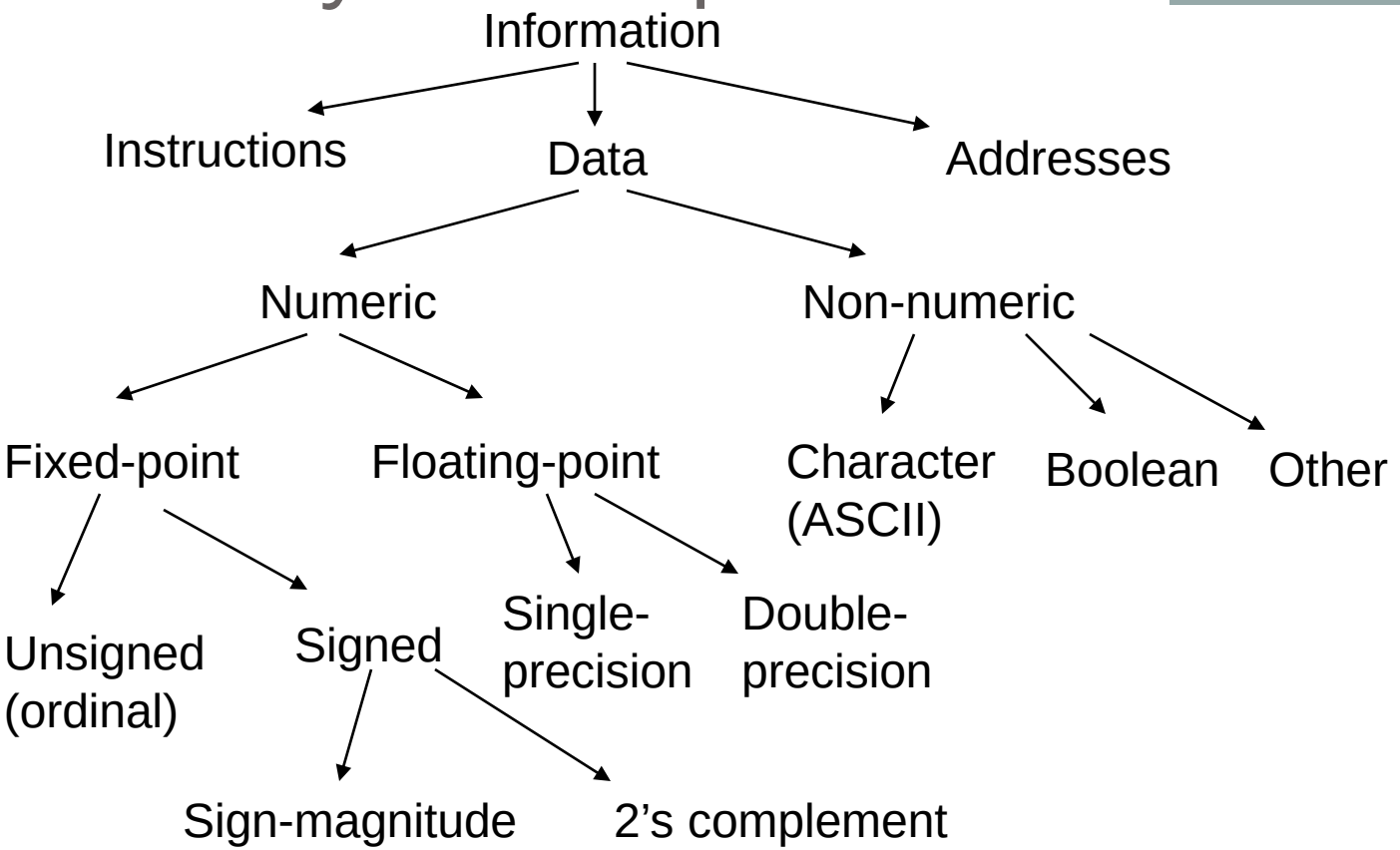
Chapter 3
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Taxonomy of Computer Information



Number Format Considerations

- Type of numbers (integer, fraction, real, complex)
- Range of values
  - between smallest and largest values
  - wider in floating-point formats
- Precision of values (max. accuracy)
  - usually related to number of bits allocated

    n bits => represent 2^n values/levels
  - Value/weight of least-significant bit
- Cost of hardware to store & process numbers (some formats more difficult to add, multiply/divide, etc.)

Unsigned Integers

- Positional number system:

$$a_{n-1}a_{n-2} \ldots a_2a_1a_0 =$$

$$a_{n-1}x2^{n-1} + a_{n-2}x2^{n-2} + \ldots + a_2x2^2 + a_1x2^1 + a_0x2^0$$

- Range = $[(2^n - 1) \ldots 0]$

- Carry out of MSB has weight 2^n

- Fixed-point fraction:

$$0.a_{-1}a_{-2} \ldots a_{-n} = a_{-1}x2^{-1} + a_{-2}x2^{-2} + \ldots + a_{-n}x2^{-n}$$

Signed Integers

- Sign-magnitude format (n-bit values):

  $A = Sa_{n-2} \ldots a_2a_1a_0$ (S = sign bit)

  - Range = $[-(2^{n-1}-1) \ldots +(2^{n-1}-1)]$
  - Addition/subtraction difficult (multiply easy)
  - Redundant representations of 0
- 2's complement format (n-bit values):

  -A represented by $2^n - A$

  - Range = $[-(2^{n-1}) \ldots +(2^{n-1}-1)]$
  - Addition/subtraction easier (multiply harder)
  - Single representation of 0

MIPS

- MIPS architecture uses 32-bit numbers. What is the range of integers (positive and negative) that can be represented?

  Positive integers: 0 to 2,147,483,647

  Negative integers: - 1 to - 2,147,483,648

- What are the binary representations of the extreme positive and negative integers?

  0111 1111 1111 1111 1111 1111 1111 1111 $= 2^{31} - 1 = 2{,}147{,}483{,}647$

  1000 0000 0000 0000 0000 0000 0000 0000 $= -2^{31} = -2{,}147{,}483{,}648$

- What is the binary representation of zero?

  0000 0000 0000 0000 0000 0000 0000 0000

Computing the 2's Complement

To compute the 2's complement of A:

- Let $A = a_{n-1}a_{n-2} \ldots a_2a_1a_0$
- $2^n - A = 2^n - 1 + 1 - A = (2^n - 1) - A + 1$

$$
\begin{array}{l}
\quad 1 \quad 1 \quad \ldots \quad 1 \quad 1 \quad 1 \\
- \; a_{n-1} \; a_{n-2} \; \ldots \; a_2 \; a_1 \; a_0 \quad + 1 \\
\hline
\quad a'_{n-1}a'_{n-2} \ldots a'_2a'_1a'_0 + 1 \quad \text{(one's complement + 1)}
\end{array}
$$

2's Complement Arithmetic

- Let $(2^{n-1}-1) \geq A \geq 0$ and $(2^{n-1}-1) \geq B \geq 0$

- Case 1: A + B
  - $(2^{n}-2) \geq (A + B) \geq 0$
    - Since result $< 2^{n}$, there is no carry out of the MSB
  - Valid result if $(A + B) < 2^{n-1}$
    - MSB (sign bit) = 0
  - Overflow if $(A + B) \geq 2^{n-1}$
    - MSB (sign bit) = 1 if result $\geq 2^{n-1}$
    - Carry into MSB

2's Complement Arithmetic

- Case 2: A - B
  - Compute by adding: A + (-B)
  - 2's complement: A + (2^n - B)
  - $-2^{n-1} <$ result $< 2^{n-1}$ (no overflow possible)
  - If A ≥ B: $2^n + (A - B) \geq 2^n$
    - Weight of adder carry output = 2^n
    - Discard carry (2^n), keeping (A-B), which is ≥ 0
  - If A < B: $2^n + (A - B) < 2^n$
    - Adder carry output = 0
    - Result is 2^n - (B - A) =

      2's complement representation of -(B-A)

2's Complement Arithmetic

- Case 3: -A - B
  - Compute by adding: (-A) + (-B)
  - 2's complement: $(2^n - A) + (2^n - B) = 2^n + 2^n - (A + B)$
  - Discard carry (2^n), making result $2^n - (A + B)$

    $= $ 2's complement representation of $-(A + B)$
  - $0 \geq$ result $\geq -2^n$
  - Overflow if $-(A + B) < -2^{n-1}$
    - MSB (sign bit) = 0 if $2^n - (A + B) < 2^{n-1}$
    - no carry into MSB

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Relational Operators

Compute A-B & test ALU "flags" to compare A vs. B

ZF = result zero OF = 2's complement overflow

SF = sign bit of result CF = adder carry output

	Signed	Unsigned
A = B	ZF = 1	ZF = 1
A <> B	ZF = 0	ZF = 0
A ≥ B	(SF ⊕ OF) = 0	CF = 1 (no borrow)
A > B	(SF ⊕ OF) + ZF = 0	CF · ZF' = 1
A ≤ B	(SF ⊕ OF) + ZF = 1	CF · ZF' = 0
A < B	(SF ⊕ OF) = 1	CF = 0 (borrow)

MIPS Overflow Detection

- An exception (interrupt) occurs when overflow detected for `add,addi,sub`
  - 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*

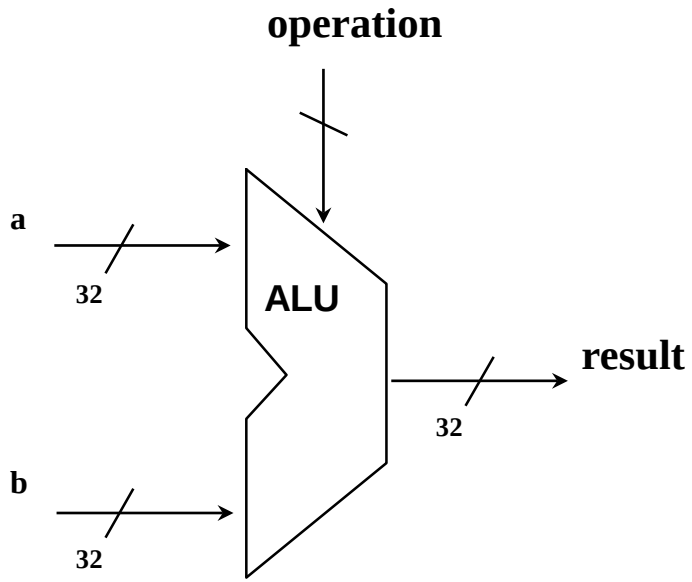
Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., 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

Designing the Arithmetic & Logic Unit (ALU)

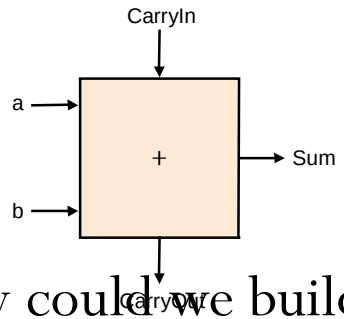
- Provide arithmetic and logical functions as needed by the instruction set
- Consider tradeoffs of area vs. performance

(Material from Appendix B)



Different Implementations

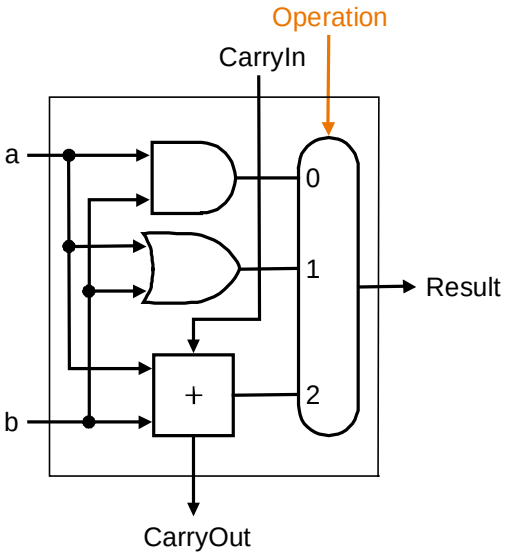
- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate (fan in)
  - Don't want to have to go through too many gates (delay)
  - For our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

