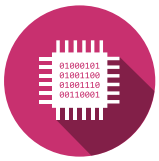


Programcounter

Labor Digital Design

Contents

1 Goals

This lab presents the development of a program code using a Read-Only Memory (ROM) and exercises the creation of a program counter Program Counter (PC).

It also exercises drawing hierarchical circuits.

2 Control ROM of the ALU

2.1 Circuit

The Figure 1 shows a simplified view of a processor, with an Arithmetic and Logical Unit (ALU), registers, and a program counter.

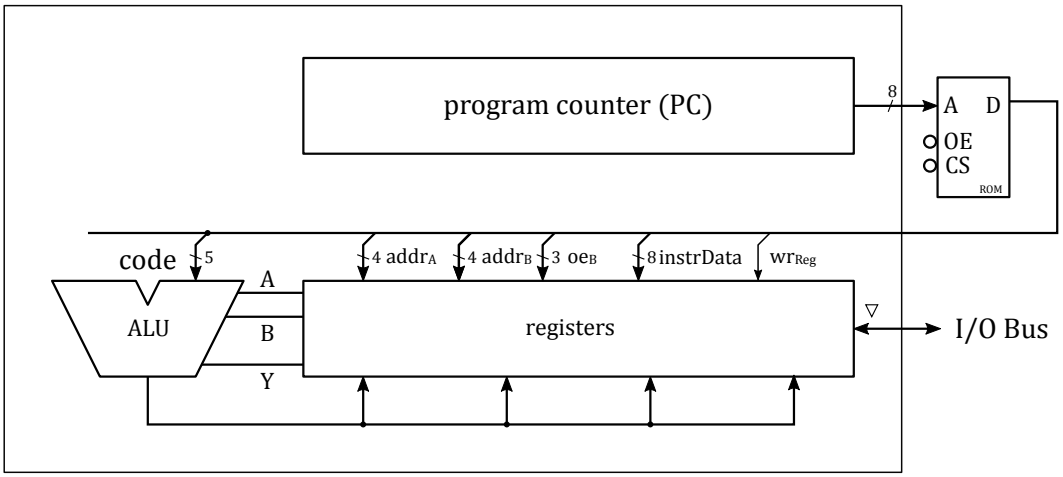


Figure 1 - Control ROM of the ALU and registers

The address of the ROM is generated by a program counter, which allows reading the stored code sequentially. The data of the ROM contains Machine Instructions (MIs) which consist of control signals for the ALU and registers.

The input Output Enable (OE) controls the high-impedance output of the ROM. The input Chip Select (CS) is the select signal for the ROM. Both must be active for the component to provide its data at the output.

2.2 Sequencing of Operations

Instructions are executed in 2 phases, as shown in Figure 2.

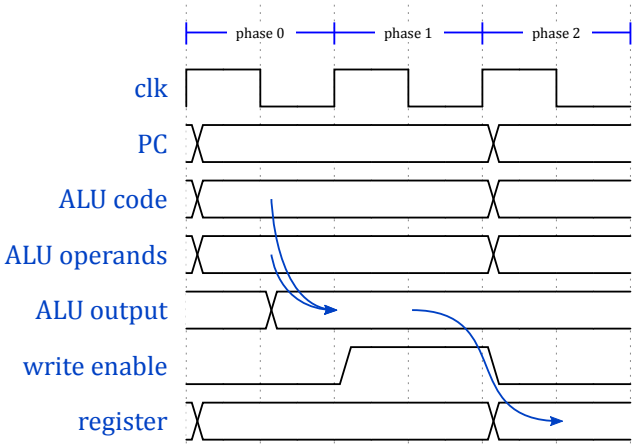
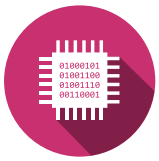


Figure 2 - Phases of instructions

The output of the ALU is stable at the end of phase 0. The new value of the selected register is latched on the rising edge of the clock at the end of phase 1.



3 Software implementation of a serial port

The Figure 3 shows the timing of the serial transmission of a data word.

