# MIPS32 CPU COURSEWORK – *team 12*

Instructor: Thomas, David B

By: Yujie Wang

Jason Tang

Yujun Han

Adam Rehman

Yinglong Liang

## The overall design of the CPU:

### *1.Overview*

The MIPS CPU implemented in this project is a two-stage CPU, used in conjunction with an Avalon-Compatible RAM interface[1]. In our case, the data and instructions are combined in one RAM module. Most instructions will complete execution within two states, one for fetching and one for execution. Load instructions require one more cycle as an extra RAM data access is needed before writing to the registers. During the fetching state, an address will be passed to the RAM, and the CPU will wait for the instruction from the RAM. When the instruction is available, it is given to the instruction register (IR) and the CPU will proceed to the first execution stage. The decoder will parse the instruction binary from the IR and output control signals to direct the flow of data in the data path of the CPU.

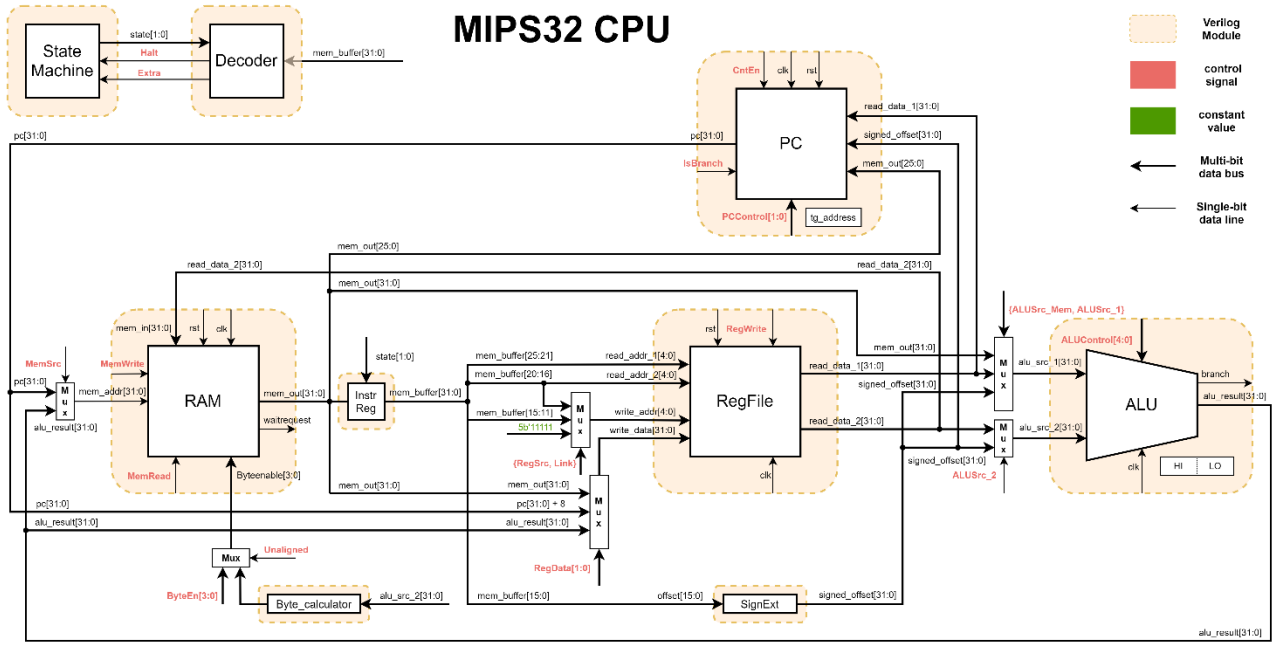


*Figure 1*

### *2. Components within the CPU*

A detailed diagram of the CPU is shown in Figure 1. There are five main components and three helper modules for the CPU. The main modules include the program counter, register file, decoder, state machine, and the arithmetic logic unit (ALU). All calculations done by the ALU are combinational, and the results are obtained almost immediately after the inputs are given. The results from the ALU can then be passed to the write data bus of the register file. In this design, the function of register file is kept as simple as possible, it will not process any data, and only acts as a temporary data storage within CPU. Complex operations on data are done in ALU. Also, the ALU contains two special registers - HI and LO, which are used to store the 64-bit results from multiplication and division. When doing these two calculations, the result will be put into the wire HILO, the combination of the two registers.

The RAM will receive its input address either from the ALU or the program counter. An address received from the ALU will be used in the execution of data transfer instructions. The program counter increments by four in the execution stage of every instruction and contains one register to store the target addresses for both conditional and non-conditional branch instructions. This is used for the branch delay slot, where the CPU will first execute the instruction following the branch instruction before jumping.

