FULL FPGA IMPLEMENTATION OF 32-BIT FSM-BASED MULTI-STATE MIPS PROCESSOR FINAL 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 FSM-based Multi-State MIPS 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 is displayed on a 7-segment LED display using memory-mapped I/O. Code for the full implementation available at `https://github.com/guntas-13/mips-processor-basys3` along with all the versions. Video demonstration of `mips_v2.0` is available at `https://www.youtube.com/watch?v=jQGQIGTQDpc` [Factorial] and `https://www.youtube.com/watch?v=N_pT4MgwUFg` [Fibonacci].

2 OVERVIEW

The following report is an extensive study regarding the implementation of MIPS32 Instruction Set on the Basys3 FPGA Board using FSM-based implementation section 4. We present forth a FSM-Based Implementation. We start by proposing the ISA in section 3. Further the section 5 provides a description (with some Verilog code snippets) of the major **Modules**. These modules, along with their datapath Figure 1, have been derived from RTL Schematic from our verilog codes. Initial motivation has been derived from Patterson & Hennessy (2013). This is followed by section 6, which contains the plan for memory management using BRAM on the FPGA board. section 7 details the several versions of our MIPS processor over many iterations. section 8 discusses the member wise contribution and section 9 describes tasks and milestones completed.

[*]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
*Order based on Roll Numbers Only.

3 FPGA-implementable MIPS ISA in our Implementation

Covering almost all categories of operations, our chosen ISA expands upon what most resources base their implementation on. Using this ISA, the user can write even recursive functions in a very high-level MIPS assembly code.

Table 1: Our MIPS ISA

Instruction	Format	Operation
`sll $rd, $rt, shamt`	R-Type	`R[$rd] ← R[$rt] << shamt`
`srl $rd, $rt, shamt`	R-Type	`R[$rd] ← R[$rt] >> shamt`
`add $rd, $rs, $rt`	R-Type	`R[$rd] ← R[$rs] + R[$rt]`
`sub $rd, $rs, $rt`	R-Type	`R[$rd] ← R[$rs] - R[$rt]`
`and $rd, $rs, $rt`	R-Type	`R[$rd] ← R[$rs] & R[$rt]`
`or $rd, $rs, $rt`	R-Type	`R[$rd] ← R[$rs] | R[$rt]`
`nor $rd, $rs, $rt`	R-Type	`R[$rd] ← ~ (R[$rs] | R[$rt])`
`slt $rd, $rs, $rt`	R-Type	`R[$rd] ← (R[$rs] < R[$rt])`
`slti $rt, $rs, imm`	I-Type	`R[$rt] ← (R[$rs] < SignExt(imm))`
`addi $rt, $rs, imm`	I-Type	`R[$rt] ← R[$rs] + SignExt(imm)`
`andi $rt, $rs, imm`	I-Type	`R[$rt] ← R[$rs] & ZeroExt(imm)`
`ori $rt, $rs, imm`	I-Type	`R[$rt] ← R[$rs] | ZeroExt(imm)`
`lw $rt, imm($rs)`	I-Type	`R[$rt] ← Mem[R[$rs] + SignExt(imm)]`
`sw $rt, imm($rs)`	I-Type	`Mem[R[$rs] + SignExt(imm)] ← R[$rt]`
`beq $rs, $rt, imm`	I-Type	`if (R[$rs] == R[$rt])` `PC ← PC + 1 + (SignExt(imm))`
`j address`	J-Type	`PC ← {PC[31:26], address}`
`jal address`	J-Type	`R[31] ← PC + 1` `PC ← {PC[31:26], address}`
`mult $rs, $rt`	R-Type	`{HI, LO} ← R[$rs] * R[$rt]`
`div $rs, $rt`	R-Type	`LO ← R[$rs] / R[$rt]` `HI ← R[$rs] % R[$rt]`
`jr $rs`	R-Type	`PC ← R[$rs]`
`mfhi $rd`	R-Type	`R[$rd] ← HI`
`mflo $rd`	R-Type	`R[$rd] ← LO`
`sysend`	Custom	`goto SINK`

4 FSM Based Implementation

Our approach uses Finite State Machine (FSM) based implementation, and considers every **stage**: `Fetch, Decode, Execute, Memory Access` and `Write-Back` as being made up of one or several **states**. States individually have a combinatorial circuit made up of the same modules. 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 IF 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. States are triggered based on control signals, that depend upon the previous state's output (whether or not it is complete).
We label this processor as **Multi-State**, since every **stage** is complete in their own **multiple states**. **The Clock Period (T_c) is then the time of longest state made up of combinational logic.** Processor specifications are as follows:

- Clock-edge and control signal triggered states
- FSM based processor implementation with each stage composed of one or more states
- Every state has a combinatorial circuit made of modules

- Integer Register Operations
- Word-Addressing (So no byte-level index)
- 2's complement System
- Register File on LUTs while Memory on BRAM
- Modules as mentioned in the report below
- All 32 Registers as specified by MIPS.

