**P5 Pipeline CPU Design Instructions**

# **Foreword**

         The experiment from P4 to P5 is a leap in difficulty and the key to success in passing this course. The design difficulty and scale of the pipeline CPU are quite large, especially the difficulty of debugging work, and it is a difficult problem for most students. This part of the assembly line engineering design method tutorial, mainly from the relevant lectures of Gao Xiaopeng teacher, combined with the relevant experience of previous students, is designed to help students successfully complete the design of the assembly line CPU.

PS: Before reading this chapter, please familiarize yourself with the theory class about single-cycle CPU knowledge, the working principle of the pipeline, and the knowledge of controlling adventure and data adventure. There’s a lot of content, you can choose the right amount.

# **Preliminary Preparation and Guiding Principles**

The engineering method is a method concept of modelling and standardizing design work to reduce development difficulty, reduce error probability and improve development efficiency on the basis of sacrificing certain design flexibility and specific work convenience. Therefore, it is inevitable that some seemingly "troublesome" behavior will occur during the development process. However, from a global perspective, the so-called sharpening of the knife does not mistakenly cut the woodwork, and sufficient preliminary preparation can greatly reduce the problems and workload of the later work. So at the beginning of the tutorial, let's talk about some guiding principles.

## **Write the experiment report first!**

The process of writing an experimental report is actually a process of thinking about experimental tasks, sorting out your own design ideas, and completely shaping your own design structure. If you start to write code when the idea is not fully finalized, it is very easy to encounter problems in finding some new problems in the coding process, or forgetting or remembering a certain design detail. Especially prone to confusion is the design of the various control signals and data ports, so that small problems are extremely difficult to detect. By pre-clearing the module function, structure design, port definition, and control signal naming, the possibility of such unnecessary bugs can be greatly reduced.

## **Normalized naming**

In pipelined CPU design, there are inevitably a large number of ports and wire variables used to transfer data in the ports, as well as the large number of data ports generated by the various pipeline components. How to name them is an important issue. Although there are habits in naming, there are some principles that can be clarified. For the data signal, the signal name, phase name, direction, such as "ALUout\_E\_out", the output port representing the calculation result of the E-stage ALU, is transmitted to the EM-level register. In short, no matter what naming principle is adopted, it is clear that the meaning is clear and ambiguity is avoided. It takes a lot of time to code to avoid a lot of potential risks, which is completely worthwhile.

## **Modular structure**

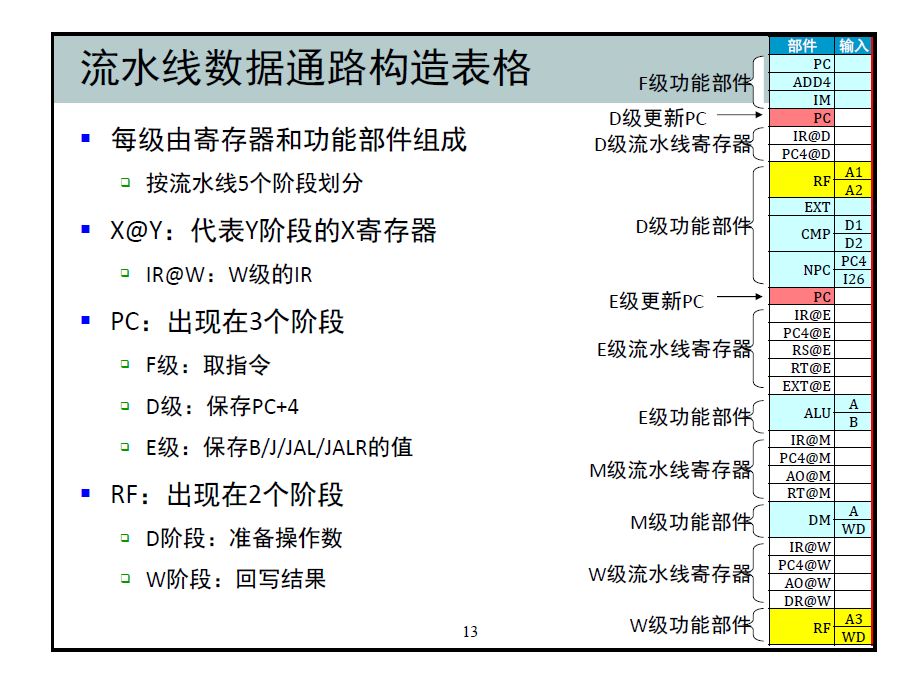
Each pipeline stage is treated as a separate module, and each stage pipeline register is also used as a separate module. Data can only be transferred between adjacent modules. Some students put all the modules together in the mips top-level file, which is certainly not wrong, but because the data lines and signals at all levels are mixed together, modify or increase or decrease a port, there will be a series of modified ports. There is a great risk to the modification of the code. By designing each flow level as a separate module, the data path between the flow level and the flow level can be clearly separated to prevent this problem to some extent.

## **Distributed decoding**

Both centralized decoding and distributed decoding are feasible solutions, but the problem of centralized decoding is that a large number of control signals need to be transmitted in one level and one level, and more ports and data lines are added. Complex structures add to the weight, increasing the likelihood of bugs in the data path. With distributed decoding, it is possible to transfer only instructions between pipeline stages, and then instantiate one and the same control module for decoding at different pipeline levels.

# **Build Data Path**

## **The first step: constructing the header**



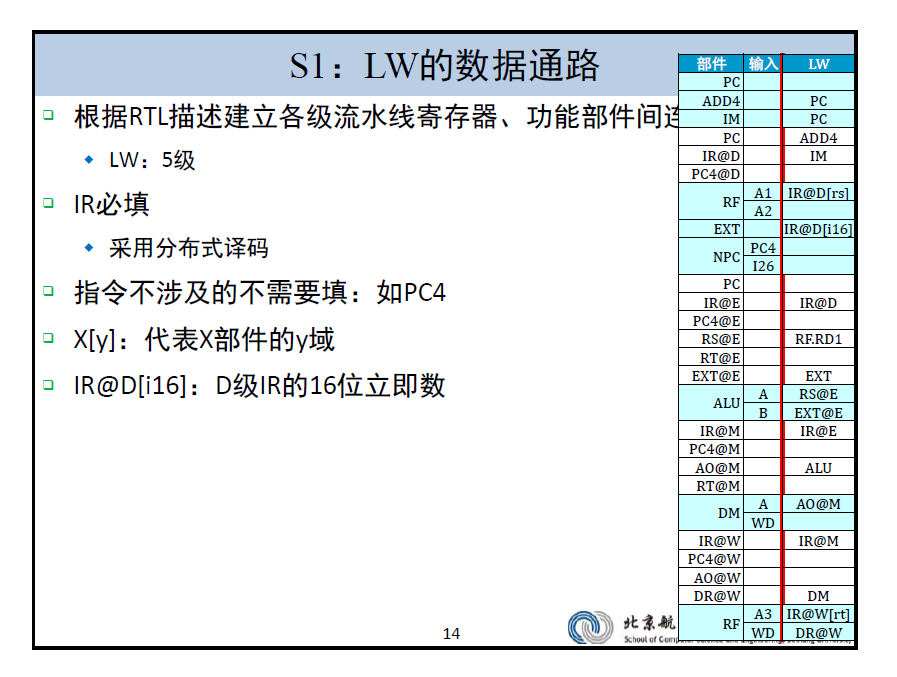
**Figure 1 from Gao Xiaopeng’s lecture notes**

