CSE 331: Microprocessor Interfacing and Embedded Systems

Assembly Language

Branching

https://cpulator.01xz.net/?sys=arm-de1soc

Control Flow

🔁 **Control Flow in Assembly Programs**

- **By default**, instructions in an assembly program execute **sequentially**, one after another.
- The **Program Counter (PC)** is automatically incremented by the processor to point to the **next instruction**.
- However, **modifying the PC during execution** allows the program to **change its normal flow**, a concept known as **control flow change**.

Control Flow

🛠️ **Ways to Change the Flow of Control**

- **Branch Instructions** – Jump to a different part of the program.
- **Conditional Execution** – Execute instructions only if certain conditions are met.
- **Subroutine Calls** – Jump to reusable blocks of code (functions) and return.
- **Interrupts** – Asynchronous control changes triggered by hardware or software events.

Data Comparison

`CMP Rn, Op2`	Compare	Set NZCV flags on $Rn - Op2$
`CMN Rn, Op2`	Compare negative	Set NZCV flags on $Rn + Op2$

- The CMP instruction subtracts the value of Op2 from the value in Rn.
- It is the same as a SUBS instruction, except that the processor discards the result.
- CMP updates the N, Z, C, and V flags per the subtraction result.

Data Comparison

`CMP Rn, Op2`	Compare	Set NZCV flags on *Rn* – *Op2*
`CMN Rn, Op2`	Compare negative	Set NZCV flags on *Rn* + *Op2*

- The CMN instruction adds the value of Op2 to the value in Rn.
- "CMN Rn, Op2" is like "ADDS Rn, Op2 " except that the result is discarded.
- CMN updates N, Z, C, and V.

Condition Testing

Suffix	Description	Flags tested
EQ	**EQ**ual	Z = 1
NE	**N**ot **E**qual	Z = 0
CS/HS	unsigned **H**igher or **S**ame	C = 1
CC/LO	unsigned **LO**wer	C = 0
MI	**MI**nus (negative)	N = 1
PL	**PL**us (positive or zero)	N = 0
VS	o**V**erflow **S**et	V = 1
VC	o**V**erflow **C**lear	V = 0
HI	unsigned **HI**gher	C = 1 & Z = 0
LS	unsigned **L**ower or **S**ame	C = 0 or Z = 1
GE	signed **G**reater or **E**qual	N = V
LT	signed **L**ess **T**han	N != V
GT	signed **G**reater **T**han	Z = 0 & N = V
LE	signed **L**ess than or **E**qual	Z = 1 or N ! = V
AL	**AL**ways	

Condition Testing

Notes:

- These suffixes can be added to most ARM instructions:
  - `MOVNE r0, #0` → Move 0 into r0 **if not equal**
  - `ADDEQ r1, r2, r3` → Add r2 + r3 into r1 **if equal**

```
CMP r1, #0 ;            CMP updates N, Z, C and V flags
RSBLT r1, r1, #0;Run r1 = 0 - r1 if r1 < e. LT= signed Less Than.
```

Branch Instructions in Assembly

- **Branch instructions** are used to **change the flow** of program execution from the normal sequential order.
- They allow the processor to **jump to a different part** of the code.
- Two main types:
  - **Unconditional branch**
  - **Conditional branch**

Unconditional vs Conditional Branch

- Unconditional Branch
  - Always changes the flow.
  - Loads the **target instruction address** into the **program counter (PC)**.
  - Execution continues from the new location.
  - Example: *`B target_label`*

Unconditional vs Conditional Branch

- Conditional Branch
  - Checks a **specific condition** before branching.
  - If the condition is **true**, control jumps to the target label.
  - Otherwise, execution continues sequentially.
  - Equivalent to:
    "If condition is true, go to label."
  - Example: *`BEQ  Label`* (branch if equal)

Using Condition Suffixes

- Branch instruction `B` can be combined with **condition suffixes** to create conditional branches:
- `BEQ` – Branch if equal
- `BNE` – Branch if not equal
- `BGT`, `BLT`, `BGE`, etc.
- These suffixes depend on the **status flags** set by previous instructions (like `CMP`).

