

Design of Digital Systems: Processor design

Group 4:

Arnaud Lhomme (s2397447), a.r.y.lhomme@student.utwente.nl, EMSYS
Ricardo Ampudia (s2316161), r.ampudiahernandez@student.utwente.nl, EMSYS
René Nijhuis (s1468677), r.h.a.nijhuis@student.utwente.nl, EMSYS
Xinyu Tian (s2221306), x.tian-1@student.utwente.nl, EMSYS

April 26, 2020

Contents

0	Introduction	3
1	Behavioural description	4
1.1	Test program	5
2	Algorithmic description	7
2.1	Subtraction	7
2.2	Logical operators	8
2.3	Branch on equal	8
2.4	Branch on greater than or equal than zero	8
2.5	Set on less than	9
2.6	Multiplication	9
2.7	Division	12
2.8	Other instructions	13
2.9	Test environment	13
3	Datapath and controller	14
3.1	Behavioural controller	14
3.2	Datapath	15
3.3	Testing of the datapath	16
3.4	Clock cycles per instruction	17
4	Testing the design with $1000e^x$	17
5	RTL description of the datapath	18
5.1	Area	18
5.2	Timing	22
5.3	Post-simulation	24
5.4	Power estimation	25
6	Final notes	27
6.1	Unresolved issues	27
7	Overview of team member activities	28

0 Introduction

The document is divided according to the provided design flow steps given for the assignment. The following instructions will provide a summarized guide on how to simulate each stage:

1. Create new ModelSim project in base folder.
2. Add all VHDL files inside the *base*, *Behaviour*, *algorithms* and *datapath* folders (26 files in total)
3. Add the VHO file *datapath/datpath.vho*
4. To simulate Behavioural:
 - File → Load → compile_behaviour.do
 - File → Load → sim_behaviour.do
 - run (e.g. run 100 us)
5. To simulate Algorithmic:
 - File → Load → compile_algorithmic.do
 - File → Load → sim_algorithmic.do
 - run (e.g. run 100 us)
6. To simulate Datapath and controller:
 - File → Load → compile_dpth_ctrl.do
 - File → Load → sim_dpth_ctrl.do
 - run (e.g. run 100 us)
7. To simulate $1000e^x$ approximation
 - File → Load → compile_dpth_ctrl_exp.do
 - File → Load → sim_dpth_ctrl_exp.do
 - run (e.g. run 100 us)
8. To run post-simulation:
 - File → Load → compile_dpth_ctrl_postsim.do
 - File → Load → sim_dpth_ctrl_postsim.do
 - run (e.g. run 100 us)

1 Behavioural description

As a first step towards the design of a processor, a behavioural description was developed. In the file *processor_types.vhd* a package was created with user defined data types and sub types to facilitate the decoding of instructions and the manipulation of op-codes. Among those contained in the package are sub types of "std_logic_vector" with different lengths (bit64, bit32, bit16, etc.) to handle bit operations.

```
SUBTYPE bit32 IS std_logic_vector (31 DOWNT0 0);
```

A *registers* type was created with an array of bit32 to use for the general purpose registers inside of the processor. Additionally, the package contains several constants for the R-Type and I-Type instructions that match their names to their corresponding binary operation codes.

The file *proc_entity.vhd* contains the definition of the entity *processor*. This entity contains all the needed signals to communicate with the memory, as well as the clock.

In the file *Behaviour/proc_behaviour.vhd*, the behavioural description of the processor is included. Additionally to the standard "ieee.std_logic_1164" and "ieee.numeric_std" libraries, the *processor_types* package was included along with the *memory_config* provided file. To be able to communicate with the the given memory file the processor entity was defined with the exact same ports (d_busout, d_busin, a_bus, write, read, ready and clk) with and additional reset input port pin.

The *behaviour* architecture which includes one process within contains the complete description of the processor for this part of the assignment. The program counter (*pc*) was defined with a "natural" type to store only positive address values. The variable *instr* is a *bit32* type that includes several aliases for the different parts that conform the instruction such as the operation code *op*, the *rs/rd/rt* and *imm* values that together will tell the processor which instructions to perform and from which location should it obtain the data from. The use of aliases ("ALIAS") greatly facilitates the tasks of decoding the instructions.

```
VARIABLE instr                : bit32;
    ALIAS op                   : bit6 IS instr(31 DOWNT0 26);
    ALIAS rs                    : bit5 IS instr(25 DOWNT0 21);
    ALIAS rt                    : bit5 IS instr(20 DOWNT0 16);
    ALIAS rd                    : bit5 IS instr(15 DOWNT0 11);
    ALIAS sha                   : bit5 IS instr(10 DOWNT0 6);
    ALIAS func                  : bit6 IS instr(5 DOWNT0 0);
    ALIAS imm                   : bit16 IS instr(15 DOWNT0 0);
```

A *register_file* variable of the previously defined *registers* type was initialised containing all zeros. Additionally *rs_int* / *rt_int* / *rd_int* variables of the type "integer" were created for later use to address the use of the register file locations.

Two procedures were taken and modified from the provided *simple_test_memory.vhd* file to be able to read and write instructions from the memory model. Procedure *mem_read* takes the input parameter of the address (*addr*) to the address bus (*a_bus*), sets the write bit to '0' and the read bit to '1', waits for the *ready* bit signal and stores the result coming from the *d_busin* port. Similarly, procedure *mem_write* sends an address to the address bus, sends the data to be stores to *d_busout*, sets the read bit to '0', the write bit to '1' and waits for the *ready* bit signal.

```
mem_read(pc, instr, clk, reset, ready, read, write, a_bus, d_busin);

mem_write((to_integer(signed(register_file(rs_int))+signed(imm))),
register_file(rt_int), clk, reset, ready, read, write, a_bus, d_busout);
```

In the main part of the VHDL description, first the *mem_read* procedure is called to obtain the current instruction and the program counter is incremented by 4 to reach the next instruction that will be read in the next cycle. A big case statement is then created to match the *func* ALIAS contained in bits 5 downto 0 from the current instruction. Depending on the function detected, the case statement will perform different routines corresponding to the behavioural descriptions of each particular instruction.

A *Behaviour/testbench_behaviour.vhd* file is also included to test the design. This testbench connects the ports of the *processor*, *memory* and *testset* entities. The last one, from the file *testset.vhd* contains the clock used in the whole program, ticking at a period of 20 ns. The behavioural description can be compiled using the script *compile_behaviour.do* and simulated with the script *sim_behaviour.do*, which uses the configuration *config_behaviour*.

