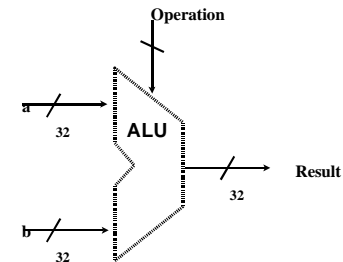


Computer Architecture – Set Four

Arithmetic

- Where we've been:
 - Performance (seconds, cycles, instructions)
 - Abstractions:
 - Instruction Set Architecture
 - Assembly Language and Machine Language
- What's up ahead:
 - Implementing the Architecture



Numbers

- Bits are just bits (no inherent meaning)
 - conventions define relationship between bits and numbers
- Binary numbers (base 2)
 - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
 - decimal: $0 \dots 2^n - 1$
- Of course, it gets more complicated:
 - numbers are finite (overflow)
 - fractions and real numbers
 - negative numbers
 - e.g., no MIPS `subi` instruction; `addi` can add a negative number)
- How do we represent negative numbers?
 - i.e., which bit patterns will represent which numbers?

Possible Representations

- | <u>Sign Magnitude</u> | <u>One's Complement</u> | <u>Two's Complement</u> |
|-----------------------|-------------------------|-------------------------|
| 000 = +0 | 000 = +0 | 000 = +0 |
| 001 = +1 | 001 = +1 | 001 = +1 |
| 010 = +2 | 010 = +2 | 010 = +2 |
| 011 = +3 | 011 = +3 | 011 = +3 |
| 100 = -0 | 100 = -3 | 100 = -4 |
| 101 = -1 | 101 = -2 | 101 = -3 |
| 110 = -2 | 110 = -1 | 110 = -2 |
| 111 = -3 | 111 = -0 | 111 = -1 |
- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

MIPS

- 32 bit signed numbers:

```

0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = + 2ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten  ← maxint
1000 0000 0000 0000 0000 0000 0000 0000two = - 2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = - 2,147,483,647ten  ← minint
1000 0000 0000 0000 0000 0000 0000 0010two = - 2,147,483,646ten
...
1111 1111 1111 1111 1111 1111 1111 1101two = - 3ten
1111 1111 1111 1111 1111 1111 1111 1110two = - 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = - 1ten

```

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
 - remember: "negate" and "invert" are quite different!
- Converting n bit numbers into numbers with more than n bits:
 - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
 - copy the most significant bit (the sign bit) into the other bits


```

0010  -> 0000 0010
1010  -> 1111 1010

```
 - "sign extension" (lbu vs. lb)

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)


```

      0111      0111      0110
    + 0110    - 0110    - 0101

```
- Two's complement operations easy
 - subtraction using addition of negative numbers


```

      0111
    + 1010

```
- Overflow (result too large for finite computer word):
 - e.g., adding two n-bit numbers does not yield an n-bit number


```

      0111
    + 0001
    1000

```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
 - overflow when adding two positives yields a negative
 - or, adding two negatives gives a positive
 - or, subtract a negative from a positive and get a negative
 - or, subtract a positive from a negative and get a positive
- Consider the operations $A + B$, and $A - B$
 - Can overflow occur if B is 0 ?
 - Can overflow occur if A is 0 ?

Effects of Overflow

- An exception (interrupt) occurs
 - Control jumps to predefined address for exception
 - Interrupted address is saved for possible resumption
 - Details based on software system / language
 - example: flight control vs. homework assignment
 - Don't always want to detect overflow
 - new MIPS instructions: `addu`, `addiu`, `subu`
- note: `addiu` still sign-extends!*
- note: `sltu`, `sltiu` for unsigned comparisons*

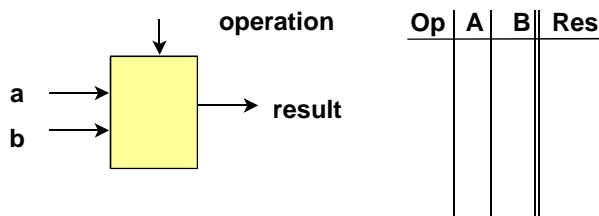
Review: Boolean Algebra & Gates

- **Problem:** Consider a logic function with three inputs: A, B, and C.

