Project 6 SoI-CMOS: Project Report

Gengwu Du, Ziyan He, Annan Liu,
Yann Pirot, Gautham Prabhakar, Lars Wennersten
gengwu || ziyanh || annanliu || yjmpirot || gauthamp || larswen @kth.se

March 14, 2022

Abstract - In the context of sending probes to remote planets where operating conditions differ form Earth is being developed a process design kit able to withstand very high temperatures. This new process, however, presents large transistors channel length and therefore requires testing, as to verify its realistic use for common digital circuitry. To showcase this new process, a CPU and an FPGA are designed as to figure out the feasibility or not of functioning embedded in 7×7mm^2 chips, as space operations imply severe constraints such as area. This project introduce the design from scratch of both very simple CPU and FPGA for demonstration and documentation purposes.

Acronyms

CB Connection Block. 9

CISC Complex Instruction Set Computer. 40

CLB Combinational Logic Block. 3, 9, 11–20, 23, 27–29, 31, 35–38, 72, 73

CMOS Complementary metal–oxide–semiconductor. 6

DFF D Flip-Flop. 6, 14, 15, 18, 19, 37, 38, 73

FIFO First In First Out. 18–23, 26, 28, 30, 33, 36, 37

FPGA Field-Programmable Gate Array. 6–9, 11–18, 20, 33–38, 71–73

GPIO General-purpose input/output. 3–5, 42, 48, 49, 52, 53, 57, 58, 63, 64, 71, 72

HDL Hardware Description Language. 34

I/O Input/Output. 41, 42, 47, 49, 52, 53

IOB Input/Output Block. 9, 15–18, 20–22, 26, 28, 31, 36, 37

ISA Instruction-set Architecture. 3, 5, 6, 39–41, 51, 52, 59, 60, 64, 65, 71, 72

LUT Lookup Table. 13, 14, 28, 34

MSB Most Significant Bit. 48, 52

MUX Multiplexer. 13, 14

MUXES Multiplexer. 14, 15, 18, 34, 37

PCB Printed Circuit Board. 6

PDK Process Design Kit. 7, 8, 12, 14, 15, 18, 34, 37, 38, 72, 73

RAM Random Access Memory. 12, 38

RISC Reduced Instruction Set Computer. 39, 40, 71, 72

RISC-V Reduced Instruction Set Computer Five. 3–7, 9, 39–44, 46–49, 51, 56–58, 63–65, 67, 71–73

SoC System on Chip. 3, 4, 41, 42, 47, 49, 64, 67, 71, 72

SoI Silicon on Insulator. 6

SWB Switching Block. 3, 9, 14, 15, 18–20, 23, 25–27, 30, 31, 37, 38, 73

XML Extensible Markup Language. 34, 35

Contents

List of Figures

List of Tables

1 Introduction

The main driving force of this project revolves around NASA's mission to send a rover to Venus in the 2030's. However, the high-temperature on the Venus surface makes it almost impossible to use standard Complementary metal–oxide–semiconductor (CMOS) circuits and Printed Circuit Boards (PCBs) used on Earth today. Further, missions to Jupiter has to fight against its highly radioactive magnetosphere when sending probes to investigate the inner moons Io and Europa. Thus, building high-temperature robust electronic circuits that also can withstand highly radioactive environments is the main objective of the project.

One of the interesting techniques to achieve it is to build circuits from Silicon on Insulator (SoI) CMOS . Because of the insulator used as a substrate, it has a higher band gap and can withstand temperatures up to 500 degrees Celsius, which is just enough to be able to work on Venus. As an additional bonus, we can manufacture digital circuits in this technology with up to 98% yield in the KTH lab. The downside is that it is using a line width of $1um$ (~2500 NAND-equivalents per mm^2, propagation delay ~300 $ps/stage$). Due to manufacturing cost of the masks for larger circuits, we are limited to $7 * 7mm$ circuits (120 k Gates).

Further, Field-Programmable Gate Arrays (FPGAs) are more and more used in Space applications due to its ability of run-time reconfiguration. The ultimate goal of this project (in ~10 years time) would be to build and manufacture an FPGA that could withstand the temperature on Venus and function reliably in the radiation belts of Jupiter.

This will be starting phase for the SoI CMOS project and this project will lay the foundation for the future development of the SoI CMOS. The project SoI CMOS aims at developing:

1. Build a synthesis library for Cadence for the target technology (300ps library).The current library only has the two basic gates needed to function, one 2-input NAND-gate and a D Flip-Flop (DFF).
    - Verify the library can synthesize with the NAND and the DFF using a basic circuit design.
2. Design a basic RISC-V processor capable of executing the RV32IM(v2.0) ISA.
    - Test the system in System Verilog to verify that it works according to specification.
    - Implement the system as an FPGA prototype.
    - Synthesize the RISC-V using the library designed to get the area estimates of the design.
    - Synthesize the designed RISC-V on a development board.
3. Study how homogeneous run-time re-configurable FPGAs are built.
    - Implement a homogeneous FPGA using VHDL.
    - Verify the FPGA using System Verilog.
    - Synthesize it using the created synthesis library.

- Synthesize the designed FPGA on a development board.

Since the two tasks of designing an FPGA and CPU are sharply distinguishable, their presentations are separated respectively into section 2 and 3, like two stand-alone papers. Section 2 describes what a FPGA is and the constraint it has upon itself, following the design for the FPGA with its verification and construction of a demonstration program. A notice for an open sourced program called OpenFPGA with its potential uses for similar projects are included. The paper ends with describing the obstacles, conclusion and possible future work. Section 3 begins as section two by explaining the concept of a RISC-V CPU and of it's structure. verification, functional simulation and synthesis are later described and the paper are to be ended with conclusion and future work.

Both sections introduce the different workflow for each device, with different teams, coding language (VHDL for FPGA and Sytem Verilog for RISC-V) and constraints and targets. Eventually, the only things the two project share are the Process Design Kit (PDK) and area limitation. Section 4 then stands as a global conclusion of these two separated sub-projects. Note the lack of a distinct library creation section. Indeed, this task had been prematurely dropped to due its severe constraints and lower necessity, as further explained notably in sections 2.4 and 2.9.

2 FPGA

2.1 Introduction

The realisation of a new, high-temperature resilient PDK is the lowest-level stage of digital design. The property of the PDK gives constraints, notably of timings and area, that will apply to the design using it. In order to demonstrate the performances of this new PDK , a basic FPGA design is proposed. The purpose of this FPGA architecture is to demonstrate the feasibility or not of a common digital circuit, therefore, it does not intend to achieve any kind of minimum performances, such as computing power or speed, nor it intends to propose enhancements or showcase state of the art features.

This works goal is to propose a basic FPGA design that can show correct functioning, and fits into a 7×7mm^2 chip. It intends to introduce the development workflow, from scratch and with little or no previous knowledge of FPGA design, and the difficulties and obstacles met when designing an FPGA .

Section 2.2 briefly reminds the functioning of an FPGA and the mandatory blocks to describe, which architectures are mainly inspired from section 2.3 theoretical framework. The exact specification required are registered into section 2.4 with a first explanation to the failure of writing this PDK specific cell library. The main section, 2.5, lists one by one every block to design, their functioning and design strategy, since the exercise of designing an FPGA consists of describing very application-specific, independent blocks, each block is detailed in its own subsection, before being all tiled-up together into the 2.5.4 top-level section. Every of these block are then verified into section 2.6.1. To run programs on the hardware, a specific bitstream setup or a compiler is required, respectively described in section 2.7 and 2.8. Since this work intends to document on the difficulties and challenges met when it comes to describe an FPGA from scratch, section 2.9 lists every issues detected and how they were solved. This FPGA sub-project has its own sub-conclusion and future work in sections 2.10 and 2.11.

2.2 Introduction to FPGA

The FPGA , unlike the RISC-V CPU, is not application specific. The main strength of the FPGA is to run many operation in parallel, each of these operation being ”on-field” re-configurable. The design of an FPGA is therefore very different from the one of the CPU. Instead of designing a complex computing pipeline, with many different blocks achieving specific tasks required to run an instruction, completing an FPGA basically requires only one simple block (the CLB), which is then duplicated to fill the allowed area. In practice, a few other blocks, extremely basic as well, are required for CLBs to communicate.

The usual FPGA architectures consists of multiple, generally identical, CLBs . A CLB is the only computing block, able to achieve any basic combinational logic operation provided a correct configuration bitstream is first set. Every CLB is connected to every other thanks to a wire net. The net interconnections allow CLBs to operate together for higher complexity operations.

The net is made of bidirectional wires, called a Connection Block (CB) , which drive the CLBs inputs and outputs to a four-directional block, referred as a SWB . The wire net then allow CLBs to communicate to one-another among the whole design, to the cost of speed and timing management with the distance. Every computing block is surrounded by four CBs on its edges, which connect to four SWBs on its corners. This nine block square is often referred as a CLB island in *Xilinx* documentation.

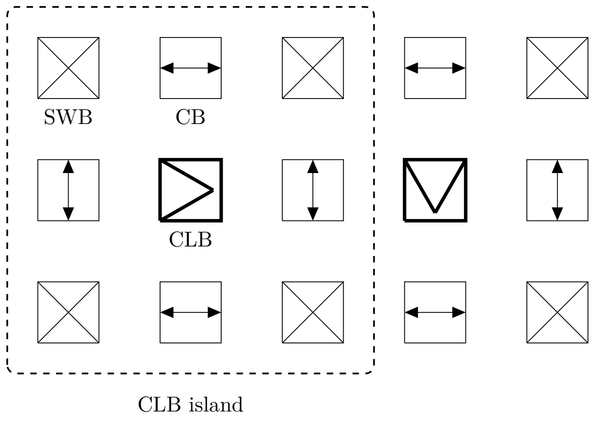


Figure 1: CLB island schematic

The peripheral routing blocks also connect to Input/Output Blocks (IOBs) , which, once configured, set the chip pads either as inputs or outputs.

The FPGA architecture is then a tiling of CLBs , each achieving its given operation. To know which operation a CLB should achieve, the user must first send to the chip a configuration bitstream, that is dependent on the software currently running. The bitstream notably contains every potential result of these operations, from which the CLBs actually only select the correct one depending of the program variables. These variables are also sent to the correct block following the bistream instruction, drawing the interconnection map between the CLBs .

Designing this FPGA is then not that much about designing the basic blocks, which in this architecture shall be designed as basic as possible as stated in section 2.4, but about defining the standards between them, and correctly connecting them altogether.

2.3 Theoretical framework

The computing power of an FPGA is somewhat immediately linear with its number of computing block, and the computing capacity of each. Because chip size is the only design constraint given, and because of the large transistors gate size, it has been demonstrated the final design could not achieve great performance anyway.

With this statement adds up the overall lack of experience of the working team in digital design. Given the time schedule, and beginning from scratch, the final FPGA target shall be a low technology device. Because some members of the team were more comfortable with the *Xilinx* working environment, it has been decided to partly reproduce the old work of Spartan I and II architectures which some datasheets and hardware descriptions are freely available. These technologies propose the advantage of having about the same amount of CLBs expected, presenting few additional features for easier learning. Overall, as the project does not intend to achieve any modern FPGA features, not much more than the default FPGA understanding is required.

2.4 Design constraints

Because of the noise constraint of operations at high temperature, the PDK under development uses a very large transistor technology of 1μm long gate channel. Combined with the necessity of area-limited chips for usage in space, these opposed constraints lead to very few available cell count to achieve a valid FPGA design, estimated around 125 thousands area unit, one area unit being the size of a NAND gate, for a $7{\times}7$mm^2 chip.

However, FPGAs architectures require a lot of wiring area as to allow CLBs to be connected to each other regardless of the layout complexity required by the programmed software. Literature gives a common value of 80 to 90% of the surface area exclusively being used to interconnections between the computing blocks. Therefore, an optimistic prediction is only 25 thousands units area being available for computing blocks.

Because of the complexity required to the cell library, it has not been possible to elaborate the FPGA design with the only information provided by the under-development NAND gate. Every area estimate in the FPGA section is then achieved by using the LSI10K library, also providing 1μm long transistors, as a coarse estimate.

A first attempt to design a modern FPGA offering many features led to excessively area consuming CLBs . Current FPGAs can achieve a few hundreds of thousands of CLBs , which is immediately excluded from the available area.

Overall, as the PDK was not yet available at the time the FPGA has been designed, the main constraint ruling the designs decision made is the area consumption. Every block shall be designed as to provide minimum features and minimum circuit complexity as to demonstrate or not the feasibility of a basic FPGA with the provided PDK and chip surface. Furthermore, although the FPGA would have better been homogeneous for space operation (and risk of cell irradiation), the limited area available deters from adding the Random Access Memory (RAM) inside the CLBs but instead moving it to an external block.

2.5 Basic blocks and Top level

2.5.1 Configurable Logic Block (CLB)

The CLB is the only computing block of the FPGA . It mainly consists of a Lookup Table (LUT) , in practice, a multiplexer which inputs wires are set by an initial bitstream setup. During software operating phase, the selection input is assigned to one of the code variable, as to select one of the bit from the configuration bitstream as output. Therefore, with the correct initial bitstream, a LUT is able to achieve potentially any basic logic function, and is re-configurable between any new software implementation.

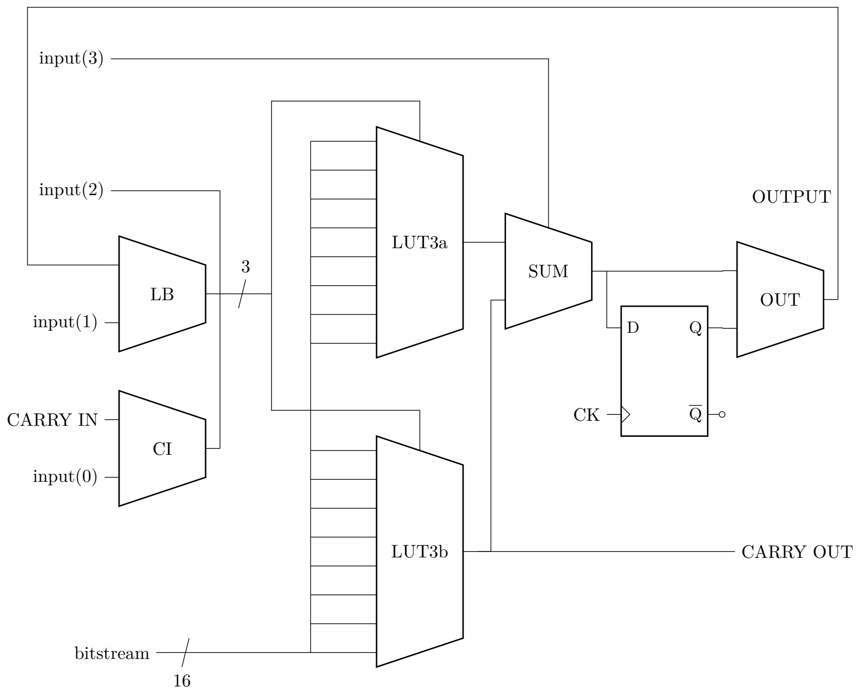


Figure 2: CLB schematic

An important limit to the LUTs capability is the width of the multiplexer. A LUT of width 16 is only capable of 4 variable bits on its selection input. Adding a single variable as input directly multiply the chip size required by a factor of 2. However, this brute upgrade is area-consuming, and leads to excessive amount of idle transistors in case of operations not needing as many variables. Modern FPGAs architecture are able to combine together smaller LUTs as to achieve wider operations, to the cost of additional logic elements and a longer configuration bitstream. Therefore, this FPGA design cannot afford such enhancements.

Nevertheless, a minimum, and almost mandatory, upgrade to the basic Multiplexer (MUX) based LUT is to separate it at the very least in two, for the very common FULL-ADDER operation, requiring two outputs, SUM and CARRY, based on the same input variables.

Therefore, based on the first Xilinx Spartan technologies, the LUT a limited

4 variables one, splitted in 2 LUT of size 3. Note a 4-LUT is the bare minimum: 2 variables for A and B, one for the input carry, and one to split the result (sum and output carry).

To complete a minimum CLB , the result of the LUT is sent either to a DFF , to achieve synchronous operations and store the result, or immediately sent at the output. Because on the final area estimate a few area was still free, but not enough to add more CLBs , a loop-back wire has been added to fasten the looping of the output back to one of the input variables, which barely costs the size of a 2-to-1 MUX and a memory DFF .

The output DFF is slightly elaborated as to allow a set/reset, read from the connections wires as two additional variables. Therefore, 6 additional 8-wide multiplexers are required to read the total of 6 inputs from the 8 connections wires. These are not displayed on figure 2.

Therefore, apart from the one 2-1 loop back MUX , the CLB area could not have been designed smaller without stripping fundamentals features.

Design elaboration gives a first area estimate by 589 unit size, leading to an expectation of 42 CLBs , which immediately suggests the PDK is way too area consuming to achieve even very low performances on a 7x7mm^2. As a comparison, the smallest Spartan I design from Xilinx would achieve 100 8-variables CLBs in the early 2000's. This design is then very coarsely estimated to be somewhat 4 times less powerful than a 20 years old technology, leading to the assertion the PDK shows unsatisfying for FPGA design in highly area constrained situation.

To drive this low-feature CLB , 42 configuration bits are required, each needing a DFF memory, of 8-units size, meaning more than half the CLB size is exclusively consumed by configuration memory.

2.5.2 Switching Block (SWB)

The switching block is responsible for driving the signals between every CLBs . A CLB is connected to 8 wires both on the input, and the opposite output side, so that the 6 inputs (4 variables, set and reset) can be selected with some liberty among the 8 wires, the same for the single output. The more the wires between the blocks the easier the layout is to connect the different CLBs together according to the software requirement. However, the more area it also requires. Because Spartan I uses 9 wires for 8-variables CLBs , decision has been made to have 8 wires in the current design, as to match a 2^3 inputs Multiplexer (MUXES) .

The minimum would have been only 6 wires, which would have very likely led to impossible layout, therefore two additional wires are added, but of a different type as the other first 6 as to minimise the area.

From Spartan I architecture, the first 6 wires only connects 2 adjacent SWBs together, and are then called "short wires". The 2 others wires only connect 2 peripheral SWBs together, and are then called "long wires".

Long wires have the advantage of not requiring a net connection a every inner SWB , which allow faster signal propagation though the device and so, to connect distant CLBs with less timing constraints. Shorts wires stop at every inner SWB , into a net. A net is made of 4 multiplexers heading into each direction, as to redirect the signal. Although this solution lead to maximum security and signal preservation, it requires a lot of gates to operate, as every

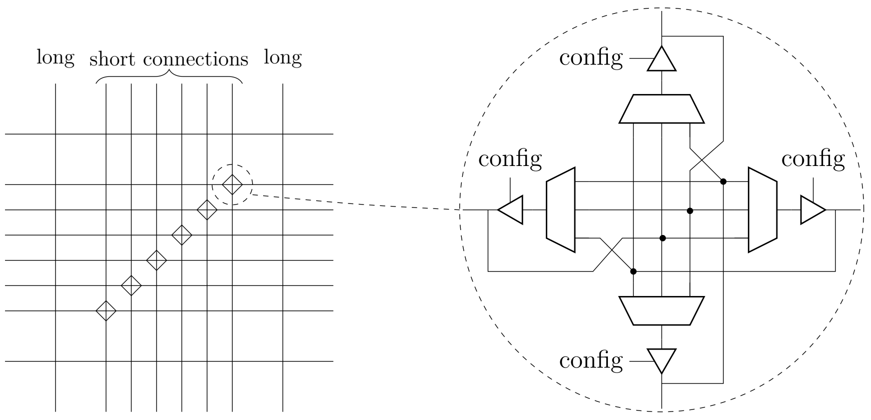


Figure 3: SWB schematic

multiplexer, having 3 inputs, needs 2 configuration DFFs , and an additional one for the output tri-state buffer, preventing writing when disabled. 4 directions, 6 nets then require 72 8-units size DFFs exclusively for the configuration, which is the same size as the entire computing block, hence the 80% area used into routing blocks.

This SWB architecture has been chosen for its simplicity, but other designs are available. To minimise routing complexity, some other designs do not connect a wire with the one immediately facing it and perform a shift, or a carefully chosen permutation as to allow a signal to change both direction and wire position among the net. Because these are advanced considerations, the chosen matrix keeps to the simplest permutation, direct connections as no more wires could be spared anyway.

The very area consuming multiplexers could be replaced by much smaller pass-gates, only consisting of a single transistor, which have the digital signal flowing from drain to source. Unfortunately, this option is rejected because of the very long gate the available transistors have. The more the signal flows through channels, without being regenerated, the more analog effects (timings and voltages drops) might degrade the signal readability. Because Spartan I already made use of such pass gates, it might be possible, once the PDK terminated, to verify if pass gates are available or not. To emphasis such hope, knowing the current FPGA could only afford very few CLBs islands, it is very unlikely any software routing would require a signal to go through more than just a few SWBs . The current SWB size estimate stands at 865 unit sizes while a pass gate based SWB is expected to only require 300.

The connection block displayed on figure 1 barely hides the 6 shorts wires and the 2 longs. Where a connection block sides a CLB input, 6 8-wide MUXES choose which wire goes to which input, the same with a 8-wide demultiplexer on the output side.

2.5.3 Input/Output Block

The IOB is in charge of either driving the output pin to write on one of the 8 adjacent connection wire, or to be written by one of them. The reading and writing could be both synchronous or not.

