Program Flow Control

• BY DEFAULT, the program counter is incremented by 4 when each instruction is executed (PC = PC + 4)

• next sequential instruction is then fetched, decoded and executed

• need to be able alter this sequential program flow in order to write more useful programs

• normally a condition is tested and a decision made whether to execute
  ▪ the next sequential instruction OR
  ▪ an instruction at a different address

• need to learn about condition code flags and branch instructions
Condition Code Flags

- CPU contains a Current Program Status Register (CPSR) containing 4 condition code flags

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>V</th>
<th>RESERVED</th>
<th>Control bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current Program Status Register (CPSR)

- the 4 flags can optionally reflect the result of an instruction (eg ADD, SUB)

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>V</th>
<th>will discuss carry and overflow later</th>
</tr>
</thead>
<tbody>
<tr>
<td>negative</td>
<td>zero</td>
<td>carry</td>
<td>overflow</td>
<td></td>
</tr>
</tbody>
</table>

\[ N = \text{MSB(result)} \]
\[ Z = 1 \text{ if result} = 0, Z = 0 \text{ if result} \neq 0 \]
Condition Code Flags

- the condition code flags are updated if the S bit, encoded in the machine code of the instruction, is set

- at the assembly language level, an “S” is appended to the instruction mnemonic (e.g. ADDS, SUBS, MOVS) to indicate that the instruction should update the condition code flags when executed
CMP Instruction

• the CMP instruction subtracts its operands (like a subtract instruction) and sets the condition code flags **without** storing the result

```
CMP R1, #3 ; set condition code flags to reflect result of R1 – 3
```

• the resulting condition codes allows the CPU to branch if the CMP operands were **equal**, **not equal**, **less than**, **less than or equal**, **greater than or equal** **OR** **greater than** each other (=, !=, <, <=, >=, >)

• since CMP always sets the condition code flags, here is no need for a CMPS mnemonic
Branch Instructions (Bxx)

- BY DEFAULT, the CPU increments the PC by 4 (the size of one instruction) to "point to" the next sequential instruction in memory
- A branch instruction can modify the PC thus breaking the pattern of sequential execution
- Bxx _ L ; xx = condition
- Branch instructions EITHER
  1. unconditional OR
     - always branches
  2. conditional
     - branches if condition TRUE
     - executes next sequential instruction if condition FALSE
     - condition based on condition code flags
# Conditional Branch Instructions

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
<th>C/C++/Java</th>
<th>Instruction</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch equal or not equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td>=</td>
<td>==</td>
<td>BEQ</td>
<td>EQual</td>
</tr>
<tr>
<td>not equal</td>
<td>≠</td>
<td>!=</td>
<td>BNE</td>
<td>Not Equal</td>
</tr>
<tr>
<td>unsigned branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>BLO (or BCC)</td>
<td>Lower</td>
</tr>
<tr>
<td>less than or equal</td>
<td>≤</td>
<td>&lt;=</td>
<td>BLS</td>
<td>Lower or Same</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>≥</td>
<td>&gt;=</td>
<td>BHS (or BCS)</td>
<td>Higher or Same</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>BHI</td>
<td>Higher</td>
</tr>
<tr>
<td>signed branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>BLT</td>
<td>Less Than</td>
</tr>
<tr>
<td>less than or equal</td>
<td>≤</td>
<td>&lt;=</td>
<td>BLE</td>
<td>Less than or Equal</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>≥</td>
<td>&gt;=</td>
<td>BGE</td>
<td>Greater than or Equal</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>BGT</td>
<td>Greater Than</td>
</tr>
<tr>
<td>branch on flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negative Set</td>
<td>BMI</td>
<td></td>
<td></td>
<td>Minus</td>
</tr>
<tr>
<td>Negative Clear</td>
<td>BPL</td>
<td></td>
<td></td>
<td>PLus</td>
</tr>
<tr>
<td>Carry Set</td>
<td>BCS (or BHS)</td>
<td></td>
<td></td>
<td>Carry Set</td>
</tr>
<tr>
<td>Carry Clear</td>
<td>BCC (or BLO)</td>
<td></td>
<td></td>
<td>Carry Clear</td>
</tr>
<tr>
<td>Overflow Set</td>
<td>BVS</td>
<td></td>
<td></td>
<td>oVerflow Set</td>
</tr>
<tr>
<td>Overflow Clear</td>
<td>BVC</td>
<td></td>
<td></td>
<td>oVerflow Clear</td>
</tr>
<tr>
<td>Zero Set</td>
<td>BEQ</td>
<td></td>
<td></td>
<td>EQual</td>
</tr>
<tr>
<td>Zero Clear</td>
<td>BNE</td>
<td></td>
<td></td>
<td>Not Equal</td>
</tr>
</tbody>
</table>
Conditional Branch Instructions

