Computer Organization and Architecture

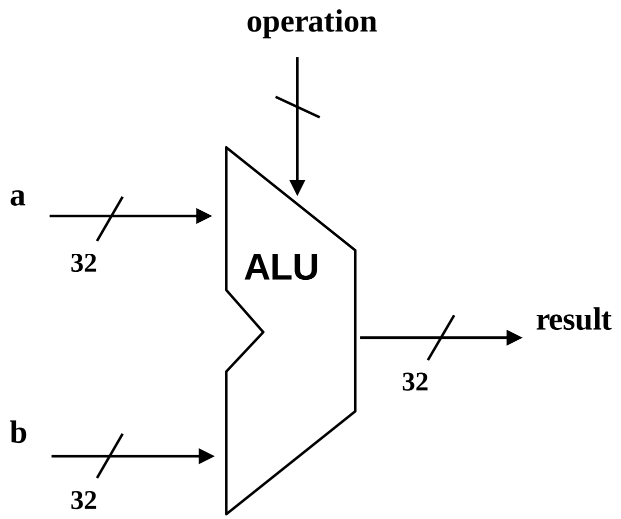
Instructor: Dr. Rushdi Abu Zneit

Slide Sources: Patterson & Hennessy

Ch. 4
Arithmetic for Computers

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)
    - $\underbrace{\text{0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...}}_{\text{n bits}}$
    - decimal: 0, …, 2^n-1
- 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:	One's Complement	Two's Complement
000 = 0	000 = 0	000 = 0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = 0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = 0	111 = -1

ambiguous zero

ambiguous zero

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:

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

or

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

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

- One's complement:

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

MIPS – 2's complement

- 32 bit signed numbers:

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

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

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

...

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

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

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

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

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

...

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

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

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

maxint

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!
  - *why does this work? Remember we don't know how to add in 2's complement yet! Later…!*
- 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 junior school (carry/borrow 1s)
  - Examples (4-bits):

0101	0110	1011	1001	1111
$\underline{0001}$	$\underline{0101}$	$\underline{0111}$	$\underline{1010}$	$\underline{1110}$

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	$0 \le X' + Y' < 2^n$ (*no CarryIn to last bit*)	$2^n \le X' + Y' < 2^{n+1} - 1$ (*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 $X + (\overline{X} + 1) = (X + \overline{X}) + 1$:

    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

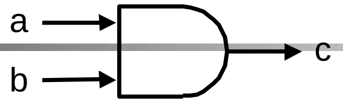
- Consider the operations A + B, and A – B
  - *can overflow occur if B is 0 ?*
  - *can overflow occur if A is 0 ?*

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

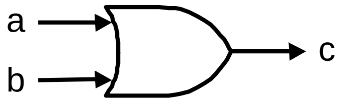
Review: Basic Hardware

1. AND gate (c = a . b)



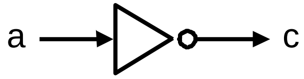
a	b	c = a . b
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate (c = a + b)



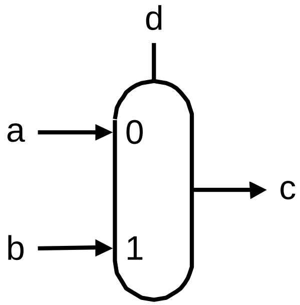
a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter (c = a)



a	c = ā
0	1
1	0

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



d	c
0	a
1	b

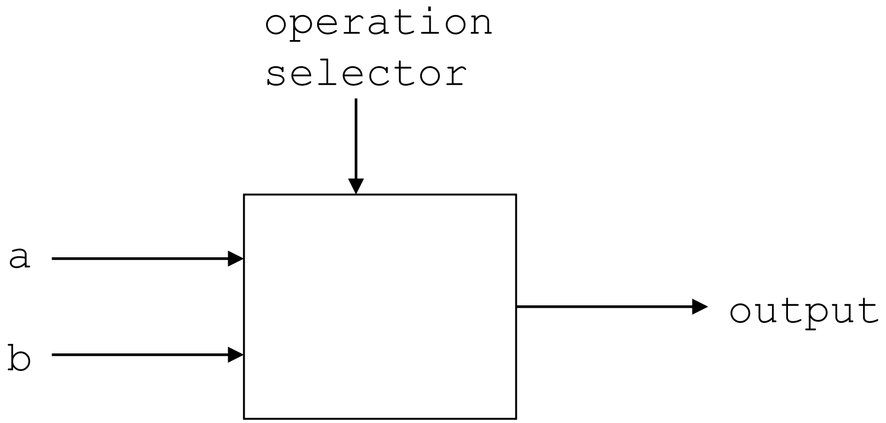
Review: Boolean Algebra & Gates

- *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

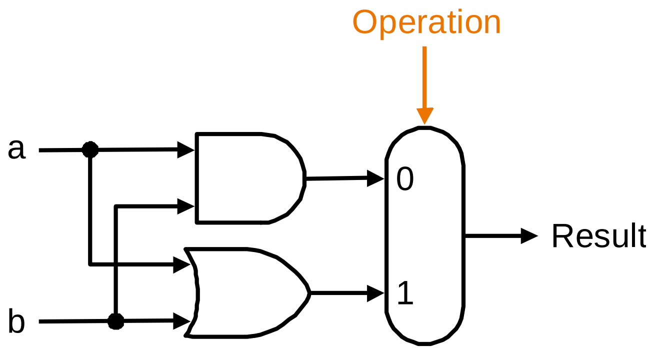
- To warm up let's build a logic unit to support the `and` and `or` instructions for MIPS (32-bit registers)
  - we'll just build a 1-bit unit and use 32 of them

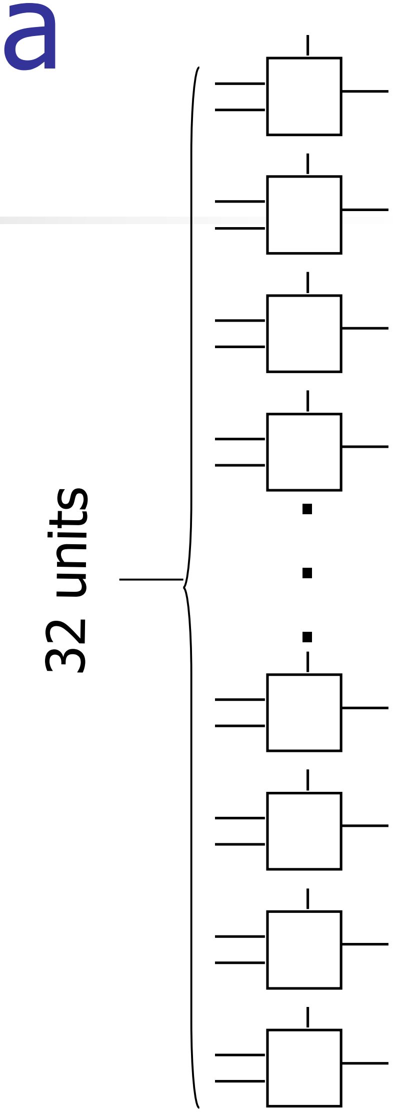


- Possible implementation using a *multiplexor* :

Implementation with a Multiplexor

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

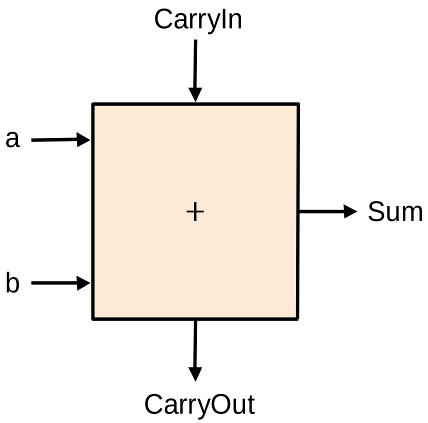




- Lets build our ALU using a MUX (multiplexor):

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:



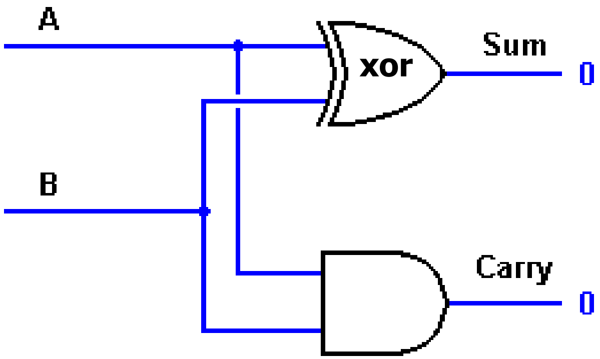
$$c_{out} = a.b + a.c_{in} + b.c_{in}$$

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

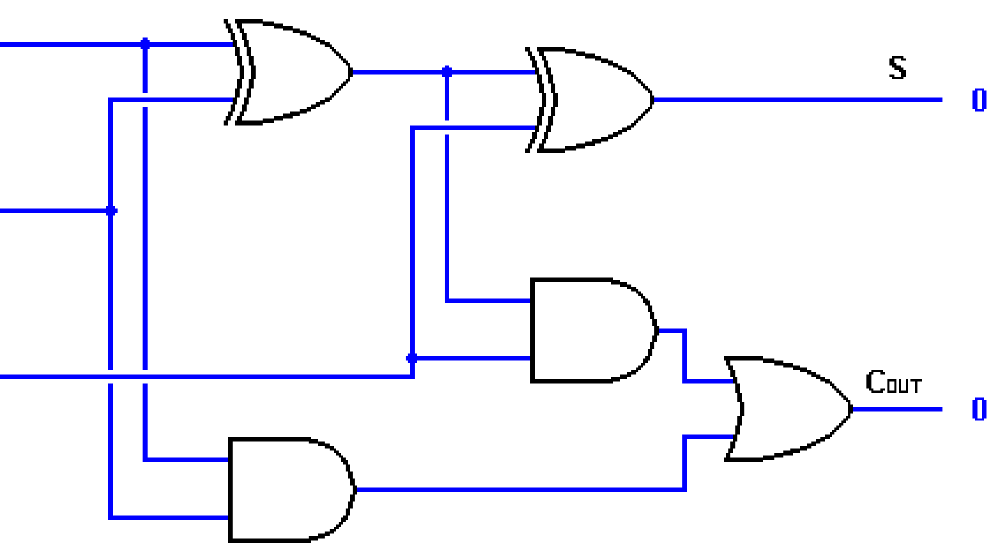
exclusive or (xor)

