Computer Organisation & Program Execution 2021

6

Control Structures

Uwe R. Zimmer - The Australian National University

References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 "Instructions: Language of the Computer"
ARM edition, Morgan Kaufmann 2017

Imperative Programming

Essential control structures for all imperative programming languages are:

- Conditionals: `if`, `case`, `switch`, …
- Open Loops: `while`, `repeat`, …
- Bound Loops: `for`, `foreach`, `forall`, …
- Procedures and Functions (already covered)

☞ How do we create those basic control structures in Assembly?

Functional programming languages are based on functions, but also on conditional expressions.

☞ How do those control structures in programming languages translate into Assembly?

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then
   Register_3 := 1;
else
   Register_3 := 0;
end if;
```

… same structure? …
many syntax versions?

```
if (register1 == register2) {
   register3 = 1;
} else {
   register3 = 0;
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of
   True  -> 1
   False -> 0
```

```
if register1 == register2:
   register3 = 1
else:
   register3 = 0
```

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then
   Register_3 := 1;
else
   Register_3 := 0;
end if;
```

```
if (register1 == register2) {
   register3 = 1;
} else {
   register3 = 0;
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of
   True  -> 1
   False -> 0
```

```
if register1 == register2:
   register3 = 1
else:
   register3 = 0
```

1. an expression (if)
2. a boolean condition (if)
3. code for True (then)
4. code for False (else)

How do either of those look in assembly?

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
    cmp   r1, r2    ; 1. Instructions to generate status flags
    beq   then      ; 2. Branch depending on the status flags
    mov   r3, #0    ; 4. Instructions for the else branch
    b     end_if
then:
    mov   r3, #1    ; 3. Instructions for the then branch
end_if:
```

It seems there are *three distinguishable code sections* and *one status flag condition*.

☞ Can we form a general pattern for this?

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
    .macro if condition_code condition then_code else_code
    \condition_code
    b\condition then
    \else_code
    b     end_if
then:
    \then_code
end_if:
    .endm
```

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
    .macro if condition_code condition then_code else_code
    \condition_code
    b\condition then
    \else_code
    b     end_if
then:
    \then_code
end_if:
    .endm
```

We might need a lot of those, hence the labels need to be unique to each if-else block.

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
    .macro if condition_code condition then_code else_code
    \condition_code
    b\condition then\@
    \else_code
    b     end_if\@
then\@:
    \then_code
end_if\@:
    .endm
```

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
    .macro if condition_code condition then_code else_code
    \condition_code
    b\condition then\@
    \else_code
    b     end_if\@
then\@:
    \then_code
end_if\@:
    .endm
```

We can now write:

```
if "cmp r1, r2", eq, "mov r3, #1", "mov r3, #0"
```

… in the general case (with lots of code in each part)
we could create macros for the individual sections as well, so we can e.g. write:

```
if compare_r1_r2, eq, load_1_to_r3, load_0_to_r3
```

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then
   Register_3 := 1;
else
   Register_3 := 0;
end if;
```

```
if (register1 == register2) {
   register3 = 1;
} else {
   register3 = 0;
}
```

```
.macro if condition_code condition then else
   \condition_code
   b\condition then
   \else
   b      end_if
then:
   \then
end_if:
   .endm
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of
   True  -> 1
   False -> 0
```

```
if register1 == register2:
   register3 = 1
else:
   register3 = 0
```

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then
   Register_3 := 1;
else
   Register_3 := 0;
end if;
```

```
if "cmp r1, r2", eq, "mov r3, #1", "mov r3, #0"
```

```
if (register1 == register2) {
   register3 = 1;
} else {
   register3 = 0;
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of
   True  -> 1
   False -> 0
```

```
if register1 == register2:
   register3 = 1
else:
   register3 = 0
```

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then
   Register_3 := 1;
else
   Register_3 := 0;
end if;
```

```
if (register1 == register2) {
   register3 = 1;
} else {
   register3 = 0;
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of
   True  -> 1
   False -> 0
```

```
if register1 == register2:
   register3 = 1
else:
   register3 = 0
```

```
   cmp   r1, r2
   beq   then
   mov   r3, #0
   b     end_if
then:
   mov   r3, #1
end_if:
```

Computational complexity: $\Theta(1)$

Loops – FOR

```
for Register_1 in 1..100 loop
   Register_3 := Register_3 + Register_1;
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
   register3 += register1;
}
```

```
for register1 in range (1, 101):
   register3 += register1
```

What are the components?

```
for Register_1 := 1 to 100 do
   Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
   Register_3 = Register_3 + Register_1
end do
```

```
for Register_1 in 1..100 do
   Register_3 += Register_1;
```

Loops – FOR

```
for Register_1 in 1..100 loop
   Register_3 := Register_3 + Register_1;
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
   register3 += register1;
}
```

```
for register1 in range (1, 101):
   register3 += register1
```

```
for Register_1 := 1 to 100 do
   Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
   Register_3 = Register_3 + Register_1
end do
```

```
for Register_1 in 1..100 do
   Register_3 += Register_1;
```

1. an **index**
2. a **start** value
3. an **end** value
4. **code** inside loop

How do either of those look in assembly?

