**P4 Verilog CPU Design Instructions**

# **Modular and Hierarchical Design**

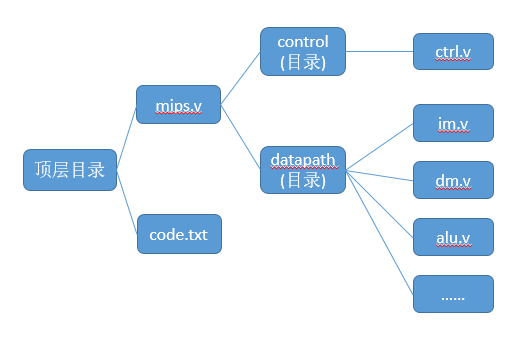
## **Overview**

         The hierarchical and structured design of complex digital logic circuits and systems implies a sequential decomposition of the system hardware design. Although the circuit design is not complicated, we hope that you can learn more through the practice of this design method, and this is very helpful for the completion of your entire project, not only saving time but also understanding more deeply. These are also very helpful for class testing. In the process, we also hope that students can constantly improve their CPU design documents, which is our guide for circuit construction. It is worth mentioning that we will check the contents of the document (design documents, thinking questions) under the class and in the class test, please be prepared!

## **The basic idea**

        In P3 you have designed the circuit diagram of the MIPS single-cycle processor, which is very helpful for the completion of P4. All you have to do is complete the mapping from the logisim circuit to the Verilog code.

        First, referring to the circuit diagram of the MIPS single-cycle processor designed in P3, the MIPS single-cycle processor designed by Verilog can be summarized as shown in the following figure: (The directory structure and file naming in the figure are for reference only)



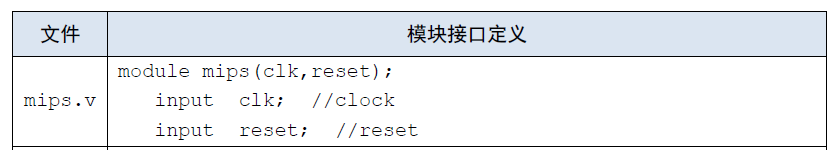
        The overall design has been completed, followed by specific modular and hierarchical design.

* The single-cycle processor includes the controller and data path and puts it into the mips.v hierarchy. The corresponding instruction code is stored in code.txt.
* The control module occupies a separate Verilog HDL file, which implements the single responsibility of the controller and keeps the module free from contamination.
* Each module in the datapath consists of a separate Verilog HDL file. This reduces the degree of coupling between the modules, reduces the influence and modification of other modules when adding corresponding functions or modifying the code, the regulations are clearer, and it is also very useful and convenient for subsequent designs.
* Multiple modules can be defined in a Verilog HDL file, so it is recommended that all mux (including all MUXs with different numbers of bits and different port counts) be modeled in a mux.v.

        The benefit of this design is that the entire project has excellent locality (including the modularity you understand, etc.). Even if it is a pc, although it is a very small module, you should do the same. Remember that the essentials of modularity are not the size of the module, but the independence of the function of the module.

## **Design Instructions**

* The processor should support the instruction set as: {**addu, subu, ori, lw, sw, beq, lui, jal, jr, nop**}.
* Addu, subu can not support overflow.
* The processor is designed for a single cycle.
* There is no need to consider delay slots.
* You need to construct a test set yourself to verify the correctness of the design. (Automatic testing through class does not mean that your design is completely free of problems)
* After each section, the explanatory content is accompanied by document writing suggestions and related thinking questions. Please complete the relevant thinking questions and attach them to the CPU design document!
* Finally, you need to submit the contents: CPU design document (including thinking questions), Verilog file packaged into .zip as the suffix format (all .v files are placed in the same folder and compressed and submitted, pay attention to the compression package size limit).
* Friendly Tip: Please read through all the contents of this Lab before designing to avoid unnecessary modifications!
* The top-level file is mips.v and the interface is defined as follows:



