

CS2006: 計算機組織

Computer Arithmetic



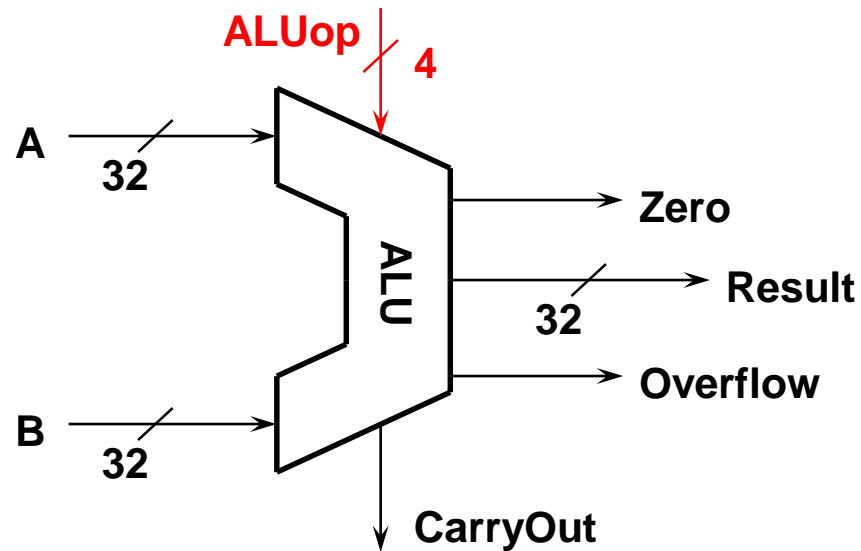
Outline

- ◆ Constructing an arithmetic logic unit (Appendix C)
- ◆ Multiplication (Sec. 3.3, Appendix C)
- ◆ Division (Sec. 3.4)
- ◆ Floating point (Sec. 3.5)

Problem: Designing MIPS ALU

- ◆ Requirements: must support the following arithmetic and logic operations
 - **add, sub**: two's complement adder/subtractor with overflow detection
 - **and, or, nor** : logical AND, logical OR, logical NOR
 - **slt** (set on less than): two's complement adder with inverter, check sign bit of result

Functional Specification



ALU Control (ALUop)

Function

0000

and

0001

or

0010

add

0110

subtract

0111

set-on-less-than

1100

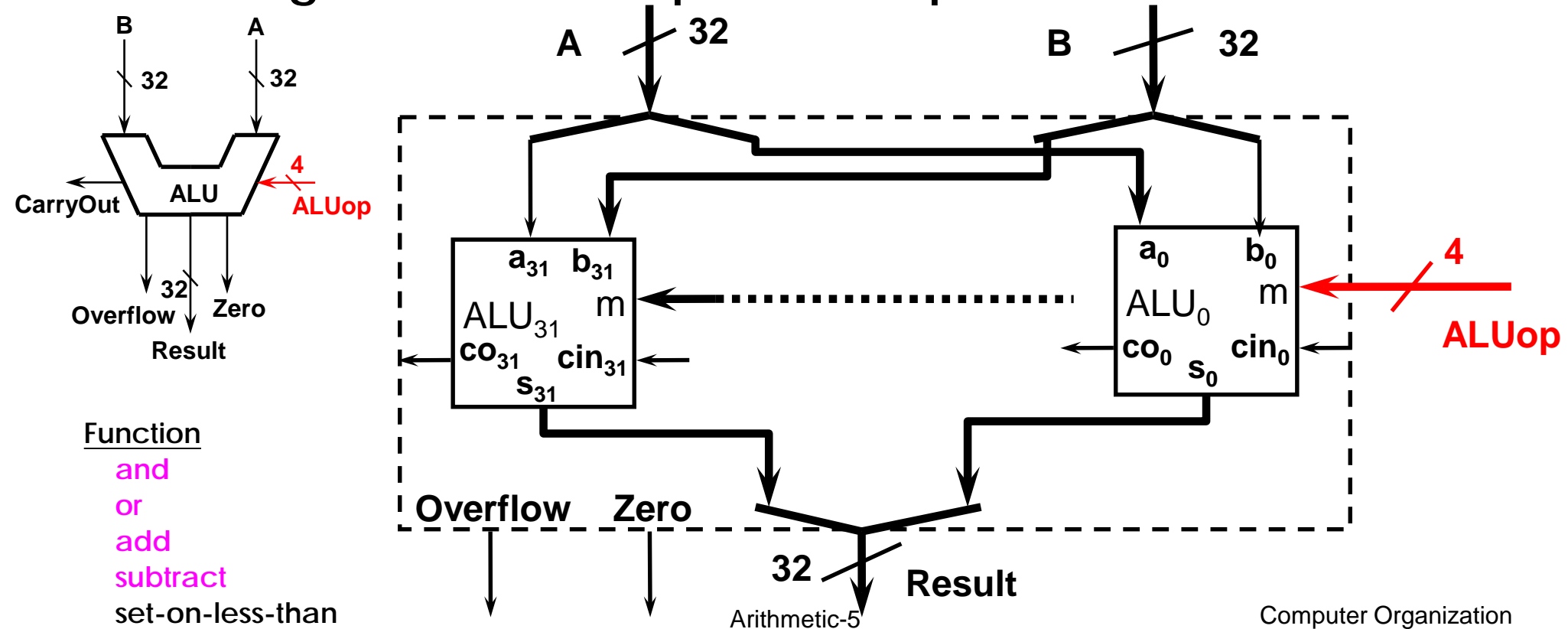
nor

A Bit-slice ALU

◆ Design trick 1: divide and conquer

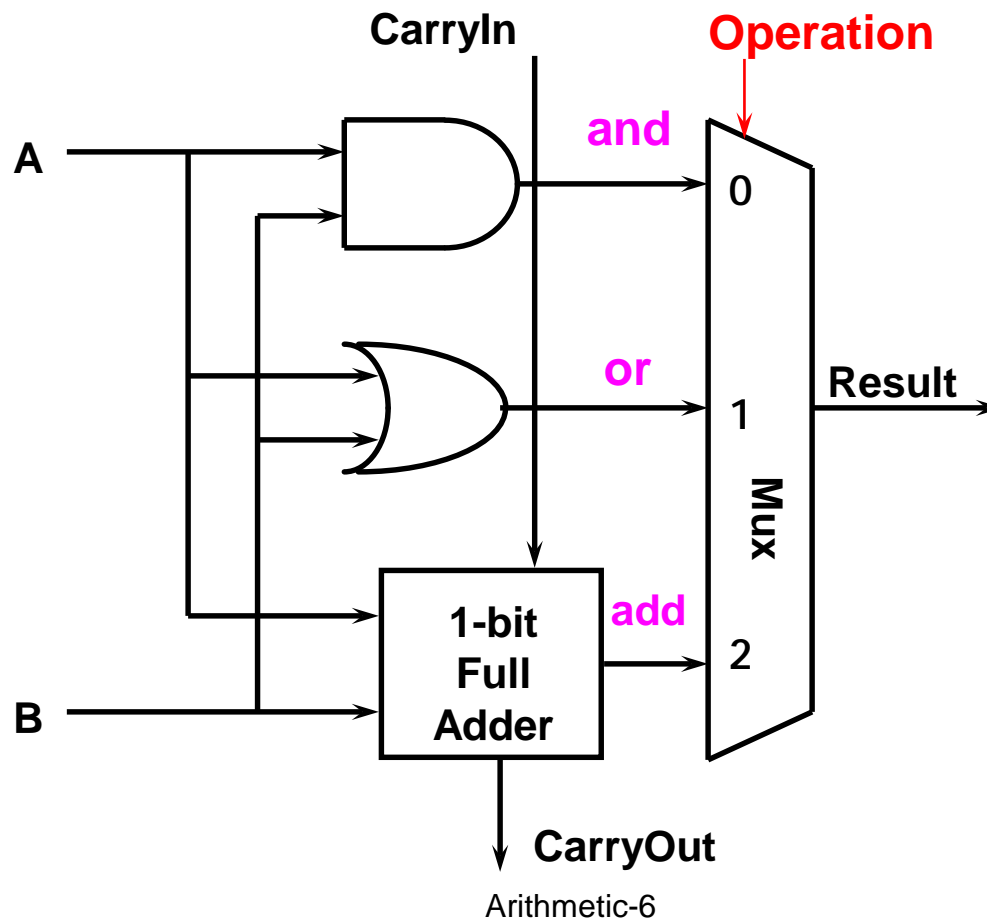
- Break the problem into simpler problems, solve them and glue together the solution

◆ Design trick 2: solve part of the problem and extend



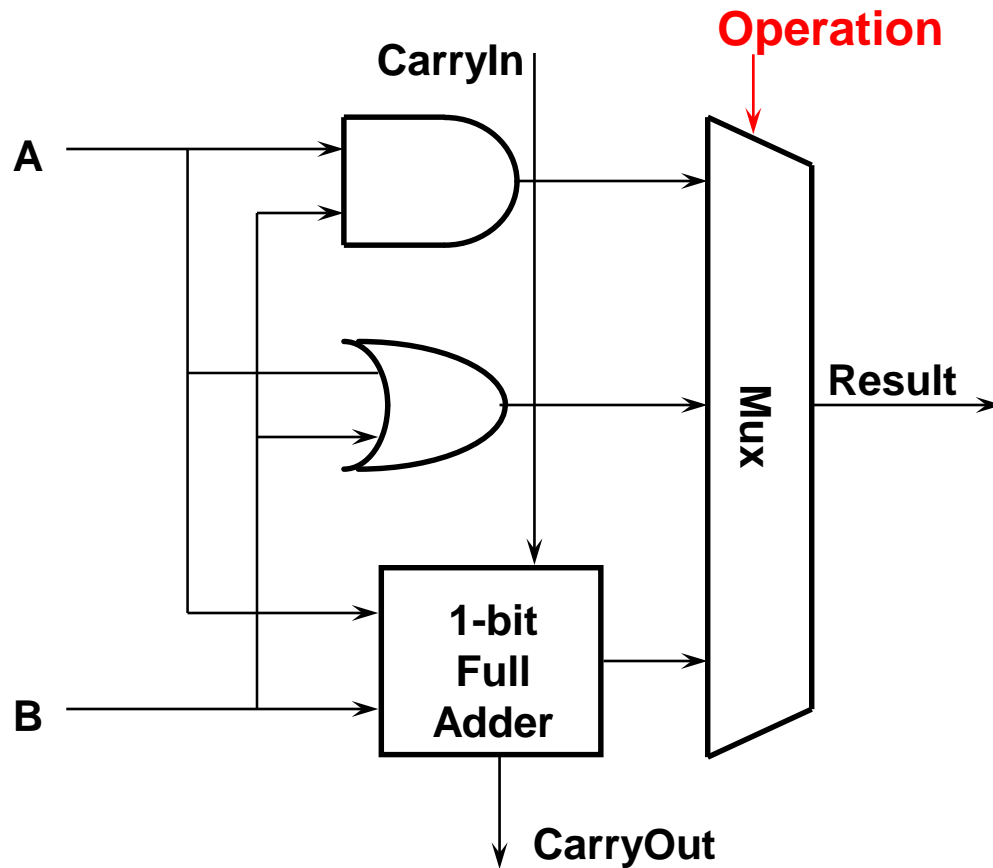
A 1-bit ALU

- ◆ Design trick 3: take pieces you know (or can imagine) and try to put them together

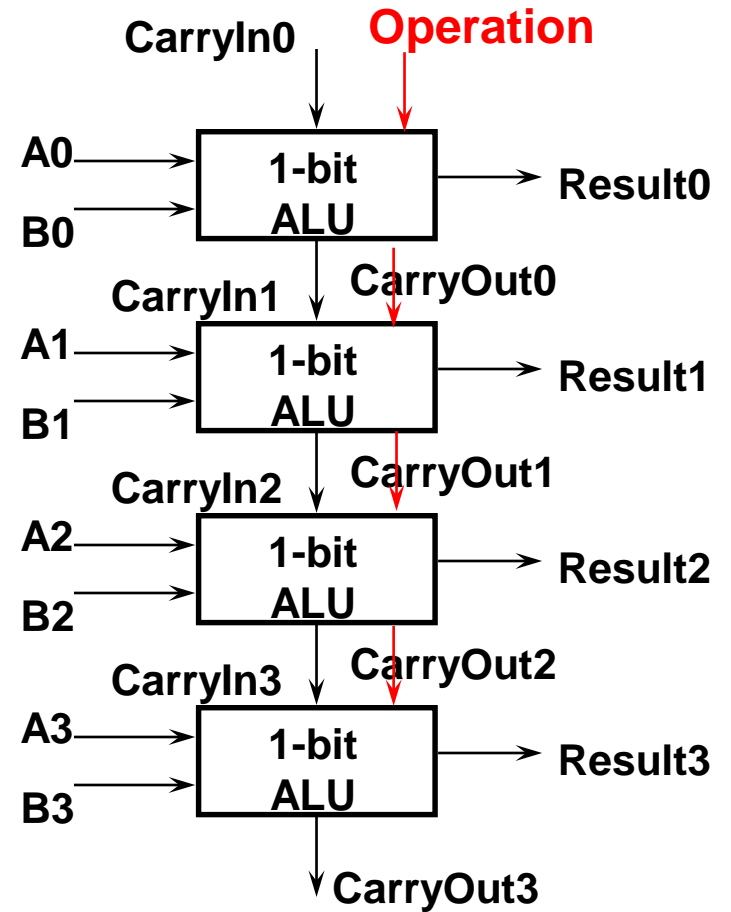


A 4-bit ALU

1-bit ALU

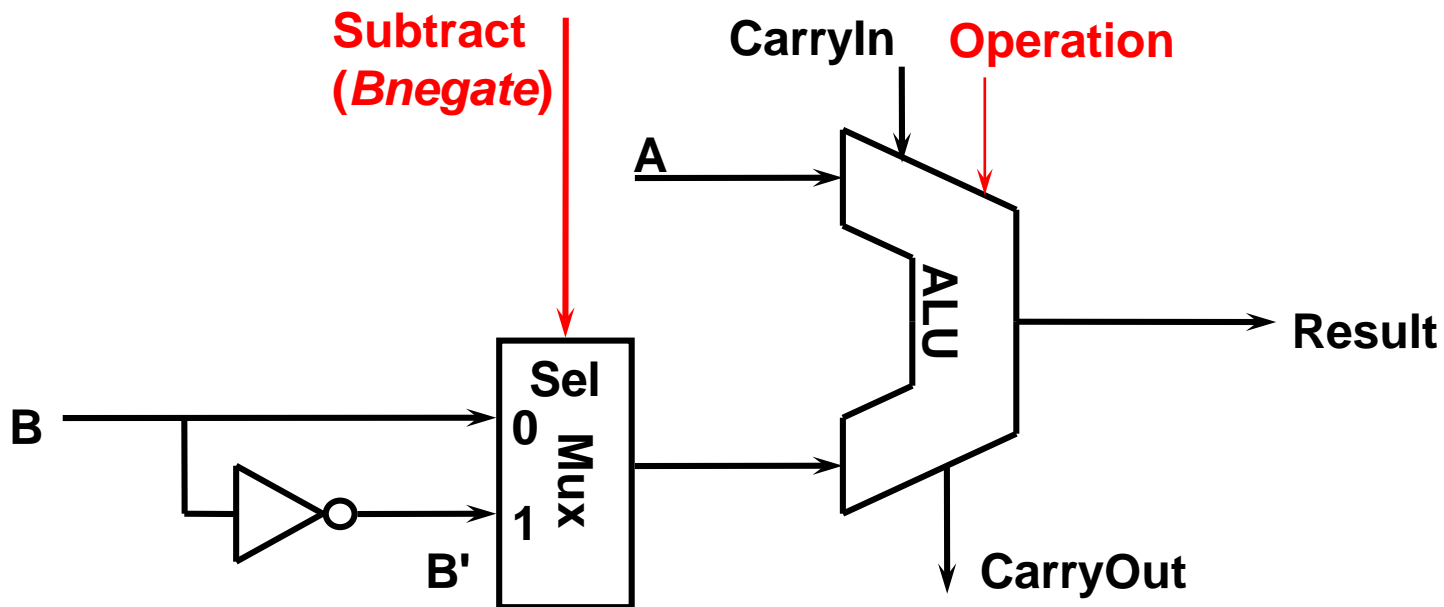


4-bit ALU

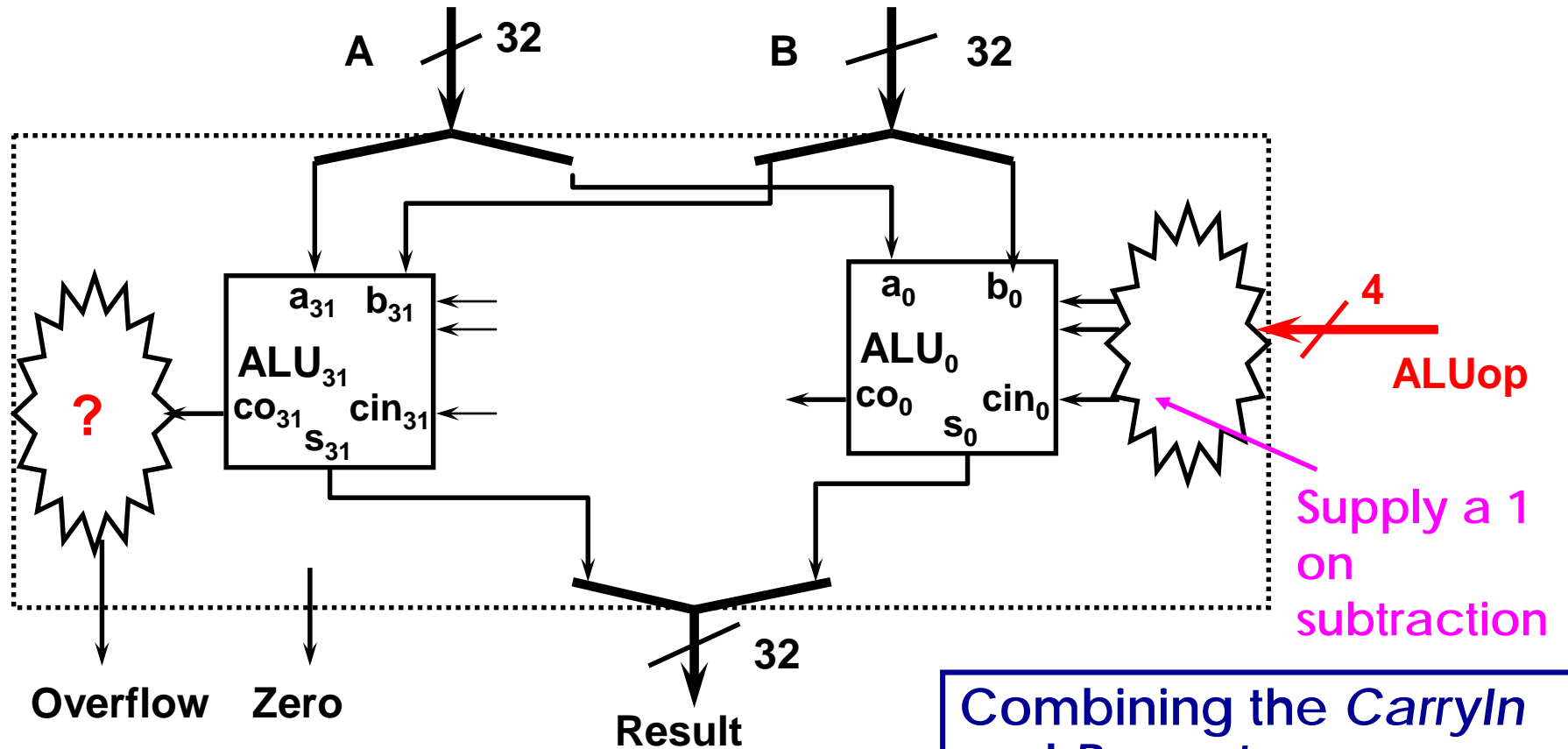


How about Subtraction?

- ◆ 2's complement: take inverse of every bit and add 1 (at c_{in} of first stage)
 - $A + B' + 1 = A + (B' + 1) = A + (-B) = A - B$
 - Bitwise inverse of B is B'

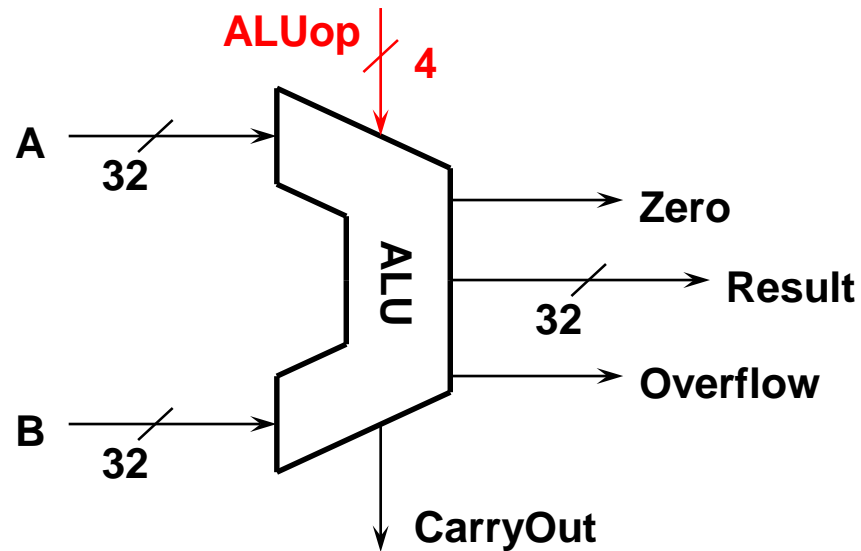


- ◆ **LSB and MSB need to do a little extra**



Combining the *CarryIn* and *Bnegate*

Functional Specification



ALU Control (ALUop)

0000

0001

0010

0110

0111

1100

Function

and

or

add

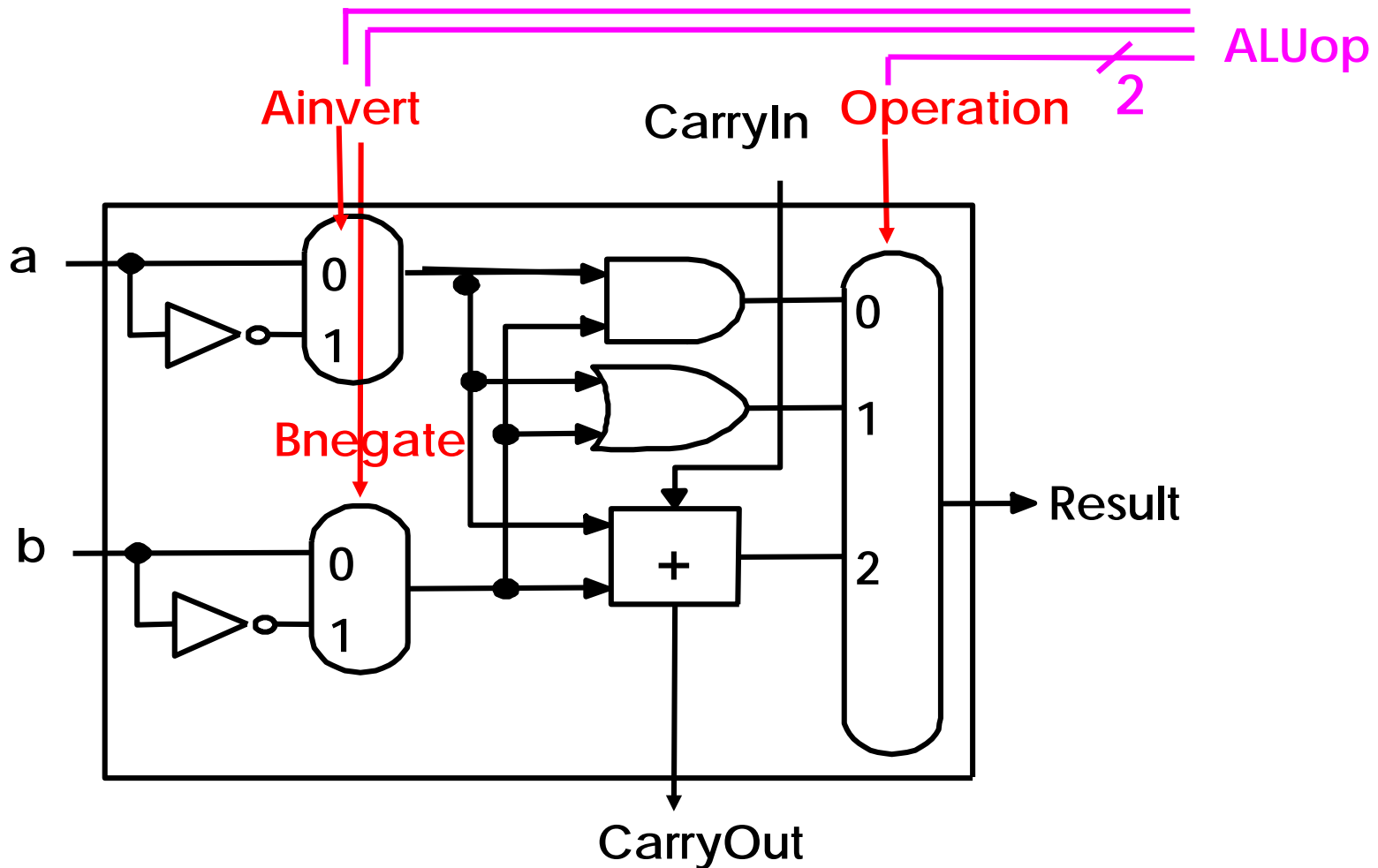
subtract

set-on-less-than

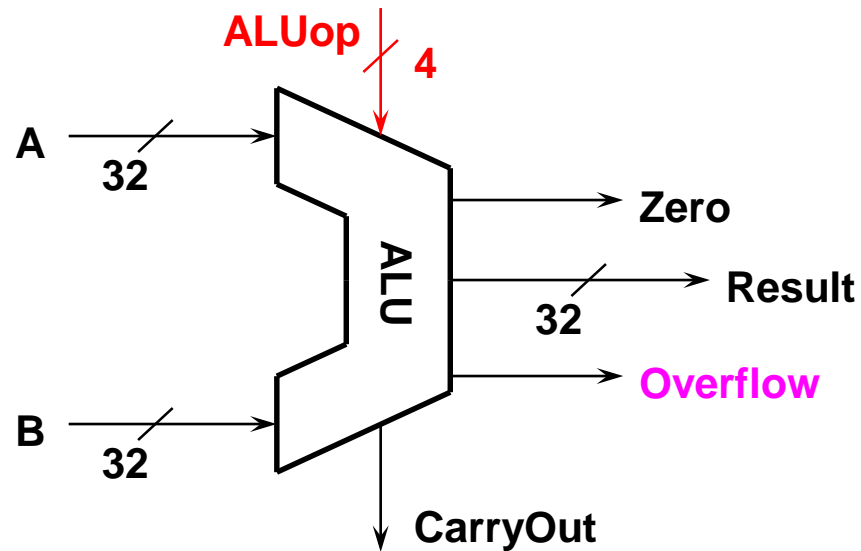
nor

Nor Operation

- ◆ $A \text{ nor } B = (\text{not } A) \text{ and } (\text{not } B)$



Functional Specification



ALU Control (ALUOp)

0000

0001

0010

0110

0111

1100

Function

and

or

add

subtract

set-on-less-than

nor

Overflow

Decimal	Binary	Decimal	2's complement
0	0000	0	0000
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
		-8	1000

Ex: $7 + 3 = 10$ but ...

