Data Sheet

Team 15

Leom, Melody — Pantelopoulos, Archontis-Angelos — Pondaven, Alexander — Tan, Si Yu Y — Vyronos, Giorgos

1. Overall CPU Architecture

Introduction

MIPS15 is a MIPS-Compatible CPU that is able to interface with Avalon Compatible Memory-Mapped interface. It is designed with functionality in mind and has not been optimised for speed and area. MIPS15 is able to execute the instructions stated in the coursework requirements as defined by the MIPS ISA Specification (Revision 3.2) for a 32-bit little-endian MIPS1.

States

MIPS15 is a bus based interface, so in order to fetch instructions and data over the same interface, MIPS15 is implemented using a multi-cycle finite-state machine to execute the instructions. There are 5 different states in MIPS15's finite-state machine: FETCH, LOAD, MEM, EXEC, HALTED.

Datapath

At the beginning, MIPS15 will be at a HALTED state, with the output active signal being deasserted. Upon assertion of the input reset signal, MIPS15's Program Counter (pc) will be loaded to the reset vector, 32'hBFC00000. The state will be updated to the FETCH state in the next clock cycle.

At FETCH state, MIPS15 asserts output read signal to the Memory. This will trigger a read request to the Memory, fetching the instruction at address 32'hBFC00000 in memory. MIPS15 CPU will then change its state to the LOAD state in the next clock cycle.

At the LOAD state, MIPS15 will continue to assert output of read signal high and will maintain this state for as many clock cycles depending on the input waitrequest signal from the memory. The instruction will be loaded into the instruction register (instr_register). The state will then go to MEM in the next clock cycle.

At the MEM stage, the Arithmetic Logic Unit (alu) in MIPS15 will calculate the memory address for load and store instructions. In order to read from or write data into memory, MIPS15 will remain in this state for as many clock cycles required till waitrequest gets deasserted. read (for load instructions) and write (for store instructions) signals will remain high in this state in order to execute memory related instructions. Other than load and store instructions, all other instructions will only be executed during the EXEC state. MIPS15 CPU will then change its state to EXEC in the next clock cycle.

At the EXEC stage, the MIPS15 control unit will send out various control signals to the different modules in the CPU and execute the instruction. The processors's 32 general-purpose registers are stored in the reg_file module, which will only be edited in EXEC state. Along with other registers such as reg_hi, reg_lo, pc, target_address_holder, the clk_enable signal from the control unit only enables changes to these registers during the EXEC state. The datapath uses the alu to evaluate various arithmetic instructions as well as the branch condition during a branch instruction. pc will also be updated in this stage with the next address. MIPS15 CPU will then go back to the FETCH state for the next instruction or HALTED state if attempting to execute address 0. For branch and jump instructions, the instruction following the branch/jump instruction will be executed first before the branch/jump occurs. This is known as the delay slot. MIPS15 handles this by using a register in the cpu, which holds the value of delay, and another register target_addr_holder, which holds the target address for the branch or jump instruction

Control Unit

MIPS15's control unit (control) takes in the CPU state and the instruction, to determine which instruction needs to be executed and sends out the appropriate control signals to the different modules in the CPU.

For further details on each module, please refer to the individual verilog files in the rtlmips_cpu folder.

2. CPU Design Decisions

MIPS15 takes a modular implementation approach. Each module of the cpu such as reg_file, alu, are all implemented in separate verilog files and are connected together in the top-level mips_cpu_bus module. Each individual module was tested comprehensively to ensure intended functionality before connecting them together. This allows for easier debugging and modification of MIPS15 to execute more instructions as required.

MIPS15 takes a multi cycle approach to allow quicker transition from harvard to bus implementation. A harvard implementation with separate instruction and data memory took a single cycle to complete every instruction. A bus based implementation would require a single memory to hold both instruction and data. Most of the harvard implementation code was preserved in the EXEC state of MIPS15. The instruction register is needed so that MIPS15 is still able to execute the instruction in the EXEC state after a load instruction is executed in MEM state.

