**Logo

Description automatically generated San Francisco Bay University**

**EE488 - Computer Architecture**

**2024 Summer Final Exam**

**Student ID: 20099 Student Name: Md Maniruzzaman**

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.**

***Answer:***

.data

message: .asciiz "Enter the number: "

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

.text

.globl main

main:

li $v0, 4

la $a0, message

syscall

li $v0, 5

syscall

move $t0, $v0

not $t0, $t0

addi $t0, $t0, 1

li $v0, 4

la $a0, result\_msg

syscall

li $v0, 1

move $a0, $t0

syscall

li $v0, 10

syscall

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:***

***\****

***\*\****

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

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

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

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

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

***Answer:***

.data

prompt: .asciiz "Enter the number of rows for the triangle: "

newline: .asciiz "\n"

star: .asciiz "\*"

.text

.globl main

main:

li $v0, 4

la $a0, prompt

syscall

li $v0, 5

syscall

move $t0, $v0

li $t1, 1

row\_loop:

bgt $t1, $t0, end\_loop

li $t2, 0

star\_loop:

bge $t2, $t1, new\_row

li $v0, 4

la $a0, star

syscall

addi $t2, $t2, 1

j star\_loop

new\_row:

li $v0, 4

la $a0, newline

syscall

addi $t1, $t1, 1

j row\_loop

end\_loop:

li $v0, 10

syscall

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.**

***Answer:***

The PC is one of the most significant registers of a 32-bit machine and carries the next instruction address during its execution. There can be many issues while designing an effective Program Counter that need to be taken into consideration.

1. Incrementing Mechanism

*Auto-Increment:* The PC is usually incremented automatically on every fetch by the

processor. Hence, it will point to the next instruction in the sequence order of the program. For example, In a 32-bit machine where the length of the instruction is 4 bytes, the PC is incremented by 4 during each fetch.

1. Branching and Jumping:

*Branch Instructions:* The branch instructions are implemented by modifying the value of

PC with the branch target address. Design an adder to calculate the new value of PC as the sum of an offset and the current PC. Use a branch offset that is derived from the instruction.

*Jump Hints:* The jump operations have indicated that a PC should be considered that can

be directly loadable with a new address from the instruction to allow the processor to be working in non-sequential execution mode. Apply.

1. Interrupts and Exceptions:

*Interrupt Service Routine, ISR:* At any instance of interrupt, the PC needs to get saved so

that when the intervening interrupt was serviced the processor can get back at the right point in the program.

*Exception Handling:* In a procedure quite comparable to interrupts, on any occurrence of

an exception the PC must be saved and appropriately restored, both so when the servicing exception has been passed, and further, when the point of execution has been determined, it can get back at the right position in the program.

1. Pipelining Considerations

*Pipeline Stalls:* It makes the PC to be handled cautiously during pipeline stalls or flushes

so that the right instructions are fetched post stall or flush.

*Branch Prediction:* If the processor is doing branch prediction, the PC must

be/speculative. That is, the PC can point to a predicted target branch, and then the target can be over-written in case it is decided later that the prediction was wrong.

1. PC Relative Addressing:

*Address Calculation:* Some instructions will utilize PC-relative addressing; for an

example, the effective address is computed by adding a signed offset to the present PC. The microarchitecture should guarantee these are dealt with accurately and with as low as conceivable overhead.