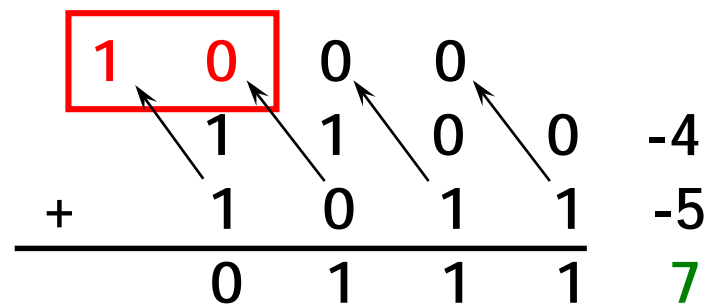
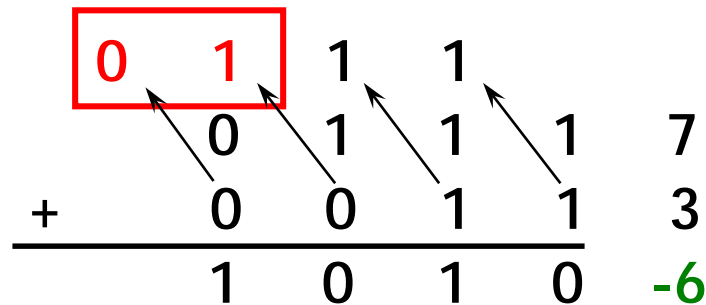
$$\begin{array}{r}
 \begin{array}{ccccccc}
 0 & 1 & 1 & 1 & & & \\
 \swarrow & \swarrow & \swarrow & \swarrow & & & \\
 0 & 0 & 1 & 1 & 1 & 1 & 7 \\
 + & 0 & 0 & 1 & 1 & & 3 \\
 \hline
 & 1 & 0 & 1 & 0 & & -6
 \end{array}
 \end{array}$$

$-4 - 5 = -9$ but ...

$$\begin{array}{r}
 \begin{array}{ccccccc}
 1 & 0 & 0 & 0 & & & \\
 \swarrow & \swarrow & \swarrow & \swarrow & & & \\
 1 & 1 & 1 & 0 & 0 & 0 & -4 \\
 + & 1 & 0 & 1 & 1 & & -5 \\
 \hline
 & 0 & 1 & 1 & 1 & & 7
 \end{array}
 \end{array}$$

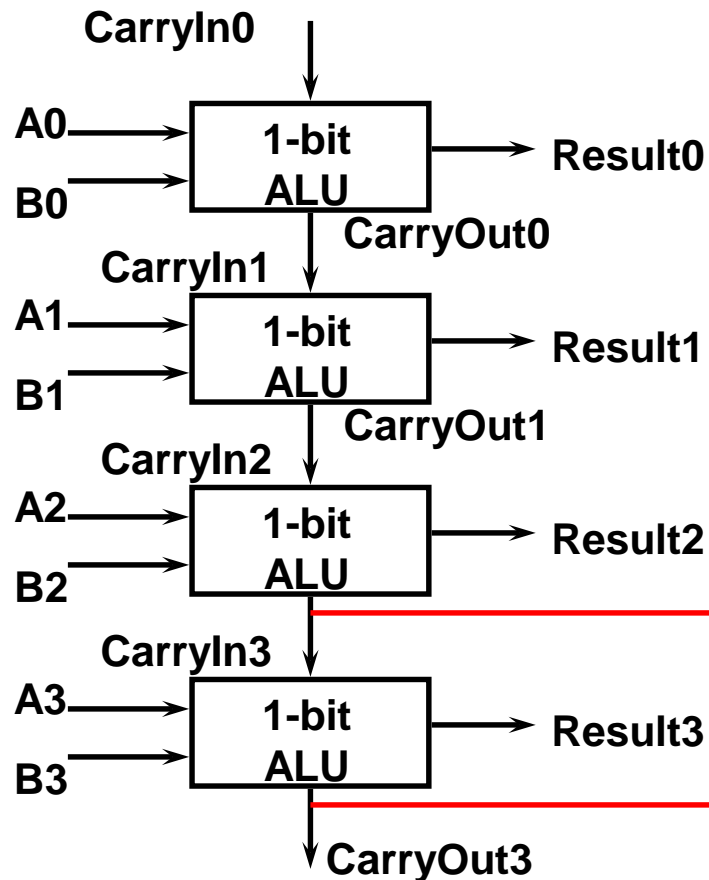
Overflow Detection

- ◆ **Overflow: result too big/small to represent**
 - $-8 \leq 4\text{-bit binary number} \leq 7$
 - When adding operands with different signs, overflow will not occur!
 - Overflow occurs when adding:
 - 2 positive numbers and the sum is negative
 - 2 negative numbers and the sum is positive \Rightarrow sign bit is set with the value of the result
 - Overflow if: **CarryIn of MSB \neq CarryOut of MSB**

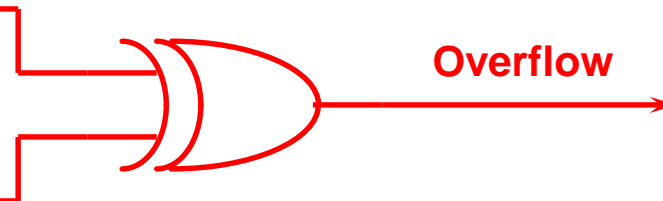


Overflow Detection Logic

- ◆ $\text{Overflow} = \text{CarryIn}[N-1] \text{ XOR } \text{CarryOut}[N-1]$



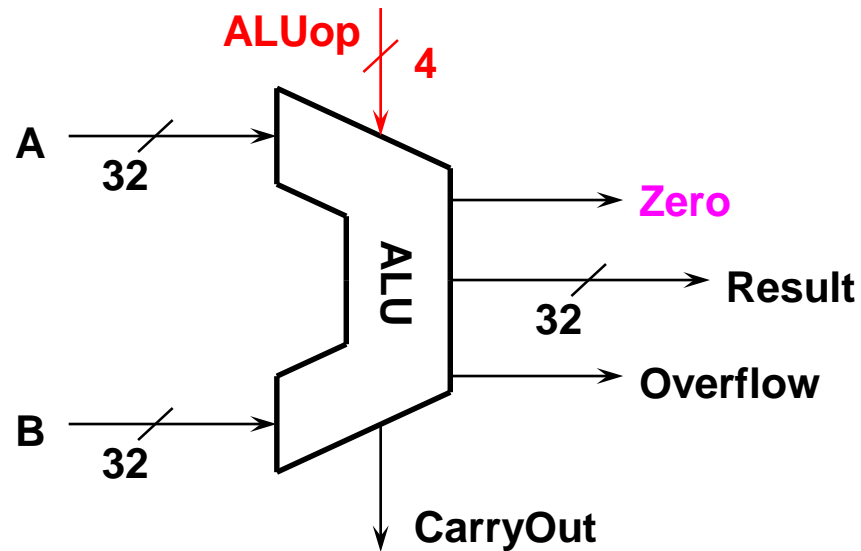
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0



Dealing with Overflow

- ◆ Some languages (ex: C) ignore overflow
 - Use MIPS `addu`, `addui`, `subu` instructions
- ◆ Other languages (ex: Ada, Fortran) require raising an exception
 - Use MIPS `add`, `addi`, `sub` instructions
 - On overflow, invoke exception handler
 - Save PC in exception program counter (EPC) register
 - Jump to predefined handler address
 - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

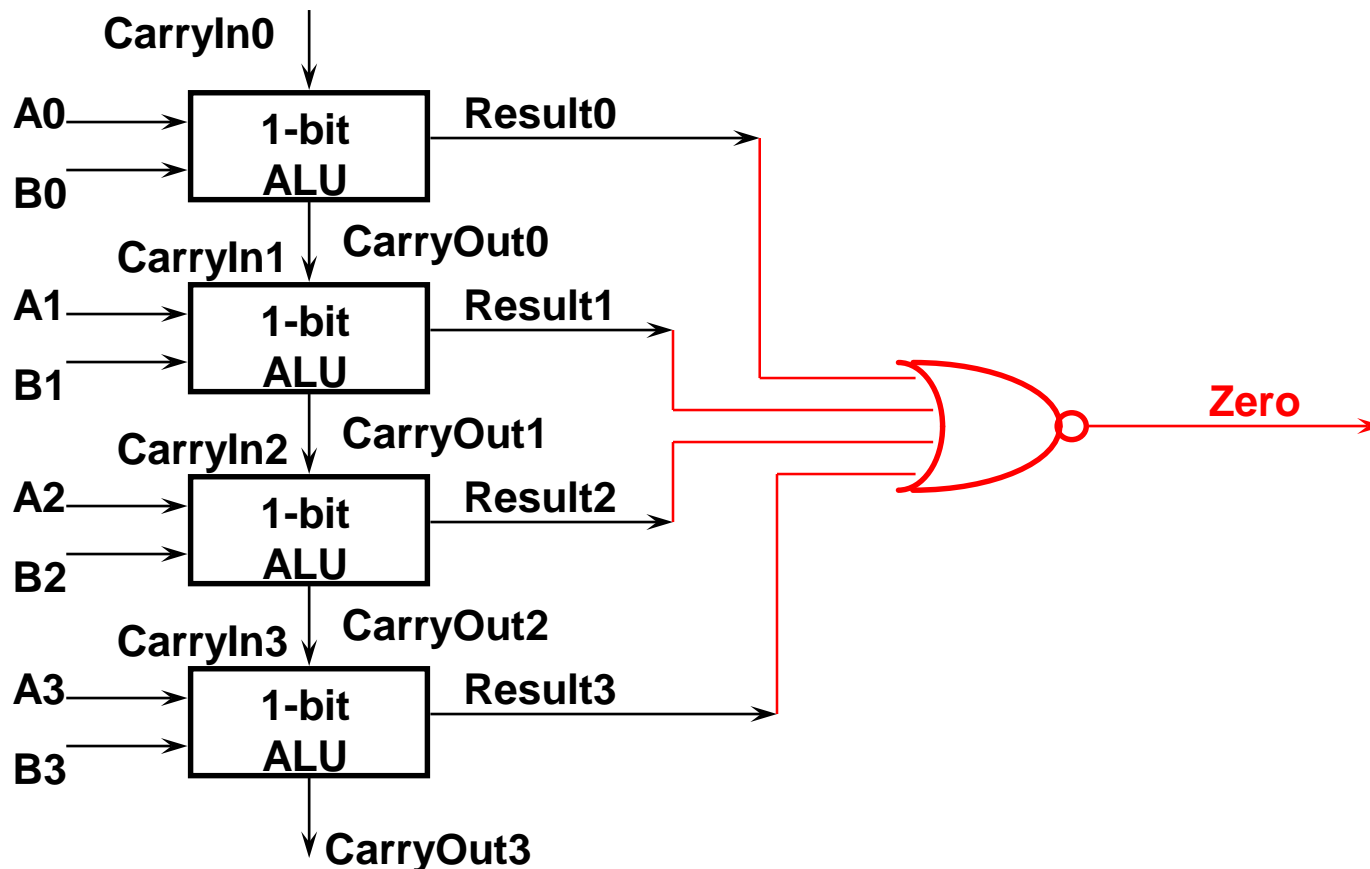
Functional Specification



<u>ALU Control (ALUop)</u>	<u>Function</u>
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

Zero Detection Logic

- Zero Detection Logic is a one BIG NOR gate (support conditional jump)



Functional Specification

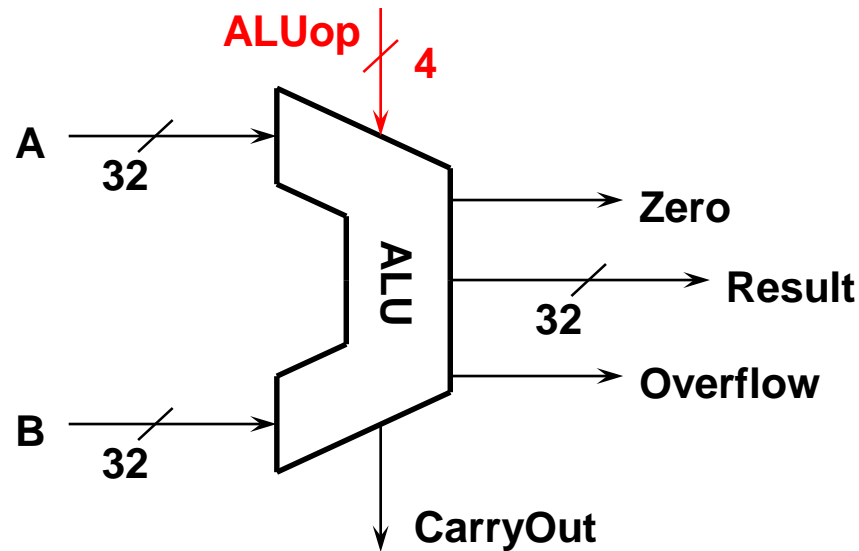


Fig. B.5.14

<u>ALU Control (ALUop)</u>	<u>Function</u>
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

Fig. B.5.13

Set on Less Than (1/3)

◆ 1-bit in ALU

(for bits 1-30)

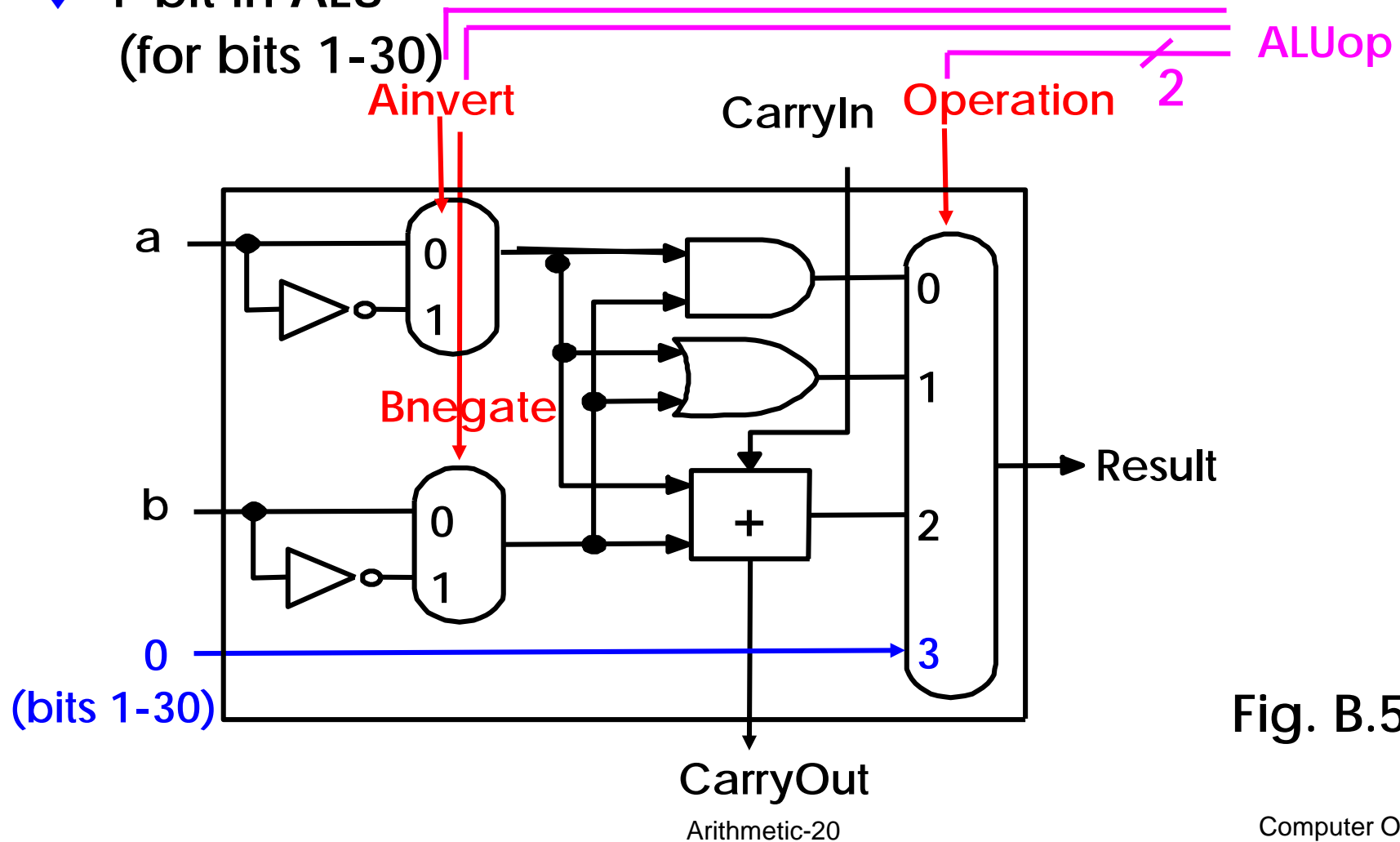


Fig. B.5.10a

Set on Less Than (2/3)

◆ Bit 31 in ALU

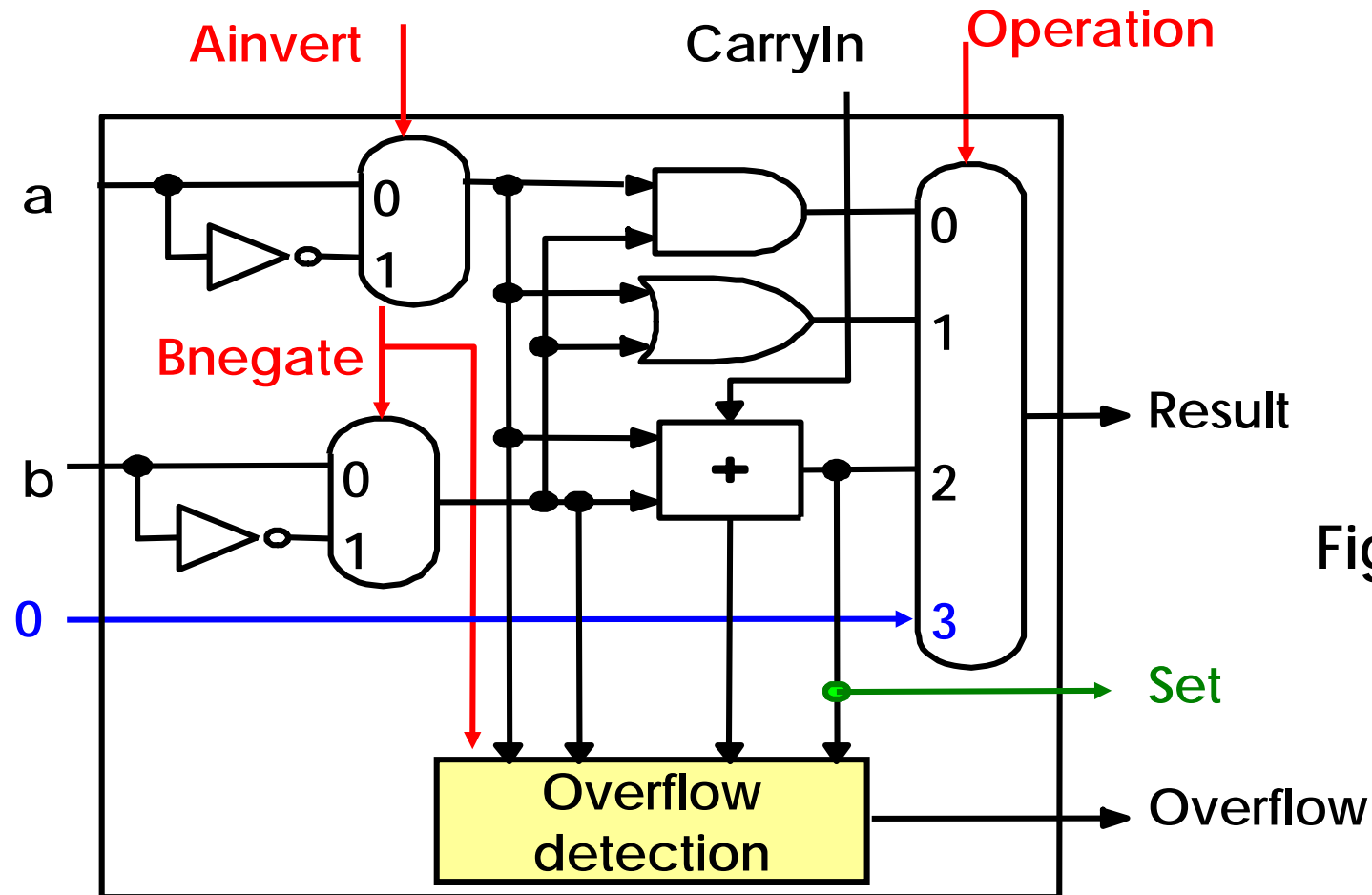
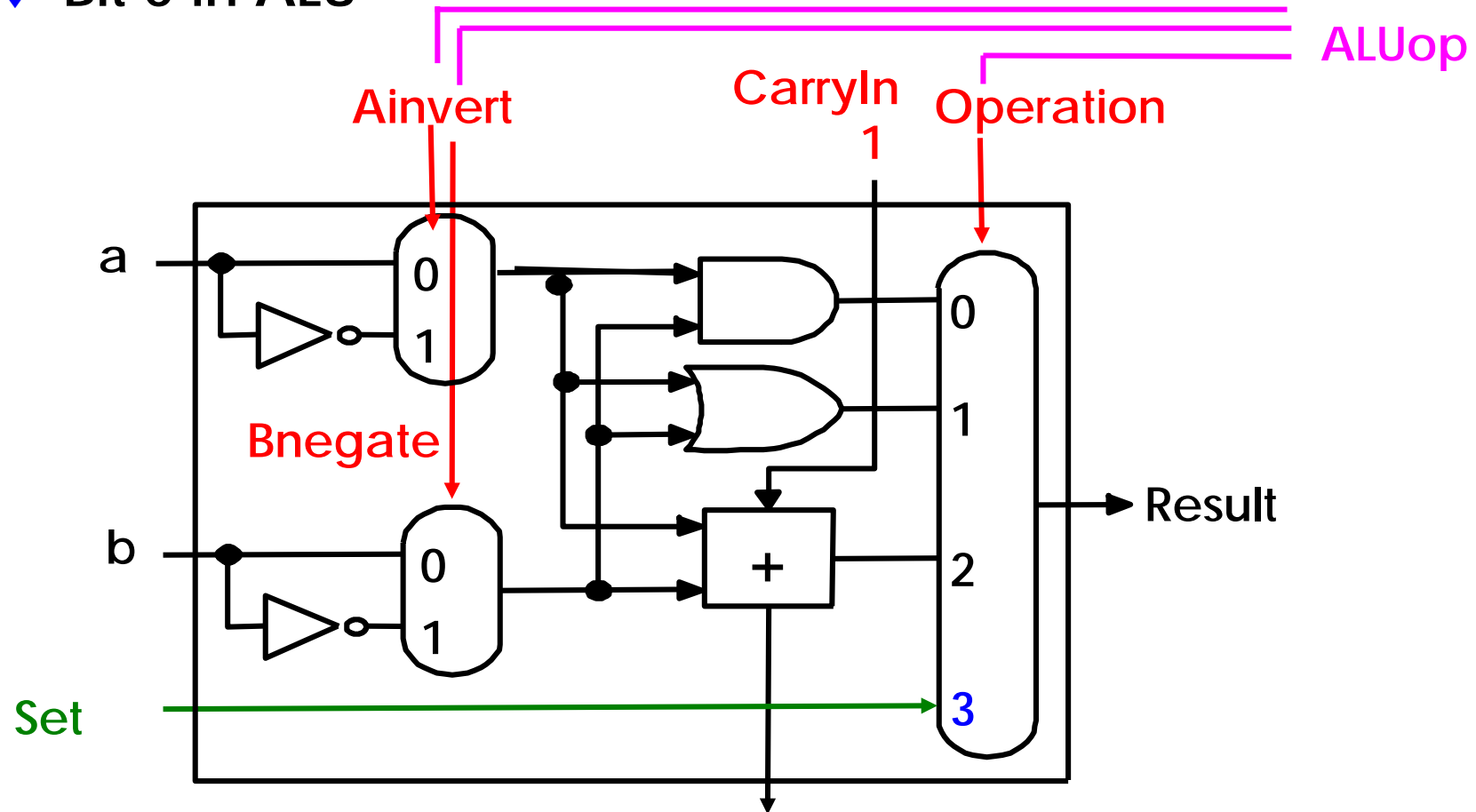