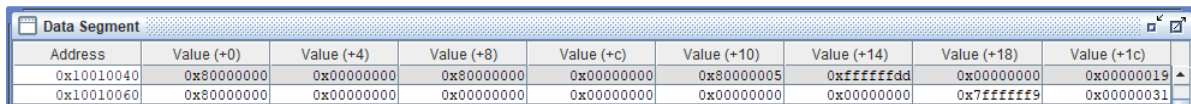
1.1 Test program

To verify that the behaviour description, as well as the algorithmic description that will be made, of the instructions is correct, the assembly file *test_program.asm* has been written and simulated in Mars 4.5. This program contains every test case for every instruction of the processor. The test cases differ for every type of instruction but are based on the following :

- *add, addi, mult, div, slt*: all the possible operations between the most negative/positive integer, a random negative/positive integer and the value 0 (or 1 for the division). e.g. *mult r10, 101, 0*

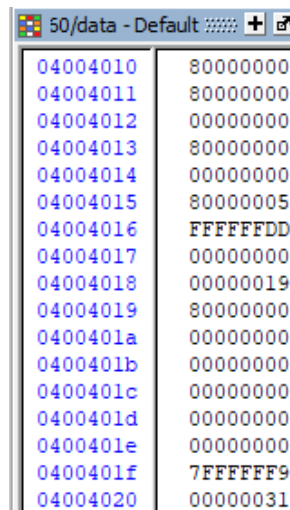
- *and, or, ori*: all the possible combinations between *0x00000000*, *0xFFFFFFFF*, a random value and its inverse. e.g. *and r10, 0xFFFFFFFF, 0xABCDEF01*
- *mfhi, mflo*: these instructions are tested in the same time as *mult* and *div*
- *beq, bgez*: comparisons between positive, negative and equal values. The possibility to jump forward and backward is also tested.
- *lui, lw, sw*: load different values into different registers, load value into a register using its address in memory, store different values into the memory

At the start of the program, the different integer values needed for the program are loaded in the register file. After every test case, or all test cases for instructions *beq* and *bgez*, the result of the operation is store into the data memory at a different address every time. Thus, when the instruction *nop* is reached at the end of the test program, comparing the data memory obtained by simulation in Mars and in the simulation of the processor allows to determine if the processor has the correct behaviour. Figures 1 and 2 show the results of both simulations are the same.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010040	0x80000000	0x00000000	0x80000000	0x00000000	0x80000005	0xffffffff	0x00000000	0x00000019
0x10010060	0x80000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x7fffffff	0x00000031

Figure 1: Data segment - MARS



Address	Value
04004010	80000000
04004011	80000000
04004012	00000000
04004013	80000000
04004014	00000000
04004015	80000005
04004016	FFFFFFDD
04004017	00000000
04004018	00000019
04004019	80000000
0400401a	00000000
0400401b	00000000
0400401c	00000000
0400401d	00000000
0400401e	00000000
0400401f	7FFFFFF9
04004020	00000031

Figure 2: Data segment - ModelSim

2 Algorithmic description

The algorithmic design, present in the file *algorithms/proc_algorithmic.vhd*, has the same pattern and uses the same entity than the behavioural description. The difference is that it use algorithmic description of the different operations.

Alternative algorithms were explored for each of the instructions and created as VHDL packages inside the *algorithms* folder. The following subsections describe the alternatives that were considered for instructions (not all instructions have alternatives).

2.1 Subtraction

The first algorithm written for the subtraction operation is based on the principle that " $x - y = x + (-y)$ ". To obtain the opposite of a signed integer, the 2's complement of that value has to be calculated with this formula : " $y' = y + 1$ ". Once the opposite of the second input vector is obtained, the two vectors just need to be added.

The second algorithm is the algorithm of the binary subtraction, and it is the algorithm selected in the design. In this algorithm, a bit by bit subtraction is performed. A bit subtraction has the following four cases:

- $0 - 0 = 0$
- $1 - 1 = 0$
- $1 - 0 = 1$
- $0 - 1 = 1$, borrow = 1

Like in a decimal subtraction, the borrow affects the next digit, in this case bit. The previous cases are cases when there is no borrow on the bit subtraction. When there is one, the cases are inverted to the following:

- $0 - 0 = 1$, borrow = 1
- $1 - 1 = 1$, borrow = 1
- $1 - 0 = 0$, borrow = 0
- $0 - 1 = 1$, borrow = 1

These 8 possibilities are applied to every bit of the input vectors, and the result of each bit subtraction is stored in the output vector.

Algorithm 1 was selected to reuse the adder and obtain a more area efficient implementation. The functions for the subtractions algorithms are in the file *algorithms/sub_instr.vhd*.

2.2 Logical operators

In order to reproduce the behaviour of the AND instruction, two algorithms were created. In the first one, the 32 bits `std_logic_vector` used for the output is initialised to '0'. Then a loop goes through all the bits of the two inputs vectors. When the bit 'n' of both input vectors is '1', then the bit 'n' of the output is set to '1'. At the end of the loop, the output vector contains the result of "input1 AND input2". The difference between the first and the second algorithm is that, in the second algorithm, the output vector is initialised to the value of the first input vector. Thus, in the loop, only the second input vector is checked. When a bit of the second input vector is '0', the same bit in the output vector is set to '0'. The functions can be found in the file *algorithms/and_instr.vhd*.

The algorithms for the OR instruction follow the same pattern as the algorithms for the AND instruction. The difference is that in the first algorithm, it is not needed to have the two input vectors to have one bit to '1' to set the output bit to '1', only one input vector is necessary. The second algorithm is the same as the AND second algorithm except that in this one, if the second input vector has one bit to '1', the same output bit is set to '1'. The ORI instruction has the same algorithms as the OR instruction. The difference is that the output vector is always initialised to the first input vector and that only the 16 lower bits are checked by the function. For this reason the second algorithm was chosen for the *and*, *or* and *ori* instructions. The functions can be found in the files *algorithms/or_instr.vhd* and *algorithms/ori_instr.vhd*.

2.3 Branch on equal

To algorithm used to compare the content between registers was based on the logic gate XNOR which produces a '1' when the operands are the same and a '0' when they are different. Based on this equality property of the logic gate, a bitwise XNOR operation is done for the corresponding bits of the operators. Since every bit must be the same between registers to guarantee equality, the AND logic gate was used between XNOR operation. A function "*beq(rt,rs)*" was created and returns an output of '1' when the 32 bits of Rt and Rs are equal and '0' otherwise.

