Introduction to Digital Design

Week 9: Registers, Counters, and Shifters, and Arithmetic

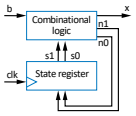
Yao Zheng
Assistant Professor
University of Hawai'i at Mānoa
Department of Electrical Engineering

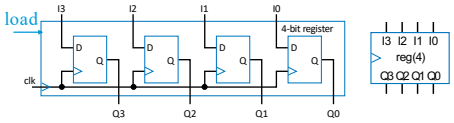
Overview

- Register in Depth
  - Parallel load and Buses.
  - Shift Register, Rotate Register, Multifunction Register.
  - Register design.
- Arithmetic
  - Adders and Incrementer.
  - Comparator.
  - Multiplier.
  - Subtractor (Two's complement in depth).
  - ALU
- Shifter and Barrel Shifter
- Counter and Timer

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



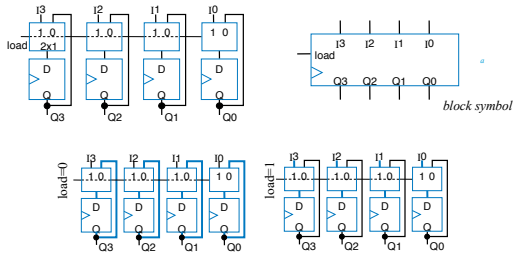


Basic register loads on every clock cycle

How extend to only load on certain cycles?

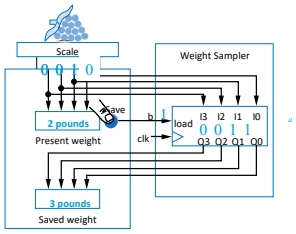
Register with Parallel Load

- 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



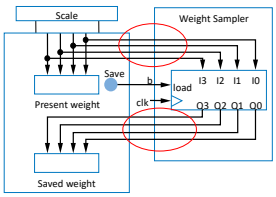
Register Example using the Load Input: Weight Sampler

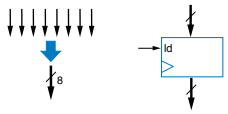
- Scale has two displays
  - Present weight
  - Saved weight
  - Useful to compare present item with previous item
- Use 4-bit parallel load register to store weight
  - Pressing button loads present weight into register
    - Register contents always displayed as "Saved weight," even when new present weight appears



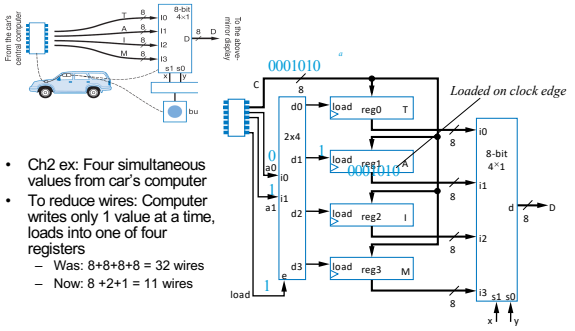
Buses

- N-bit *bus*: N wires to carry N-bit data item
  - Circuit drawings can become cluttered
- Convention for drawing buses
  - Single bold line and/or small angled line across





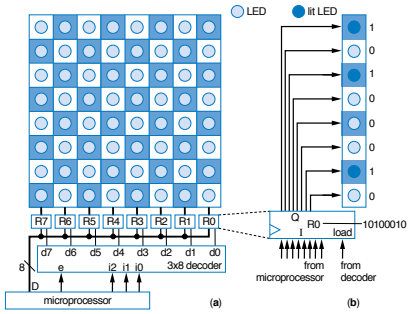
Register Example: Above-Mirror Display



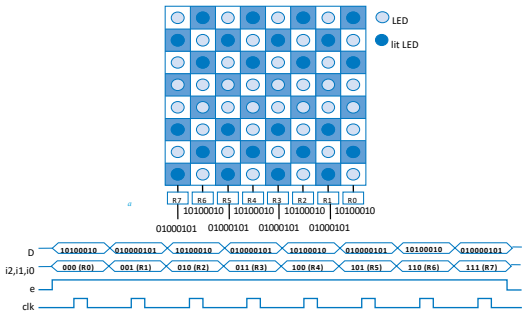
- Ch2 ex: Four simultaneous values from car's computer
- To reduce wires: Computer writes only 1 value at a time, loads into one of four registers
  - Was: 8+8+8+8 = 32 wires
  - Now: 8 +2+1 = 11 wires

Register Example: Computerized Checkerboard

- Each register holds values for one column of LEDs
  - "1" lights LED
- Microprocessor loads one register at a time
  - Occurs fast enough that user sees entire board change at once

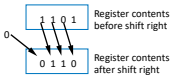


Register Example: Computerized Checkerboard



Shift Register

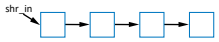
- Shift right
  - Move each bit one position right
  - Rightmost bit is "dropped"
  - Assume 0 shifted into leftmost bit



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

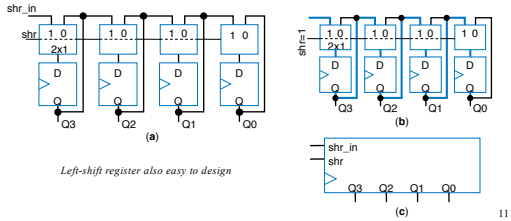
A:
1001 (original)
0100
0010
0001
0000

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



Shift Register

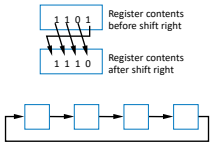
- To allow register to either shift or retain, use 2x1 muxes
  - shr: "0" means retain, "1" shift
  - shr_in: value to shift in
    - May be 0, or 1



Left-shift register also easy to design

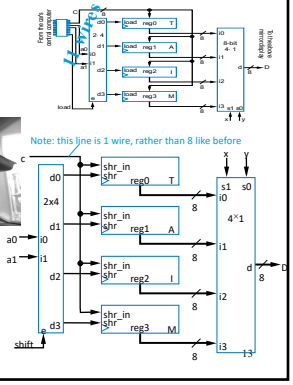
Rotate Register

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



Shift Register Example: Above-Mirror Display

- Earlier example: 8+2+1 = 11wires from car's computer to above-mirror display's four registers
  - Better than 32 wires, but 11 still a lot—want fewer for smaller wire bundles
- Use shift registers
  - Wires: 1+2+1=4
  - Computer sends one value at a time, one bit per clock cycle