Using Condition Suffixes

- **BEQ (Equal)**
  - *CMP r0, r1*
  - *BEQ equal_label*   ; Branch if r0 == r1

- **BNE (Not Equal)**
  - *CMP r0, r1*
  - *BNE not_equal_label*   ; Branch if r0 != r1

Using Condition Suffixes

- **BCS / BHS (Carry Set / Higher or Same - Unsigned)**
  - `CMP r0, r1`
  - `BCS higher_or_same_label   ; Branch if r0 >= r1 (unsigned)`

- **BCC / BLO (Carry Clear / Lower - Unsigned)**
  - `CMP r0, r1`

  `BCC lower_label   ; Branch if r0 < r1 (unsigned)`

Using Condition Suffixes

- **MI (Minus - Negative)**
  - `CMP r0, #0`
  - `BMI negative_label    ; Branch if r0 < 0`

- **BPL (Plus - Positive or Zero)**
  - `CMP r0, #0`
    `BPL positive_label    ; Branch if r0 >= 0`

Using Condition Suffixes

- **BHI (Higher - Unsigned)**
  - `CMP r0, r1`
  - `BHI unsigned_higher_label  ; Branch if r0 > r1 (unsigned)`

- **BLS (Lower or Same - Unsigned)**
  - `CMP r0, r1`

  `BLS lower_or_same_label  ; Branch if r0 <= r1 (unsigned)`

Using Condition Suffixes

- **BGE (Greater or Equal - Signed)**
  - `CMP r0, r1`
  - `BGE signed_greater_equal_label ; Branch if r0 >= r1 (signed)`

- **BLT (Less Than - Signed)**
  - `CMP r0, r1`

    `BLT signed_less_label ; Branch if r0 < r1 (signed)`

Using Condition Suffixes

- **BGT (Greater Than - Signed)**
  - `CMP r0, r1`
  - `BGT signed_greater_label ; Branch if r0 > r1 (signed)`

- **BLE (Less or Equal - Signed)**
  - `CMP r0, r1`

  `BLE signed_less_equal_label ; Branch if r0 <= r1 (signed)`

Examples

Simple comparison (unsigned) example

C	Assembly

`uint32_t x, y, z;`
`x = 0x00000001;`
`y = 0xFFFFFFFF;`
 	
`if (x > y)`
`    z = 1;`
`else`
`    z = 0;`
	`    MOV r5, #0x00000001 , r5 = x`
	`    MOV r6, #0xFFFFFFFF , r6 = y`
	`    CMP r5, r6`
	`    BLS else; branch if <=`
	`Then:`
	`    MOV r7, #1; z = 1`
	`    B Endif; skip next instruction`
	`else:`
	`    MOV r7, #0; z = 0`
	`Endif:`
	`...`

Examples

Simple comparison (signed) example

C

```c
uint32_t x, y, z;
x = 1;
y = -1;

if (x > y)
    z = 1;
else
    z = 0;
```

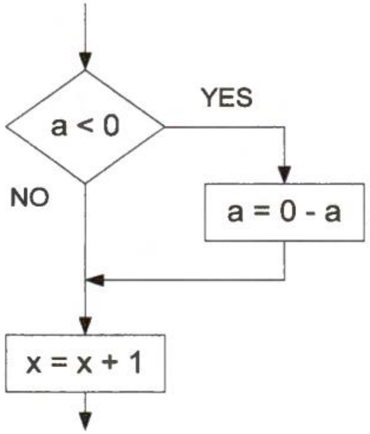
Assembly

```
    MOV r5, #1, r5 = x
    MOV r6, #-1, r6 = y
    CMP r5, r6
    BLE else; branch if signed <=
Then:
    MOV r7, #1; z = 1
    B Endif; skip next instruction
else:
    MOV r7, #0; z = 0
Endif:
...
```

If-then Statement

C Program

```c
if (a < 0 ) {
  a = 0 - a;
}
x = x + 1;
```




```
        ; r1 = a, r2 = x
        CMP r1, #0          ; Compare a with 0
        BGE endif           ; Go to endif if a ≥ 0
then    RSB r1, r1, #0      ; a = - a
endif   ADD r2, r2, #1      ; x = x + 1
```

```
        ; r1 = a, r2 = x
        CMP   r1, #0        ; Compare a with 0
        RSBLT r1, r1, #0    ; a = 0 - a if a < 0
        ADD   r2, r2, #1    ; x = x + 1
```

