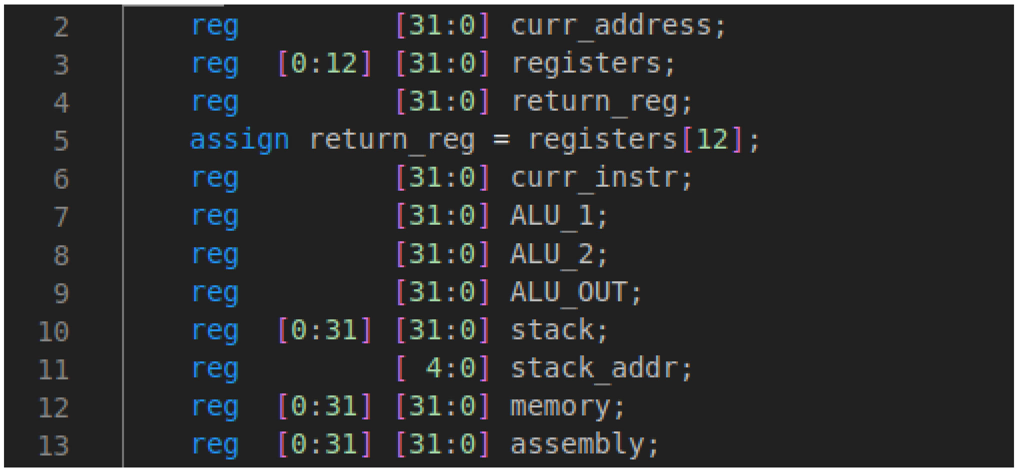
Calvin Passmore

MIPS Final Report

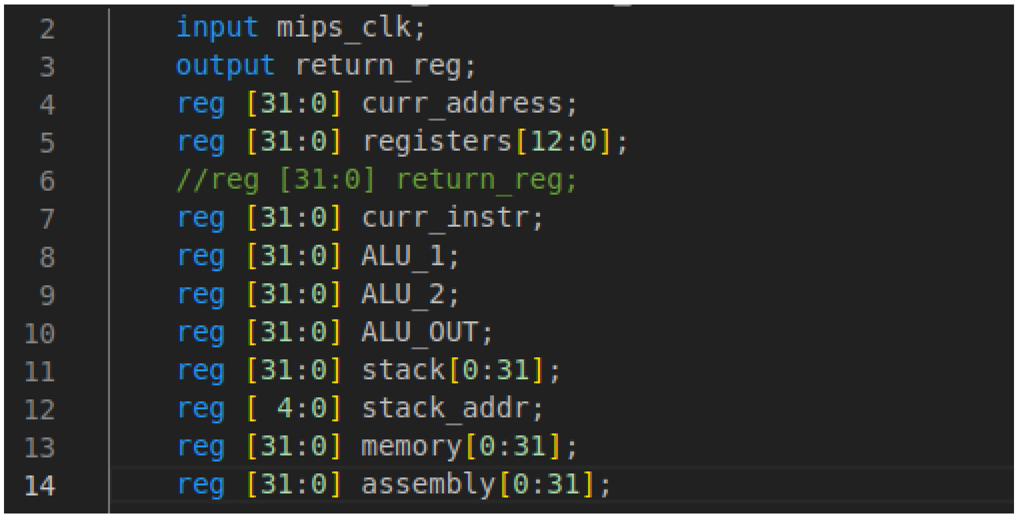
I designed and implemented a simple Microprocessor without Interlocked Pipelined Stages (MIPS). This means that each stage of the processor happens within one clock cycle, and the clock will run relatively slowly. I used a small set of instructions to simplify the assembly code. The instructions I supported are shown in the table under Specification of the Design. In the project, I was only able to use basic Verilog structures because the synthesis tools don't support System Verilog, and there are some Verilog structures the tools don't support either.

When I originally wrote my MIPS processor for simulation, I used the following form for 2D arrays



```
    reg         [31:0] curr_address;
    reg  [0:12] [31:0] registers;
    reg         [31:0] return_reg;
    assign return_reg = registers[12];
    reg         [31:0] curr_instr;
    reg         [31:0] ALU_1;
    reg         [31:0] ALU_2;
    reg         [31:0] ALU_OUT;
    reg  [0:31] [31:0] stack;
    reg         [ 4:0] stack_addr;
    reg  [0:31] [31:0] memory;
    reg  [0:31] [31:0] assembly;
```

When I put the Verilog into the "dc.tcl" script, it didn't like the syntax of the registers, memory, stack, and assembly. It gave me errors about being invalid Verilog, which I knew wasn't the case because my simulation ran perfectly with this syntax using iverilog through Cocotb. Below is what I changed the syntax to in order for the "dc.tcl" script to be happy with it.



```
    input mips_clk;
    output return_reg;
    reg [31:0] curr_address;
    reg [31:0] registers[12:0];
    //reg [31:0] return_reg;
    reg [31:0] curr_instr;
    reg [31:0] ALU_1;
    reg [31:0] ALU_2;
    reg [31:0] ALU_OUT;
    reg [31:0] stack[0:31];
    reg [ 4:0] stack_addr;
    reg [31:0] memory[0:31];
    reg [31:0] assembly[0:31];
```

This was fine, the only real difference was where I had to specify that I was using 2D arrays. Instead of doing "[size1] [size2] name", I used "[size1] name [size2]". After doing a little more research this is a more common/better way to write 2D arrays in Verilog anyway. However; this didn't solve all of the issues.