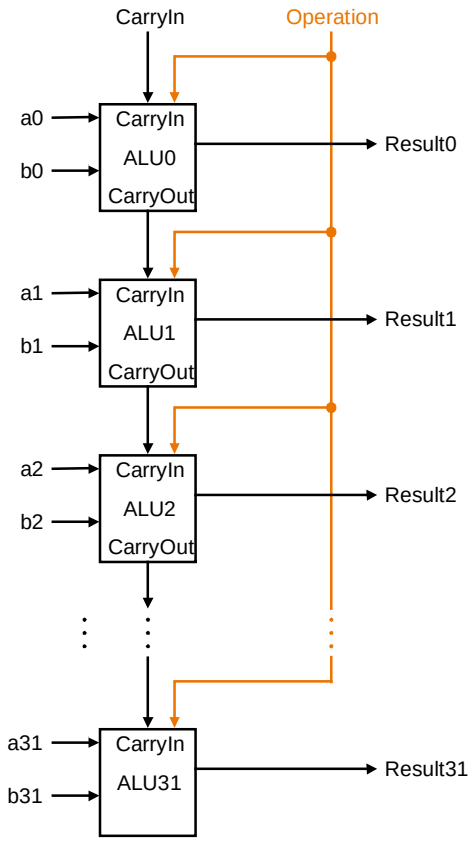


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a\ xor\ b\ xor\ c_{in}$$

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

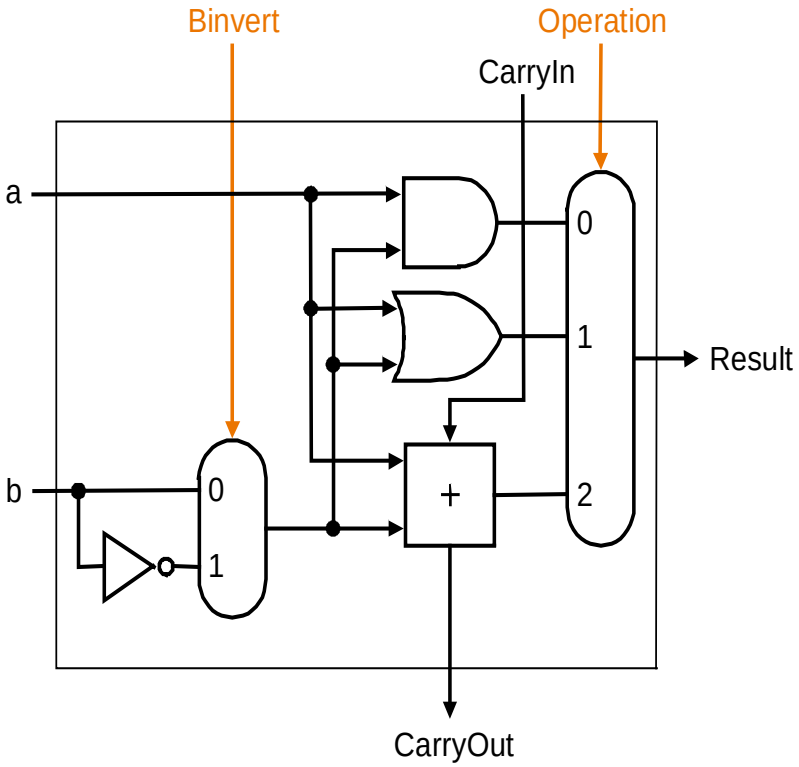
Building a 32 bit ALU





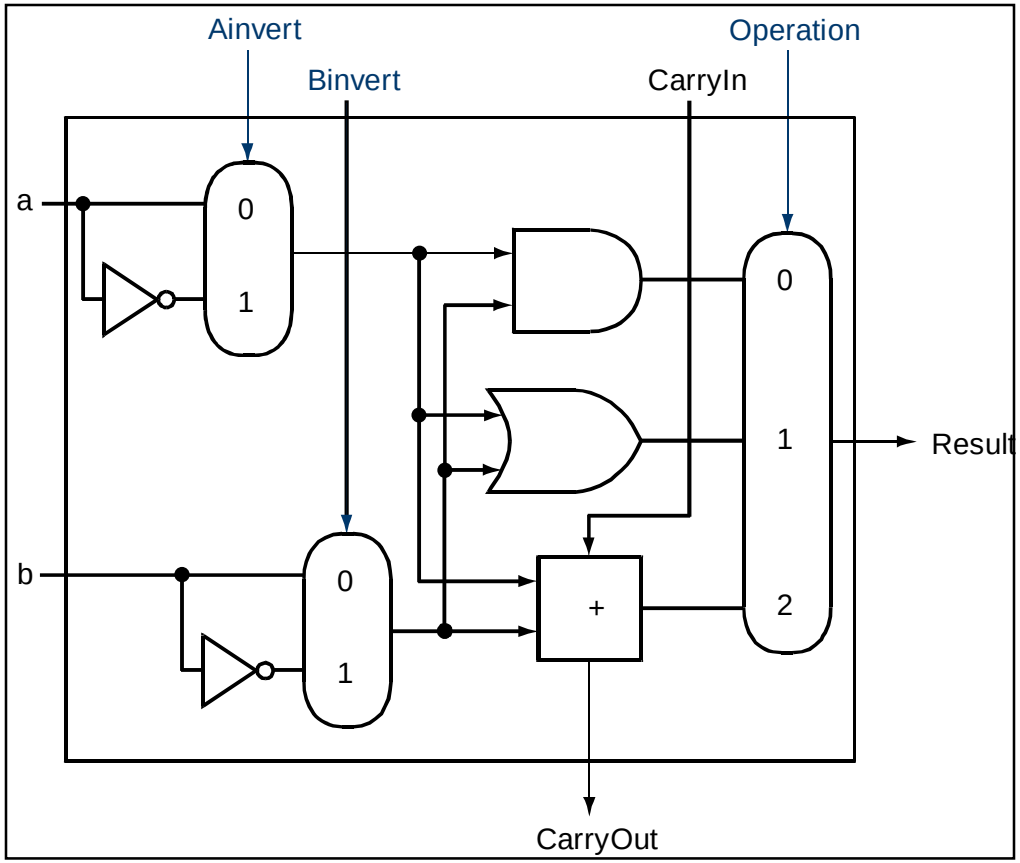
What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.



Adding a NOR function

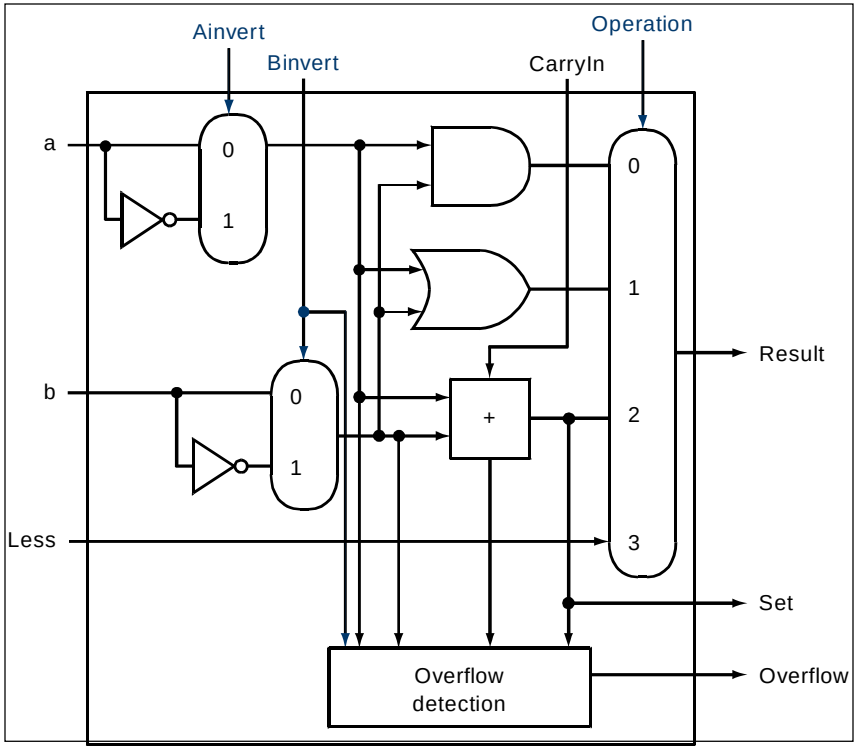
- Can also choose to invert a. How do we get "a NOR b" ?

