

ECE 152 Introduction to Computer Architecture

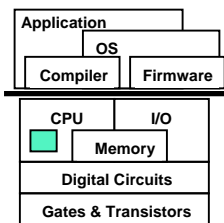
Arithmetic and ALU Design
Copyright 2008 Daniel J. Sorin
Duke University

Slides are derived from work by
Amir Roth (Penn) and Alvy Lebeck (Duke)
Spring 2008

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
 - 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
 - $10_{10} = 000001010$, $-10_{10} = 100001010$
 - 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: More On TC

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

```
0 = 100000000
-10 = 00001010
-----
-10 = 011110110 (ignore imaginary leading 0)
```

- Now, add the conventional way...

```
-10 = 11110110
+10 = 00001010
-----
0 = 100000000 (ignore imaginary leading 1)
```

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?

$$\begin{array}{r} 1 \\ 43 \\ +29 \\ \hline 72 \end{array}$$

- 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

$$\begin{array}{r} 1 \quad 111111 \\ 43 = 00101011 \\ +29 = 00011101 \\ \hline 72 = 01001000 \end{array}$$

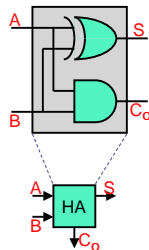
- Repeat N times
 - Add least significant bits and any overflow from previous add
 - Carry the overflow to next addition
 - Shift two addends and sum one bit to the right
 - Sum of two N-bit numbers can yield an N+1 bit number
- More steps (smaller base)
+ Each one is simpler (adding just 1 and 0)
- 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

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

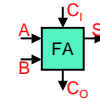


Review: The Full Adder

- 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

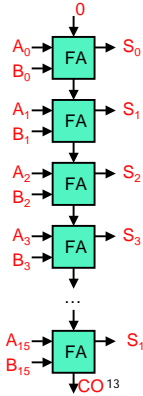
C _i	A	B	C _o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- $S = C_i'A'B + C_i'AB' + C_iA'B' + C_iAB = C_i \oplus A \oplus B$
- $C_o = C_i'AB + C_iA'B + C_iAB' + C_iAB = C_iA + C_iB + AB$
- This is called a **full adder** → what is its delay (in #gates)?



A 16-bit Adder

- Simple 16-bit adder
 - 16 1-bit full adders "chained" together
 - $CO_0 = CI_1, CO_1 = CI_2, \dots$
 - $CI_0 = 0, CO_{15}$ 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})
 - $d(CO_{15}) = 2 + \text{MAX}\{d(A_{15}), d(B_{15}), d(CI_{15})\}$
 - $d(A_{15}) = d(B_{15}) = 0, d(CI_{15}) = d(CO_{14})$
 - $d(CO_{15}) = 2 + d(CO_{14}) = 2 + 2 + d(CO_{13}) \dots$
 - $d(CO_{15}) = 32$**
 - $- 2N = \text{slow!}$**

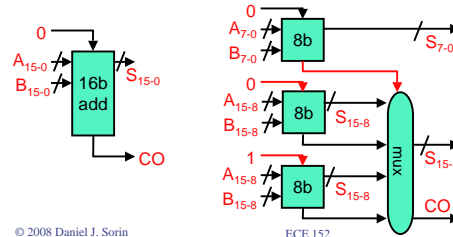


© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

A Faster 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



© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

14

How Fast Is the Faster Adder?

- $d(CO_{15}) = \text{max}\{d(CO_{15:8}), d(CO_{7:0})\} + 2$ (+2 is for mux)
- $d(CO_{15}) = \text{max}\{2*8, 2*8\} + 2 = 18$ (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!
 - $d(CO_{15}) = \text{max}\{d(CO_{15:12}), d(CO_{11:0})\} + 2$
 - $d(CO_{15}) = \text{max}\{2*4, \text{max}\{d(CO_{11:8}), d(CO_{7:0})\} + 2\} + 2$
 - $d(CO_{15}) = \text{max}\{2*4, \text{max}\{2*4, \text{max}\{d(CO_{7:4}), d(CO_{3:0})\} + 2\} + 2\} + 2$
 - $d(CO_{15}) = \text{max}\{2*4, \text{max}\{2*4, \text{max}\{2*4, 2*4\} + 2\} + 2\} + 2$
 - $d(CO_{15}) = 2*4 + 3*2 = 14$
- In general, N-bit adder in M pieces: **$2*(N/M) + (M-1)*2$**
 - 16-bit adder in 8 parts: $2*(16/8) + 7*2 = 18 > 14$???!

© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

15

The Bigger Point

- Hardware vs. Software
 - "Hardware is parallel, software is sequential"
 - What does this mean practically?
- Hardware can do things that software fundamentally can't
 - For instance: **speculation**
 - Computation is waiting for some slow input?
 - And input can only be one of two options?
 - Compute with both (slow), choose right one later (fast)**
 - Does this even make sense in software?
- In hardware, it's easier to trade resources for latency

© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

16

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}**
- $C_{16} = g_{15} + p_{15}C_{15}$
- $C_{16} = g_{15} + p_{15}(g_{14} + p_{14}C_{14})$
- $C_{16} = g_{15} + p_{15}g_{14} + p_{15}p_{14}(g_{13} + p_{13}C_{13})$
- $C_{16} = g_{15} + p_{15}g_{14} + \dots + p_{15}p_{14}\dots p_2p_1g_0 + p_{15}p_{14}\dots p_2p_1p_0C_0$
 - 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 $d(S_{15}) = ?$
 - $d(p_{14}, g_{14}) + d(c_{15} \text{ given } p_{14}, g_{14}) + d(S_{15} \text{ given } c_{15}) = 1 + 2 + 2 = 5 !!$
 - 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 $C_N \dots C_1$ 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

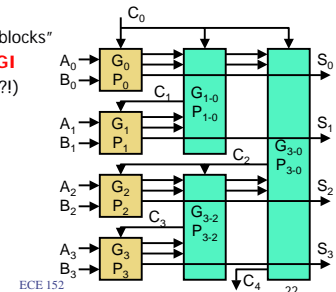
- Let's look at what we have so far (the two extremes)
 - **Ripple carry**
 - + Few small gates: no additional gates
 - Laid in series: 2N latency
 - **Infinite CLA**
 - Many big gates: $N*(N+3)/2$ additional gates, max N+1 inputs
 - + Laid in parallel: constant latency of 5 gate delays
 - We'd like something in between
 - Reasonable number of small gates
 - Sublinear (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
 - $C_1 = g_0 + p_0 C_0$, $C_2 = g_1 + p_1 C_1$, $C_3 = g_2 + p_2 C_2$, $C_4 = g_3 + p_3 C_3$
- Fully expanded (infinite hardware) CLA equations
 - $C_1 = g_0 + p_0 C_0$
 - $C_2 = g_1 + p_1 g_0 + p_1 p_0 C_0$
 - $C_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$
 - $C_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$
- Hierarchical CLA equations
 - First level:** expand C_2 using C_1 and C_4 using C_3
 - $C_2 = g_1 + p_1(g_0 + p_0 C_0) = (g_1 + p_1 g_0) + (p_1 p_0) C_0 = G_{1,0} + P_{1,0} C_0$
 - $C_4 = g_3 + p_3(g_2 + p_2 C_2) = (g_3 + p_3 g_2) + (p_3 p_2) C_2 = G_{3,2} + P_{3,2} C_2$
 - Second level:** expand C_4 using expanded C_2
 - $C_4 = G_{3,2} + P_{3,2}(G_{1,0} + P_{1,0} C_0) = (G_{3,2} + P_{3,2} G_{1,0}) + (P_{3,2} P_{1,0}) C_0$
 - $C_4 = G_{3,0} + P_{3,0} C_0$

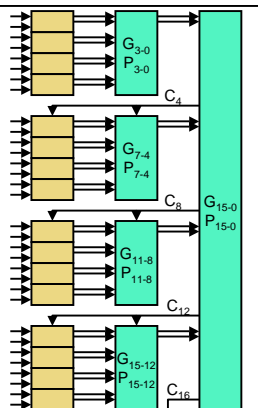
Two-Level (2L) CLA for 4-bit Adder

- 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
- 2L: bigger and slower??!