Fig. B.5.10b

Set on Less Than (3/3)

◆ Bit 0 in ALU



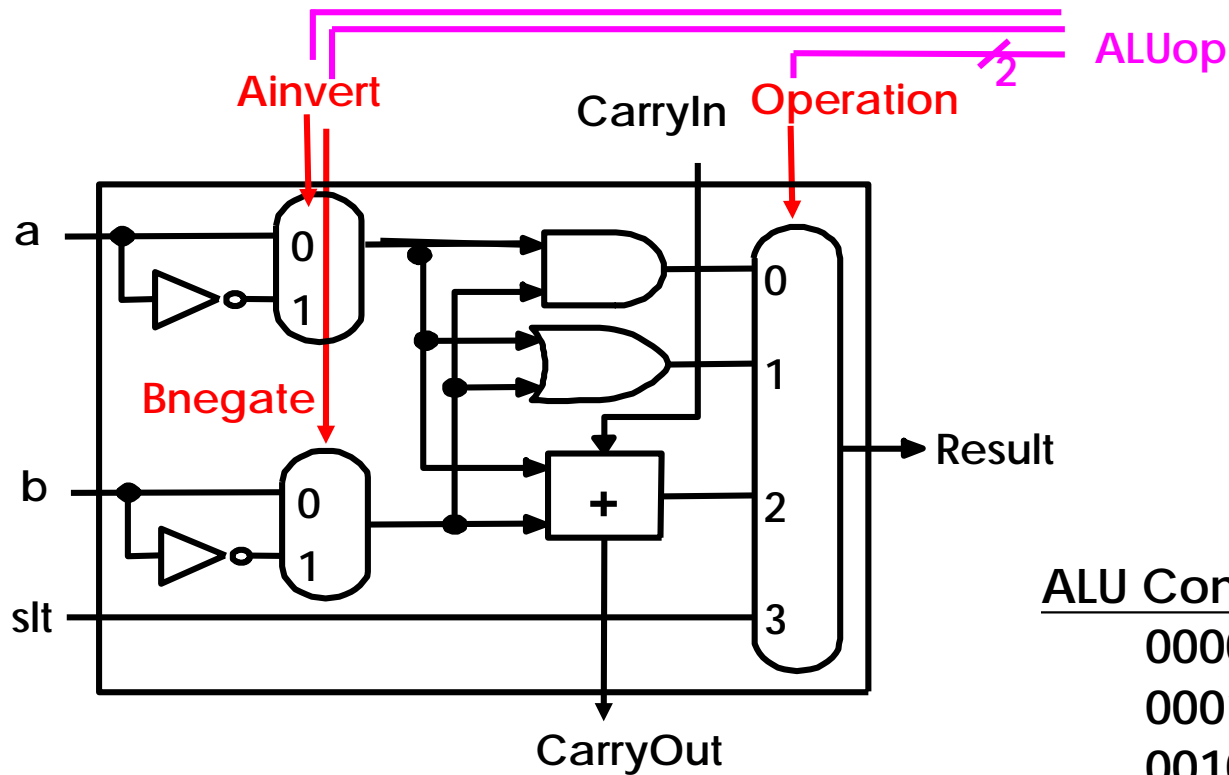
Is that all for set-on-less-than?

CarryOut

Arithmetic-22

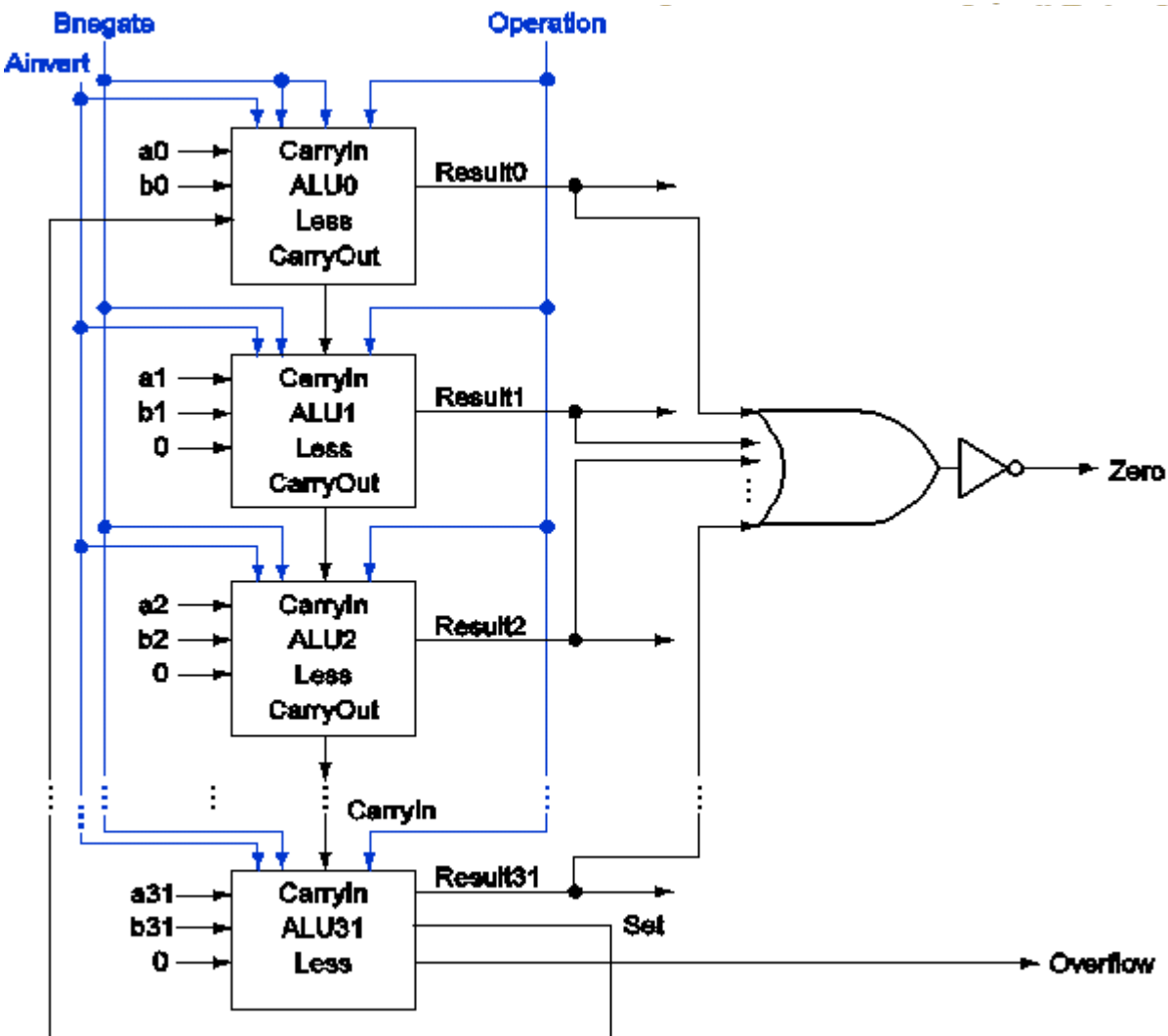
Computer Organization

ALU Control and Function



ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

Final 32-bit ALU



ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

Ripple Carry Adder

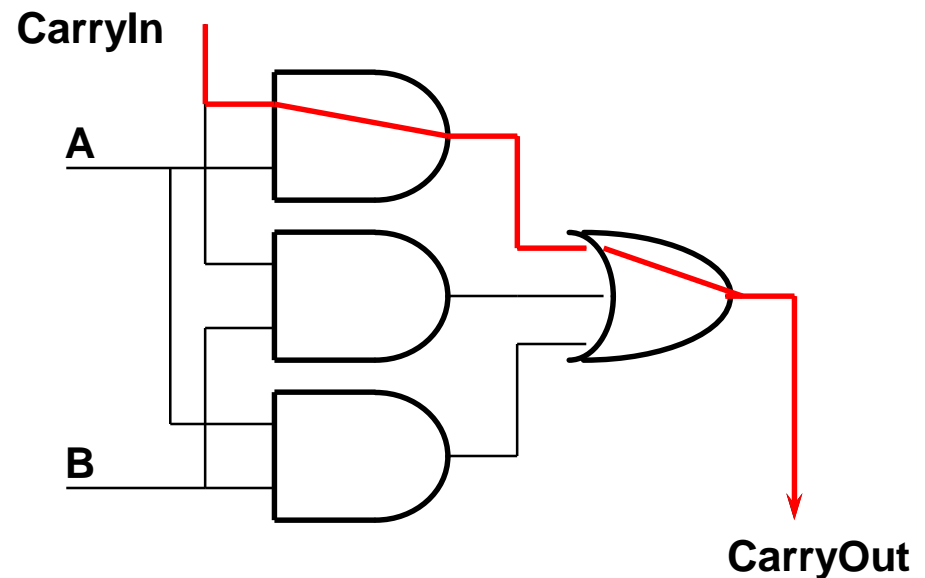
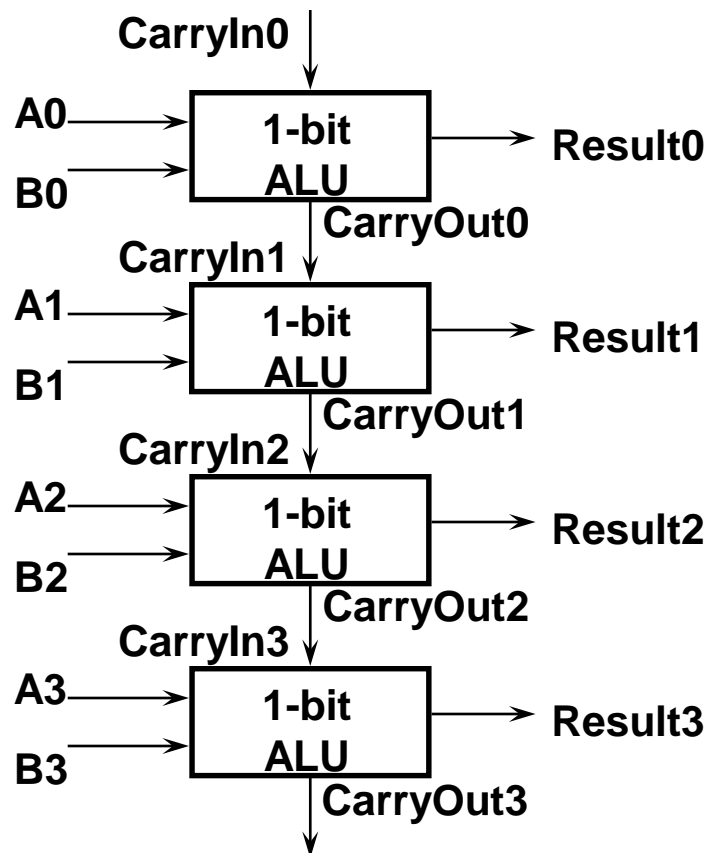
- ◆ Carry Ripple from lower-bit to the higher-bit

$$\begin{array}{r} 00111111 \\ \nwarrow \nwarrow \nwarrow \nwarrow \nwarrow \nwarrow \nwarrow \nwarrow \\ 00101010 \\ + 00010101 \\ \hline 00000000 \end{array} \quad \begin{array}{l} C_{in} = 1 \\ \swarrow \\ \text{to } 00010101 \end{array}$$

- ◆ Ripple computation dominates the run time
 - Higher-bit ALU must wait for carry from lower-bit ALU
 - Run time complexity: $O(n)$

Problems with Ripple Carry Adder

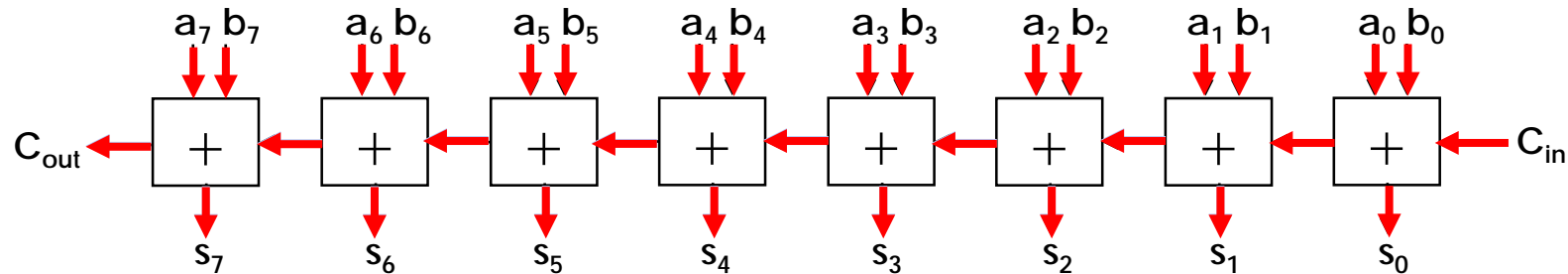
- ◆ Carry bit may have to propagate from LSB to MSB => worst case delay: N-stage delay



Design Trick: look for parallelism and throw hardware at it

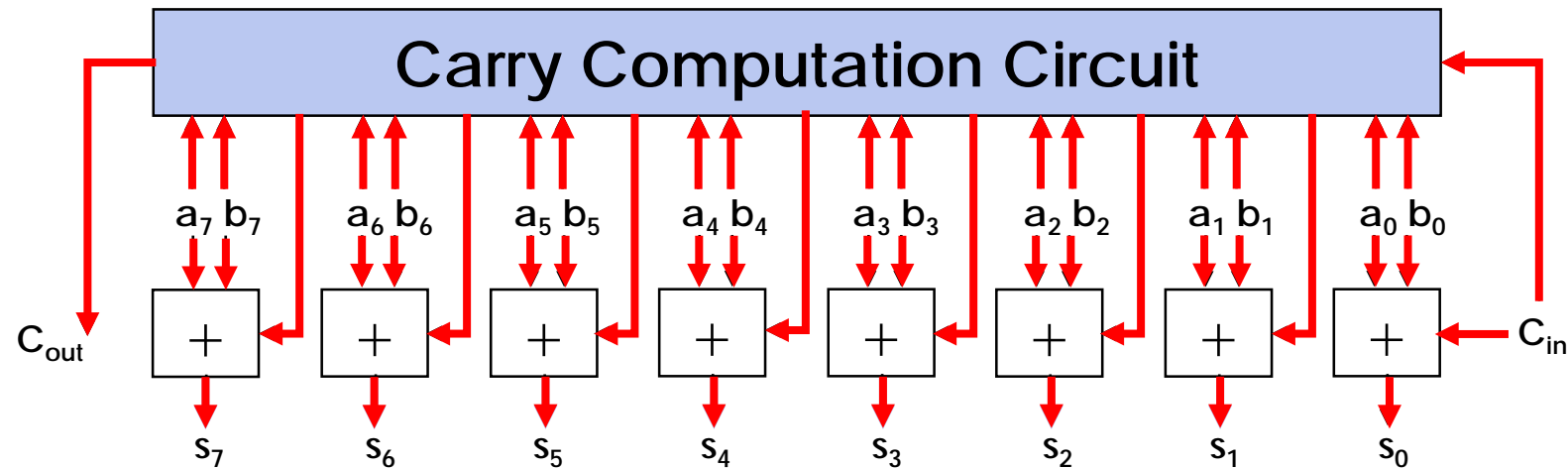
Remove the Dependency

◆ Ripple carry adder



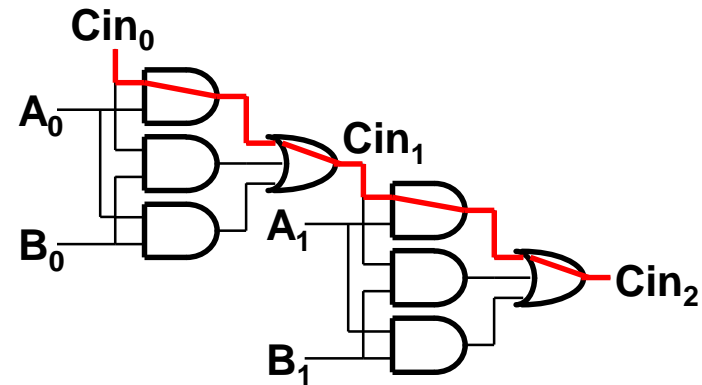
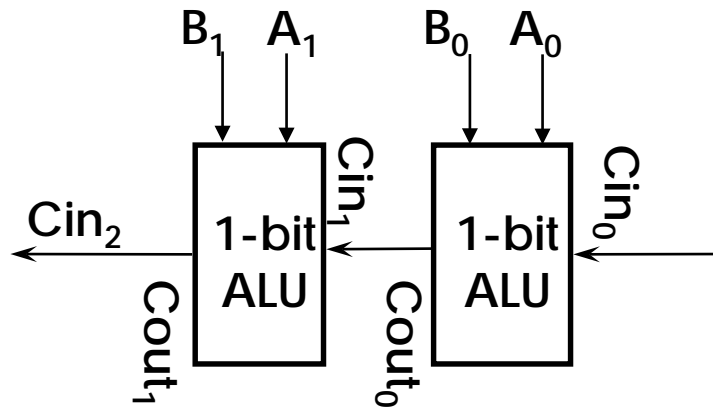
◆ Carry lookahead adder

- No carry bit propagation from LSB to MSB



Carry Lookahead: Theory (I)

(Appendix C.6)



◆ $CarryOut = (A * B) + (A * CarryIn) + (B * CarryIn)$

- $Cin_2 = Cout_1 = (A_1 * B_1) + (A_1 * Cin_1) + (B_1 * Cin_1)$

- $Cin_1 = Cout_0 = (A_0 * B_0) + (A_0 * Cin_0) + (B_0 * Cin_0)$

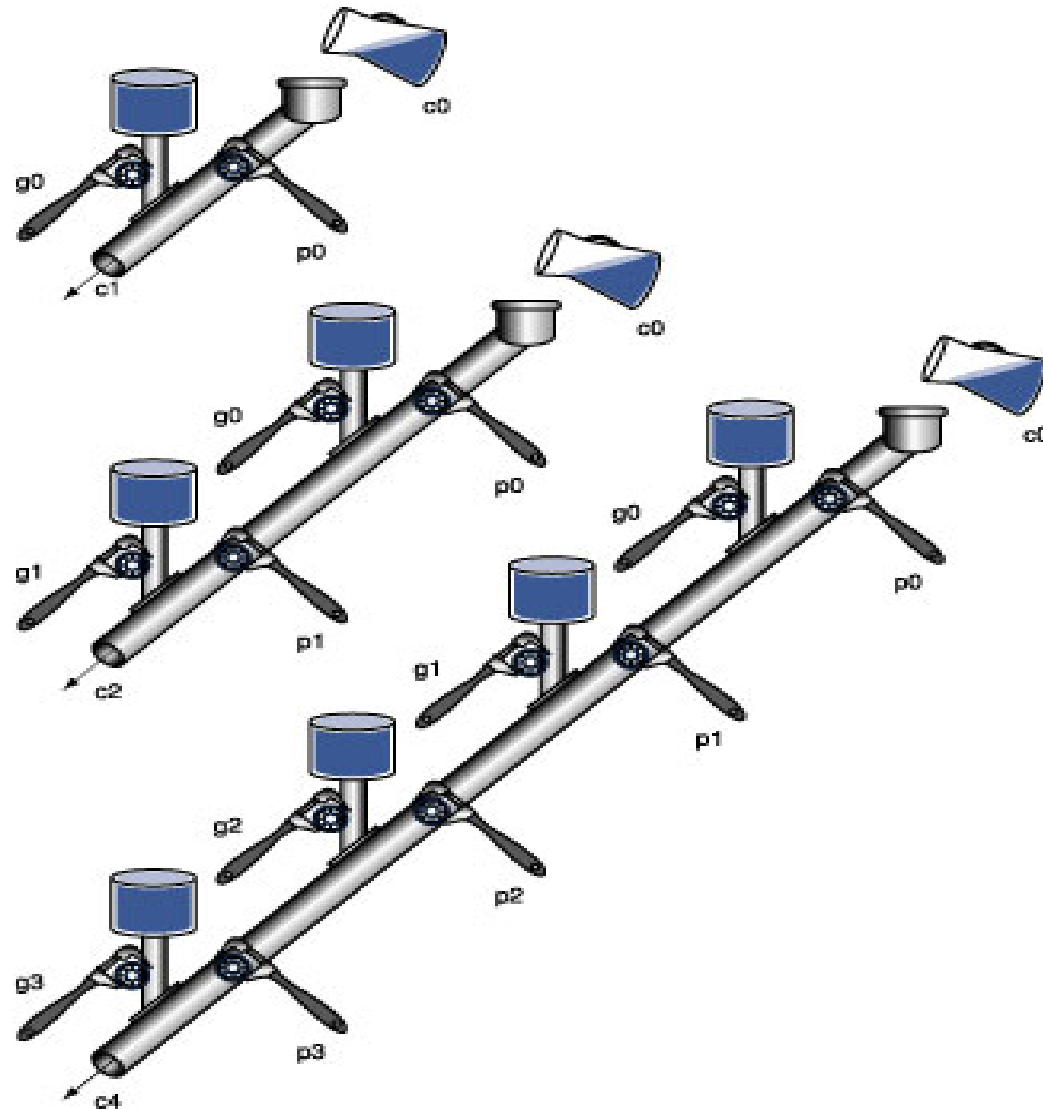
◆ Substituting Cin_1 into Cin_2 :

- $$\begin{aligned}
 Cin_2 &= (A_1 * B_1) + (A_1 * A_0 * B_0) + (A_1 * A_0 * Cin_0) + (A_1 * B_0 * Cin_0) \\
 &\quad + (B_1 * A_0 * B_0) + (B_1 * A_0 * Cin_0) + (B_1 * B_0 * Cin_0) \\
 &= (A_1 * B_1) + (A_1 + B_1) * (A_0 * B_0) + (A_1 + B_1) * (A_0 + B_0) * Cin_0
 \end{aligned}$$

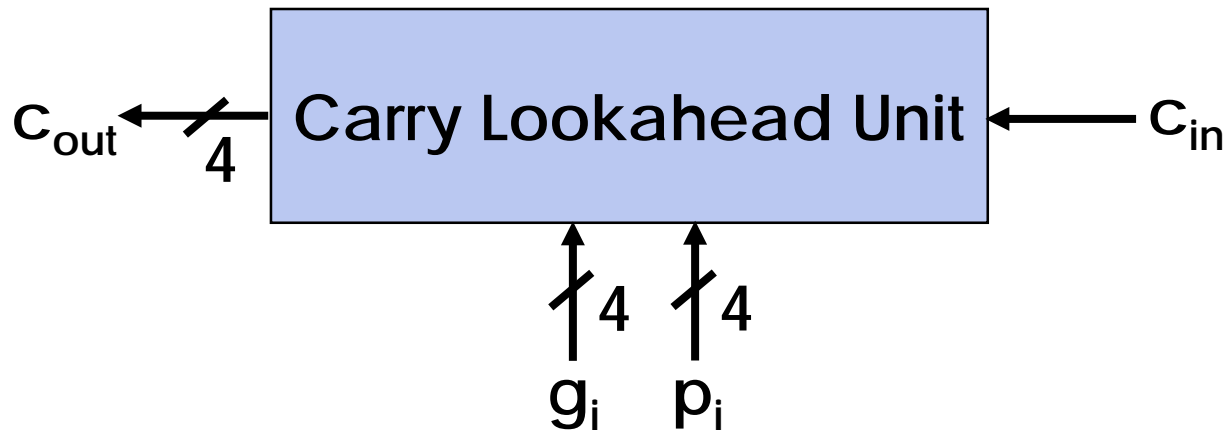
Carry Lookahead: Theory (II)

- ◆ Now define two new terms:
 - Generate Carry at Bit i: $g_i = A_i * B_i$
 - Propagate Carry via Bit i: $p_i = A_i + B_i$
- ◆ We can rewrite:
 - $Cin_1 = g_0 + (p_0 * Cin_0)$
 - $Cin_2 = g_1 + (p_1 * g_0) + (p_1 * p_0 * Cin_0)$
 - $Cin_3 = g_2 + (p_2 * g_1) + (p_2 * p_1 * g_0) + (p_2 * p_1 * p_0 * Cin_0)$
 - $Cin_4 = g_3 + (p_3 * g_2) + (p_3 * p_2 * g_1) + (p_3 * p_2 * p_1 * g_0) + (p_3 * p_2 * p_1 * p_0 * Cin_0)$
- ◆ Carry going into bit 3 is 1 if
 - We generate a carry at bit 2 (g_2)
 - Or we generate a carry at bit 1 (g_1) and bit 2 allows it to propagate ($p_2 * g_1$)
 - Or we generate a carry at bit 0 (g_0) and bit 1 as well as bit 2 allows it to propagate ...

