



香港中文大學

計算機科學與工程學系

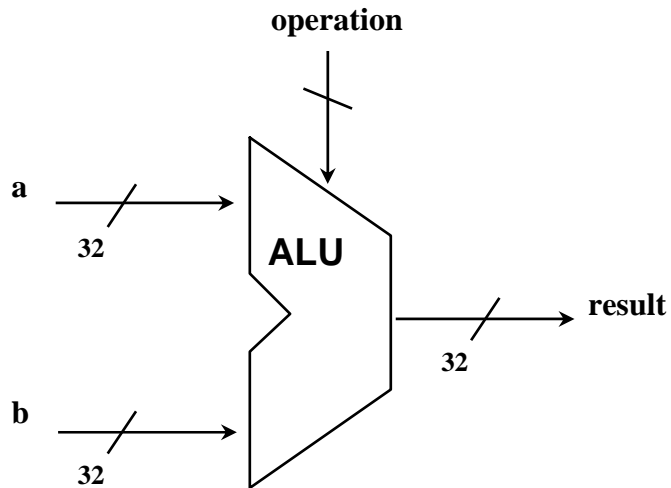
CSC 3420

Computer Systems Architecture

Chapter 4 : Arithmetic for Computer

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, 1, ... , $10^n - 1$
- **Complications:**
 - Numbers are finite (can **overflow**)
 - Fractions and real numbers
 - Negative numbers
 - Try to minimise instruction set
e.g., no MIPS subi instruction (use addi to add a negative number)
- **How do we represent negative numbers?**
i.e., which bit patterns will represent which numbers?

Possible Representations

- | Sign Magnitude: | One's Complement | Two's Complement |
|------------------------|-------------------------|-------------------------|
| 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 the best? Why?**

•32 bit signed numbers: 2's complement

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = + 1_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = + 2_{\text{ten}} \dots$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = + 2,147,483,646_{\text{ten}}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = + 2,147,483,647_{\text{ten}}$$

(maxint)

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = - 2,147,483,648_{\text{ten}}$$

(minint)

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = - 2,147,483,647_{\text{ten}}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = - 2,147,483,646_{\text{ten}}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{two}} = - 3_{\text{ten}}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = - 2_{\text{ten}}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = - 1_{\text{ten}}$$

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
 - e.g. (4-bit numbers)
 - 0010 -> 0000 0010
 - 1010 -> 1111 1010
 - "sign extension" (lbu vs. lb)

Addition & Subtraction

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

$$\begin{array}{r} 0111 \\ + 0110 \\ \hline 1101 \end{array} \qquad \begin{array}{r} 0111 \\ - 0110 \\ \hline 0001 \end{array} \qquad \begin{array}{r} 0110 \\ - 0101 \\ \hline 0001 \end{array}$$

- **Two's complement operations easy**

- subtraction using addition of negative numbers (- 0110 = 1010)

$$\begin{array}{r} 0111 \\ + 1010 \\ \hline 0001 \end{array}$$

- **Overflow (result too large for finite computer word):**

- e.g., adding two (+ve) n-bit numbers does not yield an (+ve) n-bit number

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$

*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 (pos + neg)
 - when signs are the same for subtraction (pos – pos / neg – neg)
- **Overflow occurs when the value affects the sign:**
 - overflow when adding two positives yields a negative (pos + pos -> neg)
 - or, adding two negatives gives a positive (neg + neg -> pos)
 - or, subtract a negative from a positive and get a negative (pos – neg -> neg)
 - or, subtract a positive from a negative and get a positive (neg – pos -> pos)
- **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

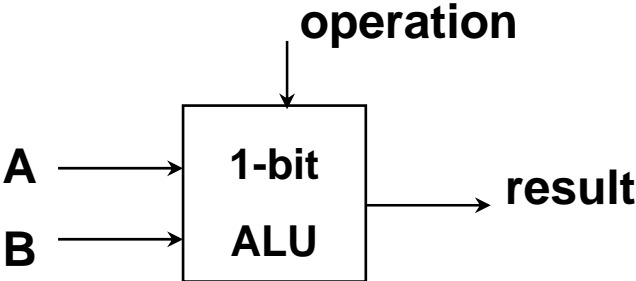
- An **exception (interrupt) occurs**
 - Control jumps to predefined address for exception
 - Interrupted address is saved for possible resumption
- **Details based on software system / language**
 - example: flight control vs. homework assignment
- **Don't always want to detect overflow**
 - new MIPS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

note: addiu still sign-extends!

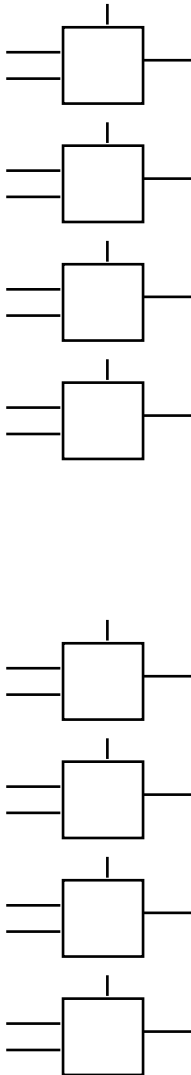
note: sltu, sltiu for unsigned comparisons

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



| op | A | B | RES |
|-------------|---|---|-----|
| andi | 0 | 0 | 0 |
| | 0 | 1 | 0 |
| | 1 | 0 | 0 |
| | 1 | 1 | 1 |
| ori | 0 | 0 | 0 |
| | 0 | 1 | 1 |
| | 1 | 0 | 1 |
| | 1 | 1 | 1 |

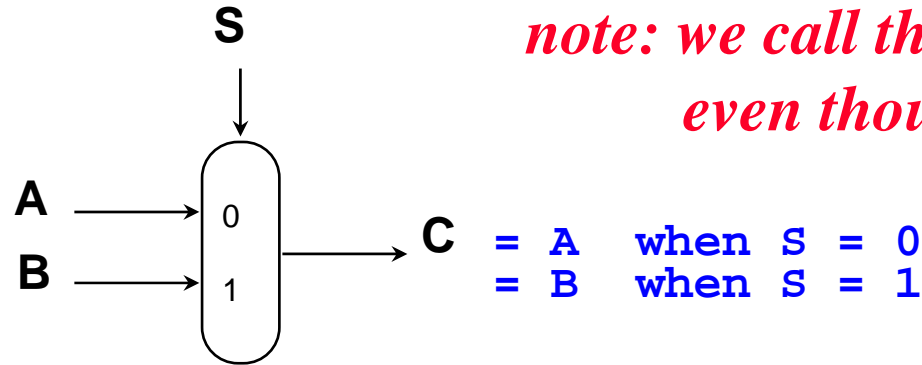


- Possible Implementation (**sum-of-products**):

$$result = andi \cdot (A \cdot B) + ori \cdot (A + B)$$

Review: The Multiplexor

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

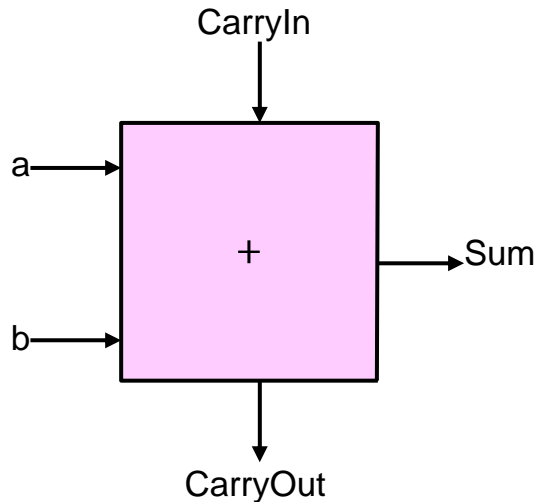


*note: we call this a 2-input mux
even though it has 3 inputs!*

- Lets build our ALU using a MUX:

Different Implementations

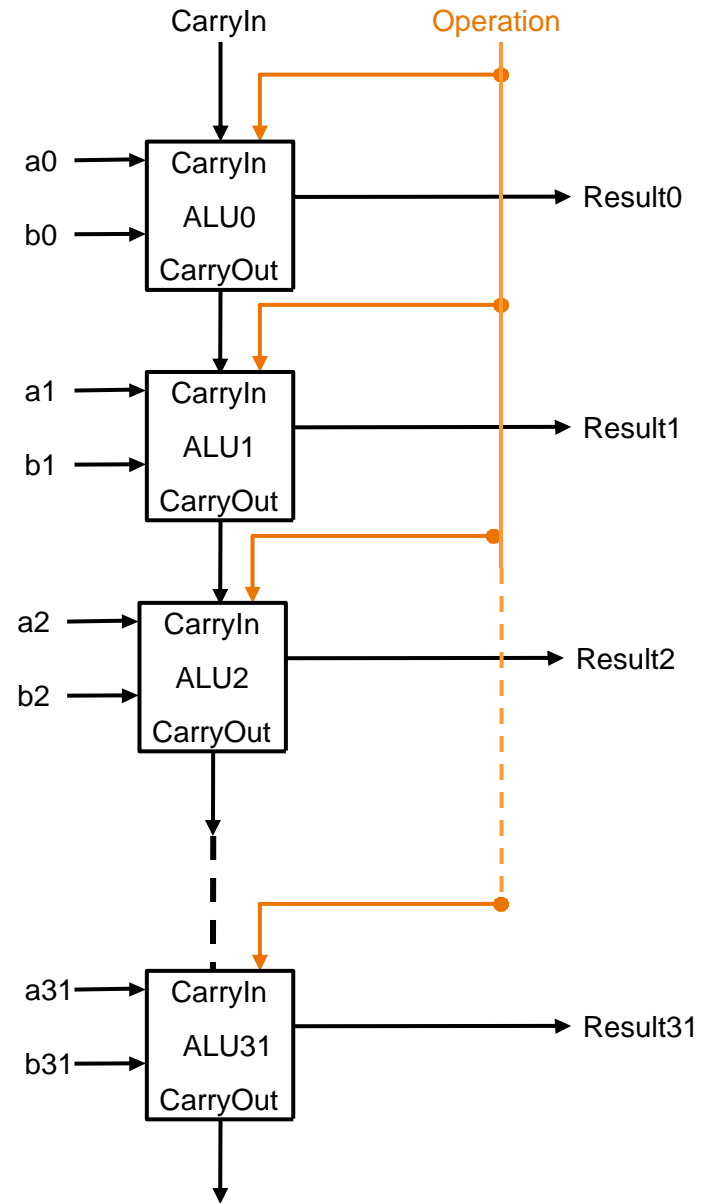
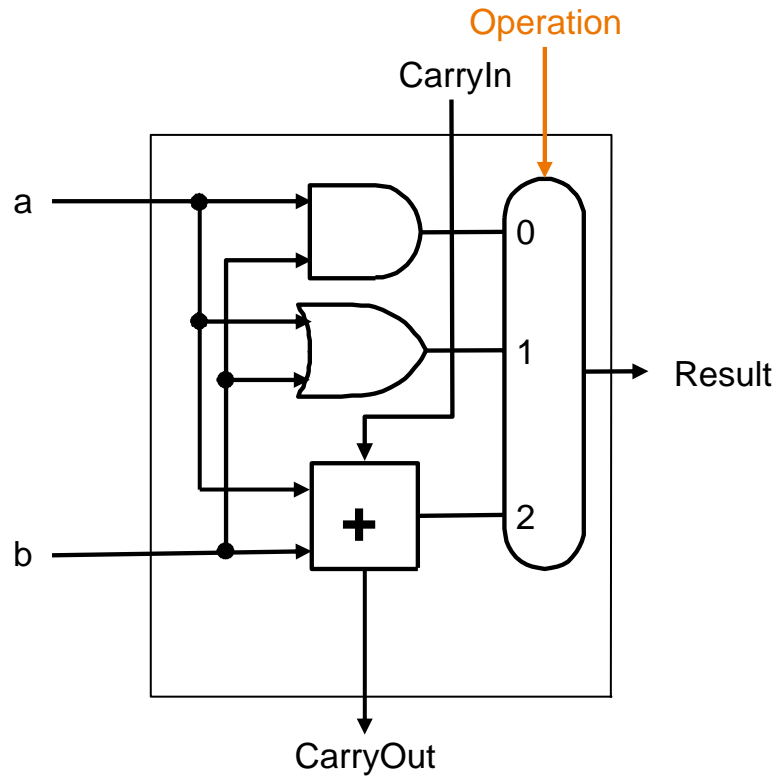
- Not easy to decide the “best” way to build something
 - Don't want too many inputs to a single gate (**Fanin problem**)
 - Don't want to have to go through too many gates (**Timing problem**)
 - For the following lectures, ease of comprehension (**領悟**) is more important
- Let's look at a **1-bit ALU for addition**:



$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$
$$sum = a \text{ XOR } b \text{ XOR } c_{in}$$

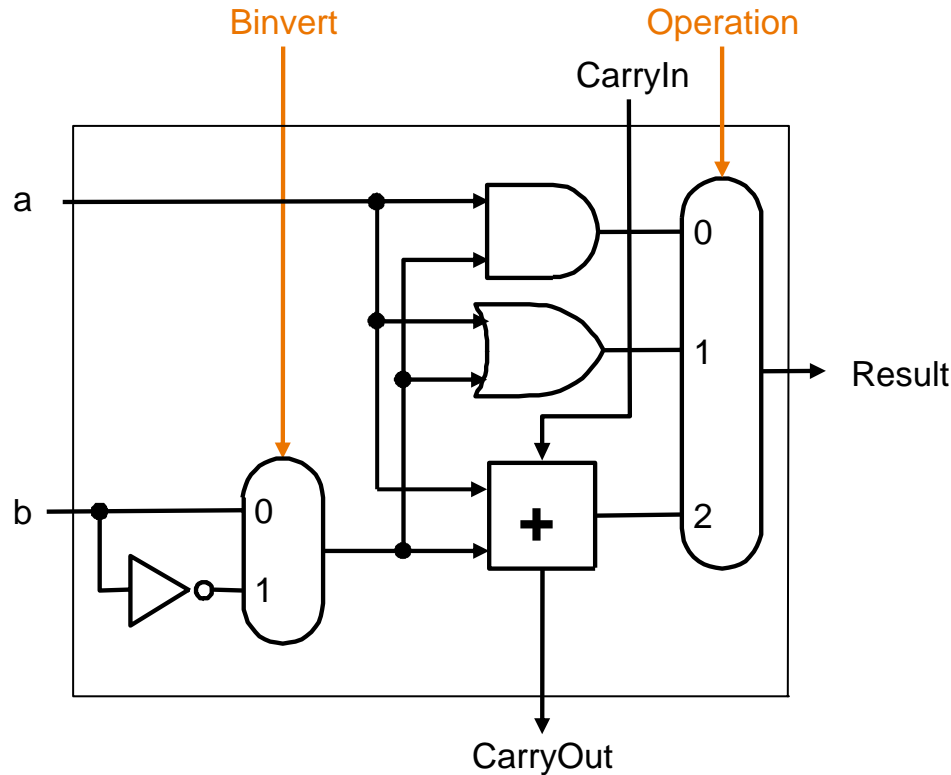
- How could we build a 1-bit ALU for ADD, AND and OR?
- How could we build a 32-bit ALU?

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:



Is the answer for 32-bit subtract correct?

Why?

