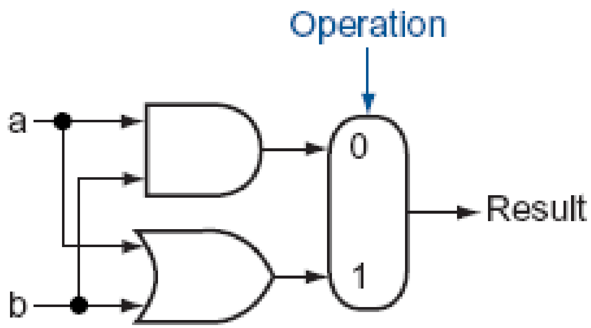
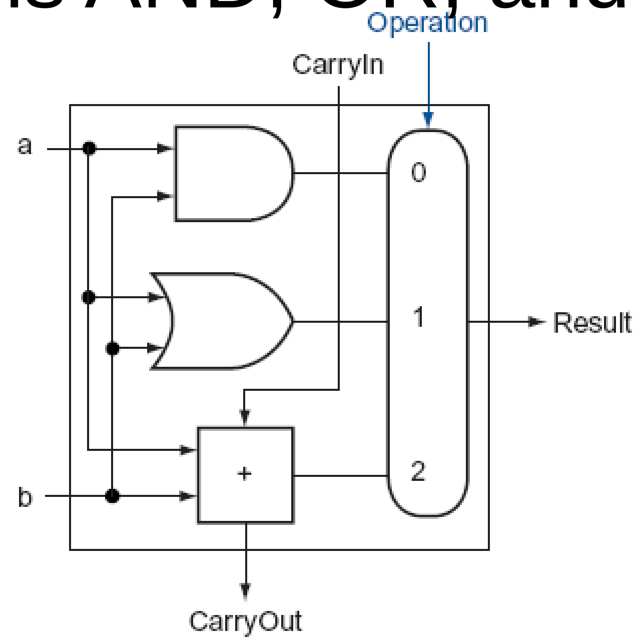
Chapter 3

Arithmetic for Computers

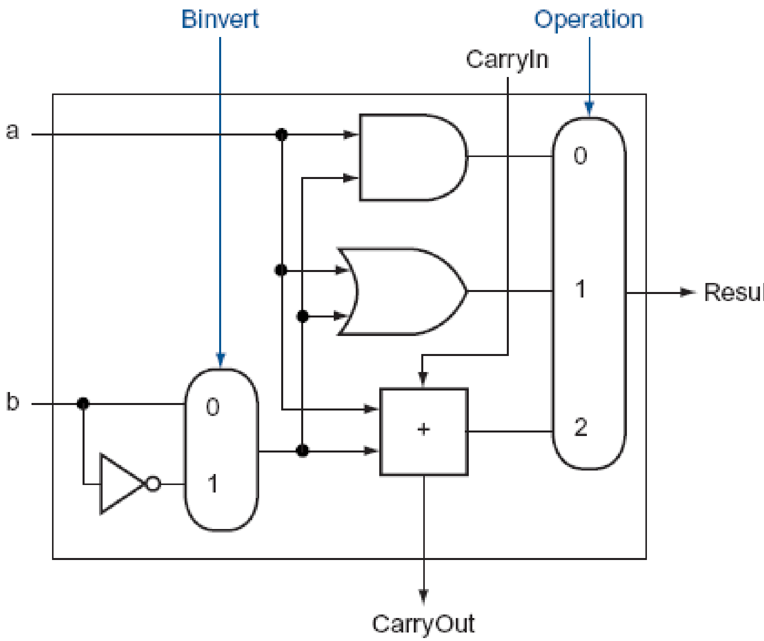
Basic Arithmetic Logic Unit

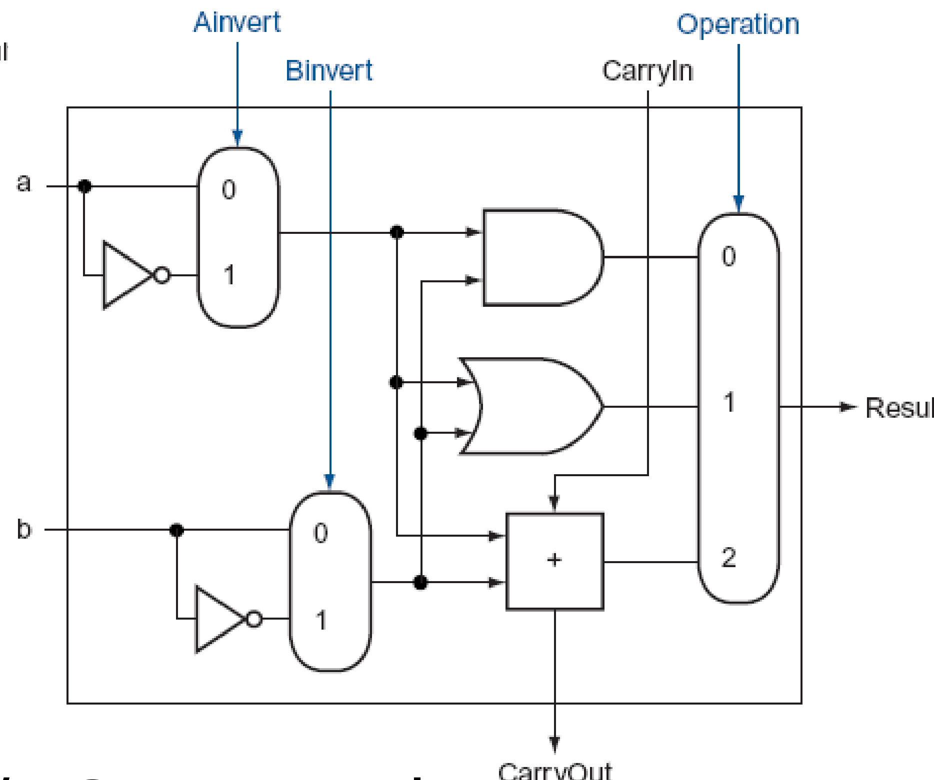
- One-bit ALU that performs AND, OR, and addition





