MIPS Data Sheet – Group 5

Architecture & Design of CPU

The overall goal of this coursework was to achieve a working and synthesisable MIPS-compatible CPU. The design of this CPU architecture focused much more on functionality and comprehensibility of the CPU rather than efficiency & performance, whether that be in amount of hardware used or the CPI. This CPU follows Harvard architecture, with separate instruction and data memories, meaning the CPU was much simpler to debug and implement in order to achieve the above goal. The basic Harvard architecture model has a single control unit that interacts with both memories, any inputs or outputs and the ALU. This processor has additional, but common components, such as registers, a program counter as well as a few more unique ones to accommodate some instructions that were designed for processors with a different architecture. These were, a state machine, delay-slot register, load & store blocks, data address block and some extra logic throughout the CPU (see *Figure A*).

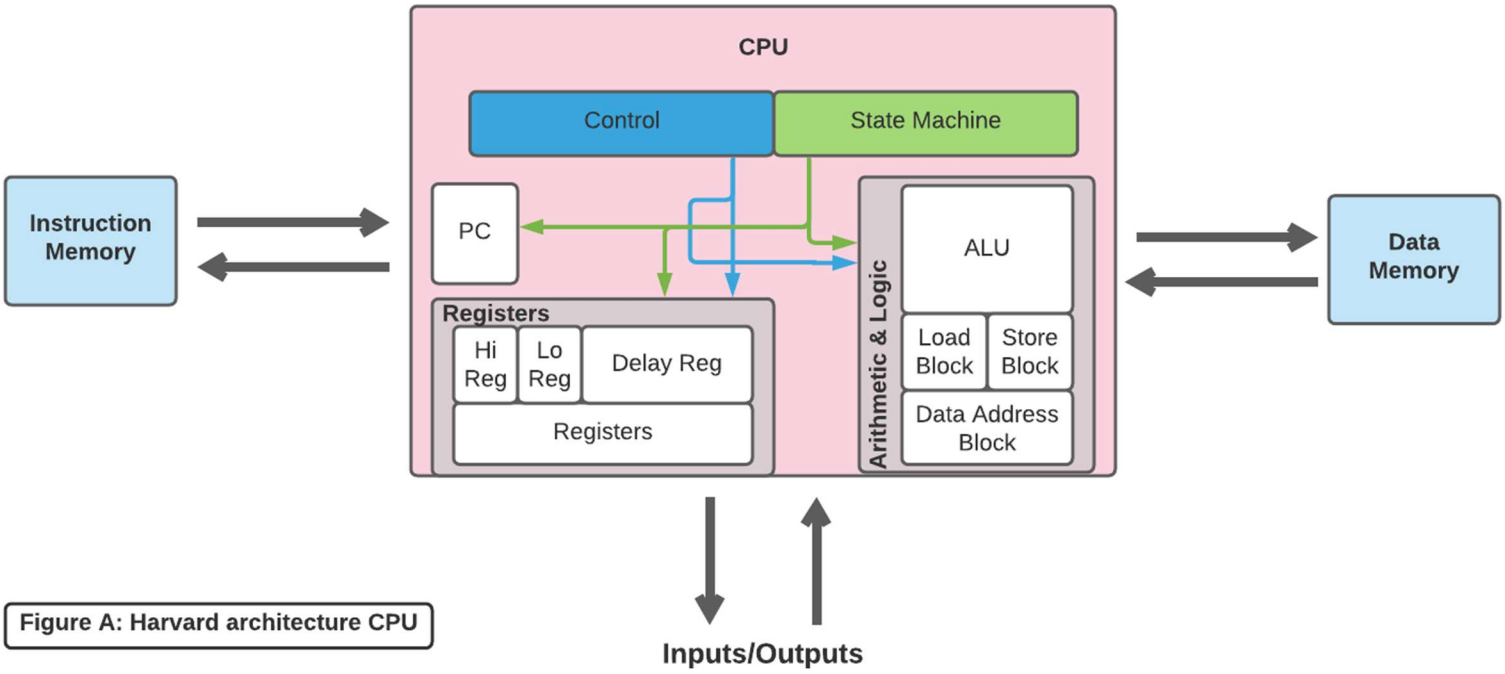


Figure A: Harvard architecture CPU

Most instructions only require one state, as reads are combinatorial, but a few instructions require two states as they need to both read and write to the data memory, therefore a state machine was required. There was the choice of making the state machine variable, in order to reduce the overall average CPI. This added complexity to the CPU and made testing and debugging more difficult. To keep the CPU robust, all instructions were executed in 2 cycles. Ideally, the size of this CPU would have been minimised, given the ideal target CPI of 2. Many instructions require the same hardware to execute, so hardware could have been reused between instructions. However, the size of this CPU is not assessed in this coursework, so more than necessary hardware was implemented in this CPU so to maximise robustness. This reduced all the additional control signals that would be necessary to reuse hardware. Control is much less centralised: a lot of it is determined within blocks themselves.

