Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

05 – Flow Control

CS1021 – Introduction to Computing I

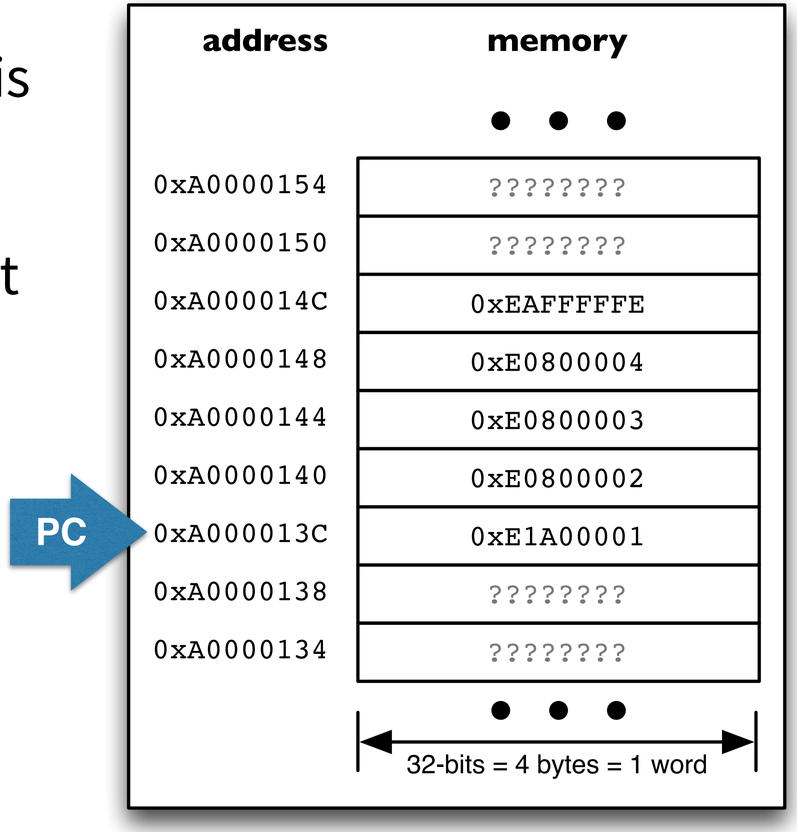
Dr Jonathan Dukes | jdukes@tcd.ie
School of Computer Science and Statistics

Program 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**



Example – 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
```

For illustration purposes only! Not valid ARM Assembly Language Syntax!!

jdukes@tcd.ie

Example – x^y


```
result = 1
while (y != 0) {
  result = result × x
  y = y – 1
}
```

Iteration

```
        LDR   r1, =3                ; test with x = 3
        LDR   r2, =4                ; test with y = 4
        MOV   r0, #1                ; result = 1

while
        CMP   r2, #0
        BEQ   endwh                 ; while (y != 0) {
        MUL   r0, r1, r0            ;  result = result × x
        SUB   r2, r2, #1            ;  y = y - 1
        B     while                 ; }
endwh

stop    B     stop
```

BEQ – **B**ranch if **EQ**ual

CMP Instruction

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

Subtraction allows us to determine equality (= or ≠) or inequality (< ≤ ≥ >)

Don't care about absolute value of result (i.e. don't care ***by how much*** *x* is greater than *y*, only whether it is or not.)

CMP always sets the Condition Code Flags – no need for **CMPS**

```
        CMP   r2, #0                ; subtract 0 from r2, ignoring result but
                                    ; updating the CC flags
        BEQ   endwh                 ; if the result was zero then branch to endwh
        ...   ...                   ; otherwise (if result was not zero) then keep
                                    ; going (with sequential instruction path)
endwh
```

Pseudo-code


```
while
        CMP   r2, #0
        BEQ   endwh                 ; while (y != 0) {
        MUL   r0, r1, r0            ;  result = result * x
        SUB   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 – informally structured comments

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

Example – Absolute Value

Design and write an assembly language program to compute the absolute value of an integer stored in register r1. The result should also be stored in r1.

```
if (value < 0)
{
   value = 0 – value
}
```

```
        LDR   r1, =-5                     ; test with value = -5

        CMP   r1, #0                      ; if (value < 0)
        BGE   endifneg                    ; {
        RSB   r1, r1, #0                  ;  value = 0 - value
endifneg                                  ; }
```

