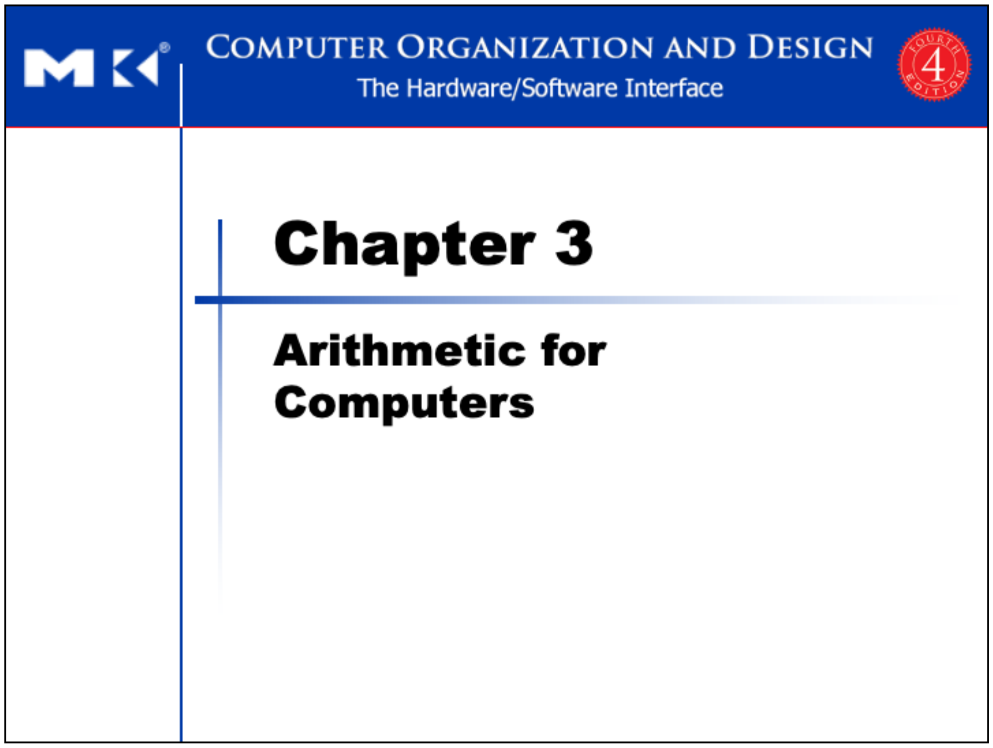
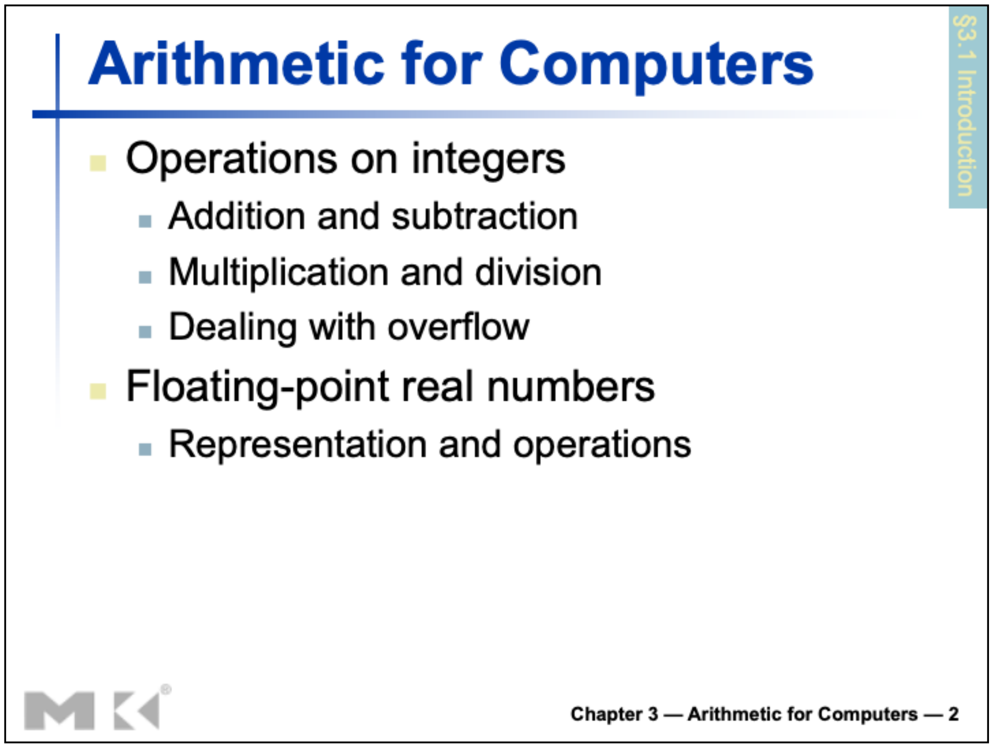


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

Chapter 3

Arithmetic for Computers

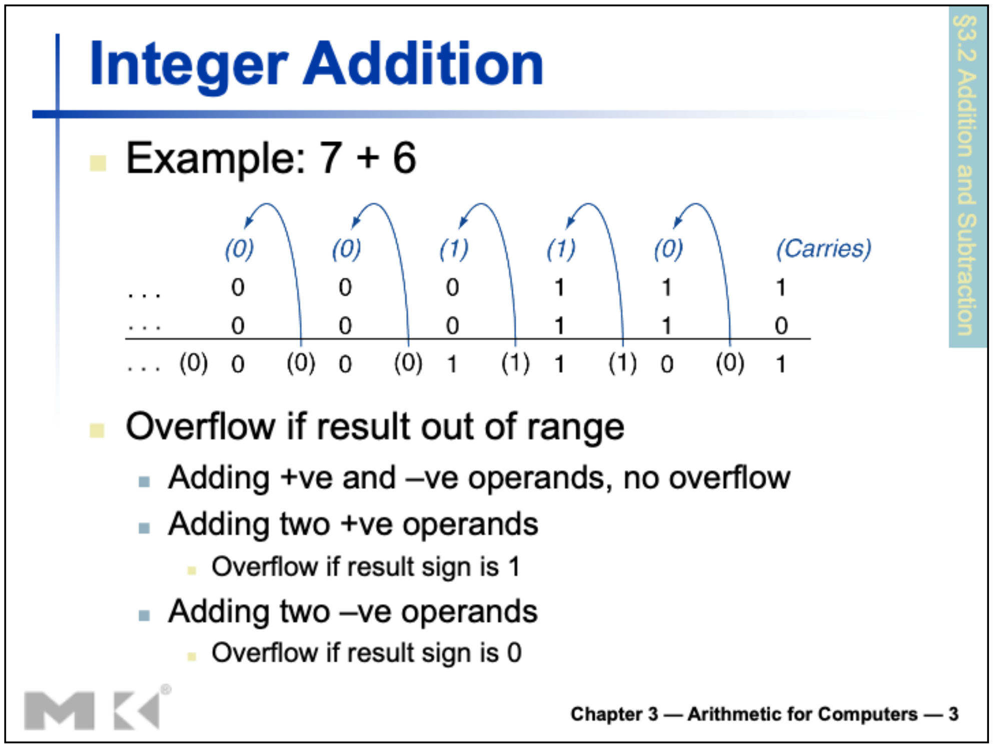


Arithmetic for Computers

§3.1 Introduction

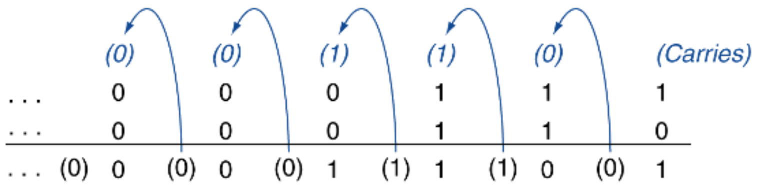
- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Chapter 3 — Arithmetic for Computers — 2



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

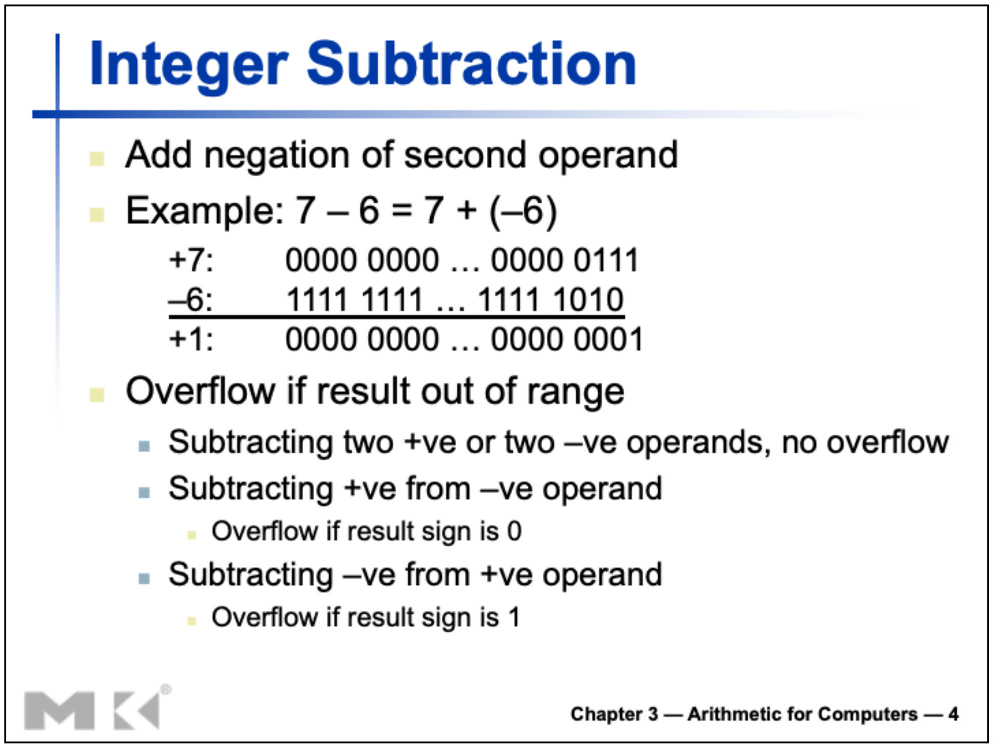
Chapter 3 — Arithmetic for Computers — 3

§3.2 Addition and Subtraction

- The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 +1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

- When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, -10 + 4 = -6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

- Knowing when overflow cannot occur in addition and subtraction is all well and good, but how do we detect it when it *does* occur? Clearly, adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed. The lack of a 33rd bit means that when overflow occurs, the sign bit is set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This spurious sum means a carry out occurred into the sign bit.



