VRIJE
UNIVERSITEIT
BRUSSEL

1ste Master Elektronica-ICT Industrieële Ingenieurswetenschappen

Computerarchitectuur

MIPS Processor Ontwerp en Implementatie

Jurgen Vandendriessche

14 oktober 2019

Ingenieurswetenschappen

Inhoudsopgave

Introduction

Nowadays, most programmers write their applications in what we call “the Higher Level Programming languages”, such as Java, C#, Delphi, etc. These applications are then compiled into machine code. In order to run this machine code the underlying hardware needs be able to “understand” the proposed code. The aim of this practical course is to give an inside on the principles of a working system. Therefore, this course is more focussed on the design of a system, rather than learning the VHDL syntax. The system handled in this course is the MIPS processor. The proposed methodologies can also be applied to other system designs and implementations. These systems include digital signal processing blocks, communication modules, etc.
This course will start with a simple factorial example algorithm. The necessary building blocks involved for this algorithm, with respect to the MIPS processor, will be analysed and implemented. Once designed, the system will be simulated and ported to a Field Programmable Gate Array. In order to validate the design, one must make sure that both simulation and real implementation run without any problem.
Once the basic building blocks are implemented, one can add some extra functionalities to this processor. Examples include communication modules, a module for reprogramming the processor, etc.

1 From design to VHDL

1.1 The factorial algorithm

The factorial is a well known mathematical tool in the field of probability calculations. Despite its fast growing output, the factorial has a rather simple formula, given by:

$$x \quad = n! = \prod_{i=0}^{i=n-1} (n-i) \tag{1.1}$$

The above formula can be rewritten as:

$$x = n\,(n-1)\,(n-2)\ldots(n-n+1) \tag{1.2}$$

E.g.: if n equals 4, than the result x would be: $x = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. For small values (i.e. with results smaller than the maximum size of an unsigned integer), one can easily compute the factorial with code snippet 1.1.

Algorithm 1.1: C-code of factorial.

```c
int result=value;
/* we assume the starting value of value is always
bigger than 1 */
while (value!=0)
{
    result=result*value;
    value=value-1;
}
```

The above code is written in a “High Level Programming Language” and is human-readable. The High Level Programming Languages enable abstraction of the underlying machine and all the instructions the latter supports. To make sure a given machine can run the proposed algorithm, this algorithm needs to be compiled into machine code. Machine code consists of binary numbers, such as ’0’ (zeros) and ’1’ (ones), and is therefore less human readable. A still readable language for the programmer, but very close to the machine code, is the assembly code. Assembly and machine code are a one-to-one translation, this is, only one assembly code corresponds to one machine code and vice verse.
In (MIPS) assembly code the previous code snippet would result into:

Algorithm 1.2: MIPS assembly code of factorial.

```
addi $s1,$zero,4  ;value = 4;
add  $s2,$zero,$s1 ;result = 0;
addi $s3,$zero,0  ;set the lower boundery of the while loop
mul  $s2,$s2,$s1  ;result=result*value
sub  $s1,$s1,1    ;value=value-1
bneq $s3,$s3,-8   ;if result!=0, goto address 16
                  ;-> multiply operation
```

Several different architectures exist(e.g. x86, x64, MIPS, etc.). Each of those architectures supports its own instruction set. Each of them defines what a processor is able to handle and which operations are not supported. Taking a closer look at code snippet 1.2, one can remark that the processor should be able to accomplish following operations:

- add, subtract and multiply two numbers,
- calculate next address for fetching next instruction from memory,
- branch when conditions are (not) met.

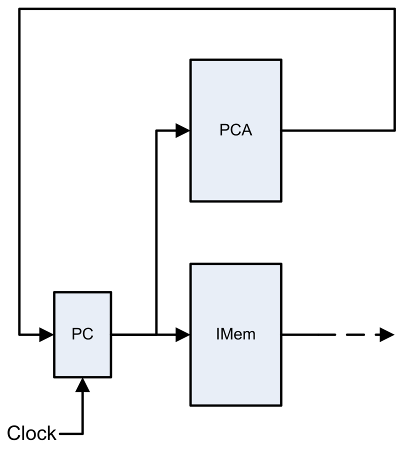
The basic mathematical operations such as addition, subtraction, multiplication and division are commonly used in computer software. They allow the programmer to make basic calculations. The "fetch next" operation is invisible to the programmer, but it assumes that the processor automatically calculates the address of the next instruction it fetches. The branch instructions resembles to the "fetch next" operation, except that the next chosen instruction is not located at the next memory address, but at a given offset from the current address. When trying to run a given algorithm, one should be certain that the processor supports all of the expected features!

1.2 Building modules

As mentioned in previous section, the processor contains at least following features:

- Instruction Memory,
- a module for holding the current program counter and one for calculating the next instruction address. These are respectively the Program Counter (PC) and the Program Counter Adder (PCA),
- Arithmetic Logical Unit (ALU),
- branch (not equal) method (if-then-else statement).

The Instruction Memory, Program Counter and Program Counter Adder are one of the most important components of the MIPS processor. Without these the processor won't be able to run any instructions. Therefore the focus will be set on these three components. The other components can be added as the processor implementation evolves.
Following the instructions needed by the factorial problem, a first draft design would look like picture 1.1.



Figuur 1.1: First design of the processor.

