MIPS 32-bit CPU & Testbench Coursework

Team 20: Alexandra Neagu, Chenghong Ren, Tanglitong Zhang, TszHang Wong, Zack Reeves

CPU Design and Architecture

Overview

The overall architecture follows a synthesisable MIPS-compatible CPU design, which interfaces with the world using a single Avalon compatible memory-mapped bus, that gives it access to memory and other peripherals. The CPU issues read and write transactions, including instructions and data over the same memory interface. The focus of our design is to implement a functional Big-Endian CPU with the highest area efficiency possible.

Design Decisions

The data path shown in *Figure 1* can easily offer an idea of the CPU's major computational blocks implemented and their interconnection of I/O ports. Scalability, maintenance, regularity, and area efficiency are the main factors we considered during the design process. As "Simplicity favours regularity" ***[1],*** our design is constructed to be open to future developments and can be easily debugged and maintained. The following section presents the primary design decisions taken to achieve those factors.

Firstly, the implemented MIPS I instructions ***[2] c***an be classified into four different types, each performing distinct procedures according to the CPU's state. Considering this, we decided to create a singular control block that sets the behavioural signals of each specific instruction type to the block components. This highly structured and centralized design allows for easy future upgrading of implementation onto MIPS II. A more in-depth discussion of this decision can be found in a lower section.

Secondly, our design approach pursues a methodology of 5-CPI (cycles per instruction). Alongside implementing additional intermediate registers, this approach presents the opportunity for the CPU to be adapted into a pipelined architecture in the future. Similarly, the waitrequest input port enables the CPU to support all kinds of memory-mapped devices. Therefore, those two characteristics illustrate the high degree of flexibility our CPU design has for real-world integration.

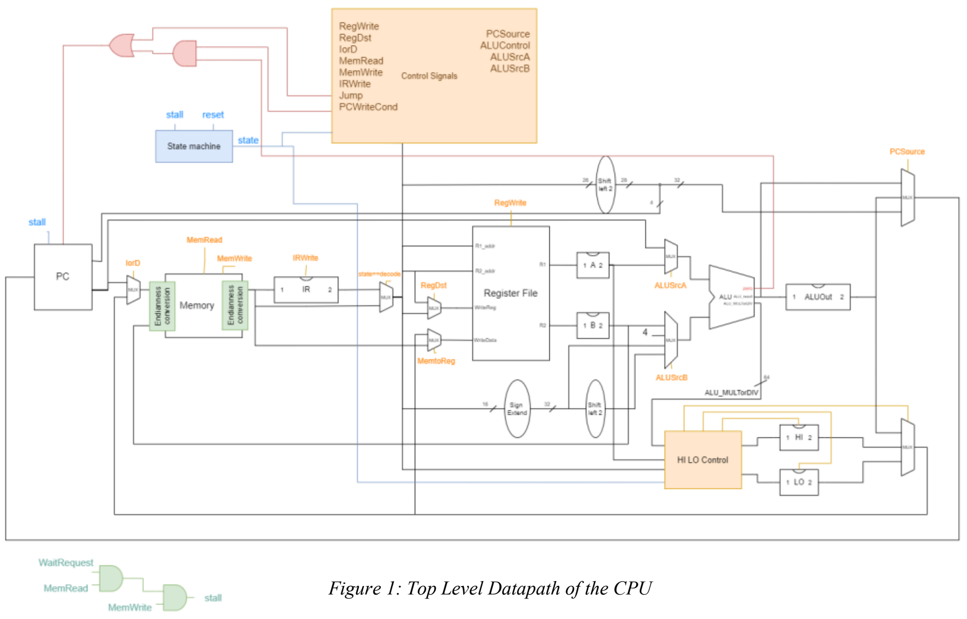

Figure 1: Top Level Datapath of the CPU

Thirdly, we assumed that the compiler always optimises the time efficiency, so all arithmetic operations are implemented only once in the ALU module. Hence the ALU would be used multiple times to operate an instruction. For instance, a branch-and-link instruction uses the ALU for calculating the target address, return address, and condition.

Additionally, taking the branch delay slot into account, two special registers are added to the PC block in contrast to the standard MIPS CPU. The $branch$ register indicates if the previous instruction is a Jump/Branch instruction and the $PC\_Branch\_Address$ register holds the branch target address.

