Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

Chapter 6
Flow Control in Assembly

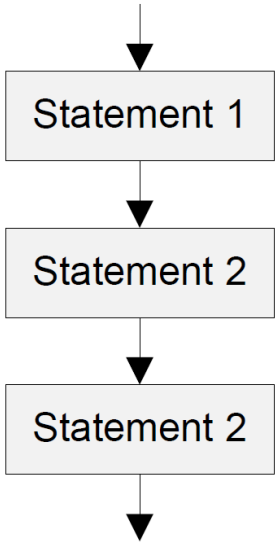
Z. Gu

Fall 2025

Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

Three Control Structures

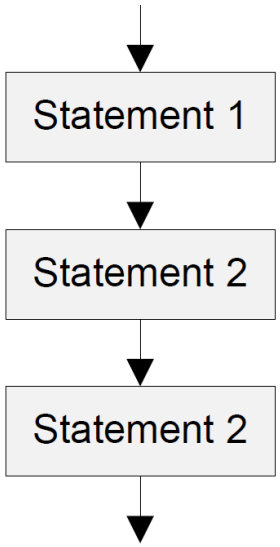
- Sequence Structure
  - Computer executes statements (instructions), one after another, in the order listed in the program



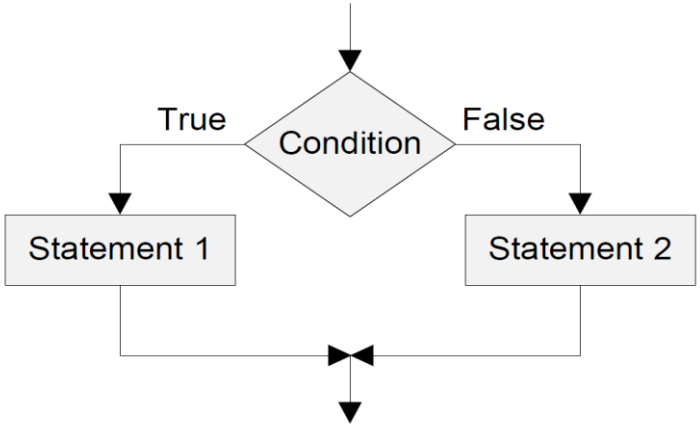
Sequence Structure

Three Control Structures

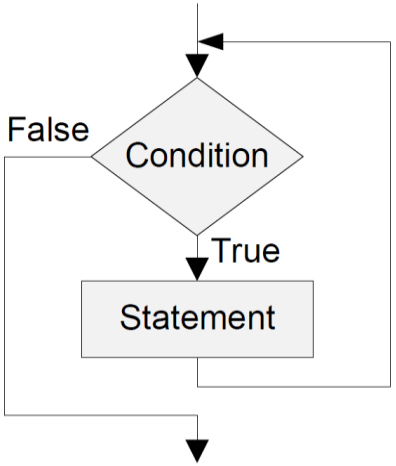
- **Selection Structure**
  - **If-then-else**
- **Loop Structure**
  - **while loop**
  - **for loop**



Sequence Structure



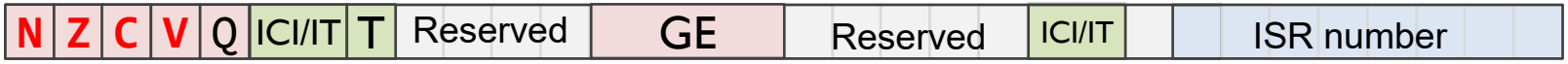
Selection Structure



Loop Structure

Review: Condition Flags

Program Status Register (PSR)



- **Negative** bit
  - `N = 1` if most significant bit of result is `1`
- **Zero** bit
  - `Z = 1` if all bits of result are `0`
- **Carry** bit
  - For unsigned addition, `C = 1` if carry takes place
  - For unsigned subtraction, `C = 0` (carry = not borrow) if borrow takes place
  - For shift/rotation, C = last bit shifted out
- **oVerflow** bit
  - `V = 1` if adding 2 same-signed numbers produces a result with the opposite sign
    - Positive + Positive = Negative, or
    - Negative + negative = Positive
  - Non-arithmetic operations does not touch V bit, such as `MOV`, `AND`, `LSL`, `MUL`

Updating Condition Flags

- Method 1: append with "**S**"
  - `ADD r0,r1,r2` → `ADDS r0,r1,r2`
  - `SUB r0,r1,r2` → `SUBS r0, r1, r2`

