Implementation of Pipelined Processor Design

제출일	2023, 06, 16	전공	컴퓨터학과
과목	컴퓨터구조	학번	2023320046
담당교수	김영근	이름	정승우

제출일	2023, 06, 16	전공	컴퓨터학과
과목	컴퓨터구조	학번	2019320076
담당교수	김영근	이름	한건희

Contents

#1 Overview of the pipelined ARM processor

The pipelined ARM processor offers a great solution for improving the throughput of digital systems. By dividing the single-cycle processor into five pipeline stages, it allows for concurrent execution of instructions, resulting in significantly enhanced performance.

The heart of the pipelined ARM processor is its pipelined datapath, which consists of five stages: fetch, decode, execute, memory, and writeBack. These stages are carefully designed to evenly distribute the workload and address slower operations like memory access and register file operations. This balanced approach enables an increase in clock frequency, leading to improved overall throughput.

Below shows each pipeline stage:

Fetch: This stage retrieves instructions from the instruction memory.
Decode: The processor reads the source operands from the register file and decodes the instructions to generate the necessary control signals.
Execute: In this stage, the processor performs computations using the Arithmetic Logic Unit (ALU).
Memory: This stage involves reading from or writing to the data memory.
Writeback: The writeback stage is responsible for writing back the results of the executed instruction to the register file, if applicable.

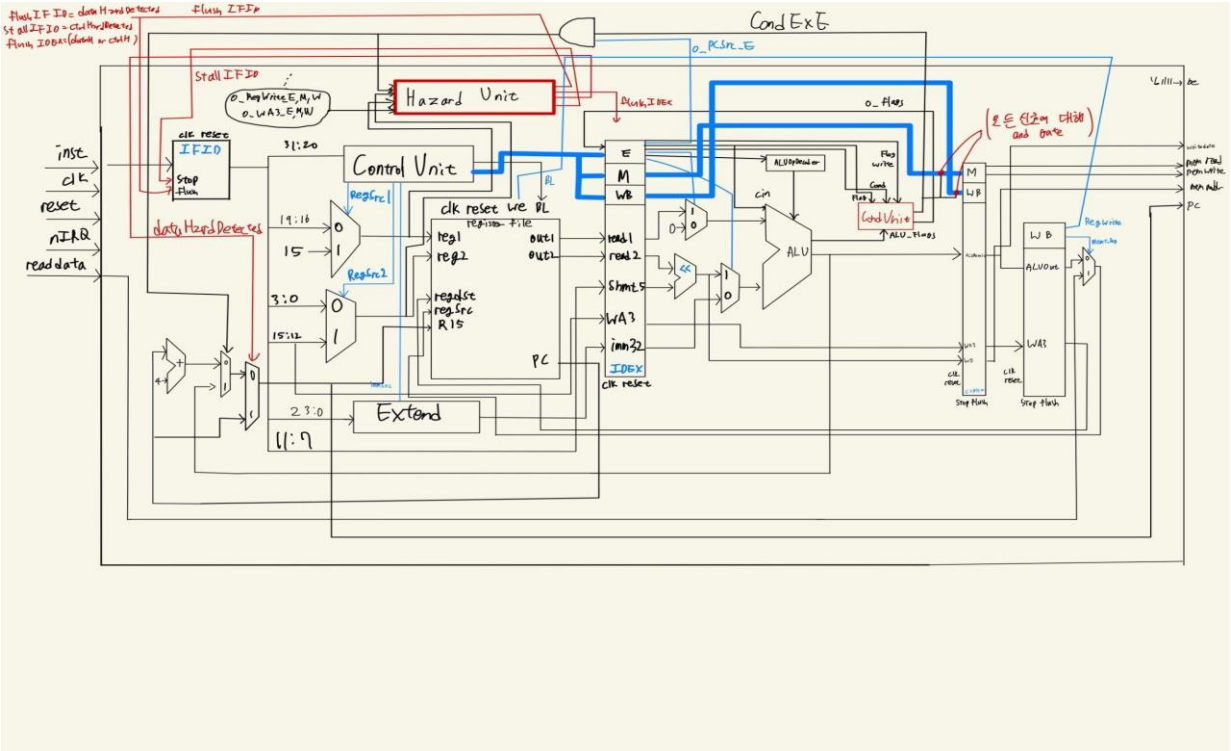
By operating on instructions independently in each stage and utilizing stage registers, instructions flow smoothly through the pipeline,

enabling concurrent processing and significantly improving throughput.

