**Chapter 2 Materials and methods**

1. **Materials**
   1. **Hardware**
2. Digilent Nexys A7 ([Xilinx Artix-7 FPGA](http://www.xilinx.com/products/silicon-devices/fpga/artix-7.html) XC7A100T-1CSG324C)
   1. **Software**
3. Microsoft Windows 10 Professional 64 Bit
4. Xilinx Vivado 2019.1 WebPack
5. Github
6. RARS – RISC-V Assembler and Runtime Simulator
7. Visual Studio Code
   1. **Reference books, articles, and web pages**
      1. **Book**
8. Computer Organization and Design RISC-V Edition [1]
9. Digital Design and Computer Architecture [2]
10. Introduction to computing system [3]
    * 1. **Articles and Documents**
11. Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2 User Guide (UG582) [4]
12. Vivado Design Suite AXI Reference Guide (UG1037) [5]
13. Combining Branch Predictors [6]
14. Slave Memories and Dynamic Storage Allocation [7]
15. RowHammer: A Retrospective [8]
16. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [9]
17. Note 5 of ECE-495/595 in ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, OAKLAND UNIVERSITY [10]
    * 1. **Web pages**
18. Longxin Cup final report (github/TrivialMIPS) [11]
19. lowrisc-chip (github/lowRISC) [12]
20. riscv\_soc (github/ultraembedded) [13]
21. Nexys A7 Reference Manual [14]
22. Xilinx forum [15]
23. Digilent forum [16]
24. Design of Digital Circuit (course of ETH Zurich) [17]
25. **Procedure**
    1. **Week 1**
26. Weekly Periodic Progress
27. Discuss what the expected demo of the project with group members
28. Discuss what the project needed to do (general construction of separate part)
29. Search reference of the system on chip and CPU core
30. Test the function of Digilent Nexys A7 board
31. Create the github page and determine the timetable of different parts
32. Weekly Progress Explanation

Since this project included many engineering and practical work, aim and expectation should be first discussed and determined. Based on the products designed by students from Tsinghua University, an system on chip was expected to be constructed successfully if some of components could be directly utilized without designing and testing [11]. Therefore, in this project, it was proper to meet the requirements that the aim of the hardware is to basically support the running of operating system and high-level design and protocol were unecessary.

In addition, preparation of Central Processing Unit (CPU) and peripheral of System on Chip (SoC) were also involved in the first week to help the hardware design. Most importantly, this project asked students to use Github to achieve the progress tracking and code sharing. Moreover, Github repository could also back up the progress of the project so that the risk of losing work would be low.

* 1. **Week 2**

1. Weekly Periodic Progress
2. Discuss which instruction set the operating system (xv6 or Linux kernel) needed (RV32IM)
3. Construct the simple Arithmetic Logic Unit Structure on Vivado 2019.1
4. Update ALU structure and support RV32M extension without verification (commits on Feb 8, 2020)
5. Update ALU Control Unit without verification (commits on Feb 8, 2020)
6. Construct register file (32 registers following RISC-V specification)
7. Weekly Progress Explanation

Before designing and constructing system on chip and Central Processing Unit, instruction set architecture would be the most significant abstraction to be fixed. RV32I and extension RV32M were necessary for xv6 or Linux Kernel. Among all the parts of the core, arithmetic logic unit (ALU) functioned as calculating the result; thus, it would be firstly designed. In addition, 32 register files were also designed because they were simple to construct when using static RAM in the FPGA board.

* 1. **Week 3**

1. Weekly Periodic Progress
2. Test the ALU with new extension support and fix the bugs (commits on Feb 9, 2020)
3. Construct Control Unit, and test ALU control and register file (commits on Feb 10, 2020)
4. Confirm the function of SDRAM to DDR Component provided by Digilent on board (commits on Feb 12, 2020)
   1. Advantages: It could be directly used to save time
   2. Disadvantages: minimum writing time was 260ns and reading time was 210ns, which caused huge memory operation latency. In addition, it didn’t have the output for a normal bus that guarantees the communication between memory and CPU core
   3. Problem existing: After reading or writing many times to memory, it would obviously lose data (maybe caused by refresh rate of DDR2 SDRAM in this FPGA)
5. Construct register between each stage used in pipeline processor
6. Test Control Unit and fix the bugs (commits on Feb 15, 2020)
7. Weekly progress explanation

Testing the components occupied half of the time in the hardware design, and in this project, all the unit tests were automati, but they didn’t contain all the circumstances. Additionally, control part included *ControlUnit* and *ALUControl* because separation between control logic and ALU control logic could simplify the design complexity. Furthermore, SDRAM to DDR Component took much time to test whether it could be correctly used, and the result were negative.

* 1. **Week 4**

1. Weekly Periodic Progress
2. Discuss how the operating system communicate with the CPU (system call)
3. Search information to solve the problem of DDR2 SDRAM, which cause the whole project not to meet the expectation progress.
4. Replace on chip DDR2 RAM with LUT RAM and Block RAM
   1. Can not support the compile file of operating system. Therefore, memory support is necessary
5. Weekly progress explanation

There were two main problems in this week. One was memory management unit while the other was the meaning of system call. The first one was due to the strange results when testing the SDRAM to DDR component and its low performance. The second one was relative to the student who designed the operating system because system call was responsible for all the I/O and other operations. It also included user mode or kernel mode in the interaction between operating system and the hardware. Hence, most of the work in this week was to search information, learn how to solve the problems, and discuss about the problems.

* 1. **Week 5**

1. Weekly periodic progress
2. Integrate all the components to be a pipeline processor with 5 stages (commits on Feb 27, 2020)
3. Construct Hazard Unit to solve control dependencies and data dependencies by data forwarding and pipeline stall (commits on Feb 28, 2020)
4. Construct static branch prediction (commits on Feb 28, 2020)
5. Simple test about the Central Processing Unit (commits on Feb 29, 2020)
6. Weekly progress explanation

In this week, Central Processing Unit was integrated by different components designed before and there were some pipeline issues in 5 stage pipeline processor. Data dependencies and control dependencies would affect the correctness of the processor and Hazard Unit was responsible for dealing with these problems. More details about the issues would be introduced in methods section. In addition, in this project, due to the limited time, static branch prediction “not taken” were used in the branch instruction.

* 1. **Week 6**

1. Weekly periodic progress
2. Construct cache system (commits on Mar 2, 2020)
3. Construct communication between cache system and memory by AXI4 full bus (commits on Mar 3, 2020)
4. Construct communication between cache and CPU core (commits on Mar 3, 2020)
5. Try to integrate operating system into the SoC
   1. Problem: No GPIO, No USB, Unknown bugs in cache system
6. Weekly progress explanation

In the final week, memory management unit and cache system were constructed to enable the system on chip work at the lowest level. These two parts were difficult, and it asked students to know much about the memory system and bus. Moreover, integrating boot, operating system and hardware were unsuccessful because of the loss of components and unknown bugs in hardware design especially in cache part.

1. **Method**
   1. **Hardware Description Language and FPGA board**

Hardware Description Language was the high-level abstraction of the digital circuit and it was largely applied in Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC). This project used FPGA as the hardware platform and Verilog as the hardware description language. In addition, some IP core provided by Xilinx was written by VHDL, which is more stricter than Verilog but more complex than Verilog.

FPGA board in this project was Digilent Nexys A7 and the reason why this board would be used in this project was its 128MB DDR2 SDRAM. Because the expectation of the project was to load the operating system into the system on chip, only static RAM on FPGA board could not support large program.

* 1. **Central Processing Unit**
     1. **RISC-V Instruction Set Architecture (ISA)**

RISC-V instruction was specified by the RISC-V Foundation and there was an official document to talk about all the instructions [9]. The document contained different instruction set architecture with different bit length per unit. For example, 64-bit and 32-bit instruction set would not be the same. In this project, due to the design complexity of 64-bit processor and the limitation of programmable logic slices, processor was chosen to be 32-bit. In addition, some instruction sets were discarded in this project to save time. RV32I Base Integer Instruction Set and ‘M’ Standard Extension for Integer were determined to be achieved in this project. RV32I was one of the basic instruction sets in the specification of RISC-V Foundation, which included the basic instructions like *JAL, ADD* and so on. RV32M Extension was one of the extension instruction sets and it included all the multiplication, division, remainder instructions.

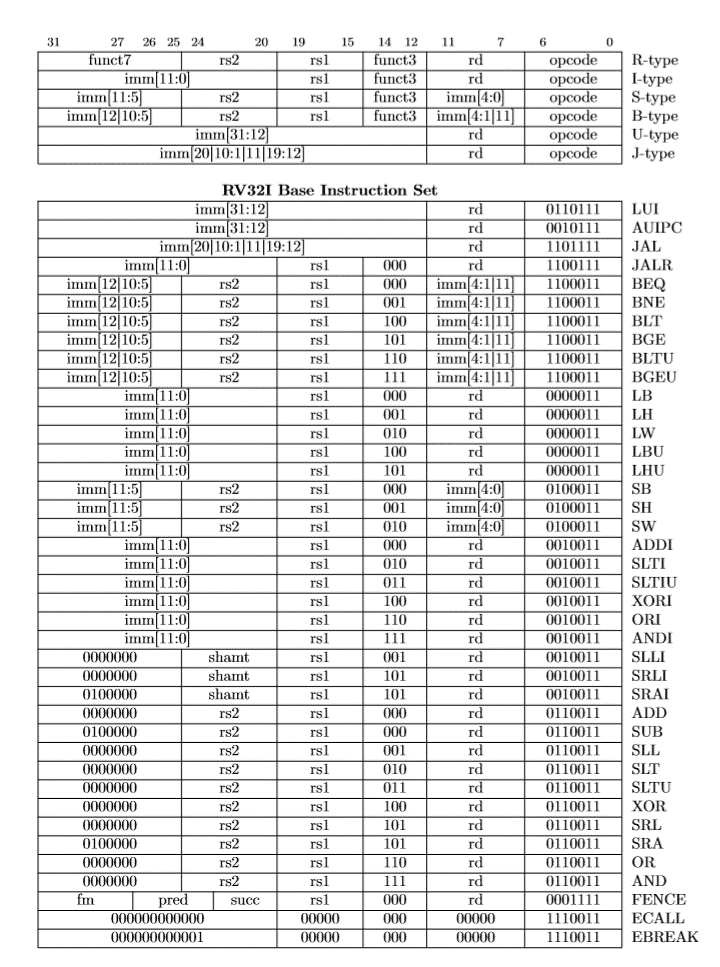


Figure 1: RV32I Instruction Set List [9]

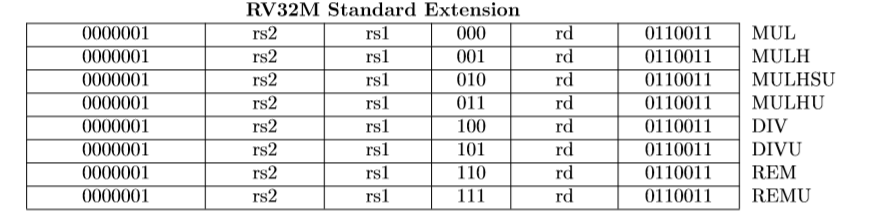


Figure 2: RV32M Instruction Set List [9]

Figure 1 and 2 showed the type of all the instruction in RV32I and RV32M. The detail of instructions could be read in the specification provided by RISC-V foundation and in this report, only the types of instructions would be introduced. They were R-type instruction, I-type instruction, S-type instruction, B-type instruction, U-type instruction, and J-type instruction.

