ECSE324 : Computer Organization

Arithmetic

Christophe Dubach

Fall 2020

Timestamp: 2020/11/27 15:13:38

Disclaimer

Lectures are recorded live and will be posted unedited on *mycourses* on the same day.

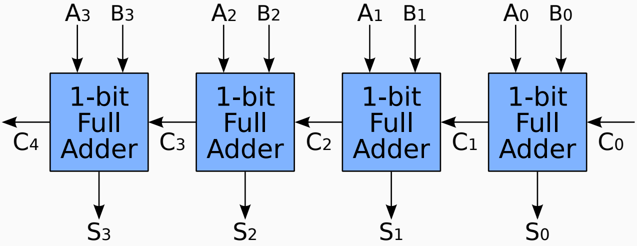
It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the book, the course webpage or ask on Piazza for clarifications.

Adder / Subtracter

Textbook§9.1,9.2

Ripple Carry Adder (recap)

$$S = A + B$$

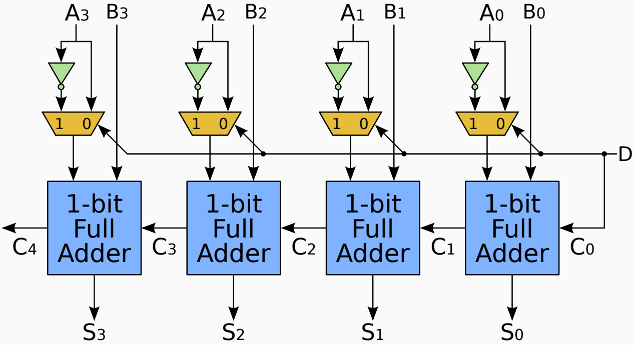


source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg en:User:Cburnett / CC BY-SA

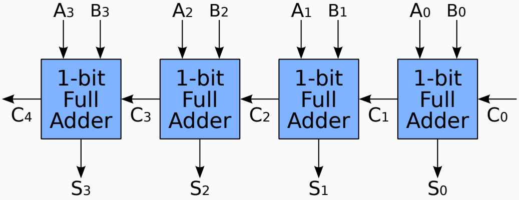
Addition/Subtraction in hardware (recap)

$B - A = B + (-A)$: form 2's complement of A and add to B.

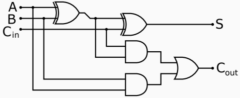
In hardware, inverse the bits and add one using the carry-in. D signal selects between addition and subtraction.



source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder-subtracter.svg en:User:Cburnett / CC BY-SA



Full adder gates:



source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg en:User:Cburnett / CC BY-SA

The MSB bit (S_3) depends on having computed the carry of each preceding bits (at least a delay of two gates between carries).

> The delay for the last carry bit is proportional to the total number of bits involved in the addition!
> In the case of 32bit (or 64bits) integer, this could account for a significant delay!

> Perhaps we could compute C_i in a constant time?
> (independently of the number of input bits)

Carry Lookahead Addition

The addition of two digits generates if it *always* produce a carry

- consider 58 + 71: 5 + 7 generates, but 8 + 1 does not.

Definition

For binary addition, $A_i + B_i$ generates if and only if both A_i and B_i are 1. $G_i = A_i \cdot B_i$ ($= A_i$ AND B_i)

The addition of two digits propagates if it is carried *only when* there is an input carry

- consider 53 + 41: 5+4 propagates, but 3+1 does not.

Definition

For binary addition, $A_i + B_i$ propagates if and only if one of A_i or B_i is 1. $P_i = A_i + B_i$ ($= A_i$ OR B_i)

A_i	B_i	C_i	C_{i+1}	**Carry Type**
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	Propagate
1	0	0	0	
1	0	1	1	Propagate
1	1	0	1	Generate
1	1	1	1	Propagate & Generate

> As can be seen, the addition of two bits produces a carry only when it *generates* or when there is a carry in and it *propagates*.