Compound Boolean expression

C Program	Assembly Program

// x is a signed integer
if(x <= 20 || x >= 25){
    a = 1;
}
	; r0 = x, r1 = a
	        CMP r0, #20 ; compare x and 20
	        BLE then ; go to then if x <= 20
	        CMP r0, #25 ; compare x and 25
	        BLT endif ; go to endif if x < 25
	then MOV r1, #1 ; a = 1
	endif

Compound Boolean expression

<table>
<tr><th>C Program</th><th>Assembly Program</th></tr>
<tr><td>

```c
if(x > 20 && x < 25){
   a = 1;
}
```

</td><td>

```
        ; Assume r0 = x, r1 = a
        CMP  r0, #20   ; compare x with 20
        BLE  endif     ; go to endif if x <= 20
        CMP  r0, #25   ; compare x with 25
        BGE  endif     ; go to endif if x >= 25
        MOVS r1, #1    ; a = 1
endif
```

</td></tr>
</table>

Compound Boolean expression

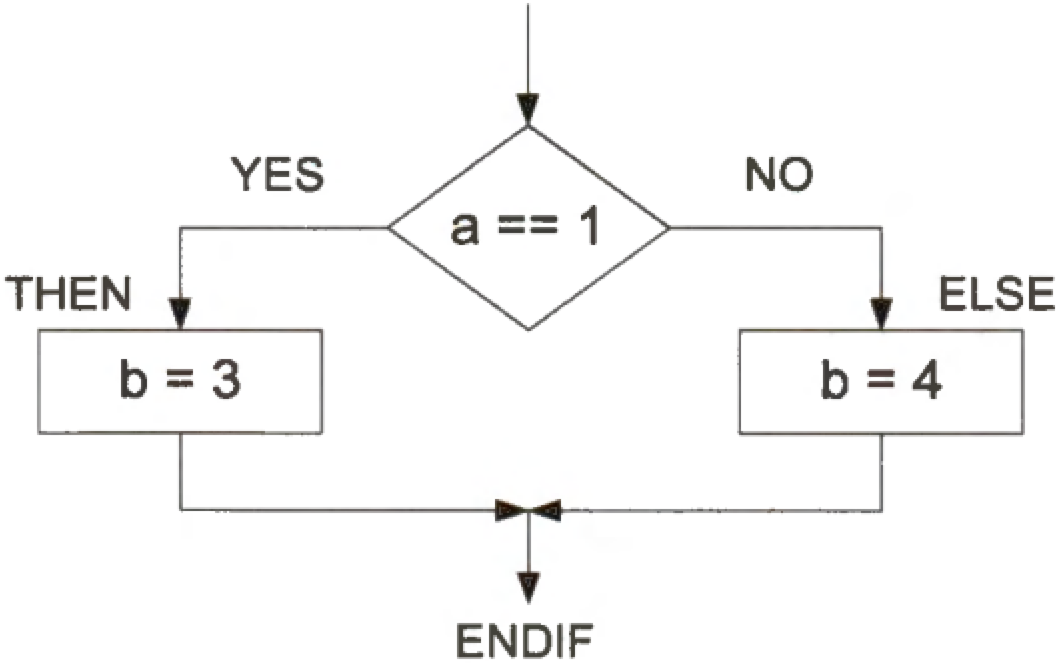
```
if ( x == 5 || (x > 20 && x < 25) )
    a = 1;
```

```
        ; Assume r0 = x, r1 = a
        CMP  r0, #5     ; compare x with 5
        BEQ  then       ; if x == 5, go to then

        CMP  r0, #20    ; compare x with 20
        BLE  endif      ; go to endif if x ≤ 20

        CMP  r0, #25    ; compare x with 25
        BGE  endif      ; go to endif if x ≥ 25

then    MOVS r1, #1     ; a = 1
endif
```

