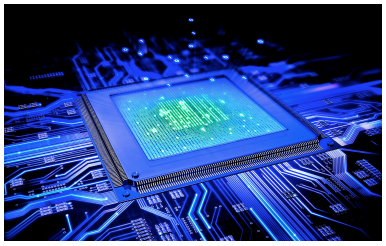


Processor Assignment-2

IMT2022025-Varnit Mittal
IMT2022076-Mohit Naik

November 2023

1 Introduction

This project is done as part of the Computer Architecture Course under Prof. Nanditha Rao. This assignment has two parts to it. The first part is a non-pipelined simulator which is written in CPP. The processor simply takes binary machine code as input from a file and then writes the output to another file. The output contains the total number of clock cycles, all the non-zero values registers with their contents and all the non-zero valued data memory locations with their content. Our processor is word-addressable and not byte-addressable.

The second part of the assignment is written in Python which is a pipelined processor simulator which simulates the working of an actual MIPS pipelined processor and gives the total number of clock cycles required to run a set of instructions provided to the simulator through a file which contains binary instructions set. The output file also contains the desired output which your MIPS program intends to do.

Both the simulators support limited instructions(add, addi, sub, slt, lw, sw, beq, jmp) and ALU can only perform addition, subtraction and comparison. Multiplication and division can also be performed if the instruction set contains the corresponding addition and subtraction logic. The comparisons are done just like a real MIPS processor using a zero register.

2 Assumptions Made

⇒ Memory is integer aligned, and integer indexed i.e instructions and data occupy one location, represented by integers instead of hexadecimal. The PC therefore increments by 1, not 4.

⇒ The instruction memory is variable size, while the data memory is 100 locations wide.

⇒ The register file also contains integer values instead of 32-bit binary data.

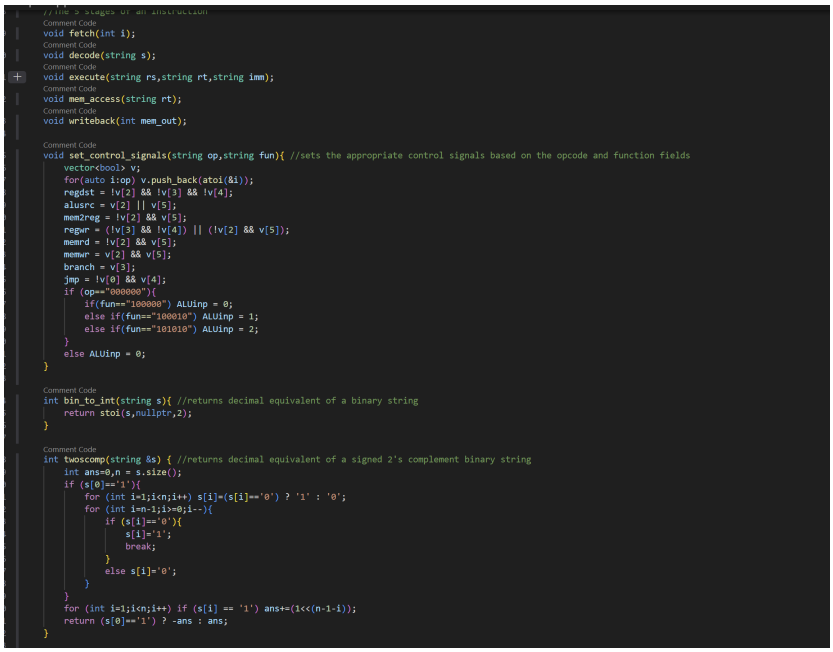
⇒ The ALU control unit in the original MIPS processor is replaced with a single *alucon* signal.

⇒ The output will display the total number of clock cycles, the total number of instructions fetched, and all nonzero registers and nonzero data locations.

⇒ The instruction memory, data memory, register file, all ports, program counter, clock, and all control signals are declared as global variables.

