Politecnico di Torino

III Facoltà di Ingegneria

Laboratory 3
RISCV Processor

Master degree in Electronic Engineering

Authors: Group 21

Dilillo Nicola S284963
Moncalvo Stefano S290315
Carrano Lorenzo S281565

GitHub Repository

Contents

CHAPTER 1

Introduction

During the verificatoin steps of HDL design different testbench had been used to check the correctness of implementation. Each of them has been manualy modified to verify the correct bheavior of design under different input oand no automatic tools had been used in order to automatize this process.

The UVM is the lastest tecniquie used to create modular testbench. It's a free framework based on System Verilog and exploid the Object-Oriented paradigma. This paradigm entails the separation of concerns while building the testbench architecture and the specific test that can be performed: multiple tests can be applied to the same architecture just by modifying the input stimuli sequence (and constraints) and by selecting different Design Under Test (DUT) properties to be observed. Alternatively, same tests can be applied to different testbench architectures.

In this laboratory the generation of different input has been randomly and the right output to compare has been elaborated based on a reference model, implemented through the UVM class, properly create according the DUT. In this way the only operation to perfome in order to change the testbench are:

- choose the proper DUT
- select a range for input value
- modify the model reference and the FSM to follow the design bheavior

CHAPTER 2

Adder

First step has been test the correctness of the behavior of the SystemVerilog description of an adder by using QuestaSim. In order to evaluate performance, in terms of coherence with the expected behavior, it is needed to check the amount of matches and mismatches. For each input sequence, it was observed that the output of the adder description to be tested and the output of the gold model were coincidental, thus we can conclude that the overall behavior of the described adder is correct.

2.1 Changing the parallelism

Changing the parallelism means to modiify, of course, both the interface of the unit under test and size of each input/output sequence. Even in this case, no mismatches have been detected by comparation with the golden model.

2.2 Constraints Definition

It is possible to define some constraints over the randomly generated input sequences employed in the testm, for example using the inside operator, to define a range of values for one of the inputs of the adder. Even in this case, no mismatches have been observed.

2.3 Uncorrect Gold Model

At this point, a gold model with an uncorrect behavior has been employed for the test, and thus in this case it is expected to observe mismatches inside each record of the transcript. For each mismatch, a warning is signaled in the transcript, and an error is reported inside the final summary.

2.4 Forcig Overflow Condition

At this step the behavior of the unit under test in case of overflow has been tested. In order to do that, it has been necessary to set proper constraints over the input sequences. It has been possible to observe that the adder correctly handles overflow conditions.

CHAPTER 3

Control Unit

The control unit is the entity that regulates the control signals of the pipeline and it has been designed hardwired. It receives as input the OPCODE, as well as the FUNC3 and FUNC7 when present. According to those values it generates the correct control signals for the components present in the datapath. The signals are then pipelined in order to arrive to the corresponding component exactly when they are needed. Specifically the OPCODE specifies what kind of instruction has to be executed, while the FUNC codes are used to control the operation performed by the ALU.

In case of stall the control unit sees nothing and external component switch the value of control signal.

The control signals are:

- branch, in case of conditional branch instruction, wait for the computation of the condition before performing or not the jump;
- branch_j, in case of unconditional instruction, jump always;
- ALUSrc, to choose between value of source register two and immediate;
- ALUSrc_PC, to choose between value of source register one and program counter;
- ALU_control_signals, to choose the ALU operation;
- MemWrite, to write on data memory;
- MemRead, to read from data memory;
- RegWrite, to write in register file;
- MemToReg, to decide if write value back from read data memory or from ALU result.

