MIPS 32 BIT CPU Team-0

Contents:

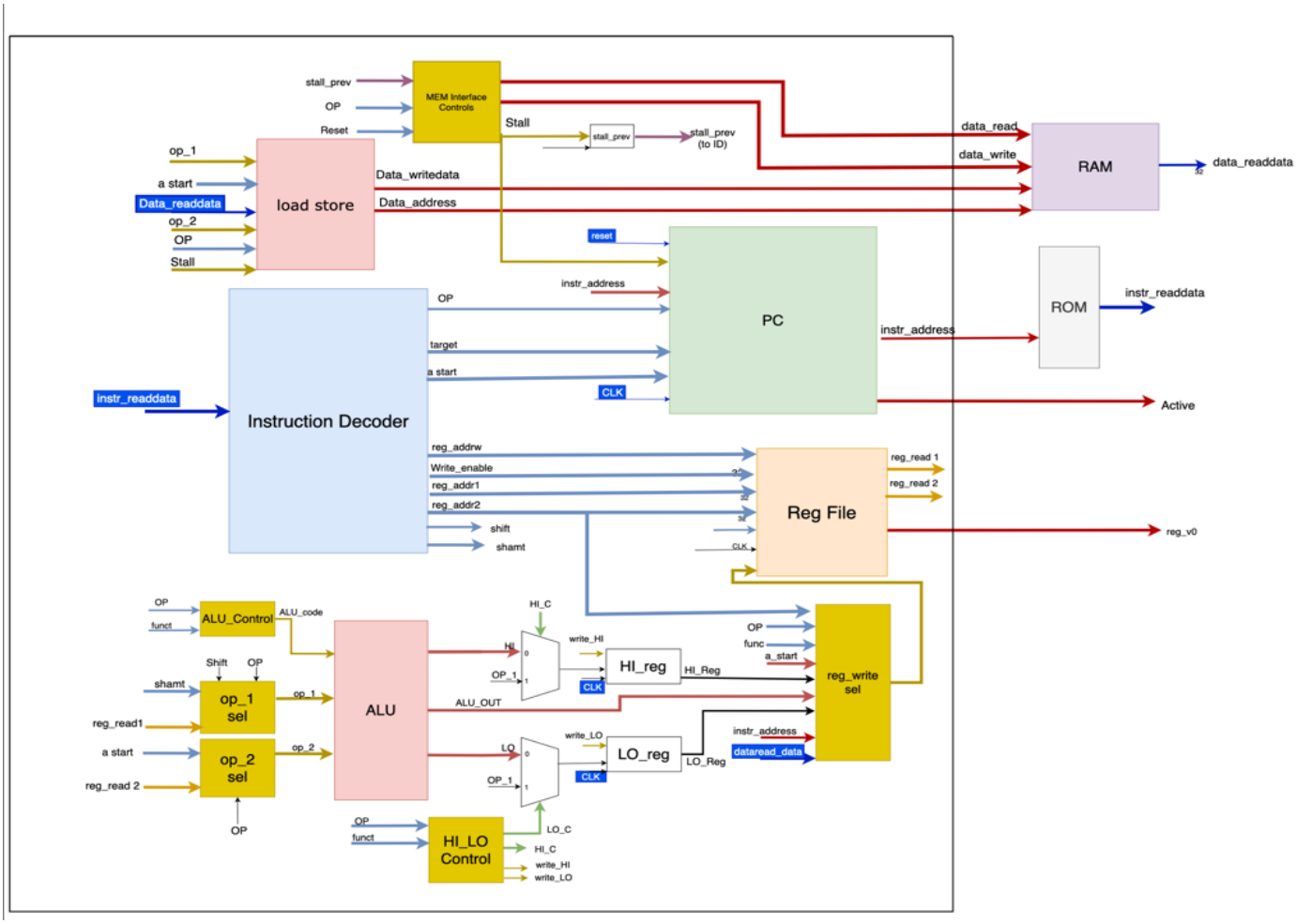
Area and timing analysis:

The percentages (%) indicate similarity to the reference device, EP4CE15F23C6.

These are currently the best possible results that we can achieve with given client provided information. More refined values can be obtained once more detailed platform timing is available.

**Maximum Clock Frequency at given temperatures and operation Voltages.*

FPGA Characteristics: (Using Intel Quartus)	
1200mV 85C Fmax*	89.15MHz
1200mV 0C Fmax*	99.53MHz
Total Logic elements	9160/15408 (59%)
Total Registers	1155
Total Pins	198/344 (58%)
Total PLLs	0/4



