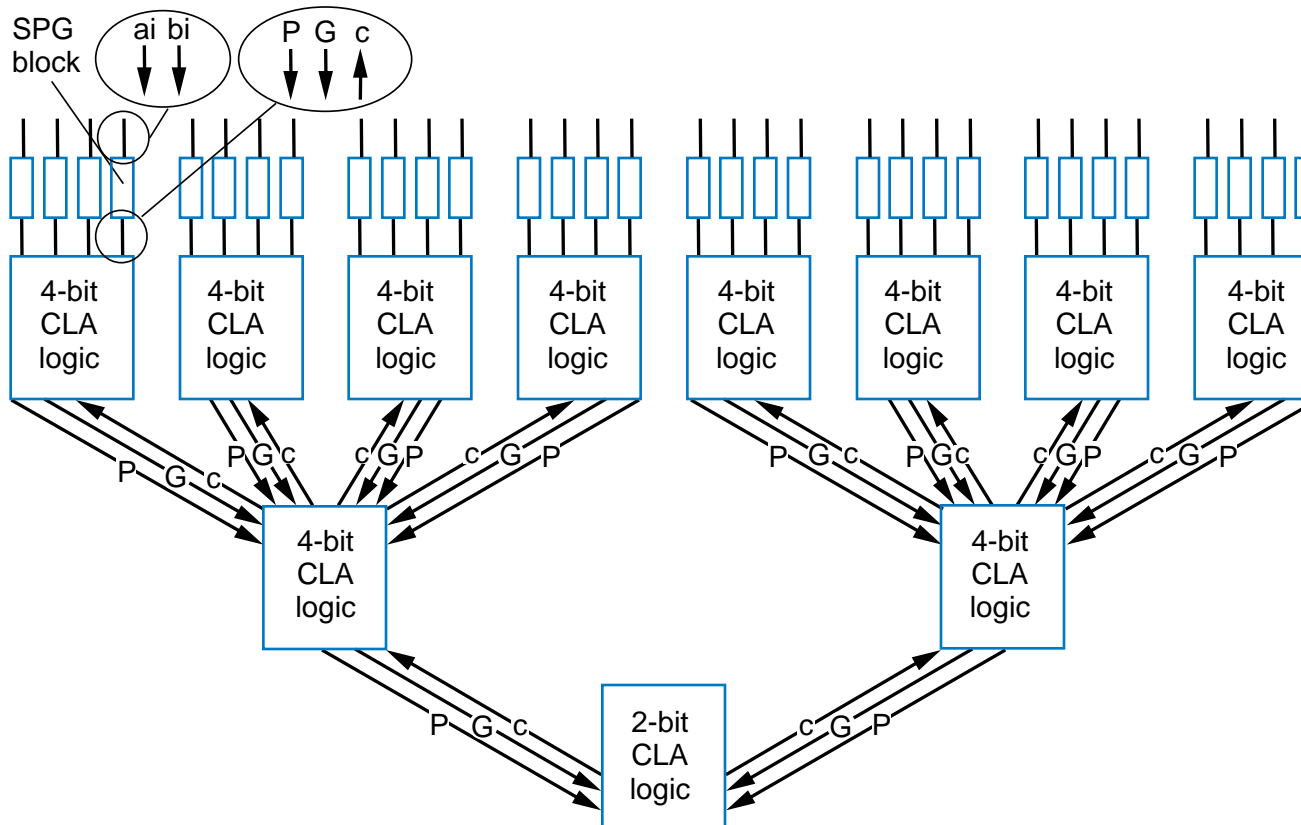
Hierarchical Carry-Lookahead Adders

- Better solution – Rather than rippling the carries, just *repeat* the carry-lookahead concept
 - Requires minor modification of 4-bit CLA adder to **output P and G**



Hierarchical Carry-Lookahead Adders

- Hierarchical CLA concept can be applied for larger adders
- 32-bit hierarchical CLA:
 - Only about 8 gate delays (2 for SPG block, then 2 per CLA level)
 - Only about 14 gates in each 4-bit CLA logic block

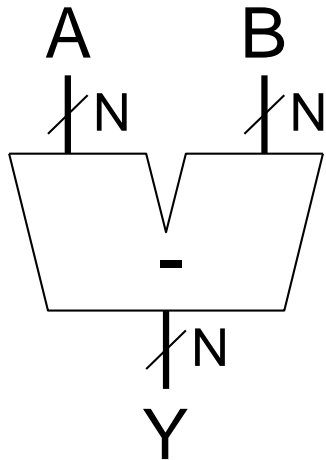


Q: How many gate delays for 64-bit hierarchical CLA, using 4-bit CLA logic?

A: 16 CLA-logic blocks in 1st level, 4 in 2nd, 1 in 3rd -- so still just 8 gate delays (2 for SPG, and 2+2+2 for CLA logic). *CLA is a very efficient method.*

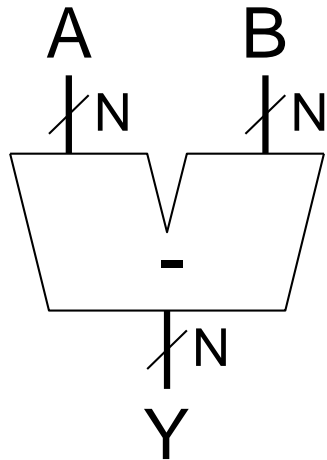
Subtractor

Symbol

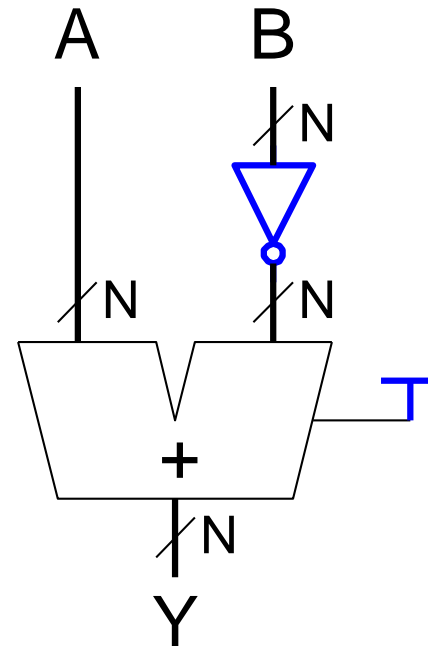


Subtractor

Symbol

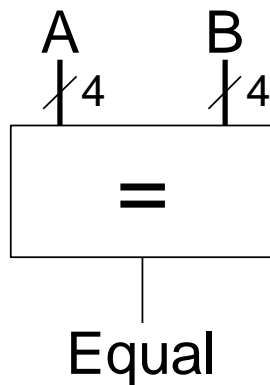


Implementation



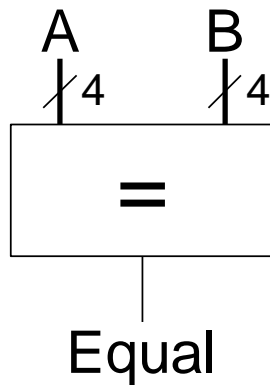
Comparator: Equality

Symbol

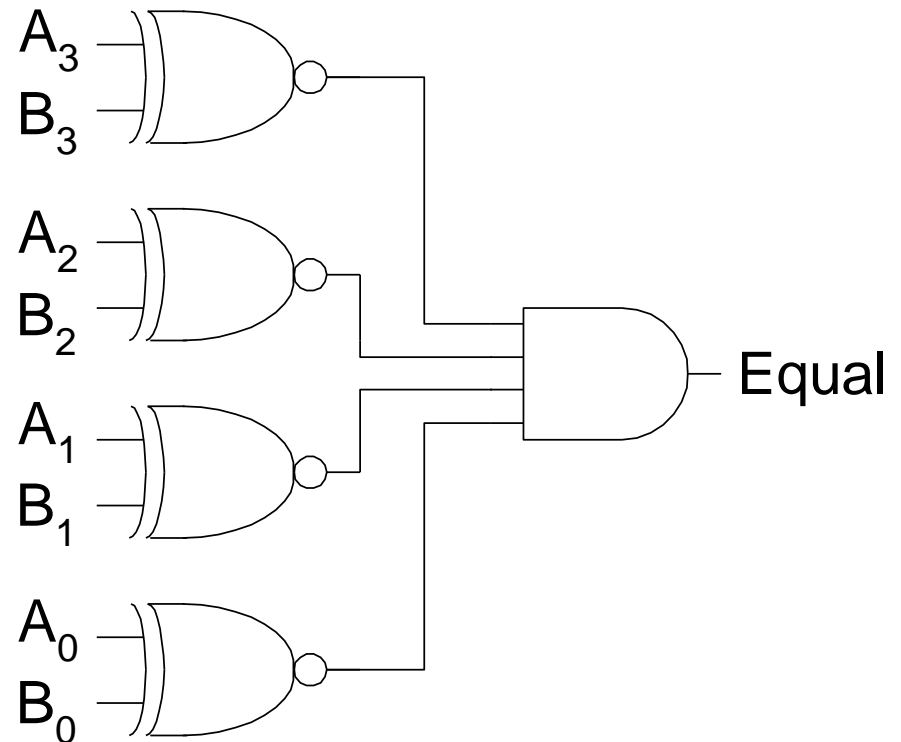


Comparator: Equality

Symbol



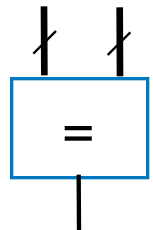
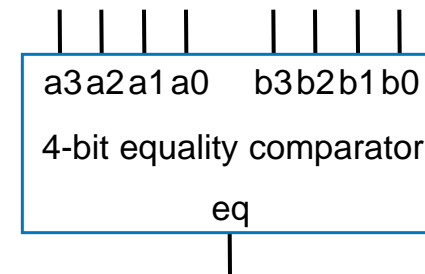
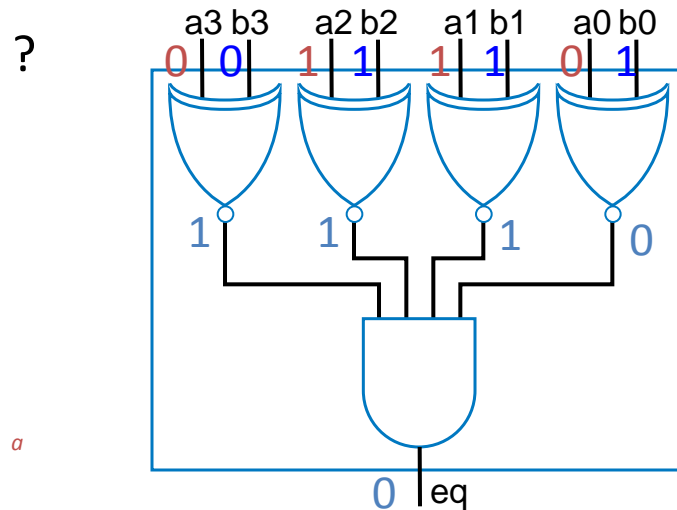
Implementation



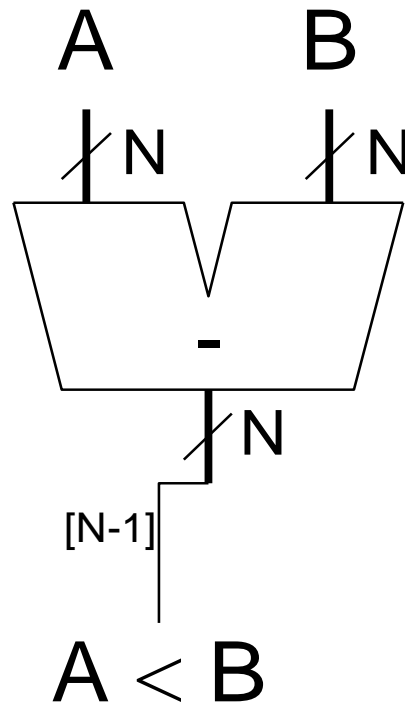
Comparators

- ***N-bit equality comparator***: Outputs 1 if two N-bit numbers are equal
 - 4-bit equality comparator with inputs A and B
 - a_3 must equal b_3 , $a_2 = b_2$, $a_1 = b_1$, $a_0 = b_0$
 - Two bits are equal if both 1, or both 0
 - $eq = (a_3b_3 + a_3'b_3') * (a_2b_2 + a_2'b_2') * (a_1b_1 + a_1'b_1') * (a_0b_0 + a_0'b_0')$
 - Note that function inside parentheses is XNOR
 - $eq = (a_3 \text{ xnor } b_3) * (a_2 \text{ xnor } b_2) * (a_1 \text{ xnor } b_1) * (a_0 \text{ xnor } b_0)$

0110 = 0111 ?



Comparator: Less Than



Magnitude Comparator

- *N-bit magnitude comparator:*

Two N-bit inputs A and B, outputs whether $A > B$, $A = B$, or $A < B$, for

- How design? Consider comparing by hand.
- First compare a_3 and b_3 . If equal, compare a_2 and b_2 . And so on.
- Stop if comparison not equal (the two bits are 0 and 1, or 1 and 0)—whichever of A or B has the 1 is thus greater. If never see unequal bit pair, then $A = B$.

A=1011 B=1001

1011 1001 Equal

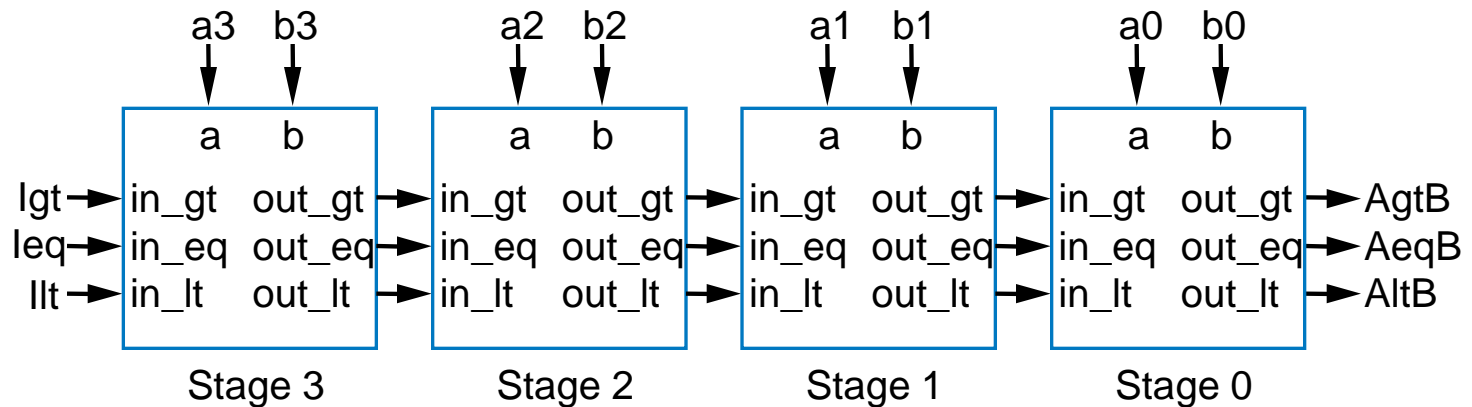
1011 1001 Equal

1011 1001 Not equal

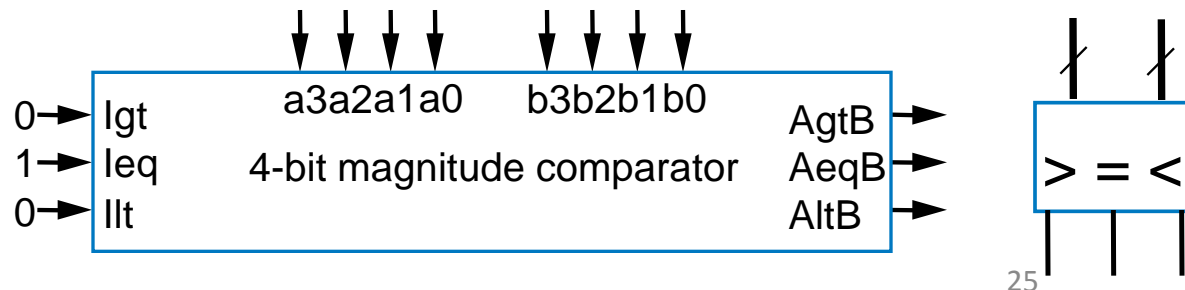
So $A > B$

Magnitude Comparator

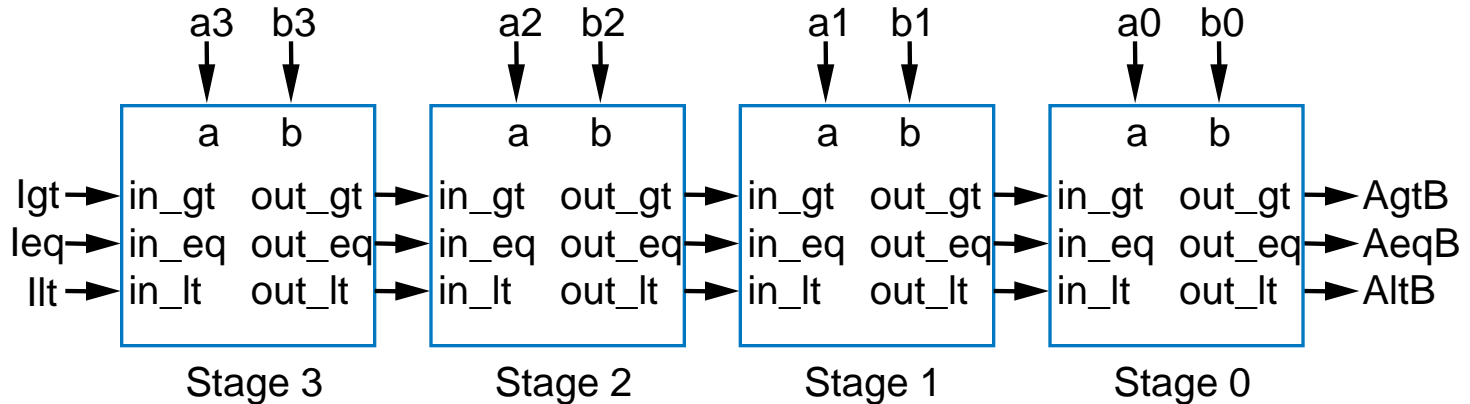
- By-hand example leads to idea for design
 - Start at left, compare each bit pair, pass results to the right
 - Each bit pair called a *stage*
 - Each stage has 3 inputs taking results of higher stage, outputs new results to lower stage



How design each stage?



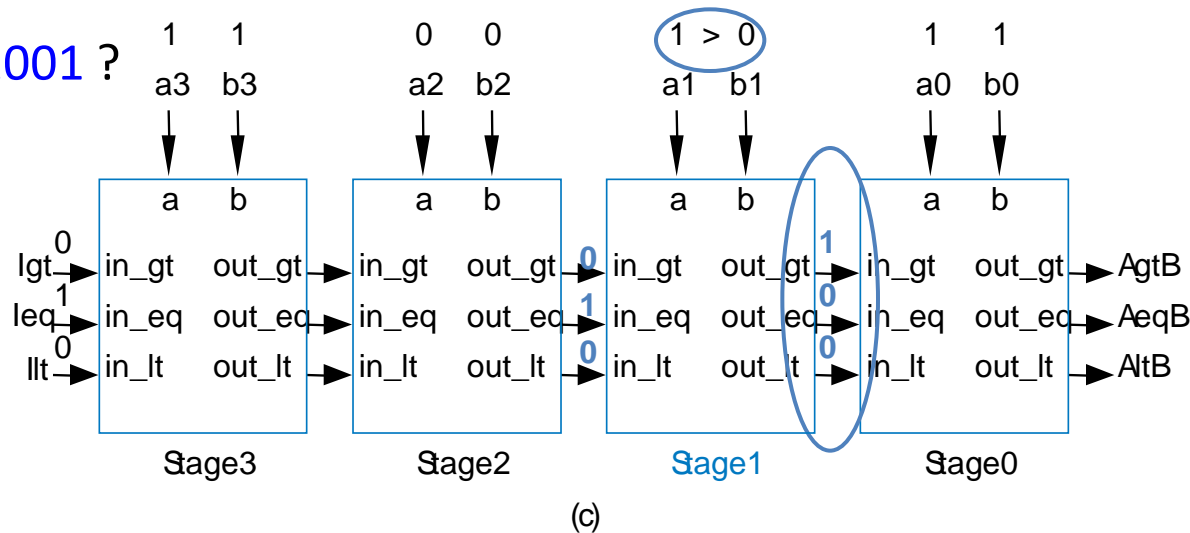
Magnitude Comparator



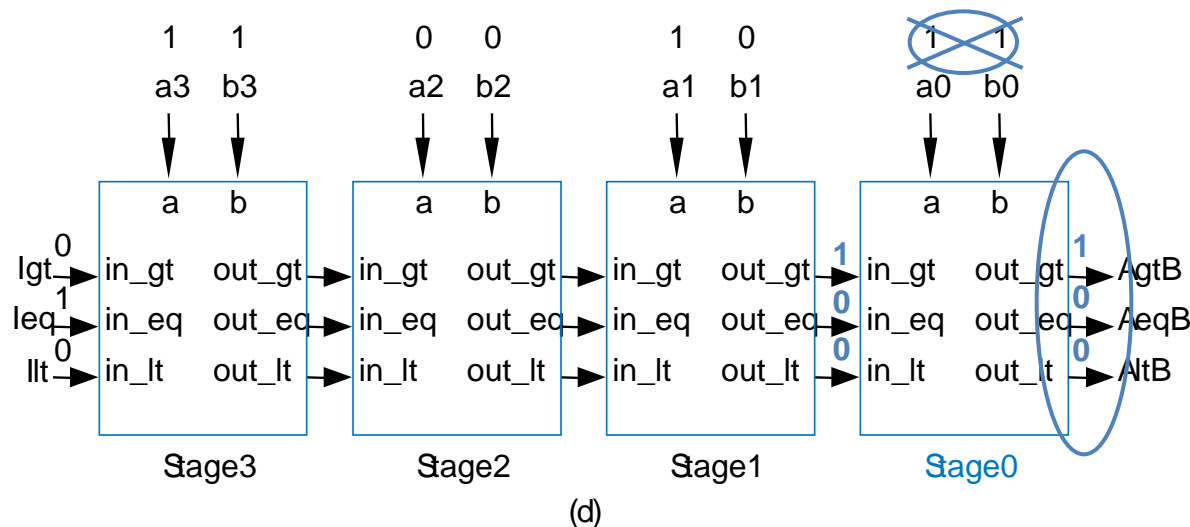
- Each stage:
 - $out_gt = in_gt + (in_eq * a * b')$
 - $A > B$ if already determined in higher stage, or if higher stages equal but in this stage $a=1$ and $b=0$
 - $out_lt = in_lt + (in_eq * a' * b)$
 - $A < B$ if already determined in higher stage, or if higher stages equal but in this stage $a=0$ and $b=1$
 - $out_eq = in_eq * (a \text{ XNOR } b)$
 - $A=B$ (so far) if already determined in higher stage and in this stage $a=b$ too
 - Simple circuit inside each stage, just a few gates (not shown)

Magnitude Comparator

1011 = 1001 ?

