

Chapter 3

Arithmetic for Computers

Jiang Jiang

jiangjiang@ic.sjtu.edu.cn



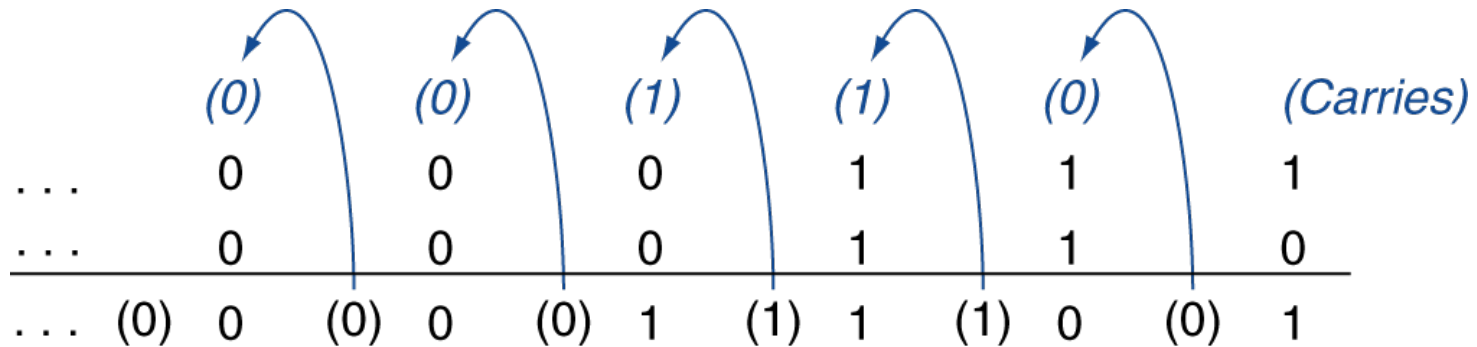
[Adapted from *Computer Organization and Design*,
4th Edition, Patterson & Hennessy, © 2008, MK]

Arithmetic for Computers

- Operations on integers
 - Addition and subtraction
 - Multiplication and division
 - Dealing with overflow
- Floating-point real numbers
 - Representation and operations
- ALU: Arithmetic Logic Unit
 - HW performs addition, subtraction, and usually logical operations such as AND and OR

Integer Addition

■ Example: $7 + 6$



- **Overflow** if result out of range (**2's complement**)
 - 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
<hr/>	
+1:	0000 0000 ... 0000 0001

- **Overflow** if result out of range (**2's complement**)
 - Subtracting two +ve or two -ve operands, no overflow
 - Subtracting +ve from -ve operand
 - Overflow if result sign is 0
 - Subtracting -ve from +ve operand
 - Overflow if result sign is 1

Dealing with Overflow

- Some languages ignore overflow
 - E.g., C
 - Use MIPS `addu`, `addiu`, `subu` instructions
 - Checked by **programmer**
- Other languages require raising an exception
 - E.g., Ada, Fortran
 - 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

