



Πανεπιστήμιο Ιωαννίνων

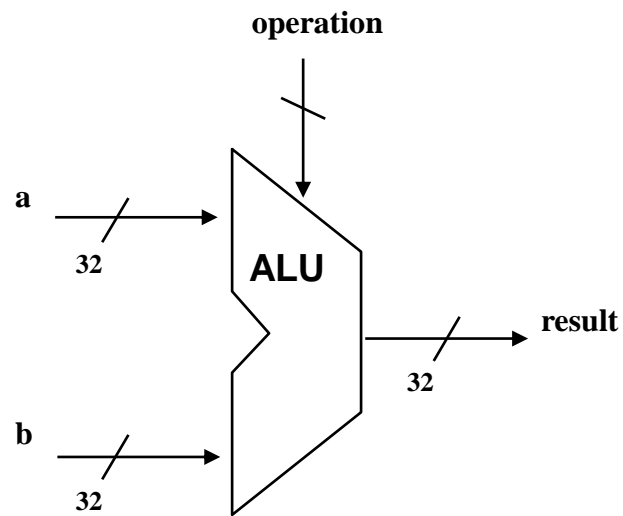
Ειδικά Θέματα Αρχιτεκτονικής και Προγραμματισμού Μικροεπεξεργαστών

Ενότητα 3: Arithmetic for Computers


Διδάσκων: Βαρτζιώτης Φώτιος
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Arithmetic

- ▶ Where we've been:
 - ▶ performance
 - ▶ 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 integers (base 2)
 - ▶ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
 - 
 - ▶ decimal: 0, ..., $2^n - 1$ n bits
- ▶ Of course it gets more complicated:
 - ▶ bit strings are *finite*, but
 - ▶ for some *fractions* and *real* numbers, finitely many bits is not enough, so
 - ▶ *overflow* & *approximation* errors: e.g., represent 1/3 as binary!
 - ▶ *negative* integers
- ▶ How do we represent negative integers?
 - ▶ which bit patterns will represent which integers?

Possible Representations

Sign Magnitude:

000 = 0
001 = +1
010 = +2
011 = +3
100 = 0
101 = -1
110 = -2
111 = -3

ambiguous zero

One's Complement

000 = 0
001 = +1
010 = +2
011 = +3
100 = -3
101 = -2
110 = -1
111 = 0

ambiguous zero

Two's Complement

000 = 0
001 = +1
010 = +2
011 = +3
100 = -4
101 = -3
110 = -2
111 = -1

unequal no. of negatives and positives; unique zero

Issues:

- ▶ *balance* - equal number of negatives and positives
- ▶ *ambiguous zero* - whether more than one zero representation
- ▶ ease of arithmetic operations

▶ Which representation is best? Can we get both balance and non-ambiguous zero?

Representation Formulae

► Two's complement:

$$x_n x_{n-1} \dots x_0 = x_n * -2^n + x_{n-1} * 2^{n-1} + \dots + x_0 * 2^0$$

or

$$x_n X' = x_n * -2^n + X' \text{ (writing rightmost } n \text{ bits } x_{n-1} \dots x_0 \text{ as } X')$$

$$= \begin{cases} X', & \text{if } x_n = 0 \\ -2^n + X', & \text{if } x_n = 1 \end{cases}$$

► One's complement:

$$x_n X' = \begin{cases} X', & \text{if } x_n = 0 \\ -2^n + 1 + X', & \text{if } x_n = 1 \end{cases}$$

MIPS - 2's complement

► 32 bit signed numbers:

0000	0000	0000	0000	0000	0000	0000	0000	$_{\text{two}}$	=	0_{ten}	
0000	0000	0000	0000	0000	0000	0000	0001	$_{\text{two}}$	=	$+ 1_{\text{ten}}$	
0000	0000	0000	0000	0000	0000	0000	0010	$_{\text{two}}$	=	$+ 2_{\text{ten}}$	
...											
0111	1111	1111	1111	1111	1111	1111	1110	$_{\text{two}}$	=	$+ 2,147,483,646_{\text{ten}}$	maxint
0111	1111	1111	1111	1111	1111	1111	1111	$_{\text{two}}$	=	$+ 2,147,483,647_{\text{ten}}$	
1000	0000	0000	0000	0000	0000	0000	0000	$_{\text{two}}$	=	$- 2,147,483,648_{\text{ten}}$	
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	$_{\text{two}}$	=	$- 1_{\text{ten}}$	minint

Negative integers are exactly those that have leftmost bit 1

Two's Complement Operations

- ▶ Negation Shortcut: To *negate* any two's complement integer (except for minint) *invert* all bits and *add 1*

- ▶ note that *negate* and *invert* are different operations!

- ▶ Sign Extension Shortcut: To convert an n-bit integer into an integer with more than n bits - i.e., to make a narrow integer fill a wider word - *replicate the most significant bit (msb)* of the original number to fill the new bits to its left

- ▶ *Example:* 4-bit 8-bit

0010 = 0000 0010

1010 = 1111 1010

- ▶ *why is this correct? Prove!*

MIPS Notes

- ▶ `lb` vs. `lbu`
 - ▶ signed load sign extends to fill 24 left bits
 - ▶ unsigned load fills left bits with 0's
- ▶ `slt` & `slti`
 - ▶ compare signed numbers
- ▶ `sltu` & `sltiu`
 - ▶ compare unsigned numbers, i.e., treat both operands as non-negative

Two's Complement Addition

- ▶ Perform add just as in 1st semester (carry/borrow 1s)

- ▶ Examples (4-bits):

0101	0110	1011	1001	1111
<u>0001</u>	<u>0101</u>	<u>0111</u>	<u>1010</u>	<u>1110</u>

Do these sums **now!!** Remember all registers are 4-bit including result register!

So you have to **throw away** the carry-out from the msb!!

- ▶ Have to beware of *overflow* : if the *fixed* number of bits (4, 8, 16, 32, etc.) in a register *cannot represent the result* of the operation
 - ▶ *terminology alert*: overflow *does not mean* there was a carry-out from the msb that we lost (though it sounds like that!) - it means simply that the result in the fixed-sized register is incorrect
 - ▶ as can be seen from the above examples there are cases when the result is correct even after losing the carry-out from the msb

Two's Complement Addition: Verifying Carry/Borrow method

- Two $(n+1)$ -bit integers: $X = x_n X'$, $Y = y_n Y'$

Carry/borrow add $X + Y$	$0 \leq X' + Y' < 2^n$ (no CarryIn to last bit)	$2^n \leq X' + Y' < 2^{n+1} - 1$ (CarryIn to last bit)
$x_n = 0, y_n = 0$	ok	not ok (overflow!)
$x_n = 1, y_n = 0$	ok	ok
$x_n = 0, y_n = 1$	ok	ok
$x_n = 1, y_n = 1$	not ok (overflow!)	ok

- *Prove the cases above!*
- Prove if there is *one more bit* (total $n+2$ then) available for the result then there is no problem with overflow in add!

Two's Complement Operations

- ▶ *Now verify the negation shortcut!*

- ▶ consider $X + (\bar{X} + 1) = (X + \bar{X}) + 1$:

associative law - but what if there is overflow in one of the adds on either side, i.e., the result is wrong...!

- ▶ think *minint* !

- ▶ *Examples:*

- ▶ $-0101 = 1010 + 1 = 1011$

- ▶ $-1100 = 0011 + 1 = 0100$

- ▶ $-1000 \neq 0111 + 1 = 1000$

Detecting Overflow

- ▶ *No overflow* when adding a positive and a negative number
- ▶ *No overflow* when subtracting numbers with the same sign
- ▶ *Overflow occurs* when the result has “wrong” sign (*verify!*):

Operation	Operand A	Operand B	Result Indicating Overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

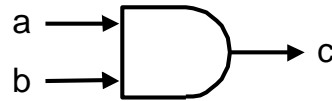
- ▶ Consider the operations $A + B$, and $A - B$
 - ▶ *can overflow occur if B is 0 ?*
 - ▶ *can overflow occur if A is 0 ?*

Effects of Overflow

- ▶ If an *exception* (interrupt) occurs
 - ▶ control jumps to predefined address for exception
 - ▶ interrupted address is saved for possible resumption
- ▶ Details based on software system/language
 - ▶ SPIM: see the EPC and Cause registers
- ▶ Don't always want to cause exception on overflow
 - ▶ `add, addi, sub` *cause exceptions* on overflow
 - ▶ `addu, addiu, subu` *do not cause exceptions* on overflow

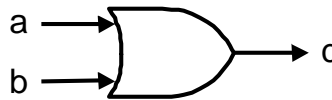
Review: Basic Hardware

1. AND gate ($c = a \cdot b$)



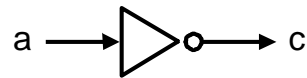
a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate ($c = a + b$)



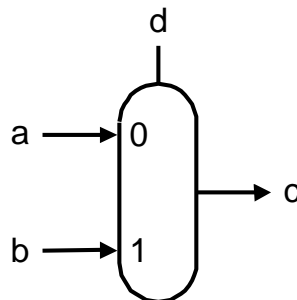
a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)



a	$c = \bar{a}$
0	1
1	0

4. Multiplexor (if $d = 0$, $c = a$; else $c = b$)



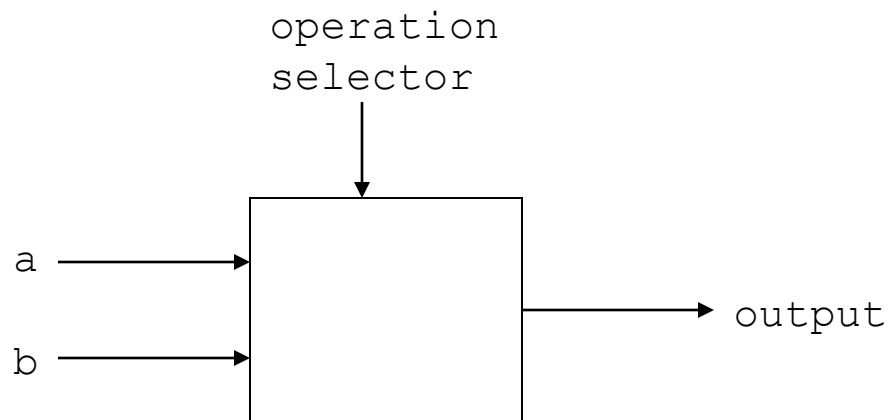
d	c
0	a
1	b

Review: Boolean Algebra & Gates

- ▶ *Problem:* Consider logic functions with three inputs: A, B, C.
 - ▶ output D is true if at least one input is true
 - ▶ output E is true if exactly two inputs are true
 - ▶ output F is true only if all three inputs are true
- ▶ *Show the truth table for these three functions*
- ▶ *Show the Boolean equations for these three functions*
- ▶ *Show an implementation consisting of inverters, AND, and OR gates.*

A Simple Multi-Function Logic Unit

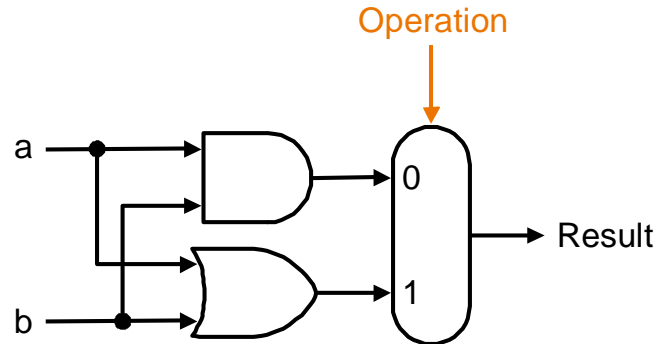
- ▶ To warm up let's build a logic unit to support the `and` and `or` instructions for MIPS (32-bit registers)
 - ▶ we'll just build a 1-bit unit and use 32 of them



- ▶ Possible implementation using a *multiplexor* :

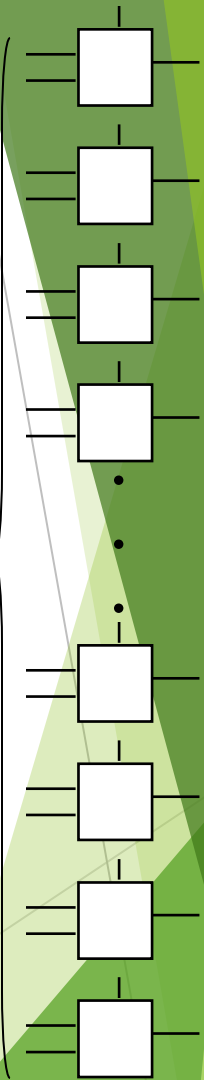
Implementation with a Multiplexor

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



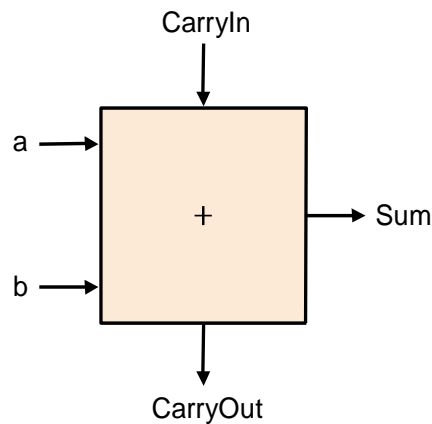
- Lets build our ALU using a MUX (multiplexor):

32 units



Implementations

- ▶ Not easy to decide the *best* way to implement something
 - ▶ do not want too many inputs to a single gate
 - ▶ do not want to have to go through too many gates (= levels)
 - ▶ for our purposes, ease of comprehension is important
- ▶ Let's look at a 1-bit ALU for addition:



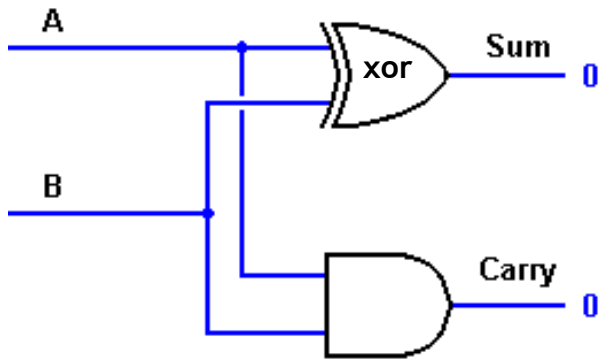
$$C_{out} = a.b + a.c_{in} + b.c_{in}$$

$$\begin{aligned} \text{sum} &= a.\overline{b}.\overline{c_{in}} + \overline{a}.b.\overline{c_{in}} + \\ &\quad \overline{a}.\overline{b}.c_{in} + a.b.c_{in} \\ &= a \oplus b \oplus c_{in} \end{aligned}$$