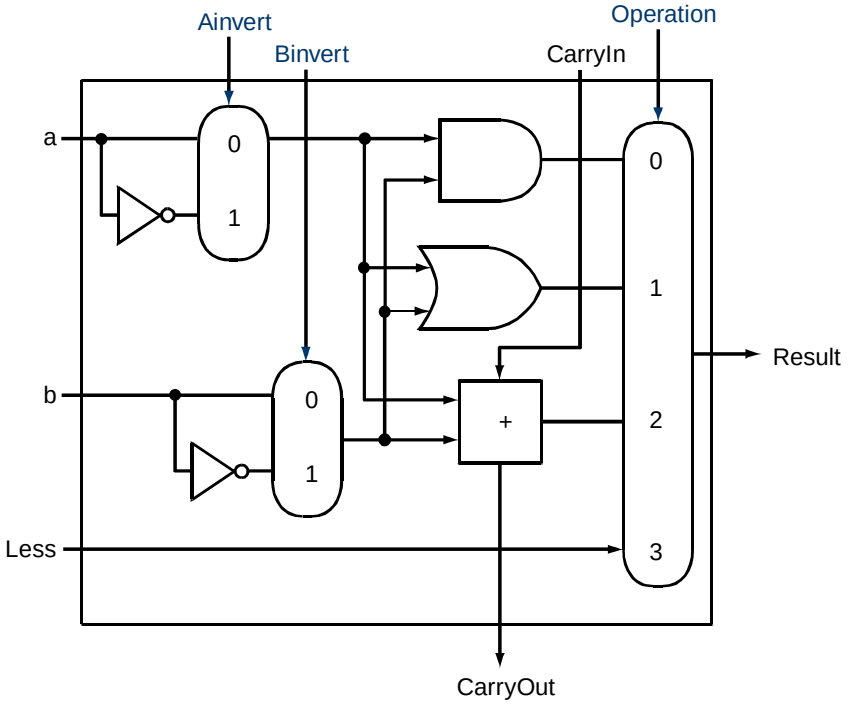


Tailoring the ALU to the MIPS

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

Supporting slt

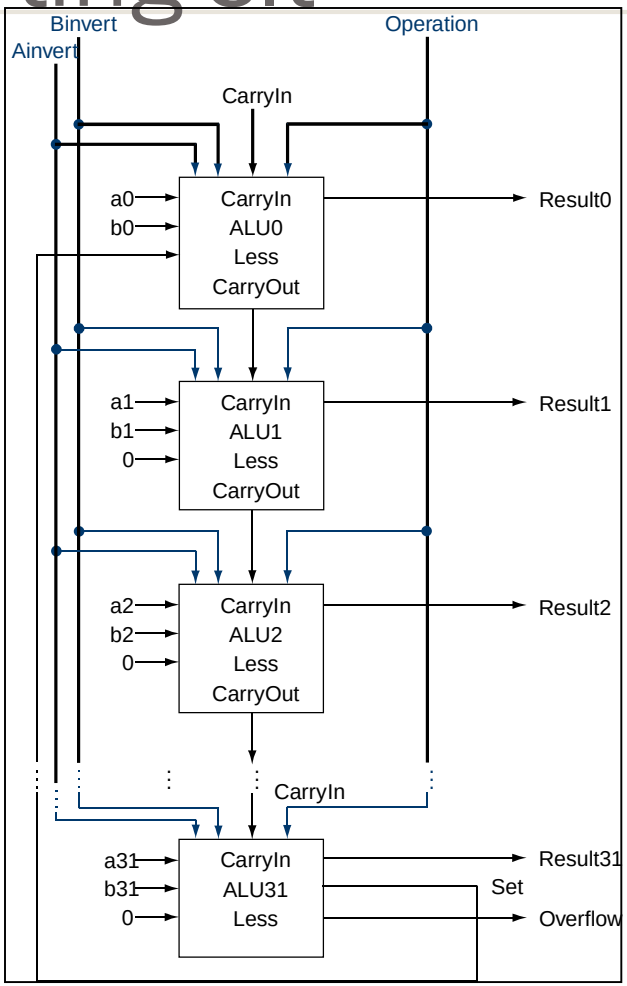




Use this ALU for most significant bit

all other bits

Supporting slt

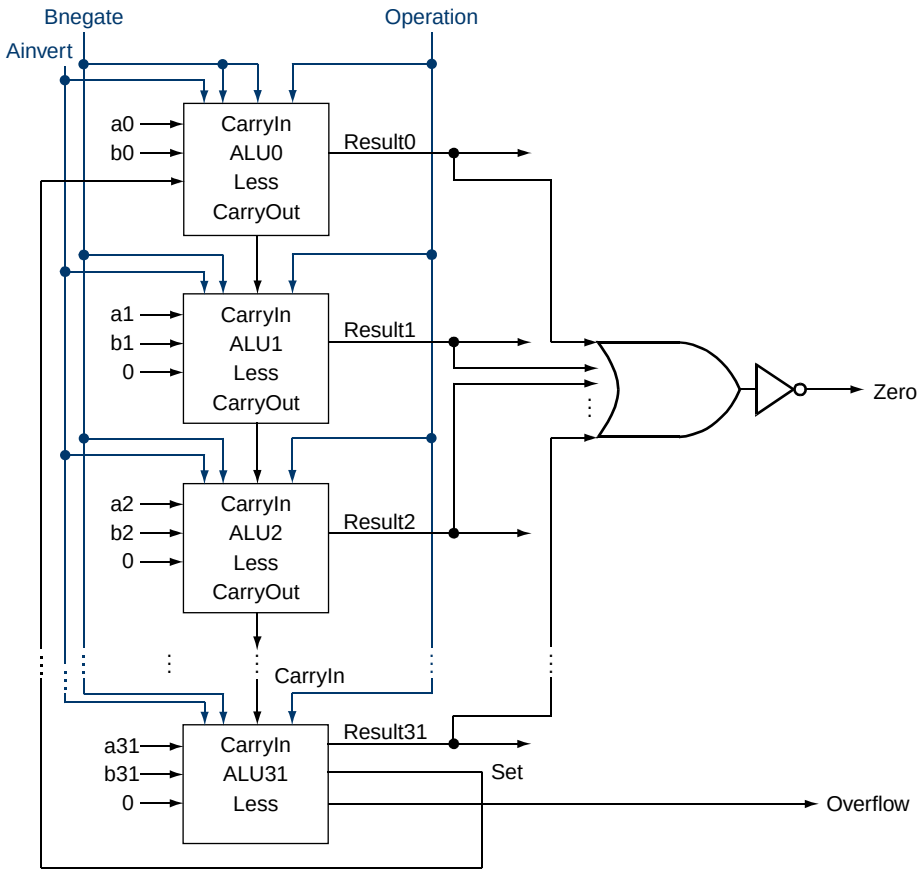


Test for equality

- Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

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



Conclusion

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

Problem: ripple carry adder is slow

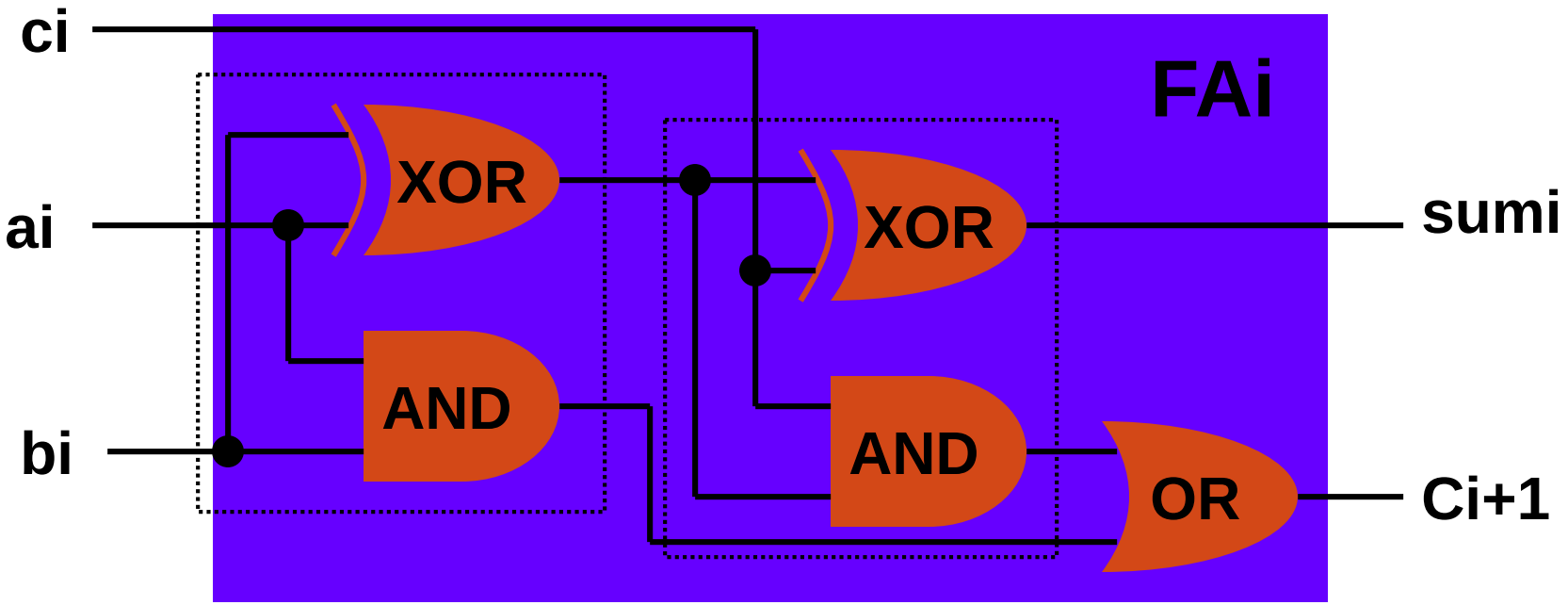
- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

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

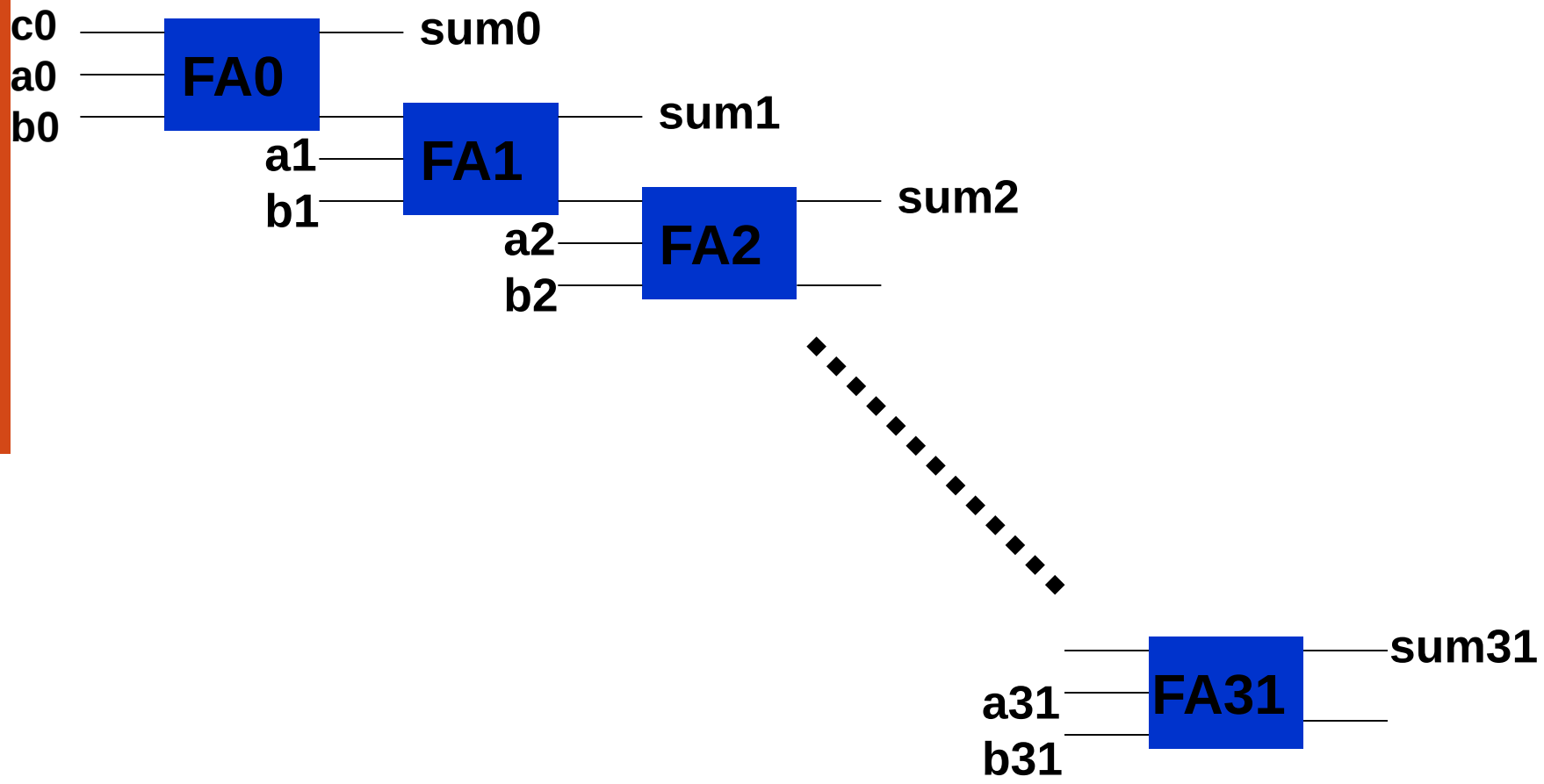
$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \qquad c_2 =$$
$$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \qquad c_3 =$$
$$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \qquad c_4 =$$

Not feasible! Why?

One-bit Full-Adder Circuit



32-bit Ripple-Carry Adder



How Fast is Ripple-Carry Adder?

- Longest delay path (critical path) runs from cin to sum31.
- Suppose delay of full-adder is 100ps.
- Critical path delay = 3,200ps
- Clock rate cannot be higher than $10^{12}/3{,}200 = 312\text{MHz}$.
- Must use more efficient ways to handle carry.

Fast Adders

- In general, any output of a 32-bit adder can be evaluated as a logic expression in terms of all 65 inputs.
- Levels of logic in the circuit can be reduced to $\log_2 N$ for N-bit adder. Ripple-carry has N levels.
- More gates are needed, about $\log_2 N$ times that of ripple-carry design.
- Fastest design is known as carry lookahead adder.

N-bit Adder Design Options

Type of adder	Time complexity (delay)	Space complexity (size)
Ripple-carry	O(N)	O(N)
Carry-lookahead	$O(\log_2 N)$	$O(N \log_2 N)$
Carry-skip	$O(\sqrt{N})$	O(N)
Carry-select	$O(\sqrt{N})$	O(N)

Reference: J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, San Francisco, California, 1990.

Carry-lookahead adder

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

$$c_1 = g_0 + p_0 c_0$$

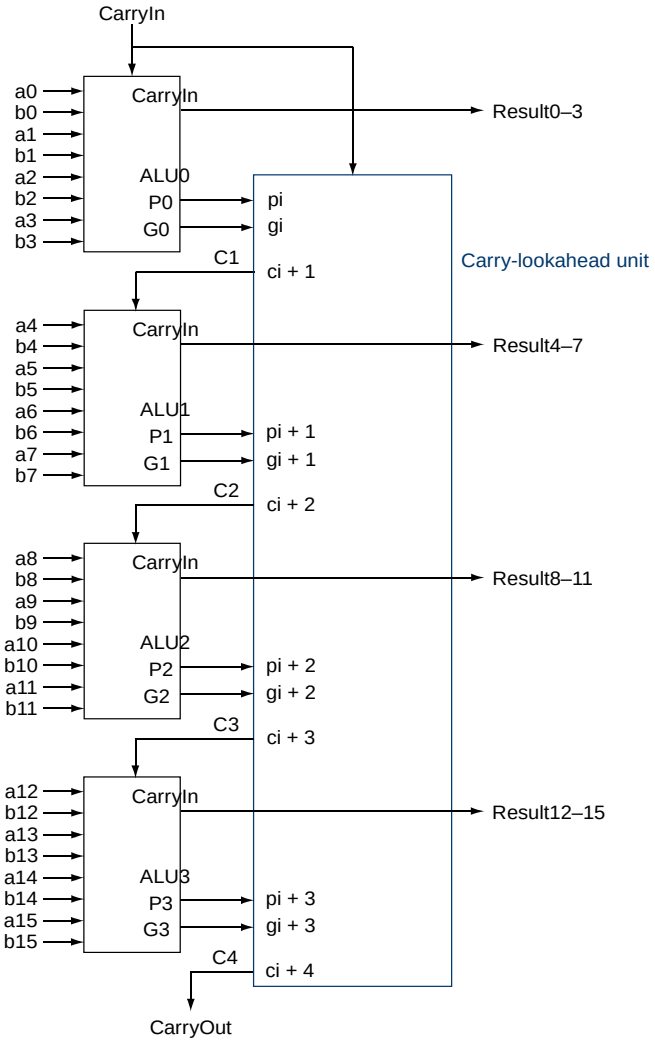
$$c_2 = g_1 + p_1 c_1 \quad c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 c_2 \quad c_3 = \ldots$$