Dealing with Overflow

■ add rd,rs,rt

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ← GPR[rs] + GPR[rt]
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp31..0)
endif
```

■ addu rd,rs,rt

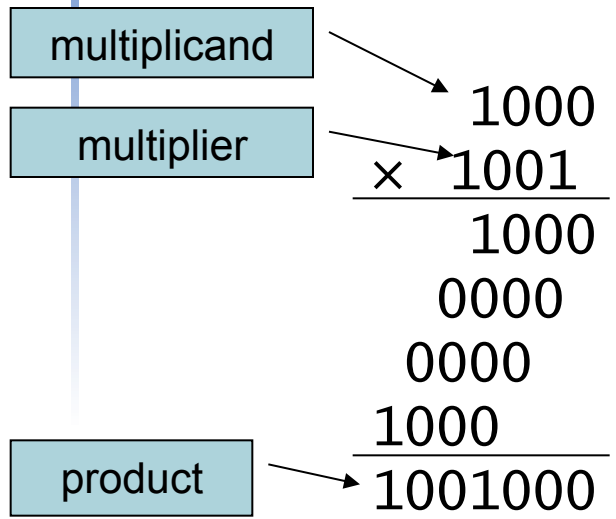
```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← sign_extend(temp31..0)
```

Arithmetic for Multimedia

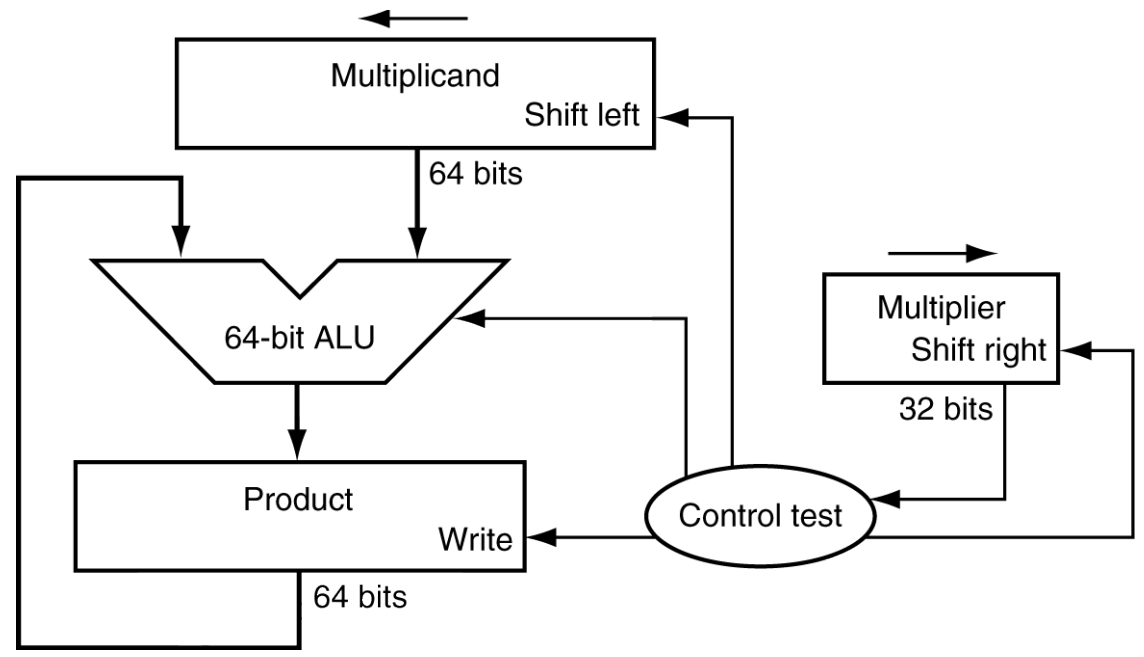
- Graphics and media processing operates on **vectors** of **8-bit** and **16-bit** data
 - 8 bits: byte, 16 bits: halfword
 - Use 64-bit adder, with partitioned **carry chain**
 - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
 - SIMD (single-instruction, multiple-data), **vector** or **Sub-Word Parallel**
- Saturating operations
 - On overflow, result is set to largest representable value (largest positive or most negative number)
 - C.f. 2s-complement modulo arithmetic
 - E.g., clipping in audio, saturation in video
- Multimedia extensions to ISAs
 - MDMX (MIPS Digital Media eXtension), c.f. Intel SSE

Multiplication

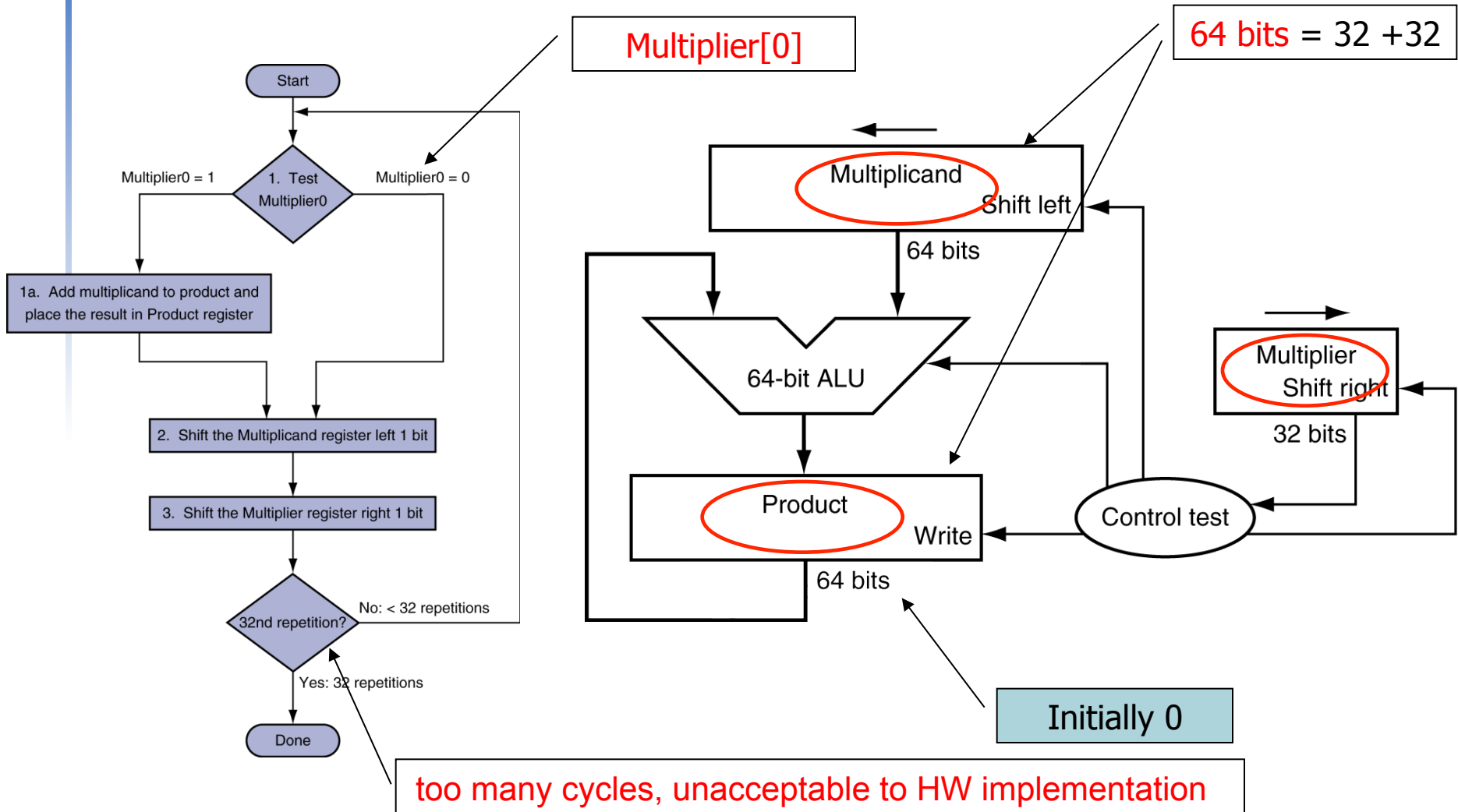
- Start with long-multiplication approach



Length of product is the sum of operand lengths

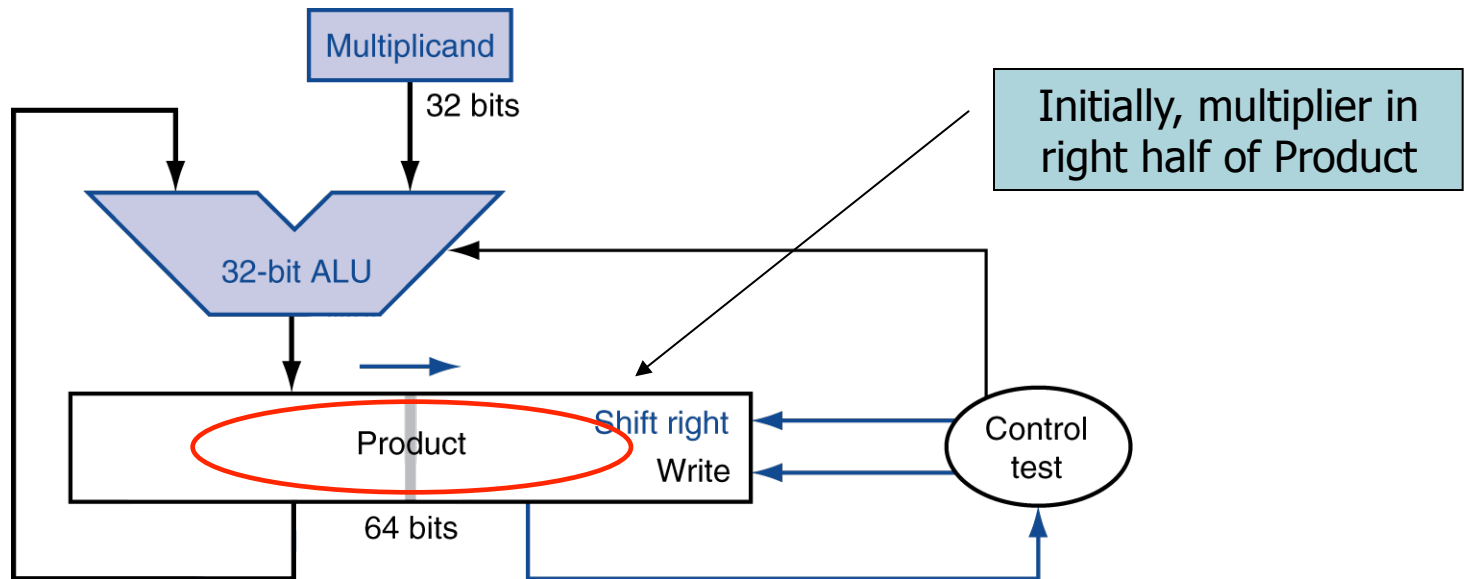


Multiplication Hardware



Optimized Multiplier

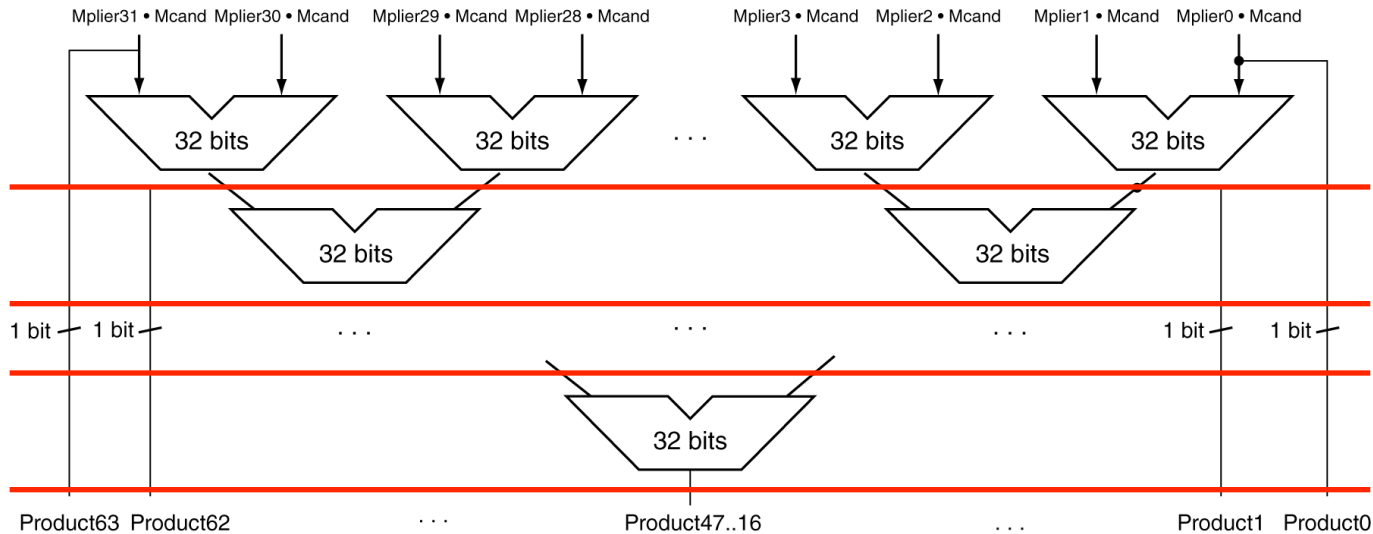
- Perform steps in parallel: add/shift



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

Faster Multiplier

- Uses multiple adders
 - Parallel tree, $\log_2(32)$ or 5 32-bit add time