1. PC Protection:

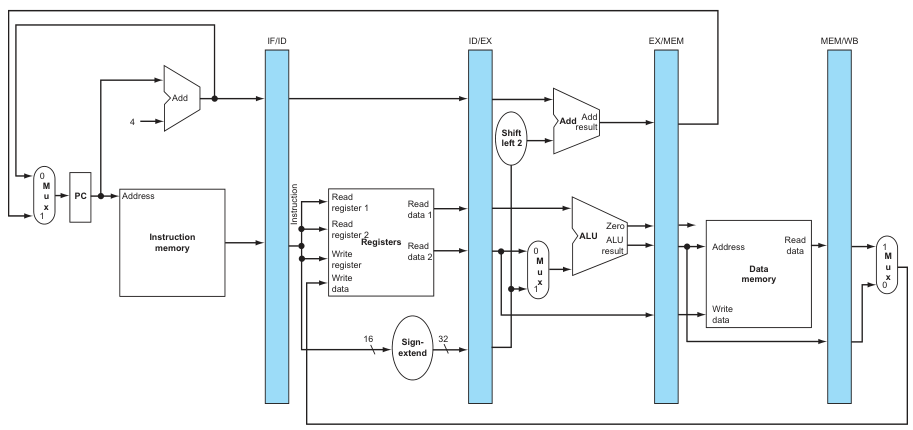
*Security:* In newer designs, even security cannot be allowed to set the PC without

authorization. To achieve this, features such as CFI don't allow setting the PC.

1. The PC in a General-Purpose Register

*Flexibility:* Some architectures use the PC himself as general-purpose register for some

instructions; thus, for such designs, the PC needs to be easily modifiable and usable without disruption of the flow of the program.



*Fig. : Example diagram of 32-bit instruction fetched from memory and the incremented 32-bit PC*

*address*

These considerations help ensure that the PC is designed efficiently to handle the requirements of a 32-bit machine, enabling effective instruction execution, control flow management, and integration with other processor components​.

1. **If an instruction can be executed in the single cycle, explain in the details why all current processor architecture takes multiple cycle datapath**

***Answer:***

Any modern processor architecture is based on a multicycle datapath rather than a single-cycle datapath. This approach has two advantages over the single-cycle datapath:

* Each functional unit (e.g., Register File, Data Memory, ALU) can be used more than once in the course of executing an instruction, which saves hardware (and, thus, reduces cost).
* Each instruction step takes one cycle, so different instructions have different execution times. In contrast, the single-cycle datapath that we designed previously required every instruction to take one cycle, so all the instructions move at the speed of the slowest.

A diagram of a computer data register

Description automatically generated

*Fig. : Example diagram of Simple multicycle datapath with buffering registers (Instruction register, Memory data register, A, B, and ALUout)*

All current processor architecture takes multiple cycle datapath and few reasons are given below:

1. Complexity of Modern Instructions:

*Variety of Instructions:* Modern processors can be designed to support a great variety of

instructions with respect to their complexity. While simple instructions, for instance, register-to-register moves, could be finished within a cycle, the more complex ones require several stages - like memory access, floating-point operations, or division - for example, fetch, decode, execute, memory access, and write-back.

*Different Cycle Requirements*: Some single-cycle datapath would compel all instructions

to retire in one clock cycle, regardless of their natural complexity. The implication is that the length of the clock cycle would be forced to be the length of the slowest instruction. Due to this, simple instructions would lose performance unnecessarily.

1. Optimizing Clock Speed:

*Faster Clock Cycles:* In a multi-cycle datapath, the length of the clock cycle need not be

that of the slowest stage of instruction execution, rather dictated by the fastest stage of instruction execution, hence allowing higher rates of processor's clock since that the critical path is shorter.

*Efficient Use of Resources:* In multi-cycle designs, functional units are reused for

different instruction execution stages and thus utilize processor resources effectively.

1. Pipeline Implementation:

*Instruction Pipelining:* This is the concept that modern processors implement to overlap

more than one instruction in the execution process at different stages. The multi-cycle datapath has a better predisposition towards pipelining since each pipeline stage corresponds to a different cycle.

*Improved Throughput:* Pipelining improved the throughput of instructions; it was now

possible to start a new instruction every clock cycle, which would not have been possible using a single cycle datapath in which every single instruction would occupy the entire datapath for one cycle.

1. Memory Access Delays:

*Memory Latency:* Accessing memory can often be the most time-consuming operation

when executing an instruction. If the processor were single-cycle, then the clock cycle would have to be long enough to permit access to memory, which would greatly reduce the overall speed of the processor.

*Separate Stages for Memory:* By using multi-cycle design, memory access can be moved

into its own stage, enabling the remaining parts of the datapath to be clocked more quickly.

