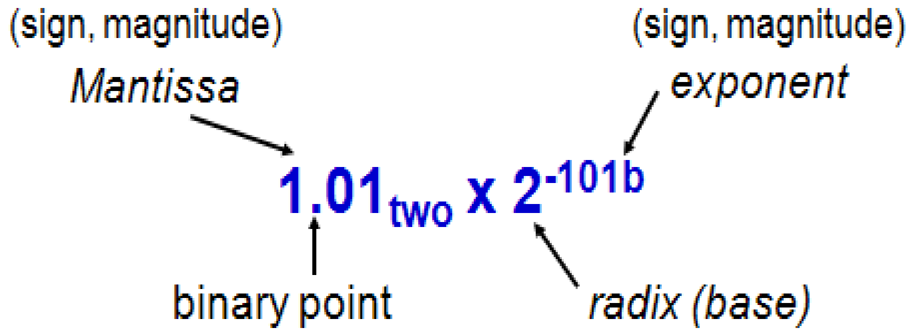
Lecture 5 [Supplement]: Floating Point numbers
[For Self-Study]



Topics

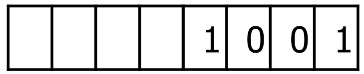
- Division and multiplication
  - Algorithms
  - MULTIPLY and DIVIDE in MIPS
- Floating point numbers
  - Binary floating point arithmetic
  - Introduction to IEEE Standard 754
  - Real life (and death) examples of floating point errors
  - Floating point support in MIPS

Arithmetic by shifting

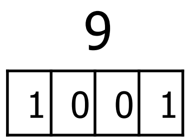
- For a base ***n*** representation
  - a shift to the left is like multiplying by **`n`**

    ```
    sll rd, rs, 2
    ```

  - a shift to the right is like dividing by **`n`**



- PITFALLS
  - **multiplying** numbers by shifting left may result in overflow
    - but can be used with caution for small integers, for example
  - **division** by arithmetic (not logical) right shift
    - positives rounded down



    - negatives? also rounded down?

Division by shifting

- Example: **-7/2**

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1

- shift by one to right (sign extend)

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

- Let's check the result

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
																															1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

- the result is -4 , BUT we expected -3

MULTIPLY (unsigned)

- Paper and pencil example (unsigned):

```
Multiplicand          1000
Multiplier            1001
                      ----
                      1000
                     0000
                    0000
                   1000
                  --------
Product           01001000
```

- Observation:
  - **m** bits x **n** bits = **m + n** bit product
  - multiplication must be able to cope with **overflow**
  - with only 1's and 0's -> we either **add** the multiplicand or **do nothing**

MULTIPLY (unsigned)

- Pseudo-code implementation m x n (Unsigned)

INPUT

m := Multiplicand;

n := Multiplier; */* We view n as a string of bits: n[3], n[2], n[1], n[0] */*

OUTPUT

result := m x n;

```
    1000
    1001
 --------
    1000
   0000
  0000
 1000
 --------
 01001000
```

BEGIN

SET result = 0;

SET i = 0;

REPEAT

IF n[i] = 1 THEN result = result + m; **;otherwise skip Addition**

arithmetic shift m left by 1 place; **;keep Shifting m**

i = i + 1;

UNTIL i = 4;

PRINT result;

END

Multiplication algorithms

- Implementation of multiplication (in hardware or software)
  - by a series of **shifts** and **additions**
  - as many additions as many bits in the multiplier
- Optimisations
  - for **0**'s bits in the multiplier the **addition** is skipped
  - clever use of the multiplicand, multiplier and product registers
  - looking at more bits of the multiplier for each step (like in Booth's Algorithm)

Booth's Algorithm: Elaboration

- Key observation:
  - $\underbrace{\mathbf{1}111 \ldots 111\mathbf{1}}_{n} = 2^n - 1 = 2^n - 2^0$
  - so if we encounter **a string of 1's** in the **multiplier** we can subtract the **multiplicand** at the beginning of the string, and add **multiplicand** at the end
  - instead of adding for each occurrence of 1
- Actions for pairs of "current bit, bit to the right"