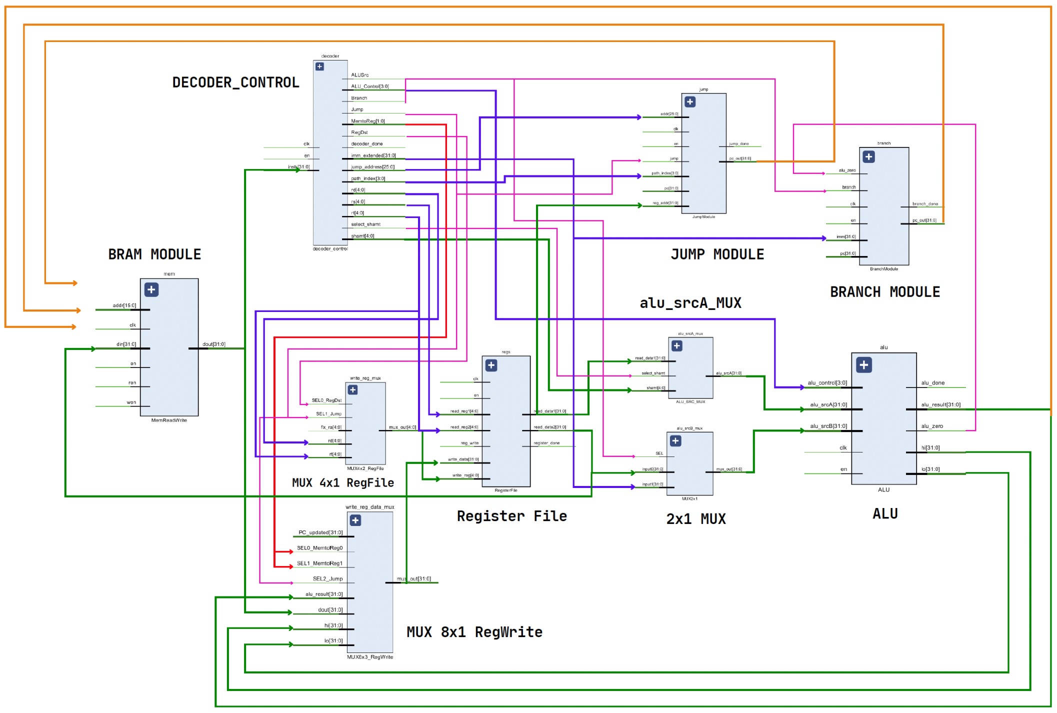


Figure 1: Modified Datapath in our Implementation

5 Modules

5.1 Top Module - FSM Control Unit

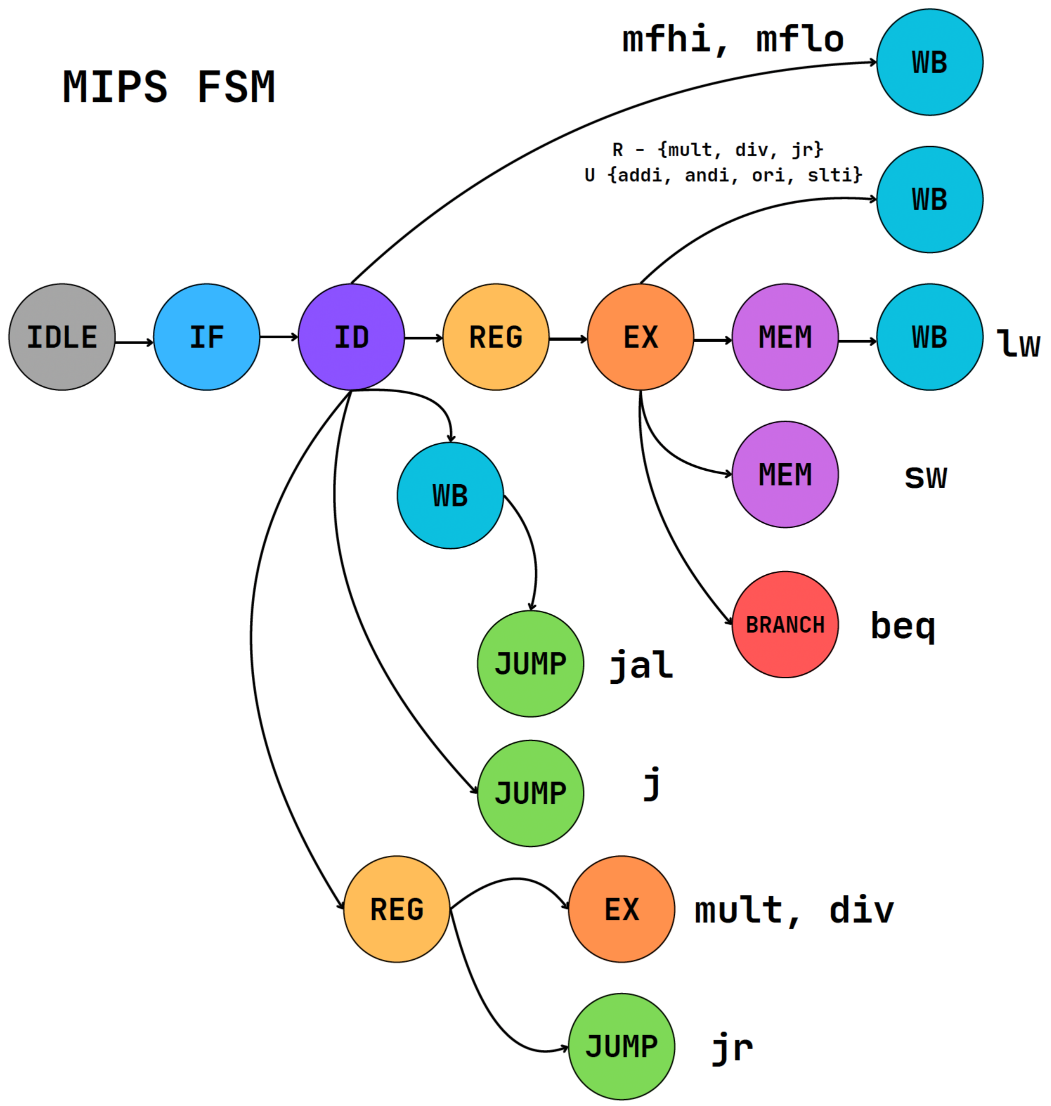


Figure 2: FSM illustrating the various paths in the Top Module.

This FSM describes the top module or the main control unit. This has several paths, which contain stages. Every path corresponds to an instruction, which is labeled alongside the last stage. A stage (example: IF) can be made up of one or more state (example: IF has 3 redundant states). All the last stages are connected to the IF stage (this connection is not shown in the diagram). There is a flush state, which is entered when all the instructions are completed (this state is not shown in the diagram). Details are mentioned in the Appendix A.

