MIPS CPU Harvard Data Sheet

AMTeam-00

Section 1: Diagram of CPU Architecture

The design for the Harvard Interface involved a top-level interface file *mips_cpu_harvard.v* which instantiates lower-level Verilog modules.

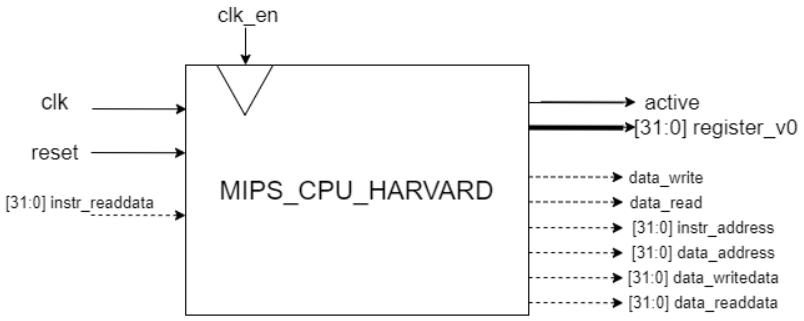

Figure 1a: Top level main CPU architecture

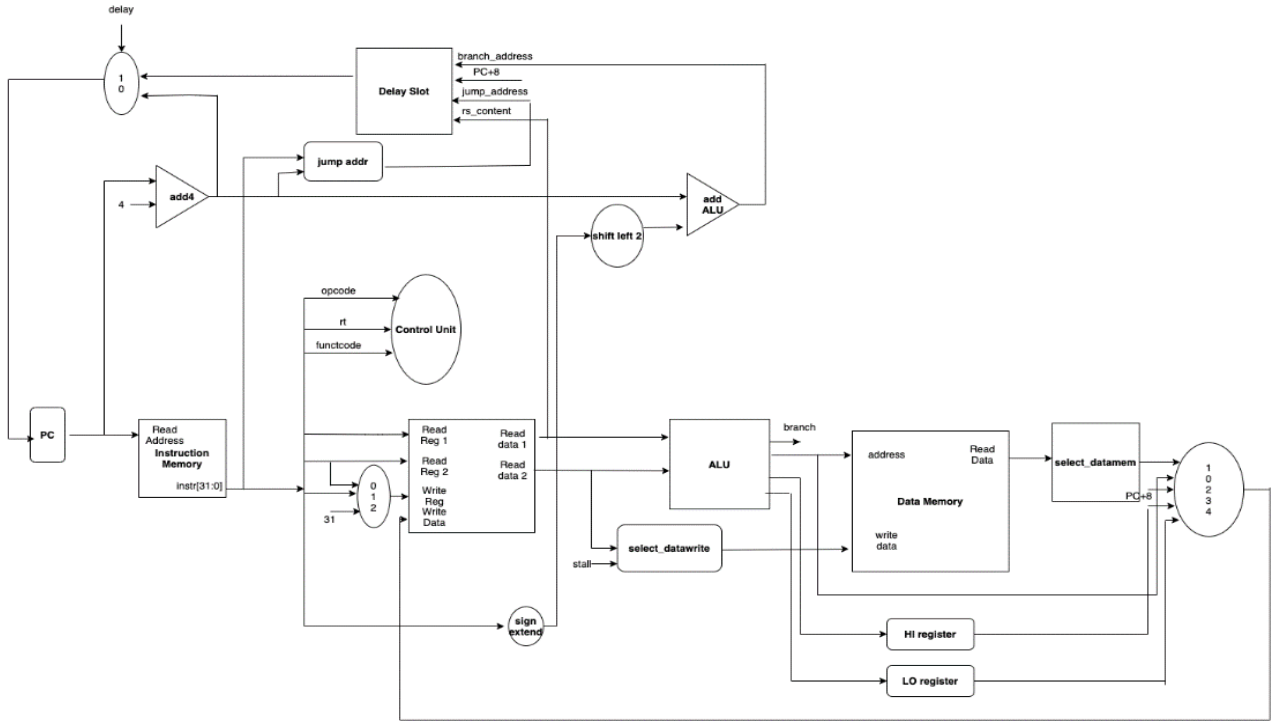

Figure 1b: Lower Level Verilog Modules

Section 2: Overall CPU Architecture

2.1 CPU Instruction Set

With regards to the design specifications, the team implemented a single cycle MIPS processor capable of executing the 32-bit little-endian MIPS1 instruction set (defined by the MIPS ISA Specification).

