**Computer Architecture**

**Final Exam(8/13/2024)**

**ID- 20113 Name- Nang Thiri Wutyi**

1. .data

prompt: .asciiz "Enter an integer: "

result\_msg: .asciiz "The 2's complement of the number is: "

.text

.globl main

main:

# Print the prompt asking for the user's input

li $v0, 4 # System call for printing a string

la $a0, prompt # Load address of the prompt string

syscall

# Read an integer from the user

li $v0, 5 # System call for reading an integer

syscall

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

# Compute the 2's complement: $t1 = -$t0

sub $t1, $zero, $t0

# Print the result message

li $v0, 4 # System call for printing a string

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

syscall

# Print the 2's complement result

li $v0, 1 # System call for printing an integer

move $a0, $t1 # Move the 2's complement result to $a0

syscall

# Exit the program

li $v0, 10 # System call for exit

syscall

1. .data

prompt: .asciiz "Enter an integer: "

newline: .asciiz "\n"

.text

.globl main

main:

# Print the prompt asking for the user's input

li $v0, 4 # System call for printing a string

la $a0, prompt # Load address of the prompt string

syscall

# Read an integer from the user

li $v0, 5 # System call for reading an integer

syscall

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

# Outer loop (rows): Loop from 1 to $t0

li $t1, 1 # Initialize $t1 to 1 (row counter)

outer\_loop:

bgt $t1, $t0, end # If $t1 > $t0, exit the loop

# Inner loop (columns): Loop from 1 to $t1

li $t2, 1 # Initialize $t2 to 1 (column counter)

inner\_loop:

bgt $t2, $t1, print\_newline # If $t2 > $t1, go to newline

# Print a star ('\*')

li $v0, 11 # System call for printing a character

li $a0, '\*' # Load the character '\*' into $a0

syscall

addi $t2, $t2, 1 # Increment column counter

j inner\_loop # Repeat inner loop

print\_newline:

# Print a newline

li $v0, 4 # System call for printing a string

la $a0, newline # Load address of the newline string

syscall

addi $t1, $t1, 1 # Increment row counter

j outer\_loop # Repeat outer loop

end:

# Exit the program

li $v0, 10 # System call for exit

syscall

1. The Program Counter (PC) in a 32-bit processor keeps track of the next instruction the processor will execute. It needs to manage the flow of instructions, handle different ways of addressing memory, and respond to changes like jumps or interrupts.

### To design the Program Counter:

1. **Basic Design**:
   * **Width**: The PC is 32 bits wide, so it can point to any address in the processor’s 4GB memory.
   * **Incrementing**: Usually, the PC moves forward by 4 bytes after each instruction, as each instruction in a 32-bit system is 4 bytes long.
2. **Updating the PC**:
   * **Sequential Execution**: For most instructions, the PC simply adds 4 to point to the next instruction.
   * **Branching and Jumping**: If the instruction involves a branch or jump, the PC needs to update to the correct target address:
     + **Branch Calculation**: The PC is updated based on a condition and the offset provided in the instruction.
     + **Direct Jumps**: The PC jumps directly to a specific address.
   * **Handling Interrupts**: When an interrupt happens, the PC is saved so the processor can return to it later, and it then moves to the address where the interrupt is handled.
3. **Pipeline Considerations**:
   * **Branch Prediction**: In a pipelined processor, branch instructions can slow things down. Predicting the outcome can help keep the PC moving smoothly, but if the prediction is wrong, the pipeline must be reset, and the PC corrected.
   * **Branch Delay Slots**: Some processors execute the instruction after a branch, regardless of whether the branch is taken or not. The PC needs to account for this.
4. **Addressing Modes**:
   * The PC must handle different ways to access memory, such as:
     + **Immediate**: Adding a fixed value to the PC.
     + **PC-Relative**: Updating the PC based on its current value.
     + **Indirect**: Using a register or memory location to get the next address.
     + **Direct**: Pointing the PC directly to the target address.
5. **Security and Control Flow**:
   * **Security**: The PC should be protected against unauthorized changes to prevent attacks.
   * **Bounds Checking**: Ensure the PC doesn’t point to invalid memory areas to avoid errors or security risks.
6. **Handling Pipeline Hazards**:
   * **Stalls and Flushes**: The PC may need to pause or reset when there are issues in the pipeline, like data delays.
   * **Out-of-Order Execution**: In advanced processors, the PC should manage multiple instruction streams, ensuring they eventually execute in the correct order.
7. **Cache Considerations**:
   * **Instruction Caching**: The PC interacts with the instruction cache, which stores commonly used instructions. A good PC design should minimize cache misses by predicting and fetching instructions in advance.

