Arithmetic Functions

Fixed and Floating Point

2s Complement Numbers

- Numbers as Integers
- Numbers as Fractions
- Numbers as Fractions with Scale

Adding Signed Integers

- Single Precision, Fixed Point

- Add numbers, use 2s complement representation for negative numbers.

- Multiple Precision

- Add multiple numbers (48, 64 bit arithmetic)
- Need to add (subtract) with Carry

Adding Fractions

- Can follow same methodology, with fixed "binary point"
- Numbers must position the binary point in the same place.
- 2s complement works for fractions

Numbers with Scale

- Can Represent as Integer or Fraction with Scale

- Adding Numbers Requires COMMON SCALE

e.g., 32 bits, + scale N, where N represents multiplier of 2^N.

Negative Fractions, using 2's Complement

- Where do we put the binary point?
- Can put it to the right of the sign bit, and assign the sign a value of 1.
- Therefore, the value to the right of the sign is ½, and the value of the sign is -1 !

Example Signed Fractional Add

- 0.1000000 scale 0 means + ½
- 1.1000000 scale 0 means – ½
- Adding them together produces:
- 0.000000000 scale 0

Note: shifting right requires sign propagation!

Example Signed Fraction with Scale

0.0001000 , scale 0 => 1/16

0.1000000 , scale 0 => ½

Add them together, we get

0.1001000 , scale 0 => 9/16

Fraction with scale

- 0.1000000 scale 3 => ½ * (2**3) = 4
- 0.1000000 scale 0 = ½
- Add them together ?
- Have to put them on common scale first: right-shift the number with the smaller scale by the difference between the scales:
- 0.0001000 scale 3 = ½
- Now add them to get 0.1001000, scale 3

Normalization

- Shift left N bits until the sign bit does not equal the adjacent bit

- 0.0000100 scale 0 = 0.1000000 scale -4 = 1/32

Multiplication

- Unsigned Decimal Multiplication

```
Decimal:    323
          x 126
            1938     = 6 x 10^0 x (323)
          + 6460     = 2 x 10^1 x (323)
            8398    (subtotal)
        +  32300     = 1 x 10^2 x (323)
           40698     product
```

multiplier x multiplicand = product

Unsigned Binary Multiplication

```
     0011 (3 x 3)
   x 0011
     0011  subtotal 1
   +00110
=    01001  subtotal 2
  +000000
=   001001  subtotal 3
 +0000000
= 0001001  subtotal 4 (3x3=9)
```