  00 - middle of string of 0's, shift, do nothing

  11 - middle of string of 1's, shift, do nothing

  01 - end of string of 1's, shift, add the shifted multiplicand

  10 - beginning of string of 1's, shift, subtract the shifted multiplicand

Booth's Algorithm: Pseudocode implementation

- Pseudo-code implementation m x n (Unsigned)

```
INPUT
        m := Multiplicand;
        n := Multiplier;  /* We view n as a string of bits: n[3], n[2], n[1], n[0] */
OUTPUT
        result := m x n;
BEGIN
        SET result = 0;
        SET i = 0;
        SET previous = 0;
        REPEAT
                current = n[i];
                IF current = 1 AND previous = 0 THEN result = result - m;
                IF current = 0 AND previous = 1 THEN result = result + m;
                shift m left 1 place;        ;keep shifting
                i = i + 1;
                previous = current;
        UNTIL i = 4;
        PRINT result;
END
```

MULTIPLY in MIPS

- MIPS registers
  - two special purpose registers **hi** and **lo**
  - **hi**: high-order word of product
  - **lo**: low-order word of product
- MIPS instructions

```
mult rs1, rs2   # (hi, lo) = rs1 * rs2 ;signed
multu rs1, rs2  # (hi, lo) = rs1 * rs2 ;unsigned
mfhi rd         # move from hi to rd
mflo rd         # move from lo to rd
```

Overflow in multiplication

- 32-bit integer result in **lo**
- logically overflow if product too big
- but software must check **hi**
  - for multu register **hi** should be zero
  - for mult register **hi** should be extended sign of **lo**
- Detecting: Multiply $s5 by $s6, product in $t7

```
mult  $s5,$s6            # perform multiplication
mfhi  $t6                # move hi to $t6
mflo  $t7                # product from lo to $t7
xor  $t6,$t6,$t7         # compare signs of hi and lo
slt  $t6,$t6,$zero       # $t6=0 if signs different
beq  $t6,$zero,Overflow  # if different there is
                         # overflow
```

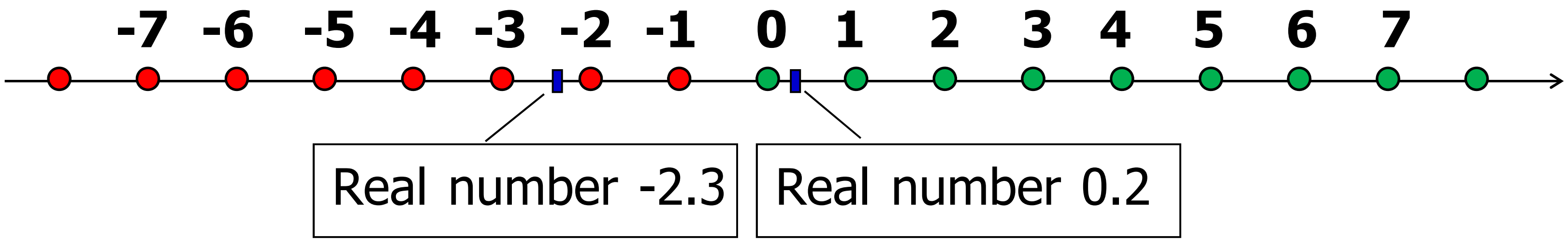
DIVIDE in MIPS

- all divide instructions put Remainder into hi register, and Quotient into lo register

```
div rs1, rs2    # divide rs1 by rs2; signed
                # quotient in lo, remainder in hi
divu rs1, rs2   # divide rs1 by rs2; unsigned
```

- Overflow and division by 0 are NOT detected by hardware
  - software takes responsibility
  - assembly language programmer or compiler

Other Numbers



- What about
  - Very large numbers? (seconds/century)

    $3,155,760,000_{ten}$ ($3.15576_{ten} \times 10^9$)
  - Very small numbers? (second / nanosecond)

