Chapter 7

IF STATEMENTS

- Imitating high-level test condition does not result in very efficient assembly-level implementation.
- High level test condition is reversed in assembly-level to avoid the need for an additional unconditional jump.

E.g. if (a > b)
{ S1 }

→

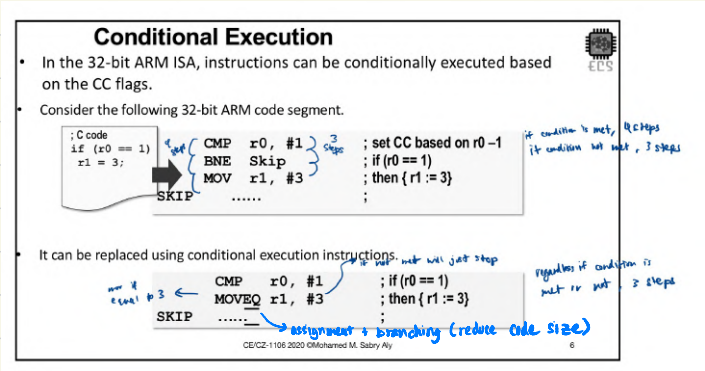
if a < b

```
CMP  a, b
BLE  Skip
{S1}
Skip
```

Note: reverse condition test of HS is LO, reverse of LT is GE

* when we see condition with if statement, reverse it, apply branch on that condition

CONDITIONAL EXECUTION



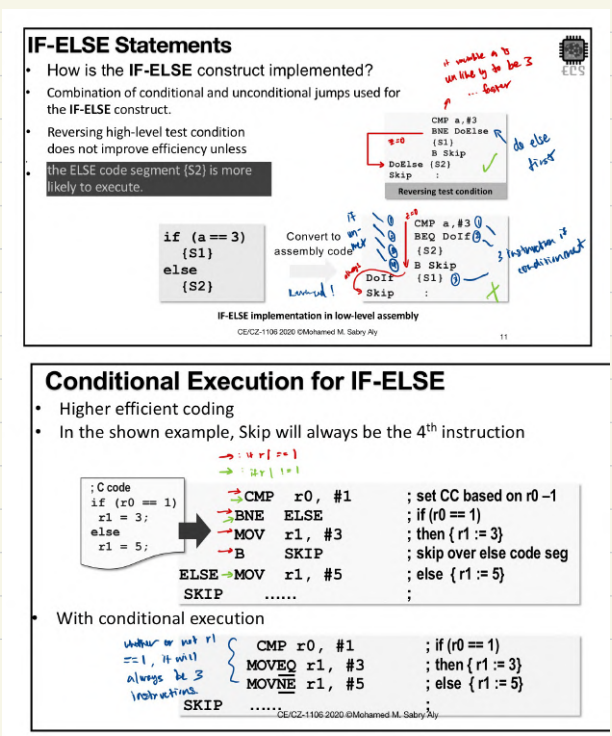
Does Conditional Execution work with just 1 Instruction?

No, as long as the status registers are not altered from comparison instruction, can do as many as you want

① Do not change the registers from instructions
→ can do all instructions conditionally, BUT doesn't change status registers
→

② When you are doing so, you are gonna make the time needed to reach the outside of if statement, when condition not met, much longer
→ need balance
⇒ if frequently mapped : better to have more compact way of doing it.
⇒ if frequently unmapped : better to do branching; saves time when running program overall

IF-ELSE STATEMENTS



COMPOUNDS "AND" CONDITIONS

- Logical AND can bind multiple basic relational conditions.

e.g. `if ((a==b) && (b>0)) {S1}`

(a==b): least likely condition

order of compound AND test

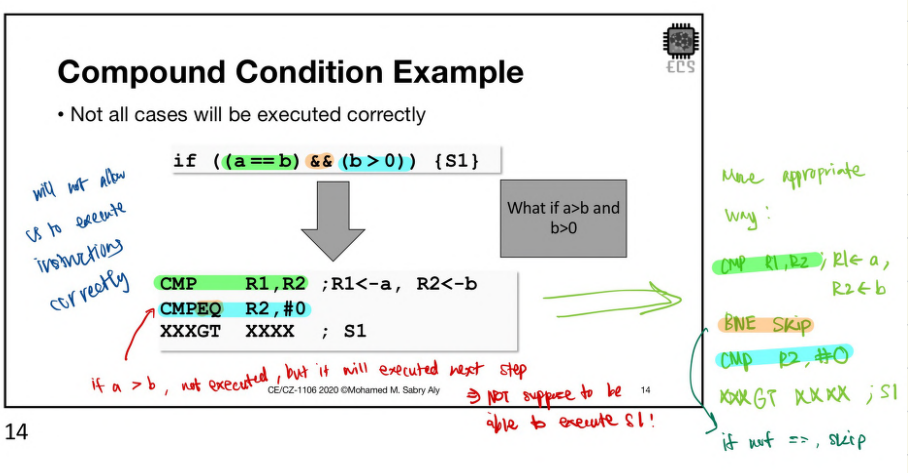
- Compilers resolve compound conditions into simpler ones.

```
    if (a!=b) then Skip
    if (b<=0) then Skip
       {S1}
Skip    :
```

→ negate first condition, skip

→ negate second condition, skip

- Elementary conditions bound by the logical AND are tested from left-to-right, in the order given in the C program.
- The first false condition means the remaining conditions are not computed. This is called fast Boolean operation.
- Keep the LEAST LIKELY condition leftmost in your program for more efficient execution