Architecture Overview

1. Architecture Overview:

The 32-bit MIPS compliant CPU has a Harvard-type interface as it relies on separate memory units for instructions and data; they are therefore accessed in parallel via distinct buses.
The CPU works in a linear and transparent way by performing most operations within one cycle, due to the combinatorial access to memory. The only instructions that pose an exception to the overall "1-instruction-1-cycle" format are store instructions that involve bytes, half words and, in a way, Jumps, which pose an exception to linearity in order to ensure compliance to the ISA (see the section on branch delay slot).
Both the buses that bridge the CPU with the instruction memory and the RAM maintain the same byte-to address mapping. Namely, the most significant byte of the word, being data or instructions, is held in bits [7:0] of any interface bus and is going to be written at mem[address+0]. This requires an inversion of endianness when words start to get processed inside the CPU before they are outputted.

a) Load/Store block

The first step of the CPU's interaction with the system it is being run on, occurs in the Load/Store block. Incoming words are put in the appropriate big-endian format and forwarded to the Instruction Decoder for further processing. As memory transactions only occur via words, the LS block is extremely important when it comes to *Store Byte* and *Store Half* instructions. To avoid overwriting the current memory content by storing single bytes using full word transactions, the LS block has the functionality to stall the Program Counter when needed. This allows for the reading of the word at the aligned address containing the required location and the insertion of the byte (or bytes) that need to be stored within the word and writing of the full word back at the aligned and correct address in the next cycle.

b) Instruction Decoder:

Once the CPU receives the instruction word, the Instruction Decoder unpacks the information needed by the rest of the CPU to perform the required operations. As a well thought design choice, the block performs more of a sorting role rather than an interpretive one as it mainly splits the instruction words into significative components and allows each specialised block to interpret them accordingly rather than extrapolating it at a higher level. This allows for controlling and operative sections to be logically brought together, thereby reducing any chances of errors by encapsulating a faulty CPU area. Furthermore, it makes the overall architecture less error-prone since associating control, operation logic and spotting dependencies become more immediate. If the instruction holds an OP that is not recognised by the CPU, the default operation will be performed (ie. the output of the ALU, depending on the contents of the instruction word will be chosen to be ***reg_file*** but the ***write_enable*** will be 0 so nothing will happen effectively.

c) Registers:

The CPU relies on 32 32-bit registers to rapidly execute programs. Some of these registers serve specific purposes such as $0 which has the value 0; $31 which contains the address of the process caller to which the PC returns to once a subroutine is complete; and v0 ($2) which is the output when the CPU halts. The contents of the destination register are determined based on the instruction executed in the ***reg_write*** selection block. This collects all possible inputs to the register file (including the ALU output, the return address for a linked jump/branch or data from memory) and appropriately outputs a value using a specific portion of the instruction representing the operation required.

d) ALU Block

In order to implement basic arithmetic and logic operations, the CPU is equipped with an ALU Block. An ***ALU_control*** signal derived from the current instruction word determines what operation is needed. In order to maintain the ***ALU_control*** as small as possible, the number of possible signals has been reduced and as such it was decided to pass the shift amount quantity as operator 1, instead of passing it as a separate input.

e) Multiplier and Divider

The multiplier and divider differ from the other arithmetic operations due to the fact that 32 bits are potentially not enough to store the result of the operations of two 32-bit operands. Therefore, we need an alternative way to store it. These instructions are still implemented through the ALU block but the result of the given operation is put into a special output of 64-bit width: ***divmult_out***. This is subdivided into two, and the results of the operations can be accessed through *Move From Lo* and *Move From Hi.* These instructions access the HI and LO registers where the results are stored. For multiplication, ***LO_reg*** and ***HI_reg*** can be seen as extensions of each other, with LO containing the least significant bytes and HI containing the most significant bytes of the result.
For division operations, the *MFLO* instruction moves the lower bits, which is the result of the integer division operation, whereas the *MFHI* instruction accesses the remainder of this operation.
It is important to note that these instructions are more complex due to the magnitude of the operations that they rely on to be performed compared to simpler operations, so they significantly affect the critical path of the CPU and as such, the maximum clock frequency.

f) Program Counter

This register controls the flow of program execution and provides the ROM with the address from which the instruction to perform is fetched. Normally it increments by four at the end of each cycle (to accommodate the fact that information is stored as bytes inside the RAM), with a few exceptions. This happens either when a "*partial Store*" instruction is happening where, as seen above, the PC needs to stall for one cycle or when the instruction is a *Branch/Jump.* In this case, given that according to the MIPS ISA, the instruction after the jump (in the so-called branch delay slot) always must be executed, the next address is going to be evaluated differently. As a design decision, the CPU elaborates the destination address in the cycle in which the jump/branch instruction is detected. This helps avoid having to cache information on the operands that had to be compared in order to determine whether conditions are met going into the next cycle in the case of branches.
If the conditions are met, a signal jump is asserted. At the end of the cycle this information, as well as the destination address are stored in flip-flops and the PC is updated after the instruction in the branch delay slot is executed.
This mechanism is carefully adapted to function when the instruction in the delay slot is a partial store, a quite delicate case as the ***jump_now*** signal will be overridden by the stall and all information on the Jump needs to be maintained for one more cycle.
When the CPU attempts to execute the instruction at address 0 it will Halt un-asserting the active signal and thus signalling that v0 is ready to be collected.

