# **Date:8/13/24**

# 

# **EE488 - Computer Architecture**

# **Final exam**

**NAME : S A SABBIRUL MOHOSIN NAIM**

**ID : 20176**

**Question ans : 01**

.data

prompt: .asciiz "Enter an integer: "

result\_msg: .asciiz "The two's complement is: "

.text

.globl main

main:

# Print the prompt message

li $v0, 4 # syscall for print\_string

la $a0, prompt # load address of the prompt message

syscall

# Read an integer from the user

li $v0, 5 # syscall for read\_int

syscall

move $t0, $v0 # store the input integer in $t0

# Calculate two's complement

# Two's complement: invert the bits and add 1

not $t1, $t0 # invert the bits of the input integer

addi $t1, $t1, 1 # add 1 to get two's complement

# Print the result message

li $v0, 4 # syscall for print\_string

la $a0, result\_msg # load address of the result message

syscall

# Print the two's complement integer

move $a0, $t1 # move the two's complement to $a0 for printing

li $v0, 1 # syscall for print\_int

syscall

# Exit program

li $v0, 10 # syscall for exit

syscall

**Question ans : 02**

.data

prompt: .asciiz "Enter an integer: "

newline: .asciiz "\n"

star: .asciiz "\*"

.text

.globl main

main:

# Print the prompt message

li $v0, 4

la $a0, prompt

syscall

# Read an integer from the user

li $v0, 5

syscall

move $t0, $v0 # move the read integer to $t0

# Outer loop for each line of the triangle

li $t1, 1 # initialize line counter to 1

line\_loop:

ble $t1, $t0, process\_line # continue if line counter is less than or equal to input

j end\_program

process\_line:

# Inner loop to print stars in each line

li $t2, 0 # initialize star counter to 0

star\_loop:

li $v0, 4

la $a0, star

syscall

addi $t2, $t2, 1 # increment star counter

blt $t2, $t1, star\_loop # continue if star counter is less than line counter

# Print newline after finishing line of stars

li $v0, 4

la $a0, newline

syscall

addi $t1, $t1, 1 # increment line counter

j line\_loop # jump back to the start of line\_loop

end\_program:

# Exit program

li $v0, 10

syscall

**Question ans : 03**

To design an effective Program Counter (PC) for a 32-bit machine, several key considerations need to be taken into account. Here's a description of an effective method to design the PC:

**1. 32-bit Register:**

- The PC should be implemented as a 32-bit register to match the machine's architecture.

- This allows addressing of up to 4GB of memory (2^32 bytes).

**2. Initialization:**

- The PC should be initialized to the starting address of the program when execution begins.

- Typically, this is done by loading a predetermined value (e.g., 0x00000000) at system reset.

**3. Increment Logic:**

- In most cases, the PC should automatically increment by 4 after each instruction fetch.

- This is because in a 32-bit machine, instructions are typically 4 bytes (32 bits) long.

- Implement an adder that adds 4 to the current PC value.

**4. Branch and Jump Support:**

- Include logic to handle non-sequential execution flows like branches and jumps.

- For conditional branches, implement a multiplexer to select between PC+4 and the branch target address.

- For jumps, allow direct loading of the jump target address into the PC.

**5. Pipeline Considerations:**

- In a pipelined processor, the PC should be updated at the beginning of each clock cycle.

- Implement logic to handle pipeline stalls and flushes, ensuring the PC maintains the correct value.

**6. Exception Handling:**

- Include mechanisms to save the current PC value when an exception occurs.

- Provide a way to load the PC with the address of the appropriate exception handler.

**7. Memory Interface:**

- Ensure the PC can interface with the instruction memory or cache.

- Implement proper timing and control signals for memory access.

**8. Clock Synchronization:**

- The PC should be updated synchronously with the processor's clock.

- Use edge-triggered flip-flops to maintain stability and prevent timing issues.

**9. Power Efficiency:**

- Implement clock gating to reduce power consumption when the processor is idle.

**10. Testability:**

- Include scan chain support for testing purposes.

- Implement debug features like the ability to read and write the PC value directly.

**11. Byte Addressing:**

- Ensure the PC increments and addresses memory correctly, considering that memory is byte-addressable but instructions are word-aligned.

**12. Modularity:**

- Design the PC as a separate module with clear interfaces to other processor components.

- This allows for easier testing, modification, and potential reuse in different designs.