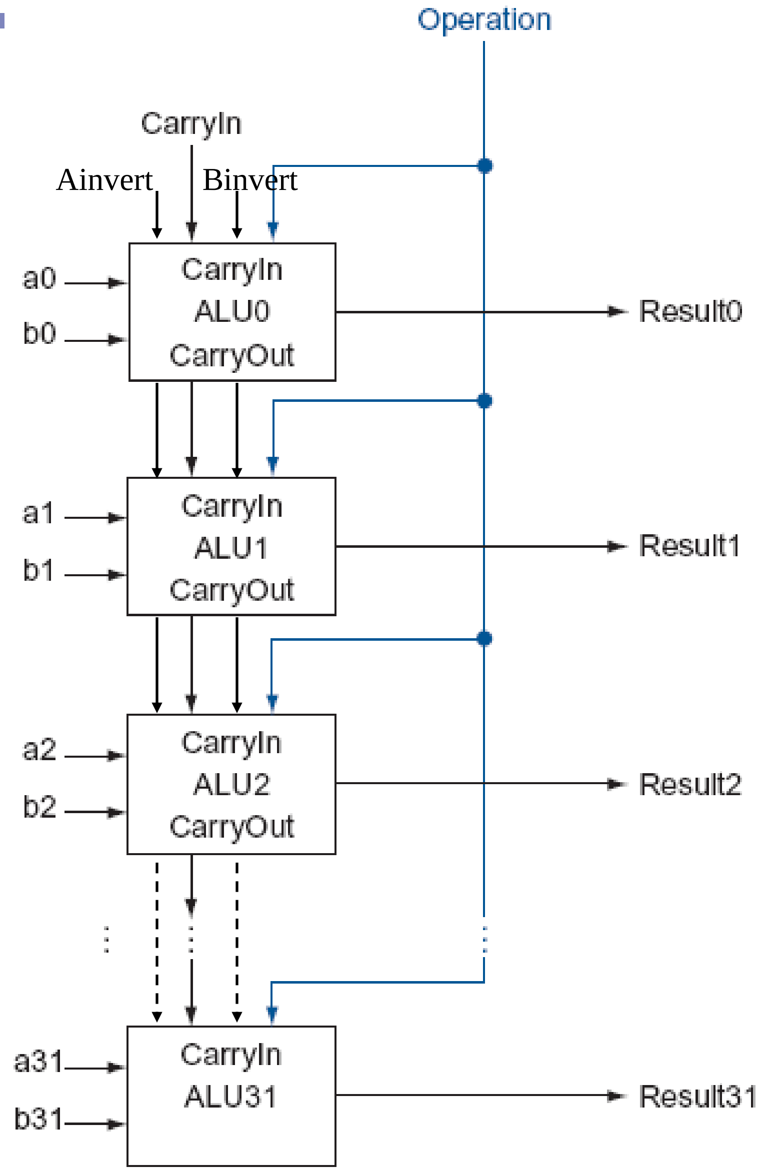
Enhanced Arithmetic Logic Unit





- ALU that have NAND/NOR operation

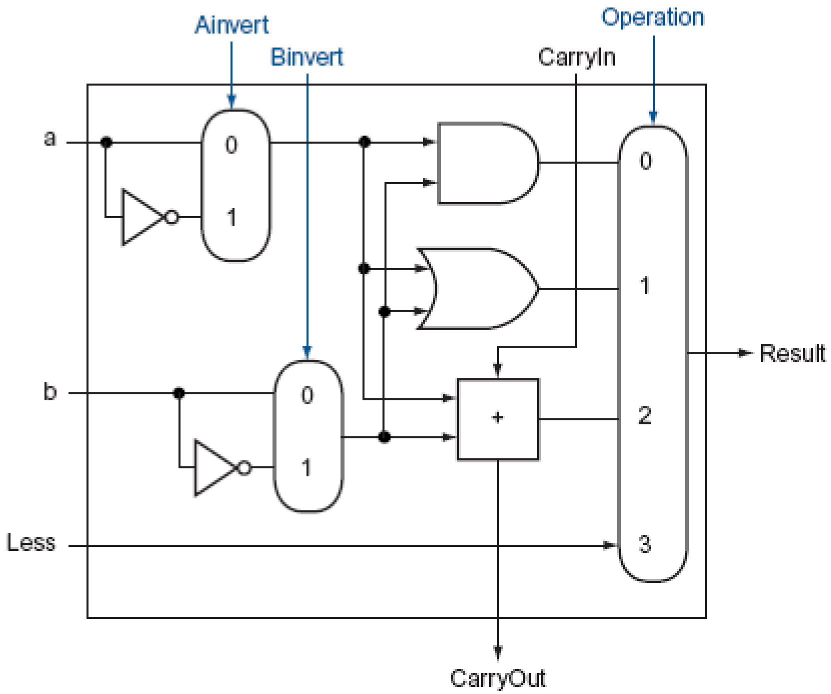
32-bit ALU

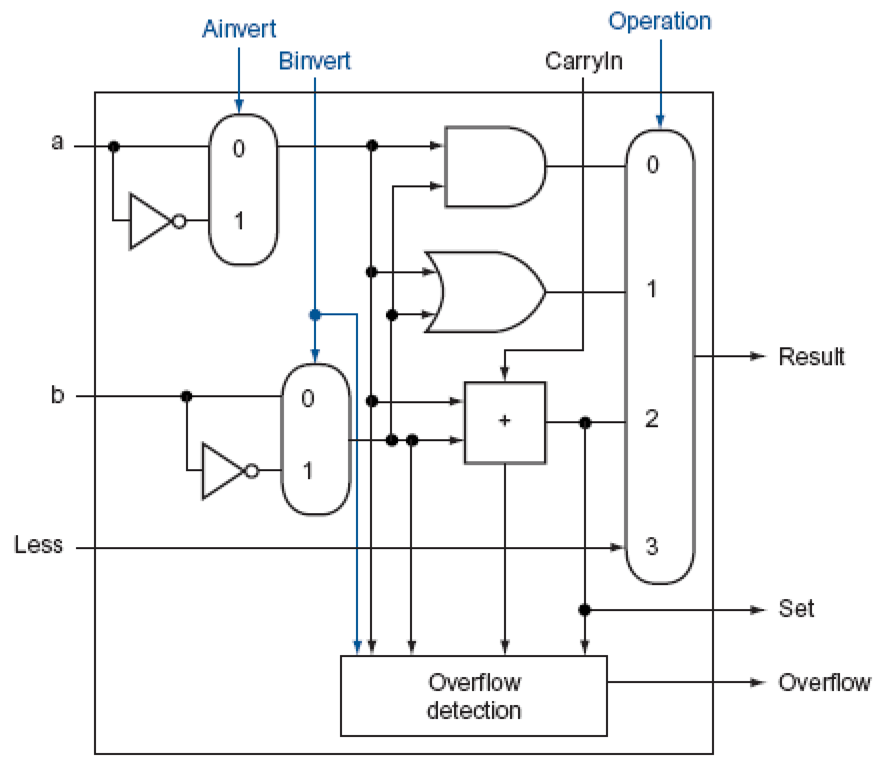


Operation
CarryIn
Ainvert
Binvert
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
a31
b31
CarryIn
ALU31
Result31

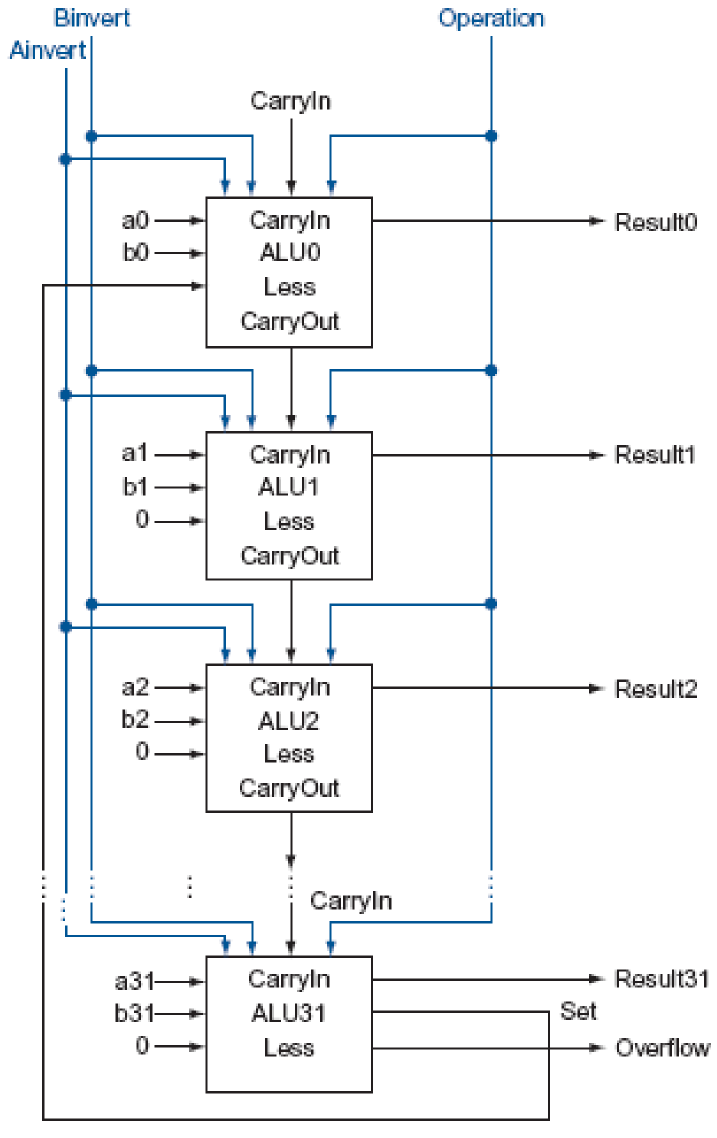
One-bit ALUs with Set Less Than

- 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





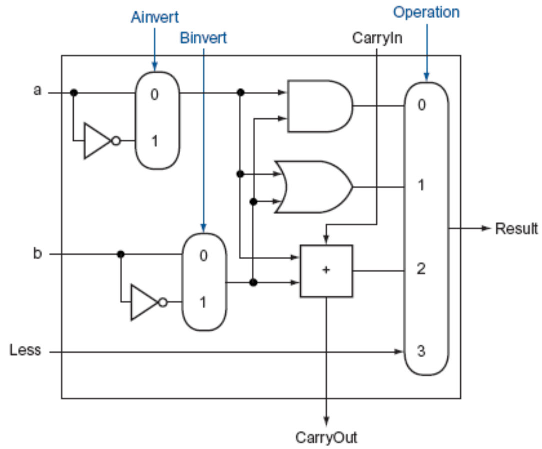
32-bit ALU with Set Less Than

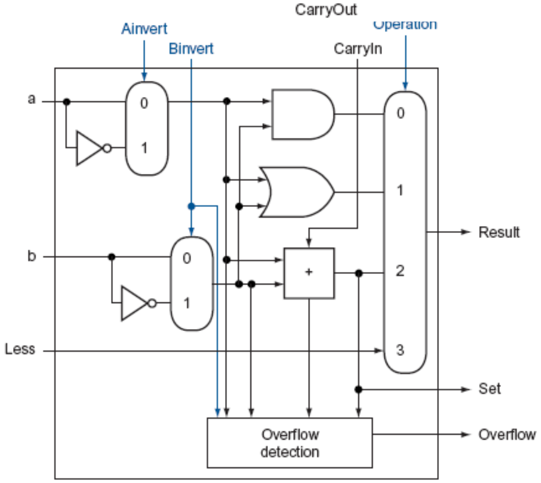


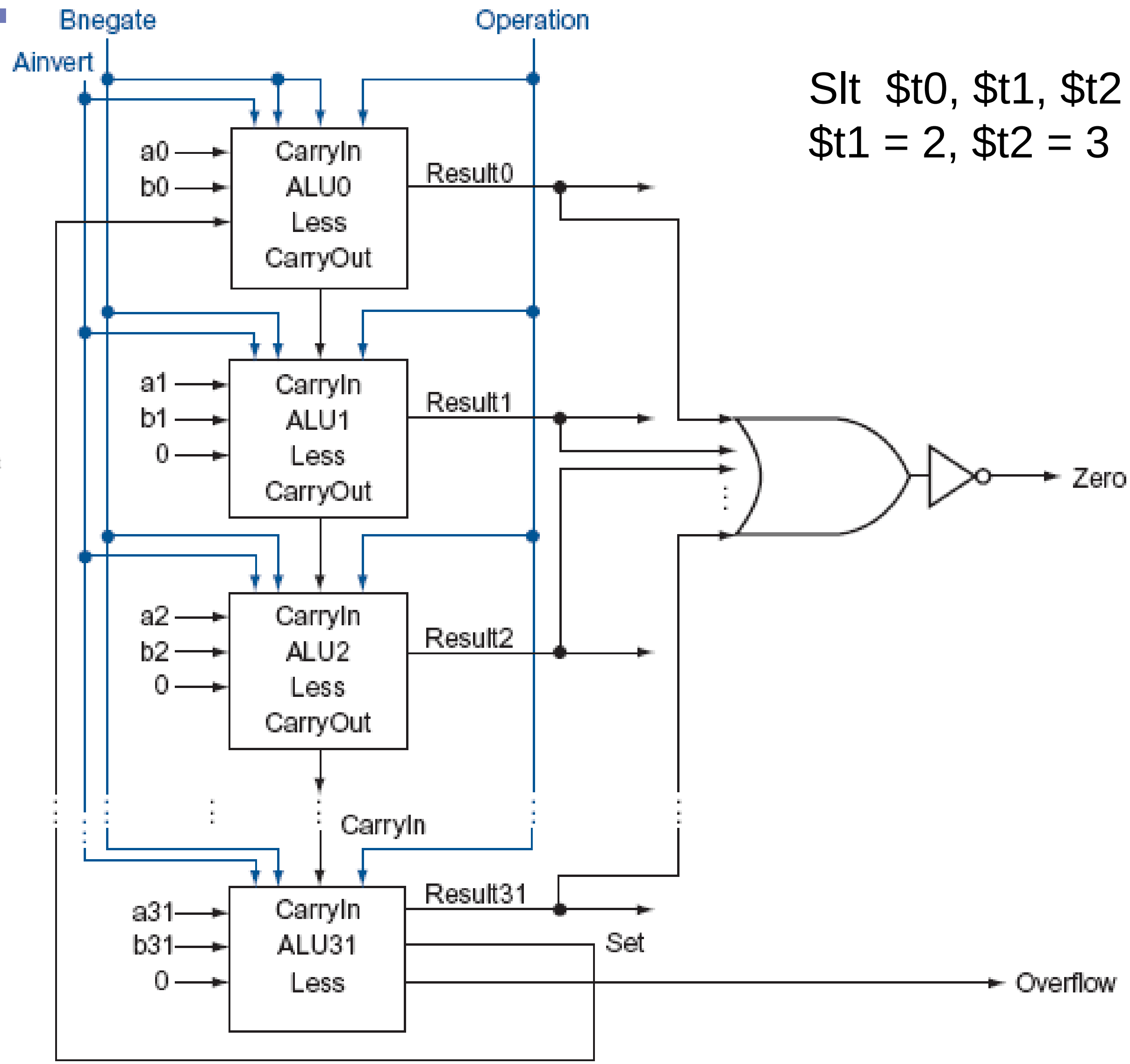
If a < b output 1
else output 0

000…00
000…01

Final 32-bit ALU







Slt $t0, $t1, $t2

$t1 = 2, $t2 = 3

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

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

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 (ex. (-b) – a)
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand (ex. b – (-a))
    - Overflow if result sign is 1