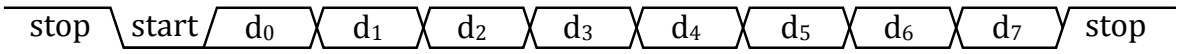


Figure 3 - Serial transmission

The serial data is transmitted on the Least Significant Bit (LSB) of the processor data bus.

3.1 Linear algorithm

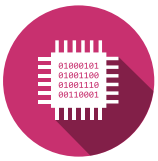
The Assembler program code stored in the ROM is shown in Listing 1.

```
LOAD        s3, FF              ; load stop bit
OUTPUT      s3                  ; output stop bit
LOAD        s3, s3              ; no operation
LOAD        s3, s3              ; no operation
LOAD        s3, s3              ; no operation
LOAD        s3, s3              ; no operation
LOAD        s0, 00              ; load start bit
OUTPUT      s0                  ; output start bit
INPUT       s1                  ; load word to send
OUTPUT      s1                  ; output word, LSB is considered
SR0         s1                  ; shift word, bit 1 -> LSB
OUTPUT      s1                  ; output bit 1
SR0         s1                  ; bit 2 -> LSB
OUTPUT      s1                  ; output bit 2
SR0         s1                  ; bit 3 -> LSB
OUTPUT      s1                  ; output bit 3
SR0         s1                  ; bit 4 -> LSB
OUTPUT      s1                  ; output bit 4
SR0         s1                  ; bit 5 -> LSB
OUTPUT      s1                  ; output bit 5
SR0         s1                  ; bit 6 -> LSB
OUTPUT      s1                  ; output bit 6
SR0         s1                  ; bit 7 -> LSB
OUTPUT      s1                  ; output bit 7
LOAD        s3, s3              ; no operation
OUTPUT      s3                  ; output stop bit
```

Listing 1 - Linear algorithm

In the linear algorithm one instruction after the other is read. There are no loops or jump within the program. Our first program counter only needs to be able to increment the address by one.

Before starting the implementation, ensure that you have read Section 4 on how to create a block a component as well as a **`For`** loop.



3.1.1 Implementation

Start by creating a new component of a 1-bit counter **`CNT/cnt_1bit`**. This counter should increment on the rising edge of the clock when **`incPC = '1'`**. Note that in our system, **`incPC`** is activated every second clock period. Ignore the signals **`loadInstrAddress`** and **`instrAddress`** to load a new value into the counter.

Finally, draw an **`n`**-bit counter using the **`FOR`** frame to generate the **`n`** bits of the counter.

- Create a new component of a 1-bit counter **`CNT/cnt_1bit`**.
- In the component **`CNT/programCounter`** use the **`FOR`** frame and the **`CNT/cnt_1bit`** to generate an **`n`**-bit counter.

3.1.2 Simulation

Simulate the system and verify the correct operation of the counter and processor.

Verify the correct operation of the counter using the testbench **`CNT_test/nanoProcess_tb`** with the simulation file **`$SIMULATION_DIR/CNT1.do`**.

3.2 Algorithm with loops

The following algorithm allows a more compact writing of the program but it uses loops or respectively jumps in the program:

```
1  LOAD    s3, FF              ; load stop bit
2  OUTPUT  s3                  ; output stop bit
3  LOAD    s2, 04              ; initialize loop counter 3
4  SUB     s2, 01              ; decrement loop counter 4
5  JUMP    NZ  03              ; loop back if not end of count 5
6  LOAD    s0, 00              ; load start bit 6
7  OUTPUT  s0                  ; output start bit 7
8  LOAD    s2, 08              ; initialize loop counter 8
9  INPUT   s1                  ; load word to send 9
10 LOAD    s3, s3              ; no operation
11 OUTPUT  s1                  ; output word, LSB is considered
12 SR0     s1                  ; next bit -> LSB
13 SUB     s2, 01              ; decrement loop counter
14 JUMP    NZ  0A              ; loop back if not end of count
15 OUTPUT  s3                  ; output bit 1
```

Listing 2 - Algorithm with loops

3.2.1 Implementation

To implement the algorithm with loops, the PC must allow loading a new value. The two signals **`loadInstrAddress`** and **`instrAddress`** are used to load a new value into the counter.

