ECE 152 / 496
Introduction to Computer Architecture

Arithmetic and ALU Design

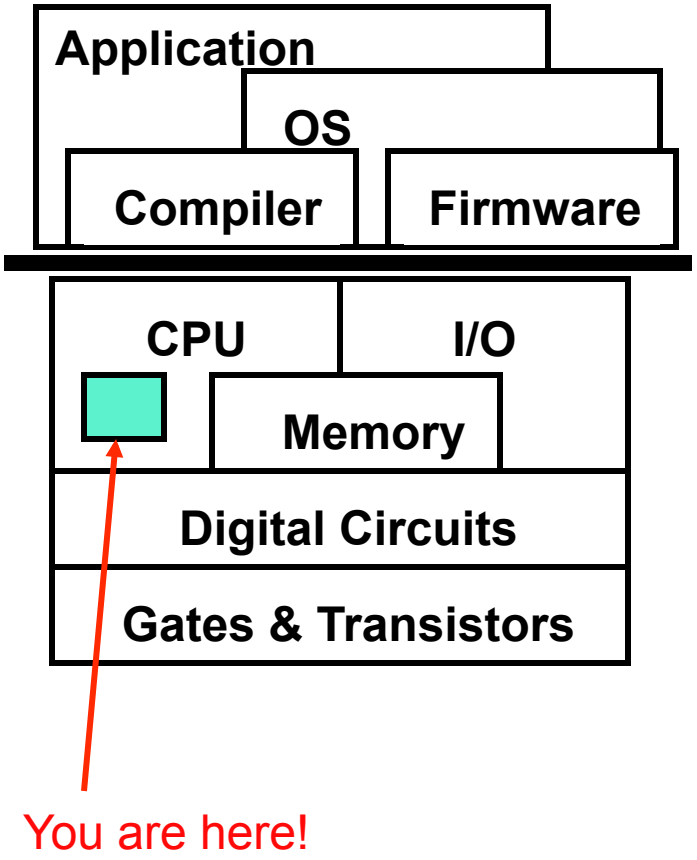
Benjamin C. Lee
Duke University

Slides from Daniel Sorin (Duke)
and are derived from work by
Amir Roth (Penn) and Alvy Lebeck (Duke)

Where We Are in This Course Right Now

- So far:
  - We know what a computer architecture is
  - We know what kinds of instructions it might execute
- Now:
  - We learn how to perform many of the most important instructions
    - Computers spend lots of time doing arithmetic and logical ops
  - Examples: add, subtract, multiply, divide, shift, rotate, load, store
  - We develop hardware for arithmetic logic unit (ALU)
- Next:
  - We learn how the computer uses and controls the ALU
  - Lots of stuff in computer besides the ALU
    - E.g., Logic to fetch and decode instructions, memory, etc.

This Unit: Arithmetic and ALU Design



- Integer Arithmetic and ALU
  - Binary number representations
  - Addition and subtraction
  - Integer ALU
  - Shifting and rotating
  - Multiplication
  - Division
- Floating Point Arithmetic
  - Binary number representations
  - FP arithmetic
  - Accuracy

Readings

- Patterson and Hennessy textbook
  - Chapter 3

Review: Fixed Width

- You've seen much of the upcoming material in ECE 52 – if none of this looks familiar, please talk with me …

- In hardware, integers have **fixed width**
  - N bits: 16, 32, or 64
  - LSB is 2^0, MSB is 2^{N-1}
  - **Unsigned number range**: 0 to 2^N–1
  - Numbers >2^N represented using multiple fixed-width integers
    - In software
    - ICQ: What happens when your C++ code specifies an integer greater than this max? What does compiler do?

Review: Two's Complement

- What about negative numbers?
  - Option I: **sign/magnitude**
    - Unsigned binary plus one bit for sign

      10_{10} = 00000101**0**, -10_{10} = 00000101**1**
    - Two representations for zero (0 and –0 are different)
    - Addition in hardware is difficult
    - **Number range**: $-(2^{N-1}-1)$ to $2^{N-1}-1$
    - \+ Matches our intuition from "by hand" decimal arithmetic
  - Option II: **two's complement (TC)**
    - leading 0s mean positive number, leading 1s negative

      10_{10} = 00001010, -10_{10} = 11110110
    - \+ One representation for 0
    - \+ Easy addition in hardware
    - **Number range**: $-(2^{N-1})$ to $2^{N-1}-1$ → not symmetric

Review: Still More On TC

- What is the interpretation of TC?
  - Same as binary, except **MSB represents -2^{N-1}**, not 2^{N-1}
    - $-10 = 11110110 = -2^7+2^6+2^5+2^4+2^2+2^1$
  - \+ Works with any width
    - $-10 = 110110 = -2^5+2^4+2^2+2^1$
    - Why? $2^N = 2*2^{N-1}$
    - $-2^5+2^4+2^2+2^1 = (-2^6+2*2^5)-2^5+2^4+2^2+2^1 = -2^6+2^5+2^4+2^2+2^1$

- Trick to negating a number quickly: **$-B = B' + 1$**
  - $-(1) = (0001)' +1 = 1110+1 = 1111 = -1$
  - $-(-1) = (1111)' +1 = 0000+1 = 0001 = 1$
  - $-(0) = (0000)' +1 = 1111+1 = 0000 = 0$
  - Think about why this works (on your own time)

Review (way back!): Decimal Addition

- Remember decimal addition from 1st grade?

```
  1
 43
+29
 72
```

  - Repeat N times
    - Add least significant digits and any overflow from previous add
    - Carry the overflow to next addition
      - **Overflow**: any digit other than least significant of sum
    - Shift two addends and sum one digit to the right
  - Sum of two N-digit numbers can yield an N+1 digit number

Review: Binary Addition

- Binary addition works the same way

```
  1      111111
 43 = 00101011
+29 = 00011101
 72 = 01001000
```

  - Repeat N times
    - Add least significant bits and any overflow from previous add
    - Carry the overflow to next addition
    - Shift two addends and sum one bit to the right
  - Sum of two N-bit numbers can yield an N+1 bit number

  – More steps (smaller base)

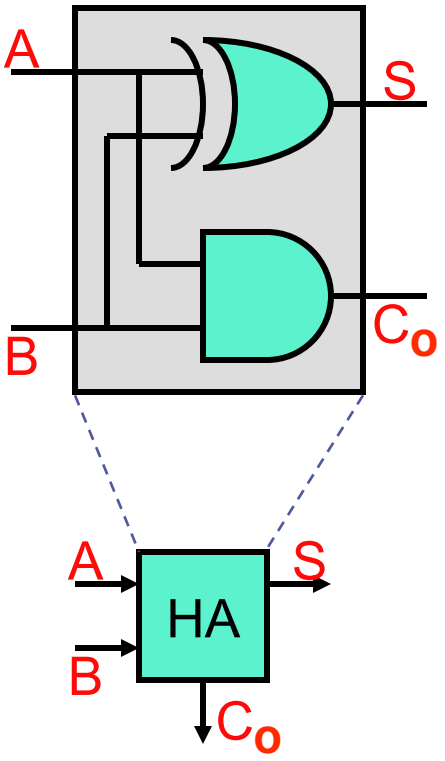
  \+ Each one is simpler (adding just 1 and 0)
    - So simple we can do it in hardware

Review: The Half Adder

- How to add two binary integers in hardware?
- Start with adding two bits
  - When all else fails … look at truth table

A	B	=	C_o	S
0	0	=	0	0
0	1	=	0	1
1	0	=	0	1
1	1	=	1	0

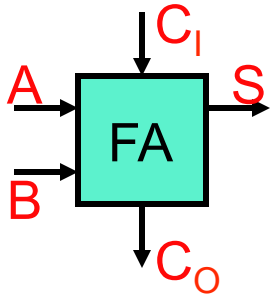
- **S = A⊕B (A XOR B)**
- **C_o (carry out) = AB**
- This is called a **half adder**



Review: The Full Adder

- We could chain half adders together, but to do that…
  - Need to incorporate a carry out from previous adder
  - Let's look at the truth table

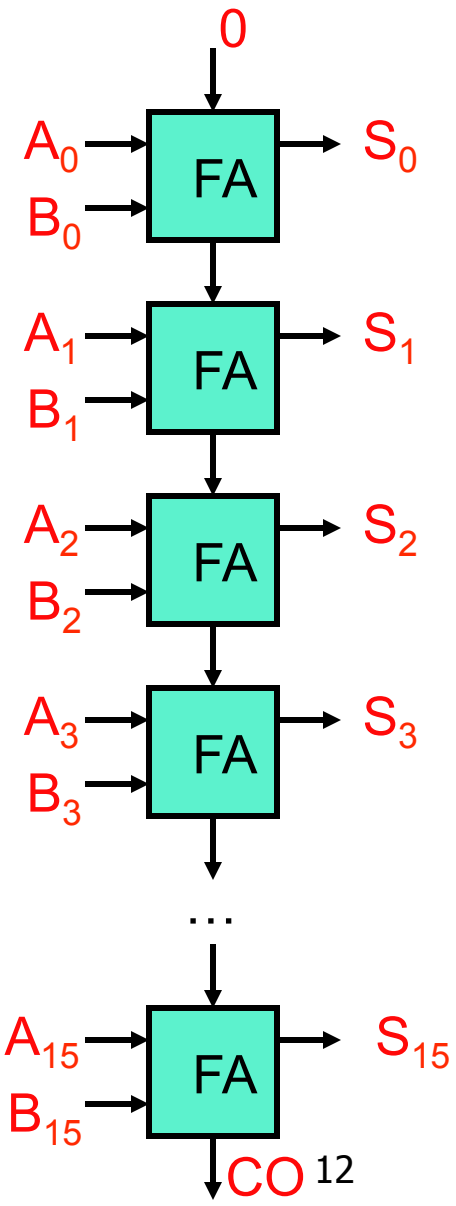
C_I	A	B	=	C_O	S
0	0	0	=	0	0
0	0	1	=	0	1
0	1	0	=	0	1
0	1	1	=	1	0
1	0	0	=	0	1
1	0	1	=	1	0
1	1	0	=	1	0
1	1	1	=	1	1



- $S = C_I'A'B + C_I'AB' + C_IA'B' + C_IAB = C_I \oplus A \oplus B$
- $C_O = C_I'AB + C_IA'B + C_IAB' + C_IAB = C_IA + C_IB + AB$
- This is a **full adder** → **ICQ:** what is its delay (in #gates)?

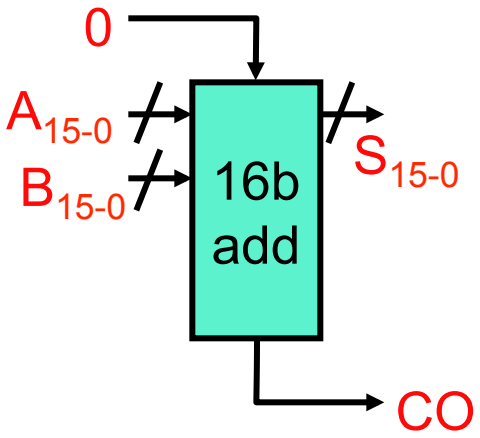
A 16-bit Adder

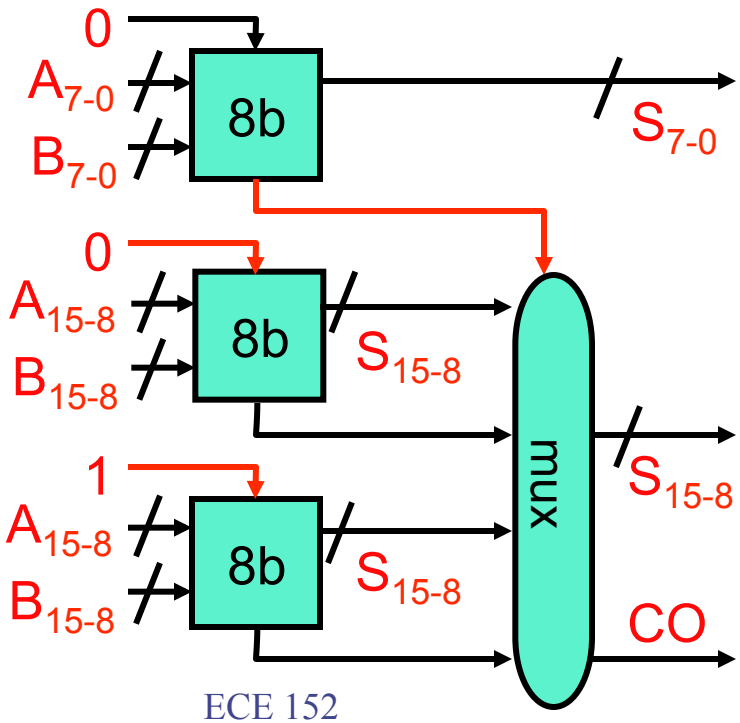
- Simple 16-bit adder
  - 16 1-bit full adders "chained" together
    - $CO_0 = CI_1$, $CO_1 = CI_2$, etc.
    - $CI_0 = 0$, CO_{15} is carry-out of entire adder
      - $CO_{15} = 1 \rightarrow$ "overflow"
- Design called **ripple-carry**: how fast is it?
  - In terms of **gate delays** (longest gate path)
    - Longest path is to CO_{15} (or S_{15})
  - $d(CO_{15}) = \mathbf{2} + MAX\{d(A_{15}),d(B_{15}),d(CI_{15})\}$
    - $d(A_{15}) = d(B_{15}) = 0$, $d(CI_{15}) = d(CO_{14})$
  - $d(CO_{15}) = 2 + d(CO_{14}) = 2 + 2 + d(CO_{13})$ …
  - **$d(CO_{15}) = 32$**
  - **2N = slow!**



A Faster (16-bit) Adder

- One option: **carry-select adder**
  - Do $A_{15-8}+B_{15-8}$ twice, once assuming CI_8 (CO_7) = 0, then once = 1
  - Choose the right one when CO_7 finally becomes available
  + Effectively cuts carry chain in half
  – But adds 8b adder and mux





How Fast Is the Faster Adder?

- $d(CO_{15}) = \max\{d(CO_{15-8}), d(CO_{7-0})\} + 2$ (+2 is for mux)
- $d(CO_{15}) = \max\{2*8, 2*8\} + 2 =$ **18** (2N delay for 8bit add)
- For dividing N-bit adder into 2 parts: **2*(N/2) + 2 = N+2**

- What if we broke up 16b adder into 4 parts?
  - Would delay be 2*(N/4) + 2 = 10? Not quite!
  - $d(CO_{15}) = \max\{d(CO_{15-12}),d(CO_{11-0})\} + 2$
  - $d(CO_{15}) = \max\{2*4, \max\{d(CO_{11-8}),d(CO_{7-0})\} + 2\} + 2$
  - $d(CO_{15}) = \max\{2*4,\max\{2*4,\max\{d(CO_{7-4}),d(CO_{3-0})\}+ 2\} + 2\}+ 2$
  - $d(CO_{15}) = \max\{2*4,\max\{2*4,\max\{2*4,2*4\} + 2\} + 2\} + 2$
  - $d(CO_{15}) = 2*4 + 3*2 =$ **14**

- In general, N-bit adder in M pieces: **2*(N/M) + (M-1)*2**
  - 16-bit adder in 8 parts: 2*(16/8) + 7*2 = **18 > 14 ???!**

Another Option

- Is the piece-wise faster adder as fast as we can go?
  - No!