$$c_4 = g_3 + p_3 c_3 \quad c_4 = \ldots$$

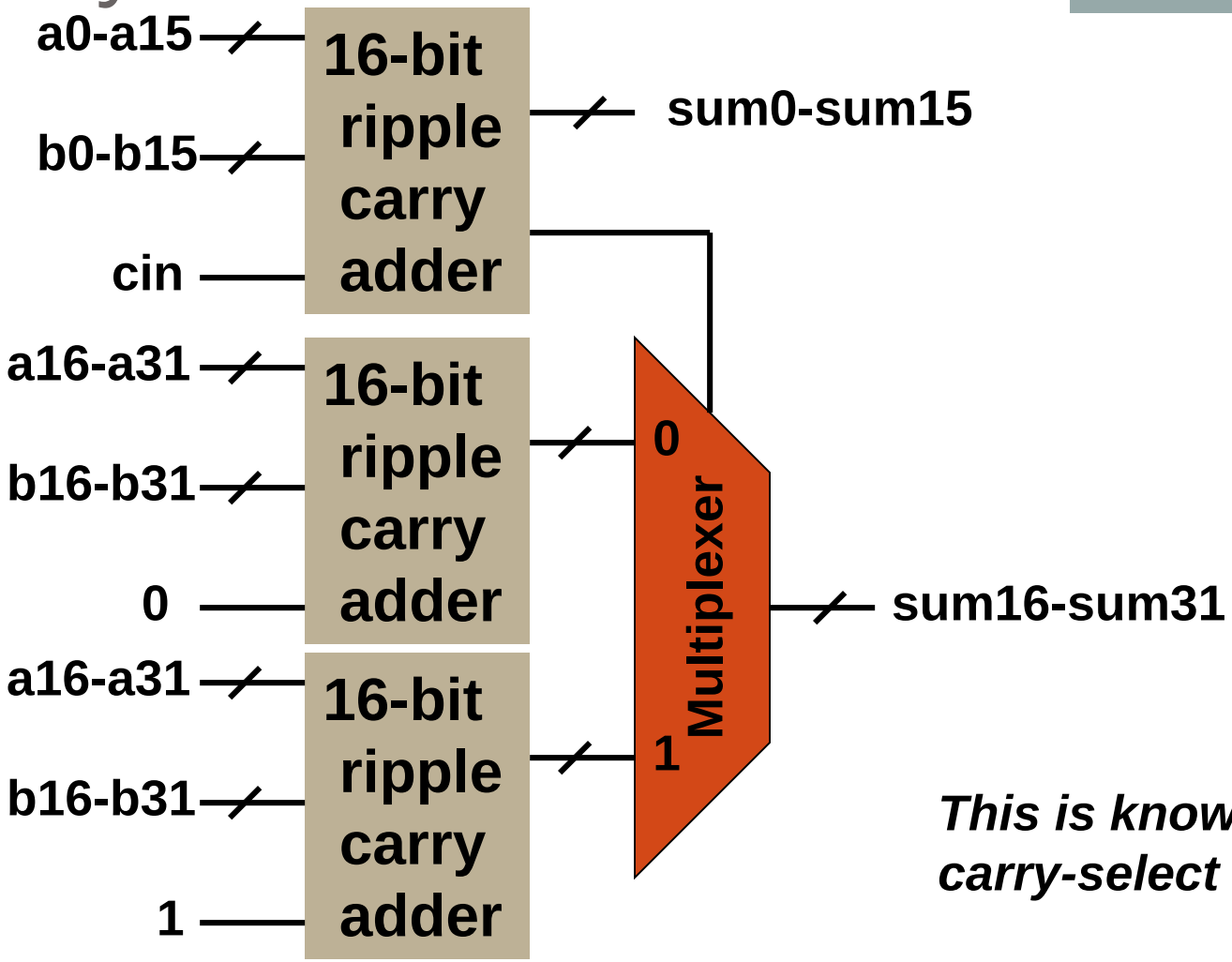
Feasible! Why?

Use principle to build bigger adders



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

Carry-Select Adder



This is known as carry-select adder

ALU Summary

- We can build an ALU to support MIPS addition
- Our focus is on comprehension, not performance
- Real processors use more sophisticated techniques for arithmetic
- Where performance is not critical, hardware description languages allow designers to completely automate the creation of hardware!

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

Multiplication

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

```
    0010   (multiplicand)
  x 1011   (multiplier)
```

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

Multiplication

- Start with long-multiplication approach

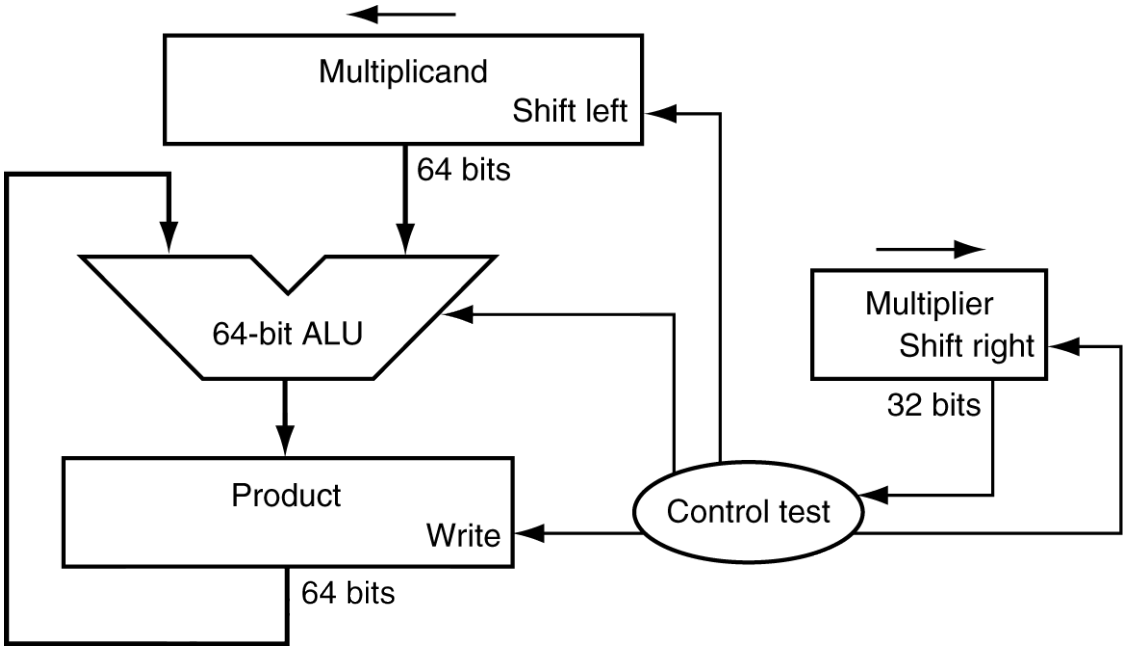
multiplicand

multiplier

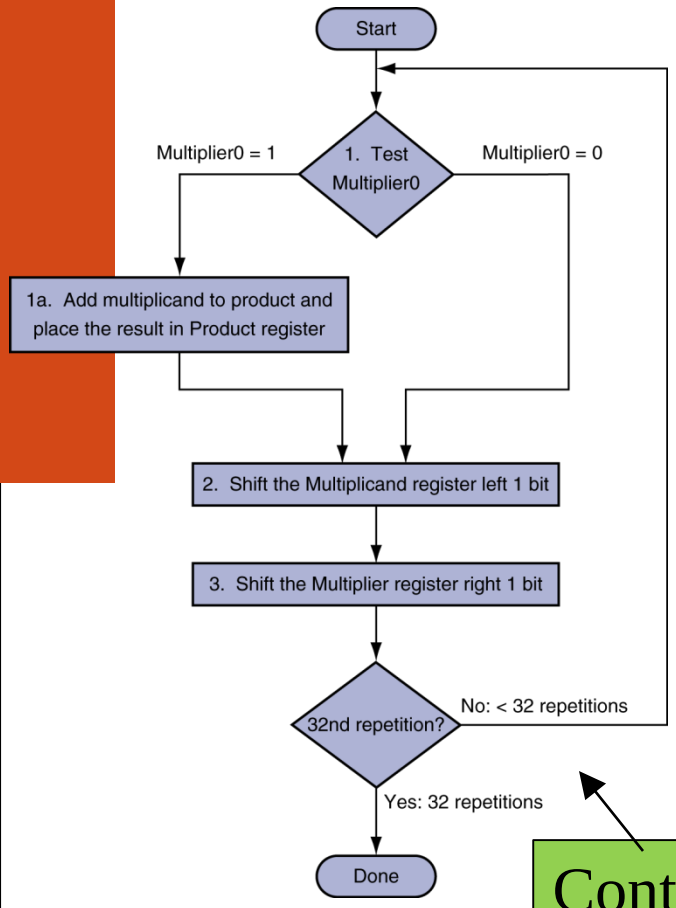
product

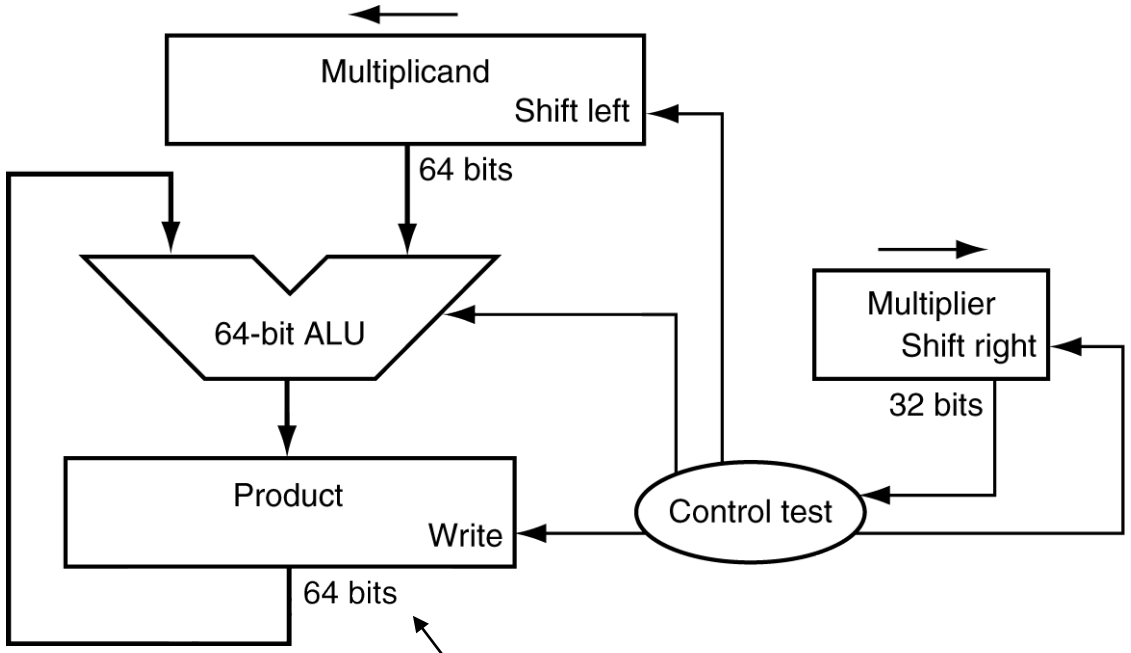
```
    1000
×   1001
    1000
   0000
  0000
 1000
1001000
```

Length of product is the sum of operand lengths



Multiplication Hardware





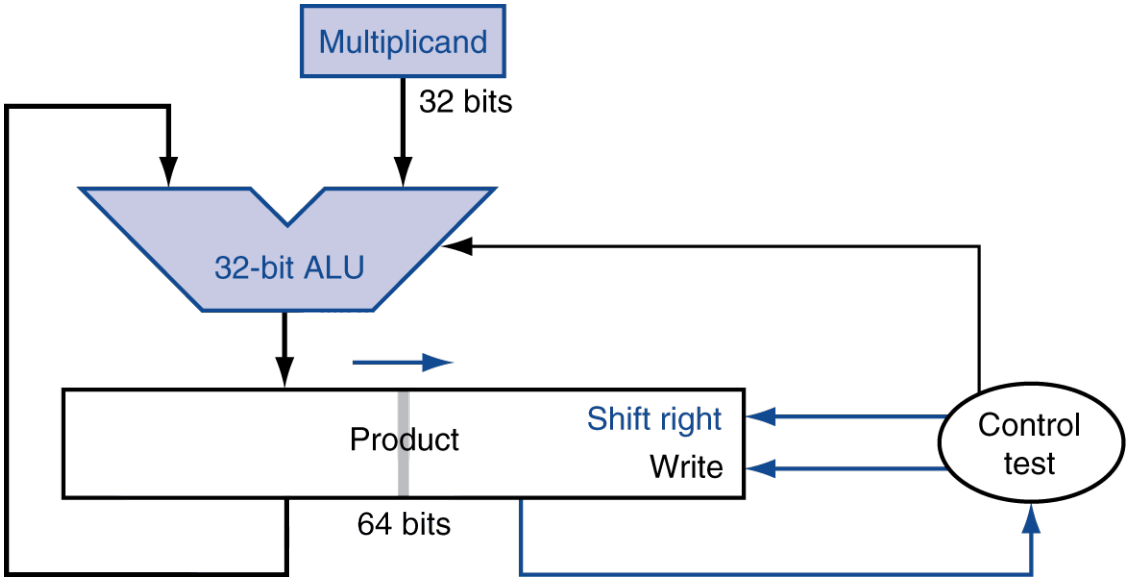
Control

Datapath

Initially 0

Optimized Multiplier

- Perform steps in parallel: add/shift



- One cycle per partial-product addition
- That's ok, if frequency of multiplications is low