R-type instruction meant that it needed two source registers and one destination register. Based on different function number, it would perform as addition, subtraction, and so on. All the R-type instruction would calculate the data from source registers and put them into the destination register.

I-type instruction meant that it needed one source registers and one destination register. The difference between I-type instruction and R-type instruction was that the second source register would be replaced by a 32 bits immediate value coming from sign-extending the immediate value in the assemble code.

S-type instruction meant that it needed two source registers and no destination register. This type of instruction would enable the program to store the data into the memory. Based on the figure 2, the immediate value consisted of bit 31 to 25 and bit 11 to 7, and then this value would be sign-extended to be a 32 bits value. Next, this immediate value would be added into the data in the first source register *rs1* so that the result would be the destination memory space. The data that would be stored was in the second source register *rs2* and function number *funct3* determined which length of the data should be stored in the memory.

B-type instruction was branch instruction. It had two source registers and one immediate value. It would compare the data in both two source registers and generate the result based on the result. If the branch hit, the program counter would get the right instruction address. The branch target address was added by sign-extended immediate value and the branch instruction address. Moreover, function code *funt3* would determine whether the comparison was signed or unsigned.

U-type instruction was special in RV32I. It only had two instructions that are LUI and AUIPC. The former was called load upper immediate and it would put 20 bits immediate value on the first 20 bits of destination registers with zeros on the last 12 bits. The latter was called add upper immediate to pc, which meant that the immediate value would be first extended with zeros on the last 12 bits and then be added to the pc, and finally store in the destination register.

J-type instruction only represented one instruction called JAL. It would add jump instruction address with sign-extended immediate value to generate the address of next instruction, and it would store the current next instruction address into the destination register.

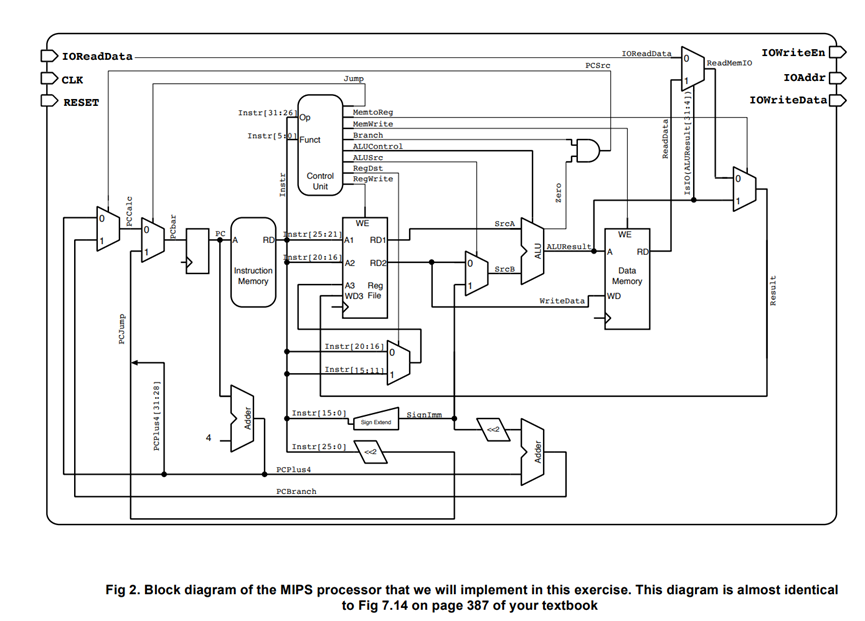
Categories of instructions help reduce the complexity when designing the arithmetic logic unit because when designing and constructing the core, instructions themselves could be separated from outer layer. That is, designers could first regard the same type instructions as entirety and then determine each instruction when designing the arithmetic logic unit and control unit.

* + 1. **Microarchitecture**

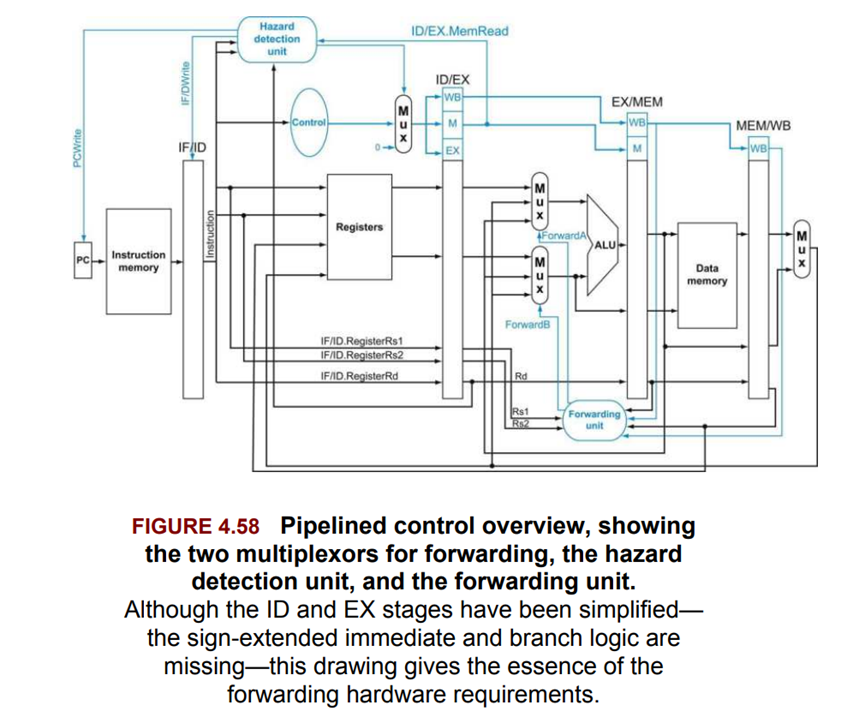
**图片包含 游戏机, 街道, 空地, 大

描述已自动生成**

**Figure 3: CPU structure diagram in the project**

****

**Figure 4: Multi-cycle MIPS processor [17]**

****

**Figure 5: Simple RV64 RISC-V processor [1]**

* + - 1. **Pipeline**

In this project, the RISC-V microprocessor utilized five-stage pipeline structure. Compared with the multi-cycle and single-cycle microprocessor, pipeline microprocessor had higher Instruction Per Cycle (IPC) so that it could have higher performance the other two structure. Furthermore, five stages were Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Operation (MEM) and Write Back (WB). IF stage meant that during the current cycle, CPU would get the instruction data from instruction cache. ID stage meant that during the current cycle, CPU would decode the instruction to get the control logic and data in the source registers from register files. It also sign-extended the immediate value to meet the requirements of the instruction. Execution stage was that CPU calculated the data and returned the results. Memory operation stage included two parts that were branch/jump logic and memory. In this stage, if the instruction was branch instruction or jump instruction, and the branch hit, program counter (PC) would get the branch target address. At the same time, if the instruction was store or load instruction, it would get the information from data cache. Write Back stage was that if the instruction needed to write data into the register, it would happen in this stage.

Five stages worked at the same cycle for different instructions and each instruction would be divided into five parts to be run one by one. In order to enable pipeline to flow quickly, there were four pipeline registers between each stage to store the data and results from previous stage, which called IF/ID Register, ID/EX Register, EX/MEM Register, and MEM/WB Register. The performance of pipeline processor could be showed by IPC. When the processor entered the full pipeline state, IPC could be 1 without any pipeline issues mentioned below. Although IPC in pipeline structure was nearly the same as the single-cycle processor, pipeline structure had much higher cycle frequency because the cycle frequency of single-cycle processor was determined by the longest instruction.

* + - 1. **Pipeline issues**

Even if the pipeline structure had such perfect performance, it had some issues when being constructed. Data dependencies and control dependencies are two issues in this project.

Data dependencies contains three dependencies that were Flow dependence (Read-after-Write), Anti dependence (Write-after-Read), and output dependence (Write-after-Write) [2]. The first data dependencies would cause incorrect results in the pipeline while the other two data dependencies would not cause problems in in-order processor, which would be discussed in in-order section and improvement section in discussion chapter. The reason was that WAR and WAW data dependencies was normal when the processor executed the instruction one by one in von Neumann model, but RAW dependency might induce next instruction to get the wrong data because the former one hadn’t written its result into the destination register. In this project, data forwarding was used to deal with these issues. Data forwarding meant to transfer the data in MEM stage or WB stage to ID stage if the instruction in ID stage needed to read the register that would be also be written back by the instruction in MEM stage or WB stage. However, there was still one condition where memory operation spent more than one cycle. It needed to stall the pipeline because the instruction was incomplete in the current stage.

Control dependencies was about the address in program counter when going to the next cycle. If instruction in IF stage was branch instruction or jump instruction, the next instruction fetched would depend on the former one. If the instruction was branch instruction, branch prediction would help solve this dependency. In this project, in order to reduce the complexity, ‘not taken’ branch prediction was applied, which meant that the processor would predict that the branch would miss and firstly used PC + 4 as the address of next instruction to be fetched. Nevertheless, if the result was that branch hit, processor will flush the IF/ID register and ID/EX register, and keep ID/EX register to be zero for extra one cycle because these two registers are running wrong instruction so that pipeline in these two stages should be flushed and wait the correct instruction.

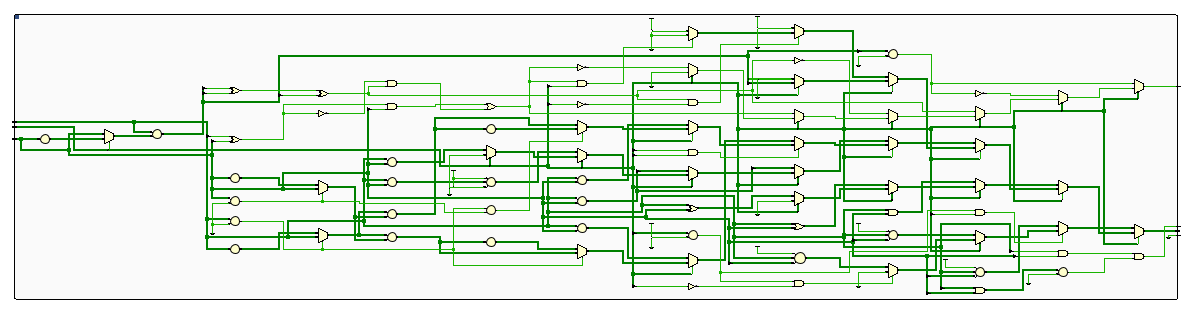
* + - 1. **In-order and scalar structure**

In-order structure meant that the processor would run the instruction following the order of the instruction. The advantages of this structure were that it was easy to construct; on the contrary, it has low performance if some instructions such as load instruction spent two or more cycles being complete. Originally, this project was expected to apply out-of-order execution, which meant that instruction would be run according to its data dependencies instead of order. However, due to the time limit, in-order processor replaced the out-of-order processor and the detail would be discussed in the future. Moreover, scalar processor was that the processor could only run one instruction at a time while superscalar processor could run multiple instructions at a time. In this project, due to the complexity and issues in superscalar processor, scalar processor was chosen to replace the latter.

