# Report of Assignment 4

Ang, Chen

118010009

The Chinese University of Hong Kong, Shenzhen

## Introduction

In this project, we implemented a 5-stage (5 clock cycles) pipeline MIPS architecture processor using Verilog HDL. The CPU contains 8 general registers and supports 26 basic MIPS instructions via ALU. Data can be transferred in and out of the simulated memory and the general registers. Branch/jump of instructions are supported via management of PC. This CPU does not handle any pipeline hazard.

## Five Stages

The five stages of the the pipeline CPU are:

1. **F**etch
   1. PC updated
   2. New instruction fetched from instruction memory at PC
2. **D**ecode
   1. Instruction split and decoded by the control unit
   2. Sign extension
3. **E**xecute
   1. ALU executes the operation
4. **M**emory I/O
5. **W**riteback
   1. Result written back into the register file

## Block Diagram

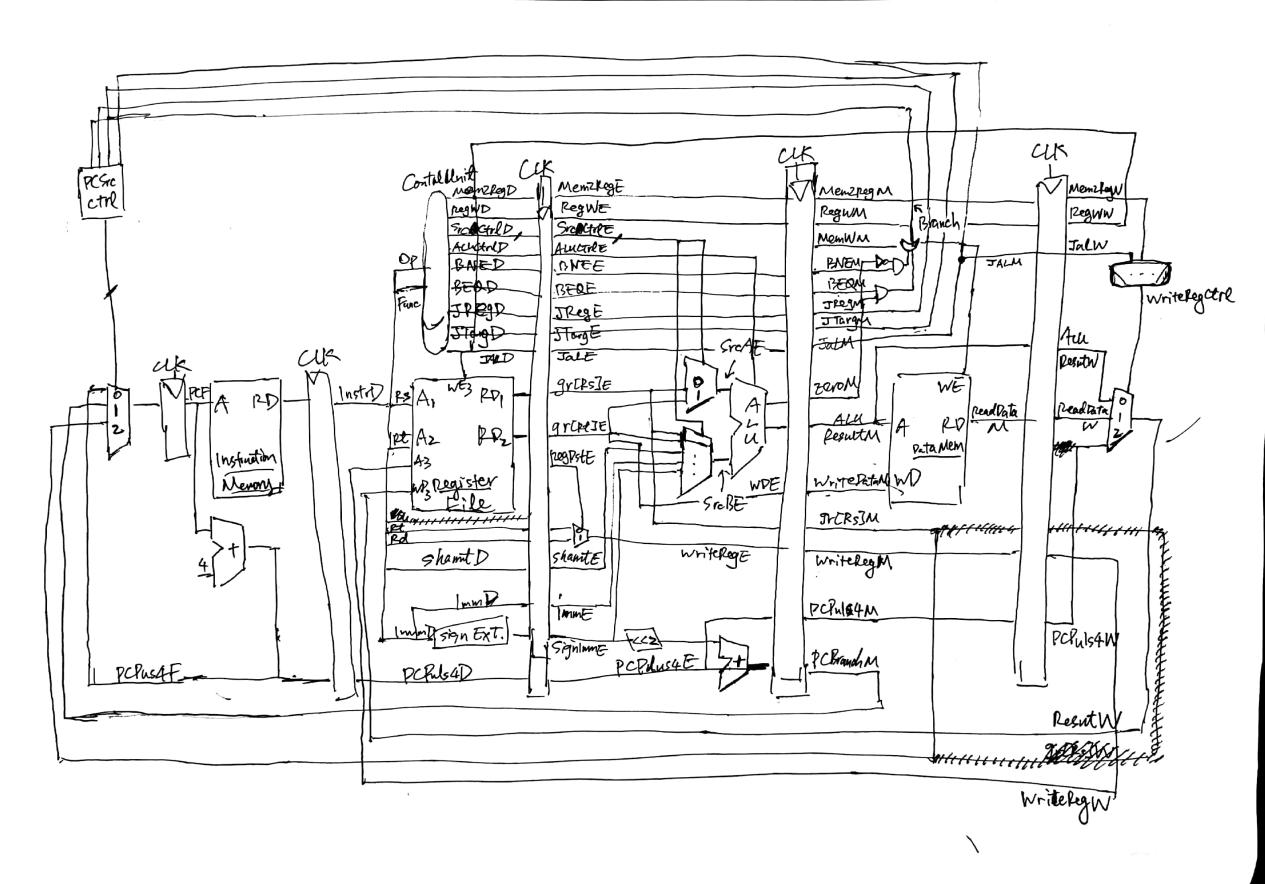


Fig. 1. *Block diagram of the MIPS architecture processor without hazard handling*

## Implementation Details

### Pipelining

The pipeling behavior of the CPU is implemented via writing sequential logic in Verilog. Specifically, each control signal CS is treated as a vector with entries corresponding to the stages in which the signal is used for. And we used the always block to advance the signals as follows:

always @(posedge CLK) begin

CS[1] <= CS[0];

CS[2] <= CS[1];

CS[3] <= CS[2];

//...

end

Logic within each stage is purely combinational and they are put in a separate always block, for example:

always @(\*) begin

PCPlus4[0] = PC + 4;

PCBranch[0] = PCPlus4[2] + (signImm[1] << 2);

opcode = instr[31 : 26];

funct = instr[05 : 00];

//...

end

### Memory and Register I/O

Both data/instruction memory and eight general registers are implemented as Verilog regs:

// MEMORY I/O

reg[31 : 0] dataMemory[10000:0];

reg[31 : 0] instrMemory[1000:0];

// REGISTER I/O

reg[31 : 0] gr[7:0];

Then both memory and registers can be accessed using indices. One thing worth noting is that dataMemory/instrMemory should not be accessed directly via the address as it is treated as a 32-bit reg (rather than byte-indexed). Therefore we must divide the address by four to obtain the correct index of the desired word in dataMemory/instrMemory.