3. Testbench Approach

Test Cases Creation Process

MIPS assembly files with simple test cases were created to test each instruction. Each instruction has multiple test case files to test every possible edge case. Each instruction had a list of possible scenarios in which it could fail. For example, the instruction "beq" has test cases for branching forwards, branching backwards, and testing the delay slot. This is converted to machine code using a modified version of JamesHeart's assembler (https://github.com/JamesHearts/MIPS32_Compiler). More complicated test cases were created in C such as a recursive implementation of fibonacci, where the mipsel-linux-gnu-gcc tool was used to compile and assemble the C code and turn it into MIPS assembly using the correct flags. Every test case was tested for correctness on the Danyqui mips interpreter (https://dannyqiu.me/mips-interpreter/) and

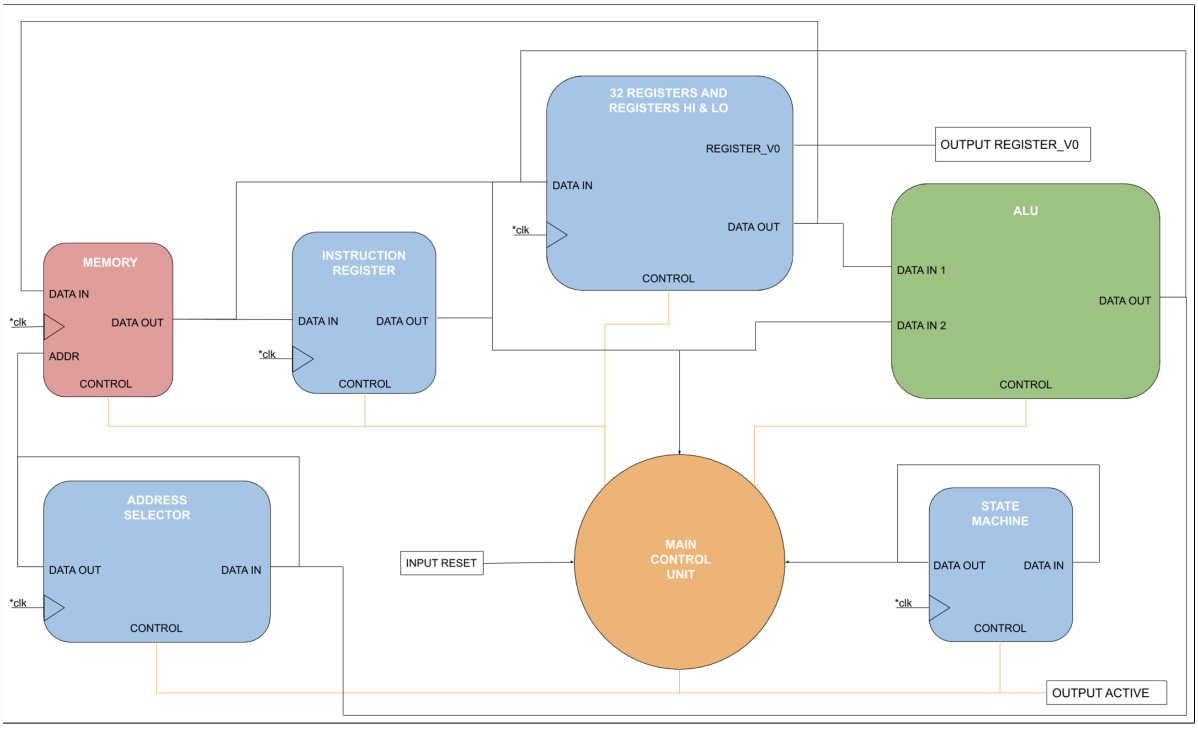


Fig. 1: CPU Architecture

MARS simulator (http://courses.missouristate.edu/kenvollmar/mars/) (for more complex instructions).