1. Scalability and Flexibility:

*Evolving Instruction Sets:* Since the fact that instruction sets will continue to evolve and

hold more and more complex instructions, then a multi-cycle datapath gives flexibility without having the need for redesigning the whole architecture for new instructions.

*Design Scalability:* Multi-cycle architectures scale better and can exploit new techniques

like speculative execution and branch prediction - that continue to improve performance significantly.

1. Energy Efficiency:

*Power Consumption:* That is, single-cycle designs will tend to use more power than their

multi-cycle equivalents, with all components of the processor awake every cycle. Whereas in multi-cycle designs, all parts of the processor are only activated in some cycles, hence reducing the overall power consumption.

Although a single-cycle datapath can, theoretically, execute an instruction in a single clock cycle, practical issues relating to instruction complexity, clock speed, memory latency, and overall efficiency have mostly pushed modern processors toward using multi-cycle datapaths. Multi-cycle datapaths offer much better resource efficiency, far higher achievable clock speeds, and far better support for advanced features like pipelining - features quite essential to high performance in modern computing environments.

1. **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.**

***Answer:***

Given the datapath illustrated, instruction execution in each cycle of the datapath. The implementational goal is balancing of the work performed per clock cycle, to minimize the average time per cycle across all instructions. For example, each step would contain one of the following:

* ALU operation
* Register file access (two reads or one write)
* Memory access (one read or one write)

Thus, the cycle time will be equal to the maximum time required for any of the preceding operations.

A diagram of a computer

Description automatically generated

*Fig.: Example diagram of the complete datapath for the multicycle implementation together with the necessary control lines.*

A table with text and images

Description automatically generated

*Table: The effect of each of the seven control signals.*

A screenshot of a computer

Description automatically generated

*Table: The action caused by the setting of each control signal*

Details are given below:

1. Control Unit Overview

The control unit in a multiple-cycle datapath is responsible for generating the correct control signals at each step of the instruction execution. This control logic is typically implemented using a finite-state machine (FSM), which dictates the sequence of operations based on the current state of the machine and the instruction being executed.

1. Finite-State Machine (FSM) for Control

The FSM used in control logic operates as follows:

*States:* Each state corresponds to a stage in the instruction execution process (e.g.,

instruction fetch, decode, execute, memory access, and write-back).

*Next State Logic:* Based on the current state and the instruction opcode, the FSM

transitions to the next state.

*Output Logic:* Depending on the current state, the FSM outputs control signals

that dictate the operations performed in that cycle (e.g., which registers to read, which ALU operation to perform).

1. Control Signals

The main control signals that need to be generated include:

*RegDst:* Determines whether the destination register is specified by the rd or rt

field.

*ALUSrc:* Selects the second input to the ALU (either a register value or an

immediate value).

*MemtoReg:* Determines whether the value to be written to a register comes from

the ALU result or memory.

*RegWrite:* Enables writing to a register.

*MemRead/Write:* Enables reading from or writing to memory.

*PCSrc:* Controls the source of the next value for the Program Counter (PC) (e.g.,

sequential address or branch target).

1. Implementing the Control Logic

To implement this in a multiple-cycle design:

*Instruction Fetch (IF) Stage:* The control signals for this stage are straightforward,

as they typically involve fetching the instruction from memory and incrementing the PC.

*Instruction Decode (ID) Stage:* During this stage, the opcode is used to determine

the type of instruction and to set up the appropriate control signals for the upcoming execution stages.

*Execution (EX) Stage:* The control unit sets the ALU control signals based on the

instruction type (e.g., add, subtract, load, store).

*Memory Access (MEM) Stage:* If the instruction involves memory access, the

control signals for memory read or write are asserted.

*Write-Back (WB) Stage:* The control unit enables the register file to write back the

result of the ALU or memory operation.

In the multicycle datapath, all operations within a clock cycle occur in parallel, but successive steps within a given instruction operate sequentially. Several implementational issues present that do not confound this view, but should be discussed. One must distinguish between (a) reading/writing the PC or one of the buffer registers, and (b) reads/writes to the register file.

