Flow control

- 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 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?

Program 6.1 – x^y

- Design and write an assembly language program to compute x^4 using repeated multiplication

```
            MOV     r0, #1          @ result = 1

            MUL     r0, r1, r0      @ result = result × value (value ^ 1)
            MUL     r0, r1, r0      @ result = result × value (value ^ 2)
            MUL     r0, r1, r0      @ result = result × value (value ^ 3)
            MUL     r0, r1, r0      @ result = result × value (value ^ 4)
```

  - 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

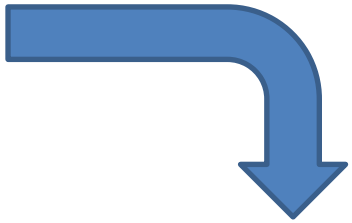
```
            MOV     r0, #1          @ result = 1

do y times:
            MUL     r0, r1, r0      @ result = result × value
            repeat
```

Not valid assembly language!!

Program 6.1a – x^y

Iteration

```
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
            B       while           @ }
endwh:

stop:       B       stop
```

Iteration

Pseudo-Code

```
while:
            BEQ     endwh       @ while (y ≠ 0) {
            MUL     r0, r1, r0  @  result = result × x
            SUBS    r2, r2, #1  @  y = y - 1
            B       while       @ }
endwh:
```

- **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

Program 6.1b - x^y

```
if (y = 0) {
    result = 1
}
else {
    result = x
    if (y > 1) {
        y = y - 1
        do {
            result = result × x
            y = y - 1
        } while (y ≠ 0)
    }
}
```

Selection

Selection

Iteration

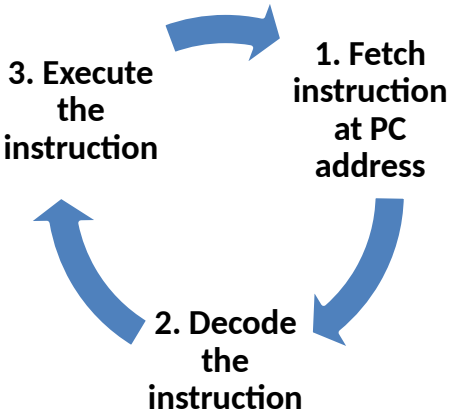
Program 6.1b - x^y

```
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 × x
            SUBS    r2, r2, #1      @    y = y - 1
            BNE     do1             @   } while (y ≠ 0)
endif2:                             @  }
endif1:                             @ }

stop:       B       stop
```

Comments – not assembled

Program Counter Modifying Instructions

- 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 **Z**ero condition code flag is set)

B – Unconditional Branch Instruction

- Unconditional Branch

```
            B       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 decode execute cycle will be from new Program Counter address

Labels and Branch Target Offsets

