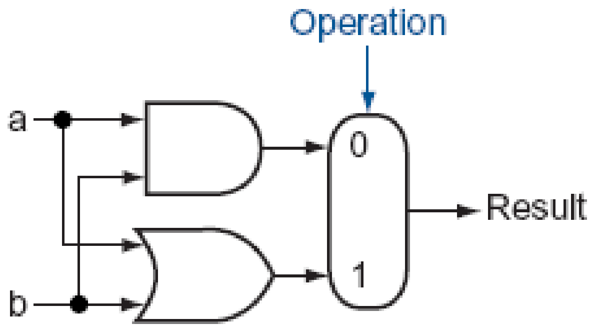
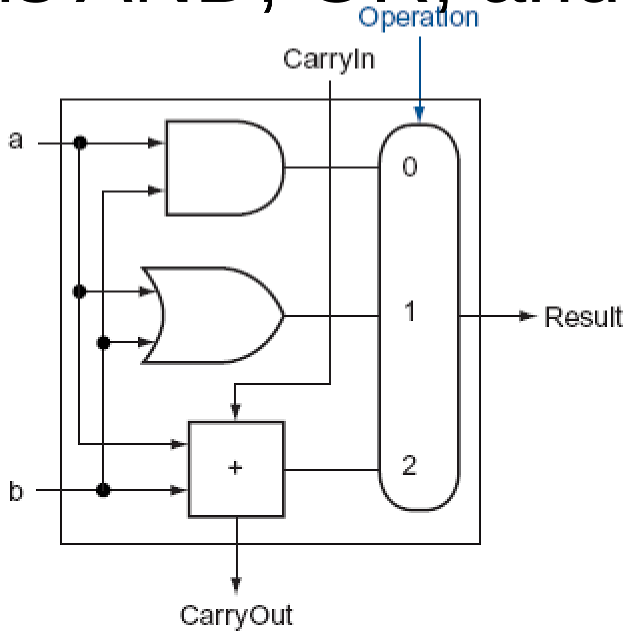
Chapter 3

Arithmetic for Computers

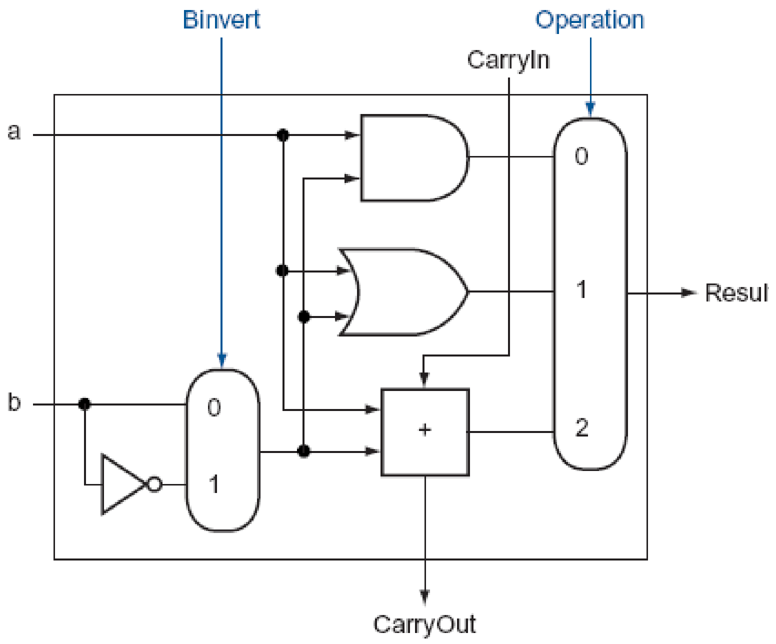
Basic Arithmetic Logic Unit

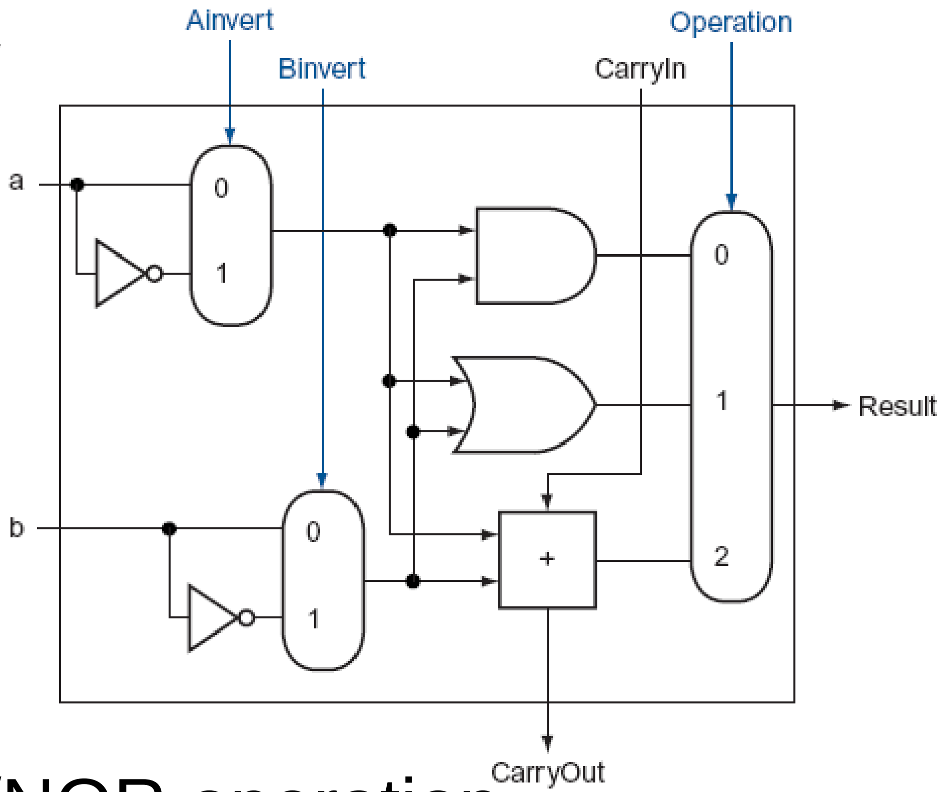
- One-bit ALU that performs AND, OR, and addition





