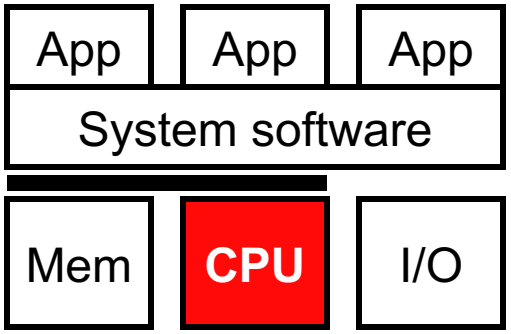
CIS 5710
Computer Organization and Design

Unit 3: Arithmetic

Based on slides by Profs. Amir Roth & Milo Martin & C.J. Taylor

This Unit: Arithmetic

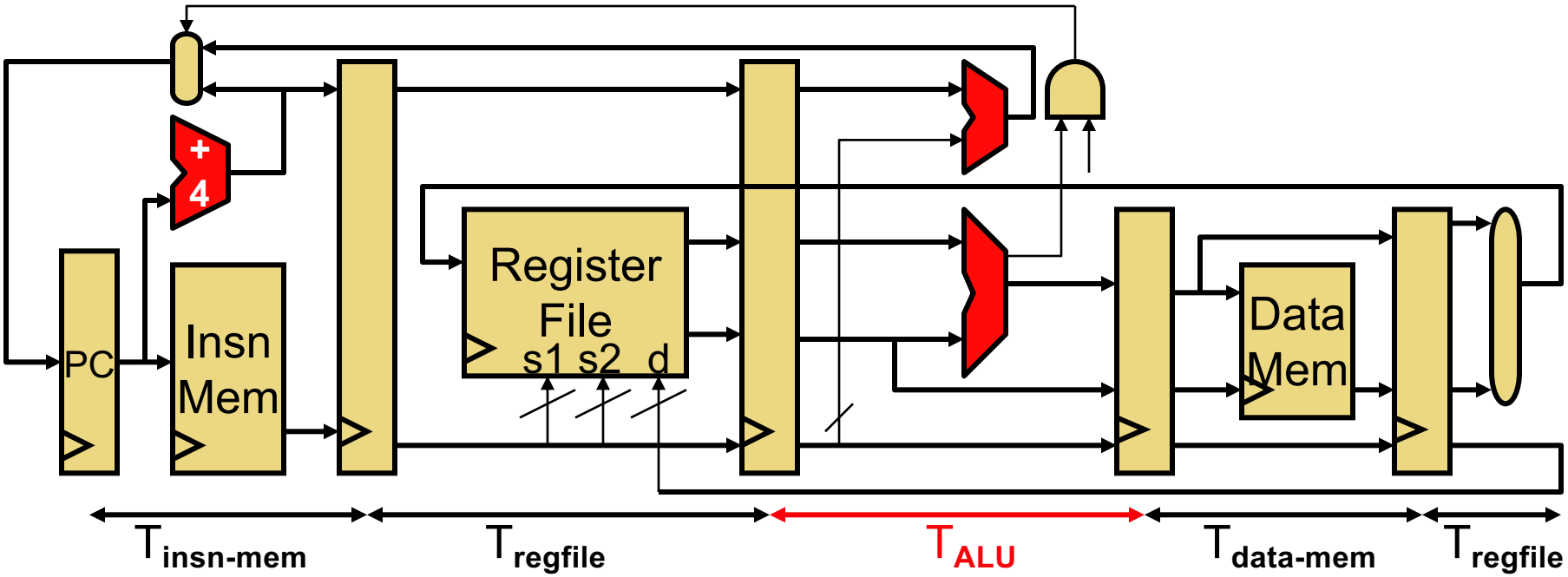


- A little review
  - Binary + 2s complement
  - Ripple-carry addition (RCA)
- Fast integer addition
  - Carry-select (CSeA)
  - Carry Lookahead Adder (CLA)
- Shifters
- Integer multiplication and division
- Floating point arithmetic

Readings

- P&H
  - Chapter 3

The Importance of Fast Arithmetic



- Addition of two numbers is most common operation
  - Programs use addition frequently
  - Loads and stores use addition for address calculation
  - Branches use addition to test conditions and calculate targets
  - All insns use addition to calculate default next PC
- Fast addition is critical to high performance

Review: Binary Integers

- Computers represent integers in binary (base2)

  `3 = 11, 4 = 100, 5 = 101, 30 = 11110`

  + Natural since only two values are represented
  - Addition, etc. take place as usual (carry the 1, etc.)

```
17 =    10001
+5 =      101
22 =    10110
```

- Some old machines use decimal (base10) with only 0/1

  `30 = 011 000`

  - Often called BCD (binary-coded decimal)
  – Unnatural for digital logic, implementation complicated & slow
  + Required for precise currency operations

Fixed Width

- On pencil and paper, integers have infinite width

- In hardware, integers have **fixed width**
  - N bits: 16, 32 or 64
  - LSB is 2^0, MSB is 2^{N-1}

  - **Range**: 0 to 2^N–1

  - Numbers >2^N represented using multiple fixed-width integers
    - In software (e.g., Java BigInteger class)

What About Negative Integers?

- **Sign/magnitude**
  - Unsigned plus one bit for sign
    `10 = 00000101, -10 = 10000101`
  + Matches our intuition from "by hand" decimal arithmetic
  – representations of both +0 and –0
  – Addition is difficult
  - symmetric range: $-(2^{N-1}-1)$ to $2^{N-1}-1$

- Option II: **two's complement (2C)**
  - Leading 0s mean positive number, leading 1s negative
    `10 = 00001010, -10 = 11110110`
  + One representation for 0 (all zeroes)
  + Easy addition
  - asymmetric range: $-(2^{N-1})$ to $2^{N-1}-1$

The Tao of 2C

- How did 2C come about?
  - "Let's design a representation that makes addition easy"
  - Think of subtracting 10 from 0 by hand with 8-bit numbers
  - Have to "borrow" 1s from some imaginary leading 1

```
  0 = 100000000
-10 =  00000010
-10 = 011111110
```

  - Now, add the conventional way…

```
-10 =  11111110
+10 =  00000010
  0 = 100000000
```

Still More On 2C

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

- 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

Addition

1st Grade: Decimal Addition

```
  1
 43
+29
 72
```

- Repeat N times
  - Add least significant digits and any overflow from previous add
  - Carry "overflow" to next decimal place
    - **Overflow**: any digit other than least significant of sum
  - Both two addends and the sum are shifted right by one

- Sum of two N-digit numbers can yield an N+1 digit number

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

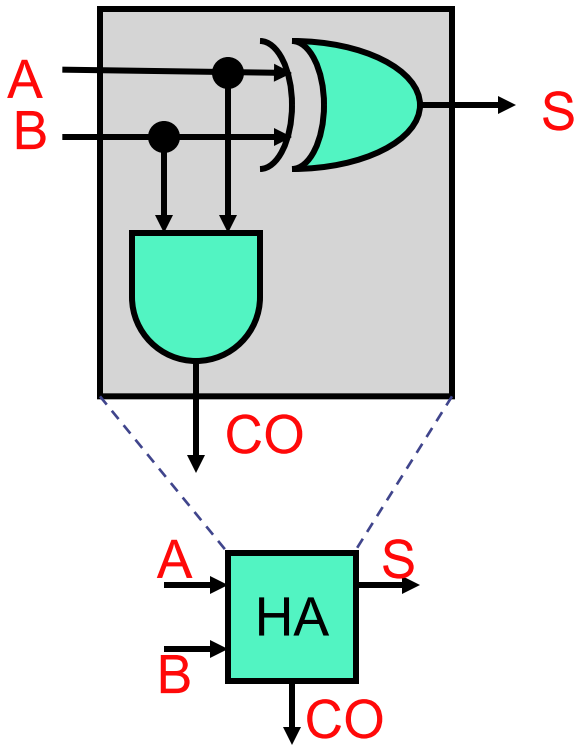
– More steps (smaller base)

\+ Each one is simpler (adding just 1 and 0)

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	=	C0	S
0	0	=	0	0
0	1	=	0	1
1	0	=	0	1
1	1	=	1	0

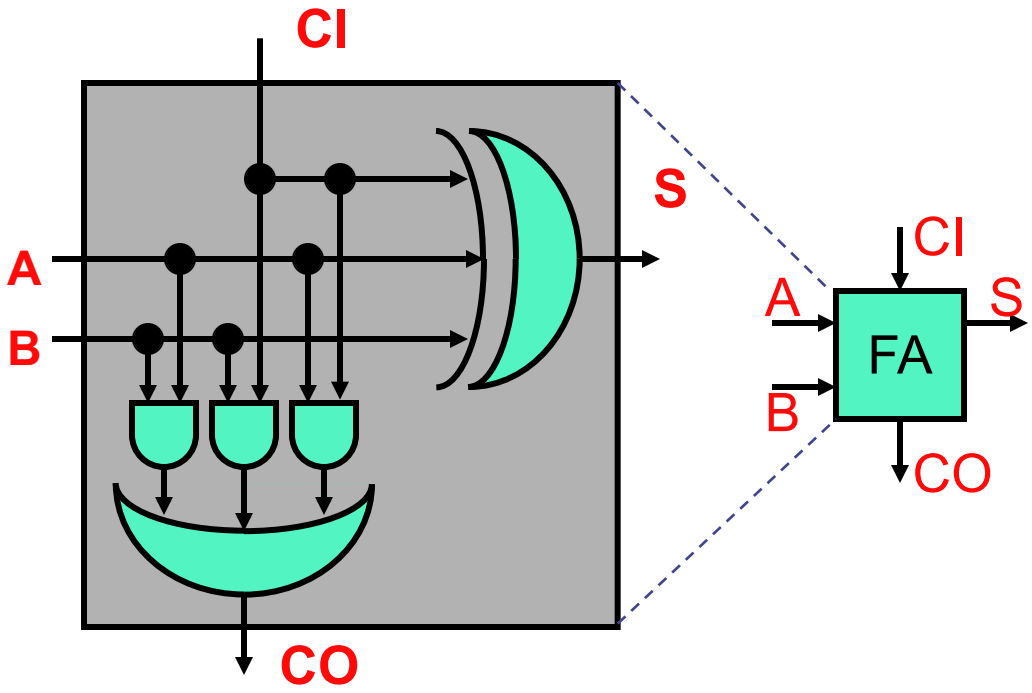


- **S = A^B**
- **CO (carry out) = AB**
- This is called a **half adder**

The Other Half

- We could chain half adders together, but to do that…
  - Need to incorporate a carry out from previous adder

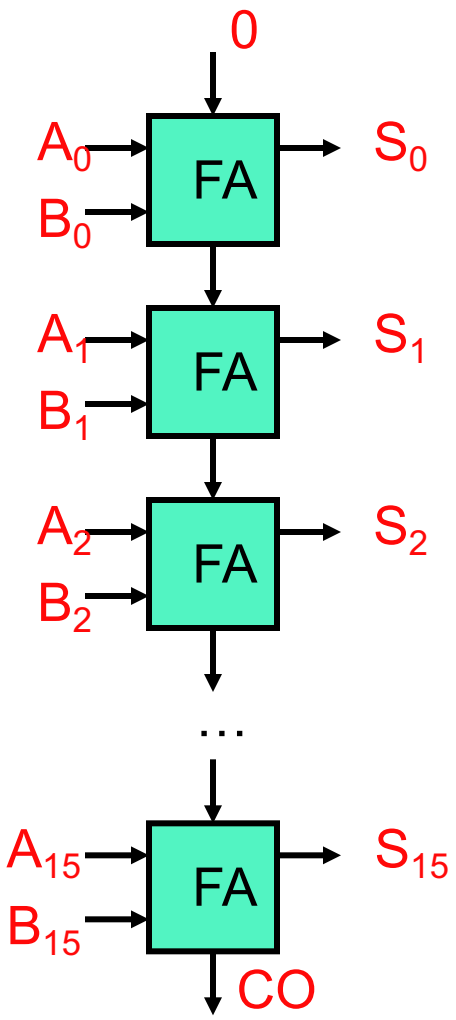
C	A	B	=	C0	S
0	0	0	=	0	0
0	0	1	=	0	1
0	1	0	=	0	1
0	1	1	=	1	0
1	0	0	=	0	1
1	0	1	=	1	0
1	1	0	=	1	0
1	1	1	=	1	1



- **S = C'A'B | C'AB' | CA'B' | CAB = C ^ A ^ B**
- **CO = C'AB | CA'B | CAB' | CAB = CA | CB | AB**
- This is called a **full adder**

Ripple-Carry Adder

- N-bit **ripple-carry** adder
  - N 1-bit full adders "chained" together
    - $CO_0 = CI_1$, $CO_1 = CI_2$, etc.
    - $CI_0 = 0$
    - CO_{N-1} is carry-out of entire adder
      - $CO_{N-1} = 1 \rightarrow$ "overflow"
- Example: 16-bit ripple carry adder
  - How fast is this?
  - How fast is an N-bit ripple-carry adder?



