CS402

Chapter 3

Arithmetic for Computers

Dr. Michael Y Choi
Illinois Institute of Technology

Arithmetic for Computers

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

Numbers in computer

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: 0...2^n-1
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number

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 operands with different signs, no overflow
  - Adding two + (positive) operands
    - Overflow if result sign is 1
  - Adding two – (negative) operands
    - Overflow if result sign is 0

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 + (positive) or two – (negative) operands, no overflow
  - Subtracting + from – operand
    - Overflow if result sign is 0
  - Subtracting – from + operand
    - Overflow if result sign is 1

Binary calculation

- Just like in grade school (carry/borrow 1s)

```
  0111        0111          0110
+ 0110      - 0110        - 0101
------------------------------------
```

- Two's complement operations easy
  - subtraction using addition of negative numbers

```
  0110
+ 1011
------
1 0001
```

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
------
  1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

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 (interrupt) 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

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
  - Turning volume knob does not silent after the highest sound level

Multiplication

- Multiplicand – first operand
- Multiplier – second operand
- Product is a result of multiplication
- Suppose n-bit multiplicand and m-bit multiplier
- Product is n+m bits long

Multiplication

- Start with long-multiplication approach

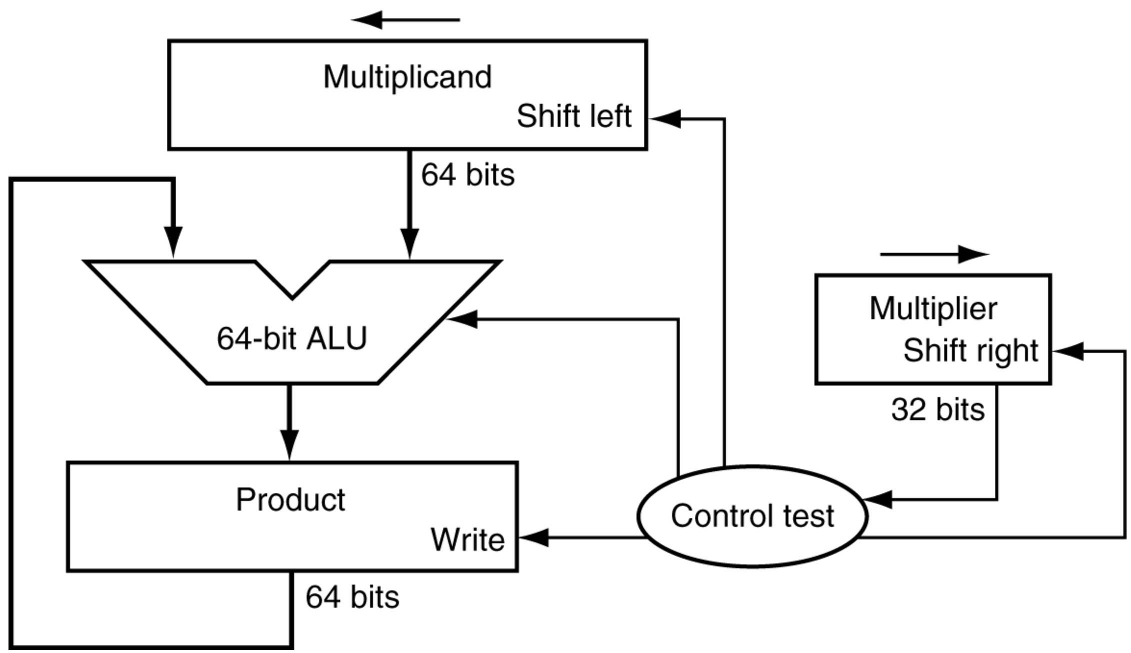
multiplicand → 1000

multiplier → × 1001

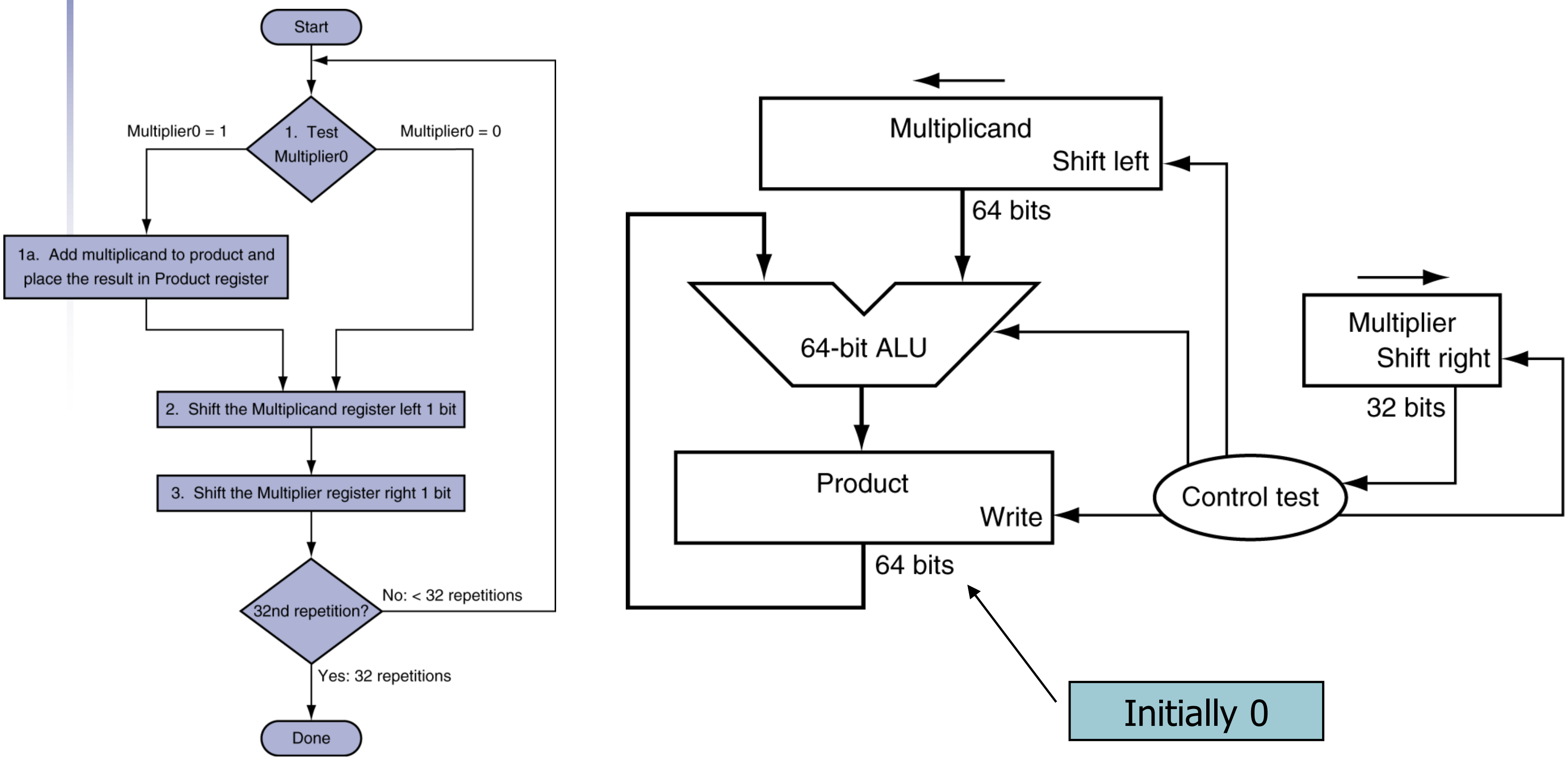
```
   1000
 × 1001
 ------
   1000
  0000
 0000
1000
-------
1001000
```