2.4 Branch on greater than or equal than zero

The algorithm for this instruction is simple, since a signed representation is being used we just need to test the most significant bit of the value contained in the register position Rs. If bit number 31 of the value under test is '0' then it is greater or equal than zero, then the program counter is increased by the immediate (*imm*) value provided.

2.5 Set on less than

A function "slt(rt,rs)" was created for this instruction's algorithm. The approach used to verify if the value in Rt is greater than Rs was to reuse the ALU and perform the operation $Rt+(-Rs)$ and evaluate the result of the operation to determine the outcome. Some design alternatives were building a 4 bit magnitude comparator or doing a subtraction $Rt-Rs$, but ultimately the approach $Rt+(-Rs)$ was selected to reuse the ALU and thus obtain a better area usage. A "comparator_slt" variable was created to store the intermediate result of $Rt+(-Rs)$ and two's complement was used to represent $-Rs$ (negate all bits of Rs and add 1). Then 4 cases were used to separate the results based on the sign bit of Rt and Rs. If Rt and Rs have different signs then if the MSB of Rt contains a '0' it will be greater than Rs; the opposite also holds true. If Rt and Rs have different signs then the sign bit of the comparator_slt variable with the result of $Rt+(-Rs)$ is examined. If the result is positive (MSB is '0') then it means that Rt is bigger than Rs, in the contrary the result is negative (MSB is '1') meaning Rs is bigger than Rt. According to this cases the output returns X"00000001" when Rt is bigger than Rs and X"00000000" when not.

2.6 Multiplication

With saving areas as priority in mind, three alternative algorithms for multiplication are considered, which are add-and-shift multiply algorithm, Booth's multiplication algorithm and multiply with separated high and low bits respectively. The functions for all the multiplication algorithms can be seen in the file *mult_instr.vhd*.

Unsigned add-and-shift algorithm, which can be seen in figure 3, has been explained in the lecture. In which (a) shows the conventional method and (b) is the refined version with efficient area usage. This algorithm is realised as *function signed_multiply* and *function addshift_multiply*. If there are N 1's in multiplier, there will be N additions. To calculate signed multiplication, cases are discussed separately to transfer signed numbers to unsigned numbers, and then after multiplication, transfer the result back to signed number. Two's complementary representation is introduced for the transformation of signed number to unsigned number, written as *function convert* in the file.

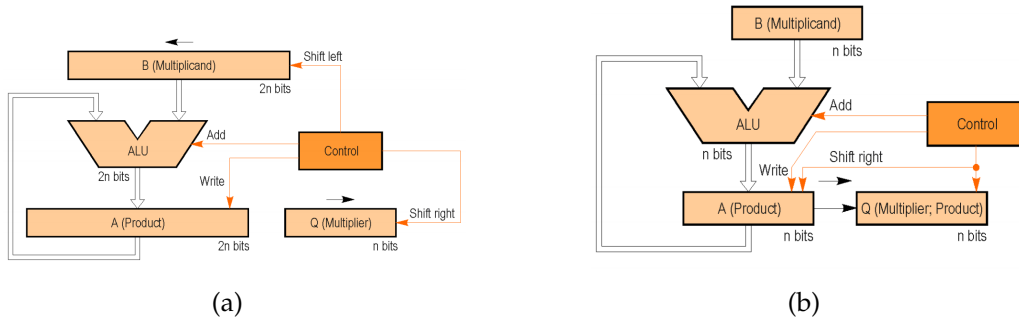


Figure 3: Add-And-Shift Multiply Algorithm

Booth's algorithm can be implemented by repeatedly adding one of two predetermined values A and S to a product P , then performing a rightward arithmetic shift on P , as shown in figure 4a. In each loop, determined by the two least significant bits of P , "01" leads to $P+A$, "10" leads to $P+S$ and do nothing for "00" and "11", followed by arithmetically right shift of P . This repeats 32 times, resulting in the final result, which is the left 64-bit of P . This algorithm is realised as *function booth_multiply*. There is also a revised version of Booth's algorithm as shown in figure 4b, which uses 32-bit ALU instead of 65-bit. This algorithm is realised as *function booth_multiply2*. Booth's algorithm works well with signed numbers. Besides, in the case of multiplier having many consecutive '0' or '1', Booth's algorithm requires less computation. When multiplicand equals the most negative number, since its two's complement notation has an overflow, it needs to be considered separately.

To compare add-and-shift algorithm and Booth's algorithm, add-and-shift requires more control signals and more computation on two's complement. Their hardware implementation needs similar area, and as thinking which one is faster, it highly depends on the input factors. This is reason why Booth's algorithm chosen was chosen for as the final algorithmic description.

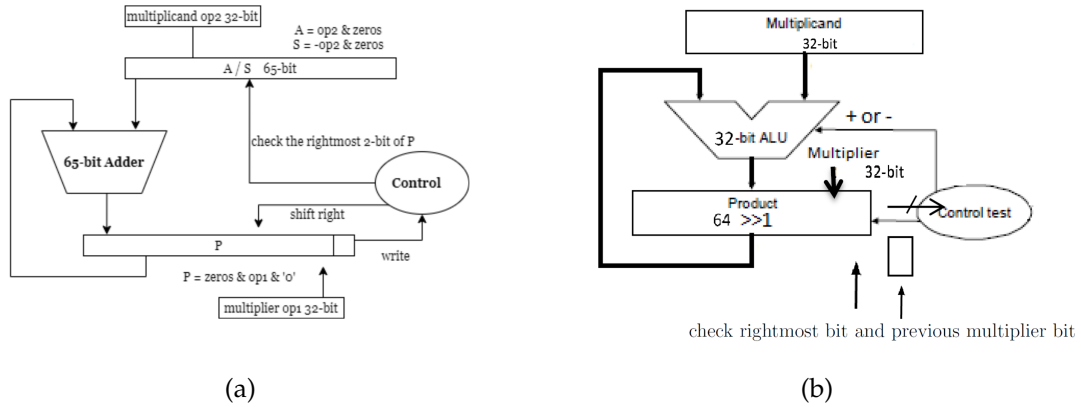


Figure 4: Booth's Multiply Algorithm

Additionally, multiply with separated high and low bits is an algorithm that realises 32-bit multiplication by implementing 4 times of 16-bit multiplications. By doing so, only 16-bit adder is needed instead of 32-bit adder, resulting in a smaller area usage. As shown in figure 5, the idea is to divide op1 and op2 to op1hi, op1lo, op2hi and op2lo and multiply every two of them separately. It uses add-and-shift algorithm on these multiplications. There are four intermediate 32-bit product results, which are located at different bit position based on different factors. The finally result is the accumulation of all the intermediate results (with correct bit position) and it is calculated separately as hi 32-bit and lo 32-bit. The overflow of intermediate addition needs to be carefully dealt with. This algorithm is not ideal because to only use a 16-bit adder, the algorithm requires more multiplexers and complex controller, which probably also consumes a lot of area. This algorithm is realised as function *signed_multiply_separate* and function *multiply_separate*.