```
while:      BEQ     endwh           @ while (y ≠ 0) {
            MUL     r0, r1, r0      @  result = result × x
            SUBS    r2, r2, #1      @  y = y - 1
            B       while           @ }
endwh:
```

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

branch target offset = ((*label address* - *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

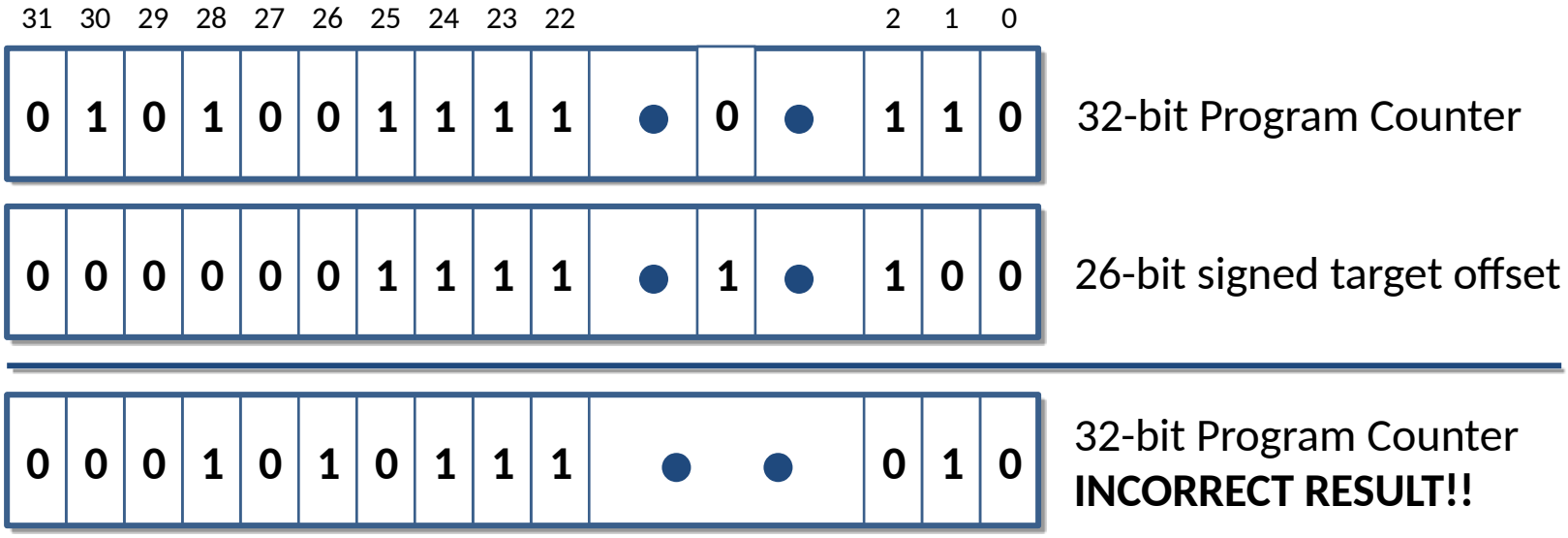
Executing Branch Instructions

PC ← PC + (*branch target offset* × 4)

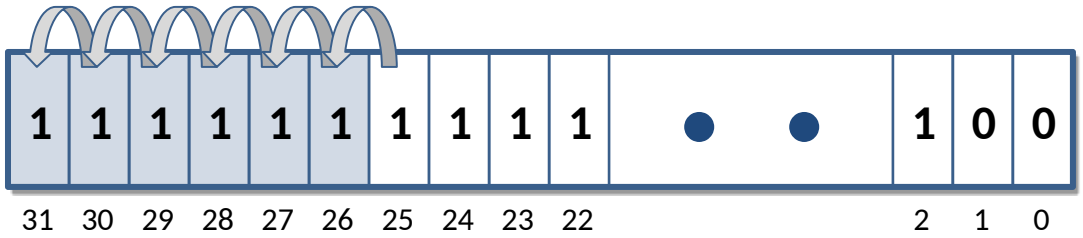
- Next fetch in fetch decode execute cycle will fetch the instruction at the new PC address

- 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

Sign Extension



- Must **sign extend** the 26-bit offset by copying the value of bit 25 into bits 26 to 31 (2's Complement system)



Labels

```
while:      BEQ     endwh           @ while (y ≠ 0) {
            MUL     r0, r1, r0      @  result = result × x
            SUBS    r2, r2, #1      @  y = y - 1
            B       while           @ }
endwh:
```

- 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

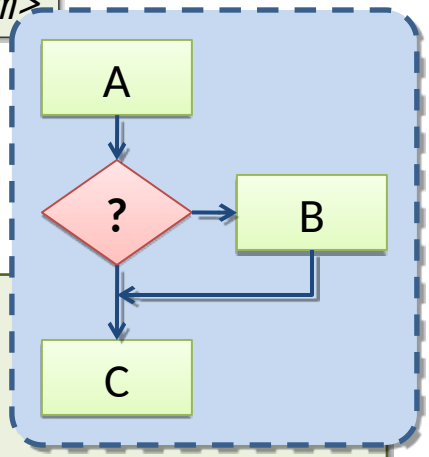
Bxx – Conditional Branch Instructions

- 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

Bxx – Conditional Branch Instructions

- Simple **selection** construct …

```
if (a ≠ b) {
    a = b
}
<rest of program>
```



- In ARM assembly language
  - assume a ↔ r0, b ↔ r1

```
            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

```
            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 - CoMPare Instruction

- Using SUBtract to compare two values, the result has to be stored somewhere, even though it is not needed

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

