Computer Organization and Assembly Language

Lab Manual

Name: Momina Akhter

Reg no: 468302

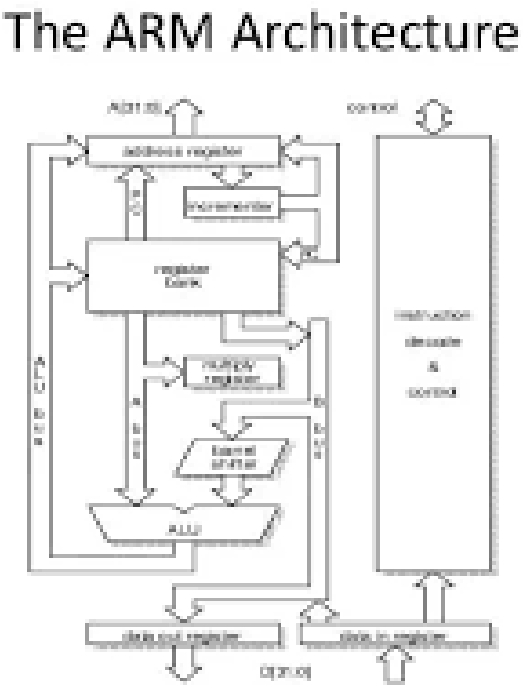
Program: ADP-CS

Semester: Third-Fall23

Understanding ARM Architecture

1.search and draw the ARM processor architecture, labeling its components.

1. CPU (Central Processing Unit): The heart of the processor, typically consisting of multiple cores. It executes instructions from programs.
2. ALU (Arithmetic Logic Unit): Performs arithmetic and logical operations like addition, subtraction, and logical comparisons.
3. Control Unit: Directs the operation of the processor by interpreting and executing instructions, managing the flow of data.
4. Register File: A small set of high-speed storage locations that store data temporarily during execution.
5. Cache (L1, L2): Memory caches (L1, L2) are used for storing frequently accessed data to speed up operations.
6. Data and Control Buses: These buses transfer data and control signals between various components of the processor and memory.
7. Peripheral Interfaces: These handle communication with external components like memory, input/output devices, and other.



Basic Assembly Instructions:

1. Write a program to load and store data using LDR and STR.

```
.global _start

.section .data
    value:  .word 42       // Define a 32-bit value with the number 42
    result: .word 0        // Allocate space for storing the result

.section .text
_start:
    // Load the value from memory into register R0
    LDR R0, =value         // Load the address of 'value' into R0
    LDR R1, [R0]           // Load the value at the address in R0 into R1

    // Store the value from R1 into 'result'
    LDR R0, =result        // Load the address of 'result' into R0
    STR R1, [R0]           // Store the value in R1 into the memory address in R0

    // Exit the program (This part depends on your environment, e.g., Linux)
    MOV R0, #0             // Return code 0
    MOV R7, #1             // syscall number for exit
   SWI 0                 // Software interrupt to exit
```

2. Perform basic arithmetic operations (addition, subtraction) using ADD, SUB.

```
ADD R1, R2, R3
SUB R1, R2, R3

LOAD R2, 5      ; Load value 5 into R2
LOAD R3, 3      ; Load value 3 into R3

ADD R1, R2, R3   ; R1 = 5 + 3 = 8
SUB R4, R2, R3   ; R4 = 5 - 3 = 2
```

Conditional Execution:

1. Write a program to compare two numbers and output the larger number.

```
LOAD R1, 5        ; Load the first number (5) into register R1
LOAD R2, 8        ; Load the second number (8) into register R2

CMP R1, R2        ; Compare the values in R1 and R2
BGT LARGER        ; If R1 > R2, branch to LARGER

; If we are here, it means R2 >= R1
MOV R3, R2        ; Move the value of R2 to R3 (R3 will hold the larger value)
JMP END           ; Jump to the end of the program
```

```
LARGER:
MOV R3, R1          ; Move the value of R1 to R3 (R3 will hold the larger value)

END:
; At this point, R3 holds the larger number
```