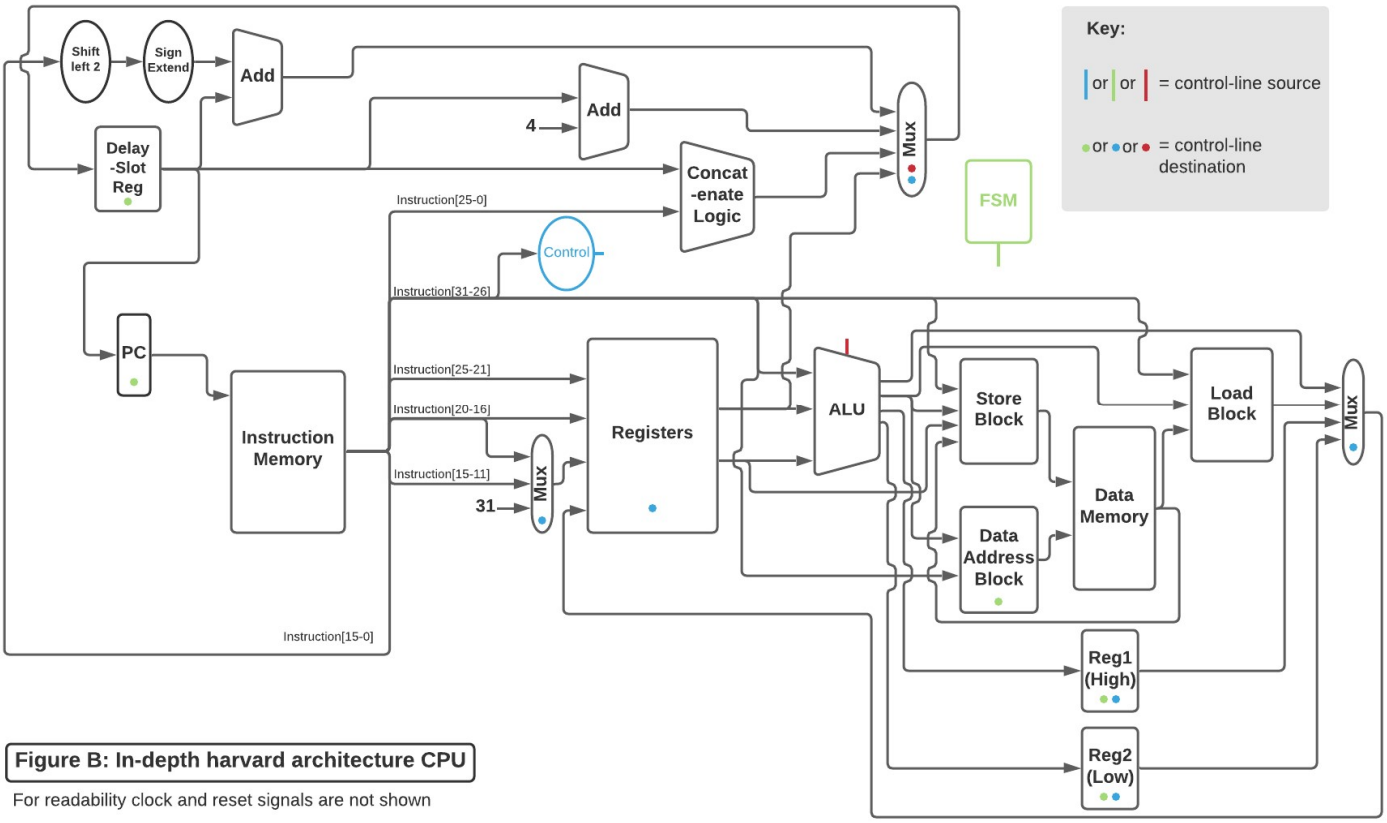


Figure B: In-depth harvard architecture CPU

For readability clock and reset signals are not shown

Initially, the CPU had a separate ALU control block, this was later integrated withing the ALU by feeding the whole instruction word as an input to the ALU. With a few instructions the ALU control block worked well, though as more instructions were implemented, it was realised more control signals were required, and the wires quickly became very messy. Therefore, it was decided that the CPU would be simpler and easier for testing if the ALU control block would be integrated within the ALU.

The purpose of the ALU was to do the bulk of the combinatorial logic. Each instruction had its own hardware and own channels within the ALU, essentially making the ALU a big decoder. The High and Low registers (which store the results of multiplication and division operations) were made separate from the ALU and placed outside so that the ALU block itself can be completely combinatorial. An interesting design choice within the ALU is the use of the synthesisable $signed() and $unsigned() operators, allowing System Verilog to implement signed arithmetic automatically, instead of it being done manually, which would be unnecessarily tedious.

Another interesting design choice was the implementation of the branch delay slot. In the specification ISA, there are no 'branch likely' instructions, meaning that all branch/jump instructions execute the next instruction before completing the next jump. This means the address of the instruction in the PC counter is known two instructions before it is due to be executed. This allows us to implement the branch delay slot by simply putting a register between the PC and the combinatorial branching logic, as seen in *Figure C*.