```
            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

Branch Instruction	Condition Code Flag Evaluation	Description
B (or BAL)	don't care	unconditional (branch always)
BEQ	Z	equal
BNE	$\bar{Z}$	not equal
BCS / BHS	C	unsigned ≥
BCC / BLO	$\bar{C}$	unsigned <
BMI	N	negative
BPL	$\bar{N}$	positive or zero
BVS	V	overflow
BVC	$\bar{V}$	no overflow
BHI	$C\bar{Z}$	unsigned >
BLS	$\bar{C} + Z$	unsigned ≤
BGE	$NV + \bar{N}\bar{V}$	signed ≥
BLT	$N\bar{V} + \bar{N}V$	signed <
BGT	$\bar{Z}(NV + \bar{N}\bar{V})$	signed >
BLE	$Z + N\bar{V} + \bar{N}V$	signed ≤

Program 6.2 - Factorial

- 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*

```
result = 1
tmp = value

while (tmp > 1) {
    result = result × tmp
    tmp = tmp - 1
}
```

Program 6.2 - Factorial

```
start:
            LDR     r1, =6          @ value = 6

            MOV     r0, #1          @ result = 1
            MOVS    r2, r1          @ tmp = value

wh1:        CMP     r2, #1          @ while (tmp > 1)
            BLS     endwh1          @ {
            MUL     r0, r2, r0      @  result = result × tmp
            SUBS    r2, r2, #1      @  tmp = tmp - 1
            B       wh1             @ }
endwh1:

stop:       B       stop
```

- BLS – Branch if Lower or Same (unsigned ≤)
- Use CMP to subtract 1 from r2
  - If r2 < 1 there will be a borrow and the **C**arry flag will be clear
  - If r2 = 1 the **Z**ero flag will be set
  - If r2 > 1 both **C**arry and **Z**ero will be clear

Program 6.3 – Shift And Add Multiplication

- 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
}
```

Program 6.3 – Shift And Add Multiplication

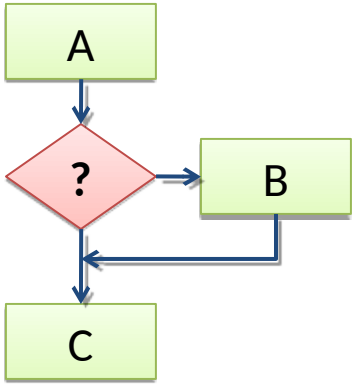
```
start:
            LDR     r1, =10                 @ test with a = 10
            LDR     r2, =6                  @ test with b = 6

            MOV     r0, #0                  @ result = 0

wh1:
            CMP     r2, #0                  @ while (b ≠ 0)
            BEQ     endwh1                  @ {
            MOVS    r2, r2, LSR #1          @  b = b >> 1
            BCC     endif1                  @  if (carry set) {
            ADD     r0, r0, r1              @   result = result + a
endif1:                                     @  }
            MOV     r1, r1, LSL #1          @  a = a << 1
            B       wh1                     @ }
endwh1:

stop:       B       stop
```

- Exercise: Modify the program to avoid unnecessary iterations if *a* is equal to 0

Selection – General Form

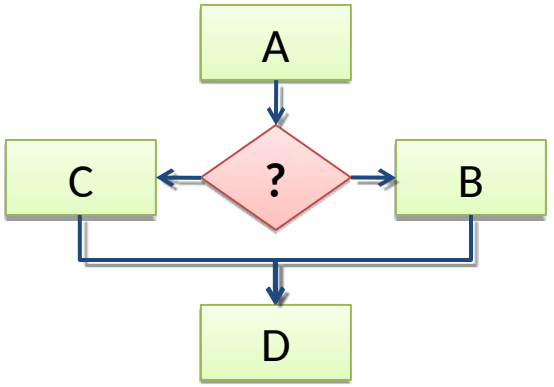
- Execute one or more instructions only if some condition is satisfied

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



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

```
if (r0 = 0) {
    r1 = 0
}
else {
    r1 = r1 × r0
}
```



Selection – General Form

- Template for if-then construct

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

```
                CMP     if necessary
                Bxx     endif on opposite <condition>
                <body>
endif:
                <rest of program>
```

- Template for if-then-else construct

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

```
                CMP     if necessary
                Bxx     else on opposite <condition>
                <if body>
                B       endif unconditionally
else:
                <else body>
endif:
                <rest of program>
```

Program 6.4 – Absolute Value (if-then)

- 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.

