Imperial College London

32-bit MIPS-I CPU Core with Avalon Bus interface

AM-Team10: Aryan Ghobadi, Benjamin Tan, Chern Heng Tan, Kee Leong Koh, Tianyi Lim, Varun Srivasta

Overview of Features

Flexibility

- Lightweight design minimises resource usage in area-critical applications
- Parameterizable features allow for flexibility in deployment
  - User-definable data and instruction cache sizes scale CPU to application requirements
- Choice of deployment strategy with both Harvard and Avalon Bus interface implementations
  - Avalon Bus interface contains cache and Harvard CPU, allowing choice between Avalon extensibility and connectivity, and Harvard compactness and low latency

Reliability

- Robust instruction-driven testing procedure ensures conformity to a truncated set of the MIPS-I ISA and the MIPS O32 ABI

Usability

- Informative testbench toolchain speeds up debugging process
- Multiple debug traces aid in pinpointing existing CPU faults

Figures of Merit (for Cyclone IV E target)

- 14,542 logic elements | 5845 registers | 138 pins
- Maximum Clock Speed: 7.93MHz (Quartus slow 1200mV 85C summary)

Overall Block Diagram

An Avalon Bus connects the CPU to external peripherals. However, users looking to use the CPU where area or cost is critical may choose to apply the internal Harvard CPU implementation instead.

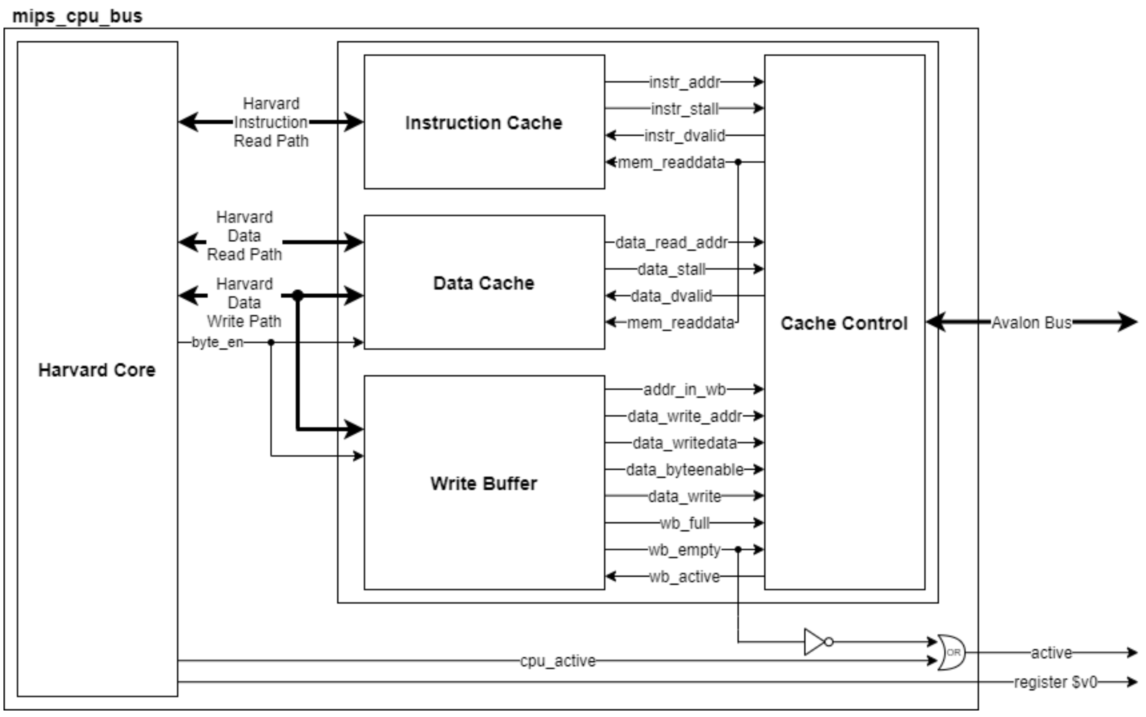


Figure 1: Block Diagram of Avalon-Bus equipped CPU

Development Process and Key Design Choices

Overview

Our Bus-based MIPS-I CPU is a **Harvard Core** wrapped in a **Cache Controller**. The **Harvard Core** implements the MIPS Instructions, while the **Cache Controller** handles memory transactions through the Avalon Bus.