Integer Subtraction

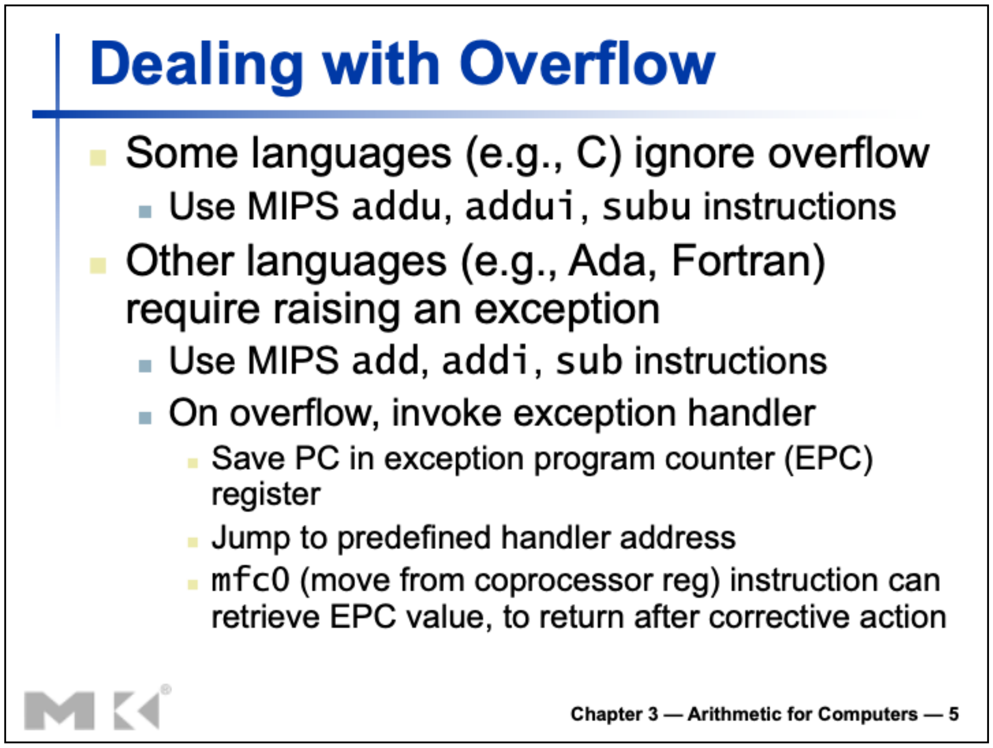
- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

```
+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

Chapter 3 — Arithmetic for Computers — 4

- There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: when the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $c - a = c + (-a)$ because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.



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

Chapter 3 — Arithmetic for Computers — 5

What about overflow with unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others.

The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

■ Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.

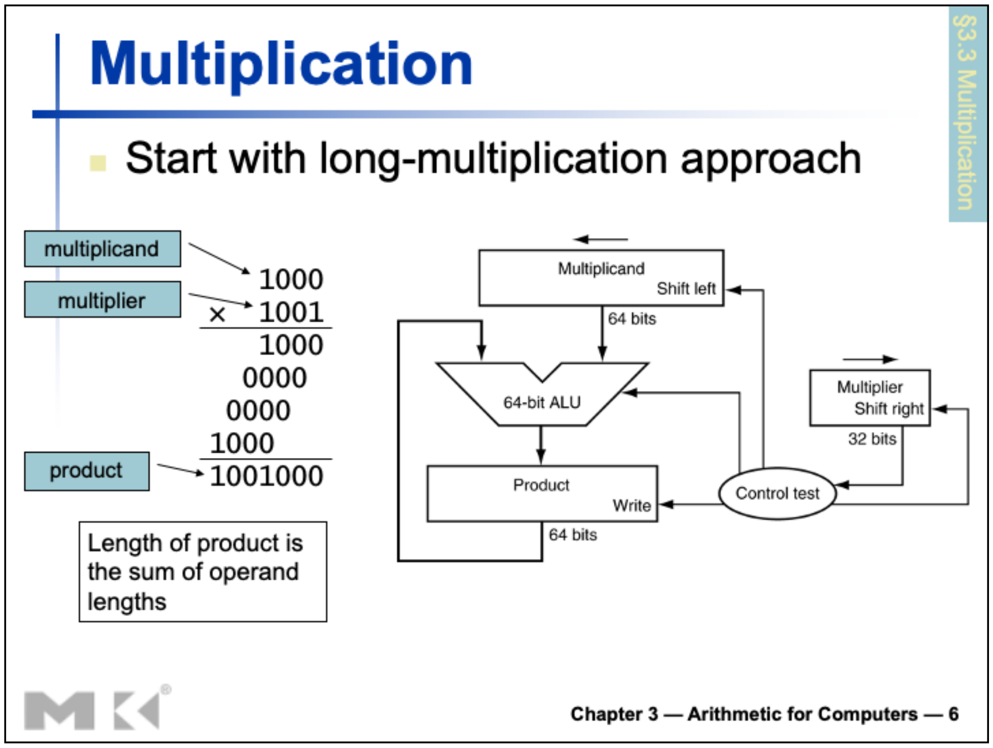
■ Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do *not* cause exceptions on overflow.

MIPS detects overflow with an **exception**, also called an **interrupt** on many computers. An exception or interrupt is essentially an unscheduled procedure call.

The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

MIPS includes a register called the *exception program counter* (EPC) to contain the address of the instruction that caused the exception.

The instruction *move from system control* (mfc0) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.



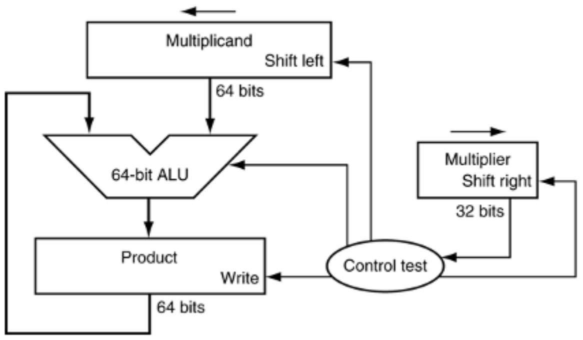
Multiplication

§3.3 Multiplication

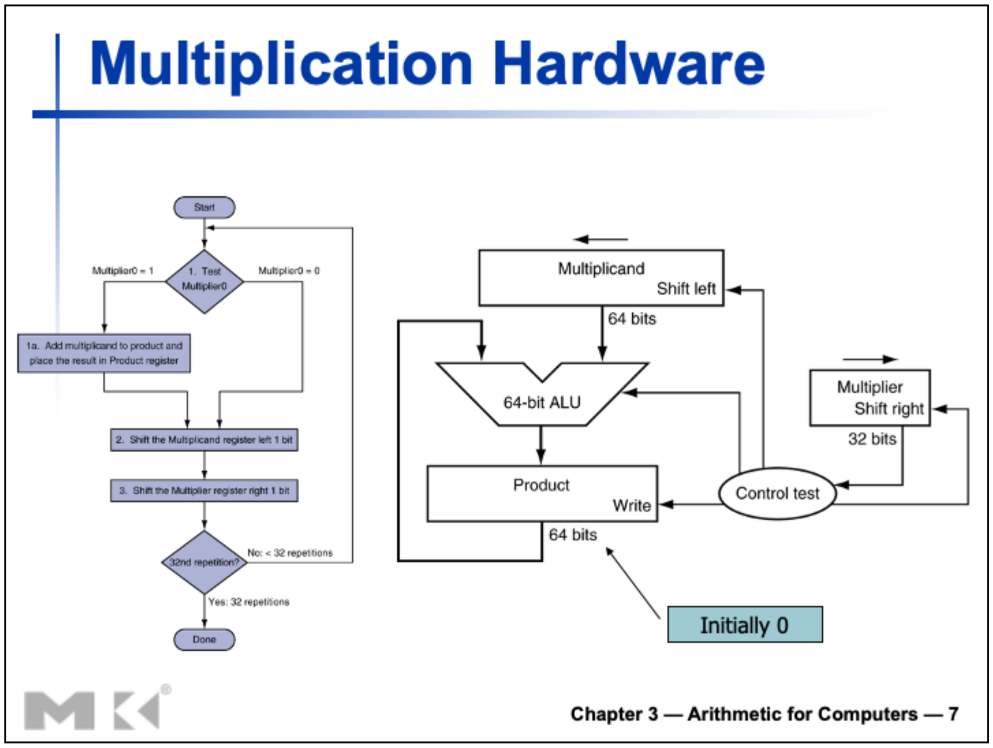
- Start with long-multiplication approach

