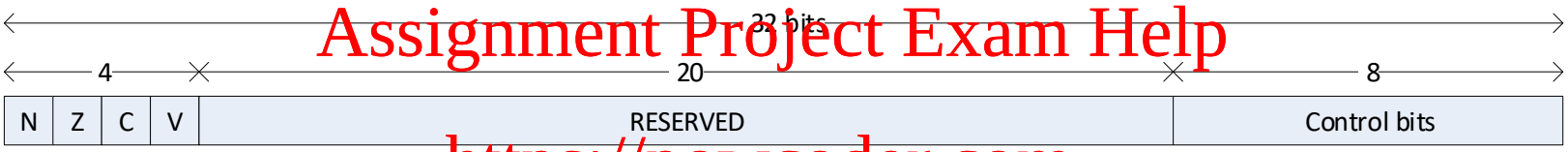
Program Flow Control

- 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



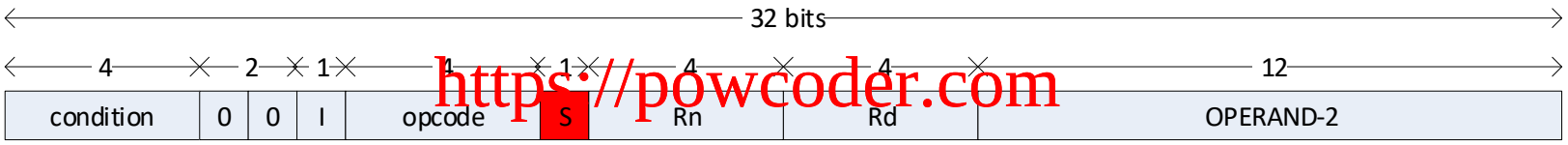
Current Program Status Register (CPSR)

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

N – negative	N = MSB(result)
Z – zero	Z = 1 if result == 0, Z = 0 if result != 0
C – carry	will discuss carry and overflow later
V – overflow	

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 (eg. 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

Description	Symbol	C/C++/Java	Instruction	Mnemonic
Branch equal or not equal				
equal	=	==	**BEQ**	EQual
not equal	≠	!=	**BNE**	Not Equal
unsigned branches				
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
signed branches				
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
branch on 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

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      …
```

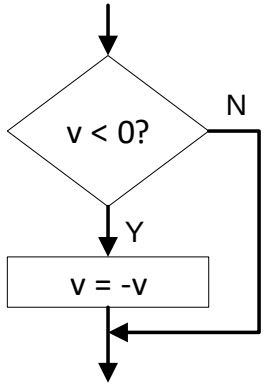

- labels start in column 1 (otherwise column 1 should be empty except for comments)
- labels must NOT begin with a digit (0..9)
- labels can contain UPPER and lower case letters, digits and _ (underscore)
- labels are case sensitive (mylabel is not the same as MyLabel)
- labels must be unique within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

```
if (v < 0) {   // brackets NOT necessary
    v = -v;    // in this case since v = -v;
}              // is a single statement
```

flowchart

- flowchart has a diamond shaped decision boxes with **N**o and **Y**es 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

