**Cube Calculus Machine**

**ECE574 High Level Synthesis**

**Matt Blackmore  
Brennan Hughes  
Alberto Patino**

**Introduction:**

Cube Calculus is an algebraic model to process Boolean and multi-valued functions.

There are three different types of cube calculus operations

1) Simple Combinational Operations

2) Complex Combinational Operations

3) Sequential Operations.

*Simple Combinational Operation*

Defined as a Single set operation on all pairs of true sets and produces one resultant cube.

Intersection and Supercube are simple combinational cube operations.

*Complex Combinational Operation*

They have two set operations and one set relation.

All variables whose pair of true sets satisfies a relation are said to be Special Variable.

Two set operations are called before and active, the active set operation is applied on true sets of special variables and before set operation to others

The examples are Prime operation, Cofactor operation, Consensus operation.

*Sequential Operations*

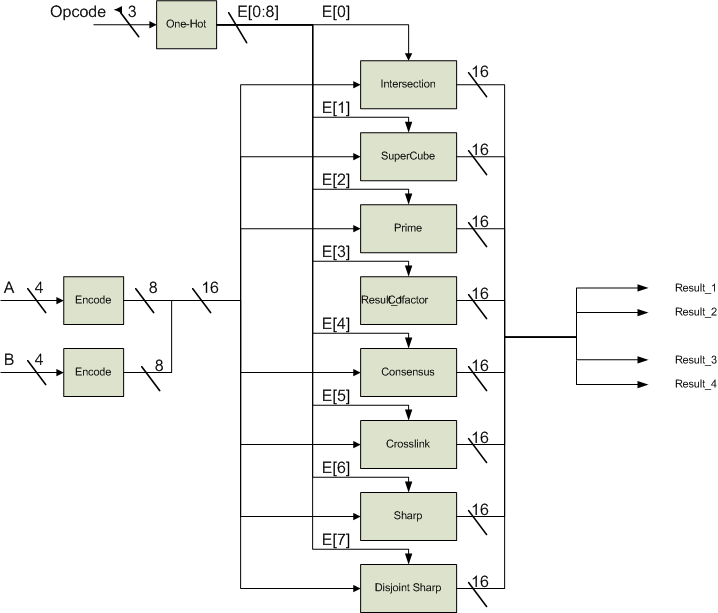
They have three set operations and one set relation.

All variables whose pair of true sets satisfies a relation are said to be Special Variable.

Three set operations are called before, active and after.

The examples are Sharp operation, Disjoined Sharp operation

**Machine:** The machine code was written to be fully expandable

****

**Arbiter**

Inputs: op\_code, cube\_A, cube\_B

Outputs: Result Cubes 1-4

Functionality:

Enables appropriate function based on op\_code,

Encodes input cubes to positional notation

Decodes outputs to named notation

The arbiter is the controller for this simple cube calculus machine. It takes an op\_code and two cubes, in named notation, as inputs and produces two outputs, one in named notation and one in positional notation. The main function of the arbiter is to encode incoming cubes into positional notation, enable the correct cube calculus function based on the op\_code, then output the resultant cube(s) in positional notation and decode the resultant positional notation back to named notation and output it as well. These two outputs are essentially one dimensional arrays of resultant cubes. Certain cube calculus functions may output between zero and ‘bit-size’ cubes; bit size being the largest number of bits in the incoming cubes. The nomenclature decided for the inputs and outputs of this project is as follows, incoming and outgoing logic highs and lows are represented by ‘1’ and ‘0’ respectively, incoming and outgoing don’t-cares are represented as ‘X’, outgoing epsilon is represented by high impedance ‘Z’. The arbiter, along with all the cube calculus function modules, is fully scalable. When creating a test bench simply specify the parameter MAX\_BITS when instantiating the arbiter to be the number of bits desired for the incoming cubes.

module CC\_Arbiter (A,B,OP\_CODE,OUTPUT\_VECTORS,POS\_OUTPUT\_VECTORS);

parameter MAX\_BITS = 4; // represents input variables bit size

parameter INTERSECTION = 3'b000;

parameter SUPERCUBE = 3'b001;

parameter PRIME = 3'b010;

parameter CONSENSUS = 3'b011;

parameter COFACTOR = 3'b100;

parameter CROSSLINK = 3'b101;

parameter SHARP = 3'b110;

parameter D\_SHARP = 3'b111;

output OUTPUT\_VECTORS, POS\_OUTPUT\_VECTORS; // outputs flattened to 1D arrays for ports

reg [MAX\_BITS\*MAX\_BITS-1:0] OUTPUT\_VECTORS; // reg [MAX\_BITS-1:0] OUTPUT\_VECTORS [MAX\_BITS-1:0];

reg [MAX\_BITS\*2\*MAX\_BITS-1:0] POS\_OUTPUT\_VECTORS; // reg [MAX\_BITS\*2-1:0] POS\_OUTPUT\_VECTORS [MAX\_BITS-1:0];

input [MAX\_BITS-1:0] A,B;

input [2:0] OP\_CODE;

reg [MAX\_BITS\*2-1:0] Apos, Bpos, enable; // input cubes and enable

wire [MAX\_BITS\*MAX\_BITS\*2-1:0] int\_vectors; // 1D intermediate cube results

reg [MAX\_BITS-1:0] i,j,k; // loop indices

// instantiate modules (one for each cube calculus function)

CC\_INT #(MAX\_BITS) func0(enable[0], Apos, Bpos, int\_vectors);

CC\_SUC #(MAX\_BITS) func1(enable[1], Apos, Bpos, int\_vectors);

CC\_PRM #(MAX\_BITS) func2(enable[2], Apos, Bpos, int\_vectors);

CC\_CNS #(MAX\_BITS) func3(enable[3], Apos, Bpos, int\_vectors);

CC\_COF #(MAX\_BITS) func4(enable[4], Apos, Bpos, int\_vectors);

CC\_CSL #(MAX\_BITS) func5(enable[5], Apos, Bpos, int\_vectors);

CC\_SHP #(MAX\_BITS) func6(enable[6], Apos, Bpos, int\_vectors);

CC\_DSP #(MAX\_BITS) func7(enable[7], Apos, Bpos, int\_vectors);

// initialization

initial

begin

enable = 0;

Apos = 0;

Bpos = 0;

OUTPUT\_VECTORS = 0;

POS\_OUTPUT\_VECTORS = 0;

end

// encode A to positional notation

always @ (A)

begin

for(i=0; i<MAX\_BITS; i=i+1 )

begin

case(A[i])

1'b0: begin Apos[2\*i+1] = 1; Apos[2\*i] = 0; end

1'b1: begin Apos[2\*i+1] = 0; Apos[2\*i] = 1; end

1'bx: begin Apos[2\*i+1] = 1; Apos[2\*i] = 1; end

default: begin Apos[2\*i+1] = 0; Apos[2\*i] = 0; end

endcase

end

end

// encode B to positional notation

always @ (B)

begin

for(i=0; i<MAX\_BITS; i=i+1 )

begin

case(B[i])

1'b0: begin Bpos[2\*i+1] = 1; Bpos[2\*i] = 0; end

1'b1: begin Bpos[2\*i+1] = 0; Bpos[2\*i] = 1; end

1'bx: begin Bpos[2\*i+1] = 1; Bpos[2\*i] = 1; end

default: begin Bpos[2\*i+1] = 0; Bpos[2\*i] = 0; end

endcase

end

end

// determine function to call (one-hot enable)

always @ (OP\_CODE)

begin

case(OP\_CODE) // synthesis full\_case

INTERSECTION: enable = 8'b00000001;

SUPERCUBE: enable = 8'b00000010;

PRIME: enable = 8'b00000100;

CONSENSUS: enable = 8'b00001000;

COFACTOR: enable = 8'b00010000;

CROSSLINK: enable = 8'b00100000;

SHARP: enable = 8'b01000000;