These guidelines illustrate how different components work in tandem, and thus the CPU can take correct actions for all required instructions. The details of main design decisions will be discussed in the next section.

## Design decisions in CPU

### *1. Load word left(LWL) and Load word right(LWR)*

One tricky aspect for design is how Load word left and Load word right are implemented. According to the MIPS ISA[2], these two instructions will specify only one register to read and write. One option is to add this function to the register file, performing a read before writing values to the register. However, adding this function causes the register file module to take another input that will only be use for these two instructions. To maintain the simplicity of the registers, the load word left and load word right instructions are done inside the ALU. This is because the ALU module can already do most operations on data, and would minimize the changes on the whole CPU design.

### *2.Run-time byte calculator*

For instructions involving byte selection at run-time such as load byte and store byte, the selected byte must be calculated separately. We set the output of *Unaligned* to high when specific bytes need to be extracted, and this is passed to a helper module called byte_calculator. When loading bytes, the offset is no longer word-aligned, so address of the word is first calculated using ALU ignoring the last two bytes. Then the byte calculator will then generate the byteenable signal by calculating offset modulo four. At runtime, the ram would read the byteenable signal, and output the exact bytes that are required by the CPU.

### *3. The program counter module and branch delay slot*

One of the most important features is the implementation of the branch delay slot. The solution we chose is to use a special register that stores the target address given by the jump/branch instruction (see Figure.1).

When encountering a jump instruction, the target address is first stored in the special register, and the program counter proceeds to the instruction immediately following the jump. After the following instruction is executed, the value of the program counter is set the address in the special register, performing a jump. This functionality is achieved by passing the program counter a control signal called IsBranch from the decoder, which will only be high when the CPU is required to jump.

# Testbench

## 1.Test flow

This test bench is designed to be able to test any CPU that satisfies the MIPS1 specification and Avalon memory-mapped interface. The main shell script contains three stages: argument validation, pre-processing on test cases, and compilation/execution/comparison of test case on the target CPU(show in figure 2). During argument validation, the main shell script assumes one compulsory argument which is the source directory of CPU and one optional argument for a specific instruction to test. The shell script would abort if an invalid source directory or an unsupported instruction is supplied. Pre-processing of the test cases removes previously generated files if they exist, and generates testcases from 'readme.md' file in each instruction folder. As shown in figure 3, each test case contains four files: assembly code, v0 reference, initial content for data section and reference for data section. The final stage compiles the given CPU with a Verilog test bench and test cases in the 'test/'directory. The result of each test case is designed to output via 'register_v0', which is then be compared against reference.