2. Testing Strategy

a) Module Testing:

Before integrating them with the rest of the CPU, the LS and ALU blocks were tested with the instructions of their competence to ensure basic functionality for the ALU and formal correctness of the output for the LS, as that was the most delicate aspect of the block.

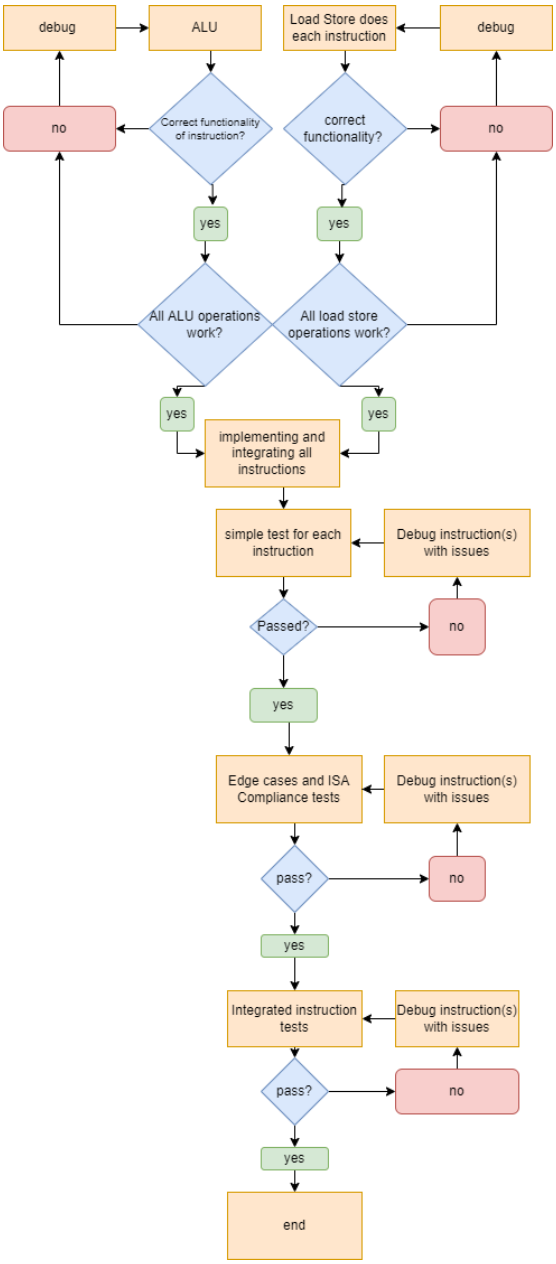
b) Initial integration:

The next step of testing was to integrate the two blocks with the rest of an (incomplete) CPU in order to implement a subset of working instructions. These were also tested to ensure the basic functionality of the integration between operative blocks and control signals.

c) Full CPU Debugging Process:

The debugging process was a four-stage intensive and thorough process with feedback at each level:

Stage 1: Each instruction was tested with complex test cases, involving randomised operator values, and we attempted to always include the smallest number of external instructions and combinations to help narrow down what the faulty instruction was. This methodology was aimed to ensure functional correctness in order to process more complicated tests in the future. With these, the output value can be directly related to the instruction performed (ie the result of an operation in case of logic/arithmetic instructions or the immediate consequence of a change in PC).



Stage 2: The next step was to test more simple instruction via instruction tests, however with the critical and highest priority criteria of testing the edge and boundary cases of the CPU. This was to ensure basic functionality and correct behaviour with respect to concepts such as signedness.

Stage 3: Once specific functionality was proven, test cases were designed to check ISA adherence were run. These targeted aspects directly related to the MIPS requirements with regards to each instruction and mainly concerned:

- The correct choice between sign-extension or zero-extension of operands during I-type instructions, including *Branches* (for which negative offsets were thoroughly tested) and the misnomer instruction ADDIU.
- Correctness of destination address during jumps: due to how the elaboration of the destination address was designed it was needed to ensure that the address was elaborated with respect to that of the branch delay slot. A particularly tricky corner case is the case of J-type where the 4 MSB of the branch delay slot address are different from those of where the jump instruction is (as the first ones need to be used to form the destination address in order to meet ISA specifications).
- Correct endianness and interpretation of words coming from memory.

Stage 4: The last step of this intensive debugging process consisted of providing the CPU with complicated programs it was designed to withstand involving complex instructions within loops which pushed the CPU to its limits. In these cases, the output is indirectly related to the instructions tested (for example it's the counter of a loop being performed).

References:

2021, "Difference between Von Neumann and Harvard architecture". https://www.geeksforgeeks.org/difference-between-von-neumann-and-harvard-architecture/

Charles Price, 1995, "MIPS ISA". https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf

David Thomas, ele50010-2021-verilog-lab. https://github.com/m8pple/elec50010-2021-verilog-lab