D\_SHARP: enable = 8'b10000000;

endcase

end

// decode intermediate results and send to output

always @ (int\_vectors)

begin

POS\_OUTPUT\_VECTORS = int\_vectors;

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS; i=i+1)

begin

case({int\_vectors[j\*MAX\_BITS\*2+(i\*2)+1],int\_vectors[j\*MAX\_BITS\*2+(i\*2)]})

2'b01: OUTPUT\_VECTORS[j\*MAX\_BITS+i] = 1'b1;

2'b10: OUTPUT\_VECTORS[j\*MAX\_BITS+i] = 1'b0;

2'b11: OUTPUT\_VECTORS[j\*MAX\_BITS+i] = 1'bx;

default: OUTPUT\_VECTORS[j\*MAX\_BITS+i] = 1'bz;

endcase

end

end

end

endmodule

**Intersection**

The intersection operation is the bit-wise AND of each cube.

module CC\_INT (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS-1:0] i,j;

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

result\_vectors[MAX\_BITS\*2-1:0] = Apos & Bpos;

for(j=1; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

else

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end

endmodule

**Supercube**

The supercube operation is the bit-wise OR of each cube.

module CC\_SUC (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS-1:0] i,j;

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

result\_vectors[MAX\_BITS\*2-1:0] = Apos | Bpos;

for(j=1; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

else

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end

endmodule

**Prime**

A’B

module CC\_PRM (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS\*2-1:0] special;

reg [MAX\_BITS-1:0] i,j;

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

special = Apos & Bpos;

for(i=0; i<MAX\_BITS; i=i+1)

begin

if({special[2\*i+1],special[2\*i]} == 2'b00)

begin result\_vectors[2\*i+1] = Apos[2\*i+1]; result\_vectors[2\*i] = Apos[2\*i]; end

else

begin {result\_vectors[2\*i+1],result\_vectors[2\*i]} = {Apos[2\*i+1],Apos[2\*i]} | {Bpos[2\*i+1],Bpos[2\*i]}; end

end

for(j=1; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

else

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end

endmodule

**Cofactor**

module CC\_COF (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS\*2-1:0] special;

reg [MAX\_BITS-1:0] i,j;

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

special = Apos | ~Bpos;

for(i=0; i<MAX\_BITS; i=i+1)

begin

if({special[2\*i+1],special[2\*i]} == 2'b11)

begin result\_vectors[2\*i+1] = 1'b1; result\_vectors[2\*i] = 1'b1; end

else

begin result\_vectors[2\*i+1] = Apos[2\*i+1] & Bpos[2\*i+1]; result\_vectors[2\*i] = Apos[2\*i] & Bpos[2\*i]; end

end

for(j=1; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

else

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end

endmodule

**Consensus**

module CC\_CNS (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS\*2-1:0] special;

reg [MAX\_BITS-1:0] i,j;

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

special = Apos & Bpos;

for(i=0; i<MAX\_BITS; i=i+1)

begin

if({special[2\*i+1],special[2\*i]} == 2'b00)

begin result\_vectors[2\*i+1] = Apos[2\*i+1] | Bpos[2\*i+1]; result\_vectors[2\*i] = Apos[2\*i] | Bpos[2\*i]; end

else

begin result\_vectors[2\*i+1] = Apos[2\*i+1] & Bpos[2\*i+1]; result\_vectors[2\*i] = Apos[2\*i] & Bpos[2\*i]; end

end

for(j=1; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

else

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end

endmodule

**Crosslink**

// RESULTANT CUBES OF CROSSLINK SHOULD BE XORed

module CC\_CSL (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS\*2-1:0] special;

reg [MAX\_BITS-1:0] i, j, k, numActives;

reg break;

initial

begin

break = 1'b0;

numActives = 0;

end

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

special = Apos & Bpos;

// determine if we can perform the operation and count number of actives

for(i = 0; i < MAX\_BITS; i=i+1)

begin

if({special[2\*i+1],special[2\*i]} == 2'b01 || {special[2\*i+1],special[2\*i]} == 2'b10) // cannot perform operation

break = 1'b1; // quit

else if ({special[2\*i+1],special[2\*i]} == 2'b00) // active

numActives = numActives + 1;

end

if(break == 1'b1) // can't perform operation, output high-z

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

if(break == 1'b0 && numActives == 0) // all don't cares on inputs = all don't cares on outputs

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'b1;

end

end

end

else if(break == 1'b0 && numActives > 0) // can perform operation and have actives

begin

k=0; //num\_outputs

for(i=0; i<MAX\_BITS; i=i+1)

begin

if({special[(i\*2)+1],special[i\*2]} == 2'b00) // active

begin

{result\_vectors[k\*MAX\_BITS\*2+(2\*i)+1],result\_vectors[k\*MAX\_BITS\*2+(2\*i)]} = 2'b11; //active becomes don't care

for(j=i+1; j<MAX\_BITS; j=j+1) //for everything 'before'(greater than) active, bring down B

begin

{result\_vectors[k\*MAX\_BITS\*2+(j\*2)+1],result\_vectors[k\*MAX\_BITS\*2+(j\*2)]} = {Bpos[2\*j+1],Bpos[2\*j]};

end //for

for(j=0; j<i; j=j+1) // for everything 'after'(less than) active, bring down A

begin

{result\_vectors[k\*MAX\_BITS\*2+(2\*j)+1],result\_vectors[k\*MAX\_BITS\*2+(2\*j)]} = {Apos[2\*j+1],Apos[2\*j]};

end //for

k=k+1; // once we find an active, increment to using the next output

end //if

end //for

end //else if

// high-z all unused outputs

for(j=MAX\_BITS-numActives; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

else //if

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end //always

endmodule

**Sharp**

A#B means that the result is the CUBE A minus CUBE B.

module CC\_SHP (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS-1:0] i,j,k,q; // loop indices

reg [MAX\_BITS\*2-1:0] special;

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

special = Apos & Bpos;

special = Apos ^ special;

for(i=0; i<MAX\_BITS; i=i+1)

begin

if({special[(i\*2)+1],special[i\*2]} == 2'b00) // if special is 00, output is all zeros

begin

for(k=0; k<MAX\_BITS\*2; k=k+1)

begin

result\_vectors[i\*MAX\_BITS\*2+k] = 1'b0;

end

end //if

else

begin

{result\_vectors[i\*MAX\_BITS\*2+(i\*2)+1],result\_vectors[i\*MAX\_BITS\*2+(i\*2)]} = {Apos[i\*2+1],Apos[i\*2]} & ~{Bpos[i\*2+1],Bpos[i\*2]};

q=i\*2+2;

for(k=0; k<MAX\_BITS\*2-q; k=k+1)

begin

result\_vectors[i\*MAX\_BITS\*2+MAX\_BITS\*2-1-k] = Apos[MAX\_BITS\*2-1-k];

end

for(k=0; k<i; k=k+1)

begin

{result\_vectors[i\*MAX\_BITS\*2+(k\*2)+1],result\_vectors[i\*MAX\_BITS\*2+(k\*2)]} = {Apos[k\*2+1],Apos[k\*2]};

end

end //else

end //for(i=0)

end //if(enable)

else

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end

endmodule

**Disjoined\_Sharp**

A#B means that the result is the CUBE A minus CUBE B.