By incorporating these features, the PC design will be effective for a 32-bit machine, providing accurate instruction sequencing, supporting various control flow mechanisms, and integrating well with other processor components. This design ensures efficient program execution while maintaining flexibility for different types of instructions and addressing modes.

**Question ans : 04**

While it's theoretically possible to execute an instruction in a single cycle, modern processor architectures predominantly use multiple-cycle datapaths for several important reasons. Let's explore these reasons in detail:

**1. Clock Speed Limitations:**

- In a single-cycle design, the clock period must be long enough to accommodate the slowest instruction.

- This severely limits the overall clock speed of the processor, as faster instructions are forced to wait.

- Multiple-cycle designs allow for higher clock speeds by breaking operations into smaller steps.

**2. Complex Instructions:**

- Modern instruction sets often include complex operations (e.g., floating-point calculations, SIMD instructions).

- These complex instructions would require an extremely long cycle time if implemented in a single cycle.

- Multiple cycles allow these operations to be broken down into manageable steps.

**3. Memory Access Latency:**

- Memory operations, especially those involving cache misses or external memory, can take significantly longer than simple ALU operations.

- A single-cycle design would need to accommodate the worst-case memory access time for every instruction.

- Multiple cycles allow memory operations to be split across cycles, with stalling mechanisms for longer operations.

**4. Power Efficiency:**

- Single-cycle designs require all components to be active for every instruction, even if not all are used.

- Multi-cycle designs allow for more granular power management, activating only necessary components in each cycle.

- This leads to significant power savings, crucial for modern processors, especially in mobile devices.

**5. Pipelining Benefits:**

- Multiple-cycle designs naturally lend themselves to pipelining, where different stages of multiple instructions can be executed simultaneously.

- This dramatically increases throughput without requiring a proportional increase in hardware.

- Pipelining is a key feature in almost all modern processors for performance enhancement.

**6. Resource Sharing:**

- Multi-cycle designs allow for efficient sharing of hardware resources among different instruction types.

- For example, the same ALU can be used for arithmetic in one cycle and address calculation in another.

- This leads to more efficient use of chip area and reduced complexity.

**7. Timing and Synchronization:**

- Breaking operations into multiple cycles makes it easier to manage timing and synchronization issues.

- It allows for the insertion of pipeline registers, which help in maintaining signal integrity and reducing clock skew.

**8. Instruction-Level Parallelism:**

- Multiple-cycle designs, especially when pipelined, allow for instruction-level parallelism.

- This means parts of multiple instructions can be executed simultaneously, increasing overall throughput.

**9. Easier Design and Verification:**

- Dividing instruction execution into multiple stages simplifies the design and verification process.

- It's easier to isolate and debug issues in smaller, discrete stages than in a complex single-cycle design.

**10. Flexibility for Different Instruction Types:**

- Different types of instructions can take different numbers of cycles as needed.

- This allows for a more efficient balance between simple and complex instructions.

**11. Support for Advanced Features:**

- Multiple-cycle designs better support advanced features like out-of-order execution, speculative execution, and branch prediction.

- These features significantly enhance performance but require complex control logic that's difficult to implement in a single cycle.

**12. Scalability:**

- Multi-cycle designs are more scalable, allowing for easier implementation of more complex instruction sets and architectural improvements over time.

In conclusion, while single-cycle execution is theoretically possible, the practical limitations in terms of performance, efficiency, and functionality make multiple-cycle datapaths the preferred choice in modern processor architectures. The benefits of increased clock speed, better resource utilization, power efficiency, and support for advanced features far outweigh the theoretical simplicity of a single-cycle design.

**Question ans : 05**

Designing the control functional block for a multiple cycle datapath in processor architecture requires careful planning and implementation. The goal is to correctly sequence the execution of each part of the instruction through the datapath across multiple cycles. Here, I will outline the steps for designing control signals and also provide a conceptual diagram.

**Key Components of Control Signals:**

**1. Instruction Fetch (IF):**

- IRWrite: Loads the fetched instruction into the instruction register.

- PCWrite: Updates the program counter to point to the next instruction.

- PCWriteCond: Conditionally updates the program counter for branch instructions.

- PCSource: Determines the source of the next program counter value (sequential, branch, or jump).

**2. Instruction Decode (ID):**

- Control Signal Generation: Based on the opcode and function fields of the instruction, generate signals to control the rest of the processor operations.

**3. Execution (EX):**

- ALUOp: Specifies the operation the ALU should perform.

- ALUSrc: Determines the second operand of the ALU (from registers or immediate).

