FULL FPGA IMPLEMENTATION OF 32-BIT MIPS PROCESSOR INTERIM REPORT - TEAM 22 - ALUMINATI

Guntas Singh Saran[*]**& Hrriday V. Ruparel** [†]**& Kishan Ved**[‡]**& Pranav Patil**[§]
Department of Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India

Prof. Sameer Kulkarni
Department of Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India

1 INTRODUCTION

This project focuses on the design and implementation of a 32-bit MIPS (Microprocessor without Interlocked Pipeline Stages) processor on a Basys3 FPGA board. MIPS is a RISC (Reduced Instruction Set Computing) architecture known for its simplicity and efficiency, making it a suitable choice for FPGA implementation. The objective is to create a functional processor capable of executing a subset of the MIPS instruction set, with the program and data stored in FPGA's Block RAM (BRAM). The output will be displayed on a 7-segment LED display using memory-mapped I/O.

2 OVERVIEW

The following report is an extensive study regarding the implementation of MIPS32 Instruction Set on the Basys3 FPGA Board. The report begins with section 3 describing a selection of a subset of MIPS32 instructions that we intend to implement. This would govern the processor components we may need to incorporate in our design and organisation. We have discussed 2 approaches in section 4 i) Single-Cycle without Parallelizing and ii) FSM Based Implementation. section 5 provides a description (with some Verilog code snippets) of the major **Modules**. These modules, along with their expected datapath Figure 1, have been adapted from Patterson & Hennessy (2013). This is followed by section 6, which contains the plan for registers and memory management using BRAM on the FPGA board. section 7 discusses the member wise contribution over the last few weeks and section 8 describes tasks remaining. Finally, section 9 details the updated milestones/timeline.

3 FPGA-IMPLEMENTABLE MIPS32 INSTRUCTION SET

After a careful study, we have filtered a **subset** of selected MIPS32 instructions for implementation on the Basys3 FPGA board. The tables in Appendix A list the selected instructions, their corresponding assembly codes, and their corresponding binary encoding as suggested in Patterson & Hennessy (2013).

[*]Guntas Singh Saran - 22110089 - guntassingh.saran@iitgn.ac.in
[†]Hrriday V. Ruparel 22110099 - hrriday.ruparel@iitgn.ac.in
[‡]Kishan Ved - 22110122 - kishan.ved@iitgn.ac.in
[§]Pranav Patil - 22110199 - pranav.patil@iitgn.ac.in

4 APPROACHES

We have researched upon 2 approaches, and we request a meeting with Sir to discuss these. They are as follows:

4.1 APPROACH 1 - SINGLE-CYCLE WITHOUT PIPELINING

This is the regular MIPS architecture that has been taught in class and is mentioned in the Textbook. This connects modules (described ahead in detail) based on the Data Path mentioned in Figure 1.