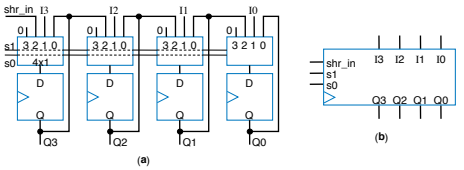


Multifunction Registers

- Many registers have multiple functions
  - Load, shift, clear (load all 0s)
  - And retain present value, of course
- Easily designed using muxes
  - Just connect each mux input to achieve desired function

Functions:

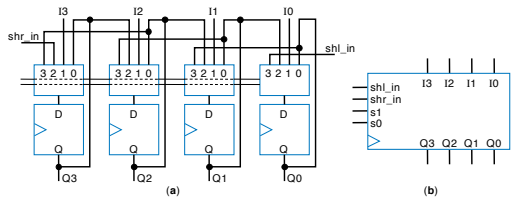
s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	(unused - let's load 0s)



14

Multifunction Registers

s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	Shift left



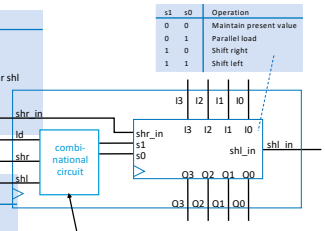
15

Multifunction Registers with Separate Control Inputs

ld	shr	shl	Operation
0	0	0	Maintain present value
0	0	1	Shift left
0	1	0	Shift right
0	1	1	Shift right – shr has priority over shl
1	0	0	Parallel load
1	0	1	Parallel load – ld has priority
1	1	0	Parallel load – ld has priority
1	1	1	Parallel load – ld has priority

Truth table for combinational circuit

Inputs ld	shr	shl	Outputs s1	s0	Note Operation
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load



$s1 = ld'*shr'*shl + ld'*shr*shl' + ld'*shr*shl$

$s0 = ld'*shr'*shl + ld$

16

Register Operation Table

- Register operations typically shown using compact version of table
  - X means same operation whether value is 0 or 1
    - One X expands to two rows
    - Two Xs expand to four rows
  - Put highest priority control input on left to make reduced table simple

Inputs ld	shr	shl	Outputs s1	s0	Note Operation
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load

ld	shr	shl	Operation
0	0	0	Maintain value
0	0	1	Shift left
0	1	X	Shift right
1	X	X	Parallel load

17

Register Design Process

- Can design register with desired operations using simple four-step process

TABLE 4.1 Four-step process for designing a multifunction register.

	Step	Description
1.	*Determine mux size*	Count the number of operations (don't forget the maintain present value operation!) and add in front of each flip-flop a mux with at least that number of inputs.
2.	*Create mux operation table*	Create an operation table defining the desired operation for each possible value of the mux select lines.
3.	*Connect mux inputs*	For each operation, connect the corresponding mux data input to the appropriate external input or flip-flop output (possibly passing through some logic) to achieve the desired operation.
4.	*Map control lines*	Create a truth table that maps external control lines to the internal mux select lines, with appropriate priorities, and then design the logic to achieve that mapping

18

Register Design Example

- Desired register operations
  - Load, shift left, synchronous clear, synchronous set
  - Want unique control input for each operation

s2	s1	s0	Operation
0	0	0	Maintain present value
0	0	1	Parallel load
0	1	0	Shift left
0	1	1	Synchronous clear
1	0	0	Synchronous set
1	0	1	Maintain present value
1	1	0	Maintain present value
1	1	1	Maintain present value

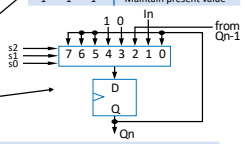
Step 1: Determine mux size

5 operations: above, plus maintain present value (don't forget this one!)

→ **Use 8x1 mux**

Step 2: Create mux operation table

Step 3: Connect mux inputs



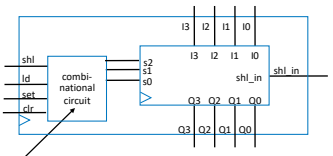
Step 4: Map control lines

s2 = clr'*set
s1 = clr'*set'*ld'*shl + clr
s0 = clr'*set'*ld + clr

Inputs				Outputs			
clr	set	ld	shl	s2	s1	s0	Operation
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s

19

Register Design Example



Step 4: Map control lines

s2 = clr'*set
s1 = clr'*set'*ld'*shl + clr
s0 = clr'*set'*ld + clr

Inputs				Outputs			
clr	set	ld	shl	s2	s1	s0	Operation
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s

20

4.3

Adders

- Adds two N-bit binary numbers
  - 2-bit adder: adds two 2-bit numbers, outputs 3-bit result
  - e.g., 01 + 11 = 100 (1 + 3 = 4)
- Can design using combinational design process of Ch 2, but doesn't work well for typical N
  - Why not?

Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

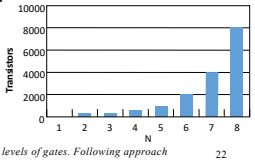
21

4.3

Why Adders Aren't Built Using Standard Combinational Design Process

- Truth table too big
  - 2-bit adder's truth table shown
    - Has $2^{(2+2)}$ = 16 rows
  - 8-bit adder: $2^{(8+8)}$ = 65,536 rows
  - 16-bit adder: $2^{(16+16)}$ = ~4 billion rows
  - 32-bit adder: ...
- Big truth table with numerous 1s/0s yields big logic
  - Plot shows number of transistors for N-bit adders, using state-of-the-art automated combinational design tool

Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0



Q: Predict number of transistors for 16-bit adder

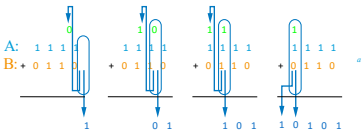
A: 1000 transistors for N=5, doubles for each increase of N. So transistors = $1000*2^{(N-5)}$. Thus, for N=16, transistors = $1000*2^{(16-5)} = 1000*2048 = 2{,}048{,}000$. Way too many!

Size comes from implementing with two levels of gates. Following approach uses more levels to achieve smaller size.