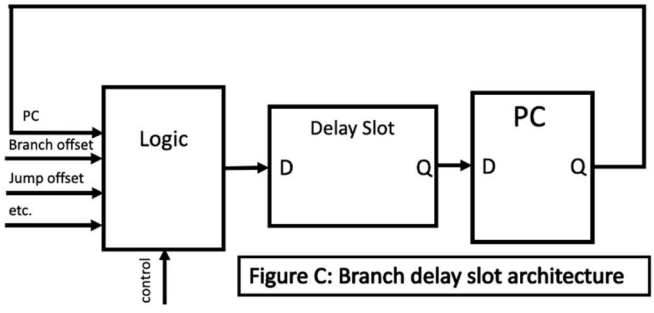


Figure C: Branch delay slot architecture

Load and Store blocks were designed to sit between the data memory and the CPU. In addition to implementing the partial load store instructions, they also handle the endianness conversion between the RAM and the memory. A RAM address block controls the address of the data RAM that the CPU is accessing in each state. Similarly, to the ALU, these blocks had no control signals going into them as the whole instruction word is a direct input.

Testing

Development

During development each component of the CPU was tested individually with their own testbench. This step was to make sure that each component worked as expected and to avoid problems when integrating the components to form a CPU. Initially the complete CPU was tested without utilising any of the memories, as it gave us more control over the inputs and timing of the CPU. On top of that, un-synthesisable $display() operators were temporarily added in the CPU to look at the values of important signals. Moreover, the CPU's memory interface was directly controlled from the testbench module. Testing like this helped identify small human-errors in the code and any misunderstanding of the CPU functionality. For example, in one instance the testbench was showing that many instructions were running twice. After some debugging it was realised that the next branch delay slot was being allocated the wrong input.