• need signed and unsigned branches
  
  ▪ depends on whether CMP operands are being interpreted as signed or unsigned integers

• there also branch instructions that directly test the condition flags N, Z, C and V
Using branch instructions in Assembly Language

• example

    B      L1 ; unconditional branch to L1
    ...

    L1 ...
    CMP   R0, #42 ; set condition code flags
    BEQ   L2 ; conditional branch to L2
    ...

    L2 ...

• labels start in column 1 (otherwise column 1 should be empty except for comments)
• labels must NOT begin with a digit (0..9)
• labels can contain UPPER and lower case letters, digits and _ (underscore)
• labels are case sensitive (mylabel is not the same as MyLabel)
• labels must be unique within an assembly language file
Compute the absolute value

- write assembly language instructions to compute the absolute value of the signed integer stored in register R1

- algorithm expressed in pseudo-code and as a flowchart

  ```
  if (v < 0) {
      v = -v;
  }
  ```

- flowchart has a diamond shaped decision boxes with No and Yes exit points

- statement v = -v is conditionally executed (if v < 0)

- flowcharts considered old-fashioned by some, but they are a good way of illustrating how an algorithm works
Compute the absolute value ...

• \(-v\) computed by calculating \(0 - v\)

\[
\begin{align*}
\text{CMP} & \quad R1, \#0 \quad ; \quad v < 0 \, ? \\
\text{BGE} & \quad L1 \quad ; \quad \geq \, (\text{opposite condition to} \,< \, \text{in pseudo code}) \\
\text{RSB} & \quad R1, R1, \#0 \quad ; \quad v = 0 - v \\
\end{align*}
\]

L1 ...

• note the use of RSB (reverse subtract)

• signed branch since \(v\) is a signed integer

• RSB executed if \(v < 0\)
Compute the maximum of 3 values

- write assembly language instructions to compute the maximum value of 3 signed integers a, b and c
- algorithm expressed in pseudo-code and as a flowchart
  
  ```
  max = a;
  if (b > max)
      max = b;
  if (c > max)
      max = c
  ```
- statements max = b and max = c are conditionally executed depending on the values of a, b and c
Compute the maximum of 3 values ...

- assume the maximum value stored in R0
- assume the 3 variables a, b and c are stored in R1, R2 and R3 respectively

```
MOV R0, R1 ; max = a
CMP R2, R0 ; b > max ?
BLE L1 ; <= (opposite condition to > in pseudo code)
MOV R0, R2 ; max = b
L1 CMP R3, R0 ; c < max ?
BLE L2 ; <= (opposite condition to > in pseudo code)
MOV R0, R3 ; max = c
L2 ...
```

- signed branches as a, b and c are signed integers
Compute n!

- write assembly language instructions to compute n factorial
- algorithm expressed in pseudo-code and as a flowchart

```
n = 6;
r = 1;
while (n > 1) {
    r = r * n;
    n = n - 1;
}
```

- n is modified by algorithm
- value of r each time around loop
  \[ r = 1, r = 1*6, r = 1*6*5, r = 1*6*5*4, r = 1*6*5*4*3, r = 1*6*5*4*3*2 \]
- \( r = 720 = 6! \)
Compute n! ...

- result in R0 and n in R1

```
MOV R1, #6 ; n = 6
MOV R0, #1 ; r = 1
L1 CMP R1, #1 ; n > 1 ?
    BLS L2 ; <= (opposite condition to > in pseudo code)
MUL R0, R1, R0 ; r = r*n
SUB R1, R1, #1 ; n = n - 1
B L1 ; repeat
L2 ...
```

- unsigned branch (a signed branch would work equally well here)
Compute the $n^{th}$ Fibonacci Number

- write an assembly language program to compute the $n^{th}$ Fibonacci number $F_n$
- the $n^{th}$ Fibonacci number is defined recursively as follows
  \[ F_n = F_{n-1} + F_{n-2} \text{ where } F_0 = 0 \text{ and } F_1 = 1 \]
- 0, 1, 1, 2, 3, 5, 8, ...

```plaintext
n = 6;
fa = 0;
fb = 1;
while (n > 1) {
    tmp = fb;
    fb = fa + fb;
    fa = tmp;
    n = n - 1;
}
```

edges case

- $n \leq 0$?
- $n$ such that result $> 2^{32} - 1$ ?
- $n$ such that result $> 2^{31} - 1$ ?

which applies depends on whether integers are interpreted as being signed or unsigned
Compute the $n^{th}$ Fibonacci Number ...