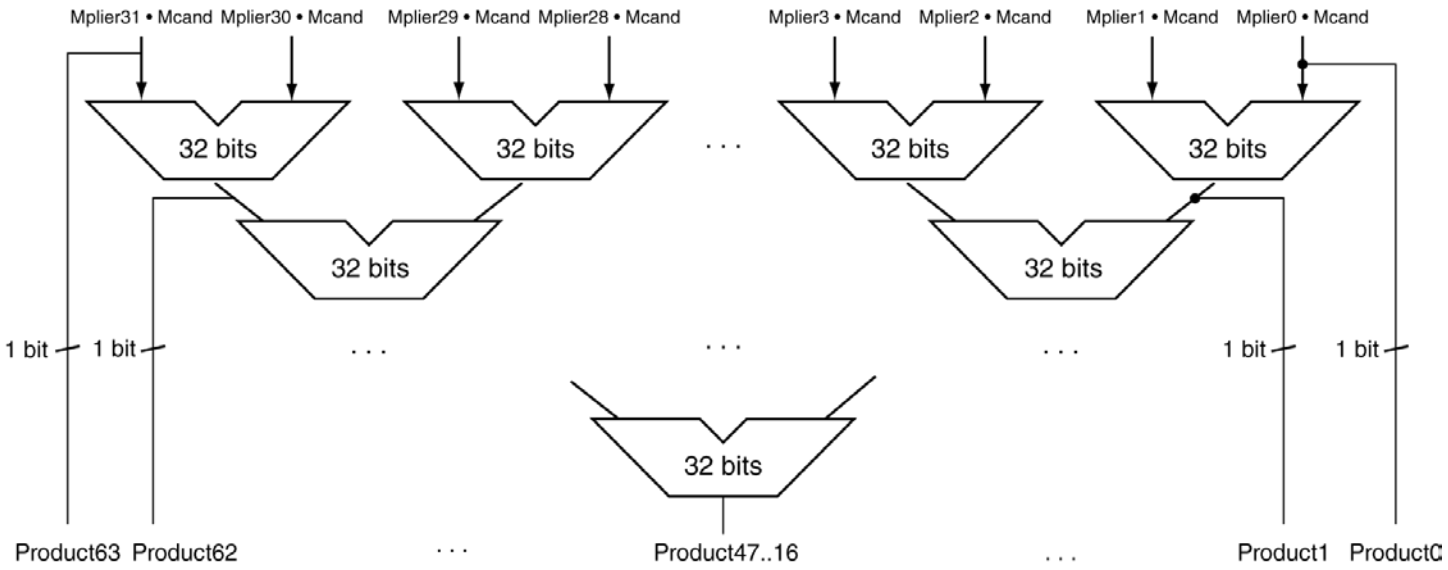
Example: $0010_{two} \times 0011_{two}$

$0010_{two} \times 0011_{two} = 0110_{two}$, i.e., $2_{ten} \times 3_{ten} = 6_{ten}$

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	LSB=1 => Prod=Prod+Mcand	0010	0010 0011
	Right shift product	0010	0001 0001
2	LSB=1 => Prod=Prod+Mcand	0010	0011 0001
	Right shift product	0010	0001 1000
3	LSB=0 => no operation	0010	0001 1000
	Right shift product	0010	0000 1100
4	LSB=0 => no operation	0010	0000 1100
	Right shift product	0010	0000 0110

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

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

Multiplying Signed Numbers with Boothe's Algorithm

- Consider A x B where A and B are signed integers (2's complemented format)
- Decompose B into the sum B1 + B2 + … + Bn

$$A \times B = A \times (B1 + B2 + \ldots + Bn)$$
$$= (A \times B1) + (A \times B2) + \ldots (A \times Bn)$$

- Let each Bi be a single string of 1's embedded in 0's:

…0011…1100…

- Example:

$$\begin{aligned} 0110010011100 = \quad & 0110000000000 \\ + \quad & 0000010000000 \\ + \quad & 0000000011100 \end{aligned}$$

Boothe's Algorithm

- Scanning from right to left, bit number u is the first 1 bit of the string and bit v is the first 0 left of the string:

$$\begin{array}{rl} & \overset{v}{0}\ \ldots\ 0\ \overset{u}{1}\ \ldots\ 1\ 0\ \ldots\ 0 \\ \text{Bi} = & \\ = & 0\ \ldots\ 0\ 1\ \ldots\ 1\ 1\ \ldots\ 1 \qquad (2^v - 1) \\ & -\ 0\ \ldots\ 0\ 0\ \ldots\ 0\ 1\ \ldots\ 1 \qquad (2^u - 1) \end{array}$$

$$= (2^v - 1) - (2^u - 1)$$

$$= 2^v - 2^u$$

Boothe's Algorithm

- Decomposing B:

$$A \times B = A \times (B1 + B2 + \dots)$$
$$= A \times [(2^{v1} - 2^{u1}) + (2^{v2} - 2^{u2}) + \dots]$$
$$= (A \times 2^{v1}) - (A \times 2^{u1}) + (A \times 2^{v2}) - (A \times 2^{u2}) \dots$$

- A x B can be computed by adding and subtracted shifted values of A:
  - Scan bits right to left, shifting A once per bit
  - When the bit string changes from 0 to 1, subtract shifted A from the current product $P - (A \times 2^{u})$
  - When the bit string changes from 1 to 0, add shifted A to the current product $P + (A \times 2^{v})$

Booth Algorithm: Example 1

- $7 \times 3 = 21$

```
      0111              multiplicand    = 7
    ×0011(0)            multiplier      = 3
  ---------
  11111001    bit-pair 10, add -7 in two's com.
              bit-pair 11, do nothing
  000111      bit-pair 01, add 7
  ---------   bit-pair 00, do nothing
  00010101    = 21
```

Booth Algorithm: Example 2

- $7 \times (-3) = -21$

0111	multiplicand	= 7
×1101(0)	multiplier	= -3
11111001	bit-pair 10, add -7 in two's com.	
0000111	bit-pair 01, add 7	
111001	bit-pair 10, add -7 in two's com.	
	bit-pair 11, do nothing	
11101011	- 21	

Booth Algorithm: Example 3

- -7 × 3 = -21

1001 multiplicand = -7 in two's com.

×0011(0) multiplier = 3

00000111 bit-pair 10, add 7

bit-pair 11, do nothing

111001 bit-pair 01, add -7

bit-pair 00, do nothing

11101011 - 21

Booth Algorithm: Example 4

- $-7 \times (-3) = 21$

1001	multiplicand	= -7 in two's com.
×1101(0)	multiplier	= -3 in two's com.

00000111	bit-pair 10, add 7
1111001	bit-pair 01, add -7 in two's com.
000111	bit-pair 10, add 7
	bit-pair 11, do nothing
00010101	21

Booth Advantage

Serial multiplication

```
  00010100   20
×00011110   30
  00000000
  00010100
  00010100
  00010100
  00010100
  00000000
  00000000
  00000000
000001001011000   600
```

Four partial product additions

Booth algorithm

```
  00010100   20
×00011110   30
  00000000
  00010100
  00010100
  00010100
  00010100
  00000000
  00000000
  00000000
000001001011000   600
```

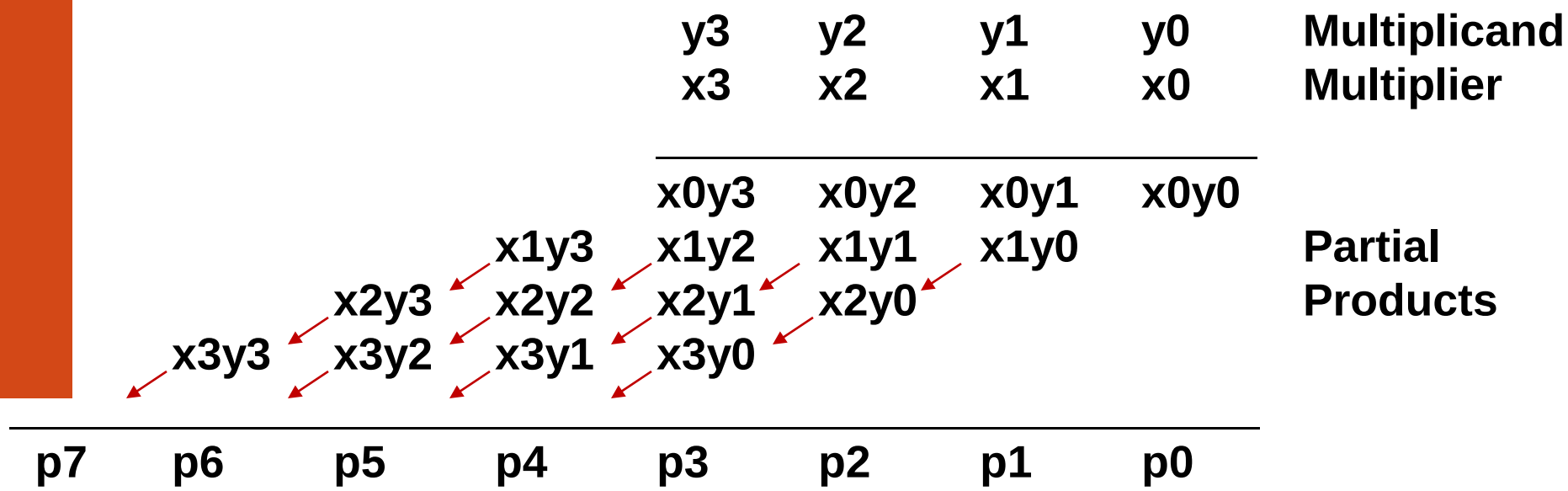
Two partial product additions

Adding Partial Products

				y3	y2	y1	y0	Multiplicand
				x3	x2	x1	x0	Multiplier
				x0y3	x0y2	x0y1	x0y0	
		carry←	x1y3	x1y2	x1y1	x1y0		Partial
	carry←	x2y3	x2y2	x2y1	x2y0			Products
carry←	x3y3	x3y2	x3y1	x3y0				
p7	p6	p5	p4	p3	p2	p1	p0	

Requires three 4-bit adders. Slow.

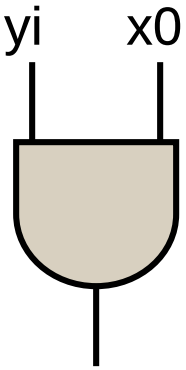
Array Multiplier: Carry Forward



Note: Carry is added to the next partial product. Adding the carry from the final stage needs an extra stage. These additions are faster but we need four stages.