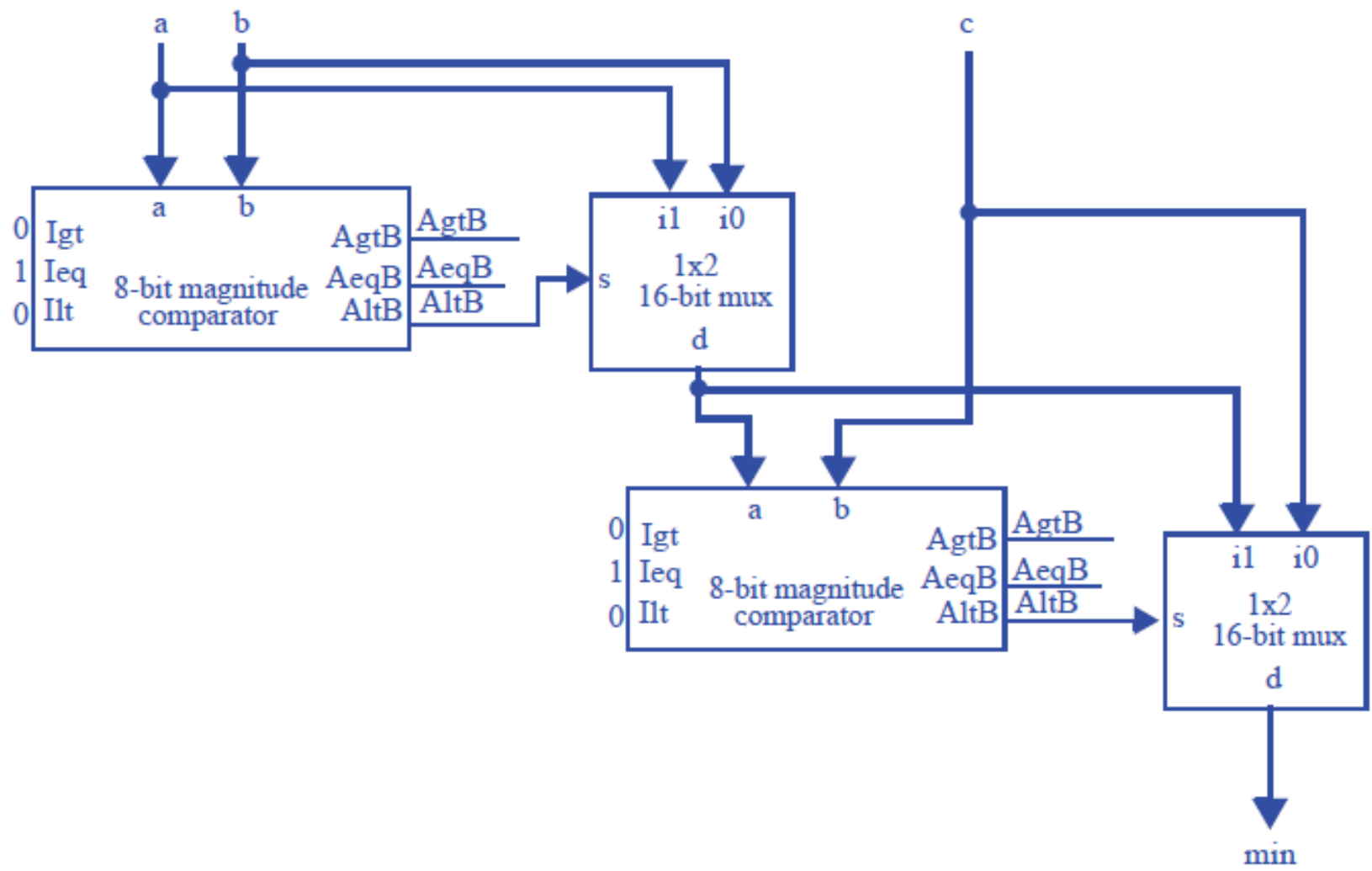


- Final answer appears on the right
- Takes time for answer to “ripple” from left to right
- Thus called “carry-ripple style” after the carry-ripple adder
 - Even though there’s no “carry” involved



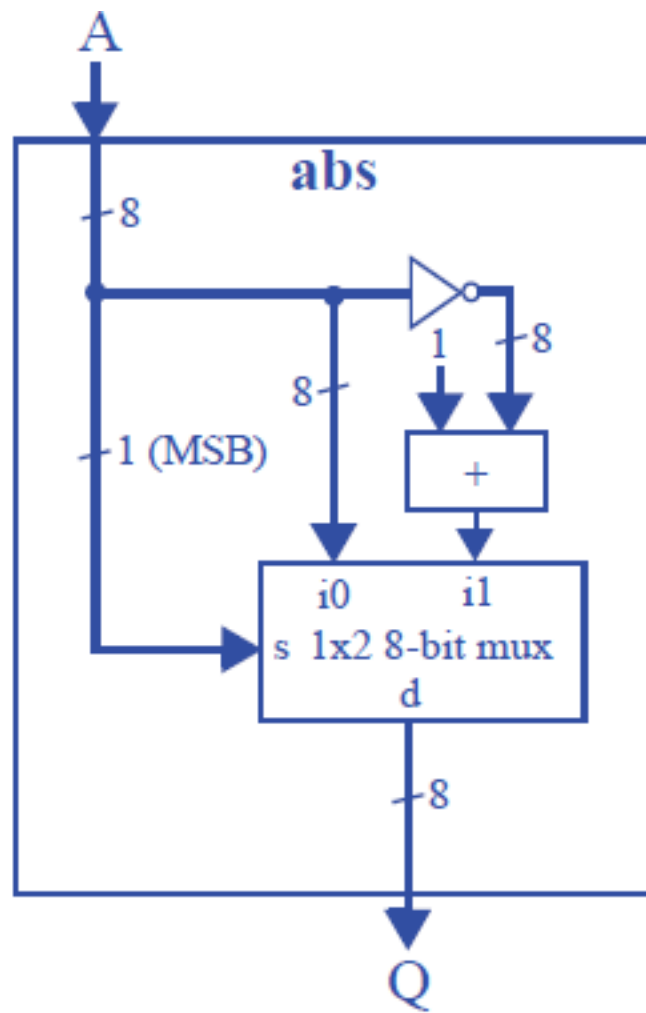
Exercise

Use magnitude comparators and logic to design a circuit that computes the minimum of three 8 bit numbers

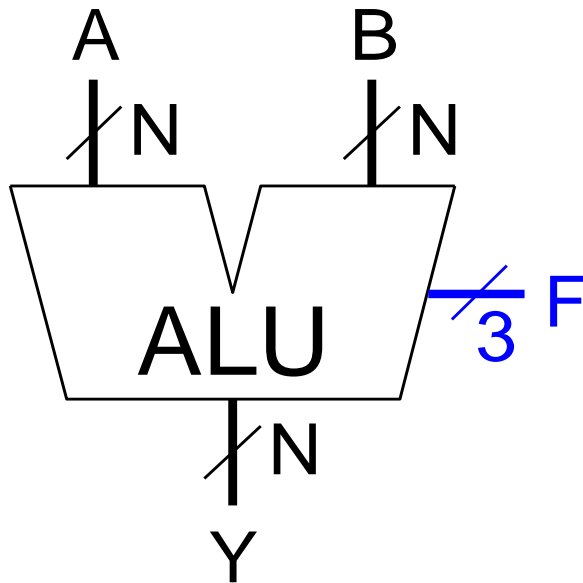


Exercise

- 4.38 Create an absolute value component `abs` with an 8-bit input `A` that is a signed binary number, and an 8-bit output `Q` that is unsigned and that is the absolute value of `A`. So if the input is 00001111 (+15) then the output is also 00001111 (+15), but if the input is 11111111 (-1) then the output is 00000001 (+1).

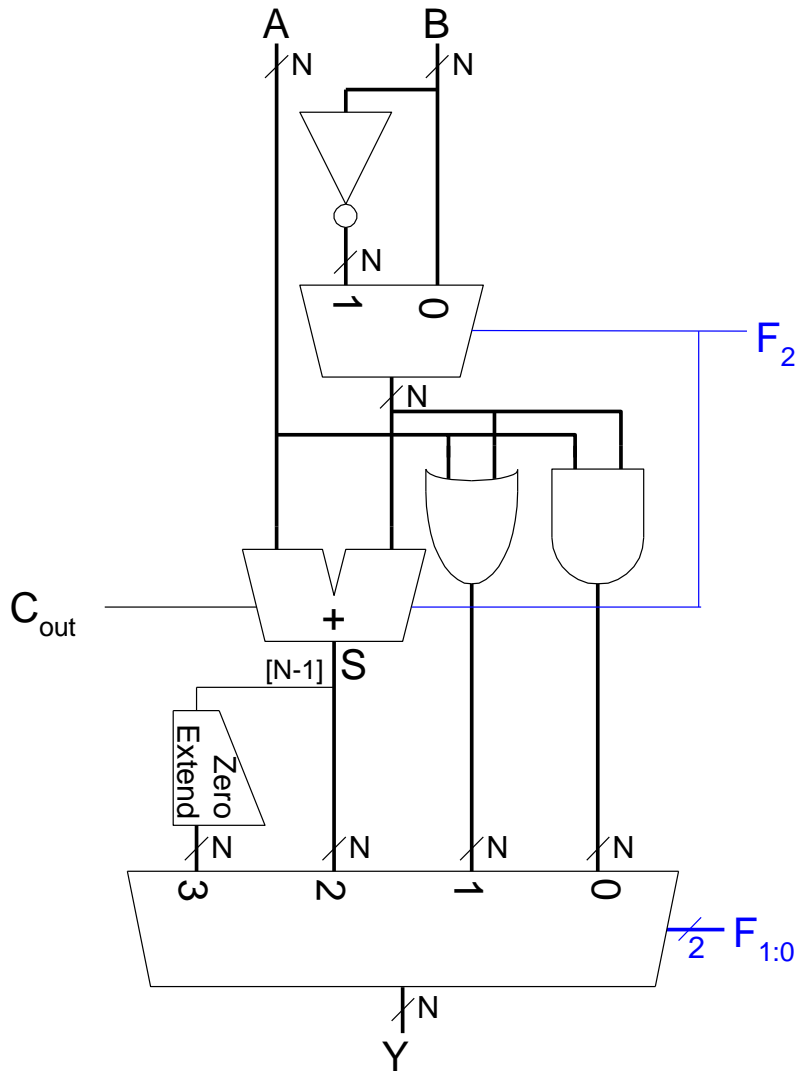


Arithmetic Logic Unit (ALU)



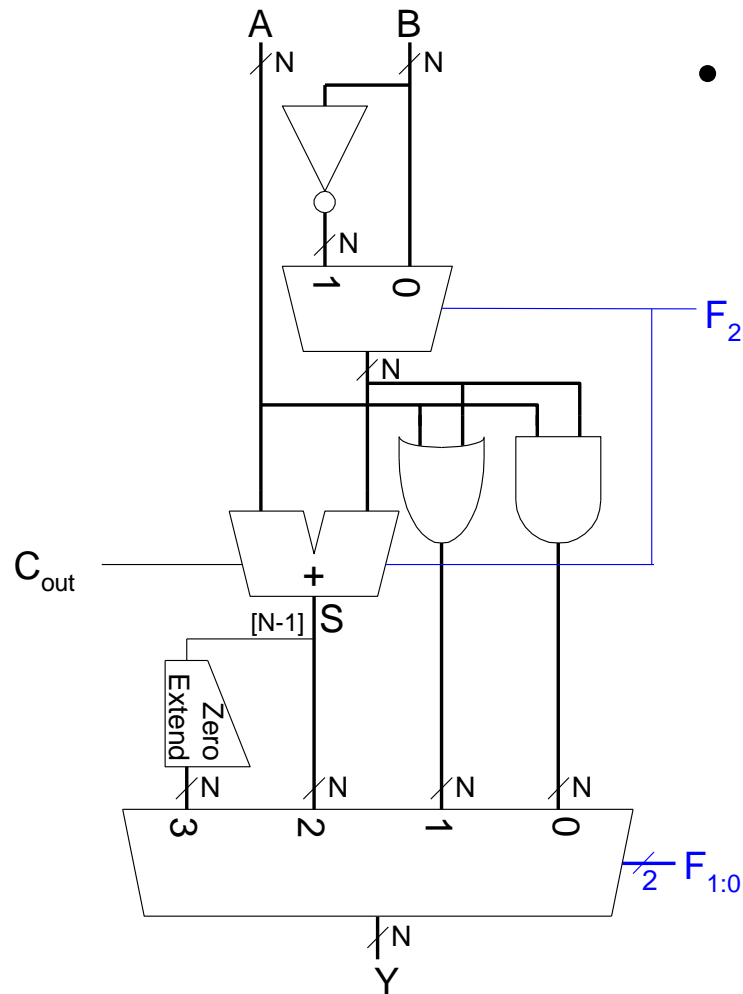
$F_{2:0}$	Function
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

ALU Design



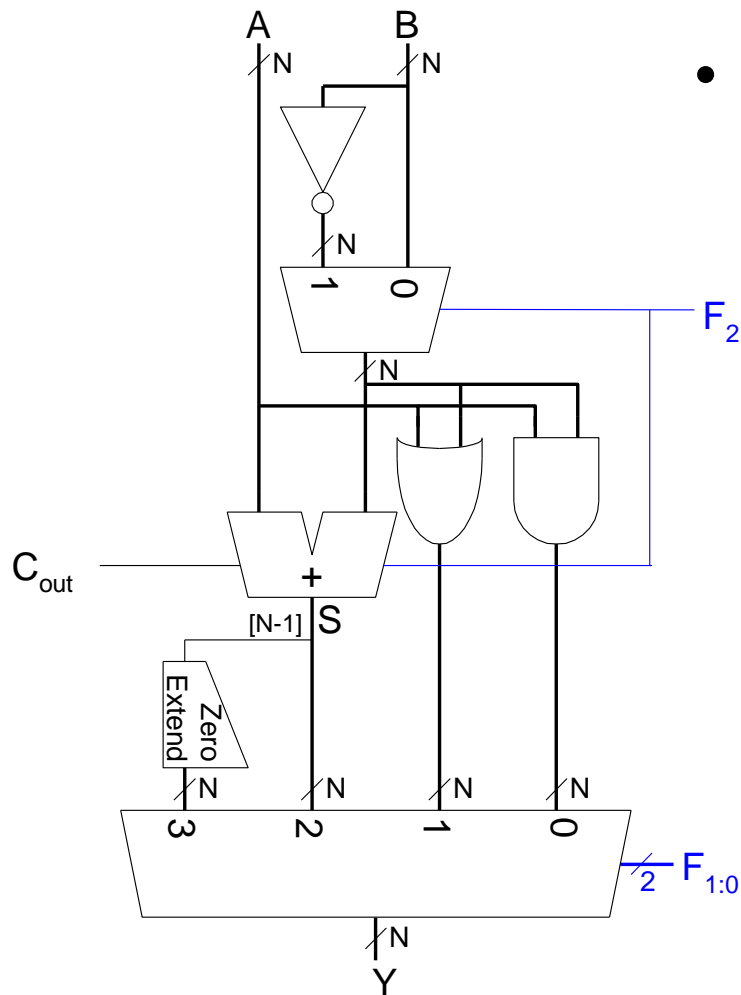
$F_{2:0}$	Function
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$
 - $A < B$, so Y should be 32-bit representation of 1 (0x00000001)
 - $F_{2:0} = 111$
 - $F_2 = 1$ (adder acts as subtractor), so $25 - 32 = -7$
 - -7 has 1 in the most significant bit ($S_{31} = 1$)
 - $F_{1:0} = 11$ multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

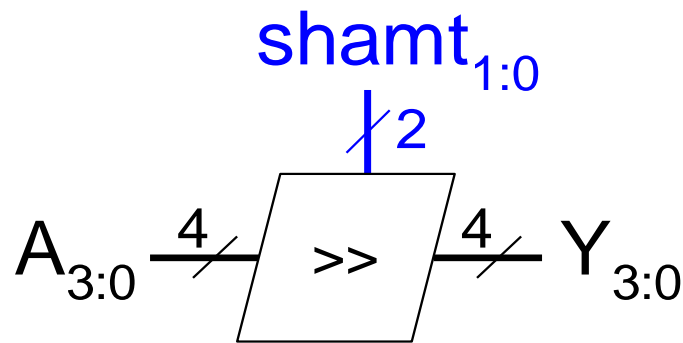
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
 - Ex: 11001 >> 2 =
 - Ex: 11001 << 2 =
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
 - Ex: 11001 >>> 2 =
 - Ex: 11001 <<< 2 =
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
 - Ex: 11001 ROR 2 =
 - Ex: 11001 ROL 2 =

Shifters

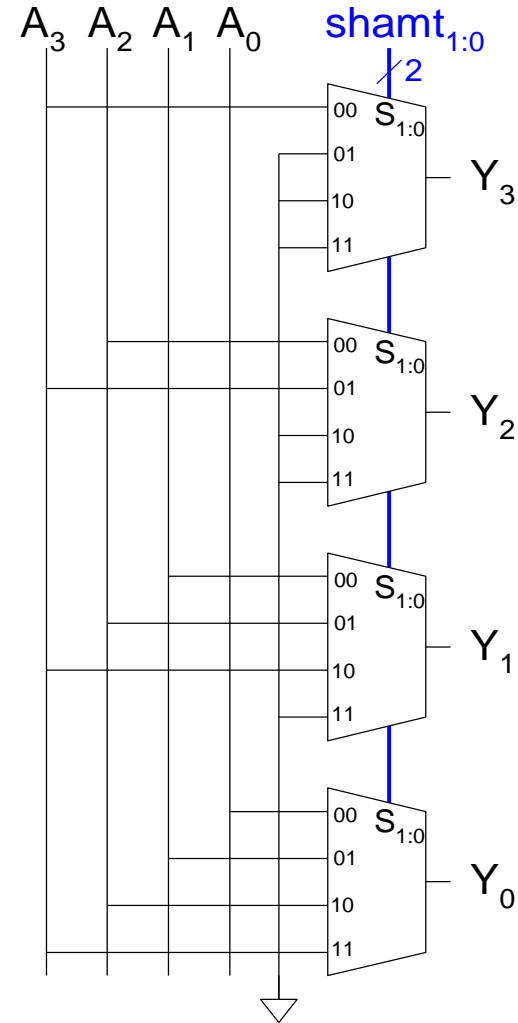
- **Logical shifter:**
 - Ex: $11001 \gg 2 = 00110$
 - Ex: $11001 \ll 2 = 00100$
- **Arithmetic shifter:**
 - Ex: $11001 \ggg 2 = 11110$
 - Ex: $11001 \lll 2 = 00100$
- **Rotator:**
 - Ex: $11001 \text{ ROR } 2 = 01110$
 - Ex: $11001 \text{ ROL } 2 = 00111$

Shifter Design

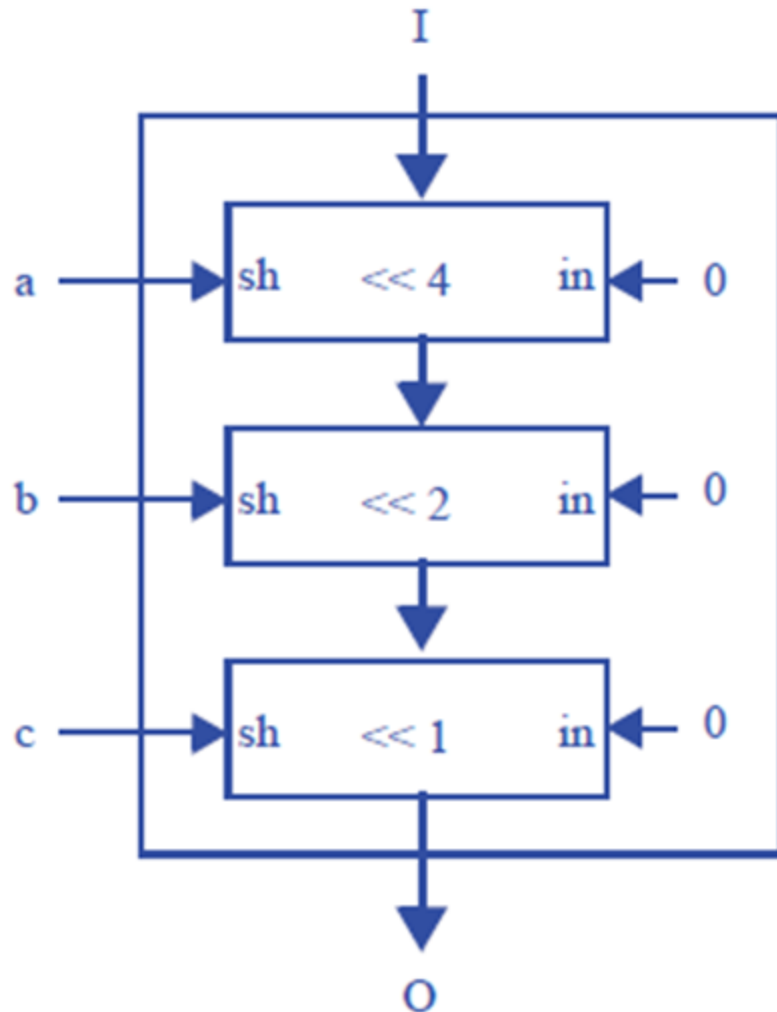


How to extend it to N-bit shifter?

Need $2^N \times 1$ MUX for each bit.
Too expensive!



Barrel Shifter



Shifts either 4 bits or 0 bits based on input a

Shifts either 2 bits or 0 bits based on input b

Shifts either 1 bits or 0 bits based on input c

Shifters as Multipliers, Dividers

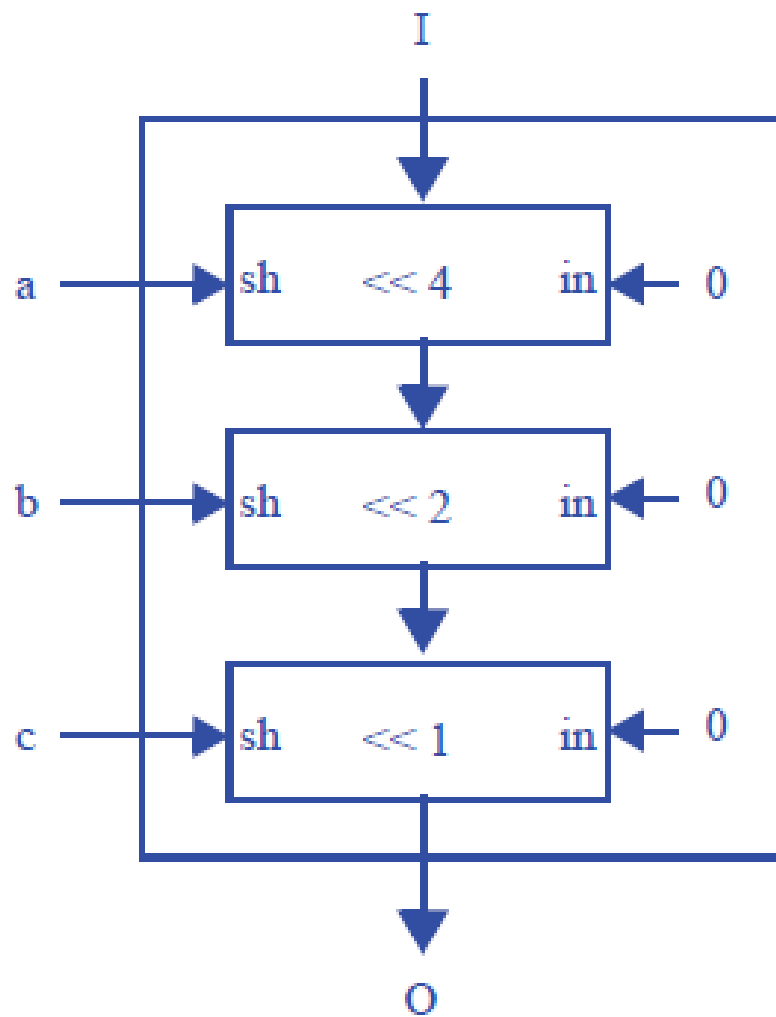
- $A \ll N = ?$
- $A \gg N = ?$

Shifters as Multipliers, Dividers

- $A \ll N = A \times 2^N$
 - **Example:** $00001 \ll 2 = 00100$ ($1 \times 2^2 = 4$)
 - **Example:** $11101 \ll 2 = 10100$ ($-3 \times 2^2 = -12$)
- $A \gg N = A \div 2^N$
 - **Example:** $01000 \gg 2 = 00010$ ($8 \div 2^2 = 2$)
 - **Example:** $10000 \gg 2 = 11100$ ($-16 \div 2^2 = -4$)

Exercise

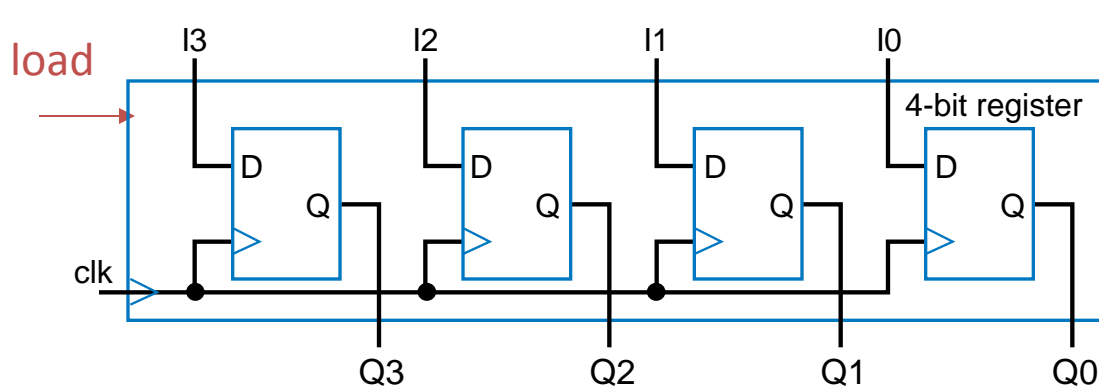
- 4.46 Design a special multiplier circuit that can multiply its 16-bit input by 1, 2, 4, 8, or 16, or 32, specified by three inputs a, b, c (abc=000 means no multiply, abc=001 means multiply by 2, abc=010 means by 4, abc=011 means by 8, abc=100 means by 16, abc=101 means by 32). *Hint: A simple solution consists entirely of just one copy of a component from this chapter. (Component use problem).*