Instruction classification

As mentioned in the first point, we categorize the instructions into four types according to their repeated usages of some CPU blocks: Arithmetic, Jump, Conditional Branch, and Memory Referencing instructions.

Their behaviour is distributed over the five cycles implemented: FETCH, DECODE, EXECUTION, MEMORY_ACCESS, WRITE_BACK. For the first two states, all instructions behave as follows. During FETCH, instructions are inputted in the CPU, stored into the $\text{instruction register (IR)}$, and the $\text{program counter (PC)}$ gets incremented by four. During DECODE, values from two registers from the register file are prepared, and the return address used by link instructions is computed. The CPU always has the next address ($PC + 4$) or return address ($PC + 8$) prepared early in the FSM period, as these values might be used later, which gives rise to a general speed up, and the CPU doesn't receive any delay penalty.

For the remaining three states, operations vary depending on the type of instructions.

Arithmetic-type Instructions

These instructions are associated with values stored in registers or immediate values included in the instruction words. During EXECUTION, the corresponding operation is applied to the received inputs within the ALU. The result is then written into the destination register during MEMORY ACCESS.

Jump-type Instructions

Unlike other instructions which adapt the value of $PC + 4$, these instructions allow the PC to be rewritten by a specific value. During EXECUTION, the target address is computed or obtained from the immediate operand. This would then be passed to $PC\_Branch\_Address$ register in MEMORY ACCESS to achieve the jump after the execution of the delay branch instruction.

Conditional Branch Instructions

This type of instruction is similar to the Jump-type, but it only rewrites PC if its specific condition is met. Take instruction BGEZ as an example (*Figure 2*). During EXECUTION, the target address is computed. The branch condition is then checked during MEMORY ACCESS to determine whether the computed target address should write into the $PC\_Branch\_Address$ register. Thus, deciding to complete the branch or not.

If Jump or Conditional Branch Instructions followed with a link, then during EXECUTE, the CPU would store the precomputed return address (from DECODE stage) into register $31 to return after exiting a subroutine.

Memory Reference Instructions

These instructions involve the reference and modifications of data in the memory. During EXECUTION, the target address is computed. In the following states, the data in that address would either be read out or rewritten by the given data depending on whether it is a store or load instruction. For load instructions, the value read from the memory unit is then loaded into the corresponding register during WRITE BACK.

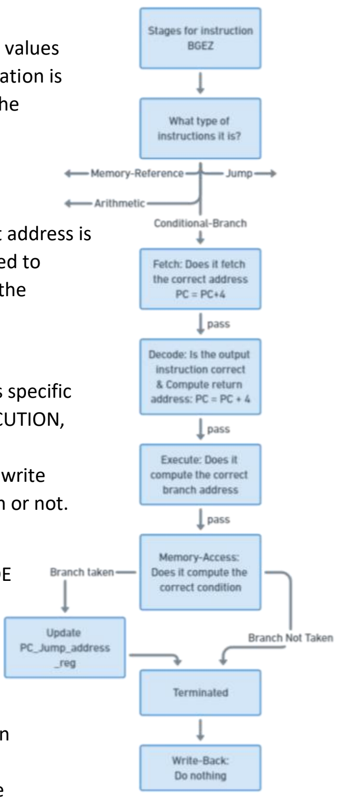
Stages for instruction BGEZ
What type of instructions it is?
Memory-Reference
Jump
Arithmetic
Conditional-Branch
Fetch: Does it fetch the correct address PC = PC+4
pass
Decode: Is the output instruction correct & Compute return address: PC = PC + 4
pass
Execute: Does it compute the correct branch address
pass
Memory-Access: Does it compute the correct condition
Branch taken
Update PC_Jump_address _reg
Branch Not Taken
Terminated
Write-Back: Do nothing

Figure 1 Step by step visualisation of the 5 cycles, BGEZ

CPU Performance

Our CPU has been tested with various sized test cases. It allowed us to notice the advantages and issues of our design. Firstly, the CPU has a fixed 5-cycle FSM as it is a general-purpose CPU and is expected to execute a mixture of memory-reference (5 cycles) and arithmetic (4 cycles) instructions. If most instructions were arithmetic, then a volatile FSM would be more efficient because it would skip unnecessary cycles for specific short instructions. However, in our case, the cost of frequently changing the total number of stages according to the instruction may overwhelm the benefit it brings.

