16.482 / 16.561 Computer Architecture and Design

Instructor: Dr. Michael Geiger

Fall 2013

Lecture 2:

Arithmetic in computers

Lecture outline

- Announcements/reminders
  - Sign up for the course discussion group
  - HW 1 to be posted, due 2/6
- Review: MIPS ISA
- Today's lecture: Arithmetic for computers

Review: MIPS addressing modes

- MIPS implements several of the addressing modes discussed earlier
- To address operands
  - Immediate addressing
    - Example: addi $t0, $t1, **150**
  - Register addressing
    - Example: sub **$t0, $t1, $t2**
  - Base addressing (base + displacement)
    - Example: lw $t0, **16($t1)**
- To transfer control to a different instruction
  - PC-relative addressing
    - Used in conditional branches
  - Pseudo-direct addressing
    - Concatenates 26-bit address (from J-type instruction) shifted left by 2 bits with the 4 upper bits of the PC

Review: MIPS integer registers

Name	Register number	Usage
`$zero`	0	Constant value 0
`$v0-$v1`	2-3	Values for results and expression evaluation
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	Temporary registers
`$s0-$s7`	16-23	Callee save registers
`$t8-$t9`	24-25	Temporary registers
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

- List gives mnemonics used in assembly code
  - Can also directly reference by number ($0, $1, etc.)
- Conventions
  - $s0-$s7 are preserved on a function call (callee save)
  - Register 1 ($at) reserved for assembler
  - Registers 26-27 ($k0-$k1) reserved for operating system

Review: MIPS data transfer instructions

- For all cases, calculate effective address first
  - MIPS doesn't use segmented memory model like x86
  - Flat memory model → EA = address being accessed
- lb, lh, lw
  - Get data from addressed memory location
  - Sign extend if lb or lh, load into rt
- lbu, lhu, lwu
  - Get data from addressed memory location
  - Zero extend if lb or lh, load into rt
- sb, sh, sw
  - Store data from rt (partial if sb or sh) into addressed location

Review: MIPS computational instructions

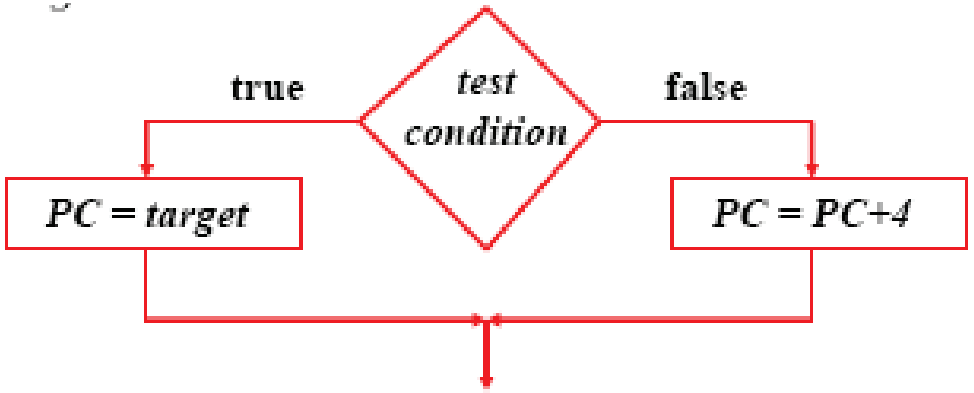
- Arithmetic
  - Signed: add, sub, mult, div
  - Unsigned: addu, subu, multu, divu
  - Immediate: addi, addiu
    - Immediates are sign-extended
- Logical
  - and, or, nor, xor
  - andi, ori, xori
    - Immediates are zero-extended
- Shift (logical and arithmetic)
  - srl, sll – shift right (left) logical
    - Shift the value in rs by shamt digits to right or left
    - Fill empty positions with 0s
    - Store the result in rd
  - sra – shift right arithmetic
    - Same as above, but sign-extend the high-order bits
  - Can be used for multiply / divide by powers of 2

Review: computational instructions (cont.)

- Set less than
  - Used to evaluate conditions
    - Set rd to 1 if condition is met, set to 0 otherwise
  - slt, sltu
    - Condition is rs < rt
  - slti, sltiu
    - Condition is rs < immediate
    - Immediate is *sign-extended*
- Load upper immediate (lui)
  - Shift immediate 16 bits left, append 16 zeros to right, put 32-bit result into rd

Review: MIPS control instructions

- Branch instructions test a condition
  - Equality or inequality of rs and rt
    - beq, bne
    - Often coupled with slt, sltu, slti, sltiu
  - Value of rs relative to rt
    - Pseudoinstructions: blt, bgt, ble, bge
- Target address → add sign extended immediate to the PC
  - Since all instructions are words, immediate is shifted left two bits before being sign extended



Review: MIPS control instructions (cont.)

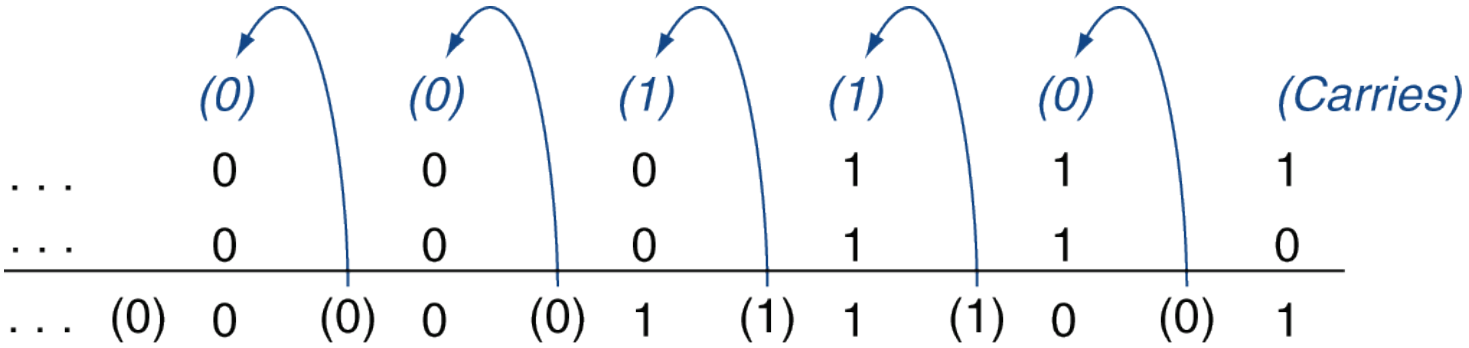
- *Jump* instructions unconditionally branch to the address formed by either
  - Shifting left the 26-bit target two bits and combining it with the 4 high-order PC bits
    - j
  - The contents of register $rs
    - jr
- *Branch-and-link* and *jump-and-link* instructions also save the address of the next instruction into $ra
  - jal
  - Used for subroutine calls
  - jr $ra used to return from a subroutine

