RISC-V FPGA Implementation

Milestone 4

Mohamed Abdelfattah

Muhammad Azzazy

Marc Boutros

Abdallah Gabara

INTRODUCTION

RISC-V is an open source Instruction Set Architecture that enables developers in both hardware and software to innovate . Since emerged from academics and research it provides a great source of flexibility and freeness in terms of designing new hardware and software related features. RISC-V has a lot of cores that are built on it and also a lot of SoC platforms. The power of RISC-V comes from the fact that it is highly flexible with a variety of versions to enable different levels of usage and development. Also users can add their own features. One of the very basic versions of RISC-V is RISC-V 32-I which is an ISA that is designed only for integer operations and for 32-bit word-register sizes. As the name implies it is a risc ISA which means it has simple instructions that can be combined to handle larger operations and tasks.

Specifications

In this project, it is required to develop a RISC-V 32I processor supporting all instructions in the documentation using Verilog. The cycle of the processor is pipelined so that an instruction takes 5 clock cycles with a 2 clock cycles gap between the start of 2 consecutive instructions as illustrated in the figure below. This 2 clock cycles gap is introduced due to the presence of a single memory for instructions. This 2 cycles gap guarantees that the memory will not be accessed for data and instructions at the same clock cycle which reduces potential hazards. Furthermore, the processor is to support the compressed instructions form which require additional hardware. The implementation should be able to handle any type of data dependency that might cause hazards, either through the forwarding unit or through stalling the whole pipeline until the hazard is resolved.

Data Path

* Please refer to the data path that will be attached as a separate file (since the path is large)

The data path started as a single stage (single cycle implementation) which was very slow due to the inefficiency in utilizing the hardware resources available. We were required to upgrade the already implemented single cycle datapath to a fully pipelined one. The full pipeline is composed of five main stages with a register between each of the stages to preserve the values of the propagating instruction between the clock cycles. I will give an overview of the pipeline as a whole (illustrations of specific additional units will appear later in the report)

1) Fetch stage:

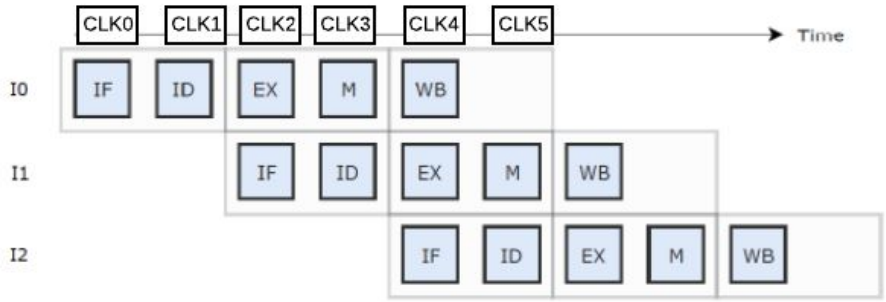
This stage is responsible for fetching the instruction from the memory and this is done through having a register (PC) pointing always at the next instruction to be fetched from the memory. Since we only have one memory which is single-ported with the data and instructions are together in the same memory, The fetching in our design occurs once each two clock cycles to avoid accessing the memory for data and instructions in the same clock cycle. In this stage, the new value of the PC has to be determined in order to fetch the correct instruction from the memory; thus the PC itself has many sources that enter a multiplexer to be selected from based on some control signals that originate from different units in different stages in the pipeline. One of the important sources for the PC is PC_ADDER which is responsible for incrementing the pc (four bytes to jump for the following instruction) in the memory. This is not always the case since the instruction

could be a branch or a jump instruction and in that case the PC should not jump by four.

2) Decoding stage:

In this stage, the instruction that was fetched from the memory has to be decoded in order to gather useful information about the instruction (the type of the instruction, the source registers, …). In our implementation, this stage is very critical. Starting with the basic units in this stage, we have the register file which contains 32 registers of 32 bits each. These registers are initialized to a value of zero and accessed through different control lines. These registers are very fast in operation (close to the speed of the processor) and that's why we use them instead of reading from the memory every time. Between the fetching and decoding stage there is a register that saves the information of the instruction that was fetched and which will propagate to the decoding stage since a new instruction will be fetched after two clock cycles. The second important unit is the immediate generator unit which extends the immediate value in some instructions based on the type of each instruction. The third important unit is the control unit which is the main controller of the data path as it generates all the control signals of the instruction that is in the decoding stage and these signals are to propagate with this instruction to be used in different other stages until the instruction is fully executed. The control unit governs the operations of most of the other units in the processor and it only requires the type of the instruction. The last two units exist there as a way of avoiding data dependency hazards and also avoiding wasting clock cycles (in case of branching). The first one of these two is the forwarding/hazards detection unit. This unit is responsible for forwarding some of the operands in case that the pipeline did not fully output them. Since all the instructions have to pass through all the stages, sometimes this might cause

a situation where the data (operand) is ready in an early stage but the pipeline has to finish all the stages in order to hand this data to the instruction that needs it. This is the main reason why we have the forwarding unit. An example is illustrated in the following figure