In Boolean algebra:

$$C_{i+1} = G_i + P_i C_i$$

$$\textit{where}$$

$$G_i = A_i B_i$$

$$P_i = A_i + B_i$$

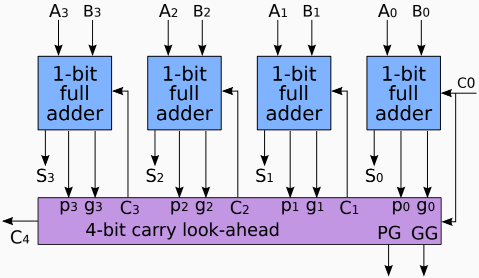
In the case of a four bit adder, we have:

$$
\begin{aligned}
C_1 &= G_0 + C_0 P_0 \\
C_2 &= G_1 + G_0 P_1 + C_0 P_0 P_1 \\
C_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2 \\
C_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3
\end{aligned}
$$

Delay for G_i and P_i is one gate for each (only depends on A_i and B_i).

> 👍 Therefore, the delay for any of the carry bit C_i is only 3 gates. Assuming we can have more than 2 inputs per AND/OR gate, and assuming fan-out is not a problem.

Carry-Lookahead Adder (a.k.a. fast adder)



source: https://en.wikipedia.org/wiki/File:4-bit_carry_lookahead_adder.svgen:User:Cburnett / CC BY-SA

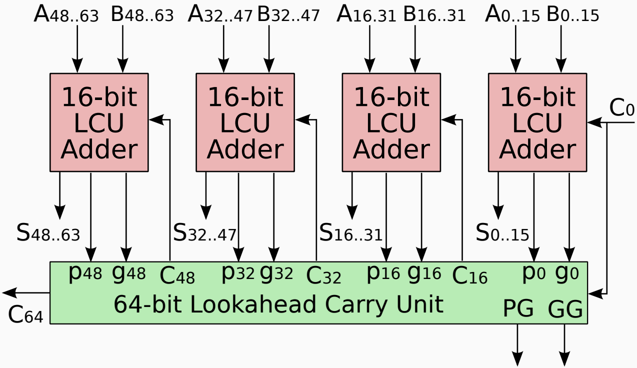
- PG = Propage Group = $P_0 P_1 P_2 P_3$
- GG = Generate Group = $G_3 + G_2 P_3 + G_1 P_3 P_2 + G0 P_3 P_2 P_1$

P_i, G_i only depends on $A_i, B_i \Rightarrow$ delay of computing PG and GG independent on number of bits (assuming again we can have more than 2 inputs per AND/OR, and fan-out not a problem).

Multi-level Carry-Lookahead Adder

carry-lookahead adders can be combined to form larger adders.

e.g. 64-bits adder



source: https://commons.wikimedia.org/wiki/File:64-bit_lookahead_carry_unit.svg en:User:Cburnett / CC BY-SA

Multiplier

Textbook§9.3

General Algorithm

Decimal multiplication

$$\begin{array}{lr} & 11 \\ \times & 13 \\ \hline & 33 \\ + & 11 \\ \hline & 143 \end{array}$$

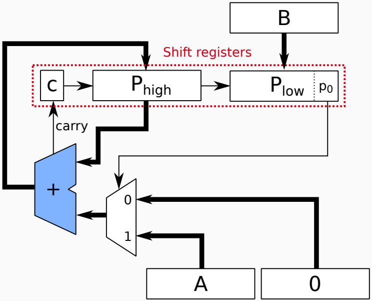
Binary multiplication

$$\begin{array}{lrr} & 1011 & (11) \\ \times & 1101 & (13) \\ \hline & 1011 & \\ & 0000 & \\ & 1011 & \\ + & 1011 & \\ \hline & 10001111 & (143) \end{array}$$

Shift and Add Multiplier (sequential circuit)

Multiplication in binary is equivalent to a series of shifts and additions.

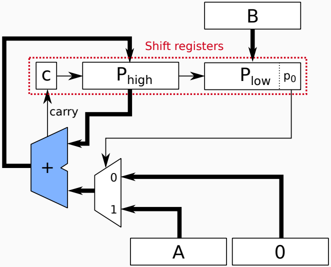
- Multiplicand A, Multiplier B, n bits wide
- Product P = A × B, 2 × n bits wide



Sequential Control Algorithm:

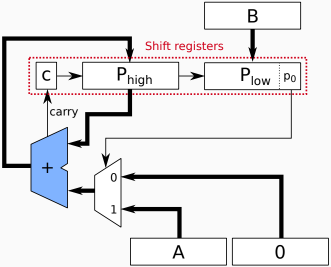
1. Initialize P_{high} with zero and P_{low} with B
2. Update P and C with the result of addition
3. Shift C, P and B right by 1
4. Repeat steps 2–3, $n - 1$ times

