Binary Arithmetic

- Addition

\[
\begin{array}{c}
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & + & 6 \\
\hline
1 & 0 & 0 & 0 & 1
\end{array}
\]

\[
\begin{array}{c}
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & + & 11 & + & 17 \\
\hline
1 & 0 & 0 & 0 & 0 & 1
\end{array}
\]

\[
\begin{array}{c}
1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & + & 22 & + & 33 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]
What happens if we run out of digits?

- Adding two numbers each stored in 1 byte (8 bits) may produce a 9-bit result

$$\begin{array}{c}
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}$$

8-bit result was $01000011_2$ or $67_{10}$

- Added $156_{10} + 167_{10}$ and expected to get $323_{10}$
- 8-bit result was $01000011_2$ or $67_{10}$
- Largest number we can represent in 8-bits is 255
- The “missing” or “left-over” 1 is called a carry (or carry-out)
Some instructions can optionally update the **Condition Code Flags** to provide information about the result of the execution of the instruction:

- e.g. whether the result of an addition was zero, or negative or whether a carry occurred

<table>
<thead>
<tr>
<th>Condition Code Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>N – <strong>Negative</strong></td>
</tr>
<tr>
<td>Z – <strong>Zero</strong></td>
</tr>
<tr>
<td>V – <strong>Overflow</strong></td>
</tr>
<tr>
<td>C – <strong>Carry</strong></td>
</tr>
</tbody>
</table>
The Condition Code Flags (N, Z, V, C) can be optionally updated to reflect the result of an instruction.

S-bit in a machine code instruction is used to tell the processor whether the Condition Code Flags should be updated, based on the result.

- e.g. ADD instruction
- Condition Code Flags only updated if S-bit (bit 20) is 1

In assembly language, we cause the Condition Code Flags to be updated by appending “S” to the instruction mnemonic (e.g. ADDS, SUBS, MOVS).
• **ADDS** causes the Condition Code Flags to be updated
• 32-bit arithmetic
• Expected result?
• Does the result fit in 32-bits?
• Will the carry flag to be set?
• Examine flags using μVision IDE

Program 4.1 – Carry

```
start
  LDR r0, =0xC0000000
  LDR r1, =0x70000000
  ADDS r0, r0, r1
stop   B     stop
```
Negative Numbers

- The following binary value is stored in memory. What does the value represent?
  - $01001101_2$

- Interpretation!

- How can we represent signed values, and negative values such as $-17_{10}$ in particular, in memory?

- How can we tell whether any given value in memory represents an unsigned value, a signed value, an ASCII character, ...
- Represent signed values in the range [-127 ... +127]
- Two representations of zero (+0 and -0)
- Need special way to handle signed arithmetic
- Remember: interpretation! (is it -8 or 2,147,483,656?)
A 12-hour clock is an example of modulo-12 arithmetic.

- If we add 4 hours to 10 o’clock we get 2 o’clock.
- If we subtract 4 from 2 o’clock we get 10 o’clock (not -2 o’clock!)
2’s Complement

- A 4-bit number system allows us to represent 16 values
- Ignoring carries from addition gives us modulo-16 arithmetic

- \((15 + 1) \mod 16 = 0\)
  - and \(-1 + 1 = 0\)
- \((14 + 2) \mod 16 = 0\)
  - and \(-2 + 2 = 0\)
- \((14 + 4) \mod 16 = 2\)
  - and \(-2 + 4 = 2\)
2’s Complement

Remember: interpretation!!
2’s Complement

- What is the range of signed values that can be represented with 32 bits using the 2’s Complement system?
- How would the values -4 and +103 be represented using a 32-bit 2’s Complement system?
- How many representations for zero are there?
- How can you tell whether a value represented using a 2’s Complement system is positive or negative?
- How can we change the sign of a number represented using a 2’s Complement number system?
Represent $97_{10}$ using 2’c complement

- $97_{10} = 0110\ 0001_2$
- Inverting gives $1001\ 1110_2$
- Adding 1 gives $1001\ 1111_2$