Managing hazards is a central challenge in pipelined systems, as subsequent instructions may depend on the results of previous instructions that have not completed. Techniques such as forwarding, stalls, and flushes are employed to address these hazards and ensure accurate instruction execution. Special attention is given to the design of the pipelined datapath to maintain synchronization. Signals associated with each instruction must progress through the pipeline in harmony to ensure proper operation. The register file and PC logic are carefully designed to support this synchronization.

In conclusion, the pipelined ARM processor revolutionizes throughput by enabling simultaneous execution of multiple instructions across distinct stages. It effectively addresses delays in memory access and register file operations, making it an essential feature in modern high-performance microprocessors.

The figure below is a block diagram of the pipelined processor of our implementation.



#2 Explanation about control signals

In an ARM pipelined processor, the control signals play a crucial role in coordinating and synchronizing the various stages of the pipeline. These control signals determine the operations to be performed at each stage of the pipeline and ensure that the instructions are executed correctly.

In our code, newControlUnit module generates control signals. Below is the explanation about our implementation.

Inputs:

inst (27:20) - Input representing the instruction.
Rd (15:12) - Input representing the destination register address.

Outputs:

RegSrc1 - Output indicating the source of the first register operand.
RegSrc2 - Output indicating the source of the second register operand.
immSrc (1:0) - Output indicating the source of the immediate value.
BL - Output indicating the branch and link control signal.
NZCVWrite - Output indicating the control signal for writing to the NZCV register.
ALUSrc1 - Output indicating the source of the first ALU operand.
ALUSrc2 - Output indicating the source of the second ALU operand.
InstOp (3:0) - Output indicating the ALU operation.
PCSrc - Output indicating the source of the program counter (PC).
MemWrite - Output indicating the memory write control signal.
MemRead - Output indicating the memory read control signal.
RegWrite - Output indicating the register write control signal.
MemtoReg - Output indicating the memory-to-register transfer control signal.

The newControlUnit module is responsible for generating control

signals based on the input instruction (inst) and the destination register address (Rd). These control signals determine various aspects of the processor's operation, ensuring proper execution of instructions in the pipeline.

The control signal generation is implemented whenever there is a change in the input signals. The module utilizes conditional statements and a case statement to determine the appropriate control signal values based on the instruction and register address.

The control signal generation logic is described as follows:

Branch and Link Control (BL):

If the most significant bit of inst is set or the destination register (Rd) is equal to 4'b1111, BL is set to indicate a branch with link operation. Otherwise, BL is not set.

Register Operand Sources (RegSrc1 and RegSrc2):

The sources for the first and second register operands depend on the instruction being executed. This information is encoded in the instruction itself.
The specific values for RegSrc1 and RegSrc2 are determined based on the instruction and assigned accordingly.

Immediate Sources (immSrc):

The source of the immediate value depends on bits [20:21] of the instruction.
The specific values for immSrc are determined based on the instruction and assigned accordingly.

ALU Operation (InstOp):

The ALU operation to be performed depends on bits [24:21] of the instruction.
The specific values for InstOp are determined based on the instruction

and assigned accordingly.

Program Counter Source (PCSrc):

The source of the program counter (PC) depends on the branch instruction being executed.
The specific value for PCSrc is determined based on the instruction and assigned accordingly.

Memory Read/Write Control (MemRead and MemWrite):

If the instruction is a load (LDR), MemRead is set to enable memory read.
If the instruction is a store (STR), MemWrite is set to enable memory write.
For other instructions, both MemRead and MemWrite are not set.
Register Write Control (RegWrite):
If the instruction writes to a register (excluding the condition flags register), RegWrite is set.
Otherwise, RegWrite is not set.
Memory-to-Register Transfer Control (MemtoReg):
If the instruction is a load (LDR), MemtoReg is set to enable memory-to-register transfer.
For other instructions, MemtoReg is not set.

Condition Flags Write Control (NZCVWrite):

If the instruction writes to the condition flags register, NZCVWrite is set.
Otherwise, NZCVWrite is not set.
The control signals are stored in a 17-bit register (control) within the module and then assigned to the corresponding output ports.
The newControlUnit module enables efficient instruction decoding and control signal generation, ensuring proper execution of instructions in the processor pipeline. By accurately determining the control signals based on the instruction and destination register address, the module contributes to the overall functionality and performance of the

processor.

#3 How we handle Data hazard & Control hazard

There is a module to handle those hazards, newHazardUnit.

The newHazardUnit module is responsible for detecting and handling both data hazards and control hazards.