```
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
```

Program 6.5 – max(a, b) (if-then-else)

- Design and write an assembly language program that evaluates the function max(*a*, *b*), where *a* and *b* are integers stored in r1 and r2 respectively. The result should be stored in r0.

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

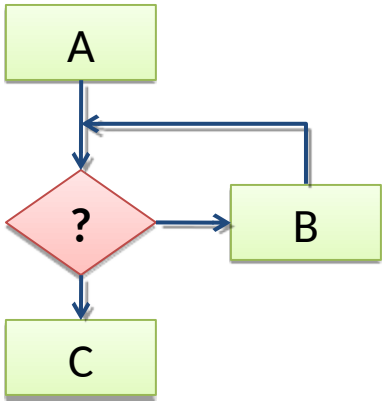
```
start:
            LDR     r1, =5                  @ test with a = 5
            LDR     r2, =6                  @ test with b = 6

            CMP     r1, r2                  @ if (a ≥ b)
            BLT     else1                   @ {
            MOV     r0, r1                  @  max = a
            B       endif1                  @ } else {
else1:      MOV     r0, r2                  @  max = b
endif1:                                     @ }
```

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

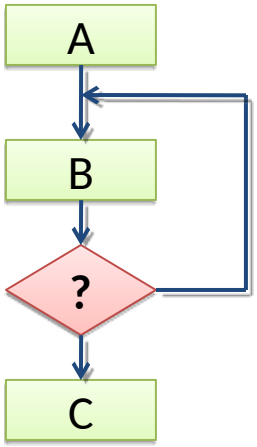
```
while ( <condition> ) {
    <body>
}
```

Condition tested at start of loop
Body executed zero, one or more times




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

Condition tested at end of loop
Body executed one or more times



Iteration – General Form

- Template for while construct

```
<initialize>

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

```
            <initialize>

while:      CMP     if necessary
            Bxx     endwh opposite <condition>
            <body>
            B       while unconditionally
endwh:      <rest of program>
```

- Template for do-while construct

```
<initialize>

do {
    <body>
} while ( <condition>
)

<rest of program>
```

```
            <initialize>

do:
            <body>
            CMP     if necessary
            Bxx     do on <condition>
            <rest of program>
```

Program 6.6 – n^{th} Fibonacci Number (while)

- 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.

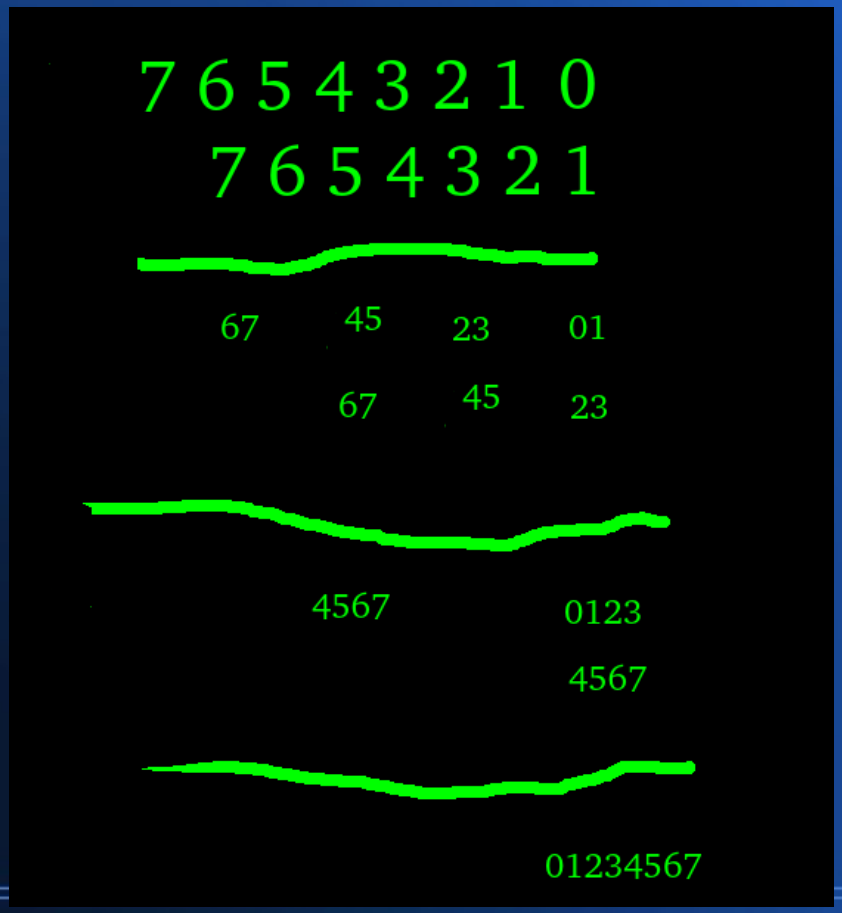
```
fn2 = 0
fn1 = 1
result = fn1
curr = 1
while (curr < n)
{
     tmp = result
     result = fn2 + fn1
     fn2 = fn1
     fn1 = tmp
     curr = curr + 1
}
```

