

FORMAL VERIFICATION ON GCD UNIT

Abhinandan Majumdar
MS. Computer Engineering
am2993@columbia.edu

CSEE 6832

Fundamentals of Computer Systems

Michael Theobald

D.E. Shaw Researcher

michael.theobald@gmail.com

Franjo Ivancic

NEC Researcher

ivancic@gmail.com

ABSTRACT

A GCD Computer is a unit which computes the Greatest Common Divisor of two numbers based upon Euclid's Algorithm. Computation of the Greatest Common Divisor (GCD) of long integers is heavily used in computer algebra systems, because it occurs in normalization of rational numbers and other important sub-algorithms. In a typical algebraic computation more than half of the time is spent for calculating GCD of long integers, which sometimes becomes the critical path of the whole design. As a part of Formal Verification Course, we worked towards Formally Verifying an already existing a RTL model of GCD Module using VIS as a Model Checking Tool to verify its correctness in all possible scenarios.

INDEX

1. INTRODUCTION	4
1.1 Algorithm	4
1.2 Properties	4
2. DESIGN	6
2.1 RTL Design	6
2.2 Entity Level Design	6
2.3 Control Unit	6
2.4 Datapath Unit	7
3. VIS – A MODEL CHECKING TOOL	8
3.1 Architecture	8
4. WORK DONE	11
4.1 VHDL to Verilog Conversion	11
4.2 CTL Coding	11
4.3 VIS Modeling	11
5. RESULTS	13
5.1 Simulation Results	13
5.2 Model Checking Analysis	14
5.3 Analysis of an example	14
6. CONCLUSION	15
6.1 Result Summary	15
6.2 Experience	15
7. REFERENCE	16
APPENDIX	17

1. INTRODUCTION

In mathematics, the greatest common divisor (gcd), sometimes known as the greatest common factor (gcf) or highest common factor (hcf), of two non-zero integers, is the largest positive integer that divides both numbers without remainder.

1.1 Algorithm

The classical Euclid's algorithm to compute GCD of two numbers is given as below.

```
Input x, y;
while ( x ≠ y ) do
  if ( x > y )
    then x := x - y;
  else y := y - x;
  end if;
end while;
z := x;
end;
```

In this algorithm, we take two inputs as x and y, then perform subtraction of smaller no. from the bigger no. and replace it for the later part i.e. bigger no. and iterate until both the nos. are equal. In such a case, z return the GCD(x,y) i.e. the largest no. dividing both the nos. x and y.

1.2 Properties

The complexity of Euclid's algorithm depends solely on how complex the nos. are. The probability that two nos. selected would eventually yield a solution is $6/\pi^2 d^2$ where $d \mid x, y$ and $x \mid d$ and $y \mid d$ are co-prime. This algorithm eventually leads to a GCD for all cases, except if the any one of the input is equal or less than zero, where it goes to infinite iterations.

1. DESIGN

2.1 RTL Design

To start with the design, we first convert the Euclidean Algorithm in Register Transfer Logic (RTL) where the input lines x and y are stored in registers R_x and R_y which are being used for further computation. Register R_z stores the final result of computation i.e. $GCD(x,y)$.

```
Rx := x, Ry := y;
while ( Rx • Ry ) do
if ( Rx > Ry )
then Rx := Rx - Ry;
else Ry := Ry - Rx;
end if;
end while;
Rz := Rx;
```

2.2 Entity Level Design

Following is the basic entity for GCD. x and y are 8 bit input lines, clk is the clock driving the unit, en is the enable signal when asserted, starts the computation. At the output end, z is the 8 bit output line and EOC gets set when the computation is over.

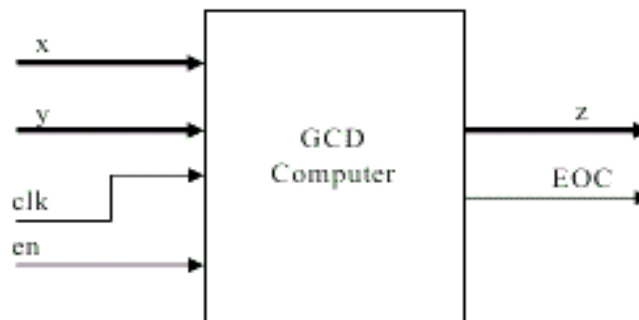


Fig 2.1: GCD Unit Entity

The entity is partitioned into Control and Datapath Unit, which are described in detail below.