2. Implement a conditional block using CMP, BEQ, BNE:

```
LOAD R1, 5          ; Load the first number (5) into register R1
LOAD R2, 5          ; Load the second number (5) into register R2

CMP R1, R2          ; Compare the values in R1 and R2

BEQ EQUAL           ; If R1 == R2, branch to EQUAL block
BNE NOTEQUAL        ; If R1 != R2, branch to NOTEQUAL block

EQUAL:
; Code to execute if numbers are equal
MOV R3, 1           ; Store 1 in R3 to indicate equality
JMP END             ; Jump to the end of the program

NOTEQUAL:
; Code to execute if numbers are not equal
MOV R3, 0           ; Store 0 in R3 to indicate inequality

END:
; End of the program
```

; R3 holds 1 if numbers were equal, 0 if they were not equal.

Loops in Assembly:

1. Write a program to calculate the sum of the first N natural numbers.

```
LOAD R1, N        ; Load the value of N (e.g., 5) into register R1
MOV R2, 0         ; Initialize sum (R2) to 0
MOV R3, 1         ; Initialize counter (R3) to 1

LOOP:
CMP R3, R1        ; Compare counter (R3) with N (R1)
BEQ DONE          ; If counter equals N, branch to DONE

ADD R2, R2, R3    ; Add counter (R3) to the sum (R2)
INC R3            ; Increment the counter (R3)
JMP LOOP          ; Repeat the loop

DONE:
; At this point, R2 contains the sum of the first N natural numbers

; The program ends here, and the sum is stored in R2
```

2. Implement a multiplication operation using iterative addition.

```
LOAD R1, A        ; Load the first number (A) into register R1
LOAD R2, B        ; Load the second number (B) into register R2
MOV R3, 0         ; Initialize result (R3) to 0
MOV R4, 0         ; Initialize counter (R4) to 0

LOOP:
CMP R4, R2        ; Compare counter (R4) with B (R2)
BEQ DONE          ; If counter equals B, branch to DONE

ADD R3, R3, R1    ; Add A (R1) to result (R3)
INC R4            ; Increment counter (R4)
JMP LOOP          ; Repeat the loop

DONE:
; At this point, R3 contains the result of A * B

; The program ends here, and the result is stored in R3
```

Arrays in Assembly

1. Write a program to find the maximum value in an array.

```
LOAD R4, N          ; Load the length of the array (N) into R4
LOAD R5, array      ; Load the address of the array into R5
MOV R2, [R5]        ; Set R2 (max_value) to the first element of the array
MOV R3, 1           ; Set the counter (index) to 1 (start from the second element)

LOOP:
CMP R3, R4          ; Compare counter (R3) with the length of the array (R4)
BEQ DONE            ; If counter equals the length, we're done

ADD R6, R5, R3      ; Calculate the address of the current element: R6 = base address + index
MOV R1, [R6]        ; Load the current element into R1
CMP R1, R2          ; Compare current element (R1) with max_value (R2)
BLT NEXT            ; If current element is smaller, skip updating max_value

MOV R2, R1          ; If current element is larger, update max_value

NEXT:
INC R3              ; Increment the index (counter)
JMP LOOP            ; Repeat the loop

DONE:
; At this point, R2 contains the maximum value in the array

; The program ends here, and the maximum value is stored in R2
```

2. Sort an array using the bubble sort algorithm.

Program to sort an array using the Bubble Sort algorithm

LOAD R4, N ; Load the length of the array (N) into R4 LOAD R5, array ; Load the address of the array into R5

MOV R2, 0 ; Initialize the outer loop index (R2) to 0

OUTER_LOOP: CMP R2, R4 ; Compare the outer loop index (R2) with the length (R4) BEQ DONE ; If index equals length, the array is sorted

MOV R3, 0 ; Initialize the inner loop index (R3) to 0

INNER_LOOP: CMP R3, R4 ; Compare the inner loop index (R3) with the length minus the outer loop index BEQ NEXT_OUTER_LOOP ; If inner index equals length - outer index, exit inner loop

ADD R6, R5, R3 ; Calculate the address of the current element: R6 = base address + R3 ADD R7, R5, R3+1 ; Calculate the address of the next element: R7 = base address + (R3 + 1) MOV R1, [R6] ; Load the current element into R1 MOV R8, [R7] ; Load the next element into R8

CMP R1, R8 ; Compare the current element (R1) with the next element (R8) BLT SKIP_SWAP ; If current element is less than or equal to the next, skip swapping

; Swap elements if R1 > R8 MOV [R6], R8 ; Store the next element (R8) at the current position (R6) MOV [R7], R1 ; Store the.