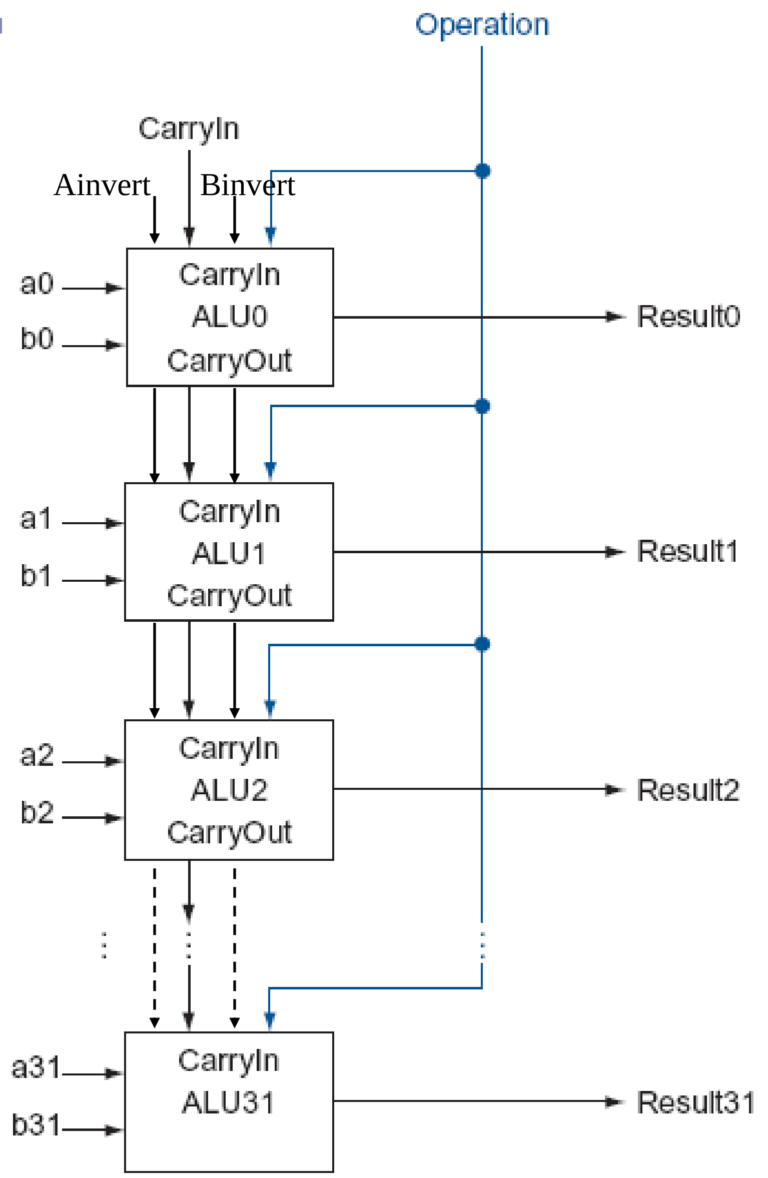
Enhanced Arithmetic Logic Unit





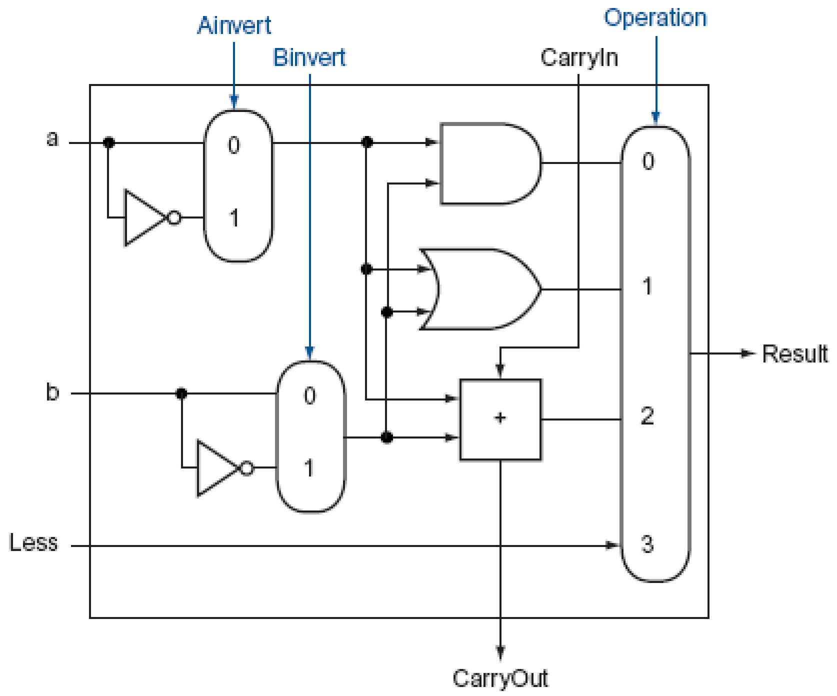
- ALU that have NAND/NOR operation

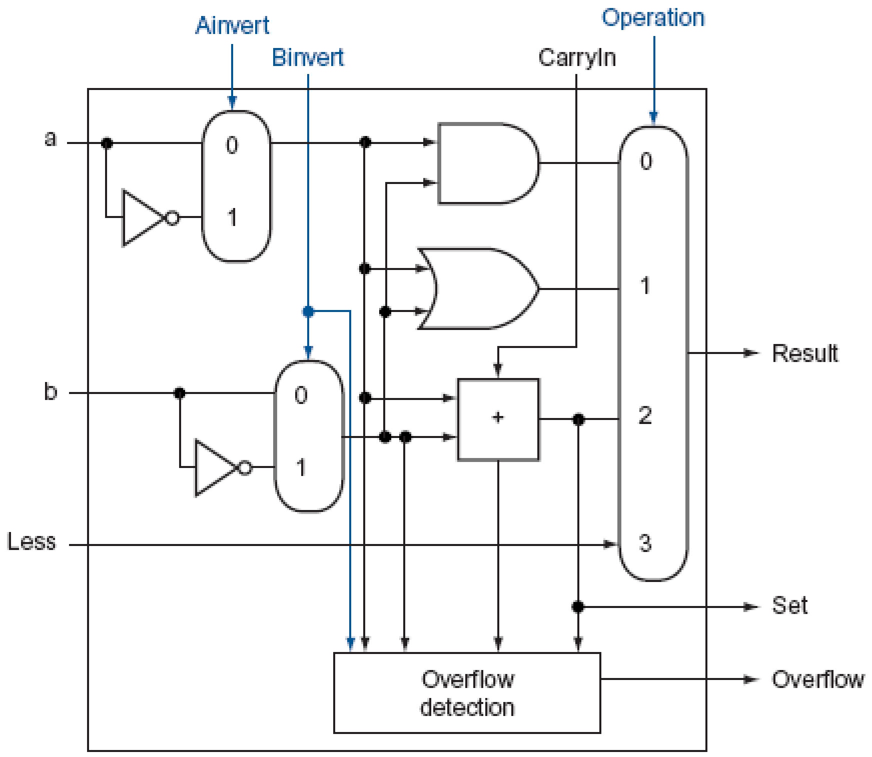
32-bit ALU


Operation
CarryIn
Ainvert
Binvert
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
a31
b31
CarryIn
ALU31
Result31

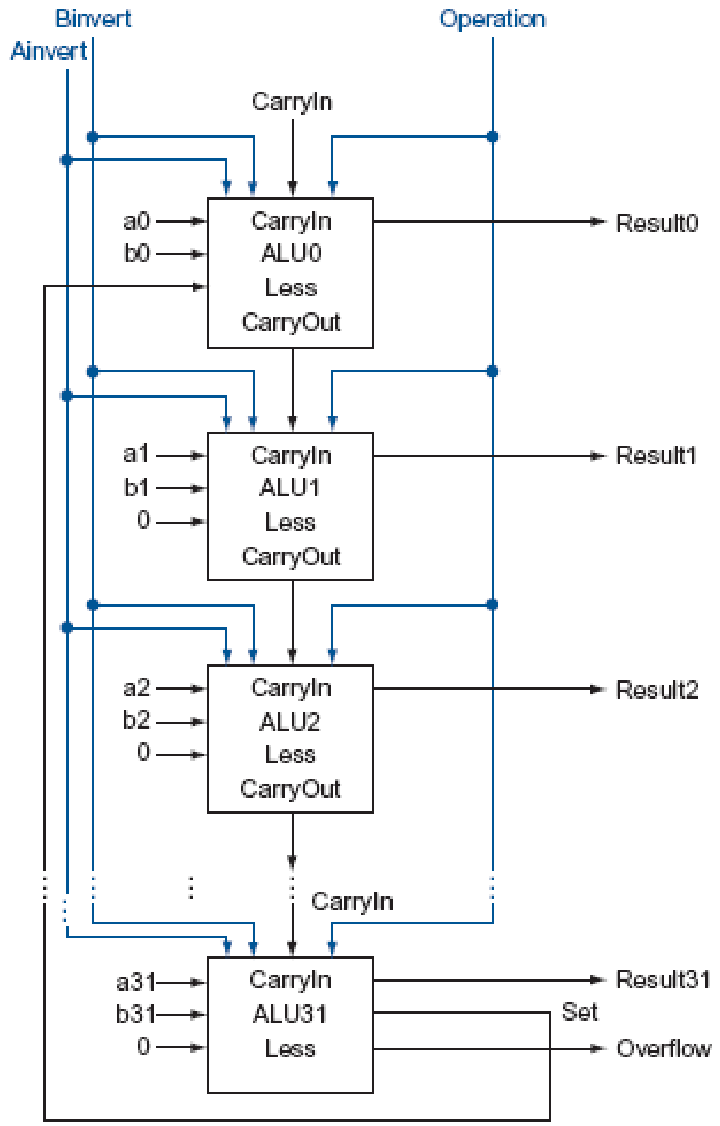
One-bit ALUs with Set Less Than

- Set less than instruction requires a subtraction and then sets all but the least significant bit to 0, with the lsb set to 1 if a < b
- Less signal line
  - lsb – signed bit
  - All but the lsb – 0

