L4 (CHAPTER 7)

Programming in Assembly
Part 3: Control Structures

Zonghua Gu, 2018

Acknowledgement: some slides taken from instructor resources of Daniel Lewis textbook

ARM programming model

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13: Stack Pointer (SP)
R14: Link Register (LR)
R15: Program Counter (PC)

CPSR (Current Program Status Register)

- Every arithmetic, logical, or shifting operation sets CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow).

Comparison Instructions

- These instructions set only the NZCV bits of CPSR, with no other effect.
  - CMP : compare
  - CMN : negated compare
  - TST : bit-wise test
  - TEQ : bit-wise negated test

Condition Codes

Condition Code	*Meaning*	*Requirements*
EQ	Equal	Z = 1
NE	Not equal	Z = 0
CS	Carry set	C = 1
CC	Carry clear	C = 0
MI	Minus/negative	N = 1
PL	Plus/positive or zero (non-negative)	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 || Z = 1
GE	Signed ≥ (“Greater than 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 || N ≠ V
AL	Always (unconditional)	only used with IT instruction

The condition is described as the state of a specific bit in the CPSR register. For example, when we compare two numbers a and b, and they turn out to be equal, we set the Zero bit (Z = 1), because a – b = 0. In this case we have EQual condition. If the first number was bigger, we would have a Greater Than condition and in the opposite case – Lower Than. There are more conditions, like Lower or Equal (LE), Greater or Equal (GE) and so on. Any one of these may be appended to any instruction mnemonic when used inside an If-Then-Else (IT) block.

ARM flow of control

- All operations can be performed conditionally, testing CPSR:
  - EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
  - You should never access CPSR bits directly. Always use condition codes for control flow decisions.
- Branch operation:

```
B fblock ;branch to code line with label
         ;“fblock”
```

  - Can be performed conditionally.

```
BNE fblock ;branch to code line with label
           ;“fblock” if NE bit is set in CPSR
```

Branch Instructions

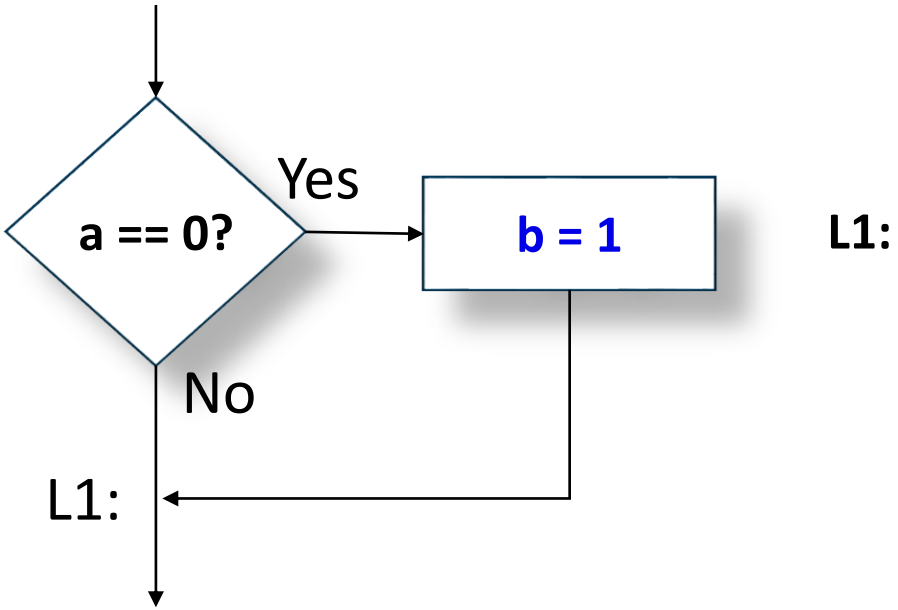
Branch Instructions	***Operation***	***{S}***	***Notes***
B{c} label	PC ← PC + imm	n/a	"c" is an *optional* condition code (see next slide)
IT$c_1c_2c_3$ cond	Each c_i is one of T, E, or *empty*	n/a	Controls 1-4 instructions in "IT block"
BL label	PC ← PC + imm; LR ← rtn adr	n/a	Subroutine call
BX reg	PC ← reg	n/a	"BX LR" often used as function return

B{c} Instructions

Compare	***Signed***	***Unsigned***
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
>=	BGE	Reverse Operands & Use BLS
<	BLT	Reverse Operands & Use BHI
<=	BLE	BLS