Assessment Questions

1. Describe the difference between RISC and CISC architecture.

	RISC	CISC
Instruction Set Complexity	RISC processors have a simpler instruction set	CISC processors have a more complex instruction set
Instruction Execution time	RISC processors typically execute instruction in a single clock cycle	CISC processors may take multiple clock cycles to execute a single instruction
Power consumption	RISC processors consume less power	CISC consumes more power

2. Explain the role of program counter in ARM architecture.

In the ARM (Advanced RISC Machines) architecture, **the Program Counter (PC)** is a crucial register that plays a central role in the execution of instructions points to the address of the next instruction to be fetched from the main memory when the previous instruction has been completed successfully. **Program counter** also functions to count the number of instructions.

3. What is the significance of condition codes in ARM assembly?

Condition codes in ARM assembly language are crucial for controlling the flow of program execution. Here's their significance:

Conditional Execution:

Condition codes enable conditional execution of instructions, allowing the program to make decisions based on previous results.

Decision Making:

Condition codes facilitate decision-making by providing information about the outcome of previous instructions, such as equality, inequality, or overflow.

Efficient Coding:

Condition codes allow for more efficient coding by reducing the number of instructions needed to implement conditional logic.

Improved Performance:

By minimizing the number of instructions and branches, condition codes can improve program performance.

Common Condition Codes

ARM assembly language uses the following condition codes:

1. EQ (Equal)
2. NE (Not Equal)
3. CS (Carry Set)

4. CC (Carry Clear)

5. MI (Minus, i.e., Negative)

6. PL (Plus, i.e., Positive or Zero)

7. VS (Overflow)

8. VC (No Overflow)

9. HI (Higher)

10. LS (Lower or Same)

11. GE (Greater or Equal)

12. LT (Less Than)

13. GT (Greater Than)

14. LE (Less or Equal)

Instruction Format:

Condition codes are specified in the instruction format using a 4-bit condition code field.

Example Instructions:

1. BNE (Branch if Not Equal)

2. BEQ (Branch if Equal)

3. BMI (Branch if Minus, i.e., Negative)

4. How does the ARM pipeline improve performance?

The ARM pipeline is a critical component of the ARM processor architecture, designed to improve performance by breaking down the instruction execution process into a series of stages. Here's how the ARM pipeline enhances performance:

Stages of the ARM Pipeline

1. Fetch: Instruction fetch from memory.
2. Decode: Instruction decoding and register access.
3. Execute: Instruction execution, including arithmetic and logical operations.
4. Memory Access: Memory access for load and store instructions.
5. Write-Back: Result write-back to registers.

Performance Enhancements

1. Increased Throughput: The pipeline allows for multiple instructions to be processed simultaneously, increasing overall throughput.
2. Improved Instruction-Level Parallelism: The pipeline enables the processor to execute multiple instructions in parallel, improving instruction-level parallelism.
3. Reduced Instruction Execution Time: By breaking down instruction execution into stages, the pipeline reduces the execution time for individual instructions.

4. Better Resource Utilization: The pipeline optimizes resource utilization by minimizing idle time and maximizing the use of execution units.

5. Enhanced Predictability: The pipeline's stage-by-stage execution makes instruction execution more predictable, allowing for better optimization.

5. Compare direct and indirect addressing modes in ARM assembly.

In ARM assembly language, addressing modes determine how the processor accesses and manipulates data. Here's a comparison of direct and indirect addressing modes:

Direct Addressing Mode

In direct addressing mode, the address of the operand is specified directly in the instruction.