5.2 BRAM Module

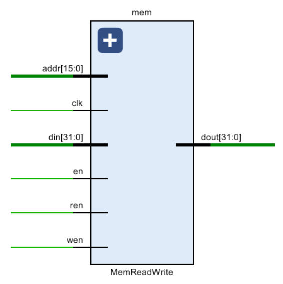


Figure 3: BRAM Read-Write Module

Listing 1: BRAM Read-Write Module in Verilog

```
module MemReadWrite(
    input clk,
    input en,
    input ren,
    input wen,
    input [15:0] addr, // This should be word address
    input [31:0] din,
    output [31:0] dout);

blk_mem_gen_0 memory(.clka(clk), .ena((en & ren) | (en & wen)),
.wea(wen), .addra(addr), .dina(din), .douta(dout));

endmodule
```

5.3 The Decoder Control Unit

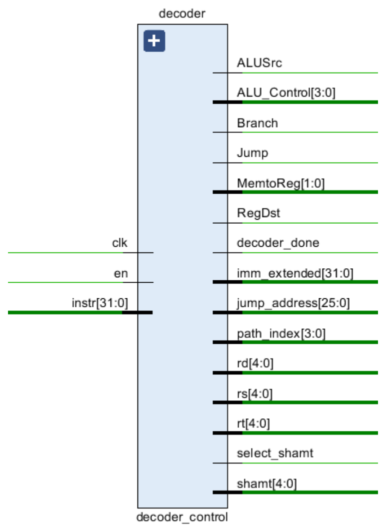


Figure 4: Decoder Control Module

Listing 2: BRAM Read-Write Module in Verilog

```
module decoder_control(
    input clk,
    input en,
    input [31:0] instr,
    output reg RegDst,
    /*
    will go as select line in the
    MUX for instr[20:16] (rt) or instr[15:11] (rd)
    */
    output reg Jump,
    output reg Branch,
    output reg MemRead,
    // this is the input ren in MemReadWrite.v
    output reg [1:0] MemtoReg,
    /*
    These are the 2 select lines to choose
    from 4 options in the MUX for the write data
    to the register file -> 00: alu_result (in alu.v),
    01: dout (in MemReadWrite.v), 10: hi, 11: lo (both in alu.v)
    */
    output reg [3:0] ALU_Control,
    /*
    this is the input [3:0] alu_control
    in the alu.v
    */
    output reg MemWrite, // this is the input wen in MemReadWrite.v
    output reg ALUSrc,
    /*
    this is the select line in the MUX for the
    second operand of the ALU (either the immediate value
    or the value from the register file)
    */
    output reg RegWrite,
    /*
    this is the input reg_write in
    the register_file.v
    */
    output reg [31:0] imm_extended,
    /*
    this is the immediate
    value extended to 32 bits
    */
    output reg [4:0] rs,
    output reg [4:0] rt,
    output reg [4:0] rd,
    output reg [4:0] shamt,
    output reg [25:0] jump_address,
    output reg [3:0] path_index,
    output reg decoder_done,
    output reg select_shamt
);
```

5.4 Register File

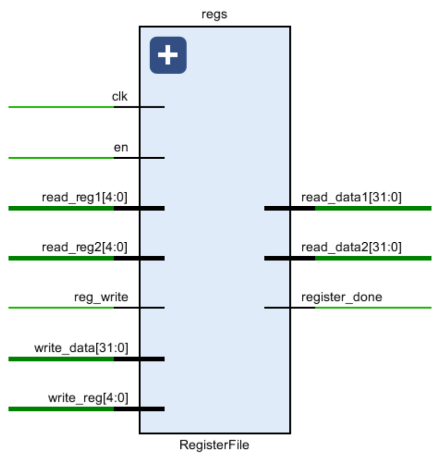


Figure 5: Register File Module

Listing 3: Register File Module

```
module RegisterFile(
    input clk,
    input en,
    input reg_write,
    input [4:0] read_reg1,
    input [4:0] read_reg2,
    input [4:0] write_reg,
    input [31:0] write_data,
    output reg [31:0] read_data1,
    output reg [31:0] read_data2,
    output reg register_done
);
```

5.5 ALU

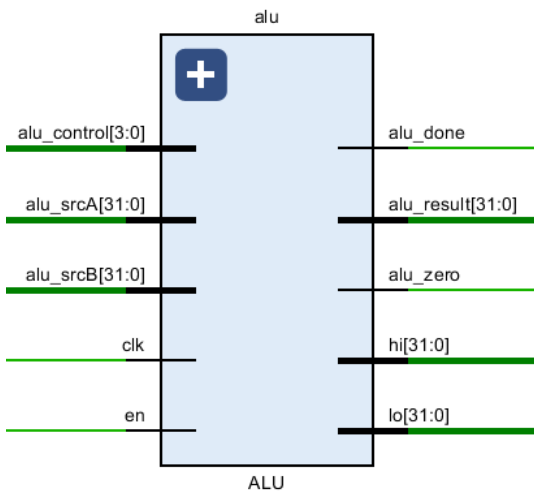


Figure 6: The ALU Module

Listing 4: ALU Module Definition in Verilog

```
module ALU(
    input clk,
    input en,
    input [3:0] alu_control,
    input [31:0] alu_srcA,          // this corresponds to $rs or shamt
    input [31:0] alu_srcB,          // this corresponds to $rt
    output reg [31:0] alu_result,   // this corresponds to $rd
    output reg [31:0] hi,
    output reg [31:0] lo,
    output reg overflow,
    output reg alu_done,
    output alu_zero
);

    parameter ADD  = 4'b0000;
    parameter SUB  = 4'b0001;
    parameter AND  = 4'b0010;
    parameter OR   = 4'b0011;
    parameter NOR  = 4'b0100;
    parameter SLT  = 4'b0101;
    parameter SLL  = 4'b0110;
    parameter SRL  = 4'b0111;
    parameter MULT = 4'b1000;
    parameter DIV  = 4'b1001;
```