A Plumbing Analogy for Carry Lookahead (1, 2, 4 bits)

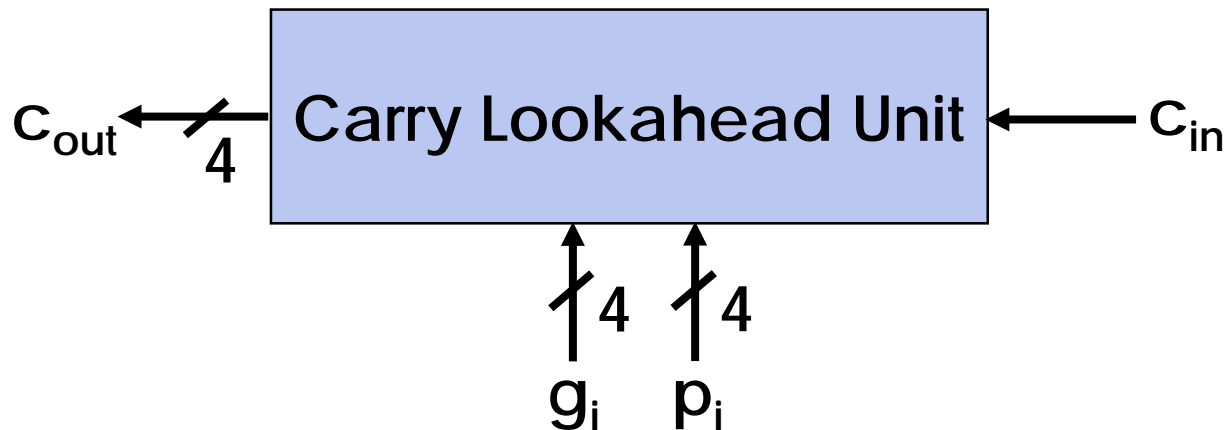


Carry Lookahead Unit



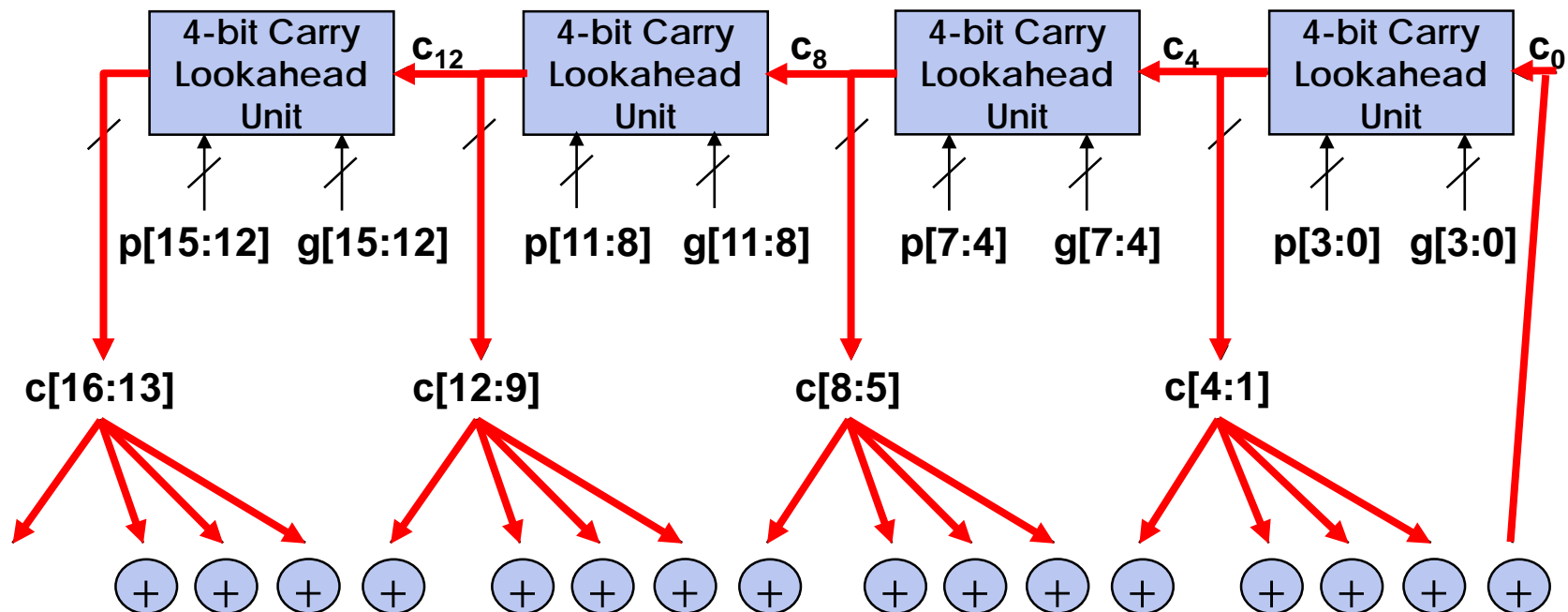
Common Carry Lookahead Adder

- ◆ Expensive to build a "full" carry lookahead adder
 - Just imagine length of the equation for $C_{in_{31}}$
- ◆ Common practices:
 - Cascaded carry look-ahead adder
 - Multiple level carry look-ahead adder



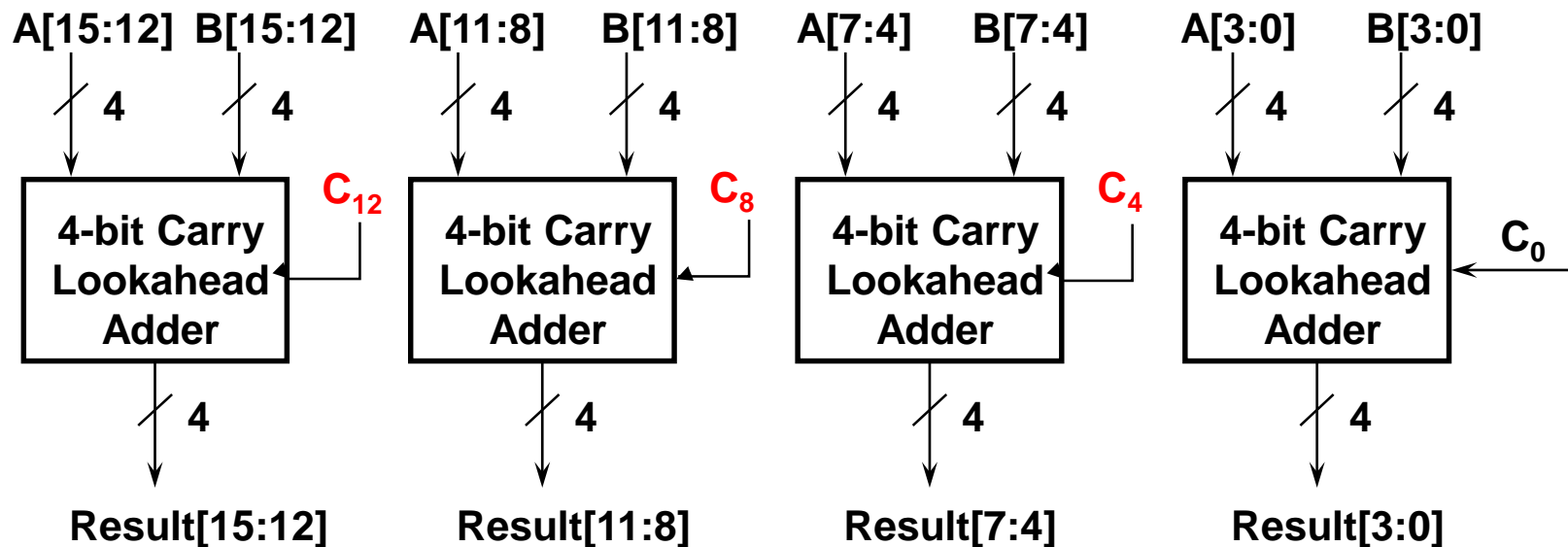
Cascaded Carry Lookahead

- ◆ Connects several N-bit lookahead adders to form a big one



Multiple Level Carry Lookahead

- ◆ View an N-bit lookahead adder as a block
- ◆ Where to get C_{in} of the block ?



- Generate "super" P_i and G_i of the block
- Use next level carry lookahead structure to generate block C_{in}

Recap of Carry Lookahead Theory

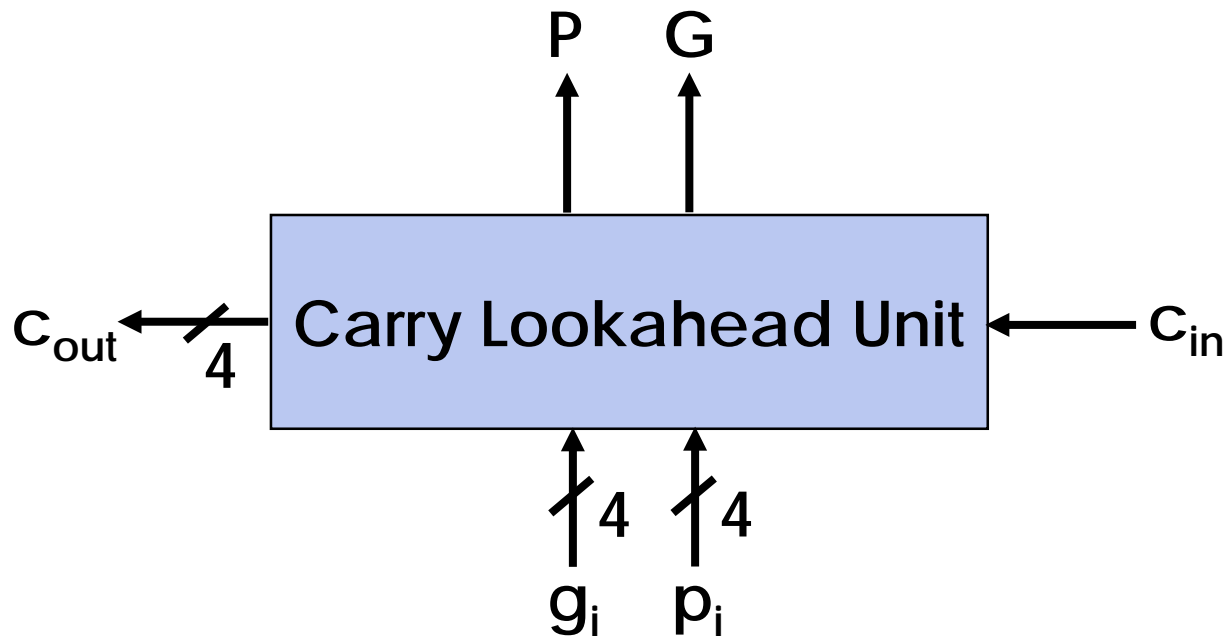
◆ Now define two new terms:

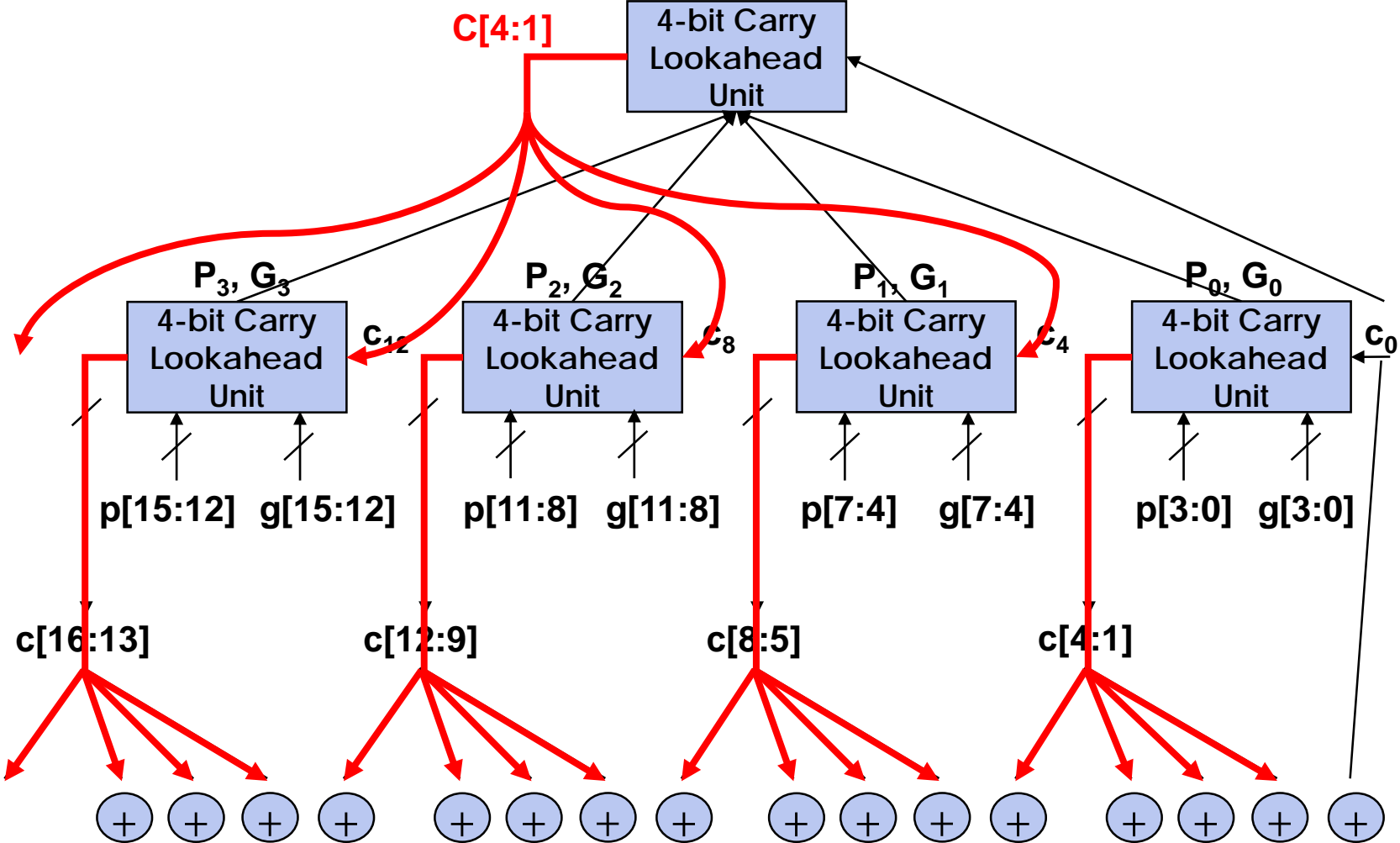
- Generate Carry at Bit i: $g_i = A_i * B_i$
- Propagate Carry via Bit i: $p_i = A_i + B_i$

◆ We can rewrite:

- $Cin_1 = g_0 + (p_0 * Cin_0)$
- $Cin_2 = g_1 + (p_1 * g_0) + (p_1 * p_0 * Cin_0)$
- $Cin_3 = g_2 + (p_2 * g_1) + (p_2 * p_1 * g_0) + (p_2 * p_1 * p_0 * Cin_0)$
- $Cin_4 = g_3 + (p_3 * g_2) + (p_3 * p_2 * g_1) + (p_3 * p_2 * p_1 * g_0) + (p_3 * p_2 * p_1 * p_0 * Cin_0) = G + (P * Cin_0)$
- G: CarryOut can be generated among these 4 bits
 - $G = g_3 + (p_3 * g_2) + (p_3 * p_2 * g_1) + (p_3 * p_2 * p_1 * g_0)$
- P: CarryOut can be propagated among these 4 bits
 - $P = p_3 * p_2 * p_1 * p_0$

Carry Lookahead Unit





A Plumbing Analogy for Carry Lookahead (Next Level P0 and G0)

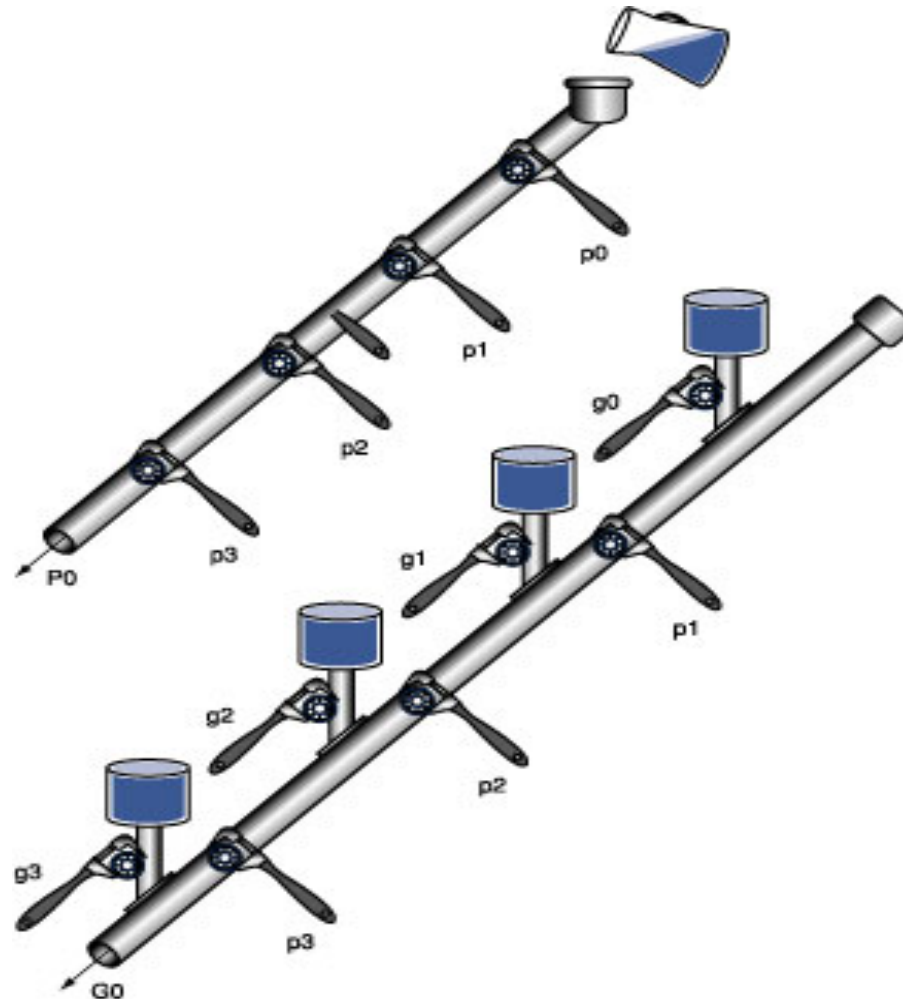
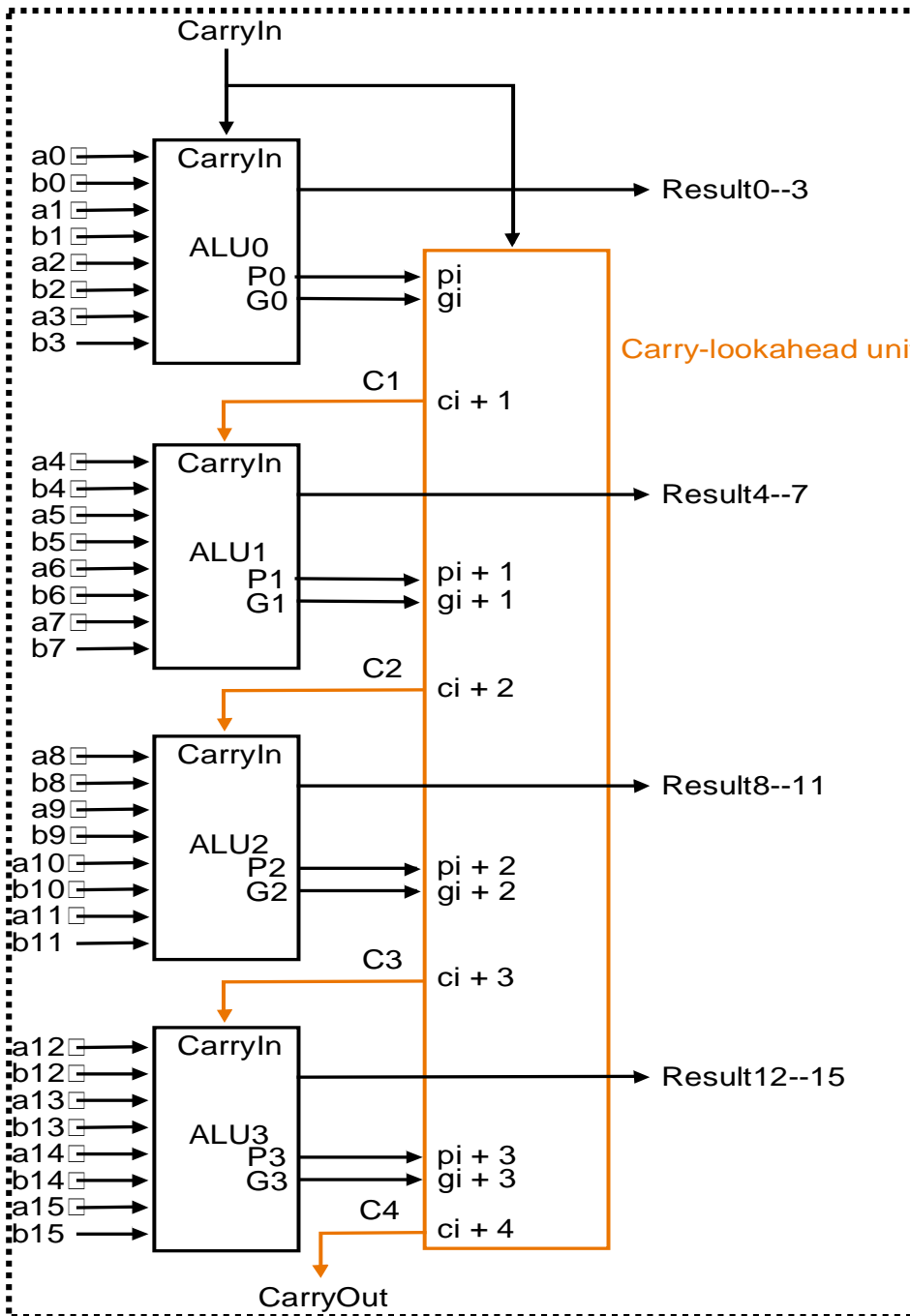


