Federal University of Santa Catarina – Science and Technology Centre

Technical Report – “Design and implementation in VHDL for FPGAs of a multi-cycle RISC-V based architecture”

EEL 510389 – Digital Systems and Reconfigurable Devices

Prof. Eduardo Augusto Bezerra

Student: Vinícius Pimenta Bernardo

2020

1. Introduction

The goal of this paper is to present the main characteristics of the proposed architecture that was developed for the final project of the Digital Systems and Reconfigurable Devices subject. It was described using VHSIC Hardware Description Language (VHDL) and implements the base integer instruction set, 32-bit wide (RV23I). It is also 5-staged pipelined featuring a Hazard Detection Unit and a Forwarding Unit.

The full diagram of the architecture can be found on attachment 1.

2. Development

For this project, Quartus Lite (v19.0) was the main software used to design and synthetize the hardware, alongside with ModelSim for simulation, using VHSIC Hardware Description Language (VHDL). Intel Quartus is a fully fledged programmable logic device software produced by Intel, which enables analysis and synthesis of HDL designs. The one being used is the Lite version, which is free, and the chosen target board was the Cyclone IV GX FPGA. VHDL stands for Very High-Speed Integrated Circuit Hardware Description Language and is commonly used to describe digital and mixed-signal systems such as field-programmable gate arrays (FPGAs), which is the focus of this project, and integrated circuits.

The microarchitecture is strongly inspired by David Patterson's and John Hennessy's Computer Organization and Design RISC-V Edition and the “maestro” core available in the riscv.org page. The project does not aim to be competitive and was developed for learning and academic purposes only. The instructions of the implemented ISA are divided in six core instruction formats, as shown in the table below:

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	**funct7**	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
I	**imm[11:0]**		**rs1**	**funct3**	**rd**	**opcode**
S	**imm[11:5]**	**rs2**	**rs1**	**funct3**	**imm[4:0]**	**opcode**
SB	**imm[12:|10:5]**	**rs2**	**rs1**	**funct3**	**imm[4:1|11]**	**opcode**
U	**imm[31:12]**				**rd**	**opcode**
UJ	**imm[20|10:1|11|19:12]**				**rd**	**opcode**

Table 1 Instruction Types

These formats have the intent of keeping the fields in the same position as much as possible. The list of implemented instruction is as follows:

- R-type:
  - Addition – add rd rs1 rs2.
  - Subtraction – sub rd rs1 rs2.
  - Logical and – and rd rs1 rs2.
  - Logical or – or rd rs1 rs2.
  - Logical xor – xor rd rs1 rs2;
  - Shift left logical – sll rd rs1 rs2.
  - Shift right logical – srl rd rs1 rs2.
  - Shift less than – slt rd rs1 rs2.
- I-type:
  - Load byte – lb rd offset(rs1).

  - Load word – lw rd offset(rs1).
  - Addition immediate – addi rd rs1 imm.
  - Logical and immediate – andi rd rs1 imm.
  - Logical or immediate – ori rd rs1 imm.
  - Logical xor immediate – xori rd rs1 imm.
  - Jump and link register – jalr rd rs1 offset.
- S-type:
  - Store byte – sb rd offset(rs1).
  - Store word – sw rd offset(rs1).
- SB-type:
  - Branch on equal – beq rs1 rs2 offset.
  - Branch on not equal – bne rs1 rs2 offset.
  - Branch less than – blt rs1 rs2 offset.
  - Branche greater than equal – bge rs1 rs2 offset.
- U-type:
  - Load immediate – lui rd imm.
- UJ-type:
  - Jump and link – jal rd offset.

The core is divided in four main components: a Control Unit, a Datapath, a Forwarding Unit, and a Hazard Detection Unit. There are a total of five stages, that are connected in the following order: Instruction Fetch Stage (IF), Instruction Decode Stage (ID), Execute Stage (EX), Memory Access Stage (MEM) and Write Back Stage (WB). The main components overlap and are within these stages. The Datapath goes from the Instruction Fetch all the way to the Write Back. The Control and Hazard Detection Units are within the Instruction Decode Stage, and the Forwarding Unit stays at the Execute Stage.