Characteristics:

1. Address specified directly: The address of the operand is encoded in the instruction.

2. Fast access: Direct addressing mode provides fast access to data since the address is already known.

3. Limited flexibility: Direct addressing mode is less flexible since the address is hardcoded.

Example:

LDR R0, =0x1000

In this example, the instruction loads the value at address 0x1000 into register R0.

Indirect Addressing Mode

In indirect addressing mode, the address of the operand is stored in a register or memory location.

Characteristics:

1. Address stored in register/memory: The address of the operand is stored in a register or memory location.
2. More flexible: Indirect addressing mode is more flexible since the address can be changed dynamically.
3. Slower access: Indirect addressing mode provides slower access to data since the address needs to be retrieved first.

Example:

LDR R0, [R1]

6. What is the difference between LDR and LDM instructions?

	LDR	LDM
Number of registers	LDR loads single register	LDM loads multiple registers
Base register increment	In LDM, the base register is incremented after the load if the! Symbol is used	In LDR, the base register is not incremented
Use cases	Suitable for loading single values, such as integers or pointers	Suitable for loading multiple values, such as structure or arrays
Example	ARM LDR R0, [R1, #4]	ARM LDM R0! {R1, R2, R3}

7. Explain the use of the stack pointer in subroutine calls.

In ARM assembly language, the stack pointer (SP) plays a crucial role in subroutine calls. Here's how it's used:

Role of the Stack Pointer

The stack pointer is a register that keeps track of the current top of the stack. When a subroutine is called, the current state of the processor,

including registers and the return address, needs to be saved. This is where the stack comes in.

Pushing Registers onto the Stack

Before calling a subroutine, the caller pushes the necessary registers onto the stack using the STM (Store Multiple) instruction. This instruction stores multiple registers onto the stack, starting from the address pointed to by the SP.

Storing the Return Address

The return address is stored on the stack using the BL (Branch and Link) instruction. This instruction branches to the subroutine and stores the return address in the link register (LR).

Updating the Stack Pointer

After pushing the registers and storing the return address, the SP is updated to point to the new top of the stack. This is typically done using the SUB instruction, which subtracts a value from the SP.

Subroutine Execution

The subroutine executes, using the stack to store local variables and temporary data.

Returning from the Subroutine

When the subroutine finishes executing, it returns to the caller using the LDM (Load Multiple) instruction to restore the registers from the stack. The return address is retrieved from the LR, and the program counter (PC) is updated to point to the instruction following the BL instruction.

Restoring the Stack Pointer

Finally, the SP is restored to its original value using the ADD instruction, which adds a value to the SP.

Example Code

Here's an example of how the stack pointer is used in a subroutine call:

arm

; Caller code

STMDB SP! {R4-R6, LR}; Push registers onto stack

BL my subroutine; Call subroutine

LDMIA SP! {R4-R6, PC}; Restore registers and return

```
; Subroutine code
my subroutine
    ; Subroutine code here
    BX LR; Return to caller
```

In this example, the caller pushes the registers R4-R6 and the link register (LR) onto the stack using the STMDB instruction. The BL instruction calls the subroutine and stores the return address in the LR. The subroutine executes and returns to the caller using the BX instruction. Finally, the caller restores the registers and returns using the LDMIA instruction.

8. How are interrupts handled in ARM architecture?

In ARM architecture, interrupts are handled through a combination of hardware and software mechanisms. Here's an overview of how interrupts are handled:

Interrupt Handling Process

1. Interrupt Detection: The processor detects an interrupt request.
2. Interrupt Acknowledgment: The processor acknowledges the interrupt.
3. Context Saving: The processor saves the current context.