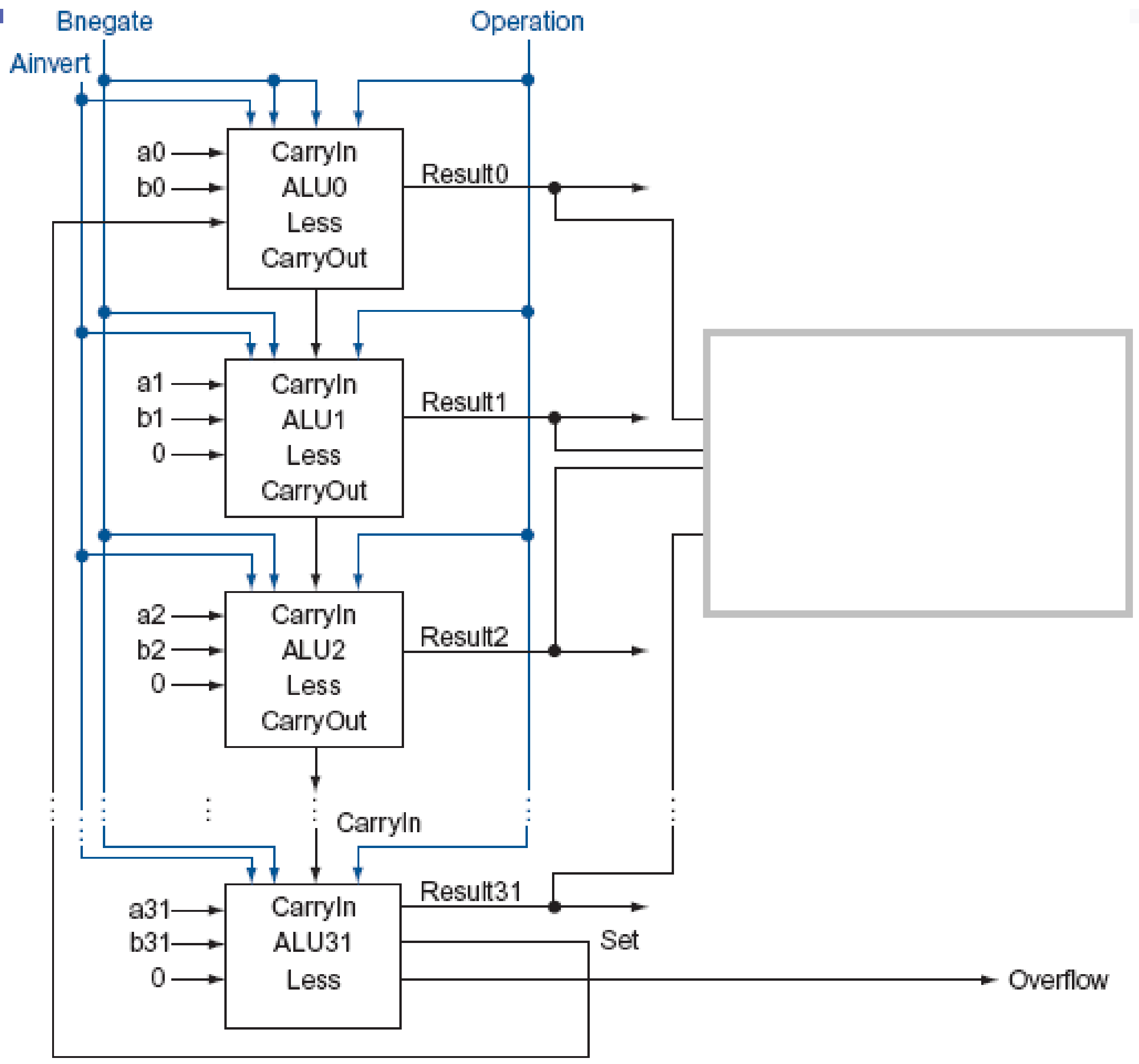




32-bit ALU with Set Less Than



Final 32-bit ALU

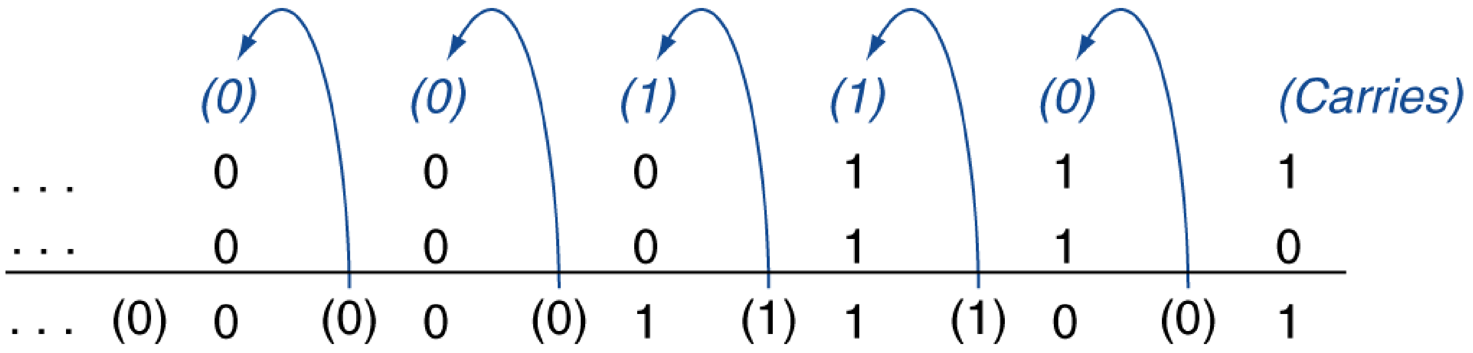


Arithmetic for Computers

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

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if
  - Adding two –ve operands
    - Overflow if

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
  - Subtracting –ve from +ve operand
    - Overflow if

Dealing with Overflow

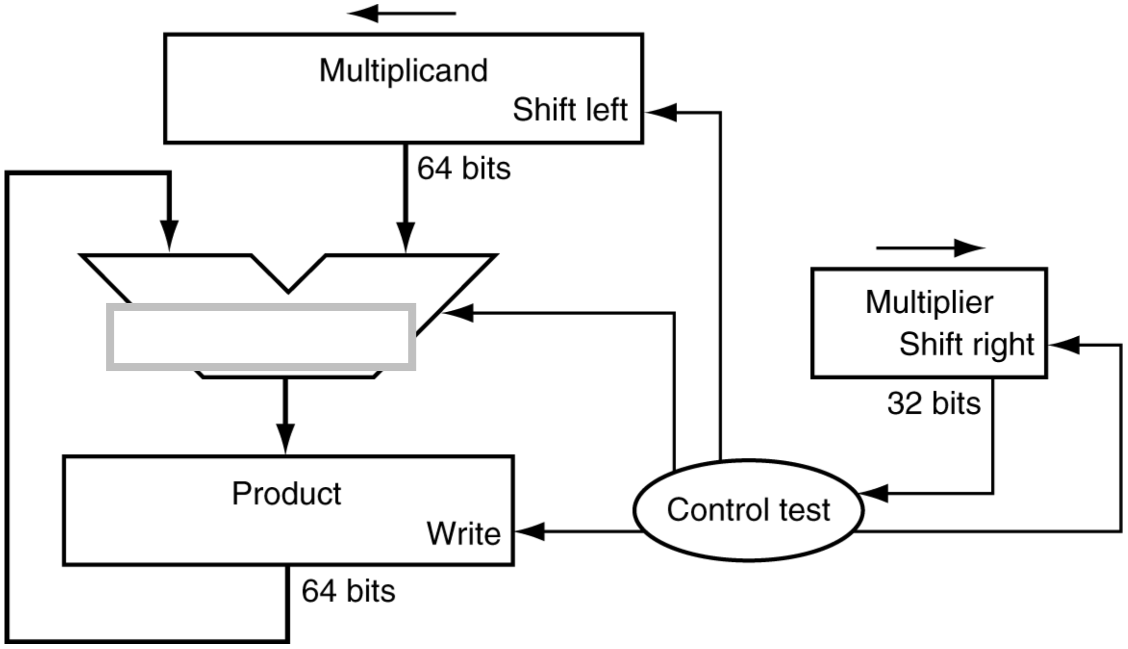
- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register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

Multiplication

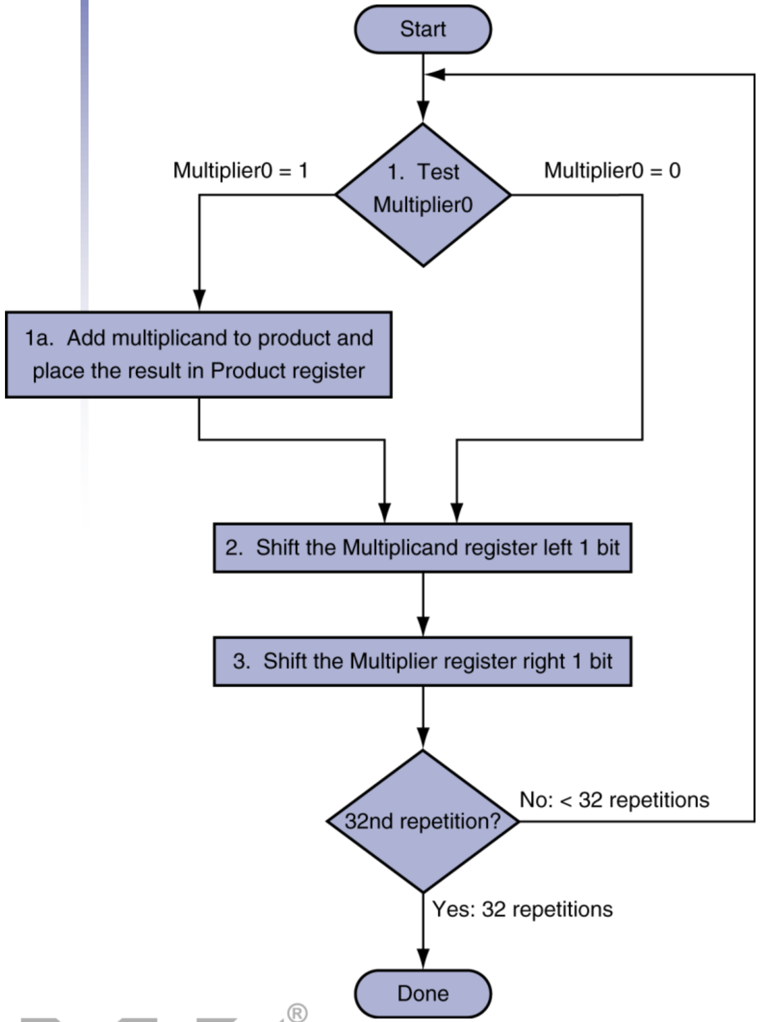
- Start with long-multiplication approach

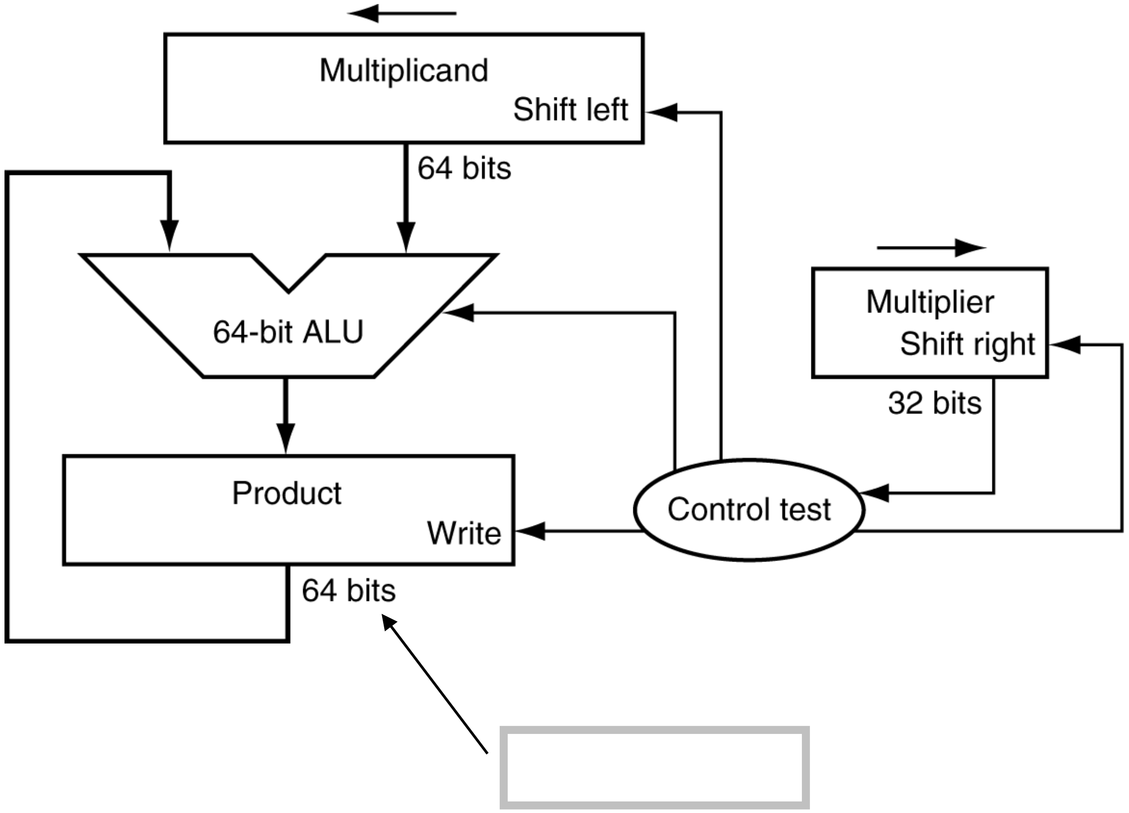
```
      1000
×     1001
   -------
      1000
     0000
    0000
   1000
   -------
   1001000
```

