**Logo

Description automatically generated San Francisco Bay University**

**EE488 - Computer Architecture**

**2024 Summer Final Exam**

**Student ID: 20101 Student Name: TIKILI DAVID NIMIBOFA**

1. Write MIPS assembly program to convert a given integer in a register to 2’s complement number and print it out on the monitor.

1. Write a program in MIPS assembly to read-in an integer from the keyboard, and then print the following triangle starts.

*e.g Enter an integer: 7*

*The outputs will be:*

*\**

*\*\**

*\*\*\**

*\*\*\*\**

*\*\*\*\*\**

*\*\*\*\*\*\**

*\*\*\*\*\*\*\**

1. For the 32-bits machine, each instruction in the memory needs to access and move to the registers in the processor by the program counter (PC), describe the effective method to design PC.
2. If an instruction can be executed in the single cycle, explain in the details why all current processor architecture takes multiple cycle datapath
3. In the control functional block design of multiple cycle datapath, explain how to create all different control signals for the different types of instructions and draw a diagram.

1. Five-stages pipelining architecture in a processor consists of *Fetch, Decode, Execute, Memory*, and *Writeback* stages. If six instructions including three R-types and three I-types are executed in this system with *1* GHz, explain what the instruction execution latency and throughput (per *10*ns) are.
2. If the length varies for R-type and load instructions, describe the design methods in order to avoid structural hazards in the pipelined processor design.

**QUESTION 1**

.data

newline: .asciiz "\n"

.text

.globl main

main:

# Read integer input from the user

li $v0, 5 # Load syscall for reading an integer

syscall # Execute syscall, result stored in $v0

move $t0, $v0 # Move the input integer to $t0

# Perform 2's complement

not $t1, $t0 # $t1 = ~($t0)

addi $t1, $t1, 1 # $t1 = $t1 + 1

# Print result

li $v0, 1 # Load syscall for printing integer

move $a0, $t1 # Load 2's complement result into $a0

syscall # Execute syscall

# Print newline

li $v0, 4 # Load syscall for printing string

la $a0, newline # Load address of newline into $a0

syscall # Execute syscall

# Exit

li $v0, 10 # Load syscall for exit

syscall # Execute syscall

**QUESTION 2**

.data

newline: .asciiz "\n"

star: .asciiz "\*"

.text

.globl main

main:

# Read integer from keyboard

li $v0, 5 # Read integer syscall

syscall

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

li $t1, 1 # Initialize row counter

print\_row:

move $t2, $t1 # Initialize column counter

print\_star:

li $v0, 4 # Print string syscall

la $a0, star # Load the address of star

syscall

sub $t2, $t2, 1 # Decrement column counter

bgtz $t2, print\_star # Continue printing stars if $t2 > 0

# Print newline

li $v0, 4

la $a0, newline

syscall

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

ble $t1, $t0, print\_row # Continue to next row if $t1 <= $t0

# Exit

li $v0, 10

syscall

**QUESTION 3**

**Effective Design of the Program Counter (PC) in a 32-bit Machine**

The Program Counter (PC) is a key component in a 32-bit system, tasked with keeping track of the address for the next instruction to be executed. Below is a more detailed breakdown of the PC's efficient design:

**1**. **Core Functionality**

- Incrementing: After fetching each instruction, the PC is increased to point to the following instruction. In a 32-bit system, where each instruction is 4 bytes, the PC increments by 4.

- Handling Jumps and Branches: For jump (J) and branch (BEQ, BNE) commands, the PC needs to load a new address instead of just incrementing, allowing the processor to change the execution flow based on conditions or unconditionally (as with jump instructions).

**2**. **Integration with Pipeline**

- PC Write Control: In a pipelined architecture, the PC may need to be updated at various stages of the pipeline (e.g., during branch resolution). The control logic must determine when and with what value the PC should be updated.

- Branch Prediction: To keep the pipeline efficient, especially in deeply pipelined systems, a branch predictor might be used to anticipate the outcome of a branch instruction. If the prediction is accurate, the pipeline continues without interruption. If incorrect, the PC must be corrected, and the pipeline adjusted accordingly.

**3. Managing Exceptions and Interrupts**

- Saving and Restoring PC: When an exception or interrupt happens, the PC’s current value is usually stored in a special register (like the Exception Program Counter, EPC) so that after handling the exception, the processor can resume execution from the correct spot.

