Signed Numbers and ALUs

Dr Geoff Merrett
ELEC1202: Digital Systems and Microprocessors
These Lectures

Previously

- Unsigned Number Representation
  - Decimal
  - Binary
  - Hexadecimal
  - BCD
  - Converting to and from most of these different number systems

- Unsigned Addition
  - Decimal, hexadecimal, binary

These lectures

- Signed Number Systems
  - Sign-Mantissa
  - Negative Excess
  - One’s Complement
  - Two’s Complement
  - Adding Signed Numbers
  - Subtracting Signed Numbers

- Computation in Hardware
  - The Half Adder
  - The Full Adder
  - The Multi-Bit Adder
  - 2’s Complement Multi-Bit Adder
  - The ALU
Number Systems

Signed Numbers
Basic Terminology

- Positive numbers
- Negative numbers
- Integer
- Fractional

'Large' vs. 'Small' on the number line.
Binary Numbers

- To convert from Binary to Decimal:

```
0 1 0 1 1 0 1 0

= 64 + 16 + 8 + 2

= 90_{10}
```

\[ = \sum d_i \cdot 2^i \]
Signed Number Systems

• How do represent negative numbers in binary?

• In decimal, we add a ‘-’ sign to the left hand side of the number
  – e.g. ‘-56’

• In binary, we could add a ‘-’ sign to the left hand side of the number
  – e.g. ‘-1011010’
  – How do you represent a minus sign as a bit?

• This is our first number representation...
Sign Mantissa

- The Most Significant Bit (MSB) is used to indicate the sign of the number
- A ‘1’ indicates a negative number, ‘0’ a positive number

\[
\begin{align*}
\text{10001101}_2 &= -1 \times (8 + 4 + 1) \\
&= -13
\end{align*}
\]
• How is zero represented?
  − There are two zeros = +0₁₀ (00000000₂) and -0₁₀ (10000000₂)

• What is the range?
  − -127₁₀ (11111111₂) -> +127₁₀ (01111111₂) : -(2^{n-1} - 1) -> +(2^{n-1} - 1)
• **Sign-Mantissa**

  - How would we add two numbers represented using sign-mantissa?
    - If signs are the same, add the numbers and give the result the same sign
    - If signs are different
      - Compare magnitudes and subtract smaller from larger
      - Result has the sign of the larger
Negative Excess

*a.k.a excess-K or offset binary*

- To convert a signed number to excess form, add $2^{n-1}$ (the ‘excess’) to it.
- E.g. What is -13 in 8-bit Negative Excess?

$$2^{n-1} = 2^7 = 128$$

$$-13 + 128 = 115$$

$$011110011_2 = 115$$

- To convert back (i.e. Excess form to a signed value), subtract $2^{n-1}$ from it.
- E.g. What is 01110011_2 (Excess Representation) in decimal?

$$011110011_2 = 115$$

$$115 - 128 = -13$$

$$=-13$$
Negative Excess

- How is Zero represented?
  - There is only one zero = 100000002

- What is the range?
  - -128_{10} (000000002) -> +127_{10} (111111112) : -(2^{n-1}) -> +(2^{n-1} - 1)
Negative Excess

Addition is awkward due to the ‘fake offset’
Adding Signed Numbers

• ...using sign-mantissa?
  – If signs are the same, add the numbers and give the result the same sign
  – If signs are different
    • Compare magnitudes and subtract smaller from larger
    • Result has the sign of the larger

• ...using negative excess?
  – Awkward. Always working with an offset, or ‘fake’, zero...
  – And addition will add the offset twice.

• ...using complement number systems
  – Instead of using a sign bit, we take the complement of the number
  – Can add and subtract numbers without sign and magnitude checks
1’s Complement

- This is the ‘Diminished Radix-Complement’ for Binary (Base or Radix 2)
- In 1’s Complement, the complement of an $n$ bit number is obtained by subtracting it from $2^n-1$.
- E.g. What is -13 in 8-bit 1’s Complement?

\[
\begin{align*}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1_2 \\
= & +13 \\
2^8 - 1 & = 255 \\
255 - 13 & = 242
\end{align*}
\]

\[
\begin{align*}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0_2 \\
= & -13
\end{align*}
\]

- We have actually just complemented all of the bits!