```
multiplicand      1000
multiplier      × 1001
                  ----
                  1000
                 0000
                0000
               1000
               -------
product        1001000
```

Length of product is the sum of operand lengths



Chapter 3 — Arithmetic for Computers — 6

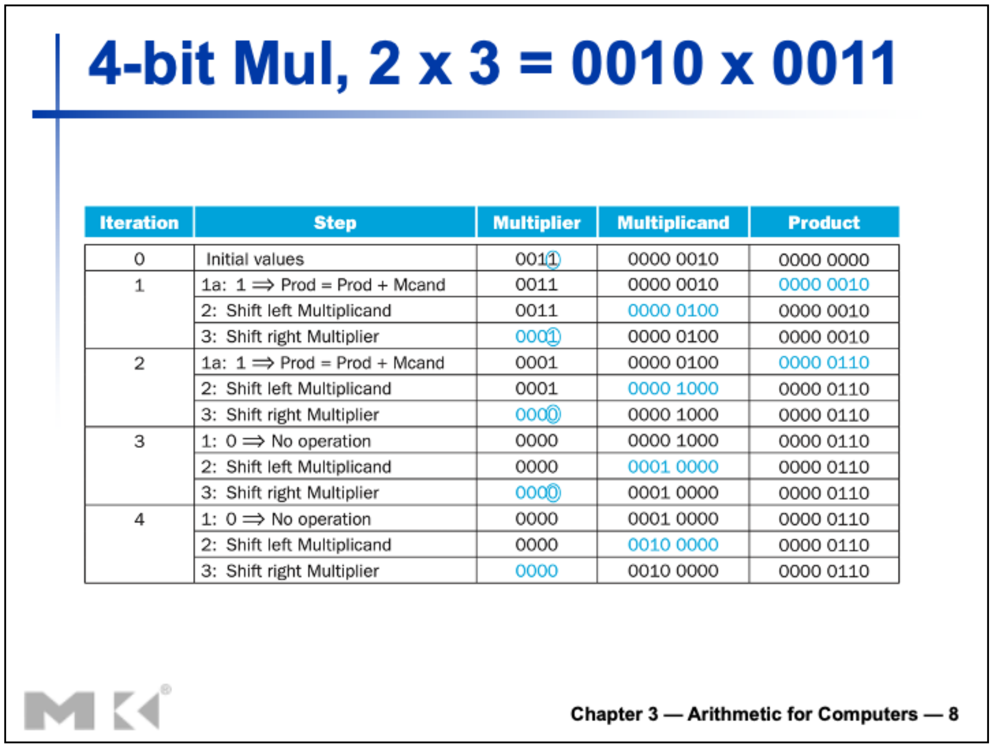


Multiplication Hardware

Chapter 3 — Arithmetic for Computers — 7

- Let’s assume that the multiplier is in the 32-bit Multiplier register and that the 64- bit Product register is initialized to 0.
- From the paper-and-pencil example above, it’s clear that we will need to move the multiplicand left one digit each step, as it may be added to the intermediate products. Over 32 steps, a 32-bit multiplicand would move 32 bits to the left. Hence, we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and zero in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

- Figure shows the three basic steps needed for each bit
- The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product. If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers.
- If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

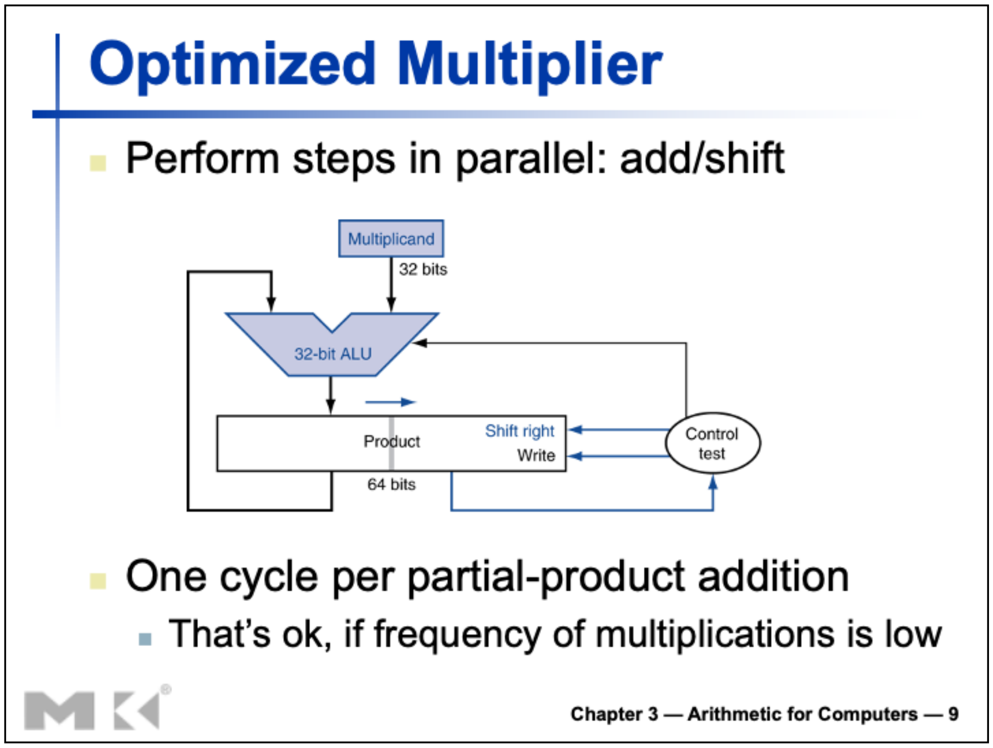


4-bit Mul, 2 x 3 = 0010 x 0011

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

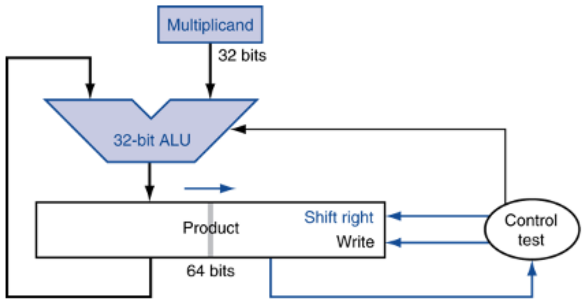
Chapter 3 — Arithmetic for Computers — 8

The bit examined to determine the next step is circled in color.



Optimized Multiplier

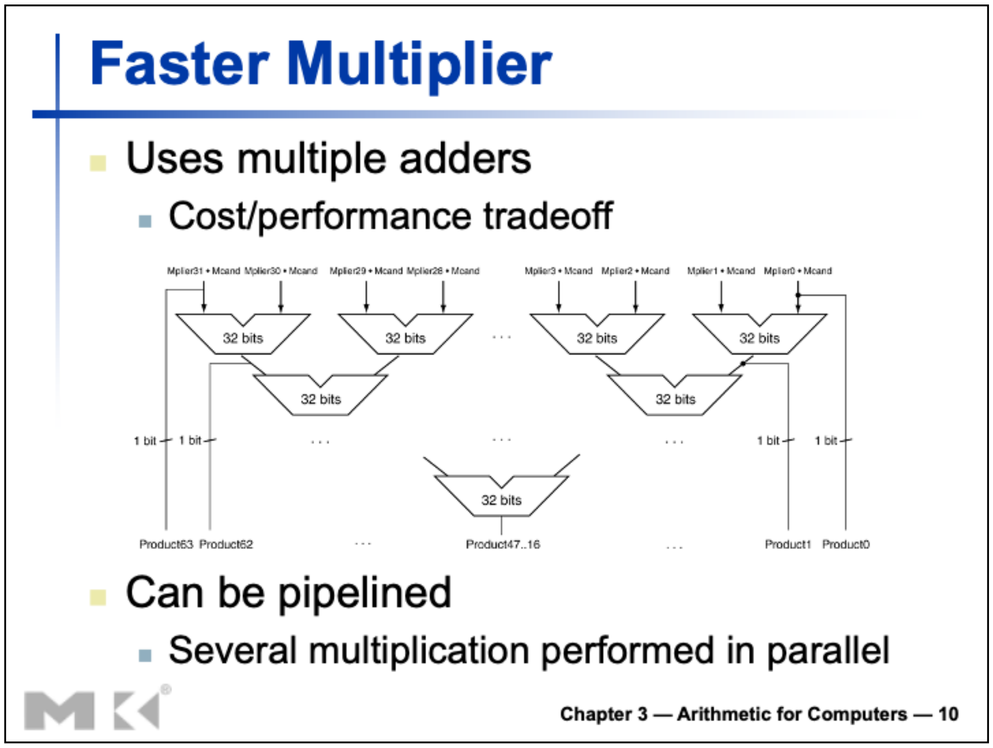
- Perform steps in parallel: add/shift



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

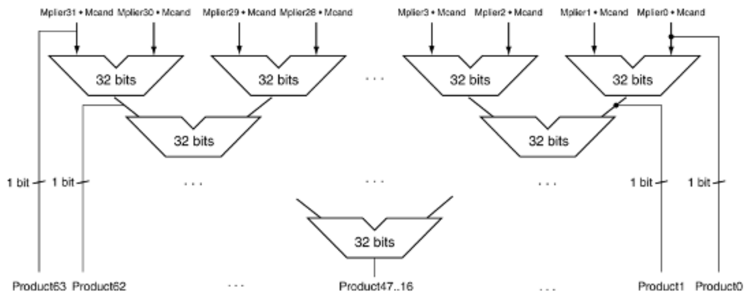
Chapter 3 — Arithmetic for Computers — 9

- Refined version of the multiplication hardware: Compare with the first version, the Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from first version before.)



Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



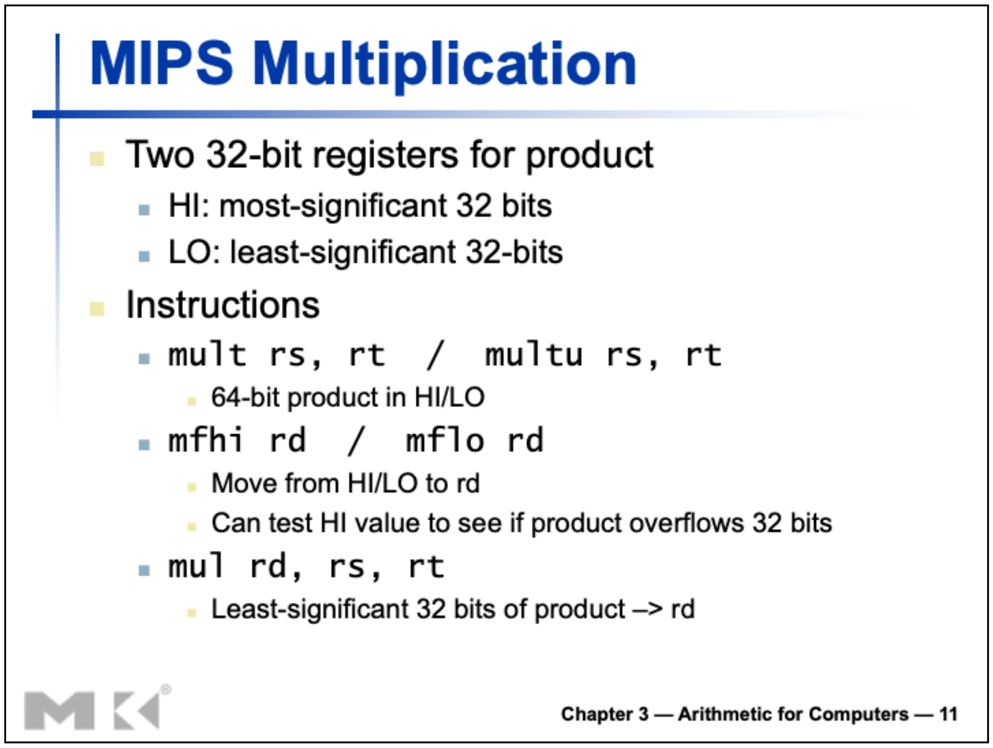
- Can be pipelined
  - Several multiplication performed in parallel

Chapter 3 — Arithmetic for Computers — 10

- **Moore's Law** has provided so much more in resources that hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits. Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.

- Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder. A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high.

- An alternative way to organize these 32 additions is in a parallel tree, as the figure shows. Instead of waiting for 32 add times, we wait just the log2 (32) or five 32-bit add times.

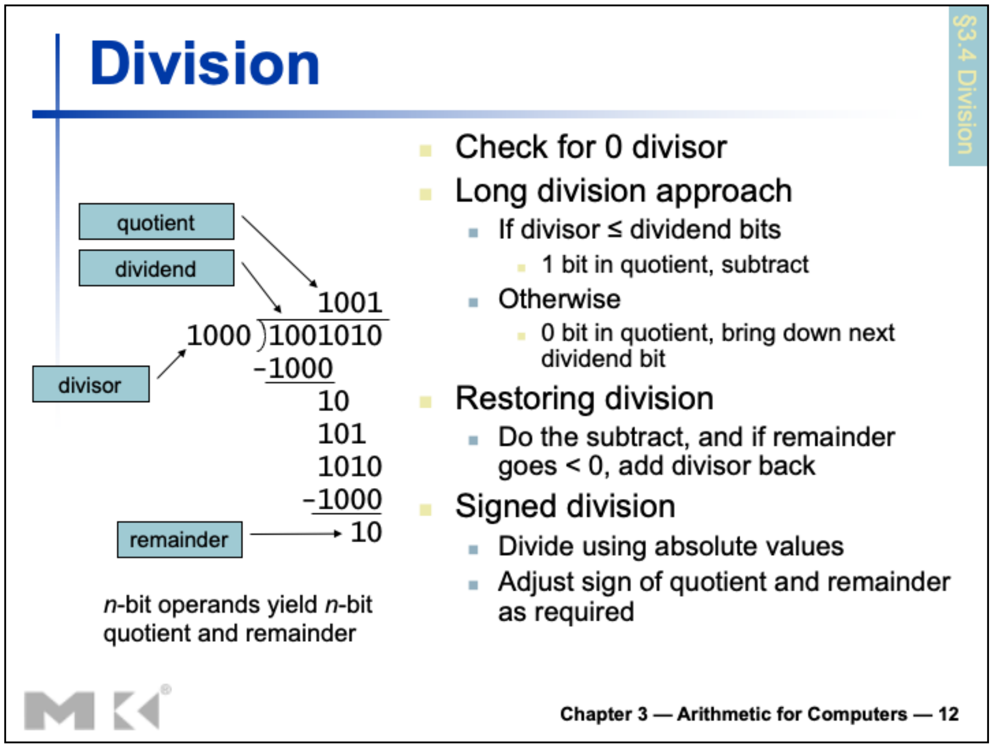


MIPS Multiplication

- 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

Chapter 3 — Arithmetic for Computers — 11

- MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*.
- To produce a properly signed or unsigned product, MIPS has two instructions: multiply (mult) and multiply unsigned (multu).