- Requires n clock cycles to compute result



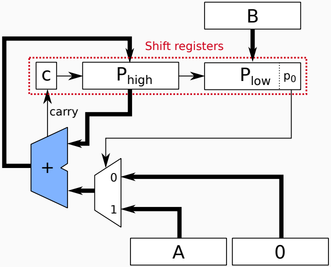
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1



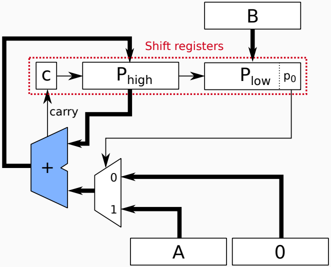
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1



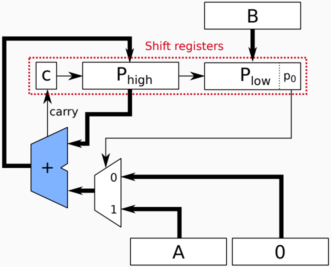
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1
shift	?	0101	1110	0



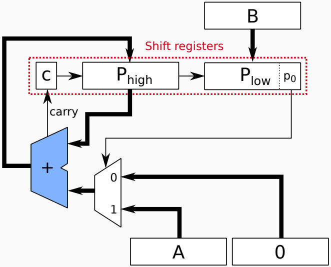
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1
shift	?	0101	1110	0
update	0	0101	1110	0



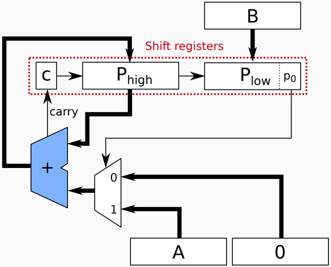
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1
shift	?	0101	1110	0
update	0	0101	1110	0
shift	?	0010	1111	1



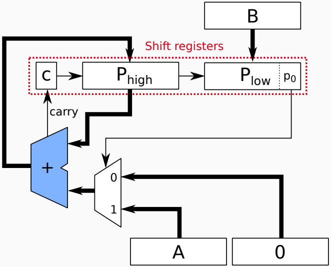
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1
shift	?	0101	1110	0
update	0	0101	1110	0
shift	?	0010	1111	1
update	0	1101	1111	1



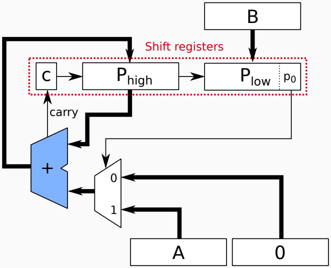
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1
shift	?	0101	1110	0
update	0	0101	1110	0
shift	?	0010	1111	1
update	0	1101	1111	1
shift	?	0110	1111	1



Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1
shift	?	0101	1110	0
update	0	0101	1110	0
shift	?	0010	1111	1
update	0	1101	1111	1
shift	?	0110	1111	1
update	1	0001	1111	1



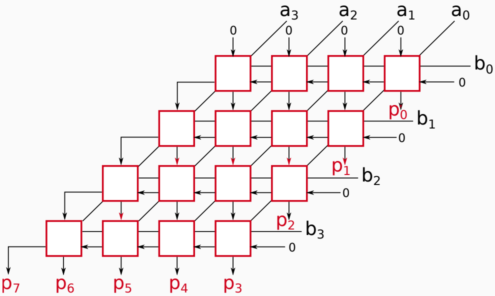
Example: $A = 1011_2 = 11_{10}$, $B = 1101_2 = 13_{10}$

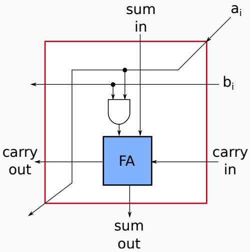
Action	C	P_{high}	P_{low}	p_0
init	?	0000	1101	1
update	0	1011	1101	1
shift	?	0101	1110	0
update	0	0101	1110	0
shift	?	0010	1111	1
update	0	1101	1111	1
shift	?	0110	1111	1
update	1	0001	1111	1
shift	?	1000	1111	1