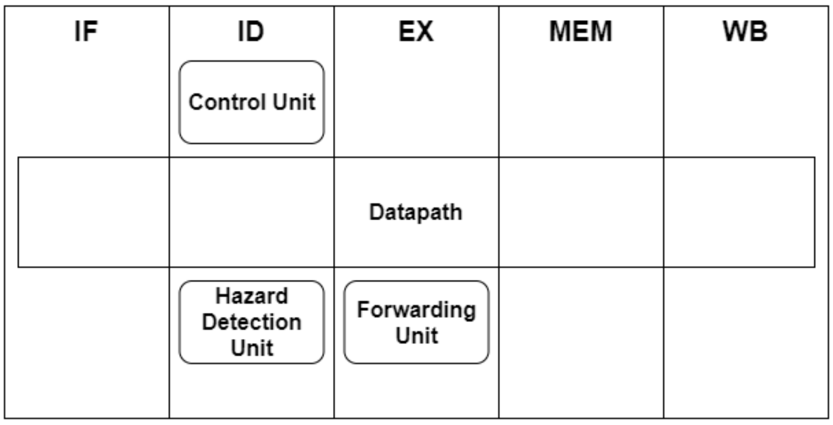


Figure 1 Components and Stages representation

The instructions reside in the Instruction Memory. In this stage, a 32-bit instruction is fetched from this memory. The Program Counter (PC) is a register that holds the address that is presented to the instruction memory. The next PC address is calculated by either incrementing by four or as a result of a branch or jump instruction.

All instructions have at most two register inputs. In the Instruction Decode stage, the index of these register are identified and presented to the register memory as addresses. In this stage, there is appropriate logic to determine if the instruction is ready to execute. If not, both Instruction Fetch and Decode stages may stall. If the instruction decoded is a branch or jump, the target address of the branch or jump is computed in parallel with reading the register file.

The Execute stage is where the actual computation occurs, on an ALU. Instructions can take one or more cycles to execute depending on the type of operation: Register-Register Operations takes one cycle; Memory Reference takes two cycles, as the ALU adds a register and an offset to produce an address by the end of the cycle; Multi-Cycle Instructions, such as multiplication, which takes more than two cycles and will not be implemented in this project.

If data memory needs to be accessed, it is done in the Memory Access stage, otherwise, in single cycles instructions, the result from the previous stage is forwarded to the next one.

During the Write Back stage, both single cycle and two cycle instructions write their results into the register file. This can happen while another stage – the instruction decode – is accessing the source registers in the register file for another instruction as well.

Sometimes, two instructions might try to use the same resource (structural hazard), or an instruction might try to use data that is not yet available (data hazard). A Hazard Detection Unit and a Forwarding Unit is used to avoid these types of situations.

For the implementation of the project, the idea was to make it as well divided in small and general components as possible. For this reason, components like registers, adders, and multiplexers could be widely replicated along major components.

a. Control Unit

The Control Unit (controller.vhd) takes as input the instruction from the decode stage and returns to the datapath all the signals necessary for its correct behavior. The logic behind the control unit is to decode the opcode and classify it based on instruction clusters (R-type, I-type, S-type, SB-type, U-type or UJ-type) and then in the instruction itself.

These control signals can be grounded in case of a flush, acting as a sel for a 16-bits 2 inputs multiplexer (mux16_2_1.vhd), originated from the Hazard or Jump Unit.