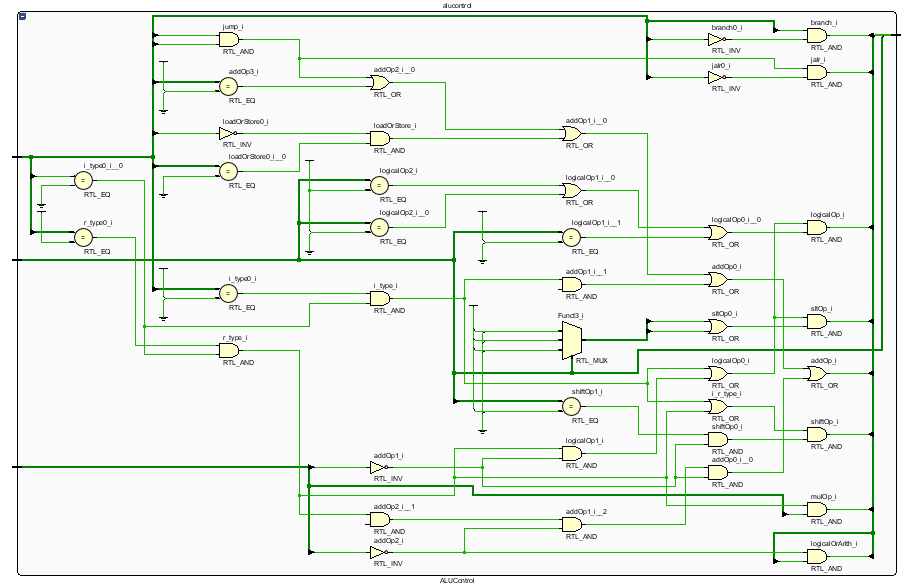
* + 1. **Design and Construction**
       1. **Arithmetic Logic Unit (ALU) and ALU Control**

In this project, ALU component would get three inputs, which were two sources, and one ALU control logic. According to the RISC-V instruction set architecture, there were six types of instructions and they would be dealt with by different logics and finally outputted the result by multiplexer. The most difficult parts in ALU were unsigned operation and RV32M instruction set. The former was due to the boundary problem of the unsigned number and the latter was because the RV32M instruction included multiplication, division, and remainder operations that toke much resource and complexity on the FPGA board. Multiplication, Division, and remainder toke the Register transfer level (RTL) synthesis circuit as the solution because this could largely lower the complexity students met when designing their own digital logic circuit for these three operations. However, RTL synthesis logic were not optimized, and it would spend 200ns or more to calculate a 32 bits multiplication; thus, this problem would affect the cycle frequency of the processor.

In this project, the control logic of ALU was separated from the Control Unit because integration might increase the complication of the control Unit and make the designer difficult to debug. In addition, it can lower the codes and registers because ID/EX register didn’t need to store the ALU control logic, but it would increase the latency in the execution stage. Furthermore, to simplify the code, ALU control logic is divided into different parts to represent which instruction it was. The ALU Control component would generate a 11 bits result, which consists of *addOp, branch, shiftOp, logicalOp, mulOp, sltOp, jalr, logicalOrArith, Funct3*. These bits were determined by *opocode, funct3, funct7* that decoded in the ID stage. For example, *ADD* instruction and *SUB* instruction were only different on *funct3*, which meant that one is 000 while the other is 010. Thus, *addOp* would be 1 if the instruction was *ADD* instruction and be 0 if the instruction was *SUB* instruction. Only the bit addOp had two meanings about the value because addition and subtraction are opposite and other bits only represent whether it was or not.



**Figure 6: ALU Elaborated Design Schematic**



**Figure 7: ALUControl Elaborated Design Schematic**

* + - 1. **Register File**

Register File was the place to contain 32 registers. To keep the register read and write in one cycle, static RAM in the FPGA board was used. Therefore, the processor could make sure that reading registers and writing to registers would be quick enough.

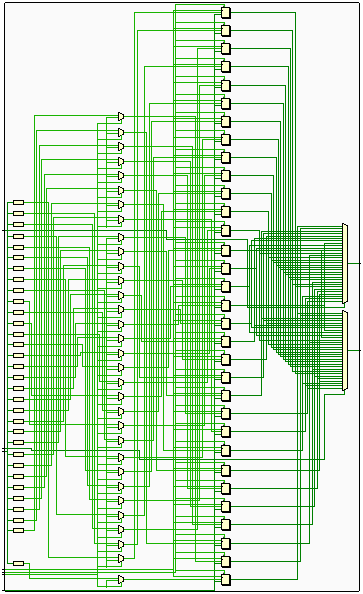


Figure 8: RegisterFile Elaborated Design Schematic

* + - 1. **Stage Registers**

There were four stage registers called IF/ID register, ID/EX Register, EX/MEM register, and MEM/WB register. They were to store the data of each stage to keep the pipeline flow. They were constructed by static RAM in FPGA because they wouldn’t occupy many resources so that the usage of static RAM could keep the registers to finish store during a short time (less than one cycle).

* + - 1. **Control Unit**

In this project, ControlUnit was responsible for generating the control logic except for the ALU control logic. It would compose all the logic into three different logic clusters that are *EX\_control*, *MEM\_control*, *WB\_control*. The first one was all the control logic useful in EX stage, the second one was that useful in MEM stage, and the final one was that useful in WB stage.

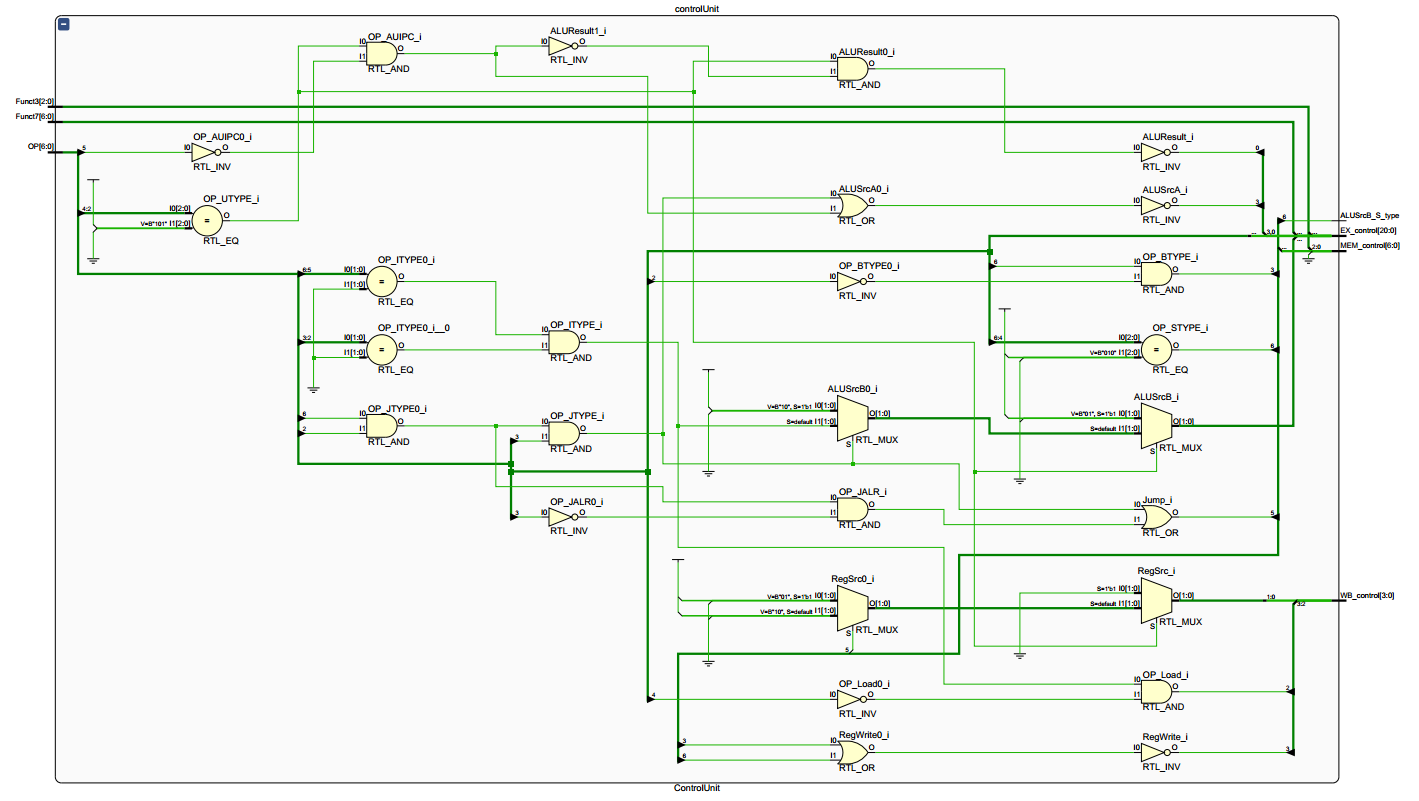


Figure 8: ControlUnit Elaborated Design Schematic

* + - 1. **Hazard Unit**

Hazard Unit was responsible for dealing with the pipeline issues such as data dependencies. It would get the destination register of instruction in MEM stage and WB stage, and then compared them with two sources register in the ID stage. If source registers were the same as the destination register, which was flow dependence (RAW), the hazard Unit would enable the result in MEM stage or WB stage to be the result of the following source registers. Moreover, if instruction in Execution stage was load instruction, it was necessary to stall the pipeline if the instruction in ID stage needed to read the same register that is the destination register of the instruction in Execution stage. To stall the pipeline, hazard unit will flush the ID/EX register and fill it with zero. It would also keep all the logic in Fetch stage and Decode stage same through an Enable Control logic circuit.

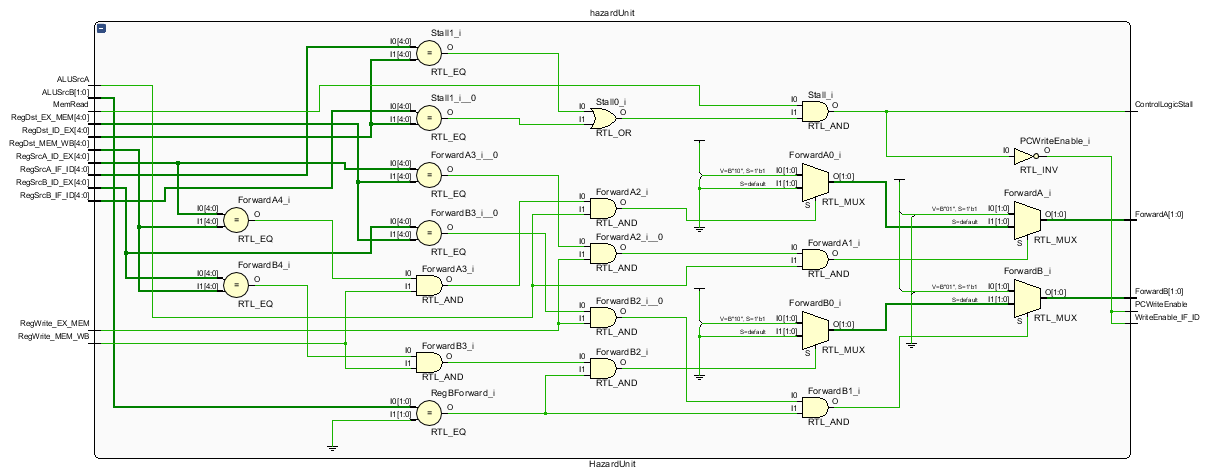


Figure 9: HazardUnit Elaborated Design Schematic

* + - 1. **CoreTop**

CoreTop was the top of the core, which connected all the components to enable them work together. It included a register called Program Counter (PC), the choice of source data for the ALU, bits dispatching of sign-extended immediate value, and the wire connection of each component.