Array Multiplier (combinational circuit)

> Instead of computing in time, we can compute in space: *systolic array* of cells

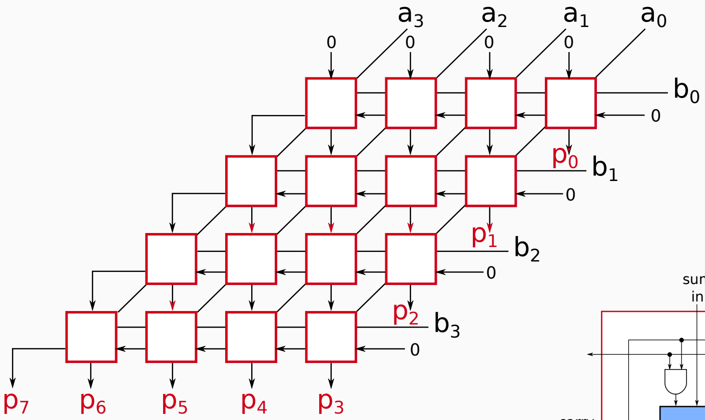
- b_i controls whether a_i is added or not
- partial sum propagates downwards

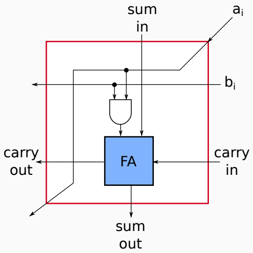




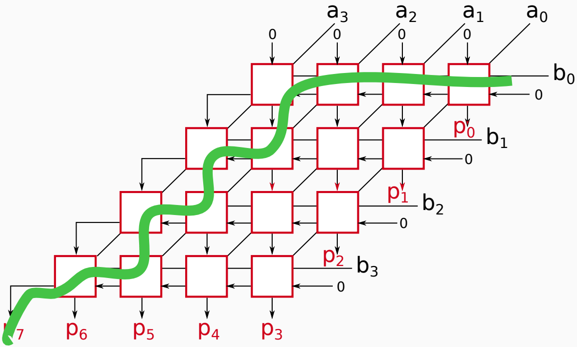
Much faster than shift-and-add multiplier, but at the cost of space

Example: 1011 × 1101





Propagation delay: $n + 2 \cdot (n - 1) = 3 \cdot n - 2$
$\cong 3 \cdot n$



Fast multiplier

Textbook§9.5

Carry save addition

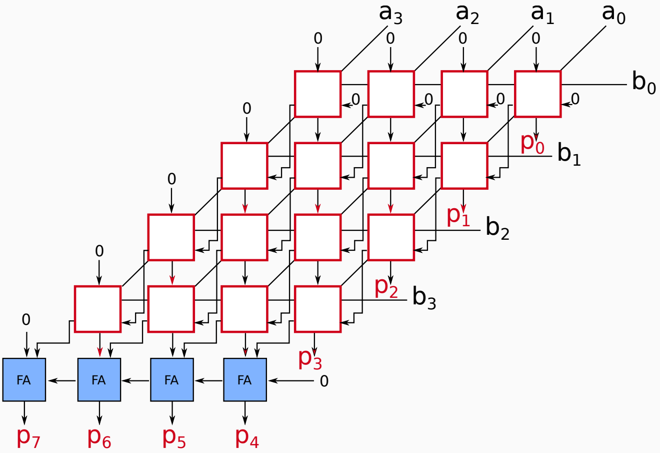
- When summing multiple values, can *save* the carry and add it instead of using a ripple carry adder.
- Full 1-bit adder can add 3 bits!
- Only the last step requires a ripple carry adder.

```
      1011   (11)
×     1101   (13)
--------------------
      1011
     0000
    1011
+  1011
--------------------
  10001111  (143)
```

```
      1011
+    0000       carry save add
--------------------------------
     01011
+    000                 carry

     01011
+   1011        carry save add
--------------------------------
    100111
+   010                  carry

    100111
   1011
+   010         carry save add
--------------------------------
   1101111
+  010                   carry

   1101111
+  010       ripple carry add
--------------------------------
   1
  10001111
```

