

# Lecture 9

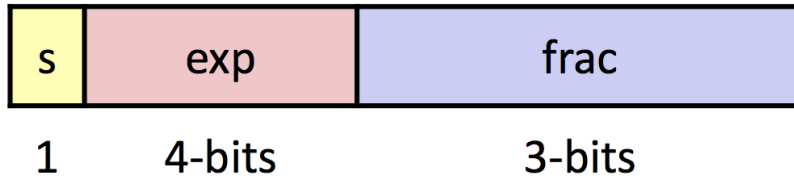
<2016-05-02 Mon>

## Contents

<b>1</b>	<b>Floating Point</b>	<b>2</b>
1.0.1	Tiny Floating Point Example . . . . .	2
1.1	Distribution of Values . . . . .	2
1.2	Special Properties of the IEEE Encoding . . . . .	3
1.3	Floating Point Operation . . . . .	3
1.3.1	Round to Even . . . . .	4
1.3.2	Floating Point Multiplication . . . . .	4
1.3.3	Floating Point Addition . . . . .	5
1.3.4	Floating Point in C . . . . .	6
1.4	Summary . . . . .	7
<b>2</b>	<b>Optimization</b>	<b>7</b>
2.1	Optimizing Compilers . . . . .	7
2.2	Generally Useful Optimizations . . . . .	8
2.2.1	Procedure Calls . . . . .	8
2.2.2	Example of Memory Aliasing . . . . .	8
2.3	Instruction-Level Parallelism . . . . .	9
2.3.1	Superscalar Processor . . . . .	10
2.3.2	Pipelined Functional Units . . . . .	10
2.3.3	Loop Unrolling . . . . .	11

# 1 Floating Point

## 1.0.1 Tiny Floating Point Example



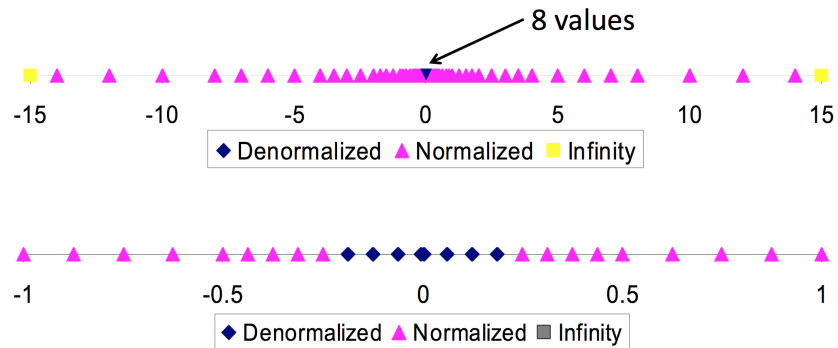
- 8-bit floating point representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last 3 bits are the frac

state	s	exp	frac	E	value	explanation
denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/512	closest to 0
	0	0000	010	-6	2/512	
	...					
	0	0000	110	-6	6/512	
normalized numbers	0	0000	111	-6	7/512	largest denorm
	0	0001	000	-6	8/512	smallest norm
	0	0001	001	-6	9/512	
	...					
	0	0110	110	-1	14/16	
	0	0110	111	-1	15/16	closest to 1 below
	0	0111	000	0	1	
	0	0111	001	0	9/8	closest to 1 above
	...					
	0	1110	110	7	224	
	0	1110	111	7	240	largest norm
	0	1111	000	n/a	inf	

## 1.1 Distribution of Values

- 6-bits IEEE-like format
  - e = 3 exponent bits

- $f = 2$  fraction bits
- bias is  $2^{3-1} - 1 = 3$
- distribution gets denser toward 0



## 1.2 Special Properties of the IEEE Encoding

- floating point 0 is the same as integer 0
  - all bits 0
- can almost use unsigned integer comparison
  - must first compare sign bits
  - must consider  $-0 = 0$
  - NaN (s) problematic
    - \* will be greater than any other values
    - \* what should comparison yield
  - otherwise OK
    - \* denormalized vs normalized
    - \* normalized vs infinity

## 1.3 Floating Point Operation

- basic idea
  - first compute exact result
  - make it fit into desired precision
    - \* possibly overflow if exponent too large
    - \* possibly round to fit into fraction part of float

### 1.3.1 Round to Even

#### 1. Rounding Modes

mode	1.40	1.60	1.50	2.50	-1.50
towards 0	1	1	1	2	-1
round down $-\infty$	1	1	1	2	-2
round up $+\infty$	2	2	2	3	-1
nearest even	1	2	2	2	-2

- round-to-even: default rounding mode
  - hard to get any other kind without dropping into assembly
  - all other are statistically biased
    - \* sum of set of positive numbers will consistently be over or under estimated
- applying to other decimal places / bit positions
  - when exactly halfway between 2 possible values
    - \* **round so that least significant digit is even**