If-then-else Statement

C Program

```c
if (a == 1)
    b = 3;
else
    b = 4;
```



If-then-else Statement

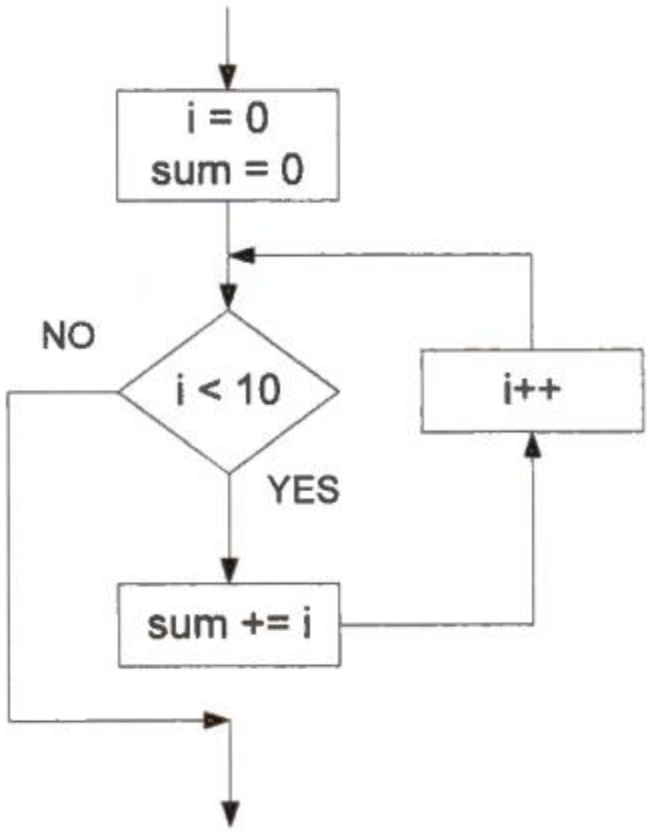
Assembly Program 1	Assembly Program 2
`        ; r1 = a, r2 = b`	`; r1 = a, r2 = b`
`        CMP r1, #1   ; compare a and 1`	`CMP   r1, #1  ; compare a and 1`
`        BNE else     ; go to else if a ≠ 1`	`MOVEQ r2, #3  ; b = 3 if a = 1`
`then    MOV r2, #3   ; b = 3`	`MOVNE r2, #4  ; b = 4 if a ≠ 1`
`        B   endif    ; go to endif`	
`else    MOV r2, #4   ; b = 4`	
`endif`	

For Loop

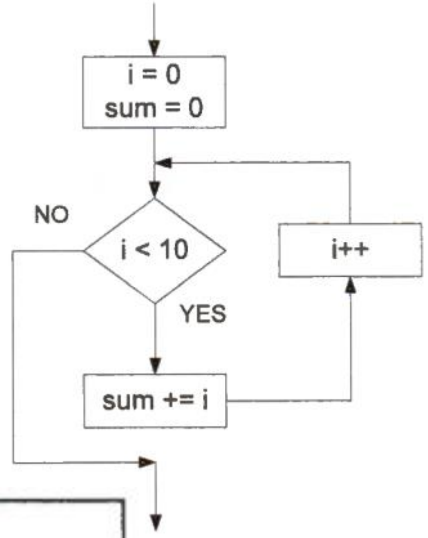
- How does a For loop work?
- How does the program flows?

C Program

```c
int i;
int sum = 0;

for(i = 0; i < 10; i++){
  sum += i;
}
```



For Loop



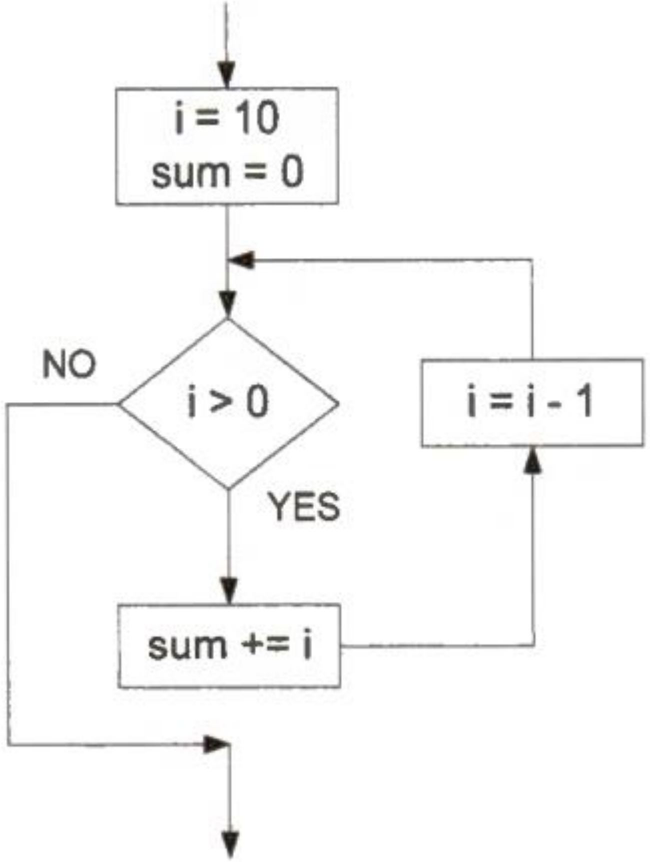
Assembly Program 1	Assembly Program 2	Assembly Program 3