5.6 Branch module

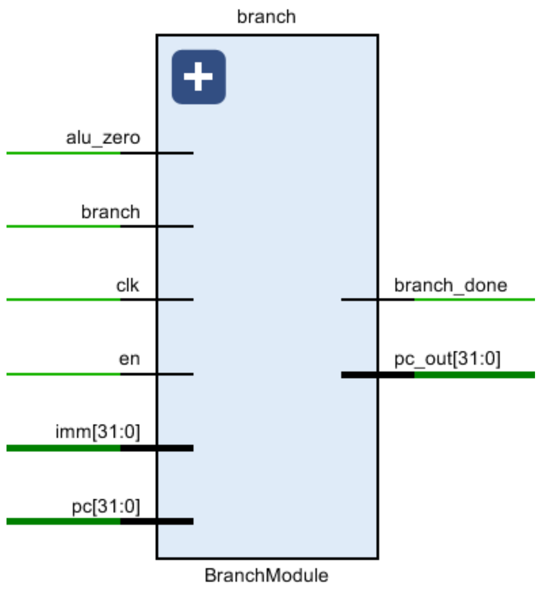


Figure 7: Branch module

Listing 5: Branch Module

```verilog
module BranchModule(
    input clk,
    input en,
    input branch,
    input alu_zero,
    input [31:0] imm,
    input [31:0] pc,
    output reg [31:0] pc_out,
    output reg branch_done
);

initial begin
    branch_done = 1'b0;
end

always @ (posedge (clk & en)) begin
    if (en) begin
        pc_out <= (branch & en & alu_zero)? pc + imm: pc;
        branch_done <= 1'b1;
    end
    else begin
        branch_done <= 1'b0;
    end
end

endmodule
```

5.7 Jump Module

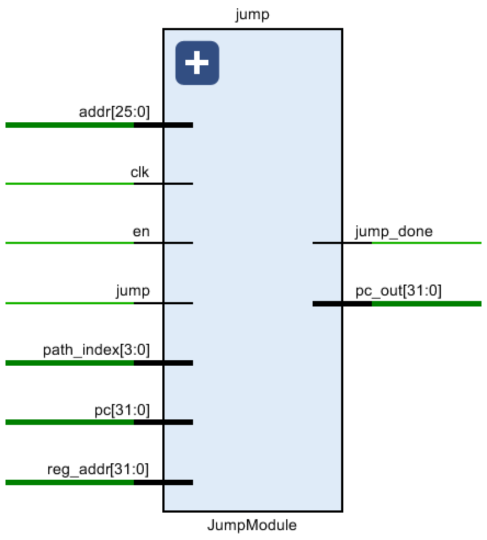


Figure 8: Jump Module

Listing 6: Jump Module

```verilog
module JumpModule(
    input clk,
    input en,
    input jump,
    input [31:0] pc,
    input [25:0] addr,
    input [3:0] path_index,
    input [31:0] reg_addr,
    output reg [31:0] pc_out,
    output reg jump_done
);

    initial begin
        jump_done = 1'b0;
        pc_out <= 32'd0;
    end

    always @ (posedge (clk & en)) begin
        if (en) begin
            if ((path_index==4'd5 | path_index==4'd6)) begin
                pc_out <= {pc[31:26], addr};
            end
            else if(path_index==4'd8) begin
                pc_out <= reg_addr;
            end
            else begin
                pc_out <= pc;
            end
            jump_done <= 1'b1;
        end
        else begin
            jump_done <= 1'b0;
        end
    end

endmodule
```

5.8 2x1 MUX for ALU Source B

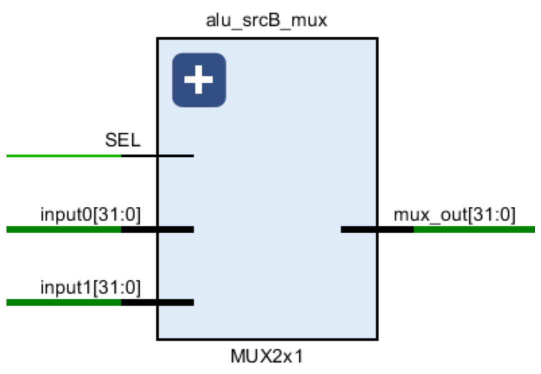


Figure 9: 2x1 MUX

Listing 7: MUX2x1 Module

```verilog
module MUX2x1 #(parameter WIDTH = 32) (
    input [WIDTH-1:0] input0,
    input [WIDTH-1:0] input1,
    input SEL,
    output reg [WIDTH-1:0] mux_out
);

    always @(*) begin
        case (SEL)
            1'b0: mux_out = input0;
            1'b1: mux_out = input1;
            default: mux_out = 32'bx;
        endcase
    end
endmodule
```

5.9 New 2x1 MUX for ALU Source A

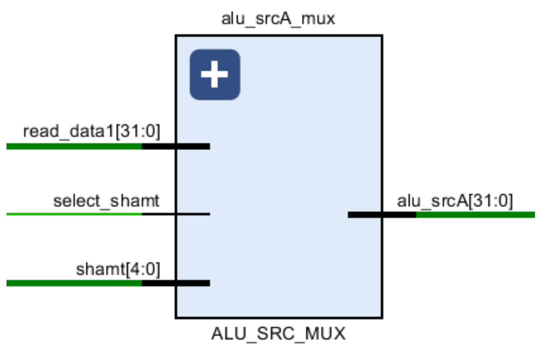


Figure 10: 2x1 ALU Src A MUX

Listing 8: MUX2x1 Module

```
module ALU_SRC_MUX (
    input [31:0] read_data1,
    input [4:0] shamt,
    input select_shamt,
    output [31:0] alu_srcA
);

    assign alu_srcA = (select_shamt) ? {27'b0, shamt} : read_data1;
    // Zero-extend shamt to 32 bits

endmodule
```