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

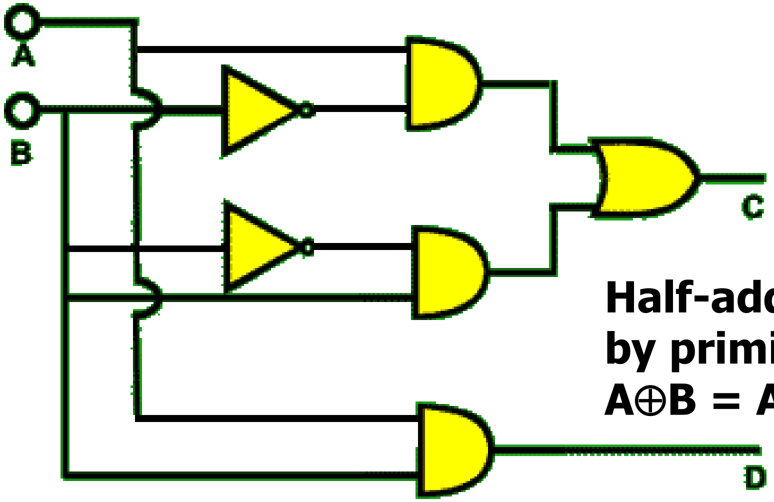
1-bit Adder Logic



Half-adder with one xor gate

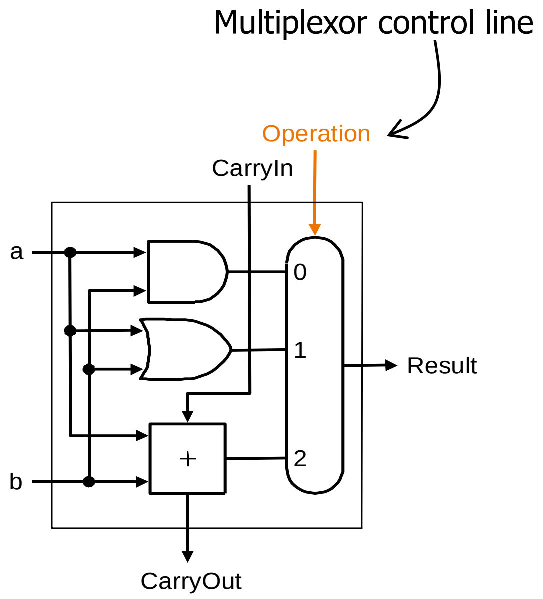


Full-adder from 2 half-adders and an or gate

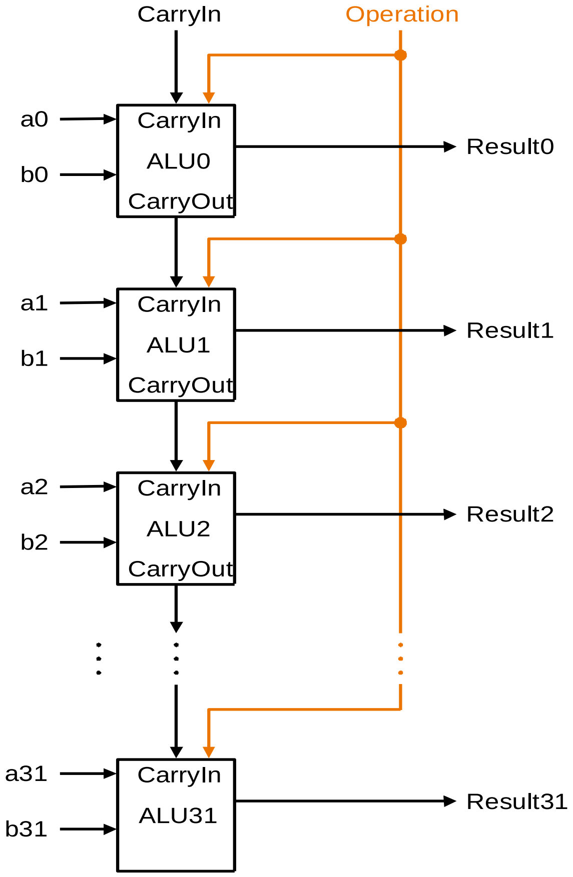


Half-adder with the xor gate replaced by primitive gates using the equation $A \oplus B = A.\overline{B} + \overline{A}.B$

Building a 32-bit ALU



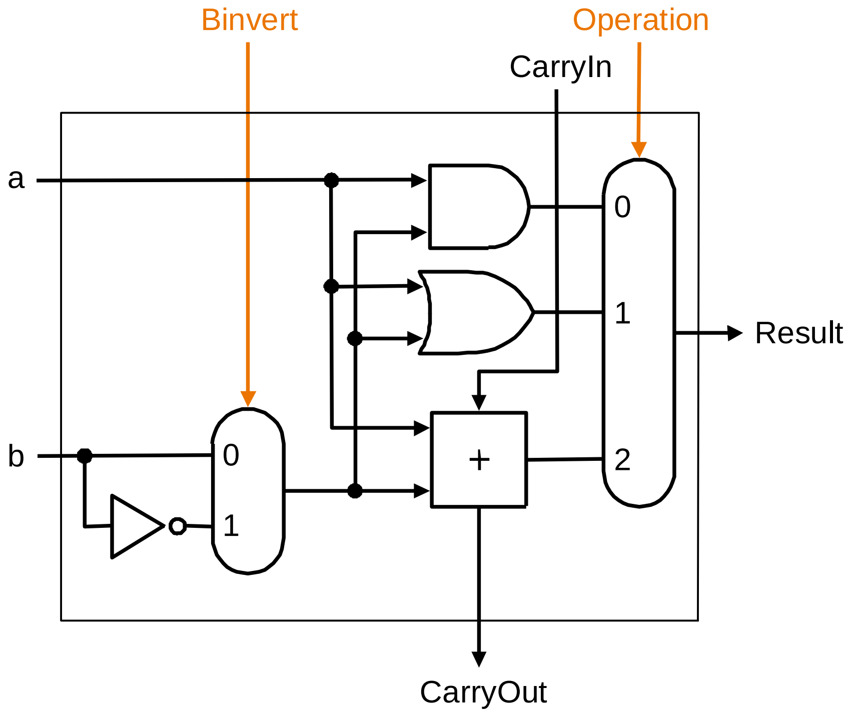
1-bit ALU for AND, OR and add



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

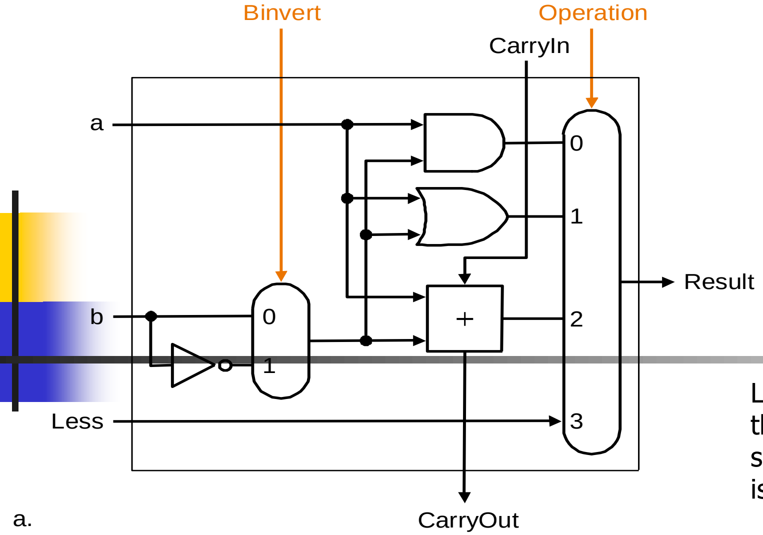
- Need to support the *set-on-less-than* instruction
  - e.g., `slt $t0, $t3, $t4`
  - remember: `slt` is an *R-type instruction* that produces 1 if rs < rt and 0 otherwise
  - idea is to use subtraction: rs < rt ⇔ rs – rt < 0. Recall msb of negative number is 1
  - two cases after subtraction rs – rt:
    - <u>if no overflow</u> then rs < rt ⇔ most significant bit of rs – rt = 1
    - <u>if overflow</u> then rs < rt ⇔ most significant bit of rs – rt = 0
  - why?
  - e.g., $5_{ten} – 6_{ten}$ = 0101 – 0110 = 0101 + 1010 = 1111 (ok!)

    $-7_{ten} – 6_{ten}$ = 1001 – 0110 = 1001 + 1010 = 0011 (overflow!)
  - therefore

    set bit = msb of rs – rt ⊕ overflow bit

    where *set bit,* which is output from ALU31, gives the result of `slt`
    - Fig. 4.17(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`

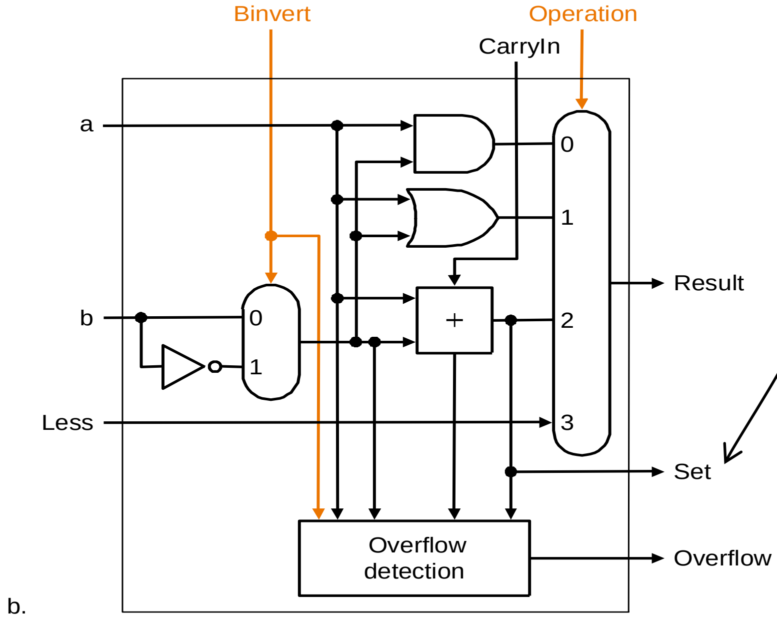
Supporting `slt`



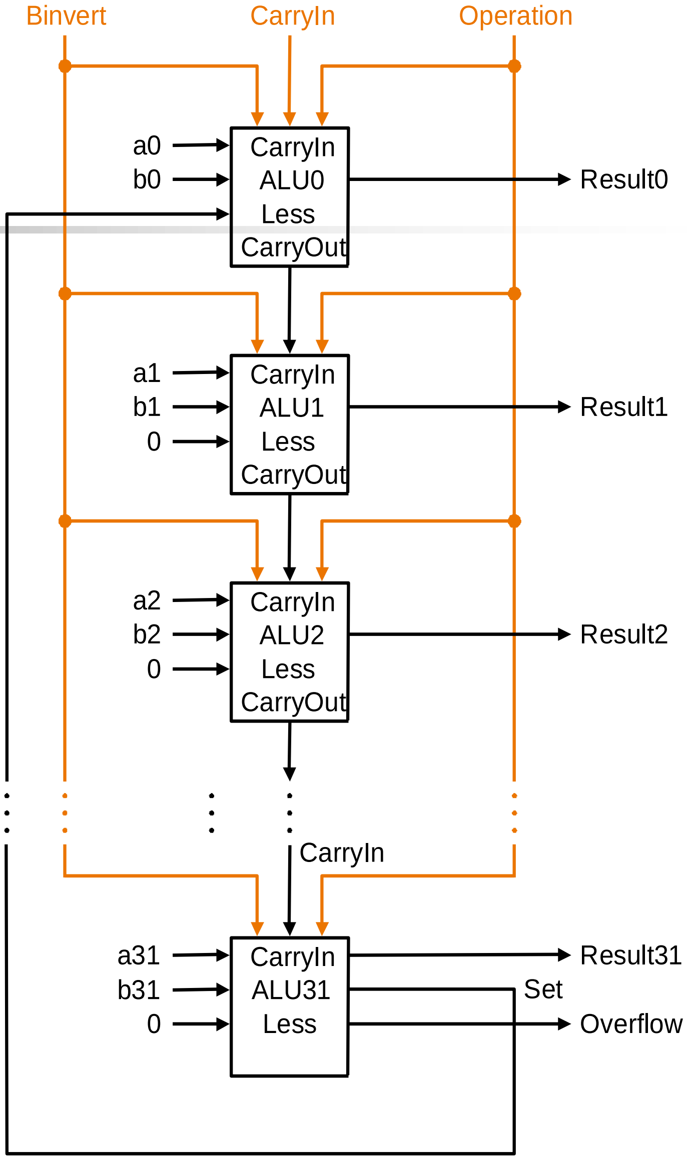
1- bit ALU for the 31 least significant bits

Less input of the 31 most significant ALUs is always 0

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)



1-bit ALU for the most significant bit

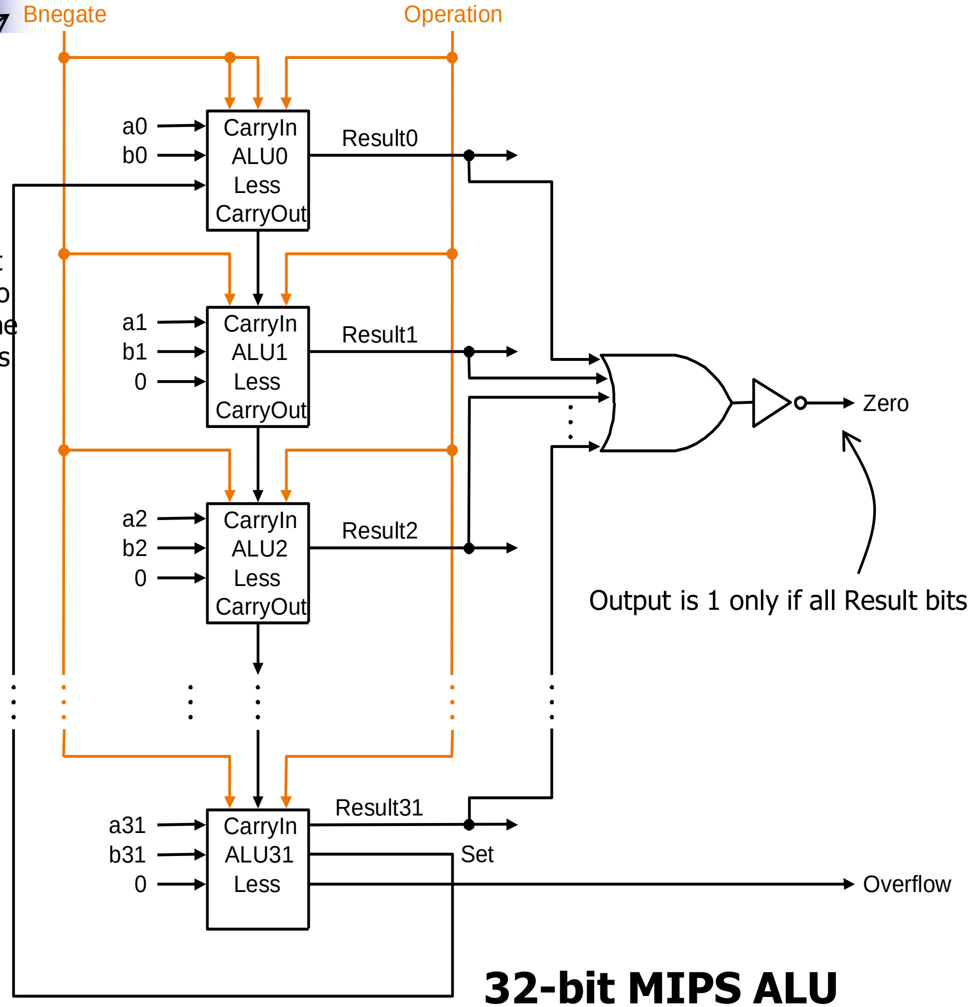


32-bit ALU from 31 copies of ALU at top left and 1 copy of ALU at bottom left in the most significant position

