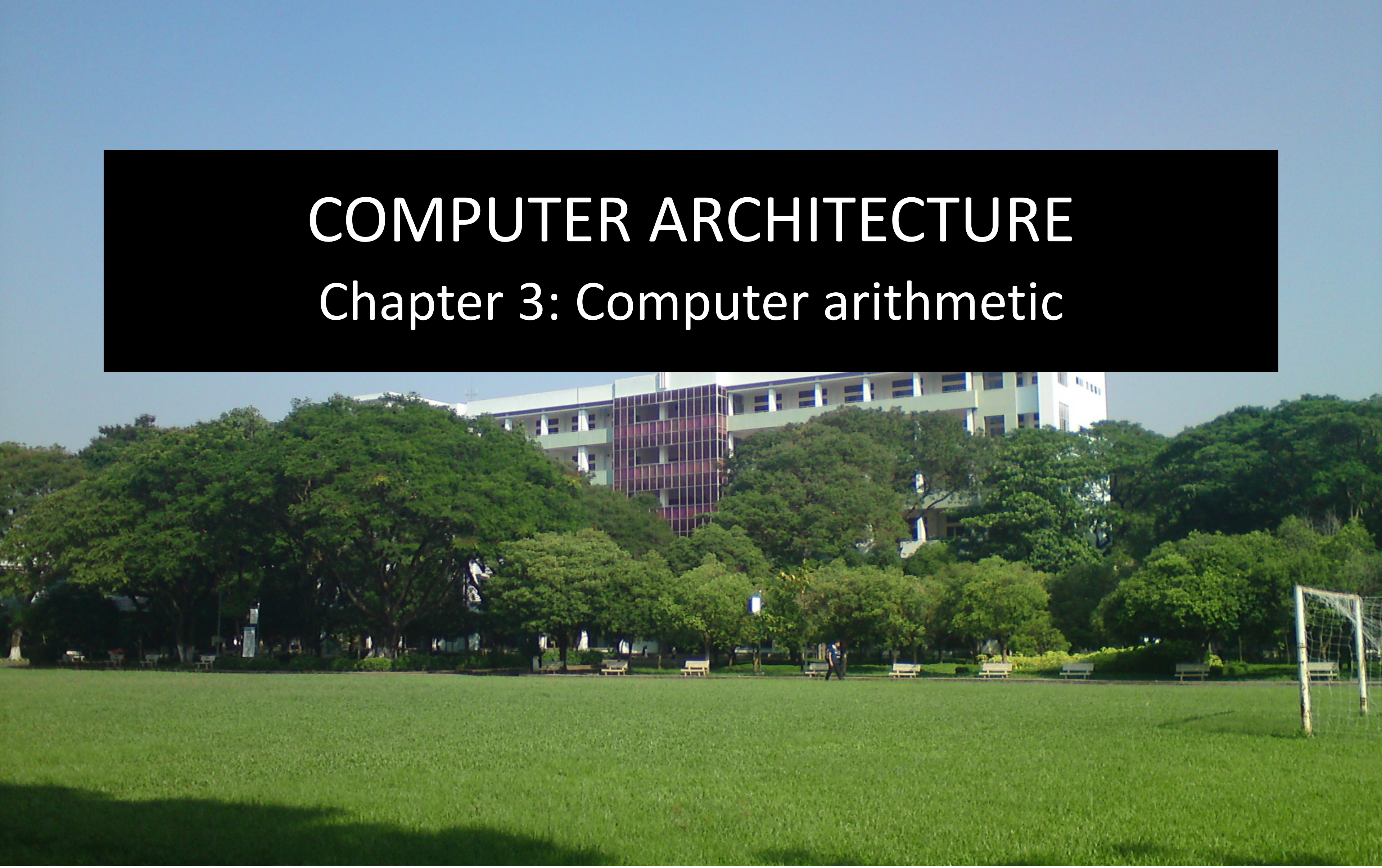
COMPUTER ARCHITECTURE

Chapter 3: Computer arithmetic

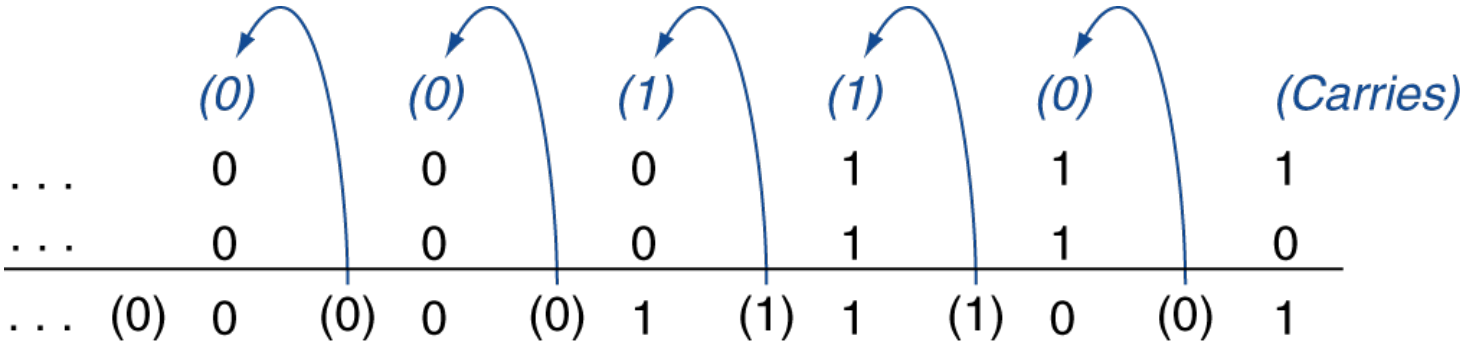


Outline

- Integer operations
  - Addition and subtraction
  - Multiplication and division
- Floating-point numbers
  - Representation
  - Operations and instructions

INTEGER OPERATIONS

Integer addition



- Example: $7_{10} + 6_{10} = 0111_2 + 0101_2$
- Overflow: 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

