Default flow of execution of a program is **sequential**

- After executing one instruction, the next instruction in memory is executed sequentially by incrementing the program counter (PC)

To write useful programs, **sequence** needs to be combined with **selection** and **iteration**
Selection

- if `<some condition>` then execute `<some instruction(s)>`
- if `<some condition>` then execute `<some instruction(s)>`
  otherwise execute `<some other instruction(s)>`
- Examples?

Iteration

- while `<some condition>` is met, repeat executing `<some instructions>`
- repeat `<some instruction(s)>` until `<some condition>` is met
- repeat executing `<some instruction(s)>` x number of times
- Examples?
Design and write an assembly language program to compute $x^4$ using repeated multiplication

Program 6.1 – $x^y$

- Practical but inefficient and tedious for small values of $y$
- Impractical and very inefficient and tedious for larger values
- Inflexible – would like to be able to compute $x^y$, not just $x^4$

```assembly
MOV r0, #1 ; result = 1
MUL r0, r1, r0 ; result = result \times value (value \ ^ 1)
MUL r0, r1, r0 ; result = result \times value (value \ ^ 2)
MUL r0, r1, r0 ; result = result \times value (value \ ^ 3)
MUL r0, r1, r0 ; result = result \times value (value \ ^ 4)
```

Not valid assembly language!!
Program 6.1a – \( x^y \)

result = 1
while (y ≠ 0) {
    result = result × x
    y = y - 1
}

start
LDR r1, =3 ; test with x = 3
LDR r2, =4 ; test with y = 4
MOV r0, #1 ; result = 1
MOVS r2, r2 ; set condition code flags
while
BEQ endwh ; while (y ≠ 0) {
MUL r0, r1, r0 ; result = result × x
SUBS r2, r2, #1 ; y = y - 1
}while
endwh
stop B stop
### Pseudo-Code

**Pseudo-code** is a useful tool for developing and documenting assembly language programs

- No formally defined syntax
- Use any syntax that you are familiar with (and that others can read and understand)
- Particularly helpful for developing and documenting the **structure** of assembly language programs
- Not always a “clean” translation between pseudo-code and assembly language

```
while
    BEQ     endwh
    MUL     r0, r1, r0
    SUBS    r2, r2, #1
    B       while

; while (y ≠ 0) {
;    result = result × x
;    y = y - 1
; }
```
Program 6.1b - $x^y$

```c
if (y == 0) {
    result = 1
} else {
    result = x
    if (y > 1) {
        do {
            result = result * x
            y = y - 1
        } while (y != 0)
    }
}
```
Program 6.1b - $x^y$

start
LDR r1, =3 ; test with $x = 3$
LDR r2, =4 ; test with $y = 4$
CMP r2, #0 ; if ($y = 0$)
BNE else1 ; {
MOV r0, #1 ; result = 1
B endif1 ; }
else1 ; else {
MOV r0, r1 ; result = $x$
CMP r2, #1 ; if ($y > 1$)
BLS endif2 ; {
SUBS r2, r2, #1 ; $y = y - 1$
do1 ; do {
MUL r0, r1, r0 ; result = result $\times x$
SUBS r2, r2, #1 ; $y = y - 1$
BNE do1 ; } while ($y \neq 0$)
endif2 ;
endif1 ;
stop B stop

Comments – not assembled
By default, the processor increments the Program Counter (PC) to “point” to the next instruction in memory ...

... causing the sequential path to be followed

Using a **PC modifying instruction**, we can modify the value in the Program Counter to “point” to an instruction of our choosing, breaking the pattern of sequential execution

PC Modifying Instructions can be

- **unconditional** – always update the PC
- **conditional** – update the PC only if some condition is met (e.g. the **Zero** condition code flag is set)
### Unconditional Branch

- **B** \(\rightarrow\) *Label* ; Branch unconditionally to label
  
  ... ... ; ...
  
  ... ... ; more instructions
  
  ... ... ; ...

- *Label* some instruction ; more instructions
  
  ... ... ; ...

### Machine code for Branch instruction

- Branch target offset is added to current Program Counter value
- Next fetch in fetch \(\rightarrow\) decode \(\rightarrow\) execute cycle will be from new Program Counter address
Labels and Branch Target Offsets

- Use labels to specify branch targets in assembly language programs
  - Assembler calculates necessary branch target offset

  \[ \text{branch target offset} = \frac{((\text{label address} - \text{branch inst. address}) - 8)}{4} \]

- Branch target offset could be negative (branch backwards)
- All ARM instructions are 4 bytes (32-bits) long and must be stored on 4-byte boundaries in memory
- So, branch target offset can be divided by 4 before being stored in the machine code branch instruction
- Allows signed 26-bit target offsets to be stored in 24 bits

```
while
  BEQ endwh
  MUL r0, r1, r0 ; result = result × x
  SUBS r2, r2, #1 ; y = y - 1
  B while
endwh
```
Executing **Branch Instructions**