22

Alternative Method to Design an Adder: Imitate Adding by Hand

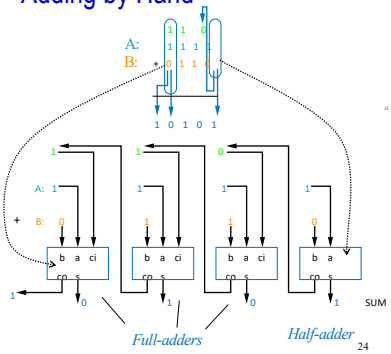
- Alternative adder design: mimic how people do addition by hand
- One column at a time
  - Compute sum, add carry to next column



23

Alternative Method to Design an Adder: Imitate Adding by Hand

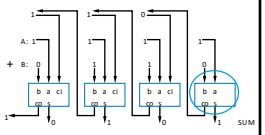
- Create component for each column
  - Adds that column's bits, generates sum and carry bits



24

Half-Adder

- ***Half-adder***: Adds 2 bits, generates sum and carry
- Design using combinational design process from Ch 2



Step 1: Capture the function

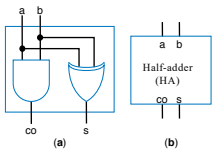
Inputs		Outputs	
a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Step 2A: Create equations

co = ab

s = a'b + ab' (same as s = a xor b)

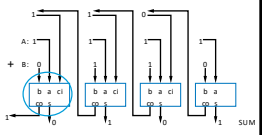
Step 2B: Implement as circuit



25

Full-Adder

- ***Full-adder:*** Adds 3 bits, generates sum and carry
- Design using combinational design process from Ch 2



Step 1: Capture the function

Inputs			Outputs	
a	b	ci	co	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

Step 2A: Create equations

co = a'bc + ab'c + abc' + abc

co = a'bc +abc +ab'c +abc +abc' +abc

co = (a'+a)bc + (b'+b)ac + (c'+c)ab

co = bc + ac + ab

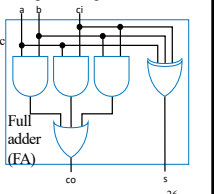
s = a'b'c + a'bc' + ab'c' + abc

s = a'(b'c + bc') + a(b'c' + bc)

s = a'(b xor c)' + a(b xor c)

s = a xor b xor c

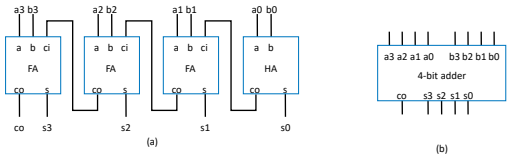
Step 2B: Implement as circuit



26

Carry-Ripple Adder

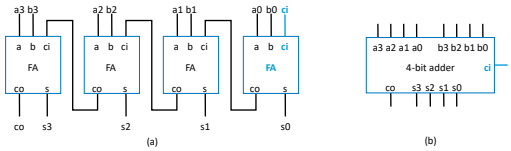
- Using half-adder and full-adders, we can build adder that adds like we would by hand
- Called a ***carry-ripple adder***
  - 4-bit adder shown: Adds two 4-bit numbers, generates 5-bit output
    - 5-bit output can be considered 4-bit "sum" plus 1-bit "carry out"
  - Can easily build any size adder



27

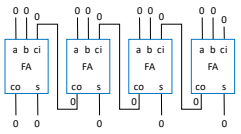
Carry-Ripple Adder

- Using full-adder instead of half-adder for first bit, we can include a "carry in" bit in the addition
  - Useful later when we connect smaller adders to form bigger adders

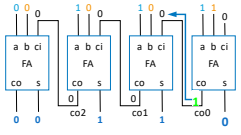


28

Carry-Ripple Adder's Behavior



Assume all inputs initially 0



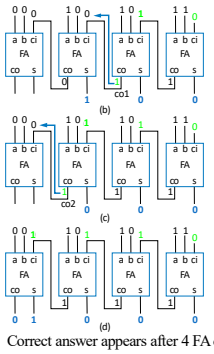
0111+0001
(answer should be 01000)

Output a fter 2 ns (1 FA del ay)

Wrong answer—is there a problem? No—just need more time for carry to ripple through the chain of full adders.

29

Carry-Ripple Adder's Behavior



0111+0001
(answer should be 01000)

Outputs after 4ns (2 FA delays)

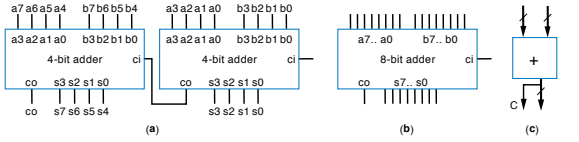
Outputs after 6ns (3 FA delays)

Output after 8ns (4 FA delays)

Correct answer appears after 4 FA delays

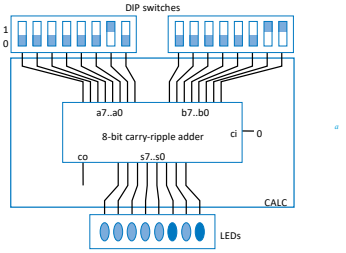
30

Cascading Adders



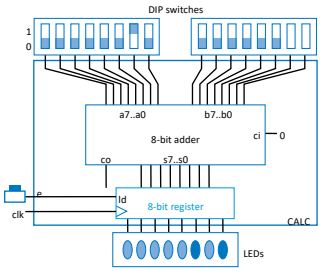
Adder Example: DIP-Switch-Based Adding Calculator

- Goal: Create calculator that adds two 8-bit binary numbers, specified using DIP switches
  - DIP switch: Dual-inline package switch, move each switch up or down
  - Solution: Use 8-bit adder



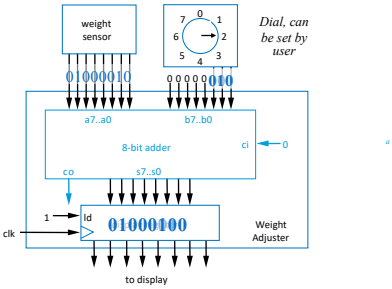
Adder Example: DIP-Switch-Based Adding Calculator

- To prevent spurious values from appearing at output, can place register at output
  - Actually, the light flickers from spurious values would be too fast for humans to detect—but the principle of registering outputs to avoid spurious values being read by external devices (which normally aren't humans) applies here.