Fig. B.6.2

A Carry Lookahead Adder



A	B	Cout	
0	0	0	kill
0	1	Cin	propagate
1	0	Cin	propagate
1	1	1	generate

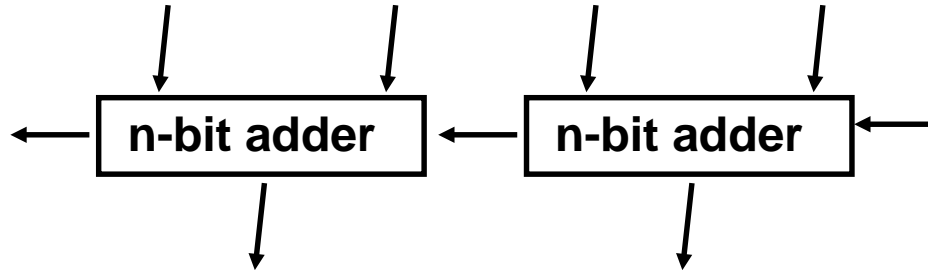
$$G = A * B$$

$$P = A + B$$

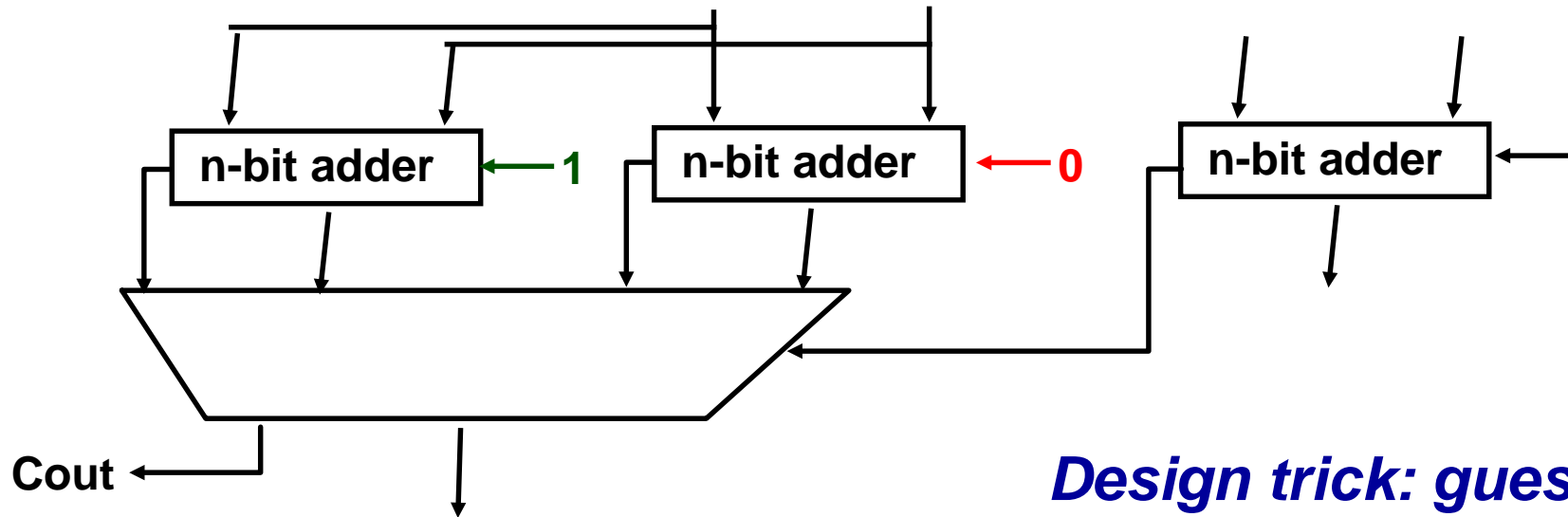
Fig. B.6.3

Carry-select Adder

$$CP(2n) = 2 * CP(n)$$



$$CP(2n) = CP(n) + CP(\text{mux})$$



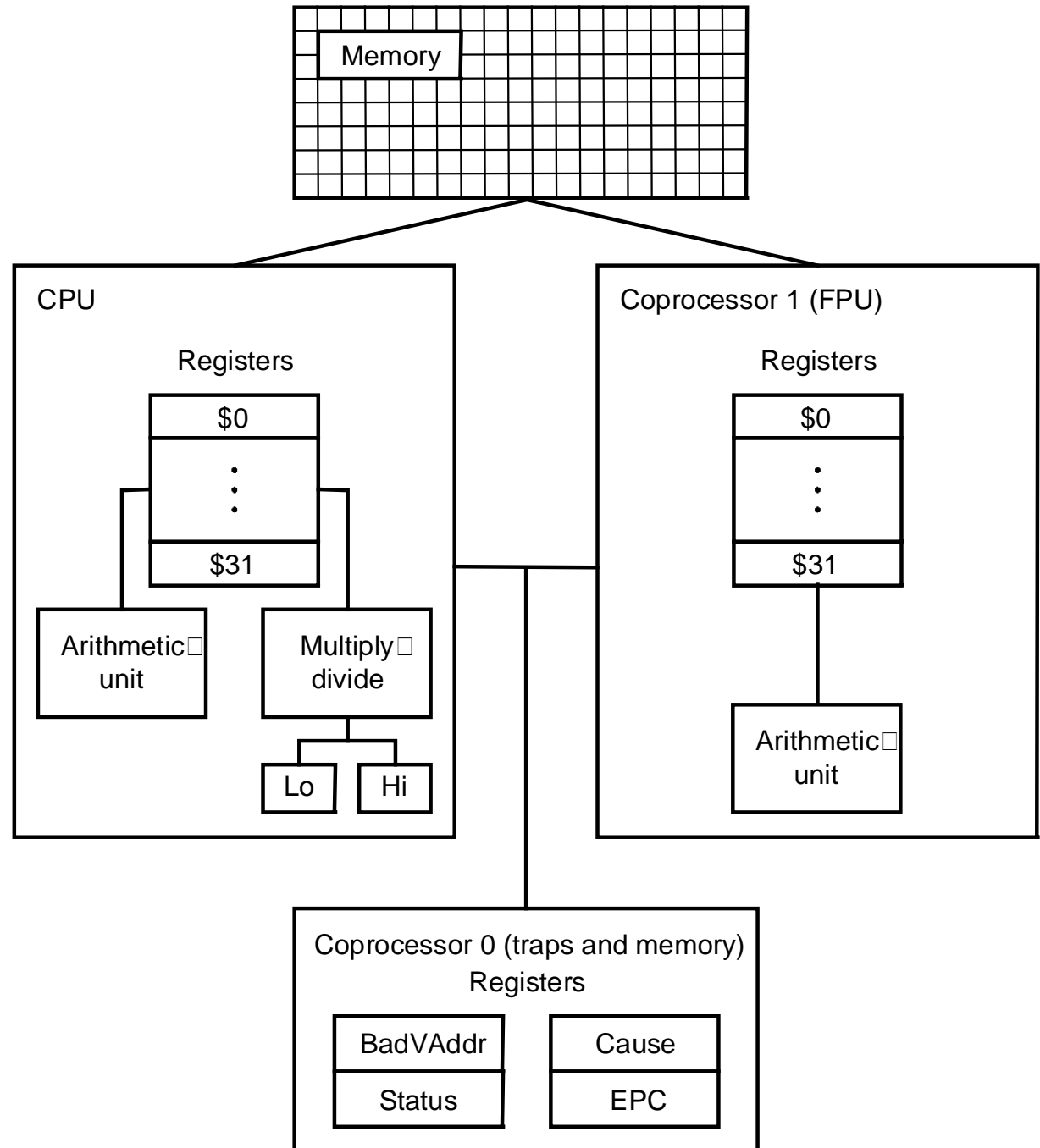
Arithmetic for Multimedia

- ◆ Graphics and media processing operates on vectors of 8-bit and 16-bit data
 - Use 64-bit adder, with partitioned carry chain
 - Operate on 8x8-bit, 4x16-bit, or 2x32-bit vectors
 - SIMD (single-instruction, multiple-data)
- ◆ Saturating operations
 - On overflow, result is largest representable value
 - c.f. 2s-complement modulo arithmetic
 - Ex: clipping in audio, saturation in video

Outline

- ◆ Constructing an arithmetic logic unit (Appendix C)
- ◆ Multiplication (Sec. 3.3, Appendix C)
- ◆ Division (Sec. 3.4)
- ◆ Floating point (Sec. 3.5)

MIPS R2000 Organization



Multiplication in MIPS

`mult $t1, $t2` `# t1 * t2`

- ◆ No destination register: product could be $\sim 2^{64}$; need two special registers to hold it
- ◆ 3-step process:

`$t1` `01111111111111111111111111111111`

`X $t2` `01000000000000000000000000000000`

`00011111111111111111111111111111` | `11000000000000000000000000000000`

HI

LO

`mfhi $t3`

`$t3` `00011111111111111111111111111111`

`mflo $t4`

`$t4` `11000000000000000000000000000000`

MIPS Multiplication

- ◆ Two 32-bit registers for product
 - HI: most-significant 32 bits
 - LO: least-significant 32 bits
- ◆ Instructions
 - mult rs, rt / multu rs, rt
 - 64-bit product in HI/LO
 - mfhi rd / mflo rd
 - Move from HI/LO to rd
 - Can test HI value to see if product overflows 32 bits
 - mul rd, rs, rt
 - Least-significant 32 bits of product → rd

Unsigned Multiply

- ♦ Paper and pencil example (unsigned):

Multiplicand
Multiplier

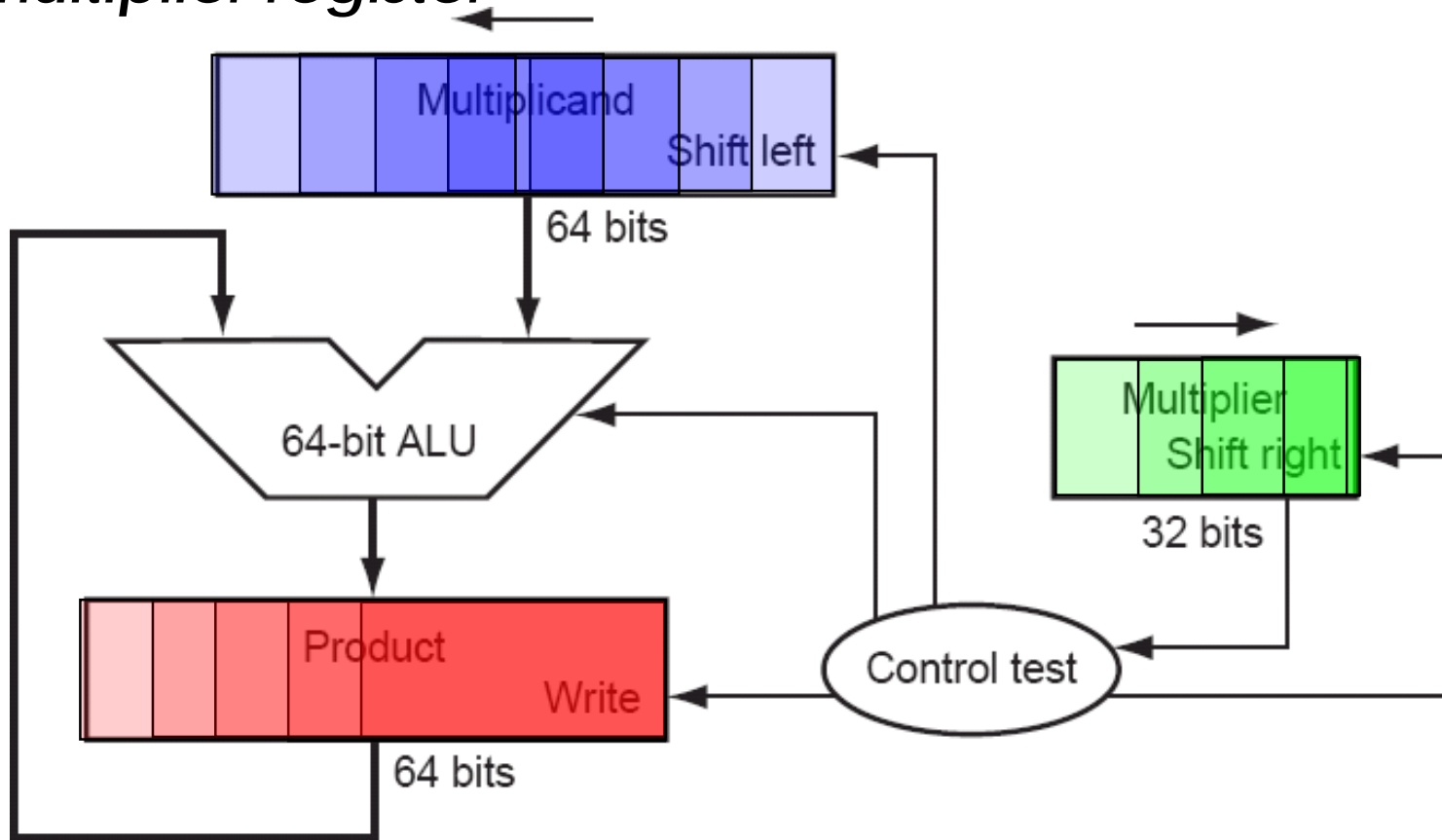
$$\begin{array}{r} 1000_{\text{ten}} \\ \times 1001_{\text{ten}} \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 01001000_{\text{ten}} \end{array}$$

Product

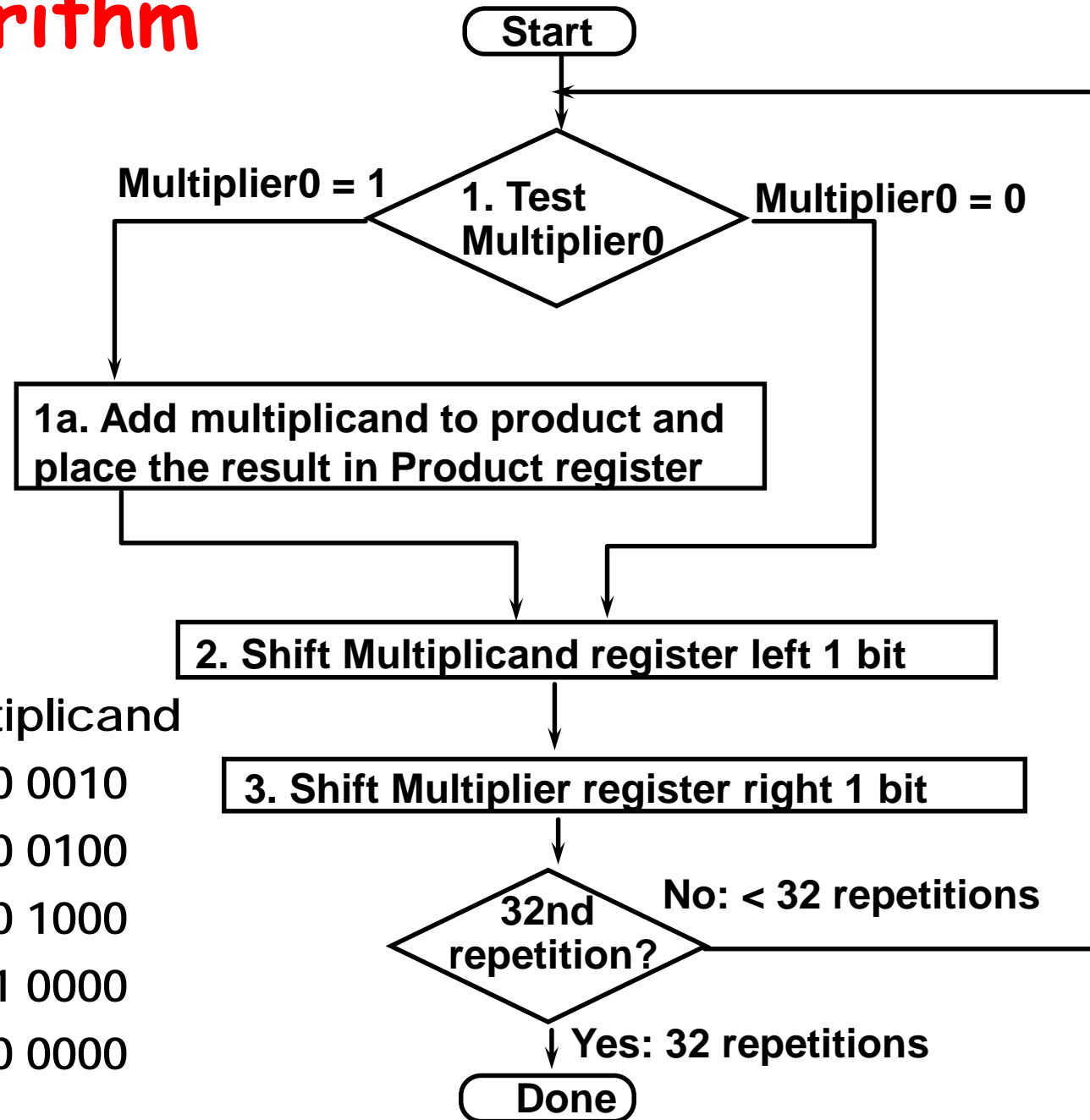
- ♦ m bits x n bits = m+n bit product
- ♦ Binary makes it easy:
 - 0 => place 0 (0 × multiplicand)
 - 1 => place a copy (1 × multiplicand)
- ♦ 2 versions of multiply hardware and algorithm

Unsigned Multiplier (Ver. 1)

- ◆ 64-bit *multiplicand* register (with 32-bit multiplicand at right half), 64-bit ALU, 64-bit *product* register, 32-bit *multiplier* register



Multiply Algorithm (Ver. 1)



0010 x 0011

Product	Multiplier	Multiplicand
0000 0000	001 1	0000 0010
0000 0010	000 1	0000 0100
0000 0110	000 0	0000 1000
0000 0110	000 0	0001 0000
0000 0110	0000	0010 0000

Observations: Multiply Ver. 1

- ◆ 1 clock per cycle → ~100 clocks per multiply
 - Ratio of multiply to add 5:1 to 100:1
- ◆ Half of the bits in multiplicand always 0
→ 64-bit adder is wasted
- ◆ 0's inserted in right of multiplicand as shifted
→ least significant bits of product never changed once formed
- ◆ Instead of shifting multiplicand to left, shift product to right?
- ◆ Product register wastes space → combine Multiplier and Product register

Unsigned Multiply

- ♦ Paper and pencil example (unsigned):

Multiplicand
Multiplier

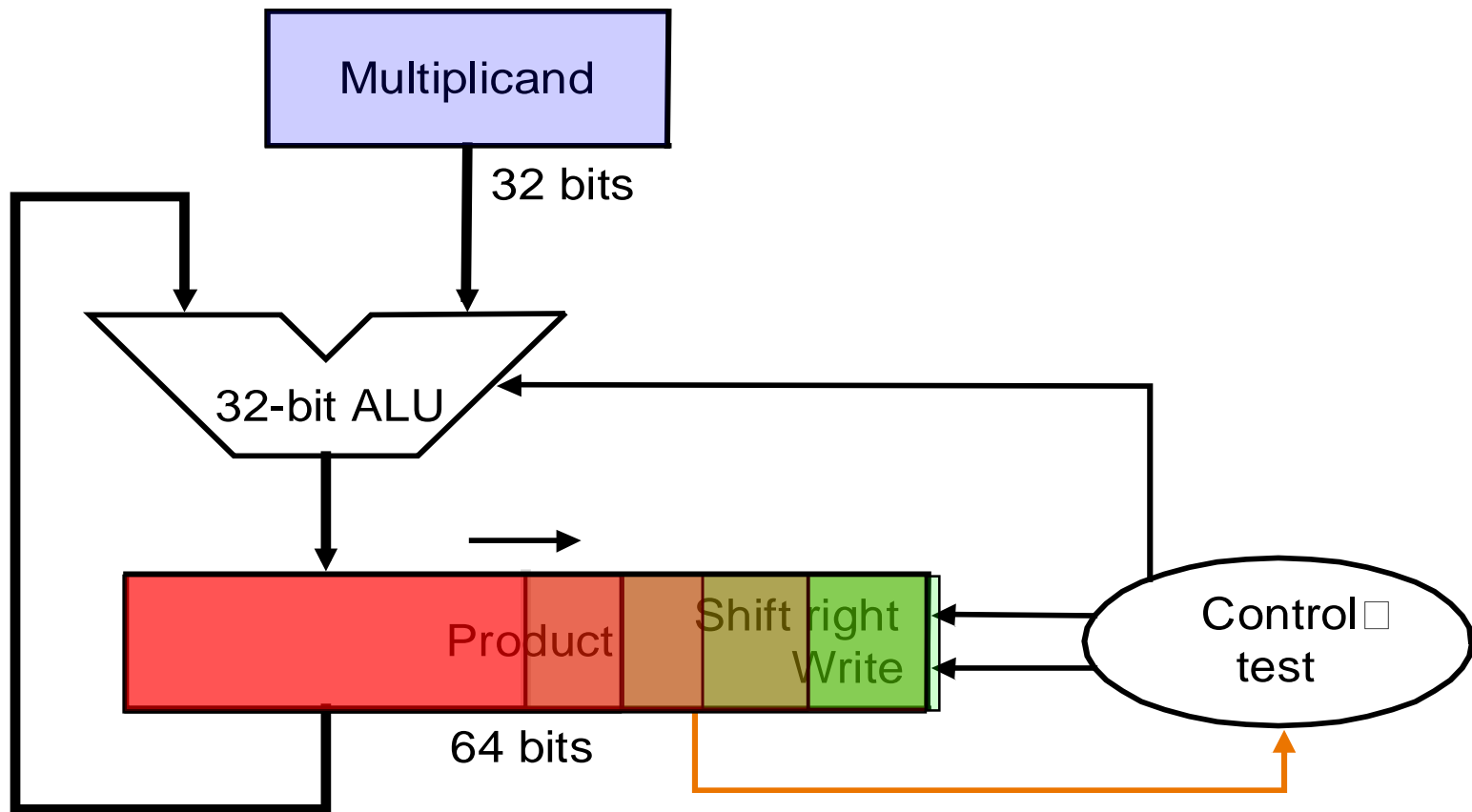
$$\begin{array}{r} 1000_{\text{ten}} \\ \times 1001_{\text{ten}} \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 01001000_{\text{ten}} \end{array}$$

Product

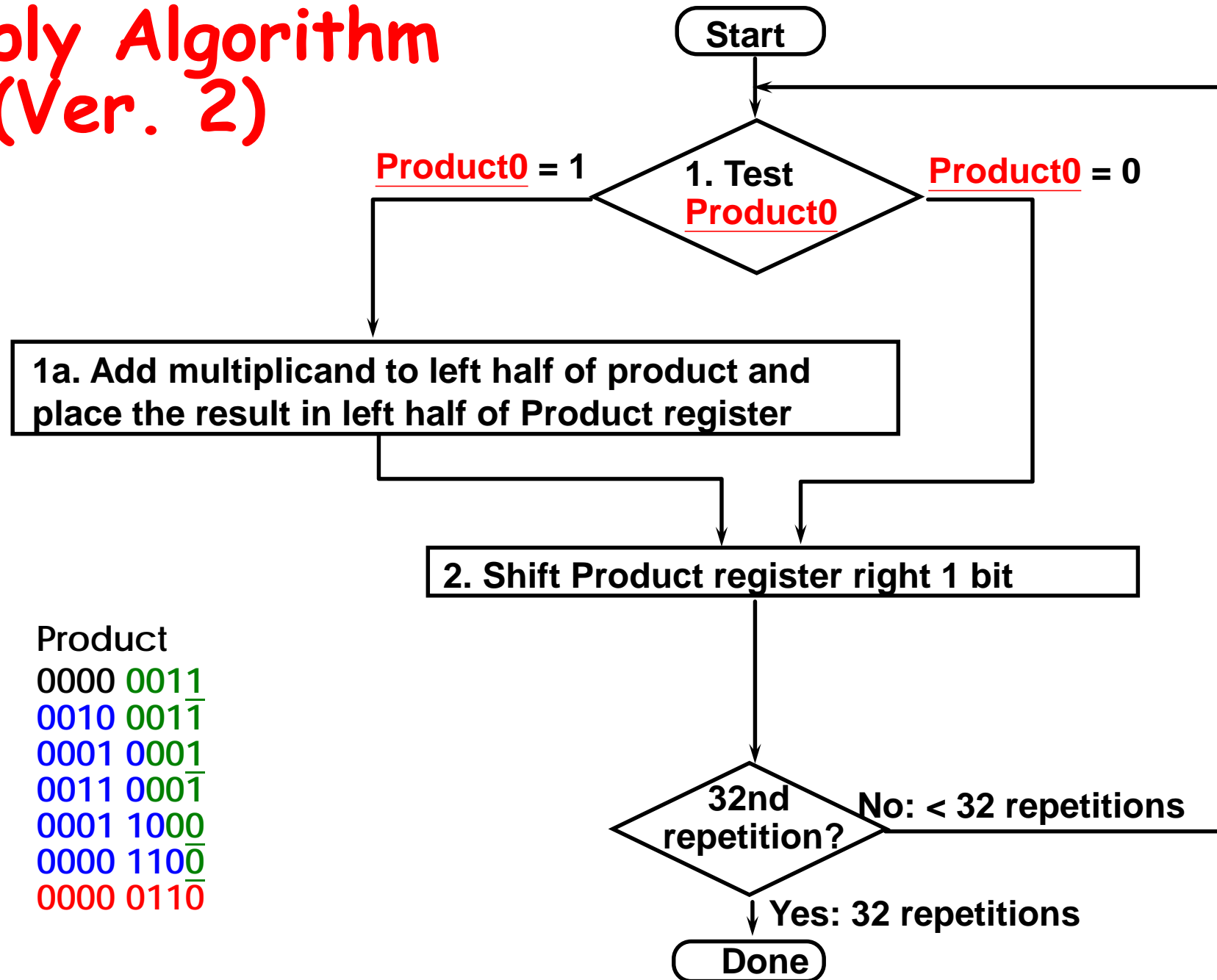
- ♦ m bits \times n bits = $m+n$ bit product
- ♦ Binary makes it easy:
 - 0 => place 0 (0 \times multiplicand)
 - 1 => place a copy (1 \times multiplicand)
- ♦ 2 versions of multiply hardware and algorithm

Unsigned Multiplier (Ver. 2)

- ◆ 32-bit Multiplicand register, 32-bit ALU, 64-bit Product register (HI & LO in MIPS), (0-bit Multiplier register)



Multiply Algorithm (Ver. 2)



0010 x 0011

Multiplicand	Product
0010	0000 0011
0010	0010 0011
0010	0001 0001
0010	0011 0001
0010	0001 1000
0010	0000 1100
0010	0000 0110

Observations: Multiply Ver. 2

- ◆ 2 steps per bit because multiplier and product registers combined
- ◆ MIPS registers HI and LO are left and right half of Product register
→ this gives the MIPS instruction MultU
- ◆ What about signed multiplication?
 - The easiest solution is to make both positive and remember whether to complement product when done (leave out sign bit, run for 31 steps)
 - Apply definition of 2's complement
 - sign-extend partial products and subtract at end
 - Booth's Algorithm is an elegant way to multiply signed numbers using same hardware as before and save cycles

Signed Multiply

- ♦ Paper and pencil example (signed):

Multiplicand	1001 (-7)
Multiplier	X 1001 (-7)
	<hr/> 11111001
	+ 0000000
	+ 000000
	- 11001
Product	<hr/> 00110001 (49)

- ♦ Rule 1: Multiplicand sign extended

- ♦ Rule 2: Sign bit (s) of Multiplier

- 0 => 0 × multiplicand
- 1 => -1 × multiplicand

- ♦ Why rule 2 ?