Length of product is the sum of operand lengths



Multiplication Hardware


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

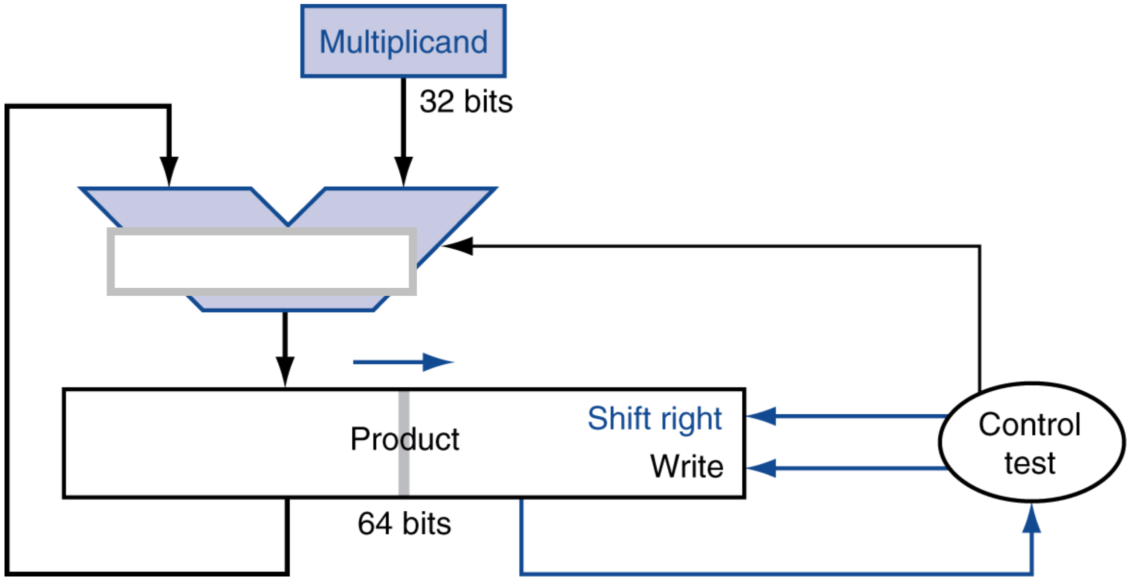


Optimized Multiplier

```
  0010
  0111
----------
  0010
 0010   ←
----------
 00110
0010
----------
001110
```

```
  0010
  0111
----------
  0010
  00010  →
  0010
----------
  00110
  000110
  0010
----------
  001110
```

- Perform steps in parallel: add/shift



```
1000
1011          add
--------
1000 **1011**     shift
01000 **101**
1000          add
--------
11000 **101**     shift
011000 **10**
0000          add
---------
011000 **10**     shift
0011000 **1**
1000          add
---------
1011000 **1**     shift
01011000
```

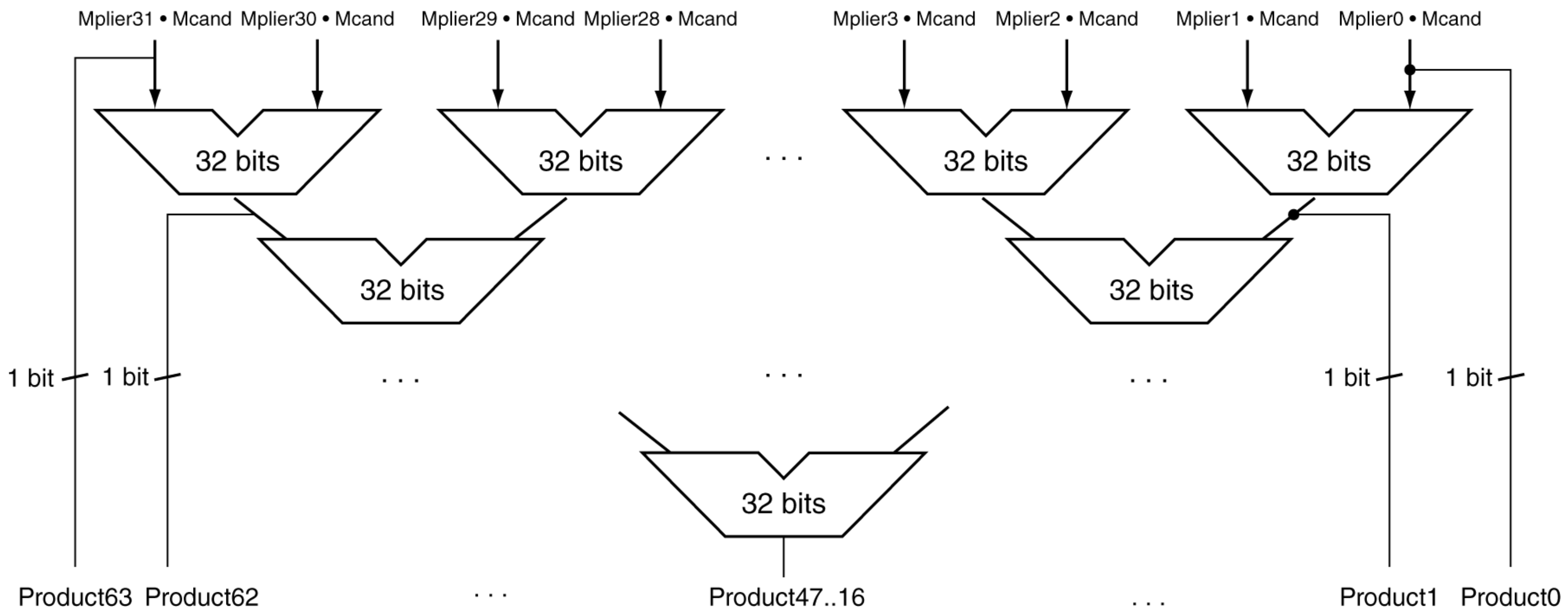
- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

Multiplication Example

Iteration	Step	Product Register (Product	: Multiplier)	Multiplicand
0	Initial value	0000	001$\boxed{1}$	0010
1	1: 1→Prod+=Mcand	0010	0011	0010
	2: shift right Preg	0001	000$\boxed{1}$	0010
2	1: 1→Prod+=Mcand	0011	0001	0010
	2: shift right Preg	0001	100$\boxed{0}$	0010
3	1: 0→no operation	0001	1000	0010
	2: shift right Preg	0000	110$\boxed{0}$	0010
4	1: 0→no operation	0000	1100	0010
	2: shift right Preg	0000	0110	0010

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

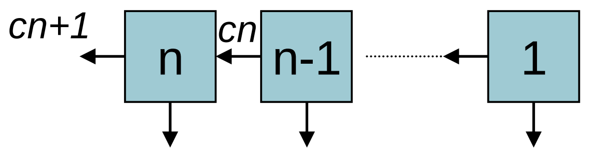


- Can be pipelined
  - Several multiplication performed in parallel

Fast Carry Using the First Level of Abstraction

- *ci+1: carry output of level i, carry input of level i+1*

$$\begin{aligned} c_{i+1} &= (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) \\ &= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i \end{aligned}$$



- *For example*

$$c2 = (a1 \cdot b1) + (a1 + b1) \cdot ((a0 \cdot b0) + (a0 + b0) \cdot c0)$$

- *We can define generate gi and propagate pi*

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i + b_i \end{aligned}$$

such that $c_{i+1} = g_i + p_i \cdot c_i$

- *if ai = bi = 1* $c_{i+1} = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1$

- *if ai = 1, bi = 0 or ai = 0, bi = 1* $c_{i+1} = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$

- $$\begin{aligned} c1 &= g0 + (p0 \cdot c0) \\ c2 &= \mathbf{g1} + (\mathbf{p1} \cdot \mathbf{c1}) = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \\ c3 &= \mathbf{g2} + (\mathbf{p2} \cdot \mathbf{c2}) = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \end{aligned}$$

  $\overset{1}{1}$ or $\overset{1}{0}$ 1 $\overset{1}{1}$ or $\overset{1}{0}$ 1 or $\overset{1}{0}$ $\overset{1}{0}$ 1 or $\overset{1}{0}$ $\overset{1}{0}$ $\overset{1}{0}$ 1

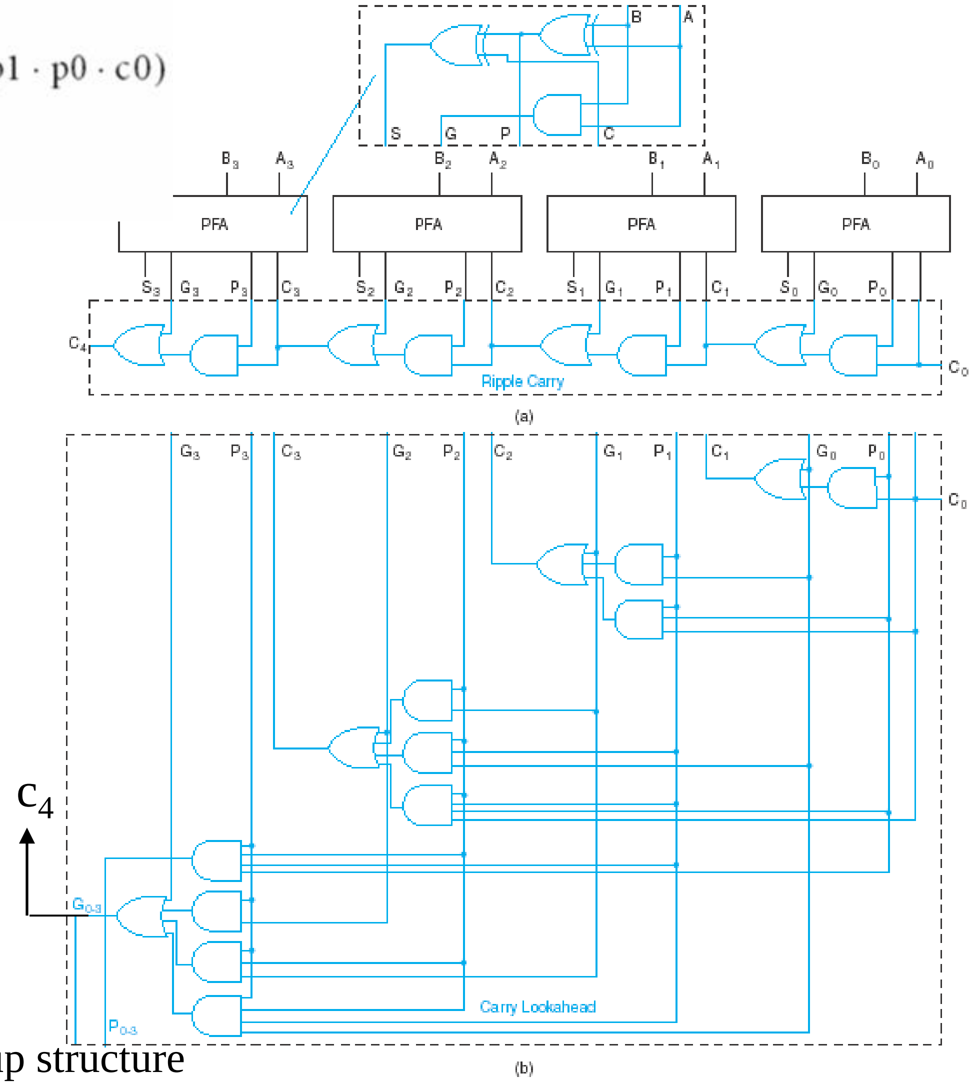
  $$\begin{aligned} c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ &\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0) \end{aligned}$$