In summary, designing a PC for a 32-bit processor involves making sure it accurately tracks instructions, handles different control flows, manages pipeline challenges, and maintains security. Basic functions like incrementing the PC and handling jumps are crucial, while features like branch prediction and security checks enhance the overall performance and safety of the processor.

1. Modern processors use a multi-cycle datapath architecture, even though some instructions can be executed in a single cycle, due to several key factors that enhance efficiency and performance.

Firstly, the complexity of modern instructions varies widely. While simple operations like addition might be completed in a single cycle, more complex instructions, such as multiplication, division, or memory access, require multiple steps. A multi-cycle datapath divides the execution process into several stages—fetch, decode, execute, memory access, and write-back. This division allows the processor to handle complex instructions effectively.

In terms of resource efficiency, a single-cycle architecture would require all parts of the processor, like the ALU, memory, and registers, to be active simultaneously, demanding extensive hardware that might be underutilized for simpler instructions. In contrast, a multi-cycle approach allows the reuse of hardware components across different cycles, reducing the need for excessive hardware and making the design more cost-effective.

Pipelining and parallelism are also crucial advantages of multi-cycle architectures. In pipelining, different stages of multiple instructions can overlap, significantly increasing instruction throughput. Additionally, multi-cycle designs support advanced features like superscalar execution, where multiple instructions are processed in parallel. This wouldn’t be feasible in a single-cycle design due to resource competition.

Power efficiency is another important consideration. Running everything in one cycle consumes a lot of power, as all components are powered simultaneously. In a multi-cycle design, only the necessary components are active during each cycle, leading to reduced power consumption—an essential factor for battery-powered devices.

Flexibility and scalability are enhanced with multi-cycle architectures. They support complex processor features like out-of-order execution, speculative execution, and branch prediction, which require instructions to be executed over multiple cycles. This flexibility also makes it easier to scale and integrate new features into existing designs without requiring a complete overhaul.

Memory access times also justify the use of a multi-cycle design. Accessing external memory can take multiple cycles, and a single-cycle design would either stall the processor or require a significantly longer cycle time, leading to inefficiency. The use of cache hierarchies in modern processors further complicates memory access, necessitating a multi-cycle approach to manage data movement effectively.

Data and control hazards, which occur when instructions depend on the results of previous instructions or when branch instructions alter the flow of execution, are better managed in a multi-cycle architecture. This design allows for the detection and handling of such hazards, often by inserting delays or adjusting the pipeline as needed.

Finally, multi-cycle architectures allow for performance optimization. The processor’s clock speed can be optimized for the most common scenarios, rather than the worst-case ones, allowing for faster average performance. The workload is also better balanced across cycles, leading to more even processing and improved overall efficiency.

In conclusion, while single-cycle execution might seem simpler, the benefits of multi-cycle architectures—including handling complex instructions, improving resource and power efficiency, supporting advanced features, and optimizing performance—make them the preferred choice for modern processors.

1. In a multi-cycle datapath, the control unit is responsible for generating the appropriate control signals at each step of the instruction execution. Since instructions are executed over multiple cycles, the control signals must vary depending on the current phase of the instruction (fetch, decode, execute, memory access, write-back) and the type of instruction (e.g., arithmetic, load/store, branch).