Adder Example: Compensating Weight Scale

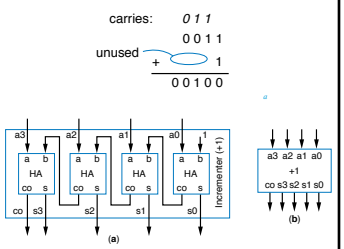
- Weight scale with compensation amount of 0-7
  - To compensate for inaccurate sensor due to physical wear
  - Use 8-bit adder



Incrementer

- Adds 1 to input A

Inputs				Outputs				
a3	a2	a1	a0	c0	s3	s2	s1	s0
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0	0
0	1	0	0	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	1	0	0	0	1	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	1	0
1	0	1	0	0	1	0	1	1
1	0	1	1	0	1	1	0	0
1	1	0	0	0	1	1	0	1
1	1	0	1	0	1	1	1	0
1	1	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0

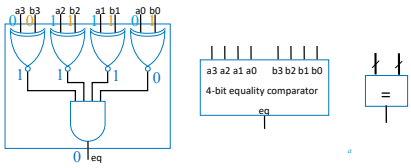


Could design using combinational design process, but smaller design uses carry-ripple, only need half-adders

Comparators

- ***N-bit equality comparator***: Outputs 1 if two N-bit numbers are equal
  - 4-bit equality comparator with inputs A and B
    - a3 must equal b3, a2 = b2, a1 = b1, a0 = b0
      - Two bits are equal if both 1, or both 0
      - eq = (a3b3 + a3'b3') * (a2b2 + a2'b2') * (a1b1 + a1'b1') * (a0b0 + a0'b0')
    - Note that function inside parentheses is XNOR
      - eq = (a3 xnor b3) * (a2 xnor b2) * (a1 xnor b1) * (a0 xnor b0)

0110 = 0111 ?



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 a3 and b3. If equal, compare a2 and b2. 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

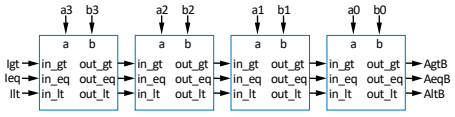
1011 1001 Equal

1011 1001 **Not equal**

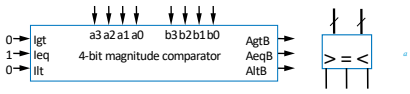
So A > B

Magnitude Comparator

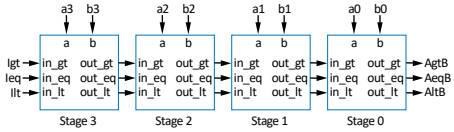
- By-hand example leads to idea for design
  - Start at left, compare each bit pair, pass results to the right
  - Each bit pair called a *stage*
  - Each stage has 3 inputs taking results of higher stage, outputs new results to lower stage



How design each stage?



Magnitude Comparator



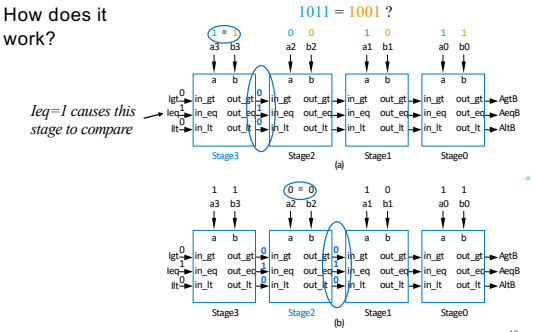
- Each stage:
  - out_gt = in_gt + (in_eq * a * b')
    - A>B if already determined in higher stage, or if higher stages equal but in this stage a=1 and b=0
  - out_lt = in_lt + (in_eq * a' * b)
    - A<B if already determined in higher stage, or if higher stages equal but in this stage a=0 and b=1
  - out_eq = in_eq * (a 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

- How does it work?

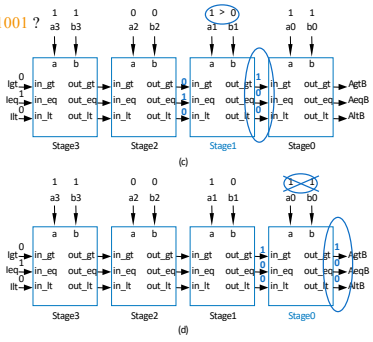
1011 = 1001 ?

Ieq=1 causes this stage to compare



Magnitude Comparator

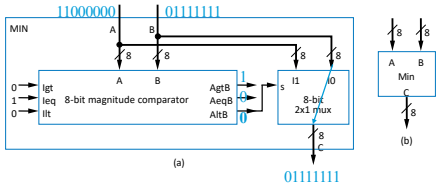
1011 = 1001 ?



- 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

Magnitude Comparator Example: Minimum of Two Numbers

- Design a combinational component that computes the minimum of two 8-bit numbers
  - Solution: Use 8-bit magnitude comparator and 8-bit 2x1 mux
    - If A<B, pass A through mux. Else, pass B.



Multiplier – Array Style

- Can build multiplier that mimics multiplication by hand
  - Notice that multiplying multiplicand by 1 is same as ANDing with 1

```
  0110   (the top number is called the multiplicand)
  0011   (the bottom number is called the multiplier)
  ----   (each row below is called a partial product)
  0110   (because the rightmost bit of the multiplier is 1, and 0110*1=0110)
 0110    (because the second bit of the multiplier is 1, and 0110*1=0110)
0000     (because the third bit of the multiplier is 0, and 0110*0=0000)
+0000    (because the leftmost bit of the multiplier is 0, and 0110*0=0000)
--------
00010010 (the product is the sum of all the partial products: 18, which is 6*3)
```

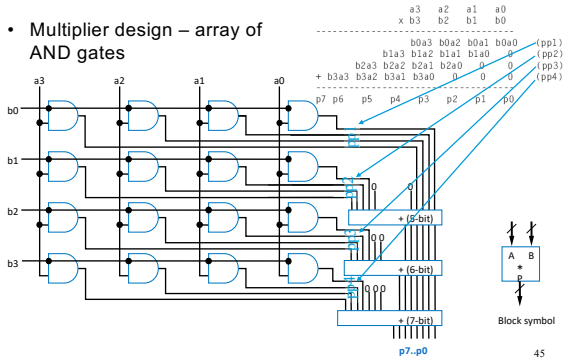

Multiplier – Array Style

- Generalized representation of multiplication by hand

```
                    a3   a2   a1   a0
                  x b3   b2   b1   b0
-------------------------------------
                  b0a3 b0a2 b0a1 b0a0   (pp1)
             b1a3 b1a2 b1a1 b1a0   0    (pp2)
        b2a3 b2a2 b2a1 b2a0   0    0    (pp3)
+ b3a3 b3a2 b3a1 b3a0   0    0    0    (pp4)
-------------------------------------
p7  p6   p5   p4   p3   p2   p1   p0
```

