CZ1006
CE1006

Control Flow Constructs

A/P Goh Wooi Boon

Control Flow Constructs

Conditional Constructs

Learning Objectives (6a)

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

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
        JGT DoIF
        JMP Skip
DoIF    {S1}
Skip
```

  - 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
        JLE Skip
        {S1}
Skip
```

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

Past Code Example (Review)
Find Largest Number

```
       MOV   R0,#0x100     ;setup pointer to first array element
       MOV   R1,#9
       MOV   R3,[R0]
Loop   INC   R0
       MOV   R2,[R0]       ;get next no. in array
       CMP   R2,R3         ;compare R3 and R2 (i.e. R2-R3)
       JLO   Skip          ;branch if R2 <  current max (i.e. R3)
       MOV   R3,R2         ;update current max. in R3 with R2
Skip   DEC   R1            ;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
}
```

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).

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.

```
          CMP a,#3
          JNE DoElse
          {S1}
          JMP Skip
DoElse    {S2}
Skip      :
```

Reversing test condition

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

Convert to assembly code

```
          CMP a,#3
          JEQ DoIf
          {S2}
          JMP Skip
DoIf      {S1}
Skip      :
```

IF-ELSE implementation in low-level assembly

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.

Compound OR Conditions

- How are compound OR conditions handled?

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

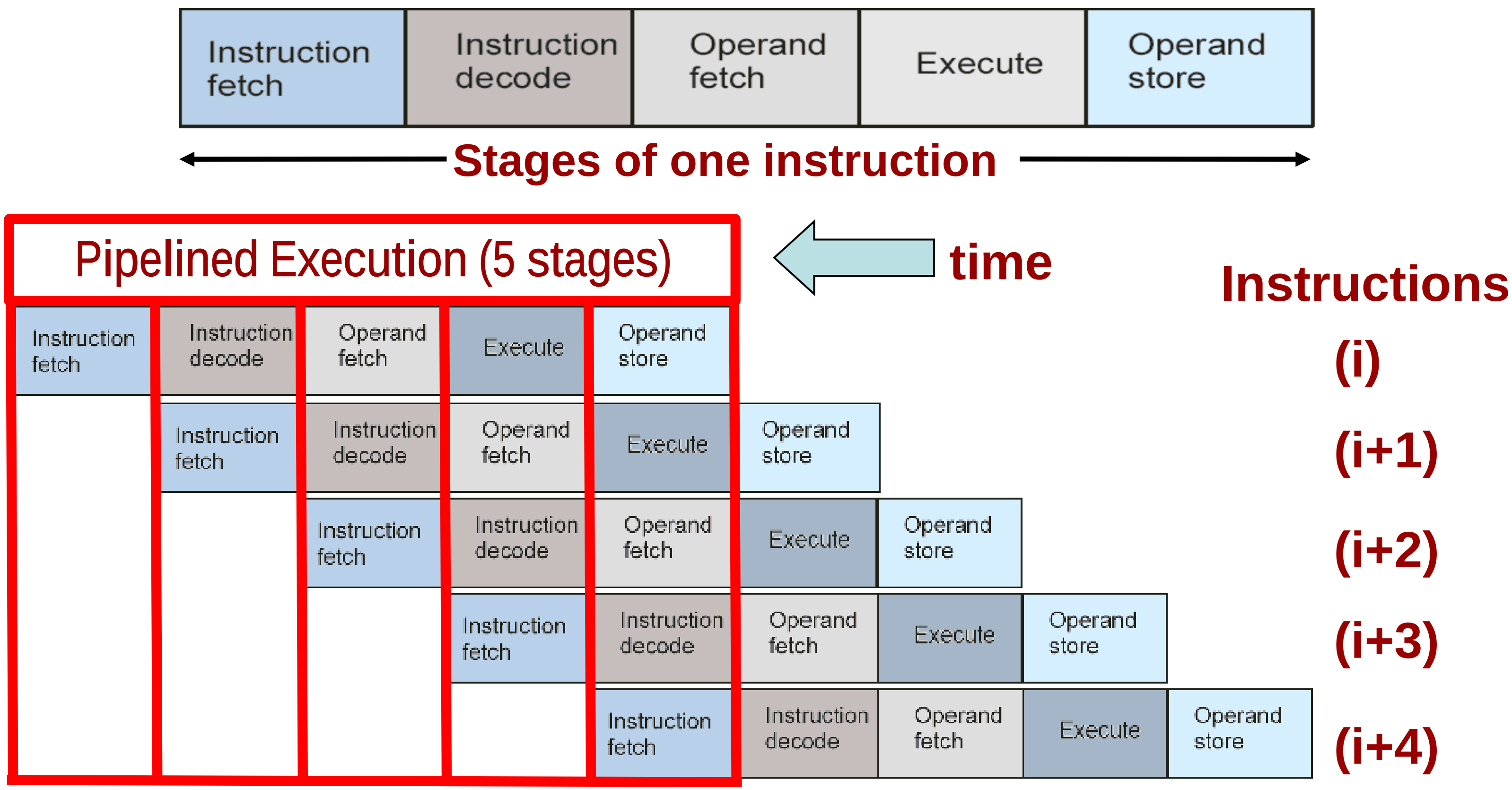
- Most compilers eliminate an unconditional jump at the end of the compound OR series by **reversing** the **last conditional** test.

```
      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.

Pipelining and Execution

- Pipelining keeps all parts of the CPU busy by partially executing several different instructions simultaneously, each at different stage of completion.



- Non-sequential execution reduces pipeline efficiency.

Branchless Logic

- Branchless logic avoid using conditional jump instructions when implementing logical constructs.
  - `Jcc` 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 [X],#0x002
ROR [X]
```

