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

4.1 – Flow Control

CSU11021 – Introduction to Computing I

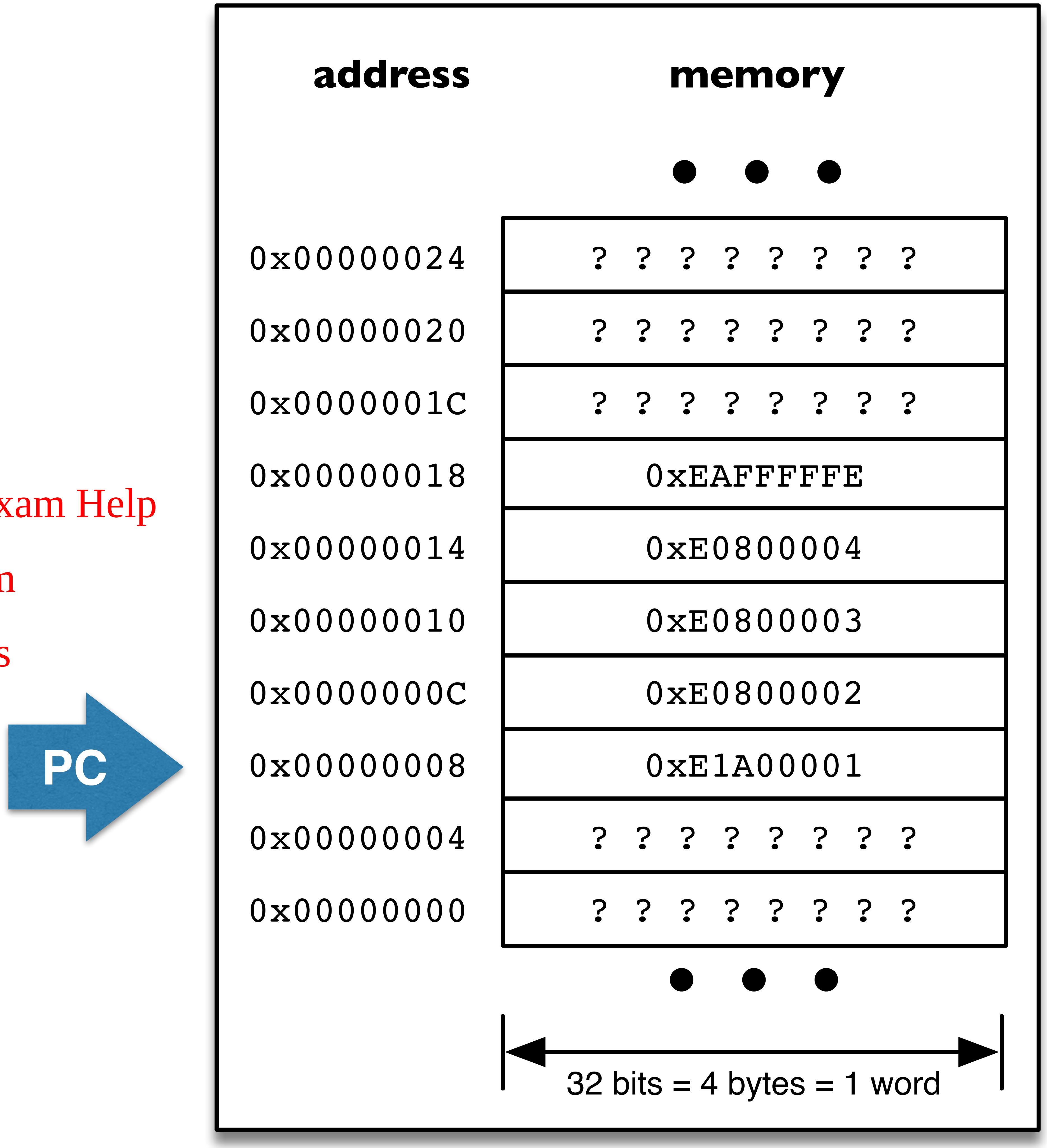
Dr Jonathan Dukes | jdukes@tcd.ie
School of Computer Science and Statistics

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**



address	memory
	● ● ●
0x00000024	? ? ? ? ? ? ? ?
0x00000020	? ? ? ? ? ? ? ?
0x0000001C	? ? ? ? ? ? ? ?
0x00000018	0xEAFFFFFE
0x00000014	0xE0800004
0x00000010	0xE0800003
0x0000000C	0xE0800002
0x00000008	0xE1A00001
0x00000004	? ? ? ? ? ? ? ?
0x00000000	? ? ? ? ? ? ? ?
	● ● ●

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, R0, R1      @ result = result × value
    repeat
```

Example – x^y

