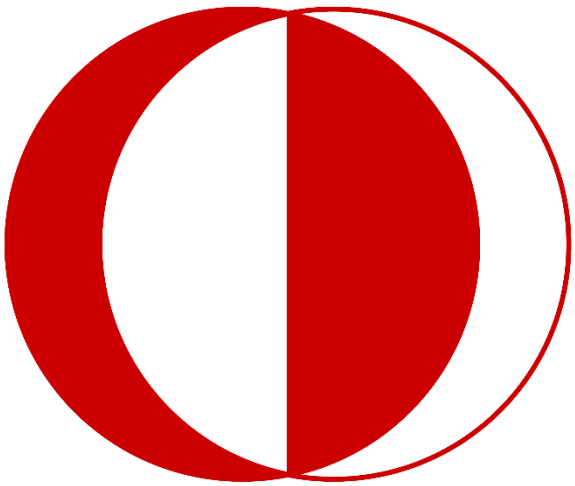

EE 446 Labaratory Project

Pipelined Processor with Hazard Unit and Branch Predictor

Yasincan Bozkurt 2304202

Uğur Tokdam 2305498

Table of Contents

Inroduction

The functionality and architecture of processors are constantly changing in the quickly developing field of computing to keep up with the escalating demands for speed and efficiency. One such advancement is the pipelined processor, a device that improves the performance of the processor by enabling the simultaneous execution of several instructions. While this instruction overlap significantly speeds up the process, it also brings certain complexity, most notably risks. The extended 32-bit pipelined processor we designed and implemented, complete with an integrated hazard unit and branch predictor, is described in detail in this project report. Our objective was to successfully negotiate these complexity and develop a system that effectively manages hazards and precisely forecasts branching to maximize performance. The processor's capability was then shown by embedding it inside the DE1-SoC board's FPGA.

Objectives

This project's main goal is to improve the 32-bit pipelined processor design from the last lab session. The improvements to the processor's datapath and control unit include the addition of a hazard unit and a branch predictor. The designed processor can now execute every instruction in the expanded set thanks to our extension of the instruction set. Finally, in order to demonstrate the functionality of our design, we integrated it into the DE1-SoC board's FPGA.

Contributions

The project was executed collaboratively with equal contribution from both team members. Contributions can be seen in Table 1.

Group Member	Yasincan Bozkurt	Uğur Tokdam
Contributions	Control Unit, Branch Predictor	Datapath, Hazard Unit

Table 1. Contributions

Design and Implementation

Our design began with a working pipelined processor from the 4th lab work. We then extended this base design with a hazard unit and a branch predictor.

Datapath

We added more instructions, such BEQ, BL, and BX, to the 32-bit pipelined processor created in Lab 4 by extending its Instruction Set Architecture (ISA) in the datapath. We created a thorough datapath that implemented a hazard unit and a branch predictor in order to accommodate the whole range of operations. This improvement changed a number of our current modules in addition to increasing the capabilities of the processor. We changed the register file to only allow write operations on the clock's negative edge, making it possible to store instruction results in the registers more effectively. We were able to better align our write operations with the cycle time and increase the system's overall clock speed by using the clock's negative edge for writing.

The ARM architecture's fundamental concepts define how data processing instructions work. They include operations such as ADD, SUB, AND, ORR, MOV, and CMP. The five phases of our processor's pipeline—Fetch, Decode, Execute, Memory, and Writeback—are used to carry out each of these instructions.

The instruction is read from memory during the fetch step using the current Program Counter (PC) value. Reading the fetched instruction, deciphering the opcode, identifying the operation type (such as

ADD, SUB, AND, etc.), and reading the required operands from the register file are all steps in the decoding process.

In the Execute stage, the appropriate Arithmetic Logic Unit (ALU) operation is performed based on the opcode. The shift operation on the second operand for data processing instructions is also implemented here, as per the requirements.

Memory operations occur in the Memory stage. However, for data processing instructions, this stage does not alter the state as these instructions do not involve explicit memory access.

Finally, the result of the operation is written back to the destination register in the Writeback stage. When it comes to data hazard handling in our datapath, we adopted a forwarding approach. Data hazards occur when instructions that depend on each other are too close in the pipeline, resulting in incorrect execution due to outdated or incorrect operand values. To mitigate this, our design employs a hazard detection unit that identifies potential data hazards and controls multiplexers for data forwarding.