When I tried to run the above through "place.tcl", and solve different errors, I got to the error "check line 5 near the text [for the issue: 'syntax error'". Meaning that Innovus doesn't agree with dc_shell on how 2D Verilog arrays should be instantiated if Innovus supports 2D arrays at all.

Motivation

I chose this project because I have previously designed processors in other classes, but never got to synthesize them. I thought it would be interesting to see how a processor is laid out in transistors.

Specification of the Design

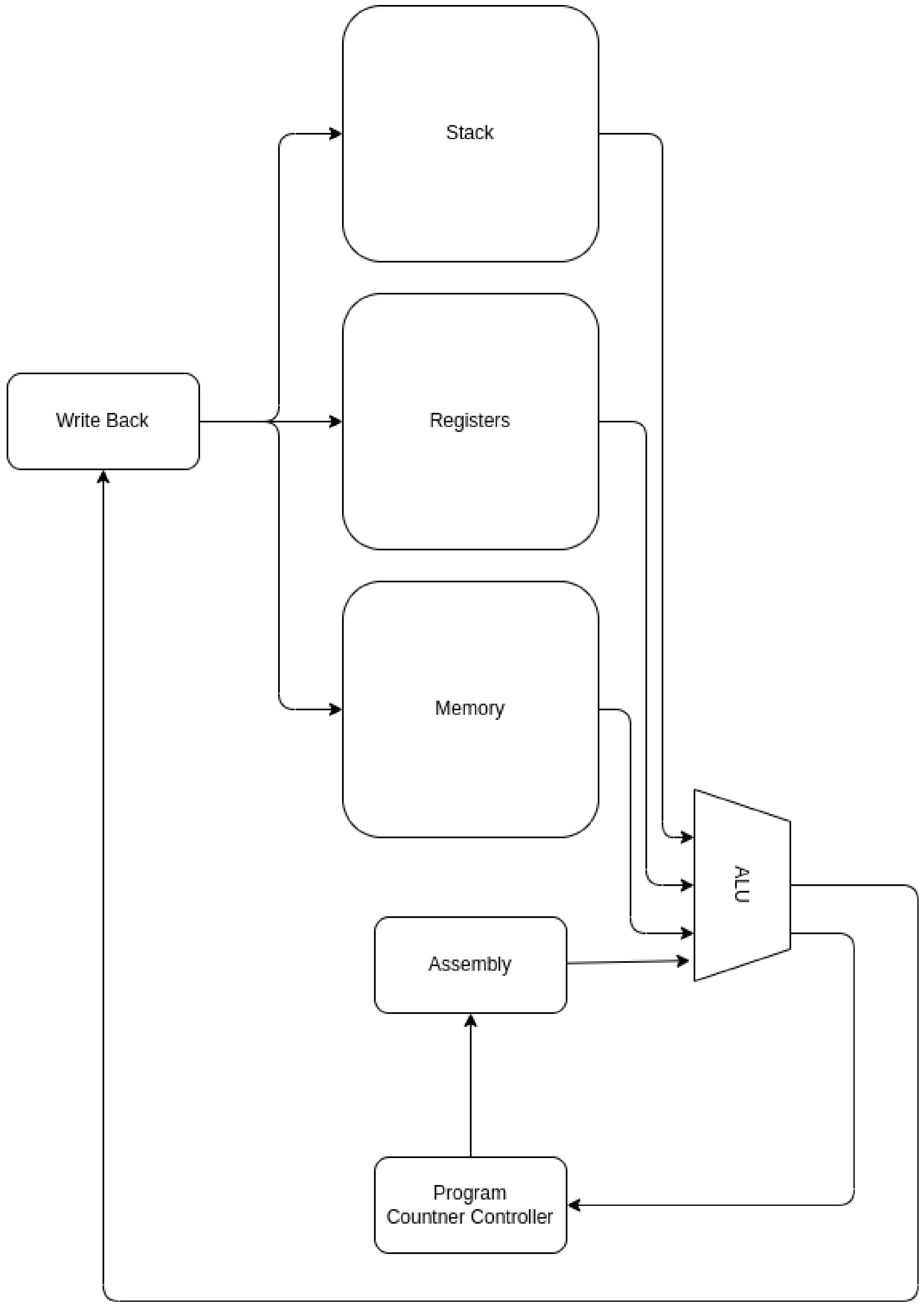
I designed a MIPS processor, that implemented a variety of opcodes. I included four memory types: assembly, stack, memory, and registers. Using these four memory locations and an ALU, the processor is able to do a lot of basic programs. I used my golden model to output the assembly, using C #define macros to make the assembly more human-readable. See the below table for the available OpCodes.

