# Advanced Computer Architecture

## Computer Assignment 1: MIPS Pipeline

**Amirmahdi Joudi: 810101325** **Negin Safari: 810101339**

## Abstract:

In this assignment, a pipeline design of MIPS processor with support a subset of instructions is designed. This processor has 5 stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). It also supports forwarding and hazard unit.

## Description:

The datapath for this design is shown in Figure 1. Instructions and Data memories are considered independent, so there is not memory access conflict. To prevent pipeline from stalling because of branch instructions, a comparator for branch condition check is put in ID stage to calculate equality earlier, so hazard unit stalls pipeline just for memory read dependency. Instructions supported by this design are shown in Table 1. The controller of this design is a simple lookup table that searches 'opcode' and 'func' bits to issue specific controlling signals. There is also an ALU controller that uses decoded opcodes and 'func' bits to do computations, like Figure 2.

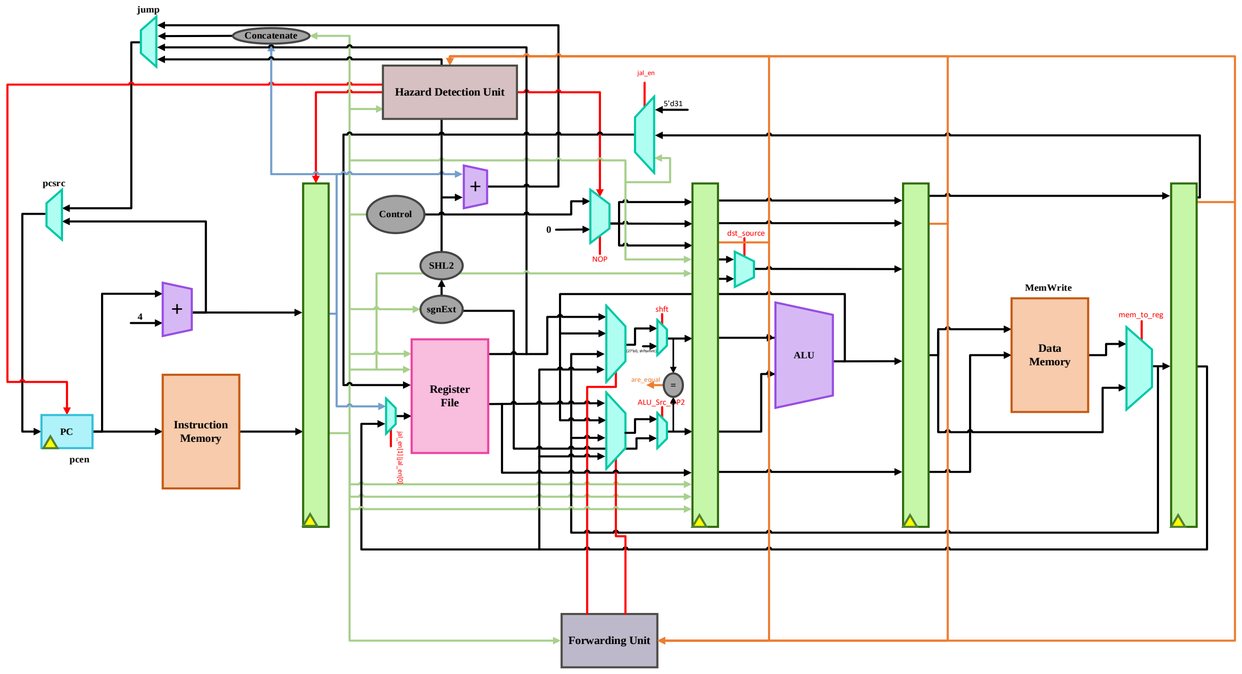
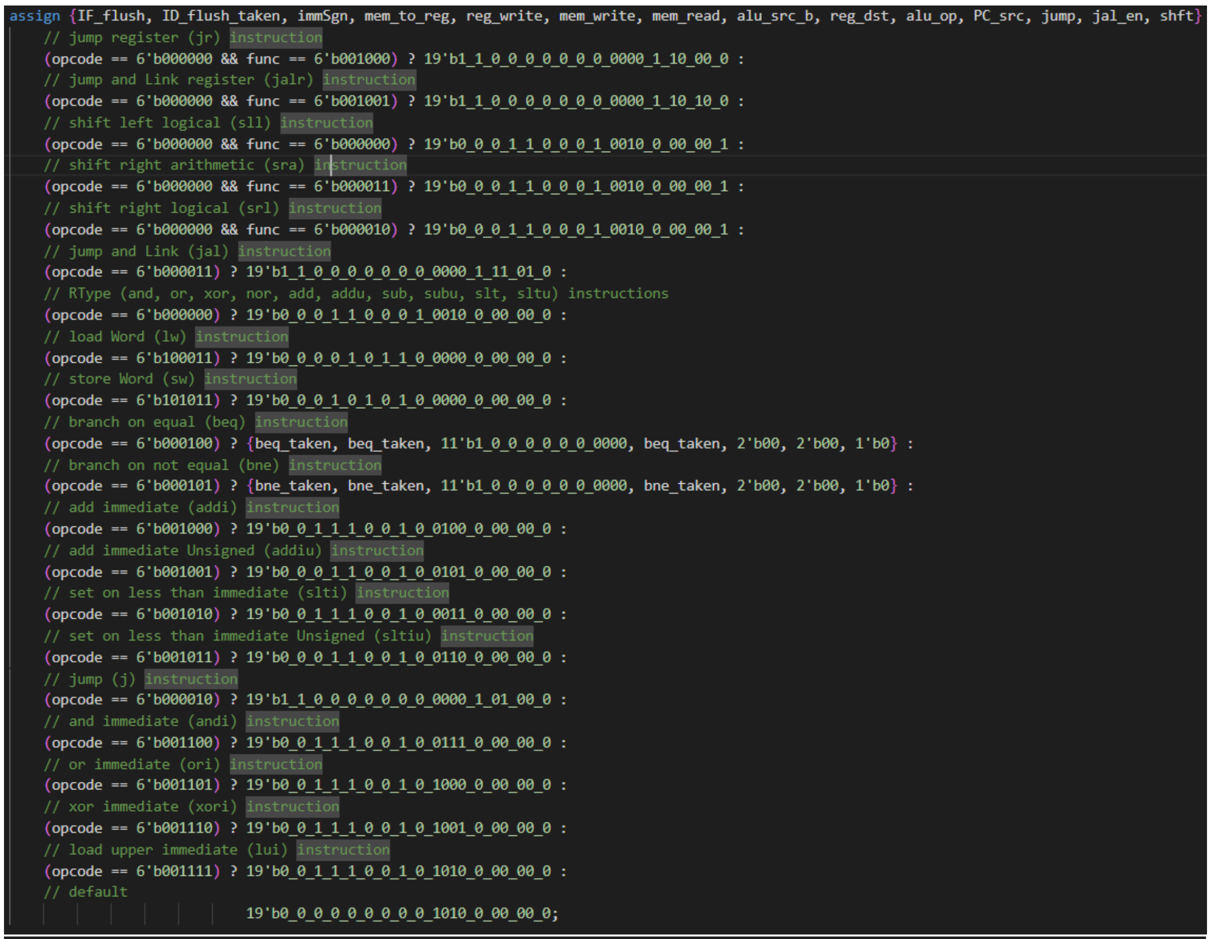
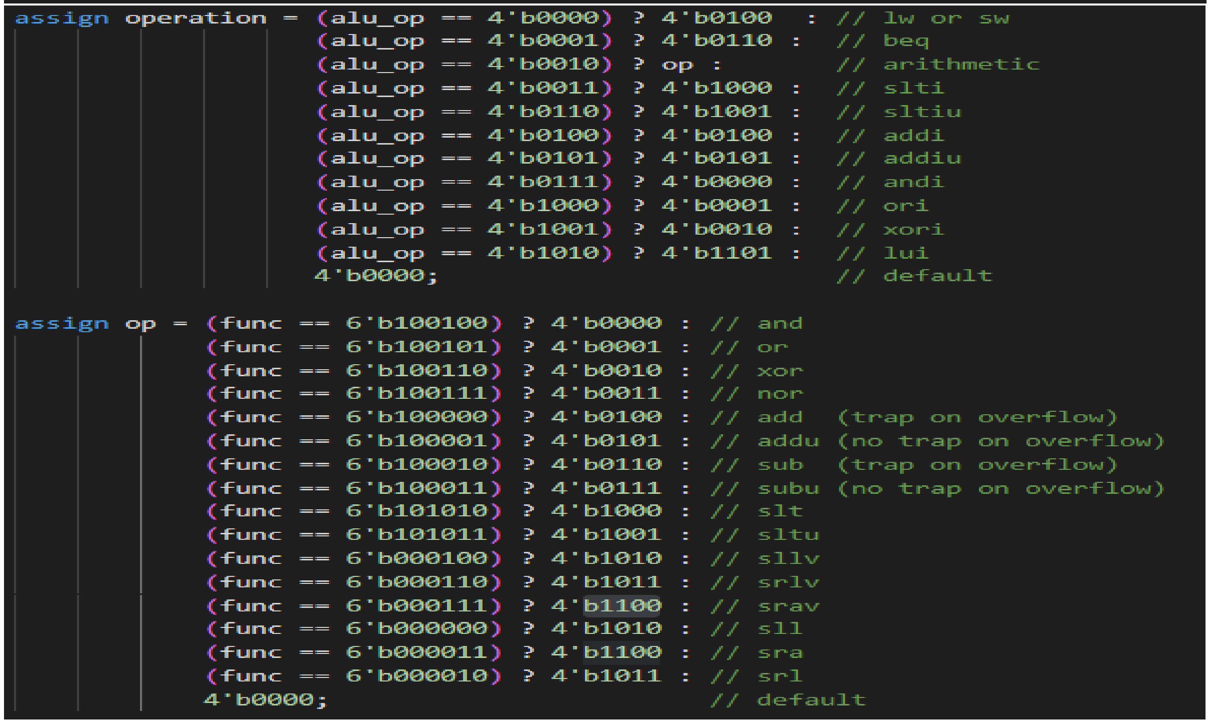