 - Cost/performance tradeoff

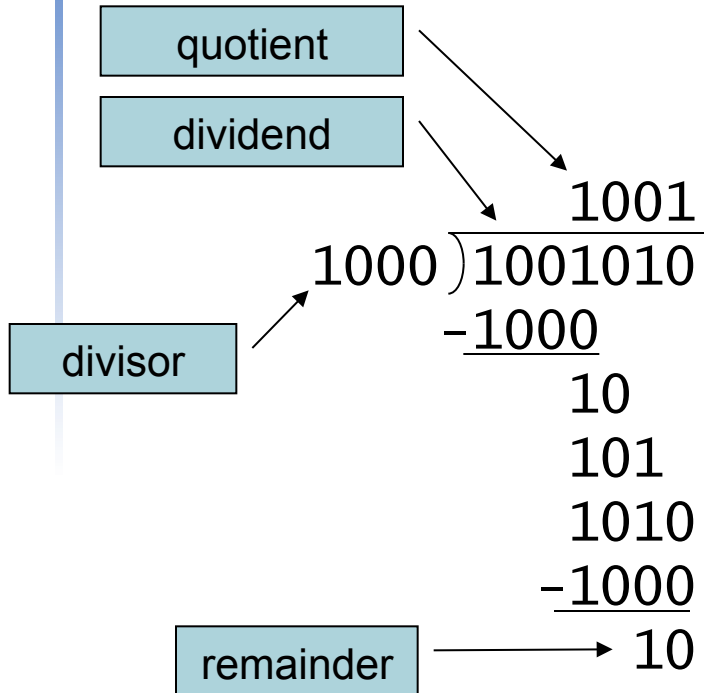


- Can be **pipelined**
 - Several multiplication performed in parallel
 - More resource

MIPS Integer Multiplication

- Two special 32-bit registers for **64-bit product**
 - HI: most-significant 32 bits
 - LO: least-significant 32-bits
- Instructions
 - `mult rs, rt / multu rs, rt` (**2 operands**)
 - **64-bit product in HI/LO**
 - `mfhi rd / mflo rd` (**1 operand**)
 - Move from HI/LO to rd
 - **Can test HI value to see if product overflows 32 bits**
 - `mul rd, rs, rt` (**32b result**)
 - Least-significant 32 bits of product → rd
 - **Pseudoinstruction**

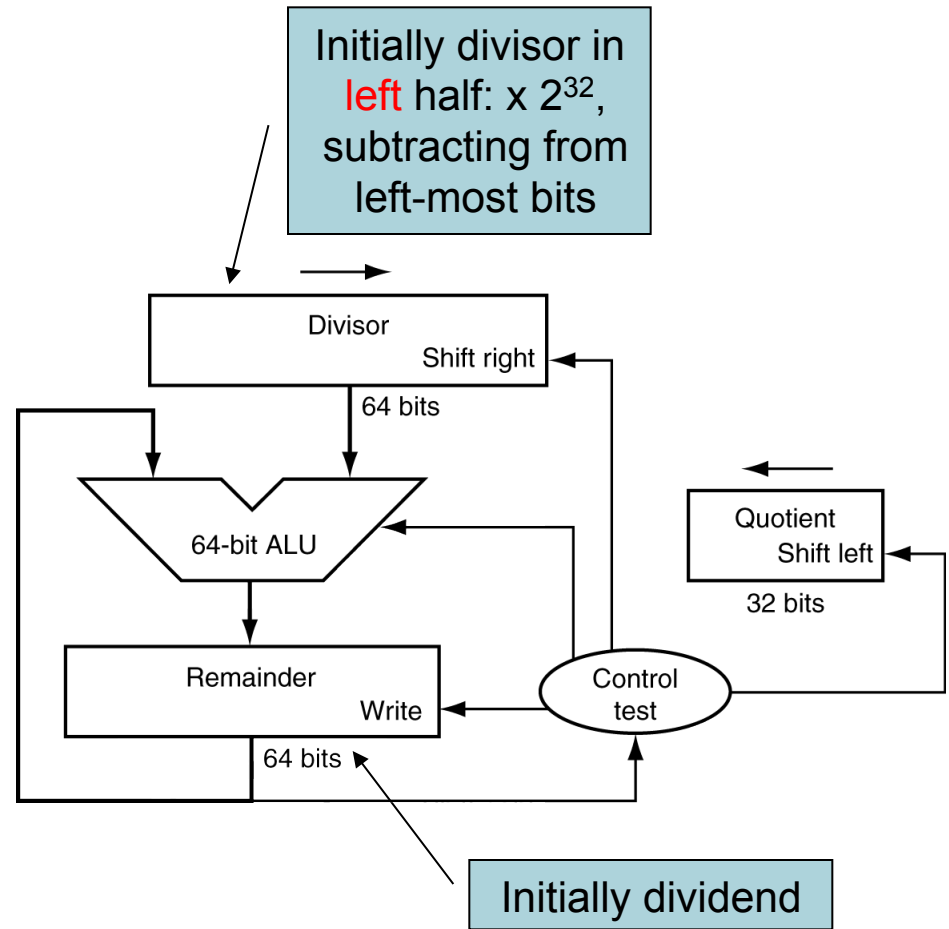
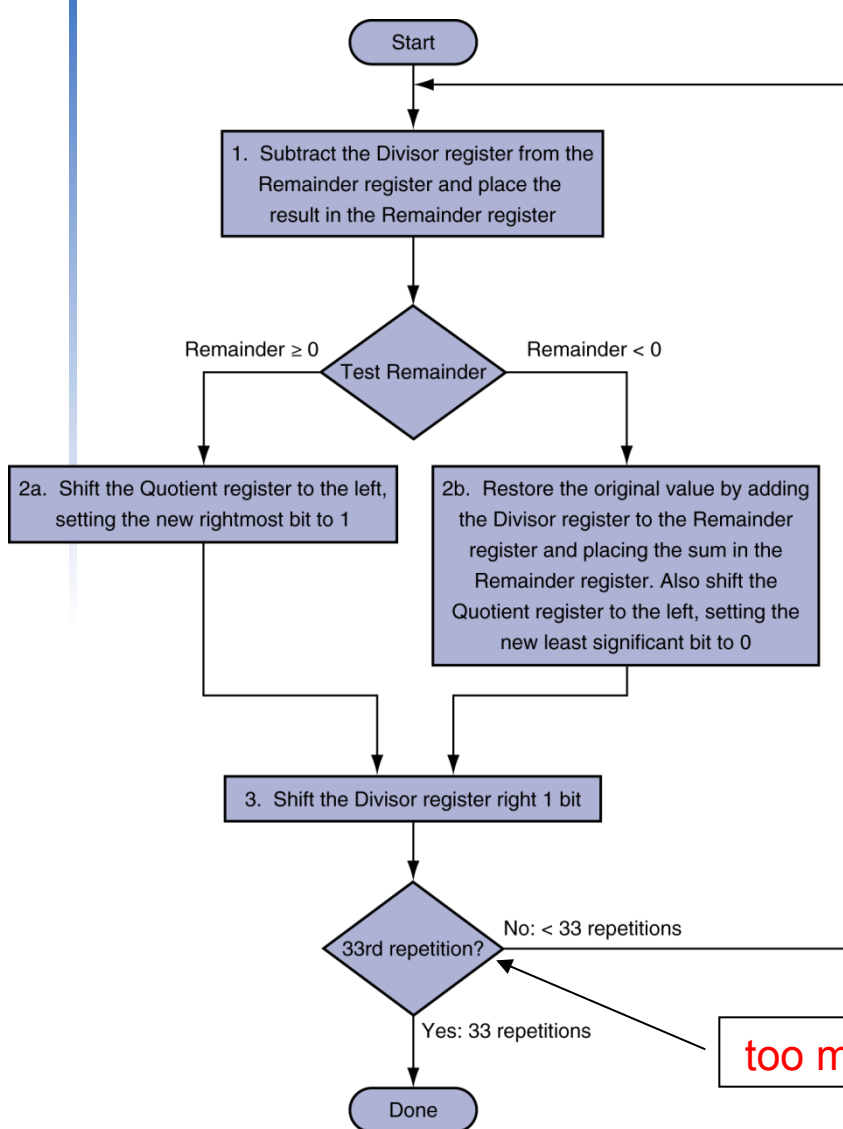
Division



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

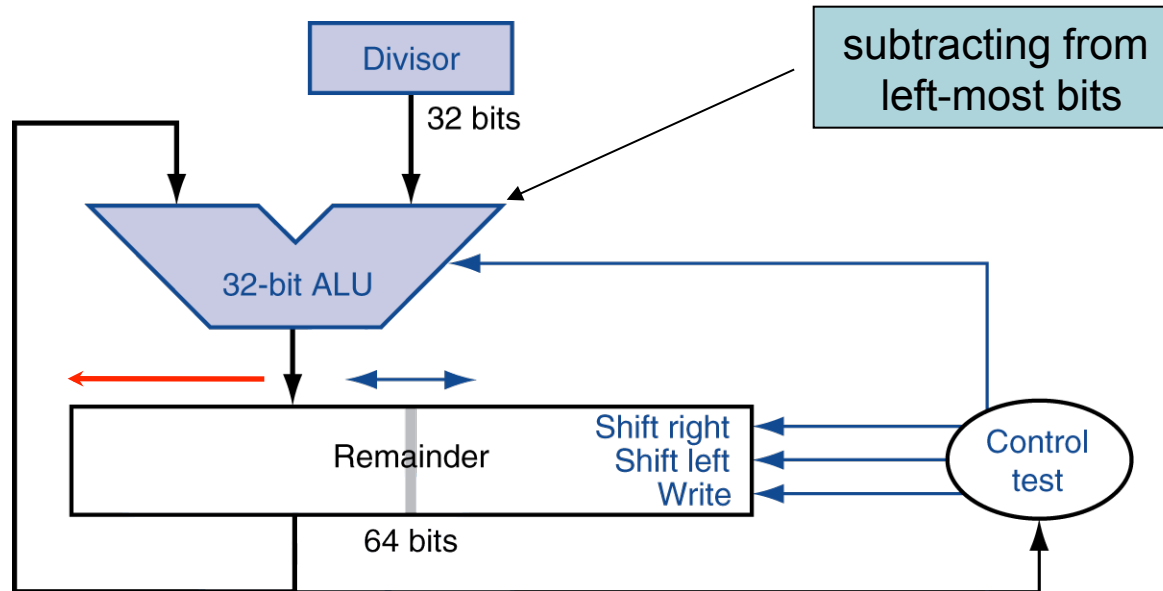
- Check for 0 divisor
- Long division approach
 - If divisor \leq dividend bits
 - 1 bit in quotient, subtract
 - Otherwise
 - 0 bit in quotient, bring down next dividend bit
- Restoring division
 - Do the subtract, and if remainder goes < 0 , add divisor back
- Signed division
 - Divide using absolute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

Division Hardware



too many cycles, unacceptable to HW implementation

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
 - As a rule, the dividend and remainder must have the same sign, no matter what the signs of the divisor and quotient
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