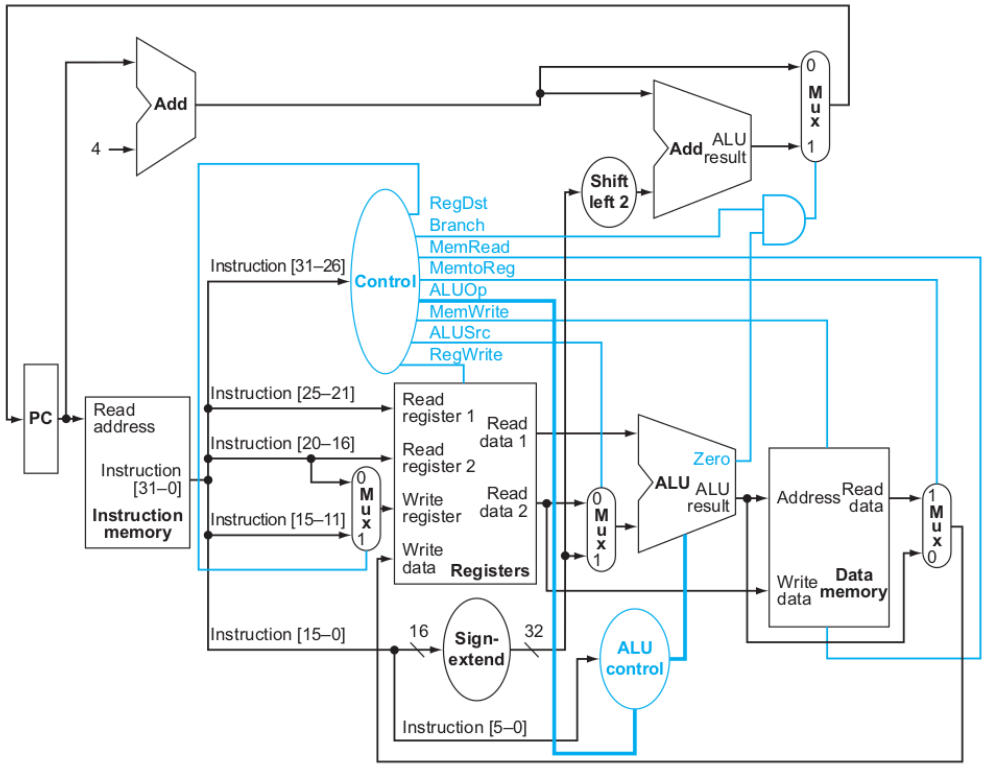

Figure 1: Expected Datapath

In this approach, we connect modules to make a huge combinatorial circuit. Processor specifications are as follows:

- Single-Cycle without Pipelining
- Clock-edge triggered processor
- Clock duration is determined by the longest path
- Integer Register Operations (no floats)
- Little Endian
- 2's complement System
- Register File on LUTs while Memory on BRAM
- Modules as mentioned in the report below
- Registers as mentioned in the report below

Though this is theoretically a good approach, we have identified **potential issues that might occur when we implement this on an FPGA :**

- It is difficult to estimate gate delays on the FPGA
- The duration of the clock cycle will be determined by the time taken by the longest operation. Thus, the waiting time of other faster operations increases.
- Reading and writing from BRAM is slow. BRAM depends upon the clock cycle, and if memory access takes longer time, this will cause a lag.

4.2 Approach 2 - FSM Based Implementation

To solve the above problems, we have researched upon this alternative approach. This approach uses Finite State Machine (FSM) based implementation, and consideres every action: Fetch, Decode, Execute, Memory Access and Write as a separate state. States individually have a combinatorial circuit made up of the same modules as used in Approach 1. We pass the same clock to every state, and a state gets triggered based on it's control signal and a clock edge.

For example, data access from BRAM might lag and takes longer time, but it will remain in the F state. The control signal of the next state won't be triggered till this is complete. Once the BRAM access is done, the control signals of next states are triggered and then based on the clock edge, operations are performed. At this edge, we can also start the F state of the next instruction.

In this manner, we can prevent lags from occurring and every state will operate on the clock of the FPGA, we don't need to design a custom clock, that is of a longer duration. States are triggered based on control signals, that depend upon the previous state's output (whether or not it is complete). This allows us to use the clock available in the FPGA itself, which is a very high frequency clock.

Processor specifications are as follows:

- FSM based processor implementation
- Clock-edge and control signal triggered states
- Every state has a combinatorial circuit made of modules
- Integer Register Operations (no floats)
- Little Endian
- 2's complement System
- Register File on LUTs while Memory on BRAM
- Modules as mentioned in the report below
- Registers as mentioned in the report below

5 Modules

Regardless of the approach selected, the modules remain the same. These are listed below. If we choose the combinatorial approach, these modules are linked together as mentioned in the DataPath in Figure 1. Else, if we choose the FSM based approach, these modules are present as combinatorial logic in different states.

5.0.1 Main CPU Module

This is the master module, which handles all the control logic and clock triggers to other modules and describes the processor's datapath. It is an abstraction of the processor and closely resembles the *top module* in Verilog.