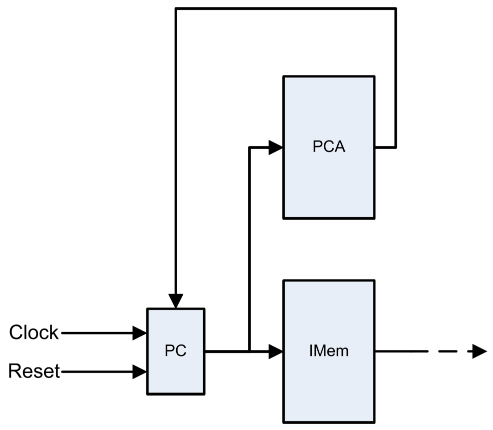
1.2.1 A closer look

The Program Counter, Instruction Memory and Program Counter Adder are the first 3 blocks to be designed. The purpose of each block (technically) is as follows:

- The Program Counter holds the current address of the current instruction. On each clock-cycle it reads the address at the PC_IN port, stores it and forwards it to the PC_OUT port.
- The Program Counter Adder adds a constant of 4 to the current program counter.
- The Instruction Memory contains all the necessary instructions to run the given algorithm. The input of this block is the program counter provided by the Program Counter. The output is the instruction corresponding to the given instruction address (Program Counter).

This very small structure works fine. But if the algorithm stalls, the machine could not be restarted! Unless someone plugs out the power cord, the machine will remain in the same state for ever! Another way to restart the machine can be achieved by adding a reset button. A possible candidate block for resetting the machine is the Program Counter. By resetting the Program Counter, the output of this block is reset to the first address proposed when the machine starts. All other blocks follow this reset (or change), and the reset will propagate through the complete system! Another problem to consider is how to load the instructions in the Instruction Memory. When the machine starts, that

memory (especially if volatile) will be empty, so no single instruction will reside there. One possible solution could be to add a small ROM (Read Only Memory) and to run the very first instructions from there. This first program attempts to read a device with non-volatile memory (e.g. flash, hard drive,...) and stores all these instructions into memory. After the read and store operation the machine is able to execute instructions located in the Instruction Memory. The first program (located into a ROM device) is also called a boot-loader. However, very small programs can reside into the ROM. Doing so makes the boot-loader unnecessary.
After the added modifications, and without the boot-loader mechanism, the final system would look like figure 1.2.



Figuur 1.2: The updated design of the processor.

1.2.2 VHDL

VHSIC Hardware Description Language (VHDL) is, as the name already suggests, a language to describe the behavior of a (digital) hardware system. Once the system is designed and written in VHDL, it can be synthesized to a bit-file. This file represents the system in '0' and '1'. After the bit-file has been generated, it can be downloaded onto an Field Programmable Gate Array (FPGA). The FPGA then "runs" the system as a real system would do. Since one can describe small (e.g. OR-ing two signals,etc.) as well as the most advanced circuits (e.g. processors, etc.)in this language, it is well suited for developing the MIPS processor.
The MIPS processor is subdivided into sub blocks (PC, IMem, etc.). This subdivision makes it easier to handle large circuits. One should try to split big circuits into smaller pieces (cf. object oriented programming - divide and conquer). Although VHDL shares a lot of definitions with an object oriented language (block definition - class definition, block behavior - code inside each class-member), VHDL still remains a hardware description language. This means a designer has to think in terms of Finite State Machines and/or combinatorial logic and not in terms of a sequential language.

1.3 Design in VHDL

1.3.1 Program Counter

One of the first blocks to be designed is the program counter (PC). According to previous sections, the PC has following ports (see figure 1.2):

- PC_IN: the next program counter (input, 32bits),
- PC_OUT: the current program counter (output, 32 bits),
- Clock (input, single line),
- Reset (input, single line).

These ports are visible to the outside world and are defined within the bock's entity. Code snippet 1.3 represents the port declarations.

Algorithm 1.3: VHDL port declaration of the program counter (PC).

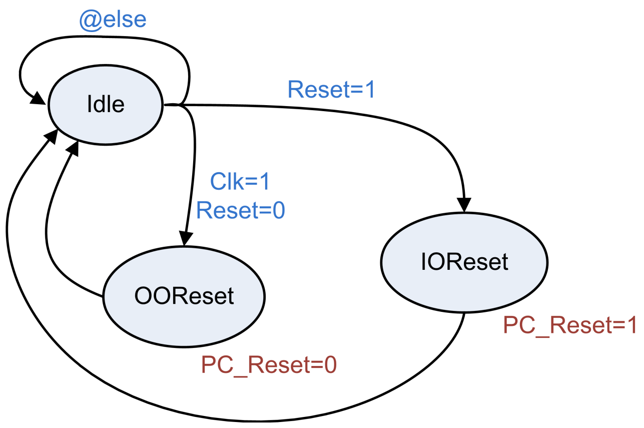
```vhdl
entity PC is
port(
    Clk:    IN  STD_LOGIC;
    Reset:  IN  STD_LOGIC;
    PC_IN:  IN  STD_LOGIC_VECTOR(31 downto 0);
    PC_OUT: OUT STD_LOGIC_VECTOR(31 downto 0)
);
end PC;
```

The first aspect in the definitions is the width of the different buses. The clock (Clk) and the reset (Reset) lines are one bit wide and are defined as "STD_LOGIC". PC_IN and PC_OUT are defined as 32 bit buses and declared as "STD_LOGIC_VECTOR(31 downto 0)". The bus vector definition "31 downto 0" means that the leftmost bit (first wire), with index 31, represents the most significant bit of the bus. Wire zero represents the rightmost bit (least significant bit). In case the bus definition is reversed, the vector definition would look like: "STD_LOGIC_VECTOR(0 to 31)".
The second aspect is the direction in which the data "flows". Clk, Reset and PC_IN are defined is input ports (IN), while PC_OUT is the only output port of the block's entity. One should also notice that the last port declared in the definition always ends without semicolon.
The other part of the block's definition contains the behavior. The complete behavior is enclosed within the "Behavioral" declaration. Before writing any single line of VHDL code, one should list the necessary behavior:

- PC_OUT is updated once every clock cycle. A good trigger is the rising edge of the clock.

- PC_OUT is reset to '0' when the Reset pin has been triggered (Reset = '1'). PC_OUT remains zero as long as Reset = '1'. At next rising edge of the clock following Reset = '0' the PC_OUT will be updated again.