If-Then Statement with Branch Instruction

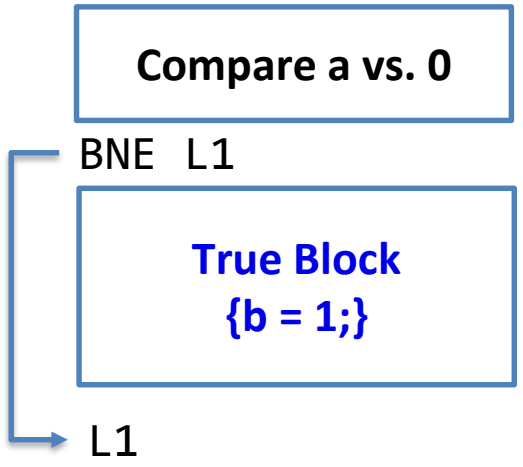
if (a == 0) {b = 1;}



Shorthand notation

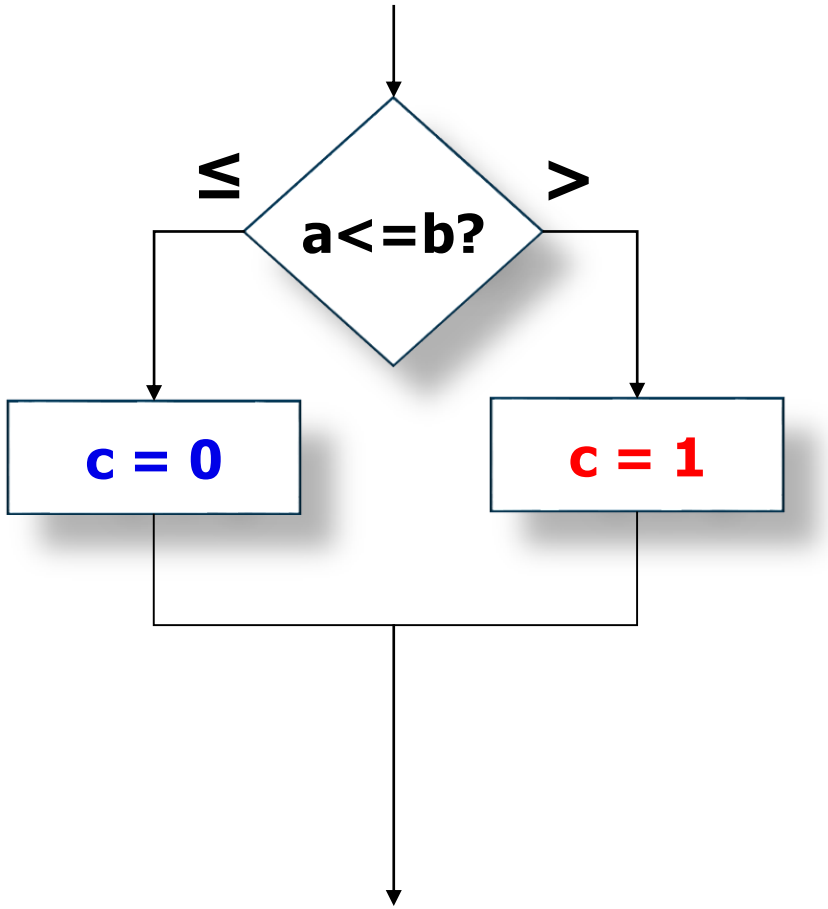
ADR R1, a
LDR R0, [R1]

```
LDR     R0, [addr(a)]
CMP     R0,#0 ;Compare a with 0
BNE     L1 ;branch to L1 if not
           ;equal (a!=0)
LDR     R0,=1
STR     R0, [addr(b)]
L1:
```