“Flip the bits”
1’s Complement

- How do we interpret a 1’s Complement Number?
- If the Sign Bit (MSB) is 0, the binary number in the remaining bits is positive
- If the Sign Bit (MSB) is 1, the binary number is negative...
  - Complement/Invert/Flip all of the bits
  - This gives the equivalent ‘positive’ value of the number

\[
\begin{array}{ccccccccc}
& & & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 2 \\
& & & \text{MSB} & & & & \text{LSB} & & & \\
\text{sign} & & 1 & & 1 & & 1 & & 0 & & 0 & \\
\end{array}
\]

- \( \text{Sign bit is a ‘1’, therefore a negative number...} \)
  Complement the bits \( \rightarrow 00001101_2 = 13_{10} \)
  Therefore, \( 1110010_2 \) in 1’s Complement = \(-13_{10}\)
1’s Complement

- How is Zero represented?
  - There are two zeros = \(+0_{10}\) (00000000₂) and \(-0_{10}\) (1111111₁₂)

- What is the range?
  - \(-127_{10}\) (10000000₂) \(\rightarrow\) \(+127_{10}\) (0111111₁₂) : \(-\left(2^{n-1} - 1\right)\) \(\rightarrow\) \(+\left(2^{n-1} - 1\right)\)
1’s Complement

An alternative way of considering 1’s Complement numbers...

<table>
<thead>
<tr>
<th>Most Significant Bit</th>
<th>Least Significant Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 0 1</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td>1 0 1 0 0 0 0 1 0</td>
<td>2</td>
<td>-93</td>
</tr>
</tbody>
</table>

Value interpretation:
- Positive numbers are represented as their binary form.
- Negative numbers are represented in 1’s Complement form.
1’s Compliment

- Sign-Mantissa

- One’s Complement

- Negation is easy! (simply take the complement)
- Addition/subtraction hardware complex; MSB has a weight of -127 (not power of 2)
- Two values of zero must still be detected
2’s Complement

- This is the ‘Radix-Complement’ for Binary (Base or Radix 2)
- In 2’s Complement, the complement of an \( n \) bit number is obtained by subtracting it from \( 2^n \).
- E.g. What is -13 in 8-bit 1’s Complement?

\[
\begin{align*}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1_2 \\
& = & +13 \\
2^8 &= 256 \\
256 - 13 &= 243 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1_2 \\
& = & -13 \\
\end{align*}
\]

- We have complemented all of the bits and added one!

“Flip the bits and add one”
2’s Complement

- How do we interpret a 2’s Complement Number?
- If the Sign Bit (MSB) is 0, the binary number in the remaining bits is positive
- If the Sign Bit (MSB) is 1, the binary number is negative...
  - Complement/Invert/Flip all of the bits
  - Add 1
  - This gives the equivalent ‘positive’ value of the number

-> Sign bit is a ‘1’, therefore a negative number...

Complement the bits – 00001101₂ = 13₁₀

Add one – 00001110₂ = 14₁₀

Therefore, 1110010₂ in 2’s Complement = -14₁₀
2’s Complement

- How is Zero represented?
  - There is only one zero = 00000000₂

- What is the range?
  - -128₁₀ (10000000₂) -> +127₁₀ (01111111₂) : -(2ⁿ⁻¹) -> +(2ⁿ⁻¹ - 1)
2’s Complement

- An alternative way of considering 2’s Complement numbers...

-128
- 64  32  16  8  4  2  1

0 1 0 0 0 0 0 1 = 65

most significant bit

least significant bit

1 0 1 0 0 0 0 1 0 = -94

most significant bit

least significant bit
2’s Complement

- Negative Excess

- One’s Complement

- Two’s Complement

Interesting to note that the Two’s Complement number line is identical to that of Negative Excess, except that the MSB is complemented...
2’s Complement

• Zero has only one representation.

• The sign bit has a weight of \(-2^{n-1}\) – a power of 2! (128 in our examples)
  – Therefore addition/subtraction hardware is simple
  – For this reason, 2’s complement representation has been universally adopted in virtually all processor architectures.
Signed Numbers: A Summary

• Sign-Mantissa Representation
  – To negate a number, change the sign bit (the MSB)
  – If the MSB is 0, the number is +ve. If the MSB is 1, the number is –ve.
  – The magnitude of the number is given by the least significant \( n-1 \) bits.