RSB – Reverse SuBtract
r = b - a instead of **r = a - b**

Branch instructions

By default, the processor increments the Program Counter (PC) (by 4 bytes – 1 instruction) to “point” to the next sequential instruction in memory

causing the **sequential** path to be followed

Using a **branch** instruction, we can modify the value in the Program Counter to “point” to an instruction of our choosing

breaking the pattern of **sequential** execution

branch instructions can be

unconditional – always update the PC (i.e. always branch)

conditional – update the PC only if some condition is met
(condition is based on Condition Code Flags, e.g. if the Zero flag is set)

B – Unconditional Branch Instruction (and Labels)


```
        B       label                   ; Branch unconditionally to label

        ...     ...                     ; ...
        ...     ...                     ; more instructions
        ...     ...                     ; ...

label   some    instruction             ; more instructions
        ...     ...                     ; ...
```

Labels …

must be unique (within a .s file)

can contain UPPER and lower case letters, numerals and the underscore _ character

are case sensitive (mylabel is not the same label as MyLabel)

must <u>not</u> begin with a numeral

jdukes@tcd.ie

Bxx – Conditional Branch Instruction

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 ("runtime")

- 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

Example – Max

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
}
```

BLT – Branch if Less Than

i.e. from a preceding **CMP a,b** branch if a < b

```
        LDR   r1, =5              ; test with a = 5
        LDR   r2, =6              ; test with b = 6

        CMP   r1, r2              ; if (a ≥ b)
        BLT   elsmaxb             ; {
        MOV   r0, r1              ;  max = a
        B     endab               ; }
elsmaxb                           ; else {
        MOV   r0, r2              ;  max = b
endab                             ; }
```


jdukes@tcd.ie

Description	Symbol	Java	Instruction	Mnemonic
Equality				
equal	=	==	BEQ	EQual
not equal	≠	!=	BNE	Not Equal
Inequality (unsigned values)				
less than	<	<	BLO (or BCC)	LOwer
less than or equal	≤	<=	BLS	Lower or Same
greater than or equal	≥	>=	BHS (or BCS)	Higher or Same
greater than	>	>	BHI	HIgher
Inequality (signed values)				
less than	<	<	BLT	Less Than
less than or equal	≤	<=	BLE	Less than or Equal
greater than or equal	≥	>=	BGE	Greater than or Equal
greater than	>	>	BGT	Greater Than
Flags				
Negative Set			BMI	MInus
Negative Clear			BPL	PLus
Carry Set			BCS (or BHS)	Carry Set
Carry Clear			BCC (or BLO)	Carry Clear
Overflow Set			BVS	oVerflow Set
Overflow Clear			BVC	oVerflow Clear
Zero Set			BEQ	EQual
Zero Clear			BNE	Not Equal

Flow Control Cheat Sheet

ARM Conditional Branch Instructions

Description	Symbol	Java	Instruction	Mnemonic
Equality				
equal	=	==	BEQ	**EQ**ual
not equal	≠	!=	BNE	**N**ot **E**qual
Inequality (unsigned values)				
less than	<	<	BLO (or BCC)	**LO**wer
less than or equal	≤	<=	BLS	**L**ower or **S**ame
greater than or equal	≥	>=	BHS (or BCS)	**H**igher or **S**ame
greater than	>	>	BHI	**HI**gher
Inequality (signed values)				
less than	<	<	BLT	**L**ess **T**han
less than or equal	≤	<=	BLE	**L**ess than or **E**qual
greater than or equal	≥	>=	BGE	**G**reater than or **E**qual
greater than	>	>	BGT	**G**reater **T**han
Flags				
Negative Set			BMI	**MI**nus
Negative Clear			BPL	**PL**us
Carry Set			BCS (or BHS)	**C**arry **S**et
Carry Clear			BCC (or BLO)	**C**arry **C**lear
Overflow Set			BVS	o**V**erflow **S**et
Overflow Clear			BVC	o**V**erflow **C**lear
Zero Set			BEQ	**EQ**ual
Zero Clear			BNE	**N**ot **E**qual

Equality and Inequality Mnemonics are based on a previous execution of a compare (`CMP`) instruction of the form `CMP Rx, Ry`. For example, `BLE label` will branch to `label` if *Rx* is less than or equal to *Ry*.

Pseudo Code Examples

Pseudo Code		ARM Assembly Language
`if (x <= y)` `{` `   x = x + 1;` `}`	*assume x and y are <u>signed</u> values*	`        CMP   Rx, Ry` `        BGT   label` `        ADD   Rx, Rx, #1` `label`
`if (x < y)` `{` `   z = x;` `}` `else {` `   z = y;` `}`	*assume x and y are <u>unsigned</u> values*	`        CMP   Rx, Ry` `        BHS   label1` `        MOV   Rz, Rx` `        B     label2` `label1` `        MOV   Rz, Ry` `label2`
`while (x > 2)` `{` `   y = x * y;` `   x = x - 1;` `}`	*assume x and y are <u>unsigned</u> values*	`label1  CMP   Rx, #2` `        BLS   label2` `        MUL   Ry, Rx, Ry` `        SUB   Rx, Rx, #1` `        B     label1` `label2`

ARM Flow Control “Cheat Sheet” available on Blackboard

Not available in exams, but you will have access to more formal documentation, including a description of each conditional branch instruction

jdukes@tcd.ie

Example – Factorial

Design and write an assembly language program to compute *n*!, where *n* is a non-negative integer stored in register r0

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