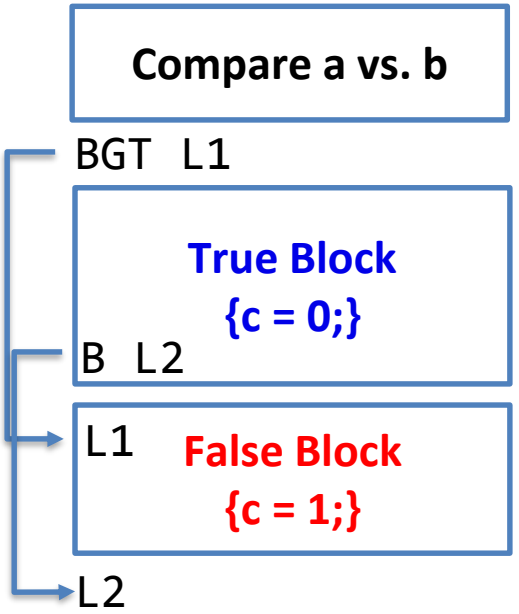


If-Then-Else Statement with Branch Instruction: Option 1

if (a <= b) {c = 0;} else {c=1;}

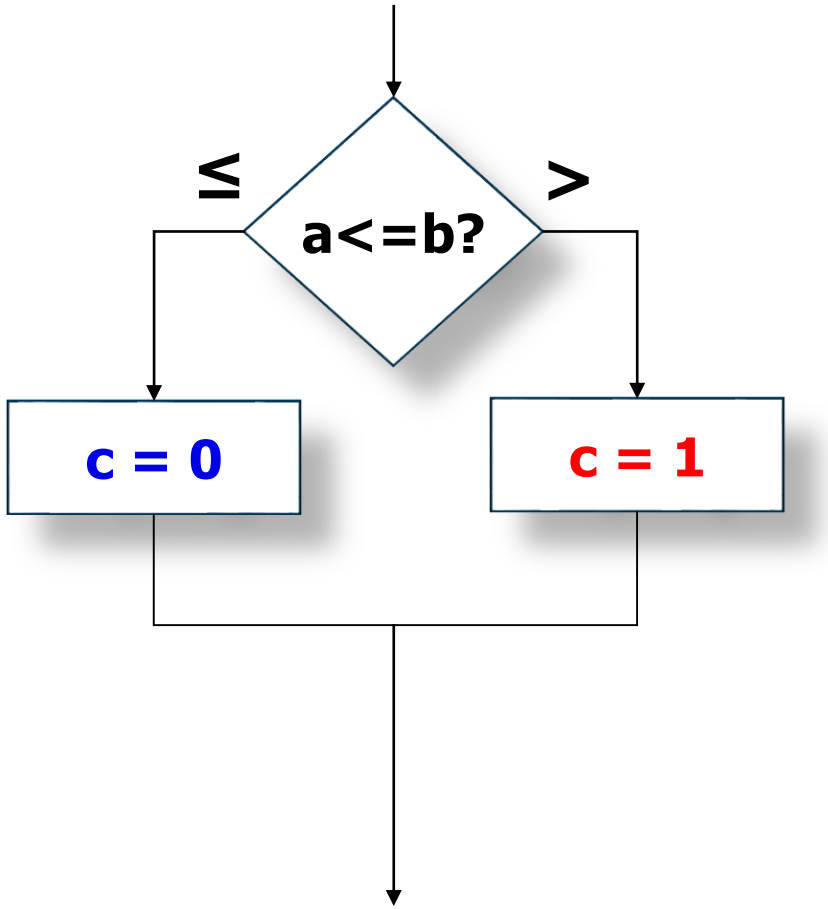



```
        LDR     R0, [addr(a)]
        LDR     R1, [addr(b)]
        CMP     R0,R1
        BGT     L1
        LDR     R0,=0
        B       L2
L1:     LDR     R0,=1
L2:     STR     R0, [addr(c)]
        ...
```

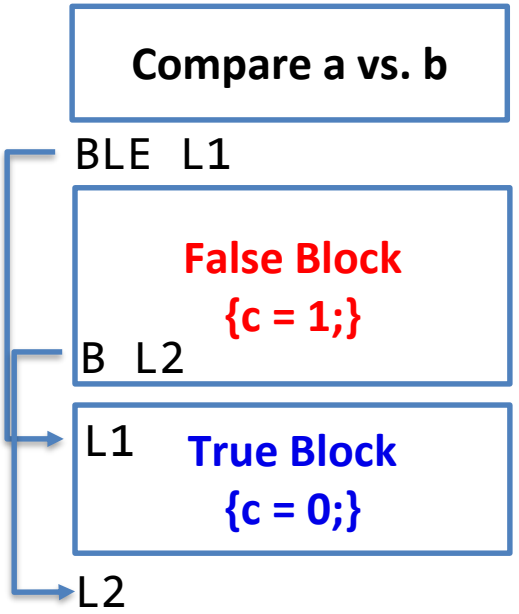


If-Then-Else Statement with Branch Instruction: Option 2

if (a <= b) {c = 0;} else {c=1;}




```
        LDR     R0, [addr(a)]
        LDR     R1, [addr(b)]
        CMP     R0,R1
        BLE     L1
        LDR     R0,=1
        B       L2
L1:     LDR     R0,=0
L2:     STR     R0, [addr(c)]
        ...
```