= \((32+8+2) \times -1\)
\[\text{= -42}\]

• Negative Excess Representation
  – A number line from 00000000\(_2\) (= -128\(_{10}\)) to 11111111\(_2\) (= +127\(_{10}\))
  – If the MSB is 1, the number is +ve. If the MSB is 0, the number is –ve.
  – A number in excess representation is the value + \(2^{n-1}\) (E = V + \(2^{n-1}\))

\[\text{= 170-128}\]
\[\text{= 42}\]

• 1’s Complement Representation
  – If the MSB is 0, the number is +ve. If the MSB is 1, the number is –ve.
  – To negate a number, flip the bits (take the complement)

\[\text{=(64+16+4+1) \times -1}\]
\[\text{or... \ -127+32+8+2}\]
\[\text{= -85}\]

• 2’s Complement Representation
  – If the MSB is 0, the number is +ve. If the MSB is 1, the number is –ve.
  – To negate a number, flip the bits (take the complement) and add 1.

\[\text{=((64+16+4+1)+1) \times -1}\]
\[\text{or... \ -128+32+8+2}\]
\[\text{= -86}\]

\[\text{1 0 1 0 1 0 1 0}_2\]
Recap Exercise

1. Convert 00110100_2 to decimal.

2. Convert, using successive division, \(76_{10}\) to:
   a) unsigned 8-bit binary,
   b) unsigned hexadecimal.

3. Express the numbers \(+57_{10}\) and \(-57_{10}\) in signed 8-bit binary using:
   a) sign-mantissa,
   b) 1’s complement,
   c) 2’s complement,
   d) negative excess.
But how do we actually **add** or **subtract** signed numbers?
2’s Complement Arithmetic

- Addition in 2’s complement is the same as for unsigned arithmetic
2’s Complement Arithmetic

- To subtract B from A, we can add A to \(-B\)
  \[ A - B = A + (\neg B) = A + (\bar{B} + 1) \]

- E.g. Using 2’s complement 8-bit binary, subtract \(40_{10}\) from \(29_{10}\) \((29_{10} - 40_{10})\)

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
- & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\hline
=+29_{10} & =+40_{10}
\end{array}
\]

To verify...

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
+ & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0
\end{array}
\]

(flip the bits)

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
=+11_{10}
\end{array}
\]

(add 1)

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
\end{array}
\]

(flip the bits)

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\hline
\end{array}
\]

(add 1)

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
\end{array}
\]

Sum Bit

Carry Bit
2’s Complement Arithmetic

- Previous example resulted in a negative number (i.e. \( A - B < 0 \) as \( B > A \))
- What happens if \( A - B > 0 \), i.e. \( B > A \)? E.g. \( 27_{10} - 11_{10} \):

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
- & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\hline
& 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
+ & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

(Add One)

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

But what about this carry bit?

\( = +27_{10} \)

\( = +11_{10} \)

\( (flip \ the \ bits) \)

\( (add \ one) \)

\( = +16_{10} \)
Unsigned MSB Carry-Out (Overflow)

- How do we detect overflow?
- For example, add $186_{10} + 99_{10}$:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

- $186_{10} + 99_{10} = 285_{10} = 10001101_2$ (i.e. doesn’t fit in 8 bits)
- If the Most Significant Carry Bit = 1, overflow has occurred.
2’s Complement MSB Carry-Out (OK)

- In the first example \((29_{10} - 40_{10} = -11_{10})\), the 2’s complement arithmetic did not actually ‘cross’ zero by returning a negative result:

- In the second example, \((27_{10} - 11_{10} = 16_{10})\), the 2’s complement arithmetic ‘crossed’ zero by returning a positive result:

- The carry bit is an artefact of the complementing process, and can be ignored.
Computation in Hardware
(Basic) Computer Architecture

Source: https://secure.ecs.soton.ac.uk/notes/ellabs/databook/avr/ATmega48_full.pdf
Arithmetic Logic Unit

- What is its purpose?
- What are its inputs?
- What are its outputs?
Unsigned Binary Arithmetic
Binary Addition

- Multi-bit Addition
- For example, add $27_{10} + 11_{10}$:

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
+ & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

- $27_{10} + 11_{10} = 38_{10} \ (00100110_2)$
Unsigned Binary Addition

- 1-digit Decimal Addition

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Carry Bit</th>
<th>Sum Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

- 1-bit Binary Addition

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Carry Bit</th>
<th>Sum Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The Half Adder

- The Half Adder adds two bits A and B, producing a Sum and Carry output.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_{out}</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ Sum = \bar{A}B + AB \]
\[ = A \oplus B \]

\[ C_{out} = AB \]
module halfadder (input logic A, B, output logic Sum, Cout);
    assign Sum = A ^ B;
    assign Cout = A & B;
endmodule
Unsigned Binary Addition

- Multi-bit Addition
- For example, add $27_{10} + 11_{10}$:

```
  0 0 0 0 1 1 0 1 1
  + 0 0 0 0 0 1 0 1 1
  = 0 0 1 0 0 1 1 0 0
```

