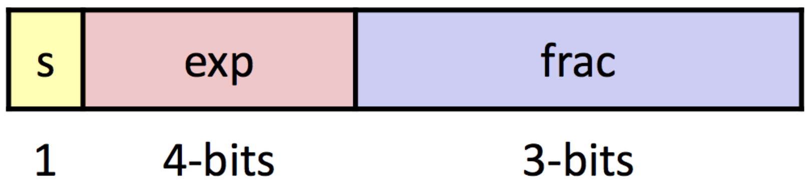
Lecture 9

<2016-05-02 Mon>

Contents

1 Floating Point

1.0.1 Tiny Floating Point Example



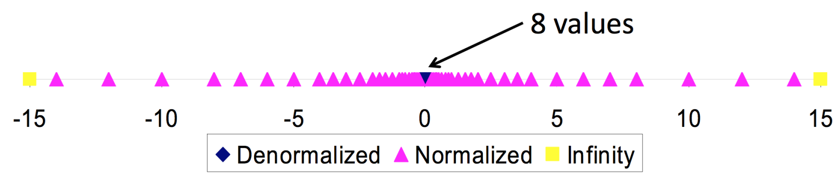
- 8-bit floating point representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last 3 bits are the frac

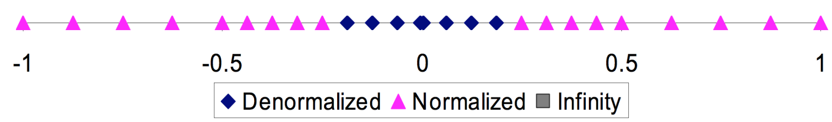
state	s	exp	frac	E	value	explanation
	0	0000	000	-6	0	
denormalized	0	0000	001	-6	1/512	closest to 0
numbers	0	0000	010	-6	2/512	
	...					
	0	0000	110	-6	6/512	
	0	0000	111	-6	7/512	largest denorm
	0	0001	000	-6	8/512	smallest norm
	0	0001	001	-6	9/512	
normalized	...					
numbers	0	0110	110	-1	14/16	
	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 $2^{3\text{-}1}$ - 1 = 3
– ditribution 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
    * possibily overflow if exponent too large
    * possibily 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 -∞	`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 wothout 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 $= 100\ldots_2$

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

- $(-1)^{s1} M 2^{E1} \times (-1)^{s2} M 2^{E2}$
- exact result (-1)$^{\mathrm{s}}$ M 2$^{\mathrm{E}}$
  - sign `s`: `s1 ^ s2`

  - significand `M`: `M1 * M2`
  - exponent `E`: `E1 + E2`

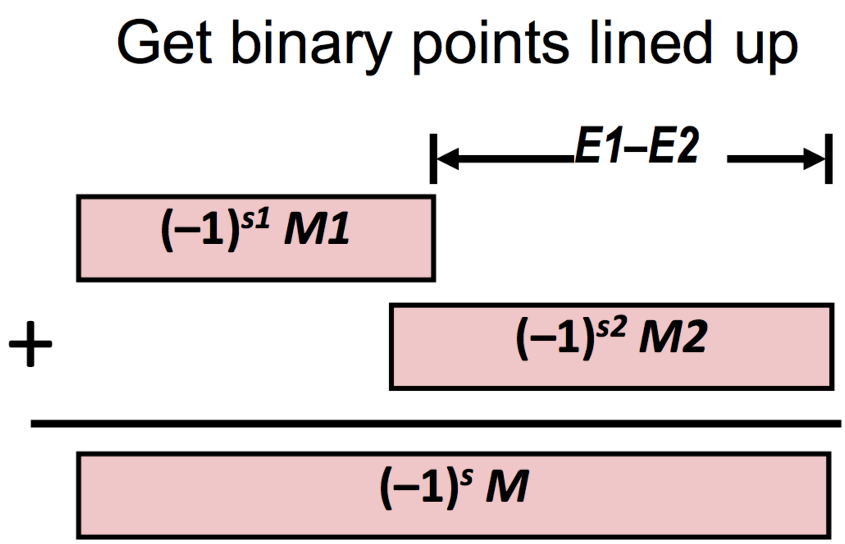
- fixing
  - if `M` $>= 2$, shift `M` right, increment `E`
  - if `E` out of range, overflow
  - round `M` to fit `frac` precision

- 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 $>= 2$, 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



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 $<=$ 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`	`-(-f)`	true	
`d < 0.0`	`(d * 2 < 0.0)`	true	
`d > f`	`-f < -d`	true	
	`d*d >= 0.0`	true	
	`(d+f) - d == f`	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"
  - potiential memory aliasing
  - potiential procedure side-effect

2.2 Generally Useful Optimizations

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

2.2.1 Example of Memory Aliasing

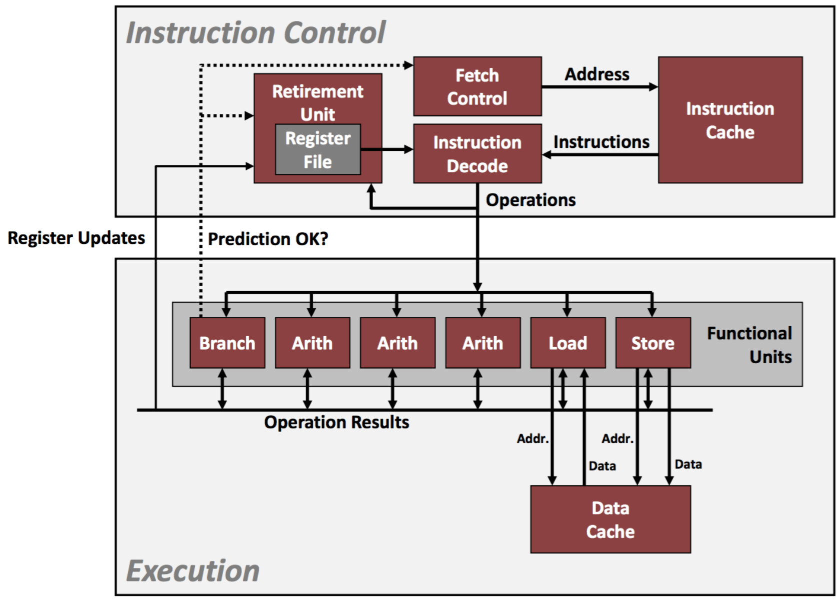
```
void sum_row2(double *a, doubke *b, long n) {
  long i, j;
  for (int i = 0; i < n; ++i) {
    b[i] = 0;
    for (int j = 0; j < n; ++j)
      b[i] += a[i*n + j];
  }
}

void sum_row2(double *a, doubke *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 possiblities 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



2.4 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

2.4.1 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

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**