Barrel shifter component

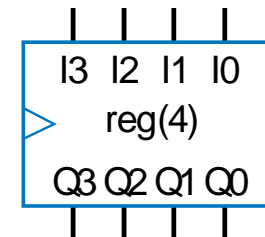
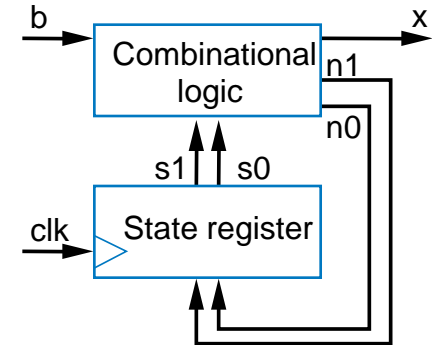
Registers

- ***N-bit register***: Stores N bits, N is the *width*
 - Common widths: 8, 16, 32
 - Storing data into register: *Loading*
 - Opposite of storing: *Reading* (does not alter contents)
- Basic register of Ch 3: Loaded every cycle
 - Useful for implementing FSM—stores encoded state



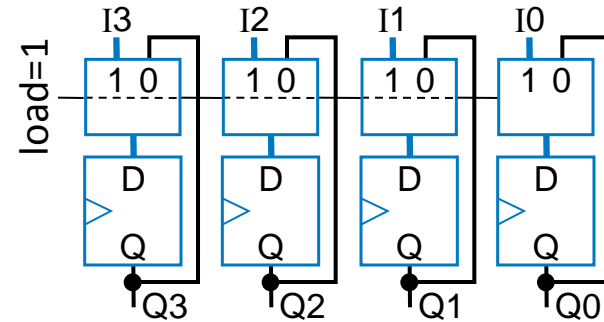
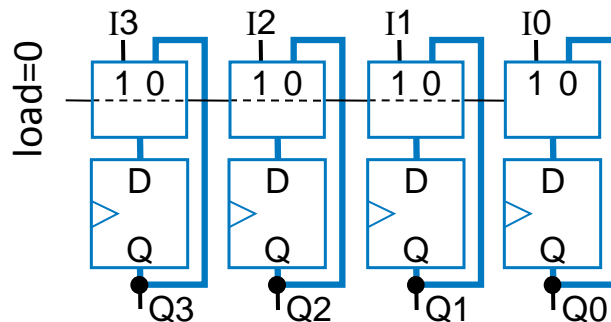
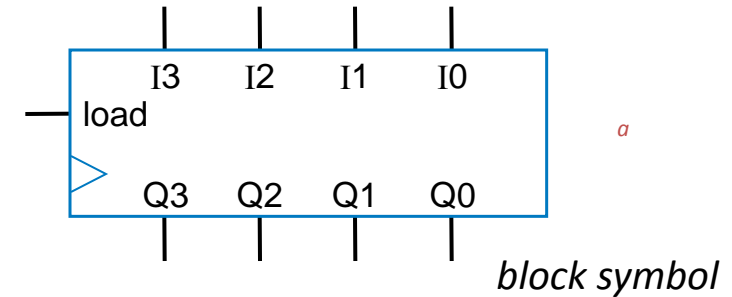
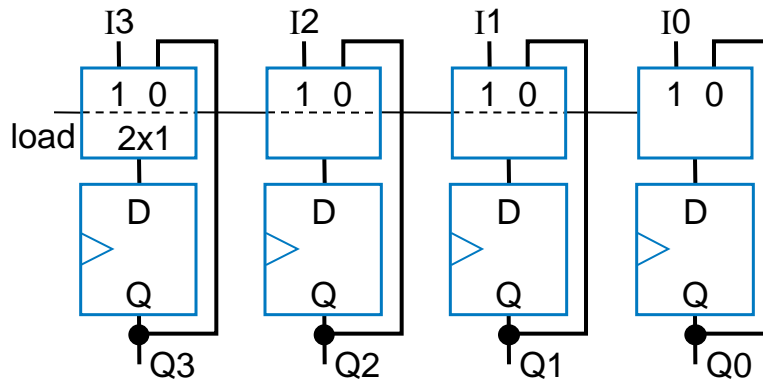
Basic register loads on every clock cycle

How extend to only load on certain cycles?



Register with Parallel Load

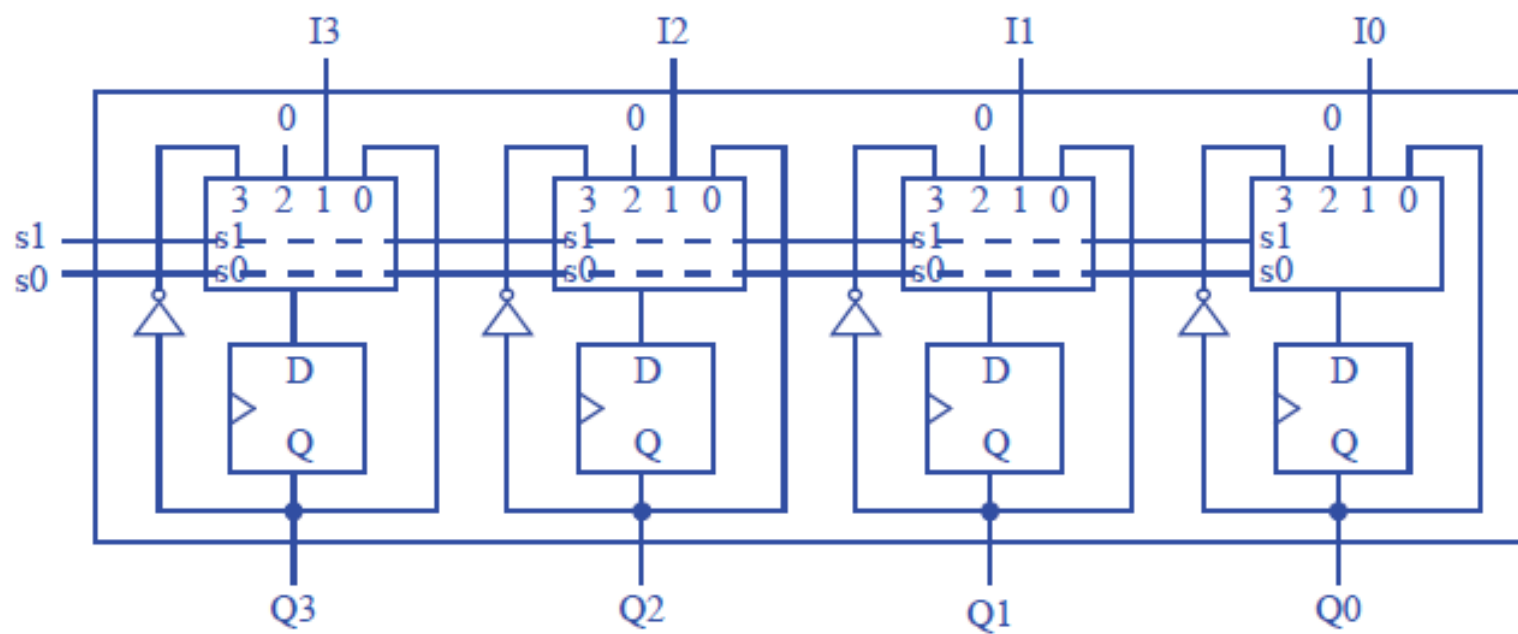
- Add 2x1 mux to front of each flip-flop
- Register's *load* input selects mux input to pass
 - load=0: existing flip-flop value; load=1: new input value



Exercise

Design a 4-bit register with 2 control inputs s_1 and s_0 , 4 data inputs I_3 , I_2 , I_1 and I_0 , and 4 data outputs Q_3 , Q_2 , Q_1 and Q_0 . When $s_1s_0=00$, the register maintains its value. When $s_1s_0=01$, the register loads $I_3..I_0$. When $s_1s_0=10$, the register clears itself to 0000. When $s_1s_0=11$, the register complements itself, so for example 0000 would become 1111, and 1010 would become 0101. (*Component design problem*).

```
module register(  
    input logic clk,  
    input logic [1:0] select,  
    input logic [3:0] In,  
    output logic [3:0] Q  
);  
  
    always_ff @(posedge clk)  
        case(select)  
            2'b00: Q <=Q;  
            2'b01: Q <=In;  
            2'b10: Q <=4'b0000;  
            2'b11: Q <=~Q;  
        endcase  
endmodule
```



Exercise

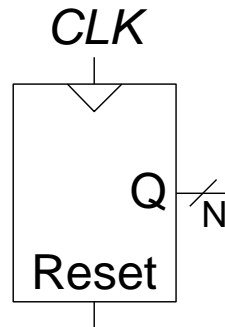
- 4.5 Design an 8-bit register with 2 control inputs s_1 and s_0 , 8 data inputs $I_7..I_0$, and 8 data outputs $Q_7..Q_0$. $s_1s_0=00$ means maintain the present value, $s_1s_0=01$ means load, and $s_1s_0=10$ means clear. $s_1s_0=11$ means to swap the high nibble with the low nibble (a nibble is 4 bits), so 11110000 would become 00001111, and 11000101 would become 01011100. (*Component design problem*).



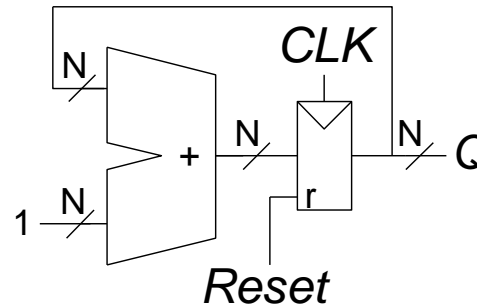
Counters

- Increments on each clock edge
- Used to cycle through numbers. For example,
 - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- Example uses:
 - Digital clock displays
 - Program counter: keeps track of current instruction executing

Symbol

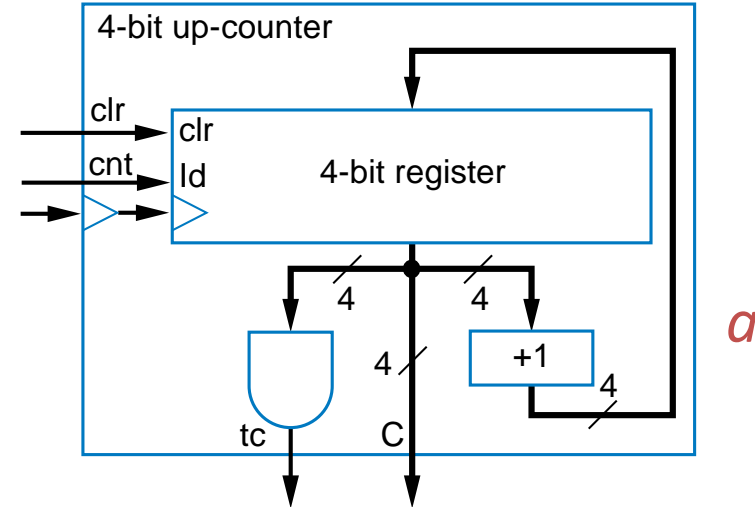
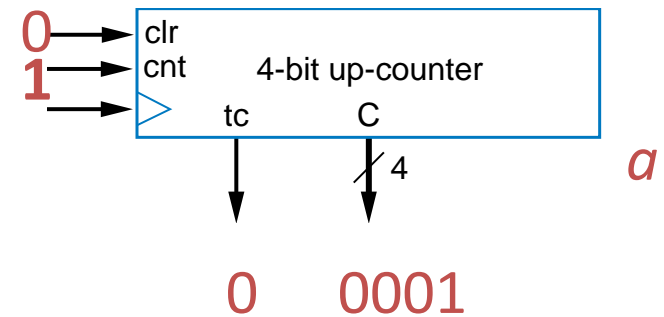


Implementation



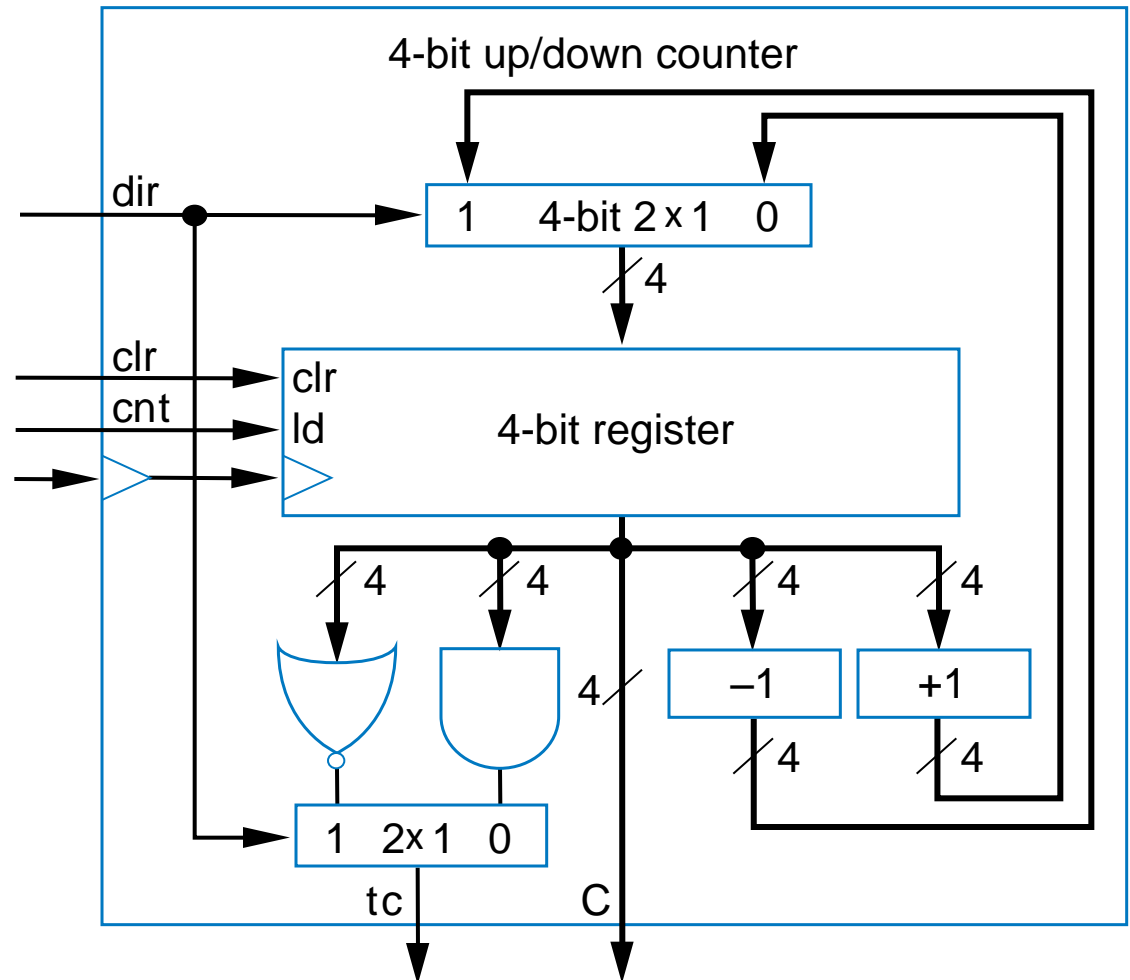
Counters and Timers

- ***N-bit up-counter***: N-bit register that can increment (add 1) to its own value on each clock cycle
 - 0000, 0001, 0010, 0011, ..., 1110, 1111, 0000
 - Count “rolls over” from 1111 to 0000
 - Terminal (last) count, tc, equals 1 during value just before rollover
- Internal design
 - Register, incrementer, and N-input AND gate to detect terminal count



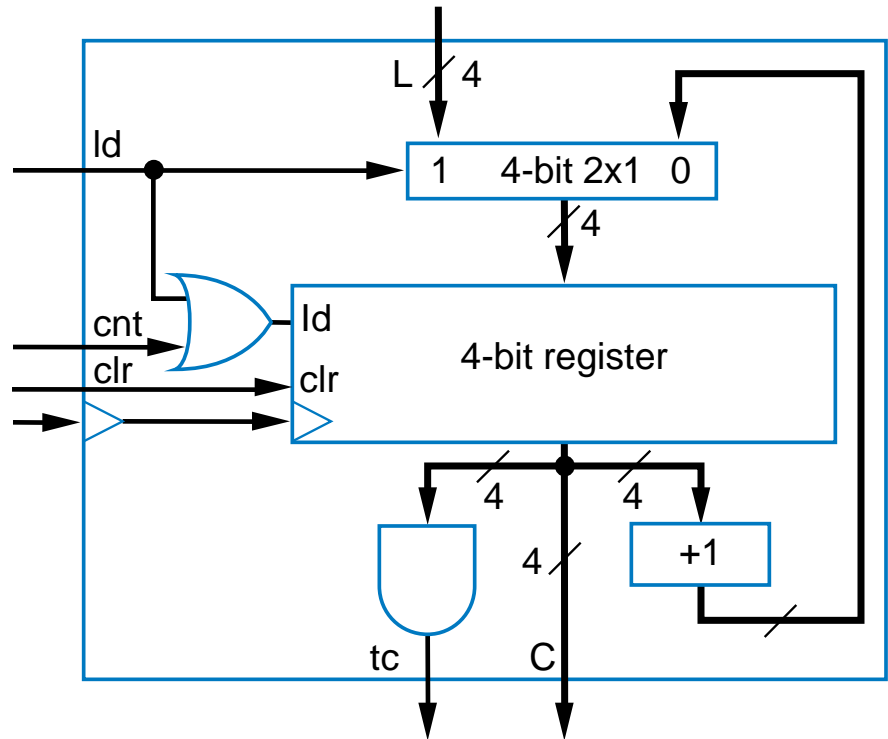
Up/Down-Counter

- Can count either up or down
 - Includes both incrementer and decrementer
 - Use dir input to select, via 2x1 mux: dir=0 means up
 - Likewise, dir selects appropriate terminal count value (all 1s or all 0s)



Counter with Load

- Up-counter that can be loaded with external value
 - Designed using 2x1 mux. Id input selects incremented value or external value
 - Load the internal register when loading external value or when counting
 - Note that Id has priority over cnt

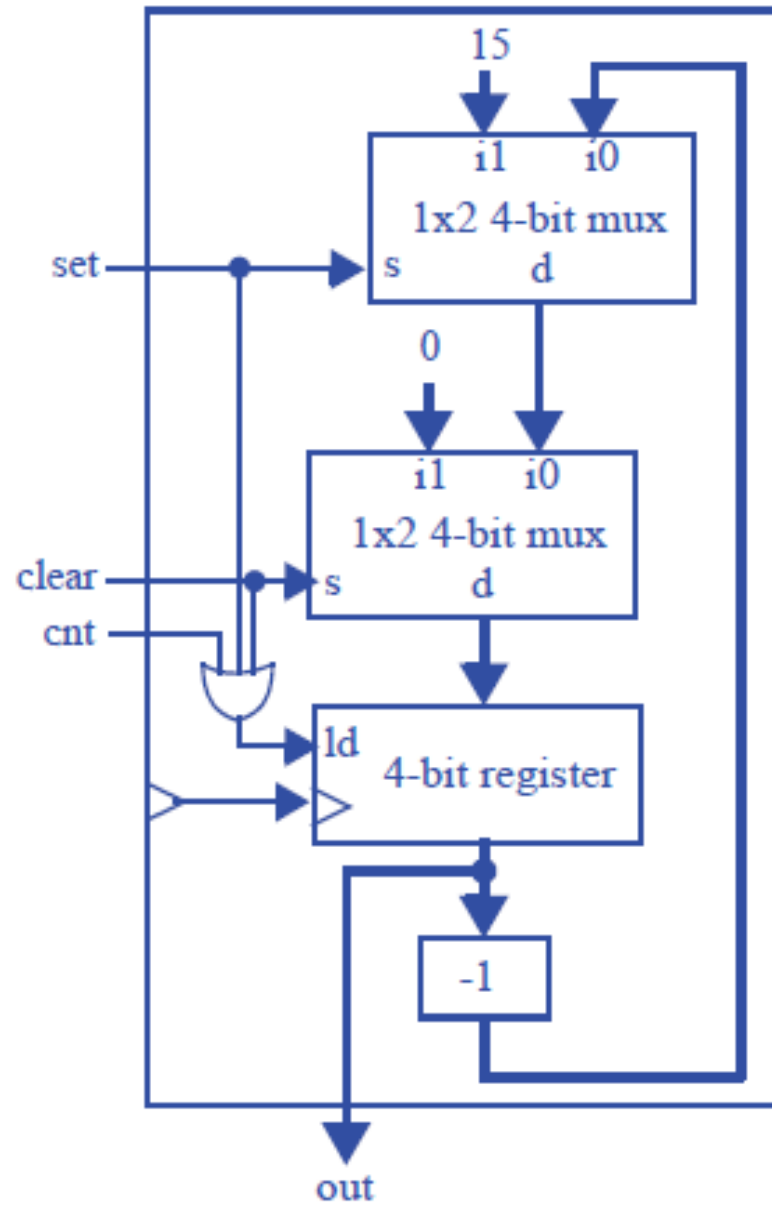


Exercise

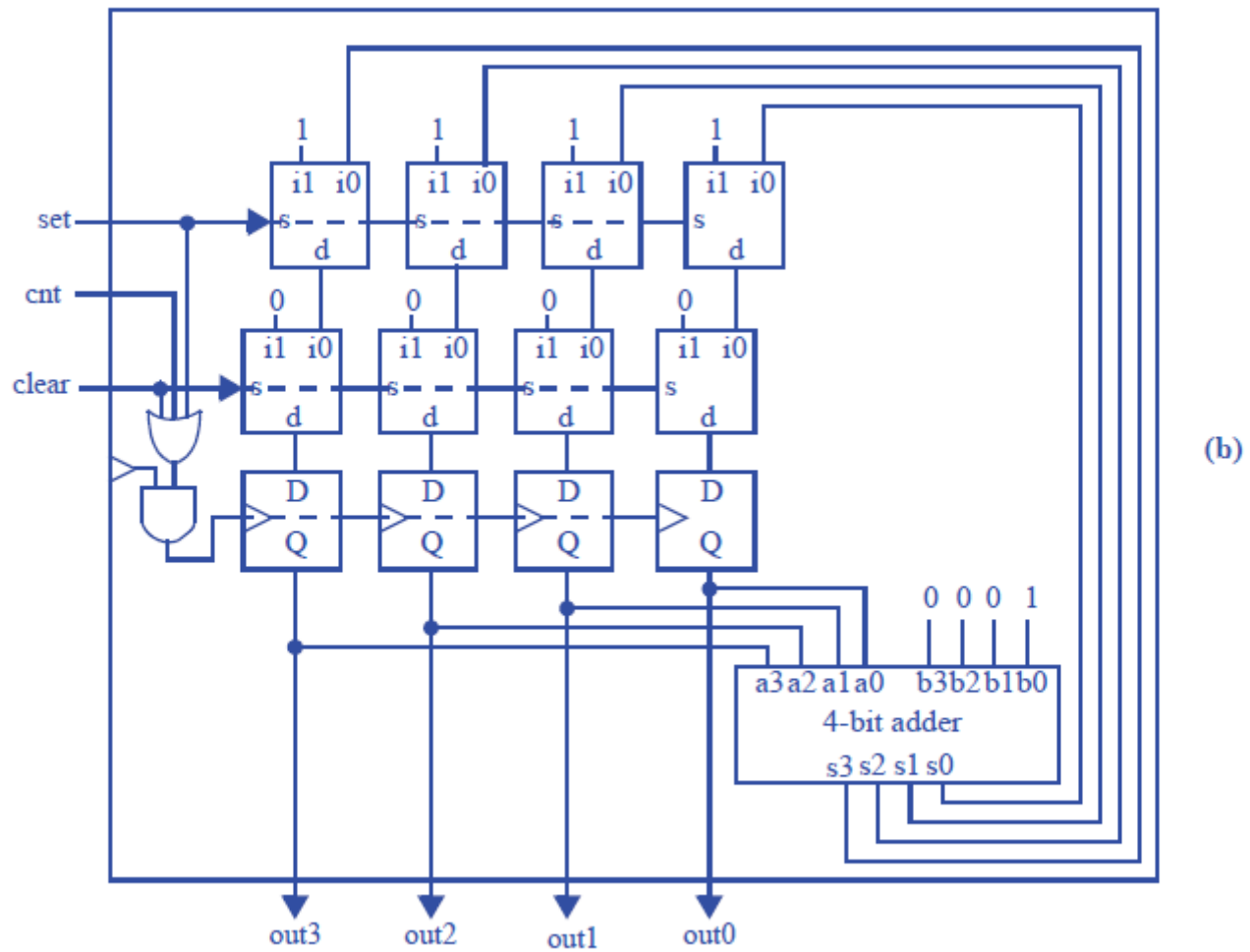
- 4.52 Design a 4-bit down-counter that has three control inputs: *cnt* enables counting up, *clear* synchronously resets the counter to all 0s, and *set* synchronously sets the counter to all 1s, (a) using a parallel load register as a building block, (b) using flip-flops and muxes directly by following the register design process of Section 4.2. (*Component design problem*).

```
module count4(  
    input clk,  
    input cnt,  
    input set,  
    input clear,  
    output logic [3:0] count  
);  
  
    always_ff @(posedge clk)  
        if (clear) count <= 4'b0000;  
        else if (set) count <= 4'b1111;  
        else if (cnt) count <= count-1;  
  
endmodule
```


*we'll give clear
precedence over set*



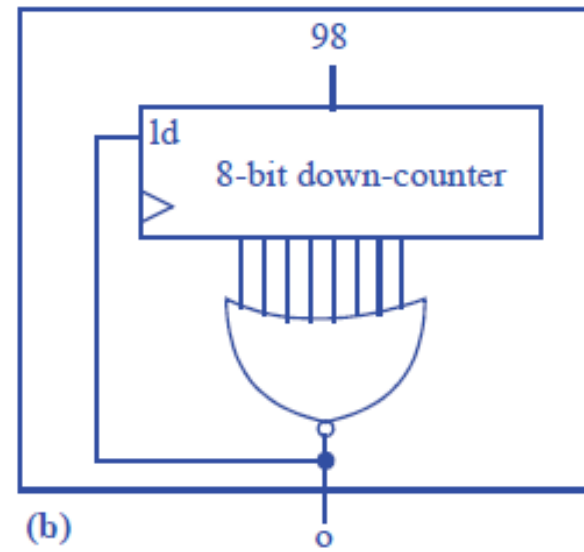
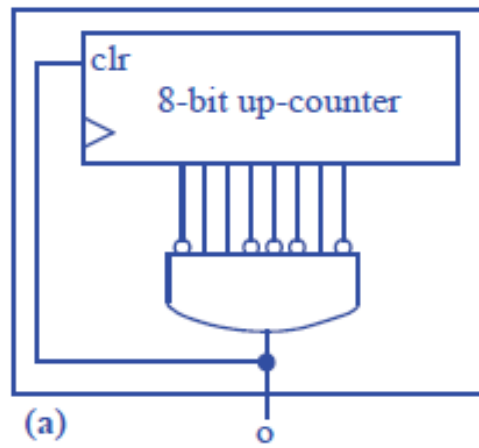
(a)



Exercise

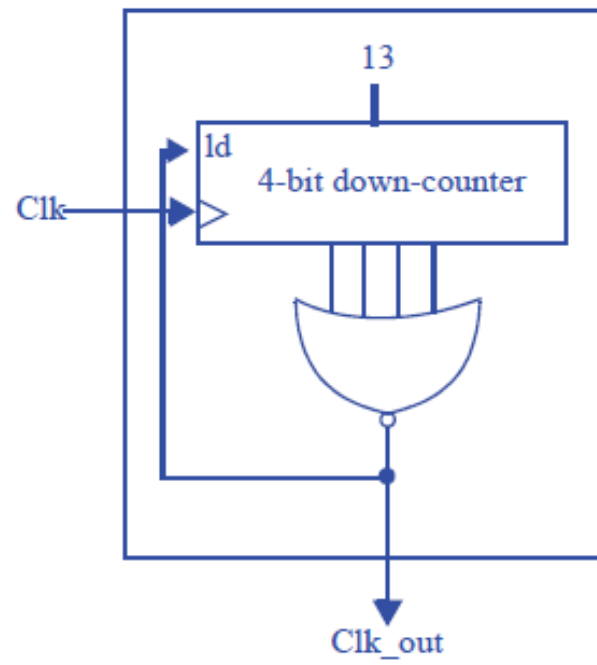
4.57 Design a circuit that outputs a 1 every 99 clock cycles:

- a. Using an up-counter with a synchronous clear control input, and using extra logic,
- b. Using a down-counter with parallel load, and using extra logic.
- c. What are the tradeoffs between the two designs from parts (a) and (b)?



