CSEN 210: COMPUTER ARCHITECTURE

PROJECT REPORT

Counting Occurrences of a Value in an Unsorted Array

Team Members

VENKAT SAI SESHA REDDY DUGASANI & SHIVA KUMAR REDDY RANGAPURAM

CONTENTS

CHAPTER 1

INTRODUCTION

In today's tech landscape, we're surrounded by an array of powerful programming languages. However, at their core, computers comprehend only one language: machine code. Thus, every piece of code we write must ultimately be translated into machine code instructions. Even a seemingly straightforward task in a high-level language like C necessitates a series of intricate commands in machine code, such as ARM or MIPS, to execute on the hardware.

The multi-cycle datapath, functioning as a pipeline, facilitates the concurrent execution of numerous instructions. Within each clock cycle, multiple instructions advance through different stages within the datapath. Over the years, numerous endeavors have aimed to refine pipeline performance and accelerate processing. These improvements entail adjustments to pipeline stages, control signals, and hardware components, each tailored to optimize efficiency.

In our current endeavor, we're crafting an Instruction Set Architecture (ISA) and developing a tailored pipeline solution for efficiently counting occurrences of a value in an unsorted array.

CHAPTER 2

OBJECTIVE

The aim of this project is to define an Instruction Set Architecture (ISA) and a pipeline implementation of this architecture to perform the following operation:

INPUT:

An unsorted array A = [a1, a2, ….an], size of the array n, a target value ‘v’

OUTPUT:

How many items in the array are equal to v?

CHAPTER 3

INSTRUCTION SET ARCHITECTURE (ISA)

3.1 ASSUMPTIONS IN THE DESIGN:

The general assumptions are as follows:

- Word length is exactly four bytes.
- 64 registers are available in the processor, each 32 bits (2's complement).
- Memory addresses are 32 bits.
- Memory access and arithmetic operations take 2 ns each.
- Unused bits in the instruction format are assumed to be set to zero.

Assumptions specific to the design of ISA:

- The status register, containing flags set after a comparison operation, is integrated into the ALU.
- The Control Unit works with the CMP instruction and Branch instruction directly based on the status register value.
- All initial values required by the program are directly loaded from predetermined memory addresses into registers at the start of program execution.
- The result of the computation is stored directly into a dedicated register (e.g., R4) instead of using a separate store instruction to write it back to memory.
- Array access is performed using a register-based approach, where the array index is stored in a dedicated register. This approach is compatible

with pipelining and facilitates efficient memory access scheduling and pipeline management.

3.2 INSTRUCTION SET:

What is an Instruction Set Architecture (ISA)?

Instruction Set Architecture (ISA) serves as a hardware-software interface defining the operations a processor can execute, facilitating communication between software programs and the underlying hardware. It encompasses the set of instructions that software developers can use to write programs and the hardware mechanisms that execute those instructions.

Instructions in our Instruction Set Architecture (ISA) are **32-bit** each. The ISA includes the following instruction formats/types:

3.3 INSTRUCTION FORMATS:

Load RD, RS:

1. Opcode**:** 6 Bits.
2. Register Source Operand 1[**RS**]: 6 Bits.
3. Register Destination [**RD**]: 6 Bits.
4. Unused: 14 bits (may be used for future functionalities)

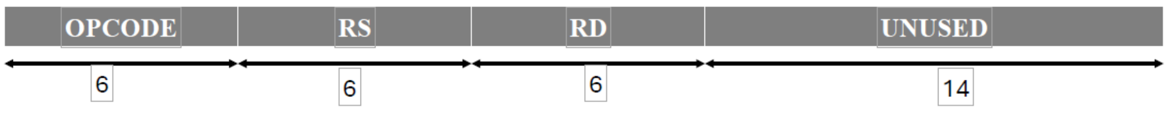

Figure: Load Instruction Format

Compare RA, RB:

1. Opcode**:** 6 Bits.
2. Register Source Operand [**RA**]: 6 Bits.
3. Register Source Operand [**RB**]: 6 Bits.
4. Unused**:** 14 Bits.