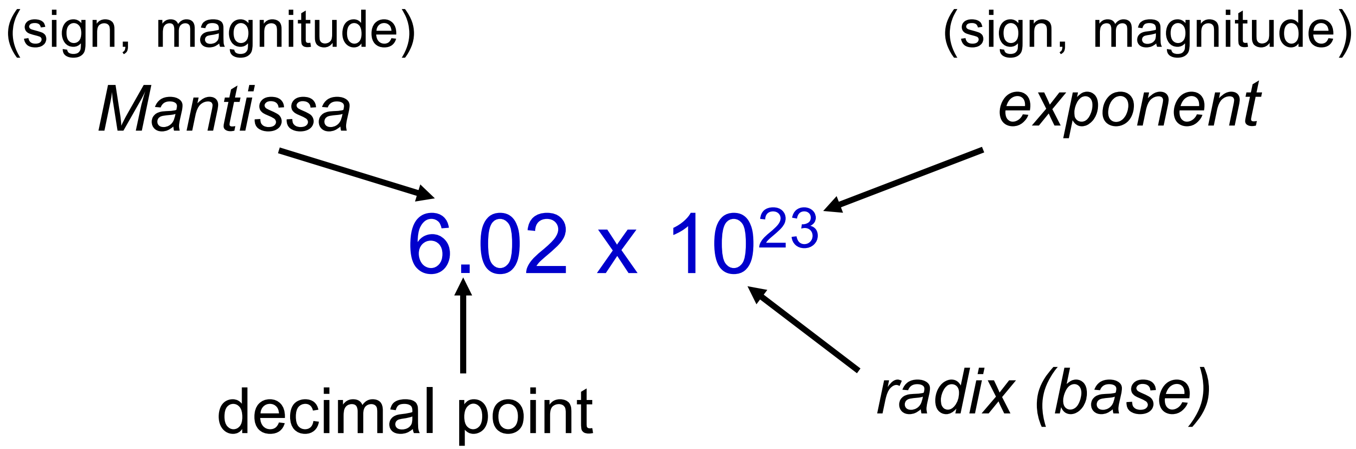
    0.000000001_{ten} ($1.0_{ten} \times 10^{-9}$)
  - Rationals

    2/3 (0.666666666. . .)
  - Irrationals

    $2^{1/2}$ (1.414213562373. . .)
  - Transcendentals

    e (2.718...), π (3.141...)

Recall Scientific Notation



- E.g. Alternatives to represent **1/1,000,000,000**
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}, … [floating point?]
  - Normalized: 1.0×10^{-9}
- Normal form: **no leading zeros , 1 digit to left of decimal point**
  - Simplifies data exchange, increases accuracy
  - Ensures single representation for every value

Scientific Notation for Binary Numbers

(sign, magnitude) *Mantissa*

(sign, magnitude) *exponent*

$$1.01_{two} \times 2^{-101b}$$

binary point

radix (base)

- Normal format: $\mathbf{1.xxxxxxxxxx_{two} * 2^{yyyy}{}_{two}}$

  leading digit is always **1**, so called 'hidden' or 'implied' **1**, and is implemented in hardware.

