Andy Jeong, Brenda So, Gordon Macshane

ECE 151A Spring 2016

22 February 2016

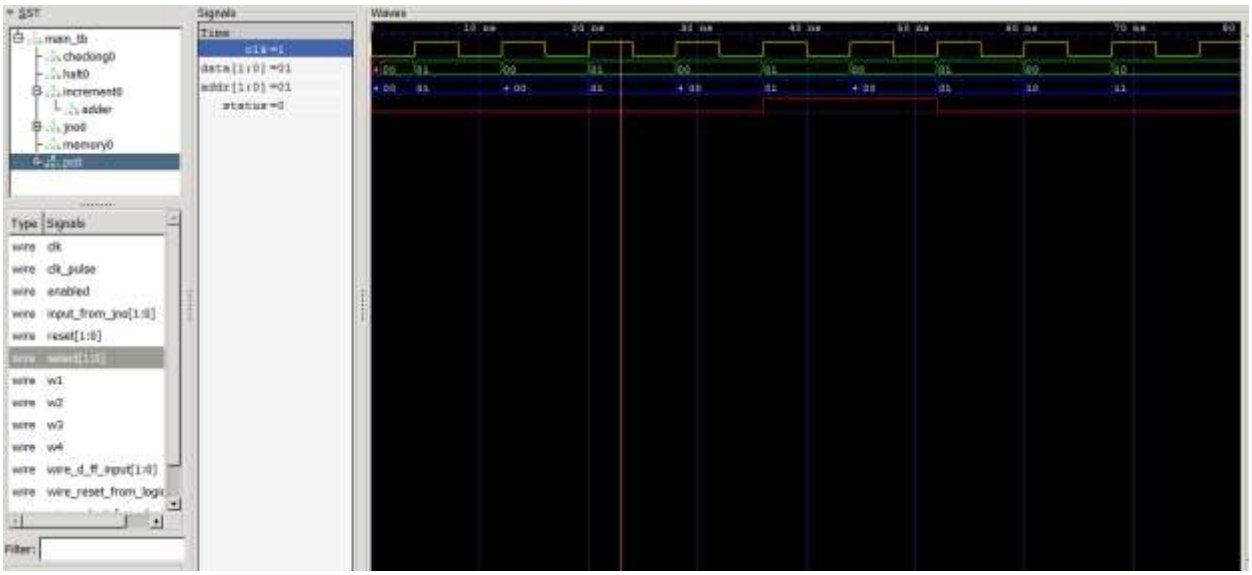
Project 1: Paper Processor

Abstract

This project aims to construct a circuit on Verilog Hardware Description Language whereby the paper processor (https://sites.google.com/site/kotukotuzimiti/Paper_Processor) is implemented. The Verilog code takes in the modules (.v) – *checking, dff_reset_negedge, dff_reset_posedge, halt, increment, jno, programcounter, fulladder, memory* – which take the inputs and outputs the data bits according to the primitive logic gates event-driven cases (always block) and time delays (#). The *instruction.bin* file takes in the instruction codes, which are picked out based on the logic, and determines the data output.

Diagram (refer to *waveform.png*)**)**

Waveform 1: Paper Processor implementation for time duration = 69 (ns)



Sample Output *(refer to sample_output.txt)*

VCD info: dumpfile pp.vcd opened for output.

time = 0 PC = xx instruct = xx out = 00 stat = 0

time = 1 PC = 00 instruct = 00 out = 00 stat = 0

time = 5 PC = 01 instruct = 01 out = 01 stat = 0

time = 13 PC = 10 instruct = 00 out = 01 stat = 0

time = 14 PC = 00 instruct = 00 out = 01 stat = 0

time = 21 PC = 01 instruct = 01 out = 01 stat = 0

time = 22 PC = 01 instruct = 01 out = 10 stat = 0

time = 29 PC = 10 instruct = 00 out = 10 stat = 0

time = 30 PC = 00 instruct = 00 out = 10 stat = 0

time = 37 PC = 01 instruct = 01 out = 10 stat = 0

time = 38 PC = 01 instruct = 01 out = 11 stat = 0

time = 45 PC = 10 instruct = 00 out = 11 stat = 0

time = 46 PC = 00 instruct = 00 out = 11 stat = 0

time = 53 PC = 01 instruct = 01 out = 11 stat = 0

time = 54 PC = 01 instruct = 01 out = 00 stat = 1

time = 61 PC = 10 instruct = 00 out = 00 stat = 1

time = 69 PC = 11 instruct = 10 out = 00 stat = 1

Compiling and Running the Code (refer to *Makefile*)