Arithmetic for Computers

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

Integer Addition

- Example: 7 + 6



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

Multiplication

- Start with long-multiplication approach

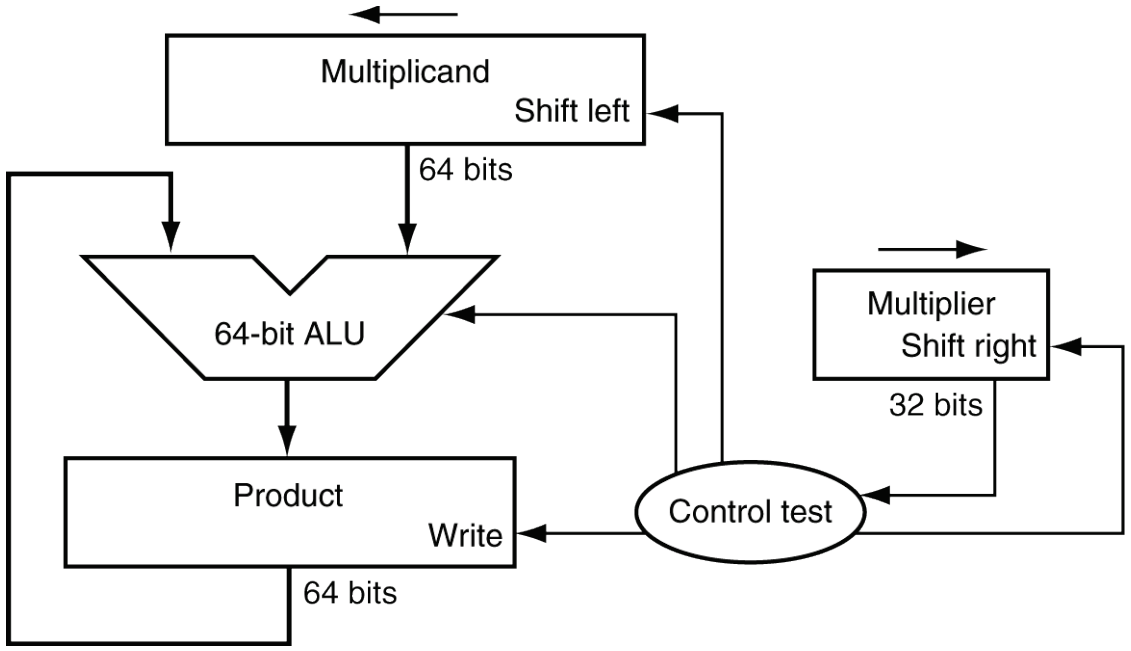
multiplicand

multiplier

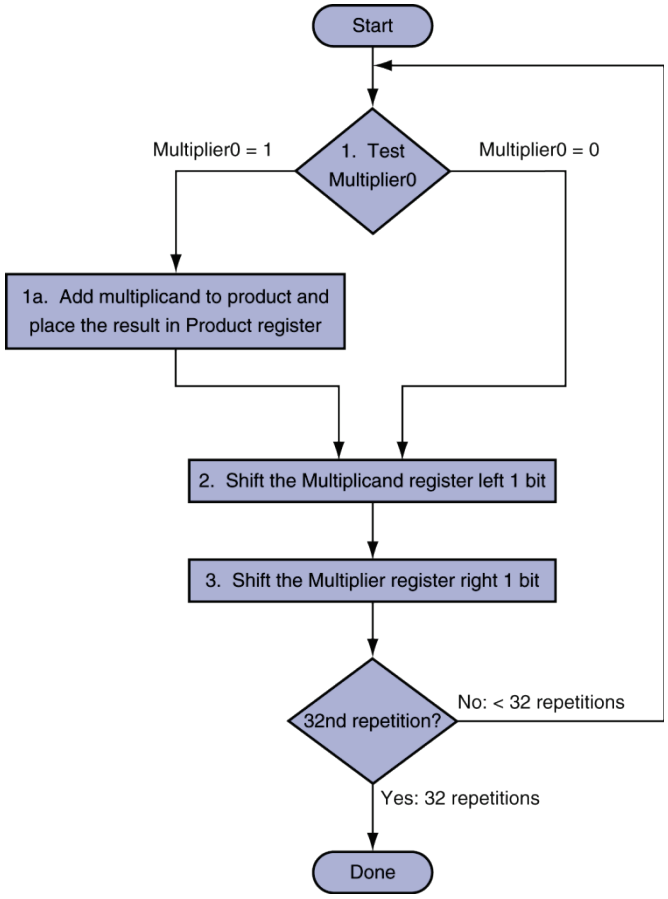
product

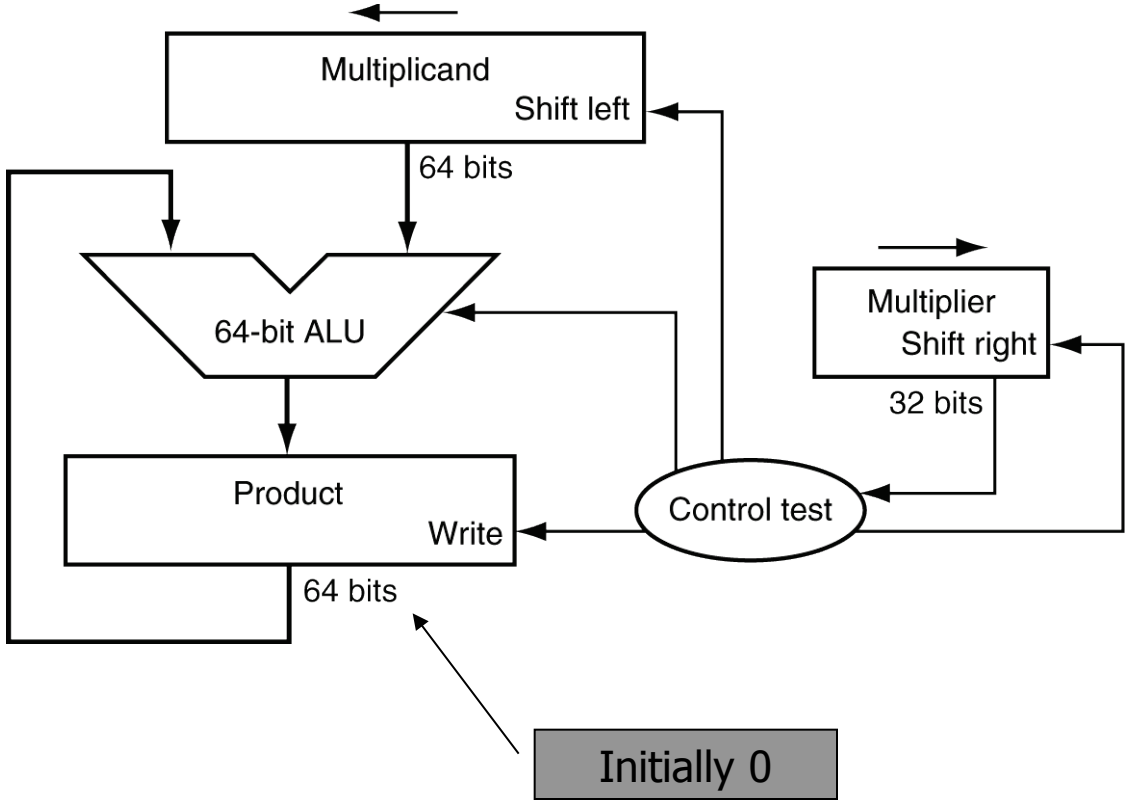
```
    1000
  × 1001
  ------
    1000
   0000
  0000
 1000
------
1001000
```

Length of product is the sum of operand lengths



Multiplication Hardware





Integer multiplication (step by step)

0100 (multiplicand)

x 0011 (multiplier)

00000000 (running product)

- Steps:
  - Initialize product register to 0

Integer multiplication (step by step)

```
      ← 1000 (multiplicand)
      x  0011 (multiplier)
  00000000 (running product)
+      0100
  00000100 (new running product)
```

- Steps:
  - Initialize product register to 0