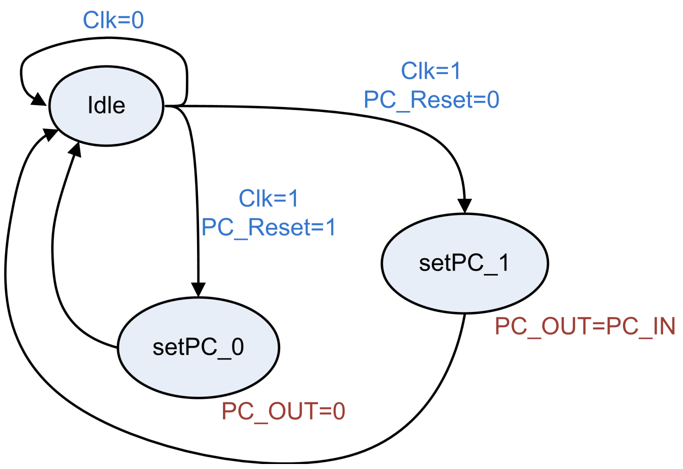
This description leads to following two Finite State Machines (FSM): resetPC and setPC (figures 1.3 and 1.4). Note that the conditions for going into a new state is represented by blue labels. Red labels next to the different states represent the corresponding resulting changes.



Figuur 1.3: resetPC Finite State Machine. "OOReset"and "IOReset" represent "Out Of Reset" and "In Of Reset". "@else" represents the "goto to next state" statement when all other conditions to other states are not met.

Figure 1.3 represents the FSM of the reset option. This FSM handles the asynchronous trigger of Reset (why asynchronous?). When the reset pin has been triggered, an internal reset signal ("PC_Reset") is triggered and remains '1' as long as the external reset pin is '1'. The PC_Reset signal is lowered at next rising edge of the clock following Reset = '0'. The second FSM (figure 1.4) handles the output of the Program Counter. When PC_Reset is triggered, the current program counter remains zero. In the other case the PC_OUT is set to the next program counter (PC_IN). The latter combination ensures that the reset has only effect on the rising edge of the clock.

Since there are two complementary FSM describing the behavior of the Program Counter, one can define two distinct processes. A process in VHDL is a method which allows one to describe how the dataflow is controlled, to describe combinatorial logic, etc. Processes are not always mandatory, but help towards better readability and easier subdivision of tasks. The first process describes the setPC approach, while the other one describes resetPC. Code snippet 1.4 and 1.5 represent the corresponding VHDL code of resetPC and setPC.



Figuur 1.4: setPC Finite State Machine. PC_Reset is the added internal signal from the resetPC state machine.

Algorithm 1.4: Process of resetPC.

```
resetPC:process(Reset,Clk)
begin
    if (Reset='1') then
        PC_RESET<='1';
    elsif ((rising_edge(clk)) and (Reset='0')) then
        PC_RESET<='0';
    end if;
end process resetPC;
```

The resetPC triggers and lowers the PC_RESET. According to the value of PC_RESET, the setPC process sets the output program counter to either a zero or a new instruction address. Therefore the PC_RESET signal should be “visible” to both setPC and resetPC. This can be done by declaring the signal PC_RESET at the complete block model (code snippet 1.6).

Although both processes are a one-to-one translation of the corresponding FSM, some (syntax) points should be discussed here:

- PC_OUT is an output signal (output port). All (outgoing) signals can be assigned using the “<=” operator.
- Values can be assigned to variables (registers used within the block/process) using the “:=” operator.
- Single ’bits’ are marked with with single quotes (’0’ or ’1’). By contrast, a multibit value is marked with double quotes (“0100101..”).

Algorithm 1.5: Process of setPC.

```
setPC:process(Clk)
begin
    if (rising_edge(Clk)) then
        if (PC_RESET='1') then
            PC_OUT <= (others=>'0');
        else
            PC_OUT <= PC_IN;
        end if;
    end if;
end process setPC;
```

Algorithm 1.6: Behavior of PC.

```
architecture Behavioral of PC is
    signal PC_RESET:STD_LOGIC;
begin
    setPC:process(Clk,PC_IN,RESET_TRIGGER)
    ....
    end process resetPC;
    resetPC:process(Reset,Clk)
    ....
    end process setPC;
end Behavioral;
```

- If one wants to assign a zero to a bus or register, the “(others=>’0’)” statement can be used instead of “000000000....”.
- The rising edge checking method on the clock requires special attention. Rising_edge is only true when the involved signal (clk) changes from ’0’ to ’1’. This latter is of great importance if one wants to have a processor with “regular clock”. The counterpart of rising_edge if of course falling_edge. The use of this method should be well considered (i.e. not used on IO connecting buttons/switches,etc.).
- As each process definitions starts with <name>:process(<triggers>), it ends with “end process <name>”.
- Comments can be added by preceding them with “–” (i.e. double-dash).
- Code is not case sensitive: “TESTBLOCK” has the same meaning as “testblock”!

Sometimes it's necessary to add some features imported from libraries. Adding libraries is done by naming the library and to "use" it. This is shown in code snippet 1.7.

Algorithm 1.7: Importing libraries.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

The complete code of PC is shown in code snippet 1.8.