COMPOUND "OR" CONDITIONS

- Logical OR can bind multiple basic relational conditions.

e.g. `if ((a == 1) || (a == 2)) {S1}` — most likely condition

- Most compilers eliminate an unconditional jump at the end of the compound "OR" series by reversing the last conditional test

```
        if (a == 1) then DoIf
        if (a != 2) then Skip
DoIf    {S1}
Skip    :
```

} saves instructions & remove unnecessary jumps

- Keep the MOST LIKELY condition leftmost in your program for more efficient execution
- If >2 conditions, just negate last condition

Compound Condition Example

```
if ((a == 1) || (a == 2)) {S1}
```

if a = 1, 3 instructions executed

With conditional assignment
S1 needs to be replicated.
Not suitable for multiple instructions

```
CMP    R1,#1  ;R1<-a
XXXEQ  XXXX   ; S1
CMPNE  R1,#2
XXXEQ  XXXX   ; S1
```

replicated

⇒ 4 instructions executed

A more holistic approach

```
        CMP    R1,#1
        BEQ    doIF
        CMPNE  R1,#2
doIF    XXXEQ  XXXX   ; S1
```

Bcc : conditional jump

BRANCHLESS LOGIC

- Branchless logic avoid using conditional jump instructions when implementing logical constructs.
- Bcc instructions may result in costly flushing operations when the wrong next instruction is pre-fetched into the CPU's pipeline
- How is branchless logic implemented?

→ Exploit arithmetic relationship to transform the test condition into the corresponding desired outcome. Can only be applied in special cases and desired outcome are usually Boolean value

→ condition execution can be used to avoid branching

⇒ also greatly improves flow-control efficiency

e.g.

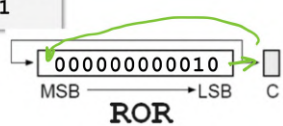
```
if ((X & 2) == 2)
  X = true;
else
  X = false;
```

R1 = 0x001

R1 = 0x000

```
AND R1,R1,#0x02
ROR R1,R1,#1
```

rotation to the right



- Branchless logic and conditional execution techniques can help keep the CPU pipeline efficient by maintaining sequential execution

→ no matter whether or not condition is met, always running same set of instructions.

1. if
2. switch
3. Loops

Chapter 7: Flow-control Constructs

Mohamed M. Sabry Aly
N4-02c-92

1

Learning Objectives

- Be able to convert a high level IF and IF-ELSE construct to its low-level equivalent.
- Describe compute-efficient considerations for compound AND and OR constructs.
- Describe and analyze simple branchless logic constructs.

2

IF Statements

- How is the **IF** construct implemented?
- **Imitating** the high-level test condition does not result in very efficient assembly-level implementation.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
      CMP a,b
      BGT DoIF
      B Skip
DoIF  {S1}
Skip
```

branch if greater than

- High-level test condition is **reversed** in assembly-level to avoid the need for an additional unconditional jump.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
      CMP a,b
      BLE Skip
      {S1}
Skip
```

optimized

Note: The reverse condition test of **HS** is **LO**, reverse of **LT** is **GE**

3

Don't use SUBS as it will mess up R1

when we see condition with if statement, reverse it, apply branch on that condition

Find Largest Number

```
      MOV   R0,#0x100     ;setup pointer to first array element
      MOV   R1,#9
      LDR   R3,[R0]
Loop  LDR   R2,[R0,#4]!
      CMP   R2,R3         ;compare R3 and R2 (i.e. R2-R3)
      BLO   Skip          ;branch if R2 < current max (i.e. R3)
      MOV   R3,R2         ;update current max. in R3 with R2
Skip  SUBS  R1,R1,#1      ;decrement 1 from counter register
      BNE   Loop          ;jump back to Loop if not zero
```

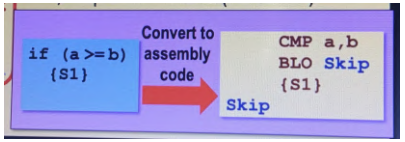
```
if R2 >= R3   // R2 larger or equal to current max
{
  R3 = R2;    // then update R3 with new max R2
}
```

{S1}

if R2 is lower than R3, then go to skip. if not, executed step right away

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

4



Can We Avoid Reversion?

```
        MOV   R0,#0x100     ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR   R2,[R0,#4]!
        CMP   R2,R3
        BLO   Skip
        MOV   R3,R2
Skip    SUBS  R1,R1,#1
        JNE   Loop
```

```
if R2 >= R3   // R2 larger or equal to current max
{
  R3 = R2;    // then update R3 with new max R2
}
```

```
        CMP   R2,R3
        BHS   Assign
        SUBS  R1,R1,#1
        BNE   Loop
Assign  MOV   R3,R2
        SUBS  R1,R1,#1
        BNE   Loop
```

Redundant

5

Conditional Execution

- In the 32-bit ARM ISA, instructions can be conditionally executed based on the CC flags.
- Consider the following 32-bit ARM code segment.

```
; C code
if (r0 == 1)
 r1 = 3;
```

```
CMP   r0, #1    ; set CC based on r0 –1
BNE   Skip      ; if (r0 == 1)
MOV   r1, #3    ; then { r1 := 3}
SKIP  ......    ;
```

- It can be replaced using conditional execution instructions.

```
CMP   r0, #1    ; if (r0 == 1)
MOVEQ r1, #3    ; then { r1 := 3}
SKIP  ......    ;
```


6

Largest with Conditional Execution

- Significant reduction in Code size

```
        MOV   R0,#0x100     ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR   R2,[R0,#4]!
        CMP   R2,R3         ;compare R3 and R2 (i.e. R2-R3)
        MOVGE R3,R2         ;update current max. in R3 with R2
        SUBS  R1,R1,#1      ;decrement 1 from counter register
        JNE   Loop          ;jump back to Loop if not zero
```

```
if R2 >= R3   // R2 larger or equal to current max
{
  R3 = R2;    // then update R3 with new max R2
}
```


7

Does Conditional Execution work with Just 1 Instruction?

