Revised: 5/9/2024

Assembly Language III

Michael C. Hackett

Assistant Professor, Computer Science

COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

- Multiplication
- Division
- Floating Point Instructions
- Input/Output
- Floating Point Data Representation

Multiplication

- The easiest way to multiply numbers is through repeated addition:
  - $5 \times 4 = 4 + 4 + 4 + 4 + 4$

- To multiply $x \times y$, we could use a loop
  - We add y to an accumulator every iteration and use x to control the loop (or vice-versa)

- However, this is terribly inefficient for larger numbers
  - $500 \times 400 = 400 + 400 + 400 + 400 + 400 + 400 + 400 \ldots$

Multiplication

- A faster way to multiply numbers is achieved using bit shifting
  - Recall that shifting a number to the left is equivalent to multiplying the number by some power of 2

- Using bit shifting and some addition, multiplication can be sped up significantly

Multiplication

- Consider the following:
  - 100×20
  - $100 \times 20 = 64(20) + 32(20) + 4(20) = 2^6(20) + 2^5(20) + 2^2(20)$

- Three shift instructions and two add instructions are needed
  - Left shift 20 by 6 bits
  - Add with 20 left shifted by 5 bits
  - Add with 20 left shifted by 2 bits

Multiplication

- The Shift-and-Add Algorithm
  - The first operand is the multiplier and the second operand is the multiplicand
  1. Initialize an accumulator to zero
  2. If the multiplier is zero, terminate and return the accumulator (the product)
  3. If the multiplier is odd, add the multiplicand to the accumulator
  4. If the multiplier is even:
     1. Left shift the multiplicand by 1 bit
     2. Right shift the multiplier by 1 bit
  5. Repeat step 2

Multiplication

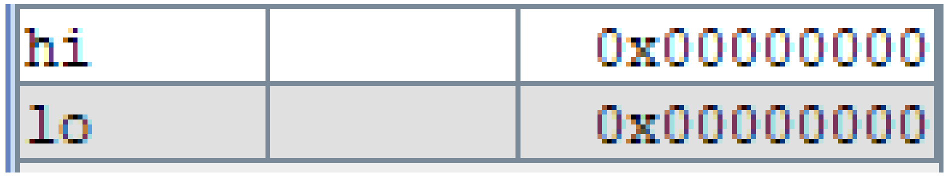
- When multiplying two binary numbers, the product will often use more bits than the operands
  - $1001 \times 0100 = 10100$

- When multiplying an M-bit number by a N-bit number, $M + N$ bits should be provided for the product

- MIPS has a mnemonic that performs multiplication on two 32-bit numbers
  - The result is 64 bits

Multiplication

- Since registers are 32-bits in size, the product is split between two registers
    - The hi register contains the higher-order 32 bits of the product
    - The lo register contains the lower-order 32 bits of the product
    - These registers are separate from the general registers



hi		0x00000000
lo		0x00000000

The hi and lo registers in MIPS

Multiplication

- The **`mult`** mnemonic multiplies two operands and stores the result to the hi and lo registers.

```
mult $rs, $rt
```

  - $rs = Source Register 1
  - $rt = Source Register 2

- To **m**ove the data **f**rom **hi** to a different register:

```
mfhi $rd
```

  - $rd = Destination Register

- To **m**ove the data **f**rom **lo** to a different register:

```
mflo $rd
```

Division

- When dividing an integer by another integer, we have two results: the quotient and remainder
  - Both of which are integers

- For example
  - $22 \div 4 = 5\ \mathit{with\ a\ remainder\ of}\ 2$
  - $38 \div 5 = 7\ \mathit{with\ a\ remainder\ of}\ 3$
  - $60 \div 6 = 10\ \mathit{with\ a\ remainder\ of}\ 0$

Division

- A quick way to divide numbers is achieved using bit shifting, much like the shift-and-add algorithm for multiplication
  - Recall that shifting a number to the right is equivalent to dividing the number by some power of 2

- The shift-and-subtract algorithm combines bit shifting and subtraction to perform division

Division

- Since division gives two results, two neighboring registers are used.
  - The first (left) register is the quotient
  - The second (right) register is the remainder

- The following algorithm is used to left shift a register pair:
  1. Left shift the left register by 1 bit
  2. Check if the high-order bit of the right register is 1
     - If so, add 1 to the left register
  3. Left shift the right register by 1 bit

Division

- The Shift-and-Subtract Algorithm
  - The first operand is the dividend and the second operand is the divisor
  1. Initialize the quotient and remainder to 0
  2. Shift the quotient left
  3. Shift the remainder-divisor register pair left
  4. If the remainder is greater than or equal to the divisor:
     1. Subtract the divisor from the remainder, placing the result in the remainder
     2. Increment the quotient
  5. Repeat step 2, once for each bit in the word (eg. 32 times)

Division

- The **`div`** mnemonic divides two operands and stores the remainder to the hi register and the quotient to the lo register.

```
div  $rs, $rt
```

  - $rs = Source Register 1 (Dividend)
  - $rt = Source Register 2 (Divisor)

- To **m**ove the data **f**rom **hi** to a different register:

```
mfhi $rd
```

  - $rd = Destination Register
- To **m**ove the data **f**rom **lo** to a different register:

```
mflo $rd
```

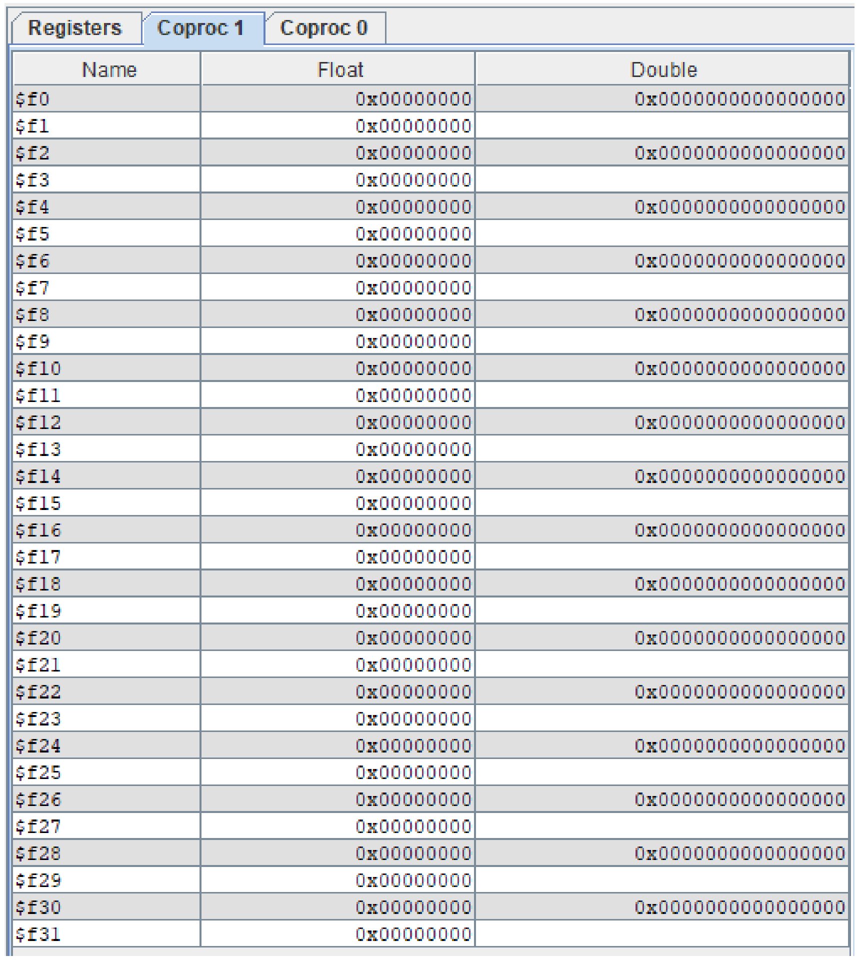
Floating Point Instructions

- For non-integer numbers, the **floating-point** data type is used.
  - Examples of non-integers: 5.678 and -22.1
- MIPS has two floating point data types- *float* and *double*
  - These are identical to the float and double types in Java
- The float type is a 32-bit (single precision) number
- The double type is a 64-bit (double precision) number
  - We will discuss what *single precision* and *double precision* mean later in this lecture. For now, you only need to remember the size (in bits) of each.

Floating Point Instructions

- MIPS has 32 additional registers that are used for holding floating point numbers.
    - Registers $f0 through $f31
- Each of these registers, like the registers previously seen, are 32 bits in size
- The registers are in a separate part of the CPU
    - Coprocessor 1

Floating Point Instructions



Name	Float	Double
$f0	0x00000000	0x0000000000000000
$f1	0x00000000	
$f2	0x00000000	0x0000000000000000
$f3	0x00000000	
$f4	0x00000000	0x0000000000000000
$f5	0x00000000	
$f6	0x00000000	0x0000000000000000
$f7	0x00000000	
$f8	0x00000000	0x0000000000000000
$f9	0x00000000	
$f10	0x00000000	0x0000000000000000
$f11	0x00000000	
$f12	0x00000000	0x0000000000000000
$f13	0x00000000	
$f14	0x00000000	0x0000000000000000
$f15	0x00000000	
$f16	0x00000000	0x0000000000000000
$f17	0x00000000	
$f18	0x00000000	0x0000000000000000
$f19	0x00000000	
$f20	0x00000000	0x0000000000000000
$f21	0x00000000	
$f22	0x00000000	0x0000000000000000
$f23	0x00000000	
$f24	0x00000000	0x0000000000000000
$f25	0x00000000	
$f26	0x00000000	0x0000000000000000
$f27	0x00000000	
$f28	0x00000000	0x0000000000000000
$f29	0x00000000	
$f30	0x00000000	0x0000000000000000
$f31	0x00000000	

Floating Point Instructions

- To initialize memory with a single-precision value, the **`.float`** directive is used in the program's data section.

```
.data
x:    .float     6.789
```

Floating Point Instructions

- The **`l.s`** (**l**oad **s**ingle-precision) mnemonic loads a single precision non-integer from memory to a register

```
l.s    $fd, label
```

  - $fd = Destination Register
  - label = Symbolic (or non-symbolic) memory reference

```
l.s    $f0, x
```

Floating Point Instructions

- The **`mov.s`** (**mov**e **s**ingle-precision) mnemonic moves a single precision non-integer from one register to another

```
mov.s       $fd, $fs
```

  - $fd = Destination Register
  - $fs = Source Register

```
mov.s       $f6, $f0
```

Floating Point Instructions

- Arithmetic for single precision floating-point data:

```
add.s      $fd, $fs, $ft
sub.s      $fd, $fs, $ft
mul.s      $fd, $fs, $ft
div.s      $fd, $fs, $ft
```

  - $fd = Destination Register
  - $fs = Source Register 1
  - $ft = Source Register 2

Floating Point Instructions

- Special instructions are used to compare the equality of single-precision values.
- The **`c.eq.s`** (**c**ompare **eq**uality **s**ingle-precision) mnemonic compares two floats for equality

```
c.eq.s      $fs, $ft
```

  - $fs = Source Register 1
  - $ft = Source Register 2

- A 1 is stored to a special 1-bit condition code if the result is true
- A 0 is stored to the condition code if the result is false

Floating Point Instructions

- The **`c.lt.s`** (**c**ompare **l**ess **t**han **s**ingle-precision) mnemonic compares two floats for a less than relationship

```
c.lt.s      $fs, $ft
```

  - $fs = Source Register 1
  - $ft = Source Register 2

- A 1 is stored to the condition code if the result is true
- A 0 is stored to the condition code if the result is false

Floating Point Instructions

- The **`c.le.s`** (**c**ompare **l**ess than or **e**qual **s**ingle-precision) mnemonic compares two floats for a less than or equal relationship

```
c.le.s    $fs, $ft
```

  - $fs = Source Register 1
  - $ft = Source Register 2

- A 1 is stored to the condition code if the result is true
- A 0 is stored to the condition code if the result is false

Floating Point Instructions

- To branch after a comparison, the **`bc1t`** mnemonic branches to a specified label if the condition code contains 1

```
bc1t   label
```

  - label = Section to jump to if the condition code contains 1

Floating Point Instructions

- To branch after a comparison, the **`bc1f`** mnemonic branches to a specified label if condition code contains 0

```
bc1f   label
```

  - label = Section to jump to if condition code contains 0

Floating Point Instructions

- To print a float, the number **must** be placed in register $f12
- The system call code for printing a float is 2
    - 2 **must** be placed in register $v0

```
mov.s       $f12, $f0    #Copies the value in $f0 to $f12
li          $v0, 2       #Sets the syscall code for printing a float
syscall                  #Prints the float in register $f12
```

Floating Point Instructions

- For 64-bit double-precision values, pairs of registers are used.
  - Though, only the first register is specified in instructions

- By convention, the even numbered registers are used as the first register of a double-precision number.
  - $f0 and $f1
  - $f2 and $f3
  - $f4 and $f5
  - and so on…

Floating Point Instructions

- To initialize memory with a double-precision value, the **`.double`** directive is used in the program's data section.

```
.data
x:    .float     6.789
y:    .double    3.1415
```

Floating Point Instructions

- The **`l.d`** (**l**oad **d**ouble-precision) mnemonic loads a double-precision non-integer from memory to a register

```
l.d    $fd, label
```

  - $fd = Destination Register
  - label = Symbolic (or non-symbolic) memory reference

```
l.d    $f0, y
```

Floating Point Instructions

- The **`mov.d`** (**mov**e **d**ouble-precision) mnemonic moves a double precision non-integer from one register pair to another

```
mov.d  $fd, $fs
```

  - $fd = Destination Register
  - $fs = Source Register

```
mov.d  $f6, $f0
```

  - The value in pair $f0 and $f1 will be moved to the register pair $f6 and $f7

Floating Point Instructions

- Arithmetic for double precision floating-point data:

```
add.d    $fd, $fs, $ft
sub.d    $fd, $fs, $ft
mul.d    $fd, $fs, $ft
div.d    $fd, $fs, $ft
```

  - $fd = Destination Register
  - $fs = Source Register 1
  - $ft = Source Register 2

Floating Point Instructions

- The **`c.eq.d`** (**c**ompare **eq**uality **d**ouble-precision) mnemonic compares two doubles for equality

```
c.eq.d     $fs, $ft
```

  - \$fs = Source Register 1
  - \$ft = Source Register 2

- A 1 is stored to the condition code if the result is true
- A 0 is stored to the condition code if the result is false

Floating Point Instructions

- The **`c.lt.s`** (**c**ompare **l**ess **t**han **d**ouble-precision) mnemonic compares two doubles for a less than relationship

```
c.lt.d      $fs, $ft
```

  - $fs = Source Register 1
  - $ft = Source Register 2

- A 1 is stored to the condition code if the result is true
- A 0 is stored to the condition code if the result is false

Floating Point Instructions

- The **`c.le.d`** (**c**ompare **l**ess than or **e**qual **d**ouble-precision) mnemonic compares two doubles for a less than or equal relationship

```
c.le.d     $fs, $ft
```

  - $fs = Source Register 1
  - $ft = Source Register 2

- A 1 is stored to the condition code if the result is true
- A 0 is stored to the condition code if the result is false

Floating Point Instructions

- To branch after a comparison, the process is the same as it was for floats
  - The **`bc1t`** mnemonic branches to a specified label if the condition code contains 1

```
bc1t   label
```

    - label = Section to jump to if the condition code contains 1

  - The **`bc1f`** mnemonic branches to a specified label if the condition code contains 0

```
bc1f   label
```

    - label = Section to jump to if the condition code contains 0

Floating Point Instructions

- To print a double, the number **must** be placed in register $f12
    - Would be $f12 and $f13

- The system call code for printing a double is 3
    - 3 **must** be placed in register $v0

```
mov.d          $f12, $f4      #Copies the value in $f4 and $f5 to $f12 and $f13
li             $v0, 3         #Sets the syscall code for printing a double
syscall                       #Prints the double in registers $f12 and $f13
```

Input/Output

- By now, you have seen examples of using the following system call codes for printing data of varying types

Service	System Call Code	Arguments	Result
Print an int	1	\$a0 = integer	Prints integer in \$a0
Print a float	2	\$f12 = float	Prints float in \$f12
Print a double	3	\$f12 = double	Prints double in \$f12/\$f13
Print a string	4	\$a0 = string	Prints string in \$a0

Input/Output

- We use a similar process to read keyboard input.

- It is important to remember to print a prompt before reading keyboard input.
  - Otherwise the program will pause and wait for input
  - It may not be clear to the user that the program is waiting for their typed entry

Input/Output

- To read an integer entered with the keyboard…
    - The system call code for reading an integer is 5
        - 5 **must** be placed in register $v0

    - The entered integer will be stored to register $v0

```
li      $v0, 5
syscall
```

Input/Output

- To read a float entered with the keyboard…
    - The system call code for reading a float is 6
        - 6 **must** be placed in register $v0
    - The entered float will be stored to register $f0
        - If there was data in $f0 it will be overwritten

```
li      $v0, 6
syscall
```

Input/Output

- To read a double entered with the keyboard…
    - The system call code for reading a double is 7
        - 7 **must** be placed in register $v0
    - The entered double will be stored to register $f0
        - If there was data in $f0 and $f1, it will be overwritten

```
li     $v0, 7
syscall
```

Input/Output

- To read a string entered with the keyboard…
  - The system call code for reading a string is 8
    - 8 **must** be placed in register $v0
  - The starting address of where the entered string will be stored must be in register $a0
    - The `.space` directive reserves the specified number of bytes
  - The maximum length (in bytes) of the string must be stored in $a1

```
la    $a0, entry
li    $a1, 1000
syscall

.data
entry: .space    1000
```

Input/Output

- To terminate a program…
  - The system call code for exiting the program is 10
    - 10 **must** be placed in register $v0

```
li      $v0, 10
syscall
```

Input/Output

Service	System Call Code	Arguments	Result
Print an int	1	$a0 = integer	
Print a float	2	$f12 = float	
Print a double	3	$f12 = double	
Print a string	4	$a0 = string	
Read an int	5		Integer in $v0
Read a float	6		Float in $f0
Read a double	7		Double in $f0
Read a string	8	$a0 = storage, $a1 = length	
Exit	10		

Floating Point Data Representation

- While 101.11 is a perfectly valid base 2 number, we have no way to specify a “binary point” in memory.
  - Like how we don’t have a way to specify a “-” for a negative number

$$\begin{array}{cccccc} 1 & 0 & 1 & . & 1 & 1 \\ 2^2 & 2^1 & 2^0 & & 2^{-1} & 2^{-2} \end{array}$$

$$(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$$

$$(1 \times 4) + (0 \times 2) + (1 \times 1) + (1 \times 0.5) + (1 \times 0.25)$$

$$4 + 0 + 1 + 0.5 + 0.25 = 5.75_{10}$$

Floating Point Data Representation

- Two such number formats for representing non-integers are:
  - Fixed Point Notation
  - Floating Point Notation

- Floating point is the more common of the two.

Floating Point Data Representation

- Fixed Point Notation
  - A specific number of bits allocated for digits to the left of the binary point.
  - A specific number of bits allocated for digits to the right of the binary point.

Binary	Decimal
00**0**	0.0
00**1**	0.5
01**0**	1.0
01**1**	1.5
10**0**	2.0
10**1**	2.5
11**0**	3.0
11**1**	3.5

Binary	Decimal
0**00**	0.00
0**01**	0.25
0**10**	0.50
0**11**	0.75
1**00**	1.00
1**01**	1.25
1**10**	1.50
1**11**	1.75

Floating Point Data Representation

- In floating-point, non-integers are represented via scientific notation.
  - The decimal number 12.41 could be re-written as 1.241×10^1
  - The decimal number -0.003 could be rewritten as -3.0×10^{-2}

- The binary number 1001.1 is expressed in scientific notation as $1.0011_2 \times 2^3$
  - Where $2^3 = 4 = 100_2$
  - The number 1.0011 is referred to as the significand or *mantissa*; The digits to the right of the binary point are called the *fraction*.

Floating Point Data Representation

$$1.01101 \times 2^{3} = 1011.01$$

$$-1.11 \times 2^{-1} = -0.111$$

- The placement of the binary point “floats” to where it needs to be.

Floating Point Data Representation

- We’ll begin with half-precision (16-bits) floating point format
- Memory space is allotted for:
  - A sign bit – 1 bit
  - The exponent – 5 bits
  - The mantissa – 10 bits
- (This example will use 16 bits to store 11.25, or 1.01101×2^3)
- This number is positive, so the sign bit is zero.

Sign Bit	Exponent	Fraction
0		

Floating Point Data Representation

- 1.01101×2^3
- The exponent is expressed as 2^{b-1}-1 more than the exponent's actual value.
  - Where *b* is number of bits allotted for the exponent.
  - $3 + (2^{5-1}-1) = 3 + (2^4-1) = 3 + 15 = 18 = 10010_2$

Sign Bit	Exponent	Fraction
0	10010	

Floating Point Data Representation

- 1.01101×2^3
- The mantissa is supposed to start with “1.” so only the fractional bits will be stored.

Sign Bit	Exponent	Fraction
0	10010	0110100000

Floating Point Data Representation

- Working backwards with a different number:

Sign Bit	Exponent	Fraction
1	10011	0110100000

- A 1 in the sign bit means the number is negative.
- Exponent: $10011_2 = 19$ -> $19 – (2^{5-1}-1) = 19 – (2^4-1)= 19 – 15 = 4$
- Mantissa: 1.01101

$$-1.01101_2 \times 2_{10}{}^4 = -1.01101_2 \times 1000_2 = -10110.1_2 = -22.5_{10}$$

Floating Point Data Representation

- Next, we'll demonstrate the same number but this time use single-precision (32-bits) floating point format
- Memory space is allotted for:
  - A sign bit – 1 bit
  - The exponent – 8 bits
  - The mantissa – 23 bits
- (This example will use 32 bits to store 11.25, or 1.01101×2^3)
  - This number is positive, so the sign bit is zero.

Sign Bit	Exponent	Fraction
0		

Floating Point Data Representation

- 1.01101×2^3
- The exponent is again expressed as 2^{b-1}-1 more than the exponent's actual value.
  - Where *b* is number of bits allotted for the exponent.
  - $3 + (2^{8-1}-1) = 3 + (2^7-1) = 3 + 127 = 130 = 10000011_2$

Sign Bit	Exponent	Fraction
0	10000011	

Floating Point Data Representation

- 1.01101×2^3
- The mantissa is supposed to start with "1." so only the fractional bits will be stored.

Sign Bit	Exponent	Fraction
0	10000011	01101000000000000000000

Floating Point Data Representation

- Finally, we'll demonstrate the same number using double-precision (64-bits) floating point format
- Memory space is allotted for:
  - A sign bit – 1 bit
  - The exponent – 11 bits
  - The mantissa – 52 bits
- (This example will use 64 bits to store 11.25, or 1.01101×2^3)
  - This number is positive, so the sign bit is zero.

Sign Bit	Exponent	Fraction
0		

Floating Point Data Representation

- 1.01101×2^3
- The exponent is again expressed as $2^{b-1}-1$ more than the exponent's actual value.
  - Where *b* is number of bits allotted for the exponent.
  - $3 + (2^{11-1}-1) = 3 + (2^{10}-1) = 3 + 1023 = 1026 = 10000000010_2$

Sign Bit	Exponent	Fraction
0	10000000010	

Floating Point Data Representation

- 1.01101×2^3
- The mantissa is supposed to start with “1.” so only the fractional bits will be stored.

Sign Bit	Exponent	Fraction
0	10000000010	0110100000000000000000000000000000000000000000000000

Floating Point Data Representation

- Half Precision – 16 bits
  - One sign bit
  - Five exponent bits
  - Ten fractional bits
- Single Precision – 32 bits
  - One sign bit
  - Eight exponent bits
  - Twenty-three fractional bits
- Double Precision – 64 bits
  - One sign bit
  - Eleven exponent bits
  - Fifty-two fractional bits