Algorithm 1.8: VHDL code of the Program Counter.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
---------------------------------------------------------------
entity PC is
    port(
        Clk:    IN  STD_LOGIC;
        Reset:  IN  STD_LOGIC;
        PC_IN:  IN  STD_LOGIC_VECTOR(31 downto 0);
        PC_OUT: OUT STD_LOGIC_VECTOR(31 downto 0)
    );
end PC;
---------------------------------------------------------------
architecture Behavioral of PC is
    signal PC_RESET:STD_LOGIC;
begin
    -- the setPC process
    setPC:process(Clk)
    begin
        if (rising_edge(Clk)) then
            if (PC_RESET='1') then
                PC_OUT <= (others=>'0');
            else
                PC_OUT <= PC_IN;
            end if;
        end if;
    end process setPC;
    -----------------------------------------------------------
    -- the resetPC process
    resetPC:process(Reset,Clk)
    begin
        if (Reset='1') then
            PC_RESET<='1';
        elsif ((rising_edge(clk)) and (Reset='0')) then
            PC_RESET<='0';
        end if;
    end process resetPC;
end Behavioral;
```

1.3.2 Instruction Memory

After the Program Counter proposed the new instruction address, the Instruction Memory delivers the corresponding instruction to the following sections of the processor. This means that the Instruction Memory needs to store the necessary instruction into memory. Each instruction has its own address, marked by the 32 bit vector of the program counter. When a given instruction is needed, the Instruction Memory delivers a 32 bit wide instruction. The following considerations have to be taken into account for the Instruction Memory:

- Each instruction is one word long (32 bits).
- Each instruction is addressed by a multiple of 4 bytes (1 word).
- For tiny programs a small memory of 32 words will be sufficient. This means that the first instruction lies at address 0 and the last possible instruction at address 31. One can simply add more memory capacity to augment the number of instructions, but for really large memory an external memory together with a boot-loader should be considered.

The VHDL code for the complete Instruction Memory is given in code snippet 1.9.
Aside from the previously commented syntax features, some words have to be added:

- Each instruction has a length of one word, hence the subtype "word".
- The memory is a full type (i.e. contains several words). The definition of memory should however be clear.
- The variable "myMem" is an instantiation of type memory. This variable contains all the instruction of the Instruction Memory. It is initialized with 8 instructions (here just random values). The given instructions are given in hexadecimal format, hence the "X" preceding each value.
- Each instruction lies at a multiple of 4 bytes. By contrast, the program counter counts in terms of bytes. To avoid systematic "jumps" by four instructions when the new program counter is proposed, the program counter is divided by 4. Division by 4 also means: "take all but the two least significant bits", (PC(31 downto 2)). Since an index is of type "integer", one has to convert the resulting value from logic_vector to integer. To perform this conversion, one can use the "conv_integer" operator.

Algorithm 1.9: VHDL implementation of the Instruction Memory.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--------------------------------------------------------------
entity IMem is
port(
    PC:             IN      STD_LOGIC_VECTOR(31 downto 0);
    Instruction:    OUT     STD_LOGIC_VECTOR(31 downto 0)
);
end IMem;
--------------------------------------------------------------
architecture Behavioral of IMem is
begin
    ----------------------------------------------------------
    MemoryPC:process(PC)
        subtype word  is STD_LOGIC_VECTOR(31 downto 0);
        type   memory is array(0 to 7) of word;
        variable   myMem: memory :=
            (X"00000001", X"00000010",
             X"00000100", X"00001000",
             X"00010000", X"00100000",
             X"01000000", X"10000000");
    begin
        Instruction<=myMem(conv_integer(PC(31 downto 2)));
    end process MemoryPC;
    ----------------------------------------------------------
end Behavioral;
```

1.3.3 Program Counter Adder

The last block in the first design is the PCA block. Although very small, it's a very important block in the design. Without this block, the processor won't be able to fetch the next instruction. The implementation is left as exercise to the reader.

1.4 Bringing it all together - towards the MIPS processor

The first three blocks of the MIPS processor have been developed: the Program Counter, the Instruction Memory and the Program Counter Adder. These blocks play a complementary role in the complete architecture of the MIPS processor. To fulfil there role, they have to be connected to each other in a proper way. VHDL provides a way to interconnect the different blocks to each other. The interconnections are made in a new VHDL module: MIPS_PROCESSOR. First of all, the MIPS processor has to know the ports of the underlying blocks. To do so, the port definitions are declared in the architecture (code snippet 1.10).

Algorithm 1.10: Component description of the blocks within the MIPS processor.

```
architecture Behavioral of MIPS_PROCESSOR is
---------------------------------------------------------------
    component PC
    port(
        Clk:    IN  STD_LOGIC;
        Reset:  IN  STD_LOGIC;
        PC_IN:  IN  STD_LOGIC_VECTOR(31 downto 0);
        PC_OUT: OUT STD_LOGIC_VECTOR(31 downto 0)
    );
    end component;
    -----------------------------------------------------------
    component IMem
    port(
        PC:             IN  STD_LOGIC_VECTOR(31 downto 0);
        Instruction:    OUT STD_LOGIC_VECTOR(31 downto 0)
    );
    end component;
    -----------------------------------------------------------
    component PCA
    port(
        PC_IN:  IN      STD_LOGIC_VECTOR(31 downto 0);
        PC_OUT: OUT     STD_LOGIC_VECTOR(31 downto 0)
    );
    end component;
    -----------------------------------------------------------
    signal program_counter:   STD_LOGIC_VECTOR(31 downto 0);
    signal program_counter4:  STD_LOGIC_VECTOR(31 downto 0);
begin
...
end Behavioral;
```