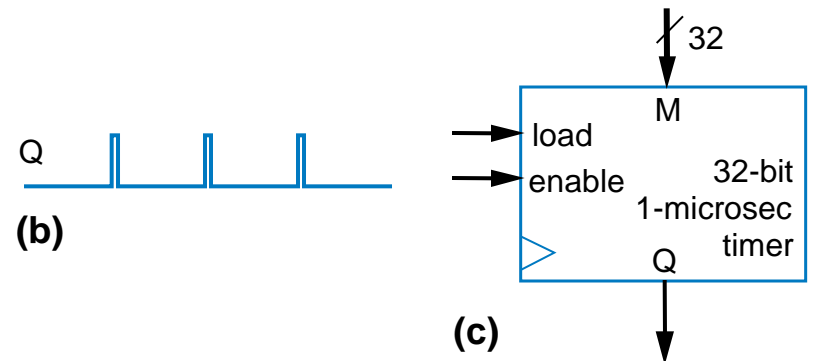
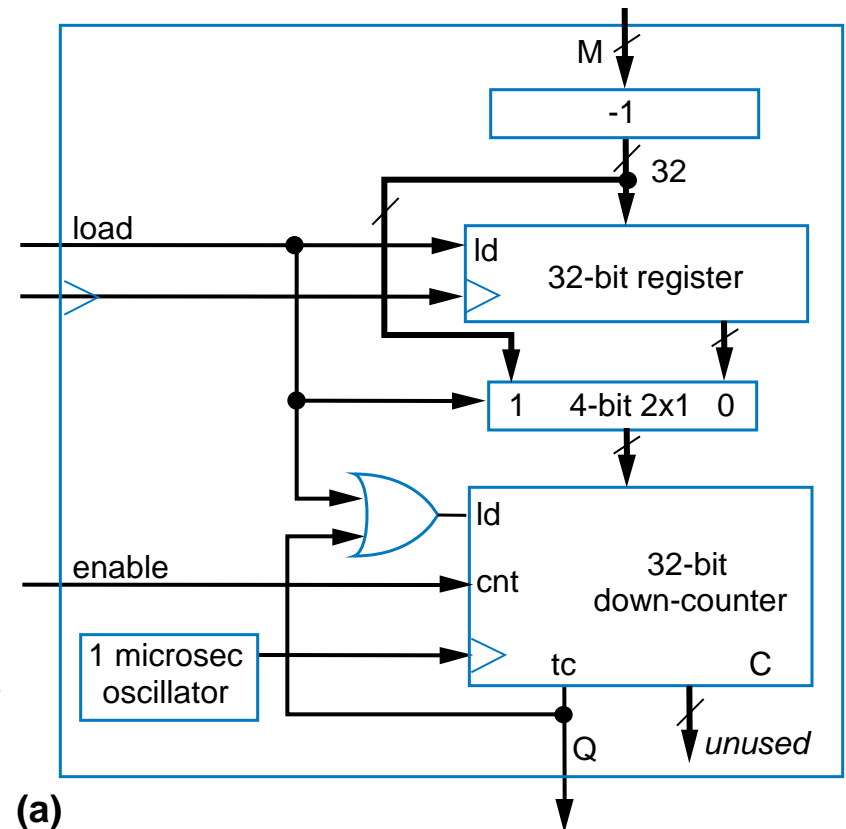
Exercise

- 4.59 Create a clock divider that converts a 14 MHz clock into a 1 MHz clock. Use a down-counter with parallel load. Clearly indicate the width of the down counter and the counter's load value. (Component use problem.)



Timers

- Pulses output at user-specified timer interval when enabled
 - “Ticks” like a clock
 - Interval specified as multiple of base time unit
 - If base is 1 microsec and user wants pulse every 300 ms, loads 300,000 into timer
- Can design using oscillator, register, and down-counter



```
module timer(input logic clk, load, enable, logic [31:0] M,  
             output logic Q, logic[31:0] downcount);  
  
    logic [31:0] next;  
    assign next = downcount-1;  
  
    always_ff @(posedge clk)  
        if (load | Q) downcount <= M-1;  
        else if (enable) downcount <= next;  
  
    always_ff @(posedge clk)  
        if (downcount==0) Q=1;  
        else Q=0;  
  
endmodule
```

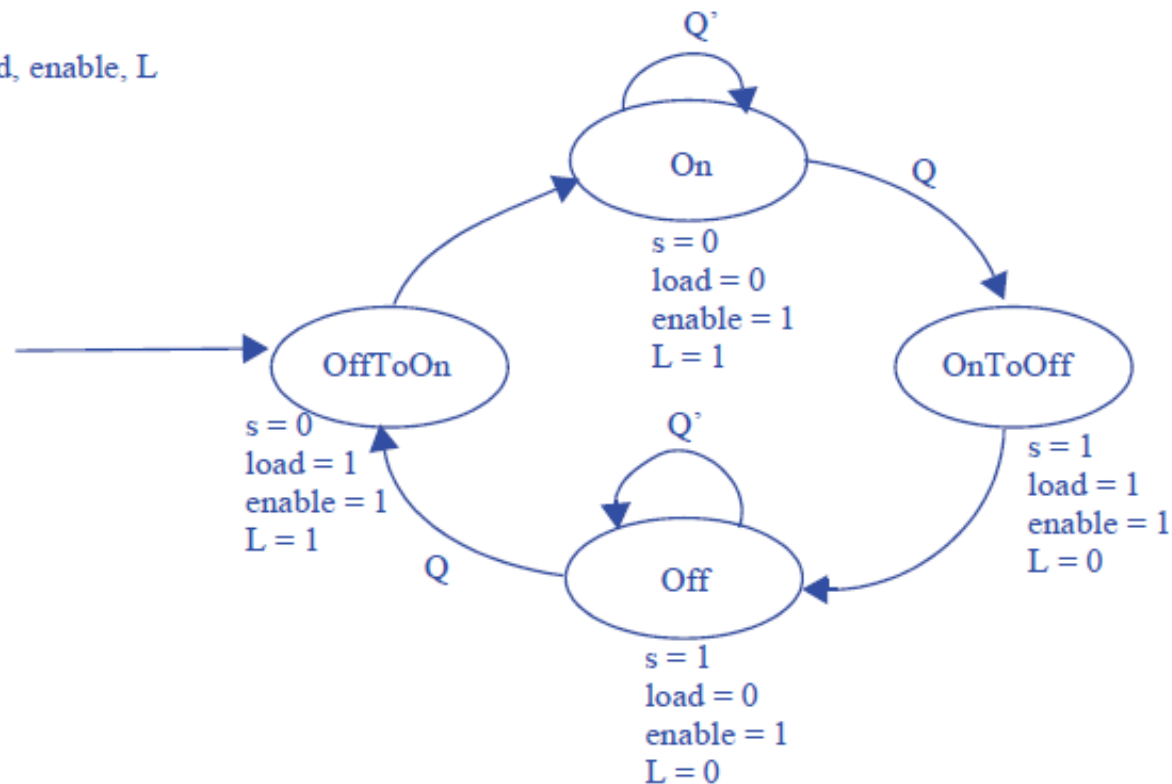

Exercise

- 4.60 Assuming a 32-bit microsecond timer is available to a controller and a controller clock frequency of 100 MHz, create a controller FSM that blinks an LED by setting an output L to 1 for 5 ms and then to 0 for 13 ms, and then repeats. Use the timer to achieve the desired timing (i.e., do not use a clock divider). For this example, the blinking rate can vary by a few clock cycles. (Component use problem.)

Assuming the timer's input is connected to a 1x2 32-bit mux whose i0 is 5000 and whose i1 is 13000, the mux's select line is called 's', one possible FSM would be:

Inputs: Q

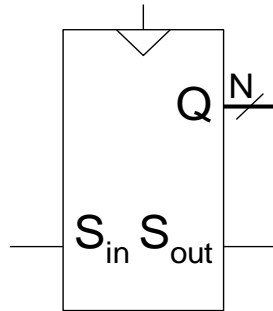
Outputs: s, load, enable, L



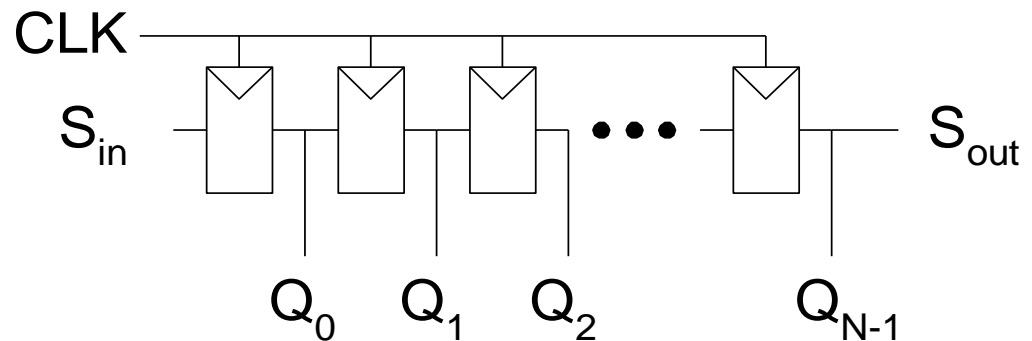
Shift Registers

- Shift a new bit in on each clock edge
- Shift a bit out on each clock edge
- *Serial-to-parallel converter*: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)

Symbol:

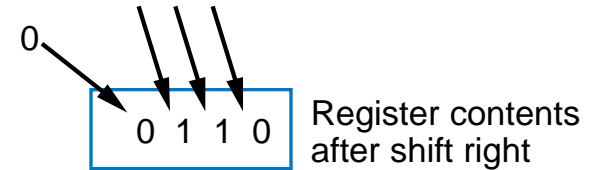


Implementation:



Shift Register

- Shift right
 - Move each bit one position right
 - Rightmost bit is “dropped”
 - Assume 0 shifted into leftmost bit



Q: Do four right shifts on 1001, showing value after each shift

A: 1001 (original)

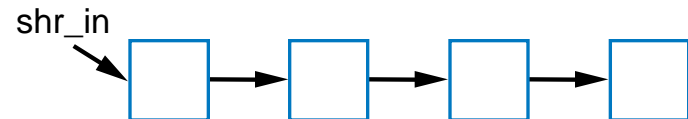
0100

0010

0001

0000

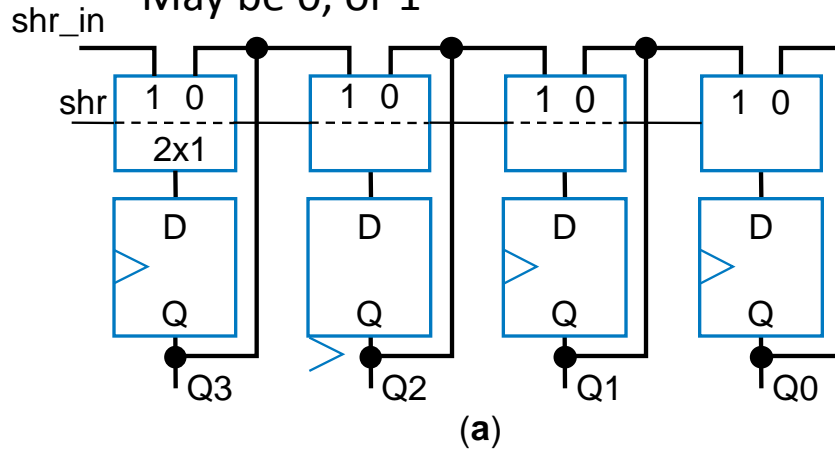
- Implementation: Connect flip-flop output to next flip-flop's input



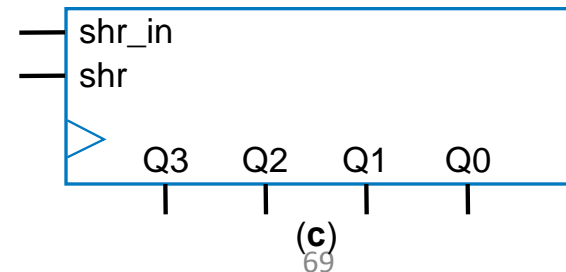
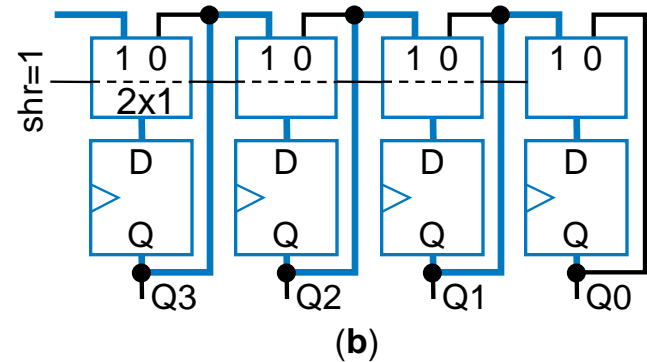
Shift Register

- To allow register to either shift or retain, use 2x1 muxes
 - shr: “0” means retain, “1” shift
 - shr_in: value to shift in

• May be 0, or 1

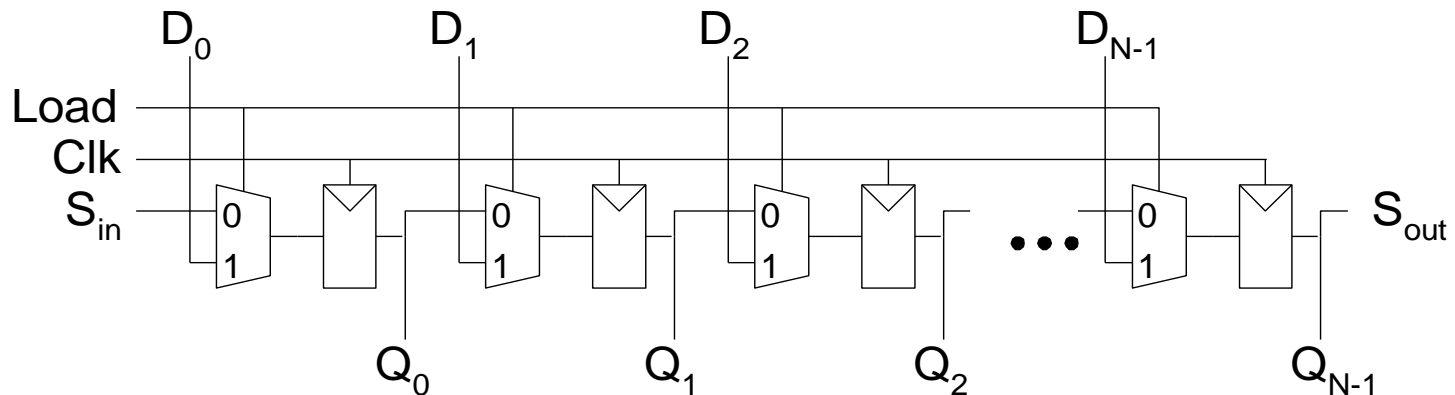


Left-shift register also easy to design



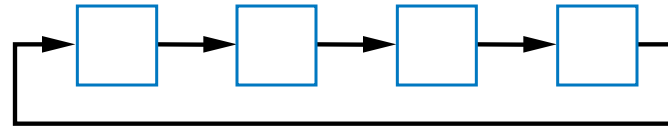
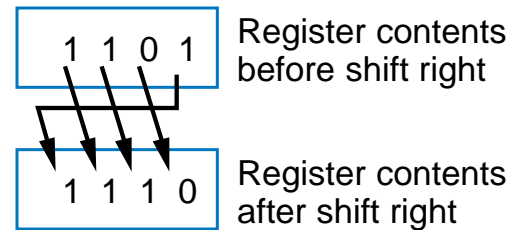
Shift Register with Parallel Load

- When $Load = 1$, acts as a normal N -bit register
- When $Load = 0$, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out})



Rotate Register

- Rotate right: Like shift right, but leftmost bit comes from rightmost bit



Example: Above-Mirror Car Display

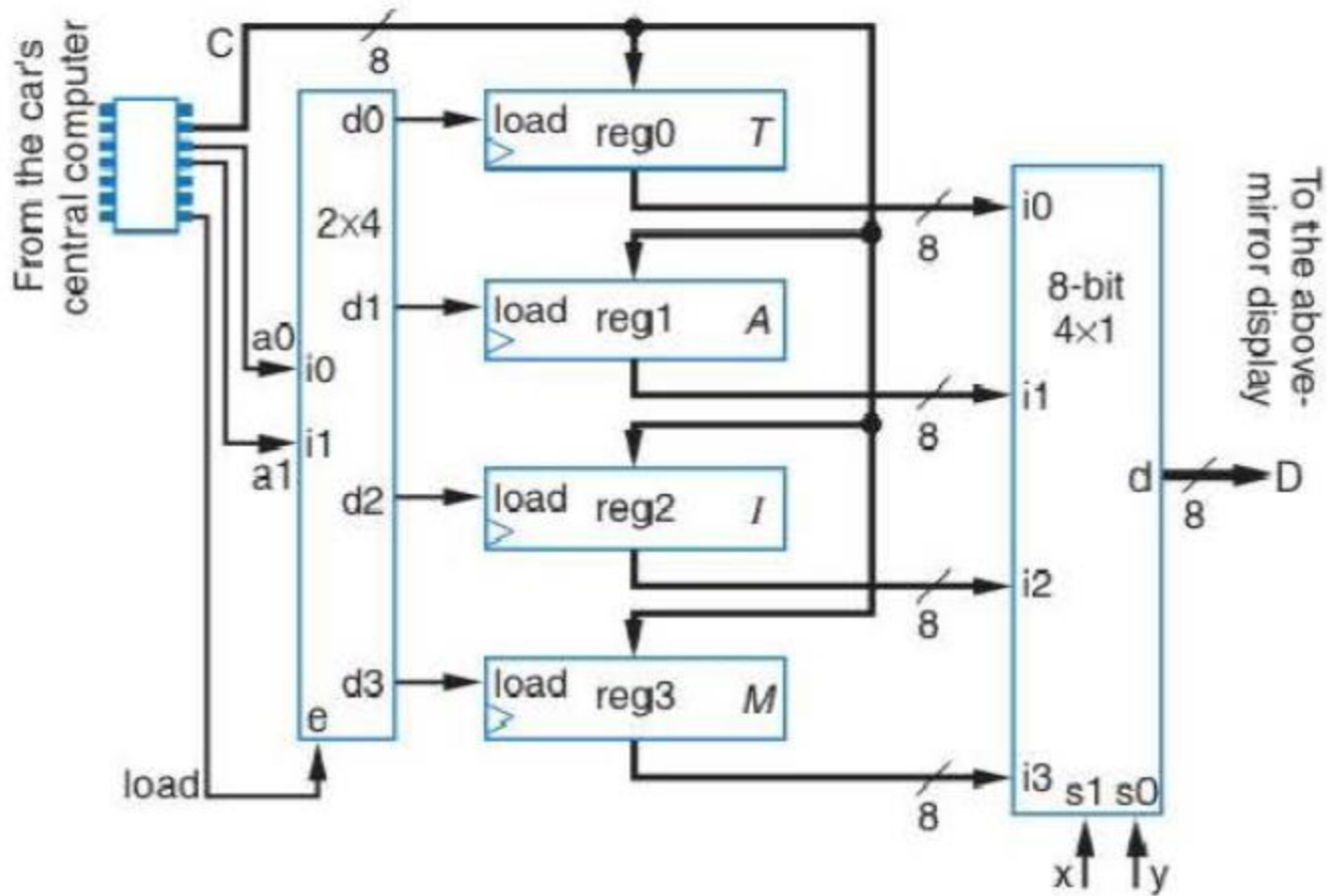
Design the following system to be used for an above-mirror car display: At any point in time, one of the four 8-bit values need to be displayed:

- 1) temperature,
- 2) fuel economy,
- 3) fuel remaining,
- 4) speed.

Operation:

- The type of the value is determined by the **2-bit signals a_1a_0 or x_1x_0** .
- The car's central computer can update these values at arbitrary times and in arbitrary order. It sends the data to your system over an **8-bit bus C** after setting the **2-bit signal a_1a_0** and **single-bit signal load**.
- Depending on the value of **x_1x_0** , your system should output the corresponding 8-bit value to the display system through an **8-bit bus D**.