Loops – FOR

Assuming the values have already been transferred from memory into registers:

```
    mov   r1, #1    ; set index to start value
for:
    cmp   r1, #100  ; check whether it went beyond its end value
    bgt   end_for   ; if so, stop the loop
    add   r3, r1    ; do the work
    add   r1, #1    ; increment the index
    b     for
end_for:
```

We can find the index, the start and end values and the body code.

☞ Can we form a general pattern for this?

Loops – FOR

Assuming the values have already been transferred from memory into registers:

```
    .macro for register, from, to, body
    mov   \register, #\from
for\@:
    cmp   \register, #\to
    bgt   end_for\@
    \body
    add   \register, #1
    b     for\@
end_for\@:
    .endm
```

Loops – FOR

Assuming the values have already been transferred from memory into registers:

```
    .macro for register, from, to, body
    mov   \register, #\from
for\@:
    cmp   \register, #\to
    bgt   end_for\@
    \body
    add   \register, #1
    b     for\@
end_for\@:
    .endm
```

We can now write:

```
for r1, 1, 100 "add r3, r1"
```

… in the general case (with lots of code inside the loop or multiple loops):

```
for r1, 1, 100, loop_body
for r1, 1, 100, "for r2, 1, 100, loop_body"
```

Loops – FOR

```
for Register_1 in 1..100 loop
   Register_3 := Register_3 + Register_1;
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
   register3 += register1;
}
```

```
for register1 in range (1, 101):
   register3 += register1
```

```
for Register_1 := 1 to 100 do
   Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
   Register_3 = Register_3 + Register_1
end do
```

```
for Register_1 in 1..100 do
   Register_3 += Register_1;
```

```
   .macro for register, from, to, body
   mov    \register, #\from
for\@:
   cmp    \register, #\to
   bgt    end_for\@
   \body
   add    \register, #1
   b      for\@
end_for\@:
   .endm
```

Loops – FOR

```
for Register_1 in 1..100 loop
   Register_3 := Register_3 + Register_1;
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
   register3 += register1;
}
```

```
for register1 in range (1, 101):
   register3 += register1
```

```
for Register_1 := 1 to 100 do
   Register_3 := Register_3 + Register_1;
```

```
for r1, 1, 100 "add r3, r1"
```

```
do Register_1 = 1, 100
   Register_3 = Register_3 + Register_1
end do
```

```
for Register_1 in 1..100 do
   Register_3 += Register_1;
```

Loops – FOR

```
for Register_1 in 1..100 loop
   Register_3 := Register_3 + Register_1;
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
   register3 += register1;
}
```

```
for register1 in range (1, 101):
   register3 += register1
```

```
for Register_1 := 1 to 100 do
   Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
   Register_3 = Register_3 + Register_1
end do
```

```
for Register_1 in 1..100 do
   Register_3 += Register_1;
```

```
    mov     r1, #1
for:
    cmp     r1, #100
    bgt     end_for
    add     r3, r1
    add     r1, #1
    b       for
end_for:
```

Computational complexity: $\Theta(n)$

Loops – WHILE

```
while Register_1 < 100 loop
   Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
   register1 = register1 * register1;
}
```

```
while register1 < 100:
   register1 = register1 ** 2
```

What are the components?

```
while Register_1 < 100 do
   Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do
   Register_1 = Register_1 ** 2
enddo
```

```
while (Register_1 < 100) {
   Register_1 = Register_1 ** 2;
}
```

Loops – WHILE

```
while Register_1 < 100 loop
   Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
   register1 = register1 * register1;
}
```

```
while register1 < 100:
   register1 = register1 ** 2
```

```
while Register_1 < 100 do
   Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do
   Register_1 = Register_1 ** 2
enddo
```

```
while (Register_1 < 100) {
   Register_1 = Register_1 ** 2;
}
```

1. an expression (if)
2. a boolean condition (if)
3. code inside the loop

Loops – WHILE

```
    b       while_condition
while:
    mul     r1, r1              ; 3. Loop body
while_condition:
    cmp     r1, #100            ; 1. Instructions to generate status flags
    blt     while               ; 2. Branch depending on the status flags
```

☞ Can we form a general pattern for this?