        The contents of the header mainly include the components included in each stage, the various pipeline registers, and the data they input. We will construct our data path based on them.

In the header, we need to write each function module and each data input of these function modules. Note that the control signal input (such as op, en) is not included in the header, nor is it included. As shown in Figure 1, CMP has two data inputs, ALU also has two data inputs, and RF we write read and write functions in D and W levels respectively (in space, the two flow levels are in the same place, but In time, the instruction passes through the D level and then through the W level), so there are four data inputs.

## **Step 2: Analyze the instructions and fill out the form**

        In the construction part of the data path, we ignore the various risks brought by the pipeline, and think about how each instruction works in a single cycle, that is, from which component the data flows to which part, and finally where the result goes, and finally Written in the form. The following figure shows the data path structure of the LW instruction.

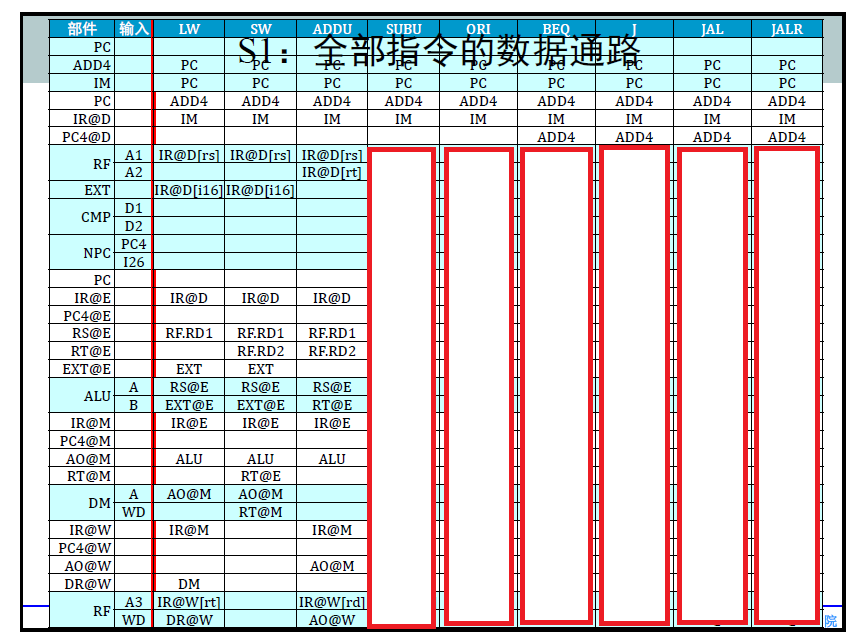


**Figure 2 data table fill in the example**

As we can see, for the LW instruction, first the input source of the PC is ADD4, the source of the instruction data line is IM (instruction memory), the corresponding value of the rs register is read from the RF, and the instruction is extended from the EXT. The value of the immediate value, which is passed to the ALU part in turn, calculates the memory address.

The address port of the DM is read from the OUTPUT of the ALU (abbreviated as AO in the figure), and the read data is transferred to the RF write data port, and the RF write address port is from the rt field of the instruction. Fill in a column.

It should be noted that the data cannot be transferred across the pipeline level registers, so the ALU output at the M level must pass through the E/M stage pipeline. A port is left blank if it is not needed during the execution of this instruction. By analogy, each instruction occupies one column, and we complete all the instructions in turn, as shown below: (some instructions have been coded)

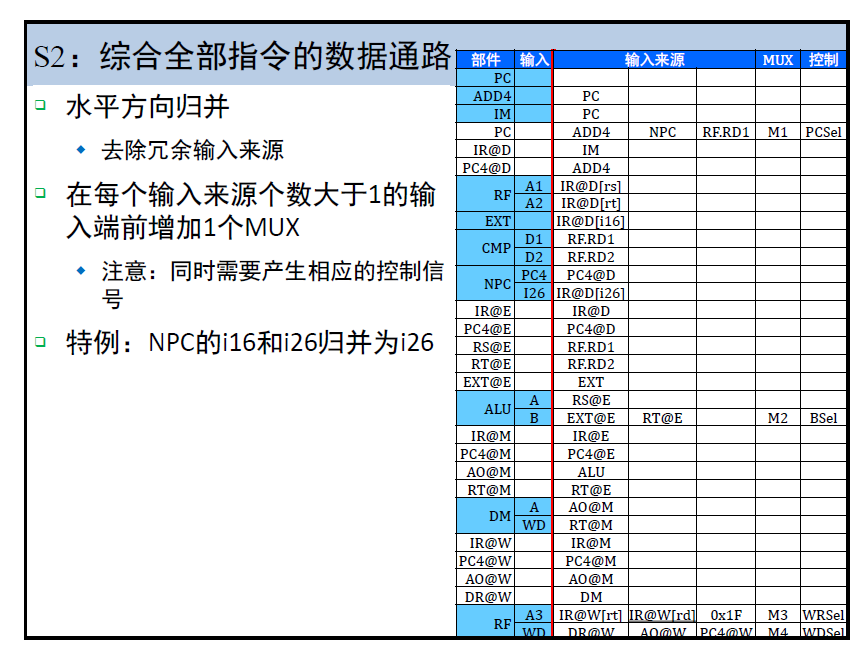


**Figure 3 Partial data path diagram**

These nine instructions basically include various basic instruction types. After completing this part, most of the instructions of P5~P6 can be simply analogized. Therefore, students are strongly recommended to complete it themselves.