Additionally, we have simulated our CPU using Intel's Quartus Prime software which allowed us to evaluate a timing and area analysis. We have chosen the FPGA device with the most similar specification, Cyclone IV E, to simulate the closest performance to real-world implementation of our CPU. The results of the achieved timing analysis show that the simulated device had a conservative lower-bound state machine frequency of $3.92\mathrm{MHz}$, which was done under worse-case conditions of $1200mV$ and $85°\mathrm{C}$. This result could highly be affected by the critical path which requires $127ns$ to complete. This longest delay path pointed out by Quartus can also be seen in $Figure\ 1$ as staring at the output of $memory$ and ending at input $ALU\_MULTorDIV$ of registers HI, LO. So, it signifies the path of the $div(u)$ or $mult(u)$ instructions computed combinatorically. This leaves time-related decisions to the compiler; hence, we don't have control over its time efficiency. A possible improvement could be dividing the 32-bit words into smaller variables and applying the operation sequentially. Thus, allowing us a degree of control over the computation speed.

This simulation proves that our overall focus was the area efficiency, as the project utilises $9{,}180$ logic elements out of $15{,}408$ on the FPGA. We have tried performing the simulation onto other modules and sized devices, and we noticed that we are achieving the best result for this model of FPGA. Fact which might be due to high compatibility between the CPU's and the device's specifications for logic elements and ports.

CPU Testing and Testbench

To assert the correctness and clear out all the potential hazards of our CPU, we created two testing methods.

1. Testbench

To ensure each instruction performs correctly, we have created a testbench that reviews the output of different test programmes. The tests cover both general cases for the instructions and edge cases that have a higher chance of getting the CPU to output the wrong value.

Our testbench design expects all test cases must have the last executed value stored in register $\$v0$ (register of index 2) for an accurate check of the instructions called. Also, the programs must include a jump instruction which will set the PC to address 0, hence signalling the ending of the program and setting the CPU's $active$ signal to low. Therefore, the checking process is completed by comparing the value of register $\$v0$ to an expected reference point for the specific test case.

Besides overviewing the transfers between the CPU and the Memory, the testbench also sets the clock, reset flag and time duration restrictions. Hence, the testbench observes whether the CPU fails to complete a test program in the time required ($10{,}000$ cycles) and halts the process resulting in aborting the test case and will send out an error code.

So, as shown in $Figure\ 3$, following the check of a successful compilation, the comparison between the given and expected output can lead to one of the following situations:

- The test program fails due to the CPU file not finishing the program in the expected amount of time
- The test program fails due to a different output result than the expected value
- The test program successfully passes

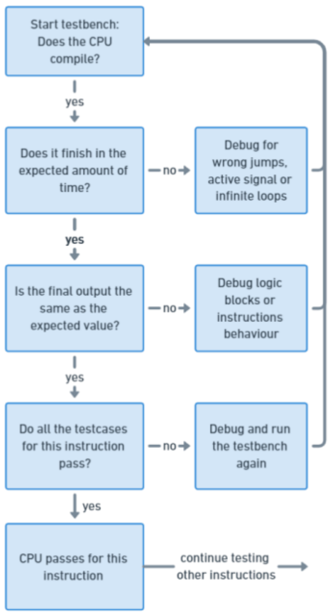
Start testbench: Does the CPU compile?
yes
Does it finish in the expected amount of time?
no
Debug for wrong jumps, active signal or infinite loops
yes
Is the final output the same as the expected value?
no
Debug logic blocks or instructions behaviour
yes
Do all the testcases for this instruction pass?
no
Debug and run the testbench again
yes
CPU passes for this instruction
continue testing other instructions

Figure 2 Testing procedure

Another architectural choice made is our approach for building a memory of size 2^{18} $bytes$ that signifies 2^{16} $word\ addresses$. To avoid address clashes, the memory is divided into two halves: the first half between addresses 0 and $BFBFFFFF$ is assigned to data, and the second half from address $BFC00000$ onwards is designated for instructions. We have taken this approach to respect the load/store instructions' 16-bit immediate which signifies the data address that will be used by the CPU.