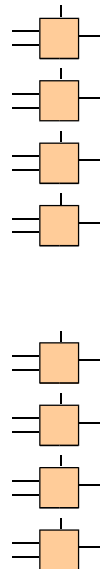
Output D is true if at least one input is true
Output E is true if exactly two inputs are true
Output F is true only if all three inputs are true
- **Show the truth table for these three functions.**
- **Show the Boolean equations for these three functions.**
- **Show an implementation consisting of inverters, AND, and OR gates.**

An ALU (arithmetic logic unit)

- Let's build an ALU to support the andi and ori instructions
 - we'll just build a 1 bit ALU, and use 32 of them

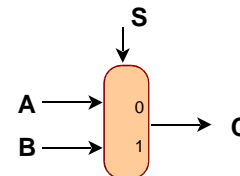


- **Possible Implementation (sum-of-products):**



Review: The Multiplexor

- **Selects one of the inputs to be the output, based on a control input**

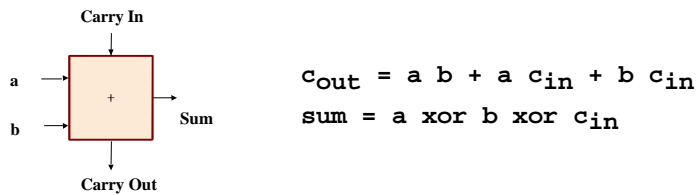


*note: we call this a 2-input mux
even though it has 3 inputs!*

- Lets build our ALU using a MUX:

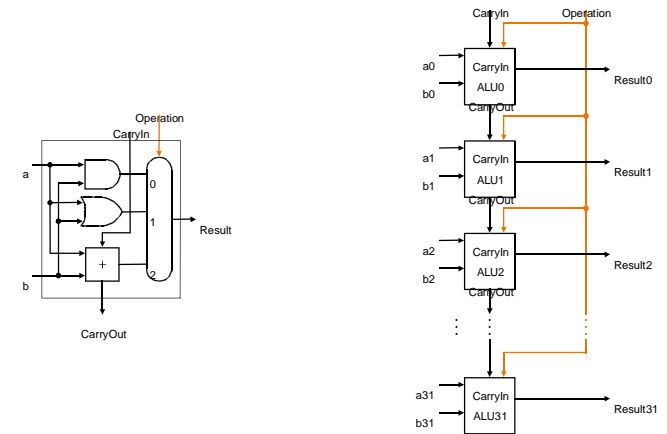
Different Implementations

- Not easy to decide the “best” way to build something
 - Don’t want too many inputs to a single gate
 - Don’t want to have to go through too many gates
 - for our purposes, ease of comprehension is important
- Let’s look at a 1-bit ALU for addition:



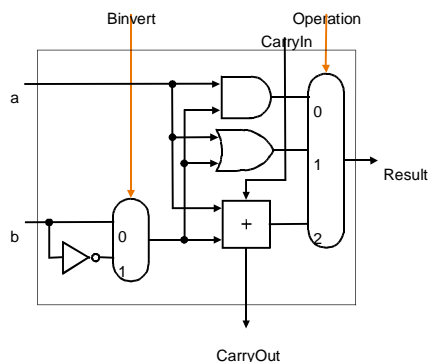
- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU



What about subtraction (a – b) ?

- Two’s complement approach: just negate b and add.
- How do we negate?
- A very clever solution:

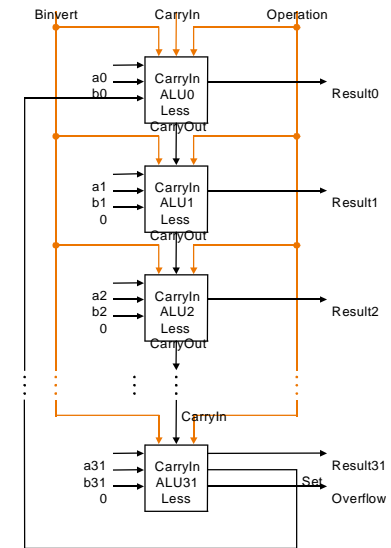
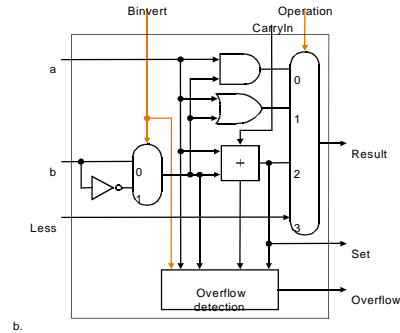
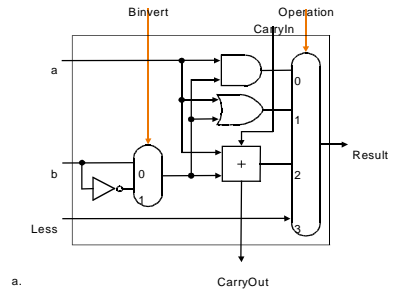


Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
 - remember: slt is an arithmetic instruction
 - produces a 1 if $rs < rt$ and 0 otherwise
 - use subtraction: $(a-b) < 0$ implies $a < b$
- Need to support test for equality (beq \$t5, \$t6, \$t7)
 - use subtraction: $(a-b) = 0$ implies $a = b$

Supporting slt

- Can we figure out the idea?

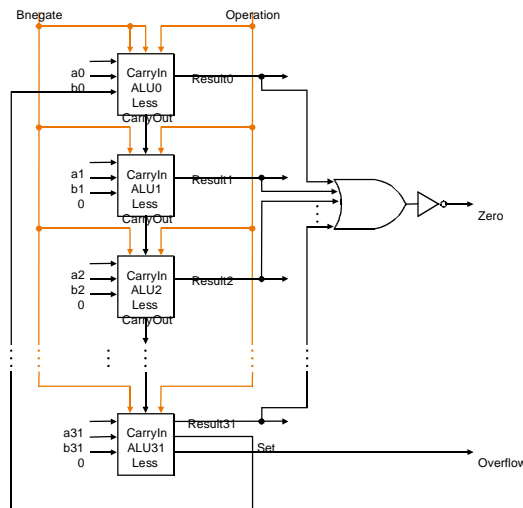


Test for equality

- Notice control lines:

000 = and
001 = or
010 = add
110 = subtract
111 = slt

•Note: zero is a 1 when the result is zero!



Conclusion

- We can build an ALU to support the MIPS instruction set
 - key idea: use multiplexor to select the output we want
 - we can efficiently perform subtraction using two's complement
 - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
 - all of the gates are always working
 - the speed of a gate is affected by the number of inputs to the gate
 - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
 - Clever changes to organization can improve performance (similar to using better algorithms in software)
 - we'll look at two examples for addition and multiplication

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
 - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

$$\begin{aligned}
 c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
 c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \quad c_2 = \\
 c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \quad c_3 = \\
 c_4 &= b_3c_3 + a_3c_3 + a_3b_3 \quad c_4 =
 \end{aligned}$$

Problem: ripple carry adder is slow

Two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

By successive substitutions of c_{i+1} by c_i

$$\begin{aligned}
 c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
 c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
 c_2 &= (b_0c_0 + a_0c_0 + a_0b_0) b_1 + (b_0c_0 + a_0c_0 + a_0b_0) a_1 + a_1b_1 \\
 &= b_0c_0b_1 + a_0c_0b_1 + a_0b_0b_1 + b_0c_0a_1 + a_0c_0a_1 \\
 &\quad + a_0b_0a_1 + a_1b_1 \\
 c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \quad c_3 = \\
 c_4 &= b_3c_3 + a_3c_3 + a_3b_3 \quad c_4 =
 \end{aligned}$$

Not feasible! Why?

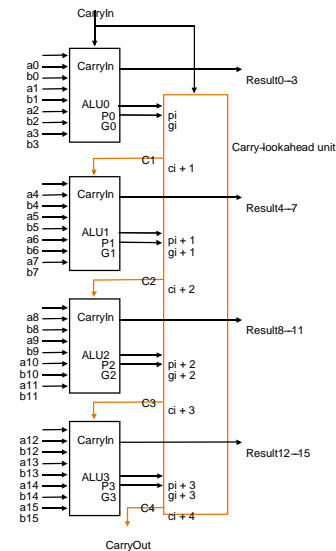
Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
 - If we didn't know the value of carry-in, what could we do?
 - When would we always generate a carry? $g_i = a_i b_i$
 - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

$$\begin{aligned}
 c_1 &= g_0 + p_0c_0 \\
 c_2 &= g_1 + p_1c_1 \quad c_2 = \\
 c_3 &= g_2 + p_2c_2 \quad c_3 = \\
 c_4 &= g_3 + p_3c_3 \quad c_4 =
 \end{aligned}$$

Feasible! Why?

Use principle to build bigger adders



- Can't build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

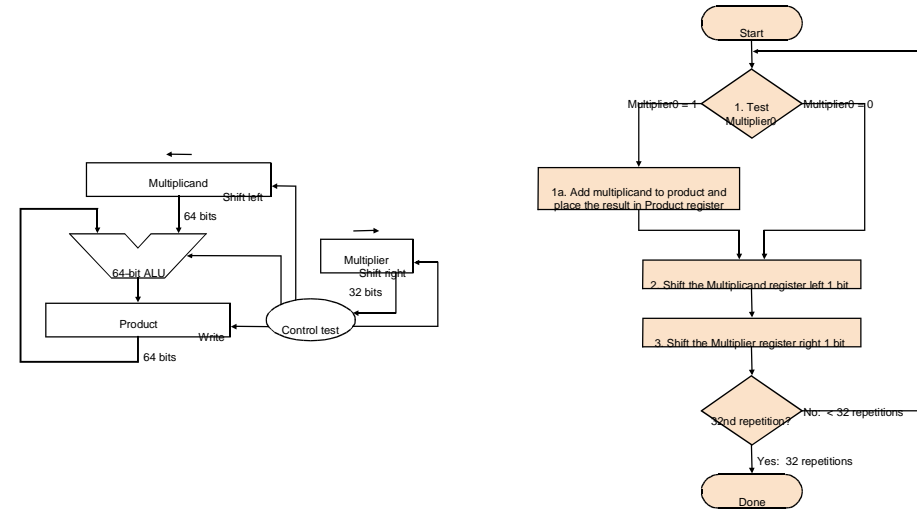
Multiplication

- More complicated than addition
 - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on gradeschool algorithm

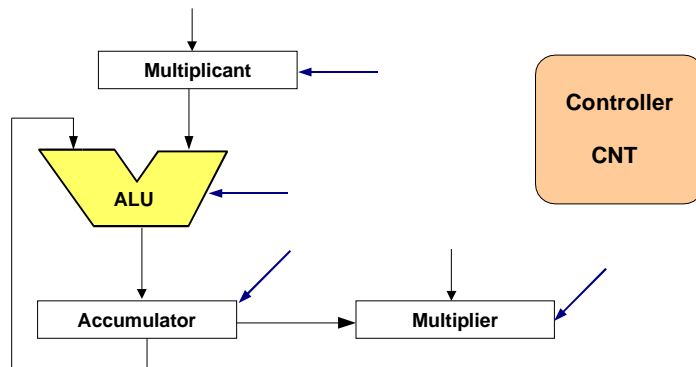
$$\begin{array}{r} 0010 \text{ (multiplicand)} \\ \times 1011 \text{ (multiplier)} \\ \hline \end{array}$$

- Negative numbers: convert and multiply
 - there are better techniques, we won't look at them

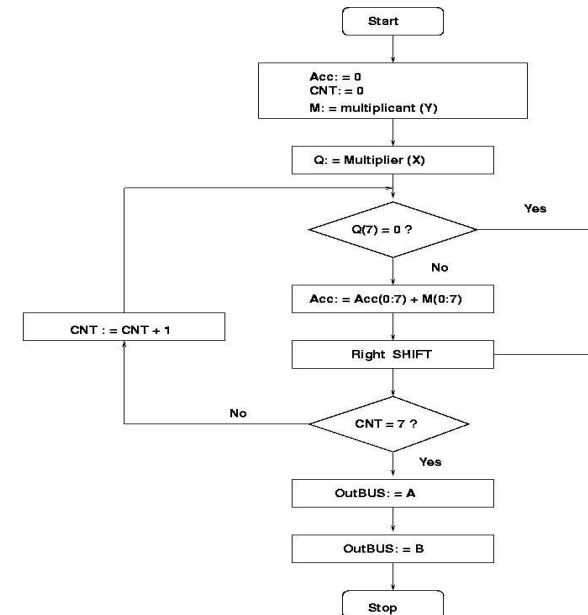
Multiplication: Implementation



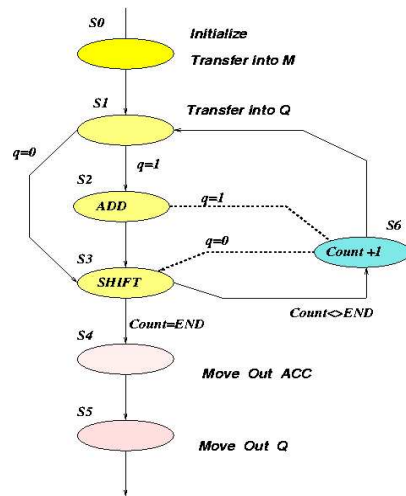
Multiplication: Implementation



Multiplication: Algorithm



State Control Machine



Division Algo

Acc	A	
00000	1110	Divide 14=1110 by 3=11. B contains 0011
00001	110	step 1: Shift
<u>-00011</u>		step 2: subtract
-00010	1100	step 3: result negative; set quotient bit to 0
00001	1100	step 4: restore
00011	100	step 1: shift
<u>-00011</u>		step 2: subtract
00000	1001	step 3: result non-negative; set quotient bit to 1
00001	001	step 1: shift
<u>-00011</u>		step 2: subtract
00010	0010	step 3: result is negative; set quotient bit 0
00001	0010	step 4: restore
00010	010	step 1: shift
<u>-00011</u>		step 2: subtract
00001	0100	step 3: result is negative; set quotient bit to 0
00010	0100	step 4: restore. Quot = 0100. Remainder = 00010

Array Multiplier

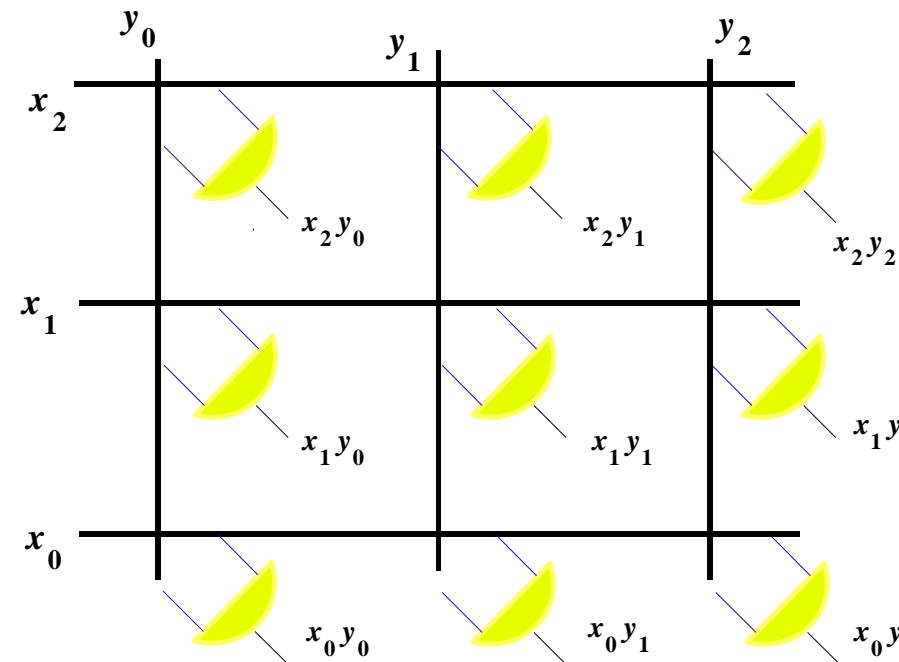
$$(x_0 2^2 + x_1 2^1 + x_2 2^0)(y_0 2^2 + y_1 2^1 + y_2 2^0)$$

$$x_0 y_0 2^{(2+2)} + x_0 y_1 2^{(2+1)} + x_0 y_2 2^{(2+0)}$$

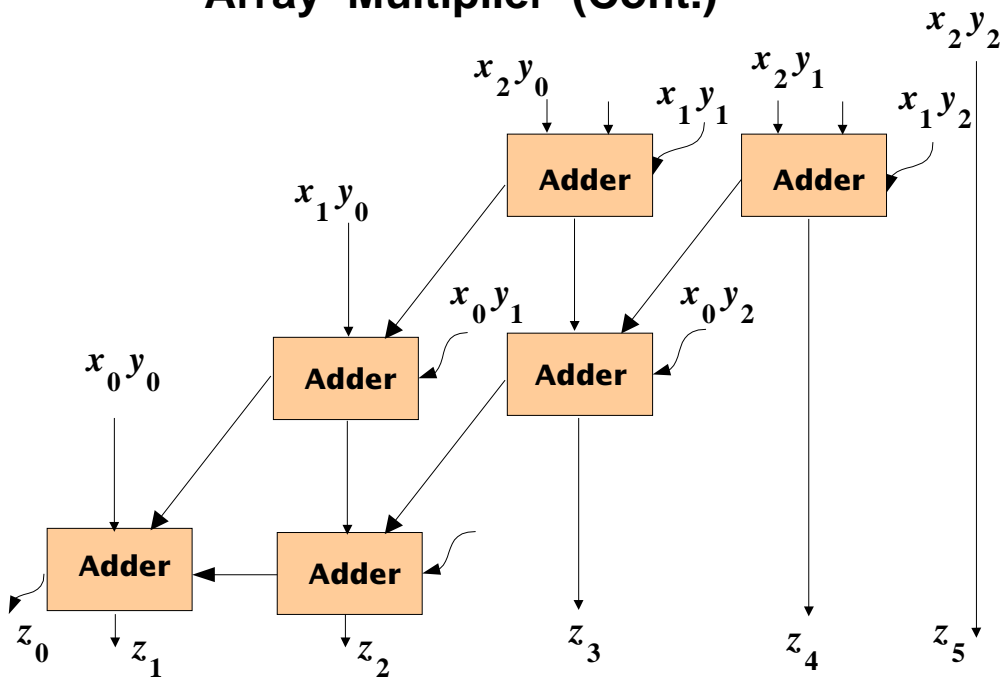
$$+ x_1 y_0 2^{(1+2)} + x_1 y_1 2^{(1+1)} + x_1 y_2 2^{(1+0)}$$

$$+ x_2 y_0 2^{(0+2)} + x_2 y_1 2^{(0+1)} + x_2 y_2 2^{(0+0)}$$

AND Array



Array Multiplier (Cont.)



Array Multiplier Basics

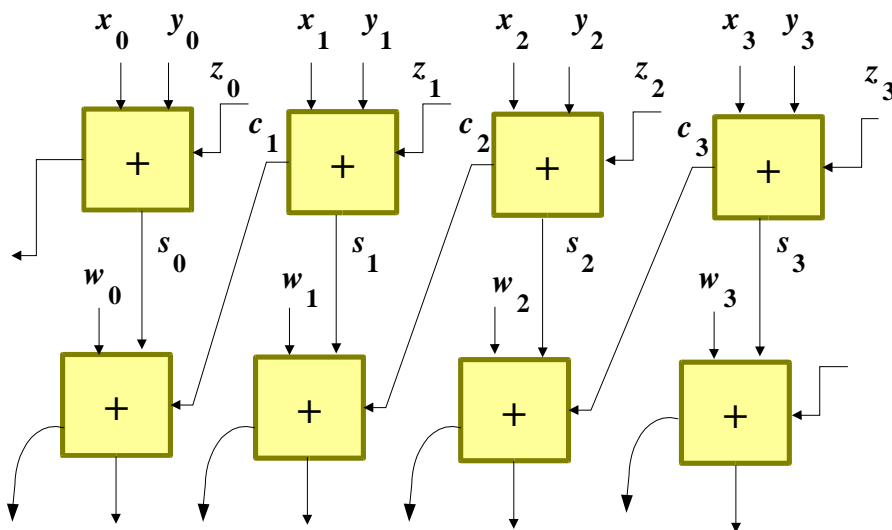
Multiplication Time for n -bit numbers = $2(n-1)D + D'$

where D and D' are the propagation delays of an Adder and an AND gate.