4-bit Carry Look-Ahead Adder

$$
\begin{aligned}
c1 &= g0 + (p0 \cdot c0) \\
c2 &= \mathbf{g1 + (p1 \cdot c1)} = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \\
c3 &= \mathbf{g2 + (p2 \cdot c2)} = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \\
c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
&\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)
\end{aligned}
$$

C_4: *4* AND gates and *1* OR gate
C_n: *n* AND gates and *1* OR gate

4-bit CLA adder



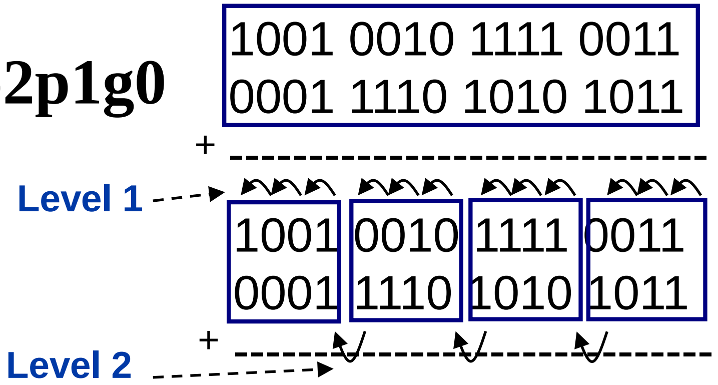
group structure

Fast Carry Using the Second Level of Abstraction

- The concept can be extended another level by considering *group generate* (g0-3) and *group propagate* (p0-3) functions:

  g0-3 = g3 + p3g2 + p3p2g1 + p3p2p1g0

  p0-3 = p3p2p1p0



- Using these two equations:

  c4 = g0-3 + p0-3c0

  c8 = g4-7 + p4-7c4

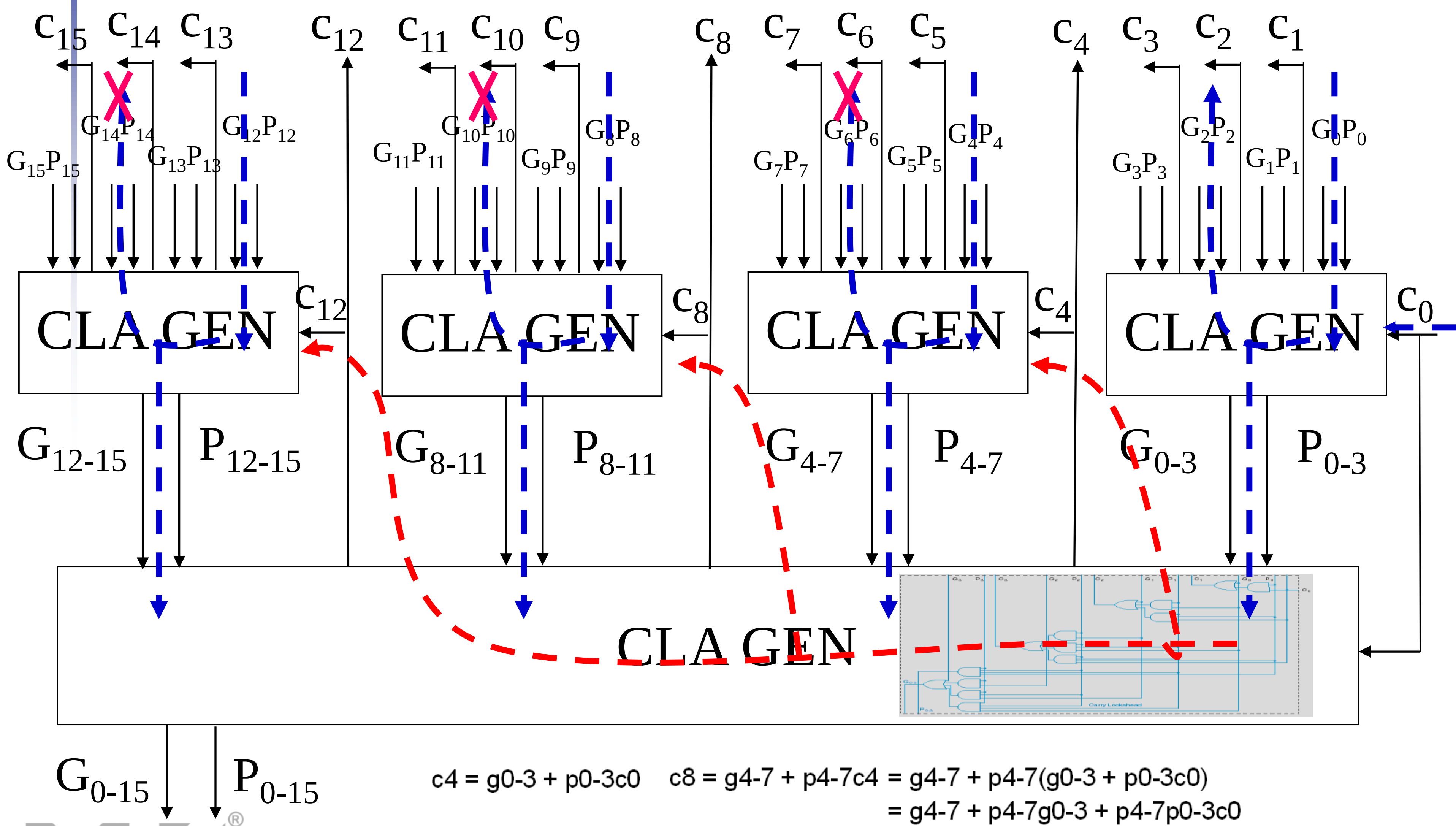
  = g4-7 + p4-7(g0-3 + p0-3c0)

  = g4-7 + p4-7g0-3 + p4-7p0-3c0

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0 C_0$$

- Thus, it is possible to have four 4-bit adders that use one of the same carry lookahead circuit to speed up 16-bit addition

16-bit Two-Level Carry Look-Ahead Adder

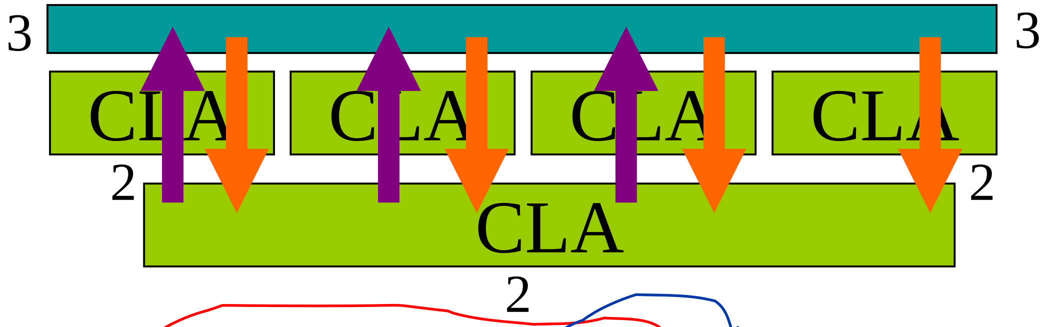


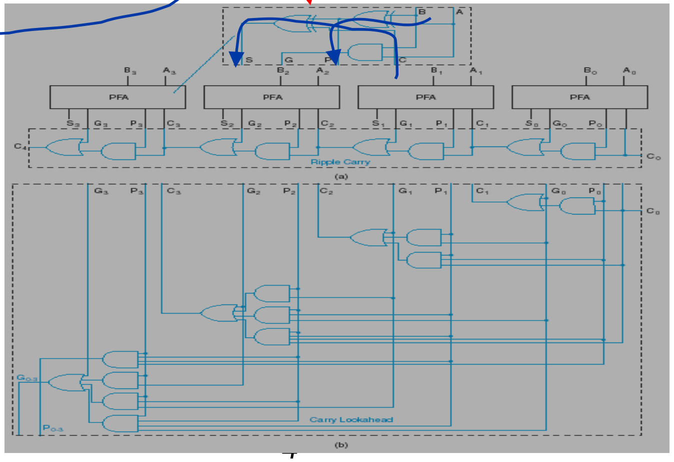
c_{15} c_{14} c_{13} c_{12} c_{11} c_{10} c_9 c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1
$G_{15}P_{15}$ $G_{14}P_{14}$ $G_{13}P_{13}$ $G_{12}P_{12}$
$G_{11}P_{11}$ $G_{10}P_{10}$ G_9P_9 G_8P_8
G_7P_7 G_6P_6 G_5P_5 G_4P_4
G_3P_3 G_2P_2 G_1P_1 G_0P_0
CLA GEN
c_{12}
CLA GEN
c_8
CLA GEN
c_4
CLA GEN
c_0
$G_{12\text{-}15}$ $P_{12\text{-}15}$
$G_{8\text{-}11}$ $P_{8\text{-}11}$
$G_{4\text{-}7}$ $P_{4\text{-}7}$
$G_{0\text{-}3}$ $P_{0\text{-}3}$
CLA GEN
Carry Lookahead
$G_{0\text{-}15}$ $P_{0\text{-}15}$