## **The third step: Construct a multi-selector for input ports that may have multiple data sources.**

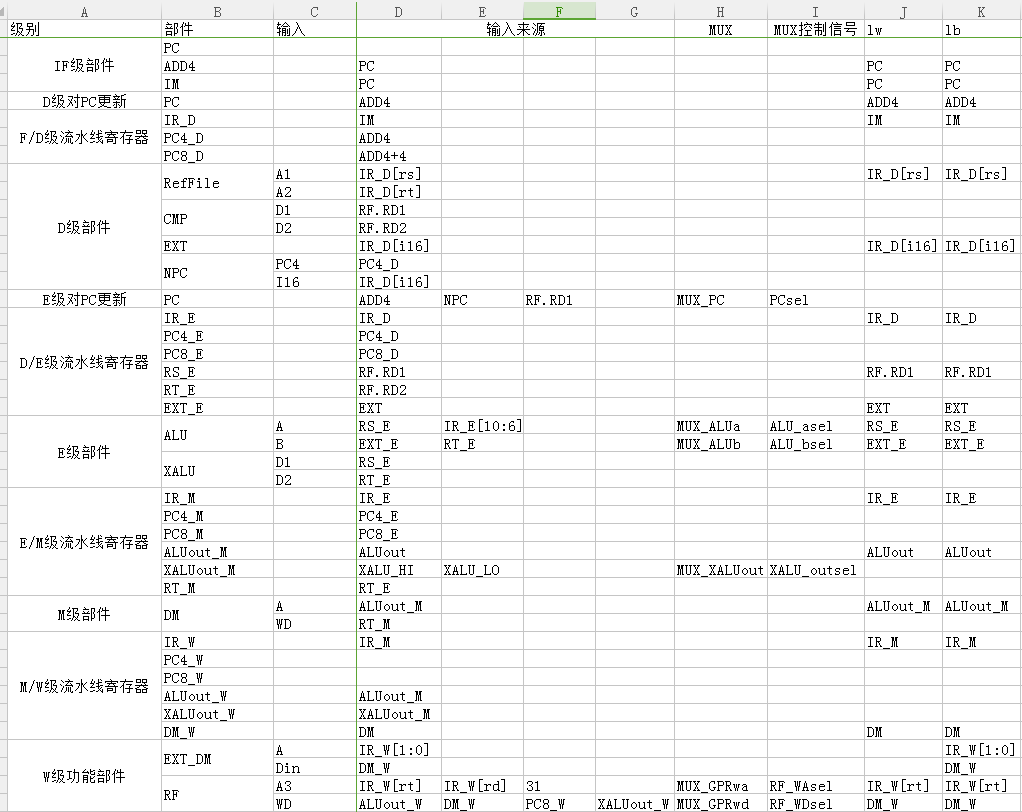
Looking at Figure 3, we can see that some ports have multiple possible data sources, such as the A port of the ALU. The input may be EXT@E or RT@E depending on the instruction, so we need to add a multi-selector to the input port. . In the table, we build the multi-selector as follows:



**Figure 4 Complete Data Path Diagram**

As shown in Figure 4, we add the "Input Source" column group, and the subsequent MUX (multi-selector) column and the corresponding multi-selector control signal column. If a data port has only one input, only one of these columns is required. If there is more than one possible input for a data port, write it to the data source column and create a new multi-selector and corresponding control signal in the MUX column.

**PS:** Do not recommend such a naming method in the high boss figure, M1234 or something... For example, for the B data input port of the ALU unit, it is recommended to use the MUX\_ALU\_B named multi-selector and use the MUX\_ALU\_Bsel to name the multi-select signal. The time to knock a few more letters is nothing compared to the catastrophic consequences of mixing a multi-selector.



**Figure 5 The data path constructed by the author (most instructions are omitted)**

After completing the above three steps, the remaining instruction parts of P5 and P6 are actually quite simple, and the data path of most similar instructions is completely the same. It is only necessary to add a calculation function to the corresponding calculation module and construct a corresponding control signal in the decoder module.

# **Analysis of Data Risks**

We have already discussed the construction of the data path without considering the risk. This part is relatively labor intensive, but it is relatively uncomplicated. The most troublesome, difficult to understand, and most error-prone part of the pipeline is the part of dealing with data adventures. It can be said that this part is also the core of P5 and P6, and it is also the most difficult part of the students. In this section we will show you how to use the engineering method to implement the pause and forward mechanism to deal with data risk issues.

         Almost every instruction needs to get a certain data input, and then some instructions will also produce data output. The reason why the pipeline is risky is because the data required by the instruction is just the data supplied by the previous instruction, and when the subsequent instruction needs to use the data, the previously supplied data has not been stored in the register file, which causes the subsequent instructions to be abnormal. Read the correct data. Therefore, we analyze the pause and forwarding situation from the demand data and the behavior of the supply data.

Demander: For an instruction, the actual need for register data is some hardware component. For example, for the addu instruction, the data is required to be in the ALU of the pipeline E level. For the BEQ instruction, the data consuming is the comparator CMP located in the D stage of the pipeline. For the SW instruction, there is an EX-level ALU (this data is used to calculate the storage address), and a MEM-level DM (the specific value to be stored here).

        Supplier: All suppliers are pipeline stages that store various data from the previous level. As for why it is necessary to provide data by the pipeline level register instead of providing data by ALU or DM, beyond the scope of this article, you may wish to understand for a while that this is a hard design requirement. For specific reasons, you may be interested in consulting the relevant materials or Teacher and classmates discuss.

        By analyzing the data consumers and suppliers, we can clarify the strategies for dealing with data risks. Assume that the data I need now is actually calculated, but it has not yet entered the register file. Then we can use Forwarding to solve the problem, that is, not to refer to the value of the register file, but directly from the supplier of the downstream class. The result of the calculation is sent to the demander of the previous flow level for reference. If the data we need is not yet calculated. Then we can only pause (Stall), let the pipeline stop working, wait until the data we need is calculated, and then start the following work.

        So, how do we determine if the data we need is already calculated? We propose a simple and efficient decision model for this: demand time - supply time model.

        For a certain data requirement of an instruction, we define the demand time Tuse: when the instruction is at the D level, the corresponding data must be used after several clock cycles.

        For example, for BEQ instructions, the data is used immediately, so Tuse=0.

        For the addu instruction, it waits for the next clock cycle before it enters the EX level to use the data, so Tuse=1.