- Method 2: using compare instructions

Instruction	Brief description	Flags
CMP	Compare	N,Z,C,V
CMN	Compare Negative	N,Z,C,V
TEQ	Test Equivalence	N,Z,C
TST	Test	N,Z,C

Updating Condition Flags

Instruction	Operands	Brief description	Flags
CMP	Rn, Op2	Compare	N,Z,C,V
CMN	Rn, Op2	Compare Negative	N,Z,C,V
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C

- **Update the status flags**
  - No need to add S bit.
  - No need to specify destination register.
- Operations are:
  - `CMP` operand1 - operand2, but result not written
  - `CMN` operand1 + operand2, but result not written
  - `TST` operand1 & operand2, but result not written
  - `TEQ` operand1 ^ operand2, but result not written
- Examples:
  - `CMP r0, r1`
  - `TST r2, #5`

Updating Condition Flags: CMP and CMN

```
CMP Rn, Operand2
CMN Rn, Operand2
```

- Update N, Z, C and V according to the result
- **CMP** subtracts Operand2 from Rn.
  - Same as SUBS, except result is discarded.
- **CMN** adds Operand2 to Rn.
  - Same as ADDS, except result is discarded.

Example of CMP

$$f(x) = |x|$$

```
        Area absolute, CODE, READONLY
        EXPORT __main
        ENTRY

__main PROC
        CMP     r1, #0
        RSBLT   r1, r1, #0 ; r1 = -r1 if r1 < 0

done    B done      ; deadloop

        ENDP
        END
```

RSBLT r1, r1, #0:: conditional execution of the RSB instruction with the condition code LT. If the condition LT (less than) is true, then set r1=0−r1

Updating Condition Flags: TST and TEQ

```
TST Rn, Operand2  ; Bitwise AND
TEQ Rn, Operand2  ; Bitwise Exclusive OR
```

- Update N and Z according to the result
- Can update C during the calculation of Operand2
- Do not affect V
- TST performs bitwise AND on Rn and Operand2.
  - Same as ANDS, except result is discarded.
- TEQ performs bitwise Exclusive OR on Rn and Operand2.
  - Same as EORS, except result is discarded.

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

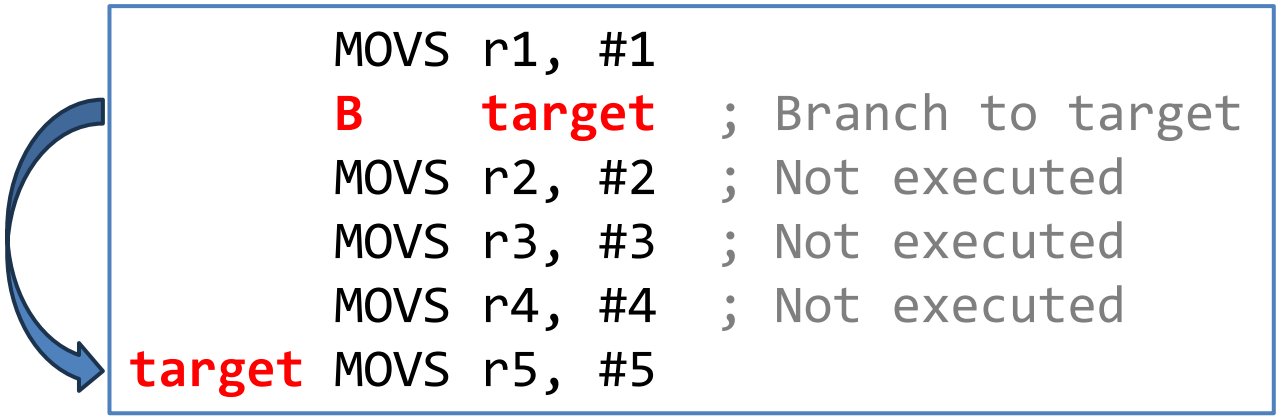
x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Unconditional Branch Instructions

Instruction	Operands	Brief description
B	`label`	Branch
BL	`label`	Branch with Link
BLX	`Rm`	Branch indirect with Link
BX	`Rm`	Branch indirect

- *B label*
  - cause a branch to label.
- *BL label*
  - copy the address of the next instruction into `r14` (lr, the link register), and
  - cause a branch to label.
- *BX Rm*
  - branch to the address held in Rm
- *BLX Rm*:
  - copy the address of the next instruction into `r14` (lr, the link register) and
  - branch to the address held in Rm