  - Multiplier bit = 1 → add multiplicand to product, then shift multiplicand left

Integer multiplication (step by step)

```
←   10000 (multiplicand)
  x  0011 (multiplier)
   00000100 (running product)
+       1000
   00001100 (new product)
```

- Steps:
  - Initialize product register to 0
  - Multiplier bit = 1 → add multiplicand to product, then shift multiplicand left
  - Multiplier bit = 1 → add multiplicand to product, then shift multiplicand left

Integer multiplication (step by step)

← 100000 (multiplicand)

x 0011 (multiplier)

00001100 (running product)

- Steps:
  - Initialize product register to 0
  - Multiplier bit = 1 → add multiplicand to product, then shift multiplicand left
  - Multiplier bit = 1 → add multiplicand to product, then shift multiplicand left
  - Multiplier bit = 0 → shift multiplicand left

Integer multiplication (step by step)

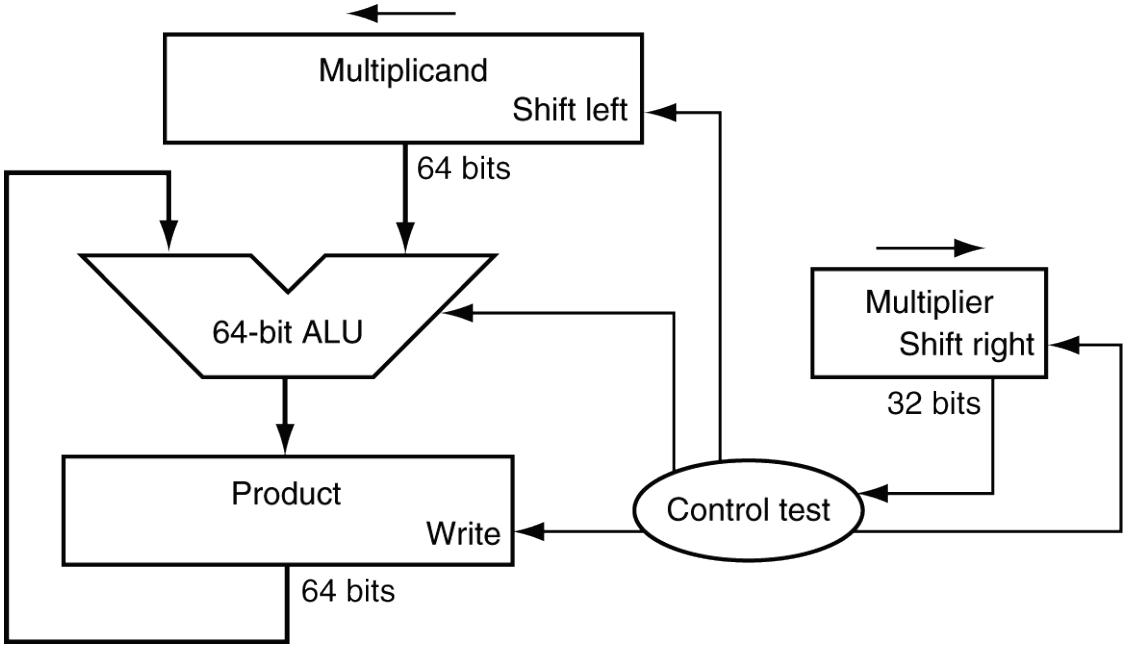
←1000000 (multiplicand)

x 0011 (multiplier)