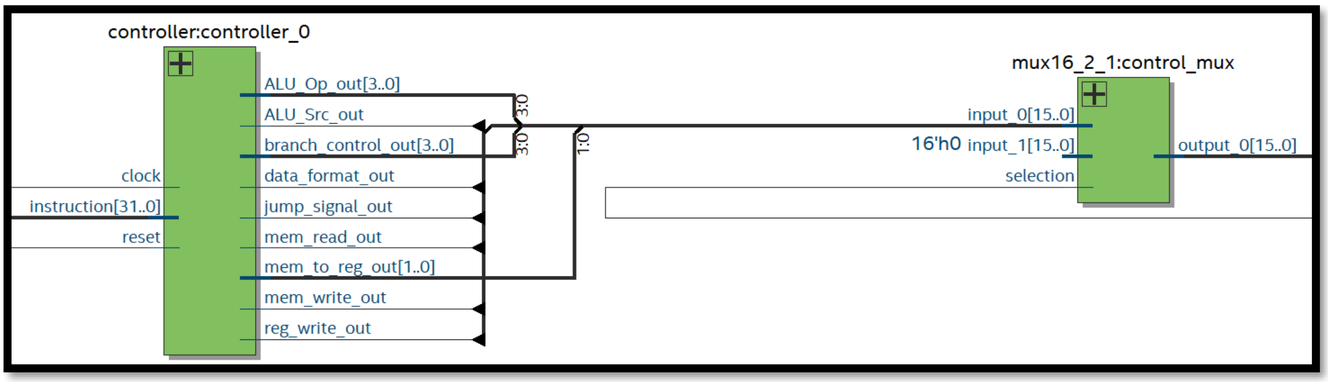


Figure 2 Control Unit diagram

b. Hazard Detection Unit

The hazard detection unit take as inputs the IF/ID.RS1, IF/ID.RS2, ID/EX.MemRead and ID/EX.RD, all coming from the datapath, and outputs a flush signal. If any of the register source address from ID stage is the same as register destination from EX stage, while Memory Read control signal is high on the EX stage, a flush is called. This is to avoid structural hazards.

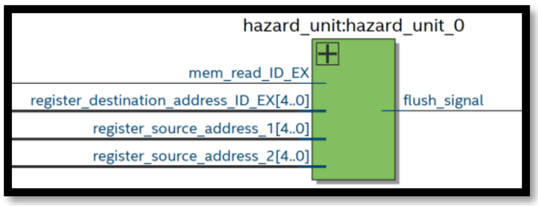

hazard_unit:hazard_unit_0
mem_read_ID_EX
register_destination_address_ID_EX[4..0]
register_source_address_1[4..0]
register_source_address_2[4..0]
flush_signal

Figure 3 Hazard Detection Unit diagram

c. Forwarding Unit

The forwarding unit take as inputs the ID/EX.RS1, ID/EX.RS2, EX/MEM.MemToReg, MEM/WB.MemToReg, EX/MEM.RD and MEM/WB.RD, all coming from the datapath, and outputs the two forwarding signals. The data that can be forwarded are the ALU output and Data Memory output from either MEM or WB stages.

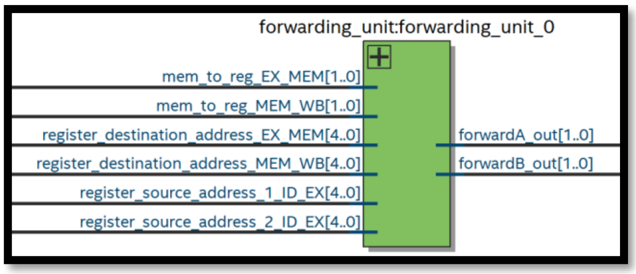

forwarding_unit:forwarding_unit_0
mem_to_reg_EX_MEM[1..0]
mem_to_reg_MEM_WB[1..0]
register_destination_address_EX_MEM[4..0]
register_destination_address_MEM_WB[4..0]
register_source_address_1_ID_EX[4..0]
register_source_address_2_ID_EX[4..0]
forwardA_out[1..0]
forwardB_out[1..0]

Figure 4 Forwarding Unit diagram

d. Datapath

The datapath is composed of many basic components, such as registers, adders, and multiplexers, along with some specific blocks (Program counter, stage division registers, Instruction Memory, Instruction Decoder, Jump Target Unit, Register File, ALU and Data

Memory), which can also have basic components inside. It takes as inputs all the control, forwarding, and flush signals, and outputs pipeline information to these components.