Figure 1. MIPS pipeline datapath



```
assign {IF_flush, ID_flush_taken, immSgn, mem_to_reg, reg_write, mem_write, mem_read, alu_src_b, reg_dst, alu_op, PC_src, jump, jal_en, shft}
    // jump register (jr) instruction
    (opcode == 6'b000000 && func == 6'b001000) ? 19'b1_1_0_0_0_0_0_0_0_0000_1_10_00_0 :
    // jump and Link register (jalr) instruction
    (opcode == 6'b000000 && func == 6'b001001) ? 19'b1_1_0_0_0_0_0_0_0_0000_1_10_10_0 :
    // shift left logical (sll) instruction
    (opcode == 6'b000000 && func == 6'b000000) ? 19'b0_0_0_1_1_0_0_0_1_0010_0_00_00_1 :
    // shift right arithmetic (sra) instruction
    (opcode == 6'b000000 && func == 6'b000011) ? 19'b0_0_0_1_1_0_0_0_1_0010_0_00_00_1 :
    // shift right logical (srl) instruction
    (opcode == 6'b000000 && func == 6'b000010) ? 19'b0_0_0_1_1_0_0_0_1_0010_0_00_00_1 :
    // jump and Link (jal) instruction
    (opcode == 6'b000011) ? 19'b1_1_0_0_0_0_0_0_0_0000_1_11_01_0 :
    // RType (and, or, xor, nor, add, addu, sub, subu, slt, sltu) instructions
    (opcode == 6'b000000) ? 19'b0_0_0_1_1_0_0_0_1_0010_0_00_00_0 :
    // load Word (lw) instruction
    (opcode == 6'b100011) ? 19'b0_0_0_0_1_0_1_1_0_0000_0_00_00_0 :
    // store Word (sw) instruction
    (opcode == 6'b101011) ? 19'b0_0_0_1_0_1_0_1_0_0000_0_00_00_0 :
    // branch on equal (beq) instruction
    (opcode == 6'b000100) ? {beq_taken, beq_taken, 11'b1_0_0_0_0_0_0_0000, beq_taken, 2'b00, 2'b00, 1'b0} :
    // branch on not equal (bne) instruction
    (opcode == 6'b000101) ? {bne_taken, bne_taken, 11'b1_0_0_0_0_0_0_0000, bne_taken, 2'b00, 2'b00, 1'b0} :
    // add immediate (addi) instruction
    (opcode == 6'b001000) ? 19'b0_0_1_1_1_0_0_1_0_0100_0_00_00_0 :
    // add immediate Unsigned (addiu) instruction
    (opcode == 6'b001001) ? 19'b0_0_0_1_1_0_0_1_0_0101_0_00_00_0 :
    // set on less than immediate (slti) instruction
    (opcode == 6'b001010) ? 19'b0_0_1_1_1_0_0_1_0_0011_0_00_00_0 :
    // set on less than immediate Unsigned (sltiu) instruction
    (opcode == 6'b001011) ? 19'b0_0_0_1_1_0_0_1_0_0110_0_00_00_0 :
    // jump (j) instruction
    (opcode == 6'b000010) ? 19'b1_1_0_0_0_0_0_0_0_0000_1_01_00_0 :
    // and immediate (andi) instruction
    (opcode == 6'b001100) ? 19'b0_0_1_1_1_0_0_1_0_0111_0_00_00_0 :
    // or immediate (ori) instruction
    (opcode == 6'b001101) ? 19'b0_0_1_1_1_0_0_1_0_1000_0_00_00_0 :
    // xor immediate (xori) instruction
    (opcode == 6'b001110) ? 19'b0_0_1_1_1_0_0_1_0_1001_0_00_00_0 :
    // load upper immediate (lui) instruction
    (opcode == 6'b001111) ? 19'b0_0_1_1_1_0_0_1_0_1010_0_00_00_0 :
    // default
                          19'b0_0_0_0_0_0_0_0_0_1010_0_00_00_0;
```



```
assign operation = (alu_op == 4'b0000) ? 4'b0100  : // lw or sw
                   (alu_op == 4'b0001) ? 4'b0110 :  // beq
                   (alu_op == 4'b0010) ? op :       // arithmetic
                   (alu_op == 4'b0011) ? 4'b1000 :  // slti
                   (alu_op == 4'b0110) ? 4'b1001 :  // sltiu
                   (alu_op == 4'b0100) ? 4'b0100 :  // addi
                   (alu_op == 4'b0101) ? 4'b0101 :  // addiu
                   (alu_op == 4'b0111) ? 4'b0000 :  // andi
                   (alu_op == 4'b1000) ? 4'b0001 :  // ori
                   (alu_op == 4'b1001) ? 4'b0010 :  // xori
                   (alu_op == 4'b1010) ? 4'b1101 :  // lui
                   4'b0000;                         // default

assign op = (func == 6'b100100) ? 4'b0000 : // and
            (func == 6'b100101) ? 4'b0001 : // or
            (func == 6'b100110) ? 4'b0010 : // xor
            (func == 6'b100111) ? 4'b0011 : // nor
            (func == 6'b100000) ? 4'b0100 : // add  (trap on overflow)
            (func == 6'b100001) ? 4'b0101 : // addu (no trap on overflow)
            (func == 6'b100010) ? 4'b0110 : // sub  (trap on overflow)
            (func == 6'b100011) ? 4'b0111 : // subu (no trap on overflow)
            (func == 6'b101010) ? 4'b1000 : // slt
            (func == 6'b101011) ? 4'b1001 : // sltu
            (func == 6'b000100) ? 4'b1010 : // sllv
            (func == 6'b000110) ? 4'b1011 : // srlv
            (func == 6'b000111) ? 4'b1100 : // srav
            (func == 6'b000000) ? 4'b1010 : // sll
            (func == 6'b000011) ? 4'b1100 : // sra
            (func == 6'b000010) ? 4'b1011 : // srl
            4'b0000;                        // default
```