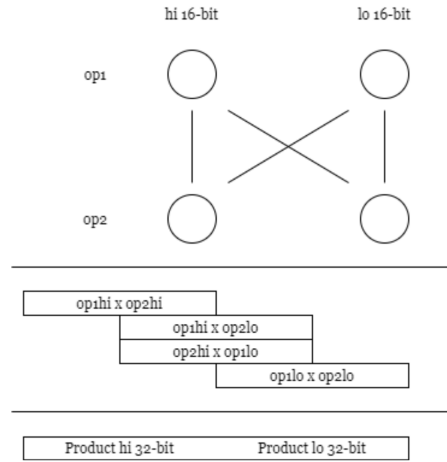


Figure 5: Separate Multiplication

2.7 Division

Three algorithms are written for the division. For these algorithms, if one of the operand is a negative number, the 2's complement is first calculated to have only positive number and the resulted values can be inverted again at the end following these rules:

- Dividend and remainder have the same sign
- Quotient is negative if signs of dividend and divisor differ

For the specific cases where the most negative 32 bits integer is involved, different operations on the sign are performed. When the divisor is the most negative number, it is replaced by 1 and the quotient and remainder are inverted at the end of the calculation. When it's the denominator, the denominator and the remainder are bitwise inverted while the quotient gets a 2's complement operation. The first algorithm is an algorithm based on repeated subtraction while the second one is the long binary division algorithm based on the algorithm of figure 6.

The third algorithm is based on the work made by Takagi et al [2], and is an area efficient type of division. Unfortunately, the function does not work on all the test cases and, hence, the algorithm implemented is the long binary division algorithm.

```

if D = 0 then error(DivisionByZeroException) end
Q := 0 -- Initialize quotient and remainder to zero
R := 0
for i := n - 1 .. 0 do -- Where n is number of bits in N
  R := R << 1 -- Left-shift R by 1 bit
  R(0) := N(i) -- Set the least-significant bit of R equal to bit i of the numerator
  if R ≥ D then
    R := R - D
    Q(i) := 1
  end
end
end

```

Figure 6: Long binary division algorithm

2.8 Other instructions

No alternatives were identified for the instruction `mflo`, `mfhi`, `lui`, `lw`, `sw` and `nop`.

2.9 Test environment

In order to check that the algorithmic version of the processor outputs the correct result, a new entity has been created, called *test_env_algo*. A diagram of the new test environment can be seen in figure 7. The aim of this entity's architecture is to ensure that the algorithmic architecture is reading the same instruction from the memory, and writing the same result to the memory than the behaviour architecture. If the algorithmic one is not reading from the same address, or writing to the same address, an assertion is raised to notify the error. The same happens if the data written to the memory is not the same from both architectures. If the test environment receives the same address, command and data from the two architectures, it transmits everything to the memory and back from the memory to both architectures. With this compare entity between the processor architectures, it is possible to make sure that the algorithmic design gives the correct result at every instruction because it is the only way to reach the *nop* instruction at the end of the program stored in the memory.

To implement this new entity, a new configuration, called *config_algo*, has been created in a new testbench file, called *algorithms/testbench_algo.vhd*. This configuration maps the architecture to the correct definition in the testbench. To compile, and simulate using the correct configuration, the algorithmic design, the two script files *compile_algorithmic.do* and *sim_algorithmic.do* can be executed.

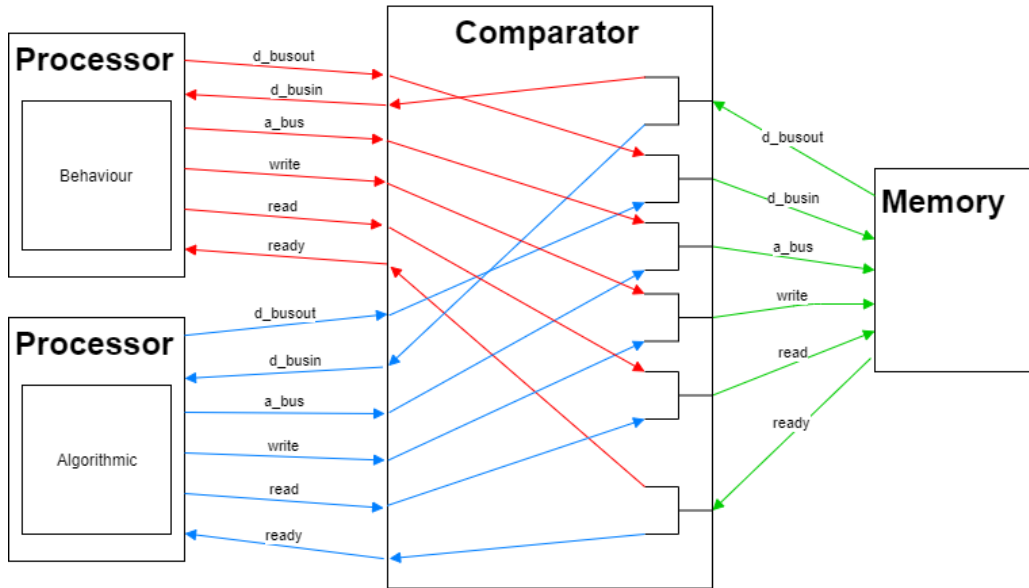


Figure 7: Diagram of the test environment

3 Datapath and controller

3.1 Behavioural controller

The controller entity and its behavioural description can be found in the file *datapath/controller.vhd*. The inputs of the controller are the clock, the operation and function codes of the instruction fetched and the ready signal from the memory to notify that the codes are ready to be read. Depending on the operation codes, the controller sets the bits of the "std_logic_vector" *control_bus*, an output connected to the datapath. This bus is composed of the following elements:

- *init*: used to synchronize the controller and the datapath, so that the datapath doesn't start executing before the instruction is processed by the controller
- *reg_dst*: If '1' the write register of the datapath is "rd", else "rt"
- "branch": Notify the datapath that the instruction is a branch instruction
- *mem_read*: Notify the datapath that the instruction implies a read in the data memory
- *mem_to_reg*: If '1', the data to be written in the write register comes from the memory, else it comes from the result of the ALU

- *mem_write*: Notify the datapath that the instruction implies a write in the data memory
- *reg_write*: Notify the datapath that the instruction implies a write in the register file
- *alu_src*: If '1', the source of the second read register is the immediate value, else "rt"
- *alu_op*: Notify the datapath which operation needs to be performed in the ALU

Each of this parameters is then set to the appropriate value depending on the current instruction.