1. Data Hazard Handling:

Data hazards occur when a dependent instruction relies on the result of a previous instruction that has not yet produced its output. The newHazardUnit module detects data hazards by comparing the register addresses of the write operations (o_WA3_E, o_WA3_M, o_WA3_W) with the read registers (read1 and read2) in the execute (E), memory (M), and writeback (W) stages.
If any of the conditions (read1 == o_WA3_E || read2 == o_WA3_E), (read1 == o_WA3_M || read2 == o_WA3_M), or (read1 == o_WA3_W || read2 == o_WA3_W) evaluate to 1, it indicates a match between a read register and a write address, and a data hazard is present. The dataHzrdDetected output is set to 1 when a data hazard is detected.

2. Control Hazard Handling:

Control hazards occur when the control flow of the program is altered, leading to potential incorrect instruction execution.
The newHazardUnit module partially handles control hazards. The i_PCSrc_D input represents the control hazard signal, indicating

whether a control hazard has occurred. i_PCSrc_D is made by o_PCSrc_E && CondExE. And CondExE is from output of newConditionUnit model, which decides to make that instruction valid or not.
The ctrlHzrdDetected output is assigned the value of i_PCSrc_D, indicating whether a control hazard has been detected.
If ctrlHzrdDetected is set to 1, it implies a control hazard is present.

3. Stall and Flush Handling:

If a data hazard is detected (dataHzrdDetected == 1), the stallIFID output is set to 1.
The stallIFID signal causes the fetch (IF) and decode (ID) stages to be stalled, allowing time for the data hazard to be resolved.
Additionally, the flushIFID output is set to 1 if a control hazard is detected (ctrlHzrdDetected == 1).
The flushIFID signal indicates that the instructions in the fetch (IF) and decode (ID) stages should be discarded or flushed to prevent incorrect execution.
The flushIDEX output is set to 1 if either a data hazard or a control hazard is detected.
This ensures that instructions in the instruction decode (ID) and execute (EX) stages are flushed to avoid executing incorrect or potentially outdated instructions.

In summary, the newHazardUnit module handles both data hazards and, to some extent, control hazards by detecting hazards, setting appropriate control signals, and managing the stall and flush conditions.

The image attached below is a code for intuitive understanding.

```
newHazardUnit HazardUnit(
 .read1(reg1_D[i_RegSrc1_D]),
 .read2(reg2_D[i_RegSrc2_D]),
 .o_RegWrite_E(o_RegWrite_E),
 .o_WA3_E(o_WA3_E),
 .o_RegWrite_M(o_RegWrite_M),
 .o_WA3_M(o_WA3_M),
 .o_RegWrite_W(o_RegWrite_W),
 .o_WA3_W(o_WA3_W),
 .i_PCSrc_D(o_PCSrc_E && CondExE),
 .dataHzrdDetected(dataHzrdDetected),
 .ctrlHzrdDetected(ctrlHzrdDetected),
 .stallIFID(stallIFID),
 .flushIFID(flushIFID),
 .flushIDEX(flushIDEX));

assign PCNext[0] = i_wire_pc + 4;
assign PCNext[1] = i_ALUResult_E;
assign DataHzrdPCNext[0] = PCNext[o_PCSrc_E && CondExE];
assign DataHzrdPCNext[1] = i_wire_pc; // current pc, same

registerfile registerFile(.clk(clk), .reset(reset), .reg1(reg1_D[i_RegSrc1_D]),
 .reg2(reg2_D[i_RegSrc2_D]), .regdst(o_WA3_W), .regsrc(Result_W[o_MemtoReg_W]),
 .R15(DataHzrdPCNext[dataHzrdDetected]), .BL(BL), .we(o_RegWrite_W),
 .pc(o_wire_pc), .out1(i_read1_D), .out2(i_read2_D));
assign i_wire_pc = o_wire_pc;
assign pc = o_wire_pc;
```

#4 Limitation and more idea

Since lack of knowledge about Verilog, although we put a lot of time fixing the problem, it did not work. Because Functional simulation is not working as well as timing simulation, the problem is not about the critical path of our CPU.

There might be some problems regarding to combinational logics, which is SLACK. We tried to understand the contents of timing analysis, it was hard.

We did our best to solve it, but the problems are not fixable for us.

If we had enough knowledge about Verilog and how to use quartus program, we might solve the problem.

Reference

Sarah Harris, David Harris (2015). ***Digital Design and Computer Architecture*: (ARM Edition)**. Morgan Kaufmann Publishers Inc.