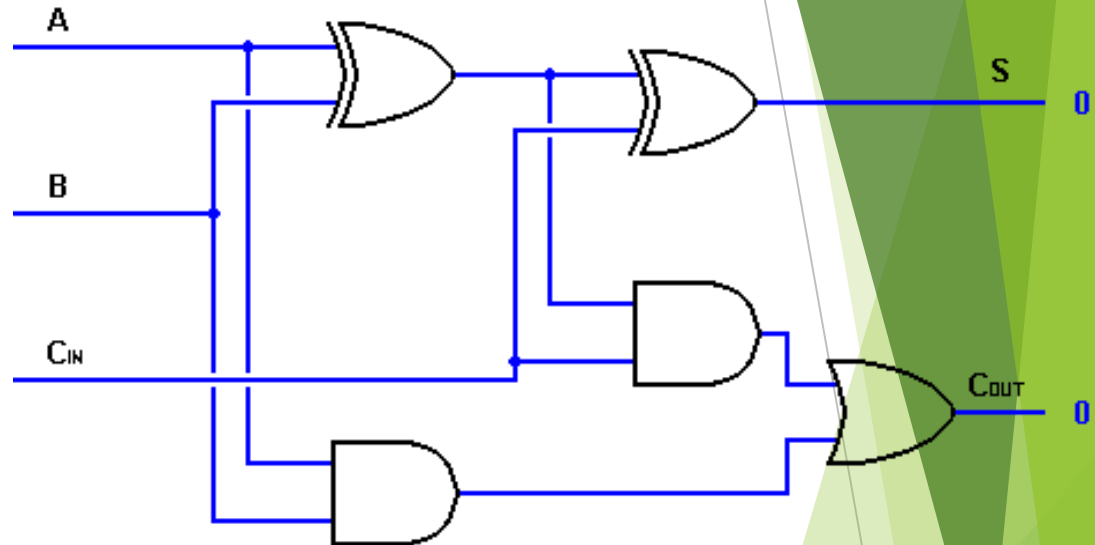
exclusive or (xor)

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

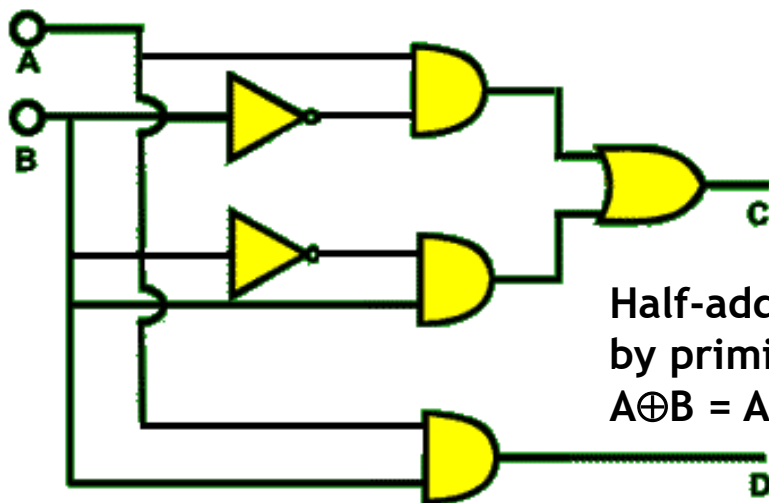
1-bit Adder Logic



Half-adder with one xor gate

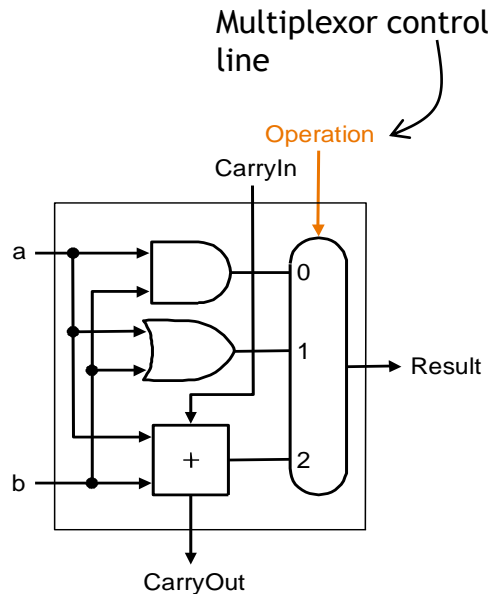


Full-adder from 2 half-adders and an or gate

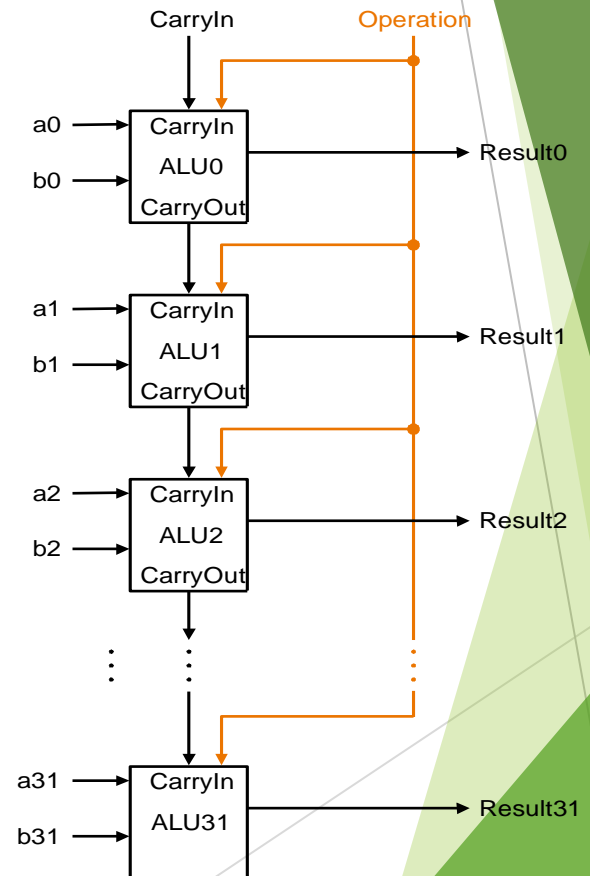


Half-adder with the xor gate replaced by primitive gates using the equation $A \oplus B = A.B + A.\overline{B}$

Building a 32-bit ALU



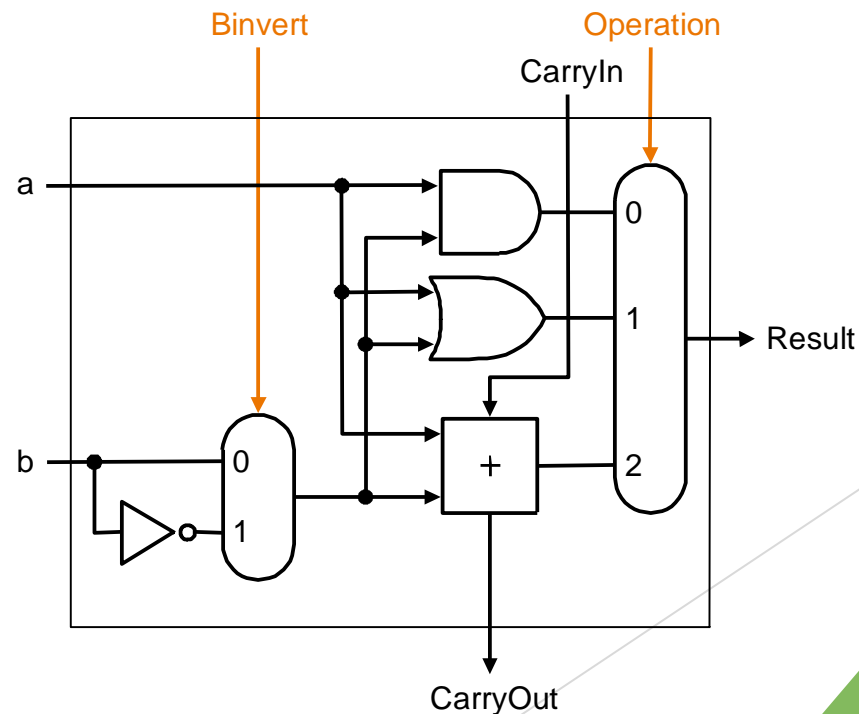
1-bit ALU for AND, OR and add



Ripple-Carry Logic for 32-bit ALU

What about Subtraction (a - b) ?

- ▶ Two's complement approach: just negate b and add.
- ▶ How do we negate?
 - ▶ recall *negation shortcut* : invert each bit of b and set CarryIn to *least significant bit (ALU0)* to 1

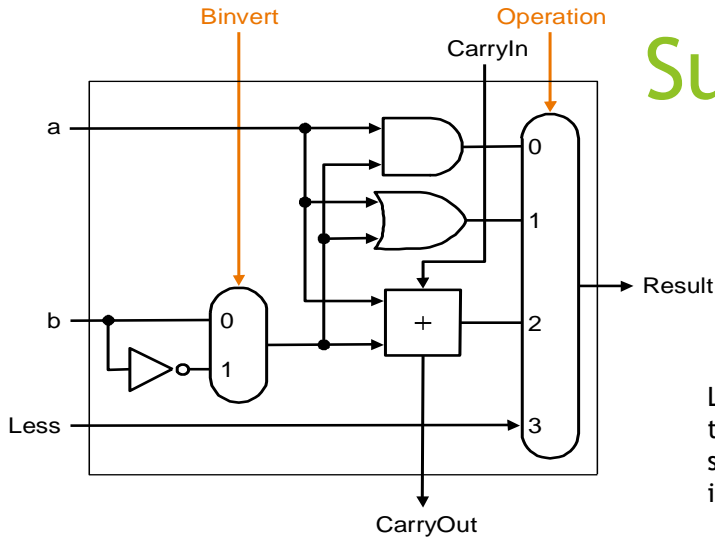


Tailoring the ALU to MIPS:

Test for Less-than and Equality

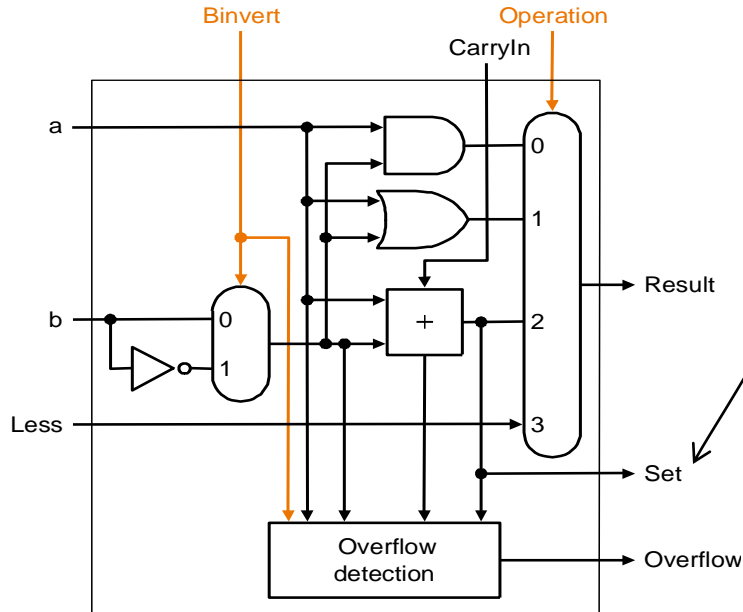
- ▶ Need to support the *set-on-less-than* instruction
 - ▶ e.g., `slt $t0, $t3, $t4`
 - ▶ remember: `slt` is an *R-type instruction* that produces 1 if $rs < rt$ and 0 otherwise
 - ▶ idea is to use subtraction: $rs < rt \Leftrightarrow rs - rt < 0$. Recall msb of negative number is 1
 - ▶ two cases after subtraction $rs - rt$:
 - ▶ if no overflow then $rs < rt \Leftrightarrow$ most significant bit of $rs - rt = 1$
 - ▶ if overflow then $rs < rt \Leftrightarrow$ most significant bit of $rs - rt = 0$
 - ▶ why?
 - ▶ e.g., $5_{\text{ten}} - 6_{\text{ten}} = 0101 - 0110 = 0101 + 1010 = 1111$ (ok!)
 $-7_{\text{ten}} - 6_{\text{ten}} = 1001 - 0110 = 1001 + 1010 = 0011$ (overflow!)
 - ▶ therefore
$$\text{set bit} = \text{msb of } rs - rt \oplus \text{overflow bit}$$
where *set bit*, which is output from ALU31, gives the result of `slt`
 - ▶ Fig. (lower) indicates set bit is the adder output - *not correct* !!
 - ▶ set bit is sent from ALU31 to ALU0 as the *Less* bit at ALU0; all other Less bits are hardwired 0; so Less is the 32-bit result of `slt`

Supporting slt

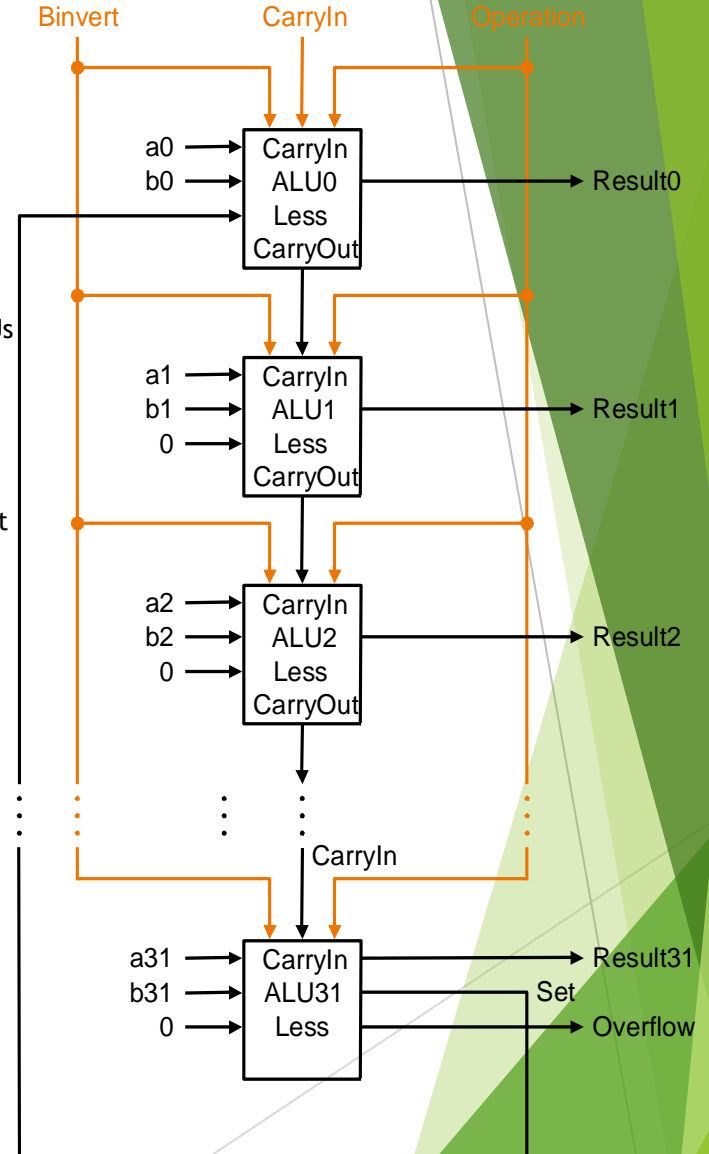


a.
1-bit ALU for the 31 least significant bits

Extra set bit, to be routed to the Less input of the least significant 1-bit ALU, is computed from the most significant Result bit and the Overflow bit (it is not the output of the adder as the figure seems to indicate)



