**Week 1**

1. Discuss what the expected demo of the project with group members
2. Discuss what the project needs to do (general construction of separate part)
3. Search reference of the system on chip and CPU core
   1. Reference Book: Computer Organization and Design RISC-V edition, Digital Design and Computer Architecture, Introduction to Computing System
   2. Reference website or report: ["Longxin" Cup final project](https://github.com/trivialmips/nontrivial-mips), [lowRISC](http://modernhackers.com/build-your-own-risc-v-architecture-on-fpga/) (constructed by University of Cambridge), [Xlinx forum](https://forums.xilinx.com/), [Digilent forum](https://forum.digilentinc.com/), [Design of Digital Circuit](https://safari.ethz.ch/digitaltechnik/doku.php) (course of ETH Zurich)
4. Test the function of Digilent Nexys A7 board
5. Create the [github](https://github.com/dingqy/ELEC222-Project) page and determine the time table of separate parts

**Week 2**

1. Discuss which instruction set the operating system (xv6 or Linux kernel) needs (RV32IM)
2. Construct the simple Arithmetic Logic Unit Structure on Vivado 2019.1 ([commit detail](https://github.com/dingqy/riscv-core/commit/01af63b6e616400fa81630cb02f4b011370f3002))
3. Update ALU structure and support RV32M extension without verification ([commits on Feb 8, 2020](https://github.com/dingqy/riscv-core/commits/master))
4. Update ALU Control Unit without verification ([commits on Feb 8, 2020](https://github.com/dingqy/riscv-core/commits/master))
5. Construct register file (32 registers following RISC-V specification)

**Week 3**

1. Construct Control Unit, and test ALU control and register file ([commits on Feb 10, 2020](https://github.com/dingqy/riscv-core/commits/master))
2. Test the ALU with new extension support and fix the bugs ([commits on Feb 9, 2020](https://github.com/dingqy/riscv-core/commits/master))
3. Confirm the function of [SDRAM to DDR Component](https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-sram-to-ddr-component/start?_ga=2.78907863.2105935891.1581160391-832371281.1575559289) provided by Digilent on board ([commits on Feb 12, 2020](https://github.com/dingqy/riscv-core/commits/master))
   1. Advantages: It can be directly used to save time
   2. Disadvantages: minimum writing time is 260ns and reading time is 210ns, which causes huge memory operation latency. In addition, it doesn’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 will obviously lose data (maybe caused by refresh rate of DDR2 SDRAM in this FPGA)
4. Construct register between each stage used in pipeline processor
5. Test Control Unit and fix the bugs ([commits on Feb 15, 2020](https://github.com/dingqy/riscv-core/commits/master))