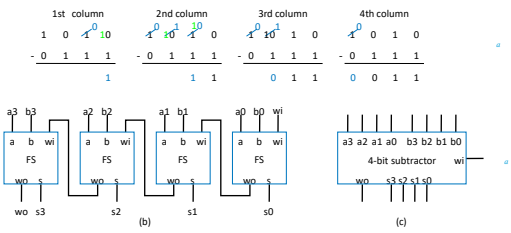

Multiplier – Array Style

- Multiplier design – array of AND gates



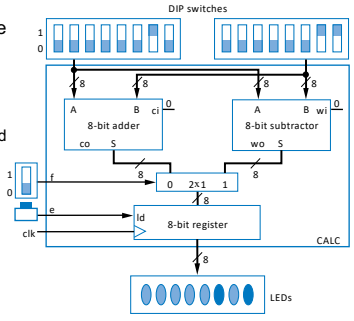
Subtractors and Signed Numbers

- Can build subtractor as we built carry-ripple adder
  - Mimic subtraction by hand
  - Compute the borrows from columns on left
    - Use full-subtractor component:
      - wi is borrow by column on right, wo borrow from column on left



Subtractor Example: DIP-Switch Based Adding/Subtracting Calculator

- Extend earlier calculator example
  - Switch f indicates whether want to add (f=0) or subtract (f=1)
  - Use subtractor and 2x1 mux



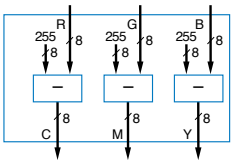
Subtractor Example: Color Space Converter – RGB to CMYK

- Color
  - Often represented as weights of three colors: red, green, and blue (RGB)
    - Perhaps 8 bits each (0-255), so specific color is 24 bits
      - White: R=11111111 (255), G=11111111, B=11111111
      - Black: R=00000000, G=00000000, B=00000000
      - Other colors: values in between, e.g., R=00111111, G=00000000, B=00001111 would be a reddish purple
  - Good for computer monitors, which mix red, green, and blue lights to form colors

- Printers use opposite color scheme
  - Because inks *absorb* light
  - Use complementary colors of RGB: Cyan (absorbs red), reflects green and blue, Magenta (absorbs green), and Yellow (absorbs blue)

Subtractor Example: Color Space Converter – RGB to CMYK

- Printers must quickly convert RGB to CMY
  - C=255-R, M=255-G, Y=255-B
  - Use subtractors as shown



49

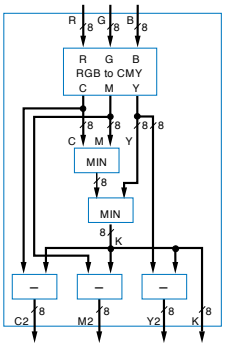
Subtractor Example: Color Space Converter – RGB to CMYK

- Try to save colored inks
  - Expensive
  - Imperfect – mixing C, M, Y doesn't yield good-looking black
- Solution: Factor out the black or gray from the color, print that part using black ink
  - e.g., CMY of (250,200,200)= (200,200,200) + (50,0,0).
    - (200,200,200) is a dark gray – use black ink

50

Subtractor Example: Color Space Converter – RGB to CMYK

- Call black part K
  - (200,200,200): K=200
  - (Letter "B" already used for blue)
- Compute minimum of C, M, Y values
  - Use MIN component designed earlier, using comparator and mux, to compute K
  - Output resulting K value, and subtract K value from C, M, and Y values
  - Ex: Input of (250,200,200) yields output of (50,0,0,200)



51

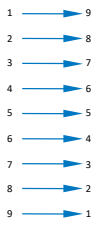
Representing Negative Numbers: Two's Complement

- Negative numbers common
  - How represent in binary?
- Signed-magnitude
  - Use leftmost bit for sign bit
    - So -5 would be:
      1101 using four bits
      10000101 using eight bits
- Better way: Two's complement
  - Big advantage: Allows us to perform subtraction using addition
  - Thus, only need adder component, no need for separate subtractor component

52

Ten's Complement

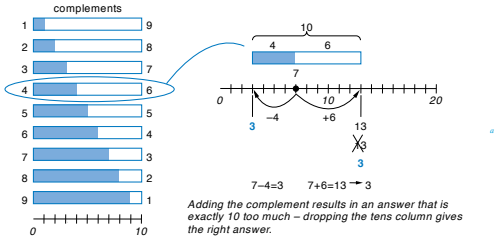
- Before introducing two's complement, let's consider ten's complement
  - But, be aware that computers DO NOT USE TEN'S COMPLEMENT. Introduced for intuition only.
  - Complements for each base ten number shown to right. Complement is the number that when added results in 10



53

Ten's Complement

- Nice feature of ten's complement
  - Instead of subtracting a number, adding its complement results in answer exactly 10 too much
  - So just drop the 1 – results in subtracting using addition only



Adding the complement results in an answer that is exactly 10 too much – dropping the tens column gives the right answer.

54

Two's Complement is Easy to Compute: Just Invert Bits and Add 1

- Hold on!
  - Sure, adding the ten's complement achieves subtraction using addition only
  - But don't we have to perform *subtraction* to have determined the complement in the first place? E.g., we only know that the complement of 4 is 6 by subtracting 10-4=6 in the first place.
- True. But in binary, it turns out that the two's complement can be computed **easily**
  - Two's complement of 011 is 101, because 011 + 101 is 1000
  - Could compute complement of 011 as 1000 – 011 = 101
  - **Easier method: Just invert all the bits, and add 1**
  - The complement of 011 is 100+1 = 101. It works!