b.
1-bit ALU for the most significant bit



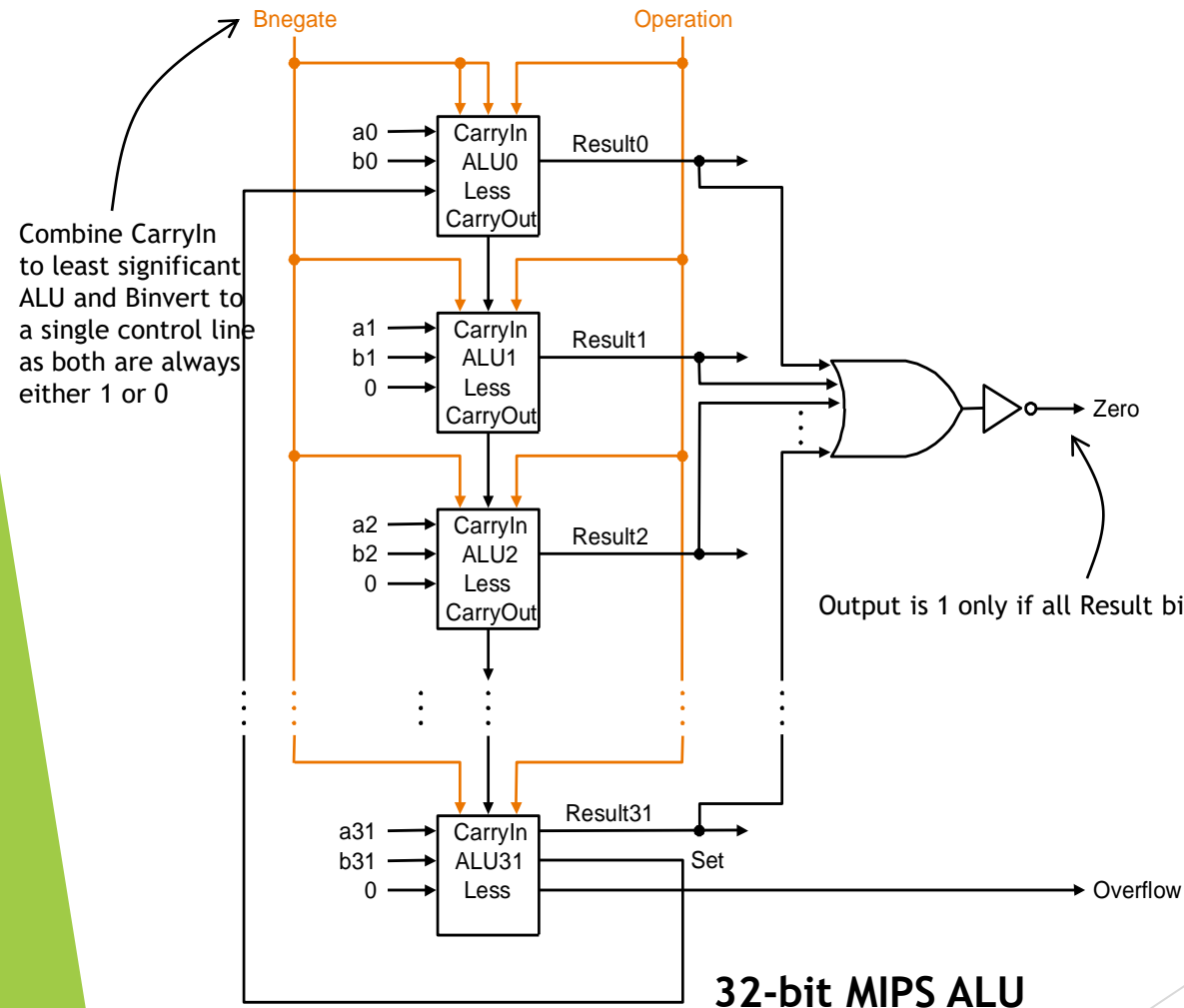
32-bit ALU from 31 copies of ALU at top left and 1 copy of ALU at bottom left in the most significant position

Tailoring the ALU to MIPS:

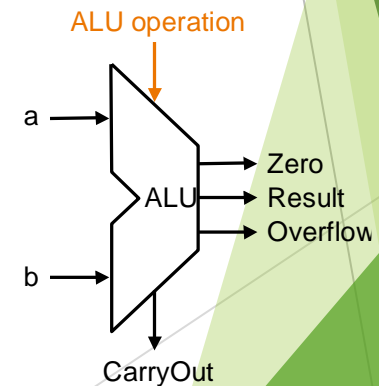
Test for Less-than and Equality

- ▶ What about logic for the *overflow bit* ?
 - ▶ overflow bit = carry *in* to msb \oplus carry *out* of msb
 - ▶ verify!
 - ▶ logic for overflow detection therefore can be put in to ALU31
- ▶ Need to support *test for equality*
 - ▶ e.g., `beq $t5, $t6, $t7`
 - ▶ use subtraction: $rs - rt = 0 \Leftrightarrow rs = rt$
 - ▶ *do we need to consider overflow?*

Supporting Test for Equality



ALU control lines		Function
Bnegate	Operation	
0	00	and
0	01	or
0	10	add
1	10	sub
1	11	slt



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 gates are always working
 - ▶ speed of a gate depends on number of inputs (fan-in) to the gate
 - ▶ speed of a circuit depends on number of gates in series (particularly, on the *critical path* to the deepest level of logic)
- ▶ Speed of MIPS operations
 - ▶ clever changes to organization can improve performance (similar to using better algorithms in software)
 - ▶ we'll look at examples for addition, multiplication and division

Problem: Ripple-carry Adder is Slow

- ▶ *Is a 32-bit ALU as fast as a 1-bit ALU? Why?*
- ▶ *Is there more than one way to do addition? Yes:*
 - ▶ one extreme: *ripple-carry* - carry ripples through 32 ALUs, slow!
 - ▶ other extreme: sum-of-products for each CarryIn bit - super fast!
 - ▶ CarryIn bits:

$$c_1 = b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0$$

Note: c_i is CarryIn bit into i th ALU;
 c_0 is the forced CarryIn into the
least significant ALU

$$\begin{aligned} c_2 &= b_1 \cdot c_1 + a_1 \cdot c_1 + a_1 \cdot b_1 \\ &= a_1 \cdot a_0 \cdot b_0 + a_1 \cdot a_0 \cdot c_0 + a_1 \cdot b_0 \cdot c_0 \quad (\text{substituting for } c_1) \\ &\quad + b_1 \cdot a_0 \cdot b_0 + b_1 \cdot a_0 \cdot c_0 + b_1 \cdot b_0 \cdot c_0 + a_1 \cdot b_1 \end{aligned}$$

$$\begin{aligned} c_3 &= b_2 \cdot c_2 + a_2 \cdot c_2 + a_2 \cdot b_2 \\ &= \dots = \text{sum of 15 4-term products...} \end{aligned}$$

- ▶ How fast? But not feasible for a 32-bit ALU! Why? Exponential complexity!!

Two-level Carry-lookahead Adder: First Level

- ▶ An approach between our two extremes
- ▶ Motivation:
 - ▶ if we didn't know the value of a carry-in, what could we do?
 - ▶ when would we always generate a carry? (generate) $g_i = a_i \cdot b_i$
 - ▶ when would we propagate the carry? (propagate) $p_i = a_i + b_i$
- ▶ Express (carry-in equations in terms of generate/propagates)
$$c_1 = g_0 + p_0 \cdot c_0$$
$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$
$$c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$
$$c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$
$$+ p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$
- ▶ Feasible for 4-bit adders - with wider adders unacceptable complexity.
 - ▶ solution: build a *first level using 4-bit adders*, then a *second level on top*

Two-level Carry-lookahead Adder:

Second Level for a 16-bit adder

- Propagate signals for each of the four 4-bit adder blocks:

$$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

$$P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$

$$P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$

$$P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$

- Generate signals for each of the four 4-bit adder blocks:

$$G_0 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$

$$G_1 = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4$$

$$G_2 = g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g_9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8$$

$$G_3 = g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}$$

Two-level Carry-lookahead Adder: Second Level for a 16-bit adder

- CarryIn signals for each of the four 4-bit adder blocks (see earlier carry-in equations in terms of generate/propagates):

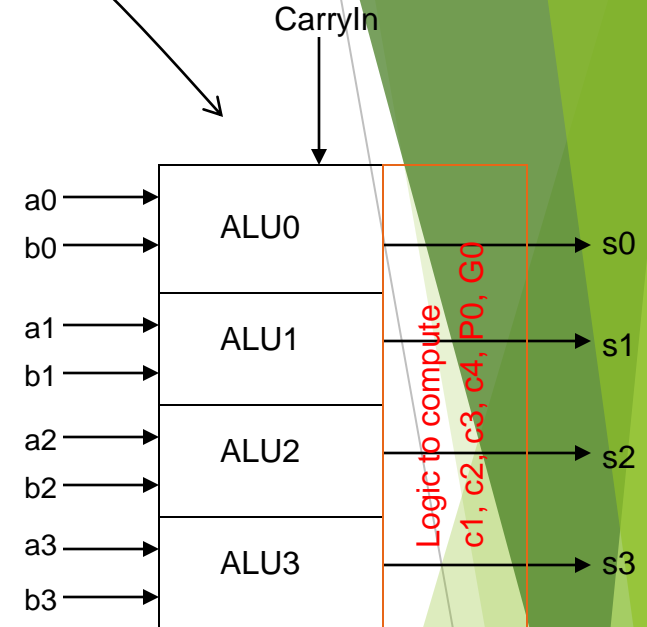
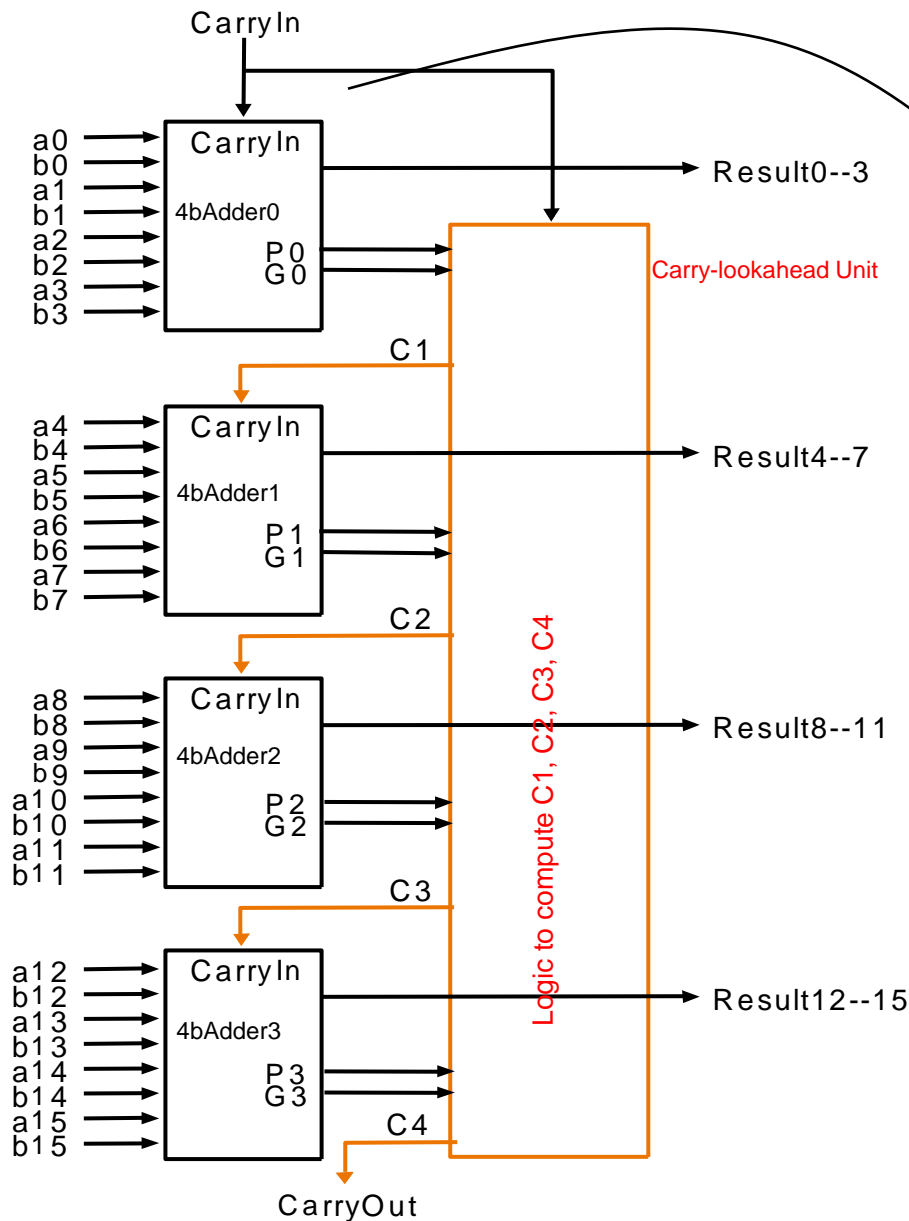
$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + \\ P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

Carry-lookahead Logic



Blow-up of 4-bit adder:
 (conceptually) consisting of four 1-bit ALUs plus logic to compute all CarryOut bits and one super generate and one super propagate bit. Each 1-bit ALU is exactly as for ripple-carry except c1, c2, c3 for ALUs 1, 2, 3 comes from the extra logic

16-bit carry-lookahead adder from four 4-bit adders and one carry-lookahead unit

Two-level Carry-lookahead Adder: Second Level for a 16-bit adder

- ▶ Two-level carry-lookahead logic *steps*:
 1. compute p_i 's and g_i 's at each 1-bit ALU
 2. compute P_i 's and G_i 's at each 4-bit adder unit
 3. compute C_i 's in carry-lookahead unit
 4. compute c_i 's at each 4-bit adder unit
 5. compute results (sum bits) at each 1-bit ALU
- ▶ E.g., add using carry-lookahead logic:
 - ▶ 0001 1010 0011 0011
 - ▶ 1110 0101 1110 1011
- ▶ *Compare times for ripple-carry vs. carry-lookahead for a 16-bit adder assuming unit delay at each gate*

Multiply

- ▶ shift-add method:

Multiplicand	1000
Multiplier	$\times 1001$
	<hr/>
	1000
	0000
	0000
	1000
Product	<u>01001000</u>