Unconditional Branch Instructions: A Simple Example



```
        MOVS r1, #1
        B    target  ; Branch to target
        MOVS r2, #2  ; Not executed
        MOVS r3, #3  ; Not executed
        MOVS r4, #4  ; Not executed
target  MOVS r5, #5
```

- A label marks the location of an instruction
- Labels helps human to read the code
- In machine program, labels are converted to numeric offsets by assembler

Condition Codes

- The possible condition codes are listed below:

Suffix	Description	Flags tested
EQ	EQual	Z==1
NE	Not Equal	Z==0
CS/HS	Unsigned Higher or Same	C==1
CC/LO	Unsigned LOwer	C==0
MI	MInus (Negative)	N==1
PL	PLus (Positive or Zero)	N==0
VS	oVerflow Set	V==1
VC	oVerflow Clear	V==0
HI	Unsigned HIgher	C==1 and Z==0
LS	Unsigned Lower or Same	C==0 or Z==1
GE	Signed Greater or Equal	N==V
LT	Signed Less Than	N!=V
GT	Signed Greater Than	Z==0 and N==V
LE	Signed Less than or Equal	Z==1 or N!=V
AL	ALways	

Note AL is the default and does not need to be specified

Signed Greater or Equal (N == V)

`CMP r0, r1`

perform subtraction `r0 – r1`, without saving the result

	N = 0	N = 1
V = 0	• No overflow, implying the result is correct. • The result is non-negative, • Thus `r0 – r1 ≥ 0`, i.e., `r0 ≥ r1`	• No overflow, implying the result is correct. • The result is negative. • Thus `r0 – r1 < 0`, i.e., `r0 < r1`
V = 1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as non-negative and in fact it should be negative. • Thus `r0 – r1 < 0` in reality, i.e., `r0 < r1`	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as negative and in fact it should be non-negative. • Thus `r0 – r1 ≥ 0` in reality., i.e. `r0 ≥ r1`

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed Greater or Equal (N == V)

`CMP r0, r1`

perform subtraction `r0 – r1`, without saving the result

	N = 0	N = 1
V = 0	• r0 = +7 (00111) • r1 = +3 (00011) • r0 – r1 = +4 (00100); • result non-negative and no signed overflow, so N=0, V=0 ⇒ GE holds	• r0 = +3 (00011) • r1 = +7 (00111) • r0 – r1 = –4 (11100) • result negative with no overflow, so N=1, V=0 ⇒ LT holds
V = 1	• r0 = –10 (10110) • r1 = +7 (00111) • r0 – r1 = –17, outside range [–16,+15]; result is 00111 (decimal 7), whose sign bit is 0 so N=0, but signed overflow occurs so V=1 ⇒ LT holds	• r0 = +10 (01010) • r1 = –7 (11001) • r0 – r1 = +17, outside range [–16,+15]; result is 10001 (decimal –15), whose sign bit is 1 so N=1, but signed overflow occurs so V=1 ⇒ GE holds

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed Greater or Equal (N == V)

`CMP r0, r1`

perform subtraction `r0 – r1`, without saving the result

	N = 0	N = 1
V = 0	r0 ≥ r1	r0 < r1
V = 1	r0 < r1	r0 ≥ r1

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed Greater or Equal (N == V)

`CMP r0, r1`

perform subtraction `r0 - r1`, without saving the result

	N = 0	N = 1
V = 0	1	0
V = 1	0	1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed vs. Unsigned Comparison