- Note that the result of the multiplication of two 32-bit numbers yields a 64-number.
- The 32 most significant bits will be held in HI special register (accessible by mfhi instruction) and the 32 least significant bits will be held in LO special register (accessible by mflo instruction):
- Sample code:
  - mult $a0, $a1
  - mfhi $a2 # 32 most significant bits of multiplication to $a2
  - mflo $v0 # 32 least significant bits of multiplication to $v0



Division

quotient
dividend
divisor
remainder
1001
1000)1001010
-1000
10
101
1010
-1000
10
§3.4 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

Chapter 3 — Arithmetic for Computers — 12

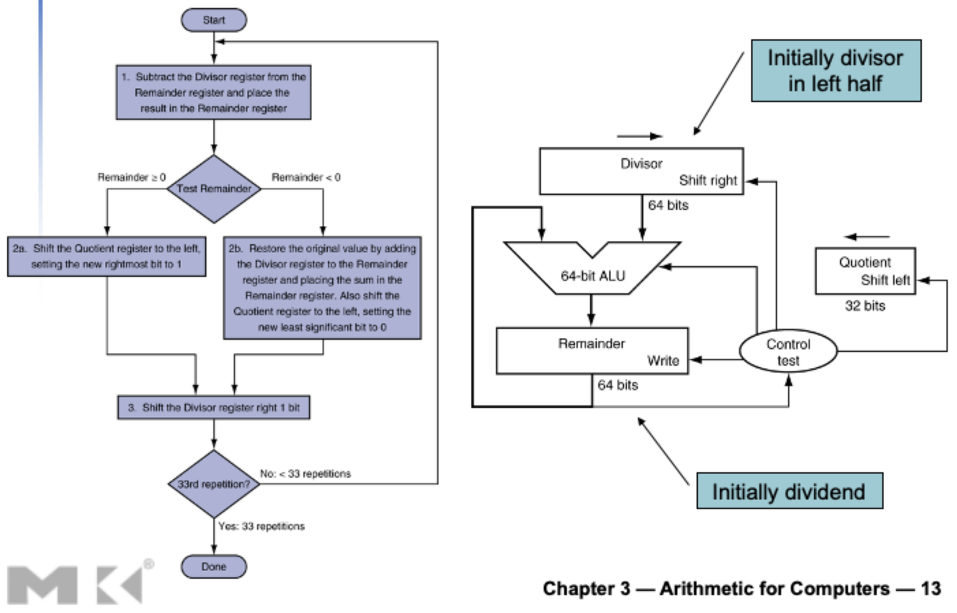
- Divide's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, are accompanied by a second result, called the **remainder**.

- Dividend = Quotient x Divisor + Remainder

- The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses only the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the dividend: it's either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.

- signed division algorithm negates the quotient if the signs of the operands (divisor & dividend) are opposite and makes the sign of the nonzero remainder match the dividend.

Division Hardware

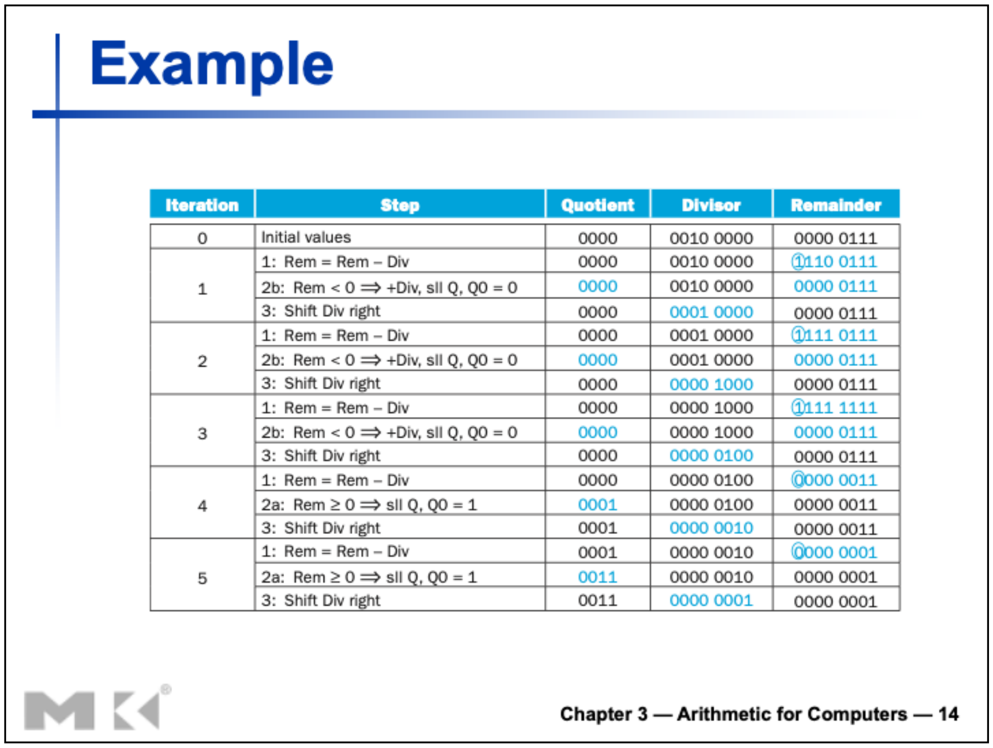

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done
Initially divisor in left half
Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
64 bits
Control test
Initially dividend

Chapter 3 — Arithmetic for Computers — 13

- Right - shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

- Left - shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend.
- It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction.
- If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a).
- If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b).
- The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

In short, If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.



Example

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

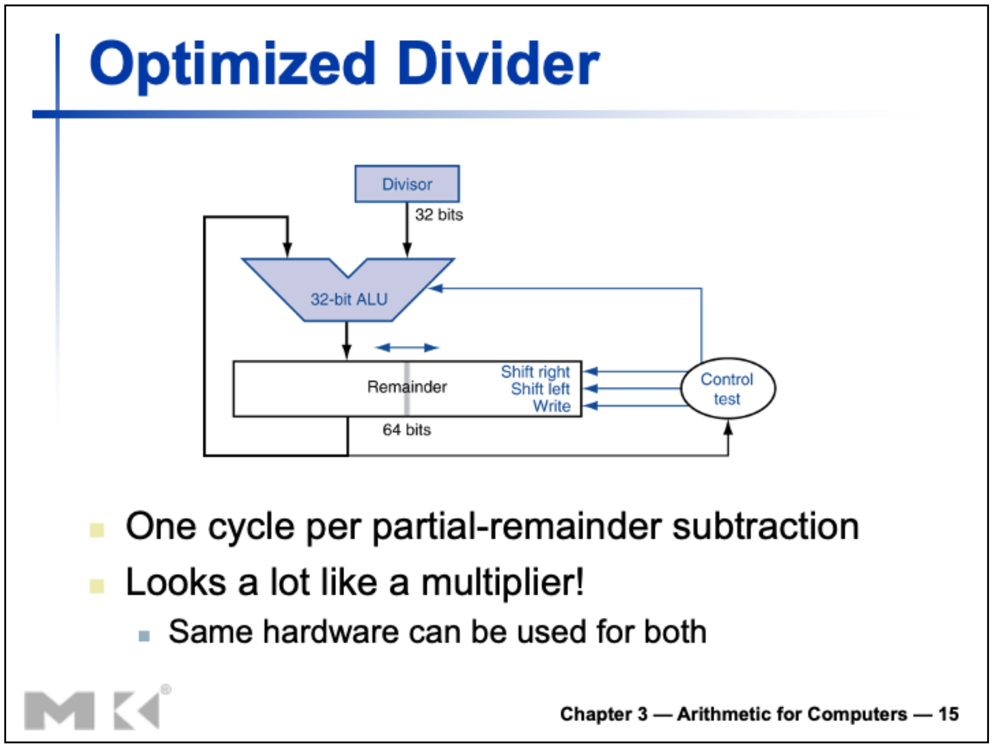
Chapter 3 — Arithmetic for Computers — 14

Division example using the algorithm.

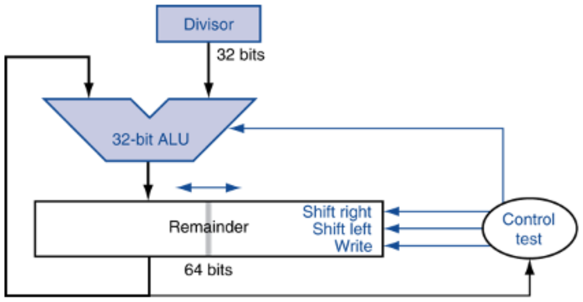
Divisor: 0010

Dividend: 0000 0111 (stored in reminder register)

The bit examined to determine the next step is circled in color.



Optimized Divider



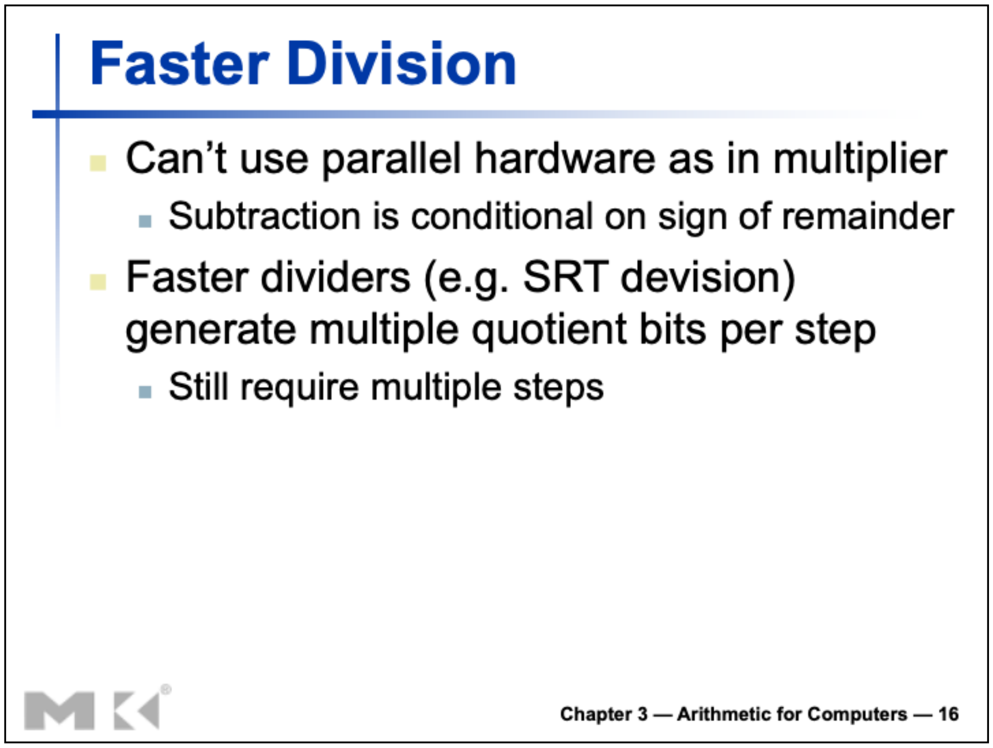
- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Chapter 3 — Arithmetic for Computers — 15

The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits.

Compared to the previous hardware, the ALU and Divisor registers are halved and the remainder is shifted left.

This version also combines the Quotient register with the right half of the Remainder register.



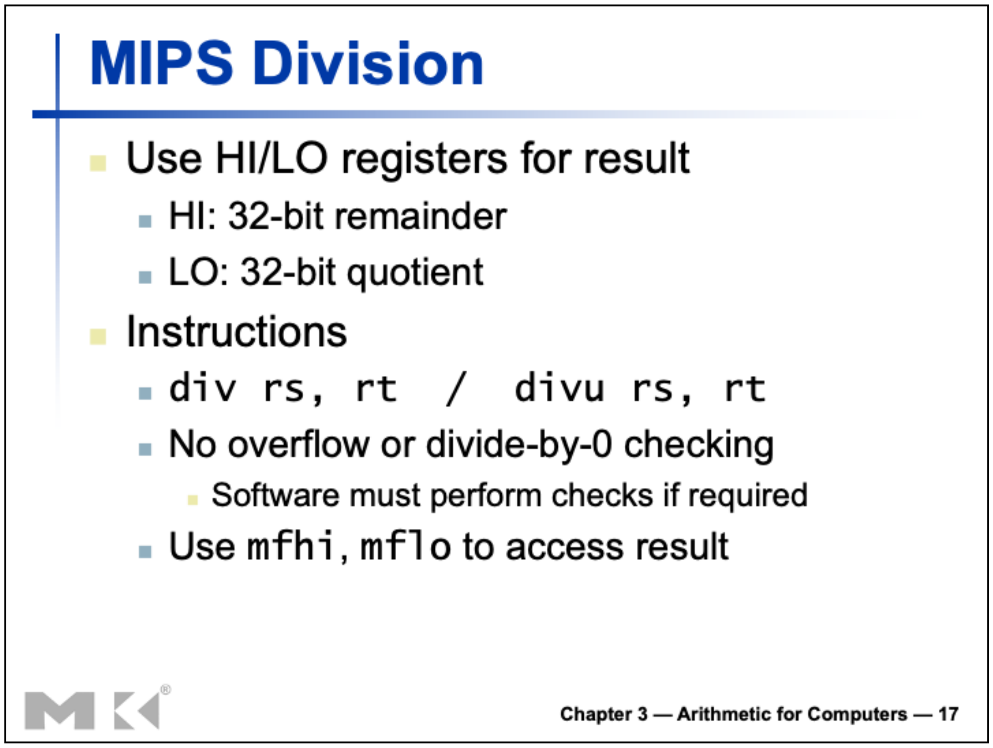
