Pipelined W450
Design Documentation

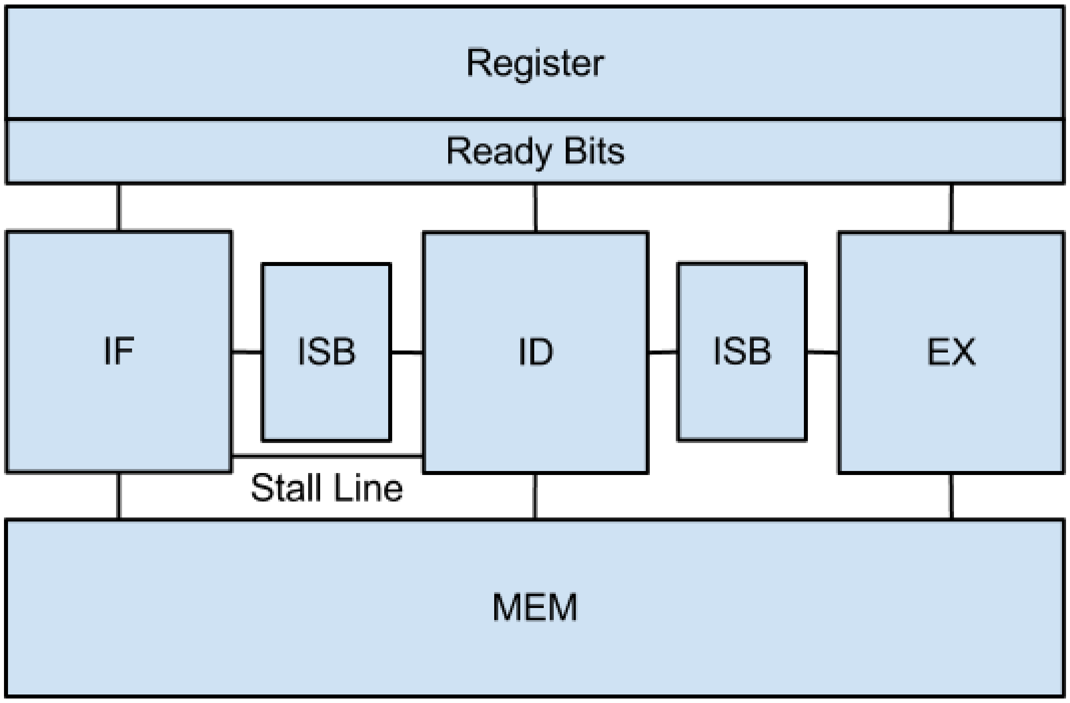
Ze Long
20294594

Overview

The pipelined W450 is divided into 3 stages: IF, ID, and EX. IF fetches instructions. ID decodes operands, fetches immediates, and executes branch instructions. EX executes memory load, arithmetic instructions, memory store, and register store. The stages are connected through inter-stage buffers consisting of sets of registers.

The correctness of the design was sufficient to allow addArray.data to pass, but the insertion sort used in Task 2 couldn't pass. The overall implementation was neither modular nor elegant, due to the author's insufficient verilog knowledge and practice.

High Level Schematic



Design

W450 is a 3 stage CISC processor. The stages are based on classic RISC design of the 5-stage MIPS architecture. Due to the CISC nature though, EX and MEM stages are combined to simplify the implementation, which eliminates the need to have a separate WB stage. Register RAW hazard is dealt with a halt signal from ID to IF, and branch hazard was dealt by ignoring the following instruction if the branch was taken.

The ISA specifies that memory read can happen with in an arithmetic instruction. Without translating the instructions into RISC instructions, the easiest way to accomplish this is by integrating the usual MEM stage into EX, allowing EX to read and write to memory. For any instruction, EX will detect if the instruction indirects to memory through r0, and reads memory port 2 for data. Then EX will execute ALU operations if any is needed, which is followed by a register store or a memory store if the instruction redirects its results to memory. Besides conceptual simplicity, another advantage of this scheme is that by keeping memory accesses within a single stage, memory RAW hazard is eliminated.

Register RAW hazard is resolved by allowing ID to stall IF when ID recognize that a register is being written. The registers are attached with a ready bit. The ID stage will recognize instructions that write to registers, and unset the ready bit of the register being written. Before ID fetches the value in a register as the operands of an instruction, it checks if the ready bit is set, and stalls IF if not, until the ready bit is set by EX after previous instruction is completed.