- $X = s x_{n-2} x_{n-3} \dots x_1 x_0$ (2's complement)

- $\text{Value}(X) = -1 \times s \times 2^{n-1} + x_{n-2} \times 2^{n-2} + \dots + x_0 \times 2^0$

Booth's Algorithm: Motivation

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

	0010 _{two}	
x	0110 _{two}	
+	0000	shift (0 in multiplier)
+	0010	add (1 in multiplier)
+	0010	add (1 in multiplier)
+	0000	shift (0 in multiplier)
	00001100 _{two}	

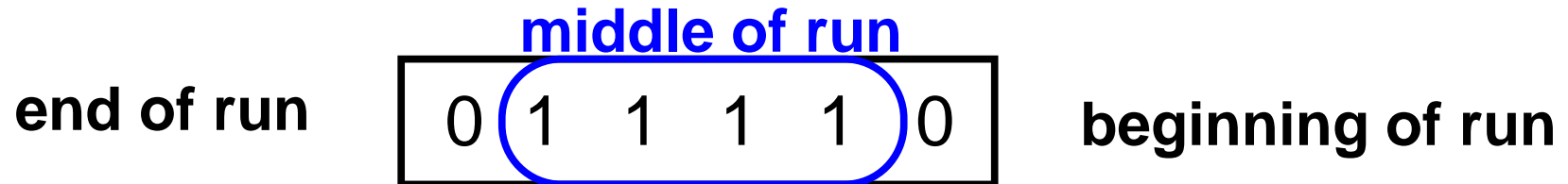
- ◆ Can get same result in more than one way:

$$6 = -2 + 8 \qquad 0110 = -00010 + 01000$$

- ◆ Basic idea: replace a string of 1s with an initial subtract on seeing a one and add after last one

	0010 _{two}	
x	0110 _{two}	
	0000	shift (0 in multiplier)
-	0010	sub (first 1 in multiplier)
	0000	shift (mid string of 1s)
+	0010	add (prior step had last 1)
	00001100 _{two}	

Booth's Algorithm: Rationale



Current Bit to bit	right	Explanation	Example	Op
1	0	Begins run of 1s	0000111 <u>1</u> 000	sub
1	1	Middle run of 1s	00001 <u>11</u> 1000	none
0	1	End of run of 1s	000 <u>01</u> 111000	add
0	0	Middle run of 0s	0 <u>00</u> 01111000	none

Originally for speed (when shift was faster than add)

♦ Why it works?

$$\begin{array}{r}
 -10 \\
 + 100000 \\
 \hline
 011110
 \end{array}$$

Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:
 - 00: Middle of a string of 0s, no arithmetic op.
 - 01: End of a string of 1s, so add multiplicand to the left half of the product
 - 10: Beginning of a string of 1s, so subtract multiplicand from the left half of the product
 - 11: Middle of a string of 1s, so no arithmetic op.
2. As in the previous algorithm, shift the Product register right (arithmetically) 1 bit

Booths Example (2 x 7)

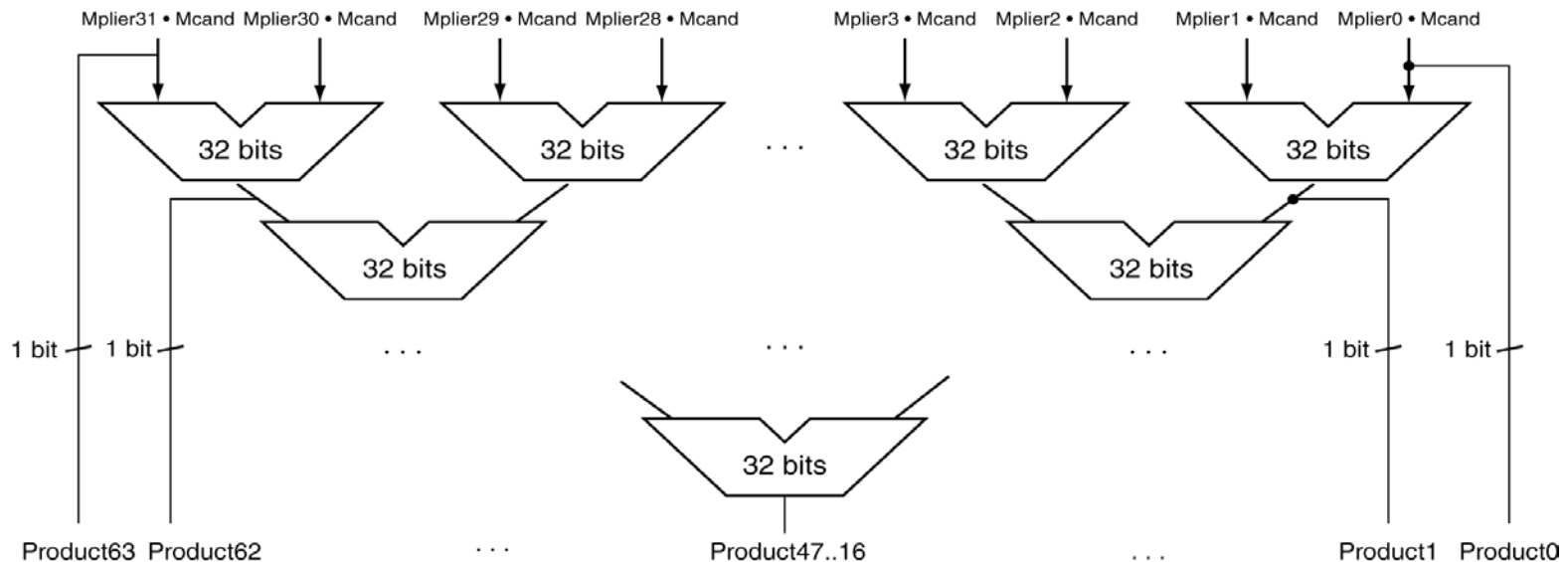
Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. $P = P - m$	1110	+1110	
		1110 0111 0	shift P (sign ext)
1b.	0010	<u>1</u> 111 0011 1	11 -> nop, shift
2.	0010	<u>1</u> 111 1 001 1	11 -> nop, shift
3.	0010	<u>1</u> 111 11 00 1	01 -> add
4a.	0010	+0010	
		0001 11 00 1	shift
4b.	0010	0000 1110 0	done

Booths Example (2 x -3)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. $P = P - m$	1110	+1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	<u>1</u> 111 0 110 1	01 -> add
	0010	+0010	
2a.		0001 0 110 1	shift P
2b.	0010	0000 10 11 0	10 -> sub
	1110	+1110	
3a.	0010	1110 10 11 0	shift
3b.	0010	<u>1</u> 111 010 1 1	11 -> nop
4a.		1111 010 1 1	shift
4b.	0010	<u>1</u> 111 1010 1	done

Faster Multiplier

- ◆ A combinational multiplier
- ◆ Use multiple adders
 - Cost/performance tradeoff



- ◆ Can be pipelined
 - Several multiplication performed in parallel

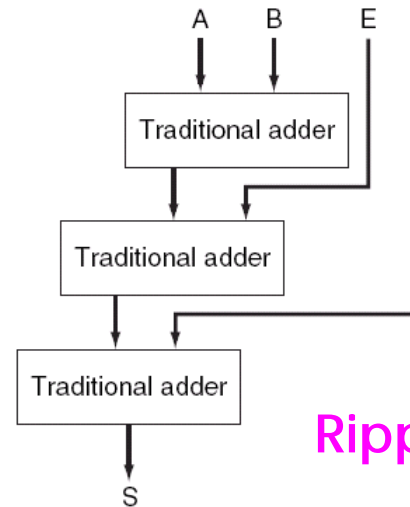
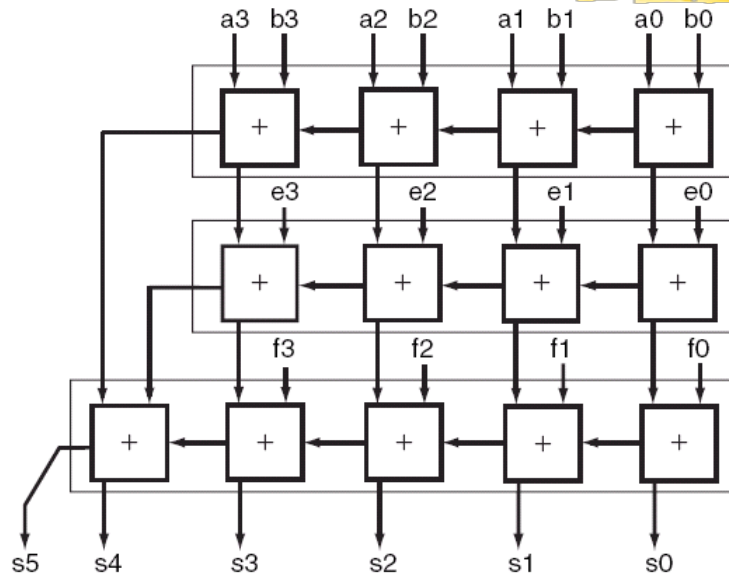
Wallace Tree Multiplier

- ◆ Use carry save adders: three inputs and two outputs

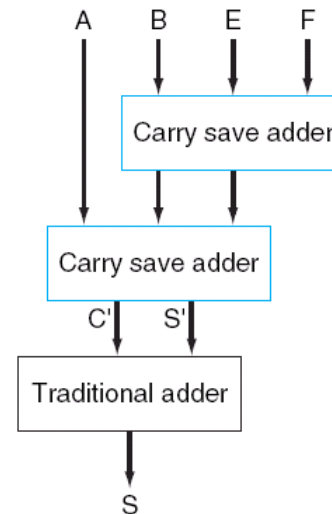
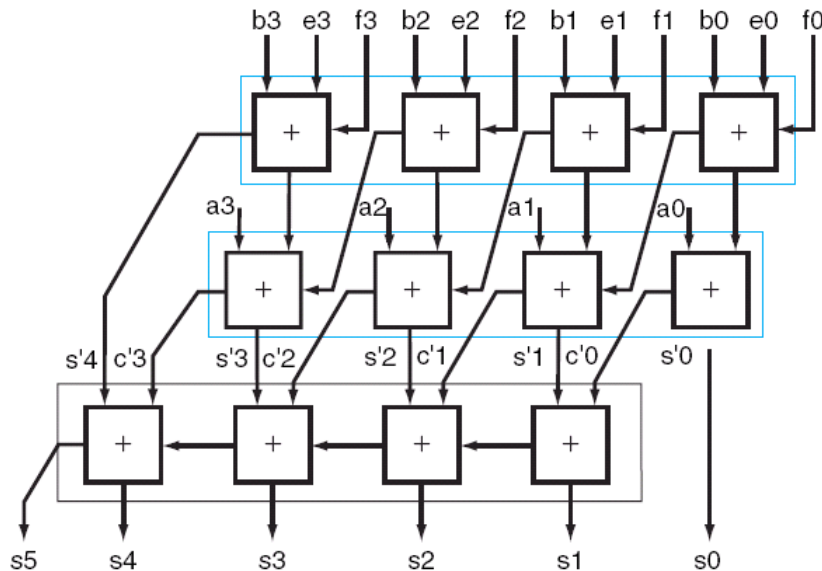
$$\begin{array}{r} 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0 \\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 1 \\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \\ \hline 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ \text{(sum)} \\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ \text{(carry)} \end{array}$$

- ◆ 8 1-bit full adders
 - One full adder delay (no carry propagation)
- ◆ The last stage is performed by regular adder
- ◆ What is the minimum delay for 16 x 16 multiplier ?

Ripple Carry Adder vs. Carry Save Adder



Ripple Carry Adder

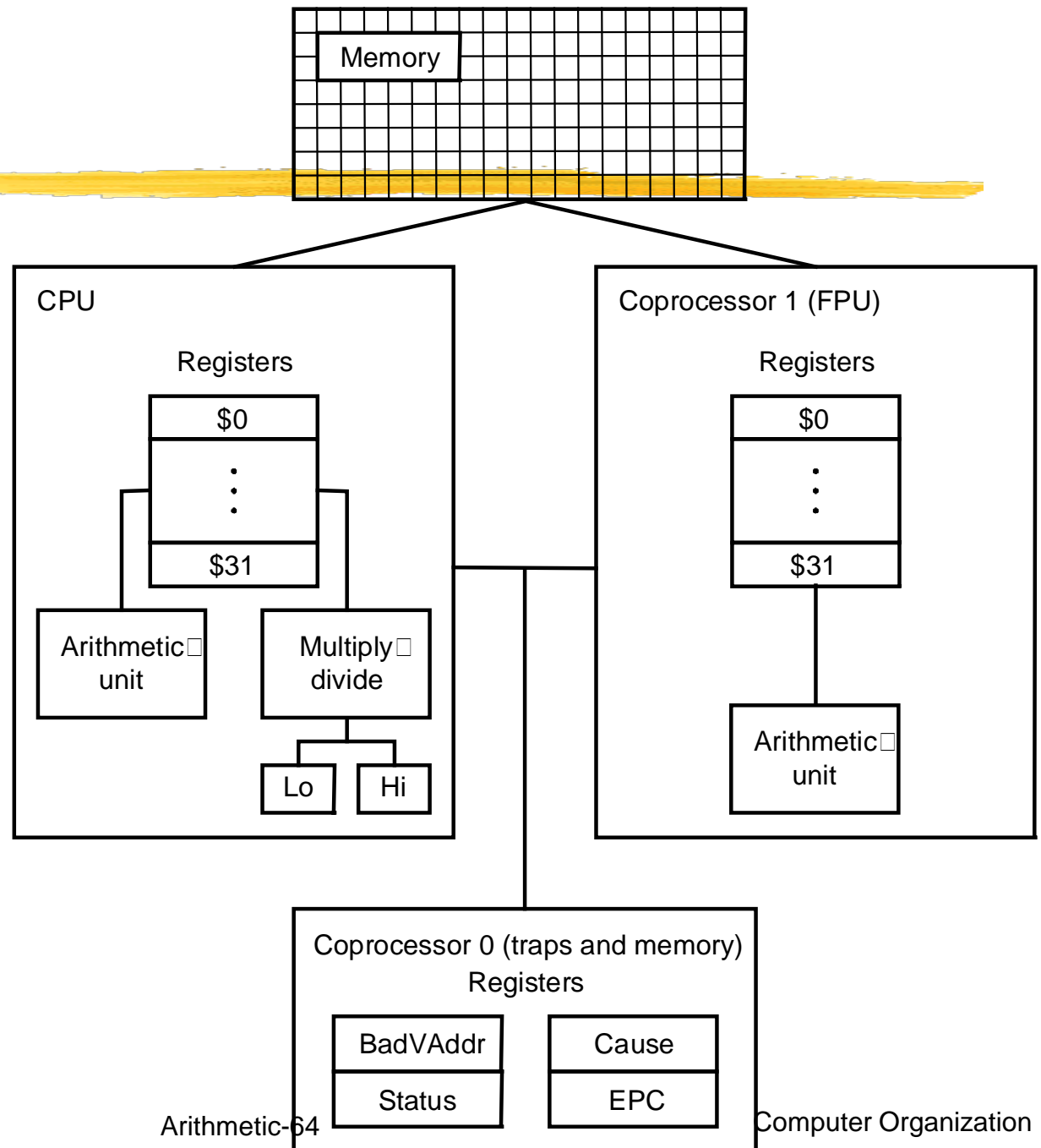


Carry Save Adder

Outline

- ◆ Constructing an arithmetic logic unit (Appendix C)
- ◆ Multiplication (Sec. 3.3, Appendix C)
- ◆ Division (Sec. 3.4)
- ◆ Floating point (Sec. 3.5)

MIPS R2000 Organization



Division in MIPS

`div $t1, $t2 # t1 / t2`

- ◆ Quotient stored in LO, remainder in HI

`mflo $t3 #copy quotient to t3`

`mfhi $t4 #copy remainder to t4`

- ◆ 3-step process

- ◆ Unsigned division:

`divu $t1, $t2 # t1 / t2`

- Just like `div`, except now interpret `t1`, `t2` as unsigned integers instead of signed
 - Answers are also unsigned, use `mfhi`, `mflo` to access
- ◆ No overflow or divide-by-0 checking
 - Software must perform checks if required

Divide: Paper & Pencil

		1001 _{ten}	Quotient
Divisor	1000 _{ten}	1001010 _{ten}	Dividend
		-1000	
		0010	
		0101	
		1010	
		-1000	
		10 _{ten}	Remainder

- ◆ See how big a number can be subtracted, creating quotient bit on each step
- ◆ Binary
 - 0 => place 0 (0 × divisor)
 - 1 => place a copy (1 × divisor)
- ◆ Two versions of divide, successive refinement
- ◆ Both dividend and divisor are positive integers

Divide Hardware (Version 1)

- ◆ 64-bit *Divisor* register (initialized with 32-bit divisor in left half), 64-bit ALU, 64-bit *Remainder* register (initialized with 64-bit dividend), 32-bit *Quotient* register

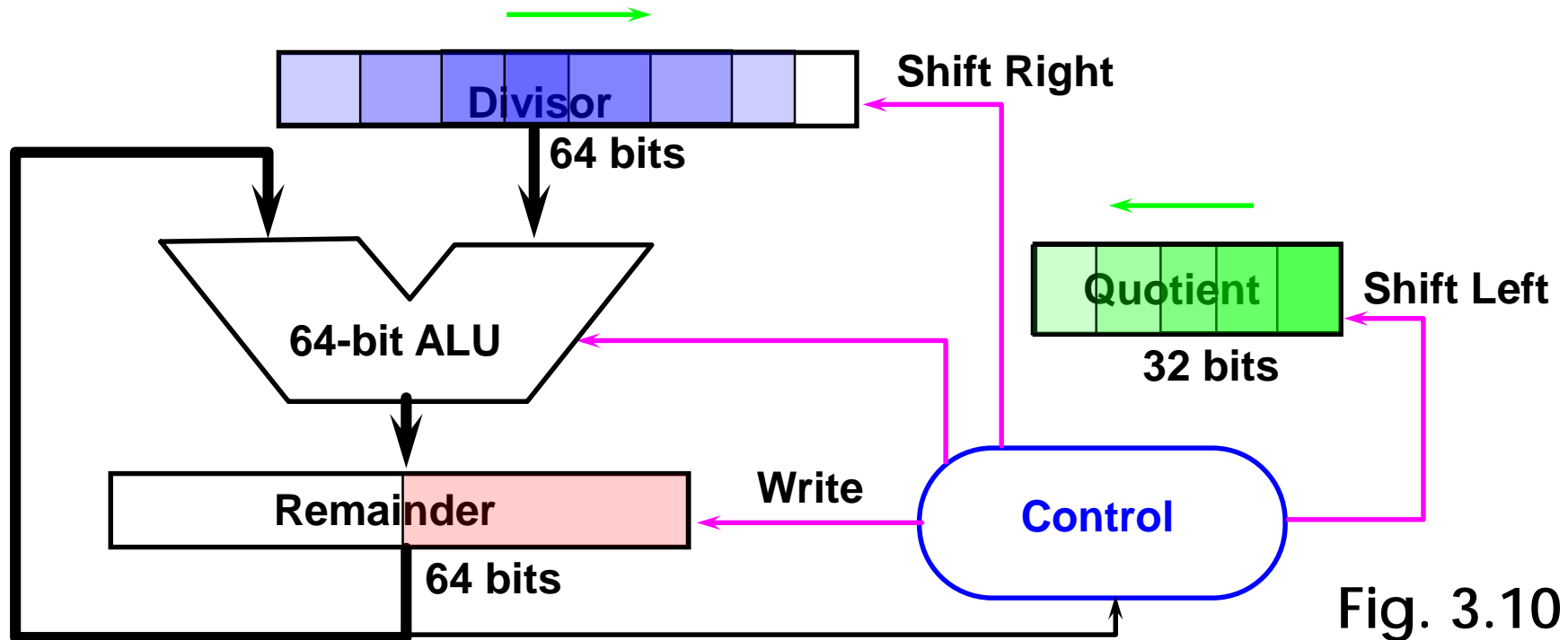


Fig. 3.10

Divide Algorithm (Version 1)

Quot.	Divisor	Rem.
0000	<u>00100000</u>	00000111 11100111 00000111
0000	<u>00010000</u>	00000111 11110111 00000111
0000	<u>00001000</u>	00000111 11111111 00000111
0000	<u>00000100</u>	00000111 00000011 00000011
0001		00000011 00000001 00000001
0001	<u>00000010</u>	00000011 00000001 00000001
0011		00000001 00000000 00000000
0011	<u>00000001</u>	00000001 00000000 00000000

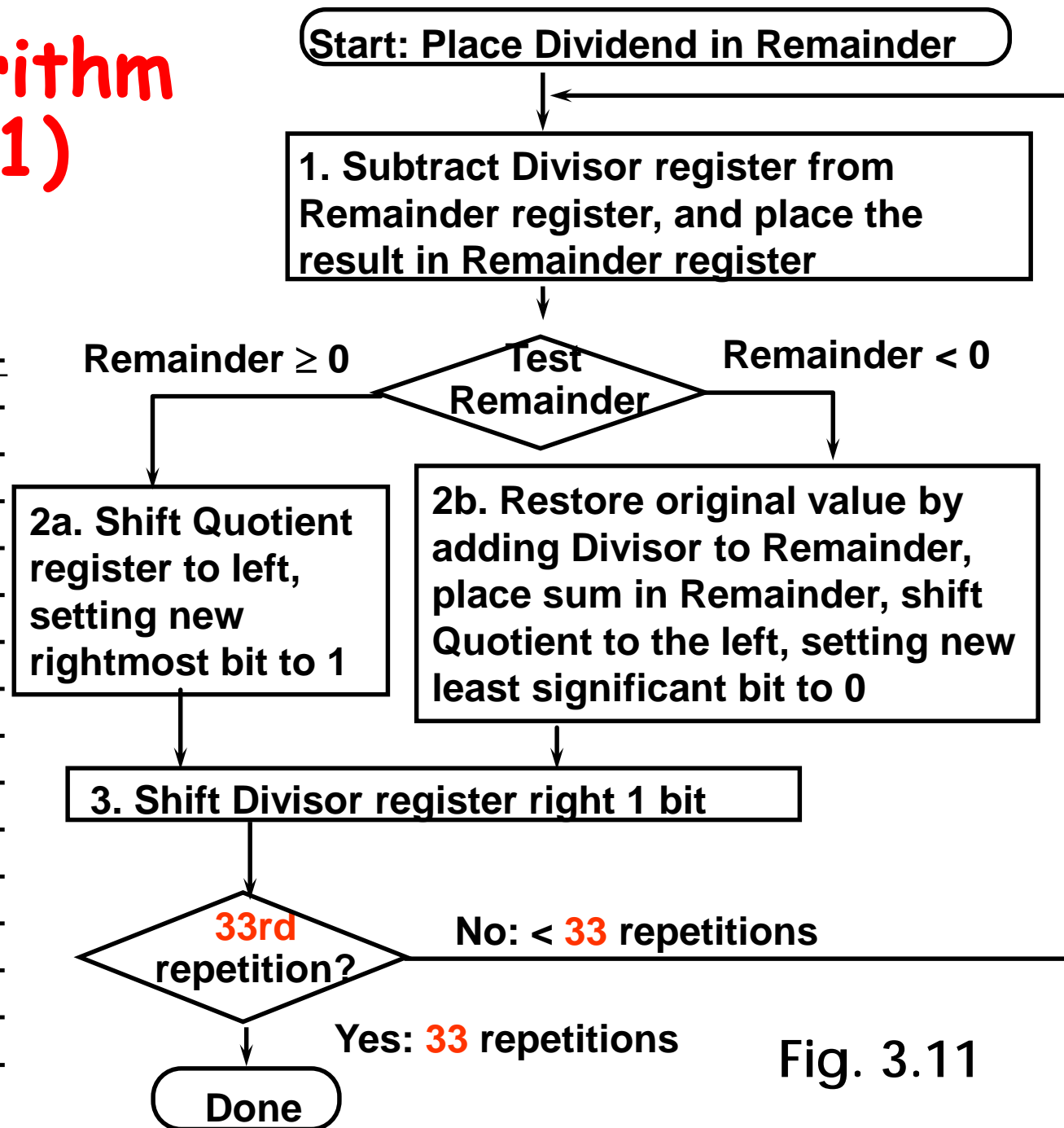


Fig. 3.11

Observations: Divide Version 1

- ◆ Half of the bits in divisor register always 0
 - => 1/2 of 64-bit adder is wasted
 - => 1/2 of divisor is wasted
- ◆ Instead of shifting divisor to right, shift remainder to left?
- ◆ 1st step cannot produce a 1 in quotient bit (otherwise quotient is too big for the register)
 - => switch order to shift first and then subtract
 - => save 1 iteration
- ◆ Eliminate Quotient register by combining with Remainder register as shifted left