Faster Division

- 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

Chapter 3 — Arithmetic for Computers — 16

-**Moore's Law** applies to division hardware as well as multiplication, so we would like to be able to speed up division by throwing hardware at it. We used many adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

- There are techniques to produce more than one bit of the quotient per step. The *SRT division* technique tries to **predict** several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong predictions. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division, there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step.
- The accuracy of this fast method depends on having proper values in the lookup table. The fallacy on page 231 in Section 3.9 shows what can happen if the table is incorrect.



MIPS Division

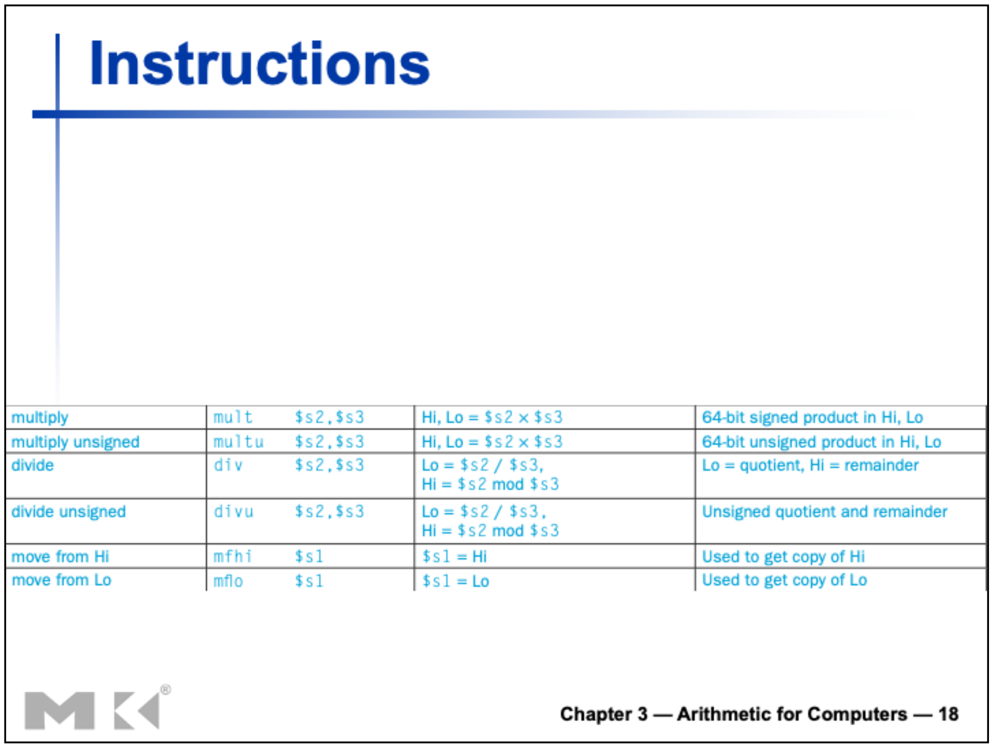
- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt / divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

Chapter 3 — Arithmetic for Computers — 17

MIPS has two instructions: *divide* (div) and *divide unsigned* (divu).

The MIPS assembler allows divide instructions to specify three registers, generating the mflo or mfhi instructions to place the desired result into a general-purpose register.

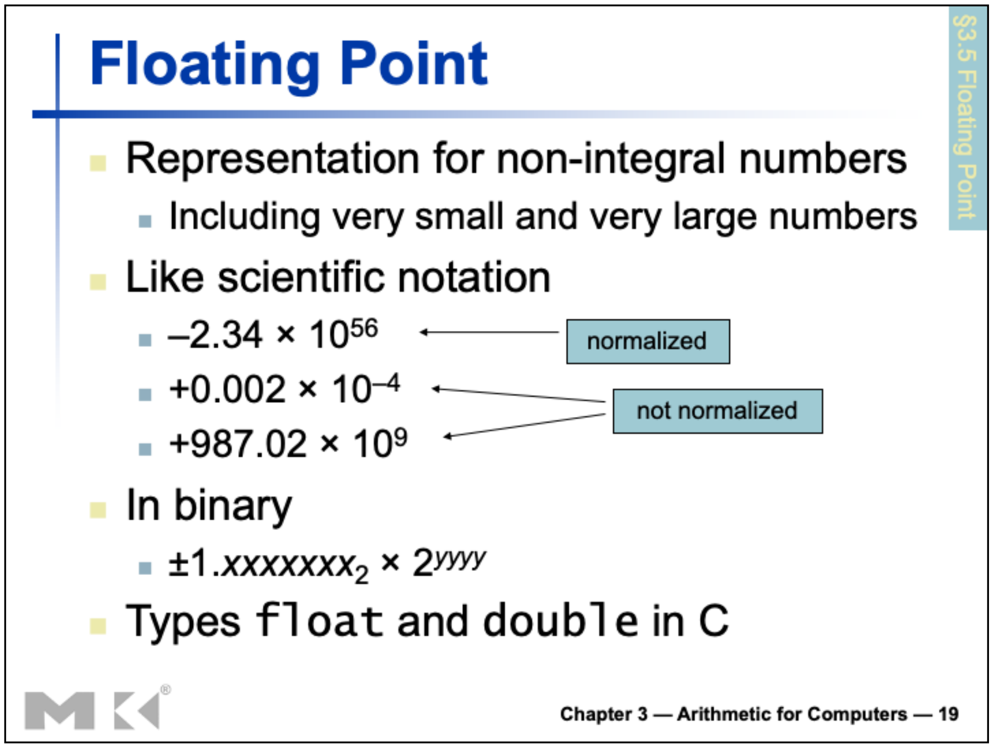
Hi contains the remainder, and Lo contains the quotient after the divide instruction completes.



Instructions

multiply	`mult $s2,$s3`	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
multiply unsigned	`multu $s2,$s3`	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
divide	`div $s2,$s3`	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
divide unsigned	`divu $s2,$s3`	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
move from Hi	`mfhi $s1`	$s1 = Hi	Used to get copy of Hi
move from Lo	`mflo $s1`	$s1 = Lo	Used to get copy of Lo

Chapter 3 — Arithmetic for Computers — 18



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

§3.5 Floating Point
Chapter 3 — Arithmetic for Computers — 19

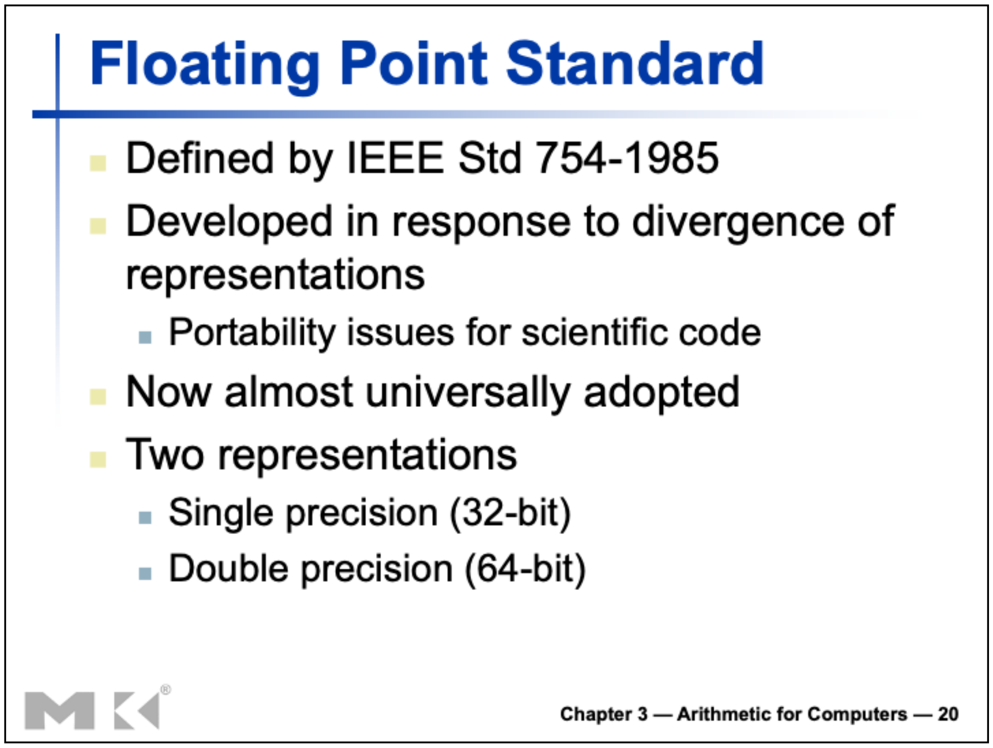
- For example, decimal 1234.567 is normalized as 1.234567×10^3 by moving the decimal point so that only one digit appears before the decimal. The exponent expresses the number of positions the decimal point was moved left (positive exponent) or moved right (negative exponent).

- Similarly, the floating-point binary value 1101.101 is normalized as 1.101101×2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3. Here are some examples of normalizations:

Binary || Value	**Normalized As (mantissa). ||**	**Exponent**
1101.101	1.101101	3
.00101	1.01	-3
1.0001	1.0001	0
10000011.0	1.0000011	7

You may have noticed that in a normalized mantissa, the digit 1 always appears to the left of the decimal point. In fact, the leading 1 is omitted from the mantissa's actual storage because it is redundant.