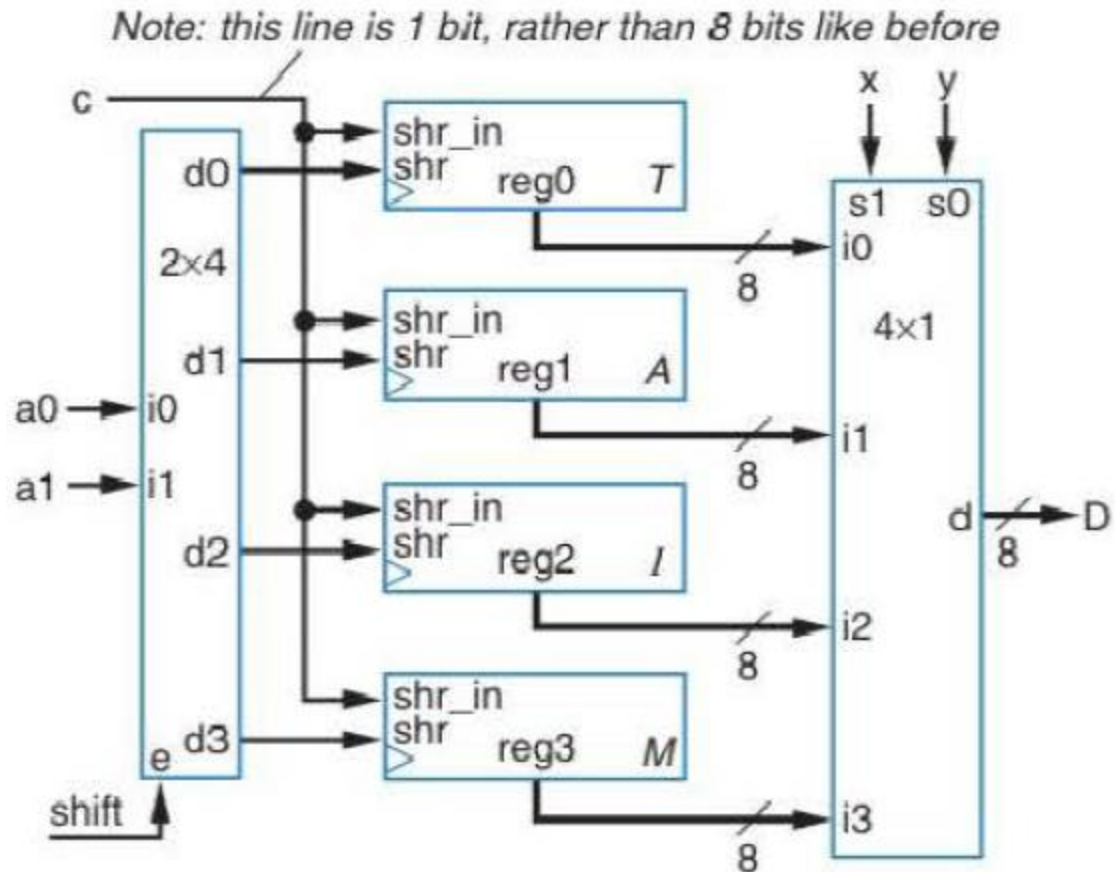
Example: Above-Mirror Car Display



Example: Above-Mirror Car Display (cont'd)

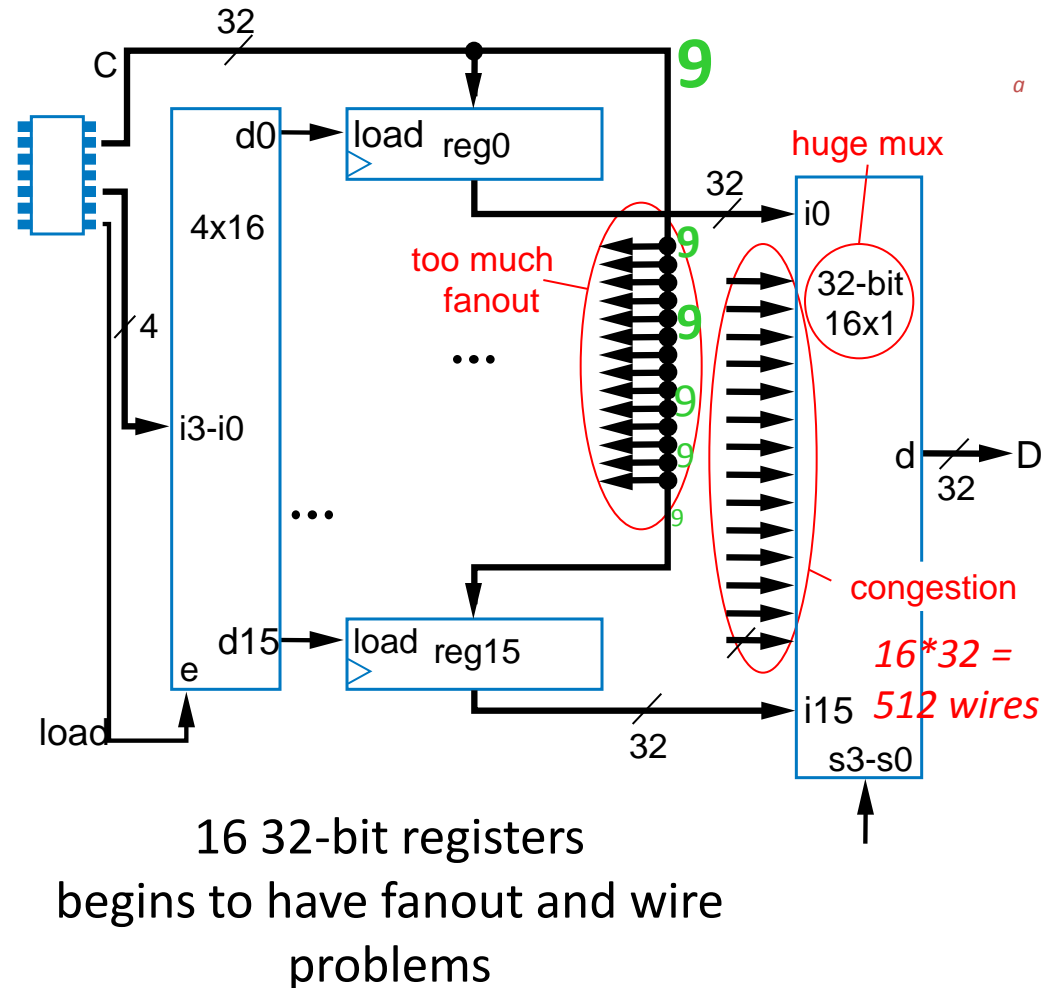
How can we reduce the number of wires from car computer to your system from 11 to 4?

Example: Above-Mirror Car Display (cont'd)



Register Files

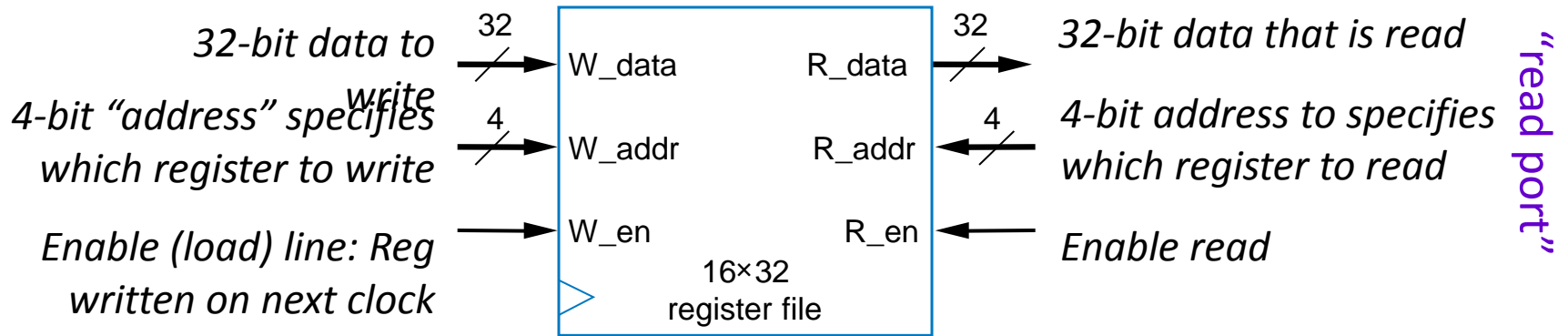
- Accessing one of several registers is:
 - OK if just a few registers
 - Problematic when many
 - Ex: Earlier above-mirror display, with 16 registers
 - Much fanout (branching of wire): Weakens signal
 - Many wires: Congestion



Register File

- **MxN register file:** Efficient design for one-at-a-time write/read of many registers
 - Consider 16 32-bit registers

Called “write port”

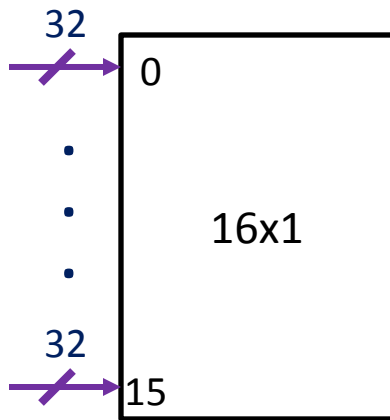


“read port”

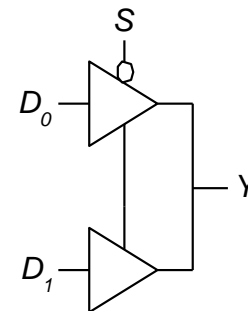
a

Internal Implementation

- How to handle the large fanout problem?
 - Use buffers/repeaters
- How to implement the 32-bit 16x1 MUX efficiently?

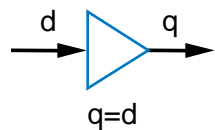


Reminder: MUX implementation with tri-state buffers:



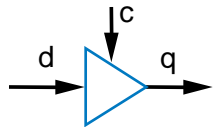
Register File

- Internal design uses drivers and bus

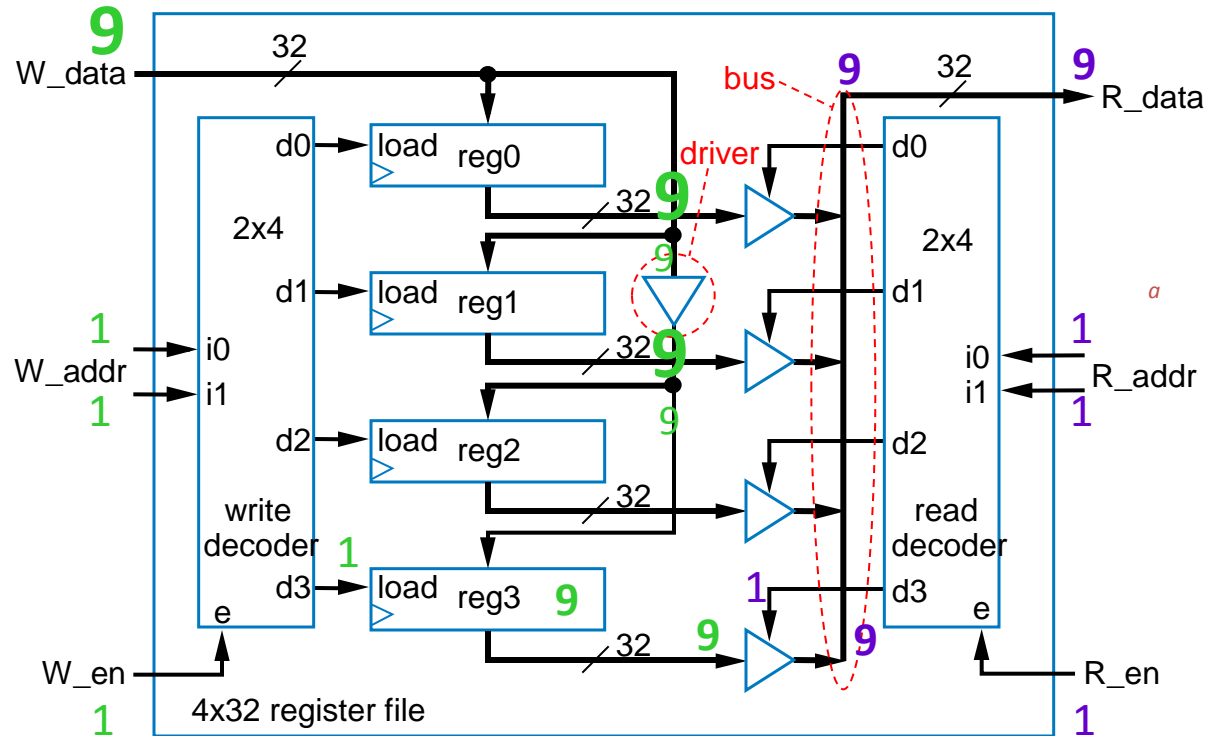


Boosts signal

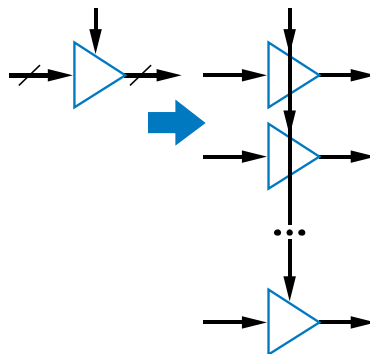
three-state driver



$c=1: q=d$
 $c=0: q=Z' d$
like no connection



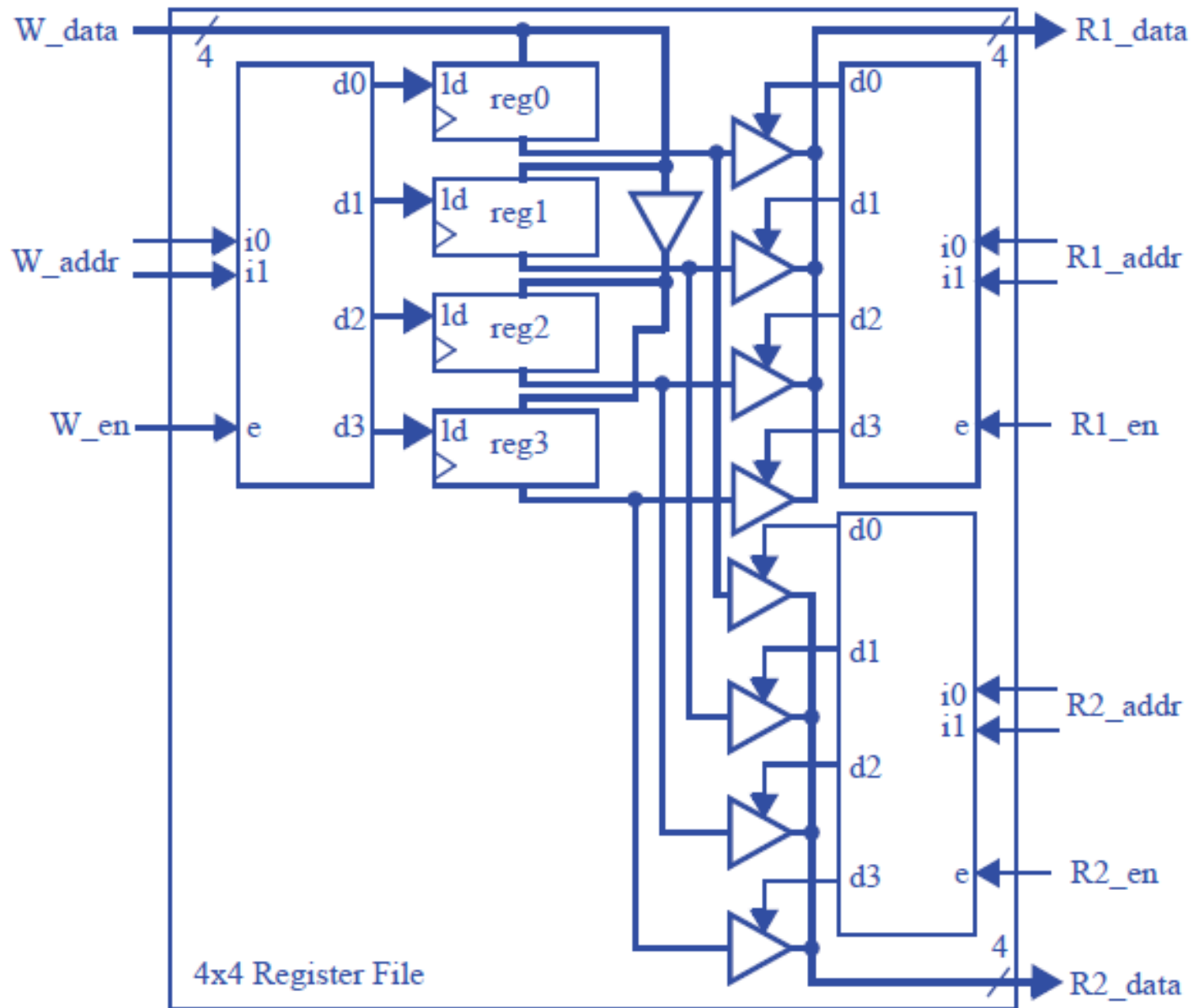
Internal design of 4x32 RF; 16x32 RF follows similarly



Note: Each driver in figure actually represents 32 1-bit drivers

Exercise

4.62 Design a 4x4 three port (2 read, 1 write) register file. (*Component design problem*).



Exercise

- 4.64 A 4x4 register file's four registers initially each contain 0101.
- Show the input values necessary to read register 3 and to simultaneously write register 3 with the value 1110.
 - With these values, show the register file's register values and output values before the next rising clock edge, and after the next rising clock edge.

a.) $W_data = 1110$, $W_addr = 11$, $W_en = 1$, $R_addr = 11$, $R_en = 1$.

b.) Before rising edge:

$R0 = 0101$

$R1 = 0101$

$R2 = 0101$

$R3 = 0101$

$R_data = 0101$

After rising edge:

$R0 = 0101$

$R1 = 0101$

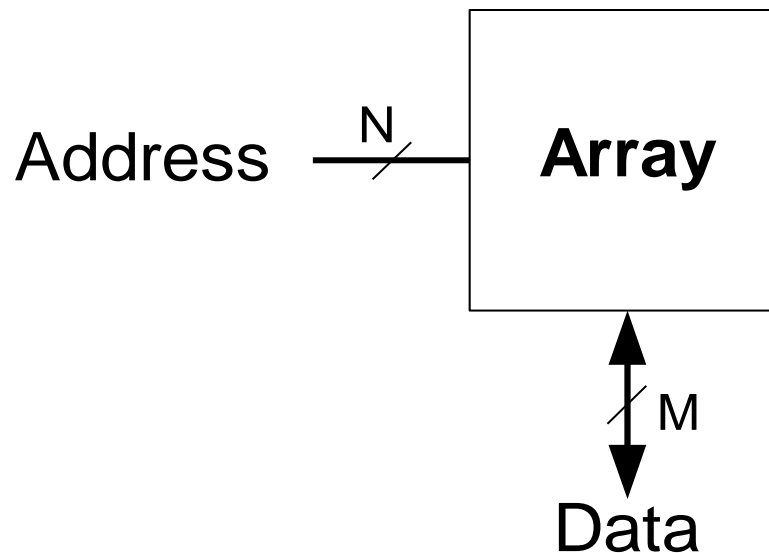
$R2 = 0101$

$R3 = 1110$

$R_data = 1110$

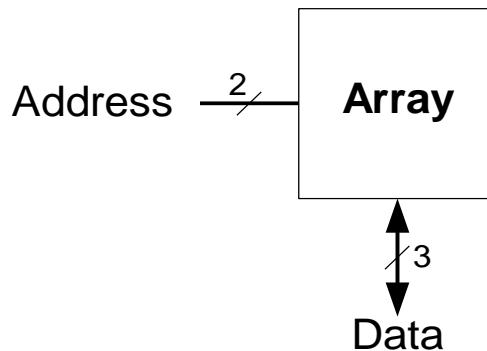
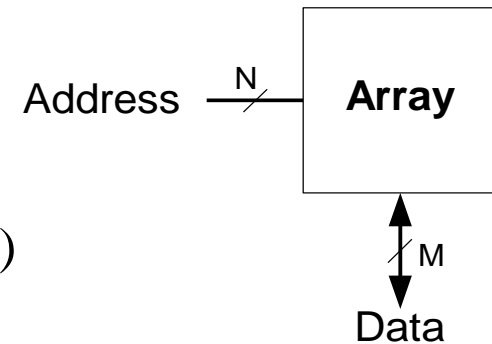
Memory Arrays

- 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

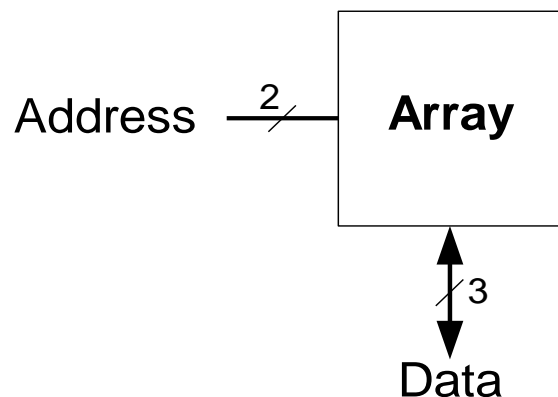
- 2-dimensional array of bit cells
- Each bit cell stores one bit
- N address bits and M data bits:
 - 2^N rows and M columns
 - **Depth:** number of rows (number of words)
 - **Width:** number of columns (size of word)
 - **Array size:** depth \times width = $2^N \times M$



Address	Data			
11	0	1	0	<div style="display: flex; align-items: center;"> <div style="width: 10px; height: 10px; border: 1px solid black; margin-right: 5px;"></div> <div style="text-align: center;"> ↑ depth ↓ </div> </div>
10	1	0	0	
01	1	1	0	
00	0	1	1	
	<div style="display: flex; align-items: center;"> <div style="width: 10px; height: 10px; border: 1px solid black; margin-right: 5px;"></div> <div style="text-align: center;"> ← width → </div> </div>			

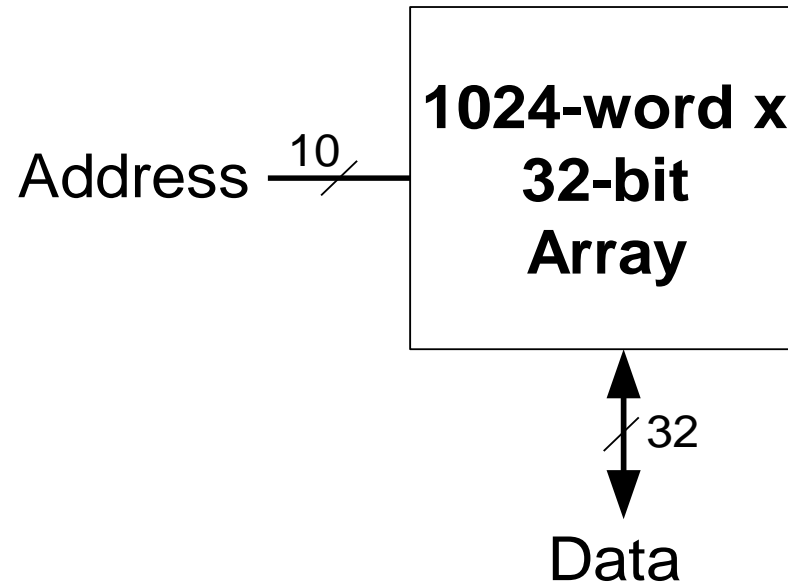
Memory Array Example

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

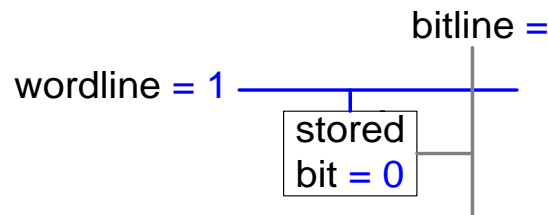
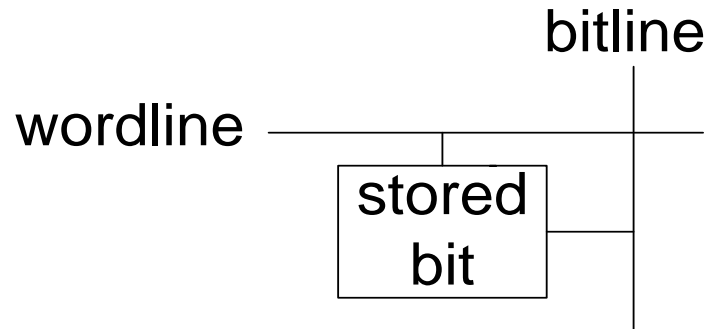


Address	Data			
11	0	1	0	depth ↑ ↓
10	1	0	0	
01	1	1	0	
00	0	1	1	
	width ←→			

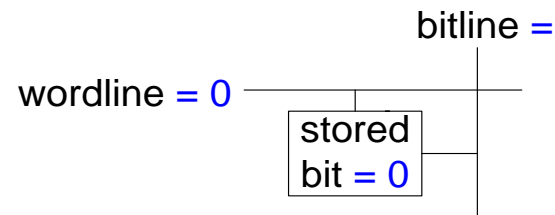
Memory Arrays



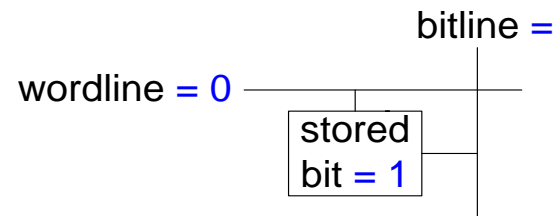
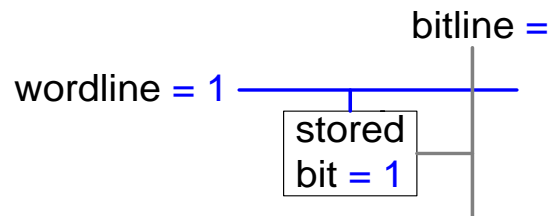
Memory Array Bit Cells