Component cost $\sim n^2$

ANDs and Adders can combine into a single cell.

Carry-Save Adder - 2 Stages



Carry Save Adder: Basics

The n -bit Carry Save Adder consists of n disjoint Adders

Inputs: 3 n -bit numbers to be added

Outputs: n sum bits (s_k) ; n carry bits (c_k)

No carry propagation within adder

$m \geq 3$ numbers may be added together by using a tree structure of carry-save adders.

Floating Point (a brief look)

- We need a way to represent
 - numbers with fractions, e.g., 3.1416
 - very small numbers, e.g., .000000001
 - very large numbers, e.g., 3.15576×10^9
- Representation:
 - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
 - more bits for significand gives more accuracy
 - more bits for exponent increases range
- IEEE 754 floating point standard:
 - single precision: 8 bit exponent, 23 bit significand
 - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit
- Exponent is “biased” to make sorting easier
 - all 0s is smallest exponent all 1s is largest
 - bias of 127 for single precision and 1023 for double precision
 - summary: $(-1)^{\text{sign}} \times (1 + \text{significand}) \times 2^{\text{exponent} - \text{bias}}$
- Example:
 - decimal: $-.75 = -3/4 = -3/2^2$
 - binary: $-.11 = -1.1 \times 2^{-1}$
 - floating point: exponent = 126 = 01111110
 - IEEE single precision: 10111111010000000000000000000000

Floating Point Adder

$$A = a 2^p \qquad B = b 2^q$$

$$A + B = a 2^p + b 2^q = c 2^r$$

$$r = \max(p, q) \qquad t = |p - q| \qquad \min(p, q)$$

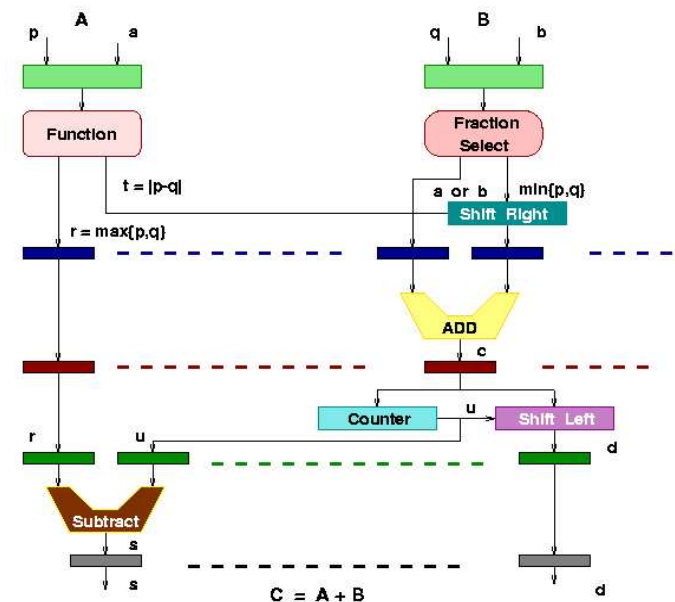
Algorithm

- Compare p and q ;
- Shift Right fraction of min exponent $\min(p, q)$ by t
- Add shifted fraction
- Count # of zeros, say u ; shift left leading zero to norm

Shift Right = Divide by 2

Shift Left = Multiply by 2

Floating Point Adder



Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
 - IEEE 754 keeps two extra bits, guard and round
 - four rounding modes
 - positive divided by zero yields “infinity”
 - zero divide by zero yields “not a number”
 - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
 - see text for description of 80x86 and Pentium bug!

Summary

- Computer arithmetic is constrained by limited precision
 - Bit patterns have no inherent meaning but standards do exist
 - two’s complement
 - IEEE 754 floating point
 - Computer instructions determine “meaning” of the bit patterns
 - Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).
-
- We are ready to move on (and implement the processor)
 - you may want to look back (Section 4.12 is great reading!)