ISA project- Documentation

Matan Adar- 322357542

Nevo Heller- 322401662

Introduction

This project involves the implementation of an assembler that translates assembly code into machine code, and a simulator that execute programs and implements a simple CPU.
In addition, we wrote 4 assembly test codes, that will test both the assembler and the simulator.

Below are the details of the CPU's **registers** and **instruction set**:

Register Number	Register Name	Purpose
0	$zero	Constant zero
1	$imm1	Sign-extended immediate 1
2	$imm2	Sign-extended immediate 2
3	$v0	Result value
4	$a0	Argument register
5	$a1	Argument register
6	$a2	Argument register
7	$t0	Temporary register
8	$t1	Temporary register
9	$t2	Temporary register
10	$s0	Saved register
11	$s1	Saved register
12	$s2	Saved register
13	$gp	Global pointer (static data)
14	$sp	Stack pointer
15	$ra	Return address

Opcode	Name	Operation
0	add	R[rd] = R[rs] + R[rt] + R[rm]
1	sub	R[rd] = R[rs] - R[rt] - R[rm]
2	mac	R[rd] = R[rs] * R[rt] + R[rm]
3	and	R[rd] = R[rs] & R[rt] & R[rm]
4	or	R[rd] = R[rs] | R[rt] | R[rm]
5	xor	R[rd] = R[rs] ^ R[rt] ^ R[rm]
6	sll	R[rd] = R[rs] << R[rt]
7	sra	R[rd] = R[rs] >> R[rt] (arithmetic shift)
8	srl	R[rd] = R[rs] >> R[rt] (logical shift)
9	beq	if (R[rs] == R[rt]) pc = R[rm][low 11:0]
10	bne	if (R[rs] != R[rt]) pc = R[rm][low 11:0]
11	blt	if (R[rs] < R[rt]) pc = R[rm][low 11:0]
12	bgt	if (R[rs] > R[rt]) pc = R[rm][low 11:0]
13	ble	if (R[rs] <= R[rt]) pc = R[rm][low 11:0]
14	bge	if (R[rs] >= R[rt]) pc = R[rm][low 11:0]
15	jal	R[rd] = pc + 1, pc = R[rm][11:0]
16	lw	R[rd] = MEM[R[rs] + R[rt]] + R[rm]
17	sw	MEM[R[rs] + R[rt]] = R[rm] + R[rd]
18	reti	PC = IORegister[7]
19	in	R[rd] = IORegister[R[rs] + R[rt]]
20	out	IORegister[R[rs] + R[rt]] = R[rm]
21	halt	Halts execution

Assembler Implementation

The assembler is implemented in C and follows a two-pass approach to translate assembly code into machine code. Below are key parts of the assembler and their functionality:

1. Parsing and Cleaning Assembly Lines

The function get_line() processes each line of assembly code by:

Removing leading/trailing whitespace and comments (#).

Replacing commas with spaces.

Condensing multiple spaces into a single space for easier tokenization.

2. Handling Labels

The assembler maintains a label table to map label names to instruction addresses:

addLabel() stores a label and its corresponding address.

get_label_address() retrieves the address of a given label.

3. Decoding Opcodes and Registers

get_opcode() converts an instruction name (e.g., add, sub) into its corresponding opcode.

get_reg_code() maps register names (e.g., $t0, $a1) to their numeric codes.

4. Encoding Instructions

The function encodeInstruction() formats instructions into 48-bit machine code, structured as follows:

Bits	Field	Description
47:40	opcode	Operation code
39:36	rd	Destination register
35:32	rs	Source register 1
31:28	rt	Source register 2
27:24	rm	Source register 3
23:12	imm1	First immediate value
11:0	imm2	Second immediate value

5. Handling .word Directives

The assembler supports memory initialization using .word:

This stores predefined values in memory before execution begins.

6. Two-Pass Assembly Process

First pass: Identifies and stores label addresses.

Second pass: Translates instructions into machine code using stored labels.

7. Generating Output Files

The assembler writes:

Instruction memory (.txt) – Encoded machine instructions.

Data memory (.txt) – Memory initialization values.