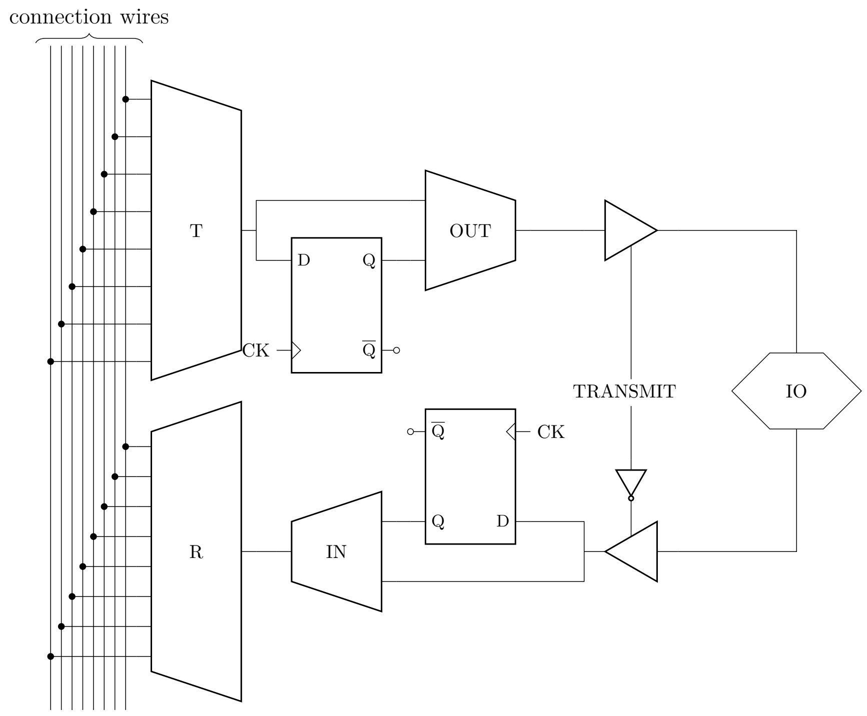


Figure 4: IOB schematic

The architecture is once again chosen minimum, perhaps excessively, missing some very basic features, once again to save area. For a pin to be possibly used as an input and an output, two different path connect the IO pin to the 8 connections wires, receive (R) and transmit (T). Two simple tri-state buffers prevent writing errors, when one path is open, the other is necessarily closed.

This IOB accomplish no additional features apart from the mandatory synchronous or not choice. Note that once configured, the IOB is then locked until next software configuration to either input or output, which is not the general minimum specification from an FPGA .

Making the pin to be driven as a full input/output, not just one of the two, would simply require the TRANSMIT signal to be read from the connection wires, just like the T path. However, its initial value would then have been a variable from the connections wires, not a configuration bit, leading to unknown initial state. To prevent this, additional blocks would have been further required to separate an initial state read form the bitstream with the later read variables, demanding even more area than just a third path. For this reason, this solution is not implemented as to put every effort into allowing more CLBs , as to demonstrate their very low number even with many classic FPGA features stripped.

The IOB area estimate using the LSI10K library gives a 171 units-size gates area, standing a few times lower than the other blocks.

2.5.4 Top level

Once every block has been designed, it is then necessary to assemble them all together as to form the FPGA architecture. Unlike most digital project, the FPGA top level looks more like a geometrical tiling game than simple direct wiring between serial blocks.

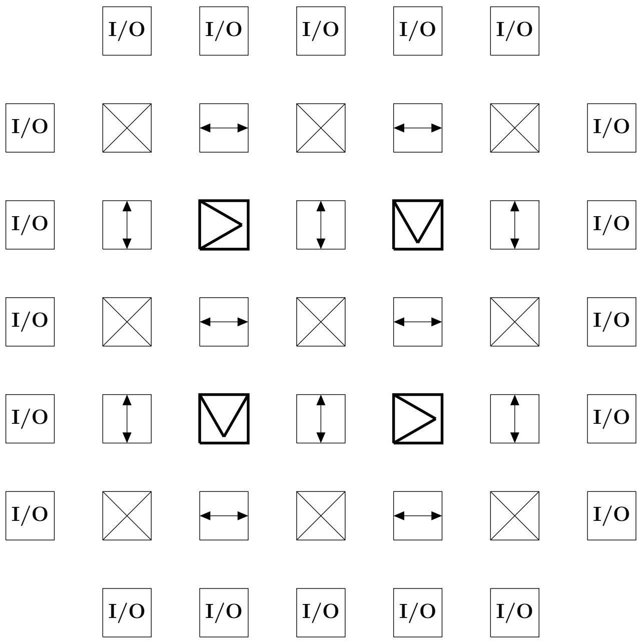


Figure 5: 2×2 FPGA tiling

Because is not yet know at the beginning of the code writing how many CLBs the design can afford, the width and height of the final architecture (the product being the number of CLBs) is written as generic. Initially, the number of connections wires was first meant to be generic as well, as their required number is the result of a complex mathematical problem out of reach from this project. However, due to several difficulties and code rewriting, this option has been dropped as to finish a working design in due time. Few corrections are required to bring back this feature in case of 6+2 connections wires would show unnecessary high or low (though it is a convenient sweet spot for power of two multiplexer size).

The top level architectures also has its few design decisions to be made. Although the tiling is constrained by the "island" architecture adopted (figure 1), it remains to be decided the number of IOBs , their position, the carry chain path, and the configuration bitstream path.

To make the FPGA homogeneous, it is required no area nor direction present a different configuration than the others, apart from the I/Os necessarily on sides. Because isotropy is not such a hard requirement as homogeneity, CLBs could all face the same direction. However, to make the software routing easier, without adding too much on the code complexity, half the CLBs have their inputs and outputs vertical, "looking downward", the other half, horizontal "looking rightward". This also proposes the advantages of favouring I/Os being used as inputs on the top-left side and outputs on the bottom-right for slightly easier software routing and pin management.

FPGA usually require many I/Os because of the parallel operations it can achieve. However, because of the expected very low amount of CLBs available this number may not have to stand much higher than a few scores. Furthermore, the PDK also enforces a maximum distance between the different pins, allowing slightly more than a hundred of them on each side of the $7{\times}7\text{mm}^2$ chip. With an expected amount by 40 CLBs expected, this maximum is certain never to be reached. Looking Spartan I specifications, the 100 CLBs design would make use of 77 I/Os left to the user. This design will have an IOBs placed on the side of each peripheral block (both connection wires and SWBs), thus, if the number of CLBs is $i \times j$ the number of IOBs is $2(2i+1)+2(2j+1)$. With the expected 7×6 design this would give 56 I/O pins, although this would sound too much, the final designs reveals to be 8×8 for 68 I/Os, which is reasonable when compared to Spartan I.

The top level describes the wiring between the blocks. Adding to the 6 short wires that connect every two adjacent SWB and the 2 long wires that connect opposite IOBs that side a SWB , every block should connect to the configuration First In First Out (FIFO) . The second path required is the carry chain between every CLB , to accelerate the carry propagation not going through the 8 connections wires.

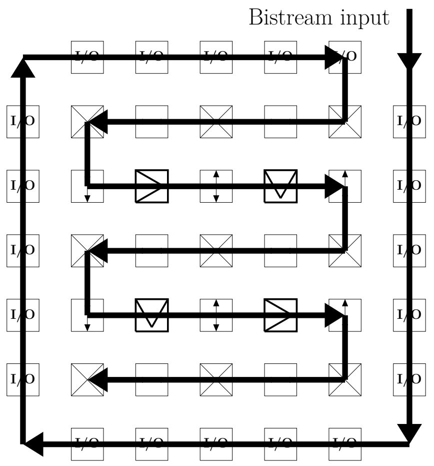
The configuration path is not critical, as it only matters for the setup phase, generally not having any timing constraint. The chosen path is therefore pretty much irrelevant, apart from routing feasibility, and has only be chosen to simplify the VHDL code, IOBs served first, then the core blocks.

The carry chain path is a more critical choice. The Spartan I architecture describes a carry chain path allowing some parallelism, with the last carry result of a line delivering every CLB of the next one. Although implementing it barely cost few 2-1 MUXES , it has not been implemented. Because the design is expected to achieve so few CLBs , implementing parallelism is not really pertinent. Instead the carry chain is a single path that serpentines through the lines, with the input on top left, and output either on bottom left or right depending on the height parity.

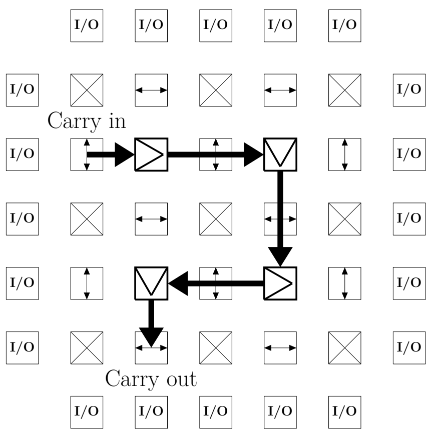
Once the full code is written, and carefully debugged, serial elaborations are run with increasing width and height until the 125.000 size is reached, giving a maximum FPGA size of 8×8 for 121.000 unit gates count, using LSI10K. Although this achieves "much" more CLBs than first expected, it absolutely does not stand any comparison with modern designs. Tiling several $7{\times}7\text{mm}^2$ chips together would not decrease the gap much, as the ratio order of magnitude stands at thousands.

Basic block	area	%
CLBs	38.5k	32%
IOBs	12.5k	10%
SWBs	70k	58%
among all		
DFF FIFO	74k	61%

Table 1: area share of each basic block, area share of the sole configuration bitstream memory



(a) Configuration bitstream path



(b) Carry chain path

Figure 6: Top level specific signal path

The 121.000 units size area is roughly split as displayed on table 1. From the table, it is visible that the area share of the CLBs is much more than the 20% expected, due to the very minimum featured other blocks. This ratio could be even further improved by replacing the multiplexers based nets among the SWBs with pass gates ones. The area gain is expected than 30k unit size.

What is probably even more noticeable is the huge area occupied by the FIFO memory. This area is only used by the about 9.200 DFFs long FIFO that stores the configuration bistream. Because this memory has no need for speed, not even to be structured necessarily as a FIFO , more area saving memory cells should be considered.

2.6 Verification for FPGA

2.6.1 Blocks verification

This section describes the verification strategy planned for the FPGA . The functional verification for the FPGA is done both at block level and system-level. Firstly, each individual module is tested independently to make sure of the correctness of its function. The verification strategy is dependent of each module. When each module is right, the top level module is then verified checking its simulation waveform.

- At block level: Functional units such as CLB , IOB , SWB unit are verified as a stand-alone blocks according to the verification specification as to ensure the RTL adheres the functional specification. Because of the limited time verify technical lack of competence and design imperfections, some units are not tested using randomization but instead by using the simulation waveform with some specific bit stream set to verify the correctness of the units.
- At system-level : because it is unfeasible to load the desired configuration bistream into the FPGA through the FIFO , a nosetup version for the toplevel has been designed, which bypasses the FIFO module and directly loads the bit stream of the corresponding module as a parallel input directly into the targeted block. This problem does not occur when importing the program to the fpga board.

2.6.2 FIFO Verification specification

The FIFO is a very basic but important block in the FPGA design, the long bitstream can only be loaded by the FIFO block. Whether it is CLB , SWB or IOB , they all need the FIFO module to pass the bit stream. For the verification about this part, it only needs to make sure whether the input data(i_configdata) is saved in the lowest bit and whether the most significant data has been output.
FIFO Functional specification.

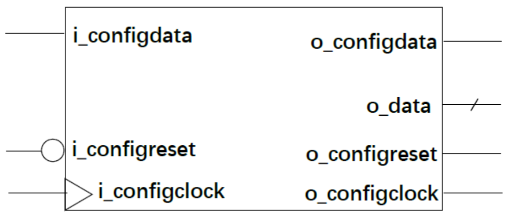


Figure 7: FIFO

- Clocked on positive edge of clock
- Has an active low reset
- All loads to the FIFO is sequential

FIFO Verification Strategy.

- Assert i_configreset for 10 ns(active low) to reset the DUT
- Generate i_configdata using randomization
- Specify property for assertions and for coverage
  property FIFO_low ;
  @(negedge clk)disable iff(!i_configreset)
  u1.r_outdata(0)== i_configdata;
  endproperty
  property FIFO_high ;
  @(negedge clk)disable iff(!i_configreset)
  o_configdata== u1.r_outdata(g_width-1);
  endproperty
- Check assertion pass count and cover property count (Assertion pass count should be 100 percent)

2.6.3 IOB Verification specification

The IOB block is in charge of either driving the output pin to write on one of the 8 adjacent connection wire, or to be written by one of them. For the two modes, the verification is done separately and the information about the ports is shown as follow. According to these figure, the io_long and io_short

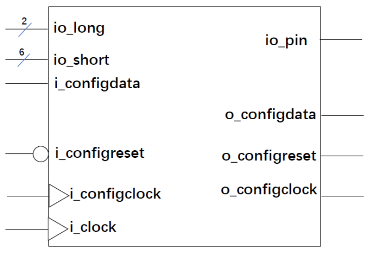


Figure 8: IOB_transmit

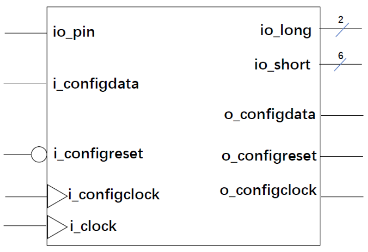


Figure 9: IOB_receive

are the 8 adjacent connection wires which will represent input and output separately when the IOB module is in transmit mode and receive mode. i_clock is a built-in clock which is used to determine whether the output is sequential or combinational. Because the FIFO block is a sub-module for the IOB module, not much validation is put to o_configdata, o_configreset and o_configclock as they have been verified in FIFO verification. As shown on figure 4, the IOB includes transmit interface multiplexer, receive interface de-multiplexer, transmit combinational or sequential multiplexer and receive combinational or sequential multiplexer. All of them need control bits which are loaded by the FIFO module. Therefore, more attention is required when checking the meaning of specific i_configdata and verify that i_configdata implements its corresponding functions. For the i_configdata, the meaning about each bit is shown as follow.

- config(2 downto 0) transmit MUX interface selection
- config(5 downto 3) receive MUX interface selection

- config(6) selects the transmit (a)synch MUX
- config(7) selects the receive (a)synch MUX
- config(8) select input or output pin mode 1 = transmit, 0 = receive

IOB Functional specification.

- Clocked on positive edge of i_configclock for the i_configdata
- Has an active low reset
- Clocked on positive edge of i_clock when the transmit is in sequential mode
- Clocked on positive edge of i_clock when the receive is in sequential mode
- transmit MUX interface selects which of the 8 adjacent connection wires is used as the output of the I/O port
- receive MUX interface selects which of the 8 adjacent connection wires get the data from the I/O port

IOB Verification Strategy

- Because the configdata is loaded by the FIFO module. It means that it needs to wait a few cycles before the configdata is fully loaded. Hence it is difficult to realise randomisation of the configdata and so, some different configdata are preset, and the simulation waveform observed to make sure the correctness of IOB modules. The process about the verification is divided to two parts(transmit and receive). The specific steps are as follows,
- For the process about transmitting
  - Set i_configreset for 5 ns(active low) to reset the configdata
  - Give specific i_configdata
  - Set the io_pin
  - Check the simulation waveform and make sure the output of the io_pin is right
- For the process about receiving
  - Set i_configreset for 5 ns(active low) to reset the configdata
  - Give specific i_configdata
  - Set the io_long and io_short
  - Check the simulation waveform and make sure the output of the io_long or io_short is right

IOB simulation result

The simulation result about IOB module is shown as follow. It has been divided to two parts that: transmit and receive. Because of the FIFO block, the i_configdata is loaded when clocked on positive edge of i_configclock. To make the simulation result obvious, the i_configclock is set to 0 when a whole i_configdata is fully loaded. And for the transmitting process, the i_configdata is taken 100100011 as the control bit stream. It means that: (1) it is in transmit mode, the data is transmitted to io_pin from the 8 adjacent connection wires, (2) the io_pin is clocked on positive edge of i_clock and (3) the data is from the third wire which is the io_short[2]. Only the io_short[2] is set as one and the others are set as 0 to make the result clear. According to the figure 10, it is observable that when the i_configdata is loaded fully, the io_pin goes from a high-impedance state to a 0 and to a 1 when the i_clock is on the rising edge, which achieves the behaviour wanted. Other i_configdata are tested as well, with the most representative of them are shown here. And for the process of

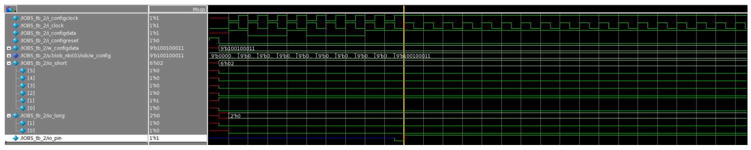

Msgs
/IOBS_tb_2/i_configclock 1'h1
/IOBS_tb_2/i_clock 1'h1
/IOBS_tb_2/i_configdata 1'h1
/IOBS_tb_2/i_configreset 1'h0
/IOBS_tb_2/w_configdata 9'b100100011
/IOBS_tb_2/u3/iob_nbi(0)/iob/w_config 9'b100100011
/IOBS_tb_2/io_short 6'h02
[5] 1'h0
[4] 1'h0
[3] 1'h0
[2] 1'h0
[1] 1'h1
[0] 1'h0
/IOBS_tb_2/io_long 2'h0
[1] 1'h0
[0] 1'h0
/IOBS_tb_2/io_pin 1'h1

Figure 10: IOB_transmit_simulation

receiving the data, the control bit stream is set as 001100011. It means that: (1) it is in the receive mode, the data is from io_pin to the the 8 adjacent connection wires, (2) the 8 adjacent connection wires is clocked on positive edge of i_clock and (3) the data is to the fourth wire which is the io_short[1]. The io_pin is set as 0 at the beginning and when the i_configdata is loaded fully, io_pin is set as 1 to check if the output is clocked on positive edge of the i_clock. According to the figure 11, one can find that it achieves the effect targeted. The data from the io_pin is output to the io_short[1].And other i_configdata is tested as well, and the most representative of them are shown here.

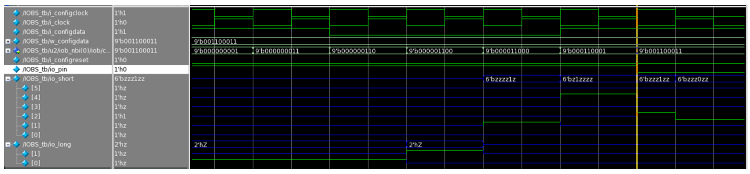

/IOBS_tb/i_configclock 1'h1
/IOBS_tb/i_clock 1'h0
/IOBS_tb/i_configdata 1'h1
/IOBS_tb/w_configdata 9'b001100011
/IOBS_tb/u2/iob_nbi(0)/iob/c... 9'b001100011
/IOBS_tb/i_configreset 1'h0
/IOBS_tb/io_pin 1'h0
/IOBS_tb/io_short 6'bzzz1zz
[5] 1'hz
[4] 1'hz
[3] 1'hz
[2] 1'h1
[1] 1'hz
[0] 1'hz
/IOBS_tb/io_long 2'hz
[1] 1'hz
[0] 1'hz

Figure 11: IOB_receive_simulation

2.6.4 SWB Verification specification

The switching block is responsible for driving the signals between every CLBs . By the SWB , we can change the direction of the data flow. The SWB block is mainly consisted of the Net block and the FIFO block. The Net block is the most important part for the SWB block, which can change the direction of data transfer. And FIFO is used to load the bit stream which will not be paid more attention. On the other hand, according to the figure 3, the SWB have four directions(top, bottom, left and right) and each direction has 6 wires. Therefore, we decide to divide this verification to two parts which are the verification about the Net block alone and the verification about the whole SWB block.

The Net block verification
For the Net block, because it has four directions, we test the output values in the four directions separately as is shown as follow.

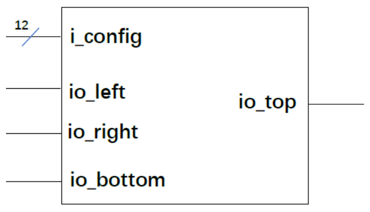


Figure 12: Net_output_to_top

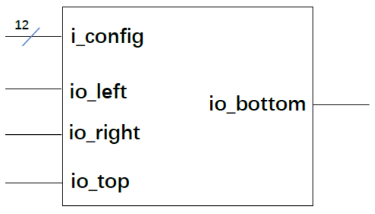


Figure 13: Net_output_to_bottom

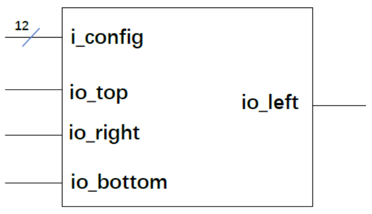


Figure 14: Net_output_to_left

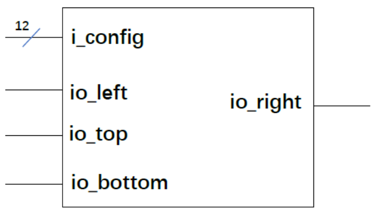


Figure 15: Net_output_to_right

According to these figure, three of the four directions are used as input and the remaining one is used as output. Using this way to make sure all of situation included. On the other hand, the Net block also need the configuration bits to control the output of the data. For one Net Block, the configuration is 12 bits and the meaning about each bit of the configuration bits is shown as follow.

- i_config order 1 is shortcut, 0 is open circuit
- config(0)=1 => write on left wire
- config(1)=1 => write on bottom wire
- config(2)=1 => write on right wire
- config(3)=1 => write on top wire
- config(4-5) => sel to write on left, 00 bottom, 01 top, 10 right
- config(6-7) => sel to write on bottom, 00 right, 01 left, 10 top
- config(8-9) => sel to write on right, 00 bottom, 01 top, 10 left
- config(10-11) => sel to write on top, 00 right, 01 left, 10 bottom

Net functional specification

- Clocked on positive edge of clock
- For one configuration bitstreams, only one direction wire can be written.
- When the bitstreams is loaded, make sure the output is from the right wire according to the bitstream

- the output is combinational

Net Verification strategy

We have four different direction wires, all of them are have same principle. Therefore, the Net Verification strategy is only for the situation about writing on the bottom. The other three wires we also test but here we just give an example of one of them.