- Next fetch in fetch $\rightarrow$ decode $\rightarrow$ execute cycle will fetch the instruction at the new PC address.
- 26-bit branch target offset may be negative.
- Must sign-extend a *less-than-32-bit* value before using it to perform 32-bit arithmetic.
- i.e. 26-bit branch target offset must be sign-extended to form a 32-bit value before adding it to the 32-bit Program Counter.

$$PC \leftarrow PC + (\text{branch target offset} \times 4)$$
- Must **sign extend** the 26-bit offset by copying the value of bit 25 into bits 26 to 31 (2’s Complement system)
### Rules

- Must be unique
- Can contain UPPER and lower case letters, numerals and the underscore _ character
- Are case sensitive (mylabel is not the same label as MyLabel)
- Must not begin with a numeral
Unconditional branch instructions are necessary but they still result in an instruction execution path that is pre-determined when we write the program.

To write useful programs, the choice of instruction execution path must be deferred until the program is running:
- i.e. The decision to take a branch or continue following the sequential path must be deferred until “runtime”

Conditional branch instructions will take a branch only if some condition is met when the branch instruction is executed, otherwise the processor continues to follow the sequential path.
### Simple **selection** construct ...

- **In ARM assembly language**
  - assume $a \leftrightarrow r0$, $b \leftrightarrow r1$

```assembly
compare r0 and r1
branch to label endif if they are equal
MOV r0, r1
endif
<rest of program>
```

- Compare $a$ and $b$ by subtracting $b$ from $a$ (SUBS)
- SUBS will set Condition Code Flags. If $a$ is equal to $b$, **Zero** flag will be set. If **Zero** flag is set, branch over $a = b$ using `BEQ`

```assembly
SUBS r12, r0, r1 ; store result anywhere ... not needed
BEQ endif ; take branch if Zero flag set (by SUBS)
MOV r0, r1
endif
<rest of program>
```
Using SUBtract to compare two values, the result has to be stored somewhere, even though it is not needed.

CMP - CoMPare Instruction

- **CMP** (CoMPare) instruction performs a subtraction and updates the Condition Code Flags **without storing the result of the subtraction**.

```plaintext
SUBS  r12, r0, r1 ; store result anywhere ... not needed
BEQ  endif ; take branch if Zero flag set (by SUBS)
MOV  r0, r1
endif <rest of program>

CMP  r0, r1 ; update CC Flags, throw away result
BEQ  endif ; take branch if Zero flag set (by SUBS)
MOV  r0, r1
endif <rest of program>
```
(Un-) Conditional Branch Instructions

<table>
<thead>
<tr>
<th>Branch Instruction</th>
<th>Condition Code Flag Evaluation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B (or BAL)</td>
<td>don’t care</td>
<td>unconditional (branch always)</td>
</tr>
<tr>
<td>BEQ</td>
<td>Z</td>
<td>equal</td>
</tr>
<tr>
<td>BNE</td>
<td>$\bar{Z}$</td>
<td>not equal</td>
</tr>
<tr>
<td>BCS / BHS</td>
<td>$C$</td>
<td>unsigned $\geq$</td>
</tr>
<tr>
<td>BCC / BLO</td>
<td>$\bar{C}$</td>
<td>unsigned $&lt;$</td>
</tr>
<tr>
<td>BMI</td>
<td>$N$</td>
<td>negative</td>
</tr>
<tr>
<td>BPL</td>
<td>$\bar{N}$</td>
<td>positive or zero</td>
</tr>
<tr>
<td>BVS</td>
<td>$V$</td>
<td>overflow</td>
</tr>
<tr>
<td>BVC</td>
<td>$\bar{V}$</td>
<td>no overflow</td>
</tr>
<tr>
<td>BHI</td>
<td>$C\bar{Z}$</td>
<td>unsigned $&gt;$</td>
</tr>
<tr>
<td>BLS</td>
<td>$\bar{C} + Z$</td>
<td>unsigned $\leq$</td>
</tr>
<tr>
<td>BGE</td>
<td>$NV + \bar{N}\bar{V}$</td>
<td>signed $\geq$</td>
</tr>
<tr>
<td>BLT</td>
<td>$N\bar{V} + \bar{N}V$</td>
<td>signed $&lt;$</td>
</tr>
<tr>
<td>BGT</td>
<td>$\bar{Z}(NV + \bar{N}\bar{V})$</td>
<td>signed $&gt;$</td>
</tr>
<tr>
<td>BLE</td>
<td>$Z + N\bar{V} + \bar{N}V$</td>
<td>signed $\leq$</td>
</tr>
</tbody>
</table>
Design and write an assembly language program to compute $n!$, where $n$ is a non-negative integer stored in register r0

$$n! = \prod_{k=1}^{n} k \quad \forall n \in \mathbb{N}$$

Algorithm to compute the factorial of some value

```assembly
result = 1
tmp = value

while (tmp > 1) {
    result = result * tmp
    tmp = tmp - 1
}
```
BLS – Branch if Lower or Same (unsigned ≤)