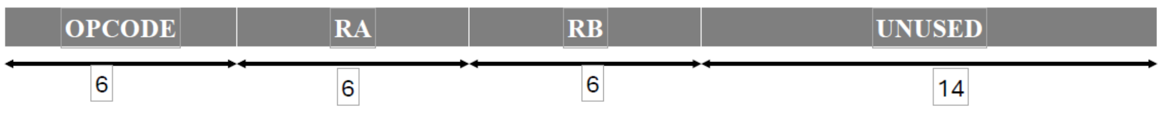

Figure: Compare Instruction Format

ADDI RD, RS, Immediate:

1. Opcode**:** 6 Bits.
2. Register Source Operand 1[**RS**]: 6 Bits.
3. Register Destination [**RD**]: 6 Bits.
4. Immediate: 14 bits

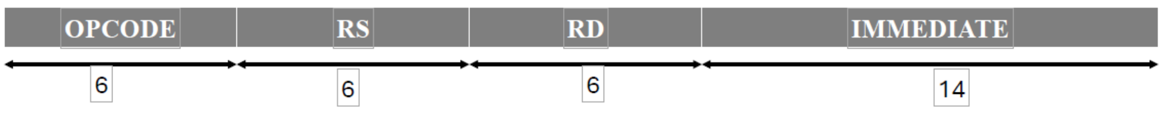

Figure: ADDI Instruction Format

Jump Offset:

1. Opcode**:** 6 bits.
2. Offset: 26 bits

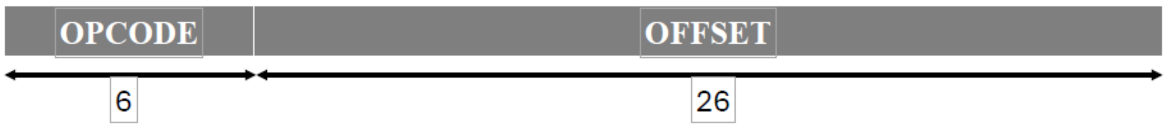

Figure: Jump Offset Format

Branch Offset:

1. Opcode**:** 6 bits.
2. Offset: 26 bits

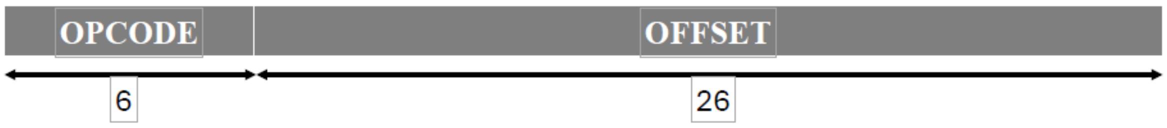

Figure: Branch Offset Format

CHAPTER 4

ASSEMBLY CODE

INITIALIZATION:

```
    LOAD R1, A     ; Load the base address of the array into R1
    LOAD R2, n     ; Load the size of the array into R2
    LOAD R3, v     ; Load the value v into R3
    LOAD R4, #0    ; Initialize the count of matching elements to 0
    LOAD R5, #0    ; Initialize an index i to 0 (to iterate over the array)
```

LOOP:

```
    CMP R5, R2          ; Compare i with n and set status flags
    BEQ END             ; If previous comparison was equal, jump to END
    LOAD R6, [R1+R5]    ; Load A[i] into R6
    CMP R6, R3          ; Compare A[i] with v and set status flags
    BEQ MATCH           ; If previous comparison was equal, jump to MATCH
    ADDI R5, R5, #1     ; Increment i using ADDI
    JMP LOOP            ; Jump back to the start of the loop
```

MATCH:

```
    ADDI R4, R4, #1     ; Increment the count because A[i] == v
    ADDI R5, R5, #1     ; Increment i using ADDI (continue to the next iteration)
    JMP LOOP            ; Jump back to the start of the loop
```

END:

; Result is now in register R4

; Continue with other operations using the result in R4

4.1 Breakdown of Instructions:

Below is the breakdown focusing on the essential control signals for each type of instruction in the provided assembly code:

LOAD Instruction (e.g., LOAD R1, A):

IF (Instruction Fetch): Fetch instruction from memory.

