**VE370 Intro to Computer Organization**

**Project 2 Team Report**

Zhou Zhanpeng

Shen Yang

Liu Yihua

Peng Haotian 518370910107

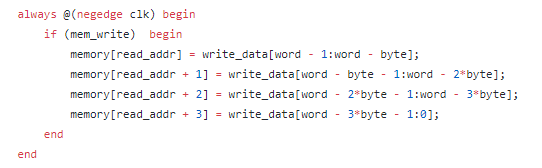
2020. 11. 11

**Description**

**Memory**

The implementation of the memory part includes **Data Memory** (module *data\_mem­* in the source code), **Instruction Memory** (module *Instr\_mem* in the source code) and **Register File** (module *reg\_file* in the source file). There are some typical designing details in this project:

1. The data is byte-addressable. Though this project is based on word addresses, the real memory units supported by MIPS should be byte-addressable. There are some typical MIPS instructions that needs the addresses of byte, such as lb, lbu and lh. Though those instruction are not needed in this project, we still consider this condition in case we could develop more instructions through this project in future.
2. The data memory and register file are driven by the negative edge of clock in this project. This design may lengthen the critical path, but it could avoid some hazards:



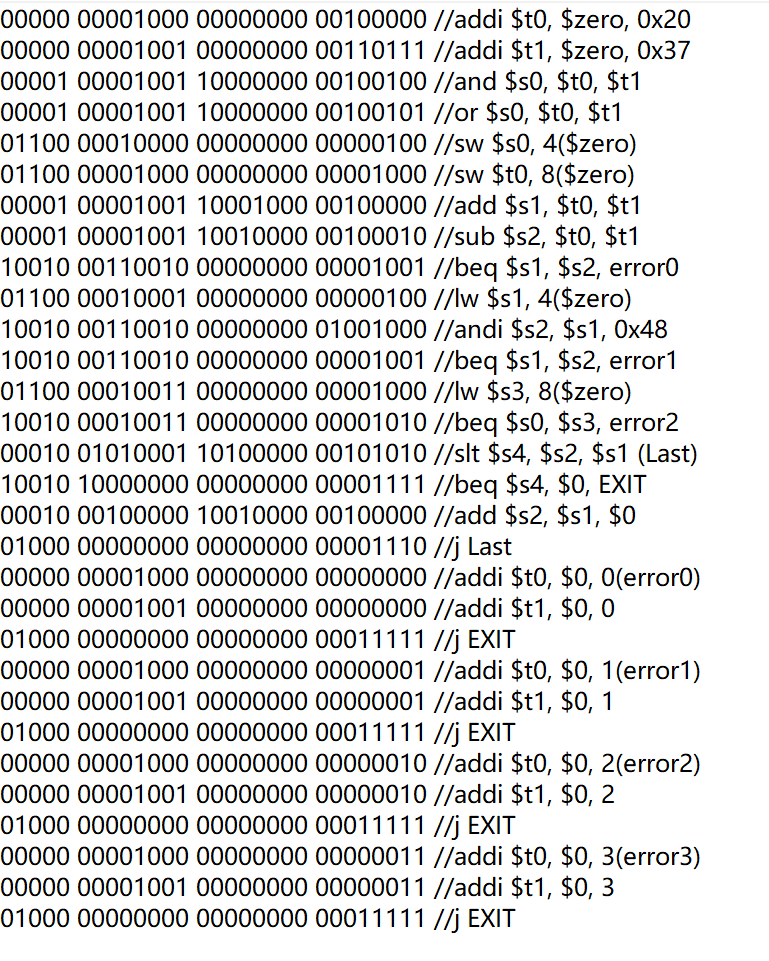
Take a part of the implementation of data memory as an example. If we update data in the positive edge, which is the beginning of a new clock cycle, the value of some variables may stay as the former clock cycle while some already change. If the write data has changed and the address of the write data stays the same, then wrong data will be updated into the write address of the last instruction. With the time of the first half of the clock cycle, this hidden hazard could be solved.

The detailed descriptions of the three memory units are shown below:

**Data Memory (*data\_mem*)**

Data memory is used to store most data. In this project, due to the restriction of the software, we limit the capacity as 1000 words (4KB). The module of data memory takes 6 variables, including 5 inputs ( *read\_addr, write\_data, mem\_write, mem\_read, clk*) and 1 output (*read\_data*). The module will read or write data through a 32\*1000 register called memory. If *mem\_write* = 1, the output variable *read\_data* will be updated with the data stores in *read\_addr* in the *memory.*

**Instruction Memory (*instr\_mem*)**

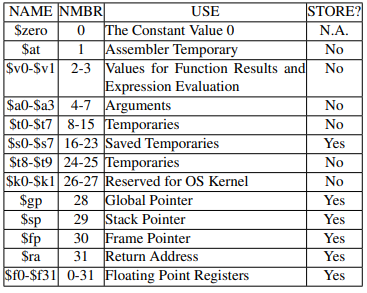
 Instruction memory is used to stores the MIPS instructions in a program in words. The PC could get the data of an instruction by searching the address of that data. In this project, we limit the capacity of the instruction memory as 42 words, which means it can hold at most 42 instructions. The module of instruction memory takes 2 variables, including 1 input (*read\_addr*) and 1 output (*instruction*). To simulate a program, the data in the memory is initialized as follows:

The initial data in the instruction memory

The module will read the *read\_addr* and find the correspond instruction to update in *instruction.*

**Register File(*reg\_file*)**

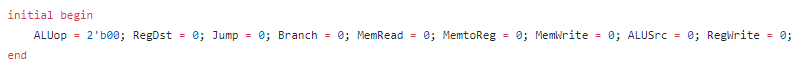
Register file is used to store the data of the 32 most commonly used registers. The order of these registers is shown below:



In this project, the module of data memory takes 8 variables, including 6inputs (*read\_addr1, read\_addr2, write\_addr, write\_data, regwrite, clk*) and 2outputs (*read\_data1, read\_data2*). To simplify the process, the data of every register is initialized as 32 bits 0. The *write\_data* will be updated to correspond register at each negative edge. The *read\_data* will be assigned to correspond register whenever the *rrad\_addr* changes (read is asynchronous while write is synchronous).

**Control Unit**

The control unit is used to generate control signals from different 6 bits opcodes. From the structure of pipeline in VE370 lectures, the control unit should generate 9 outputs as follows:



In this project, to support both beq and bne instructions, we introduce another output called *beq. Beq* will only be 1 when the instruction is beq. If both *branch* and *beq* signals equal to 1 we could determine the instruction as beq. If *branch* signal equals to 1 while *beq* signal equal to 0, then the instruction is bne.

**Discussion**

In this project, we developed a 32-bit pipelined MIPS processor in Verilog. We simulate the program in Vivado and implemented it on FPGA board. Beyond the lw hazard and forwarding unit, we also resolved beq related hazards. In general, the program went smooth, but there are still some concerns:

1. To avoid hazards as much as possible, we update the data in the memory on negative edge of the clock, which may lengthen the critical path and impact the performance.
2. When consider the control hazards, we only introduce IF. Flush. Without considering ID. Flush and EX. Flush, we couldn’t solve control hazards ahead of ID stage.

The performance of the program can be improved by introducing more instructions (there are lw, sw, add, addi, sub, and, andi, or, slt, beq, bne and j now), extend the capacity of data memory and instruction memory, and reducing the critical path, to list a few. These improvements may need more stages and more complicated hazard detection unit, which could be implemented in the future.