- Vectoring: The PC can be directed to a specific address (vector address) where the exception handler is located, allowing the processor to jump directly to the appropriate code for handling the exception.

**4. Support for PC-relative Addressing**

- PC-relative Instructions: Some instructions (like branches) calculate their target address based on the current PC value. The PC-relative addressing mode adds an offset to the current PC to compute the target address, facilitating more compact code and easier relocation of code segments.

**5. PC and Memory Access**

- Memory Mapping: The PC must correspond to valid instruction memory addresses, ensuring that as the PC changes due to an increment or a branch, it points to valid instructions within the program’s memory space.

- Virtual Memory: In systems that use virtual memory, the PC might deal with virtual addresses. The memory management unit (MMU) translates these virtual addresses into physical addresses during instruction fetching.

**6. Optimizing Instruction Fetch**

- Caching: To speed up instruction fetching, the PC may work closely with an instruction cache (I-cache), which stores recently accessed instructions, reducing the time required to fetch the next instruction by avoiding slower main memory accesses.

- Prefetching: Some architectures use instruction prefetching, where upcoming instructions are loaded into a buffer, making the next instruction readily available when the PC increments.

**7. PC in Multi-threaded Environments**

- Thread Context: In multi-threaded processors, each thread maintains its own PC. The processor must switch between different PCs as it alternates between threads, ensuring each thread’s execution context is correctly managed.

- Context Switching: During a context switch (e.g., in multitasking operating systems), the PC of the currently running process is saved, and the PC of the next process is loaded, enabling smooth transitions between processes.

**Example PC Design Considerations:**

- 32-bit Incrementer: A 32-bit incrementer is used to add 4 to the current PC value.

- Multiplexer for PC Update: A multiplexer (MUX) selects the source of the next PC value, which could be the incremented PC, a branch target address, a jump address, or an exception vector.

- PC Register: A 32-bit register holds the current value of the PC and is updated each clock cycle or as needed, depending on the control signals.

**QUESTION 4**

In contemporary processors, especially those using RISC (Reduced Instruction Set Computing) architectures like MIPS, instructions are generally executed across multiple cycles rather than within a single cycle. This method, known as a multi-cycle datapath, provides several advantages in terms of performance, resource efficiency, and adaptability.

**1. Why Not Single-Cycle Execution?**

In a single-cycle datapath, every instruction, regardless of complexity, must be completed within a single clock cycle. This approach has notable drawbacks:

- Cycle Time Constraint: The cycle duration would need to accommodate the slowest instruction, meaning even simple operations (like register-to-register moves) would take more time than necessary.

- Resource Inefficiency: All hardware components (ALU, memory, etc.) must be available and utilized within that one cycle, leading to either underuse (for simpler instructions) or the requirement for complex, costly hardware capable of performing multiple tasks simultaneously.

**2. Advantages of a Multi-Cycle Datapath**

A multi-cycle datapath overcomes these challenges by dividing instruction execution into distinct steps, each taking a single cycle. Here’s why this approach is beneficial:

- Optimized Cycle Duration: Each stage or step in the instruction execution process can be tailored to take exactly the time it needs, rather than the time required by the most complex instruction. This allows for a shorter overall cycle time, boosting the clock speed and enhancing performance.

- Resource Reusability: In a multi-cycle design, hardware resources like the ALU or memory can be used in different cycles. For example, the ALU might first calculate an address during the instruction decode phase and then perform an arithmetic operation in the execution phase. This reduces the need for redundant hardware, lowering the processor's cost and complexity.

- Increased Flexibility: Different instructions (e.g., arithmetic, memory access, branching) require varying sequences of operations. A multi-cycle approach can more easily accommodate these diverse requirements than a single-cycle design.

**3. Stages in a Multi-Cycle Datapath**

Instruction execution in a multi-cycle datapath is typically broken down into the following stages:

1. Instruction Fetch (IF):

- The instruction is retrieved from memory.

- The Program Counter (PC) is updated to point to the next instruction.

2. Instruction Decode (ID):

- The fetched instruction is interpreted to determine its type and necessary operands.

- The control unit generates the appropriate control signals.

- The required registers are read.

3. Execution (EX):

- The ALU carries out the needed operation (e.g., addition, subtraction, address calculation).