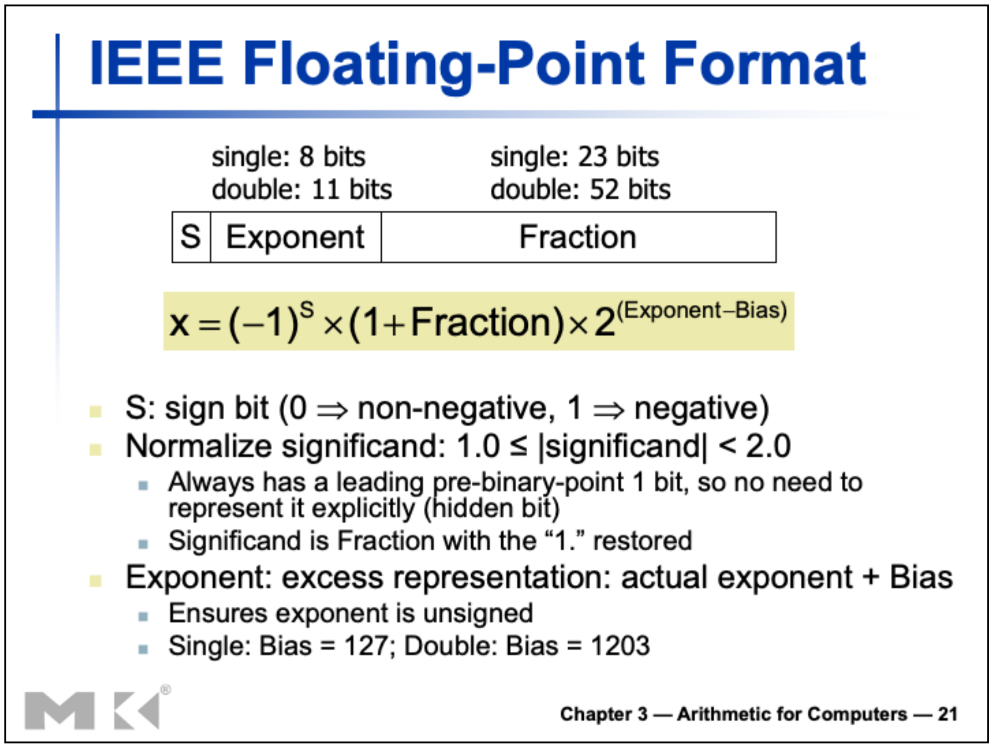
More details http://cstl-csm.semo.edu/xzhang/Class%20Folder/CS280/Workbook_HTML/FLOATING_t ut.htm#:~:text=The%20sign%20of%20a%20binary,bit%20indicates%20a%20positive%20number.&text=Before%20a%20floating%2Dpoint%20binary,its%20mantissa%20must%20be%20normalized.



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)

Chapter 3 — Arithmetic for Computers — 20



IEEE Floating-Point Format

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

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

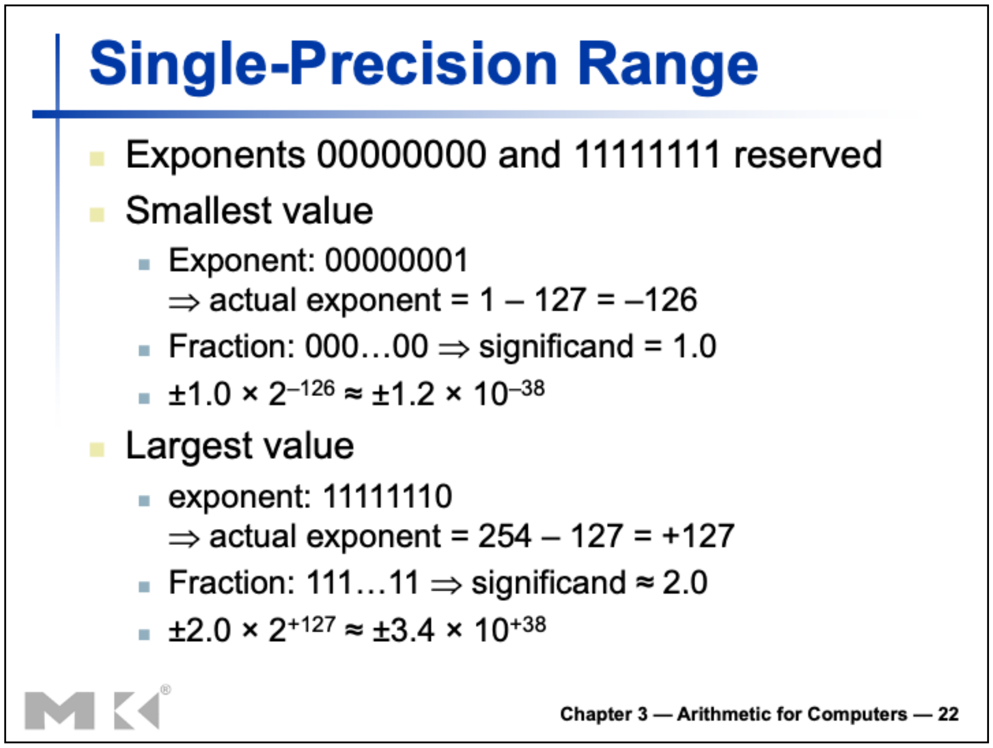
- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Chapter 3 — Arithmetic for Computers — 21

We have now reached the point where we can combine the sign, exponent, and fraction into the binary IEEE short real representation. Using Figure 1 as a reference, the value 1.101 x 2^0 would be stored as sign = 0 (positive), fraction = 101, and exponent = 01111111 (the exponent value is added to 127). The leading "1." was dropped from the fraction. Here are more examples:

Binary Value |||	**Biased Exponent.** ||||.	**Sign,Exponent, Fraction**
-1.11	127	1, 01111111, 11000000000000000000000
+1101.101	130	0, 10000010, 10110100000000000000000
-.00101	124	1 , 01111100, 01000000000000000000000
+100111.0	132	0, 10000100, 00111000000000000000000
+.0000001101011	120	0, 01111000, 10101100000000000000000

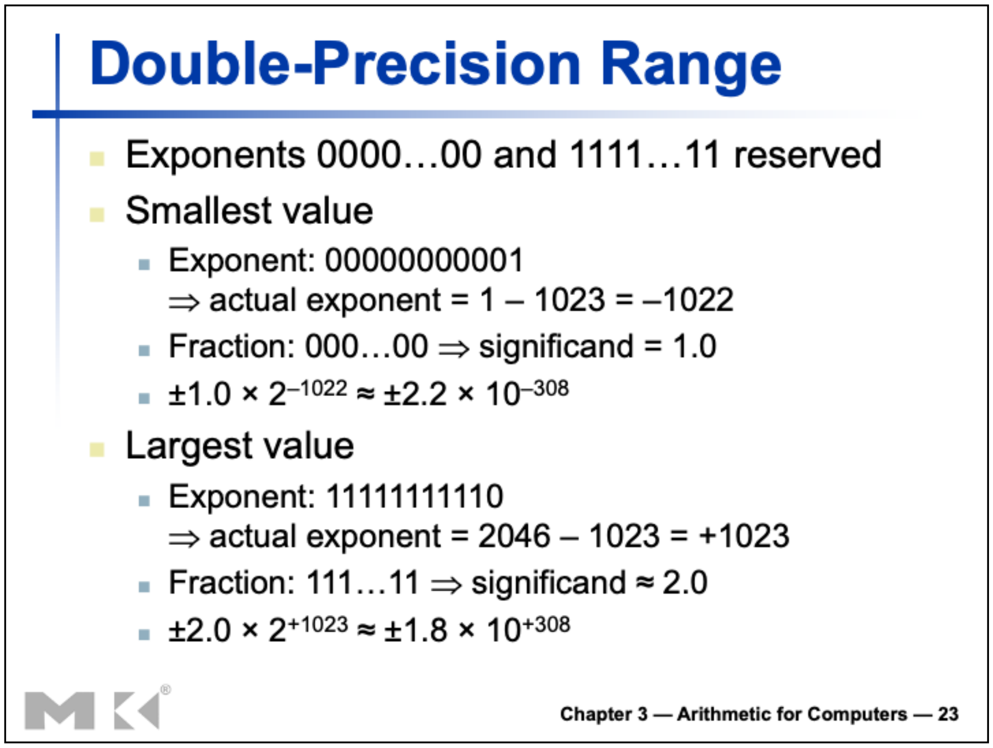
Significant = 1 + Fraction



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

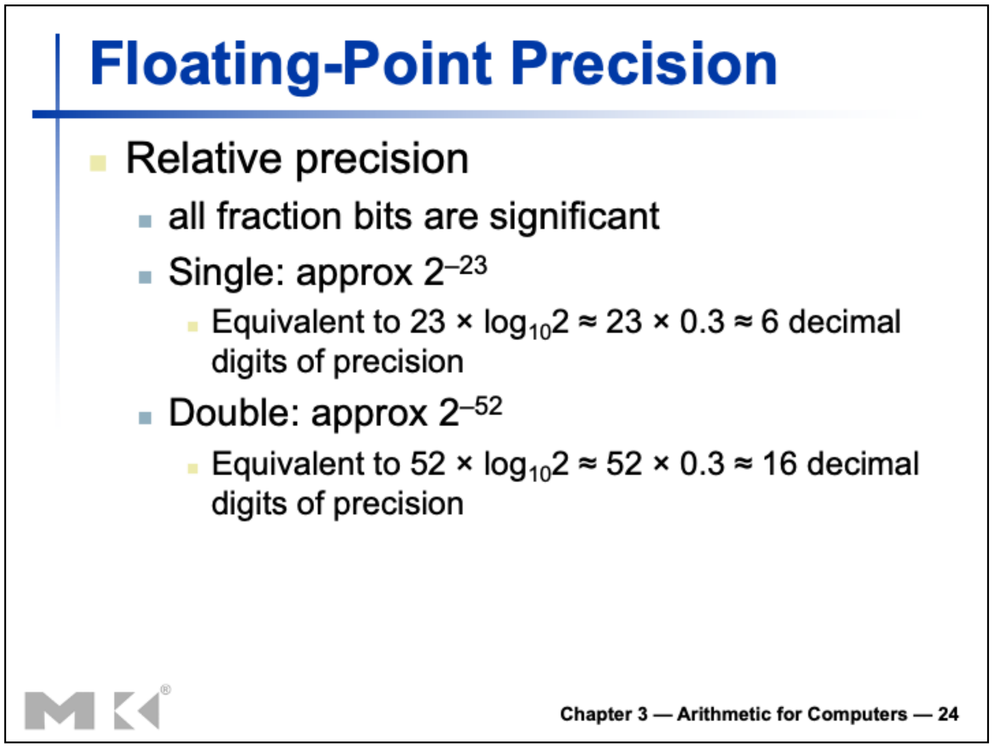
Chapter 3 — Arithmetic for Computers — 22



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

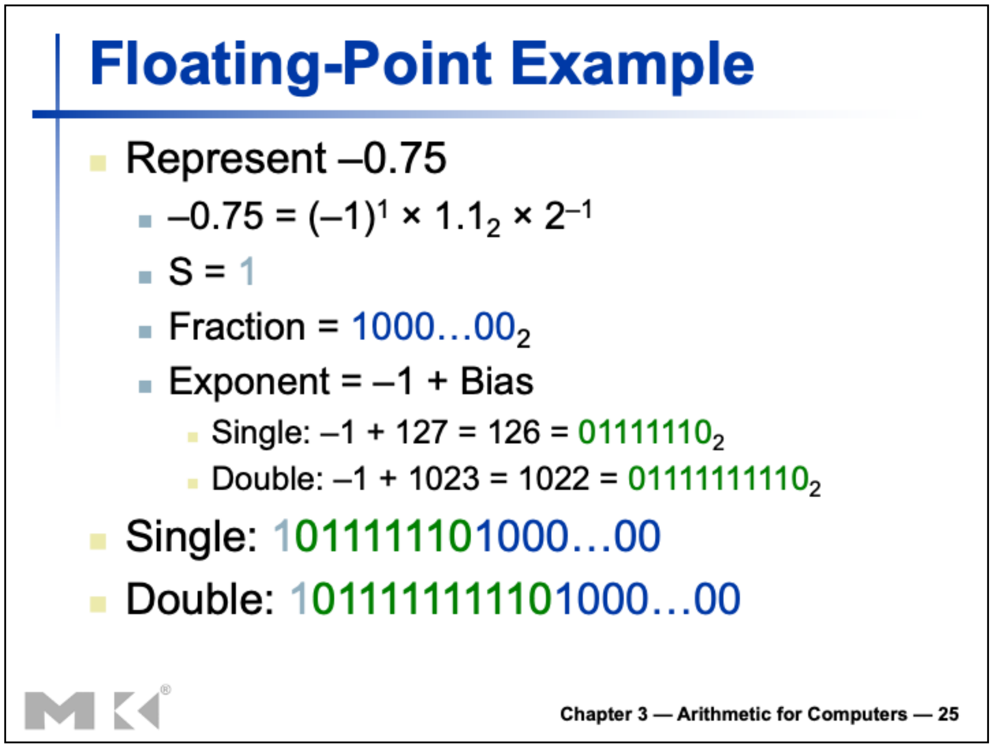
Chapter 3 — Arithmetic for Computers — 23



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

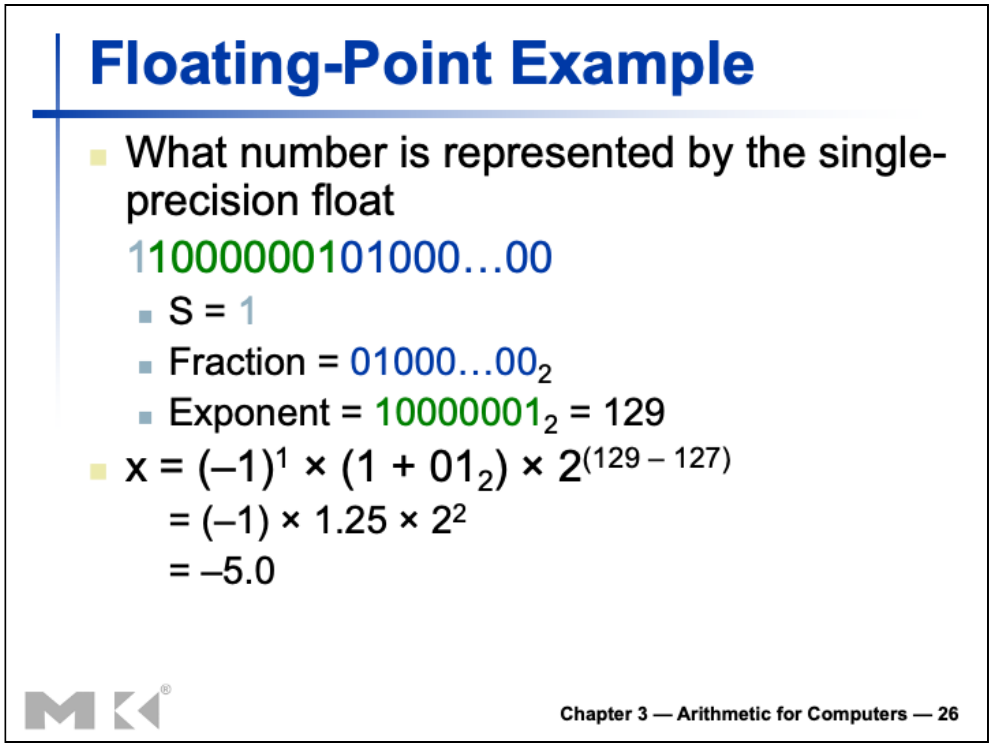
Chapter 3 — Arithmetic for Computers — 24



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: 1011111101000…00
- Double: 1011111111101000…00

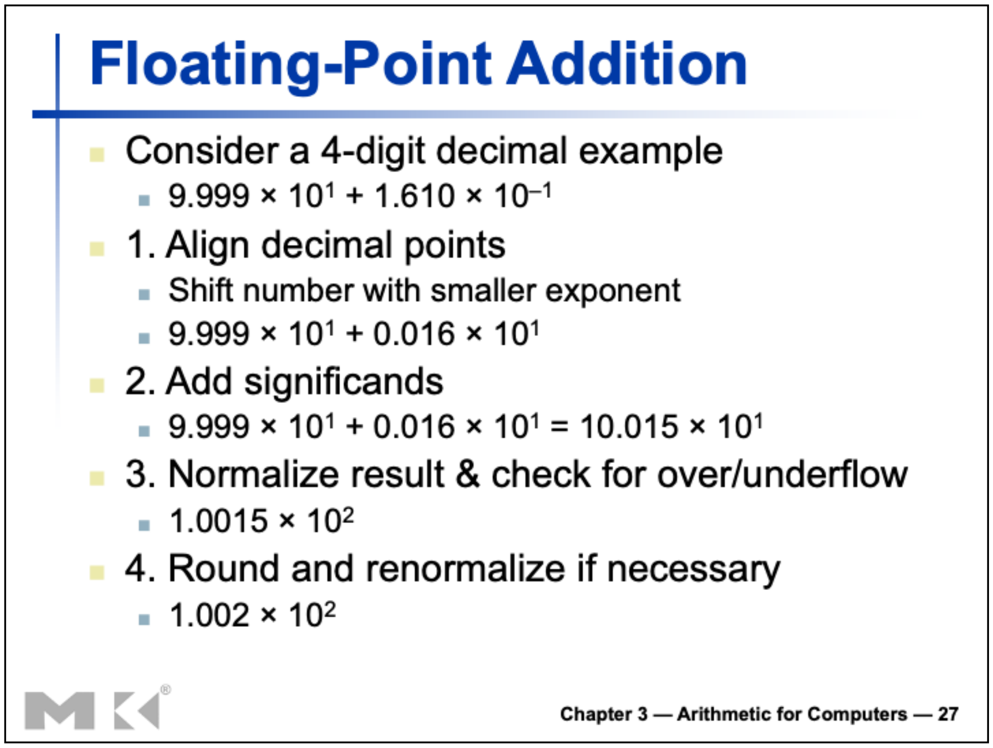
Chapter 3 — Arithmetic for Computers — 25



Floating-Point Example

- What number is represented by the single-precision float
  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

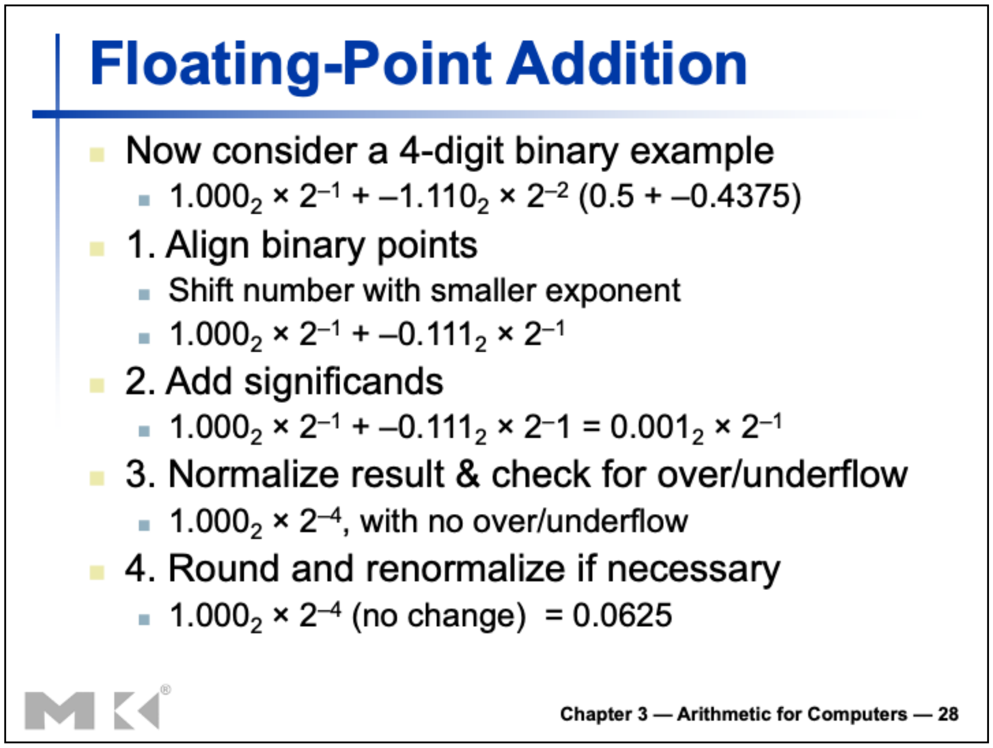
Chapter 3 — Arithmetic for Computers — 26



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}

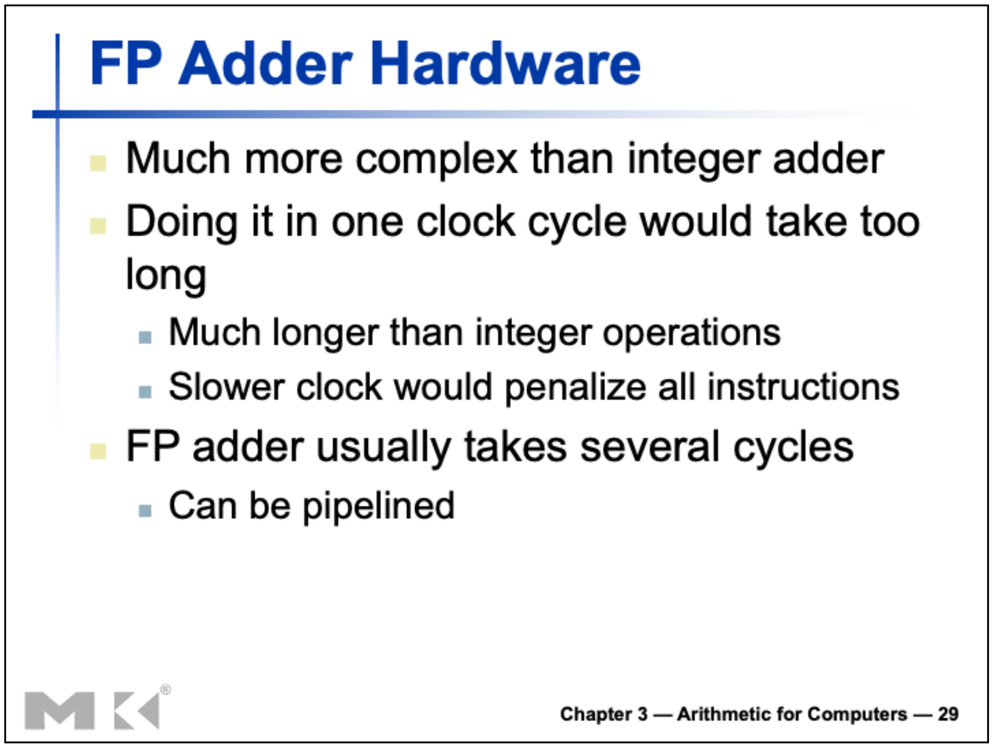
Chapter 3 — Arithmetic for Computers — 27



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

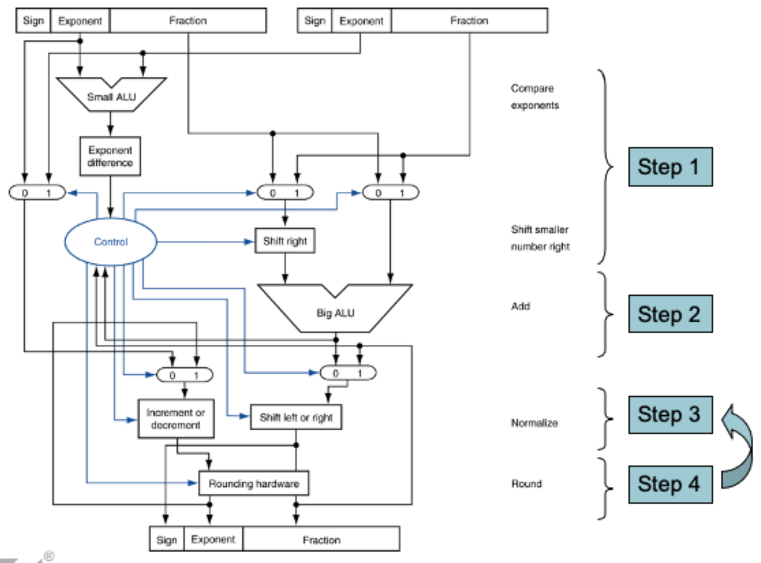
Chapter 3 — Arithmetic for Computers — 28



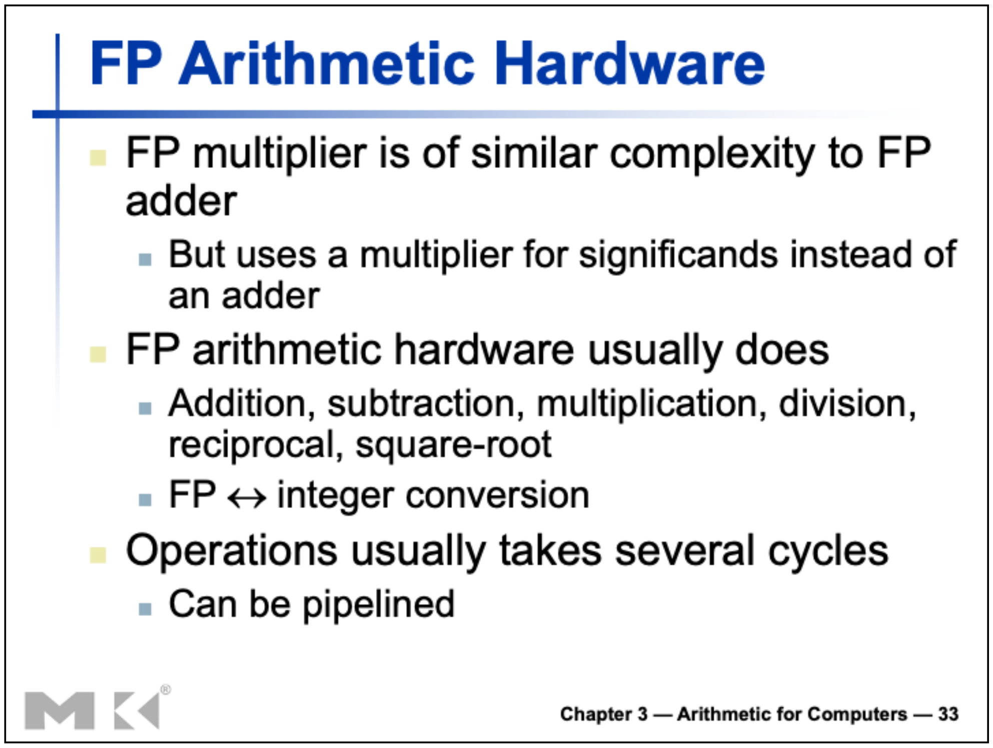
FP Adder Hardware

- 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



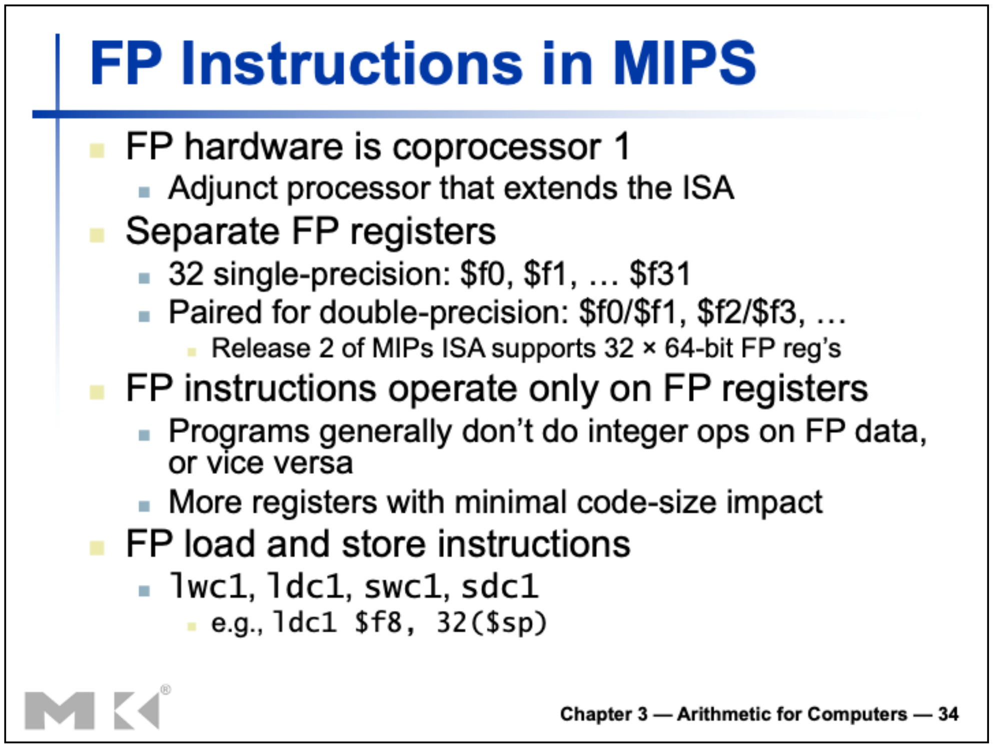
Chapter 3 — Arithmetic for Computers — 30



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

