Computer Architecture CSF342
Lab sheet 2

Topic - Conditional branching and looping and loading storing other datatypes.

---

1. Data Types and Memory Operations

Data Types:

- **Byte (`.byte`)**: 1 byte (e.g., `value: .byte 0x0F`)
- **Halfword (`.half`)**: 2 bytes (e.g., `value: .half 256`)
- **Word (`.word`)**: 4 bytes (e.g., `array: .word 1, 2, 3`)
- **String (`.asciiz`)**: Null-terminated sequence (e.g., `str: .asciiz "Hello"`)

Memory Access Instructions:

- Load byte: `lb $t0, 2($s1)` → Loads byte at `$s1 + 2` into `$t0` (sign-extends)
- Load unsigned byte: `lbu $t0, 2($s1)`
- Store word: `sw $t2, 8($s3)` → Stores `$t2` at `$s3 + 8`
- **Alignment**: Words require 4-byte alignment (addresses divisible by 4).

---

2. Control Flow: Branching and Jumping

Unconditional Jump:

```
j loop_start   # Jump directly to label "loop_start"
```

Conditional Branching:

Instruction	Meaning	Example
beq	Branch if equal	beq $t0, $t1, equal_case
bne	Branch if not equal	bne $t0, $t1, not_equal
bgt	Branch if greater than*	bgt $t0, $t1, greater
blt	Branch if less than*	blt $t0, $t1, smaller

* `bgt`/`blt` are pseudo-instructions. MIPS uses `slt` (set less than) + `bne`/`beq` for comparisons.

Example: if-else (C → MIPS)

```
// C code
if (a == b) { c = 10; }
```

```
else { c = 20; }

# MIPS equivalent
    bne $s0, $s1, else   # if (a != b) → else
    li $s2, 10           # c = 10
    j end_if
else:
    li $s2, 20           # c = 20
end_if:
```

---

3. Example: Even-Odd Checker

```
.data
prompt: .asciiz "Enter a number: "
even_msg: .asciiz "Even!"
odd_msg: .asciiz "Odd!"

.text
main:
    li $v0, 4            # Print prompt
    la $a0, prompt
    syscall

    li $v0, 5            # Read integer → $v0
    syscall

    andi $t0, $v0, 1     # LSB = 0 → even, 1 → odd
    beq $t0, $zero, even

    li $v0, 4            # Print "Odd!"
    la $a0, odd_msg
    syscall
    j exit

even:
    li $v0, 4            # Print "Even!"
    la $a0, even_msg
    syscall

exit:
    li $v0, 10           # Exit
    syscall
```

---

4. Loops: Summing an Array

C While Loop:

```
int sum = 0, i = 0;
while (i < 10) {
    sum += array[i];
    i++;
}
```

MIPS Implementation:

```
.data
array: .word 5, 3, 8, 1, 7, 2, 9, 4, 6, 10   # 10 elements
msg: .asciiz "The sum of the array is: "

.text
main:
    li $t0, 10          # i = 10
    li $t1, 0           # sum = 0
    la $s0, array       # Base address of array

loop:
    beqz $t0, showsum   # Exit if i == 0 , direct comparison with zero
    lw $t2, 0($s0)      # Load array[i]
    add $t1, $t1, $t2   # sum += array[i]
    addi $s0, $s0, 4    # Move to next word (address += 4)
    subi $t0, $t0, 1    # i--
    j loop

showsum:
    la $a0 msg
    li $v0 4
    syscall
    move $a0 $t1
    li $v0 1
    syscall

end:
    # $t1 now holds the sum
    li $v0, 10
    syscall
```

---

5. Non-Evaluative Tasks

Task 1: GCD Calculator

- **Input**: Two positive integers from the user.
- **Logic**: Use Euclidean algorithm (repeated subtraction/division).
- **Sample**:

```
Enter first number: 54
Enter second number: 24
GCD: 6
```

```
.data
msg1 : .asciiz "Enter the first number : "
msg2 : .asciiz "Enter the second number : "
msg3 : .asciiz "GCD: "

.text
main:
```

```
li $v0,4
la $a0, msg1
syscall

li $v0, 5
syscall

move $s1,$v0

li $v0,4
la $a0,msg1
syscall

li $v0,5
syscall

move $s2,$v0

blt $s1,$s2,find_gcd
move $s3,$s1
move $s1,$s2
move $s2,$s3


find_gcd:
beq $s1,$0, printresult
div $s2, $s1
move $s2, $s1
mfhi $s1
j find_gcd

printresult:
li $v0,4
la $a0,msg3
syscall

li $v0,1
move $a0,$s2
syscall

li $v0,10
syscall
```

Task 2: Array First Derivative

- **Input**: Hardcoded integer array of size 10 (e.g., `[5, 2, 7, 1, 3, 8, 4, 9, 6, 0]`).
- **Output**: Array of size 9 where `derivative[i] = array[i+1] - array[i]`.
- **Sample**:

```
Input: [5, 2, 7, 1, 3, 8, 4, 9, 6, 0]
Output: [-3, 5, -6, 2, 5, -4, 5, -3, -6]
```

.data

```
array1: .word 5, 2, 7, 1, 3, 8, 4, 9, 6, 0
array2: .space 36

.text
.globl main
main:
   li $t0, 0
   li $t4, 9
   la $s0, array1
   la $s1, array2
   la $s2, array2

loop:
   beq $t0, $t4, printarr
   lw $t1, 0($s0)
   lw $t2, 4($s0)
   sub $t3, $t2, $t1
   sw $t3, 0($s1)

   addi $t0, $t0, 1
   addi $s0, $s0, 4
   addi $s1, $s1, 4
   j loop

printarr:
   li $t0, 9
   la $s2, array2

print_loop:
   beqz $t0, endprog
   lw $a0, 0($s2)
   li $v0, 1
   syscall


   li $a0, 32
   li $v0, 11
   syscall

   addi $t0, $t0, -1
   addi $s2, $s2, 4
   j print_loop

endprog:
   li $v0, 10
   syscall
```

Appendix: Branching and Looping Tips

1. **Label Naming**: Use descriptive names (`loop1`, `check_negative`, `exit_program`).
2. **Branch Conditions**:
   - Use `slt` (set less than) for custom comparisons:

```
slt $t0, $s1, $s2   # $t0 = 1 if $s1 < $s2
bne $t0, $zero, label
```

3. **Infinite Loop Safety**: Always increment loop counters *before* jumping.
4. **Array Traversal**:
   - Calculate offsets: `address = base + (index * 4)` for words.
   - Use pointers: Increment `$s0` by 4 after each `lw`.
5. **Optimization**:
   - Move loop-invariant code outside (e.g., loading constant values).
   - Minimize branches in tight loops.

Debugging: Step through loops in MARS to verify counter values and branches!

For Loop Example: String Reversal

C Code

```c
char str[] = "hello";
int len = 5;
for (int i = 0, j = len-1; i < j; i++, j--) {
    char temp = str[i];
    str[i] = str[j];
    str[j] = temp;
}
```

MIPS Equivalent

```
.data
str: .asciiz "hello"
len: .word 5

.text
main:
    la $s0, str         # Base address
    li $t0, 0           # i = 0
    lw $t1, len
    addi $t1, $t1, -1   # j = len - 1

loop:
    bge $t0, $t1, print  # Exit if i >= j

    # Load str[i]
    add $t2, $s0, $t0   # Address of str[i]
    lb $t3, 0($t2)      # $t3 = str[i]
```

```
    # Load str[j]
    add $t4, $s0, $t1  # Address of str[j]
    lb $t5, 0($t4)     # $t5 = str[j]

    # Swap
    sb $t5, 0($t2)     # str[i] = str[j]
    sb $t3, 0($t4)     # str[j] = temp

    # Update counters
    addi $t0, $t0, 1   # i++
    addi $t1, $t1, -1  # j--
    j loop

print:
    move $a0 $s0
    li $v0 4
    syscall

end:
    li $v0, 10
    syscall
```

A.4 Precautions for Conditional Branching

Follow these to avoid common control-flow errors:

1. **Order of Checks Matters**

```
# ❌ Danger: May skip critical code
beq $t0, 5, skip
addi $s0, $s0, 1  # This executes if $t0 != 5!
skip:
```

   Fix: Place dependent instructions **after** the branch target.

2. **Use Jumps to Skip Else-Blocks**

```
# ✅ Correct if-else structure
bne $s0, $s1, else
# If-block instructions
j end_if          # ← Critical jump to skip else-block
else:
# Else-block instructions
end_if:
```

3. **Avoid Backward Branches in Delays**
   If using delayed branching (advanced):

```
# ❌ Unpredictable if branch taken
loop:
  addi $t0, $t0, 1
  bne $t0, 10, loop  # Avoid backward branches in delay slots
```

4. **Test Edge Cases**

    - Always check loop bounds with values:
        - `i = 0` (first iteration)
        - `i = n-1` (last iteration)
        - `n = 0` (no iterations)

5. **Signed vs. Unsigned Branches**
   Use `blt`/`bgt` for signed comparisons, `bltu`/`bgtu` for unsigned:

```
# Check if $t0 (0xFFFFFFFF) > 5 (unsigned)
li $t1, 5
bltu $t1, $t0, large  # Correctly treats 0xFFFFFFFF as 4.2e9
```

Key Insight:

> "Branching in assembly is like directing traffic – a single missed sign causes collisions. Always map your logic to labels *before* coding."

These practices prevent infinite loops, incorrect skips, and off-by-one errors prevalent in control-flow logic.