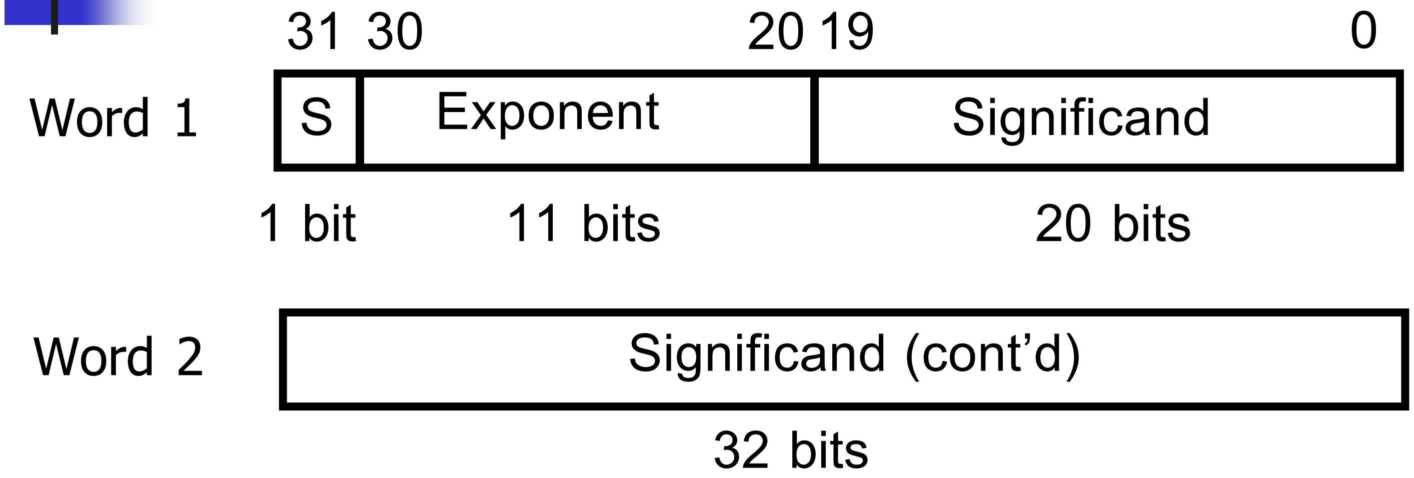
  - **1.xxxxxxxxxx**: Mantissa
  - **xxxxxxxxxx**: significand (significant positions)
  - **yyyy**: exponent

IEEE 754 Floating Point Standard

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- Word Size (32 bits, 23-bit Significand Single Precision)
- Value: **$(-1)^S$ x Mantissa x $2^{Exponent}$** [broken into 3 parts]
- Range: Represent numbers as small as **2.0×10^{-38}** to as large as **2.0×10^{38}**
  - if result too large? ($> 2.0 \times 10^{38}$), **Overflow** => Exponent larger than can be represented in 8-bit Exponent field
  - if result too small? (>0, $< 2.0 \times 10^{-38}$), **Underflow** => Negative exponent larger than can be represented in 8-bit Exponent field
- Issues: increase range (Exponent field) and accuracy (no. of significant positions)

IEEE 754 Floating Point Standard



- Multiple of Word Size (64 bits, 52-bit Significand for Double Precision)
- Representing Mantissa: If significand bits left-to-right are $s_1, s_2, s_3, \ldots$ then, Mantissa: $1.s_1s_2s_3\ldots$; the FP value is:

$$(-1)^S \times (\mathbf{1}+(s_1 \times 2^{-1})+(s_2 \times 2^{-2})+(s_3 \times 2^{-3})+\ldots) \times 2^{Exponent}$$

NOTE: $1.s_1s_2s_3\ldots$

2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	…
1	s_1	s_2	s_3	s_4	s_5	…

IEEE 754 Floating Point Standard

$$(-1)^S \times (\mathbf{1}+(s_1 x 2^{-1})+(s_2 x 2^{-2})+(s_3 x 2^{-3})+\ldots) \times 2^{\mathbf{Exponent}}$$

- Representing Exponent (Binary signed pattern)
  - 2’s comp?
    - Not as intuitive as Unsigned numbers for comparison
  - Excess Notation
    - where: 0000 0000 is most negative, and 1111 1111 is most positive; comparison is as intuitive as Unsigned numbers
    - subtract a bias number to get real number (or add the bias number to get excess-exponent)

IEEE 754 uses bias of **127** for single precision

$$(-1)^s \times (1.\text{Significand}) \times 2^{(\mathbf{Excess\_Exponent-127})}$$

IEEE 754 uses bias of **1023** for double precision

$$(-1)^s \times (1.\text{Significand}) \times 2^{(\mathbf{Excess\_Exponent-1023})}$$

Converting Decimal to FP

$(-1)^S \times (\mathbf{1}+(s_1 x 2^{-1})+(s_2 x 2^{-2})+(s_3 x 2^{-3})+\ldots) \times 2^{\textbf{Excess-Exponent}}$

- Example: representation of -0.75
  - Change appearance:
  - Work out three parts: S, Mantissa, and Exponent
    - Sign? 1
    - 1.Significand? $1.5_{ten} = 1.100_{two}$

2^0 (1)	2^{-1} (0.5)	2^{-2} (0.25)	2^{-3}	2^{-4}	…
1	**1**	**0**	**0**	**0**	

- Exponent?

