Owen Huang 1173481

ELEN20006 Project Report

Project Structure

Part 1

- Interface
    - Stage 1: Instantiating and connecting soc module to decimal and hexadecimal modules inside top level module.
    - Stage 2: Creating synchroniser module to synchronise external input signals to soc module, as well implementing the debounce module to appropriately process the synchronised input signals.
    - Stage 3: Testing synchroniser and debounce modules with testbenches and on DE1-SoC.
    - Stage 4: Implementing hexadecimal and decimals and testing in soc module with 23-bit counter.
    - Stage 5: Implementing Enable Generator module using Finite State Machine.
    - Stage 6: Implementing Falling Edge Detector and synchronising remain soc module switches.
- CPU (Central Processing Unit)
    - Stage 7: Implementing and connecting ROM and CPU modules
    - Stage 8: Instantiating the Register File, Instruction Splitter, Controller and Arithmetic Logic Unit (ALU) modules inside the CPU Modules and wiring them together and to the inputs and outputs of the CPU.
    - Stage 9: Implementing unconditional jump and pure move command by modifying ALU, Controller and Instruction pointer modules.
    - Stage 10: Implementing all Move commands in the command group via modifying ALU, Controller, Register File and CPU modules.
    - Stage 11: Implementing Accumulate commands
    - Stage 12: Implementing remaining Jump commands
    - Stage 13: Implementing Atomic Test and Clear
    - Stage 14: Finish connecting remaining inputs and outputs with wires.

Part 2 (Calculator Flow)

Generalised Pseudo-Code Instruction Set (Implementation in Instruction Memory Module)

Initialisation

- stack size = zero; LEDs off; dval = 0 (CPU Registers in an unknown state after reset);

Wait

- Jmp(push): if (Push button released);
- Jmp(pop): if (Pop button released);
- Jmp(add): if (Add button released);
- Jmp(mult): if (Multiplication button released);
- Default: Jmp(Wait);

push

- Mov (stack up);
- Mov (‘DINP to stack[0]);
- Stack[0] on 7-segment display;
- Jmp(overflow): if (Register[4] == 8);
- Both Overflow LEDs off;
- Shift Register[4] to the left;
- Register[4] content to LED pins;
- Default: Jmp(Wait);

overflow

- Stack Overflow LED; Arithmetic Overflow LED off;
- Default: Jmp(Wait);

pop
- Jmp(wait): if (Register[4] == 0);
- Both Overflow LEDs off;
- Shift stack down;
- Shift Register[4] to right;
- Register[4] contents to LED pins;
- Dval = 0;
- Jmp(wait): if (Register[4] == 0);
- Display stack[0] on the display;
- Default: Jmp(Wait);

add
- Arithmetic Overflow LED Off;
- Jmp(wait): if (Register [4] == 0 or 1);
- Stack[0] = Stack[0] + Stack[1];
- Stack[0] on 7-segment display;
- if (overflow): Arithmetic Overflow LED On;
- Stack Overflow LED Off;
- Mov(rest of stack down);
- Shift Register[4] to right;
- Register[4] contents to LED pins;
- Default: Jmp(Wait);

mult
- Jmp(wait): if (stack == empty);
- Arithmetic Overflow LED Off;
- Jmp(normal): if (stack elements => 2);
- Stack[0] = 0;
- 0 on 7-segment display;
- Default: Jmp(Wait);

normal
- Stack[0] = Stack[0] * Stack[1];
- if (overflow): Arithmetic Overflow LED On;
- Stack Overflow LED Off;
- Stack[0] on 7-segment display;
- Shift stack down;
- Shift Register[4] to right;
- Register[4] contents to LED pins;
- Default: Jmp(Wait);

Module Details

- top: Top level entity containing inputs (SW[9:0] and KEY[3:0]) and outputs (LEDR[9:0] and HEX5 to HEX0) needed for the DE1-SoC to produce a fully functional CPU and RPN calculator.
    - soc: Processes inputs from “top” module and outputs the instruction pointer to the hexadecimal display module and current 8-bit number/register to the decimal display module.
        - synchroniser and debounce: Debounce takes input of synchronised signal and only changes outputs when synchronised signal has been constant for the desired amount fo time (30ms).
        - enable_gen: Generates constant output of HIGH if in turbo mode, otherwise output remains low and only HIGH every ENABLE_CNT clock cycles as determined by a counter and resetn.
        - cpu: Contains all necessary sub-designs to process enable_gen, reg_din, and instruction_memory inputs to ouput the instruction pointer and current number to disp_hex and disp_decimal respectively.
            - register_file: Outputs flag register and register 29 plus 30 via inputs from the instruction splitter, controller and alu modules.

