Instruction Architectures and Compilers

Module leaders: Dr. David Thomas, Prof. Peter Y. K. Cheung

Authors: Michal Palič, Václav Pavlíček, Pablo Romo Gonzalez, Arthika Sivathasan, Henry Hausamann, Yusuf Salim

1 General Design of the CPU

1.1 Overview

The following product is a synthesisable MIPS I [3] compatible CPU with an Avalon [1] compatible memory-mapped interface allowing for easy interaction with standard memory and memory mapped peripherals. The CPU employs a Von Neumann architecture, allowing a large degree of flexibility with program and data sizes. Its set of 32-bit big-endian MIPS1 instructions makes development or migration simple thanks to existing fully featured and well tested toolchains. This version is a highly modular three cycle design suitable for lower performance applications.

1.2 Architecture and Modular Components

The CPU was implemented as a set of SystemVerilog modules interconnected by data and control signals. The modularity of this architecture is showcased in Fig. 1 allows for easy domain specific improvements for any particular use case. The seven main components are highlighted in orange. The black arrows represent the flow of data, and the red connections highlight the control signals.

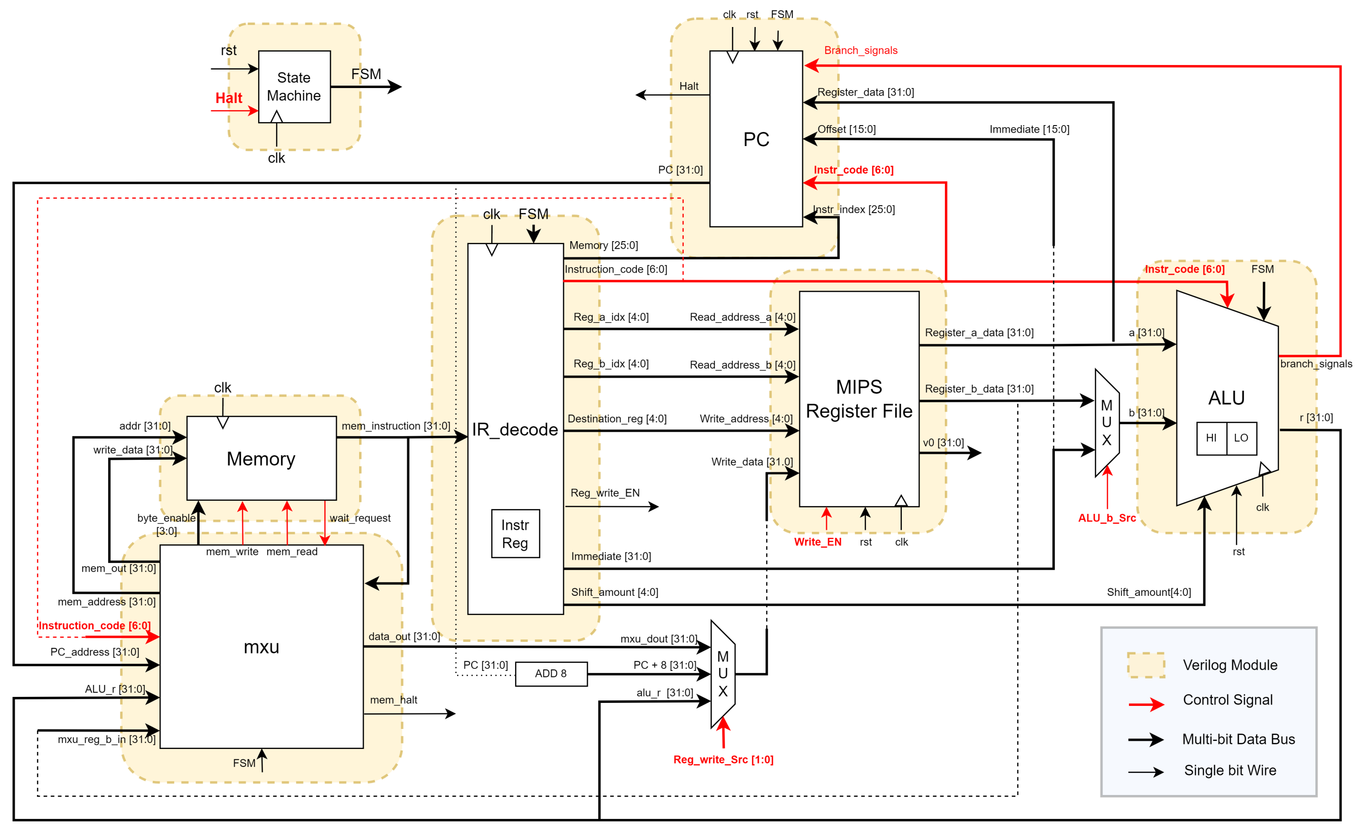


Figure 1: MIPS CPU block diagram.