Quantifying Adder Delay

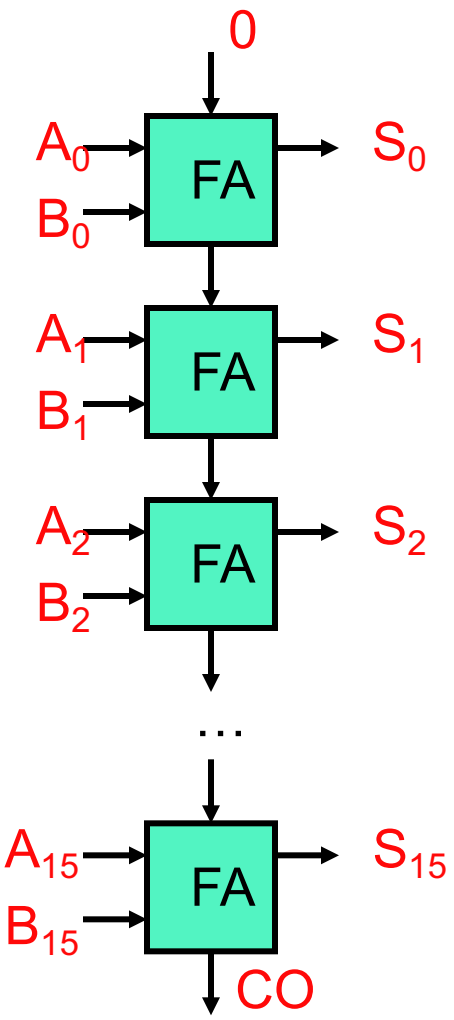
- Combinational logic dominated by **gate delays**
  - How many gates between input and output?
  - Array storage dominated by wire delays
  - Longest delay or **critical path** is what matters

- Can implement any combinational function in "2" logic levels
  - 1 level of AND + 1 level of OR (PLA)
  - NOTs are "free": push to input (DeMorgan's) or read from latch
  - Example: delay(FullAdder) = 2
    - d(CarryOut) = delay(AB | AC | BC)
    - d(Sum) = d(A ^ B ^ C) = d(AB'C' | A'BC' | A'B'C | ABC) = 2
    - Note '^' means Xor (just like in C & Java)

- Caveat: "2" assumes gates have few (<8 ?) inputs

Ripple-Carry Adder Delay

- Longest path is to CO_{15} (or S_{15})
  - delay(CO_{15}) = 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**
- **d(CO_{N-1}) = 2N**
  - **Too slow!**
  - **Linear in number of bits**
- Number of gates is also linear



Division

4th Grade: Decimal Division

```
   9          //          quotient
3 |29         // divisor | dividend
 -27
   2         // 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

```
                    1001  = 9
3 |29 = 0011 |011101
  -24 =     - 011000
    5 =       000101
  - 3 =     - 000011
    2 =       000010
```

Binary Division Hardware

- 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

- Complication: **largest** q such that (q*divisor) < dividend
  - How do you know if (1*divisor) < dividend?
  - Human can "eyeball" this
  - Computer does not have eyeballs
    - it must subtract and see if result is negative

Software Divide Algorithm

- Can implement this algorithm in software
- Inputs: `dividend` and `divisor`
- Outputs: `quotient = dividend / divisor`
  `rem = dividend % divisor`