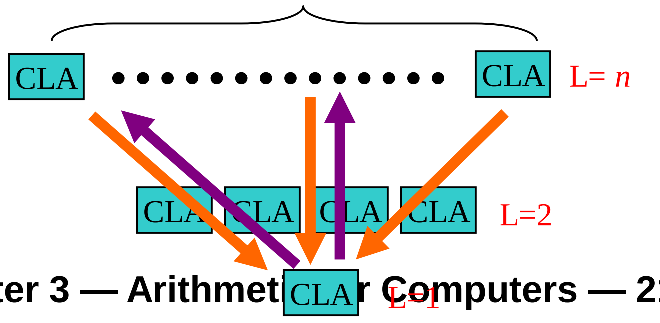
c4 = g0-3 + p0-3c0   c8 = g4-7 + p4-7c4 = g4-7 + p4-7(g0-3 + p0-3c0)
= g4-7 + p4-7g0-3 + p4-7p0-3c0

Carry Lookahead Example

- Specifications 1:
  - 16-bit CLA
  - Delays:
    - NOT = 1
    - XOR = Isolated AND = 3
    - AND-OR = 2
  - Longest Delays:
    - Ripple carry adder*
      = 3 + 15 × 2 + 3 = 36
    - CLA = 3 + 3 × 2 + 3 = 12
- Specification 2:
  - Exclusive OR = *2* gate delays (GDs)
  - *2*-level *16*-bit CLA delay = *10* GDs
  - *3*-level *64*-bit CLA delay = *14* GDs
  - *n*-level *4^n*-bit CLA delay = 4







Simplified Multiplication

- Consider 01110 = 1 x 2^3 + 1 x 2^2 + 1 x 2^1 (three additions)
- One important observation - another faster calculation
  - 01110 = 1 x 2^4 - 1 x 2^1 (one addition and one subtraction)
- Multiplication has similar property
  - The process on the left is traditional operation
  - The process on the right applies the above concept
    - $0010 \times 0110 = 0 \times (0010 \times 2^0) + 1 \times (0010 \times 2^1) + 1 \times (0010 \times 2^2) + 0 \times (0010 \times 2^3)$
    - $0010 \times 0110 = 0 \times (0010 \times 2^0) - 1 \times (0010 \times 2^1) + 0 \times (0010 \times 2^2) + 1 \times (0010 \times 2^3)$

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

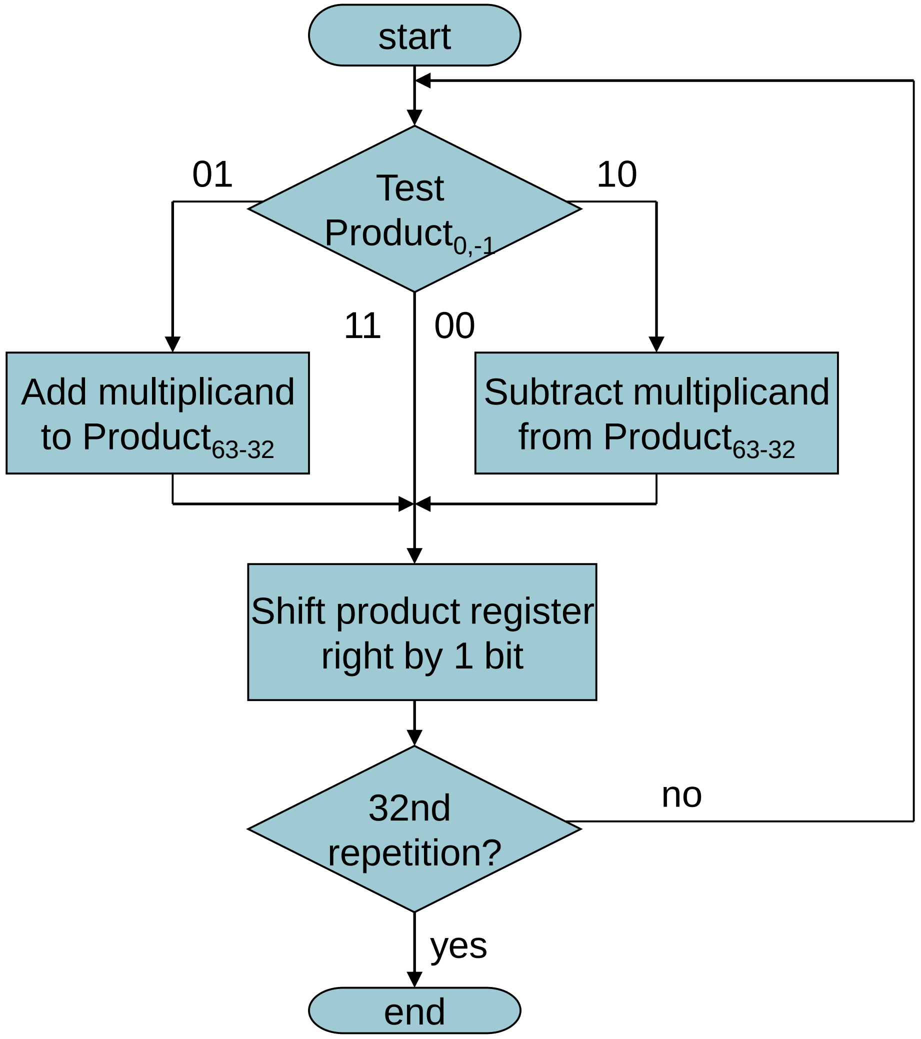
```
        0010two
x       0110two
      ---------
+       0000    shift (0 in multiplier)
-      0010     sub (first 1 in multiplier)
+ 0000          shift (middle of string of 1s)
+0010           add (prior step had last 1)
 -----
00001100two
```

Booth's Algorithm

Current bit	Bit to the right	Explanation	example
1	0	Beginning of a run of 1s	000011110 00
1	1	Middle of a run of 1s	00001111000
0	1	End of a run of 1s	00001111000
0	0	Middle of a run of 0s	00001111000

- Booth's algorithm
  - Based on the current and previous bits, do one of the following
    - 00: middle of a string of 0s, so no arithmetic operation.
    - 01: end of a string of 1s, so add the multiplicand to the left half of the product
    - 10: beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
    - 11: middle of a string of 1s, so no arithmetic operation.
  - As in the previous algorithm, shift the product register right 1 bit

Booth's Algorithm


start
Test Product$_{0,-1}$
01
10
11
00
Add multiplicand to Product$_{63-32}$
Subtract multiplicand from Product$_{63-32}$
Shift product register right by 1 bit
32nd repetition?
no
yes
end

Examples for Booth's Algorithm

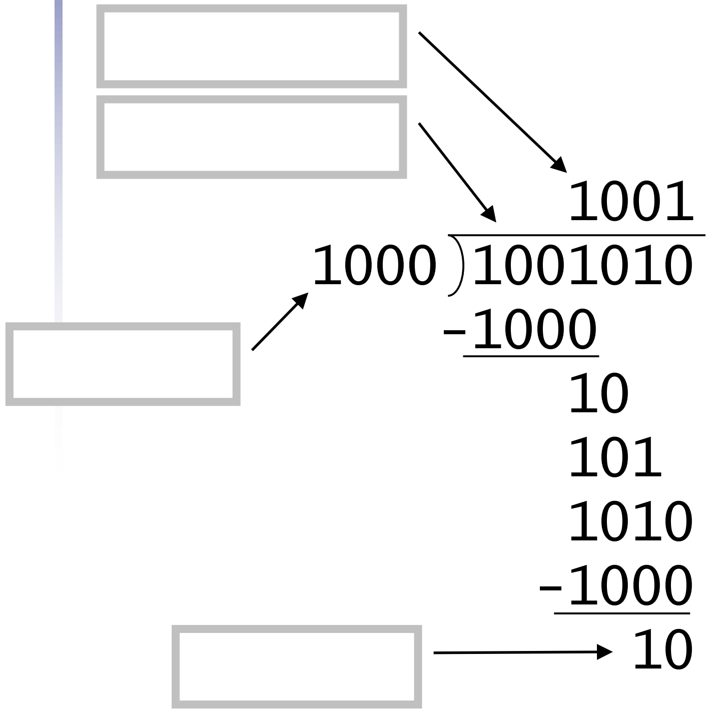
Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1c: 10 ⇒ Prod = Prod – Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1d: 11 ⇒ no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 ⇒ no operation	0001 1000	1b: 01 ⇒ Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: 11 ⇒ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