- store result and $fb$ in R0, $fa$ in R1, $n$ in R2 and tmp in R3

```
MOV  R2, #6 ; n = 6
MOV  R1, #0 ; fa = 0
MOV  R0, #1 ; fb = 1
L0   CMP  R2, #1 ; n > 1?
      BLE  L1 ; <= (opposite condition to > in pseudo code)
      MOV  R3, R0 ; tmp = fa
      ADD  R0, R0, R1 ; fb = fb + fa
      MOV  R1, R3 ; fa = tmp
      SUB  R2, R2, #1 ; n = n - 1
      B    L0 ; repeat
L1   ...
```

- signed branch (BLE branch less than or equal)
- unsigned branch BLS (branch lower or same) would also work here
**FLOW CONTROL**

**IF ... THEN ... ELSE**

```c
if (x < 9) {
    x = x + 1;
} else {
    x = 0;
}
```

// assume x in R1
// execute if condition TRUE

```c
CMP R1, #9 ; x < 9?
BGE L1 ; >= (opposite condition to < in pseudo-code)
ADD R1, R1, #1 ; x = x + 1
B L2 ; skip else
L1 MOV R1, #0 ; x = 0
L2 ...
```

• signed branches (assume x is a signed integer)
• must remember to skip ELSE part if condition TRUE
Conditional AND

```java
if ((x > 40) && (x < 50)) {    // && (AND)
    y = y + 1;        // s0
} else {
    z = z + 1;        // s1
}
```

• if both comparisons TRUE then execute s0 else execute s1

• comparisons made in order, left to right

• called “conditional AND” since second comparison doesn’t need to be made if first comparison FALSE

• if both comparisons TRUE, execute s0 and branch to end of construct (skip s1)

• if either comparison FALSE, execute s1
Conditional AND ...

- $x$ in R1, $y$ in R2 and $z$ in R3

```
CMP  R1, #40  ; if $x > 40$
BLE  L1      ; goto L1  (opposite condition to pseudo-code)
CMP  R1, #50  ; if $x < 50$
BGE  L1      ; goto L1  (opposite condition to pseudo-code)
ADD  R2, R2, #1 ; $y = y + 1$ (s0 executed if condition TRUE)
B    L2      ; goto L2  (skip else statement)
ADD  R3, R3, #1 ; $z = z + 1$ (s1 executed if condition FALSE)
```

- signed branches (assume $x$ is a signed integer)
Conditional OR

```java
if ((x == 40) || (x == 50)) { // || (OR)
    y = y + 1; // s0
} else {
    z = z + 1; // s1
}
```

- if either comparison TRUE then execute s0 else execute s1
- comparisons made in order, left to right
- called “conditional OR” since second comparison doesn’t need to made if first one is TRUE
- assume x in R1, y in R2 and z in R3
Conditional OR ...

CMP R1, #40 ; if x == 40
BEQ L1 ; goto L1
CMP R1, #50 ; if x == 50
BNE L2 ; goto L1 (opposite condition to pseudo code)

L1 ADD R2, R2, #1 ; y = y + 1 (s0 executed if condition TRUE)
B L3 ; goto L3 (skip s1)

L2 ADD R3, R3, #1 ; z = z + 1 (s1 executed if condition FALSE)

• one way to generate code
More on the Condition Code Flags

• 4 condition code flags N, Z, C and V

• Negative (N) – set if MSB(result) == 1

• Zero (Z) – set if result == 0

• Carry (C) – discuss in more detail now

• Overflow (V)
CARRY Flag on Addition

• what happens if two 8 bit unsigned integers are added and the result is too large to fit in 8 bits?