```
remainder = 0;
quotient = 0;
for (int i = 0; i < 32; i++) {
  remainder = (remainder << 1) | (dividend >> 31);
  if (remainder >= divisor) {
    quotient = (quotient << 1) | 1;
    remainder = remainder - divisor;
  } else {
    quotient = (quotient << 1) | 0;
  }
  dividend = dividend << 1;
}
```

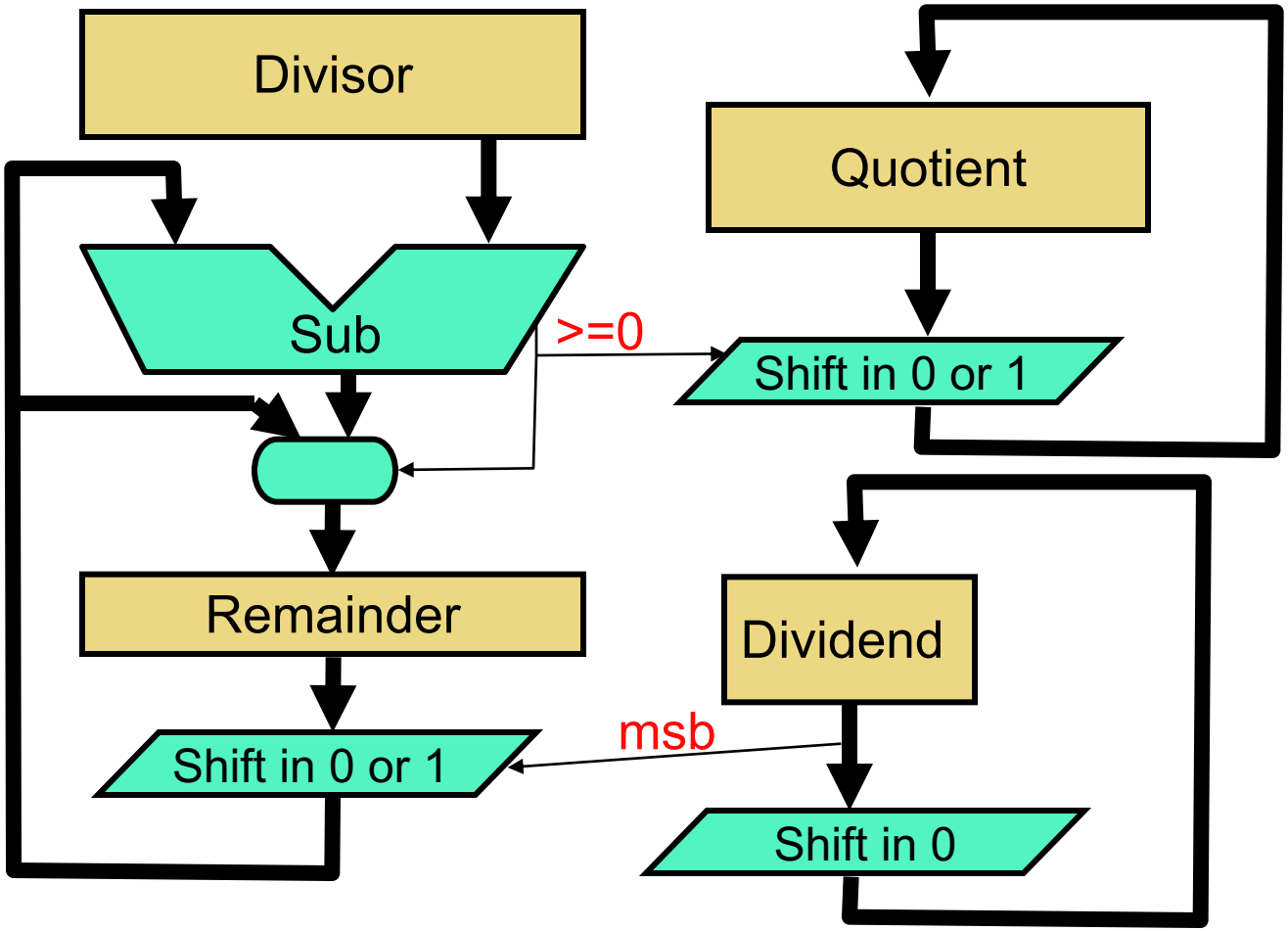
Divide Example

- Input: Divisor = `00011` , Dividend = `11101`

Step	Remainder	Quotient	Remainder	Dividend
0	00000	00000	00000	11101
1	00001	00000	00001	11010
2	00011	00001	00000	10100
3	00001	00010	00001	01000
4	00010	00100	00010	10000
5	00101	01001	00010	00000

- Result: Quotient: 1001, Remainder: 10

Divider Circuit



- N cycles for n-bit divide

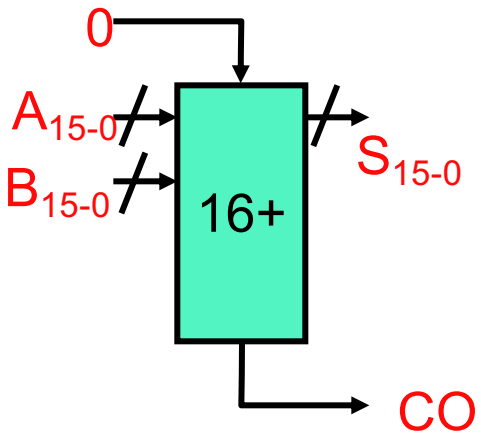
Fast Addition

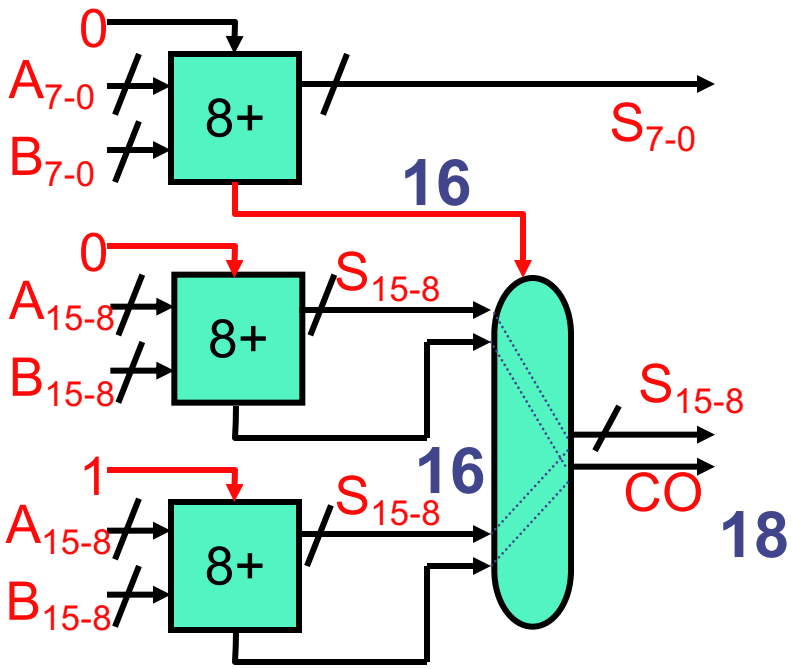
Theme: Hardware != Software

- Hardware can do things that software fundamentally can't
  - And vice versa (of course)
- In hardware, it's easier to trade **resources** for **latency**
- One example of this: **speculation**
  - Slow computation waiting for some slow input?
  - Input one of two things?
  - **Compute with both (slow), choose right one later (fast)**
- Does this make sense in software? Not on a single core
- Difference? hardware is parallel, software is sequential

Carry-Select Adder

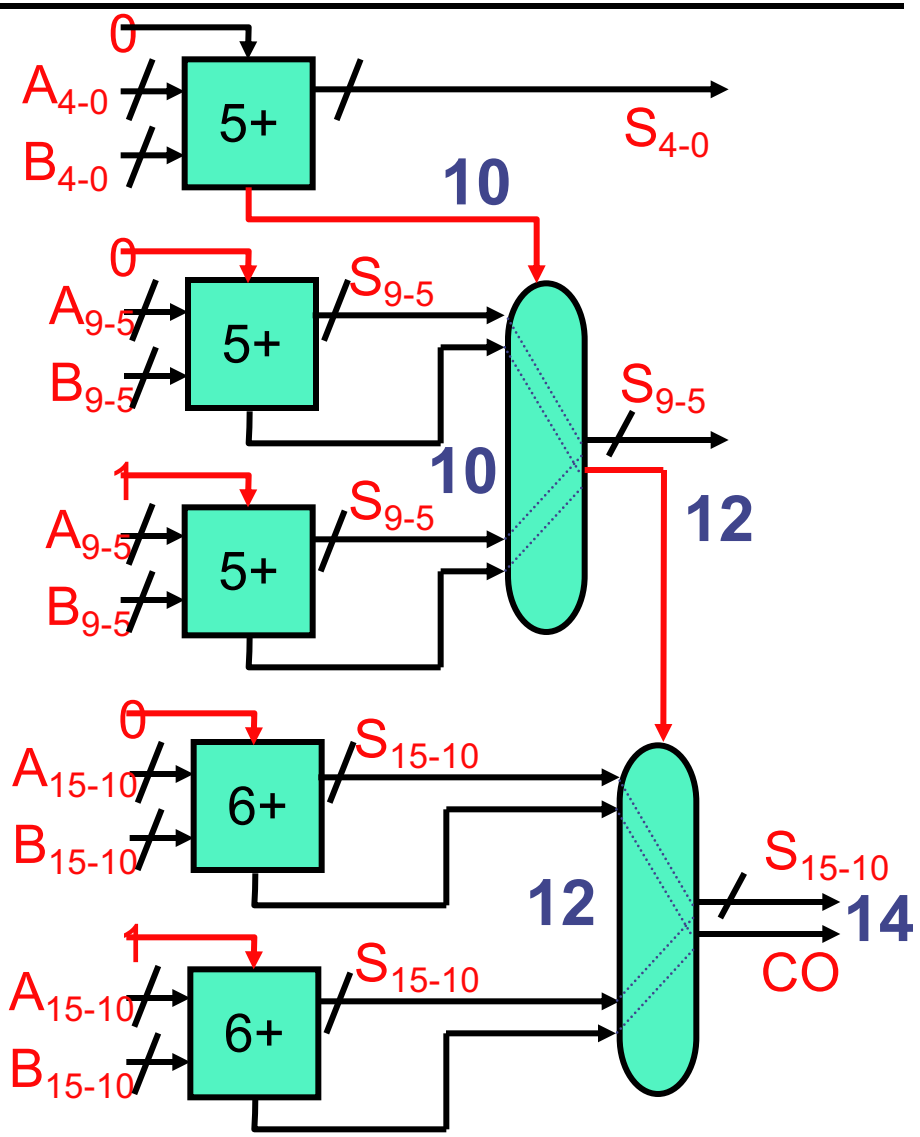
- **Carry-select adder**
  - Do $A_{15-8}+B_{15-8}$ twice, once assuming C_8 (CO_7) = 0, once = 1
  - Choose the correct one when CO_7 finally becomes available
  + Effectively cuts carry chain in half (break critical path)
  – Extra mux increases delay
  - Delay?





Multi-Segment Carry-Select Adder

- Multiple segments
  - Example: 5, 5, 6 bit = 16 bit
- Hardware cost
  - Still mostly linear (~2x)
  - Compute each segment with 0 and 1 carry-in
  - Serial mux chain
- Delay
  - 5-bit adder (10) + Two muxes (4) = 14



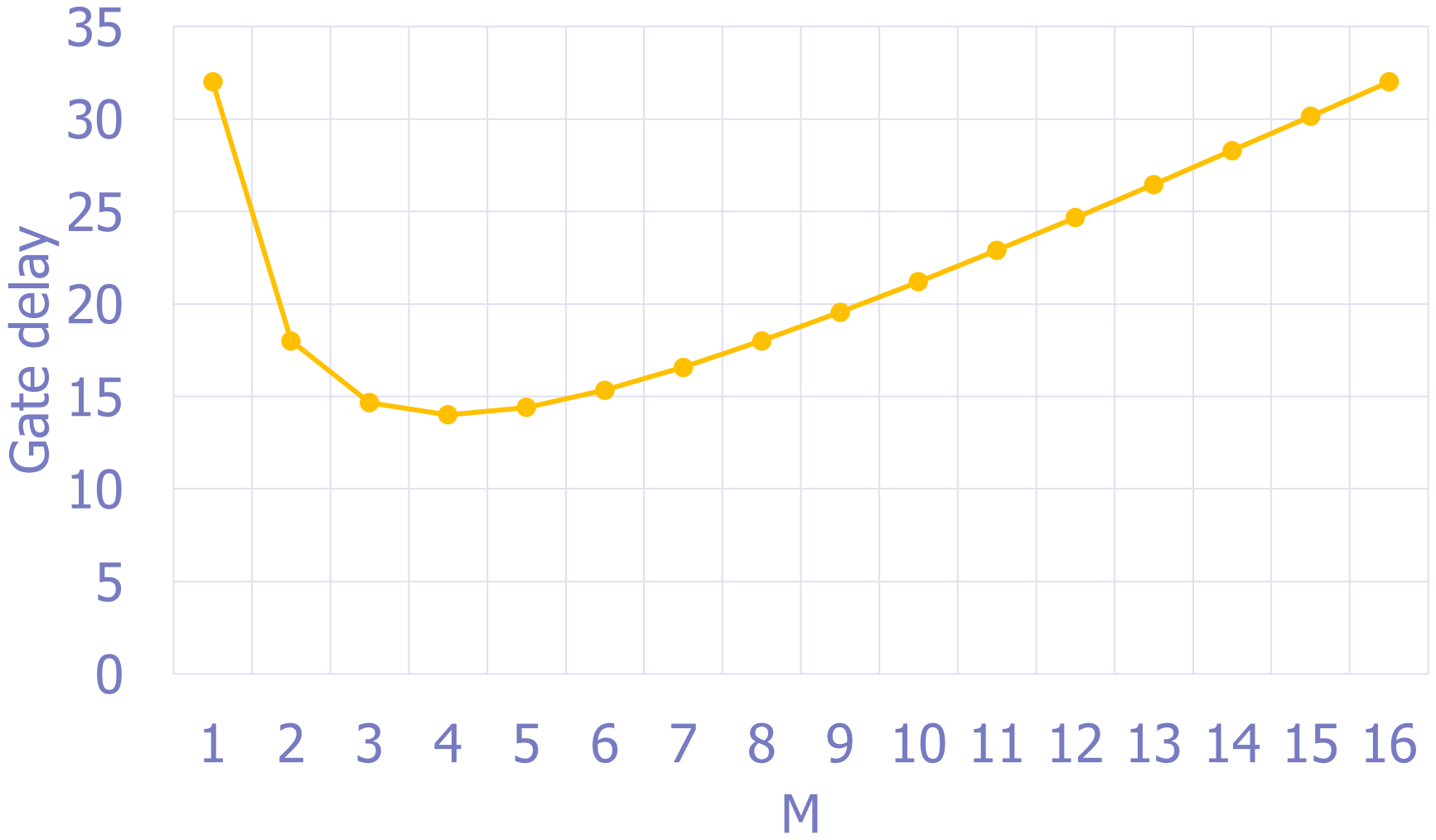
Carry-Select Adder Delay

- What is **two segment** carry-select adder delay?
  - $d(CO_{15}) = MAX(d(CO_{15-8}), d(CO_{7-0})) + 2$
  - $d(CO_{15}) = MAX(2*8, 2*8) + 2 =$ **18**
  - In general: **2*(N/2) + 2 = N+2** (vs **2N** for RCA)

- What about **four equal segments**?
  - Would it 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**

- N-bit adder in M equal pieces: **2*(N/M) + (M–1)*2**
  - 16-bit adder in 8 parts: 2*(16/8) + 7*2 = **18**

16b Carry-Select Adder Delay



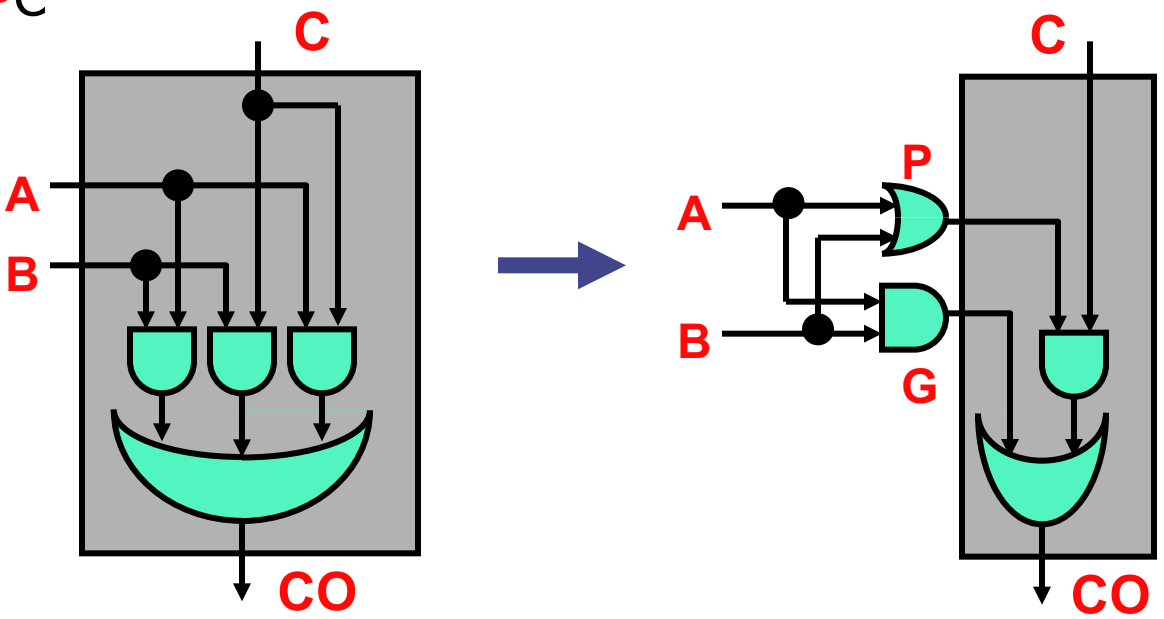
Another Option: Carry Lookahead

- Is carry-select adder as fast as we can go?
  - Nope!

- 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 (CLA)**

A & B → Generate & Propagate

- Let's look at the single-bit carry-out function
  - CO = AB|AC|BC
  - Factor out terms that use only A and B (available immediately)
  - CO =(**AB**)|(**A|B**)C
  - (**AB**): generates carry-out regardless of incoming C → rename to **G**
  - (**A|B**) : propagates incoming C → rename to **P**
  - CO = **G**|**P**C



Infinite Hardware CLA

- Can expand $C_{1...N}$ in terms of G's, P's, and C_0
  - Example: C_{16}