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

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;
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    ......           ;
```

  - It can be replaced using conditional execution instructions.

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

Summary

- The **IF** and **ELSE** constructs are implemented using one or more conditional jump (`Jcc`).
  - Using the **reverse** conditional test can help the IF construct execute more efficiently.
- 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.

Control Flow Constructs

Switch and Loop Constructs

Learning Objectives (6b)
1. Describe how SWITCH constructs can be implemented efficiently for consecutive narrow and random wide cases. 2. Contrast the implementations of pre and post-test loop constructs like WHILE, DO-WHILE and FOR.

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

Switch – Running & Narrow Values

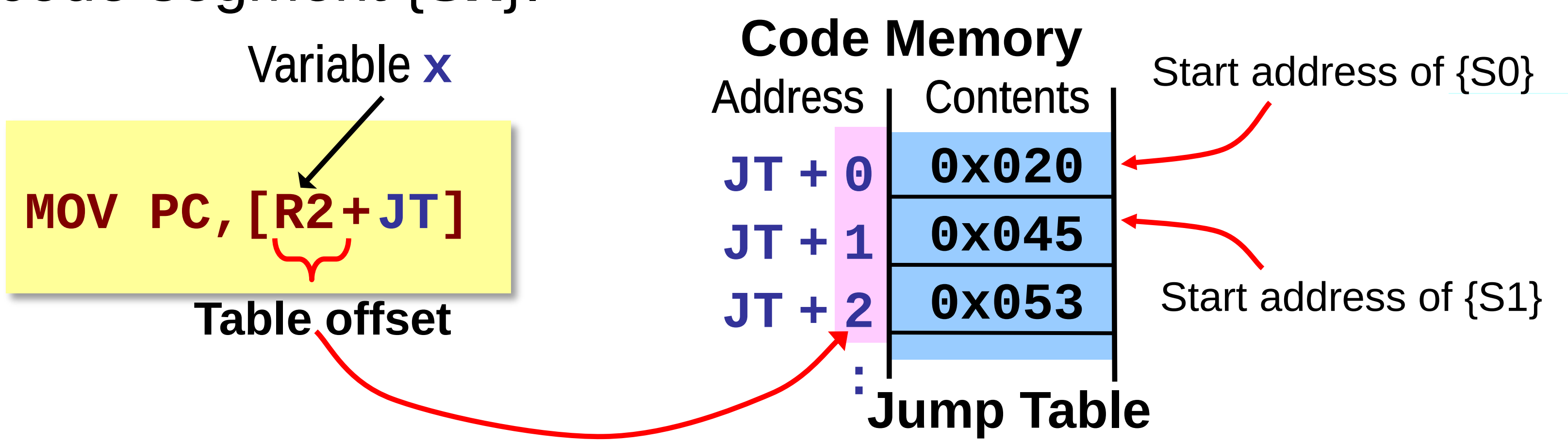
- If cases are consecutive narrow range values, a **Jump Table** is used to avoid testing each case in turn.
  - Jump table contains list of **start addresses** for the code segments that is associated with each case values.
  - Var **x** on which the switch is decided, acts as an **offset** into the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

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

case 1:
  {S1};
  break;

case 2:
  {S2};
  break;

case 3:
  {S3};
}
```