The assembler ensures correct instruction formatting and validates labels, registers, and immediates before final encoding.

Simulator Implementation

This simulator, also implemented in C, replicates the CPU execution cycle, ensuring correct program behavior and hardware interaction.

1. CPU and Memory Setup

Memory (data_memory and instruction_memory) is initialized.

CPU and I/O registers are set up.

2. Instruction Execution

decode_instruction() extracts fields from a 48-bit instruction.

process_instruction() executes ALU, memory, and branch operations.

3. Peripheral Handling

handle_disk_operations() handles disk read/write operations.

handle_monitor_operations() updates the display buffer.

update_timer() manages timer interrupts.

handle_led_and_display_operations() logs LED and 7-segment changes.

4. Simulation Loop

The simulation runs in a loop, fetching and executing instructions.

Clock cycles increment (increment_clock_cycle()), updating peripherals.

Interrupts are handled (check_and_handle_interrupts()).

Execution stops when a halt instruction is encountered.

Assembly Tests

1.mulmat.asm

```
# Initialize loop variables

add $a0, $zero, $zero, $zero, 0, 0   # $a0 := i = 0

:Outer_Loop

add $a1, $zero, $zero, $zero, 0, 0   # $a1 := j = 0

:Inner_Loop

add $t2, $zero, $zero, $zero, 0, 0   # $t2 := sum = 0

add $s2, $zero, $zero, $zero, 0, 0   # $s2 := k = 0

:Multiply_Loop

mac $t0, $a0, $imm1, $s2, 4, 0       # $t0 = i * 4 + k

add $t0, $t0, $imm1, $zero, 0x100, 0 # $t0 = &matrix1[i * 4 + k]

mac $t1, $s2, $imm1, $a1, 4, 0       # k * 4 + j

add $t1, $t1, $imm1, $zero, 0x110    # $t1 = &matrix2[k * 4 + j]

lw $t0, $t0, $zero, $zero, 0, 0      # $t0 = MEM[&matrix1[i * 4 +
k]] = matrix1[i * 4 + k]

lw $t1, $t1, $zero, $zero , 0, 0     # $t1 = matrix2[k * 4 + j]

mac $t2, $t0, $t1, $t2, 0, 0         # sum += matrix1[i * 4 + k] *
matrix2[k * 4 + j]

++add $s2, $s2, $imm1, $zero, 1, 0     # k

blt $zero, $s2, $imm1, $imm2, 4, Multiply_Loop # if k < 4 jump
to Multiply_Loop

mac $t0, $a0, $imm1, $a1, 4, 0       # $t0 = i * 4 + j

sw $zero, $t0, $imm1, $t2, 0x120, 0  # MEM[i * 4 + j + 0x120] =
sum = matrix_result[i * 4 + j] = sum
```

```
++add $a1, $imm1, $a1, $zero, 1, 0     # j

blt $zero, $a1, $imm1, $imm2, 4, Inner_Loop  # if (j < 4) jump to Inner_Loop

++add $a0, $imm1, $a0, $zero, 1, 0     # i

blt $zero, $a0, $imm1, $imm2, 4, Outer_Loop  # if (i < 4) jump to Outer_Loop

halt $zero, $zero, $zero, $zero, 0, 0  # exit the program


Matrix 1 Data #

word 0x100 1.

word 0x101 2.

word 0x102 3.

word 0x103 4.

word 0x104 5.

word 0x105 6.

word 0x106 7.

word 0x107 8.

word 0x108 1.

word 0x109 2.

word 0x10A 3.

word 0x10B 4.

word 0x10C 5.

word 0x10D 6.
```

word 0x10E 7.

word 0x10F 8.

Matrix 2 Data #

word 0x110 9.

word 0x111 2.

word 0x112 4.

word 0x113 6.

word 0x114 4.

word 0x115 5.

word 0x116 42.

word 0x117 3.

word 0x118 9.

word 0x119 6.

word 0x11A 4.

word 0x11B 3.

word 0x11C 4.

word 0x11D 7.

word 0x11E 42.

word 0x11F 3.