    - $C_{16} = G_{15} \mid P_{15}C_{15}$
    - $C_{16} = G_{15} \mid P_{15}(G_{14} \mid P_{14}C_{14})$
    - $C_{16} = G_{15} \mid P_{15}G_{14} \mid \ldots \mid P_{15}P_{14}\ldots P_2P_1G_0 \mid P_{15}P_{14}\ldots P_2P_1P_0C_0$
  - Similar expansions for C_{15}, C_{14}, etc.

- How much does this cost?
  - C_N needs: N AND's + 1 OR's, largest have N+1 inputs
  - $C_N \ldots C_1$ needs: **N*(N+1)/2** AND's + **N** OR's, max N+1 inputs
  - N=16: 152 total gates, max input 17
  - N=64: 2144 total gates, max input 65
- And how fast is it really?
  - Not that fast, unfortunately, 17-input gates are really slow

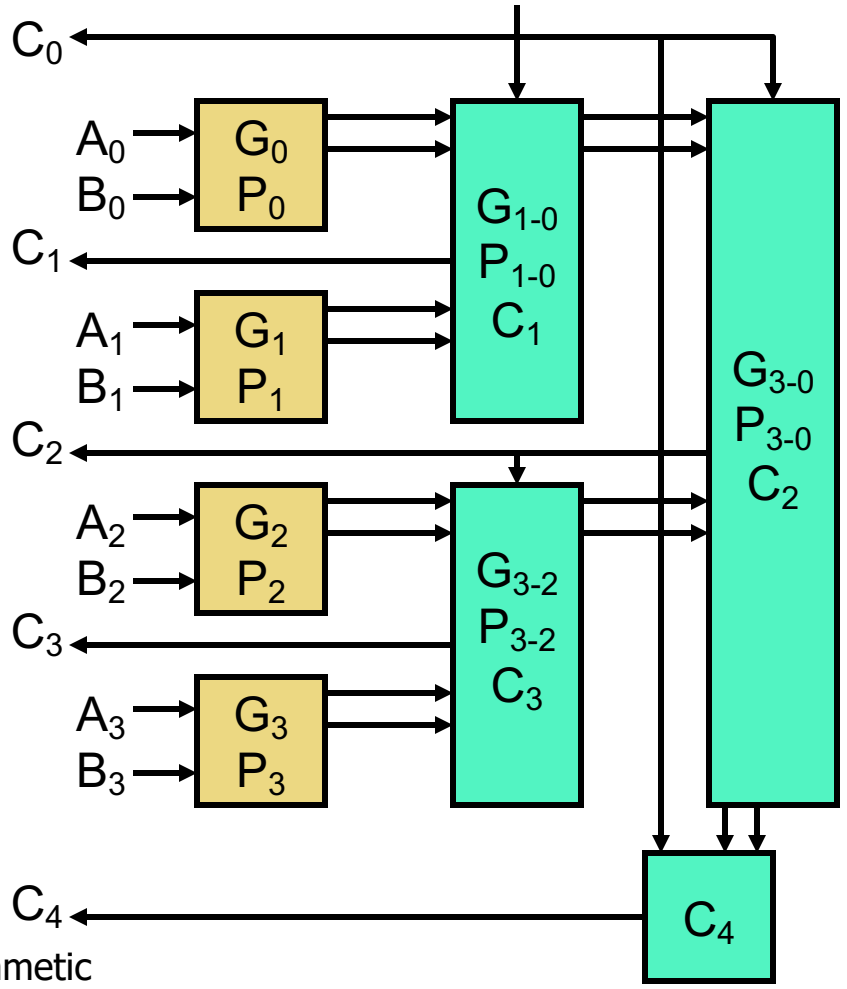
3b Infinite CLA example

Compromise: Multi-Level CLA

- **Ripple carry**
  - \+ Few small gates: 5N gates, max 3 inputs
  - – Laid in series: 2N (linear) latency
- **Infinite hardware CLA**
  - – Many big gates: N*(N+3)/2 additional gates, max N+1 inputs
  - \+ Laid in parallel: constant 4 latency
- Is there a compromise?
  - Reasonable number of small gates?
  - Sublinear (doesn't have to be constant) latency?
  - **Yes, multi-level CLA**: exploits hierarchy to achieve this

Carry Lookahead Adder (CLA)

- Calculate "propagate" and "generate" based on A, B
  - Not based on carry in
- Combine with tree structure
- High-level idea
  - **Tree gives logarithmic delay**
  - Reasonable area



Individual G & P → Windowed G & P

- **Windowed G/P**: useful abstraction for multi-level CLA
- Individual carry equations
  - $C_1 = G_0 \mid P_0C_0$, $C_2 = G_1 \mid P_1C_1$
- Infinite hardware CLA equations
  - $C_1 = G_0 \mid P_0C_0$
  - $C_2 = G_1 \mid P_1G_0 \mid P_1P_0C_0$
- Group terms into "windows"
  - $C_2 = \mathbf{(G_1 \mid P_1G_0)} \mid \mathbf{(P_1P_0)}C_0$
  - $C_2 = \mathbf{G_{1\text{-}0}} \mid \mathbf{P_{1\text{-}0}}C_0$
- $G_{1\text{-}0}$, $P_{1\text{-}0}$ are window G & P
  - a single bit summarizing information from all the bits in the window
  - $G_{1\text{-}0}$: carry-out generated by bits [1:0]
  - $P_{1\text{-}0}$: carry-out propagated by bits [1:0]
    - would a carry-in to the start of the window create a carry-out?

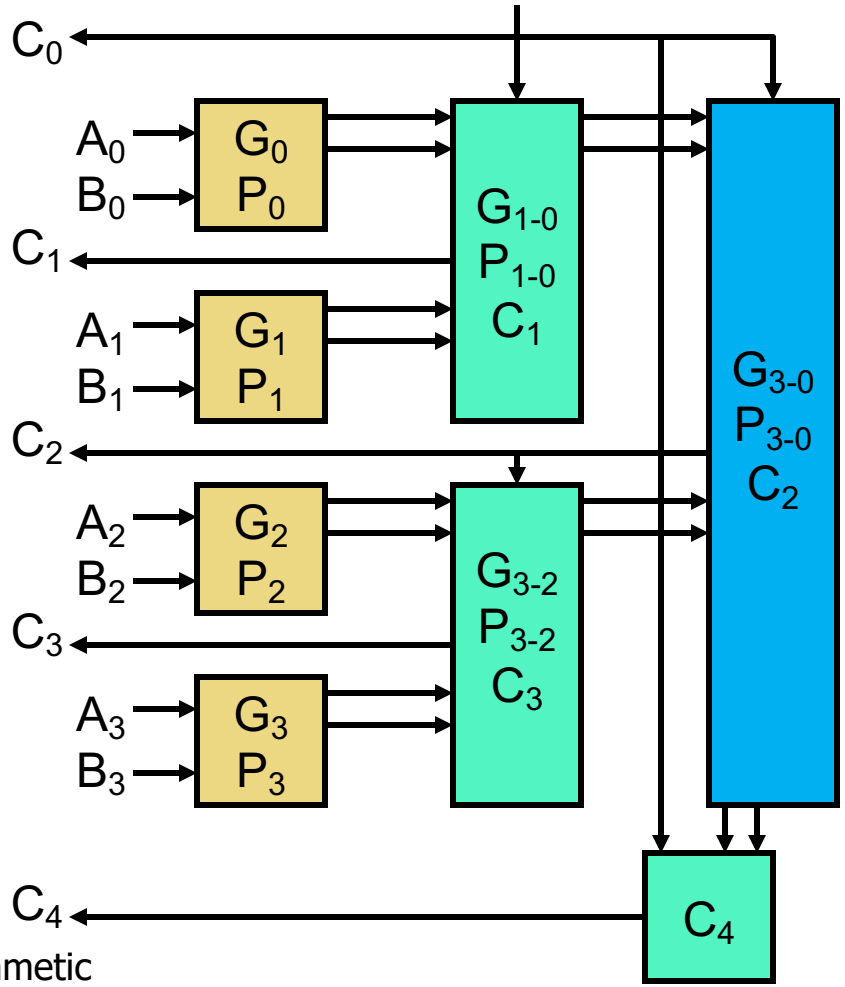
3b Windowed G&P Example

Two-Level CLA for 4-bit Adder

- Individual carry equations
  - $C_1 = G_0 \mid P_0C_0$, $C_2 = G_1 \mid P_1C_1$, $C_3 = G_2 \mid P_2C_2$, $C_4 = G_3 \mid P_3C_3$
- Infinite hardware CLA equations
  - $C_1 = G_0 \mid P_0C_0$
  - $C_2 = G_1 \mid P_1G_0 \mid P_1P_0C_0$
  - $C_3 = G_2 \mid P_2G_1 \mid P_2P_1G_0 \mid P_2P_1P_0C_0$
  - $C_4 = G_3 \mid P_3G_2 \mid P_3P_2G_1 \mid P_3P_2P_1G_0 \mid P_3P_2P_1P_0C_0$
- Hierarchical CLA equations
  - **First level**: expand C_2 using C_1, C_4 using C_3
    - $C_2 = G_1 \mid P_1(G_0 \mid P_0C_0) = (G_1 \mid P_1G_0) \mid (P_1P_0)C_0 = G_{1\text{-}0} \mid P_{1\text{-}0}C_0$
    - $C_4 = G_3 \mid P_3(G_2 \mid P_2C_2) = (G_3 \mid P_3G_2) \mid (P_3P_2)C_2 = G_{3\text{-}2} \mid P_{3\text{-}2}C_2$
  - **Second level**: expand C_4 using expanded C_2
