Project Overview

ARM Processors

COA PROGRAMMING ASSIGNMENT

<Mooksh jain>
<23FE10CSE00500>
<SECOND YEAR (K)>

An Overview

ARM (Advanced RISC Machine) processors are a family of computer processors based on the RISC (Reduced Instruction Set Computer) architecture. Developed initially by ARM Holdings, they are renowned for their power efficiency and performance, making them the backbone of modern mobile devices, embedded systems, and other energy-conscious technologies.
One of the defining features of ARM processors is their simplicity. Unlike CISC (Complex Instruction Set Computer) architectures, ARM processors use a streamlined set of instructions, enabling faster execution and reduced energy consumption. This design has propelled their adoption in smartphones, tablets, IoT devices, and even servers, where low power consumption is critical.
ARM processors support a variety of instructions, with basic operations like **LDR** (Load Register) for loading data from memory and **STR** (Store Register) for storing data into memory. These instructions, along with others like **MOV** (Move) and **ADD** (Addition), provide a foundation for efficient computation.

Advantages of ARM Processors

- **Energy Efficiency:** ARM processors consume less power, making them ideal for mobile and battery-operated devices.
- **Performance:** With a focus on RISC architecture, they execute instructions quickly and efficiently.
- **Cost-Effective:** ARM processors are cheaper to produce compared to more complex architectures.
- **Scalability:** Widely used across different devices, from smartphones to embedded systems and servers.
- **Versatile Ecosystem:** Supported by extensive development tools, software libraries, and a large developer community.

Disadvantages of ARM Processors

- **Limited Processing Power:** Not as powerful as some CISC (Complex Instruction Set Computing) architectures for heavy computational tasks.
- **Compatibility:** Software designed for x86 architecture (common in desktops) often needs to be adapted or emulated.
- **Single-Thread Performance:** Typically slower in single-threaded applications compared to some other architectures.
- **Complex Licensing:** ARM’s licensing model may increase costs for customization.

Basic ARM Instructions

1. Data Transfer Instructions:

- **LDR (Load Register):**
  Loads data from memory into a register.
  Example: LDR R8, [R5, R4, LSL #2] – Load data from memory location (R5 + R4 * 4) into R8.
- **STR (Store Register):**
  Stores data from a register into memory.
  Example: STR R10, [R7, R4, LSL #2] – Store the value in R10 at memory location (R7 + R4 * 4).
- **LDR Rn, =value:**
  Loads an immediate value (like a constant or address) directly into a register.
  Example: LDR R6, =arr – Load the address of arr into R6.

2. Arithmetic Instructions:

- **ADD (Add)**:
  Adds two registers or adds an immediate value to a register and stores the result in a destination register.
  Example: ADD R5, R5, #1 – Add 1 to R5 and store the result in R5.
- **MUL (Multiply):**
  Multiplies two registers and stores the result in a destination register.
  Example: MUL R10, R8, R9 – Multiply R8 and R9 and store the result in R10.

3. Initialization and Program Termination:

- **MOV (Move):**
  Copies an immediate value or the contents of one register into another.
  Example: MOV R4, #0 – Set R4 to 0.
- **.global _start:**
  Marks the entry point for the program. This is an assembler directive indicating the start of the program execution.
- **END:**
  Marks the end of the program (commonly used in assembly code to indicate termination).

4. Logic and Bitwise Instructions:

- **LSL (Logical Shift Left):**
  Shifts the contents of a register left by a specified number of bits, filling with zeroes. Often used for array indexing (multiplying by powers of 2).
  Example: LDR R7, [R5, R4, LSL #2] – Load from R5 + (R4 * 4).
- **LSR (Logical Shift Right):**
  Shifts the contents of a register right by a specified number of bits, filling the leftmost bits with zeros. This operation is often used for performing division by powers of 2, especially in unsigned integer operations.

5. Comparison and Branch Instructions:

- **CMP (Compare):**
  Compares two registers by subtracting the second register from the first, updating condition flags.
  Example: CMP R4, #10 – Compare R4 with 10.
- **B (Branch):**
  Jumps to the target address unconditionally.
  Example: B loop_start – Branch to the label loop_start.
- **BGE (Branch if Greater or Equal):**
  Jumps to the target address if the comparison (from CMP) shows that the first register is greater than or equal to the second register.
  Example: BGE end_loop – Branch to end_loop if R4 >= 10.
- **BGT (Branch if Greater Than):**
  Jumps if the first register is greater than the second register.
- **BNE (Branch if Not Equal):**
  Jumps if the registers are not equal.
- **BLE (Branch if Less or Equal):**
  Jumps if the first register is less than or equal to the second.

<CODE>

1.Write an ARM assembly program to compute: 4 + 5 - 19. Save the result in r1.

```
START
    MOV r0, #4                          ; Load 4 into r0
    ADD r0, r0, #5                      ; Add 5 to r0 (r0 = 4 + 5)
    SUB r1, r0, #19                     ; Subtract 19 from r0, store result in r1

    END                                 ; End of program
```

2.IMMEDIATE ADDRESSING INTO REGISTER

```
START
    MOV r0, #10       ; Load immediate value 10 into register r0
    MOV r1, #20       ; Load immediate value 20 into register r1
    ADD r2, r0, r1    ; Add values in r0 and r1, store result in r2
    SUB r3, r2, #5    ; Subtract immediate value 5 from r2, store result in r3

    END               ; End of program
```

3.USE OF LOGICAL (AND ,ORR,EOX,MVN)OPERATORS

```
START
    MOV r0, #10        ; Load unsigned value 10 into r0
    MOV r1, #12        ; Load unsigned value 12 into r1

    AND r2, r0, r1     ; Perform bitwise AND between r0 and r1, store result in r2
    ORR r3, r0, r1     ; Perform bitwise OR between r0 and r1, store result in r3
    EOR r4, r0, r1     ; Perform bitwise XOR between r0 and r1, store result in r4

    END                ; End of program
```

4.Multiplication instruction

```
 START
   MOV r0, #6 ; Load 6 into r0
   MOV r1, #7 ; Load 7 into r1
   MUL r2, r0, r1 ; Multiply r0 and r1, store the result in r2

   END ; End of program
```

5.Pre increment and Post increment

```
start:
  LDR R0, =array      ; Load the address of 'array' into R0
  LDR R1, [R0]       ; Load value at address in R0 (array[0]) into R1
  LDR R2, [R0, #8]!     ; Pre-increment: Load value at address (R0 + 8) into R2 and then
increment R0 by 8
  LDR R3, [R0], #8     ; Post-increment: Load value at address in R0 (array[2]) into R3
and then increment R0 by 8

  .data           ; Data section
array:
  .word 10, 20, 30, 40
```

6. logical shift right and left code

```
 MOV R0, #0b11010000 ; Load a binary value (208 in decimal) into R0
 MOV R1, #2 ; Load the shift amount (2) into R1

 #Logical Shift Left (LSL)
 LSL R2, R0, R1 ; Perform Logical Shift Left on R0 by R1 (2 positions), result in R2

 #Logical Shift Right (LSR)
 LSR R3, R0, R1 ; Perform Logical Shift Right on R0 by R1 (2 positions), result in R3

 END ; End of program
```

7.for(i=10; i>0; i--) { c[i] = a[i] + b[i];}

```
start:
    MOV R0, #10               ; Initialize i = 10 (R0 holds i)
    LDR R1, =a                 ; Load address of array a into R1
    LDR R2, =b                 ; Load address of array b into R2
    LDR R3, =c                 ; Load address of array c into R3

loop_start:
    CMP R0, #0                ; Compare i (R0) with 0
    BEQ loop_end               ; If i == 0, break out of loop

    LDR R4, [R1, R0, LSL #2]  ; Load a[i] into R4 (shift i by 2 for word offset)
    LDR R5, [R2, R0, LSL #2]  ; Load b[i] into R5 (shift i by 2 for word offset)
    ADD R6, R4, R5            ; Add a[i] and b[i], store result in R6
    STR R6, [R3, R0, LSL #2]  ; Store result in c[i]

    SUB R0, R0, #1            ; Decrement i
    B loop_start               ; Jump back to the start of the loop

loop_end:
    END                        ; End of program

    .data
a:  .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
b:  .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
c:  .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  ; Initialize c to 0
```

8.if ((r1 == r3) && (r5 == r6)) { r7 = r7 + 10;}

```
    CMP R1, R3           ; Compare R1 with R3
    BNE SKIP             ; If R1 != R3, jump to SKIP

    CMP R5, R6           ; Compare R5 with R6
    BNE SKIP             ; If R5 != R6, jump to SKIP

    ADD R7, R7, #10      ; If both conditions are true, increment R7 by 10

SKIP:                    ; Skip label. #Continue with the rest of the code
```

9.code for compare

```
start:
    MOV R0, #5              ; Load value 5 into R0
    MOV R1, #10             ; Load value 10 into R1

    CMP R0, R1              ; Compare R0 and R1 (effectively R0 - R1)

    BEQ equal               ; If R0 == R1, branch to 'equal'
    BGT greater             ; If R0 > R1, branch to 'greater'
    BLT less                ; If R0 < R1, branch to 'less'

equal:
    MOV R2, #1              ; If R0 == R1, set R2 to 1
    B end                   ; Jump to 'end'

greater:
    MOV R2, #2              ; If R0 > R1, set R2 to 2
    B end                   ; Jump to 'end'

less:
    MOV R2, #3              ; If R0 < R1, set R2 to 3
    B end                   ; Jump to 'end'

end ; Program ends here
```

10.if ((x > y) && (x > z)) {
count++;}

```
    MOV R0, #5                  ; Load value 5 into R0 (x)
    MOV R1, #3                  ; Load value 3 into R1 (y)
    MOV R2, #2                  ; Load value 2 into R2 (z)
    MOV R3, #0                  ; Initialize count to 0

    CMP R0, R1                  ; Compare x (R0) with y (R1)
    BLE skip_first_check        ; If x <= y, skip the first comparison

    CMP R0, R2                  ; Compare x (R0) with z (R2)
    BLE skip_second_check       ; If x <= z, skip the second comparison

    ADD R3, R3, #1              ; If x > y and x > z, increment count

skip_first_check:
skip_second_check:
    ; Continue with the rest of the program
    ; R3 contains the value of count
```

10.for (i = 0; i < 5; i++) {
sum += arr[i];}

```
start:
    MOV R4, #0          ; Initialize i = 0
    MOV R5, #0          ; Initialize sum = 0
    LDR R6, =arr        ; Load the address of arr into R6

loop_start:
    CMP R4, #5          ; Compare i with 5
    BGE end_loop        ; If i >= 5, exit the loop

    LDR R7, [R6, R4, LSL #2]  ; Load arr[i] into R7 (R6 + i * 4)
    ADD R5, R5, R7      ; Add arr[i] to sum
    ADD R4, R4, #1      ; Increment i
    B loop_start        ; Branch back to loop_start

end_loop:
    ; Program ends here
```

11.Array Assignment (Copying Values)

```
for (i = 0; i < 5; i++) {
    B[i] = A[i];
}
```

```
start:
    MOV R4, #0          ; Initialize i = 0
    LDR R5, =A          ; Load address of array A into R5
    LDR R6, =B          ; Load address of array B into R6

loop_start:
    CMP R4, #5          ; Compare i with 5
    BGE end_loop        ; If i >= 5, exit the loop

    LDR R7, [R5, R4, LSL #2]  ; Load A[i] into R7 (R5 + i * 4)
    STR R7, [R6, R4, LSL #2]  ; Store A[i] into B[i] (R6 + i * 4)
    ADD R4, R4, #1      ; Increment i
    B loop_start        ; Branch back to loop_start

end_loop:
    ; Program ends here
```

12.for (i = 0; i < 5; i++) {
C[i] = A[i] * B[i];
}

```
start:
    MOV R4, #0          ; Initialize i = 0
    LDR R5, =A          ; Load address of array A into R5
    LDR R6, =B          ; Load address of array B into R6
    LDR R7, =C          ; Load address of array C into R7

loop_start:
    CMP R4, #5          ; Compare i with 5
    BGE end_loop        ; If i >= 5, exit the loop

    LDR R8, [R5, R4, LSL #2]  ; Load A[i] into R8
    LDR R9, [R6, R4, LSL #2]  ; Load B[i] into R9
    MUL R10, R8, R9     ; Multiply A[i] * B[i] and store in R10
    STR R10, [R7, R4, LSL #2] ; Store result in C[i]

    ADD R4, R4, #1      ; Increment i
    B loop_start        ; Branch back to loop_start

end_loop:
    ; Program ends here
```