- Generate io data and configuration bitstreams using randomization to cover a wide range of DUT
- Specify coverage group
  - covergroup cg @(negedge i_configclockin);
  - wire_left: coverpoint config_left bins config_left=[0:3];
  - wire_right: coverpoint config_right bins config_right=[0:3];
  - wire_top: coverpoint config_top bins config_top=[0:3];
  - wire_bottom: coverpoint config_bottom bins config_bottom=[0:3];
  - wire_test: cross wire_right,wire_left,wire_top,wire_bottom;
  - endgroup
- Specify property for assertions and for coverage
  property test_bottomfromright;
  @(posedge i_configclockin)
  if (w_netconfig_test[7:6]==2'b00)
  io_bottom == io_right;
  endproperty: test_bottomfromright
  bottom_from_right: assert property(test_bottomfromright);

  property test_bottomfromtop;
  @(posedge i_configclockin)
  if (w_netconfig_test[7:6]==2'b10)
  io_bottom == io_top;
  endproperty: test_bottomfromtop
  bottom_from_top: assert property(test_bottomfromtop);

  property test_bottomfromleft;
  @(posedge i_configclockin)
  if (w_netconfig_test[7:6]==2'b01)
  io_bottom == io_left;
  endproperty: test_bottomfromleft
  bottom_from_left: assert property(test_bottomfromleft);

The whole SWB block verification

For the whole SWB block, according to the figure 3, 6 wires are included in the SWB block which means that the SWB includes 6 Net blocks. Therefore, the whole SWB block system needs the 72 bits of the configuration to control the data in each line can be loaded in the correct direction. Besides, like the

IOB , FIFO block is one of the main components of the SWB . It means that the SWB needs 72 clock cycles at least to load a whole configuration bits. During the 72 clock cycles, the output of the SWB is illegal. On the other hand, it must cause multi-driven when the two different direction wires transmit the data to the one wire at the same time. Therefore, using the randomization to generate the configuration bits to test this block is impossible and unnecessary and we do the verification like the IOB ,that is, choosing some specific bit streams and setting the io_configclock as a certain value when the specific configuration bitstreams are loaded fully to make sure the simulation result can be kept.The figure about SWB verification strategy is shown as follow. Here

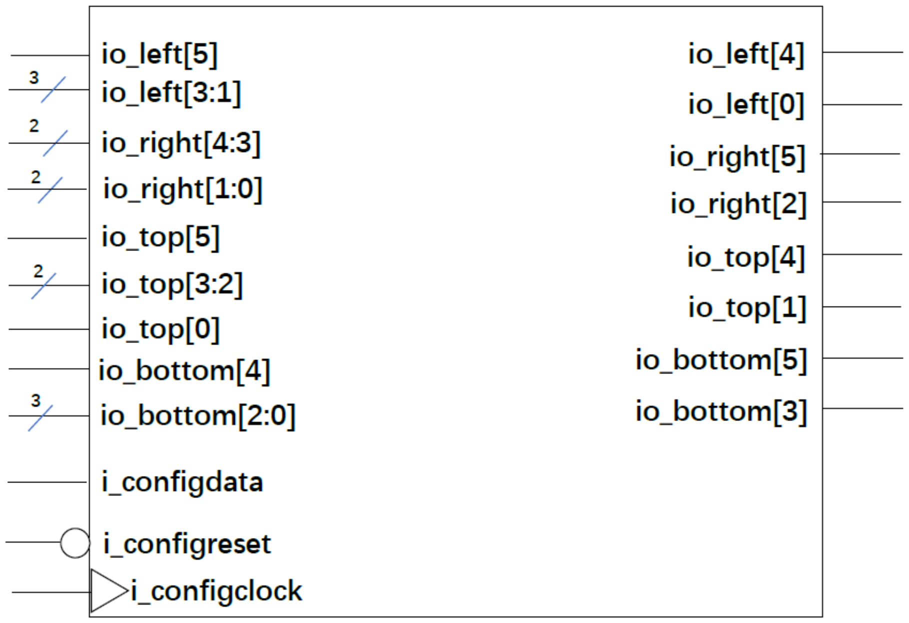


Figure 16: SWB verification strategy

we choose a bunch of bitstreams which can realize (1) io_bottom writes on the io_left on the first wire (2)io_left writes on the io_top on the second wire (3)io_top writes on the io_right on the third wire (4)io_right writes on the io_bottom on the fourth wire (5)io_bottom writes on the io_top and io_right writes on the io_left on the fifth wire (6)io_top writes on the io_bottom and io_left writes on the io_right on the sixth wire. To make sure the wires do not multi-driven, the six wires in each direction are shown in the 16 as input and output.

SWB Functional specification.

- Clocked on positive edge of i_configclock for the i_configdata
- Has an active low reset
- Data in the six wires in each direction transmits correctly according to the configuration bits

SWB Verification Strategy

- Set i_configreset for 5 ns(active low) to reset the configdata
- Give specific i_configdata
- After fully loading the i_configdata, assign the value to specific io_left,io_right,io_top and io_bottom.
- Check the simulation waveform and make sure the output of the io_left,io_right,io_top and io_bottom is right

The simulation result

The simulation result about the Net block is shown as follow. According to the figure 17, the io_bottom is not set the value when the configuration bit is not loaded. And here the config_enable is set as ”0010” which is means that only the io_bottom can be written. Hence, after loading the configuration bit is loaded. The io_bottom can get the value from the corresponsible wire according to the configuration bit. And according to the figure 18, the total coverage and the assertion hit rate can up to 100% which means that the Net module is totally right.

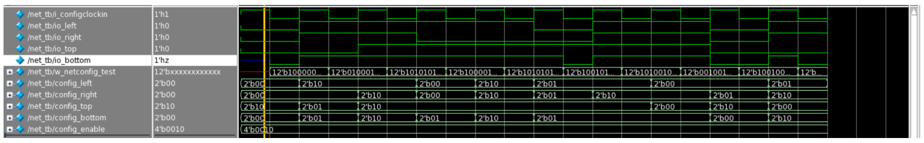
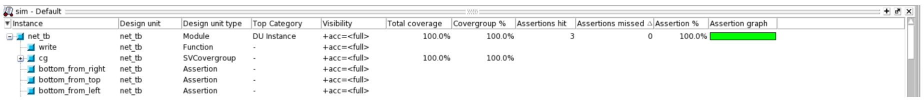


Figure 17: NET simulation waveform



sim - Default

Instance	Design unit	Design unit type	Top Category	Visibility	Total coverage	Covergroup %	Assertions hit	Assertions missed	Assertion %	Assertion graph
net_tb	net_tb	Module	DU Instance	+acc=<full>	100.0%	100.0%	3	0	100.0%	🟩
write	net_tb	Function	-	+acc=<full>						
cg	net_tb	SVCovergroup	-	+acc=<full>	100.0%	100.0%				
bottom_from_right	net_tb	Assertion	-	+acc=<full>						
bottom_from_top	net_tb	Assertion	-	+acc=<full>						
bottom_from_left	net_tb	Assertion	-	+acc=<full>						

Figure 18: Simulation assertion and coverage rate for the Net

The simulation result about the SWB is shown in figure 19.Here we just take one of the configuration bitstreams as an example, in practice we will also test other different configuration bitstreams.As shown in the figure 19,At 5ns, i_configreset is set from 1 to 0, and then the configuration bits are imported. After 72 cycles, the configuration bits are completely imported, and i_configclock is set to a constant value to ensure that the output results remain unchanged. At this time, the assign value is given to six lines in each direction, and the final simulation result is as expected.

2.6.5 CLB Verification specification

The purpose of the CLB is to achieve any programmable basic function. When this block wants to realize some function, it will pre-load this fucntion results, and then, when the software is running, just picking the correct one.

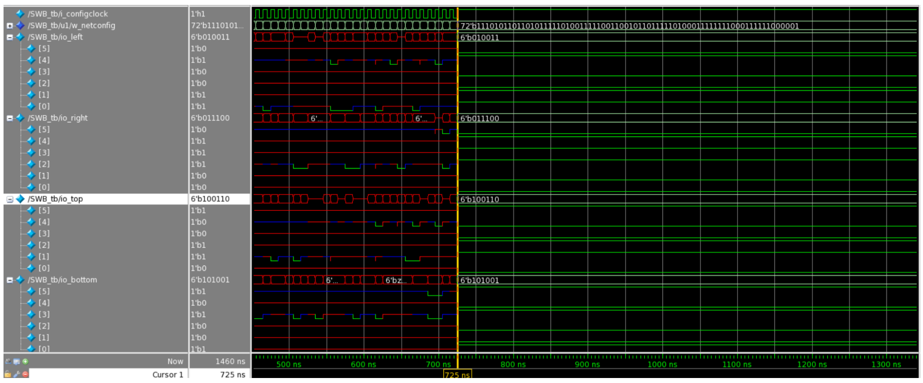

/SWB_tb/i_configclock 1'h1
/SWB_tb/u1/w_netconfig 72'b1110101...
/SWB_tb/io_left 6'b010011
/SWB_tb/io_right 6'b011100
/SWB_tb/io_top 6'b100110
/SWB_tb/io_bottom 6'b101001
Now 1460 ns
Cursor 1 725 ns
72'b111010110110101111101001111100110010110111110100011111111000111111000001
500 ns
600 ns
700 ns
725 ns
800 ns
900 ns
1000 ns
1100 ns
1200 ns
1300 ns

Figure 19: SWB Simulation

These pre-loaded result are 16 bits long, and are stored by FIFO. Hence, like the IOB , this block also needs to load the specific configuration bit firstly and then the expected result will be got. CLB Functional specification.

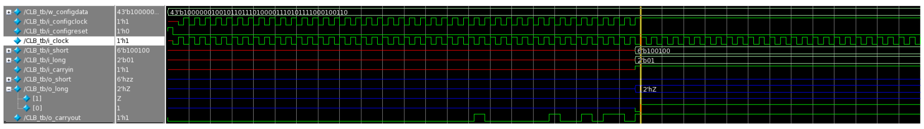
- Clocked on positive edge of i_configclock for the i_configdata
- Clocked on positive edge of i_clock when the output mode is sequential
- Has an active low reset
- choose some specific bitstreams to check if it meets initial requirements.

CLB Verification Strategy

- Set i_configreset for 5 ns(active low) to reset the configdata
- Give specific i_configdata
- After fully loading the i_configdata, assign the value to i_long,i_short.
- Check the simulation waveform and make sure the output of the o_long,o_short.

CLB Simulation result

The simulation result about the CLB block is shown in the figure 20.To keep the right bitstream doesn't change, when the the configuration bits are loaded fully by the FIFO block, we keep the i_configclock as a constant value. What's more, if we input the i_short and i_long before the bitstream loaded, the qusta sim will cause some problem about iterative problem which is because the unnecessary configuration bits generated in the process of importing the bitstream will make clb in a infinite loop state. Hence, the i_short and i_long are assigned value when bitstream is loaded fully. The bitstream for the CLB is 43 bits. Here the specific configuration bits can realize the input(0) is 1, input(1) is 1 and input(2) is one according to the figure 2. Besides, the LUT about the fully-adder is loaded in the configuration bits. Therefore, it is easy to see that the output of these CLB , that is, the carryout is 1 and the sum is one which is output in the o_long[0]. And because the output mode is sequential, the right data can be loaded when the i_clock is on positive edge as the configuration bits are loaded fully.



```
/CLB_tb/w_configdata    43'b100000...
/CLB_tb/i_configclock   1'h1
/CLB_tb/i_configreset   1'h0
/CLB_tb/i_clock         1'h1
/CLB_tb/i_short         6'b100100
/CLB_tb/i_long          2'b01
/CLB_tb/i_carryin       1'h1
/CLB_tb/o_short         6'hzz
/CLB_tb/o_long          2'hZ
  [1]                   Z
  [0]                   1
/CLB_tb/o_carryout      1'h1
```

Figure 20: CLB Simulation

2.6.6 Top-Level Verification specification

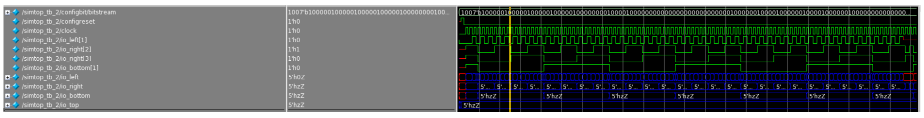
Top level is the final part for the verification. For this block, it more like a geometrical tiling game than simple direct wiring between serial blocks.Ensuring that the connections between modules and modules are as shown in the figure 6 is all that is needed for this part of the verification. And in this part, due to the iterative problem of CLB when the configuration bits import to each block, this design will be in infinite loop state, so initially we chose to import the bitstreams directly to each module. Top level Functional specification.

- Clocked on positive edge of i_clock
- Has an active low reset
- Choose some specific bitstreams to check if it meets initial requirements.

Top level Verification Strategy

- Set i_configreset for 5 ns(active low) to reset the configdata
- Give specific w_configdata
- Fully loading the w_configdata correctly
- Check the simulation waveform

Top level simulation result
The result about the top level is shown as follow. The bitstreams is 1007 bits for the whole FPGA. The configuration bits we load is to realize a counter according to the i_clock, and we can find that the signal chosen is selected is a counter.



```
/simtop_tb_2/configbit/bitstream   1007'b10000010000010000010000010000000100...
/simtop_tb_2/configreset           1'h0
/simtop_tb_2/clock                 1'h0
/simtop_tb_2/io_left[1]            1'h0
/simtop_tb_2/io_right[2]           1'h1
/simtop_tb_2/io_right[3]           1'h0
/simtop_tb_2/io_bottom[1]          1'h0
/simtop_tb_2/io_left               5'h0Z
/simtop_tb_2/io_right              5'hzZ
/simtop_tb_2/io_bottom             5'hzZ
/simtop_tb_2/io_top                5'hzZ
```

Figure 21: Top level Simulation

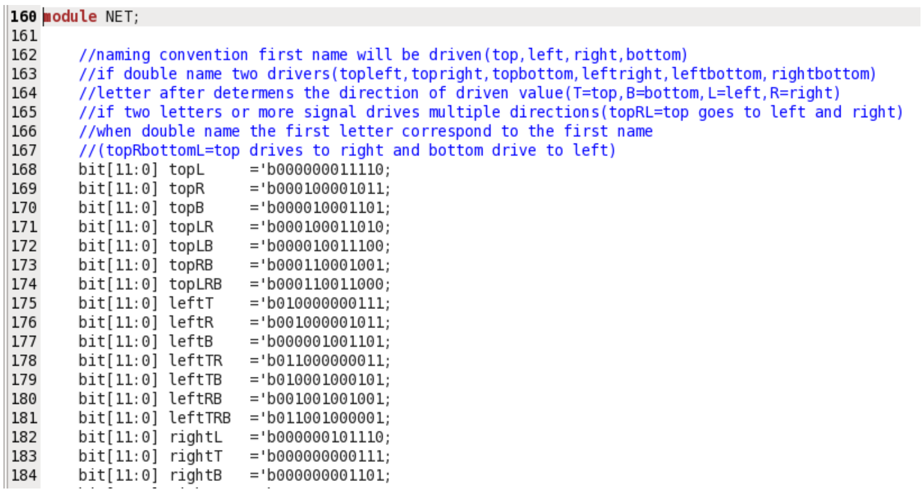
2.7 Configuration bistream setup

Before being able to run any software, a software-specific configuration bitstream must be initialised. Through the configuration FIFO , every block receives its configuration at an hard-implemented location. The FIFO path is drawn on figure 6a.

A system verilog file has been written to slightly automate the process of writing the configuration bitstream. Even for a simple 2×2, the bistream is already a thousand bits long, preventing any direct by-hand writing.

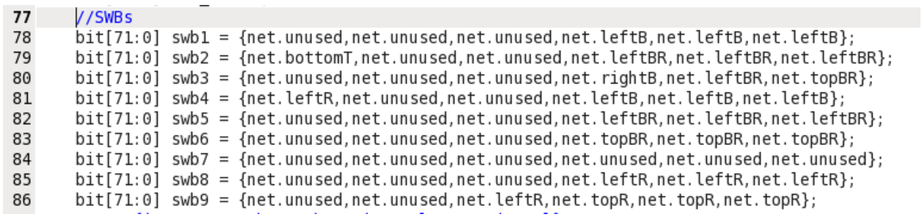
To simplify the bitstream writing, every possible configuration is first written once, given a label, and then simply called when necessary. To implement a software, the user needs to write down the concatenation of each label, at the correct position inside the resulting string, that could run their software. Every basic block code contains a header with a short description of the expected bitstream order and the role of each bit.

This can be shown in figure 22, the figure shows how the SWB was written by dividing in layers and ultimately added to the final bitstream. First layer (figure 22a) is the binary coding for a function of one net inside a SWB , for each possible functionality(connecting a path to one or several paths) hard coding the bits that can easily be confirmed.



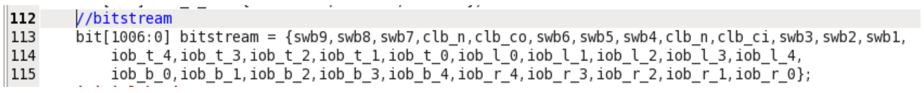
```
module NET;

    //naming convention first name will be driven(top,left,right,bottom)
    //if double name two drivers(topleft,topright,topbottom,leftright,leftbottom,rightbottom)
    //letter after determens the direction of driven value(T=top,B=bottom,L=left,R=right)
    //if two letters or more signal drives multiple directions(topRL=top goes to left and right)
    //when double name the first letter correspond to the first name
    //(topRbottomL=top drives to right and bottom drive to left)
    bit[11:0] topL     ='b000000011110;
    bit[11:0] topR     ='b000100001011;
    bit[11:0] topB     ='b000010001101;
    bit[11:0] topLR    ='b000100011010;
    bit[11:0] topLB    ='b000010011100;
    bit[11:0] topRB    ='b000110001001;
    bit[11:0] topLRB   ='b000110011000;
    bit[11:0] leftT    ='b010000000111;
    bit[11:0] leftR    ='b001000001011;
    bit[11:0] leftB    ='b000001001101;
    bit[11:0] leftTR   ='b011000000011;
    bit[11:0] leftTB   ='b010001000101;
    bit[11:0] leftRB   ='b001001001001;
    bit[11:0] leftTRB  ='b011001000001;
    bit[11:0] rightL   ='b000000101110;
    bit[11:0] rightT   ='b000000000111;
    bit[11:0] rightB   ='b000000001101;
```

(a) Labels for net configuration



```
    //SWBs
    bit[71:0] swb1 = {net.unused,net.unused,net.unused,net.leftB,net.leftB,net.leftB};
    bit[71:0] swb2 = {net.bottomT,net.unused,net.unused,net.leftBR,net.leftBR,net.leftBR};
    bit[71:0] swb3 = {net.unused,net.unused,net.unused,net.rightB,net.leftBR,net.topBR};
    bit[71:0] swb4 = {net.leftR,net.unused,net.unused,net.leftB,net.leftB,net.leftB};
    bit[71:0] swb5 = {net.unused,net.unused,net.unused,net.leftBR,net.leftBR,net.leftBR};
    bit[71:0] swb6 = {net.unused,net.unused,net.unused,net.topBR,net.topBR,net.topBR};
    bit[71:0] swb7 = {net.unused,net.unused,net.unused,net.unused,net.unused,net.unused};
    bit[71:0] swb8 = {net.unused,net.unused,net.unused,net.leftR,net.leftR,net.leftR};
    bit[71:0] swb9 = {net.unused,net.unused,net.leftR,net.topR,net.topR,net.topR};
```

(b) Configuration of Switching blocks



```
    //bitstream
    bit[1006:0] bitstream = {swb9,swb8,swb7,clb_n,clb_co,swb6,swb5,swb4,clb_n,clb_ci,swb3,swb2,swb1,
        iob_t_4,iob_t_3,iob_t_2,iob_t_1,iob_t_0,iob_l_0,iob_l_1,iob_l_2,iob_l_3,iob_l_4,
        iob_b_0,iob_b_1,iob_b_2,iob_b_3,iob_b_4,iob_r_4,iob_r_3,iob_r_2,iob_r_1,iob_r_0};
```

(c) Configuration for whole bitstream

Figure 22: Configuration bitstream file

In figure 22b shows all SWB in the demo as they have six nets in each of them, this helps during debugging if the order of nets have been shifted or remembered incorrectly. If a change is needed writing a different label is less error prone then changing 12 individual ones and zeros. The final figure 22c shows the variable the bitstream saves and in which order all basic blocks are in. For the CLBs and IOBs are written in similar manor as the SWBs to simplify bitstream generation.

No further automation is achieved, as this is the task of a compiler, which role is to generate a bitstream from any software that could run on this device. Still, an attempt has been made to use the *Open-fpga* software as a tool to automate the bistream generation, without success, because of the VHDL source code not matching the verilog requirements.

As a demonstration of a bitstream configuration that would lead to run a software, the configuration bistream of a 4-bits counter has been ”manually” written, with the method above. Although the design has been demonstrated able to achieve 64 CLBs , without any compiler, only a 2×2 design and its thousand bits long bistream is simulated. There is no expected reason for the

8×8 design to be faulty would the 2×2 runs correctly, because the top level makes extensive use of *for* loops, with no numbers hard-coded.