- Interpreted as a 2’s complement integer, $1001\ 1111_2 = -97_{10}$
- Interpreted as an unsigned integer, $1001\ 1111_2 = 159_{10}$

- $(159 + 97) \mod 256 = 0$

- Correct interpretation is the responsibility of the programmer, not the CPU, which does not “know” whether a value $1001\ 1111_2$ in R0 is $-97_{10}$ or $159_{10}$
Adding $0110\ 0001_2$ ($+97_{10}$) and $1001\ 1111_2$ ($-97_{10}$)

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

• Ignoring the carry bit yields correct signed result of 0

Changing sign of $1001\ 1111_2$ ($-97_{10}$)

• Invert bits and add 1 again!!
• Inverting gives $0110\ 0000_2$
• Adding 1 gives $0110\ 0001_2$ ($+97_{10}$)
Program 4.2 – Negate Value

- Write an assembly language program to change the sign of the value stored in r0
- Sign of a 2’s Complement value can be changed by inverting the value and adding one

```
start
  LDR r0, =7 ; value = 7 (simple test value)
  MVN r0, r0 ; value = NOT value (invert bits)
  ADD r0, r0, #1 ; value = value + 1 (add 1)
stop
  B stop
```

- Use of LDR r0, =7 to load a test value (7) into r0
- MVN instruction moves a value from one register to another register and negates (inverts) the value (note same register)
- Use of ADD with immediate constant value #1 (note syntax)
Subtraction

- $A - B$

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

- $A + (TC(B))$

\[
\begin{array}{c c c c c c c c}
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]
Subtraction

- **A − B**

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

- **A + (TC(B))**

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

- When performing a subtraction operation, if the **carry bit** is set then a borrow **did not** occur, otherwise ...
Chapter 4 of the ARM Architecture Reference Manual (known as “ARMARM”)

**ADD**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>shifter operand</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADD adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADD can optionally update the condition code flags, based on the result.

**Syntax**

ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

where:

- **<cond>** is the condition under which the instruction is executed. *The condition field* on page A3-3. If **<cond>** is omitted, the AL (always) condition is used.

- **S** Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:
  - If <Rd> is not R15, the N and Z flags are set according to the result of the addition, and the C and V flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
  - If <Rd> is R15, the CPSR of the target register is copied to the CPSR. This form of the
ARMARM tells us how each instruction (optionally) effects the Condition Code Flags

- e.g. ADD

```plaintext
Operation

if ConditionPassed(cond) then
    Rd = Rn + shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand)
        V Flag = OverflowFrom(Rn + shifter_operand)
```

`UNPREDICTABLE`
• e.g. SUBtract

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand
if S == 1 and Rd == R15 then
    if CurrentModeHasSPSR() then
        CPSR = SPSR
    else UNPREDICTABLE
else if S == 1 then
    N Flag = Rd[31]
    Z Flag = if Rd == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - shifter_operand)
    V Flag = OverflowFrom(Rn - shifter_operand)
```
Zero and Negative Flags

- Zero Condition Code Flag is (optionally) set if the result of the last instruction was zero.
- Negative Condition code flag is (optionally) set if the result of the last instruction was negative:
  - i.e. if the Most Significant Bit (MSB) of the result was 1.
- e.g. SUBtract instruction