    - $C_4 = G_{3\text{-}2} \mid P_{3\text{-}2}(G_{1\text{-}0} \mid P_{1\text{-}0}C_0) = (G_{3\text{-}2} \mid P_{3\text{-}2}G_{1\text{-}0}) \mid (P_{3\text{-}2}P_{1\text{-}0})C_0$
    - $C_4 = G_{3\text{-}0} \mid P_{3\text{-}0}C_0$

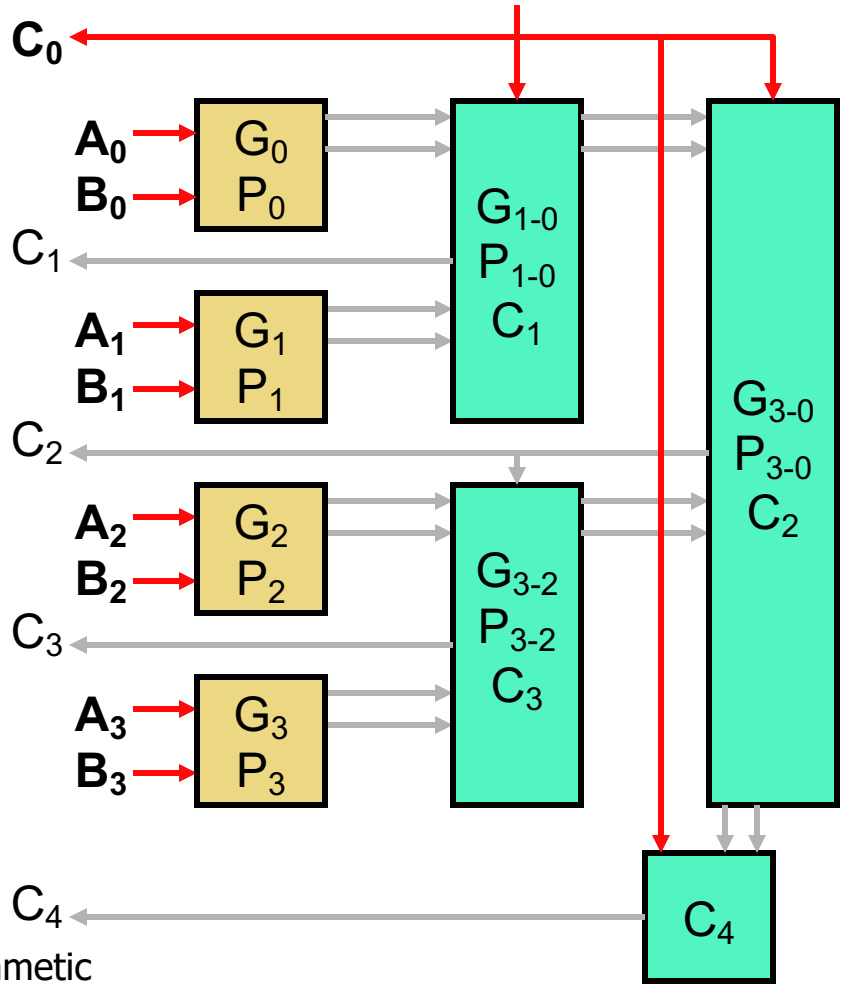
Two-Level CLA for 4-bit Adder

- Top first-level CLA block
  - Input: C_0, G_0, P_0, G_1, P_1
  - Output: C_1, G_{1-0}, P_{1-0}
- Bottom first-level CLA block
  - Input: C_2, G_2, P_2, G_3, P_3
  - Output: C_3, G_{3-2}, P_{3-2}
- Second-level CLA block
  - Input: C_0, G_{1-0}, P_{1-0}, G_{3-2}, P_{3-2}
  - Output: C_2, G_{3-0}, P_{3-0}
- These 3 blocks are "the same"
- C_4 block
  - Input: C_0, G_{3-0}, P_{3-0}
  - Output: C_4



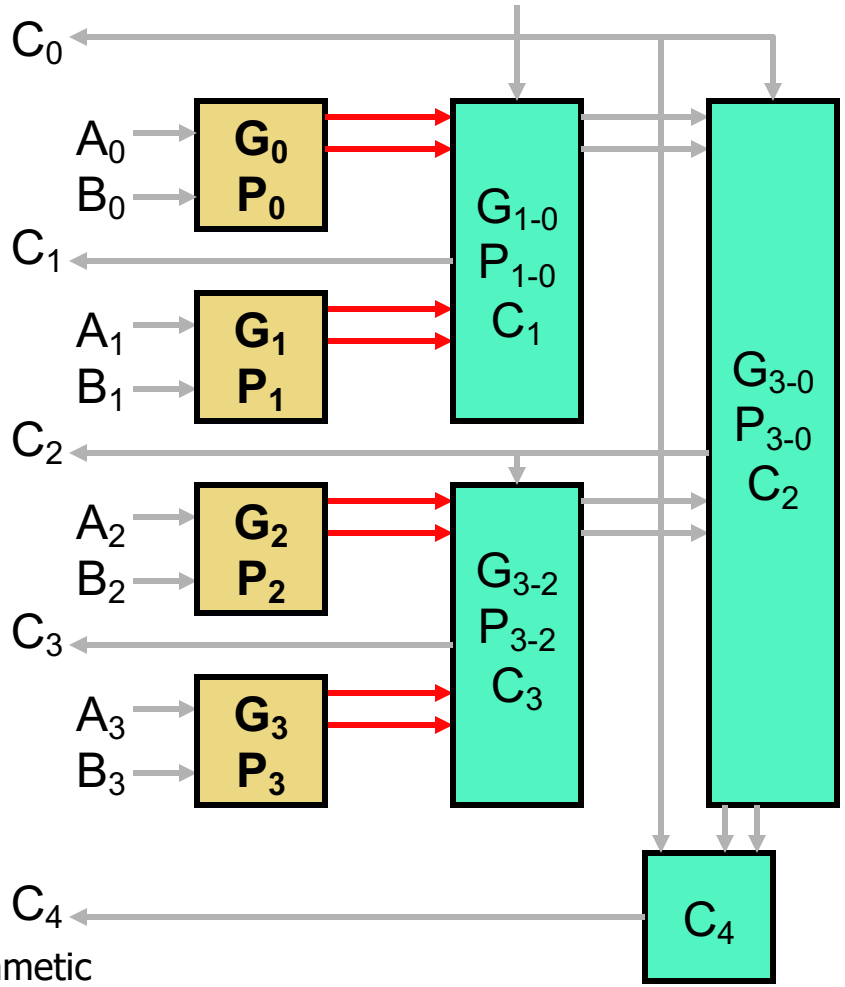
4-bit CLA Timing: d0

- Which signals are ready at 0 gate delays (d0)?
  - C_0
  - A_0, B_0
  - A_1, B_1
  - A_2, B_2
  - A_3, B_3



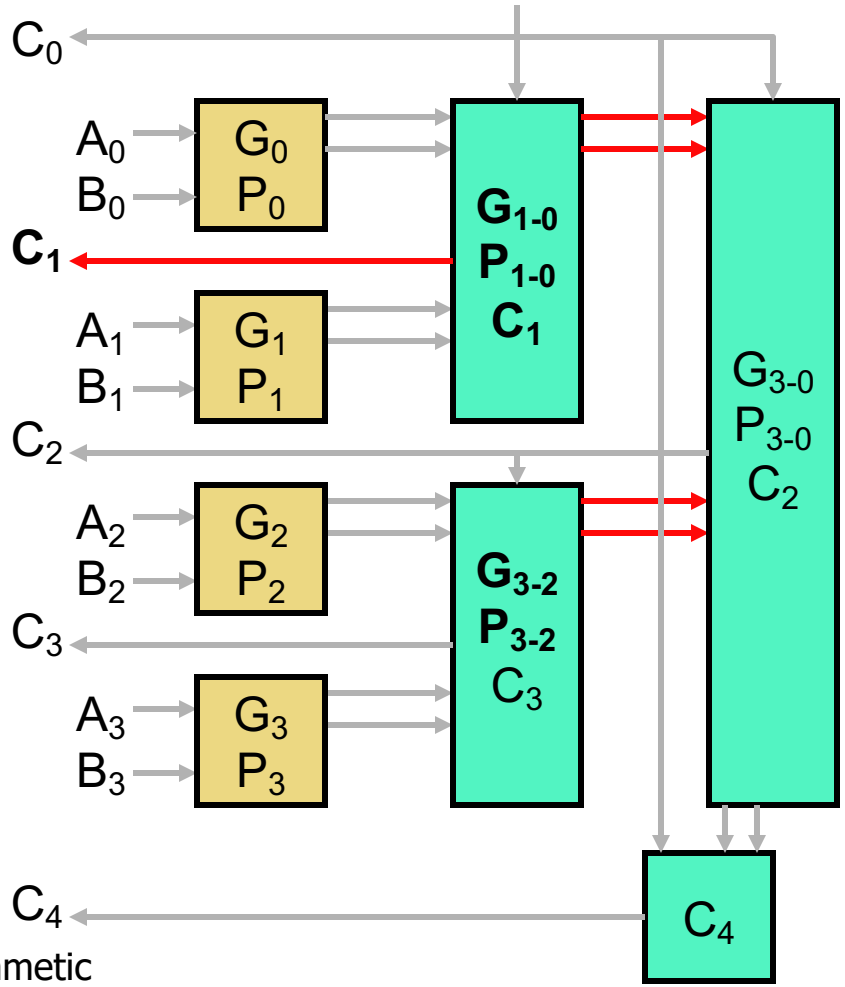
4-bit CLA Timing: d1

- Signals ready from before
  - d0: C_0, A_i, B_i
- New signals ready at d1
  - $P_0 = A_0 \mid B_0$, $G_0 = A_0B_0$
  - $P_1 = A_1 \mid B_1$, $G_1 = A_1B_1$
  - $P_2 = A_2 \mid B_2$, $G_2 = A_2B_2$
  - $P_3 = A_3 \mid B_3$, $G_3 = A_3B_3$



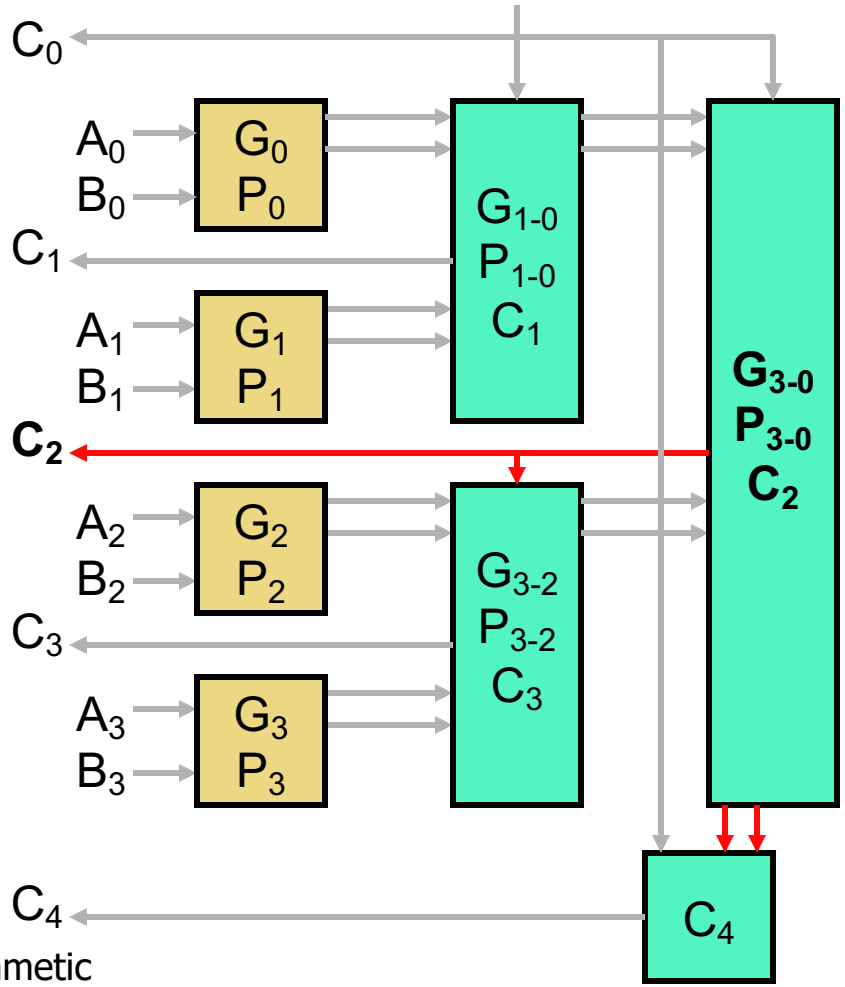
4-bit CLA Timing: d3

- Signals ready from before
  - d0: C_0, A_i, B_i
  - d1: P_i, G_i
- New signals ready at d3
  - $P_{1-0} = P_1P_0$
  - $G_{1-0} = G_1 \mid P_1G_0$
  - $C_1 = G_0 \mid P_0C_0$
  - $P_{3-2} = P_3P_2$
  - $G_{3-2} = G_3 \mid P_3G_2$
  - C_3 is not ready



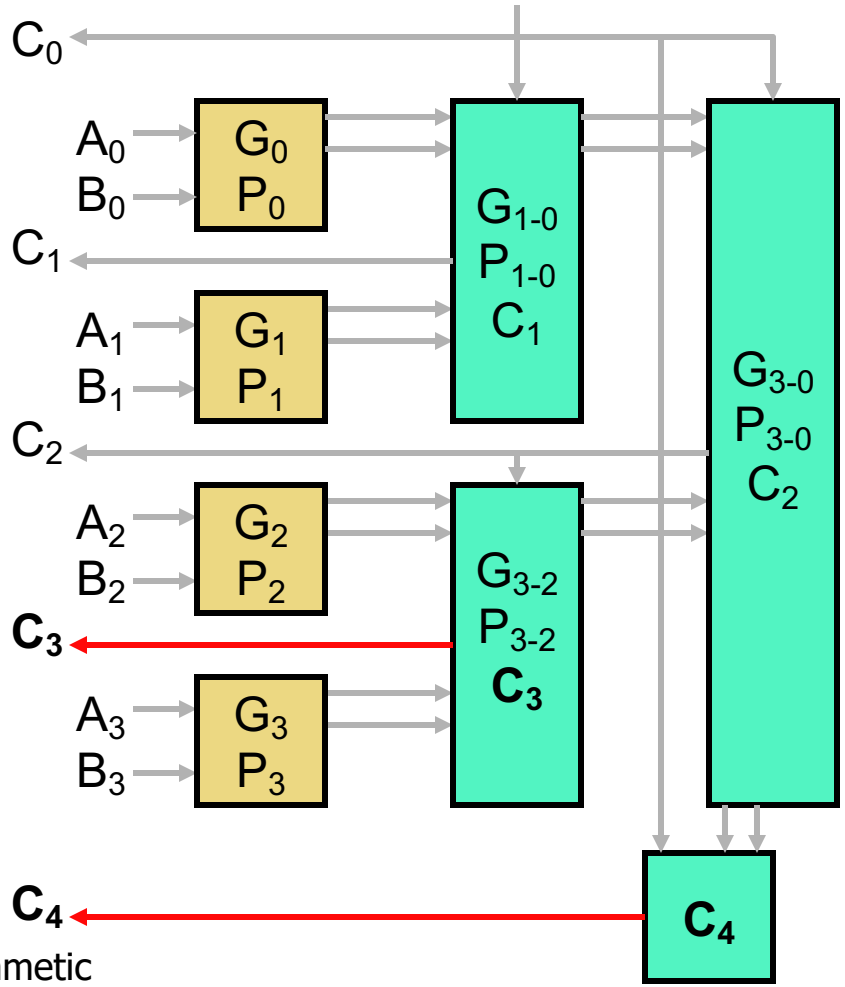
4-bit CLA Timing: d5

- Signals ready from before
  - d0: C_0, A_i, B_i
  - d1: P_i, G_i
  - d3: P_{1-0}, G_{1-0} , C_1 , P_{3-2} , G_{3-2}
- New signals ready at d5
  - $P_{3-0} = P_{3-2}P_{1-0}$
  - $G_{3-0} = G_{3-2} \mid P_{3-2}G_{1-0}$
  - $C_2 = G_{1-0} \mid P_{1-0}C_0$



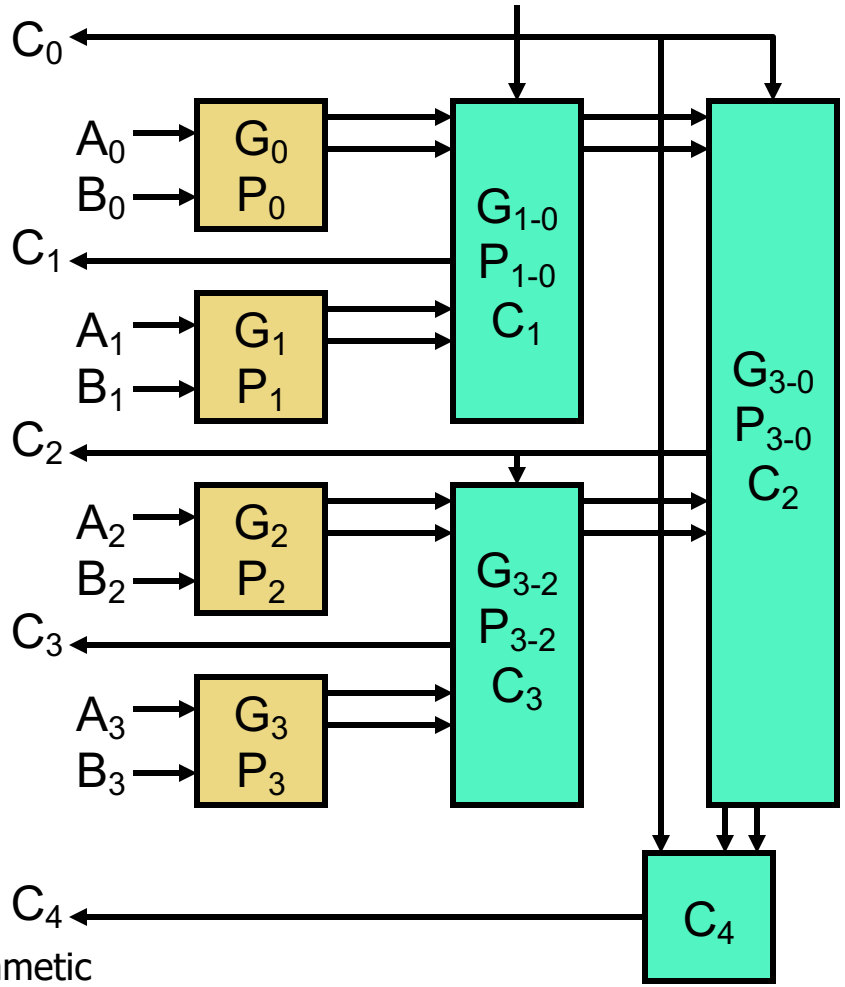
4-bit CLA Timing: d7

- Signals ready from before
  - d0: C_0, A_i, B_i
  - d1: P_i, G_i
  - d3: P_{1-0}, G_{1-0} , C_1 , P_{3-2} , G_{3-2}
  - d5: P_{3-0}, G_{3-0}, C_2
- New signals ready at d7
  - $C_3 = G_2 \mid P_2C_2$
  - $C_4 = G_{3-0} \mid P_{3-0}C_0$
- S_i ready d1 after C_i



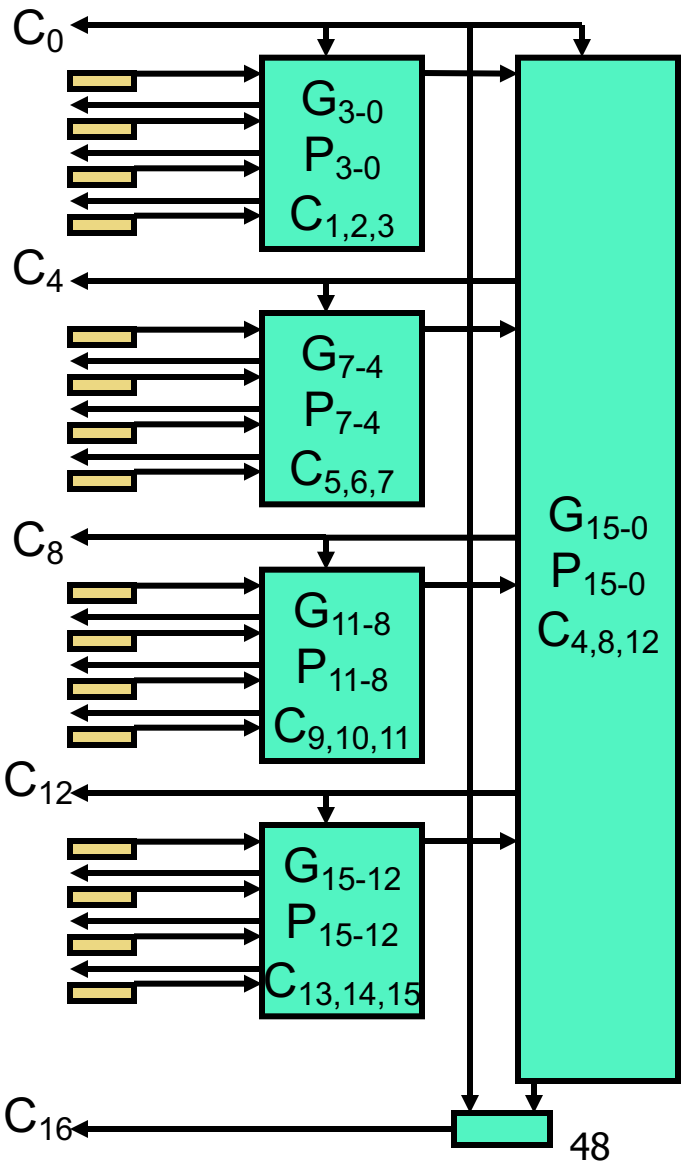
Two-Level CLA for 4-bit Adder

- Size?
  - CLA blocks: 5 gates, max 3 inputs (akin to infinite CLA with N=2)
    - 3 of these
  - C_4 block: 2 gates, max 2 inputs
  - Total: **17 gates, max 3 inputs**
    - Infinite: 14, 5
- Latency?
  - 2 for "top" CLA, 4 for "first-level"
    - G/P go "up", C go "down"
  - Total: **8** (7 for CLA, 1 for sum)
    - Infinite: 4
- 2L is bigger **and** slower ☹



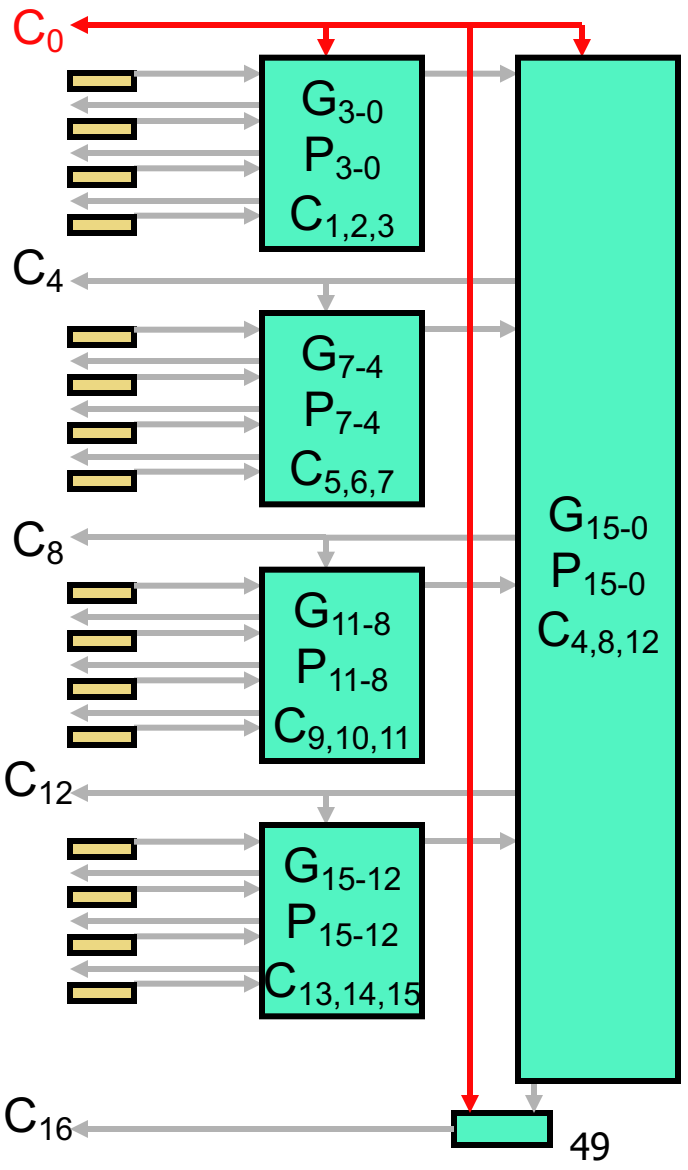
Two-Level CLA for 16-bit Adder

- 4 G/P inputs per level
- Hardware?
  - First level: 14 gates * 4 blocks
  - Second level: 14 gates * 1 block
  - C_{16} block: 2 gates
  - Total: **72 gates**
    - largest gate: **4 inputs**
    - Infinite: 152 gates, 17 inputs
- Latency?
  - Total: **8** (1 + 2 + 2 + 2 + 1)
  - Infinite: 4 (1 + 2 + 1)
- CLA for a 64-bit adder?



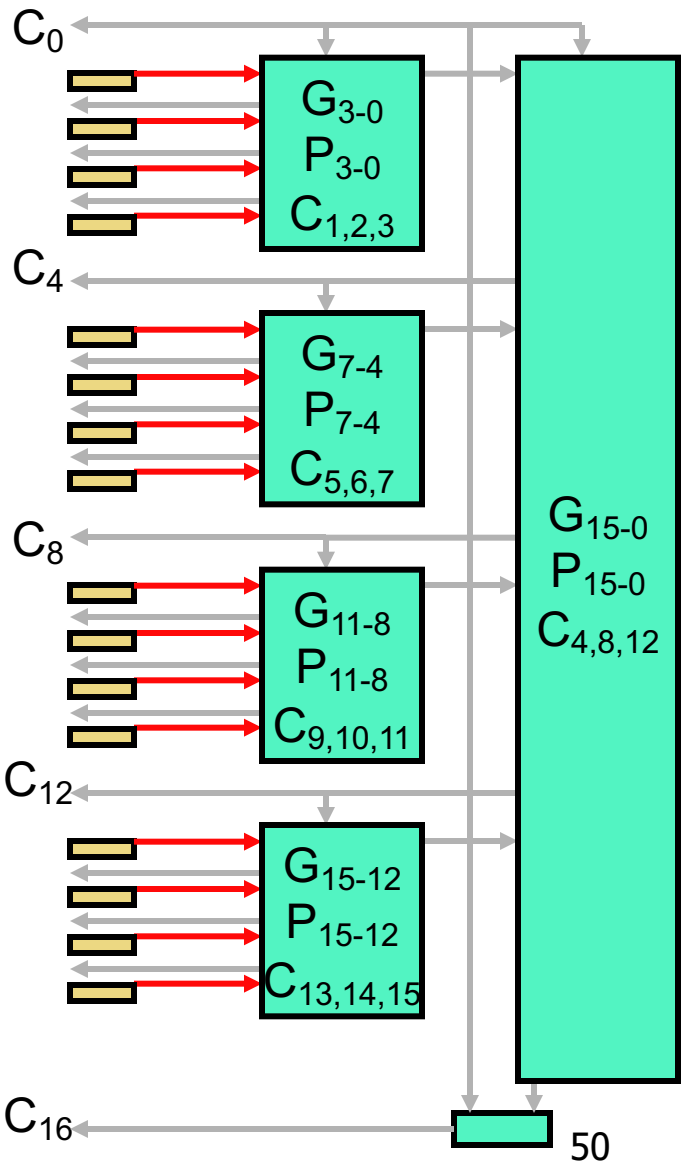
16-bit CLA Timing: d0

- What is ready after **0** gate delays?