```
        CMP   R1, #0          ;  v < 0 ?
        BGE   L1              ; >= ( opposite condition to < in pseudo code)
        RSB   R1, R1, #0      ; v = 0 - v
L1      …
```

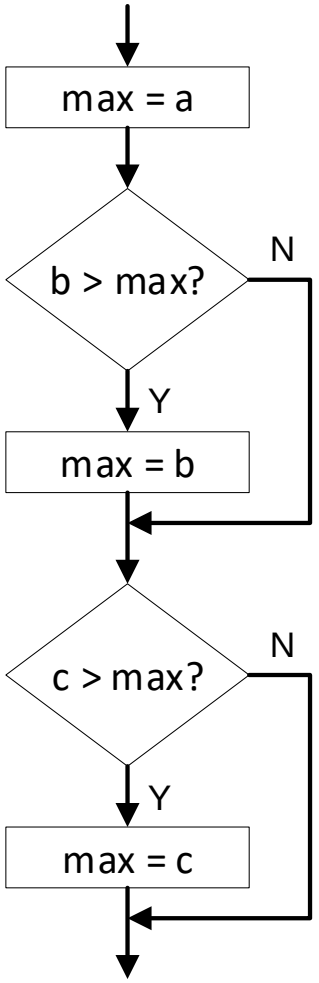

- note the use of RSB (reverse subtract)
- signed branch since v is a signed integer
- RSB executed if v < 0

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



flowchart

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

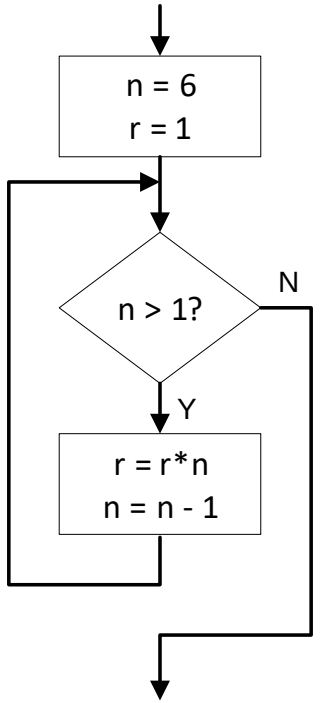
```
        MOV   R0, R1          ; max = a
        CMP   R2, R0          ; b > max ?
        BLE   L1              ; <= (opposite condition to > in pseudo code)
        MOV   R0, R2          ; max = b
L1      CMP   R3, R0          ; c < max ?
        BLE   L2              ; <= (opposite condition to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

  $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$

- 0, 1, 1, 2, 3, 5, 8, …


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

pseudo code

~~edge case~~

n <= 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

IF … THEN … ELSE

```
if (x < 9) {                    // assume x in R1
    x = x + 1;                  // execute if condition TRUE
} else {
    x = 0;                      // execute if condition FALSE
}
```

```
    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