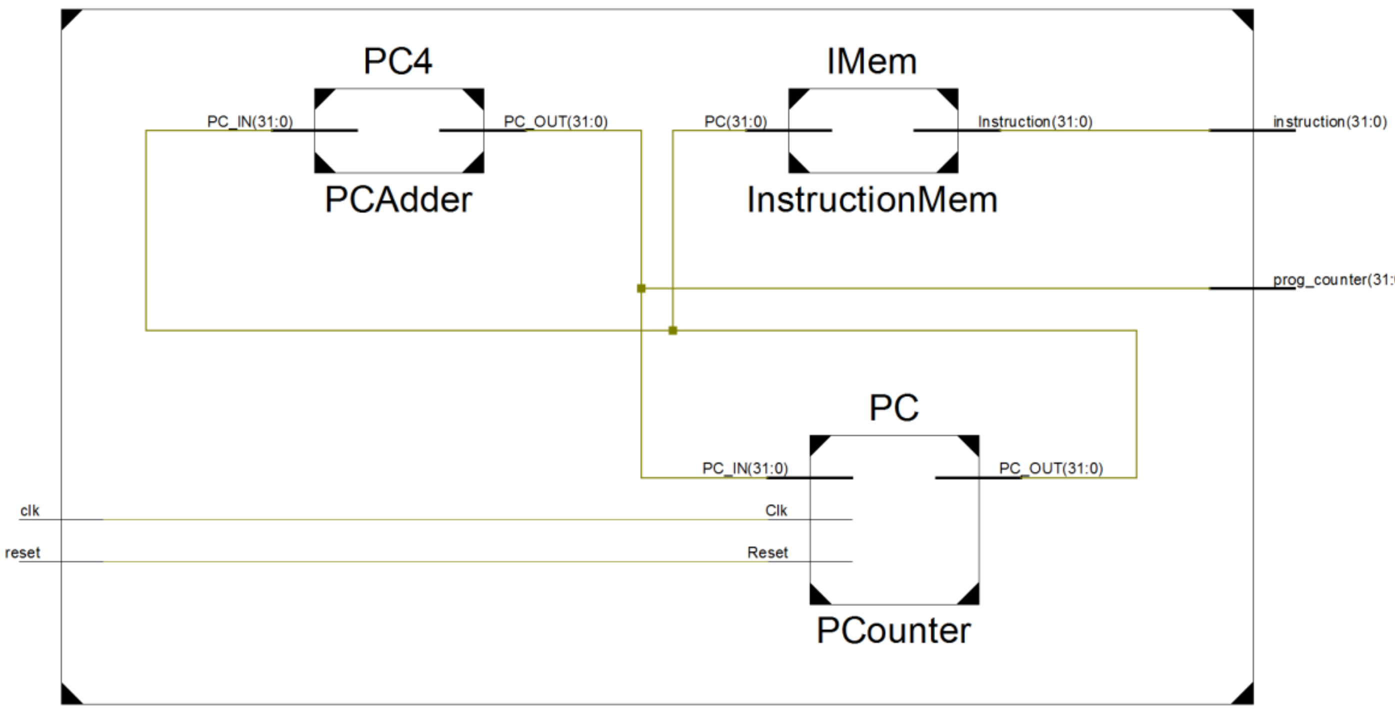
One can remark that the three definitions are almost the same as the port definitions of the individual blocks themselves. The only change that occurs is the keyword "component" instead of "entity". The signals "program_counter" and "program_counter4" are internal buses.
The final interconnections are made in the implementation. Code snippet 1.11 illustrates

all these aspects. For debugging purposes, ports “instruction” and “prog_counter” are defined as output ports in the MIPS processor.

Algorithm 1.11: Interconnection between the modules within the MIPS processor.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
---------------------------------------------------------------
entity MIPS_PROCESSOR is
    port(
        clk:            IN      STD_LOGIC;
        reset:          IN      STD_LOGIC;
        instruction:    OUT STD_LOGIC_VECTOR(31 downto 0);
        prog_counter:   OUT STD_LOGIC_VECTOR(31 downto 0)
    );
end MIPS_PROCESSOR;
---------------------------------------------------------------
architecture Behavioral of MIPS_PROCESSOR is
...
begin
---------------------------------------------------------------
    PCounter:PC port map(
        Clk=>clk,
        Reset=>reset,
        PC_OUT=>program_counter,
        PC_IN=>program_counter4
    );
    -----------------------------------------------------------
    InstructionMem:IMem port map(
        PC=>program_counter,
        Instruction=>instruction
    );
    -----------------------------------------------------------
    PCAdder:PCA port map(
        PC_IN=>program_counter,
        PC_OUT=>program_counter4
    );
    prog_counter<=program_counter4;
---------------------------------------------------------------
end Behavioral;
```

If the processor is correctly designed (i.e. is bug free), the complete processor can be synthesized into a RTL scheme (figure 1.5).



Figuur 1.5: RTL schematic of the MIPS processor.

2 Design validation

As modukes are been designed and implemented, one needs to make sure the obtained system is viable. Therefore, validation plays an important role during the design and development. All the modules that have been created need to be tested on errors. To do so all modules created can be tested through “testbenches”. Testbenches are an efficient way to visualize the behaviour of a module, especially for the relations between the given input(s) and corresponding output(s). The visualization is done by showing the logic values of the input(s) and output(s) in a graph. In this chapter we will review the testing methods and testbenches of the proposed modules of the previous chapter.

2.1 Instruction Memory

The Instruction Memory has a quite simple correlation between the input (Program Counter) and the output (instruction). Since the Instruction Memory has to deliver the correct instruction according to the proposed program counter, one can easily tabulate the expected input-output relation sequence. A possible input-output relation of the Instruction Memory (code snippet 1.9) can be represented by table 2.1.

Program Counter (input)	Instruction (output)
0x00000000	0x00000001
0x00000004	0x00000010
0x00000008	0x00000100
0x0000000C	0x00001000
0x00000010	0x00010000

Tabel 2.1: Expected input-output relation between the program counter and the instruction.

Xilinx ISE offers a way to test modules (see Appendix). Once the new raw testbench file (IMem_tb) for the Instruction Memory has been created, some parts have to be rewritten in order to be fully functional. The raw created testbench is shown in code snippet 2.2. One should remark the similarities with the MIPS_Processor module. In the MIPS_Processor several sub-modules are brought together to form one system. The same principle is applied to testbenches, where the to be tested module is imported as a component. However, since IMem is a unit under test (uut), all input and output signals of IMem are redefined as internal signals.
Another important facet of testbenches is the way timings are done. Since testbenches are run in a simulator, there will be no clock to trigger the system. In order to make sure the system can be tested, a (testbench) build in method for clocking has to be provided. This

can be done by enabling the “<clock>_process :process” process. Enabling the clock-process means changing the name of the process in a useful name (e.g. “clk_process”), and by adding a clock signal (e.g. “clk”). Generally the clock (clk) has a predefined period of 10 ns.
Last, but not least: the unit under test has to be tested (i.e. stimulated) with a certain testvector. Applying this testvector (i.e. the stimuli) of the uut will be done into the “stim_proc: process” process.

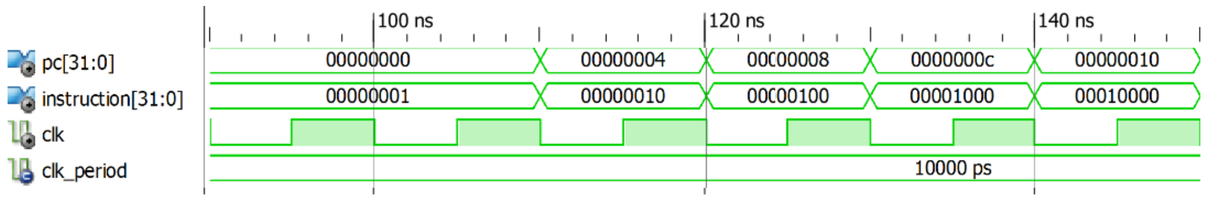
Algorithm 2.1: Stimulus process of the Instruction Memory.