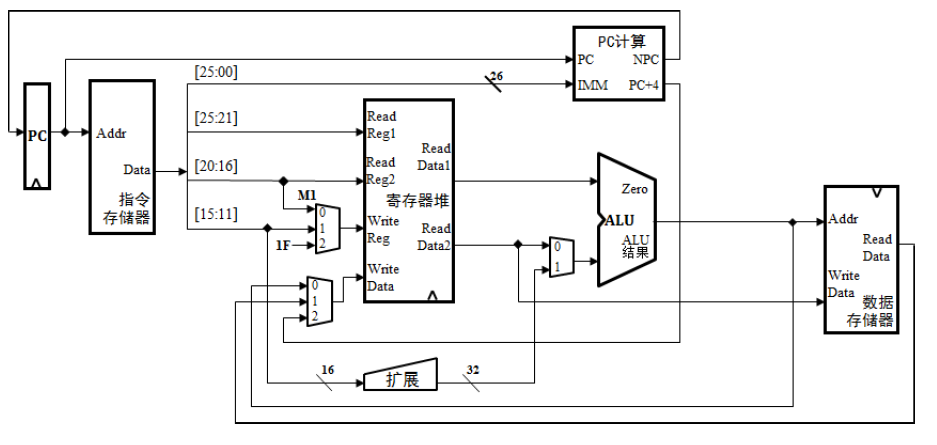
# **Data Path Design**

## **Overview**

        The idea of ​​overall design and modular design is explained above. The specific design of the data path is described below.

## **Design Flow**

        Since P3 has already completed the use of logisim to build a MIPS single-cycle processor, it is easy to draw a data path architecture diagram. The following figure is a reference data path architecture diagram:

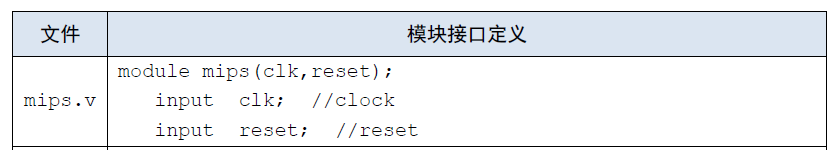


* This diagram is for reference only and there may be some errors, so you are encouraged to design a better data path architecture from the perspective of proper functional division of the data path.
* If you make a big adjustment, please don't contradict the modular and hierarchical design.

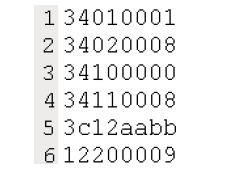
        With the data path architecture diagram, we started writing Verilog code. Because we have completed the design and implementation of GRF, ALU, and IM in the previous stages of learning and testing, the completion of other components is more skilled, and it is left to the students to complete on their own, but must comply with the following requirements:

* The main function of the PC is to finish outputting the current instruction address and save the next instruction address. After reset, the PC points to 0x0000\_3000, here the address of the first instruction. The purpose is to match the MARS Memory Configuration. The test program for the teacher will be generated by MARS, and its configuration mode is as shown below:

* The interface definitions for the following modules are available for reference:



* When creating an IM with Verilog HDL, the instructions in code.txt must be loaded into the IM by reading the file. It is not allowed to assign values ​​directly to the contents of IM in Verilog HDL code.
* The format of code.txt is shown below. Each instruction occupies 1 line, and the instruction hexadecimal code is stored in text mode.



* IM capacity is 4KB (32bit × 1024 words)
* DM capacity is 4KB (32bit × 1024 words)

        After each component has been completed, the next step is to hand over the components to the datapath for unified management, which is the bottom-up process.

        According to the data path architecture diagram of your design and the input and output ports of each component module, define some internal variables in datapath.v, and use the logical relationship between modules to connect the component modules together to make it a whole. Finally, the input and output ports of the controller signal are reserved.

# **IM Design**

## **Overview**

         The specific design method of the data path is mentioned above. The specific construction of the IM component will be described in more detail below.

## **Design Flow**

        In the previous project, we have used the ROM module in logisim to store the corresponding instruction as our IM. In the use of ISE, there is no such memory module that has been built for everyone to use, and we need to design it ourselves.

         In fact, the design of IM is not complicated, mainly to generate an array of 32-bit registers to store 32-bit instructions. The number of supported instructions directly determines the size of the array.

         After designing the register array, you need to read the instructions into the IM. Here, we use the **$readmemh** command to complete the corresponding functions. For specific usage, you can view the link: system task, system task test.

# **Controller Design**

## **Overview**

        The design of the controller is relatively simple compared to the data path. Verilog's code implementation logic is also simpler than the logisim circuit. As long as you clear your mind and find the logical relationship between the function of the part and the instruction, be careful not to miss it.