2.binom.asm

Start

lw $a0, $imm1, $zero, $zero, 0x100, 0 #load n-value to a0

```
lw $a1, $imm1, $zero, $zero, 0x101, 0 #load k-value to a1
add $ra, $imm2, $zero, $zero, 0, End #set last return address
add $s2, $zero, $zero, $zero, 0, 0 #initial add value
Init
sw $s2, $sp, $zero, $zero, 0, 0 #push added value to stack
sw $a1, $sp, $imm1, $zero, 1, 0 #push k-value to stack
sw $a0, $sp, $imm1, $zero, 2, 0 #push n-value to stack
sw $ra, $sp, $imm1, $zero, 3, 0 #push return address to stack
add $sp, $sp, $imm1, $zero, 4, 0 #move stack head
xor $t1, $a0, $a1, $zero, 0, 0 #set t1 to 0 if n==k
xor $t2, $a1, $zero, $zero, 0, 0 #set t2 to 0 if k==0
beq $zero, $t1, $zero, $imm2, 0, Stop #stop if first condition was satisfied
beq $zero, $t2, $zero, $imm2, 0, Stop #stop if second condition was
satisfied
Main
add $s2, $zero, $zero, $zero, 0, 0 #set result to 0
sub $a0, $a0, $imm1, $zero, 1, 0 # n = n-1
sub $a1, $a1, $imm1, $zero, 1, 0 # k = k-1
jal $ra, $zero, $zero, $imm2, 0, Init #Binom(n-1,k-1)
add $s2, $s2, $v0, $zero, 0, 0 #add return
add $a1, $a1, $imm1, $zero, 1, 0 # k = k+1
jal $ra, $zero, $zero, $imm2, 0, Init #Binom(n-1,k-1)
add $s2, $s2, $v0, $zero, 0, 0
beq $zero, $zero, $zero, $imm1, Rollback, 0 #prepare for return
Stop
```

```
add $v0, $imm1, $zero, $zero, 1, 0 #Set v0=1
sub $sp, $sp, $imm1, $zero, 4, 0 #roll back the stack
lw $s2, $sp, $zero, $zero, 0, 0 #pop added value from stack
lw $a1, $sp, $imm1, $zero, 1, 0 #pop k-value from stack
lw $a0, $sp, $imm1, $zero, 2, 0 #pop n-value from stack
lw $ra, $sp, $imm1, $zero, 3, 0 #pop return address from stack
beq $zero, $zero, $zero, $ra, 0, 0 #return
Rollback
add $v0, $s2, $zero, $zero, 0, 0 #set $v0=Binom(n-1,k)+Binom(n-1,k-1)
sub $sp, $sp, $imm1, $zero, 4, 0 #roll back the stack
lw $s2, $sp, $zero, $zero, 0, 0 #pop added value from stack
lw $a1, $sp, $imm1, $zero, 1, 0 #pop k-value from stack
lw $a0, $sp, $imm1, $zero, 2, 0 #pop n-value from stack
lw $ra, $sp, $imm1, $zero, 3, 0 #pop return address from stack
beq $zero, $zero, $zero, $ra, 0, 0 #return
End
sw $v0, $imm2, $zero, $zero, 0, 0x102 #Store the result
halt $zero, $zero, $zero, $zero, 0, 0 # halt
word 0x100 12 #the value of n.
word 0x101 5 #the value of k.
```