Each assembly file has the result stored in register v0 ($2), and has a corresponding reference file with the expected result (to test if it passed or failed). The hex files are converted into byte files with a C++ program, which separates each word into bytes from least significant to most significant byte (to be read with the $readmemh function in the bus memory).

Bus Memory

The bus memory verilog implements a mapped memory with a shortened bus width of 11 bits (as the full 32 bits would take too long to test all test cases). This provides a new 512 word memory (which is the minimum instruction space to fit the longest test case - fibonacci). Addresses from the CPU that are greater than or equal to 0xBFC00000 are mapped down to a new reset vector at word 128 (byte address 0x00000200). Data addresses from test cases occupy the words below 128, so they do not need to be mapped to a different address.

The memory has a variable waitrequest so that each read and write takes a random number of cycles from 0 to 13 to have the read result at the output or be ready to be written to. This functionality tests that the CPU can handle any memory with any number of wait request cycles (including zero in a perfect RAM).

Byte enable is implemented by storing the memory as an array of bytes and reading or writing a byte only when the corresponding bit in the byte enable signal is asserted. There are three states in the memory: IDLE (when nothing is being written or read), BUSY (there has been a write or read assertion and the waitrequest signal needs to be held high), and CHILL (when the number of stall cycles has passed, wait request is de-asserted, and it needs to output the read data or write to memory).

Byte enable is implemented by storing the memory as an array of bytes and reading or writing a byte only when the corresponding bit in the byte enable signal is asserted.

Testbench

The verilog file instantiates the CPU and the bus memory with the testcase byte file and a random number of stalls for each cycle. It also generates the clock cycles, and finishes running if the active signal is de-asserted or the number of cycles exceeds 100,000. At the end, it outputs the value of register v0 to be compared with the reference files for each test case.

4. Diagram of Testbench

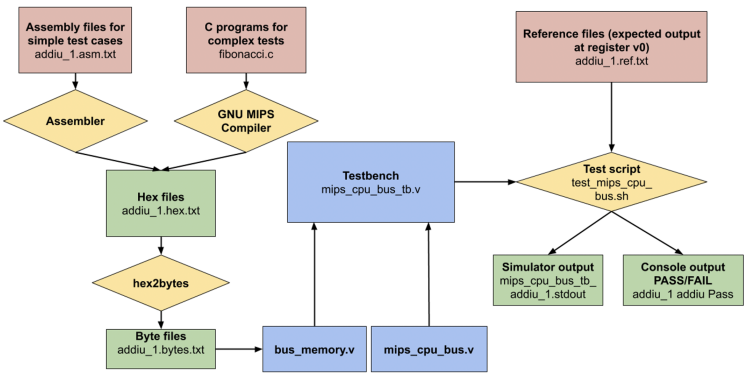
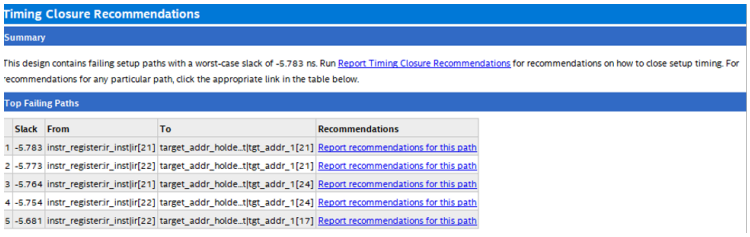


Fig. 2: TestBench Flowchart

5. Area and Time Summary with Quartus

Time Summary	Fmax = 7.83 MHz
Critical Path	Slack = -27.637, from Instruction Register to Register HI
Area Summary	Total Logic Elements: 9,447 / 15,408 (61%) – Combinational only: 8265 – Register only: 207 – Combinational with register: 975