Tailoring the ALU to MIPS: Test for Less-than and Equality

- What about logic for the *overflow bit* ?
  - overflow bit = carry *in to* msb ⊕ 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 ⇔ rs = rt
  - *do we need to consider overflow?*

Supporting Test for Equality

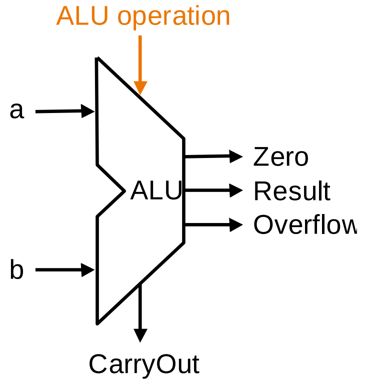


Combine CarryIn to least significant ALU and Binvert to a single control line as both are always either 1 or 0

Output is 1 only if all Result bits are 0

32-bit MIPS ALU

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



Symbol representing ALU

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 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:

$c_1 = b_0.c_0 + a_0.c_0 + a_0.b_0$

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

$c_2 = b_1.c_1 + a_1.c_1 + a_1.b_1$

$= a_1.a_0.b_0 + a_1.a_0.c_0 + a_1.b_0.c_0$ (substituting for c_1)

$+ b_1.a_0.b_0 + b_1.a_0.c_0 + b_1.b_0.c_0 + a_1.b_1$

$c_3 = b_2.c_2 + a_2.c_2 + a_2.b_2$

$= \ldots =$ sum of 15 4-term products…

- 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) $g_i = a_i \, . \, b_i$
  - when would we propagate the carry? (propagate) $p_i = a_i + b_i$
- Express (carry-in equations in terms of generate/propagates)

$$c_1 = g_0 + p_0 . c_0$$

$$c_2 = g_1 + p_1 . c_1 = g_1 + p_1 . g_0 + p_1 . p_0 . c_0$$

$$c_3 = g_2 + p_2 . c_2 = g_2 + p_2 . g_1 + p_2 . p_1 . g_0 + p_2 . p_1 . p_0 . c_0$$

$$c_4 = g_3 + p_3 . c_3 = g_3 + p_3 . g_2 + p_3 . p_2 . g_1 + p_3 . p_2 . p_1 . g_0 + p_3 . p_2 . p_1 . p_0 . c_0$$

- 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):

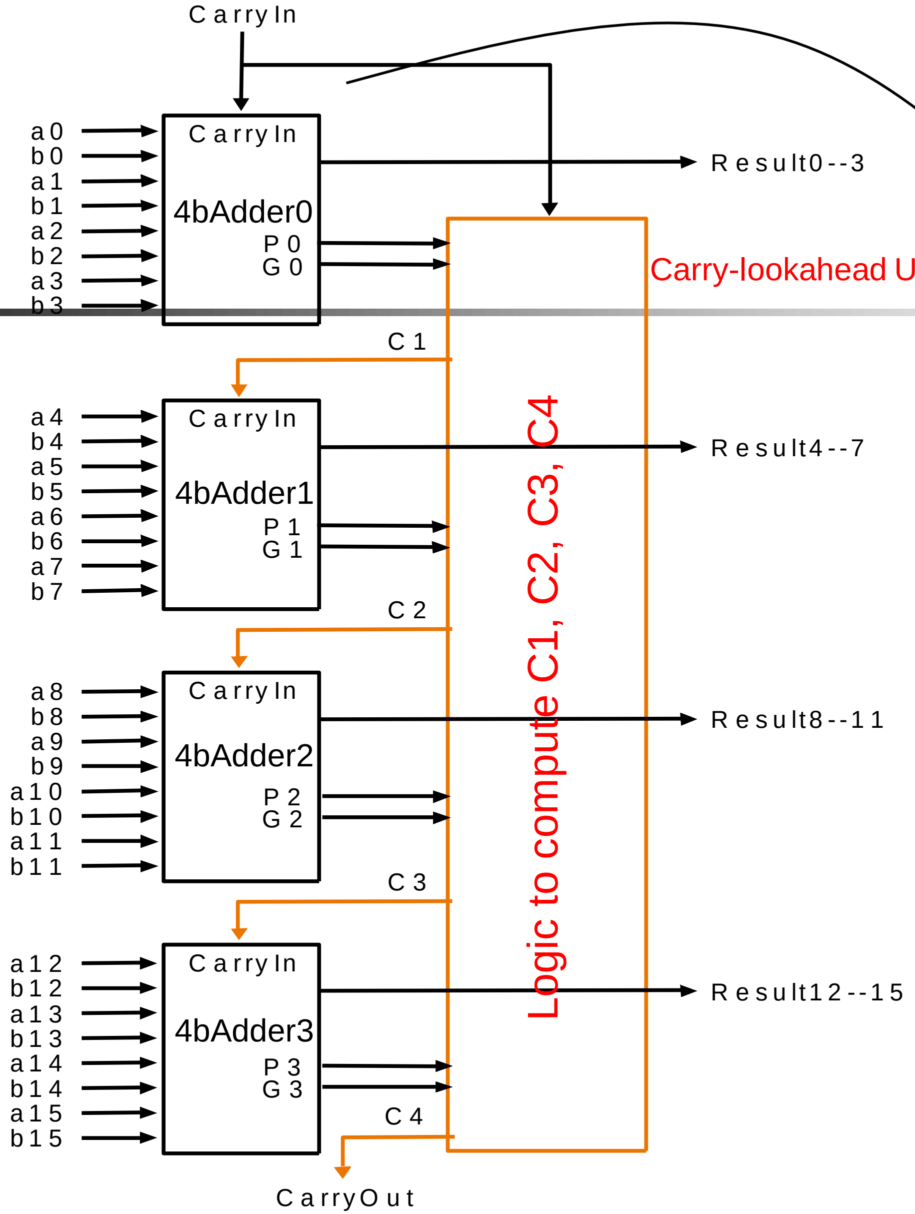
$C_1 = G_0 + P_0.c_0$

$C_2 = G_1 + P_1.G_0 + P_1.P_0.c_0$

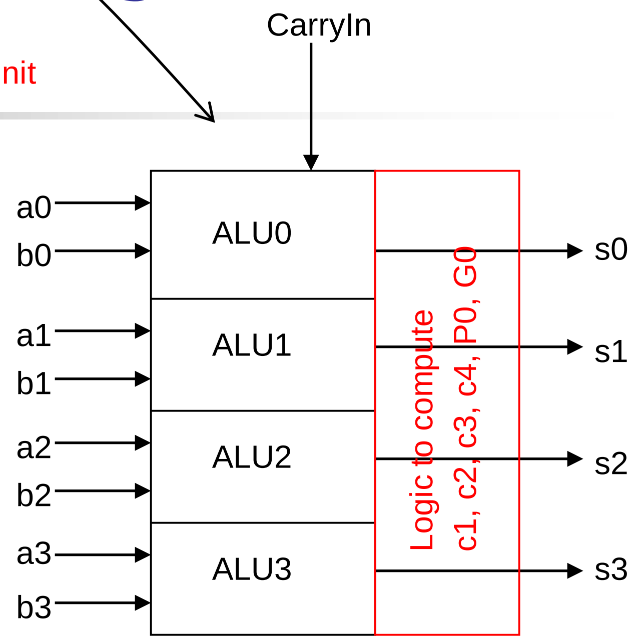
$C_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0$

$C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 +$
$P_3.P_2.P_1.P_0.c_0$

Carry-lookahead Logic



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



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

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

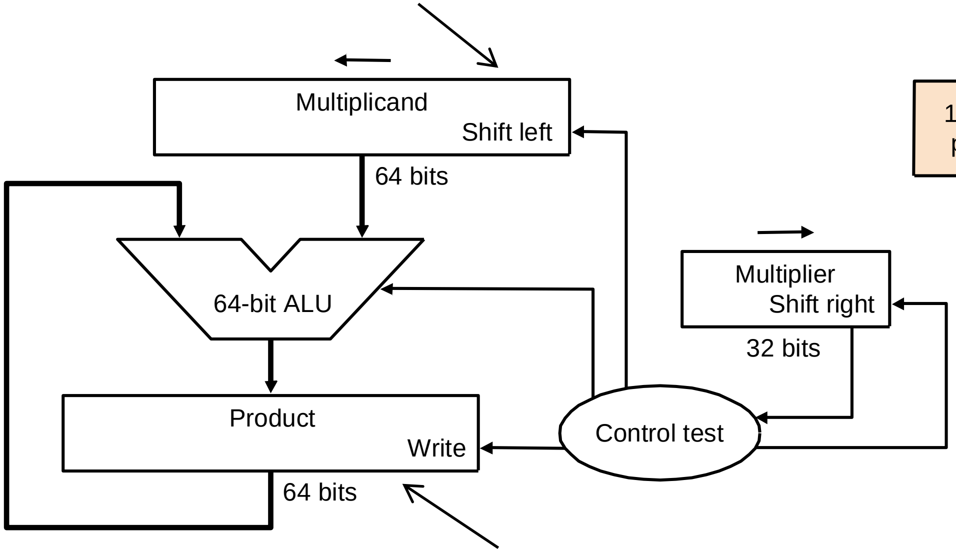
- Grade school shift-add method:

```
Multiplicand        1000
Multiplier        x 1001
                  ------
                    1000
                   0000
                  0000
                 1000
               ---------
Product        01001000
```

- m bits x 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

32-bit multiplicand starts at right half of multiplicand register



Multiplicand

Shift left

64 bits

64-bit ALU

Multiplier

Shift right

32 bits

Product

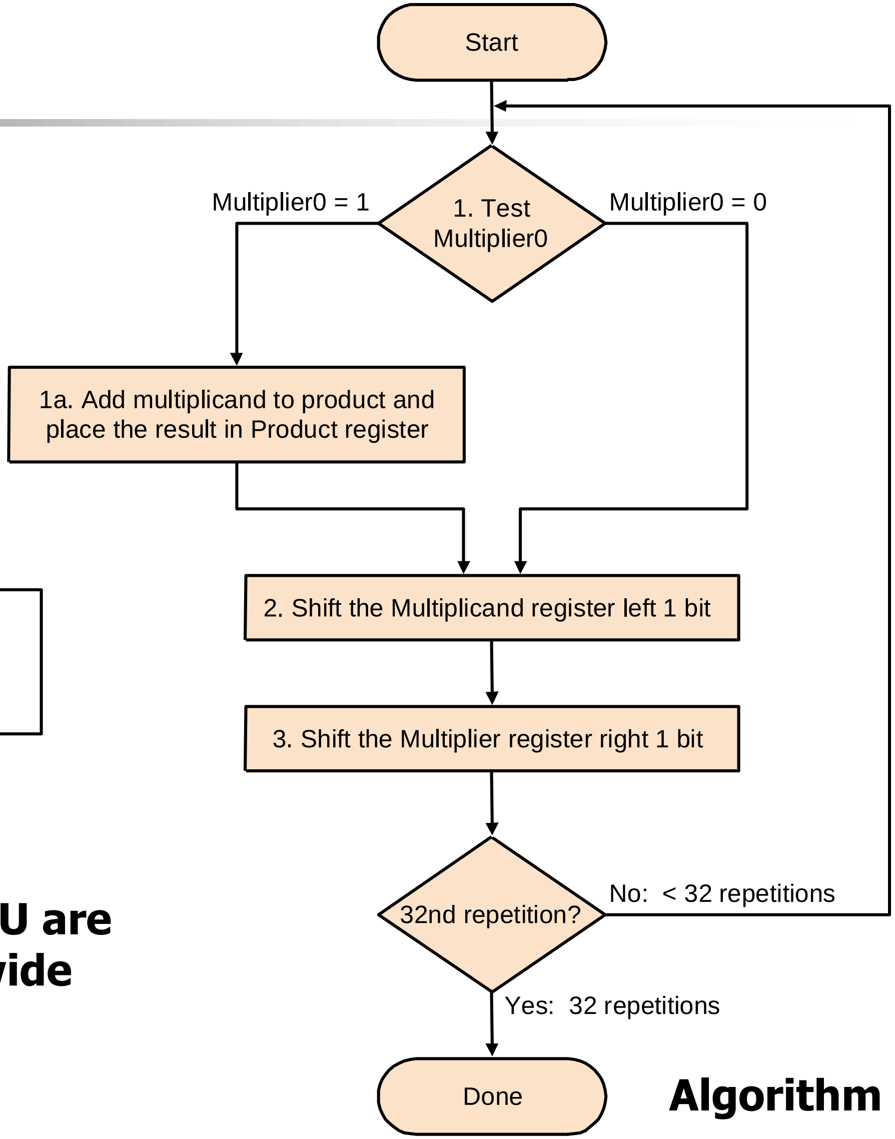
Write

64 bits

Control test

Product register is initialized at 0

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



Start

1. Test Multiplier0

Multiplier0 = 1

Multiplier0 = 0

1a. Add multiplicand to product and place the result in Product register

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?