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

Overflow Detection for Signed & Unsigned Addition

Signed addition	addu	$t0, $t1, $t2
	xor	$t3, $t1, $t2
	slt	$t3, $t3, $zero
	bne	$t3, $zero, No_overflow
	xor	$t3, $t0, $t1
	slt	$t3, $t3, $zero
	bne	$t3, $zero, Overflow
Unsigned addition	addu	$t0, $t1, $t2
	nor	$t3, $t1, $zero
	sltu	$t3, $t3, $t2
	bne	$t3, $zero, Overflow

01101

10010

Arithmetic for Multimedia

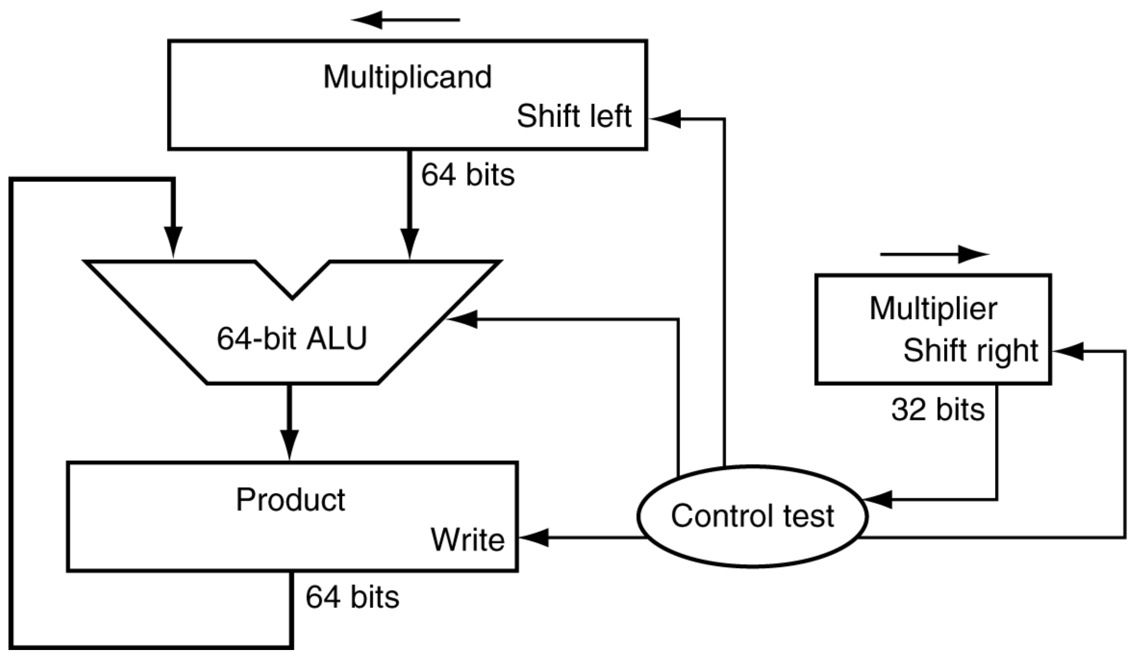
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Multiplication

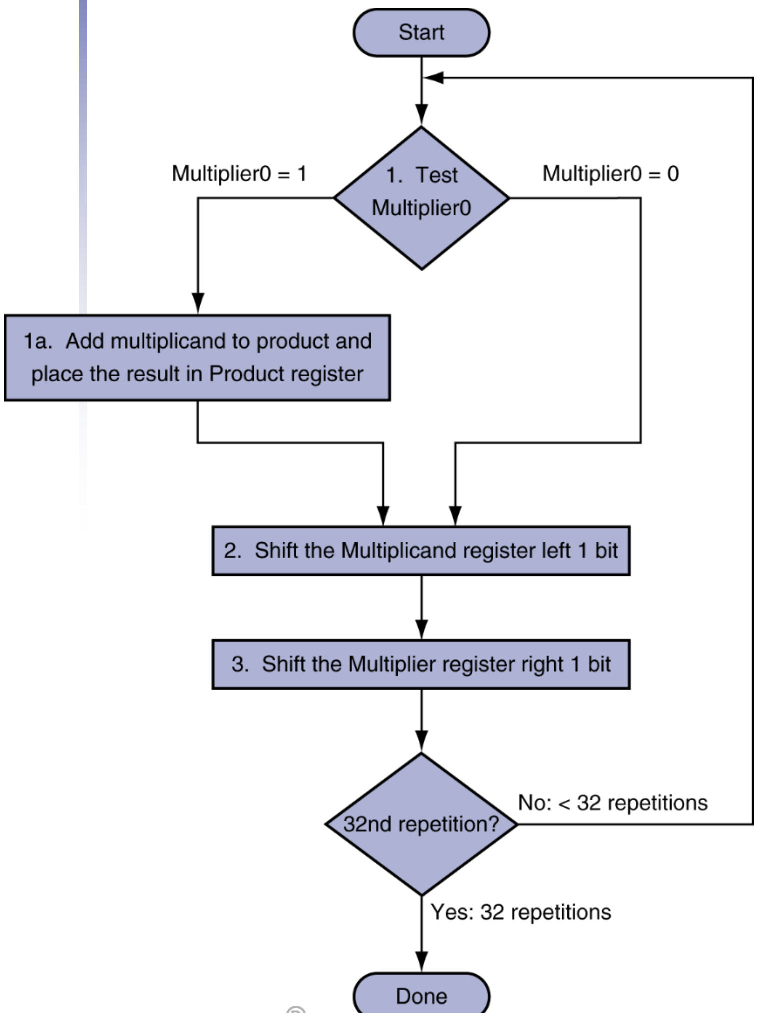
- Start with long-multiplication approach

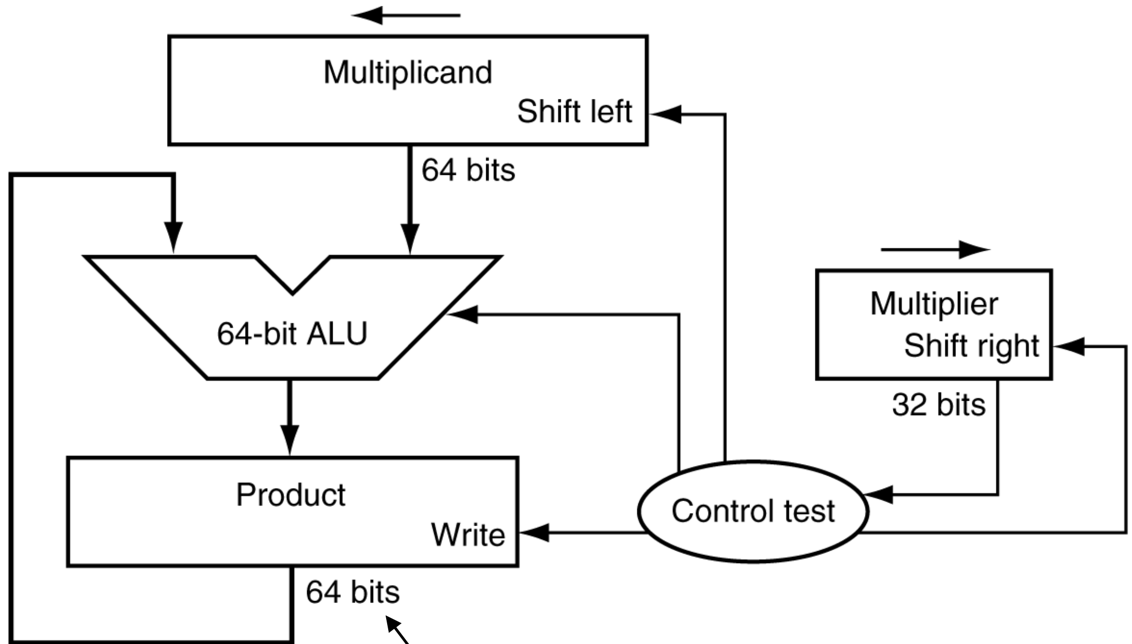
```
multiplicand      1110
multiplier        1011
              ×
                  1110
                 11100
                000000
               1110000
product       10011010
```