module CC\_DSP (enable, Apos, Bpos, result\_vectors);

parameter MAX\_BITS = 4;

output result\_vectors;

reg [MAX\_BITS\*MAX\_BITS\*2-1:0] result\_vectors;

input enable;

input [MAX\_BITS\*2-1:0] Apos, Bpos;

reg [MAX\_BITS-1:0] i,j,k,q; // loop indices

reg [MAX\_BITS\*2-1:0] special;

always @ (enable or Apos or Bpos)

begin

if(enable)

begin

special = Apos & Bpos;

special = Apos ^ special;

for(i=0; i<MAX\_BITS; i=i+1)

begin

if({special[(i\*2)+1],special[i\*2]} == 2'b00) // if special is 00, output is all zeros

begin

for(k=0; k<MAX\_BITS\*2; k=k+1)

begin

result\_vectors[i\*MAX\_BITS\*2+k] = 1'b0;

end

end //if

else

begin

{result\_vectors[i\*MAX\_BITS\*2+(i\*2)+1],result\_vectors[i\*MAX\_BITS\*2+(i\*2)]} = {Apos[i\*2+1],Apos[i\*2]} & ~{Bpos[i\*2+1],Bpos[i\*2]};

q=i\*2+2;

for(k=0; k<MAX\_BITS\*2-q; k=k+1)

begin

result\_vectors[i\*MAX\_BITS\*2+MAX\_BITS\*2-1-k] = Apos[MAX\_BITS\*2-1-k] & Bpos[MAX\_BITS\*2-1-k];

end

for(k=0; k<i; k=k+1)

begin

{result\_vectors[i\*MAX\_BITS\*2+(k\*2)+1],result\_vectors[i\*MAX\_BITS\*2+(k\*2)]} = {Apos[k\*2+1],Apos[k\*2]};

end

end //else

end //for(i=0)

end //if(enable)

else

begin

for(j=0; j<MAX\_BITS; j=j+1)

begin

for(i=0; i<MAX\_BITS\*2; i=i+1)

begin

result\_vectors[j\*MAX\_BITS\*2+i] = 1'bz;

end

end

end

end

endmodule

**Testbench:** Test of basic machine functionality

module CC\_test ();

parameter MAX\_BITS = 4;

reg [MAX\_BITS-1:0] a,b;

reg [2:0] op\_code;

wire [MAX\_BITS\*MAX\_BITS-1:0] output\_1D; // flattened 1D array

wire [MAX\_BITS\*2\*MAX\_BITS-1:0] pos\_output\_1D; // flattened 1D array

reg [MAX\_BITS-1:0] output\_2D [MAX\_BITS-1:0];

reg [MAX\_BITS-1:0] i,j;

CC\_Arbiter #(MAX\_BITS) test(a,b,op\_code,output\_1D,pos\_output\_1D);

initial

begin

// unflatten output

for(i=0; i<MAX\_BITS; i=i+1)

begin

for(j=0; j<MAX\_BITS\*2; j=j+1)

begin

output\_2D[i][j] = output\_1D[i\*MAX\_BITS\*2+j];

end

end

a = 4'b0xxx; b = 4'bxx0x; op\_code = 3'b000; //\*\*\*\*\*\*\*\*\*\*

#20 a = 4'bxx10; b = 4'b10xx; op\_code = 3'b000; // INTERSECTION

#20 a = 4'bxx01; b = 4'b1x0x; op\_code = 3'b000; //\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b0000; b = 4'b0x01; op\_code = 3'b001; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b1010; b = 4'b1010; op\_code = 3'b001; // SUPERCUBE

#20 a = 4'b1101; b = 4'b1001; op\_code = 3'b001; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b0101; b = 4'b1x1x; op\_code = 3'b010; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b1010; b = 4'b0101; op\_code = 3'b010; // PRIME

#20 a = 4'b1001; b = 4'b0x01; op\_code = 3'b010; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b110x; b = 4'b10xx; op\_code = 3'b011; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b1010; b = 4'b0101; op\_code = 3'b011; // CONSENSUS

#20 a = 4'b10x1; b = 4'b0x01; op\_code = 3'b011; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b111x; b = 4'b1xxx; op\_code = 3'b100; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b1010; b = 4'b0101; op\_code = 3'b100; // COFACTOR

#20 a = 4'b10x1; b = 4'b0x01; op\_code = 3'b100; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b0x0x; b = 4'b1x1x; op\_code = 3'b101; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b1010; b = 4'b0101; op\_code = 3'b101; // CROSSLINK

#20 a = 4'b111x; b = 4'b001x; op\_code = 3'b101; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b0xxx; b = 4'bxx11; op\_code = 3'b110; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b101x; b = 4'bxx11; op\_code = 3'b110; // SHARP

#20 a = 4'bxx01; b = 4'b01xx; op\_code = 3'b110; //\*\*\*\*\*\*\*\*\*\*\*

#20 a = 4'b0xxx; b = 4'bxx11; op\_code = 3'b111; //\*\*\*\*\*\*\*\*\*\*\*

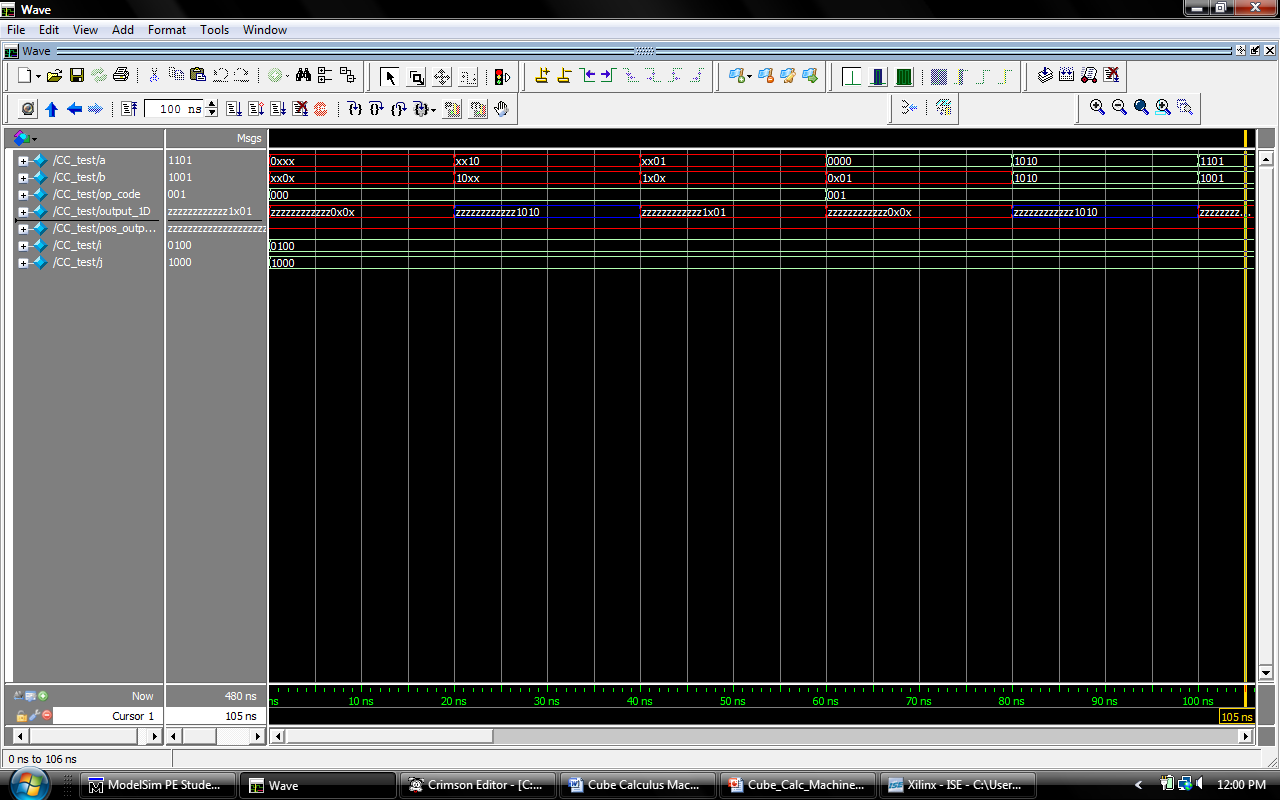
#20 a = 4'b101x; b = 4'bxx11; op\_code = 3'b111; // DISJOINT SHARP