- ▶ m bits \times n bits = $m+n$ bit product

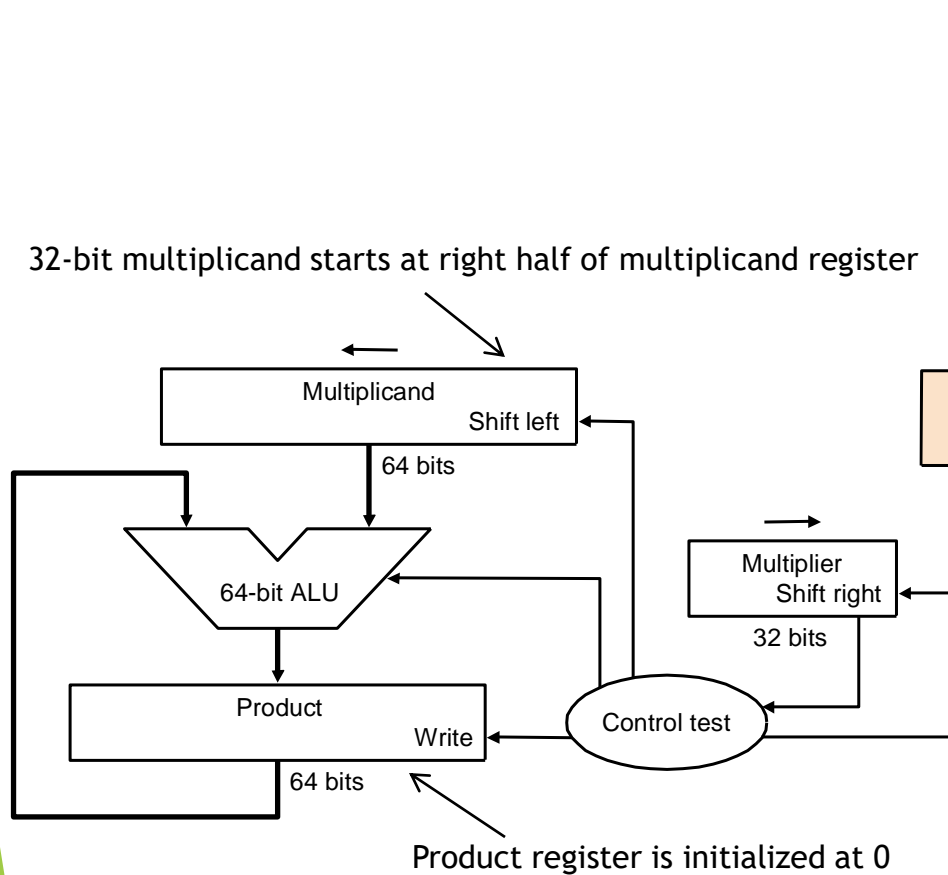
- ▶ Binary makes it easy:

- ▶ multiplier bit 1 => copy multiplicand (1 x multiplicand)

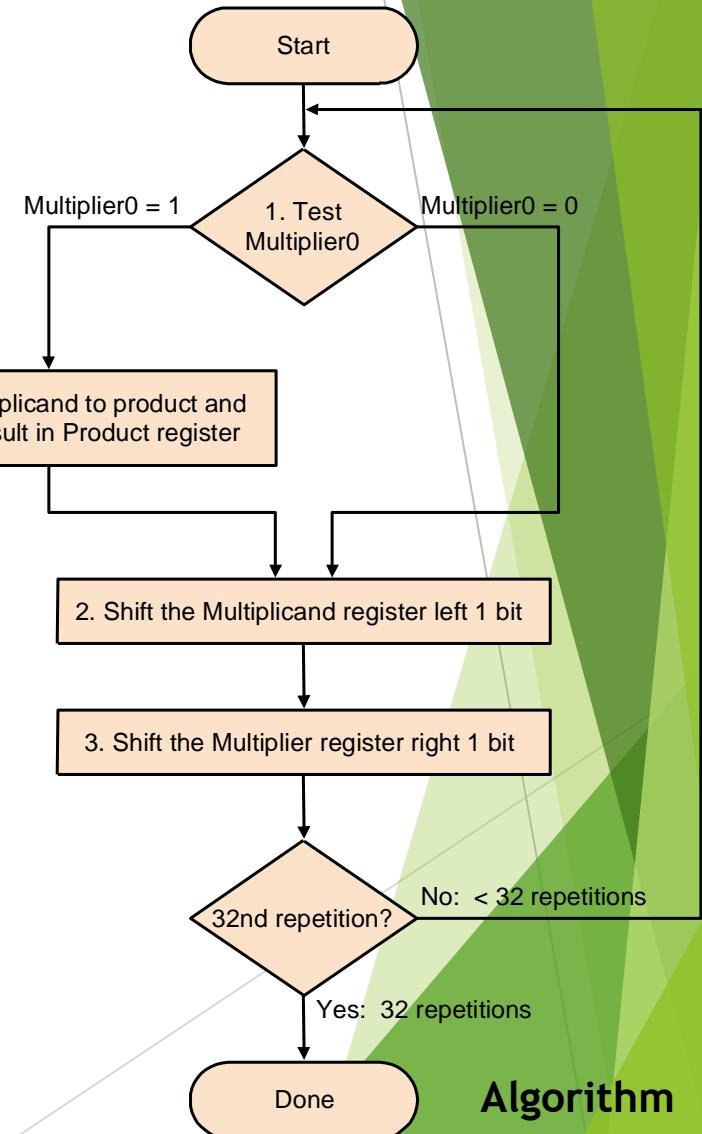
- ▶ multiplier bit 0 => place 0 (0 x multiplicand)

- ▶ 3 versions of multiply hardware & algorithm

Shift-add Multiplier Version 1

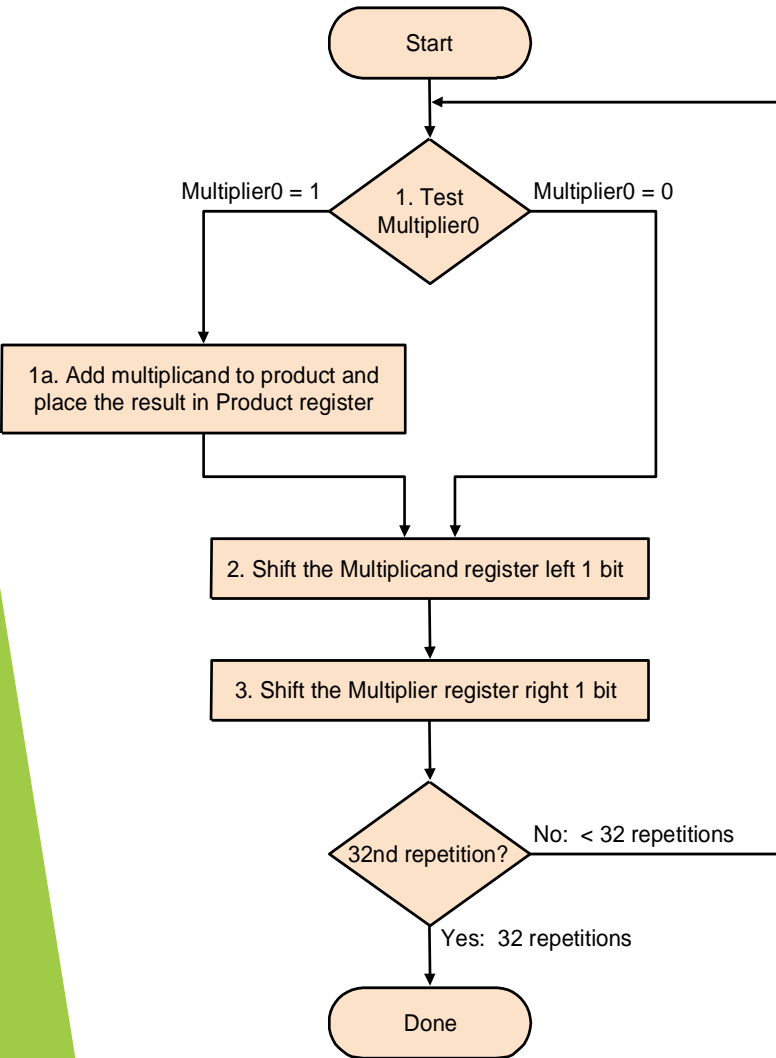


Multiplicand register, product register, ALU are 64-bit wide; multiplier register is 32-bit wide



Algorithm

Shift-add Multiplier Version1



Example: 0010 * 0011:

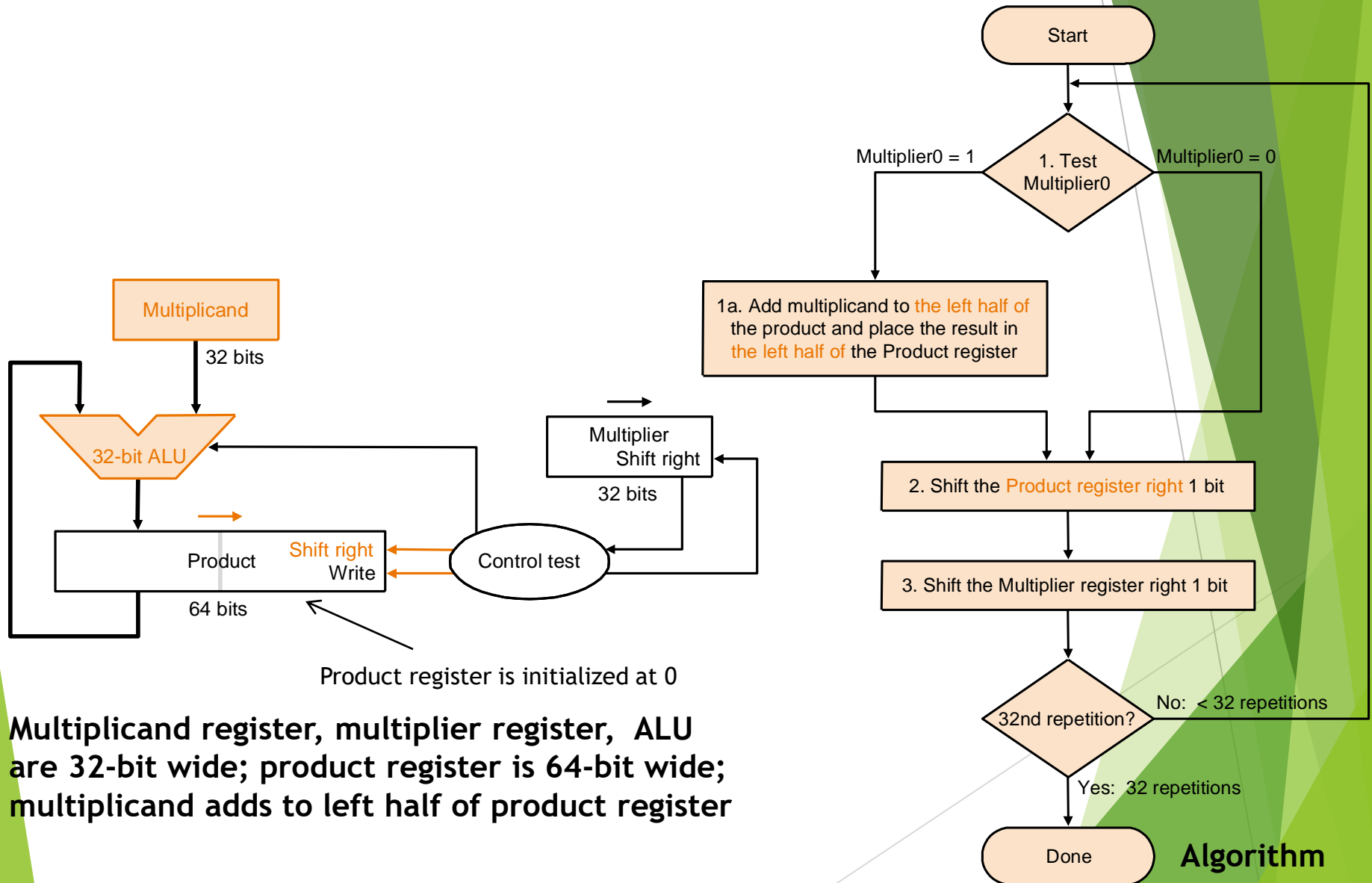
Iteration	Step	Multiplier	Multiplicand	Product
0	init values	0011	0000 0010	0000 0000
1	1a	0011	0000 0010	0000 0010
	2	0011	0000 0100	0000 0010
	3	0001	0000 0100	0000 0010
2	...			

Algorithm

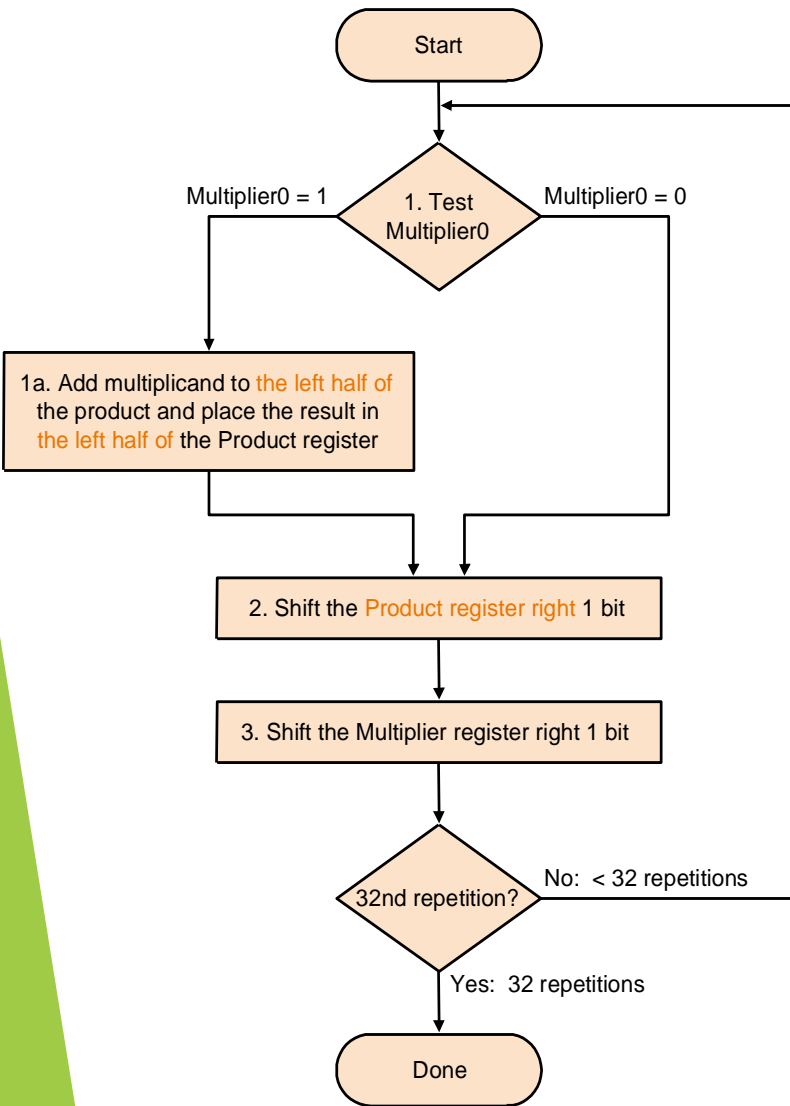
Observations on Multiply Version 1

- ▶ 1 step per clock cycle \Rightarrow nearly 100 clock cycles to multiply two 32-bit numbers
- ▶ Half the bits in the multiplicand register always 0
 \Rightarrow 64-bit adder is wasted
- ▶ 0's inserted to right as multiplicand is shifted left
 \Rightarrow least significant bits of product never change once formed
- ▶ Intuition: instead of shifting multiplicand to left, shift product to right...