```
x = 3;
y = 4;
result = 1;
while (y != 0) {
  result = result * x;
  y = y - 1;
}
```

```
  MOV     R0, #1          @ result = 1
While:
  CMP     R2, #0
  BEQ     EndWh           @ while (y != 0) {
  MUL     R0, R0, R1      @  result = result × x
  SUB     R2, R2, #1      @  y = y - 1
  B       While           @ }
EndWh:
```

CMP Instruction

CMP (CoMPare) instruction performs a subtraction **without storing the result of the subtraction**

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

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

Properties of the result are remembered by the processor

```
  CMP   R2, #0           @ Subtract 0 from r2, remembering the properties
                         @   of the result but not the value of the result
  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

```
  MOV     R0, #1          @ result = 1
While:
  CMP     R2, #0
  BEQ     EndWh           @ while (y != 0) {
  MUL     R0, R0, R1      @  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 be stored in R0.

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

```
    MOV     R0, R1          @ result = value
    CMP     R0, #0          @ if (result < 0)
    BGE     EndIfNeg        @ {
    RSB     R0, R0, #0      @   result = 0 - result
EndIfNeg:                   @ }
```

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

4.2 – Branch Instructions

CSU11021 – Introduction to Computing I

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

Branch instructions

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

causing the **sequential** path to be followed

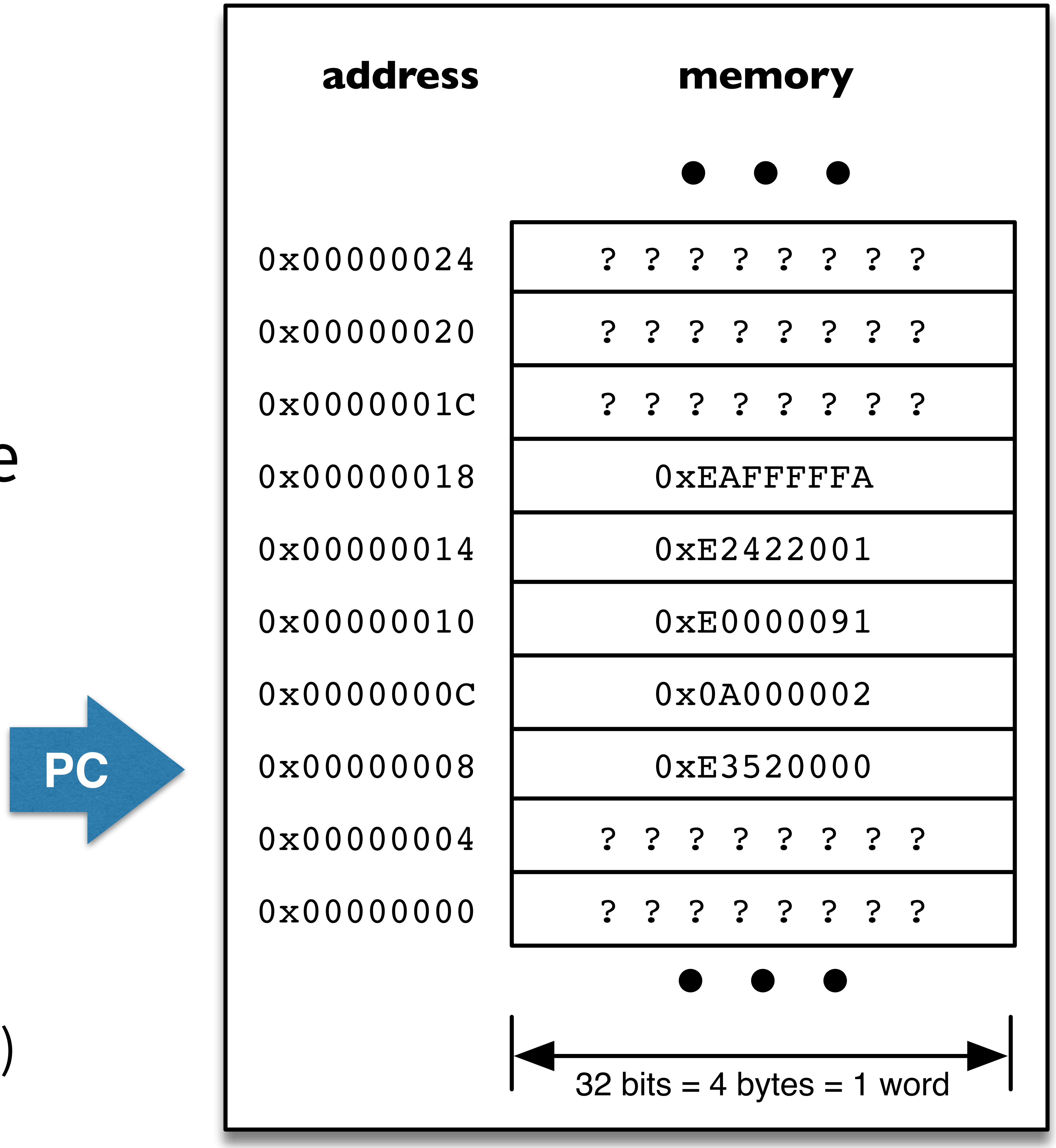
Using a **branch** instruction, we can modify the value in the PC to “point” to an instruction of our choosing

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, based on the preceding CoMParison (`CMP`)



address	memory
	• • •
0x00000024	? ? ? ? ? ? ? ?
0x00000020	? ? ? ? ? ? ? ?
0x0000001C	? ? ? ? ? ? ? ?
0x00000018	0xEAFFFFFA
0x00000014	0xE2422001
0x00000010	0xE0000091
0x0000000C	0x0A000002
0x00000008	0xE3520000
0x00000004	? ? ? ? ? ? ? ?
0x00000000	? ? ? ? ? ? ? ?
	• • •

B – Unconditional Branch Instruction (and Labels)