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.**

***Answer:***

According to the question,

Clock Speed = 1 GHz (1 ns per clock cycle)

Pipelining basics

Pipeline Stages = 5

1. Fetch : Instruction is fetched from memory.
2. Decode : Instruction is decoded, and necessary registers are read.
3. Execute : The operation is executed.
4. Memory : Data memory is accessed if needed.
5. Writeback : The result is written back to the register file.

Each stage takes one clock cycle to complete.

Instructions: 6 instructions (3 R-type, 3 I-type)

Pipeline Execution Overview

The pipeline is filled sequentially as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cycle** | **Fetch** | **Decode** | **Execute** | **Memory** | **writeback** |
| 1 | 1 |  |  |  |  |
| 2 | 2 | 1 |  |  |  |
| 3 | 3 | 2 | 1 |  |  |
| 4 | 4 | 3 | 2 | 1 |  |
| 5 | 5 | 4 | 3 | 2 | 1 |
| 6 | 6 | 5 | 4 | 3 | 2 |
| 7 |  | 6 | 5 | 4 | 3 |
| 8 |  |  | 6 | 5 | 4 |
| 9 |  |  |  | 6 | 5 |
| 10 |  |  |  |  | 6 |

Instruction Execution Latency:

Here, Latency is defined as the time taken for a single instruction to pass through all stages of the pipeline.

Now, Instruction 1:

It enters the pipeline in cycle 1 and finishes at the end of cycle 5.

Latency for Instruction 1 = 5 cycles × 1 ns = 5 ns

Instruction Throughput:

* Throughput is the number of instructions completed per unit of time.
* In a pipelined architecture, after the pipeline is filled, a new instruction completes every clock cycle.
* Let's calculate the throughput over a 10 ns period:

Pipeline Execution:

* *Cycle 1 :* First instruction (Instr 1) enters Fetch stage.
* *Cycle 2 :* Instr 1 moves to Decode, Instr 2 enters Fetch.
* *Cycle 3 :* Instr 1 moves to Execute, Instr 2 to Decode, Instr 3 enters Fetch.
* *Cycle 4 :* Instr 1 moves to Memory, Instr 2 to Execute, Instr 3 to Decode, Instr 4 enters

Fetch.

* *Cycle 5 :* Instr 1 moves to Writeback, Instr 2 to Memory, Instr 3 to Execute, Instr 4 to

Decode, Instr 5 enters Fetch.

* *Cycle 6 :* Instr 1 completes, Instr 2 moves to Writeback, Instr 3 to Memory, Instr 4 to

Execute, Instr 5 to Decode, Instr 6 enters Fetch.

* *Cycle 7 :* Instr 2 completes, Instr 3 moves to Writeback, Instr 4 to Memory, Instr 5 to

Execute, Instr 6 to Decode.

* *Cycle 8 :* Instr 3 completes, Instr 4 moves to Writeback, Instr 5 to Memory, Instr 6 to

Execute.

* *Cycle 9 :* Instr 4 completes, Instr 5 moves to Writeback, Instr 6 to Memory.
* *Cycle 10:* Instr 5 completes, Instr 6 moves to Writeback.
* *Cycle 11:* Instr 6 completes.

Calculation:

For, the sixth instruction (Instruction 6):

It enters the pipeline in cycle 6 and finishes at the end of cycle 10.

Latency for Instruction 6 = 5 cycles × 1 ns = 5 ns

Thus, the latency for each instruction is 5 ns.

Pipeline Throughput, Throughput refers to the number of instructions completed per unit time.

After the pipeline is fully filled by the 5th cycle, one instruction is completed per cycle.

By the end of the 10th cycle, 5 instructions have been completed.

Total time considered: 10 ns

Number of instructions completed in 10 ns: 5 instructions

Throughput Calculation:

Throughput = (Number of Instructions Completed / Total Time)

= 5 instructions / 10 ns

= 0.5 instructions/ns

Therefore,

Instruction Latency: Each instruction has a latency of 5 ns.

Instruction Throughput: 5 instructions are completed in 10 ns, resulting in a throughput of 0.5 instructions/ns.