```plaintext
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else
            UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - shifter_operand)
    V Flag = OverflowFrom(Rn - shifter_operand)
```
2’s Complement Examples

- Add +97 and +45

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

- Result is 1000 1110\(_2\) (142\(_{10}\), or -114\(_{10}\))

- If we were interpreting values as signed integers, we got an incorrect result
  - With 8-bits, the highest +ve integer we can represent is +127
  - We added two +ve numbers and obtained a –ve result
  - 1000 1110\(_2\) (-114\(_{10}\))
  - The result is outside the range of the **signed** number system
If the result of an addition or subtraction gives a result that is outside the range of the signed number system, then an overflow has occurred.

The processor sets the overflow Condition Code Flag after performing an arithmetic operation to indicate whether an overflow has occurred.

Current Program Status Register

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
<th>Reserved</th>
<th>Control Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>8 7 0</td>
</tr>
</tbody>
</table>

Condition Code Flags
The Carry and Overflow flags are set by the processor using a set of rules.

Remember, the processor does not “know” whether we, as programmers, are choosing to interpret stored values as signed or unsigned values:

- e.g. we could interpret the binary value $10001110_2$ as either $142_{10}$ (unsigned) or $-114_{10}$ (signed).
- we could also interpret it as the ASCII code for “Ä”.
- ...

The C and V flags are set by the processor and it is the responsibility of the programmer to choose:

- whether to interpret C or V
- how to interpret C or V
Overflow Rules

- **Addition rule (r = a + b)**
  \[\overline{V} = 1 \text{ if } \text{MSB}(a) = \text{MSB}(b) \text{ and } \text{MSB}(r) \neq \text{MSB}(a)\]
  - i.e. overflow occurs for addition if the operands have the same sign and the result has a different sign

- **Subtraction rule (r = a − b)**
  \[\overline{V} = 1 \text{ if } \text{MSB}(a) \neq \text{MSB}(b) \text{ and } \text{MSB}(r) \neq \text{MSB}(a)\]
  - i.e. overflow occurs for subtraction if the operands have different signs and the sign of the result is different from the sign of the first operand
Signed interpretation: (+112) + (-80) = +32
Unsigned interpretation: 112 + 176 = 288
By examining the V flag, we know that if were interpreting the values as signed integers, the result is correct
If we were interpreting the values as 8-bit unsigned values, C = 1 tells us that the result was too large to fit in 8-bits
Signed: \((-80) + (-80) = -160\)

Unsigned: \(176 + 176 = 352\)

By examining the \(V\) flag \((V = 1)\), we know that if were interpreting the values as signed integers, the result is outside the range of the signed number system.

If we were interpreting the values as 8-bit unsigned values, \(C = 1\) tells us that the result was too large to fit in 8-bits.

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

\[+\]

\[\begin{array}{cccccccc}
C_{\text{carry}} &= 1 \\
O_{\text{overflow}} &= 1
\end{array}\]
Many instructions can optionally cause the processor to update the Condition Code Flags (N, Z, V, and C) to reflect certain properties of the result of an operation:

- Append “S” to instruction in assembly language (e.g. ADDS)
- Set S-bit in machine code instruction

- **N** flag set to 1 if result is negative (if MSB is 1)
- **Z** flag is set to 1 if result zero (all bits are 0)
- **C** flag set if carry occurs (addition) or borrow does not occur (subtraction)
- **V** flag set if overflow occurs for addition or subtraction
### Example: 64-bit Addition

- **64-bit addition on a 32-bit processor**

  ![Binary Addition Diagram]

- Split operation into two 32-bit operations
- Add lower 32 bits, followed by upper 32 bits
- Any “carry-out” from the addition of the lower 32 bits must be “carried-in” to the addition of the upper 32 bits
Use **ADDS** instruction to add lower 32 bits, causing the processor to set the **C**arry flag if a carry-out occurs.

Use **ADC** instruction to add upper 32 bits, **plus 1 if the **C**arry flag was set** (by the preceding **ADDS** instruction).

### Program 4.3 – 64-bit Addition

- **start**
  - ; Set some test values for A and B
    - LDR r2, =0x00000E44 ; Aupr
    - LDR r3, =0x32A84FE6 ; Alwr
    - LDR r4, =0x00000000 ; Bupr
    - LDR r5, =0xF4E00322 ; Blwr
  - ; Add A and B
    - ADDS r0, r3, r5 ; Rlwr = Alwr + Blwr, update Cout
    - ADC r1, r2, r4 ; Rupr = Aupr + Blwr + Cout

- **stop**
  - B stop

- Same technique can be extended to larger values