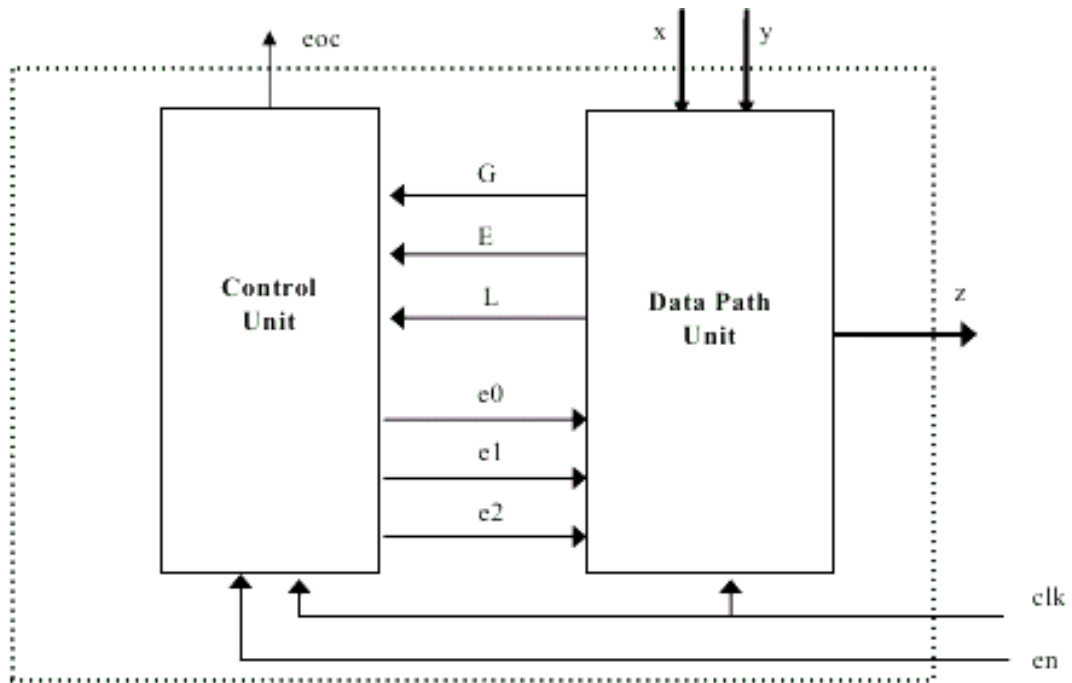


Fig 2.2: Architecture of GCD Unit

2.3 Control Unit

The Control Unit is a sequential circuit and an implementation of Finite State Machine which takes up G, E, L values being computed from Datapath Unit, and based upon that, sets $e0, e1, e2$ and eoc . Following is the FSM and the state transition diagram indicating the behavior of GCD Control Unit.