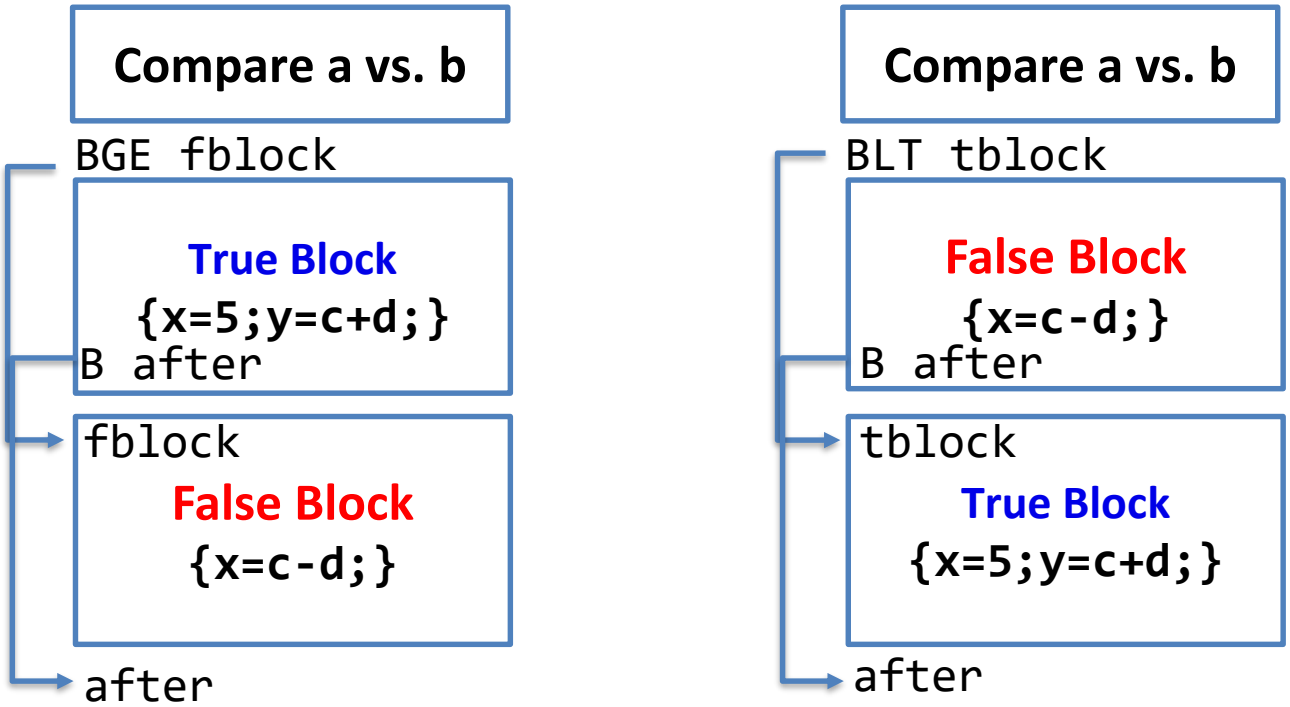


Another Example:

- C code:

```
if (a < b) {x = 5; y = c + d;} else {x = c - d;}
```

- Assembler code:



Assembler Part 1: Compute and Test condition

```
; compute and test condition
    ADR R4,a ; get address for a
    LDR R0,[R4] ; get value of a
    ADR R4,b ; get address for b
    LDR R1,[R4] ; get value for b
    CMP R0,R1 ; compare a and b; set GE bit in CPSR to 1 if a>=b
    BGE fblock ; Branch-if-GE: if GE bit is 1 (a>=b), branch to
                    ; label fblock (false block)
```

Assembler Part 2: True Block

```
; true block: { x = 5; y = c + d; }
   MOV R0,#5 ; set register R0=5
   ADR R4,x ; get address for variable x
   STR R0,[R4] ; store value 5 into x
   ADR R4,c ; get address for variable c
   LDR R0,[R4] ; get value of c
   ADR R4,d ; get address for variable d
   LDR R1,[R4] ; get value of d
   ADD R0,R0,R1 ; compute c+d
   ADR R4,y ; get address for y
   STR R0,[R4] ; store c+d into y
   B after ; branch around false block
```

Assembler Part 3: False Block

```
; false block {x = c - d;}
fblock ADR R4,c ; get address for variable c
    LDR R0,[R4] ; get value of c
    ADR R4,d ; get address for variable d
    LDR R1,[R4] ; get value for d
    SUB R0,R0,R1 ; compute a-b
    ADR R4,x ; get address for x
    STR R0,[R4] ; store value of x
after ...
```