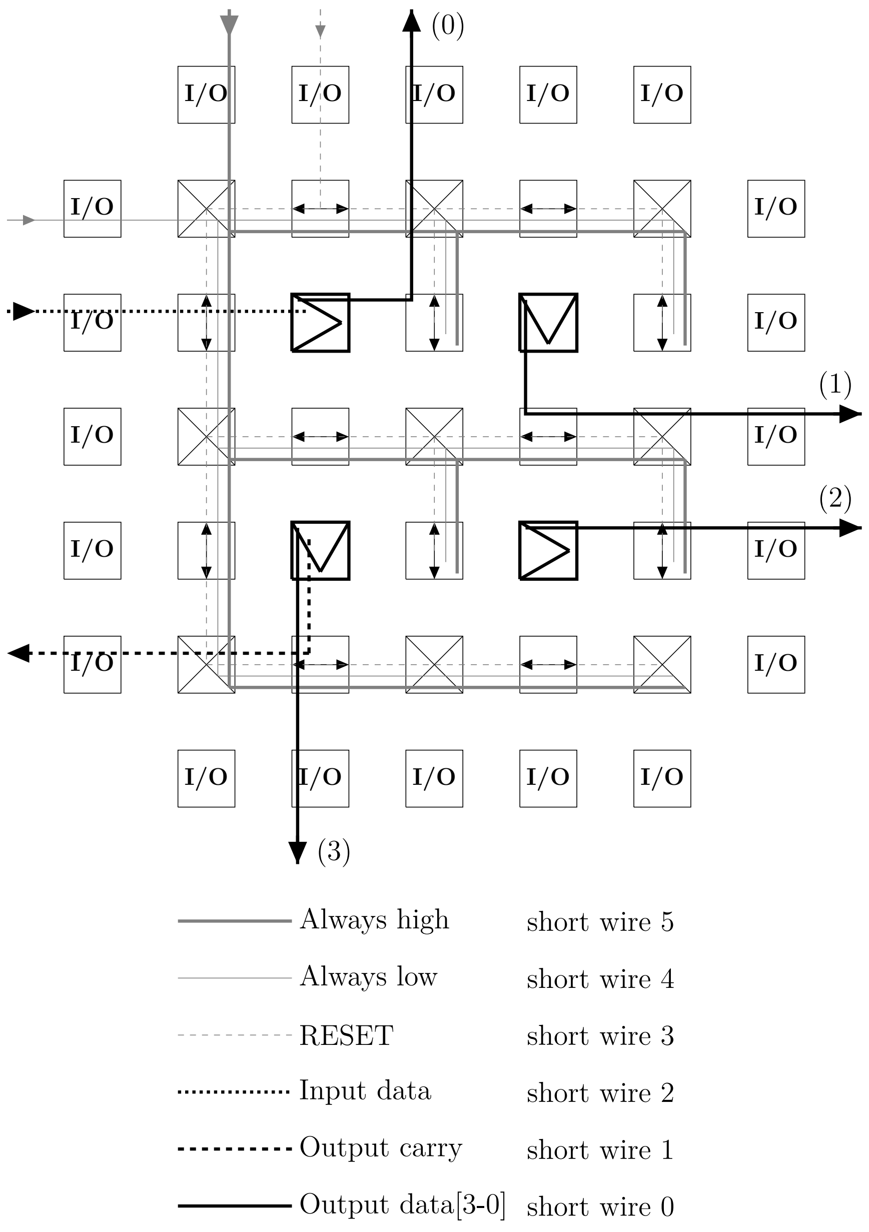


Figure 23: 4 bits counter software routing

Figure 23 displays the resulting wire nets once the configuration bitstream is set. This schematic then gives the pin map and were to look for the results on the digital simulation.

Because the bitstream is still flowing through the FIFO during the configuration phase, it is not possible to digitally simulate it. Indeed, until the last clock tick, the bistream is shifted from its final position, giving quite random configuration to the previous blocks, and zeroes to the last blocks, generating loops and instability. Because the FIFO frequency can be much higher than the rest of the FPGA design, this problem is actually rather insignificant. Physical testing, emulating the design on another FPGA confirms this: the leds effectively blinking in the correct order, being driven by the output data bus.

2.8 OpenFPGA investigation

As an alternative to generate a bitstream for the 4-bit counter demo, the open sourced FPGA IP generator OpenFPGA was investigated to use as compiler for the FPGA design. The following section will explain how and why OpenFPGA would be a fitting tool for future work, but also why it was not used for this project.

Figure 24: Logo from OpenFPGAs github

OpenFPGA is a linux shell operated tool made for prototyping custom FPGA architectures faster than previous development strategies. What takes a team of senior architects and engineers through normal means can be made by a single architect in a time frame of days instead of months. This is possible due to a semi-custom design flow, by the use of Extensible Markup Language (XML) files. XML structured as text file with custom tags that describes the feature and structure of its file. With OpenFPGA these XMLs can auto generate Verilog files describing the architecture from the transistor-level to the top-level of the FPGA fabric all fully costumizable. Follow with auto generation of testbenches and bitstreams of the prototype architecture, even generating the architecture from Verilog files through the use of integrated and compatible third-party tools.

2.8.1 OpenFPGA operation

The requirement to produce a physical FPGA fabric is two XML files one describing the architecture and the second describing the simulation settings. In the architecture file is information about all the circuit modules like LUTs ,MUXES and other primitives that builds the FPGA . Configuration- and technology libraries are also described in it. The configuration library describes how all primitives are connected with one another, and the technology library describes the technology which one can bind for instance the PDK library if it would have been available.

With these two files the complete FPGA can be generated with Verilog files. After the generation a testbench can be generated following the simulation settings XML file including internal test points(for simulation purpose only). Since the output is in Verilog all tools compatible with its type can be used for verification of the design. A second possibility is to use Hardware Description Language (HDL) files such as Verilog or VHDL in a third party tool named Yosys to generate files compatible with OpenFPGA , which would later generate the architecture XML file.

With the XMLs files a bitstream from a Verilog design can be generated with the internal tool FPGA-Bitstream. This helps to test each part of the FPGA , to see if memories are correctly set and implement designs in the prototype FPGA . This feature was the reason for investigation of OpenFPGA capability's, simplifying the bitstream generation process.

OpenFPGA supports also a fabric-key operation, which allow users to create secured chip. If one would try to generate a bitstream without the correct key it would either generate a erroneous bitstream or output an error during generation. This enables more security for open source tooling if wanted.These operation are only a part of the complete functionality of OpenFPGA .

2.8.2 Why OpenFPGA was not used

Even with the capability's earlier shown the tool was not used. Underlying reasons for the decision was that the size of resulting demo would only require 4×4 CLB design and the following.

- OpenFPGA was introduced late in the project and since the designs was described in VHDL conversions to Verilog would be needed. Afterwards XML generation or binding would be required, And the lacking knowledge in XML language amongst the team was weighed.
- since the size of the required bitstream was shy of 1100 bits which is regarded as very small size. The above conversation and generation would result in less yield regarding hours spent and given output. But if the bitstream would been larger as the size of bitstream exponentially grows with larger FPGA design, the gain would have been greater to use OpenFPGA .

However in future works when requirements as chips size might become bigger or a smaller technology is used, resulting in bigger FPGA design. It would be wise to consider OpenFPGA from the beginning of the project as design tool.

2.9 Obstacles & Troubleshooting

This project does not intend to design a state of the art FPGA , which has been demonstrated unfeasible anyway, but to document on the design flow to create a basic FPGA that can demonstrate effectively working.

First off, foreseeing the poor number of CLBs , and by confirming the unfeasibility of a more complex architecture by reproducing a smaller version of *Xilinx* ultra-scale architecture and checking for its area, decision has been taken to strip every non fundamental feature from an FPGA . This is also done as to maximise the chance of a quickly working demonstration; the team on charge having little experience with FPGA design if not any in digital design.

Unlike CPUs, and at least for the very simple design targeted, the most design complexity does not come from the basic blocks, which are extremely simple, but the top level having to tile them altogether. Basic blocks came with little difficulties, and mostly require careful standardisation for easy implementation in the top level. To facilitate the top level design, the configuration FIFO is not designed as a single long chain from which every block read a specific address range, but is instead split among every block as to have a clearer idea of its physical localisation on the chip.

Nevertheless, one major difficulty appeared when dealing with both readable and writable wires. With little knowledge of VHDL language, the inout keyword used to describe input/output wires revealed cumbersome and prone to compilation errors. Furthermore, poor decisions have first been taken on the exact standard of each block, requiring extensive use of inout wires.

In a first design iteration, CLBs and IOBs would not read their inputs directly on the connection wires but had an input interface corresponding to their logic purpose (4 variables inputs, SET and RESET for CLBs , Rx and Tx for IOBs). A specific connection interface block, consisting of the several multiplexers required would connect these blocks to the connections wires. Therefore on the initial figure 5, the connection blocks were real blocks describing these multiplexers. This had been done as to sharply distinguish the basic blocks role, but revealed unnecessarily cumbersome; the multiplication of interfaces between blocks led to wrong inout wires driving. Instead, on a second design iteration, these interfaces have been push into the CLBs and IOBs codes, leading the so-called connection blocks visible on figure 1 to only hide the direct wires on the top level description, but not any cell anymore.

The same difficulties happened when describing the I/O pins. Because of the lack of technical knowledge when using the inout type, the compiler would detect a multi-driven net at the I/O pin location, because of the two Rx and Tx paths. A few VHDL code tricks were enough to allow the design to compile but would still give later errors when trying to move the code to the physical FPGA board for testing.

Eventually, when every basic block was able to compile and simulate, the main difficulty was actually to combine them altogether. The top level description is then more of a puzzle game to correctly tile them altogether without faulty connections nor short circuits. Because of this uncanny exercise, its verification lacked method. Apart from checking every wire one by one on an elaborated view of the design, making extensive use of sketches and schematics to verify the wire net, it has not been possible to formally verify the interconnections. Every top level wiring has been checked by hand on a 2×2 design.

An odd-height design is also checked as it changes the side of the carry chain output location.

More difficulties come when downloading the design to the testing FPGA board. Indeed, the software requires to map the design to its own CLBs , and would detect wire loops. Because of the unique FPGA architecture, where every block is connected to one-another, these wire loops are bound to happen but do not represent actual code errors. The only solution found on the not very beginner friendly software interface was to manually shut down the error flag. This, however, would also shut down real error loops detected.

These loops, and their poor handling, will also make timing analysis unavailable, which unfortunately was already the case. The only information known about the PDK at the time of the design phase is the 300ps taken to go through a NAND gate, and the 1μm long gate. A basic cell library has been created based on this information, extrapolated to create DFFs , MUXES and NOR gates. However, this basic library showed insufficient for the software used to synthesise the design, asking for more and more mandatory cells and wires constraints. Because the cell library was not the project target, nor the timing analysis, the FPGA design being only required to show proof of working, it has been dropped and replaced with the LSI10K library, having different timings.

Last but not least, it is also impossible to simulate the FPGA during the configuration phase. A regular FPGA use imply a first phase, during which the software is loaded in. The configuration bitstream flows through the FIFO until the last bit is correctly set. During this loading, every block is then randomly configured during one clock period, possibly taking illegal configuration. Because the FPGA is not meant to give results during this phase, it is not an issue the CLBs give meaningless results. However, it is enough for the simulation software to detect infinite loops, because, for instance, of the loop-back feature of the CLBs generating an oscillator. Artificially increasing a delay to every CLBs output does not solve the issue either, as the looping could also happen in the IOBs or even among SWBs . Therefore, the loading phase remains unable to simulate, preventing any testing.

For this reason, the whole design is copied and slightly modified, to remove the FIFO . In every block, the FIFO path is removed and replaced with parallels inputs, set up by the testbench. This ”nosetup” version allows to simulate the circuit without the loading phase. Because the FIFO is a simple block, there is not much risk for it to not operate correctly. This is confirmed by the physical simulation, where the counter software was able to operate correctly, leading to the confident assertion the FIFO chain works as intended.

2.10 Conclusion

The FPGA architecture design demonstrates operative, both by digital and physical demonstration with simple examples. However, although this project objective has never been to propose a powerful design, it remains that this current design can barely be called an FPGA .

Core features had to be stripped in order to keep simplicity and to deliver a working architecture in due time. The one most important lacking feature being the bidirectional I/Os. This design also lacks moderns features like combined CLBs , more complex connection wires net, and most importantly, distributed RAM blocks. Although this design achieve homogeneity, it is more by area constraints than by real decision-making. The necessary huge RAM space needed by any FPGA will have to be pushed to external chips.

Overall, although the architecture proves working, a lot of trickery had to be achieved to make the circuit operate. Because of poor softwares knowledge, many error flags have been manually shut down as to quickly move to simulation, but the verification globally lacks rigour.

Furthermore, the architecture barely ”works”, but with so little performances that FPGA design based on this PDK , unless perhaps with more advanced techniques and knowledge, looks rather discouraged.

2.11 Future work

Although the FPGA conclusion is pessimistic, a few modification could be made to slightly increase its performances, though still not reaching any form of competitiveness.

A major possible upgrade could be the replacement of the nets among SWBs with pass gates, would the PDK allow it. On the current 8×8 design, this could easily save up to 40% of the area as explored into SWB section.

The configuration bistream memory could also be replaced, for more area effective memory cells than DFFs . Replacing 8 units size DFFs by 6 units size latches for instance, could save up to 15% of the total area, (less than that would the pass gates be used).

The verification of the architecture lacks rigour. Only a single counter software has been tested, on a separate 2×2 elaborated design. Although increasing the number of CLBs is not expected to change the circuit correct functioning, this has not been tested due to the lack of a compiler. Further verification of the whole design shall not be too much.

Would this project be continued, the next design step is to move to the software part. To run a software, the FPGA first requires a configuration bitstream, inherent to any new software, it is therefore required to develop a compiler specific to this architecture to allow any software to run on this device. Since the *Open-FPGA* method has not been available, a whole new compiler is required, yet its complexity should be limited due to the low-feature design.

3 RISC-V

3.1 Introduction

The goal of RISC-V work is to design a basic RISC-V processor that is capable of executing the RV32IM(v2.0) ISA, and fits into a $7 \times 7\text{mm}^2$ chip. It intends to introduce the design method of the processor, including a 3-stage pipeline design, verification strategy applied to the design, and all the difficulties met during the project.

Section 3.2 briefly introduces the concepts of a RISC-V ISA and explains the design of Reduced Instruction Set Computer (RISC) top level and all components, which include all pipeline stages of the processor and peripheral circuits. More unique design strategies used in the processor are included in section 3.3 with explanation of the DEMO program used in presentation. The designs at different stages are verified in section 3.4 to verify the correctness of the design. After the whole design is completed, functional simulations are performed to test all functions the processor supposed to have, which is described in section 3.5. Since this design will be manufactured in later research or study, we use two different synthesis tools to synthesis the processor design, and the results are recorded in section 3.6. This RISC-V sub-project has its own sub-conclusion and future work in sections 3.7 and 3.8.

3.2 Theoretical Framework

3.2.1 ISA

A RISC is a computer designed to simplify the instructions which are given to computer to execute a task. In a RISC processor, each instruction performs only one function unlike in a Complex Instruction Set Computer (CISC) processor. This makes it simpler to implement an instruction pipeline which allows processor running at a higher speed. RISC-V is a new ISA based on RISC principles. It is designed to support computer architecture research and education. Provided under open source licenses, it requires no fees to use.

A RISC-V ISA is defined as a base integer ISA , which must be present in any implementation, combined with optional extensions to the base ISA . The base integer ISAs are very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encoding. A base is restricted to a minimal set of instructions sufficient for basic functions which can be used as a skeleton of customized ISA .

RISC-V is actually a family of related ISAs. There are currently 4 base ISAs in RISC-V family: RV32I, RV64I, RV128I and RV32E. Each base instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers. There are two primary base integer variants, RV32I and RV64I, which provide 32-bit and 64-bit address space respectively. RV32E is a variant of RV32I, which has half the number of integer registers and was designed for small microcontrollers. All base ISAs use a two's-complement representation for signed integer values.

RISC-V has been designed to support extensive customization and specialization. Each base integer ISA can be extended with one or more optional instruction-set extensions. A set of standard extensions are defined to provide integer multiply/divide, atomic operations, and single and double-precision floating-point arithmetic. The base integer ISA is named ”I”, and contains integer computational instructions, integer loads, integer stores, and control-flow instructions. The standard integer multiplication and division extension is named ”M”, and adds instructions to multiply and divide values held in the integer registers.

In the base RV32I ISA , there are six instruction formats (R/I/S/B/U/J). All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. Figure 25 shows the formats of RISC-V base instruction types.

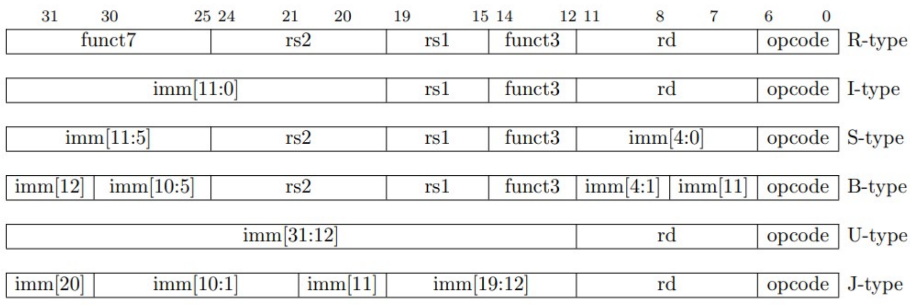

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 25: RISC-V base instruction formats

The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Immediate numbers are always sign-extended. The sign bit for all immediate numbers is always in bit 31 of the instruction to speed sign-extension circuitry.

During the project, we designed a basic RISC-V processor capable of executing the RV32IM ISA which includes Base 32-bit Integer Instruction Set and Standard Extension for Integer Multiplication and Division. RV32I contains 40 unique instructions and can be reduced to 38 in total. M extension contains instructions that multiple or divide values held in two registers.

3.2.2 RISC-V SoC

A RISC-V platform can contain one or more RISC-V processing cores together with other non-RISC-V-compatible cores, fixed-function accelerators, various physical memory structures, Input/Output (I/O) devices, and an interconnect structure to allow the components to communicate.

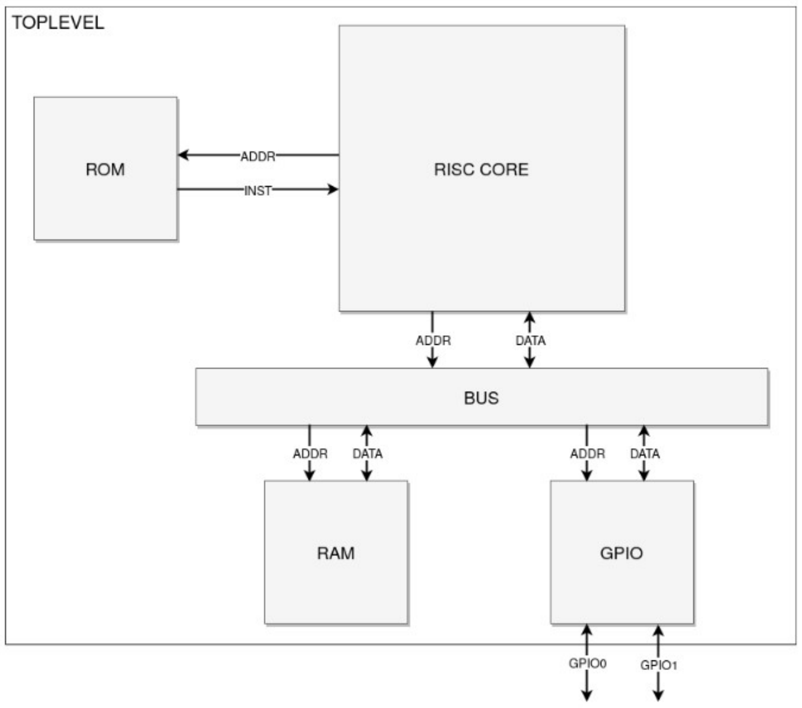


Figure 26: RISC-V SoC structure

We have one RISC-V processing core in our SoC design. A component is termed a core if it contains an independent instruction fetch unit. A RISC-V core might support multiple hardware threads through multithreading, but our core does not support multithreading. The design of the RISC-V core will be explained later.

A RISC-V core might have additional specialized instruction-set extensions or an added coprocessor, which indicates a unit that is attached to a RISC-V core and is mostly sequenced by a RISC-V instruction stream that contains instruction-set extensions. We have a divider attached to the RISC-V core, which is used to execute division and remain instructions from RV32M extension.

Besides the processors, we have a ROM which is used to store instruction to be executed and a RAM which is used as the main memory. We use a bus module to connect all components together, and a GPIO component that acts as I/O device and allows the SoC to interact with outer world.

3.2.3 Core

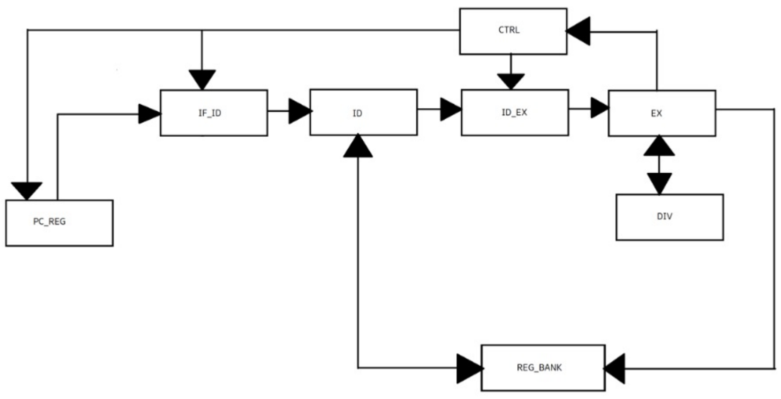


Figure 27: RISC-V core structure

The RISC-V core, or the RISC-V processor, will continuously execute instructions which are stored in the ROM. It will fetch (output) data signal from (to) RAM or GPIO according to instruction being executed. The RISC-V core has a 3-stage pipeline, which allows the processor to run at higher frequency. The 3 stages are:

1. instruction fetch
2. instruction decode, read from registers or RAM
3. computation and data writeback

The processor has 8 components including the division coprocessor, which are pc (program counter register), if_id (instruction fetch and pipeline unit), id (instruction decoder), id_ex (pipeline unit between id and ex), ex (execution), div (division), ctrl (control), and reg (registers). Each component will be further explained in next paragraphs.