```
-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
    -- insert stimulus here
    PC<=X"00000000";
    wait for clk_period;
    PC<=X"00000004";
    wait for clk_period;
    PC<=X"00000008";
    wait for clk_period;
    PC<=X"0000000C";
    wait for clk_period;
    PC<=X"00000010";
    wait for clk_period;
    wait;
end process;
```

The stimuli of the uut are already defined (see table 2.1). These stimuli represent the program counter signal (PC). Code snippet 2.1 shows how to insert the proper stimuli. Since the program counter is updated every clock-cycle, the instruction updates every clock-cycle. One must read the waveforms as follows: “when pc updates, the instruction updates”. The result of the testbenches are shown in waveforms (figure 2.1). One can remark that at each update of the program counter, the instruction delivers the right value.

Algorithm 2.2: Raw VHDL testbench code generated by Xilinx ISE.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY IMem_tb IS
END IMem_tb;
-- Component Declaration for the Unit Under Test (UUT)
ARCHITECTURE behavior OF IMem_tb IS
    COMPONENT IMem
        PORT(
            PC : IN std_logic_vector(31 downto 0);
            Instruction : OUT std_logic_vector(31 downto 0)
        );
    END COMPONENT;
    --Inputs
    signal PC : std_logic_vector(31 downto 0) := (others => '0');
    --Outputs
    signal Instruction : std_logic_vector(31 downto 0);
    -- No clocks detected in port list. Replace <clock> below with
    -- appropriate port name
    constant <clock>_period : time := 10 ns;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: IMem PORT
    MAP (
        PC => PC,
        Instruction => Instruction
    );
    -- Clock process definitions
    <clock>_process :process
    begin
        <clock> <= '0';
        wait for <clock>_period/2;
        <clock> <= '1';
        wait for <clock>_period/2;
    end process;
    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        wait for <clock>_period*10;
        -- insert stimulus here
        wait;
    end process;
END;
```



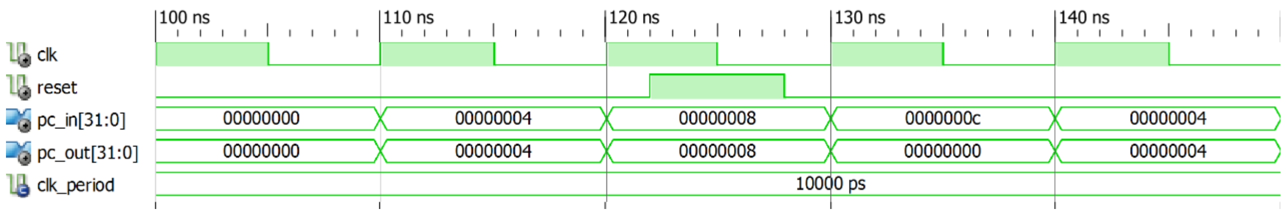
Figuur 2.1: Timings of the Instruction Memory.

2.2 Program Counter

The Program Counter has three inputs signals and only one output signal. The input-output relationship is somewhat more tricky than the one for Instruction Memory. Indeed, aside from the Clk and the PC_IN inputs which are synchronous, the Reset pin needs to be tested in an asynchronous way. Therefore the input-output relation will be accompanied with timings. This is shown in table 2.2. Code snippet 2.3 illustrates the stimulus process. The final waveforms are shown in figure 2.2.
One should however remark that all testbenches start at 100 ns. The first 100 ns are needed for the initialization of the testbench and are therefore not usable for the programmer. Testing before the 100ns could lead to strange and incorrect behaviours!

Time (ns)	Reset	PC_IN	PC_OUT
100	0	0x00000000	0x00000000
110	0	0x00000004	0x00000004
120	0	0x00000008	0x00000008
122	1	0x00000008	0x00000008
128	0	0x00000008	0x00000008
130	0	0x0000000C	0x00000000
140	0	0x00000004	0x00000004

Tabel 2.2: PC input-output relation. Note that the Clk pin has been omitted.



Figuur 2.2: Timings of the Program Counter.