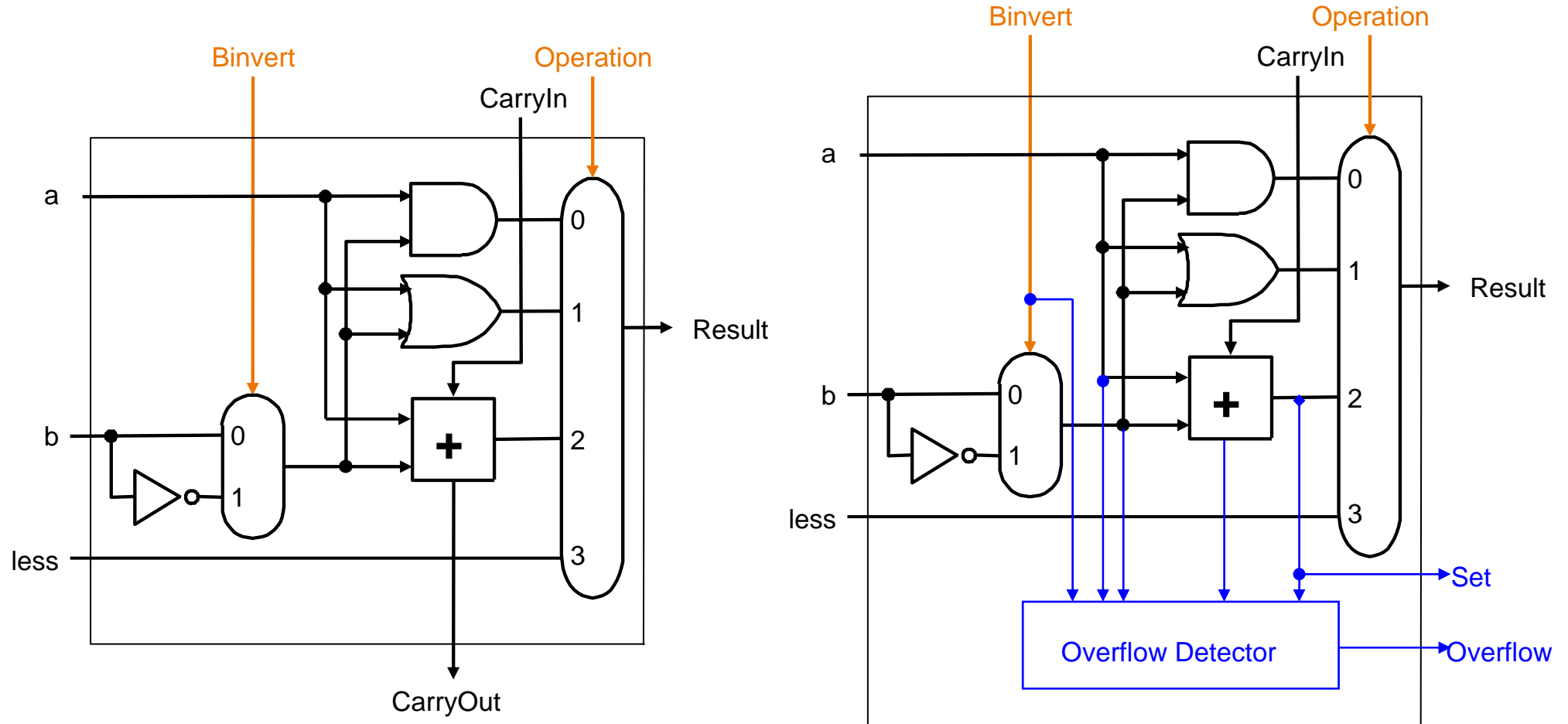
How to correct it?

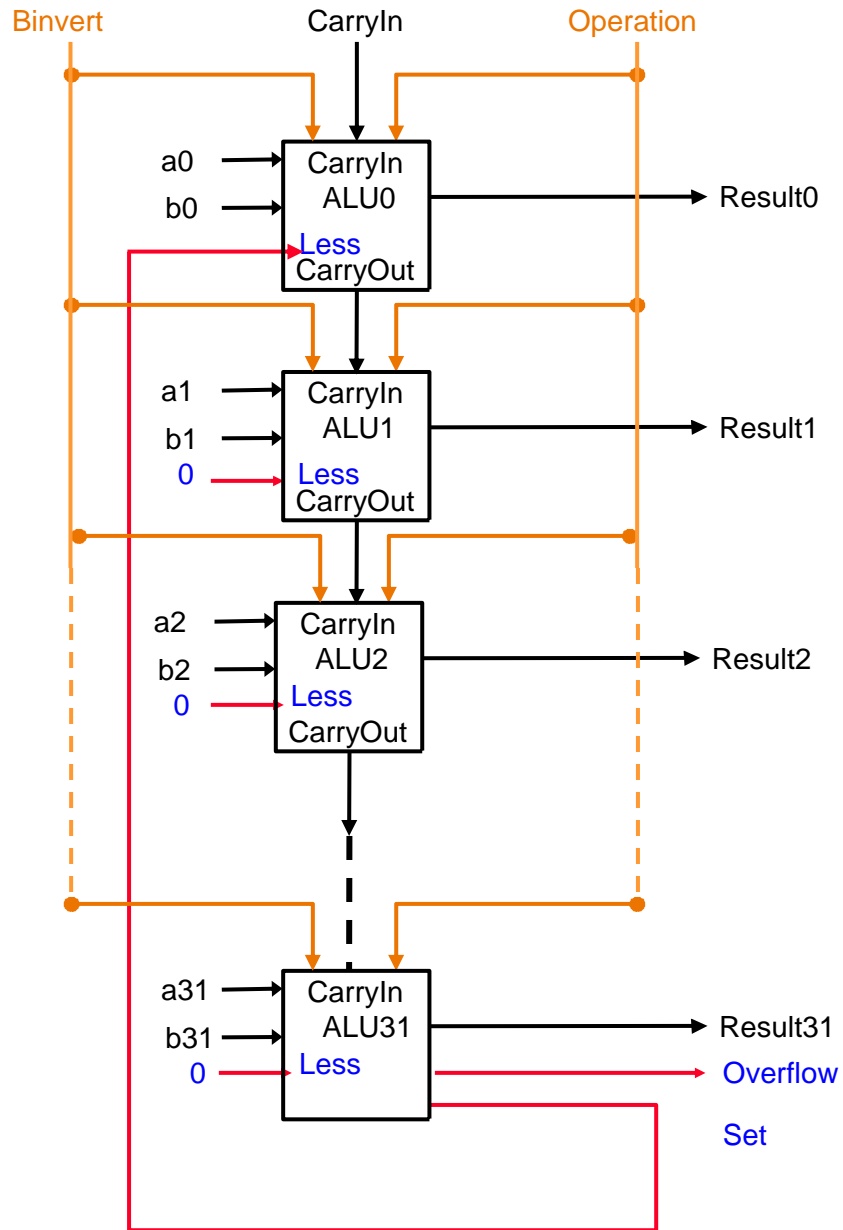
Setting Test Conditions

- Need to support the **set-on-less-than** instruction (e.g. `slt` in MIPS)
 - 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?



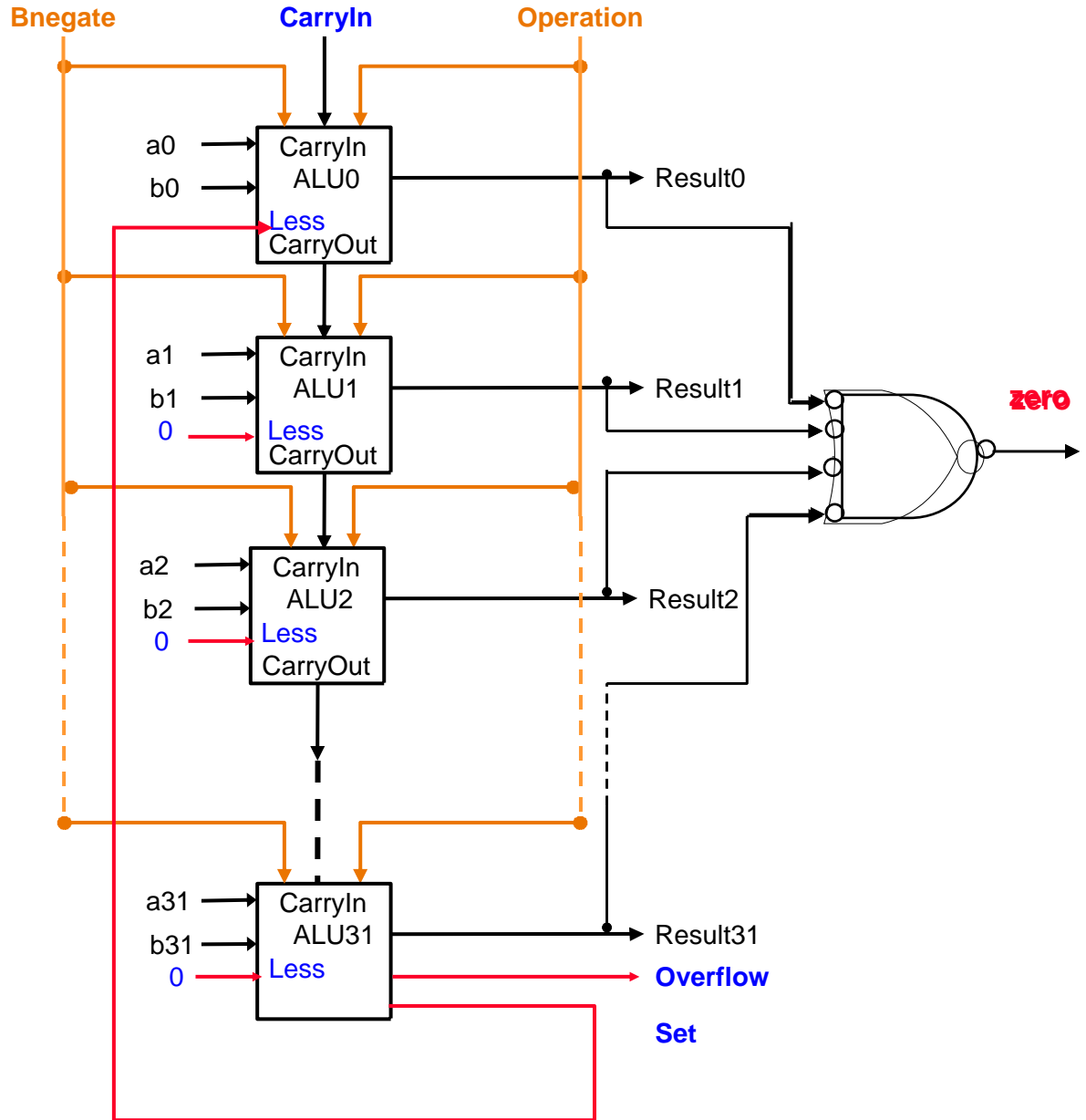


Test for equality

Control lines: (**Operation**)

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

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



Summary

- We can build **an ALU to support an 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 (in parallel)
 - 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 organisation can improve performance
(similar to using better algorithms in software)
 - we'll look at two examples: addition and multiplication

Problem: ripple carry adder is slow

- Why our 32-bit ALU is not as fast as a 1-bit ALU?

$$c_0 = a_0b_0 + b_0c_{in} + a_0c_{in}$$

$$\begin{aligned}c_1 &= a_1b_1 + b_1c_0 + a_1c_0 \\ &= a_1b_1 + (b_1 + a_1)(a_0b_0 + b_0c_{in} + a_0c_{in})\end{aligned}$$

$$\begin{aligned}c_2 &= a_2b_2 + b_2c_1 + a_2c_1 \\ &= a_2b_2 + [b_2 + a_2][a_1b_1 + (b_1 + a_1)(a_0b_0 + b_0c_{in} + a_0c_{in})]\end{aligned}$$

$$\begin{aligned}c_3 &= a_3b_3 + b_3c_2 + a_3c_2 \\ &= b_3c_3 + \{b_3 + a_3\}\{*****\}\end{aligned}$$

Not feasible! Why?

(Huge sum-of-products)

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

Is there more than one way to do addition?

- two extremes: ripple carry and sum-of-products

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**? $p_i = a_i + b_i$

$$c_0 = g_0 + p_0 c_{in}$$

$$c_1 = g_1 + p_1 c_0$$

$$c_2 = g_2 + p_2 c_1$$

$$c_3 = g_3 + p_3 c_2$$

$$c_1 = g_1 + p_1 \cdot (g_0 + p_0 c_{in})$$

$$c_2 = g_2 + p_2 \cdot [g_1 + p_1 \cdot (g_0 + p_0 c_{in})]$$

$$c_3 = g_3 + p_3 c_2 \cdot \{\dots\dots\dots\}$$

- **Did we get rid of the ripple? Why?**

A typical 32-bit Carry-lookahead adder

1. For all the 32-bit, generate all the g_i and p_i

$$g_0 = a_0 \cdot b_0$$

$$p_0 = a_0 + b_0$$

$$g_1 = a_1 \cdot b_1$$

$$p_1 = a_1 + b_1$$

$$g_2 = a_2 \cdot b_2$$

$$p_2 = a_2 + b_2$$

•

•

•

•

•

$$g_{31} = a_{31} \cdot b_{31}$$

$$p_{31} = a_{31} + b_{31}$$

(All 32 g_i and p_i are generated in parallel)

2. Divide the 32-bits into eight 4-bit blocks

Work out if this block will generate or propagate a carry

| Block | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
|-------|-------------|-------------|-------------|-------------|-------------|-----------|---------|---------|
| Bits | 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |

For B0

$$BG_0 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$

$$BP_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

For B1

$$BG_1 = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4$$

$$BP_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$

.

.

For B7

$$BG_7 = g_{31} + p_{31} \cdot g_{30} + p_{31} \cdot p_{30} \cdot g_{29} + p_{31} \cdot p_{30} \cdot p_{29} \cdot g_{28}$$

$$BP_7 = p_{31} \cdot p_{30} \cdot p_{29} \cdot p_{28}$$

3. Divide the eight 4-bits blocks into two super blocks

Work out super block generate and super block propagate

| Super block | SB1 | | | | SB0 | | | |
|-------------|-------------|-------------|-------------|-------------|-------------|-----------|---------|---------|
| Block | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
| Bits | 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |

For SB0

$$SBG_0 = BG_3 + BP_3 \cdot BG_2 + BP_3 \cdot BP_2 \cdot BG_1 + BP_3 \cdot BP_2 \cdot BP_1 \cdot BG_0$$

$$SBP_0 = BP_3 \cdot BP_2 \cdot BP_1 \cdot BP_0$$

For SB1

$$SBG_1 = BG_7 + BP_7 \cdot BG_6 + BP_7 \cdot BP_6 \cdot BG_5 + BP_7 \cdot BP_6 \cdot BP_5 \cdot BG_4$$

$$SBP_1 = BP_7 \cdot BP_6 \cdot BP_5 \cdot BP_4$$

4. Work out the carries from the super blocks

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------------------|------------|----|----|----|-----------|----|----|----|------------|----|----|----|-----------|----|----|----|-----------|----|----|----|-----------|----|---|---|-----------|---|---|---|-----------|---|---|---|
| Super block | SB1 | | | | | | | | SB0 | | | | | | | | | | | | | | | | | | | | | | | |
| Block | B7 | | | | B6 | | | | B5 | | | | B4 | | | | B3 | | | | B2 | | | | B1 | | | | B0 | | | |
| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

From SB0: $C_{15} = SBG_0 + SBP_0 \cdot C_{in}$

From SB1: $C_{31} = SBG_1 + SBP_1 \cdot SBG_0 + SBP_1 \cdot SBP_0 \cdot C_{in}$

5. Work out the carries from the blocks

| | | | | | | | | |
|-------------|-------------|-------------|-------------|-------------|-------------|-----------|---------|---------|
| Super block | SB1 | | | | SB0 | | | |
| Block | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
| Bits | 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |

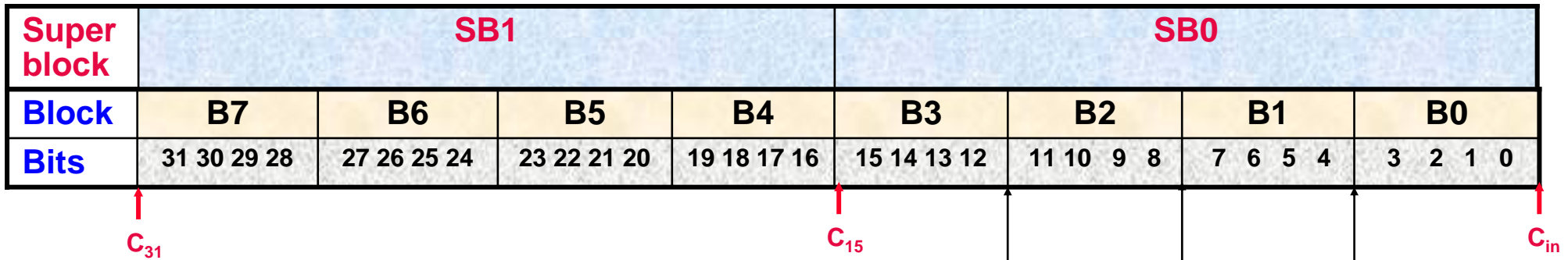
C_{31}

C_{15}

$$C_{19} = BG_4 + BP_4 \cdot C_{15}$$

$$C_{23} = BG_5 + BP_5 \cdot BG_4 + BP_5 \cdot BP_4 \cdot C_{15}$$

$$C_{27} = BG_6 + BP_6 \cdot BG_5 + BP_6 \cdot BP_5 \cdot BG_4 + BP_6 \cdot BP_5 \cdot BP_4 \cdot C_{15}$$

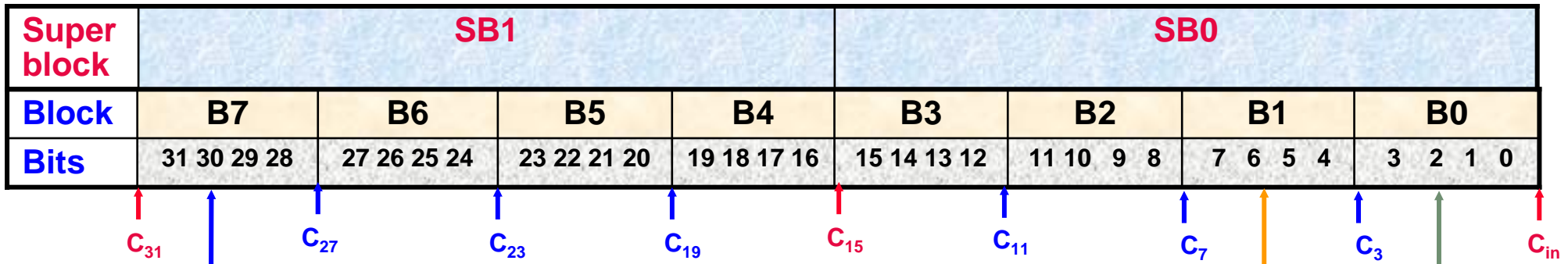


$$C_{11} = BG_2 + BP_2 \cdot BG_1 + BP_2 \cdot BP_1 \cdot BG_0 + BP_2 \cdot BP_1 \cdot BP_0 \cdot C_{in}$$

$$C_7 = BG_1 + BP_1 \cdot BG_0 + BP_1 \cdot BP_0 \cdot C_{in}$$

$$C_3 = BG_0 + BP_0 \cdot C_{in}$$

6. Work out all the missing carries



For B0

$$c_0 = g_0 + p_0 \cdot c_{in}$$

$$c_1 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_{in}$$

$$c_2 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_{in}$$

For B1

$$c_4 = g_4 + p_4 \cdot c_3$$

$$c_5 = g_5 + p_5 \cdot g_4 + p_5 \cdot p_4 \cdot c_3$$

$$c_6 = g_6 + p_6 \cdot g_5 + p_6 \cdot p_5 \cdot g_4 + p_6 \cdot p_5 \cdot p_4 \cdot c_3$$

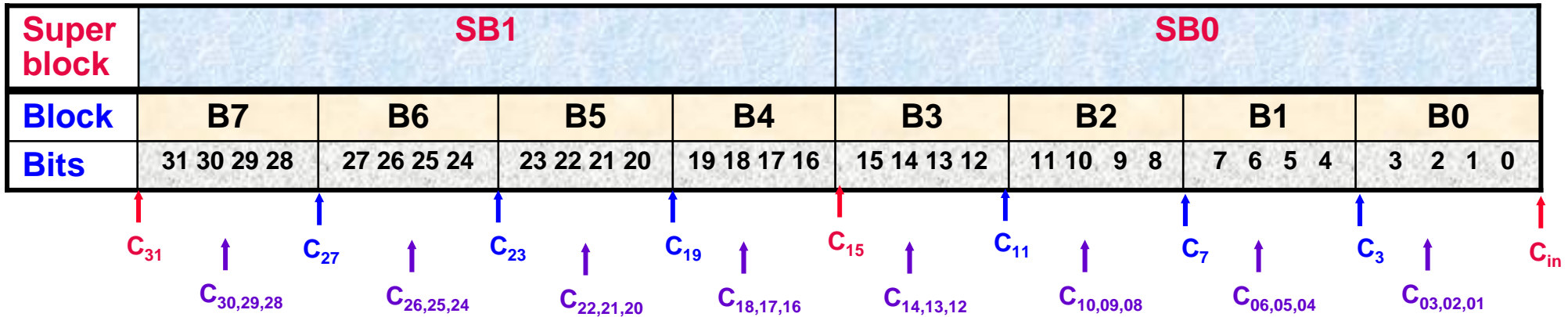
For B7

$$c_{28} = g_{28} + p_{28} \cdot c_{27}$$

$$c_{29} = g_{29} + p_{29} \cdot g_{28} + p_{29} \cdot p_{28} \cdot c_{27}$$

$$c_{30} = g_{30} + p_{30} \cdot g_{29} + p_{30} \cdot p_{29} \cdot g_{28} + p_{30} \cdot p_{29} \cdot p_{28} \cdot c_{27}$$

7. Work out all the sums (all the carries are now known)



$$S_0 = a_0 \oplus b_0 \oplus c_{in}$$

$$S_1 = a_1 \oplus b_1 \oplus c_0$$

.

$$S_n = a_n \oplus b_n \oplus c_{n-1}$$

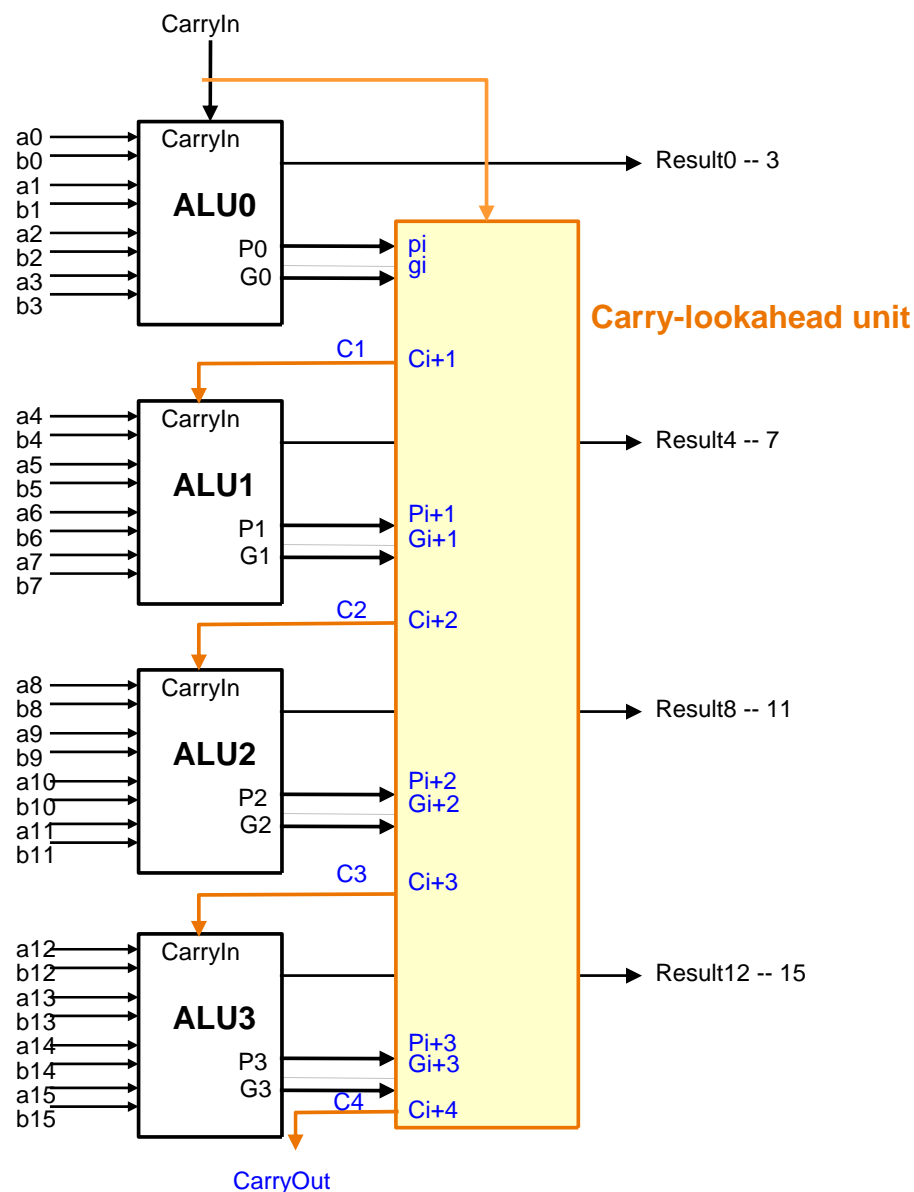
- All 32 S_i are generated in parallel
- Any short cut for S_i ?
- How fast is the circuit?
- How much does it cost?
- How to build even bigger adders?

Recap

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bits | 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 09 08 | 07 06 05 04 | 03 02 01 00 |
| pi, gi | p[31-00] pi = ai + bi | | | | g[31-00] gi = ai . Pi | | | |
| Block | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
| B_{Pi}, B_{Gi} | BP7,BG7 | BP6,BG6 | BP5,BG5 | BP4,BG4 | BP3,BG3 | BP2,BG2 | BP1,BG1 | BP0,BG0 |
| Super Block | Super Block 1 | | | | Super Block 0 | | | |
| SB_{Pi}, SB_{Gi} | SBP1, SBG1 | | | | SBP0, SBG0 | | | |
| Super Block | Super Block 1 | | | | Super Block 0 | | | |
| Super Block C_i | C₃₁ = SBG1 + SBP1.SBG0 + SBP1.SBP0.C_{in} | | | | C₁₅ = SBG0 + SBP0.C_{in} | | | |
| Block | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
| Block level C_i's | | C₂₇ | C₂₃ | C₁₉ | | C₁₁ | C₀₇ | C₀₃ |
| Block | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
| Other C_i's | C₃₀ C₂₉ C₂₈ | C₂₆ C₂₅ C₂₄ | C₂₂ C₂₁ C₂₀ | C₁₈ C₁₇ C₁₆ | C₁₄ C₁₃ C₁₂ | C₁₀ C₀₉ C₀₈ | C₀₆ C₀₅ C₀₄ | C₀₂ C₀₁ C₀₀ |
| Bits | 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 09 08 | 07 06 05 04 | 03 02 01 00 |
| Sums | S[31-00] | | | | S_i = pi NEQ ci | | | |

Use Building Blocks to construct bigger adders

- Build 4-bit ALU blocks with P_i and G_i
- Build a Block Carry Look Ahead Unit.
- 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 (chip) area**
- **Let's look at 3 versions based on primary school algorithm**

- | | | | | | | |
|----------------------------|----------|----------|----------|----------|----------------|------------------------------------|
| | 0 | 0 | 1 | 0 | (multiplicand) | (2 in decimal) |
| x | 1 | 0 | 1 | 1 | (multiplier) | (11 in decimal) |
| <hr style="width: 100%;"/> | | | | | | |
| | 0 | 0 | 1 | 0 | | |
| | 0 | 0 | 1 | 0 | | |
| | 0 | 0 | 0 | 0 | | |
| | 0 | 0 | 1 | 0 | | |
| <hr style="width: 100%;"/> | | | | | | |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| | | | | | (product) | (22 in decimal) (unsigned numbers) |

- **Negative numbers: convert and multiply**
 - Other techniques exist

◦ **What do we learn from the example?**

◦

| | |
|--------------|---------------|
| Multiplicand | 1 0 0 0 |
| Multiplier | x 1 0 0 1 |
| | <hr/> |
| | 1 0 0 0 |
| | 0 0 0 0 |
| | 0 0 0 0 |
| | 1 0 0 0 |
| | <hr/> |
| Product | 1 0 0 1 0 0 0 |

◦ **m bits x n bits = m+n bit product**

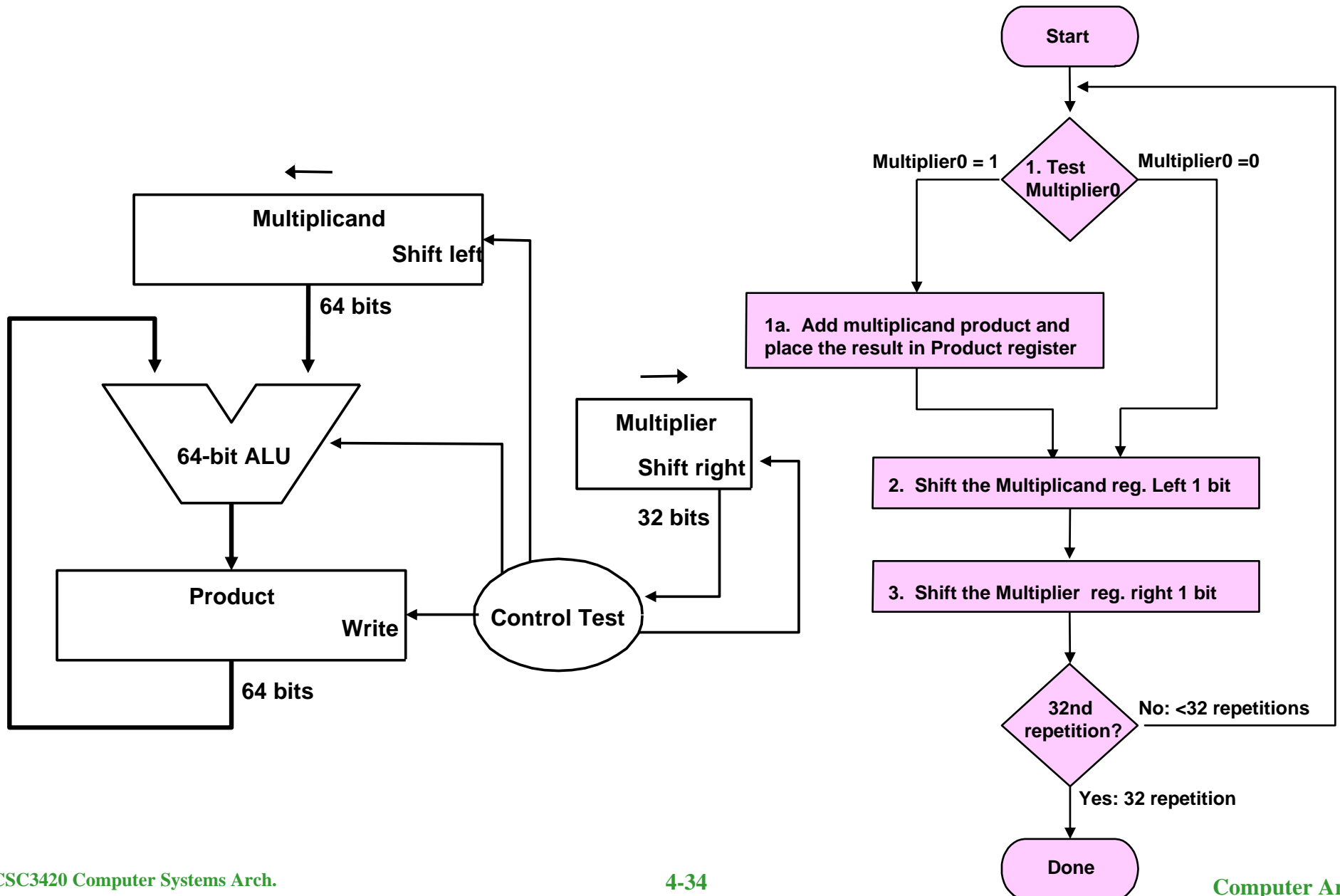
◦ **Binary makes it easy:**

- **0 => place 0 (0 x multiplicand)**
- **1 => copy multiplicand (1 x multiplicand)**