ID (Instruction Decode): Decode as LOAD.

EX (Execute): Calculate memory address if needed (for immediate values, this might be just a pass-through).

MEM (Memory Access): Read from memory.

WB (Write Back): Write data to the specified register.

CMP Instruction (e.g., CMP R5, R2):

IF: Fetch instruction.

ID: Decode as CMP.

EX: Perform subtraction (R5 - R2), set flags based on result, no result written to register.

MEM: Not used.

WB: Not used.

BEQ/BNE Instruction (e.g., BEQ END)

IF: Fetch instruction.

ID: Decode as BEQ/BNE.

EX: Check condition flags; if condition matches, calculate branch target address.

MEM: Not used directly, but may involve flushing the pipeline if a branch is taken.

WB: Not used.

ADDI Instruction (e.g., ADDI R5, R5, #1):

IF: Fetch instruction.

ID: Decode as ADDI.

EX: Perform addition with immediate value.

MEM: Not used.

WB: Write the result to the target register.

JMP Instruction (e.g., JMP LOOP)

IF: Fetch instruction.

ID: Decode as JMP.

EX: Calculate jump target address.

MEM: Not used, but may involve flushing the pipeline.

WB: Not used.

General Pattern:

LOAD: Fetch -> Decode -> Execute (Address Calculation) -> Memory Access -> Write Back.

CMP: Fetch -> Decode -> Execute (Set Flags) -> (Skip Memory Access and Write Back).

Branch (BEQ/BNE): Fetch -> Decode -> Execute (Check Flags and Calculate Target) -> (Conditional Pipeline Flush).

Arithmetic Immediate (ADDI): Fetch -> Decode -> Execute (Arithmetic Operation) -> Write Back.

Jump (JMP): Fetch -> Decode -> Execute (Calculate Target) -> (Potential Pipeline Flush).

CHAPTER 5

DATAPATH AND CONTROL

PIPELINE:

In our setup, we've implemented a pipeline for our datapath. This pipeline enables the concurrent execution of multiple instructions. It consists of a total of 5 stages, outlined as follows:

IF: Instruction fetch

ID: Instruction decode and register fetch

EX: Execution and effective address calculation

MEM: Accessing the data memory

WB: Write back to the register.

COMPONENTS:

In our datapath design, we have added **two** new components to assist in achieving the execution of the operation. The components are explained as follows:

1. **Compare Unit:**
   This component is integrated into ALU to perform comparison operations specific to the ISA . The compare unit takes two values as input and tests them for equality. Suppose it takes two values a and b, it works and provides the output as follows:

   Sets the status register to 1, if a == b

   Sets the status register to 0, otherwise.

2. **Status Register:**

   The status register is used and works with the compare unit , based on the result of the compare unit it gets updated and the status register's signal is directly related to branch operation, as branch works based on the result of the compare unit.

DATAPATH:

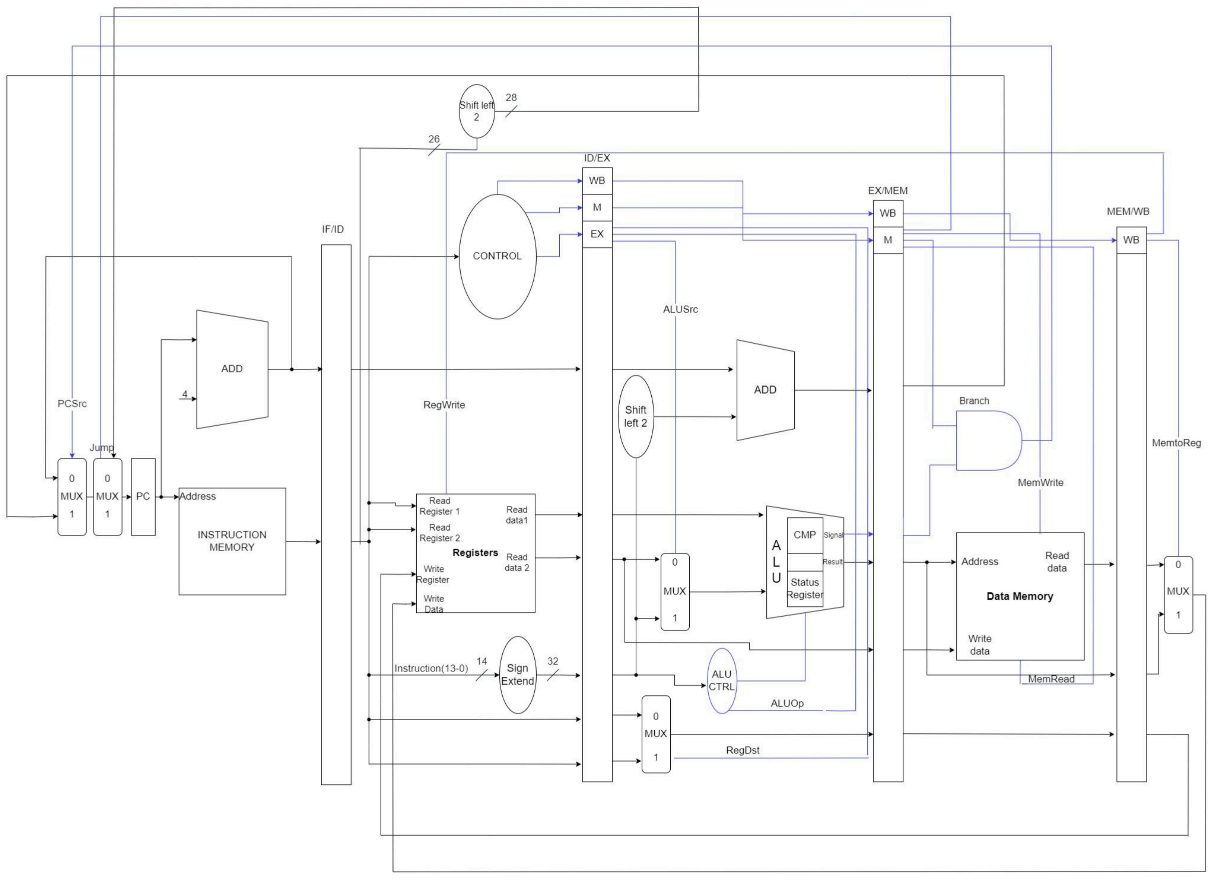

Figure: Datapath implementing the operation

Control Signals:

For LOAD R1, A (and similarly for other LOAD instructions):
RegDst: Not relevant, since the destination register is directly specified by the instruction.
ALUSrc: Set to 1 because the instruction uses an immediate value or address as one of the operands for the ALU, to calculate the effective memory address.
MemtoReg: Set to 1, indicating that the data to be written back to the register comes from memory, not from the ALU result.
RegWrite: Enabled, allowing the loaded value from memory to be written into the register.
MemRead: Enabled, as data is being read from memory.
MemWrite: Disabled, since this operation reads from, rather than writes to, memory.
Branch: Disabled, as this instruction does not involve a branch.
ALUOp: Configured for address calculation, typically addition.
PCSrc: Set to 0, meaning the next instruction address comes from the sequential flow, not a branch or jump address.
For COMPARE R5, R2:
RegDst: Not relevant, as no data is being written to a destination register.
ALUSrc: Set to 0, indicating that the ALU operation uses values from registers (R5 and R2 in this case).
MemtoReg: Not relevant, since there's no data being written to a register.
RegWrite: Disabled, as this operation does not result in a register write.
MemRead: Disabled, as no memory access is required.
MemWrite: Disabled, as no memory access is required.
Branch: Not directly relevant to this operation itself, but the result affects subsequent branch decisions.

ALUOp: Configured for subtraction, as comparison in many architectures is performed via subtraction.
PCSrc: Set to 0, since this isn't a branch instruction; it sets the stage for a possible branch based on flags set by the ALU operation.