```
result = 1
tmp = value

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

Example – Factorial


```
       LDR  r1, =6              ; test value = 6

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

whmul  CMP  r2, #1              ; while (tmp > 1)
       BLS  endwhmul            ; {
       MUL  r0, r2, r0          ;  result = result × tmp
       SUB  r2, r2, #1          ;  tmp = tmp - 1
       B    whmul               ; }
endwhmul
```

BLS – Branch if Lower or Same (unsigned ≤) – evaluates Carry and Zero flags

Use CMP to subtract 1 from r2

- **If r2 < 1 there will be a "borrow-in" so the Carry flag will be clear (*R2 + TC(1) would set C flag*)**
- **If r2 = 1 the Zero flag will be set**
- If r2 > 1 Carry will be set and Zero will be clear

So … branch if C set or Z clear

Selection - General Form

Template for if-then construct

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

```
        CMP   variables or constants in <condition>
        Bxx   endiflabel on opposite <condition>
        <body>
endiflabel
        <rest of program>
```

Template for if-then-else construct

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

```
        CMP   variables or constants in <condition>
        Bxx   elselabel on opposite <condition>
        <if body>
        B     endiflabel unconditionally
elselabel
        <else body>
endiflabel
        <rest of program>
```

Iteration – General Form

Template for while construct

```
<initialize>

while ( <condition> )
{

    <body>

}
<rest of program>
```

```
          <initialize>

whilelabel
          CMP    variables or constants in <condition>
          Bxx    endwhlabel on opposite <condition>
          <body>
          B      whilelabel unconditionally
endwhlabel
          <rest of program>
```

Template for do-while construct

```
<initialize>

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

<rest of program>
```

```
          <initialize>

dolabel
          <body>
          CMP   variables or constants in <condition>
          Bxx   dolabel on <condition>

          <rest of program>
```

Example – n^{th} Fibonacci Number

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 register R1.

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

Example – n^{th} Fibonacci Number


```
start
        LDR   r1, =4                ; test with n = 4

        MOV   r3, #0                ; fn1 = 0
        MOV   r0, #1                ; fn = 1
        MOV   r2, #1                ; curr = 1
whn     CMP   r2, r1                ; while (curr < n)
        BHS   endwhn                ; {
        ADD   r2, r2, #1            ;  curr = curr + 1
        MOV   r4, r0                ;  tmp = fn
        ADD   r0, r0, r3            ;  fn = fn + fn1
        MOV   r3, r4                ;  fn1 = tmp
        B     whn                   ; }
endwhn
```

Compound Conditions – AND


```
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
        BLO    endif                ;  AND
        CMP    r1, #50              ;  x < 50)
        BHS    endif                ; {
        ADD    r2, r2, #1           ;  y = y + 1
endif                               ; }
        ...    ...
```

Compound Conditions – OR


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

Test each condition 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
        BLO     then                    ;   ||
        CMP     r1, #50                 ;  x ≥ 50)
        BLO     endif                   ; {
then    ADD     r2, r2, #1              ;  y = y + 1
endif                                   ; }
        ...     ...
```

Example – 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 subtracting 0x20 from the ASCII code

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

Example – Upper Case


```
      LDR  r0, ='d'                ; test with char = 'h'

      CMP  r0, #'a'                ; if (char ≥ 'a'
      BLO  notLcAlpha              ;  &&
      CMP  r0, #'z'                ;  char ≤ 'z')
      BHI  notLcAlpha              ; {
      SUB  r0, r0, #0x20           ;  char = char – 0x20
notLcAlpha                         ; }
```

Algorithm ignores characters not in the range [‘a’, ‘z’]

Note use of #’a’, #’z’ for convenience instead of #61 and #7A

Assembler converts ASCII symbol to character code

jdukes@tcd.ie