No, as long as the status registers are not altered from the comparison instruction

8

Average cycles:

5.4 instructions per iteration

Find Largest Number, and Store Its Index

No conditional execution

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
      R0+=4;R2=X[R0];
      if(R2>=R3){
            R3=R2;
            R4=10-R1;
      }
}
```

```
      MOV   R0,#0x100   ;setup pointer to first array element
      MOV   R1,#9       ;load 9 into counter register
      MOV   R4,#0       ;load 0 into index_max register
      LDR   R3,[R0]     ; 1st no. in array is current max
Loop  LDR   R2,[R0,#4]! ;get next no. in array
      CMP   R2,R3       ;compare R3 and R2 (i.e. R2-R3)
      BLO   Skip        ;branch if R2 <  current max (i.e.
      MOV   R3,R2       ;update current max. in R3 with R2
      RSUB  R4,R1,#10   ;Store the index in R4
Skip  SUBS  R1,R1,#1    ;decrement 1 from counter register
      BNE   Loop        ;jump back to Loop if not zero
```

unsigned numbers

find next element

9

Find Largest Number, and Store Its Index

With conditional execution

much shorter

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
      R0+=4;R2=X[R0];
      if(R2>=R3){
            R3=R2;
            R4=10-R1;
      }
}
```

```
      MOV   R0,#0x100   ;setup pointer to first array element
      MOV   R1,#9       ;load 9 into counter register
      MOV   R4,#0       ;load 0 into index_max register
      LDR   R3,[R0]     ; 1st no. in array is current max
Loop  LDR   R2,[R0,#4]! ;get next no. in array
      CMP   R2,R3       ;compare R3 and R2 (i.e. R2-R3)
      MOVGE R3,R2       ;update current max. in R3 with R2
      RSUBGE R4,R1,#10  ;Store the index in R4
      SUBS  R1,R1,#1    ;decrement 1 from counter register
      BNE   Loop        ;jump back to Loop if not zero
```

find 6 instructions per iteration

whether or not condition is met

10

In this case:
The more the condition wasn't, without conditional execution would lead to a faster program run overall.

IF-ELSE Statements

- How is the **IF-ELSE** construct implemented?
- Combination of conditional and unconditional jumps used for the **IF-ELSE** construct.
- Reversing high-level test condition does not improve efficiency unless
- the ELSE code segment {S2} is more likely to execute.

it would a is likely to be 3 ... faster

```
      CMP a,#3
      BNE DoElse
      {S1}
      B Skip
DoElse {S2}
Skip   :
```

Reversing test condition

do else first

```
if (a == 3)
  {S1}
else
  {S2}
```

Convert to assembly code

```
      CMP a,#3
      BEQ DoIf
      {S2}
      B Skip
DoIf  {S1}
Skip   :
```

3 instruction if conditionment

IF-ELSE implementation in low-level assembly

11

Conditional Execution for IF-ELSE

- Higher efficient coding
- In the shown example, Skip will always be the 4th instruction

```
; C code
if (r0 == 1)
 r1 = 3;
else
 r1 = 5;
```

```
      CMP   r0, #1      ; set CC based on r0 –1
      BNE   ELSE        ; if (r0 == 1)
      MOV   r1, #3      ; then { r1 := 3}
      B     SKIP        ; skip over else code seg
ELSE  MOV   r1, #5      ; else { r1 := 5}
SKIP  .....             ;
```

- With conditional execution

whether or not r1 == 1, it will always be 3 instructions

```
      CMP r0, #1        ; if (r0 == 1)
      MOVEQ r1, #3      ; then { r1 := 3}
      MOVNE r1, #5      ; else { r1 := 5}
SKIP  .....             ;
```


12

Compound AND Conditions

- How are compound AND conditions handled?
- Logical AND can bind multiple basic relational conditions.

e.g. `if ((a == b) && (b > 0)) {S1}` — order of compound AND test

- Compilers resolve compound conditions into simpler ones.

```
    if (a != b) then Skip
    if (b <= 0) then Skip
       {S1}
Skip    :
```

- Elementary conditions bound by the logical AND are tested from **left-to-right**, in the order given in the C program.
- The first false condition means the remaining conditions are not computed. This is called the **fast Boolean operation**.
- Keep the **least likely** condition **leftmost** in your program for more efficient execution.

13

Compound Condition Example

- Not all cases will be executed correctly

```
if ((a == b) && (b > 0)) {S1}
```

What if a>b and b>0

```
CMP     R1,R2 ;R1<-a, R2<-b
CMPEQ   R2,#0
XXXGT   XXXX  ; S1
```


14

Compound OR Conditions

- How are compound OR conditions handled?

e.g. `if ((a == 1) || (a == 2)) {S1}`

- Most compilers eliminate an unconditional jump at the end of the compound OR series by **reversing** the **last conditional** test.

```
     if (a == 1) then DoIf
     if (a != 2) then Skip
DoIf {S1}
Skip   :
```

- The conditional test that is **most likely** to be **true** should be kept leftmost.