Loops – WHILE

```
    .macro while while_expression, while_condition, body
    b     while_condition\@
while\@:
    \body
while_condition\@:
    \while_expression
    b\while_condition while\@
    .endm
```

We can now write:

```
while "cmp  r1, #100", lt, "mul r1, r1"
```

… try to re-write our power functions from the previous chapter with the macros you have now.

Loops – WHILE

```
while Register_1 < 100 loop
   Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
   register1 = register1 * register1;
}
```

```
while register1 < 100:
   register1 = register1 ** 2
```

```
while Register_1 < 100 do
   Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do
   Register_1 = Register_1 ** 2
enddo
```

```
while (Register_1 < 100) {
   Register_1 = Register_1 ** 2;
}
```

```
.macro while while_expression, while_condition, body
   b     while_condition\@

while\@:
   \body

while_condition\@:
   \while_expression
   b\while_condition while\@
   .endm
```

Loops – WHILE

```
while Register_1 < 100 loop
   Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
   register1 = register1 * register1;
}
```

```
while register1 < 100:
   register1 = register1 ** 2
```

```
while "cmp r1, #100", lt, "mul r1, r1"
```

```
while Register_1 < 100 do
   Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do
   Register_1 = Register_1 ** 2
enddo
```

```
while (Register_1 < 100) {
   Register_1 = Register_1 ** 2;
}
```

Loops – WHILE

```
while Register_1 < 100 loop
   Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
   register1 = register1 * register1;
}
```

```
while register1 < 100:
   register1 = register1 ** 2
```

```
while Register_1 < 100 do
   Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do
   Register_1 = Register_1 ** 2
enddo
```

```
while (Register_1 < 100) {
   Register_1 = Register_1 ** 2;
}
```

```
   b       while_condition
while:
   mul     r1, r1
while_condition:
   cmp     r1, #100
   blt     while
```

Computational complexity: *Undefined*

Conditionals – CASE (indexed)

```
type Colour is (Red, Green, Blue);
```

These values can be represented by (which is also the default in most systems)

```
for Colour use (
   Red   => 0,
   Green => 1,
   Blue  => 2);
```

Assuming that `Register_1` is associated with this type, we can then expect a highly efficient implementation of a case construct such as:

```
case Register_1 is
   when Red   => Register_2 := Register_3;
   when Green => Register_2 := Register_4;
   when Blue  => Register_2 := Register_5;
end case;
```

Conditionals – CASE (indexed)

A table based branching implementation of:

```
case Register_1 is
   when Red   => Register_3 := Register_2;
   when Green => Register_4 := Register_2;
   when Blue  => Register_5 := Register_2;
end case;
```

would look like:

Conditionals – CASE (indexed)

```
    tbh    [PC, r1, lsl #1]              ; PC used as base of branch table, r1 is index
branch_table:
    .hword (case_red   - branch_table)/2   ; case_red   16 bit offset
    .hword (case_green - branch_table)/2   ; case_green 16 bit offset
    .hword (case_blue  - branch_table)/2   ; case_blue  16 bit offset
case_red:
    mov    r3, r2                          ; Code for case Red
    b end_case
case_green:
    mov    r4, r2                          ; Code for case Green
    b end_case
case_blue:
    mov    r5, r2                          ; Code for case Blue
end_case:
```

The complexity of this operation is $\Theta(1)$, e.g. it is independent of the number of cases!
Can we generate this via a macro automatically in one line?, for instance as:

```
indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"
```

Conditionals – CASE (indexed)

.. yes, but as the number of cases is variable, we need to write this recursively:

```
    .macro indexed_case index case_body other_cases:vararg
    indexed_case_id \@, \index, “\case_body”, \other_cases  ; add a unique id
    .endm

    .macro indexed_case_id id index case_body other_cases:vararg
    tbh [pc, \index, lsl #1]
branch_table_\id:
    table_entry \id, i, “\case_body”, \other_cases  ; build up the table entries
    case_entry  \id, i, “\case_body”, \other_cases  ; add the codes with a label each
indexed_case_end_\id:
    .endm
```

…

The parts which are actually producing code are **highlighed**.

… recursive parts are following on the next page … hold on to something!