5.10 4x1 MUX for Read Data 2 of Register File

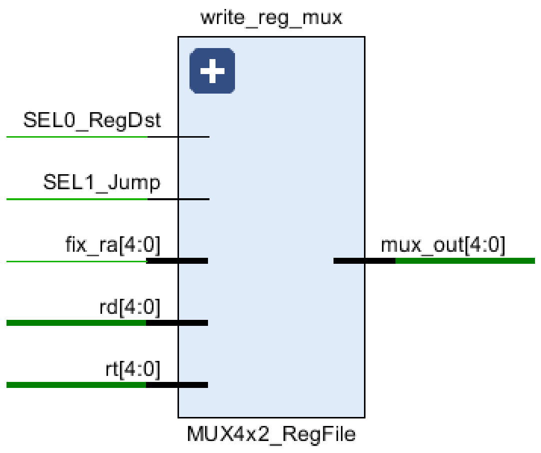


Figure 11: 4x1 MUX for `$rt, $rd, 31`

Listing 9: MUX4x1 Module

```
module MUX4x1_RegFile(
    input [4:0] rt,
    input [4:0] rd,
    input [4:0] fix_ra,
    input SEL0_RegDst,
    input SEL1_Jump,
    output reg [4:0] mux_out
);

    always @(*) begin
        case ({SEL1_Jump, SEL0_RegDst})
            2'b00: mux_out = rt;         // For I-Types RegDst = 0
            2'b01: mux_out = rd;         // For R-Types RegDst = 1
            2'b10: mux_out = fix_ra;     // For J-Types jal R[31]
            // ($ra) = PC + 1
            default: mux_out = 5'bx;
        endcase
    end

endmodule
```

5.11 8x1 MUX for Write Data of Register File

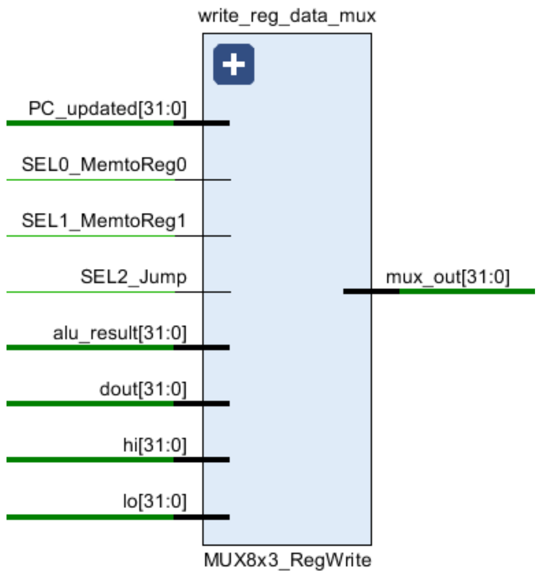


Figure 12: 8x1 MUX for `alu_result, mem_out, hi, lo, PC_updated`

Listing 10: MUX8x1 Module

```
module MUX8x1_RegWrite(
    input [31:0] alu_result,
    input [31:0] dout,
    input [31:0] hi,
    input [31:0] lo,
    input [31:0] PC_updated,
    // For jal instruction need PC + 1 to be written to register file
    input SEL0_MemtoReg0,
    input SEL1_MemtoReg1,
    input SEL2_Jump,
    // For jal instruction need PC + 1 to be written to register file
    output reg [31:0] mux_out
);

    always @(*) begin
        case ({SEL2_Jump, SEL1_MemtoReg1, SEL0_MemtoReg0})
            3'b000: mux_out = alu_result;
            3'b001: mux_out = dout;
            3'b010: mux_out = hi;
            3'b011: mux_out = lo;
            3'b100: mux_out = PC_updated;
            default: mux_out = 32'bx;
        endcase
    end

endmodule
```

6 BRAM Memory

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

- Width = 32 bits
- Number of addresses = 51200
- Data Segment Start Address = 6300
- Address = Word-Addressable