15

Compound Condition Example

```
if ((a == 1) || (a == 2)) {S1}
```

With conditional assignment
S1 needs to be replicated.
Not suitable for multiple instructions

```
CMP     R1,#1 ;R1<-a
XXXEQ   XXXX  ; S1
CMPNE   R1,#2
XXXEQ   XXXX  ; S1
```

A more holistic approach

```
    CMP     R1,#1
    BEQ     doIF
    CMPNE   R1,#2
doIF XXXEQ  XXXX  ; S1
```


16

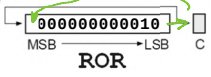
Branchless Logic

- Branchless logic avoid using conditional jump instructions when implementing logical constructs.
- **Bcc** instructions may result in costly **flushing** operations when the wrong next instruction is pre-fetched into the CPU's pipeline.
- How is branchless logic implemented?
- **Exploit arithmetic relationship** to transform the test condition into the corresponding desired outcome. Can only be applied in special cases and desired outcomes are usually Boolean values.

e.g.

```
if ((X & 2) == 2)
  X = true;
else
  X = false;
```

```
AND R1,R1,#0x02
ROR R1,R1,#1
```

- **Conditional execution** can be used to avoid branching.

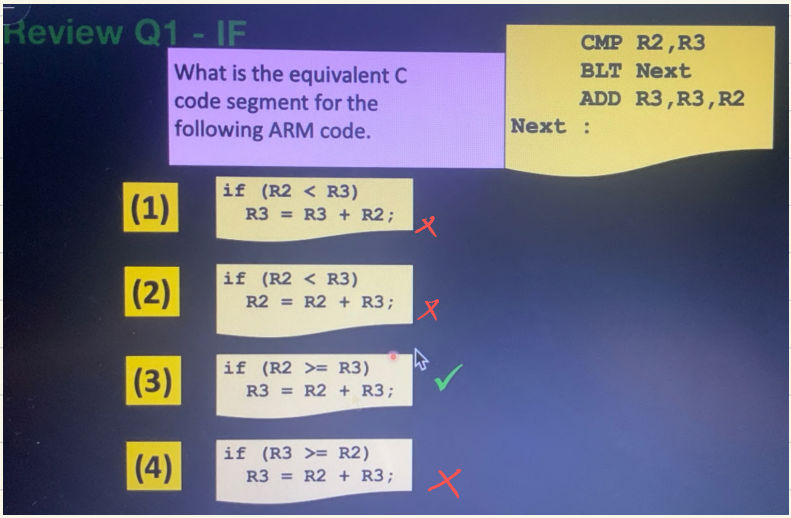
17

Summary

- The **IF** and **ELSE** constructs are implemented using one or more conditional jump (**Bcc**).
  - Using the **reverse** conditional test can help the IF construct execute more efficiently.
- **Conditional execution in ARM** greatly improves flow-control efficiency.
- Sequencing the **least likely** or **most likely** conditional test can help improve execution speed of compound **AND** and **OR** respectively .
- **Branchless logic** and **conditional execution** techniques can help keep the CPU pipeline efficient by maintaining **sequential** execution.

18

19/2 lecture



(1) Need to use reverse condition for "<", which is ">=" or BGE for efficient implementation

(2) Same as (1) but in ADD, R3 is destination register. So ADD R3, R3, R2 gives R3 = R2 + R3