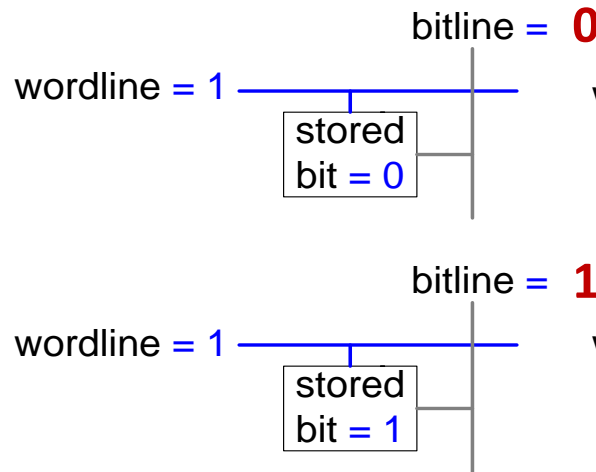
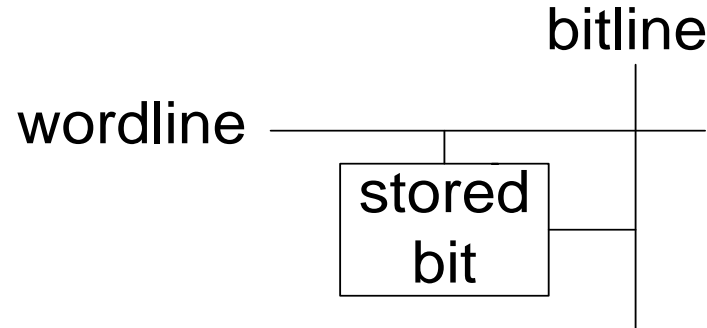
(a)



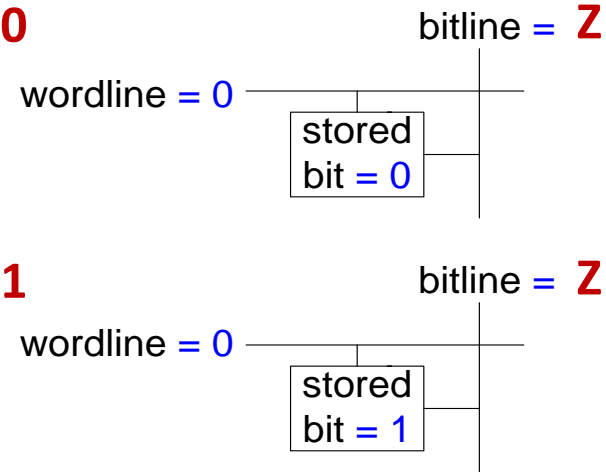
(b)



Memory Array Bit Cells



(a)

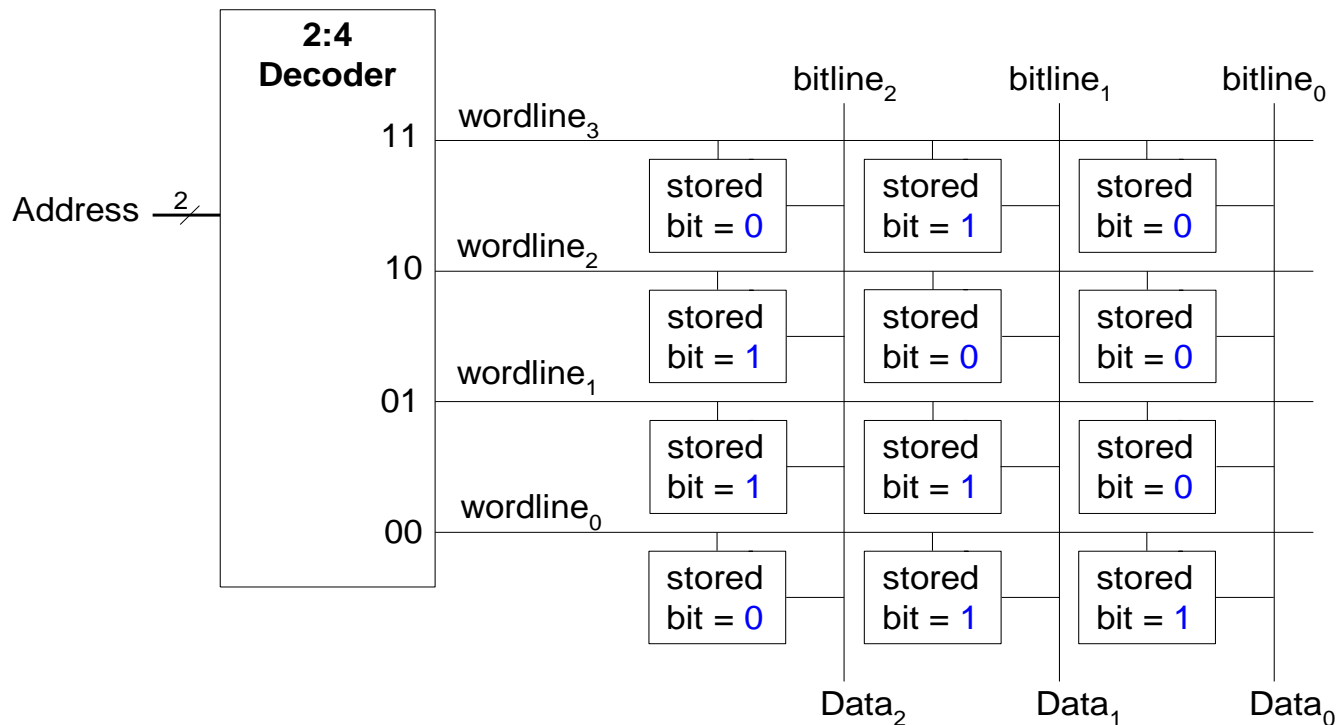


(b)

Memory Array

- **Wordline:**

- like an enable
- single row in memory array read/written
- corresponds to unique address
- only one wordline HIGH at once



Types of Memory

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

RAM: Random Access Memory

- **Volatile:** loses its data when power off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

Historically called *random* access memory because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)

ROM: Read Only Memory

- **Nonvolatile:** retains data when power off
- Read quickly, but writing is impossible or slow
- Flash memory in cameras, thumb drives, and digital cameras are all ROMs

Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.



Types of RAM

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
 - DRAM uses a capacitor
 - SRAM uses cross-coupled inverters

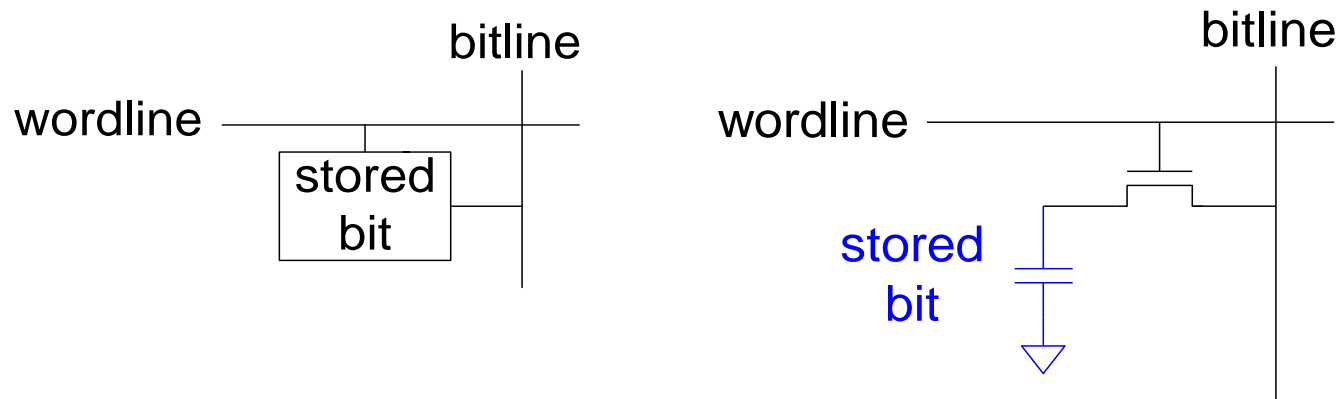
Robert Dennard, 1932 -

- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM in virtually all computers

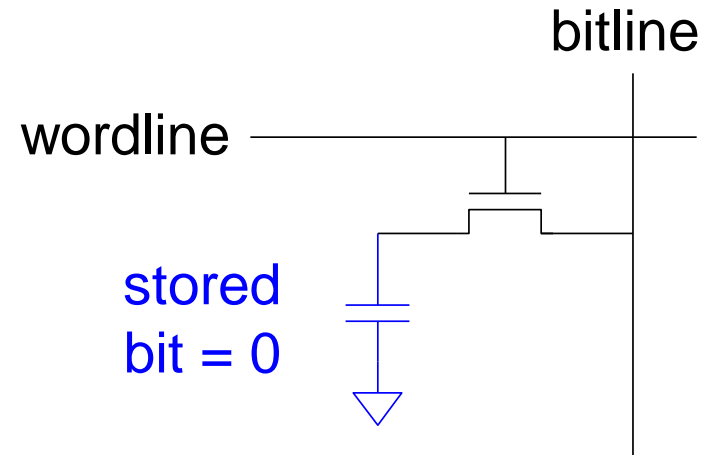
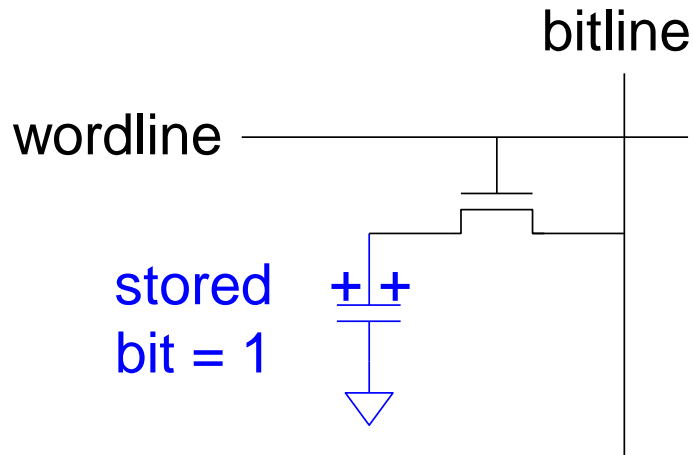


DRAM

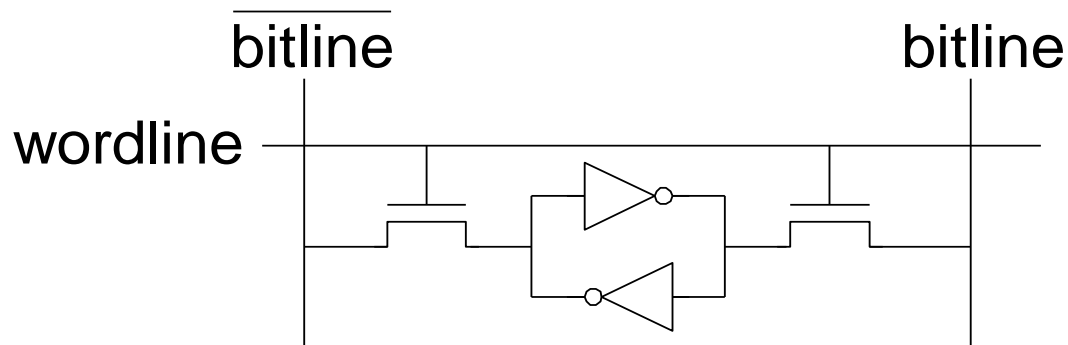
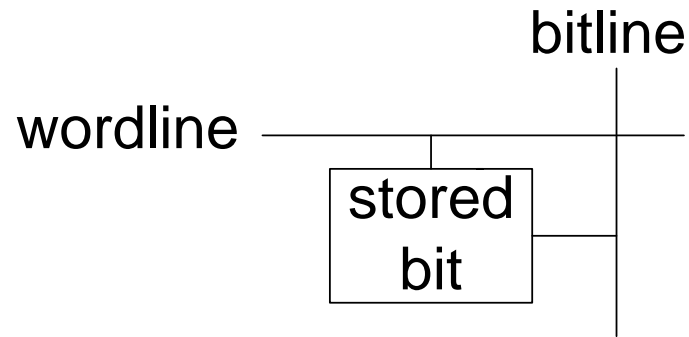
- Data bits stored on capacitor
- *Dynamic* because the value needs to be refreshed (rewritten) periodically and after read:
 - Charge leakage from the capacitor degrades the value
 - Reading destroys the stored value



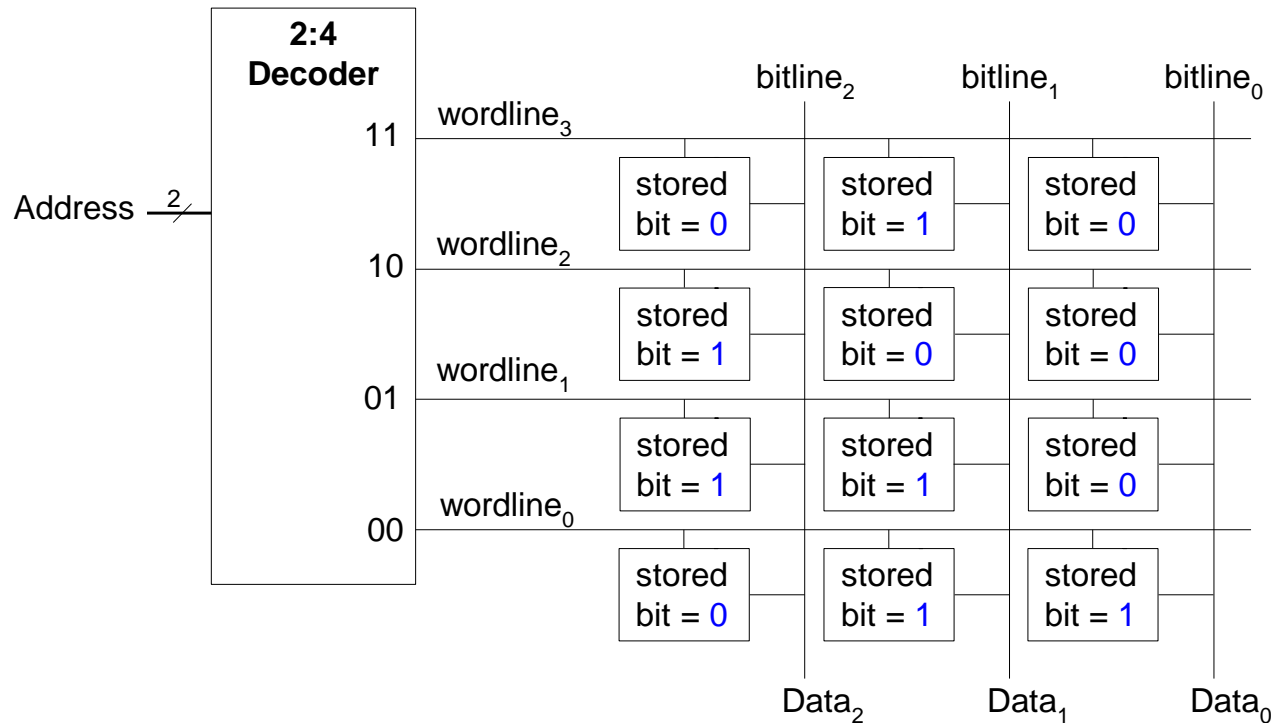
DRAM



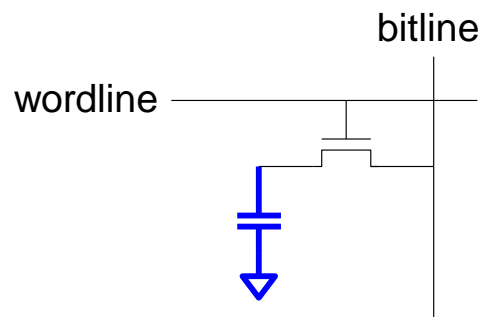
SRAM



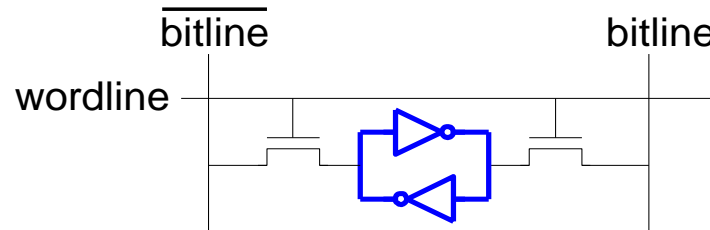
Memory Arrays Review



DRAM bit cell:



SRAM bit cell:



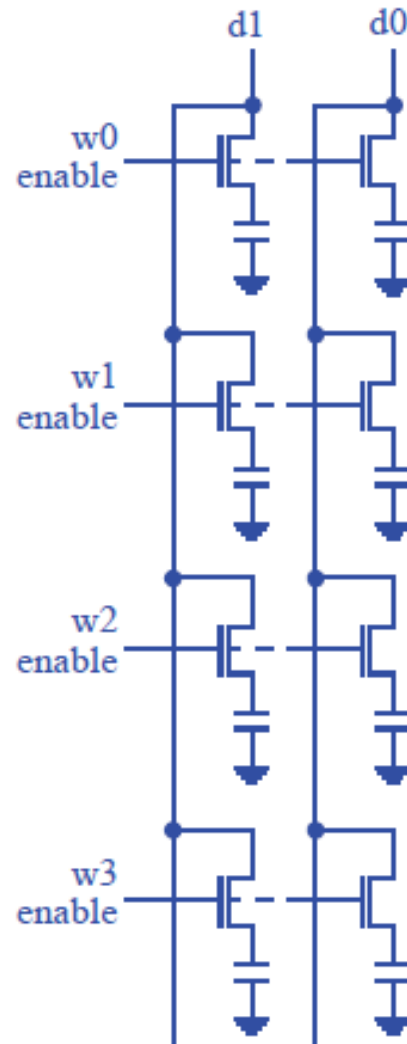
Memory Comparison

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

Exercise

5.39 Draw a circuit of transistors showing the internal structure for all the storage cells for a 4x2 DRAM (four words, two bits each), clearly labelling all internal components and connections.

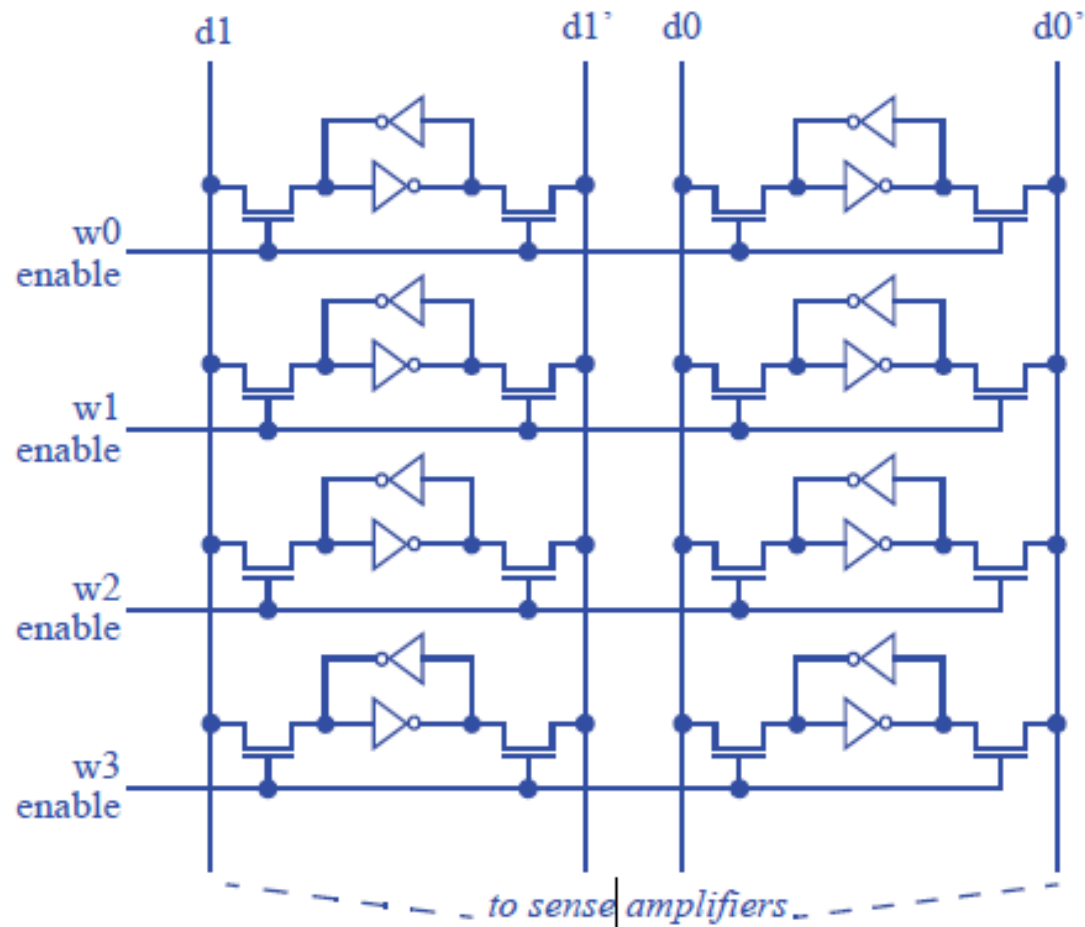
Solution



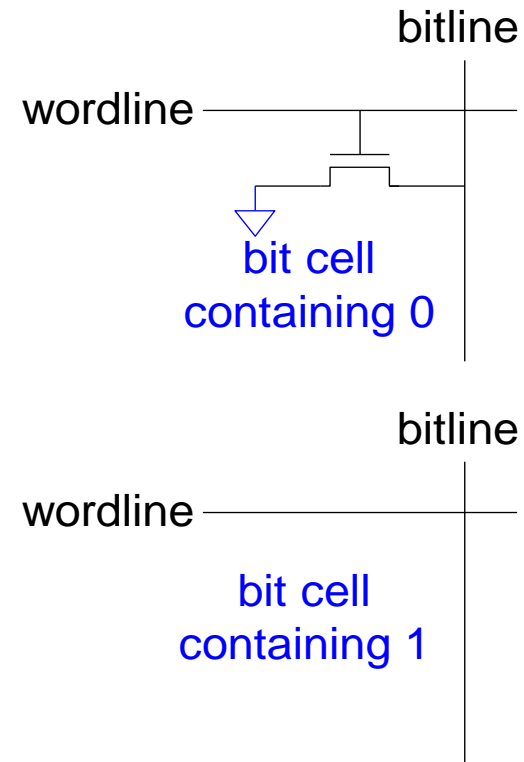
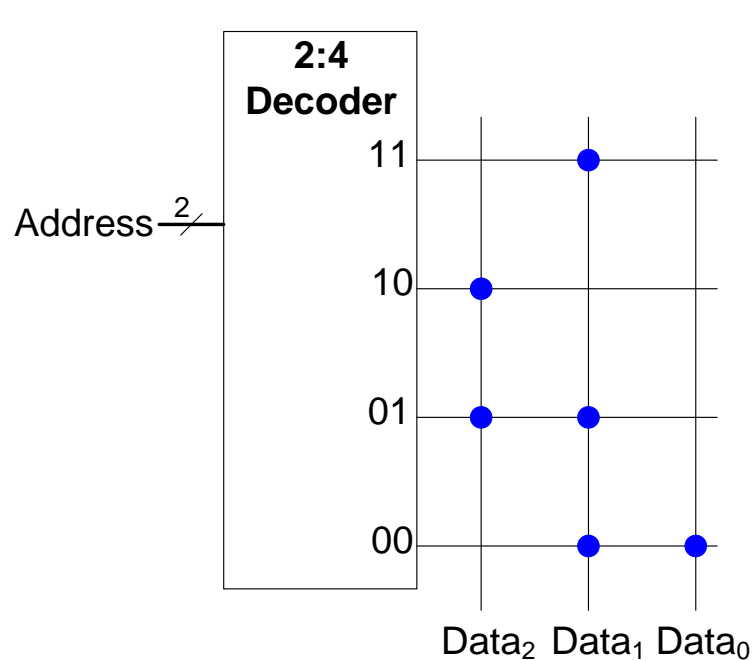
Exercise

5.40 Draw a circuit of transistors showing the internal structure for all the storage cells for a 4x2 SRAM (four words, two bits each), clearly labelling all internal components and connections.