***need iverilog (Icarus Verilog) and GTKwave installed**

To compile, run and show waveform using gtkwave simulator:

make –f Makefile

What is under *Makefile*?

```
SOURCE = main_tb.v
WAVE = gtkwave
all:
        iverilog -o pp.vvp $(SOURCE)
        vvp pp.vvp
        $(WAVE) pp.vcd
clean:
        rm -rf *.vvp *.vcd
```

Optional:

To compile separately on terminal:

iverilog -o <objectname> main_tb.v

To run the compiled object file:

vvp <objectname>

To change the instruction:

change the bits in instruct.bin

Paper Processor Design

The 2-bit Paper Processor Instructions

inc	//increments the address, register
jno $00_{(2)}$	//jump if not overflow
hlt	//halt
nop	//no operation

Verilog Code

Main Testbench Code (refer to main_tb.v)

```
//
// School: The Cooper Union
// Course: ECE151A Spring 2016
// Assignment: Paper Processor
// Group members: Andy Jeong, Brenda So, Gordon Macshane
// Date: 2/22/2016
//
//------------------Instructions----------------------
//       Instructions Guide:
//                INC: 00(2)
//                JNO: 01(2)
//                HLT: 10(2)
//
//       Instructions for the 2-bit paper processor:
//                IS1:     INC
//                IS2,3:  JNO 00(2)
//                IS4:     HLT
//
`timescale 1ns/1ns

`include "memory.v"
`include "increment.v"
`include "programcounter.v"
`include "halt.v"
`include "jno.v"
`include "checking.v"
`include "dff_reset_negedge.v"
`include "dff_reset_posedge.v"

module main_tb();

reg reg_status;                                          // status register
reg reg_clock;                                                   // clock register
reg [1:0] reg_reset;                                     //used to initialize the
program counter
reg [1:0] reg_output;                                            // output
registers

wire wire_clock_from_jno;                 //Pulse from JNO, imitating monostable timer
wire [1:0] wire_adder_to_output;                  // INC logic :connect adder to reg
out
wire wire_adder_to_status;                                       // ING logic:
ddding to status
wire [1:0] wire_instruction_from_memory;                         // output
instruction from ram
wire [1:0] wire_instruction_checked;                             // actual
instruction after checking
```

```
wire wire_monostable_clock;                          // monostable clock induced by inc
logic
wire wire_clock_from_halt;
        // clock induced by halt
wire wire_monostable_clock_status_0;
wire wire_enabled;
wire wire_enabled_status;
wire [1:0] wire_program_counter;                                    // Counter:
program counter output

initial begin

        //print out on console
        $monitor ("PC = %b instr = %b reg = %b status = %b",  wire_program_counter,
wire_instruction_from_memory, reg_output, reg_status);

        reg_clock = 0;                              // initialize clock to be 0
        reg_status = 0;                    // initialize status to be 0
        reg_output= 2'b00;                          // initialize value register
        reg_reset= 2'b00;                           // Resettting logic
        #1 reg_reset = 2'b11;              //Resetting program counter
        #1 reg_reset= 2'b00;
        #80 $finish;                                //simulation runs for 80 seconds
end

initial begin
        $dumpfile ("pp.vcd");    //waveform file output in .vcd format
    $dumpvars;
end

always @(posedge wire_monostable_clock_status_0)
begin
        reg_output[0] <= wire_adder_to_output[0];
        reg_output[1] <= wire_adder_to_output[1];
        reg_status <= wire_adder_to_status;
end

always
        #4 reg_clock = ~reg_clock; // Generate clock

memory memory0(wire_program_counter,wire_instruction_from_memory);


programcounter pc0(wire_program_counter, wire_clock_from_jno, reg_reset,
wire_instruction_from_memory, wire_clock_from_halt, wire_enabled_status);

and a1(wire_monostable_clock_status_0, wire_monostable_clock, !reg_status);

checking checking0(wire_instruction_checked, wire_enabled,
wire_instruction_from_memory);
```

```
increment increment0(wire_adder_to_output,
wire_adder_to_status,wire_monostable_clock,wire_clock_from_halt,
reg_output,wire_instruction_checked);

jno jno0(wire_enabled, wire_enabled_status, wire_clock_from_jno, reg_status,
wire_instruction_checked, wire_clock_from_halt);

halt halt0(wire_clock_from_halt, reg_clock,
wire_instruction_checked);

endmodule
```