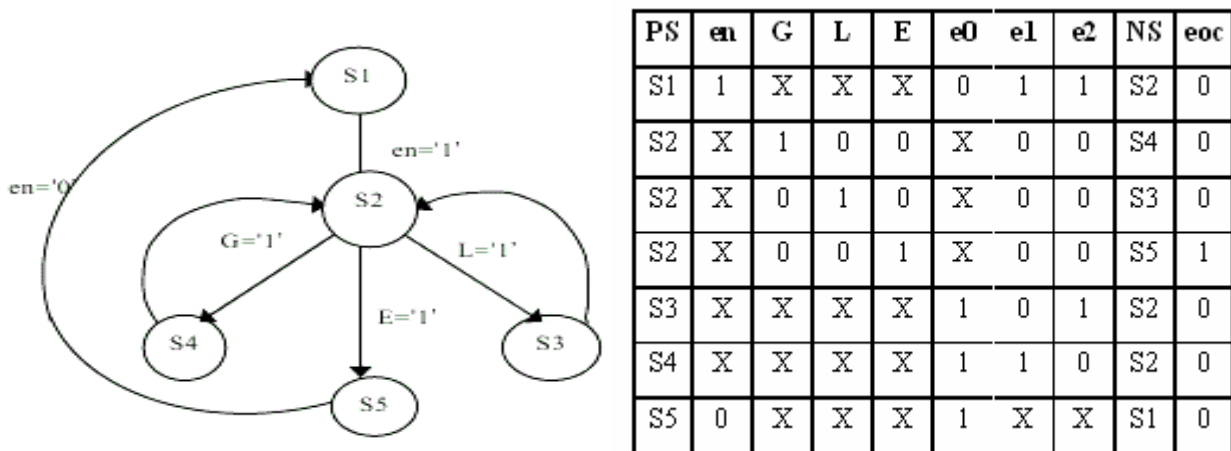


Fig 2.3: FSM and State Transition Table of Control unit

2.4 Datapath Unit

As a datapath unit, we have set of multiplexes controlled by $e0$, $e1$ which are in turn controlled by control unit. In the 1st most iteration, i.e. in state $S1$, x and y are latched to R_x and R_y , and if $R_x < R_y$, L signal is set, swapping the values of R_x and R_y for the subtractor to compute $R_y - R_x$ and set $e2$ to store the value in R_y only. Similarly if $R_x > R_y$, G signal is set. The subtractor computes $R_x - R_y$ and sets the values of R_x when the control unit sets $e1$. The iteration stop when the E signal is set to 1 indicating R_x and R_y are equal which is being fed to R_z enable signal to latch the z value as $GCD(x,y)$. The EOC signal is set to one after one clock cycle delay so as to make the z value stable before it can be used further.

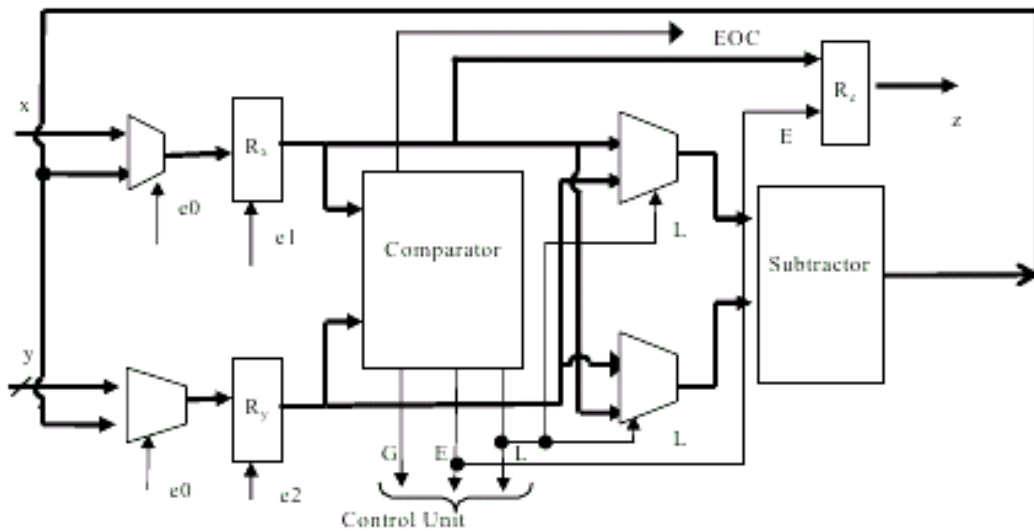


Fig 2.4: Datapath unit of GCD

2. VIS – A MODEL CHECKING TOOL

VIS (Verification Interacting with Synthesis) is basically a verification and synthesis system for finite-state hardware systems, developed at Berkeley and Boulder. It is an integrated system for model checking by specifying the properties which we want to verify written in CTL form, performs hierarchical synthesis, as well as verification.

3.1 Architecture.

VIS is divided into three parts: a common front end for reading in a description of a design, verification (VIS-v), and synthesis (VIS-s). The individual characteristics are described below.

