MIPS CPU Data Sheet

I. INTRODUCTION

The CPU designed is a 32 – bit synthesizable Avalon bus interfaced MIPS-compatible CPU, able to implement a subset of the MIPS ISA.

II. DESIGN

Design Structure

The CPU is made up of basic sections: State machine (there are 4 states), Program Counter(PC), Control Blocks, Instruction Register(IR), ALU, Register File, Branch Block, Bus-Interface Unit, various smaller units which perform functions, and a large unit which combines the smaller, individual sections.

The memory is not a part of the CPU itself. The CPU uses an Avalon Bus interface to communicate with the external memory.

Design decisions - States

The state machine was designed to include five different states: Halt, Fetch, Decode, Exec1, Exec2. Once the CPU begins operating, it is initialized to begin in the halt state, until the CPU is reset. From then, the CPU changes state each clock cycle in the order: Fetch, Decode, Exec1, Exec2, then repeating back to Fetch.

The decision to make each instruction last four cycles was made for various reasons. Firstly, it ensures that there are enough separate cycles for different memory access occasions (in load and store instructions, memory is accessed twice by the CPU, once to fetch the current instruction code and once to either read data or write data to the memory). The instruction register and certain registers in the register file also need to be updated during an instruction. Secondly, a four-cycle CPU allows for plenty of potential to improve the CPU by pipelining it in the future. This would massively increase the Instructions Per Second of the CPU, by increasing the Instructions Per Cycle massively, with a small increase in latency,

Below is a table showing what happens at each cycle (particularly, when memory is accessed and when registers update) in load instructions, store instructions and other instructions:

	Posedge(Fetch)	**Posedge (Decode)**	**Posedge(Exec1)**	**Posedge(Exec2)**
Load	Registers in the register file update if necessary. CPU requests instruction code from memory.	Memory outputs instruction code as an input to the IR.	IR outputs current instruction. CPU requests to read data from memory via bus interface..	PC updates. CPU receives data needed from memory.
Store	Registers in the register file update if necessary. CPU requests instruction code from memory.	Memory outputs instruction code as an input to the IR.	IR outputs current instruction. CPU requests to write data to memory via bus interface.	PC updates. CPU has completed writing data to memory.
Normal	Registers in the register file update if necessary. CPU requests instruction code from memory.	Memory outputs instruction code as an input to the IR.	IR outputs current instruction ALU conducts arithmetic operations, with the result going to the register file.	PC updates.

Summary of each block

Bus-interface *(mips_cpu_bus_bus.v)***:** The bus interface handles all memory transactions, ensuring that the CPU reads and writes from memory in compliance with the Intel Avalon standard. The bus ensures that data remains consistent when sharing between a big-endian CPU and a RAM that could be little-endian or big-endian.

Control blocks *(mips_cpu_bus_control_logic.v, mips_bus_cpu_alu_control.v)*: The main control block takes the current instruction code from the instruction register and classifies it, outputting logic values which activate and provide data to other blocks based the current instruction. The ALU control unit uses data from the control block to further provide the ALU information about what operation the ALU must perform in the current instruction.

Instruction register *(mips_cpu_bus_instruction_register.v)*: Stores the current instruction code which the CPU receives from memory via the bus interface. This value is stored until the beginning of the next instruction

Register file *(mips_cpu_bus_register_file.v)*: Contains 32 registers, 31 of which can have variable values, and Register-0 being assigned to always be equal to 0. These registers can be updated and accessed during instructions.