- $27_{10} + 11_{10} = 38_{10} (00100110_2)$
## The Full Adder

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Carry Bit</th>
<th>Sum Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
# The Full Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C&lt;sub&gt;in&lt;/sub&gt;</th>
<th>C&lt;sub&gt;out&lt;/sub&gt;</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Sum**

\[
Sum = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + ABC_{in} + A\bar{B}\bar{C}_{in} \\
= C_{in}(AB + \bar{A}\bar{B}) + \bar{C}_{in}(\bar{A}B + A\bar{B}) \\
= C_{in}(A \oplus B) + \bar{C}_{in}(A \oplus B) \\
= C_{in} \oplus (A \oplus B) \\
= A \oplus B \oplus C_{in}
\]
## The Full Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C\textsubscript{in}</th>
<th>C\textsubscript{out}</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
C\text{out} = AB + AC\text{in} + BC\text{in}
\]
The Full Adder

\[ \text{Sum} = A \oplus B \oplus C_{\text{in}} \]

\[ C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}} \]
The Full Adder

module fulladder1 (input logic A,B,Cin output logic Sum,Cout);

    assign Sum = A ^ B ^ Cin;
    assign Cout = A & B | A & Cin | B & Cin;

endmodule
The Full Adder (2)

\[ \text{Sum} = A \oplus B \oplus C_{\text{in}} \]
\[ = (A \oplus B) \oplus C_{\text{in}} \]

\[ C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}} \]
\[ = AB + \overline{A}BC_{\text{in}} + A\overline{B}C_{\text{in}} \]
\[ = AB + C_{\text{in}}(\overline{A}B + A\overline{B}) \]
\[ = AB + C_{\text{in}}(A \oplus B) \]
The Full Adder (2)

\[ Sum = (A \oplus B) \oplus C_{in} \]

\[ C_{out} = AB + C_{in} (A \oplus B) \]
module fulladder2(input logic A,B,Cin  output logic Sum,Cout);

logic Sum1,Cout1,Cout2;  // local signals
halfadder h1 (A,B,Sum1,Cout1);  // halfadder 1
halfadder h1(Cin,Sum1,Sum,Cout2);  // halfadder 2
assign Cout = Cout1 | Cout2;  // OR gate

endmodule
The Multi-bit Adder

- Multi-bit Adder with ‘Ripple Carry’
The Multi-bit Adder

- Carry propagation?

- Delay = AND + OR

- The gate structure can be optimised for a combined 2-input AND-OR gate
Exercise: The Multi-bit Adder

- An 8-bit adder uses a ripple carry chain with the propagation delay of 3ns in the carry propagation path in each bit. The XOR gates at each bit have a 12ns propagation delay. What is the total addition time?

Is the total delay:
- a) 42ns?
- b) 24ns?
- c) 36ns?
- d) 45ns?
- e) 48ns?
Signed Binary Arithmetic
2’s Complement Arithmetic

- To subtract B from A, we can add A to \(-B\)... 

\[ A - B = A + (-B) = A + (\overline{B} + 1) \]

- E.g. Using 2’s complement 8-bit binary, subtract 40\text{_{10}} \text{ from } 29\text{_{10}} \text{ (29\text{_{10}} - 40\text{_{10}}) }

\[
\begin{array}{c c c c c c c c}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
- \hspace{1cm} & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

\(=+29_{10}\)

\(=+40_{10}\)

\[\text{To verify...}\]

\[
\begin{array}{c c c c c c c c}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
\text{flip the bits}
\end{array}
\]

\[
\begin{array}{c c c c c c c c}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\text{add 1}
\end{array}
\]

\(=+11_{10}\)
2’s Complement Multi-bit Adder
2’s Complement Multi-bit Adder

- What is the $F$ (‘function’) input for?
  - $F = 0 \to \text{Add}; F = 1 \to \text{Subtract}$

- The XOR gates ‘flip the bits’ of $B$ if $F = 1$

- Connecting $C_o$ to $F$ ‘adds one’ to the value when subtracting

<table>
<thead>
<tr>
<th>$F$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$B_n$</td>
</tr>
<tr>
<td>1</td>
<td>$\overline{B_n}$</td>
</tr>
</tbody>
</table>
Unsigned Overflow Detection

- How do we detect overflow?
- For example, add $186_{10} + 99_{10}$:

\[
\begin{array}{cccccccc}
& 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
\hline
& 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

