ELEC50010: Final Report
Imperial College London

Neel Dugar
CID: 01722569

Bertil de Germay
CID: 01767839

Ulusoy Pelin
CID: 01729307

Perkin Alexandria
CID: 01706843

Tanguy Perron
CID: 01722386

Thomas Lewis
CID: 01707143

December 21, 2020

1 Introduction

The task was to develop a MIPS CPU simulator, which could execute MIPS-1 little-endian binaries, alongside a testbench and set of test cases. The testbench was intended to test our processor, and to see whether the implementation itself was accurate and followed the specification. In order to tackle this problem, our project followed the Harvard single-cycle model which allowed our understanding of the concepts behind the MIPS CPU-design implementation to be improved. This microarchitecture is based on the execution of an entire instruction in a single cycle.

2 CPU architecture

The resulting Harvard CPU is structured around two main axes: the data path (including the program counter) and the decoder. The latter produces combinatorial control signals by decoding each entity going out of the instruction memory and encoding appropriate responses useful to drive the data flow around the CPU design. On the other hand, the data path is based on multiple modules such as registers, ALU, adders and multiplexers, that interact together with both the instruction and data memories to form a closed circuit where all the computing happens: using the information stored in the memory unit by moving, merging, or modifying it. The division of the design in such a manner produces one sequential block, the data path changes dependent on the instruction that is executed.

The overall design of the processor relies upon the efficiency of its submodules, namely the register file. This module acts as a fast memory unit containing 32 storage locations of 4 bytes each. Variables and addresses can be stored in those registers to then be either used for computation or stored back into the data memory.

The architecture of the CPU is as follows (see Figure 1 for reference): the program counter gives the instruction memory an address to fetch the instruction from. This instruction is processed by the decoder and all required control signals are set to arrange the data path in such a way that all information will flow around the CPU through each of the relevant submodules, without interfering with the restricted memory locations. After the control signals were set, the register file outputs the data from all required registers and sends it to the ALU and memory. The ALU itself performs most of the calculations (mathematical operations, bitwise operations, etc.) to either operate on values from registers and/or the immediate of the instruction, or on addresses that will be used for storage in the data memory or set the program counter to a specific address to read the instruction memory at.

The data memory can receive an address from the ALU, to which it will write or read the data from the registers. In the latter case, the output of this memory will go through a selection module called *LoadSelector*, which selects the exact bytes and bits of data that are required for each load instruction. Addresses and values out of the ALU or memory are then written back into the register file. J-type instructions do not use this regular path, as they do not require neither the ALU nor the data memory, instead Jump instructions directly modify the program counter to change the order of execution of the instructions stored in the instruction memory. Such address changes can come from either a register or the immediate of the instruction, with the addition of storing the address of the next instruction to be executed before jumping for linking jumps (JAL, JALR, as well as branch and link instructions). Branch instructions however do use the ALU only for comparison in order to determine if a jump should be executed. The CPU also contains a delay slot. This module is present right before the PC on the data path and delays all addresses flowing inside the PC by one cycle (and is therefore initialized at the reset vector and one instruction location). This allowed jump and branch instructions to take effect one cycle after being read by the CPU while allowing all other instructions to increment the PC normally without disrupting the execution of the program.

3 Design decisions

The data memory can receive an address from the ALU, to which it will write or read the data from the registers. In the latter case, the output of this memory will go through a selection module called LoadSelector, which selects the exact bytes and bits of data that are required for each load instruction. Addresses and values out of the ALU or memory are then written back into the register file. J-type instructions do not use this regular path, as they do not require neither the ALU nor the data memory, instead Jump instructions directly modify the program counter to change the order of execution of the instructions stored in the instruction memory. Such address changes can come from either a register or the immediate of the instruction, with the addition of storing the address of the next instruction to be executed before jumping for linking jumps (JAL, JALR, as well as branch and link instructions). Branch instructions however do use the ALU only for comparison in order to determine if a jump should be executed. The CPU also contains a delay slot. This module is present right before the PC on the data path and delays all addresses flowing inside the PC by one cycle (and is therefore initialized at the reset vector and one instruction location). This allowed jump and branch instructions to take effect one cycle after being read by the CPU while allowing all other instructions to increment the PC normally without disrupting the execution of the program.

