Binary division

- Dividend = divisor × quotient + remainder
- Given dividend and divisor, we want to obtain quotient (Q) and remainder (R)
- We will start from our paper & pencil method

### Hardware design 1

- 64-bit ALU
- 64-bit shift register
- Divisor
- Shift right
- 64 bits
- 64-bit ALU
- Remainder
- Write
- 64 bits
- Control test
- Quotient
- Shift left
- 32 bits

### Hardware design 2

- 32-bit ALU
- 32-bit shift register
- Divisor
- Shift left
- 32 bits
- 64-bit shift register
- Remainder
- Write
- 64 bits
- Control test
Hardware design 3

Example

- Let’s do 0111/0010 (7/2) – unsigned

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Hardware design 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift remainder left by 1</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>remainder = remainder – divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) ⇒ +divisor; shift left; r0=0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>remainder = remainder – divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) ⇒ +divisor; shift left; r0=0</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>remainder = remainder – divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&gt;0) ⇒ shift left; r0=1</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>remainder = remainder – divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&gt;0) ⇒ shift left; r0=1</td>
</tr>
<tr>
<td>done</td>
<td>0010</td>
<td>shift “left half of remainder” right by 1</td>
</tr>
</tbody>
</table>

Exercise sheet

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Hardware design 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Initial values</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>shift remainder left by 1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>done</td>
<td></td>
<td>shift “left half of remainder” right by 1</td>
</tr>
</tbody>
</table>

Restoring division

- The three hardware designs we saw are based on the notion of “restoring division”
  - At first, attempt to subtract divisor from dividend
  - If the result of subtraction is negative – it rolls back by adding divisor
    - This step is called “restoring”
- It’s a “trial-and-error” approach; can we do better?
Non-restoring division

- Let’s revisit the restoring division designs
  - Given remainder R (R<0) after subtraction
  - By adding divisor D back, we have (R+D)
  - After shifting the result, we have $2 \times (R+D) = 2 \times R + 2 \times D$
  - If we subtract the divisor in the next step, we have $2 \times R + 2 \times D - D = 2 \times R + D$

- This is equivalent to
  - Left-shifting R by 1 bit and then adding D!

Example, non-restoring division

- Let’s again do 0111/0010 (7/2) – unsigned

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Hardware design 3, non-restoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values 0000 0111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift remainder left by 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 1110</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) $\Rightarrow$ shift left; r0=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1110 1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>remainder = remainder + divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) $\Rightarrow$ shift left; r0=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111 1100</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>remainder = remainder + divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) $\Rightarrow$ shift left; r0=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011 1000</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) $\Rightarrow$ shift left; r0=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010 0011</td>
</tr>
<tr>
<td>done</td>
<td>0010</td>
<td>shift “left half of remainder” right by 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001 0011</td>
</tr>
</tbody>
</table>

Exercise sheet

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Hardware design 3, non-restoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Initial values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift remainder left by 1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>done</td>
<td></td>
<td>shift “left half of remainder” right by 1</td>
</tr>
</tbody>
</table>

Floating-point (FP) numbers

- Computers need to deal with real numbers
  - Fractional numbers (e.g., 3.1416)
  - Very small numbers (e.g., 0.0000001)
  - Very larger numbers (e.g., $2.7596 \times 10^9$)

- Components in an FP number
  - $(-1)^{sign} \times \text{significand (a.k.a. mantisa)} \times 2^{\text{exponent}}$
  - More bits in significand gives higher accuracy
  - More bits in exponent gives wider range

- A case for FP representation standard
  - Portability issues
  - Improved implementations
  $\Rightarrow$ IEEE-754
### Format choice issues

- Example floating-point numbers (base-10)
  - $1.4 \times 10^{-2}$
  - $-20.0 = -2.00 \times 10^{1}$

- What components do we have?
  - Sign
  - Significand
  - Exponent

- Representing sign is easy.
- Significand is unsigned.
- Exponent is a signed integer. What method do we use?

### IEEE 754

- A standard for representing FP numbers in computers
  - Single precision (32 bits): 8-bit exponent, 23-bit significand
  - Double precision (64 bits): 11-bit exponent, 52-bit significand

#### Biased representation

- Yet another binary number representation
  - Signed number allowed

- 000…000 is the smallest number
- 111…111 is the largest number

- To get the real value, subtract a pre-determined “bias” from the unsigned evaluation of the bit pattern
- In other words, representation = value + bias

#### IEEE 754 example

- $-0.75_{10}$
  - Same as $-3/4$ or $-3/2^{2}$
  - In binary, $-11_{\text{two}}/2^{2}_{\text{ten}}$ or $-0.11_{\text{two}}$
  - In a normalized form, it’s $-1.1_{\text{two}} \times 2^{-1}$

- In IEEE 754
  - Sign bit is 1 – number is negative!
  - Significand is 0.1 – the leading 1 is implicit!
  - Exponent is -1 – or 126 in biased representation
IEEE 754 summary

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Represented Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent</td>
<td>Fraction</td>
<td>Exponent</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>non-zero</td>
<td>0</td>
</tr>
<tr>
<td>1~254</td>
<td>anything</td>
<td>1~2046</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>255</td>
<td>non-zero</td>
<td>2047</td>
</tr>
</tbody>
</table>

**Denormal number**

- Smallest normal: $1.0 \times 2^{\text{Emin}}$
- Below, use denormal: $0.f \times 2^{\text{Emin}}$
- $e = \text{E}_{\text{min}} - 1, f \neq 0$
- #00000000000000000000000000000000

**NaN**

- Not a Number
- Result of illegal computation
  - $0/0$, infinity/infinity, infinity – infinity, ...
  - Any computation involving a NaN
- $e = \text{E}_{\text{max}} + 1, f \neq 0$
- #11111111111111111111111111111111
- Many NaN’s

**Values represented with IEEE 754**

<table>
<thead>
<tr>
<th>Type</th>
<th>Sign</th>
<th>Exponent</th>
<th>Significand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td>00000000</td>
<td>00000000000000000000000000000000</td>
<td>0.0</td>
</tr>
<tr>
<td>One</td>
<td>0</td>
<td>01111111</td>
<td>00000000000000000000000000000000</td>
<td>1.0</td>
</tr>
<tr>
<td>Minus One</td>
<td>0</td>
<td>01111111</td>
<td>00000000000000000000000000000000</td>
<td>-1.0</td>
</tr>
<tr>
<td>Smallest denormalized number</td>
<td>*</td>
<td>00000000</td>
<td>00000000000000000000000000000000</td>
<td>$\pm 2^{-23} \times 2^{-126} = \pm 5.88 \times 10^{-39}$</td>
</tr>
<tr>
<td>“Middle” denormalized number</td>
<td>*</td>
<td>00000000</td>
<td>10000000000000000000000000000000</td>
<td>$\pm 2^{-23} \times 2^{-126} = \pm 5.88 \times 10^{-39}$</td>
</tr>
<tr>
<td>Largest denormalized number</td>
<td>*</td>
<td>00000000</td>
<td>11111111111111111111111111111111</td>
<td>$\pm 2^{-127} = 1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td>Smallest normalized number</td>
<td>*</td>
<td>00000000</td>
<td>00000000000000000000000000000000</td>
<td>$\pm 2^{-127} = 1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td>Largest normalized number</td>
<td>*</td>
<td>11111111</td>
<td>11111111111111111111111111111111</td>
<td>$\pm 2^{-214} \times 2^{1023} = \pm 3.4 \times 10^{38}$</td>
</tr>
<tr>
<td>Positive infinity</td>
<td>0</td>
<td>11111111</td>
<td>00000000000000000000000000000000</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>Negative infinity</td>
<td>1</td>
<td>11111111</td>
<td>00000000000000000000000000000000</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>Not a number</td>
<td>*</td>
<td>11111111</td>
<td>non zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

* Sign bit can be either 0 or 1.
FP arithmetic operations

- We want to support four arithmetic functions (+, −, ×, /)
- (+, −): Must equalize exponents first. Why?
- (×, /): Multiply/divide significand, add/subtract exponents.
- Use “rounding” when result is not accurate
- Exception conditions
  - E.g., Overflow, underflow (what is underflow?)
- Error conditions
  - E.g., divide-by-zero

Overflow and underflow

- Overflow
  - The exponent is too large to fit in the exponent field
- Underflow
  - The exponent is too small to fit in the exponent field

FP addition

1. Align binary points
2. Add significands
3. Normalize result

(Example)

\[
0.5_{ten} - 0.4375_{ten} = 1.000_{two} \times 2^{-1} - 1.110_{two} \times 2^{-2}
\]

FP multiplication

1. Compute exponents
2. Multiply significands
3. Normalize result
4. Set sign

(Example)

\[
(1.000_{two} \times 2^{-1}) \times (-1.110_{two} \times 2^{-2})
\]
**Accuracy and rounding**

- **Goal**
  - IEEE 754 guarantees that the maximum error is ±½ u.l.p. compared with infinite precision
  - u.l.p. = Units in the Last Place = distance to the next floating-point value larger in magnitude

- **Rounding using extra bits**
  - Alignment step in the addition algorithm can cause data to be discarded (shifted out on right)
  - Multiplication step
  - IEEE 754 defines three types of extra bits: G (guard), R (round), S (sticky)

**Guard bit examples**

- Assume 5-bit significand
- Add $1.0000 \times 2^0 + 1.1111 \times 2^{-2}$
- Multiply $1.0000 \times 2^0 \times 1.1001 \times 2^{-2}$

**Rounding modes**

- IEEE 754 has four rounding modes
  - Round to nearest even (default)
  - Round towards plus infinity
  - Round towards minus infinity
  - Round towards 0

- Round bit is calculated to the right of Guard bit
- Sticky bit is used to determine whether there are any 1 bit truncated below Guard and Round bits
- It can be shown that “Round to nearest even” minimizes the mean error introduced by rounding

**Pentium processor divide flaw**

- Pentium FP divider algorithm generates multiple bits per step
  - FPU uses MSBs of divisor and dividend/remainder to guess next 2 bits of quotient
  - Guess is taken from a lookup table: -2, -1, 0, +1, +2
  - Guess is multiplied by divisor and subtracted from remainder to generate a new remainder
  - SRT division (Sweeny, Robertson, and Tocher): Used in most CPUs
- Pentium processor table = 7 bits remainder + 4 bits divisor = 11 bits, $2^{11}$ entries
  - 5 entries of divisors omitted: 1.0001, 1.0100, 1.0111, 1.1010, 1.1101 from the table
  - Fix is just add 5 entries back into the table
  - Eventually, it cost Intel $300M