The **Harvard Core** implements instruction functionality without handling the complexity that comes with memory access. The **Cache Controller** handles this complexity. This makes debugging easier and allows for efficient distribution of the development workload.

Furthermore, each component was developed in stages, being first validated on its own testbench before integration. This streamlines the development process by minimising the number of bugs at each development stage.

Harvard Core

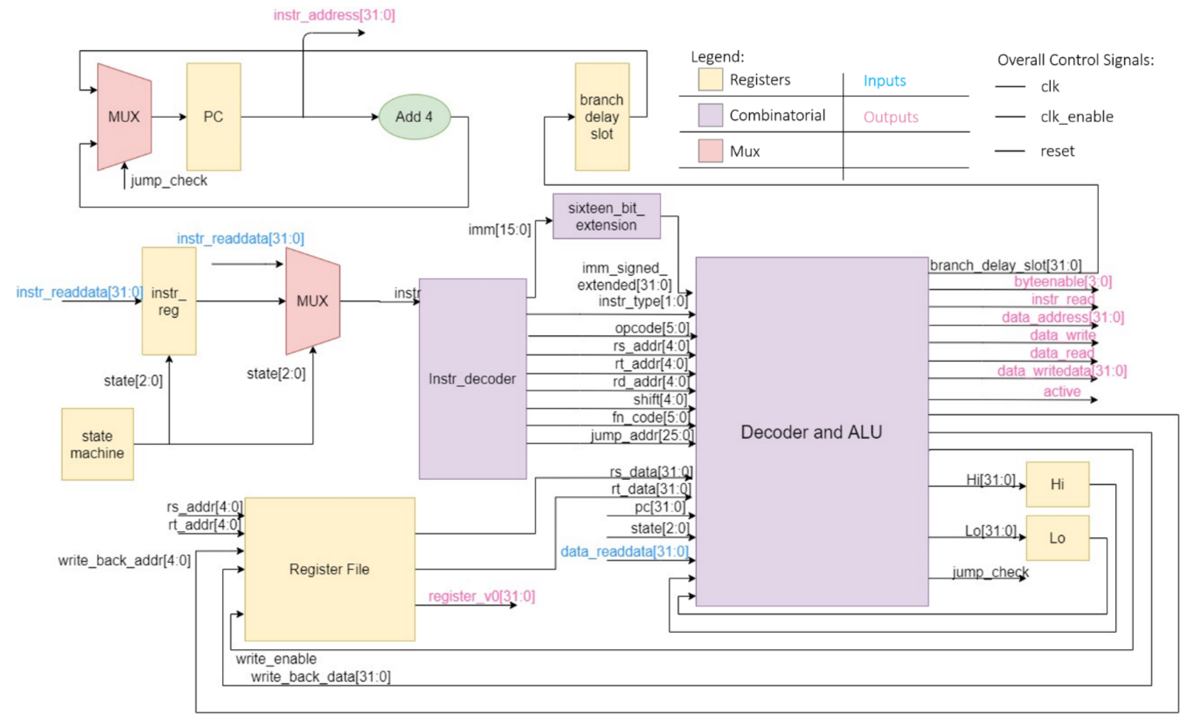


Figure 2: Harvard CPU Core

The **Harvard Core** is a non-pipelined and multi-cycle CPU. It contains 3 distinct modules: The **Register File**, **Decoder and ALU**, as well as a sign-extension module which handles the 16-bit signed extension of the immediate field. All registers are synchronous to the clock and will update on the rising edge of the clock. As the Harvard Core is non-pipelined, the branch delay slot has to be implemented intentionally using an extra 32-bit register *branch_delay_slot* which holds the address that is being branched to.

The **Register File** implements 32 registers as per the MIPS-I spec, with register *$zero* always set as the constant 0. The **Decoder and ALU** module take in the different fields that is broken down from the instruction word and use those fields to determine the inputs and behaviour of the ALU. The **Decoder** will also handle what is the expected data and instruction addresses to be given to the **Cache Controller** for memory accesses. The *byteenable* bus included with the Harvard interface allows for a more convenient adaptation to the Avalon bus, but does not otherwise affect the functionality of the **Harvard Core**.