Shift-add Multiplier Version 2



Shift-add Multiplier Version 2



Example: 0010 * 0011:

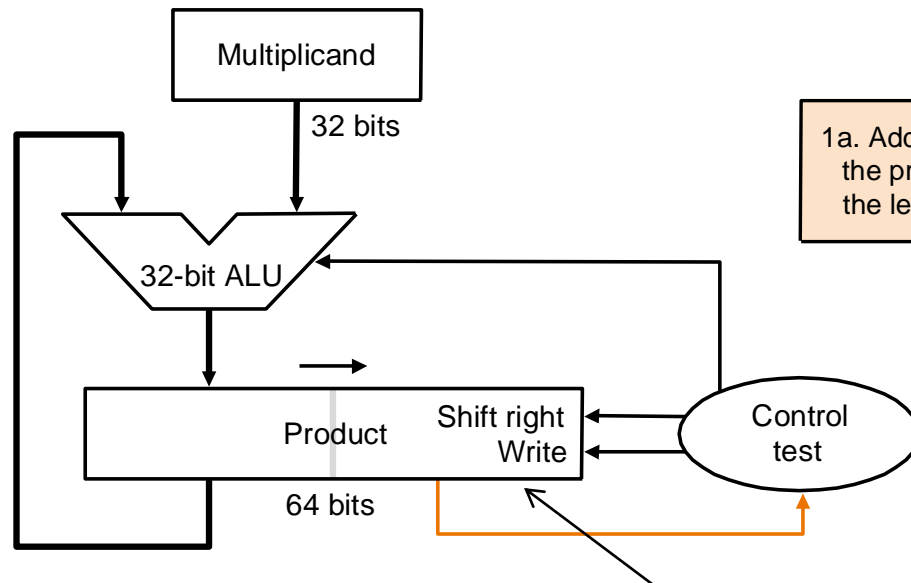
Iteration	Step	Multiplier	Multiplicand	Product
0	init values	0011	0010	0000 0000
1	1a	0011	0010	0010 0000
	2	0011	0010	0001 0000
	3	0001	0010	0001 0000
2	...			

Algorithm

Observations on Multiply Version 2

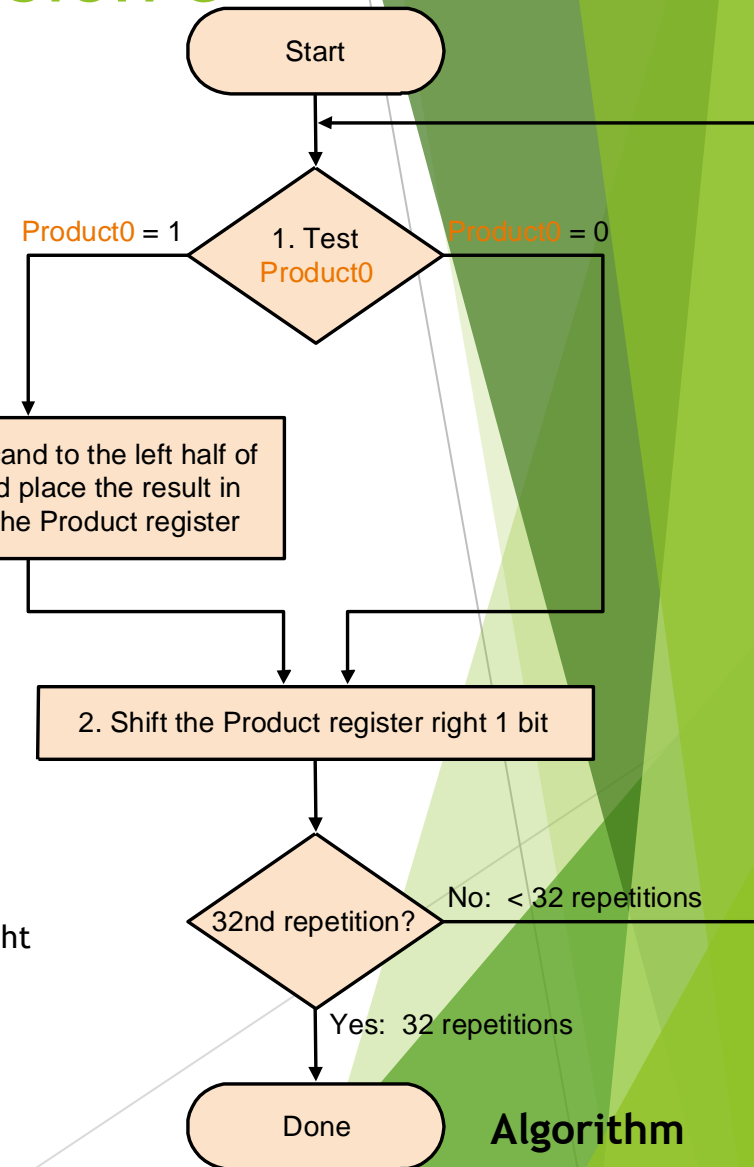
- ▶ Each step the product register wastes space that exactly matches the current size of the multiplier
- ▶ Intuition: combine multiplier register and product register...

Shift-add Multiplier Version 3



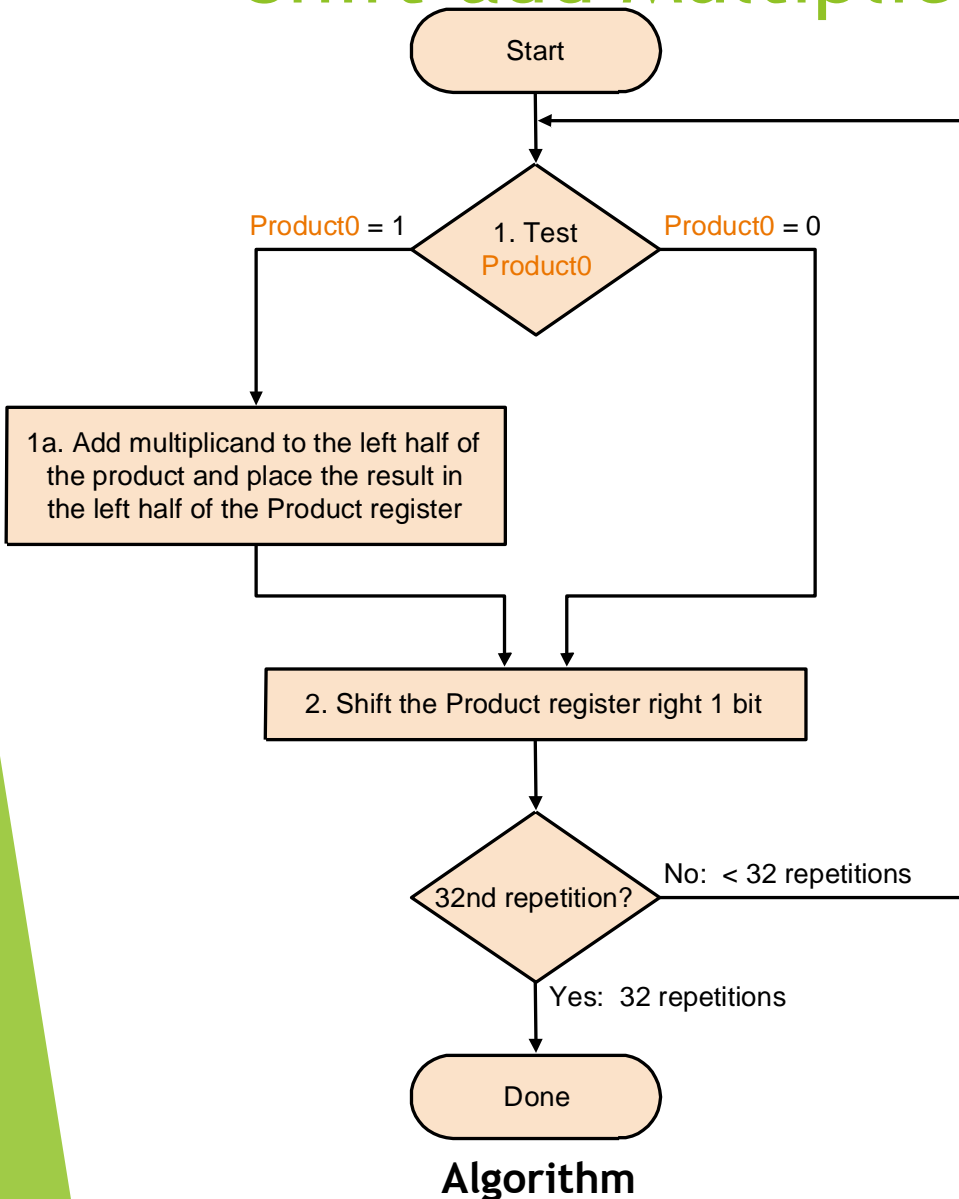
Product register is initialized with multiplier on right

No separate multiplier register; multiplier placed on right side of 64-bit product register



Algorithm

Shift-add Multiplier Version 3



Example: 0010 * 0011:

Iteration	Step	Multiplicand	Product
0	init values	0010	0000 0011
1	1a	0010	0010 0011
2	2	0010	0001 0001
2	...		

Observations on Multiply Version 3

- ▶ 2 steps per bit because multiplier & product combined
- ▶ What about *signed* multiplication?
 - ▶ easiest solution is to make both positive and remember whether to negate product when done, i.e., leave out the sign bit, run for 31 steps, then negate if multiplier and multiplicand have opposite signs
- ▶ Booth's Algorithm is an elegant way to multiply signed numbers using same hardware - it also often quicker...

Motivating Booth's algorithm

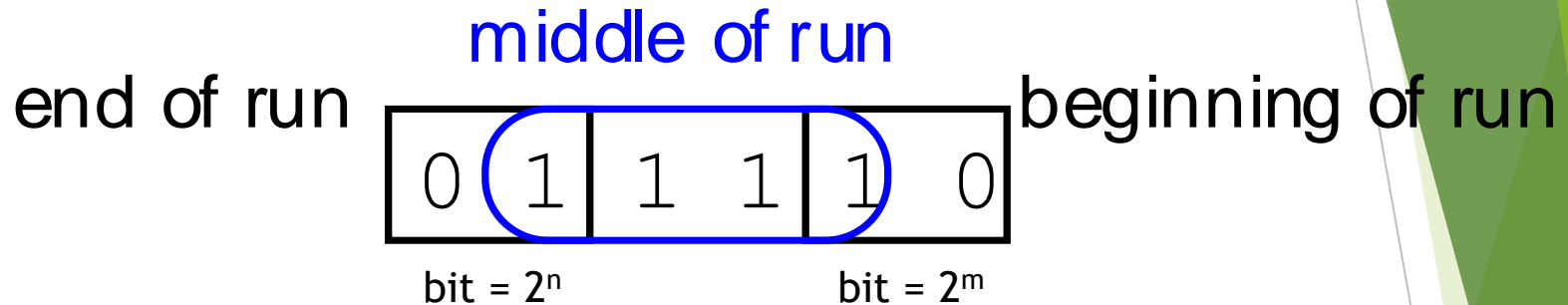
- Example $0010 * 0110$. Traditional:

$$\begin{array}{r} 0010 \\ \times 0110 \\ \hline 0000 \quad \text{shift (0 in multiplier)} \\ 0010 \quad \text{add (1 in multiplier)} \\ 0010 \quad \text{add (1 in multiplier)} \\ 0000 \quad \text{shift (0 in multiplier)} \\ \hline 00001100 \end{array}$$

- Same example. But observe there are two successive 1's in multiplier $0110 = 2^2 + 2^1 = 2^3 - 2^1$, so can replace successive 1's by subtract and then add:

$$\begin{array}{r} 0010 \\ 0110 \\ \hline 0000 \quad \text{shift (0 in multiplier)} \\ -0010 \quad \text{sub (first 1 in multiplier)} \\ 0000 \quad \text{shift (middle of string of 1's)} \\ 0010 \quad \text{add (previous step had last 1)} \\ \hline 00001100 \end{array}$$

Motivating Booth's Algorithm



- Math idea: string of 1's ...011...10... has successive 1's

value the sum $2^n + 2^{n-1} + \dots + 2^m = 2^{n+1} - 2^m$

- Replace a string of 1s in multiplier with an *initial subtract* when we first see a one and then later *add* after the last one
 - What if the string of 1's started from the left of the (2's complement) number, e.g., 11110001 – would the formula above have to be modified?!

Booth from Multiply Version 3

- *Modify Step 1* of the algorithm Multiply Version 3 to consider *2 bits* of the multiplier: *the current bit and the bit to the right* (i.e., the current bit of the previous step). Instead of two outcomes, now there are four:

<u>Case</u>	<u>Current Bit</u>	<u>Bit to the Right</u>	<u>Explanation</u>	<u>Example</u>	<u>Op</u>
1a	0	0	Middle of run of 0s	0 <u>00</u> 1111000	none
1b	0	1	End of run of 1s	00 <u>01</u> 111000	add
1c	1	0	Begins run of 1s	000111 <u>10</u> 00	sub
1d	1	1	Middle of run of 1s	00011 <u>11</u> 000	none

- *Modify Step 2* of Multiply Version 3 to *sign extend* when the product is shifted right (*arithmetic right shift*, rather than *logical right shift*) because the product is a signed number
- *Now draw the flowchart for Booth's algorithm !*
- Multiply Version 3 and Booth share the same hardware, *except* Booth requires one extra flipflop to remember the bit to the right of the current bit in the product register - which is the bit pushed out by the preceding right shift

Booth Example (2 x 7)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	10 -> sub $P = P - M$
1c.	0010	1110 0111 0	shift P (sign ext)
2.	0010	1111 0011 1	11 -> nop
1d.	0010	1111 0011 1	shift P (sign ext)
2.	0010	1111 1001 1	11 -> nop
1d.	0010	1111 1001 1	shift P (sign ext)
2.	0010	1111 1100 1	01 -> add $P = P + M$
1b.	0010	0001 1100 1	shift P (sign ext)
2.	0010	0000 1110 0	done

Booth Algorithm (2 * -3)

Operation	Multiplicand	Product	next?
0.initial value	0010	0000 1101 0	10 -> sub P = P - M
1c.	0010	1110 1101 0	shift P (sign ext)
2.	0010	1111 0110 1	01 -> add P = P + M
1b.	0010	0001 0110 1	shift P (sign ext)
2.	0010	0000 1011 0	10 -> sub P = P - M
1c.	0010	1110 1011 0	shift P
2.	0010	1111 0101 1	11 -> nop
1d.	0010	1111 0101 1	shift P
2.	0010	1111 1010 1	done

Verifying Booth's Algorithm

► multiplier $a = a_{31} a_{32} \dots a_0$, multiplicand = b

► $a_i \quad a_{i-1} \quad \text{Operation}$

0 0 nop

0 1 add b

1 0 sub b

1 1 nop

►
i.e., if $a_{i-1} - a_i = \begin{cases} 0, & \text{nop} \\ +1, & \text{add } b \\ -1, & \text{sub } b \end{cases}$

► Therefore, Booth computes sum:

$$(a_{-1} - a_0) * b * 2^0$$

$$+ (a_0 - a_1) * b * 2^1$$

$$+ (a_1 - a_2) * b * 2^2$$

...

$$+ (a_{30} - a_{31}) * b * 2^{31}$$

= ... *simplify telescopic sum!* ...