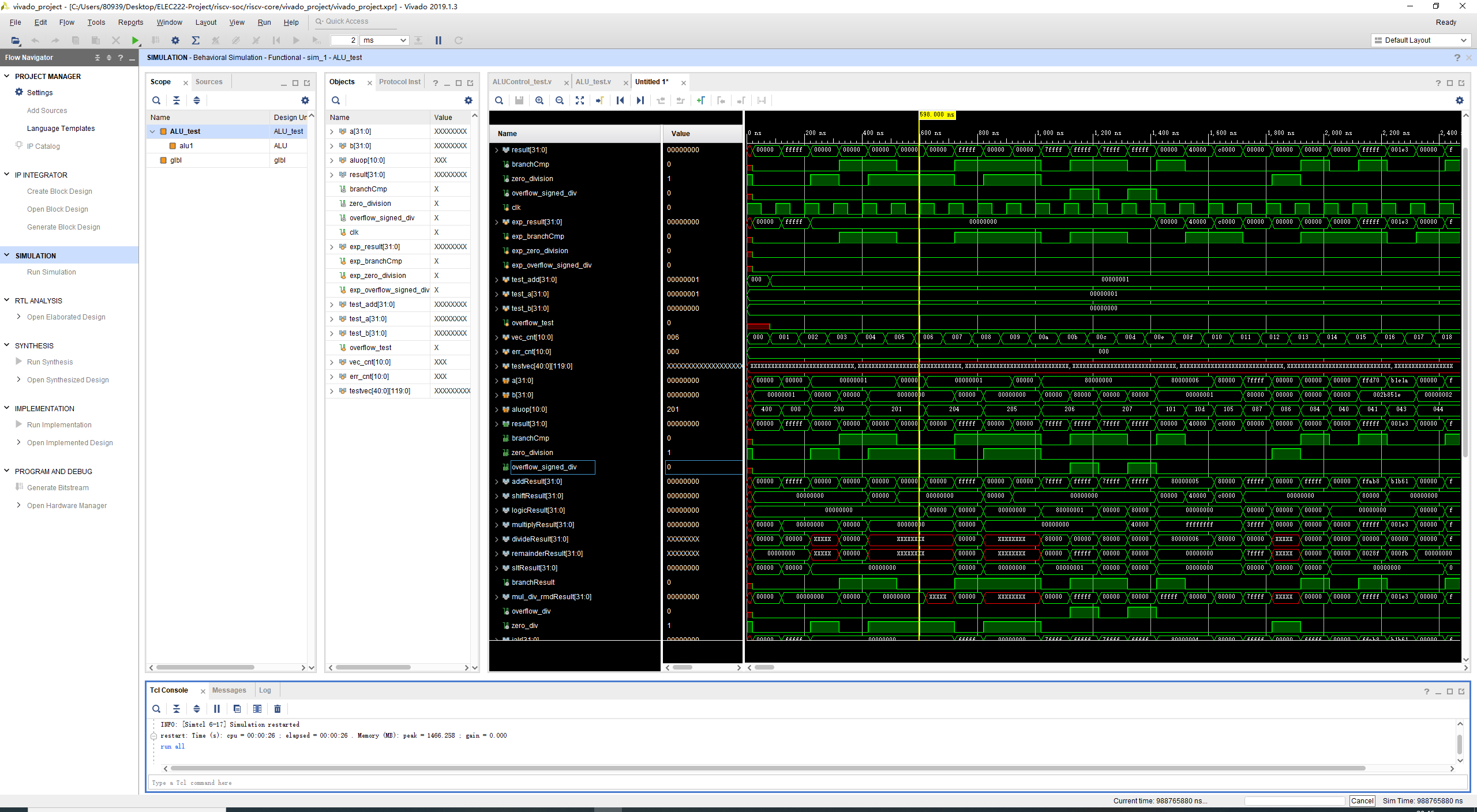


Figure 1. Test of ALU

**Week 4**

1. Discuss how the operating system communicate with the CPU (system call)
2. Search information to solve the problem of DDR2 SDRAM, which cause the whole project not to meet the expectation progress.
3. 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

**Week 5**

1. Integrate all the components to be a pipeline processor with 5 stages ([commits on Feb 27, 2020](https://github.com/dingqy/riscv-core/commits/master))
2. Construct Hazard Unit to solve control dependencies and data dependencies by data forwarding and pipeline stall ([commits on Feb 28, 2020](https://github.com/dingqy/riscv-core/commits/master))
3. Construct Static branch prediction ([commits on Feb 28, 2020](https://github.com/dingqy/riscv-core/commits/master))