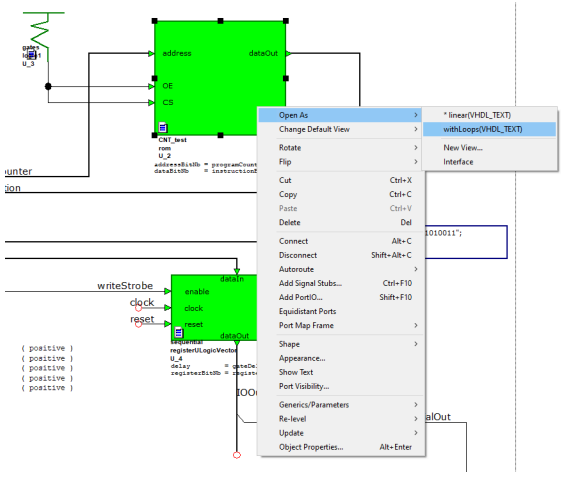
Modify the hierarchical circuit of the program counter of the microprocessor to allow loading a new value into the counter.

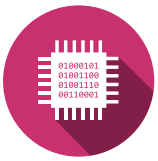
Modify the program counter **`CNT/programCounter`** to allow loading a new value.

3.2.2 Simulation

Simulate the system and verify the correct operation of the counter and processor. Change the default view of the ROM to select the version with code including loops names **`withLoops`**.

1. **`Right-click`** on the component,
2. Select **`Open As`**,
3. Click on **`withLoops`**





Verify the correct operation of the counter using the test bench available **`CNT_test/nanoProcess_tb`** with the simulation file **`$SIMULATION_DIR/CNT2.do`**.

3.3 Comparison

Compare the two algorithms in terms of transmission speed (baudrate) and code size.

- Compare the transmission speed (baudrate) of both algorithms.
- Compare the code size of both algorithms in Listing 1 and Listing 2.

4 Blocks, Components, and `For` Loops

In the linear algorithm, the Program Counter (PC) is an unidirectional counter. It does not allow loading a new value. You need to design this counter to be parameterizable, so that it can be used to create a counter of any bitsize N simply by changing a **`generic`** parameter.

The following sections will guide you through creating the counter using **`FOR Generate`** loop and creating a reusable component.

4.1 Creating a Block

To create a new block, you need to add an element by clicking on the **`Add Block`** button (see Figure 4).

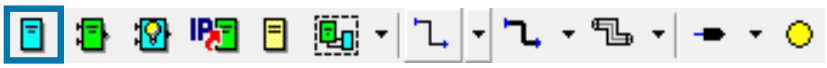


Figure 4 - Add a Block

After clicking the button, a new block can be added by clicking on the schematic. Once the block is added, the necessary Input/Outputs (I/Os) can be connected:

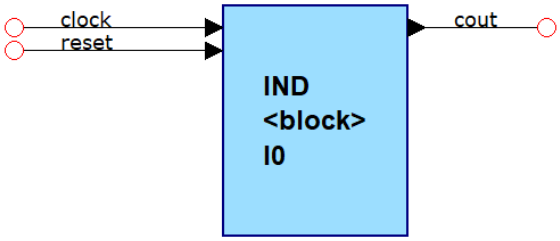
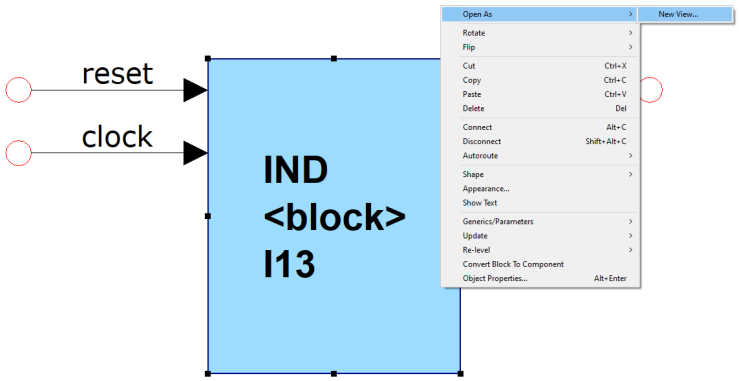