In the Fetch stage, instructions are fetched for decoding through

instr = instrMemory[PC / 4];

Using rs, rt from the split instruction we can read data stored in the registers:

gr[rs]; gr[rt];

In the Memory stage, data can be pulled out:

readData = dataMemory[ALUResult / 4];

or written into dataMemory:

If (memWrite) dataMemory[ALUResult / 4] = writeData;

Finally in the Writeback stage, the result can be written back to register file:

if (regWrite) gr[writeReg] = result;

### Control Unit

The control unit processes the opcode and function codes during the Decode stage which would eventually tell the ALU which operation to perform through ALUctrl. Other control signals are generated as well, including

* srcActrl/srcBctrl(0~4), which determine the source of operand A/B;
* branchEQ & branchNE, which, combined with ZeroFlag, control loading branch target into PC.
* JReg/JTarg, which control the source of the jump target;
* memToReg/Jal, which determine data source to be written into the register;
* memWrite/regWrite, which enables data write into memory/register;
* regDst, which determines the destination register to write data into.

These signals are generated within the case block (upon the opcode and function code of the instruction) of Verilog. Some of the control signals of each instruction are shown below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Instr* | *Operation* | *ALUctrl* | *srcActrl* | *srcBctrl* | *regDst* |
| add | + | 0 | rs (1) | rt (0) | rd (1) |
| addu |
| addi | signImm (3) | rt (0) |
| addiu |
| lw |
| sw |
| sub | - | 1 | rt (0) | rd (1) |
| subu |
| beq | x |
| bne |
| and | & | 2 |
| andi | imm (2) | rt (0) |
| nor | ~| | 3 | rt (0) | rd (1) |
| or | | | 4 |
| ori | imm (2) | rt (0) |
| xor | ^ | 5 | rt (0) | rd (1) |
| xori | imm (2) | rt (0) |
| slt | slt (±) | 6 | rt (0) | rd (1) |
| slti |
| sltu | slt (Ø) | 7 |
| sltiu |
| sll | << | 8 | rt (0) | shamt (4) |
| sllv | rs (1) |
| srl | >> | 9 | shamt (4) |
| srlv | rs (1) |
| sra | >>> | 10 | shamt (4) |
| srav | rs (1) |