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

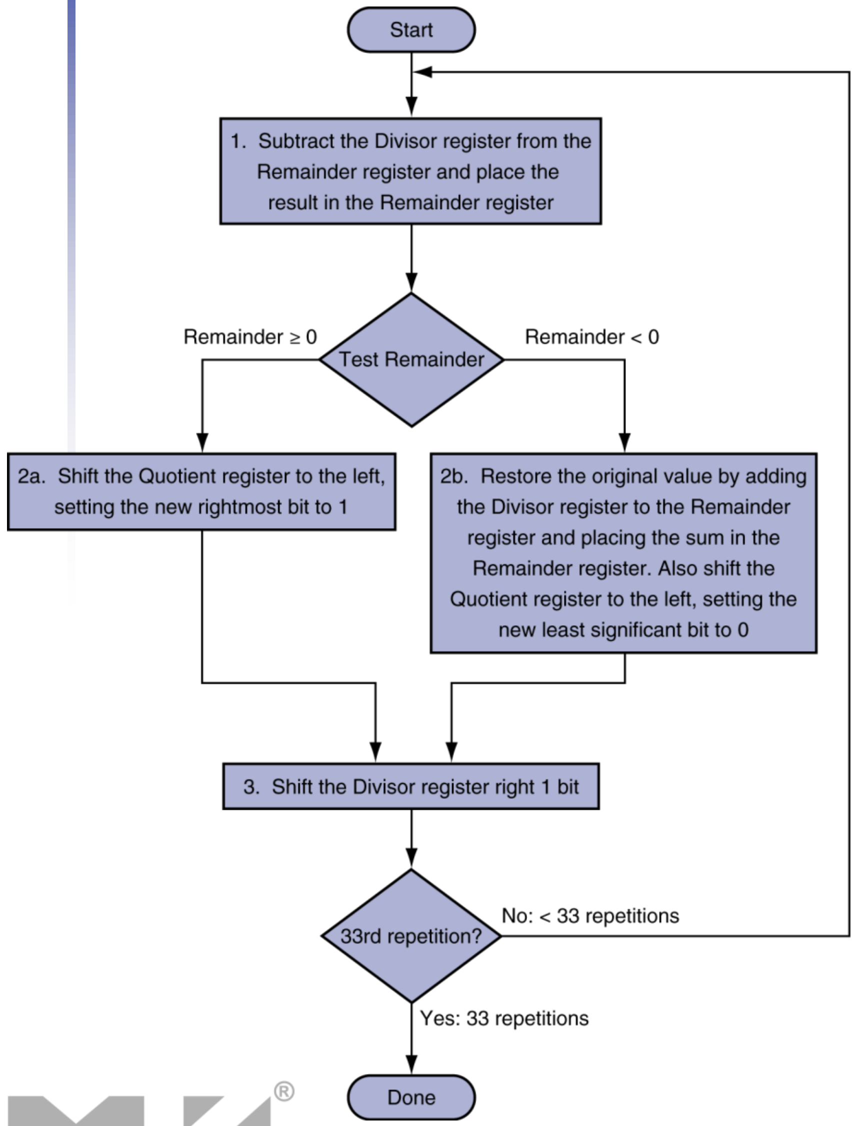
Division



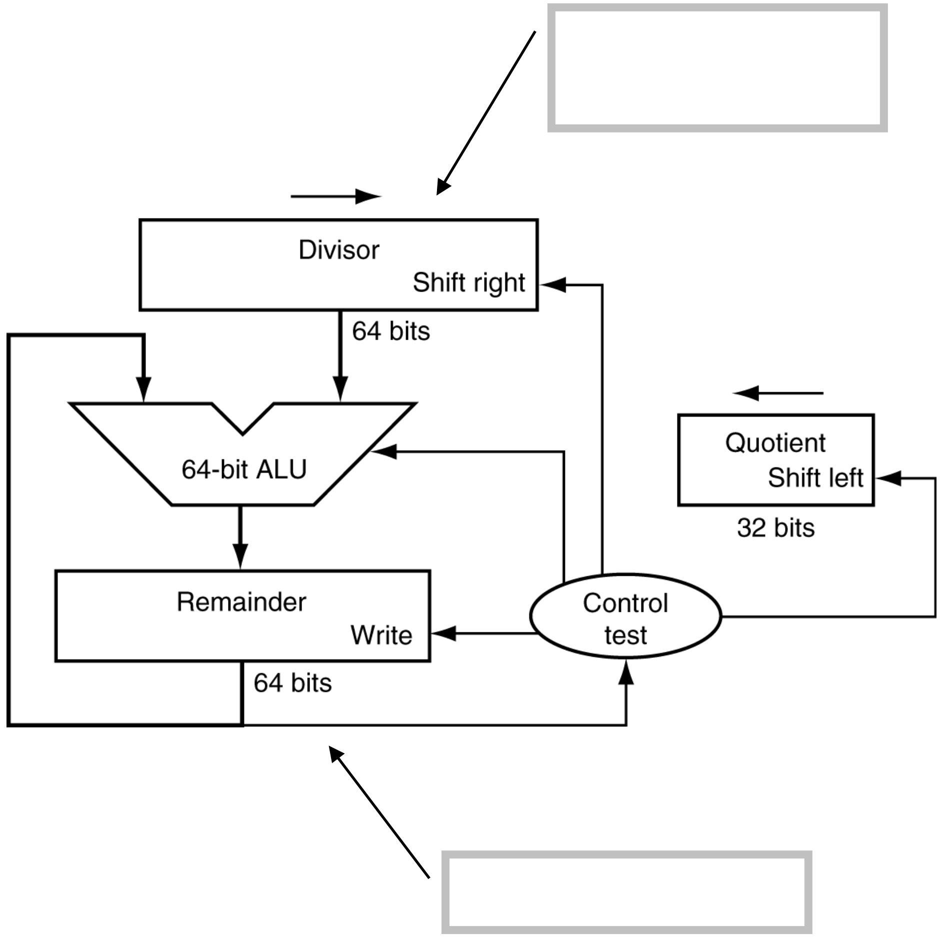
n-bit operands yield *n*-bit quotient and remainder

- 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



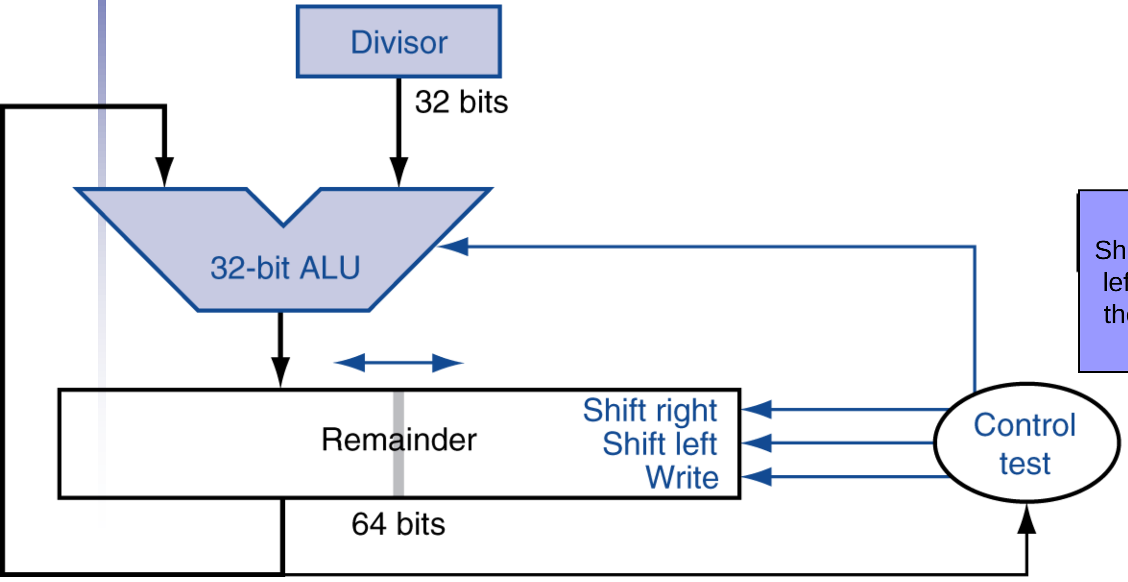
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing 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

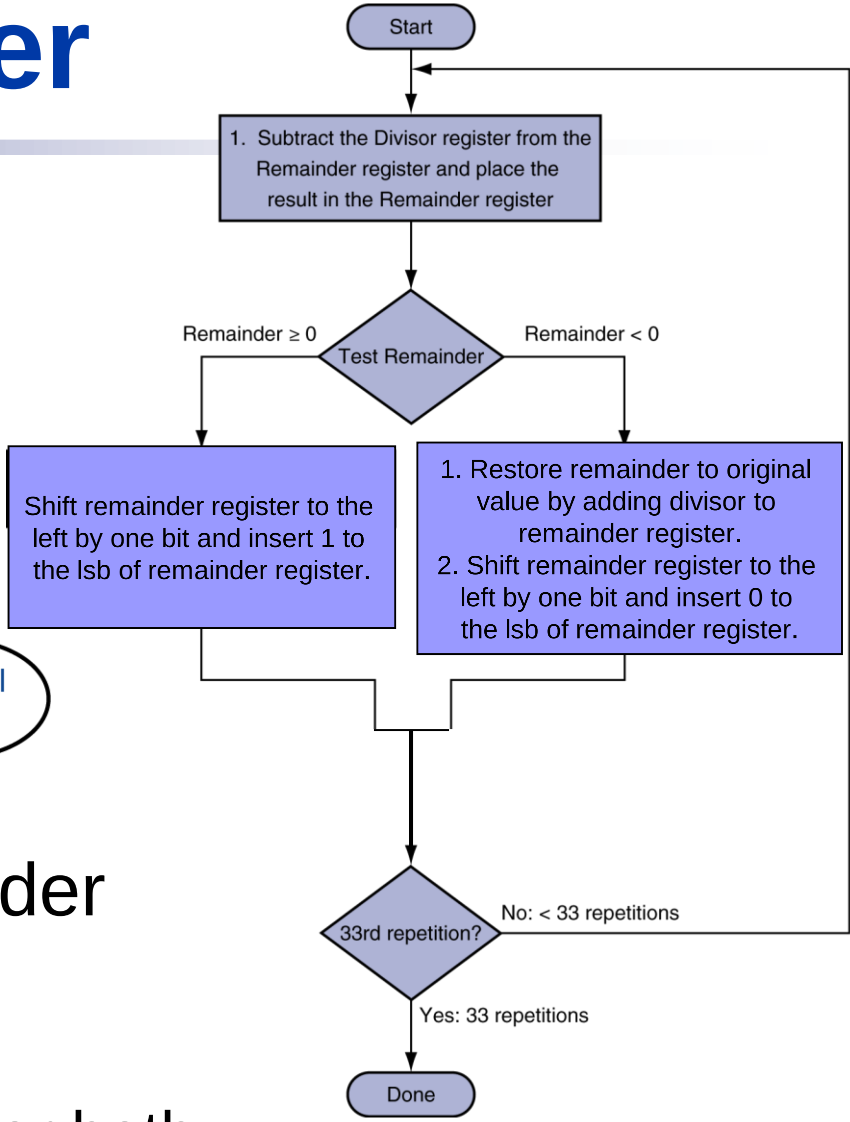


Division Example

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Optimized Divider



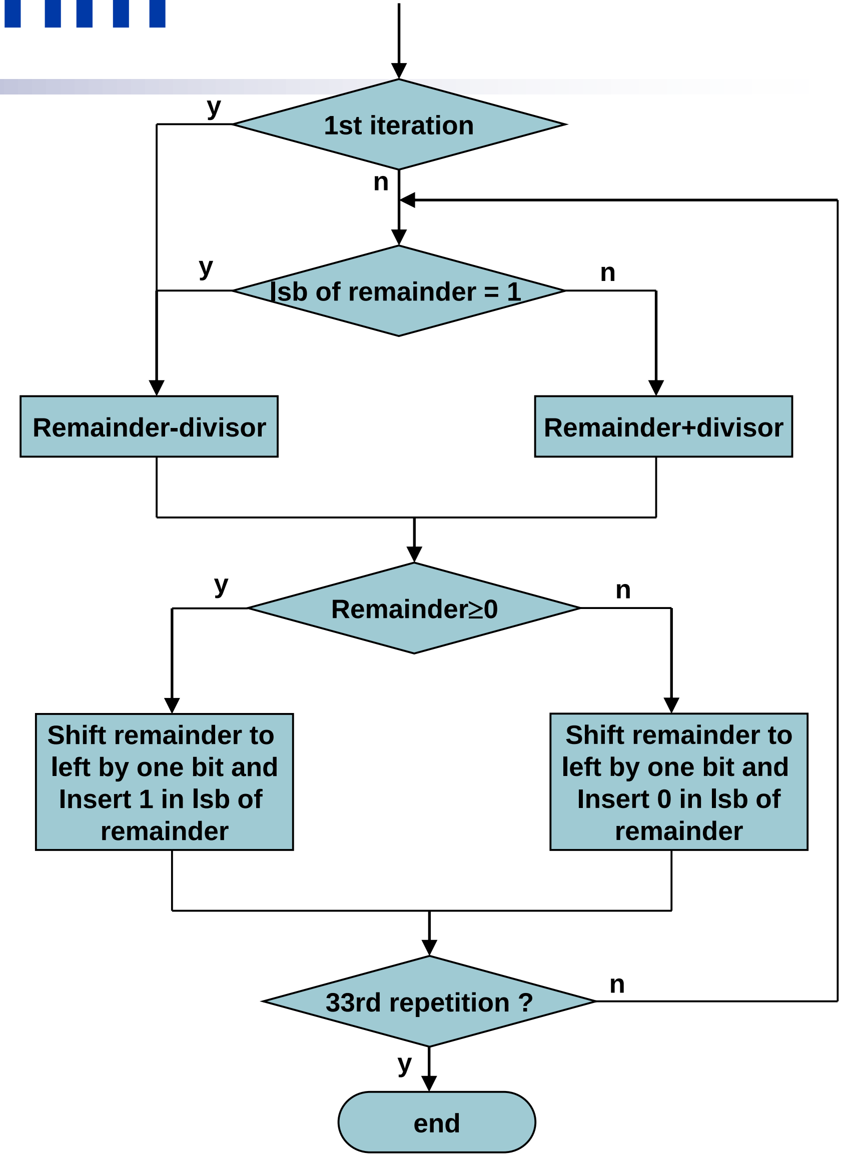


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
Shift remainder register to the left by one bit and insert 1 to the lsb of remainder register.
1. Restore remainder to original value by adding divisor to remainder register.
2. Shift remainder register to the left by one bit and insert 0 to the lsb of remainder register.
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

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

Division Algorithm

- Restoring algorithm
  - r – d (assuming < 0 => quotient = 0)
  - Restore r by adding d: (r – d) + d
  - Next iteration: SLL for 2r, then 2r – d
  - SLL (shift left logical) is nearly free
  -
- Non-restoring algorithm (on the right)
  - r – d (assuming < 0 => quotient = 0)
  - Next iteration: SLL for 2(r – d)
  - Want 2r – d: 2(r – d) + d
  -



1st iteration
y
n
lsb of remainder = 1
y
n
Remainder-divisor
Remainder+divisor
Remainder≥0
y
n
Shift remainder to left by one bit and Insert 1 in lsb of remainder
Shift remainder to left by one bit and Insert 0 in lsb of remainder
33rd repetition ?
n
y
end

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) 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

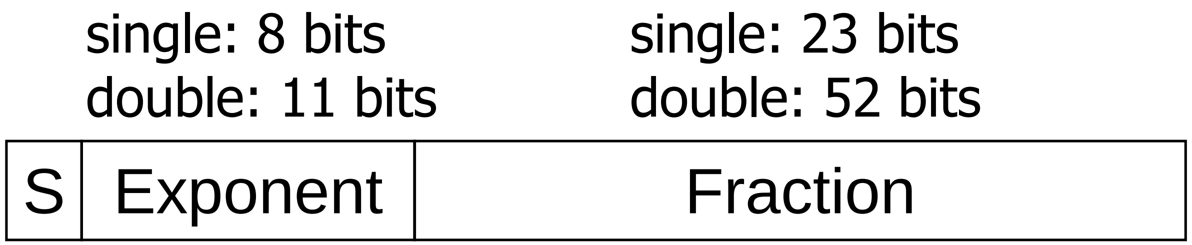
Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision
  - Double precision

IEEE Floating-Point Format



- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand:
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation:
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent:
    ⇒ actual exponent =
  - Fraction: 000…00 ⇒ significand = 1.0
  -
- Largest value
  - Exponent:
    ⇒ actual exponent =
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  -

3 bits

		+11
3:	011	110
2:	010	101
1:	001	100
0:	000	011
-1:	111	010
-2:	110	001
-3:	101	000
-4:	100	111

8 bits

127	254
…	…
0	127
-1	126
…	
-126	1
-127	0
-128	255

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

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

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single:
- Double:

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = 10000001_2 = 129
- x =

  =

  =

Denormal Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{1-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

IEEE 754 Encoding of FPN

Single Precision		Double Precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	±denormalized number
1-254	Anything	1-2046	Anything	±floating-point number
255	0	2047	0	$\pm\infty$
255	Nonzero	2047	Nonzero	NaN

- Smallest positive single precision normalized number
  =
- Smallest positive single precision denormalized no. (Hint: Fraction is 23-bit)
  =
- ∞ must obey mathematical conventions: $F + \infty = \infty$; $F/\infty = 0$

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

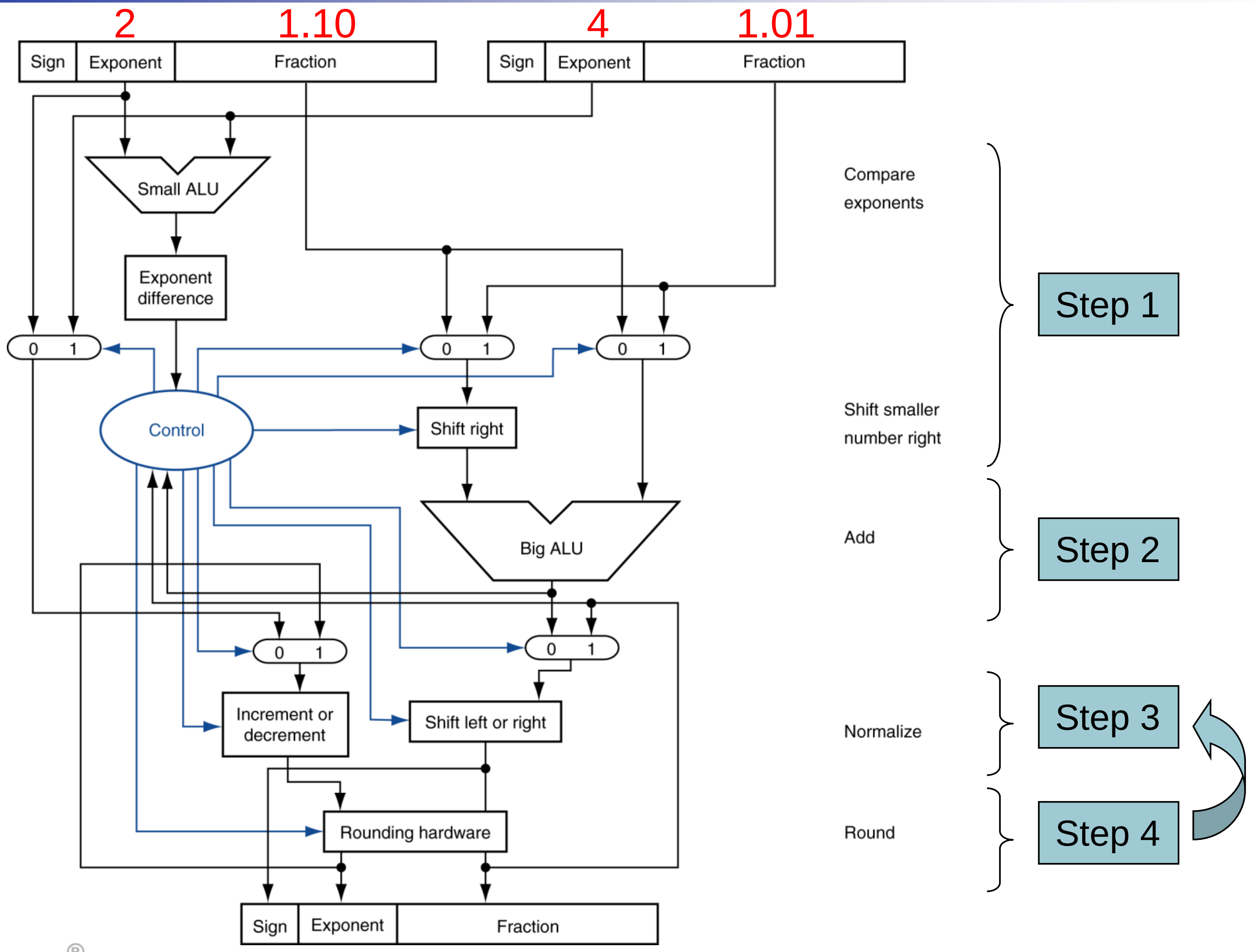
Floating-Point Addition

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

FP Adder Hardware

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

FP Adder Hardware



Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

Interpretation of Data

The BIG Picture

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

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

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

- Need to validate parallel programs under varying degrees of parallelism

Right Shift and Division

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

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

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