- RegDst: Selects the register destination (for R-type instructions).

**4. Memory Access (MEM):**

- MemRead: Enables reading from memory (for load instructions).

- MemWrite: Enables writing to memory (for store instructions).

- Branch: Determines whether a branch should be taken.

**5. Write Back (WB):**

- RegWrite: Enables writing to the register file.

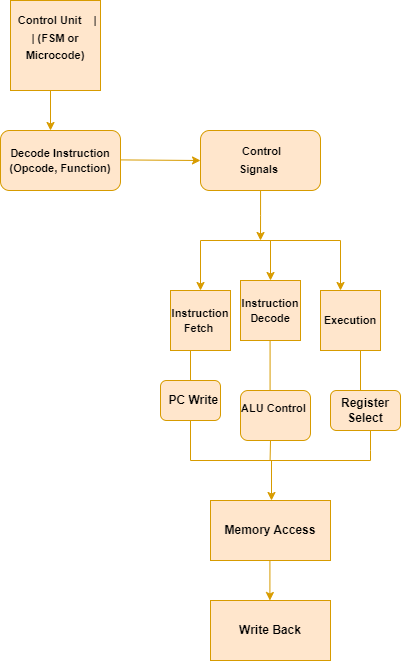
- MemtoReg: Determines whether the value to write comes from memory or the ALU.

**Control Signal Generation:**

Control signals are typically generated by a finite state machine (FSM) or a microprogrammed control unit that outputs the appropriate signals based on the current instruction type and state of execution. The control unit uses the opcode and function codes from the instruction to generate these signals.

**Diagram Concept:**

Here’s a simplified conceptual diagram that might represent how control signals interact with a multiple cycle datapath.



**Implementing the FSM Control:**

**1. State Transitions:**

- Each major operation phase (Fetch, Decode, Execute, Memory, Write Back) is represented as a state in the FSM.

- Transitions between states are based on the completion of the required actions in each state.

**2. Generating Control Signals:**

- In the fetch state, `PCWrite` and `IRWrite` are asserted.

- In the decode state, signals like `ALUSrc`, `ALUOp`, and `RegDst` are set based on decoding the instruction.

- Execute, Memory Access, and Write Back states assert specific controls based on the instruction type (load, store, arithmetic, etc.).

This approach of using an FSM to generate control signals ensures that each stage of the instruction execution pipeline is correctly managed and timed, which is critical for maintaining correct operation and high performance in a multiple cycle datapath.

**Advanced Implementations:**

For more complex processors, control units might also integrate elements like branch prediction and out-of-order execution logic, which further complicates the control signal logic but improves overall efficiency and speed. These elements would typically be represented as additional blocks or layers in the control unit diagram.

**Question ans : 06**

Let's analyze the five-stage pipelining architecture for the given scenario and calculate the instruction execution latency and throughput.

Given:

- 5 pipeline stages: Fetch, Decode, Execute, Memory, and Writeback

- 6 instructions: 3 R-types and 3 I-types

- Clock frequency: 1 GHz (1 clock cycle = 1 ns)

**Instruction Execution Latency:**

The latency is the time it takes for a single instruction to complete all five stages.

Latency = Number of stages × Clock cycle time

= 5 stages × 1 ns/stage

= 5 ns per instruction

**Throughput:**

In an ideal pipeline without stalls or hazards, once the pipeline is full, we complete one instruction per clock cycle.

Throughput = Number of instructions / Time period

= 1 instruction / 1 ns

= 1 instruction per ns

For 10 ns:

Throughput (per 10 ns) = 10 instructions

**Pipeline Execution:**

Let's visualize the execution of the 6 instructions over time:

**Clock Cycle: 1 2 3 4 5 6 7 8 9 10**

Instruction 1: **F D E M W**

Instruction 2:  **F D E M W**

Instruction 3: **F D E M W**

Instruction 4: **F D E M W**

Instruction 5:  **F D E M W**

Instruction 6:  **F D E M W**

```

F: Fetch, D: Decode, E: Execute, M: Memory, W: Writeback

**Analysis:**

- The first instruction takes 5 cycles to complete (5 ns latency).

- From the 5th cycle onward, one instruction completes every cycle.

- By the 10th cycle, 6 instructions have been fetched, and 5 have completed.

**Actual Throughput for this scenario:**

In 10 ns, 5 instructions have completed fully.

Actual Throughput (per 10 ns) = 5 instructions

**Conclusion:**

1. Instruction Execution Latency: 5 ns per instruction

2. Theoretical Throughput (per 10 ns): 10 instructions

Actual Throughput for given 6 instructions (per 10 ns): 5 instructions

3. The difference between theoretical and actual throughput is due to the initial pipeline fill time. In a continuous stream of instructions, the throughput would approach the theoretical maximum of 1 instruction per ns.

4. The type of instructions (R-type or I-type) doesn't affect the latency or throughput in this ideal scenario, assuming no data hazards or structural hazards. In a real processor, different instruction types might have different execution times or resource requirements, potentially affecting the pipeline efficiency.

5. This analysis assumes an ideal scenario without pipeline stalls, branch mispredictions, or data hazards. In real-world scenarios, these factors would reduce the actual throughput and increase average latency.

**Question ans : 07**

To address varying instruction lengths between R-type and load instructions while avoiding structural hazards in a pipelined processor design, several methods can be employed. Let's explore these design approaches:

**1. Uniform Pipeline Stage Duration:**

- Design all pipeline stages to have the same duration, based on the longest instruction type.

- Pro: Simplifies pipeline control and avoids structural hazards.

- Con: May introduce unnecessary delays for shorter instructions.

**2. Multiple-Cycle Execution Stage:**

- Allow the Execute stage to take multiple cycles for complex instructions.

- Use a completion signal to indicate when the instruction is ready to move to the next stage.

- Pro: Accommodates varying instruction lengths without slowing down simpler instructions.

- Con: Requires more complex control logic.

**3. Parallel Functional Units:**

- Implement separate functional units for different instruction types (e.g., ALU for R-type, address calculation unit for loads).

- Allow instructions to use the appropriate unit without conflict.

- Pro: Improves throughput and avoids structural hazards.

- Con: Increases hardware complexity and cost.

**4. Instruction Queues:**

- Implement instruction queues between pipeline stages.

- Allow faster instructions to proceed while slower ones are still executing.

- Pro: Improves overall pipeline efficiency.

- Con: Requires additional hardware and control logic.

**5. Dynamic Scheduling:**

- Implement out-of-order execution with techniques like Tomasulo's algorithm.

- Allow instructions to execute as soon as their operands are ready, regardless of program order.

- Pro: Maximizes resource utilization and handles varying instruction lengths effectively.

- Con: Significantly increases design complexity.

**6. Pipeline Interlocking:**

- Implement a mechanism to stall the pipeline when a long-latency instruction is detected.

- Resume normal operation once the instruction completes.

- Pro: Simple to implement and effective at preventing hazards.

- Con: Can lead to performance penalties due to frequent stalls.

**7. Instruction Predecoding:**

- Add a predecode stage that quickly determines instruction type and potential resource needs.

- Use this information to guide instruction routing and scheduling.

- Pro: Allows for more informed decision-making in later stages.

- Con: Adds an extra pipeline stage, potentially increasing latency.

**8. Resource Reservation:**

- Implement a reservation system where instructions "book" required resources in advance.

- Stall or reroute instructions if resources are unavailable.

- Pro: Prevents structural hazards by ensuring resource availability.

- Con: Requires additional control logic and may introduce delays.

**9. Superscalar Architecture:**

- Implement multiple execution units that can handle different instruction types simultaneously.

- Use instruction dispatch logic to route instructions to appropriate units.

- Pro: Significantly improves throughput for mixed instruction types.

- Con: Increases design complexity and hardware cost.

**10. Variable-Length Pipeline:**

- Design a pipeline with a variable number of stages based on instruction type.

- Use bypass paths for shorter instructions to skip unnecessary stages.

- Pro: Optimizes execution time for each instruction type.

- Con: Complicates pipeline control and hazard detection.

**11. Micro-operation Decomposition:**

- Break down complex instructions into simpler micro-operations.

- Execute these micro-ops through a uniform pipeline.

- Pro: Simplifies pipeline design and execution.

- Con: May increase the total number of pipeline stages for complex instructions.

**Implementation Strategy:**

To effectively implement these methods:

1. Analyze the instruction set to identify varying length instructions.

2. Determine the frequency and impact of each instruction type.

3. Choose a combination of methods that best fits the processor's design goals (performance, power efficiency, chip area, etc.).

4. Implement robust hazard detection and resolution mechanisms.

5. Use simulation and benchmarking to validate the design and optimize performance.

By carefully applying these design methods,we can create a pipelined processor that efficiently handles varying instruction lengths while minimizing structural hazards, leading to improved overall performance and resource utilization.