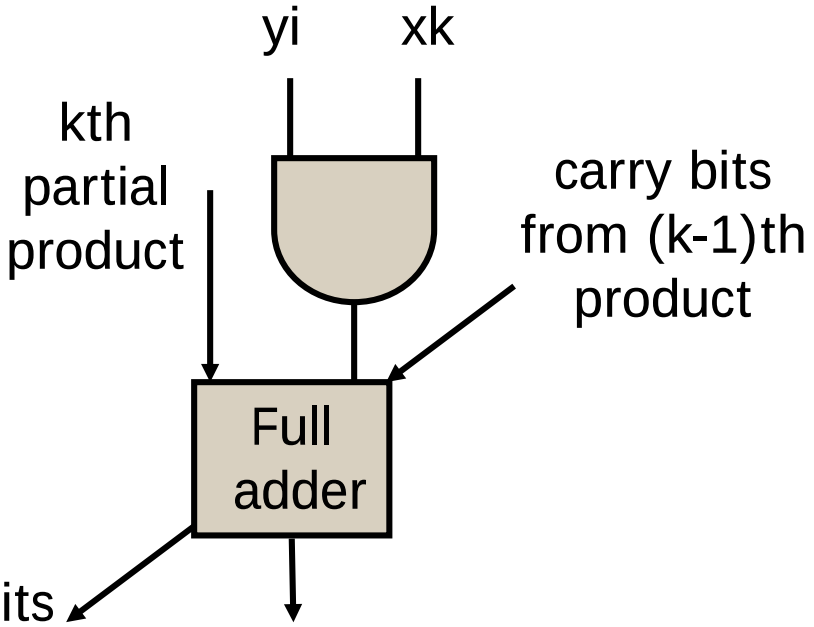
Basic Building Blocks

- Two-input AND
- Full-adder

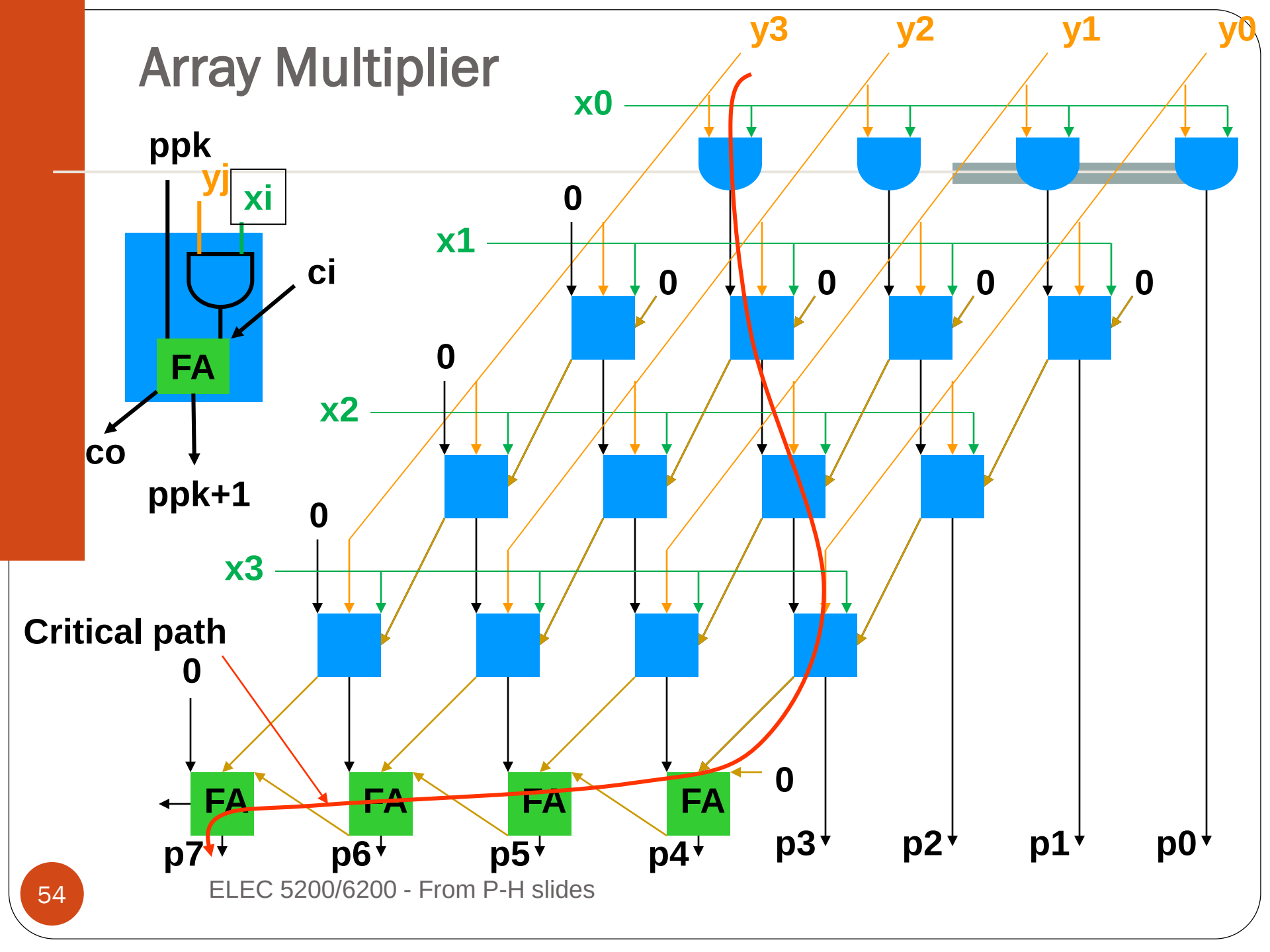


p0i = x0yi

0th partial product



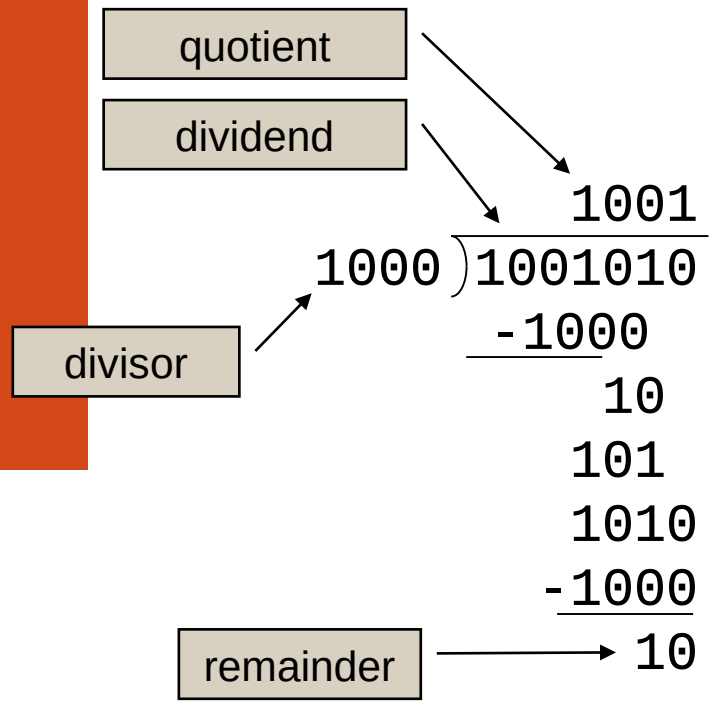
Array Multiplier



Types of Array Multipliers

- Baugh-Wooley Algorithm: Signed product by two's complement addition or subtraction according to the MSB's.
- Booth multiplier algorithm
- Tree multipliers
- Reference: N. H. E. Weste and D. Harris, *CMOS VLSI Design, A Circuits and Systems Perspective, Third Edition*, Boston: Addison-Wesley, 2005.

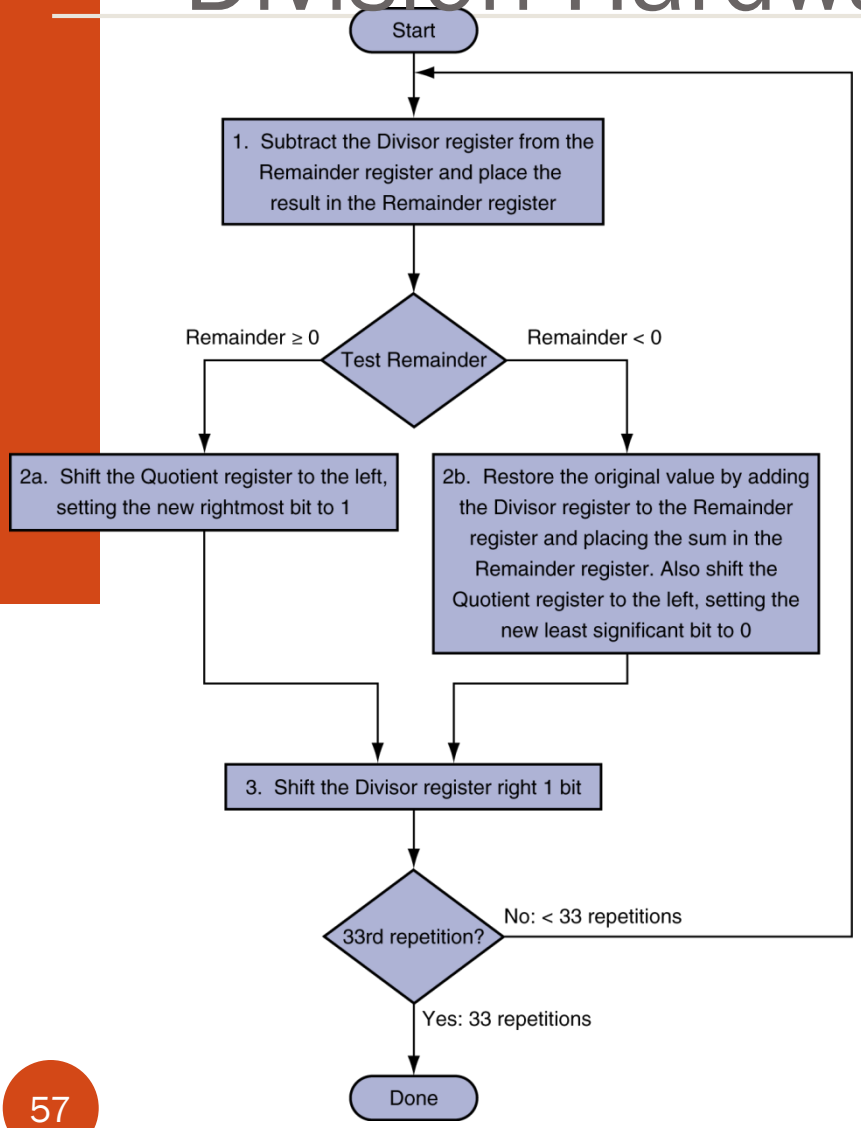
Division

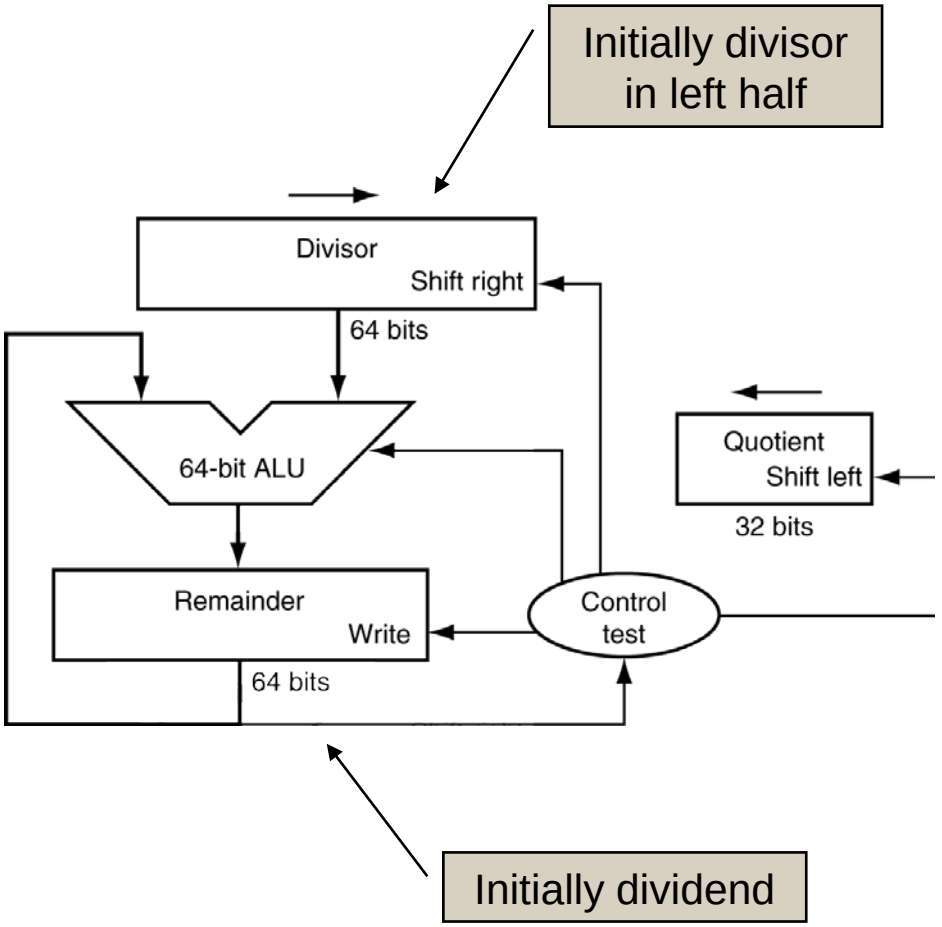


n-bit operands yield *n*-bit quotient and remainder

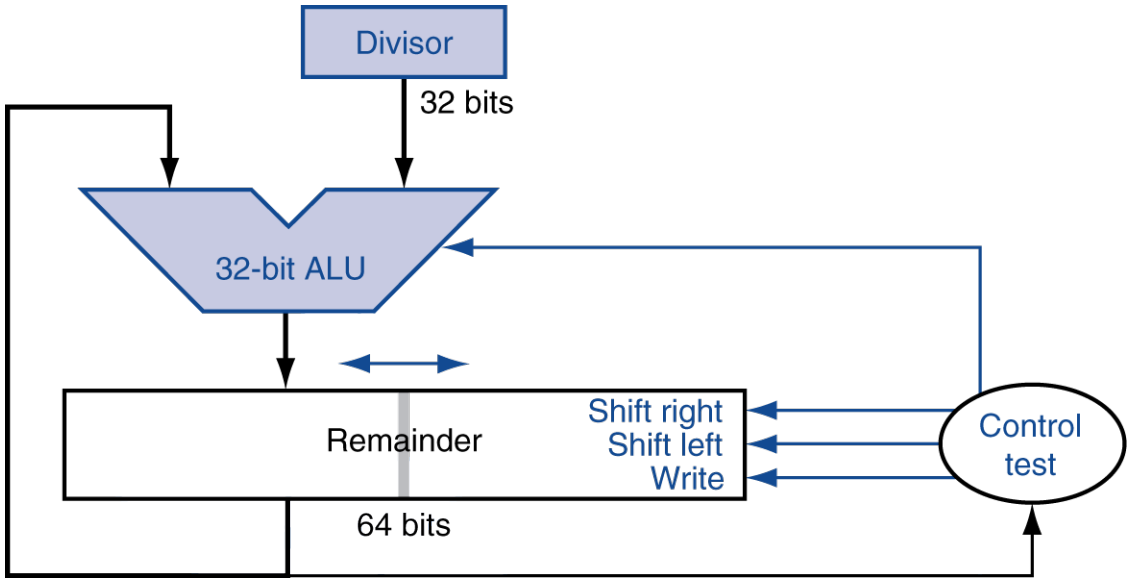
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware



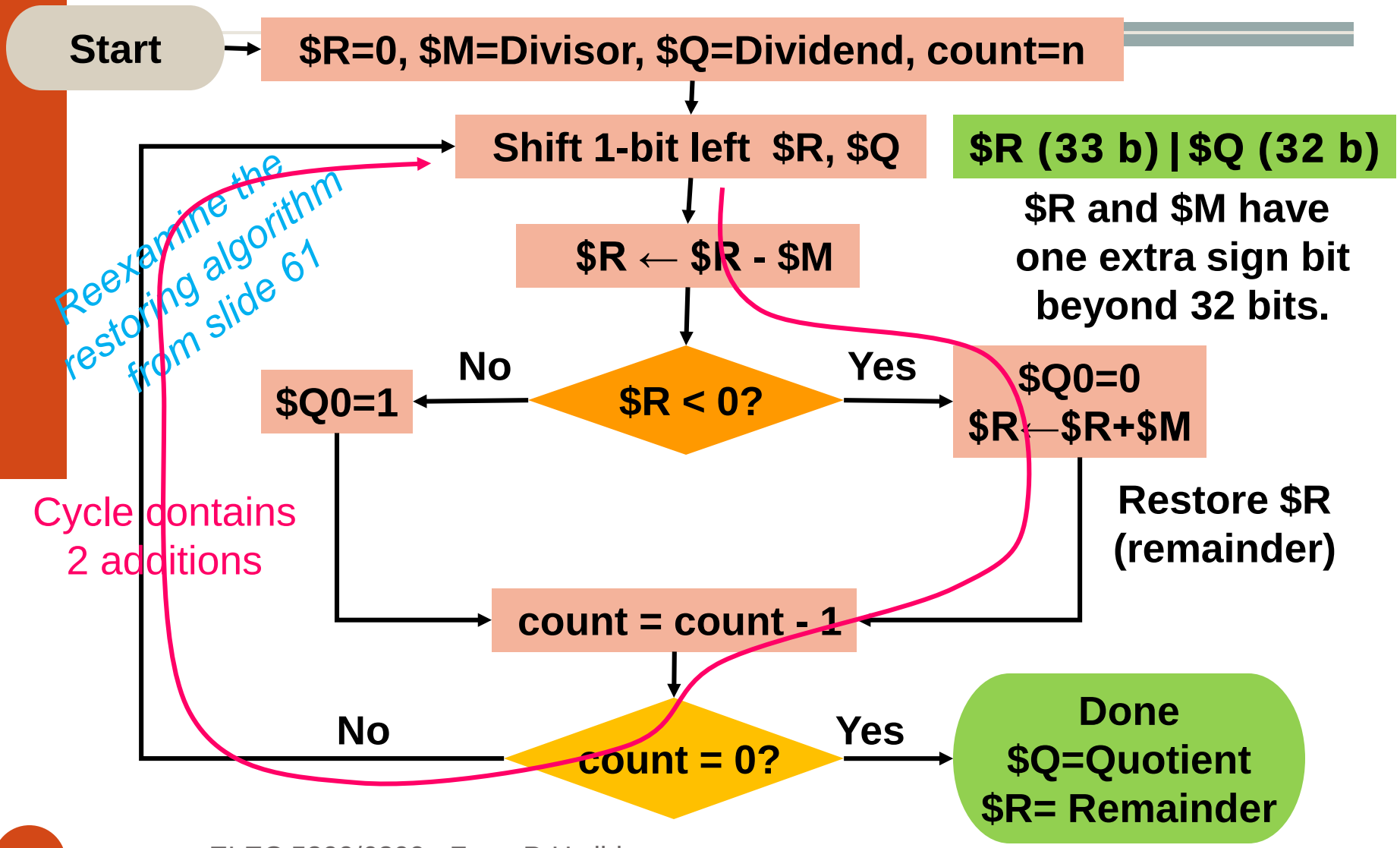


Optimized Divider

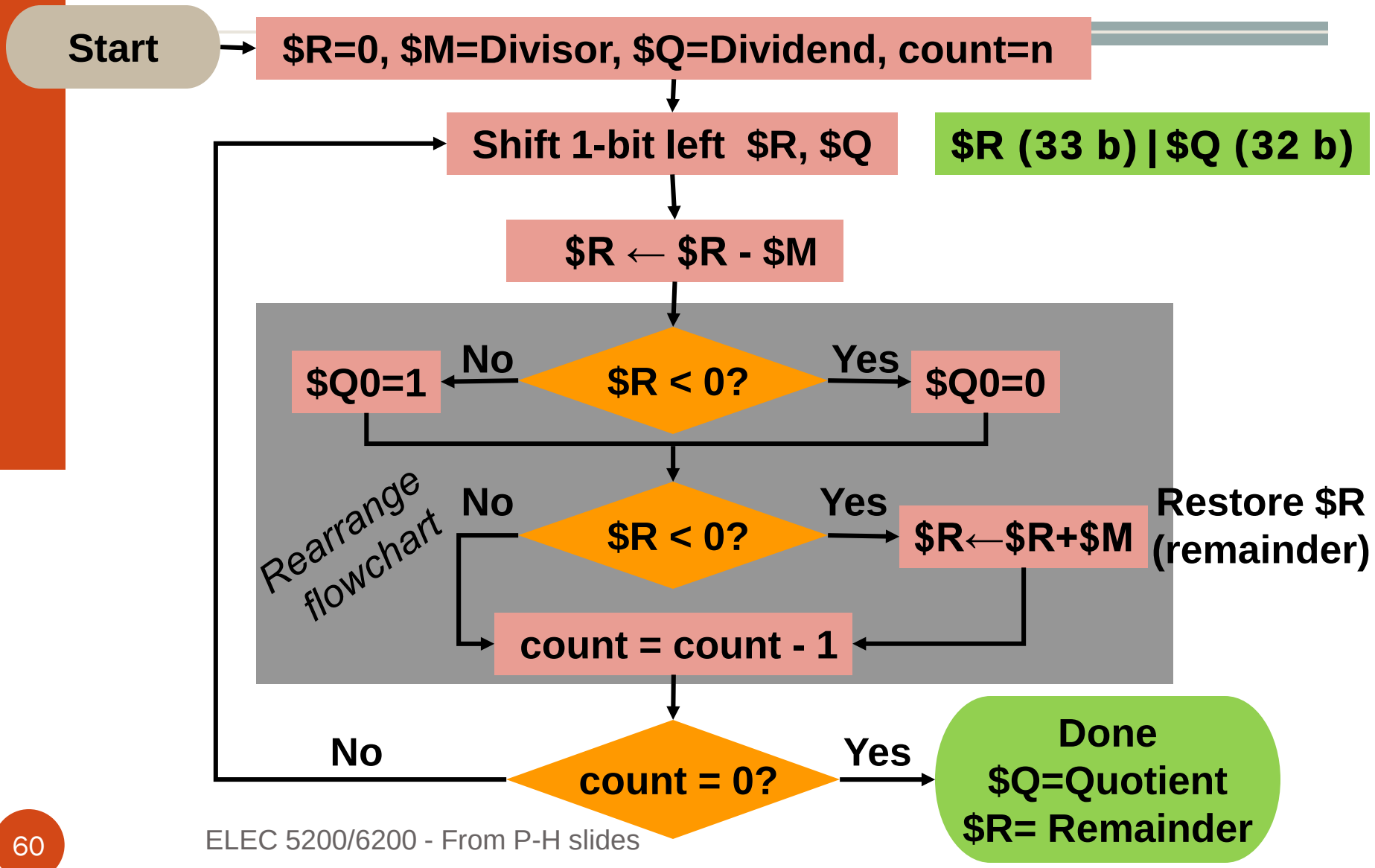


- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

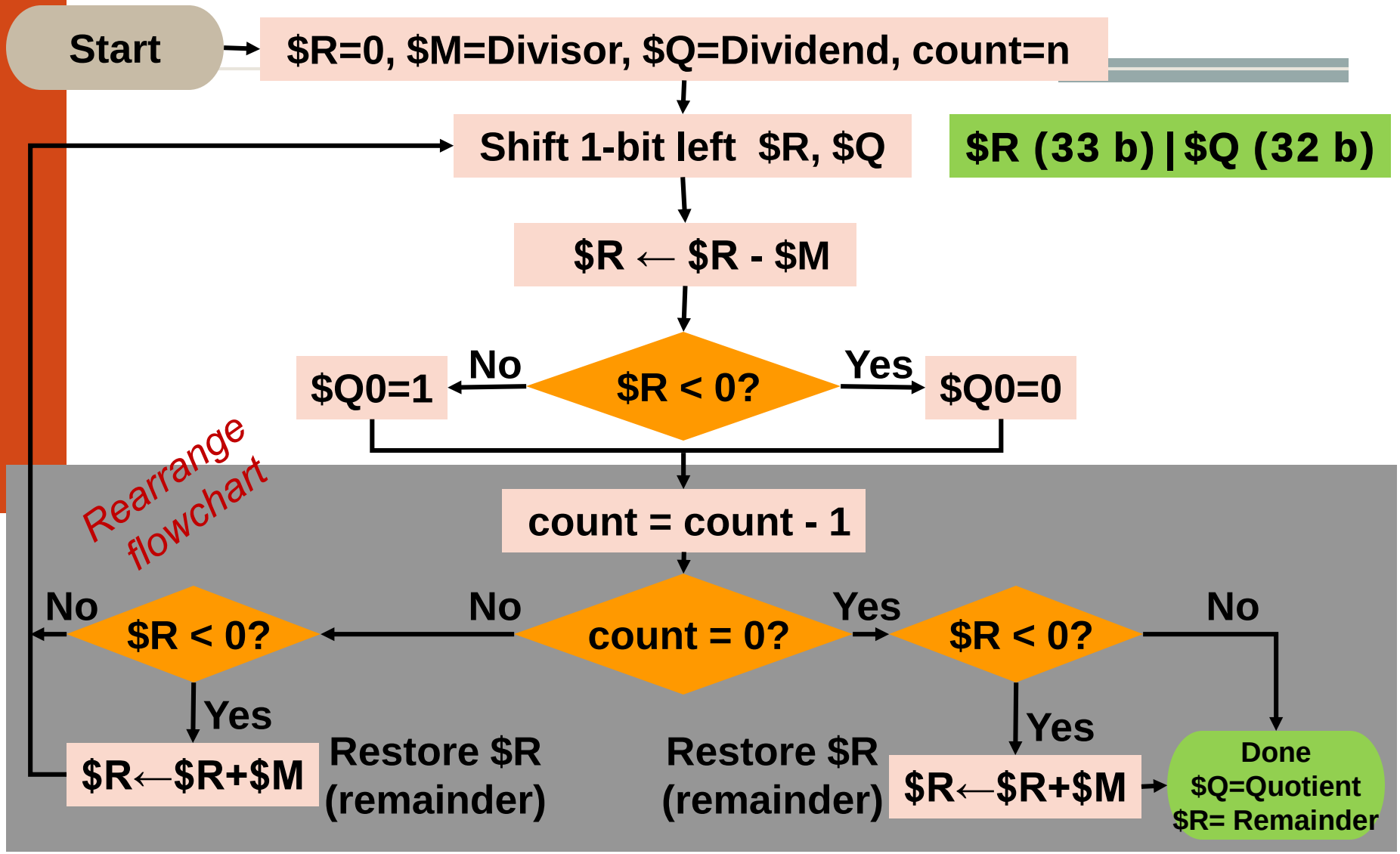
Deriving a Better Algorithm (1)



