DRAFT 1

Design

The ambition with our CPU design was to have a high CPI as well as a low CPS. One way to achieve this is to distribute as much workload across each state, reducing the CPUs critical path. This design choice was to make the CPU pipelined as easy as possible.

The CPU has 6 states: Instruction Fetch (IF); Instruction Decode (ID); Execution (EX); Memory (MEM); Write Back (WB) and HALT.

* MEM
  + IFF load or store operation:
    - Access memory
  + IFF branch
    - Replace PC\_next with destination address
* WB
  + Write ALUOut into appropriate registers
* HALT
* IF
  + Retrieve instruction sotred in memory
  + Increment Program Counter(PC)
* ID
  + Determine operands and opcodes from retrieved data
* EX
  + Compute operation determined by opcode

The instruction which most likely will define the critical path areload/store instructions, since they require utilising all 5 functional states (not including HALT state), LF; LD; EX; MEM; WB. Asides from load instructions, all other instructions would take 4 cycles to execute LF; LD; EX, WB.

LF state loads data in from memory into the instruction register; ID splits the memory into appropriate operands for the ALU and Control Unit to operate with. The EX-state utilises these operands to conduct instructions by virtue of the opcode returned from the instruction register. If a store instruction, the MEM state is utilised, if not the WB (Write Back) state is called for all other instructions. Within the MEM state, read/write handlers are set for either the MEM or LF states accordingly. From MEM, the WB state is called, which will run alignment handling as well as writing loaded data to the appropriate register.

The maintainability and scalability of the CPU also affected design choices made in the implementation of the CPU. By virtue of this, it was decided that the control block would manage instruction handling as well as any interrupt handling (e.g., misaligned load/stores or invalid instructions intending to write on the $zero register), whereas the ALU only computed the solution of the instruction itself. Introducing new instructions with this foundation would be quite simple since the design is so robust and simple.

All instructions handled within the ALU were done using in-built Verilog operators. This is because the Icarus Verilog Module synthesis would further optimise all prewritten instructions, furthermore, using well written, documented instructions reduces risk of overcomplexity as well as utilises stable modern features which an industry made CPU would utilise.

Although exception handling is not required by the specification, rudimentary trap handling have been made in place, such as accessing invalid register addresses. By virtue of this, an illegal operation to a register would never execute, which further adds to the reliability of the CPU itself.

Special function

* HI/LO
  + Store the 64-bit output of both multiplication and division type instructions
* PC\_jump
  + PC\_jump will store the address to jump to give the branch or jump instruction computed appropriately.
* PC\_next
  + Stores the next address which the PC will use

To always compute the instruction stored adjacent to the branch instruction regardless of the branch case being successful, a 2-bit finite-state-machine which determines if the following instruction computed, then branch, if successful. The case where two branches are sequential to each over is an exception and is not required to be handled, hence any other instruction asides branch or jump can follow a branch or jump type instruction. This was implemented to ease the process of making this CPU pipelined.

Flags

Stall: Specifies if CPU has is attempted to access memory when *waitrequest* is high. The outcome of this is the state is stalled and re-executed

Test Bench

Timing & Area