Cache Controller

The **Cache Controller** interfaces with the Avalon Bus and presents a Harvard-like interface to the **Harvard Core**. It contains data and instruction caches, as well as a write-buffer, as seen in Fig 1.

Spatial locality cannot be exploited due to the 1-word wide Avalon Bus, so exploiting temporal locality is crucial. Hence, both caches are 4-way associative with a Pseudo-Least Recently Used replacement scheme.

As the Avalon Bus cannot read and write simultaneously, priority is given to reading over writing to minimise the cycles spent waiting for instruction fetch.

Cache Misses

If the requested memory address is not found in cache, the **Harvard Core** is stalled until the data is valid on the Avalon Bus. This data then passed over the Harvard interface to the **Harvard Core**.

Write Buffer

While there are no cache misses, the **Write Buffer** writes to memory in the background, separate from the **Harvard Core** executing instructions. Hence, unless it is full, it will not stall instruction execution, reducing the number of stall cycles.

Cache Coherency

The data in the **Data Cache** may have been replaced before the **Write Buffer** has updated the corresponding memory address.

If the currently accessed address is still in the **Write Buffer**, and there is a **Data Cache** stall, there is a chance that what is fetched from memory is out of date. Hence, the CPU stalls until that memory address has been written to, preserving coherency between the cache and memory.

Lastly, the CPU de-asserts *active* only when the **Harvard Core** has de-asserted *cpu_active* and the **Write Buffer** is empty, ensuring that both instruction execution and memory operations are complete.

Testing and Validation

Testbench

Assertions and safeguards were implemented in the testbench to help with debugging, including:

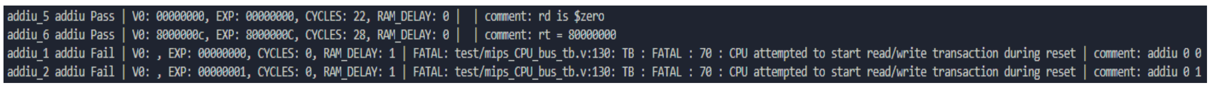
- Timeout cycles to ensure that the tests do not run indefinitely

- Holding *reset* for an arbitrary 3 cycles to ensure that the CPU works for different reset lengths

Lastly, the testbench also asserts that the test CPU obeys Avalon memory transactions.

Test Script

The Test Script compiles the CPU, runs instruction specific tests, and function tests. It then outputs a log file, waveform file and the test result. An example output is shown below:



```
addiu_5 addiu Pass | V0: 00000000, EXP: 00000000, CYCLES: 22, RAM_DELAY: 0 |  | comment: rd is $zero
addiu_6 addiu Pass | V0: 8000000c, EXP: 8000000C, CYCLES: 28, RAM_DELAY: 0 |  | comment: rt = 80000000
addiu_1 addiu Fail | V0: , EXP: 00000000, CYCLES: 0, RAM_DELAY: 1 | FATAL: test/mips_CPU_bus_tb.v:130: TB : FATAL : 70 : CPU attempted to start read/write transaction during reset | comment: addiu 0 0
addiu_2 addiu Fail | V0: , EXP: 00000001, CYCLES: 0, RAM_DELAY: 1 | FATAL: test/mips_CPU_bus_tb.v:130: TB : FATAL : 70 : CPU attempted to start read/write transaction during reset | comment: addiu 0 1
```

Figure 3:Test Script Example Output

The output shows the number of clock cycles of execution, expected and actual *$v0* output, a brief comment about the test case, and prints error messages.

In addition, the script repeats each testcase with different *waitrequest* delays to confirm the CPU can handle this aspect of the Avalon specification.

Approach to Test Cases

1) *jr*, *addiu* and *lw* were tested first as all testcases were set up using these 3 instructions and required them to function
2) A standardised data file was used for most test cases as they could reuse the same memory values
3) Individual instruction tests were formulated to check base functionality and edge cases
4) Function tests were written to check more complex interaction between different instructions.