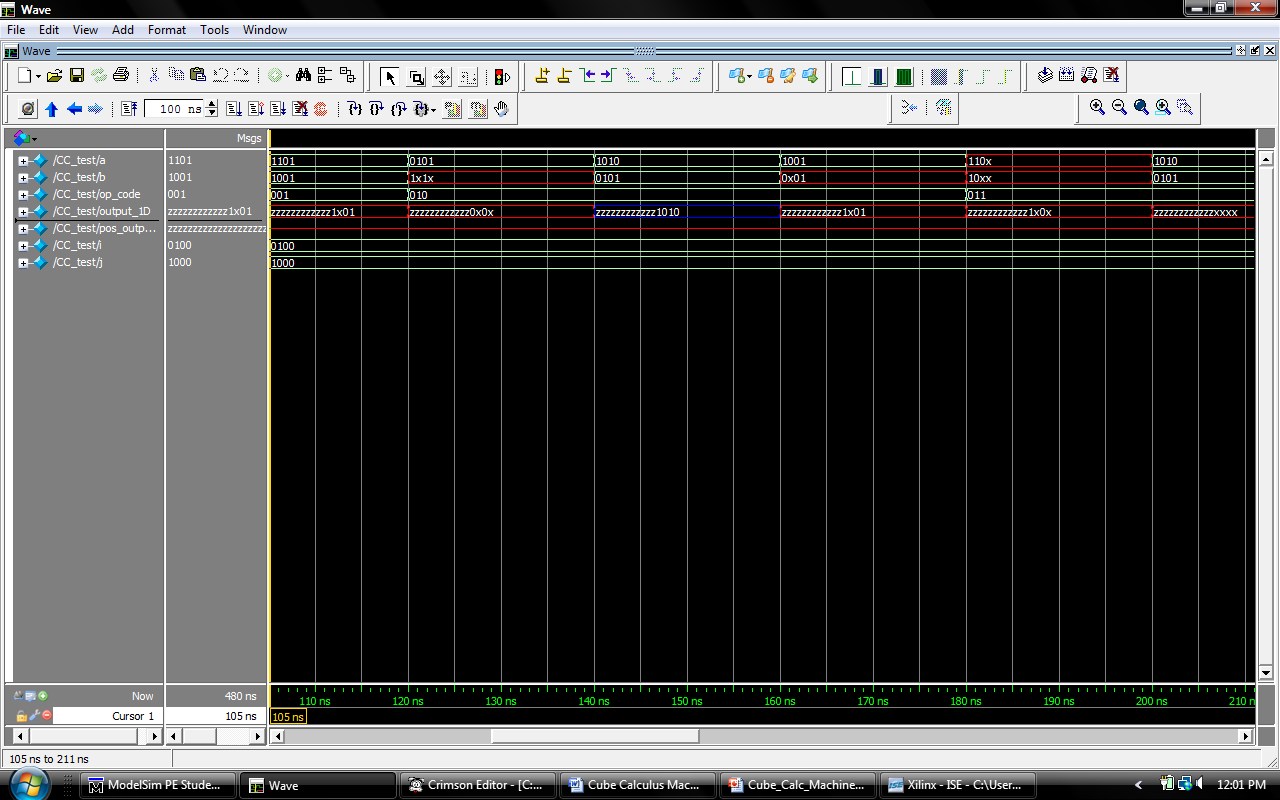
#20 a = 4'bxx01; b = 4'b01xx; op\_code = 3'b111; //\*\*\*\*\*\*\*\*\*\*\*

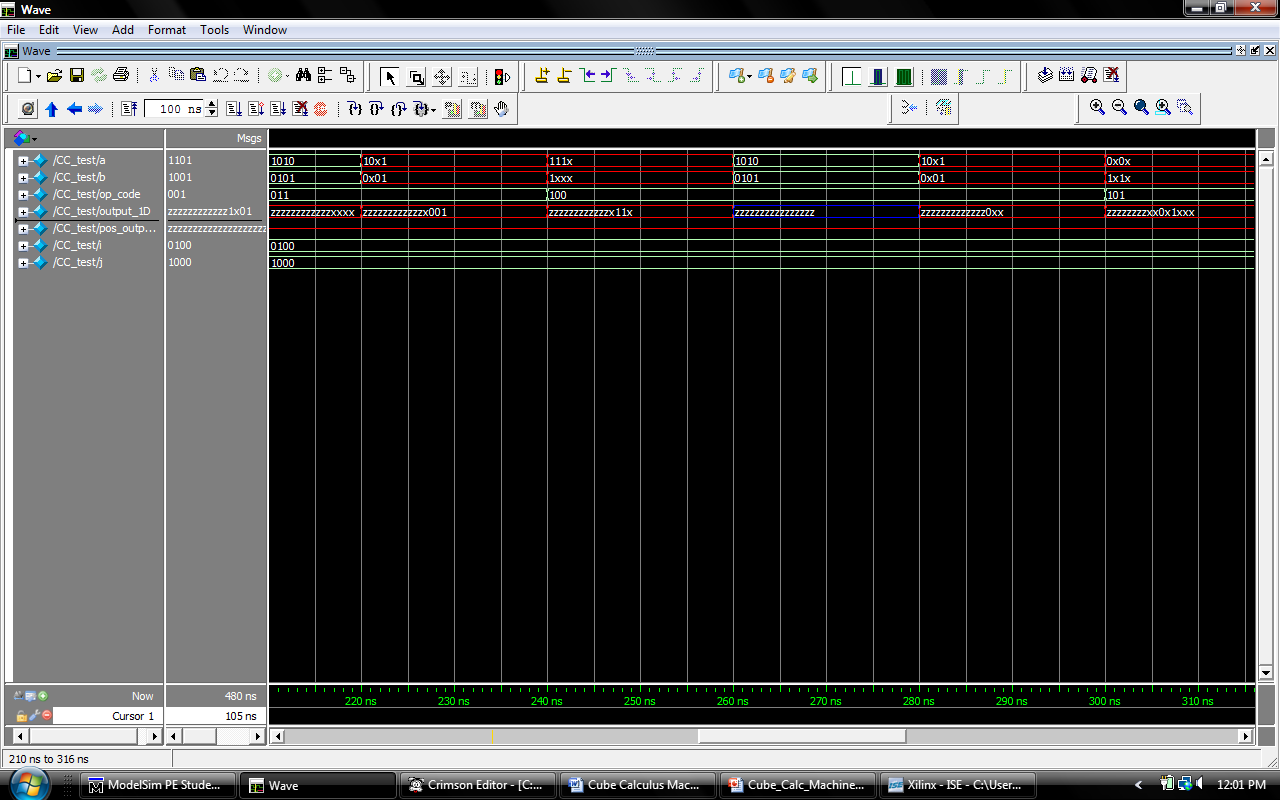
#20 $finish;