Conditionals – CASE (indexed)

…

```
    .macro table_entry id case_nr case_body other_cases:vararg
    .hword   (case_\id\()_\case_nr - branch_table_\id)/2
    .ifnb \other_cases
    table_entry \id, \case_nr\()i, \other_cases     ; still more entries to add
    .endif
    .endm

    .macro case_entry id case_nr case_body other_cases:vararg
case_\id\()_\case_nr:
    \case_body
    b     indexed_case_end_\id
    .ifnb \other_cases
    case_entry \id, \case_nr\()i, \other_cases      ; still more entries to add
    .endif
    .endm
```

… yes, this is a bit more involved than the previous macros, yet it is here to demonstrate that more complex and dynamic structures can also be macro generated.

Conditionals – CASE (indexed)

```
case Register_1 is
   when Red   => Register_3 := Register_2;
   when Green => Register_4 := Register_2;
   when Blue  => Register_5 := Register_2;
end case;
```

```
indexed_case r1, “mov r3, r2”, “mov r4, r2”, “mov r5, r2”
```

Conditionals – CASE (indexed)

```
case Register_1 is
   when Red   => Register_3 := Register_2;
   when Green => Register_4 := Register_2;
   when Blue  => Register_5 := Register_2;
end case;
```

```
   tbh    [PC, r1, lsl #1]
branch_table:
   .hword (case_red   - branch_table)/2
   .hword (case_green - branch_table)/2
   .hword (case_blue  - branch_table)/2
case_red:
   mov    r3, r2
   b end_case
case_green:
   mov    r4, r2
   b end_case
case_Blue:
   mov    r5, r2
end_case:
```

Computational complexity: $\Theta(1)$

Side remark: if you disassemble such a structure, it will look different.

☞ How and why?

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 :: Int -> Int -> Int -> Int
r0 r1 r2 r3
   | r1 <  r2  = r1
   | r1 >  r2  = r2
   | r1 == r2  = 0
   | otherwise = error "How did I get here?"
```

```
  switch (r1) {
    case 4 : r0 = r1;
      break;
    case 5 : r0 = r2;
      break;
    case 6 : r0 = 0;
  }
```

... similar structure? ...
many syntax versions?

```
r0 := (if    r1 < r2 then r1
       elsif r1 > r2 then r2
       elsif r1 = r2 then 0
       else Integer'Invalid);
```

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 :: Int -> Int -> Int -> Int
r0 r1 r2 r3
   | r1 <  r2  = r1
   | r1 >  r2  = r2
   | r1 == r2  = 0
   | otherwise = error "How did I get here?"
```

```
  switch (r1) {
    case 4 : r0 = r1;
      break;
    case 5 : r0 = r2;
      break;
    case 6 : r0 = 0;
  }
```

1. guards
2. guard conditions
3. guard expressions / statements

```
r0 := (if    r1 < r2 then r1
       elsif r1 > r2 then r2
       elsif r1 = r2 then 0
       else Integer'Invalid);
```

Conditionals – CASE (guarded expressions, list of conditions)

```
    cmp   r1, r2
    blt   case_a
    cmp   r1, r2
    bgt   case_b
    cmp   r1, r2
    beq   case_c
    b     end_case
case_a:
    mov   r0, r1
    b     end_case
case_b:
    mov   r0, r2
    b     end_case
case_c:
    mov   r0, #0
    b     end_case
end_case:
```

1. guards
2. guard conditions
3. guard expressions / statements

Generated by:

```
case "cmp r1, r2", lt, "mov r0, r1",
"cmp r1, r2", gt, "mov r0, r2",
"cmp r1, r2", gt, "mov r0, #0"
```

Conditionals – CASE (indexed)

This is again recursive to handle the variable number of cases:

```
    .macro case expression condition case_body other_cases:vararg
    case_id \@, “\expression”, \condition, “\case_body”, \other_cases
    .endm

    .macro case_id id expression condition case_body other_cases:vararg
    guards_rec \id, i, “\expression”, \condition, “\case_body”, \other_cases
    cases_rec  \id, i, “\expression”, \condition, “\case_body”, \other_cases
end_case_\id:
    .endm

…
```

The parts which are actually producing code are **highlighed**.

… and we still need to generate the list of guards, followed by the list of code sections.

Conditionals – CASE (indexed)

…

```
.macro guards_rec id case_nr expression condition case_body other_cases:vararg
\expression
b\condition case_\id\()_\case_nr
.ifnb \other_cases
guards_rec \id, \case_nr\()i, \other_cases
.else
b     end_case_\id
.endif
.endm

.macro cases_rec id case_nr expression condition case_body other_cases:vararg
case_\id\()_\case_nr:
\case_body
b     end_case_\id
.ifnb \other_cases
cases_rec \id, \case_nr\()i, \other_cases
.endif
.endm
```

Keep in mind:

Macro programming is pure textual replacement.