- instruction_splitter: Splits a 32-bit instruction from instruction_memory as per the sub-categories defined in the manual.
- controller: Receives command group and command signal from instruction_splitter to determine the specific command to be performed via a case statement containing all command groups and commands.
- alu: Receives specific command to be performed from controller and operand_a and operand_b assignments to execute the appropriate action for the command and output the result to b_data_in in the register_file module.

        - instruction_memory: Location of instruction set that is passed into the cpu module.
    - disp_decimal: Outputs decimal representation of signed 8-bit number (-128 to 127) to 4 Seven segment displays using 4 instances of snum_sseg (on per seven segment display) chained together by the “eno” and “xo” variables.
        - snum_to_sseg: Displays least significant digit on seven segment display and determines if there any digits or negative signs that need to be passed onto the next snum_to_sseg instance.
    - disp_hex: Displays 8-bit (8’h00 - 8’hFF) number on two seven segment displays using one instance of SSeg per seven segment display.
        - sseg_encoder: Displaying hexadecimal digit, negative sign, or blank on a 7-segment display.

Test Benches

Synchroniser

```
// Synchroniser Test Bench

`timescale 1ms/1us

module tb_synchroniser;

	reg clk, in;
	wire out;

	synchroniser sync_test
	(
		.clk(clk) ,
		.in(in),
		.out(out)
	);

	// Testing for Signs of Metastability
	initial begin
		in = 1;
		clk = 0;
		repeat(50) #5 clk = !clk;
		repeat(50) #5 in = !in;
	end

endmodule
```

Debounce

```
// Debounce Test Bench

`timescale 1ms/1us

module tb_debounce;

	reg clk, enable, resetn, sig_i;
	wire sig_o;

	debounce debounce_test
	(
		.clk(clk),
		.enable(enable),
		.resetn(resetn),
		.sig_i(sig_i),
		.sig_o(sig_o)
	);

	// Checking if sig_i Value Appears at sig_o after 30ms as sig_i is Held Constant for 35ms
	initial begin
		clk = 0;
		enable = 1;
		resetn = 0;
		sig_i = 1;
		repeat(50) #5 clk = !clk;
		repeat(7) #35 sig_i = !sig_i;
	end

endmodule
```

Enable Generator

```
// Enable Gen Test Bench

`timescale 1ms/1us

module tb_enable_gen;

	reg clk, reset, mode;
	wire enable_out;

	enable_gen enable_gen_test
	(
		.clk(clk),
		.reset(1'b0),
		.mode(1'b0),
		.enable_out(enable_out)
	);

	// Testing if enable_out Outputs HIGH every ENABLE_CNT Clock Cycles
	initial begin
		clk = 0;
		repeat(100) #5 clk = !clk;
	end

endmodule
```

Top Level

```
// Top Level Module Test Bench

module tb_top();
	reg clk =0;
	reg [9:0] sw = 0;
	reg [3:0] key = 0;
	wire [9:0] ledr;
	wire [6:0] h0,h1,h2,h3,h4,h5;

top DUT
(
	.CLOCK_50(clk),
	.KEY(key),
	.SW(sw),
	.LEDR(ledr),
	.HEX0(h0),
	.HEX1(h1),
	.HEX2(h2),
	.HEX3(h3),
	.HEX4(h4),
	.HEX5(h5)
);

	/* Aim is to Simulate Top Level Module and Compare the Behaviour of Outputs
		to those Observed when testing on the DE1-SoC Board */
	initial forever #10 clk = !clk;

	initial #100000000 $stop;

endmodule
```

Further Notes

<u>Timing Issues</u>

- Synchroniser
  - Reduces metastability as two cascaded flip flops ensure that extra clock cycle passes before result from the first flip flop is recorded by the second flip flop and output, allowing any for any metastable conditions to become stable in that given time.
- Debouncing
  - Needed for reset switch to avoid misinterpreting a single button press as many. Achieved by implementing module requiring the synchronised input to remain constant for 30ms before outputting the input, and otherwise not changing the output if the debouncing period is not met.