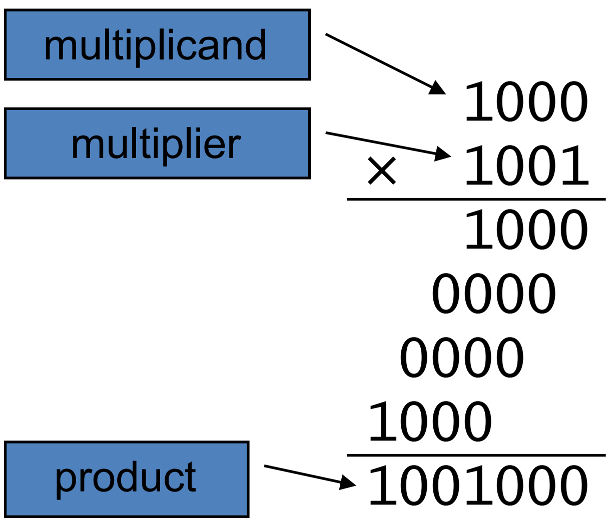
Integer subtraction

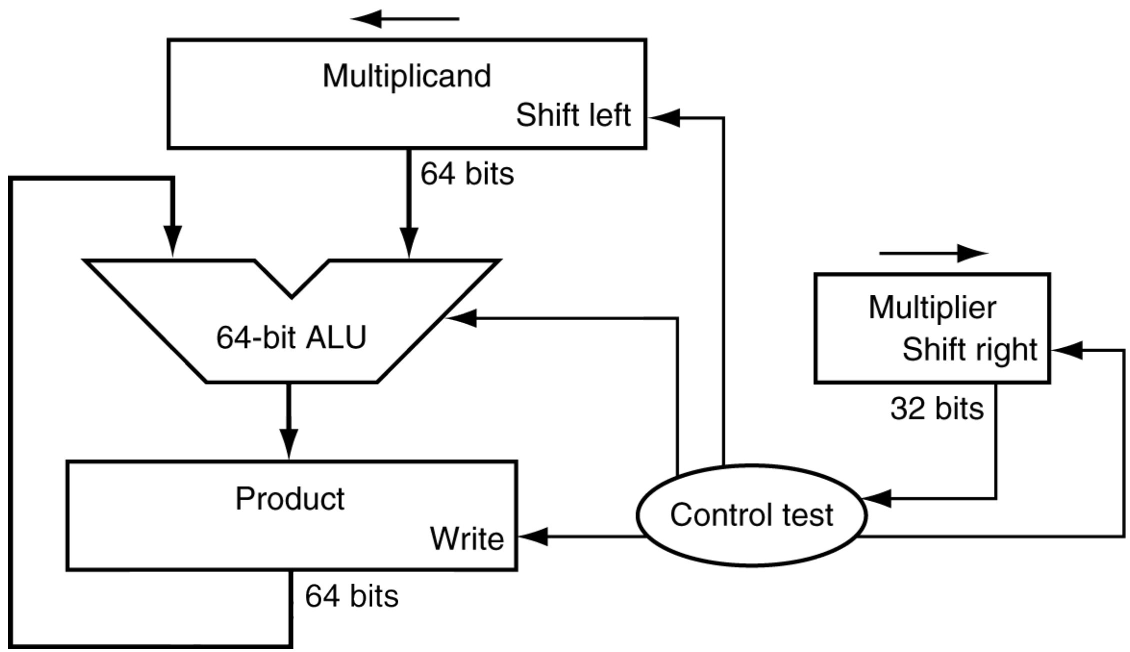
- Add negation (2’s complement) of the second operand
- Example: $7 - 6 = 7 + (-6) = 0111_2 + 1010_2 = 0001_2$
- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, **no overflow**
  - Subtracting +ve from –ve operand: $-7 - 6$
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand: $7 - (-6)$
    - Overflow if result sign is 1