        For the sw instruction, it requires GPR[rs] data to calculate the address at the EX level, and GPR[rt] is required to store the value at the MEM level, so for rs data, its Tuse\_rs=1, for rt data, its Tuse\_rt=2.

Feature 1: is a fixed value, the Tuse of each instruction is certain

Feature 2: An instruction can have two Tuse values

        For the data output of an instruction, we define the supply time Tnew as: an instruction at a certain pipeline level, how many clock cycles it can calculate the result and store it in the pipeline level register.

        For example, for the addu instruction, when it is in the EX level, the result is not yet stored in the pipeline level register, so its Tnew=1 at this time, and when it is in the MEM or WB level, the result has been written into the pipeline. Level register, so Tnew=0 at this time.

Feature 1: is a dynamic value, each instruction has a different Tnew value at different stages of the pipeline

Feature 2: An instruction will only have one Tnew value at a time (one instruction has only one result)

        When a data conflict occurs between two instructions (the write register of the previous instruction is equal to the read register of the following instruction), we can judge the strategy based on the Tnew and Tuse values.

Tnew=0, indicating that the result has been calculated. If the instruction is at the WB level, it can be solved by the internal forwarding design of the register (for internal forwarding, please consult the data or discuss with the classmate) and do not need any operation. If the instruction is not at the WB level, it can be resolved by forwarding the result.

Tnew<=Tuse, indicating that the required data can be calculated in time, and can be solved by forwarding the result. (What is the difference with 1?)

Tnew>Tuse, indicating that the required data cannot be calculated in time, and the pipeline must be suspended.

        We have found that for a pause, it is only necessary to compare the instruction at the ID level with the subsequent instruction, and it is simple to handle. For forwarding, we need to compare the instructions at all levels and add forwarding data paths between the levels, which is more complicated. Therefore, we first discuss the suspension mechanism.

# **Construction of the suspension mechanism**

From the discussion in the previous section, we know that the decision to suspend the mechanism is relatively easy. That is, comparing the Tuse of the instruction currently at the ID level with the Tnew of the instruction at the subsequent pipeline level, if both reads and writes the same register and the corresponding Tnew>Tuse, the pipeline is suspended.

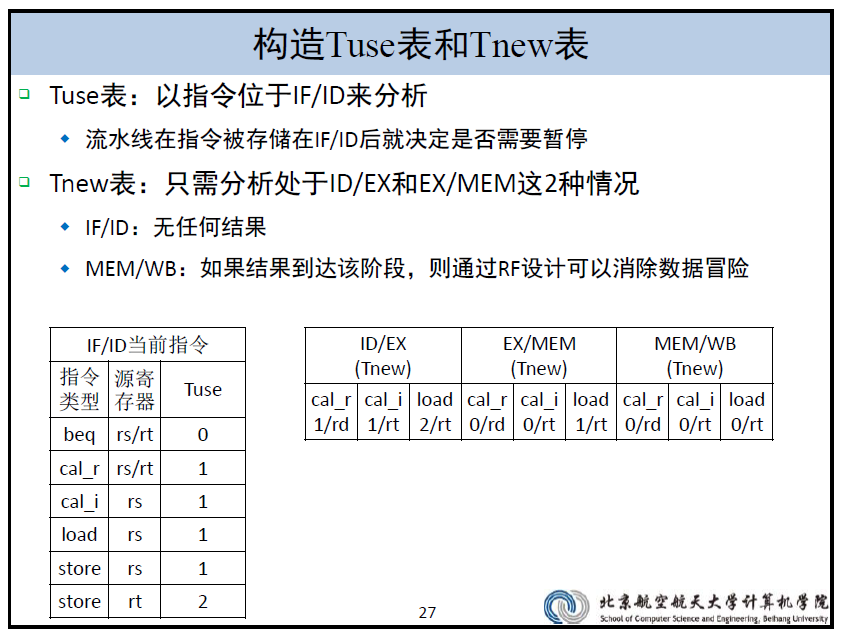
        In the same way, we use engineering analysis and construction methods to analyze various situations in a simple and clear way by suspending the mechanism table, avoiding various errors in the brain repair process.

        First, we classify the various instructions according to their read behavior of the register file. The workflow of the same type of instructions is similar.

* Cal\_r class (R-R, register and register for calculation): add, addu, or, etc.
* Cal\_i class (R\_I, register and constant calculation): addi, lui, ori, etc.
* Beq class (judge by CMP comparator): beq, bne, bgez, etc.
* Load class (read memory value): lw, lb, lbu, lh, etc.
* Save class (save values ​​to memory): sw, sb, sh, etc.
* The four instructions of the J class each have subtle differences, and it is recommended that the four instructions be processed separately.

        Among them, only cal\_r, cal\_i, load three types of instructions and jal, jalr will produce results. (jal, jalr is omitted in the next analysis, please think for yourself!)

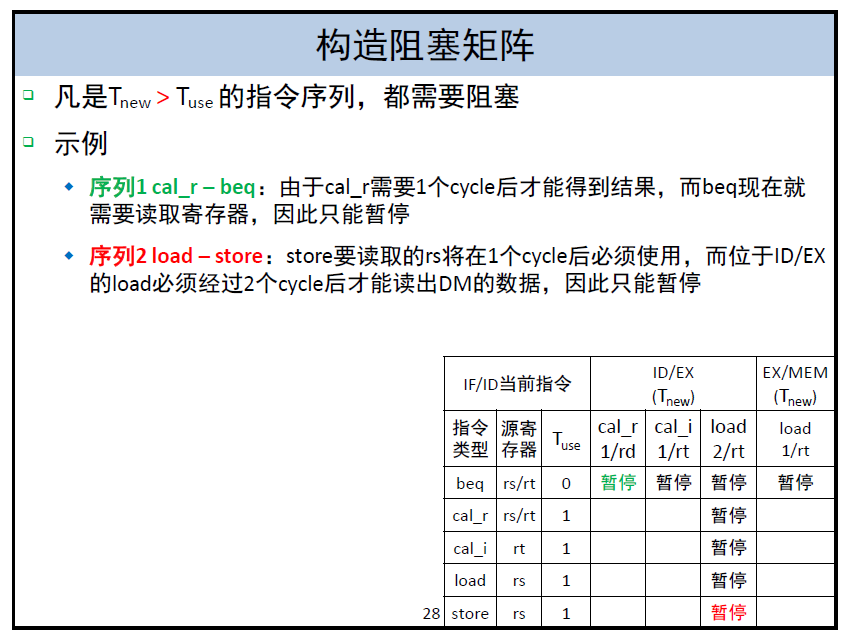
        Next, we use the horizontal row to represent the various instructions at the ID level, the demand data; the vertical column indicates the subsequent flow levels, and can provide various instructions for the data. Construct a table frame:



As shown in the figure, for each horizontal line, we specify which type of instruction is used. The register read by this type of instruction corresponds to which field of the instruction, and what is the Tuse of the data read. For each column, we specify which flow level (ID/EX in the figure indicates that the instruction is in the ID/EX stage pipeline register, equivalent to the EX level), and each flow level is written as Which type of instruction, what kind of instruction is Tnew at this pipeline level, and the data storage register it generates corresponds to which field of the instruction.

