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?

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

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

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

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
Conditionals – IF-ELSE

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

```
cmpeq 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.

How do either of those look in assembly?

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

Conditionals – IF-ELSE

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

```
.Conditionals -- IF-ELSE

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

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
```
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

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;
done

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

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

Loops – FOR

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

\[
\begin{align*}
  & \text{mov } r1, \#1 \quad \text{set index to start value} \\
  & \text{cmp } r1, \#100 \quad \text{check whether it went beyond its end value} \\
  & \text{bgt \ end_for} \quad \text{if so, stop the loop} \\
  & \text{add } r3, r1 \quad \text{do the work} \\
  & \text{add } r1, \#1 \quad \text{increment the index} \\
  & \text{b } \text{for } \text{end_for;}
\end{align*}
\]

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

Can we form a general pattern for this?
Control Structures

Loops – FOR

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

```
macro for register, from, to, body
mov 
for@:
cmp 
do
add 
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 – WHILE

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

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

What are the components?

Computational complexity: \( \Theta(n^2) \)

Can we form a general pattern for this?
Loops – WHILE

Control Structures

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

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

while register1 < 100 do
    register1 = register1 ** 2;
enddo

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

We can now write:

while \texttt{cmp r1, #100}, \texttt{lt}, \texttt{mul r1, r1}\

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

Computational complexity: Undefined
Conditionals – CASE (indexed)

The complexity of this operation is $O(1)$, e.g., it is independent of the number of cases!

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
   \.ends

.. macro indexed_case_id id index case_body other_cases: vararg
   \tbh [pc, \index, \lsl \#1]
   \branch_table_id:
   \table_entry \& \id, \"case_body\", \other_cases \; build up the table entries
   \case_entry \& \id, \"case_body\", \other_cases \; add the codes with a label each
   \indexed_case_end_id:
   \.ends

\tbh [PC, r1, \lsl \#1] \; PC used as base of branch table, r1 is index
\branch_table:
   \.word (case_red - branch_table)/2 \; case_red 16 bit offset
   \.word (case_green - branch_table)/2 \; case_green 16 bit offset
   \.word (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 parts which are actually producing code are highlighted.

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
   \.ends

.. macro indexed_case_id id index case_body other_cases: vararg
   \tbh [pc, \index, \lsl \#1]
   \branch_table_id:
   \table_entry \& \id, \"case_body\", \other_cases \; build up the table entries
   \case_entry \& \id, \"case_body\", \other_cases \; add the codes with a label each
   \indexed_case_end_id:
   \.ends

\tbh [PC, r1, \lsl \#1] \; PC used as base of branch table, r1 is index
\branch_table:
   \.word (case_red - branch_table)/2 \; case_red 16 bit offset
   \.word (case_green - branch_table)/2 \; case_green 16 bit offset
   \.word (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 parts which are actually producing code are highlighted.
### Conditionals – CASE (guarded expressions, list of conditions)

```plaintext
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?"
```

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

`: Computational complexity: Θ(1)`

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

... similar structure? ... many syntax versions?
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);
```

Keep in mind:
Macro programming is pure textual replacement.
The result is a text which is then translated by the assembler into machine code.

---

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

---

```
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

---

```
--
.macro guards_rec id case_nr expression condition case_body other_cases:vararg
  case_id id, "expression", \condition, "case_body", other_cases
.ends

.macro case_id id expression condition case_body other_cases:vararg
  guards_rec id, i, "expression", \condition, "case_body", other_cases
.ends
```

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.
Control Structures

Conditionals – CASE (guarded expressions, list of conditions)

\[ r0 :: Int \rightarrow Int \rightarrow Int \rightarrow Int \]
\[ r0 \ r1 \ r2 \ r3 \]
\[ | r1 < r2 \Rightarrow r1 \]
\[ | r1 > r2 \Rightarrow r2 \]
\[ | r1 = r2 \Rightarrow 0 \]
\[ | \text{otherwise} = \text{error "How did I get here?"} \]

```haskell
r0 :: Int -> Int -> Int -> Int
| 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;
}
```

Computational complexity: \(O(n)\)

```
cmp r1, r2
blt case_a
cmp r1, r2
bgt case_b
```

```
case_a:
  mov r0, r1
  b end_case
```

```
for (r1, 1, 100) "add r3, r1"
mov r1, #1
for:
  cmp r1, #100
  beq then
  mov r3, #0
  while:
  mul r1, r1
  b while_condition
  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
  bgt end for
  add r3, r1
  add r1, #1
  b for
  end if:
```

Control Structures
Control Structures

Summary

Control Structures

- Assembler Macros
  - Local labels
  - Recursive macros

- Control Structures in machine code
  - IF
  - WHILE
  - FOR
  - CASEs

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

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
  b end_case

toh [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
  b 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

beq case_a

beq case_b

beq case_c

b end_case

bend_case:
```

You can form all common sequential control structures
(or generate them via macros if you wish)

(including function calls)