  - C_0



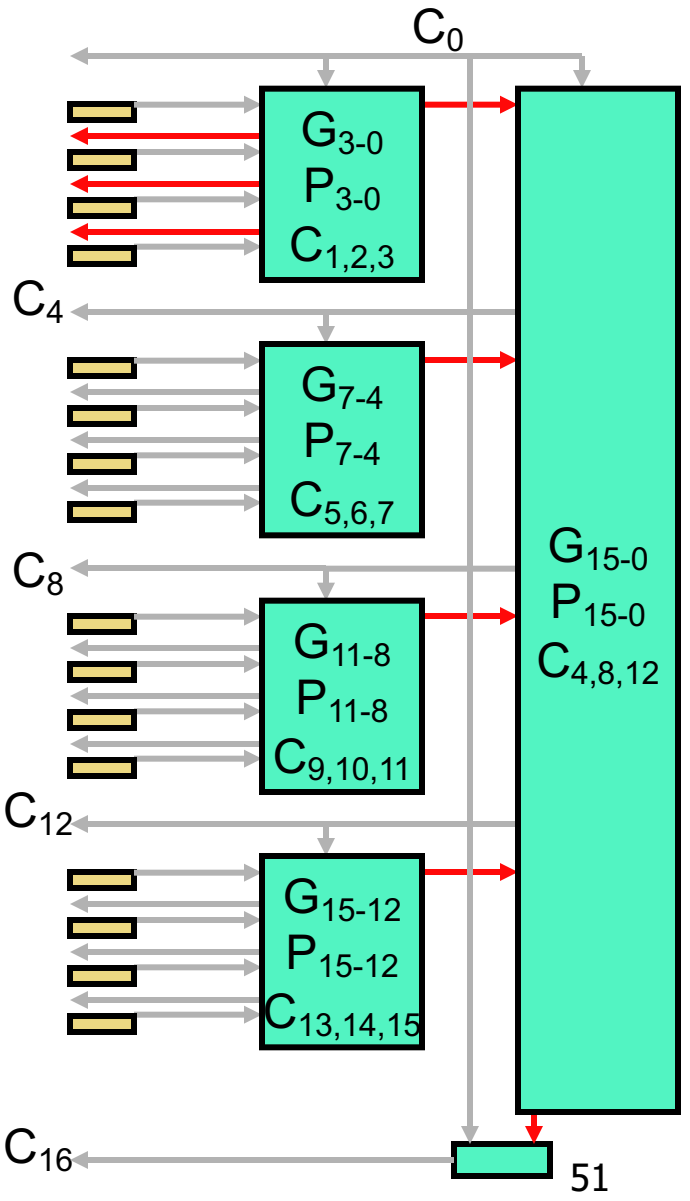
16-bit CLA Timing: d1

- What is ready after **1** gate delay?
  - Individual G/P



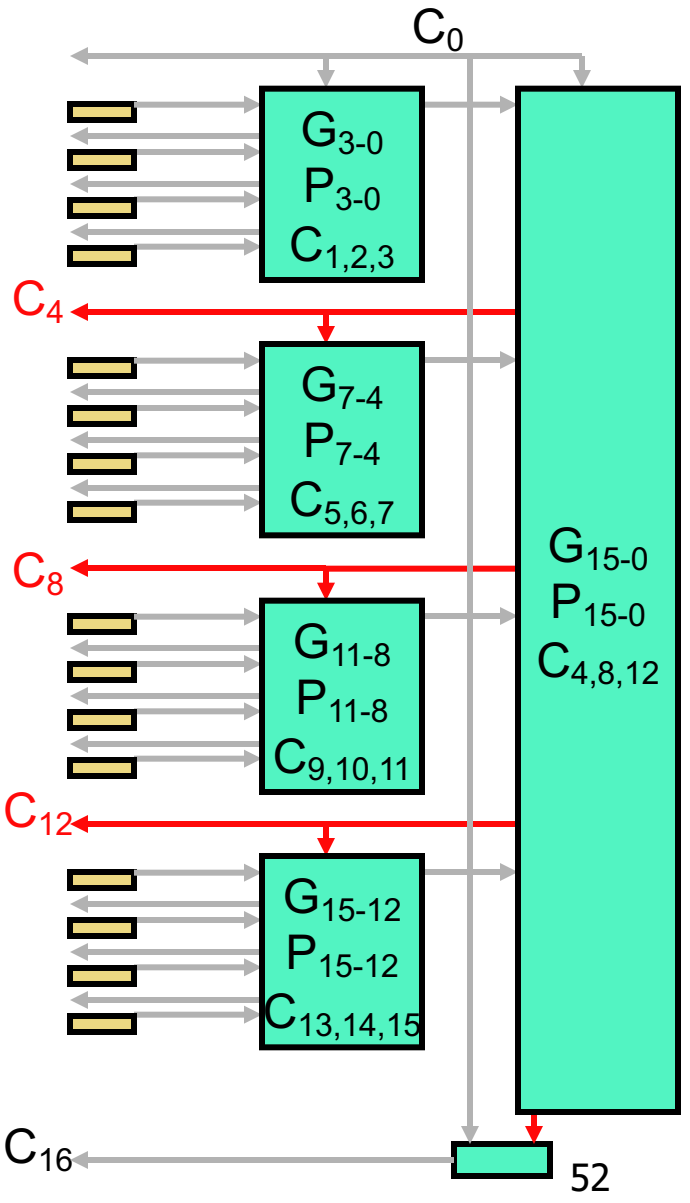
16-bit CLA Timing: d3

- What is ready after **3** gate delays?
  - 1st level group G/P
  - Interior carries of 1st group
    - C_1, C_2, C_3



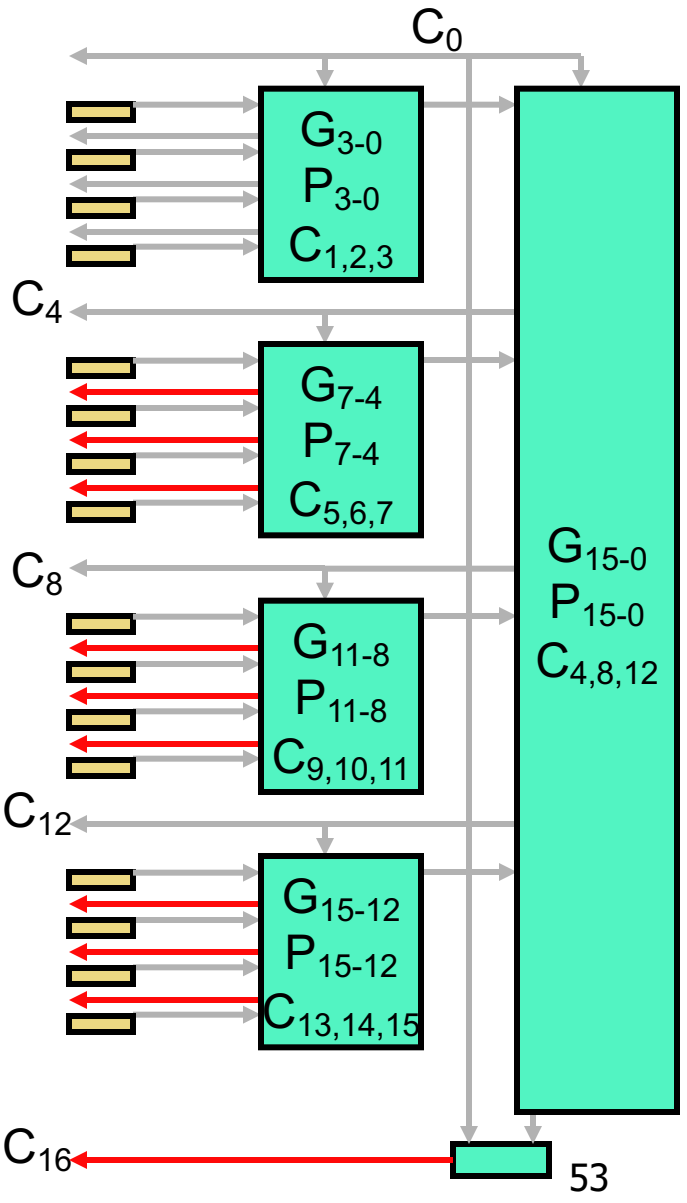
16-bit CLA Timing: d5

- And after **5** gate delays?
  - Outer level group G/P
  - Outer level "interior" carries
    - C_4, C_8, C_{12}



16-bit CLA Timing: d7

- And after **7** gate delays?
  - C_{16}
  - First level "interior" carries
    - 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 1 gate delay after C_i
  - All sum bits ready after 8 delays!
  - Same as 2-level 4-bit CLA
  - All 2-level CLAs have similar delay structure



Adders In Real Processors

- Real processors super-optimize their adders
  - Ten or so different versions of CLA
  - Highly optimized versions of carry-select
  - Other gate techniques: carry-skip, conditional-sum
  - Sub-gate (transistor) techniques: Manchester carry chain
  - Combinations of different techniques
    - Alpha 21264 used CLA+CSelA+RippleCA
    - Used at different levels

- Even more optimizations for incrementers
  - Why?

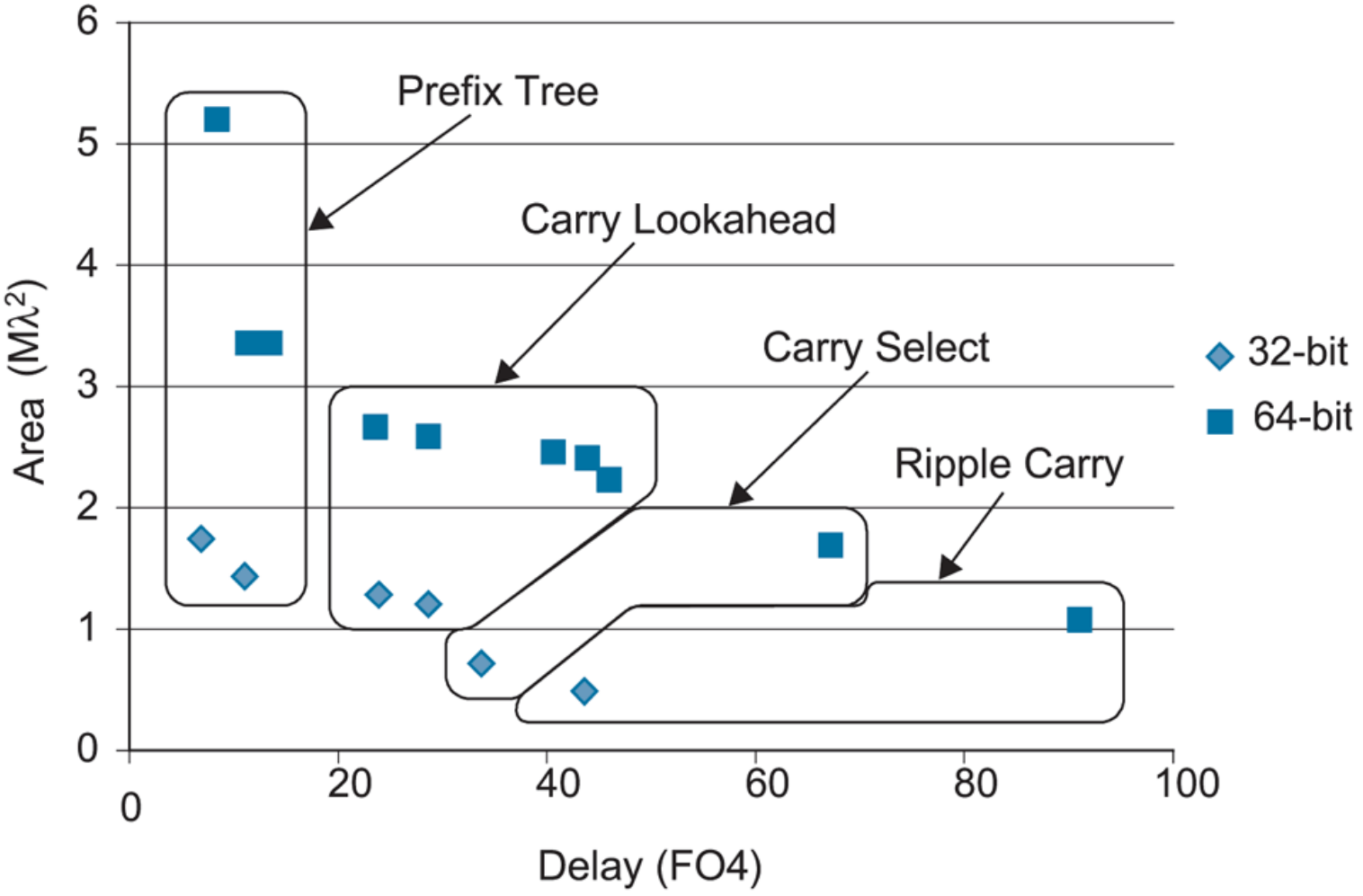


FIG 10.47 Area vs. delay of synthesized adders

Shifts & Rotates

Shift and Rotation Instructions

- Left/right shifts are useful…
  - Fast multiplication/division by small constants (next)
  - Bit manipulation: extracting and setting individual bits in words

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

    `srl 110011, 2 = 001100`

    `sra 110011, 2 = 111100`
  - Caveat: for negative numbers, `sra` is **not** equal to division by 2
    - Consider: -1 / 16 = ?

- Rotations are less useful…
  - But almost "free" if shifter is there
  - MIPS and LC4 have only shifts, x86 has shifts and rotations

Compiler Opt: Strength Reduction

- **Strength reduction**: compilers will do this (sort of)

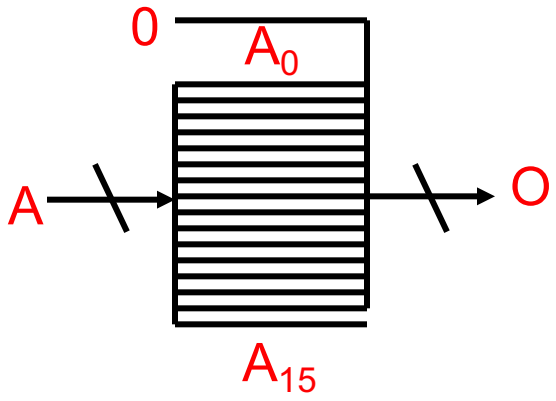
```
A * 4 = A << 2
A * 5 = (A << 2) + A
A / 8 = A >> 3          (only if A is unsigned)
```

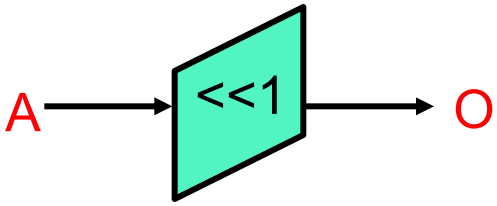
  - Useful for address calculation: all basic data types are 2^M in size

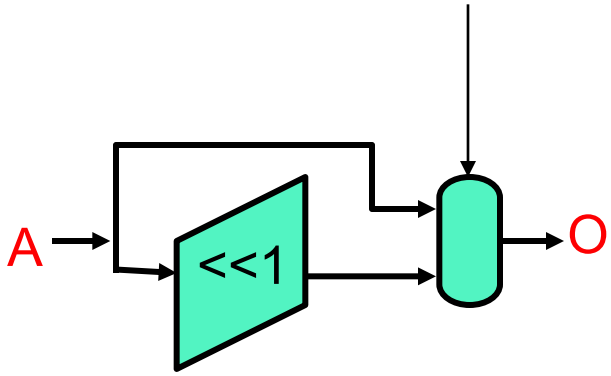
```
int A[100];
&A[N] = A+(N*sizeof(int)) = A+N*4 = A+N<<2
```

A Simple Shifter

- The simplest 16-bit shifter: can only shift left by 1
  - **Implement using wires (no logic!)**
- Slightly more complicated: can shift left by 1 or 0
  - Implement using wires and a multiplexor (mux16_2to1)

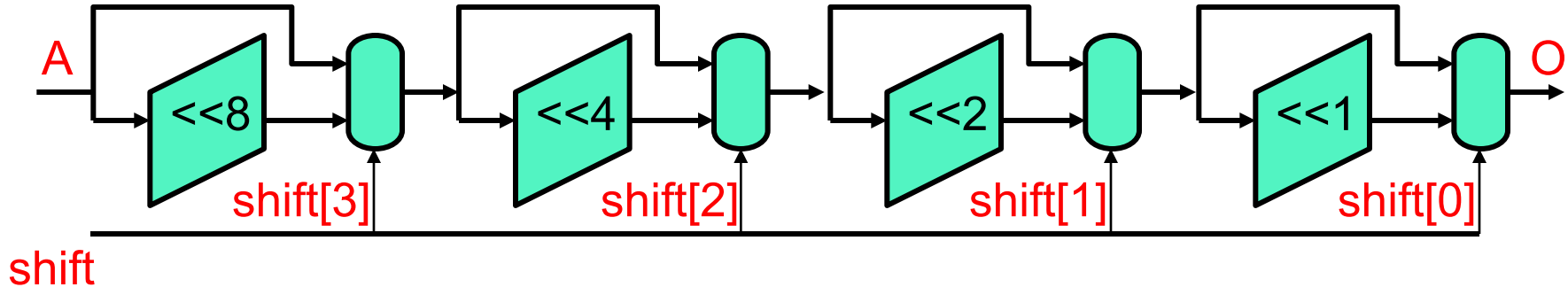






Barrel Shifter

- What about shifting left by any amount 0–15?
- 16 consecutive "left-shift-by-1-or-0" blocks?
  - Would take too long (how long?)
- **Barrel shifter**: 4 "shift-left-by-X-or-0" blocks (X = 1,2,4,8)
  - What is the delay?



- Similar barrel designs for right shifts and rotations

Shifter in Verilog

- Logical shift operators `<<` `>>`
  - performs zero-extension for >>