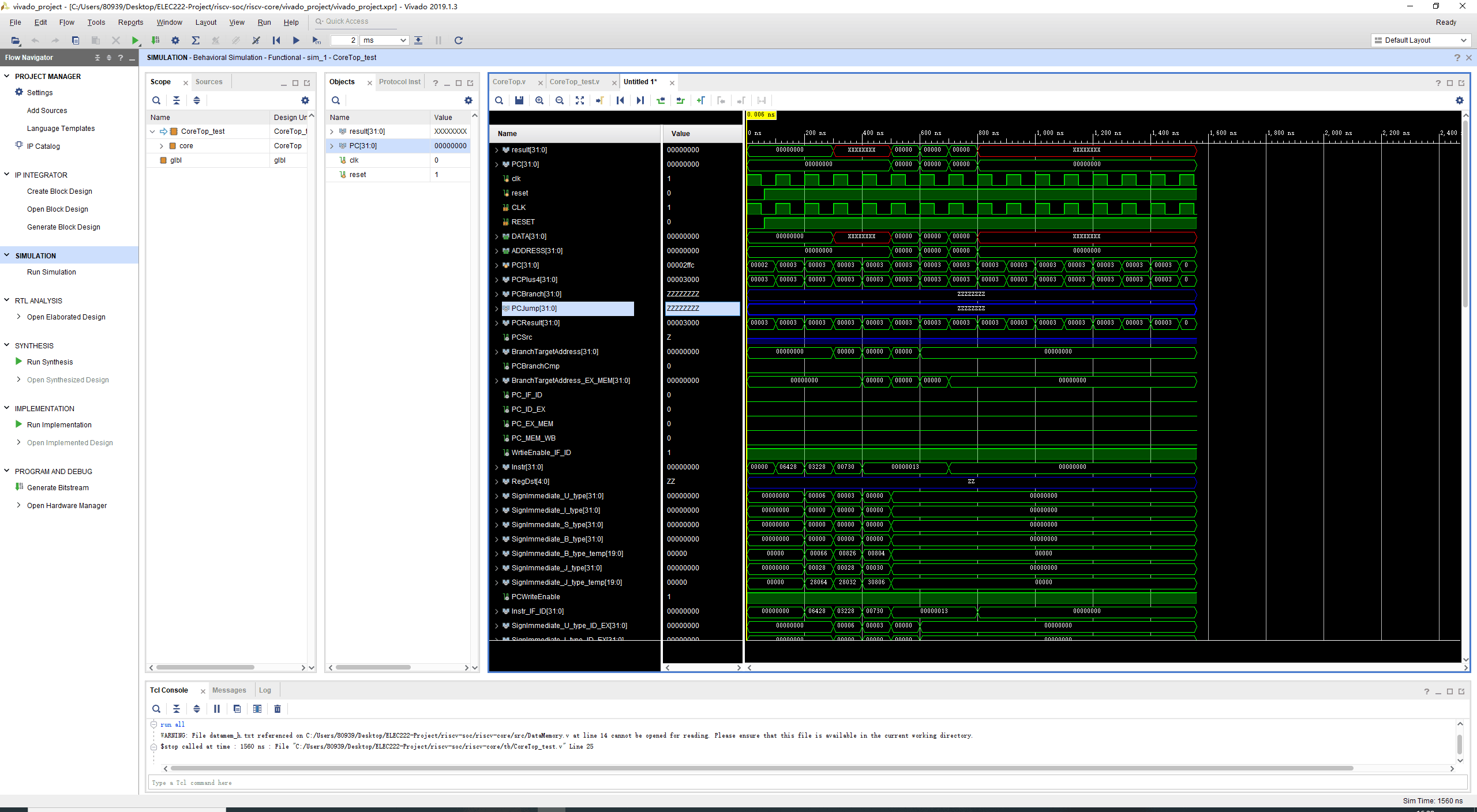


Figure 2. Test of Core

**Week 6**

1. Construct Cache System ([commits on Mar 2, 2020](https://github.com/dingqy/riscv-core/commits/master))
   1. Instruction cache: Single Port Distributed RAM (Xilinx Parameterized Macro)
   2. Data cache Single Port Block RAM (Xilinx Parameterized Macro)
2. Construct communication between cache system and memory by AXI4 full bus ([commits on Mar 3, 2020](https://github.com/dingqy/riscv-core/commits/master))
   1. Reference document: [UG586 MIG 4.2 User Guide](https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_2/ug586_7Series_MIS.pdf), Note 5 of ECE-495/595 in ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT, OAKLAND UNIVERSITY
   2. Tools: Vivado Block Design
   3. IP core: Xilinx Memory Interface Generator, Xilinx AXI4 peripheral, Xilinx AXI4 interconnect
3. Construct communication between cache and cpu core ([commits on Mar 3, 2020](https://github.com/dingqy/riscv-core/commits/master))
   1. Asynchronous FIFO (Xilinx Parameterized Macro)
4. Try to integrate operating system into the SoC
   1. Problem: No GPIO, No USB, Cache system have unknown bugs, boot can not correctly load the system

**Further improvement**

1. System on Chip
   1. General-purpose input/output (GPIO) to AXI4 interface (memory)
   2. USB driver to AXI4 interface (memory)
   3. PS/2 and VGA driver to AXI4 interface (memory)
   4. Complete verification of all the components
2. CPU Core
   1. Reorder buffer and Out of order execution
   2. Dynamic branch prediction
      1. Reference document: S. McFarling, "Combining Branch Predictors" DEC WRL Technical Report" 1993
   3. Superscalar processor

Notice: Other simulation figures will not be showed and it all exists on the github repository; thus, if user want to see the simulation figure, it can directly run behavioral simulation in Vivado with files in “tb” file folder.