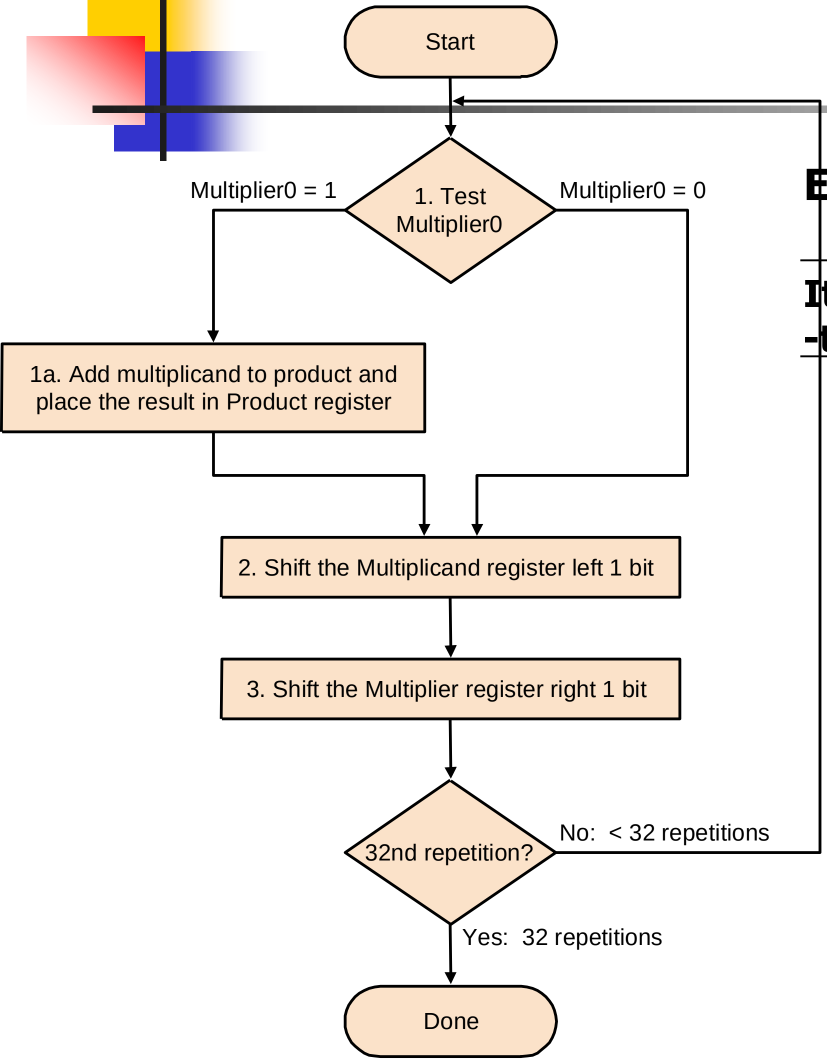
No: < 32 repetitions

Yes: 32 repetitions

Done

Algorithm

Shift-add Multiplier Version1



Start

1. Test Multiplier0

Multiplier0 = 1

Multiplier0 = 0

1a. Add multiplicand to product and place the result in Product register

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

Algorithm

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	…			

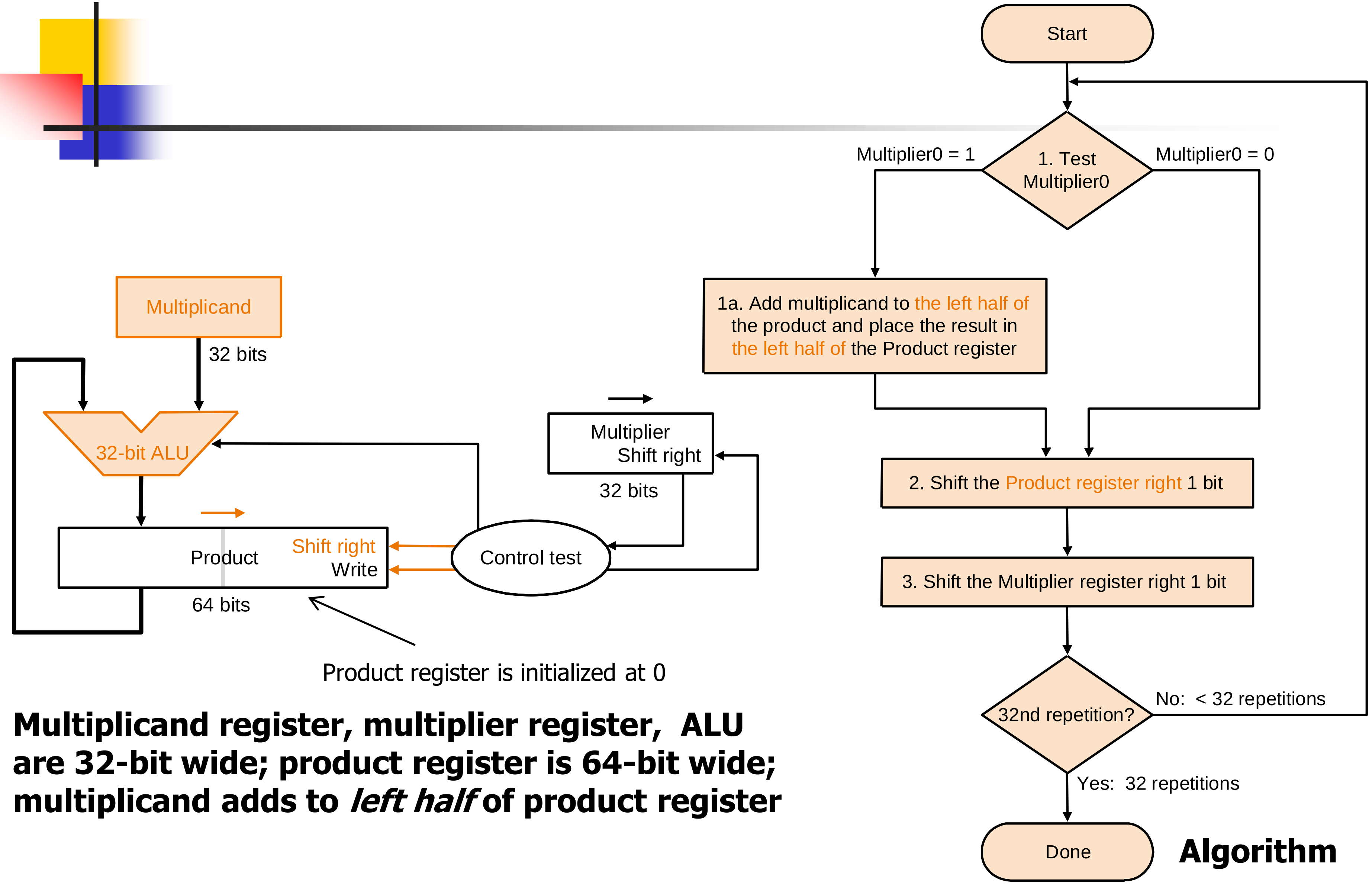
Observations on Multiply Version 1

- 1 step per clock cycle ⇒ nearly 100 clock cycles to multiply two 32-bit numbers
- Half the bits in the multiplicand register always 0

  ⇒ 64-bit adder is wasted
- 0's inserted to right as multiplicand is shifted left

  ⇒ 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

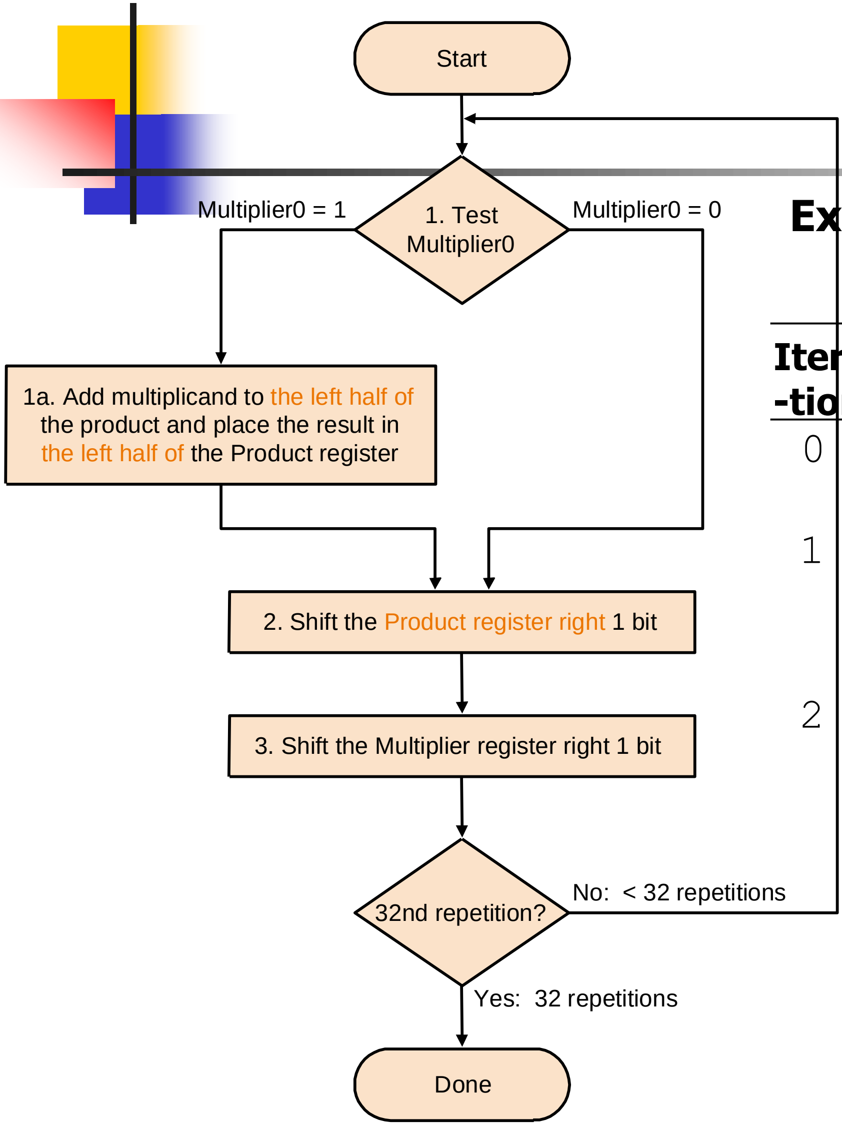


Product register is initialized at 0

Multiplicand register, multiplier register, ALU are 32-bit wide; product register is 64-bit wide; multiplicand adds to *left half* of product register

Algorithm

Shift-add Multiplier Version 2


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Algorithm

Example: 0010 * 0011:

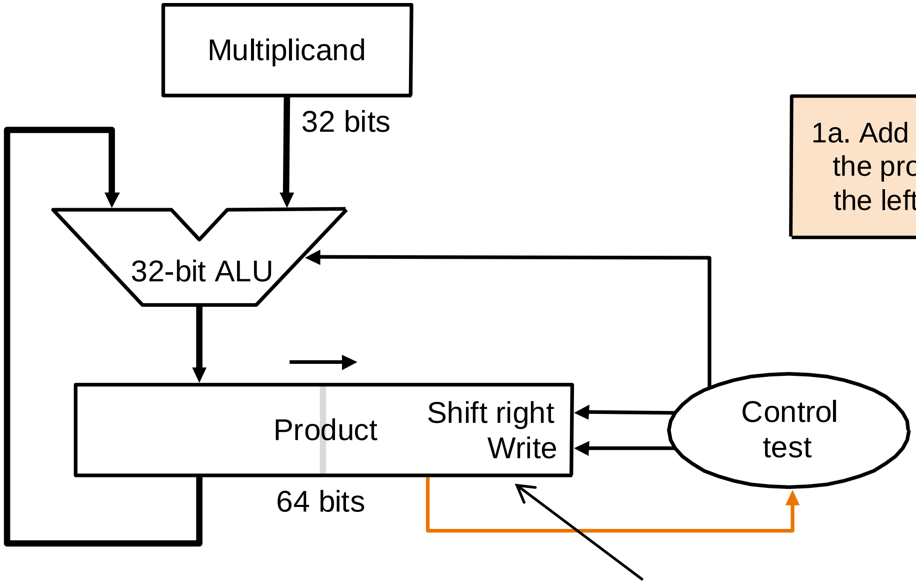
Itera -tion	Step	Multiplier	Multiplicand	Product
0	init values	0011	0010	0000 0000
1	1a	0011	0010	0010 0000
	2	0011	0010	0001 0000
	3	0001	0010	0001 0000
2	…			

Observations on Multiply Version 2

- Each step the product register wastes space that exactly matches the current size of the multiplier

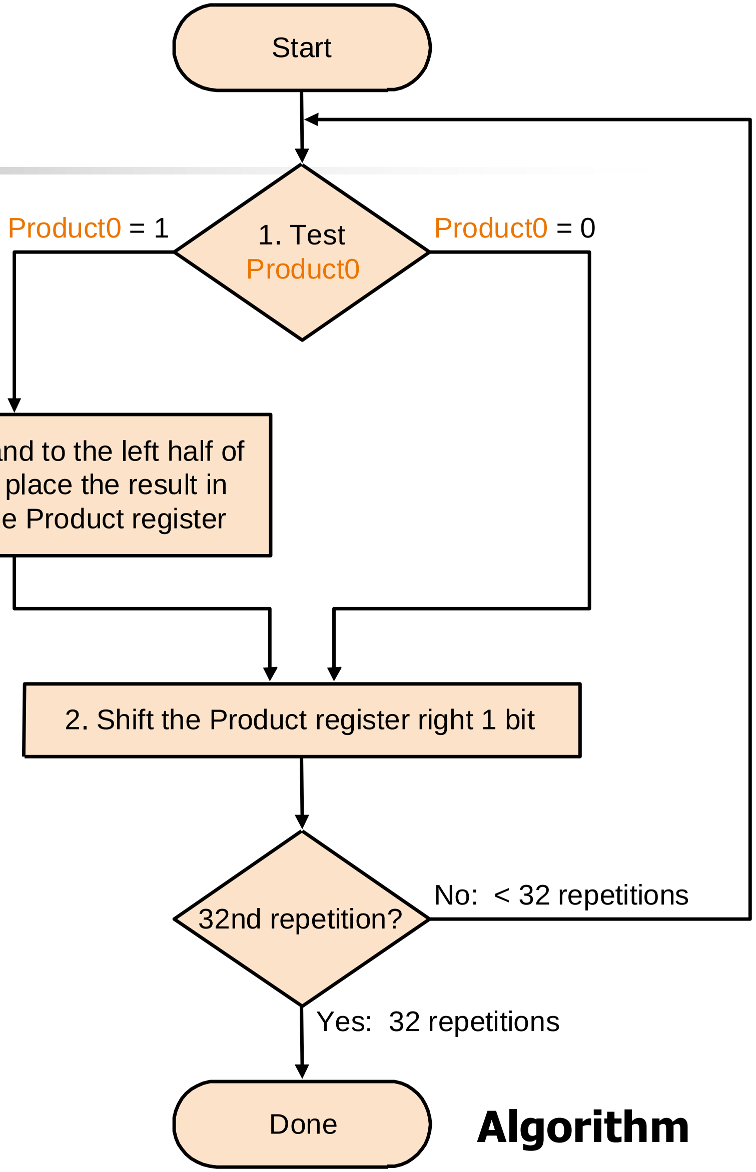
- Intuition: combine multiplier register and product register…

Shift-add Multiplier Version 3



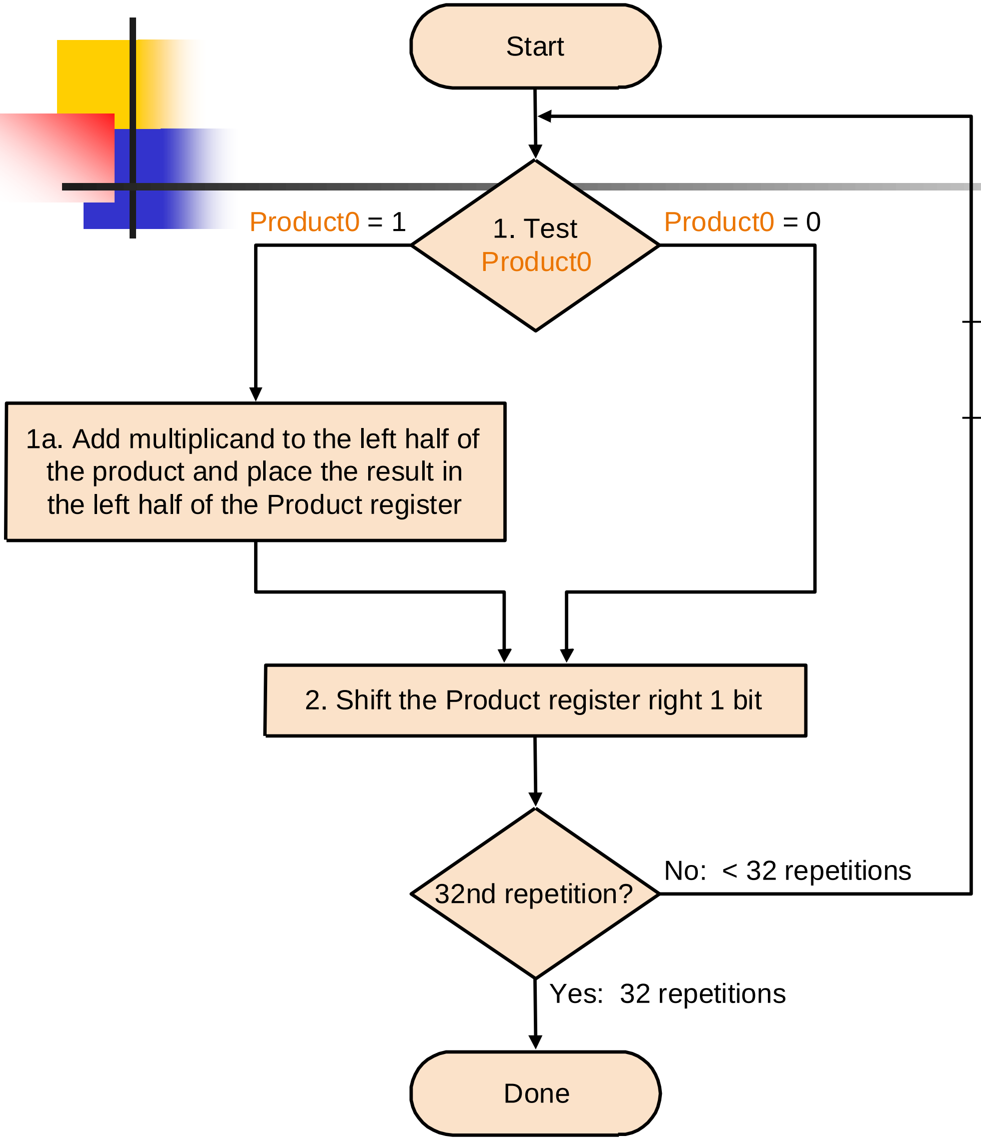
Product register is initialized with multiplier on right

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


Start
1. Test
Product0
Product0 = 1
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Algorithm

Shift-add Multiplier Version 3



Start

1. Test Product0

Product0 = 1

Product0 = 0

1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register

2. Shift the Product register right 1 bit

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

Algorithm

Example: 0010 * 0011:

Itera-tion	Step	Multiplicand	Product
0	init values	0010	0000 0011
1	1a	0010	0010 0011
	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

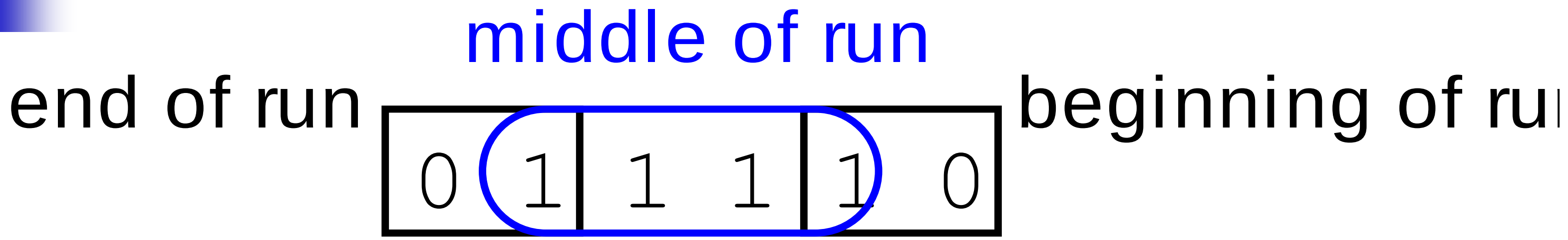
- Example 0010 * 0110. Traditional:

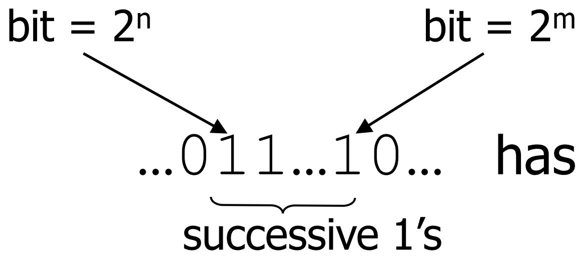
```
          0010
       x  0110
          ----
          0000   shift (0 in multiplier)
         0010    add (1 in multiplier)
        0010     add (1 in multiplier)
       0000      shift (0 in multiplier)
      --------
      00001100
```

- 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:

```
          0010
          0110
          ----
          0000   shift (0 in multiplier)
        -0010    sub  (first 1 in multiplier)
        0000     shift (middle of string of 1's)
       0010      add (previous step had last 1)
      --------
      00001100
```

Motivating Booth’s Algorithm





- Math idea: string of 1’s …011…10… has

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

- 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:

Case	Current Bit	Bit to the Right	Explanation	Example	Op
1a	0	0	Middle of run of 0s	0001111000	none
1b	0	1	End of run of 1s	0001111000	add
1c	1	0	Begins run of 1s	0001111000	sub
1d	1	1	Middle of run of 1s	0001111000	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 = $a_{31}\ a_{32}\ldots\ a_0$, multiplicand = b
-

a_i	a_{i-1}	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:

$$\begin{aligned} & (a_{-1} - a_0) \quad * b * 2^0 \\ + & (a_0 - a_{\ 1}) \quad * b * 2^1 \\ + & (a_1 - a_2) \quad * b * 2^2 \\ & \quad \ldots \\ + & (a_{30} - a_{31}) \quad * b * 2^{31} \end{aligned}$$

= … *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

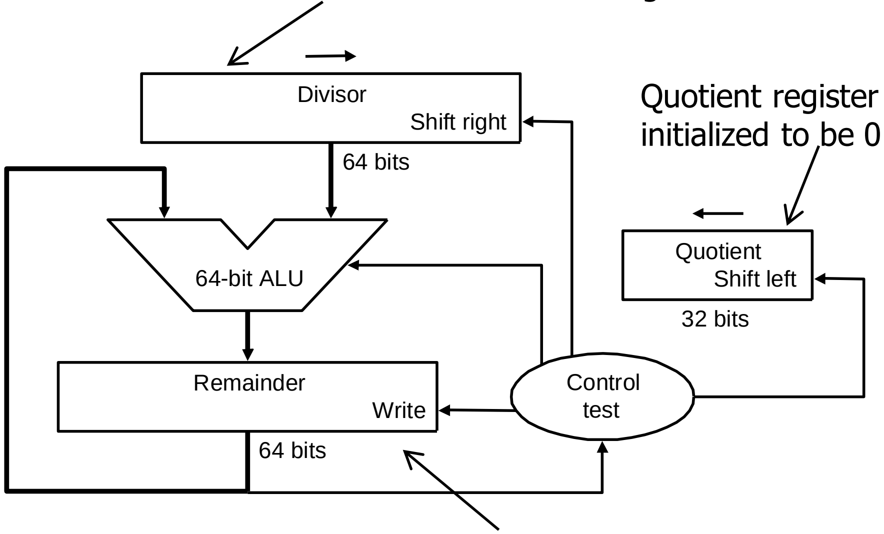
Divide

```
                  1001  Quotient
              _________
Divisor 1000  |1001010  Dividend
              -1000
               ----
                  10
                  101
                  1010
                 -1000
                  ----
                    10  Remainder
```