Compare	Signed	Unsigned
>	GT	HI
≥	GE	HS
<	LT	LO
≤	LE	LS
==	EQ	
≠	NE	

Conditional codes applied to branch instructions

Compare	Signed	Unsigned
>	BGT	BHI
>=	BGE	BHS
<	BLT	BLO
<=	BLE	BLS
==	BEQ	
!=	BNE	

Branch Instructions

	Instruction	Description	Flags tested
Unconditional Branch	`B label`	Branch to label	
Conditional Branch	`BEQ label`	Branch if **EQ**ual	`Z = 1`
	`BNE label`	Branch if **N**ot **E**qual	`Z = 0`
	`BCS/BHS label`	Branch if unsigned **H**igher or **S**ame	`C = 1`
	`BCC/BLO label`	Branch if unsigned **LO**wer	`C = 0`
	`BMI label`	Branch if **MI**nus (Negative)	`N = 1`
	`BPL label`	Branch if **PL**us (Positive or Zero)	`N = 0`
	`BVS label`	Branch if o**V**erflow **S**et	`V = 1`
	`BVC label`	Branch if o**V**erflow **C**lear	`V = 0`
	`BHI label`	Branch if unsigned **HI**gher	`C = 1 & Z = 0`
	`BLS label`	Branch if unsigned **L**ower or **S**ame	`C = 0 or Z = 1`
	`BGE label`	Branch if signed **G**reater or **E**qual	`N = V`
	`BLT label`	Branch if signed **L**ess **T**han	`N != V`
	`BGT label`	Branch if signed **G**reater **T**han	`Z = 0 & N = V`
	`BLE label`	Branch if signed **L**ess than or **E**qual	`Z = 1 or N = !V`

Number Interpretation

Which is greater?

`0xFFFFFFFF` or `0x00000001`

- If they represent signed numbers, the latter is greater (**`1 > -1`**).
- If they represent unsigned numbers, the former is greater (**`4294967295 > 1`**).

Which is Greater: 0xFFFFFFFF or 0x00000001?

It's **software's responsibility** to tell computer how to interpret data:

- If written in C, declare the signed *vs* unsigned variable
- If written in Assembly, use signed *vs* unsigned branch instructions

```
signed int x, y ;
x = -1;
y = 1;
if (x > y)
   ...
```

```
    MOV r6, #0xFFFFFFFF
    MOV r5, #0x00000001
    CMP  r5, r6
    BLE  Then_Clause
    ...
```

BLE: Branch if less than or equal, signed ≤

```
unsigned int x, y ;
x = 4294967295;
y = 1;
if (x > y)
   ...
```

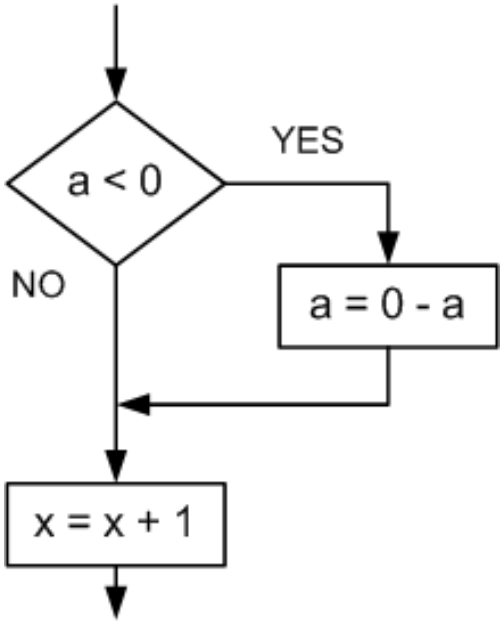
```
    MOV r6, #0xFFFFFFFF
    MOV r5, #0x00000001
    CMP  r5, r6
    BLS  Then_Clause
    ...
```

BLS: Branch if lower or same, unsigned ≤

If-then Statement

C Program

```
// a is signed integer
if (a < 0 ) {
  a = 0 – a;
}
x = x + 1;
```



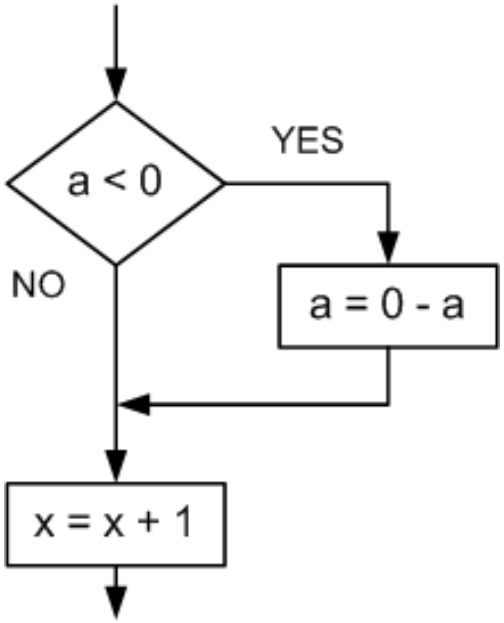
Implementation 1:

```
        ; r1 = a (signed integer), r2 = x
        CMP r1, #0          ; Compare a with 0
        BGE endif           ; Go to endif if a ≥ 0
then    RSB r1, r1, #0      ; a = - a;
endif   ADD r2, r2, #1      ; x = x + 1
```