<u>Instruction Pointer, Registers and Instructions</u>

- Instruction (Commands to be executed by CPU)
  - Substitution of binary instructions for constants occurs at compile time.
- Register
  - Memory used to store and transfer data plus instructions being used immediately by CPU
  - Register file contains all of the registers that the CPU uses
    - Program variables
    - Special registers
    - Instructions and data
- Instruction Pointer
  - Special-purpose register containing memory address ("points to") of the next instruction to be executed by CPU.
  - Incremented after fetching instruction

"Special" Registers 28, 29 and 30

- Directly connected to the output ports of the register file.
- Register 28 (Data Input Register)
    - Stored data output to the DINP pins of the CPU
        - Holds the latched contents of the din CPU pins
- Register 29 (General Output Register)
    - Stored data output to the GOUT pins of the CPU
        - Gout[5:0] Outputs to the 6 left-most LEDs
        - Gout[7] Determines if Dval is enabled or not
- Register 30 (Data Output Register)
    - Stored data output to the DOUT pins of the CPU
        - Dout[7:0] Outputs to disp_decimal

FLAG register (Register 31)

- Special status register containing current state of CPU

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
		Shift	Overflow	Sample	Btn 2	Btn 1	Btn 0
		SHFT	OFLW	SMPL			

2's Complement Arithmetic in Verilog

- Number range represented using binary, positive integers use normal binary convention with leading 0 bit and negative number have a leading 1 with digits inverted.
    - Subtracting one number from another same as making one number negative and adding
- Implementation in Verilog involves an assign statement where original number in binary form is inverted using the bitwise not operator. 1 is then added to the result to produce the leading digit 1 to indicate a negative number, thus applying 2s complement.

disp_decimal Module

- 8-bit number (-128 to 127) and an enable signal inputs to output to 4 seven segment displays
    - Enable low => all displays are off.
    - Enable high => display 8-bit signed number as a decimal with leading zeros.
    - -8'd20 gives [] [-] [2] [0].
- disp_decimal uses four instantiations of snum_to_sseg, with each instance first displaying the current least significant digit before pass on the rest of the digits to the next instance, if there are any. If not, then zero will be displayed for all remaining digits on the seven-segment display. The eno variable controls the passing of digits to following instances and is set to the value of input enable for the first instantiation of snum_to_sseg.

Machine Code

Bits 31-29	Bit 28-26	Bit 25-17	Bit 16-8	Bit 7-0
Command Group	Command	Argument 1	Argument 2	Address

- First bit of Argument 1 and 2 indicates if 8 following bits represent number (0) or register (1)
- Instruction Set Constants

```
// Command Groups
`define NOP 3'b000 // No Operation
`define JMP 3'b001 // Jump
`define ATC 3'b010 // Atomic Test and Clear
`define MOV 3'b011 // Move
`define ACC 3'b100 // Accumulate

// Jump Commands
`define UNC 3'b000 // Unconditional Jump
`define EQ  3'b010 // Jump on Equality
`define ULT 3'b100 // Jump On Unsigned Less Than
`define SLT 3'b101 // Jump On Signed Less Than
`define ULE 3'b110 // Jump On Unsigned Less Than Or Equal To
`define SLE 3'b111 // Jump On Signed Less Than Or Equal To

// Move Commands
`define PUR 3'b000 // Pure Move
`define SHL 3'b001 // Move and Shift Left
`define SHR 3'b010 // Move and Shift Right

// Accumulate Commands
`define UAD 3'b000 // Unsigned Addition
`define SAD 3'b001 // Signed Addition
`define UMT 3'b010 // Unsigned Multiplication
`define SMT 3'b011 // Signed Multiplication
`define AND 3'b100 // Bitwise AND
`define OR  3'b101 // Bitwise OR
`define XOR 3'b110 // Bitwise XOR

// Special Registers
`define DINP 8'd28 // Data Input Register Address
`define GOUT 8'd29 // General Purpose Register Address
`define DOUT 8'd30 // Data Ouput Register Address
`define FLAG 8'd31 // Flag Register Address
```