Timing Closure Recommendations

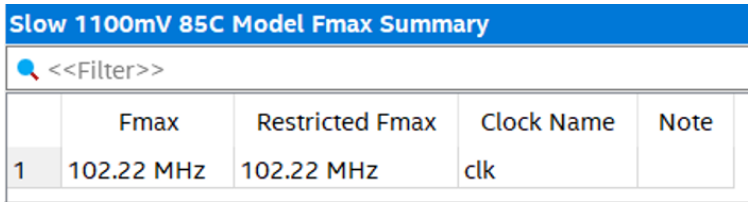
Summary

This design contains failing setup paths with a worst-case slack of -5.783 ns. Run Report Timing Closure Recommendations for recommendations on how to close setup timing. For recommendations for any particular path, click the appropriate link in the table below.

Top Failing Paths

	Slack	From	To	Recommendations
1	-5.783	instr_registerir_inst|ir[21]	target_addr_holde..t|tgt_addr_1[21]	Report recommendations for this path
2	-5.773	instr_registerir_inst|ir[22]	target_addr_holde..t|tgt_addr_1[21]	Report recommendations for this path
3	-5.764	instr_registerir_inst|ir[21]	target_addr_holde..t|tgt_addr_1[24]	Report recommendations for this path
4	-5.754	instr_registerir_inst|ir[22]	target_addr_holde..t|tgt_addr_1[24]	Report recommendations for this path
5	-5.681	instr_registerir_inst|ir[22]	target_addr_holde..t|tgt_addr_1[17]	Report recommendations for this path

Fig. 3: Top Failing Paths[1][2]

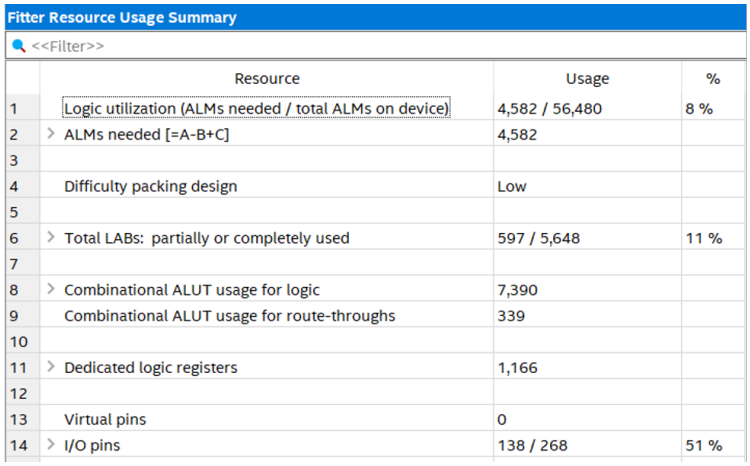


Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	102.22 MHz	102.22 MHz	clk	

Fig. 4: Slow 1100mV, 85^{o}C, Model Fmax Summary



Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	4,582 / 56,480	8 %
2	ALMs needed [=A-B+C]	4,582	
3			
4	Difficulty packing design	Low	
5			
6	Total LABs: partially or completely used	597 / 5,648	11 %
7			
8	Combinational ALUT usage for logic	7,390	
9	Combinational ALUT usage for route-throughs	339	
10			
11	Dedicated logic registers	1,166	
12			
13	Virtual pins	0	
14	I/O pins	138 / 268	51 %

Fig. 5: Resource Usage Summary

References

[1] D. Thomas, *Final Year 1 Project - CPU Design*. https://bb.imperial.ac.uk/bbcswebdavpid-1768920-dt-content-rid-6361823_1courses13390.201910Evaluating%20digital%20designs.pdf, 2020.05.15.

[2] E. Stott, *Evaluating Digital Designs*. https:web.microsoftstream.comvideod6466e6c-7b3d-41d8-8b12-bb5d7a6e8520?referrer=https:%2F%2Fbb.imperial.ac.uk%2F, 2020.06.02.