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
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?
Control Structures

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

... same structure? ... many syntax versions?
### 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

How do either of those look in assembly?
Conditionals – IF-ELSE

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

```assembly
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

displays: 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:

```plaintext
.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;
}

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 := case register_1 == register_2 of
    True -> 1
    False -> 0

if register1 == register2:
    register3 = 1
else:
    register3 = 0
Conditionals – IF-ELSE

if \( \text{Register}_1 = \text{Register}_2 \) then
    \( \text{Register}_3 := 1 \);
else
    \( \text{Register}_3 := 0 \);
end if;

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

\( \text{Register}_3 := (\text{if \( \text{Register}_1 = \text{Register}_2 \) then 1 else 0}); \)

\( \text{register}_3 \) \( \text{register}_1 \) \( \text{register}_2 \) = case \( \text{register}_1 \) == \( \text{register}_2 \) of
    True \to 1
    False \to 0
if \( \text{register1} == \text{register2} \):
    \( \text{register3} = 1 \)
else:
    \( \text{register3} = 0 \)

\( \text{Computational complexity: } \Theta(1) \)
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

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

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

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

What are the components?
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;
end do

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

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

How do either of those look in assembly?
Loops – FOR

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

```assembly
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
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;
end do

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

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

.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;
do

for r1, 1, 100 “add r3, r1”

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

Control Structures

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;
end do

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

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

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

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

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;
enddo

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

while (Register_1 < 100) {
  Register_1 = Register_1 ** 2;
}
Loops – WHILE

while_condition:
  cmp  r1, #100
  blt  while

while:
  mul  r1, r1

3. Loop body

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 “\texttt{cmp r1, #100}”, \texttt{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
endwhile

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
  while_condition@
  while@:
  body
  while_condition@
  while_expression
  b\while_condition while@
.endm
Loops – WHILE

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

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

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

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

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

while “cmp r1, #100”, lt, “mul r1, r1”
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;
enddo

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

Computational complexity: Undefined
type Colour is (Red, Green, Blue);

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

```plaintext
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:

```plaintext
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:

```plaintext
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:

.branch_table (case_red - branch_table)/2 ; case_red 16 bit offset
.branch_table (case_green - branch_table)/2 ; case_green 16 bit offset
.branch_table (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:

```
.macros indexed_case index case_body other_cases
indexed_case_id @, \index, "\case_body", \other_cases ; add a unique id
.endm

.macros indexed_case_id id index case_body other_cases
.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
.endm

indexed_case_end\_id:
  .endm
```

... The parts which are actually producing code are highlighted.

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

Computational complexity: $\Theta(1)$

Side remark: if you disassemble such a structure, it will look different. How and why?

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:
Conditionals – CASE (guarded expressions, list of conditions)

\[ r_0 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ r_0 \ r_1 \ r_2 \ r_3 \]
\[ | \ r_1 < \ r_2 \ = \ r_1 \]
\[ | \ r_1 > \ r_2 \ = \ r_2 \]
\[ | \ r_1 == \ r_2 \ = \ 0 \]
\[ | \ \text{otherwise} = \text{error "How did I get here?"} \]

\[
\text{switch (r1) \{} \\
\text{case 4 : } r_0 = r_1; \\
\quad \text{break}; \\
\text{case 5 : } r_0 = r_2; \\
\quad \text{break}; \\
\text{case 6 : } r_0 = 0; \\
\text{\}}
\]

\[
r_0 := (\text{if } \ r_1 < \ r_2 \ \text{then } r_1 \ \\
\quad \text{elsif } r_1 > \ r_2 \ \text{then } r_2 \ \\
\quad \text{elsif } r_1 = \ r_2 \ \text{then } 0 \ \\
\quad \text{else Integer'Invalid});
\]
**Conditionals – CASE** (guarded expressions, list of conditions)

\[
r0 :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Int}
\]

\[
r0 \ r1 \ r2 \ r3
\]

| \ r1 < \ r2 \ = \ r1  \\
| \ r1 > \ r2 \ = \ r2  \\
| \ r1 == \ r2 \ = \ 0  \\
| \ otherwise = \ \text{error} \ "How \ did \ I \ get \ here?"

\[
\text{switch (r1)} \{
\text{case 4 : } r0 = \ r1;
\text{break;}
\text{case 5 : } r0 = \ r2;
\text{break;}
\text{case 6 : } r0 = \ 0;
\}
\]

\[
r0 : = (\text{if } \ r1 < \ r2 \ \text{then } \ r1
\text{elsif } \ r1 > \ r2 \ \text{then } \ r2
\text{elsif } \ r1 = \ r2 \ \text{then } \ 0
\text{else Integer'}\text{Invalid});
\]

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

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 highlighted.  

... 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)

\[
\begin{align*}
  r0 &:: \text{Int} \to \text{Int} \to \text{Int} \to \text{Int} \\
  r0 & r1 r2 r3 \\
  | \ r1 < \ r2 & = r1 \\
  | \ r1 > \ r2 & = r2 \\
  | \ r1 == \ r2 & = 0 \\
  | \ \text{otherwise} & = \text{error} \ "\text{How did I get here?}" \\
\end{align*}
\]

\[
\begin{align*}
  \text{switch (} r1 \text{)} \{ \\
  &\text{case } 4 : r0 = r1; \\
  &\quad \text{break;} \\
  &\text{case } 5 : r0 = r2; \\
  &\quad \text{break;} \\
  &\text{case } 6 : r0 = 0; \\
  \}
\end{align*}
\]

\[
\begin{align*}
  r0 &:: (\text{if } \ r1 < \ r2 \ \text{then } r1 \\
  &\quad \text{elsif } r1 > \ r2 \ \text{then } r2 \\
  &\quad \text{elsif } r1 == \ r2 \ \text{then } 0 \\
  &\quad \text{else } \text{Integer'Invalid});
\end{align*}
\]
Control Structures

Conditionals – CASE (guarded expressions, list of conditions)

\[
\text{r0} :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Int}
\]

\[
\text{r0 r1 r2 r3}
\]

\[
| \text{r1 < r2} = \text{r1} \\
| \text{r1 > r2} = \text{r2} \\
| \text{r1 == r2} = 0 \\
| \text{otherwise} = \text{error “How did I get here?”}
\]

\[
\text{switch (r1)} \{
\text{case 4 : r0 = r1; break;}
\text{case 5 : r0 = r2; break;}
\text{case 6 : r0 = 0;}
\}
\]

\[
\text{r0 := (if r1 < r2 then r1} \\
\text{elsif r1 > r2 then r2} \\
\text{elsif r1 = r2 then 0} \\
\text{else Integer’Invalid);}
\]

Computational complexity: \(O(n)\)
Control Structures

if “cmp r1, r2”, eq, “mov r3, #1”, “mov r3, #0”

for r1, 1, 100 “add r3, r1”

mov r1, #1
for:
  cmp r1, #100
  beq then
  mov r3, #0
  b end_if
then:
  mov r3, #1
end_if:

while “cmp r1, #100”, lt, “mul r1, r1”

b while_condition
while:
  mul r1, r1
while_condition:
  cmp r1, #100
  blt while
Control Structures

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

```
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:
```

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

    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