For BRANCH END:

RegDst, ALUSrc, MemtoReg: Not relevant, as this operation involves PC manipulation rather than ALU operations or data movement to/from registers.
RegWrite: Disabled, as there's no register write operation here.
MemRead: Disabled, no memory access is involved.
MemWrite: Disabled, no memory write operation is involved.
Branch: Enabled, indicating that this instruction potentially alters the sequential flow based on a condition.
ALUOp: Though not directly performing an ALU operation, it's set based on the previous CMP instruction for condition checking.
PCSrc: Set to 1 if the branch condition is met, instructing the PC to load the branch address.

For ADDI R5, R5, #1:

RegDst: Not relevant, as the destination register is specified by the instruction itself.
ALUSrc: Set to 1, as the instruction uses an immediate value as one of the operands.
MemtoReg: Set to 0, indicating that the ALU's result is written back to the register.
RegWrite: Enabled, allowing the ALU result to be written into the register.
MemRead: Disabled, as no memory read operation is involved.
MemWrite: Disabled, as no memory write operation is involved.
Branch: Disabled, this is an arithmetic operation, not a branch.
ALUOp: Configured for addition, as this is an add immediate type of instruction.

PCSrc: Set to 0, continuing with the next sequential instruction.

For JMP LOOP (and similar for any jump):

RegDst, ALUSrc, MemtoReg: Not relevant, as these concern data paths not used by a jump operation.

RegWrite: Disabled, as there's no register write operation in a jump.

MemRead: Disabled, as no memory read operation is involved.

MemWrite: Disabled, as no memory write operation is involved.

Branch: Enabled, although it's an unconditional jump rather than a conditional branch.

ALUOp: Not relevant, since the jump does not use the ALU for computation.

PCSrc: Set to 1, indicating that the PC should be updated to the jump address rather than the next sequential instruction.

CHAPTER 6

EMULATION OF MIPS INSTRUCTIONS

We've devised equivalent instructions in our ISA for select MIPS instructions. The table below outlines these equivalents for the specified MIPS instructions:

MIPS Instruction	Assembly Operation
`la $t1, A`	LOAD R1, A
`lw $t2, n`	LOAD R2, n
`lw $t3, v`	LOAD R3, v
`li $t4, 0`	LOAD R4, #0
`li $t5, 0`	LOAD R5, #0
`slt $t6, $t5, $t2`	CMP R5, R2
`beq $t6, $zero, end`	BEQ END
`lw $t7, 0($t1)`	LOAD R6, [R1+R5]
`bne $t7, $t3, else`	CMP R6, R3
`addi $t4, $t4, 1`	ADDI R4, R4, #1
`addi $t5, $t5, 1`	ADDI R5, R5, #1
`j loop`	JMP LOOP
`else:`	-
`j loop`	JMP LOOP
`end:`	END

CHAPTER 7

Performance Analysis: Enhanced Architecture vs. Traditional MIPS

Key Enhancements

Our design implements a focused instruction set and optimizes memory interactions, leading to superior performance over traditional MIPS. A critical improvement is the use of a dedicated register for storing intermediate results, minimizing memory STORE operations.

Performance Metrics

- Instruction Efficiency: Our architecture requires (5 + 5n) instructions, where n represents loop iterations. This efficiency stems from eliminating the need for extra STORE instructions by leveraging a dedicated result register.
- Execution Time: Calculated as (Instruction Count) × (CPI) × (Cycle Time), our model boasts an execution time of (5 + 5n) nanoseconds, assuming a CPI of 1 and a Cycle Time of 1ns.

Comparison with Traditional MIPS

In a standard MIPS setup, lacking a dedicated result register might necessitate an additional STORE operation per iteration, potentially increasing the instruction count to (5 + 7n) for similar tasks. This increase directly impacts execution time and resource utilization.

Conclusion

Our architecture demonstrates a clear advantage in reducing instruction count and execution time by optimizing the use of registers and minimizing memory operations. This design choice not only streamlines processing but also enhances overall system efficiency compared to a traditional MIPS approach, making it a superior solution for tasks requiring intensive data manipulation and evaluation.