Name	**OpCode**	**Meaning**
NOOP	0x0	Do nothing
Load	0x1	Load from memory into a register
Load Number	0x2	Load a scalar into the register
Store	0x3	Store from the register to memory
Add	0x4	Add two registers
Subtract	0x5	Subtract two registers
XOR	0x6	XOR two registers into a single register
And	0x7	AND individual bits in two registers into one register
Jump	0x8	Jump to address
Jump if non 0	0x9	Jump to address if the given register is not 0
Push	0xA	Push the register value onto the stack
Pop	0XB	Pop the top value off the stack

The OpCodes were implemented in the following manner.

OPCODE	DESTINATION	SOURCE 1	SOURCE 2	Formula
NOOP	-	-	-	-
LOAD	Destination Register	-	Memory Address	*Register = *Memory
LDNM	Destination Register	-	Static Number	*Register = num
STR	Memory Address	-	Source Register	*Memory = *Register
ADD	Destination Register	Source Register 1	Source Register 2	*Destination = *SRC1 + *SRC2
SUB	Destination Register	Source Register 1	Source Register 2	*Destination = *SRC1 - *SRC3
XOR	Destination Register	Source Register 1	Source Register 2	*Destination = *SRC1 XOR *SRC4
AND	Destination Register	Source Register 1	Source Register 2	*Destination = *SRC1 & *SRC5
JMP	Jump address	-	-	Jump to Jump address
JMP0	Jump address	-	Source Register	Jump if *SRC != 0
PUSH	-	-	Source Register	*Stack = *SRC
POP	Destination Register	-	-	*Destination = *Stack

Block Diagram



Simulation Results

For the simulation, I used Cocotb. Cocotb is a Python library/package that reads and simulates HDLs for easy test benches. I chose Cocotb because I have used it previously in my job and it worked well. All the code I included in the Appendix: Code.

I chose to verify that my Verilog was working correctly by comparing the values at each register, memory location, and stack address for each clock cycle to the output of my golden model outputs, for which I had printed the same information. A summary of the results is shown below. The most important thing to watch for is at the end when all memory locations should match, since the assembly I wrote for it output all results to memory.

From the Cocotb simulation

```
***********Done***********
Mem[ 0] = 0xF - 15
Mem[ 1] = 0x3DB - 987
Mem[ 2] = 0x3 - 3
Mem[ 3] = 0xF - 15
Mem[ 4] = 0xC - 12
Mem[ 5] = 0x3 - 3
```

From the golden model output

```
***********Done***********
Mem[ 0] = 0xF - 15
Mem[ 1] = 0x3DB - 987
Mem[ 2] = 0x3 - 3
Mem[ 3] = 0xF - 15
Mem[ 4] = 0xC - 12
Mem[ 5] = 0x3 - 3
```

Synthesis Results

Area Report

```
****************************************
Report : area
Design : s35932
Version: T-2022.03-SP2
Date   : Wed Apr 19 11:44:18 2023
****************************************
```

```
Information: Updating design information... (UID-85)
Library(s) Used:

    No libraries used.

Number of ports:                          2
Number of nets:                          0
Number of cells:                         0
Number of combinational cells:           0
Number of sequential cells:              0
Number of macros/black boxes:            0
Number of buf/inv:                       0
Number of references:                    0

Combinational area:              0.000000
Buf/Inv area:                    0.000000
Noncombinational area:           0.000000
Macro/Black Box area:            0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                 0.000000
Total area:                 undefined
1
```

Timing Report

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : s35932
Version: T-2022.03-SP2
Date   : Wed Apr 19 11:44:18 2023
****************************************

Operating Conditions: TCCOM   Library: fsd0a_a_generic_core_tt1v25c
Wire Load Model Mode: enclosed
```

```
  Startpoint: return_reg (internal pin)
  Endpoint: return_reg (output port clocked by mips_clk)
  Path Group: (none)
  Path Type: max

  Point                                    Incr       Path
  --------------------------------------------------------
  return_reg (out)                         0.00       0.00 r
  data arrival time                                   0.00
  --------------------------------------------------------
  (Path is unconstrained)