Use CMP to subtract 1 from r2

- If r2 < 1 there will be a borrow and the Carry flag will be clear
- If r2 = 1 the Zero flag will be set
- If r2 > 1 both Carry and Zero will be clear
Design and write an assembly language program that uses shift-and-add multiplication to multiply the value in r1 by the value in r2, storing the result in r0.

```
result = 0
while (b ≠ 0)
{
    b = b >> 1
    if (carry set) {
        result = result + a
    }
    a = a << 1
}
```
Exercise: Modify the program to avoid unnecessary iterations if \( a \) is equal to 0
Selection – General Form

- Execute one or more instructions only if some condition is satisfied

```java
if (r0 = 0) {
    r1 = 0
}
```

- Choose between two (or more) sets of instructions to execute

```java
if (r0 = 0) {
    r1 = 0
} else {
    r1 = r1 × r0
}
```
Selection – General Form

- Template for if-then construct

```c
if ( <condition> )
{
    <body>
}
<rest of program>
```

- Template for if-then-else construct

```c
if ( <condition> )
{
    <if body>
}
else {
    <else body>
}
<rest of program>
```
Design and write an assembly language program to compute the absolute value of an integer stored in register r1. The absolute value should be stored in r0.

```assembly
if (value < 0)
{
  value = 0 - value
}

start
  LDR r1, =-5 ; test with value = -5
  CMP r1, #0 ; if (value < 0)
  BGE endif1 ; {
  RSB r0, r1, #0 ;  result = 0 - value
  endif1 ; }

stop
  B stop
```
Design and write an assembly language program that evaluates the function \( \text{max}(a, b) \), where \( a \) and \( b \) are integers stored in r1 and r2 respectively. The result should be stored in r0.

```assembly
start
  LDR r1, =5      ; test with a = 5
  LDR r2, =6      ; test with b = 6
  CMP r1, r2      ; if (a ≥ b)
  BLT else1      ; {       
    MOV r0, r1    ; max = a       
  } else {        ; } else {       
    MOV r0, r2    ; max = b       
  } endif1       ; }       
```

Program 6.5 – \( \text{max}(a, b) \) (if-then-else)
### Iteration – General Form

- Execute a block of code, the loop body, multiple times
- Loop condition determines number of iterations (zero, one or more)
- Condition tested at beginning or end of loop

#### General Form

- **while (condition) {**
  - `<body>`
  - **}**

<table>
<thead>
<tr>
<th>Condition tested at start of loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body executed zero, one or more times</td>
</tr>
</tbody>
</table>

- **do {**
  - `<body>`
  - **} while (condition) **

<table>
<thead>
<tr>
<th>Condition tested at end of loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body executed one or more times</td>
</tr>
</tbody>
</table>
Iteration – General Form

- Template for while construct

```
<initialize>
while ( <condition> )
{
    <body>
}
<rest of program>
```

```
<initialize>
while CMP if necessary
    Bxx endwh on opposite<br/>&lt;condition&gt;
    <body>
    B while unconditionally
endwh <rest of program>
```

- Template for do-while construct

```
<initialize>
do {
    <body>
} while ( <condition> )
<rest of program>
```

```
<initialize>
do<br/>&lt;body&gt;
    CMP if necessary<br/>&lt;body&gt;
    Bxx do on &lt;condition&gt;
<rest of program>
```
The \( n \)th Fibonacci number is defined as follows:

\[
F_n = F_{n-2} + F_{n-1}
\]

with \( F_0 = 0 \) and \( F_1 = 1 \)

Design and write an assembly language program to compute the \( n \)th Fibonacci number, \( F_n \), where \( n \) is stored in r1.

Program 6.6 – \( n \)th Fibonacci Number (while)

```assembly
fn2 = 0
fn1 = 1
result = fn1
curr = 1
while (curr < n)
{
    tmp = result
    result = fn2 + fn1
    fn2 = fn1
    fn1 = tmp
    curr = curr + 1
}
```
- BCS – Branch if Carry Set (unsigned ≥)
- Use CMP to subtract r1 from r2
  - If r2 ≥ r1 there will be no borrow and the Carry flag will be set
  - If r2 < r1 there will be a borrow and the Carry flag will be clear
### while Construct Revisited

- A more efficient but less intuitive while construct

<table>
<thead>
<tr>
<th>Initialize</th>
<th>&lt;initialize&gt;</th>
</tr>
</thead>
</table>
| while (condition) { | }
| <body> | }
| <rest of program> | <initialize> |
| while (condition) { | }
| <body> | }
| <rest of program> |