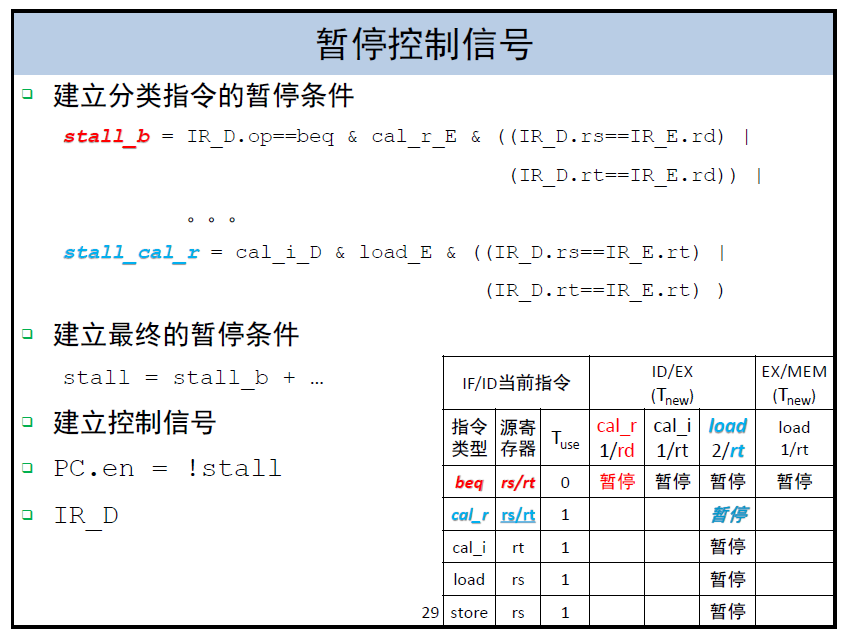
         Then, for each small row formed by each column of the horizontal row, it corresponds to a potential conflict situation. We only need to compare the Tuse corresponding to this small cell. Tnew who is big and small can get this situation. Whether a conclusion of suspension is needed.

         Omit the column of Tnew=0 (there must be no pause at this time), we can construct a table of pauses:



As you can see, in this way, we have a simple and clear analysis of what needs to be suspended. When we need a new instruction, if the instruction belongs to an instruction class that has already been analyzed, then this table does not need to be corrected at all. If the instruction belongs to a new instruction class, we can follow the engineering analysis method: if the instruction has output, add a new column; if the instruction has input, add a new horizontal line; then improve Tnew, Tuse and target register information The next step is to simply add this form to the pause form after adding the new instruction.

After getting the pause form, we only need to calculate the pause condition of the classification instruction, and then logically sum and OR based on these conditions. As shown below:



When we decide that we need to pause, we need to perform three operations:

* Freeze the PC and not change the value of the PC
* Clear the ID/EX pipeline level register, which is equivalent to inserting a NOP instruction.
* Freeze the IF/ID pipeline stage register so that its value does not change.

         So, we have completed the construction of the suspension mechanism. When we insert a new instruction, we only need to analyze Tuse, Tnew according to the format of the table, and then we can construct its corresponding pause judgment signal.

# **Construction of forwarding mechanism**

After processing the pause, the next thing we have to deal with is the relatively complex forwarding mechanism.

        First, let's sort out the specific principles of the forwarding mechanism. The principle of forwarding is that the following instructions need to use the result of the previous instruction, and when the subsequent instruction is executed, the result of the previous instruction has not yet been written to the register file. At this time, we add a number of data channels, and directly transfer the results of the previous instructions, so as to avoid risks without suspending.

        The situation and mechanism of forwarding is more complicated than the suspension. Many students analyze or miss the analysis when they are forwarded or wrong, which leads to the emergence of BUG. In this section we will discuss the engineering approach to easily and clearly construct all forwarding situations to avoid these problems.

        According to our previous data requirements - supply model, when two instructions have data risk, and Tnew <= Tuse, we can solve the problem by forwarding the results of the previous instructions to the following instructions. Therefore, our first task is to obtain the forwarding result of each case by analyzing each combination of instructions. This process can be done by constructing a forwarding analysis table.

        Much like the pause analysis table, we still use the horizontal line to indicate the instructions that need to use the register, except for the pause, the pause only needs to consider the IF/ID level of the instruction; and the forwarding needs to consider the requirements of each pipeline level, such a The whole table will become very bloated, and we can use some strategies to simplify the construction of this form.

## **The first step: analyze the forwarding situation, construct a forwarding multi-selector**

For any instruction, the reference data is essentially a reference to the value of the register file. By parsing the instruction set, we know that the value of the reference register must be the rs field of the reference instruction or the register corresponding to the rt field. So for each stage, we only need two kinds of data, namely the demand rs register, or the demand rt register. For the MEM phase, the rs register is no longer needed, so there are only five cases for the data: the ID phase requires rs, the rt register, the EX phase requires rs, the rt register, and the MEM phase requires the rt register. . However, the instructions only correspond to one or two of these requirements. For example, the rs, rt register of the beq instruction requirement ID stage, the rs, rt register of the ad stage instruction requirement EX stage.

        Next, every data requirement, its correct data, may be the value previously read from the register file, or it may be the result of other pipeline stages not yet written to the register file, so we need these five data requirements Separately establish a forwarding multi-selector to control whether the data should be selected as the original register reading result as data or the subsequent stream-level calculation result as data.