product → 1001000

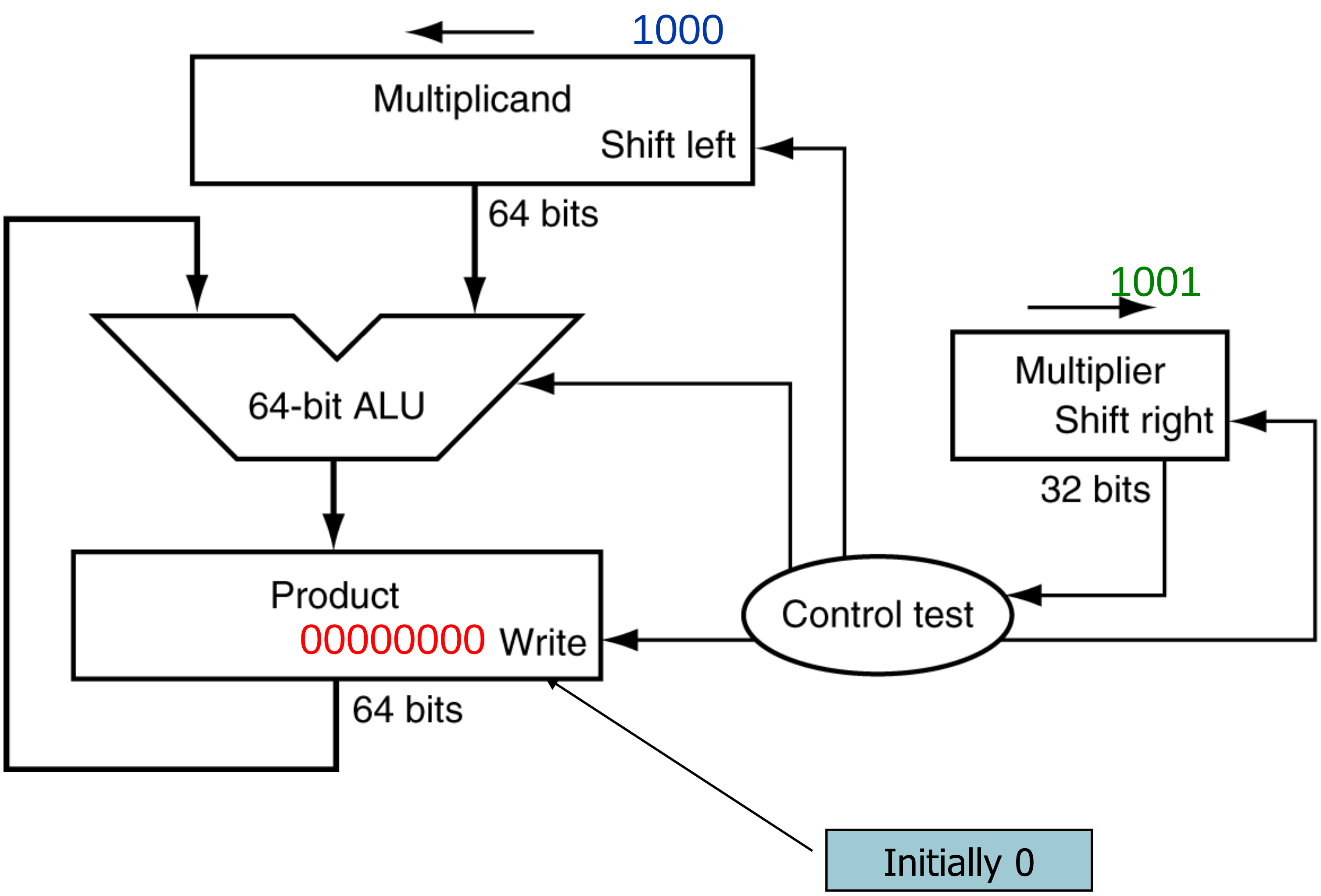
Length of product is the sum of operand lengths