Unsigned Binary Multiplication

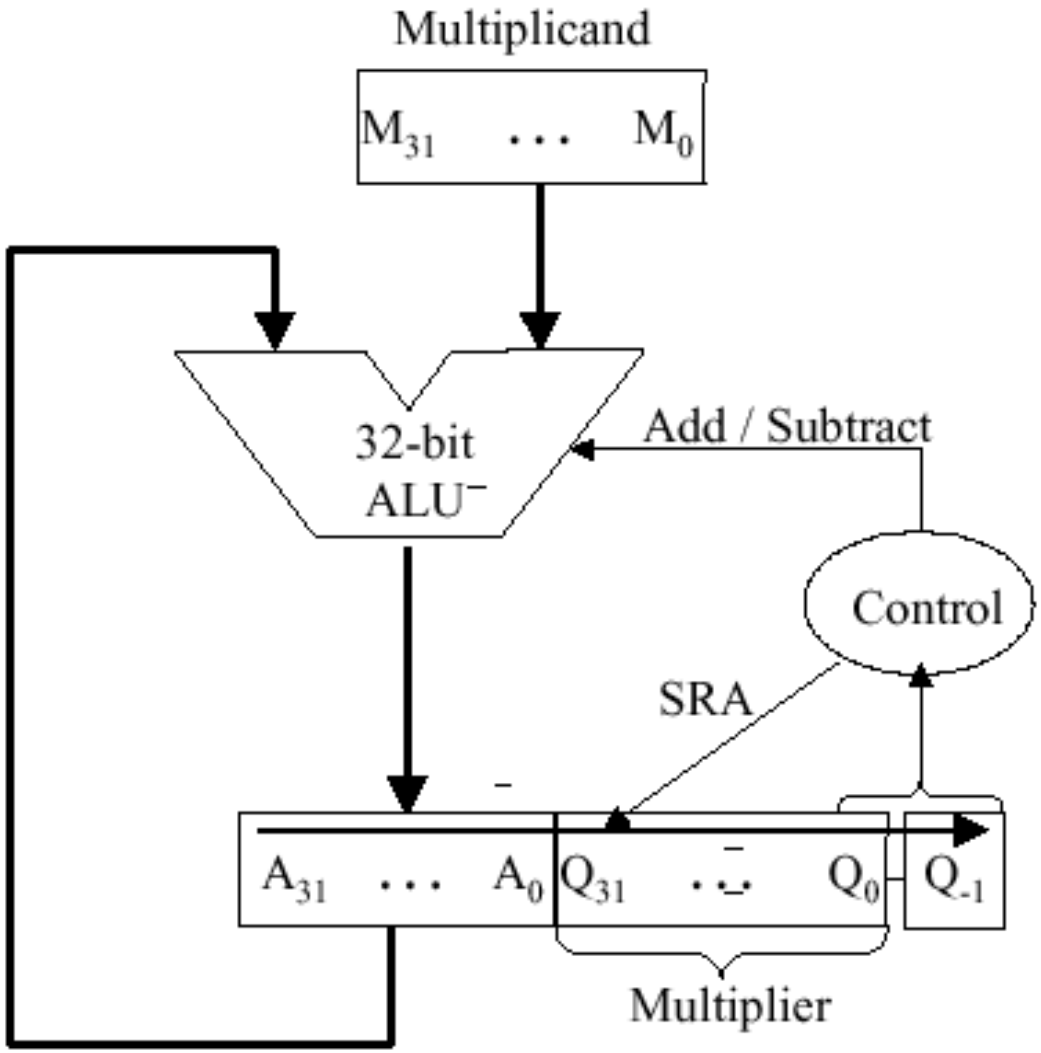
```
     0111   (7x5)
   x 0101
     0111   subtotal 1
   +00000
=  000111   subtotal 2
  +011100
=  100011   subtotal 3
 +0000000
= 0100011   subtotal 4  (0x23 = 35)
```

Unsigned Binary Multiplication

```
     1111   (15x15)
   x 1111
    01111   subtotal 1
   +11110
=  101101   subtotal 2
 + 111100
= 1101001   subtotal 3
+ 1111000
=  11100001  subtotal 4
```