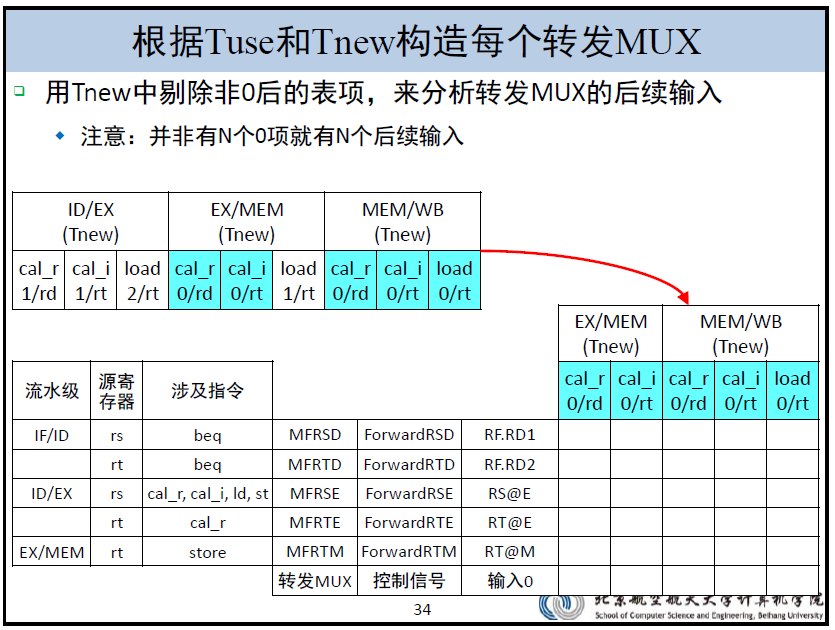


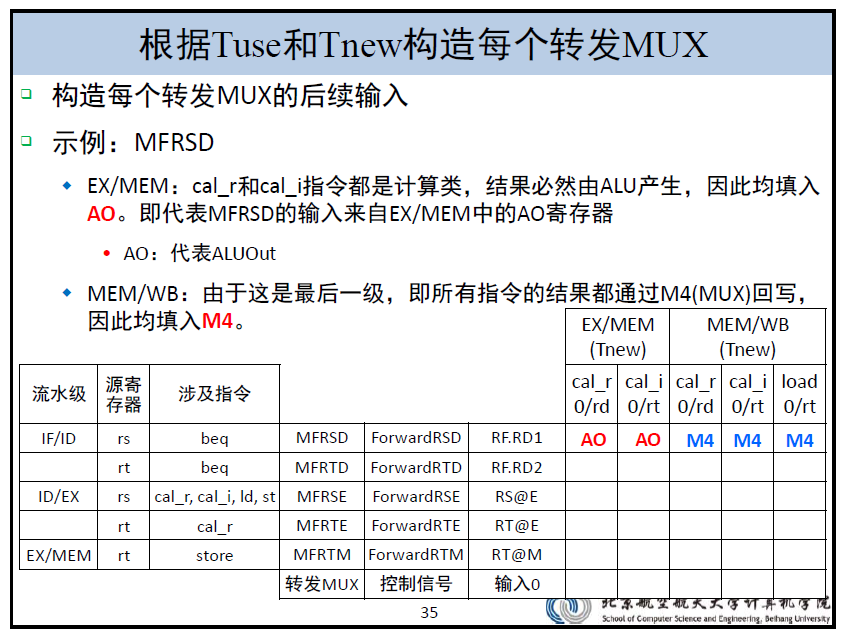
Interpretation of the table: There are five data requirements, one for each of the five rows of the table. Each data requirement corresponds to several types of instructions. When we add instructions later, no new situation will be generated. It is only necessary to add the instructions to the box related to the instructions according to the input data of the instructions.

         The name of the forwarding MUX is: M (MUX multi-selector) F (forwarding) RS (requirement for register RS) D (in phase D)

         Correspondingly, for each case of the forwarding MUX, a special control signal is set. When the control signal is input to 0, that is, when it is not forwarded, the multi-selector outputs the data directly read from the register file before.

         As with the construction of the pause table, the columns are used to represent the subsequent pipeline levels, and each instruction of the data can be provided. Note that when Tnew<=Tuse, only the current situation can be solved by forwarding, when Tnew=0, The result has been that this can only be actually forwarded. So, we save the item in the column, Tnew≠0.



At this point, we basically constructed the forwarding table. The next step is to analyze the potential conflicts of the horizontal column corresponding to each empty space in the table, and fill in the grid. In this case, which data should be selected by the MUX as the final Read data. 

For the M4 in the above figure, you can refer to the relevant naming of Mr. Gao Xiaopeng in the Data Path Construction section.

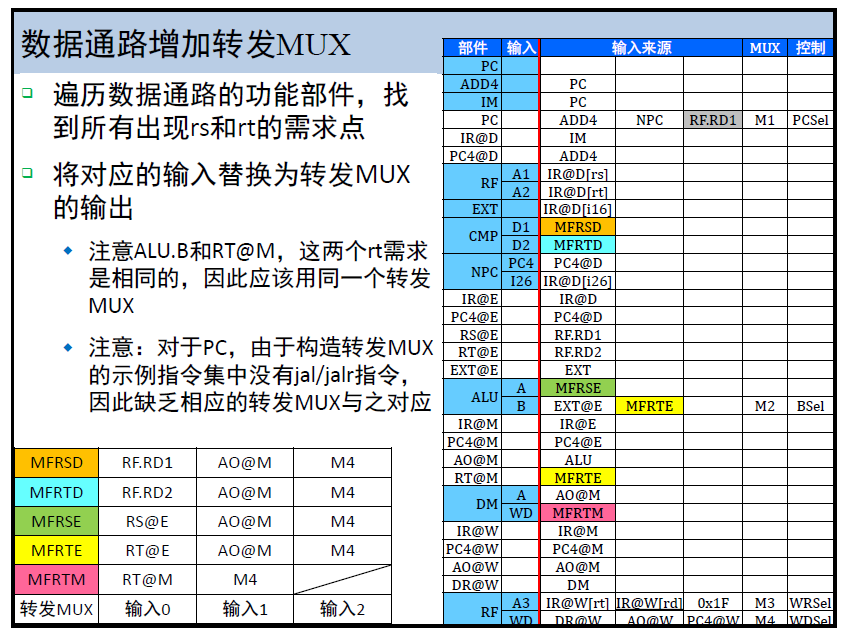
         At this point, if we need to add an instruction, first consider: Does it need to use data? If necessary, add the involved instructions in the horizontal line depending on the use of the registers.

