Unit 8 Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Use scientific notation

normalized : 1 digit for the integer part & not 0

not normalized

IEEE Standard

  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized

    2.0×10^{-7}

  - $+987.02 \times 10^{9}$ ← not normalized
- In binary

  must be 1,

  fraction

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

Floating Point Standard

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

IEEE Floating-Point Format

	exp	fraction
Single 32	8.	23
Double 64.	11.	52

+1 sign bit

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction
1.	8/11	23/52

$2^{?}$ 127 : 1?9

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

first bit

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Significand is Fraction with the 1 bit integer and a point, ie, "1."
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single Bias = 127; Double Bias = ~~1203~~ 1023

1 8. 23.
1 11 52.

Single-Precision Range

- Exponents 00000000 and 11111111 reserved

NOT ALLOWED

in machine code

in IEEE format

- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $2.0 \times 2^{+127} \approx 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $2.0 \times 2^{+1023} \approx 1.8 \times 10^{+308}$

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

max 6 decimal digits

  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx$ 16 decimal digits of precision

max 16 decimal digits

Q1 Base 10 → IEEE binary

Floating-Point Example

(if Qn asks 4 digits → 1.100)

significand

- Represent –0.75
  - $–0.75 = (–1)^1 \times 1.1_2 \times 2^{–1}$ ← Manul calculation
  - S = 1 (sign.)
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2

$-0.75_{10} = \boxed{?_2}$

S	exp	Fraction
1	8	23

0.75_{10}
$\times\ 2$
1.5
0.5
$\times\ 2$
1.0

$0.11_2 = 0.75_{10}$

$0.11_2 = 0.11_2 \times 2^0$
$0.11_2 = 1.1_2 \times 2^{-1}$
$\Rightarrow -0.75_{10} = -1.1_2 \times 2^{-1}$
$\Rightarrow -0.75_{10} = (-1)^1 \times 1.1 \times 2^{-1}$

$\underline{Exp} - 127 = -1$
$\Rightarrow Exp = 126$
$= 01111110_2$

Fraction: $1.1 \rightarrow \underbrace{1.00000\ldots000}_{23\ bits}$

⇒ Answer: Single precision.
$1\,|\,\underbrace{01111110}_{8\ bits}\,|\,\underbrace{1000\ldots000}_{23\ bits}$

Double precision:
$Exp = -1 + 1023 = 1022$
$= 01111111110$

- Single: 1011111101000…00 ← Store in Mem
- Double: 1011111111101000…00

Q2 IEEE → Base 10.

Floating-Point Example

- What number is represented by the single-precision float

  exp.

  11000000101000…00

  - S = 1
  - Fraction = 01000…00$_2$ => 1.01
  - Fxponent = 10000001$_2$ = 129 => 129 − 127 = 2.

  $= (-1)^1 \times 1.01 \times 2^2$

  1.01_2

- $x = (-1)^1 \times (\cancel{1 + 01_2}) \times 2^{(129 - 127)}$

  $= (-1) \times \cancel{1.25 \times 2^2}$ $101_2 \times 2^0$

  $= -5.0$

to convert $(-1)^1 \times 1.01 \times 2^2$ to base 10

=> change $2^2 \to 2^0$, shift $1.01 \to 101_2$

$101_2 = 5$ => add the sign → −5

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Q3:-

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points

  smaller to bigger one.
  - Shift fractional number on 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}

MK MORGAN KAUFMANN

(4)

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

$1.000_2 \times 2^{-4}$

	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
Value:	1	0.5	0.25	.125	.0625
Bits:	0	0	0	0	1

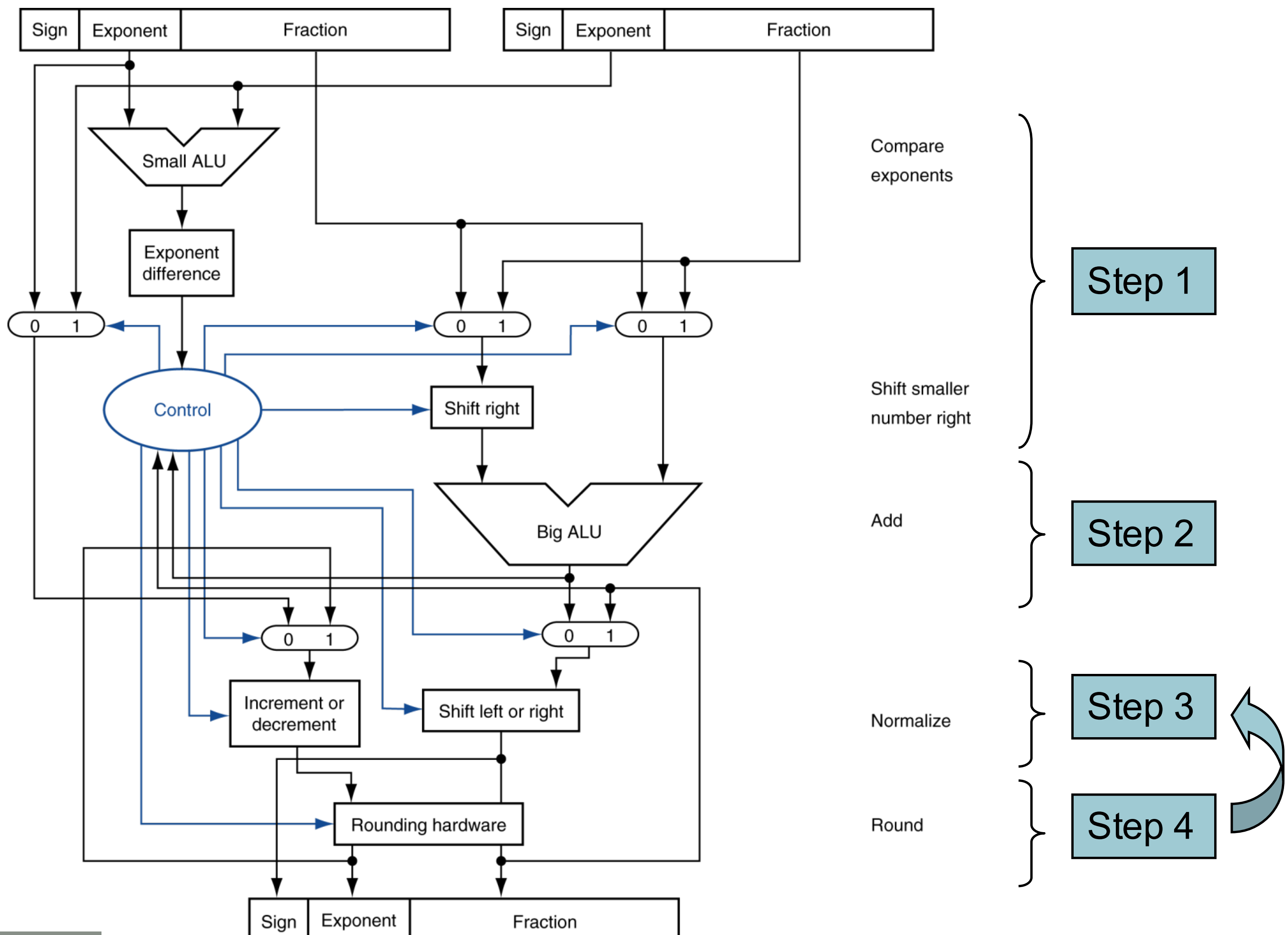
$\Rightarrow 1.000_2 \times 2^{-4} = 0.0625$

FP Adder Hardware

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

FP Adder Hardware

good to know



(5)

Floating-Point Multiplication

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

(6)

Floating-Point Multiplication

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

  if qn does not ask, we keep 4 digit.
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

good to know

- Separate FP registers
  - 32 single-precision: \$f0, \$f1, … \$f31
- FP instructions operate only on FP registers
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

load word floating.

FP Instructions in MIPS

good to know

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is eq, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

good & know

- C code:

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

  - fahr in $f12, result in $f0, literals in global memory space
- Compiled MIPS code:

```
f2c: lwc1   $f16, const5($gp)
     lwc2   $f18, const9($gp)
     div.s  $f16, $f16, $f18
     lwc1   $f18, const32($gp)
     sub.s  $f18, $f12, $f18
     mul.s  $f0,  $f16, $f18
     jr     $ra
```

FP Example: Array Multiplication

g2k

- 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

g^{2k}

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]
```

…

g2k

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
```

Accurate Arithmetic

g 2k

- IEEE Std 754 specifies additional rounding control

  2 extra bits.

  - Extra bits of precision (guard, round)
  - 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

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

g^{2k}

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- X86 FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

g^{2k}

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

Matrix Multiply

g 2k

- Unoptimized code:

```
void dgemm (int n, double* A, double* B, double* C)
{
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
   {
    double cij = C[i+j*n]; /* cij = C[i][j] */
    for(int k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
   }
}
```

Matrix Multiply

g2k

- x86 assembly code:

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx         # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
   element of A
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

Matrix Multiply

- Optimized C code:

```c
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
              _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Matrix Multiply

g 2k

- Optimized x86 assembly code:

```
vmovapd (%r11),%ymm0          # Load 4 elements of C into %ymm0
mov %rbx,%rcx                 # register %rcx = %rbx
xor %eax,%eax                 # register %eax = 0
vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
add $0x8,%rax                 # register %rax = %rax + 8
vmulpd (%rcx),%ymm1,%ymm1     # Parallel mul %ymm1,4 A elements
add %r9,%rcx                  # register %rcx = %rcx + %r9
cmp %r10,%rax                 # compare %r10 to %rax
vaddpd %ymm1,%ymm0,%ymm0      # Parallel add %ymm1, %ymm0
jne 50 <dgemm+0x50>           # jump if not %r10 != %rax
add $0x1,%esi                 # register % esi = % esi + 1
vmovapd %ymm0,(%r11)          # Store %ymm0 into 4 C elements
```

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers ← shifting does not hold the same logic for signed numbers.
  - Arithmetic right shift: replicate the sign bit

shift only works for unsigned numbers

Who Cares About FP Accuracy?

- Important for scientific code
  - for everyday consumer use also
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used including the FP instrucions
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

Unit 8 Homework

1. Use the IEEE754 binary floating point format to show the decimal number “-1.37510” in single and double precision binary floating point format.
2. What decimal number is represented by this IEEE754 single precision floating number ?
   1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 02
3. Add following two decimal floating numbers together, use the binary normalized scientific notation and show the steps, keep the result significand in 4 digits, the exponent in two decimal digits: -0.810 0.62510
4. Multiply following two decimal floating numbers together, use the binary normalized scientific notation and show the steps, keep the result significand in 4 digits, the exponent in two decimal digits: -0.810 0.62510

* MARS Project 3 : Write a MIPS assembly program so that it can convert a Fahrenheit degree to Celsius degree, and convert a Celsius degree to Fahrenheit degree. Using your student ID as the input temperature and making several left shit or right shift to make the student ID number becomes 25 to 35 degree Celsius then convert to Fahrenheit; and then make your student ID doing several left or right shift to become 60 to 100 Fahrenheit degree, then convert o Celsius.