- For load/store instructions, the effective memory address is calculated.

4. Memory Access (MEM):

- For load instructions, data is fetched from memory.

- For store instructions, data is written to memory.

- For branch instructions, the PC is updated to the branch target address if the branch is taken.

5. Write Back (WB):

- The result of the ALU operation or the data fetched from memory is written back to the appropriate register.

These stages allow various parts of the processor to operate concurrently, maximizing the efficiency of the datapath.

**4. Control Unit and State Machine**

In a multi-cycle datapath, the control unit is crucial in managing instruction execution across multiple cycles. It does so by implementing a finite state machine (FSM) that transitions through a series of states corresponding to different instruction execution stages.

- State Transitions: The control unit moves from one state to another based on the current state, the type of instruction being executed, and the outcomes of operations (e.g., whether a branch condition is satisfied).

- Control Signals: In each state, the control unit generates signals that direct the operation of various components (e.g., ALU control, memory read/write, register write). These signals ensure the correct movement of data between registers, the ALU, and memory at the appropriate times.

**5. Example: Load Word (LW) Instruction Execution**

Let's explore how a `lw` (load word) instruction is executed in a multi-cycle datapath:

1. Instruction Fetch (IF):

- The instruction is retrieved from memory using the address in the PC.

- The PC is incremented by 4 to point to the next instruction.

2. Instruction Decode (ID):

- The instruction is decoded to recognize it as a `lw`.

- The base address and offset for the load operation are identified.

- The base register is read.

3. Execution (EX):

- The ALU computes the effective address by adding the base address from the register to the offset specified in the instruction.

4. Memory Access (MEM):

- The computed address is used to access memory, and the word at that address is read into a temporary register.

5. Write Back (WB):

- The data fetched from memory is written back to the destination register specified in the instruction.

This sequence illustrates how the multi-cycle approach enables more detailed and efficient handling of the instruction execution process.

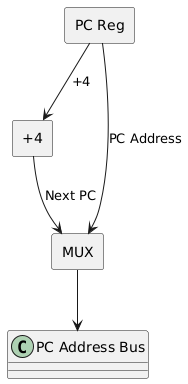
**6. Pipeline Considerations**

Although the multi-cycle datapath differs from pipelining, understanding it is essential when considering how pipelined processors operate. In a pipelined processor, different instructions can be in different execution stages simultaneously, which is facilitated by breaking down instruction execution into these discrete cycles.

**Summary**

To sum up, a multi-cycle datapath enhances efficiency and flexibility in instruction execution by dividing the process into manageable stages. This allows for optimized cycle times, resource reuse, and better accommodation of various instruction types. The control unit, through a state machine, manages these stages to ensure that instructions are executed correctly and efficiently.

**Example Diagram Simplified**

****

**QUESTION 5**

In a multi-cycle datapath, instructions are executed over multiple clock cycles, with each cycle handling a distinct step in the execution process. The control unit in a multi-cycle design is more intricate than in a single-cycle design because it must produce different control signals at various stages of instruction execution.

**Steps to Generate Control Signals**:

1. Instruction Fetch (IF):

During the first cycle, every instruction, regardless of type, is fetched from memory, and the Program Counter (PC) is updated. The control signals involved in this step include:

- PCWrite: Enables the updating of the PC.

- MemRead: Activates reading from memory.

- IRWrite: Allows the instruction register to capture the fetched instruction.

- ALUSrcA = 0: Selects the PC as the input to the ALU.

- ALUOp = Add: Increments the PC.

- PCSource = ALUResult: Uses the ALU output as the new value for the PC.

2. Instruction Decode and Register Fetch (ID):

In this cycle, the instruction is decoded, and necessary registers are fetched. The control signals are:

- ALUSrcA = 0: Continues to select the PC as the ALU input.

- ALUSrcB = 4: Uses the immediate value from the instruction as the ALU's second input.

- ALUOp = Add: Used for calculating branch target addresses, if needed.

- RegDst = x: Not relevant at this stage.

- MemRead, MemWrite: Both are disabled during this step.

3. Execution, Memory Address Calculation, or Branch Completion:

- For R-type instructions: The ALU executes the operation specified by the instruction.

- ALUSrcA = 1: Takes data from the register as the ALU's input.

- ALUSrcB = 0: Uses the output from the register file.

