Floating Point

EE480, Spring 2016

Hank Dietz

http://aggregate.org/hankd/

UNIVERSITY OF KENTUCKY

References

- The EE380 textbook & notes
- A little 16-bit float CGI form:

`http://super.ece.engr.uky.edu:8088/cgi-bin/float16.cgi`

- More online at:

`http://aggregate.org/EE480/`

Remember EE380?

- IEEE 754 standard
  - Standardized formats & accuracy
  - Sign + magnitude encoding of fraction
  - 2's comp. exponent (plus bias)
- Basic operations:
  - Add and subtract
  - Multiply
  - Reciprocal

IEEE 754 Details

- Predictive infinities and NaNs
  - Gracefully overflow with +/- ∞
  - NaNs really about *when to handle errors*
- Denormalized numbers
  - Values near 0 don't get normalized
  - Often simplified to just 0 as a special case
- Rounding modes
  - Can specify rounding up, down, toward 0…
  - Extra “guard” bits used to preserve accuracy

Let's Keep It Simple Here...

- Simplified 16-bit floating point…
  just implement the top 16 bits of 32-bit float
  - No ∞ nor NaN, and 0 is the only subnormal
  - No rounding, guard bits (lousy accuracy)
- float16[15] sign bit, 1 means negative
- float16[14:7] exponent, +bias
- float16[6:0] mantissa; normalized implies 1 MSB
- `http://super.ece.engr.uky.edu:8088/cgi-bin/float16.cgi`

Can Use 32-bit `float` To Check

- Just implements the top 16 bits of 32-bit float so just look at those bits…

```
typedef unsigned short float16;
float16 f16; unsigned int i;

i = (f16 << 16);
 … *((float *) &i) …
f16 = (I >> 16);
Sign = ((i & 0x8000) ? 1 : 0);
Exp = ((i >> 7) & 0xff);
Frac = ((I & 0x7f) + (f16 ? 0X80 : 0));
```

Normalization Issues

- Normalization requires shifting until 1 in MSB
  - Need to count leading zeros
  - Barrel shift to the left (multiply by 2^k)
- For addition, denormalization requires shifting
  - Pick smaller exponent, compute difference
  - Barrel shift to the right (divide by 2^k)
- You want to do this combinatorially…

Barrel Shifter

- Simple trick: $\log_2$ decomposition

```
module srl(dst, src, shift);
output reg[7:0] dst; input wire[7:0] src, shift;
reg[7:0] by1, by2, by4;
always @(*) begin
  by1 = (shift[0] ? {1'b0, src[7:1]} : src);
  by2 = (shift[1] ? {2'b0, by1[7:2]} : by1);
  by4 = (shift[2] ? {4'b0, by2[7:4]} : by2);
  dst = (shift[7:3] ? 0 : by4);
end
endmodule
```

Counting Leading Zeros

- A binary search for the most significant 1 bit

```
module lead0s(d, s);
output reg[4:0] d; input wire[15:0] s;
reg[7:0] s8; reg[3:0] s4; reg[1:0] s2;
always @(*) begin
 if (s[15:0] == 0) d = 16; else begin
  d[4] = 0;
  {d[3],s8} = ((|s[15:8]) ? {1'b0,s[15:8]} : {1'b1,s[7:0]});
  {d[2],s4} = ((|s8[7:4]) ? {1'b0,s8[7:4]} : {1'b1,s8[3:0]});
  {d[1],s2} = ((|s4[3:2]) ? {1'b0,s4[3:2]} : {1'b1,s4[1:0]});
  d[0] = !s2[1];
 end
end
endmodule
```

Addition Algorithm: `r=a+b`

- Denormalize so that `a `EXP == b `EXP`
- Add/subtract fractions, depending on signs
- Set sign of result
- Normalize

Multiplication Algorithm: `r=a*b`

- Set sign of result
- Add exponents
- Multiply fractions (8 bit * 8 bit)
- Normalize

Reciprocal Algorithm: `r=1.0/x`

- Guess & iteratively refine guess
- Note that `2.0f` in our format is `0x4080`

```
typedef union { float f; int i; } fi_t;
float recip(float x)
{
  fi_t t;
  t.f = guess(x);
  t.f *= (2.0f – (t.f * x)); // 1st iter
  t.f *= (2.0f – (t.f * x)); // 2nd iter
  return(t.f);
}
```

Reciprocal Algorithm: `r=1.0/x`

- A really sneaky way to guess, using the fact that $1.0/2^n$ is 2^{-n}, which can be computed by int sub…

```
typedef union { float f; int i; } fi_t;
float recip(float x)
{
  fi_t t;
  t.f = x;
  t.i = magic – t.i; // guess
  t.f *= (2.0f – (t.f * x)); // 1st iter
  t.f *= (2.0f – (t.f * x)); // 2nd iter
  return(t.f);
}
```

Reciprocal Algorithm: `r=1.0/x`

- Try all; best `magic` is **`0x7eea`**
  average 3.98 bits bad without iterations!
- Min max error is 7 bits, using `0x7f00`

```
typedef union { float f; int i; } fi_t;
float recip(float x)
{
  fi_t t;
  t.f = x;
  t.i = magic – t.i; // guess
  return(t.f);
}
```

A Better Reciprocal Guess

- Can actually do better quite easily using a lookup table (ROM) to invert the mantissa
  - Low 7 bits of mantissa replaced by lookup
  - Exponent is either:
    - `254 – exp` *iff low mantissa bits were 0*
    - `253 – exp` *otherwise*
- Note that subnormals are still special cases; 1/0 should produce NaN, but we'll allow 0 here
- Iteratively improve if more mantissa bits needed

Reciprocal Lookup Table

- Here's the 7-bit mantissa reciprocal table:

```
00,7e,7c,7a,78,76,74,72,70,6f,6d,6b,6a,68,66,65,
63,61,60,5e,5d,5b,5a,59,57,56,54,53,52,50,4f,4e,
4c,4b,4a,49,47,46,45,44,43,41,40,3f,3e,3d,3c,3b,
3a,39,38,37,36,35,34,33,32,31,30,2f,2e,2d,2c,2b,
2a,29,28,28,27,26,25,24,23,23,22,21,20,1f,1f,1e,
1d,1c,1c,1b,1a,19,19,18,17,17,16,15,14,14,13,12,
12,11,10,10,0f,0f,0e,0d,0d,0c,0c,0b,0a,0a,09,09,
08,07,07,06,06,05,05,04,04,03,03,02,02,01,01,00
```

`float`/`int` Conversions

- An integer is a denormalized float…
- 16-bit `int` to `float`:
  - Make int positive, set sign
  - Take most significant 1 + 7 more bits
  - Set exponent to normalize result
- `float` to 16-bit `int`:
  - Take (positive) 8-bit fraction part
  - Barrel shift integer appropriately
  - Negate if sign was set