Single purpose processor design for GCD computation.

Anisha Sharma
School of Computing and Electrical Engineering,IIT Mandi,
Himachal Pradesh, India.
b20029@students.iitmandi.ac.in

I. Abstract

The Aim of the Experiment is to design a Custom Single purpose processor for GCD computation by first creating algorithm diagram and then convert algorithm to “complex” state machine known as FSMD and then finally creating the Datapath and Controller . .

Index Terms = Embedded , Datapath , Controller, GCD , FSMD .

II. Introduction and Highlights of Contribution

A single-purpose processor is a digital system intended to solve a specific computation task. While a manufacturer builds a standard single-purpose processor for use in a variety of applications, we build a custom single purpose processor to execute a specific task within our embedded system. First, performance may be fast, due to fewer clock cycles resulting from a customized datapath, and due to shorter clock cycles resulting from simpler functional units, less multiplexors, or simpler control logic. Second, size may be small, due to a simpler datapath and no program memory. In fact, the processor may be faster and smaller than a standard one implementing the same functionality, since we can optimize the implementation for our particular task. It has a Controller and a Dtapath .

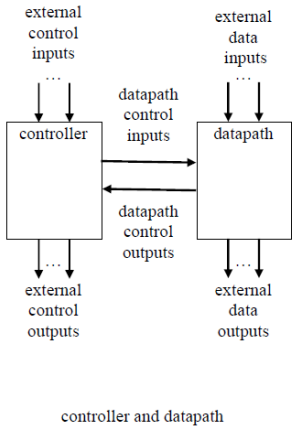

Fig. 1: Controller and Datapath

Here , we are computing a Greatest Common Divisor . It have xi and yi as inputs and di as outputs . The output should represent the GCD of the inputs. Thus, if the inputs are 12 and 8, the output should be 4. If the inputs are 13 and 5, the output should be 1.

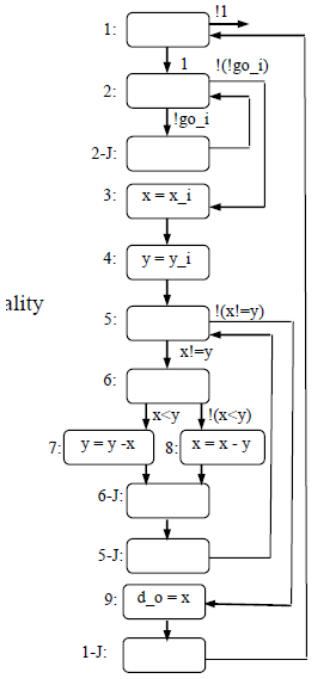

Fig. 2: State Diagram

We are now well on our way to designing a custom single-purpose processor that executes the GCD program. Our next step is to divide the functionality into a datapath part and a controller part, as shown in Figure 4.4. The datapath part should consist of an interconnection of combinational and sequential components. The controller part should consist of a basic state diagram, i.e., one containing only boolean actions and conditions. We construct the datapath through a four-step process:-

1. First, we create a register for any declared variable. In the example, these are x and y.
2. Second, we create a functional unit for each arithmetic

operation in the state diagram. In the example, there are two subtractions, one comparison for less than, and one comparison for inequality, yielding two subtractors and two comparators

3. Third, we connect the ports, registers and functional units. For each write to a variable in the state diagram, we draw a connection from the write's source (an input port, a functional unit, or another register) to the variable's register.

4. Finally, we create a unique identifier for each control input and output of the datapath components.

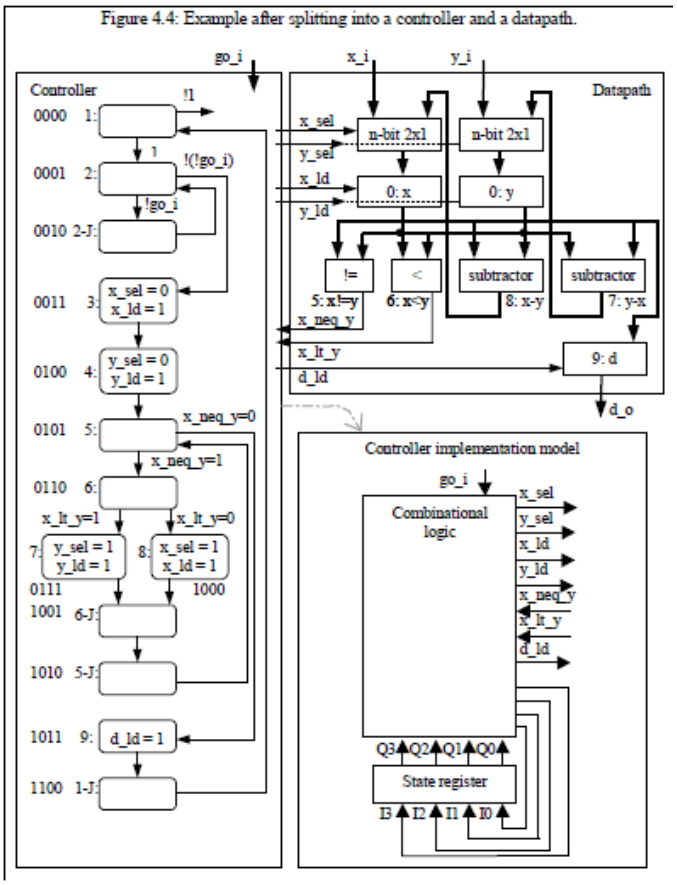

Fig. 3: After splitting into a controller and a datapath.

Now that we have a complete datapath, we can build a state diagram for our controller. The state diagram has the same structure as the complex state diagram. However, we replace complex actions and conditions by boolean ones, making use of our datapath. We replace every variable write by actions that set the select signals of the multiplexor in front of the variable's register's such that the write's source passes through, and we assert the load signal of that register. We replace every logical operation in a condition by the corresponding functional unit control output. We can then complete the controller design by implementing the state diagram using our sequential design technique .

III. Implemented Hardware Architecture and Discription

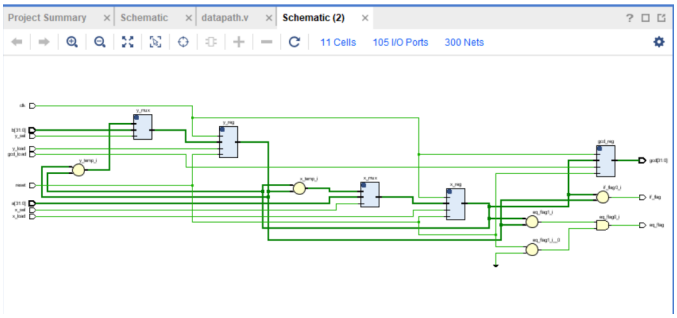

Fig. 4: Schematic of the Datapath

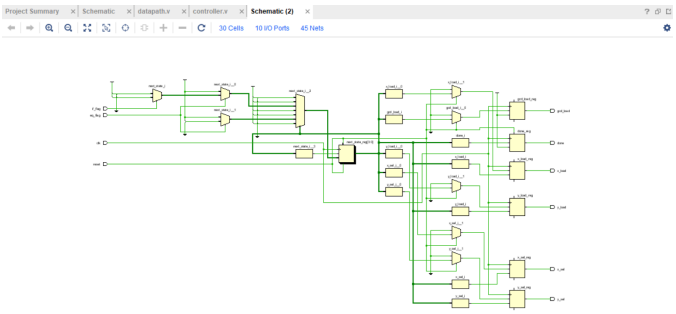

Fig. 5: Schematic of the Controller

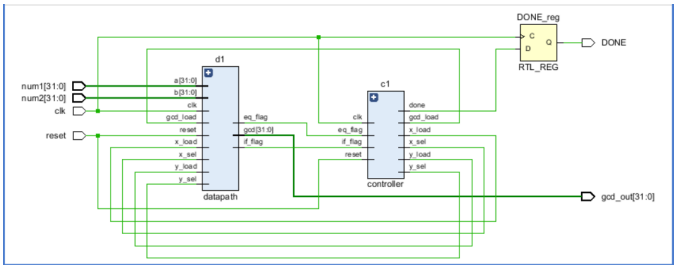

Fig. 6: Schematic of the GCD

IV. Design Flow Adopted in the Experiment:-

A. *Verilog Code of the Components of GCD :-*

Project Summary | gcd.v | datapath.v | mux.v | register.v | controller.v | gcd_tb.v

C:/Users/Admin/project_4/project_4.srcs/sources_1/new/datapath.v

```
`timescale 1ns / 1ps

module datapath(
    input clk, reset, x_sel, y_sel, x_load, y_load, gcd_load,
    input [31:0]a, b,
    output reg eq_flag, if_flag,
    output [31:0]gcd
    );
    // wire [31:0] gcd;
     //reg eq_flag, if_flag;
     wire [31:0] x_regout;
     wire [31:0] y_regout;
     wire [31:0] xmux_out, ymux_out;
     wire [31:0] x_temp, y_temp;

    mux x_mux(                                      //mux for X
    .sel(x_sel),
    .in0(x_temp), .in1(a), .mux_out(xmux_out)
    );

    mux y_mux(                                        //  mux for Y
    .sel(y_sel),
    .in0(y_temp), .in1(b), .mux_out(ymux_out)
    );

    register x_reg(
    .clk(clk),
    .reset(reset),
    .load(x_load),
    .data(xmux_out), .out(x_regout)
    );
```

```
    );
     register gcd_reg(
    .clk(clk),
    .reset(reset),
    .load(gcd_load),
    .data(x_regout), .out(gcd)
    );
     assign x_temp = x_regout - y_regout;
     assign y_temp = y_regout - x_regout;
     always@(x_regout or y_regout)
     begin
        if((x_regout == y_regout) & (reset == 1'b0))
            begin
            $display("eq_flag set");
            eq_flag <= 1'b1;
            end
        else
            begin
            $display("eq_flag not set");
            eq_flag <= 1'b0;
            end
     end
     always@(x_regout or y_regout)
     begin
        if(x_regout < y_regout) // Including a reset condition check might be needed
        begin
        $display("if_flag set");
            if_flag <= 1'b1;
            end
        else
            if_flag <= 1'b0;
```

Fig. 7: Verilog Code of the Datapath of the GCD

C:/Users/Admin/project_4/project_4.srcs/sources_1/new/controller.v

```
`timescale 1ns / 1ps

module controller(
    input clk,
    input reset,
    input eq_flag,
    input if_flag,
    output reg done,
    output reg x_load,
    output reg y_load,
    output reg x_sel,
    output reg y_sel,
    output reg gcd_load
    );

     reg [3:0]present_state, next_state;
     parameter start = 1,
                in = 2,
                test = 3,
                test2 = 4,
                update1 = 5,
                update2 = 6,
                stop = 7;
     always@(posedge clk or posedge reset)
     begin
     if(reset)
        begin
        present_state <= start;
        next_state <= start;
        done <= 0;
        $display("during reset");
```

```
        $display("during reset");
        end
     else
        case(next_state)
         start: begin
                    next_state <= in;
                    $display("next state is test");
                    end
         test: begin
                    x_sel <= 1'b0;
                    y_sel <= 1'b0;
                    x_load <= 1'b0;
                    y_load <= 1'b0;
                    //if( eq_flag == 1'b1)
                    //  next_state <= done;
                    //else
                    next_state <= test2;
                   end
          test2: begin
                    x_sel <= 1'b0;
                    y_sel <= 1'b0;
                    x_load <= 1'b0;
                    y_load <= 1'b0;
                    if( eq_flag == 1'b1)
                        next_state <= stop;
                    else if( if_flag == 1'b1)
                        next_state <= update1;
                    else
                        next_state <= update2;

                    end
```

Fig. 8: Verilog Code of the Controller

```
                    end
          update2: begin        // y = y - x;
                        x_load <= 1'b1;
                        y_load <= 1'b0;
                        x_sel <= 1'b0;
                        next_state <= test;
                        $display("update1");
                       end
          update1: begin
                        y_load <= 1'b1;
                        x_load <= 1'b0;
                        y_sel <= 1'b0;
                      next_state <= test;
                       end
          in: begin
                x_sel <= 1'b1; y_sel <= 1'b1;
                x_load <= 1'b1; y_load <= 1'b1;
                if( eq_flag == 1'b1)
                    next_state <= stop;
                else
                    next_state <= test;
              end
          stop: begin
              gcd_load <= 1'b1;
              done <= 1;
              next_state <= stop;
              end
        endcase
    end
/*
```

```
`timescale 1ns / 1ps

module mux(
    input sel,
    input [31:0] in0, in1,
     output [31:0] mux_out
    );

assign mux_out = (sel==0) ? in0 : in1;

endmodule
```

Fig. 9: Verilog Code of the 2x1 Multiplexar

```
`timescale 1ns / 1ps

module register(
    input clk,
    input reset,
    input load,
    input [31:0] data,
    output reg [31:0] out
    );
    //reg [31:0] out;
    always@(posedge clk)
    begin
    if(reset == 1'b1) begin
        out <= 0;
    end else if(load == 1'b1) begin
        out <= data;
    end else begin
        out <= out;
    end
    end

endmodule
```

Fig. 10: Verilog Code of the Register

C:/Users/Admin/project_4/project_4.srcs/sources_1/new/gcd.v

```
`timescale 1ns / 1ps

module gcd(
    input [31:0] num1, num2,
    output [31:0] gcd_out,
    output reg DONE,
    input clk, reset
    );

    wire xsel, ysel, xload, yload, gcdload, eqflag, ifflag;
    wire done;

     datapath d1( .clk(clk), .reset(reset), .x_sel(xsel), .y_sel(ysel), .x_load(xload),
     .y_load(yload), .gcd_load(gcdload),
     .a(num1), .b(num2), .eq_flag(eqflag), .if_flag(ifflag),
    .gcd(gcd_out));

     controller c1( .clk(clk),
    .reset(reset),
    .eq_flag(eqflag),
    .if_flag(ifflag),
    .done(done),
    .x_load(xload),
    .y_load(yload),
    .x_sel(xsel),
    .y_sel(ysel),
    .gcd_load(gcdload));

    always @(posedge clk) begin
        DONE <= done;
    end
```

Fig. 11: Verilog Code of the GCD which includes both datapath and Controller

```
module gcd_tb;
    reg [31:0] num1;
    reg [31:0] num2;
    reg clk;
    reg reset;
    wire [31:0] gcd_out;
    wire done;
    gcd uut (.num1(num1), .num2(num2), .gcd_out(gcd_out),.DONE(done), .clk(clk), .reset(reset));
initial begin
        num1 = 32'd0;num2 = 32'd0;
        clk = 0;
        reset = 1;
        end
    always
    #10 clk = ~clk;
initial begin
    #3000;
    $finish;
    end
initial begin
#80; reset = 0;#1000 reset = 1; #20 reset = 0;#1000 reset = 1;#20 reset = 0;
    end
initial begin
    #30;
num1 = 32'd3; num2 = 32'd5;
    #1000
num1 = 32'd21; num2 = 32'd7;
    #1000
num1 = 32'd91; num2 = 32'd273;
    end
```

Fig. 12: Testbench of GCD

If we talk about the logic flow then as we can see from the code of Datapath ,many components like Multiplexars , Registers , Subtractors , Comparators are used and their code is instanciated .

As we cans see from the state diagram , after goi signal becomes high value of xi and yi is load into the registers through multiplexars according to their select line . Now , these x and y input goes to comparator which creates the signal xlty (x is less than y) and xngy (x is not equal to y) so that we can decide x-y is done or y-x and then this value is feedback to one of the inputs of multiplexars untill x becomes equal to y .

V. Experimental Results and Discussions

A. Simulations of the GCD:-

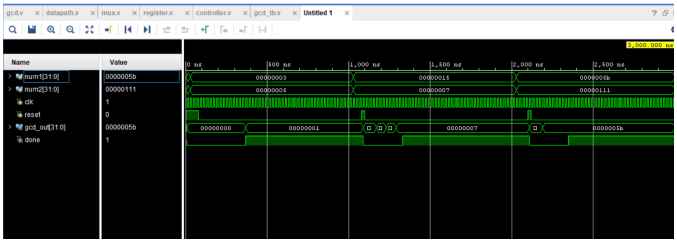

Fig. 13: Simulations of the GCD

Timing Analysis

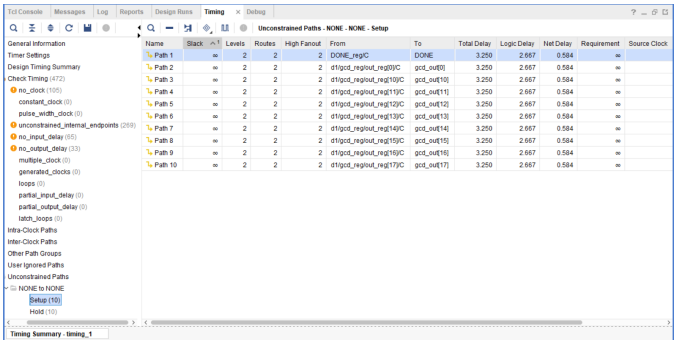

Fig. 14: Setup Time

So , Total Delay for Setup is 3.250 .
Logic Delay = 2.667
Net Delay = 0.584

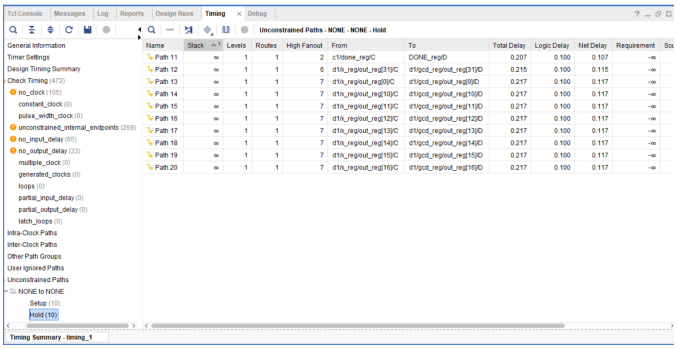

Fig. 15: Hold Time

So , Total Delay for Hold is 0.207 .
Logic Delay = 0.1
Net Delay = 0.107

VI. Conclusions

1. By writing simple codes of multiplexer and register and comparators , we can write verilog code of GCD's Datapath and Controller by instanciating them .
2 . Schemetic Diagram of the GCD is aslo easily provided .
3 . Simulations of the GCD is also given by writing its testbench code and as we can see by simulations it is giving correct result (GCD of 3 and 5 is 1 only) .

VII. Refernces

1. Embedded System Design book by Frank Vahid and Tony Givargis