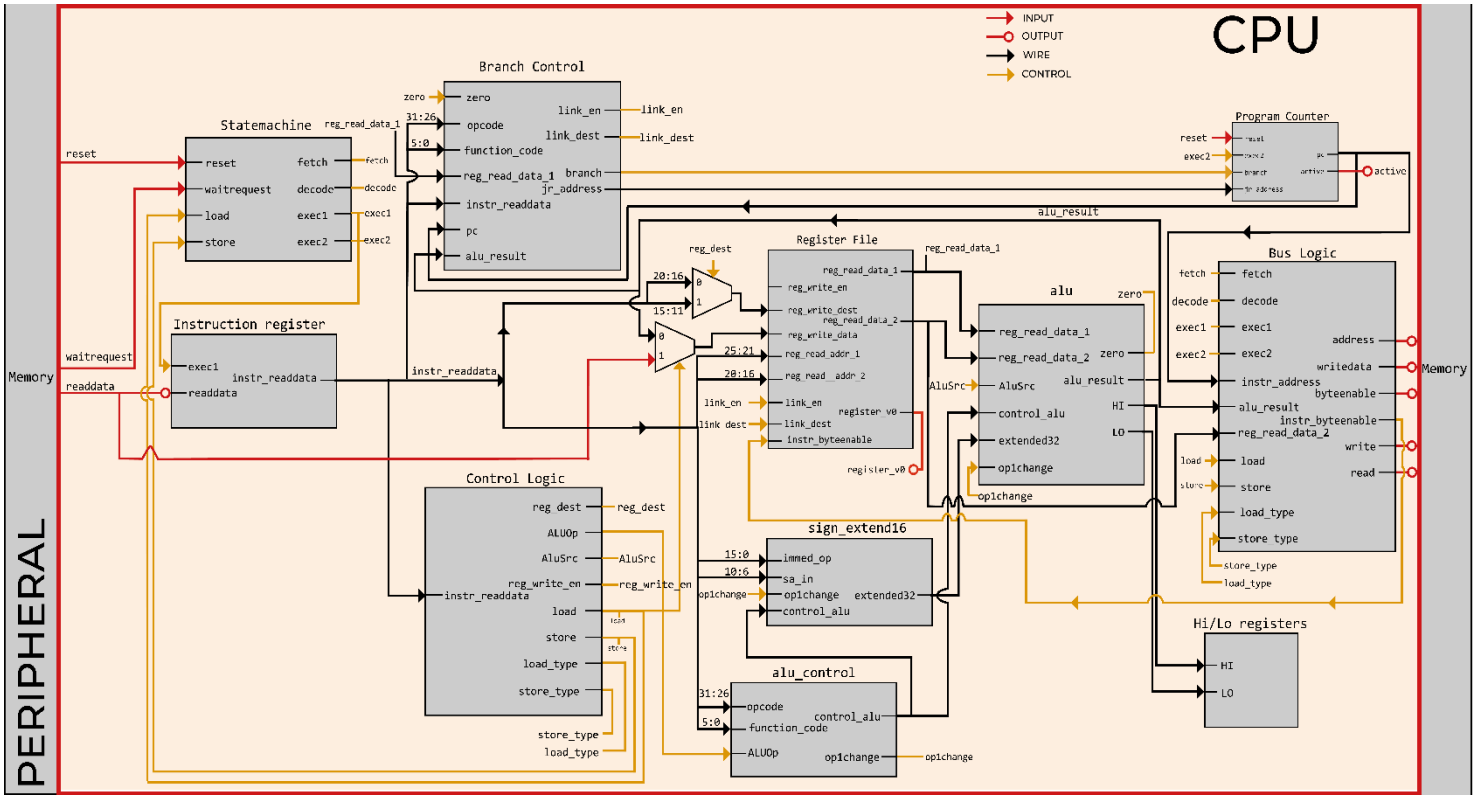
ALU *(mips_cpu_bus_alu.v)*: Performs all arithmetic operations(signed, unsigned, bitwise and arithmetic) during all instructions, as well as performs comparisons which are used when executing conditional branch instructions.

Program counter(PC) *(mips_cpu_bus_pc.v)***:** Stores the memory location of the current program, as well as computes the memory location of the next instruction, or takes the memory location of the next instruction from the branch block if the current instruction is a jump instruction. Once a jump occurs, the PC is configured to perform the next immediate instruction before jumping to the new memory location declared by the jump instruction. This is implemented by implementing a full instruction-cycle delay before jumping to a jump instruction in the PC.

Branch block *(mips_cpu_bus_branch.v)*: Computes the memory location of the instruction to be jumped to during a branch instruction and sends this data to the PC, as-well as an active-high logic value which informs the PC if the current instruction is a branch instruction.

State machine *(mips_cpu_bus_statemachine.v)***:** Stores the current state of the CPU. Information about the current state is sent to all sequential blocks which operate based on which state the CPU is in (more detail provided above, in design decisions). The state machine also ensures that the CPU doesn't continue until the next state if the current state requires memory access, and the memory hasn't informed the CPU that the memory transaction for the current cycle is complete.