3.circle.asm

```
lw $t0, $imm2, $zero, $zero, 7, 0x100     # Load r from memory address 0x100 into $t0
blt $zero, $t0, $imm1, $imm2, 1, halt     # If r < 1, halt the program
(255) out $zero, $imm1, $zero, $imm2, 21, 255   # Set monitor pixel color to white
mac $s0, $t0, $t0, $zero, 0, 0            # Compute r^2 and store it in $s0
jal $ra, $zero, $zero, $imm2, 0, main     # Jump to main, storing return address in $ra
```

```
:halt
Stop execution #                halt $zero, $zero, $zero, $zero, 0, 0
:main
add $s1, $zero, $zero, $zero, 0, 0       # Initialize y = 0
:Loop1
add $s2, $zero, $zero, $zero, 0, 0       # Initialize x = 0
:Loop2
sub $t1, $s2, $imm1, $zero, 127, 0       # Compute x - 127
sub $t2, $s1, $imm1, $zero, 127, 0       # Compute y - 127
mac $t1, $t1, $t1, $zero, 0, 0           # Square (x - 127) and store in $t1
mac $t2, $t2, $t2, $zero, 0, 0           # Square (y - 127) and store in $t2
add $t1, $t1, $t2, $zero, 0, 0           # Compute (x - 127)^2 + (y - 127)^2
bgt $zero, $t1, $s0, $imm2, 0, L1        # If distance > r^2, skip drawing pixel
mac $t2, $s1, $imm1, $s2, 256, 0         # Compute pixel address: (y * 256) + x
out $zero, $imm1, $zero, $t2, 20, 0      # Set monitor address to pixel location
out $zero, $imm1, $zero, $imm2, 22, 1   # Write white color to pixel (monitorcmd = 1)

:L1
add $s2, $s2, $imm1, $zero, 1, 0         # Increment x
blt $zero, $s2, $imm1, $imm2, 256, Loop2  # If x < 256, continue inner loop
add $s1, $s1, $imm1, $zero, 1, 0         # Increment y
blt $zero, $s1, $imm1, $imm2, 256, Loop1  # If y < 256, restart outer loop
beq $zero, $zero, $zero, $ra, 0, 0       # Return to caller
word 0x100 0                    # Data section: radius value at address 0x100.
```

4.disktest.asm

```
# Enable disk interrupt (IRQ1)
# Enable IRQ1 for disk operations    out $zero, $zero, $imm1, $imm2, 1, 1
```

```
# Set stack pointer to 2048 (1 << 11)
# Initialize stack pointer              sll $sp, $imm1, $imm2, $zero, 1, 11

# Assign IRQ handler address
# Set interrupt handler to irq_service  out $zero, $imm1, $zero, $imm2, 6, irq_service

# Initialize disk registers
# Set disk sector number to 7           out $zero, $imm1, $zero, $imm2, 15, 7
# Set disk buffer address to 0x0        out $zero, $imm1, $zero, $imm2, 16, 0x0

# Set initial operation mode and sector counter
# $t2 = 1 (READ mode)                   add $t2, $imm1, $zero, $zero, 1, 0
# $s2 = 7 (sector count)                add $s2, $imm1, $zero, $zero, 7, 0

process_disk:
# Read disk status
# $t0 = diskstatus                      in $t0, $imm1, $zero, $zero, 17, 0

# If disk is busy (diskstatus == 1), skip issuing command
beq $zero, $t0, $imm1, $imm2, 1, skip_command

# Issue disk operation (READ or WRITE)
```

```
# diskcmd = $t2                    out $zero, $imm1, $zero, $t2, 14, 0

skip_command:
# If remaining sectors > 0, continue processing
    bne $zero, $s2, $imm1, $imm2, 0xffffffff, process_disk

# End execution
            halt $zero, $zero, $zero, $zero, 0, 0

irq_service:
# Check if last operation was WRITE
beq $zero, $t2, $imm1, $imm2, 2, switch_to_write

# Increment sector number
                    add $t1, $imm1, $s2, $zero, 1, 0

# Update disk sector register
                out $zero, $imm1, $zero, $t1, 15, 0

# Change mode to WRITE
                    add $t2, $imm1, $zero, $zero, 2, 0

# Return from interrupt
  beq $zero, $zero, $zero, $imm2, 0, irq_exit

switch_to_write:
```

```
# Change mode back to READ
                    add $t2, $imm1, $zero, $zero, 1, 0

# Decrease sector count
                      sub $s2, $s2, $imm1, $zero, 1, 0

# Update disk sector register
                   out $zero, $imm1, $zero, $s2, 15, 0

irq_exit:
# Clear IRQ1 status and return from interrupt
            out $zero, $zero, $imm2, $zero, 0, 4
             reti $zero, $zero, $zero, $zero, 0, 0
```