**Wiki**

1. **CPU core**
   1. **CPU core diagram**

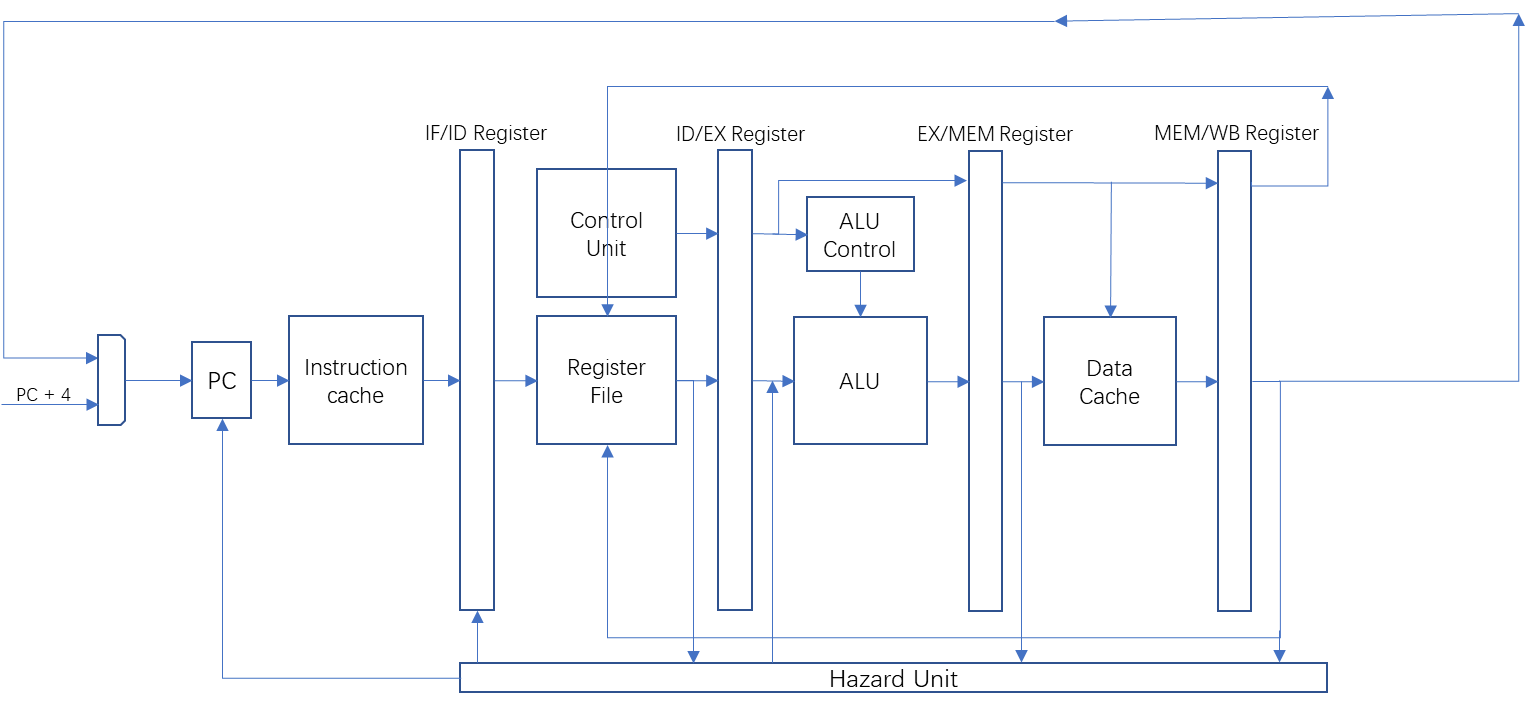


Figure 1. The simple diagram of CPU core in the project

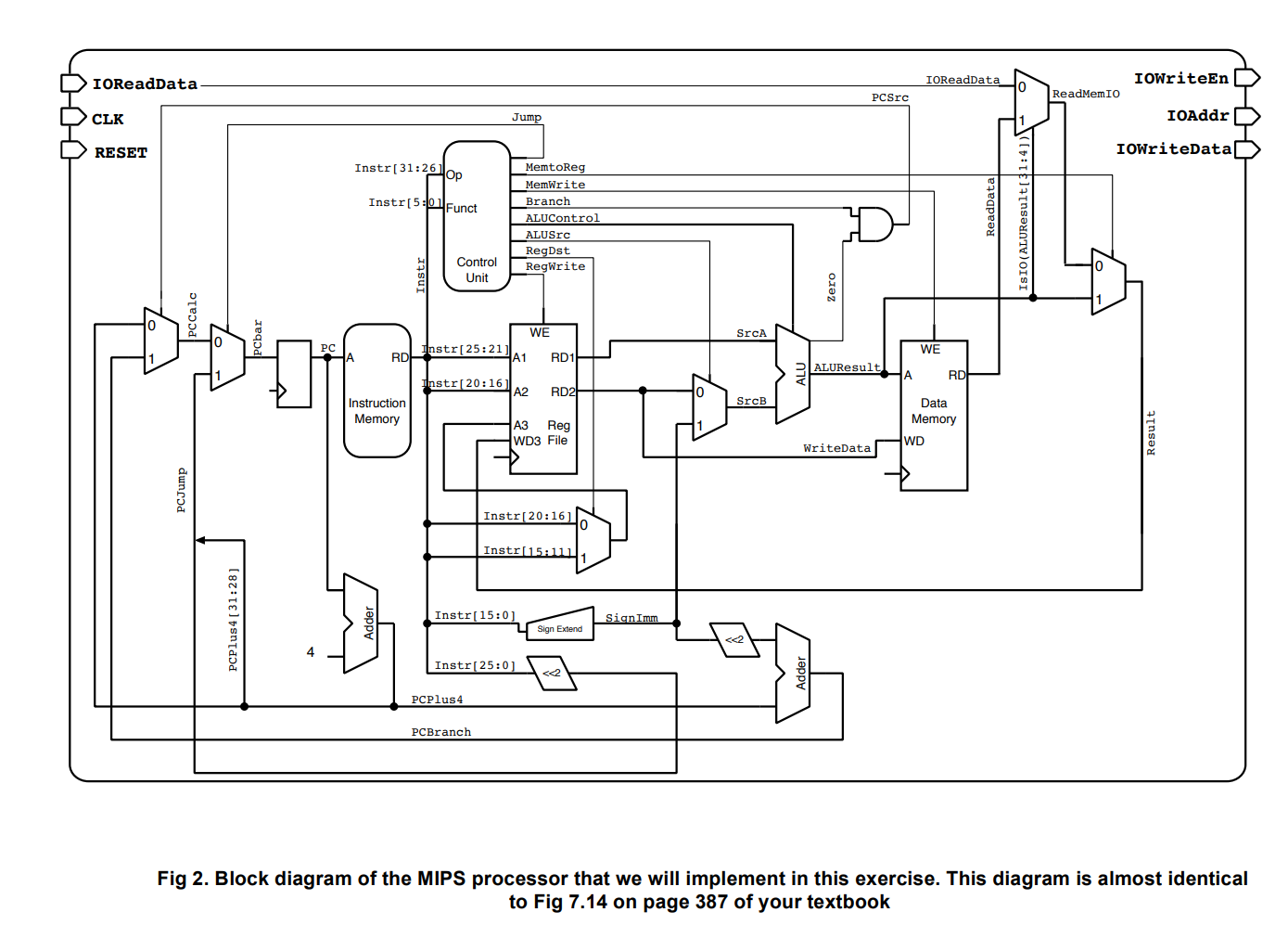


Figure 2. CPU structure in Lab of Design of Digital Circuit (ETHZ)