Q: What is the two's complement of 0101? A: 1010+1=1011 (check: 0101+1011=10000)

Q: What is the two's complement of 0011? A: 1100+1=1101

55

Two's Complement

- Two's complement can represent negative numbers
  - Suppose have 4 bits
  - Positive numbers 0 to 7: 0000 to 0111
  - Negative numbers
    - -1: Take two's complement of 1: 0001 → 1110+1 = 1111
    - -2: 0010 → 1101+1 = 1110 …
    - -8: 1000 → 0111+1 = 1000
    - So -1 to -8: 1111 to 1000
  - Leftmost bit indicates sign of number, known as *sign bit*. 1 means negative.
- Signed vs. unsigned N-bit number
  - Unsigned: 0 to 2^N-1
    - Ex. Unsigned 8-bit: 0 to 255
  - Signed (two's complement): -2^{N-1} to 2^{N-1}-1
    - Ex. Signed 8-bit: -128 to 127

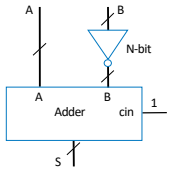
Quick method to determine magnitude of negative number—4-bit: subtract right 3 bits from 8. Ex. 1*110*: -(8 – 6) = -2

Or just take two's complement again: 1110 → -(0001+1) = -0010 = -2

56

Two's Complement Subtractor Built with an Adder

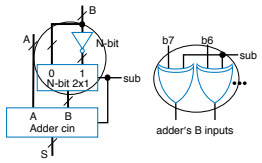
- Using two's complement
  A – B = A + (-B)
  = A + (two's complement of B)
  = A + invert_bits(B) + 1
- So build subtractor using adder by inverting B's bits, and setting carry in to 1



57

Adder/Subtractor

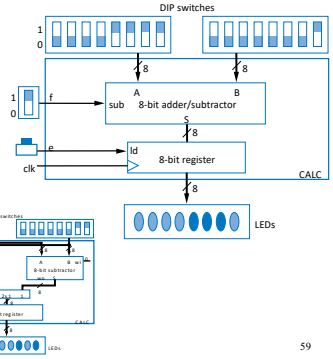
- Adder/subtractor: control input determines whether add or subtract
  - Can use 2x1 mux – sub input passes either B or inverted B
  - Alternatively, can use XOR gates – if sub input is 0, B's bits pass through; if sub input is 1, XOR inverts B's bits



58

Adder/Subtractor Example: Calculator

- Previous calculator used separate adder and subtractor
- Improve by using adder/subtractor, and two's complement numbers



59

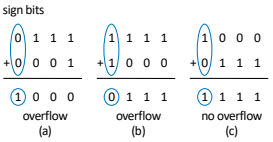
Overflow

- Sometimes result can't be represented with given number of bits
  - Either too large magnitude of positive or negative
  - Ex. 4-bit two's complement addition of 0111+0001 (7+1=8). But 4-bit two's complement can't represent number >7
    - 0111+0001 = 1000 WRONG answer, 1000 in two's complement is -8, not +8
  - Adder/subtractor should indicate when overflow has occurred, so result can be discarded

60

Detecting Overflow: Method 1

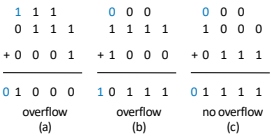
- For two's complement numbers, overflow occurs when the two numbers' sign bits are the same but differ from the result's sign bit
  - If the two numbers' sign bits are initially different, overflow is impossible
    - Adding positive and negative can't exceed largest magnitude positive or negative
- Simple overflow detection circuit for 4-bit adder
  - overflow = a3'b3's3 + a3b3s3'
  - Include "overflow" output bit on adder/subtractor



If the numbers' sign bits have the same value, which differs from the result's sign bit, overflow has occurred.

Detecting Overflow: Method 2

- Even simpler method: Detect difference between carry-in to sign bit and carry-out from sign bit
- Yields simpler circuit: overflow = c3 xor c4



If the carry into the sign bit column differs from the carry out of that column, overflow has occurred.

Arithmetic-Logic Unit: ALU

- ***ALU***: Component that can perform various arithmetic (add, subtract, increment, etc.) and logic (AND, OR, etc.) operations, based on control inputs

TABLE 4.2 Desired calculator operations

Inputs x	y	z	Operation	Sample output if A=00001111, B=00000101
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

Multifunction Calculator without an ALU

- Can build using separate components for each operation, and muxes
  - Too many wires, also wastes power computing operations when only use one result at given time

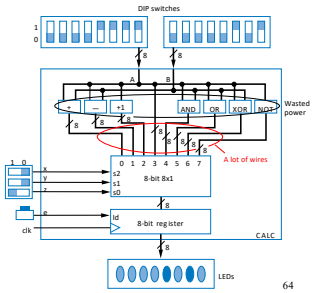
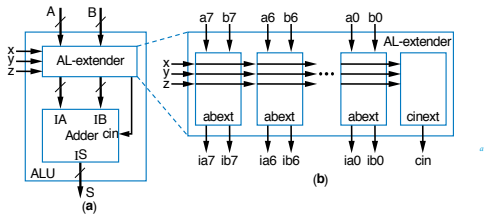


TABLE 4.2 Desired calculator operations

Inputs x	y	z	Operation	Sample output if A=00001111, B=00000101
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

ALU

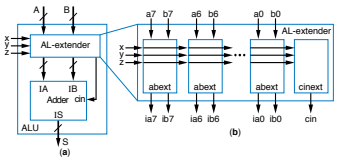
- More efficient design uses ALU
  - ALU design not just separate components multiplexed (same problem as previous slide)
  - Instead, ALU design uses single adder, plus logic in front of adder's A and B inputs
    - Logic in front is called an *arithmetic-logic extender*
  - Extender modifies A and B inputs so desired operation appears at output of the adder



Arithmetic-Logic Extender in Front of ALU

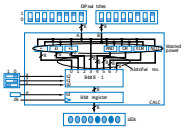
TABLE 4.2 Desired calculator operations

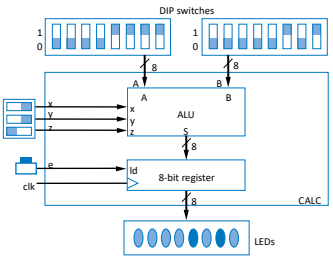
Inputs x	y	z	Operation	Sample output if A=00001111, B=00000101
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000



- xyz=000 Want S=A+B : just pass a to ia, b to ib, and set cin=0
- xyz=001 Want S=A-B : pass a to ia, b' to ib and set cin=1 (two's complement)
- xyz=010 Want S=A+1 : pass a to ia, set ib=0, and set cin=1
- xyz=011 Want S=A : pass a to ia, set ib=0, and set cin=0
- xyz=100 Want S=A AND B : set ia=a*b, b=0, and cin=0
- Others: likewise
- Based on above, create logic for ia(x,y,z,a,b) and ib(x,y,z,a,b) for each abext, and create logic for cin(x,y,z), to complete design of the AL-extender component

ALU Example: Multifunction Calculator



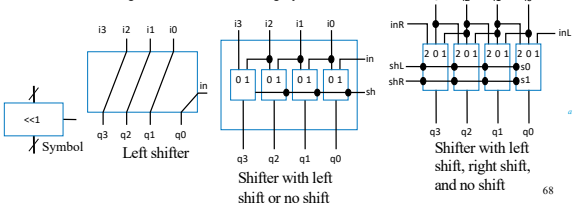


- Design using ALU is elegant and efficient
  - No mass of wires
  - No big waste of power