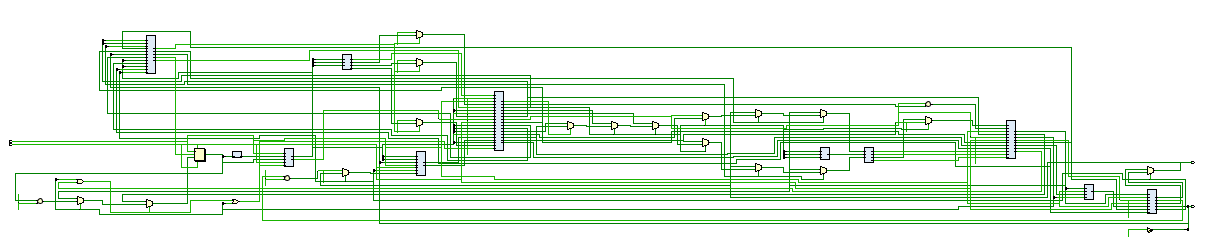


Figure 10: CoreTop Elaborated Design Schematic

* 1. **System on Chip**

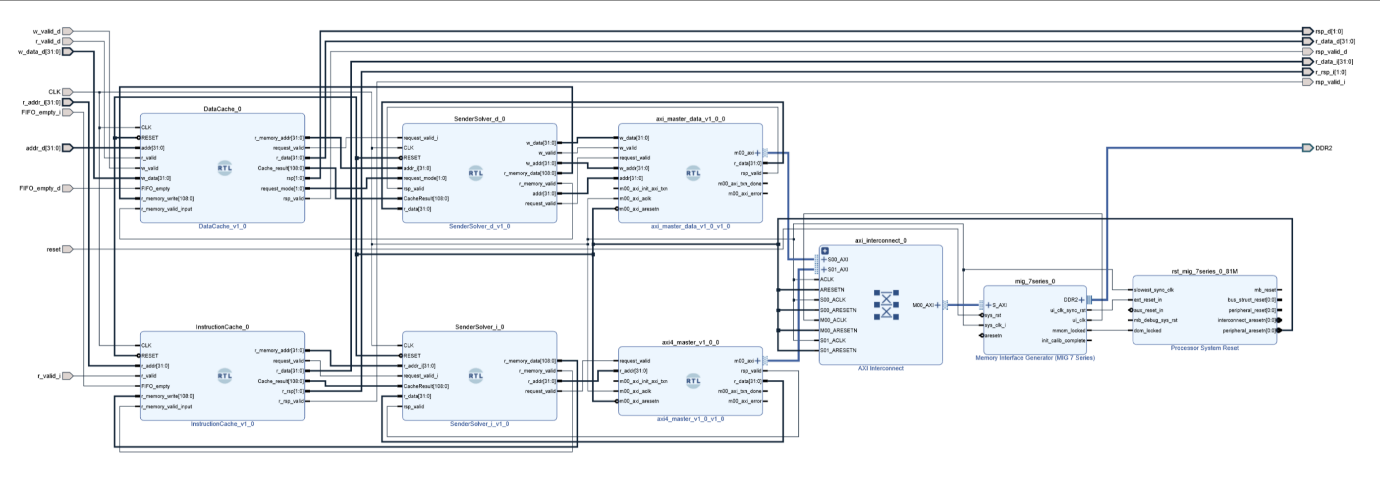


Figure 13: System on chip Block Design Schematic (Only Cache and MIG)

As the figure 13 showed, the system on chip part was designed by Vivado block design so that designer could not spend much time on port naming and connecting.

* + 1. **Cache**

Due to the time limitation, cache part including structure design and code would have many problems and unknown bugs. Therefore, some design described in the method part might be incorrect because it was about the current design. In discussion section, some of the problems and their solution would be contained. Moreover, there were no automatic tests in cache part.

* + - 1. **Storage Structure**

Data/Instruction Cache was constructed by Xilinx Parameterized Macros (XPM) Single Port RAM (xpm\_memory\_spram). This single port RAM used LUT to construct a distributed RAM so that the RAM could provide result within one cycle. The size of cache was 111616 bits, which was 1 KB because the byte size of cache is 109. Since the size of cache was small, cache could use LUT RAM; otherwise, Block RAM would be better with higher latency. However, instruction cache should keep low latency because instruction cache was not necessary to be large enough for the processor. Furthermore, to increase the hit rate of the cache, cache used two associativity. Two associativity meant that one word in cache could store two data, which helped the cache hit because there was enough space to avoid the situation that instruction read A, read B, and read A again.

One byte of the cache consisted of Valid bit (V), Used bit (U), Dirty bit (D), Tag bits, Data bits. The detail of the arrangement in the cache was as the figures 12 showed. Set bits in address was to choose which block in the cache and due to 1KB size of cache, 10 bits of Set would be necessary. In addition, cache used least recently used policy to evict the block in cache. Used bit was to show which one in the cache is recently used and then evict the other one for the new data. Moreover, cache would only write the dirty block into the memory if it had to be evicted so as to reduce the memory operation.

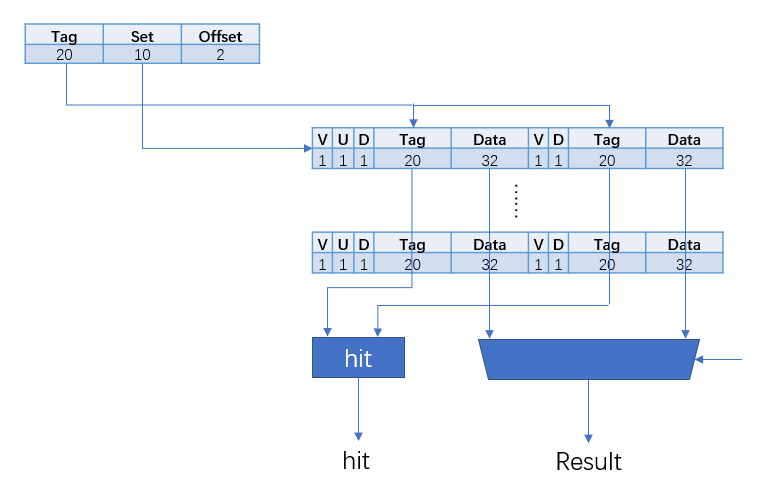


Figure 12: Storage Diagram and Basic cache structure

* + - 1. **Cache Structure**

In this project, cache was constructed by three stage pipeline structure. The first stage was Read stage, the second one was Comparison stage, and the final one was Response Stage. The first stage was to read the data from cache according to the tag that was bits 11 to 2. The second stage was to compare the offset with the address that instruction required to read or write. There were several results of comparison stage, which were cache hit (read instruction), cache hit (write instruction), cache miss (read instruction), and cache miss (write instruction). Cache hit (write instruction) would replace the data reading from cache during comparison stage. In Response stage, if the cache miss, it would response to the CPU that it needed preparation and sent the request to a component specially solving it. If the cache hit, it would response to CPU that all data was ready, and CPU would get the result when they required the data again. The difference between instruction cache and data cache was that instruction cache didn’t take writing into account and it would only deal with the reading request.

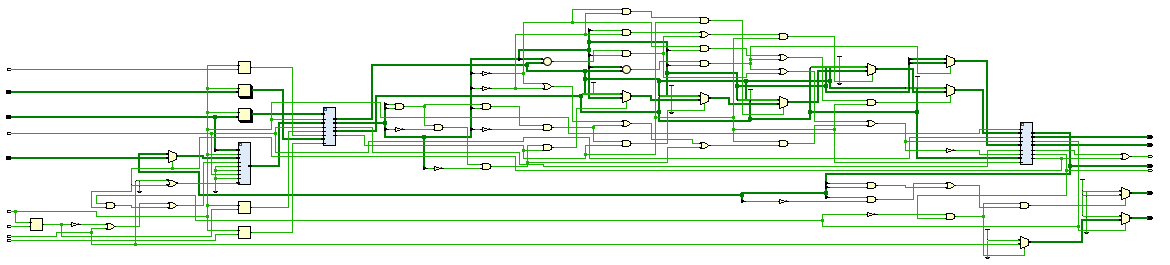


Figure 11: Data Cache Elaborated Design Schematic

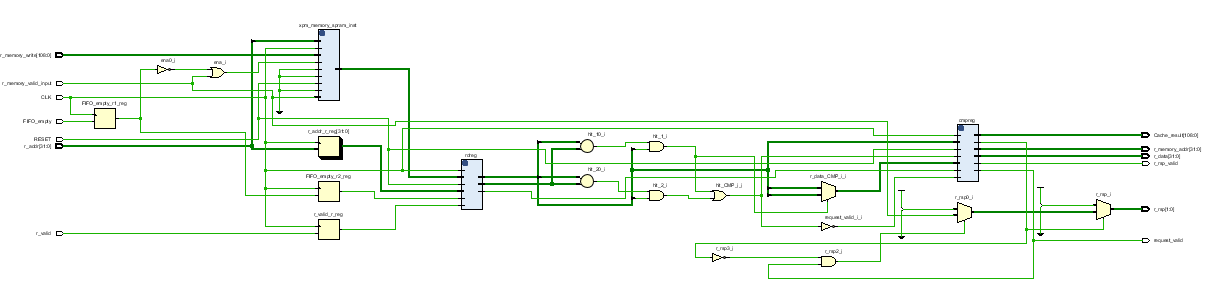


Figure 12: Instruction Cache Elaborated Design Schematic

* + - 1. **Communication between CPU and cache**

Communication between cache and CPU was designed to use asynchronous FIFO. This asynchronous FIFO was constructed by Asynchronous FIFO (xpm\_fifo\_async) in Xilinx Parameterized Macro (XPM). There were two FIFOs between Cache and CPU, and the data pushed would have valid bit and data bits to inform Cache or CPU whether the operation was ready or not.

* + 1. **Memory Interface and Request Solver**

Request Solver part would have some problems and unknown bugs because there were no tests due to time limitation. However, memory interface would work correctly because it had been tested when dealing with SDRAM to DDR component. This test would be provided with more detail in the later section.

* + - 1. **Request Solver**

Request solver included two parts that are sender solver part and Advanced eXtensible Interface (AXI) part. The former would deal with the request from the cache and translate them into proper code type and then sent them to the AXI master. The latter included two AXI4 master and an AXI interconnect to translate the information from sender solver into what the AXI4 bus required.

The reason why cache would not directly translate the request into AXI4 mode was that it was complicated if the cache included all the communication between memory and cache. Therefore, sender solver would have duty on this. It would not only deal with the request reading or writing data into the memory but also got the result from AXI4 master, and then returned them to the cache. The difference between sender solver for data cache and sender solver for instruction cache was that instruction cache only needed to read data from memory while the other one needed to read or write the data from or into the memory.

AXI part included two AXI4 masters and one AXI interconnect. The version of AXI4 master is 1.0 and the version of AXI interconnect is 2.1 (Rev. 21). Both of two components used Xilinx IP core to simplify the design complexity. AXI4 protocol was developed by ARM and it could solve the communication problems between different components. Since memory interface only supports AXI4 FULL instead of AXI4 LITE, a simple form of AXI4 protocol, all the AXI part should be designed for AXI4 FULL protocol. AXI protocol was burst-based and it had five independent transaction channels that were Write Address Channel, Write Data Channel, Write Response Channel, Read Address Channel, Read Data Channel. More details about AXI4 FULL bus would refer to UG1037 and the figure below [5]. In Vivado 2019.1 block diagram design, it would automatically connect the AXI4 master and slave. Only data sending and receiving in both two masters would be dealt by designer. AXI4 master was responsible for sending request in the protocol; thus, the sender solver would connect to an AXI4 master and sent the information to that component. In this project, Memory interface would become AXI slave, which would receive requests and respond. AXI interconnect was to help multiple AXI4 master to connect to one slave in this project. It would be synthesized by Vivado automatically and didn’t need to be changed.