Exercise

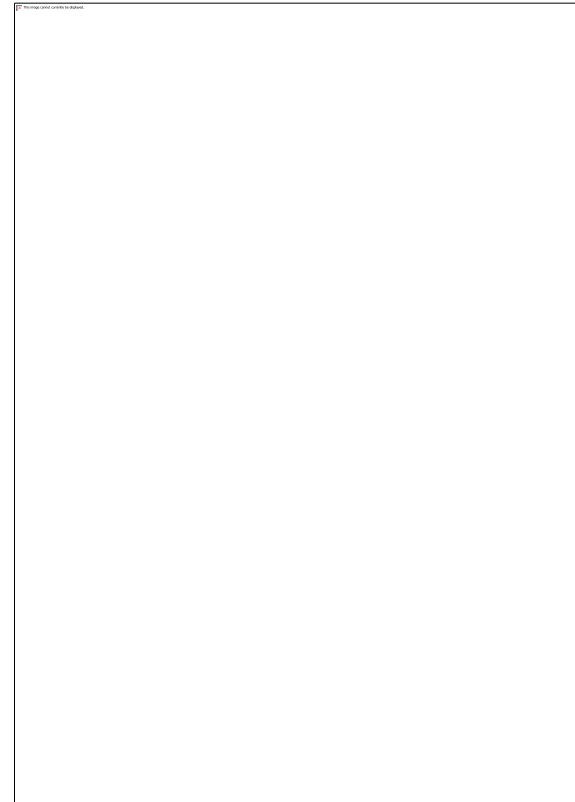


ROM: Dot Notation

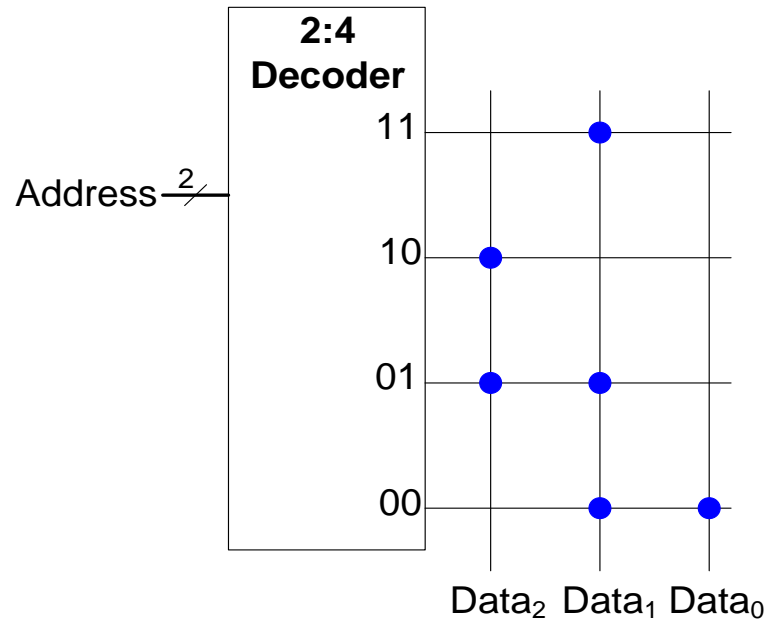


Fujio Masuoka, 1944 -

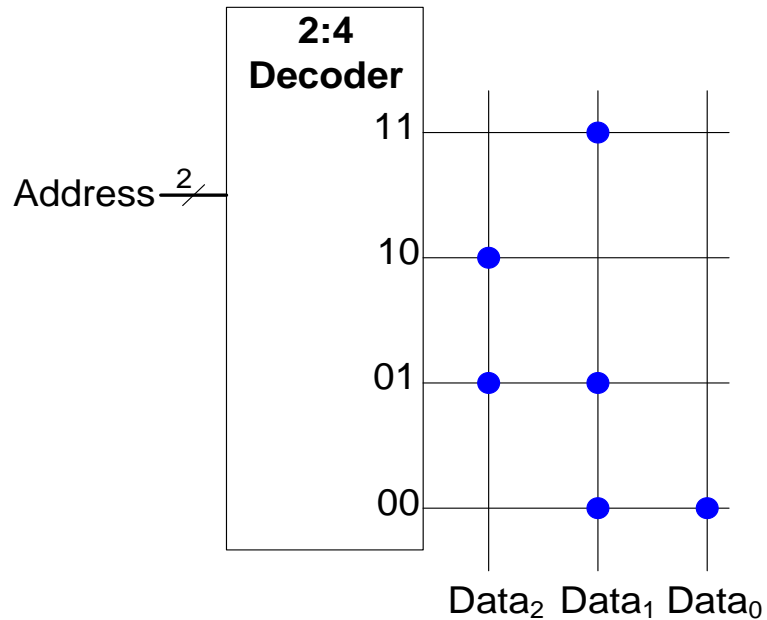
- Developed memories and high speed circuits at Toshiba, 1971-1994
- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's
- The process of erasing the memory reminded him of the flash of a camera
- Toshiba slow to commercialize the idea; Intel was first to market in 1988
- Flash has grown into a \$25 billion per year market



ROM Storage

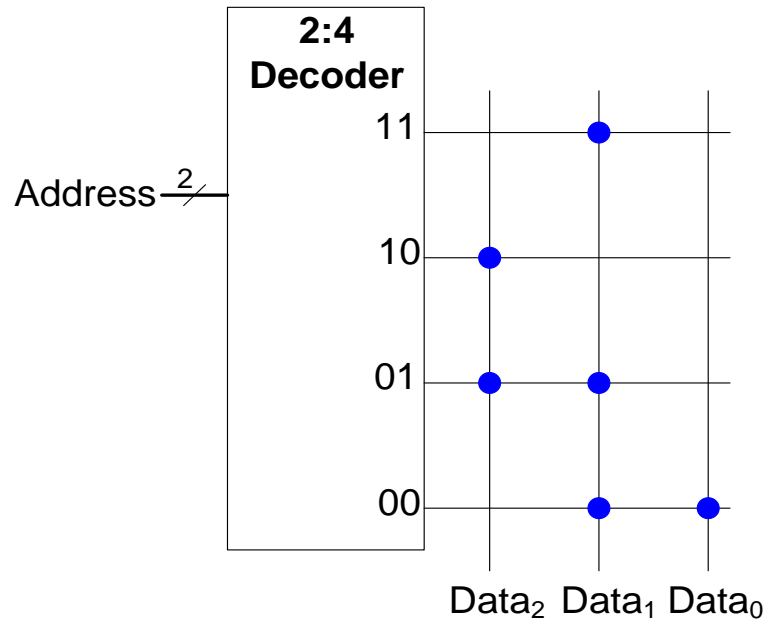


ROM Storage

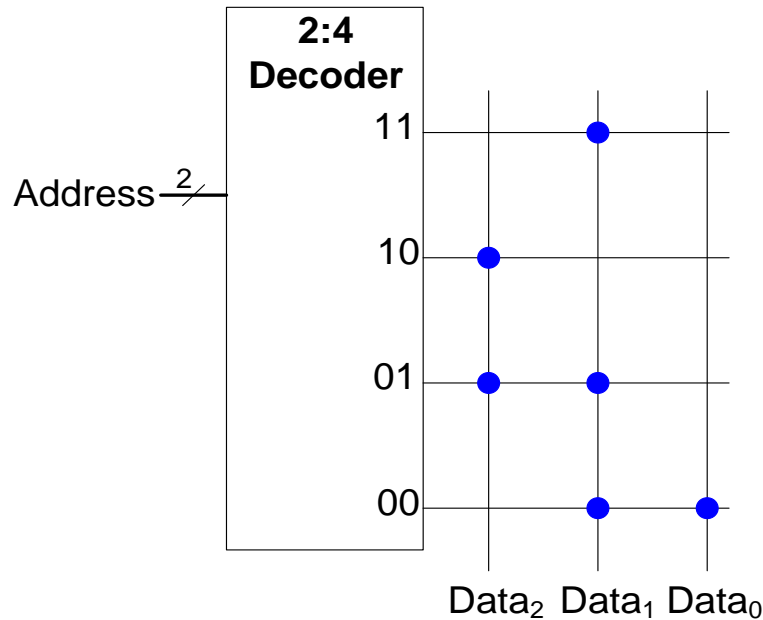


Address	Data			depth ↑ ↓
11	0	1	0	
10	1	0	0	
01	1	1	0	
00	0	1	1	
			width ←→	

ROM Logic



ROM Logic



$$Data_2 = A_1 \oplus A_0$$

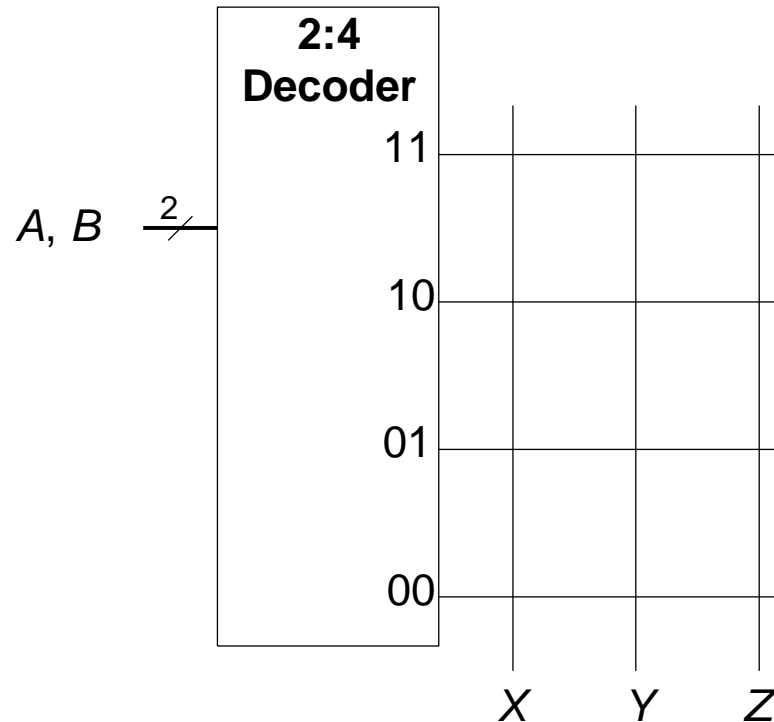
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1} \overline{A_0}$$

Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$ -bit ROM:

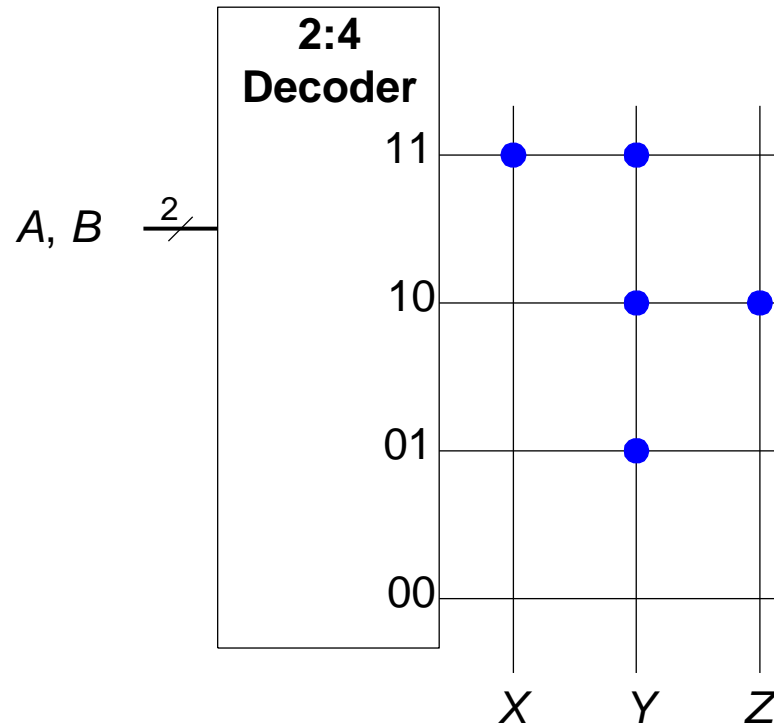
- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$



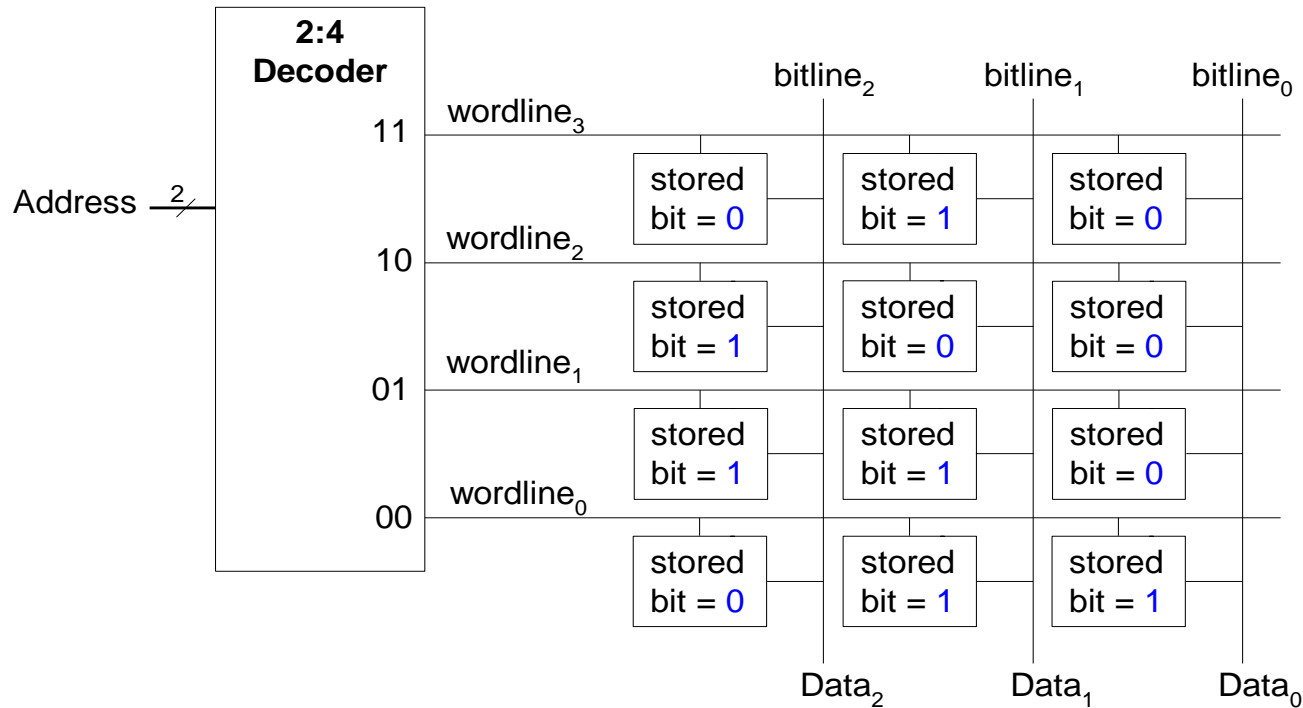
Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$ -bit ROM:

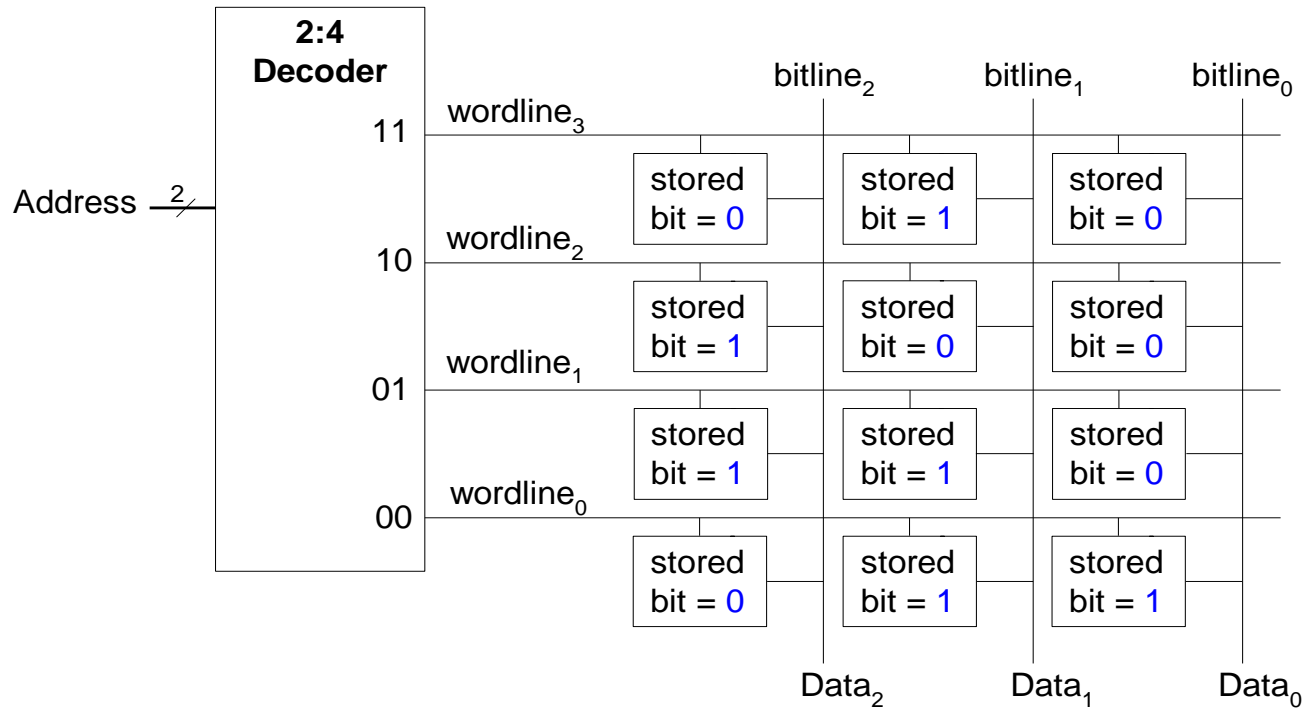
- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$



Logic with Any Memory Array



Logic with Any Memory Array



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \bar{A}_1 + A_0$$

$$Data_0 = \bar{A}_1 \bar{A}_0$$

Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$ -bit memory array:

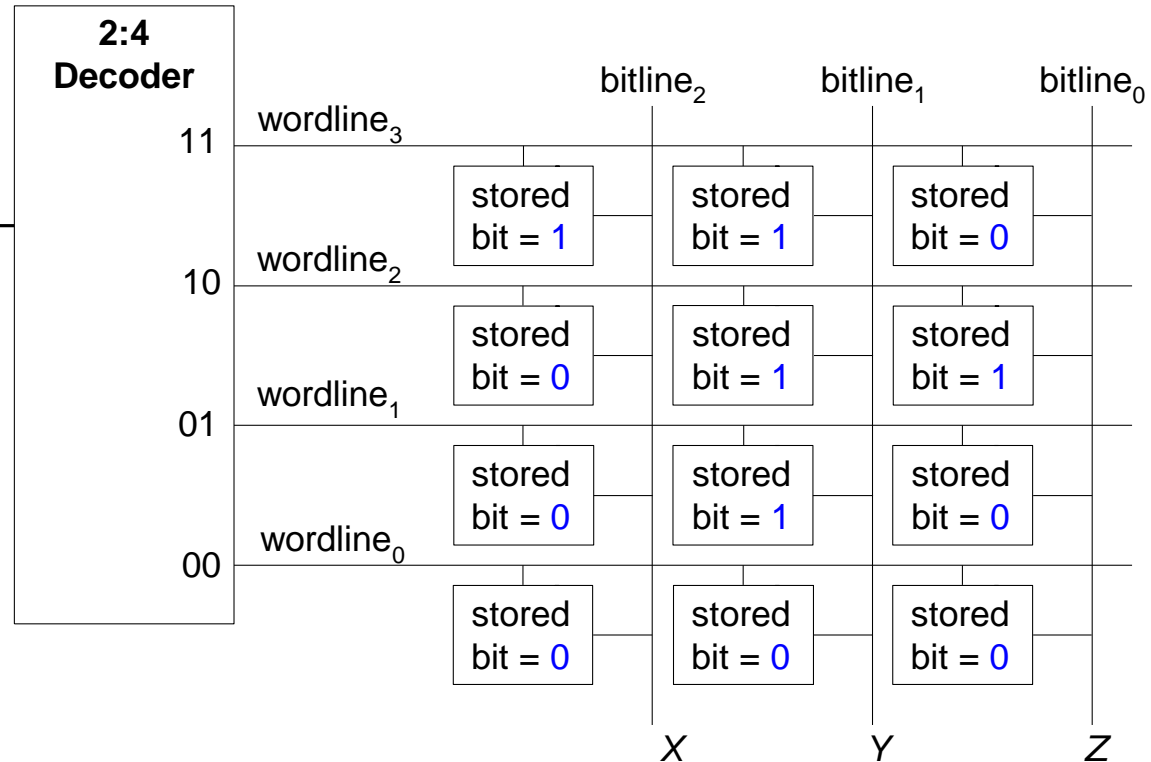
- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$

Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$ -bit memory array:

- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$

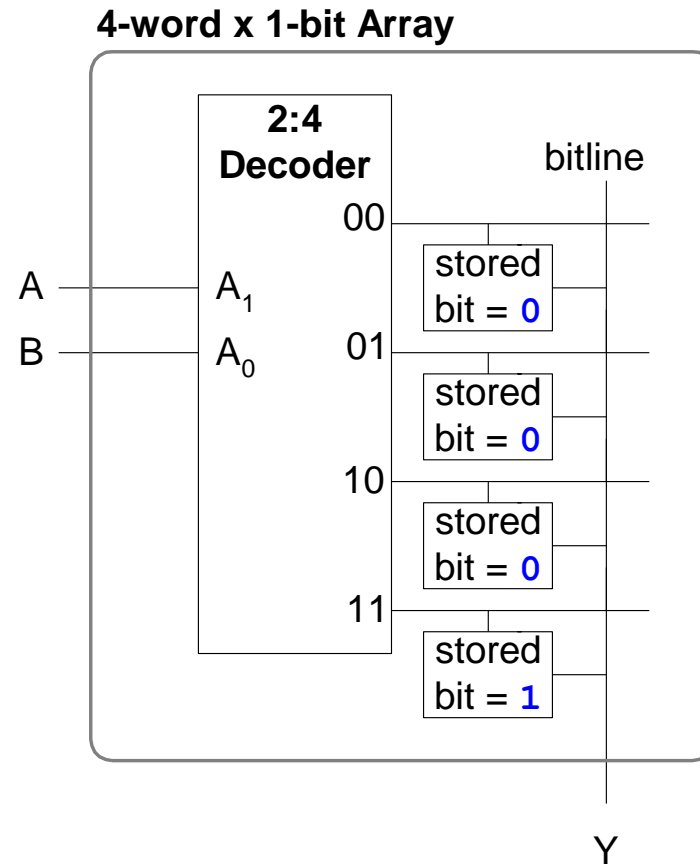
A, B $\xrightarrow{2/}$



Logic with Memory Arrays

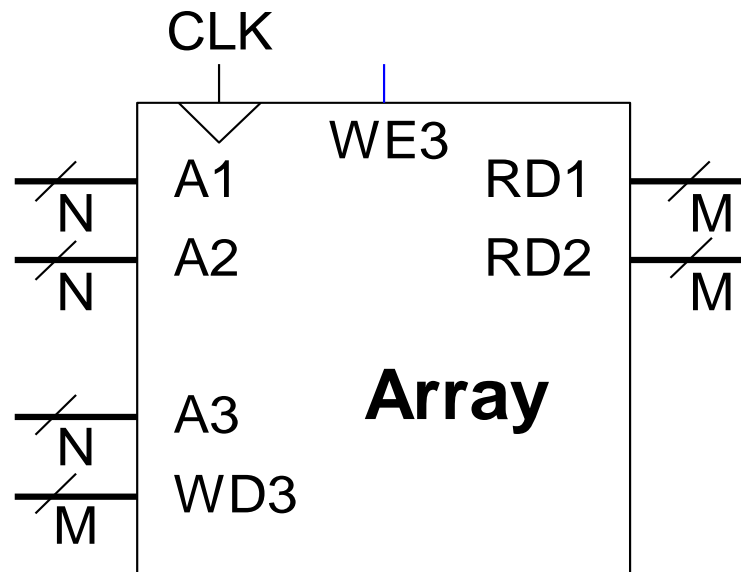
Called *lookup tables* (LUTs): look up output at each input combination (address)

Truth Table		
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



Multi-ported Memories

- **Port:** address/data pair
- 3-ported memory
 - 2 read ports (A1/RD1, A2/RD2)
 - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory



SystemVerilog Memory Arrays

```
// 256 x 3 memory module with one read/write port
module dmem( input  logic      clk, we,
              input  logic[7:0]  a
              input  logic [2:0] wd,
              output logic [2:0] rd);

    logic [2:0] RAM[255:0];

    assign rd = RAM[a];

    always @(posedge clk)
        if (we)
            RAM[a] <= wd;
endmodule
```