Branch hazard is handled by setting a internal register in the ID stage to ignore the next instruction if the branch is taken. Since there is only one stage before ID, the register only need to be a flag, rather than a counter. IF will simply keep fetching instructions according to the PC, and ID will ignore instructions if its internal flag is set to ignore the following instruction.

The execution of the processor is triggered by both the rising edge and falling edge of a clock cycle. This scheme gives a good division of time frame for timing and ordering execution.

On the rising edge of each cycle, each stage fetches data from previous stage and execute necessary processes. For example, ID stage will detect register read dependency at the rising edge of each cycle, and stall the pipeline if needed. This is because IF stage will fetch the instruction on the falling edge of a clock cycle, which makes it too late for ID to detect dependency and stall on the falling edge when IF has already fetched the next instruction.

On the falling edge, each stage will execute its main function, and pass the result to the next stage. For example, ID will fetch the register values, immediate values, and execute branch instructions in this stage. For instructions other than branch instructions, necessary information is passed down to EX through the ISB registers.

Implementation

The three stages of W450 is separated in three “always” blocks in the w450 module. Stages are connected through ISB consisting a set of registers. The set of registers represents what is needed by the future stages to execute. Each set of registers is separated into two groups, where one group is used for the previous stage to fill in data for the next cycle, and the other group is used for executing the current cycle.

For example, between ID and EX stage, there are 11 registers setup:

```
reg             EX_rdy;
reg [n-1:0]     IR_ex[1:0];
reg             ignore_r0_indirect_ex[1:0];
reg [n-1:0]     operands_ex[1:0];
reg [n-1:0]     operands_cur_ex[1:0];
reg [n-1:0]     operands_immediate_ex[1:0];
```

where EX_rdy is a ready flag that is unset when ID stalls; IR_ex is the instruction register for EX; ignore_r0_indirect_ex is a parameter passed down to EX; operands_ex and operands_cur_ex are the operands fetched from the register; and operands_immediate_ex is the immediate operand. Except operands_ex, all registers are declared as arrays, where the 0th element is used by the ID stage to fill in values for the next cycle, and the 1st element is used by the EX stage to fetch values from the 0th elements, and execute the instruction. Operands_ex is like the 0th elements, and operands_cur_ex the 1st. Due to limitation imposed by verilog which does not support multidimensional arrays, they had to be declared separately.

The IF stage only operates on the falling edge of the clock, where it fetches the instruction into the ISB of the ID stage. In case of stalling, it omits this step and waits for the next cycle.

The rising edge of the ID stage will fetch the instruction passed down from IF, except if ID has previously left flag to ignore the next instruction in order to resolve branch hazard. Then it reads the instruction and analyze if the registers read by the instruction are ready, by checking the ready bits of each register. In the case of a detected dependency, ID sets the stall line so that IF will not fetch the next instruction or increment PC. On the rising edge, ID also fetches the immediate slot, and increments PC in order to ensure IF will fetch the correct instruction on the next cycle.

The falling edge of the ID stage will fetch the values of each register read, and pass them down to the EX stage. In the case of a conditional branch instruction, ID will execute the instruction directly, compare the operands, and set PC according to the offset. After a branch execution, ID will set the flag which makes ID ignores the next instruction. The falling edge of the ID stage also detects if a instruction writes to a register, and unset the registers ready bit for dependency detection.

The rising edge of the EX stage simply fetches all the parameters passed down from the ID stage.

The falling edge of the EX stage executes the instruction. In the case that an instruction reads memory through r0 redirection, EX will fetch the value supplied by memory port 2. Note that since the only way to access memory in w450 is through r0 redirection, the read / write address are both fixed to the value of r0. Once all the operands are fetched, EX will calculate the result, and write the result back to the destination register / memory location. If the target is a register, then the ready bit of the register is also set to unblocking waiting instruction in ID.

The EX stage will ignore branch instructions as they are executed by the ID stage already.

Issues

The major issue during implementation was timing of operations. Since all the operations happens at either the rising edge or falling edge of the clock, often times the operations from different stages will cause race conditions when they access shared resources such as memory. The implemented solution was to add timing delay to the critical instructions to guarantee the order they execute.

Due to either the timing delays or other factors, the implementation was only tested through addArray.data, while the sortArray.data used for task 2 did not pass. Further investigations are needed to make the processor implementation fully correct.