IITB-RISC: EE309 Course Project

Team ID-20

Group Members:

1. Keshav Maheshwari – 22B3951
2. Reeyansh Shah – 22B0412
3. Nishant Bhave – 22B2144
4. Rajwardhan Toraskar – 22B0721

Work Distribution:

1. Keshav Maheshwari – ALU, Dataforwarding,Report
2. Reeyansh Shah – Datapath, Hazard Detection,Testing
3. Nishant Bhave – Datapath,Hazard Detection,Testing
4. Rajwardhan Toraskar – Testbench

Introduction

We Designed a 6-stage pipelined processor, *IITB-RISC-23*. *IITB-RISC* is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The *IITB-RISC-23* is a 16-bit computer system with 8 registers. It should follow the standard 6 stage pipelines (Instruction fetch, instruction decode, register read, execute, memory access, and write back). The architecture should be optimized for performance, i.e., should include hazard mitigation techniques. Hence, it should have implemented forwarding mechanism.

IITB-RISC is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The *IITB-RISC-23* is an 8-register, 16-bit computer system. It has 8 general purpose registers (R0 to R7). Register R0 is always stores Program Counter. All addresses are byte addresses and instructions. Always it fetches two bytes for instruction and data. This architecture uses condition code register which has two flags Carry flag (*C*) and Zero flag (*Z*). The *IITB-RISC-23* is very simple, but it is general enough to solve complex problems. The architecture allows predicated instruction execution and multiple load and store execution.

1 Components

Let’s have a look at the components that we’ve used. We've employed 8 registers

grouped into a Register File, a memory unit, and an ALU. Moreover, we've incorporated combinational logic in each of these elements to determine their inputs and outputs. Below, we outline each component utilized in our VHDL code.

ALU

Input Ports: ALU A, ALU B, ALU load A, ALU load B, r dest, C, Z, it3, it5.

Output Ports: Output, C out and Z out.

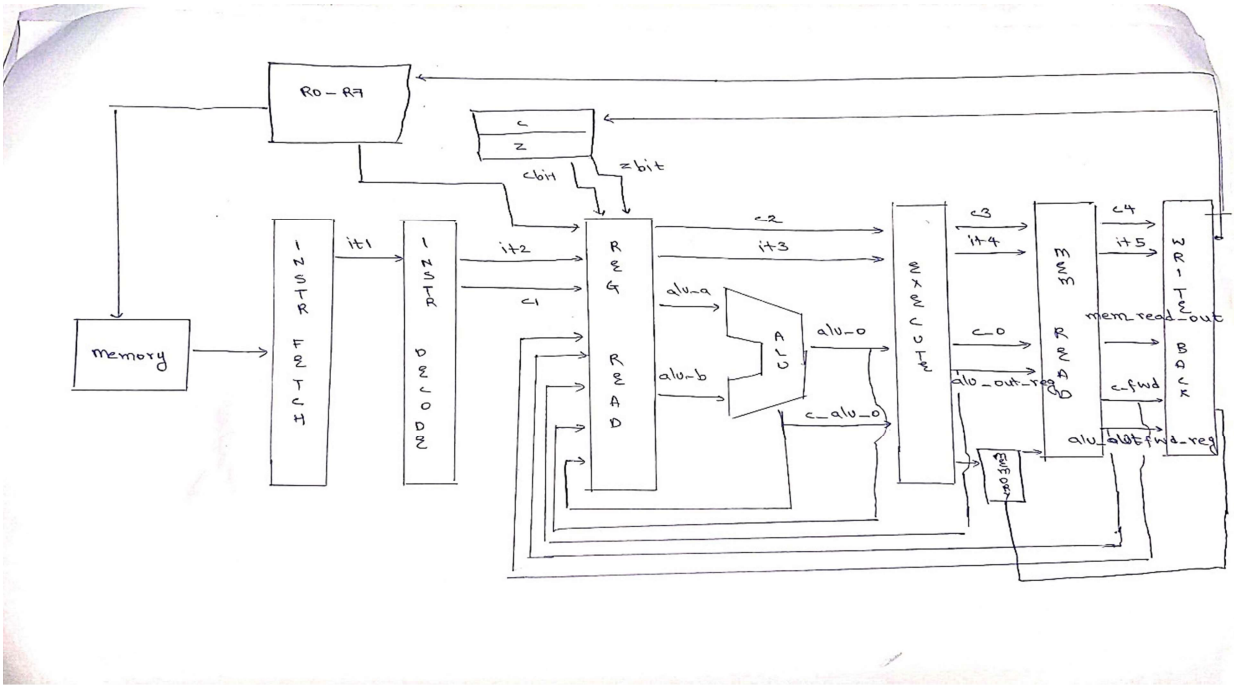
Functioning - During the register read stage, the ALU receives accurate inputs. The updating of the C and Z flags hinges on their designation for writing in a specific instruction, determined by the it3 register. To ensure precise data forwarding, the ALU's output is directed to the destination register only if a write operation is intended; otherwise, the previous value of the destination register is maintained to prevent incorrect inputs in data forwarding scenarios.
In cases where a load instruction is followed immediately by an arithmetic or logical operation involving the loaded value, the output of the memory read stage is fed directly to the ALU.
This strategy leverages the delay introduced by the load instruction, ensuring that the memory output is available for computation before being written into the register file. Additional checks, including the use of additional ports and the decoding of the it5 register to determine if it was a load instruction, are necessary to handle preceding instructions in this process.