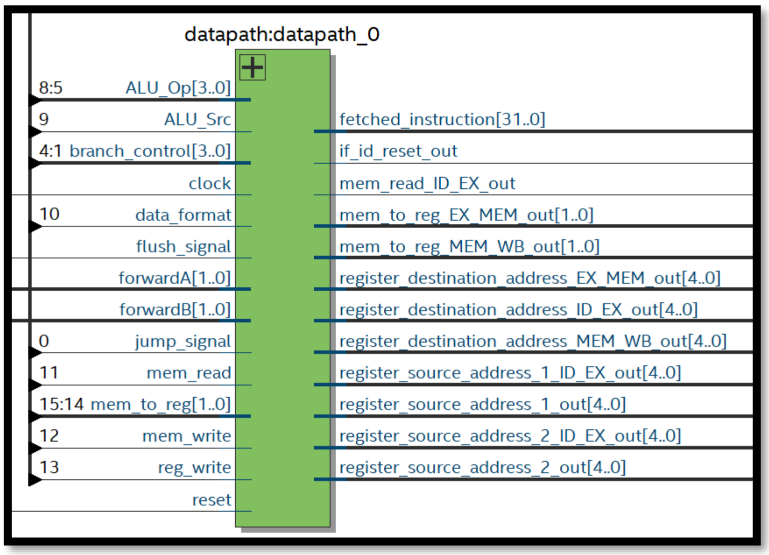

datapath:datapath_0
8:5
ALU_Op[3..0]
9
ALU_Src
4:1 branch_control[3..0]
clock
10
data_format
flush_signal
forwardA[1..0]
forwardB[1..0]
0
jump_signal
11
mem_read
15:14 mem_to_reg[1..0]
12
mem_write
13
reg_write
reset
fetched_instruction[31..0]
if_id_reset_out
mem_read_ID_EX_out
mem_to_reg_EX_MEM_out[1..0]
mem_to_reg_MEM_WB_out[1..0]
register_destination_address_EX_MEM_out[4..0]
register_destination_address_ID_EX_out[4..0]
register_destination_address_MEM_WB_out[4..0]
register_source_address_1_ID_EX_out[4..0]
register_source_address_1_out[4..0]
register_source_address_2_ID_EX_out[4..0]
register_source_address_2_out[4..0]

Figure 5 Datapath diagram

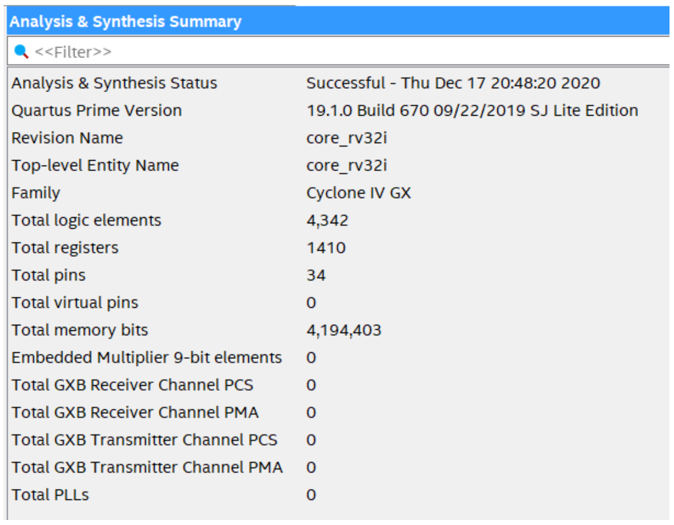
The program counter and the stage dividers are composed by groups of registers of different sizes, as shown on the table below.

Program counter		IF/ID		ID/EX		EX/MEM		MEM/WB	
instruction address	32 bits	instruction address	32 bits	instruction address	32 bits	instruction address	32 bits	instruction address	32 bits
		instruction	32 bits	register destination address	5 bits	register destination address	5 bits	register destination address	5 bits
				register source address 1	5 bits	ALU_Result	32 bits	ALU_Result	32 bits
				register source address 2	5 bits	Memory output data	32 bits	Memory output data	32 bits
				register source data 1	32 bits	mem to reg	2 bits	mem to reg	2 bits
				register source data 2	32 bits	mem read	1 bit	reg write	1 bit
				immediate	32 bits	mem write	1 bit		
				Alu_Op	4 bits	data format	1 bit		
				Alu_Src	1 bit	reg write	1 bit		
				mem to reg	2 bits				
				mem read	1 bit				
				mem write	1 bit				
				data format	1 bit				
				reg write	1 bit				

Table 2 Pipeline dividers sizes.

3. Simulation

After completing the design, it was synthesized using Quartus Lite, the analysis summary can be accessed, along with the RTL netlist.