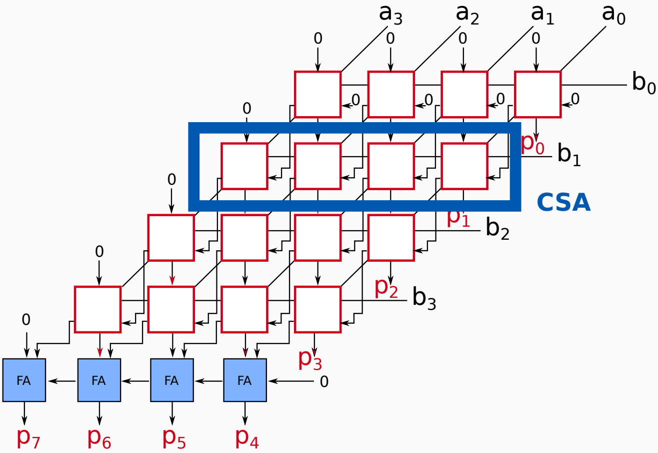
Multiplication with Carry Save Adder

- Instead of propagating the carry to the left, propagate it downwards.
- Last stage performs the carry propagation to the left.

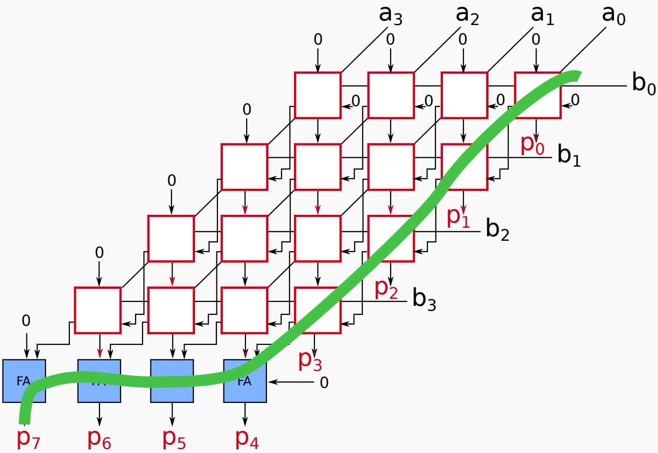


In a given row, each cell is independent from its neighbour:

- There is no more any carry to propagate within the row.
- An entire row is computed in parallel!



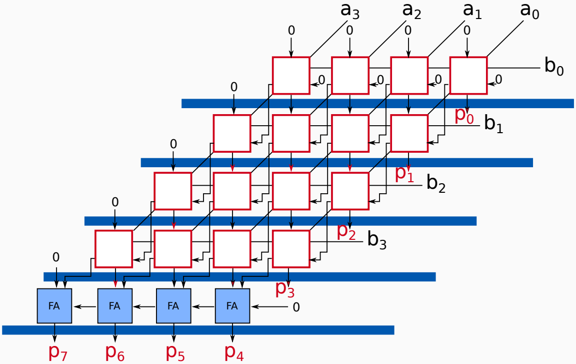
Propagation delay: $n + n$
$= 2 \cdot n$



Pipelined Array Multiplier

Can pipeline the array to increase throughput

- Insert registered between each row
- Each clock cycle, new data is fed into the pipeline



Addition Tree

> Break down the sum into separate parts that are solve independently.

A+B+C+D+E+F

$$\begin{array}{lrr}
 & 101011 & (43)\\
\times & 001101 & (13)\\
\hline
 & 101011 & A\\
 & 000000 & B\\
 & 101011 & C\\
 & 101011 & D\\
 & 000000 & E\\
+ & 000000 & F\\
\hline
 & 001000101111 & (559)
\end{array}$$

A+B+C

$$\begin{array}{lrr}
 & 101011 & A\\
 & 000000 & B\\
+ & 101011 & C\\
\hline
 & 10000111 & S_0\\
 & 001010000 & C_0
\end{array}$$

D+E+F

$$\begin{array}{lrr}
 & 101011 & D\\
 & 000000 & E\\
+ & 000000 & F\\
\hline
 & 00101011 & S_1\\
 & 00000000 & C_1
\end{array}$$