Memory and Register File

The memory architecture of this processor consists of 16-bit wide storage elements organized in a stack comprising 1024 rows, with each row being 16 bits wide. The total storage capacity of this memory amounts to 16,384 bits or 4 kilobytes. The decision to opt for a 16-bit addressable memory, rather than a 16-byte one, was driven by the aim to minimize the demand on FPGA resources. Using a 16-byte addressable memory proved problematic during compilation, as it exceeded the available FPGA resources, particularly encountering errors with 216 registers.

As for the register file, it comprises 8 registers labeled R0 to R7, each 16 bits wide. Notably, R0 always stores the program counter.

Datapath



All the components in our project are attached according to the above flowchart.

Pipeline Stages

Instruction fetch

To update the instruction register (it1) with the fetched instruction from memory, we initially encountered a challenge requiring an if-else block to verify whether the preceding instruction was a load or branch. In such cases, the updating of the program counter (R0) needed to be halted. However, we encountered conflicts due to identical blocks in both the instruction fetch and write-back stages. To address this, we merged these blocks and streamlined the R0 updating process within the Write Back stage. Ultimately, since all operations occur simultaneously at the clock edge, relocating this process to a different block does not pose any issues.

When the R0 change is 1, indicating a branch or jump instruction, the write-back signals for the subsequent four instructions are disabled to prevent modifications to the register file or memory. The count of these next four instructions is tracked using the R0 change wait counter

Instruction Decode

The instruction fetched from memory serves to assign control signals (reg write, mem write, C write, and Z write). A signal, 'is it load,' is set to 1 if the decoded instruction corresponds to a load operation. Additionally, the program counter (R0) will be updated for jump instructions.

Based on whether the fetched instruction is a load or a jump, the c1 signal is updated to determine whether to write into memory or update the registers with different values, as follows:

If 'is it load' is 0 and the R0 change wait counter is 0, then c1 = xxx01 if the register has to be updated, or c1 = xxx10 if memory has to be updated. For all other cases, c1 = xxx00.

Similarly, the C and Z flags are updated based on whether a carry or zero is generated or not. The c1 signal is updated as follows:

If 'is it load' is 0 and the R0 change wait counter is 0, then c1 = x11xx if both C and Z need to be updated, or c1 = x01xx if only Z needs to be updated, and c1 = x00xx if neither of them needs to be updated.

Register Read

In this stage, ALU inputs are assigned. Data forwarding occurs from the next three stages, considering whether the previous instruction was an arithmetic or logical operation that modified the register file value. Additionally, carry and Z inputs are assigned in this stage and undergo data forwarding as well.

For branch or jump instructions, we provide the program counter (R0) and immediate value as inputs to the ALU to calculate the required jump address.

Execution

The ALU executes the instruction based on the inputs provided to it. Its output, along with the C flag and control bits from the execution stage (it3 and c2), are forwarded to the next stage. Additionally, in this stage, data forwarding for load-dependent instructions is facilitated by mapping the memory read output to the ALU inputs.

Crucially, if a write operation does not occur, the ALU output mirrors the old value of the destination register. This precautionary measure helps prevent data forwarding errors in subsequent stages.

