ELEC50010 Instruction Architectures & Compilers

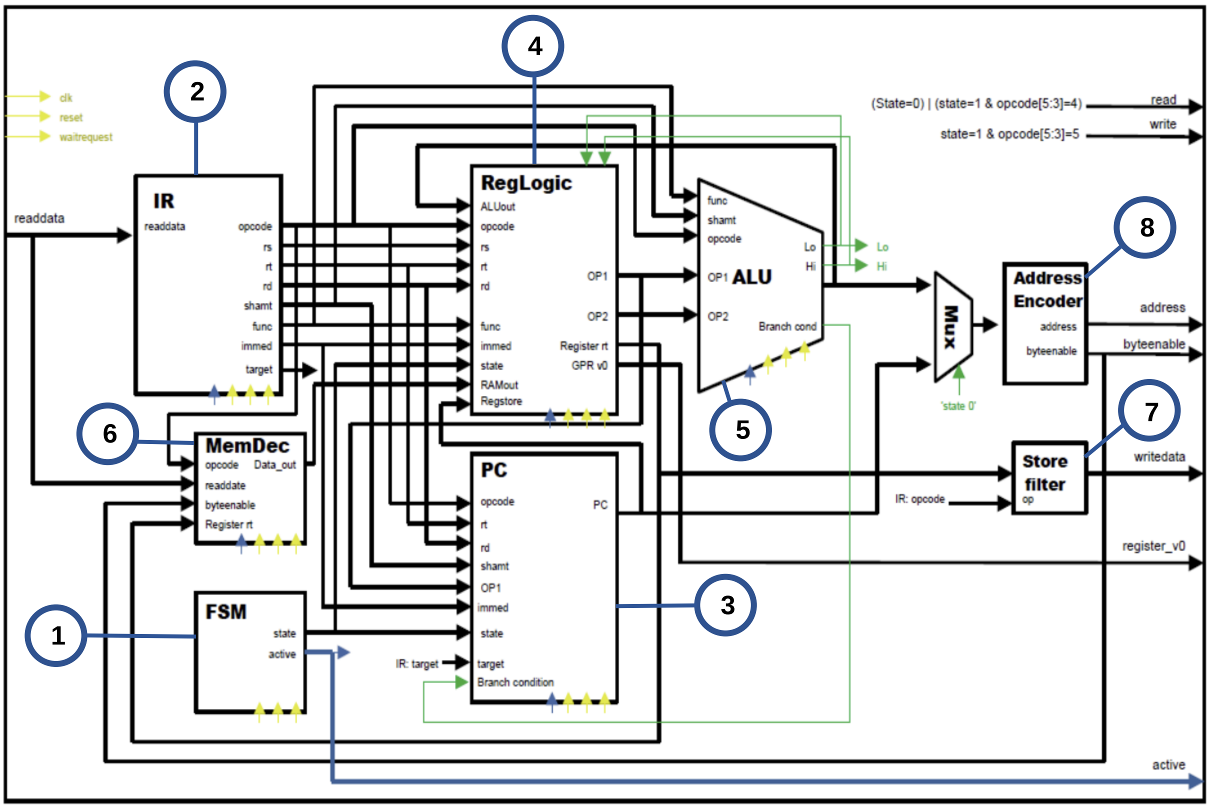
Second Quarter Coursework

Group 6

Imperial College London

MIPS-compatible CPU “TurboOlaf 3000” Datasheet

Architecture



Chronological Component analysis – Cycles and Stage

① Finite State Machine

② IR Block

③ Program Counter

④ Register file and Control Logic Block

⑤ Arithmetic Logic Unit

⑥ Address Encoder

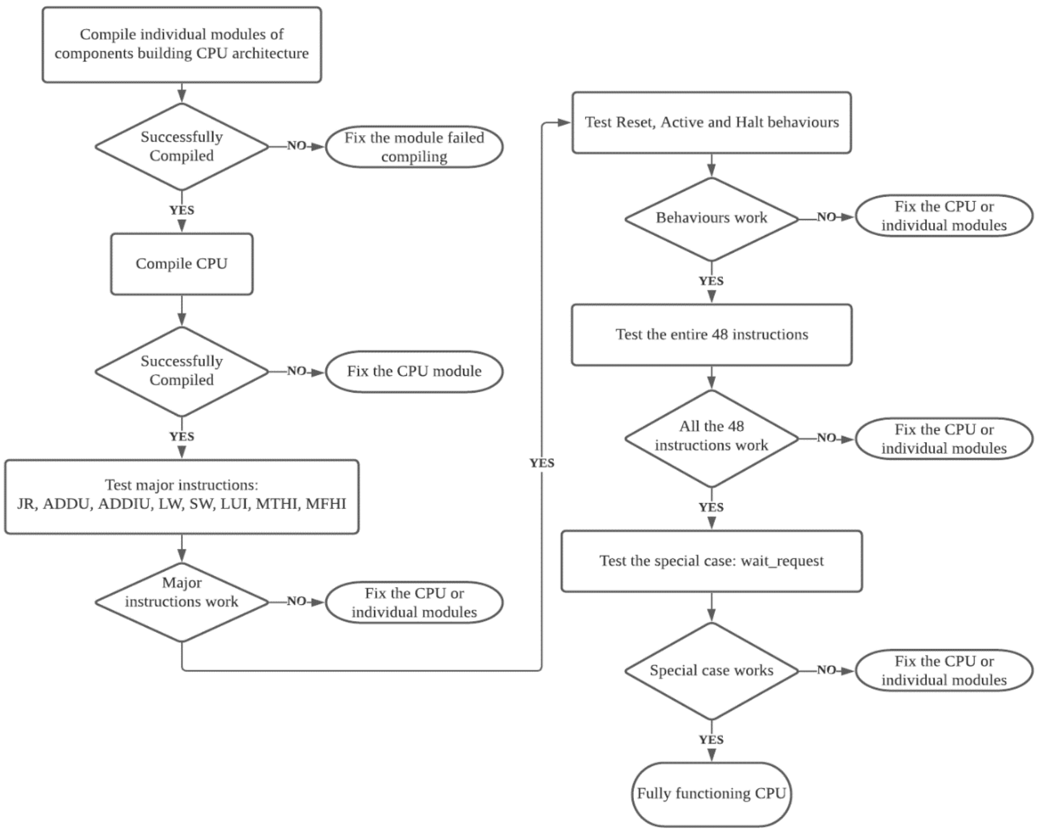
⑦ Store Filter

⑧ Memory decoder

Design Decision

Component Number	Design Decision & Purpose
①	Finite State Machine is designed to receive ***memory address*** and ***opcode*** providing ***state (2-bit)*** and ***active*** as output. The output state can represent 3 different states: FETCH, EXEC1, and EXEC2. The State Machine only cycles to EXEC2 when during a Load instruction since an extra cycle is needed to receive data from memory.
②	IR Block is designed to receive ***instruction input (32-bits)*** and provide ***opcode***, ***register index r_s, register index r_t***, ***destination register r_d***, ***shift amt***, ***function code*** and ***immediate***. It achieves that by splitting the instruction bits in 2 different ways. These outputs are then used by various blocks like the Register File, the ALU and the PC. In order to make the CPU faster (keep the CPI down), the IR must produce its outputs on EXEC1, but it must also retain these same outputs in EXEC2. This is accomplished by immediately using the instruction in EXEC1 and storing the instruction in a register for EXEC2.
③	The PC is in charge of producing the next instruction address. Normally the next address is PC+1 (or 4-byte addresses in this case) but the PC performs jump instructions such as jump (J) and branch instructions such as branch on equal (Whether the branch happens is determined by certain conditions). For Jump, the new address is independent and for Branch the current address is used as a "base". This functionality is crucial for the existence of functions and subroutines and enables the CPU to execute significantly more complicated programs. Finally, some instructions demand the PC stores the address it jumped from so that it can be used to jump back when a subroutine ends for example. This address is stored in the Register File.