±: signed; Ø: unsigned

Table 1. *Grouping of supported instructions by operations(ALUctrl.) srcA, srcB, regDst and their encoding listed.*

### Main ALU

The main ALU conducts the operation encodes by ALUctrl on two input operands. Thus ALU needs to first decode the ALUctrl. This decoding is again realized using the case instruction. What remains is to simply to tell the ALU to conduct the basic operations listed above as most of these binary operations are built into the Verilog. Some operations among them do need to be distinguished between signed and unsigned versions, while ***regular operators in Verilog are by default unsigned*** . Luckily the Verilog has an built-in function $signed() which handles this scenario for us.

By directly converting both operands a, b into $signed(a) and $signed(b), any operation X on them will automatically become signed as well. Another type of operation worth mentioning is the shifting. Sometimes when the shift amount is indicated by a register as in sllv or srav, the amount for shifting could be too wide (>32) and cause an error. This is handled by reducing the shift amount modulo 32, or equivalently, anding it with 0x1F, which prevents undefined shifting behavior.

### Branch and Jump (PC management)

PC is updated at each clock either to PCPlus4 from stage F (in normal cases), or PCBranch from stage M (for successful branches), which is

PCBranch = PCPlus4 + (signImm << 2);

or PCReg from stage M (for jr), or PCTarg from stage M (for j and jal). The source of the PC is controlled by three control signals: branchCtrl, jumpReg, and jumpTarg. The branchCtrl indicates a successful branching condition, namely,

branchCtrl = (branchEq & zeroFlag) | (branchNE & !zeroFlag);

enabling the source PCBranch. (Otherwise it would be normal increment of PC by 4). For jump instructions, jumpReg and jumpTarg enables PCReg and PCTarg, respectively.

### Examples

**Input:**

// Data Memory

uut.dataMemory[0] = 32'h0000\_00ab; // address 0x00

uut.dataMemory[1] = 32'h0000\_3c00; // address 0x04

// Instruction Memory

uut.instrMemory[0] = {6'b100011, `gr0, `gr1, 16'h0000}; // lw gr1, gr0(0) (gr1 <= memory[0x00])

uut.instrMemory[1] = {6'b100011, `gr0, `gr2, 16'h0004}; // lw gr2, gr0(4) (gr2 <= memory[0x04])

uut.instrMemory[2] = {6'b101011, `gr0, `gr0, 16'h0008}; // sw gr0, gr0(8) (data[0x08] <= 0)

uut.instrMemory[3] = {6'b001000, `gr0, `gr3, 16'h0001}; // addi gr3, gr0, 1 (gr3 <= 0 + 1)

uut.instrMemory[4] = {6'b001001, `gr0, `gr3, 16'h0002}; // addiu gr3, gr0, 2 (gr3 <= 0 + 2)

uut.instrMemory[5] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100000}; // add gr3, gr1, gr2 (gr3 <= gr1 + gr2)

uut.instrMemory[6] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100010}; // sub gr3, gr1, gr2 (gr3 <= gr1 - gr2)

uut.instrMemory[7] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100001}; // addu gr3, gr1, gr2 (gr3 <= gr1 + gr2)

uut.instrMemory[8] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100011}; // subu gr3, gr1, gr2 (gr3 <= gr1 - gr2)

uut.instrMemory[9] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100100}; // and gr3, gr1, gr2 (gr3 <= gr1 & gr2)

uut.instrMemory[10] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100101}; // or gr3, gr1, gr2 (gr3 <= gr1 | gr2)

uut.instrMemory[11] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100111}; // nor gr3, gr1, gr2 (gr3 <= gr1 ~| gr2)

uut.instrMemory[12] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100110}; // xor gr3, gr1, gr2 (gr3 <= gr1 ^ gr2)