checking.v

```
//////////////////////////////////////////////////////////////////////////////////
// Module Name: checking.v
// Description: checks if either of two inputs is HIGH at a certain time
//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

module checking(instruct_checked, enabled, instruct_from_jno);

//-------------Input Ports------------------------------
input [1:0] instruct_from_jno;
input enabled;

//-------------Output Ports-----------------------------
output [1:0] instruct_checked;

//input and output can be declared again as wires to pass into modules
wire [1:0] instruct;
wire enabled;
wire [1:0] instruct_checked;

or o1(instruct_checked[1], enabled, instruct_from_jno[1]);
or o2(instruct_checked[0], enabled, instruct_from_jno[0]);

endmodule
```

dff_reset_negedge.v

```
//////////////////////////////////////////////////////////////////////////////////
// Module Name: dff_reset_negedge.v
// Description: D flip flop that sets and resets at negedge
//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

module dff_reset_negedge (clk, s, r, d, output_q, output_not_q);
        input clk, s, r, d;
        output output_q, output_not_q;
        reg output_q, output_not_q;
        always @(negedge r) begin
                output_q = 1'b0;
                output_not_q = 1'b1;
                end
        always @(negedge s) begin
                output_q = 1'b1;
                output_not_q = 1'b0;
                end
        always @(posedge clk) begin
                output_q = d;
                output_not_q = ~d;
                end

endmodule
```

dff_reset_posedge.v

```
////////////////////////////////////////////////////////////////////////////////
// Module Name: dff_reset_posedge.v
// Description: D flip flop that sets and resets at posedge
////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

module dff_reset_posedge (clk, s, r, d, output_q, output_not_q);
        input clk, s, r, d;
        output output_q, output_not_q;
        reg output_q, output_not_q;
        always @(posedge r) begin
                output_q = 1'b0;
                output_not_q = 1'b1;
                end
        always @(posedge s) begin
                output_q = 1'b1;
                output_not_q = 1'b0;
                end
        always @(posedge clk) begin
                output_q = d;
                output_not_q = ~d;
                end

endmodule
```

halt.v

```
//////////////////////////////////////////////////////////////////////////////////
// Module Name: halt.v
// Description: takes clock input and according to the current instruction it
// determines if the clock should go on or not
//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

module halt(pulse, clk, instruction_from_jno);

//-------------Input Ports------------------------------
input [1:0] instruction_from_jno;              //input a 2 bit register from JNO
input clk;

//-------------Output Ports-----------------------------
output pulse;

wire [1:0] instruct;
wire clk;
wire pulses;
wire [1:0] w;

//----------------------Instructions--------------------

and a1(w[1], instruction_from_jno[1], !instruction_from_jno[0]);
not n1(w[0], w[1]);
and(pulse, clk, w[0]);

endmodule
```

increment.v

```
//////////////////////////////////////////////////////////////////////////////////
// Module Name: increment.v
// Description: increments 2-bit value from previous
//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

`include "fulladder.v"

module increment(adder_to_output, adder_to_status, output_monostable, output_pulse,
out, instruct_checked);

//-------------Input Ports------------------------------
input [1:0] instruct_checked;     //input a 2 bit register from JNO
input [1:0] out;
input output_pulse;

//-------------Output Ports-----------------------------
inout [1:0] adder_to_output;
output adder_to_status;
output output_monostable;

//input and output can be declared again as wires to pass into modules
wire [1:0] instructA;
wire [1:0] out;
wire output_pulse;
wire [1:0] adder_to_output;
wire status;
reg output_monostable;

//-------------Intermediate Wires-----------------------
wire w1;

//------------------instructions-----------------------

and (w1, output_pulse, !instruct_checked[1], !instruct_checked[0]);

initial begin
output_monostable = 0;
end

always @(posedge w1)
begin
        output_monostable = 1;
        #2
        output_monostable = 0;
end
```

```
fulladder adder(adder_to_status, adder_to_output, out);

endmodule
```