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

Divide Version 1

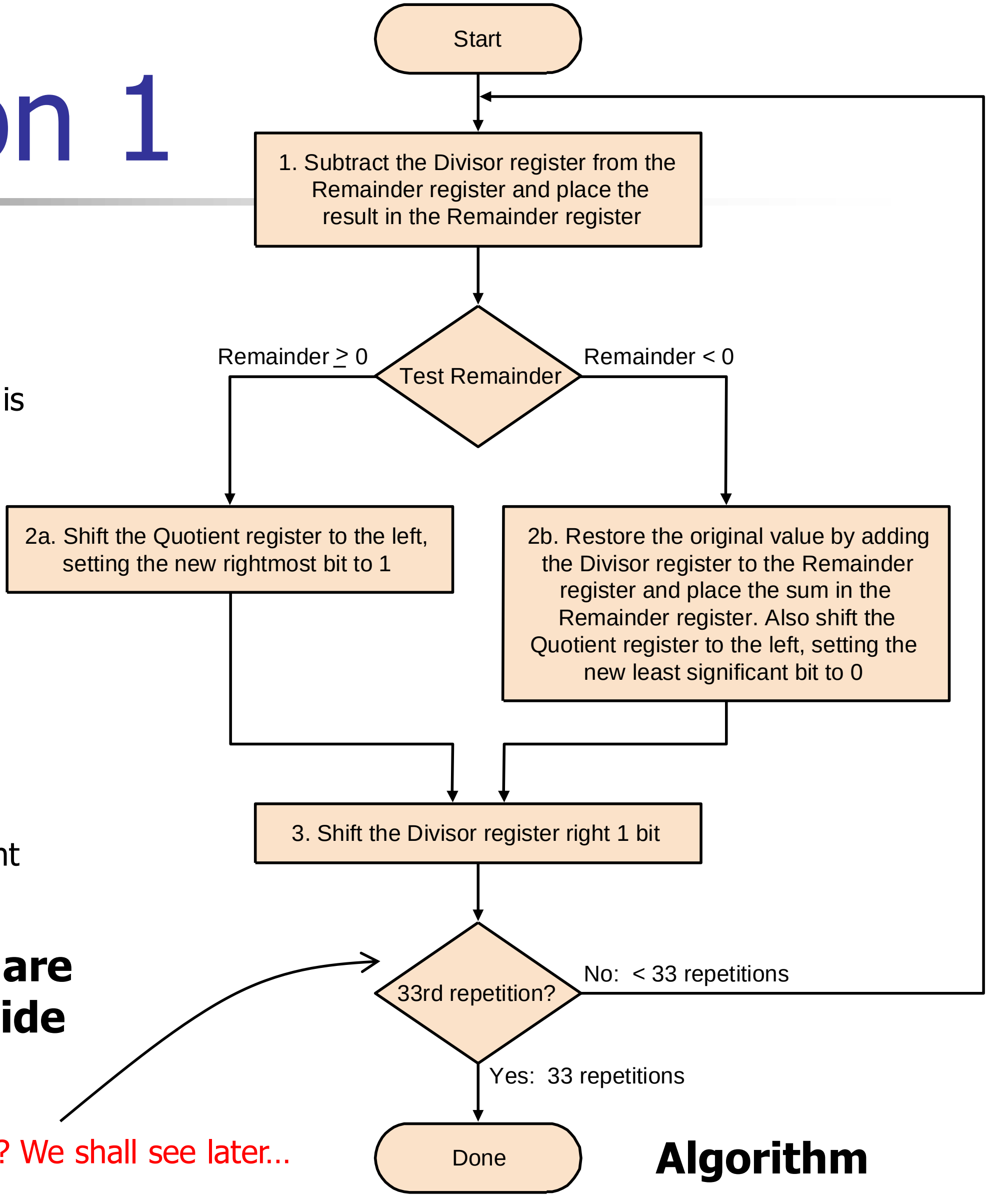
32-bit divisor starts at left half of divisor register



Quotient register is initialized to be 0

Remainder register is initialized with the dividend at right

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



Start

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

Test Remainder

Remainder ≥ 0

Remainder < 0

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

2b. Restore the original value by adding the Divisor register to the Remainder register and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

33rd repetition?

No: < 33 repetitions

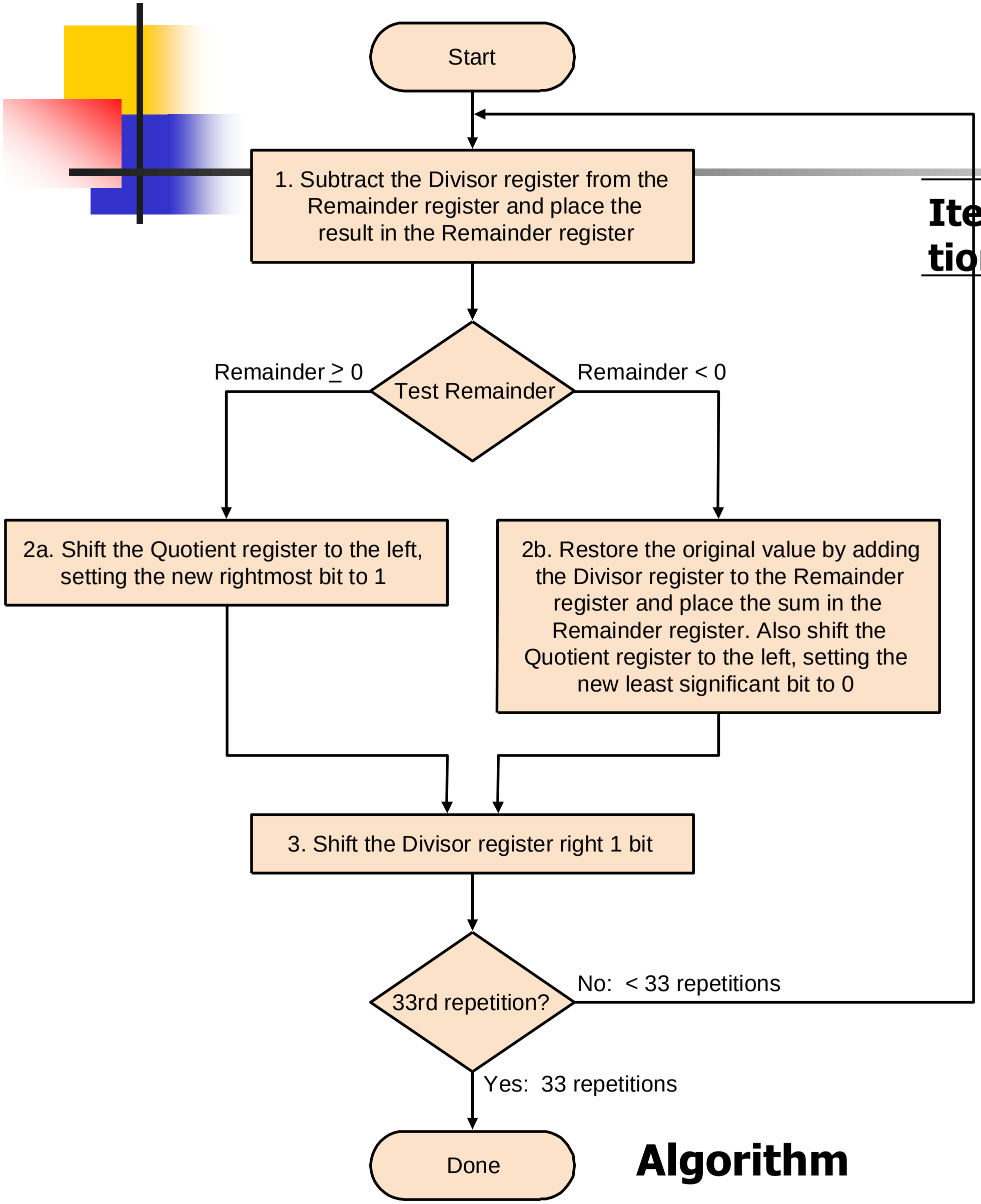
Yes: 33 repetitions

Done

Why 33? We shall see later…

Algorithm

Divide Version 1



Start

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

Test Remainder

Remainder ≥ 0

Remainder < 0

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

2b. Restore the original value by adding the Divisor register to the Remainder register and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

33rd repetition?

No: < 33 repetitions

Yes: 33 repetitions

Done

Algorithm

Example: 0111 / 0010:

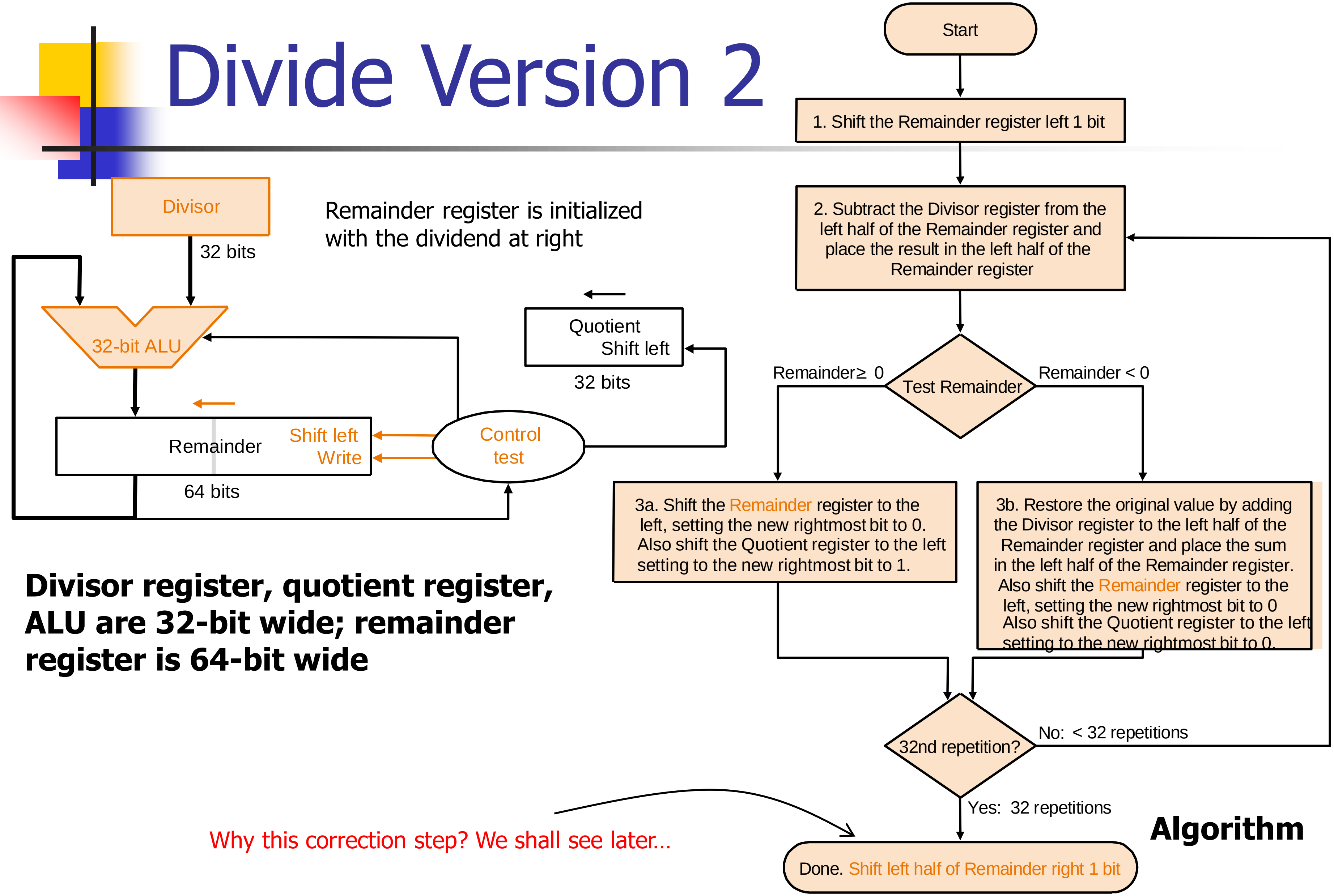
Itera-tion	Step	Quotient	Divisor	Remainder
0	init	0000	0010 0000	0000 0111
1	1	0000	0010 0000	1110 0111
	2b	0000	0010 0000	0000 0111
	3	0000	0001 0000	0000 0111
2	…			
3				
4				
5				

Observations on Divide Version 1

- Half the bits in divisor always 0
  - ⇒ 1/2 of 64-bit adder is wasted
  - ⇒ 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



Remainder register is initialized with the dividend at right

Start

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

Test Remainder

Remainder≥ 0

3a. Shift the Remainder register to the left, setting the new rightmost bit to 0. Also shift the Quotient register to the left setting to the new rightmost bit to 1.