* + 1. **Memory Interface and SRAM to DDR component**
       1. **SRAM to DDR component**

Before talking about the memory interface, SRAM to DDR component was given by Digilent that could enable the designer to regard DDR2 SDRAM as static RAM. The FPGA board had 128MB DDR2 SDRAM, and the word size was 16 bits. Digilent provided all the options and constraints of this component. Designer only needed to use eight ports to communicate with memory. Since it didn’t have valid bit transfer, before reading and writing, there should be a period to keep the input stable. On the official document, the minimum cycle time for reading was 210ns and writing was 260ns. However, there were some problems in this component, and they would be mentioned in the later chapter.

* + - 1. **Memory Interface**

Due to the problems in SRAM to DDR component, Memory Interface Generator (MIG) provided by Xilinx could generate a memory interface to replace this component. The version of MIG was 4.2 (Rev. 1). It could automatically generate an AXI4 interface to connect to the AXI4 interconnect component.

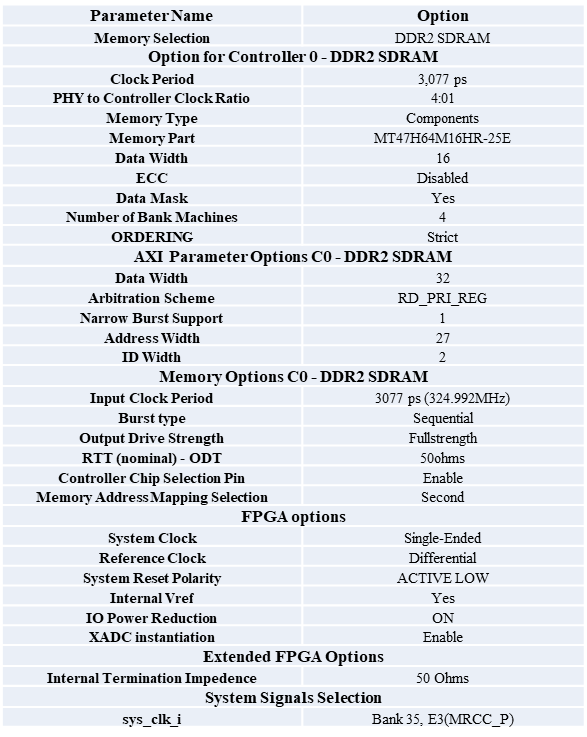


Figure 13: Memory Interface Generator Option for Digilent Nexys A7

* + 1. **System clock generation**

On FPGA board, there was only one clock generator and its frequency was 100MHz. However, to fulfill the clock requirements of cache and MIG, clock wizard, an IP core provided by Xilinx would be utilized. It could use system clock to generate the clock whose frequency was expected. For example, since MIG needs 3077ps and 200MHz clock, these two clocks frequency both needed to be generated by clock wizard.

1. **Other tools**
   1. **Github**

All the project file was uploaded to the Github repository. The reason was to track all the progress in the project and help three students share their codes. In addition, it could back up the project file to avoid unexpected situation.

* 1. **Visual Studio Code**

It was the Integrated Development Environment for software. However, its plug-ins supported developing Verilog in a convenient way. It also supported xvlog (a tool for checking Verilog error) provided by Xilinx.

* 1. **RARS – RISC-V Assembler and Runtime Simulator**

This simulator was constructed from MARS 4.5, a MIPS assembler and runtime simulator. It could generate RISC-V machine code quickly. In this project, this simulator was used to test the CPU through the RISC-V program.

* 1. **Vivado WebPack 2019.1**

It was the Electronics Design Automation tool for FPGA development built by Xilinx. It could automatically synthesize, implement and generate bitstream based on Verilog or VHDL code. It could also simulate the Verilog or VHDL code, which could test the design efficiently. In addition, it also supported block diagram design so that AXI4 bus design become simpler. Moreover, it integrated many IP cores that are used in this project to reduce the complexity of designing a large complicated system.

**Chapter 3 Result**

1. **Github repository**

Project Github repository webpage: https://github.com/dingqy/ELEC222-Project

RISC-V Core Github repository webpage: https://github.com/dingqy/riscv-core

All work had been uploaded into the repository and it also included the progress track of each student

1. **Testing Result**
   1. **Automatic tests**

All the tests in current project progress were unit tests about each component. It would generate corresponding inputs and compare output with the expected signal. The test of core was to use simple RISC-V program to observe the signal in the circuit. Due to the uncomplete cache, during the test, instruction memory and data memory were all replaced by simple static RAM on FPGA board.

Due to the time limitation, some tests were not finished. They are complete verification of core, cache test, request solver test, and SoC test. In addition, since due to temporary change of Control Unit, current Control Unit test could be utilized. However, during simple core test and previous Control Unit test, it could work successfully.

* 1. **Testing Diagram**

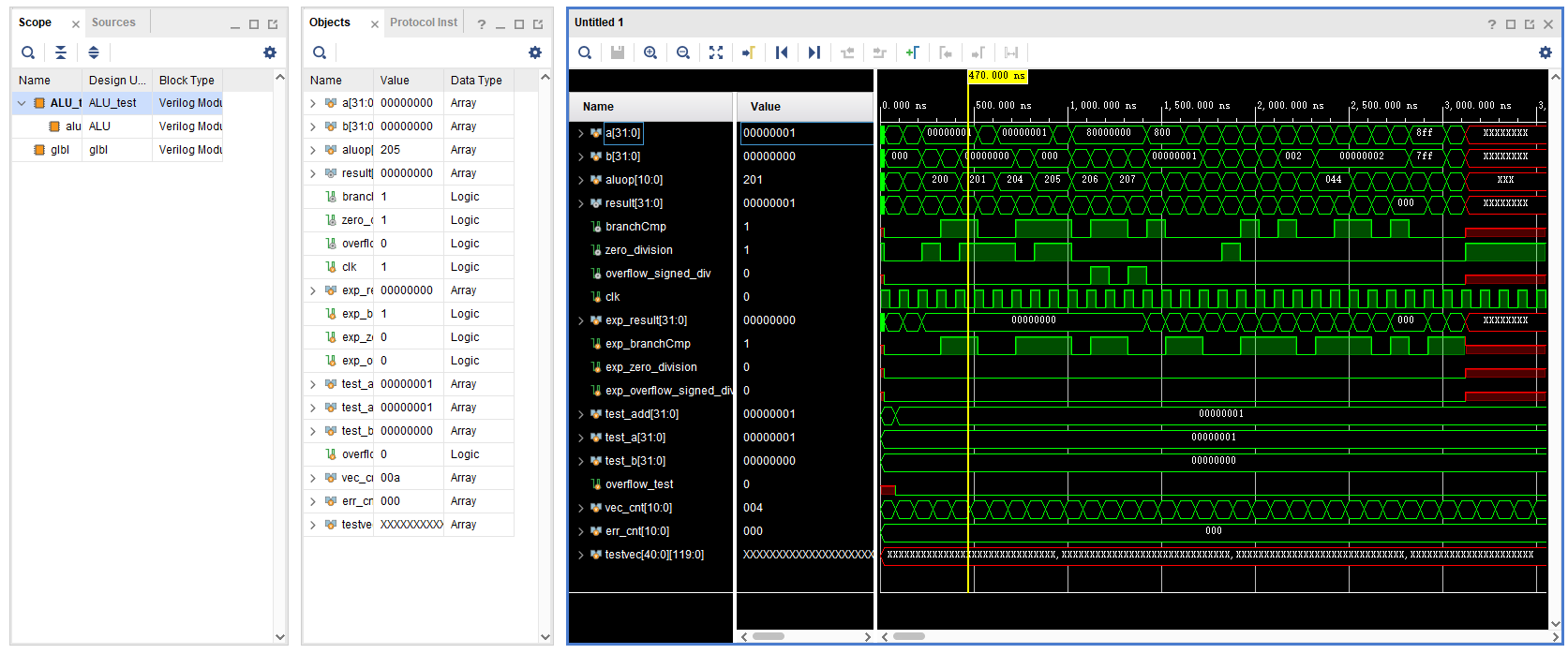


Figure 14: ALU Simulation Diagram

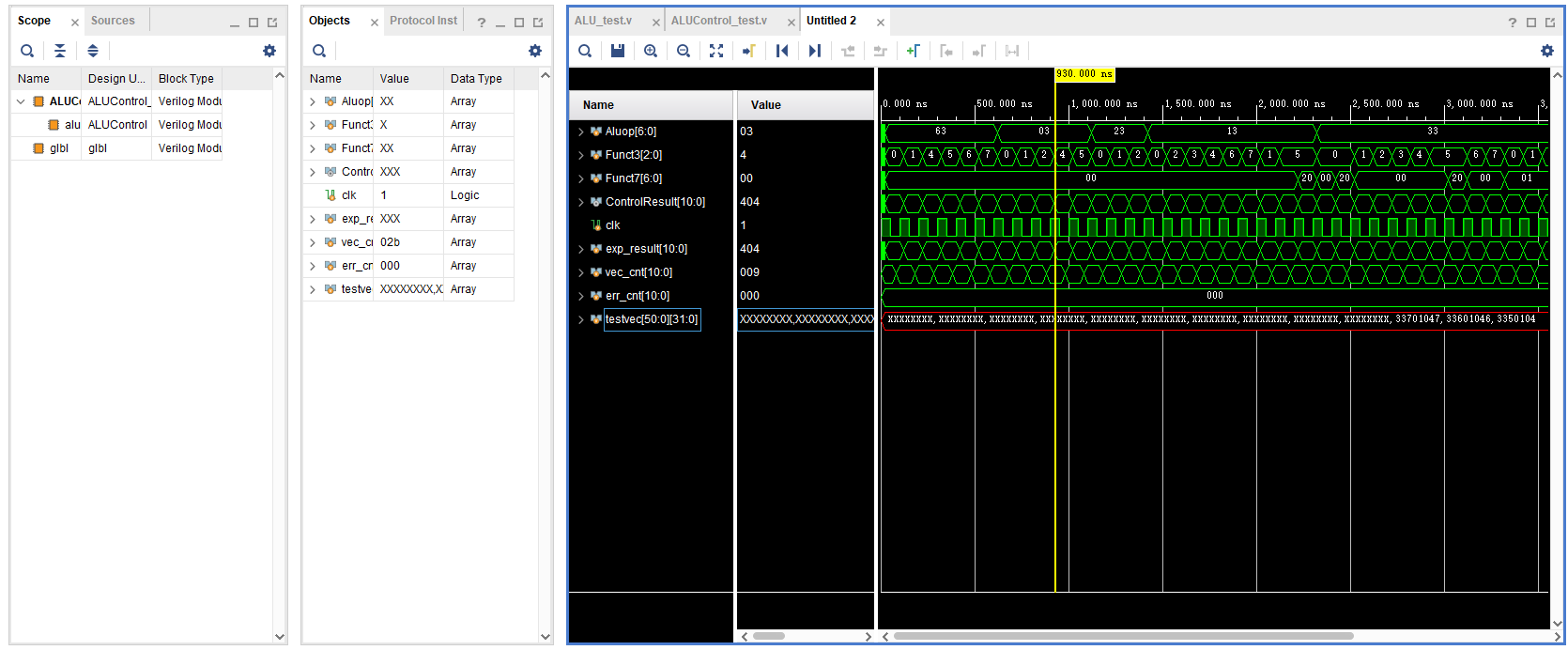


Figure 15: ALU Control Simulation Diagram

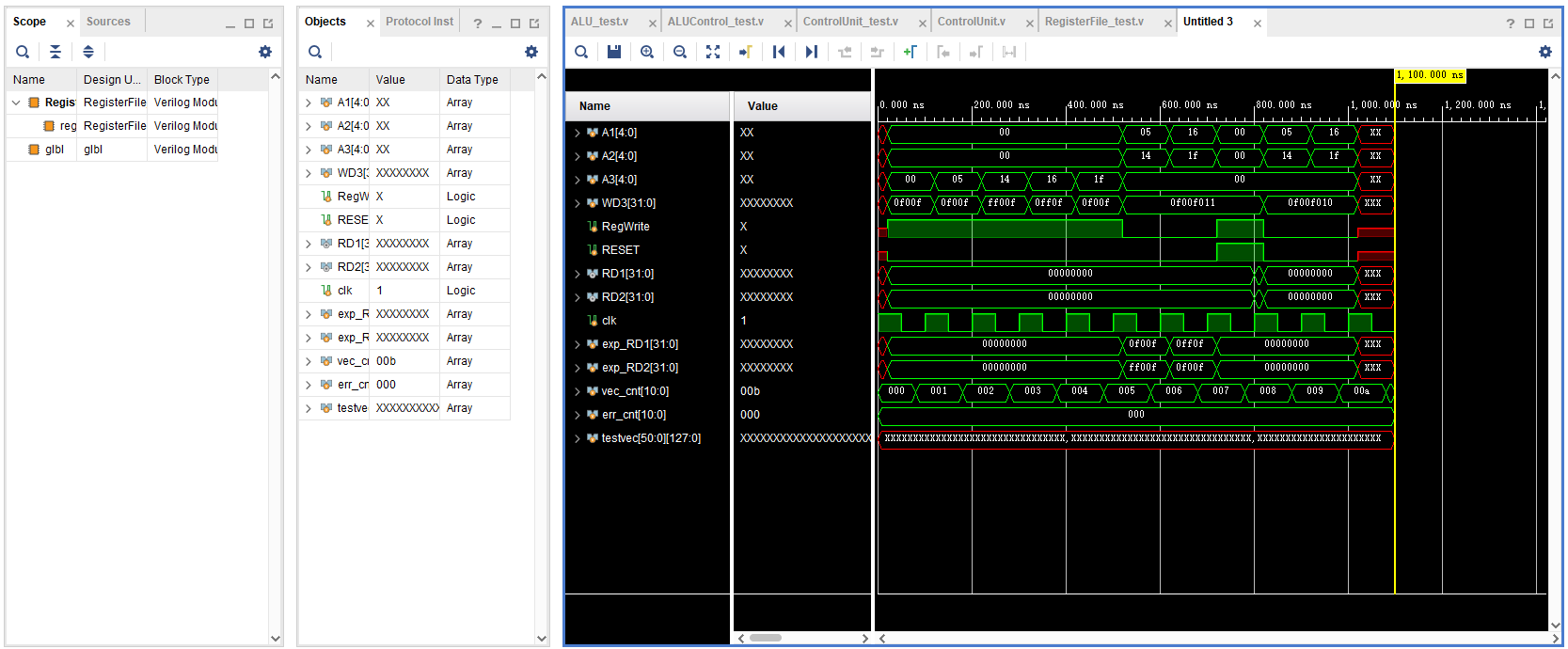


Figure 16: RegisterFile Control Simulation Diagram

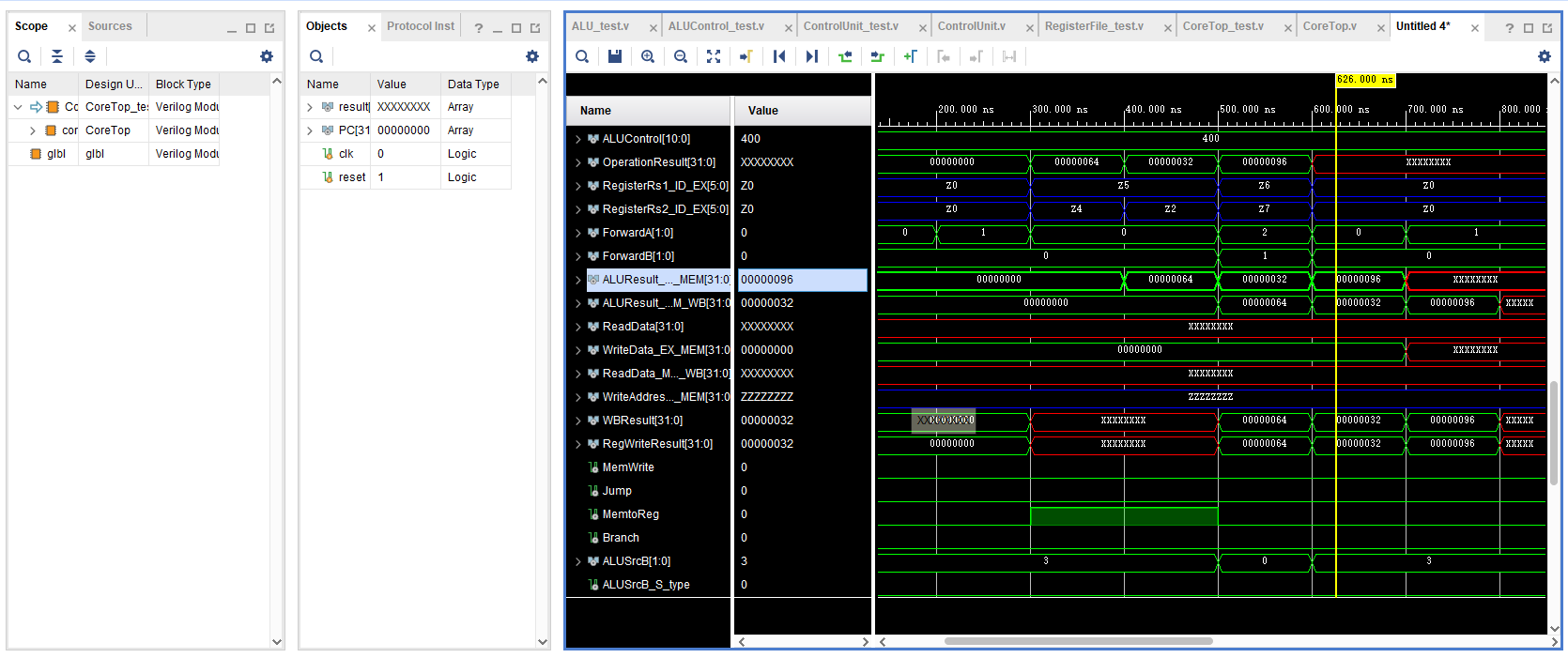


Figure 17: CoreTop Control Simulation Diagram (part of the simulation diagram)

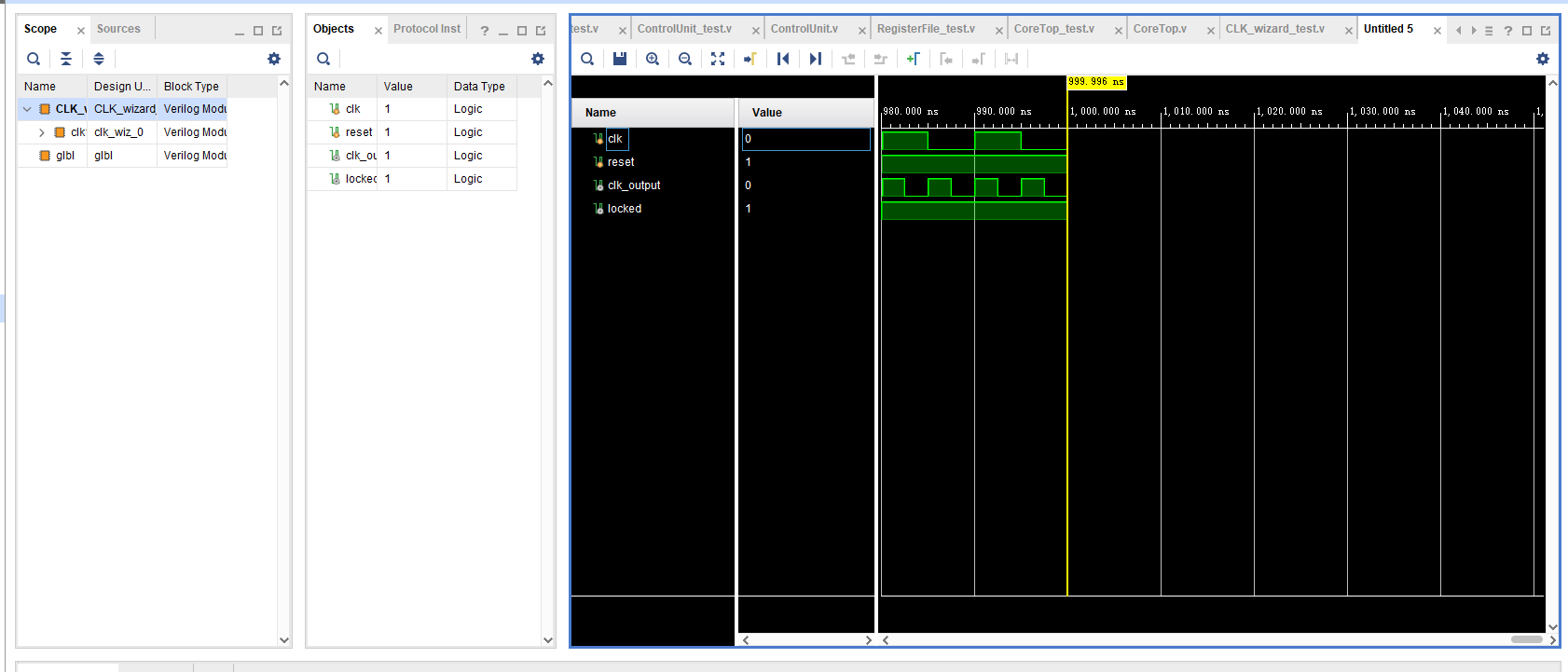


Figure 18: Clock Wizard Control Simulation Diagram

* 1. **SRAM to DDR component test**

This test was implemented to execute on the FPGA board; thus, there was no simulation diagram. In ‘tb’ file folder, there was *MMU\_test.xdc* and *MMU\_test*. The former was the constraints of the test and the latter was the circuit built in the FPGA board. The final result was that when writing and reading repeatedly from one memory location, the LED on FPGA board to show the data in the memory would be firstly correctly and then become the value that was unknown, and finally, all the LED would not light. One of the possible reasons was the memory fresh rate and detail about this result would be discussed below.

**Chapter 4 Discussion**

1. **Explanation of results**

In this project, there were no data collection and analysis; thus, all the results were the simulation result. In addition, since not all the components were completed or worked correctly, the whole System on Chip could not have direct exhibition on the board. However, all the simulation showed that the 5-stage pipeline processor core could work, although they were not tested entirely. Moreover, SRAM to DDR component testing were shown on the board and there existed some problems when actually testing on the board. Additionally, CPU benchmark was not applied due to the time limitation and the default main frequency was 100 MHz.