Memory Read

The data is read from memory address given by alu out. The output from ALU, C flag, and the controls bits of memory read stage (it4 and c3) are forwarded to the next stage.

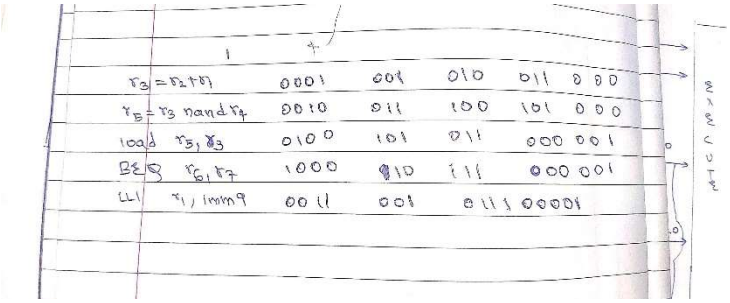
Write Back

If mem write is 1, then the data in the register address given in the instruction is stored in the memory address provided by the ALU output. Since mem write is set to 1 only if it isn't during a load or branch waiting time, no additional signals need to be checked.

The subsequent changes are executed only if the r0 change wait counter is 0, a value set in the instruction fetch stage. If 'is it load' is 1, indicating a load instruction, neither the R0 nor the register file contents are altered. Thus, we resend the same instruction again for one cycle and set load wait over to 1, which resets 'is it load' to 0, allowing subsequent instructions to continue normally.

If it is an instruction other than a load, then the R0 is incremented by 1 (R0+1) to proceed to the next instruction. The register files are updated according to the instruction, and the carry and zero flags are set based on the forwarded data. We continuously check the instruction or0ode to determine if a write operation should indeed occur.

For branch and jump instructions, the R0 change wait counter increments from 0 to 4 over 4 cycles until the branch/jump instruction is executed. After execution, the R0 is set to the new value, and normal execution resumes.



Hazards Control

Data Dependency for Arithmetic/Logical

In our system, we implement data forwarding in the register read stage from three subsequent stages. This process is managed seamlessly without halting the clock or R0 update because the ALU output becomes available immediately after the inputs are adjusted based on the preceding instruction. Whether the ALU output is effectively written onto the register file or not is managed by considering the or0ode of the instruction being executed in the ALU and the previous values of the C and Z flags. This approach ensures efficient and synchronized operation without interruptions

Data Dependency for Load

In this scenario, introducing a one-cycle delay, or halt, is crucial because the ALU initially computes the address from which data needs to be fetched, and the output becomes available from memory only after two cycles. To address this, we implement a separate

update block for signals ALU load A and ALU load B. This update occurs at the rising edge of the clock instead of the typical falling edge, ensuring that the ALU output is obtained with just a one-cycle delay. This adjustment facilitates the synchronization of data flow, allowing for seamless processing within the system.

Branch/Jump and Load Instructions

To accommodate the necessary halts in our system, we implement specific mechanisms based on the instruction type:

Load Instructions:

When encountering a load instruction, we need to halt the R0 update and set write signals to 0 for one cycle. This ensures that the processor waits until the data is fetched from memory before proceeding.

This halt is managed using the "is it load" signal.

Branch and Jump Instructions:

For branch and jump instructions, we need to wait until the R0 is updated before proceeding. This requires a delay of 4 cycles.

To achieve this, we utilize the "r0 change" and "r0 change wait counter" signals.

The "r0 change" signal indicates whether a branch or jump instruction has been encountered, while the "r0 change wait counter" tracks the number of cycles until the R0 is updated.

During this waiting period, the processor halts its operation until the R0 is updated, ensuring correct branching and jumping behavior.

By leveraging these signals effectively, we can implement the necessary halts and delays to ensure proper instruction execution and maintain the integrity of the processor's operation.

Testing

To simulate and test the processor we followed these steps:

1. Write the instructions using assembly language in the 'Instructions.txt' file located in the 'Testing' directory. Ensure that instructions are written without commas.
2. Execute the 'tester.py' script. This script will generate the corresponding binary format instructions and save them in a file called 'decoded.txt' in the same folder.
3. Copy the content of 'decoded.txt'.
4. Paste the copied content into your VHDL file where the memory is initialized. This will ensure that the processor is initialized with the correct instructions for testing.