CHAPTER 4

Floating Point Multiplier

4.1 Testbench Modifications

The floating point multiplier, in which the MBE multiplier tested in the previous section is used for the multiplication of the significands, is a pipelined architecture. This means that the results of the multiplications are ready some clock cycles after the generation of the inputs. In order to compensate for the added latency, the verification has to be delayed with respect to the generation of the inputs. After an initial latency of six clock cycles necessary to perform the first multiplication, the testbench handles the multiplier in a pipelined way, performing and verifying one operation every three clock cycles. This is possible by saving the generated inputs in a chain of two registers; since the FSM is composed of three states, this is equivalent to delaying the inputs of six clock cycles. When a new result is ready, it is confronted with the correct one computed using the inputs present in the second register of the chain. The FSM is composed of three states:

- Initial: all control signals are initialized, then the FSM moves on to the WAIT state.
- Wait: the inputs are saved in order to be delayed, and the results of the multiplications are displayed. Two flags are set, in_inter.ready = 0 to stop the generation of new inputs, and out_inter.valid = 1 to signal the presence of a new valid output to confront with the reference one.
- Send: when the results have been compared, in_inter.ready is set to 1 to generate a new input and out_inter.valid = 0.

4.2 Input Generation

Since it is not possible to generate random single-precision floating point numbers, random 32-bits sequences are generated. Using the functions $shortrealtobits() and $bitstoshortreal() it is possible to convert from floating point numbers to their binary representation and viceversa. To make the results more readable, some constraints have been applied to the input generation: in particular, the bits from 30 to 23, which represent the exponent of the floating point number, are forced to be equal to 128 for both generated inputs.

4.3 Testbench Run

During the testbench run no mismatches occurred, as shown in the transcript file.

```
# FPMUL: input A = 3.210777, input B = -2.400374
# FPMUL: input A = 01000000010011010111110101011111, input B = 11000000000110011001111110111001
# FPMUL: output OUT = -7.707065
# FPMUL: output OUT = 11000000111101101010000001000110
# refmod: input A = 3.210777, input B = -2.400374, output OUT = -7.707065
# refmod: input A = 01000000010011010111110101011111, input B = 01000000010011010111110101011111,
output OUT = 11000000111101101010000001000110
# UVM_INFO @ 3015: uvm_test_top.env_h.comp [Comparator Match]
# FPMUL: input A = 3.084183, input B = -3.686763
# FPMUL: input A = 01000000010001010110001100111111, input B = 11000000011010111111001111101110
# FPMUL: output OUT = -11.370651
# FPMUL: output OUT = 11000001001101011110111000110000
# refmod: input A = 3.084183, input B = -3.686763, output OUT = -11.370651
# refmod: input A = 01000000010001010110001100111111, input B = 01000000010001010110001100111111,
output OUT = 11000001001101011110111000110000
# UVM_INFO @ 3045: uvm_test_top.env_h.comp [Comparator Match]
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 3045: reporter [TEST_DONE]
'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../tb/env.sv(42) @ 3045: uvm_test_top.env_h [env] Reporting matched 101
#
# — UVM Report Summary —
#
# ** Report counts by severity
# UVM_INFO : 106
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [Comparator Match] 101
# [Questa UVM] 2
# [RNTST] 1
# [TEST_DONE] 1
# [env] 1
```

CHAPTER 5

Testbench

5.1 Memory component

For the testbench two more components have been written to mimic RISCV memories. Both of them work on the negative edge of the clock and only the needed cells have been allocated in order to speed up the simulation.

For data memory the data_memory entity has been used. It is a memory with one single address used for writing and reading, one input port for data to be written and one output port for data to be read. Two different signal activate the writing or the reading on the memory.

For instruction memory the instruction_memory entity has been used. It has one input port for address and one output port for read instruction.

5.2 ASM code

For testbench an assembly code has been written that finds the minimum absolute value in a set of data, with cardinality six. Through a simulator it has been tested and the binary instructions code has been stored in instruction memory while the data in data memory, the two previous entities, both in the right address.

The following set of instructions is used to calculate the absolute value.

```
lw x8,0(x4)
srai x9,x8,31
xor x10,x8,x9
andi x9,x9,0x1
add x10,x10,x9
```

If the simulation discovers a right behavior at the end, while the last instruction performed an infinite loop on itself, it will be possible to see in the memory, at position seven, the value 3 that corresponds to the minimum absolute value.