2.2 Datapath

Datapath components in the Harvard CPU include the program counter (PC), register file, virtual memory as provided by the testbench and Arithmetic Logic Unit (ALU).

General process of executing a program is as follows:

1. Instruction fetching: utilising the address in the PC to fetch the instruction from memory.
2. Instruction decoding: determining the fields within the instruction and sending control signals to circuitry from the control unit block.
3. Instruction execution: perform the operation indicated by the instruction.
4. Memory: access data memory interface if required

5. Writeback: update contents of register file.
6. Update the PC to hold the address of the next instruction.

2.3 CPU Interface

The team implemented the MIPS Processor using the Harvard interface.

The Harvard interface provides separate instruction and data memory interfaces. Instruction memory and data memory support combinatorial read paths and single cycle write paths.

2.4 Reset behaviour

The result of reset signal being set high is that all registers are set to zero, and the next instruction to be executed is at address 0xBFC00000. The CPU does not initiate any memory read or write transactions while reset is high.

2.5 CPU Halt

The active flag in the Harvard interface serves as an indicator that the CPU has finished executing instructions. It is set low on the next rising edge of the clock if instruction address is equal to 0.

Section 3: Design Decisions

3.1 Harvard Interface

3.1.1 Implementation of Delay Slot

The objective of the delay slot block was to implement the delay slot—the instruction after a jump or branch instruction is executed before the jump or branch is executed.

Inputs to the delay slot block include the branch target address computed during branch instruction, jump target address computed during jump instruction, contents of register Rs, and the address of the second instruction following the branch or jump (*PC+8*). The delay slot block selects an input based on the control signals Jump, JR from the control unit and Branch from ALU, which asserts these signals during jump, jump register and branch instructions respectively.

When the control unit detects a jump or branch instruction, it outputs a combinatorial signal *delay_early.* On the rising edge of the clock, the signal *delay* will be updated to the value of *delay_early,* which would be high. *delay* is the control signal to a two-input multiplexer that selects between the output of the delay slot and *PC+4.* Therefore, as *delay* is only asserted in the cycle following the jump or branch instruction, the instruction immediately following the jump or branch instruction is always executed before the jump or branch to the target instruction takes place.

3.1.2 Implementation of Multiply and Divide Instructions

The inclusion of multiply and divide instructions in the MIPS ISA necessitated the creation of registers HI and LO that were wired as outputs of the ALU. Although MFHI and MFLO instructions were not included in the specification, the team decided to include these instructions to facilitate the execution of the testbench in testing multiply and divide instructions.

3.1.3 Implementation of JR Instruction

Implementation of the JR instruction involved the addition of a port into the multiplexers to select Write Register and Write Data into the Register files. A wired connection from *Read Data 1* to the delay slot was included for implementation as well.

3.1.4 Implementation of Partial Load Instructions

Partial Load instructions are comprised of LB, LBU, LH, LHU. The team implemented the *select_datamem* module in order to execute these instructions. Inputs to the module are the two least significant bits of *data_address,* the 32-bit output from data memory *data_readdata,* and opcode of the instruction. The contents of the 8-bit byte or 16-bit halfword at the memory location specified by the effective address are fetched, extended and written into the register file through assignments in the *select_datamem* module.

3.1.5 Implementation of Partial Store Instructions

Partial Store Instructions are comprised of SB and SH. As the harvard interface did not possess a *byteenable* signal to the memory interfaces, the team decided to implement the partial store instructions by executing the instruction

over 2 clock cycles. The first clock cycle was to read the word at the memory location specified by the effective address (*data_readdata*) and capture the word using a register. The second clock cycle was to write the least-significant 8-bit byte or 16-bit halfword of the register rt to be stored in data memory at the location specified by the effective address. The concatenation and replacement of 8 bits or 16 bits in *data_readdata* was implemented by the *select_datawrite* module.

In order to execute the instructions over 2 clock cycles, a *stall* signal was added in the interface, and was set high for the first cycle of the partial store instructions. As an input to the control unit, *stall* determined if a memory read or write operation would be executed during the first and second cycles of the instruction respectively.