MIPS Notes

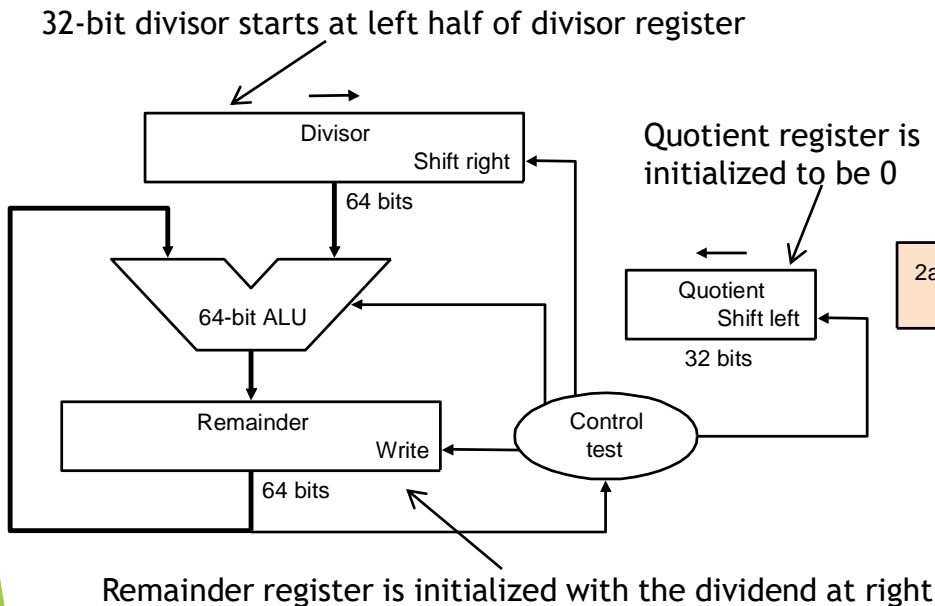
- ▶ MIPS provides two 32-bit registers `Hi` and `Lo` to hold a 64-bit product
- ▶ `mult`, `multu` (unsigned) put the product of two 32-bit register operands into `Hi` and `Lo`: overflow is ignored by MIPS but can be detected by programmer by examining contents of `Hi`
- ▶ `mflo`, `mfhi` moves content of `Hi` or `Lo` to a general-purpose register
- ▶ Pseudo-instructions `mul` (without overflow), `mulo` (with overflow), `mulou` (unsigned with overflow) take three 32-bit register operands, putting the product of two registers into the third

Divide

		1001	Quotient
Divisor 1000		1001010	Dividend
		<u>-1000</u>	
		10	
		101	
		1010	
		<u>-1000</u>	
		10	Remainder

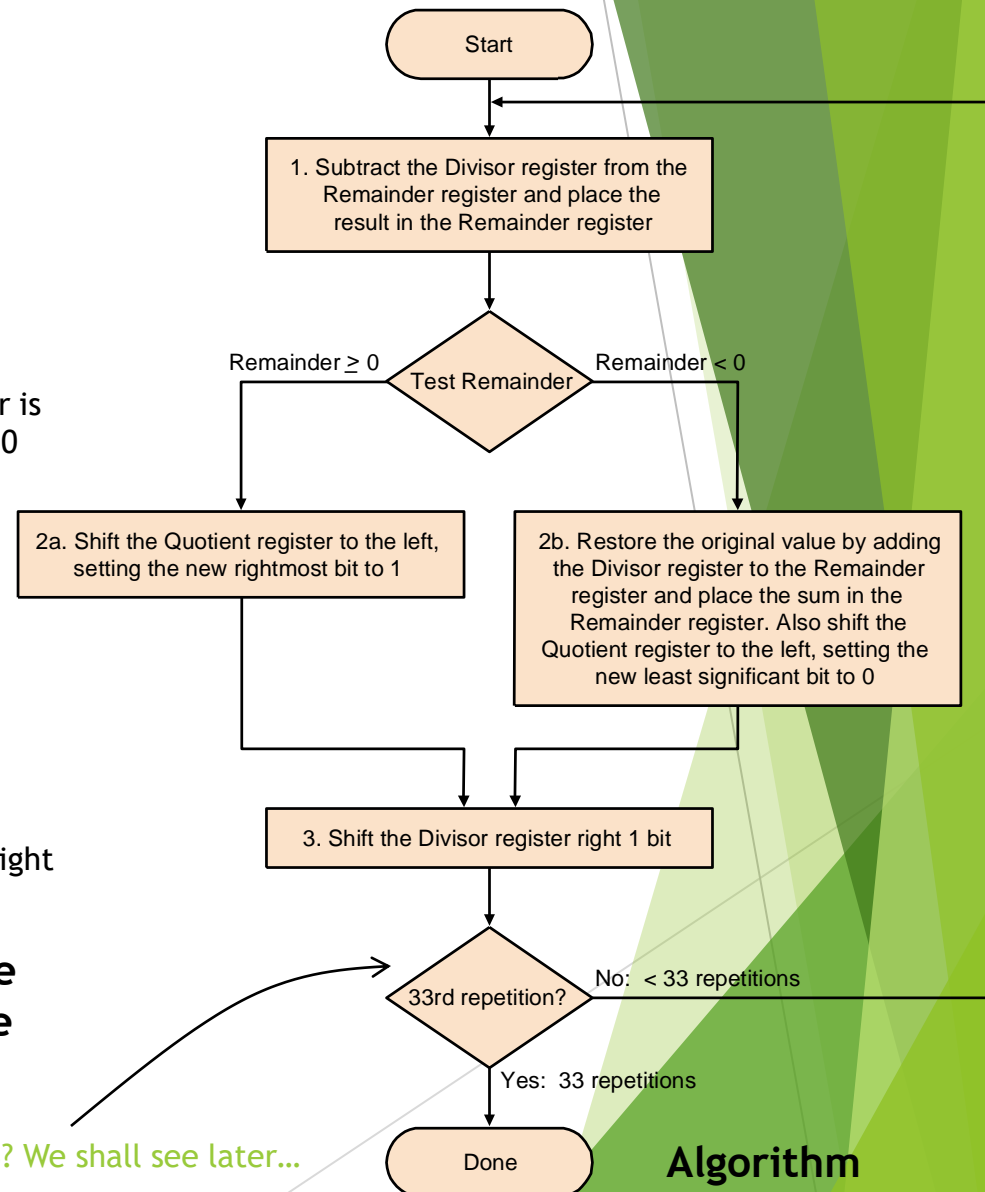
- ▶ see how big a multiple of the divisor can be subtracted, creating quotient digit at each step
- ▶ Binary makes it easy \Rightarrow *first*, try $1 * \text{divisor}$; *if too big*, $0 * \text{divisor}$
- ▶ Dividend = (Quotient * Divisor) + Remainder
- ▶ 3 versions of divide hardware & algorithm:

Divide Version 1



Divisor register, remainder register, ALU are 64-bit wide; quotient register is 32-bit wide

Why 33? We shall see later...

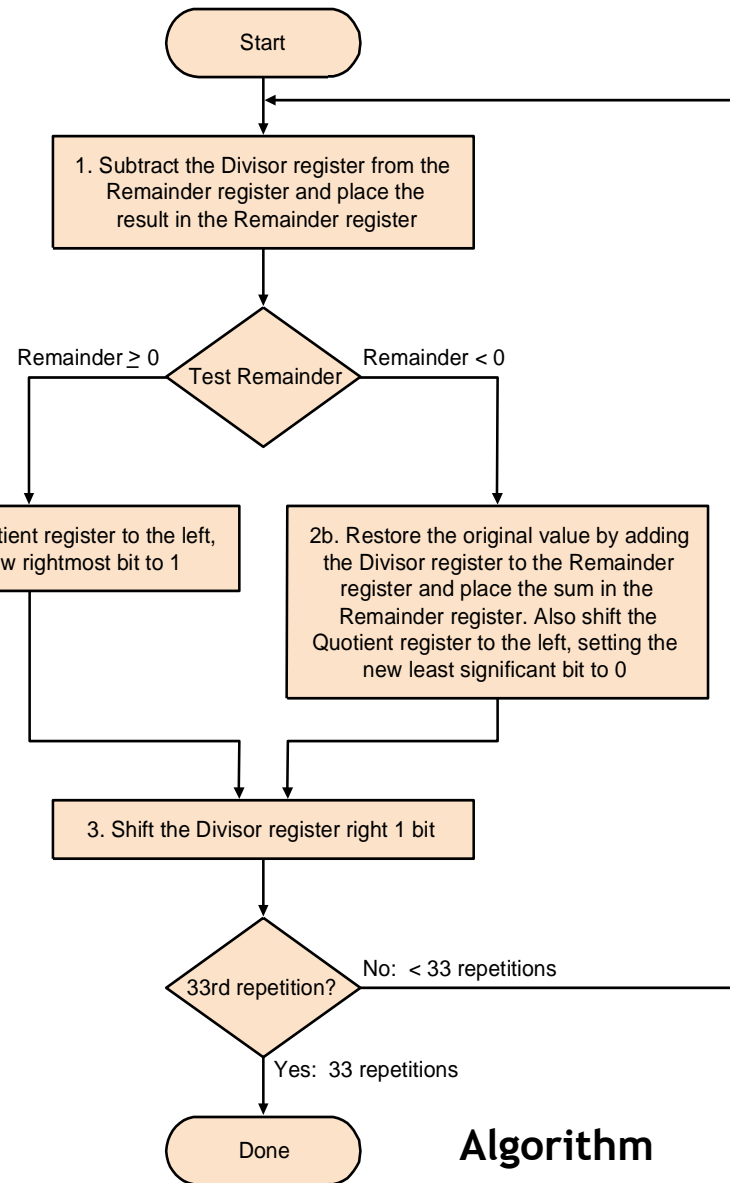


Algorithm

Divide Version 1

Example: 0111 / 0010:

Iteration	Step	Quotient	Divisor	Remainder
0	init	0000	0010 0000	0000 0111
1	1	0000	0010 0000	1110 0111
	2b	0000	0010 0000	0000 0111
	3	0000	0001 0000	0000 0111
2	...			
3				
4				
5				

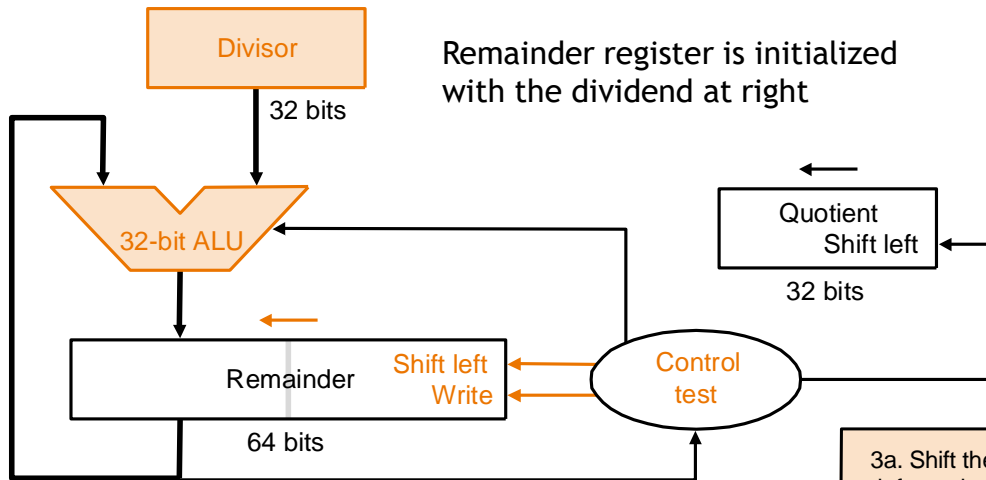


Algorithm

Observations on Divide Version 1

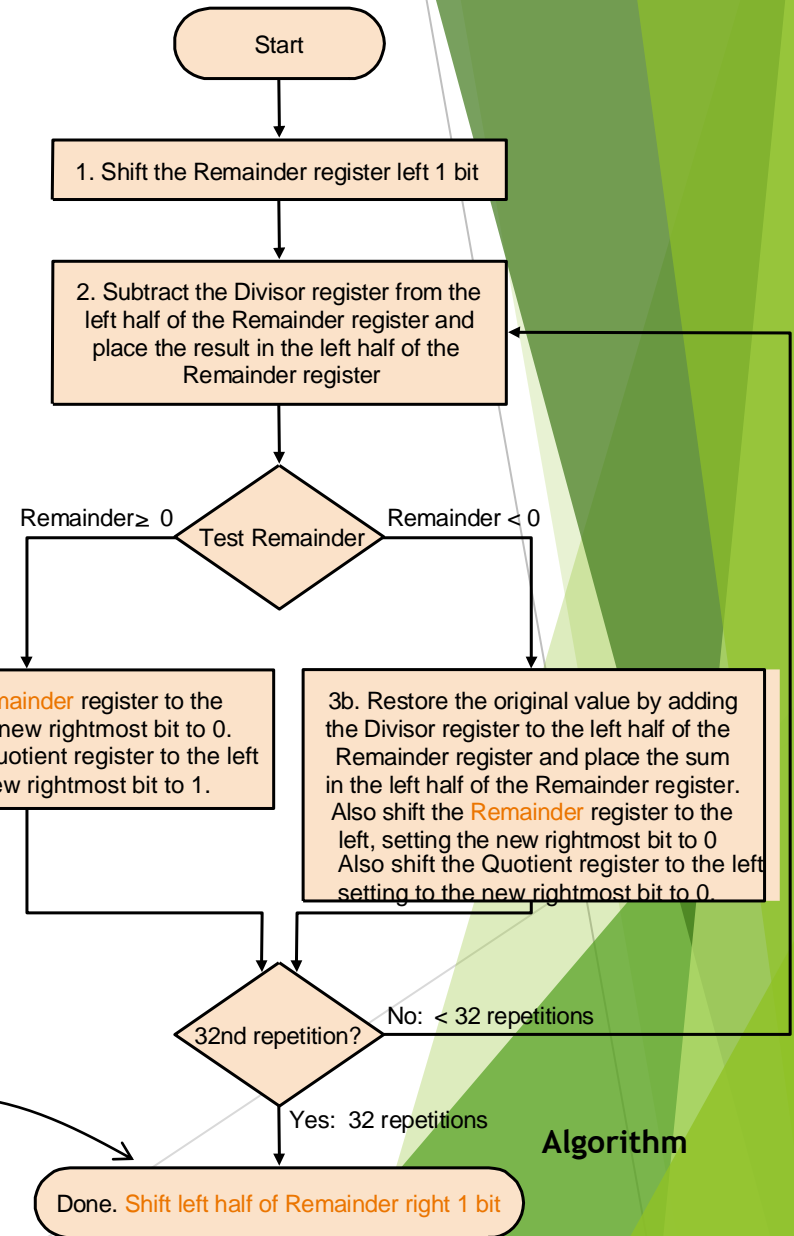
- ▶ Half the bits in divisor always 0
 - ▶ \Rightarrow 1/2 of 64-bit adder is wasted
 - ▶ \Rightarrow 1/2 of divisor register is wasted
- ▶ Intuition: instead of shifting divisor to right, shift remainder to left...
- ▶ Step 1 cannot produce a 1 in quotient bit - as all bits corresponding to the divisor in the remainder register are 0 (remember all operands are 32-bit)
- ▶ Intuition: switch order to shift first and then subtract - can save 1 iteration...