Divide Hardware (Version 2)

- ◆ 32-bit Divisor register, 32-bit ALU, 64-bit Remainder register, (0-bit Quotient register)

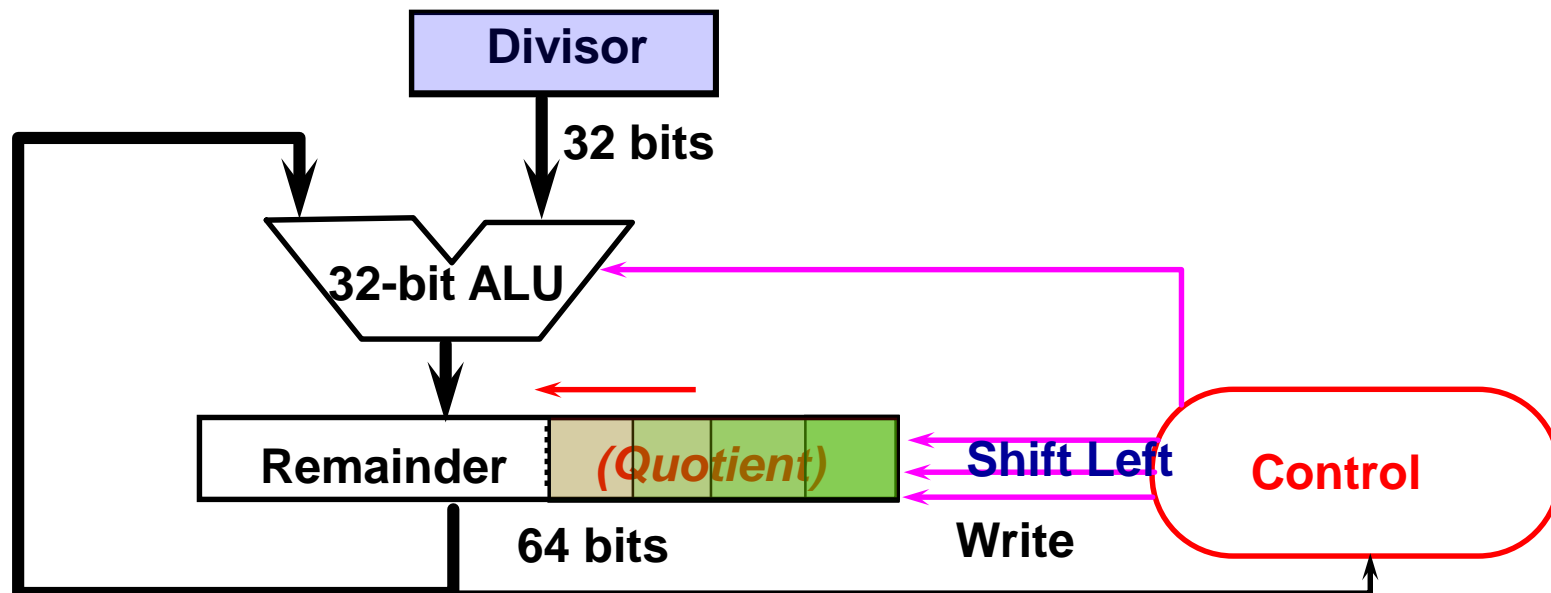
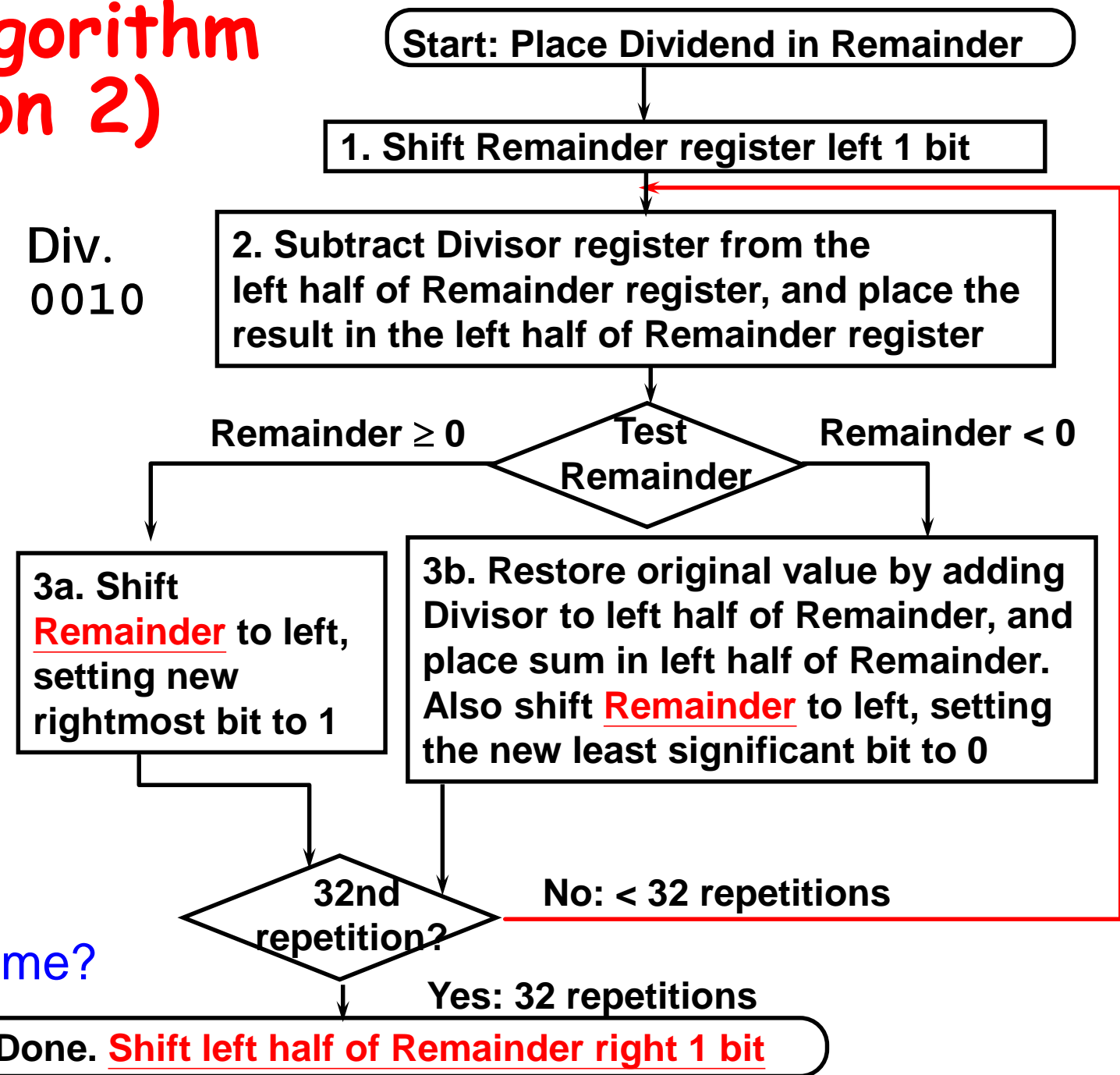


Fig. 3.13

Divide Algorithm (Version 2)

Step	Remainder	Div.
0	0000	0111 0010
1.1	0000	1110
1.2	1110	1110
1.3b	<u>0001</u>	1100
2.2	1111	1100
2.3b	<u>0011</u>	1000
3.2	0001	1000
3.3a	<u>0011</u>	0001
4.2	0001	0001
4.3a	<u>0010</u>	0011
	0001	0011



Is it correct all the time?

Divide

◆ Signed Divides:

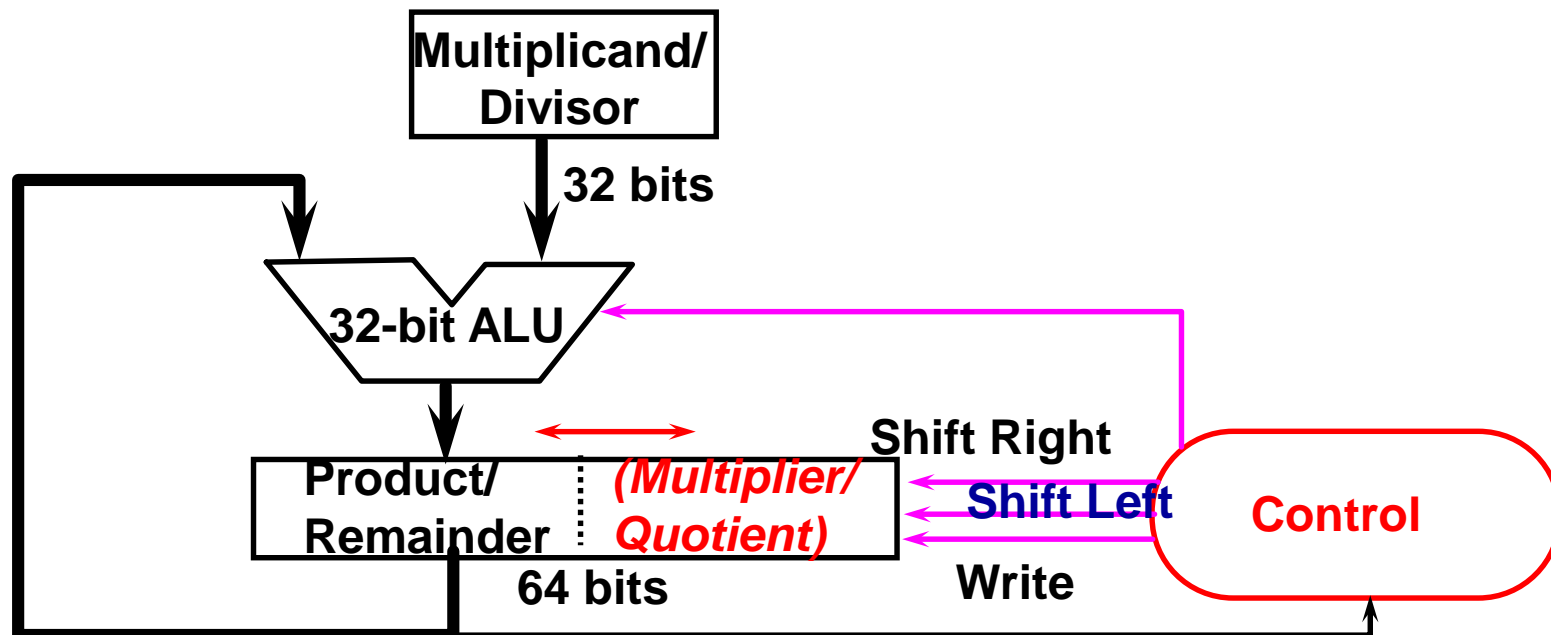
- Remember signs, make positive, complement quotient and remainder if necessary
- Let Dividend and Remainder have same sign and negate Quotient if Divisor sign & Dividend sign disagree,
- Ex: $-7 \div 2 = -3$, remainder = -1
 $-7 \div -2 = 3$, remainder = -1
- Satisfy Dividend = Quotient x Divisor + Remainder

Observations: Multiply and Divide

- ◆ Same hardware as multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- ◆ HI and LO registers in MIPS combine to act as 64-bit register for multiply and divide

Multiply/Divide Hardware

- ◆ 32-bit Multiplicand/Divisor register, 32-bit ALU, 64-bit Product/Remainder register, (0-bit Multiplier/Quotient register)



MIPS Multiply/Divide Summary

◆ Start multiply, divide

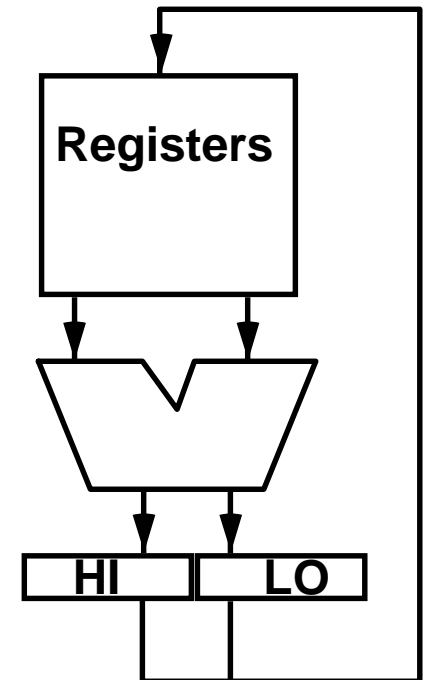
- MULT rs, rt $HI-LO = rs \times rt$ // 64-bit signed
- MULTU rs, rt $HI-LO = rs \times rt$ // 64-bit unsigned
- DIV rs, rt $LO = rs \div rt$; $HI = rs \% rt$ (%: mod)
- DIVU rs, rt // 64-bit unsigned

◆ Move result from multiply, divide

- MFHI rd $rd = HI$
- MFLO rd $rd = LO$

◆ Move to HI or LO

- MTHI rd $HI = rd$
- MTLO rd $LO = rd$



Outline

- ◆ Constructing an arithmetic logic unit (Appendix C)
- ◆ Multiplication (Sec. 3.3, Appendix C)
- ◆ Division (Sec. 3.4)
- ◆ Floating point (Sec. 3.5)

Floating Point: Motivation

◆ What can be represented in N bits?

Unsigned	0	to	$2^n - 1$
2's Complement	-2^{n-1}	to	$2^{n-1} - 1$
1's Complement	$-2^{n-1} + 1$	to	$2^{n-1} - 1$
Excess M	$-M$	to	$2^n - M - 1$

◆ But, what about ...

- very large numbers?

9,349,398,989,787,762,244,859,087,678

- very small numbers?

0.000000000000000000000000000045691

- rationals $\frac{2}{3}$

- irrationals $\sqrt{2}$

- transcendentals e, π

Floating Point: Example

♦ Floating Point

- $A = 31.48$

- $3 \rightarrow 3 \times 10^1$

- $1 \rightarrow 1 \times 10^0$

- $4 \rightarrow 4 \times 10^{-1}$

- $8 \rightarrow 8 \times 10^{-2}$

♦ Scientific notation

- $A = 3.148 \times 10^1$

- $3 \rightarrow 3 \times 10^0 \times 10^1$

- $1 \rightarrow 1 \times 10^{-1} \times 10^1$

- $4 \rightarrow 4 \times 10^{-2} \times 10^1$

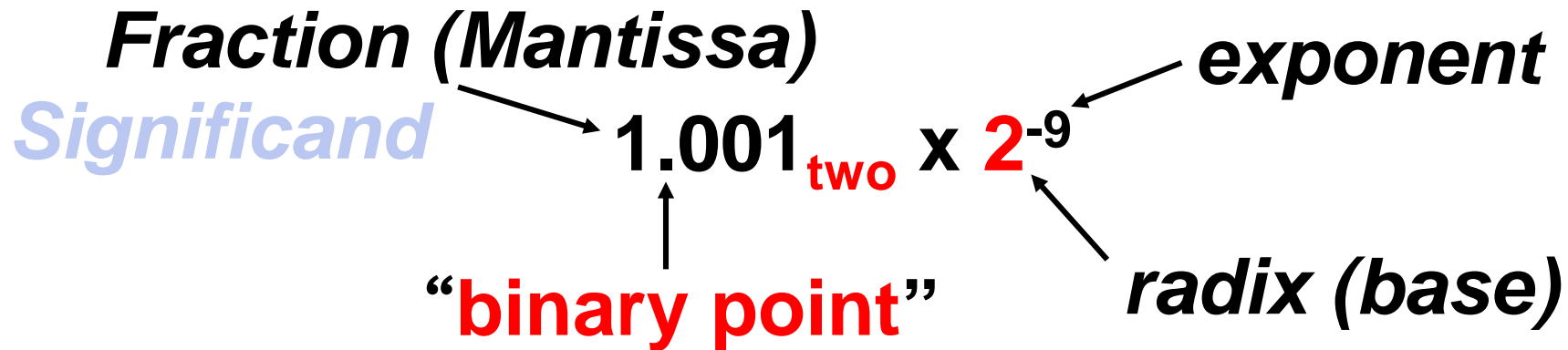
- $8 \rightarrow 8 \times 10^{-3} \times 10^1$

Scientific Notation: Decimal

Fraction (Mantissa)
Significand → 3.5_{ten} × 10⁻⁹ ← *exponent*
↑
“*decimal point*” ← *radix (base)*

- ◆ Normalized form: no leading 0s
(exactly one digit to left of decimal point)
- ◆ Alternatives to represent 0.0000000035
 - Normalized: 3.5×10^{-9}
 - Not normalized: 0.35×10^{-8} , 35.0×10^{-10}

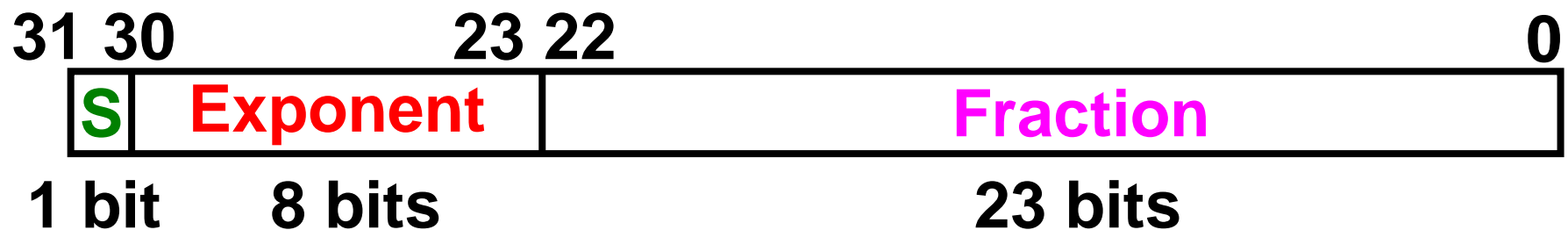
Scientific Notation: Binary



- ◆ Computer arithmetic that supports it is called floating point, because the binary point is not fixed, as it is for integers
- ◆ Normalized form: no leading 0s (exactly one digit to left of binary point)
- ◆ Alternatives to represent $1/2^9 + 1/2^{12}$
 - Normalized: 1.001×2^{-9}
 - Not normalized: 0.1001×2^{-8} , 10.01×2^{-10}

FP Representation

- ◆ Normalized format: $\pm 1.xxxxxxxxxx_{\text{two}} \times 2^{\pm yyy_{\text{two}}}$
- ◆ Want to put it into multiple words: 32 bits for *single-precision* and 64 bits for *double-precision*
- ◆ A simple single-precision representation:



S represents sign

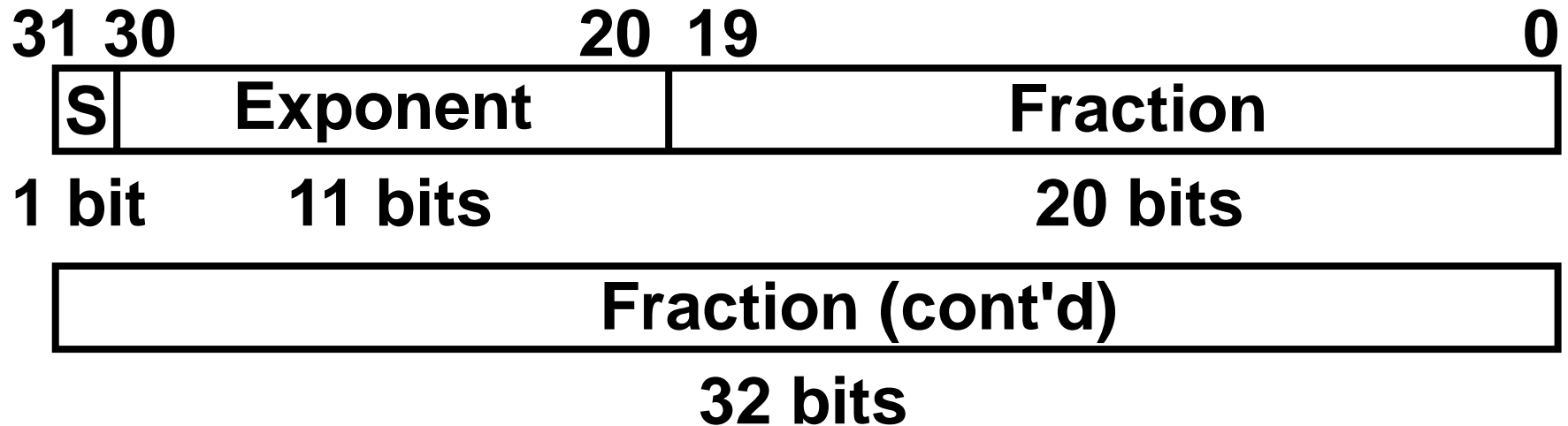
Exponent represents y's

Fraction represents x's

- ◆ Represent numbers as small as $\sim 1.2 \times 10^{-38}$ to as large as $\sim 3.4 \times 10^{38}$

Double Precision Representation

- ◆ Next multiple of word size (64 bits)



- ◆ Double precision (vs. single precision)
 - Represent numbers almost as small as $\sim 2.2 \times 10^{-308}$ to almost as large as $\sim 1.8 \times 10^{308}$
 - But primary advantage is greater accuracy due to larger fraction

IEEE 754 Standard (1/4)

- ◆ Regarding single precision, DP similar
- ◆ Sign bit:
 - 1 means negative
 - 0 means positive
- ◆ Fraction:
 - To pack more bits, leading 1 implicit for normalized numbers (hidden leading 1 bit)
 - 1 + 23 bits single, 1 + 52 bits double
 - always true: $0 \leq \text{Fraction} < 1$
(for normalized numbers)
- ◆ Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Standard (2/4)

◆ Exponent:

- Need to represent positive and negative exponents
- Also want to compare FP numbers as if they were integers, to help in value comparisons
- How about using 2's complement to represent?
Ex: 1.0×2^{-1} versus $1.0 \times 2^{+1}$ (1/2 versus 2)

1/2

0	1111 1111	000 0000 0000 0000 0000 0000
---	-----------	------------------------------

2

0	0000 0001	000 0000 0000 0000 0000 0000
---	-----------	------------------------------

*If we use integer comparison for these two words,
we will conclude that $1/2 > 2!!!$*

Biased (Excess) Notation

◆ Biased 7

0000	-7
0001	-6
0010	-5
0011	-4
0100	-3
0101	-2
0110	-1
0111	0
1000	1
1001	2
1010	3
1011	4
1100	5
1101	6
1110	7
1111	8

IEEE 754 Standard (3/4)

- ◆ Instead, let notation 0000 0000 be most negative, and 1111 1111 most positive
- ◆ Called biased notation, where bias is the number subtracted to get the real number
 - IEEE 754 uses bias of 127 for single precision: Subtract 127 from Exponent field to get actual value for exponent
 - 1023 is bias for double precision

$$126 - 127 = -1$$

1/2

0	0111 1110	000 0000 0000 0000 0000 0000
---	-----------	------------------------------

2

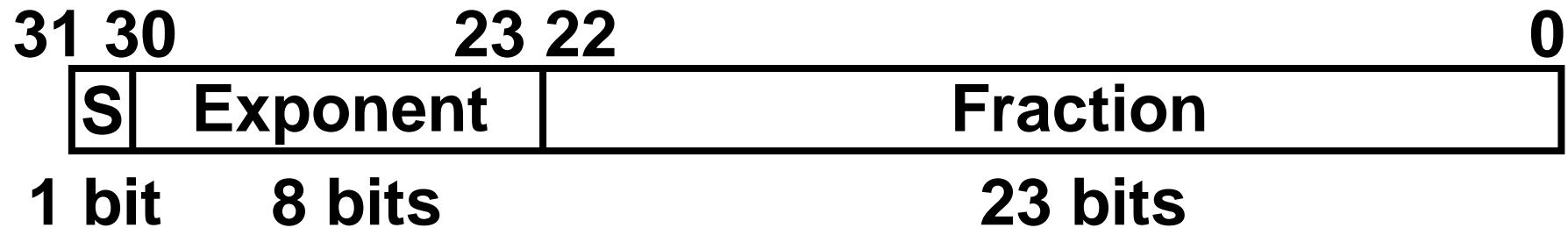
0	1000 0000	000 0000 0000 0000 0000 0000
---	-----------	------------------------------

$$128 - 127 = 1$$

We can use integer comparison for floating point comparison.