- Another approach to using additional resources
  - Instead of redundantly computing sums assuming different carries, use redundancy to compute carries more quickly
    - This approach is called **carry lookahead addition (CLA)**

Review: Carry Lookahead Addition (CLA)

- Let's look at the carry function
  - $C_{16} = CO_{15} = A_{15}B_{15}+A_{15}C_{15}+B_{15}C_{15} =(A_{15}B_{15})+(A_{15}+B_{15})C_{15}$
- Very important insights into CLA:
  - $(A_{15}B_{15})$ generates a carry regardless of C_{15} →rename to **g_{15}**
  - $(A_{15}+B_{15})$ propagates C_{15} → rename to **p_{15}**

  - $C_{16} = g_{15}+p_{15}C_{15}$
  - $C_{16} = g_{15}+p_{15}(g_{14}+p_{14}C_{14})$
  - $C_{16} = g_{15}+p_{15}g_{14}+p_{15}p_{14}(g_{13}+p_{13}C_{13})$
  - $C_{16} = g_{15}+p_{15}g_{14}+ \ldots + p_{15}p_{14}\ldots p_2p_1g_0 + p_{15}p_{14}\ldots p_2p_1p_0p_0$
    - Important note: can compute C_{16} in 2 levels of logic!
  - Similar functions for C_{15} $(=CO_{14})$, etc.
    - In general: $C_i = g_{i-1}+p_{i-1}C_{i-1}$

Infinite Carry Lookahead

- Previous slide's CLA functions assume "infinite" hardware
  - Performance? Critical path is d(S_{15}) = ?
    - d(p_{14},g_{14})+d(c_{15} given p_{14},g_{14})+d(S_{15} given c_{15})=1+2+2 = 5 !!
    - Constant delay, i.e., not a function of N
  - But not very practical in terms of hardware
  - Assume 2N gates to compute p_i and g_i initially (ICQ: why 2N?)
  - Computation of a single C_N needs the following hardware:
    - N AND gates + 1 OR gate, and largest gates have N+1 inputs
  - Computation of all C_N…C_1 needs:
    - **N*(N+1)/2** AND gates + **N** OR gates, max N+1 inputs
  - Not too bad if N=16: 152 gates, max input 17
  - Pretty bad if N=64: 2144 gates, max input 65
  - Big circuits are slow and high input gates are slow

Motivation for Multi-Level Carry Lookahead

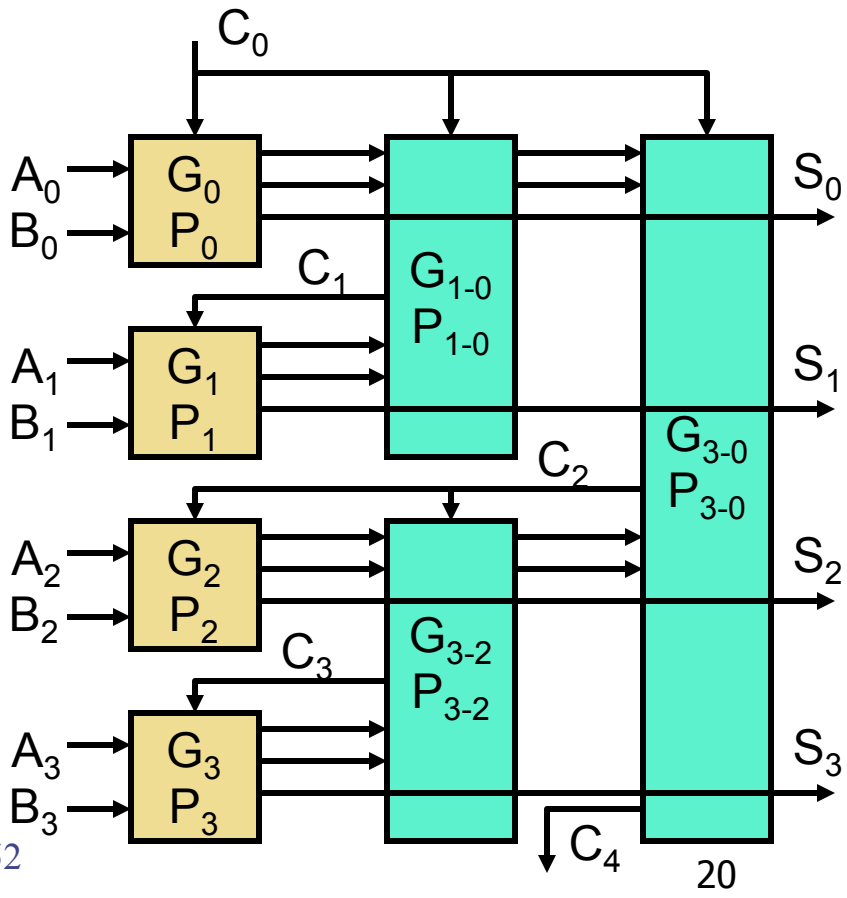
- Let's look at what we have so far (the two extremes)
  - **Ripple carry**
    - \+ Few small gates: no additional gates used to speed up addition
    - – Logic in series: 2N latency
  - **Infinite CLA**
    - – Many big gates: N*(N+3)/2 additional gates, max N+1 inputs
    - \+ Logic in parallel: constant latency of 5 gate delays
  - We'd like something in between
    - Reasonable number of small gates
    - Sub-linear (doesn't have to be constant) latency
  - **Multi-level CLA**
    - Exploits hierarchy to achieve good compromise between the two extremes

Two-Level CLA for 4-bit Adder

- Individual carry equations
  - $C_1 = g_0+p_0C_0$, $C_2 = g_1+p_1C_1$, $C_3 = g_2+p_2C_2$, $C_4 = g_3+p_3C_3$
- Fully expanded (infinite hardware) CLA equations
  - $C_1 = g_0+p_0C_0$
  - $C_2 = g_1+p_1g_0+p_1p_0C_0$
  - $C_3 = g_2+p_2g_1+p_2p_1g_0+p_2p_1p_0C_0$
  - $C_4 = g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0+p_3p_2p_1p_0C_0$
- Hierarchical CLA equations
  - **First level**: expand C_2 using C_1 and C_4 using C_3
    - $C_2 = g_1+p_1(g_0+p_0C_0) = (g_1+p_1g_0)+(p_1p_0)C_0 = G_{1-0}+P_{1-0}C_0$
    - $C_4 = g_3+p_3(g_2+p_2C_2) = (g_3+p_3g_2)+(p_3p_2)C_2 = G_{3-2}+P_{3-2}C_2$
  - **Second level**: expand C_4 using expanded C_2
    - $C_4 = G_{3-2}+P_{3-2}(G_{1-0}+P_{1-0}C_0) = (G_{3-2}+P_{3-2}G_{1-0})+(P_{3-2}P_{1-0})C_0$
    - $C_4 = G_{3-0}+P_{3-0}C_0$

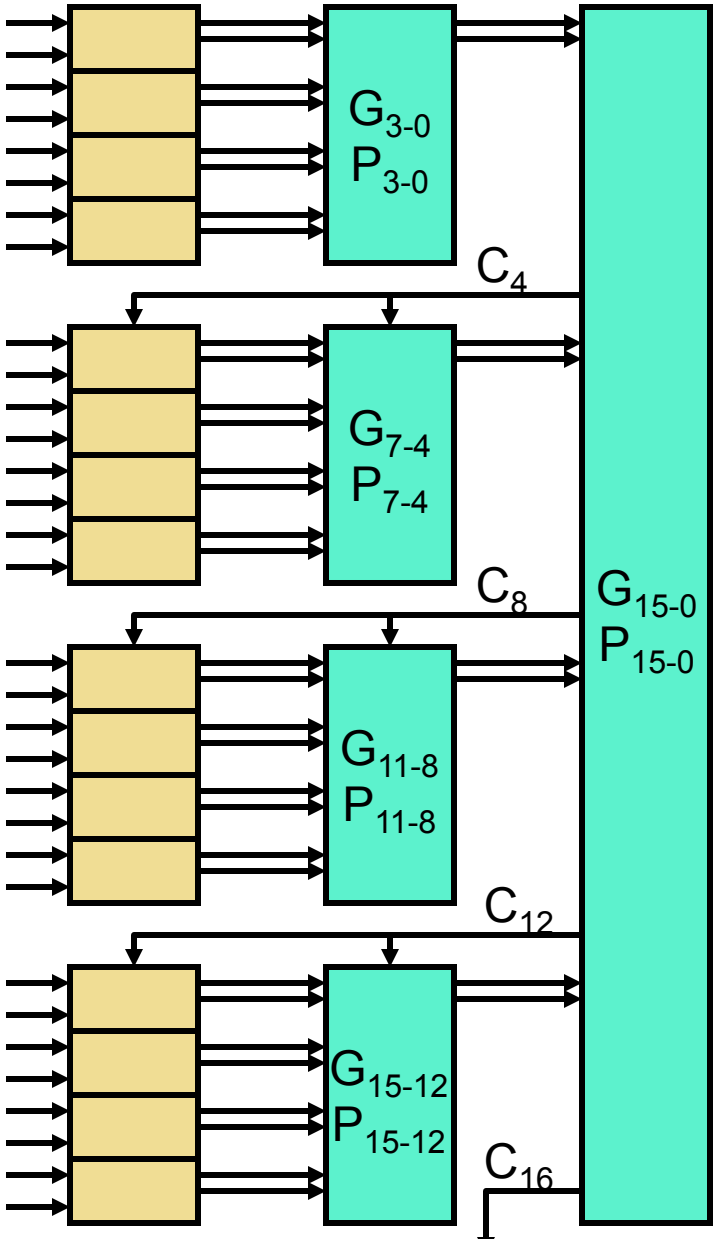
Two-Level (2L) CLA for 4-bit Adder

- Hardware?
  - First level: block is infinite CLA for N=2
    - 5 gates per block, max # gate inputs (MNGI)=3
    - 2 of these "blocks"
  - Second level: 1 of these "blocks"
  - Total: **15 gates & 3 MNGI**
    - Infinite CLA: 14 & 5 (?!)
- Latency?
  - Total: **9 (ICQ: why?)**
    - Infinite CLA: 5
- 2 level: bigger and slower??!
  - ICQ: what happened?



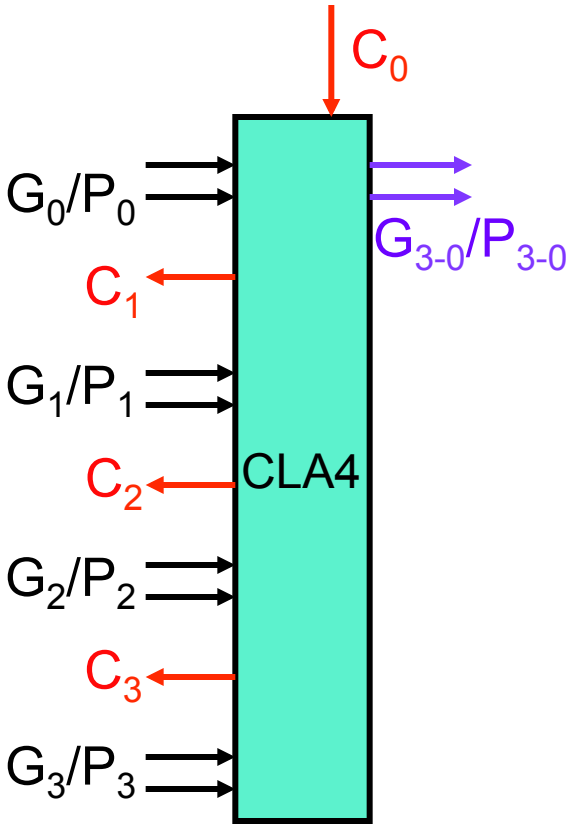
Two-Level CLA for 16-bit Adder

- 4 G/P inputs per level
- Hardware?
  - First level: 14&5 * 4 blocks
  - Second level: 14&5 * 1 block
  - Total: **70&5**
    - Infinite: 152&17
- Latency?
  - Total: **9** (1 + 2 + 2 + 2 + 2)
  - Infinite: 5
- That's more like it!
  - CLA for a 64-bit adder?