Divide Version 2



Divisor register, quotient register, ALU are 32-bit wide; remainder register is 64-bit wide

Why this correction step? We shall see later...

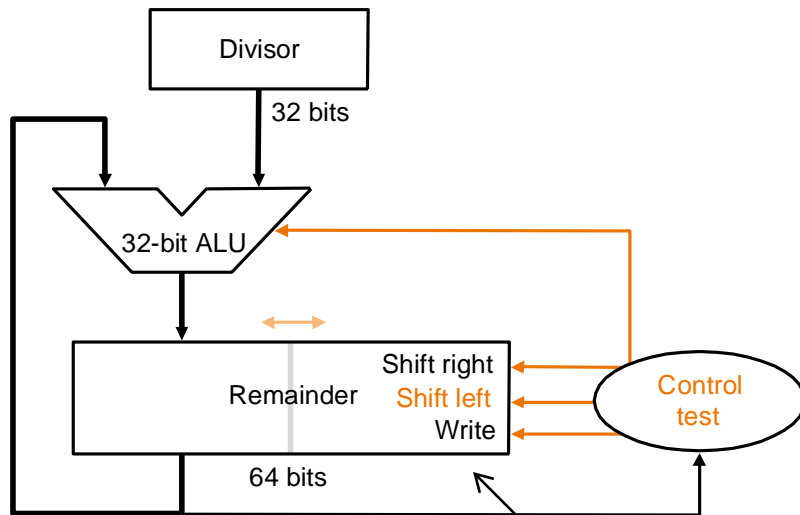


Algorithm

Observations on Divide Version 2

- ▶ Each step the remainder register wastes space that exactly matches the current size of the quotient
- ▶ Intuition: combine quotient register and remainder register...

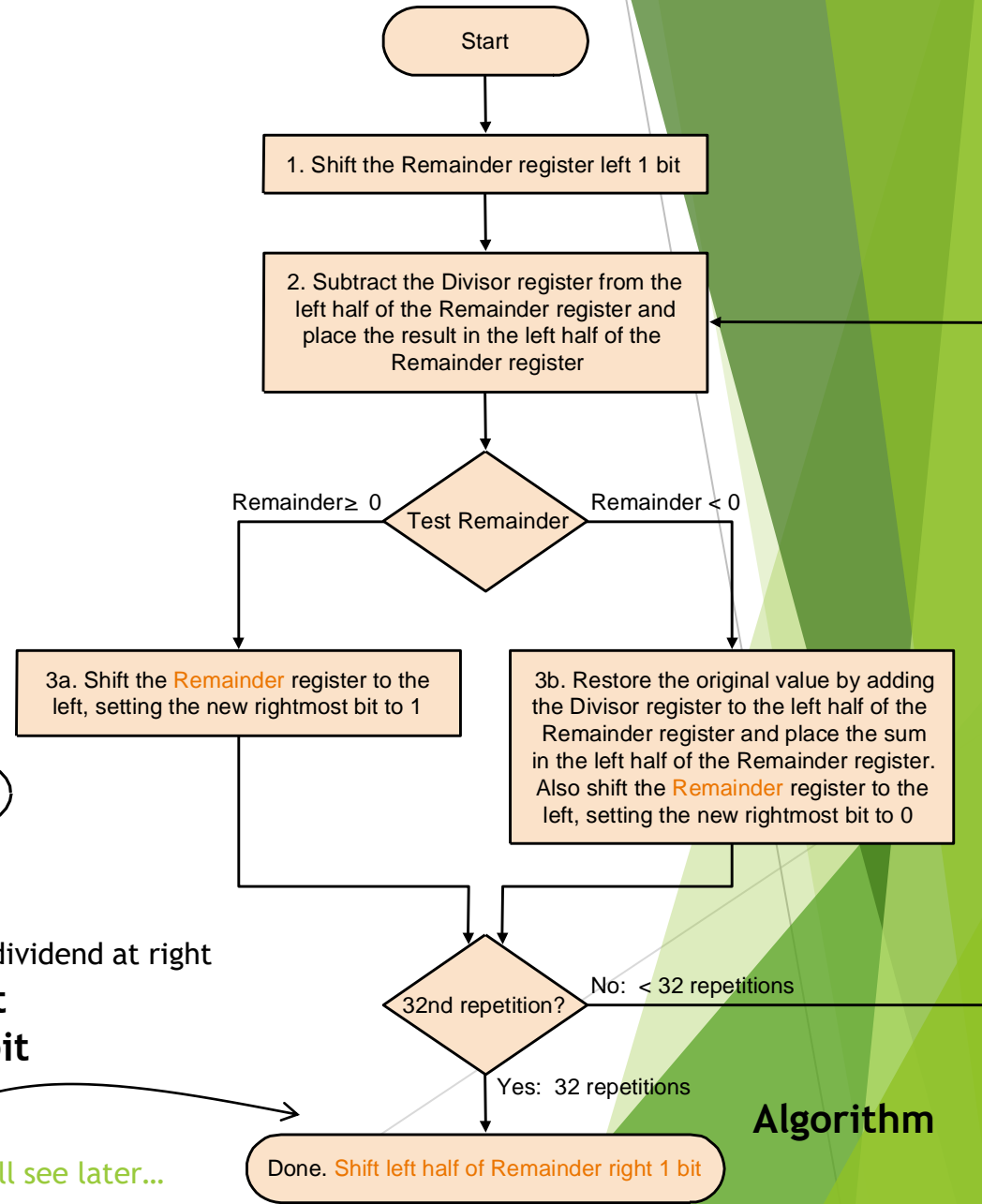
Divide Version 3



Remainder register is initialized with the dividend at right

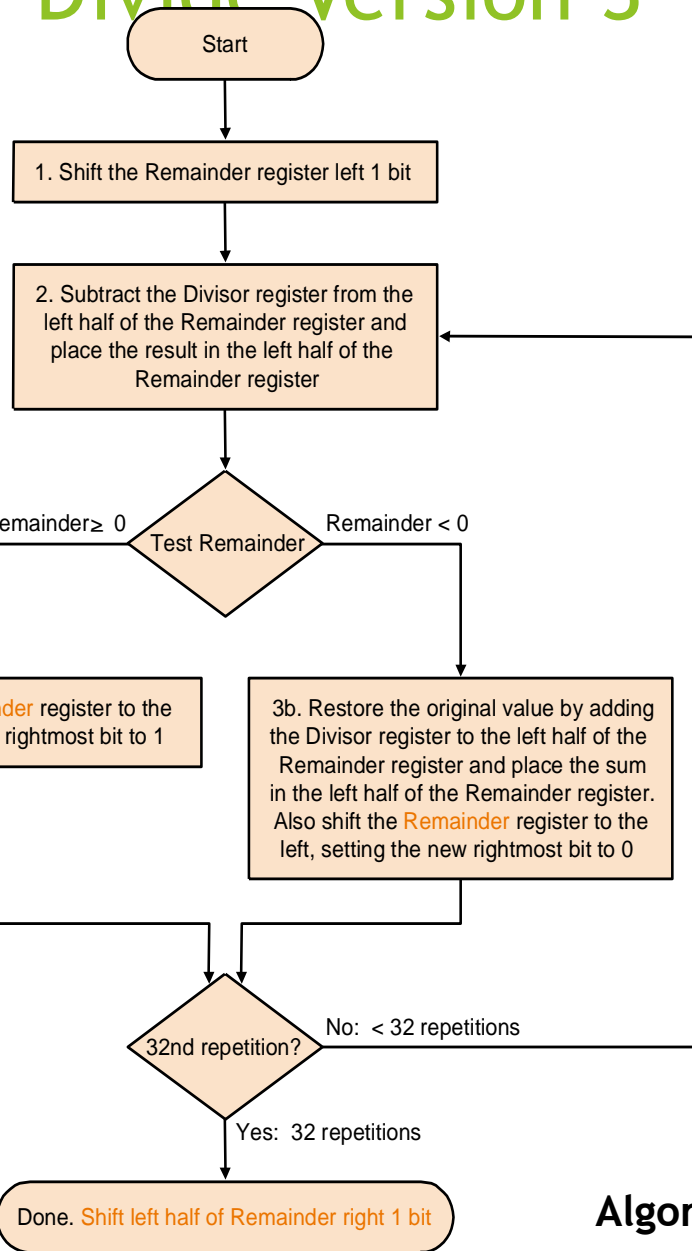
No separate quotient register; quotient is entered on the right side of the 64-bit remainder register

Why this correction step? We shall see later...



Algorithm

Divide Version 3



Example: 0111 / 0010:

Iteration	Step	Divisor	Remainder
0	init	0010	0000 0111
	1	0010	0000 1110
1	2	0010	1110 1110
	3b	0010	0001 1100
2	...		
3			
4			

Algorithm

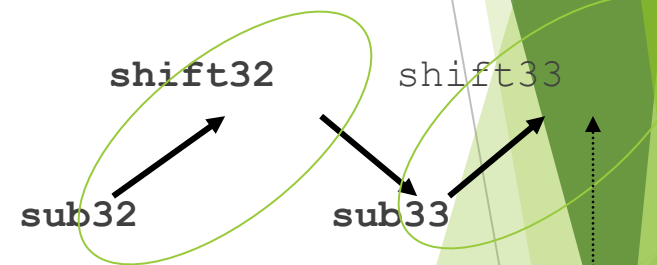
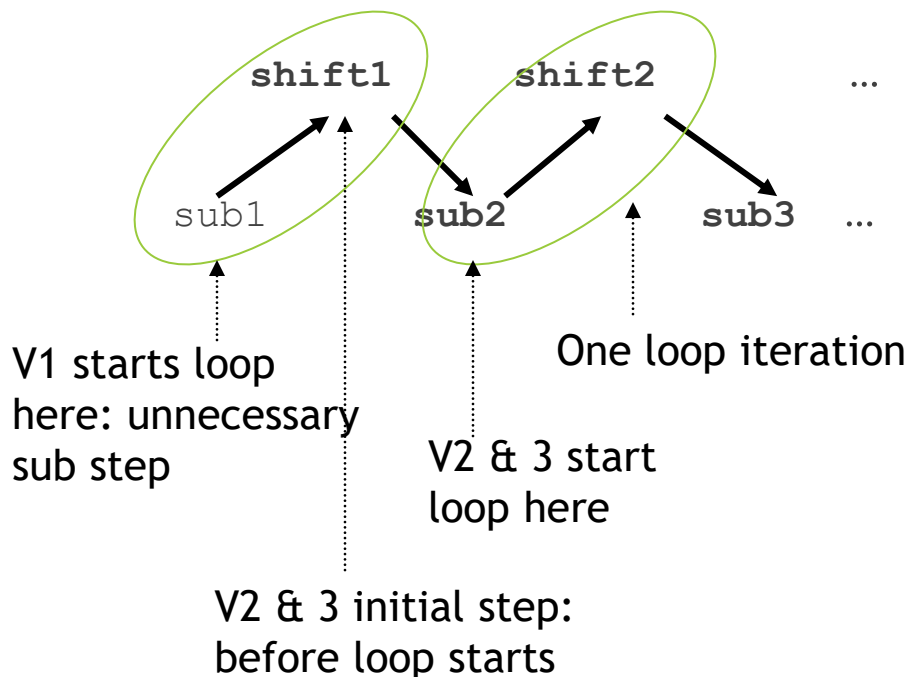
Number of Iterations

- ▶ Why the extra iteration in Version 1?
- ▶ Why the final correction step in Versions 2 & 3?

Ovals represent loop iterations

Shift: see the version descriptions for which registers are shifted

Main insight - $\text{sub}(i+1)$ must actually follow shift_i of the divisor (or remainder, depending on version) and the resulting bit in the quotient appears on $\text{shift}(i+1)$



Critical situation! Only the quotient shift is necessary as it corresponds to the outcome of the previous sub. So V1 is ok even though the last divisor shift is redundant, as final divisor is ignored anyway; V2 & 3 must repair remainder as it has shifted left one time too many

Observations on Divide Version 3

- ▶ *Same hardware as Multiply Version 3*
- ▶ *Signed divide:*
 - ▶ make both divisor and dividend positive and perform division
 - ▶ negate the quotient if divisor and dividend were of opposite signs
 - ▶ make the sign of the remainder match that of the dividend
 - ▶ this ensures always
 - ▶ $\text{dividend} = (\text{quotient} * \text{divisor}) + \text{remainder}$
 - ▶ $-\text{quotient} (x/y) = \text{quotient} (-x/y)$ (e.g. $7 = 3*2 + 1$ & $-7 = -3*2 - 1$)

MIPS Notes

- ▶ `div` (signed), `divu` (unsigned), with two 32-bit register operands, divide the contents of the operands and put remainder in `Hi` register and quotient in `Lo`; overflow is ignored in both cases
- ▶ pseudo-instructions `div` (signed with overflow), `divu` (unsigned without overflow) with three 32-bit register operands puts quotients of two registers into third

Floating Point

- ▶ We need a way to represent
 - ▶ numbers with fractions, e.g., 3.1416
 - ▶ very small numbers (in absolute value), e.g., .00000000023
 - ▶ very large numbers (in absolute value) , e.g., -3.15576×10^{46}
- ▶ Representation:
 - ▶ *scientific*: sign, exponent, significand form: binary point
▶ $(-1)^{\text{sign}} * \text{significand} * 2^{\text{exponent}}$. E.g., $-101.001101 * 2^{111001}$
 - ▶ more bits for *significand* gives more accuracy
 - ▶ more bits for *exponent* increases range
 - ▶ if $1 \leq \text{significand} < 10_{\text{two}} (=2_{\text{ten}})$ then number is *normalized*, **except** for number 0 which is normalized to significand 0
 - ▶ E.g., $-101.001101 * 2^{111001} = -1.01001101 * 2^{111011}$ (normalized)

IEEE 754 Floating-point Standard

- ▶ IEEE 754 floating point standard:

- ▶ single precision: one word

31	bits 30 to 23	bits 22 to 0
sign	8-bit exponent	23-bit significand

- ▶ double precision: two words

31	bits 30 to 20	bits 19 to 0
sign	11-bit exponent	upper 20 bits of 52-bit significand
bits 31 to 0		
lower 32 bits of 52-bit significand		

IEEE 754 Floating-point Standard

- ▶ Sign bit is 0 for positive numbers, 1 for negative numbers
- ▶ Number is assumed normalized and leading 1 bit of significand left of binary point (for non-zero numbers) is *assumed* and not shown
 - ▶ e.g., significand 1.1001... is represented as 1001...,
 - ▶ **exception** is number 0 which is represented as all 0s (see next slide)
 - ▶ for other numbers:

$$\text{value} = (-1)^{\text{sign}} * (1 + \text{significand}) * 2^{\text{exponent value}}$$