Program 6.6 – n^{th} Fibonacci Number (while)

```
start:
            LDR     r1, =4          @ test with n = 4

            MOV     r3, #0          @ fn2 = 0
            MOV     r4, #1          @ fn1 = 1
            MOV     r0, r4          @ result = fn1
            MOV     r2, #1          @ curr = 1
wh1:        CMP     r2, r1          @ while (curr < n)
            BCS     endwh1          @ {
            MOV     r5, r0          @  tmp = result
            ADD     r0, r3, r4      @  result = fn2 + fn1
            MOV     r3, r4          @  fn2 = fn1
            MOV     r4, r5          @  fn1 = tmp
            ADD     r2, r2, #1      @  curr = curr + 1
            B       wh1             @ }
endwh1:

stop:       B       stop
```

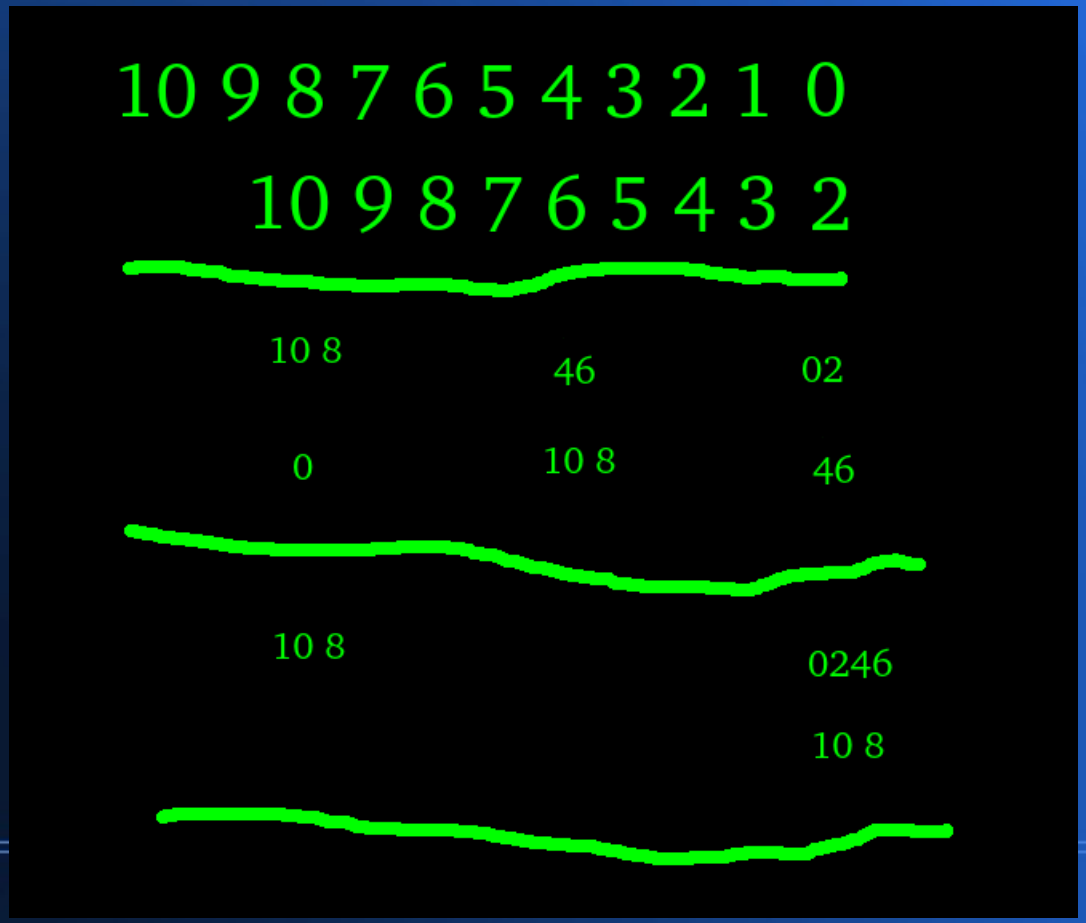
- BCS – Branch if Carry Set (unsigned ≥)
- Use CMP to subtract r1 from r2
  - If r2 ≥ r1 there will be no borrow and the **C**arry flag will be set
  - If r2 < r1 there will be a borrow and the **C**arry flag will be clear

```
EOR     r1, r1, r1, LSR #1      ; tmp = tmp EOR tmp << 1
EOR     r1, r1, r1, LSR #2      ; tmp = tmp EOR tmp << 2
EOR     r1, r1, r1, LSR #4      ; tmp = tmp EOR tmp << 4
```