 - A typical value today is 4 bits
 - The key is **guessing** the value to subtract
 - Still require multiple steps

MIPS Division

- Use HI/LO registers for result as multiplication
 - 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
 - `div rd, rs, rt`: pseudoinstruction

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



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

- S: sign bit (0 \Rightarrow non-negative, 1 \Rightarrow negative)
- **Normalize** significand: $1.0 \leq |\text{significand}| < 2.0$
 - Always has a leading pre-binary-point 1 bit, so **no need** to represent it explicitly (**hidden bit**)
 - Significand is Fraction with the “1.” To restore: **1+Fraction**
- Exponent: excess representation: **actual exponent + Bias**
 - Ensures exponent is **unsigned, biased notation**
 - **IEEE 754’s bias**
Single: Bias = **127** = $2^7 - 1$; Double: Bias = **1203** = $2^{10} - 1$
 - Bias is $2^{(n-1)} - 1$, c.f. conventional $2^{(n-1)}$

Floating-Point Example

- Represent -0.75
 - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
 - $S = 1$
 - Fraction = $1000\dots00_2$
 - Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 011111111110_2$
- Single: $10111111101000\dots00$
- Double: $101111111111101000\dots00$

Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

- $S = 1$
 - Fraction = $01000...00_2$
 - 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

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

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

