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
    - Read/Write flags handled
  + IFF branch
    - Replace PC\_next with destination address
* WB
  + Write ALUOut to registers or memory
  + Handles alignment errors
* HALT
  + TODO: 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 using appropriate operands

The instruction which most likely will define the critical path are load instructions since they require utilising 5 functional states. All other instructions would take less than 5 cycles to execute.

The maintainability and scalability of the CPU also affected design choices made in the implementation of the CPU. An example of this is to have the control black manage instruction handling as well as any simple 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. With such a simple and robust design, introducing new instructions would be quite unchallenging.

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 Registers

* 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

The specification requires the testbench assesses each instruction. This was achieved through having a robust testbench which can be easily modified to examine each instruction through a bash script.

This is done through passing in two text files, one which acts as the memory the CPU will operate upon, and the other being the contents of the RAM if the CPU operated as desired. Both are loaded into the test bench in the form of vectors, and post operation are compared to each over. The Verilog test bench specifies if the sequence of operations is successful.

Every text file are 200 lines long, which are meant to represent the range

[BFBFFF9C, BFC00064] within a fully constructed RAM file, hence instructions begin operation at line 101 within the 1-offset text file.

* Foundational instructions
  + Initially all load and respective store operation are inspected since they will be relied upon by every other instruction test.
* Simple instructions and their immediate equivalent
  + The first set of instructions that will be tested from this are arithmetic, bitwise and set instructions. This is because they are computationally simple instructions to examine.
  + Once an instruction has been deemed functional their immediate equivalent is tested. This is because converting the hexadecimal code into the immediate equivalent is quite simple.
* Control flow instructions
  + Afterwards, J type instructions are tested, this is because they are the simplest instruction type which alters control flow and if the jump type instructions can be confirmed functional, failures in the testing of branch instructions are less probable.
  + Once J types have been deemed operational, branch type instructions are tested. This is because they are the most complex instruction in the specified instruction set due to the altering of control flow based off conditions. Afterwards, the remaining instructions tested are immediate instructions which finalise the testing of each instruction in the instruction set.

Timing & Area