NUST CHIP DESIGN CENTRE

Digital Design Verification

Lab Manual # 15 – RISC-V Procedures and Instruction Formats (Sorting using RISC-V Assembly language)

Name	*Khalil Rehman*
Instructor	*Sir Umer*
Date	*21th July 2025*

Lab Tasks:

Task # 1

```
.data
a: .word 56         # a
b: .word 98         # b
result: .word 0     # Memory location to store result

.text
.globl main

main:
    # Load addresses and values of a and b
    la t0, a        # Load address of a into t0
    lw t1, 0(t0)    # Load value of a into t1
    la t0, b        # Load address of b into t0
    lw t2, 0(t0)    # Load value of b into t2

loop:
    # Check if b == 0, if yes, exit loop
    beq t2, zero, end_loop

    # temp = b
    mv t3, t2       # t3 = temp = b

    # Calculate a % b using repeated subtraction
    mv t4, t1       # t4 = a (will become remainder)
    mv t5, t2       # t5 = b
mod_loop:
    blt t4, t5, mod_done  # If remainder < divisor, done
    sub t4, t4, t5        # Subtract divisor from remainder
    j mod_loop
mod_done:

    # b = a % b (which is now in t4)
    mv t2, t4

    # a = temp (which is in t3)
    mv t1, t3

    j loop          # Repeat the process

end_loop:
    # Store result (GCD is in t1)
    la t0, result
    sw t1, 0(t0)

    # Exit program
    li a7, 10
    ecall
```

Execution:

Final Value

Task # 2

```
#Bubble Sort
.data
array:  .word 12, 5, 7, 3, 19, 1, 25, 8, 2, 10, 4, 15   # 12 elements
n:      .word 12                                        # array length

    .text
    .globl _start
_start:
    # load n into s0
    la   t0, n
    lw   s0, 0(t0)          # s0 = n (12)

    addi s2, s0, -1         # passes = n-1

outer_loop:
    blt  s2, x0, done_sort  # if passes < 0 -> done

    li   s1, 0              # i = 0

inner_loop:
    bge  s1, s2, next_pass  # if i >= passes -> break inner

    la   t1, array          # base address of array
    slli t2, s1, 2          # offset = i*4
    add  t3, t1, t2         # t3 = &array[i]

    lw   t4, 0(t3)          # t4 = array[i]
    lw   t5, 4(t3)          # t5 = array[i+1]

    ble  t4, t5, no_swap    # if array[i] <= array[i+1] -> skip swap

    # swap
    sw   t5, 0(t3)
    sw   t4, 4(t3)

no_swap:
    addi s1, s1, 1
    j    inner_loop

next_pass:
    addi s2, s2, -1
    j    outer_loop

done_sort:
    # nothing printed, results are in memory
    j    done_sort          # stop here

```

Execution:

Registers Memory Cache VDB

Address	+3	+2	+1	+0
0x10000030	0	0	0	12
0x1000002C	0	0	0	25
0x10000028	0	0	0	19
0x10000024	0	0	0	15
0x10000020	0	0	0	12
0x1000001C	0	0	0	10
0x10000018	0	0	0	8
0x10000014	0	0	0	7
0x10000010	0	0	0	5
0x1000000C	0	0	0	4
0x10000008	0	0	0	3
0x10000004	0	0	0	2
0x10000000	0	0	0	1

Jump to -- choose -- Up Down

Task # 3

```
.data
array: .word -1, 22, 8, 35, 5, 4, 11, 2, 1, 78
length: .word 10

.text
.globl main

main:
    la a0, array        # Load array address
    li a1, 0            # low index = 0
    lw a2, length       # high index = length-1
    addi a2, a2, -1

    jal ra, quicksort   # Call quicksort

    # Exit program
    li a7, 10
    ecall

# QuickSort function
# a0: array address
# a1: low index
# a2: high index
quicksort:
    addi sp, sp, -16        # Allocate stack space
    sw ra, 0(sp)            # Save return address
    sw a1, 4(sp)            # Save low
    sw a2, 8(sp)            # Save high

    bge a1, a2, end_sort    # If low >= high, return

    jal ra, partition       # Call partition
    sw a0, 12(sp)          # Save pivot index

    # Recursive call for left partition
    lw a1, 4(sp)           # Load low
    lw a2, 12(sp)          # Load pivot index
    addi a2, a2, -1        # pivot_index - 1
    jal ra, quicksort
```

```
    # Recursive call for right partition
    lw a1, 12(sp)           # Load pivot index
    addi a1, a1, 1          # pivot_index + 1
    lw a2, 8(sp)            # Load high
    jal ra, quicksort

end_sort:
    lw ra, 0(sp)            # Restore return address
    addi sp, sp, 16         # Deallocate stack space
    ret

# Partition function
# a0: array address
# a1: low index
# a2: high index
# Returns: pivot index in a0
partition:
    # Calculate address of pivot (last element)
    slli t0, a2, 2          # high * 4 (word offset)
    add t0, a0, t0          # address of array[high]
    lw t1, 0(t0)            # t1 = pivot value

    addi t2, a1, -1         # i = low - 1 (smaller element index)
    mv t3, a1               # j = low (current element index)

partition_loop:
    bge t3, a2, end_partition_loop  # Exit if j >= high

    # Calculate address of array[j]
    slli t4, t3, 2          # j * 4
    add t4, a0, t4          # address of array[j]
    lw t5, 0(t4)            # t5 = array[j]

    bgt t5, t1, skip_swap   # If array[j] > pivot, skip

    addi t2, t2, 1          # i++

    # Swap array[i] and array[j]
    slli t6, t2, 2          # i * 4
    add t6, a0, t6          # address of array[i]
    lw t0, 0(t6)            # temp = array[i]
    sw t5, 0(t6)            # array[i] = array[j]
```

```
675     sw t5, 0(t6)            # array[i] = array[j]
676     sw t0, 0(t4)            # array[j] = temp
677
678 skip_swap:
679     addi t3, t3, 1          # j++
680     j partition_loop
681
682 end_partition_loop:
683     # Swap array[i+1] and array[high] (pivot)
684     addi t2, t2, 1          # i++
685     slli t0, t2, 2          # i * 4
686     add t0, a0, t0          # address of array[i]
687
688     slli t3, a2, 2          # high * 4
689     add t3, a0, t3          # address of array[high]
690
691     lw t4, 0(t0)            # temp = array[i]
692     lw t5, 0(t3)            # temp2 = array[high]
693     sw t5, 0(t0)            # array[i] = array[high]
694     sw t4, 0(t3)            # array[high] = temp
695
696     mv a0, t2               # Return pivot index (i)
697     ret
```

Execution:

0x10000024	0	0	0	78
0x10000020	0	0	0	1
0x1000001C	0	0	0	2
0x10000018	0	0	0	11
0x10000014	0	0	0	4
0x10000010	0	0	0	5
0x1000000C	0	0	0	35
0x10000008	0	0	0	8
0x10000004	0	0	0	22
0x10000000	-1	-1	-1	-1

Conclusion:

The conclusion of this lab is that implementing procedures like GCD, Bubble Sort, and Quick Sort in RISC-V assembly reinforces understanding of control flow, recursion, and stack management. It demonstrates the practical application of RISC-V instruction formats and the importance of efficient algorithm design in low-level programming.