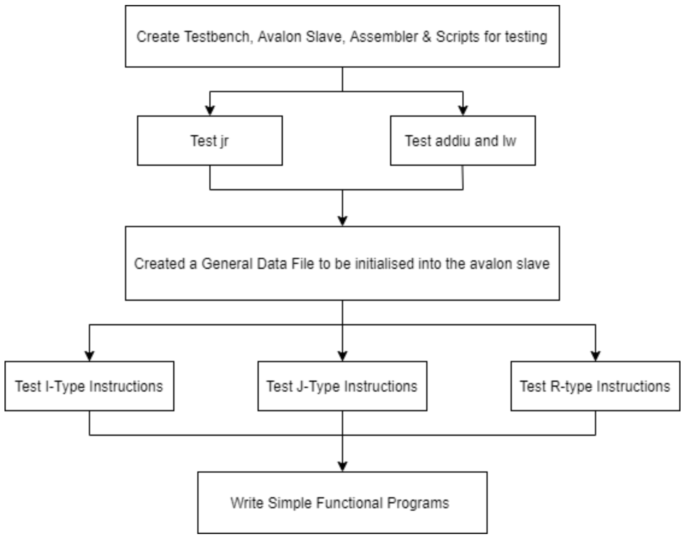

Create Testbench, Avalon Slave, Assembler & Scripts for testing
Test jr
Test addiu and lw
Created a General Data File to be initialised into the avalon slave
Test I-Type Instructions
Test J-Type Instructions
Test R-type Instructions
Write Simple Functional Programs

Figure 4: Test Framework Flowchart

Instruction Specific Tests

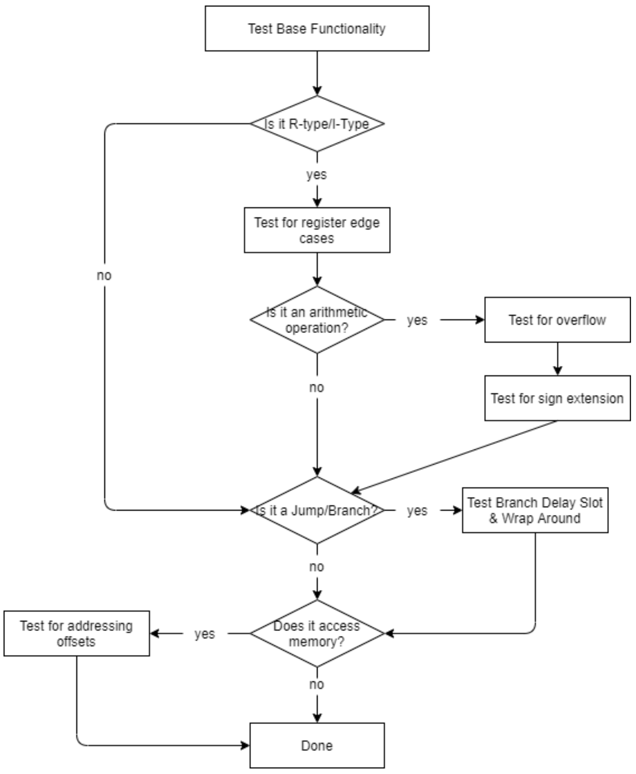


Figure 5: Instruction Testing Thought Process

Examples of register edge cases include:
- *rd* is *$zero*
- *rd* is same as *rs/rt*
- *rd* is not empty

Overflow was tested using:
- 0xFFFFFFFF (largest positive unsigned)
- 0x80000000 (largest negative)
- 0x7FFFFFFF (largest positive signed)

Some instruction specific test cases were also written (eg: *j, jal* and *jalr* were also tested to run from data memory to ensure that the CPU allowed instructions in data memory to run properly and that the jump/link addresses still function correctly.)

All test cases were setup using only *jr, lw, addiu* and the instruction being tested (excluding *mult/div* which required using *mfhi/mflo*). This allowed errors to be easily pinpointed.

Edge cases were individually tested using values which minimised failure due to other reasons so that we can isolate the problems. Test cases were also written with pseudo-random values to prevent coincidental passing.

For *sb, sh* and *sw*, the testbench outputs the content of RAM after execution and checks it against an expected output RAM to ensure that memory has indeed been modified.

Functional Tests

Test cases were created with a heavy emphasis on testing loop and recursive functionality. With Factorial, Fibonacci, as well as Iterative and Binary Array Search testcases, this ensures that the CPU under test can perform in a more real-world context.