Analysis & Synthesis Summary

<<Filter>>

Analysis & Synthesis Status	Successful - Thu Dec 17 20:48:20 2020
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	core_rv32i
Top-level Entity Name	core_rv32i
Family	Cyclone IV GX
Total logic elements	4,342
Total registers	1410
Total pins	34
Total virtual pins	0
Total memory bits	4,194,403
Embedded Multiplier 9-bit elements	0
Total GXB Receiver Channel PCS	0
Total GXB Receiver Channel PMA	0
Total GXB Transmitter Channel PCS	0
Total GXB Transmitter Channel PMA	0
Total PLLs	0

Figure 6 Quartus Lite Analysis & Synthesis Summary

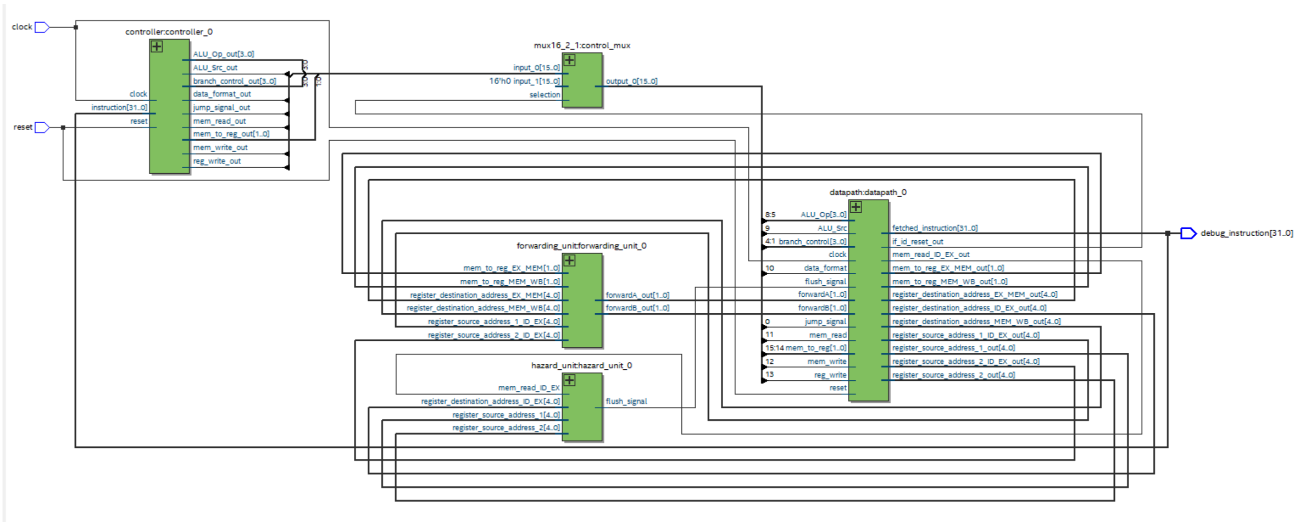


Figure 7 Quartus Lite RTL Viewer

ModelSim was used to simulate and verify the proposed architecture. ModelSim is a multi-language environment for simulation of hardware description languages. Simulations are performed using a graphical user interface and can be used independently or in conjunction with Intel Quartus and other software.

For this matter, a test bench VHDL file that generates a 50ns clock cycle (core_rv32i.vhd) was used to trigger the core. Instruction memory can be written by editing the memory initialization file (progmem.mif) with the binary generated from Ripes, that will be read by a Wizard-Generated vhd file (progmem.vhd). (Note: first address won't be executed and must be skipped, using a nop).