Chapter 3 — Arithmetic for Computers — 33



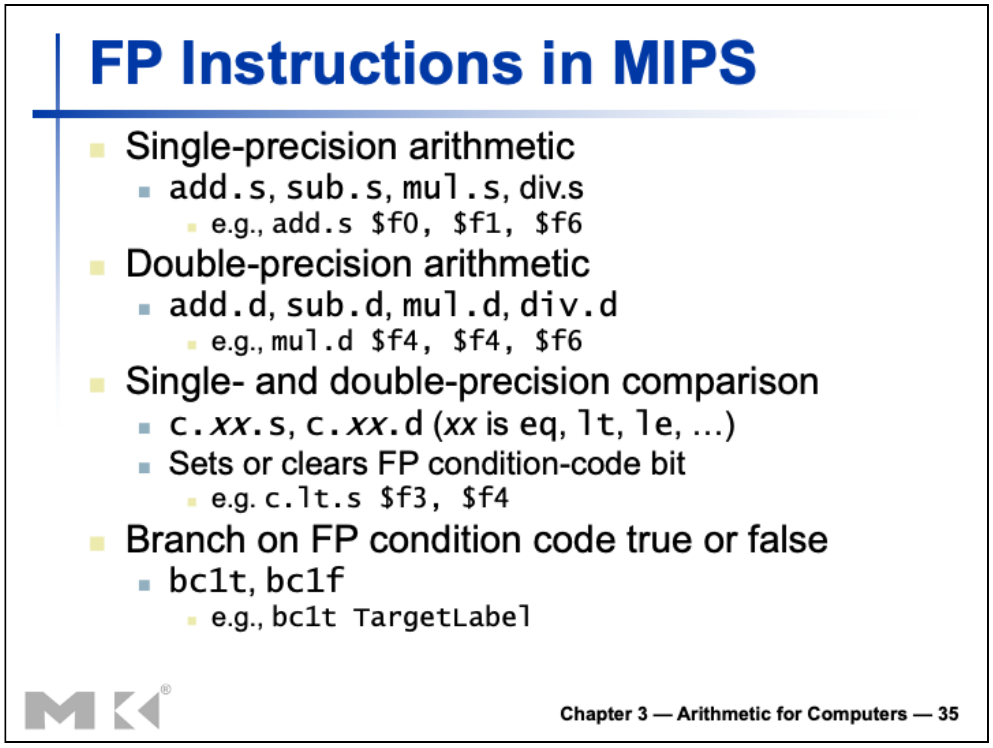
FP Instructions in MIPS

- 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)`

Chapter 3 — Arithmetic for Computers — 34

- One issue that architects face in supporting floating-point arithmetic is whether to use the same registers used by the integer instructions or to add a special set for floating point.
- Because programs normally perform integer operations and floating-point operations on different data, separating the registers will only slightly increase the number of instructions needed to execute a program.
- The major impact is to create a separate set of data transfer instructions (load, store) to move data between floating-point registers and memory.Hence, they included separate loads and stores for floating-point registers: lwc1 and swc1.

- Another reason for separate integers and floating-point registers is that microprocessors in the 1980s didn't have enough transistors to put the floating-point unit on the same chip as the integer unit. Hence, the floating-point unit, including the floating- point registers, was optionally available as a second chip. Such optional accelerator chips are called *coprocessors,* and explain the acronym for floating-point loads in MIPS: lwc1 means load word to coprocessor 1, the floating-point unit.



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`

Chapter 3 — Arithmetic for Computers — 35

- The MIPS designers decided to add separate floating-point registers—called $f0, $f1, $f2, ...—used either for single precision or double precision.

MIPS supports the IEEE 754 single precision and double precision formats with these instructions:

■ Floating-point *addition, single* (add.s) and *addition, double* (add.d)

■ Floating-point *subtraction, single* (sub.s) and *subtraction, double* (sub.d)

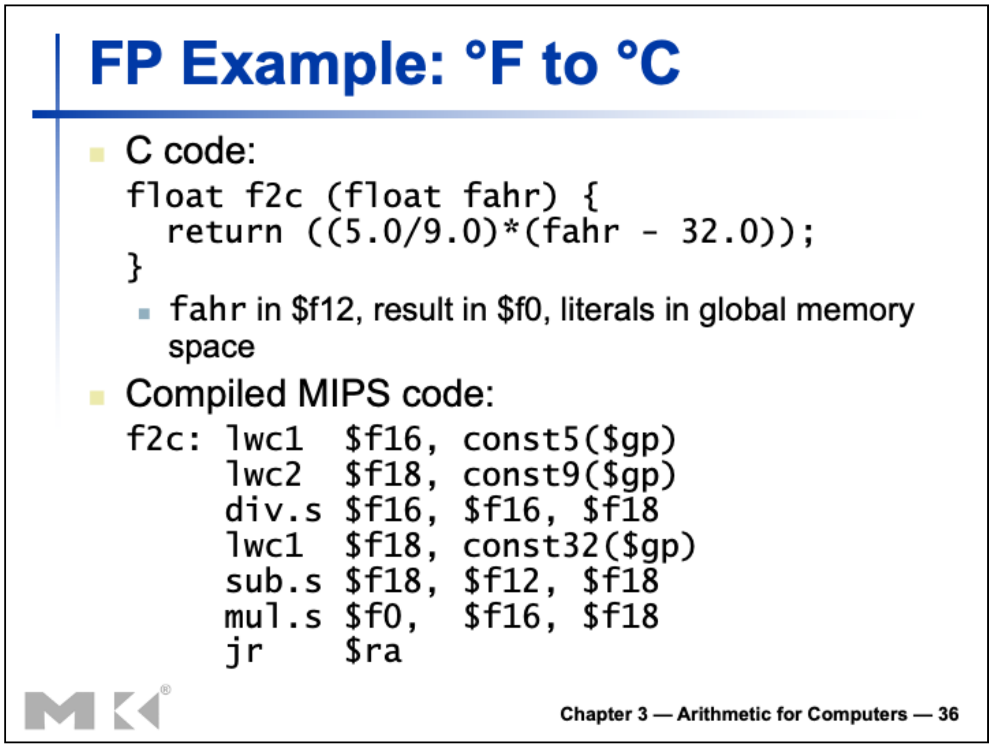
■ Floating-point *multiplication, single* (mul.s) and *multiplication, double* (mul.d)

■ Floating-point *division, single* (div.s) and *division, double* (div.d)

■ Floating-point *comparison, single* (c.x.s) and *comparison, double* (c.x.d), where x may be *equal* (eq), *not equal* (neq), *less than* (lt), *less than or equal* (le), *greater than* (gt), or *greater than or equal* (ge)

■ Floating-point *branch, true* (bclt) and *branch, false* (bclf)

- A double precision register is really an even-odd pair of single precision registers, using the even register number as its name. Thus, the pair of single precision registers $f2 and $f3 also form the double precision register named $f2.



FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

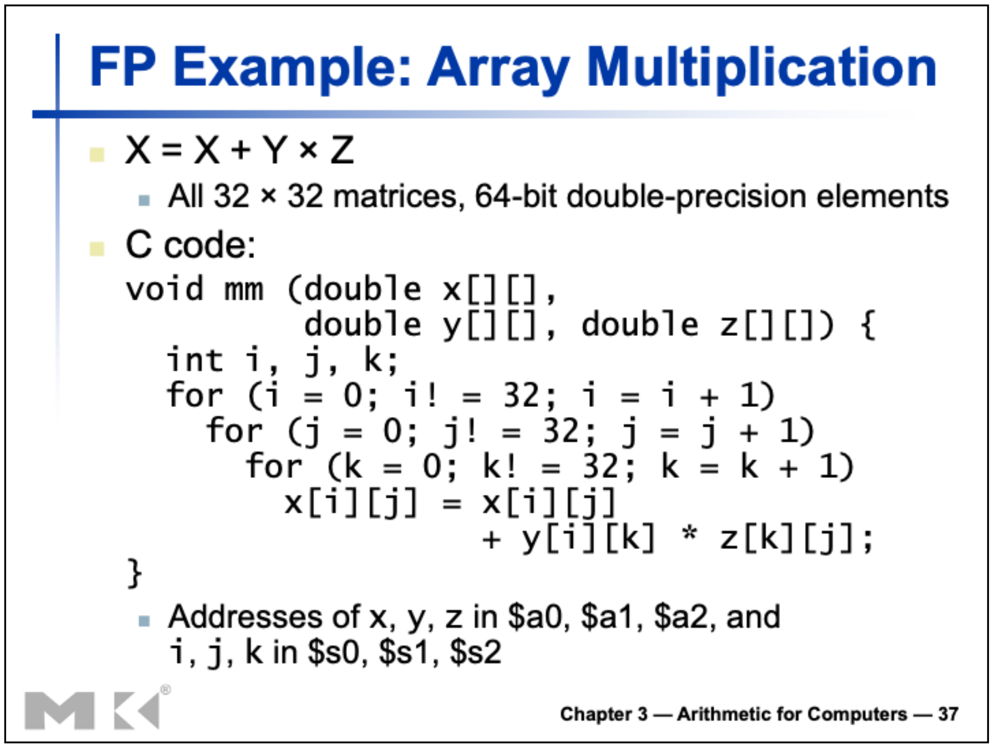
  - fahr in $f12, result in $f0, literals in global memory space

- 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
```