In this CPU, instructions are executed over the course of 3 cycles. One to fetch the instruction to execute from memory, and two to execute data operations such as writing to registers, loading and storing. The CPU is composed

of the following seven main Verilog modules. These include the *MIPS register file*, the *state machine*, the program counter (*PC*), the arithmetic logic unit (*ALU*), the *IR_decode* block and the memory execution unit (*MXU*).
The *PC block* controls the flow of program execution by providing the current instruction address to the memory. All branches and jumps have a delay of one instruction, therefore implementing the branch delay slot which has proven useful when pipelining the design. Furthermore, the *PC block* controls the halt signal of the CPU, which is used to terminate the execution of the program. As per the specification, the CPU is halted when the instruction at address `0x0` is executed, and the reset vector is `0xBFC00000`.
The *ALU* performs combinatorial arithmetic and bitwise operations on its two operands. The result is then used either by the *MXU* in case of a memory access or saved by the *register file*. Notably, the *ALU* can perform multiplication, division and modulo, storing the results in two 32-bit registers *HI* and *LO*. These form a full 64-bit result. As the internal bus is only 32 bits wide, the result must be read in two parts using the *MFHI* and *MFLO* instructions.
The *IR_decode* block stores the current instruction during execution. It decodes the stored instruction into its respective fields and type, which are then used to configure the data path for correct instruction execution. This block is central to the function of the 3-cycle design and controls every other module.
The required operands are stored in the *register file* with one write port and two read ports. Register names are mapped to indices using the standard MIPS mapping table.
The *MXU* is used to interact with memory. Based on its inputs, the *MXU block* issues either write or read requests to the memory and then appropriately converts between the Avalon memory and internal representations of data. This block is responsible for stalling the CPU when the read or write request to the memory cannot be completed in the desired cycle.

2 Key Design Decisions

2.1 General design decisions

A key aim was ease of module integration and potential implementation of additional features. One measure used to facilitate this was encapsulation of related functionality in blocks with defined interfaces and behaviour. To simplify control signal integration, custom internal instruction opcodes were used, all of which were decoded in the *IR_decode block*, and were assigned to their respective instruction names as aliases, improving module readability. Looking ahead with regard to adding extra functionality, a conscious decision was made to use three instruction cycles to simplify the implementation of pipelining.

2.2 Memory testing facilities

Additional time and effort were used to develop facilities for ensuring that the CPU complies with the Avalon bus specification, so that no compatibility issues would arise in practice. The most notable block sought to emulate the randomness of the Avalon memory bus when multiple devices are sharing it. A 7-bit linear feedback shift register was used to generate a pseudo-random sequence of delays from request to response, one to four cycles in length. Such a memory module was used to run each test program once as part of the test suite. The long repetition period of the LFSR is expected to reasonably represent the complex interactions of more devices on the bus.

2.3 Experimental features

2.3.1 Pipelining

To maximise the throughput of the initial version of the CPU, pipelining was implemented. This feature allows the CPU to work on three instructions at once which increases the overall performance of the CPU. Pipelining was implemented by splitting each instruction into three stages – FETCH, EXEC1 and EXEC2. Different stages of consecutive instructions are then executed simultaneously. Potential problems that may arise during execution of the program are hazards that cause conflicts between pipeline stages. Data hazards were solved by the implementation of a data forwarding unit. Structural hazards due to memory accesses were resolved by inserting NOP instructions into the pipeline. Pipelining is supported only when the simple memory model is used.