$S_0 + C_0 + S_1$

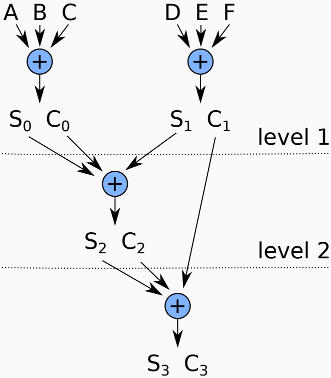
$$\begin{array}{rrl} & 10000111 & S_0 \\ & 001010000 & C_0 \\ + & 00101011 & S_1 \\ \hline & 00110001111 & S_2 \\ & 00010100000 & C_2 \end{array}$$

$C_1 + S_2 + C_2$

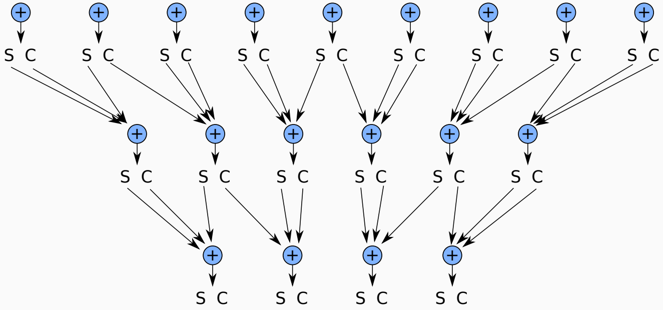
$$\begin{array}{rrl} & 00000000 & C_1 \\ & 00110001111 & S_2 \\ + & 00010100000 & C_2 \\ \hline & 00100101111 & S_3 \\ & 00100000000 & C_3 \end{array}$$

full carry-ripple addition of S3+C3:

$$\begin{array}{rrl} & 00100101111 & S_3 \\ + & 00100000000 & C_3 \\ \hline & 01000101111 & 559 \end{array}$$



Addition tree with 3-2 Reducer



At each level, reduce by a factor $3/2 = 1.5$.

Complexity of multiplication of n bits number is now:
$\cong log_{1.5}(n) \cong 1.7 log_2(n)$ for inputs of n bits.

Other techniques

Several other complementary techniques (no discussed in this class) exist for designing fast adders:

- Bit-pair recoding / Booth algorithm
- Wallace/Dadda Tree multiplier

Multiplication of Signed Numbers

Textbook§9.4

Signed numbers multiplication

- If the multiplicand is negative, we could sign extend during additions
  - Example: $-11 \times 13 = 10101 \times 01101$ in two's complement (5 bits)

$$
\begin{array}{cll}
 & 10101 & \quad(-11) \\
\times & 01101 & \quad(13) \\
\hline
 & 1111110101 & \\
 & 000000000 & \\
 & 11110101 & \\
 & 1110101 & \\
+ & 000000 & \\
\hline
 & 1101110001 & (-143)
\end{array}
$$

- Alternatively, if the either the multiplier or the multiplicand is negative, negate it and negate the result.
- If both multiplier/multiplicand are negative, negate both numbers and proceed as usual.

Floating Point Numbers and Operations

Textbook§9.7

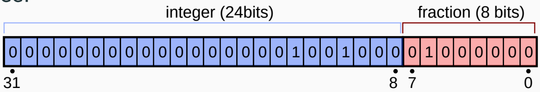
Fixed Point Representation

Consider these two examples of numbers:

- Integer: $72 = 2^3 + 2^6 = 1001000_2$
- Real: $72.25 = ???_2$ *How to represent this number in binary?*

Fixed point representation:

- Reserver a fixed number of bits for the integer part and the remaining for the fractional part.
- For instance:



source: Modified by Christophe Dubach. Original from Vectorization: Stannered, CC BY-SA 3.0 https://commons.wikimedia.org/wiki/File:Float_example.svg

- $...001001000.01000000_2 = 2^3 + 2^6 + 2^{-2} = 72.25_{10}$

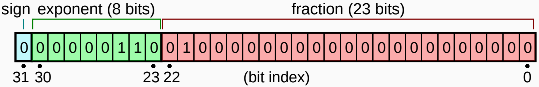
Problems:

- amount of precision after the point is *fixed*
- cannot represent extremelly large or extremelly small numbers

Floating Pointer Representation

Represented with a sign bit s, an unsigned exponent exp, and an unsigned mantissa fraction m.