(3) reverse condition correct

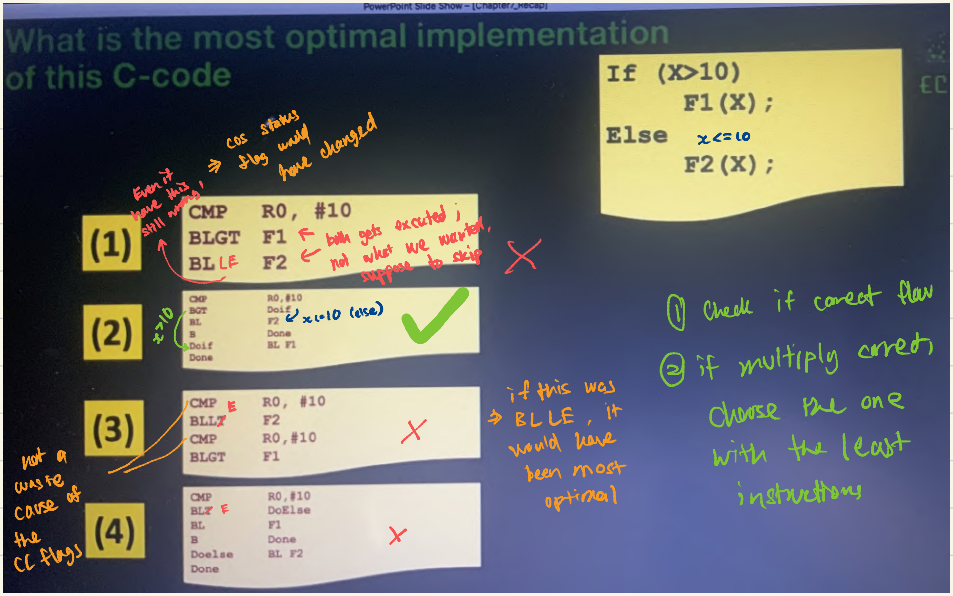
CMP R2, R3

↑

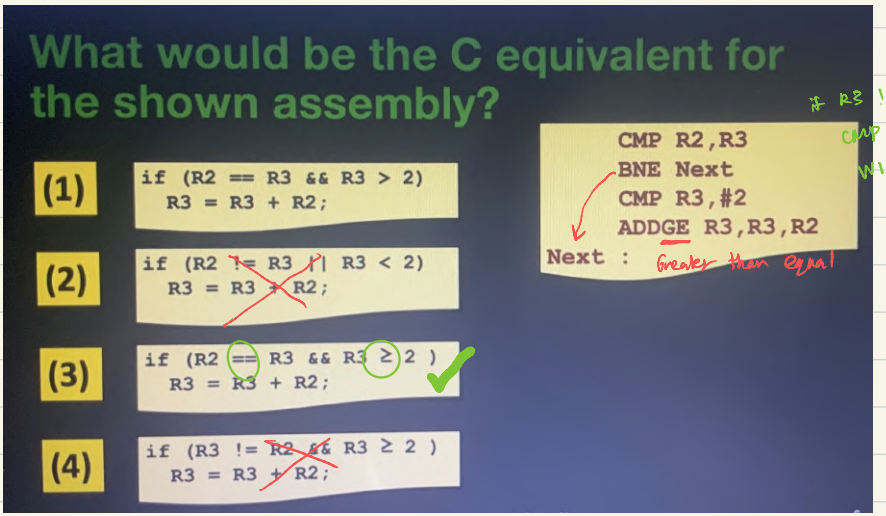
comes first

(4) reverse condition correct

but wrong order of R2, R3 in CMP instruction


What is the most optimal implementation of this C-code
If (X>10)
F1(X);
Else
F2(X);
x<=10
(1)
CMP R0, #10
BLGT F1
BL LE F2
Even if none this still running,
cos status flag would have changed
both gets executed, not what we wanted, suppose to skip
(2)
CMP R0,#10
BGT Doif
BL F2
B Done
Doif BL F1
Done
x>10
x<=10 (else)
(3)
CMP R0, #10
BLLE F2
CMP R0,#10
BLGT F1
not a waste cause of the CC flags
if this was BL LE, it would have been most optimal
(4)
CMP R0,#10
BLE DoElse
BL F1
B Done
Doelse BL F2
Done
(1) Check if correct flow
(2) if multiply correct, choose the one with the least instructions

22/2 lecture


What would be the C equivalent for the shown assembly?
(1)
if (R2 == R3 && R3 > 2)
R3 = R3 + R2;
(2)
if (R2 != R3 || R3 < 2)
R3 = R3 + R2;
(3)
if (R2 == R3 && R3 ≥ 2)
R3 = R3 + R2;
(4)
if (R3 != R2 && R3 ≥ 2)
R3 = R3 + R2;
CMP R2,R3
BNE Next
CMP R3,#2
ADDGE R3,R3,R2
Next :
Greater than equal
if R3 != R2, CMP & ADD GE WILL NOT BE EXECUTED

Chapter 7: Flow-control Constructs

Mohamed M. Sabry Aly
N4-02c-92

1

Learning Objectives

- Describe how SWITCH constructs can be implemented efficiently for consecutive narrow and random wide cases.
- Contrast the implementations of pre and post-test loop constructs like WHILE, DO-WHILE and FOR.

2

Switch Statement

- How is the **SWITCH** construct implemented?
- The assembly code produced varies between compilers and depends on the nature and range of the case values.
- Two different SWITCH scenarios are examined:

```
switch(x) {
 case 0 : {S0};
          break;
 case 1 : {S1};
          break
 case 2 : {S2};
          break;
 case 3 : {S3};
}
```

Running values narrow range

```
switch(x) {
 case 1    : {S0};
             break;
 case 10   : {S1};
             break
 case 100  : {S2};
             break;
 case 1000 : {S3};
}
```

Random values wide range

3

consecutive range of values or something completely random

Switch – Running & Narrow Values

long sequence before SS is executed

```
switch(x)
{
case 0:
 {S0};
 break;

case 1:
 {S1};
 break;

case 2:
 {S2};
 break;

case 3:
 {S3};
}
```

```
if(x == 0)
 {S0};
else if(x == 1)
 {S1};
else if(x == 2)
 {S2};
else if(x == 3)
 {S3};
```

Not efficient when there are many cases and the value of x is the largest.

Is there a better way?

Standard cascade if-else-if implementation

4

jump table only works with evenly spaced cases ⇒ sequential

Best case scenario: if R1 value == 0, 6 instructions executed (fastest)

Worst case scenario: (case 3) 9 instructions

Best case scenario: if R1 value == 0 6 instructions executed

Worst case scenario: (case 3) 6 instructions

Jump Table: whether it is worst or best case, same set of instructions

Switch – Running & Narrow Values

- If cases are consecutive narrow range values, a **Jump Table** is used to avoid testing each case in turn.
- Jump table contains list of **start addresses** for the code segments that is associated with each case values.
- Var **x** on which the switch is decided, acts as an **offset** into the table to access the corresponding start address.
- Start address is loaded into **PC** to execute the required code segment {Sx}.

```
switch(x)
{
case 0:
 {S0};
 break;

case 1:
 {S1};
 break;

case 2:
 {S2};
 break;

case 3:
 {S3};
}
```

```
MOV  R1, #0x08
LDR PC,[R1,R2,LSL#2]
```

Variable x, multiply by 4; Base addr; Table offset

Code Memory

Address	Contents	
JT + 0	0x020	Start address of {S0}
JT + 4	0x048	
JT + 8	0x05C	Start address of {S1}
:		Jump Table

Note: Time taken is on average less than the equivalent if-else-if cascade and is independent of number of cases in the switch construct.