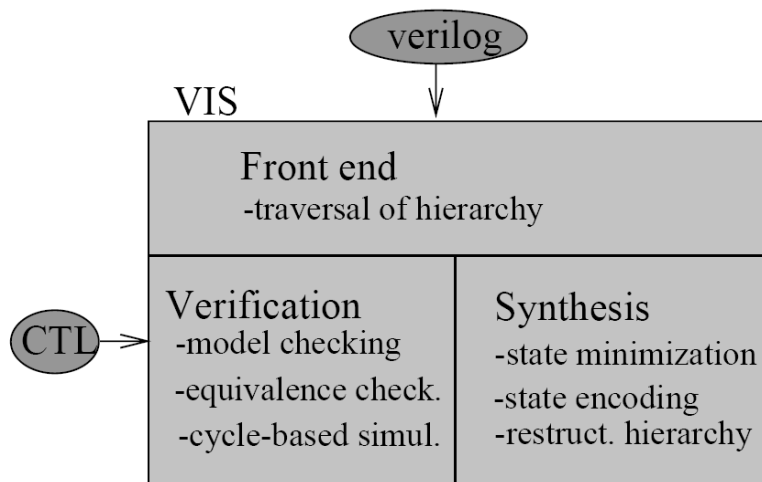


Fig 3.1 : VIS Architecture

- a) **Front End:** This portion reads and traverses a hierarchical system described in BLIF-MV (Berkeley Logic Interchange Format), which may have been compiled from a high level language like Verilog. BLIF-MV is a language designed for describing hierarchical sequential systems with non-determinism. A system can be composed of interacting sequential systems, each of which can be again described as a collection of communicating sequential systems. This makes it possible to describe systems in a hierarchical fashion. Although BLIF, the input language for the logic optimization

system SIS, also has constructs for describing hierarchies, they are automatically flattened into a single-level circuit once they are read in because the internal data structure of SIS does not support hierarchical representations. In VIS, however, the original hierarchy is preserved in internal data structures so that true hierarchical synthesis/verification is possible. BLIF-MV is an extension of BLIF which accepts non-deterministic behaviors. This is done by allowing non-deterministic gates in descriptions. Non-deterministic gates generate an output arbitrarily from the set of pre-specified outputs. These allow us to model non-deterministic systems in the VIS environment, which is crucial in formal verification since designs in early stages are likely to contain non-determinism. BLIF-MV supports multi-valued variables, which can be used to simplify system descriptions. Here, we use an external compiler VL2MV for generating Verilog to BLIF-MV format.

VL2MV extracts a set of interacting finite state machines (FSMs) that preserve the behavior of the source Verilog program defined in terms of simulated results. Allocation of hardware gates to operators in Verilog (resource binding) is based on the assumption of unlimited resources, where resources are all possible gates expressible in one table in BLIF-MV. No scheduling and optimization are performed, so the extracted FSMs are not guaranteed to be optimal (for area, speed, and so on). In order to optimize the logic, a synthesis program like SIS can be invoked on modules of the system.

- b) Verification** – This is basically the core part of VIS which actually performs Formal Verification of a system as per the constraints/specification/properties specified or expected. It has the capability to accept the properties by two theoretical approaches. The first is temporal logic model checking, where the properties to be checked are expressed as formulas in a temporal logic, and the system is expressed as a finite state system. In particular, Computational Tree Logic (CTL) model checking is a technique pioneered by Clarke and Emerson to verify whether a finite state system satisfies properties expressed as formulas in a branching-time temporal logic called CTL. On the other hand, there are certain properties that are not expressible in CTL, but can be

expressed as ω -automata. The second approach, language containment, requires the description of the system and properties as ω -automata, and verifies correctness by checking that the language of the system is contained in the language of the property.

VIS, as a model checking tool supports both the approaches. It accepts properties to be verified in CTL logic. It also checks for Language emptiness; however Language Containment is still under research. It also checks for Fairless Constraints specified as Büchi Automata i.e. set of states that are visited infinitely often. The internal VIS data structures have the capability to support more complicated fairness constraints to make it more effective.

c) **Synthesis** - VIS also has the capability to interface with SIS to optimize logic modules; hence, VIS is an integrated system for hierarchical synthesis, as well as verification. It supports a full-fledged hierarchical synthesis flow that translates a Verilog description into an optimized multi-level circuit at the gate level. VIS can interact with SIS in order to optimize the existing logic. There are two possible goals/scenarios:

(i) **Synthesis for verification** - Synthesis can be used to optimize the logic that represents the system, for simpler verification.

(ii) **Front-end to synthesis** - Files described in Verilog and compiled into *blif_mv* using VL2MV or another tool) can be synthesized by using VIS and SIS together.