`      MOV r0, #0   ; i`
`      MOV r1, #0   ; sum`
 	
`      B   check`
`loop  ADD r1, r1, r0`
`      ADD r0, r0, #1`
`check CMP r0, #10`
`      **BLT** loop`
`endloop`
	`      MOV r0, #0   ; i`
	`      MOV r1, #0   ; sum`
	
	`loop  CMP r0, #10`
	`      **BGE** endloop`
	`      ADD r1, r1, r0`
	`      ADD r0, r0, #1`
	`      B   loop`
	`endloop`
		`      MOV r0, #0   ; i`
		`      MOV r1, #0   ; sum`
		
		`loop  CMP   r0, #10`
		`      **ADDLT** r1, r1, r0`
		`      **ADDLT** r0, r0, #1`
		`      **BLT**   loop`
		`endloop`

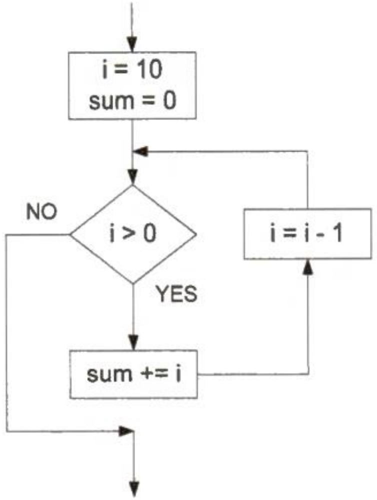
While Loop

C Program

```c
int i = 10;
int sum = 0;

while( i > 0 ){
  sum += i;
  i--;
}
```



While Loop



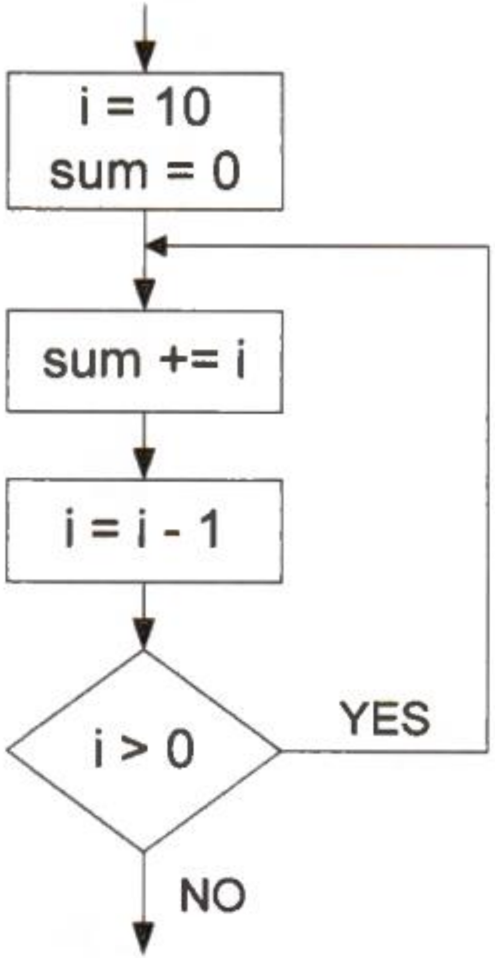
Assembly Program 1	Assembly Program 2	Assembly Program 3

<pre>   MOV   r0, #10 ; i
      MOV   r1, #0 ; sum
 	
      B   check
loop ADD   r1, r1, r0
      SUB   r0, r0, #1
check CMP   r0, #0
      BGT   loop
endloop</pre>
	<pre> MOV r0, #10 ; i
	      MOV r1, #0 ; sum
	
	loop CMP r0, #0
	      BLE endloop
	      ADD r1, r1, r0
	      SUB r0, r0, #1
	      B loop
	endloop</pre>
		<pre> MOV r0, #10 ; i
		      MOV r1, #0 ; sum
		
		loop CMP r0, #0
		      ADDGT r1, r1, r0
		      SUBGT r0, r0, #1
		      BGT loop
		endloop</pre>