1
```

Violation Report

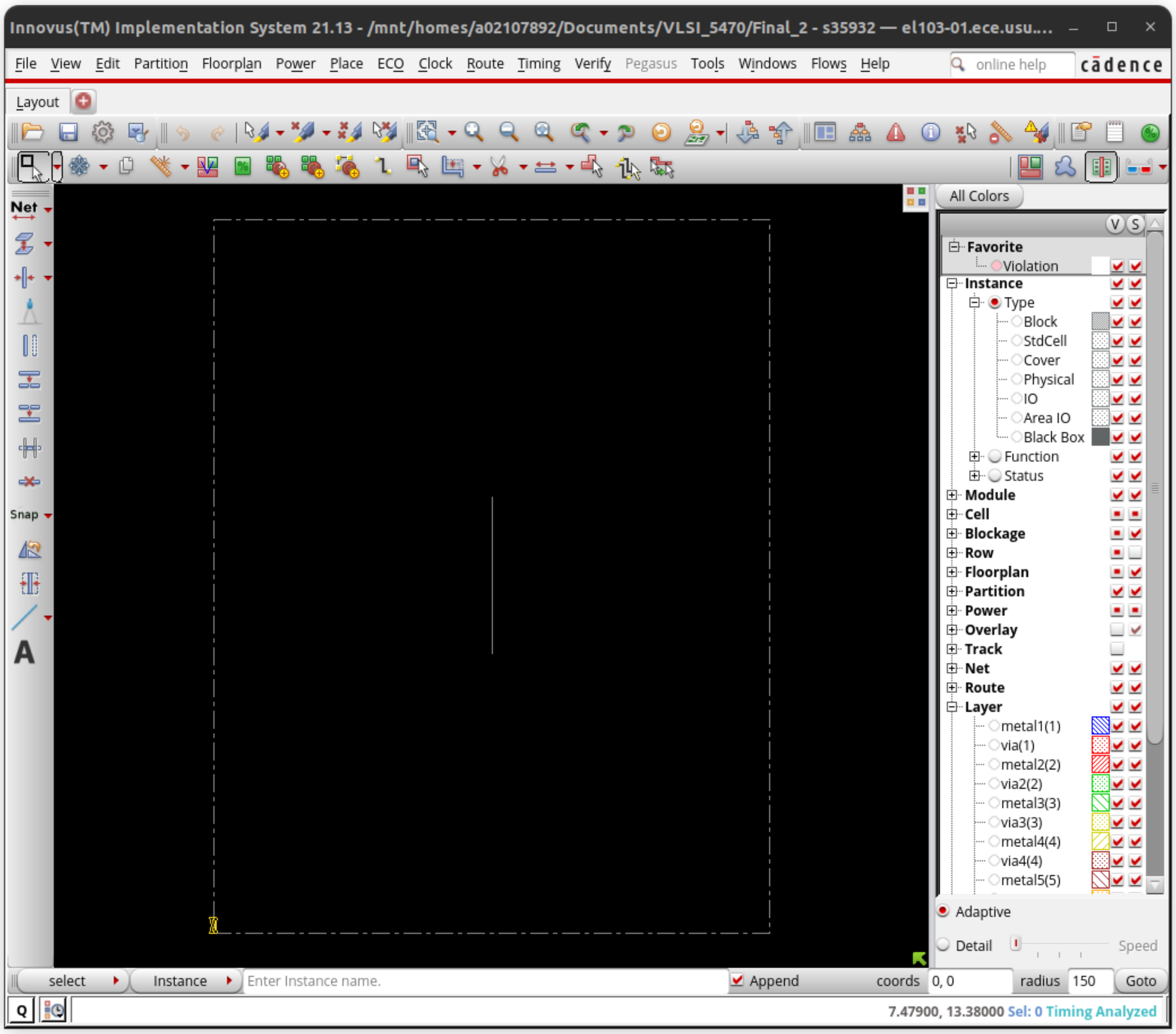
```
****************************************
Report : constraint
        -all_violators
Design : s35932
Version: T-2022.03-SP2
Date   : Wed Apr 19 11:44:18 2023
****************************************

This design has no violated constraints.

1
```

Place and Route Results

Placement and routing didn't work, due to dc_shell and Innovus not agreeing on how 2D Verilog arrays should be written. The only placement/floorplan I got is shown in the figure below.



Verification Results

Because placement and routing didn't work, the verification script was never written.

Appendix: Code

Golden Model

Defs.h

```
#ifndef _DEFS_H_
#define _DEFS_H_

//##########################################
// Defines
//##########################################

#define STACK_LEN 32
#define NUM_REG 12
#define MEM_LEN 32
#define ASSEMBLY_LEN 32

//##########################################
// Struct
//##########################################

typedef struct instruction_t {
   uint16_t opcode;
   uint8_t dest;
   uint8_t src1;
   uint8_t src2;
} instruction_t;

// OPCODE DESTINATION SOURCE1 SOURCE2

//##########################################
// OPCODES
//##########################################

#define NOOP 0b00000000   // NOOP
#define LOAD 0b00000001   // Load from memory
#define LDNM 0b00000010   // Load number
#define STR  0b00000011   // Store into memory
#define ADD  0b00000100   // Add
```

```
#define SUB  0b00000101    // Subtract
#define XOR  0b00000110    // XOR
#define AND  0b00000111    // AND
#define JMP  0b00001000    // Jump
#define JMP0 0b00001001    // Jump if non 0
#define PUSH 0b00001010    // Push to stack
#define POP  0b00001011    // Pop from stack

//##########################################
// Registers
//##########################################

#define REG_0 0b00000000
#define REG_1 0b00000001
#define REG_2 0b00000010
#define REG_3 0b00000011
#define REG_4 0b00000100
#define REG_5 0b00000101
#define REG_6 0b00000110
#define REG_7 0b00000111
#define REG_8 0b00001000
#define REG_9 0b00001001
#define REG_A 0b00001010
#define REG_B 0b00001011
#define REG_R 0b00001100
#define ADDR  0b00001101
#define INSTR 0b00001110
#define STACK 0b00001111