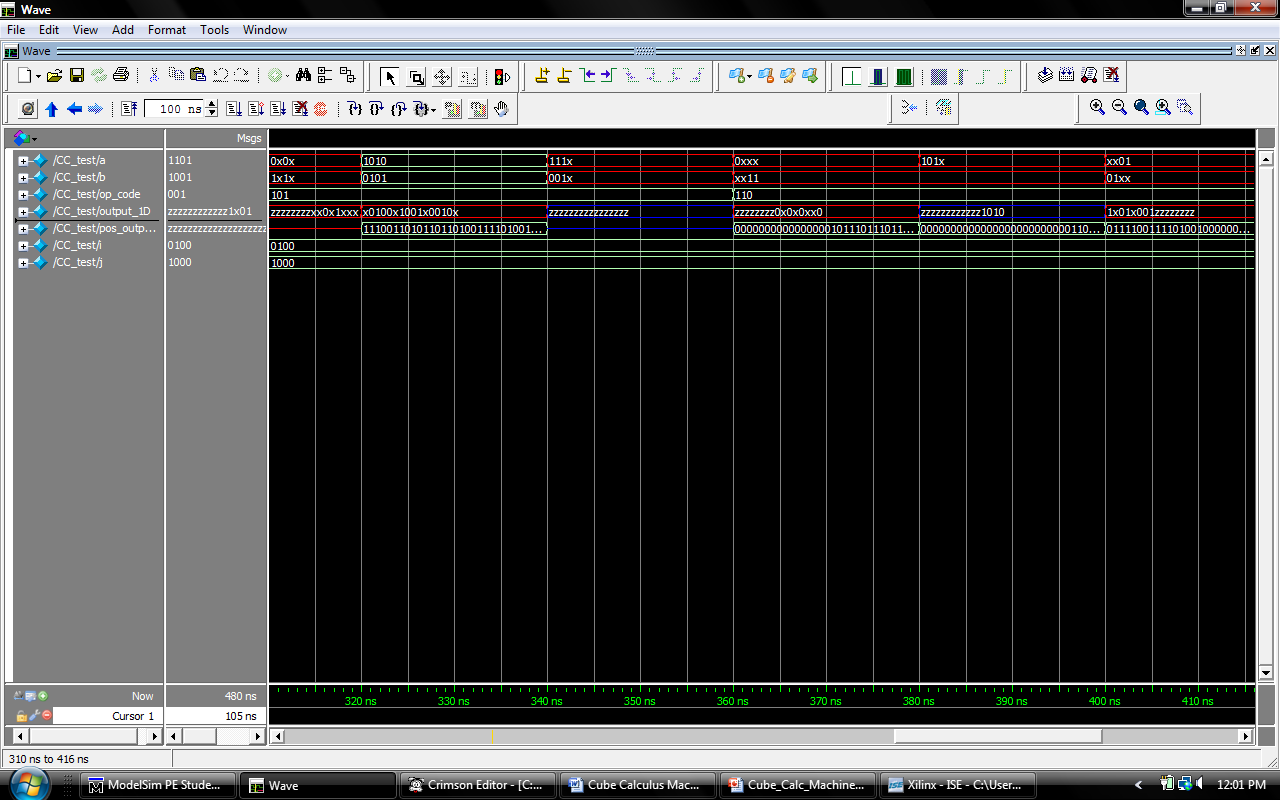
end

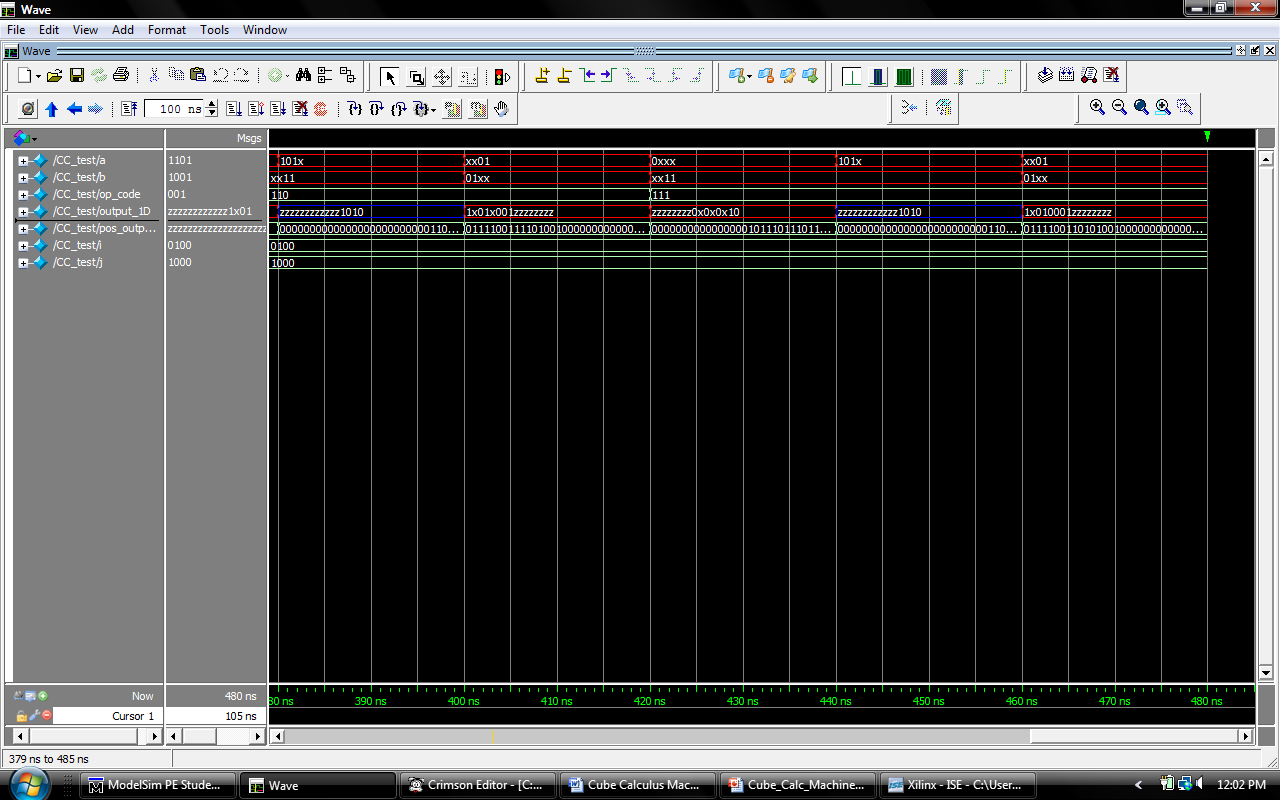
endmodule

****

****

****

****

****

**Verification of Intersection, supercube and crosslink**

Intersection and supercube are the simplest of cube calculus functions. Intersection represents the overlap of two cubes and is therefore the bit-wise AND of both cubes involved. The supercube function represents those parts of cubes that do not overlap and is therefore the bit-wise OR of both cubes involved. These two functions will only ever output one resultant cube each. This means that any resultant cubes beyond the first will be set to the high-impedance state to indicate a lack of result. When these two functions are not enabled they will output high-impedance so as to not disturb the shared result bus back to the arbiter.

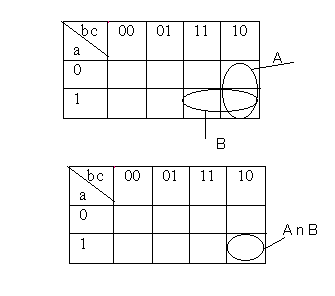


Figure 1 – Example of Intersection Inputs and Output

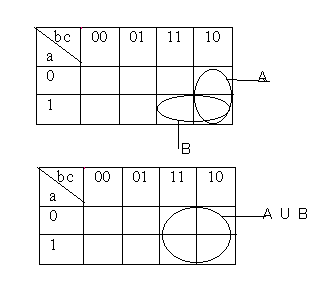


Figure 2 – Example of Supercube Inputs and Output

**Crosslink**

The crosslink function represents the shared distance between two cubes and the resultant cube(s) will be that shared space. The incoming cubes are first ANDed together to determine special variables. The special variable are those that end up as epsilon (00 in positional notation) after the AND is applied. If any of the variables become 01 or 10 (1 and 0 in positional notation), thus indicating overlap, then this function cannot be applied and the module will return a high-impedance state for all outputs indicating a lack of result. If there are not special variables, the don’t-cares will populate the output. If this function can be applied and there are special variables then for each special variable there will be one output. For each special variable, the output for that position will be don’t-care. The output ‘before’ the special variable will be the second cube. The output ‘after’ the special variable will be the first cube. Once the output calculations are complete, any unused resultant cube space will be set to high-impedance to indicate a lack of result. As will intersection and supercube, when this function is not enabled it will output high-impedance so it does not disturb the shared result bus back to the arbiter.

