Lecture 7
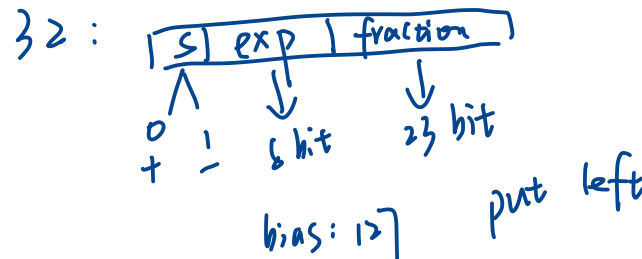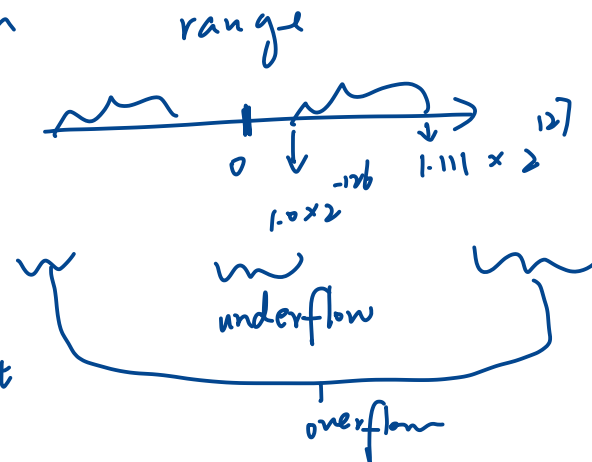
Floating Point Arithmetic

$\left\{\begin{array}{l} 32\text{-bit : single precision} \\ 64 \qquad\quad \text{double} \end{array}\right.$

range

$32:$

| S | exp | fraction |

$0 \uparrow$
$+ \quad - \qquad 8\ bit \qquad 23\ bit$

bias: 127

put left

underflow

overflow

$0 \quad \downarrow \quad 1.111 \times 2^{127}$
$\qquad 1.0 \times 2^{-126}$

$1. xxx \times 2^{y}$

$64: \dfrac{\qquad\qquad\qquad\qquad}{1 \qquad 11 \qquad 52}$

bias: 1023

$\left|\dfrac{A}{A}\right| \le \dfrac{2^{-23} \times 2^{y}}{1 \times 2^{y}} = 2^{-23}$

Lecture 7

Floating Point Arithmetic

add: $\quad x_1 \times 2^{y_1} \quad x_2 \times 2^{y_2}$

1. align
   change the SMALLER one
   $\downarrow$
   shift right

2. add

3. normalize ⎫
4. rounding ⎭ maybe normalize again

multiply:

let $x_1 \times x_2$

add y_1 & y_2 together

Recap

- Operations on integers
  - Addition and subtraction
  - Multiplication and division

Outline

- Floating point representation

- Floating point addition

- Floating point multiplication

- MIPS floating point instructions

- Subword parallellism

Floating Point

- Representation for non-integral numbers

  - Including very small and very large numbers

- Like scientific notation

  - -2.34×10^{56} ← normalized

  - $+0.002 \times 10^{-4}$

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

- In binary

  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

  - 1.*xxxxxxx* is called *significand, mantissa, coefficient*

  - *yyyy* is called *exponent*

- 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

$$\pm 1.xxxxxxx_2 \times 2^y$$

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent (=y+Bias)	Fraction (xxxx)

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

- S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)

- Normalized significant: 1.xxxx
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - $1.0 \leq |\text{significant}| < 2.0$

- Exponent: excess representation: actual exponent (y) + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Floating-Point Example

- Represent −0.75

  - ◆ $−0.75 = -(0.5+0.25) = -0.11_2 = (−1)^1 \times 1.1_2 \times 2^{−1}$

  - ◆ S = 1

  - ◆ Fraction = $1000...00_2$

  - ◆ Exponent = $−1$ + Bias

    - Single: $−1 + 127 = 126 = 01111110_2$

    - Double: $−1 + 1023 = 1022 = 01111111110_2$

- Single: 10111111011000...00

- Double: 101111111111101000...00

Floating-Point Example

- What number is represented by the single-precision float

    11000000101000...00

    - S = 1

    - Fraction = $01000...00_2$

    - Exponent = 10000001_2 = 129

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

    $= (-1) \times 1.25 \times 2^2$

    $= -5.0$

Single-Precision Range

- Exponents 00000000 and 11111111 are 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

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

- Range: $(-2.0 \times 2^{127}, -1.0 \times 2^{-126}], [1.0 \times 2^{-126}, 2.0 \times 2^{127})$