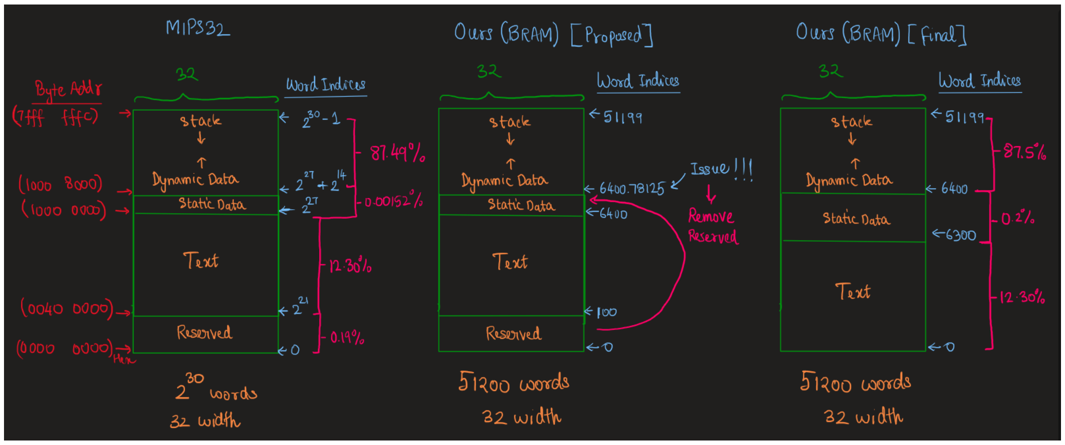
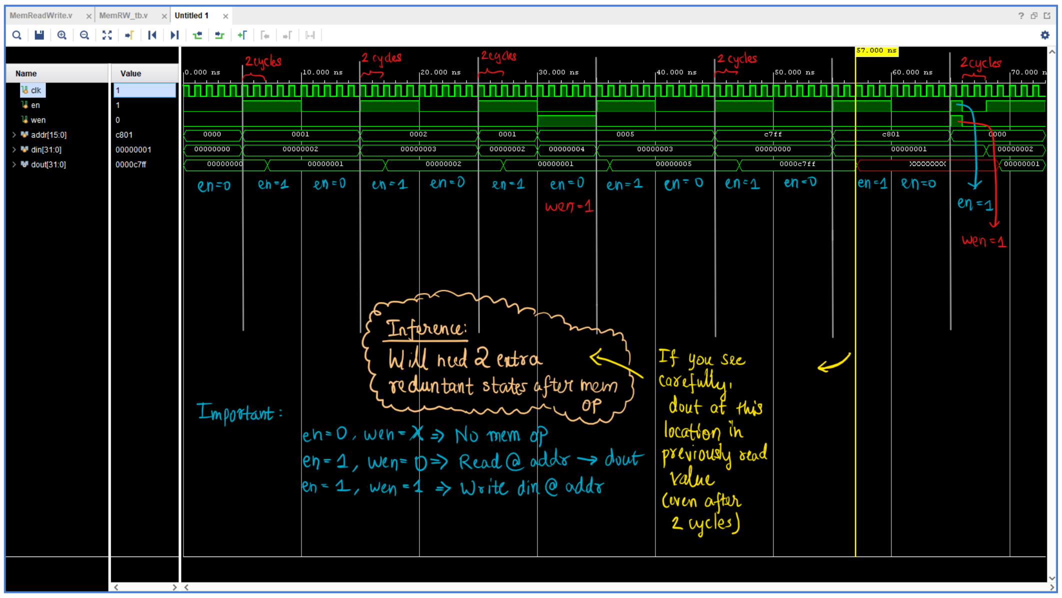


Figure 13: DRAM Organization in comparison to the MIPS Memory Organization.



Important:
en = 0, wen = X ⇒ No mem op
en = 1, wen = 0 ⇒ Read @ addr → dout
en = 1, wen = 1 ⇒ Write din @ addr

Inference:
Will need 2 extra redundant states after mem op

If you see carefully, dout at this location in previously read value (even after 2 cycles)

Figure 14: Clock Cycles taken by the BRAM Module for reading of data.

7 Versions of our Code

1. `mips_v1.0`: Uses done and complex sequential logic for feedback based FSM. Each module (except MEM) have `clk` and `done` signal. **Simulation-ready code**.
2. `mips_v1.1`: Simple optimization. `en` on posedge and `done` checking on negedge (instead of posedge). **Simulation-ready code**.
3. `mips_v2.0`: Stable extension of `mips_v1` for FPGA implementation with 7-segment LED. **Factorial of 7 and computation of Fibonacci numbers (upto 21 can be displayed on the seven segment display) works on this and shown in the *demonstration***.
4. `mips_v2.1`: Unstable and incomplete extension of `mips_v1.1` for FPGA implementation. Multiple Driven nets issue at `en` of all modules. Fix: Add mux at each `en` that selects which clock edge will drive the `en` (sel = clk, 0 - negedge, 1 - posedge).
5. `mips_v3.0`: Unstable extension of `mips_v2`. Kishan's attempt to remove `done` signals and simplify sequential logic. **Simulation not run**.
6. `mips_v3.1`: (Multi-State Processor - Each state takes exactly 1 clock cycle) Stable extension of `mips_v2` with successful removal of `done` signals and simplification of modules to combinatorial blocks (without `clk`). Further optimized the memory read states and reduced the overall clock cycles by 1 for memory read. But now, decode consumes 2 clock cycles.

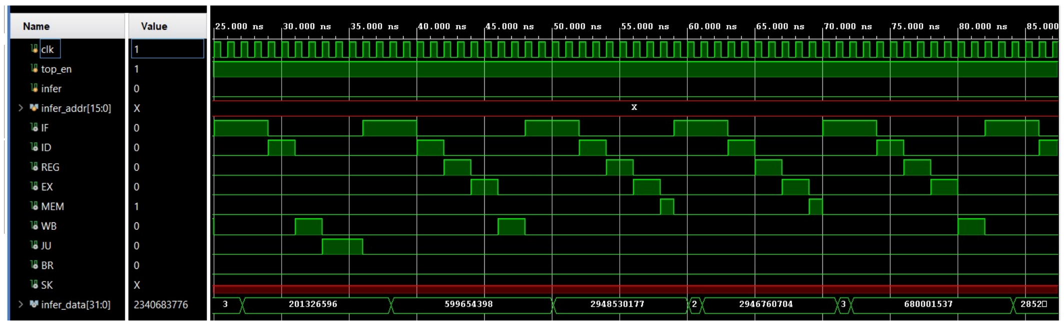


Figure 15: `mips_v1.0` showing multiple cycle for each state wherein each stage may emcompass multiple states.