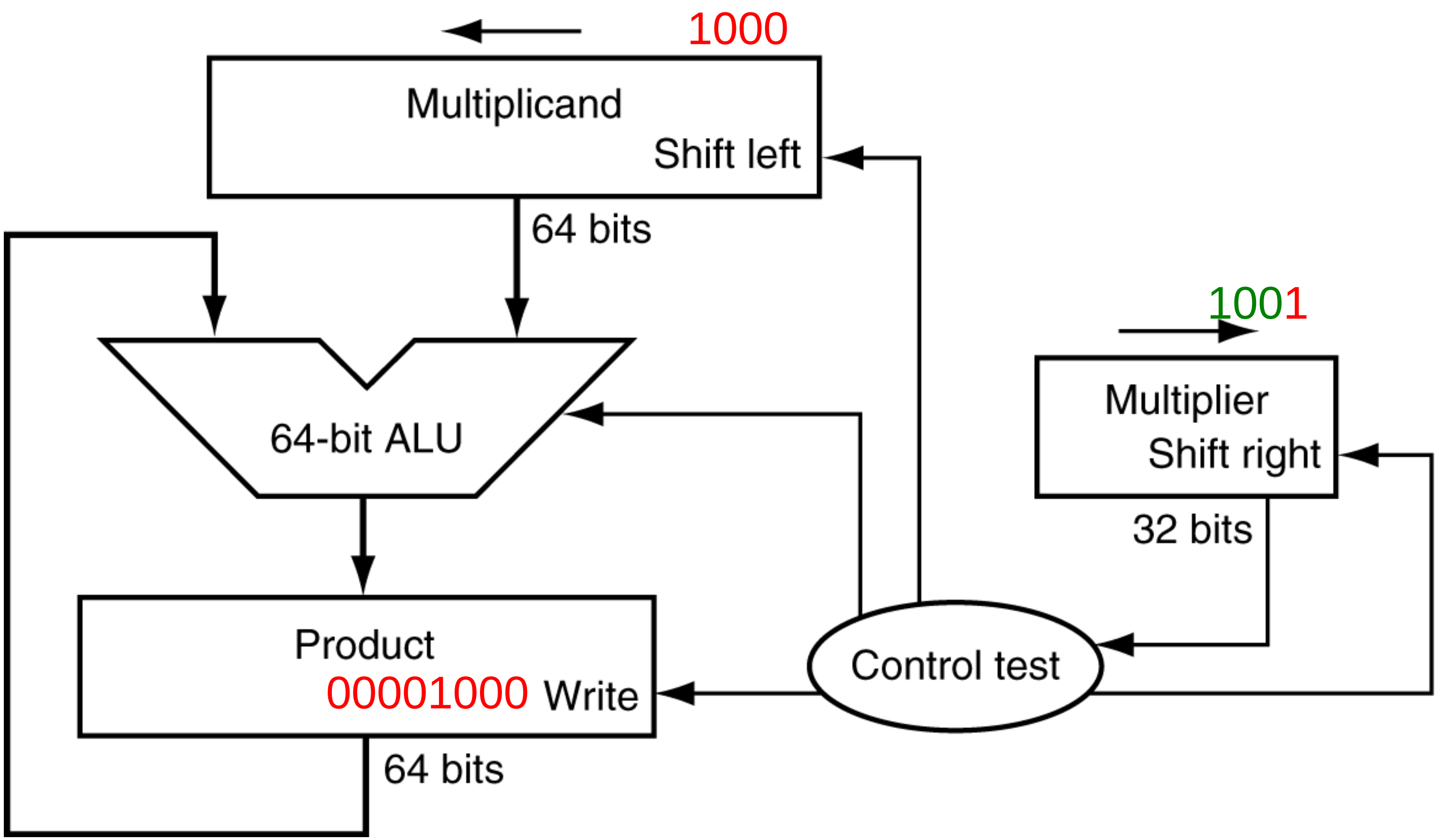
Multiplication Hardware



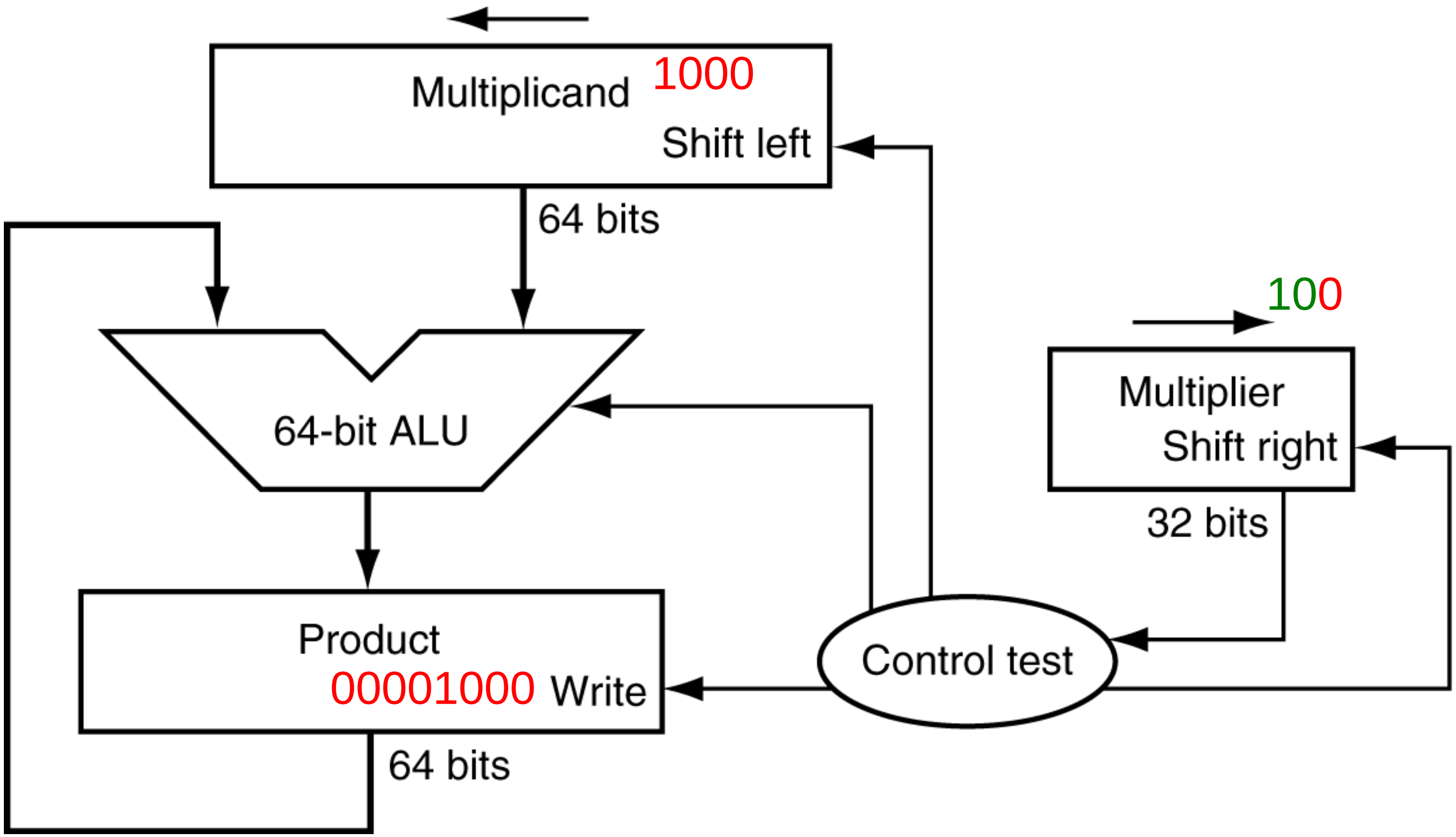
Multiplication Hardware



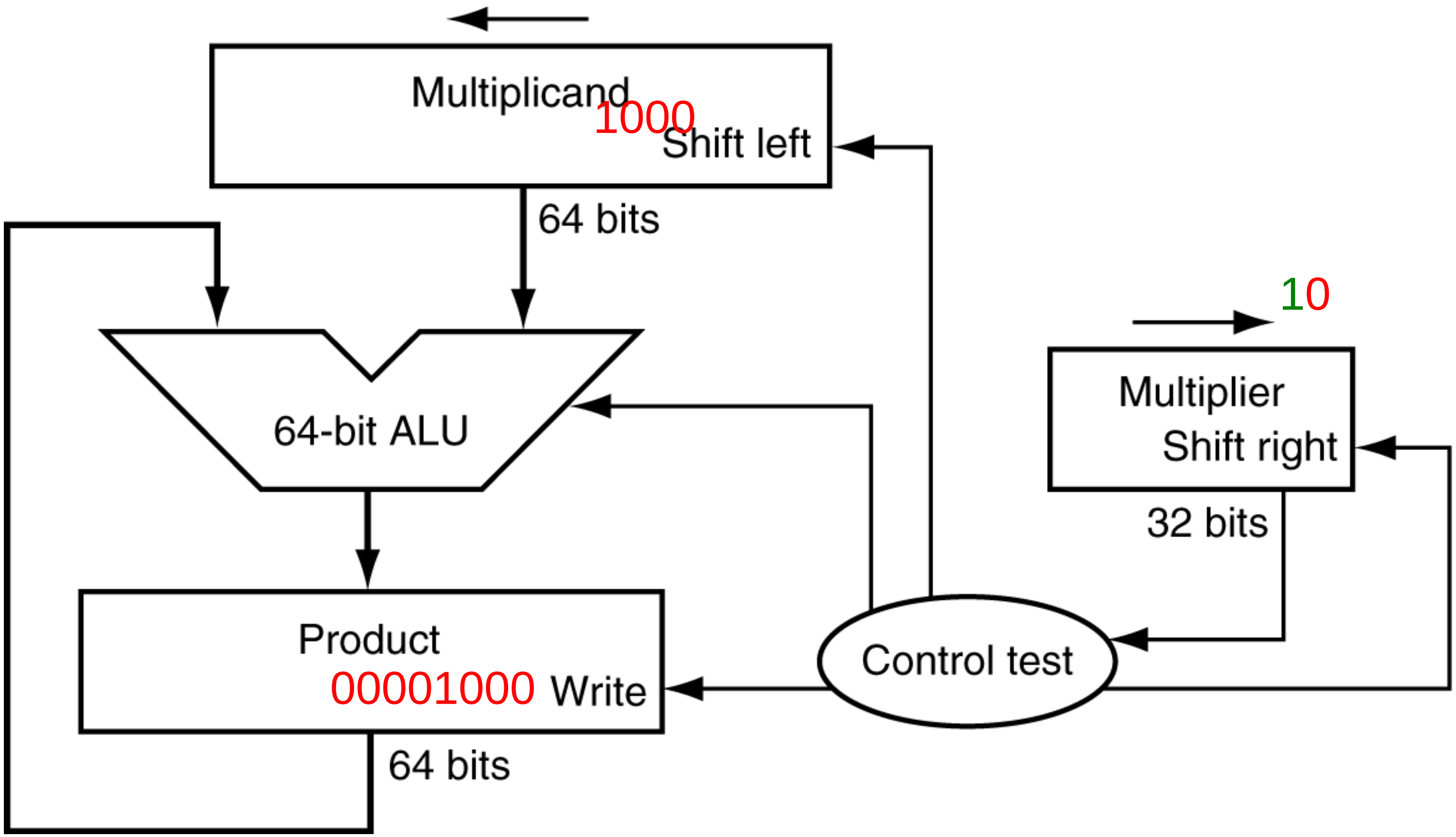
Multiplication Hardware



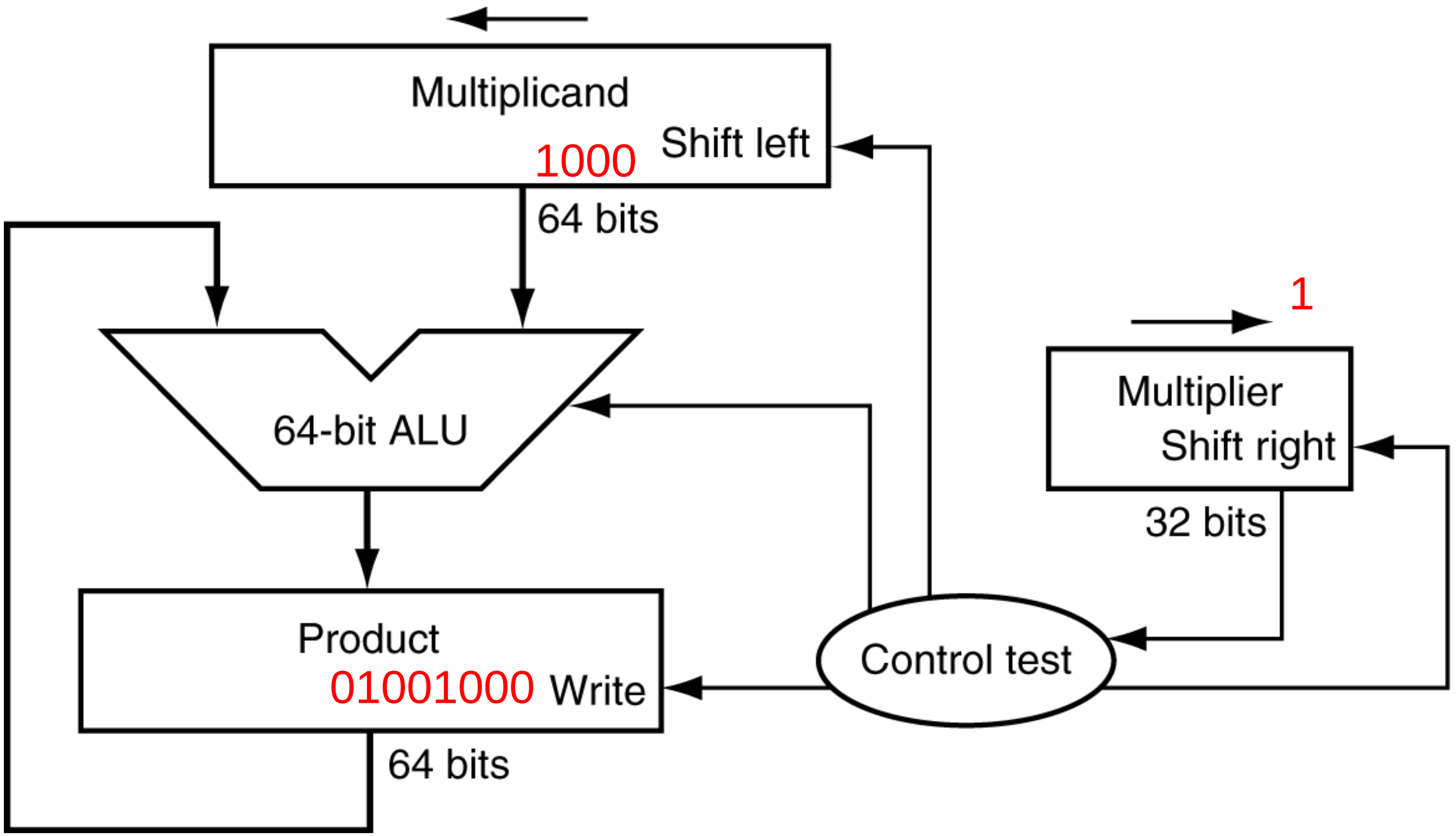
Multiplication Hardware



Multiplication Hardware



Multiplication Hardware

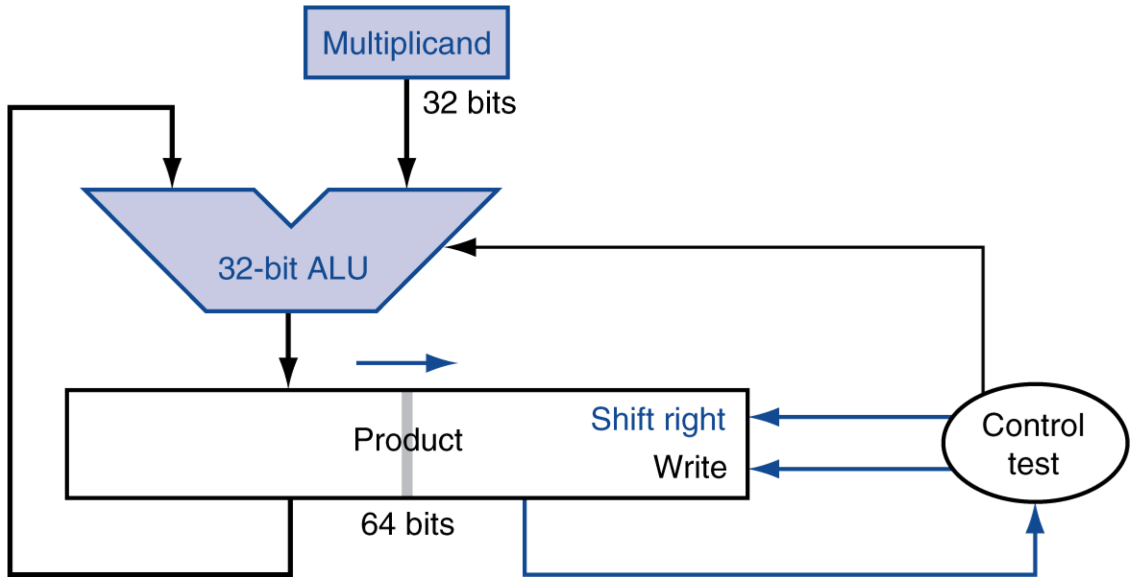


Multiplication Hardware

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Optimized Multiplier

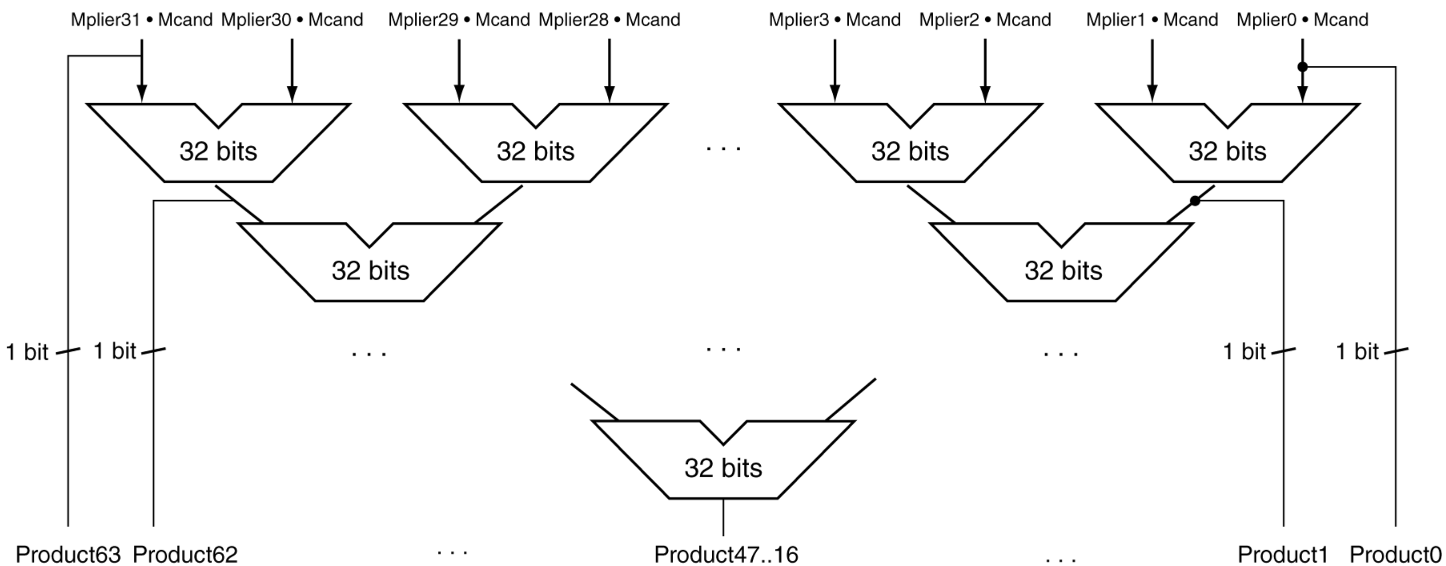