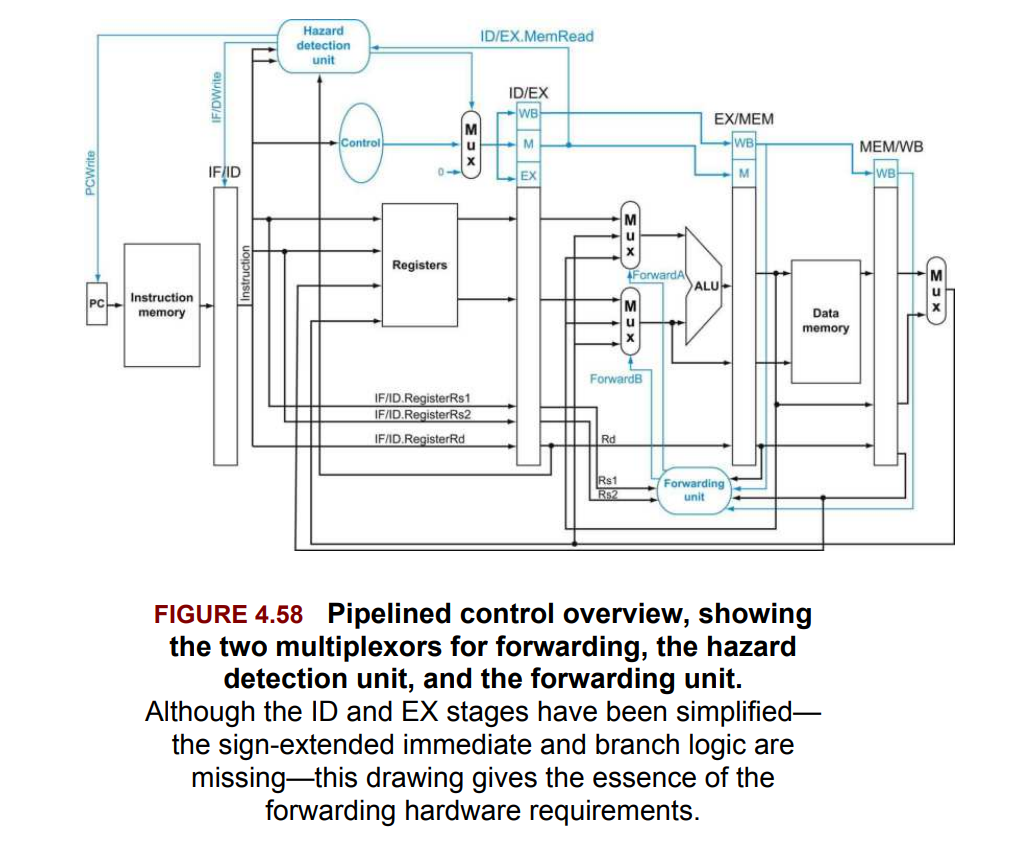


Figure 3. The structure diagram on Computer Organization and design RISC-V edition

The first diagram is the simple representation of the CPU core in this project and the detail information will be exhibited in the following sections. The latter two figures are referenced when designing the CPU core in this project but these two design is much more different from the one in the project.

* 1. **Implementation**
     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. **Brief introduction about each verilog file**
     1. **CoreTop.v**

This top file includes all the connection between each components in the 5 stage pipeline processor. In the instruction fetch stage, Instruction memory/cache will return the value whose address is program counter in one or more cycles and then push them into the IF/ID register. Then the instruction will come into the next stage, decode stage. In this stage, the instruction will separate the instruction into different parts according to the RISC-V specification and also push them into ID/EX register. Next, if the instruction needs to do some calculation, it will be finished in the execution stage. It will also be pushed into EX/MEM register for memory operation stage. Finally, Write Back stage will happen if it is necessary.

* + 1. **InstructionMemory.v**

This file is the temporary memory for instruction cache. It only uses register in FPGA to store instruction.

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

This file includes all the information the decode stage needs and it will store new information on rising edge of the clock.

* + 1. **ControlUnit.v**

This file provides control logic based on the instruction and it will be transmitted every cycle until the instruction is completed.

* + 1. **RegisterFile.v**

This file includes 32 registers to store information and it supports synchronous read and write operation at the same rising edge of cycle.

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

This file includes all the information the execution stage needs and it will store new information on rising edge of the clock.

* + 1. **ALUControl.v**

This file will further deal with the information constructed by control unit and help the ALU generate the current calculation result.

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

This file includes all the information the memory operation stage needs and it will store new information on rising edge of the clock.