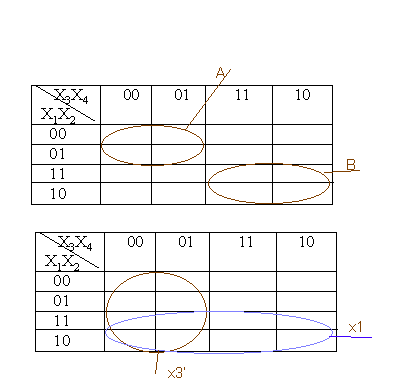


Figure 3 – Example of Crosslink Inputs and Outputs

**Edge-Node Test**

The edge-node test is a means to verify if the intersection and supercube functions are working properly. The goal is to provide input ON and OFF cubes (cubes of 1s and cubes of 0s respectively) and the module should output a table showing if there are links, or edges, between the ON cubes. First, the input ON cubes are supercubed with each other to produce intermediate cubes. Next these intermediate cubes are each intersected will all the OFF cubes. This produces (ON cubes) x (OFF cubes) number of results. Finally, each of these results is examined and if there is one result within a section of results (section being all results that correspond to one intermediate cube) that does not have epsilon (00 in positional notation) then there is an edge between the two ON cubes that were supercubed to create that intermediate cube. To illustrate this concept please refer to figure 5 below. It is a screenshot of the actual displayed output from running this test. The edge-node table at the end shows the edges between the ON cubes. A one means there is an edge and a zero means there is not.

Figure 4 – Karnaugh Map of ON and OFF cubes

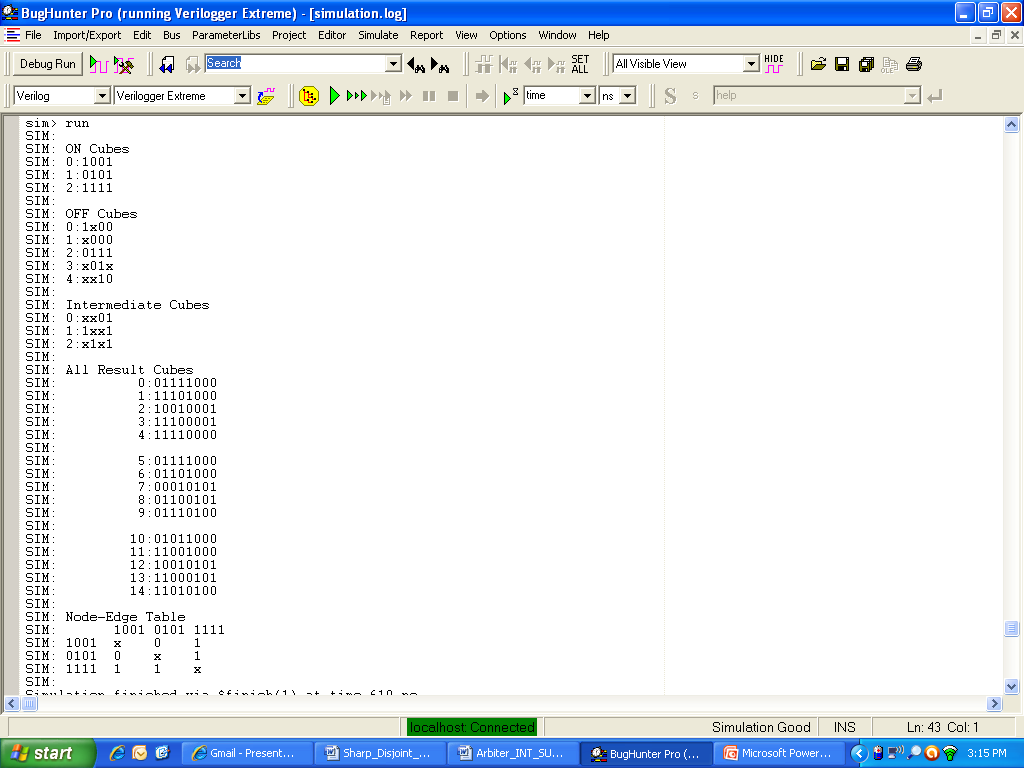


Figure 5 – Screenshot of Edge-Node Test Results

**Testing**

**// initialize the on and off cubes for the test vector**

ON\_cubes[0] = 4'b1001;

ON\_cubes[1] = 4'b0101;

ON\_cubes[2] = 4'b1111;

OFF\_cubes[0] = 4'b1x00;

OFF\_cubes[1] = 4'bx000;

OFF\_cubes[2] = 4'b0111;

OFF\_cubes[3] = 4'bx01x;

OFF\_cubes[4] = 4'bxx10;

// calculate intermediate cubes with supercube

op\_code = 3'b001;

a = ON\_cubes[0];

b = ON\_cubes[1];

intermediate[0] = out\_1; // store intermediate 0

a = ON\_cubes[0];

b = ON\_cubes[2];

intermediate[1] = out\_1; // store intermediate 1

a = ON\_cubes[1];

b = ON\_cubes[2];

intermediate[2] = out\_1; // store intermediate 2

**// calculate resultant cubes with intersection**

op\_code = 3'b000;

for(k=0; k<3; k=k+1)

begin

for(i=0; i<5; i=i+1)

begin

a = intermediate[k];

b = OFF\_cubes[i];

RESULT\_cubes[k\*5+i] = out\_pos\_1;

end

end

**// determine if edges exists**

for(k=0; k<3; k=k+1) // ON\_cubes

begin

for(i=0; i<5; i=i+1) // OFF\_cubes

begin

result\_edges[k] = 1'b1; // reset edge result for each new set of cube tests

for(j=0; j<8; j=j+2) // 8-bit positional notation cubes

begin

if({RESULT\_cubes[k\*5+i][j+1],RESULT\_cubes[k\*5+i][j]} != 2'b00) // epsilon

result\_edges[k] = result\_edges[k] & 1'b1;

else

result\_edges[k] = 1'b0;

end

if(result\_edges[k] == 1'b1) //if one of the OFF cubes has no epsilon, edge found

begin j=8; i=5; end // break loops

end

end

**Verification of Sharp and Disjoined\_sharp**

**Sharp**

The sharp operation is a sequential cube calculus operation. This result of this operation will be the first cube without second cube. As an example, let us analyze the result of the equation **XXX1 #basic X11X (Figure1).**

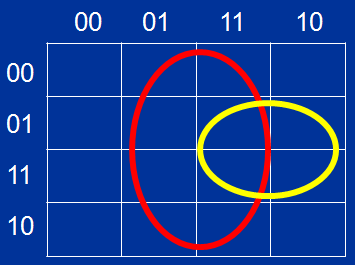


Figure 1 Cube#1 = XXX1, Cube#2 = X11X

Graphically, it is trivial to determine the results from the equation. The cube XXX1 will remain, but all the minterms covered by X11X will not be included. The graphical result of this operation is showed in Figure2. The cover the area of the first cube without the 2nd cube two different cubes must be used and there will be overlap.



Figure 2 Result of CubeA # CubeB

Determining the result in code is not as simple when working with code. To do this, we follow this algorithm:

* Determine special variables by finding all positions in which .
* For each the special variables:
  + Apply the operation to the special variable
  + For all other bits, use the result .

**Disjoint Sharp**

The Disjoint sharp operation is a sequential cube calculus operation. This result of this operation is the same as the sharp operation except for the fact that the resulting cubes will be disjoint.. As an example, let us analyze the result of the equation **XXX1 #dbasic X11X (Figure1).**

Figure 3 shows the results of the disjoint operation.

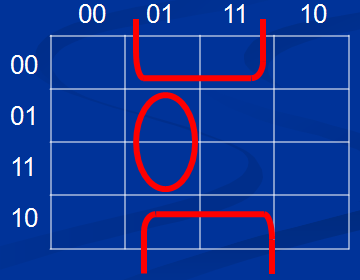


Figure 3 Result of CubeA D# CubeB

To replicate this operation in code, the following algorithm must be followed. This algorithm is very similar to the sharp algorithm, but differs for all the bits after the special variable.

* Determine special variables by finding all positions in which .
* For each the special variables:
  + Apply the operation to the special variable
  + For all the bits after the special variable, apply the operation .
  + For all the bits before the special variable, use the result .

**Verification of Sharp/Disjoint Sharp**

Verifying that each of these operations works can be done using the same algorithm. For simplicity, all further equations will be using the disjoint sharp operation, but it should be understood that sharp will produce the same result with a different set of cubes.