Maximum value of unsigned 15x15 = 0xE1 = 225

Multiplier Logic Elements



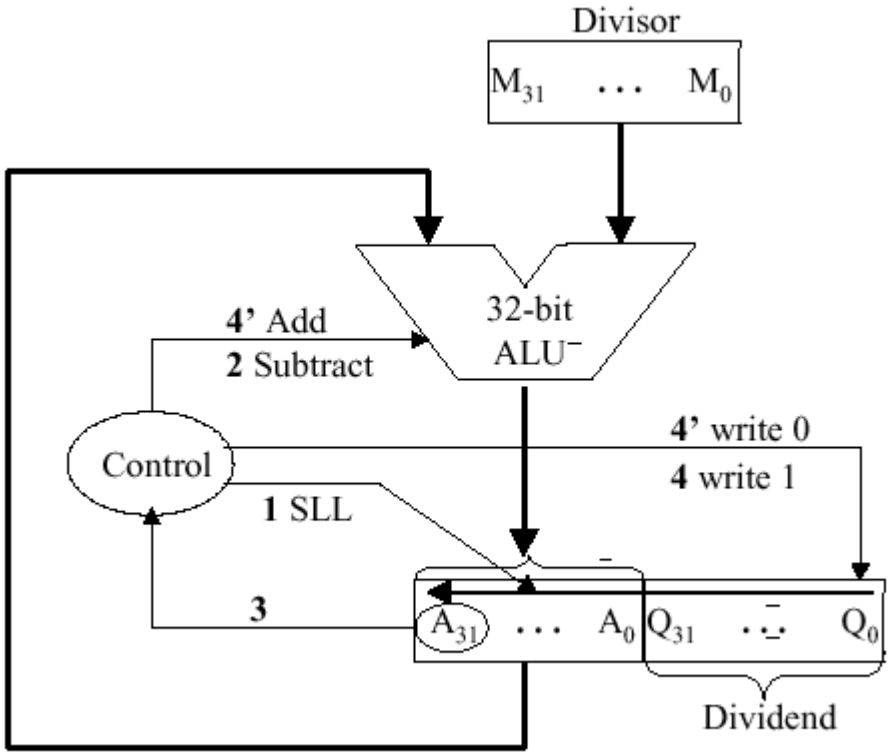
Example scaled fraction

0.10000000

Division

```
                  1 0 0 1 0
1000  |  1 0 0 1 0 1 0 0
         -1 0 0 0
               1
               1 0
               1 0 1
               1 0 1 0
              -1 0 0 0
                    1 0
```

ALU Divider Circuit



MIPS Math Operations

- div – fixed point divide (Lo=quot; Hi=rmdr)
- divu
- mult – fixed point mult (hi,lo) = R[Rs]*R[Rt]
- multu
- mfhi – Move from hi register
- mflo – Move from lo register
- Uses “high” and “low” registers

Floating Point Operations

- Require a standard representation for floating point (scaled numeric) numbers
- Typically operate using a sign-magnitude with an exponent
- IEEE 754 is a standard representation for single and double precision
- Single: 1 sign, 8 exponent, 23 fraction
- Double: 1 sign, 11 exponent, 53 fraction
- Double Extended, 1 sign, 15 exponent, 64 “fraction”
- Quad (128) and Half (16) precision also possible

Format vs. Operations

- IEEE 754 is Storage format, but doesn't describe internal operation
- Floating Point Operations can be done as software subroutines
- Coprocessor – main processor waits for FPU
- 'Coprocessor 1' on MIPS
- Independent FP Regs (32 Single, 16 Double) on MIPS.

Extending Precision

- If Single Precision, then fraction fits within 32 bits (it's only 23 bits!) so we can perform normal addition or subtraction operations (after we place the fractions on a common scale) with 32 bit arithmetic
- If Double Precision, then fraction is bigger than a 32 bit word, and we have to use extended precision using adds with carry.

Implementation using IEEE 754

- The “hidden sign” and “bias” used by IEEE 754 are not readily useful for internal arithmetic.
- Best practice: Prefix your floating point operation by “unpacking” the IEEE 754 value into variables that easily represent the fraction and scale. These should be easily convertible to 2’s complement numbers.
- After the operation, repack the number using the IEEE 754 convention.

Before we begin:

IEEE 754 funny values

- Plus zero = 0
- Minus zero has sign=1, remaining portion = 0
- Plus Infinity, sign=0, exponent=11111's, fraction is 0
- Minus infinity, sign=1, exponent=11111's, fraction is 0
- Not A Number "NaN", sign=0, exponent=11111's, fraction is not zero.

Use of “Bias”

- The Bias value (127 or 1023) is subtracted from the exponent to get the actual power of 2 exponent.

- This means we can compare (sort) floating point numbers as if they were signed integers!

Extracting the Exponent

- In IEEE 754 Single Precision, the “biased exponent” is located in bits 23-30 (an 8 bit field)
- If the entire word is zero (see previous slide), the value is zero. Add operations of zero do nothing.
- The bias amount is decimal 127. To obtain the “real” exponent, subtract 127 from the biased exponent to product a signed number. Use 2’s complement representation for the internal exponent.

Obtaining the Fractional Portion

- If the floating point number is not zero, then the fractional portion has a hidden ‘1’ bit, whose value is 1.0.
- The fraction (hidden 1) can be placed in bit 30 of a 2’s complement representation for the fraction.
- The remaining fraction bits are placed in the next 23 bits to the right of the hidden 1 (bits 6-29). Bits 0-5 are set to 0.
- The fraction thus takes 25 bits, including a reserved sign bit, with the binary point being located between bits 29 and 30.

Represent Fraction as Signed, 2's Complement

- IFF the floating point number is negative (i.e., the sign=1), then negate the fraction using standard 2's Complement arithmetic.
- Now represented as a 2's complement number, the fraction's sign bit will be 1 if it is negative.