Algorithm 2.3: Stimulus process of the Program Counter.

```
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
    PC_in<=(others=>'0');
    wait for Clk_period;
    PC_in<=PC_OUT+4;
    wait for Clk_period;
    PC_in<=PC_OUT+4;
    -- deal the asynchronous Reset pin
    wait for 2 ns;
    Reset<='1';
    wait for 6 ns;
    Reset<='0';
    wait for 2 ns;
    PC_in<=PC_OUT+4;
    wait for Clk_period;
    PC_in<=PC_OUT+4;
    wait for Clk_period;
    PC_in<=PC_OUT+4;
    wait;
end process;
```

2.3 Program Counter Adder

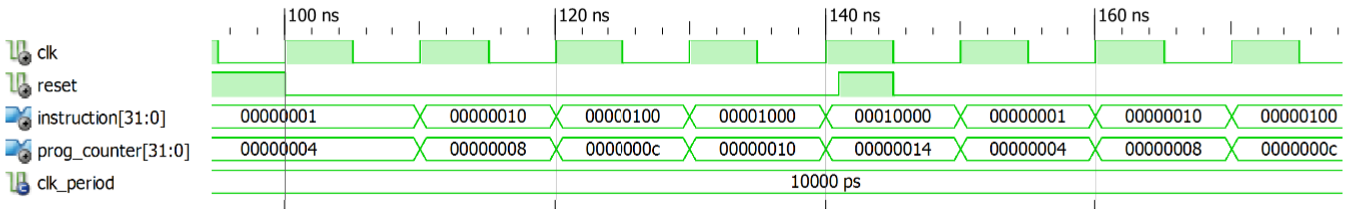
The last module to be tested is the Program Counter Adder. This is left as exercise to the reader.

2.4 The MIPS processor

Testing all the components together happens in the same fashion as tesing a single module. Since the tests are done on all included modules at the same time, the testing vectors need to be representative for all modules. Table 2.3 lists a possible testvector. Note that the first reset is needed in order to initialise the processor. As long as the reset-pin has not been triggered, the processor won't execute the first instruction. Therefore the first 100 ns are spent resetting the processor. Figure 2.3 shows the waveform of the MIPS processor.

Time (ns)	Reset	Prog_counter	Instruction
<100	1	0x00000004	0x00000001
100	0	0x00000004	0x00000001
110	0	0x00000008	0x00000010
120	0	0x0000000C	0x00000100
130	0	0x00000010	0x00001000
140	0	0x00000014	0x00010000
141	1	0x00000014	0x00010000
145	0	0x00000014	0x00010000
150	0	0x00000004	0x00000001
160	0	0x00000008	0x00000010
170	0	0x0000000C	0x00000100

Tabel 2.3: MIPS processor input-output relation. Note that the Clk pin has been omitted.



Figuur 2.3: Timings of the MIPS processor.

3 Porting to FPGA

The last step in the design development is porting the MIPS processor to a real FPGA board. The FPGA has one (or more) clock, digital IO ports. On some FPGAs there are also some DSP capabilities available. Since the MIPS processor only handles digital information (and needs a clock), this chapter will only handle the clock and digital IO.

3.1 User Constraints File

All information the MIPS processor has to share with the outside world, has to pass through the IO-ports of the FPGA. This can be done by redirecting the digital information to the corresponding IO-ports. Redirecting is typically done in an User Contstraints File (i.e. UCF, see Appendix ?? for more info about how to create an UCF). An example of UCF is given in code snippet 3.1.

Algorithm 3.1: UCF of the MIPS procesor. Note that "prog_counter" has not been added in this snippet.

```
NET "clk"             LOC=L15;
NET "reset"           LOC=A10;
NET "instruction<0>"  LOC=U18;
NET "instruction<1>"  LOC=M14;
NET "instruction<2>"  LOC=N14;
NET "instruction<3>"  LOC=L14;
NET "instruction<4>"  LOC=M13;
NET "instruction<5>"  LOC=D4;
NET "instruction<6>"  LOC=P16;
NET "instruction<7>"  LOC=N12;
```

A few points should be discussed here:

- All nets from the schematic (figure 1.5) that have to be connected to IO pins are listed here as NETs (left names).
- The pin locations are listed as "LOC=" followed by the actual portname. A portname contains a character followed by a number, which are representing the indices of a two-dimensional array (BGA - Ball Grid Array of the FPGA chip).
- All nets are inclosed within double quotes.

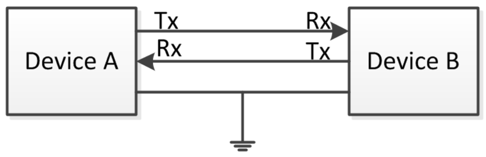
- If a net contains multiple “wires” (i.e. bus), each “wire” is taken by index “<index>”.

4 Additional features

By implementing and connecting all the building blocks as described previously, one should be able to construct the basis of the MIPS processor. This processor will be able to execute the instructions from the Instruction Memory, do some calculations, access the Data Memory for storage, etc. From a conceptual perspective, this is sufficient. However, in real implementations, a processor is almost always implemented with connectivity possibilities to the outside world. One could add the possibility of reprogramming the MIPS processor, some digital IO-ports, a UART communication module, a SPI module, timers, interrupt handling, etc. The UART-communication module and the possibility to reprogram the MIPS processor are explained below. This non-exhaustive list of features can be completed with other user defined features. One should bear in mind that the major requirement to each feature is a complete documentation and reference of usage for the programmer.

4.1 UART module

The Universal Asynchronous Receiver and Transmitter (i.e. UART) consists of a communication module in each device and three parallel wires between 2 devices (figure 4.1). On one hand, the communication module ensures the correct sending and receiving operations of the data. On the other hand, the wires ensure the connectivity between two devices. The connections between the two devices are defined as: a wire for sending (Tx), a wire for receiving (Rx) and a common ground (GND) wire.



Figuur 4.1: UART communication between two devices.