         Second, does it provide data results? If it is provided, it needs to start at the level at which it can calculate the result, and add a new column at each level (such as jal, jal instruction, result from ID level, then the table needs to add a large column) At the same time, in the corresponding new space, fill in the forwarding data source in this case.

         After completing the construction of the forwarding multi-selector, we need to add the forwarding multi-selector to the data path. We only need to keep in mind that any reference to register data in the data path table should be changed to reference the corresponding forwarding multi-selector output data!

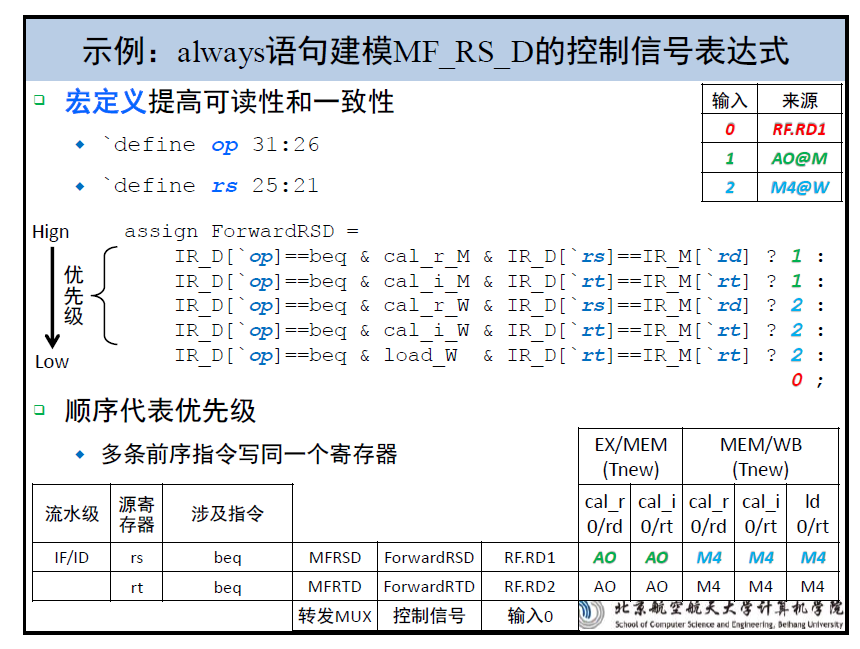
         Therefore, we should modify our previous data path table based on the just forwarded analysis.





## **The second part: Construct the control signal of each forwarding multi-selector according to the completed forwarding table.**

The construction of the forwarding control signal can refer to the construction of the pause control signal, and sequentially analyze each row of the table, from left to right, if the condition of the grid is established (the front and rear instructions conform to the table, and the corresponding target registers of the instruction are the same) , set the multi-selection signal of this multi-select MUX to the current control signal value you set. Also note that if the current pipeline level instruction conflicts with the two pipeline instructions at the same time, and both need to be forwarded to resolve, then we choose to forward the calculation result of the instruction closer to the current instruction! (Think about why?)



The construction of the forwarding mechanism is the most challenging part of the process of building the CPU, so we provide another way of constructing the forwarding mechanism in the discussion area for your reference. Students can also post their thoughts or ideas on CPU construction in the discussion area, exchange and promote each other.

# **Systematic debugging**

I explained how we used engineering methods to help us design pipeline CPUs, but design is only part of our work, and more importantly, how to debug it when there is a problem with our design. And debugging is by no means waiting for the output window of ISE to be dazed, nor is it a chance to look for luck, debugging can also have a systematic strategy.

1. Problem recurrence:

        We found a bug that was definitely the result of our ISE simulation results and the results of the MARS assembler. At this point, we need to patiently run step by step (you can also run 10 steps or more at a time to quickly find the location of the problem) until we find that the value of a register or memory is different from the MARS operation. That is to reproduce the "case" of this BUG.

2. Error tracing and positioning:

        When we find that a write data is wrong, nothing more than a few possibilities: either the data is calculated correctly, we are connected to the wrong line; or the data is wrong. At this point, we need to record the current time, then restart the simulation, run to the previous clock cycle, check whether the data at the previous moment is wrong, if it is correct, it means the connection problem; if there is an error, the explanation may be calculation Error, or control signal error, then further check the corresponding control signals and data at the previous moment. Until a certain moment, its previous state is accurate, only at this moment, the result of the instruction execution and the expected problem. On this basis, we can check the data of each part to see which component has an error in the calculation of which data, thus correcting the error.

        The above two are just the principles of the general nature. They are relatively empty. When you are debugging, you must remember to avoid watching the BUG fall into a crash or irritability. The code and data size of the calculation group are far more than the previous C language program, so we must be organized when debugging, first find out when it started to show the problem, and then step back and follow the error signal until the error signal Once it appears, it will be more efficient to check the bug. In the classroom test of P6, the author wrote a bug because of keyboard misoperation. In the last two hours, a 300-line test program was tracked and run for nearly 400 clock cycles. This tiny error was successfully located and corrected. This method of "problem reproduction - error traceability and positioning" was used.

# **The Real Homework**

## **Overview**

Based on the last experiment, a pipeline processor was designed using Verilog language.

## **Design Notes**

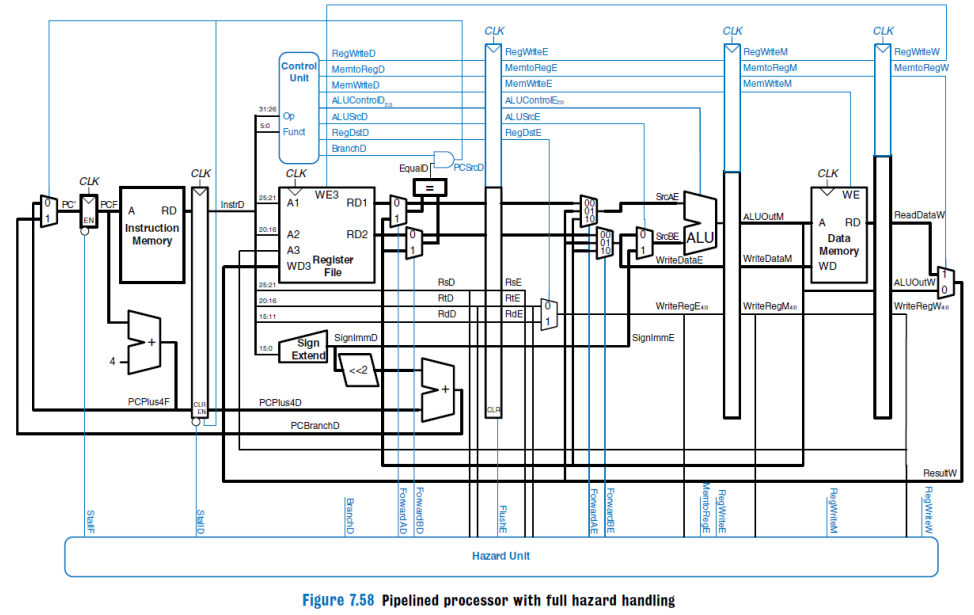
1. The processor should support the MIPS-lite2 instruction set.