5

Jump Table Example

ADR loads TBL to R3

Cascaded if-else

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
}
```

```
      LDR R1, [R2]
      CMP R1, #0
      BEQ C1
      CMP R1, #1
      BEQ C2
      CMP R1, #2
      BEQ C3
      MOV R0, #66 ;'B'
RES   STR R0, [R2,#4]
      .
      .
      .
C1    MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Jump Table

```
      LDR R1, [R2]
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
0xA00 C1   MOV R0, #70 ;'F'
      B RES
0xA08 C2   MOV R0, #68 ;'D'
      B RES
      C3   MOV R0, #67 ;'C'
      B RES
```

Address	Contents	
TBL + 0	0xA00	grade 1 or 0
TBL + 4	0xA08	grade 2 or 4
TBL + 8	0xA10	grade 3 or 4
:		Jump Table

6

worst case = 9

best case = 6

Worse = best = 6

① if value in R1 == 1,
② TBL as base address
③ R1 = 1, multiply by 4 (LSL #2)
TBL + 4
④ address 0x A08

every case address is 4 bytes away ⇒ 1 word

Jump table provided in exam

Jump Table Example (Conditional Execution)

Cascaded if-else

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
}
```

```
      LDR R1, [R2]
      CMP R1, #0
      MOVEQ R0,#70
      CMP R1, #1
      MOVEQ R0,#68
      CMP R1, #2
      MOVEQ R0,#67
      CMP R1, #3
      MOVEQ R0, #66 ;'B'
RES   STR R0, [R2,#4]
      .
      .
      .
```

Worst case = Best case = 10 instructions

Jump Table

```
      LDR R1, [R2]
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
0xA00 C1   MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents	
TBL + 0	0xA00	
TBL + 4	0xA08	
TBL + 8	0xA10	
:		Jump Table

7

⇒ more compact

Jump Table Example (Default)

Cascaded if-else

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
Default:
G='X';
}
```

```
      LDR R1, [R2]
      CMP R1, #0
      BEQ C1
      CMP R1, #1
      BEQ C2
      CMP R1, #2
      BEQ C3
      CMP R1,#3
      BEQ C4
      MOV R0, #88 ;'X'
RES   STR R0,[R2,#4]
      .
      .
C1    MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
C4    MOV R0, #66; 'B'
      B RES
```

Jump Table

```
      LDR R1, [R2]
      MOV R0, #88; 'X'
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
0xA00 C1   MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents	
TBL + 0	0xA00	
TBL + 4	0xA08	
TBL + 8	0xA10	
:		Jump Table

8

problem with this is that even if Larger than 3, still executed

(worst case in this case is 3)

The more cases, the worst case will need more instructions

Jump Table Example (Default)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
Default:
G='X';
}
```

Cascaded if-else

```
        LDR R1, [R2]
        CMP R1, #0
        BEQ C1
        CMP R1, #1
        BEQ C2
        CMP R1, #2
        BEQ C3
        CMP R1,#3
        BEQ C4
        MOV R0, #88 ;'X'
RES     STR R0,[R2,#4]
        .
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
C4      MOV R0, #66; 'B'
        B RES
```

Jump Table

```
        LDR R1, [R2]
        MOV R0, #88; 'X'
        CMP R1, #3
        BGT RES
        ADR R3, TBL
        LDR PC, [R3,R1, LSL #2]
0xA00 RES STR R0, [R2,#4]
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

9

Example

- Create a subroutine that estimates the grading statistics of grades
  - Input: array of grades: 'A','B','C','D', passes in characters
  - The array terminates with a grade of 0
- Calculate statistics
  - Number of top grading marks
  - Number of B-grading marks
  - Other passing grades
- Pass address of grade array in stack, return three parameters also in stack

10

Example

- Create a subroutine
  grades
  - Input: array of grade
  - The array terminates
- Calculate statistics
  - Number of top gradi
  - Number of B-gradin
  - Other passing grade

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                          *top++;
                          break;
                   case ('B'):
                          *Bgrading++;
                          break;
                   case ('C'):
                          *other++;
                          break;
                   case ('D'):
                          *top++;
                          break;
                   default:
                          done=1;
        }while (done==0);
}
```

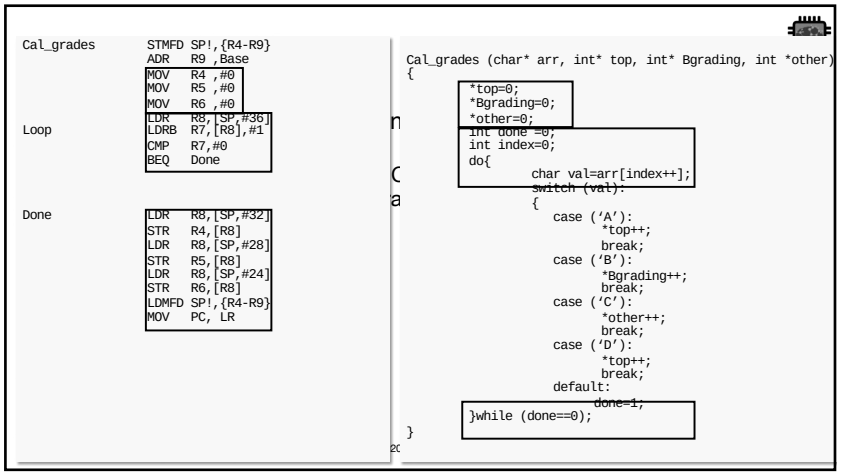
11

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0

                LDMFD SP!,{R4-R9}
                MOV   PC, LR
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                          *top++;
                          break;
                   case ('B'):
                          *Bgrading++;
                          break;
                   case ('C'):
                          *other++;
                          break;
                   case ('D'):
                          *top++;
                          break;
                   default:
                          done=1;
        }while (done==0);
}
```