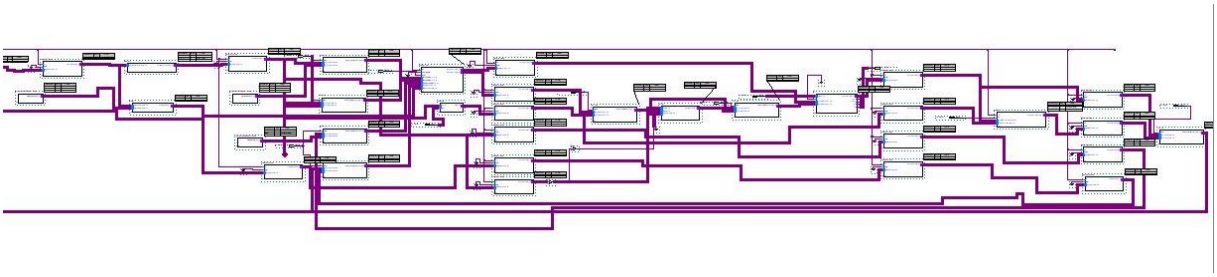

Figure 1. Datapath

The pipeline's instructions are examined by the hazard detection unit to see if any of them provide a data risk. When a data hazard is identified, the hazard unit stops the current cycle's write activity by delaying that stage and its predecessors until the required data is available. The dependent instruction won't execute until the source instruction has written its results, due to this stalling mechanism.

In addition, the hazard unit facilitates data forwarding by directing the multiplexers to forward the appropriate data to the ALU. This allows the dependent instruction to access the most recent data directly from the pipeline, instead of waiting for it to be written back to the register file. This mechanism significantly improves the efficiency of our datapath, ensuring correct and efficient execution of instructions.

Our processor's memory instructions include the STR (Store) and LDR (Load) operations. The pipeline stages used for data processing instructions (Fetch, Decode, Execute, Memory, and Writeback) are also used for the execution of these memory instructions.

The memory instruction is retrieved from memory during the Fetch stage using the current Program Counter (PC) value.

The Decode stage decodes the instruction that was fetched and accesses the required source registers. It reads the source register that will be saved for the STR instruction. It reads the base address register to execute the LDR instruction.

The effective address is determined during the Execute stage. In our implementation, the memory address where the data will be loaded from or put to is determined by adding an instantaneous offset to the value of the base register. The real memory operation happens in the Memory stage. Data from the source register is saved in the memory location determined in the Execute step for the STR

instruction. The data is read from the estimated memory address for the LDR operation. Only the LDR instruction makes use of the Writeback stage, where data received from memory is sent to the target register.

For handling data hazards in memory instructions, we have implemented the hazard unit and data forwarding, similar to data processing instructions. Data hazards can occur in memory instructions, especially when a memory write operation (STR) is followed by a memory read operation (LDR) with overlapping data. When a data hazard is detected, the hazard unit stalls the pipeline, ensuring that the data is written to memory before it's read by the subsequent instruction.

Additionally, we have implemented data forwarding in the memory stage for LDR instructions. If a subsequent instruction is dependent on the value loaded by the LDR instruction, the data is forwarded directly to the dependent instruction once it's available. This forwarding mechanism reduces the time that the dependent instruction needs to wait, thus improving the overall pipeline efficiency.

Branch instructions in our datapath include BEQ (Branch if Equal), B (unconditional Branch), BL (Branch with Link), and BX (Branch and Exchange). These instructions are critical for control flow in programs and require special handling in pipelined architectures to mitigate control hazards. Using the current Program Counter (PC) value, the branch instruction is fetched from memory during the fetch stage. Stage of Decode translates the retrieved instruction. Here, the condition is assessed for conditional branches like BEQ, and the link register (R14) is updated with the return address for branches like BL. The branch operation is handled by the Execute stage. If the BEQ condition is satisfied, the target address is updated on the PC. B's computer is constantly updated. Along with changing the link register during the decoding step, the PC is updated for BL in a manner similar to B. The value in the designated register is updated in the PC for BX.

Due to the lag between fetching a branch instruction and updating the PC in the Execute step, control hazards develop in the pipeline. Following instructions from the incorrect path may be fetched and sent to the pipeline during this delay, which could result in improper execution. We have put in place a branch prediction unit and a hazard unit to deal with control hazards. As soon as the branch instruction is decoded, the branch prediction unit forecasts the outcome of the branch. In order to lessen the impact of branch misprediction, this prediction is utilized to get instructions from the expected path.

The hazard unit intervenes if the branch prediction was inaccurate (branch misprediction). It supplies the pipeline with instructions from the proper path after flushing the instructions that entered it through the incorrect way. The correctness of the program is preserved by the flushing of stages, which makes sure that instructions from the wrong path are not carried out. By reducing the control risks brought on by branch instructions, our datapath design promotes effective and accurate program execution.

Hazard Unit

In our pipelined processor design, the hazard unit is developed to handle data and control dangers, guaranteeing that programs run quickly while maintaining their accuracy. When an instruction tries to read a register value that hasn't yet been modified by a previous instruction, data dangers happen. We employ a forwarding and stopping method to control these data risks. We use forwarding in order to do away with the requirement for pausing while dealing with risks brought on by data activities. We avoid wasting cycles waiting for the write-back stage by sending the computed value back to the stage where it was needed.

Control hazards occur from the pipeline's inability to predict the following instruction to fetch, especially when branch instructions are involved. Depending on the branch instruction type, we handle

control threats differently. A branch prediction technique is used to forecast the branch's expected conclusion as soon as feasible for branch instructions that employ immediate values (B, BL, and their conditional variations). Despite the fact that this tactic typically increases effectiveness, it is possible for the branch predictor to produce erroneous predictions. We resort to flushing the improperly obtained instructions from the pipeline when a misprediction occurs.