#define NULL_REG 0b00000000

#endif // _DEFS_H_
```

Mips.c

```
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include "defs.h"

```

```
instruction_t fib_assembly[ASSEMBLY_LEN] = {
   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = LOAD, .dest = REG_0,     .src1 = NULL_REG, .src2 = 0x0},
// Read from memory the Fib number to calculate (+1)
   {.opcode = LDNM, .dest = REG_A,     .src1 = NULL_REG, .src2 = 0x1},
// Load a 1 into a register
   {.opcode = LDNM, .dest = REG_B,     .src1 = NULL_REG, .src2 = 0x0},
// Load a 0 into a register
   {.opcode = LDNM, .dest = REG_1,     .src1 = NULL_REG, .src2 = 0x0},
// Load the first two Fibonacci numbers
   {.opcode = LDNM, .dest = REG_2,     .src1 = NULL_REG, .src2 = 0x1},
   //Loop Start
   {.opcode = SUB,  .dest = REG_0,     .src1 = REG_0,    .src2 = REG_A},
// Decrement the counter

   {.opcode = ADD,  .dest = REG_3,     .src1 = REG_1,    .src2 = REG_2},
// Add the two Fib numbers
   {.opcode = ADD,  .dest = REG_1,     .src1 = REG_2,    .src2 = REG_B},
// Shift the new Fib numbers
   {.opcode = ADD,  .dest = REG_2,     .src1 = REG_3,    .src2 = REG_B},

   {.opcode = JMP0, .dest = 0x6,       .src1 = NULL_REG, .src2 = REG_0},
// If the Reg 0 is not 0, jump to the beginning of the loop
   //Loop End
   {.opcode = STR,  .dest = 0x1,       .src1 = NULL_REG, .src2 = REG_3},
// Load the Fib value into the return register
   //FIbonacci end

   //Use other opcodes
   {.opcode = LOAD, .dest = REG_0,     .src1 = NULL_REG, .src2 = 0x2},
// Load the first number
   {.opcode = LOAD, .dest = REG_1,     .src1 = NULL_REG, .src2 = 0x3},
// Load the second number
   {.opcode = AND,  .dest = REG_2,     .src1 = REG_1,    .src2 = REG_0},
// And the numbers
   {.opcode = PUSH, .dest = NULL_REG, .src1 = NULL_REG, .src2 = REG_2},
// Put the And on the stack
   {.opcode = XOR,  .dest = REG_2,     .src1 = REG_1,    .src2 = REG_0},
// XOR The two numbers
```

```
   {.opcode = PUSH, .dest = NULL_REG, .src1 = NULL_REG, .src2 = REG_2},
// Pusht the XOR value
   {.opcode = POP,  .dest = REG_0,    .src1 = NULL_REG, .src2 = NULL_REG},
// Get the xor value from the stack
   {.opcode = STR,  .dest = 0x4,      .src1 = NULL_REG, .src2 = REG_0},
// Put the XOR in the memory
   {.opcode = POP,  .dest = REG_0,    .src1 = NULL_REG, .src2 = NULL_REG},
// Get the AND value
   {.opcode = STR,  .dest = 0x5,      .src1 = NULL_REG, .src2 = REG_0},
// Store the AND value in memory
   {.opcode = JMP,  .dest = 27,       .src1 = NULL_REG, .src2 = NULL_REG},
// JUMP Across the next store
   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = STR,  .dest = 0x6,      .src1 = NULL_REG, .src2 = REG_3},
// This should be skipped, if it isn't memory 6 will be the fib number

   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = NOOP, .dest = NULL_REG, .src1 = NULL_REG, .src2 = NULL_REG
},
   {.opcode = JMP,  .dest = 31,       .src1 = NULL_REG, .src2 = NULL_REG
},//Dead Loop
};
uint32_t fib_memory[MEM_LEN] = {
   0xF,    // Fib number to calculate (+1)
   0x0,    // Fib answer
   0x3,    // Checking other opcodes
   0xF,
};
```

```
#define print_info print_register_info(curr_addr, curr_instr, return_reg,
ALU_1, ALU_2, ALU_OUT, registers, stack)
void print_register_info(uint32_t curr_addr, instruction_t curr_instr,
uint32_t return_reg, uint32_t ALU_1, uint32_t ALU_2, uint32_t ALU_OUT,
uint32_t registers[], uint32_t stack[]) {
   printf("******************************\n");
   printf("Current Address: 0x%X\n", curr_addr);
   printf("Current Instruction:\n");
   printf("\tOPCODE: 0x%X\n", curr_instr.opcode);
   printf("\tDest:   0x%X\n", curr_instr.dest);
   printf("\tSRC1:   0x%X\n", curr_instr.src1);
   printf("\tSRC2:   0x%X\n", curr_instr.src2);
   printf("ALU_1: 0x%X\n", ALU_1);
   printf("ALU_2: 0x%X\n", ALU_2);
   printf("ALU_OUT: 0x%X\n", ALU_OUT);
   printf("Return Reg: 0x%X\n", return_reg);
   for (int i = 0; i < NUM_REG; i++) {
       printf("Reg %2d: 0x%X \tStack[%2d]: 0x%X\n", i, registers[i], i,
stack[i]);
   }
}