Hamming Check bit c0 in bit position 0 is calculated to produce even parity for bits 2, 4, 6, 8, and 10

```
@ Generate check bit c0

EOR     R2, R0, R0, LSR #2      @ Generate c0 parity bit using parity tree
EOR     R2, R2, R2, LSR #4      @ ... second iteration ...
EOR     R2, R2, R2, LSR #8      @ ... final iteration
```



Program 6.7 – Parity (do-while)

- Modify Program 5.6 to replace the three EOR instructions with an iterative loop using a do-while construct
- Original Program 5.6

```
start:
            LDR     r0, =0x16

            MOV     r1, r0                          @ tmp = value

            EOR     r1, r1, r1, LSR #1          @ tmp = tmp EOR tmp << 1
            EOR     r1, r1, r1, LSR #2          @ tmp = tmp EOR tmp << 2
            EOR     r1, r1, r1, LSR #4          @ tmp = tmp EOR tmp << 4

            AND     r1, r1, #0x00000001         @ clear all but LSB
            ORR     r0, r0, r1, LSL #7          @ set parity bit in MSB pos

stop:       B       stop
```

Program 6.7 – Parity (do-while)

```
start:
            LDR     r0, =0x16

            MOV     r2, #1                      @ shift = 1
            MOV     r1, r0                      @ tmp = value

do:                                             @ do {
            EOR     r1, r1, r1, LSR r2          @  tmp = tmp EOR tmp <<
shift:
            MOV     r2, r2, LSL #1              @  shift = shift × 2
            CMP     r2, #4                      @ } while (shift ≤ 4)
            BLS     do                          @

            AND     r1, r1, #0x00000001         @ clear all but LSB
            ORR     r0, r0, r1, LSL #7          @ set parity bit in MSB pos

stop:       B       stop
```

- do-while construct is appropriate as the algorithm calls for one or more iterations (never zero)
- Perform logical shift left by 1, 2 and 4 bit positions (2^0, 2^1 and 2^2 bit positions)

while Construct Revisited

- A more efficient but less intuitive while construct

```
<initialize>

while ( <condition> ) {
    <body>
}

<rest of program>
```

```
            <initialize>

            B       testwh unconditionally
while:      <body>
testwh:     CMP     if necessary
            Bxx     while on <condition>
            <rest of program>
```

```
wh1:        CMP     r2, #1          @ while (tmp > 1)
            BLS     endwh1          @ {
            MUL     r0, r2, r0      @  result = result × value
            SUBS    r2, r2, #1      @  tmp = tmp - 1
            B       wh1             @ }
endwh1:
```