3.2 Datapath

The datapath entity and synthesizable description can be found in the file *datapath/datapath.vhd*. The datapath receives from the memory the bits 25 down to 0 of the instruction, corresponding to "rs", "rt", "rd" and the immediate value. It also has an input the control bus sent by the controller and all the signals needed to communicate with the memory.

The diagram representing the datapath is shown in figure 8, the blue inputs representing the control bus parameters. The datapath description works as a state machine and is composed of 7 phases, each one active on the rising edge of the clock:

1. **IF**: The PC is written in the address bus to fetch the next instruction, then the datapath stays in this state until the controller sets the control bus
2. **ID**: The values from the control bus are used to determine the registers to read and write, and perform the sign extension on the immediate value.
3. **EX**: The ALU executes the operation and writes the result in "alu_res", or "alu_long_res" for multiplication and division.
4. **MEM**: If the instruction needs to read or write in the memory, the ALU result is written into the address bus and the state changes to "MEM.RD" or "MEM.WR". If not the state changes to "WB".
5. **MEM.RD**: The datapath stays in this state until the memory answers with the data read
6. **MEM.WR**: The datapath stays in this state until the memory acknowledge that the write succeed
7. **WB**: It's in this state the calculation is finished if the operation was a "nop". Otherwise the data is taken from "alu_res" or "mem_res" depending on the control bus

and written into the correct register. For multiplication and division, the data is taken from "alu_long_res" into the HI and LO registers. It's also in this state that the branch occurs and the program counter takes its next value. Then the state changes to "IF" and the procedure starts again.

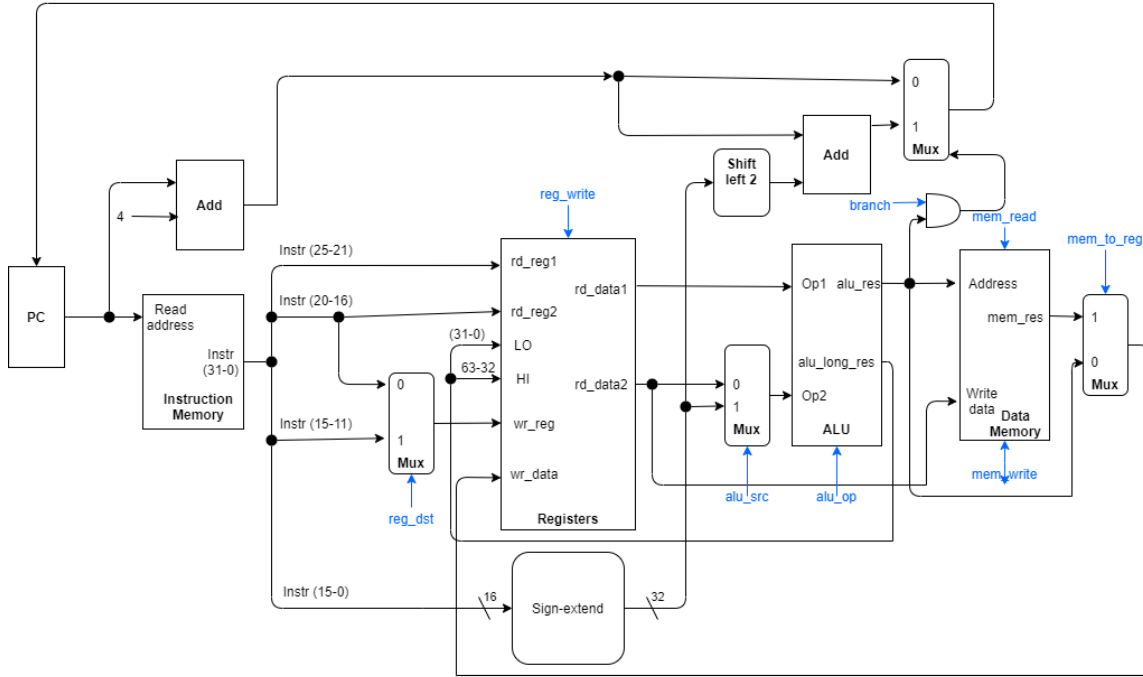


Figure 8: Diagram of the datapath

3.3 Testing of the datapath

Like for the algorithmic description, a testbench (*datapath/testbench_dpth_ctrl.vhd*) and a test environment (*datapath/test_env_dpth_ctrl.vhd*) are created to compare the result of the datapath description with the behaviour description. The testbench maps all the ports of the different entities with the test environment. The test environment acts as the relay between the memory and both description and check on every read and write that the behaviour and datapath descriptions have the same addresses and data. This environment uses the test program mentioned earlier to test every possible cases. It can be compiled using the script *compile_dpth_ctrl.do* and simulate with *sim_dpth_ctrl.do*. When simulated, the "nop" instruction is reached before any other assertions is raised, and the content of the data memory is the same as the one obtained with the MARS simulator, meaning that both datapath and behaviour description give the correct result.

3.4 Clock cycles per instruction

To test how many clock cycles each of the instructions takes to execute, *test_env_dpth_ctrl* was used with slight modifications to ignore the execution of the behavioural processor signals. A test program created was used to verify the time of execution as it uses the complete instruction set. The measurement of clock cycles per instruction (CPI) was measured between the instant the program counter goes from the datapath to the memory through the address bus (a_bus j= pc in state 0) and the time the new instruction starts (again marked with a_bus j= pc in state 0). Table 1 shows the times of execution for the given instruction set:

Table 1: CPI

Instruction	Clock Cycles
bgez	10
beq	10
and	10
or	10
ori	10
addi	10
sub	10
div	10
mflo	10
mfhi	10
mult	10
slt	10
lui	10
lw	14
sw	14

4 Testing the design with $1000e^x$

The given approximation program to calculate $1000e^x$ was tested and the result was validated as correct with the MARS simulator. To test the program it can be compiled first with the macro file *compile_dpth_ctrl_exp.do* and then simulated with *sim_dpth_ctrl_exp.do*. Figure 9 shows the result of the calculation (0x00000170) written to memory data address in 0x04004002 in the ModelSim simulation and figure 10 confirms the expected result obtained with the MARS simulator.