* MIPS-lite2={ **addu, subu, ori, lw, sw, beq, lui, j, jal, jr, nop** }

1. The processor is designed for the pipeline.

## **Design requirements**

1. Regarding the top-level view of the pipeline you are designing, we recommend Figure 7-58 in Digital Design and Computer Architecture.



* This figure is for reference only and does not support all the instructions of this project.
* Be sure to separate the controller from the conflicting unit (Hazard Unit in the figure above) so that you can develop it. That is, the independent controller module and the hazard module are defined.
* The design of the controller is no different from a single cycle.

1. The design of the pipeline is the first goal in pursuit of performance, so it is necessary to support forwarding as much as possible to solve the data adventure. This point is a large part of the final score of this project. When the test is conducted, the total number of cycles run by the test program will be judged. I hope everyone will treat it with caution.
2. For class b and j instructions, the pipeline design must support the delay slot, so the design requires attention to use PC+8.
3. The forwarding data source designed to solve the data risk must be a staged pipeline register that does not allow direct forwarding of the feature's output.

* For example, the output of the ALU is not allowed to be directly used as a forwarding input. It can only be ALUOutM.

1. The instruction memory (IM, instruction memory) and data memory (DM, data memory) requirements are as follows:

* IM: The capacity is 4 KB (32 bit/word x 1024 words).
* DM: The capacity is 4 KB (32 bit/word x 1024 words).

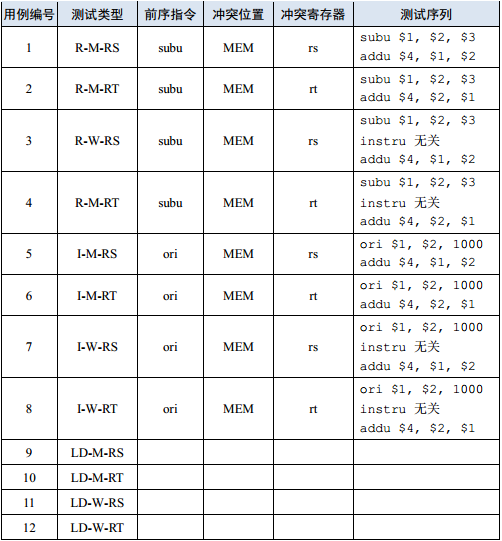
1. The initial address of the PC is 0x0000\_3000, which is the same as the initial address of the code we want to set in Mars.
2. In the program, both the module name and the variable name should be meaningful and regular, and to maintain consistency in the experiments before and after, it is best not to change arbitrarily, otherwise it will be very troublesome to change, and it is very easy to make mistakes.

## **Testing requirements**

1. The assembly test program you write must ensure that all instructions should be tested adequately.
2. Conflicts are the focus of your testing. Your test program must adequately test for data conflicts and branch conflicts, and consider combinations of instructions that may create conflicts.
3. For conflicting test types, you can use X - Y - Z to indicate that they have the following meanings.

* X: The type of pre-order instruction that generated the conflict.
* Y: At which stage the preamble instruction conflicts with the current instruction.
* Z: A register that generates a collision.

For example, we want to test the conflict of addu instructions. Here are some examples, but not all cases are considered, and the rest need to be filled by yourself.



1. If you think carefully, you will find that the coverage analysis of the above conflicts and the design of the pipeline are actually mutually reinforcing. Your test case set is constructed according to your pipeline design, and your pipeline design can be refined based on the test case set.

## **Additional requirements**

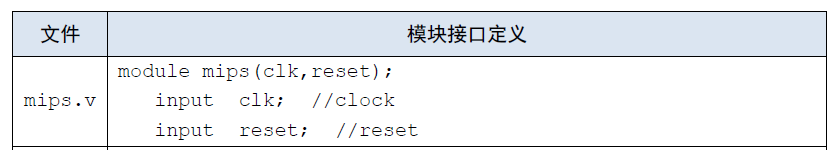
For the engineering files you write, we will use the test platform for unified testing, so some restrictions must be observed.

* The file name of the outermost mips module must be mips.v, and the module in this file must also be named mips.

# **Online test instructions**

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

(note the space), where $time represents the current time, WPC represents the storage address of the corresponding instruction, starting at 0x0000\_3000; Waddr represents the address of the input 5-bit write register; WData represents the value of the input 32-bit write register.

For example, if the address is 0 to register the data 0 at the instruction 100 of 0x0000\_3000, then "100@00003000: $3 <= 00000000" should be output, where 00003000 should be the 16-bit address of the instruction corresponding to the operation, less than 8 bits. Need to fill in zero.

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

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

For example, if the address is at address 0x0000\_3004 at instruction 100 and data 0 is written at address 0x00001004, then "100@00003004: \*00001004 <= 00000000" should be output.

* 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.

## **Additional instructions**

* Since this design is a pipeline CPU, data conflicts and control conflicts between various instructions should be resolved by using forwarding. If you cannot use forwarding, you can choose to use pause.
* It is also because of this, the solution to conflict resolution in CPU design is different, and the number of cycles required to run the test program is different. The better the design, the less the number of cycles required.
* For our experiments, we don't require everyone to design perfectly, but at least the common conflicts that can be solved using forwarding are solved by forwarding. In the class test, we will set different cycle ranges according to the test program. Your CPU should run a complete test program in this range, otherwise we will not let you pass the class test even if the result is correct.
* When testing in class, we will definitely focus on the solution to conflicts in your design. A large part of the score depends on the number of cycles your CPU runs and the Q&A.
* For the behavior of the instruction, Mars and the instruction set are not exactly the same. It is recommended that you simply try to find out how the new instruction works in Mars before adding the instruction. This is not a hassle. To quote a classmate's words - sluggishness can lead to tragedy.