In order to correctly send and receive data, the two devices need to be synchronized. This synchronization comprises the data speed between the two devices, the start and stop sequence and the idle conditions. The data speed is generally chosen at a multiple of 9600 baud per second (Bps). The idle condition occurs when none of the devices is sending data, and is defined by a logic ‘1’ on the transmission lines. In order to start sending a byte, the sender first pulls its transmission TX-line to ground (logic ‘0’) for one cycle. During the next 8 cycles, the sender sends the full byte, bit by bit. Each bit transmission also involves one cycle. After sending one byte, the sender closes the connection by pushing the transmission (TX) line to a logic ‘1’ (also for one cycle). Each cycle duration T_{cycle} in the transmission corresponds to: $T_{cycle} \approx \frac{1}{baudrate}$. Table 4.1 summarises the transmission transaction.

Bit	Value
Idle	1
Start	0
Data[0-7]	X
Stop[1..2]	1

Tabel 4.1: Value table of the UART transaction. Note that the data sent contains 8 databits.

4.1.1 Connecting the UART-module to the MIPS processor

Although the UART communication between two devices is rather simple, the most difficult part resides in the method in which the UART module is interfaced with the MIPS processor. One of the most used methods is done by addressing the UART module in the same way the data memory is addressed. This method - Memory Mapped IO (i.e. MMIO) - does not require additional instructions, and permits a user to attach many devices on the MIPS processor. In order to apply the MMIO methodology to the UART module, one must take some parameters into account. In order to efficiently send and receive data (bytes), the module must be able to accomplish following tasks:

- Send and receive data, a small buffer needs to be provided in order to temporarily store data.
- Send the data when the processor requests it.
- When data is received, the processor is notified of data reception.

To facilitate the interaction with the processor, the UART module will be implemented with addressable registers. These registers contain the send and receive buffers, a status register and a configuration register. These buffers can be written (send-buffer) and read (receive-buffer) from the processor. The status register can have multiple purposes. The processor can read the current state of the UART module (i.e. is it receiving data, is the sending done, etc.), the processor can assign some values to this register in order to let the module take actions (i.e. sending data). The configuration register is used to assign the necessary parameters to the UART module (i.e. the baud rate). An example of such MMIO addressing is given in table 4.2.

Functionality	Address
UART	0xE1000000 to 0xE1000040
Send-buffer (16 bytes)	0xE1000000 to 0xE1000010 (0 bytes offset)
Receive-buffer (16 bytes)	0xE1000010 to 0xE1000020 (16 bytes offset)
Status register (2 bytes)	0xE1000020 (32 bytes offset)
Configuration register (2 bytes)	0xE1000030 (48 bytes offset)

Tabel 4.2: Different addresses within the UART device.

4.2 Reprogramming the MIPS processor

In previous chapters we introduced the methods on which instructions the instructions are stored and fetched by the MIPS processor. The implementation described uses a fixed memory with a predefined sequence of instructions. By doing so, the processor needs a complete rebuild with the VHDL implementation tools. To prevent this fastidious task, one can also make sure the MIPS processor can be reprogrammed. To fulfil this, some points need to be discussed.
Firstly, the processor needs to be able to acquire new binary code. To do so, one can add a simple UART receiver, alongside with an internal Instruction Memory flasher mechanism. The internal flasher has two purposes. It ensures that the received bytes are grouped together and stored into the Instruction Memory. Aside of that, it also resets the complete processor by resetting the Program Counter to the first address. To disable further execution of the MIPS processor during the reprogramming, the flasher also locks the Program Counter to the first address.
One important point to be discussed here is the code transaction between the programming computer and the MIPS processor. Although the UART communication might seem to robust enough, the programming sequence needs to be secured against wrong input (i.e. a regular UART communication instead of the programming sequence). Therefore, before programming, the programming computer must send a programming sequence before proceeding to the programming. In the processor, the flasher checks the sequence before allowing to reflash the Instruction Memory. After the processor has been reflashed, the complete sequence must be checked on integrity. The integrity check ensures that all instructions are correctly passed from the computer to the MIPS, and no hazardous or harmful instructions have been sent and flashed. This check can be applied padding the programming sequence with the exclusive OR of all the bytes. The processor can check this exclusive OR by comparing the complete sequence against the expected result (i.e. 0, check this out!).

A Xilinx ISE software

A.1 Creating a new project

- Goto “File” → “New Project”
- Choose name and location of the new project. Select “HDL” for a VHDL toplevel, or “schematic” for a scheme as toplevel. Click “Next”.
- Select the appropriate device (not important for testbenches). Select “VHDL” in the “Preferred Language” option. Click “Next”. Click “Finish”.

A.2 Adding a new VHDL-module

- Right-click ont the project root (device name in the “Design view”) and select “New source”.
- Select “VHDL Module” in the left pane. Choose a name and click “Next”.
- Enter the necessary ports (can also be done by writing the code by hand) and change entity name if desired. Click “Next”. Click “Finish”.

A.3 Creating an User Constraints File

- Select the target toplevel in the “Design view”.
- In the “Processes” view, unfold the “User Constraints” option.
- Double click on “Floorplan Area/IO/Logic (Planahead)”. Click “Yes” at next window.
- The UCF has been added to the project. I a new window opens, close it.
- Another trick is to manually create an UCF in the same folder as the project, and to add it afterwards into the project.
- Enjoy editting the UCF!

A.4 Testbenches

A.4.1 Creating testbenches

- Right-click ont the project root (device name in the "Design view") and select "New source".
- Select "VHDL Test Bench" in the left pane. Choose a name (typically ends with "_tb") and click "Next".
- Select the to module that has to be tested. Click "Next". Click "Finish".

A.4.2 Running testbenches

- Switch to "Simulation" in the "Design view".
- Select the desired testbench in the "Design view".
- Unfold the "ISim Simulator" node in the "Processes view". Double click on "Simulate Behavioral Model".
- If all went well a new (ISim) window opens with the waveforms.
- One can relaunch (after modifying the testbench code) the simation from ISim by clicking "Re-launch" in the ISim window.
- In the waveforms right-click and "Radix" and click on the desired value representation within the waveforms.