17th March 2025

ISA PROGRAM REPORT

Bhone Pyae Kyaw

6530327 Section - 541

Table of Contents

ISA Architecture Design and Implementation Report

1. ISA Overview

1.1 Instruction Format

Each instruction follows a fixed-length 32-bit structure, broken down into the following fields:

Field	Size (bits)
Opcode	5 bits
Target Reg	3 bits
Source Reg	3 bits
Immediate	21 bits
Total	32 bits

1.2 Registers

There are 8 General Purpose Registers (GPRs), identified as r0 through r7. They are represented by 3-bit binary codes as shown below:

Register Name	Code
r0	000
r1	001
r2	010
r3	011
r4	100
r5	101
r6	110
r7	111

---

2. Supported Instructions

The ISA supports five arithmetic and data manipulation instructions:

Instruction	Opcode	Description	Cycles
mov	00000	Move immediate or register	1
add	00001	Add immediate or register	1
sub	00010	Subtract immediate or register	1
mul	00011	Multiply immediate or register	4
div	00100	Divide immediate or register	6

---

3. Cycle Timing Justification

mov, add, sub: These are simple instructions that typically require only 1 cycle to execute.

mul: Requires 4 cycles due to increased computational complexity.

div: Requires 6 cycles, reflecting the more complex division compared to other instructions.

---

4. Instruction Encoding and Decoding

The **encode_instruction()** function converts human-readable instructions into their binary representation based on the instruction format. Immediate values are sign-extended to 21 bits, and registers are encoded as 3-bit fields.

Example:

Instruction: mov r1 25

Binary: 00000 001 000 000000000000011001

---

5. Execution Logic

The **execute_instructions()** function simulates a CPU fetching, decoding, and executing instructions. It supports:

- Handling immediate and register values.
- Performing arithmetic operations.
- Division-by-zero checking.
- Cycle count accumulation

---

6. Performance Metrics

- Total Cycle Count
- Total Instruction Count (excluding end)
- CPI (Cycles Per Instruction): Total Cycle Count / Total Instruction Count – up to 2 decimal place.

---

7. Conclusion

The designed ISA and simulator demonstrates:

- Instruction encoding and decoding.
- Basic arithmetic and data manipulation.
- Cycle counting and performance measurement (CPI).

8. SIMULATION PROGRAM CODE

```python
# ISA Architecture
# ------------------------------------
# 32 bits instruction
# 8 general purpose registers r0 - r7
# op-codes - mov, add, sub, mul, div

# Encoded instruction structure
# Opcode(5 bits) target_reg(3 bits) source_reg(3 bits) immi(21 bits) = 32 bits

opcodes = {
    'mov' : '00000',
    'add' : '00001',
    'sub' : '00010',
    'mul' : '00011',
    'div' : '00100'
}

cycle_values = {
    '00000' : 1, #mov
    '00001' : 1, #add
    '00010' : 1, #sub
    '00011' : 4, #mul
    '00100' : 6  #div
}

gprs = {
    'r0': '000',
    'r1': '001',
    'r2': '010',
    'r3': '011',
    'r4': '100',
    'r5': '101',
    'r6': '110',
    'r7': '111'
}
```

```python

gprs_values = {
    '000' : 0,
    '001' : 0,
    '010' : 0,
    '011' : 0,
    '100' : 0,
    '101' : 0,
    '110' : 0,
    '111' : 0
 }

def encode_instruction(instruction):
    #Opcode(5 bits) target_reg(3 bits) source_reg(3 bits) immi(21 bits) = 32 bits

    instruction_components = instruction.split()

    opcode = opcodes[instruction_components[0]]

    target_reg = gprs[instruction_components[1]]

    #check whether source register is a register or a number
    #for number case
    if instruction_components[2].lstrip('-').isnumeric():
        source_reg = '000'
        immi = int(instruction_components[2])

        #for handling negative value
        if immi < 0:
            immi = format((1 << 21) + immi, '021b')
        else:
            immi = format(immi & 0x1FFFFF, '021b')
```

```python
    #for register case
    else:
        source_reg = gprs[instruction_components[2]]
        immi = '0' * 21

    # concatenate to get the binary instruction
    binary_instruction = f"{opcode} {target_reg} {source_reg} {immi}"
    return binary_instruction
```

```python
def execute_instructions(instructions):
    global total_cycle_counts

    print("-------------------------------------------------------------------------------")
    print("  PC  | User's instruction   | Binary Encoed Instruction             |    Cycles")
    print("-------------------------------------------------------------------------------")

    for index, i in enumerate(instructions):
        if i == 'end 0 0':
            return
        else:
            binary_instruction = encode_instruction(i)

            binary_instruction_split = binary_instruction.split()

            opcode = binary_instruction_split[0]
            target_reg = binary_instruction_split[1]
            source_reg = binary_instruction_split[2]
            immi = binary_instruction_split[3]

            if immi[0] == '1':
                immi = int(immi, 2) - (1 << 21)
            else:
                immi = int(immi, 2)

            cycle = cycle_values[opcode]
            total_cycle_counts += cycle
            pc = index
            print(f" {pc:>4} | {i:<20} | {binary_instruction:<36} | {cycle:>6}")

            if opcode == '00000': #mov
                if source_reg == '000':
                    gprs_values[target_reg] = immi
                else:
```

```python
                if source_reg == '000':
                    gprs_values[target_reg] = immi
                else:
                    gprs_values[target_reg] = gprs_values[source_reg]

            if opcode == '00001': #add
                if source_reg == '000':
                    gprs_values[target_reg] += immi
                else:
                    gprs_values[target_reg] += gprs_values[source_reg]

            if opcode == '00010': #sub
                if source_reg == '000':
                    gprs_values[target_reg] -= immi
                else:
                    gprs_values[target_reg] -= gprs_values[source_reg]

            if opcode == '00011': #mul
                if source_reg == '000':
                    gprs_values[target_reg] *= immi
                else:
                    gprs_values[target_reg] *= gprs_values[source_reg]

            if opcode == '00100': #div
                if source_reg =='000':
                    if immi == 0:
                        print(f"Divided by zero, skip the instruction {pc}")
                        continue
                    gprs_values[target_reg] //= immi
                else:
                    if gprs_values[source_reg] == 0:
                        print(f"Divided by zero, skip the instruction {pc}")
                        continue
                    gprs_values[target_reg] //= gprs_values[source_reg]
```

```python
                        print(f"Divided by zero, skip the instruction {pc}")
                        continue
                    gprs_values[target_reg] //= gprs_values[source_reg]

instructions = [
    "mov r1 15",
    "add r1 10",
    "mov r2 5",
    "mul r2 r1",
    "sub r3 r2",
    "div r2 5",
    "mov r4 r3",
    "end 0 0"
]



total_cycle_counts = 0
total_instructions = len(instructions) - 1

execute_instructions(instructions)
print("-----------------------------------------------------------------------------")

print("Execution Completed!")
print()

print(f"Total Cycle counts after executing all the instructions = {total_cycle_counts}")
print(f"Total Instruction counts = {total_instructions}")
print(f"CPI = {round(total_cycle_counts / total_instructions, 2) }")
print()

print("After executing instructions, each register contains - ")
for reg, val in gprs_values.items():
    bit_form = format(val & 0xFFFFFFFF, '032b')
    print(f" {reg:>4} (r{int(reg, 2)})  | {val:<16} | {bit_form}")

```