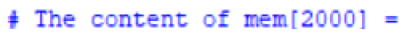
*Figure 2. Controller codes*

Table 1. Bolded instructions are implemented in this design

| | |
|---|---|
| **R-Type** | **add, addu, sub, subu, slt, sltu, and, or, xor, nor, sll, srl, sla, sllv, arlv, slav, jr, jalr,** mult, multu, div, divu, mfhi, mthi, mflo, mtlo |
| **I-Type** | **addi, addiu, slti, sltiu, andi, ori, xori, lui, lw, sw, beq, bne** |
| **J-Type** | **j, jal** |
| FR-Type | add.s, sub.s, mul.s, div.s, abs.s, neg.s, c.eq.s, c.lt.s, c.le.s<br>add.d, sub.d, mul.d, div.d, abs.d, neg.d, c.eq.d, c.lt.d, c.le.d |
| FI-Type | l.s, s.s, l.d, s.d, bc1t, bc1f |

## Tests:

All codes are available in folder 'codes'. In memory section there is a text file named 'TEST_INST.txt' including one or more samples for each instruction and their expected results. This can be used for instructions testing. There is another file named 'inst.txt' which is a program that reads 10 signed words from address 1000 in memory (file 'data.txt') and writes their largest one in location 2000 in memory. This location will be printed at the end, like Figure 3. Macro for waves named 'waves.do' is saved in project folder and can be loaded to waveform window. Figure 4 shows some signals from ALU and RefisterFile units.



```
# The content of mem[2000] =          73
```