4. Vector Table Lookup: The processor looks up the ISR address in the vector table.

5. ISR Execution: The processor executes the ISR.

6. Context Restoration: The processor restores the saved context.

7. Interrupt Completion: The processor completes the interrupt handling process.

Types of Interrupts

1. FIQ (Fast Interrupt Request): High-priority interrupts that require immediate attention.

2. IRQ (Interrupt Request): Normal-priority interrupts.

3. SWI (Software Interrupt): Interrupts generated by software.

ARM Interrupt Modes

1. FIQ Mode: Handles high-priority interrupts.

2. IRQ Mode: Handles normal-priority interrupts.

3. Supervisor Mode: Handles system calls and exceptions.

4. Abort Mode: Handles memory access faults.

5. Undefined Mode: Handles undefined instructions.

ARM architecture handles interrupts through a combination of hardware mechanisms (interrupt controllers and signals) and software mechanisms

(ISRs, vector tables, and context switching). The processor supports multiple interrupt modes and types, ensuring efficient and prioritized handling of interrupts

9. What are the advantages of using thumb instructions in ARM?

Thumb instructions are a subset of the ARM instruction set that provide a more compact and efficient way of coding. Here are the advantages of using Thumb instructions in ARM:

Advantages of Thumb Instructions

- **Code Density**

Thumb instructions are 16 bits long, which is half the size of ARM instructions (32 bits). This results in a significant reduction in code size, making it ideal for applications where memory is limited.

- **Improved Code Execution**

Thumb instructions are designed to be more efficient and execute faster than ARM instructions. This is because Thumb instructions are optimized for the most common operations, reducing the number of cycles required to execute a task.

- **Enhanced Performance**

Thumb instructions provide better performance in terms of instructions per clock cycle (IPC). This is because Thumb instructions are designed to minimize the number of cycles required to execute a task, resulting in improved overall performance.

- **Reduced Power Consumption**

Thumb instructions consume less power than ARM instructions due to their compact size and efficient execution. This makes Thumb instructions ideal for battery-powered devices where power consumption is a critical factor.

- **Easier Debugging**

Thumb instructions provide easier debugging due to their simpler and more consistent instruction format. This makes it easier for developers to identify and fix errors, reducing the overall development time.

- **Better Support for Embedded Systems**

Thumb instructions are designed to support embedded systems, which often have limited resources.

10. Write a code snippet for swapping two numbers without using a third variable

Here is a code snippet in ARM assembly language that swaps two numbers without using a third variable:

```
arm
; Swap two numbers without using a third variable

; R0 and R1 are the two numbers to be swapped

SWAP:
    ; XOR R0 and R1
    EOR R0, R0, R1
    ; XOR R1 and R0 (now contains XOR of original R0 and R1)
    EOR R1, R1, R0
    ; XOR R0 (now contains XOR of original R0 and R1) and R1 (now contains original R0)
    EOR R0, R0, R1

    ; R0 and R1 are now swapped
    BX LR
```

This code uses the XOR swap algorithm, which works by using the XOR operation to swap the values of two registers without using a third register.

11. Define the term “endianness” and its impact on memory storage in ARM.

Endianness refers to the order in which bytes are stored in memory. It's a fundamental concept in computer architecture, and it affects how data is represented and manipulated in memory.

Impact on Memory Storage in ARM

ARM processors can operate in both big endian and little-endian modes. The choice of endianness affects how data is stored and accessed in memory.

- **Big Endian Mode**

In big endian mode, the ARM processor stores data in memory with the MSB first. This means that:

- Multi-byte values (such as integers and floating-point numbers) are stored with the most significant byte at the lowest memory address.

- Byte addressing is done from the most significant byte to the least significant byte.

- **Little Endian Mode**

In little endian mode, the ARM processor stores data in memory with the LSB first. This means that:

- Multi-byte values are stored with the least significant byte at the lowest memory address.
- Byte addressing is done from the least significant byte to the most significant byte.

12. How does the barrel shifter in ARM instructions work?

The barrel shifter is a critical component of the ARM instruction set architecture. It's a versatile and powerful feature that enables the processor to perform various shift and rotate operations on data.

How the Barrel Shifter Works.....?

The barrel shifter is a combinational logic circuit that performs shift and rotate operations on a 32-bit or 64-bit data value. It's called a "barrel" shifter because it can shift the bits in a circular fashion, wrapping around

from the most significant bit (MSB) to the least significant bit (LSB) or vice versa.