Here's how the control signals are created:

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

* **Signals**:
  + **PCWrite**: Enables writing to the Program Counter (PC) so that it can update to the next instruction address.
  + **MemRead**: Enables reading from memory to fetch the instruction.
  + **IRWrite**: Enables writing the fetched instruction into the Instruction Register (IR).
  + **ALUSrcA**: Selects the Program Counter (PC) as the input to the ALU.
  + **ALUSrcB**: Selects '4' to add to the PC for incrementing to the next instruction address.
  + **ALUOp**: Sets the ALU to perform an addition operation.
* **Purpose**: Fetch the next instruction from memory and prepare the PC for the next instruction.

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

* **Signals**:
  + **RegDst**: Specifies the destination register for the instruction.
  + **RegWrite**: Allows writing to a register.
  + **ALUSrcA**: Selects the register containing the base address for memory operations.
  + **ALUSrcB**: Selects the immediate value or offset for address calculation.
  + **ALUOp**: Determines the ALU operation (e.g., add, subtract, etc.).
* **Purpose**: Decode the instruction, read the necessary registers, and prepare for the next stage of execution.

#### **3. Execution (EX) Stage:**

* **Signals**:
  + **ALUSrcA**: Selects the appropriate input for the ALU (e.g., a register value or immediate value).
  + **ALUSrcB**: Provides the second operand to the ALU (e.g., a register value, immediate, or offset).
  + **ALUOp**: Specifies the operation to be performed by the ALU (e.g., add, subtract, etc.).
  + **Branch**: Indicates if the instruction is a branch, affecting the PC.
  + **MemRead**: Instructs the memory to read data if required (for load operations).
  + **MemWrite**: Instructs the memory to write data if required (for store operations).
* **Purpose**: Perform the actual computation or address calculation needed by the instruction.

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

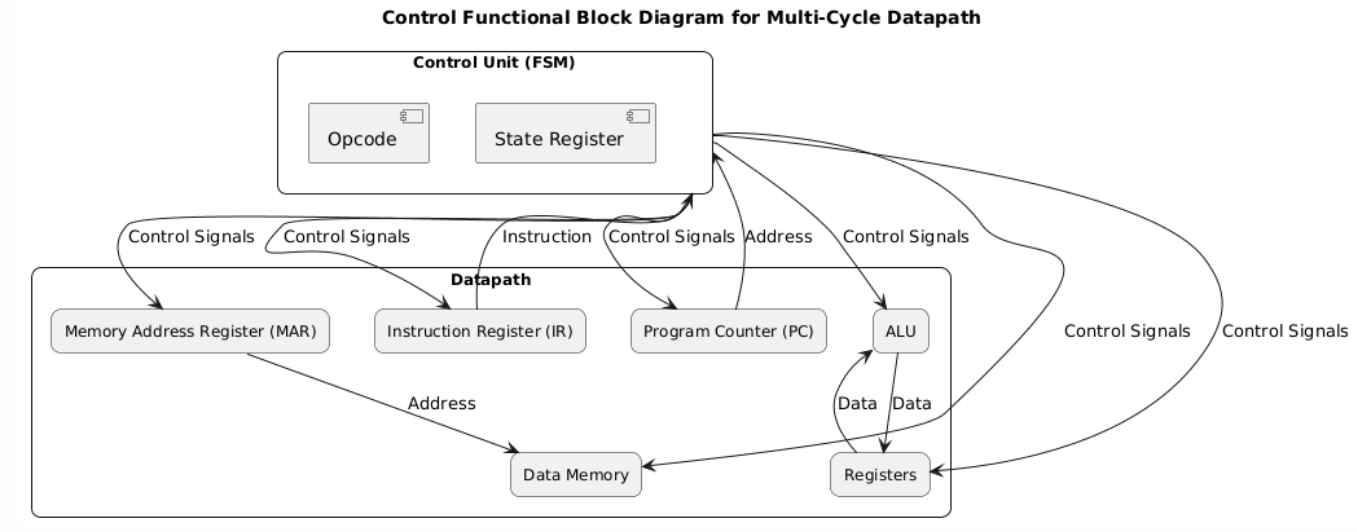
* **Signals**:
  + **MemRead**: Enables reading data from memory for load instructions.
  + **MemWrite**: Enables writing data to memory for store instructions.
  + **ALUSrcB**: Selects the address generated in the EX stage.
  + **Branch**: If the instruction is a branch, update the PC based on the branch condition.
* **Purpose**: Access memory for load/store instructions or update the PC for branch instructions.