3. WORK DONE

We started with the VHDL coded design of GCD and proceeded as follows.

4.1 VHDL to VERILOG Conversion

We converted our existing design of GCD originally coded in VHDL to Verilog hierarchically by XHDL software of x-tekcorp Inc. Then, we rewrote the code, so as to be accepted by VIS according to its specifications like, inclusion of all modules in a single file, removal of non-blocking statements to remove non-determinism error, hard-coding some of the values to avoid indeterminism etc.

4.2 CTL CODING

We intended to verify following properties and have written corresponding CTLs to formally verify the GCD unit, in a file named *gcd.ctl*.

- a) The iteration of GCD Algorithm should eventually converge for almost all input possibilities ($start=1 \rightarrow AF(eoc=1)$).
- b) For a particular input case where one of the input is zero, the system should loop infinitely and hence must fail ($start=1 \rightarrow AF(eoc=1)$).
- c) After comparison, only one of the enable input of R_x and R_y should be 1 ($start=1 \rightarrow AG(!(e1=1) \wedge (e2=1))$).

4.3 VIS MODELLING

We did following steps to formally verify our CTLs for the GCD unit.

- a) **Convert Verilog to BLIF_MV** – We used *vl2mv* compiler to read our verilog source and convert it into BLIF_MV file using the command *vl2mv*.
- b) **Read BLIF_MV file** – In VIS environment, we read the mv file by using the command *read_blif_mv gcd.mv*
- c) **InitVerify** - We run *init_verify* command which basically does three things
 - (i) *flatten_hierarchy* – It flattens the hierarchical description into a single network (netlist of multi-valued logic gates). The output is computed from

the inputs of the design by the network circuit, which consists of logic gates, interconnections between them, and latches to represent the sequential elements. *flatten_hierarchy* automatically checks each table in the network for being deterministic (except for pseudo-inputs) and completely specified.

(ii) ***static_order*** – It converts the network representation into a functional description that represents the output and next state variables as a function of the inputs and current state variables. We use the BDD (binary decision diagram) and its extension the MDD (multivalued decision diagram) to represent boolean and discrete functions. Before creating the MDDs, it is necessary to order the variables in the support of the MDD. This is accomplished by the *static_order* command, which gives an initial ordering. Networks with combinational cycles cannot be ordered. If the MDD variables have already been ordered, then *static_order* does nothing.

(iii) ***build_partition_mdd*** - The *build_partition_mdds* command computes the transition function MDDs. Depending on the partitioning method selected, the MDDs for the combinational outputs (COs) are built in terms of either the combinational inputs (CIs) or some subset of intermediate nodes of the network. The MDDs built are stored in a DAG called a “partition”. The vertices of a partition correspond to the CIs, COs, and any intermediate nodes used. Each vertex has a multi-valued function (represented by an MDD) expressing the function of the corresponding network node in terms of the partition vertices in its transitive fanin. Hence, the MDDs of the partition represent a partial collapsing of the network.