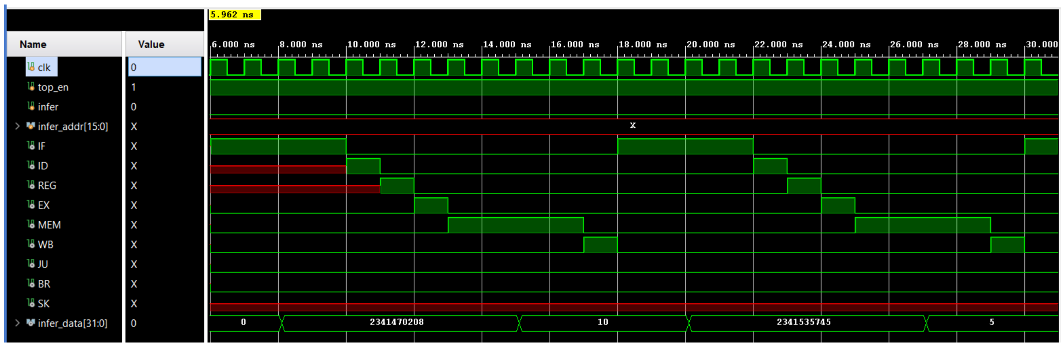
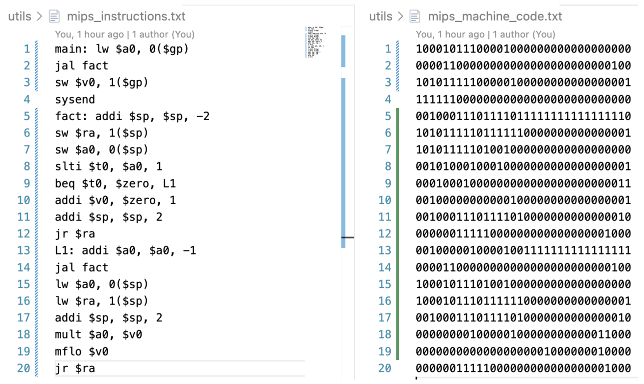


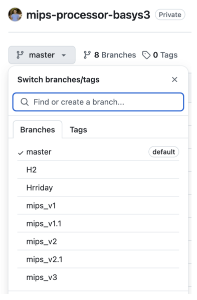
Figure 16: `mips_v3.0` showing single cycle for each state wherein each stage may emcompass multiple states.


utils > mips_instructions.txt
You, 1 hour ago | 1 author (You)
utils > mips_machine_code.txt
You, 1 hour ago | 1 author (You)

```
main: lw $a0, 0($gp)
jal fact
sw $v0, 1($gp)
sysend
fact: addi $sp, $sp, -2
sw $ra, 1($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 2
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 1($sp)
addi $sp, $sp, 2
mult $a0, $v0
mflo $v0
jr $ra
```

```
10001011100001000000000000000000
00001100000000000000000000000100
10101111100001000000000000000001
11111100000000000000000000000000
00100011101111011111111111111110
10101111101111110000000000000001
10101111101001000000000000000000
00101000100010000000000000000001
00010001000000000000000000000011
00100000000000100000000000000001
00100011101111010000000000000010
00000011111000000000000000001000
00100000100001001111111111111111
00001100000000000000000000000100
10001011101001000000000000000000
10001011101111110000000000000001
00100011101111010000000000000010
00000000100000100000000000011000
00000000000000000001000000010000
00000011111000000000000000001000
```

(a) Our MIPS to Machine Code Parser.



(b) Various Iterations of our Processor.

Figure 17: Combined view of the MIPS to Machine Code Parser and Processor Iterations.

8 Work Done - Member wise

The Table 2 shows work done member wise. 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 2: Tasks Achieved - Member Wise

S.No.	Team Members	Responsibilities
1	Hrriday	Contributed to the Verilog code and designing of the datapath. Handled memory input and output and BRAM constraints.
2	Kishan	Contributed to the Verilog code and designing of the FSM (states and paths).
3	Guntas	Contributed to the Verilog code and ideation of the datapath, made the presentation and report.
4	Pranav	Wrote the parser to convert MIPS instructions to binary code followed by generation of the coe file.

9 Tasks Achieved/Key Milestone

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)***

10 FPGA Setup

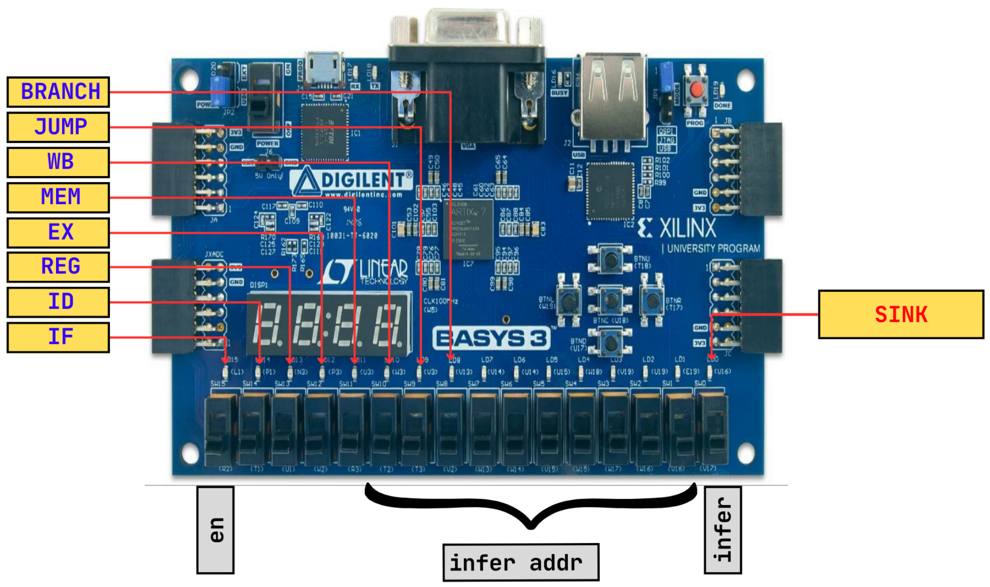


Figure 18: FPGA with Memory Mapped I/O showing the States in our implementation.