## **Design Flow**

        First, analyze the control signals obtained by each component from the controller, and enumerate them all. For example, the ALU needs the controller to output an ALUOp to control which operation (addition, subtraction, or the like) is performed by the ALU, but it is completed at P3. The layout of the controller circuit is arranged, so only the control signals generated by the newly added instructions need to be added here. This step is very simple for you who have already designed the P3, and there is no need to worry about the details.

        It is recommended to list a table, record the operation code of each instruction and the value of the corresponding control signal.

        This is very helpful for subsequent simulation debugging and class testing. Because during simulation debugging, you can first check whether the table is correct, and then go to the code to correspond to each other, the bug is clear at a glance. For adding instructions, it is very convenient to just add the new opcode and the corresponding control signal value to the back of the table and map it to the Verilog HDL code. As for the specific design of the form, it varies from person to person, and students can refer to each other.

        Finished the preparation work before writing the code, the following begins to enter the coding phase.

        Here are a few ways to write:

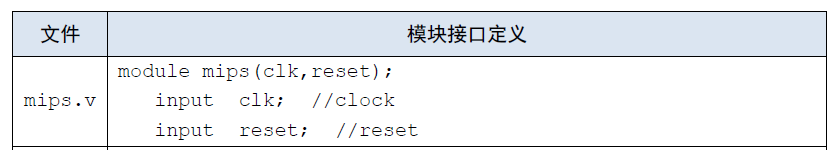
* Using if-else (or case) to complete the correspondence between the values ​​of the opcode and the control signal;
* Using the assign statement to complete the correspondence between the values ​​of the opcode and the control signal;
* Use macro definition
* ......

        Perhaps you have a better coding method to complete the design of the controller, which requires the students to explore and compare and discover.

# **Online Testing FAQ**

## **Submit request:**

    The following modules must strictly meet the following interface definitions:



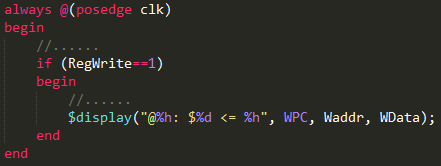
* This top-level module must be built in the Verilog HDL design and does not allow modification of the module name, port signals, and the name/type of the variable.
* In the grf module, if data is to be written when the rising edge of each clock arrives (that is, when the write enable signal is 1 and not reset), the written position and the written value are output in the format of

$display("@%h: $%d <= %h", WPC, Waddr, WData);

(Note the space), where WPC indicates the storage address of the corresponding instruction, starting from 0x0000\_3000; Waddr indicates the address of the input 5-bit write register; WData indicates the value of the input 32-bit write register. .

For example, if an instruction with address 0x0000\_3000 writes data 0 to register No. 3, it should output "@00003000: $3 <= 00000000", where 00003000 should be the 16-bit address of the instruction corresponding to the operation, and less than 8 bits need to be filled with zero. .

An example of adding an output statement is as follows:



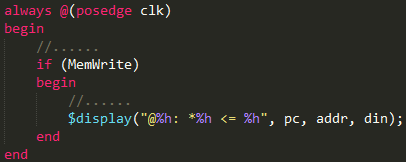
* In the dm module, if data is to be written when the rising edge of each clock arrives (that is, when the write enable signal is 1 and not reset), the written position and the written value are output in the format of

$display("@%h: \*%h <= %h",pc, addr,din);

(note the space). Where pc is the address of the instruction corresponding to the operation, which is consistent with the grf module requirement, addr indicates the 32-bit address to be stored in the data, and din indicates the value of the input 32-bit write dm.

For example, if an instruction with an address of 0x0000\_3004 is to write data 0 at the address 0x00001004, then "@00003004: \*00001004 <= 00000000" should be output.

An example of adding an output statement is as follows:



* You need to construct a test set yourself to verify the correctness of the design. (Automatic testing through class does not mean that your design is completely free of problems).
* After reset, the PC points to 0x0000\_3000, here the address of the first instruction. Note that it is consistent with the settings in MARS. Do not output if some memory cells are reset during reset.
* Only package all .v files, compress the naming format (.zip), and the top file must be named mips.v.
* All instructions are subject to the behavior specified in the instruction set. See the MIPS-C instruction set (the instruction set is described for single-cycle, note that single-cycle and pipelined instructions are slightly different in behavior, but the end result is consistent, the program runs. The final result should be consistent with Mars). If you have any questions about this experiment, please submit it in the discussion area in time.