00001100 (running product)

- Steps:
  - Initialize product register to 0
  - Multiplier bit = 1 → add multiplicand to product, then shift multiplicand left
  - Multiplier bit = 1 → add multiplicand to product, then shift multiplicand left
  - Multiplier bit = 0 → shift multiplicand left
  - Multiplier bit = 0 (MSB) → shift multiplicand left

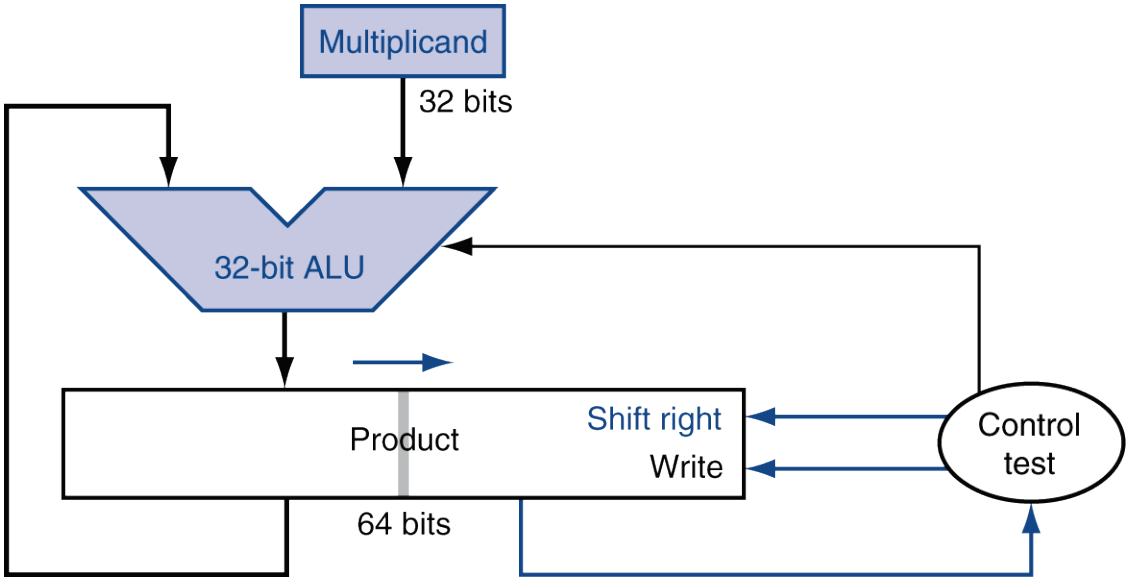
Refining multiplication hardware



- Significant wasted space in this hardware: where?
  - Multiplicand register: always half filled with zeroes
  - No need for a 2n-bit multiplicand register
  - No need for a 2n-bit ALU

Optimized Multiplier

- Perform steps in parallel: add/shift



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

Refined hardware example

0100 (multiplicand)
x 0011 (multiplier)

00000011 (running product + multiplier)
+ 0100
01000011
→00100001 (product after 1st iteration: add & shift)
+ 0100
01100001
→00110000 (product after 2nd iteration: add & shift)

→00011000 (product after 3rd iteration: shift)
→00001100 (product after 4th iteration: shift)

Refined hardware example 2

- Show how the refined hardware handles the multiplication of 6 x 7

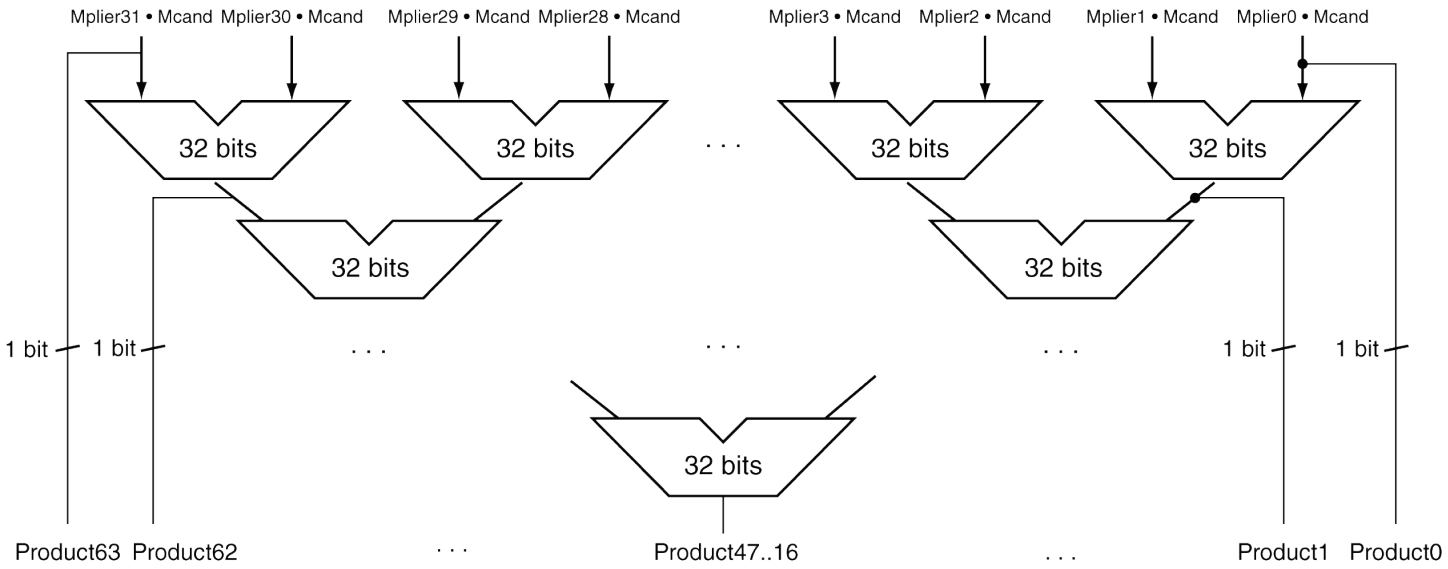
Refined hardware example 2

0110 (multiplicand = 6)
x 0111 (multiplier = 7)