  Real Exponent: -1

  Excess Exponent: $-1 + 127 = 126_{ten} = (\quad ? \quad)_{two}$

31	30						23	22																							0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Converting FP to Decimal

$(-1)^S \times (\mathbf{1}+(s_1 x 2^{-1})+(s_2 x 2^{-2})+(s_3 x 2^{-3})+\ldots) \times 2^{\textbf{Excess-Exponent}}$

- Example
  - Work out three parts: S, Mantissa, and Exponent
    - Sign? 0
    - 1.Significand? 1. 101 0101 0100 0011 0100 0010 $_{two}$ = (?)$_{ten}$

	S_1	S_2	S_3	S_4	
2^0 (1)	2^{-1} (0.5)	2^{-2} (0.25)	2^{-3}	2^{-4}	…
1 •	1	0	1	0	…

    - Exponent?

      Excess Exponent: 0110 1000$_{two}$ = 104$_{ten}$

      Real Exponent: 104 - 127=-13 [Bias adjustment]

31	30							23	22																						0
0	0	1	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0	0	1	0

Basic FP Addition Algorithm

$(-1)^S \times (\mathbf{1}+(s_1 x 2^{-1})+(s_2 x 2^{-2})+(s_3 x 2^{-3})+\ldots) \times 2^{\textbf{Excess-Exponent}}$

- For addition (or subtraction) of X to Y (X<Y):
  - (1) Compute ***D = ExpY - ExpX*** (align binary point)
  - (2) Right shift (*ManX*) by *D* bits => ***(ManX)*2^(ExpX-ExpY)*** $(ManX)*2^{(ExpX-ExpY)}$
  - (3)Compute $(ManX)*2^{(ExpX - ExpY)} + ManY$
- Floating Point addition is NOT associative
  - $(x + y) + z \neq x + (y + z)$