Deal 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** (hardware)
    - 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

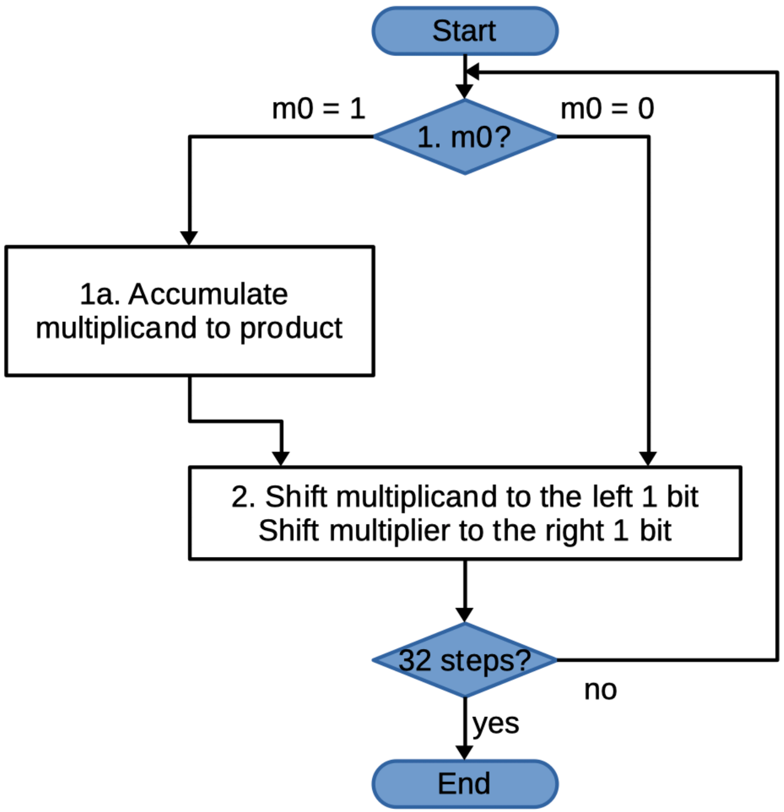
Hardware for multiplication

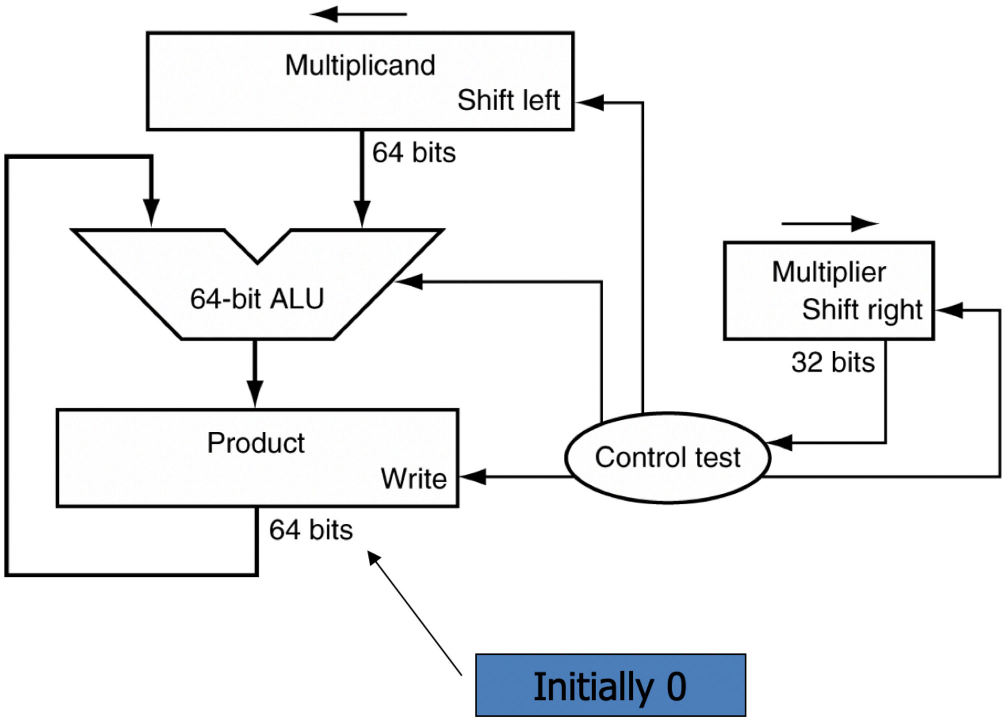




Length of product is the sum of operand lengths

Hardware operation





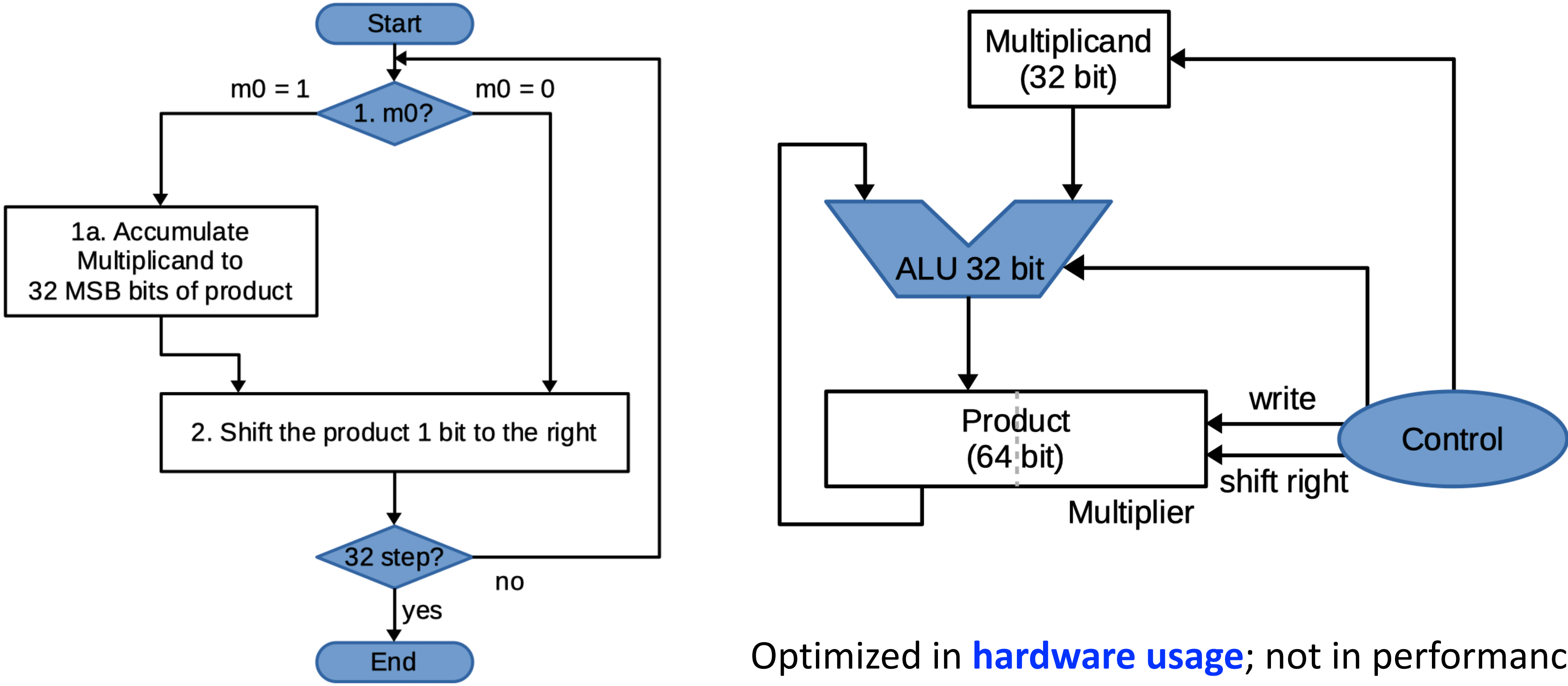
m0: LSB bit of the multiplier

Example

- Using 4-bit numbers, calculate $2_{10} \times 3_{10} = 0010_2 \times 0011_2$

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 hardware



Optimized in **hardware usage**; not in performance

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
  - mul rd, rs, rt
    - **ONLY** least-significant 32 bits of product → rd