00000111 (running product + multiplier)
+ 0110
01100111
→00110011 (product after 1st iteration: add & shift)
+ 0110
10010011
→01001001 (product after 2nd iteration: add & shift)
+ 0110
10101001
→01010100 (product after 3rd iteration: add & shift)
→00101010 (product after 4th iteration: shift)

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

Multiplying signed values

- What happens if you multiply (-4) x 5?
  - Use 4-bit values, assume solution has 8 bits

```
          1 1 0 0
        x 0 1 0 1
        ---------
          1 1 0 0
        0 0 0 0
      1 1 0 0
    0 0 0 0
    ---------------
  0 0 1 1 1 1 0 0
```

- Fixing this problem:
  - On paper: sign-extend multiplicand
  - In hardware: sign-extend product when shifting

Multiplying signed values (pt. 2)

- What if we switch multiplier/multiplicand?

```
          0 1 0 1
        x 1 1 0 0
      ---------
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 1
0 0 1 0 1
---------------
0 0 1 1 1 1 0 0
```

- Sign extension of multiplicand or shifted product doesn't work!

Booth's Algorithm

- Similar to last hardware method
  - Will add one of two values into left half of product register: multiplicand or its negative
- Need to add
  - One bit to left of product, multiplicand, and –(multiplicand)
    - Use sign extension to fill this bit
  - One additional bit to right of product
    - Starts as 0
- Check last two bits of product register
  - If equal (00 or 11), just shift right
  - If 01, add multiplicand and then shift right
  - If 10, add –(multiplicand) and then shift right
  - Note: use arithmetic right shifts (copy sign bit)
- Ignore extra bits when done to get product

Booth's example: initial setup

- 5 x (-3)

0	0	1	0	1						Multiplicand (5) with extra 0
1	1	0	1	1						-(Multiplicand) (-5) with extra 1
0	0	0	0	0	1	1	0	1	0	Product register with extra 0 on right and left; multiplier is in right half of register

Booth's example: step 1

- Checking rightmost 2 bits (underlined below)
- Rightmost bits = 10
  - Add –(multiplicand), then shift

	0	0	1	0	1						**Multiplicand (5) with extra 0**
	1	1	0	1	1						**-(Multiplicand) (-5) with extra 1**
	0	0	0	0	0	1	1	0	1	0	**Initial value of product register**
+	1	1	0	1	1						**-(Multiplicand)**
	1	1	0	1	1	1	1	0	1	0	**Result of addition**
	1	1	1	0	1	1	1	1	0	1	**Result after right shift**

Booth's example: step 2

- Rightmost bits = 01
  - Add multiplicand then shift

	0	0	1	0	1					Multiplicand (5) with extra 0
	1	1	0	1	1					-(Multiplicand) (-5) with extra 1
	1	1	1	0	1	1	1	1	0	1	Product after step 1
+	0	0	1	0	1					Multiplicand
	0	0	0	1	0	1	1	1	0	1	Result of addition
	0	0	0	0	1	0	1	1	1	0	Result after right shift

Booth's example: step 3

- Rightmost bits = 10
  - Add –(multiplicand) then shift

	0	0	1	0	1					Multiplicand (5) with extra 0
	1	1	0	1	1					-(Multiplicand) (-5) with extra 1
	0	0	0	0	1	0	1	1	1	0	Product after step 2
+	1	1	0	1	1					-(Multiplicand)
	1	1	1	0	0	0	1	1	1	0	Result of addition
	1	1	1	1	0	0	0	1	1	1	Result after right shift

Booth's example: step 4

- Rightmost bits = 11
  - Just shift right

0	0	1	0	1						Multiplicand (5) with extra 0
1	1	0	1	1						-(Multiplicand) (-5) with extra 1
1	1	1	1	0	0	0	1	1	1	Product after step 3
1	1	1	1	1	0	0	0	1	1	Final result

Ignore extra bits to get actual product

Example 2

- Show the operation of Booth's algorithm for
  - -8 x 6

Booth's example 2: initial setup

- -8 x 6

1 1 0 0 0	Multiplicand (-8) with extra 1
0 1 0 0 0	-(Multiplicand) (8) with extra 0
0 0 0 0 0 0 1 1 0 0	Product register with extra 0 on right and left; multiplier is in right half of register

Booth's example 2: step 1

- Checking rightmost 2 bits (underlined below)
- Rightmost bits = 00
  - Shift right

1	1	0	0	0						**Multiplicand (-8) with extra 1**
0	1	0	0	0						**-(Multiplicand) (8) with extra 0**
0	0	0	0	0	0	1	1	0	0	**Initial value of product register**
0	0	0	0	0	0	0	1	1	0	**Result after right shift**

Booth's example 2: step 2

- Rightmost bits = 10
  - Add –(multiplicand) then shift

	1	1	0	0	0						Multiplicand (-8) with extra 1
	0	1	0	0	0						-(Multiplicand) (8) with extra 0
	0	0	0	0	0	0	0	1	1	0	Product after step 1
+	0	1	0	0	0						-Multiplicand
	0	1	0	0	0	0	0	1	1	0	Result of addition
	0	0	1	0	0	0	0	0	1	1	Result after right shift