Denormal Numbers

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

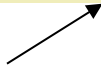
$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

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

Floating-Point Addition

- Consider a 4-digit decimal example

- $9.999 \times 10^1 + 1.610 \times 10^{-1}$

1. Align decimal points

Align the number with smaller exponent, **match the larger 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**

$$1.0015 \times 10^2$$

4. Round and renormalize if necessary

$$1.002 \times 10^2$$

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, **match the larger 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**

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

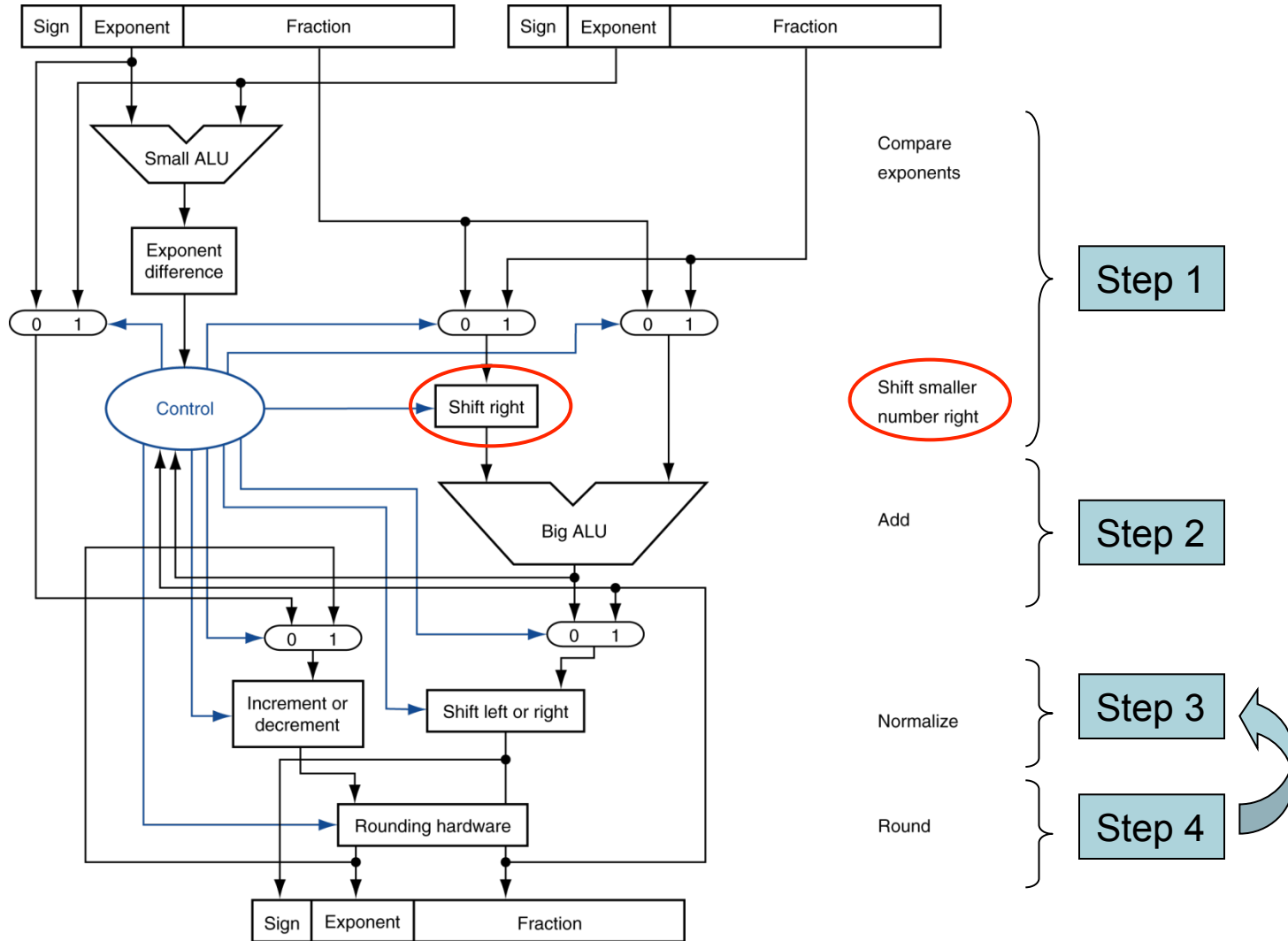
4. Round and renormalize if necessary

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

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**
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:

For biased exponents, subtract bias from sum

Unbiased: $-1 + -2 = -3$

Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

Biased sum !!

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**, especially Big ALU

FP Instructions in MIPS

- FP hardware is coprocessor 1 (**c1**)
 - Adjunct processor that extends the ISA
 - Coprocessors are alternate execution units, with register files **separate** from the CPU
 - MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3