```
wire [15:0] a = b << c[3:0];
```

- Arithmetic shift operator `>>>`
  - performs sign-extension
  - requires a signed wire input

```
wire signed [15:0] b;
wire [15:0] a = b >>> c[3:0];
```

Multiplication

3rd Grade: Decimal Multiplication

```
    19     // multiplicand
*   12     // multiplier
    38
+ 190
   228     // product
```

- Start with product 0, repeat steps until no multiplier digits
  - Multiply multiplicand by least significant multiplier digit
  - Add to product
  - 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 may have N+M digits

Binary Multiplication: Same Refrain

```
    19 =          010011    // multiplicand
*   12 =          001100    // multiplier
-------------------------
     0 =  000000000000
     0 =  000000000000
    76 =  000001001100
   152 =  000010011000
     0 =  000000000000
+    0 =  000000000000
-------------------------
   228 =  000011100100    // product
```

± Smaller base → more steps, each is simpler

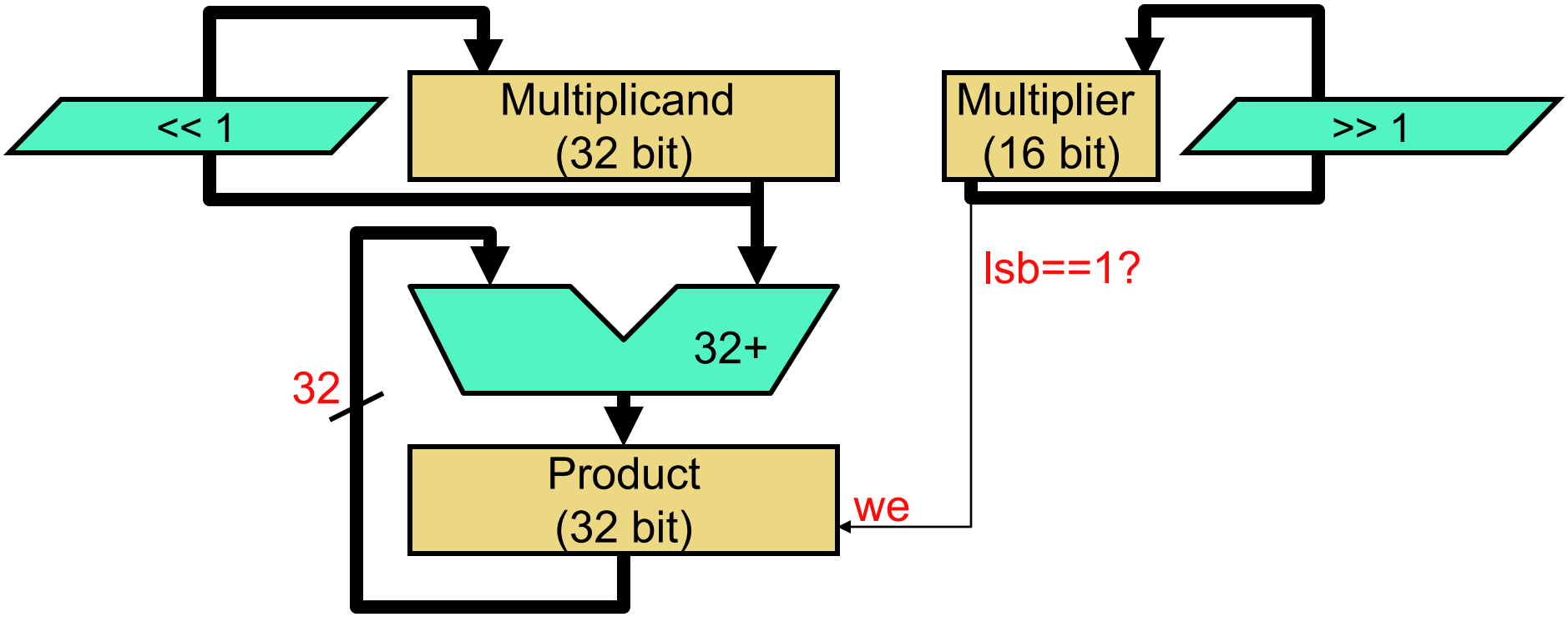
- Multiply multiplicand by **least significant multiplier digit**
  + 0 or 1 → no actual multiplication, add multiplicand or not
- Add to total: we know how to do that
- Shift multiplicand left, multiplier right by one digit

Software Multiplication

- Can implement this algorithm in software
- Inputs: `md` (multiplicand) and `mr` (multiplier)

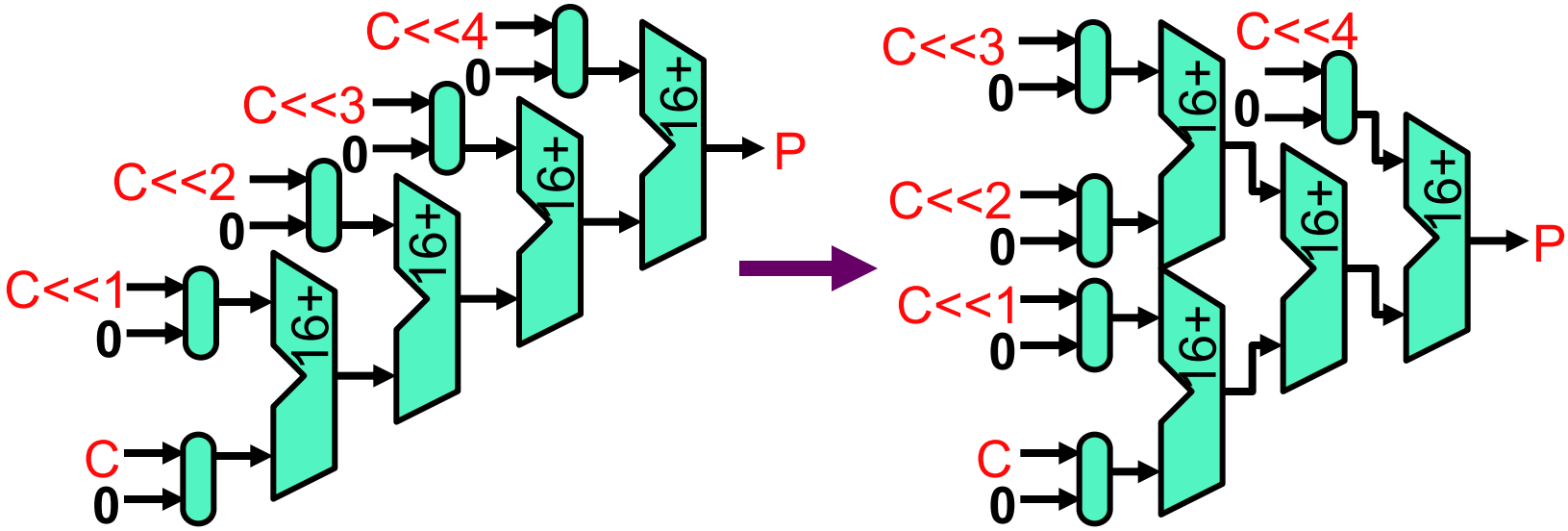
```
int pd = 0;  // product
int i = 0;
for (i = 0; i < 16 && mr != 0; i++) {
   if (mr & 1) {
     pd = pd + md;
   }
   md = md << 1;    // shift left
   mr = mr >> 1;    // shift right
}
```

Hardware Multiply: Sequential



- **Control**: repeat 16 times
  - If least significant bit of multiplier is 1…
    - Then add multiplicand to product
  - Shift multiplicand left by 1
  - Shift multiplier right by 1

Hardware Multiply: Combinational



- Multiply by N bits at a time using N adders
  - Example: N=5, terms (P=product, C=multiplicand, M=multiplier)
  - P = (M[0] ? (C) : 0) + (M[1] ? (C<<1) : 0) +
    (M[2] ? (C<<2) : 0) + (M[3] ? (C<<3) : 0) + …
  - Arrange like a tree to reduce gate delay critical path
- Delay? N^2 vs N*log N? Not that simple, depends on adder
  - Approx "2N" versus "N + log N", with optimization: O(log N)

Partial Sums/Carries

- Observe: carry-outs don't have to be chained immediately
  - Can be saved for later and added back in

```
  00111 = 7
+ 00011 = 3
  00100     // partial sums (sums without carrries)
+ 00110     // partial carries (carries without sums)
  01010 = 10
```

  - Partial sums/carries use simple half-adders, not full-adders
  + Aren't "chained" → can be done in two levels of logic
  – Must sum partial sums/carries eventually, and this sum is chained
    - d(CS-adder) = 2 + d(normal-adder)
  - What is the point?

Three Input Addition

- Observe: only 0/1 carry-out possible even if 3 bits added