Remainder < 0

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0 Also shift the Quotient register to the left setting to the new rightmost bit to 0.

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done. Shift left half of Remainder right 1 bit

Algorithm

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

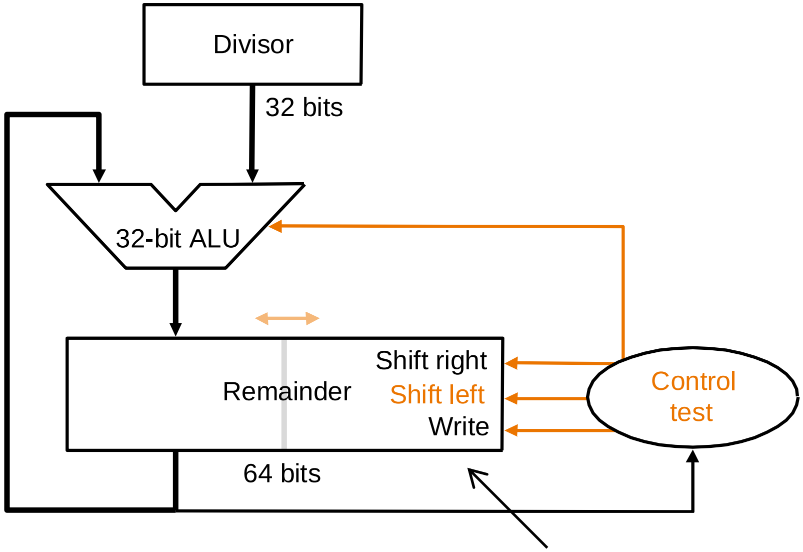
Why this correction step? We shall see later…

Observations on Divide Version 2

- Each step the remainder register wastes space that exactly matches the current size of the quotient

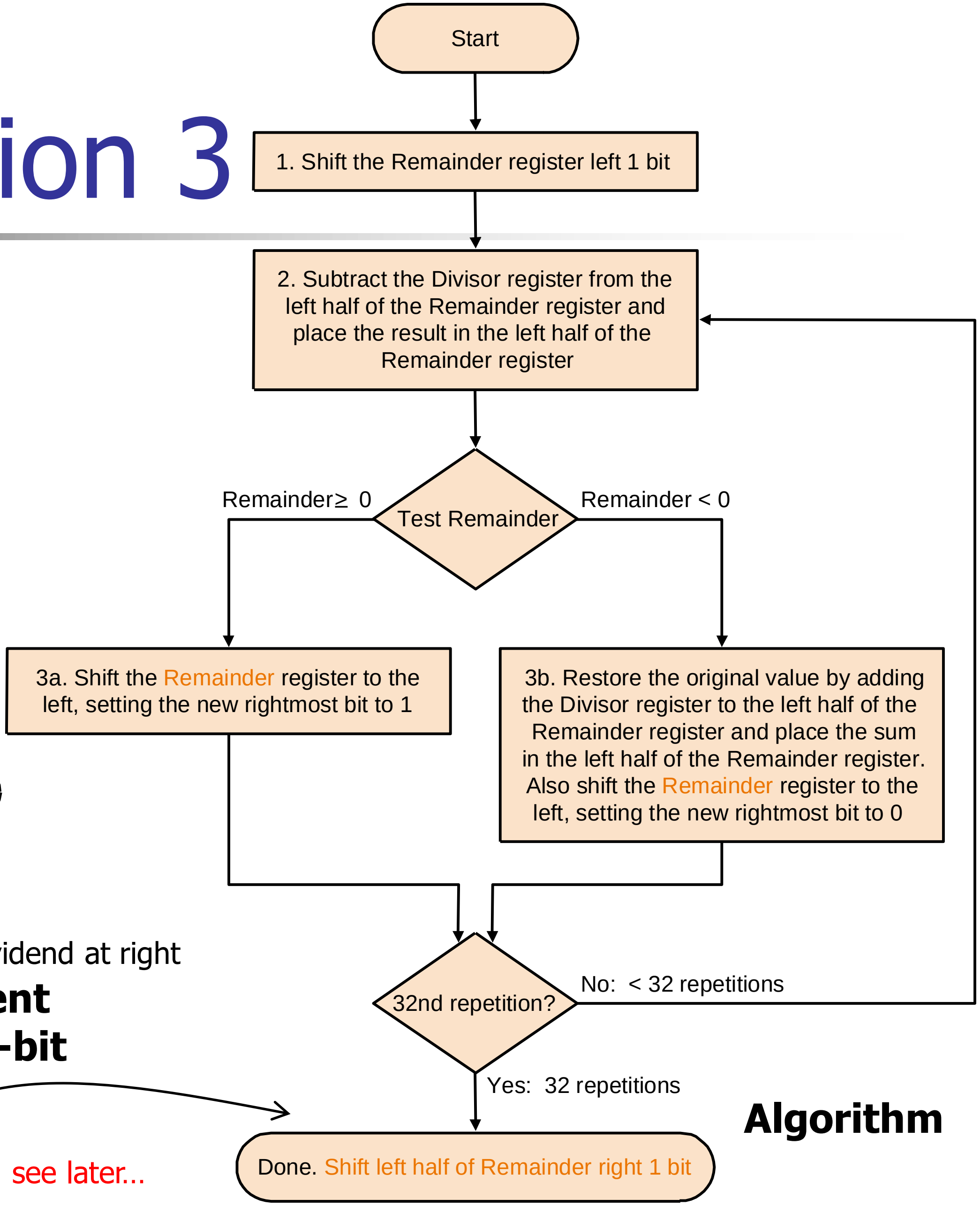
- Intuition: combine quotient register and remainder register…

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



1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

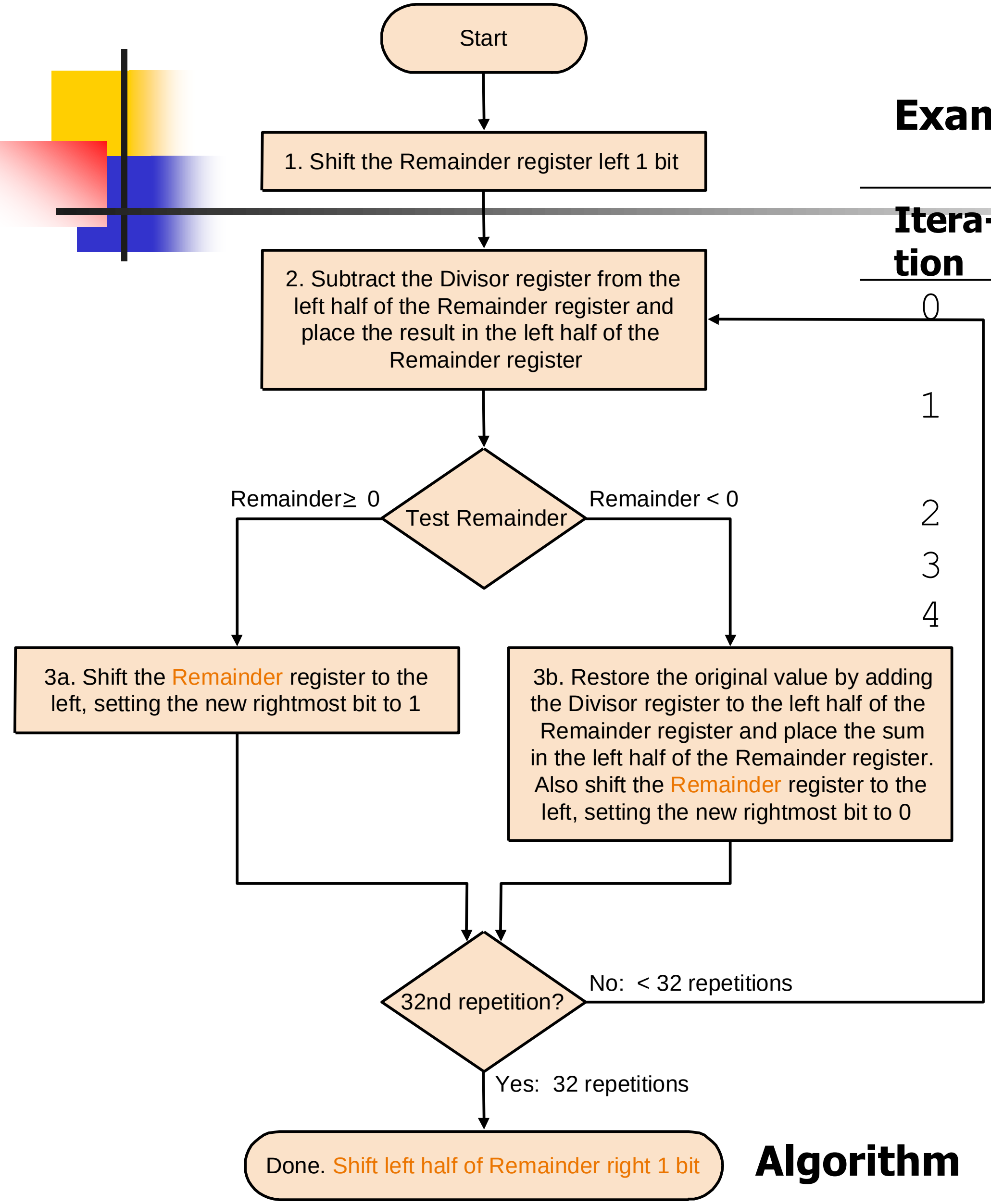
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

Done. Shift left half of Remainder right 1 bit

Algorithm

Why this correction step? We shall see later…

Divide Version 3



Start

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

Test Remainder

Remainder≥ 0

Remainder < 0

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done. Shift left half of Remainder right 1 bit

Algorithm

Example: 0111 / 0010:

Itera-tion	Step	Divisor	Remainder
0	init	0010	0000 0111
	1	0010	0000 1110
1	2	0010	1110 1110
	3b	0010	0001 1100
2	…		
3			
4			

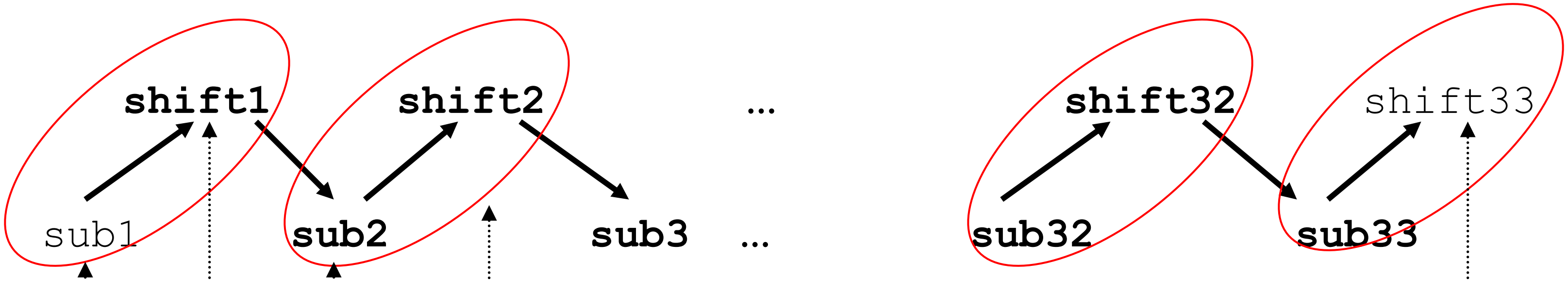
Number of Iterations

- 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 – sub(i+1) must actually follow shifti of the divisor (or remainder, depending on version) and the resulting bit in the quotient appears on shift(i+1)