Original construct

```
            B       testwh1         @ while (tmp > 1) {
wh1:        MUL     r0, r2, r0      @  result = result × value
            SUBS    r2, r2, #1      @  tmp = tmp - 1
testwh1:    CMP     r2, #1          @ }
            BHI     wh1             @
endwh1:
```

Revised construct

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:                              @ }
            ...     ...
```

Program 6.8 – Upper Case

- 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

```
if (char ≥ ‘a’ AND char ≤ ‘z’)
{
    char = char . NOT(0x00000020)
}
```

- Alternatively, subtract 0x20 from the ASCII code

```
if (char ≥ ‘a’ AND char ≤ ‘z’)
{
    char = char – 0x20
}
```

Program 6.8 – Upper Case

```
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         @  char = char . 0xFFFFFFDF
@           BIC     r0, r0, #0x00000020         @  <alternative 1>
@           SUB     r0, r0, #0x20               @  <alternative 2>
endif:                                          @ }

stop:       B       stop
```

- 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

Conditional Execution

- Branches can negatively effect performance
- Program 6.4 – Absolute Value

```
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:       B       stop
```

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

Conditional Execution

- Program 6.5 – max(*a*, *b*)

```
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

Structured Programming

Every branching/looping pattern used in a piece of code adds a cognitive burden. Goto (ie branches) lets you create infinite such patterns, rapidly overwhelming people's ability to understand the code. Therefore, way back in the mists of time people (especially Dijkstra, 1968, Go to Statements Considered Harmful) settled on a standard set of control structures to replace most uses of goto, and that set has grown only slowly over time.

- Functions and procedures
- If/then/else and (sometimes) switch/case or case/in.
- While/for/do/repeat loops
- break, continue, and return statements
- Iterators, generators, list comprehensions (in some languages)

Spaghetti code

Spaghetti code is a pejorative phrase for source code that has a complex and tangled control structure, especially one using many GOTO statements, exceptions, threads, or other "unstructured" branching constructs. It is named such because program flow is conceptually like a bowl of spaghetti, i.e. twisted and tangled. Spaghetti code can be caused by several factors, such as continuous modifications by several people with different programming styles over a long life cycle.

Structured programming greatly decreases the incidence of spaghetti Code. When using the many forms of assembly language (and also the underlying machine code) the danger of writing spaghetti code is especially great.

Dijkstra argued that goto statements were harmful because they complicate two important and related tasks:

Proving that a program fragment is correct

Describing what a program has done so far.

As a simple example, consider a loop:

```
while (n != 0)
        { /* do something */ }
```

Because this loop has n != 0 as its condition, we know that when the loop terminates normally, n is zero. If there is a break statement inside the loop, it is possible that n will be nonzero when the loop terminates. This possibility shows how break statements can make it harder to understand what programs do.

How do we deal with the possibility of break statements in our programs when we analyse them? In one of two ways:

(1) by including whatever conditions apply at the time the break is executed among the conditions that might apply when the loop terminates,

(2) by ensuring that the loop-termination conditions are true when the break is executed.

Suppose we rewrite our while loop

```
top: if (n == 0)
        goto bottom;
        /* do something */
        goto top;
bottom:
```

On the surface, this loop is no harder to analyze than the earlier version--after all, it behaves in the same way. However, when we added the label "bottom," we opened the possibility that a goto statement anywhere else in the program might jump there.

So now, instead of merely having to inspect the loop to understand what it does, we must inspect the entire program.

Sequence, Selection and Iteration

A sequence is one of the basic logic structures in computer programming. In a sequence structure, an action, or event, leads to the next ordered action in a predetermined order. The sequence can contain any number of actions, but no actions can be skipped in the sequence.

A selection (also called a decision) is also one of the basic logic structures in computer programming. In a selection structure, a question is asked, and depending on the answer, the program takes one of two courses of action, after which the program moves on to the next event. This structure is sometimes referred to as an if-then-else because it directs the program to perform in this way: If Condition A is True then perform Action X else perform Action Y.

An iteration is a single pass through a group/set of instructions. Most programs contain loops of instructions that are executed over and over again. The computer repeatedly executes the loop, iterating through the loop.