④	Register file and Control Logic block receive ***ALU output (32-bits)***, ***opcode***, ***register_s (r_s) index***, ***register_t (r_t) index***, ***destination register (r_d) index***, ***immediate***, ***shift amt***, ***function code***, ***RAM data out***, ***Regstore (coming from PC)*** and ***State***. This component includes the register file and generates many control signals which determine the **Regfiles** inputs and outputs based on the inputs listed above. Thus, what is written into the register file, and when writing is enabled is controlled within this block. This block also determined Op1 and Op2, which are the inputs to the ALU.
⑤	Arithmetic Logic Unit includes adder-subtracter logic block that performs the add, sub and logic instructions. There is a multiplier and divider block which performs multiplication and division. Finally, there is also a shift block that executes shift instructions. The ALU has 3 outputs. The main ***ALU output*** is determined according to the instruction opcode. The ***Hi*** and ***Lo*** outputs always contain the current result in the Hi and Lo registers. The ***Branch Conditions*** are always computed and outputted for every instruction and the PC only uses the ones it needs.
⑥	Address Encoder is designed to receive ***opcode*** and distinguishes the instructions into 4 types: ***load, store, link and other***. The main function of this Block is to receive an address and based on the opcode adjust output signals ***memory address (32-bits)*** and ***byte enable(4-bits)*** which are both connected to memory.
⑦	Store filter is designed to filter out data coming from the register file before it is sent to memory, based on the type of Store instruction the CPU is implementing (Word, Halfword, Byte).
⑧	Memory Decoder receives data from RAM during Load instruction. Based on the byte enable signal produced in the previous cycle, it filters out unnecessary bytes so that the data is correctly presented to the register block where it is stored.