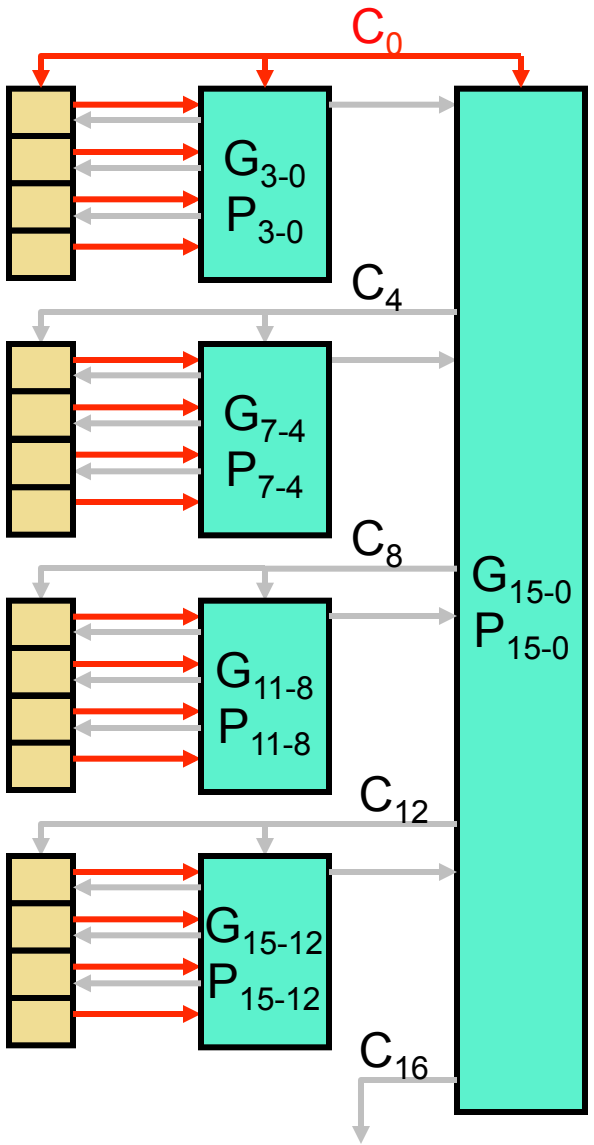
A Closer Look at CLA Delay

- CLA block has "individual" G/P inputs
  - Uses them to perform **two** calculations
  - Group G/P on way up tree
  - Group interior carries on way down tree
    - Given group carry-in from level above
  - Group carry-in for outer level (C_0) ready at 0
  - Outer level G/P, interior carries in parallel



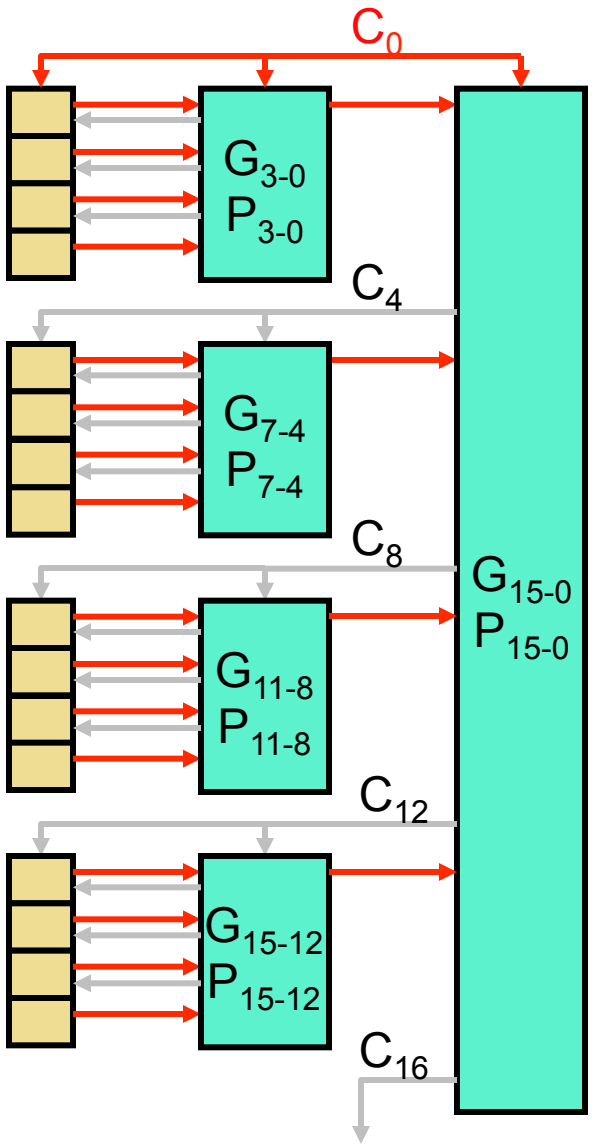
CLA Tree Signal Timing: d1

- Signals ready after 1 gate delay
  - C_0
  - Individual G/P



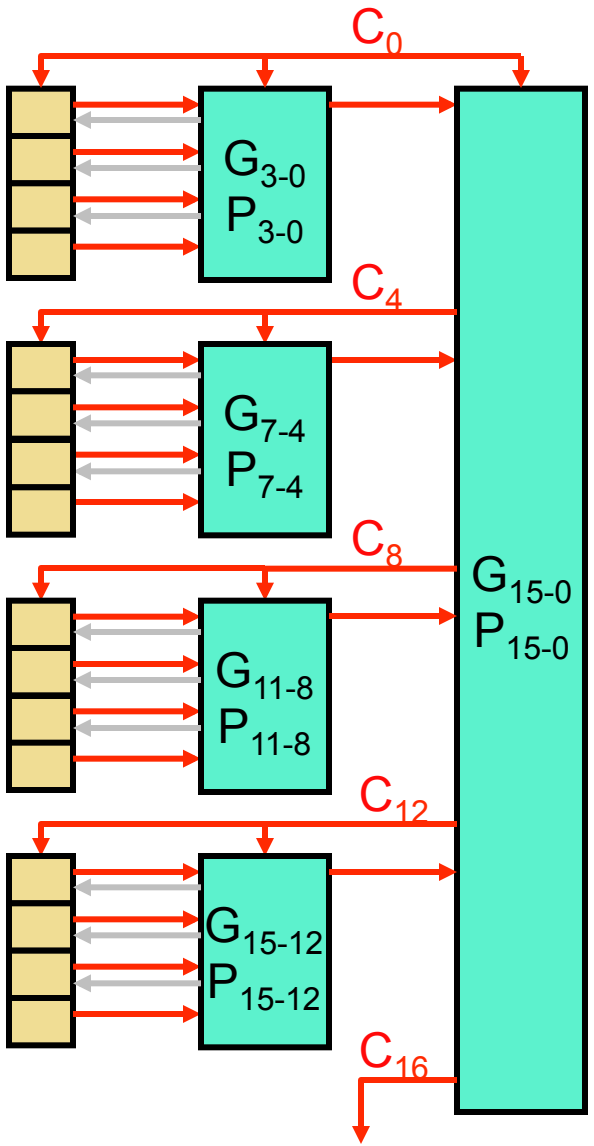
CLA Tree Signal Timing: d3

- What is ready after 3 gate delays?
  - First level group G/P



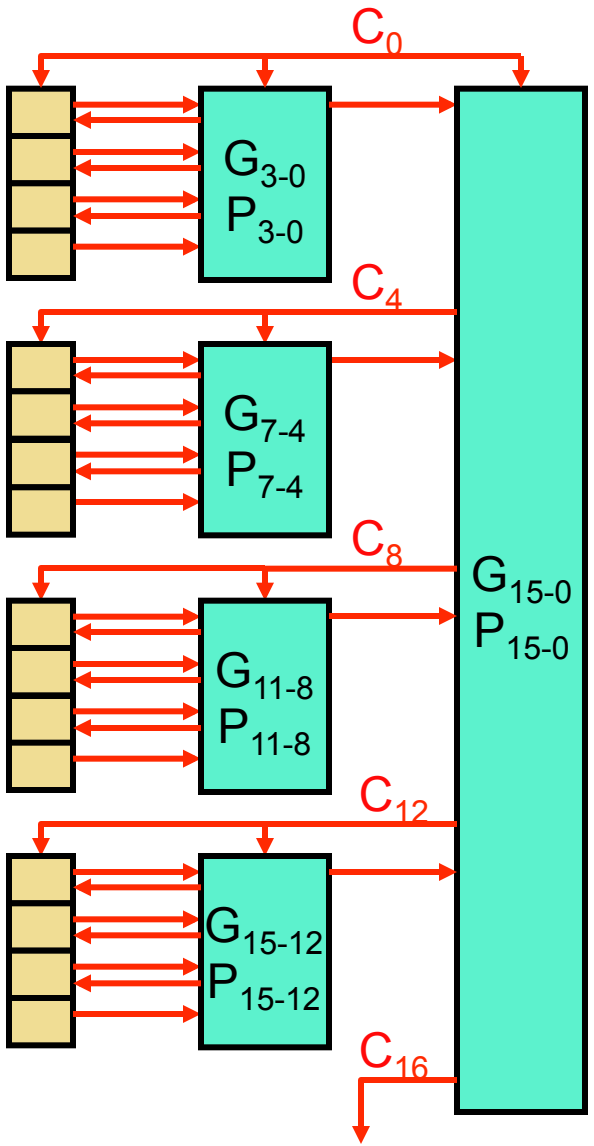
CLA Tree Signal Timing: d5

- And after 5 gate delays?
  - Outer level "interior" carries
    - C_4, C_8, C_{12} , C_{16}



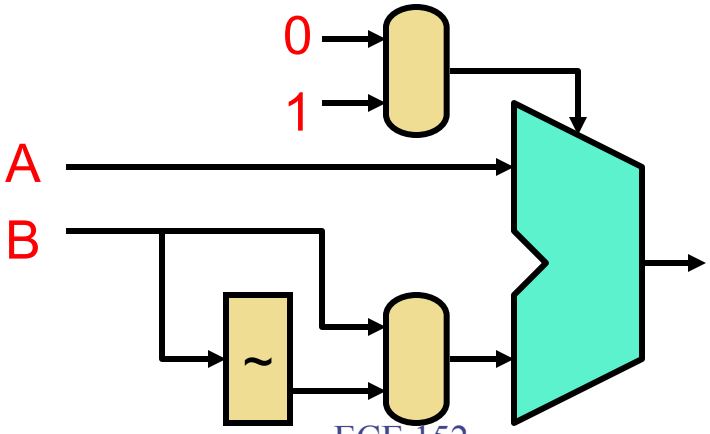
CLA Tree Signal Timing: d7

- And after 7 gate delays?
  - First level "interior" carries
    - C_1, C_2, C_3
    - C_5, C_6, C_7
    - C_9, C_{10}, C_{11}
    - C_{13}, C_{14}, C_{15}
    - Essentially, all remaining carries
- S_i ready 2 gate delays after C_i
  - All sum bits ready after 9 delays!



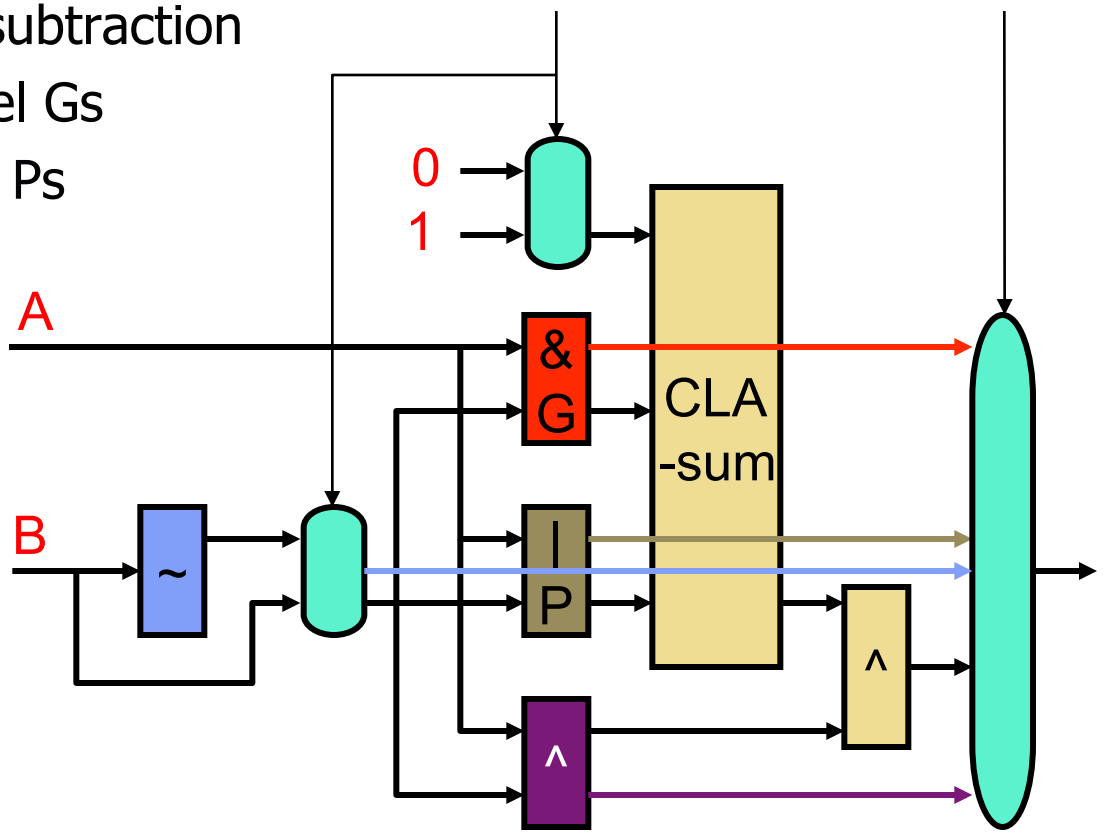
Subtraction: Addition's Tricky Pal

- Sign/magnitude subtraction is mental reverse addition
  - Two's complement subtraction **is** addition
- How to subtract using an adder?
  - **`sub A, B = add A, -B`**
  - Negate B before adding (fast negation trick: –B = B' + 1)
- Isn't a subtraction then **a negation and two additions**?
  + No, an adder can implement A+B+1 by setting the carry-in to 1
  + Clever, huh?

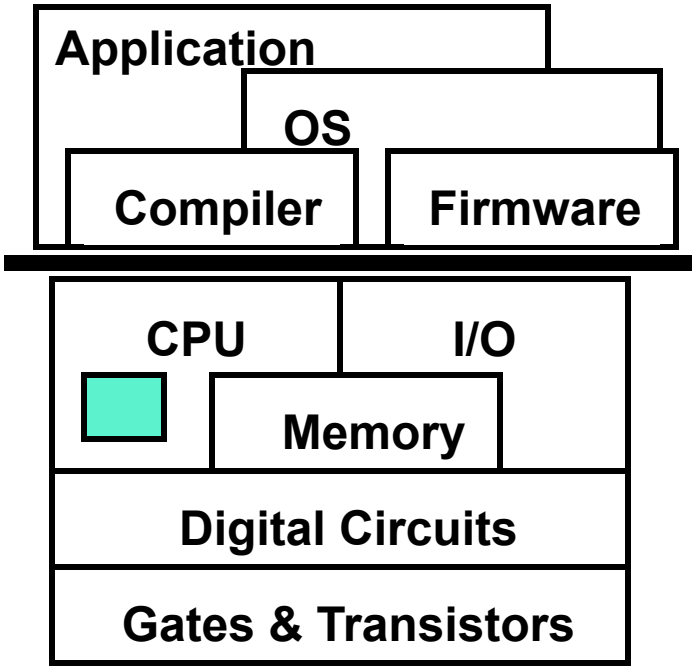


A 16-bit ALU

- Build an ALU with functions: **add/sub**, **and**, **or**, **not**, **xor**
  - All of these already in CLA adder/subtracter
  - **add A B**, **sub A B** (done already)
  - **not B** is needed for subtraction
  - **and A,B** are first level Gs
  - **or A,B** are first level Ps
  - **xor A,B**?
    - $S_i = A_i \hat{} B_i \hat{} C_i$



This Unit: Arithmetic and ALU Design



- Integer Arithmetic and ALU
  - Binary number representations
  - Addition and subtraction
  - The integer ALU
  - Shifting and rotating
  - Multiplication
  - Division
- Floating Point Arithmetic
  - Binary number representations
  - FP arithmetic
  - Accuracy