uut.instrMemory[13] = {6'b001100, `gr0, `gr3, 16'h1111}; // andi gr3, gr0, 0x1111 (gr3 <= 0 & 0x1111)

uut.instrMemory[14] = {6'b001101, `gr0, `gr3, 16'h1111}; // ori gr3, gr0, 0x1111 (gr3 <= 0 | 0x1111)

uut.instrMemory[15] = {6'b000000, `gr0, `gr1, `gr3, 5'b00001, 6'b000000}; // sll gr3, gr1, 1 (gr3 <= gr1 << 1)

uut.instrMemory[16] = {6'b000000, `gr0, `gr1, `gr3, 5'b00001, 6'b000010}; // srl gr3, gr1, 1 (gr3 <= gr1 >> 1)

uut.instrMemory[17] = {6'b000000, `gr0, `gr1, `gr3, 5'b00001, 6'b000011}; // sra gr3, gr1, 1 (gr3 <= gr1 >>> 1)

uut.instrMemory[18] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b000100}; // sllv gr3, gr2, gr1 (gr3 <= gr2 << gr1)

uut.instrMemory[19] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b000110}; // srlv gr3, gr2, gr1 (gr3 <= gr2 >> gr1)

uut.instrMemory[20] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b000111}; // srav gr3, gr2, gr1 (gr3 <= gr2 >>> gr1

uut.instrMemory[21] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b101010}; // slt gr3, gr1, gr2 (gr3 <= gr1 < gr2)

uut.instrMemory[22] = {6'b000100, `gr1, `gr2, 16'hffff}; // beq gr1, gr2, -1 (branch -1 if gr1 - gr2 == 0)

uut.instrMemory[26] = {6'b000101, `gr1, `gr2, 16'h0003}; // bne gr0, gr0, 3 (branch 3 if gr1 - gr2 != 0)

uut.instrMemory[30] = {6'b000000, `gr1, `gr2, `gr3, 5'b00000, 6'b100000}; // add gr3, gr1, gr2 (gr3 <= gr1 + gr2)

uut.instrMemory[31] = {6'b000010, 26'h000008c}; // j 0x8c (jump to 0x8c unconditionally, should be next instruction)

uut.instrMemory[35] = {6'b000011, 26'h000009c}; // jal 0x9c (jump to 0x9c and link, should be next instruction)

uut.instrMemory[39] = {6'b000000, `gr0, 15'h0000, 6'h8}; // jr gr0 (jump to gr0 unconditionally, should start over)

// Initialization

uut.PC = 0;

uut.gr[0] = 0;

clk = 1;

*Intentionally left blank*

**Output:**

**| Fetch |** **Decode | Execute | Memory IO | Writeback |**

PC**F** : instr**D** : srcA**E** : srcB**E** :ALUOut**E**: writeData**M**:readData**M**: result**W**: gr1 : gr2 : gr3

00000000

00000004:8c010000:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx

00000008:8c020004:00000000:00000000:00000000: xxxxxxxx :xxxxxxxx :xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx

0000000c:ac000008:00000000:00000004:00000004: xxxxxxxx :000000ab :xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx

**============================================ Load & Store ============================================**

00000010:20030001:00000000:00000008:00000008: xxxxxxxx :00003c00 :000000ab:000000ab:xxxxxxxx:xxxxxxxx

00000014:24030002:00000000:00000001:00000001: 00000000 :xxxxxxxx :00003c00:000000ab:00003c00:xxxxxxxx

00000018:00221820:00000000:00000002:00000002: xxxxxxxx :000000ab :00000008:000000ab:00003c00:xxxxxxxx

**========================================== Arithmetic Operations ====================================**

0000001c:00221822:000000ab:00003c00:00003cab: xxxxxxxx :000000ab :00000001:000000ab:00003c00:00000001

00000020:00221821:000000ab:00003c00:ffffc4ab: 00003c00 :xxxxxxxx :00000002:000000ab:00003c00:00000002

00000024:00221823:000000ab:00003c00:00003cab: 00003c00 :xxxxxxxx :00003cab:000000ab:00003c00:00003cab