Figure 5 - New block wired

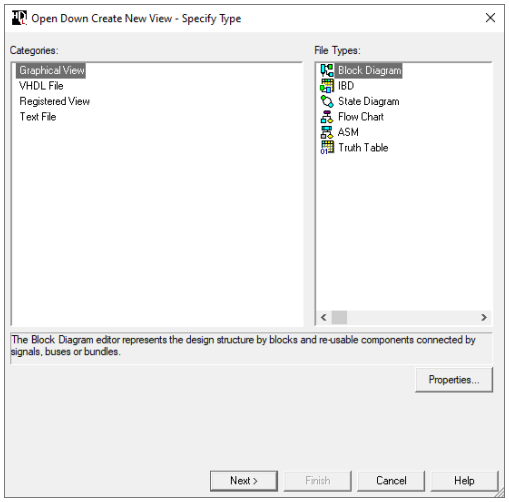
> ⚠ A **`blue block`** cannot be copied and pasted as it exists only once. Only the **`green block`** (components) can be copied and pasted.

Once the I/Os are wired, the type of block content must be selected (**`Block Diagram`**/**`State Diagram`**/**`VHDL file`**).

1. **`Right-click`** on the block
2. Select **`Open As`**
3. Click on **`New View...`**



The following window opens, allowing you to choose the type of block to create. Select the desired block type and click **`Finish`**.



`Link`	`Block type`	`Icon`
`Graphical View/Block Diagram`	block diagram	
`Graphical View/State Diagram`	state machine	
`VHDL File/Architecture`	VHDL code	

4.1.1 Block diagram

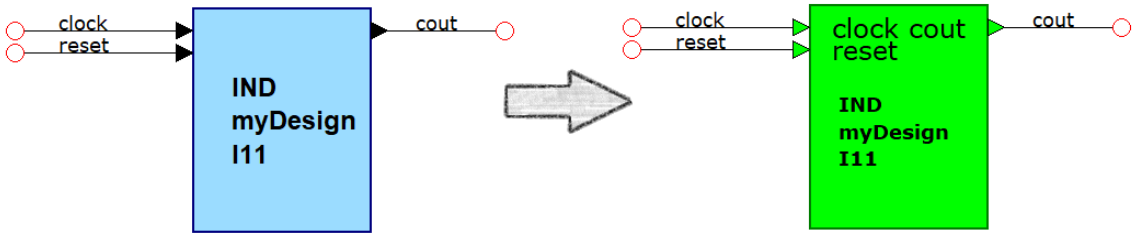
- Select **`Graphical View/Block Diagram`**
- Click on **`Next`**
- Enter the name of the block under **`Design unit`**
- Fill in the I/O table
  - Ensure that the types are correct
  - Define the limits for multi-bit types
- Click on “Finish”

The I/Os can still be added, removed, and modified when editing the schematic.

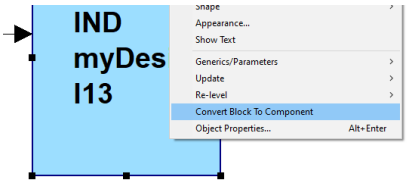
4.2 Convert a Block to a Component (blue to green)

Converting the block to a component (from blue to green) allows copying and pasting the block and makes it available in the project library.

Blue blocks are ideal for quickly creating a block interface, while green components are essential for reusing the elements elsewhere in the project.



1. **`Right-click`** on the blue block
2. Click on **`Convert Block to component`**

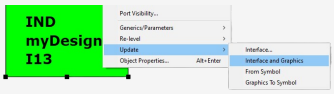


4.3 Update the component interface

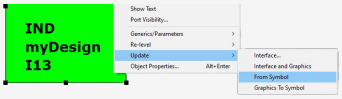
The interface of a component consists of several parts: **`Inputs`** and **`Outputs`**, **`Generics`**, and **`Symbol`**. These different parameters can be added, removed, and modified.

