Design

The ambition with our CPU design was to have a high CPI. One way to achieve this is to distribute as much workload across each state, resulting in each state so that each instruction can be performed simultaneously as well as independently with a one cycle difference. This is beneficial since different states within the CPU does enough work such that if pipelined, the overall CPS would be marginally smaller than one with a lower CPI. This design choice was to make the CPU pipelined as easy as possible.

Diagram

Description automatically generated

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

* MEM

**Given wait request is low**

* + IFF store instruction
    - Data is byte reversed and written to memory with computed address
    - PC incremented and IF of next state is entered

**Given wait request is high**

* + State is repeated
* WB
  + IFF R-type
    - Appropriate register is written back to by virtue of opcode
  + IFF I-type
    - ALUOut is written into register RT
  + IFF Jump or Branch
    - PC\_jump set to value of register RS and if the instruction is JALR, register RD will be set to ALUOut
* HALT
  + TODO: HALT
* IF
  + Retrieve instruction sorted in memory
  + PC\_next defined
* ID
  + Data retrieved endian reversed so data in big endian
  + Determine operands and opcodes from retrieved data
* EX
  + Compute operation determined by opcode using appropriate operands
  + IFF R-type
    - Computes instruction which is determined by the function code
  + IFF J type
    - Compute the PC\_Jump
  + IFF I type
    - Computes instruction which is determined by the opcode
  + IFF Jump or Branch
    - PC incremented
    - Enters IF of consecutive operation

The instruction which will define the critical path are load instructions since they require utilising 5 functional states. All R-type and I-type and store instructions will take 4 cycles besides branch and jump instructions which take 3.

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 block manage instruction handling as well as any simple interrupt handling (e.g., misaligned load/stores or 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. In addition to this, preventing certain operations adds to the reliability of the CPU.

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

Special Registers

* HI/LO
  + HI stores the upper 32 bits of the multiplication and modulo solution
  + LO stores the lower 32 bits of the multiplication and division result
* PC\_jump
  + PC\_jump will store the address to jump to, given 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 consecutive instruction has been computed, then branches accordingly. The case where two branches are sequential to each over is an exception and is not required to be handled by specification, hence any other instruction aside from 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 re-executed.

RegWrite: Implemented to ensure register only written to if an R-type or I-type

Write: Ensures that memory is written upon only when CPU is conducting appropriate instruction

Read: Ensures that memory is read from when CPU is conducting appropriate instruction

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 by 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, the vectors are compared against each over. If identical, the test passes, otherwise the test failed.

Every text file is 200 lines long, which is meant to represent the concatenated ranges {[0, 99], [BFC00000, BFC00064]} within a fully constructed RAM file, hence instructions begin operation at line 101 within the 1-offset text file.

The text files are generated through a Python coded assembler which floods the values of RAM into two text files, one containing the RAM post operation and one pre operation.

TODO: Group decision on what graphs to actually use in dragram

* 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.

CPU Timing and Area Summary

Device name: EP4CE15F23C6

A timing and area analysis was conducted using the application Intel Quartus Lite with the FPGA model Cyclone 4. This was done to determine the worst-case timing as well as determine the real estate the synthesised code would take.

Timing Analysis

The conditions that the analysis was conducted upon was both 0C and 85C under 1200mV. The CPU was not restricted to a clock but instead to the maximum clock frequency to get an authentic estimation of what the …

Fitter Analysis

The outcome of the Fitter analysis was that the CPU design utilises 25% of the logical elements on the FPGA (3918 out of 15408). This is quite impressive and likely a result of only using one module for the entire CPU. The use of case switches resulted in most of the logical elements used, if the intention of the synthesis was to produce a device which utilised as little area on a board all case statements could be replaced with if statements, however this was not the aim of the synthesis of the CPU but was to show it is synthesisable.

Bibilography:

Intel Corporation – FGPA University Programme, 2019, Using Timing Analyser. [Online]

Avaliable at: <https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Verilog/Timequest.pdf>

[Accessed: 7th December 2021]

Bozzetti T., Easse E., Tieu C., 2014, MIPS Converter. [Online]

Avaliable at: <https://www.eg.bucknell.edu/~csci320/mips_web/>

[Accessed: 24th November 2021]

Price C., 1995, MIPS IV Instruction Set (Revision 3.2). [Online]

Avaliable at: <https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf>

[Accessed: 19th November 2021]

Soares J., Rocha R., 2019, #4 : Encoding MIPS Instructions. [Online]

Avaliable at: <https://www.dcc.fc.up.pt/~ricroc/aulas/1920/ac/apontamentos/P04_encoding_mips_instructions.pdf>

[Accessed: 19th November 2021]

MIPS Instruction Set. [Online]

Avaliable at: <https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf>

[Accessed: 19th November 2021]

OpenCores, 2018, Most MIPS I(TM) opcodes. [Online]

Avaliable at: <https://opencores.org/projects/plasma/opcodes>

[Accessed: 19th November 2021]