1. **Limitation**
   1. **Test limitation**

Test limitation is one of the most significant problems during this project because test may take huge amount of time to include all the circumstances. Moreover, from unit tests to system tests, they are too complicated to test them simply. For example, only a CPU core instruction set test needs to test more than fifty instructions and their dependency issues. Therefore, in this project, there are only unit tests of CPU core component and simple system test of CPU core due to the limited time of the project.

* 1. **Performance limitation**

The core is a 5-stage pipeline scalar in-order processor. The Instruction Per Cycle (IPC) was lower than 1. In addition, due to the large memory operation time, it may be difficult to keep the frequency up to 100 MHz. For example, if the project applied SRAM to DDR component, which spends 210ns for reading and 260ns for writing, memory operation may waste more than 20 cycles. Therefore, the design of cache would determine the performance of the CPU. Moreover, performance limitation of the SoC may affect the final exhibition when executing operating system.

* 1. **Application limitation**

Due to limited tests and uncomplete components, before generating bitstream into the FPGA board, all the results will be the simulation of Vivado. It doesn’t mean that all the logic circuit would work correctly on FPGA board. This would cause many unknown problems and bugs when actually using the board. For example, SRAM to DDR component behave normally when simulating in the Vivado, but it caused some strange problems when testing on the board.

FPGA board was also one of application limitation because when the system on chip become more and more complex, the logic circuit needs more LUTs, RAMs, and I/Os, which caused FPGA not to support the logic. However, in the current progress of the project, it is not a problem, but it may be if the system becomes complete.

1. **Problems and obstacles**
   1. **RISC-V instruction set determination**

Choosing proper instruction set is a large problem in this project. Due to little knowledge about the hardware/software interface, determination of instruction set architecture affects the progress of making plan of the project. Originally, RV32I is enough for operating system. However, when designing operating system, RV32IMA is necessary. Nevertheless, RV32A may be unnecessary because it is used to support atomicity of multi-thread. Finally, RV32IM are determined and becomes two instruction sets designed and constructed in the core.

* 1. **Scalability and readability of component design**

The style of naming components and wires, and comments are essential when writing Verilog code. Compared with VHDL, Verilog is not as strict as VHDL so that when designing complicated system, Verilog may cause less understandability. Therefore, comments and proper names can help a lot when writing Verilog code. However, it still lowers the efficiency because there were too much wires and components, which induces designer to spend a lot of time reading code and understanding what code means.

Scalability are also crucial when designing. Arithmetic Logic Unit is designed twice because when designing at first time, the scalability and readability of the Verilog code are low so that RV32M extension set could not be correctly added into that ALU design. In addition, more than fifty instructions also increase the difficult of whole structure due to the different function of instruction. For example, there were signed instructions and unsigned instructions that make logic more difficult to distinguish.

* 1. **The complexity of Cache design**

There are different types of cache including L1 cache, L2 cache, and L3 cache. In this project, to simplify the design, there is only L2 cache; however, it doesn’t mean that it is a proper design choice because it needs a lot of experiments to find an efficient way to decide the proper memory hierarchy.

Moreover, the associativity of cache is also a problem when designing the structure of the cache. One associativity is the simplest design, but it may cause a large number of cache miss. Therefore, to avoid cache miss that induces the lower performance, two associativity were used in this project. However, it makes the cache design be more complicated and causes a lot of bugs and uncorrected connection.

* 1. **Memory Management Unit Design**
     1. **SRAM to DDR component**

After testing the SRAM to DDR component provided by Digilent. There were two main problems. The one was that after read and write to a location many times, the data in such memory location would lose. The other one was that reading and writing memory were too slow to enable the processor to have expected performance.

For the first problem, the ILA IP core showed that SRAM to DDR component outputted the right waveform. Therefore, one of the possible reasons was the charge loss in DDR2 SDRAM. This component doesn’t set the DDR2 memory fresh rate and according to the RowHammer problem, after several reading and writing, the data in memory may be lost [8].

The reason for the other problem was that this component didn’t take performance into consideration as the Digilent forum mentioned [16]. Hence, 210ns reading period and 260ns writing period could not be solved if using such component. However, it was not acceptable for an expected 100MHz CPU.

* + 1. **Memory Interface Generator**

Memory Interface Generator (MIG) was an IP core provided by Xilinx, which can automatically generating an memory interface for the FPGA board. However, the option of this FPGA board was difficult to find. Digilent official website only provided parts of the option for this DDR2 SDRAM component and other option should be tested or be necessary to read many documents.

Furthermore, the interface part of MIG IP core could use regular RAM interface or AXI4 interface, but it is difficult to understand what actually these two interfaces work.

* + 1. **Advanced eXtensible Interface 4 (AXI4)**

The first problem is how AXI4 worked. It is necessary to read several hundred pages in UG1037 to understand how AXI4 protocol transfer data between two components [5]. In addition, there exists a lot of ports in AXI4 protocol on Memory Management Unit AXI4 slave interface, which should be distinguished clearly when designing AXI4 connection.

* 1. **Test**

One of the most important problem is test problem. It is because tests may take much time to validate the correction of a component that the time spending on tests was almost the same as design. However, there were still many components that were not tested entirely or even not tested. Furthermore, there is a complicated verification process about RISC-V System on Chip and it is too difficult to follow in this project.

* 1. **Debug problem**

Debug in Vivado is not as easy as that in software development. The first debug way is to simulate the Verilog code and observe all the signal in the simulation. However, for a relatively large system, it is inefficient to find debug by eyes. Therefore, debug in this project causes huge time waste. The other debug way is to use ILA core when testing the Verilog code on board. Nevertheless, using ILA core will be more difficult than testing by simulation because it is not simple to let the core show what the designer wants to know.

* 1. **Project management and time management**

The project management was difficult and complex because students didn’t know how long it would take for students to finish their parts. Additionally, there were too many unknown problems for this project. The most serious problem that affected the progress in System on Chip part was SRAM to DDR component. To replace this component, students had to use two weeks, one-third of the whole project, to read relative reference books and documents and then started to design. It also caused the project to stop at constructing instead of integrating all the parts including operating system, hardware and boot program. Due to no General Purpose Input/Output (GPIO) and not enough static RAM on FPGA board, there was no way to load the xv6 operating system into the memory so that the group could not start the final test. Six weeks also asked the students to strictly follow the plan of the project with no extra time for the risk of difficult problems, but the reality was that the expectation would not happen at most of the time.

1. **Improvement**
   1. **Current bugs and their improvement**

|  |  |
| --- | --- |
| Bugs | Solution |
| Data cache uses single port LUT memory (cannot read and write at the same time) | Replace single port memory with dual port distributed memory in XPM |
| When cache hit and the block is dirty, current logic circuits could not correctly send a request to store the data in block. | Add an extra port for the previous data in the block so that the request can include correct data. |
| No AXI4 design in AXI4 master component | Construct new logic circuit in this component |
| Cache can not response when it fetches the block that CPU wants. | Add more logic circuit for updating the condition of cache to CPU |
| CPU should request again if the cache doesn’t have the data in the memory that CPU request to. | When writing back to cache, the data could also be pushed to the CPU through bus. |

* 1. **CPU core Improvement**

Since the current CPU core is in-order and scalar, the improvement can change the structure of the core. The expected change is out-of-order execution and superscalar processor. Out-of-order execution needs a scheduler and a reorder buffer to keep the instruction order. It can lower the penalty when memory operation needs to stall the pipeline because it can enable another instruction that doesn’t depend on the current instruction to be executed. Two scalar processor can be better in this project because superscalar processor can execute multiple instructions at the same time and two scalar processor will not increase too much complexity.

Branch prediction is also an important part about the performance of CPU core. In this project, to simplify the design, static branch prediction is used; however, dynamic branch prediction will be better to apply. Combining local branch prediction and global branch prediction could predict the loop and if condition better [6]. In addition, Branch Target Buffer (BTB) can also be used to increase the performance when executing branch instruction.

Interrupt controller is also the one that operating system requires. All the I/O and error needs interrupt controller to stop the processor and work all them. System call in Operating System is to use *ECALL* instruction to inform the CPU to enter kernel mode such as reading from I/O device.

* 1. **Cache Improvement**

Due to time limitation, there was less tests on cache part so there were many bugs as mentioned above. Thus, the fixation of bugs was the most essential improvement. In addition, the hierarchy of cache could be more separated, which means that there could be L1 cache and L3 cache instead of only L2 cache. On the other hand, cache latency should be taken into consideration. Most importantly, tests should be increased for cache part because it would be more difficult than simulating on the Vivado because designer could only know what happens when they generating bitstream into the FPGA board and use ILA core to observe the signal in the digital circuit.

* 1. **System on chip**

In this project, many parts of system on chip are not completed. General Purpose Input/Output (GPIO), PS/2 Driver, VGA Driver, Ethernet Driver, USB Driver, UART/JTAG controller. To enable the project to be the same as expectation, GPIO, VGA, PS/2, and UART are necessary because GPIO can help designer load the program into the memory and UART driver can keep the boot program and boot the operating system. VGA and PS/2 Driver are for exhibition and these two parts can allow the keyboard and screen to work successfully. All the components will be connected by AXI4 FULL protocol.

* 1. **Project management Improvement**

Since the project is a system construction project, the work division in the project can be better. Currently, one designs all the system part, one designs operating system, another transplants uboot onto the system on chip. However, this work division makes the student who designs hardware affords huge workload, which causes the System on Chip part to be delayed as long as some unexpected problem happens. Therefore, an improvement is that two students can design the hardware, which can be separated that one student focus on CPU core and the other one can construct the peripheral on the System on Chip. All the software part can be constructed by one student because it can use existing operating system and boot program, and just needs to transplant them into the proper form.

Time management can also be improved for such a complicated system. Due to only six weeks normal period, more preparation should be made before the starting time. For example, the instruction set determination cab be ensured before starting the project. Additionally, the expectation of the project should also be lowered to fulfill the time limitation such as transplanting ucore instead of xv6 operating system.

* 1. **Ethical, Security, Intellectual, Commercial aspects**

Due to the open-source character of RISC-V, only IP core provided by Xilinx may have some limitation for using. For example, if using IP core for commercial usage, designer should first apply a license from Xilinx. Otherwise, the designer may violate the regulations of Xilinx.

Security for this project was a large problem but not necessary for current progress. For example, for DDR2 SDRAM, RowHammer problem can be one of the computer hardware security problems [8].

RISC-V performs well in embedded system industry; thus, if the power efficiency of the SoC in this project could be optimized, commercial usage may be possible, but it would also be a difficult problem to weigh between the performance and power consumption.

* 1. **Sustainability**

This project uses FPGA as the development board. However, FPGA can be regarded as a perfect development tool for design instead of manufacturing because the resource usability is low for FPGA.

* 1. **Future of the technology**

RISC-V was created by University of California, Berkeley, and now it has been invested by many IC companies. Since RISC-V should not take compatibility of old software and hardware into account, the instruction set can be as few as possible and the structure of RISC-V was also simple than MIPS and ARM processor. Therefore, RISC-V can have better resource usability and as it is open-source, developers can design what they need so that the development community of RISC-V is welcomed. Most importantly, Internet of things may be one of the future technologies and RISC-V, which is processor who is an open-source, lower power assumption, and with highly developed community, were likely to be welcomed in such embedded system industry.

**Reference**

[1] D. A. Patterson, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1 edition ed. Morgan Kaufmann, 2017.

[2] D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[3] S. Patel and Y. Patt, *Introduction to Computing Systems: from bits & gates to C & beyond*. McGraw-Hill Professional, 2019.