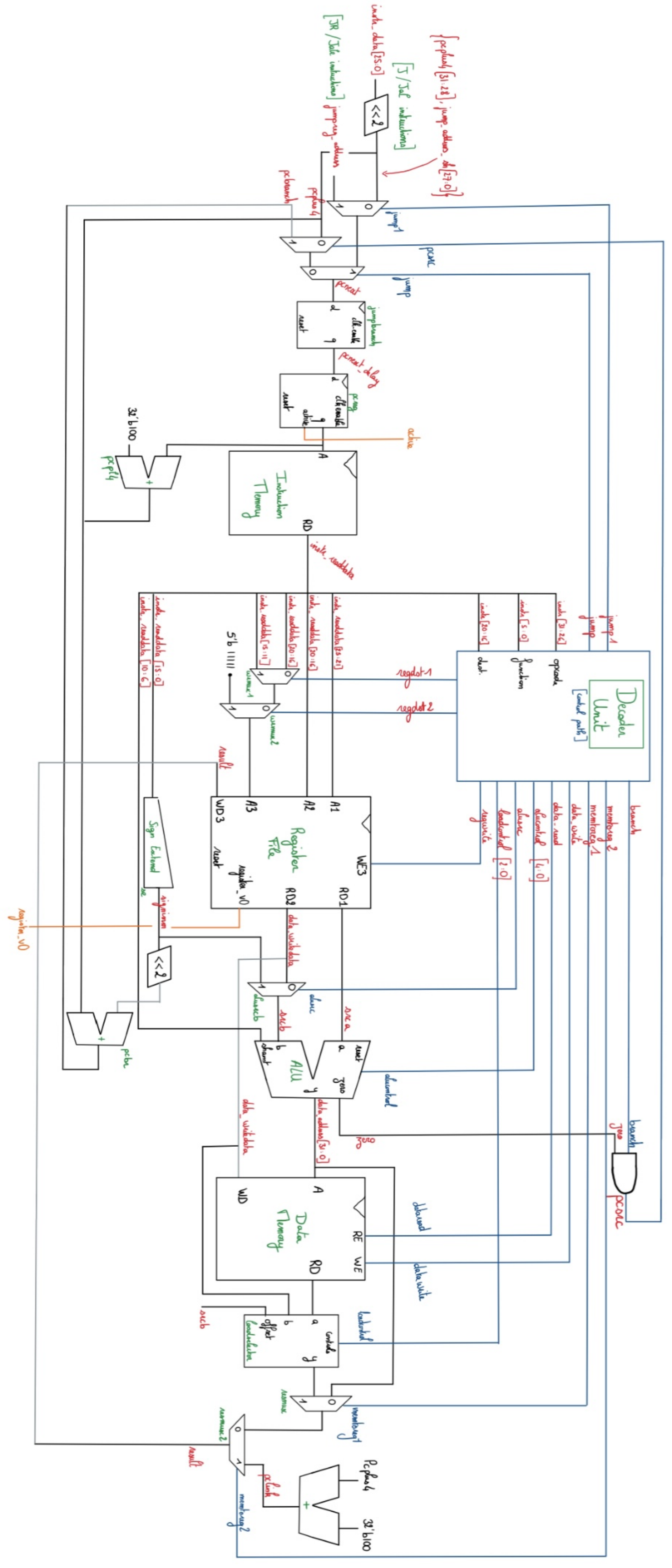

Figure 1: The architecture of the CPU.

4 Testing approach

The testing approach used is called unit testing and relies on the creation of mini-unit tests written in C, that were compiled and assembled into hex binaries. These unit tests were written to target particular instructions in assembly, while still being able to be executed on a reference machine like QEMU. Those hex binaries, containing both instruction and data memories to be loaded into a test bench, written in SystemVerilog load the binaries into the target CPU using a basic memory interface. References output to the test cases were also generated with QEMU user mode, this allowed the team to execute binaries and create reference files by capturing the exit code of these MIPS binaries. The test benches were then executed, and the target CPU's source files linked, this would produce a final output if the test bench is executed within the target cycle count, this output stored in register v0 should match the reference output if the specific case passed. The architecture is outlined in Figure 2.

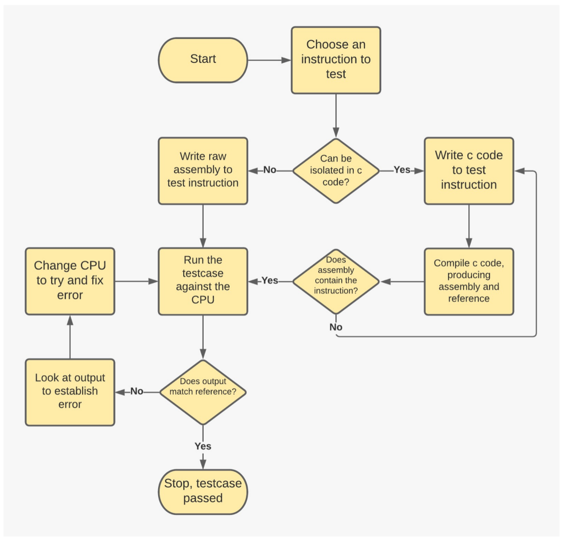
Start
Choose an instruction to test
Can be isolated in c code?
No
Yes
Write raw assembly to test instruction
Write c code to test instruction
Compile c code, producing assembly and reference
Does assembly contain the instruction?
Yes
No
Run the testcase against the CPU
Change CPU to try and fix error
Does output match reference?
No
Look at output to establish error
Yes
Stop, testcase passed

Figure 2: The test architecture.

We also implemented a continuous testing environment using Github Actions and a custom Docker container, which would generate the reference files and the simulations on the fly and run tests against our current committed CPU. We also used a linter and formatter in this continuous integration, using Google's open-source tool called Verible. For our test bench, we noticed we had a lot of iteration through test cases and so we ended up parallelising the test suite and the generation which lead to a $\sim$ 5x speedup on an 8-core processor, using bash background tasks (see Quartus analysis in Table 1).

Parameters' name	Value
Maximum clock frequency	2.55MHz
Core static power dissipation	85.70mW
Core dynamic power dissipation	23.18mW
Total logic elements used	13,371/28,848 (46%)
Combinatorial with no register	12,282
Register only	197
Combinatorial with a register	892

Table 1: Quartus Analysis (Cyclone IV E 'Auto')