- Perform steps in parallel: add/shift



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

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

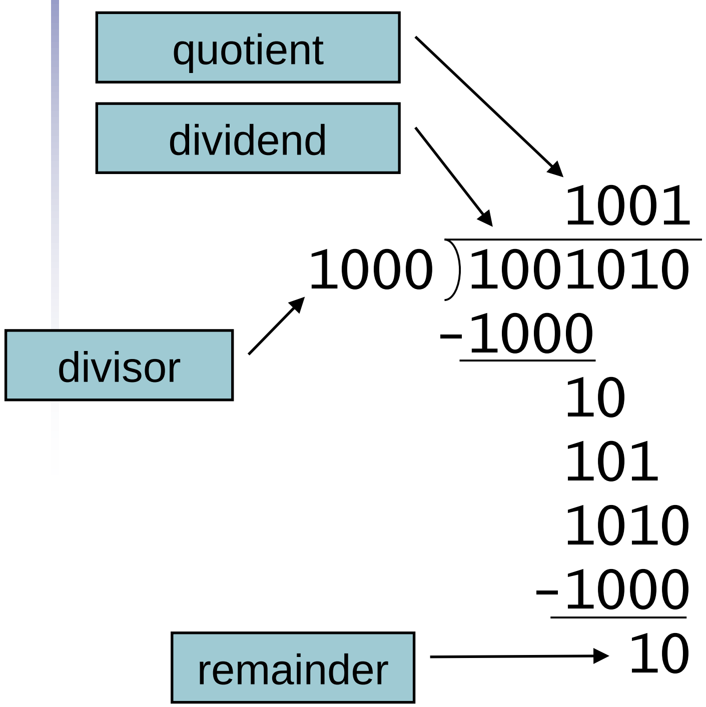


- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

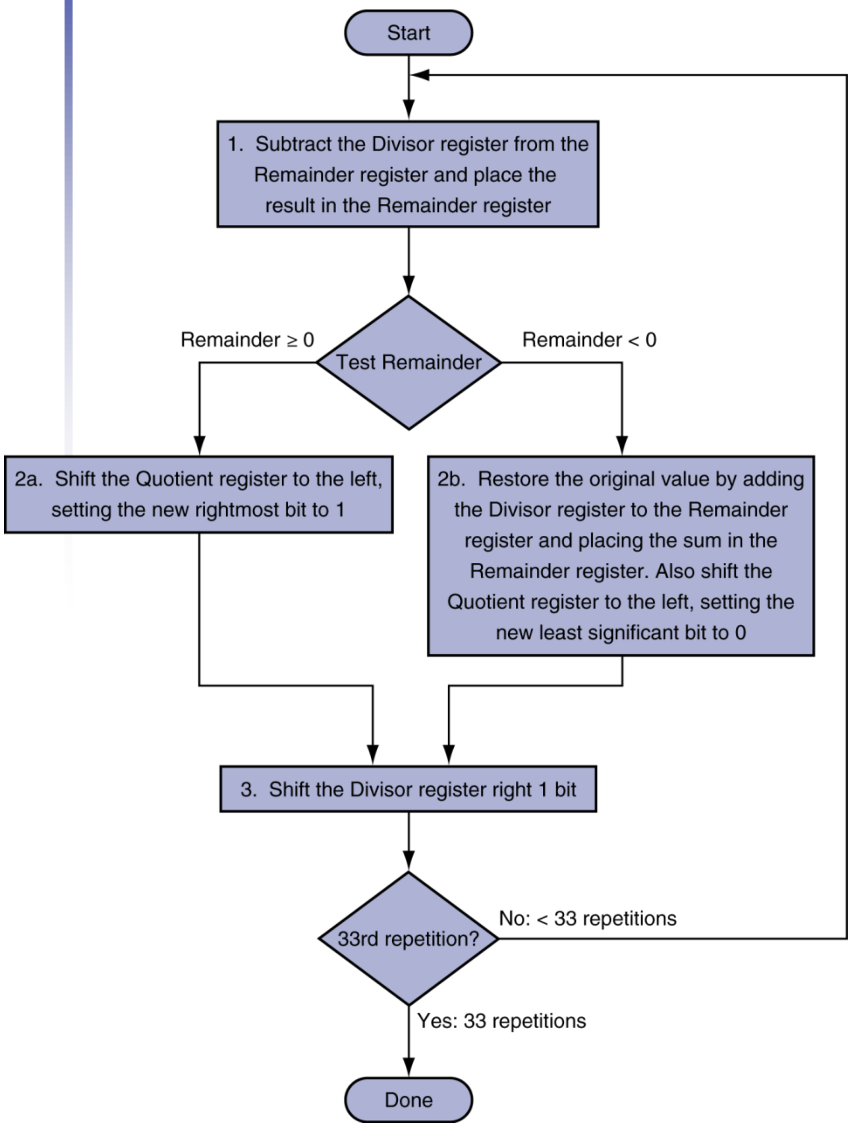
Division

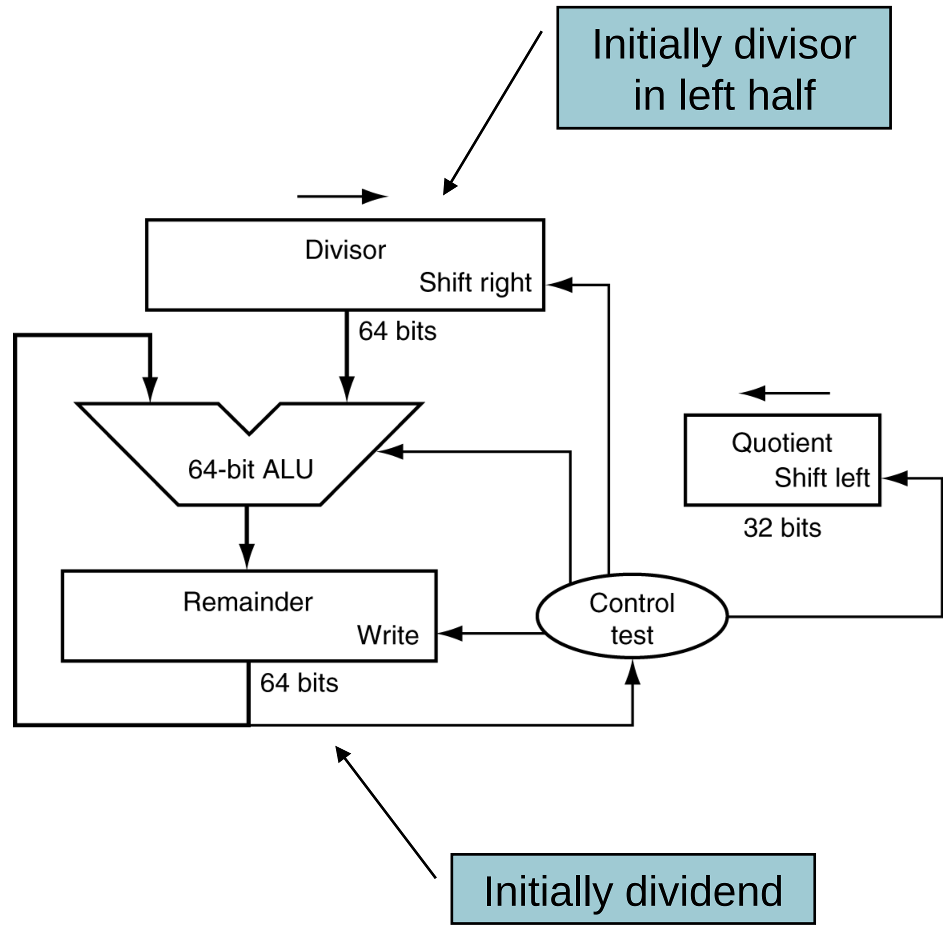


n-bit operands yield *n*-bit quotient and remainder

- 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



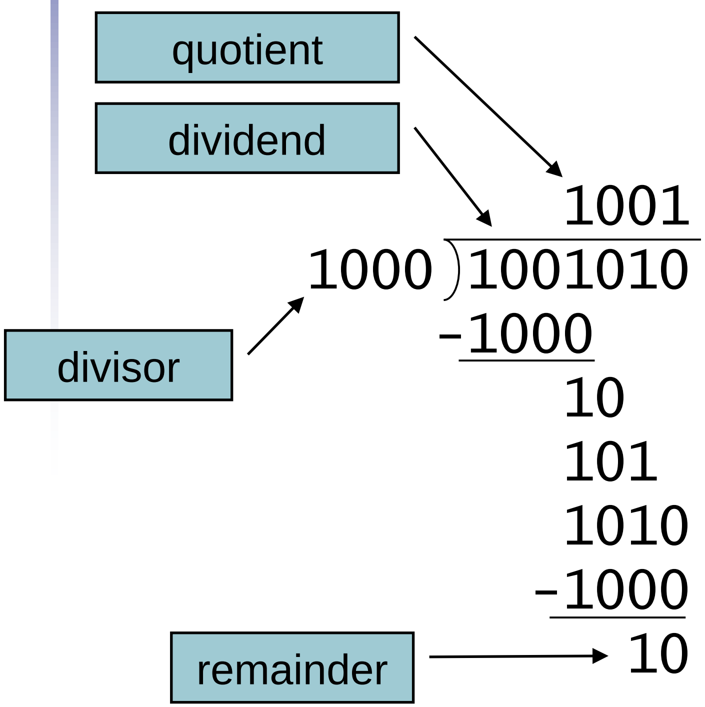


Division Hardware

Why restore by +Div multiple times?