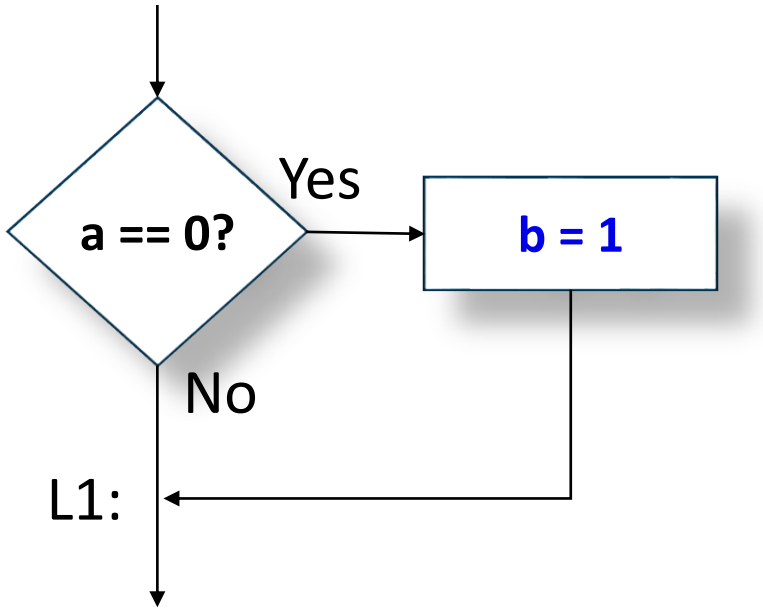
IT Instruction

The structure of the IT instruction is "IF-Then-(Else)" and the syntax is a construct of the two letters T and E:

IT refers to If-Then (next instruction is conditional)
ITT refers to If-Then-Then (next 2 instructions are conditional)
ITE refers to If-Then-Else (next 2 instructions are conditional)
ITTE refers to If-Then-Then-Else (next 3 instructions are conditional)
ITTEE refers to If-Then-Then-Else-Else (next 4 instructions are conditional)

If-Then Statement with IT Instruction

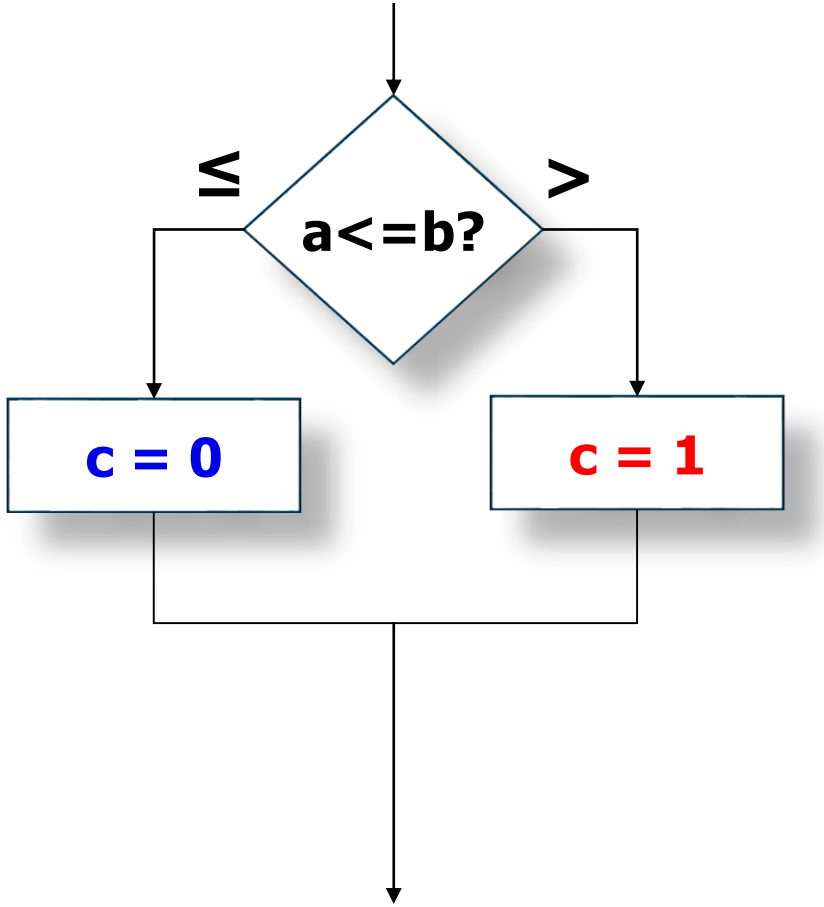
if (a == 0) b = 1 ;




```
LDR     R0,address(a)
CMP     R0,#0 ;Compare R0 with 0
ITT     EQ ;If-Then-Then: next 2
;instructions are conditional

LDREQ   R0,=1 ; in case EQ=1
STREQ   R0,address(b); in case EQ=1
```