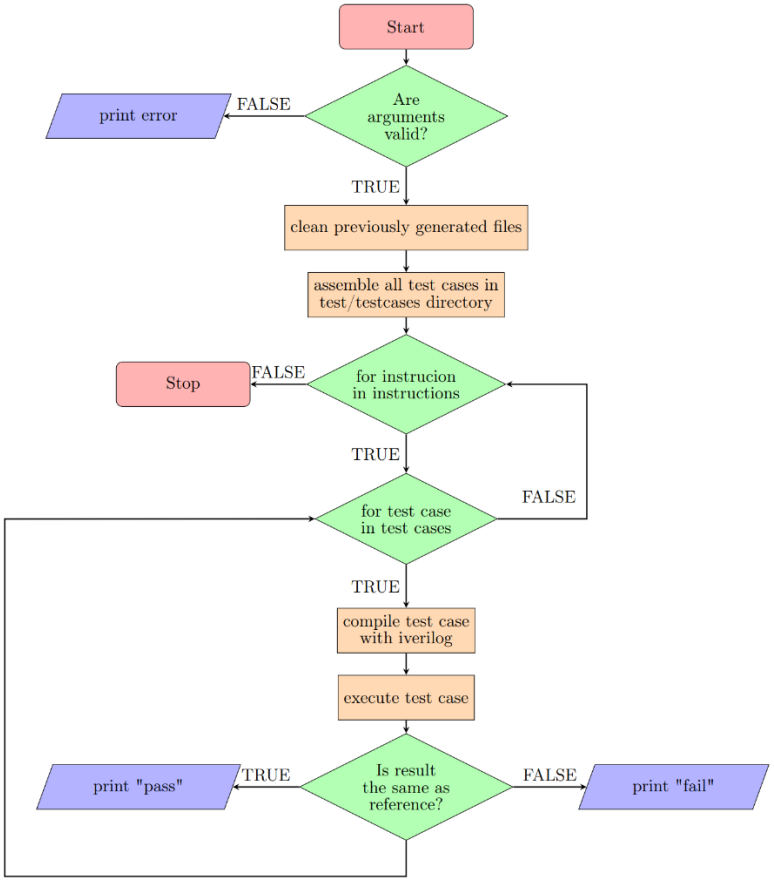


*Figure 2*

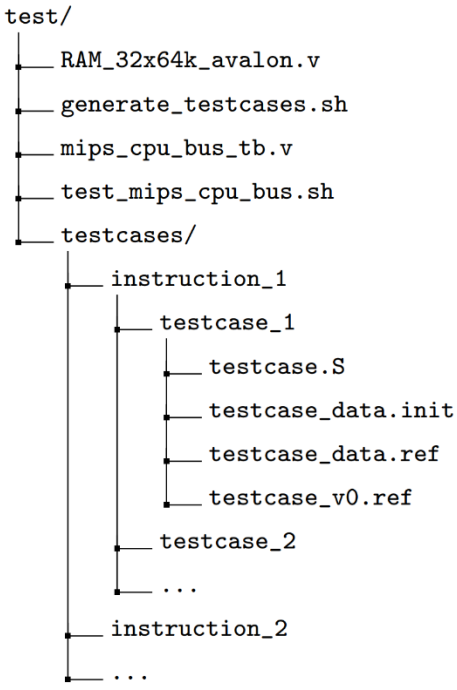


*Figure 3*

## 2.Testcase Design

In order to make the testing comprehensive, test cases are designed to focus on covering all edge cases. Edges cases are included based on the type of instructions, such as data transfer instructions and arithmetic instructions. For example, to test branch instruction `bgze`, test cases cover situations when the specified register contains negative, positive integer or zero. In addition, as the specification mentions that `bgez` takes its offset as signed integer, there is also test case where the offset is negative integer in 2's complement to

test its capability of jump backwards. Also, most data transfer instructions are tested with both negative and positive address offsets. Another example are `div` and `divu` instructions, `div` divides two oprands as signed integer whereas `divu` takes oprands as unsigned. The test cases of `div` are therefore reused for `divu` but with different references. The reference is obtained by executing the assembly codes on MARS simulator[3].

# Area and timing analysis

## *1.Time analysis*

The Intel Cyclone IV E FPGA on Quartus Prime is used to perform Timing and Area analysis. We simulate the CPU under the conditions of 1200mv giving us the results in figure 4. The maximum clock rate for the CPU is 7.48MHz under 85-degree conditions. The number is slightly increased to 8.38MHz at zero degrees Celsius.

| | **Fmax** | **Restricted Fmax** | **Clock name** |
|---|---|---|---|
| **1200mv 85C model** | 7.48MHz | 7.48MHz | CLK |
| **1200mv 0C model** | 8.38MHz | 8.38MHz | CLK |

*Figure 4*

In the design, most of the instructions will be completed in two to three states, as we are aiming for a low CPI, instead of optimizing the critical path. The analysis shows the worst-case timing path is from registers to ALU HILO, which means the worst case will happen which instructions like MULT or DIV is executing. We can improve the critical path by introducing pipeline design in ALU that allow multiply calculation happen at the same time.

## *2. Area Analysis*

Area is also an important factor in the design, and it is usually measured by the number of logic elements and registers. We will focus on these two figures in the analysis, figure xxx shows a summary short area analysis from the Intel Cyclone IV.

| Resource | Usage |
|---|---|
| **Total logic elements** | 8,565/15,408(56%) |
| --Combinational with no register | 7415 |
| --Registers only | 465 |
| --Combinational with register | 685 |
| **Total registers** | 1,150/17,056(7%) |
| --Dedicated logic registers | 1,150/15,408(7%) |
| --I/O registers | 0/1648(0%) |

*Figure 5*

As shown in figure 5, a large number 8565 logic elements are used in this CPU, which is about the half of the maximum number. One reason for this could be the expensive calculations in the ALU module. Especially for Multiply and division, they would require very large combinational logics. 1150 registers are used in the design, the register file would be one large source for this, other modules like state machine and program counter contain several registers. Overall, the result from this analysis is realistic and provides a reliable summary of how CPU would behave if it is on a real board.

**References:**

1. Intel, Avalon Interface Specifications, published on 2020.05.26, available online at https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf
2. MIPS ISA, available at https://www.mips.com/products/architectures/mips32-2/
3. MARS simulator, detail on: http://courses.missouristate.edu/kenvollmar/mars/index.htm