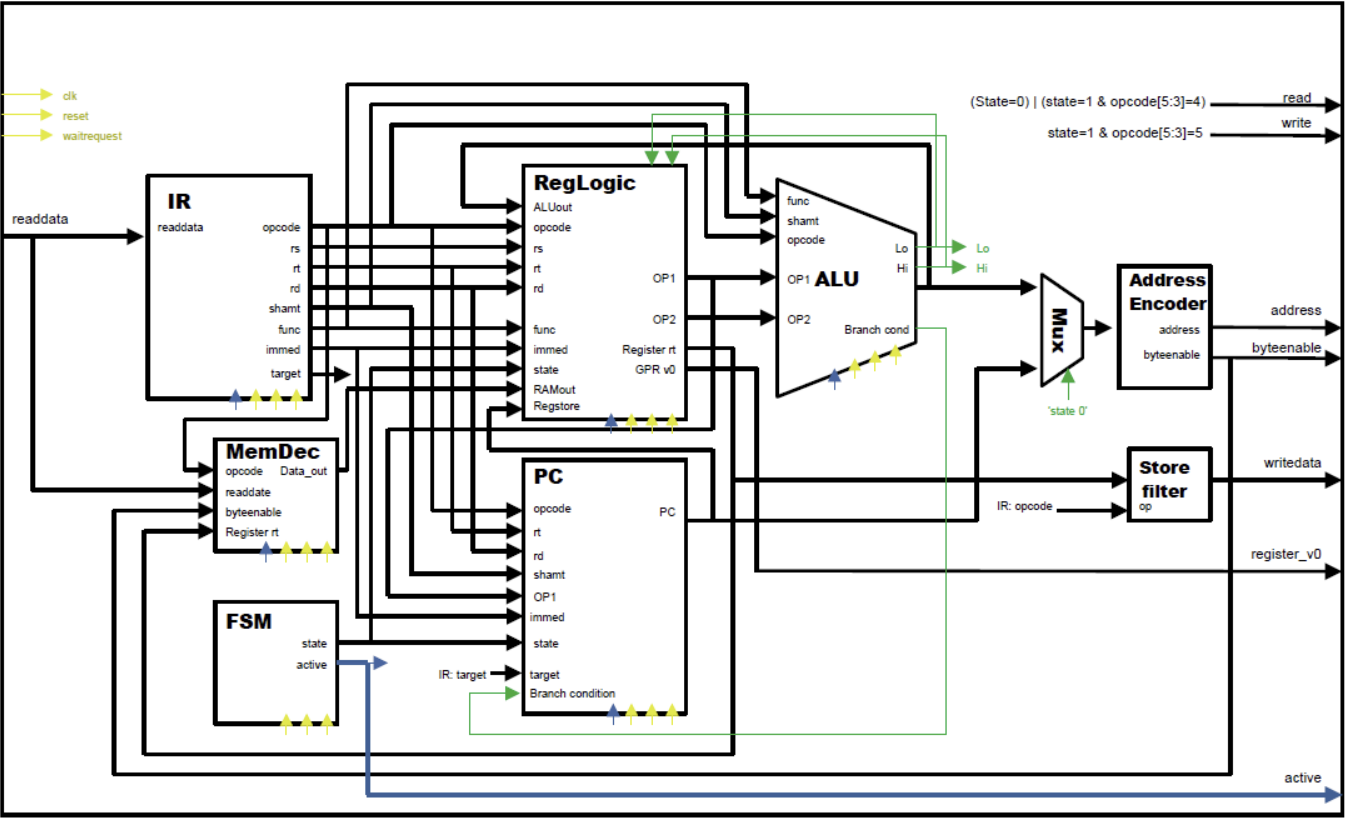
ELEC50010 Instruction Architecture & Compiler