- For instance:



source: Modified by Christophe Dubach. Original by Vectorization: Stannered, CC BY-SA 3.0 https://commons.wikimedia.org/wiki/File:Float_example.svg

- $0.01000\ldots_2 \times 2^{00000110_2} = (2^{-2}) * (2^6) = 0.25 * 64 = 16_{10}$
- $0.11100\ldots_2 \times 2^{00000010_2} = (2^{-1} + 2^{-2} + 2^{-3}) * (2^2) = 0.875 * 4 = 3.5_{10}$

The value represented is $\boxed{-1^s \times m \times 2^e}$.

Note that:

- the mantissa m is expressed as a "fraction"
- the exponent can be represented in 2's complement (signed) or using biased notation (explained later)

Normalized representation

Consider these following encodings for the same real number:

Base 2	**S**	**Exponent**	**Mantissa fraction**
0.0011×2^0	0	0000000	001100...
0.011×2^{-1}	0	1111111	011000...
0.11×2^{-2}	0	1111110	110000...

All equivalent, this encoding is wasteful!

What if we encode all our numbers as on the last row? In this case, the first bit of the mantissa is always one.

> We could omit it and shift left all other bits
> ⇒ One extra bit available in the mantissa!

Using *normalized* representation:

1.1×2^{-3}	0	1111101	1000000...

Now the value represented is $\boxed{-1^s \times (1.m) \times 2^e}$.

Biased Exponent

Instead of using a signed number for the exponent, sometimes it might be useful to use an unsigned value.

- This might simplify the hardware when comparing floating points value (which involves comparing the exponents).
- Since the exponent must be able to represent both positive and negative numbers, the trick consits of subtracting a bias from the exponent

The value represented becomes $\boxed{-1^s \times (1.m) \times 2^{e-bias}}$, where e is an unsigned (postive) number.

The bias is typically chosen in the middle of the valid range for e. *e.g.* if e is an 8-bit value, the bias would be $2^8/2 = 2^7 = 128$

IEEE 754 Floating-Point Representation

Most modern machines uses the IEEE 754 floating point standard to represent single precision (32-bit) real numbers:

- 1 sign bit s
- 8-bit exponent e with a bias of 127 ($\neq 128$)
- 23 bits mantissa m (normalized)

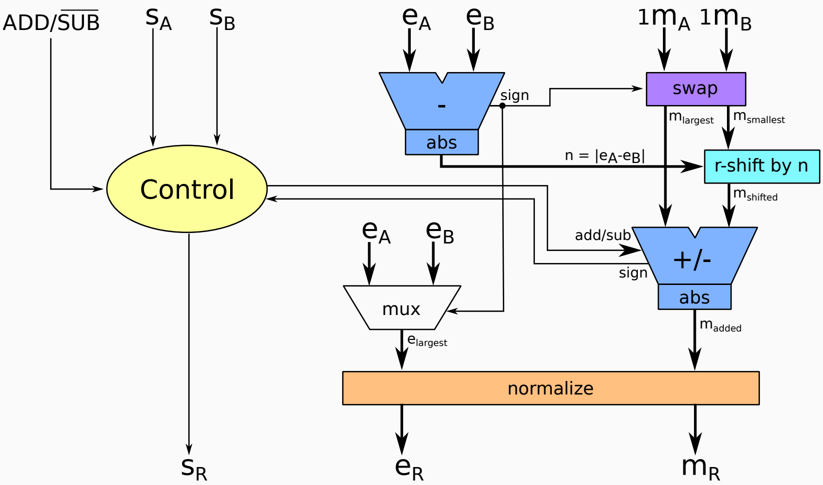
The value represented is: $\boxed{-1^s \times (1.m) \times 2^{e-127}}$

Note that:

- $e = 1111111$ is reserved to represents not-a-number (NaN), infinite or special case (*e.g.* division by zero).
- $e = 0000000$ is used un-normalized numbers, *i.e.* number smaller than 2^{-126} and zero

For double-precision (64-bit), exponent is 11 bits and mantissa 52 bits.