The result is a text which is then translated by the assembler into machine code.

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 :: Int -> Int -> Int -> Int
r0 r1 r2 r3
   | r1 <  r2  = r1
   | r1 >  r2  = r2
   | r1 == r2  = 0
   | otherwise = error "How did I get here?"
```

```
  switch (r1) {
    case 4 : r0 = r1;
      break;
    case 5 : r0 = r2;
      break;
    case 6 : r0 = 0;
  }
```

```
case "cmp r1, r2", lt, "mov r0, r1",
"cmp r1, r2", gt, "mov r0, r2",
"cmp r1, r2", gt, "mov r0, #0"
```

```
r0 := (if    r1 < r2 then r1
       elsif r1 > r2 then r2
       elsif r1 = r2 then 0
       else Integer'Invalid);
```

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 :: Int -> Int -> Int -> Int
r0 r1 r2 r3
   | r1 <  r2  = r1
   | r1 >  r2  = r2
   | r1 == r2  = 0
   | otherwise = error "How did I get here?"
```

```
  switch (r1) {
    case 4 : r0 = r1;
      break;
    case 5 : r0 = r2;
      break;
    case 6 : r0 = 0;
  }
```

```
r0 := (if    r1 < r2 then r1
       elsif r1 > r2 then r2
       elsif r1 = r2 then 0
       else Integer'Invalid);
```

```
    cmp   r1, r2
    blt   case_a
    cmp   r1, r2
    bgt   case_b
    cmp   r1, r2
    beq   case_c
    b     end_case
case_a:
    mov   r0, r1
    b     end_case
case_b:
    mov   r0, r2
    b     end_case
case_c:
    mov   r0, #0
    b     end_case
end_case:
```

Computational complexity: $O(n)$

Control Structures

```
while "cmp r1, #100", lt, "mul r1, r1"
```

```
    b       while_condition
while:
    mul     r1, r1
while_condition:
    cmp     r1, #100
    blt     while
```

```
if "cmp r1, r2", eq, "mov r3, #1", "mov r3, #0"
```

```
    cmp     r1, r2
    beq     then
    mov     r3, #0
    b       end_if
then:
    mov     r3, #1
end_if:
```

```
for r1, 1, 100 "add r3, r1"
```

```
    mov     r1, #1
for:
    cmp     r1, #100
    bgt     end_for
    add     r3, r1
    add     r1, #1
    b       for
end_for:
```

```
case "cmp r1, r2", lt, "mov r0, r1",
"cmp r1, r2", gt, "mov r0, r2",
"cmp r1, r2", gt, "mov r0, #0"
```

```
indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"
```

```
    tbh     [PC, r1, lsl #1]
branch_table:
    .hword  (case_red   - branch_table)/2
    .hword  (case_green - branch_table)/2
    .hword  (case_blue  - branch_table)/2
case_red:
    mov     r3, r2
    b end_case
case_green:
    mov     r4, r2
    b end_case
case_Blue:
    mov     r5, r2
end_case:
```

```
    cmp     r1, r2
    blt     case_a
    cmp     r1, r2
    bgt     case_b
    cmp     r1, r2
    beq     case_c
    b       end_case
case_a:
    mov     r0, r1
    b       end_case
case_b:
    mov     r0, r2
    b       end_case
case_c:
    mov     r0, #0
    b       end_case
end_case:
```

```
switch r1,
  4, "mov r0, r1",
  5, "mov r0, r2",
  6, "mov r0, #0"
```

```
indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"
```

```
    tbh     [PC, r1, lsl #1]
branch_table:
    .hword  (case_red   - branch_table)/2
    .hword  (case_green - branch_table)/2
    .hword  (case_blue  - branch_table)/2
case_red:
    mov     r3, r2
    b end_case
case_green:
    mov     r4, r2
    b end_case
case_Blue:
    mov     r5, r2
end_case:
```

```
    cmp     r1, #4
    beq     case_a
    cmp     r1, #5
    beq     case_b
    cmp     r1, #6
    beq     case_c
    b       end_case
case_a:
    mov     r0, r1
    b       end_case
case_b:
    mov     r0, r2
    b       end_case
case_c:
    mov     r0, #0
    b       end_case
end_case:
```

☞ You can form all common sequential control structures

(or generate them via macros if you wish)

(including function calls)

Summary

Control Structures

- **Assembler Macros**
  - Local labels
  - Recursive macros
- **Control Structures in machine code**
  - IF
  - WHILE
  - FOR
  - CASEs