3 Code Explanation

3.1 Decoder logic



```
//the 5 stages of an instruction
Comment Code
void fetch(int i);
Comment Code
void decode(string s);
Comment Code
void execute(string rs,string rt,string imm);
Comment Code
void mem_access(string rt);
Comment Code
void writeback(int mem_out);

Comment Code
void set_control_signals(string op,string fun){ //sets the appropriate control signals based on the opcode and function fields
    vector<bool> v;
    for(auto i:op) v.push_back(atoi(&i));
    regdst = !v[2] && !v[3] && !v[4];
    alusrc = v[2] || v[5];
    mem2reg = !v[2] && v[5];
    regwr = (!v[3] && !v[4]) || (!v[2] && v[5]);
    memrd = !v[2] && v[5];
    memwr = v[2] && v[5];
    branch = v[3];
    jmp = !v[0] && v[4];
    if (op=="000000"){
        if(fun=="100000") ALUinp = 0;
        else if(fun=="100010") ALUinp = 1;
        else if(fun=="101010") ALUinp = 2;
    }
    else ALUinp = 0;
}

Comment Code
int bin_to_int(string s){ //returns decimal equivalent of a binary string
    return stoi(s,nullptr,2);
}

Comment Code
int twoscomp(string &s) { //returns decimal equivalent of a signed 2's complement binary string
    int ans=0,n = s.size();
    if (s[0]=='1'){
        for (int i=1;i<n;i++) s[i]=(s[i]=='0') ? '1' : '0';
        for (int i=n-1;i>=0;i--){
            if (s[i]=='0'){
                s[i]='1';
                break;
            }
            else s[i]='0';
        }
    }
    for (int i=1;i<n;i++) if (s[i] == '1') ans+=(1<<(n-1-i));
    return (s[0]=='1') ? -ans : ans;
}
```

Figure 1: Decoder

This is the main logic behind the decoder which sets the necessary control signals in order to load, store, and perform operations on registers with the help of ALU or writing back to a register using the opcode of the instructions. The *digital logic* used to set control signals is made using K-maps using the truth table taken directly from *Hennessy and Patterson*.
The decoder also decodes the values of registers, immediate field, address field, and opcode and uses them accordingly to the control signals generated. The *regdst* control signal decides the destination register.

In the pipelined processor, the decode phase also handles RAW hazards or load-word hazards.

3.2 Arithmetic Logic Unit

This part of the code simulates the ALU of an actual MIPS processor and passes the results to the memory phase.

```
Executes the instruction in the Ex stage of the pipeline and updates the Ex/Mem pipeline register accordingly.
"""

reg1 = IdEx['reg1']
reg2 = IdEx['reg2']
offset = IdEx['offset']
p1 = IdEx['pc']

if(ExMem['writeport']==IdEx['reg1'] and len(instMem)!=1):
    if IdEx['alusrc']:
        Aluresult = ExMem['Aluresult']+offset
    else:
        Aluresult = ExMem['Aluresult']+RegFile[reg2]

    if(IdEx['branch']):
        zero = ExMem['Aluresult']-RegFile[reg2]      You, 24 seconds ago • Uncommitted changes
        ExMem['zero']=zero

    if(IdEx['Aluinp']==1):
        Aluresult = ExMem['Aluresult']-RegFile[reg2]

    elif(IdEx['Aluinp']==2):
        if(ExMem['Aluresult']<RegFile[reg2]):
            Aluresult=1
        else:
            Aluresult=0

elif(ExMem['writeport']==IdEx['reg2'] and len(instMem)!=1):
    if IdEx['alusrc']:
        Aluresult = RegFile[reg1]+offset
    else:
        Aluresult = RegFile[reg1]+ExMem['Aluresult']

    if(IdEx['branch']):
        zero = RegFile[reg1]-ExMem['Aluresult']
        ExMem['zero']=zero

    if(IdEx['Aluinp']==1):
        Aluresult = RegFile[reg1]-ExMem['Aluresult']
    elif(IdEx['Aluinp']==2):
        if(RegFile[reg1]<ExMem['Aluresult']):
            Aluresult=1
        else:
            Aluresult=0
```

Figure 2: ALU

The *execute* function is where the ALU performs addition or comparison, and the *beq* and *jmp* instructions are also handled here using appropriate control signals. If the instruction is not a *beq* or *jmp*, the PC simply increments by 1.