#### 2. rounding binary numbers

- binary fractional numbers
  - "even" when least significant bit is 0
  - "half" way when bits to right of rounding position =  $100 \dots_2$

Table 1: round to nearest  $1/4$  (2 bits right of binary point)

value	binary	rounded	action	bounded
3/32	0.00011	0.00	down	0
3/16	0.00110	0.01	up	1/4
7/8	0.11100	1.00	up	1
5/8	0.10100	0.10	down	1/2

### 1.3.2 Floating Point Multiplication

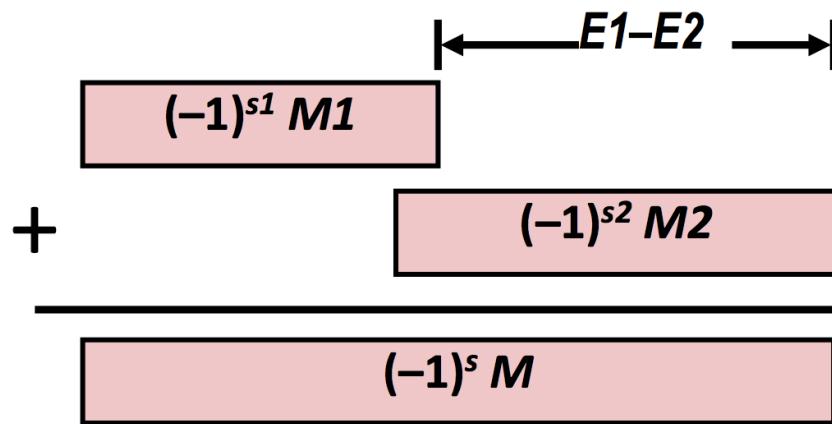
- $(-1)^{s_1} M 2^{E_1} \times (-1)^{s_2} M 2^{E_2}$
- exact result  $(-1)^s M 2^E$ 
  - sign  $s$ :  $s_1 \wedge s_2$

- significand M:  $M1 * M2$
- exponent E:  $E1 + E2$
- fixing
  - if  $M \geq 2$ , shift M right, increment E
  - if E out of range, overflow
  - round M to fit **frac** precision
- implementation
  - biggest chore is multiplying significand

### 1.3.3 Floating Point Addition

- $(-1)^{s1} M 2^{E1} + (-1)^{s2} M 2^{E2}$
- exact result:  $(-1)^s M 2^2$ 
  - sign **s**, significand M:
    - \* result of signed align & add
  - exponent E: E1
- fixing
  - if  $M \geq 2$ , shift M right, increment E
  - if  $M < 1$ , shift M left **k** positions, decrement E by **k**
  - overflow if E out of range
  - round M to fit **frac** range

Get binary points lined up



#### 1.3.4 Floating Point in C

- C guarantees two levels
  - `float` single precision
  - `double` double precision
- conversion/casting
  - casting between `int`, `float`, and `double` changes bit representation
  - `double/float => =int`
    - \* truncates fractional part
    - \* like rounding towards 0
    - \* not defined when out of range or NaN: generally sets to TMIN
  - `int => =double`
    - \* exact conversion, as long as `int` has  $\leq 53$  word size
  - `int => =float`
    - \* will round according to rounding mode

#### 1. Floating Point Puzzle

```
int x;
float f;
double d;
```

condition	expression	result	explanation
f	<code>-(-f)</code>	true	
<code>d &lt; 0.0</code>	<code>(d * 2 &lt; 0.0)</code>	true	
<code>d &gt; f</code>	<code>-f &lt; -d</code>	true	
	<code>d*d &gt;= 0.0</code>	true	
	<code>(d+f) - d == f</code>	false	add a large number d to a small number f, precision over

## 1.4 Summary

- IEEE Floating Point has clear mathematical properties
- represents numbers of form  $M \times 2^E$
- can reason about operations independent of implementation
  - as if computed with perfect precision then rounded
- not the same as real arithmetic

# 2 Optimization

## 2.1 Optimizing Compilers

- provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- don't improve asymptotic efficiency
  - up to programmers to select best overall algorithms
  - big-O savings are (often) more important than constant factors
- have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effect

## 2.2 Generally Useful Optimizations

optimizations that you or the compiler should do regardless of processor/compiler

### 2.2.1 Procedure Calls

- procedure may have side effects
  - alter global state each time called
- function may not return same value for given arguments
  - depends on other parts of global state
  - procedure `lower` could interact with `strlen`

```
void lower(char *s) {  
    size_t i;  
    for (i = 0; i < strlen(s); i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= 'A' - 'a';  
}
```

- `strlen` called multiple times
- move `strlen` outside of loop, since result does not change from 1 iteration to another

### 2.2.2 Example of Memory Aliasing

- aliasing
  - 2 different memory references specify single location
  - easy to happen in C
    - \* since allowed to do address arithmetic
    - \* direct access to storage structures
  - git in habit of introducing local variables
    - \* accumulating within loops
    - \* your way of telling compiler not to check for aliasing



```

void sum_row2(double *a, double *b, long n) {
    long i, j;
    for (int i = 0; i < n; ++i) {
        b[i] = 0;
        for (int j = 0; j < n; ++j)
            b[i] += a[i*n + j];
    }
}

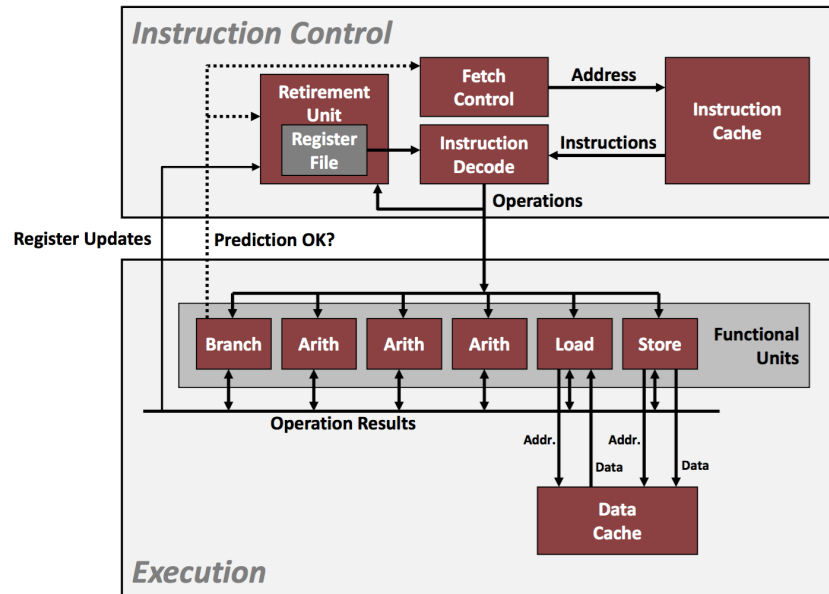
void sum_row2(double *a, double *b, long n) {
    long i, j;
    for (int i = 0; i < n; ++i) {
        double val = 0; /* remove memory aliasing */
        for (int j = 0; j < n; ++j)
            val += a[i*n + j];
        b[i] = val;
    }
}

```

- code updates `b[i]` on every iteration
- must consider possibilities that these updates will affect program behavior

## 2.3 Instruction-Level Parallelism

- need general understanding of modern processor design
  - hardware can execute multiple instructions in parallel
- performance limited by data dependencies
- simple transformations can yield dramatic performance improvement
  - compilers often cannot make these transformations
  - lack of associativity and distributivity in floating point arithmetic



### 2.3.1 Superscalar Processor

- Def: a superscalar processor can issue and execute multiple instructions in one cycle. Instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically
- Benefit: without programming effort, superscalar processor can take advantage of the instruction level parallelism that most programs have
- most modern CPUs are superscalar

### 2.3.2 Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1*p2;
    return p3;
}
```

- divide computations into stages
- pass partial computations from stage to stage

- stage  $i$  can start on new computation once values passed to  $i+1$
- e.g. complete 3 multiplications in 7 cycles, even though each requires 3 cycles (9 cycles in total)

	Time						
	1	2	3	4	5	6	7
Stage 1	$a*b$	$a*c$			$p1*p2$		
Stage 2		$a*b$	$a*c$			$p1*p2$	
Stage 3			$a*b$	$a*c$			$p1*p2$

### 2.3.3 Loop Unrolling

- loop unrolling: perform more useful work per iteration

```

for (i = 0; i < limit; i += 2)
    x = (x OP d[i]) OP d[i+1];

/* loop unrolling with reassociation */
for (i = 0; i < limit; i += 2)
    x = x OP (d[i] OP d[i+1]);

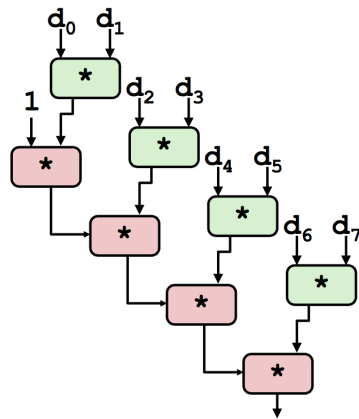
/* loop unrolling with separate accumulators */
for (i = 0; i < limit; i += 2) {
    x0 = x0 OP d[i];
    x1 = x1 OP d[i+1];
}

```

- loop unrolling
  - helps integers add
  - still sequential dependency
- loop unrolling with reassociation
  - faster for floating point
  - nearly 2X speedup for `int *`, `float +`, `float *`
  - breaks sequential dependency
- loop unrolling with separate accumulators

- `int +` makes use of 2 load units
- 2X speedup (over loop unrolling) for `int *`, `float +`, `float *`

```
x = x OP (d[i] OP d[i+1]);
```



```
x0 = x0 OP d[i];  
x1 = x1 OP d[i+1];
```