```
        B       MyLabel         @ Branch unconditionally to label MyLabel

        ...     ...             @ ...
        ...     ...             @ more instructions
        ...     ...             @ ...

MyLabel:
        <some instruction>      @ more instructions
        ...     ...             @ ...
```

Labels …

when you define them, must end with a colon:

must be unique (within a .s file) – only the first definition is used

must begin with a letter, . (dot) or_ (underscore) but not a numeral

can contain UPPER and lower case letters, numerals, or _ (underscores)

are case sensitive (so mylabel is not MyLabel)

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

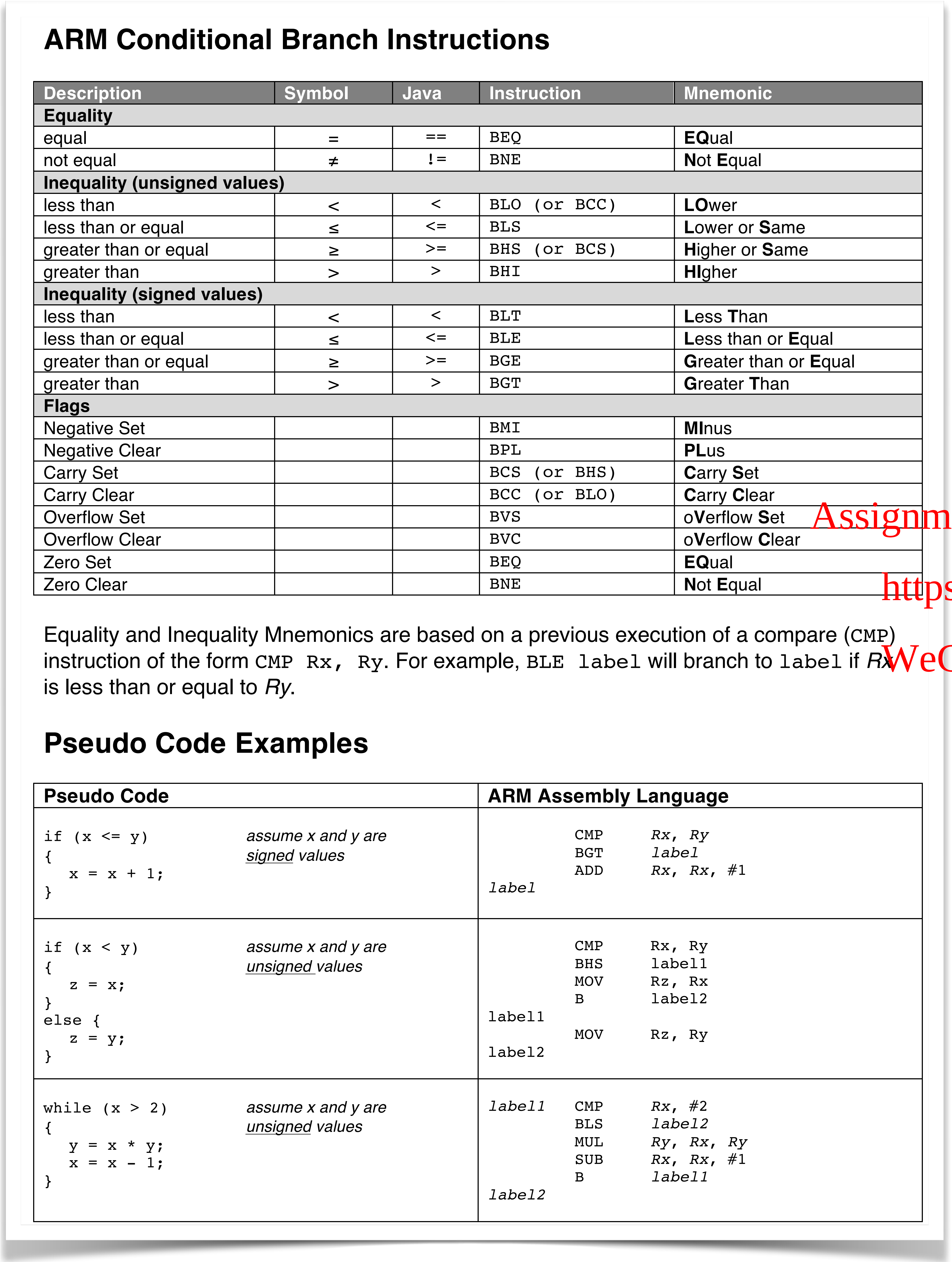
Design and write an assembly language program that evaluates the function max(a, b), where a and b are integers stored in R1 and R2 respectively. The result should be stored in R0.

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