If-Then-Else Statement with IT Instruction

if (a <= b) {c = 0;} else {c=1;}




```
LDR     R0, [addr(a)]
LDR     R1, [addr(b)]
CMP     R0, R1 ;Compare R0 with R1
ITE     GT ;If-Then-Else: next 2
;instructions are conditional

LDRGT   R0,=1 ; in case GT=1
LDRLE   R0,=0 ; in case LE=1
STR     R0, [addr[c])
```

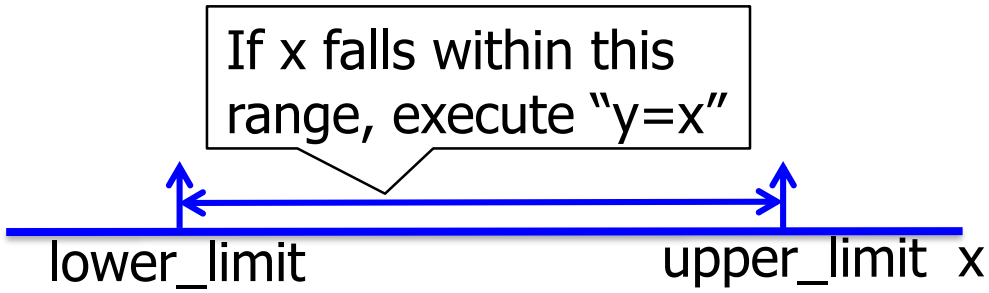
IT Instruction: More Examples

```
ITE    GT                ; Next 2 instructions are conditional
ADDGT  R1, R0, #55       ; in case GT(Greater-Than)=1
ADDLE  R1, R0, #48       ; in case GT=0 (LE: Less-or-Equal)

ITTE   NE                ; Next 3 instructions are conditional
ANDNE  R0, R0, R1        ; in case NE(Not-Equal)=1
ADDSNE R2, R2, #1        ; in case NE(Not-Equal)=1
MOVEQ  R2, R3            ; in case NE=0 (EQ: Equal)

ITTEE  EQ                ; Next 4 instructions are conditional
MOVEQ  R0, R1            ; in case EQ(Equal)=1
ADDEQ  R2, R2, #10       ; in case EQ(Equal)=1
ANDNE  R3, R3, #1        ; in case EQ=0 (NE: Not-Equal)
BNE    L1                ; in case EQ=0 (NE: Not-Equal), branch to statement
with label L1. Branch can only be used as the last instruction of an IT
block
```

Compound Conditionals: Example 1

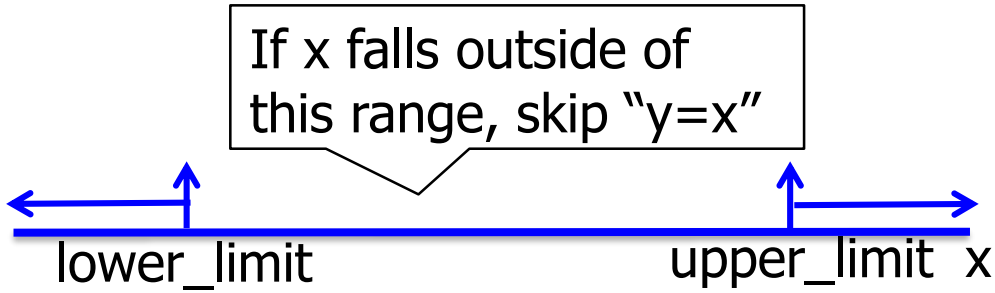
C code:

```
if (lower_limit <= x && x <= upper_limit) y = x ;
```



Equivalent C code:

```
if (!(lower_limit <= x && x <= upper_limit)) goto L1
y = x ;
L1:
```



Compound Conditionals: Example 1

Equivalent C code (with DeMorgan's law):

```
if (x < lower_limit || x > upper_limit) goto L1
y = x ;
L1:
```

Equivalent C code:

```
if (x < lower_limit) goto L1
if (x > upper_limit) goto L1
y = x ;
L1:
```