Because negative remainder means the remainder is still not enough value to subtract divisor → to make enough digits

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 $\Longrightarrow$ +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 $\Longrightarrow$ +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 $\Longrightarrow$ +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 $\Longrightarrow$ 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 $\Longrightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

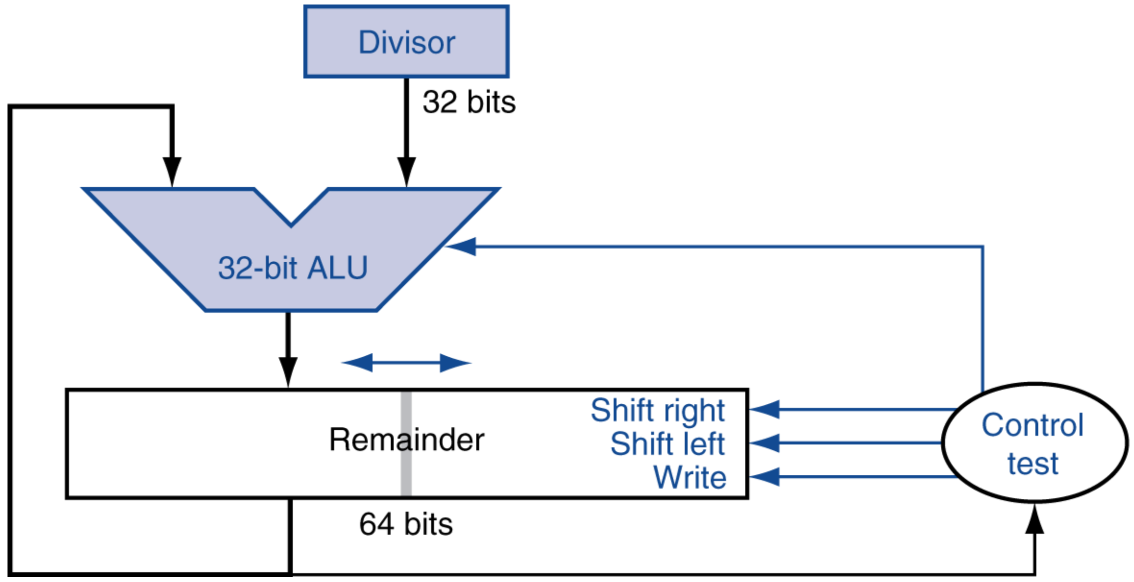
Division - review



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

Optimized Divider



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

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

Sweeney, Robertson, and Tocher

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

Move from hi, move from low

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Example of real numbers:
  - -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