<table>
<thead>
<tr>
<th>Revise</th>
<th>&lt;initialize&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>B testwh unconditionally</td>
<td>Bxx while on condition</td>
</tr>
<tr>
<td>testwh</td>
<td>CMP if necessary</td>
</tr>
<tr>
<td>endwh1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>wh1</td>
</tr>
<tr>
<td>BLS</td>
</tr>
<tr>
<td>MUL</td>
</tr>
<tr>
<td>SUBS</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>endwh1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Revised construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>B testwh1</td>
</tr>
<tr>
<td>wh1</td>
</tr>
<tr>
<td>SUBS</td>
</tr>
<tr>
<td>testwh1</td>
</tr>
<tr>
<td>BHI</td>
</tr>
</tbody>
</table>
### Compound Conditions

- **Logical conjunction**

  ```
  if (x ≥ 40 AND x < 50) {
    y = y + 1
  }
  ```

- **Test each condition and if any one fails, branch to end of if-then construct (or if they all succeed, execute the body)**

  ```
  ... ...
  CMP r1, #40 ; if (x ≥ 40
  BCC endif ; AND
  CMP r1, #50 ; x < 50)
  BCS endif ; {
  ADD r2, r2, #1 ; y = y + 1
  endif ; }
  ... ...
  ```
### Compound Conditions

- **Logical disjunction**

  ```
  if (x < 40 OR x ≥ 50)
  {
    z = z + 1
  }
  ```

- **Test each conditions and if they all fail, branch to end of if-then construct (or if any test succeeds, execute the body without testing further conditions)**

  ```
  CMP     r1, #40 ; if (x < 40
  BCC     then ; ||
  CMP     r1, #50 ; x ≥ 50)
  BCC     endif ; {
  then    ADD     r2, r2, #1 ; y = y + 1
  endif   ; }
  ```
Design and write an assembly language program that will convert the ASCII character stored in r0 to UPPER CASE, if the character is a lower case letter (a-z)

Can convert lower case to UPPER CASE by clearing bit 5 of the ASCII character code of a lower case letter

$\text{if (char} \geq \text{‘a’ AND char} \leq \text{‘z’)}$
$\{ \text{char = char . NOT(0x00000020)} \}$

Alternatively, subtract 0x20 from the ASCII code

$\text{if (char} \geq \text{‘a’ AND char} \leq \text{‘z’)}$
$\{ \text{char = char} - 0x20 \}$
Program 6.8 – Upper Case

- Algorithm ignores characters not in the range ['a', 'z']
- Option to use AND, BIC or SUB instructions to achieve same result
- Use of '#a', '#z' for convenience instead of #61 and #7a
  - Assembler converts ASCII symbol to character code

```assembly
start
  LDR r0, ='d'
  ; test with char = 'h'
  CMP r0, #'a'
  ; if (char ≥ 'a'
  BCC endif ; &&
  CMP r0, #'z'
  ; char ≤ 'z'
  BHI endif ; {
  AND r0, r0, #0xFFFFFFDF
  ; BIC r0, r0, #0x00000020 ; <alternative 1>
  ; SUB r0, r0, #0x20 ; <alternative 2>
  ; }
  endif
stop B stop
```
Branches can negatively effect performance

Program 6.4 – Absolute Value

```c
if (value < 0)
{
    value = 0 - value
}
```

Original assembly language program

```
start
    LDR    r1, =-5 ; test with value = -5
    CMP    r1, #0 ; if (value < 0)
    BGE    endif1 ; {
    RSB    r0, r1, #0 ; result = 0 - value
    endif1
    stop
    B      stop
```
Conditional Execution

- ARM instruction set allows any instruction to be executed conditionally
  - based on Condition Code Flags
  - exactly the same way as conditional branches

- Revised Program 6.4 - Absolute Value

```
start
  LDR r1, =-5 ; test with value = -5
  CMP r1, #0 ; if (value < 0)
  RSBLT r0, r1, #0 ; result = 0 - value
  } stop

stop   B    stop
```

- Reverse subtract (RSB) is only executed if the less-than condition is satisfied
Conditional Execution

- Program 6.5 – max($a, b$)

```plaintext
if (a ≥ b) {
    max = a
} else {
    max = b
}
```

- Revised Program 6.5 using conditional execution

```
start
LDR      r1, =5          ; test with $a = 5$
LDR      r2, =6          ; test with $b = 6$
CMP      r1, r2          ; if (a ≥ b) {
MOVGE    r0, r1          ;     max = a
} else {
MOVLT    r0, r2          ;     max = b
}
stop     B stop
```

- Either MOVGE or MOVLT will be executed