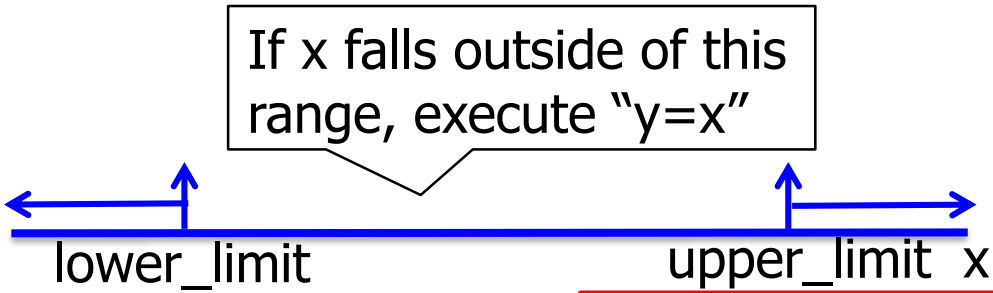
Assembler code:

```
      LDR   R0,[addr(x)]
      LDR   R1,lower_limit
      CMP   R0,R1
      BLT   L1
      LDR   R1,upper_limit
      CMP   R0,R1
      BGT   L1
      STR   R0,[addr(y)]
L1:   ...
```

Compound Conditionals: Example 2 (Option 1)

C code:

```
if (x < lower_limit || upper_limit < x) y = x ;
```



Equivalent C code:

```
if (x < lower_limit) goto L1
if (x > upper_limit) goto L1
goto L2
L1: y = x ;
L2:
```

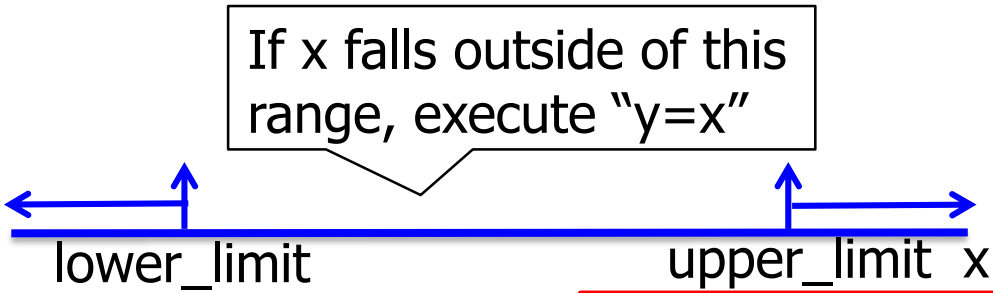
Assembler code:

```
      LDR   R0,[addr(x)]
      LDR   R1,lower_limit
      CMP   R0,R1
      BLT   L1
      LDR   R1,upper_limit
      CMP   R0,R1
      BGT   L1
      B L2
L1:   STR   R0,[addr(y)]
L2:   ...
```

Compound Conditionals: Example 2 (Option 2)

C code:

```
if (x < lower_limit || upper_limit < x) y = x ;
```



Equivalent C code:

```
if (x < lower_limit) goto L1
if (x <=upper_limit) goto L2
L1: y = x ;
L2:
```

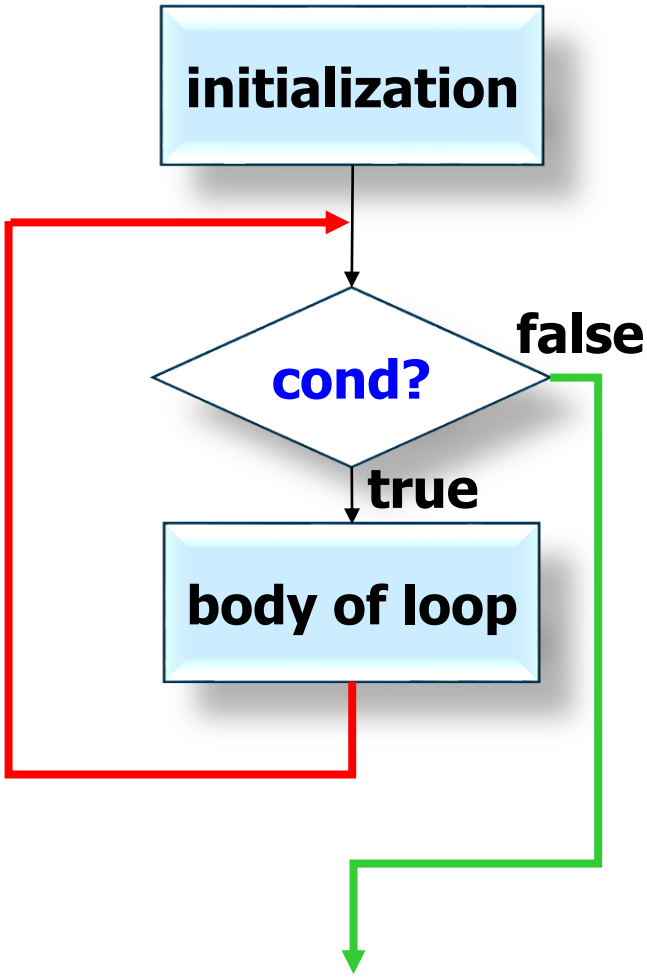
More efficient, with fewer branch instructions