Booth's example 2: step 3

- Rightmost bits = 11
  - Shift right

1	1	0	0	0						Multiplicand (-8) with extra 1
0	1	0	0	0						-(Multiplicand) (8) with extra 0
0	0	1	0	0	0	0	0	1	1	Product after step 2
0	0	0	1	0	0	0	0	0	1	Result after right shift

Booth's example 2: step 4

- Rightmost bits = 01
  - Add multiplicand then shift

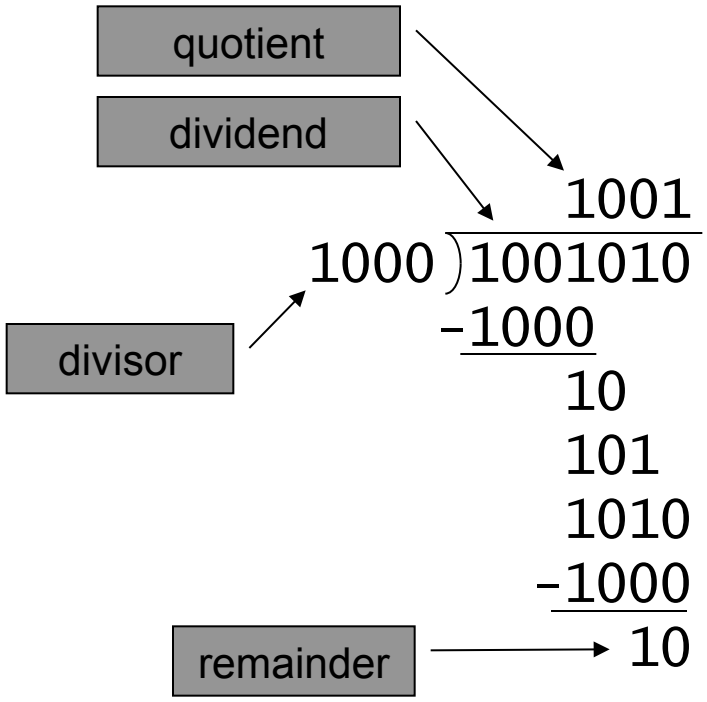
	1	1	0	0	0					Multiplicand (-8) with extra 1
	0	1	0	0	0					-(Multiplicand) (8) with extra 0
	0	0	0	1	0	0	0	0	0	1 Product after step 3
+	1	1	0	0	0					1Multiplicand
	1	1	0	1	0	0	0	0	0	1 Result of addition
	1	1	1	0	1	0	0	0	0	0 Final result