CPU diagram



III. TESTING APPROACH & TESTBENCH

The flowchart illustrated below describes the testbench strategy. The command line at the start is used to run the testbench, with '*test_mips_cpu_bus.sh'* being the shell script followed by the required argument *'[source_directory]'* which is the relative path of a directory containing a Verilog CPU. The second argument, *'[instruction]'*, is an optional string argument specifying the lower-case name of a MIPS instruction to test. If this argument is not specified, the testbench will proceed to test all testcases.

The selection of the assembly testcase files is achieved with the help of bash script wildcards and for loops, by iterating over files with the pattern *'test/testcases/*.s'* in the case where the *'[instruction]'* argument is not specified, or iterating over files with the pattern *"test/testcases/[instruction]-*.s"* where the *'[instruction]*' argument is specified. The selected files are then advanced successively into the next testbench stages. In order to make this testcase file selection method work, the testcase files within the *'test/testcases'* directory are named in the format *'[instruction]-case1/2/3/4.s'*, for example, *'addiu-case1.s', 'addiu-case2.s', 'jr-case1.s'*, etc. Since the testcase file names are unique for the testcase while only containing the allowed characters specified in the coursework specification, the Testcase-id is chosen to be the filename of the testcase without the path directory and the *'.s'* extension, which are removed using the '*basename'* command in the bash script file.

With regards to the testbench output, the testbench prints additional lines for each testcase to stderr to facilitate in debugging. One line is printed to stdout for each testcase, and follows the format:

addiu-case1 addiu Pass
addiu-case2 addiu Pass
addiu-case3 addiu Pass

The above stdout output is an example when '*addiu'* is specified in the optional *'[instruction]'* argument, otherwise, if the *'[instruction]'* argument is left blank, all testcases would be executed in the testbench. Due to the absence of the instructions *mfhi* and *mflo,* the instructions that modify *HI* and *LO* were instead tested by executing an *addiu* instruction that modifies register *$v0* after such instructions to at least determine the presence of a compilation or runtime error in those instructions, if any.

The testbench uses four python files: *assemble.py*, *mips.py*, *register.py* and *simulator.py*. *assemble.py* converts the testcase assembly language into binary machine code. After writing them into the specified text file, it is then fed into the CPU memory. *mips.py* is used in *assemble.py* to read and interpret testcase files. *register.py* links the name of the registers to the compiler registers in the MIPS. *simulator.py* simulates the testcases and produces an expected register *$v0* result of them.

The files *assemble.py* and *simluator.py* both use the *re* library, which stands for 'regular expression library'. It is used to recognize the instruction, register file names and immediate values in each line. Within *simulator.py,* we divided the instruction types into different functions. All logical and arithmetic operations were handled by the *logical_arithmetic()* function, having a dictionary of instructions as keys and corresponding signs as the values. *'mthi'* and *'mtlo'* instructions were handled by *move()* function and similarly for *set_less_than(), load_store(), branch()* and *jump()* functions.

The testbench is dependent on the final value stored in register *$v0* for both the simulator reference output and the Verilog CPU output. Therefore, the final value in register *$v0* from the Verilog testbench module is written to a text file *'test/cpu_output/tb_output.txt'* using the *'$fwrite'* command in *mips_cpu_bus_tb.v*. This is then compared to the reference register *$v0* value produced by the python simulator stored in the 'test/reference' directory, and we extract the register *$v0* value from the simulator output by using the *'grep'* command in the bash script to look for the pattern 'reg_v0 is:' and then use *'sed'* to delete the leading string and save just the register *$v0* value into a new file within the 'test/reference' directory. The final register *$v0* values from the CPU and the simulator are then compared and if they are the equal then the testcase is passed, else it fails.

IV. Cyclone IV E 'Auto': Area & Timing Summary

Flow Status:	Successful – Tuesday December 14th 16:18:39 2021		
Quartus Prime Version:	21.1.0 Build 842 10/21/2021 SJ Lite Edition		
Revision Name:	quartusSV		
Top-level Entity Name:	mips_cpu_bus		
Family:	Cyclone IV E		
Total logic elements:	708 / 6,272 (11 %)		
Total registers:	227		
Total pins	138 / 180 (77 %)		
Total virtual pins	0		
Total memory bits	1,856 / 276,480 (< 1 %)		
Embedded Multiplier 9-bit elements:	0 / 30 (0 %)	Fmax:	234.3MHz
Total PLLs:	0 / 2 (0 %)	Restricted Fmax:	234.3MHz
Device:	EP4CE6F17C6	Worst Case Set-up Slack:	-10.425 ns
Timing Models:	Final	Clock Name:	clk