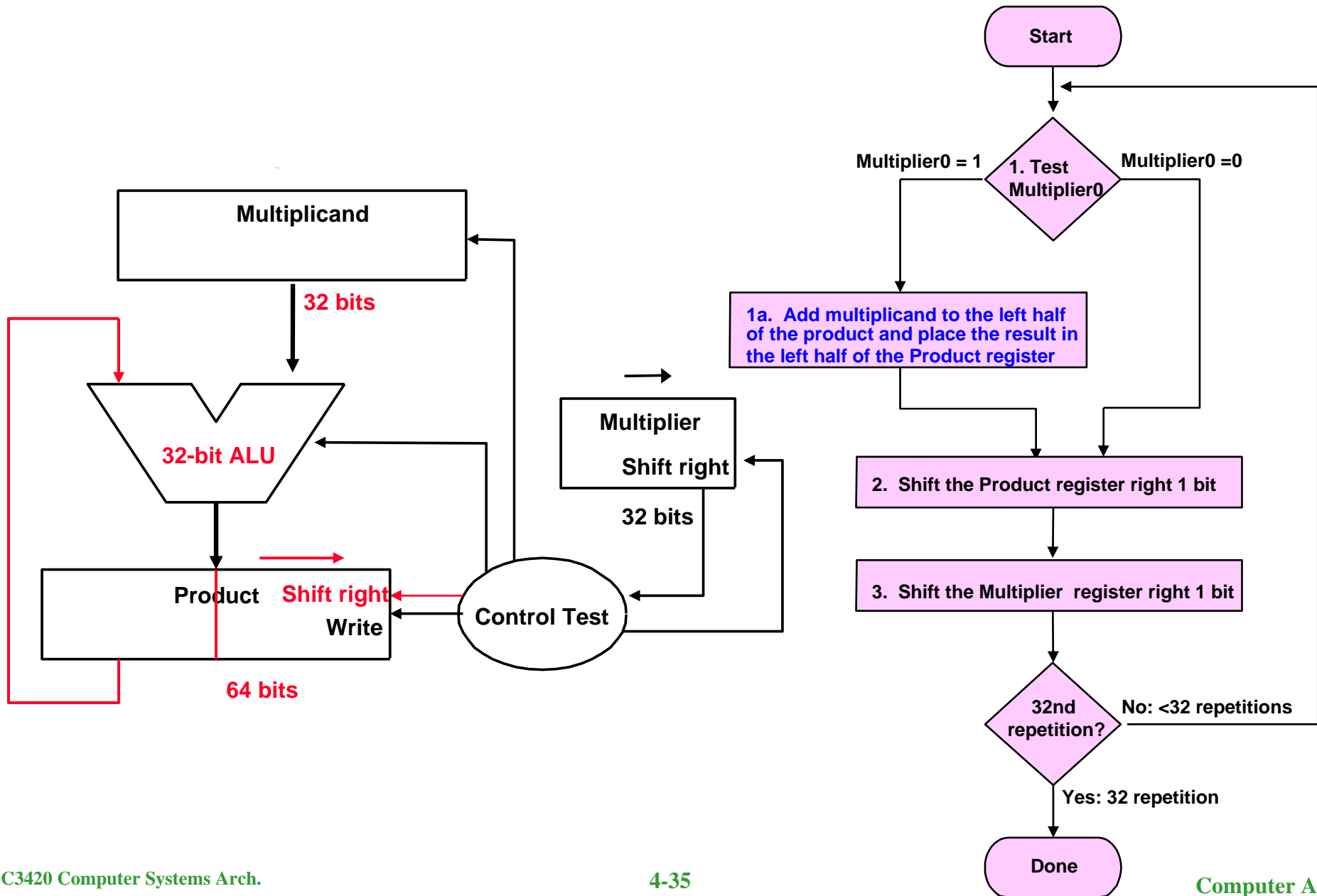
◦ **3 versions of multiply hardware & algorithm (successive refinement) will be presented**

- **see text book figs 3.5 – 3.7**
- **32-bit Multiplicand register, 32 or 64-bit ALU, 64-bit Product register,**
- **32-bit Multiplier (shift) register**

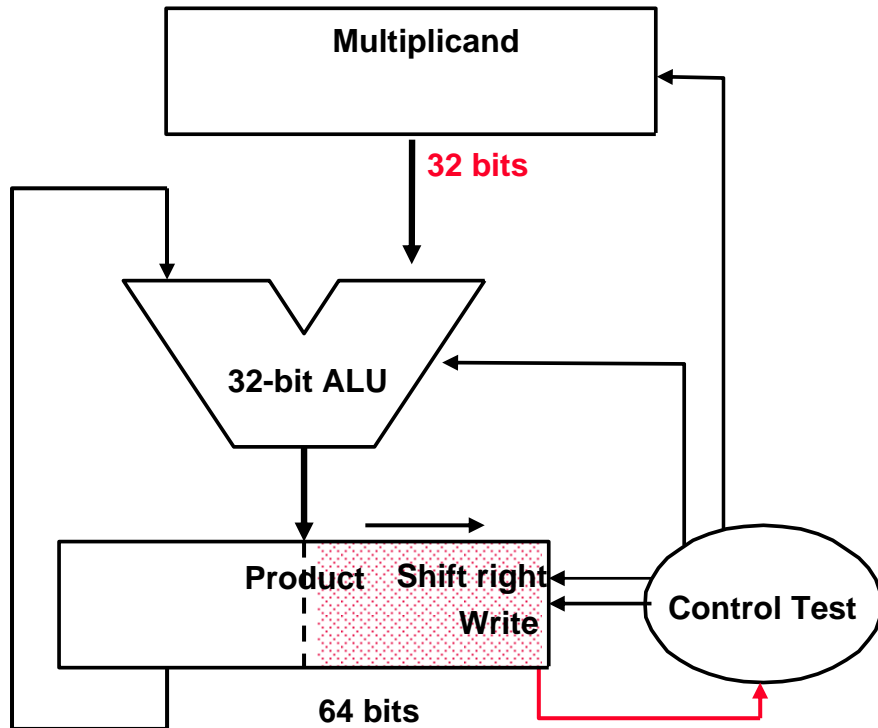
Simple Add (64-bit adder)-shift Multiplier



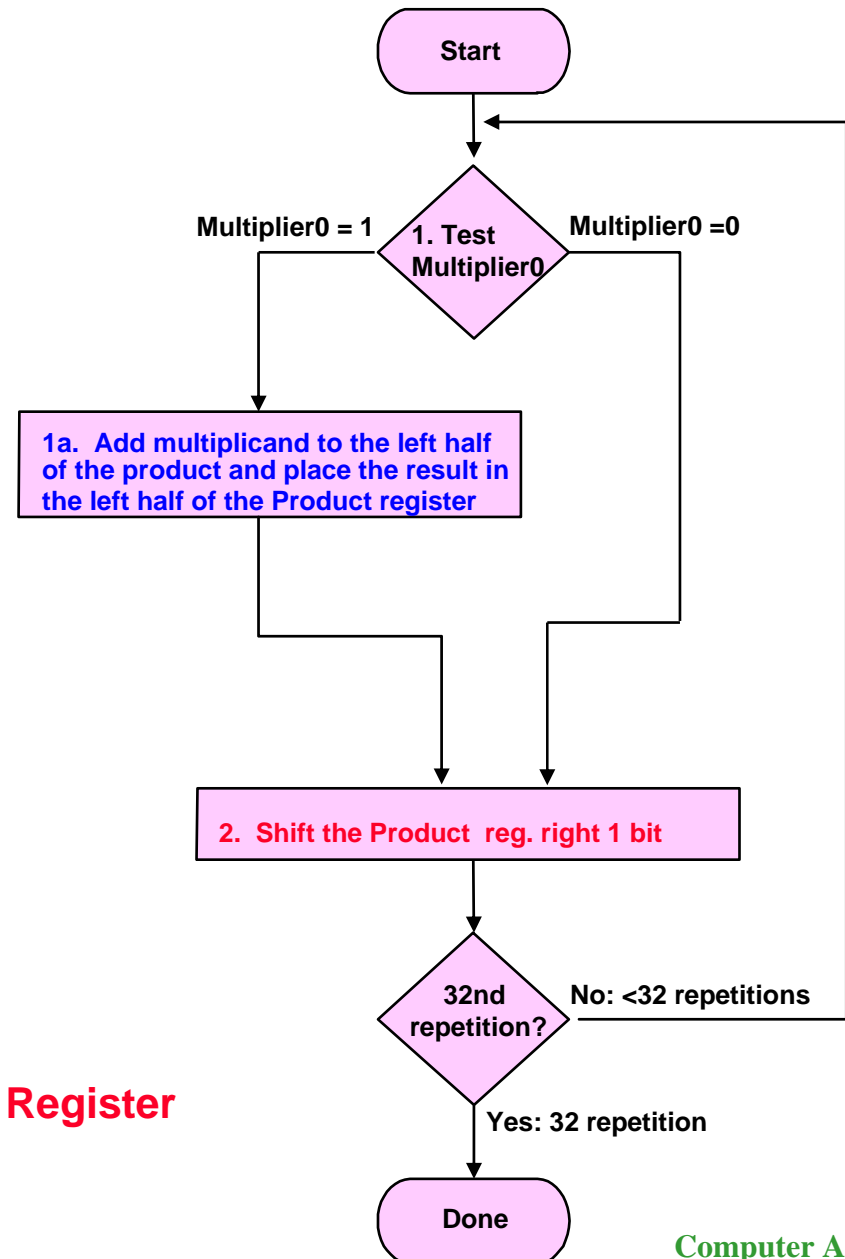
Second Version: 32-bit ALU/Multiplicand register



Final Version: No 32-bit multiplier register



Place multiplier in the LS half of the Product Register



Signed Multiplication

- Lets consider the following (4-bit) signed multiplication example:

+7 (0111) -7 (1001)
+6 (0110) -6 (1010)

Case 1: (+ve MPD, +ve MPR)

| | | |
|-------|-----------------|-------|
| | 0 1 1 1 | (+7) |
| * | 0 1 1 0 | (+6) |
| <hr/> | | |
| | 0 0 0 0 1 1 1 0 | |
| | 0 0 0 1 1 1 0 0 | |
| | <hr/> | |
| | 0 0 1 0 1 0 1 0 | (+42) |

Case 2: (-ve MPD, +ve MPR)

| | | |
|-------|-----------------|-----------------|
| | 1 0 0 1 | (- 7) |
| * | 0 1 1 0 | (+6) |
| <hr/> | | |
| | 1 1 1 1 0 0 1 0 | (sign extended) |
| | 1 1 1 0 0 1 0 0 | (sign extended) |
| | <hr/> | |
| | 1 1 0 1 0 1 1 0 | (- 42) |

Case 3: (+ve MPD, -ve MPR)

| | | |
|-------|-----------------|---------|
| | 0 1 1 1 | (+7) |
| * | 1 0 1 0 | (- 6) |
| <hr/> | | |
| | 0 0 0 0 1 1 1 0 | |
| | 0 0 1 1 1 0 0 0 | |
| | <hr/> | |
| | 0 1 0 0 0 1 1 0 | (+70??) |

Signed Multiplication (negative multiplier)

- **Intuition** : (often used)
 - convert multiplier to a positive number
 - perform multiplication
 - convert result back to the expected polarity
- **Better method** :

$$2^{2n} - [A \times (2^n - B)]$$

$$= 2^{2n} + A \times B - 2^n \times A$$

Case 3: (+ve MPD, -ve MPR)

| | | |
|---|-------------------|---------------------|
| | 0 1 1 1 | (+7) |
| * | 1 0 1 0 | (- 6) |
| | 0 0 0 0 1 1 1 0 | |
| | 0 0 1 1 1 0 0 0 | |
| | 0 1 0 0 0 1 1 0 | |
| | - 0 1 1 1 0 0 0 0 | (-2 ⁿ A) |
| | 1 1 0 1 0 1 1 0 | (- 42) |

Case 4: (-ve MPD, -ve MPR)

| | | |
|---|-------------------|---------------------|
| | 1 0 0 1 | (- 7) |
| * | 1 0 1 0 | (- 6) |
| | 1 1 1 1 0 0 1 0 | (sign extended) |
| | 1 1 0 0 1 0 0 0 | (sign extended) |
| | 1 0 1 1 1 0 1 0 | |
| | - 1 0 0 1 0 0 0 0 | (-2 ⁿ A) |
| | 0 0 1 0 1 0 1 0 | (+ 42) |

Booth Algorithm (Read it if you are interested)

- **Observations on the add-shift Multipliers**
 - Both add and shifts are performed in one clock cycle by hardware
 - Needs N clocks for N-bit multiplication
 - MIPS registers Hi and Lo are left and right half of Product
 - Gives us MIPS instruction MultU
- **Multiplier performance is critical**
 - Seek ways to reduce clocks per bit
 - Multiply more bits per clock
 - Most multiplier has less than 8 significant bits
- **Booth's Algorithm**
 - A more elegant way to multiply shorter numbers

Motivation for Booth's Algorithm (Read it if you are interested)

- Example $2 \times 6 = 0010 \times 0110$:

| | | | | | | | |
|--|---|---|---|---|---|------------------------------|---|
| | | 0 | 0 | 1 | 0 | | |
| x | | 0 | 1 | 1 | 0 | | |
| <hr style="border: 0.5px solid black;"/> | | | | | | | |
| + | | 0 | 0 | 0 | 0 | shift(0 in multiplier) | |
| + | 0 | 0 | 1 | 0 | | add & shift(1 in multiplier) | |
| + | 0 | 0 | 1 | 0 | | add & shift(1 in multiplier) | |
| + | 0 | 0 | 0 | 0 | | shift(0 in multiplier) | |
| <hr style="border: 0.5px solid black;"/> | | | | | | | |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

- **ALU with ADD or Subtract gets same result in more than one way:**

$$6 = -2 + 8 \quad (\text{in decimal})$$

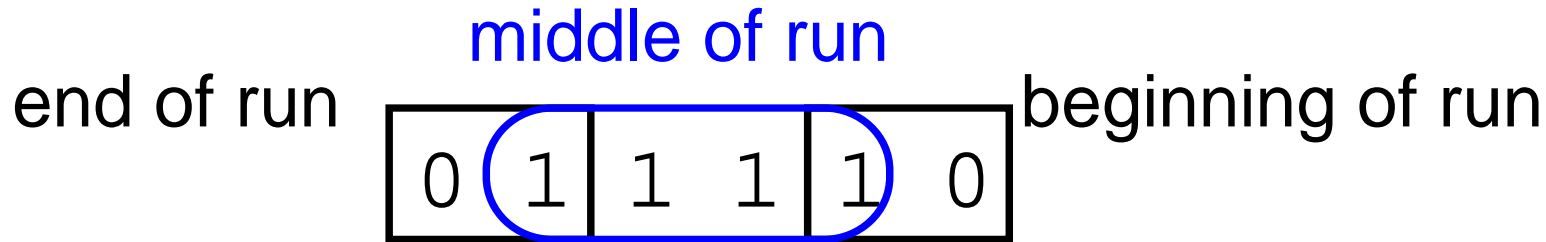
$$0110 = -0010 + 1000 \quad (\text{in binary})$$

- **Replace a string of 1's in multiplier with an initial subtract** when we first see a one and then later add for the bit after the last one.