Ignore extra bits to get actual product

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

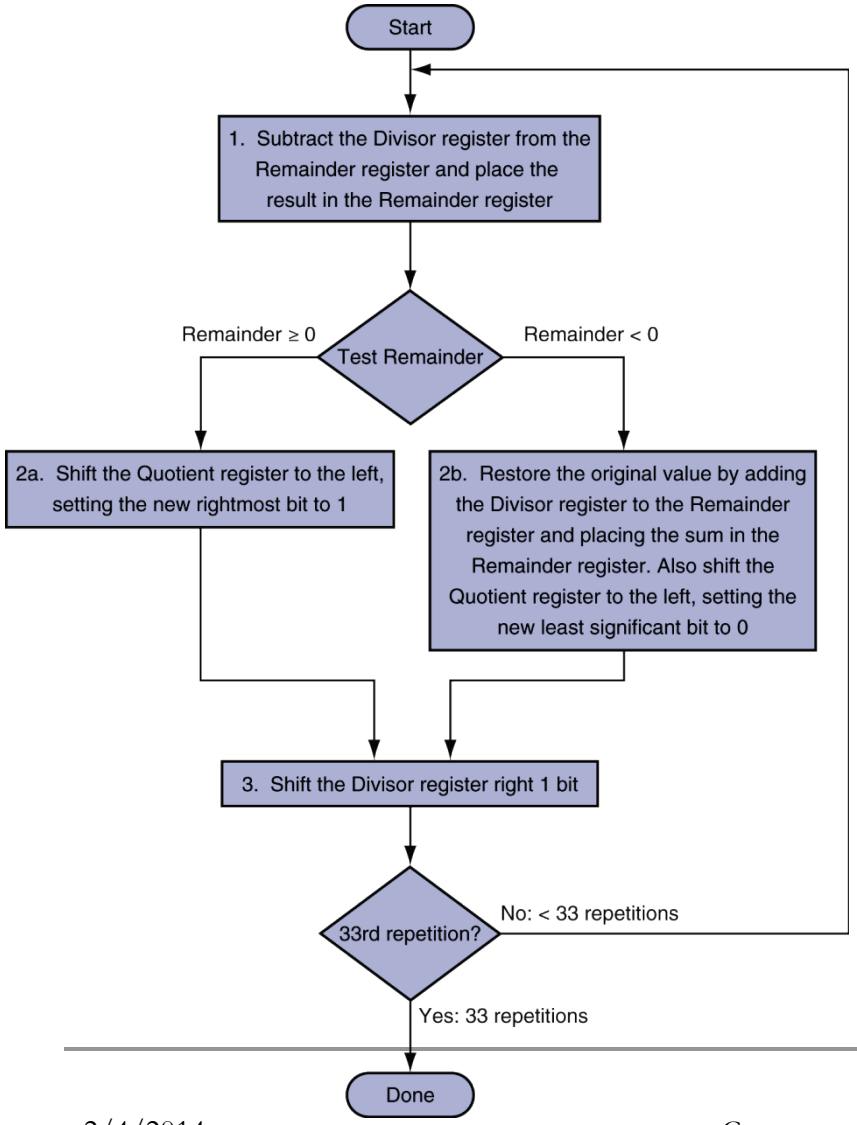
Division

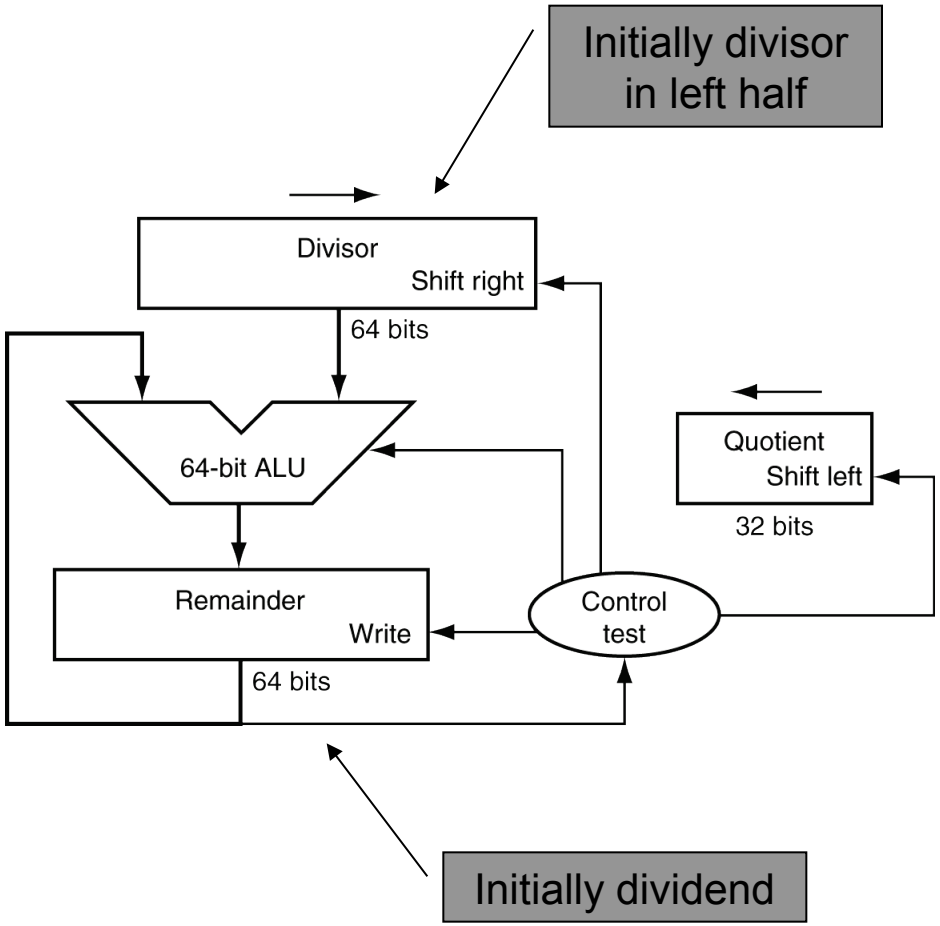


n-bit operands yield *n*-bit quotient and remainder

- 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

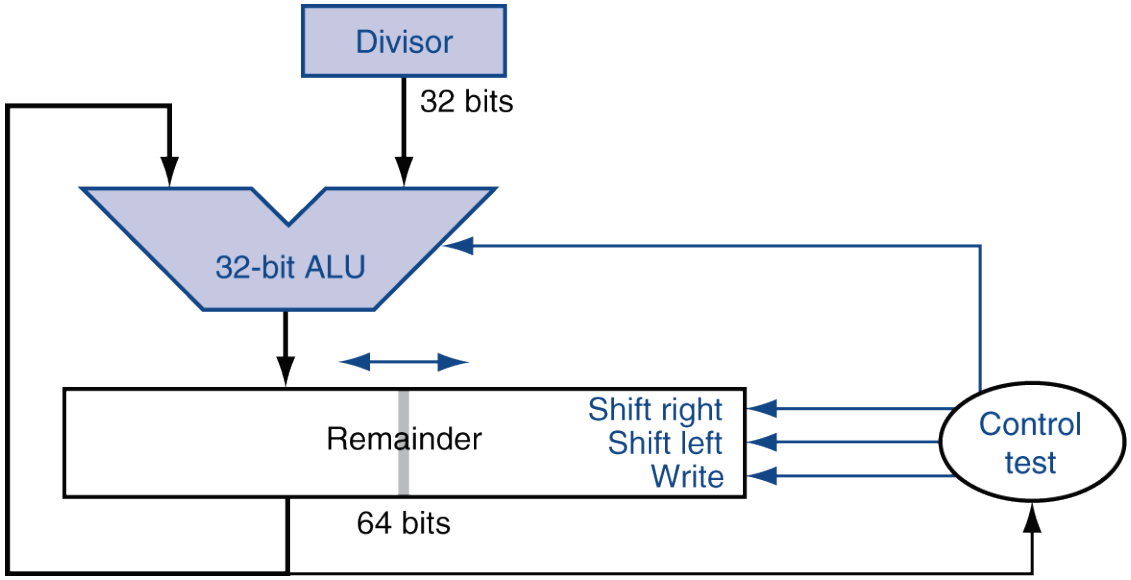
Division Hardware





Optimized Divider

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



Faster Division

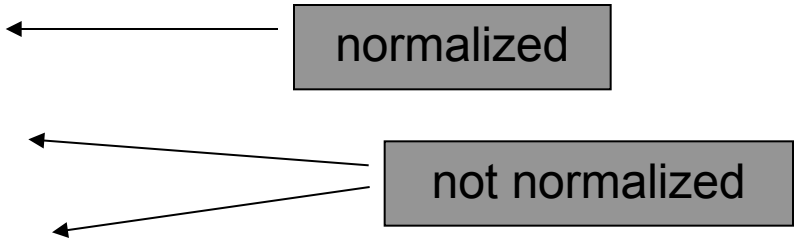
- Can't use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT devision) generate multiple quotient bits per step
  - Still require multiple steps

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}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