5.0.2 Decoder Module

This will just split the instruction into the respective instruction formats. The following is how we will implement this in Verilog:

Listing 1: Decoder Module in Verilog

```
module decoder(
    input wire [31:0]  instruction,
    output wire [5:0]  opcode,
    output wire [4:0]  rs,
    output wire [4:0]  rt,
    output wire [4:0]  rd,
    output wire [4:0]  shamt,
    output wire [5:0]  funct,
    output wire [15:0] imm,
    output wire [25:0] target
);

assign opcode       = instruction[31:26];
assign rs           = instruction[25:21];
assign rt           = instruction[20:16];
assign rd           = instruction[15:11];
assign shamt        = instruction[10:06];
assign funct        = instruction[05:00];
assign imm          = instruction[15:00];
assign target       = instruction[25:00];

endmodule
```

5.0.3 PC UPDATE MODULE

This will update PC to the next instruction (basically, it will add 4 to PC). All instructions are of 4 bytes. See Figure 2.

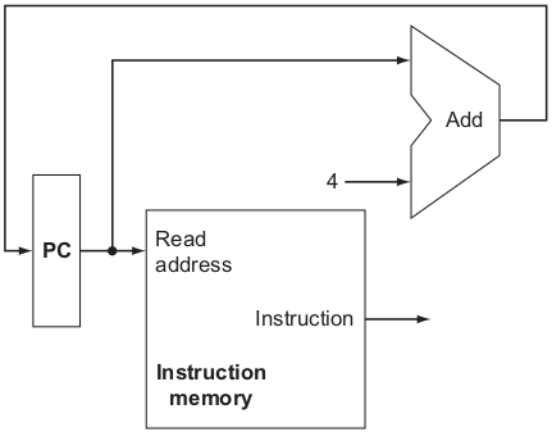

Figure 2: PC update module

5.0.4 JUMP ADDRESS MODULE

This module decodes that 26-bit target in the jump type instructions by firstly making it a multiple of 4 by left shifting by 2 and then prepending the first 4 bits of the Program Counter PC.

Listing 2: Jump Address Module

```
module jump_addr (
    input wire [25:0] jump_relative_addr;
    input wire [3:0]  pc_upper;
    output reg [31:0] jump_addr;
);

always @(*)
begin
  jump_addr[27:0]  <= jump_relative_addr << 2;
  jump_addr[31:28] <= pc_upper;
end

endmodule
```

5.0.5 Module to read and write data

This module is to read data from 2 registers. The input to this module is the address of the 2 source registers and the address of the register to write and the data to write. The output of this module is the data read from the 2 source registers. See Figure 3.

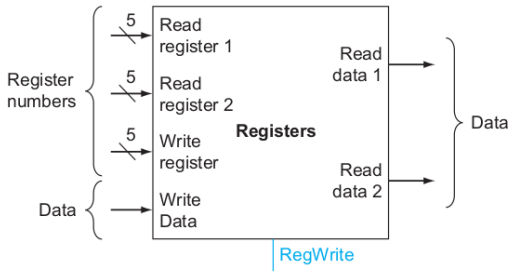

Figure 3: Read and Write module

Listing 3: Read-Write Module Definition in Verilog

```
module rw_module(
    input wire        clock;
    input wire        read_enabled;
    input wire [4:0]  read_addr_s;
    input wire [4:0]  read_addr_t;
    input wire        write_enabled;
    input wire [4:0]  write_addr;
    input wire [31:0] write_data;
    output wire [31:0] outA;
    output wire [31:0] outB;
);

// Implemention will be defined in the later stages of the project
```

5.0.6 ALU MODULE

The Arithematic Logic Unit will be responsible for performing different operations on the operands based on the instruction. This is controlled by the 4 bit ALU operation. The exact number of bits needed will be decided by us later. See Figure 4.