Section 4: Testing

4.1 Test Summary

The completed CPU is tested using a combination of C++, Verilog and bash scripts. There are handwritten test cases and randomly generated test cases, where the manual test cases are searching for specific bugs, and the randomly generated values are a sanity check for the general case and/or identifying unusual errors. Each test case is a modified assembly file, containing an optional description, data values and the expected value of register v0, which gets turned into assembly by a C++ program. Expected outputs were generated via C++ and external resources (QTSpim), in conjunction with the specification[1].

4.2 Testing Flow

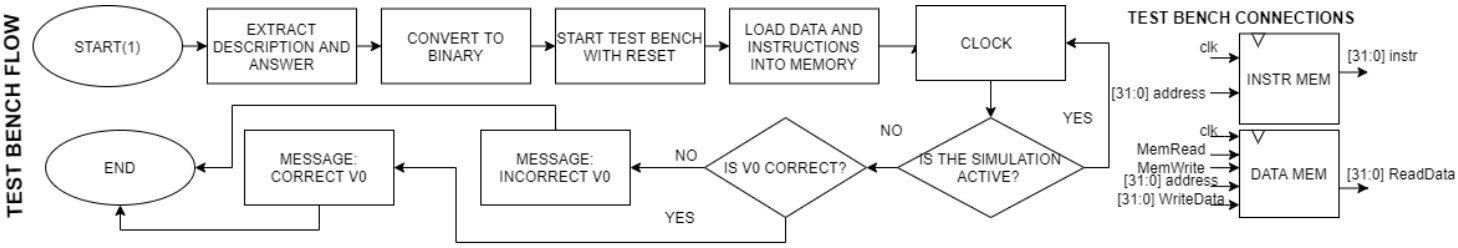

Figure 2: Test Bench flow

The test bench aims to perform functional and integration tests on any compatible Harvard interface. All test cases are turned into three binaries (data, instruction, and answer) using a basic assembler and numeric labels; a disassembler cross checked the values. The binaries are placed in separate data and instruction memories, where offsets are used to simulate a range of memory addresses. These are joined to the CPU in the main testbench, where the CPU is reset and run. Instructions were tested in order of frequency with other test cases; for this test bench the key instructions are ADDIU, LW, and JR as they can be used to modify registers easily, so are useful in branch/jump tests. Initially arithmetic operations were tested, then branches and memory, in order of implementation. If the CPU ends with the expected output without timing out, it is deemed to have passed.

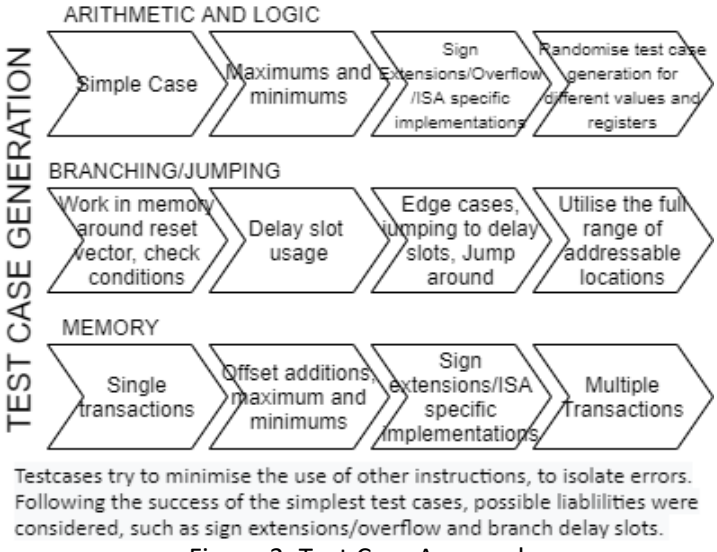

Figure 3: Test Case Approach

For each test case, the CPU is recompiled with new data and instructions, stored in separate memory locations. The testbench will finish once inactive or too many cycles have elapsed. Free form comments describe tests and contain extra information to aid debugging - the value of V0 may be correct even if it is looping infinitely. Testbench and CPU outputs are captured in an output folder as a stdout file which is parsed for the correct value and the keyword success with the testbench prefix. Test cases can be run individually, as part of an instruction set or all together; each test case will also generate a waveform file which can be used to identify timing problems or unexpected changes.