jno.v

```
////////////////////////////////////////////////////////////////////////////////
// Module Name: jno.v
// Description: from the current instructions, outputs ENABLE for checking and counter
////////////////////////////////////////////////////////////////////////////////
`timescale 1ns/1ns

module jno(enable, enable_status, openpulse, sta, instruct, pulses);

//-------------Input Ports------------------------------

input [1:0] instruct;    //input a 2 bit register from JNO
input pulses;
input sta;

//-------------Output Ports-----------------------------

output enable;
output enable_status;
output openpulse;

//-------------Input ports Data Type--------------------
// By rule all the input ports should be wires
wire [1:0] instruct;
wire pulses;
wire sta;

//-------------Output Ports Data Type-------------------
// Output port can be a storage element (reg) or a wire
wire enable;
wire enable_status;

//-------------Intermediate Wires-----------------------
reg s;
reg r;
wire not_enable, not_enable_status;
wire w1, w2;
reg mem;
reg pulser;
wire openpulse;
reg openpulser;

//------------------Instructions-----------------------
```

```
initial begin
s = 0;
r = 0;
pulser = 0;
openpulser = 0;
end

and a1(w1, !instruct[1], instruct[0]);
and a2(w2, !sta, !instruct[1], instruct[0]);
and a3 (openpulse, !sta, openpulser);

always @(posedge w1)
begin
        #1
        pulser = 1;
        #8
        openpulser = 1;
        #4
        openpulser = 0;
        #1
        pulser = 0;
        #1
        pulser = 1;
        #1
        pulser = 0;
end
dff_reset_negedge dff3(pulser, s, r, w1, enable, not_enable); //enabling the checking
dff_reset_negedge dff4(pulser, s, r, w2, enable_status, not_enable_status); //enabling
for counter

endmodule
```

programcounter.v

```
//////////////////////////////////////////////////////////////////////////////////
// Module Name: programcounter.v
// Description: counter
//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

module programcounter(select, clk_pulse, reset, input_from_jno, clk, enabled);

//-------------Input Ports------------------------------

input [1:0] input_from_jno;       //input a 2 bit register from JNO
input clk;                        //input clock signal
input enabled;            //input enabling JNO register addresses
input [1:0] reset;                //for resetting
input clk_pulse;

//-------------Output Ports------------------------------

output [1:0] select;

//-------------Input ports Data Type-------------------
// By rule all the input ports should be wires
wire [1:0] input_from_jno;
wire clk;
wire enabled;
wire [1:0] R;
wire clk_pulse;

//-------------Output Ports Data Type-------------------
// Output port can be a storage element (reg) or a wire
wire [1:0] select;

//-------------Intermediate Wires-----------------------

wire [1:0] wire_select_from_logic;
wire w1, w2,w3,w4, w5, w6;
wire [1:0] wire_d_ff_input;
wire [1:0] wire_set_from_logic;
wire [1:0] wire_reset_from_logic;

//------------------Instructions-----------------------
and a1(wire_set_from_logic[0], clk_pulse, enabled, input_from_jno[0]);
and a2(wire_set_from_logic[1], clk_pulse, enabled, input_from_jno[1]);
and a3(w3, clk_pulse, enabled, !input_from_jno[0]);
and a4(w4, clk_pulse, enabled, !input_from_jno[1]);
or o1(wire_reset_from_logic[0], w3, reset[0]); //R[0] is 0, w1 goes from 0 to 1
or o2(wire_reset_from_logic[1], w4, reset[1]); //
```

```
dff_reset_posedge dff0(clk, wire_set_from_logic[0], wire_reset_from_logic[0],
wire_d_ff_input[0], select[0], wire_select_from_logic[0]);
dff_reset_posedge dff1(clk, wire_set_from_logic[1], wire_reset_from_logic[1],
wire_d_ff_input[1], select[1], wire_select_from_logic[1]);

and a5(w1, wire_select_from_logic[0], select[1]);
and a6(w2, wire_select_from_logic[1], select[0]);
or o3(wire_d_ff_input[1], w1, w2);
and a7(wire_d_ff_input[0], wire_select_from_logic[0],1);

endmodule
```