```
 00111 = 7
 00011 = 3
+00010 = 2
 00110      // partial sums (sums without carrries)
+00110      // partial carries (carries without sums)
 01100 = 12
```

  - Partial sums/carries use full adders

+ Still aren't "chained" → can be done in two levels of logic

- The point is delay(CS-adder) = 2 + delay(normal-adder)…
- …even for adding 3 numbers!
- 2 + delay(normal-adder) < 2 * delay(normal-adder)

Hardware != Software: Part Deux

- Recall: hardware is parallel, software is sequential
- Exploit: evaluate independent sub-expressions in parallel

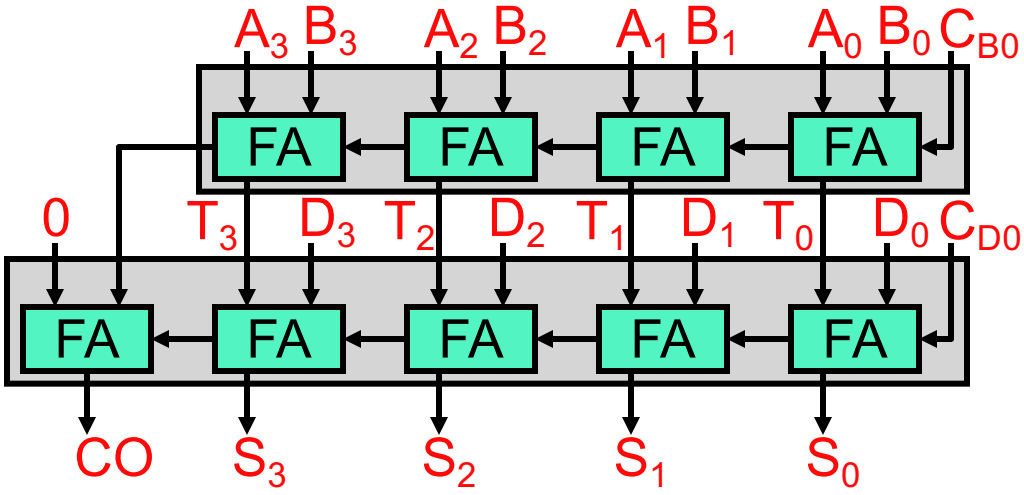
- Example I: S = A + B + C + D
  - Software? 3 steps: (1) S1 = A+B, (2) S2 = S1+C, (3) S = S2+D
  + Hardware? 2 steps: (1) S1 = A+B, S2=C+D, (2) S = S1+S2

- Example II: S = A + B + C
  - Software? 2 steps: (1) S1 = A+B, (2) S = S1+C
  - Hardware? 2 steps: (1) S1 = A+B (2) S = S1+C
  + Actually hardware can do this in 1.2 steps! (CSA adder)

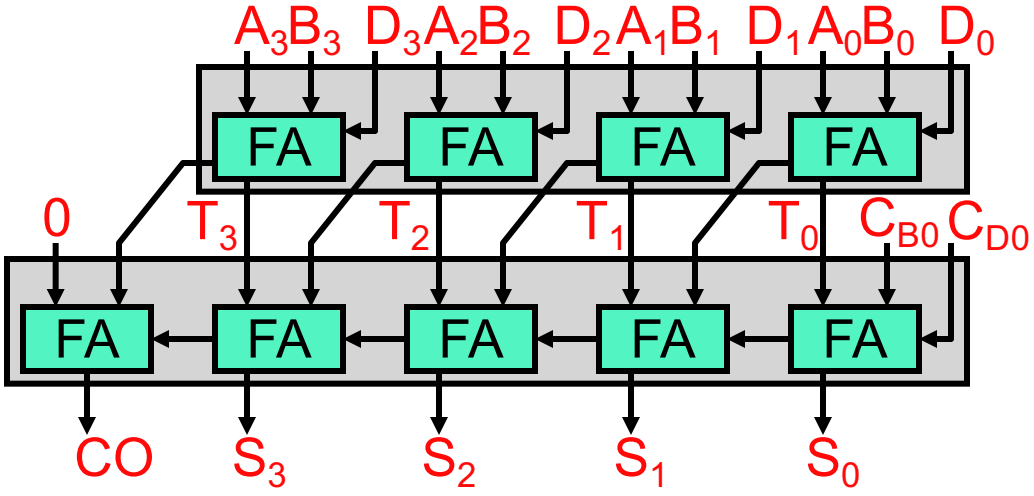
Carry Save Addition (CSA)

- **Carry save addition (CSA):** delay(N adds) < N*d(1 add)
  - Enabling observation: unconventional view of full adder
    - 3 inputs (A,B,C_{in}) $\rightarrow$ 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
    - (A,B,C_{inT}) $\rightarrow$ (T,C_{outT}), chain C_{outT} to C_{inT+1}
    - (T,D,C_{inS}) $\rightarrow$ (S,C_{outS}), chain C_{outS} to C_{inS+1}
  - Notice: we have **three independent inputs** to feed first adder
    - (A,B,D) $\rightarrow$ (T,C_{outT}), **no chaining (CSA: 2 gate levels)**
      - T: A+B+D (the **partial sum**)
      - C_{outT}: A+BD (the **partial carry**)
    - (T,C_{outT},C_{inS}) $\rightarrow$ (S,C_{outS}), chain C_{outS} to C_{inS+1}

Carry Save Addition (CSA)

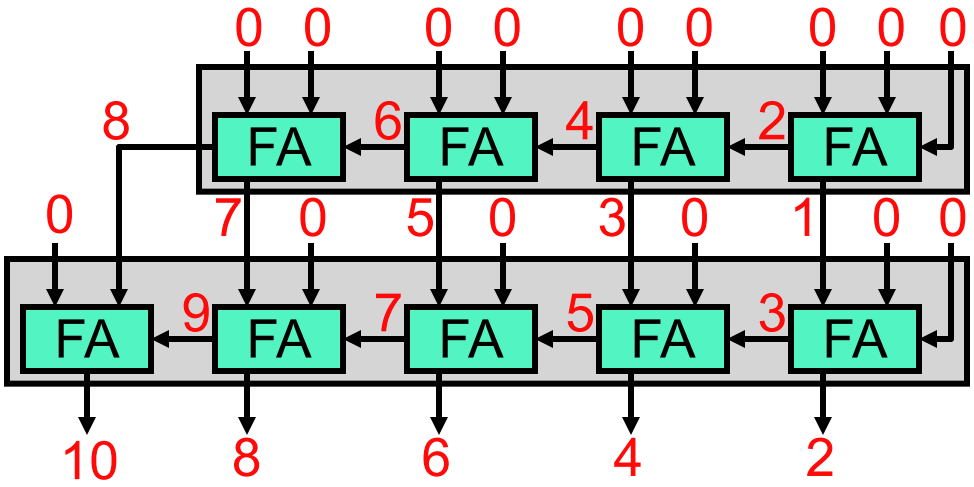


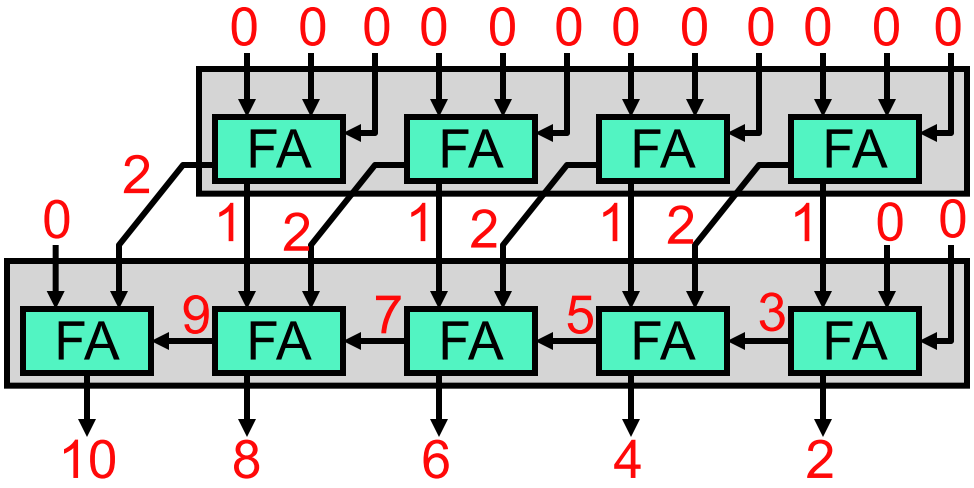
- 2 RC adders



- CSA+RC adder
  - Subtraction works too

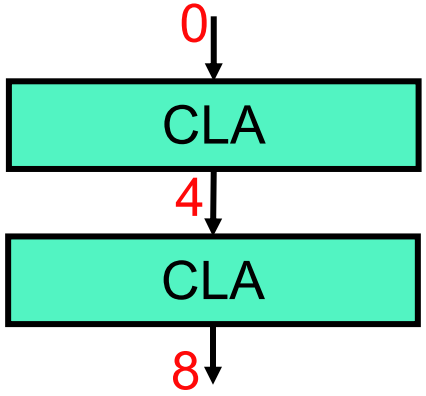
Carry Save Addition Delay



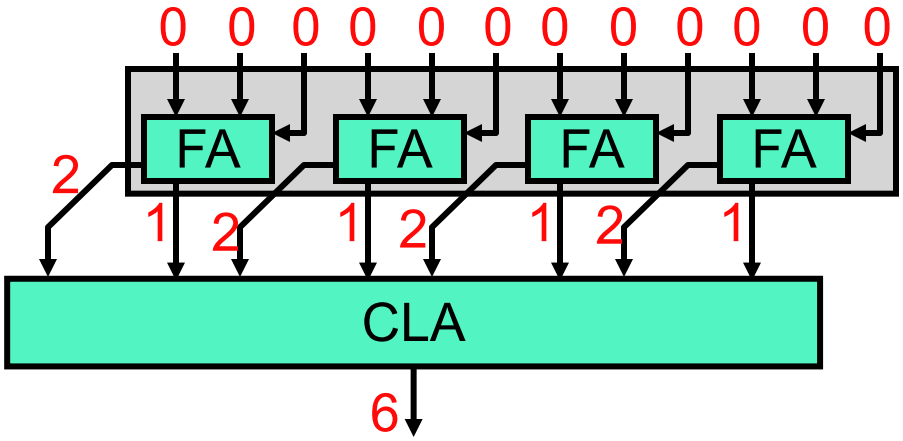


- CSA isn't faster :-(
  - why not?

Let's use CLA instead of RC



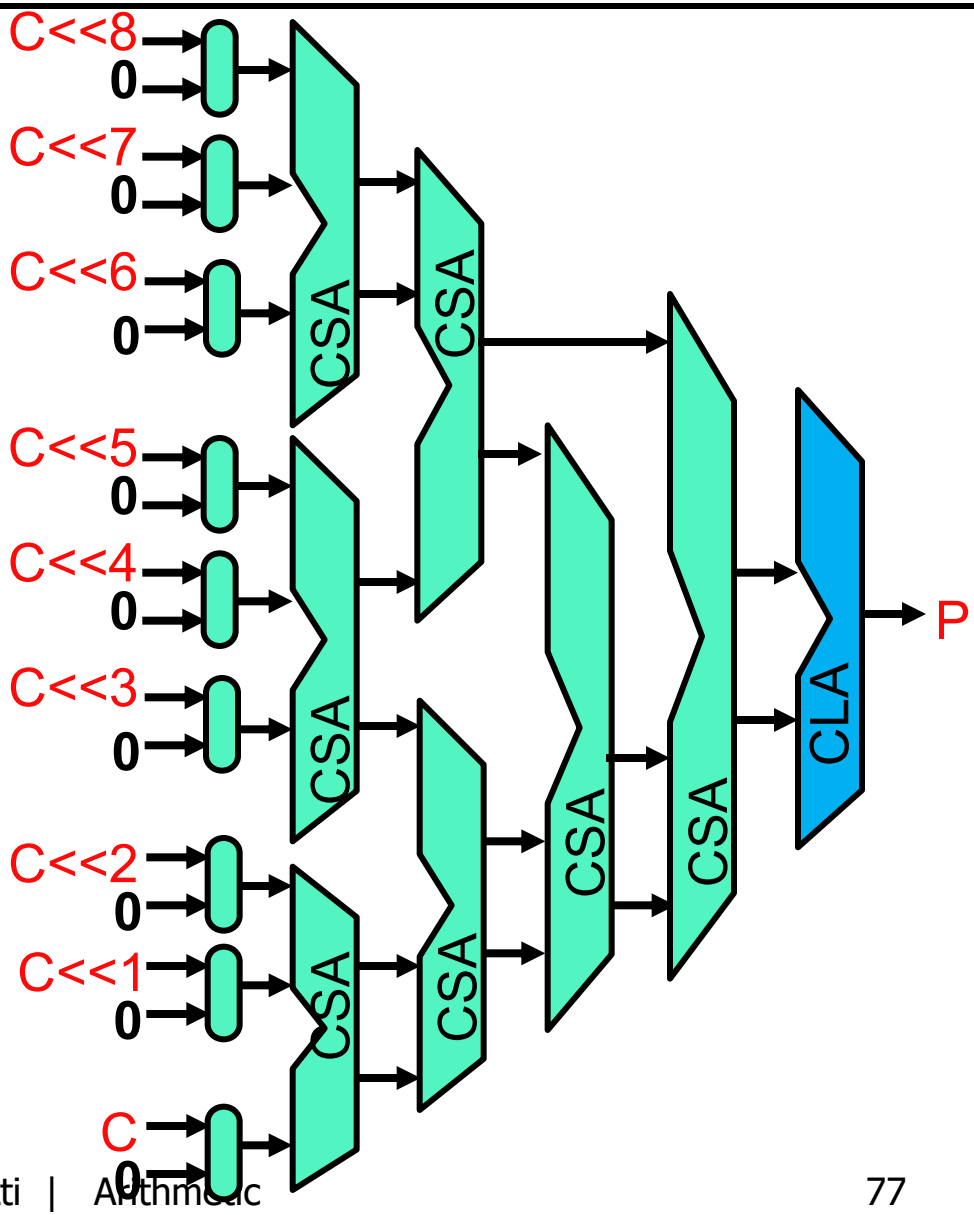
- 4-bit CLA
  - all outputs available at the same time, after 4 gate delays
  - see slide 45. Two-Level CLA for 4-bit Adder



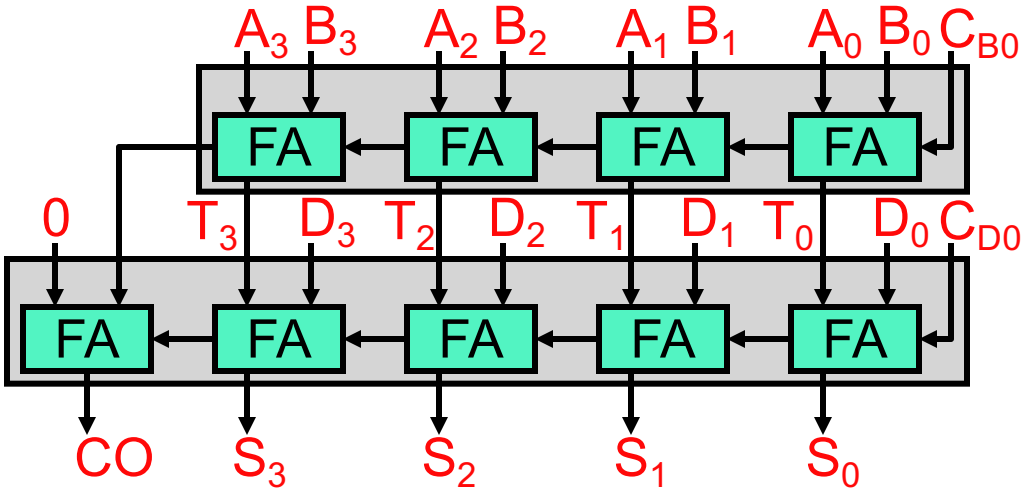
- Now CSA helps
  - even large gains with more/larger numbers to add

CSA Tree Multiplier

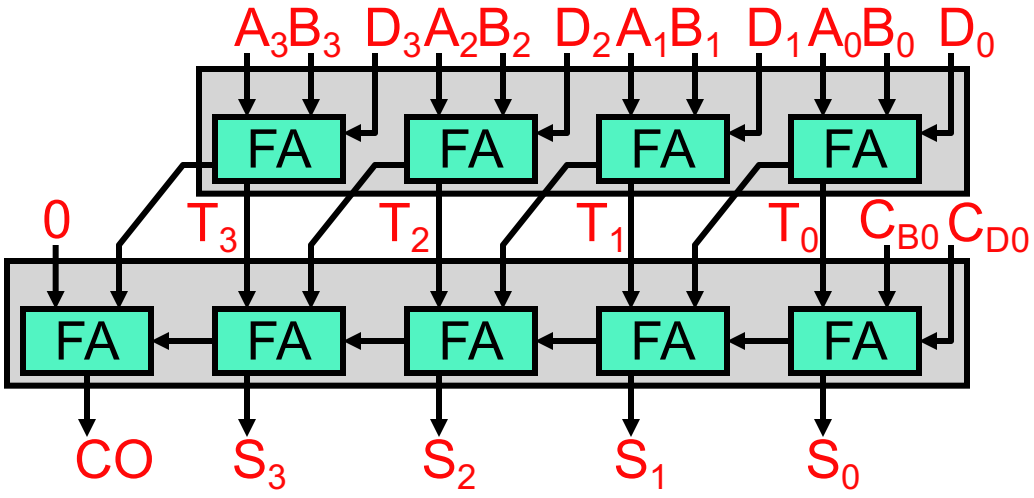
- Use 3-to-2 CSA adders
  - Build a tree structure
  - called a "Wallace Tree"
- 16-bit
  - Start: 16 bits
  - 1st: 5*(3->2)+1 = 11
  - 2nd: 3*(3->2)+2 = 8
  - 3rd: 2*(3->2)+2 = 6
  - 4th: 2*(3->2)+0 = 4
  - 5th: 1*(3->2)+1 = 3
  - 6th: 1*(3->2)+0 = 2
  - 7th: CLA
  - delay: 2+(6*2)+8 = 22



Consecutive Addition: Carry Save Adder



- 2 N-bit RC adders
  + 2 + d(add) gate delays
- M N-bit RC adders delay
  - Naïve: O(M*N)
  - Actual: O(M+N)



- M N-bit Carry Select?
  - Delay calculation tricky
- Carry Save Adder (CSA)
  - 3-to-2 CSA tree + adder
  - Delay: O(log M + log N)

Floating Point

Floating Point (FP) Numbers

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

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

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

- Aside: best not used for currency values

Scientific Notation

- **Scientific notation**:
  - 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: all representations are just approximations
    - Sometimes wierdos 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
  - Used when exact precision is less important (e.g., 3D games)

- **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
  - Used for scientific computations

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

Floating Point is Inexact

- Accuracy problems sometimes get bad
  - FP arithmetic not associative: (A+B)+C not same as A+(B+C)
  - Addition of big and small numbers (summing many small numbers)
  - Subtraction of two big numbers
- Example, what's $(1*10^{30} + 1*10^{0}) - 1*10^{30}$?
  - Intuitively: $1*10^{0} = 1$
  - But: $(1*10^{30} + 1*10^{0}) - 1*10^{30} = (1*10^{30} - 1*10^{30}) = 0$
- Reciprocal math: "x/y" versus "x*(1/y)"
  - Reciprocal & multiply is faster than divide, but less precise
- Compilers are generally conservative by default
  - GCC flag: –ffast-math (allows assoc. opts, reciprocal math)
- **Numerical analysis**: field formed around this problem
  - Re-formulating algorithms in a way that bounds numerical error
- In your code: never test for equality between FP numbers
  - Use something like: if (abs(a-b) < 0.00001) then …

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

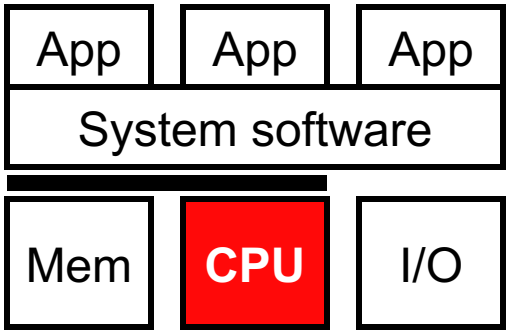
Arithmetic Latencies

- Latency in cycles of common arithmetic operations
- Source: *Agner Fog,* *https://www.agner.org/optimize/#manuals*
  - AMD Ryzen core

	Int 32	Int 64	Fp 32	Fp 64
Add/Subtract	1	1	5	5
Multiply	3	3	5	5
Divide	14-30	14-46	8-15	8-15

- Divide is variable latency based on the size of the dividend
  - Detect number of leading zeros, then divide
- Why is FP divide faster than integer divide?

Summary



- Integer addition
  - Most timing-critical operation in datapath
  - Hardware != software
    - Exploit sub-addition parallelism
- Fast addition
  - Carry-select: parallelism in sum
- Multiplication
  - Chains and trees of additions
- Division
- Floating point

- Next: single-cycle datapath