Pipelined CPU with Every-other-Cycle Instruction Issuing

This is the main diagram of the pipeline. Now assume that I0 is a load instruction and that it writes in register X, now assume the I1 is an R-type instruction the uses register X as one of its operands. After clock cycle 3 (clock cycle zero is at the first IF) the value will be ready at the memory stage and at the same time the next instruction is in need of this value, without the forwarding unit we would either stall until the value is written to the register (this is not possible also because the value need to be available in the decoding stage) or we will read a wrong value of the register. Using the forwarding unit enables us to use the value directly from the memory stage without the need to complete the full pipeline

This unit does not only handle forwarding but it also handles hazard detection where

there is a real dependency that requires the whole pipeline to stall, this unit produces the suitable signals that will stall the pipeline.

The final unit is the branching unit. This unit can be implemented either in this stage, the execution stage or the memory stage. The earlier this unit in the pipeline, the faster the branch is resolved (whether it should be taken or not) and the lower the number of wasted clock cycles in case of a miss branch decision, that's why we decided to implement this unit as early as possible. This early implementation comes at the cost of introducing new hazards that were not accounted for and thus the requirement of extra hardware to be added to be able to handle these hazards (more will be illustrated when we talk about the branch unit)

3) Execution stage:

This is a very important stage in the pipeline as it accounts for the main processing unit of the processor: the ALU. The ALU is capable of performing various functionalities and thus there has to exist a separate control unit that will instruct the ALU of the operation it should carry based on some control signals from the main control unit of the system. This value that the ALU will evaluate will be used either as an address or as data based on the type of instruction.

4) Memory stage:

In this stage, the data/address that was generated by the ALU will be used to access (read) or write to the single memory resulting in updating its content.

5) Write Back stage:

This is the last stage of the pipeline where the value that was generated either by the ALU or read from the memory will be written back to the register file (not necessarily).

Branching

The branching unit is placed in the decoding stage rather than the execution stage. This reduces the potential hazards due to the fact that the decision of whether to branch or not is taken before the next instruction get decoded. If a branching instruction is to be executed, the branching unit receives the relevant 2 integers to be compared from 2 muxes. These 2 muxes are controlled by a signal coming from the forwarding unit to determine whether to use the value from the execution phase of the previous instruction or the value stored in the registers. Consequently, The branching occurs in the next cycle.

Compressed instructions

In order to support compressed instruction a special module is added to handle instructions in this format. Once an instruction word is fetched from the memory, the instruction is passed to an extender module. The extender checks the format of the word. If the word ends with 11, the instruction is outputted with no changes as it is not a compressed instruction. Otherwise, the instruction is extended to its 32-bit version so that the processor is able to execute it. In order for instructions to be executed correctly, the PC must be incremented by 2 in the event of a compressed instruction and 4 otherwise. This is done by introducing a mux to select between 2 or 4 as an addition to the PC based on a select signal derived by the least significant bits of the instruction being fetched. This way, the PC is incremented by 2 in the case of a compressed instruction and by 4 in the case of a uncompressed instruction

Forwarding

Forwarding to the execution stage is needed only if the preceding instruction is loading a value from memory to one and/or both the registers that should provide the values to be operated upon by the ALU. Forwarding is needed as well for the decoding stage. This is the case since the branching modules are operating during the decoding stage of the instruction, therefore, registers holding values for the comparison as well as registers holding values for the branching address (as with jalr) need to be ready at this stage whereas the previous instruction is in the memory stage. The forwarding unit can forward the output of the execution stage as well as that of the memory once the hazard is properly dealt with as shown in the following section.

Hazards

The only hazard that requires special handling in the implemented hardware occurs when a jump instruction is preceded by a load instruction. In this case a stall signal is activated to halt the datapath for 1 cycle so that the jump instruction is implemented correctly when the needed values are forwarded from the memory stage in the following cycle.

Bram

A first attempt was made to use the FPGA's block ram for the single memory by letting Vivado infer it automatically. This would not work however, whenever the support for instructions writing to multiple bytes in memory are supported.

A second attempt was made by using Vivado's IP configurator, this approach was abandoned since if the word size in memory was 1 byte, multiple clock cycles (2*4 in case of 4 bytes) would be for each word read or write. On the other hand, having each memory location holing 4 bytes and allowing byte-writing to support byte and halfword writes would still produce the need for more than 2 clock cycles due to the fact that the RISCV architecture is only byte aligned. Not having the word alignment constraint means that a load word to a memory location like (33) would still require more clock cycles as the entire word required by the instruction is stored in two distinct memory locations. The need for extra clock cycles made implementing the memory using BRAMs cumbersome.