void printbin(uint32_t num, int bits) {
   for (int i = bits - 1; i >= 0; i--) {
       printf("%d", (num >> i) & 1);
   }
}

void print_assembly(instruction_t assembly[]) {
   for (int i = 0; i < ASSEMBLY_LEN; i++) {
       printbin(assembly[i].opcode, 8);
       printbin(assembly[i].dest, 8);
       printbin(assembly[i].src1, 8);
       printbin(assembly[i].src2, 8);
       printf("\n");
   }
}

void main() {
   uint32_t *memory = fib_memory;
```

```
    instruction_t *assembly = fib_assembly;
    uint32_t stack[STACK_LEN] = {};
    uint32_t stack_index = 0;
    uint32_t registers[NUM_REG] = {};
    int curr_addr = 0;
    instruction_t curr_instr;
    uint32_t return_reg = 0;
    uint32_t ALU_1 = 0;
    uint32_t ALU_2 = 0;
    uint32_t ALU_OUT = 0;

    while (true) {
        // Instruction Fetch
        curr_instr = assembly[curr_addr++];

        // Info Prep
        switch(curr_instr.opcode) {
            case NOOP:
            case LOAD:
            case LDNM:
            case STR:
            case JMP:
            case JMP0:
            case PUSH:
            case POP:
                ALU_1 = 0;
                ALU_2 = 0;
                break;

            case ADD:
            case SUB:
            case XOR:
            case AND:
                ALU_1 = registers[curr_instr.src1];
                ALU_2 = registers[curr_instr.src2];
                break;

            default:
                printf("Error! INVALID OPCODE info prep\n");
                exit(2);
                break;
        }

        //ALU
        switch(curr_instr.opcode) {
            case NOOP:
            case LOAD:
            case LDNM:
            case STR:
            case JMP:
            case JMP0:
            case PUSH:
            case POP:
                ALU_OUT = 0;
                break;

            case ADD:
                ALU_OUT = ALU_1 + ALU_2;
                break;

            case SUB:
                ALU_OUT = ALU_1 - ALU_2;
                break;

            case XOR:
                ALU_OUT = ALU_1 ^ ALU_2;
                break;

            case AND:
                ALU_OUT = ALU_1 & ALU_2;
                break;

            default:
                printf("Error! INVALID OPCODE ALU\n");
                exit(2);
                break;
        }

        //Memory & Write
        switch(curr_instr.opcode) {
            case NOOP:
                break;

            case LOAD:
                registers[curr_instr.dest] = memory[curr_instr.src2];
                break;

            case LDNM:
                registers[curr_instr.dest] = curr_instr.src2;
                break;

            case STR:
                memory[curr_instr.dest] = registers[curr_instr.src2];
                break;

            case JMP:
                curr_addr = curr_instr.dest;
                break;

            case JMP0:
                if (registers[curr_instr.src2] != 0) {
                    curr_addr = curr_instr.dest;
                }
                break;

            case PUSH:
                stack[stack_index++] = registers[curr_instr.src2];
                break;

            case POP:
                registers[curr_instr.dest] = stack[--stack_index];
                break;

            case ADD:
            case SUB:
            case XOR:
            case AND:
                if (curr_instr.dest == REG_R) { //If setting the return
register
                    return_reg = ALU_OUT;
                }
                else {
                    registers[curr_instr.dest] = ALU_OUT;
                }
                break;

            default:
                printf("Error! INVALID OPCODE Memory\n");
                exit(2);
                break;
        }

        print_info;

        if (curr_addr >= ASSEMBLY_LEN - 1) {
            break;
        }
    }

    printf("***********Done***********\n");
    for (int i = 0; i < MEM_LEN; i++) {
        printf("Mem[%2d] = 0x%X - %d\n", i, memory[i], memory[i]);
    }

    printf("\n\n");
    print_assembly(assembly);
}
```

Mips.v

```
module mips(input clk);
    reg          [31:0] curr_address;
    reg   [0:12] [31:0] registers;
    reg          [31:0] return_reg;
    assign return_reg = registers[12];
    reg          [31:0] curr_instr;
    reg          [31:0] ALU_1;
    reg          [31:0] ALU_2;
    reg          [31:0] ALU_OUT;
    reg  [0:31] [31:0] stack;
    reg         [ 4:0] stack_addr;
    reg  [0:31] [31:0] memory;
    reg  [0:31] [31:0] assembly;

    reg [7:0] opcode;
    reg [7:0] dest;
    reg [7:0] src1;
    reg [7:0] src2;
    assign {opcode, dest, src1, src2} = curr_instr;