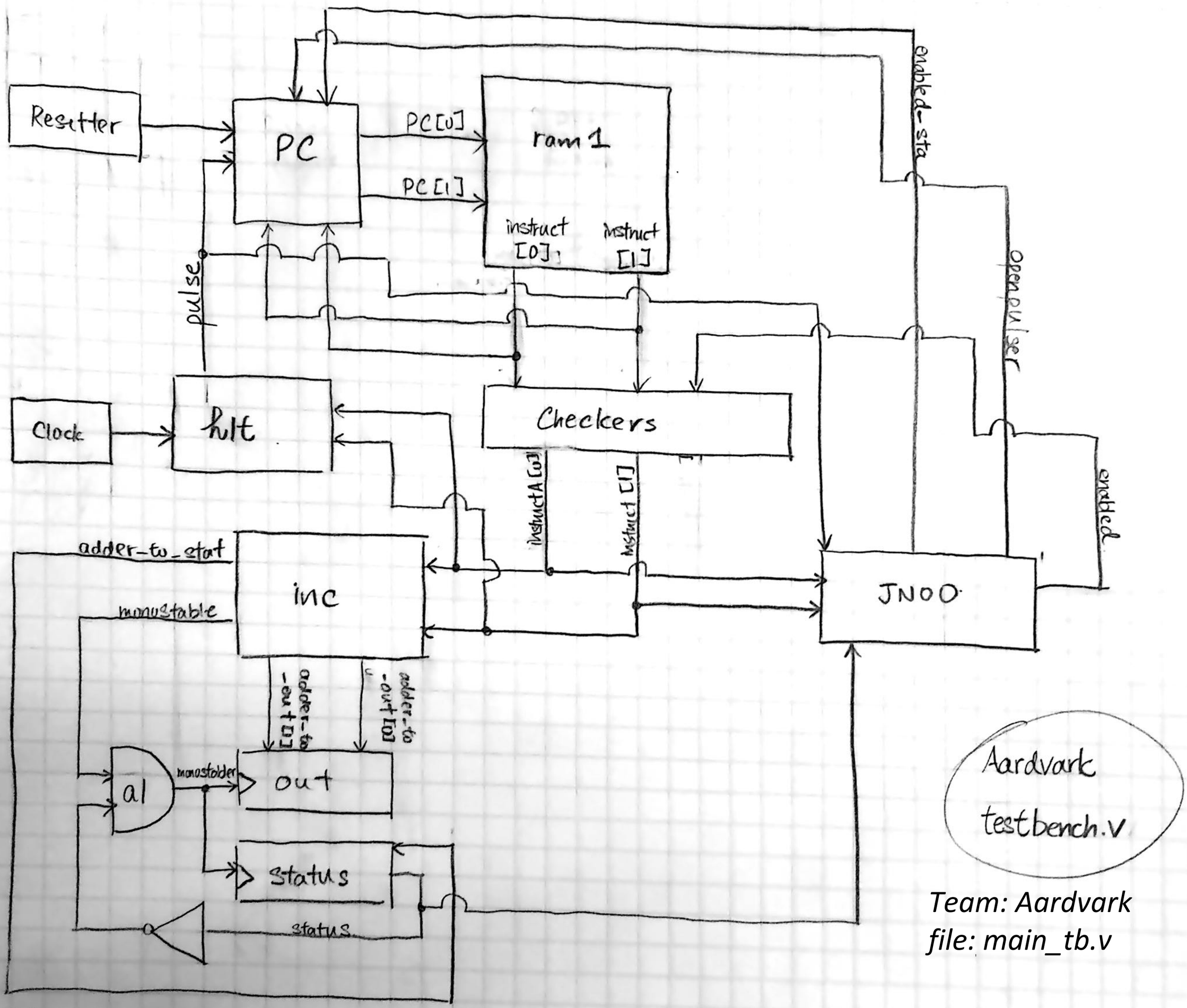
fulladder.v

```
////////////////////////////////////////////////////////////////////////////////
// Module Name: fulladder.v
// Description: Full Adder where 1 is always added to the previous number
////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

module fulladder(status, sum, a);

//-------------Input Ports------------------------------

inout [1:0] a; //input a 2 bit register

//-------------Output Ports-----------------------------

output [1:0] sum; //output 2 bit registers
output status; //see whether registers overflown

//-------------Input ports Data Type--------------------
// By rule all the input ports should be wires
wire [1:0] a;
wire [1:0] b;

//-------------Output Ports Data Type-------------------
// Output port can be a storage element (reg) or a wire
wire [1:0] sum;
wire status;

//-------------Intermediate Wires-----------------------
wire w0, w1, w2, w3;

//------------------Instructions-----------------------
```

```
//set LSB of input to sum[0]
xor u0(sum[0],a[0], 1'b1);

//set MSB of input to sum[1]
and u1(w0, a[0], 1'b1);
xor u2(w1, a[1], 1'b0);
and u3(w2, a[1], 1'b0);
and u4(w3, w0, w1);
xor u5(sum[1], w0, w1);

//set carry out to be status
or u6(status,w2, w3);

endmodule
```

memory.v

```
////////////////////////////////////////////////////////////////////////////////
// Module Name: memory.v
// Description: loads data from instruction binary file
////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ns

module memory(addr, data);

parameter Instructions = "./instruction.bin";

//-------------Input Ports------------------------------
input [1:0] addr;

//-------------Output Ports-----------------------------
output [1:0] data;

reg [1:0] ram_reg [0:3];
wire [1:0] addr;
wire [1:0] data;

//-------------load instructions---------------
//2 bit wide, 4 bit deep register memory
initial begin
        $readmemb(Instructions, ram_reg);
end

assign data = ram_reg[addr];

endmodule
```



Aardvark

testbench.v

Team: Aardvark

file: main_tb.v