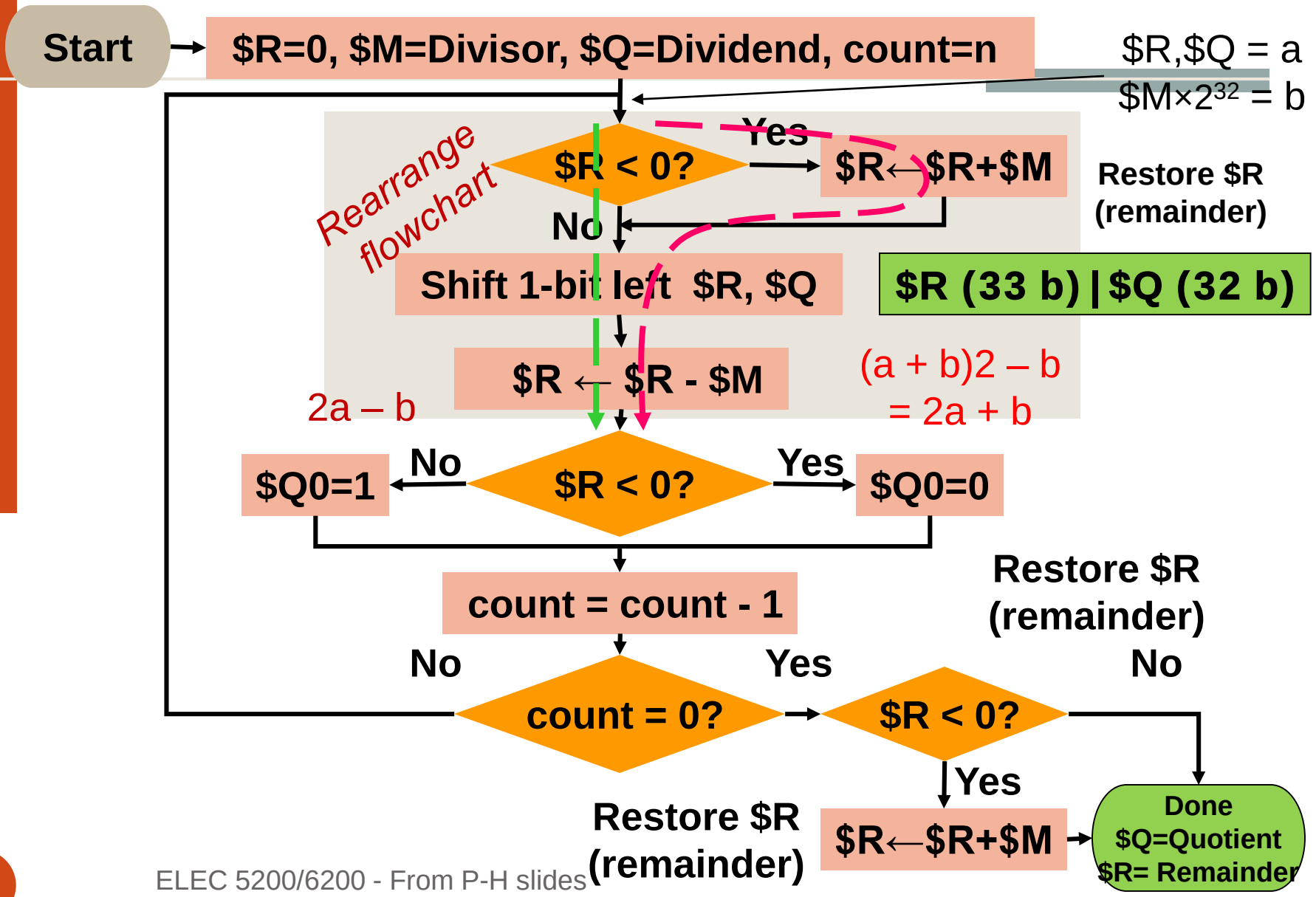
Deriving a Better Algorithm (2)



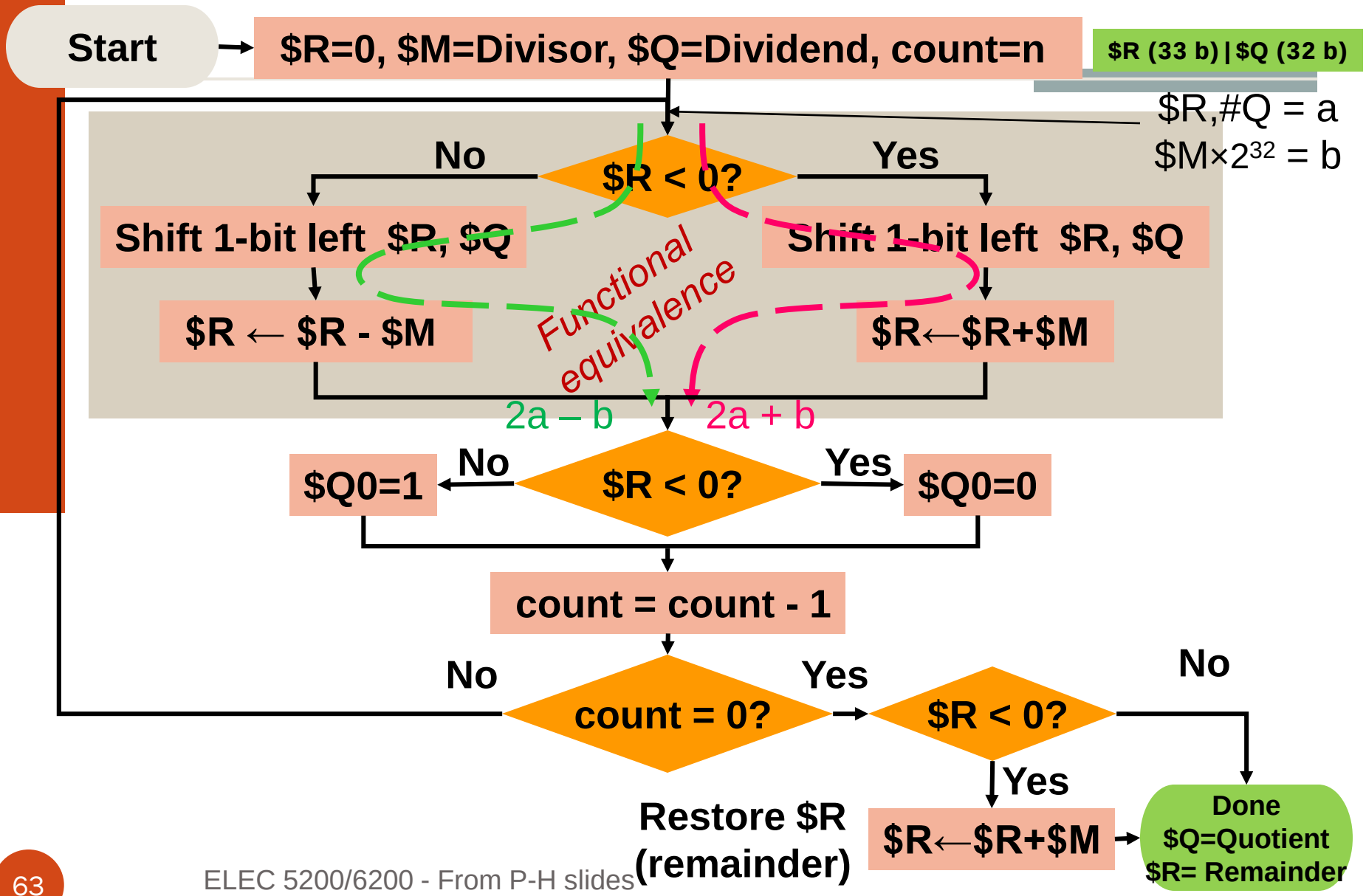
Deriving a Better Algorithm (3)



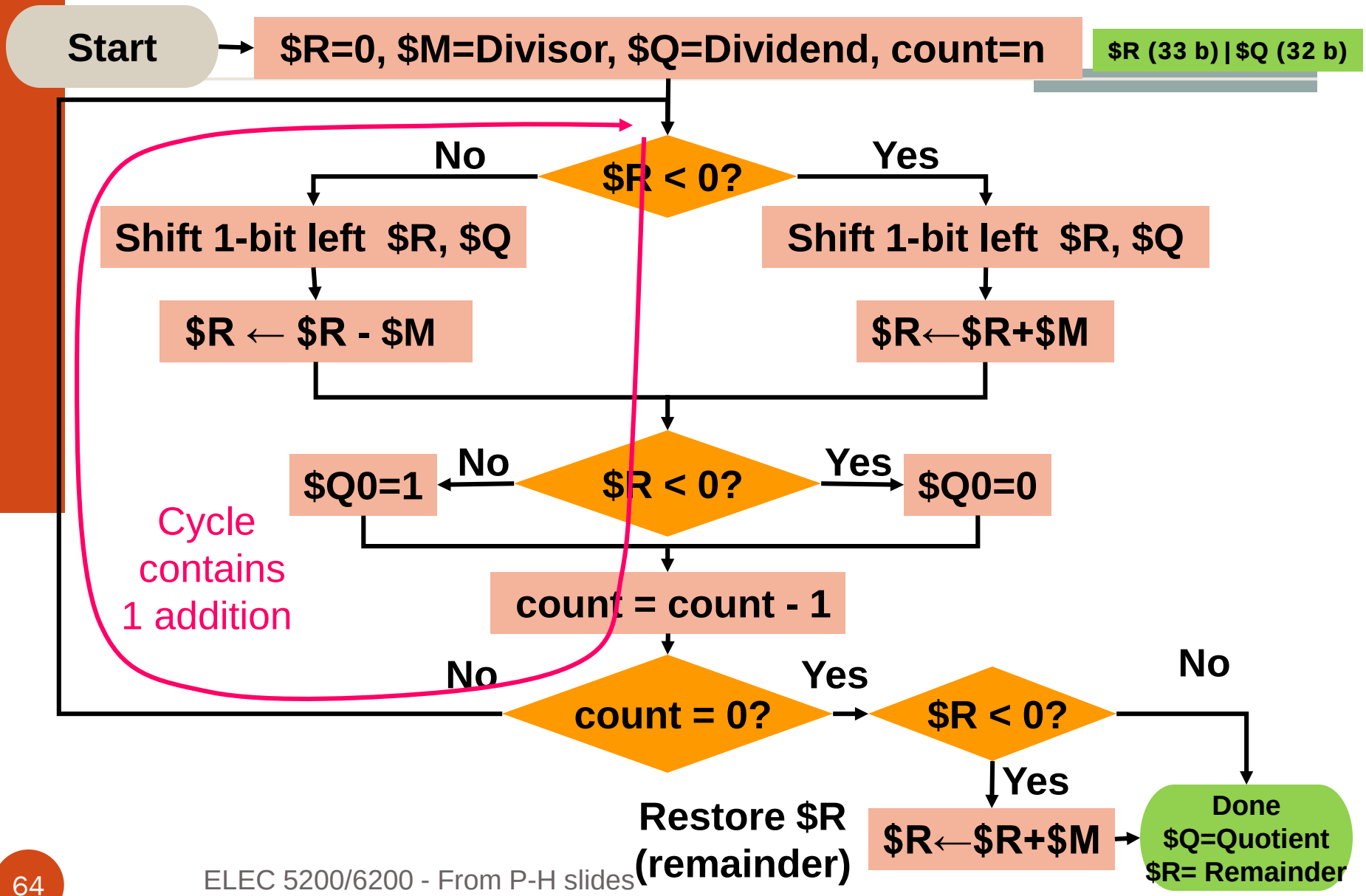
Deriving a Better Algorithm (4)



Deriving a Better Algorithm (4)



Non-Restoring Division Algorithm



Non-Restoring Division

- Avoids the addition in the restore operation – does exactly one add or subtract per cycle.
- Non-restoring division algorithm:
  - **Step 1: Repeat 32 times**
    - **if sign bit of $R is 0**
      Left shift $R,Q one-bit and subtract, $R ← $R - $M
      else (sign bit of $R is 1)
      Left shift $R,Q one-bit and add, $R ← $R + $M
    - **if sign bit of resulting $R is 0**
      Set Q0 = 1
      else (sign bit of resulting $R is 1)
      Set Q0 = 0
  - **Step 2: (after 32 Step 1 iterations) if sign bit of $R is 1, add $R ← $R + $M**

Non-Restoring Division: 8/3 = 2 (Rem=2)

Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4

Initialize	**$R**	**= 0 0 0 0 0**	**$Q =**	**1 0 0 0**	**$M =**	**0 0 0 1 1**
Step 1, L-shift	$R,Q	= 0 0 0 0 1	$Q =	0 0 0 ?	$M =	0 0 0 1 1
Subtract	- $M	= 1 1 1 0 1				
	$R	= 1 1 1 1 0	$Q =	0 0 0 0		
Step 1, L-shift	$R,Q	= 1 1 1 0 0	$Q =	0 0 0 ?	$M =	0 0 0 1 1
Add	+$M	= 0 0 0 1 1				
	$R	= 1 1 1 1 1	$Q =	0 0 0 0		
Step 1, L-shift	$R,Q	= 1 1 1 1 0	$Q =	0 0 0 ?	$M =	0 0 0 1 1
Add	+$M	= 0 0 0 1 1				
	$R	= 0 0 0 0 1	$Q =	0 0 0 1		
Step 1, L-shift	$R,Q	= 0 0 0 1 0	$Q =	0 0 1 ?	$M =	0 0 0 1 1
Subtract	-$M	= 1 1 1 0 1				
	$R	= 1 1 1 1 1	$Q =	0 0 1 0 *Final quotient*		

Step 2, Add $R ← $R + $M = 11111+00011 = 00010 (Final remainder)

Faster Division

- Can't use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT devision) generate multiple quotient bits per step
  - Still require multiple steps

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt  /  divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result