[1] https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf

Some key values to be tested for extension and overflow are 0xffffffff and 0x00000000, whilst branching and jumping are tested across the +/-128KB and 256MB ranges respectively. In each case they were tested with signed values (for traversing forwards and backwards) and a pre determined memory file is loaded to an offset location in instruction memory. The random test cases also make use of different register locations, to increase the parts of the CPU being tested.

Note that individual components also have test benches for unit tests to aid with CPU specific debugging; these were used in the early stages of development, before using assembly test cases.

Section 5: Test Approach Flow

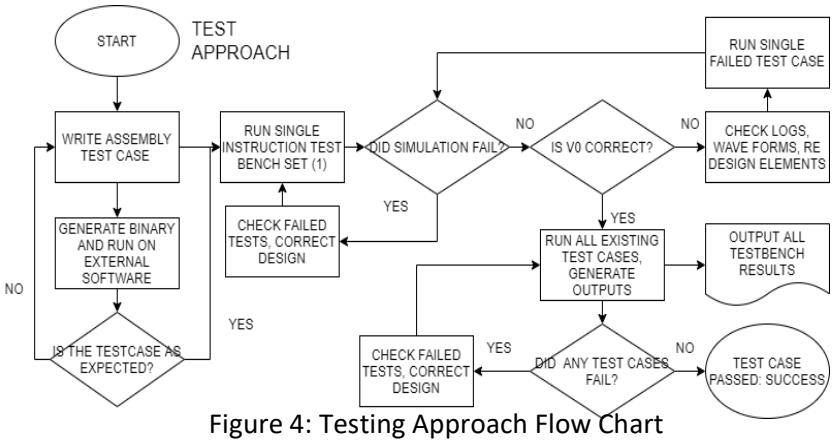

Figure 4: Testing Approach Flow Chart

Having been developed separately from the CPU, the testbench is able to maintain objectivity. The CPU developers can run the test script for instructions and identify missing functionality without knowing the inner workings of the test bench. An iterative approach ensures the instructions worked as intended without compromising existing features; once one instruction worked, all tests would be re-run.

The approach is followed for each instruction, regardless of type. Following the success of all instructions, some scripts with common functions such as looping were tested.

Section 6: Area and Timing Summary

6.1 CPU Timing Analysis

Analysis of CPU speed using Quartus Prime Lite was run on the MIPS CPU. The maximum clock frequency obtained for the Harvard CPU was 8.1 MHz, which indicates that the minimum clock period should be 123.5 ns.

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	8.1 MHz	8.1 MHz	clk	
2	67.82 MHz	67.82 MHz	instr_readdata[26]	
3	83.95 MHz	83.95 MHz	instr_readdata[0]	

Figure 5: Timing analysis from Quartus

6.2 CPU Area Analysis

Due to the relatively larger area required to instantiate the full MIPS instruction set architecture, the Harvard CPU takes up 48% of total available logic elements.

Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Total logic elements	13,948 / 28,848 (48 %)
1	-- Combinational with no register	12794
2	-- Register only	6
3	-- Combinational with a register	1148
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	8825
2	-- 3 input functions	4187
3	-- <=2 input functions	930
4	-- Register only	6
4		
5	Logic elements by mode	
1	-- normal mode	11341
2	-- arithmetic mode	2601
6		
7	Total registers*	1,154 / 30,421 (4 %)
1	-- Dedicated logic registers	1,154 / 28,848 (4 %)
2	-- I/O registers	0 / 1,573 (0 %)
8		
9	Total LABs: partially or completely used	987 / 1,803 (55 %)
10	Virtual pins	0
11	I/O pins	198 / 329 (60 %)

Figure 6: Area analysis from Quartus

6.3 CPU Power Analysis

The total thermal power consumption of the CPU was similarly analysed using the Power Analyser Tool in Quartus. The Harvard CPU consumes 184.48 mW of power.

Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Sun ... 20 17:52:31 2020
Quartus Prime Version	19.1.0 Build 670 0...19 SJ Lite Edition
Revision Name	mips_cpu_harvard
Top-level Entity Name	mips_cpu_harvard
Family	Cyclone IV E
Device	EP4CE30F23C6
Power Models	Final
Total Thermal Power Dissipation	184.48 mW
Core Dynamic The...ower Dissipation	0.00 mW
Core Static Therm...Power Dissipation	86.02 mW
I/O Thermal Power Dissipation	98.46 mW
Power Estimation Confidence	Low: user provided...t toggle rate data

Figure 7: Power analysis from Quartus