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 - FSM Control Unit (Top Module)

State Transitions and Actions

1. **IDLESRC (Idle State)**:
    - The control unit remains in this state until the top-level enable signal (`top_en`) is asserted.
    - If `top_en` is high, the state transitions to `FETCH`.
2. **FETCH (Instruction Fetch)**:
    - In this state, the control unit fetches an instruction from memory by asserting `mem_en`, `mem_ren` (read enable), and setting the memory address to the program counter (`pc[15:0]`).
    - The instruction fetched is stored in `instr_reg`.
    - The control unit asserts the `IF` signal to indicate that the instruction is being fetched.
    - After fetching the instruction, the state transitions to `RED1`.
3. **RED1, RED2, RED3 (Read Stages)**:
    - These states are intermediate stages where the control unit does minimal work other than proceeding to the next state.
    - The state progresses from `RED1` to `RED3`, ultimately reaching the `DECODE` state.
4. **DECODE (Instruction Decode)**:
    - The control unit decodes the fetched instruction using the `decoder` module.

- If `decoder_done` is asserted (meaning decoding is complete), the control unit checks the `path_index` from the `decoder`.
- Based on the `path_index`, it decides the next state. For example:
  - If `path_index == 0` or `path_index == 6`, it transitions to `REGWRITE`.
  - If `path_index == 5`, it transitions to `JUMP`.
  - If `path_index == 9`, it transitions to `SINK`.
  - Otherwise, the state transitions to `REGFILE`.

5. **REGFILE (Register File Access)**:
   - In this state, the control unit enables the register file access by asserting `reg_en`.
   - It retrieves the values from the registers and prepares them for the execution stage.
   - Based on `path_index`, the state transitions as follows:
     - If `path_index == 1, 2, 3, 4, 7`, the state moves to `EXECUTE`.
     - If `path_index == 8`, the state moves to `JUMP`.
6. **EXECUTE (ALU Execution)**:
   - The control unit initiates ALU operations by asserting `alu_en` and the appropriate `ALU_Control` signal.
   - It waits for the `alu_done` signal to be asserted, indicating that the ALU operation is complete.
   - Based on the `path_index`, the control unit decides the next state:
     - If `path_index == 1`, it transitions to `REGWRITE`.
     - If `path_index == 2` or `path_index == 3`, it transitions to `MEMORY`.
     - If `path_index == 7`, it transitions to `FETCH`.
     - If `path_index == 4`, it transitions to `BRANCH`.
7. **MEMORY (Memory Access)**:
   - Depending on the `path_index`, the control unit either reads from or writes to memory.
   - If `path_index == 2`, the state transitions to `RED4` to handle a memory read.
   - If `path_index == 3`, the state transitions to `FETCH` to complete a memory write operation.
8. **RED4, RED5, RED6 (Memory Read Stages)**:
   - These states are intermediate stages for handling memory reads and writes.
   - After `RED6`, the control unit transitions to `REGWRITE`.
9. **REGWRITE (Writeback)**:
   - The control unit asserts `WB` to enable the write-back to the register file.
   - Depending on `path_index`, the state transitions to either `FETCH` or `JUMP`.
   - Once the write-back operation is complete, the state returns to `FETCH`.
10. **JUMP (Jump Handling)**:
    - The control unit handles jump instructions by asserting `JU` and using the jump address from the instruction.
    - Once the jump operation is complete (indicated by `jump_done`), the state transitions back to `FETCH`.
11. **BRANCH (Branch Handling)**:
    - In this state, the control unit manages conditional branches using the branch condition (`alu_zero`) and the branch offset.
    - Once the branch operation is complete (indicated by `branch_done`), the state transitions back to `FETCH`.
12. **SINK (Special Instruction Handling)**:
    - In the `SINK` state, the control unit handles special cases such as inference or debugging.
    - If the `infer` signal is asserted, the control unit reads from a specific memory location and drives the display logic.

Control Signals and Their Purpose

- The control unit uses various control signals (`IF`, `ID`, `REG`, `EX`, `MEM`, `WB`, `JU`, `BR`, `SK`) to indicate the current stage of operation.
- The state machine adjusts these signals based on the current state and the decoded instruction.
- Intermediate states such as `RED1` to `RED6` are used for timing control and sequencing.

Listing 11: Jump Module

```
module ControlUnit(
    input fast_clk,
    input top_en,
    input infer,
    input [9:0] infer_addr,
    output reg ID,
    output reg IF,
    output reg REG,
    output reg EX,
    output reg MEM,
    output reg WB,
    output reg JU,
    output reg BR,
    output reg SK,
    output [6:0] LED_out,
    output [3:0] Anode_Activate
);
    wire [15:0] infer_data;
    assign infer_data = (infer)? mem_dout[15:0]: 16'd0;
    // STATES
    parameter IDLESRC = 4'b0000;
    parameter FETCH = 4'b0001;
    parameter RED1 = 4'b0010;
    parameter RED2 = 4'b0011;
    parameter RED3 = 4'b0100;
    parameter DECODE  = 4'b0101;
    parameter REGFILE  = 4'b0110;
    parameter EXECUTE  = 4'b0111;
    parameter MEMORY  = 4'b1000;
    parameter RED4 = 4'b1001;
    parameter RED5 = 4'b1010;
    parameter RED6 = 4'b1011;
    parameter REGWRITE  = 4'b1100;
    parameter JUMP = 4'b1101;
    parameter BRANCH = 4'b1110;
    parameter SINK = 4'b1111;

    initial begin
        pc <= 32'd0;
        state <= IDLESRC;
        counter <= 4'd0;
        clk <= 1'd1;
    end

    always @ (posedge fast_clk) begin
        if (counter == 25'd1250000) begin
            counter <= 0;
            clk <= ~clk;
        end
        else begin
            counter <= counter + 1;
        end
    end

    always @ (posedge (clk & top_en))
    begin
        case(state)
            IDLESRC: begin
                if (top_en) state <= FETCH;
                else state <= IDLESRC;
            end

    // And so on...
```