Shifts

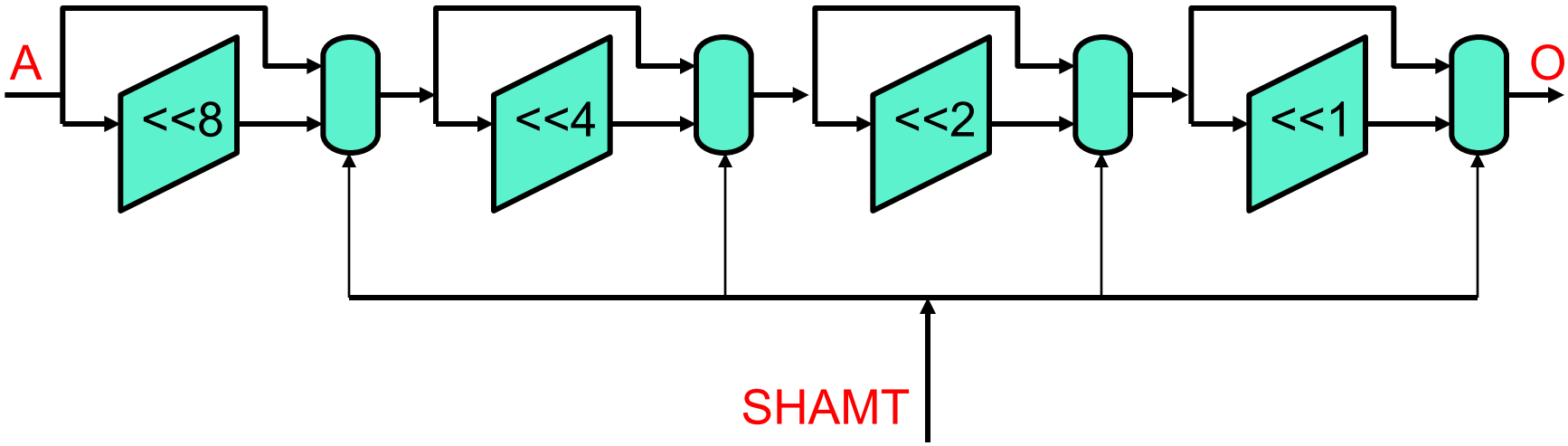
- Shift: move all bits in a direction (left or right)
  - Denoted by << (left shift) and >> (right shift) in C/C++/Java
- ICQ: Left shift example: 001010 << 2 = ?
- ICQ: Right shift example: 001010 >> 2 = ?
- Shifts are useful for
  - Bit manipulation: extracting and setting individual bits in words
  - Multiplication and division by powers of 2
    - A * 4 = A << 2
    - A / 8 = A >> 3
    - A * 5 = (A << 2) + A
  - Compilers use this optimization, called **strength reduction**
    - Easier to shift than it is to multiply (in general)

Rotations

- Rotations are slightly different than shifts
  - 1101 rotated 2 to the right = ?
- Rotations are generally less useful than shifts
  - But their implementation is natural if a shifter is there
  - MIPS has only shifts

Barrel Shifter

- What about shifting left by any amount from 0 to 15?
  - Cycle input through "left-shift-by-1" up to 15 times?
    - Complicated, variable latency
  - 16 consecutive "left-shift-by-1-or-0" circuits?
    - Fixed latency, but would take too long
  - **Barrel shifter**: four "shift-left-by-X-or-0" circuits (X = 1,2,4,8)



Right Shifts and Rotations

- Right shifts and rotations also have barrel implementations
  - But are a little different

- Right shifts
  - Can be **logical** (shift in 0s) or **arithmetic** (shift in copies of MSB)

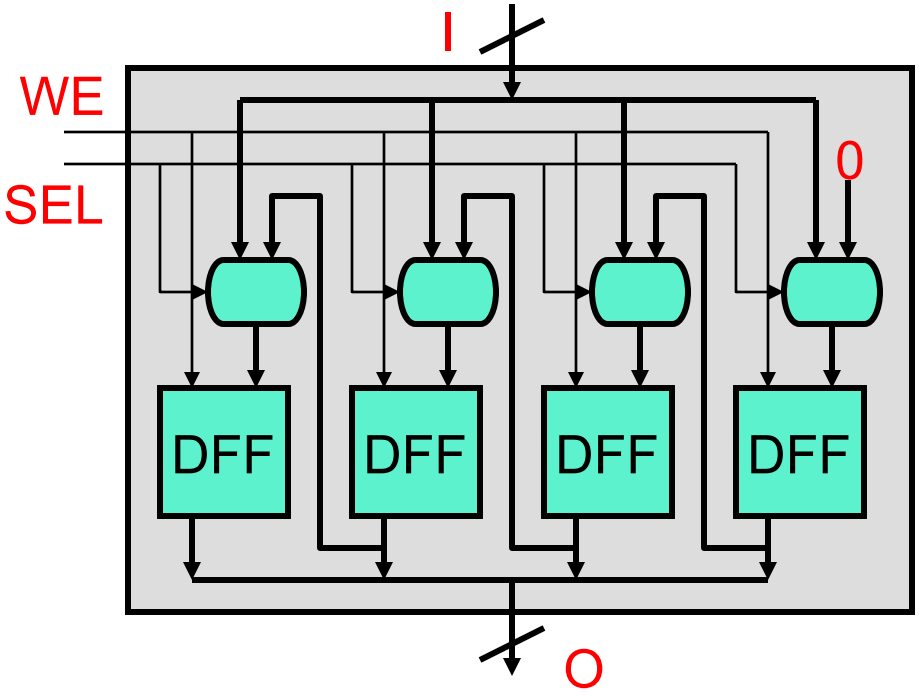
    `srl 110011,2` → `result is 001100`

    `sra 110011,2` → `result is 111100`
  - Caveat: `sra` is not equal to division by 2 of negative numbers
  - Why might we want both types of right shifts?

- Rotations
  - Mux in wires of upper/lower bits

Shift Registers

- **Shift register**: shift in place by constant quantity
  - Sometimes that's a useful thing



Base10 Multiplication

- Remember base 10 multiplication from 3rd grade?

```
    43     // multiplicand
*   12     // multiplier
    86
+ 430
  516      // product
```

  - Start with running total 0, repeat steps until no multiplier digits
    - Multiply multiplicand by least significant multiplier digit
    - Add to total
    - Shift multiplicand one digit to the left (multiply by 10)
    - Shift multiplier one digit to the right (divide by 10)

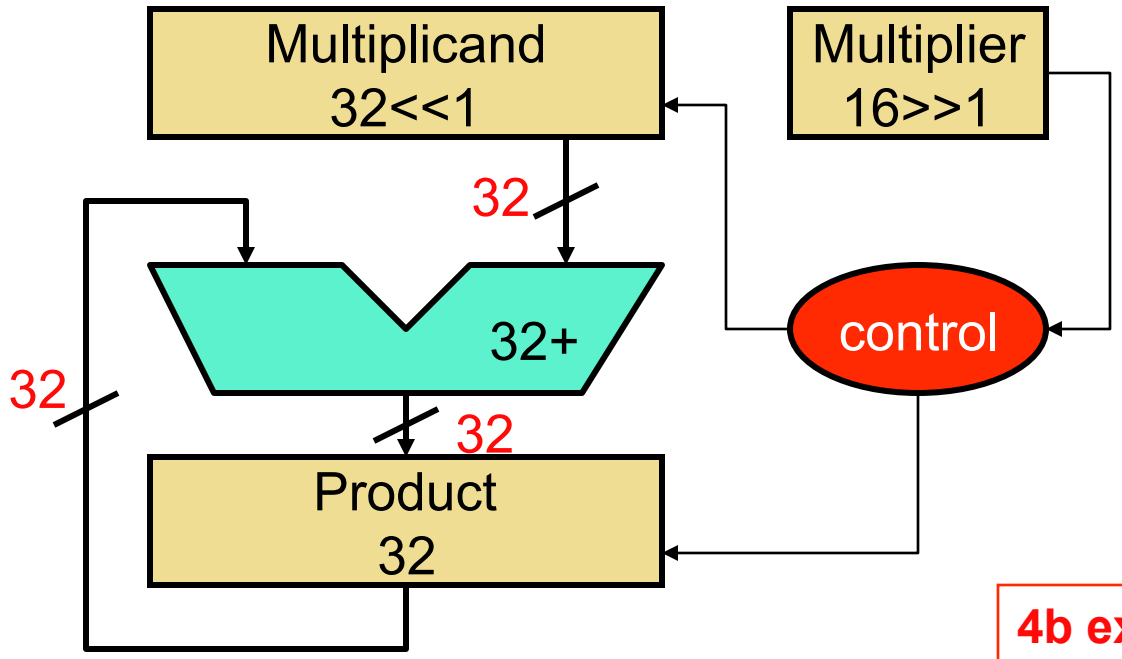
  - Product of N-digit and M-digit numbers potentially has N+M digits

Binary Multiplication

```
   43 = 00000101011    // multiplicand
*  12 = 00000001100    // multiplier
-------------------
    0 = 00000000000
    0 = 00000000000
  172 = 00010101100
+ 344 = 00101011000
-------------------
  516 = 01000000100    // product
```

- Same thing except …
  - – There are more individual steps (smaller base)
  - + But each step is simpler
  - Multiply multiplicand by **least significant multiplier bit**
    - 0 or 1 → no actual multiplication, just add multiplicand or not
  - Add to total: we know how to do that
  - Shift multiplicand left, multiplier right by one bit: **shift registers**

Simple 16x16=32bit Multiplier Circuit

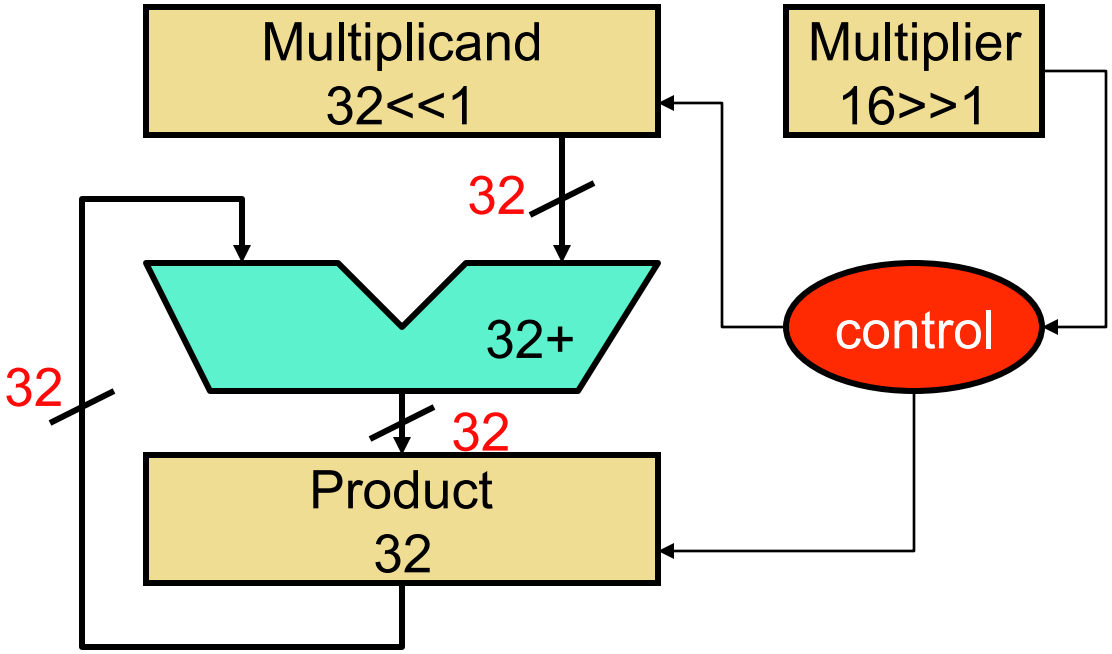


4b example:

0101 x 0110

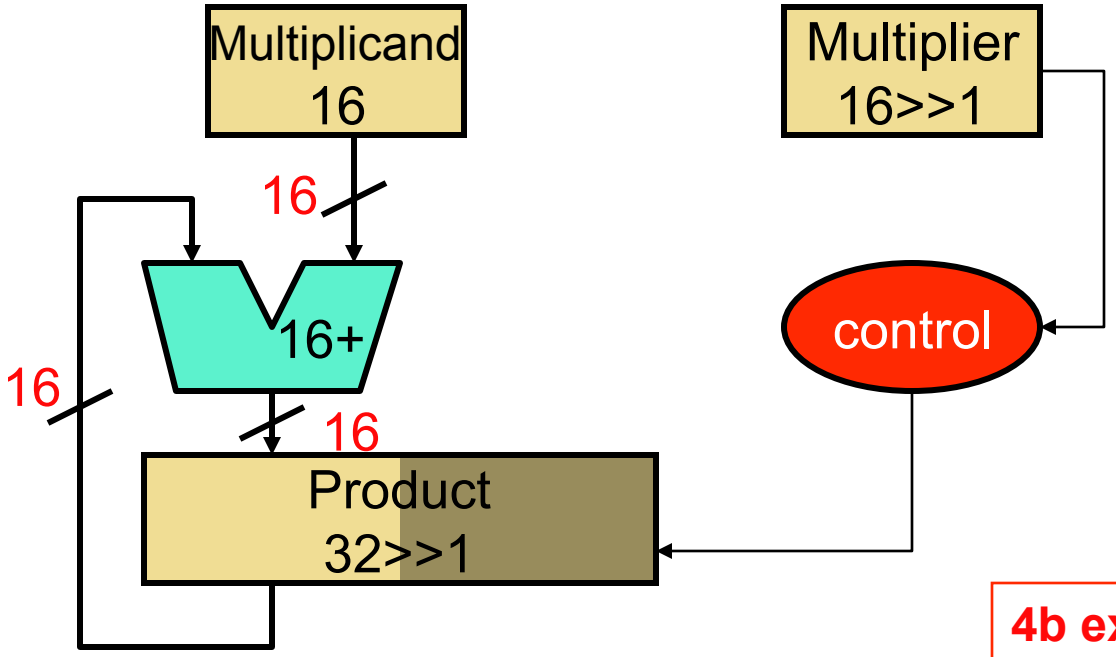
- **Control algorithm**: repeat 16 times
  - If LSB(multiplier) == 1, then add multiplicand to product
  - Shift multiplicand left by 1
  - Shift multiplier right by 1

Inefficiencies with Simple Circuit



- Notice
  - 32-bit addition, but 16 multiplicand bits are always 0
    - And 0-bits are always moving
  - Solution? Instead of shifting multiplicand left, shift product right