Testbench

The testing suite was implemented using two shell scripts. The first takes two arguments, the directory of the CPU, and the instruction to be tested. The second script takes in the directory but has an optional second argument which is used to specify an instruction to be tested. Depending on whether an instruction is included as an argument, the second script will either test every instruction or just the one specified by running the first script multiple time or just once. This allowed the code to be more organised and avoided messy loops.

A testbench for each instruction is comprised of a data RAM, an instruction RAM and a top-level testbench module. The method was to connect the CPU to the programmed RAM and switch the CPU on, waiting for the CPU to halt which signified the program had finished and inspecting the contents of the RAM to see if the program had been processed correctly. For each testbench the shell script produced two log files: one which contained any compilation errors or warnings, and one which displayed any runtime outputs of the testbench. This made debugging a failed testbench much simpler and more straightforward (as seen in *Figure D*).

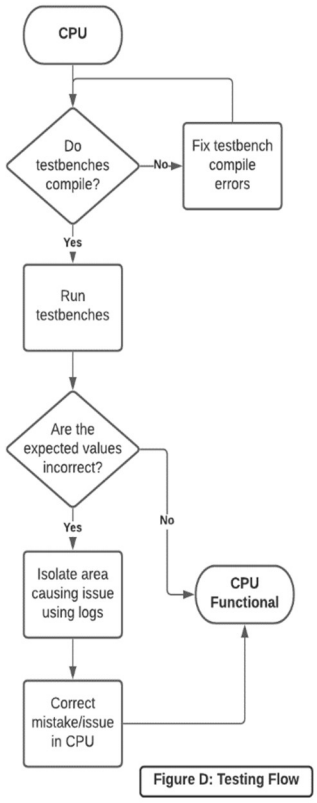


Figure D: Testing Flow

For most of the testbenches, systematic testing was used to eliminate bias and test a large number of cases. The data that was systematically generated (in most cases using an arithmetic series) came from a random start point for the instructions to do this. Stratified testing was also used for some instructions to test cases which were known to possibly cause issues. For example, this included making sure offsets could handle negative numbers, testing edge-cases and overflows for unsigned and signed instructions to make sure they functioned correctly.

The testbenches included large programs which tested many cases rather than many programs testing only a few cases. This was purely for the sake of efficiency. However, it did mean that the v0 output was rarely used as it only shows the value of a single register at the end of the program. It was preferable to store all the relevant registers in the data RAM and check it after the CPU halts. The MIPS reset address is a massive number (0xBFC0000), it is impossible for Verilog to simulate a RAM with this capacity. Therefore, only small parts of the RAM were simulated for the testbenches.

The specification states that a testbench should be run for each instruction. However, without a way to initialise the starting values for the general-purpose registers, it is impossible to test all the instructions completely independently from other instructions. Hence in most of the testbenches, it is required as a minimum that load word and store word instructions function correctly as they are the only way a CPU can interact with memory. Therefore, if the testbench is failing for a particular instruction, the user should firstly check that *lw* and *sw* are working before investigating further.

Though not required by the spec, an example program was included, which computed the Fibonacci series up to a term specified by data in the memory and stored the results in memory. This demonstrated implementing a realistic program that a client might use the CPU and added completeness to the testing.

Area and Timing Summary

Additionally, a timing and area analysis of this CPU was conducted using Quartus Prime Lite (*Intel, 2021*). This was done by mapping the CPU design made onto an Intel Cyclone IV E FPGA. In doing so, information regarding the worst-case resource usage and the timing was made available, so as to judge the efficacy of the CPU.