5.3 ABS function

The abs operation that has been created it's not a real instruction, it can't be compiled and doesn't exist an absolute instruction for integer registers. In the previous version of asm code the absolute value was calculated through different operations while now they are now substituted in the following way.

```
lw  x8,0(x4)
abs  x10,x8,0
```

For this reason, after removing the four instruction that calculated the absolute value of a register, during ASM simulation a NOP instruction has been added to calculate the offset values used from the jumps. While, in the instruction memory, the NOP instruction has been substituted from the abs instruction, codified manually, bit by bit, to achieve the right behavior. At the end the result must be the same as in the previous version.

5.4 Conclusion

The first simulation lasts 101 clock cycles the second one lasts only 83.

CHAPTER 6

Synthesis and Place & Route

6.1 Shynthesis

The two different designs have been synthesized using Synopsys Design Compiler, in order to obtain the maximum working frequency and an area estimation of the circuits. The following table summarizes the obtained results.

	RISCV	**Modified RISCV**
Min. Period	3.55 ns	3.6 ns
Max. Frequency	281.7 MHz	277.8 MHz
Area	14006.23 μm^2	14802.1 μm^2

Table 6.1: **Design Compiler Reports**

As expected, the modified design with the added component occupies more area. Moreover, since the added hardware is placed along the critical path, the maximum frequency is also slightly lower; however this is compensated by the reduced time needed for the computation of the instruction.

The report of the *elaborate* command lists the inferred memory devices inserted in the netlist. Every element was checked to avoid the insertion of latch instead of flip-flops. Finally, both synthesized netlists were simulated with Modelsim to confirm they still behaved correctly.

6.2 Place & Route

After the synthesis of the circuits, place & route was performed using Cadence Innovus. This requires to perform the following steps:

- Importing the design;
- Floorplanning;
- Power planning and routing;
- Cell placing;
- Signal routing;
- Timing and design analysis.

The procedure returned no violations in connections and geometry. The resulting netlist was saved together with the final gate count.

	RISCV	**Modified RISCV**
Area	13533.0 μm^2	14133.6 μm^2

Table 6.2: **Innovus Reports**

Innovus was able to optimize the netlists and save a noticeable amount of area on both designs, with respect to the value obtained with Synopsys. Also in this case, the produced netlists were simulated to verify the compliance with the original design.

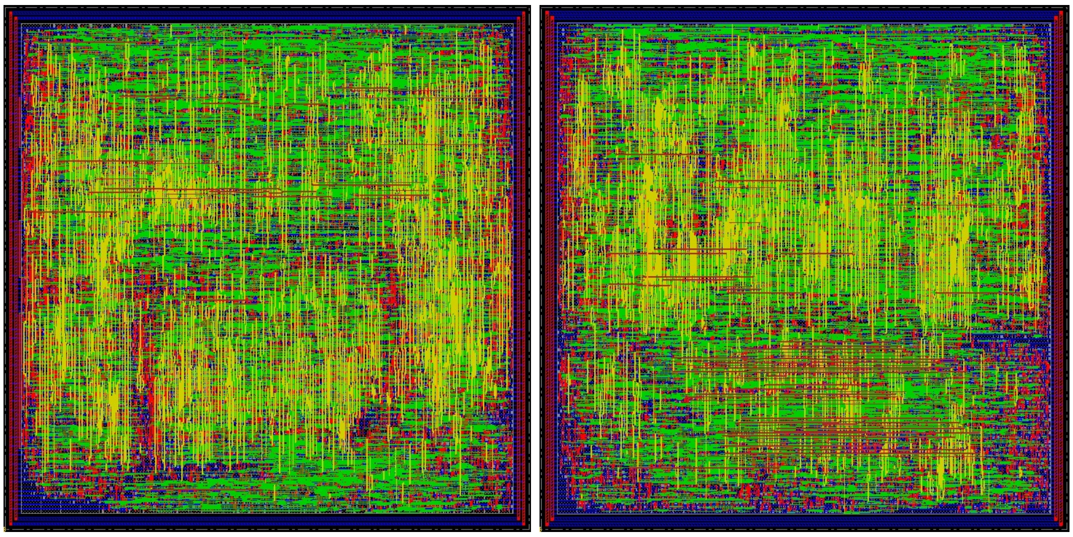
(a) RISCV (b) Modified RISCV

Figure 6.1: **Place and route of the two designs**