    localparam NOOP = 8'h0;
    localparam LOAD = 8'h1;
    localparam LDNM = 8'h2;
    localparam STR  = 8'h3;
    localparam ADD  = 8'h4;
    localparam SUB  = 8'h5;
    localparam XOR  = 8'h6;
    localparam AND  = 8'h7;
    localparam JMP  = 8'h8;
    localparam JMP0 = 8'h9;
    localparam PUSH = 8'hA;
    localparam POP  = 8'hB;

    initial begin
        assembly[ 0] <= 32'b00000000000000000000000000000000;
        assembly[ 1] <= 32'b00000001000000000000000000000000;
        assembly[ 2] <= 32'b00000010000010100000000000000001;
        assembly[ 3] <= 32'b00000010000010110000000000000000;
        assembly[ 4] <= 32'b00000010000000010000000000000000;
        assembly[ 5] <= 32'b00000010000000100000000000000001;
        assembly[ 6] <= 32'b00000101000000000000000000001010;
        assembly[ 7] <= 32'b00000100000000110000000100000010;
        assembly[ 8] <= 32'b00000100000000010000001000001011;
        assembly[ 9] <= 32'b00000100000000100000001100001011;
        assembly[10] <= 32'b00001001000001100000000000000000;
        assembly[11] <= 32'b00000011000000010000000000000011;
        assembly[12] <= 32'b00000001000000000000000000000010;
        assembly[13] <= 32'b00000001000000010000000000000011;
        assembly[14] <= 32'b00000111000000100000000100000000;
```

```
        assembly[15] <= 32'b00001010000000000000000000000010;
        assembly[16] <= 32'b00000110000000100000000100000000;
        assembly[17] <= 32'b00001010000000000000000000000010;
        assembly[18] <= 32'b00001011000000000000000000000000;
        assembly[19] <= 32'b00000011000001000000000000000000;
        assembly[20] <= 32'b00001011000000000000000000000000;
        assembly[21] <= 32'b00000011000001010000000000000000;
        assembly[22] <= 32'b00001000000110110000000000000000;
        assembly[23] <= 32'b00000000000000000000000000000000;
        assembly[24] <= 32'b00000011000001100000000000000011;
        assembly[25] <= 32'b00000000000000000000000000000000;
        assembly[26] <= 32'b00000000000000000000000000000000;
        assembly[27] <= 32'b00000000000000000000000000000000;
        assembly[28] <= 32'b00000000000000000000000000000000;
        assembly[29] <= 32'b00000000000000000000000000000000;
        assembly[30] <= 32'b00000000000000000000000000000000;
        assembly[31] <= 32'b00001000000111110000000000000000;

        memory[ 0] <= 32'hF; // Fib number to calculate (+1)
        memory[ 1] <= 32'h0; // Fib answer
        memory[ 2] <= 32'h3; // Checking other opcodes with store and load
        memory[ 3] <= 32'hF;
        memory[ 4] <= 32'h0;
        memory[ 5] <= 32'h0;
        memory[ 6] <= 32'h0;
        memory[ 7] <= 32'h0;
        memory[ 8] <= 32'h0;
        memory[ 9] <= 32'h0;
        memory[10] <= 32'h0;
        memory[11] <= 32'h0;
        memory[12] <= 32'h0;
        memory[13] <= 32'h0;
        memory[14] <= 32'h0;
        memory[15] <= 32'h0;
        memory[16] <= 32'h0;
        memory[17] <= 32'h0;
        memory[18] <= 32'h0;
        memory[19] <= 32'h0;
        memory[20] <= 32'h0;
        memory[21] <= 32'h0;
        memory[22] <= 32'h0;
        memory[23] <= 32'h0;
        memory[24] <= 32'h0;
        memory[25] <= 32'h0;
        memory[26] <= 32'h0;
        memory[27] <= 32'h0;
        memory[28] <= 32'h0;
        memory[29] <= 32'h0;
        memory[30] <= 32'h0;
        memory[31] <= 32'h0;

        registers[0]  <= 32'h0;
        registers[1]  <= 32'h0;
        registers[2]  <= 32'h0;
        registers[3]  <= 32'h0;
        registers[4]  <= 32'h0;
        registers[5]  <= 32'h0;
        registers[6]  <= 32'h0;
        registers[7]  <= 32'h0;
        registers[8]  <= 32'h0;
        registers[9]  <= 32'h0;
        registers[10] <= 32'h0;
        registers[11] <= 32'h0;
        registers[12] <= 32'h0;
        curr_address  <= 32'h0;