We also decided to implement the memory interface to be human-readable so that the data files have 4 bytes per line which signify the expected big-endian formatting of the instructions as used by the CPU (after endianness conversion). By implementing this, the memory interface converts the reader-friendly file to a memory respecting the small endianness requirements. While not affecting the 32-bit inputs and outputs of the CPU, these implementations offer the opportunity of easily enlarging the testbench and debugging the errors of the CPU.

Commented [王1]: We also decided to implement the memory interface to be human-readable by having four bytes per line in the data files, which signify the expected big-endian formatting of the instructions, as used by the CPU (after endianness conversion).

Commented [王2]: allow easy enlargement of the testbench and debugging of the errors of the CPU to happen.

Additionally, the implementation of randomising the value of an active high waitrequest signal creates a more real world-like behaviour of the memory interface. This further checks for the versatility of the tested CPU.

Finally, the $test\_mips\_cpu\_bus.sh$ script merges the testbench files by calling all the necessary commands for compilation, execution, and comparison of the output files. The script is run with the below formats:

$test\_mips\_cpu\_bus.sh$	$directory\_path\_of\_cpu$	$specific\_instruction\_to\_test$	Tests all testcases available for a specific instruction
$test\_mips\_cpu\_bus.sh$	$directory\_path\_of\_cpu$		Test all testcases for all instructions
$test\_mips\_cpu\_bus.sh$	$help$		Prints the available testcases

2. Complex test cases

To ensure that the CPU could perform correctly even in a complex scenario, we crafted a carefully designed test case that consists of a mixture of all instruction types, and more importantly, complex branching logics (including a subroutine and consecutive jump). Some instructions used include LW, LB, LH, SW, ADDU, ADDIU, DIVU, MFHI, ANDI, BEQ, JAL, JR.

Throughout building our CPU, we have created multiple similar test cases. They tested a group of similar instructions, such as a sequential shifting test case, an edge-case test for multiplication and division instructions, a loop conditional branching program, etc.

These test cases are not in the general testbench file as they could not be classified under one specific instruction.

References

[1] F. K. Gürkaynak and M. Püschel, "MIPS Assembly," 2013. [Online]. Available: https://syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/15_MIPS_Assembly.pdf. [Accessed 16 Dec 2021].

[2] C. Price, "MIPS IV Instruction Set," Sept 1995. [Online]. Available: https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf. [Accessed 20 Nov 2021].

[3] Whimsical Flowcharts, "Diagrams anf Flowcharts," Whimsical, 2021. [Online]. Available: https://whimsical.com/. [Accessed 13 Dec 2021].

[4] D. A. Patterson and J. L. Hennessy, Computer Organisation and Design, Elsevier Inc., 2014.

data small endian	Testcase		0xBFC…
0x00000000	LW $20 0x8($0)	$20 <= 0x0000005A	0x0
0x0000001E	LW $25 0x10($0)	$25 <= 0x0001DD7B	0x4
0x0000005A	ADDU $10 $20 $25	$10 <= 0x0001DDD5	0x8
0xFFFFFC7C	SW $10 0x800($0)	mem[0x800] <= 0x0001DDD5	0xC
0x0001DD7B	LB $11 0x803($0)	$11 <= 0xffffffD5	0x10
0x00000001	DIVU $11 $10	LO, HI <= 0x8927,0x3762	0x14
0x00001388	MFHI $18	$18 <= 0x3762	0x18
0x00010000	ADDIU $21 $zero 0x3762	$21 <= 0x3762	0x1C
0x0001DDD5	ANDI $20 $18 0x3fff	$20 <= 0x3762	0x20
1st jump start	BEQ $20 $21 0x5	branch should be taken	0x24
	No-op	Do nothing	0x28
2nd end, 3rd start	JR $31	Jump back	0x2C
	ADDIU $5 $5 0x1	$5<=0x1389	0x30
	J 0x00000(failed)	Fail STOP	0x34
	ADDIU $2 $5 0x1	Fail $2<=1	0x38
1st end, 2nd start	JAL 0x3F000B	$31 <= 0xBFC00040	0x3C
	LH $5 0x1A($0)	$5 <= 0x1388	0x40
3rd end, end code	JR 0x0 SUCCESS		0x44
	ADDIU $2 $5 0x1	$2 <= 0x138A $5=0x1389	0x48