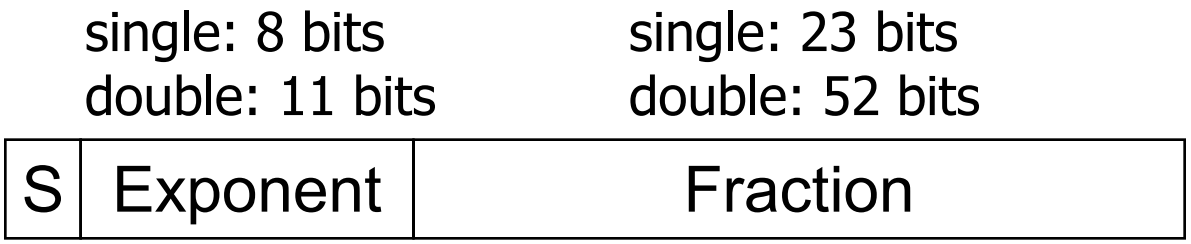


- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format



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

- 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
- Actual exponent = (encoded value) - bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023
- Encoded exponents 0 and 111 ... 111 reserved

FP characteristics

FP type	Min	Max	Precision
Single	Exponent = 1 → Actual exp = 1 – 127 = -126 Fraction = 0 → Significand = 1.0 **$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$**	Exponent = 254 → Actual exp = 254 – 127 = 127 Fraction = 111 ... 111 → Significand ≈ 2.0 **$\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$**	Approx. 2^{-23} → 6 decimal digits
Double	Exponent = 1 → Actual exp = 1 – 1023 = -1022 Fraction = 0 → Significand = 1.0 **$\pm 1.0 \times 2^{-1022}$** **$\approx \pm 2.2 \times 10^{-308}$**	Exponent = 2046 → Actual exp = 2046 – 1023 = 1023 Fraction = 111 ... 111 → Significand ≈ 2.0 **$\pm 2.0 \times 2^{+1023}$** **$\approx \pm 1.8 \times 10^{+308}$**	Approx. 2^{-54} → 16 decimal digits

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: 10111111101000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  1100000010100000…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 Addition

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

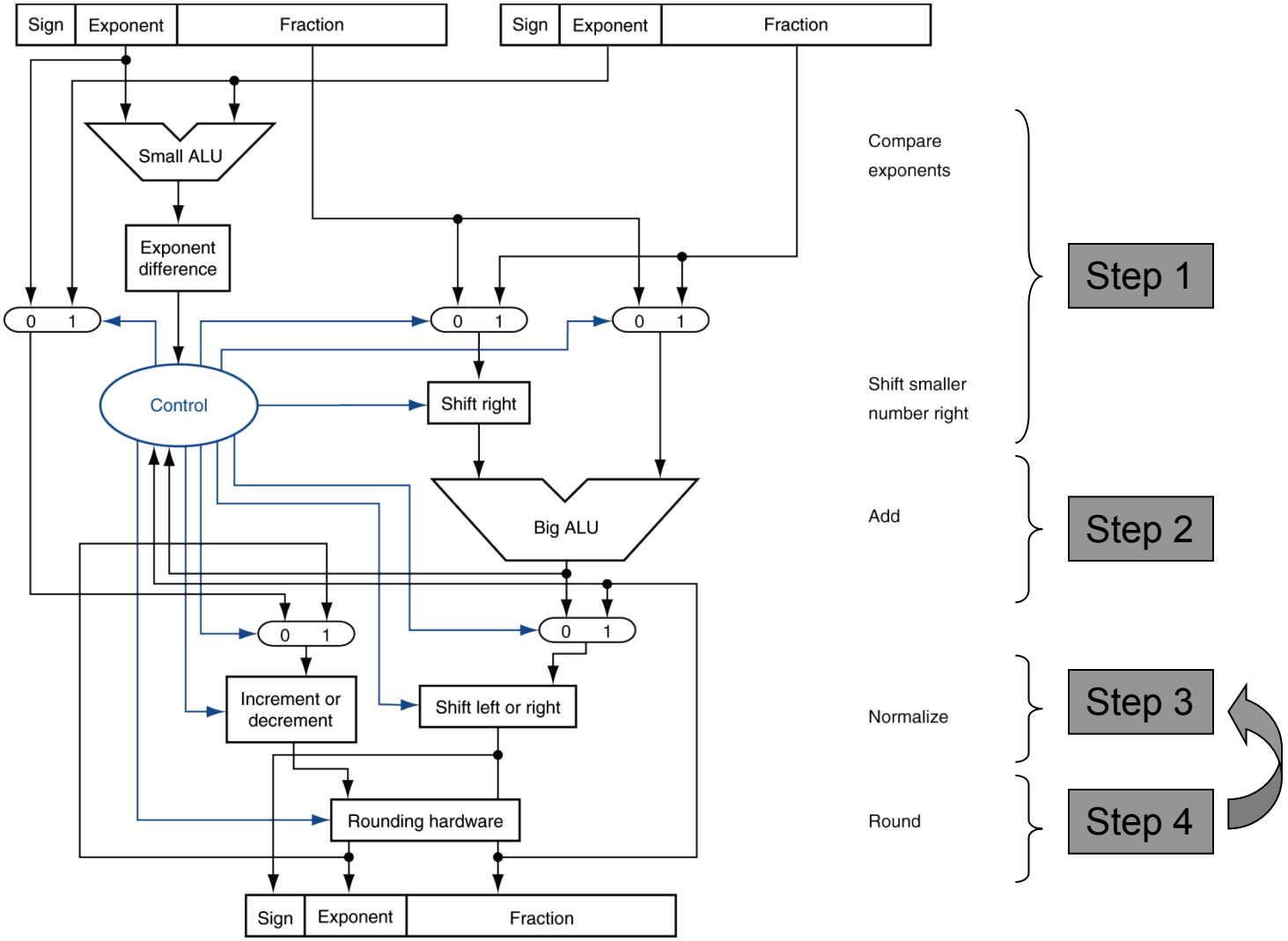
Floating-Point Addition

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

FP Adder Hardware

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

FP Adder Hardware



Floating-Point Multiplication

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

Floating-Point Multiplication

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

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
```

Final notes

- Next time:
  - Basic datapath and control design
- Announcements/reminders
  - HW 1 to be posted; due 2/6