- For example

| | | | | | | | |
|--|---|---|---|---|---|--------------------------------|-------------------------------------|
| | | 0 | 0 | 1 | 0 | | |
| x | | 0 | 1 | 1 | 0 | | |
| <hr style="border: 0.5px solid black;"/> | | | | | | | |
| + | | 0 | 0 | 0 | 0 | shift (0 in multiplier) | |
| + | 1 | 1 | 0 | 0 | 1 | 0 | add -MD(1 in multiplier) |
| + | 0 | 0 | 0 | 0 | | shift (middle of string of 1s) | |
| + | 0 | 0 | 1 | 0 | | add (prior step had last 1) | |
| <hr style="border: 0.5px solid black;"/> | | | | | | | |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Booth's Algorithm Insight (Read it if you are interested)



| Current Bit | Bit to the Right | Explanation | Example |
|-------------|------------------|--------------------------|---------------------|
| 1 | 0 | Beginning of a run of 1s | 000111 <u>1</u> 000 |
| 1 | 1 | Middle of a run of 1s | 000111 <u>1</u> 000 |
| 0 | 1 | End of a run of 1s | 000 <u>1</u> 111000 |
| 0 | 0 | Middle of a run of 0s | <u>0</u> 001111000 |

Originally for Speed since shift faster than add for earlier machines

Booth's Algorithm (Read it if you are interested)

1. Scan the multiplier from right (l.s.) to left (m.s.) and do the followings:
 - a. **00**: Middle of a string of 0s, so no arithmetic operations.
 - b. **10**: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
 - c. **01**: End of a string of 1s, so add the multiplicand to the left half of the product
 - d. **11**: Middle of a string of 1s, so no arithmetic operation.
2. As in the previous algorithm, shift the Product register right (arithmetic) 1 bit.

Try the following examples ($6 * 3$ and $6 * -3$)

| | | | |
|--------------|--------------------|--------------|--------------------|
| Multiplicand | Product/multiplier | Multiplicand | Product/multiplier |
| 0110 | 0000 0011 | 0110 | 0000 1101 |

3. The worst case is when the multiplier is a series of alternating "1" and "0"
1 0 1 0 1 0 . . . 1 0 because arithmetic is need at every bit position

A hardware solution: Multiplier Matrix

- Can we deal with more than 1 bit of multiplier at any time?
- Bits are not added in parallel because carries are generated sequentially (ripple effect)
- Can we add all bits and ignore the carries for the time being?
- Based on **Carry-save adder**

Addition of three bits (A, B, C)

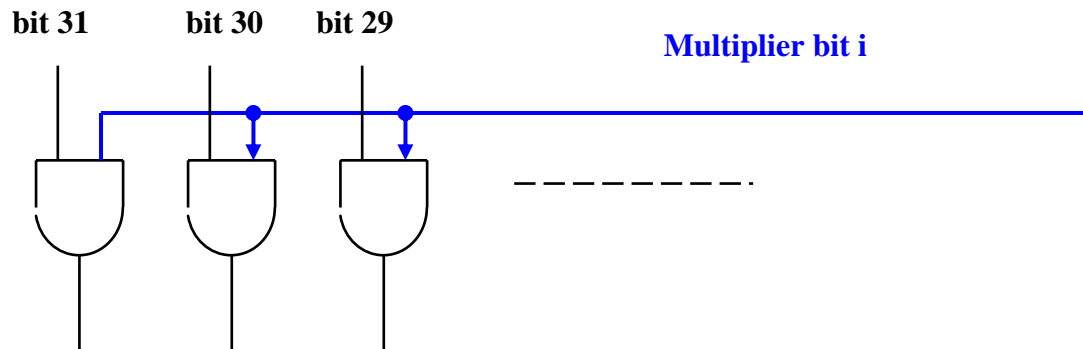
| <u>A</u> | <u>B</u> | <u>C</u> | <u>Carry</u> | <u>Sum</u> |
|----------|----------|----------|--------------|------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$$sum = \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot C$$

$$carry = A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C + A \cdot B \cdot C$$

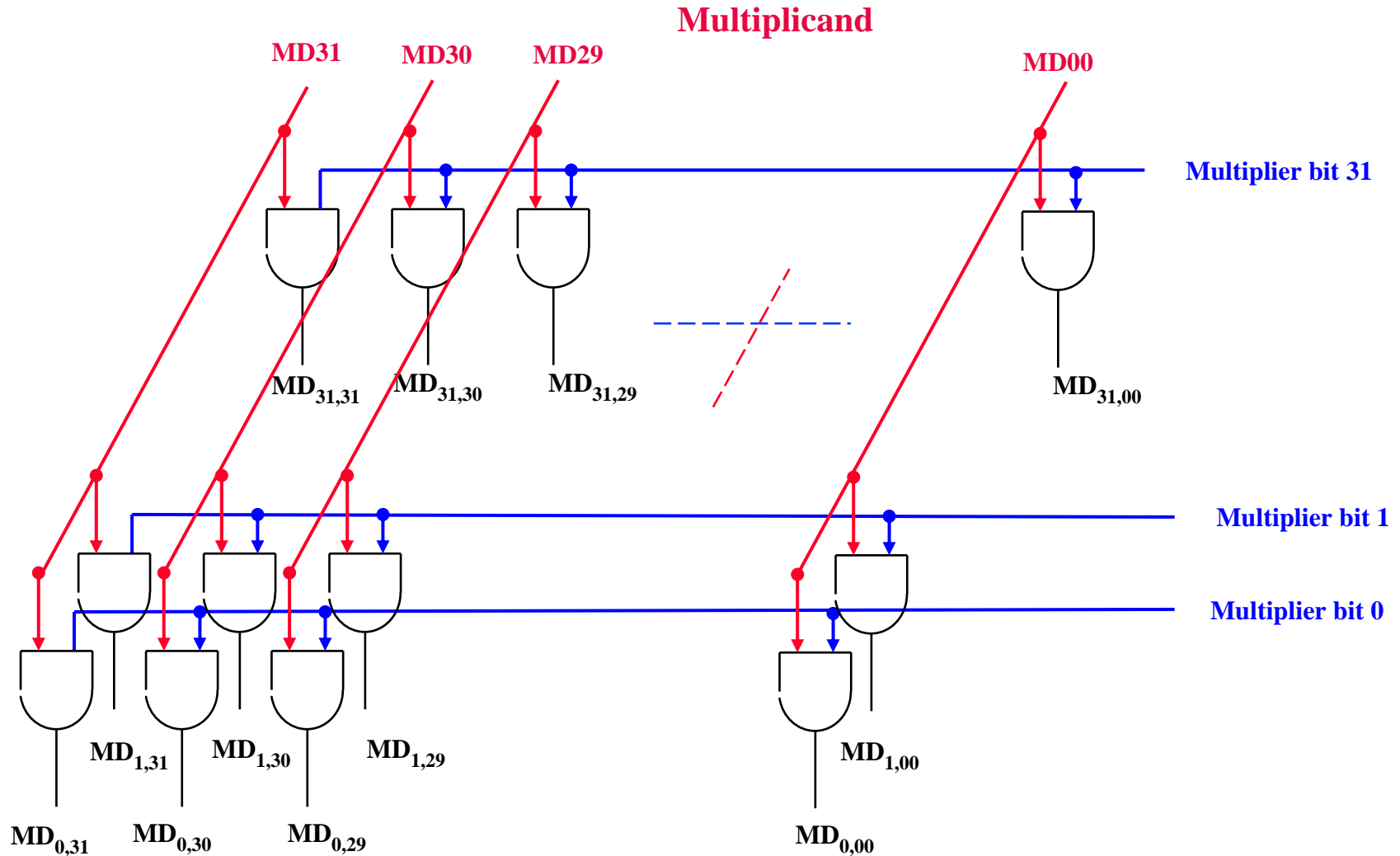
A 32 bit multiplier matrix

- Use 31 levels of carry save adder and each is 32 bit wide
- **For level i:**

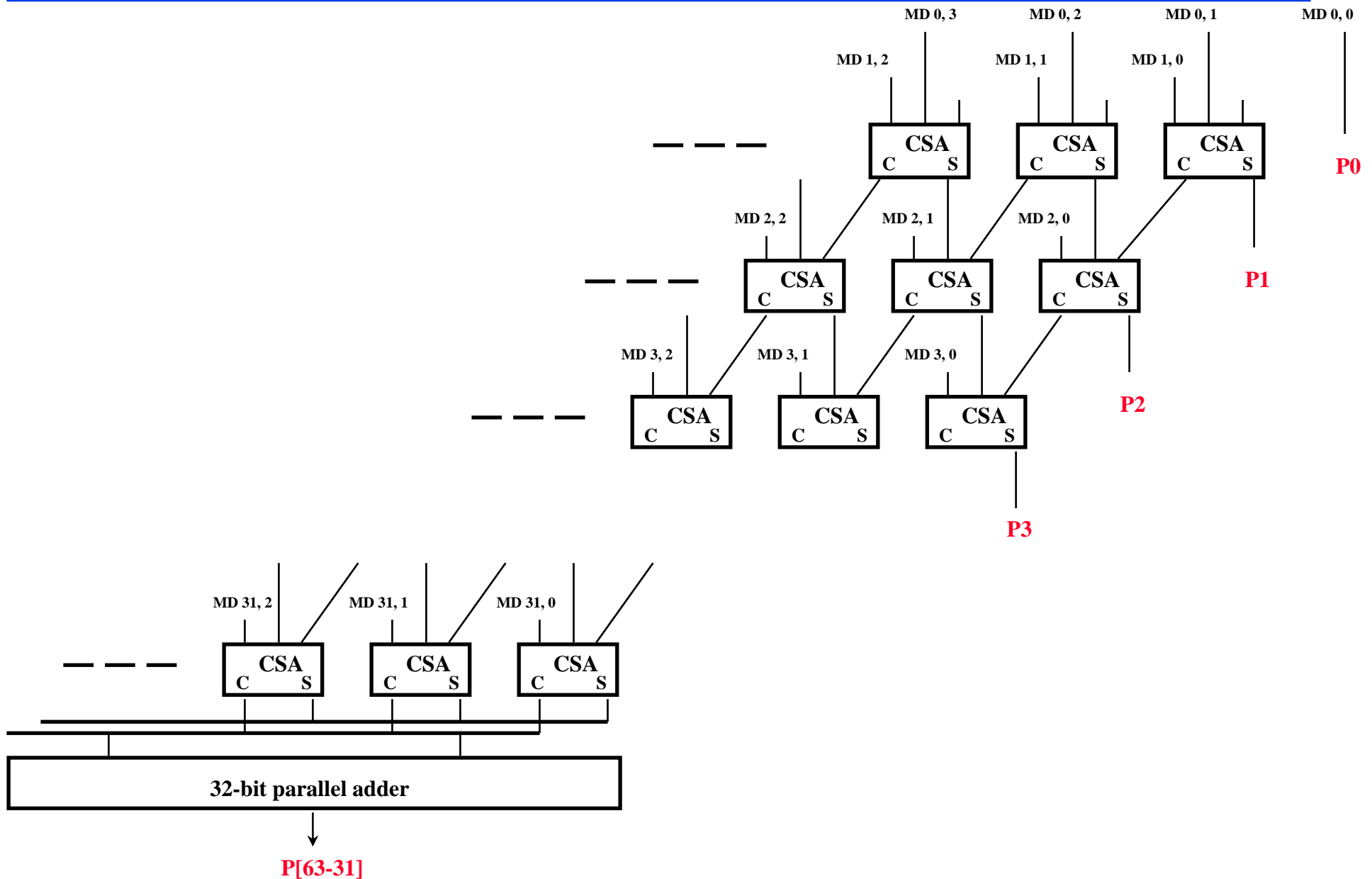


A 32 bit multiplier matrix

- Use 31 levels of carry save adder and each is 32 bit wide



n-1 levels of Carry Save Adders:



- **Achieve speed at an extra cost**

$(n - 1) * n$ [carry-save-adders] + n^2 [2-input AND gates]

Each carry-save-adder = 2 * [3-3-3-3 AND-OR gate]

- **Speed and cost compromise**

Use k levels of carry-save-adders

n/k clocks delay

- **Signed Multiplication**

Need sign extension and use 2's complement addition

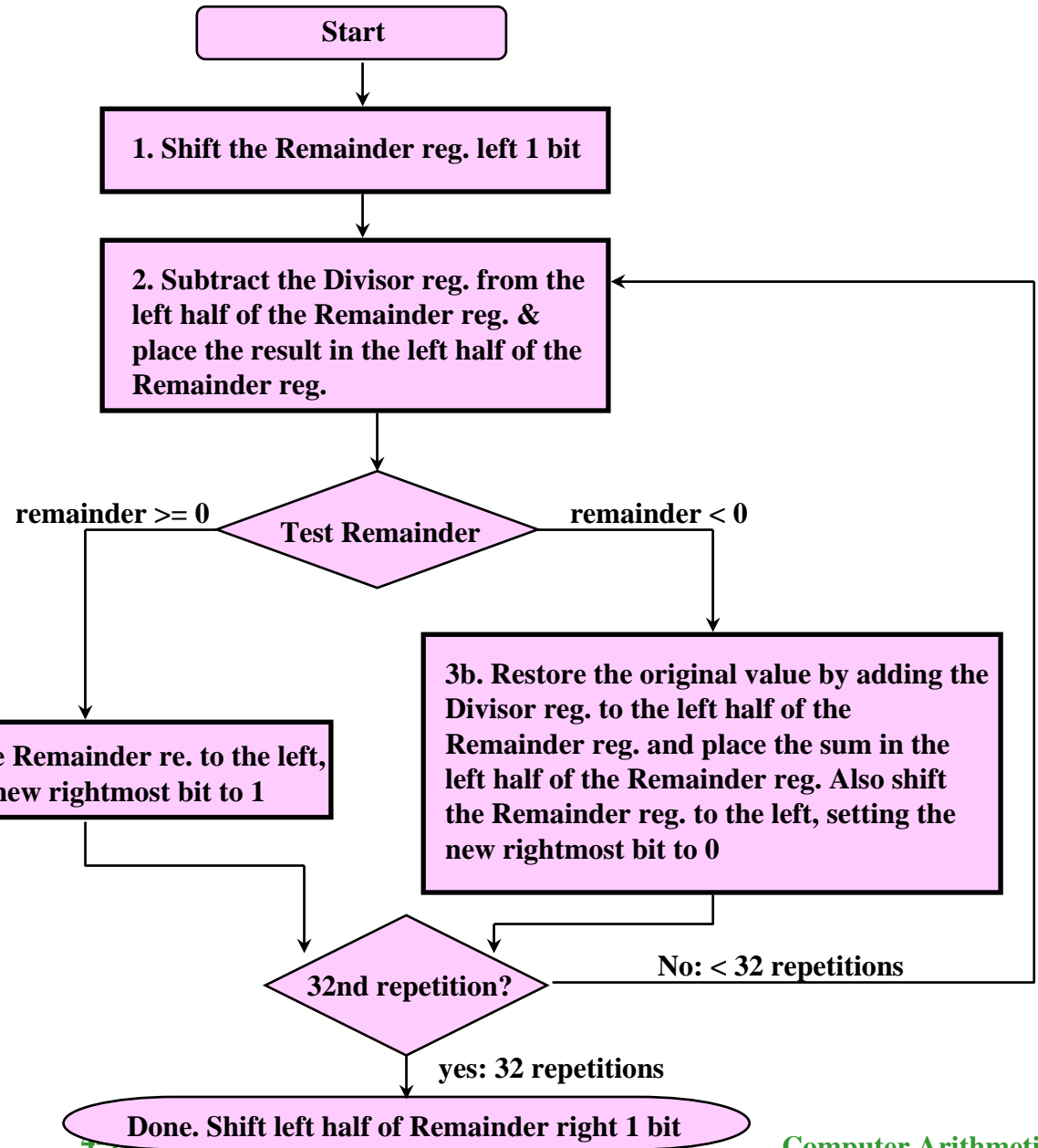
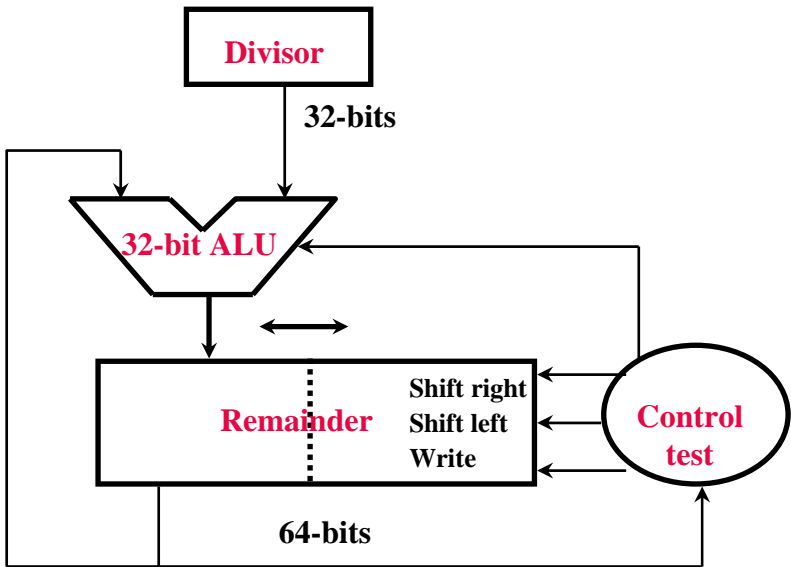
Final sign correction subtraction