Figure 3. Maximum value in array, written in location 2000

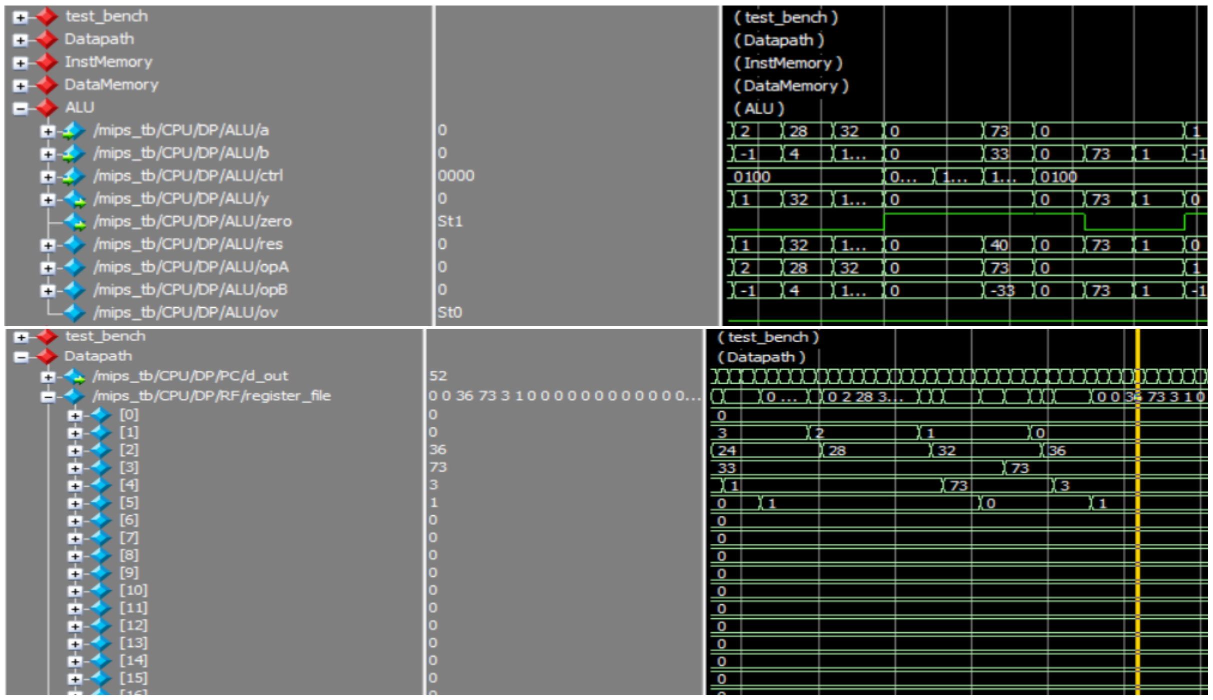

test_bench
Datapath
InstMemory
DataMemory
ALU
/mips_tb/CPU/DP/ALU/a 0
/mips_tb/CPU/DP/ALU/b 0
/mips_tb/CPU/DP/ALU/ctrl 0000
/mips_tb/CPU/DP/ALU/y 0
/mips_tb/CPU/DP/ALU/zero St1
/mips_tb/CPU/DP/ALU/res 0
/mips_tb/CPU/DP/ALU/opA 0
/mips_tb/CPU/DP/ALU/opB 0
/mips_tb/CPU/DP/ALU/ov St0
test_bench
Datapath
/mips_tb/CPU/DP/PC/d_out 52
/mips_tb/CPU/DP/RF/register_file 0 0 36 73 3 1 0 0 0 0 0 0 0 0 0 0 0...
[0] 0
[1] 0
[2] 36
[3] 73
[4] 3
[5] 1
[6] 0
[7] 0
[8] 0
[9] 0
[10] 0
[11] 0
[12] 0
[13] 0
[14] 0
[15] 0

Figure 4. ALU and RegiterFile waveforms

**Summary:**

This design is a 5-stage pipeline MIPS processor with limited instruction, supporting forwarding and hazard handling for implemented instructions. IF stage reads instructions, ID decodes instruction, issues them for next stages, EX does arithmetic and logical computations, MEM does data memory read and write access, and WB commits an instruction.