        stack[0]      <= 32'h0;
        stack[1]      <= 32'h0;
        stack[2]      <= 32'h0;
        stack[3]      <= 32'h0;
        stack[4]      <= 32'h0;
        stack[5]      <= 32'h0;
        stack[6]      <= 32'h0;
        stack[7]      <= 32'h0;
        stack[8]      <= 32'h0;
        stack[9]      <= 32'h0;
        stack[10]     <= 32'h0;
        stack[11]     <= 32'h0;
        stack[12]     <= 32'h0;
        stack[13]     <= 32'h0;
        stack[14]      <= 32'h0;
        stack[15]      <= 32'h0;
        stack[16]      <= 32'h0;
        stack[17]      <= 32'h0;
        stack[18]      <= 32'h0;
        stack[19]      <= 32'h0;
        stack[20]      <= 32'h0;
        stack[21]      <= 32'h0;
        stack[22]      <= 32'h0;
        stack[23]      <= 32'h0;
        stack[24]      <= 32'h0;
        stack[25]      <= 32'h0;
        stack[26]      <= 32'h0;
        stack[27]      <= 32'h0;
        stack[28]      <= 32'h0;
        stack[29]      <= 32'h0;
        stack[30]      <= 32'h0;
        stack[31]      <= 32'h0;
        stack_addr     <= 32'h0;
    end

    always@(posedge clk) begin
        curr_instr <= assembly[curr_address];

        // Address
        case (opcode)
            JMP:
                curr_address <= dest;

            JMP0: begin
                if (registers[src2] != 0)
                    curr_address <= dest;
            end

            default:
                curr_address <= curr_address + 1;

        endcase

        case (opcode)
```

```
            NOOP: begin end

            LOAD:
                registers[dest] <= memory[src2];

            LDNM:
                registers[dest] <= src2;

            STR:
                memory[dest] <= registers[src2];

            PUSH: begin
                stack[stack_addr] <= registers[src2];
                stack_addr <= stack_addr + 1;
            end

            POP: begin
                stack_addr <= stack_addr - 1;
                registers[dest] <= stack[stack_addr - 1];
            end

            ADD:
                registers[dest] <= registers[src1] + registers[src2];

            SUB:
                registers[dest] <= registers[src1] - registers[src2];

            XOR:
                registers[dest] <= registers[src1] ^ registers[src2];

            AND:
                registers[dest] <= registers[src1] & registers[src2];

            default: begin end

        endcase
    end

endmodule
```

Cocotb Test Bench

```python
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge, ClockCycles

REG_LEN = 32
NUM_REG = 12

def get_stack(dut):
    stack_str = str(dut.stack.value)
    stack_depth = int(len(stack_str) / REG_LEN)
    stack = []
    for address in range(stack_depth):
        stack.append([])
        for index in range(REG_LEN):
            stack[address].append(stack_str[index + (address * REG_LEN)])

    int_stack = []
    for point in stack:
        int_stack.append(int(''.join(bit for bit in point),2))
    return int_stack

def get_mem(dut):
    mem_str = str(dut.memory.value)
    memory_depth = int(len(mem_str) / REG_LEN)
    memory = []
    for address in range(memory_depth - 1):
        memory.append([])
        for index in range(REG_LEN):
            to_put = mem_str[index + (address * REG_LEN)]
            memory[address].append(to_put)

    int_mem = []
    for mem in memory:
        int_mem.append(int("".join(bit for bit in mem),2))
    return int_mem

def get_registers(dut):
    reg_str = str(dut.registers.value)
    registers = []
    for reg in range(NUM_REG):
        registers.append([])
        for index in range(REG_LEN):
            registers[-1].append(reg_str[index + (reg * REG_LEN)])

    int_reg = []
    for reg in registers:
        int_reg.append(int("".join(bit for bit in reg),2))
    return int_reg

def get_bin_value(input) -> str:
    if 'x' in str(input):
        return str(input)
    else:
        return f"{int(str(input),2):X}"

def print_dut_status(dut):
    print(f"Current Address: 0x{get_bin_value(dut.curr_address.value)}")
    print("Current Instruction")
    print(f"\tOPCODE: 0x{get_bin_value(dut.opcode.value)}")
    print(f"\tDest:   0x{get_bin_value(dut.dest.value)}")
    print(f"\tSRC1:   0x{get_bin_value(dut.src1.value)}")
    print(f"\tSRC2:   0x{get_bin_value(dut.src2.value)}")
    print(f"Return Reg: 0x{get_bin_value(dut.return_reg.value)}")
    registers = get_registers(dut)
    stack = get_stack(dut)
    for index in range(len(registers)):
        print(f"Reg {index:2}: 0x{registers[index]:X} \tStack[{index:2}]:
0x{stack[index]:X}")

@cocotb.test()
async def test_run(dut):
    clock = Clock(dut.clk, 100, units='us')
    cocotb.start_soon(clock.start())

    await ClockCycles(dut.clk, 1)
    while (int(str(dut.curr_address.value),2) < 31):
        await ClockCycles(dut.clk, 1)
        print("*******************************")
        print_dut_status(dut)
    print("**********Done***********")

    memory = get_mem(dut)
    for index in range(len(memory)):
        print(f"Mem[{index:2}] = 0x{memory[index]:X} - {memory[index]}")

    # Got these from the golden model
    assert memory[0] == 0xF
    assert memory[1] == 0x3DB
    assert memory[2] == 0x3
    assert memory[3] == 0xF
    assert memory[4] == 0xC
    assert memory[5] == 0x3
    assert memory[6] == 0x0
```