- ▶ Exponent is *biased* to make sorting easier
 - ▶ all 0s is smallest exponent, all 1s is largest
 - ▶ bias of 127 for single precision and 1023 for double precision
 - ▶ therefore, for non-0 numbers:

$$\text{value} = (-1)^{\text{sign}} * (1 + \text{significand}) * 2^{\overbrace{(\text{exponent} - \text{bias})}^{\text{equals exponent value}}}$$

IEEE 754 Floating-point Standard

- ▶ Special treatment of 0:
 - ▶ if exponent is all 0 and significand is all 0, then the value is 0 (sign bit may be 0 or 1)
 - ▶ if exponent is all 0 and significand is *not* all 0, then the value is $(-1)^{\text{sign}} * (1 + \text{significand}) * 2^{-127}$
 - ▶ therefore, all 0s is taken to be 0 and not 2^{-127} (as would be for a non-zero normalized number); similarly, 1 followed by all 0's is taken to be 0 and not -2^{-127}
- ▶ *Example* : Represent -0.75_{ten} in IEEE 754 single precision
 - ▶ decimal: $-0.75 = -3/4 = -3/2^2$
 - ▶ binary: $-11/100 = -.11 = -1.1 \times 2^{-1}$
 - ▶ IEEE single precision floating point exponent = bias + exponent value
 $= 127 + (-1) = 126_{\text{ten}} = 01111110_{\text{two}}$
 - ▶ IEEE single precision: 10111111010000000000000000000000

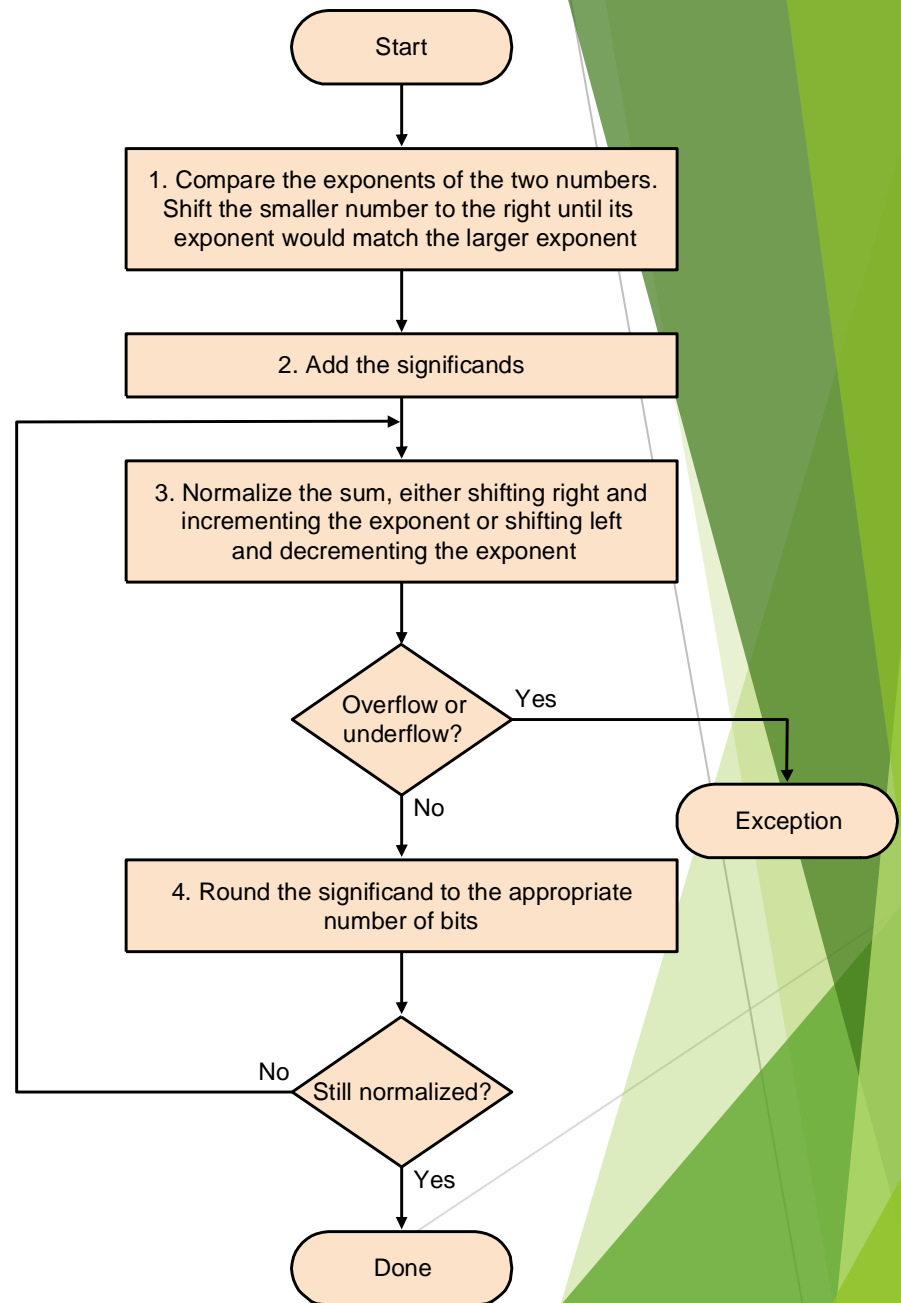
↑
sign

{
exponent

{
significand

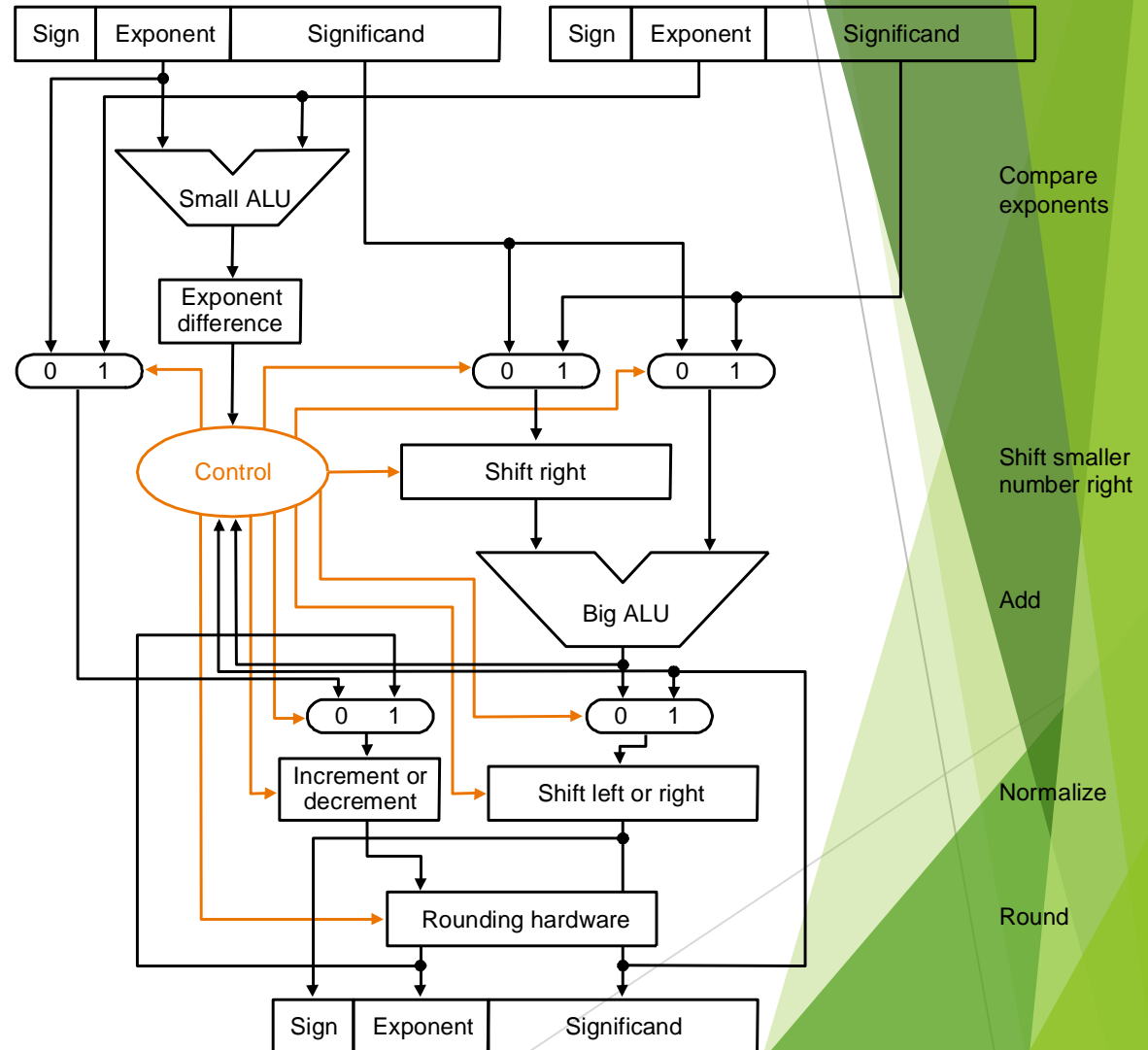
Floating Point Addition

► Algorithm:



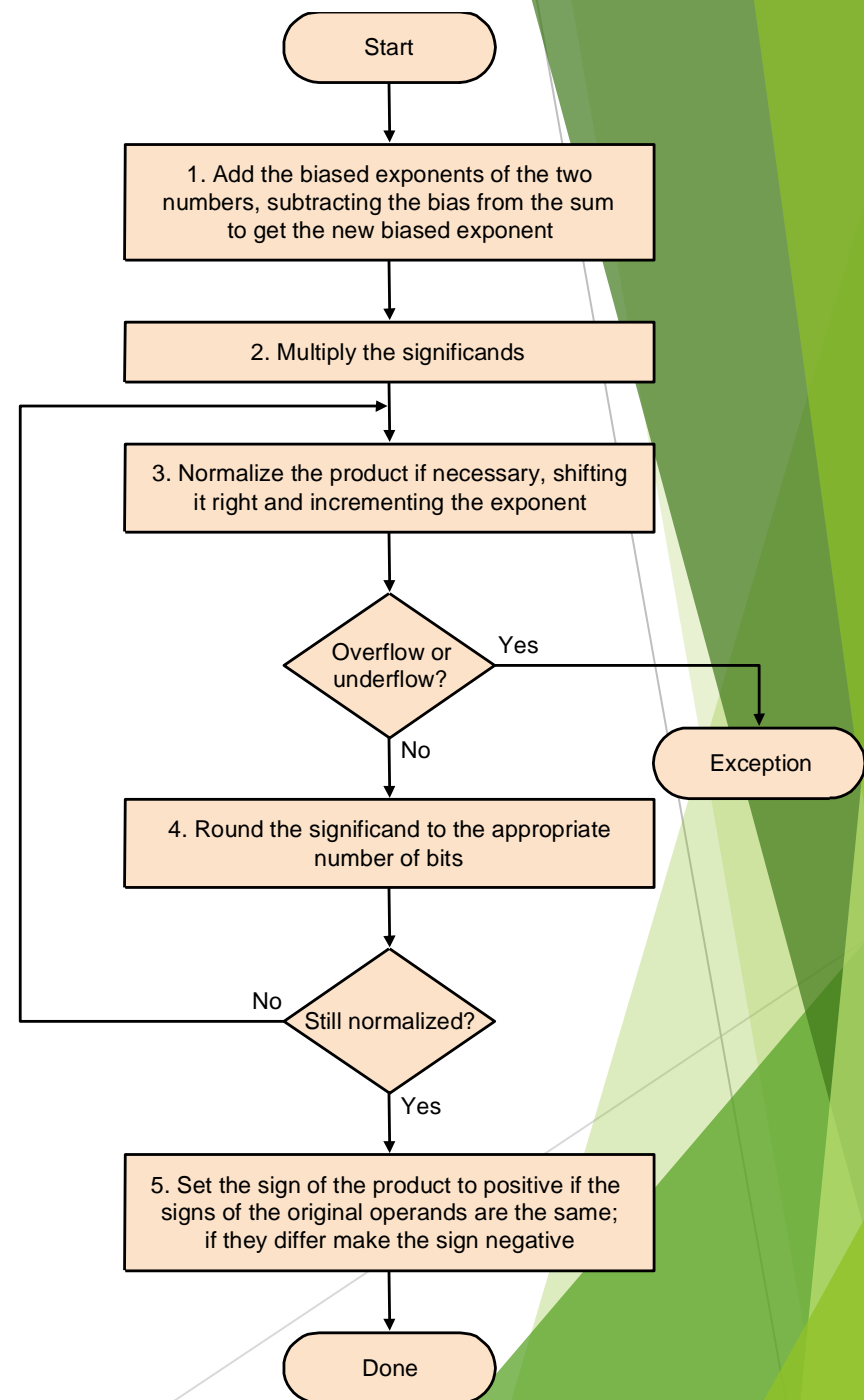
Floating Point Addition

► Hardware:



Floating Point Multiplication

► Algorithm:



Floating Point Complexities

- ▶ In addition to *overflow* we can have *underflow* (number too small)
- ▶ *Accuracy* is the problem with both overflow and underflow because we have only a finite number of bits to represent numbers that may actually require arbitrarily many bits
 - ▶ limited precision \Rightarrow rounding \Rightarrow rounding error
 - ▶ IEEE 754 keeps *two extra bits*, *guard* and *round*
 - ▶ four rounding modes
 - ▶ positive divided by zero yields *infinity*
 - ▶ zero divide by zero yields *not a number*
 - ▶ other complexities
- ▶ Implementing the standard can be tricky
- ▶ Not implementing the standard can be even worse
 - ▶ see text for discussion of Pentium bug!

MIPS Floating Point

- ▶ MIPS has a *floating point coprocessor* (numbered 1, SPIM) with thirty-two 32-bit registers \$f0 - \$f31. Two of these are required to hold doubles. Floating point instructions must use only even-numbered registers (including those operating on single floats). SPIM simulates MIPS floating point.
- ▶ Floating point *arithmetic*: `add.s` (single addition), `add.d` (double addition), `sub.s`, `sub.d`, `mul.s`, `mul.d`, `div.s`, `div.d`
- ▶ Floating point *comparison*: `c.x.s` (single), `c.x.d` (double), where `x` may be `eq`, `neq`, `lt`, `le`, `gt`, `ge`
- ▶ Other instructions...

Summary

- ▶ Computer arithmetic is constrained by limited precision
- ▶ Bit patterns have no inherent meaning but standards do exist:
 - ▶ two's complement
 - ▶ IEEE 754 floating point
- ▶ Computer instructions determine *meaning* of the bit patterns.
- ▶ Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation)
- ▶ Read *Computer Arithmetic Algorithms* by I. Koren
 - ▶ it is easy-to-read and shows new algorithms for arithmetic