[4] X. Inc., "Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2 User Guide (UG586)." [Online]. Available: <https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_2/ug586_7Series_MIS.pdf>

[5] X. Inc., "Vivado Design Suite AXI Reference Guide (UG1037)." [Online]. Available: <https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf>

[6] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, 1993.

[7] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Transactions on Electronic Computers,* no. 2, pp. 270-271, 1965.

[8] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 2019.

[9] A. Waterman, K. Asanovic, and S. Inc., "The RISC-V instruction set manual, volume I: Unprivileged ISA," in *Document Version 20191213*, 2019.

[10] O. U. ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, "Notes - Unit 5." [Online]. Available: <http://www.secs.oakland.edu/~llamocca/Courses/F16_ECE495/Notes%20-%20Unit%205.pdf>

[11] "NonTrivial-MIPS." <https://github.com/trivialmips/nontrivial-mips> (accessed.

[12] "lowrisc-chip." <https://github.com/lowRISC/lowrisc-chip> (accessed.

[13] "riscv\_soc." <https://github.com/ultraembedded/riscv_soc> (accessed.

[14] D. Inc. "Nexys A7 Reference Manual." <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual> (accessed.

[15] X. Inc. "Community Forums." <https://forums.xilinx.com/> (accessed.

[16] D. Inc. "Forum." <https://forum.digilentinc.com/?_ga=2.166305335.861222769.1586779602-1437218926.1586248521> (accessed.

[17] O. Mutlu. "Design of Digital Circuits - Spring 2019." <https://safari.ethz.ch/digitaltechnik/spring2019/doku.php?id=schedule> (accessed.

**Appendix A**

**Description of component Input/Output in this project**

1. **CoreTop.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| CLK (input) | The clock of the processor |
| RESET (input) | Reset the processor (all register goes to zaro) |
| DATA [31:0] (output) | Used for debug |
| ADDRESS [31:0] (output) | Used for debug |

1. **InstructionMemory.v** (It is used temporarily and will be replaced by InstructionCache.v)

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| A [5:0] (input) | Address of instruction |
| RD [31:0] (output) | Instruction read from memory |

1. **IF\_ID\_Register.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| PC [31:0] (input) | Program Counter |
| CLK (input) | The clock of the processor |
| RESET (input) | Register goes to zero |
| Instr [31:0] (input) | The instruction read from memory/cache |
| flush (input) | Used when the pipeline needs to stall |
| PC\_o [31:0] (output) | Output of program counter |
| Instr\_o [31:0] (output) | Output of instruction |

1. **ControlUnit.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| OP [6:0] (input) | Opcode of instruction (Instruction[6:0]) |
| Funct3 [2:0] (input) | Funct3 number of instruction |
| Funct7 [6:0] (input) | Fucnt7 number of instruction |
| EX\_control [20:0] (output) | Use to control execution stage |
| MEM\_control [6:0] (output) | Use to control Memory operand fetch |
| WB\_control [3:0] (output) | Use to control Write Back stage |

1. **RegisterFile.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| A1 [4:0] (input) | Register number in rs1 |
| A2 [4:0] (input) | Register number in rs2 |
| A3 [4:0] (input) | Register number needed to write back |
| WD [31:0] (input) | Write Data |
| RegWrite (input) | Write Enable |
| CLK (input) | The clock of the processor |
| RESET (input) | Register goes to zero |
| RD1 [31:0] (output) | Data in rs1 |
| RD2 [31:0] (output) | Data in rs2 |

1. **ID\_EX\_Register.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| CLK (input) | The clock of the processor |
| RESET (input) | Register goes to zero |
| Enable (input) | Write Enable (pipeline stall) |
| flush (input) | Register goes to zero (branch target hit) |
| SrcA\_i [31:0] (input) | Data in rs1 input |
| SrcB\_i [31:0] (input) | Data in rs2 input |
| EX\_control\_i [20:0] (input) | Execution control input |
| MEM\_control\_i [6:0] (input) | Memory control input |
| WB\_control\_i [3:0] (input) | Write Back control input |
| U\_type\_immediate\_i [31:0] (input) | U-type instruction immediate value input |
| J\_type\_immediate\_i [31:0] (input) | J-type instruction immediate value input |
| I\_type\_immediate\_i [31:0] (input) | I-type instruction immediate value input |
| B\_type\_immediate\_i [31:0] (input) | B-type instruction immediate value input |
| S\_type\_immediate\_i [31:0] (input) | S-type instruction immediate value input |
| RegDst\_i [4:0] (input) | Register destination (rd) input |
| PC\_i [31:0] (input) | Program counter input |
| ALUSrcB\_S\_type\_i (input) | Control SrcB if the instruction type is S-type |
| RegisterRs1\_i [4:0] (input) | Register rs1 (use for forwarding) |
| RegisterRs2\_i [4:0] (input) | Register rs2 (use for forwarding) |
| EX\_control [20:0] (output) | Execution control output |
| MEM\_control [6:0] (output) | Memory control output |
| WB\_control [3:0] (output) | Write Back control output |
| U\_type\_immediate [31:0] (output) | U-type instruction immediate value output |
| J\_type\_immediate [31:0] (output) | J-type instruction immediate value output |
| I\_type\_immediate [31:0] (output) | I-type instruction immediate value output |
| RegDst [4:0] (output) | Register destination (rd) output |
| PC [31:0] (output) | Program counter input |
| SrcA [31:0] (output) | Data in rs1 input |
| SrcB [31:0] (output) | Data in rs2 input |
| B\_type\_immediate [31:0] (output) | B-type instruction immediate value input |
| S\_type\_immediate [31:0] (output) | S-type instruction immediate value input |
| RegisterRs1 [4:0] (output) | Register rs1 (use for forwarding) |
| RegisterRs2 [4:0] (output) | Register rs2 (use for forwarding) |
| ALUSrcB\_S\_type (output) | Control SrcB if the instruction type is S-type |

1. **ALUControl.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| Aluop [6:0] | Opcode of instruction (Instruction[6:0]) |
| Funct3 [2:0] | Funct3 number of instruction |
| Funct7 [6:0] | Fucnt7 number of instruction |
| ControlResult [10:0] | Alu control logic |

1. **ALU.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| a [31:0] (input) | Input 1 |
| b [31:0] (input) | Input 2 |
| aluop [10:0] (input) | ALU control logic |
| result [31:0] (output) | Calculation result |
| branchCmp (output) | Branch comparison result |
| zero\_division (output) | Divide zero |
| overflow\_signed\_div (output) | Division overflow |

1. **EX\_MEM\_Register.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| CLK (input) | The clock of the processor |
| RESET (input) | Register goes to zero |
| MEM\_control\_i [6:0] (input) | Memory control input |
| WB\_control\_i [3:0] (input) | Write Back control input |
| ALUResult\_i [31:0] (input) | ALU calculation result |
| StoreData\_i [31:0] (input) | Data needed to be stored in memory (s-type instruction) |
| branchCmp (input) | Branch comparison result |
| zero\_division (input) | Divide zero |
| overflow\_signed\_div (input) | Division overflow |
| RegDst\_i [4:0] (input) | Register destination (rd) input |
| PC\_i [31:0] (input) | Program counter input |
| BranchTargetAddress\_i [31:0] (input) | Branch Target Address |
| WB\_control [3:0] (output) | Write Back control output |
| ALUResult [31:0] (output) | ALU calculation result |
| StoreData [31:0] (output) | Data needed to be stored in memory (s-type instruction) |
| branchCmp (output) | Branch comparison result |
| zero\_division (output) | Divide zero |
| overflow\_signed\_div (output) | Division overflow |
| RegDst [4:0] (output) | Register destination (rd) output |
| PC [31:0] (output) | Program counter output |
| MEM\_control [6:0] (output) | Memory control output |
| BranchTargetAddress [31:0] (output) | Branch Target Address |

1. **DataMemory.v** (It is used temporarily and will be replaced by DataCache.v)

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| CLK (input) | The clock of the processor |
| A | Address of reading memory |
| WE | Write Enable |
| WD | Write Data |
| RD | Read Data |

1. **MEM\_WB\_Register.v**

|  |  |
| --- | --- |
| Input/output name | Input/output function |
| CLK (input) | The clock of the processor |
| RESET (input) | Register goes to zero |
| WB\_control\_i [3:0] (input) | Write Back control input |
| ReadData\_i [31:0] (input) | Data read from memory |
| ALUResult\_i [31:0] (input) | ALU calculation result |
| PC\_i [31:0] (input) | Program Counter of current instruction |
| WB\_control [3:0] (output) | Write Back Control logic |
| RegDst [4:0] (output) | Register Destination |
| ReadData [31:0] (output) | Data read from memory output |
| ALUResult [31:0] (output) | ALU calculation result output |
| PC [31:0] (output) | Program Counter of current instruction output |
| RegDst\_i [4:0] (input) | Register Destination output |

1. **HazardUnit.v**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| RegSrcA\_ID\_EX [4:0] (input) | Register rs1 in execution stage |
| RegSrcB\_ID\_EX [4:0] (input) | Register rs2 in execution stage |
| RegDst\_EX\_MEM [4:0] (input) | Register Destination in memory operation stage |
| RegDst\_MEM\_WB (input) | Register Destination in write back stage |
| RegWrite\_EX\_MEM (input) | Register Write control in memory operation stage |
| RegWrite\_MEM\_WB (input) | Register Write control in write back stage |
| ALUSrcA (input) | ALU Source A control logic |
| ALUSrcB [1:0] (input) | ALU Source B control logic |
| MemRead (input) | Determine whether the instruction in memory operation stage needs to read memory |
| RegDst\_ID\_EX [4:0] (input) | Register Destination in execution stage |
| RegSrcA\_IF\_ID [4:0] (input) | Register rs1 in decode stage |
| RegSrcB\_IF\_ID [4:0] (input) | Register rs2 in decode stage |
| ForwardA [1:0] (output) | Determine whether ALU Source A needs data forwarding |
| ForwardB [1:0] (output) | Determine whether ALU Source B needs data forwarding |
| WriteEnable\_IF\_ID (output) | Control the Write Enable of IF/ID register |
| ControlLogicStall (output) | Make control logic input zero |
| PCWriteEnable (output) | Control the PC to be next PC value or to stall |

1. **Cache System (Block Design)**

|  |  |
| --- | --- |
| **Input/output name** | **Input/output function** |
| w\_valid\_d (input) | CPU request write data into memory |
| r\_valid\_d (input) | CPU request read data from memory |
| w\_data\_d [31:0] (input) | The data that needed to be written |
| CLK (input) | The Cache Clock (three times as Processor clk) |
| r\_addr\_i [31:0] (input) | The address of which CPU wants to read instruction |
| FIFO\_empty\_i (input) | The empty port of FIFO between CPU and instruction cache |
| addr\_d [31:0] (input) | The address of which CPU wants to read or write data |
| FIFO\_empty\_d (input) | The empty port of FIFO between CPU and data cache |
| reset (input) (input) | Reset cache |
| r\_valid\_i (input) | CPU request read instruction from memory |
| rsp\_d [1:0] (output) | Response to CPU from data cache |
| r\_data\_d [31:0] (output) | Data read from memory |
| rsp\_valid\_d (output) | Response valid port from data cache |
| r\_data\_i [31:0] (output) | Instruction read from memory |
| r\_rsp\_i [1:0] (output) | Response to CPU from instruction cache |
| rsp\_valid\_i (output) | Response valid port from instruction cache |
| DDR2 (output) | DDR2 port connection to physical memory |