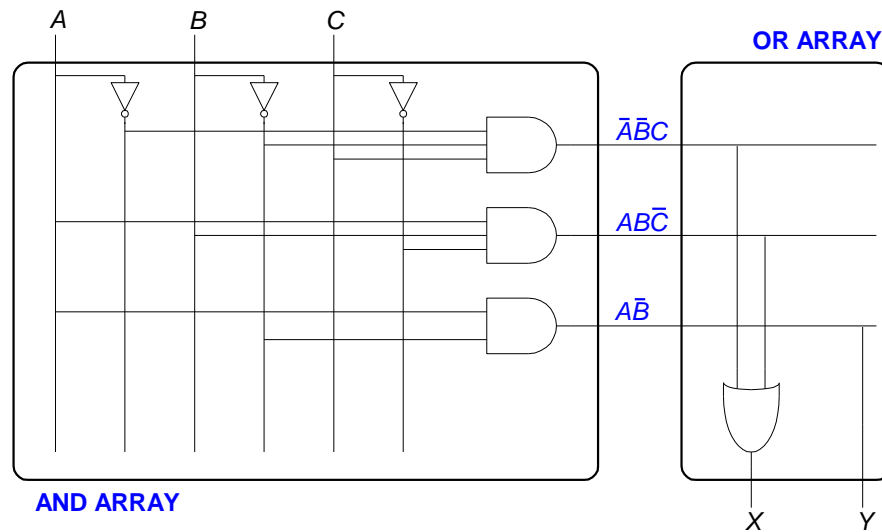
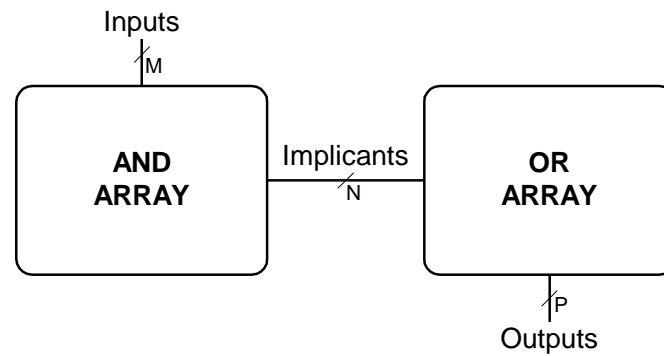
Logic Arrays

- **PLAs** (Programmable logic arrays)
 - AND array followed by OR array
 - Combinational logic only
 - Fixed internal connections
- **FPGAs** (Field programmable gate arrays)
 - Array of Logic Elements (LEs)
 - Combinational and sequential logic
 - Programmable internal connections

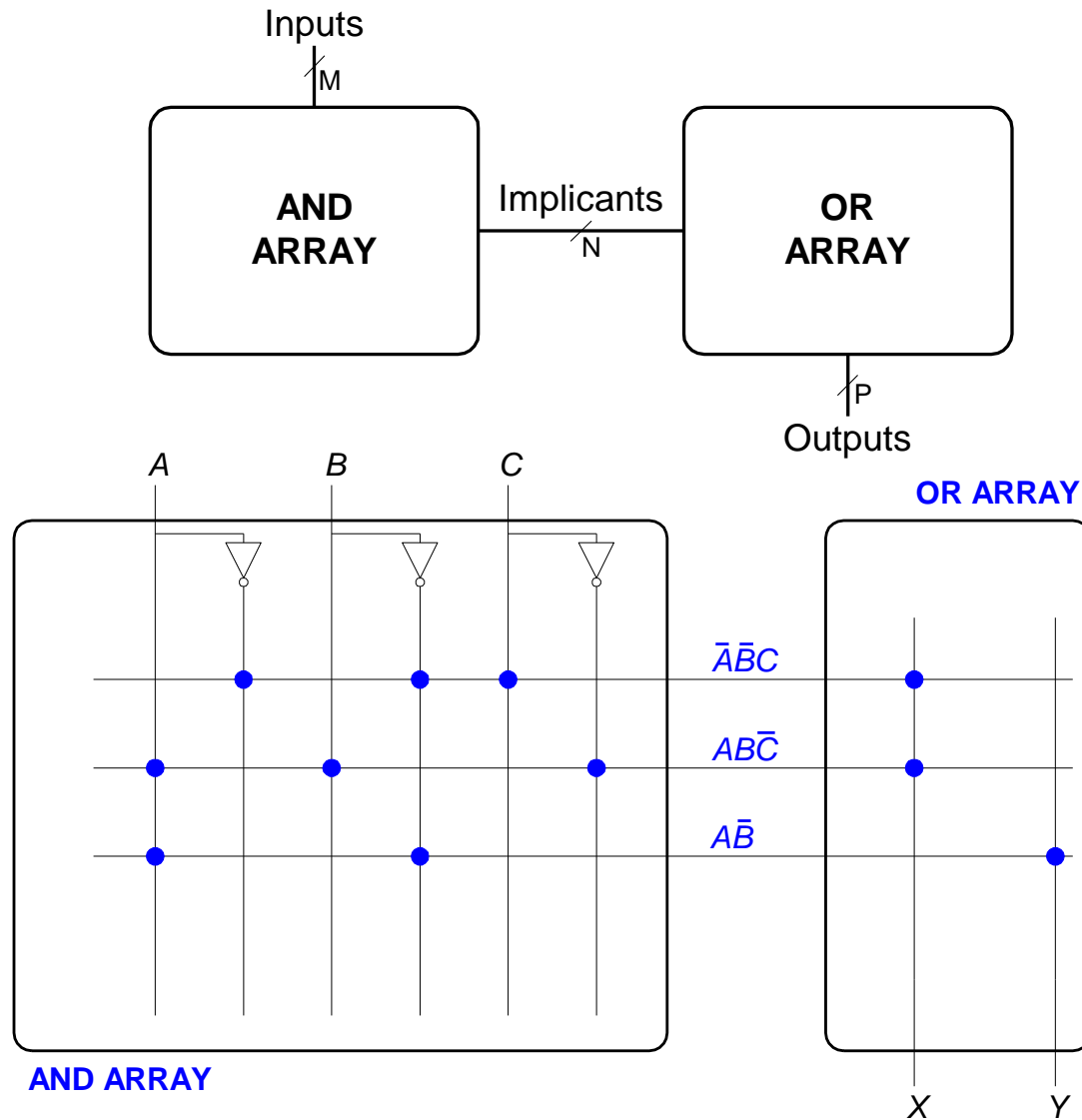
PLAs

PLAs

- $X = \bar{A}\bar{B}C + ABC\bar{C}$
- $Y = A\bar{B}$



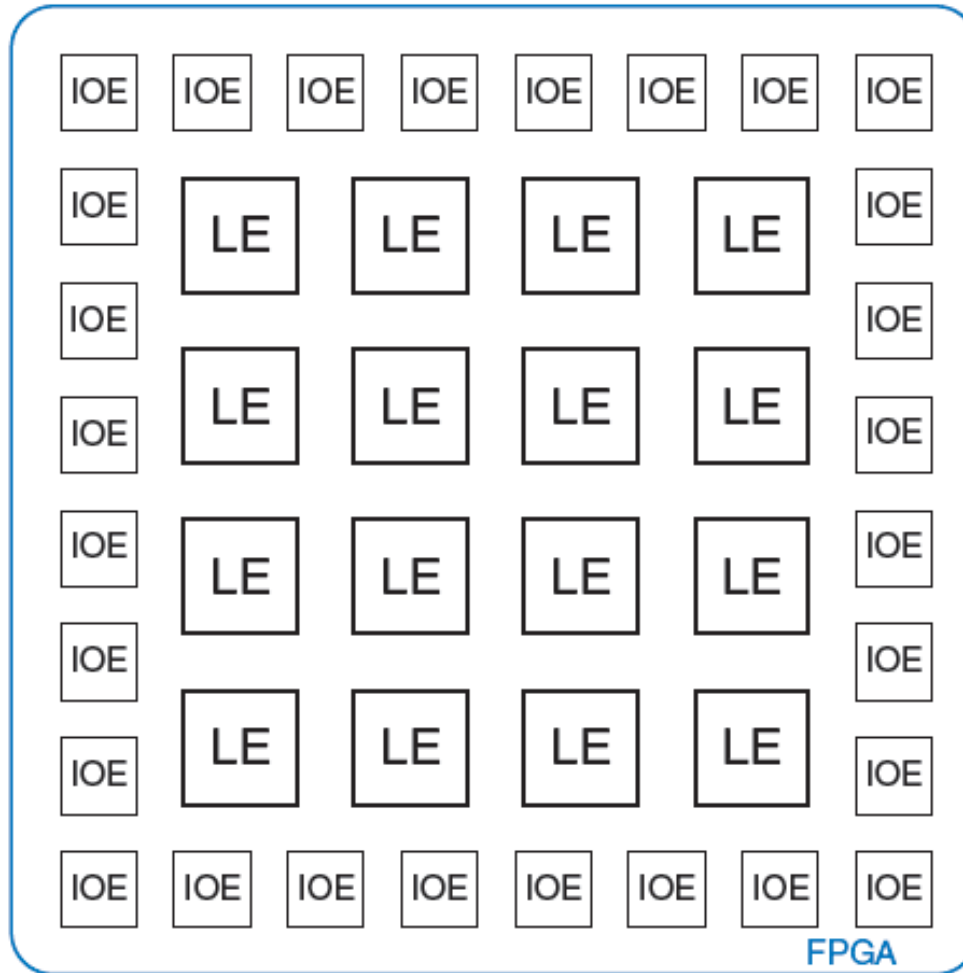
PLAs: Dot Notation



FPGA: Field Programmable Gate Array

- Composed of:
 - **LEs** (Logic elements): perform logic
 - **IOEs** (Input/output elements): interface with outside world
 - **Programmable interconnection:** connect LEs and IOEs
 - Some FPGAs include other building blocks such as multipliers and RAMs

General FPGA Layout



CLB : Configurable
Logic Block

IOB: Input-Output
Block

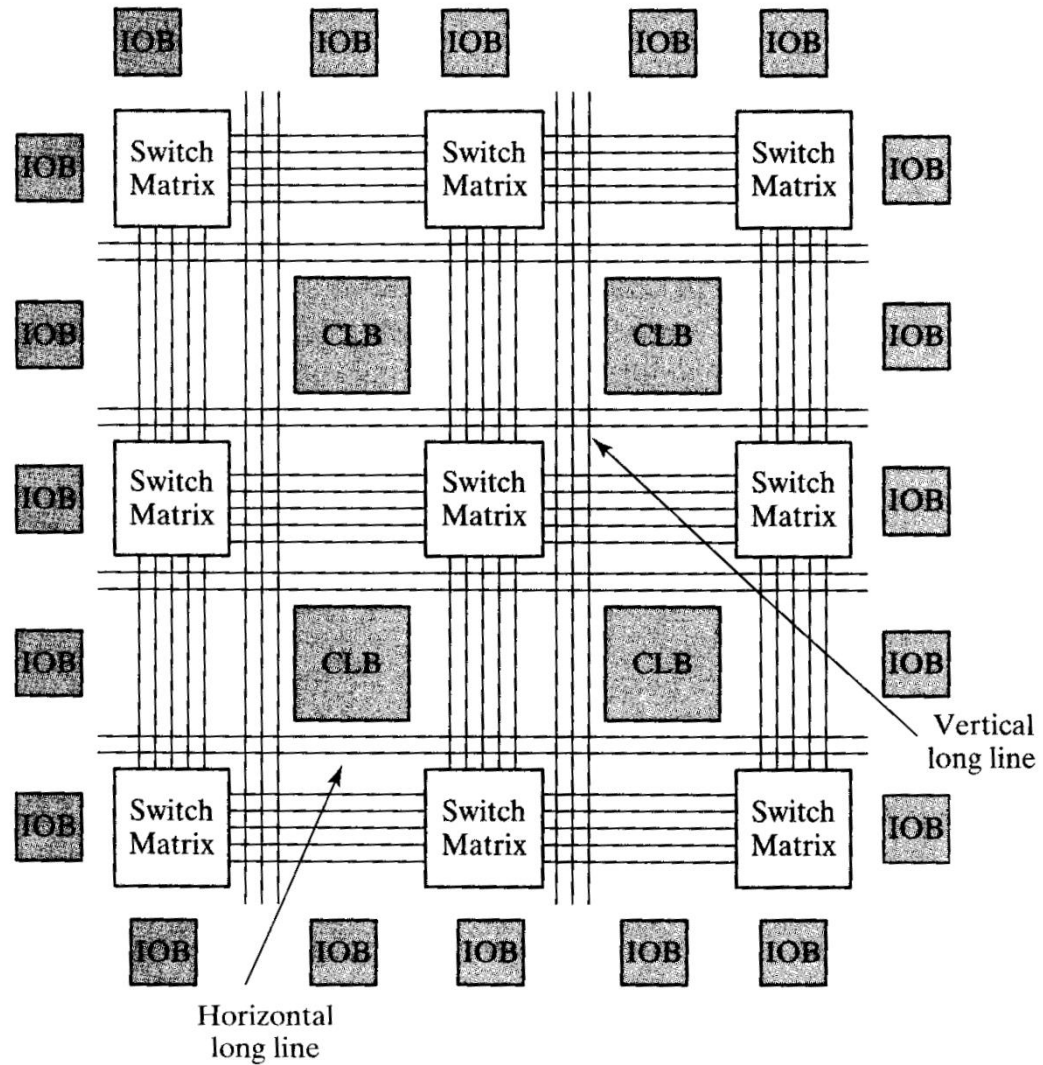
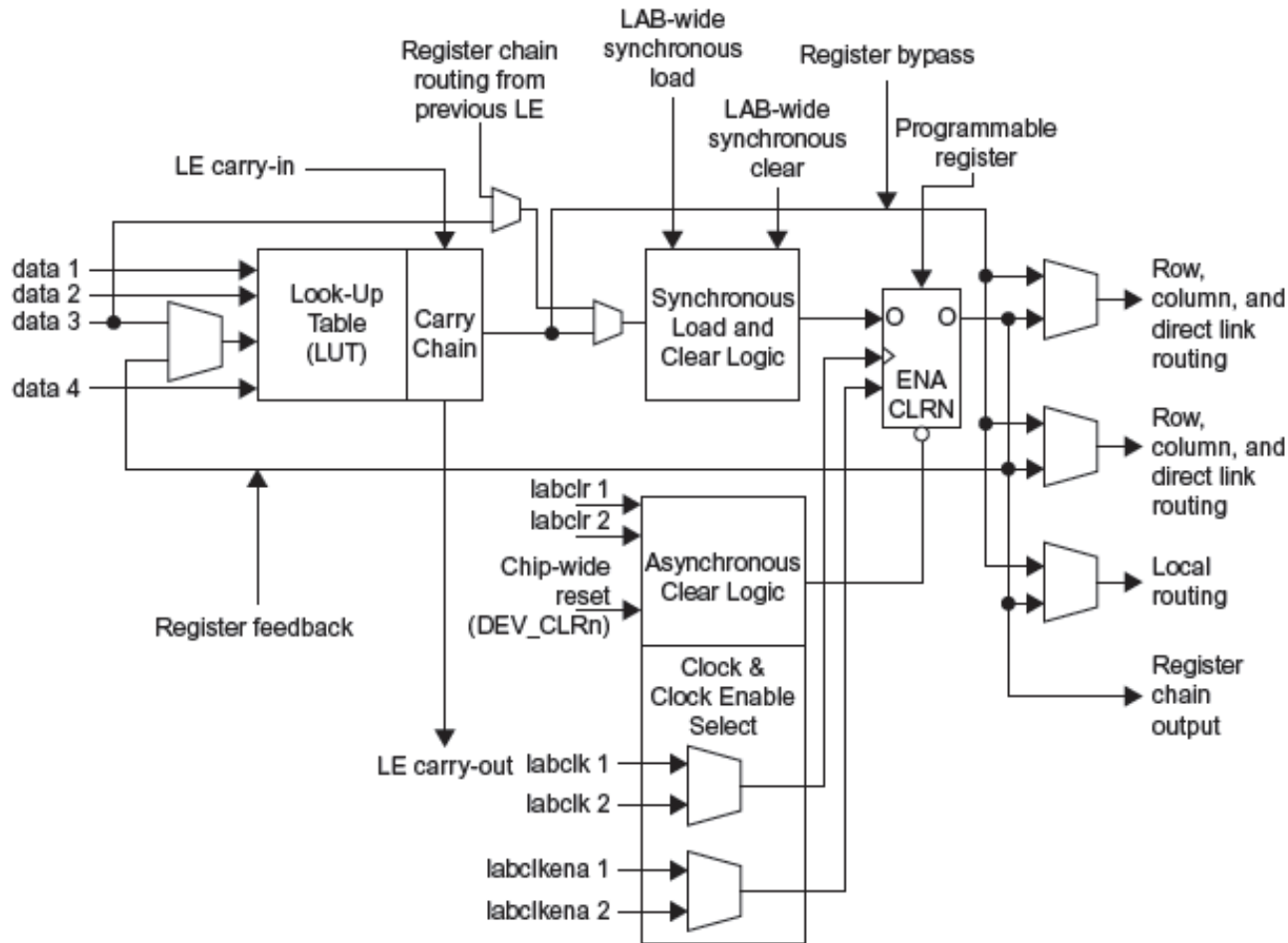


FIGURE 7.21
Basic architecture of Xilinx Spartan and predecessor devices

LE: Logic Element

- Composed of:
 - **LUTs** (lookup tables): perform combinational logic
 - **Flip-flops**: perform sequential logic
 - **Multiplexers**: connect LUTs and flip-flops

Altera Cyclone IV LE



Altera Cyclone IV LE

- The Spartan CLB has:
 - 1 four-input LUT
 - 1 registered output
 - 1 combinational output

LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

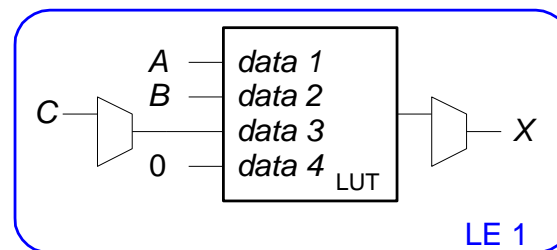
- $X = \overline{A}\overline{B}C + A\overline{B}C$
- $Y = A\overline{B}$

LE Configuration Example

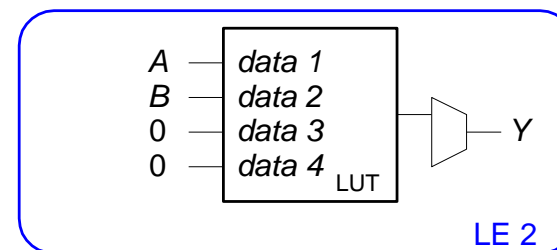
Show how to configure a Cyclone IV LE to perform the following functions:

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

(A) data 1	(B) data 2	(C) data 3	data 4	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

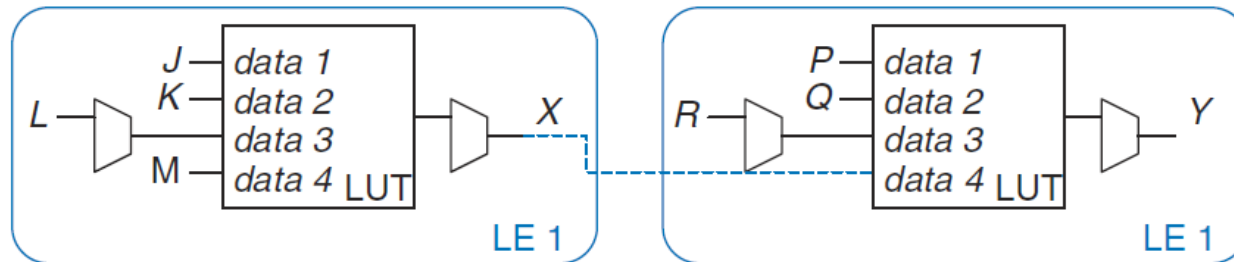


(A) data 1	(B) data 2	data 3	data 4	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0



Exercise

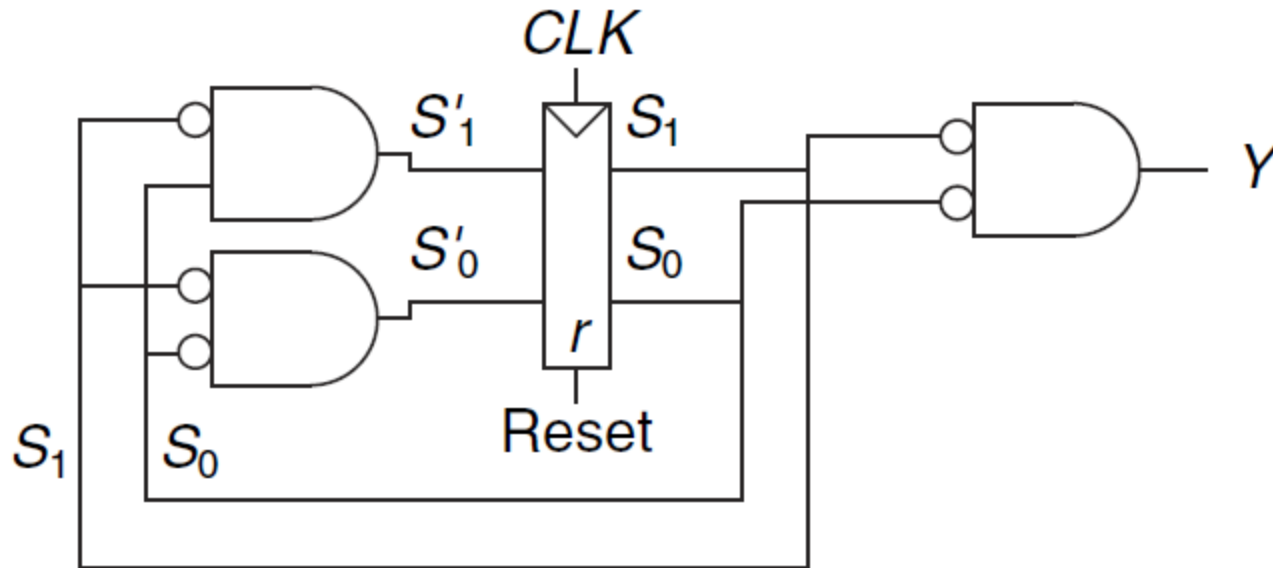
Implement the function $Y = JKLM PQR$ using Cyclone IV LEs.



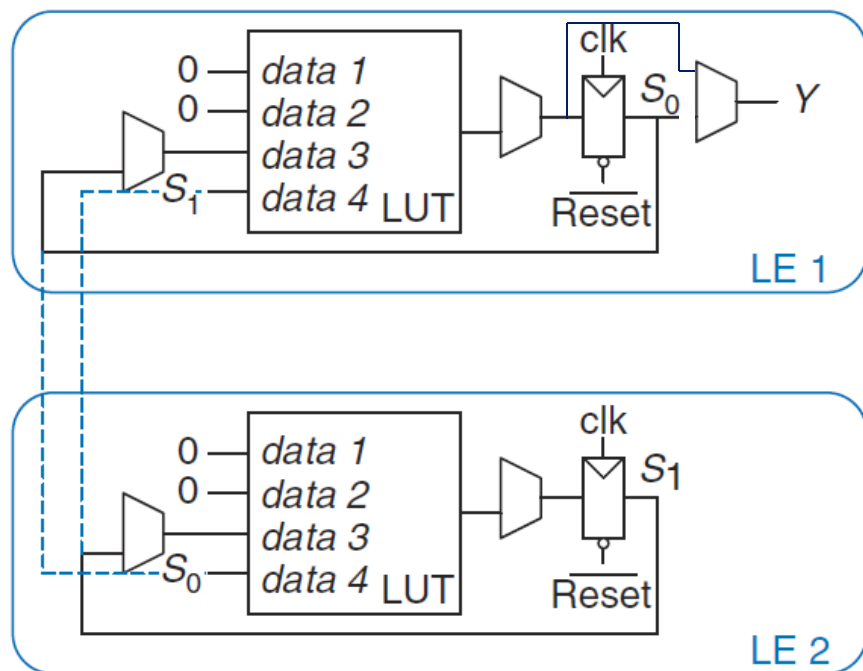
(J)	(K)	(L)	(M)	(X)	(P)	(Q)	(R)	(X)	(Y)
data 1	data 2	data 3	data 4	LUT output	data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	0	0	0	0	1	0	0
0	0	1	1	0	0	0	1	1	0
0	1	0	0	0	0	1	0	0	0
0	1	0	1	0	0	1	0	1	0
0	1	1	0	0	0	1	1	0	0
0	1	1	1	0	0	1	1	1	0
1	0	0	0	0	1	0	0	0	0
1	0	0	1	0	1	0	0	1	0
1	0	1	0	0	1	0	1	0	0
1	0	1	1	0	1	0	1	1	0
1	1	0	0	0	1	1	0	0	0
1	1	0	1	0	1	1	0	1	0
1	1	1	0	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1

Exercise

Implement the following sequential circuit using Cyclone IV LEs



Solution



<i>data 1</i>	<i>data 2</i>	(S_0) <i>data 3</i>	(S_1) <i>data 4</i>	(S_0') LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

<i>data 1</i>	<i>data 2</i>	(S_1) <i>data 3</i>	(S_0) <i>data 4</i>	(S_1') LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

FPGA Design Flow

Using a CAD tool (such as Altera's Quartus II)

- **Enter the design** using schematic entry or an HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design