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.

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};
```

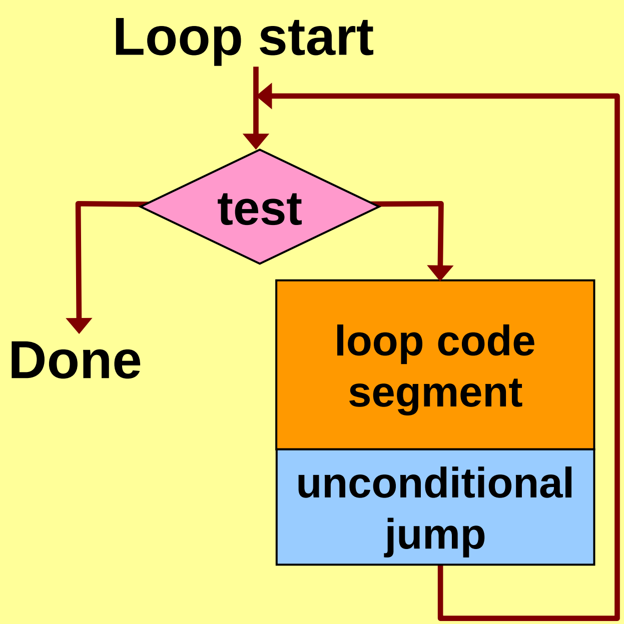
standard if-else-if implementation

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

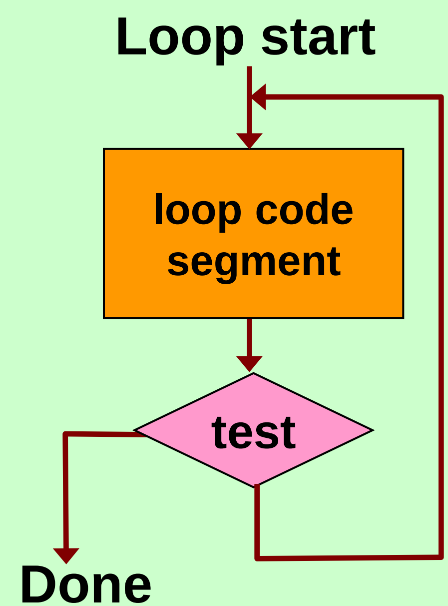
forked if-else-if implementation

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**.



Pre-test
(e.g. WHILE)



Post-test
(e.g. DO-WHILE)

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.

```
WHILE (VarX > 0)
{
     Loop segment
}
```

```
Back   CMP [VarX],#0
       JLE Exit
        :
        :   Loop segment
        :
       JMP Back
Exit    :
```

Implementation of WHILE in VIP assembly language

Note: `VarX` is an address label. In VIPAS, you will need to replace it with a numeric address value.

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.

```
DO {
      Loop segment
} WHILE (VarX > 0)
```

```
Back    :
        :     Loop segment
        :
     CMP [VarX],#0
     JGT Back
        :
```

Implementation of DO- WHILE in VIP assembly language

Note: `VarX` is an address label. In VIPAS, you will need to replace it with a numeric address value.

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**.

```
FOR (N=0; N<5; N++)
{
    Loop segment (x5)
}
```

```
        MOV [N],#0
Back    CMP [N],#5
        JGE Exit
         :  } Loop segment
         :
        INC [N]
        JMP Back
Exit     :
```

Implementation of FOR in VIP assembly language

Note: **N** is an address label. In VIPAS, you will need to replace it with a numeric address value.

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).