- **Registers and pc register:**

For RV32I, 32 registers are each 32 bits wide. All 32 registers have synchronous write and asynchronous read. Register x0 is hardwired with all bits equal to 0. General purpose registers $x1 - x31$ hold values that various instructions interpret as a collection of Boolean values, or as two's complement signed binary integers or unsigned binary integers.

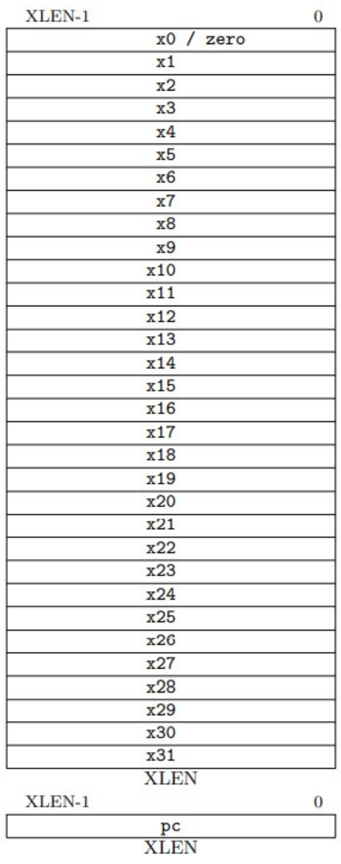


Figure 28: RISC-V core registers (XLEN = 32)

The register bank has two read ports and one write ports. Two read ports allow two registers to be read in one clock cycle, which happens in the instruction decode stage of the pipeline. In order to read the value from the registers, 5-bit register address must be given to select a register. The output will appear right after the address is asserted. For writing to one of the register, write register address, write enable, and write data must be given. The data will be written into the register after the posedge of the clock.

There is one additional register: the program counter pc holds the address of the current instruction. The value of pc will increase 0x4 at each posedge of the clock, in order to point to the next instruction stored in the ROM. When the jump flag is asserted, which means a jump or a branch instruction is executed, the pc register will take the jump address as input at next posedge instead of increasing 0x4 to execute the jump or branch instruction. When the hold flag is asserted, which means the pipeline need to be stop, the pc register will take current output as input at next posedge instead of increasing 0x4 to hold the pipeline and prevent fetching next instruction.

When the active low reset signal is asserted, all 32 registers will be reset to 0x00000000. The program counter will be also reset to 0x00000000, which

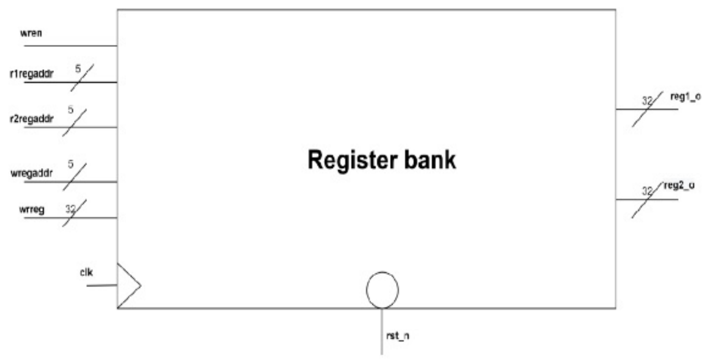


Figure 29: RISC-V core register bank

points to the first instruction stored in the ROM. All registers will start running at the first posedge after de-asserting the reset signal.

- **Pipeline unit if_id and id_ex:**

if_id unit fetches instructions from ROM and passes to instruction decoder id unit at each posedge. id_ex fetches decoded instructions from id and data from registers and passes them to execution unit ex. When the reset signal or the hold flag is asserted, both units will output instruction NOP to prevent further execution in the next unit. Other output signals will be 0x0 because all of them are data signals rather than control signals.

In fact, if_id and id_ex are two groups of registers, which are placed between two large combinational circuits to get shorter critical path in timing and to create a pipeline structure in the data path. So these two units are called pipeline units. By shortening the critical path, the clock period is shorter, then a higher clock frequency can be reached. However, using pipeline structure in the processor brings some problem to the design. First, when a branch or jump instruction is executing, the whole pipeline needs to be emptied. And some clock cycles will be lost according to the design. Next, because the division operation needs more than one clock cycle to execute, the pipeline needs to be paused during the division operation. Control signals is designed to solve this problem.

- **Instruction decoder:**

The instruction decoder id is a large combinational logic circuit. id decodes input instructions and passes the decoded instructions to id_ex unit for later execution. id will output register read addresses to the register bank to fetch register data and pass the register data to id_ex unit. If the instruction requires register writing, id will also output register write enable and write address to register bank for later register writing in execution stage.

Each instructions have at most 6 parts:

1. opcode
2. function3

3. function 7
4. destination register
5. source register 1
6. source register 2

The decoder first extracts opcode, so it knows the type of the instruction. Then the instructions are decoded according their function3, which indicates different instructions in the same type. Sometimes the function 7 is also needed to decode the instruction, for example instruction ADD and MUL have same opcode and same function3, but their function7 are different. After knowing the exact instruction, the decoder will extract register addresses and operands from the instruction itself, and output these signals for later execution.

- **Execution unit:**

The execution unit ex is a key combinational circuit in which the functions of all instructions are achieved. According to instruction input, the ex unit will process the data input and output the result. Most instructions can be finished by ex unit in one clock cycle, but all division and remain instructions require div unit and need 32 clock cycles to finish. The ex unit fetches data from pipeline unit id_ex, co-processor div, and from interconnection unit bus. It uses these data to finish the instructions and output its data to bus, registers. It also outputs to div unit and ctrl unit, when there is a division or remain instruction, a hold flag, or a jump flag needed.

- **Division unit:**

The division unit can be used to calculate the quotient and remainder. It used the following method to implement:

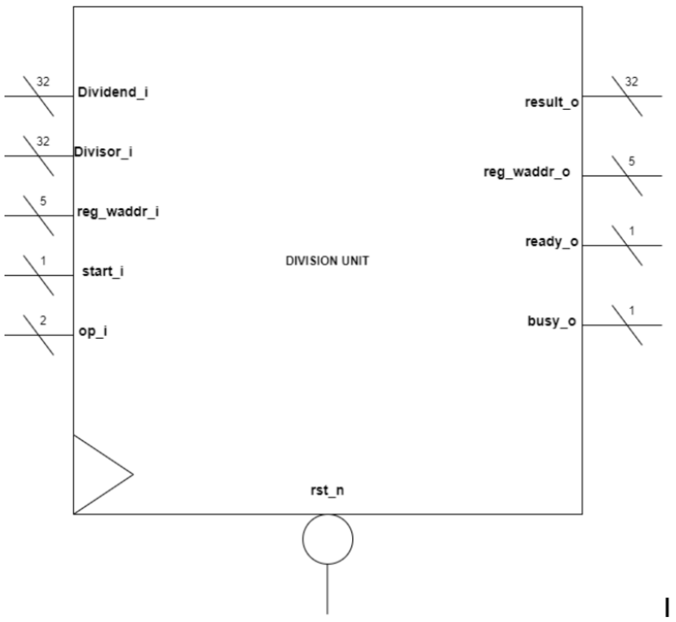
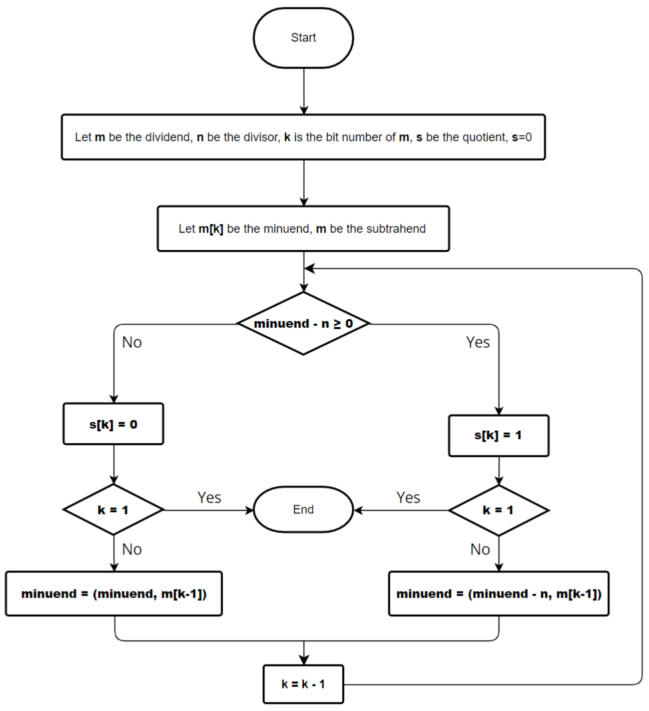


Figure 30: Division unit



Start
Let m be the dividend, n be the divisor, k is the bit number of m, s be the quotient, s=0
Let m[k] be the minuend, m be the subtrahend
minuend - n ≥ 0
No
Yes
s[k] = 0
s[k] = 1
k = 1
Yes
End
Yes
k = 1
No
No
minuend = (minuend, m[k-1])
minuend = (minuend - n, m[k-1])
k = k - 1

Figure 31: RISC-V Division unit algorithm

- **Control unit:**

The control (ctrl) unit handles the jump and hold signal of the pipeline. The flag output from ex unit and bus go into the ctrl unit, then are processed by the control logic. At last, ctrl unit output hold and jump flag, and jump address to all pipeline units to control the pipeline.

- **Interrupt functional unit and CSR register:**

Interrupt can be regarded as a special type of jump. There are two kinds of interruption. One is synchronous interruption, which is produced by instruction ECALL or EBREAK. The other one is asynchronous interruption, which is produced by peripheral like GPIO, UART etc. Therefore, in this unit, a state machine would be used to determine which interrupt type it is. Different interrupt types need to read and write different CSR registers.

In this report, the demo core did not use interrupt functional unit and CSR register. Because the ECALL and the EBREAK instruction are not in our target instruction set. The asynchronous interruption would be sent from bus and received by control (ctrl) unit.

3.2.4 ROM & RAM

Our RISC-V processor has a single byte-addressable address space of 2^{32} bytes for all memory address. A word of memory is defined as 32 bits (4 bytes). A halfword is 16 bits (2 bytes), and a double word is 64 bits (8 bytes). The memory address space is circular, so that the byte at address $2^{32}-1$ is adjacent to the byte at address zero.

Different address ranges of the address space may (1) be vacant, or (2) contain main memory, or (3) contain one or more I/O devices. Reads and writes of I/O devices may have visible side effects, but accesses to main memory cannot.

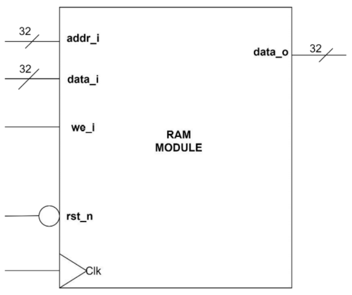


Figure 32: RAM

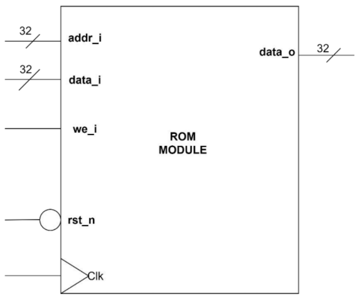
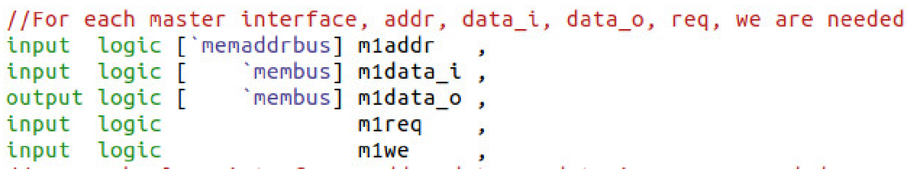


Figure 33: ROM

ROM is a program storage module which is used for storing instruction list. According to the structure of RISC-V SoC , the input of the ROM is from the pc_regs to get address and output to the if_id. While RAM is a data storage module which is used for saving the data. And it can only communicate with bus. The information about their ports are shown as follow.The input about these module are addr_i and data_i which are represented by 32 bits, write enable signal, reset and clock and the output is data_o which is represented by 32 bits.Their design have same principle. In the process of writing, both of them are clocked on positive edge of clock, and controlled by the write enable signal.In the process of reading, both of them are combination. It means that they are not effected by the clock. When reset bit is low, they will immediately output the value stored at the current address.

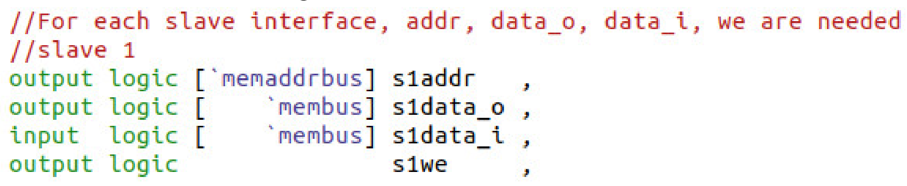
3.2.5 Bus

Bus is a combinational circuit that connect the bus master RISC-V core to different bus slaves such as RAM and GPIO . The bus has a master interface and a slave interface, which must be matched when a component wants to connect to the bus.



```
//For each master interface, addr, data_i, data_o, req, we are needed
input  logic [`memaddrbus] m1addr   ,
input  logic [    `membus] m1data_i ,
output logic [    `membus] m1data_o ,
input  logic               m1req    ,
input  logic               m1we     ,
```

Figure 34: Bus master interface



```
//For each slave interface, addr, data_o, data_i, we are needed
//slave 1
output logic [`memaddrbus] s1addr   ,
output logic [    `membus] s1data_o ,
input  logic [    `membus] s1data_i ,
output logic               s1we     ,
```

Figure 35: Bus slave interface

When the master component wants to write to a slave, it sets up address, input data, bus request, and write enable signals. The bus then creates a bus grant signal inside the bus, which ensures this bus access. According to the address input, the bus outputs address, write data, and write enable signals to the selected slave. The data is written into the slave.

When the master component wants to read from a slave, it sets up address, bus request signals. The bus then creates a bus grant signal inside the bus, which ensures this bus access. According to the address input, the bus outputs address signal to the selected slave. The output of the slave will be fetched by the bus and be passed to the bus master.

The selection of bus master is done by fixed priority bus arbitration. According to the setting of the bus, different bus masters have different bus priorities. Bus master with higher priority will be granted first. The selection of the slave is done by address segment. Currently, the first two Most Significant Bits (MSBs) are used to indicate bus slave. Because only two MSBs are used, at most 4 slave can be connect to the bus. The bus selection bits take 2 bits away from the address, so for each slave, 2^{30} bytes can be accessed, which means the maximum size of each slave is 2^{30} bytes.

3.2.6 GPIO

For the GPIO , it is just a simple I/O port module in our RISC-V processor, mainly used for lighting debugging.

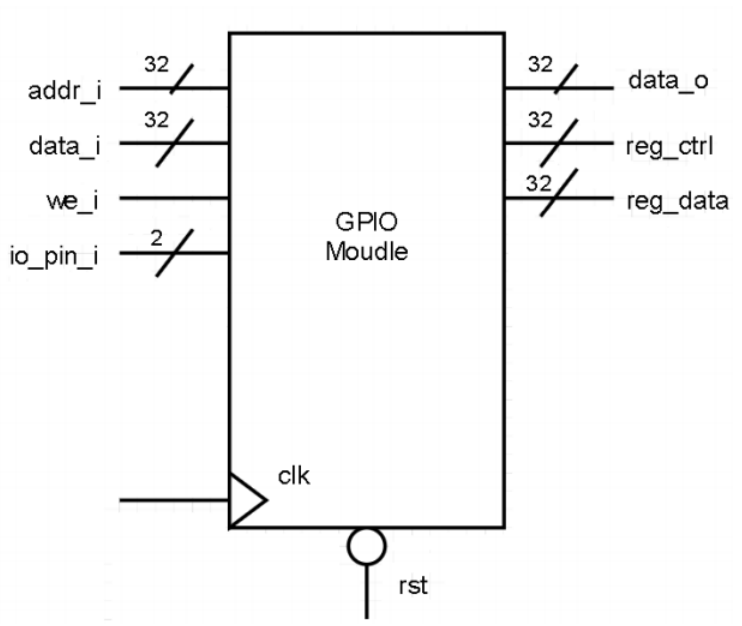


Figure 36: GPIO

Both the data and address of GPIO is the size of one word, that is, 32 bits. According to the structure of RISC-V SoC , we have designed two GPIO modules. The data and address are passed to the GPIO module through the bus, and the data in the GPIO is also transmitted to RISC-V through the bus.The information about its ports are shown as follow. It is similar to the ROM and RAM. GPIO has built-in control registers and data registers. For control bits, every two bits can control the type of one I/O port. Therefore, each module can control 16 I/O ports. The modes of these I/O ports are input, output and high impedance state. In the process of writing, it is clocked in positive edge of clock and controlled by the reset bit and write enable signal. It has two situations in the process of writing. When the write enable signal is active, the lower four bits of the input address will determine whether the input data is stored in the data register or the control register, while if the write enable signal is low, io_pin will input the data to the built-in data registers and the built-in control register will determine which bit io_pin will output to the built-in data register. In the process of read, GPIO is not effect by the clock and write enable signal but effected by the reset signal. And the lower four bits of the input address determine whether the data_o is from the control register or the data register.

3.3 Design Strategy

3.3.1 C2 internal signal

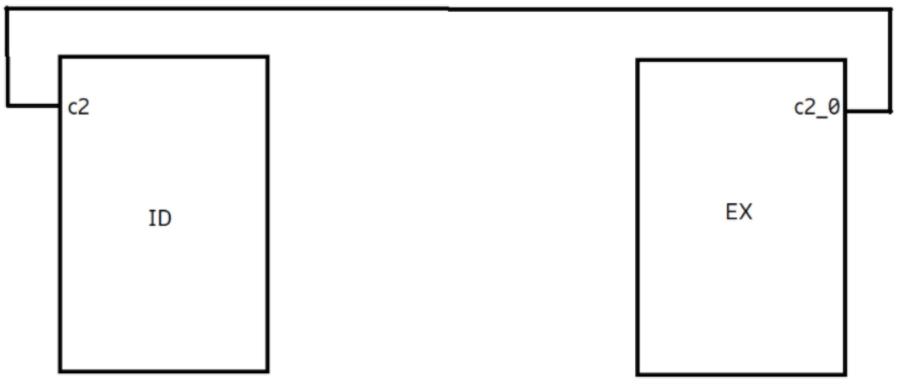


Figure 37: C2 internal signal connection

The C2 signal as can be seen from Figure 37 is used to connect the ex and the and id unit. The reason why we defined C2 was to ensure that every time the jump instructions result in the jump_flag being asserted or the division instruction asserted the div_start flag, these flags should be asserted for more than one clock cycle in the case of jump instructions and 32 cycles in the case of division instructions and during the period when these flags are asserted none of the instructions fetched after the jump instrutions are to be passed on to the execution unit instead we pass NOP instruction to the execution unit. The ex unit controls for how many clock cycles C2 should be asserted for and the id unit depending on the value of C2 passes the instructions or NOP to the id_ex unit.

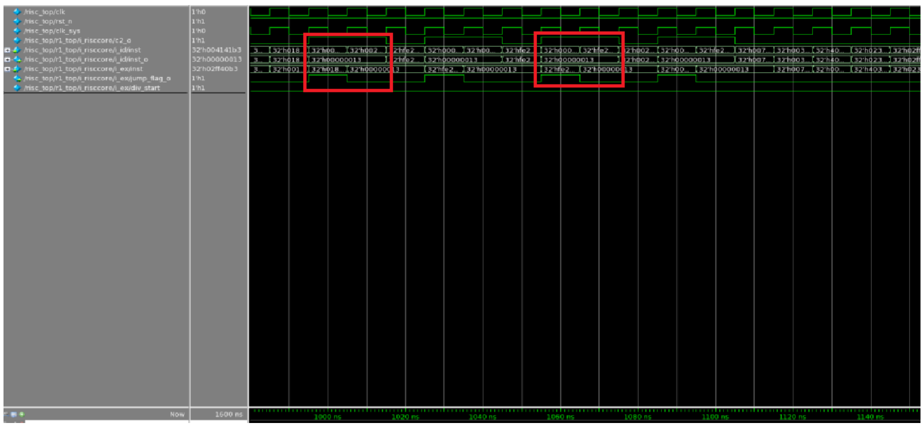

/risc_top/clk 1'h0
/risc_top/rst_n 1'h1
/risc_top/clk_sys 1'h0
/risc_top/r1_top/i_risccore/c2_o 1'h1
/risc_top/r1_top/i_risccore/i_id/inst 32'h004141b3
/risc_top/r1_top/i_risccore/i_id/inst_o 32'h00000013
/risc_top/r1_top/i_risccore/i_ex/inst 32'h02ff40b3
/risc_top/r1_top/i_risccore/i_ex/jump_flag_o 1'h1
/risc_top/r1_top/i_risccore/i_ex/div_start 1'h1

Figure 38: c2_o asserted when a jump instruction is executed

From Figure 39 is can be seen that every time the div_start is asserted the C2 is asserted for 32 cycles to ensure that no instruction other than a NOP (32'h00000013) are output from the id unit.