* + 1. **DataMemory.v**

This file is the temporary memory for data cache. It only uses register in FPGA to store instruction.

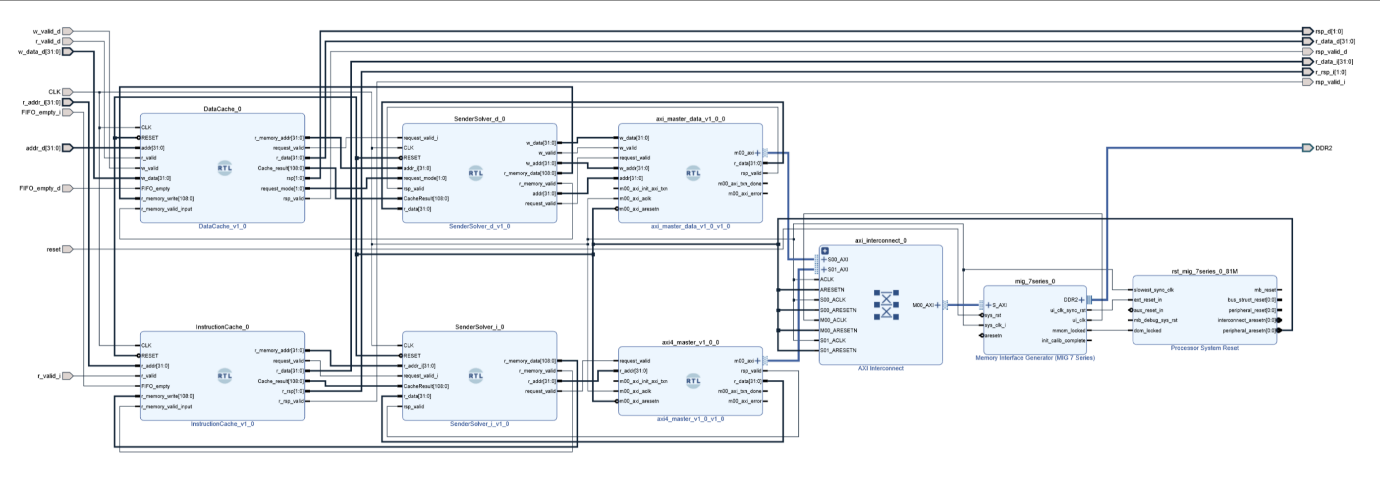
* + 1. **MEM\_WB\_Register.v**

This file includes all the information the write back stage needs and it will store new information on rising edge of the clock.

* + 1. **HazardUnit.v**

Hazard Unit deal with the control and data dependencies in the pipeline. If the instruction in decode stage needs register values that will be written by the instruction in memory operation stage or write back stage, the value in the latter two stages can be forwarded to the former instruction so that the pipeline doesn’t need to be stalled. If the instruction after load instruction read register value that is the same as the destination register of load instruction, it should be stalled until the load instruction comes into write back stage.

1. **System on chip**
   1. **Cache System**
      1. **Cache System diagram**



* + 1. **Implementation**

|  |  |
| --- | --- |
| **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 |

* + 1. **Brief introduction of the function of cache system**

To begin with, CPU will put its request on the asynchronous FIFO between cache and CPU so that cache will fetch it in the next cycle of cache. Next, cache includes three stage pipeline that are read from cache, comparison, and response. In the reading stage, it will read the set based on 2 to 11 bits and then push them into the register. Then the comparison logic will compare the tag, the first 20 bits of address. If the tag is the same, it means cache hit and it will do write or read operation in the final stage. Otherwise, cache will request SenderSolver to further deal with address. Moreover, cache use least recently used (LRU) policy to replace the block that is not needed and write back policy to write dirty block into memory. After that, SenderSolver will encapsulate information necessary to read or write memory and send it to the AXI master. AXI master is responsible for sending the address and receive data from memory through AXI4 bus. Instruction cache and Data cache will use AXI4 interconnect to communicate with AXI4 slave interface of Memory Interface Generator. Furthermore, the result will be sent by AXI4 bus and then dealt with by SendSolver to write back into the cache. The cache will send messages to ask the CPU send request again and finally, CPU will get the correct result.

* + 1. **Memory Interface Generator Settings**