IEEE 754 Standard (4/4)

- ◆ Summary (single precision):



$$(-1)^S \times (1.\text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

- ◆ Double precision are same, except with exponent bias of 1023

Example: FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

- ◆ Sign: 0 => positive
- ◆ Exponent:
 - $0110\ 1000_{\text{two}} = 104_{\text{ten}}$
 - Bias adjustment: $104 - 127 = -23$
- ◆ Fraction:
 - $1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$
 $= 1.0 + 0.666115$
- ◆ Represents: $1.666115_{\text{ten}} \times 2^{-23} \approx 1.986 \times 10^{-7}$

Example 1: Decimal to FP

- ◆ Number = - 0.75
 - = - $0.11_{\text{two}} \times 2^0$ (scientific notation)
 - = - $1.1_{\text{two}} \times 2^{-1}$ (normalized scientific notation)
- ◆ Sign: negative => 1
- ◆ Exponent:
 - Bias adjustment: $-1 + 127 = 126$
 - $126_{\text{ten}} = 0111\ 1110_{\text{two}}$

1	0111 1110	100 0000 0000 0000 0000
---	-----------	-------------------------

Example 2: Decimal to FP

◆ A more difficult case: representing $1/3$?

$$= 0.33333..._{10} = 0.0101010101..._2 \times 2^0$$

$$= 1.0101010101..._2 \times 2^{-2}$$

- Sign: 0

- Exponent = $-2 + 127 = 125_{10} = 01111101_2$

- Fraction = 0101010101...

0	0111 1101	0101 0101 0101 0101 0101 010
---	-----------	------------------------------

Single-Precision Range

- ◆ Exponents 00000000 and 11111111 reserved
- ◆ Smallest value
 - Exponent: 00000001
 \Rightarrow actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- ◆ Largest value
 - exponent: 11111110
 \Rightarrow actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 \Rightarrow **significand ≈ 2.0**
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- ◆ Exponents 0000...00 and 1111...11 reserved
- ◆ Smallest value
 - Exponent: 000000000001
 \Rightarrow actual exponent = $1 - 1023 = -1022$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- ◆ Largest value
 - Exponent: 111111111110
 \Rightarrow actual exponent = $2046 - 1023 = +1023$
 - Fraction: 111...11 \Rightarrow **significand ≈ 2.0**
 - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- ◆ Relative precision
 - all fraction bits are significant
 - Single: approx 2^{-23}
 - Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
 - Double: approx 2^{-52}
 - Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Zero and Special Numbers

- ◆ What have we defined so far? (single precision)

<u>Exponent</u>	<u>Fraction</u>	<u>Object</u>
0	0	<u>???</u>
0	nonzero	<u>???</u>
1-254	anything	+/- floating-point
255	0	<u>???</u>
255	nonzero	<u>???</u>

Representation for 0

◆ Represent 0?

- Exponent: all zeroes
- Fraction: all zeroes, too
- What about sign?
- +0: 0 00000000 000000000000000000000000
- -0: 1 00000000 000000000000000000000000

◆ Why two zeroes?

- Helps in some limit comparisons

Special Numbers

- ◆ What have we defined so far? (single precision)

<u>Exponent</u>	<u>Fraction</u>	<u>Object</u>
0	0	0
0	nonzero	<u>???</u>
1-254	anything	+/- floating-point
255	0	<u>???</u>
255	nonzero	<u>???</u>

- ◆ Range:

$$1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$$

What if result too small? (>0 , $< 1.2 \times 10^{-38}$ => Underflow!)

$$(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$$

What if result too large? ($> 3.4 \times 10^{38}$ => Overflow!)

Gradual Underflow

- ◆ Represent denormalized numbers (denorms)
 - Exponent : all zeroes
 - Fraction : non-zeroes
 - Allow a number to degrade in significance until it become 0 (gradual underflow)
- The smallest normalized number
 - $1.0000\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{-126}$
- The smallest de-normalized number
 - $0.0000\ 0000\ 0000\ 0000\ 0000\ 001 \times 2^{-126}$

Special Numbers

- ◆ What have we defined so far? (single precision)

Exponent

0

0

1-254

255

255

Fraction

0

nonzero

anything

0

nonzero

Object

0

denorm

+/- floating-point

???

???

Representation for +/- Infinity

- ◆ In FP, divide by zero should produce +/- infinity, not overflow
- ◆ Why?
 - OK to do further computations with infinity
Ex: $X/0 > Y$ may be a valid comparison
- ◆ IEEE 754 represents +/- infinity
 - Most positive exponent reserved for infinity
 - Fractions all zeroes

S	1111 1111	0000 0000 0000 0000 0000 000
---	-----------	------------------------------

Special Numbers (cont'd)

- ◆ What have we defined so far? (single-precision)

Exponent

0

0

1-254

255

255

Fraction

0

nonzero

anything

0

nonzero

Object

0

denom

+/- fl. pt. #

+/- infinity

???

Representation for Not a Number

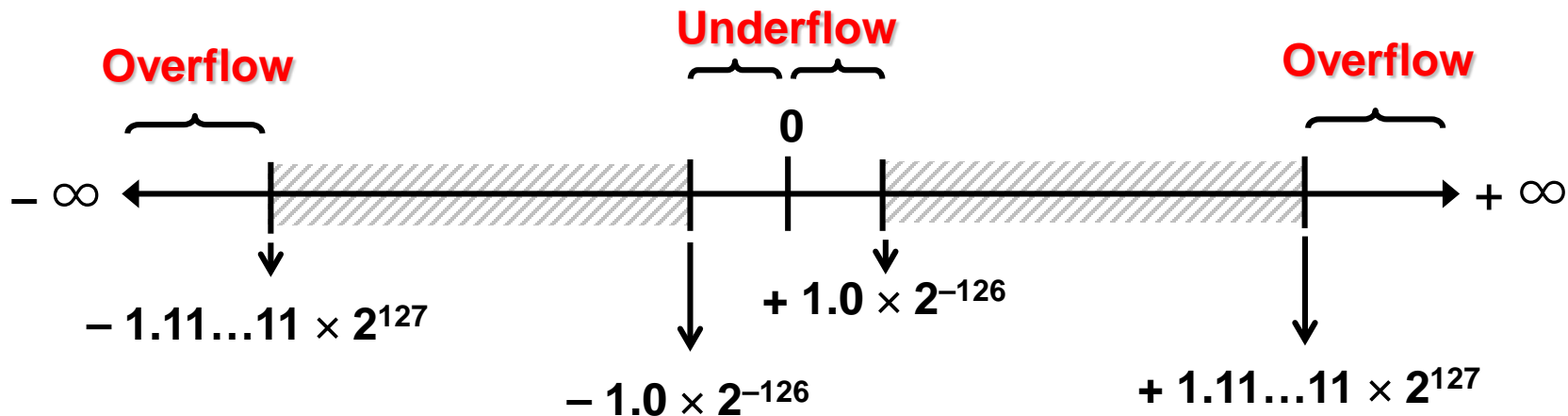
- ◆ What do I get if I calculate $\text{sqrt}(-4.0)$ or $0/0$?
 - If infinity is not an error, these should not be either
 - They are called *Not a Number* (NaN)
 - Exponent = 255, fraction nonzero
- ◆ Why is this useful?
 - Hope NaNs help with debugging?
 - They contaminate: $\text{op}(\text{NaN}, X) = \text{NaN}$
 - OK if calculate but don't use it

Special Numbers (cont'd)

- ◆ What have we defined so far? (single-precision)

<u>Exponent</u>	<u>Fraction</u>	<u>Object</u>
0	0	0
0	nonzero	denom
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN

Range of Single Precision Floating Point Number



Adapted from Prof. Tseng's Class Material

Decimal Addition

- ◆ $A = 3.71345 \times 10^2$, $B = 1.32 \times 10^{-4}$, Perform $A + B$

$$\begin{array}{r} 3.71345 \times 10^2 \\ + 0.00000132 \times 10^2 \\ \hline 3.71345132 \times 10^2 \end{array}$$

- ◆ $A = 3.71345 \times 10^2$
- ◆ $B = 1.32 \times 10^{-4} = 0.00000132 \times 10^2$
- ◆ $A + B = (3.71345 + 0.00000132) \times 10^2$

Right shift 2 – (-4) bits

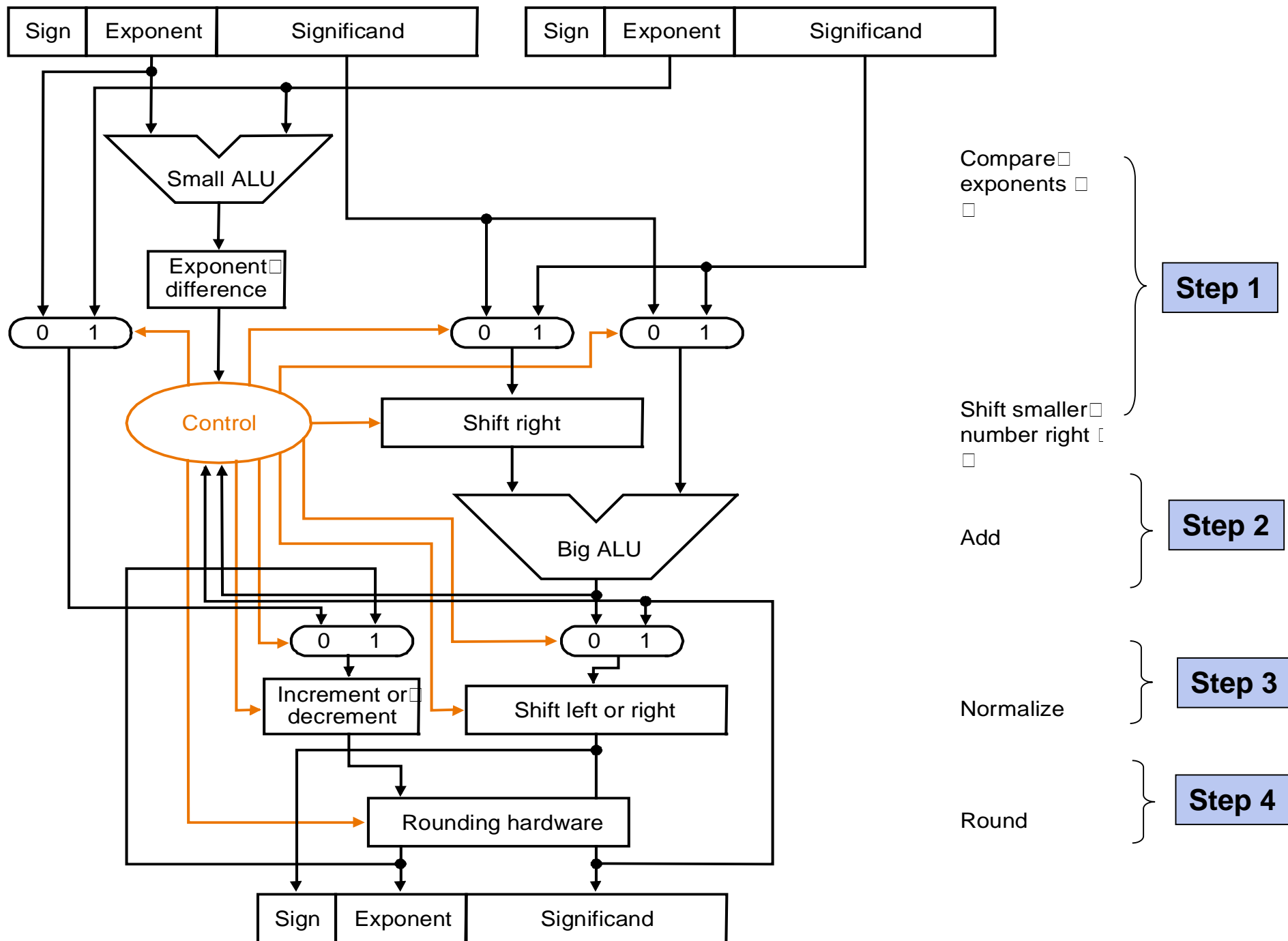
Floating-Point Addition

Basic addition algorithm:

- (1) Align binary point :compute $Y_e - X_e$
 - ◆ right shift the smaller number, say X_m , that many positions to form $X_m \times 2^{X_e - Y_e}$
- (2) Add mantissa: compute $X_m \times 2^{X_e - Y_e} + Y_m$
- (3) Normalization & check for over/underflow if necessary:
 - left shift result, decrement result exponent
 - right shift result, increment result exponent
 - check overflow or underflow during the shift
- (4) Round the mantissa and renormalize if necessary

Floating-Point Addition Example

- ◆ Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + -0.4375)
- ◆ 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- ◆ 2. Add mantissa
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- ◆ 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- ◆ 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625



FP Adder Hardware

- ◆ Much more complex than integer adder
- ◆ Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- ◆ FP adder usually takes several cycles
 - Can be pipelined

Decimal Multiplication

- ◆ $A = 3.12 \times 10^2$, $B = 1.5 \times 10^{-4}$, Perform $A \times B$

$$\begin{array}{r} 3.12 \times 10^2 \\ \times 1.5 \times 10^{-4} \\ \hline 4.68 \times 10^{-2} \end{array}$$

- ◆ $A = 3.12 \times 10^2$
- ◆ $B = 1.5 \times 10^{-4}$
- ◆ $A \times B = (3.12 \times 1.5) \times 10^{(2+(-4))}$

Floating-Point Multiplication

Basic multiplication algorithm

- (1) Add exponents of operands to get exponent of product
doubly biased exponent must be corrected:

$$X_e = 7$$

$$Y_e = -3 \quad X_e = 1111 \quad = 15 \quad = 7 + 8$$

$$\text{Excess } 8 \quad Y_e = 0101 \quad = 5 \quad = -3 + 8$$

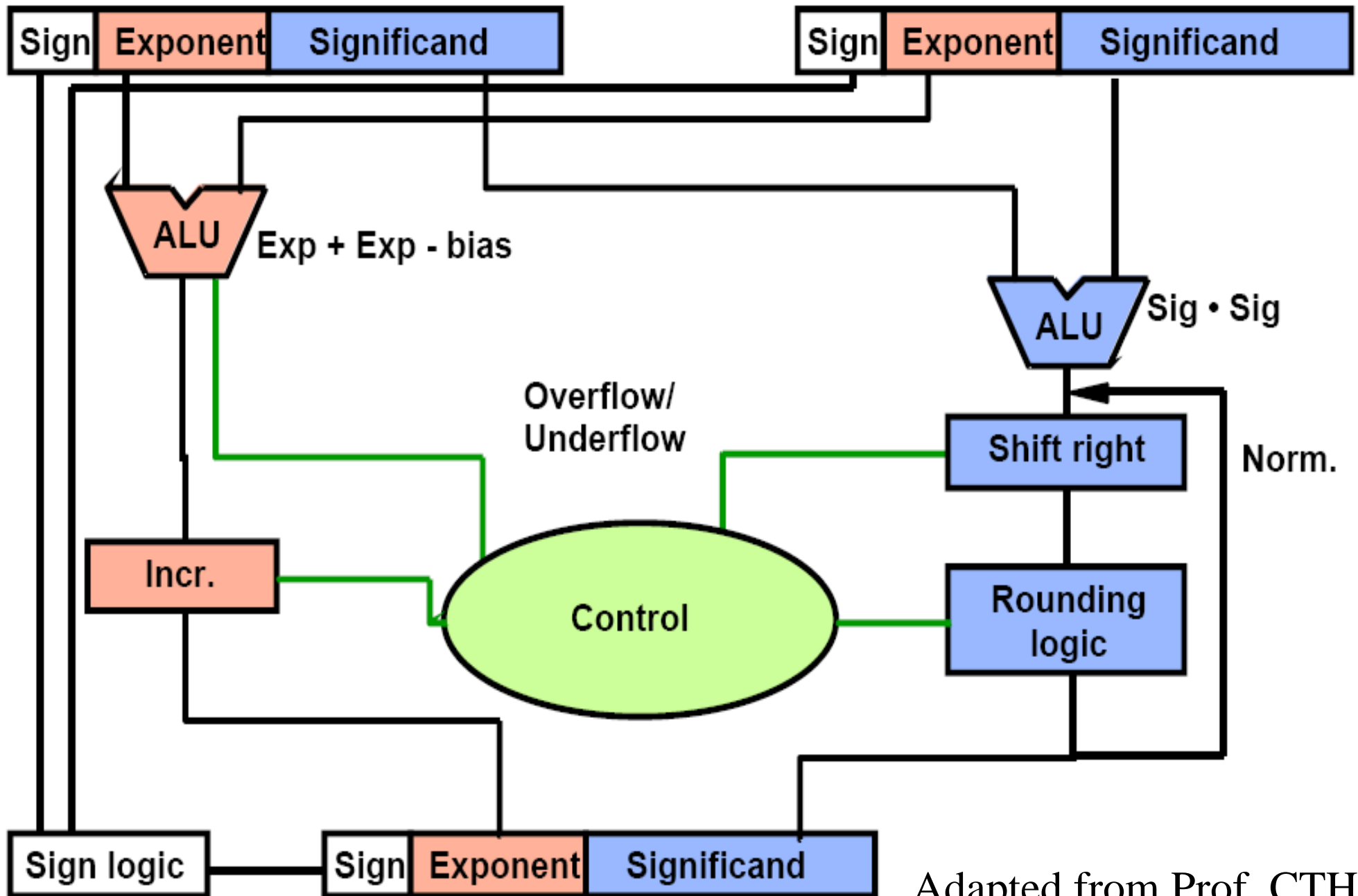
$$10100 \quad 20 \quad 4 + 8 + 8$$

need extra subtraction step of the bias amount

- (2) Multiplication of operand mantissa
- (3) Normalize the product & check overflow or underflow during the shift
- (4) Round the mantissa and renormalize if necessary
- (5) Set the sign of product

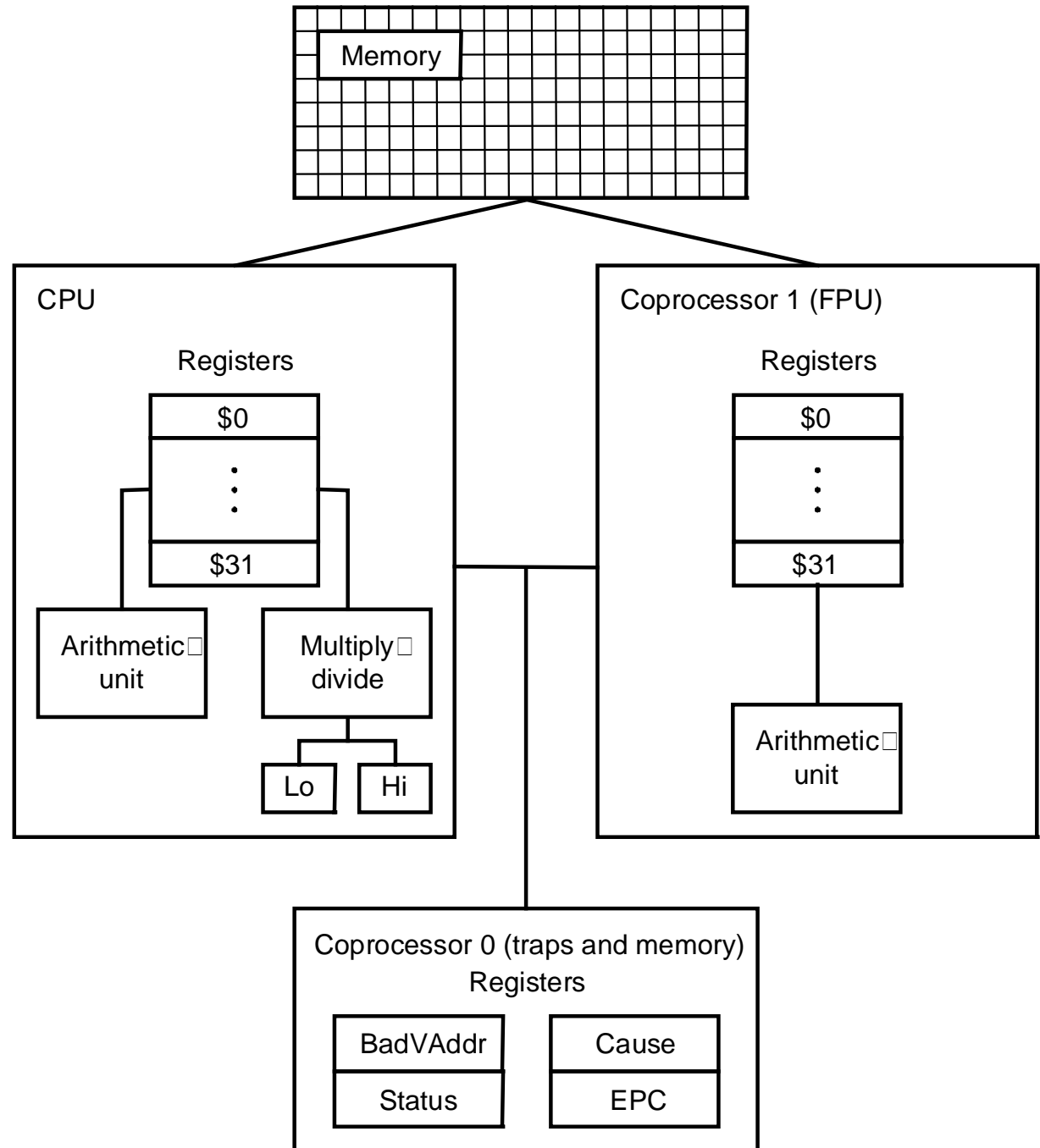
Floating-Point Multiplication Example

- ◆ Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- ◆ 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- ◆ 2. Multiply operand mantissa
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- ◆ 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- ◆ 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- ◆ 5. Determine sign:
 - $-1.110_2 \times 2^{-3} = -0.21875$



Adapted from Prof. CTH

MIPS R2000 Organization



MIPS Floating Point

- ◆ Separate floating point instructions:
 - Single precision: `add.s`, `sub.s`, `mul.s`, `div.s`
 - Double precision: `add.d`, `sub.d`, `mul.d`, `div.d`
- ◆ FP part of the processor:
 - contains 32 32-bit registers: `$f0`, `$f1`, ...
 - most registers specified in `.s` and `.d` instruction refer to this set
 - separate load and store: `lwc1` and `swc1`
 - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0/$f1`, `$f2/$f3`
 - Instructions to move data between main processor and coprocessors:
 - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.

Interpretation of Data

The BIG Picture

- ◆ Bits have no inherent meaning
 - Interpretation depends on the instructions applied
- ◆ Computer representations of numbers
 - Finite range and precision
 - Need to account for this in programs

Associativity

◆ Floating Point add, subtract associative ?

		$(x+y)+z$	$x+(y+z)$
x	-1.50E+38	0.00E+00	-1.50E+38
y	1.50E+38		1.50E+38
z	1.0	1.0	
		1.00E+00	0.00E+00

- ◆ Therefore, Floating Point add, subtract are not associative!
 - Why? FP result approximates real result!
 - This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Associativity in Parallel Programming

- ◆ Parallel programs may interleave operations in unexpected orders
 - Assumptions of associativity may fail
- ◆ Need to validate parallel programs under varying degrees of parallelism

x86 FP Architecture

- ◆ Originally based on 8087 FP coprocessor
 - 8 x 80-bit extended-precision registers
 - Used as a push-down stack
 - Registers indexed from TOS: ST(0), ST(1), ...
- ◆ FP values are 32-bit or 64 in memory
 - Converted on load/store of memory operand
 - Integer operands can also be converted on load/store
- ◆ Very difficult to generate and optimize code
 - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FI LD mem/ST(i) FI ST P mem/ST(i) FLDPI FLD1 FLDZ	FI ADD P mem/ST(i) FI SUB RP mem/ST(i) FI MUL P mem/ST(i) FI DI VRP mem/ST(i) FSQRT FABS FRNDI NT	FI COM P FI UCOM P FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- ◆ Optional variations
 - **I**: integer operand
 - **P**: pop operand from stack
 - **R**: reverse operand order
 - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- ◆ Adds 4 × 128-bit registers
 - Extended to 8 registers in AMD64/EM64T
- ◆ Can be used for multiple FP operands
 - 2 × 64-bit double precision
 - 4 × 32-bit double precision
 - Instructions operate on them simultaneously
 - Single-Instruction Multiple-Data

Right Shift and Division

- ◆ Left shift by i places multiplies an integer by 2^i
- ◆ Right shift divides by 2^i ?
 - Only for unsigned integers
- ◆ For signed integers
 - Arithmetic right shift: replicate the sign bit
 - e.g., $-5 / 4$
 - $11111011_2 \gg 2 = 11111110_2 = -2$
 - Rounds toward $-\infty$
 - c.f. $11111011_2 \ggg 2 = 00111110_2 = +62$

Who Cares About FP Accuracy?

- ◆ Important for scientific code
 - But for everyday consumer use?
 - "My bank balance is out by 0.0002¢!" ☹
- ◆ The Intel Pentium FDIV bug, 1994
 - Recall cost: USD \$500M
 - The market expects accuracy
 - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- ◆ ISAs support arithmetic
 - Signed and unsigned integers
 - Floating-point approximation to reals
- ◆ Bounded range and precision
 - Operations can overflow and underflow
- ◆ MIPS ISA
 - Core instructions: 54 most frequently used
 - 100% of SPECINT, 97% of SPECFP
 - Other instructions: less frequent