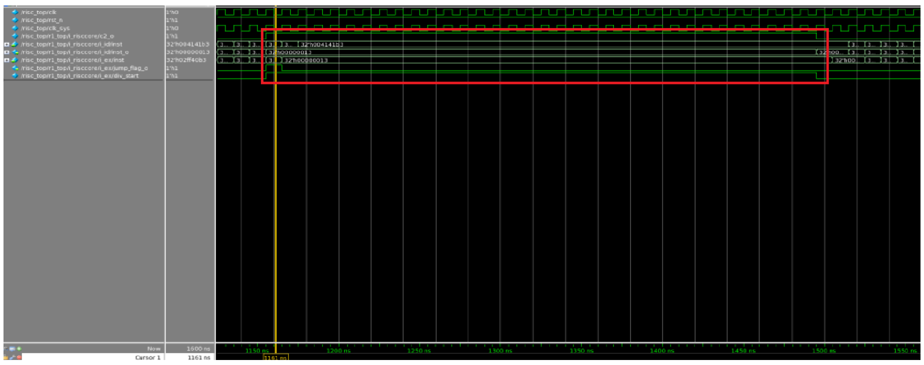

/risc_top/clk
/risc_top/rst_n
/risc_top/clk_sys
/risc_top/r1_top/i_risccore/c2_o
/risc_top/r1_top/i_risccore/i_id/inst
/risc_top/r1_top/i_risccore/i_id/inst_o
/risc_top/r1_top/i_risccore/i_ex/inst
/risc_top/r1_top/i_risccore/i_ex/jump_flag_o
/risc_top/r1_top/i_risccore/i_ex/div_start
1'h0
1'h1
1'h0
1'h1
32'h004141b3
32'h00000013
32'h02ff40b3
1'h1
1'h1
32'h004141b3
32'h00000013
32'h00000013

Figure 39: c2_o asserted when a div instruction is executed

3.3.2 Latch used in the Division unit

This latch is intended to update next_minunend_temp according to the count value. The next_minuend_temp should be updated before the next consecutive clock cycle as it is used to assign value for the intermediate division result.

```
always @(count,rst_n) begin
        if (rst_n==`rstenable) begin
                next_minuend_tmp = `zeroword;
        end else if(|count)
                next_minuend_tmp = {minuend_tmp[30:0],dividend_r[30]};
        else begin
                next_minuend_tmp = next_minuend_tmp;
        end
end
```

Figure 40: Intended latch in the Division unit

3.3.3 Instruction set implemented for functional verification

The following set of base instructions from the RV32I ISA were implemented in the functional unit of the RISC-V core.

I-Type	R-Type	M-Type	S-Type	B-Type	J-Type
ADDI	ADD	MUL	SW	BEQ	JAL
ANDI	AND	DIV		BNE	
ORI	OR	REM		BLT	
XORI	XOR			BGE	
SRLI	SUB				
SLLI	SLL				
LW	SRL				
NOP	SLL				

Table 2: Implemented ISA

3.3.4 DEMO program explanation

No.	Instructions(Hex)	Instructions	Explanation
1	ADDI X31,X0,1	00100F93	Set $x31$ to 1
2	SLLI X31,X31,31	01FF9F93	Set the MSB of $x31$ to 1 to address GPIO
3	SW X31,X5,0	005FA023	Set GPIO ctrl reg to 5
4	SW X31,X3,4	003FA223	Set GPIO data reg to 0011(1000)
5	ADDI X29,X0,20	01400E93	Loop for 20 times ($index = 20$)
6	XOR X3,X3,X11	00B1C1B3	($X3 = not\ X3$)
7	SW X31,X3,4	003FA223	Set GPIO data reg to 1000(0011)
8	SUB X29,X29,X1	401E8EB3	$index - 1$
9	BNE X29,X0,-24	FE0E94E3	Jump to instruction 6 if $index! = 0$

Table 3: DEMO program

Before the DEMO program, registers x1-x30 are given values 1-30. Instruction 1 and 2 set $x31$ to $0x80000000$, which acts as the index address referring to the GPIO registers. Instruction 3 set the control register in GPIO to 5. This register controls the I/O mode of GPIO ports. The value 5 means both GPIO ports are set to output mode. Instruction 4 set the data register in GPIO to 3(0011). This register controls the output value of GPIO ports. The value 3 means the output values of both GPIO ports are 1(LED on). Instruction 5-9 is a loop whose index controls the total LED change times. For our DEMO program, we set the change time to 20, so the LED will turn on for 10 times and turn off for 10 times. During each iteration, the value stored in $x3$ is inverted by xor-ing the value with $x11$(1011), because $x\ xor\ 1$ equals to $NOT\ x$. Instruction 7 stores the inverted value into the GPIO data register, so the LED turns off(on). Instruction 8 decreases the loop index by 1 and if it is reduced to 0, the program ends, if not, the program will jump to instruction 6 and starts next iteration.

3.4 Verification Specification

This section will describe the verification strategy that was planned for the verification sign off for all the blocks and top level DUT.

3.4.1 RAM and ROM Verification specification

- RAM and ROM functional specification.
  - Clocked on positive edge of clock
  - Has an active low reset
  - Write enable signal (we_i) to be asserted to write to RAM and ROM.
  - All writes are sequential
  - All Reads are combinational
- RAM and ROM Verification strategy.
  - Assert rst_n for 10ns (active low) to reset the DUT
  - Assert we_i high same time as de-asserting rst_n
  - Generate addr_i and data_i using randomization to cover a wide range of DUT
  - Specify property for assertions and for coverage
    property ADDR;
    @(negedge clk)disable iff(!rst_n)
    (we_i|->(u1._ram[addr_i[31:2]]==data_o););
    endproperty
  - Check assertion pass count and cover property count (Assertion pass count should be 100 percent)

3.4.2 GPIO Verification specification

- GPIO Functional specification.
  - Clocked on positive edge of clock
  - Has an active low reset
  - Write enable signal(we_i) to be asserted to write to GPIO
  - All write to the GPIO is sequential
  - Read from GPIO are combinational
  - Each two bits of reg_control one I/O mode, 16 I/O can be control
  - 0: Z 1 input 2 output
  - io_pin_i is used to do lighting test.
- GPIO Verification Strategy.
  - Assert rst for 10 ns(active low) to reset the DUT
  - Assert we_i high same time as de-asserting rst
  - Generate addr_i and data_i using randomization when we_i is high to initialize

– Create a special situation that we_i is low and let the io_pin output the value in reg_data to make sure the assertion A3 and A4 are hit.

– Generate addr_i and data_i using randomization to cover a wide range of DUT.

– Specify coverage group
covergroup cg @(negedge clk);
Enable: coverpoint we_i bins EN_LOW[]=0;
bins EN_HIGH[]=1;
INPUT_ADDRESS: coverpoint addr_i_testbins addr[]=[0:15];
INPUT_DATA_gpio: coverpoint gpio_ctrl_testbins gpio[]=[0:15];
Enable_ADDRESS: cross Enable, INPUT_ADDRESS;
GPIO_Enable_ADDRESS: cross Enable_ADDRESS, INPUT_DATA_gpio;
endgroup

Specify property for assertions and for coverage
property wrt_ctrl;
@(negedge clk||we_i==1'b1)disable iff(!rst)
(addr_i[3:0]==3'h0|->(reg_ctrl==data_i));
endproperty

A1: assert property(wrt_ctrl);
property wrt_data;
@(negedge clk||we_i==1'b1)disable iff(!rst)
(addr_i[3:0]==3'h4|->(reg_data==data_i));
endproperty

A2: assert property(wrt_data);
property wrt_io_pin_0;
@(negedge clk||we_i==1'b0)disable iff(!rst)
(u3.gpio_ctrl[1:0]==2'b10|->(reg_data[0]==io_pin_i[0]));
endproperty

A3: assert property(wrt_io_pin_0);
property wrt_io_pin_1;
@(negedge clk||we_i==1'b0)disable iff(!rst)
(u3.gpio_ctrl[3:2]==2'b10|->(reg_data[1]==io_pin_i[1]));
endproperty

A4: assert property(wrt_io_pin_1);
property red_data_reg;
@(negedge clk||we_i==1'b0)disable iff(!rst)
(addr_i[3:0]==4'h4|->(data_o==u3.gpio_idata));
endproperty

A5: assert property(red_idata_ireg);
property red_ictrl_ireg;
@(negedge clk||we_i==1'b0)disable iff(!rst)
(addr_ii[3:0]==4'h0|->(data_io==u3.gpio_ictrl));
endproperty

A6: assert property(red_ictrl_ireg);

3.4.3 Register bank functional specification

- Register Bank Functional specification.
  - Clocked on a positive edge of clock
  - Active low reset
  - Number of registers in the register bank=32
  - Width of each register =32 bits
  - Register x0 is a read only register and has a value of 32'b00
  - Write enable (wren) asserted to write to the register and the register write address should not be 32'b0
  - All writes are sequential
  - All reads are combinational
- Register Bank Verification strategy.
  - Assert reset for 10ns to reset the DUT
  - Initialize both the register read addresses to zero
  - Assert the write enable (wren) signal
  - Randomize the write address and the data to be written in(wrreg) to thoroughly exercise the design
  - Randomize the read addresses of the register bank
  - Visual inspection of the waveform to verify the functional behaviour

3.4.4 Division unit verification strategy

Main goal of this verification environment is to test if the division unit yields the right results in exactly 32 clock cycles as specified in the functional specifications.

- Division unit verification specification.
  - Assert rst_n for 10ns
  - Assert division start flag(start_i) 10ns after reset has been de-asserted
  - Assign test values for dividend_i and divisor_i
  - Toggle opcode between ‘instdiv(Divison operation) and ‘instrem(Reminder operation)
  - Set the write register address(reg_waddr_i) to check if the same value appears on the output (reg_waddr_o)
  - Always block that checks when division ready flag is set and de-asserts the start flag

3.4.5 First 3 pipeline stages (if_id, id and id_ex) verification strategy

The main goal for this verification environment was to test if the instructions fetched by the fetch unit(if_id) is passed on to the decode unit (id) correctly and if the id unit is able to decode the register addresses and register data according to the instructions fetched and pass them as inputs to the execution stage.

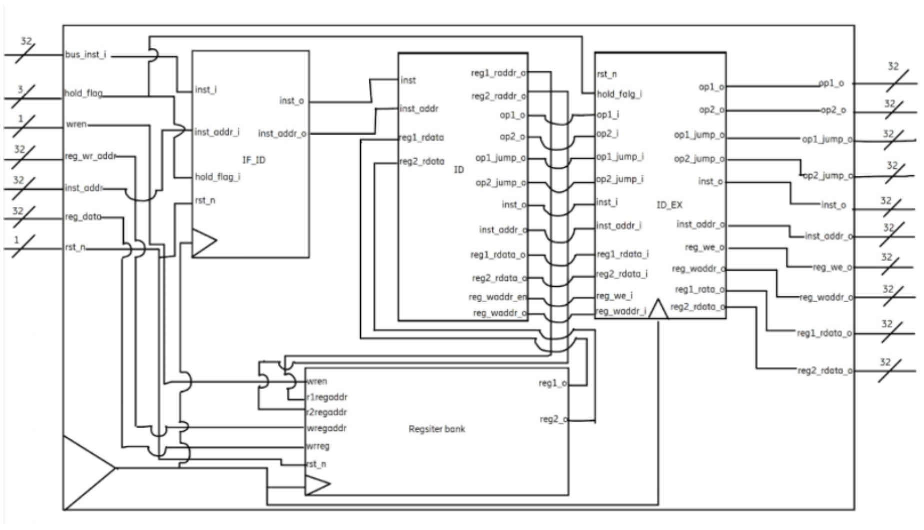


Figure 41: First three pipeline stages of the RISC-V core

- First 3 pipeline stage verification specification.
  - The DUT has to be modified to support additional ports as input and output so that the test bench can drive test_instructions and other test signals to the DUT and monitor it at the output ports. **The following are the additional input ports declared.**
    1. hold_flag
    2. wren
    3. reg_wr_addr
    4. reg_data

    The following are the additional output ports declared.
    1. op1_o
    2. op2_o
    3. op1_jump_o
    4. op2_jump_o
    5. inst_o
    6. inst_addr_o
  - Declare an array for storing instructions to be tested.
  - Assert rst_n for 10ns

  - After de-asserting rst_n set the register write enable(wren) high to write initial values to the registers.
  - Load a value of 2'h2 to the first 4 registers or can even randomize the register to be written into.
  - De-assert register write signal (wren)
  - Randomly fetch instructions from the instructions array and pass it as input to the DUT
  - Test for hold flag functionality with the following values 3'b001, 3'b010 and 3'b100 and probe the signals of the fetch unit to check the effect of these hold flags.

3.4.6 RISC-V core verification strategy:

The main goal for this verification environment is to test the whole data flow from the input to the output of the RISC-V core. This environment is not intended to verify the functionality of the core with bus or the other peripherals such as the ROM, RAM and the GPIO . The pc register needs to updated in a way that it increments in steps of 1 and not steps of 4 because of the temporary ROM in place.

- RISC-V core verification specification.
  - Since we don't have the instruction memory which passes the instruction to the if_id unit we will use a temporary ROM instead.
  - The ROM is to be loaded with instructions to load the registers with initial values first and then can be loaded with other instructions to be tested.
  - Assert the rst_n for 10ns and after de-asserting rst_n probe the bus_inst_i input of the if_id unit to check if the instructions are fetched according to the increments of the program counter.
  - Use a monitor to know what instruction is being executed at what clock cycle.

NOTE: Can not randomize instructions to the bus_inst_i input port because in case of a branch instructions. We need to fetch instructions according to the corresponding change in the program counter due to the branch and randomizing instructions simply hides this effect.

3.4.7 RISC-V Top verification plan(Directed test)

The main goal of this verification environment is to verify the entire data flow of the RISC-V design which includes the core integrated with the bus, the ROM, the RAM and the GPIO .

- RISC-V core verification specification.
  - The DUT clock port must be modified with a mux to select between the system clock (sys_clk) and the test clock(tst_clk) that is fed into the DUT.The tst_clk will be used for loading the ROM with the instructions. After loading the instructions the system clock will be turned on for the core.

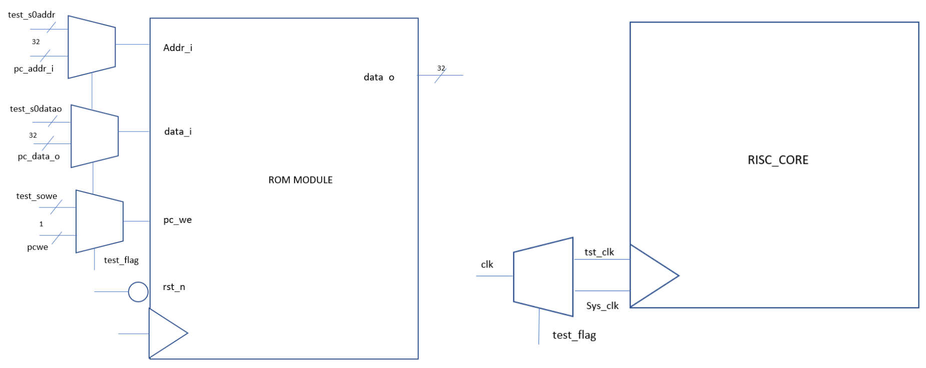


Figure 42: ROM and DUT module after the addition of testport muxes for verification purpose

  - ADD MUX TO ALL THE INPUTS OF THE ROM TO SELECT BETWEEN THE INITIAL_TEST PHASE WHERE ROM IS LOADED WITH ALL INSTRUCTIONS AND POST_INITIAL_TEST PHASE WHERE THE CORE TAKES OVER THE CONTROL SIGNALS OF THE ROM.
  - Define a temporary register with all the test instructions and use this register to load the ROM.
  - Assert the rst_n for 10ns.
  - After de-asserting the reset start the INITIAL_TEST phase where all the instructions from the temporary register will be loaded to the ROM.
  - In the POST_INITIAL_RUN phase Assert rst_n again for 10ns after ROM initialization is complete.
  - The initial_test_phase and the post_initial_test phase are selected by using the test_flag. Assert the test_flag for the initial_test phase and de-assert it for the post_initial_test phase.

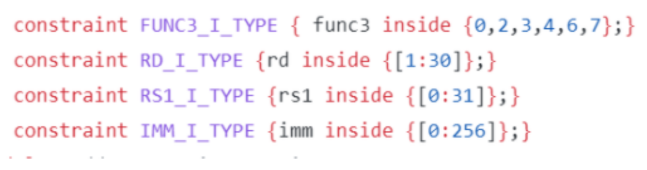
3.4.8 RISC-V Top verification plan(Randomized test)

The main goal of this verification environment is to verify the entire data flow of the RISC-V design which includes the core integrated with the bus, the ROM, the RAM and the GPIO . The difference between this test and the directed test is that in the directed test, the ROM is loaded with instructions that were targeting only a particular register, and integer values and also not all instructions of every type were tested. Where the randomization test covers a wider range of instructions, both in terms of registers and immediate values as well as type of instructions.

- RISC-V core verification specification.
  - The DUT clock port and the ROM ports needs to be modified the same way as that of directed test.

  - Define a separate class for every type of instructions. The class definitions could be like I_type_instructions, A_type_instructions, R_type_instructions, M_type_instructions, LSW_type_instruction, S_type_instructions.
  - Every class of instruction has it owns constraints to ensure the instructions are structured according to the ISA.
  - Initial_test phase:
    * Assert the rst_n for 10ns.
    * After de-asserting the reset start the initial_test phase where all the instructions from the temporary register will be loaded to the ROM by asserting the test_flag.
    * De-assert the test_flag after loading the instructions into the ROM.
  - Post_initial_test phase:
    * In the post_initial_test phase after ensuring the test_flag has been de-asserted, assert rst_n again for 10ns after ROM initialization is complete.

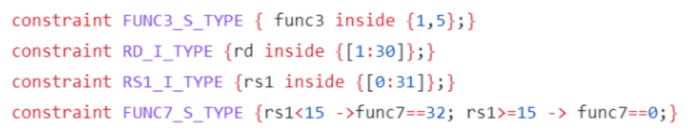
- Constraints for generating instructions according to the ISA
  - Class I_type instructions have the following constraints:



```
constraint FUNC3_I_TYPE { func3 inside {0,2,3,4,6,7};}
constraint RD_I_TYPE {rd inside {[1:30]};}
constraint RS1_I_TYPE {rs1 inside {[0:31]};}
constraint IMM_I_TYPE {imm inside {[0:256]};}
```

Figure 43: Constraints for I type instructions

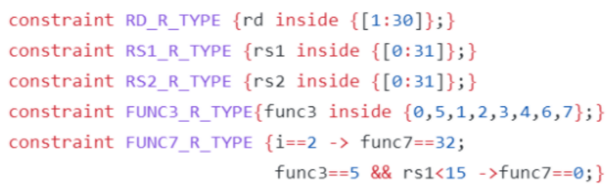
  - Class S_type instructions have the following constraints:



```
constraint FUNC3_S_TYPE { func3 inside {1,5};}
constraint RD_I_TYPE {rd inside {[1:30]};}
constraint RS1_I_TYPE {rs1 inside {[0:31]};}
constraint FUNC7_S_TYPE {rs1<15 ->func7==32; rs1>=15 -> func7==0;}
```

Figure 44: Constraints for S type instructions

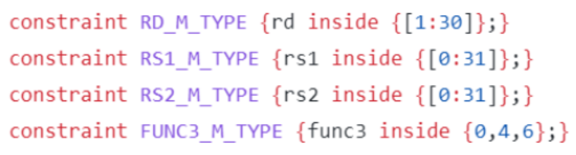
– Class R_type instructions have the following constraints:



```
constraint RD_R_TYPE {rd inside {[1:30]};}
constraint RS1_R_TYPE {rs1 inside {[0:31]};}
constraint RS2_R_TYPE {rs2 inside {[0:31]};}
constraint FUNC3_R_TYPE{func3 inside {0,5,1,2,3,4,6,7};}
constraint FUNC7_R_TYPE {i==2 -> func7==32;
                          func3==5 && rs1<15 ->func7==0;}
```

Figure 45: Constraints for R type instructions

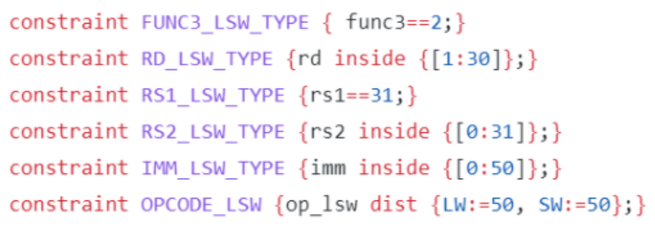
– Class M_type instructions have the following constraints:



```
constraint RD_M_TYPE {rd inside {[1:30]};}
constraint RS1_M_TYPE {rs1 inside {[0:31]};}
constraint RS2_M_TYPE {rs2 inside {[0:31]};}
constraint FUNC3_M_TYPE {func3 inside {0,4,6};}
```

Figure 46: Constraints for M type instructions

– Class LSW_type instructions have the following constraints:

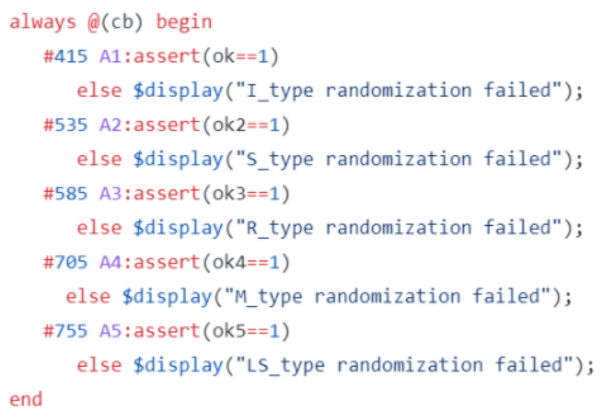


```
constraint FUNC3_LSW_TYPE { func3==2;}
constraint RD_LSW_TYPE {rd inside {[1:30]};}
constraint RS1_LSW_TYPE {rs1==31;}
constraint RS2_LSW_TYPE {rs2 inside {[0:31]};}
constraint IMM_LSW_TYPE {imm inside {[0:50]};}
constraint OPCODE_LSW {op_lsw dist {LW:=50, SW:=50};}
```

Figure 47: Constraints for LSW type instructions

- The first 31 instructions to be loaded into the ROM are the ADDI's to load initial values to the registers.
- Randomize the classes one after the other and frame the instruction to be loaded into the ROM according to the ISA .
- Define a monitor class to collect coverage for the rd, rs1, rs2, immediate_values and the ROM address written into.
- Always use a default clocking block for the test bench.