- Scientific notation – single digit to the left of the decimal point
- Normalized number – a scientific notation with no leading zero
- Non-normalized number – vice versa
  - 1.0×10^9 Normalized
  - 0.1×10^{-4} Non-normalized
  - 10.0×10^9 Non-normalized

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) e.g. float
  - Double precision (64-bit)   e.g. double

IEEE Floating-Point Format

single: 8 bits single: 23 bits → **31 bits**
double: 11 bits double: 52 bits → **63 bits**

S	Exponent	Fraction

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

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ 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 = 1203

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 $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

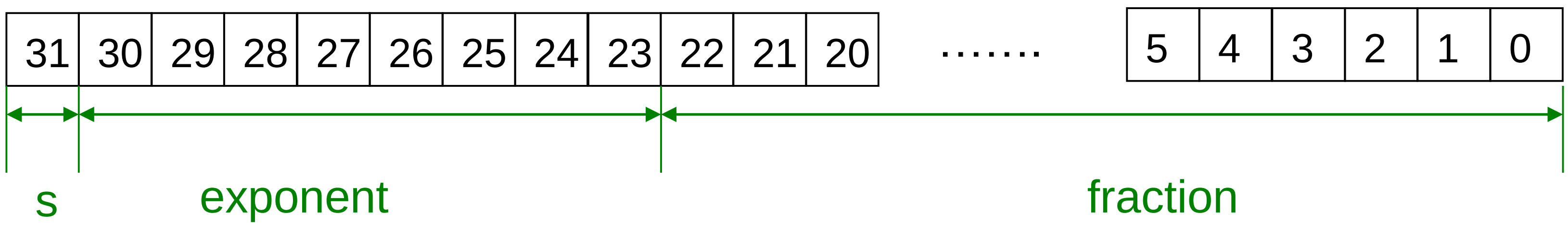
Single-Precision Range

31	30	29	28	27	26	25	24	23	22	21	20	…….	5	4	3	2	1	0
s	exponent								fraction									

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

- 0 00000001 00000000…..000000000
  $\Rightarrow$ actual exponent = 1 – 127 = –126
- Fraction: 000…00 $\Rightarrow$ significand = 1.0
- $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

Single-Precision Range



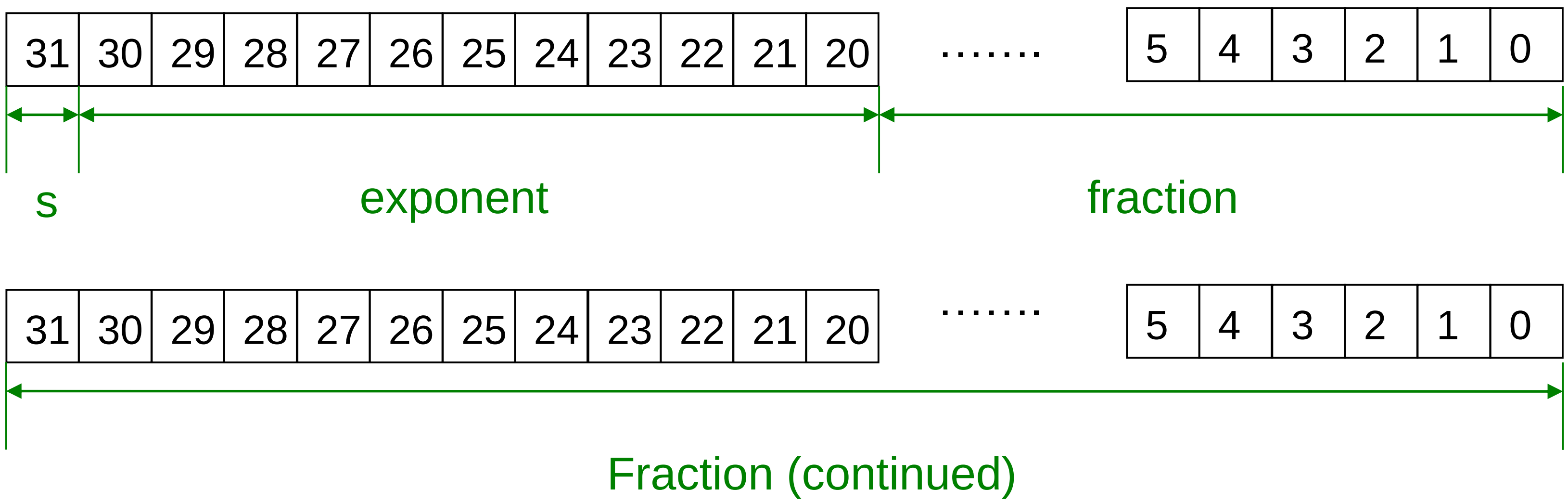
$$x = (-1)^S \times \boxed{(1+\text{Fraction})} \times 2^{\boxed{(\text{Exponent}-\text{Bias})}}$$

- 0 11111110 11111111…..111111111
  $\Rightarrow$ actual exponent = 254 – 127 = 127
- Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
- $\pm 2.0 \times 2^{127} \approx \pm 3.4 \times 10^{38}$

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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Double-Precision Range



- 0000000010000000.....000000000
  $\Rightarrow$ actual exponent = 1 – 1023 = –1022
- Fraction: 000…00 $\Rightarrow$ significand = 1.0
- $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

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
  - $-0.75 = (-1)^1 \times \boxed{1.1_2} \times 2^{-1}$ $= 1.5_{10}$
  - 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 = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

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

  $= -5.0$

Floating-Point Example

- A decimal value 0.75,
  - $0.75 = 1.0 \times 0 + 0.5 + 0.25$

    $= 1 \times 2^{-1} + 1 \times 2^{-2}$

    $= 0.11_2$

  Convert to scientific notation: $0.11_2 \times 2^0$

  $= 1.1_2 \times 2^{-1}$

Binary non whole number

- In decimal, **3.75 and 3 . and 3 $^{75}/_{100}$ all represent the same number**
- • When using a decimal point, positions to the right of the decimal point indicate increasingly negative powers of 10: 10^{-1}, 10^{-2}, ….
- • **Example: $3.75 = 3 \cdot 10^{0} + 7 \cdot 10^{-1} + 5 \cdot 10^{-2}$**
- • Dividing by 10*n shifts the decimal point n digits to the left.*
- • **Example: 0.75 = 75 / 100, so 3.75 = 3 $^{75}/_{100}$ = 3 3/4**