This means that in a 10 ns period, the pipeline completes 5 instructions after the initial pipeline fill.

1. **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.**

***Answer:***

Structural hazards arise when various instructions in the pipeline want the same hardware resource simultaneously. With instructions of variable length such as R-type and load instructions, the individual should be able to design a mechanism that will allow the processing of the various types of instructions without competing for hardware resources. The following is how you can design methods to avoid structural hazards:

1. Resource Duplication

*Duplicate Resources:* One obvious way to eliminate structural hazards is to duplicate the

resources that may lead to conflicts. For example, one may have separate instances for those resources that have to be accessed by both the R-type and load instructions in the same cycle, thus leading to independent ALUs, register file ports, separate memory access paths, etc.

*Multiple Functional Units:* For instance, you can have one functional unit that executes R-type

instructions and another that performs load/store instructions. This would allow both types of instructions to be dispatched to different functional units in the same cycle without leading to a resource conflict.

1. Pipeline Interleaving

*Interleaved pipeline stages:* By this method, you can stagger the stages of the pipeline in a way

that enforces different types of instructions to use the same resource in different cycles. For example, if an R-type instruction uses the ALU in the EX stage, and a load instruction accesses memory in the MEM stage, design the pipeline so that these stages do not overlap in such a way as to cause conflicts.

*Staggered Instruction Execution:* The processor could be designed not to execute two

instructions using the same resource within a certain number of cycles of each other, so it would greatly reduce the possibility of a structural hazard.

*Pipelined Functional Units:* Another way is to actually pipeline the functional units. For

example, if the memory unit is pipelined, it will reduce the likelihood of conflicts, where multiple memory operations are in different stages of execution at a time.

*Deep Pipelining:* Making certain resources' pipelines deep, such as memory, would allow

multiple instructions to use those without waiting for each other; in principle, this would avoid structural hazards.

c. Dynamic Scheduling (Out-of-Order Execution)

*Out-of-Order Execution:* Refers to the characteristic of changing the order of instructions by

the processor dynamically. If, for example, a load instruction is going to cause a structural hazard involving memory access, it can reorder execution and execute those instructions which are completed without causing any hazards due to them.

*Reservation Stations:* Along with dynamic scheduling, reservation stations carry instructions that

are waiting on resources to free up. After the appropriate resources are freed up, the instruction is executed without structural hazards.

1. Instruction Queuing and Buffering

*Instruction Buffers:* In your pipeline design, if an instruction is not allowed to proceed due to a

possible resource conflict, it is possible to incorporate buffers for instructions that can hold them temporarily in the pipeline. It is a form of queueing instructions and finally releasing them when the desired resources are available.

*Load/Store Buffers:* It may be worthwhile to provide special buffers for the load and store

instructions for better management of the memory access such that other instructions can continue in the pipeline without waiting for the access to memory to be completed.

1. Detection of Hazards and Mechanisms for Stalling

*Hazard Detection Units:* A hazard detection unit can be added to the pipeline control logic to

detect when a structural hazard is bound to happen. When detected, the pipeline control can insert stalls or bubbles to delay the instructions causing the hazard until the resource becomes available.

*Pipeline Stalling:* The processor can inject stalls (no-operations or NOPs) into the pipeline to

delay execution of instructions that would otherwise cause structural hazards. This will ensure that each instruction gets all the resources it needs without colliding with others.

1. Cache Memory Being Used

*Separate Instruction and Data Caches:* Having different caches for instructions and for data will

make conflicts between instruction fetch and data loads or stores impossible leading to elimination of all the structural hazards associated with access to memory.

*Multiported Caches:* Creation of multiported caches can allow multiple memory operations to

proceed without a conflict; this solution is especially useful when load/store instructions occur quite frequently in memory.

Structural hazards in pipelines with multiple instruction lengths can be managed through careful architectural design, such as resource duplication, pipeline interleaving, and advanced techniques like out-of-order execution. These methods ensure that different instruction types can be processed efficiently without causing delays due to resource contention.