67

Shifters

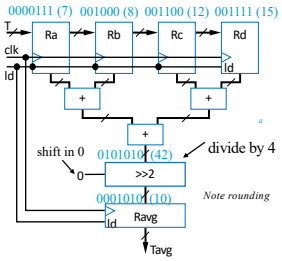
- Shifting (e.g., left shifting 0011 yields 0110) useful for:
  - Manipulating bits
  - Converting serial data to parallel (remember earlier above-mirror display example with shift registers)
  - Multiply/divide by 2 (unsigned numbers only)
    - Shift left once is same as multiplying by 2. Ex: 0011 (3) becomes 0110 (6)
      - Why? Essentially appending a 0 -- Note that multiplying decimal number by 10 accomplished just be appending 0, i.e., by shifting left (55 becomes 550)
    - Shift right once same as dividing by 2



68

Shifter Example: Temperature Averager

- Four registers storing a history of temperatures
- Want to output the average of those temperatures
- Add
  - Use three adders
- Then divide by four
  - Same as shift right by 2



69

Strength Reduction

- Many multiplications not by power of 2 constants (2, 4, 8, …)
  - E.g., by 5, or 10
  - Could use multiplier component, but shifters are small and fast
- Replace multiplication by shifts and adds
  - Operator *strength reduction* (multiply is "stronger" than shift/add)
  - E.g., 5*C → 4*C + C (4*C same as C<<2)
- Replacing division by shifts and adds slightly harder
  - Approximate fraction using fraction having power of 2 denominator
  - E.g., C/5 = 0.20*C, approx. equals (102/512)*C = 0.199*C
    - (102/512)*C = C*(64+32+4+2)/512 = (C*64 + C*32 + C*4+C*2)/512
    - = ((C<<6) + (C<<5) + (C<<2) + (C<<1)) >> 9

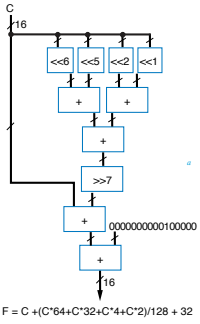
70

Celsius to Fahrenheit Converter

- F = C*9/5 + 32
  - F = C + C*4/5 + 32
  - F = C + 4*(C*64+C*32+C*4+C*2)/512 +32 (1/5 from prev. slide)
  - F = C + (C*64+C*32+C*4+C*2)/128+32

Note: Uses wider buses, padded on the left, to avoid losing bits during shifts

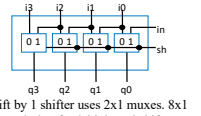
- Determine biggest possible internal values, set width accordingly
- Do divisions as late as possible to reduce rounding errors



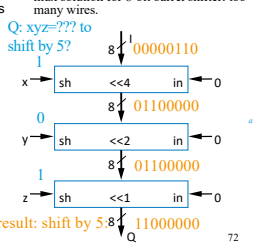
71

Barrel Shifter

- A shifter that can shift by any amount
  - 4-bit barrel left shift can shift left by 0, 1, 2, or 3 positions
  - 8-bit barrel left shifter can shift left by 0, 1, 2, 3, 4, 5, 6, or 7 positions
    - (Shifting an 8-bit number by 8 positions is pointless -- you just lose all the bits)
- Could design using 8x1 muxes
  - Too many wires
- More elegant design
  - Chain three shifters: 4, 2, and 1
  - Can achieve any shift of 0..7 by enabling the correct combination of those three shifters, i.e., shifts should sum to desired amount



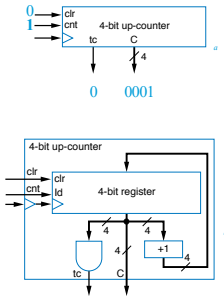
Shift by 1 shifter uses 2x1 muxes. 8x1 mux solution for 8-bit barrel shifter: too many wires.



72

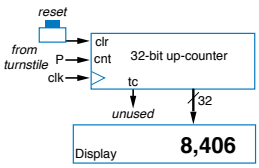
Counters and Timers

- ***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, equals1 during value just before rollover
- Internal design
  - Register, incrementer, and N-input AND gate to detect terminal count



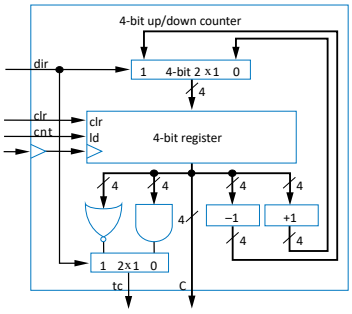
Counter Example: Turnstile with Display

- Count people passing through a turnstile
  - Assume turnstile generates 1-clock pulse on P
  - Connect count to display
  - Button can reset count
- Simple solution using up-counter



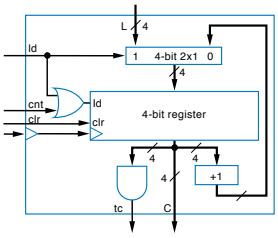
Up/Down-Counter

- 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)



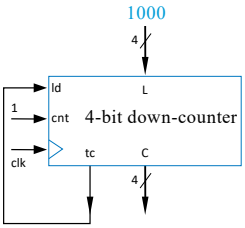
Counter with Load

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



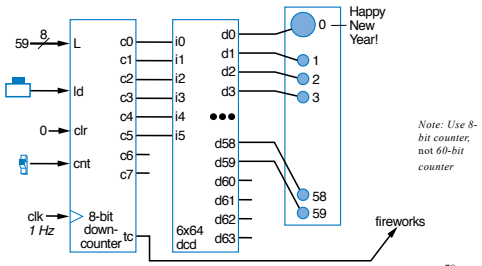
Counter with Parallel Load

- Useful to create pulses at specific multiples of clock
  - Not just at N-bit counter's natural wrap-around of 2^N
- Example: Pulse every 9 clock cycles
  - Use 4-bit down-counter with parallel load
  - Set parallel load input to 8 (1000)
  - Use terminal count to reload
    - When count reaches 0, next cycle loads 8.
  - Why load 8 and not 9? Because 0 is included in count sequence:
    - 8, 7, 6, 5, 4, 3, 2, 1, 0 → 9 counts