Assembler code:

```
     LDR   R0,[addr(x)]
     LDR   R1,lower_limit
     CMP   R0,R1
     BLT   L1
     LDR   R1,upper_limit
     CMP   R0,R1
     BLE   L2
L1:  STR   R0,[addr(y)]
L2:  ...
```

Loops: Basic Structure

C code flowchart:



Assembler code (option 1):

```
top: CMP
B{!cond} done
    Loop Body
B top
done
```

Assembler code (option 2):

```
top: CMP
    Loop Body
CMP
B{cond} top
```

Loops: Predetermined #Iterations

C code:

```
for (n = 0; n < 100; n++)
{
 … //Loop body
}
```

Assembler code (option 1):

```
        LDR     R0,=0
top:    CMP     R0,#100
        BGE     done ;Branch greater than or equal to (n>=100)
        …
        ADD     R0,R0,#1
        B       top
done:
```

Assembler code (option 2):

```
        LDR     R0,=0
top:    …
        ADD     R0,R0,#1
        CMP     R0,#100
        BLT     top ;Branch  less than (n<100)
done:
```

More efficient, with fewer branch instructions.

Option 2 Caveat

- Option 2 executes the loop for at least one iteration
- Not correct for loops with 0 iterations!

```
for (n = 0; n < 0; n++)
{
 … //Loop body is never executed
}
```

- But this is a pathological example
  - Normal loops should have at least 1 iteration

Larger Example: FIR filter

- **C:**

```
for (int i=0, f=0; i<N; i++)
    f = f + c[i]*x[i];
```

- **Assembler:**

```
; loop initiation code
    MOV R0,#0 ; use R0 for loop index i
    MOV R8,#0 ; use R8 for array offset
    ADR R2,N ; get address for variable N
    LDR R1,[R2] ; get value of N and put in register R1
    MOV R2,#0 ; use R2 for f
    ADR R3,c ; load R3 with base address of int array c
    ADR R5,x ; load R5 with base address of int array x
; loop body
    loop LDR R4,[R3,R8] ; load c[i] at memory address R3+R8
                        ; into R4
    LDR R6,[R5,R8]   ; load x[i] at memory address R5+R8
                     ; into R6
    MUL R4,R4,R6 ; compute c[i]*x[i] and store result in R4
    ADD R2,R2,R4 ; add into running sum f
    ADD R8,R8,#4 ; add 4 (one word) to array offset R8
    ADD R0,R0,#1 ; add 1 to loop index i
    CMP R0,R1 ; Compare i and N
    BLT loop ; if i < N, continue and go back to label "loop"
```

Reg	Meaning
R0	Loop index i, incremented by 1 at each loop iteration
R1	Loop bound N
R2	Running sum f
R3	Base address of array c
R4	c[i]
R5	Base address of array x
R6	x[i]
R8	Array offset, incremented by 4 at each loop iteration since both c and x are 4-byte int arrays

Loops: Variable # Iterations

```
//GCD(a,b) computes the Greatest Common Divisor of a and b
while (a != b)
{
   if (a > b) a = a – b ;
   else b = b – a;
}
```

```
        LDR     R0, [addr(a)]
        LDR     R1, [addr(b)]
top:    CMP     R0, R1
        BEQ     done
        ITE     GT ;If-Then-Else: next
                ;2 instructions are conditional
        SUBGT   R0,R0,R1 ;in case GT=1
        SUBLE   R1,R1,R0 ;in case GT=0
        B top ;go back to top
done:   STR     R0, [addr(a)]
        STR     R1, [addr(b)]
```

Summary

- ARM programming model
- Two alternative control flow schemes:
  - Branch (conditional) instructions
  - ITE family of instructions
- Loops
  - Two options