Length of product is the sum of operand lengths



Multiplication Hardware

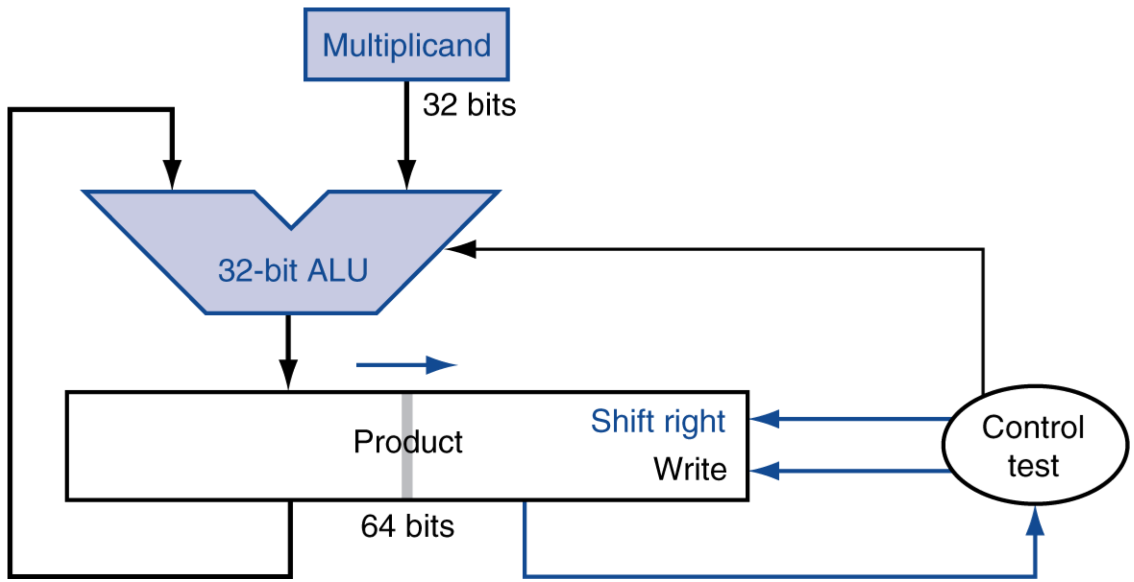




Initially 0

Optimized Multiplier

- Perform steps in parallel: add/shift



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

0010
0111

0010
0010

00110
0010

001110

0010
0111

0010
00010
0010

00110
000110
0010

001110

1000
1011 add

1000 **1011** shift
01000 **101**
1000 add

11000 **101** shift
011000 **10**
0000 add

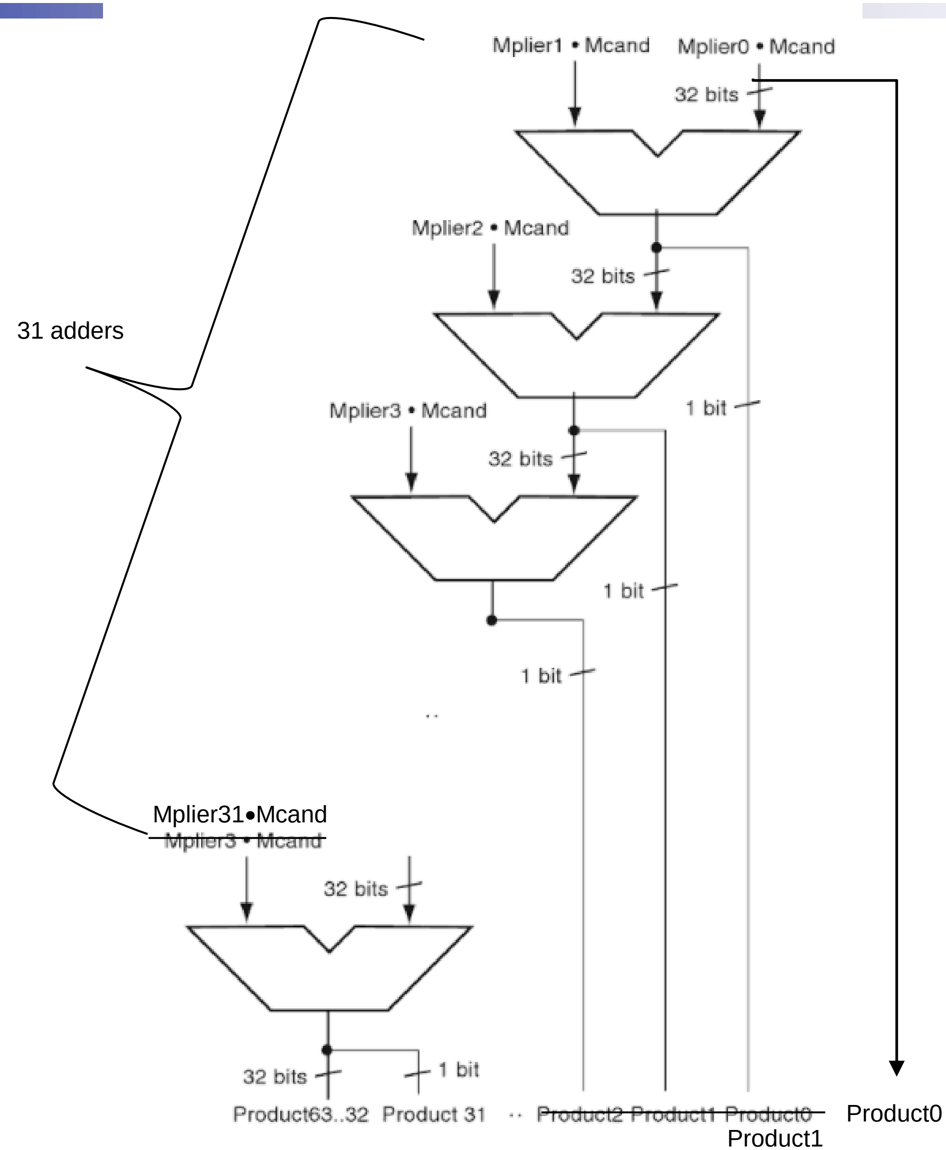
011000 **10** shift
0011000 **1**
1000 add

1011000 **1** shift
01011000

Multiplication Example

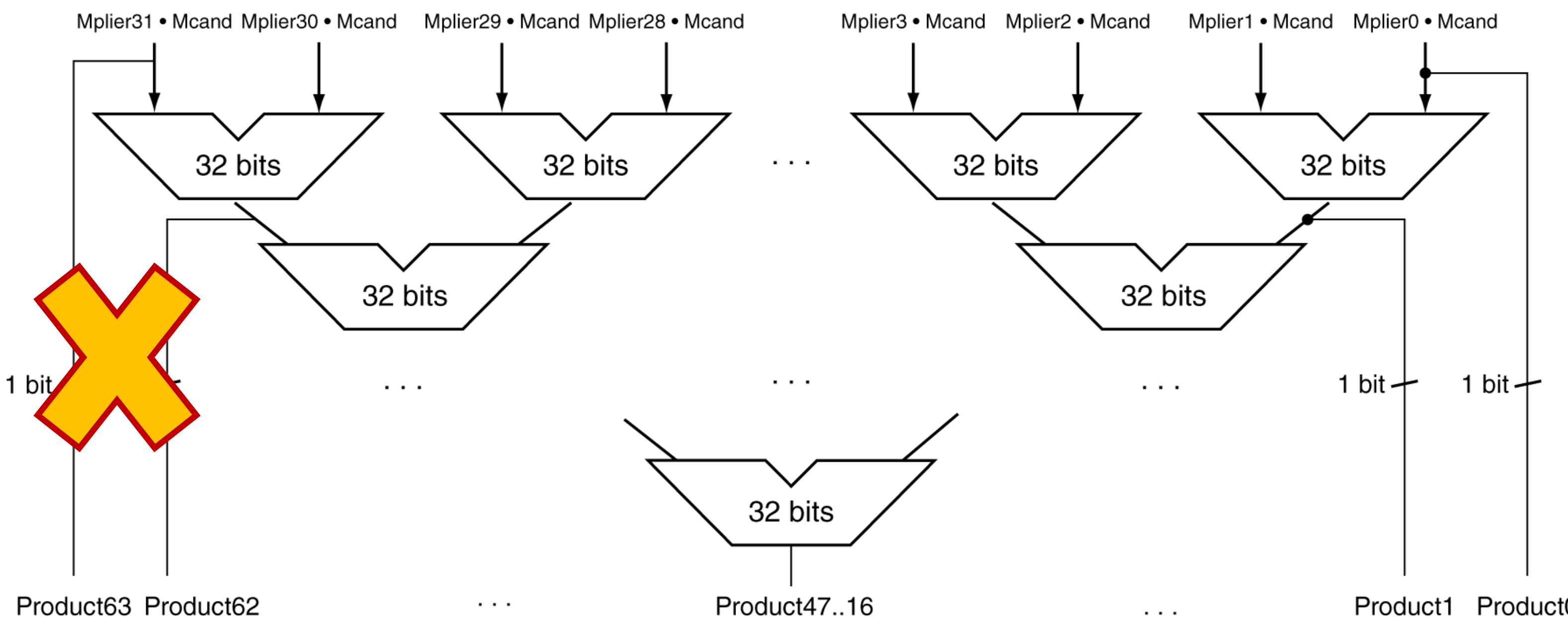
Iteration	Step	Product Register (Product :	Multiplier)	Multiplicand
0	Initial value	0000	0011	0010
1	1: 1→Prod+=Mcand	0010	0011	0010
	2: shift right Preg	0001	0001	0010
2	1: 1→Prod+=Mcand	0011	0001	0010
	2: shift right Preg	0001	1000	0010
3	1: 0→no operation	0001	1000	0010
	2: shift right Preg	0000	1100	0010
4	1: 0→no operation	0000	1100	0010
	2: shift right Preg	0000	0110	0010

Faster Multiplier



Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

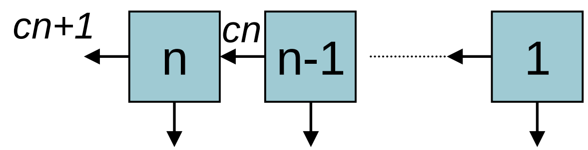


- Can be pipelined
  - Several multiplication performed in parallel

Fast Carry Using the First Level of Abstraction

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

$$c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i)$$
$$= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i$$



- *For example*

$$c_2 = (a_1 \cdot b_1) + (a_1 + b_1) \cdot ((a_0 \cdot b_0) + (a_0 + b_0) \cdot c_0)$$

- *We can define generate gi and propagate pi*

$$g_i = a_i \cdot b_i$$
$$p_i = a_i + b_i$$

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

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

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

- $$c_1 = g_0 + (p_0 \cdot c_0)$$
$$c_2 = \mathbf{g_1 + (p_1 \cdot c_1)} = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0)$$
$$c_3 = \mathbf{g_2 + (p_2 \cdot c_2)} = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot c_0)$$

1 or 1 1 1 or 1 1 or 1 1 1 or 1 1 1 1
1 0 1 0 0 0 0 0 0

$$c_4 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0)$$
$$+ (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0)$$