Division

| | | | |
|----------------|------|--|------------------------------------|
| Divisor | 1000 | $\begin{array}{r} 101 \\ \hline 1000 \overline{) 101010} \\ - 1000 \\ \hline 10 \\ 101 \\ 1010 \\ - 1000 \\ \hline 10 \end{array}$ | Quotient Dividend |
| | | | Remainder |

- For each step, create a quotient Q_i where
 - $Q_i = 0$ if remainder $<$ divisor
 - $Q_i = 1$ if remainder \geq divisor
- Dividend = Remainder - $Q_i \times$ Divisor
- 3 versions of divider based on successive refinements (see figs 3.11 – 3.13 in your textbook)
- Version 3
 - 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

Divider : version 3



Division: Non-restoring algorithms

- Version 3 is based on a restoring algorithm
- Main functions of each divide step are **“subtract; shift remainder; repeat”**
- Same as partial remainder - $2^{-i} * \text{Divisor}$
 - If result_positive
 - quotient bit = 1
 - else
 - quotient bit = 0 ; restore remainder (addition)
- **Modified algorithm**
 - If result_positive
 - quotient bit = 1
 - else
 - quotient bit = 0
 - In the next step do (remainder + divisor * 2^{-1}) instead of ($- 2^{-1} * \text{Divisor}$)
- The **sign of the remainder determines what to do next**
 - Positive : subtraction;** **Negative : addition**
 - Quotient bit = (remainder ≥ 0)
- Advantage : No restoring (addition) delays

Non-restoring Divider (Cont'd)

- **Same Hardware as add/shift Multiplier with modifications**
 - ALU needs to add or subtract
 - 63-bit register to shift left and shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- **Signed Divides:**
 - Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
 - For the result
 - Dividend and Remainder must have same sign
 - Quotient negated if Divisor sign & Dividend sign disagree

Iterative Divider

- **Use fast multiplier** for division which is usually available
- A factor **F_i** is generated during each iteration and is used to **multiply both the divisor V and dividend D** . F_i is so chosen that the sequence

$$V \times F_0 \times F_1 \times F_2 \dots \quad \text{converges rapidly towards 1}$$

- This implies that $D \times F_0 \times F_1 \times F_2 \dots$ also converges rapidly towards the desired quotient Q since
$$Q = \frac{D \times F_0 \times F_1 \times F_2 \dots}{V \times F_0 \times F_1 \times F_2 \dots}$$

If the denominator converges towards 1, the numerator must also converge towards Q .

- Algorithm **relies on good starting value** and selection of F_i
- Based on numerical techniques to find the next trial value
- **Need 4 to 5 iterations** for reasonable accuracy (number of trials depends on the required accuracy)

The Harvard method of Division (based on iterative method)

$$\begin{aligned}\frac{A}{B} &= \frac{p}{1 - \varepsilon} \\ &= \frac{p \cdot (1 + \varepsilon)}{(1 - \varepsilon) \cdot (1 + \varepsilon)} \\ &= \frac{p'}{1 - \varepsilon'} \quad \text{where} \quad \varepsilon' = \varepsilon^2\end{aligned}$$

- **Quadratic convergence**
- **Each iteration doubles the number of good digits;**
- **Costs 2 multiplies (in parallel)**
- **Use table look-up to guess initial value of (1/B)**

Table Look-up for Arithmetic

- $C = A \text{ op } B$
- $C = F(A, B) = M(A, B)$ through pre-compute, pre-store

- But **storage cost can be staggering**

(A, B) as address : $2n$ bits $\Rightarrow 2^{2n}$ words

| | | | | | | |
|------------|---|----|-----|----------|----------|-------|
| $n =$ | 1 | 2 | 4 | 8 | 16 | |
| $2^{2n} =$ | 4 | 16 | 256 | 2^{16} | 2^{32} | words |

- **Argument reduction**

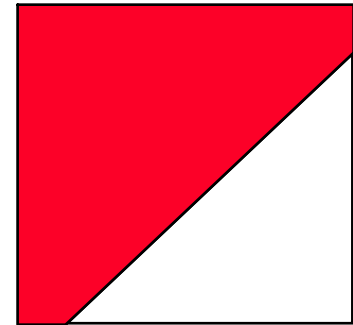
Commutative functions : $(A \text{ op } B) = (B \text{ op } A)$

$$C = M(P, Q) \quad P = \min(A, B); \quad Q = \max(A, B)$$

Because Q is always $> P$; the square matrix
is reduced to a near triangular matrix.

Saving : almost 0.5

Total : $\sim 2^{2n-1}$ bits



Single argument functions

C = F (B); C = M (B)

e.g. \sqrt{B} , B^2 *etc.*

Possible to reduce a multiplication into a single square look-up table

$$A \cdot B = \frac{1}{4} \cdot [(A + B)^2 - (A - B)^2]$$

Cost is still very high 2^n word.

Recap :

- **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).**

Deriving requirements of ALU

- **Start with instruction set architecture: must be able to do all the operations in ISA**
- **Trade-offs of cost and speed based on frequency of occurrence, hardware budget**

MIPS arithmetic instructions

| <u>Instruction</u> | <u>Example</u> | <u>Meaning</u> | <u>Comments</u> |
|--------------------|-------------------|---|--------------------------------|
| add | add \$1,\$2,\$3 | $\$1 = \$2 + \$3$ | 3 operands; exception possible |
| subtract | sub \$1,\$2,\$3 | $\$1 = \$2 - \$3$ | 3 operands; exception possible |
| add immediate | addi \$1,\$2,100 | $\$1 = \$2 + 100$ | + constant; exception possible |
| add unsigned | addu \$1,\$2,\$3 | $\$1 = \$2 + \$3$ | 3 operands; no exceptions |
| subtract unsigned | subu \$1,\$2,\$3 | $\$1 = \$2 - \$3$ | 3 operands; no exceptions |
| add imm. unsign. | addiu \$1,\$2,100 | $\$1 = \$2 + 100$ | + constant; no exceptions |
| multiply | mult \$2,\$3 | Hi, Lo = $\$2 \times \3 | 64-bit signed product |
| multiply unsigned | multu \$2,\$3 | Hi, Lo = $\$2 \times \3 | 64-bit unsigned product |
| divide | div \$2,\$3 | Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3 | Lo = quotient, Hi = remainder |
| divide unsigned | divu \$2,\$3 | Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3 | Unsigned quotient & remainder |
| Move from Hi | mfhi \$1 | $\$1 = \text{Hi}$ | Used to get copy of Hi |
| Move from Lo | mflo \$1 | $\$1 = \text{Lo}$ | Used to get copy of Lo |

MIPS logical instructions

| <i>Instruction</i> | <i>Example</i> | <i>Meaning</i> | <i>Comment</i> |
|---------------------|-------------------|----------------------------|--------------------------------|
| and | and \$1,\$2,\$3 | $\$1 = \$2 \& \$3$ | 3 reg. operands; Logical AND |
| or | or \$1,\$2,\$3 | $\$1 = \$2 \$3$ | 3 reg. operands; Logical OR |
| xor | xor \$1,\$2,\$3 | $\$1 = \$2 \oplus \$3$ | 3 reg. operands; Logical XOR |
| nor | nor \$1,\$2,\$3 | $\$1 = \sim(\$2 \$3)$ | 3 reg. operands; Logical NOR |
| and immediate | andi \$1,\$2,10 | $\$1 = \$2 \& 10$ | Logical AND reg, constant |
| or immediate | ori \$1,\$2,10 | $\$1 = \$2 10$ | Logical OR reg, constant |
| xor immediate | xori \$1, \$2,10 | $\$1 = \sim\$2 \& \sim 10$ | Logical XOR reg, constant |
| shift left logical | sll \$1,\$2,10 | $\$1 = \$2 \ll 10$ | Shift left by constant |
| shift right logical | srl \$1,\$2,10 | $\$1 = \$2 \gg 10$ | Shift right by constant |
| shift right arithm. | sra \$1,\$2,10 | $\$1 = \$2 \gg 10$ | Shift right (sign extend) |
| shift left logical | sllv \$1,\$2,\$3 | $\$1 = \$2 \ll \$3$ | Shift left by variable |
| shift right logical | srlv \$1,\$2, \$3 | $\$1 = \$2 \gg \$3$ | Shift right by variable |
| shift right arithm. | srav \$1,\$2, \$3 | $\$1 = \$2 \gg \$3$ | Shift right arith. by variable |

Compare and Branch

◦ Compare and Branch

- **BEQ rs, rt, offset** if $R[rs] == R[rt]$ then PC-relative branch
- **Compare to zero and Branch**

MIPS ALU requirements

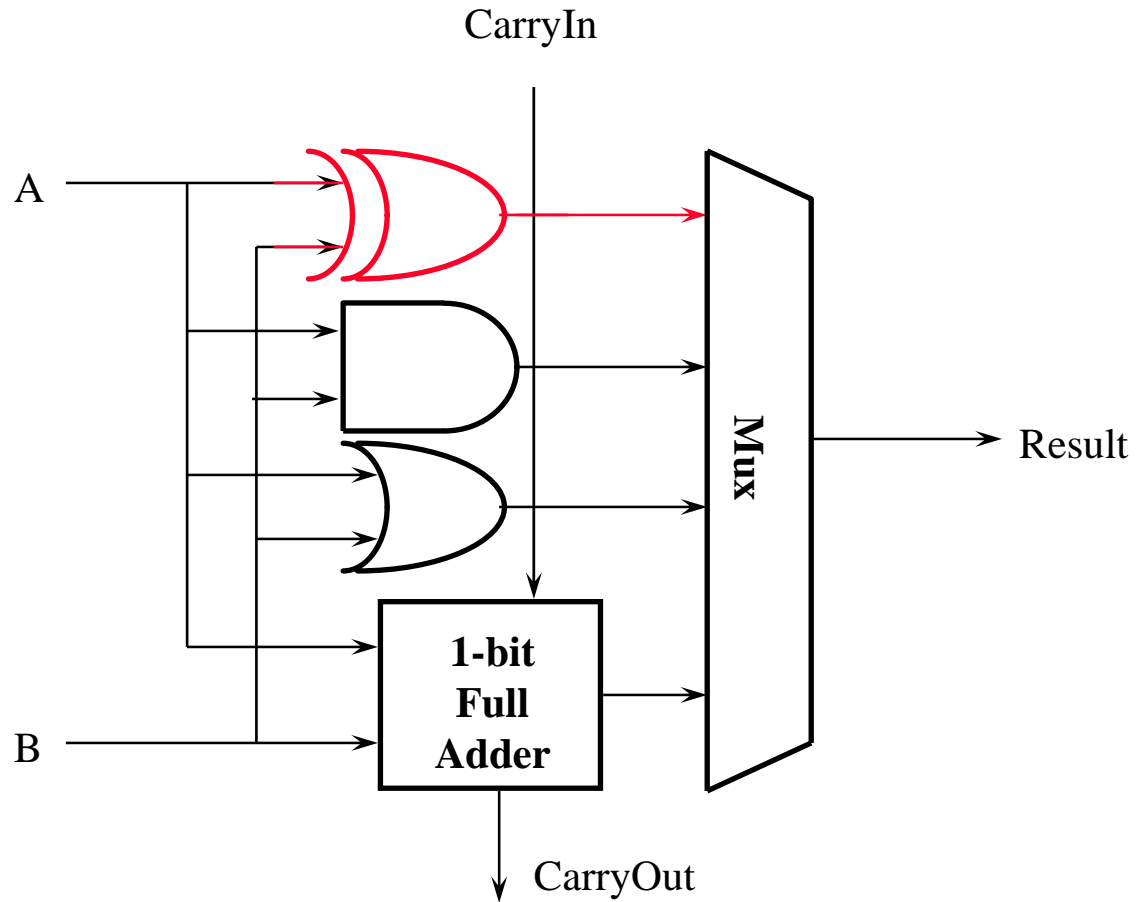
- **Add, AddU, Sub, SubU, Addl, AddIU**
=> 2's complement adder with overflow detection & inverter
- **SLTI, SLTIU (set less than)**
=> 2's complement adder with inverter, check sign bit of result
- **BEQ, BNE (branch on equal or not equal)**
=> 2's complement adder with inverter, check if result = 0
- **And, Or, Andl, Orl**
=> Logical AND, logical OR
- **The ALU presented can support these ops**

Additional MIPS ALU requirements

- **Xor, Nor, Xorl**
=> Logical XOR; logical NOR or use 2 steps: (A OR B) XOR 1111....1111
- **Sll, Srl, Sra**
=> Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- **Mult, MultU, Div, DivU**
=> Need 32-bit multiply and divide, signed and unsigned

Add XOR to ALU

- Expand Multiplexor

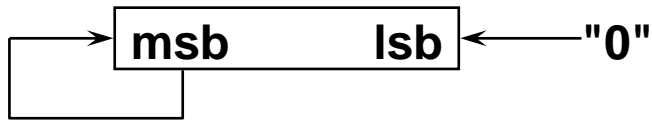


Shifters : (Three different kinds)

Logical -- value shifted in is always "0"



Arithmetic -- on right shifts, sign extend



Rotating -- shifted out bits are wrapped around (not in MIPS)



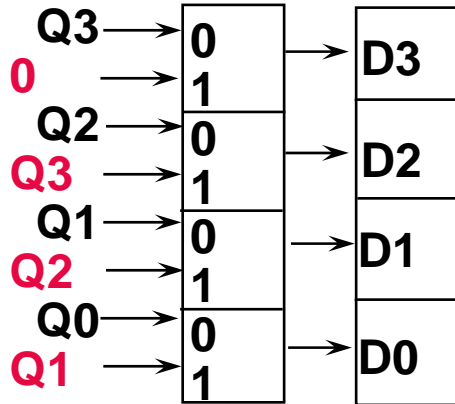
Note : Examples are single bit shifts.

A given instruction might request 0 to 32 bits to be shifted!

Multiplexor/Shifter

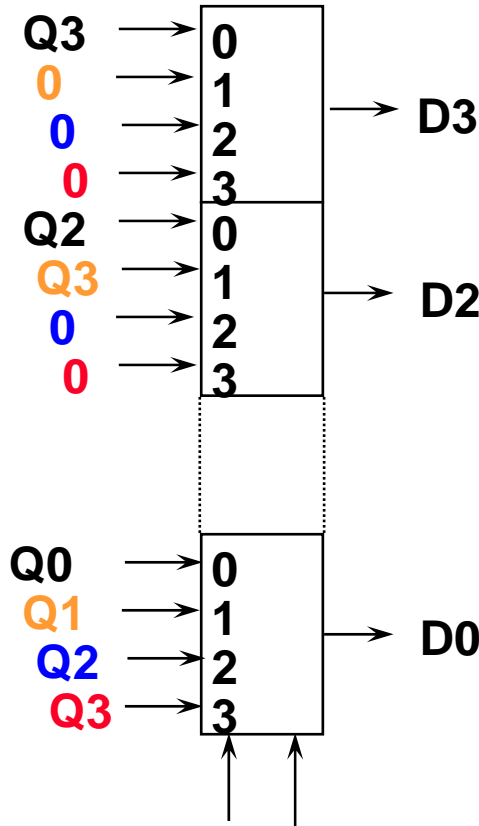
SHR 0, 1, 2, 3 bits:

SHR:



SHR/
don't
shift

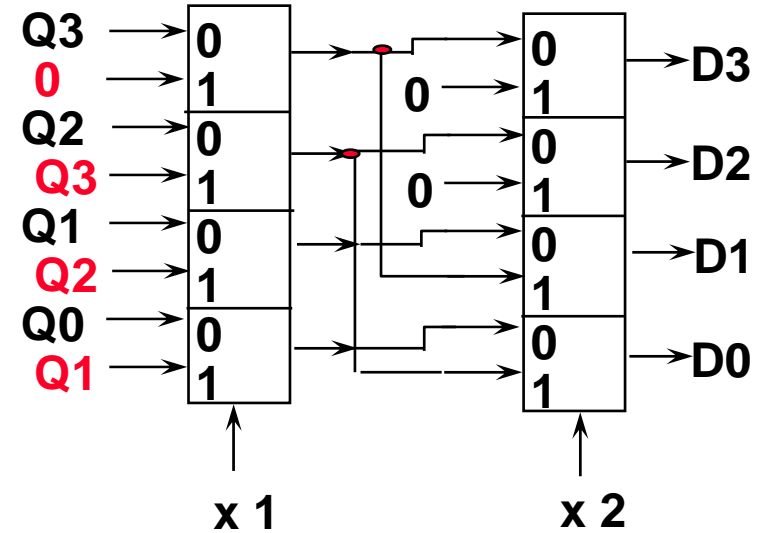
(5 inputs)



shift amount
(0,1,2,3)

4 x 4:1 Mux
1 stage

(7 inputs)

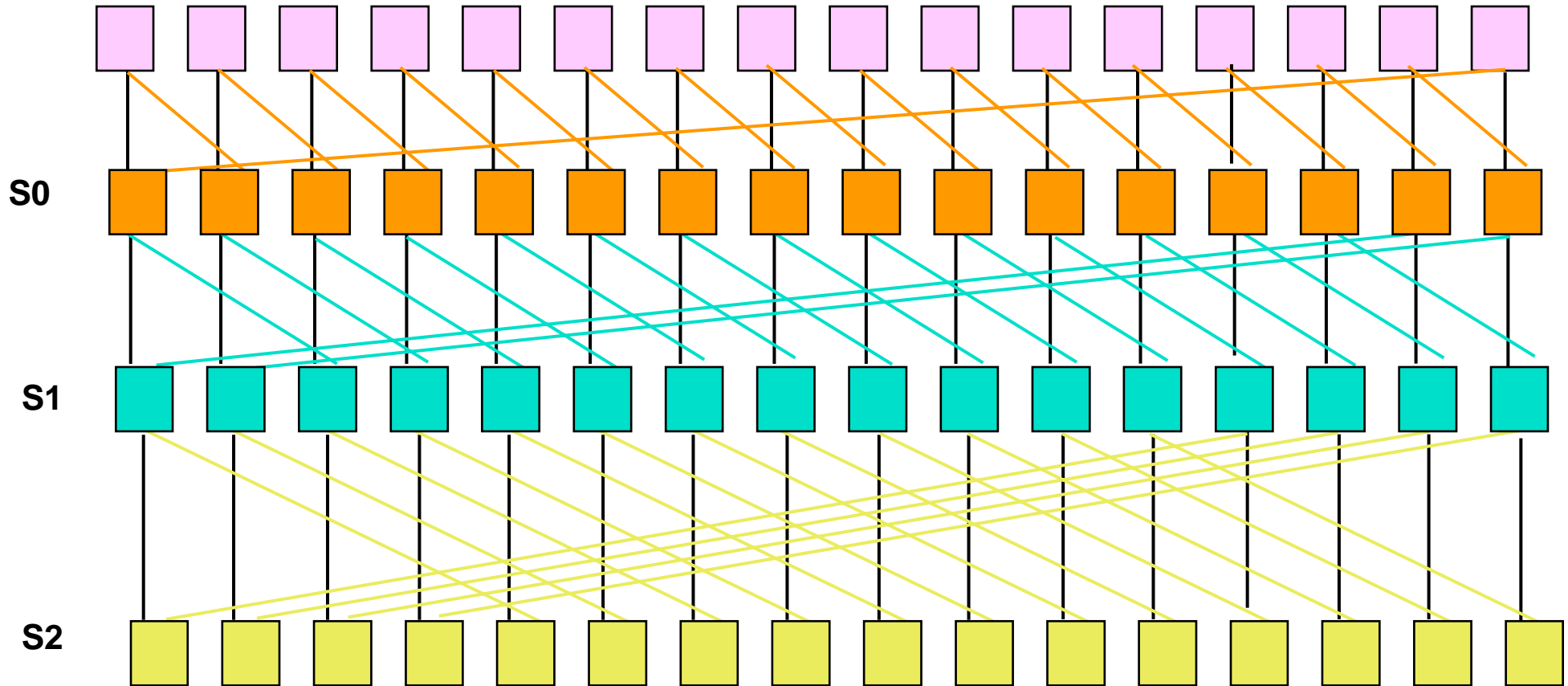


8 x 2:1 Mux
2 stages

How to do arithmetic shift right?

General Scheme

Example is base on 2:1 Multiplexers; Shift amount is $(S2\ S1\ S0)_2$



Right-to-left connections support Rotate (not in MIPS but found in others)

32 Bit Shifter

Using this scheme for 32 bit data with 0-31 bit shifts would result in many possible designs

Can use other types of multiplexers; e.g. 4:1, 8:1 etc

5 stages of (2:1) mux's: (0,1), (0,2), (0,4), (0,8), (0,16)

32 bits x 5 stages = 160 2:1 mux's!

2 stages of (4:1) mux's: (0,1,2,3), (0, 4, 8, 12) and 1 stage of (2:1) mux's: (0, 16)

32 x 4:1 + 32 x 4:1 + 32 x 2:1

1 stage of (8:1) mux's: (0,1,2,3,4,5,6,7) and 1 stage of (4:1) mux's (0, 8, 16, 24)

32 x 8:1 + 32 x 4:1

Multi-bit Shifts (continued)

Mixed strategy, **multiple control loops with more than one bit per loop**

31 bit shift:

31 iterations with a 0,1 position shifter

11 iterations with a 0,1,2,3 position shifter

5 iterations with a 0,1,2,3,4,5,6,7 position shifter

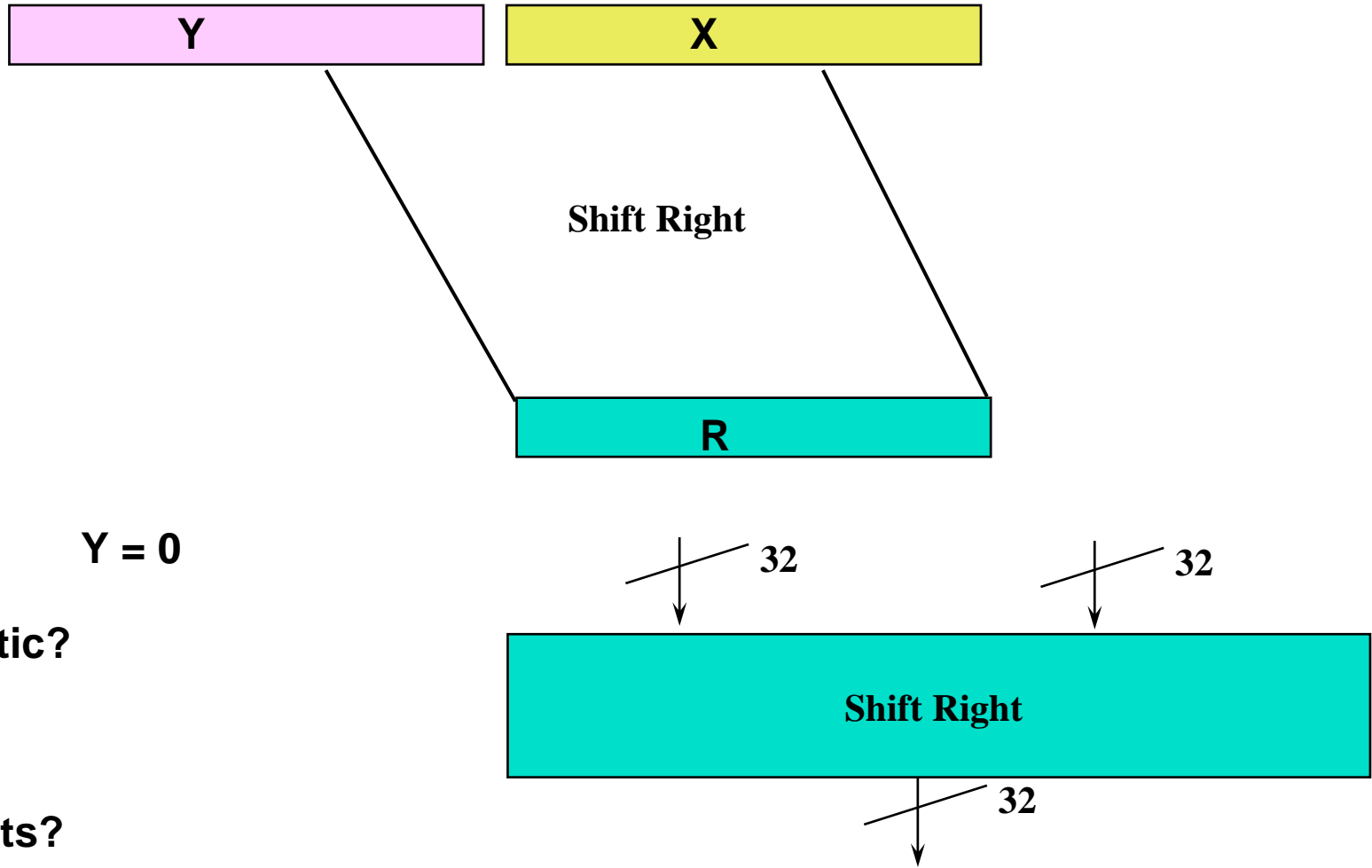
3 iterations with an 0-15 position shifter

Fortunately, **most shifts are relatively short** (0-3 often implemented)

Extra Complexity: can only do shift right so far

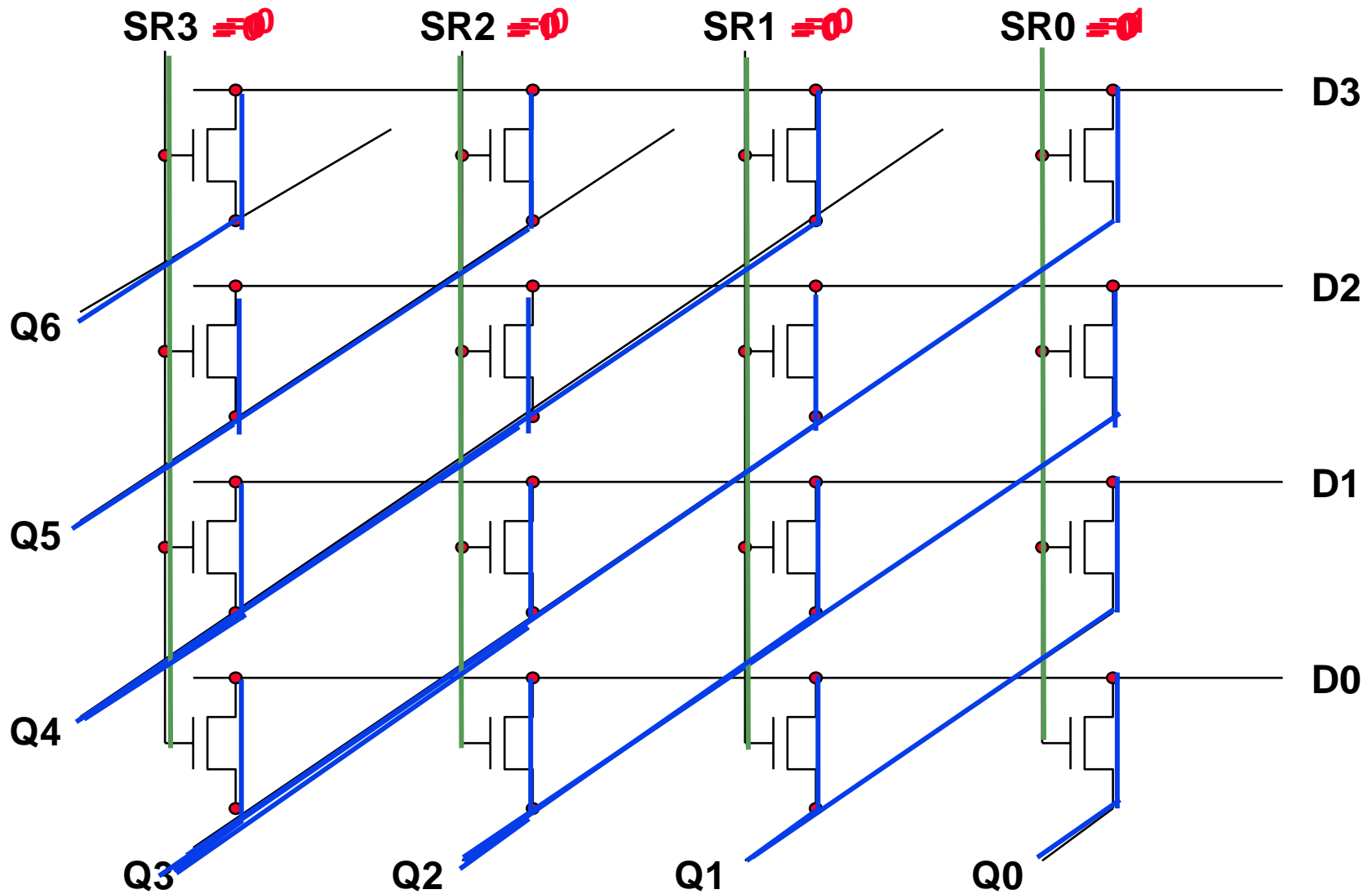
Funnel Shifter

Extract 32 bits of 64.



- Logical: $Y = 0$
- Arithmetic?
- Rotate?
- Left shifts?

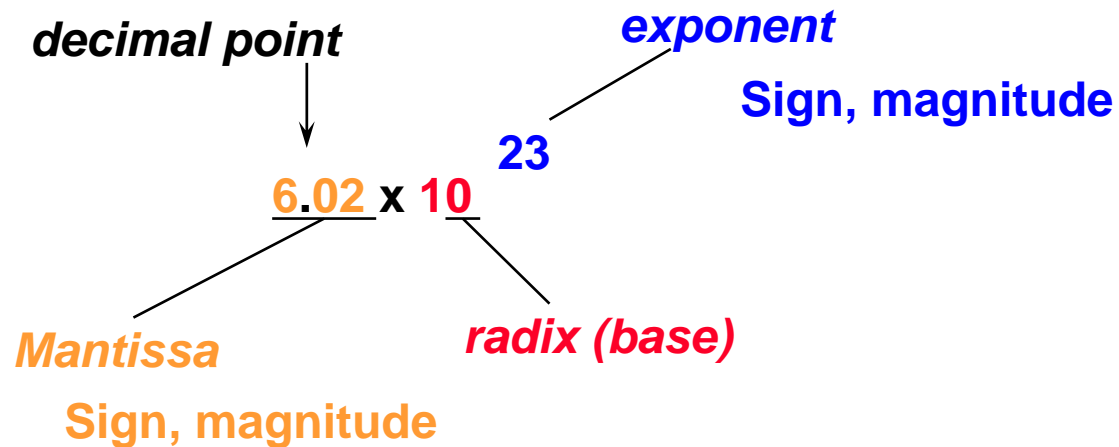
Barrel Shifter: (Technology-dependent solutions)



Floating Point Numbers

- 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

Scientific Notation



- **Representation:**

- sign, exponent, significand: $(-1)^{\text{sign}} * \text{significand} * \text{base}^{\text{exponent}}$
- more bits for significand gives more accuracy
- more bits for exponent increases range

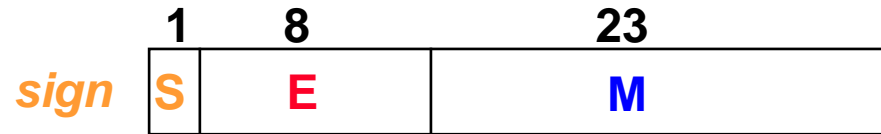
- **IEEE 754 floating point standard:**

- single precision: 8 bit exponent, 23 bit significand
- double precision: 11 bit exponent, 52 bit significand
- Leading “1” bit of significand is implicit
- **Exponent is “biased” to make sorting easier**
 - all 0s is the smallest exponent; all 1s is largest
 - bias of 127 for single precision and 1023 for double precision
 - **summary:** $(-1)^{\text{sign}} * (1 + \text{significand}) * 2^{\text{exponent} - \text{bias}}$

Floating-Point Numbers : Exponents

Representation of floating point numbers in IEEE 754 standard:

single precision



exponent:
excess 127
binary integer

mantissa:
sign + magnitude, normalized
binary significand + hidden
integer bit: 1.M

actual exponent is
 $e = E - 127$

$$N = (-1)^S 2^{E-127} (1.M) \quad 0 < E < 255$$

Magnitude of numbers that can be represented is in the range:

$$2^{-126} (1.0) \quad \text{to} \quad 2^{127} (2 - 2^{-23})$$

which is approximately:

$$1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}$$

(Is integer comparison valid on IEEE FP numbers?)

Floating-Point Numbers : Exponent (cont'd)

$$N = (-1)^S 2^{E-127} (1.M) \quad 0 < E < 255$$

0 = 0 00000000 0 ... 0

Example: -1.5 = ?

S = 1 (negative), 1 of 1.5 is hidden; $0.5_{10} = 0.1_2$; exp = 0, so $E = 127 + 0 = 01111111_2$

Hence -1.5 = 1 01111111 10 ... 0 = bfc00000H

41380000H = ? = 0100 0001 0011 10000

=> S = 0

=> E = 1000010 => e = 1000010 - 01111111 = 11

=> M = 1.0111

Exercise : find the 32-bit FP representations for 0.375, 32767 and for 0.1
find the decimal equivalent for bf300000H

Normalised Numbers

Motivations:

1. Unique representation
e.g. $0.00111 * 2^6 = 0.111 * 2^4$;
try to avoid two representations of the same number
2. Maximise accuracy

Means:

1. **Significand** : is left adjusted --> **as large as possible**,
2. **Exponent** : is **as small as possible**
(imply by the biggest possible significand)

If **Base = 2**, the significand MSB is always 1 when the significand is left adjusted. So not necessary to store this "hidden" bit in memory.

| | | | |
|---|-----|------|---|
| 0 | 011 | 1.01 | 1 |
|---|-----|------|---|

w/o hidden bit

| | | |
|---|-----|------|
| 0 | 011 | .011 |
|---|-----|------|

w/ hidden bit = improved precision

Normalised Numbers (Cont'd)

| | | | |
|---|-----|------|---|
| 0 | 011 | 1.01 | 1 |
|---|-----|------|---|

without hidden bit

| | | |
|---|-----|------|
| 0 | 011 | .011 |
|---|-----|------|

with hidden bit => improved precision

Within the FPU, the hidden bit will be inserted before the denormalisation step that precedes FP add/subtract.

Smallest normal #: 0 0 ... 0 ^{"1"} 0 ... 01

0 0 ... 0 1 ^{hidden bit} 0 ... 00 no hidden bit

2-bias

Problem: must distinguish from zero!

Other Exponent base

Base = 2^4 (hex base) , p = 3 (hex digits)

| | | | | | |
|---|------|------|---|------|------|
| 0 | 0110 | 0000 | . | 0110 | 1100 |
|---|------|------|---|------|------|

 = 0.6C

denormalised

| | | | | | |
|---|------|------|---|------|------|
| 0 | 0101 | 0110 | . | 1100 | 0000 |
|---|------|------|---|------|------|

 = 6.C0

normalised

Issues:

1. No hidden digit
2. Shifting is faster; 4 digits at a time

Basic Addition/subtraction Algorithm

For addition (or subtraction) this translates into the following steps:

- (1) compute $Y_e - X_e$
- (2) right shift X_m that many positions to form $X_m * 2^{X_e - Y_e}$
- (3) compute $X_m * 2^{X_e - Y_e} + Y_m$

If representation demands normalization, then a normalization step follows:

- (4) left shift result, decrement result exponent

right shift result, increment result exponent

continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

If result is 0 mantissa, may need to set the exponent to zero.

Basic Multiplication Algorithm

1. compute $Y_e + X_e - \text{bias}$
2. compute $X_m * Y_m$
3. normalise the result if required

Basic Division Algorithm

1. compute $Y_e - X_e + \text{bias}$
2. compute X_m / Y_m
3. normalise the result if required

Extra Bits (Guard Digits)

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you also pick up a little dirt."

By maintaining some extra bits, we can improve the accuracy.

IEEE standard: computed the result exactly and rounded.

Addition:

$$\begin{array}{r} 1.xxxxx \\ + 1.xxxxx \\ \hline 1x.xxxxxy \end{array}$$

post-normalization

$$\begin{array}{r} 1.xxxxx \\ 0.001xxxxx \\ \hline 1.xxxxxxyyy \end{array}$$

pre-normalization

$$\begin{array}{r} 1.xxxxx \\ 0.01xxxxx \\ \hline 1x.xxxxxyyy \end{array}$$

pre and post

- **Guard Digits:** digits to the right (LS end) of the significand to guard against loss of digits. Guard digits can later be reintroduced by shifting left during normalisation.
- For FP arithmetic:
 - **Addition:** carry-out shifted (right) in
 - **Subtraction:** borrow digit and guard
 - **Multiplication:** carry and guard,
 - **Division:** may also use guard to improve accuracy

Size of Guard Digit

- 1 lump (**Log₂ base bits**) is sufficient for operations between normalised numbers
- Example proof : base 2 normalised numbers
 - biggest fraction (FB) = $0.111 \dots 11 = 1 - 2^{-n}$
 - smallest fraction (FS) = $0.100 \dots 00$

For subtraction :

a. if exponent difference = 0 or 1, one lump is enough to retain the digits during pre-operation alignment shifts.

b. if exponent difference ≥ 2

worst case is **(smallest fraction) - 2^{-2} * (biggest fraction)**

$$= 0.1 - 2^{-2} * (1 - 2^{-n}) = 0.1 - 0.01 + 2^{-(n+2)}$$

$$= 0.01 + 2^{-(n+2)} \geq 0.01$$

During normalisation, only 1 shift is required,

i.e. 1 guard digit is enough

For Addition, 1 carry bit is needed.

Rounding Digits

Rounding is the method to deal with the non-zero digits to the right of the significand during normalisation?

E.g., B = 10 (decimal), p = 3:

| | | | | |
|---|---|------|----------------|--------|
| 0 | 2 | 1.69 | = 1.6900 * 10 | 2-bias |
| 1 | 0 | 7.85 | = - .0785 * 10 | 2-bias |
| 0 | 2 | 1.61 | = 1.6115 * 10 | 2-bias |

1 round digit must be carried to the right of the guard digit so that after a normalising left shift, the result can be rounded, according to the value of the round digit.

IEEE Standard:

four rounding modes: round to nearest (default)
round towards plus infinity
round towards minus infinity (chopping)
round towards 0

round to nearest: (plus half rounding)

round digit < B/2 then truncate

> B/2 then round up (add 1 to LS digit)

= B/2 then round to nearest even digit (why?)

It can be shown that this strategy minimises the mean error introduced by rounding

Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding

$$\begin{array}{r}
 d_0 . d_1 d_2 d_3 \dots d_{p-1} \ 0 \ 0 \ 0 \\
 + \ 0 . \ 0 \ 0 \ X \dots X \ X \ X \ S \\
 \hline
 \phantom{d_0 . d_1 d_2 d_3 \dots d_{p-1} \ 0 \ 0 \ 0} \ X \ X \ S
 \end{array}$$

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

$$\begin{array}{r}
 d_0 . d_1 d_2 d_3 \dots d_{p-1} \ 0 \ 0 \ 0 \\
 - \ 0 . \ 0 \ 0 \ X \dots X \ X \ X \ 0 \\
 \hline
 \phantom{d_0 . d_1 d_2 d_3 \dots d_{p-1} \ 0 \ 0 \ 0} \ X \ X \ 0
 \end{array}$$

$$\begin{array}{r}
 d_0 . d_1 d_2 d_3 \dots d_{p-1} \ 0 \ 0 \ 0 \\
 - \ 0 . \ 0 \ 0 \ X \dots X \ X \ X \ 1 \\
 \hline
 \phantom{d_0 . d_1 d_2 d_3 \dots d_{p-1} \ 0 \ 0 \ 0}
 \end{array}$$

generates a borrow

Rounding Summary:

Radix 2 minimises wobble in precision

Normal operations in (+, -, *, /) require one carry/borrow bit + one guard digit

One round digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed

May need to consider both average and maximum errors

Infinity and NaNs

Result of operation can *overflow*, i.e., result is larger than the largest number that can be represented

Overflow is not the same as divide by zero (raises a different exception)

+/- infinity

| | | |
|---|-----------|-----------|
| S | 1 . . . 1 | 0 . . . 0 |
|---|-----------|-----------|

It may make sense to continue computations with infinity
e.g., $X/0 > Y$ may be a valid comparison

Not a number, but not infinity (e.g. $\text{sqrt}(-4)$)
invalid operation exception (unless operation is = or $\lt \gt$)

NaN

| | | |
|---|-----------|----------|
| S | 1 . . . 1 | non-zero |
|---|-----------|----------|

 ← HW decides what goes here

NaNs propagate: $f(\text{NaN}) = \text{NaN}$

Exceptions

Invalid operation:

result of operation is a NaN (except = or <>)

inf. +/- inf. ; $0 * \text{inf}$; $0/0$; inf./inf. ; x remainder y where $y = 0$;

$\text{sqrt}(x)$ where $x < 0$, $x = +/- \text{inf.}$

Overflow:

result of operation is larger than largest representable number

flushed to +/- inf. if overflow exception is not enabled

Divide by 0:

$x/0$ where $x = 0$, +/- inf.;

flushed to +/- inf. if divide by zero exception not enabled

Underflow:

subnormal result OR non-zero result underflows to 0

Inexact:

rounded result not the actual result (rounding error = 0)

IEEE Standard:

Specifies defaults and allows traps to permit user to handle the exception ;

Contrast with the more usual result of aborting the computation altogether!

Pentium Bug (1995)

- Pentium FP Divider uses algorithm to generate multiple bits per steps
 - FPU uses most significant bits of divisor & dividend/remainder to guess next 2 bits of quotient
 - Guess is taken from lookup table: -2, -1,0,+1,+2 (if previous guess too large a reminder, quotient is adjusted in subsequent pass of -2)
 - Guess is multiplied by divisor and subtracted from remainder to generate a new remainder
 - Called SRT division after 3 people who came up with idea
- Pentium table uses 7 bits of remainder + 4 bits of divisor = 2^{11} entries
- 5 entries of divisors omitted: 1.0001, 1.0100, 1.0111, 1.1010, 1.1101 from PLA (fix is just add 5 entries back into PLA: cost \$200,000)
- Self correcting nature of SRT => string of 1s must follow error
 - e.g., 1011 1111 1111 1111 1111 1011 1000 0010 0011 0111 1011 0100
(2.99999892918)
- Since indexed also by divisor/remainder bits, sometimes bug doesn't show even with dangerous divisor value

Pentium bug appearance

- First 11 bits to right of decimal point always correct: bits 12 to 52 where bug can occur (4th to 15th decimal digits)
- FP divisors near integers 3, 9, 15, 21, 27 are dangerous ones:
 - $3.0 > d \approx 3.0 - 36 \times 2^{-22}$, $9.0 > d \approx 9.0 - 36 \times 2^{-20}$
 - $15.0 > d \approx 15.0 - 36 \times 2^{-20}$, $21.0 > d \approx 21.0 - 36 \times 2^{-19}$
- 0.333333×9 could be problem
- In Microsoft Excel, try $(4,195,835 / 3,145,727) * 3,145,727$
 - = 4,195,835 => not a Pentium with bug
 - = 4,195,579 => Pentium with bug
(assuming Excel doesn't already have SW bug patch)
 - Rare since error in 5th significant digit

Pentium Bug Time line

- **June 1994: Intel discovers bug in Pentium: takes months to make change, reverify, put into production: plans good chips in January 1995
4 to 5 million Pentiums produced with bug**
- **Scientist suspects errors and posts on Internet in September 1994**
- **Nov. 22 Intel Press release: “Can make errors in 9th digit ... Most engineers and financial analysts need only 4 of 5 digits. Theoretical mathematician should be concerned. ... So far only heard from one.”**
- **Intel claims happens once in 27,000 years for typical spread sheet user:**
 - **1000 divides/day x error rate assuming numbers random**
- **Dec 12: IBM claims happens once per 24 days: Bans Pentium sales**
 - **5000 divides/second x 15 minutes = 4.2 million divides/day**
 - **IBM statement: <http://www.ibm.com/Features/pentium.html>**
 - **Intel said it regards IBM's decision to halt shipments of its Pentium processor-based systems as unwarranted.**

Pentium conclusion: Dec. 21, 1994 \$500M write-off

“To owners of Pentium processor-based computers and the PC community: We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw.

The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect.

What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns.

We want to resolve these concerns.

Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer. Just call 1-800-628-8686.”

Sincerely,

**Andrew S. Grove
President /CEO**

**Craig R. Barrett
Executive Vice President
&COO**

**Gordon E. Moore
Chairman of the Board**

Summary

- **Bits have no inherent meaning: operations determine whether they are really ASCII characters, integers, floating point numbers**
- **Divide can use same hardware as multiply: Hi & Lo registers in MIPS**
- **Floating point basically follows paper and pencil method of scientific notation using integer algorithms for multiply and divide of significands**
- **IEEE 754 requires good rounding; special values for NaN, Infinity**
- **Pentium: Difference between bugs that board designers must know about and bugs that potentially affect all users**
 - **Why not make public complete description of bugs in later category?**
 - **\$200,000 cost in June to repair design**
 - **\$500,000,000 loss in December in profits to replace bad parts**
 - **How much to repair Intel's reputation?**
- **What is technologists responsibility in disclosing bugs?**