Computer Architecture
Assignment – 2

MIPS non-pipelined processor design

KNV Aditya - IMT2023033
Abhijit Dibbidi - IMT2023054
Kusumanchi Vinay - IMT2023608

Factorial:

The factorial of a non-negative integer ‘n’, denoted by ‘n!’ is the product of all positive integers less than or equal to n.

For example:

$0!=1$
$1!=1$
$2!=2\times1=2$
$3!=3\times2\times1=6$

The assembly code we wrote for finding the factorial of an integer ‘n’ is:

```
.data
n: .word 5
fac: .word 0
.text
main:
    lw $t0, n
    addi $t1, $zero, 1

loop:
    beq $t0, $zero, store
    mul $t1, $t1, $t0
    subi $t0, $t0, 1
    j loop
store:
    sw $t1, fac
```

Fibonacci numbers:

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones.

If $F(n)$ represents the n-th Fibonacci number and $F(0)=0$, $F(1)=1$ then $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$.

So, the sequence is:
0, 1, 1, 2, 3, 5, 8, 13, 21,... and so on.

The assembly code we wrote for finding the n-th number of the Fibonacci series is:

```
.data
n: .word 5
result: .word 0
.text
lw $t0, n
addi $t1, $zero, 0
addi $t2, $zero, 1

fibo:
    subi $t0, $t0, 1
    beq $t0, $zero, exit
    add $t3, $t2, $t1
    add $t1, $zero, $t2
    add $t2, $zero, $t3
    j fibo

exit:
    sw $t2, result
```

Even or Odd:

We can determine if a number is odd or even by doing bitwise AND with 1.
If the LSB of the input number is 1, it is an odd number; otherwise, it is even.
Since, in binary representation, odd numbers always have the least significant bit (LSB) as 1.

Truth-table for bitwise AND:
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

The assembly code we wrote to find if a given number is even or odd is:

```
.data
n: .word 5
result: .word 0

.text
main:
    lw $t0, n
    addi $t1, $zero, 1
    and $t2, $t1, $t0
    sw $t2, result
```