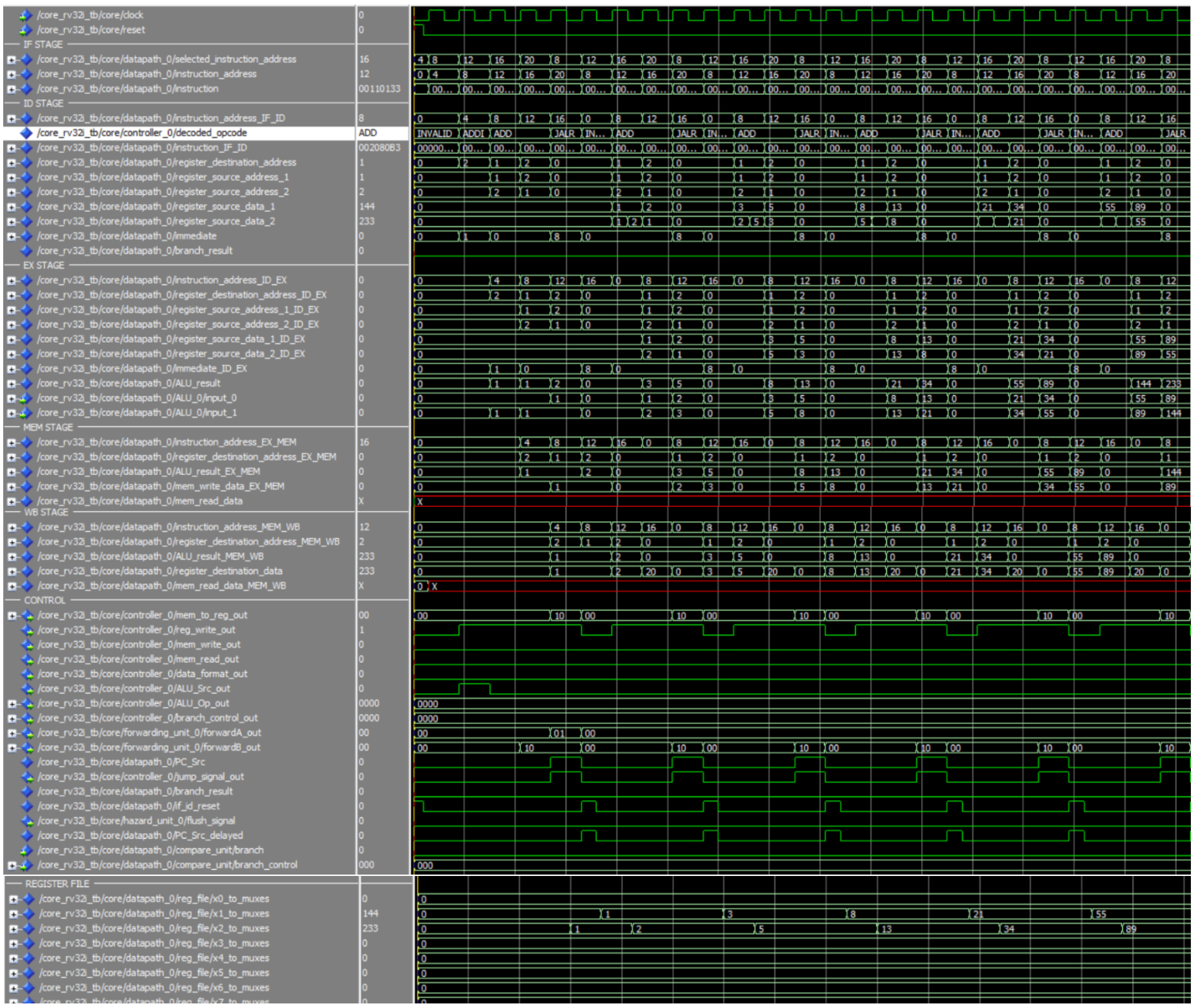
The following program was written in assembly and used to test each implemented instruction type:

```
#Test Program
nop
lui x1 1
addi x2 x0 1
xori x3 x2 1
xori x4 x2 0
ori x5 x0 1
ori x6 x0 0
andi x7 x5 1
andi x8 x6 1
add x9 x1 x2
sub x10 x1 x9
and x11 x9 x1
or x12 x9 x2
xor x13 x1 x10
sll x14 x5 x7
srl x15 x1 x14
slt x16 x1 x9
slt x17 x9 x1
addi x28 x0 0x00c0
addi x28 x28 0x000c
sw x28 0x40(x0)
nop
lw x27 0x40(x0)
addi x18 x0 5
addi x19 x0 0
a: addi x19 x19 1
bne x18 x19 a
b: sub x19 x19 x18
beq x19 x18 b
addi x20 x0 2
addi x21 x0 1
sub x20 x20 x2
jal x23 e
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
e: addi x24 x0 10
jalr x25 x23 0
```

Also, a simple Fibonacci program was implemented:

```
nop
addi x2, x0, 1
add x1, x1, x2
add x2, x2, x1
jalr x0, x0, 8
```

Using de ModelSim wave viewer, it’s possible see the signals adding up with time.