4-bit Carry Look-Ahead Adder

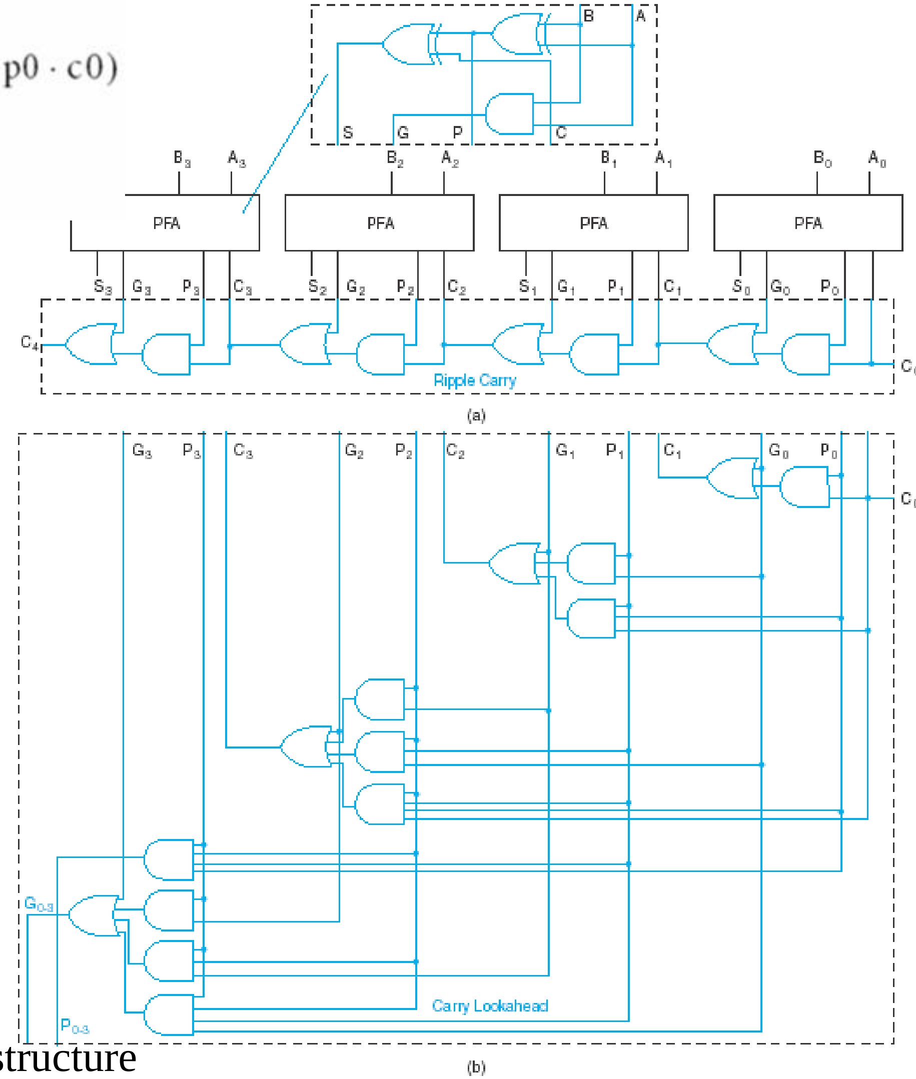
$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot c1) = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot c2) = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

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



4-bit CLA adder

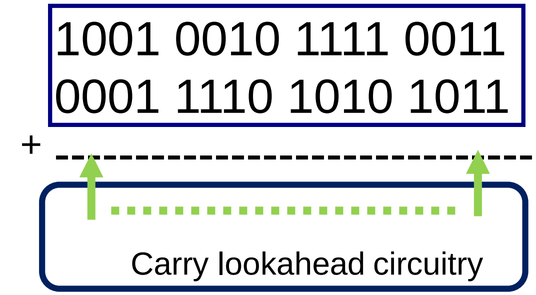
group structure

Fast Carry Using the Second Level of Abstraction

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

  g0-3 = g3 + p3g2 + p3p2g1 + p3p2p1g0

  p0-3 = p3p2p1p0



$$C_4 = g_3 + p_3\,c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0\,c_0$$

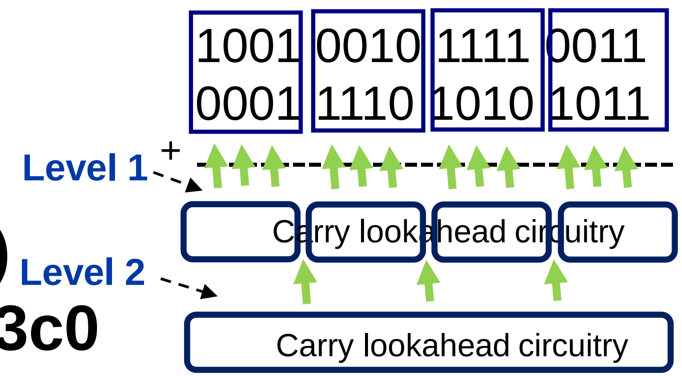
- **Using these two equations:**

  c4 = g0-3 + p0-3c0

  c8 = g4-7 + p4-7c4

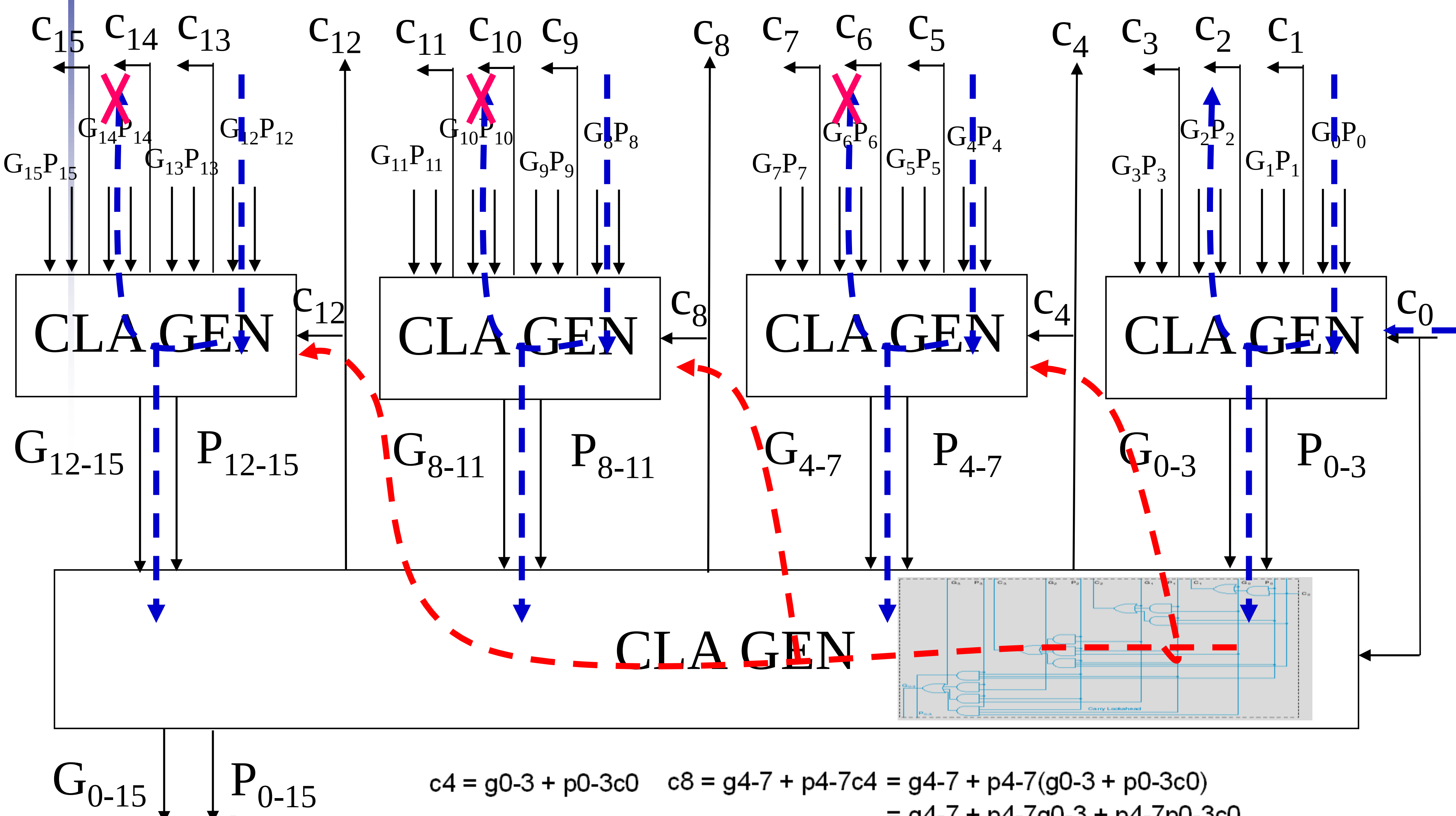
  = g4-7 + p4-7(g0-3 + p0-3c0)

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



- **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

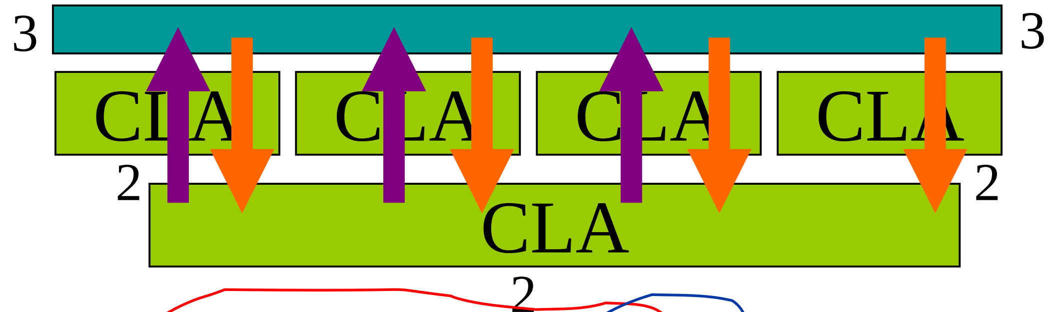


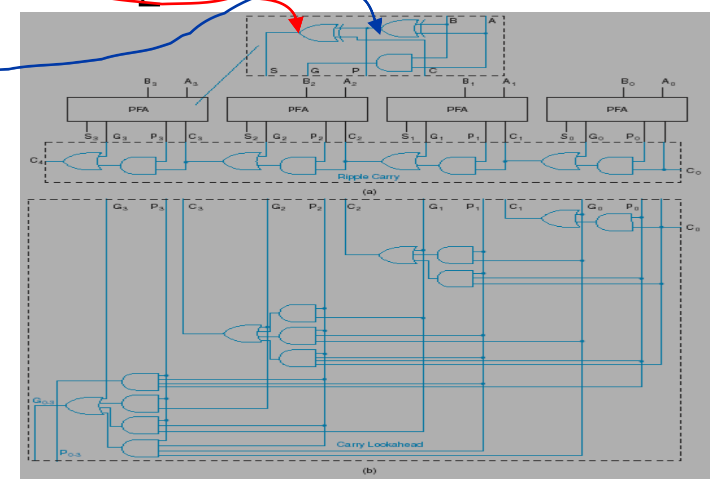
c4 = g0-3 + p0-3c0   c8 = g4-7 + p4-7c4 = g4-7 + p4-7(g0-3 + p0-3c0)

