MIPS CPU Project Report – Team 8

CPU Design and Architecture

The focal point of the design approach used with this CPU was to achieve the highest IPC possible, as all combinatorial logic was modelled without delay in Verilog simulations, meaning that the longer critical path would not affect the clock speed of the CPU in testbench. Furthermore, submitting a pipelined CPU was strongly recommended against, meaning that a maximising the IPC would lead to the most time efficient CPU, albeit higher area. The factor capping efficiency was therefore memory accesses, as these take 1 cycle to complete, leading to the development of a variable 2-3 cycle FSM which would drive the control logic of the CPU. Most instructions would be completed in 2 cycles, using 1 cycle (ignoring memory stalls) to fetch the instruction from memory, and then use a second cycle to fully execute the instruction. Any load instructions, requiring a second memory access, would instead use 3 cycles to run.

Other factors that played a key role in the design process the scalability, maintainability and extendibility of the CPU itself. As such, a central controlling block is used to manage every aspect of the CPU (see fig. 1). Instructions are processed on a case-by-case basis, with instruction identifier definitions stored in a header file, allowing individual instructions to be easily added and removed to the processor, making future MIPS II easily implementable. Any other intermediary logic is also processed by this control unit, leading to a very centralised structure making it much easier to change update logic affect the whole CPU. Explicit ALU and other arithmetic blocks were not used in the architecture of the CPU as using built-in Verilog operators allow from easier legibility, and minimising area was not a relevant factor in the design philosophy of the CPU, so any arithmetic operations are written explicitly and were assumed to be optimised by Icarus Verilog during module synthesis. Defensive programming was also used to provide a level of safety in ensuring that only MIPS I valid instructions are processed by the CPU. Exception handling is not a part of the specification for this coursework, but by having these asserts in place it will ensure the user or client isn't performing any invalid instructions. This makes the CPU more user-friendly and oriented, providing an aspect of reliability and assurance to the client.

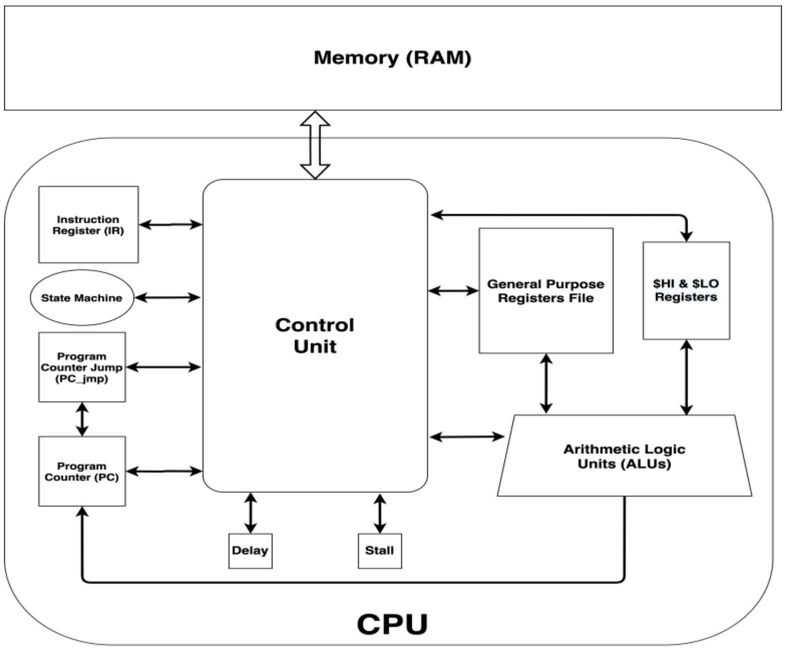
Memory (RAM)
Instruction Register (IR)
State Machine
Program Counter Jump (PC_jmp)
Program Counter (PC)
Control Unit
General Purpose Registers File
$HI & $LO Registers
Arithmetic Logic Units (ALUs)
Delay
Stall
CPU

Fig 1. CPU Architecture

Other than the standard MIPS registers, there are a few separate registers worth noting. *PC_jmp* is used to store the value that the PC will jump to in the delay slot, with the 1-bit flag *delay* indicating whether the current instruction being performed is inside the delay slot of the previous instruction. Another 1-bit flag *stall* is used to indicate whether the CPU has is in a memory stall during execution cycles, meaning it attempted to access memory whilst *waitrequest* was high, which is required to manage the CPU's internal state.

CPU Testing

As per the specification, the CPU testbench aims to target individual instruction to assert the functional correctness of each instruction. In order to accomplish this goal, it was determined that the best approach was to use a generic testbench outputting the final value of $v0 for comparison with a reference output, as this allows for a high number of relatively short, focused tests to be run on a CPU, isolating each instruction.