Address	Value
04004000	00000007
04004001	ffffffff
04004002	00000170
04004003	00000000
04004004	00000000

Figure 9: Data segment of $1000e^x$ program in ModelSim

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000007	0xffffffff	0x00000170	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 10: Data segment of $1000e^x$ program in MARS simulator

Figure 11 shows that the time where the data segment is written to memory is of $94.15 \mu s$, considering the clock frequency of 50MHz (1 clock cycle is 20 ns) then it takes 4708 clock cycles to complete the execution.

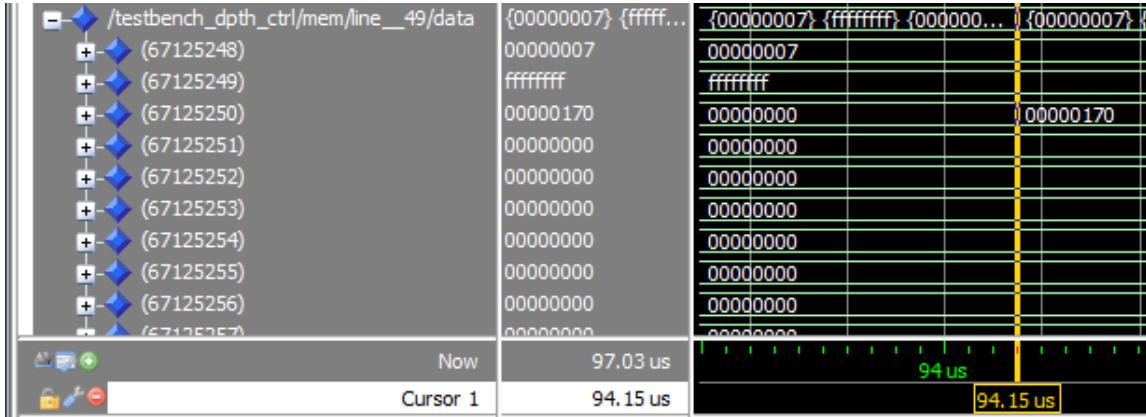


Figure 11: Execution time of $1000e^x$ program

5 RTL description of the datapath

5.1 Area

The developed datapath was then imported to Quartus Prime and synthesised for the Cyclone IV E device EP4CE40F23C6. Figure 12 shows the synthesis summary with 9,695 logic elements used (24% of those available), 1383 registers and 143 I/O pins utilised (43%


of those available). This confirms that the design is synthesizable and well below the maximum capacity for the selected device.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Sat Apr 25 21:57:31 2020
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	datapath
Top-level Entity Name	datapath
Family	Cyclone IV E
Device	EP4CE40F23C6
Timing Models	Final
Total logic elements	9,695 / 39,600 (24 %)
Total registers	1383
Total pins	143 / 329 (43 %)
Total virtual pins	0
Total memory bits	0 / 1,161,216 (0 %)
Embedded Multiplier 9-bit elements	0 / 232 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 12: Flow summary - Quartus Prime

Looking further into the Resource Usage Summary, Figure 13 shows a detailed description of the logic elements used. The Look Up Tables (LUTs) required are mostly 3 and 4 input logic functions amounting to 5,161 and 3986 cells respectively. Looking into the Technology Map Viewer most 3 and 4 input LUTs can be traced back to additions and comparators as shown in Figure 14.

Analysis & Synthesis Resource Usage Summary

 <<Filter>>

	Resource	Usage	8	▼ Total registers	1383
1	Estimated Total logic elements	10,766	1	-- Dedicated logic registers	1383
2			2	-- I/O registers	0
3	Total combinational functions	9678	9		
4	▼ Logic element usage by number of LUT inputs		10	I/O pins	143
1	-- 4 input functions	3986	11		
2	-- 3 input functions	5161	12	Embedded Multiplier 9-bit elements	0
3	-- <=2 input functions	531	13		
5			14	Maximum fan-out node	clk~input
6	▼ Logic elements by mode		15	Maximum fan-out	1383
1	-- normal mode	5468	16	Total fan-out	36948
2	-- arithmetic mode	4210	17	Average fan-out	3.26

Figure 13: Resource usage - Quartus Prime

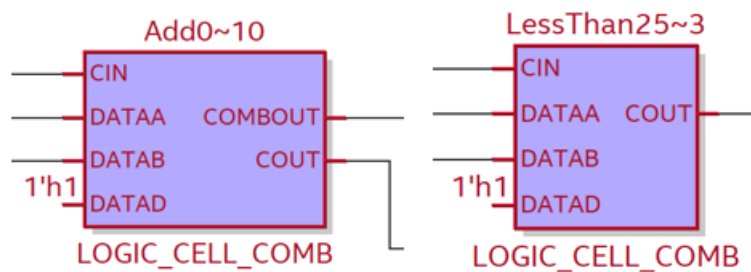


Figure 14: Technology map viewer of 2,3 and 4 input LUTs - Quartus Prime

Figure 15 shows part of the processor RTL schematic which gives an overview of the structure of the design. On the left of the schematic, the inputs correspond to the instructions coming from memory (rs, rt, rd, imm) and the control bus from the controller. Also few adders going to the program counter can be seen as expected. The big block in the center corresponds to the register file and on the right of the diagram small registers can be seen for the memory and alu results. This corresponds to the sequential description of the stages given in VHDL by means of the states.

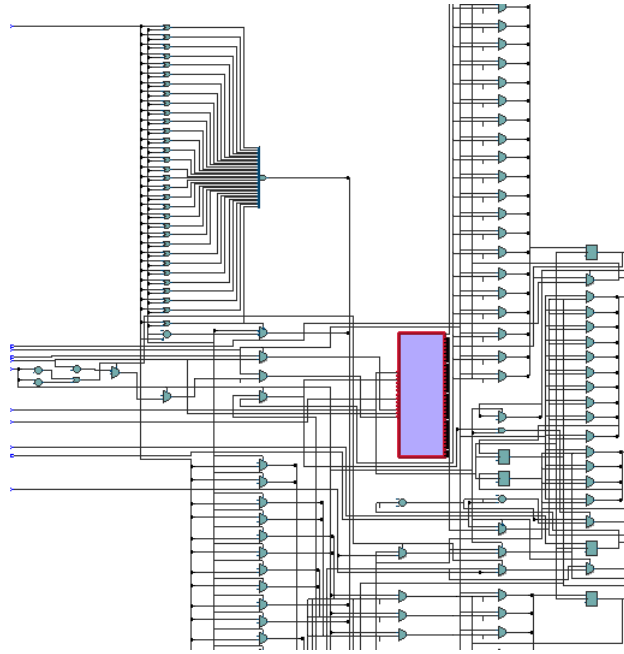


Figure 15: RTL processor schematic - Quartus Prime

Additionally to the processor schematic, Figure 16 shows other big blocks of hardware corresponding to the multiplication and division implementations. The result of the ALU in our design is always given in one clock cycle, which in the case of complex operations like multiplication and division comes at the expense of a lot of hardware components with heavy data dependent paths. This result is discussed further in the last section.