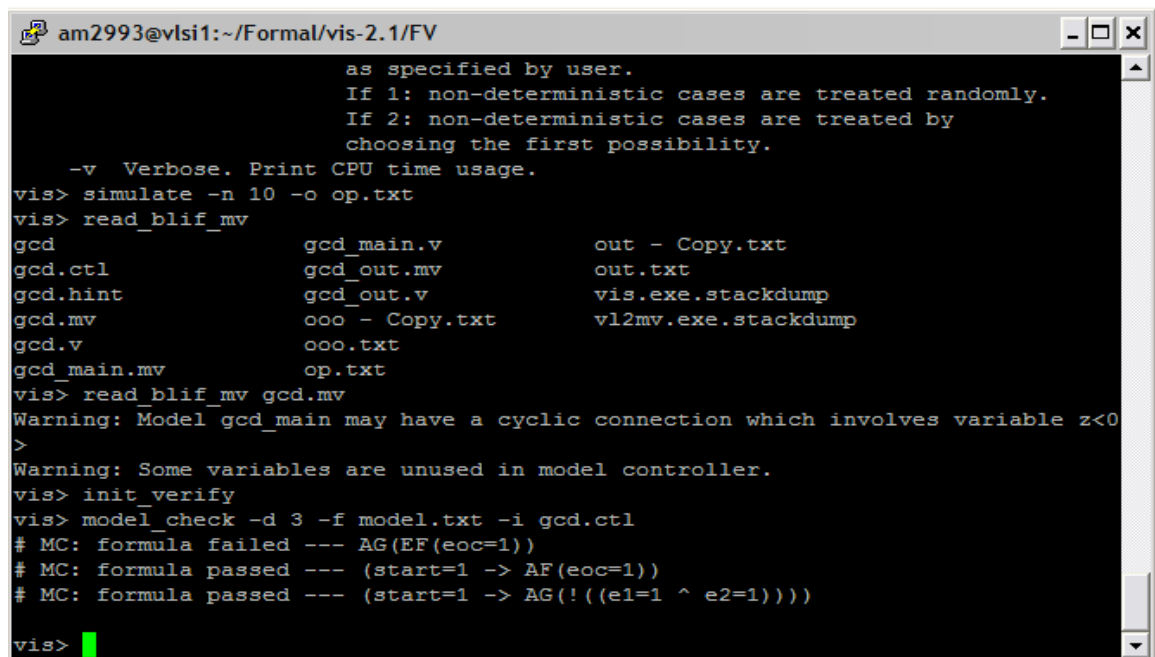
d) **ModelCheck** – Here we simply execute the *command model_check gcd.mv -d 3 -f error.out gcd.ctl*. This command specifies which CTLs passes and gives an counter example with the errorneous path if the CTL fails.

e) **Simulation** – We could also simulate the design by using the command *simulate -n 10*, where 10 indicates for 10 iterations with randomly generated input values.

5.2 Model Checking Analysis – For the properties we intended to test, we obtained following results.

- a) Property 4.2 a) passed, indication there is a way to find the gcd of two nos. and the algorithm would eventually come out of the loop and set the eoc to 1.
- b) Property 4.2 b) passed, even though we expected it to fail. This has been proven by the simulation results also that if one of the input is zero, when simulated for 1000 iterations, it loops infinitely and never yields the result.
- c) Property 4.2 c) passed, indicating it will never set both e1 and e2 to 1 just after subtraction so that the result gets stored uniquely to a single register rather than in both or the wrong one.
- d) We also tested for miscellaneous property which we knew would fail, like $eoc = 1$ globally, and it has been proven wrong by giving a counter example.

The net summary of the model checking displayed as per VIS is as below.



```
am2993@vlsi1:~/Formal/vis-2.1/FV
as specified by user.
If 1: non-deterministic cases are treated randomly.
If 2: non-deterministic cases are treated by
choosing the first possibility.
-v Verbose. Print CPU time usage.
vis> simulate -n 10 -o op.txt
vis> read_blif_mv
gcd          gcd_main.v          out - Copy.txt
gcd.ct1      gcd_out.mv          out.txt
gcd.hint     gcd_out.v           vis.exe.stackdump
gcd.mv       ooo - Copy.txt      v12mv.exe.stackdump
gcd.v        ooo.txt
gcd_main.mv  op.txt
vis> read_blif_mv gcd.mv
Warning: Model gcd_main may have a cyclic connection which involves variable z<0
>
Warning: Some variables are unused in model controller.
vis> init_verify
vis> model_check -d 3 -f model.txt -i gcd.ct1
# MC: formula failed --- AG(EF(eoc=1))
# MC: formula passed --- (start=1 -> AF(eoc=1))
# MC: formula passed --- (start=1 -> AG(!((e1=1 ^ e2=1))))
vis>
```

Fig 5.1: Screenshot of CTL properties being verified

5.3 Analysis of an Example – As a part of VIS tool, we had a GCD example, and we tried to verify that design. The results obtained were exactly similar to what we obtained for our own design. On analyzing the code further, we found it had separate support for handling 4.2 b) case, i.e. if any one the input is zero, it directly sets the $eoc = 1$. We tried to comment that case, but the property still passed.

5. CONCLUSION

6.1 Result Summary

In brief, we can summarize the results obtained in a tabular fashion as below.