If-then Statement

C Program

```
// a is signed integer
if (a < 0 ) {
  a = 0 – a;
}
x = x + 1;
```



Implementation 2:

```
        ; r1 = a, r2 = x
        CMP   r1, #0        ; Compare a with 0
        RSBLT r1, r1, #0    ; a = 0 - a if a < 0
        ADD   r2, r2, #1    ; x = x + 1
```

RSBLT r1, r1, #0:: conditional execution of the RSB instruction with the condition code LT. If the condition LT (less than) is true, then set r1=0−r1

Compound Boolean Expression

```
x > 20 && x < 25
x == 20 || x == 25
!(x == 20 || x == 25)
```

C Program	Assembly Program

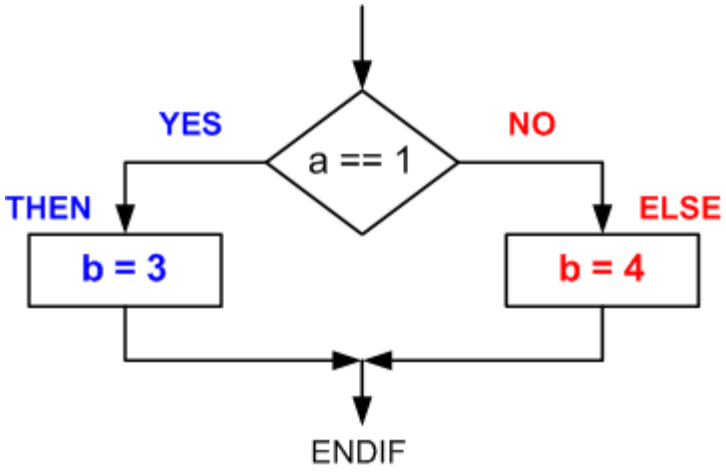
`// x is a signed integer`
`if(x <= 20 || x >= 25){`
`  a = 1`
`}`
	`  ; r0 = x`
	`  CMP  r0, #20  ; compare x and 20`
	`  BLE  then  ; go to then if x ≤ 20`
	`  CMP  r0, #25  ; compare x and 25`
	`  BLT  endif  ; go to endif if x < 25`
	`then  MOV  r1, #1  ; a = 1`
	`endif`

Logical OR operator (||) employs short-circuit evaluation, meaning it evaluates expressions from left to right and stops as soon as the result of the entire expression is determined. For (cond1||cond2): If cond1 evaluates to true (non-zero), the overall result of the || operation is already known to be true, so cond2 is not evaluated. If cond1 evaluates to false (zero), the evaluation proceeds to the next operand cond2.

If-then-else

C Program

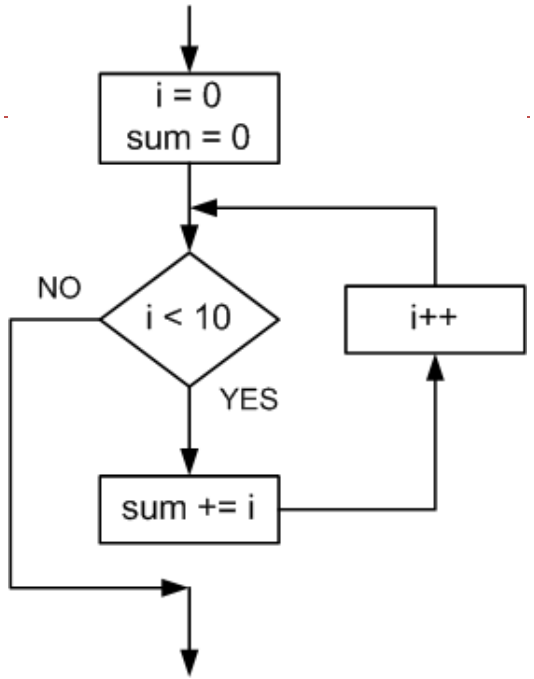
```
if (a == 1)
    b = 3;
else
    b = 4;
```




```
        ; r1 = a, r2 = b
        CMP r1, #1    ; compare a and 1
        BNE else      ; go to else if a ≠ 1
then    MOV r2, #3    ; b = 3
        B   endif     ; go to endif
else    MOV r2, #4    ; b = 4
endif
```