- Assertion to check if randomization passed or failed.
- The set of instructions generated through constrained randomization is shown in Table 4



```
always @(cb) begin
   #415 A1:assert(ok==1)
      else $display("I_type randomization failed");
   #535 A2:assert(ok2==1)
      else $display("S_type randomization failed");
   #585 A3:assert(ok3==1)
      else $display("R_type randomization failed");
   #705 A4:assert(ok4==1)
     else $display("M_type randomization failed");
   #755 A5:assert(ok5==1)
      else $display("LS_type randomization failed");
end
```

Figure 48: Assertions for randomization check

SLTIU X26,X25,86
XORI X19,X12,56
SLTI X4,X5,94
ADDI X29,X27,197
ANDI X14,X14,117
ORI X8,X7,78
ORI X20,X24,70
ADDI X15,X17,233
XORI X28,X4,117
SLTIU X21,X26,194
ANDI X3,X19,179
SLTI X7,X6,111
SRLI X6,X15,4
NOP **modify SLLI in id** *(i.e Did not implement this instruction for functional verification)*
SLLI X15,X31,4
SRLI X19,X22,4
NOP **modify SLLI in id** *(i.e Did not implement this instruction for functional verification)*
NOP **the inst SLT(fun7)** *(i.e Did not implement this instruction for functional verification)*
SRL X9,X25,X29
NOP **the inst SLT(fun7)** *(i.e Did not implement this instruction for functional verification)*
SLT X19,X31,X15
SRL X21,X6,X16
NOP **SLL actually (fun7)** *(i.e Undefined instruction according to the ISA)*
AND X23,X2,X2
NOP **the inst SLTU(fun7)** *(i.e Did not implement this instruction for functional verification)*
XOR X20,X28,X28
ADD X17,X9,X14
NOP **OR actually(fun7)** *(i.e Undefined instruction according to the ISA)*
AND X30,X15,X20
MUL X4,X27,X2
REM X1,X31,X13
DIV X2,X25,X30
REM X7,X29,X21
DIV X8,X10,X8
SW X31,X27,9
LW X16,X31,19
SW X31,X10,39
LW X5,X31,11
LW X17,X31,42

Table 4: Table of instructions generated through constrained randomization

3.4.9 RISC-V Top verification plan with GPIO test(directed test)

This verification environment intends to test the functionality of the GPIO . The only difference between this VE and the randomized test VE is the immediate value being used for the store instruction to test the GPIO are 0 and 4 and the instructions loaded in the ROM are not randomized.

- RISC-V core verification specification.
  - The DUT clock port and the ROM ports need to be modified in the same manner as used for the directed test.
  - The ROM is loaded with the add immediate instructions to load all the registers except R31 with a value corresponding to the register number (ex, Register 10 is loaded with a value of 10).
  - R31 is loaded with a value of 32'h00000002
  - The sequence of instructions used to test the GPIO are as follows:
    ADDI X31, X0, 2
    SLLI X31, X31, 30
    SW R5, R31, 0
    SW R3, R31, 4
  - The slli instructions shifts the contents of R31 by 30 bits to the left to result in a value 32'h02000000. This is the address to select the GPIO .
  - The first SW instruction is to write and read the control register of the GPIO .
  - The last SW instructions is to write and read the data register of the GPIO .

3.5 Functional Simulation

The functional verification for the RISC-V SoC was done both at block level and system-level.

- At the block level: Functional units such as ROM, RAM, GPIO , Register bank, pc register and the Division unit were verified as a stand-alone blocks according to the verification specification so as to ensure the RTL adhered according to the functional specification, especially with the division unit where the result has to computed in 32 clock cycle. The RISC-V core with the first 3 pipeline stages (if_id, id and the id_ex) was also verified according to the verification specification so as to make sure that the fetch unit fetched instructions according to the pc register's output and the id unit decoded the instructions to asserted the correct set of control signals for the ex unit and the register bank.
- The system-level functional verification for the RISC-V SoC done with two test bench architectures:
  - Directed test with all the instructions hard-coded according to the ISA , so the verifier did not have to decode the instructions being executed.
    (**NOTE:**This test was also used for verifying only the RISC-V core excluding the bus, the ROM, RAM and the GPIO)
  - Randomized test with the instructions generated through constrained randomization. The test was run with a random seed of 1 always to make it easy for the verifier to decode the instructions being executed.

3.5.1 Directed test

For this test the instructions were all hardcoded according to the ISA definitions. The first 31 instructions were ADDI's to load the register with initial values. The values loaded corresponded to the the register number.

- The first 31 instructions were ADDI's to load the register with initial values. The values loaded corresponded to the the register number.
- The next 9 instructions were the branch instructions to test if the pc register behaviour reflected the right branch decisions. Figure 49 below is the sequence of branch instructions executed and the arrows on the sides indicate expected output of the instructions when the branch is taken and the numbers represent the relative address of the instructions in the fake ROM created for the test bench:
- The next set of instructions following the branch are from the different classes of instructions as mentioned in the ISA such as ADD, OR, SLLI, DIV, REM and other instructions.
- The functionalities were verified through visual inspection of the waveform.

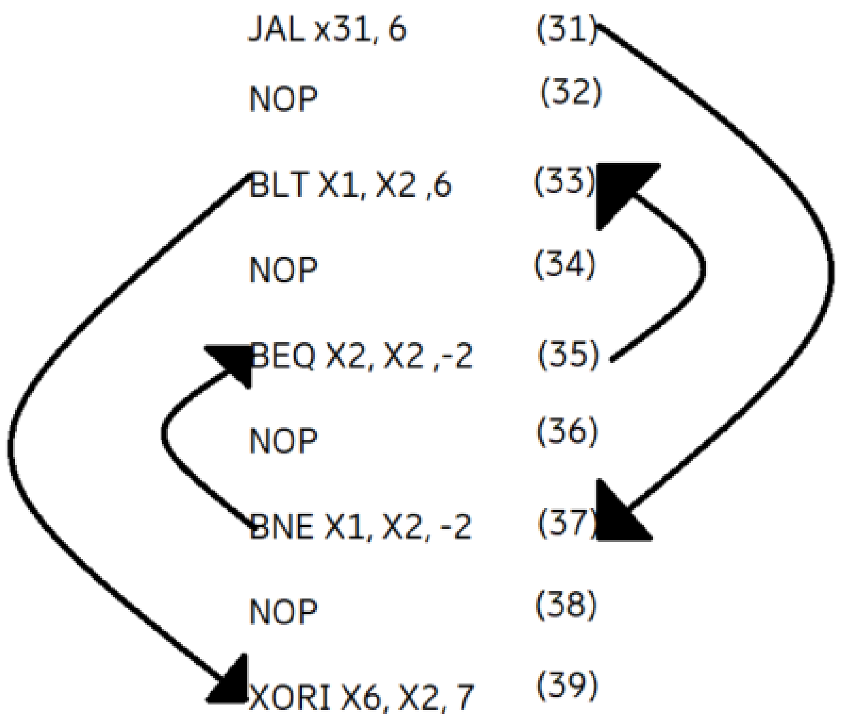


Figure 49: Branch instructions tested

3.5.2 Randomized test

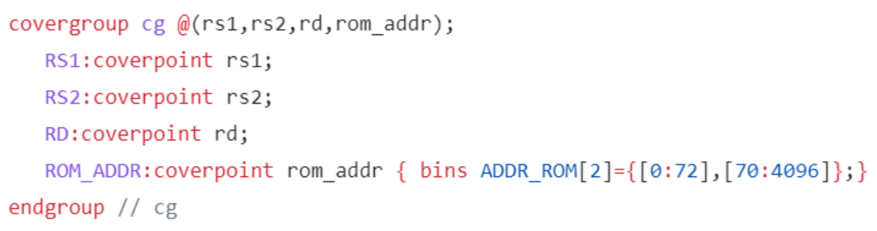
For this test the instructions were generated through constrained randomization in order to ensure all the instructions were according to the ISA definition.The constraints used can be seen in the randomized test section in the verification specification.

- The first 31 instructions were kept the same as the directed tests with ADDI's to load the register with initial values. The reason why the ADDI's were not randomized was to facilitate easier functional verification with other randomized instructions.
- The B-type and the J-Type instructions were not randomized and also the the instructions were generated through constrained randomization and then loaded into the ROM in order, this was to ensure we had better control over the test environment.
- Before the visual inspection of the waveform, the random instructions were first decoded in-order to know what results to expect from the RISC-V core. The decoded instructions are shown in the Table 4
- For verification sign off, the test environment was architected with a monitor for coverage. This is explained in the next section

3.5.3 Coverage test

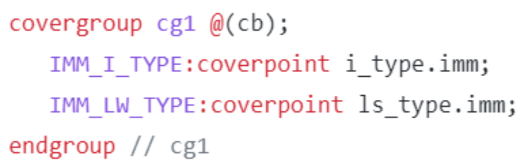
It was important to keep track of all the registers that have been exercised in the randomization test and also the immediate values that have exercised for the I-Type, load and the store instructions. In order to facilitate this, the randomized test bench were architected with a monitor that defined cover groups for the source registers(RS1,RS2), destination registers(RD), the ROM addresses exercised(ROM_ADDR) and also the immediate values.

Shown below are the cover groups used:



```
covergroup cg @(rs1,rs2,rd,rom_addr);
   RS1:coverpoint rs1;
   RS2:coverpoint rs2;
   RD:coverpoint rd;
   ROM_ADDR:coverpoint rom_addr { bins ADDR_ROM[2]={[0:72],[70:4096]};}
endgroup // cg
```

Figure 50: Cover groups for RS1,RS2,RD and ROM_ADDR



```
covergroup cg1 @(cb);
   IMM_I_TYPE:coverpoint i_type.imm;
   IMM_LW_TYPE:coverpoint ls_type.imm;
endgroup // cg1
```

Figure 51: Cover groups for the immediate values

Covergroup	Metric	Goal
/risc_top/monitor_risc/cg	60.9 %	100
Coverpoint cg::RS1	68.7 %	100
Coverpoint cg::RS2	53.1 %	100
Coverpoint cg::RD	71.8 %	100
Coverpoint cg::ROM_ADDR	50 %	100

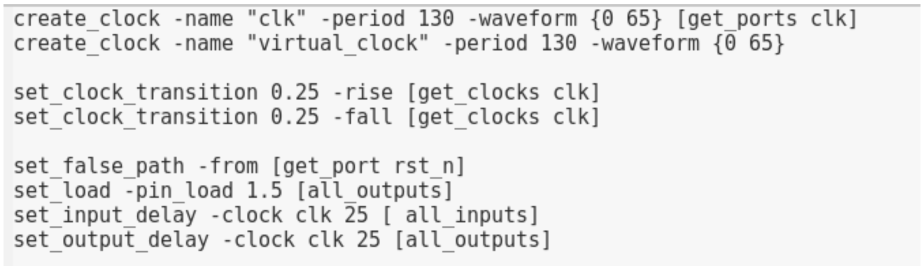
Table 5: Hit rate of the bins defined in the cover group

The table 5 shows the hit rate of the bins defined in the covergroup. For verification sign-off the expected hit rate of the source and destination register bins was expected to be more than or equal 50 %. This was because of the fact that constrained randomization used only sets of registers for generating few instructions types such as the R and the S type. On the other hand the ROM was about 4Kbytes and it was not possible to exercises all the locations for sign-off. Only the half of the memory was used for storing instructions. A detailed coverage report is shown in the appendix.

3.6 Synthesis

3.6.1 Synthesis as an ASIC

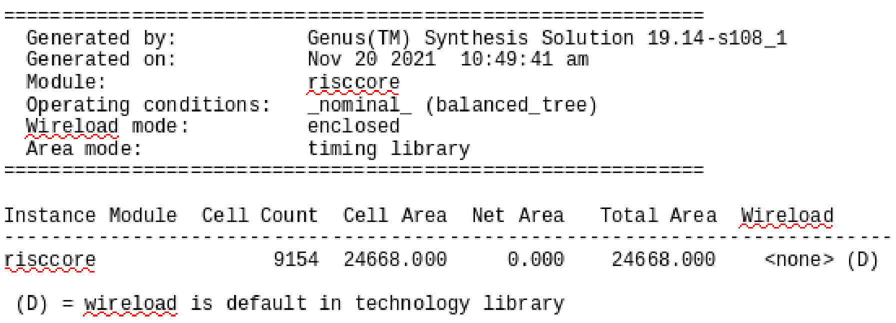
The RISC-V SoC was synthesized using Genus with LSI-10K as the target library to get the area estimates and the check for timing violations. The following was used as constraints for the synthesis:



```
create_clock -name "clk" -period 130 -waveform {0 65} [get_ports clk]
create_clock -name "virtual_clock" -period 130 -waveform {0 65}

set_clock_transition 0.25 -rise [get_clocks clk]
set_clock_transition 0.25 -fall [get_clocks clk]

set_false_path -from [get_port rst_n]
set_load -pin_load 1.5 [all_outputs]
set_input_delay -clock clk 25 [ all_inputs]
set_output_delay -clock clk 25 [all_outputs]
```

Figure 52: Constraints used for core synthesis

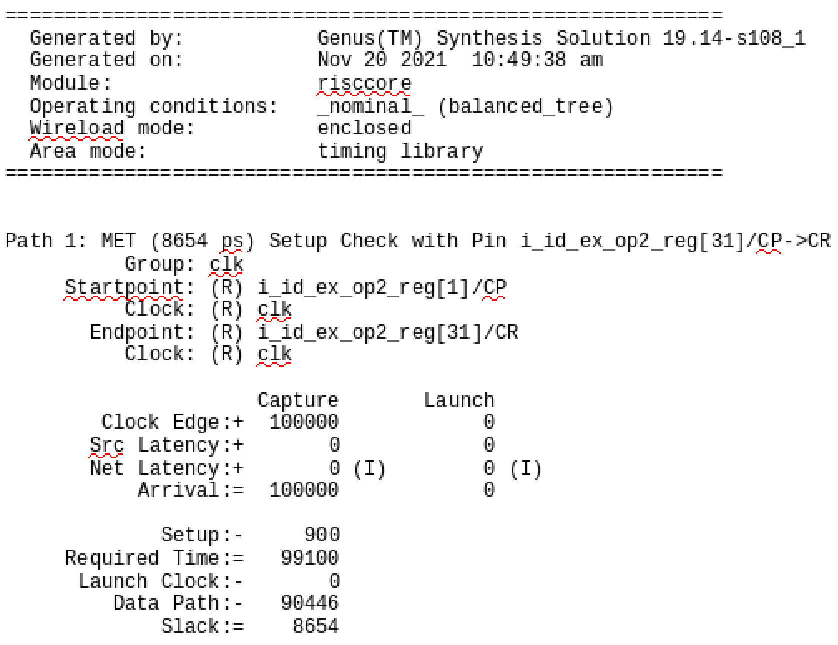
The cell count after synthesis was around 24668 gates.



```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 19.14-s108_1
  Generated on:           Nov 20 2021  10:49:41 am
  Module:                 risccore
  Operating conditions:   _nominal_ (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

Instance Module  Cell Count  Cell Area  Net Area   Total Area  Wireload
--------------------------------------------------------------------------
risccore               9154  24668.000     0.000    24668.000    <none> (D)

 (D) = wireload is default in technology library
```

Figure 53: Area report of core synthesis

The design had a positive slack of about 8ns: The design can run with a frequency of 10Mhz, it can also run with a frequency of less than 10Mhz of upto 7 Mhz but if we choose this clock frequency then we can't grantee the timing slack due to the wire-delays and pad delays when synthesized to silicon level with the target technology.

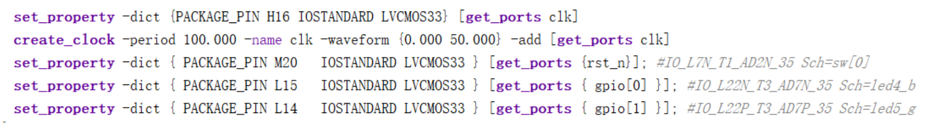


```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 19.14-s108_1
  Generated on:           Nov 20 2021  10:49:38 am
  Module:                 risccore
  Operating conditions:   _nominal_ (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================


Path 1: MET (8654 ps) Setup Check with Pin i_id_ex_op2_reg[31]/CP->CR
          Group: clk
     Startpoint: (R) i_id_ex_op2_reg[1]/CP
          Clock: (R) clk
       Endpoint: (R) i_id_ex_op2_reg[31]/CR
          Clock: (R) clk

                     Capture       Launch
        Clock Edge:+  100000            0
       Src Latency:+       0            0
       Net Latency:+       0 (I)        0 (I)
           Arrival:=  100000            0

             Setup:-     900
     Required Time:=   99100
      Launch Clock:-       0
         Data Path:-   90446
             Slack:=    8654
```

Figure 54: Timing report of core synthesis

3.6.2 Top Level Synthesis

The top level design was synthesized by Vivado. The LSI-10K was not used as the target library. The following was used as constraints for the synthesis:

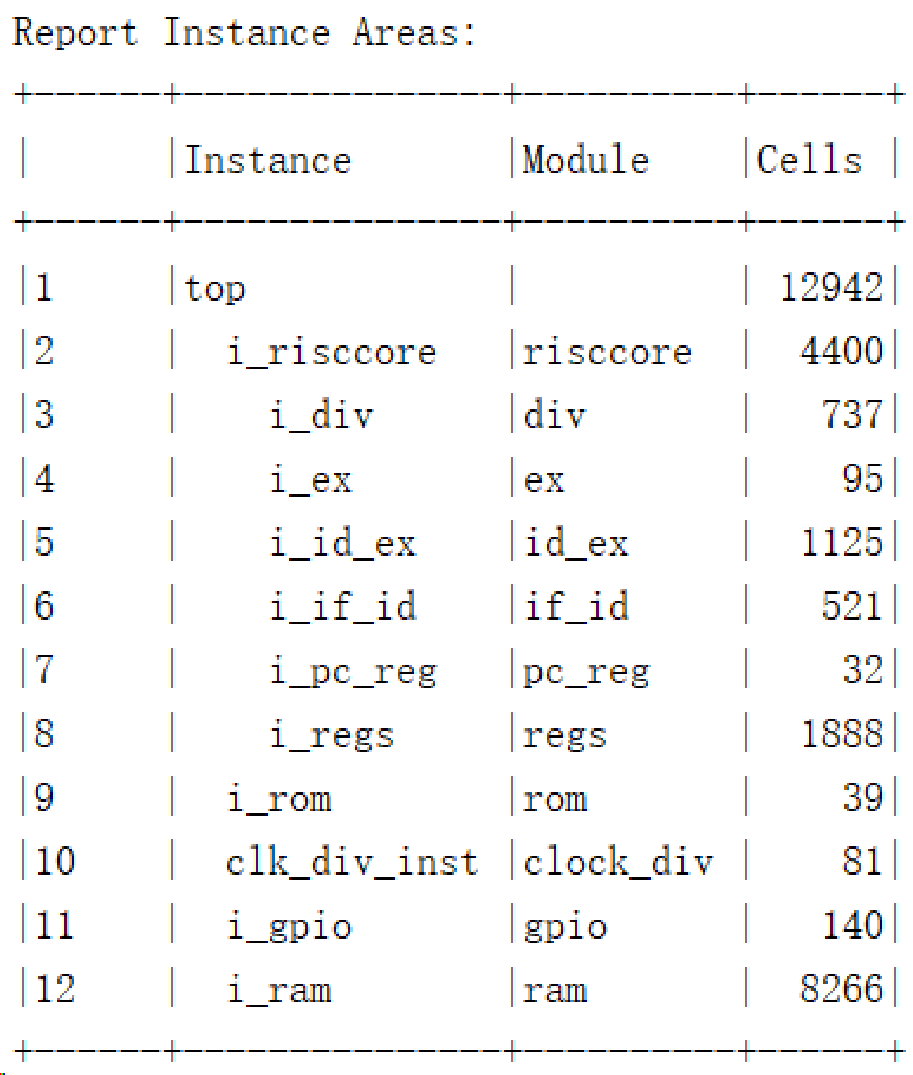


```
set_property -dict {PACKAGE_PIN H16 IOSTANDARD LVCMOS33} [get_ports clk]
create_clock -period 100.000 -name clk -waveform {0.000 50.000} -add [get_ports clk]
set_property -dict { PACKAGE_PIN M20   IOSTANDARD LVCMOS33 } [get_ports {rst_n}]; #IO_L7N_T1_AD2N_35 Sch=sw[0]
set_property -dict { PACKAGE_PIN L15   IOSTANDARD LVCMOS33 } [get_ports { gpio[0] }]; #IO_L22N_T3_AD7N_35 Sch=led4_b
set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 } [get_ports { gpio[1] }]; #IO_L22P_T3_AD7P_35 Sch=led5_g
```

Figure 55: Constraints used for top level synthesis

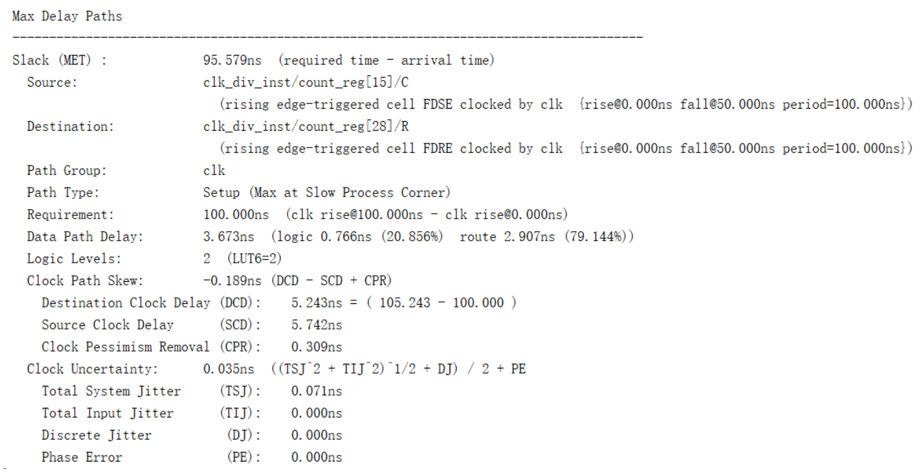
The cell count of top level design was around 12942 cells. The cell count of risccore was 4400 cells. The total is not equal to above and it seems unreasonable. The reason is that the applied library is different. In top level synthesis, each cells contains more gates than core synthesis.