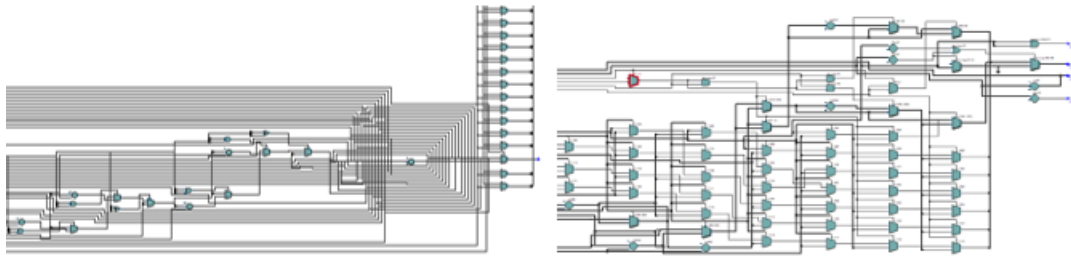


Figure 16: RTL schematic of multiplication and division - Quartus Prime

5.2 Timing

To perform timing analysis an SDC file was created with a clock period of 20ns corresponding to the frequency of 50Mhz and clock uncertainty was added as a parameter to amount for clock jitter. A delay of the input ports was considered with min and max values of 2ns and 5ns respectively, for the outputs min and max constraints of -3ns and 7ns were added (negative value indicates that the output could change before an active edge of the clock). The SDC file used can be found in "*Quartus/Timing/SDC1.sdc*".

Figures 17 and 18 show the setup and hold times for the input and output ports. The maximum setup time which could be the most problematic is found for the *rs* signal which should be stable 12.139ns before the clock signal. The minimum hold time is for the *rs* signal which can change -2.173 ns before the clock edge and still be captured properly. The maximum hold time is for the control bus which can be detected properly after 0.215ns of the clock edge.

Setup Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	> control_bus[*]	clk	9.015	9.556	Rise	clk
2	> d_busin[*]	clk	2.599	3.193	Rise	clk
3	> imm[*]	clk	5.670	6.484	Rise	clk
4	> rd[*]	clk	2.209	2.729	Rise	clk
5	ready	clk	3.721	4.193	Rise	clk
6	> rs[*]	clk	12.139	12.726	Rise	clk
7	> rt[*]	clk	10.043	10.695	Rise	clk

Figure 17: Setup times - Timing analyzer

Hold Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	> rs[*]	clk	-2.173	-2.667	Rise	clk
2	ready	clk	-1.924	-2.409	Rise	clk
3	> imm[*]	clk	-1.518	-1.995	Rise	clk
4	> rt[*]	clk	-1.358	-1.815	Rise	clk
5	> d_busin[*]	clk	-1.085	-1.517	Rise	clk
6	> rd[*]	clk	-1.025	-1.412	Rise	clk
7	> control_bus[*]	clk	0.215	0.107	Rise	clk

Figure 18: Hold times - Timing analyzer

Figures 19 and 20 show the minimum and clock to output times; all the output signals take between 6 and 8.5ns to be available after a clock edge.

Clock to Output Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	> a_bus[*]	clk	8.546	8.755	Rise	clk
2	> d_busout[*]	clk	8.256	8.363	Rise	clk
3	read	clk	6.531	6.557	Rise	clk
4	write	clk	7.547	7.615	Rise	clk

Figure 19: Clock to output times - Timing analyzer

Minimum Clock to Output Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	> a_bus[*]	clk	5.961	5.950	Rise	clk
2	> d_busout[*]	clk	6.011	6.021	Rise	clk
3	read	clk	6.321	6.344	Rise	clk
4	write	clk	7.300	7.364	Rise	clk

Figure 20: Minimum clock to output times - Timing analyzer

One problematic result obtained from the timing analysis is shown in Figure 21 a negative slack value for the *alu_long_res* signal which indicates that the timing requirements cannot be met. This is explained due to the large critical path observed for multiplication and division observed in the previous section and will be discussed further in the last section of the report.

Slow 1200mV 85C Model								
	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-138.585	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.835
2	-138.582	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.832
3	-138.582	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.832
4	-138.580	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.830
5	-138.579	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.829
6	-138.577	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.827
7	-138.577	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.827
8	-138.574	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.824
9	-138.573	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.823
10	-138.573	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.823
11	-138.572	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.822
12	-138.570	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.820
13	-138.570	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.820
14	-138.570	rd_reg2[29]	alu_long_res[54]	clk	clk	20.000	0.255	158.820

Figure 21: Failing paths - Timing analyzer

5.3 Post-simulation

After the compilation and synthesis is done in Quartus, a *datapath.vho* file is produced. This file is added to the project to replace the vhd file already present. To compile the project with post simulated file obtained, the script *compile_dpth_ctrl_postsim.do* can be used, and the script *sim_dpth_ctrl_postsim.do* should be used to start the simulation. This simulation will test the design with the $1000e^x$ test program, always comparing it with the behavioural description. Because the size of the design is very large, the simulation runs at a very slow pace (10 minutes to finish the simulation) but the "nop" instruction at the end is reached, and the content of the data memory is the same as obtained in section 4, meaning that the post simulation design has the same result as the original one (figure 22 for proof).