There are 2 other modules used in the CPU testbench:

- **RAM_32x65536**: This is used as the memory that the CPU interacts with. It contains 2^{16} word addresses (effectively 2^{18} byte addresses) whose addresses are all offset by the CPU's reset vector, giving an effective accessible address range of [0xBFC00000, 0xBFC40000). The block also allows the address 0x00000000 which is the address which causes the CPU to halt. The *waitrequest* signal is set randomly each cycle using the SystemVerilog $urandom_range() method, ensuring the CPU is capable of handling a wide range of *waitrequest* inputs.
- **mips_cpu_bus_tb**: This is the top-level entity that the testbench runs from. It simply creates an instance of the CPU and RAM, links their signals together and driving the clock of both modules at 500MHz, as well as resetting the CPU. Once the CPU halts, the testbench then outputs the contents of $v0, which can be used to assert that the CPU ran as expected. If the CPU fails to halt in 10000 cycles, the testbench will abort and exit with a failure code.

The test script runs the CPU using over 200 pre-assembled hex files, each performing a specific test, which are loaded into the RAM that the CPU interacts with as memory. In order to test a given instruction, there are some dependencies on other instructions, meaning that when performing an overarching test of the CPU, some instructions / individual tests must be run (see fig. 2). Note that it is assumed that **sll $zero $zero 0x0** performs no destructive behaviour and functions as a nop. The script runs the following tests before other instructions unless specified by the optional command-line argument:

1. **Test for jr $zero:** This instruction is used to halt the CPU, effectively allowing successful testcase termination and is required in every testcase.
2. **Test for lui:** The memory used in our testbenches are aligned to the reset vector, meaning that to accessing any part of memory requires and offset of 0xBFC00000.
3. **Test for lw:** Performing any meaningful tests for most MIPS instructions require the ability to load specific values into the ISA-visible MIPS registers.
4. **Test for addiu:** This instruction is used to effectively move the contents of specific ISA-visible registers into $v0 so that their contents can be checked with the reference. This is required for branch / jump instructions which link, as the value stored in $ra needs to be verified. It has also been used in testing other instructions to modify the value of a register without overwriting it.

By running these tests first, the testbench ensures that each instruction can be tested independently, so any particular break in an instruction will be identified by its instruction-specific tests. There are some exceptions as follows:

1. **MTHI and MFHI / MTLO and MFLO:** These 2 pairs of instructions depend upon the other to be tested, as there is no other way to interact with the HI and LO registers in the ISA, and without having direct output lines from these registers there is no way for these instructions to be tested individually.
2. **Multiplication and Division:** These instructions use the separate HI and LO registers to store the output of their operations. Therefore, the **mfhi** and **mflo** instructions are used to move the outputs of these instructions in $v0 to compare with the reference output.

As well as using the testbench in asserting the functional correctness of this CPU, some additional more complex tests were performed on the CPU, but and whilst these are not included as part of the test script as they do not isolate particular instructions, they provide additional confidence that the CPU can also perform more complex programs. Test cases written in C were converted into MIPS assembly and hex using a Makefile. mipsel-linux-gnu-gcc was used as the compiler to convert C into MIPS object files and assembly, which were then mapped into the RAM address space using a linker file, storing instructions at 0xBFC00000 and data at 0xBFC20000, and finally converted into separate data and instruction MIPS binaries. The od command was then used to convert the created binaries into hex files which can be easily imported into the RAM initialisation files used by the testbench. The qemu-mipsel simulator is used to generate a reference output from the binary generated from the C code, where the final returned is the decimal value of $v0 and represents the return value from main(). After creating the appropriate memory and reference files, a separate script can be used on a program-by-program basis to compare the output of the CPU under test and the one produced by qemu.