Double-Precision Range

- Exponents 0000…00 and 1111…11 are reserved

- Smallest value

  - Exponent: 00000000001 $\Rightarrow$ actual exponent = 1 − 1023 = −1022

  - Fraction: 000…00 $\Rightarrow$ significand = 1.0

  - $\pm 1.0 \times 2^{-1022}$ ($\approx \pm 2.2 \times 10^{-308}$)

- Largest value

  - Exponent: 11111111110 $\Rightarrow$ actual exponent = 2046 − 1023 = 1023

  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0

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

- Range: $(-2.0 \times 2^{1023}, -1.0 \times 2^{-1022}], [1.0 \times 2^{-1022}, 2.0 \times 2^{1023})$
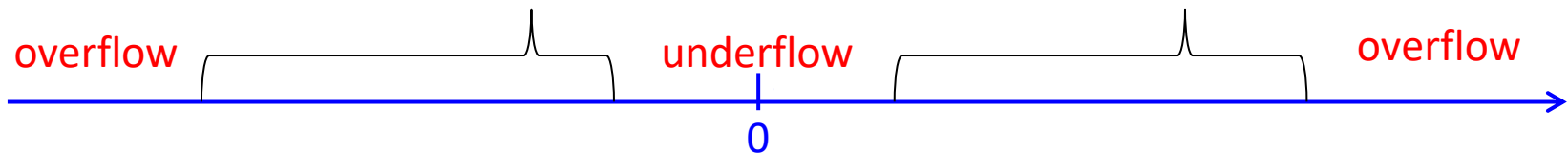
Overflow and Underflow
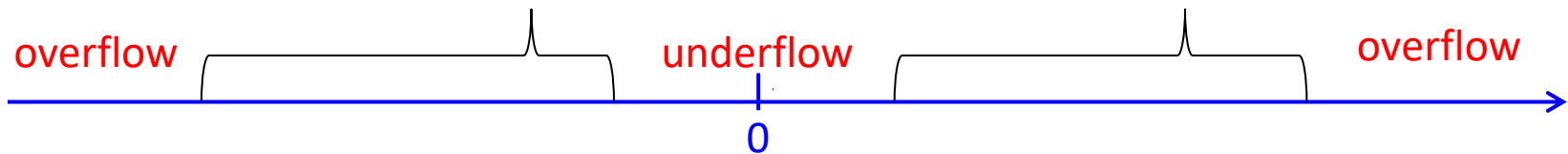
- Range of `float`: $(-2.0 \times 2^{127}, -1.0 \times 2^{-126}], [1.0 \times 2^{-126}, 2.0 \times 2^{127})$

overflow   underflow   overflow

0

- Overflow: when the exponent is too large to be represented

- Underflow: when is negative exponent is too large to be represented (when is exponent is too small to be represented)

- Examples:

  - For `float` number, 8-bit exponent, range: -126~127

  - $1*2^{128}$, $-1.1*2^{129}$ $1*2^{-127}$, $-1.1*2^{-128}$

  - For `double` number, 11-bit exponent, range: -1022~1023

  - $1*2^{1024}$, $-1.1*2^{1026}$ $1*2^{-1023}$, $-1.1*2^{-1025}$

Overflow and Underflow

- Range of `float`: $(-2.0 \times 2^{127}, -1.0 \times 2^{-126}], [1.0 \times 2^{-126}, 2.0 \times 2^{127})$

overflow   underflow   overflow

0

- Overflow: when the exponent is too large to be represented

- Underflow: when is negative exponent is too large to be represented (when is exponent is too small to be represented)

- Examples:

  - For `float` number, 8-bit exponent, range: -126~127

  - $1*2^{128}$, $-1.1*2^{129}$ Overflow; $1*2^{-127}$, $-1.1*2^{-128}$ underflow

  - For `double` number, 11-bit exponent, range: -1022~1023

  - $1*2^{1024}$, $-1.1*2^{1026}$ Overflow; $1*2^{-1023}$, $-1.1*2^{-1025}$ underflow

Reserved Numbers for IEEE 754

- $\pm 0, \pm \infty, \text{NaN}$

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

- $\pm \infty$: divided by 0

- NaN : 0/0, subtracting infinity from infinity

Floating-Point Precision

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent (=y+Bias)	Fraction (xxxx)

- **Relative precision**

  - all fraction bits are significant

  - $\Delta A/|A| = 2^{-23} \times 2^y/|1.xxx \times 2^y|$

    $\leq 2^{-23} \times 2^y/|1 \times 2^y|$

    $= 2^{-23}$

  - 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

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
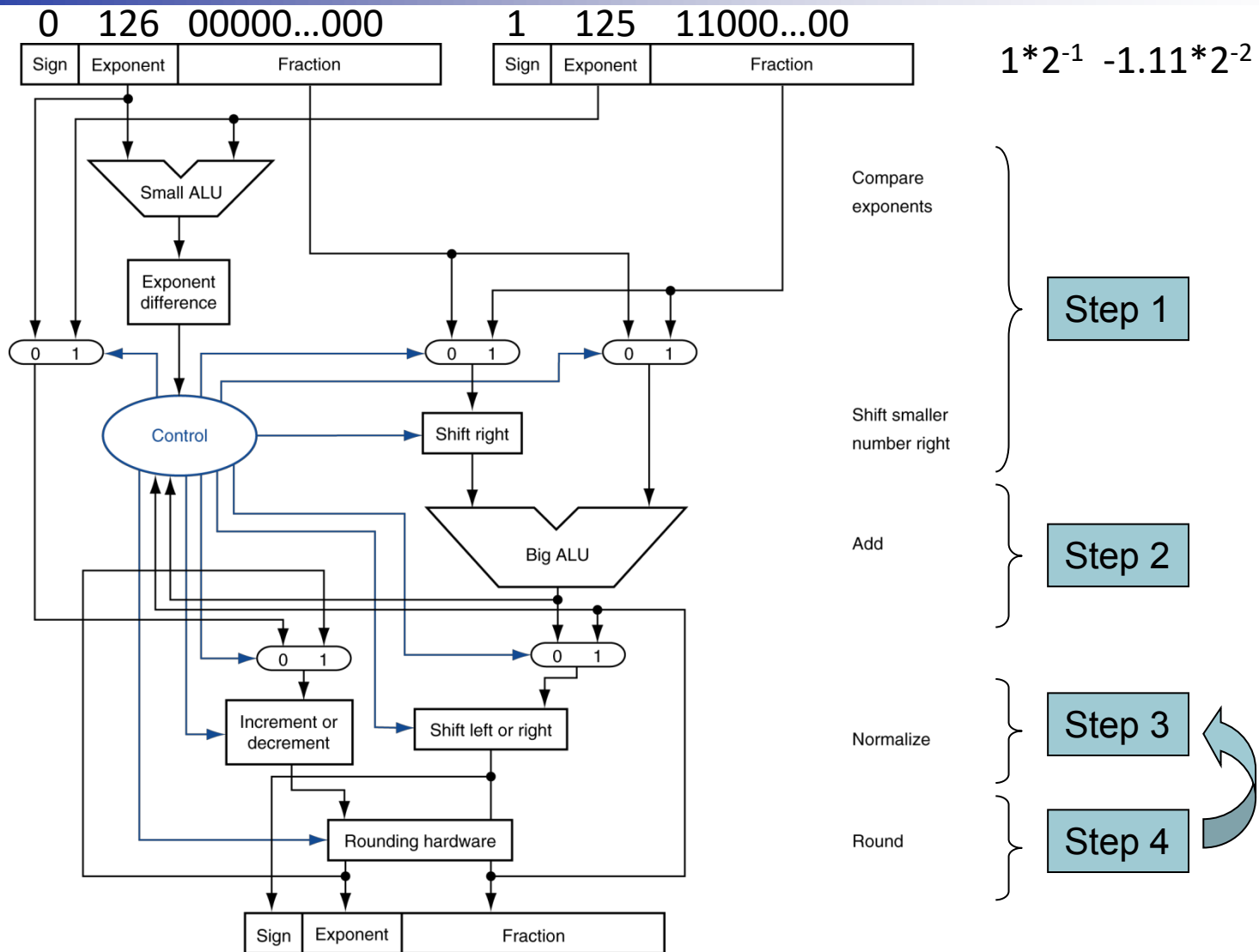 - 1.002×10^2

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 right the 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
  - $1.000_2 \times 2^{-4}$, without over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 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

$1*2^{-1}$ $-1.11*2^{-2}$

0 126 00000...000	1 125 11000...00
Sign	Exponent	Fraction	Sign	Exponent	Fraction

Compare exponents

Step 1

Shift smaller number right

Add

Step 2

Normalize

Step 3

Round

Step 4

FP Multiplication

- $1.000_{two} \times 2^{-1} \times -1.110_{two} \times 2^{-2}$

  - Compute exponent (careful!)

    (-1)+(-2)=-3, 124

  - Multiply significands (set the binary point correctly)

    $1.000 \times 1.110 = 1.110000$

  - Normalize

  - Round (potentially re-normalize)

  - Assign sign -1.11×2^{-3}

FP Arithmetic Hardware

- FP arithmetic hardware usually does
    - Addition, subtraction, multiplication, division, reciprocal, square-root
    - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
    - Can be pipelined

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
    - 1st bit: Guard bit (5), 2nd bit: Round bit (6), 3rd bit: Sticky bit
    - 2.56+234 , 237 vs. 236 (assume that we have three significant digits)

$$\begin{array}{r} 2.3400_{ten} \\ + \ 0.0256_{ten} \\ \hline 2.3656_{ten} \end{array} \qquad \begin{array}{r} 2.34_{ten} \\ + \ 0.02_{ten} \\ \hline 2.36_{ten} \end{array}$$

  - IEEE 754 guarantee one-half (0.5) ulp (units in the last place)
  - Choice of rounding modes
    - Round up, round down, truncate, round to the nearest even (for X.50)
    - To the nearest even: (oct: $\underline{0}$.50→0, $\underline{1}$.50→2, bin: $\underline{0}$.10→0, $\underline{1}$.10 →10)
  - Allows programmer to fine-tune numerical behavior of a computation
- Trade-off between hardware complexity, performance, and market requirements

Float Point 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 $\times$ 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)`

Float Point 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`

Summary of FP Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s    $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s    $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s    $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s    $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d    $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d    $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d    $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d    $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1    $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1    $f1,100($s2)`	Memory[$s2 + 100] = $f1	32-bit data to memory
Condi-tional branch	branch on FP true	`bc1t    25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f    25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

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)   #5.0
     lwc1  $f18, const9($gp)   #9.0
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp) #32.0
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

FP Example: Array Multiplication

- X = X + Y × Z

  only 1/4 time. do 4 adding operation at one time

  - 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

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]
```
…

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
```

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)
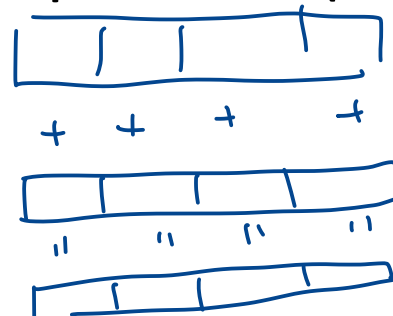
Streaming SIMD Extension 2 (SSE2)

- Adds 4 $\times$ 128-bit registers

  - Extended to 8 registers in AMD64/EM64T

- Can be used for multiple FP operands

  - 2 $\times$ 64-bit double precision

  - 4 $\times$ 32-bit single precision

  - Instructions operate on them simultaneously

    - Single-Instruction Multiple-Data

Matrix Multiply

■ Unoptimized code:

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.   for (int i = 0; i < n; ++i)
4.     for (int j = 0; j < n; ++j)
5.     {
6.       double cij = C[i+j*n]; /* cij = C[i][j] */
7.       for(int k = 0; k < n; k++ )
8.         cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.       C[i+j*n] = cij; /* C[i][j] = cij */
10.    }
11. }
```

Matrix Multiply

- ## 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
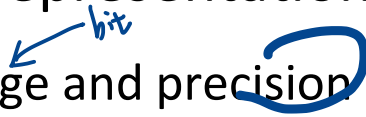
■ Optimized C code:

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.   for ( int i = 0; i < n; i+=4 )
5.    for ( int j = 0; j < n; j++ ) {
6.      __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j]
  */
7.     for( int k = 0; k < n; k++ )
8.      c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.           _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.           _mm256_broadcast_sd(B+k+j*n)));
11.   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.  }
13. }
```

Matrix Multiply

- ■ Optimized x86 assembly code:

```
1.  vmovapd (%r11),%ymm0              # Load 4 elements of C into %ymm0

2.  mov %rbx,%rcx                     # register %rcx = %rbx

3.  xor %eax,%eax                     # register %eax = 0

4.  vbroadcastsd (%rax,%r8,1),%ymm1   # Make 4 copies of B element

5.  add $0x8,%rax                     # register %rax = %rax + 8

6.  vmulpd (%rcx),%ymm1,%ymm1         # Parallel mul %ymm1,4 A elements

7.  add %r9,%rcx                      # register %rcx = %rcx + %r9

8.  cmp %r10,%rax                     # compare %r10 to %rax

9.  vaddpd %ymm1,%ymm0,%ymm0          # Parallel add %ymm1, %ymm0

10. jne 50 <dgemm+0x50>               # jump if not %r10 != %rax

11. add $0x1,%esi                     # register % esi = % esi + 1

12. vmovapd %ymm0,(%r11)              # Store %ymm0 into 4 C elements
```

Concluding Remarks

- Bits have no inherent meaning

    - Interpretation depends on the instructions applied

- Computer representations of numbers

    - Finite range and precision $A \times B \div C$

    - Need to account for this in programs

- ISAs support arithmetic

    - Signed and unsigned integers

    - Floating-point approximation to reals

- Bounded range and precision

    - Operations can overflow and underflow

Midterm

- April 15 (Sat.) 19:00-21:00

- Closed-book, no calculator allowed, a reference page will be provided.

- Content: week 1-8

- No lecture on week 9.

- Q&A session: April 14 (Fri.) 19:00-21:00