```
  CMP     R1, R2              @ if (a >= b)
  BLT     ElseMaxB            @ {
  MOV     R0, R1              @   max = a
  B       EndMax              @ }
ElseMaxB:                     @ else {
  MOV     R0, R2              @   max = b
EndMax:                       @ }
```

Description	Symbol	Java	Instruction	Mnemonic
Equality				
equal	=	==	BEQ	EQual
not equal	≠	!=	BNE	Not Equal
Inequality (unsigned values)				
less than	<	<	BLO	LOwer
less than or equal	≤	<=	BLS	Lower or Same
greater than or equal	≥	>=	BHS	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

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

Example – Factorial

Design and write an assembly language program to compute *n*!, where *n* is a non-negative integer stored in register R1. Store your result in R0.

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

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

Example – Factorial

```
    MOV    R0, #1         @ result = 1
    MOV    R2, R1         @ tmp = n
WhileMul:
    CMP    R2, #1         @ while (tmp > 1)
    BLS    EndWhMul       @ {
    MUL    R0, R0, R2     @   result = result * tmp
    SUB    R2, R2, #1     @   tmp = tmp - 1
    B      WhileMul       @ }
EndWhMul:
```

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

4.3 – Flow control templates

CSU11021 – Introduction to Computing I

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

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
EndWhLabel:
        <rest of program>
```

Template for a for construct

```
for ( <initialize>, <condition>, <update> ) {

    <body>

}
<rest of program>
```

```
        <initialize>

ForLabel:
        CMP     variables or constants in <condition>
        Bxx     EndForLabel on opposite <condition>
        <body>
        <update>
        B       ForLabel
EndForLabel:
        <rest of program>
```

Iteration – General Form

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 – Fibonacci Numbers

Fibonacci numbers are defined as follows:

$$F_n = F_{n-2} + F_{n-1}$$

with $F_0 = 0$ and $F_1 = 1$

```
fn2 = 0
fn1 = 1
fn = 1
curr = 2
while (curr < n)
{
    fn2 = fn1
    fn1 = fn
    curr = curr + 1
    fn = fn1 + fn2
}
```

Design and write an assembly language program to compute a Fibonacci number, F_n, where n is stored in register R1.

Example – Fibonacci Numbers

```
  @ Calculate Fibonacci number Fn, where n is stored in R1
  @ Store the result in R0

  MOV   R4, #0         @ fn2 = 0
  MOV   R5, #1         @ fn1 = 1
  MOV   R0, #1         @ fn = 1
  MOV   R6, #2         @ curr = 2
WhileFib:
  CMP   R6, R1         @ while (curr < n)
  BHS   EndWhFib       @ {
  MOV   R4, R5         @   fn2 = fn1
  MOV   R5, R0         @   fn1 = fn
  ADD   R6, R6, #1     @   curr = curr + 1
  ADD   R0, R5, R4     @   fn = fn1 + fn2
  B     WhileFib       @ }
EndWhFib:
```

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)

You 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

```
    CMP    R0, #'a'        @ if (char >= 'a'
    BLO    EndIfLc         @   AND
    CMP    R0, #'z'        @   char <= 'z')
    BHI    EndIfLc         @ {
    SUB    R0, R0, #0x20   @   char = char - 0x20
EndIfLc:                   @ }
```

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

Note use of #’a’, #’z’ for convenience instead of #0x61 and #0x7A

- Assembler converts ASCII symbol to character code