Implementation of an Expanded RV-32I Soft Core Processor

Rishabh C S and Om Shri Prasath

Indian Institute of Technology, Madras

December 10, 2019

Problem Statement

Letting the DMEM take as long as it wants

The current DMEM is a combinational block, we assume that the reads (Load operation) from the memory happen instantaneously. However, this is not the case. Depending on the size, structure and speed of the memory, different blocks of DMEM might take different times to respond. It is

Exception Handling

The existing design does not detect errors and exceptions in the code. In the code loaded into the IMEM, there might be cases of misaligned load/store operations, trying to write into load/store into addresses which are greater in size than what is supported, or just that the instruction read from the IMEM might not be in the RISCV ISA.

Multiply Operations from the RISCV 32M ISA

The existing CPU does not execute ´Multiplyóperations. The MUL, MULH, MULHSU, MULHU operations are all powerful operations, but take long times to implement in the data path. Having the Multiply operation in the CPU reduces the complexity of programs loaded in the IMEM substantially.

Approaching the Problems

Letting the DMEM take as long as it wants

It is very useful in designing and making the memory access more efficient if it is possible for the DMEM to take as long as it wants to access the data. This is achieved by stalling further instructions in the CPU till the data from the memory is ready. In this way, we can design hierarchical memory, and Cache systems.

- The DMEM sends a ´Validśignal when the data is ready, and the CPU is stalled till the ´Validśignal is received.
- When the CPU encounters a load/store instruction, a chip enable is set to 1, and the required addresses and data is sent to the memory. After this, till the CPU receives the valid signal, all the registers in the CPU keep retaining the data.

- If the chip enable for DMEM is 1, and the valid is 0, then a śtallśignal is generated and sent to all the different stages of the pipeline. This śtallśignal is high as long as the valid is 0 and the chip enable is 1, and as long as this śtallśignal is high, all the registers in the CPU retain the value that they hold.

Implementing Multiply instruction using the above feature The Multiply instructions take a long time to implement (around 15ns). This will slow down the data path in the pipelined design. Hence, we can implement the Multiply operation in parallel to the DMEM, similar to a peripheral. In this way, the Multiply operation can take multiply clock cycles, and then send a v́alidśignal once the operation is done. The CPU will be stalled till then.

- The multiply operation is implemented similar to a peripheral.
- In the RISCV 32M ISA, the instructions MUL, MULH, MULHU and MULHSU are similar to ALU operations. But, these instructions are implemented only in the MEM stage rather than the EX stage since we are treating multiplying as a peripheral.
- The inputs to the Multiply instruction are passed on to the MEM stage from the EX stage. The register forwarding and the opcodes are generated similar to the other ALU instructions.
- The Multiply module itself is implemented using a **pipelined version of sequential multiplying**. Thus, the Multiplication module is in itself implemented using pipelining!
- The Multiplication module gives 4 operations, MUL, MULH, MULHU or MULHSU. These instructions are implemented as per the standards in the RISCV 32M ISA. A 2-bit opcode is used to choose one of these 4 operations. This 2-bit opcode is sent by the CPU.
- Once the operation is done, the Multiply module sends the valid signal. Then, the module is reset to initial conditions so that the next multiplication can be done.

Exception Handling

There are three types of exceptions required to be handled.

- Memory Align Error
- Address outside Memory Map
- Unknown Instruction Error.

The method of handling all these errors are quite similar. Once we reach the EX stage, the error are checked there based on the condition. For memory align error, we first check whether the instruction is Load or Store. If yes, check whether it is a half word or a full word read/write. If half word, check whether address last two bits are not 01 or 11, and if word, check whether the last two bits are 01,10,11. For the address outside memory map, we directly use comparison to determine whether it is a error or not. For unknown operation, we initially set our main opcode to 001111 if we are not able to understand the operation from our decoder. Then based on this opcode, we tell whether it is an error or not.
Error Handling Flow - Once an error is encountered, the CPU goes to a particular location in IMEM (which is the last address of IMEM in our case) and that address holds a instruction which loops to the same address (thereby halting the program). Before jumping to that address, the error code along with the PC is written into the last register (x31) in the following manner pc[29:0],err (Error code is a two bit number as of now, since we are dealing with only three types of error, if needed we can increase error space and decrease pc size accordingly)
Advantages - Adding more error cases (if any), is easier, we just need to increase the length allotted for the error type identifier to be written and set its value accordingly.
Disadvantages - For the address error, we did not implement the error signal coming from DMEM or IMEM block, we assumed them to be fixed sizes and approached the problem. But since we were able to get the stall option from DMEM and implement it, this can also be done similarly.

Work Splitup

- Valid signal generation for DMEM, Multiply module Multiplier, and Integrating multiply into the pipelined architecture - Rishabh
- Exception Handling - Om Shri Prasath

Implementation Report

- The files required for implementing this assignment is :
  - rf.v
  - pipeline_wb.v
  - pipeline_mem.v
  - pipeline_ex.v
  - pipeline_id.v
  - pipeline_if.v
  - peripheral.v
  - multiply.v
  - dmem_decoder.v
  - dmem.v
  - cpu_tb.v
  - CPU.v
  - alu_32_bit.v
- The test IMEM files are :
  - imem_error_check_1.mem
  - imem_error_check_2.mem
  - imem_error_check_3.mem
  - imem_multiply.mem
  - imem3_ini.mem (For testing correct code)
  - top.v (For Spartan Board implementation)
- For runnning the test cases, change the name of the file to be loaded into IMEM memory in pipeline_if.v based on the above imem files and then the code can be simulated and tested.
- For implementing on board, the **top.v** file is already provided, we need ICON module, VIO module (with 1 bit ASYNC_OUT and 96 bit SYNC_IN, and ILA module (with triggers as pc, x31, x3 and reset, reset being the main trigger)