The design had a positive slack of about 95ns:



```
Report Instance Areas:
+------+---------------+----------+------+
|      |Instance       |Module    |Cells |
+------+---------------+----------+------+
|1     |top            |          | 12942|
|2     |  i_risccore   |risccore  |  4400|
|3     |    i_div      |div       |   737|
|4     |    i_ex       |ex        |    95|
|5     |    i_id_ex    |id_ex     |  1125|
|6     |    i_if_id    |if_id     |   521|
|7     |    i_pc_reg   |pc_reg    |    32|
|8     |    i_regs     |regs      |  1888|
|9     |  i_rom        |rom       |    39|
|10    |  clk_div_inst |clock_div |    81|
|11    |  i_gpio       |gpio      |   140|
|12    |  i_ram        |ram       |  8266|
+------+---------------+----------+------+
```

Figure 56: Area report of top level synthesis



```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             95.579ns  (required time - arrival time)
  Source:                 clk_div_inst/count_reg[15]/C
                            (rising edge-triggered cell FDSE clocked by clk  {rise@0.000ns fall@50.000ns period=100.000ns})
  Destination:            clk_div_inst/count_reg[28]/R
                            (rising edge-triggered cell FDRE clocked by clk  {rise@0.000ns fall@50.000ns period=100.000ns})
  Path Group:             clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clk rise@100.000ns - clk rise@0.000ns)
  Data Path Delay:        3.673ns  (logic 0.766ns (20.856%)  route 2.907ns (79.144%))
  Logic Levels:           2  (LUT6=2)
  Clock Path Skew:        -0.189ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    5.243ns = ( 105.243 - 100.000 )
    Source Clock Delay      (SCD):    5.742ns
    Clock Pessimism Removal (CPR):    0.309ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Figure 57: Timing report of top level synthesis

3.7 Conclusion

The RISC-V SoC designed is a 32 bit core which supports the RV32I base instruction set with an extension to also support multiply, division and reminder instructions. The execution unit and the decode unit of the core can also be extended to support other instructions such as memory ordering instructions from the RV32I ISA .The core has as separate unit for computation of division and this is not a part of the execution unit. This design of this RISC has the ROM as a part of the core so ROM accesses do not have to go through the bus. The core can support up to 16 GPIOs. The core however when synthesized for FPGA will have to make use of block ROM and RAM and hence the ROM and the RAM module will have to be replaced with their corresponding FPGA block alternatives. The RISC-V core when synthesized as an ASIC with the LSI-10K technology has a gate of around 24,668 gates and a slack of about $8ns$ and can run with $10Mhz$ clock frequency. When synthesized as a FPGA the core has a gate count of 12942 cells and a slack of about $95ns$.

3.8 Future work

The only challenge the design team faced was when trying to synthesize the design for a FPGA application. The RAM and the ROM module was not being synthesized as block RAM and ROM but rather as registers which made it impossible to fit the design post synthesis on the PYNQ-Z1 FPGA. Synthesizing the design with a block memory module using the zylinx template for block RAM and ROM is a major task for future FPGA application.

In terms of the instruction set only the a part of the instruction set was implemented. Instruction set can be extended that support memory ordering and advanced signed arithmetic operations. Another major improvement could be to improve the clock rate. Currently the core works at 10Mhz clock rate, the design can further be optimized to improve this aspect. Area is another aspect that requires a lot of optimization. The design can also be further optimized to reduce the gate count to be able fit on the 7mm x 7mm chip area requirement. Since the core uses register mapping (register 31) to execute the load and the store instructions and the GPIO, another aspect that could be worked on is to use address mapping instead (i.e reserving address spaces to access the memories and the GPIO)

4 Conclusion

Both papers on the CPU and FPGA architectures have been successfully designed and tested, however, their performances are extremely limited. The proposed technology proved to large at the time being without further optimisation, but plausible to achieve with larger chip size or smaller technology.

The cell library design has been a failure. Unlike first expected, only the NAND and DFF cells are not enough for the software used to synthesise the designs. The FPGA design would notably require much more information than only timings to elaborate, and so the creation of a complete custom temporary library has been dropped and replaced with the used of the available, size equivalent LSI10K with edited custom time delays.

The FPGA design demonstrates feasible, but extremely lacking computing power. Less than a hundred CLBs of four inputs variables do not even comply with early 2000's standards. Therefore, the use of such PDK looks rather discouraged without more expertise in the design, or the restriction to very simple software. Yet the design introduced effectively works with a simple counter software example.

The RISC-V SoC design is a 32 bit core which supports the RV32I base instruction set with an extension to also support multiply, division and reminder instructions. The execution unit and the decode unit of the core can also be extended to support other instructions such as memory ordering instructions from the RV32I ISA .The core has as separate unit for computation of division and this is not a part of the execution unit. This design of this RISC has the ROM as a part of the core so ROM accesses do not have to go through the bus. The core can support up to 16 GPIOs. The core however when synthesized for FPGA will have to make use of block ROM and RAM and hence the ROM and the RAM module will have to be replaced with their corresponding FPGA block alternatives. The RISC-V core when synthesized as an ASIC with the LSI-10K technology has a gate of around 24,668 gates and a slack of about $8ns$ and can run with $10Mhz$ clock frequency. When synthesized as a FPGA the core has a gate count of 12942 cells and a slack of about $95ns$.

Would these very basic architectures be kept for further development stages, they now both require a compiler as to be able to run software.

4.1 Summarizing Future Work

The FPGA need to get a more rigorous verification with the aid of a architecture specific compiler that would also be able to generate a bitstream for designs to the FPGA allowing any software to run.

For the RISC-V cpu a template of zylinx block RAM and ROM is needed to synthesis a more complete cpu with correct timing. The current clock rate of 10MHz could also be improved with extension of the current implemented instructions, to support memory ordering and advance signed arithmetic's and use address mapping instead of the now implemented register mapping.

Both would also need improvements regarding area consumption, FPGA particular since the performance is very low(regarding the small number of CLBs). If the PDK would support pass gates inside SWBs nets, area saves up to 40% and by replacing the DFFs in the configuration bitstream to latches or pass gates for even more area savings. overall design optimization for the RISC-V could decrease the gate count to fit a 7mm x 7mm chip.

References

[1] Paul Adrien Maurice Dirac. *The Principles of Quantum Mechanics.* International series of monographs on physics. Clarendon Press, 1981. ISBN: 9780198520115.

[2] Andrew Waterman Krste Asanovic SiFive Inc. *RISC-V ISA Specification, Volume 1 Unprivileged Spec v.* URL: `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf`. (accessed: 16.01.2022).

[3] Xilinx. *Spartan II FPGA Family Data Sheet.* URL: `https://www.xilinx.com/support/documentation/data_sheets/ds001.pdf`. (accessed: 16.01.2022).

A Coverage report from off randomized verification environment

This section will show the coverage report that was extracted from the top level randomized verification environment of the RISC-V SoC.

Covergroup	Metric	Goal	Status
Coverpoint cg::RS1	68.7%	100	Uncovered
covered/total bins:	22	32	
missing/total bins:	10	32	
% Hit:	68.7%	100	
type_option.weight=1			
type_option.goal=100			
type_option.comment=			
bin auto[0]	3	1	Covered
bin auto[1]	0	1	ZERO
bin auto[2]	3	1	Covered
bin auto[3]	0	1	ZERO
bin auto[4]	5	1	Covered
bin auto[5]	2	1	Covered
bin auto[6]	7	1	Covered
bin auto[7]	4	1	Covered
bin auto[8]	0	1	ZERO
bin auto[9]	3	1	Covered
bin auto[10]	3	1	Covered
bin auto[11]	0	1	ZERO
bin auto[12]	2	1	Covered
bin auto[13]	3	1	Covered
bin auto[14]	2	1	Covered
bin auto[15]	5	1	Covered
bin auto[16]	0	1	ZERO
bin auto[17]	2	1	Covered
bin auto[18]	0	1	ZERO
bin auto[19]	2	1	Covered
bin auto[20]	0	1	ZERO
bin auto[21]	0	1	ZERO
bin auto[22]	2	1	Covered
bin auto[23]	0	1	ZERO
bin auto[24]	5	1	Covered
bin auto[25]	8	1	Covered
bin auto[26]	2	1	Covered
bin auto[27]	8	1	Covered
bin auto[28]	3	1	Covered
bin auto[29]	3	1	Covered
bin auto[30]	0	1	ZERO
bin auto[31]	14	1	Covered

Coverpoint cg::RS2	53.1%	100	Uncovered
covered/total bins:	17	32	
missing/total bins:	15	32	
% Hit:	53.1%	100	
type_option.weight=1			
type_option.goal=100			
type_option.comment=			
bin auto[0]	0	1	ZERO
bin auto[1]	0	1	ZERO
bin auto[2]	6	1	Covered
bin auto[3]	0	1	ZERO
bin auto[4]	0	1	ZERO
bin auto[5]	0	1	ZERO
bin auto[6]	0	1	ZERO
bin auto[7]	3	1	Covered
bin auto[8]	3	1	Covered
bin auto[9]	4	1	Covered
bin auto[10]	4	1	Covered
bin auto[11]	0	1	ZERO
bin auto[12]	0	1	ZERO
bin auto[13]	3	1	Covered
bin auto[14]	3	1	Covered
bin auto[15]	3	1	Covered
bin auto[16]	3	1	Covered
bin auto[17]	0	1	ZERO
bin auto[18]	0	1	ZERO
bin auto[19]	0	1	ZERO
bin auto[20]	3	1	Covered
bin auto[21]	7	1	Covered
bin auto[22]	1	1	Covered
bin auto[23]	0	1	ZERO
bin auto[24]	3	1	Covered
bin auto[25]	0	1	ZERO
bin auto[26]	0	1	ZERO
bin auto[27]	1	1	Covered
bin auto[28]	3	1	Covered
bin auto[29]	3	1	Covered
bin auto[30]	3	1	Covered
bin auto[31]	0	1	ZERO

Coverpoint cg::RD	71.8%	100	Uncovered
covered/total bins:	23	32	
missing/total bins:	9	32	
% Hit:	71.8%	100	
type_option.weight=1			
type_option.goal=100			
type_option.comment=			
bin auto[0]	0	1	ZERO
bin auto[1]	3	1	Covered
bin auto[2]	3	1	Covered
bin auto[3]	2	1	Covered
bin auto[4]	7	1	Covered
bin auto[5]	0	1	ZERO
bin auto[6]	2	1	Covered
bin auto[7]	5	1	Covered
bin auto[8]	9	1	Covered
bin auto[9]	3	1	Covered
bin auto[10]	0	1	ZERO
bin auto[11]	0	1	ZERO
bin auto[12]	3	1	Covered
bin auto[13]	0	1	ZERO
bin auto[14]	5	1	Covered
bin auto[15]	4	1	Covered
bin auto[16]	0	1	ZERO
bin auto[17]	3	1	Covered
bin auto[18]	3	1	Covered
bin auto[19]	7	1	Covered
bin auto[20]	8	1	Covered
bin auto[21]	5	1	Covered
bin auto[22]	3	1	Covered
bin auto[23]	3	1	Covered
bin auto[24]	3	1	Covered
bin auto[25]	0	1	ZERO
bin auto[26]	2	1	Covered
bin auto[27]	0	1	ZERO
bin auto[28]	2	1	Covered
bin auto[29]	2	1	Covered
bin auto[30]	3	1	Covered
bin auto[31]	0	1	ZERO

Coverpoint cg::ROM_ADDR	50.0%	100	Uncovered
covered/total bins:	1	2	
missing/total bins:	1	2	
% Hit:	50.0%	100	
type_option.weight=1			
type_option.goal=100			
type_option.comment=			
bin ADDR_ROM[0]	131	1	Covered
bin ADDR_ROM[1]	0	1	ZERO

```
CLASS monitor_risc
TYPE /risc_top/cg1                                  3.9%    100              Uncovered
covered/total bins:                                 5       128
missing/total bins:                                 123     128
% Hit:                                              3.9%    100
type_option.weight=1
type_option.goal=100
type_option.comment=
type_option.strobe=0
type_option.merge_instances=auto(0)
Coverpoint cg1::IMM_I_TYPE                          6.2%    100              Uncovered
covered/total bins:                                 4       64
missing/total bins:                                 60      64
% Hit:                                              6.2%    100
type_option.weight=1
type_option.goal=100
type_option.comment=
Coverpoint cg1::IMM_LW_TYPE                         1.5%    100              Uncovered
covered/total bins:                                 1       64
missing/total bins:                                 63      64
% Hit:                                              1.5%    100
type_option.weight=1
type_option.goal=100
type_option.comment=
Covergroup instance \/risc_top/cover_group_handle1
3.9%                                                100     Uncovered
covered/total bins:                                 5       128
missing/total bins:                                 123     128
% Hit:                                              3.9%    100
option.name=\/risc_top/cover_group_handle1
option.weight=1
option.goal=100
option.comment=
option.at_least=1
option.auto_bin_max=64
option.cross_num_print_missing=0
option.detect_overlap=0
option.per_instance=0
option.get_inst_coverage=0
```

Coverpoint IMM_I_TYPE	6.2%	100	Uncovered
covered/total bins:	4	64	
missing/total bins:	60	64	
% Hit:	6.2%	100	
option.weight=1			
option.goal=100			
option.comment=			
option.at_least=1			
option.auto_bin_max=64			
option.detect_overlap=0			
bin auto[0:63]	1	1	Covered
bin auto[64:127]	34	1	Covered
bin auto[128:191]	1	1	Covered
bin auto[192:255]	3	1	Covered
bin auto[256:319]	0	1	ZERO
bin auto[320:383]	0	1	ZERO
bin auto[384:447]	0	1	ZERO
bin auto[448:511]	0	1	ZERO
bin auto[512:575]	0	1	ZERO
bin auto[576:639]	0	1	ZERO
bin auto[640:703]	0	1	ZERO
bin auto[704:767]	0	1	ZERO
bin auto[768:831]	0	1	ZERO
bin auto[832:895]	0	1	ZERO
bin auto[896:959]	0	1	ZERO
bin auto[960:1023]	0	1	ZERO
bin auto[1024:1087]	0	1	ZERO
bin auto[1088:1151]	0	1	ZERO
bin auto[1152:1215]	0	1	ZERO
bin auto[1216:1279]	0	1	ZERO
bin auto[1280:1343]	0	1	ZERO
bin auto[1344:1407]	0	1	ZERO
bin auto[1408:1471]	0	1	ZERO
bin auto[1472:1535]	0	1	ZERO
bin auto[1536:1599]	0	1	ZERO
bin auto[1600:1663]	0	1	ZERO
bin auto[1664:1727]	0	1	ZERO
bin auto[1728:1791]	0	1	ZERO
bin auto[1792:1855]	0	1	ZERO

bin auto[1856:1919]	0	1	ZERO
bin auto[1920:1983]	0	1	ZERO
bin auto[1984:2047]	0	1	ZERO
bin auto[2048:2111]	0	1	ZERO
bin auto[2112:2175]	0	1	ZERO
bin auto[2176:2239]	0	1	ZERO
bin auto[2240:2303]	0	1	ZERO
bin auto[2304:2367]	0	1	ZERO
bin auto[2368:2431]	0	1	ZERO
bin auto[2432:2495]	0	1	ZERO
bin auto[2496:2559]	0	1	ZERO
bin auto[2560:2623]	0	1	ZERO
bin auto[2624:2687]	0	1	ZERO
bin auto[2688:2751]	0	1	ZERO
bin auto[2752:2815]	0	1	ZERO
bin auto[2816:2879]	0	1	ZERO
bin auto[2880:2943]	0	1	ZERO
bin auto[2944:3007]	0	1	ZERO
bin auto[3008:3071]	0	1	ZERO
bin auto[3072:3135]	0	1	ZERO
bin auto[3136:3199]	0	1	ZERO
bin auto[3200:3263]	0	1	ZERO
bin auto[3264:3327]	0	1	ZERO
bin auto[3328:3391]	0	1	ZERO
bin auto[3392:3455]	0	1	ZERO
bin auto[3456:3519]	0	1	ZERO
bin auto[3520:3583]	0	1	ZERO
bin auto[3584:3647]	0	1	ZERO
bin auto[3648:3711]	0	1	ZERO
bin auto[3712:3775]	0	1	ZERO
bin auto[3776:3839]	0	1	ZERO
bin auto[3840:3903]	0	1	ZERO
bin auto[3904:3967]	0	1	ZERO
bin auto[3968:4031]	0	1	ZERO
bin auto[4032:4095]	0	1	ZERO

Coverpoint IMM_LW_TYPE	1.5%	100	Uncovered
covered/total bins:	1	64	
missing/total bins:	63	64	
% Hit:	1.5%	100	
option.weight=1			
option.goal=100			
option.comment=			
option.at_least=1			
option.auto_bin_max=64			
option.detect_overlap=0			
bin auto[0:63]	5	1	Covered
bin auto[64:127]	0	1	ZERO
bin auto[128:191]	0	1	ZERO
bin auto[192:255]	0	1	ZERO
bin auto[256:319]	0	1	ZERO
bin auto[320:383]	0	1	ZERO
bin auto[384:447]	0	1	ZERO
bin auto[448:511]	0	1	ZERO
bin auto[512:575]	0	1	ZERO
bin auto[576:639]	0	1	ZERO
bin auto[640:703]	0	1	ZERO
bin auto[704:767]	0	1	ZERO
bin auto[768:831]	0	1	ZERO
bin auto[832:895]	0	1	ZERO
bin auto[896:959]	0	1	ZERO
bin auto[960:1023]	0	1	ZERO
bin auto[1024:1087]	0	1	ZERO
bin auto[1088:1151]	0	1	ZERO
bin auto[1152:1215]	0	1	ZERO
bin auto[1216:1279]	0	1	ZERO
bin auto[1280:1343]	0	1	ZERO
bin auto[1344:1407]	0	1	ZERO
bin auto[1408:1471]	0	1	ZERO
bin auto[1472:1535]	0	1	ZERO
bin auto[1536:1599]	0	1	ZERO
bin auto[1600:1663]	0	1	ZERO
bin auto[1664:1727]	0	1	ZERO
bin auto[1728:1791]	0	1	ZERO
bin auto[1792:1855]	0	1	ZERO
bin auto[1856:1919]	0	1	ZERO

bin auto[1920:1983]	0	1	ZERO
bin auto[1984:2047]	0	1	ZERO
bin auto[2048:2111]	0	1	ZERO
bin auto[2112:2175]	0	1	ZERO
bin auto[2176:2239]	0	1	ZERO
bin auto[2240:2303]	0	1	ZERO
bin auto[2304:2367]	0	1	ZERO
bin auto[2368:2431]	0	1	ZERO
bin auto[2432:2495]	0	1	ZERO
bin auto[2496:2559]	0	1	ZERO
bin auto[2560:2623]	0	1	ZERO
bin auto[2624:2687]	0	1	ZERO
bin auto[2688:2751]	0	1	ZERO
bin auto[2752:2815]	0	1	ZERO
bin auto[2816:2879]	0	1	ZERO
bin auto[2880:2943]	0	1	ZERO
bin auto[2944:3007]	0	1	ZERO
bin auto[3008:3071]	0	1	ZERO
bin auto[3072:3135]	0	1	ZERO
bin auto[3136:3199]	0	1	ZERO
bin auto[3200:3263]	0	1	ZERO
bin auto[3264:3327]	0	1	ZERO
bin auto[3328:3391]	0	1	ZERO
bin auto[3392:3455]	0	1	ZERO
bin auto[3456:3519]	0	1	ZERO
bin auto[3520:3583]	0	1	ZERO
bin auto[3584:3647]	0	1	ZERO
bin auto[3648:3711]	0	1	ZERO
bin auto[3712:3775]	0	1	ZERO
bin auto[3776:3839]	0	1	ZERO
bin auto[3840:3903]	0	1	ZERO
bin auto[3904:3967]	0	1	ZERO
bin auto[3968:4031]	0	1	ZERO
bin auto[4032:4095]	0	1	ZERO