

Notes: Control Structures

A. Why?

Even assembler programs can be written using the basic control constructs: Sequence, Conditional, and Iterative, so it's good to know how to implement these constructs in assembler.

B. Outcomes

By the end of class you should:

- Know how to implement simple `if NZP/while NZP` structures.
- Know how to implement logical and and or (C/Java's `&&` and `||` operators).
- Know how to implement a jump table.

C. Implementing if-then Statements

- Let's define a "simple test" to be one that uses the condition codes N, Z, or P.
 - Testing for *reln* 0 (where *reln* is `<`, `≤`, `=`, `≠`, `>` or `≥`) is simple.
 - To test for *x reln y*, we test *x-y reln 0*.
- Let "not N/Z/P" stand for the opposite of whatever N/Z/P test we want.
 - E.g. say we want to branch on NZ; then "BR not NZ" is "BRP"
- Let's define "if NZP" and "while NZP" to mean `if/while` a simple test.
- `if NZP then TrueBranch` is represented as

```
Perform N/Z/P test
BR not N/Z/P LabelAfterTrueBranch
```

- E.g. "`if R1 < 0 then R0 ← 0`" is

```
AND R1, R1, #-1 ; Inspect value in R1
BRZP LABEL      ; Skip if R1 not negative
AND R0, R0, #0  ; Set R0 ← 0
```

```
LABEL .....
```

D. Implementing if-else Statements

- For a simple `if-else`, we need two labels (for the false branch and after the `if-else`) and two branches (one to jump to the false branch and one to jump around the false branch).
- `if NZP then TrueBranch else FalseBranch` is represented as

```
Perform N/Z/P test
BR (not N/Z/P) FALSEBR
```

```

    True Branch code ...
...
    BR AFTERIF
FALSEBR False Branch code ...
...
AFTERIF Code after the if-else ...

```

E. Implementing || (Short-circuiting Logical Or)

- In C/C++/Java, the || operator is a short-circuiting logical disjunction: We evaluate *Test1* || *Test2* by first checking *Test1*; if it's true, then the disjunction is true and we don't bother checking *Test2*. If *Test1* is false, then we check *Test2* and use the result as the result of the disjunction.

- `if (simpleTest1 || simpleTest2) then TrueBranch else FalseBranch` is represented using a nested if-else:

```

    if simpleTest1 then TrueBranch
    else if SimpleTest2 then TrueBranch
    else FalseBranch

```

- In the assembler code, we won't duplicate the true branch, of course; we'll branch to a label for it:

```

    Perform simpleTest1
    BR(simpleTest1 N/Z/P) TRUEBR
    Perform simpleTest2
    BR(not simpleTest2 N/Z/P) FALSEBR
TRUEBR True Branch ...
....
    BR AFTERIF
FALSEBR False Branch ...
....
AFTERIF Code after the if-else...

```

F. Implementing && (Short-circuiting Logical And)

- In C/C++/Java, the && operator is a short-circuiting logical conjunction: We evaluate *Test1* && *Test2* by first checking *Test1*; if it's false, then the conjunction is false and we don't bother checking *Test2*. If *Test1* is true, then we check *Test2* and use the result as the result of the conjunction.

- `if (simpleTest1 && simpleTest2) then TrueBranch else FalseBranch` is represented using a nested if-else:

```

if not simpleTest1 then FalseBranch
else if not SimpleTest2 then FalseBranch
else TrueBranch

```

- Again, in the assembler code, we won't duplicate a branch, of course; we'll branch to a label for it:

```

        Perform simpleTest1
BR(not simpleTest1 N/Z/P) FALSEBR
        Perform simpleTest2
BR(not simpleTest2 N/Z/P) FALSEBR
        True Branch ...
....
BR AFTERIF
FALSEBR False Branch ...
....
AFTERIF Code after the if-else...

```

G. Implementing while Loops

- `while NZP do Body` is represented as

```

TOP    Perform N/Z/P test
        BR not N/Z/P AFTER
        Body code ...
...
BR TOP
AFTER  Code after loop....

```

- I'll leave `while simpleTest1 && or || simpleTest2 do Body` for you to figure out.

H. Implementing Switch Statements With Jump Tables

- Switch/case statements can be translated two ways.
- The obvious way is using `if-else if-else if-....`
- The non-obvious way uses a **jump table**, which is a table of addresses indexed by the case value. Basically, we'll set up a table J where $J[0]$ is the address to go to if our test value is 0, $J[1]$ is the address to go to if our test value is 1, and so on.
- E.g., say we have `switch (R1) case 0: Arm0; case 1: Arm1; case 2: Arm2.`
 - (Let's assume that we know for sure that $R1 = 0, 1, \text{ or } 2.$)
 - Assume that the code for $Arm0$ is at label LABEL0 and so on.
 - Then our table J looks like

```

JMPTAB .FILL LABEL0
        .FILL LABEL1
        .FILL LABEL2

```

- To execute the `switch` statement, we can use the following code. (Let `tmp1` and `tmp2` be temporary registers, and assume that the switch value is stored at `SWITCHVAL`. In the comments, “`pt`” means “points to”):

```

LEA tmp1, JMPTAB      ; tmp1 pt jump table
LD  tmp2, SWITCHVAL   ; tmp2 ← value to switch on
ADD tmp1, tmp1, tmp2  ; tmp1 pt Jump table[switch val]
LDR tmp1, tmp1, #0    ; tmp1 pt Arm for switch val
JMP tmp1              ; Go to Arm for switch val

```

- If each case arm has a `break`; at its end, we can represent this by ending each arm’s code with a branch to the code after the switch.
- If you actually want to represent the “fall through to the next case” behavior in C/C++/Java `switch` statements, then the code for the arms has to be generated in the same order as the cases appear so that you can continue from the code for case 0 (say) to case 1 (say).