```
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)
L1      ADD   R3, R3, #1  ; z = z + 1 (s1 executed if condition FALSE)
L2
```

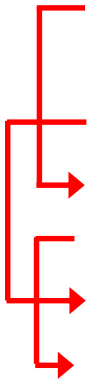

- signed branches (assume x is a signed integer)

Conditional OR

```
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)
L3
```


- 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)
- Overflow (V)

discuss in more detail now

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

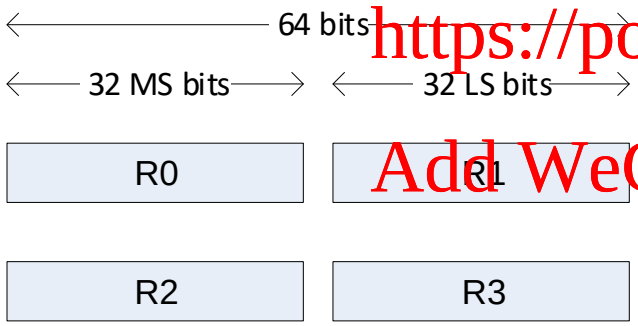
hex	unsigned
0x70	112
+ 0xA0	+ 160
1 0x10	272 = CARRY(256) + 16

CARRY

- 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

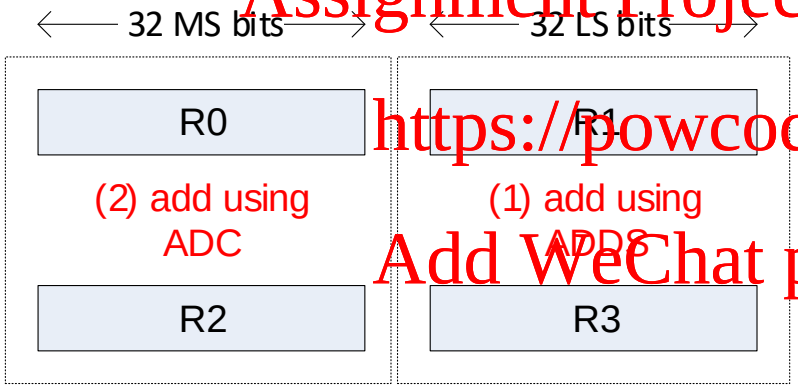


- R0:R1 and R2:R3 used to store 64 bit integers
- R0 and R2 contain the 32 most significant bits
- R1 and R3 contain the 32 least significant bits

- 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

hex	unsigned	
0x70	112	
- 0xA0	- 160	
1 0xD0	-48	= BORROW(-256) + 208 (0xD0)

BORROW

- 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

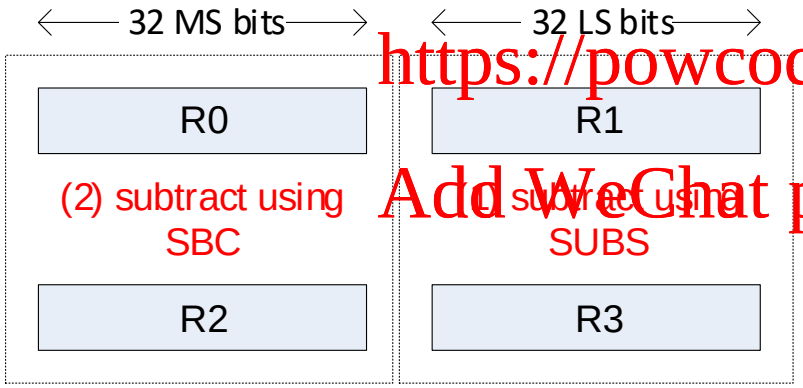
hex	unsigned	
0x70	112	
+ 0x60	- 160	
0 0xD0	-48	= BORROW(-256) + 208 (0xD0)

CARRY

- CARRY = 0 shown in RED, therefore 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)



- ALSO RSC reverse subtract with CARRY
- dst = src2 – src1 + CARRY - 1