	Decimal	Binary
x	-102	11101000
y	102	01101000
z	.000012	00001000

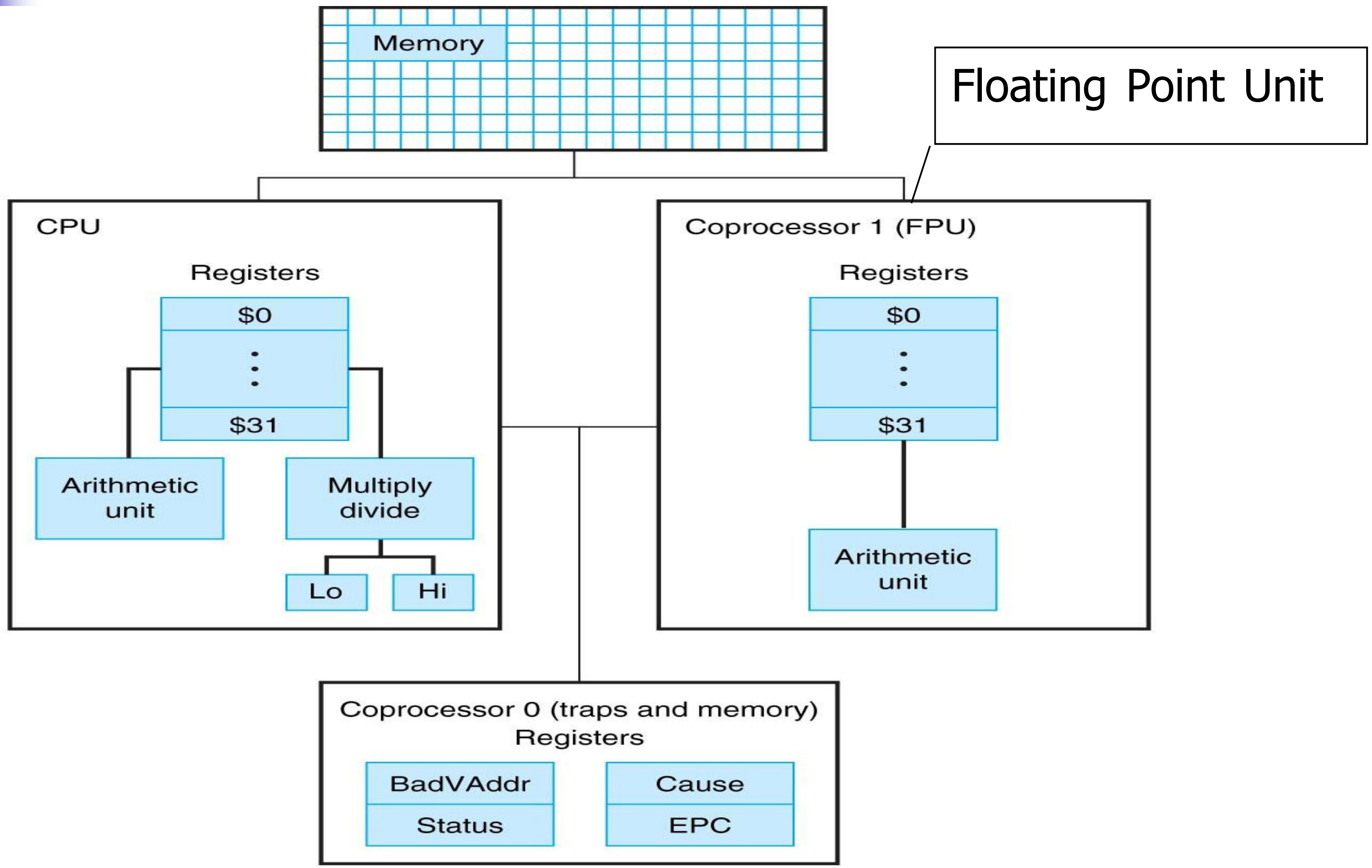
$(x + y) + z = 00000000 + 00001000 = 00001000 \;\; => (.000012)_{ten}$

$x + (y + z) = 11101000 + 01101000 = 00000000 \;\; => (0)_{ten}$

Floating Point Fallacy: is Accuracy Optional?

- FP Fallacies: FP result approximation of real result!
- July 1994: Intel discovers bug in Pentium
  - Occasionally affects bits 12-52 of Double Precision divide
- Sept: Math Prof. discovers, put on WWW
- Nov: Front page trade paper, then NY Times
  - Intel: "…several dozen people that this would affect. So far, we've only heard from one."
  - Intel claims customers see 1 error/27000 years
  - IBM claims 1 error/month, stops shipping computers with Intel CPU
- December: Intel apologizes, replace chips $300M
- Reputation? What responsibility to society?

MIPS FPU



Floating Point Unit
Memory
CPU
Registers
$0
$31
Arithmetic unit
Multiply divide
Lo
Hi
Coprocessor 1 (FPU)
Registers
$0
$31
Arithmetic unit
Coprocessor 0 (traps and memory)
Registers
BadVAddr
Status
Cause
EPC

MIPS Floating Point Architecture

- Single Precision, Double Precision versions of add, subtract, multiply, divide, compare
  - Single `add.s, sub.s, mul.s, div.s, c.lt.s`
  - Double `add.d, sub.d, mul.d, div.d, c.lt.d`
- Registers
  - Simplest solution: use existing registers
  - Normally integer and FP operations on different data, for performance could have separate registers
- MIPS provides 32 32-bit FP. reg: $f0, $f1, $f2 ...,
  - Thus need FP data transfers: lwc1, swc1
  - Double Precision? Even-odd pair of registers ($f0#$f1) act as 64-bit register: $f0, $f2, $f4, ...