V1 *starts loop* here: unnecessary sub step

V2 & 3 initial step: *before* loop starts

V2 & 3 *start loop* here

One loop iteration

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 any way; 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
    - dividend = (quotient * divisor) + remainder
    - –quotient (x/y) = 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

    - $(-1)^{\text{sign}} *$ significand $* \ 2^{\text{exponent}}$. E.g., $-101.001101 * 2^{111001}$
  - more bits for *significand* gives more accuracy
  - more bits for *exponent* increases range
  - if $1 \leq$ 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:
    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:
    value = $(-1)^{\text{sign}} * (1 + \text{significand}) * 2^{\overbrace{(\text{exponent} - \text{bias})}^{\text{equals exponent value}}}$

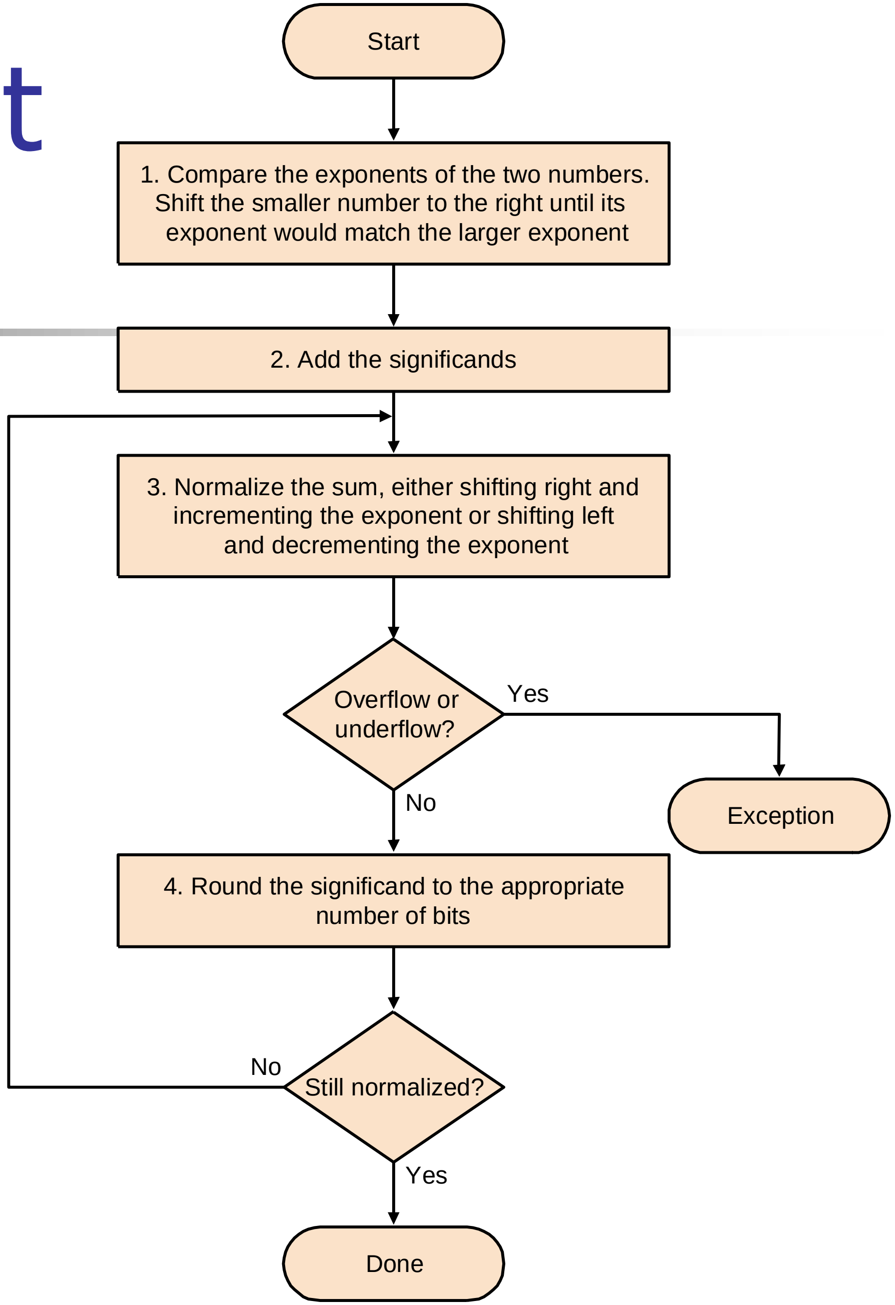
IEEE 754 Floating-point Standard

- 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: $-11/100 = -.11 = -1.1 \text{ x } 2^{-1}$
  - IEEE single precision floating point exponent = bias + exponent value $= 127 + (-1) = 126_{\text{ten}} = 01111110_{\text{two}}$
  - IEEE single precision: $\underset{\nearrow\,\text{sign}}{1}\underbrace{01111110}_{\text{exponent}}\underbrace{10000000000000000000000}_{\text{significand}}$

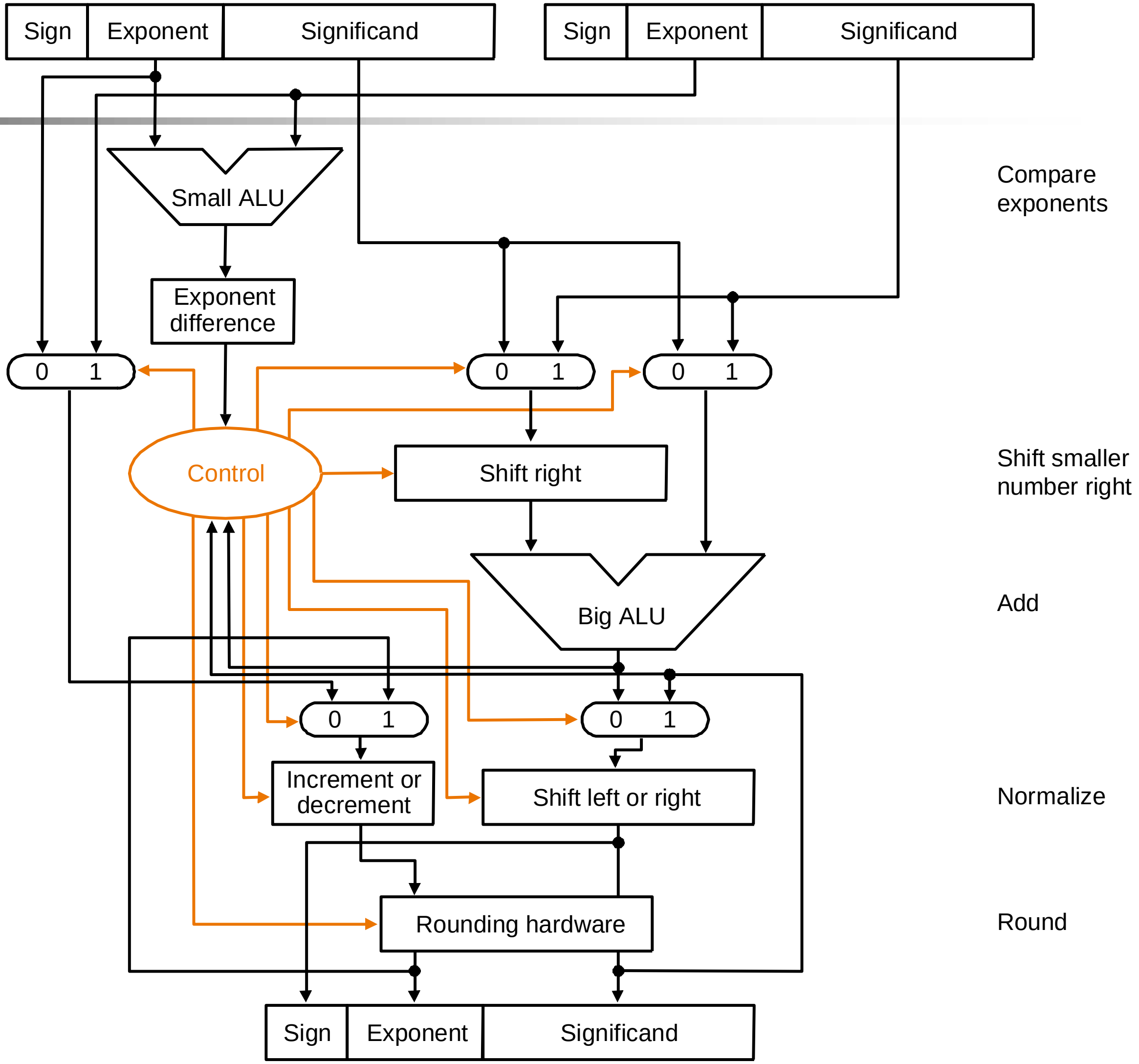
Floating Point Addition

- Algorithm:


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

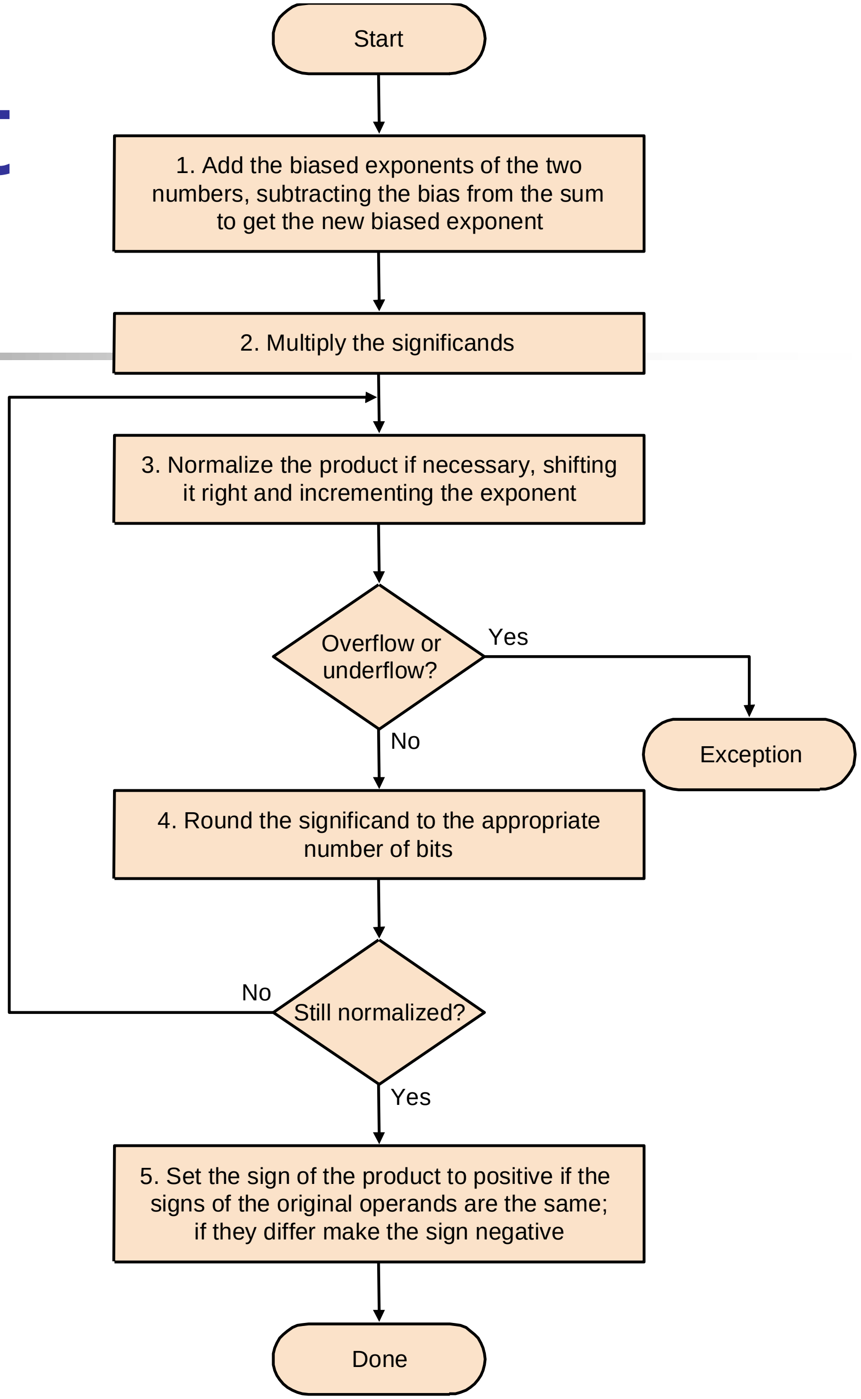
Floating Point Addition

- Hardware:



Floating Point Multpication

- Algorithm:


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

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 ⇒ rounding ⇒ 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
  - there will be assignment and projects based on Koren's material