Division

Dividend 1001010_{10} | 1000_{10} **Divisor**

-1000 | 1001_{10} **Quotient**

10

101

1010

-1000

Reminder 10_{10}

1001010_2 | 1000_2

-1000 | 1001_2

10

101

1010

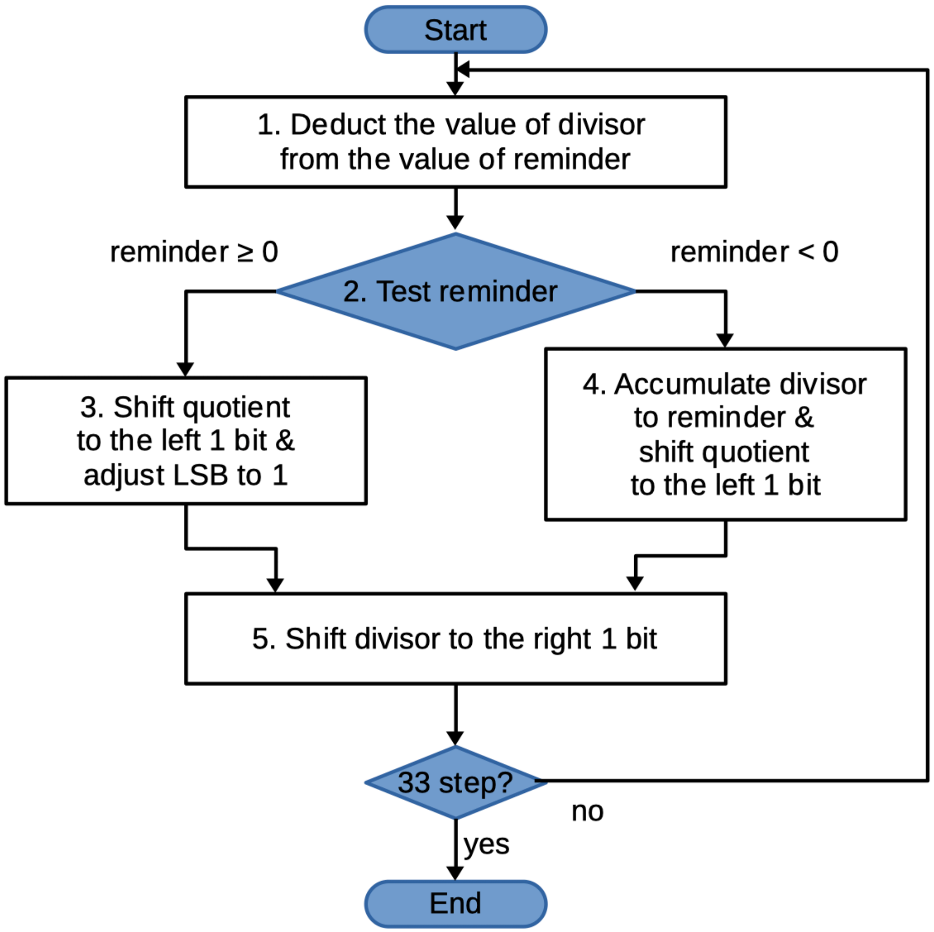
-1000

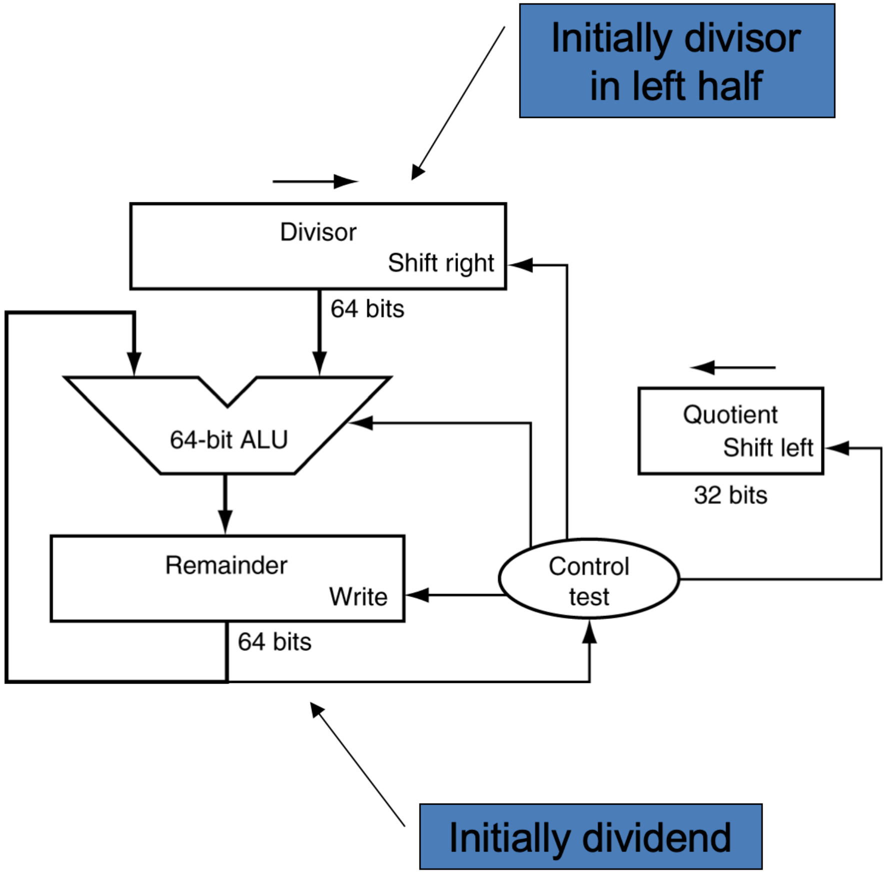
10_2

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

- Long division approach
  - if divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring divisor
  - 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