- ALUOp: Specifies the specific operation (e.g., addition, subtraction, AND, OR).

- For Load/Store instructions: The ALU computes the effective memory address.

- ALUSrcA = 1: Uses the register as input.

- ALUSrcB = immediate: The immediate value is used for the ALU operation.

- ALUOp = Add: Adds the immediate value to the base register.

- For Branch instructions: The ALU compares the values in the registers.

- ALUSrcA = 1: Takes the register value as input.

- ALUSrcB = 0: Utilizes the register file output.

- ALUOp = Subtract: Subtracts the register values to check for equality.

4. Memory Access or R-type Completion:

- For Load instructions: The system reads from memory.

- MemRead = 1: Activates the memory read operation.

- MemWrite = 0: No writing occurs.

- For Store instructions: The system writes to memory.

- MemRead = 0: Disables memory reading.

- MemWrite = 1: Enables the memory write operation.

- For R-type instructions: The ALU result is written back to the register file.

- RegWrite = 1: Enables writing to the register file.

- MemToReg = 0: The data comes from the ALU result.

5. Write-Back (WB):

- For Load instructions: The data fetched from memory is written back to the register file.

- RegWrite = 1: Enables writing to the register file.

- MemToReg = 1: Selects memory data for writing to the register.

Summary of Control Signals:

- ALUSrcA: Selects the first input for the ALU.

- ALUSrcB: Selects the second input for the ALU.

- ALUOp: Determines the operation to be performed by the ALU.

- PCWrite: Enables updating of the Program Counter.

- MemRead/MemWrite: Control memory read/write operations.

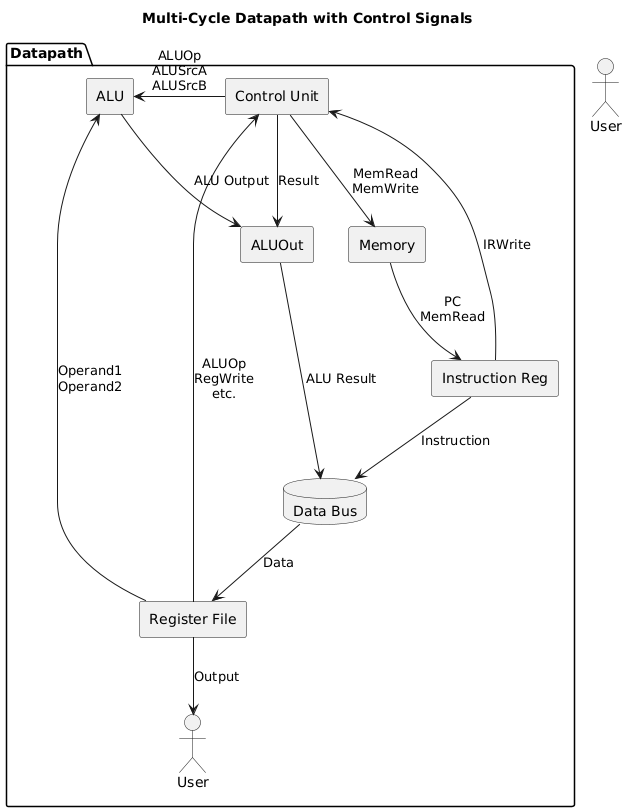
- IRWrite: Allows updating of the Instruction Register.

- RegWrite: Enables writing to the Register File.

- MemToReg: Chooses whether the data to be written to the register comes from memory or the ALU.

**Diagram of Multi-Cycle Datapath with Control Signals:**

Here's a basic conceptual diagram of a multi-cycle datapath with control signals:



**QUESTION 6**

In a five-stage pipelined architecture, where the stages are Fetch, Decode, Execute, Memory, and Writeback, each instruction sequentially moves through all five stages. The concept of pipelining in processors is crucial to understanding the difference between latency (the time it takes to complete a single instruction) and throughput (the rate at which instructions are completed).

**Instruction latency:** In a pipelined architecture, the latency for any single instruction is the time it takes to move through all five stages of the pipeline.

Since each stage takes 1 ns, the latency for any single instruction is:

Instruction Latency = 5×1 ns=5 ns

This latency is the same for both R-type and I-type instructions because all instructions go through all five stages.