Area Summary

Analysis & Synthesis Summary

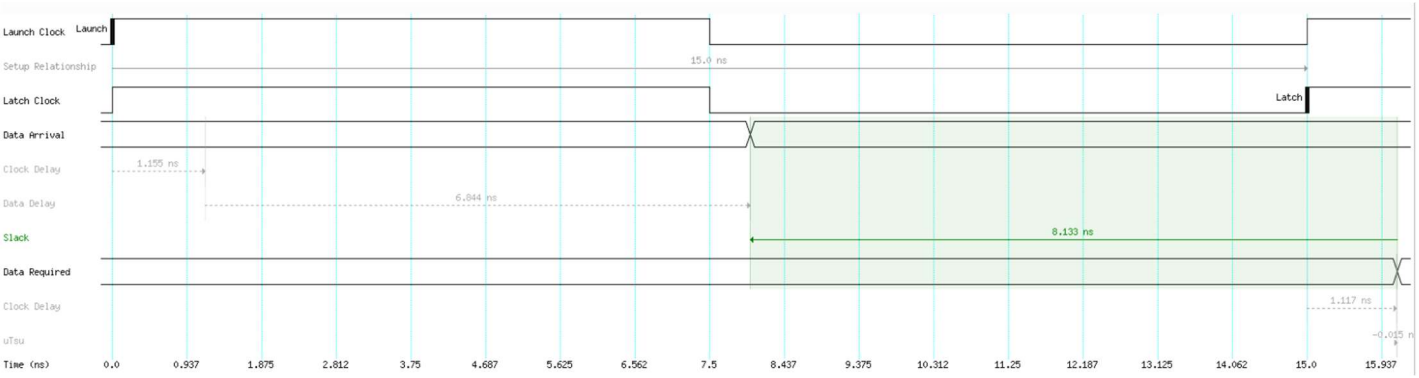
<<Filter>>

Analysis & Synthesis Status	Successful - Fri Dec 17 14:45:10 2021
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	mips_cpu_harvard
Top-level Entity Name	mips_cpu_harvard
Family	Cyclone IV E
Total logic elements	9,468
Total registers	1154
Total pins	198
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	16
Total PLLs	0

The area summary seen on the left, outlines the hardware within this CPU design. The most significant detail to look at is the total logic elements. This data is reflective of how many pieces of hardware this CPU involves. At 9,468 this is quite high, and this could be for a couple of reasons. Within the design, robustness and testability was preferred. Logical pieces, like the ALU, were created in a way that each instruction almost has its own hardware. For example, the ALU uses a case switch structure, to implement this physically would require many data paths (within the ALU) and multiplexers or control logic. The benefit of this is within its testability. Since the CPU was never intended to be built, and functionality was the primary focus, it was determined this to be a worthy trade-off. To reduce this number, a more hierarchical design could have been implemented that re-used many of the same data paths with only more control logic. For example, having a dedicated barrel shifter which could handle variable shifts.

Timing Summary

The timing analysis was run at a 1200mV and 85 degrees C. These were chosen as a worst-case scenario. In the end, the CPU was chosen to have a clock of 67 MHz with a period of 15ns. This clock had a slack of 8.3 ns. Theoretically, in a perfectly designed CPU, the clock would have 0 slack. However, since this CPU was designed for functionality and not performance, it was decided against reducing it further. This meant that it had some leeway for data arrival time and would therefore reduce the number of potential errors.



References

David A. Patterson University of California, Berkeley John L. Hennessy Stanford University (2014) *Computer Organization and Design: The Hardware / Software Interface.* 5th Edition. The Boulevard, Langford Lane, Lidlington, Oxford, Elsevier

Intel (2021) Quartus Prime Lite Edition (Version 21.1) [Software] Intel Corporation https://fpgasoftware.intel.com/?edition=lite

Kevin Liston (n.d.) *MIPS: Reference Sheet*. https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_help.html

MIPS Technologies Incorporated (2001) *MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set.* Mountain View, California. https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf

Charles Price (1995) *MIPS IV Instruction Set*. MIPS Technologies Incorporated. https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf

Open Cores. (2018) *Plasma – most MIPS I(™) opcodes.* https://opencores.org/projects/plasma/opcodes [Accessed 12th December 2021]

Tiago Bozzetti. (2014) *Mips Converter.* https://www.eg.bucknell.edu/~csci320/mips_web/ [Accessed 20th December 2021]