Unpack 754 Method (single prec)

Summary

- Test for zero – if so, set everything to 0.
- Move fraction to internal fraction, shifted right 2 bits.
- Set 0 as sign bit, ‘1’ as next bit (hidden 1). Fraction thus appears a positive number
- Extract exponent, subtract 127
- Add ‘1’ to exponent to account for shift right above.
- Convert fraction to negative using 2’s complement negate *if sign=1.*

Internal Representation

- The IEEE floating point number is now represented by two 2's Complement numbers:
- A signed Exponent, E, consisting of an exponential multiplier 2^E.
- A Signed Fraction F, with a binary point two bits from the left edge. The value of the number is thus:

$$F \times 2^E$$

Floating Point Add

- Place two operands on a common scale – operand with smaller scale gets shifted right N bits and scales are set to the same value.
- Shift right 1 bit before adding to avoid overflow; do not forget to compensate scale by adding +1!
- Numbers must be negated using 2's complement convention if sign is 1.
- Fractions are added
- Result must be negated if sign is 1; set sign of result.
- Result is normalized.

Example Floating Number: -3.5

- IEEE 754 value: 0xC0600000
- Sign = 1 (negative)
- Exponent, E, = 128 – 127 = 0x00000001
- Fraction, F, = (1 +), or 0x70000000
- 2’s Complement Fraction: (negative)

  0x90000000

Shifting Rule

- Can shift a number right or left, keeping its value by adjusting exponent.
- For Example:

  0x90000000 with Exponent 1 is equal to:

  0xC8000000 with Exponent 2, also equal to:

  0xF9000000 with Exponent 5

Add 1 to Exponent for every bit shifted right,
Subtract 1 from Exponent for every bit shifted left.

Example Floating Number: +0.5

- IEEE 754 value: 0x3F000000
- Sign = 0 (positive)
- Exponent, E, = 126 – 127 = -1 =0xFFFFFFFF
- Fraction, F, = (1 +)0, or 0x40000000
- 2’s Complement Fraction: (positive) (same)

Avoiding Overflow

- Because each number may consist of leading 1, need to shift each number right before adding them, adjusting the (common) scale by +1. Common scale is now 2.
- Add Numbers (shifted right):

```
  C8000000
+ 08000000
  D0000000
```

Add the Numbers

- Have to place on common scale.
- Number with lower Exponent gets its fraction shifted right N positions
- +0.5 has lower Exponent (-1), and the difference N is 1-(-1) = 2.
- New scaled number (B) is:

Exp = +1, Fraction = 0x10000000

Take 2's Complement of Negative

- D0000000 negated is: 0x30000000 (Exp 2).
- Set sign flag in result (negative).
- Normalize Fraction: 0x60000000 (Exp 1)
- Resulting sign=1, Exponent=127+1 = 128
- Removing hidden '1', Fract = 0b1000...., or 1.5
- Result value is $-(2^1 \times 1.5) = -3.0$

How To Normalize

- Test Fraction to see if 0. If so, result is 0 !
- While the Fraction does not have a ‘1’ in the ‘1’ position (left of the binary point)

  {

  shift left 1 bit

  decrement Exponent

  }

Pack Result Back to IEEE 754 Format

- Set Sign (Bit 31)
- Bias is 127
- Set Exponent + Bias in Bits 30-23
- Lose Bit 30 of Fraction (hidden ‘1’)
- Move Fraction Bits 29-6 to IEEE Fraction

Floating Point Negate

- Change Sign Bit

Floating Point Multiply

- Obtain “internal” 2’s Complement Representations of Exponents and Fractions, as above
- Multiply Fractions using Fixed Point Multiply
- Add Exponents
- Normalize
- Repack as IEEE Single Precision

Example:

- IEEE 754 value for 1.0 is: 0x3F800000
- sign=0, exponent=0x7E=127, fraction=0
- hidden ‘1’ therefore, $1x2^0$

- Internal format is: f=0x40000000 e=0x00000001

Example

- -8.5 = 0xC1080000
- sign=1, exp=130, fract=0001000…

- Extracted value:

  fraction=0x44000000, exp=(130-127) +1 = 4

  neg., 2’s compl=0xBC000000

MIPS mult Instruction

- Assumes binary point is between bits 30 and 31 (immediately to the right of the sign)
- However, our internal representation assumes it is one bit to the right of that (between bits 29 and 30) !
- Multiplying our fractions causes the result to be shifted two bits to the right
- Have to multiply by 4 after mult to compensate. (shift left 2 bits)

mult, Example

```
.data
arg1:   .word   0x40000000          # +1.0, fixed point
arg2:   .word   0x40000000  # + 1.0, binary point between bits 29&30

.text
#       Note: mult instruction assumes binary point immediately
#       after sign bit! Therefore, We have to scale up by 4 after mult.

main:   lw      $t1, arg1  # argument a (scaled by 1/2)
        lw      $t2, arg2  # argument b  (scaled by 1/2)
        mult    $t1, $t2    # compute a*b
        mfhi    $t0 # high portion of product
        sll     $t0,$t0, 2  # scale x 4 (correct for binary point)
                                        # the product is in $t0
```

Convert/Pack IEEE 854

- Convert 2's complement fraction to sign s and magnitude fraction.
- If not zero, shift fraction left 1 and decrement exponent.
- Convert exponent to IEEE 754 format by adding "bias" value of 127.
- Store sign, biased exponent, and fraction portion into IEEE 754 result.

Special Cases

- Exponent Overflow – produce special values for plus/minus infinity (if exponent too bit)
- Exponent Underflow – can be replaced with 0 if resulting number too small.
- Significand underflow – possible rounding if bits flow off the right edge when placing on a common scale
- Significand overflow - can be fixed by realignment.

Examples

TEST: 0.0
f = 0.000000, f(hex) = 00000000, Special case: ZERO
d = 0.000000, d(hex) = 00000000 00000000, Special case: ZERO

TEST: 1.0
f = 1.000000, f(hex) = 3F800000, fraction = .00000000, (1 + fraction) x 2**0
d = 1.000000, d(hex) = 3FF00000 00000000, fraction(hi) = .00000000, (1 + fraction) x 2**0

TEST: -1.0
f = -1.000000, f(hex) = BF800000, fraction = .00000000, (1 + fraction) x 2**0
d = -1.000000, d(hex) = BFF00000 00000000, fraction(hi) = .00000000, (1 + fraction) x 2**0

TEST: 0.5
f = 0.500000, f(hex) = 3F000000, fraction = .00000000, (1 + fraction) x 2**-1
d = 0.500000, d(hex) = 3FE00000 00000000, fraction(hi) = .00000000, (1 + fraction) x 2**-1

TEST: 0.25
f = 0.250000, f(hex) = 3E800000, fraction = .00000000, (1 + fraction) x 2**-2
d = 0.250000, d(hex) = 3FD00000 00000000, fraction(hi) = .00000000, (1 + fraction) x 2**-2

Examples

TEST: 32.0

f = 32.000000, f(hex) = 42000000, fraction = .00000000, (1 + fraction) x 2**5

d = 32.000000, d(hex) = 40400000 00000000, fraction(hi) = .00000000, (1 + fraction) x 2**5

TEST: 67.0

f = 67.000000, f(hex) = 42860000, fraction = .0C000000, (1 + fraction) x 2**6

d = 67.000000, d(hex) = 4050C000 00000000, fraction(hi) = .0C000000, (1 + fraction) x 2**6

TEST: 33554431

f = 33554432.000000, f(hex) = 4C000000, fraction = .00000000, (1 + fraction) x 2**25

d = 33554431.000000, d(hex) = 417FFFFF F0000000, fraction(hi) = .FFFFF000, (1 + fraction) x 2**24

Examples

TEST: 1.0 / 0.0

f = inf, f(hex) = 7F800000, fraction = .00000000, (1 + fraction) x 2**128

d = inf, d(hex) = 7FF00000 00000000, fraction(hi) = .00000000, (1 + fraction) x 2**1024

TEST: -1.0 / 0.0

f = -inf, f(hex) = FF800000, fraction = .00000000, (1 + fraction) x 2**128

d = -inf, d(hex) = FFF00000 00000000, fraction(hi) = .00000000, (1 + fraction) x 2**1024

TEST: 0.0 / 0.0

f = -nan, f(hex) = FFC00000, fraction = .80000000, (1 + fraction) x 2**128

d = -nan, d(hex) = FFF80000 00000000, fraction(hi) = .80000000, (1 + fraction) x 2**1024

IEEE 754 Additions

- Half Precision: 16 bits,

  Sign, 5 bits of Exponent (Bias 15),

  10 bits of Fraction

- Quad Precision: 128 bits,

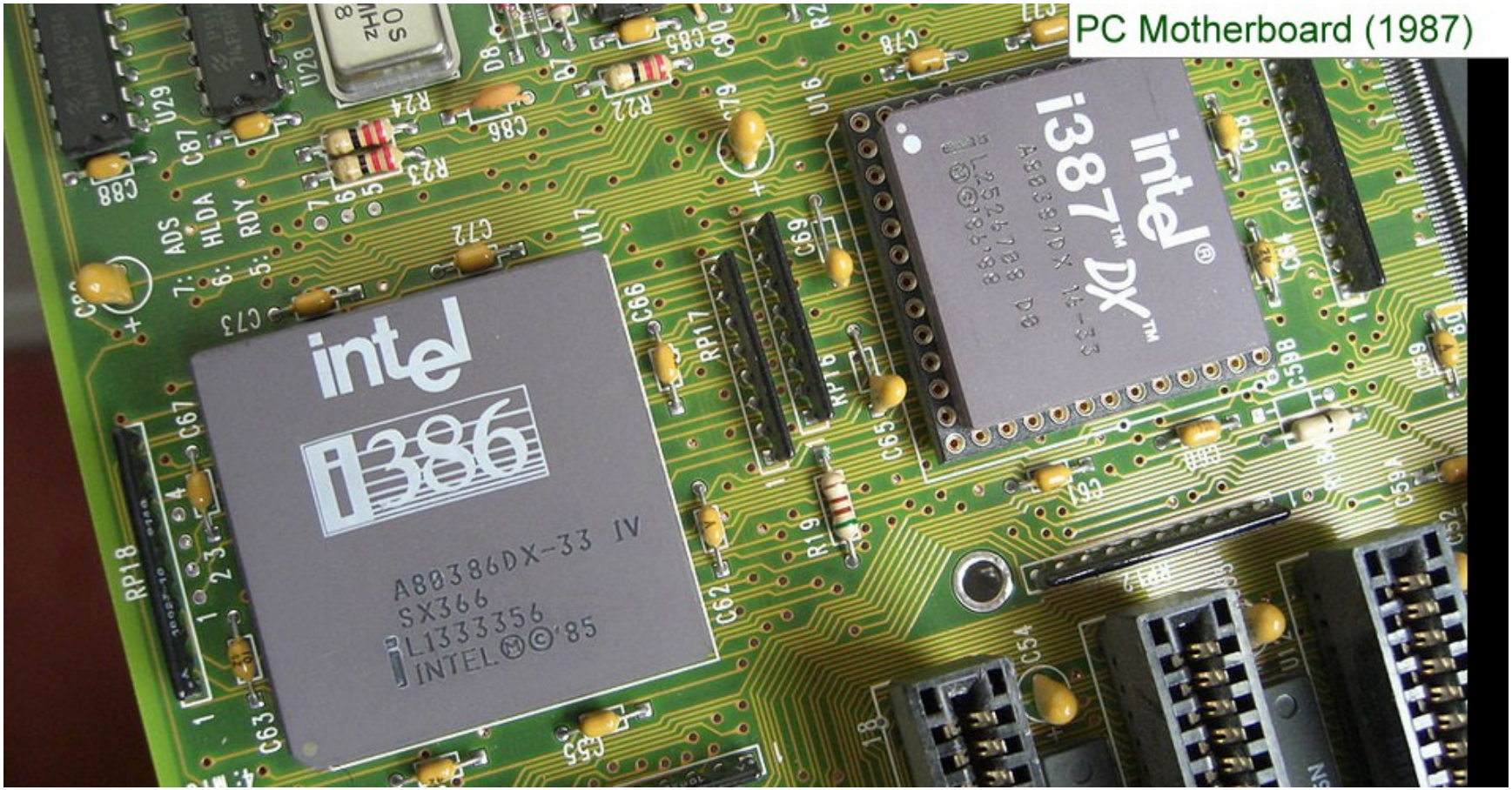
  Sign, 15 bits of Exponent (Bias 16383),

  112 bits of Fraction

Floating Point Implementation

- Software – Subroutine Library
- Coprocessor
  - Runs as I/O Device
  - Runs in parallel with CPU
  - Can run asynchronously
  - Can have multiple instances
- Extension to Instruction Set

FPU as Coprocessor



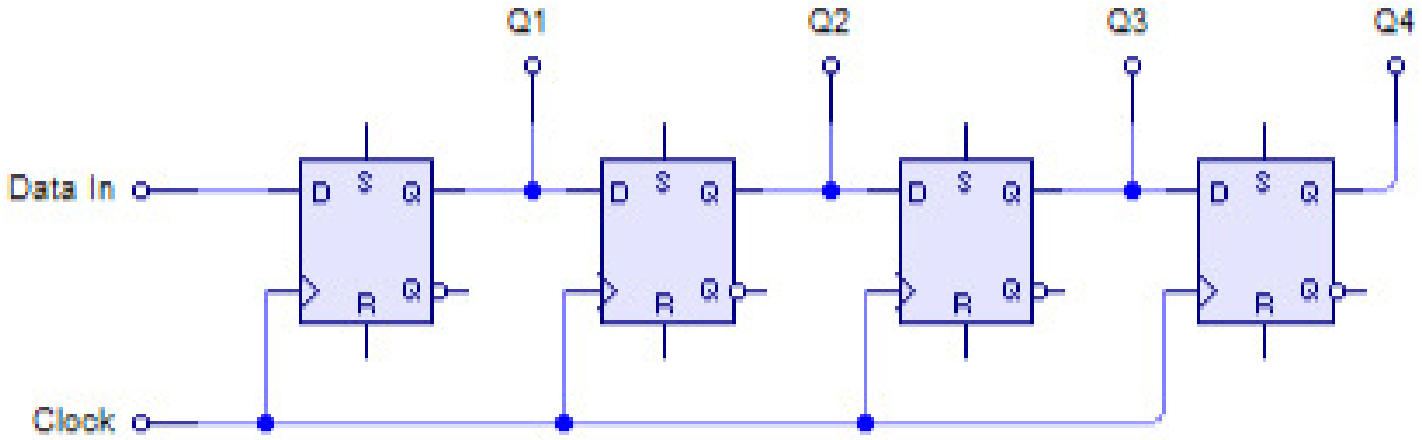
IEEE 754- HOWTO

- Extract/Unpack IEEE 754 number
- have internal representation that uses 2’s complement.
- fraction and exponent stored as 32-bit words in MIPS registers
- fraction is stored in high-order portion of 32-bit word, with binary point to the right of sign bit.
- exponent stored as a 2’s complement, signed integer.

Shifting with Special Hardware

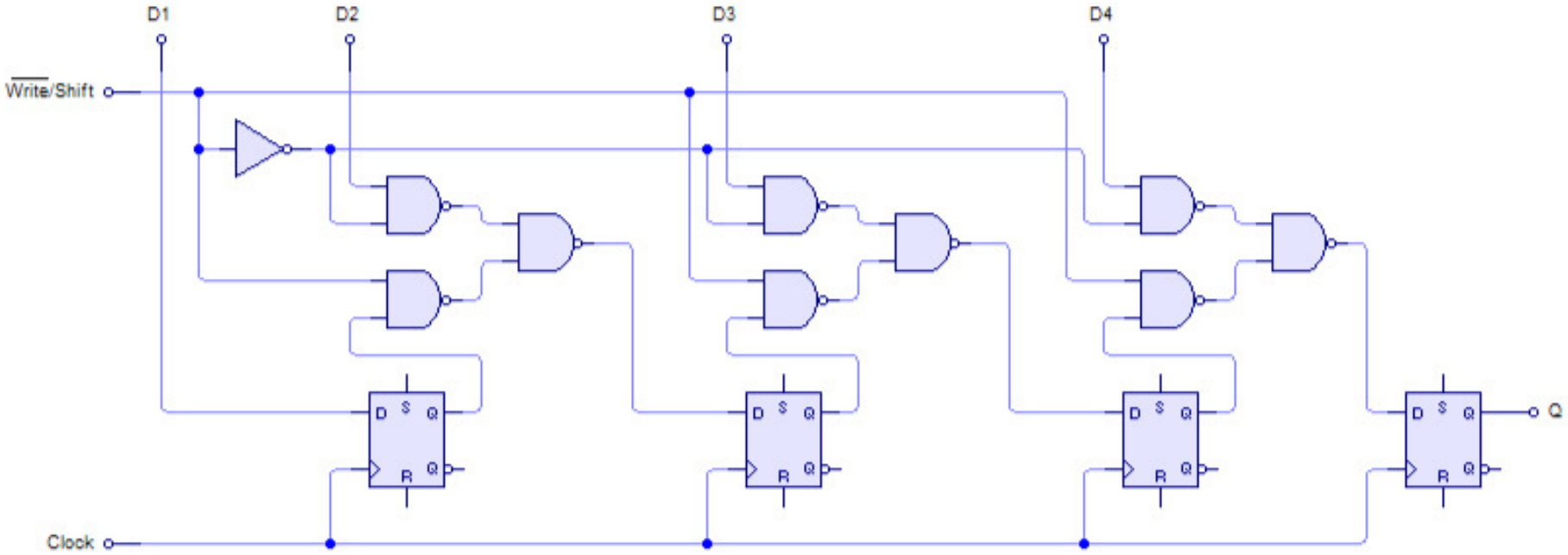
- Speeds Up Arithmetic Operations
- Can Shift Right or Left

Shift Register – Right Shift



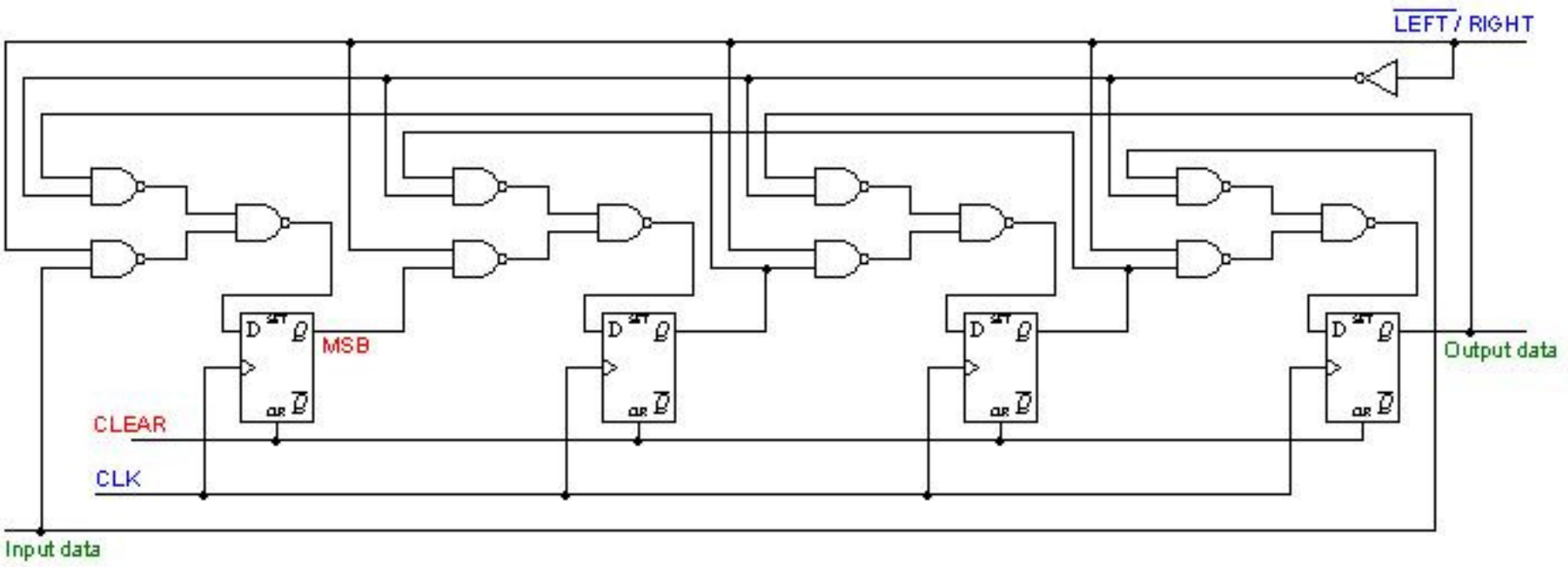
Loadable Shift Register

- Optional Load/Shift



Bidirectional Shift Registers

- Shift Left OR Right



Barrel Shifter