• adding two 8 bit unsigned integers can produce a 9 bit result

• extra bit stored in the CARRY flag

<table>
<thead>
<tr>
<th>hex</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x70</td>
<td>112</td>
</tr>
<tr>
<td>+ 0xA0</td>
<td>+ 160</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0x10</td>
<td>272 = CARRY(256) + 16</td>
</tr>
</tbody>
</table>

• CARRY flag shown in RED

• with 32 bit addition, CARRY flag has a value of $2^{32}$
64 bit Addition using CARRY flag

- ARM CPU hardware performs 32 bit operations
- How can two 64 bit integers be added?
- Use two registers to store each 64 bit integer
  - R0:R1 and R2:R3 used to store 64 bit integers
  - R0 and R2 contain the 32 most significant bits
  - R1 and R3 contain the 32 least significant bits
  - Compute R0:R1 = R0:R1 + R2:R3
64 bit Addition using CARRY flag ...

- add least significant 32 bits using ADDS (will set CARRY flag)
- add most significant bits using ADC (add with CARRY)

ADDs  R1, R1, R3    ; add least significant bits and set CARRY
ADC    R0, R0, R2    ; add most significant bits with CARRY

- method can be extended to 96, 128, ... bit addition
- large binary keys (and arithmetic) used in public key encryption (256 bit keys)
BORROW on Subtraction

• what happens if subtracting a larger 8 bit unsigned integers from a smaller one?

• results in a BORROW

<table>
<thead>
<tr>
<th>hex</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x70</td>
<td>112</td>
</tr>
<tr>
<td>- 0xA0</td>
<td>-160</td>
</tr>
<tr>
<td>0xD0</td>
<td>-48</td>
</tr>
</tbody>
</table>

= BORROW(-256) + 208 (0xD0)

• BORROW shown in RED
CARRY Flag on Subtraction ...

- turns out that BORROW = NOT CARRY (or CARRY = NOT BORROW)
- illustrate by performing subtraction by adding the 2’s complement
- \( 0x70 - 0xA0 \)
- 2’s complement \( 0xA0 = 0x5F + 1 = 0x60 \)

\[
\begin{array}{c|c}
\text{hex} & \text{unsigned} \\
0x70 & 112 \\
+ 0x60 & -160 \\
\hline
0xD0 & -48 = \text{BORROW}(-256) + 208 (0xD0)
\end{array}
\]

- \( \text{CARRY} = 0 \) shown in RED, therefore \( \text{BORROW} = 1 \)
64 bit Subtraction using CARRY flag

- compute \( R0:R1 = R0:R1 - R2:R3 \)
- subtract least significant 32 bits using SUBS (will set CARRY flag)
- subtract most significant bits using SBC (subtract with CARRY)

\[
\text{SUBS } R1, R1, R3 ; \text{ subtract least significant bits and set CARRY} \\
\text{SBC } R0, R0, R2 ; \text{ subtract most significant bits with CARRY}
\]

- SBC should really be subtract with BORROW! since \( \text{dst} = \text{src2} - \text{src1} + \text{CARRY} - 1 \)
- if BORROW (CARRY = 0) \( \text{dst} = \text{src1} - \text{src2} - 1 \) (if NOT BORROW \( \text{dst} = \text{src1} - \text{src2} \))

\[\begin{array}{c|c}
32 \text{ MS bits} & 32 \text{ LS bits} \\
\hline
\text{R0} & \text{R1} \\
\text{R2} & \text{R3}
\end{array}\]

- ALSO RSC reverse subtract with CARRY
- \( \text{dst} = \text{src2} - \text{src1} + \text{CARRY} - 1 \)
OVERFLOW flag

- if the result of an addition or subtraction is outside the signed number range then an OVERFLOW occurs.

- determined by testing the MSB of the source operands and result.

- for addition $r = a + b$

  $$V = 1 \quad \text{if} \quad \text{MSB}(a) == \text{MSB}(b) \quad \&\& \quad \text{MSB}(r) != \text{MSB}(a)$$

- for subtraction $r = a - b$

  $$V = 1 \quad \text{if} \quad \text{MSB}(a) != \text{MSB}(b) \quad \&\& \quad \text{MSB}(r) != \text{MSB}(a)$$
Example Condition Code Flags after CMP instruction

- compare 0x70000000 with 0xA0000000

\[
\begin{array}{ll}
\text{LDR} & R0, =0x70000000 ; R0 = 0x70000000 \\
\text{LDR} & R1, =0xA0000000 ; R1 = 0xA0000000 \\
\text{CMP} & R0, R1 ; \text{set flags on result of } 0x70000000 - 0xA0000000
\end{array}
\]