For Loop

```
C Program
int i;
int sum = 0;
for(i = 0; i < 10; i++){
  sum += i;
}
```



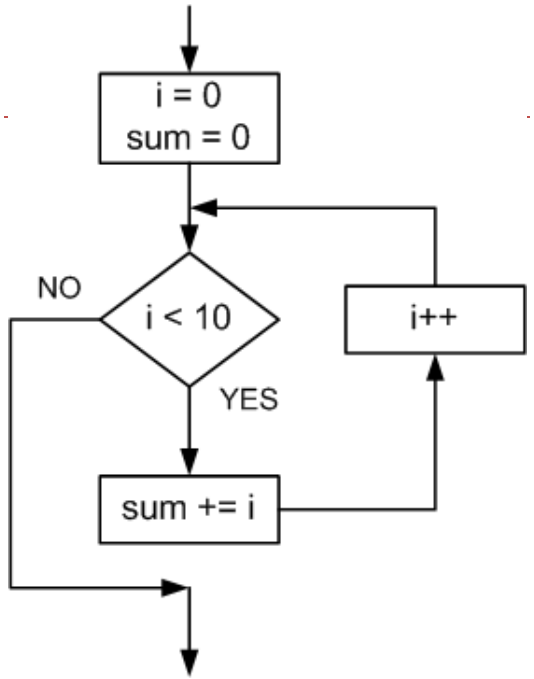
Implementation 1:

```
        MOV r0, #0   ; i
        MOV r1, #0   ; sum

        B   check
loop    ADD r1, r1, r0   ; sum += i
        ADD r0, r0, #1   ; i++
check   CMP r0, #10      ; check whether i < 10
        BLT loop         ; loop if signed less than
endloop
```

For Loop

```
C Program
int i;
int sum = 0;
for(i = 0; i < 10; i++){
  sum += i;
}
```

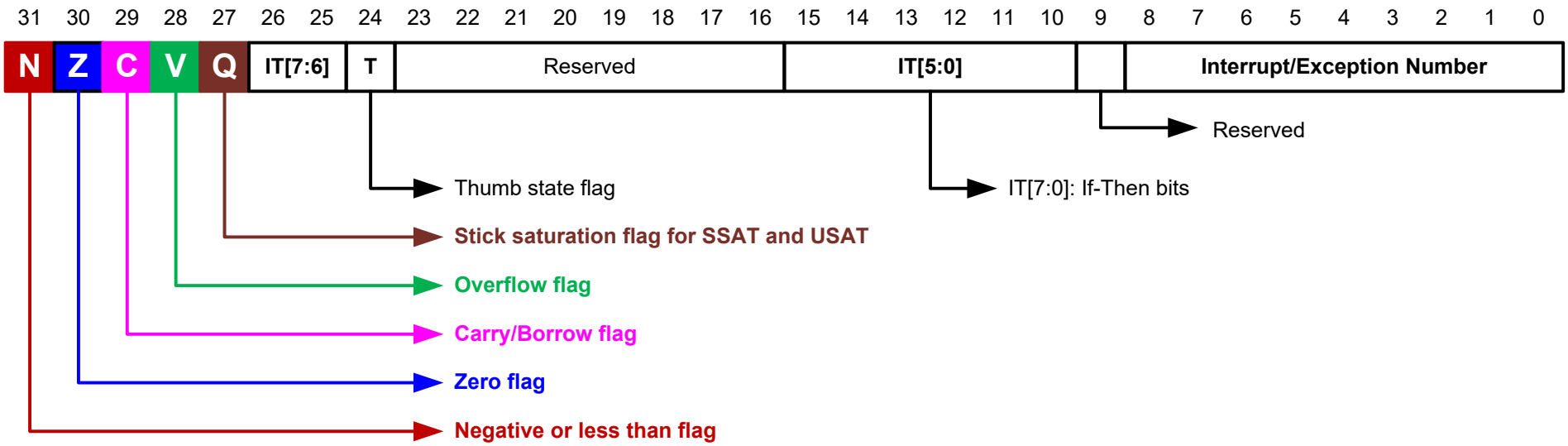


Implementation 2:

```
        MOV r0, #0   ; i
        MOV r1, #0   ; sum

loop    CMP r0, #10  ; check whether i < 10
        BGE endloop  ; skip if ≥
        ADD r1, r1, r0   ; sum += i
        ADD r0, r0, #1   ; i++
        B   loop
endloop
```

Combined Program Status Registers (xPSR)