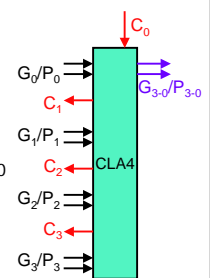
Two-Level CLA for 16-bit Adder

- 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?



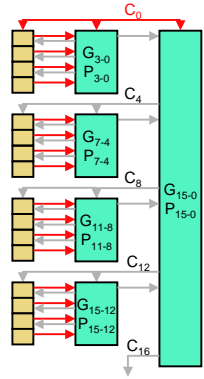
A Closer Look at CLA Delay

- 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



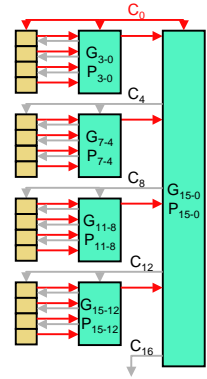
CLA Tree Signal Timing: d1

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



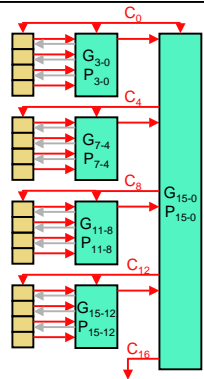
CLA Tree Signal Timing: d3

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



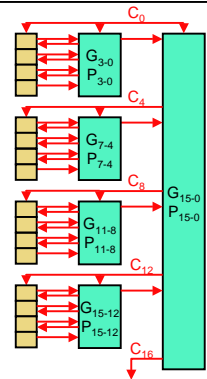
CLA Tree Signal Timing: d5

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



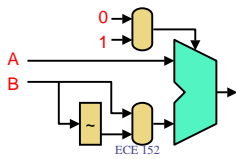
CLA Tree Signal Timing: d7

- 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!



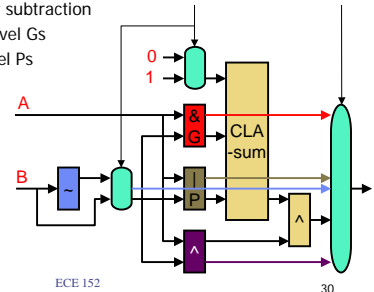
Subtraction: Addition's Tricky Pal

- 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?

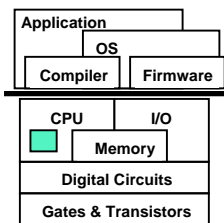


A 16-bit ALU

- Build an ALU with functions: **add/sub**, **and**, **or**, **not**, **xor**
 - All of these already in CLA adder/subtractor
 - **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**?
 - $S_i = A_i \wedge B_i \wedge C_i$



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 \ll (left shift) and \gg (right shift) in C/C++/Java
- **ICQ: Left shift example: $001010 \ll 2 = ?$**
- **ICQ: Right shift example: $001010 \gg 2 = ?$**
- Shifts are useful for
 - Bit manipulation: extracting and setting individual bits in words
 - Multiplication and division by powers of 2
 - $A * 4 = A \ll 2$
 - $A / 8 = A \gg 3$
 - $A * 5 = (A \ll 2) + A$
 - 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

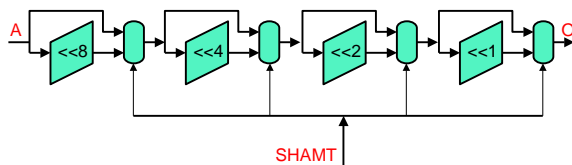
A Simple (Left) Shifter

- The simplest 16-bit shifter: can only shift left by 1
 - Implement using wires
- 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 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)
 - `sr1 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