Hardware for division



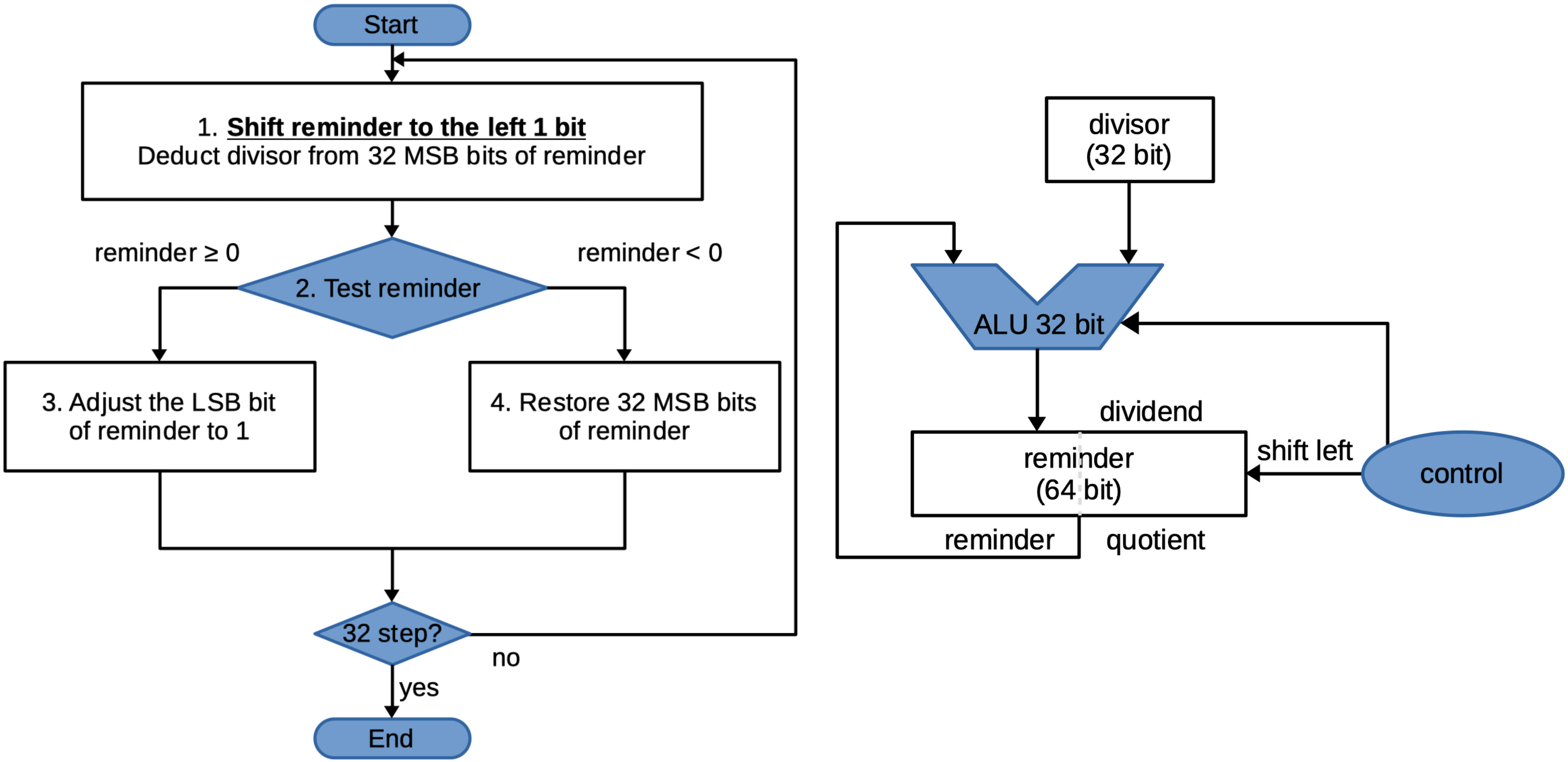


Example

- Using 4-bit numbers, calculate $7_{10} \div 2_{10} = 0111_2 \div 0010_2$

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, 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 ⟹ +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 ⟹ +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 ⟹ 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 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Optimized hardware



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
    - What are values in HI/LO if divisor is 0?
- Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

FLOATING POINT NUMBERS

Floating point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - Normalized: -2.54×10^{56}
  - Not normalized: 0.002×10^{-4}; 987.6×10^{3}
- In binary
  - $\pm 1.xxxx_2 \times 2^{yyyy}$
- In ANSI: float or double

Floating point standard

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

IEEE-754 format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$X_{10} = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

- **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: 0 ≤ |Fraction| < 1.0