**Throughput:** In a pipelined architecture, new instructions can be issued at the rate of one per clock cycle after the pipeline is full. With a 1 GHz clock, the processor completes one instruction per clock cycle after the pipeline is filled. The pipeline is filled after the first 5 cycles; after that, one instruction is completed every cycle. Over a 10 ns period (which equals 10 clock cycles at 1 GHz):

* The pipeline fills during the first 5 cycles.
* In the next 5 cycles, 5 instructions are completed.

So, the throughput is:

Throughput=5 instructions/10 ns= 0.5 instructions/ns or 5 instructions per 10ns

Therefore, Instruction Latency = 5 ns for each instruction and Throughput = 5 instructions completed per 10 ns.

**QUESTION 7**

In a pipelined processor, structural hazards arise when multiple instructions simultaneously demand the same hardware resource, leading to conflicts. This issue becomes more pronounced when the lengths of different instruction types vary, as in the case of R-type and load instructions. These varying lengths increase the potential for overlapping usage of critical hardware components like the ALU, memory, or register file. To address structural hazards in such scenarios, several design strategies can be employed:

**1. Resource Duplication**

- Concept: Duplicate the hardware resources that commonly cause conflicts.

- Example: Implement multiple ALUs if both R-type and load instructions frequently require ALU access.

- Impact: Prevents structural hazards by allowing simultaneous use of different instances of the same resource.

- Drawback: Increases the cost and complexity of the processor design.

**2. Pipeline Interleaving (Staggered Pipeline)**

- Concept: Design the pipeline stages so that different instruction types are staggered to avoid conflicts.

- Example: Align the pipeline such that the Execute stage for R-type instructions does not coincide with the Memory stage for load instructions.

- Impact: Reduces structural hazards by carefully controlling the timing of each stage.

- Drawback: Complicates the pipeline design and control logic.

**3. Operand Forwarding (Bypassing)**

- Concept: Forward data directly from one pipeline stage to another without first writing it back to the register file.

- Example: For a load instruction, immediately forward the data from the memory stage to the execution stage of the next instruction.

- Impact: Minimizes the waiting time for hardware resources, reducing the chance of hazards.

- Drawback: Adds complexity to the control unit, as it needs to manage the forwarding paths.

**4. Memory Access Time Balancing**

- Concept: Balance the timing of memory operations to ensure that memory stages do not overlap in a conflicting manner.

- Example: Use separate caches for load instructions and other memory operations to avoid conflicts.

- Impact: Minimizes memory access conflicts, reducing structural hazards.

- Drawback: Requires additional hardware, such as multiple caches, and introduces complexity in cache management.

**5. Pipeline Stall (Bubble Insertion)**

- Concept: Introduce a stall or bubble in the pipeline when a structural hazard is detected.

- Example: Delay the execution of a load instruction if it conflicts with an R-type instruction requiring the same resource.

- Impact: Simple to implement and can effectively manage hazards when they occur.

- Drawback: Stalling reduces throughput, diminishing the performance gains from pipelining.

**6. Out-of-Order Execution**

- Concept: Allow instructions to execute out of order to avoid structural hazards, ensuring correct final outcomes.

- Example: Execute an R-type instruction before a load instruction that is waiting for memory access.

- Impact: Enhances pipeline efficiency and reduces hazards.

- Drawback: Requires sophisticated hardware for instruction reordering, scheduling, and ensuring correct execution order.

7. **Varying Pipeline Depths**

- Concept: Design separate pipelines for different instruction types, each tailored to the instruction length.

- Example: A shorter pipeline for R-type instructions and a longer pipeline for load/store instructions.

- Impact: Reduces conflicts by ensuring that instructions with different lengths do not interfere with each other.

- Drawback: Increases the complexity of the control unit and pipeline management.

**Summary:**

- Resource Duplication and Pipeline Interleaving are proactive strategies that aim to avoid structural hazards through careful design.

- Operand Forwarding and Memory Access Time Balancing are reactive strategies that mitigate hazards by optimizing resource utilization and data flow.

- Pipeline Stall and Out-of-Order Execution are control-flow strategies that manage hazards when they occur, ensuring smooth pipeline operation.

- Varying Pipeline Depths offers a tailored approach, reducing conflicts by using separate pipelines, albeit at the cost of increased design complexity.

The choice of methods depends on the processor's performance goals, power consumption, cost constraints, and the specific challenges posed by varying instruction lengths.