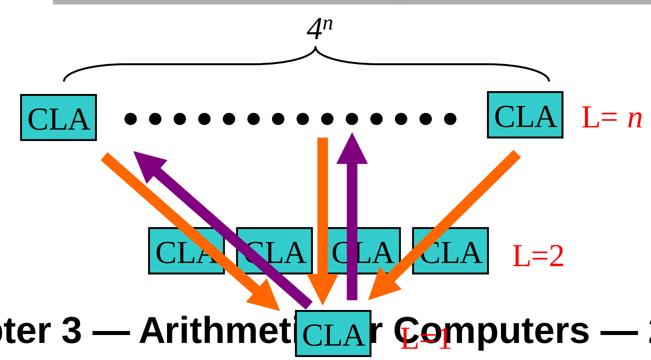
= g4-7 + p4-7g0-3 + p4-7p0-3c0

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 *4ⁿ*-bit CLA delay = *4n + 2*







Simplified Multiplication

- Consider 01110 = $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$ (three additions)
- One important observation - another faster calculation
  - 01110 = $1 \times 2^4 - 1 \times 2^1$ (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
```

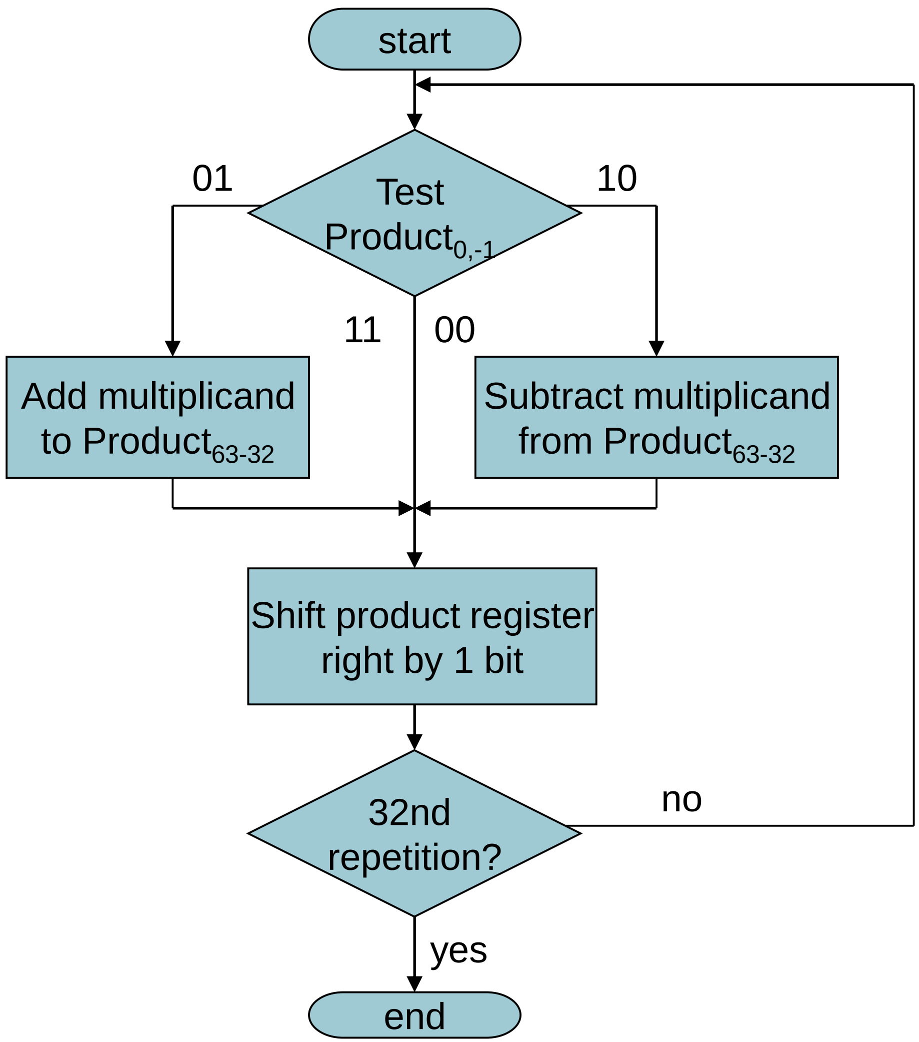
```
        0010two
x       0110two
       --------
+       0000   shift (0 in multiplier)
-      0010    sub (first 1 in multiplier)
+ 0000         shift (middle of string of 1s)
+0010          add (prior step had last 1)
 -----
00001100two
```

Booth's Algorithm

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



Examples for Booth's Algorithm

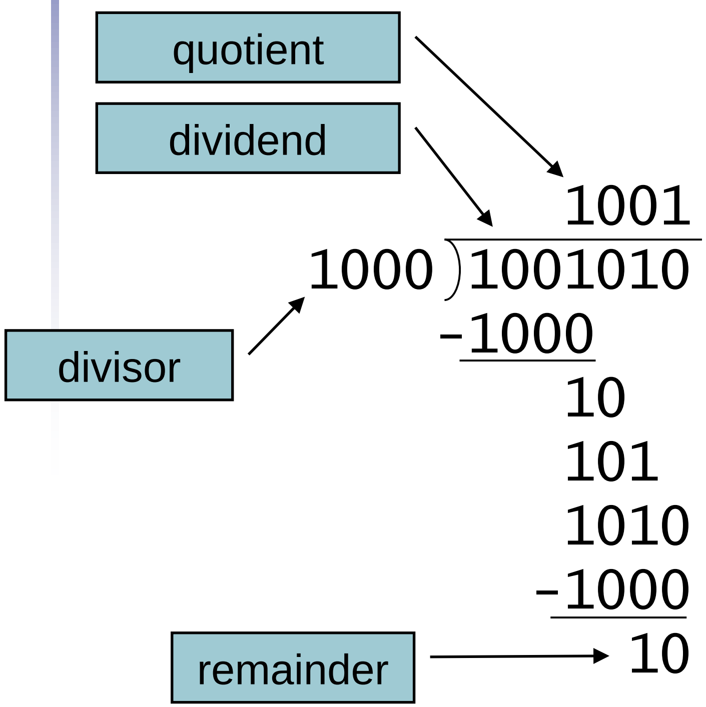
Itera-tion	Multi-plicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	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

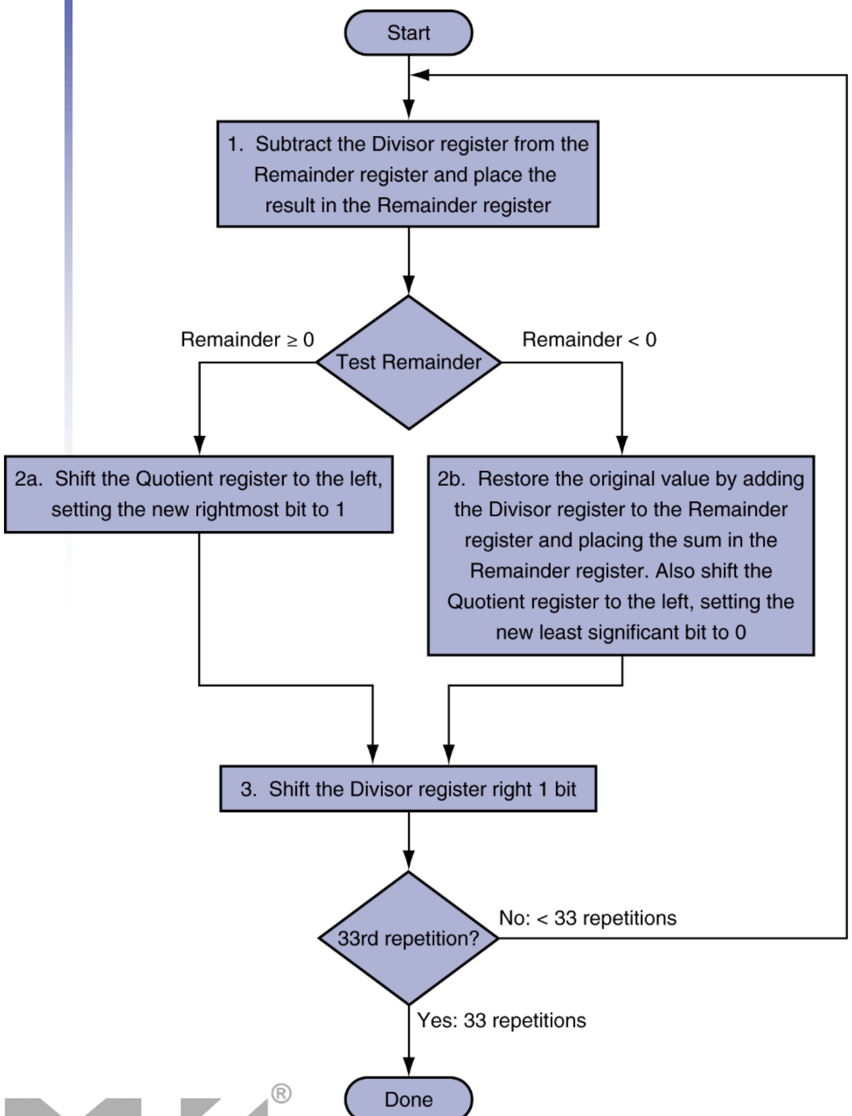
Division

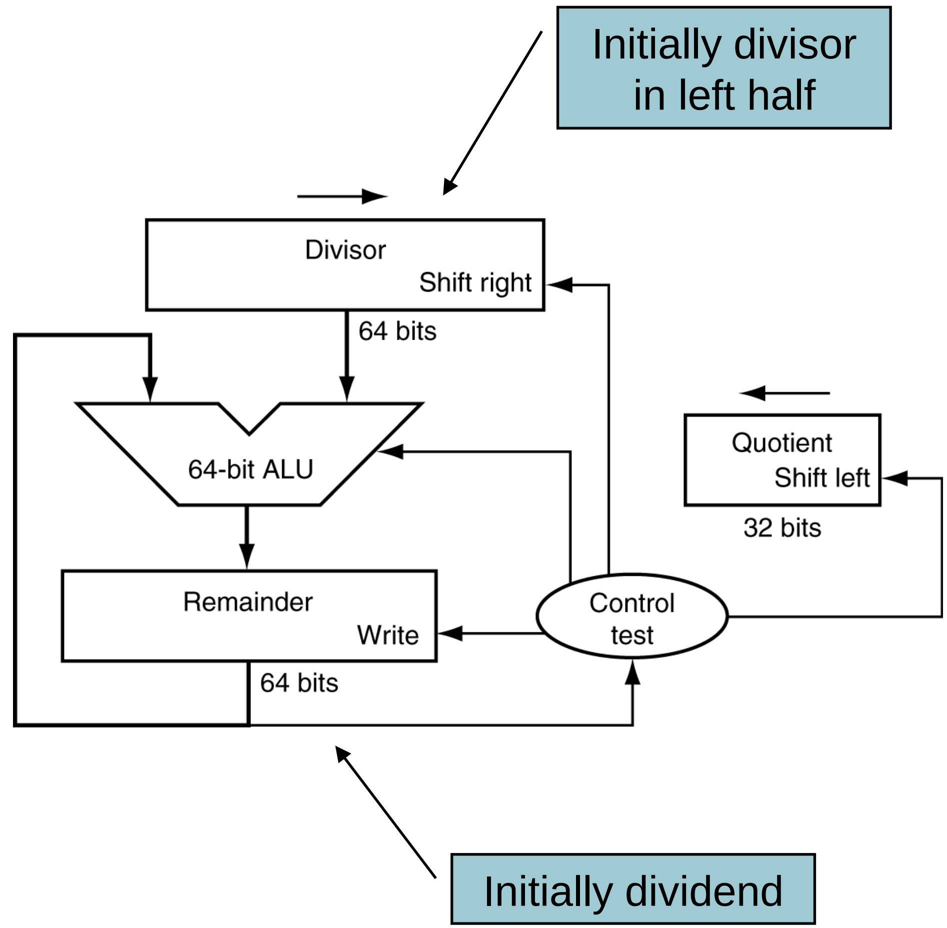


n-bit operands yield *n*-bit quotient and remainder

- 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

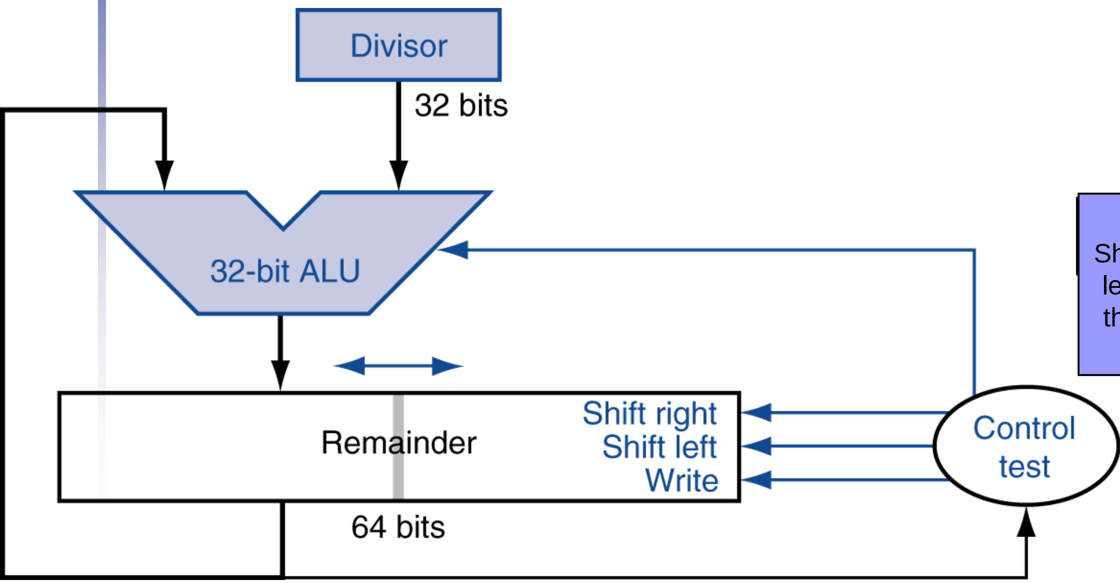


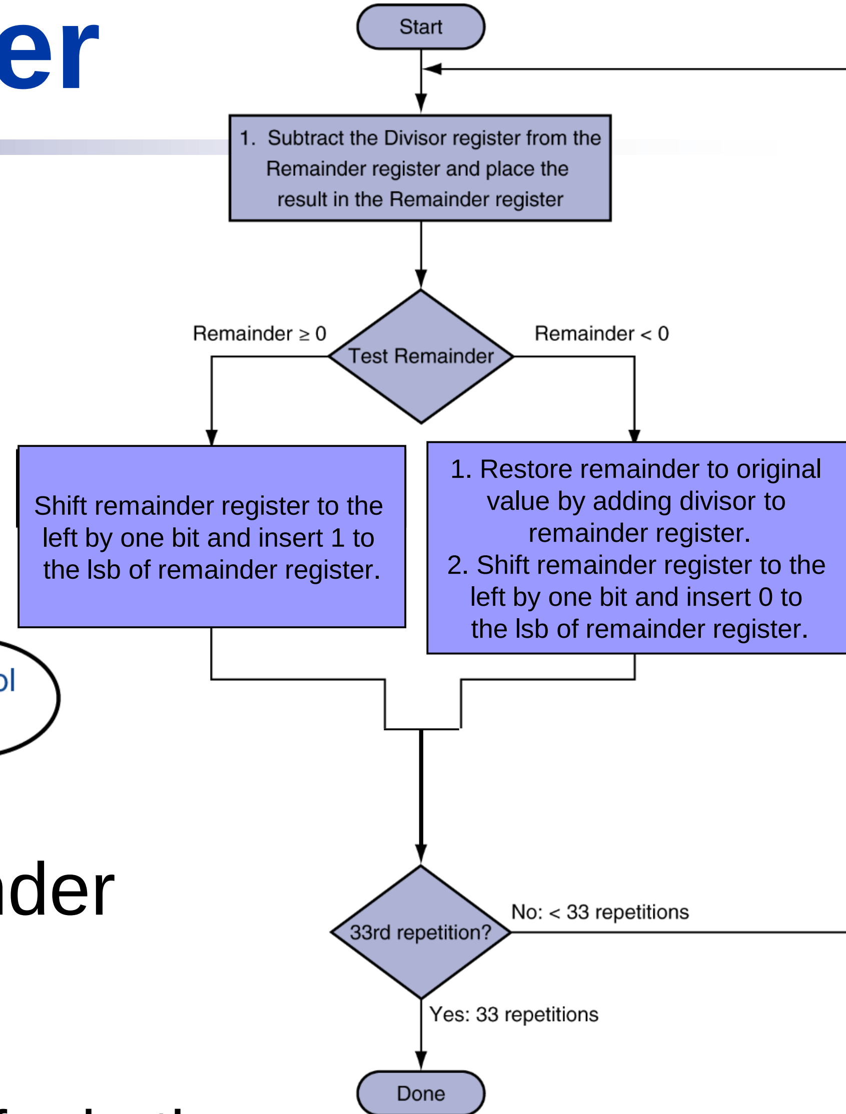


Division Example

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, QQ = 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, QQ = 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, QQ = 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, QQ = 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, QQ = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

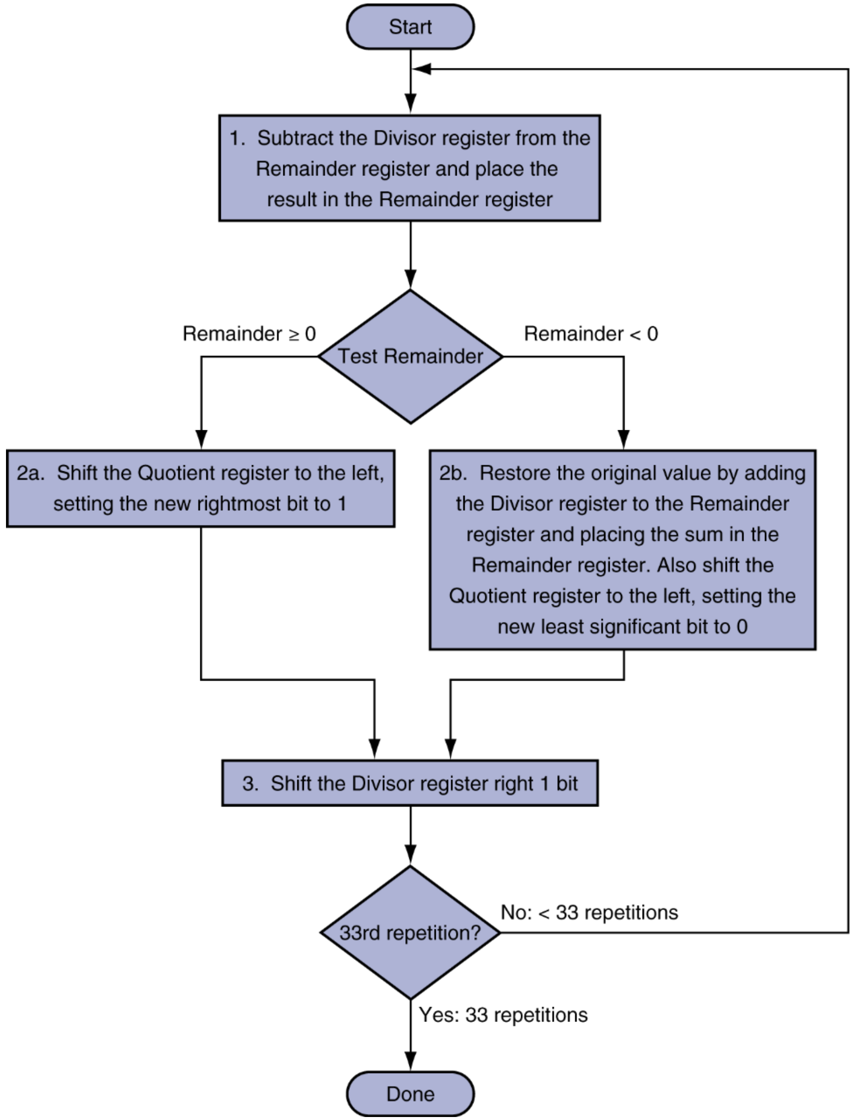
Optimized Divider





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

Restoring Algorithm



Iteration i　　　i + 1

r - d　　　2r-3d / 2r-d

r-d / r

2r-2d / 2r

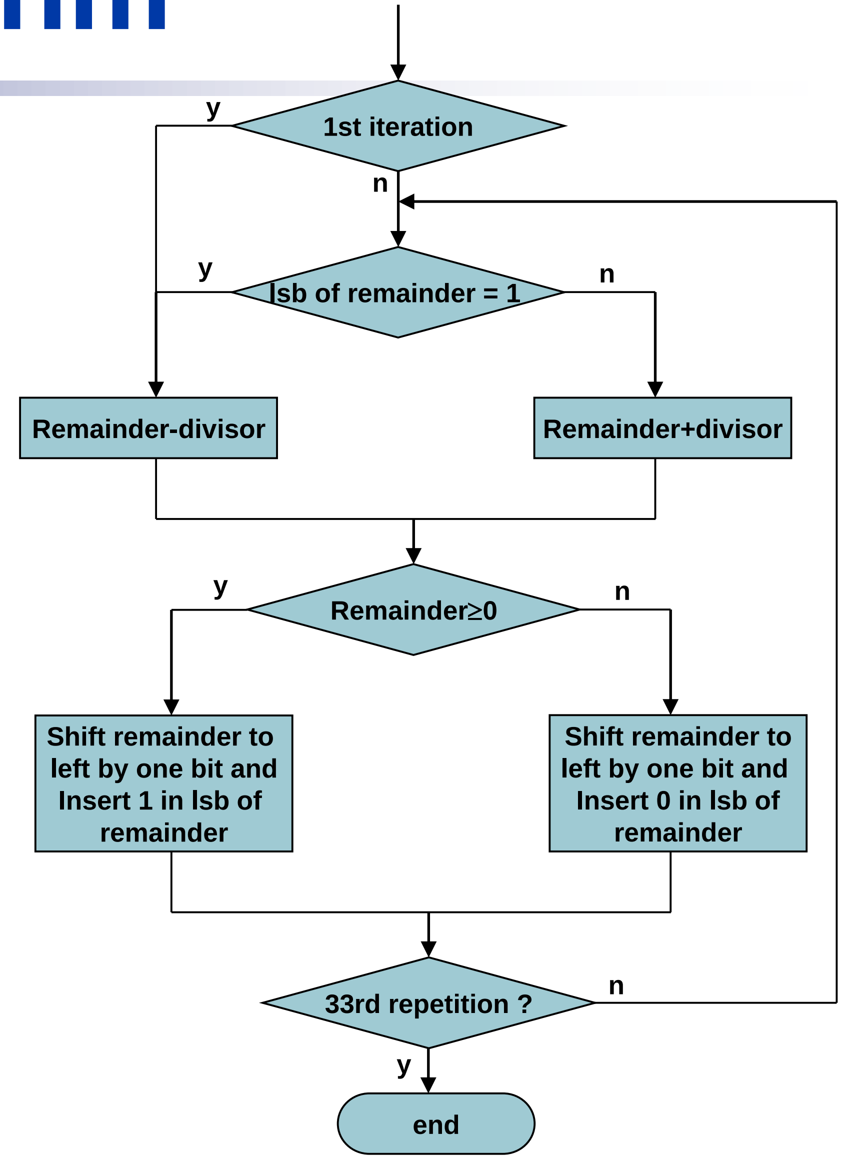
restore

Division Algorithm

- Restoring algorithm
  - r – d → quotient = 1
  - sll: 2r → quotient = 0
  next
    iteration: 2r – d (two subtractions
    and one addition)

- Non-restoring algorithm
  - r – d → quotient = 1
  - sll: 2(r – d) → quotient = 0
    next iteration: 2(r-d) + d
    = 2r – d (one subtraction
    and addition)
  - non-restoring flow is in the right

Iteration 1	Iteration 2
r-d	2r-3d/ 2r-d
2r-2d	



Faster Division

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

MIPS Division

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

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- 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 (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

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

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - 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: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

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

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

Floating-Point Precision

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

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: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal Numbers

- Exponent = 000...0 with fraction $\neq$ 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

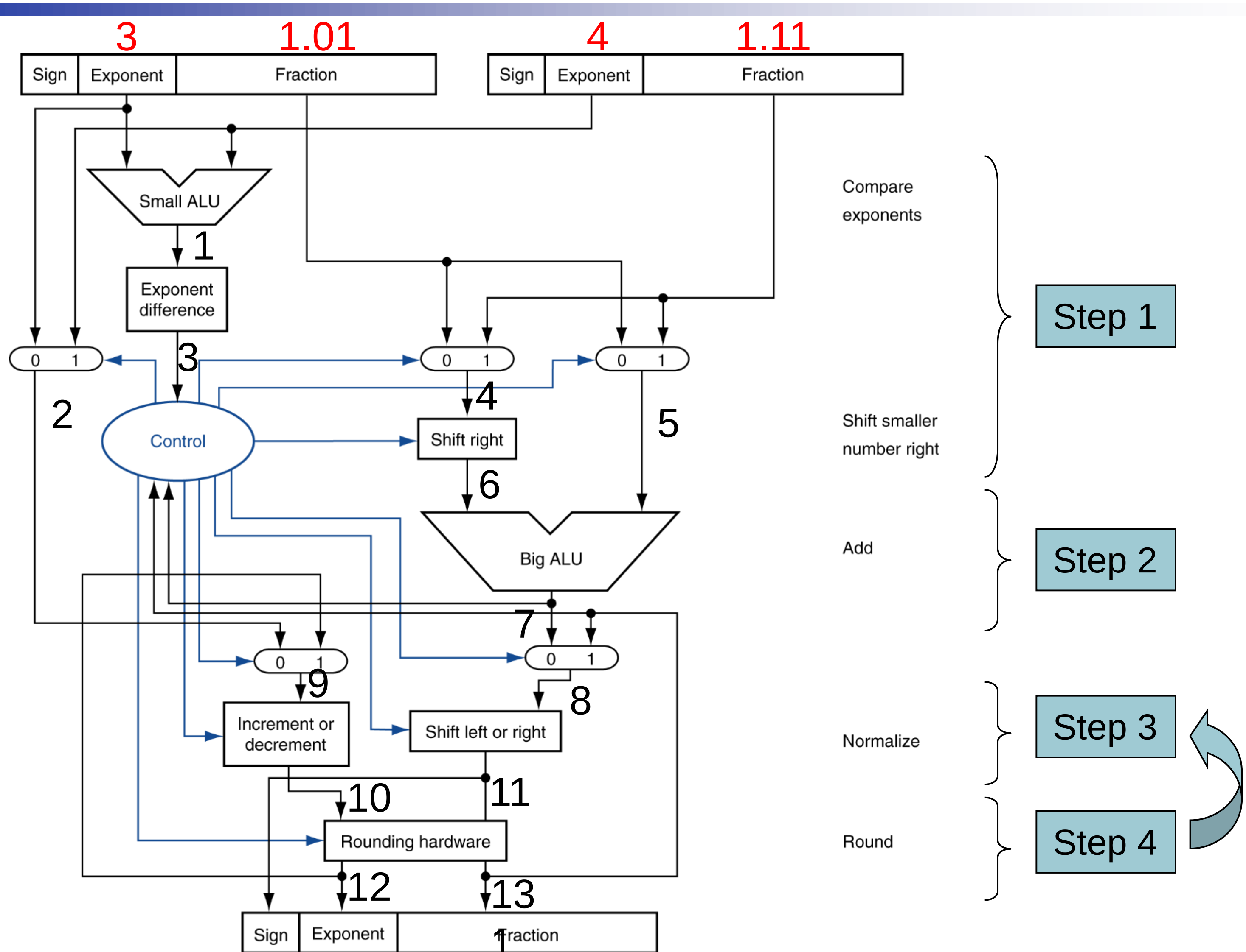
Floating-Point Addition

- 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
  - $1.000_2 \times 1.110_2 = 1.110_2 \;\Rightarrow\; 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve $\times$ –ve $\Rightarrow$ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: \$f0, \$f1, … \$f31
  - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is eq, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t 25`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1   $f16, const5($gp)
     lwc2   $f18, const9($gp)
     div.s  $f16, $f16, $f18
     lwc1   $f18, const32($gp)
     sub.s  $f18, $f12, $f18
     mul.s  $f0,  $f16, $f18
     jr     $ra
```

FP Example: Array Multiplication

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in \$a0, \$a1, \$a2, and `i`, `j`, k in \$s0, \$s1, \$s2

FP Example: Array Multiplication

- MIPS code:

```
     li    $t1, 32        # $t1 = 32 (row size/loop end)
     li    $s0, 0         # i = 0; initialize 1st for loop
L1:  li    $s1, 0         # j = 0; restart 2nd for loop
L2:  li    $s2, 0         # k = 0; restart 3rd for loop
     sll   $t2, $s0, 5    # $t2 = i * 32 (size of row of x)
     addu  $t2, $t2, $s1  # $t2 = i * size(row) + j
     sll   $t2, $t2, 3    # $t2 = byte offset of [i][j]
     addu  $t2, $a0, $t2  # $t2 = byte address of x[i][j]
     l.d   $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]
L3:  sll   $t0, $s2, 5    # $t0 = k * 32 (size of row of z)
     addu  $t0, $t0, $s1  # $t0 = k * size(row) + j
     sll   $t0, $t0, 3    # $t0 = byte offset of [k][j]
     addu  $t0, $a2, $t0  # $t0 = byte address of z[k][j]
     l.d   $f16, 0($t0)   # $f16 = 8 bytes of z[k][j]
```

…

FP Example: Array Multiplication

…

```
sll   $t0, $s0, 5        # $t0 = i*32 (size of row of y)
addu  $t0, $t0, $s2      # $t0 = i*size(row) + k
sll   $t0, $t0, 3        # $t0 = byte offset of [i][k]
addu  $t0, $a1, $t0      # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)       # $f18 = 8 bytes of y[i][k]
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16     # f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1        # $k k + 1
bne   $s2, $t1, L3       # if (k != 32) go to L3
s.d   $f4, 0($t2)        # x[i][j] = $f4
addiu $s1, $s1, 1        # $j = j + 1
bne   $s1, $t1, L2       # if (j != 32) go to L2
addiu $s0, $s0, 1        # $i = i + 1
bne   $s0, $t1, L1       # if (i != 32) go to L1
```

Accuracy of Floating-Point Operations

- Consider the addition: $2.56 \times 10^0 + 2.34 \times 10^2$

  $0.02\underline{56} \times 10^2 + 2.34\underline{00} \times 10^2 = 2.3656 \times 10^2 = 2.37 \times 10^2$

  Guard and round

  0.44 ulp (unit in the last place)

- If there are no guard and round extra bits, the result will be

  $0.02 \times 10^2 + 2.34 \times 10^2 = 2.36 \times 10^2$

- $2.345\boxed{0000000000}$ (2.34) vs. $2.345\boxed{0000000001}$ (2.35) by sticky bit (how to get?)

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

x86 FP Architecture

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

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

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

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - <u>S</u>ingle-<u>I</u>nstruction <u>M</u>ultiple-<u>D</u>ata

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 (MIPS cores and MIPS arithmetic cores): 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent