Lecture 10 Enhancing Performance at gate and register levels
Chapter 4

Contents
- Number systems
  - Fixed point numbers
    - Hardware algorithms for
      - adders,
      - subtractors,
      - multipliers and
      - dividers
    - Hardware for Arithmetic Logic Unit
  - Floating point numbers
    - Hardware algorithms for
      - adders,
      - multipliers
    - Hardware for Arithmetic Logic Unit

Design of ALU
- Functional units for arithmetic operations
  - +,-,*,/
- Functional units for logic operations
  - AND, OR
- Multiplexors to select one out of many inputs

It is important to separate
- Data path
- Control path
Hardware review

Design a circuit that will generate following three outputs from the inputs.

<table>
<thead>
<tr>
<th>3 inputs (ABC)</th>
<th>3 outputs (DEF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>100</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>101</td>
</tr>
</tbody>
</table>
Number Systems

Numbers

+ ve numbers only → memory addresses

unsigned numbers

+ and - numbers necessary for → arithmetic operations

signed numbers

Most significant bit 0 → positive
Most significant bit 1 → negative number

3 bit machine

<table>
<thead>
<tr>
<th>000</th>
<th>→</th>
<th>111 only positive numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

positive and negative numbers

<table>
<thead>
<tr>
<th>0</th>
<th>→</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>→</td>
<td>-3</td>
</tr>
</tbody>
</table>

| 000 | 011 | 111 | 100 |

Computer can be defined as
- adding machine in a limited space

Addition with limited space
Consider 4 bit register

\[
\begin{align*}
\text{ok} & \quad \left\{ \begin{array}{c}
+3 \quad 0011 \\
+2 \quad 0010 \\
\text{ok} \\
-3 \quad 1101 \\
-2 \quad 1110 \\
\text{ok} \\
\end{array} \right. \\
\text{problem} & \quad \left\{ \begin{array}{c}
+5 \quad 0101 \\
+4 \quad 0100 \\
\text{overflow} \\
\end{array} \right. \\
\end{align*}
\]

result is negative number
Example
Consider the following bit pattern. What does this represent?

\[
1000\ 1111\ 1110\ 1111\ 1100\ 0000\ 0000\ 0000 \quad \text{32bits}
\]

Could be a
- 2’s compliment number
  then it will be a negative number

  unsigned
  represents a positive number

Could be an instruction
- opcode  6 bits
- rs  5 bits
- rt  5 bits
- address  16 bits

Could be a floating point number

<table>
<thead>
<tr>
<th>S</th>
<th>exponent</th>
<th>significant</th>
</tr>
</thead>
</table>

S - sign bit + ve or -ve

Example
Convert decimal to binary

\[512_{10} \rightarrow ?\]

answer : q is 01001
- q is 10110 (2’s comp)

Example
Design a one bit ALU with one instruction with two operands.

one instruction
  can be one of the following simple operations
  AND operation,
  OR operation,
  ADD,
  SUB

So the ALU will have simple gates for logic instructions
Example:
Design a one bit ALU with two instructions with two operands
Instruction can be two of the following
    AND
    OR
    ADD

Example:
Design a one bit ALU with many instructions with two operands
    Say the instructions are A, B, C, D ....
Example:
Design a two bit ALU with one instruction with two operands

Example:
64 bit ALU – one instruction with two operands

Example:
Design a 1 bit ALU with 3 instructions. Each instruction consists of two operands.

Add, or, and
Lecture 11 - Performance Improvement of Adders

In design, implementation and manufacturing it is important to consider Cost vs Performance

Cost ↓ = reduce gates ↓
Reduce connections ↓

Performance ↑ = 4 levels \(\Rightarrow\) 2 levels (2 sec)
(AND OR AND OR, 4 sec)

Design Adder, Multiplier, Divide, Floating Point(Adder, Multiplier, Divider) Controllers

Truth Table gives
a basic equation
that will help generate the circuit

Karnaugh map gives a simplified, better circuit

Full adder Design
Full Adder requires adding 3 bits
In addition carry bit propagates to left

\[
\begin{array}{ccc}
C_2 & C_1 & C_0 \\
A_2 & B_1 & B_0 \\
B_2 & B_1 & B_0 \\
\end{array}
\]

--------------

\[
\begin{array}{cccc}
C_3 & S_2 & S_1 & S_0 \\
\end{array}
\]

Adds 3 bits
- 3 inputs
- 2 outputs

Inputs outputs
Sum

Carry

Full adder circuit

Serial adder
- adds two n bit numbers
- one full adder with feedback
- low cost adder
Design of Adders

\[ \begin{array}{c}
\text{c}_1 \\
\text{a}_n, \ldots, \text{a}_1, \text{a}_0 \\
\text{b}_n, \ldots, \text{b}_1, \text{b}_0 \\
\hline
\text{c}_{n+1}, \text{s}_n, \ldots, \text{s}_1, \text{s}_0
\end{array} \]

Example 1:
Example 2:
Ripple Carry Adder
   FA - Full adder - can add 3 bits
   Half adder can add 2 bits

时间 = 2n 秒 for n bit addition

Ripple - carry adder
   Adds two n-bit numbers
   put n full adders together
   carry ripple from FA to FA

n-bit parallel adder
   put #'s in several FAs
   no speed improvement because of the rippling effect

put 4 Full adders together in a chip
Carry lookahead adder

Problem with Ripple carry adder
Sum and carry in the nth FA depends on the previous carries
Must wait long time to generate the nth sum and carry bit

Eliminate the delay
eliminate the ripple effect
With additional hardware it is possible to generate nth carry with no delay.

\[
\begin{align*}
c_1 &= a_0 b_0 + (a_0 + b_0) c_0 \\
c_2 &= a_1 b_1 + (a_1 + b_1) c_1 \\
c_3 &= a_2 b_2 + (a_2 + b_2) c_2 \\
c_{i+1} &= a_i b_i + (a_i + b_i) c_i \\
c_{n+1} &= a_n b_n + (a_n + b_n) c_n
\end{align*}
\]

substitute \( c \) in terms of \( c_0 \)

\[
\begin{align*}
c_2 &= a_1 b_1 + (a_1 + b_1) [a_0 b_0 + (a_0 + b_0) c_0] \\
c_3 &= a_2 b_2 + (a_2 + b_2) [a_1 b_1 + (a_1 + b_1) [a_0 b_0 + (a_0 + b_0) c_0]]
\end{align*}
\]

\[
c_1 = a_0 b_0 + (a_0 + b_0) c_0
\]
\[ c_2 = a_1 b_1 + (a_1 + b_1) a_0 b_0 + (a_1 + b_1)(a_0 + b_0) c_0 \]

\[ c_3 = a_2 b_2 + (a_2 + b_2)a_1 b_1 + (a_2 + b_2)(a_1 + b_1)a_0 b_0 + (a_2 + b_2)(a_1 + b_1)(a_0 + b_0) c_0 \]

\[ c_4 = a_3 b_3 + (a_3 + b_3)a_2 b_2 + (a_3 + b_3)a_1 b_1 + \]
\[ (a_3 + b_3)(a_2 + b_2)(a_1 + b_1)a_0 b_0 + \]
\[ (a_3 + b_3)(a_2 + b_2)(a_1 + b_1)(a_0 + b_0) c_0 \]

3 level circuit

\[ c_4 \text{ can be generated in 3 msec} \]
\[ c_n \text{ also generated in 3 msec} \]

RCA takes 3n sec

CLA 3 sec
Another look at CLA

\[ x_i + y_i = P_i \]
\[ x_i y_i = G_i \]

\[ C_{i+1} = C_{i+1} P_i + G_i \]
\[ C_1 = C_0 P_0 + G_0 \]
\[ C_2 = C_1 P_1 + G_1 \]
\[ = (C_0 P_0 + G_0) P_1 + G_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \]
\[ C_3 = C_2 P_2 + G_2 \]
\[ C_4 = C_3 P_3 + G_3 = a_3 b_3 + (a_3 + b_3) a_2 b_2 + (a_3 + b_3)(a_2 + b_2) a_1 b_1 + \]
\[ (a_3 + b_3)(a_2 + b_2)(a_1 + b_1) a_0 b_0 \]
\[ = (a_3 + b_3)(a_2 + b_2)(a_1 + b_1)(a_0 + b_1) c_0 \]

\[ S_0 = A_0 \oplus B_0 \oplus C_0 \]
\[ S_1 = A_1 \oplus B_1 \oplus C_1 \]
\[ S_2 = A_2 \oplus B_2 \oplus C_2 \]

\[ G_0 = A_0 B_0 \]
\[ P_0 = (A_0 + B_0) \]

\[ G_1 = A_1 B_1 \]
\[ P_1 = (A_1 + B_1) \]

\[ S = X \oplus Y \oplus Z \]
\[ S_0 = P_0 \oplus G_0 \oplus C_0 \]

\[ S_0 = (x_0 + y_0) \oplus x_0 \oplus y_0 \oplus C_0 \]
\[ C_2 = G_1 + P_1 + G_0 + P_1 + P_0 + C_0 \]

\[ x_1 y_1 + (x_1 + y_1)(x_0 y_0) + (x_1 + y_1)(x_0 + y_0) + C_0 \]

\[ S_1 = P_1 \oplus G_1 \oplus C_1 \]

\[ x_1 + y_1 \oplus x_1 y_1 \oplus C_1 \]
4-bit CLA

generates C3  generates C2  generates C1  generates C0

design 4 bit CLA

\[ C_4 = C_3P_3 + G_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \]
RCA + CLA

To construct 8 bit adder
Put two CLAs together
ripping carry from the carry out of 1\textsuperscript{st} CLA to C0 of 2\textsuperscript{nd} CLA

\[ \text{c0 may not always be zero} \]

To construct 12 bit adder
Put three 4-bit CLAs together

To construct 16 bit adder
Put four 4-bit CLAs together
Design of Multipliers
Cost reduction
Increase performance

**Multiplication**

Example
Multiply 21 by 27.

\[
\begin{array}{c}
010101 \\
011011 \\
010101 \\
010101 \\
000000 \\
010101 \\
010101 \\
\hline
01110110001 \\
\end{array}
\]

Hardware requirements
Need
 registers to store
 Multiplicand
 Multiplier
 product
 and an adder

8 bit registers for multiplicand and product
Example
Consider the multiplication of 0010 × 0011
All registers - 8 bit

1\textsuperscript{st} step

2\textsuperscript{nd} Step
3rd Step
2nd version

Reduce cost by reducing the size of the register

4 bits registers for multiplicand and multiplier
product register 8 bits

1st step

4 bit registers
2nd step
4 bits registers

3rd step
4 bits registers
4th step
4 bits registers

5th step
4 bits registers

6th step
4 bits registers
Booth's Algorithm
- 2 bits
- check 2 bits in the multiplier
- Apply following rules
  00 – no arithmetic op
  01 – add multiplicand
  10 – sub multiplicand = add 2’s complement of multiplier
  11 – no arithmetic op

30 = 32-2
Multiplier 32 = 30-2

0011110 = 10000-10

62 = 64-2
Multiplier 62 = 64-2

0111110 = 1000000-10

Thus k consecutive 1's can be replaced by +1 at I+kth position, k-1 consecutive 0's, and -1 at ith position
  I+Kth position 1 means addition of m'cand
  -1 at Ith position means subtraction of M'cand

Example
Multiply 21 by 27 using 2-bit Booth's algorithm

```
010101
011011 0 → Shift 1 to the Right
```

```
- 010101 10 Subtract Multiplier
 000000 11 No
+ 010101 01 Add
- 010101 10 Sub
 000000 11 No
+ 010101 01 Add
```

01000110111
2's complement of m'cand

1111101011  10 Subtract Multiplier = addition of 2's complement

010101
011011 0 → Shift 1 to the Right

1111101011  10 Subtract Multiplier = addition of 2's complement
000000  11 No op
+ 010101  01 Add

1111101011  10 Sub
000000  11 No
+ 010101  01 Add

01000110111
Example:

$$0010 \times 0110$$

need 2’s complement of 0010

```
0010
  \rightarrow
  ADDER
  \rightarrow
  0000 0110
    \rightarrow
    0000 0011
      \rightarrow
      0010
        \rightarrow
        ADDER
        \rightarrow
        1110 0011
          \rightarrow
          1110 0011
            \rightarrow
            0010
              \rightarrow
              ADDER
              \rightarrow
              0000 0110
                \rightarrow
                0000 0011
                  \rightarrow
                  0010
                    \rightarrow
                    ADDER
                    \rightarrow
                    0000 0011
                      \rightarrow
                      0010
                        \rightarrow
                        ADDER
                        \rightarrow
                        0000 0110
                          \rightarrow
                          0000 0011
                            \rightarrow
                            0010
```
Rules for 2 bit Booth Algorithm

00 → 0XA
01 → 1XA
10 → -1XA
11 → 0XA
3-bit Booth
Booth’s algorithm reduces the number of operations by avoiding operations when there were strings of 0s and 1s.
Rules for 3 bit algorithm for multiplication.

<table>
<thead>
<tr>
<th>Current bits</th>
<th>Previous bit</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ai+1 )</td>
<td>( ai )</td>
<td>( ai-1 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example
010101
011011

--------
-010101  110 string - subtract the m'cand
-010101  101 string - subtract the m'cand
0101010  011 string - add 2 * the m'cand

11111101011
111101011
010101
--------
01001110111
Lecture 13 - Performance Improvements at Register Level - Hardware algorithms for fixed point Division

Quotient

\[
\begin{array}{c}
\text{Division} \\
\downarrow \\
\text{Dividend} \\
\text{Remainder}
\end{array}
\]

Example
Divide 7 by 2.
7 ÷ 2

\[
\begin{array}{c}
3 \\
2 \overline{) 7} \\
6 \\
\hline \\
1
\end{array}
\]

result
quotient 3
remainder 1

Example
Binary division
Divide 0111 by 0010

\[
\begin{array}{c}
0111 \div 0010 \\
001 1 \\
10 \overline{) 0111} \\
10 \\
\hline \\
1 1 \\
\hline \\
10 \\
\hline \\
1
\end{array}
\]

Result
Quotient 0011
Remainder 1
Will write down what we did

Subtract divisor from the dividend
Shift

Now Will take two large numbers - dividend 8 bits and divisor 4 bits

Example

Divide 1001010 by 1000

\[
\begin{array}{c}
1001 \\
\hline
1000 \overline{1001010} \\
\underline{-1000} \\
101 \\
1010 \\
\underline{- 1000} \\
10
\end{array}
\]

Result
Quotient 1001
Remainder 10

write down what We did
Subtracted the divisor from the dividend
If the remainder is less than zero
We put quotient bit as zero
Restored the remainder
Take one more bit from the dividend

Subtract the divisor from the dividend
If the remainder is greater than zero
We put quotient bit as one

Take one more bit from the dividend

Keep doing that for all the bits in dividend
Hardware units needed to perform this computation

Need

- a place to store the divisor
- a place to store the dividend
  
  Remainder will remain in the dividend register
- a place to store the quotient
- an adder/subtractor perform subtraction

Specifically we need

- 4 bit register for quotient
- 8 bit register for dividend
- 8 bit register for divisor
  
  left half of the divisor register will have divisor 4bits

an adder/subtractor and control unit

\[
\begin{array}{c}
\text{Divisor} \quad 8 \text{ bit} \\
\text{8 bit} \\
\text{Quotient} \quad 4 \text{ bit} \\
\text{8 bit} \\
\text{Alu} \\
\text{Control unit}
\end{array}
\]

When we multiply 4-bit number by 4-bit number we get an 8-bit product

Similarly, when we divide a 8-bit number by 4-bit number we get a 4-bit number as quotient
Now let us do the division by using this hardware

**Example:**
4-bit divisor

\[ 7 \div 2 \]

\[ 0111 \div 0010 \]

**1st step**

initialize registers

```
0010 0000   divisor \rightarrow shift right
```

```
Alu
```

```
Q
```

```
0000 0111   quotient register bits shifted left \leftarrow
```

Control signals to divisor, quotient and ALU

**Steps to follow**

\[ \text{Rem} = \text{Rem} - \text{Divisor} \]

If Rem < 0

1. Shift Q left, Q₀ = 0
2. Restore Remainder
3. Shift Divisor Right

if Rem > 0

1. Shift Q left, Q₀ = 1
2. Shift Divisor Right
Step 1

Rem = Rem – Divisor (this time divisor is larger so result is negative)
Rem < 0

(1) Shift Q left, Q0 = 0
(2) Restore Remainder
(3) Shift Divisor Right

2nd step

Rem = Rem – Divisor (divisor is larger so result is negative)
Rem < 0

(1) Shift Q left, Q0 = 0
(2) Restore Remainder
(3) Shift Divisor Right
3rd step

Rem = Rem – Divisor (this time divisor is larger so result is negative)
Rem < 0

(1) Shift Q left, Q₀ = 0
(2) Restore Remainder
(3) Shift Divisor Right

4th step

Rem = Rem – Divisor
Now Rem > 0

(1) Shift Q left, Q₀ = 1
(2) Shift Divisor Right
5th step
Rem = Rem – Divisor
Now Rem > 0
(1) Shift Q left, Q₀ = 1
(2) Shift Divisor Right

4-bit divisor needs 5 iterations
8-bit divisor needs 9 iterations
32-bit divisor needs 33 iterations
2nd version

Reduce the hardware costs in the implementation 1
- 8-bit divisor register → 4 bit divisor register
- 8-bit adder → 4 bit adder

Shift dividend to left
No shift in divisor

1st step
Initialize the registers
2nd step
Shift remainder left
Rem = rem-div
Rem < 0
  Restore the remainder
  Shift left Q, Q0 = 0

3rd step
Shift remainder left
Rem = rem-div
Rem < 0
  Restore the remainder
  Shift left Q, Q0 = 0
4th step
Shift remainder left
Rem = rem-div
Rem >=0

Shift left Q, Q0 = 1

5th step
Shift remainder left
Rem = rem-div
Rem >=0

Shift left Q, Q0 = 1

Final snapshot
3rd version of Division algorithm
Eliminate the quotient register
Put the quotient bit to the right most bit position
Shift the dividend register one bit left in each iteration

New hardware organization - Data path and control path

Divisor 32 bit

ALU

Remainder Quotient 64 bit

Initialize

0010

ALU

0000 0111

Shift remainder left

0010

ALU

0000 1110
Step 1
Rem = Rem – Div
Rem < 0
Restore
Shift left Remainder register
R₀ = 0

Step 1

0010

ALU

0001 1100

Step 2
Rem = Rem – Div
Rem > 0
Shift left Remainder register
Q₀ = 1

0010

ALU

0011 1000

new content

0001 1001 of remainder register
Step 3
Rem = Rem – Div
Rem > 0

Shift left Remainder register
Q₀ = 1

Step 4

0011 - quotient
0001 - remainder
Lecture 14 Performance Enhancement with added Functionality - Floating Point Adder

In Computations
We Need
Speed
Accuracy
And range

Integers does not provide the accuracy needed in many computations and the range is limited

The use of Floating point numbers provide the accuracy and range increase

Floating numbers

\[ \pm 3.656914 \times 10^2 \]

Sign  Digit  Fraction  Exponent

We can use the following format to represent the floating point numbers

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
</table>

There are two different formats to increase the precision of computations.
64 bit – Double precision
32 bit – Single precision

To avoid conflicts between different manufacturers IEEE proposed the standard in 1979

32 bit format

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>23</td>
</tr>
</tbody>
</table>

64 bit

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>52</td>
</tr>
</tbody>
</table>
Normalized Form keeps one digit left to the binary point

For example

\[ 1.000111 \]

This can be written as

\[ \pm (1 + F) \times 2^n \]

Fraction - F

Exponent - n

In the IEEE standard we have to add 127 to the exponent in single precision

In the IEEE standard we have to add 1023 to the exponent in double precision

- SP (127)
- DP (1023)

Compromise

Accuracy

Range

Fraction

Exponent

Example

Convert \( .75_{10} \) into binary

\[ .11000 \quad \text{binary} \]

Normalized form

\[ 0.11 = \quad 0 \]

IEEE standard format single precision

\[
\begin{array}{c|c}
0 & \text{-1 + 127} \\
\hline
 & 23 \text{ bit}
\end{array}
\]

Another Example  Convert \( .9_{10} \) to binary
Example
Convert $10_{10}$ into Binary

Conversion

$10_{10} = 1010_2 = 1.010 \times 2^3$

Floating point representation

\[
\begin{array}{c|c}
0 & 0100000000000000 \\
\end{array}
\]

Example:
Represent $10.5_{10}$ in the IEEE format

Convert the whole number and fractional part of the decimal number separately into binary.

\[
10.5_{10} =
\]

\[
\begin{array}{c|c}
0 & 3 + 127 \quad 0101 ... \\
\end{array}
\]

Example:
Represent $0.1_{10}$ in the IEEE format

\[
.1 = 0001 \ 1001 \ 1001 \ 1001 \ 1001 \ 1
= 1.10011 \times 2^4
\]

\[
\begin{array}{c|c}
0 & 10011 ... \\
\end{array}
\]

Now we will look at the FP Addition Hardware needed
Requirements
Let us consider the addition of two floating point numbers
\[ 9.999 \times 10^1 + 1.610 \times 10^{-1} \]

**Step 1**
- compare exponents
- shift the smaller number to right until its exponent match with the larger exponent

**Step 2**
Add significands
\[
\begin{align*}
9.999 \times 10^1 \\
0.016 \times 10^1 \\
\hline
10.015 \times 10^1
\end{align*}
\]

**Step 3**
Normalize
\[ 1.0015 \times 10^2 \]

**Step 4**
Rounding 4 digits
\[ 1.0015 \times 10^2 \rightarrow 1.002 \times 10^2 \]

Will draw the datapath roughly
Floating Point Addition
- compare exponents
  - select larger one
  - shift right the smaller one
- add significands
- normalize
- round

When drawing the datapath
  Draw major functional unit boxes without muxs first
  Then draw Muxs in different color
  Add verticle lines
  Add horizontal control signals for muxs

Compare exponents

MUX1 – selects smaller significand of the two
MUX2 – selects larger significand of the two
MUX3 – first selects the output from ALU and then output from rounding hardware
MUX4 – selects larger exponent
MUX5 – first selects larger exponent and then exponent from rounding hardware
Control path
Inputs
- Exponent difference
- Rounding hardware
- Big ALU

Output Signals from controller—nine outputs
- increment/decrement
- shift left/right
- round
- MUXs

Output signals

Control signals
- selects smaller significand of the two
- selects larger significand of the two
- first selects the output from ALU and then output from rounding hardware
- selects larger exponent
- first selects larger exponent and then exponent from rounding hardware
- Signal for rounding hardware unit
- Normalize shift left or right
- Select the result from ALU or rounding
- Shift right
Lecture 15 Performance Enhancement with added Functionality - Floating Point Multiplier for the ALU

Floating point Multiplication

Example:
\[ 1.10 \times 10^{10} \rightarrow 9.2 \times 10^{-5} \]

Steps
1. Add exponents, subtract 127 for single precision or 1023 for double
2. Multiply significands
3. Normalize the product
4. Rounding
5. Sign

1. New exp
   \[(10 + 127) + (-5 + 127) = 259 \quad \Rightarrow \text{two 127's added in the result}\]
   Subtract bits, \[259 - 127 = 132 = 5 + 127\]

2. \[1.10 \times 9.2 = 10.212000 \times 10^{5}\]
3. \[= 1.0212000 \times 10^{6}\]

   multiply two significands
1. Round 4 digits
2. 1.0212 \rightarrow 1.021
3. sign = +

Example:
\[ .5_{10} \times -.4375_{10}\]

1. \[1.000 \times 2^{-1} \times -1.110 \times 2^{-2}\]
   add exp. \(-3 \Rightarrow 124\)
2. \[1.000 \times -1.110 = 1.11000 \times 2^{-3}\]
4. 1.110

Rounding
- Intermediate results should use two extra bits
  - ground bits
  - round bits

\[
\begin{array}{c}
1.2343 \\
4.2134 \\
\hline
3.4477 \Rightarrow 5.45
\end{array}
\]

With no extra bits for intermediate computation we get less accurate result
\[1.23 + 4.21 = 5.44\]
Problem 4.1
Convert 512_{10} into a 32-bit two’s complement binary number.

Problem 4.2
Convert -1023_{10} into a 32-bit two’s complement binary number.

Problem 4.4
What decimal number does this two’s complement binary number represent:
1111 1111 1111 1111 1111 1110 0000 1100_{two}?

Problem 4.14
The Big Picture on page 299 mentions that bits have no inherent meaning. Given the bit pattern:
1000 1111 1110 1111 1100 0000 0000 0000
what does it represent, assuming that it is
a. a two’s complement integer?
b. an unsigned integer?
8FEFC000
\[ 8 \times 16^7 + 15 \times 16^6 + 14 \times 16^5 + 15 \times 16^4 + 12 \times 16^3 + 0 \times 16^2 + 0 \times 16^1 + 12 \times 16^0 \]
c. A single precision floating-point number?
   SP Floating Number
   S = 1 (negative number)
   Exponent
   0001 1111 = 31 \Rightarrow 31 -127 = -96
   Significand
   .110111111
   -1.110111111 \times 2^{-96}
   -1(1+13*16^{-1} + 15 \times 16^{-2} + 2 \times 16^{-3}) 2^{-96}

   d. A MIPS instruction?
   \begin{align*}
   & \begin{array}{cccc}
   1000 & 1111 & 0111110000000000 \\
   \text{6 bits} & \text{5 bits} & \text{5 bits} & \text{16 bits} \\
   \text{opcode} & \text{rs} & \text{rt} & \text{(negative)}
   \end{array}
   \end{align*}
   
   take 2’s complement
   -16384

   Answer : lw $15 –16384($31)

Problem 4.28
Show the IEEE754 binary representation for the floating-point number –2/3 in single and double precision.

\[-2/3 = -1.01 \times 2^{-1}\]

SP = -1 + 127 = 126
DP = -1 + 1023 = 1022
Problem 4.43
Draw the gates for the Sum bit of an adder, given the equation on page 234.

Sum = abCarryin + abCarryin + abCarryin + abCarryin

Problem 4.44
Rewrite the equation son page 242 for a carry-lookahead logic for a 16-bit adder using a new notation. First use the names for the CarryIn signals of the individual bits of the adder. That is, use $c_4$, $c_8$, $c_{12}$, ... instead of $C_1$, $C_2$, $C_3$, ... Also, let $P_{i,j}$ mean a propagate signal for bits $i$ to $j$, and $G_{i,j}$ mean a generate signal for bits $i$ to $j$. For example, the equation

$$C_2 = G_1 + (P_1.G_0) + (P_1.P_0.c_0)$$

Can be rewritten as

$$C_8 = G_{7,4} + (P_{7,4}.G_{3,0}) + (P_{7,4}.P_{3,0}.c_0)$$

This more general notation is useful in creating wider adders.

$$C_4 = G_{3,0} + P_{3,0}.c_0$$
$$C_8 = G_{7,4} + (P_{7,4}.G_{3,0}) + (P_{7,4}.P_{3,0}.c_0)$$
$$C_{12} = G_{11,8} + P_{11,8}G_{7,4} + P_{11,8}P_{7,4}G_{3,0} + P_{11,8}P_{7,4}G_{3,0}c_0$$
$$C_{16} = G_{15,12} + P_{15,12}G_{11,8} + P_{15,12}P_{11,8}G_{7,4} + P_{15,12}P_{11,8}P_{7,4}G_{3,0} + P_{15,12}P_{11,8}P_{7,4}P_{3,0}.c_0$$

Problem 4.45
Write the equations for the carry-lookahead logic for a 64-bit adder using the new notation from Exercise 4.44 and using 16-bit adders as building blocks. Include a drawing similar to Figure 4.23 in your solution.
Problem 4.49
4-bit numbers (A, B, E, F) There are times when we want to add a collection of numbers together. Suppose you wanted to add four 4-bit numbers (A, B, E, F) using 1-bit full adders. Let’s ignore carry lookahead for now. You would likely connect the 1-bit adders in the organization in the top of Figure 4.56. Below the traditional organization is a novel organization of full adders. Try adding four umbers using both organizations to convince yourself that you get the same answer.

\[
\begin{align*}
S_0 &= P_0 \oplus G_0 + C_0 \\
C_0 &= G_0 + P_0 + C_{in} \\
C_1 &= G_1 + P_1 G_1 + P_0 P_1 C_{in}
\end{align*}
\]
Booth Algorithm
00 no operation
11 no operation
01 +1 \times \text{multiplicand}
10 – 1\times \text{multiplicand}

Problem 4.53
The original reason for Booth’s algorithm was to reduce the number of operations by avoiding operations when there were strings of 0s and 1s. Revise the algorithm on page 260 to look at 3 bits at a time and compute the multiplicand 2 bits at a time. Fill in the following table to determine 2-bit Booth encoding:

<table>
<thead>
<tr>
<th>Current bits</th>
<th>Previous bit</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_i+1)</td>
<td>(a_i)</td>
<td>(a_i-1)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>