Binary non whole number

- In binary, the positions to the right of the binary point indicate negative powers of 2.
- • **Example : $1.011_2 = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}$**
- $= 1 + 1/_4 + 1/_8 = 1\ 3/_8 = 1.375_{10}$
- • Dividing by 2*n shifts the binary point n bits left; multiplying by 2n shifts right.*
- • **Example: $1.011_2 = (1011/1000)_2 = (11/_8)_{10} = 13/_8$**

Binary non whole number

- **Example:**

- **$1.375 = 1 + 0.375 = 1 + 0 \cdot 0.5 + 0.375$**

  $= 1 + 0 \cdot 0.5 + 1 \cdot .25 + 1 \cdot 0.125$

  $= 1.011_2$

Decimal to Floating Point

- Bias selection uses: $2^{k-1} - 1$
- With 8 bit number format, k = 3, so bias = 3
- IEEE 32bit number format, k = 8, so bias = 127

- Example 1: Convert 2.625 to 8 bit FP format

(Do it now…)

Decimal to Floating Point

- Example 1: Convert 2.625 to 8 bit FP format
  (Approach 1)

Decimal to Floating Point

- Example 1: Convert 2.625 to 8 bit FP format

Decimal to Floating Point

- Example 2: Convert -4.75 to 8 bit FP format

(Do it now…)

Decimal to Floating Point

- Example 2: Convert -4.75 to 8 bit FP format

Decimal to Floating Point

- Example 3: Convert 12.0 to 8 bit FP format

(Do it now…)

Decimal to Floating Point

- Example 3: Convert 12.0 to 8 bit FP format

Decimal to Floating Point

- Example 4: Convert 1.7 to 8 bit FP format

(Do it now…)

Decimal to Floating Point

- Example 4: Convert 1.7 to 8 bit FP format

Ex 4. continue

- Or an alternative verification…
  - 2^{-1} = ½ = 0.5
  - 2^{-2} = ¼ = 0.25
  - 2^{-3} = 1/8 = 0.125
  - 2^{-4} = 1/16 = 0.0625
  - 2^{-5} = 1/32 = 0.03125
  - 2^{-6} = 1/64 = 0.015625
  - 2^{-7} = 1/128 = 0.0078125

So 0.7 = 0.5 + 0.125 + 0.0625 + 0.0078125 + …

Why endless? 0.7 = 7/10 so it repeats fraction like 1/3.

Ex 4. continue

- Example 4: Convert 1.7 to 8 bit FP format

$1 = 1_2$

$0.7 \times 2 = \boxed{1}.4$ ← Generate 1 and continue with rest

$0.4 \times 2 = \boxed{0}.8$ ← Generate 0 and continue with rest

$0.8 \times 2 = \boxed{1}.6$ ← Generate 1 and continue with rest

$0.6 \times 2 = \boxed{1}.2$ ← Generate 1 and continue with rest

....

Choose only proper bits for fraction.

Fraction: 1011 Normalized: $1.1011_2 \times 2^0$

Exponent: $K - 3 = 0$, so $K = 3 = 011_2$

Sign: 0

The result is 00111011 and $3b_{hex}$

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

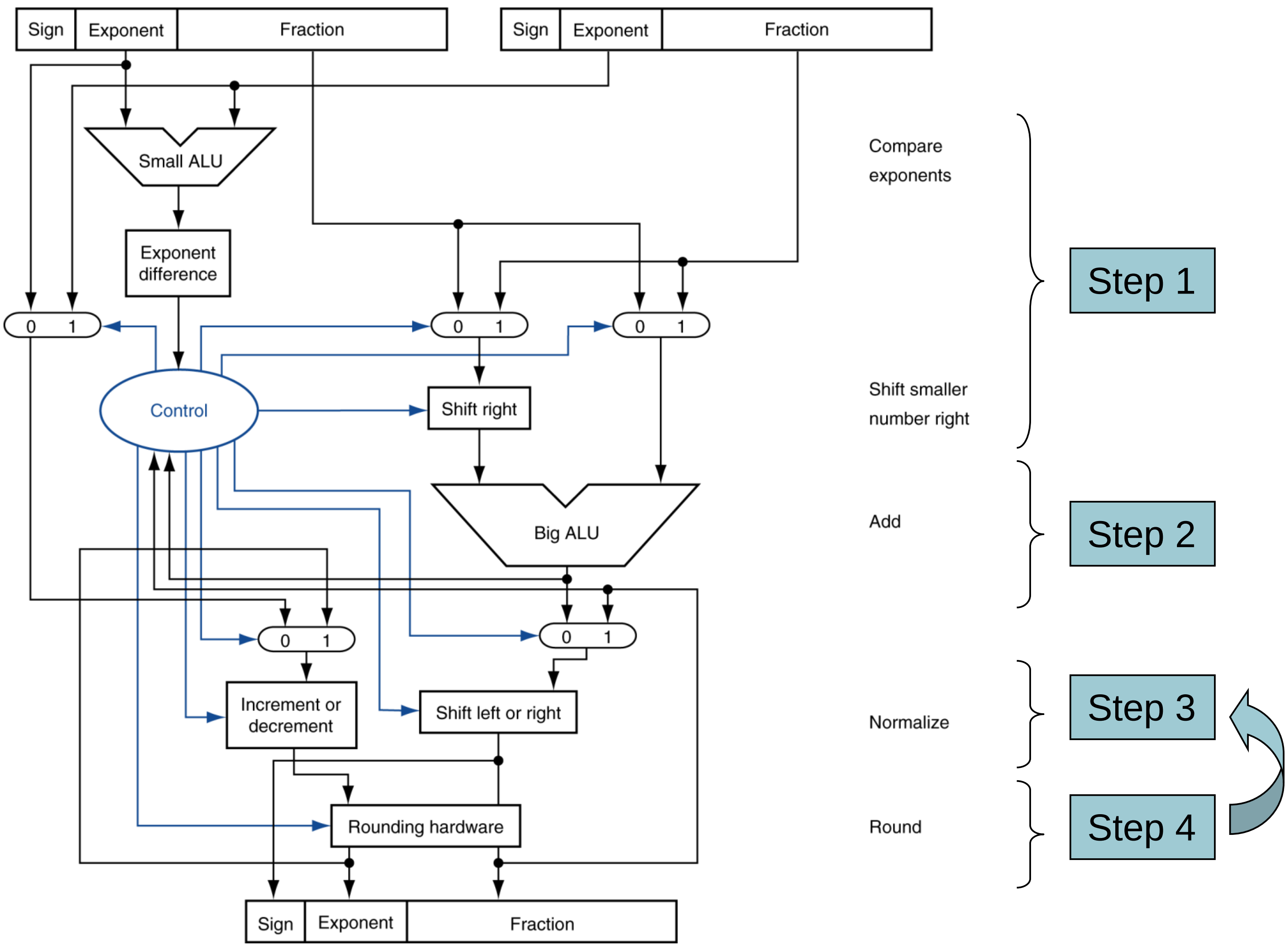
$$
\begin{aligned}
-0.111_2 &= 1.000_2 + 1_2 \\
&= 1.001_2 \\
& \quad 1.000_2 \\
+ & \quad 1.001_2 \\
= & \cancel{1}0.001_2 = 0.001_2
\end{aligned}
$$

FP Adder Hardware

Skip HW issue

- 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



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 TargetLabel`

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
```

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: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent