Arithmetic for Computers

Sections 3.1-3.4

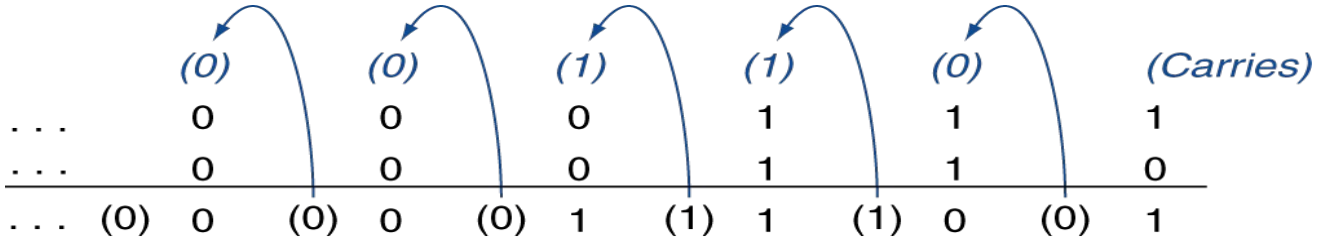
Objectives

- To understand how does hardware multiply or divide numbers
- To understand floating number representation, SW and HW implementation

Arithmetic Overflow

Example: 7 + 6

Overflow: the result of an operation cannot be represented by the rightmost hardware bits



- 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

Subtraction Rules

Unsigned Rules

2's complemnt signed representation simplifies HW design

Handling Overflow

MIPS add, addi, sub instructions raise an ***exception***

- On overflow, invoke ***exception handler***
  - Save PC in **exception program counter (EPC)** register
  - Jump to predefined handler address (KERNEL code)
  - mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action
  - More on that in Lab #5

MIPS addu, subu, … instructions does not generate exception

- You need to have additional code to handle overflow in this case if you expect it.

Exception handler Routine (OS) Kernel code

Power fault → code

.

.

.

Divide by zero → handling code

Overflow → handling code

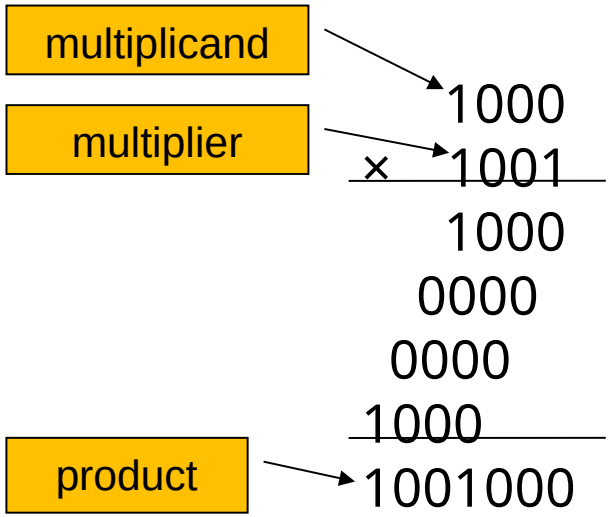
.

.

Integer Multiplication

Multiplication

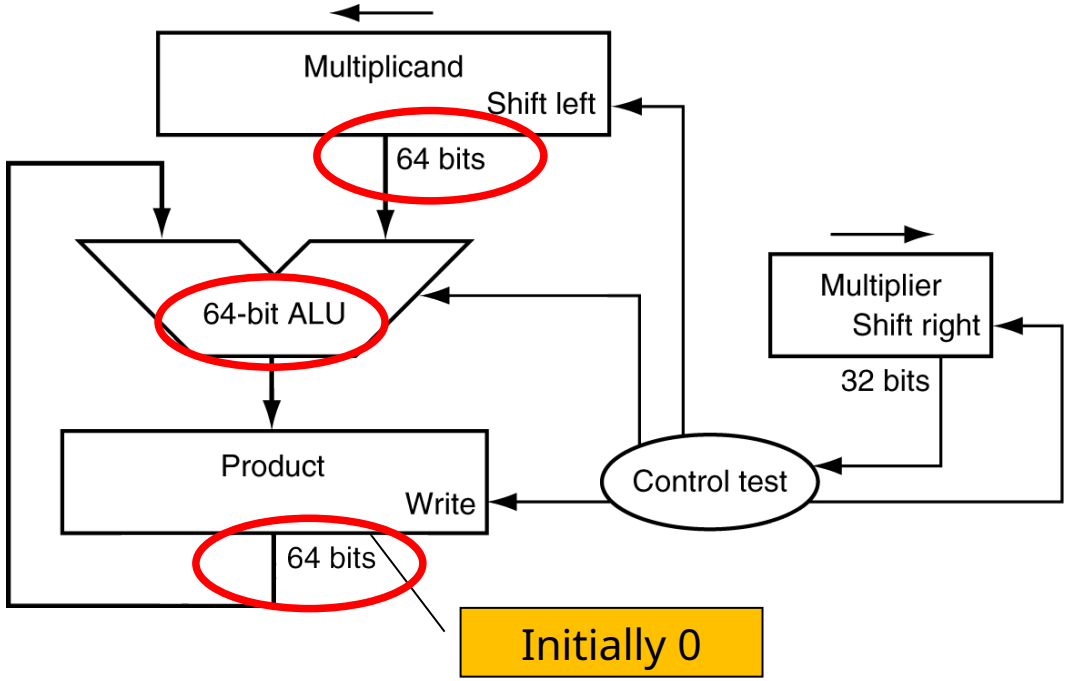
- Example: Multiply 1000 by 1001
- Start with long-multiplication approach

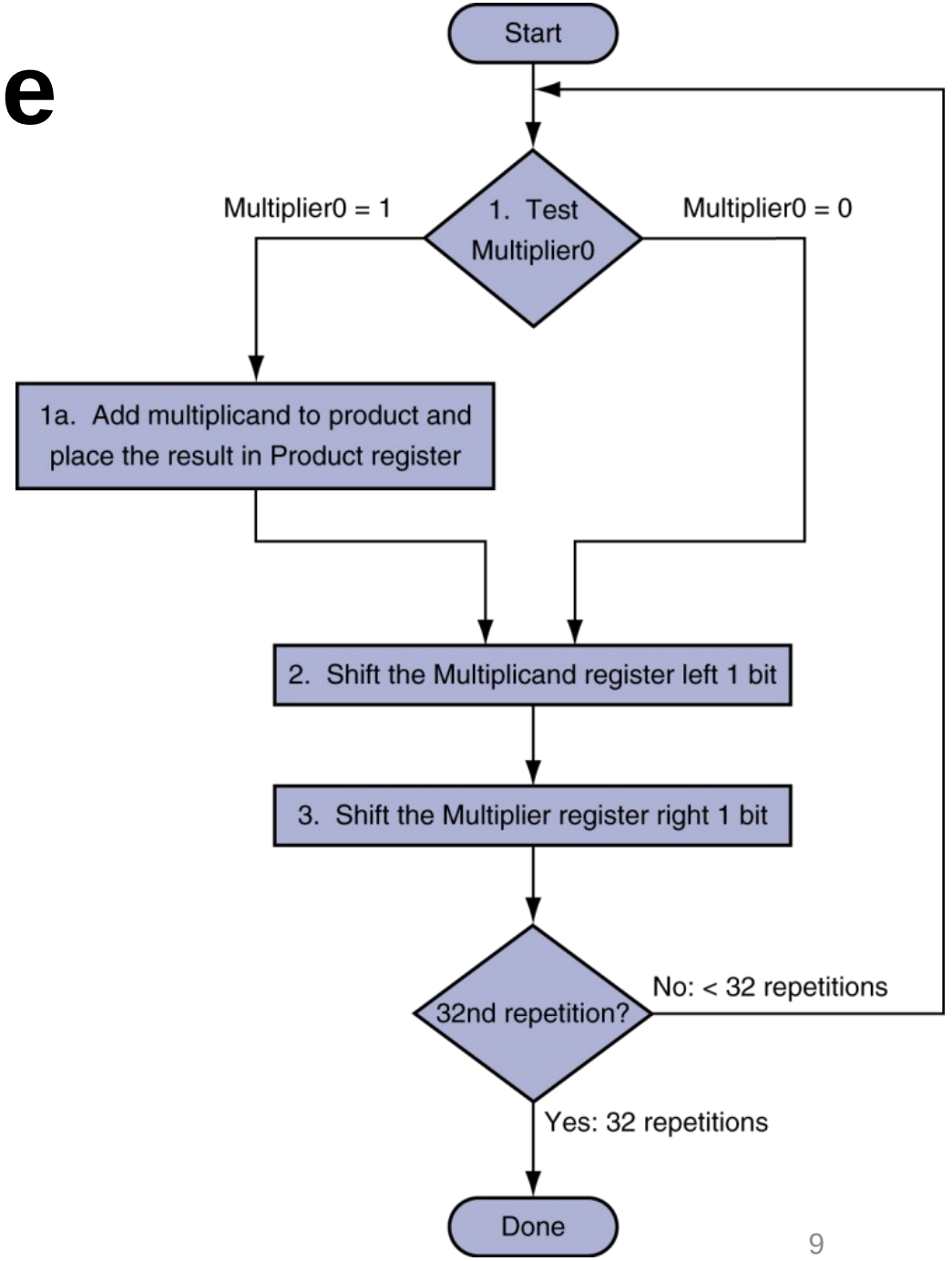


Length of product is the sum of operand lengths

Multiplication can be calculated as ***addition of shifted versions of the multiplicand***

Multiplication Hardware Implementation (1)





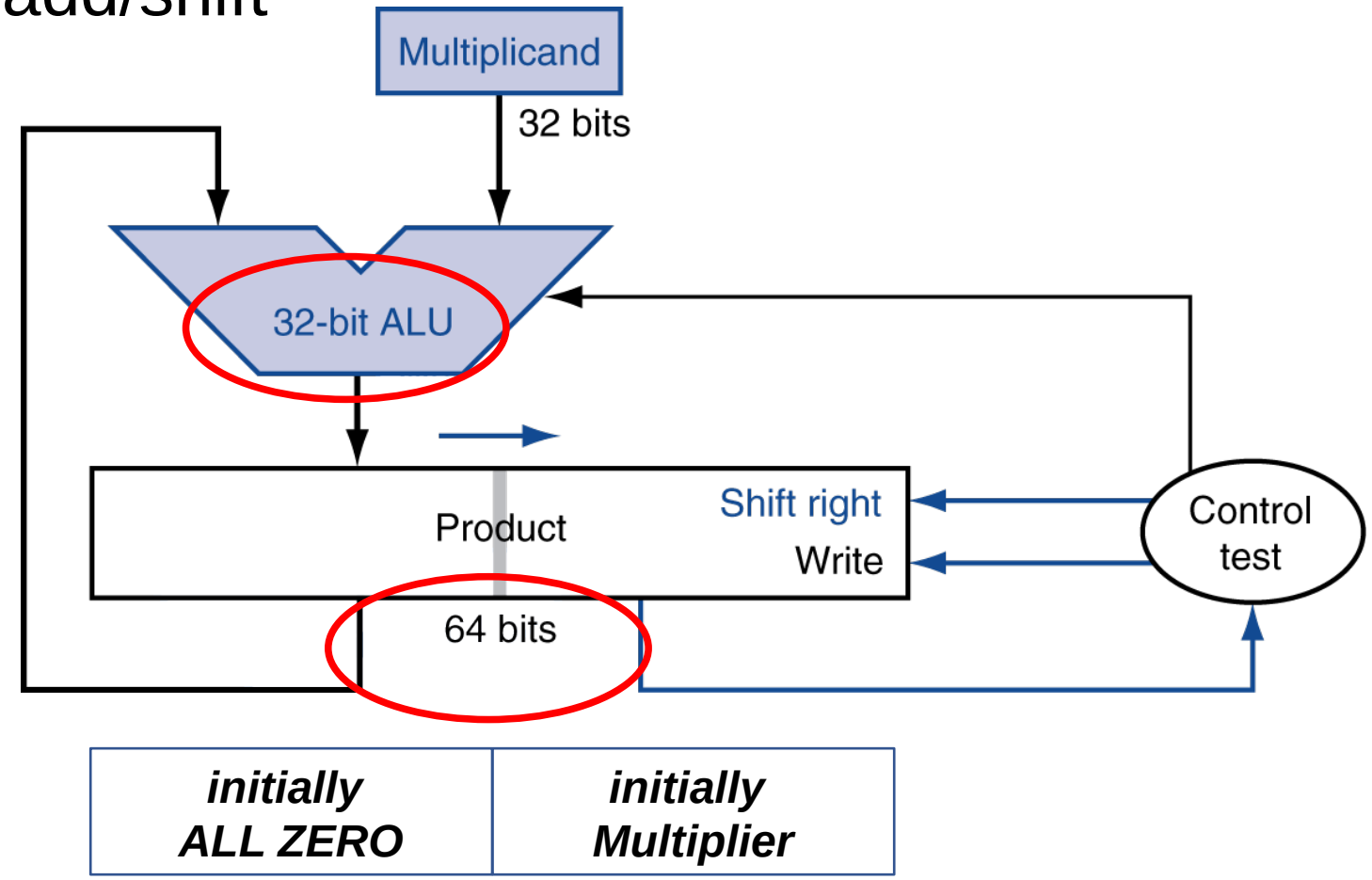
2 x 64 bit registers & **64-bit ALU**

32 clock cycles to complete the multiplication operation

Optimised Multiplier Implementation (2)

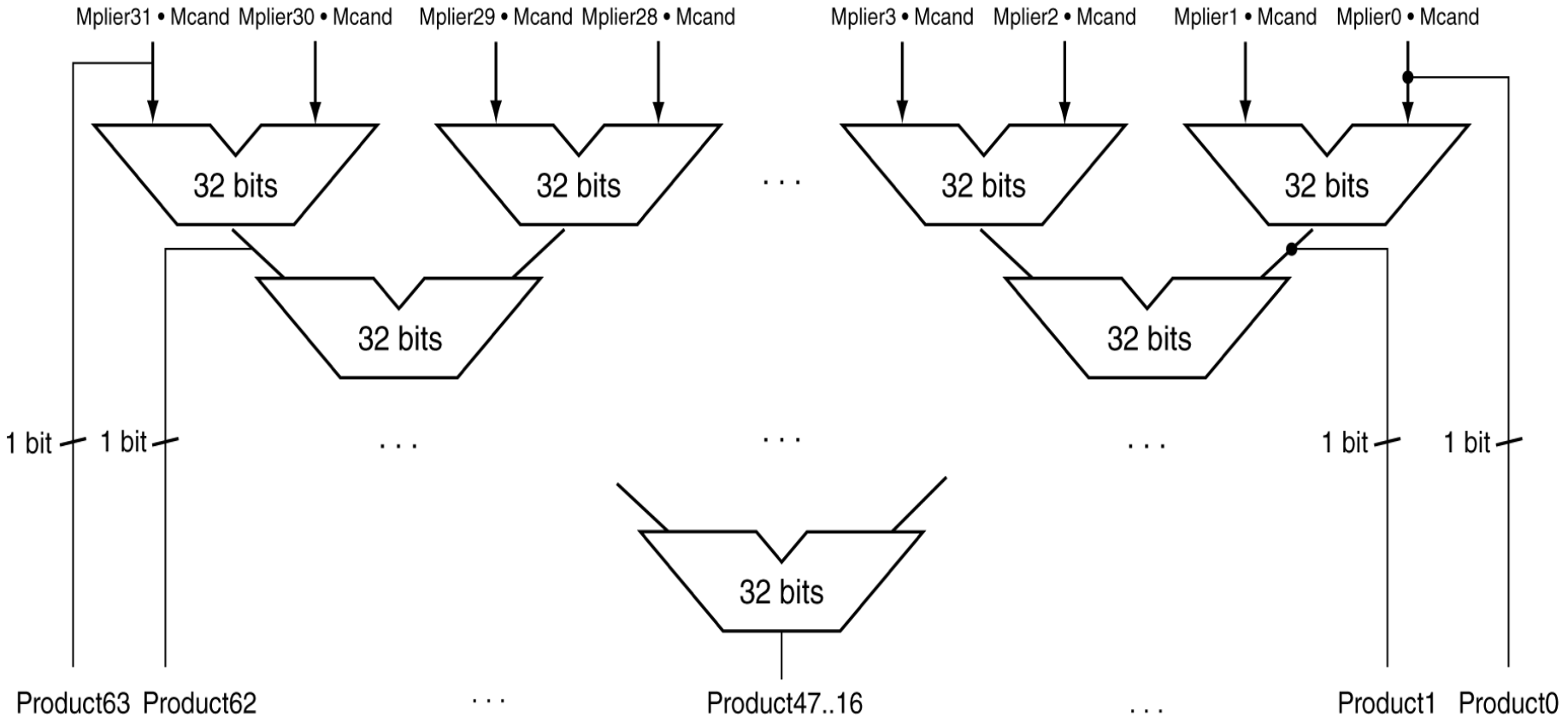
- Perform steps in parallel: add/shift

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



Faster Multiplier (implementation 3)

- Uses multiple adders [***Cost/performance tradeoff***]



MIPS Multiplication Instructions

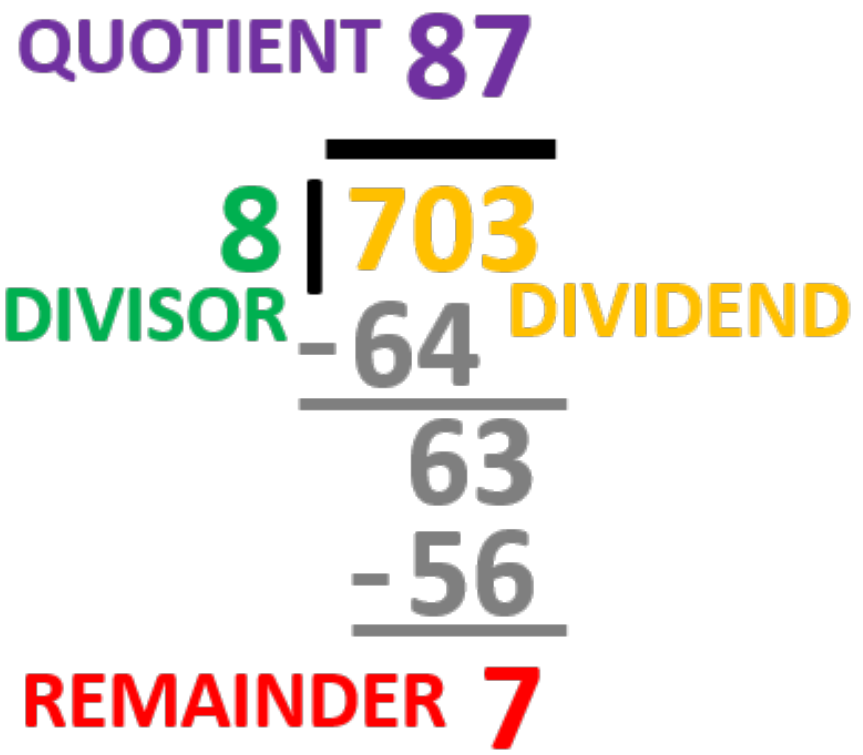
- Two 32-bit **registers** for product
  - **HI**: most-significant 32 bits
  - **LO**: least-significant 32-bits
- Instructions
  - mult rs, rt / multu rs, rt
    - 64-bit product in HI/LO
  - mfhi rd / mflo rd
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits

Multiple ISA HW implementations of distinct cost and performance exist

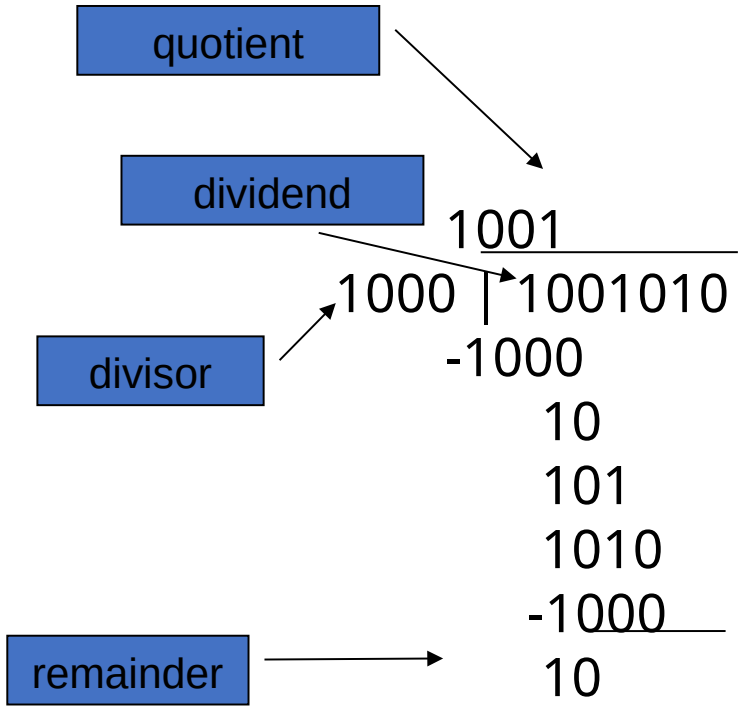
Integer Division

$$\frac{\text{DIVIDEND}}{\text{DIVISOR}} = \text{QUOTIENT R REMAINDER}$$

example:



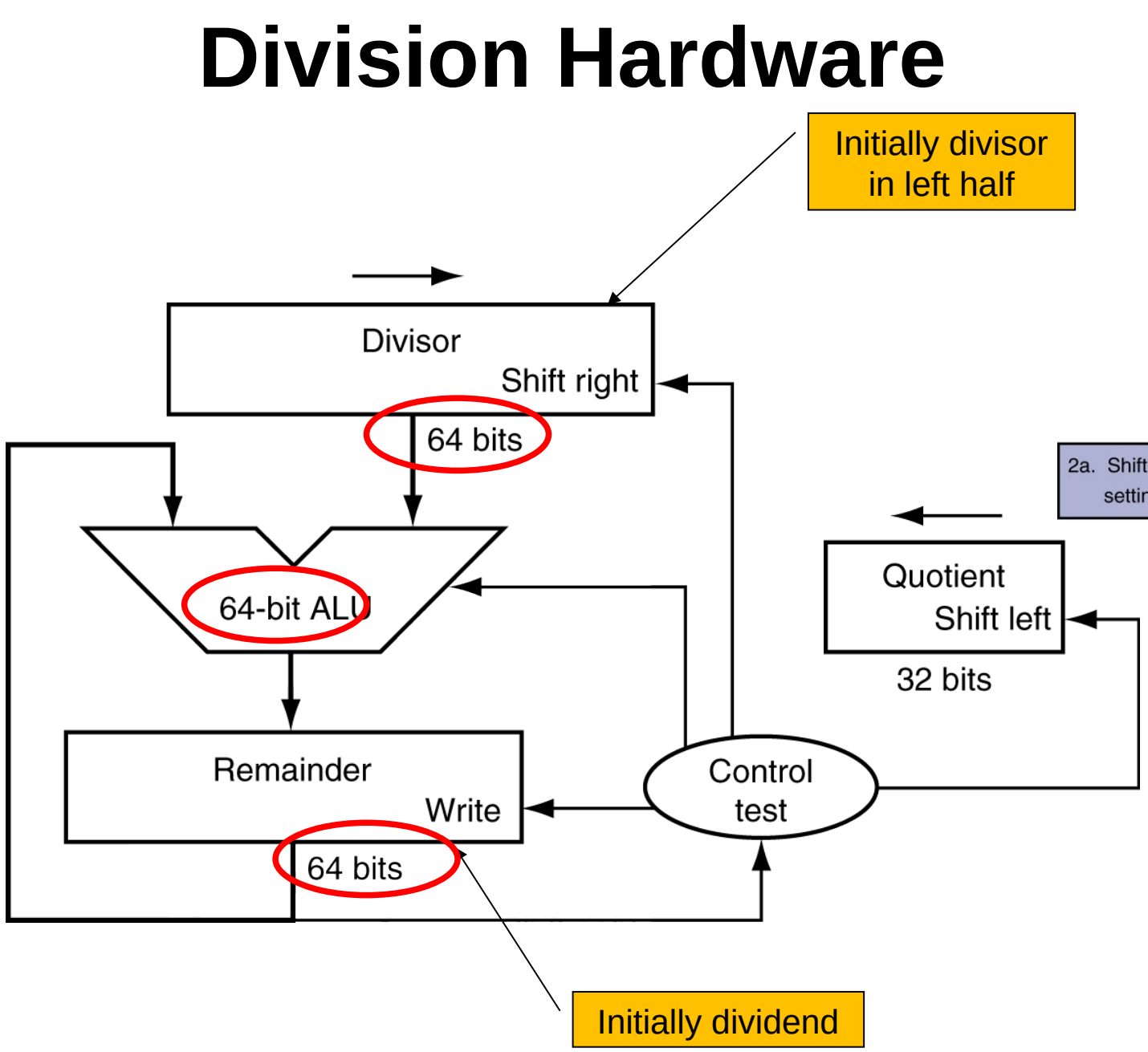
Division

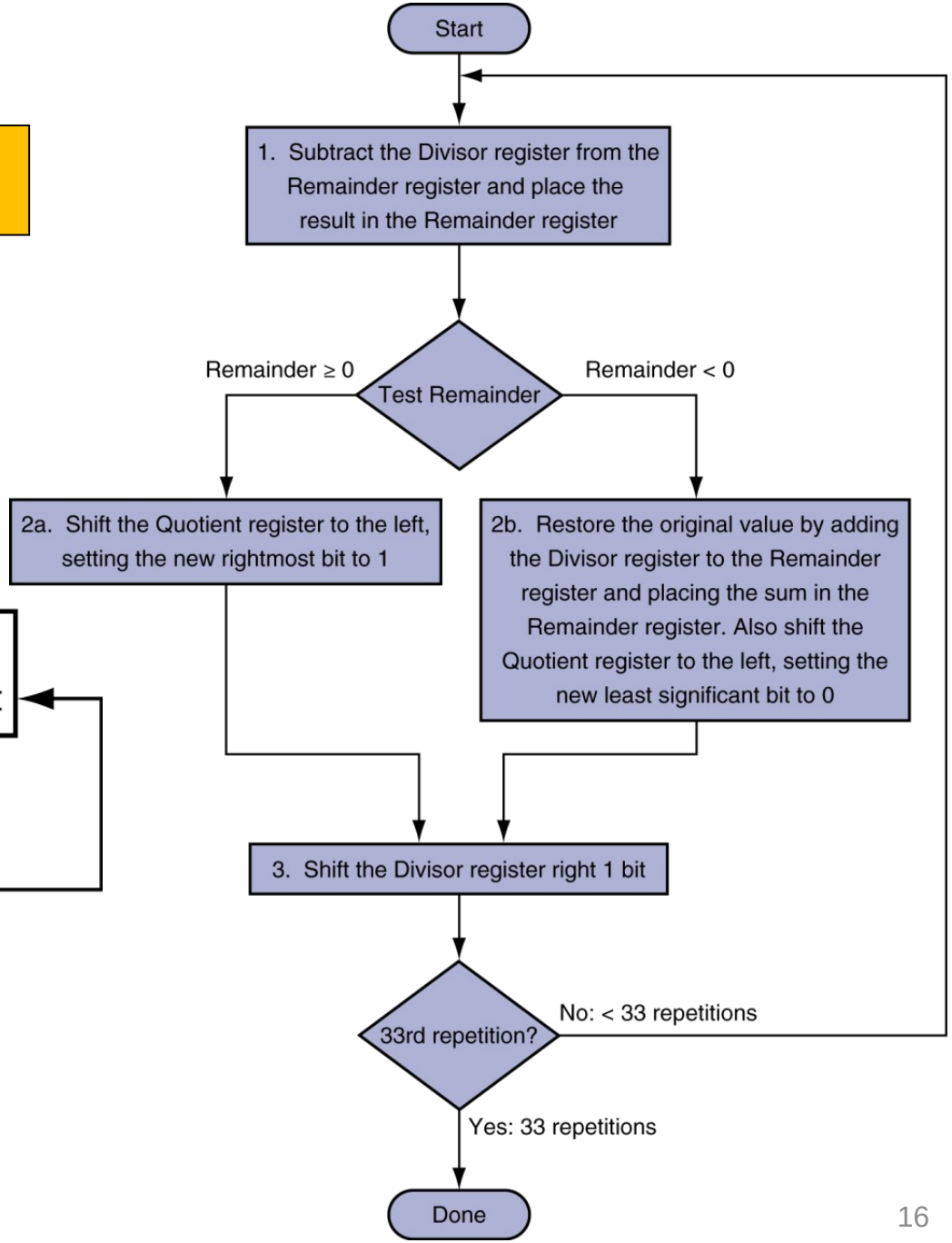


n-bit operands yield *n*-bit quotient and remainder

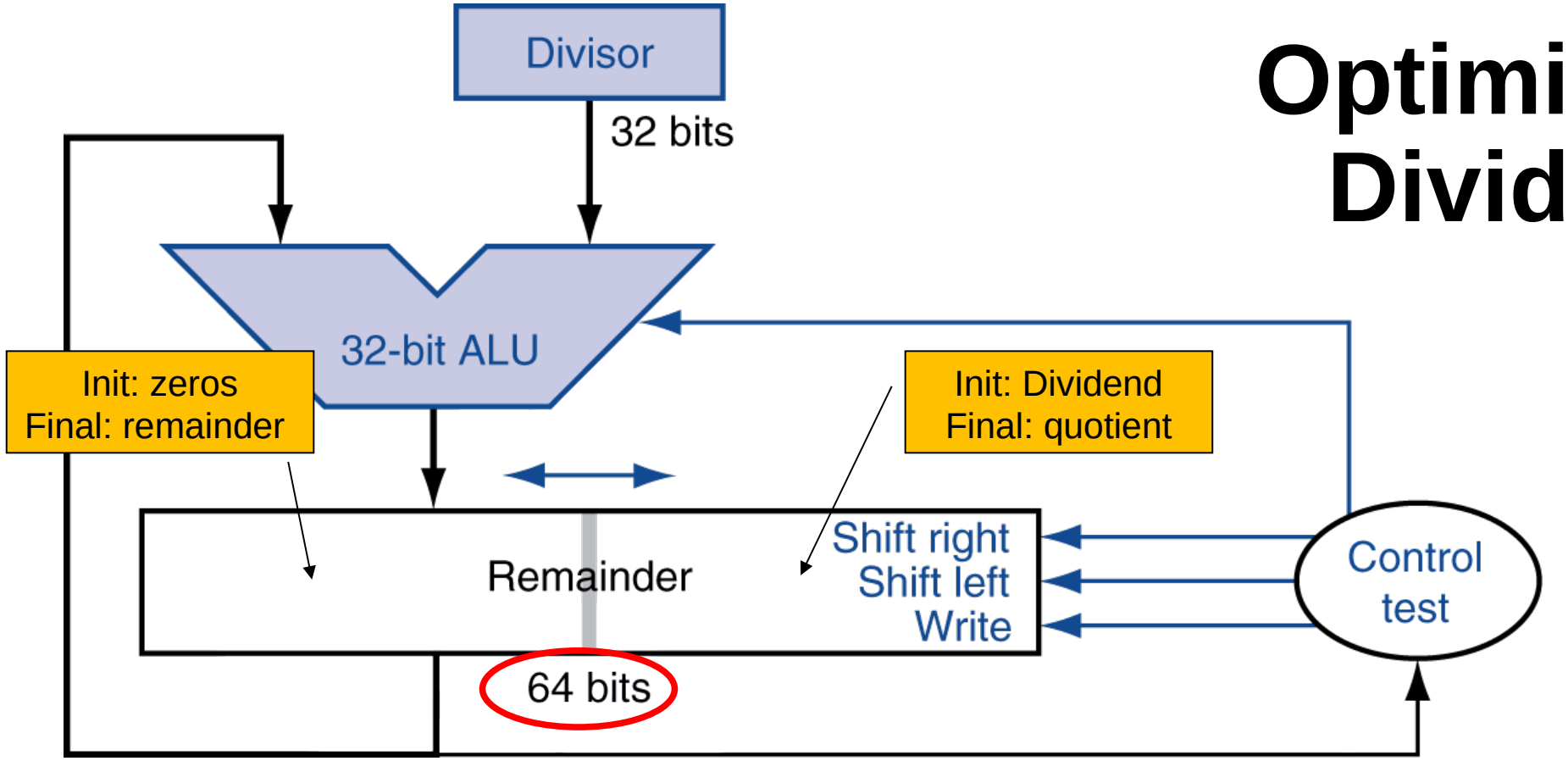
- Example divide 1001010_2 by 1000_2
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware





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 generate multiple quotient bits per step
  - Still require multiple steps
  - Solution currently uses future prediction and correction strategy

MIPS Division Instructions

- 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

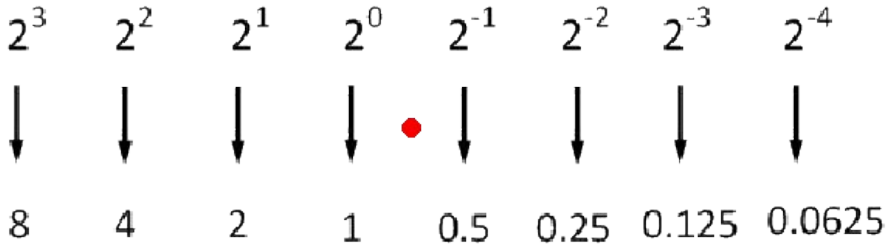
Floating point number Representation

Sections 3.5

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Types **float** and **double** in C
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In **binary**
  - Called binary points

- 5.25 = - 101.01



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 key representations**
  - Single precision (32-bit) (Float) [single]
  - Double precision (64-bit) [Double]

IEEE Floating-Point Format

$$X = \pm 1.xxxxxxx_2 * 2^{yyyy}$$

- 5.25 = - 101.01 = -1.0101 * 2^2

S: sign bit
0 ▲ non-negative,
1 ▲ negative

single(32 bits): 8 bits **single: 23 bits**
double(64 bits): 11 bits **double: 52 bits**

Excess representation
Exponent = yyyy **+ Bias**

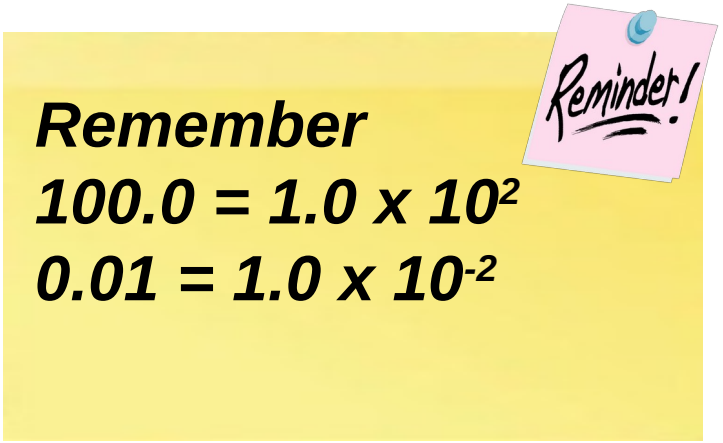
127 for single
1023 for Double

- Biased exponent is **unsigned**, ***is that good?***
- Exponents **00...0000** and **1111...11** reserved

Normalized significand:
$1.0 \le |significand| < 2.0$
Significand = 1.**Fraction**
Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (***hidden bit***)

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: 1 01111110 1000…00
- Double: 1 01111111110 1000…00



$$0.75 = 0.5 + 0.25$$
$$= 0.11_2$$
$$= 1.1 * 2^{-1}$$

Floating-Point Example

- What number is represented by the single-precision float

  1100000010100 0…00

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

- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$

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

  $= -5.0$

Single-Precision Range

- Smallest value
  - Exponent: 00000001
    - actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ▲ ***significand*** = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2_{10} \times 10^{-38}$

- Largest value
  - exponent: 11111110
    - actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ▲ ***significand*** ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4_{10} \times 10^{+38}$

Double-Precision Range

- Smallest value
  - Exponent: 00000000001
    - ▲ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ▲ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2_{10} \times 10^{-308}$

- Largest value
  - Exponent: 11111111110
    - ▲ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ▲ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8_{10} \times 10^{+308}$

Accurate Arithmetic

- Different between computer number and number in real world
  - Computer numbers have limited size → ***limited precision***
  - ***Programmers must remember these limits and write programs accordingly***
- IEEE Std 754 specifies five rounding control

- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults

Floating-Point Precision

- Relative precision
  - **Single**: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx$ **7 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**

Precision-range tradeoff

- fraction field controls precision while exponent field controls range
- fraction bits + exponent bits = fixed number (e.g., 31 bits in single precision)

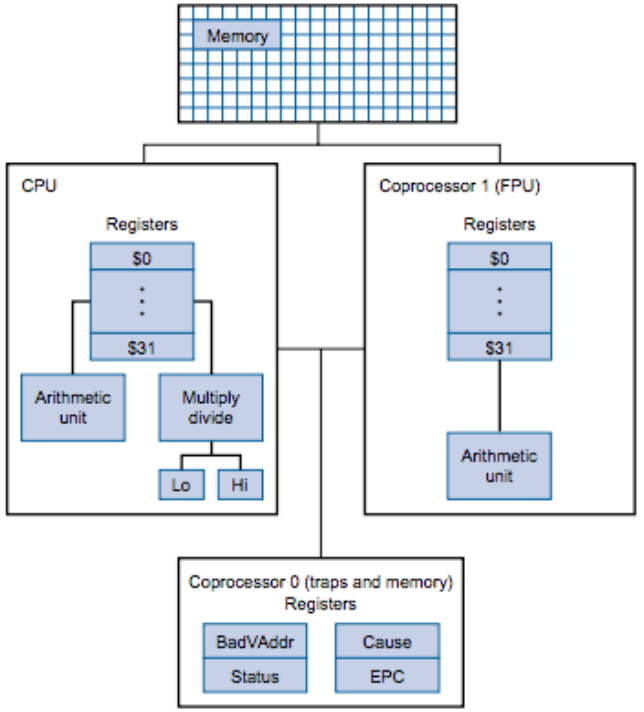
Floating-point MIPS Instruction

Sections 3.5

FP Instructions in MIPS

- **Separate FP processor (CP1)**

- 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



- **<u>FP instructions operate only on FP registers</u>**
  - Programs generally **don't** do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact

MIPS FP Instructions

- ***Arithmetic***
  - ***SINGLE*** *add.s, sub.s, mul.s, div.s*
    - e.g., ***add.s $f0, $f1, $f6***
  - **DOUBLE:** add.d, sub.d, mul.d, div.d
    - e.g., ***mul.d $f4, $f4, $f6***

MIPS floating-point instructions do not have immediate operands.

Load and store instructions

SINGLE: *lwc1, swc1*

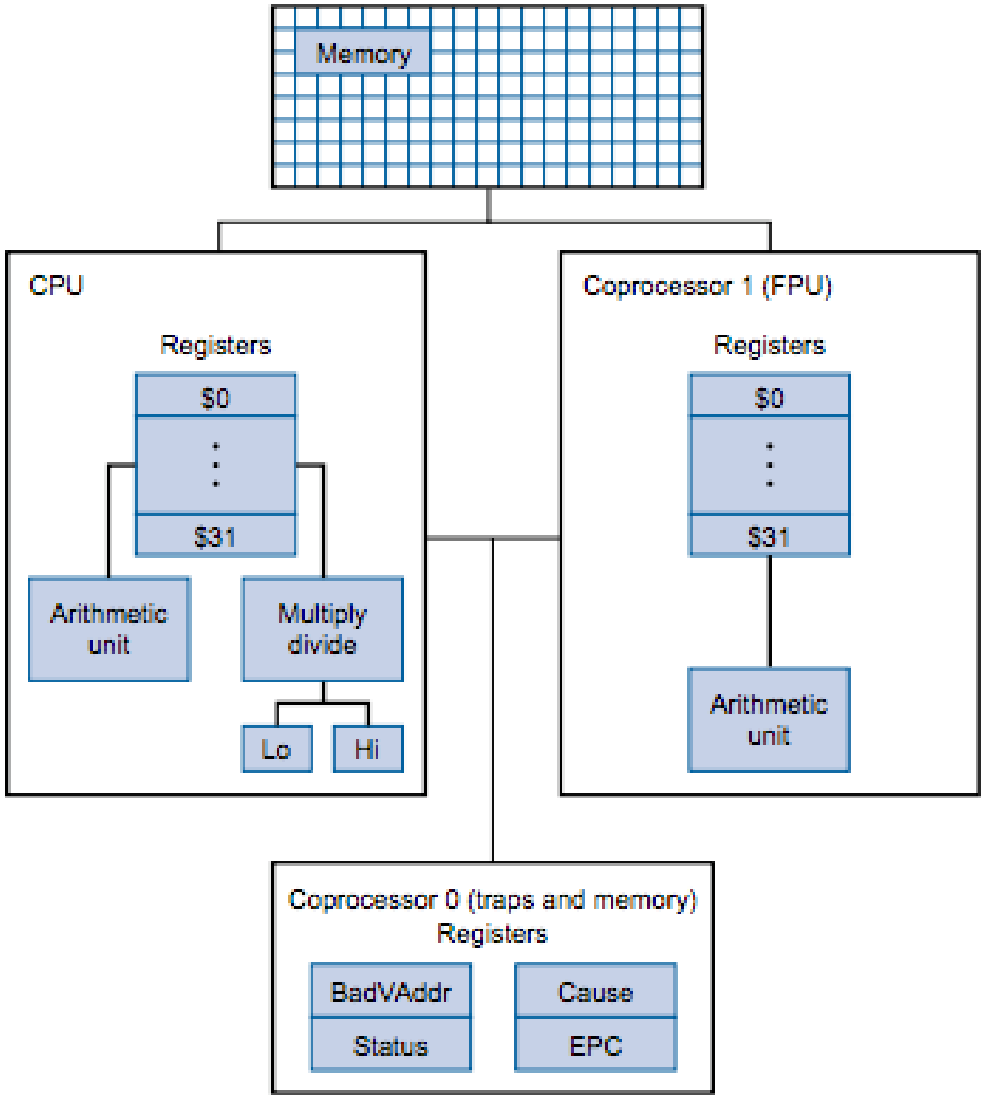
DOUBLE: *ldc1, sdc1*

e.g., ***ldc1 $f8, 32($sp)***

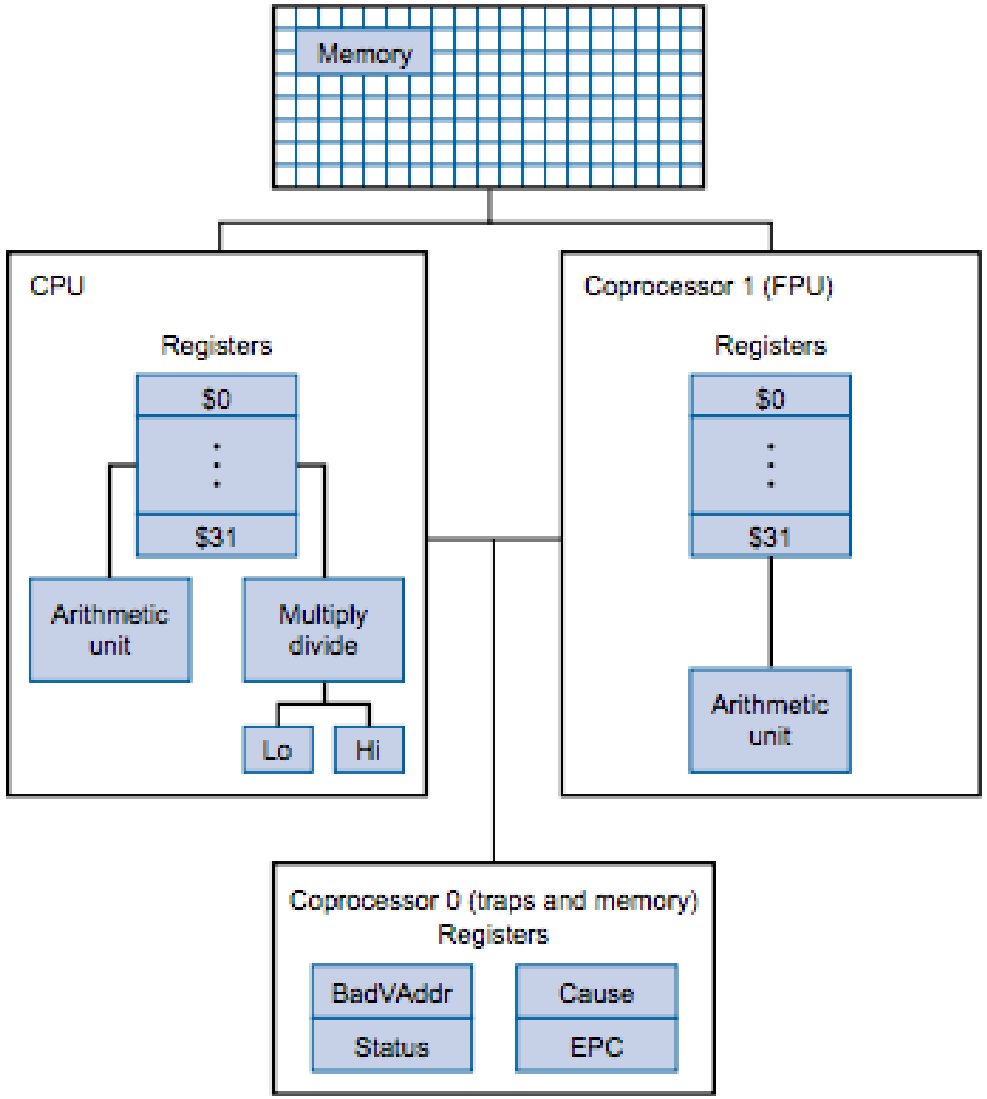
Register Transfer

TO cp1: *mtc1* $t0, $f0

FROM cp1: *mfc1* $t0, $f0



MIPS FP Instructions



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*

- **Branching**
- *bc1t* label

Conversion

cvt.**x.y** $f0,$f1

x & y: s, d, w (integer)

e.g., *cvt.d.w* $f0, $f2 #int_to_double

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
    - Note that fahr is obtained using ***syscall 6*** for single

- MIPS code:

```
# the constants are defined using .float
f2c: lwc1  $f16, const5
     lwc1  $f18, const9
     div.s  $f16, $f16, $f18
     lwc1  $f18, const32
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr      $ra
```

Floating-point MIPS HW

Sections 3.5

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 (binary)

- Now consider a 4-digit binary example
  - Add 0.5_{10} and -0.4375_{10}

$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

1. Align binary points (***Shift*** number with smaller exponent)

$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. ***Add*** significands

$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

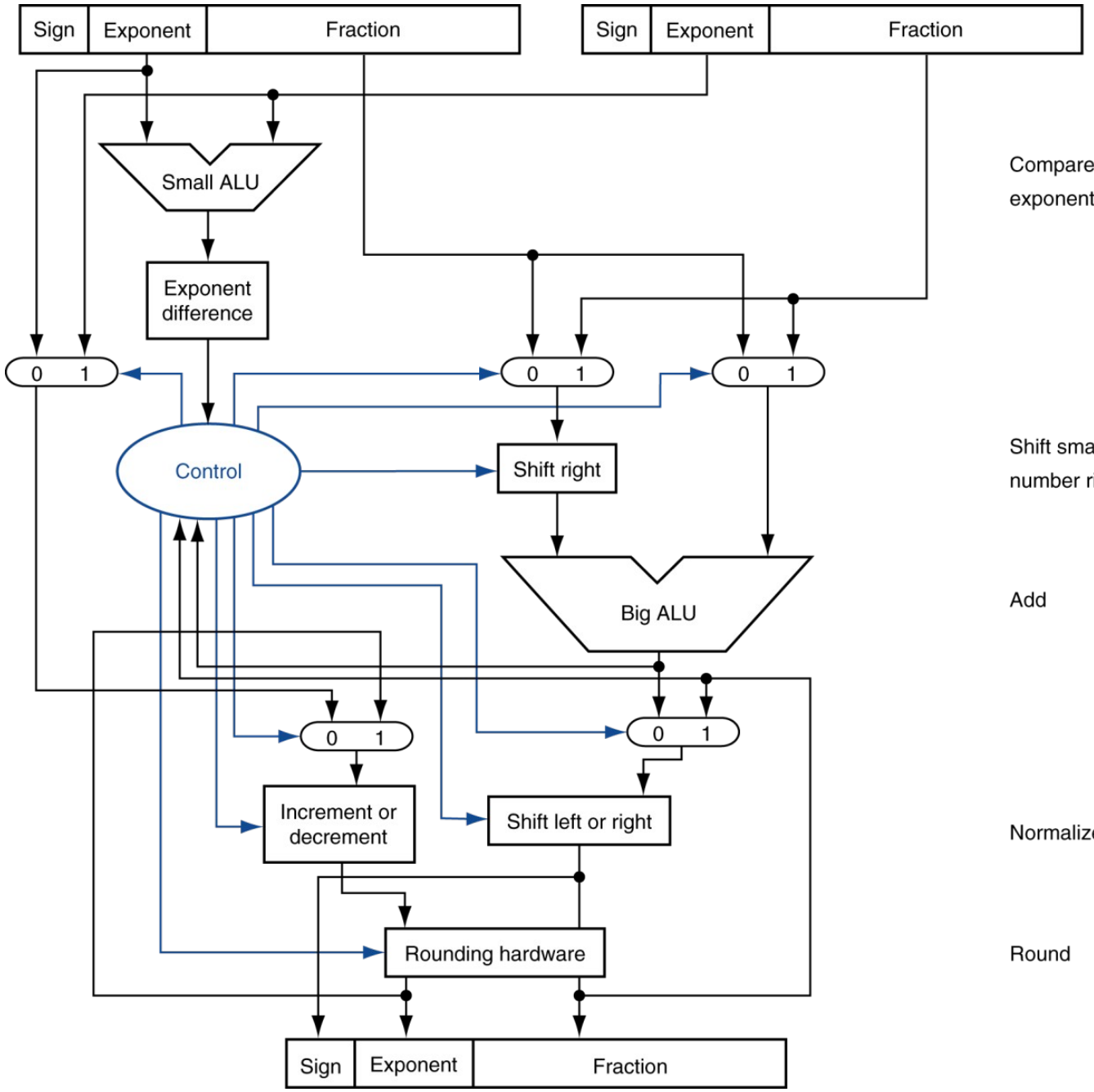
3. ***Normalize*** result & check for over/underflow

$1.000_2 \times 2^{-4}$, with no over/underflow

4. ***Round*** and renormalize if necessary

$1.000_2 \times 2^{-4}$ (no change) $= 0.0625_{10}$

FP Adder Hardware



Step 1

Step 2

Step 3

Step 4

OPTIONAL

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
  - $1.110 \times 9.200 = 10.212$ ▲ 10.212×10^{5}
- 3. **Normalize** result & check for over/underflow
  - 1.0212×10^{6}
- 4. **Round** and renormalize if necessary
  - 1.021×10^{6}
- 5. **Determine *the sign*** of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication (binary)

- Now consider a 4-digit binary example
  - Multiply 0.5_{10} and -0.4375_{10}
    - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$
- 1. ***Add exponents***
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. ***Multiply significands***
    - $1.000_2 \times 1.110_2 = 1.110_2$ ▲ $1.110_2 \times 2^{-3}$
- 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
    - $-1.110_2 \times 2^{-3} = -0.21875_{10}$

FP Arithmetic Hardware

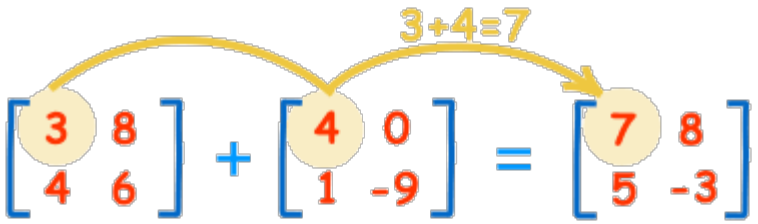
- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ⬇ integer conversion
- **Operations usually takes several cycles**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

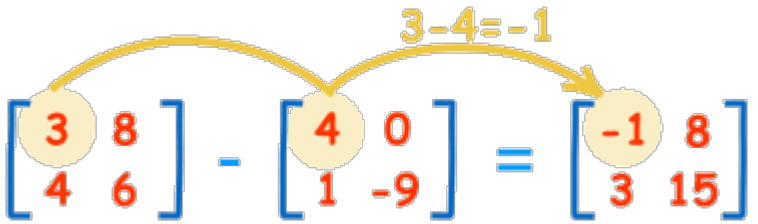
Multidimensional Arithmetic

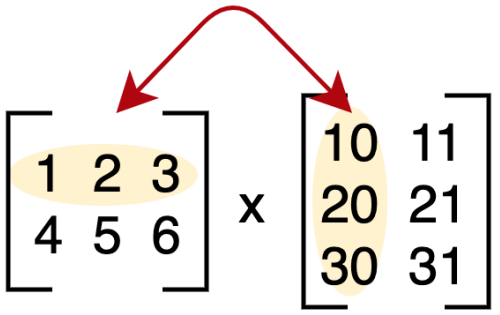
What multidimension?

Example	Terminology
0 1 2	Vector
0 1 2 3 4 5 6 7 8	Matrix

Addition and subtraction: Element wise







Multiplication

$$= \begin{bmatrix} 1x10 + 2x20 + 3x30 & 1x11 + 2x21 + 3x31 \\ 4x10 + 5x20 + 6x30 & 4x11 + 5x21 + 6x31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

https://www.mathsisfun.com/algebra/matrix-introduction.html

Matrix storage

- Storing two-dimensional data in one dimensional **(linear) memory**

Fortran C

PYTHON??

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

```
      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]
```

```
      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
```

Associativity Pitfall

a+(b+c) = (a+b)+c

- 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

Floating point addition is not generally associative!

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 in 1994
  - The market expects accuracy
  - Costed $500 Million to correct
  - No Christmas bonus for Intel Engineers

Questions

- Explain what is the meant by the design principal “make the common case fast.” Considering MIPS processor, identify a design choice that uses this principal. Identify a special case (an uncommon case) and explain how MIPS handle it.
- Floating point number representation involves splitting the data unit (e.g., word) into multiple fields. What are these fields? How the stored value is calculated? How would changing the size of these fields affect the number precision and range?
- The principal of “performance by prediction” is used in computer design. Identify one case for which this principal is used to improve the performance. Explain how this principal is used to improve the performance.
- It is well-known that the design of computer has a cost-performance tradeoff. Identify two scenarios that confirm this tradeoff and explain the tradeoff aspects.