- **Separate FP registers (FPR)**, cf. GPR
 - 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 Instructions in MIPS

- FP load and store instructions
 - `lwc1, swc1; ldc1, sdc1`
 - e.g., `ldc1 $f8, 32($sp)`
- 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 (cf. ARM)**
 - `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

- $X = X + Y \times Z$
 - 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:

	l i	\$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

Accurate Arithmetic

- IEEE Std. 754 specifies additional rounding control
 - Extra bits of precision (guard, round, sticky)
 - Choice of rounding modes
 - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
 - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements
- **Fused multiply add**
 - A floating-point instruction that performs both a multiply and an add, but **rounds only once**
 - A single rounding step increases the precision of multiply add
 - **2 flops per instruction**
 - E.g madd.d fd, fr, fs, ft (MIPS64)

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

- Integer addition is associative, while floating-point addition is not associative
- 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**
 - loads push numbers onto the stack, operations find operands in the 2 top elements of the stack, stores pop elements off the stack
 - Registers indexed from **TOS (Top of Stack)**:
 - 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
 - Intel created a more traditional fp architecture **as part of SSE2**

Streaming SIMD Extension 2 (SSE2)

- Adds 8 × 64-bit registers
 - Compiler can choose to use 8 SSE FPRs
 - Extended to 16 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
 - Single-Instruction Multiple-Data

Fallacies

- Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2
 - Left shift by i places multiplies an integer by 2^i , e.g. `sll`
 - Right shift divides by 2^i ? e.g. `srl`
 - Only for **unsigned** integers
 - For signed integers
 - **Arithmetic right shift**: replicate the sign bit, `sra`
 - E.g., $-5 / 4$, `111110112 >> 2 = 111111102`

Pitfalls

- Pitfall: The MIPS instruction `addiu` sign-extends its 16-bit immediate field

Fallacies

- Fallacy: Only theoretical mathematicians care about floating-point accuracy
 - Important for scientific code
 - But for everyday consumer use?
 - “My bank balance is out by 0.0002¢!”
 - The Intel Pentium FDIV bug
 - In Sept. 1994, a math professor discovered the bug
 - The market expects accuracy
 - This recall cost Intel \$500 million!
 - In April 1997, another FP bug was revealed in Pentium Pro and Pentium II
 - Public acknowledgement
 - Software patch

Concluding Remarks

- ISAs support arithmetic
 - Signed and unsigned integers
 - Operand: 2's complement
 - 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, 3%

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Copy Exception PC + special regs
	multiply	mult \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit unsigned product in Hi, Lo
	divide	div \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Lo = quotient, Hi = remainder
	divide unsigned	divu \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Unsigned quotient and remainder
	move from Hi	mfhi \$s1	$\$s1 = \text{Hi}$	Used to get copy of Hi
move from Lo	mflo \$s1	$\$s1 = \text{Lo}$	Used to get copy of Lo	
Data transfer	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store conditional word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Store word as 2nd half atomic swap
	load upper immediate	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Logical	AND	AND \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	OR	OR \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	NOR	NOR \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	Three reg. operands; bit-by-bit NOR
	AND immediate	ANDi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND with constant
	OR immediate	ORi \$s1,\$s2,100	$\$s1 = \$s2 \mid 100$	Bit-by-bit OR with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 \neq \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; two's complement
	set less than immediate	slti \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; two's complement
	set less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; natural numbers
set less than immediate unsigned	sltiu \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; natural numbers	
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	For procedure call