The Second Quarter Coursework

Group6

**MIPS-compatible CPU Datasheet**

**Architecture**



**4**

**1**

**7**

**8**

**6**

**2**

**5**

**3**

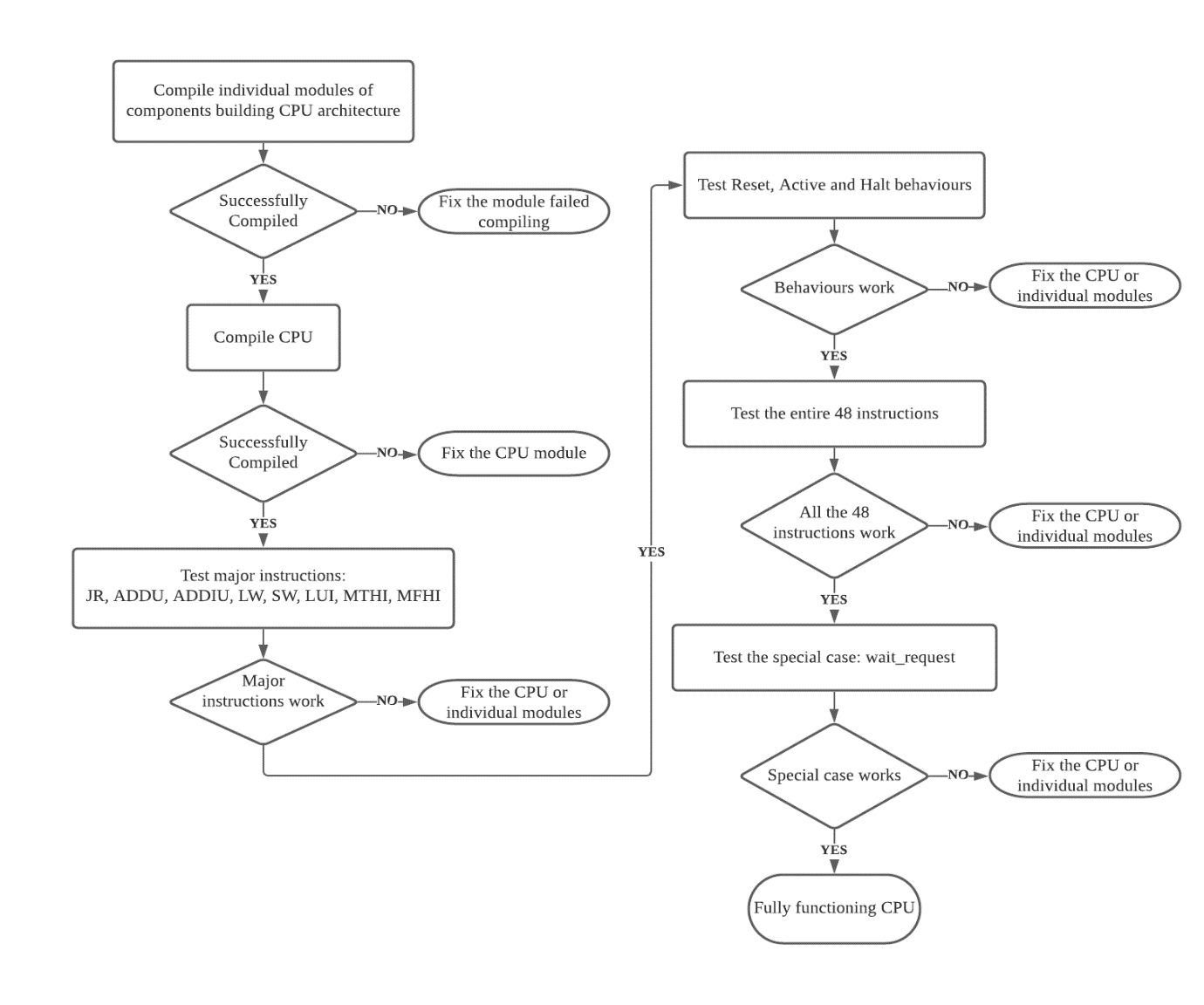
**Component analysis**

1. IR block
2. Finite State Machine
3. Memory Decoder
4. Register file and Control Logic Block
5. Program Counter
6. Arithmetic Logic Unit
7. Address Decoder
8. Store Filter