New MIPS FP arithmetic instructions

```
add.s $f0,$f1,$f2 # $f0=$f1+$f2 FP Add (single)
add.d $f0,$f2,$f4 # $f0=$f2+$f4 FP Add (double)
sub.s $f0,$f1,$f2 # $f0=$f1-$f2 FP Subtract (single)
sub.d $f0,$f2,$f4 # $f0=$f2-$f4 FP Subtract (double)
mul.s $f0,$f1,$f2 # $f0=$f1x$f2 FP Multiply (single)
mul.d $f0,$f2,$f4 # $f0=$f2x$f4 FP Multiply (double)
div.s $f0,$f1,$f2 # $f0=$f1÷$f2 FP Divide (single)
div.d $f0,$f2,$f4 # $f0=$f2÷$f4 FP Divide (double)
c.X.s $f0,$f1     # flag1= $f0 X $f1 FP Compare (single)
c.X.d $f0,$f2     # flag1= $f0 X $f2 FP Compare (double)

# where X is: eq (equal), lt (less than), le (less than
# equal) to tests flag value:
# bc1t - floating-point branch true
# bc1f - floating-point branch false
```

Example with FP Multiply [Exercise]

```
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];
}
```

- Starting *addresses* are parameters in $a0, $a1, and $a2. Integer *variables* are in $t3, $t4, $t5. Arrays 32 by 32
- Use pseudoinstructions: li (load immediate), l.d/s.d (load/store 64 bits)

MIPS code for first piece: **initilialize, x[ ][ ]**

- Initailize Loop Variables

```
mm: ...
        li $t1, 32    # $t1 = 32
        li $t3, 0     # i = 0; 1st loop
L1:     li $t4, 0     # j = 0; reset 2nd
L2:     li $t5, 0     # k = 0; reset 3rd
```

- To fetch x[i][j], skip i rows (i*32), add j

```
        sll    $t2,$t3,5     # $t2 = i * 2^5
        addu   $t2,$t2,$t4   # $t2 = i*2^5 + j
```

- Get byte address (8 bytes), load x[i][j]

```
        sll $t2,$t2,3        # i,j byte addr.
        addu $t2,$a0,$t2     # @ x[i][j]
        l.d $f4,0($t2)       # $f4 = x[i][j]
```

MIPS code for second piece: **z[ ][ ], y[ ][ ]**

- Like before, but load z[k][j] into $f16

```
L3:    sll $t0,$t5,5          # $t0 = k * 25
       addu $t0,$t0,$t4       # $t0 = k*25 + j
       sll $t0,$t0,3          # k,j byte addr.
       addu $t0,$a2,$t0       # @ z[k][j]
       l.d $f16,0($t0)        # $f16 = z[k][j]
```

- Like before, but load y[i][k] into $f18

```
       sll $t0,$t3,5          # $t0 = i * 25
       addu $t0,$t0,$t5       # $t0 = i*25 + k
       sll $t0,$t0,3          # i,k byte addr.
       addu $t0,$a1,$t0       # @ y[i][k]
       l.d $f18,0($t0)        # $f18 = y[i][k]
```

- Summary: $f4:x[i][j], $f16:z[k][j], $f18:y[i][k]

MIPS code for last piece: add/mul, loops

- Add y*z to x

```
mul.d $f16,$f18,$f16 # y[][]*z[][]
add.d $f4, $f4, $f16 # x[][]+ y*z
```

- Increment k; if end of inner loop, store x

```
addiu $t5,$t5,1      # k = k + 1
bne $t5,$t1,L3       # if(k!=32) goto L3
s.d $f4,0($t2)       # x[i][j] = $f4
```

- Increment j; middle loop if not end of j

```
addiu $t4,$t4,1      # j = j + 1
bne $t4,$t1,L2       # if(j!=32) goto L2
```

- Increment i; if end of outer loop, return

```
addiu $t3,$t3,1      # i = i + 1
bne $t3,$t1,L2       # if(i!=32) goto L1
jr $ra
```