Property Checked	CTL	Result	Conclusion
Should eventually yield the result	$(start=1 \rightarrow AF(eoc=1))$	PASS	Expected
Should go in infinite loop if one of the input is zero	$(start=1 \rightarrow AF(eoc=1))$	FAIL	Unexpected
Should never set $e1$ and $e2$ as 1 just after subtraction	$(start=1 \rightarrow AG(!(e1=1) \wedge (e2=1)))$	PASS	Expected

6.2 Experience

The experience of formally verifying a GCD unit gave us a thorough insight of what all post designing phases are, how can a system fail to work even though lot of design decisions has been taken into account, how appropriately to write a property to be tested in a CTL which covers all failure, deadlock, exception and starvation cases, and how to effectively get to a counter example to be sure of the possibility of existence of such a scenario. It also helped us to enhance our knowledge upon theoretical aspects of Formal Verification learnt in class and to effectively implement it for our project. It also gave us an experience of working with VIS and to get exposed to its functionality and limitations. Hence I am thankful to everyone associated with the project especially Prof. Michael Theobald, and Prof. Franjo Ivancic. Their constant advice, support and guidance throughout the project phase saved the work to fall into pitfalls. Hence I humbly dedicate this piece of work of mine to them without whom it is worthless.

6. REFERENCE

- [1] VIS User's Manual by Tiziano Villa, Gitanjali Swamy, and Thomas Shiple
- [2] <http://vlsi.colorado.edu/~vis/doc/blifmv/blifmv/blifmv.html>
- [3] http://en.wikipedia.org/wiki/Greatest_common_divisor
- [4] http://en.wikipedia.org/wiki/Euclidean_algorithm
- [5] http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

APPENDIX

GCD Source Code

```
module gcd_main(x,y,clk,en,z,eoc);

input[7:0] x, y;
input clk,en;
output[7:0] z;
output eoc;

wire [7:0] x;
wire [7:0] y;
wire clk;
wire [7:0] z;
wire eoc;

wire e0;
wire e1;
wire e2;
wire e3;
wire g;
wire l;
wire e;
wire gi;
wire li;
wire ei;
wire ox;
wire en;
wire [7:0] a;
wire [7:0] b;
wire [7:0] d;
wire [7:0] f;
wire [7:0] h;
wire [7:0] i;
wire [7:0] j;
wire [7:0] k;

assign gi = 1'b0;
assign li = 1'b0;
assign ei = 1'b1;

reg start;
reg [7:0] na;
reg [7:0] nb;

initial begin
na = 0;
nb = 0;
start = 0;
end
```

```

always @ (posedge clk) begin
    na = x;
    nb = y;

    // nb = 8'b00000000; //When set to zero
    start = en;
end

controller CNTRL(g,e,l,clk,start,e0,e1,e2,eoc);
mux8 MUX1(na,d,e0,a);
mux8 MUX2(nb,d,e0,b);
regis8 REGX(a,clk,e1,f);
regis8 REGY(b,clk,e2,h);
comp8 COMP(f,h,gi,ei,li,g,e,l);
mux8 MUX3(f,h,l,i);
mux8 MUX4(h,f,l,j);
sub8 SUBT(i,j,d,ox);
regis8 REGZ(f,clk,e,k);

assign z = k;
endmodule

module controller (g,e,l,clk,en,e0,e1,e2,eoc);

input g;
input e;
input l;
input clk;
input en;
output e0;
wire e0;
output e1;
wire e1;
output e2;
wire e2;
output eoc;
reg eoc;

initial eoc = 1'b0;

reg [2:0] Sreg0;
parameter S1 = 3'b001; parameter S2 = 3'b010;
parameter S3 = 3'b011; parameter S4 = 3'b100;
parameter S5 = 3'b101;

initial Sreg0 = S1;

always @(posedge clk) begin
    case(Sreg0)

        S1:
            if(en)
                begin
                    eoc = 1'b0;

```

```

        Sreg0 = S2;
    end
    else
        Sreg0 = S1;

        S2:

        if(g==1'b1)
            Sreg0 = S4;

        else if (l==1'b1)
            Sreg0 = S3;
    else
        Sreg0 = S5;

        S3:
        Sreg0 = S2;

        S4:
        Sreg0 = S2;

        S5:
        if(en==1'b0)
            begin
                eoc = 1'b0;
                Sreg0 = S1;
            end
        else
            begin
                eoc = 1'b1;
                Sreg0 = S5;
            end
        end
    endcase

    end
assign e0 = (Sreg0 == S1 & en == 1'b1) ? 1'b0 : (Sreg0 == S3) ? 1'b1 :
(Sreg0 == S4) ? 1'b1 : 1'b0;
assign e1 = (Sreg0 == S1 & en == 1'b1) ? 1'b1 : (Sreg0 == S2) ? 1'b0 :
(Sreg0 == S3) ? 1'b0 : (Sreg0 == S4)?1'b1:1'b0;
assign e2 = (Sreg0 == S1 & en == 1'b1) ? 1'b1 : (Sreg0 == S2) ? 1'b0 :
(Sreg0 == S3) ? 1'b1 : (Sreg0 == S4)?1'b0:1'b0;
//assign eoc = (Sreg0 == S5 & start == 1'b1) ? 1'b1 : 1'b0;
endmodule

module regis8(idata, clk, en, odata);
    input [7:0]  idata;
    input        clk;
    input        en;
    output [7:0] odata;
    reg [7:0]    odata;

```

```

initial odata = 8'b0;

always @(posedge clk)

    begin
        if (en == 1'b1)
            odata = idata;
        end
endmodule

module comp8(a, b, gi, ei, li, go, eo, lo);
    input [7:0] a;
    input [7:0] b;
    input      gi;
    input      ei;
    input      li;
    output     go;
    output     eo;
    output     lo;

    wire      f;
    wire      g;
    wire      h;
    wire      i;
    wire      j;
    wire      k;
    wire      l;
    wire      m;
    wire      n;
    wire      o;
    wire      p;
    wire      q;
    wire      r;
    wire      s;
    wire      t;
    wire      u;
    wire      v;
    wire      w;
    wire      x;
    wire      y;
    wire      z;

    comp C1(a[0], b[0], gi, ei, li, f, g, h);
    comp C2(a[1], b[1], f, g, h, i, j, k);
    comp C3(a[2], b[2], i, j, k, l, m, n);
    comp C4(a[3], b[3], l, m, n, o, p, q);
    comp C5(a[4], b[4], o, p, q, r, s, t);
    comp C6(a[5], b[5], r, s, t, u, v, w);
    comp C7(a[6], b[6], u, v, w, x, y, z);

```

```

    comp C8(a[7], b[7], x, y, z, go, eo, lo);
endmodule

module comp(ai, bi, gl, el, ll, gi, ei, li);
    input  ai;
    input  bi;
    input  gl;
    input  el;
    input  ll;
    output gi;
    output ei;
    output li;

    assign ei = el & ~(ai ^ bi);
    assign gi = (ai & (~bi)) | (gl & ~(ai ^ bi));
    assign li = ((~ai) & bi) | (ll & ~(ai ^ bi));
endmodule

module mux8(x0, x1, sel, y);
    input [7:0] x0;
    input [7:0] x1;
    input      sel;
    output [7:0] y;

    assign y = (sel == 1'b0) ? x0 : x1;
endmodule

module sub8(x, y, z, bout);
    input [7:0] x;
    input [7:0] y;
    output [7:0] z;
    output      bout;

    wire [7:0] t;
    wire      a;
    wire      b;
    wire      c;
    wire      d;
    wire      e;
    wire      f;
    wire      g;
    wire      h;
    assign t = ~y;
    assign a = 1'b1;

    adder AD1(x[0], t[0], a, z[0], b);
    adder AD2(x[1], t[1], b, z[1], c);
    adder AD3(x[2], t[2], c, z[2], d);

```

```

    adder AD4(x[3], t[3], d, z[3], e);
    adder AD5(x[4], t[4], e, z[4], f);
    adder AD6(x[5], t[5], f, z[5], g);
    adder AD7(x[6], t[6], g, z[6], h);
    adder AD8(x[7], t[7], h, z[7], bout);

endmodule

module adder(x, y, cin, s, cout);
    input  x;
    input  y;
    input  cin;
    output s;
    output cout;

    assign s = x ^ y ^ cin;
    assign cout = (y & cin) | (x & (cin | y));

endmodule

```

CTL Values

```

AG(EF(eoc=1));
(start=1 -> (AF(eoc=1)));
((start=1) -> AG(!((e1=1)^(e2=1))));

```