Better 16-bit Multiplier

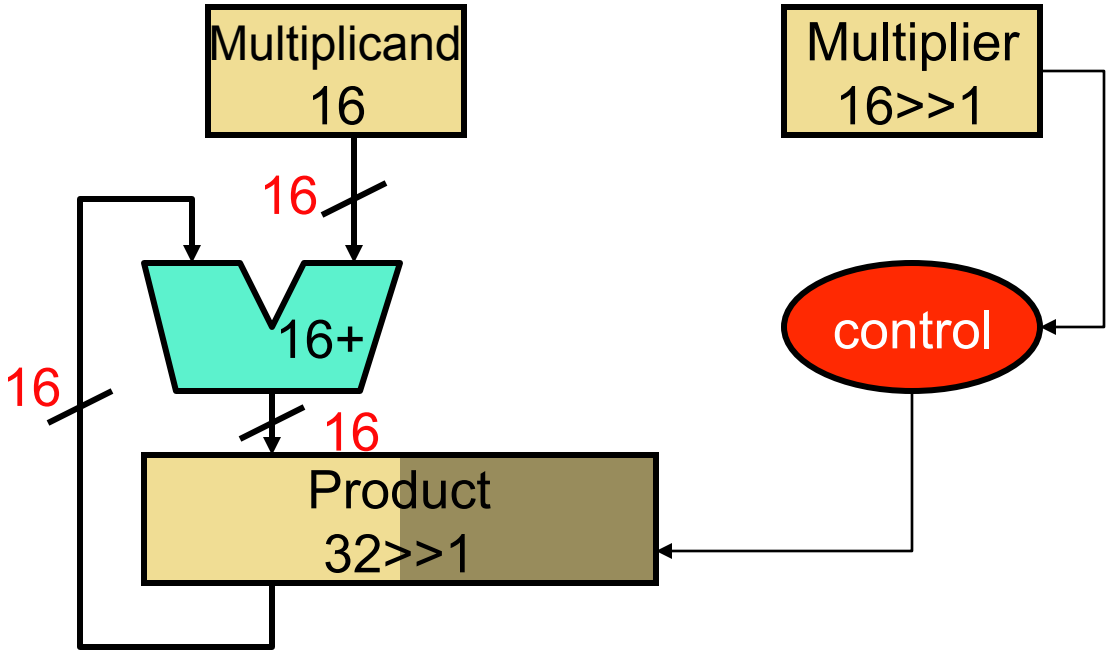


4b example:

0101 x 0110

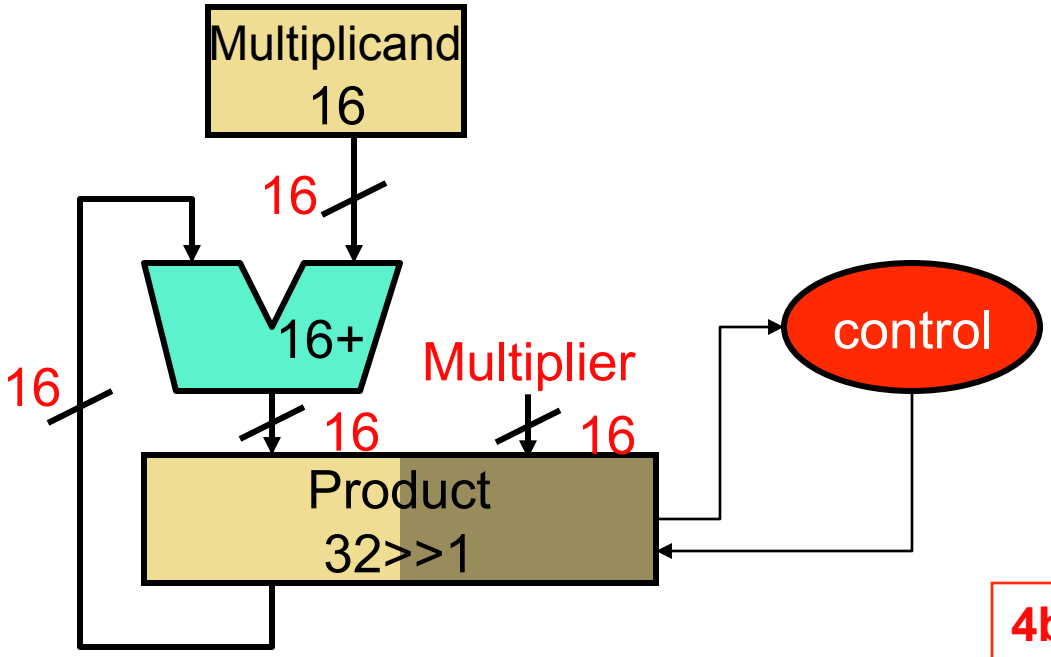
- **Control algorithm**: repeat 16 times
  - LSB(multiplier) == 1 ? Add multiplicand to upper half of product
  - Shift multiplier right by 1
  - Shift product right by 1

Another Inefficiency



- Notice one more inefficiency
  - What is initially the lower half of product gets thrown out
  - As useless lower half of product is shifted right, so is multiplier
  - Solution: use lower half of product as multiplier

Even Better 16-bit Multiplier



4b example:

0101 x 0110

- **Control algorithm**: repeat 16 times
  - LSB(multiplier) == 1 ? Add multiplicand to upper half of product
  - Shift product right by 1

Multiplying Negative Numbers

- If multiplicand is negative, our algorithm still works
  - As long as right shifts are arithmetic and not logical
  - Try 1111*0101
- If multiplier is negative, the algorithm breaks

- Two solutions
  1) Negate multiplier, then negate product
  2) Booth's algorithm

Booth's Algorithm

- Notice the following equality (Booth did)
  - $2^J + 2^{J-1} + 2^{J-2} + \ldots + 2^K = 2^{J+1} - 2^K$
  - Example: 0111 = 1000 - 0001
  - We can exploit this to create a faster multiplier

- How?
  - Sequence of N 1s in the multiplier yields sequence of N additions
  - Replace with one addition and one subtraction

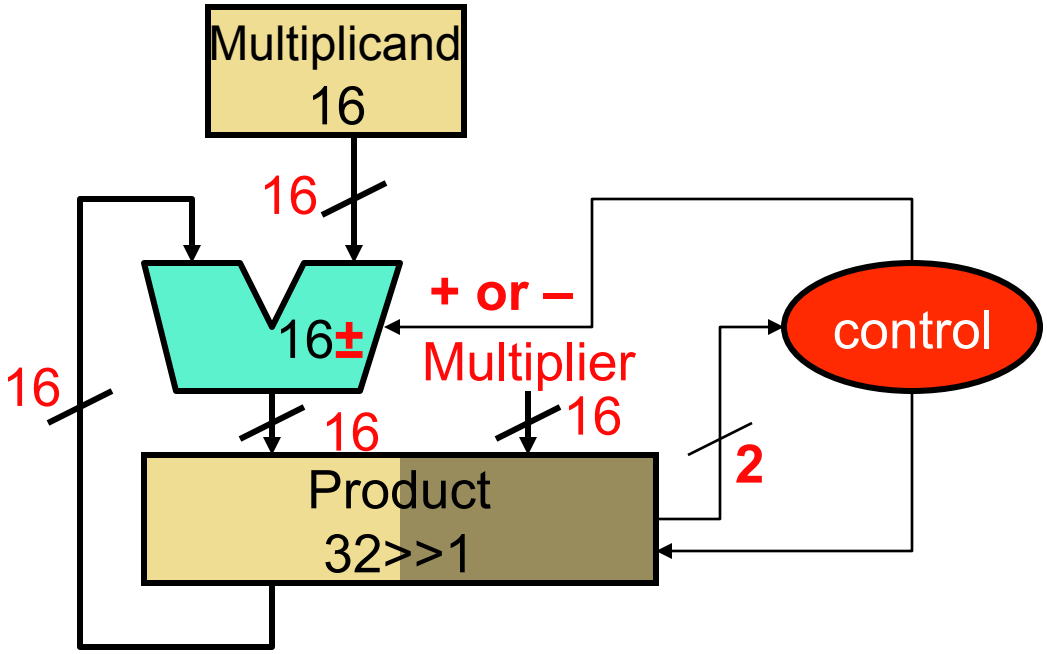
Booth In Action

- For each multiplier bit, also examine bit to its right
  - **00**: middle of a run of 0s, do nothing
  - **10**: beginning of a run of 1s, subtract multiplicand
  - **11**: middle of a run of 1s, do nothing
  - **01**: end of a run of 1s, add multiplicand

```
    43 = 00000101011
*   12 = 000000011100
-------------------------------------------
     0 = 00000000000   // multiplier bits 0_ (implicit 0)
+    0 = 00000000000   // multiplier bits 00
- 172 = 11101010100   // multiplier bits 10
+    0 = 00000000000   // multiplier bits 11
+ 688 = 01010110000   // multiplier bits 01
-------------------------------------------
   516 = 01000000100
```

ICQ: so why is Booth better?

Booth Hardware



- **Control algorithm**: repeat 16 times
  - Multiplier LSBs == 10? Subtract multiplicand from product
  - Multiplier LSBs == 01? Add multiplicand to product
  - Shift product/multiplier right by 1 (not by 2!)

Booth in Summary

- Performance/efficiency
  - + Good for sequences of 3 or more 1s
    - Replaces 3 (or more) adds with 1 add and 1 subtract
  - Doesn't matter for sequences of 2 1s
    - Replaces 2 adds with 1 add and 1 subtract (add = subtract)
  - – Actually bad for singleton 1s
    - Replaces 1 add with 1 add and 1 subtract

- Bottom line
  - Worst case multiplier (101010) requires N/2 adds + N/2 subs
    - What is the worst case multiplier for straight multiplication?
  - How is this better than normal multiplication?

Modified Booth's Algorithm

- What if we detect singleton 1s and do the right thing?

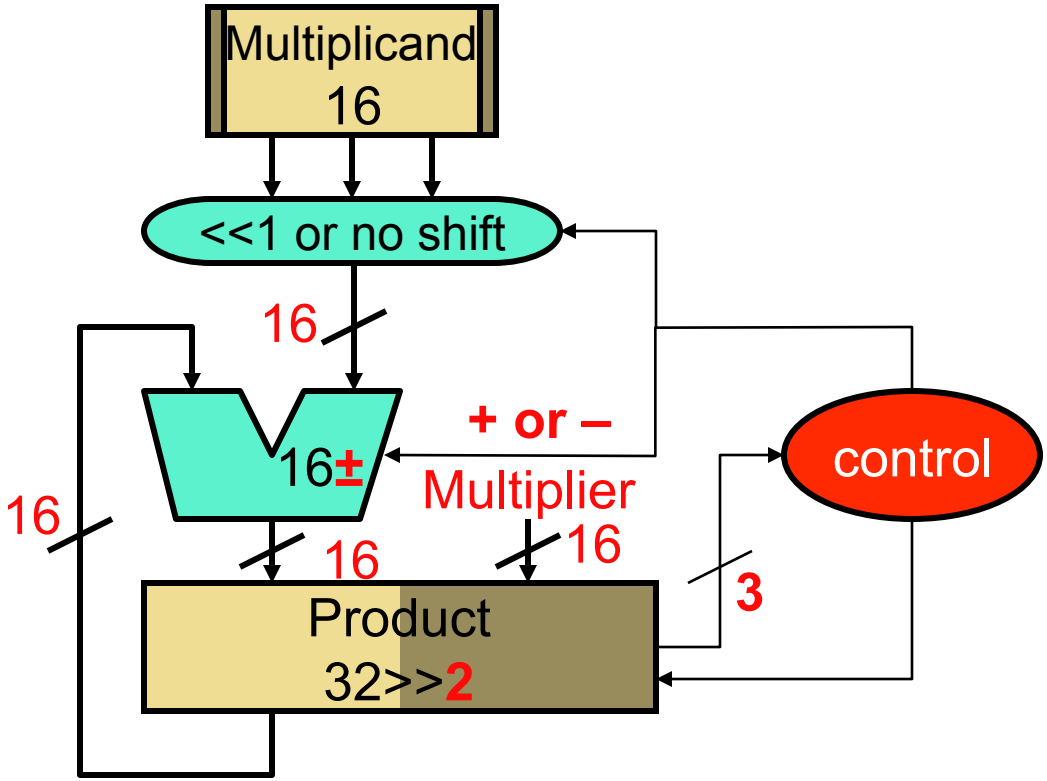
- Examine multiplier bits in groups of 2s plus a helper bit on the right (as opposed to 1 bit plus helper bit on right)
  - Means we'll need to shift product/multiplier by 2 (not 1)

  - **000**: middle of run of 0s, do nothing
  - **100**: beginning of run of 1s, subtract multiplicand<<1 (M*2)
    - Why M*2 instead of M?
  - **010**: singleton 1, add multiplicand
  - **110**: beginning of run of 1s, subtract multiplicand
  - **001**: end of run of 1s, add multiplicand
  - **101**: end of run of 1s, beginning of another, subtract multiplicand
    - Why is this? $-2^{J+1} + 2^{J} = -2^{J}$
  - **011**: end of a run of 1s, add multiplicand<<1 (M*2)
  - **111**: middle of run of 1s, do nothing

Modified Booth In Action

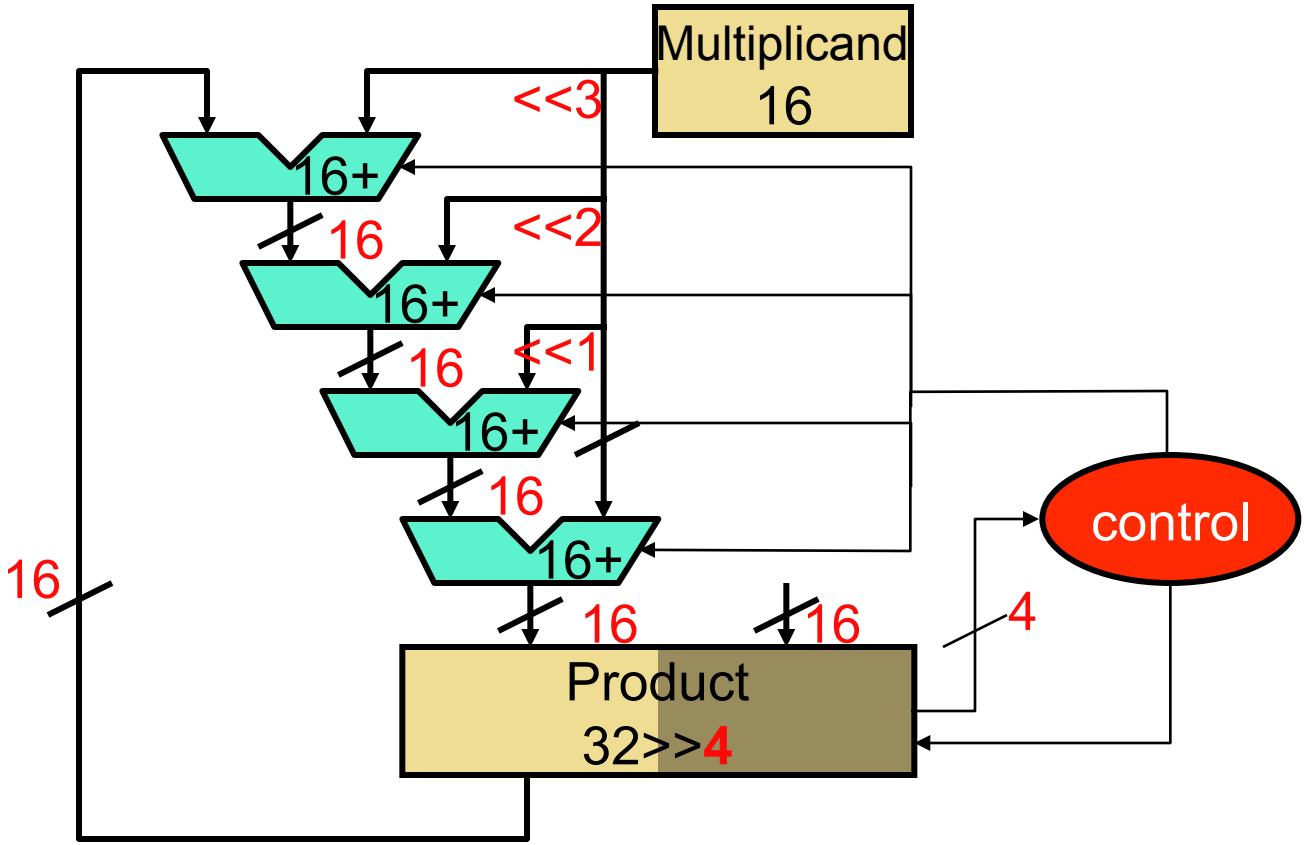
```
    43 = 00000101011
*   12 = 00000001100
     0 = 00000000000    // multiplier bits 000
- 172 = 11101010100    // multiplier bits 110
+ 688 = 01010110000    // multiplier bits 001
  516 = 01000000100
```

Modified Booth Hardware



- **Control algorithm**: repeat 8 times (not 16!)
  - Based on 3b groups, add/subtract shifted/unshifted multiplicand
  - Shift product/multiplier right by **2**

Another Multiplier: Multiple Adders

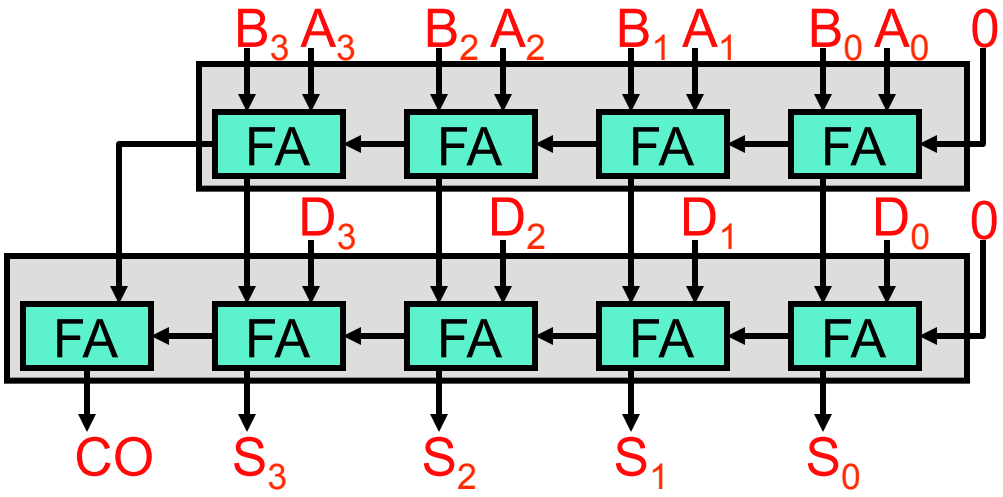


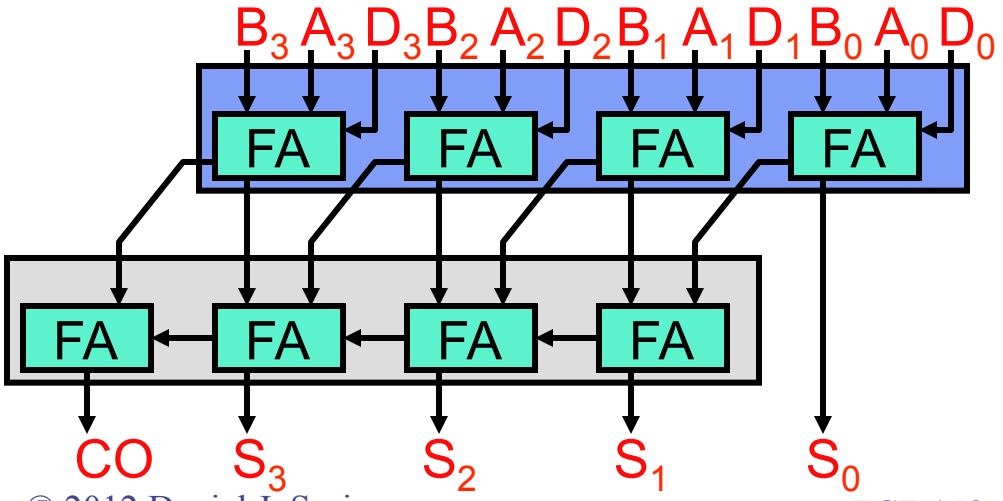
- Can multiply by N bits at a time by using N adders
  - Doesn't help: 4X fewer iterations, each one 4X longer (4*9=**36**)

Carry Save Addition (CSA)

- **Carry save addition (CSA):** d(N adds) < N*d(1 add)
  - Enabling observation: unconventional view of full adder
    - 3 inputs (A,B,C_{in}) → 2 outputs (S,C_{out})
  - If adding two numbers, only thing to do is chain C_{out} to C_{in+1}
    - But what if we are adding three numbers (A+B+D)?
  - One option: back-to-back conventional adders
    - Add A + B = temp
    - Add temp + D = Sum
  - Better option: instead of rippling carries in first addition (A+B), feed the D bits in as the carry bits (treat D bits as C bits)
    - Assume A+B+D = temp2
    - Then do traditional addition (not CSA) of temp2 and C bits generated during addition of A+B+D

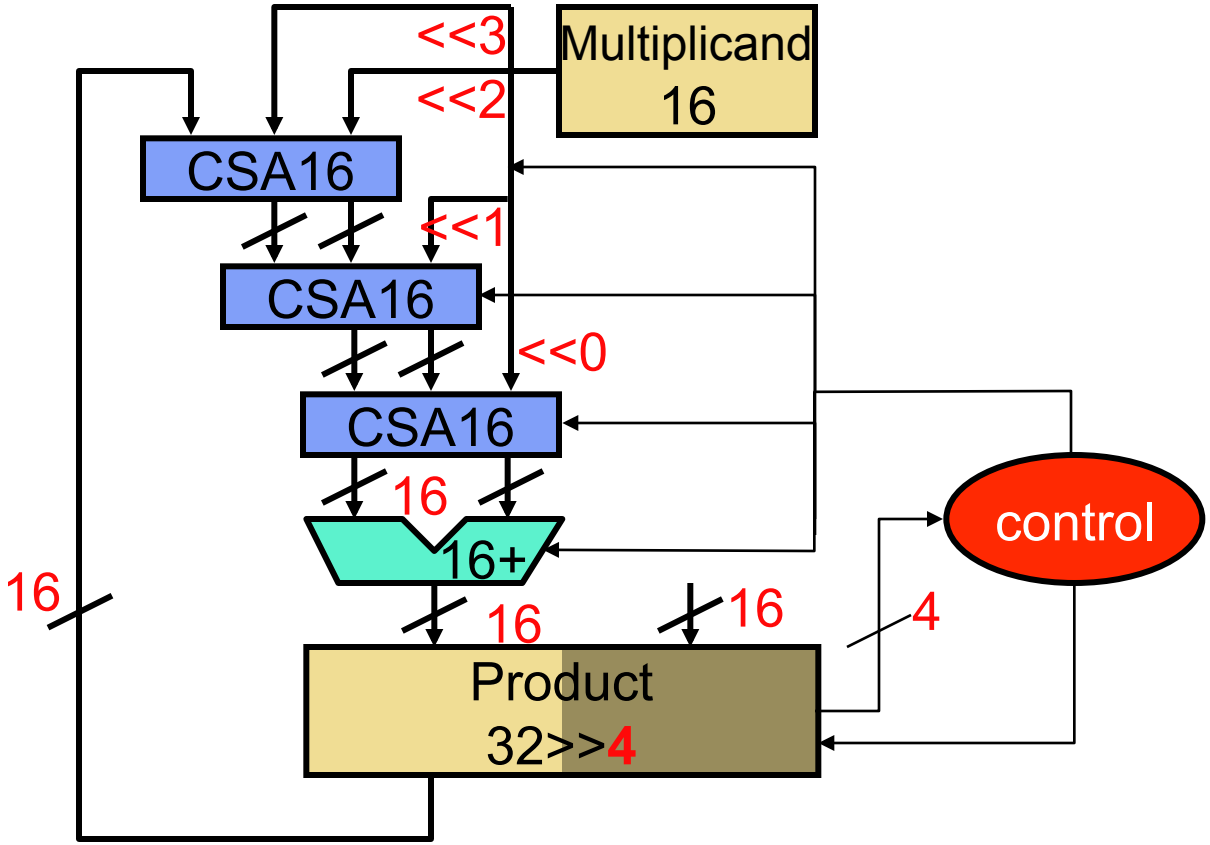
Carry Save Addition (CSA)





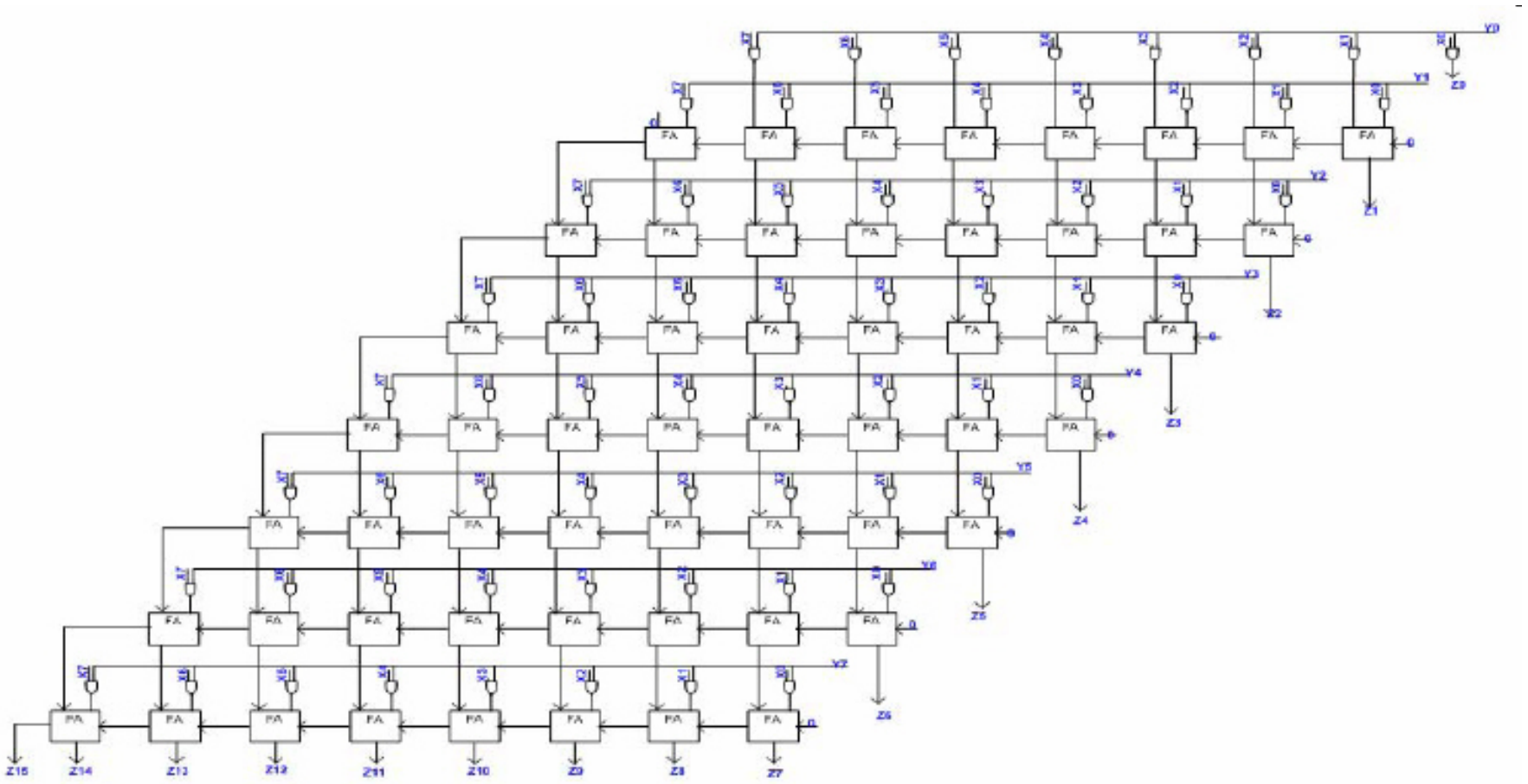
- 2 conventional adders
  - [2 * d(add)] gate levels
  - d(add16)=9
  - → d = 18
- k conventional adders
  - d = [k * d(add16)]
  - d = 9k
- CSA+conventional adder
  - d = [d(CSA) + d(add16)]
  - d(CSA) = d(1 FA) = 2
  - → d = 11
- k CSAs+conventional add
  - d = [k*d(CSA) + d(add)]
  - d = 2k + 9

Carry Save Multiplier



- 4-bit at a time multiplier using 3 CSA + 1 normal adder
  - Actually helps: 4X fewer iterations, each only (2+2+2+9=**15**)

Wallace Tree Multiplier (based on CSA)



Decimal Division

- Remember 4th grade long division?

```
      43          // quotient
12 √521           // divisor √ dividend
   -480
     41
   - 36
      5          // remainder
```

  - Shift divisor left (multiply by 10) until MSB lines up with dividend's
  - Repeat until remaining dividend (remainder) < divisor
    - Find largest single digit q such that (q*divisor) < dividend
    - Set LSB of quotient to q
    - Subtract (q*divisor) from dividend
    - Shift quotient left by one digit (multiply by 10)
    - Shift divisor right by one digit (divide by 10)

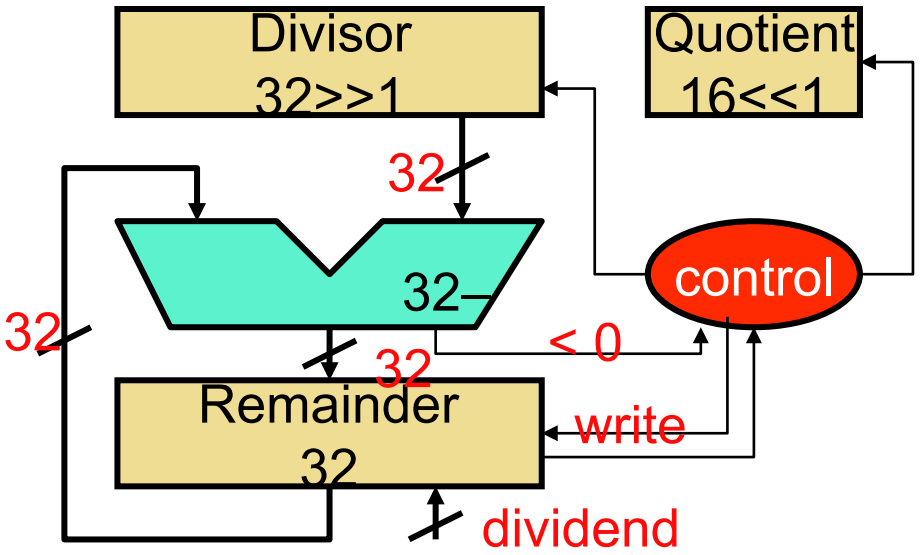
Binary Division

```
                                    101011    = 43
      ____              ____________________
  12 √521   =   01100  √  01000001001
     -384   =          -  0110000000
     ----                 ----------
      137   =               010001001
     -  0   =          -            0
     ----                 ----------
      137   =               010001001
     - 96   =          -     01100000
     ----                 ----------
       41   =                 0101001
     -  0   =          -            0
     ----                 ----------
       41   =                 0101001
     - 24   =          -       011000
     ----                 ----------
       17   =                  010001
     - 12   =          -        01100
     ----                 ----------
        5   =                     101
```

Hardware for Binary Division

- Same as decimal division, except (again)
  - – More individual steps (base is smaller)
  - + Each step is simpler
  - Find largest bit q such that (q*divisor) < dividend
    - q = 0 or 1
  - Subtract (q*divisor) from dividend
    - q = 0 or 1 → no actual multiplication, subtract divisor or not
  - One complication: **largest** q such that (q*divisor) < dividend
    - How to know if (1*divisor) < dividend?
    - Human (e.g., ECE 152 student) can eyeball this
    - Computer cannot
      - Subtract and see if result is negative

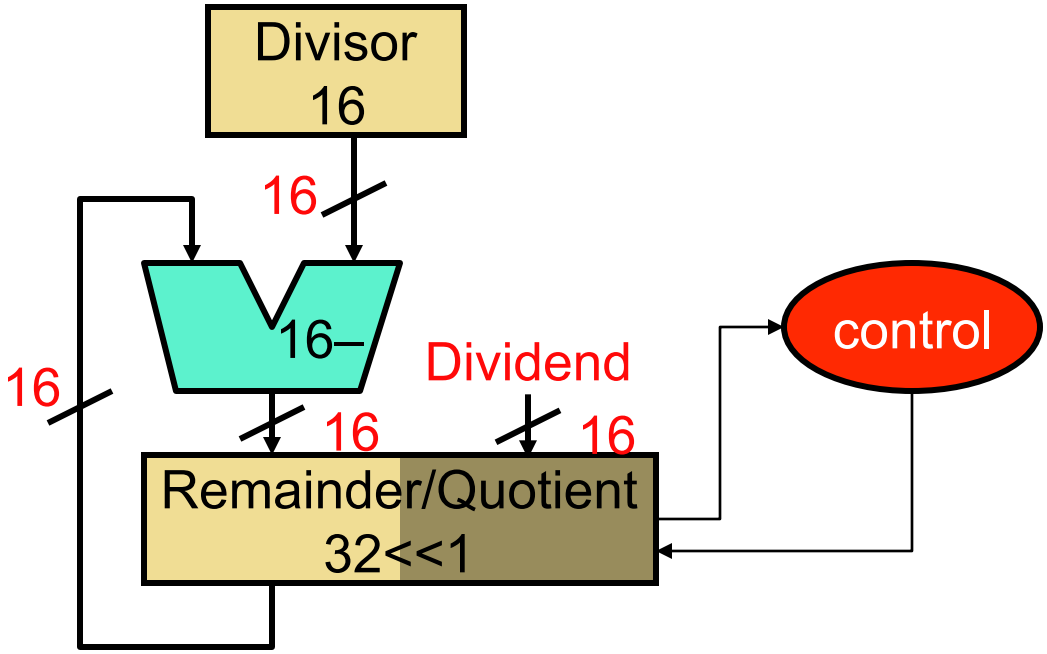
Simple 16-bit Divider Circuit



- First: Shift Divisor left to align it with Dividend

- Then repeat this loop until Divisor<Remainder
  - Subtract Divisor from Remainder (Remainder initially = Dividend)
    - Result >= 0? Remainder←Result, write 1 into Quotient LSB
    - Result < 0? Just write 0 into quotient LSB
  - Shift divisor right 1 bit, shift quotient left 1 bit

Even Better Divider Circuit

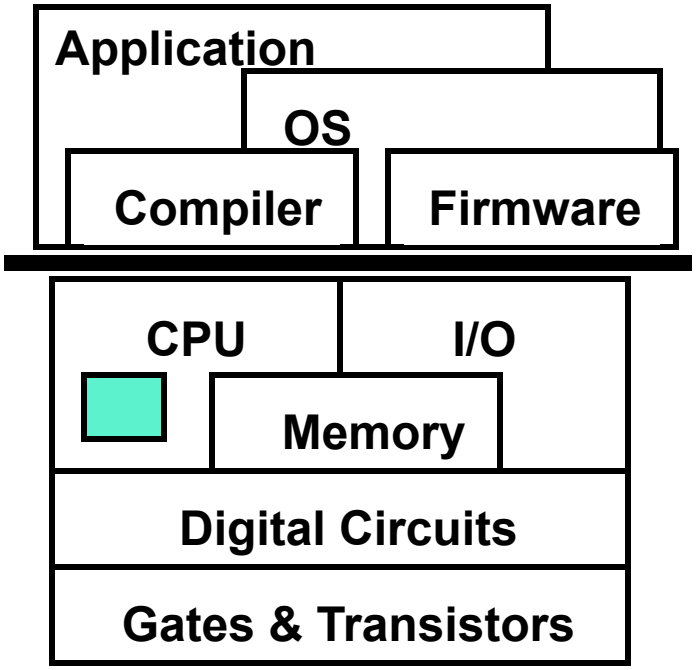


- Multiplier circuit optimizations also work for divider
  - Shift Remainder left and do 16-bit subtractions
  - Combine Quotient with right (unused) half of Remainder
  - Booth and modified Booth analogs (but really nasty)
- Multiplication and division in one circuit (how?)

Summary of Integer Arithmetic and ALU

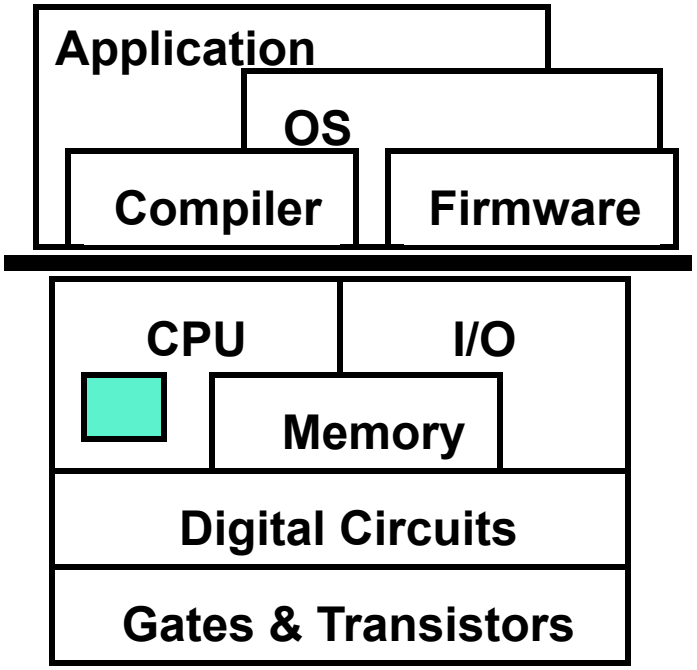
- Addition
  - Half adder full adder, ripple carry
  - Fast addition: carry select and carry lookahead
- Subtraction as addition
- Barrel shifter and shift registers
- Multiplication
  - N-step multiplication (3 refined implementations)
  - Booth's algorithm and N/2-step multiplication
- Division

This Unit: Arithmetic and ALU Design



- Integer Arithmetic and ALU
  - Binary number representations
  - Addition and subtraction
  - The integer ALU
  - Shifting and rotating
  - Multiplication
  - Division
- Floating Point Arithmetic
  - Binary number representations
  - FP arithmetic
  - Accuracy

Floating Point Arithmetic



- Formats
  - Precision and range
  - IEEE 754 standard
- Operations
  - Addition and subtraction
  - Multiplication and division
- Error analysis
  - Error and bias
  - Rounding and truncation
- Only scientists care?

Floating Point (FP) Numbers

- **Floating point numbers**: numbers in scientific notation
  - Two uses

- Use #1: real numbers (numbers with non-zero fractions)
  - 3.1415926…
  - 2.1878…
  - 9.8
  - $6.62 * 10^{-34}$
  - 5.875

- Use #2: really big numbers
  - $3.0 * 10^{8}$
  - $6.02 * 10^{23}$

The World Before Floating Point

- Early computers were built for scientific calculations
  - ENIAC: ballistic firing tables
- But didn't have primitive floating point data types
  - Circuits were big
  - Many accuracy problems
- Programmers built **scale factors** into programs
  - Large constant multiplier turns all FP numbers to integers
  - Before program starts, inputs multiplied by scale factor **manually**
  - After program finishes, outputs divided by scale factor **manually**
  - **Yuck!**

The Fixed Width Dilemma

- "Natural" arithmetic has infinite width
  - Infinite number of integers
  - Infinite number of reals
  - Infinitely more reals than integers (head… spinning…)

- Hardware arithmetic has finite width N (e.g., 16, 32, 64)
  - Can represent 2^N numbers
- If you could represent 2^N integers, which would they be?
  - Easy! The 2^{N-1} on either size of 0
- If you could represent 2^N reals, which would they be?
  - 2^N reals from 0 to 1, not too useful
  - 2^N powers of two (1, 2, 4, 8, …), also not too useful
  - Something in between: yes, but what?

Range and Precision

- **Range**
  - Distance between largest and smallest representable numbers
  - Want big range
- **Precision**
  - Distance between two consecutive representable numbers
  - Want small precision

- In fixed bit width, can't have unlimited both

Scientific Notation

- **Scientific notation**: good compromise
  - Number [S,F,E] = S * F * 2^E
  - S: **sign**
  - F: **significand** (fraction)
  - E: **exponent**
  - **"Floating point"**: binary (decimal) point has different magnitude

  + "Sliding window" of precision using notion of **significant digits**
    - Small numbers very precise, many places after decimal point
    - Big numbers are much less so, not all integers representable
    - But for those instances you don't really care anyway
  – Caveat: most representations are just approximations
    - Sometimes weirdos like 0.9999999 or 1.0000001 come up
    + But good enough for most purposes

IEEE 754 Standard Precision/Range

- **Single precision**: `float` in C
  - 32-bit: 1-bit sign + 8-bit exponent + 23-bit significand
  - Range: $2.0 * 10^{-38} < N < 2.0 * 10^{38}$
  - Precision: ~7 significant (decimal) digits

- **Double precision**: `double` in C
  - 64-bit: 1-bit sign + 11-bit exponent + 52-bit significand
  - Range: $2.0 * 10^{-308} < N < 2.0 * 10^{308}$
  - Precision: ~15 significant (decimal) digits

- Numbers $>10^{308}$ don't come up in many calculations
  - 10^{80} ~ number of atoms in universe

How Do Bits Represent Fractions?

- **Sign**: 0 or 1 → easy
- **Exponent**: signed integer → also easy
- **Significand**: unsigned fraction → not obvious!

- How do we represent integers?
  - Sums of positive powers of two
  - S-bit unsigned integer A: $A_{S-1}2^{S-1} + A_{S-2}2^{S-2} + \ldots + A_1 2^1 + A_0 2^0$
- So how can we represent fractions?
  - Sums of **negative powers of two**
  - S-bit unsigned fraction A: $A_{S-1}2^0 + A_{S-2}2^{-1} + \ldots + A_1 2^{-S+2} + A_0 2^{-S+1}$
  - More significant bits correspond to larger multipliers

Some Examples

- What is 5 in floating point?
  - Sign: 0
  - $5 = 1.25 * 2^2$
  - Significand: $1.25 = 1*2^0 + 1*2^{-2}$ = 101 0000 0000 0000 0000 0000
  - Exponent: 2 = 0000 0010

- What is –0.5 in floating point?
  - Sign: 1
  - $0.5 = 0.5 * 2^0$
  - Significand: $0.5 = 1*2^{-1}$ = 010 0000 0000 0000 0000 0000
  - Exponent: 0 = 0000 0000

Normalized Numbers

- Notice
  - 5 is $1.25 * 2^2$
  - But isn't it also $0.625 * 2^3$ and $0.3125 * 2^4$ and …?
  - With 8-bit exponent, we can have 8 representations of 5
- Multiple representations for one number is bad idea
  - Would lead to computational errors
  - Would waste bits
- Solution: choose **normal (canonical) form**
  - Disallow de-normalized numbers (some exceptions later)
  - IEEE 754 normal form: coefficient of 2^0 is always 1
    - Similar to scientific notation: one non-zero digit left of decimal
  - Normalized representation of 5 is $1.25 * 2^2$ ($1.25 = \mathbf{1*2^0}+1*2^{-2}$)
  - $0.625 * 2^3$ is de-normalized ($0.625 = \mathbf{0*2^0}+1*2^{-1}+ 1*2^{-3}$)

More About Normalization

- What is –0.5 in **normalized** floating point?
  - Sign: 1
  - $0.5 = 1 * 2^{-1}$
  - Significand: $1 = 1*2^0 = 100\ 0000\ 0000\ 0000\ 0000\ 0000$
  - Exponent: -1 = 1111 1111 (assuming 2's complement for now)

- IEEE 754: no need to represent coefficient of 2^0 explicitly
  - It's always 1
  + Buy yourself an extra bit of precision
    - Pretty cute trick

- Problem: what about 0?
  - How can we represent 0 if 2^0 is always implicitly 1?

IEEE 754: The Whole Story

- **Exponent**: signed integer → not so fast
- Exponent represented in **excess** or **bias** notation
  - N-bits typically can represent signed numbers from -2^{N-1} to $2^{N-1}-1$
  - But in IEEE 754, they represent exponents from -2^{N-1}**+2** to $2^{N-1}-1$
  - And they represent those as unsigned with an implicit $2^{N-1}-1$ added
    - Implicit added quantity is called the **bias**
    - Actual exponent is $E-(2^{N-1}-1)$
- Example: single precision (8-bit exponent)
  - Bias is 127, exponent range is –126 to 127
  - –126 is represented as 1 = 0000 0001
  - 127 is represented as 254 = 1111 1110
  - 0 is represented as 127 = 0111 1111
  - 1 is represented as 128 = 1000 0000

IEEE 754: Continued

- Notice: two exponent bit patterns are "unused"

- **0000 0000**: represents de-normalized numbers
  - Numbers that have implicit 0 (rather than 1) in 2^0
  - Zero is a special kind of de-normalized number
    - + Exponent is all 0s, significand is all 0s
    - – There are both +0 and –0, but they are considered the same
  - Also represent numbers smaller than smallest normalized numbers

- **1111 1111**: represents infinity and NaN
  - ± infinities have 0s in the significand
  - ± NaNs do not

IEEE 754: To Infinity and Beyond

- What are infinity and NaN used for?
  - To allow operations to proceed past overflow/underflow situations
  - **Overflow**: operation yields exponent greater than $2^{N-1}-1$
  - **Underflow**: operation yields exponent less than $-2^{N-1}+2$

- IEEE 754 defines operations on infinity and NaN
  - N / 0 = infinity
  - N / infinity = 0
  - 0 / 0 = NaN
  - Infinity / infinity = NaN
  - Infinity – infinity = NaN
  - Anything and NaN = NaN
  - Will not test you on these rules

IEEE 754: Final Format

- Biased exponent
- Normalized significand
- Exponent uses more significant bits than significand
  - Helps when comparing FP numbers
  - Exponent bias notation helps there too – why?

- Every computer since about 1980 supports this standard
  - Makes code portable (at the source level at least)
  - Makes hardware faster (stand on each other's shoulders)

Floating Point Arithmetic

- We will look at
  - Addition/subtraction
  - Multiplication/division

- Implementation
  - Basically, integer arithmetic on significand and exponent
    - Using integer ALUs
  - Plus extra hardware for normalization

- To help us here, look at toy "quarter" precision format
  - 8 bits: 1-bit sign + 3-bit exponent + 4-bit significand
  - Bias is 3 $(= 2^{N-1} - 1)$

FP Addition

- Assume
  - A represented as bit pattern [S_A, E_A, F_A]
  - B represented as bit pattern [S_B, E_B, F_B]
- What is the bit pattern for A+B [S_{A+B}, E_{A+B}, F_{A+B}]?
  - [S_A+S_B, E_A+E_B, F_A+F_B]? Nope!
  - So what is it then?

FP Addition Decimal Example

- Let's look at a decimal example first: 99.5 + 0.8
  - $9.95*10^{1} + 8.0*10^{-1}$

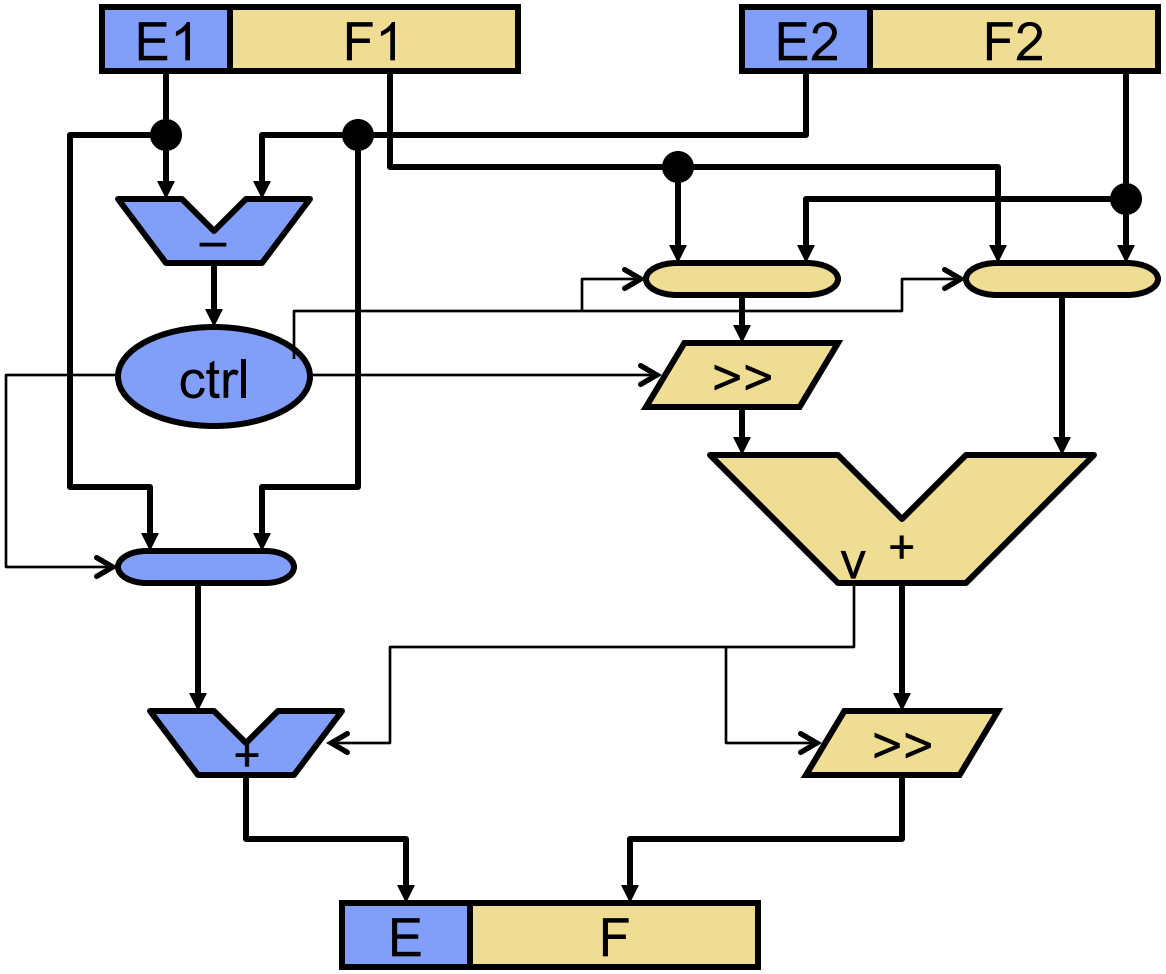
- Step I: **align exponents (if necessary)**
  - **Temporarily** de-normalize operand with smaller exponent
  - Add 2 to its exponent → must shift significand right by 2
  - $8.0*10^{-1} \rightarrow \mathbf{0.08*10^{1}}$
- Step II: **add significands**
  - $9.95*10^{1} + 0.08*10^{1} \rightarrow \mathbf{1}0.03*10^{1}$
- Step III: **normalize result**
  - Shift significand right by 1 and then add 1 to exponent
  - $10.03*10^{1} \rightarrow 1.003*10^{2}$

FP Addition (Quarter Precision) Example

- Now a binary "quarter" example: 7.5 + 0.5
  - $7.5 = 1.875*2^2 = 0\ 101\ \mathbf{1}1110$ (the **1** is the implicit leading 1)
    - $1.875 = 1*2^0+1*2^{-1}+1*2^{-2}+1*2^{-3}$
  - $0.5 = 1*2^{-1} = 0\ 010\ \mathbf{1}0000$

- Step I: **align exponents (if necessary)**
  - 0 010 **1**0000 → 0 101 **0**0010
  - Add 3 to exponent → shift significand right by 3
- Step II: **add significands**
  - 0 101 **1**1110 + 0 101 **0**0010 = 0 101 **10**0000
- Step III: **normalize result**
  - 0 101 **10**0000 → 0 110 **1**0000
  - Shift significand right by 1 → add 1 to exponent

FP Addition Hardware



What About FP Subtraction?

- Or addition of negative quantities for that matter
  - How to subtract significands that are not in TC form?
  - Can we still use an adder?

- Trick: internally and temporarily convert to TC
  - Add "phantom" –2 in front ($-1*2^1$)
  - Use standard negation trick
  - Add as usual
  - If phantom –2 bit is 1, result is negative
    - Negate it using standard trick again, flip result sign bit
    - Then ignore "phantom" bit (which is now 0 anyway)
  - You'll want to try this at home!

FP Multiplication

- Assume
  - A represented as bit pattern [S_A, E_A, F_A]
  - B represented as bit pattern [S_B, E_B, F_B]
- What is the bit pattern for A*B [S_{A*B}, E_{A*B}, F_{A*B}]?
  - This one is actually a little easier (conceptually) than addition
    - Scientific notation is logarithmic
    - **In logarithmic form: multiplication is addition**
- [S_AXOR S_B, E_A+E_B, F_A*F_B]? Pretty much, except for…
  - Normalization
  - Addition of exponents in biased notation (must subtract bias)
  - Tricky: when multiplying two normalized F-bit significands…
    - Where is the binary point?

FP Division

- Assume
  - A represented as bit pattern [S_A, E_A, F_A]
  - B represented as bit pattern [S_B, E_B, F_B]
- What is the bit pattern for A/B [$S_{A/B}$, $E_{A/B}$, $F_{A/B}$]?
- [S_AXOR S_B, E_A–E_B, F_A/F_B]? Pretty much, again except for…
  - Normalization
  - Subtraction of exponents in biased notation (must add bias)
  - Binary point placement
  - No need to worry about remainders, either

- A little bit of irony
  - Multiplication/division roughly same complexity for FP and integer
  - Addition/subtraction much more complicated for FP than integer

Accuracy

- Remember our decimal addition example?
  - $9.95*10^{1} + 8.00*10^{-1} \rightarrow 1.00\mathbf{3}*10^{2}$
  - Extra decimal place caused by de-normalization…
  - But what if our representation only has two digits of precision?
    - What happens to the **3**?
  - Corresponding binary question: what happens to extra 1s?

- Solution: **round**
  - Option I: **round down (truncate)**, no hardware necessary
  - Option II: **round up (round)**, need an incrementer
    - Why rounding up called round?
    - Because an extra 1 is half-way, which is rounded up

More About Accuracy

- Problem with both truncation and rounding
  - They cause errors to **accumulate**
    - E.g., if always round up, result will gradually "crawl" upwards
- One solution: **round to nearest even**
  - If un-rounded LSB is 1 → round up (01**1** → 10)
  - If un-rounded LSB is 0 → round down (00**1** → 00)
  - Round up half the time, down other half → overall error is stable
- Another solution: **multiple intermediate precision bits**
  - IEEE 754 defines 3: guard + round + sticky
    - Guard and round are shifted by de-normalization as usual
    - Sticky is 1 if any shifted out bits are 1
  - Round up if 101 or higher, round down if 011 or lower
  - Round to nearest even if 100

Numerical Analysis

- Accuracy problems sometimes get bad
  - Addition of big and small numbers
  - Subtraction of big numbers
  - Example, what's $1*10^{30} + 1*10^{0} - 1*10^{30}$?
    - Intuitively: $1*10^{0} = 1$
    - But: $(1*10^{30} + 1*10^{0}) - 1*10^{30} = (1*10^{30} - 1*10^{30}) = 0$

- **Numerical analysis**: field formed around this problem
  - Bounding error of numerical algorithms
  - Re-formulating algorithms in a way that bounds numerical error

One Last Thing About Accuracy

- Suppose you added two numbers and came up with
  - 0 101 **1**1111 **101**
  - What happens when you round?
  - Number becomes denormalized… arrrrgggghhh

- FP adder actually has six steps, not three
  - Align exponents
  - Add/subtract significands
  - Re-normalize
  - **Round**
  - **Potentially re-normalize again**
  - **Potentially round again**

Accuracy, Shmaccuracy?

- Only scientists care? Au contraire
- Intel 486 used equivalent of Modified Booth's for division
  - Generate multiple quotient bits per step
  - Requires you to guess quotient bits and adjust later
  - Guess taken from a lookup table implemented as PLA
- Along came Pentium
  - PLA was optimized to return 0 for "impossible" table indices
  - Which turned out not to be "impossible" after all
  - Result: precision errors in 4^{th}–15^{th} decimal places for some divisors
- "**Pentium fdiv bug**" is born

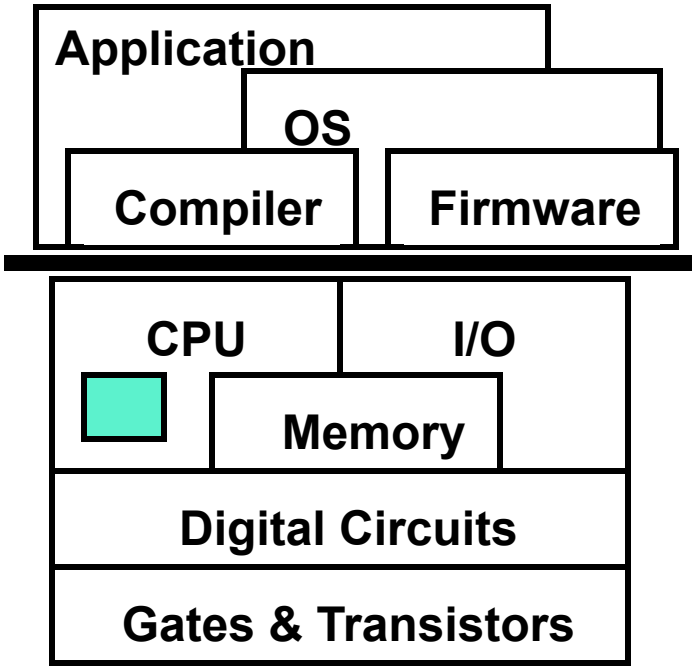
Pentium FDIV Bug

- Pentium shipped in August 1994
- Intel actually knew about the bug in July
  - But calculated that delaying the project a month would cost ~$1M
  - And that in reality only a dozen or so people would encounter it
  - They were right… but one of them took the story to EE Times
- By November 1994, firestorm was full on
  - IBM said that typical Excel user would encounter bug every month
    - Assumed 5K divisions per second around the clock
  - People believed the story
  - IBM stopped shipping Pentium PCs
- By December 1994, Intel promises full recall
  - Total cost: ~$550M
  - All for a bug which in reality maybe affected a dozen people

Summary of Floating Point

- FP representation
  - $S*F*2^E$
  - IEEE754 standard
  - Representing fractions
  - Normalized numbers
- FP operations
  - Addition/subtraction: hard
  - Multiplication/division: logarithmic no harder than integer
- Accuracy problems
  - Rounding and truncation
- Upshot: FP hardware is tough
  - Thank lucky stars that ECE 152 project has no FP

Unit Recap: Arithmetic and ALU Design



- Integer Arithmetic and ALU
  - Binary number representations
  - Addition and subtraction
  - The integer ALU
  - Shifting and rotating
  - Multiplication
  - Division
- Floating Point Arithmetic
  - Binary number representations
  - FP arithmetic
  - Accuracy