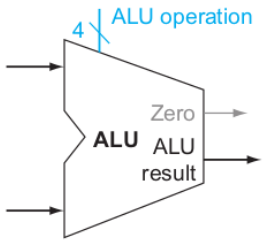

Figure 4: ALU module

Listing 4: ALU Module Declaration

```
module alu_32 (
    input wire [31:0]   input_a;
    input wire [31:0]   input_b;
    input wire [3:0]    control;
    output wire        zero;
    output reg         cout;
    output reg [31:0]   result;
);

// Implemention will be defined in the later stages of the project
```

5.0.7 MODULE FOR BRANCHING

This module is used to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. See Figure 5.

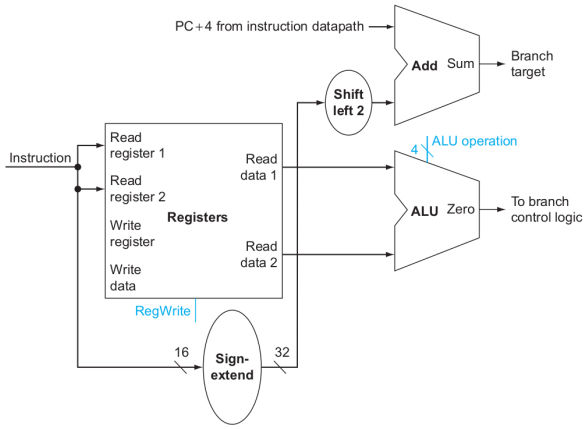

Figure 5: Module for branching

6 Registers and Memory

All details mentioned in this section remain the same regardless of the approach.

6.1 Memory

The memory will be on the Block RAM (BRAM). Here are the memory specifications:

- Width = 32 bits
- Number of addresses = 51200
- Min Address bus size = 16 bit

In total, we make use of 51200 * 32 bits = 1,638,400 bits = 1600 Kibits (1 Kibit = 1024 bits)

6.2 Registers

The register file will be implemented on LUTs. All registers are of 32 bits in size, expect the carry/overflow register which is of 1 bit. We will have the following registers:

- s type (s0 - s7)
- t type (t0 - t9)
- zero
- global pointer (gp)
- stack pointer (sp)
- frame pointer (fp)
- Return address
- Instruction register
- Address registers
- hi
- lo
- Carry/Overflow register

7 Work Done - Member wise

The Table 1 shows work done member wise over the last 2 weeks. Although we have made distinctions, these are based on the area in which the respective member spent most of his time researching and working. All of us have worked collaboratively on all topics and contributed equally to the project.

Table 1: Tasks Achieved - Member Wise

Team Members	Responsibilities
Hrriday	Researched the BRAM available on BASYS, planned the registers and memory management
Kishan	Decided instructions to implement and their opcode, researched upon the MIPS architecture to plot the expected modules and construct the final Data Path
Guntas	Researched the circuit level implementation and contributed to construction of the Data Path. Wrote the outline verilog code modules based on this.
Pranav	Researched existing liturature related to MIPS architechture and assisted other team members in all aspects

8 Tasks Remaining

The project will be divided into the following tasks:

1. **Verilog code:**
   - Implement modules in verilog code and thread them together.
2. **Memory Management:**
   - Bifurcation of Block RAM memory into data segment, code segment etc.
3. **Output Handling:**
   - Implement memory-mapped I/O for interfacing with the 7-segment display.
   - If time permits, display the output on a monitor using VGA port.
4. **Simulation and Verification:**
   - Test and verify the processor's functionality using simulation tools.
   - Write benchmark programs using assembly code and convert them into MIPS-understandable binary executable code for running on the FGPA board.

9 Key Milestones / Timeline

The project is expected to achieve the following milestones:

- **Milestone 1:** Conversion of MIPS ISA to BASYS 3 compatible instruction set. ***(1 Week)***
- **Milestone 2:** Finalizing processor design. ***(2 Weeks)***
- **Milestone 3:** Programming in Verilog, integration of BRAM with memory management and loading of program data from BRAM. ***(2 Weeks)***
- **Milestone 4:** Successful implementation of memory-mapped I/O. ***(1 Week)***
- **Milestone 5:** Processor simulation and verification. ***(2 Weeks)***
- **Milestone 6:** Final testing and demonstration on the Basys FPGA. ***(2 Weeks)***

REFERENCES

David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013. ISBN 0124077269.