Chapter 3 — Arithmetic for Computers — 36

- The first two instructions load the constants 5.0 and 9.0 into floating-point registers
- 3 - They are then divided to get the fraction 5.0/9.0
- 4,5- Next, we load the constant 32.0 and then subtract it from fahr ($f12):
- 6,7 - we multiply the two intermediate results, placing the product in $f0 as the return result, and then return



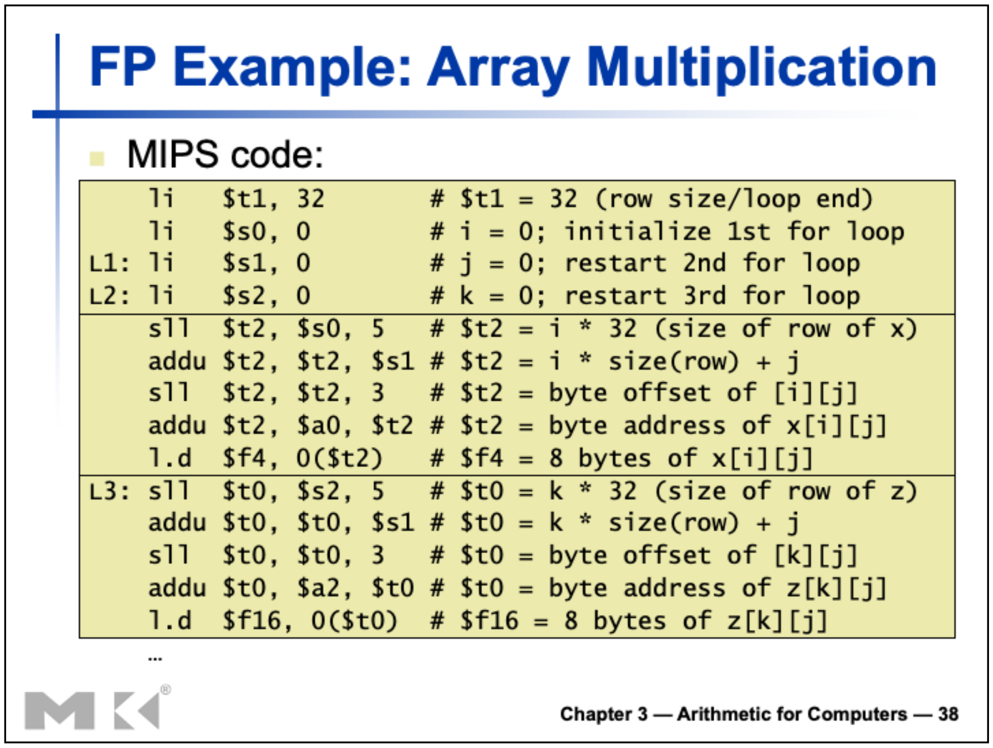
FP Example: Array Multiplication

- 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

Chapter 3 — Arithmetic for Computers — 37



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]
    …
```

Chapter 3 — Arithmetic for Computers — 38

- We keep the code simpler by using the assembly language pseudoinstructions

-- li (which loads a constant into a register), and

-- l.d and s.d (which the assembler turns into a pair of data transfer instructions, lwc1 or swc1, to a pair of floating-point registers).

- The body of the procedure starts with saving the loop termination value of 32 in a temporary register and then initializing the three *for* loop variables (i, j, k):

- sll $t2, $s0, 5 # $t2 = i * 2^5 (size of row of x)
- To calculate the address of x[i][j], we need to know how a 32 x 32, two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimension arrays, each with 32 elements. So the first step is to skip over the i "single-dimensional arrays," or rows, to get the one we want. Thus, we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead.

- Now we add the second index to select the jth element of the desired row:

addu $t2, $t2, $s1 # $t2 = i * size(row) + j

- To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by 3:

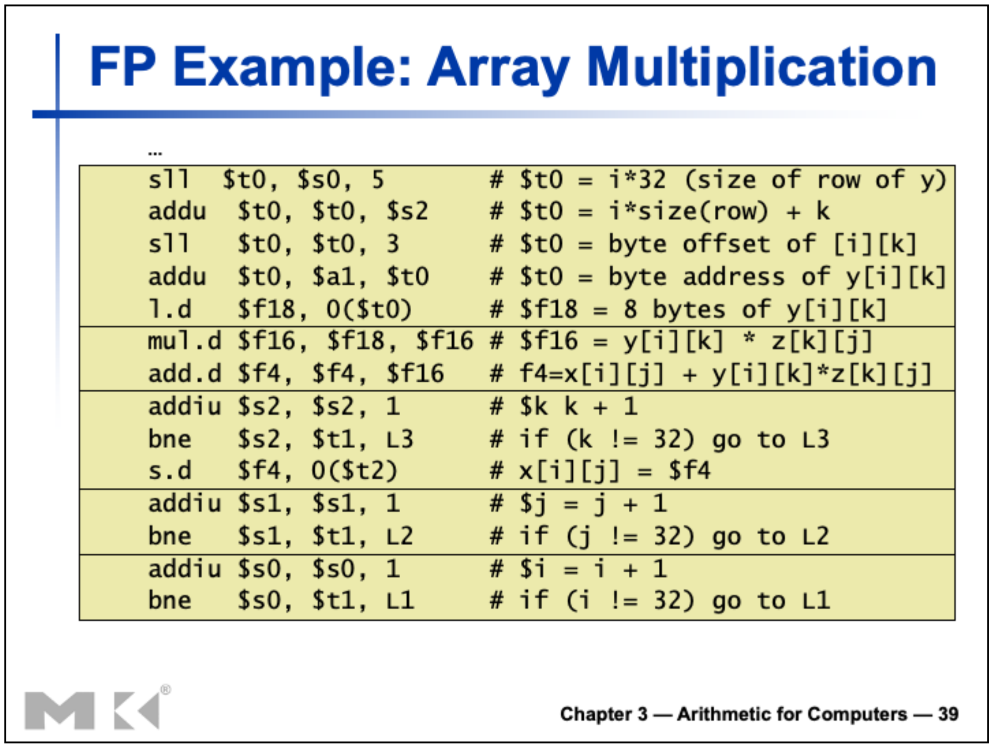
```
sll $t2, $t2, 3 # $t2 = byte offset of [i][j]
```

- Next we add this sum to the base address of x, giving the address of x[i][j], and then load the double precision number x[i][j] into $f4:

```
addu $t2, $a0, $t2   # $t2 = byte address of x[i][j]
l.d $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]
```

- The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number z[k][j].

```
L3: sll $t0, $s2, 5. # $t0=k*2^5 (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
```

Chapter 3 — Arithmetic for Computers — 39

- Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number y[i][k].

- Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of y and z located in registers $f18 and $f16, and then accumulate the sum in $f4.

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]

- The final block increments the index k and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in $f4 into x[i][j].

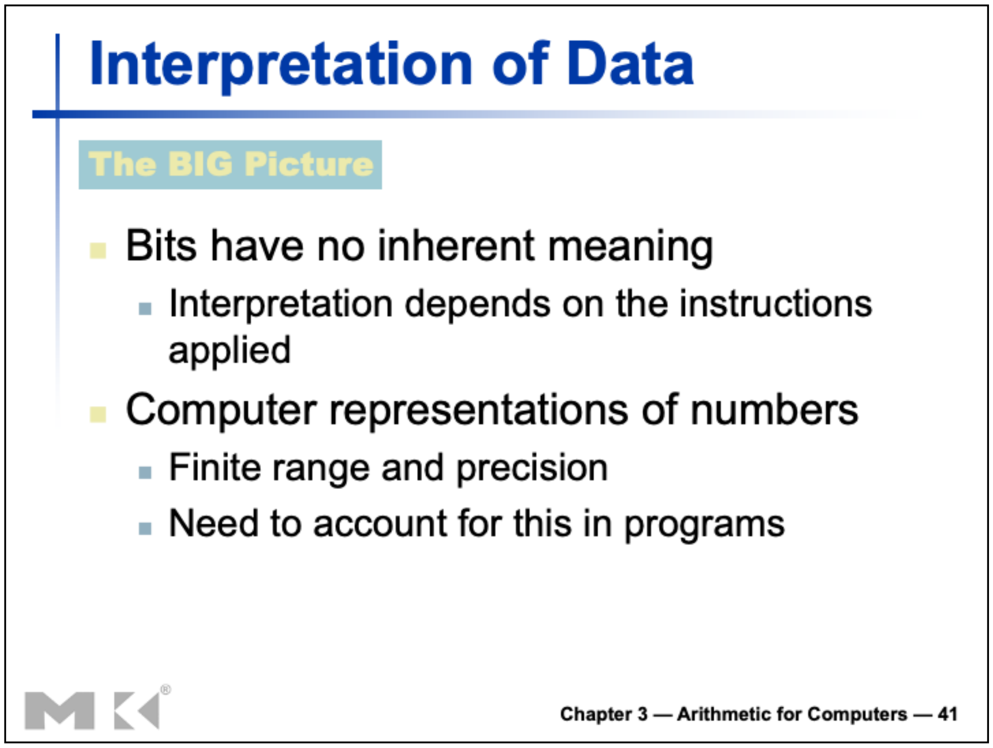
addiu $s2, $s2, 1 # $k = k + 1

bne $s2, $t1, L3 # if (k != 32) go to L3

s.d $f4, 0($t2) # x[i][j] = $f4

Similarly, these final four instructions increment the index variable of the middle and outermost loops, looping back if the index is not 32 and exiting if the index is 32.

```
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

Chapter 3 — Arithmetic for Computers — 41