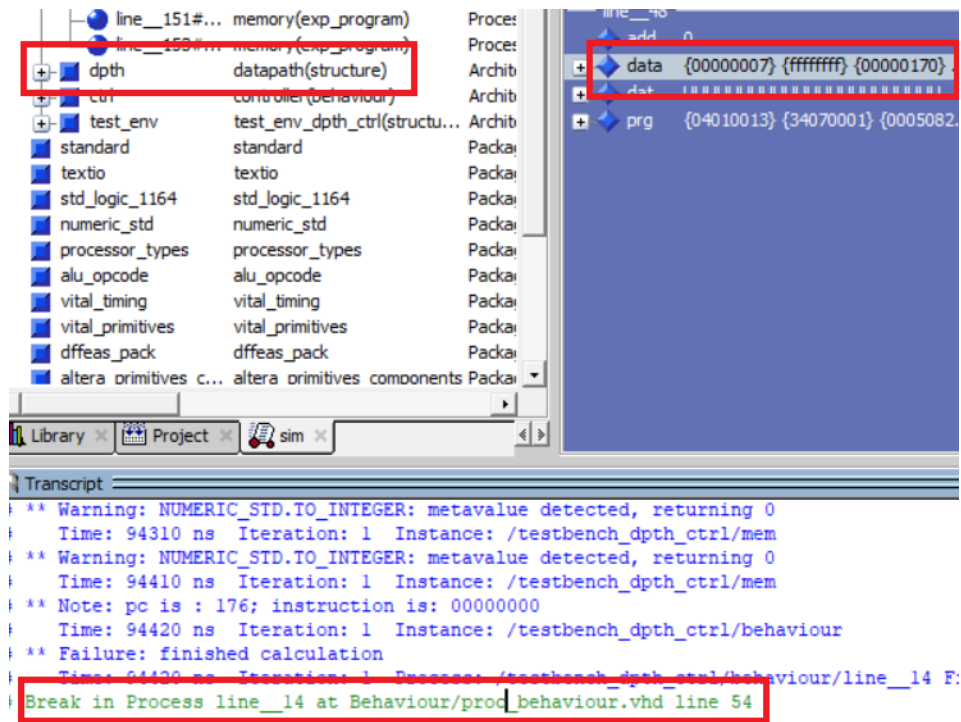


Figure 22: Post simulation result

5.4 Power estimation

Following the procedure provided by the Power Analyzer manual, after post-simulation of the design, a *vcd* file containing the toggle information of all the signals was obtained with ModelSim. This file can be found in "Quartus/Power/datapath.vcd" and was used as input for the Power Analyzer in Quartus Prime. Figure ?? shows the most relevant power information, indicating that the total thermal power dissipation is of 153.19W. The summary also shows a high confidence of the estimation due to the fact that a sufficient amount of data was provided. The dominant power consuming feature is the static power dissipation (55%) which is to be expected as the FPGA is not close to full resource usage. I/O also has an important power consumption (34%) due to a large amount of pins being used for communication with the memory through the address and data buses.

Power Analyzer Summary	
<<Filter>>	
Power Analyzer Status	Successful - Sun Apr 26 11:39:26 2020
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	datapath
Top-level Entity Name	datapath
Family	Cyclone IV E
Device	EP4CE40F23C6
Power Models	Final
Total Thermal Power Dissipation	153.19 mW
Core Dynamic Thermal Power Dissipation	15.00 mW
Core Static Thermal Power Dissipation	85.36 mW
I/O Thermal Power Dissipation	52.83 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Figure 23: Summary - Power analyzer

Further detailed information about thermal consumption by block type can be seen in Figure 24 providing information of block and routing power leaks which for this design represent a small percentage of the total except for I/O routing (14 %).

Thermal Power Dissipation by Block Type				
<<Filter>>				
	Block Type	Total Thermal Power by Block Type	Block Thermal Dynamic Power	Block Thermal Static Power (1)
1	Combinational cell	6.89 mW	1.86 mW	--
2	Clock control block	5.67 mW	0.00 mW	--
3	Register cell	2.04 mW	1.68 mW	--
4	I/O	23.22 mW	0.77 mW	22.05 mW

Routing Thermal Dynamic Power		Block Average Toggle Rate (millions of transitions / sec)	
5.03 mW		0.882	
5.67 mW		100.000	
0.36 mW		0.141	
0.40 mW		1.710	

Figure 24: Power dissipation - Timing analyzer

6 Final notes

The processor design implemented is successful in the areas of behavioural, algorithmic and datapath/controller simulation in ModelSim. It proved to be functional when using the test programs and is able to synthesis, fit in the selected device and works in post-simulation.

6.1 Unresolved issues

As mentioned in the previous sections, multiplication and division alone use a large amount of hardware to be able to compute the ALU long result. In theory this has the advantage of making the result available in 1 clock cycle but this presents two issues with area and timing requirements.

1) Area optimization for multiplication and division

Due to the amount of hardware elements, a long critical path is generated from the operators of the multiplication and division to the output of *alu_long res*. As the goal of the design was to optimize for area, this is not a very efficient design for those particular instructions.

2) Timing requirements not met for ALU long result

A second issue arises from the same cause the Timing Analyzer in Quartus indicates that the timing requirements cannot be met. This is due to the propagation delay caused by each hardware element that is part of the critical path from the operators to the ALU result.

Solution: time-area trade-off

It appears to be clear that the solution for both of these issues would be a redesign of the multiplication and division algorithms. Due to the data-dependent nature of the multiplication and division operations in which a previous result is needed to perform the next calculation, a good solution would be to place registers in between each intermediate result and reuse hardware. This would come at the cost of more clock cycles to complete the instructions and would also be necessary to build control logic. Additionally an *alu_ready* signal would be needed to indicate that the ALU result is available for the next stages of the MIPS cycle (MEM and WB).

Although this is a more complex design solution than the current one, it is a good candidate to solve the timing issues (as it would reduce the critical path) and it would optimize the design in terms of area.

7 Overview of team member activities

Arnaud Lhomme

The main activities I was involved for the project were:

- Behavioural description of the processor
- Algorithmic description of the processor
- Definition of expected test case outputs
- Algorithms: and, addi, or, ori, div
- Test programs: and, addi, or, ori
- Assemble complete test program
- Developed testset, tesbench, test environment and configurations
- Development of datapath and controller (blocks, control signal, stages)
- Block diagram
- Post simulation of design
- Report

Ricardo Ampudia

The main activities I was involved for the project were:

- Behavioural description of the processor
- Definition of expected test case outputs
- Algorithms: beq, begz and slt
- Test programs: beq, begz and slt
- Development of datapath and controller (blocks, control signal, stages)
- Tested synthesis of initial designs in Quartus to redesign datapath and controller (removed loops and latches to make description sythesizable)
- Tested design against given approximation of $1000e^x$
- Clocks per instruction table

- Timing and area analysis
- Power analysis
- Report

René Nijhuis

- Behavioural description of the processor
- Definition of test cases
- Algorithms: div

Xinyu Tian

- Behavioural description of the processor
- Algorithms: mult

References

- [1] Darshan Shah. *Different Multiplication Algorithm and Hardware*. Nirma University, 2012.
- [2] N. Takagi, S. Kadowaki and K. Takagi. *A hardware algorithm for integer division*. Computer Arithmetic, 2005.