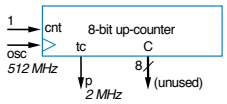
Counter Example:
New Year's Eve Countdown Display

- Chapter 2 example previously used microprocessor to count from 59 down to 0 in binary
- Can use 8-bit (or 7- or 6-bit) down-counter instead, initially loaded with 59



Common Counter Use: Clock Divider

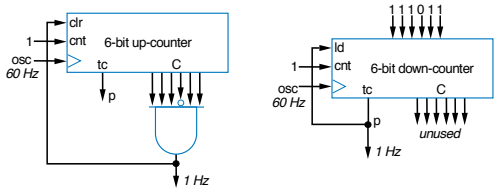
- Suppose have 512 Hz oscillator, but want 2 Hz pulse
  - Thus, want divide fast clock by 256
  - Design using 8-bit up-counter, use tc output as pulse
    - Counts from 0 to 255 (256 counts), so pulses tc every 256 cycles



79

Clock Division by Amount not Power of 2
Example: 1 Hz Pulse Generator from 60 Hz Clock

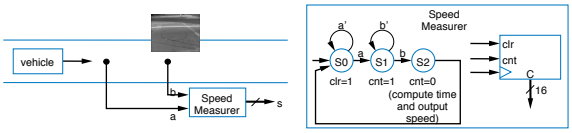
- Example: U.S. electricity standard uses 60 Hz signal
  - Device may convert that to 1 Hz signal to count seconds
- Use 6-bit up-counter
  - Up counter: Detected count of 59 clears counter to 0; tc thus pulses every 60 cycles
    - Note: Detect 59, not 60 (because the first number, 0, is included)
  - Down counter approach also possible: When count reaches 0, load 59



80

Measuring Time Between Events using an Up-Counter

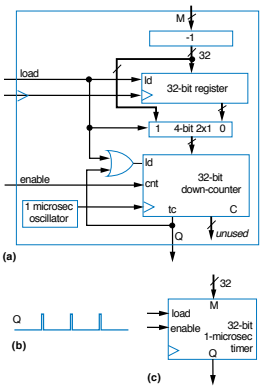
- Initially clear to 0. 1st event: Set cnt=1. 2nd event: Set cnt=0.
  - Then, multiply counted clock cycles by clock period to determine time
- Ex: Highway speed measurement system
  - Two sensors "a" and "b" in road
  - Use FSM to detect "a" becoming 1, set cnt=1. Set cnt=0 when "b" 1
  - If clock is 1 kHz (period is 1 ms), then time is C * 0.001s
    - If a and b separated by 0.01 miles, then vehicle speed in mph is: 0.01 miles / (time * (1 hour / 3600 seconds))
    - E.g., if C is 500, then speed is 0.01 / ((500*0.001) / 3600) = 72 mph



81

Timers

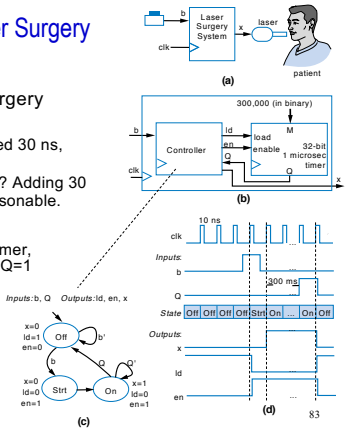
- Pulses output at user-specified timer interval when enabled
  - "Ticks" like a clock
  - Interval specified as multiple of base time unit
  - If base is 1 microsec and user wants pulse every 300 ms, loads 300,000 into timer
- Can design using oscillator, register, and down-counter



82

Timer Example: Laser Surgery System

- Recall Chpt 3 laser surgery example
  - Clock was 10 ns, wanted 30 ns, used 3 states.
  - What if wanted 300 ms? Adding 30 million states is not reasonable.
- Use timer
  - Controller FSM loads timer, enables, then waits for Q=1

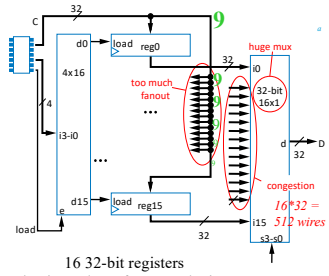


83

4.10

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

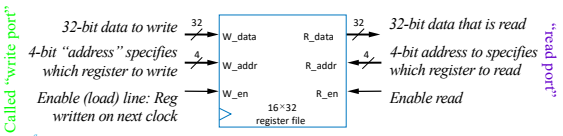


16 32-bit registers begins to have fanout and wire problems

84

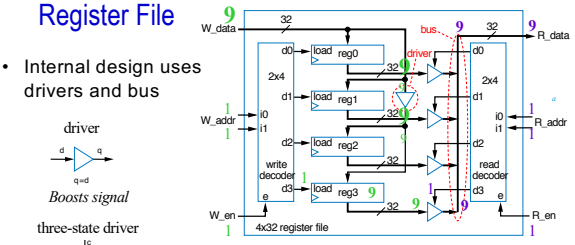
Register File

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



Register File

- Internal design uses drivers and bus

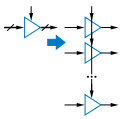


Internal design of 4x32 RF; 16x32 RF follows similarly

driver

Boosts signal

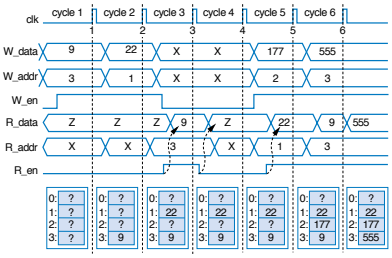
three-state driver



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

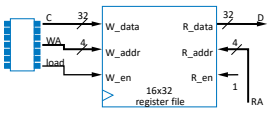
Register File Timing Diagram

- Can write one register and read one register each clock cycle
  - May be same register



Register-File Example: Above-Mirror Display

- 16 32-bit registers that can be written by car's computer, and displayed
  - Use 16x32 register file
  - Simple, elegant design
- Register file hides complexity internally
  - And because only one register needs to be written and/or read at a time, internal design is simple



Summary

- Register in Depth
  - Parallel load and Buses.
  - Shift Register, Rotate Register, Multifunction Register.
  - Register design.
- Arithmetic
  - Adders and Incrementer.
  - Comparator.
  - Multiplier.
  - Subtractor (Two's complement in depth).
  - ALU
- Shifter and Barrel Shifter
- Counter and Timers