Add/Sub Floating Point Unit (simplified)



Steps for computation:

1. Subtract the two exponents, $n = |e_A - e_B|$;
2. If sign negative, swap the two mantissas so that the mantissa corresponding to the smallest exponent is the input to the shifter;
3. Shifts right by n bits $m_{smallest}$;
4. Add/subtract the $m_{largest}$ with $m_{shifted}$ and take absolute value. ALU operation $= f(s_a, s_b, ADD/\overline{SUB})$;
5. Using the largest exponent $e_{largest}$ and the result of the ALU m_{added}, normalize the exponent and mantissa.
6. s_R depends on signs s_A and s_B and the resulting sign when computing m_{added}.

Example

$2.25 - 12.75 = -10.5$

- $A = 2.25_{10} = 10.01_2 = 1.001_2 \times 2^1$
- $B = 12.75_{10} = 1100.11_2 = 1.10011_2 \times 2^3$

	s	e	m
A	0	$10000000(127+1)$	$001000\cdots 0$
B	0	$10000010(127+3)$	$100110\cdots 0$
R	1	$10000010(127+3)$	$010100\cdots 0$

Step-by-step:

1. $n = 2$, sign $=$ positive
2. $m_{smallest} = m_A = 1.001000\cdots 0$, $m_{largest} = m_B = 1.100110\cdots 0$
3. $m_{shifted} = m_{smallest} >> 2 = 1.001000\cdots 0 >> 2 = 0.010010\cdots 0$
4. $m_{added} = |m_{largest} - m_{shifted}| = 1.01010\cdots 0$
5. mantissa is already normalized

   $\Rightarrow R_m = 010100\cdots 0,\ e = 3 + 127 = 130$
6. $s_R = 1$

result $= -1.0101_2 \times 2^3 = -1010.1_2 = -10.5_{10}$

Not getting it? Let's see what's going on in details

2.25 - 12.75

$2.25_{10} = 10.01_2 = 1.001_2 \times 2^1$

$12.75_{10} = 1100.11_2 = 1.10011_2 \times 2^3$

$$
\begin{array}{crlr}
 & 1.00100 & \times 2^1 & \text{A (smallest)} \\
- & 1.10011 & \times 2^3 & \text{B (largest)} \\
\hline
 & 0.01001 & \times 2^3 & \text{A shifted} \\
- & 1.10011 & \times 2^2 & \\
\hline
 & 00.01001 & \times 2^3 & \text{A with sign bit added} \\
+ & 10.01101 & \times 2^3 & \text{B two's complement} \\
\hline
 & 10.10110 & \times 2^3 &
\end{array}
$$

$|10.10110| \times 2^3 = 01.01010 \times 2^3 = 1010.10_2 = 10.5_{10}$

Add the sign $\Rightarrow -10.5_{10}$

Floating Point Multiplication

Easier than addition. Steps:

1. Sum the exponent (and subtract the bias, otherwise added twice)
2. Multiply the mantissas (using the implicit 1)
3. Normalize

Example: $A = 2.5$, $B = 0.75$

$2.5_{10} \times 0.75_{10} = 10.1_2 \times 0.11_2 = 1.01 \times 2^1 \times 1.1 \times 2^{-1}$

1. Sum exponent: $1 - 1 = 0$
2. Multiply mantissas: $1.01 \times 1.1 = 1.01 + 0.101 = 1.111$
3. Already normalized $\Rightarrow m_R = 1110 \cdots 0$, $e_R = 127 + 0 = 0111111$

$R = 1.111 \times 2^0 = 1.111 = 1.875_{10}$

Other considerations

Although this will not be discussed in this class, there are other details about floating point representation/operations:

- Special value encoding:

Special Value	**exponent**	**mantissa**
$\pm\infty$	255	0
$\pm$NaN (Not a Number)	255	$\neq 0$
± 0	0	0
Denormal numbers ($\pm 0.m \times 2^{-126}$)	0	$\neq 0$

- Exceptions: *e.g.* division by 0, squared root of -1 (result in NaN)
- Rounding/Truncating: sometimes we may need to reduce number of bits for the mantissa
  - We may need to do more than simply dropping a bit
  - *e.g.* in base 10, going from 4 digits to 3:
    $0.2222_{10} \cong 0.222_{10}$ whereas $0.7777_{10} \cong 0.778_{10}$

Conclusion

This lecture has:

- introduced how fast adders can be implemented hardware
- explained how multiplier can be implemented in hardware
- introduced floating point representation
- shown how floating point addition and multiplication work

The End