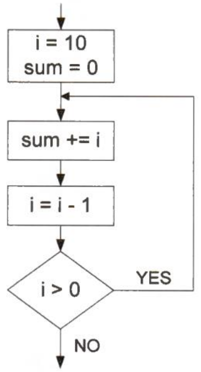
Do-While Loop

C Program

```c
int sum = 0;
int i = 10;
do{
    sum += i;
    i--;
} while( i > 0 );
```



Do-While Loop



Assembly Program 1	Assembly Program 2
`MOV r0, #10 ; i = 10`	`MOV r0, #10 ; i = 0`
`MOV r1, #0 ; sum = 0`	`MOV r1, #0 ; sum = 0`
`loop ADD r1, r1, r0 ; sum += i`	`loop ADD r1, r1, r0 ; sum += i`
`SUB r0, r0, #1 ; i--`	**`SUBS`** `r0, r0, #1 ; i--`
`CMP r0, #0`	**`BGT`** `loop`
`BGT` `loop`	`endloop`
`endloop`	

Continue

C Program	Assembly Program

<pre>int i;
int sum = 0;
 	
for(i = 0; i < 10; i++) {
    if (i == 5) // skip 5
        continue;
    sum += i;
}</pre>
	<pre> MOVS r0, #0 ; i = 0
	        MOVS r1, #0 ; sum = 0
	
	loop CMP r0, #10
	        BGE endloop
	        CMP r0, #5
	        ADDNE r1, r1, r0 ; sum += i
	        ADD r0, r0, #1 ; i++
	        B loop
	endloop</pre>

Break

Implement the break statement in Assembly

C Program

```
int i;
int sum = 0;

for(i = 0; i < 10; i++) {
    if (i == 5) // skip 5
          Break;
    sum += i;
}
```

Excercise

1. Translate the following code into a C program and explain what it does.

```
        MOV r2, #1
        MOV r1, #1

loop    CMP r1, r0
        BGT done
        MUL r2, r1, r2
        ADD r1, r1, #1
        B   loop

done    MOV r0, r2
```

Excercise

$$sum = \sum_{i=0}^{9} a_i^3$$

4. Define an array with 10 unsigned integers a_i $(0 \le i \le 9)$ in assembly code, and write an assembly program that calculates the sum of the cube of these 10 unsigned integers.

```
.data
array: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 @Define 10 unsigned integers
length: .word 10 @Store length if needed
.global main

main:
    MOV     R0, #0          @ index i = 0
    LDR     R1, =array      @ load base address of array into R1
    MOV     R4, #0          @ sum = 0

main:
   MOV    R0, #0        @ Initialize loop index i = 0
   LDR    R1, =array     @ Load base address of array
   MOV    R4, #0        @ Initialize sum = 0
```

```
loop:
   LSL    R5, R0, #2       @ R5 = i * 4 (byte offset for a[i])
   ADD    R5, R1, R5       @ R5 = &array[i] (address of a[i])
   LDR    R2, [R5]         @ Load a[i] into R2

   MUL    R3, R2, R2       @ R3 = a[i] * a[i]
   MUL    R3, R3, R2       @ R3 = a[i]^3
   ADD    R4, R4, R3       @ sum += a[i]^3

   ADD    R0, R0, #1       @ i++
   CMP    R0, #10          @ Compare i with 10
   BLT    loop             @ If i < 10, continue loop
```

Excercise

8. Test for complex roots in solution to the following quadratic equation:

$$ax^2 + bx + c = 0$$

The solution has complex roots if $b^2 - 4ac$ is smaller than `0` and real roots otherwise. Suppose a, b, and c are signed integers and they are stored in register `r0`, `r1`, and `r2`. Write an assembly program that set register `r3` to `1` if the solution has complex roots and `0` otherwise.

Excercise

11. Translate the following C program into an assembly program. The C program finds the minimal value of three signed integers. Assume *a*, *b*, and *c* is stored in register `r0`, `r1`, and `r3`, respectively. The result *min* is saved in register `r4`.

```
if (a ≤ b && a < c) {
    min = a;
} else if (b < a && b < c) {
    min = b;
} else {
    min = c;
}
```

Excercise

14. Write an assembly program that calculates the factorial of a non-negative integer n. Assume n is given in register `r0`, and the result is saved in register `r1`.

$$f(n) = \prod_{i=1}^{n} i = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$