- CMP always sets the four condition code flags

\[
\begin{array}{l}
\text{hex} & \text{unsigned} & \text{signed} \\
N = 1 & 0x70000000 & 1,879,048,192 & 1,879,048,192 \\
Z = 0 & -0xA0000000 & -2,684,354,560 & -1,610,612,736 \\
C = 0 (\text{BORROW} = 1) & -0xA0000000 & -2,684,354,560 & -1,610,612,736 \\
V = 1 & 0x0D000000 & -805,306,368 & 3,489,660,928 \\
\end{array}
\]

- remember for subtraction V = 1 if MSB(a) != MSB(b) & MSB(r) != MSB(a)

\[\text{CARRY}\]
Example Condition Code Flags after CMP instruction ...

• unsigned interpretation

  0x0D0000000 = 3,489,660,928 (1,879,048,192 - 2,684,354,560 = -805,306,368)

  subtracting a larger integer from a smaller one \therefore BORROW = 1 (CARRY = 0)

• signed interpretation

  0x0D0000000 = -805,306,368 (1,879,048,192 + 1,610,612,736 = 3,489,660,928)

  result not in range \therefore OVERFLOW = 1 (two minuses make a +)

• unsigned branches - test CARRY and ZERO flags

• signed branches test - OVERFLOW and ZERO flags
Flow Control

Check using uVision

Stop after CMP instruction

Condition code flags
Branch Instructions in Machine Code

- branch instructions are encoded in machine code as follows

```
if (condition) {
    PC = PC + 8 + 4*offset; // condition TRUE
} else {
    PC = PC + 4; // condition FALSE
}
```

- 4 bit condition field determines which condition is tested

- since instructions are 4 bytes, offset is multiplied by 4 (equivalent to appending two least significant zero bits) before being signed extended and added to the PC

- due to the fetch-decode-execute pipeline, the PC has increased by 8 by the time this addition takes place (assembler takes this into account when calculating offsets)
### Branch Condition field

<table>
<thead>
<tr>
<th>Bxx</th>
<th>code</th>
<th>Condition Code Flag Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>0000</td>
<td>Z set</td>
</tr>
<tr>
<td>NE</td>
<td>0001</td>
<td>Z clear</td>
</tr>
<tr>
<td>CS / HS</td>
<td>0010</td>
<td>C set</td>
</tr>
<tr>
<td>CC / LO</td>
<td>0011</td>
<td>C clear</td>
</tr>
<tr>
<td>MI</td>
<td>0100</td>
<td>N set</td>
</tr>
<tr>
<td>PL</td>
<td>0101</td>
<td>N clear</td>
</tr>
<tr>
<td>VS</td>
<td>0110</td>
<td>V set</td>
</tr>
<tr>
<td>VC</td>
<td>0111</td>
<td>V clear</td>
</tr>
<tr>
<td>HI</td>
<td>1000</td>
<td>C set and Z clear</td>
</tr>
<tr>
<td>LS</td>
<td>1001</td>
<td>C clear or Z set</td>
</tr>
<tr>
<td>GE</td>
<td>1010</td>
<td>N set and V set, or N clear and V clear</td>
</tr>
<tr>
<td>LT</td>
<td>1011</td>
<td>N set and V clear, or N clear and V set</td>
</tr>
<tr>
<td>GT</td>
<td>1100</td>
<td>Z clear, or N set and V set, or N clear and V clear</td>
</tr>
<tr>
<td>LE</td>
<td>1101</td>
<td>Z set, or N set and V clear, or N clear and V set</td>
</tr>
<tr>
<td>none / AL</td>
<td>1110</td>
<td>ALWAYS</td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Unconditional Branch Instruction Example

machine code for unconditional branch
Unconditional Branch Instruction Example

• assembly language

    L    B    L ; unconditional branch to L

• instruction @ address 0x00000014
• machine code 0xEAFFFFFE

    0xEA => unconditional branch
    offset => 0xFFFFFE
    sign extend 24 bit offset to 32 bits => 0xFFFFFFFFE
    multiply by 4 (by appending two least significant zero bits) => 0xFFFFFFFF8
    PC = PC + 8 + 0xFFFFFFFF8
    PC = 0x00000014 + 0x00000008 + 0xFFFFFFFF8 => 0x00000014
• unconditional branch to itself [an infinite loop]