/core_rv32i_tb/core/clock
/core_rv32i_tb/core/reset
IF STAGE
/core_rv32i_tb/core/datapath_0/selected_instruction_address
/core_rv32i_tb/core/datapath_0/instruction_address
/core_rv32i_tb/core/datapath_0/instruction
ID STAGE
/core_rv32i_tb/core/datapath_0/instruction_address_IF_ID
/core_rv32i_tb/core/controller_0/decoded_opcode
/core_rv32i_tb/core/datapath_0/instruction_IF_ID
/core_rv32i_tb/core/datapath_0/register_destination_address
/core_rv32i_tb/core/datapath_0/register_source_address_1
/core_rv32i_tb/core/datapath_0/register_source_address_2
/core_rv32i_tb/core/datapath_0/register_source_data_1
/core_rv32i_tb/core/datapath_0/register_source_data_2
/core_rv32i_tb/core/datapath_0/immediate
/core_rv32i_tb/core/datapath_0/branch_result
EX STAGE
/core_rv32i_tb/core/datapath_0/instruction_address_ID_EX
/core_rv32i_tb/core/datapath_0/register_destination_address_ID_EX
/core_rv32i_tb/core/datapath_0/register_source_address_1_ID_EX
/core_rv32i_tb/core/datapath_0/register_source_address_2_ID_EX
/core_rv32i_tb/core/datapath_0/register_source_data_1_ID_EX
/core_rv32i_tb/core/datapath_0/register_source_data_2_ID_EX
/core_rv32i_tb/core/datapath_0/immediate_ID_EX
/core_rv32i_tb/core/datapath_0/ALU_result
/core_rv32i_tb/core/datapath_0/ALU_0/input_0
/core_rv32i_tb/core/datapath_0/ALU_0/input_1
MEM STAGE
/core_rv32i_tb/core/datapath_0/instruction_address_EX_MEM
/core_rv32i_tb/core/datapath_0/register_destination_address_EX_MEM
/core_rv32i_tb/core/datapath_0/ALU_result_EX_MEM
/core_rv32i_tb/core/datapath_0/mem_write_data_EX_MEM
/core_rv32i_tb/core/datapath_0/mem_read_data
WB STAGE
/core_rv32i_tb/core/datapath_0/instruction_address_MEM_WB
/core_rv32i_tb/core/datapath_0/register_destination_address_MEM_WB
/core_rv32i_tb/core/datapath_0/ALU_result_MEM_WB
/core_rv32i_tb/core/datapath_0/register_destination_data
/core_rv32i_tb/core/datapath_0/mem_read_data_MEM_WB
CONTROL
/core_rv32i_tb/core/controller_0/mem_to_reg_out
/core_rv32i_tb/core/controller_0/reg_write_out
/core_rv32i_tb/core/controller_0/mem_write_out
/core_rv32i_tb/core/controller_0/mem_read_out
/core_rv32i_tb/core/controller_0/data_format_out
/core_rv32i_tb/core/controller_0/ALU_Src_out
/core_rv32i_tb/core/controller_0/ALU_Op_out
/core_rv32i_tb/core/controller_0/branch_control_out
/core_rv32i_tb/core/forwarding_unit_0/forwardA_out
/core_rv32i_tb/core/forwarding_unit_0/forwardB_out
/core_rv32i_tb/core/datapath_0/PC_Src
/core_rv32i_tb/core/controller_0/jump_signal_out
/core_rv32i_tb/core/datapath_0/branch_result
/core_rv32i_tb/core/datapath_0/if_id_reset
/core_rv32i_tb/core/hazard_unit_0/flush_signal
/core_rv32i_tb/core/datapath_0/PC_Src_delayed
/core_rv32i_tb/core/datapath_0/compare_unit/branch
/core_rv32i_tb/core/datapath_0/compare_unit/branch_control
REGISTER FILE
/core_rv32i_tb/core/datapath_0/reg_file/x0_to_muxes
/core_rv32i_tb/core/datapath_0/reg_file/x1_to_muxes
/core_rv32i_tb/core/datapath_0/reg_file/x2_to_muxes
/core_rv32i_tb/core/datapath_0/reg_file/x3_to_muxes
/core_rv32i_tb/core/datapath_0/reg_file/x4_to_muxes
/core_rv32i_tb/core/datapath_0/reg_file/x5_to_muxes
/core_rv32i_tb/core/datapath_0/reg_file/x6_to_muxes
Figure 8 ModelSim wave simulation of Fibonacci program

4. Final observations

The proposed architecture worked as intended, although it's missing some optimization on datapath. The Patterson's and Hennessey's architecture is very didactic, but some modification was necessary to be able to perform all of the necessary instruction, such as adding the instruction address to the write back multiplexer and modifying the unit responsible for the jump and branch target address. Future development of this architecture should focus on merging the instruction decode, control, and immediate generator units, and improving structural hazard detection reliability.

