
# Lab no 07: Single-Cycle MIPS Processor

## The purpose of this Lab is to:

1) Understand the difference between architecture and microarchitecture.
2) Understand the Single-Cycle MIPS Processor.
3) Understand the HDL code for the Single-Cycle Processor.
4) Connect sub-modules of the Single-Cycle MIPS Processor on a top-level module.
5) Convert the MIPS assembly of a simple program to machine language.
6) Write a testbench that loads the program into the instruction memory to test the processor.

## Parts: -

1. Microarchitecture
2. Single-Cycle MIPS Processor
3. HDL Representation for the Single-Cycle MIPS Processor
4. Single-Cycle MIPS Processor Testbench


## Part 1. Microarchitecture

The architecture is the programmer's view of a computer. It is defined by the **instruction set** and operand locations (registers and memory).
Computer hardware **understands only 1's and 0's**, so instructions are encoded as binary numbers in a format called **machine language**. Microprocessors are digital systems that read and execute machine language instructions.
A **computer architecture** does not define the underlying hardware implementation. **Microarchitecture** is the specific arrangement of registers, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture.
You have learned about the **MIPS architecture**, which specifies the programmer's view of the MIPS processor in terms of registers, instructions, and memory. In this lab, you will learn how to piece together a MIPS microprocessor.

## Part 2. Single-Cycle MIPS Processor

We will divide our microarchitectures into two interacting parts: the datapath and the control. The **datapath** contains structures such as memories, registers, ALUs, and multiplexers.
MIPS is a **32-bit architecture**, so we will use a 32-bit datapath. The **control unit** receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.
The **single-cycle microarchitecture** executes an entire instruction in **one cycle**. It is easy to explain and has a simple control unit.
The **program counter** is a 32-bit register. The program counter (PC) register contains the address of the instruction to execute. Because instructions are 32 bits = **4 bytes**, the next instruction is at PC + 4.

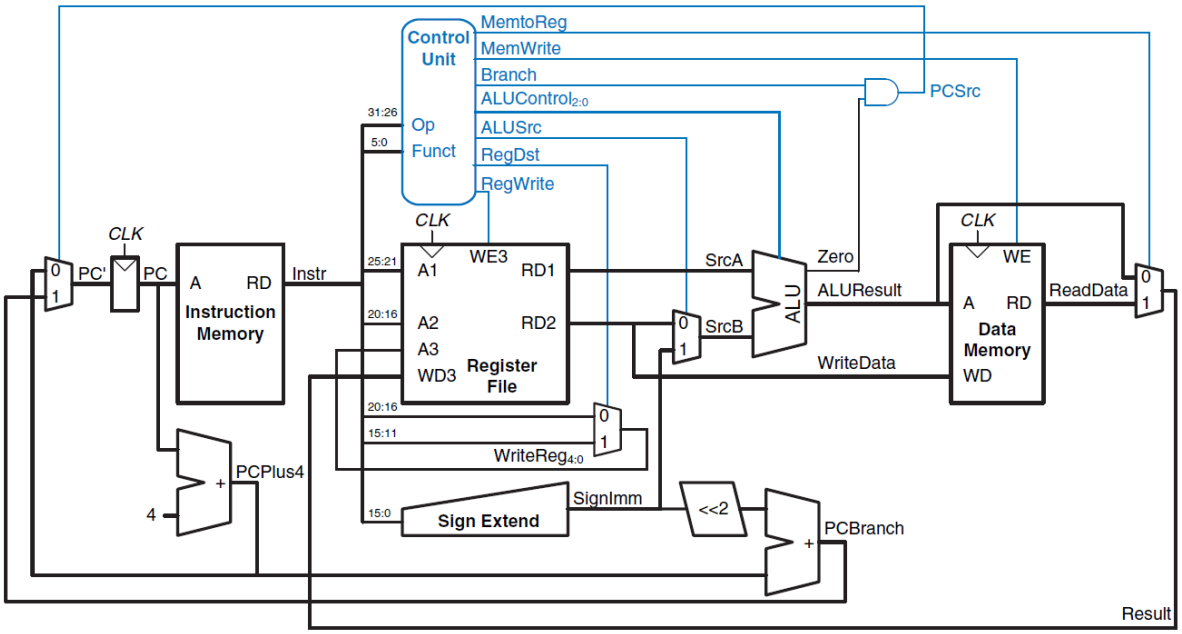



**Figure 1.** Complete single-cycle MIPS processor

The **instruction memory** has a single read port. It takes a 32-bit instruction address input (A), and reads the 32-bit data from that address onto the read data output (RD).

Treating the instruction memory as a **ROM** is an **oversimplification**. In most real processors, the **instruction memory must be writable** so that the OS can load a new program into memory.

The **32-element × 32-bit register file** has two read ports and one write port. The read ports take 5-bit address inputs (A1) and (A2) each specifying one of $2^5 = 32$ registers as source operands. They read the 32-bit register values onto read data outputs (RD1) and (RD2), respectively. The write port takes a 5-bit address input (A3), a 32-bit write data input (WD), a write enable input (WE3), and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The **data memory** has a single read/write port. If the **write enable** (WE) is 1, it writes data (WD) into address (A) on the rising edge of the clock. If the write enable is 0, it reads address (A) onto (RD).

## Part 3. HDL Representation for the Single-Cycle MIPS Processor

This section presents HDL code for the single-cycle MIPS processor supporting all of the instructions discussed in **Lecture 10**, including addi and j.

The instruction and data memories are separated from the main processor and connected by address and data busses. This is more realistic, because most real processors have **external memory**. It also illustrates how the processor can communicate with the outside world.

The processor is composed of a **datapath** and a **controller**. The controller is composed of the **main decoder** and the **ALU decoder**. Figure 2 shows a **block diagram** of the single-cycle MIPS processor interfaced to external memories.

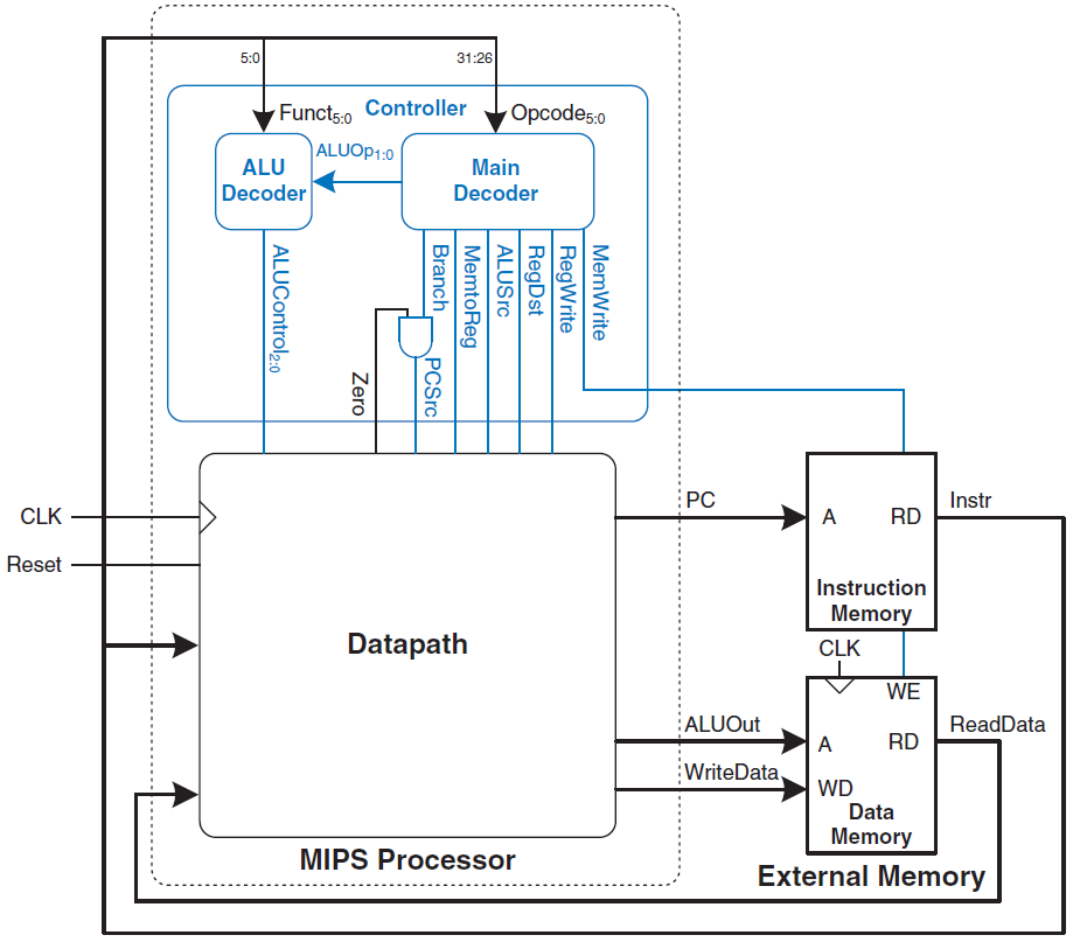
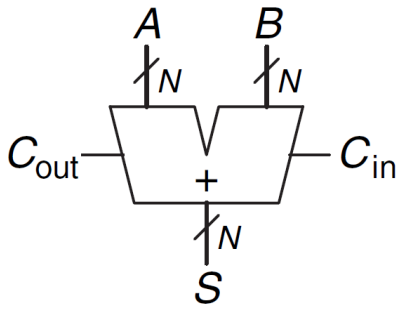



**Figure 2.** MIPS single-cycle processor interfaced to external memory

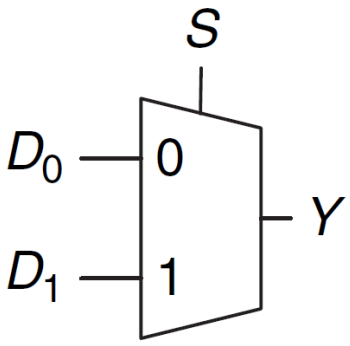
Note that all HDL files are written in **SystemVerilog**. SystemVerilog acts as a **superset** of Verilog.

## Adder



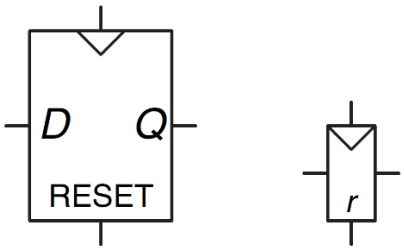

```
module adder(input logic [31:0] a, b,
             output logic [31:0] y);

     assign y = a + b;
endmodule
```

## 2:1 Multiplexer



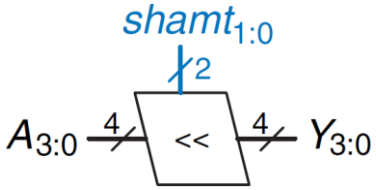

```
module mux2 #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d0, d1,
              input logic s,
              output logic [WIDTH-1:0] y);

     assign y = s ? d1 : d0;
endmodule
```

## Resettable Flip-Flop



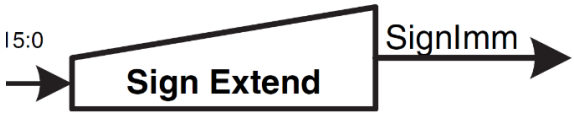

```
module flopr #(parameter WIDTH = 8)
              (input logic clk, reset,
               input logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);
     always_ff @(posedge clk, posedge reset)
          if (reset)
               q <= 0;
          else
               q <= d;
endmodule
```

## Shift Left By 2 (Multiply By 4)



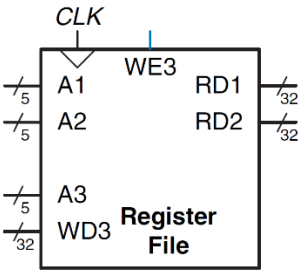

```
module sl2(input logic [31:0] a, output logic [31:0] y);
     assign y = {a[29:0], 2'b00};
endmodule
```

## Sign Extension



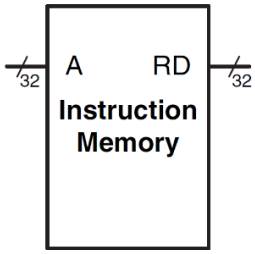

```
module signext(input logic [15:0] a, output logic [31:0] y);
     assign y = {{16{a[15]}}, a};
endmodule
```

## Register File



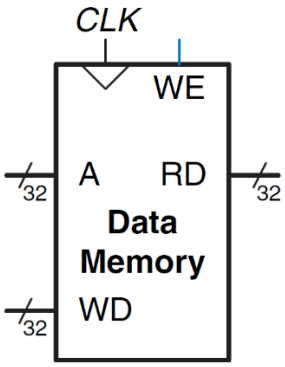

```
module regfile(input logic clk,
               input logic we3,
               input logic [4:0] ra1, ra2, wa3,
               input logic [31:0] wd3,
               output logic [31:0] rd1, rd2);
     logic [31:0] rf[31:0];

     always_ff @(posedge clk)
          if (we3)
               rf[wa3] <= wd3;
     assign rd1 = (ra1 ! = 0) ? rf[ra1] : 0;
     assign rd2 = (ra2 ! = 0) ? rf[ra2] : 0;
endmodule
```

## Instruction Memory



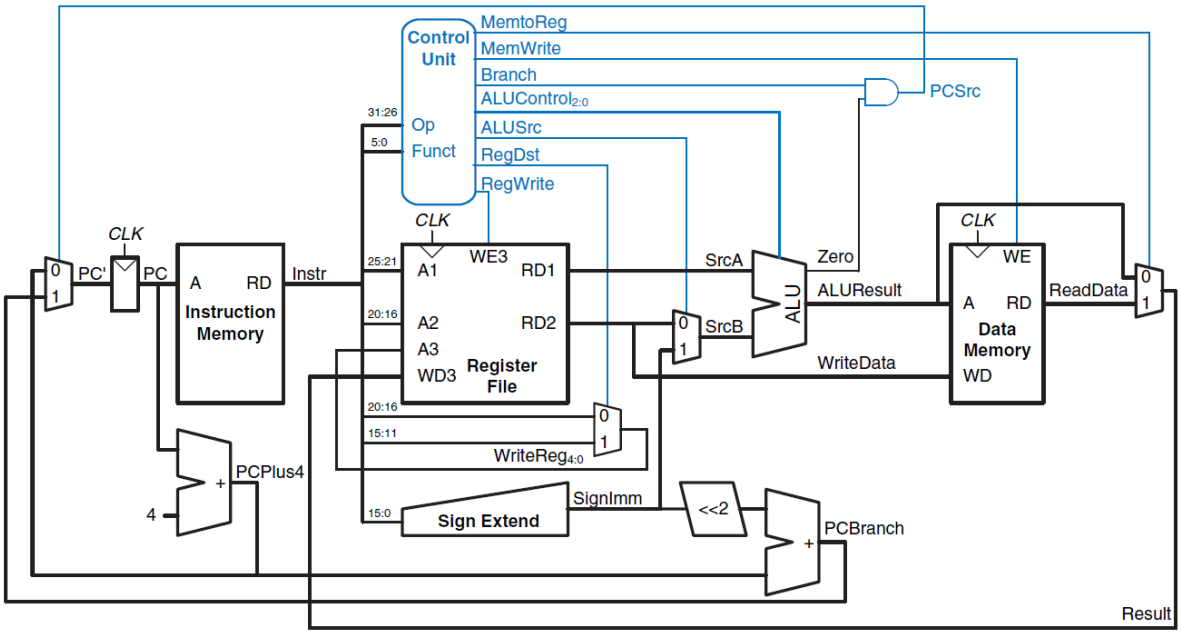

```
module imem(input logic [5:0] a,
            output logic [31:0] rd);
     logic [31:0] RAM[63:0];
     initial
          $readmemh("memfile.dat", RAM);
          assign rd = RAM[a]; // word aligned
endmodule
```

## Data Memory




```
module dmem(input logic clk, we,
            input logic [31:0] a, wd,
            output logic [31:0] rd);
     logic [31:0] RAM[63:0];
     assign rd = RAM[a[31:2]]; // word aligned
     always_ff @(posedge clk)
     if (we)
          RAM[a[31:2]] <= wd;
endmodule
```

## Datapath




```
module datapath(input logic clk, reset,
                input logic memtoreg, pcsrc,
                input logic alusrc, regdst,
                input logic regwrite, jump,
                input logic [2:0] alucontrol,
                output logic zero,
                output logic [31:0] pc,
                input logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input logic [31:0] readdata);
   logic [4:0] writereg;
   logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
   logic [31:0] signimm, signimmsh;
   logic [31:0] srca, srcb;
   logic [31:0] result;
   // next PC logic
   flopr #(32) pcreg(clk, reset, pcnext, pc);
   adder pcadd1(pc, 32'b100, pcplus4);
   sl2 immsh(signimm, signimmsh);
   adder pcadd2(pcplus4, signimmsh, pcbranch);
   mux2 #(32) pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
   mux2 #(32) pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00}, jump, pcnext);
   // register file logic
   regfile rf(clk, regwrite, instr[25:21], instr[20:16],
   writereg, result, srca, writedata);
   mux2 #(5) wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
   mux2 #(32) resmux(aluout, readdata, memtoreg, result);
   signext se(instr[15:0], signimm);
   // ALU logic
   mux2 #(32) srcbmux(writedata, signimm, alusrc, srcb);
   alu alu(srca, srcb, alucontrol, aluout, zero);
endmodule
```


## Main Decoder

| Instruction | Opcode | RegWrite | RegDst | ALUSrc | Branch | MemWrite | MemtoReg | ALUOp | Jump |
|---|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |
| addi | 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | 0 |
| j | 000010 | 0 | X | X | X | 0 | X | XX | 1 |

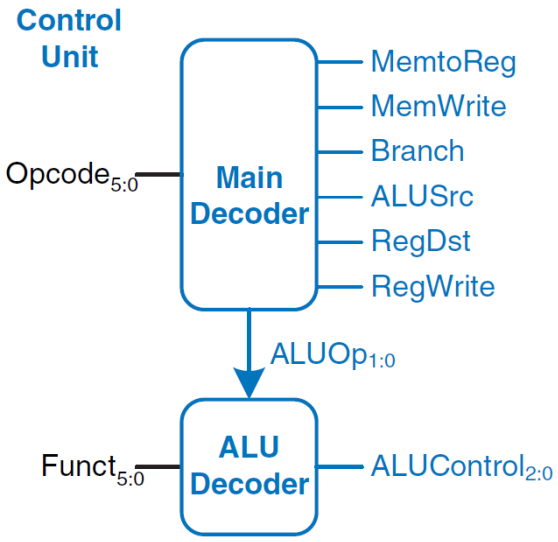
```
module maindec(input logic [5:0] op,
               output logic memtoreg, memwrite,
               output logic branch, alusrc,
               output logic regdst, regwrite,
               output logic jump,
               output logic [1:0] aluop);

    logic [8:0] controls;
    assign {regwrite, regdst, alusrc, branch, memwrite,
            memtoreg, jump, aluop} = controls;

    always_comb
    case(op)
        6'b000000: controls <= 9'b110000010; // RTYPE
        6'b100011: controls <= 9'b101001000; // LW
        6'b101011: controls <= 9'b001010000; // SW
        6'b000100: controls <= 9'b000100001; // BEQ
        6'b001000: controls <= 9'b101000000; // ADDI
        6'b000010: controls <= 9'b000000100; // J
        default: controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```


## ALU Decoder

| ALUOp | Funct | ALUControl |
|---|---|---|
| 00 | X | 010 (add) |
| X1 | X | 110 (subtract) |
| 1X | 100000 (add) | 010 (add) |
| 1X | 100010 (sub) | 110 (subtract) |
| 1X | 100100 (and) | 000 (and) |
| 1X | 100101 (or) | 001 (or) |
| 1X | 101010 (slt) | 111 (set less than) |

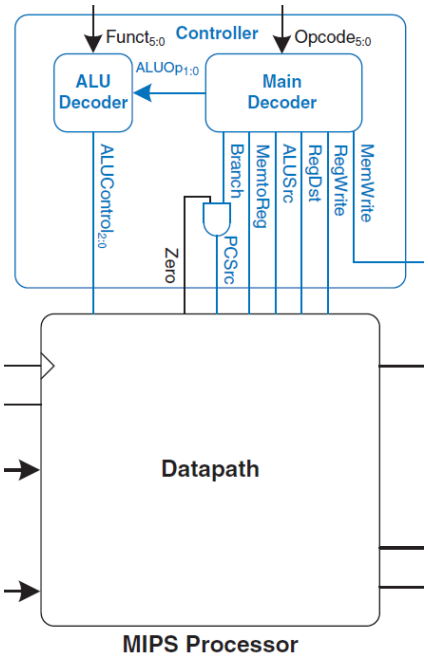
```
module aludec(input logic [5:0] funct,
              input logic [1:0] aluop,
              output logic [2:0] alucontrol);

   always_comb
   case(aluop)
      2'b00: alucontrol <= 3'b010; // add (for lw/sw/addi)
      2'b01: alucontrol <= 3'b110; // sub (for beq)
      default:
         case(funct) // R-type instructions
            6'b100000: alucontrol <= 3'b010; // add
            6'b100010: alucontrol <= 3'b110; // sub
            6'b100100: alucontrol <= 3'b000; // and
            6'b100101: alucontrol <= 3'b001; // or
            6'b101010: alucontrol <= 3'b111; // slt
            default: alucontrol <= 3'bxxx;   // ???
         endcase
   endcase
endmodule
```

## Controller




```
module controller(input logic [5:0] op, funct,
                  input logic zero,
                  output logic memtoreg, memwrite,
                  output logic pcsrc, alusrc,
                  output logic regdst, regwrite,
                  output logic jump,
                  output logic [2:0] alucontrol);

    logic [1:0] aluop;
    logic branch;

    maindec md(op, memtoreg, memwrite, branch,
               alusrc, regdst, regwrite, jump, aluop);
    aludec ad(funct, aluop, alucontrol);
    assign pcsrc = branch & zero;
endmodule
```

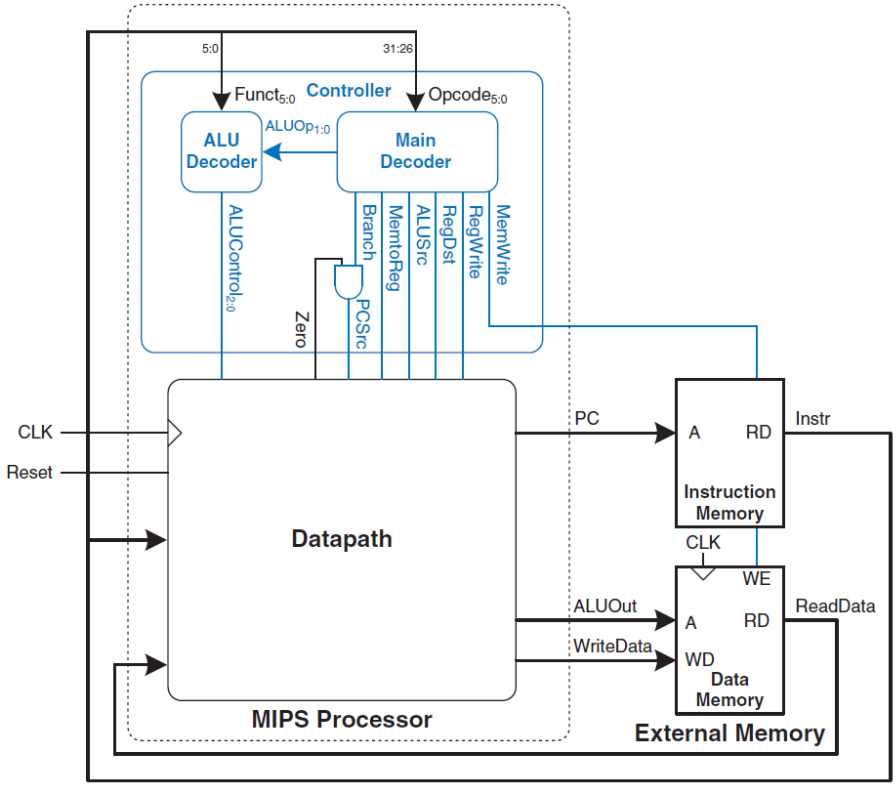
## Single-Cycle MIPS Processor




```
module mips(input logic clk, reset,
            output logic [31:0] pc,
            input logic [31:0] instr,
            output logic memwrite,
            output logic [31:0] aluout, writedata,
            input logic [31:0] readdata);
    logic memtoreg, alusrc, regdst,
    regwrite, jump, pcsrc, zero;
    logic [2:0] alucontrol;

    controller c(instr[31:26], instr[5:0], zero,
                 memtoreg, memwrite, pcsrc,
                 alusrc, regdst, regwrite, jump,
                 alucontrol);
    datapath dp(clk, reset, memtoreg, pcsrc,
                alusrc, regdst, regwrite, jump,
                alucontrol,
                zero, pc, instr,
                aluout, writedata, readdata);
endmodule
```

The instruction and data **memories are separated from the main processor** and connected by address and data busses. This is more realistic, because most real processors have **external memory**. It also illustrates how the processor can communicate with the outside world.

## MIPS Top-Level Module




```
module top(input logic clk, reset,
           output logic [31:0] writedata, dataadr,
           output logic memwrite);
   logic [31:0] pc, instr, readdata;

   // instantiate processor and memories
   mips mips(clk, reset, pc, instr, memwrite, dataadr,
             writedata, readdata);
   imem imem(pc[7:2], instr);
   dmem dmem(clk, memwrite, dataadr, writedata, readdata);
endmodule
```


## Part 4. Single-Cycle MIPS Processor Testbench

The MIPS testbench loads a program into the **instruction memory**. The machine code is stored in a **hexadecimal file** called **memfile.dat**, which is loaded by the testbench during simulation. The file consists of the **machine code for the instructions**, one instruction per line.

The program exercises all of the instructions by performing a computation that should produce the correct answer only if all of the instructions are **functioning properly**. The program will **write the value 7 to address 84 if it runs correctly**, and is unlikely to do so if the hardware is buggy.

| | Assembly | Description | Address | Machine |
|---|---|---|---|---|
| main: | addi $2, $0, 5 | # initialize $2 = 5 | 0 | 20020005 |
| | addi $3, $0, 12 | # initialize $3 = 12 | 4 | 2003000c |
| | addi $7, $3, -9 | # initialize $7 = 3 | 8 | 2067fff7 |
| | or $4, $7, $2 | # $4 = (3 OR 5) = 7 | c | 00e22025 |
| | and $5, $3, $4 | # $5 = (12 AND 7) = 4 | 10 | 00642824 |
| | add $5, $5, $4 | # $5 = 4 + 7 = 11 | 14 | 00a42820 |
| | beq $5, $7, end | # shouldn't be taken | 18 | 10a7000a |
| | slt $4, $3, $4 | # $4 = 12 < 7 = 0 | 1c | 0064202a |
| | beq $4, $0, around | # should be taken | 20 | 10800001 |
| | addi $5, $0, 0 | # shouldn't happen | 24 | 20050000 |
| around: | slt $4, $7, $2 | # $4 = 3 < 5 = 1 | 28 | 00e2202a |
| | add $7, $4, $5 | # $7 = 1 + 11 = 12 | 2c | 00853820 |
| | sub $7, $7, $2 | # $7 = 12 - 5 = 7 | 30 | 00e23822 |
| | sw $7, 68($3) | # [80] = 7 | 34 | ac670044 |
| | lw $2, 80($0) | # $2 = [80] = 7 | 38 | 8c020050 |
| | j end | # should be taken | 3c | 08000011 |
| | addi $2, $0, 1 | # shouldn't happen | 40 | 20020001 |
| end: | sw $2, 84($0) | # write mem[84] = 7 | 44 | ac020054 |


## MIPS Top-Level Module Testbench

```
module testbench();
  logic clk;
  logic reset;
  logic [31:0] writedata, dataadr;
  logic memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite);
  // initialize test
  initial
    begin
      reset <= 1; #22;
      reset <= 0;

      $display("\t\t    time \tdataadr\t\twritedata");
      $monitor("%d\t%d\t%d", $time, dataadr, writedata);
    end
  // generate clock to sequence tests
  always
    begin
      clk <= 1; #5;
      clk <= 0; #5;
    end
  // If successful, it should write the value 7 to address 84
  always @(negedge clk)
    begin
      if(memwrite) begin
        if(dataadr === 84 & writedata === 7) begin
          $display("Simulation succeeded");
          $stop;
        end
      end
    end
endmodule
```


As mentioned, the machine code is stored in a **hexadecimal file** called `memfile.dat`, which consists of the **machine code** for the instructions, **one instruction per line**.

**memfile.dat**

```
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054
```

The following **do file** is used to run the testbench of the MIPS processor. To run the do file use the command `do mips.do`

**mips.do**

```
vsim testbench
add wave -position insertpoint sim:/testbench/*
run 200
```


The single-cycle MIPS processor executes **instruction by instruction**. Each entire instruction is executed in **one cycle**. The last instruction should write the **value 7 to address 84** as shown in Figure 3, so the **simulation has succeeded**, and the HDL code is functioning properly.

**MIPS Testbench Output**

```
# time        dataadr      writedata
# 22           5            5
# 30          12            x
# 40           3            x
# 50           7            5
# 60           4            7
# 70          11            7
# 80           8            3
# 90           0            7
# 100          0            0
# 110          1            5
# 120         12           11
# 130          7            5
# 140         80            7
# 150         80            5
# 160          0            0
# 170         84            7
# Simulation succeeded
```

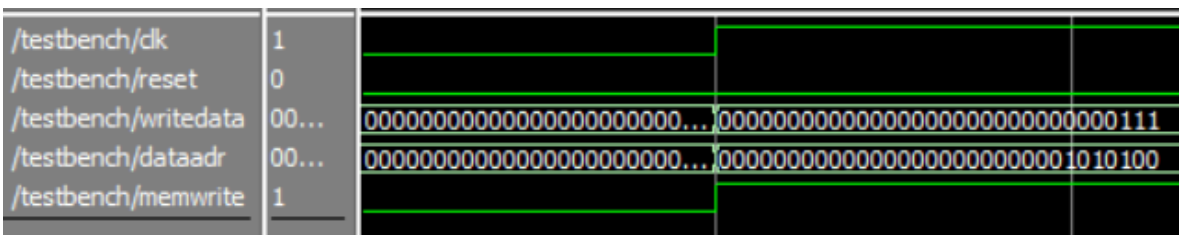



**Figure 3.** Waveforms of the testbench


## Final Project Hint

1. Write the C Code for the **Complete** **bank System**.
2. **Only** Translate the C function of **Wtime** into MIPS assembly language.
3. Convert the Wtime assembly instructions, from (2) into machine code.
4. Create Wtime.dat and upload the machine code of the Wtime into MIPS memory.
5. Verify your code by simulating the MIPS processor in ModelSim.
6. Identify two testcases and show that your code for Wtime is working correctly.