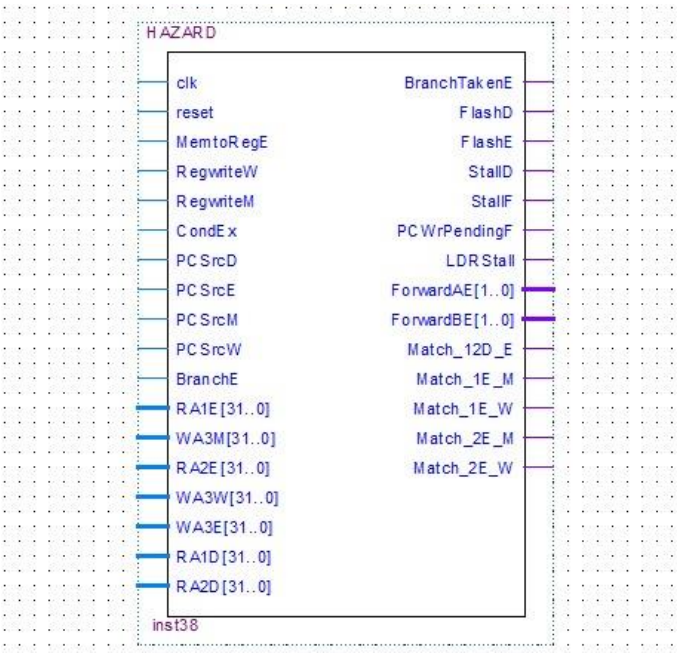

Figure 2. Hazard Unit

Branch Predictor

The branch predictor, a crucial part of our pipelined processor design, is composed of three main components: the Branch Target Buffer (BTB), the Global History Register (GHR), and the Pattern History Table (PHT). This structure allows us to predict the next instruction to be fetched, effectively handling control hazards in the case of conditional branch instructions.

The most recent branch instructions' PCs and matching branch target addresses are cached in the Branch Target Buffer. The branch operation is sped up by BTB, which can quickly supply the target address if a match is found by comparing the current PC with the stored PCs. Three submissions can fit in our BTB. When the BTB is full, the oldest entry (based on usage) will be overwritten if a new entry is inserted. A reset signal is also included in the BTB for initialization.

The predictor's most basic though crucial component is the Global History Register. It performs the function of a shift register, noting the results of the most recent branches with a '1' for taken and a '0' for not taken. For each new branch result, the register moves left while still keeping a three-bit history.

Together with the GHR, the Pattern History Table stores the results of branch decisions made in accordance with earlier branch patterns noted in the GHR. The PHT has eight parts that are each one bit broad thanks to our three-bit GHR. These elements are simply one-bit predictors, with '1' denoting that a branch was taken with a particular history and '0' denoting that it wasn't. A reset signal is also included in the PHT for initialization.

The BTB, GHR, and PHT work together to forecast the next fetch address in the case of branch instructions, greatly enhancing the pipelined processor's overall performance. Despite the possibility

of inaccurate projections, we have put in place procedures to deal with them and keep the orders executed precisely.

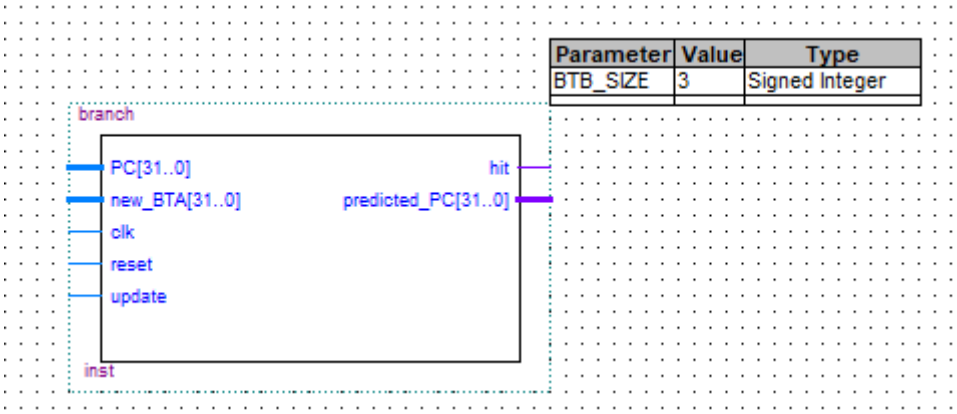

Figure 3. Branch Predictor

Conclusion

In conclusion, we successfully expanded the capabilities of a 32-bit pipelined processor by integrating a hazard unit and a branch predictor. The hazard unit adeptly manages data and control hazards to prevent cycle wastage, while the branch predictor effectively deals with control hazards linked to branch instructions.

These improvements, along with the implemented extended instruction set, have greatly enhanced the processor's performance by ensuring efficient cycle use, minimal stalling, and robust hazard handling. While further optimization possibilities exist, this project provides a solid basis for future advancements in processor design, offering valuable insights into the intricate dynamics of such systems.