00000028:00221824:000000ab:00003c00:ffffc4ab: 00003c00 :xxxxxxxx :ffffc4ab:000000ab:00003c00:ffffc4ab

0000002c:00221825:000000ab:00003c00:00000000: 00003c00 :xxxxxxxx :00003cab:000000ab:00003c00:00003cab

00000030:00221827:000000ab:00003c00:00003cab: 00003c00 :000000ab :ffffc4ab:000000ab:00003c00:ffffc4ab

00000034:00221826:000000ab:00003c00:ffffc354: 00003c00 :xxxxxxxx :00000000:000000ab:00003c00:00000000

00000038:30031111:000000ab:00003c00:00003cab: 00003c00 :xxxxxxxx :00003cab:000000ab:00003c00:00003cab

0000003c:34031111:00000000:00001111:00000000: 00003c00 :xxxxxxxx :ffffc354:000000ab:00003c00:ffffc354

00000040:00011840:00000000:00001111:00001111: 00003cab :000000ab :00003cab:000000ab:00003c00:00003cab

00000044:00011842:000000ab:00000001:00000156: ffffc354 :xxxxxxxx :00000000:000000ab:00003c00:00000000

00000048:00011843:000000ab:00000001:00000055: 000000ab :xxxxxxxx :00001111:000000ab:00003c00:00001111

0000004c:00221804:000000ab:00000001:00000055: 000000ab :xxxxxxxx :00000156:000000ab:00003c00:00000156

00000050:00221806:00003c00:000000ab:01e00000: 000000ab :xxxxxxxx :00000055:000000ab:00003c00:00000055

00000054:00221807:00003c00:000000ab:00000007: 00003c00 :xxxxxxxx :00000055:000000ab:00003c00:00000055

00000058:0022182a:00003c00:000000ab:00000007: 00003c00 :00003c00 :01e00000:000000ab:00003c00:01e00000

0000005c:1022ffff:000000ab:00003c00:00000001: 00003c00 :00003c00 :00000007:000000ab:00003c00:00000007

00000060:xxxxxxxx:000000ab:00003c00:ffffc4ab: 00003c00 :000000ab :00000007:000000ab:00003c00:00000007

00000064:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: 00003c00 :xxxxxxxx :00000001:000000ab:00003c00:00000001

**gr1 != gr2**

**No BEQ**

**============================================ Jump & Branch ===========================================**

00000068:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :ffffc4ab:000000ab:00003c00:00000001

0000006c:14220003:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000001

00000070:xxxxxxxx:000000ab:00003c00:ffffc4ab: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000001

**gr1 != gr2**

**BNE 3**

00000074:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: 00003c00 :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000001

00000078:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :ffffc4ab:000000ab:00003c00:00000001

0000007c:00221820:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000001

00000080:0800008c:000000ab:00003c00:00003cab: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000001

**Unconditional**

**jump to**

**0x8c**

00000084:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: 00003c00 :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000001

00000088:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: 00000000 :xxxxxxxx :00003cab:000000ab:00003c00:00003cab

0000008c:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00003cab

00000090:0c00009c:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00003cab

**Unconditional**

**jump to**

**0x9c and link**

00000094:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00003cab

00000098:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: 00000000 :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00003cab

0000009c:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :00000090:000000ab:00003c00:00000090

000000a0:00000008:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000090

000000a4:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000090

000000a8:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: 00000000 :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000090

**Unconditional**

**jump to**

**gr0 = 0x00**

**==============================================NEW CYCLE==============================================**

00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000090

00000004:8c010000:xxxxxxxx:xxxxxxxx:xxxxxxxx: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000090

00000008:8c020004:00000000:00000000:00000000: xxxxxxxx :xxxxxxxx :xxxxxxxx:000000ab:00003c00:00000090

0000000c:ac000008:00000000:00000004:00000004: 000000ab :000000ab :xxxxxxxx:000000ab:00003c00:00000090

**.**

**.**

**.**