Testing Approach – Flow Chart


Compile individual modules of components building CPU architecture
Successfully Compiled
NO
Fix the module failed compiling
YES
Compile CPU
Successfully Compiled
NO
Fix the CPU module
YES
Test major instructions: JR, ADDU, ADDIU, LW, SW, LUI, MTHI, MFHI
Major instructions work
NO
Fix the CPU or individual modules
YES
Test Reset, Active and Halt behaviours
Behaviours work
NO
Fix the CPU or individual modules
YES
Test the entire 48 instructions
All the 48 instructions work
NO
Fix the CPU or individual modules
YES
Test the special case: wait_request
Special case works
NO
Fix the CPU or individual modules
YES
Fully functioning CPU

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.

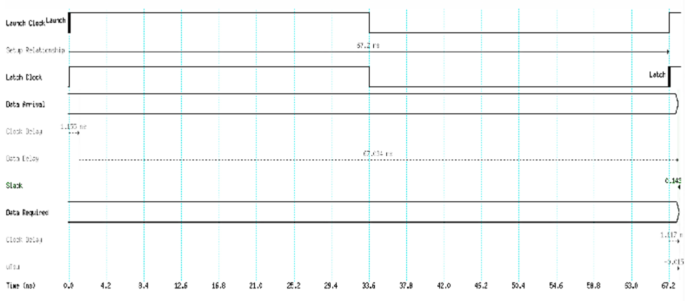
For the RAM functionality, we mapped the addresses given by the CPU to array addresses by subtracting the reset vector (0xBFC00000) from the address and dividing by 4, with a special exception for address 0 which returns 0 for CPU HALT. The RAM then returns the corresponding value stored in the array. 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.

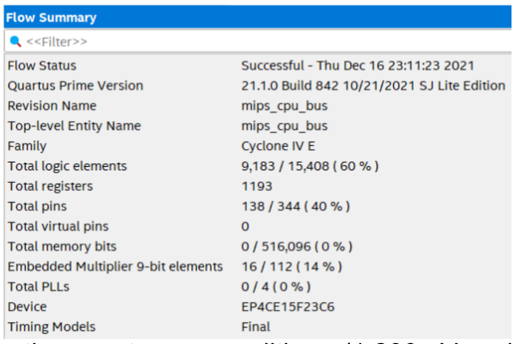
For the testbench section, we mapped out on paper the expected CPU behaviour for the different instructions and then used an instruction-to-hex converter to insert these instructions into the memory. We then generated a clock which repeats 10^4 times and then asserts that ACTIVE goes low to detect timeouts. 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




Flow Summary

Flow Status	Successful - Thu Dec 16 23:11:23 2021
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	mips_cpu_bus
Top-level Entity Name	mips_cpu_bus
Family	Cyclone IV E
Total logic elements	9,183 / 15,408 (60 %)
Total registers	1193
Total pins	138 / 344 (40 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	16 / 112 (14 %)
Total PLLs	0 / 4 (0 %)
Device	EP4CE15F23C6
Timing Models	Final

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.