```
SUBS    R1, R1, R3      ; subtract least significant bits and set CARRY
SBC     R0, R0, R2      ; subtract most significant bits with CARRY
```

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

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 if MSB(a) == MSB(b) && MSB(r) != MSB(a)
- for subtraction r = a – b

  V = 1 if MSB(a) != MSB(b) && MSB(r) != MSB(a)

Example Condition Code Flags after CMP instruction

- compare 0x70000000 with 0xA0000000

```
LDR     R0, =0x70000000     ; R0 = 0x70000000
LDR     R1, =0xA0000000     ; R1 = 0xA0000000
CMP     R0, R1              ; set flags on result of 0x70000000 – 0xA0000000
```


- CMP always sets the four condition code flags

- N = 1
- Z = 0
- C = 0 (BORROW = 1)
- V = 1

	hex	unsigned	signed
	0x70000000	1,879,048,192	1,879,048,192
-	0xA0000000	- 2,684,354,560	- -1,610,612,736
0	0xD0000000	-805,306,368	3,489,660,928

CARRY

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

Example Condition Code Flags after CMP instruction …

- unsigned interpretation

  0xD0000000 = 3,489,660,928 (1,879,048,192 - 2,684,354,560 = -805,306,368)

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

- signed interpretation

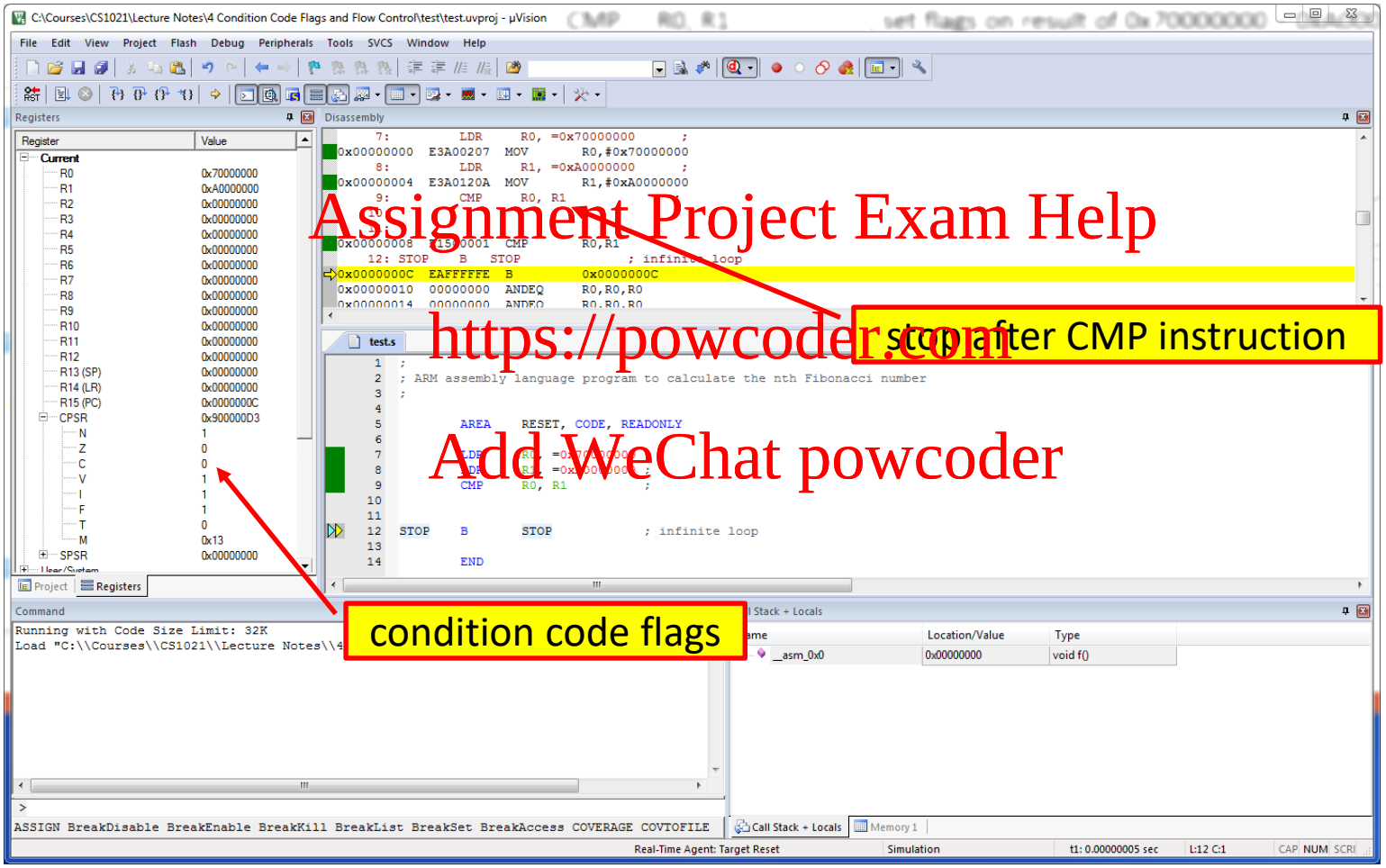
  0xD0000000 = -805,306,368 (1,879,048,192 + 1,610,612,736 = 3,489,660,928)