Condition Codes

- The possible condition codes are listed below:

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Negative	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Signed Greater or equal	N=V
LT	Signed Less than	N!=V
GT	Signed Greater than	Z=0 & N=V
LE	Signed Less than or equal	Z=1 or N!=V
AL	Always	

Note AL is the default and does not need to be specified

Conditional Execution

Add instruction	Condition	Flag tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if Minus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

Conditional Execution Example: Signed Int

```
int32_t a; //a is
        signed int
...
if (a <= 0)
  y = -1;
else
  y = 1;
```

```
a ⟶ r0
y ⟶ r1
```

```
CMP   r0, #0
MOVLE r1, #-1 ; executed if LE
MOVGT r1, #1  ; executed if GT
```

LE: Signed Less than or Equal
GT: Signed Greater Than

Conditional Execution Example: Signed Int

```
int32_t a;
...
if (a==1 || a==7 || a==11)
   y = 1;
else
   y = -1;
```

```
a ⟶ r0
y ⟶ r1
```

```
CMP   r0, #1
CMPNE r0, #7  ; executed if r0 != 1
CMPNE r0, #11 ; executed if r0 != 7
MOVEQ r1, #1
MOVNE r1, #-1
```

CMP r0, #1 compares a with 1 and sets Z=1 if equal, else Z=0.

CMPNE r0, #7 runs only if the previous compare was not equal, and if it runs, it refreshes the flags by comparing a with 7.

CMPNE r0, #11 runs only if a was not 1 and not 7, and if it runs, it compares a with 11 to set Z accordingly.

MOVEQ r1, #1 executes only when Z=1 from any of the comparisons, so y becomes 1 if a matched 1, 7, or 11.

MOVNE r1, #-1 executes only when Z=0 after all relevant compares, so y becomes -1 when none of the values matched.

Compound Boolean Expression: Signed Int

C Program	Assembly Program

int32_t x;
...
if(x <= 20 || x >= 25){
  a = 1;
}
	; x → r0, a → r1
	CMP r0, #20 ; compare x and 20
	MOVLE r1, #1 ; a=1 if less or equal
	CMP r0, #25 ; compare x and 25
	MOVGE r1, #1 ; a=1 if greater or equal
	endif

Example 1: Greatest Common Divider (GCD)

Euclid's Algorithm

```
uint32_t a, b;
while (a != b ) {
    if (a > b)
       a = a – b;
    else
       b = b – a;
}
```

```
gcd  CMP r0, r1
     SUBHI r0, r0, r1
     SUBLO r1, r1, r0
     BNE gcd
```

```
      ; suppose r0 = a and r1 = b
gcd   CMP r0, r1       ; a > b?
      BEQ end          ; if a = b, done
      BLO less         ; a < b
      SUB r0, r0, r1   ; a = a – b
      B   gcd
less  SUB r1, r1, r0   ; b = b – a
      B   gcd
end
```

Explanation of above short version:

CMP r0, r1 compares r0 and r1, setting Z=1 if r0 == r1, which is the termination condition when tested by BNE later

SUBHI r0, r0, r1 executes only if the previous compare set HI (unsigned "higher"), i.e., r0 > r1 for unsigned ints r0 and r1

SUBLO r1, r1, r0 executes only if the previous compare of unsigned ints set LO (unsigned "lower"), i.e., r0 < r1 for unsigned ints r0 and r1

BNE gcd branches back to the label if Z=0 from the CMP, i.e., while r0 != r1; when they become equal, Z=1 and the loop falls through, leaving r0 == r1 == gcd(a, b)

Example 2

```
int foo(int x, int y) {
  if ( x + y < 0 )
    return 0;
  else
    return 1;
}
```

```
foo     ADDS r0, r0, r1
        BPL  PosOrZ
        MOV  r0, #0
done    MOV  pc, lr
PosOrZ  MOV  r0, r1
        B    done
```

```
foo     ADDS    r0, r0, r1   ;  r1 = x + y,  setting CCs
        MOVPL   r0, #1       ;  return 1 if n bit = 0
        MOVMI   r0, #0       ;  return 0 if n bit = 1
        MOV     pc, lr       ;  exit foo function
```

BPL PosOrZ: Branch to PosOrZ if condition PL is true (N == 0 indicating Positive or Zero) for ADDS r0, r0, r1