12

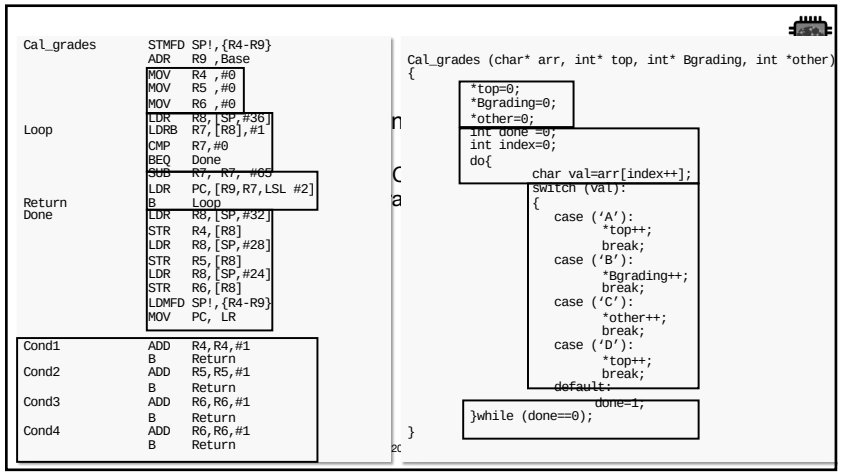


```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done

Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                          *top++;
                          break;
                   case ('B'):
                          *Bgrading++;
                          break;
                   case ('C'):
                          *other++;
                          break;
                   case ('D'):
                          *top++;
                          break;
                   default:
                          done=1;
        }while (done==0);
}
```

13

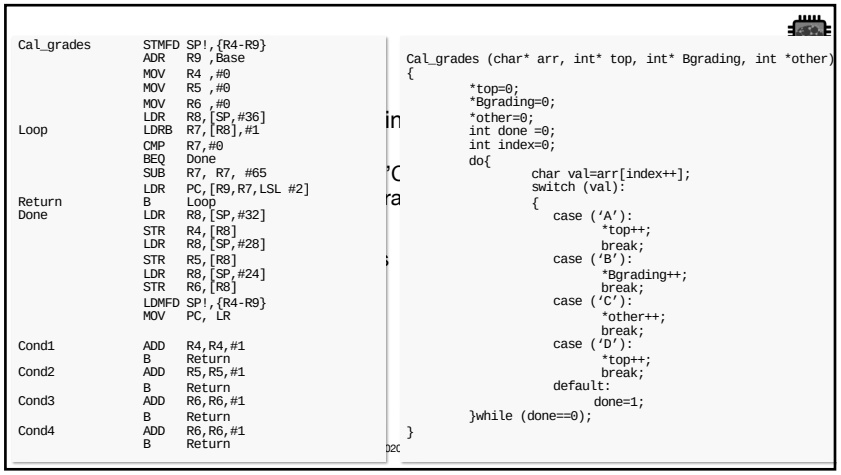


```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done
                SUB   R7, R7, #65
                LDR   PC,[R9,R7,LSL #2]
Return          B     Loop
Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR

Cond1           ADD   R4,R4,#1
                B     Return
Cond2           ADD   R5,R5,#1
                B     Return
Cond3           ADD   R6,R6,#1
                B     Return
Cond4           ADD   R6,R6,#1
                B     Return
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                          *top++;
                          break;
                   case ('B'):
                          *Bgrading++;
                          break;
                   case ('C'):
                          *other++;
                          break;
                   case ('D'):
                          *top++;
                          break;
                   default:
                          done=1;
        }while (done==0);
}
```

14



```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done
                SUB   R7, R7, #65
                LDR   PC,[R9,R7,LSL #2]
Return          B     Loop
Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR

Cond1           ADD   R4,R4,#1
                B     Return
Cond2           ADD   R5,R5,#1
                B     Return
Cond3           ADD   R6,R6,#1
                B     Return
Cond4           ADD   R6,R6,#1
                B     Return
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                          *top++;
                          break;
                   case ('B'):
                          *Bgrading++;
                          break;
                   case ('C'):
                          *other++;
                          break;
                   case ('D'):
                          *top++;
                          break;
                   default:
                          done=1;
        }while (done==0);
}
```

15

Switch – Random & Wide Values

- If cases are random wide range values, a **fork algorithm** is used to speed up the average search time and avoid testing every case (e.g. when **x** = 1000).

```
switch(x)
{
case 1:
 {S0};
 break;

case 10:
 {S1};
 break;

case 100:
 {S2};
 break;

case 1000:
 {S3};
 break;
}
```

- Due to the wide value spread, the **jump table size** will be **too large**. A cascade of if-else-if comparisons is more efficient.

```
if(x == 1)
 {S0};
else if(x == 10)
 {S1};
else if(x == 100)
 {S2};
else if(x == 1000)
 {S3};
```

standard if-else-if implementation

```
if(x <= 10) {
   if(x == 1)
    {S0};
   else if(x == 10)
    {S1};
}
else {
   if(x == 100)
    {S2};
   else if(x == 1000)
    {S3};
}
```

forked if-else-if implementation

16

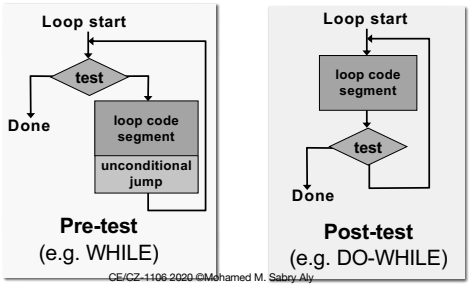
Tested?