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

- unsigned branches - test CARRY and ZERO flags

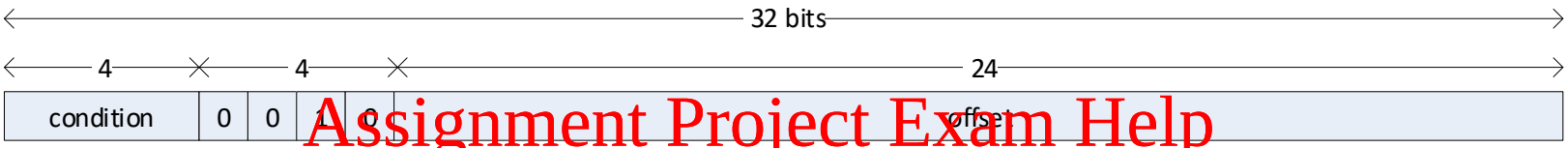
- signed branches test - OVERFLOW and ZERO flags

Check using uVision



Branch Instructions in Machine Code

- branch instructions are encoded in machine code as follows



- 4 bit condition field determines which condition is tested


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

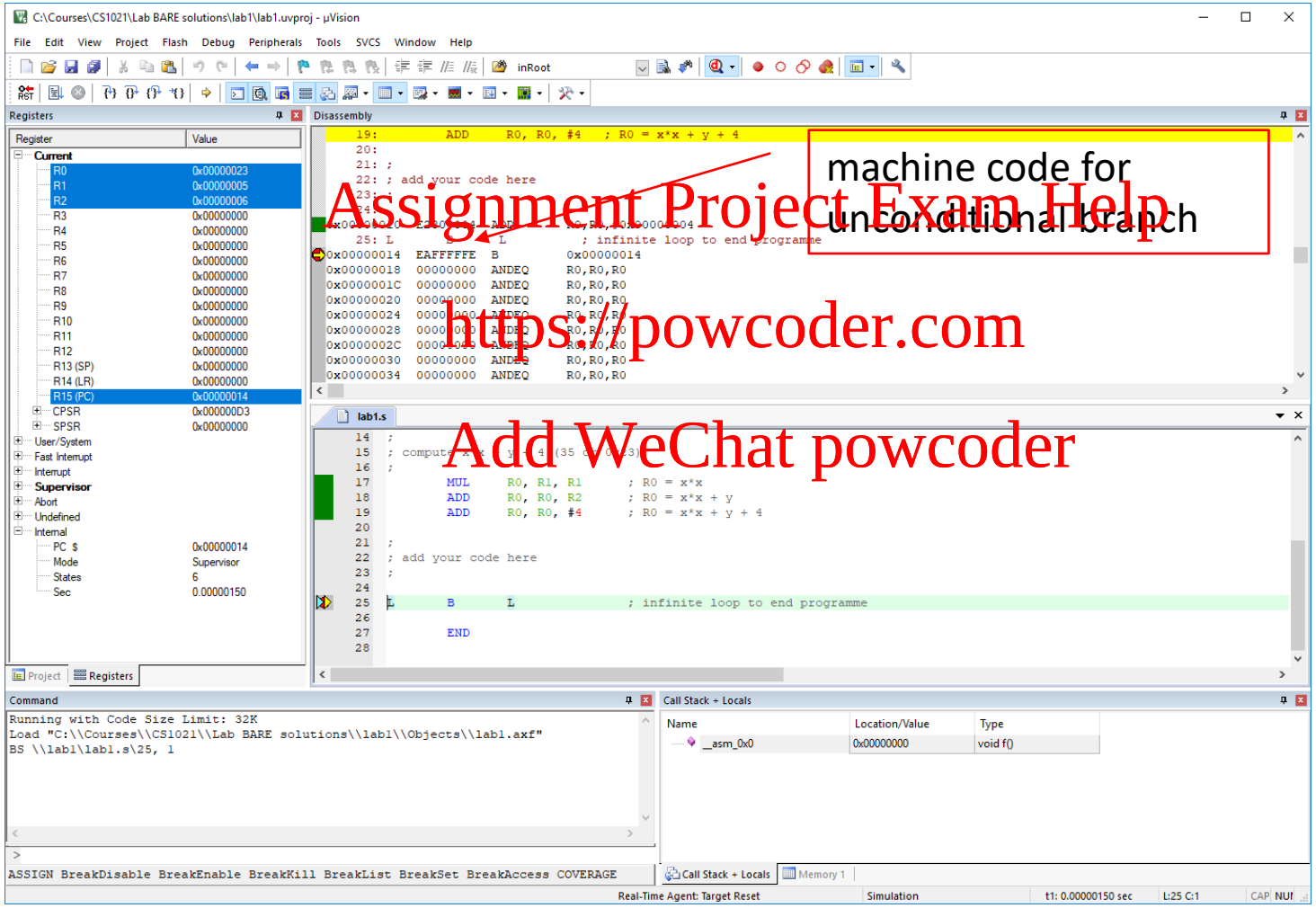

- since instructions are 4 bytes, offset is multiplied by 4 (equivalent to appending two least significant zero bits) before being signed extended and added to the PC
- due to the fetch-decode-execute pipeline, the PC has increased by 8 by the time this addition takes place (assembler takes this into account when calculating offsets)

C set = higher or same
C clear = lower

Branch Condition field

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

Unconditional Branch Instruction Example



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 => 0xFFFFFFFE
- multiply by 4 (by appending two least significant zero bits) => 0xFFFFFFF8
- PC = PC + 8 + 0xFFFFFFF8
- PC = 0x00000014 + 0x00000008 + 0xFFFFFFF8 => 0x00000014
- unconditional branch to itself [an infinite loop]