Digital Design Project

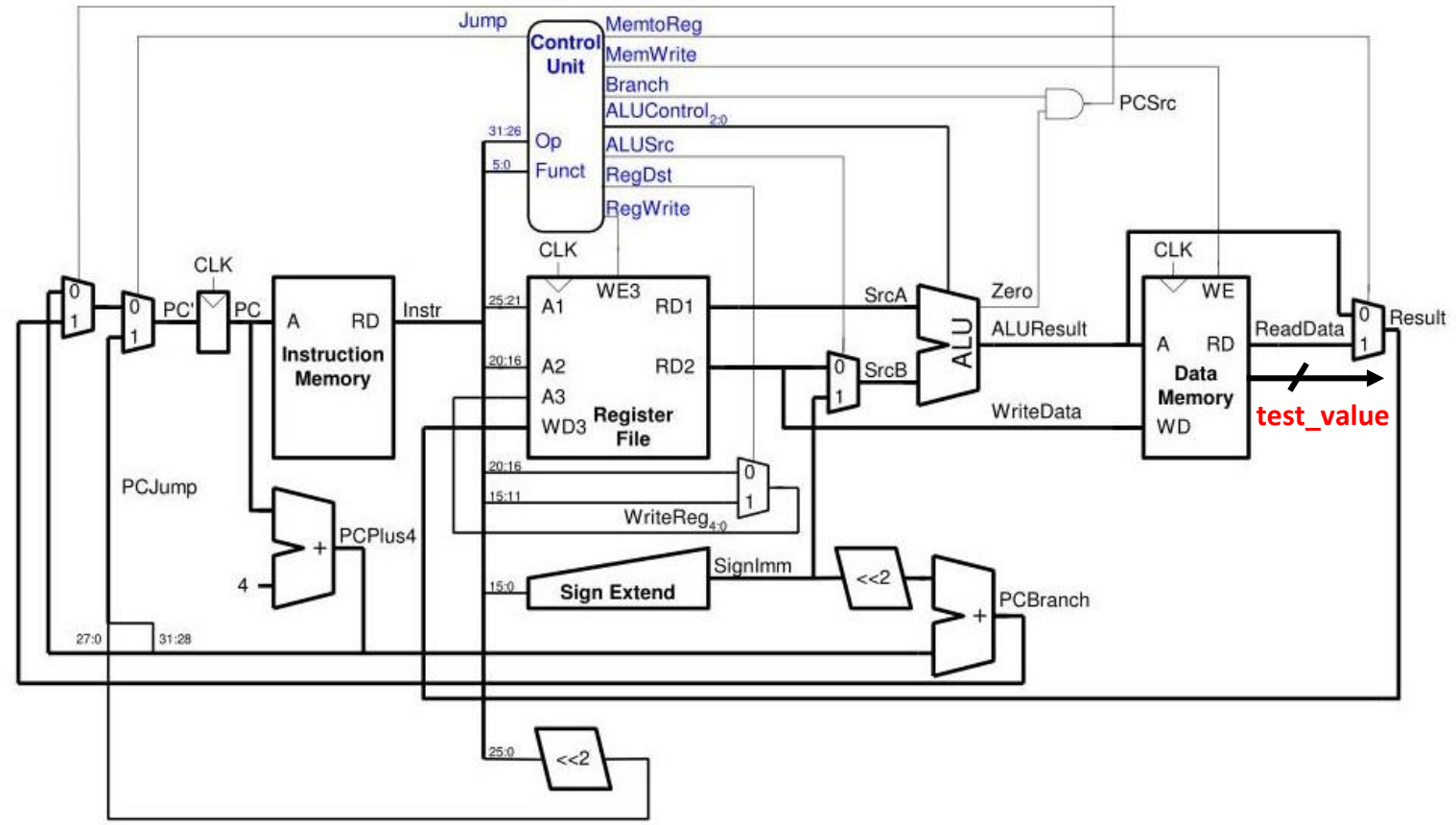
Single Cycle MIPS Processor

Introduction

In this project, you are required to implement a 32-bit single-cycle microarchitecture MIPS processor based on Harvard Architecture. The single-cycle microarchitecture executes an entire instruction in one cycle. In other words instruction fetch, instruction decode, execute, write back, and program counter update occurs within a single clock cycle.

Objective

Referring to figure one, you are required to write the RTL Verilog files for all submodules of the MIPS processor (e.g. Register File, Instruction Memory, etc.). Then, implementing the top module of the MIPS processor. Finally, you will configure this processor



on Cyclone® IV FPGA device.

Figure 1: Complete single-cycle MIPS processor.
(Retrieved from David M. Harris, Sarah L. Harris - Digital Design and Computer Architecture)

Top Module View

The processor is composed of a datapath and a controller. The controller, in turn, 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.

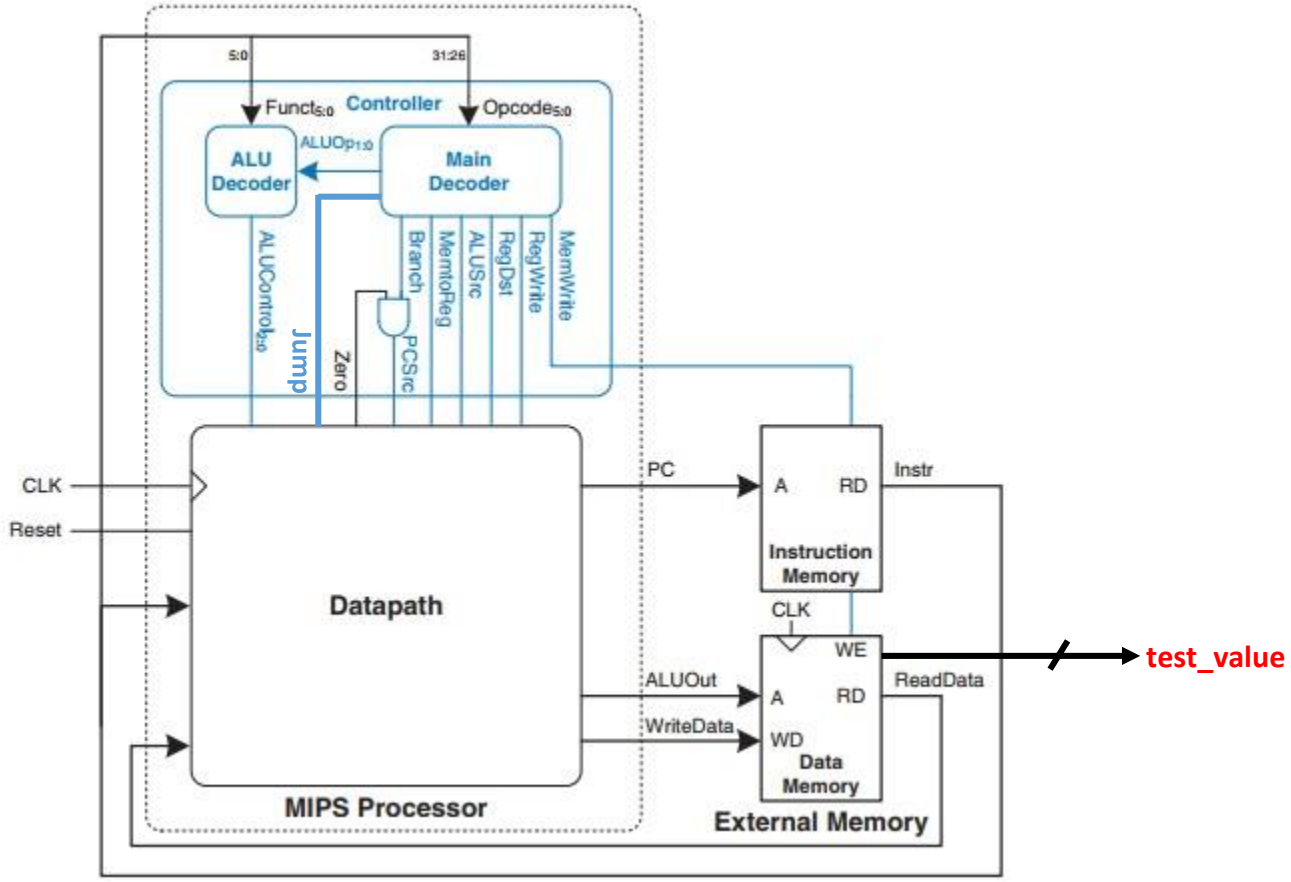
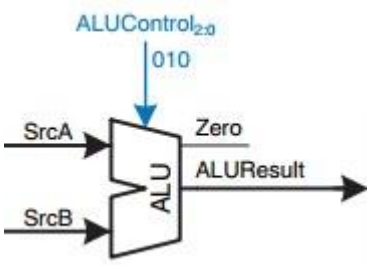


Figure 2: MIPS single-cycle processor interfaced to external memory.
(Retrieved from David M. Harris, Sarah L. Harris - Digital Design and Computer Architecture)

Main Modules

1. ALU

An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. The ALU forms the heart of most computer systems. The 3-bit ALUControl signal specifies the operation. The ALU generates a **32-bit ALUResult and a Zero flag**, that indicates whether ALUResult == 0. The following table lists the specified functions that our ALU can perform.



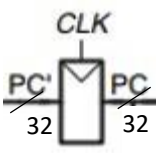
$ALUControl_{2:0}$	Function
000	SrcA (AND) SrcB
001	SrcA (OR) SrcB
010	SrcA + SrcB
011	Not Used
100	SrcA - SrcB
101	SrcA * SrcB
110	SLT
111	Not Used

Note:

- SLT is an abbreviation for **Set Less Than.** This function is responsible for setting the output ALUResult to 1 when SrcA is less than SrcB otherwise ALUResult is set to 0.
- **Don't** register the output Y or the inputs A and B

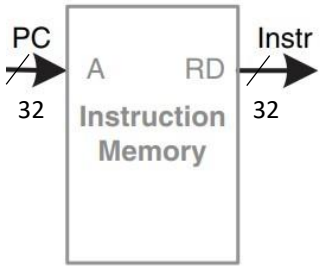
2. Program Counter

The program counter (PC) **register** contains the 32-bit address of the instruction to execute. The program counter is a synchronous unit that is updated at the rising edge of the clock signal "clk". The program counter is **asynchronously** cleared (zeroed) whenever the **active low** reset signal "**rst**" is asserted.



3. Instruction memory

The PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr. Our instruction memory is a **Read Only Memory** (ROM) that holds the program that your CPU will execute. The ROM Memory has **width = 32 bits and depth = 100 entries**. Instr is read **asynchronously**.

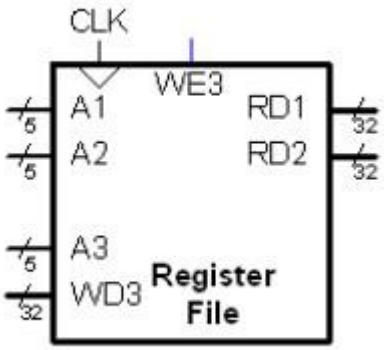


Note:

- Assume that our Memory is **not word aligned**. It supports unaligned data of 32 bits, so we **need to divide our address by 4** (PC >> 2 = PC / 4).

4. Register File

The Register File contains the 32 32-bit MIPS registers. The register file has two read output ports (RD1 and RD2) and a single input write port (WD3). The register file is **read asynchronously** and **written synchronously** at the rising edge of the clock. The register file supports simultaneous read and writes. The register file has **width = 32 bits and depth = 100 entries.**



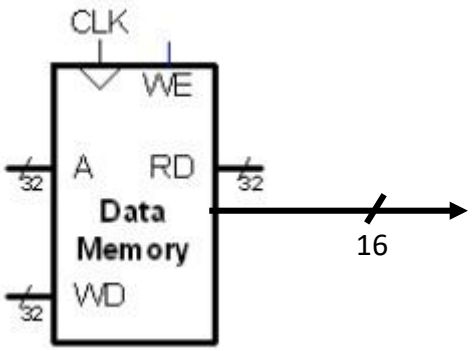
Note:

- Read asynchronously means that RD1 and RD2 are implemented using combinational logic. Thus, RD1 and RD2 are read with no respect to the clock edge.
- The register file has active low asynchronous reset signal.
- There is a write enable signal (WE3) that is used to enable writing the new value on the data bus (WD3) to the specified register address (A3).
- A1 is the register address from which the data are read through the output port RD1. Whereas A2 is corresponding to the register address of output port RD2.

5. Data Memory

The data memory is a RAM that provides a store for the CPU to load from and store to. The Data Memory has one output read port (RD) and one input write port (WD). **Reads are asynchronous** while **writes are synchronous** to the rising edge of the "clk" signal. The Word width of the data memory is **32-bits** to match the datapath width. The data memory contains **100 entries**.

test_value

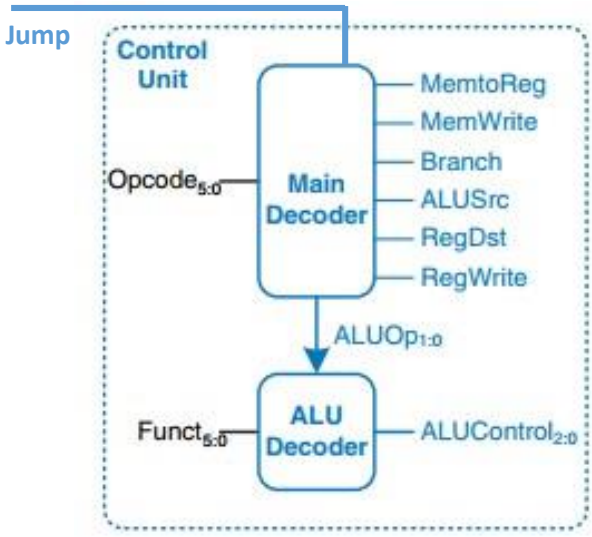


Note:

- Read asynchronously means that RD is implemented using combinational logic. Thus, RD is read with no respect to the clock edge.
- The Data Memory has active low asynchronous reset signal.
- There is a write enable signal (WE) that is used to enable writing the new value on the data bus (WD) to the specified memory address (A).
- A is the memory address from which the data are read through the output port RD.
- **test_value** is read from address **0x0000_0000** (First location on Memory) the **least 16 significant bits**. This signal is for only testing purposes to be able to see the results of the testing programs on the four digital tubes after configuration.

6. Control Unit

The control unit computes the control signals based on the opcode and funct fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$. Most of the control information comes from the opcode, but R-type instructions also use the funct field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in the figure below.



ALU decoder truth table

Note: All values are written in binary

ALUOp	Funct	ALUControl
00	xxxxxx	010
01	xxxxxx	100
10	add = 6'b10_0000	010
	sub = 6'b10_0010	100
	slt = 6'b10_1010	110
	mul = 6'b01_1100	101
Default	xxxxxx	010

Main decoder truth table

Note: All values are written in binary

opCode	jump	aluOp	memWrite	regWrite	regDest	aluSrc	memtoReg	Branch
loadWord = 6'b10_0011	0	00	0	1	0	1	1	0
storeWord = 6'b10_1011	0	00	1	0	0	1	1	0
rType = 6'b00_0000	0	10	0	1	1	0	0	0
addImmediate = 6'b00_1000	0	00	0	1	0	1	0	0
branchIfEqual = 6'b00_0100	0	01	0	0	0	0	0	1
jump_inst = 6'b00_0010	1	00	0	0	0	0	0	0
Default	0	00	0	0	0	0	0	0

Small Modules

1. Sign Extend

Sign extension simply copies the sign bit (most significant bit) of a short input (16 bits) into all of the upper bits of the longer output (32 bits).

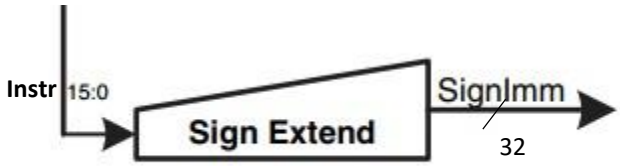
Example:

- Inst[15:0] = 0000 0000 0011 1101 →

  SignImm = **0000 0000 0000 0000** 0000 0000 0011 1101

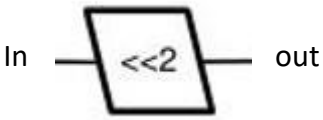
- Inst[15:0] = 1000 0000 0011 1101 →

  SignImm = **1111 1111 1111 1111** 1000 0000 0011 1101



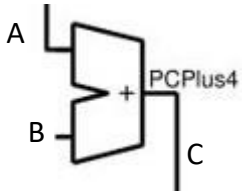
2. shift_left_twice

This block only shift the input to the left twice. You need to make this block **parameterized**, as you need two versions of it in your top module with two different data input width.



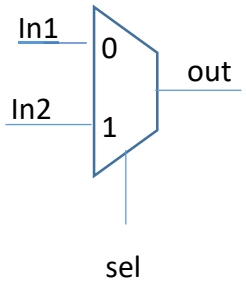
3. Adder

This is a simple combination block. It is an adder block that adds two 32-bit data inputs to each other (A and B) and produces the output to the 32-bit port C.



4. MUX

You are required to implement a **parameterized** 2X1 MUX.

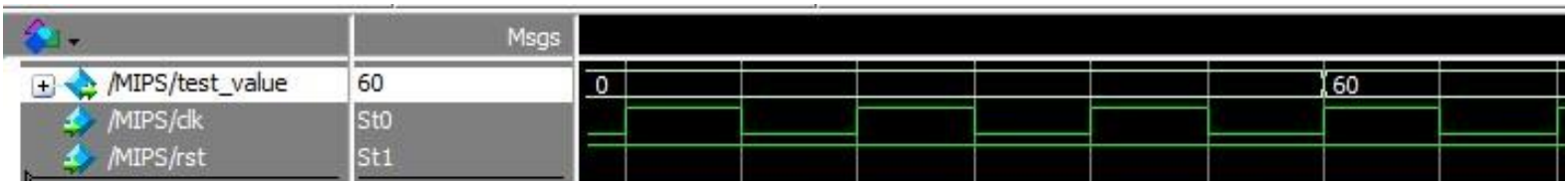


Project Testing

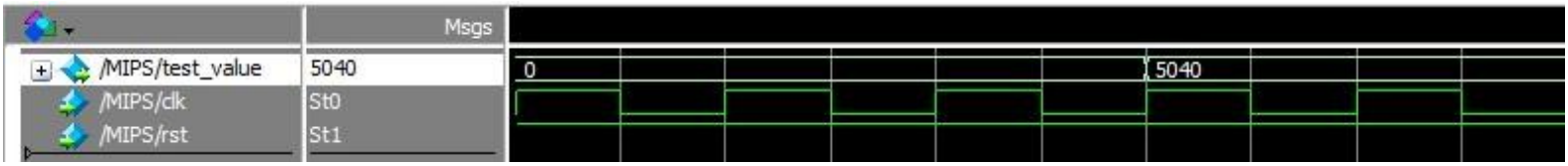
You will be given two different test programs. You are required to compile and load it onto the **Instruction Memory** (ROM Memory) and reset your processor to start executing from memory location **0000 0000h**. Each program would test some instructions. The two test programs are **GCD Program and Factorial Number Program**.

Final Simulation Results

Program 1: GCD of 120 and 180



Program 2: Factorial Number of 7



General Notes

Guidelines

- This is **not** a Computer Architecture Workshop. Therefore, **you have to study** Chapter 6 and 7 (from 7.1 to 7.3) before the implementation phase to be able to know more about this architecture and MIPS Assembly.
- If you have any question relating to the study materials or project, it is more than welcome.
- Compile your design on regular bases (after each modification) so that you can figure out new errors early. Accumulated errors are harder to track.
- **Start early and give yourself enough time for testing.**

Requirements

- Implement and integrate your architecture ○ Verilog Implementation of each module of the processor
    - ○ Verilog file that integrates the different sub-modules in a single module (Top Level named MIPS)
- Simulation the given Test machine codes that reads the program file and execute it on the processor. Then, take screenshots of your results as previously shown in section "Final Simulation Results" above.
- A screenshot from the RTL Viewer in Quartus of the MIPS Processor.