Loops

Loop constructs are distinguished by the position of their conditional test.

Pre-test loop may **never execute** its loop code segment.

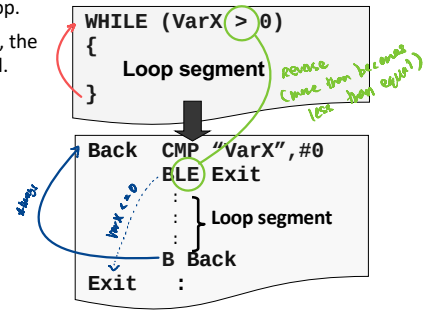
Post-test loop executes the loop segment **at least once**.



17

WHILE Implementation

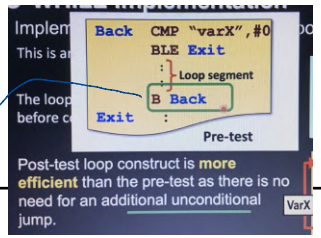
- Implementation of the **WHILE** loop constructs:
- This is an example of a **pre-test** loop.
- If the condition (**VarX > 0**) is false, the loop segment is not executed at all.



Implementation of WHILE in ARM assembly language

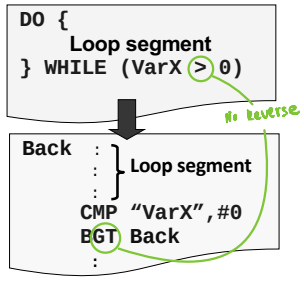
Note: **VarX** is variable. you will need to load it to a register.

18



DO-WHILE Implementation

- Implementation of the **DO-WHILE** loop constructs
- This is an example of a **post-test** loop.
- The loop segment is executed at least once before condition is tested.
- Post-test loop construct is **more efficient** than the pre-test as there is no need for an additional unconditional jump.



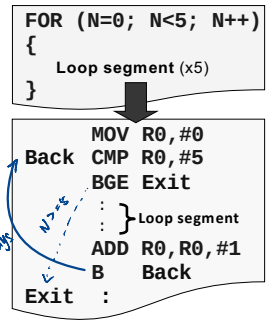
Implementation of DO- WHILE in ARM assembly language

Note: **VarX** is variable. you will need to load it to a register.

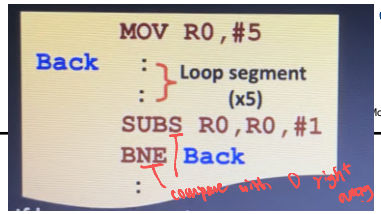
19

FOR Implementation

- Implementation of **FOR** loop constructs:
- The FOR loop is a **pre-test** loop that evaluates the condition first before executing loop segment.
- If loop segment is executed and count **N** is not used in loop segment, some optimizing compilers implement the **FOR** loop using a **post-test** with **decrement** & **test for zero**.



Implementation of FOR in ARM assembly language



20

Example: Summation 1 to N

```
Sum =0;
for(i=0;i<=N;i++)
    Sum=Sum+i;
```

Pre-test

```
        MOV     R2,#0
        MOV     R0,#0
Loop    CMP     R2,#N
        BGT     END
        ADD     R0,R0,R2
        ADD     R2,R2,#1
        B       Loop
END     STR     R0,["sum"]
```

Post-test

```
        MOV     R2, #N
        MOV     R0,#0
Loop    ADD     R0,R0,R2
        SUBS    R2,R2,#1
        BNE     Loop
        STR     R0,["sum"]
```


21

Example: Summation 1 to N

```
Sum =0;
for(i=0;i<=N;i++)
    Sum=Sum+i;
```

Pre-test

```
        MOV     R2,#0
        MOV     R0,#0
Loop    CMP     R2,#N
        BGT     END
        ADD     R0,R0,R2
        ADD     R2,R2,#1
        B       Loop
END     STR     R0,["sum"]
```

Post-test

```
        MOV     R2, 0
        MOV     R0,#0
Loop    ADD     R0,R0,R2
        ADD     R2,R2,#1
        CMP     R2,#5
        BLE     Loop
        STR     R0,["sum"]
```


22

Summary

- **SWITCH** constructs can be implemented efficiently depending nature of the case values.
  - Narrow consecutive values can benefit from a jump table.
  - Forked if-else-if can be used with wide ranged values.
- **Post-test** loops are **more efficient** that pre-test loops for the same loop segment.
- With optimised compilers, the low-level code produced may not tally directly with the high-level operations. (e.g. loop increments may be implemented as decrements for better code efficient).

23

22/2 Lecture

Pretest is used in both loop constructs — stay looping if (x >= 5) [x = x-1]

Review Q2 - Loop

What is the equivalent C code segment for the following ARM code.

```
Back  CMP R0,#5
      BLT Next
      SUB R0,R0,#1
      B Back
Next  :
```

always; x<5

(1) ✗
```
FOR (X=5; X>0; X--) {
   }
```
loop counter initialization in for loop — no initialization here

(2) ✓
```
WHILE (X >= 5) {
   X = X - 1;
   }
```

(3) ✗
```
WHILE (X < 5) {
   X = X - 1;
   }
```
"stay in loop if <5" X ⇒ should be REVERSE OF BLT

(4) ✗
```
DO {
   X = X -1;
   } WHILE (X >= 5)
```
post-test loop ⇒ Test condition after loop body

discard right away

discard cause it is post-test