The MIPS non-pipelined processor we designed for the above assembly codes using IF (Instruction fetch), ID (Instruction decode), EX(Execute), Mem (Memory access), WB (Write back):

```
class Processor:
    def __init__(self, Instruction_Memory, Data_Memory):
        self.reg = [0] * 32
        self.instr_mem = Instruction_Memory
        self.data_mem = Data_Memory
        self.pc = 4194304

    # FETCH STAGE
    def fetch(self):
        instruction = self.instr_mem[self.pc] #fetching instruction
        self.pc += 4 #incrementing pc
        return instruction

    # DECODE STAGE
    def decode(self, instruction):
        op = (instruction >> 26) & 0x3F  # 6bit
        rs = (instruction >> 21) & 0x1F  # 5bit
        rt = (instruction >> 16) & 0x1F  # 5bit
        rd = (instruction >> 11) & 0x1F  # 5bit
        shamt = (instruction >> 6) & 0x1F  # 5bit
        funct = (instruction) & 0x3F  # 6bit
        imm = (instruction) & 0xFFFF  # 16bit
        address = (instruction) & 0x3FFFFFF  # 26bit
        return op, rs, rt, rd, shamt, funct, imm, address
```

```python
# EXECUTE STAGE
def execute(self, op, rs, rt, rd, shamt, funct, imm, address):
    if op == 35:  # lw
        address = self.reg[rs] + imm
        data = self.data_mem[address]
        return data

    elif op == 43:  # sw
        address = self.reg[rs] + imm
        data = self.reg[rt]
        self.data_mem[address] = data

    elif op == 15:  # lui
        address = (imm << 16)
        return address
```

```
    elif op == 0:  # R-type
        if funct == 32:  # add
            return self.reg[rs] + self.reg[rt]
        elif funct == 34:  # sub
            return self.reg[rs] - self.reg[rt]
        elif funct == 36:  # and
            return self.reg[rs] & self.reg[rt]
        elif funct == 37:  # or
            return self.reg[rs] | self.reg[rt]
        elif funct == 42:  # slt
            if self.reg[rs] < self.reg[rt]:
                return 1
            else:
                return 0
        elif funct == 0:  # sll
            return self.reg[rt] << shamt
        elif funct == 2:  # srl
            return self.reg[rt] >> shamt

    elif op == 28 and funct == 2:  # mul
        return self.reg[rs] * self.reg[rt]
```

```
    elif op == 8:  # addi
        return self.reg[rs] + imm
    elif op == 12:  # andi
        return self.reg[rs] & imm
    elif op == 13:  # ori
        return self.reg[rs] | imm

    elif op == 4:  # beq
        if self.reg[rs] == self.reg[rt]:
            self.pc += (imm * 4)

    elif op == 5:  # bne
        if self.reg[rs] != self.reg[rt]:
            self.pc += (imm * 4)

    elif op == 2:  # j
        self.pc = (self.pc & 0xF0000000) | (address << 2) #adding 4 sign
            digits at the beginning and 2 0s at the end to make 26 bit
            addresss to 32 bit
    return
```

```
    # MEMORY ACCESS STAGE
    def memory_access(self, result):
        return result

    # WRITE BACK STAGE
    def write_back(self, op, data, rd, rt):
        if (op == 0 or op == 28):  # for R-type instructions
            self.reg[rd] = data
        else: # for I-type instructions
            self.reg[rt] = data

    def run(self):
        while self.pc < 4194304 + (4 * len(self.instr_mem)): #condition for
            instruction = self.fetch()
            op, rs, rt, rd, shamt, funct, imm, address = self.decode
                (instruction)
            result = self.execute(op, rs, rt, rd, shamt, funct, imm, address)
            if result is not None:
                data = self.memory_access(result)
                self.write_back(op, data, rd, rt)
```

```python
    def reg_name(self, reg_num):
        if reg_num == 0:
            return "0"
        elif reg_num == 31:
            return "ra"
        elif reg_num == 1:
            return "at"
        elif 2 <= reg_num <= 3:
            return f"v{reg_num - 2}"
        elif 4 <= reg_num <= 7:
            return f"a{reg_num - 4}"
        elif 8 <= reg_num <= 15:
            return f"t{reg_num - 8}"
        elif 16 <= reg_num <= 23:
            return f"s{reg_num - 16}"
        elif 24 <= reg_num <= 25:
            return f"t{reg_num - 24 + 2}"
        elif 26 <= reg_num <= 27:
            return f"k{reg_num - 26}"
        elif 28 <= reg_num <= 31:
            return f"g{reg_num - 28}"
```

```python
    def print_reg(self):
        print("Registers:")
        for i, value in enumerate(self.reg):
            reg_name = self.reg_name(i)
            print(f"${reg_name}: {value}")
        print()
```

We have taken the inputs instruction memory and data memory as shown from the machine data generated by MARS simulator:

MARS simulator machine data:

```
 Address    Code        Basic                    Source

4194304  0x3c011001  lui $1,4097          6           lw $t0, n
4194308  0x8c280000  lw $8,0($1)
4194312  0x20090001  addi $9,$0,1         7           addi $t1, $zero, 1
4194316  0x11000004  beq $8,$0,4         10           beq $t0, $zero, store
4194320  0x71284802  mul $9,$9,$8        11           mul $t1, $t1, $t0
4194324  0x20010001  addi $1,$0,1        12           subi $t0, $t0, 1
4194328  0x01014022  sub $8,$8,$1
4194332  0x08100003  j 4194316           13           j loop
4194336  0x3c011001  lui $1,4097         15           sw $t1, fac
4194340  0xac290004  sw $9,4($1)
```

Instruction memory and data memory taken in processor:

```
Instruction_Memory_fac = {
    4194304: 0x3c011001,  # lui $a1,4097
    4194308: 0x8c280000,  # lw $t0,0($a1)
    4194312: 0x20090001,  # addi $t1,$0,1
    4194316: 0x11000004,  # beq $t0,$0,4
    4194320: 0x71284802,  # mul $t1,$t1,$t0
    4194324: 0x20010001,  # addi $a1,$0,1
    4194328: 0x01014022,  # sub $t0,$t0,$a1
    4194332: 0x08100003,  # j 4194316
    4194336: 0x3c011001,  # lui $a1,4097
    4194340: 0xac290004  # sw $t1,4($a1)
}
```

```
Data_Memory_fac = {268500992: 5, 268500996: 0}
```