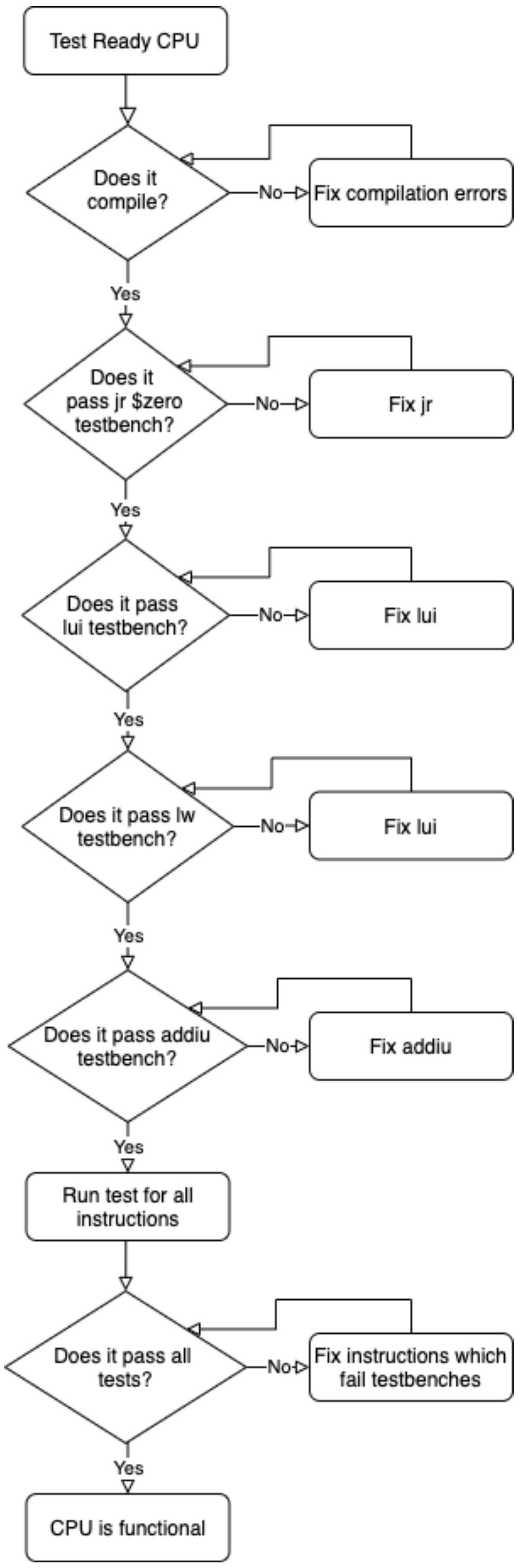
Test Ready CPU
Does it compile?
No
Fix compilation errors
Yes
Does it pass jr $zero testbench?
No
Fix jr
Yes
Does it pass lui testbench?
No
Fix lui
Yes
Does it pass lw testbench?
No
Fix lui
Yes
Does it pass addiu testbench?
No
Fix addiu
Yes
Run test for all instructions
Does it pass all tests?
No
Fix instructions which fail testbenches
Yes
CPU is functional

Fig 2. Test-flow Diagram

CPU Timing and Area Summary

A timing and area analysis was conducted using Intel Quartus Prime by configuring the CPU design onto an Intel Cyclone IV E FPGA. Information regarding worst case timings and resource utilization was obtained.

When performing a timing analysis of the CPU, the FPGA simulated under a worst-case analysis (conditions of 85°C and 1200mV), the CPU can be clocked at a frequency of 6.9MHz. This is an incredibly slow clock, and likely due to the longest data delay path in the CPU being between the HI register and the control signals IR and stall, taking 142-144ns. Looking at the CPU design, this high propagation delay stems from the use of the modulus operator for the **div** and **divu** instructions, which take a relatively enormous amount of time to complete in a single cycle. This could be improved if the design incorporated the use of a dedicated ALU / Divider block, but as all components are assumed to behave ideally in Verilog simulations it wasn't an important consideration in our design process, and so it was deemed unnecessary. If the division and modulo operations were implemented using clocked components, it is very likely that the clock rate would vastly increase. However, this would negatively affect the original design principles of the CPU, as the focal point was to minimise the CPI for the fastest possible Verilog simulation completion, where the clock rate can be set arbitrarily, and using a clocked component to perform these operations could force the CPU to wait until the values from the operations produced before continuing to execute.
Running the Fitter analysis, the design utilizes 40% (45,768 out of 114,480) of the logic elements on the FPGA. This quite a large value, and is once again likely due to the centralised design approach. By having no explicit ALU blocks in the CPU, any optimisation to reduce the number of gates needed to implement the operations performed by instructions was left to the compiler to optimise, and these results suggest that the compiler was unable to do so effectively.

Overall, the CPU performs very poorly when modelled as a physical device. However, this is no surprise as the CPU was designed for maximum performance in Verilog simulation, as this is how it is primarily benchmarked for the purposes of this assessment, and it accomplishes this perfectly with the highest possible IPC for a non-pipelined CPU. As real-world performance was unassessed, this led to valuing code legibility and implementing instruction independently much higher, without caring about the critical path and area costs.

Bibliography

Bellard, F., 2020. *Documentation - QEMU.* [Online]
Available at: https://wiki.qemu.org/Documentation
[Accessed 19 December 2020].
Bozzetti, T., Easse, E. & Tieu, C., 2014. *MIPS Converter.* [Online]
Available at: https://www.eg.bucknell.edu/~csci320/mips_web/
[Accessed 10 December 2020].
Free Software Foundation, 2020. *MIPS Options (Using the GNU Compiler Collection (GCC)).* [Online]
Available at: https://gcc.gnu.org/onlinedocs/gcc/MIPS-Options.html
[Accessed 19 December 2020].
Vollmar, K., 2017. *MARS MIPS Simulator - Missouri State University.* [Online]
Available at: https://courses.missouristate.edu/KenVollmar/mars/index.htm
[Accessed 10 December 2020].