#### **5. Write-Back (WB) Stage:**

* **Signals**:
  + **RegDst**: Selects the register to write the result to.
  + **RegWrite**: Enables writing the result back into a register.
  + **MemToReg**: Determines if the data to be written back to the register file comes from memory or the ALU.
* **Purpose**: Write the result of the instruction (either from memory or the ALU) back to the register file.

### **Creating Control Signals**

To create these control signals, the control unit typically uses a finite state machine (FSM) that changes state depending on the current phase of instruction execution and the opcode of the instruction. The FSM generates the appropriate control signals based on its current state. Each instruction type (R-type, I-type, J-type, etc.) will follow a different path through the FSM, with different control signals activated at each stage.



1. In a processor with a five-stage pipelining architecture (Fetch, Decode, Execute, Memory, and Writeback), the instruction execution latency and throughput can be calculated based on the clock speed and the number of instructions.

Instruction execution latency refers to the time it takes for a single instruction to complete all stages of the pipeline from Fetch to Writeback. In a five-stage pipeline, each stage takes one clock cycle. Thus, the latency for a single instruction to go through the entire pipeline is:

Latency = Number of stages × Clock period

Given a clock speed of 1 GHz, the clock period is,

For a five-stage pipeline:

Latency = 5 stages × 1 ns/stage = 5 ns

Instruction throughput refers to the rate at which instructions are completed and can be calculated as the reciprocal of the latency of the pipeline stage. Since the pipeline is fully utilized, a new instruction can be completed every clock cycle after the pipeline is filled.

In a fully pipelined processor, once the pipeline is filled, one instruction is completed per clock cycle. With a 1 GHz clock speed:

Throughput = 1 instruction per cycle

Since each clock cycle is 1 ns, the throughput in terms of instructions per 10 ns would be:

1. To avoid structural hazards in a pipelined processor, especially with instructions of different lengths like R-type and load instructions, several straightforward methods can be used:
2. **Resource Duplication**: By providing multiple versions of crucial hardware components, such as extra ALUs or separate memory ports, the processor can handle more simultaneous operations without conflicts. This way, one ALU can work on an R-type instruction while another handles a different instruction.
3. **Hazard Detection and Stalling**: The processor can include hazard detection units that spot when multiple instructions are trying to use the same resource at the same time. If a conflict is detected, the processor can insert delays (or "stalls") to wait until the resource is free.
4. **Instruction Scheduling**: Instructions can be reordered to avoid using the same resources simultaneously. This can be done by the compiler before the program runs or by the processor's hardware while it's running.
5. **Pipeline Buffers**: These buffers temporarily hold instructions or data that are waiting for a resource. This helps keep the pipeline moving smoothly even if some instructions are waiting for resources to become available.
6. **Load/Store Buffers**: To manage memory operations without conflicts, load and store instructions can use special buffers. These buffers queue memory requests so they don’t interfere with other instructions.
7. **Separate Paths for Instructions and Data**: Using different pathways for fetching instructions and accessing data helps prevent conflicts, as instructions and data do not compete for the same memory resources.
8. **Flexible Pipeline Stages**: Design the pipeline stages to handle varying instruction lengths without causing problems. This flexibility helps accommodate different types of instructions without running into resource issues.
9. **Dynamic Scheduling**: Techniques like Tomasulo's algorithm can dynamically reorder instructions based on current resource availability, helping to manage and reduce conflicts as instructions execute.

By using these strategies, the processor can avoid delays and maintain smooth operation even when instructions have different needs.