2.3.2 L1 Cache

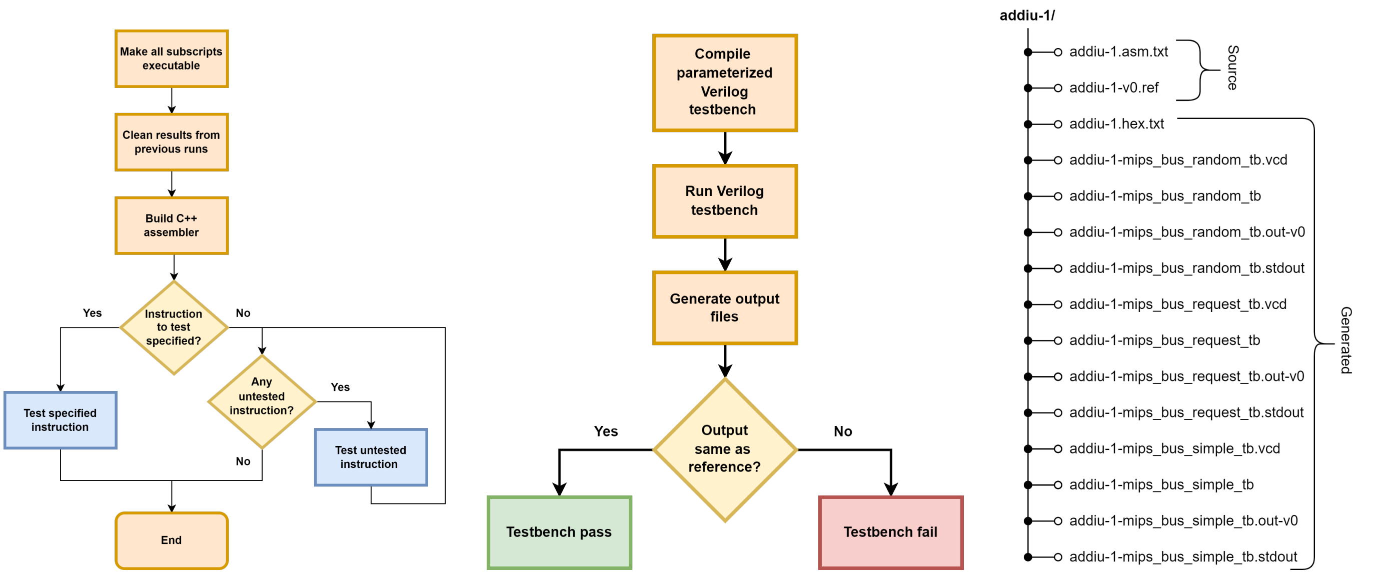
One additional feature that was implemented was that of an L1 caching system. This cache is direct mapped with 16-byte blocks and a 16kB total size. This decision was taken to give the client greater flexibility with the utilization of our IP block. The cache itself sits on the Avalon memory bus and is therefore separate and completely optional. The usage of this cache reduces the penalty for reads and writes, which results in a weaker dependence of performance on memory latency. This might be utilized to either decrease the cost of the overall system by allowing less performant memory to be used with only minor degradation of performance. Alternatively, the cache enables more innovative form factors as the further away placement of off-chip memory and the resulting increase in propagation delay is largely amortised.

2.3.3 Interrupts

Another feature implemented is the ability for the CPU to handle interrupts, such as arithmetic exceptions. The arithmetic logic unit can detect overflow, for example, and an exception signal will interrupt the main instruction stream, to execute the instruction service routine. The interrupt handler is mapped to a fixed address in the memory. This feature gives the CPU more versatility, allowing it to handle and respond to unexpected problems. It also grants more flexibility, giving the client the ability to choose how to deal with an exception, and to implement their own interrupt service routines. Alternatively the interrupt line can also be exposed to external devices to allow the CPU to respond to external events.

3 Test flow

A comprehensive test flow is part of the product, making validation of changes to the design easy. An emphasis was put on the automation of all testing processes allowing issues to be detected early in the design process. Fig. 2 describes the whole test flow. The general overview of test execution is shown in Subfig. 2a. First, all required scripts are made executable, and the testing directory is then cleaned of all output files created by previous test cases. Then the custom developed MIPS C++ assembler is built using g++. This assembler was developed with the use of a test-driven development process that ensures test coverage of critical features. An assembly program is input into the standard input of the assembler which then outputs big-endian hexadecimal representations of the provided instructions. The next part of the test flow is conditional on the tested instruction. If a particular instruction to test is provided, then only the instruction will be tested, otherwise all tests will run. Such a targeted approach allows for more efficient use of computational resources where the time to run all tests on a large design may be excessive.
The Subfig. 2b describes the test flow for a particular test case. Initially, the Verilog testbench is compiled with parameters specified for each test. The compiled file is then executed, and its output is stored to files in the test case folder. If the output of the test case is the same as the value in the reference file, then the test is marked as passing. Otherwise, the test case is marked as failing. All assembly test benches are run with three different types of memory emulation modules. These seek to replicate the different possible degrees of delay between making a memory request and its fulfilment. With the simple memory, requests are fulfilled in the next cycle. Using the request memory, the RAM first acknowledges the request which leads to a delay of two cycles. With the random memory there is a pseudo-random delay between a request and its fulfilment.
The Subfig. 2c describes the file structure for one testcase of instruction addiu-1. The folder contains two source files – the assembly and reference files that are used to set up the test cases. The reference files were generated with the MARS MIPS simulator [4]. The remaining files are generated to evaluate the test result and can be also used for debugging purposes.


Make all subscripts executable
Clean results from previous runs
Build C++ assembler
Instruction to test specified?
Yes
No
Test specified instruction
Any untested instruction?
Yes
No
Test untested instruction
End
Compile parameterized Verilog testbench
Run Verilog testbench
Generate output files
Output same as reference?
Yes
No
Testbench pass
Testbench fail
addiu-1/
addiu-1.asm.txt
addiu-1-v0.ref
Source
addiu-1.hex.txt
addiu-1-mips_bus_random_tb.vcd
addiu-1-mips_bus_random_tb
addiu-1-mips_bus_random_tb.out-v0
addiu-1-mips_bus_random_tb.stdout
addiu-1-mips_bus_request_tb.vcd
addiu-1-mips_bus_request_tb
addiu-1-mips_bus_request_tb.out-v0
addiu-1-mips_bus_request_tb.stdout
addiu-1-mips_bus_simple_tb.vcd
addiu-1-mips_bus_simple_tb
addiu-1-mips_bus_simple_tb.out-v0
addiu-1-mips_bus_simple_tb.stdout
Generated

(a) Overall test flow (b) Instruction test flow (c) Test flow file directory example

Figure 2: Testing workflow showing general test execution in Subfig. 2a, execution of particular testcase in Subfig. 2b and test file structure in Subfig. 2c.

4 Area and timing analysis

While the performance of this IP block will have a strong dependence on the utilized FPGA model, process node and or timing requirements, the following figures were achieved in one use case. The Intel Cyclone IV E FPGA and the Quartus Lite [2] suite were used to carry out the timing and area analysis of the CPU. The CPU was simulated at 1200mV at both 0C and 85C to confirm that the design will perform reliably across a range of operating conditions. The performance figures are summarised below in Table 1.

Perfomance metric / CPU metric	**3-cycle**	**Pipelined**
Restricted Fmax f [MHz] @1200mv 0C	4.6	4.34
Restricted Fmax f [MHz] @1200mv 85C	4.11	3.88
Projected CPI [1]	3	1.2

Table 1: Clock rates for non-pipelined and pipelined CPU

Upon inspection, the critical path of the 3 cycle CPU exists between the memory bus and the HI register, which is used for multiplication and division. This was expected as the relatively large 32bit word size and supported multiplication and division operations presumably lead to long internal carry chains compared to the other instructions.
The pipelined CPU clock rate ranges from 3.88MHz at 85C, to 4.34MHz at 0C. The slightly reduced clock rate is present due to the extra decode and control logic present and the resulting denser routing. This slight decrease in operating frequency is more than offset by a significant decrease in the CPI of the design. Ultimately the performance is still expected to be more than double for the pipelined version of the CPU.

4.1 Area analysis

The Intel Cyclone IV E FPGA model we used has a total of 15408 logical elements. The results in Table 2 show that our 3 cycle CPU design uses 9257 of these elements, just under two thirds of the available logical elements. The pipelined version uses only a small fraction more than the 3-cycle design.

Elements	**3 cycle**	**Pipelined**
Logical elements	9257/15408	9614/15408
Total registers	1188	1245
Logic register	1188	1245
I/O registers	0	0

Table 2: Area analysis overview

As Table 3 shows, most of the logical elements were used in the ALU, as it implements some of the most intensive operations such as multiplication and division. The register file module uses the most registers, followed by the ALU and the PC.

Module	**3 cycle**		**Pipelined**	
	Combinatorial	**Dedicated logic registers**	**Combinatorial**	**Dedicated logic registers**
mips_cpu_bus	8739	1188	9163	1245
ALU	6461	64	6484	64
IR	374	32	373	28
PC	251	65	276	97
Registerfile	1437	1024	766	1024
MXU	90	0	177	0
State Machine	4	3	N/A	N/A

Table 3: Area analysis for each module

Overall, results for the timing and area analysis suggest that the CPU design is feasible and reasonably performant on the Cyclone IV E FPGA.

[1] 20% load-store 10% jump 70% arithmeric 1 cycle memory latency

References

[1] *Avalon Interface Specifications.* URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.

[2] *Intel Quartus Prime Software Suite.* URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[3] Charles Price. *MIPS Specifications.* URL: https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf.

[4] Ken Vollmar. *MARS MIPS Simulator - Missouri State University.* URL: http://courses.missouristate.edu/KenVollmar/MARS/index.htm.