13. Why is pipelining important in ARM processors?

Pipelining is a critical technique used in ARM processors to improve performance, increase throughput, and reduce power consumption. Here are some reasons why pipelining is important in ARM processors:

Improved Performance

1. Increased Instruction-Level Parallelism (ILP): Pipelining allows multiple instructions to be executed concurrently, improving ILP and increasing overall performance.

2. Reduced Instruction Execution Time: By breaking down the instruction execution process into stages, pipelining reduces the execution time for individual instructions.

Increased Throughput

1. Higher Instruction Throughput: Pipelining enables the processor to execute more instructions per clock cycle, increasing overall throughput.

2. Better Resource Utilization: Pipelining ensures that processor resources, such as execution units and registers, are utilized more efficiently.

Reduced Power Consumption

1. Lower Clock Frequency: Pipelining allows the processor to operate at a lower clock frequency, reducing power consumption and heat generation.

2. Reduced Switching Activity: By executing instructions in a pipelined fashion, the processor reduces switching activity, which contributes to power consumption.

Improved Responsiveness

1. Faster Response to Interrupts: Pipelining enables the processor to respond more quickly to interrupts, improving overall system responsiveness.

2. Better Real-Time Performance: Pipelining helps ensure that the processor meets real-time deadlines and constraints.

14. Explain how floating-point operations differ from integer operations.

Floating-point operations differ significantly from integer operations in several ways. Here are the key differences:

Floating-point numbers are represented in a binary format that includes three main components: the sign bit, the exponent, and the mantissa. This format allows for the representation of very large and very small numbers, as well as fractions and decimals.

In contrast, integer operations involve the manipulation of whole numbers, which are typically represented in a binary format using two's complement notation.

One of the primary differences between floating-point and integer operations is the way in which arithmetic is performed. Floating-point operations require the alignment of the mantissas, adjustment of the exponents, and normalization of the results. This process can be complex and time-consuming.

Integer operations, on the other hand, involve simple binary arithmetic, such as addition, subtraction, multiplication, and division. These operations are typically faster and more straightforward than their floating-point counterparts.

Another significant difference between floating-point and integer operations is the concept of precision and rounding. Floating-point numbers have a limited number of bits available to represent the mantissa, which can lead to rounding errors and loss of precision.

Integer operations, by their very nature, do not involve rounding or precision issues, as the results of integer arithmetic are always exact.

In terms of hardware implementation, floating-point operations often require specialized hardware, such as floating-point units (FPUs) or graphics processing units (GPUs). These units are designed specifically to handle the complex arithmetic and rounding requirements of floating-point operations.

Integer operations, on the other hand, can typically be performed using the same hardware that handles general-purpose processing.

15. What are the advantages of inline assembly in ARM based C programming?

Inline assembly in ARM-based C programming offers several advantages, including:

Improved Performance

- Direct access to hardware resources, allowing for optimized code execution
- Ability to use specialized instructions and addressing modes not available in C

Low-Level Hardware Control

- Fine-grained control over hardware registers and peripherals
- Ability to implement custom hardware interfaces and protocols

Optimized Code Generation

- Ability to hand-optimize critical code sections for size and speed
- Reduced overhead of function calls and returns

Enhanced Security

- Ability to implement secure coding practices, such as secure data storage and transmission
- Fine-grained control over memory access and protection

Better Code Readability

- Ability to embed assembly code directly into C code, improving readability and maintainability
- Reduced need for separate assembly language files

Custom Instruction Implementation

- Ability to implement custom instructions and algorithms not available in standard C libraries
- Enhanced flexibility and programmability

Interfacing with Legacy Code

- Ability to interface with legacy assembly language code and libraries
- Simplified integration with existing codebases

Real-Time Systems Development

- Ability to develop real-time systems with predictable and reliable performance
- Fine-grained control over system resources and timing

Embedded Systems Development

- Ability to develop efficient and compact code for resource-constrained embedded systems
- Fine-grained control over hardware resources and peripherals