5. Reference

https://en.wikipedia.org/wiki/VHDL

Hennessy, John L.; Patterson, David A. (2011). Computer Architecture, A Quantitative Approach.

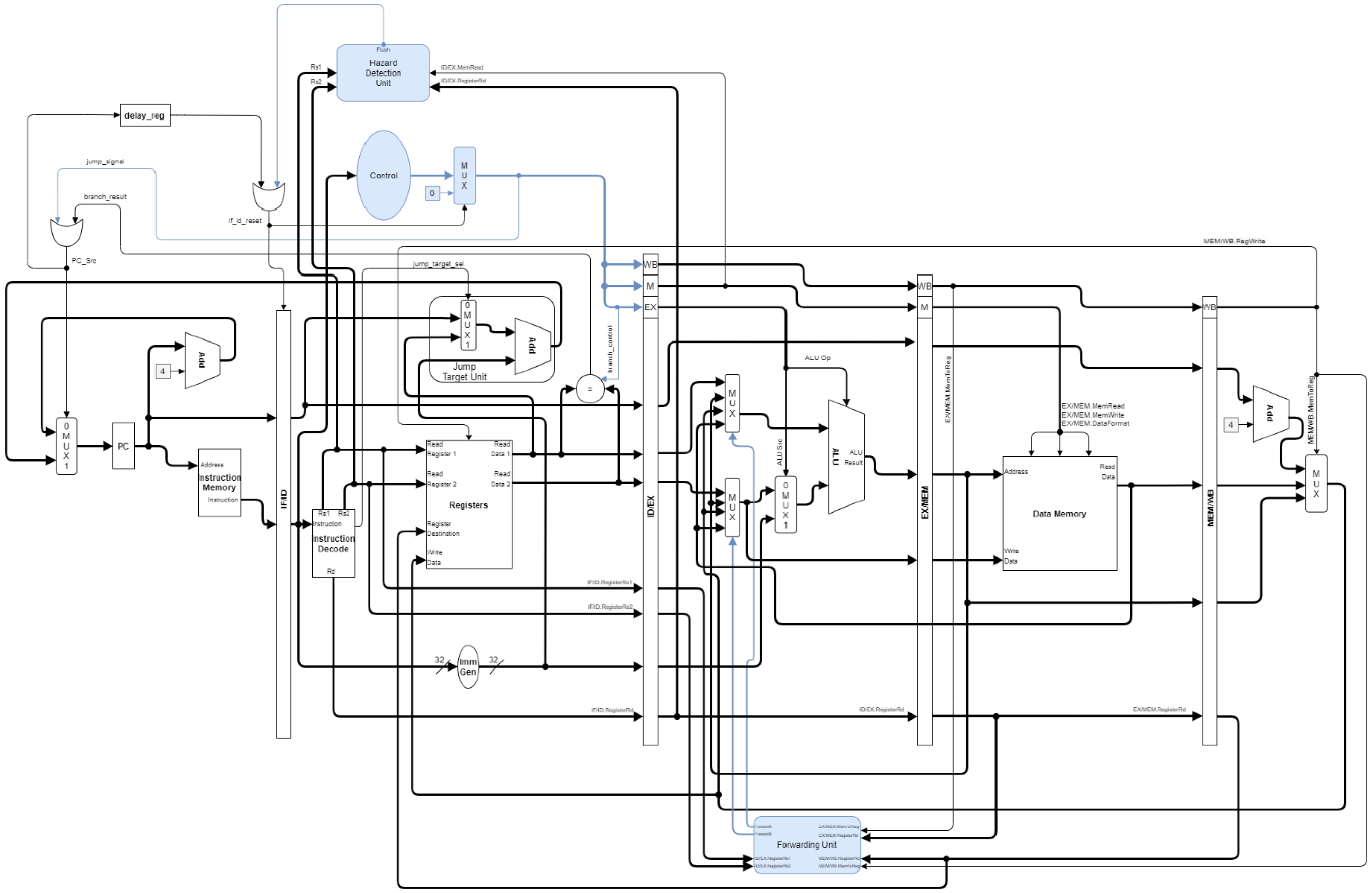
Hennessy, John L.; Patterson, David A. (2017). Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition.

https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/download.html.

https://www.elsevier.com/__data/assets/pdf_file/0011/297533/RISC-V-Reference-Data.pdf

https://github.com/Artoriuz/maestro

https://riscv.org/



Attachment 1 Architecture Block Diagram