In the pipelined processor simulator, the execute phase also has a *forwarding unit* which handles all the cases related to data hazards.

3.3 Memory

```
def memory():
    """
    This function performs memory operations based on the input flags from the ExMem stage.
    It reads from or writes to the data memory based on the memrd and memwr flags respectively.
    It updates the MemWb register with the appropriate values for the next stage.
    If the pcsrc or jmp flags are set, it returns without executing the next stage.
    """
    mem_out = 0
    if(ExMem['memrd']):
        mem_out = dataMem[ExMem['Aluresult']]
    if(ExMem['memwr']):
        dataMem[ExMem['Aluresult']] = RegFile[ExMem['reg2']]

    MemWb['AluResult'] = ExMem['Aluresult']
    MemWb['MemOut'] = mem_out

    MemWb['pc'] = ExMem['pc']
    MemWb['pcsrc']=ExMem['pcsrc']
    MemWb['regdst'] = ExMem['regdst']
    MemWb['alusrc'] = ExMem['alusrc']
    MemWb['mem2reg'] = ExMem['mem2reg']
    MemWb['regwr'] = ExMem['regwr']
    MemWb['memrd'] = ExMem['memrd']
    MemWb['memwr'] = ExMem['memwr']      You, 14 minutes ago • Uncommitted changes
    MemWb['branch'] = ExMem['branch']
    MemWb['aluop1'] = ExMem['aluop1']
    MemWb['aluop0'] = ExMem['aluop0']
    MemWb['jmp'] = ExMem['jmp']
    MemWb['writeport'] = ExMem['writeport']

    if(MemWb['pcsrc'] or MemWb['jmp']):
        return
    execute()
```

Figure 3: Memory Phase

In non-pipelined processor simulator, The *execute* function calls the *memAccess* function, which reads from or writes to memory. Again, this is controlled by the *memrd* and *memwr* control signals. In the pipelined simulator, the memory phase also handles pipeline flush in case of *control hazards* during *jump* and *branch* type of instructions.

3.4 Writeback

```
def writeback():
    """
    Writes the data back to the register file if MemWb['regwr'] is True.
    If MemWb['mem2reg'] is True, writes MemWb['MemOut'] to the register file at MemWb['writeport'].
    Otherwise, writes MemWb['AluResult'] to the register file at MemWb['writeport'].
    Calls memory() function.
    """
    if(MemWb['regwr']):
        if(MemWb['mem2reg']):
            RegFile[MemWb['writeport']] = MemWb['MemOut']
        else:
            RegFile[MemWb['writeport']] = MemWb['AluResult']
    memory()
```

Figure 4: Writeback

In both pipelined and non-pipelined simulators, the function writes the data back to the register file if *regwr* is true. If *mem2reg* is true, write data to *rd* register. Otherwise, writes to *rt* register.

4 Results

```
Total clock cycles = 130
Total instructions fetched = 26
Nonzero registers :
$t0 = 1
$t1 = 1
$t2 = 8
$t3 = 3
$t4 = 8
$s1 = 3
$s2 = 15
$s4 = 1
Nonzero data memory locations :
0 : 9
1 : 7
2 : 3
3 : 15
4 : 1
5 : 99
6 : 10
7 : 6
8 : 1
```

Figure 5: Non-Pipeline Output File

```
No. of clock cycles are: 742
The Registers with non-zero values are:
$8 = 1
$9 = 8
$10 = 8
$11 = 8
$17 = 15
$18 = 99
$20 = 1
The Data Memory locaations with non-zero values are:
dataMem[0] = 1
dataMem[1] = 1
dataMem[2] = 3
dataMem[3] = 6
dataMem[4] = 7
dataMem[5] = 9
dataMem[6] = 10
dataMem[7] = 15
dataMem[8] = 99
```

Figure 6: Pipeline output file

5 Authors

1. Varnit Mittal (IMT2022025)
2. Mohit Naik (IMT2022076)