`Once the component interface has been modified`

1. **`Right-click`** on the component
2. Select **`Update`**
3. Click on **`Interface and Graphics`**

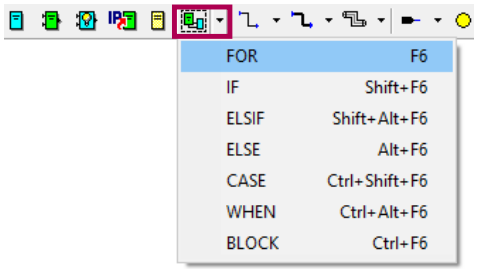


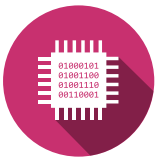
1. **`Right-click`** on the component
2. Select **`Update`**
3. Click on **`From Symbol`**



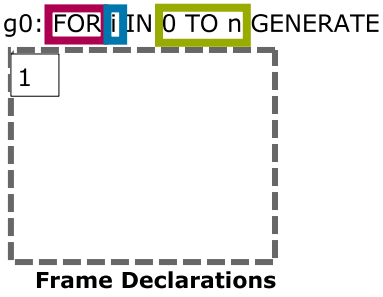
4.4 `For` Generate

The **`FOR Generate`** frame allows you to create multiple iterations of the same structure. To add a new frame, click on the **`add frame`** button and select **`FOR`**.





When you add a **FOR** frame, the following element is drawn on the schematic.



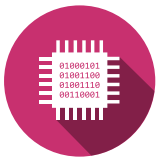
Part	Description
FOR	Type of frame (**FOR** loop)
i	Index of the loop
0 TO n	Range of the **FOR** loop

All signals and components inside the **FOR** frame will be copied `n+1` times. The index `i` goes from `0` to `n`. *Change n to a constant or generic.*

All signals inside the **FOR** frame that are unique across all copies should use index **i** in their Slice/Index.

For example: the **clock** should be the same everywhere; but a counter should output all bits. **Bit[0]**, then **bit[1]**, then **bit[2]** ... **bit[i]**.

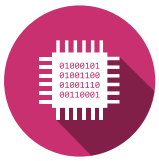
- Only signals and components (green) elements can be placed in a **FOR** loop
- No **In Port** or **Out Port** should reside inside the frame.
- No blocks (blue) can be placed inside.



5 Checkout

This is the end of the lab, you have successfully implemented a program counter that supports all instructions of the μProcessor. Before leaving the lab, make sure you have completed the following tasks:

- [ ] Circuit Design
  - [ ] The 1-bit counter block **`CNT/cnt_1bit`** has been created.
  - [ ] The **`FOR`** frame has been used to duplicate the 1-bit counter block.
  - [ ] The **`n`**-bit counter has been created.
  - [ ] The counter has been adjusted to allow loading a new value.
- [ ] Simulations
  - [ ] Both counters have been successfully tested with the testbench **`CNT_test/nanoProcess_tb`** and simulation files **`$SIMULATION_DIR/CNT1.do`** and **`$SIMULATION_DIR/CNT2.do`**.
- [ ] Comparison
  - [ ] The baud rate of both algorithms has been compared.
  - [ ] The code size of both algorithms has been compared.
- [ ] Documentation and Project Files
  - [ ] Ensure that all steps (design, simulations, comparison) are well documented in your lab report.
  - [ ] Save the project on a USB stick or in the shared network folder (**`\\filer01.hevs.ch`**).
  - [ ] Share the files with your lab partner to ensure continuity of work.



Glossary

***ALU* – Arithmetic and Logical Unit** 2, 2, 2, 2

***CS* – Chip Select** 2

***I/O* – Input/Output** 7, 7, 8, 8

***LSB* – Least Significant Bit** 3

***MI* – Machine Instruction**: A machine instruction is a binary-coded operation that a processor can execute directly. It typically consists of an opcode, operands and immediates. 2

***OE* – Output Enable** 2

***PC* – Program Counter** 1, 5

***ROM* – Read-Only Memory** 1, 2, 2, 2, 2, 2, 3, 5