**Design Decision**

|  |  |
| --- | --- |
| Component Number | Design Decision & Purpose |
| ① | IR Block is designed to receive instruction input (32-bits) and provide opcode (6-bit), 1st register number (5-bits), 2nd register number (5-bits), destination register number (5-bits), shift amt (5-bits) and function code (6-bits) by performing sectional analysis. |
| ② | Finite State Machine is designed to receive memory address (32-bits) and opcode (6-bits) providing state (1-bit) and active (2-bits) as output.  The output state (2-bits) can represent 4 different states: Fetch, Decode, Execute and MEM/LINK. |
| ③ | Memory Decoder receives instruction input (32-bits), the value of 2nd register, rt (32-bits), opcode (6-bits) and byte-enable (4-bits) to produce RAM data out (32-bits) as an output depends on the byte-enable value.  Depends on the values of opcode (6-bits) and byte-enable (4-bits), the memory decoder performs I-type instruction. |
| ④ | Register file and Control Logic block receive ALU output (32-bits), opcode (6-bits), 1st register number, rs (5-bits), 2nd register number, rt (5-bits), destination register number, rd (5-bits), immediate (32-bits), shift amt (5-bits), function code (6-bits), RAM data out (32-bits) PC (32-bits) and state (2-bits). Depends on the opcode, it produces OP1 (32-bits), OP2 (32-bits), the value of the register rt (32-bits) and register v0 (32-bits). This component is an integrated component of register and control logic block. Once it receives an opcode, it checks whether the opcode is R-type or not and provides outputs by reading the register numbers and writing the address stored in the registers. |
| ⑤ | Program Counter is designed to receive opcode (6-bits), 1st register number, rs (5-bits), 2nd register number, rt (5-bits), destination register number, rd (5-bits), shift amt (5-bits), function code (6-bits), OP1 (32-bits), offset (16-bits), target (26-bits), control (4-bits), state (2-bits). Depends on the opcode (6-bits), the PC performs jump instructions such as jump (J) and branch instructions such as branch on equal (BEQ). |
| ⑥ | Arithmetic Logic Unit includes adder-subtracter logic block, multiplier block and divider block which performs addition, subtraction, multiplication, and division. Depends on the opcode (6bits), a suitable block will be chosen for the operation and provides ALU-out (32-bits), Lo (32-bits), Hi (32-bits) and branch-conditions (4-bits) as the outputs. |
| ⑦ | Address Decoder is designed to receive opcode (6-bits) and distinguishes the instructions into 4 types: load, store, link and other. Depends on the instruction type and opcode value, Address Decoder performs specific I-type instructions such as Load Byte (LB) and provides memory address (32-bits) and byte enable (4-bits) as the outputs. |
| ⑧ | Store filter is designed to receive opcode (6-bits) and performs I-type store instructions such as store byte (SB), store halfword (SH) and store word (SW) |

**Testing Approach – Flow Chart**



We took a stage-by-stage test and fix approach which started while we were working on the individual modules, using testbenches to verify their functionality. Once all the modules were working, we compiled the full CPU and then began testing the individual instructions. We initially focused on JR, LUI, LW, and other major instructions (as detailed in the chart), as these are required for testing most of the other instructions. Furthermore, we had to implement the MFHI and MFLO instructions in order to test for multiplication and division operations, as well as MTHI and MTLO. Before we tested the other instructions, we tested the functionality of RESET, ACTIVE, and HALT behaviour as the CPU needs to exhibit proper runtime behaviour to be able to test other functions.

Once the CPU was superficially working and compiling, we wrote 2-4 test cases for each of the 48 instructions by creating a general template file that combines the functionality of a RAM module and a testbench.