A APPENDIX

MIPS Core Instructions	Name	Assembly Code	32-bit Instruction					
R Type Instructions								
			Opcode (6 bits)	rs (5 bits)	rt (5 bits)	rd (5 Bits)	Shamt (5 bits)	Funct (6 bits)
add	add	add rd, rs, rt	0	rs	rt	rd	0	0x20
add unsigned	addu	addu rd, rs, rt	0	rs	rt	rd	0	0x21
subtract	sub	sub rd, rs, rt	0	rs	rt	rd	0	0x22
subtract unsigned	subu	subu rd, rs, rt	0	rs	rt	rd	0	0x23
and	and	and rd, rs, rt	0	rs	rt	rd	0	0x24
or	or	or rd, rs, rt	0	rs	rt	rd	0	0x25
nor	nor	nor rd, rs, rt	0	rs	rt	rd	0	0x27
shift left logical	sll	sll rd, rs, rt	0	rs	rt	rd	shamt	0x00
shift right logical	srl	srl rd, rs, rt	0	rs	rt	rd	shamt	0x02
jump register	jr	jr rs	0	rs	0	0	0	0x08
set less than	slt	slt rd, rs, rt	0	rs	rt	rd	0	0x2a
set less than unsigned	sltu	sltu rd, rs, rt	0	rs	rt	rd	0	0x2b
multiply	mult	mult rs, rt	0	rs	rt	0	0	0x18
multiply unsigned	multu	multu rs, rt	0	rs	rt	0	0	0x19
divide	div	div rs, rt	0	rs	rt	0	0	0x1a
divide unsigned	divu	divu rs, rt	0	rs	rt	0	0	0x1b
move from Hi	mfhi	mfhi rd	0	0	0	rd	0	0x10
move from Lo	mflo	mflo rd	0	0	0	rd	0	0x12

Figure 6: R Type instructions

MIPS Core Instructions	Name	Assembly Code	32-bit Instruction			
I Type Instructions						
			Opcode (6 bits)	rs (5 bits)	rt (5 bits)	Immediate (16 bits)
add immediate	addi	addi rt, rs, imm	8	rs	rt	imm
add immediate unsigned	addiu	addiu rt, rs, imm	9	rs	rt	imm
and immediate	andi	andi rt, rs, imm	0xc	rs	rt	imm
or immediate	ori	ori rt, rs, imm	0xd	rs	rt	imm
load word	lw	lw rt, address	0x23	rs	rt	Offset
store word	sw	sw rt, address	0x2b	rs	rt	Offset
load halfword unsigned	lhu	lhu rt, address	0x25	rs	rt	Offset
store halfword	sh	sh rt, address	0x29	rs	rt	Offset
load byte unsigned	lbu	lbu rt, address	0x24	rs	rt	Offset
store byte	sb	sb rt, address	0x28	rs	rt	Offset
branch on equal	beq	beq rs, rt, label	4	rs	rt	Offset
branch on not equal	bne	bne rs, rt, label	5	rs	rt	Offset
set less than immediate	slti	slti rt, rs, imm	0xa	rs	rt	imm
set less than immediate unsigned	sltiu	sltiu rt, rs, imm	0xb	rs	rt	imm

Figure 7: I Type instructions

MIPS Core Instructions	Name	Assembly Code	32-bit Instruction					
J Type Instructions								
			Opcode (6 bits)	Address (26 bits)				
jump	j	j target	2	target				
jump and link	jal	jal target	3	target				
Special Instruction(s)								
nop	nop	nop	0	0	0	0	0	0

Figure 8: J Type instructions and a Special Instructions