MOVPL r0, #1: conditional move that executes only when condition PL is true

MOVMI r0, #0: conditional move that executes only when condition MI (N == 1 indicating Negative) is true

Combination

Instruction	Operands	Brief description
CBZ	Rn, label	Compare and Branch if Zero
CBNZ	Rn, label	Compare and Branch if Non-Zero

- Except that it does not change the status flags, `CBZ Rn,label` is equivalent to:

```
CMP Rn, #0
BEQ label
```

- Except that it does not change the status flags, `CBNZ Rn,label` is equivalent to:

```
CMP  Rn, #0
BNE  label
```

Break and Continue

Example code for break	Example code for continue

for(int i = 0; i < 5; i++){
  if (i == 2) **break**;
  printf(“%d, ”, i)
}
	for(int i = 0; i < 5; i++){
	  if (i == 2) **continue**;
	  printf(“%d, ”, i)
	}

Output: ??	Output: ??

Break and Continue

Example code for break	Example code for continue

`for(int i = 0; i < 5; i++){`
`  if (i == 2) break;`
`  printf("%d, ", i)`
`}`
	`for(int i = 0; i < 5; i++){`
	`  if (i == 2) continue;`
	`  printf("%d, ", i)`
	`}`

Output: 0, 1 Break: break out of the loop	Output: 0, 1, 3, 4 Continue: skip the current loop iteration

Break and Continue

C Program	Assembly Program

`// Find string length`
`char str[] = "hello";`
`int len = 0;`
 	
`for( ; ; ) {`
`  if (*str == '\0')`
`  break;`
`  str++;`
`  len++;`
`}`
	`  ; r0 = string memory address`
	`  ; r1 = string length`
	`  MOV  r1, #0  ; len = 0`
	
	`loop  LDRB r2, [r1]`
	`  CBNZ r2, notZero`
	`  B  endloop`
	`notZero  ADD  r0, r0, #1  ; str++`
	`  ADD  r1, r1, #1  ; len++`
	`  B  loop`
	`endloop`

Summary: Condition Codes

Suffix	Description	Flags tested
EQ	EQual	Z=1
NE	Not Equal	Z=0
CS/HS	Unsigned Higher or Same	C=1
CC/LO	Unsigned LOwer	C=0
MI	MInus (Negative)	N=1
PL	PLus (Positive or Zero)	N=0
VS	oVerflow Set	V=1
VC	oVerflow Cleared	V=0
HI	Unsigned HIgher	C=1 & Z=0
LS	Unsigned Lower or Same	C=0 or Z=1
GE	Signed Greater or Equal	N=V
LT	Signed Less Than	N!=V
GT	Signed Greater Than	Z=0 & N=V
LE	Signed Less than or Equal	Z=1 or N!=V
AL	ALways	

Note AL is the default and does not need to be specified

Summary: Branch Instructions

	Instruction	Description	Flags tested
Unconditional Branch	`B label`	Branch to label	
Conditional Branch	`BEQ label`	Branch if **EQ**ual	`Z = 1`
	`BNE label`	Branch if **N**ot **E**qual	`Z = 0`
	`BCS/BHS label`	Branch if unsigned **H**igher or **S**ame	`C = 1`
	`BCC/BLO label`	Branch if unsigned **LO**wer	`C = 0`
	`BMI label`	Branch if **MI**nus (Negative)	`N = 1`
	`BPL label`	Branch if **PL**us (Positive or Zero)	`N = 0`
	`BVS label`	Branch if o**V**erflow **S**et	`V = 1`
	`BVC label`	Branch if o**V**erflow **C**lear	`V = 0`
	`BHI label`	Branch if unsigned **HI**gher	`C = 1 & Z = 0`
	`BLS label`	Branch if unsigned **L**ower or **S**ame	`C = 0 or Z = 1`
	`BGE label`	Branch if signed **G**reater or **E**qual	`N = V`
	`BLT label`	Branch if signed **L**ess **T**han	`N != V`
	`BGT label`	Branch if signed **G**reater **T**han	`Z = 0 & N = V`
	`BLE label`	Branch if signed **L**ess than or **E**qual	`Z = 1 or N = !V`

Summary: Conditionally Executed

Add instruction	Condition	Flag tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if Minus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

References

- Lecture 27. Branch instructions
  - https://www.youtube.com/watch?v=_QKD7f1cmRI&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=27
- Lecture 28. Conditional Execution
  - https://www.youtube.com/watch?v=9hIxG8L5-G4&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=28