= $186_{10}$
= $99_{10}$

- $186_{10} + 99_{10} = 285_{10} = 10001101_2$ (i.e. doesn’t fit in 8 bits)
- If the Most Significant Carry Bit = 1, overflow has occurred.
2’s Complement Overflow

- We’ve seen that, unlike unsigned binary arithmetic, $C_{out} = 1$ on the MSB does not signify overflow in 2’s complement
  - Remember $27_{10} - 11_{10} = 16_{10}$ resulted in $C_{out} = 1$ on the MSB

- Furthermore, we also know that $C_{out} = 0$ on the MSB does not signify overflow in 2’s complement
  - Remember $29_{10} - 40_{10} = -11_{10}$ resulted in $C_{out} = 0$ on the MSB

- So, how do we detect overflow in 2’s complement arithmetic?
2’s Complement Overflow

- Overflow in 2’s complement arithmetic (addition and subtraction) occurs if:

1) Both addition arguments are positive (the MSB are 0) and the result is negative (the MSB is 1)

```
0 0
  ADD 1
0 0
```

2) Both addition arguments are negative (the MSB are 1) and the result is positive (the MSB is 0).

```
1 1
  ADD 0
1 1
```
2’s Complement Overflow

- An inspection of the full-adder truth table reveals that the 2’s complement overflow detection rule can be phrased as follows:

An overflow occurs in 2’s complement addition if the carry bit propagating into the MSB ($C_{n-1}$) and the carry bit generated by the MSB ($C_n$) are different. The logic XOR operation can be used to generate the overflow signal $V$.

$$V = C_{n-1} \oplus C_n$$
The Arithmetic Logic Unit (ALU)
(Basic) Computer Architecture

Source: https://secure.ecs.soton.ac.uk/notes/ellabs/databook/avr/ATmega48_full.pdf
ALU with Single Data Bus

• Accumulator
  – Stores the result from the ALU and is always operand A.
  – Is also connected to the data bus to allow the value in the accumulator to be stored in memory.

• Why?

• For example, we want to:
  – 1. Add 41 to 63 (both values stored in memory)
  – 2. Subtract 27 from the result
  – 3. Store the result in memory
ALU with Double Data Bus

Data Bus A

Data Bus B

Registers

A[]

B[]

ALU

Instruction

Control Logic

Function Select

R[]

Status Flags
The Arithmetic Logic Unit

Source: Atmel / www.atmel.com
ALU Status Bits

- Registered output flags
  - C: Carry-out from MSB
  - V: Overflow detected
  - N: Result is negative
  - Z: Result is zero
The Arithmetic Logic Unit

Function Select → ALU → Status Flags

A[] → ALU → B[]

R[]
‘An’ ALU Bitslice

The 1-bit slice of a simple ALU presented here shows logic circuitry that can program the ALU to perform a number of arithmetic and logic functions. The functions are selected with the function select bits $F_3$, ..., $F_0$. Sample functions of the ALU are shown in the table below.

<table>
<thead>
<tr>
<th>ALU Function</th>
<th>$F_4$</th>
<th>$F_3$</th>
<th>$F_2$</th>
<th>$F_1$</th>
<th>$F_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R=A$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R=M$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R=A+M$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$R=A\cdot M$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$R=A+\bar{A}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$R=\bar{A}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$R=M$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$R=A$ and $M$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R=A$ or $M$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R=A \text{ xor } M$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R=A$ and $M$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$R=A$ nor $M$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$R=A$ xor $M$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1-11. 1-bit slice of a simple ALU.

Tom Kazmierski, Digital Techniques & Microprocessors PII (Computer Architecture)
Summary

In these lectures, we have covered...

- Signed Number Systems
  - Sign-Mantissa
  - Negative Excess
  - One’s Complement
  - Two’s Complement
  - Adding Signed Numbers
  - Subtracting Signed Numbers

- Computation in Hardware
  - The Half Adder
  - The Full Adder
  - The Multi-Bit Adder
  - 2’s Complement Multi-Bit Adder
  - The ALU
Further Reading

  - Old book: Chapter 6 (Pages 178-208)
  - New book: Chapter 7 (Pages 133-157)

- M Mano, M Ciletti “Digital Design”
  - Chapter 1: Digital Systems and Binary Numbers

- J Wakerly “Digital Design - Principles & Practices”
  - Chapter 2: Number Systems and Codes
  - Chapter 6-10: Adders, Subtractors and ALUs

- R Tocci, N Widmer, G L Moss “Digital Systems - Principles and Applications”
  - Chapter 2: Number Systems and Codes
  - Chapter 6: Digital Arithmetic – Operations and Circuits
Questions?