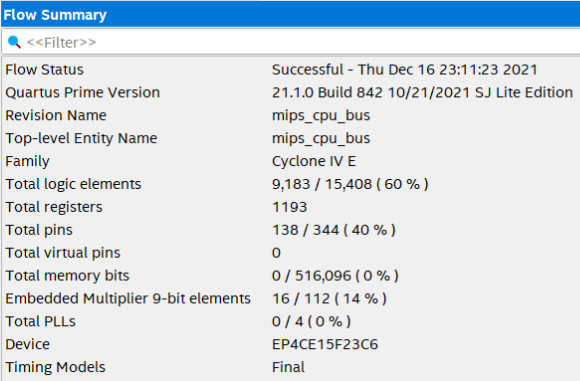
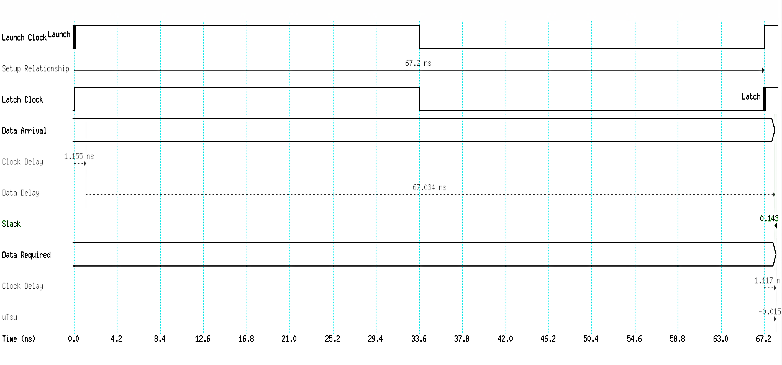
For the RAM functionality, we mapped the addresses given by the CPU to array addresses by subtracting the reset vector (0xBFC00000) from the address and dividing by 4, with a special exception for address 0 which returns 0 for CPU HALT. The RAM then returns the corresponding value stored in the array. We did not include more than 20 memory locations in the testbenches as they were sufficient for case testing and were able to compile and run significantly faster compared to a full RAM module.

For the testbench section, we mapped out on paper the expected CPU behaviour for the different instructions and then used an instruction-to-hex converter to insert these instructions into the memory. We then generated a clock which repeats 104 times and then asserts that ACTIVE goes low to detect timeouts. To start the CPU, we assigned RESET to HIGH for one cycle and then executed the instructions. If the CPU passes the first assert, we then assert that register v0 or memory (depending on the instruction being tested) contain the correct pre-calculated result(s).

As we had written 139 testbenches, some automation was necessary to test the CPU. We implemented a testing script according to the specifications, and separate debugging scripts. The testing script takes a command line parameter for the path to the directory containing the CPU Verilog files and a second, optional parameter for the instruction that is being tested, which defaults to testing all instructions if none was entered. The script then runs a check to ensure that the instruction is valid before proceeding. The script file contains an array called TESTBENCHES with individual entries for each testbench. To implement a 2D array, each array element contains a string with the name of the testbench, the instruction being tested, and any comments, delimited with a semicolon. The script then runs a ‘for’ loop over every element in the array, delimits the string into separate variables, and runs an ‘if’ statement to check if the testbench matches the instruction(s) being tested. If so, the script then compiles the testbench together with all the Verilog files in the CPU directory and checks the exit code. If the exit code is not 0, the script returns a Compilation Error Fail for that testbench and continues to the next iteration of the loop. If the testbench compiles successfully, the script then attempts to run the executable and once more checks the exit code, which would be non-0 if any of the asserts failed and correspondingly either returns a Pass or a Runtime Error Fail. Throughout the script, stdout was redirected to NULL so the only output in the terminal was the Pass and Fail messages.

Once all instructions were successfully tested, additional tests were conducted to check WAITREQUEST functionality by randomly manually inserting HIGH WAITREQUEST signals and checking for expected behaviour.

**Area and timing summary – “Cyclone IV E’ Auto” variant in Quartus**



We compiled the CPU in the Quartus synthesis tool using the worst-case conditions (1,200mV and 85°C). The fitter analysis showed that the CPU uses 9,183 logic elements in total, which make up 60% of the logic elements available on the FPGA. This was expected as area optimisation was not a priority during the design process.

The timing analysis tool initially showed that the CPU had negative slack, which signified that the default clock rate was too high. To fix this, we used the timing analyser to create a new clock running at a lower rate. After a few iterations, we optimised the slack down to 0.143ns with a clock period of 67.2ns, which resulted in our CPU being able to clock at a maximum rate of 14.88MHz. This speed is slow compared to modern CPUs available in the market as our model has not been optimised for clock rate, but rather designed for functionality.

If we had more time and resources available, there are some potential improvements we could have made to the CPU. Such improvements would involve pipelining the CPU (which however was advised against for the scope of this project), optimising the combinatorial datapath to be shorter, and running more analysis software to optimise the CPU.