- **Exponent** = actual exponent + **Bias**
  - Ensures exponent is unsigned
  - Single: **Bias** = 127; Double: **Bias** = 1023

Example

- **Question**: What is the decimal value of the floating point number 0x414C0000?
- **Answer**:
  - 0x414C0000 $\Rightarrow$ single precision
    - S = 0;
    - Exponent = $1000\_0010_2 = 130$;
    - F = $100\_1100\_0000\_\ldots\_0000_2 = 2^{-1} + 2^{-4} + 2^{-5} = 0.59375$
  - $X = (-1)^0 \times (1 + 0.59375) \times 2^{130-127} = 12.75$

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

Convert to IEEE-754

- **Step 1**: Decide S (1: negative; 0: positive)
- **Step 2**: Decide Fraction
  – Convert the integer part to Binary
  – Convert the fractional part to Binary
  – Adjust the integer and fractional parts according the Significand format (1.xxx)
- **Step 3**: Decide exponent

Example

- **Question**: what is the IEEE-754 representation of 12.75?
- **Answer**:
  - S = 0;
  - $12.75 = 1100.11_2 = 1.10011 \times 2^3$
  - Exponent = $3 + 127 = 130$
  - Fraction: $100\_1100\_0000\_0000\_0000\_0000_2$
  - $12.75 = 0x414C0000_{IEEE-754}$

Floating point addition

- **Question**: how to add two 4-digit decimal floating point numbers:

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

- **Answer**: do the following step

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}

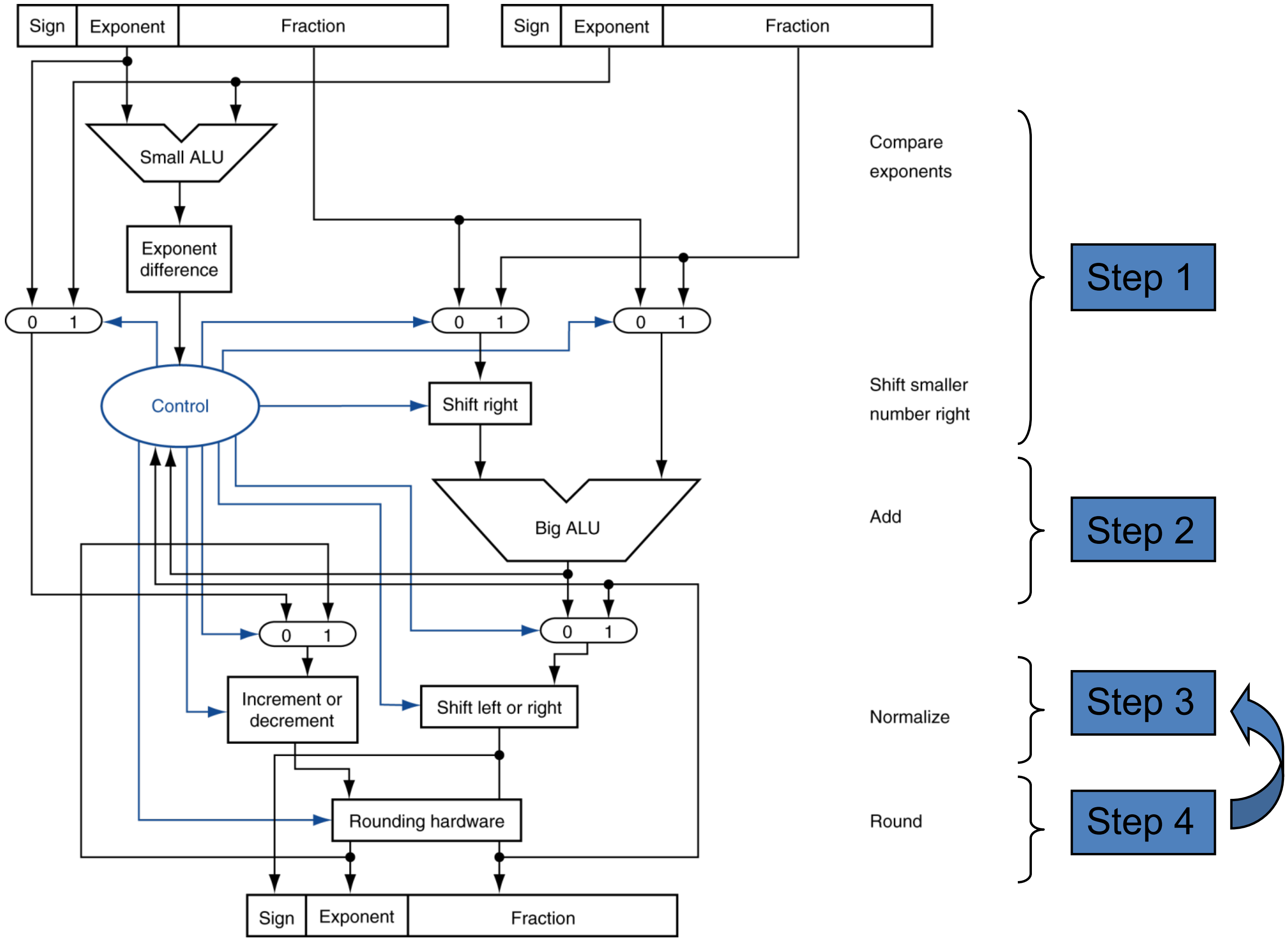
Floating point addition

- 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

Floating pointer adder hardware



Floating point multiplication

- **Question**: how to multiply two 4-digit decimal numbers:

$$1.110 \times 10^{10} \times 9.200 \times 10^{-5}$$

- Answer: do the following steps

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

$$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} = (0.5 \times -0.4375)$$

- Add exponents
  - Unbiased: $-1 + -2 = -3$
  - **Biased**: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
  - $-1.1102 \times 2^{-3} = -0.21875$

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,…
    - Odd-number registers: right half of 64-bit floating-point numbers
- 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

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

FP machine instructions

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

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
- Who Cares About FP Accuracy?

Concluding remarks

- 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

The end