The algorithm introduced in this section simply results in the compliment of the original function. This is simply down by using the high level equation:

As a graphical example, consider the cubes shown in figure 4.

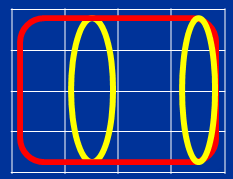


Figure 4 Original Cubes to Compliment

The yellow cubes is the function F(ABCD) and the red is clearly a logical ‘1’. It should be apparent that by taking the disjoint sharp operation the result will be the the graph shown in figure5.

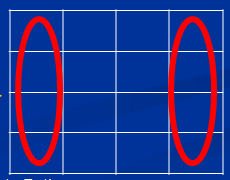


Figure 5 Results which shows the compliment

In code, we must be much more methodical in the algorithm and therefore we will use the following equation:

The order in which the cubes are applied is irrelevant as it will produce the desired result.

To fully understand this algorithm, we will review each step of the algorithm in detail. Let us assume the original equation is the set of cubes :{X100,011X,1X10}. The graphical representation of these cubes is in figure 6.

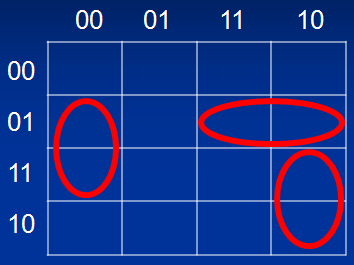


Figure 6 Graphical representation of a function F(ABCD)

The first step in this algorithm is to pick a random cube and apply either sharp or disjoint operation to XXXX (logical ‘1’). The graphical representation is in figure 7. The red cube which is filled will represent the current cube under test. In this case, the cube X100 was chosen.

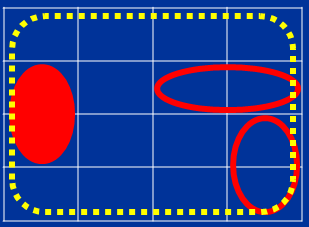


Figure 7 Step #1 in the algorithm. XXXX # C1

The result of this operation is shown in figure 8. The dotted yellow cubes are the result of step#1.

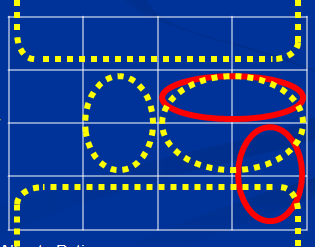


Figure 8 Result of 1 # X100

Step #2 requires that any cube left from the original equation be applied the sharp operation with every cube which resulted from the operation in Step #1. In this case, we will have 3 different equations in step #2:

X0XX #011X

X11X # 011X

X101 # 011X

The graphical representation is shown in figure 9.

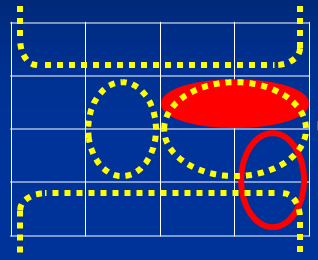


Figure 9 Step#2 equations

Two of the equations in step #2 clearly show that there is no interception and therefore the will not change. Figure10 shows the results of Step #2.

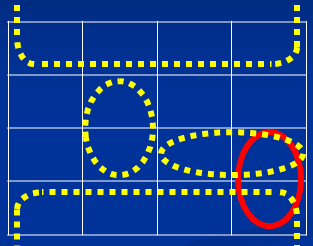


Figure 10 The result of step #2

The final step in this algorithm is quite clear as there is only on cube left in the original function. The result of this step will be the compliment of the original function. Figure 11 shows the final result.

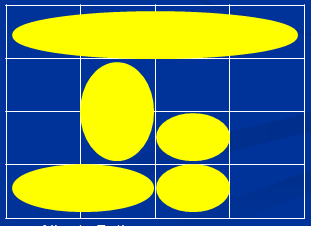


Figure 11 Final result

In order to verify the code written to handle this function, we will input the function in the previous example. Below is the debug output which shows that the code is working as expected.

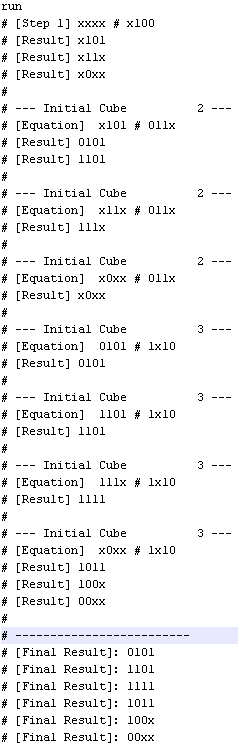


Figure 12 Output from the verilog code to verify the function disjoint sharp

Figure 12 shows the result of the algorithm written in verilog matches the expected output from the example above.

**Synthesis:** The machine was synthesized using a Spartan 3 FPGA

CUBE2 Project Status

Project File:

cube2.ise

Current State:

Programming File Generated

Module Name:

CC\_Arbiter

Errors:

No Errors

Target Device:

xc3s1000-4ft256

Warnings:

26 Warnings

Product Version:

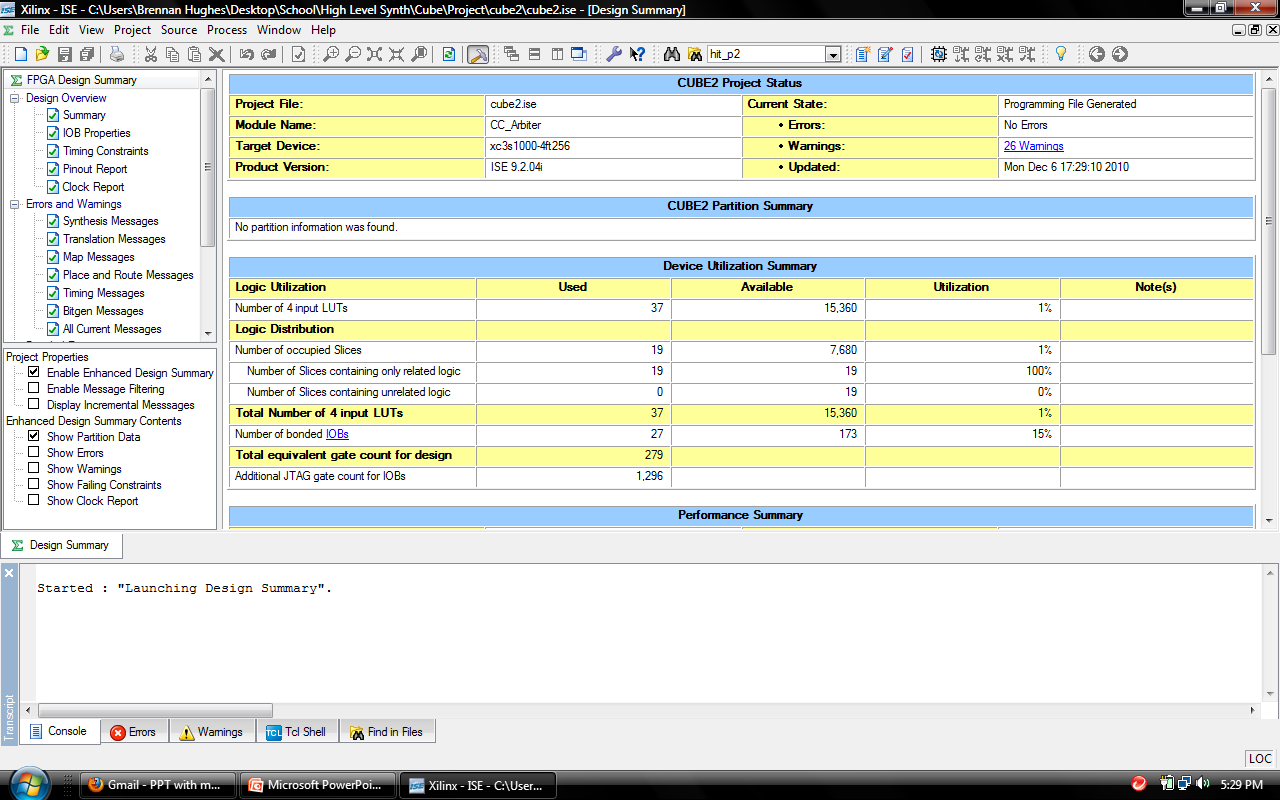
ISE 9.2.04i

Updated:

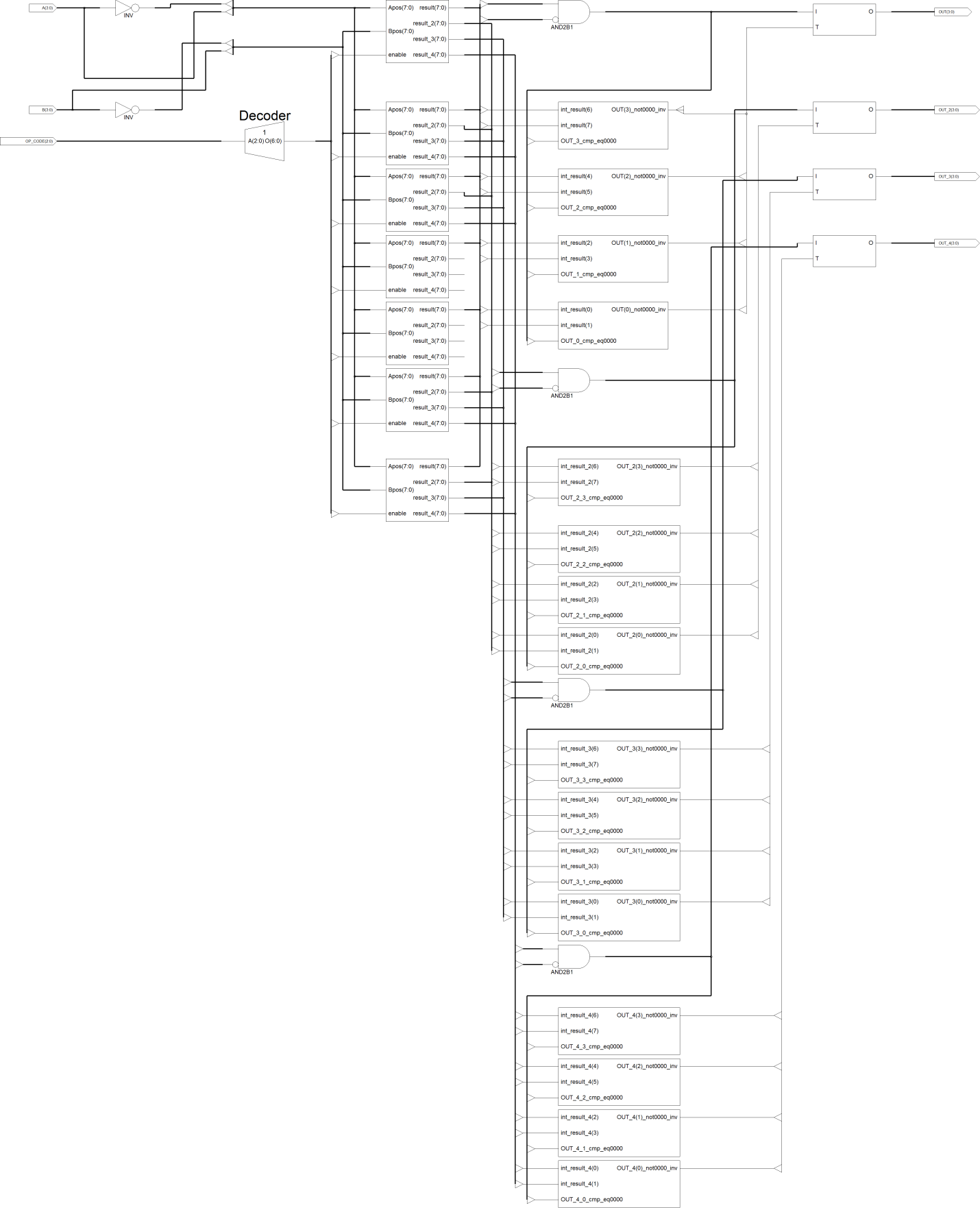
Sun Dec 12 11:29:37 2010

CUBE2 Partition Summary

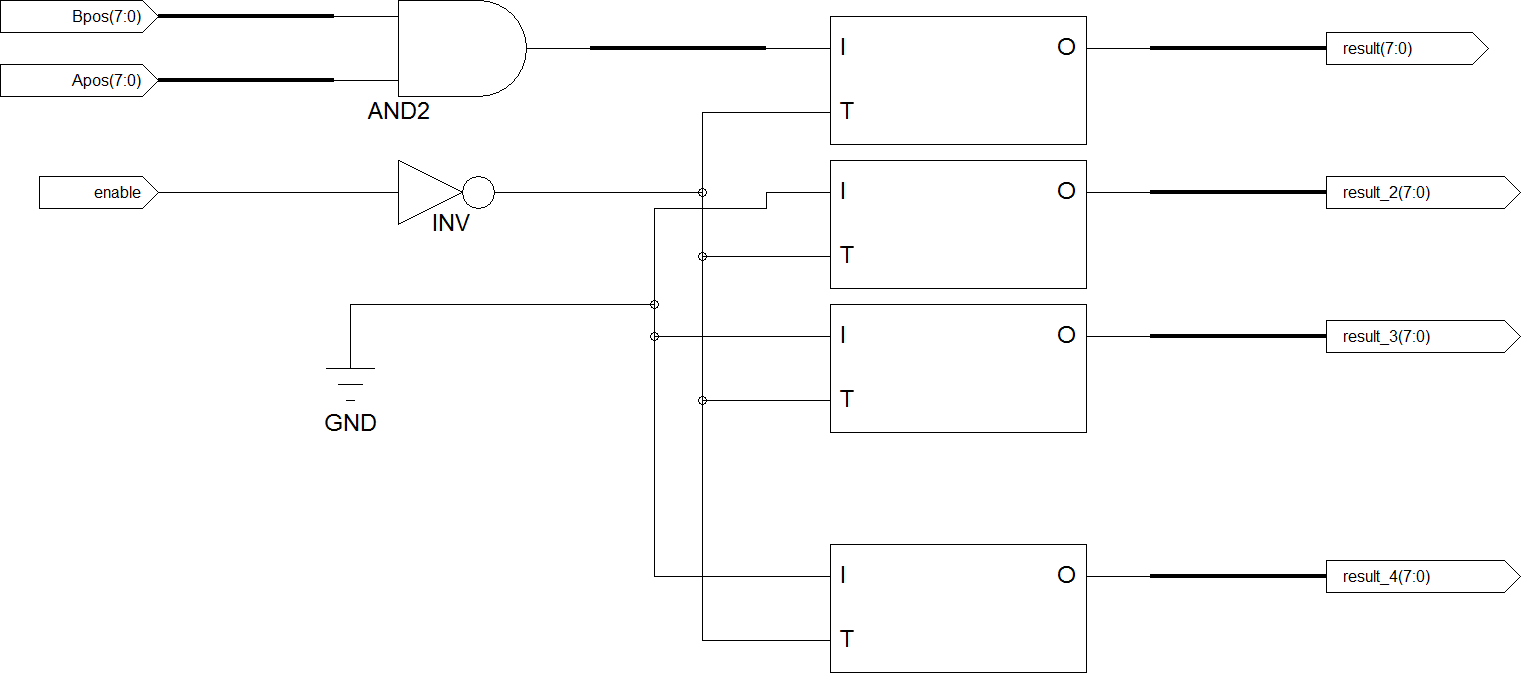
No partition information was found.



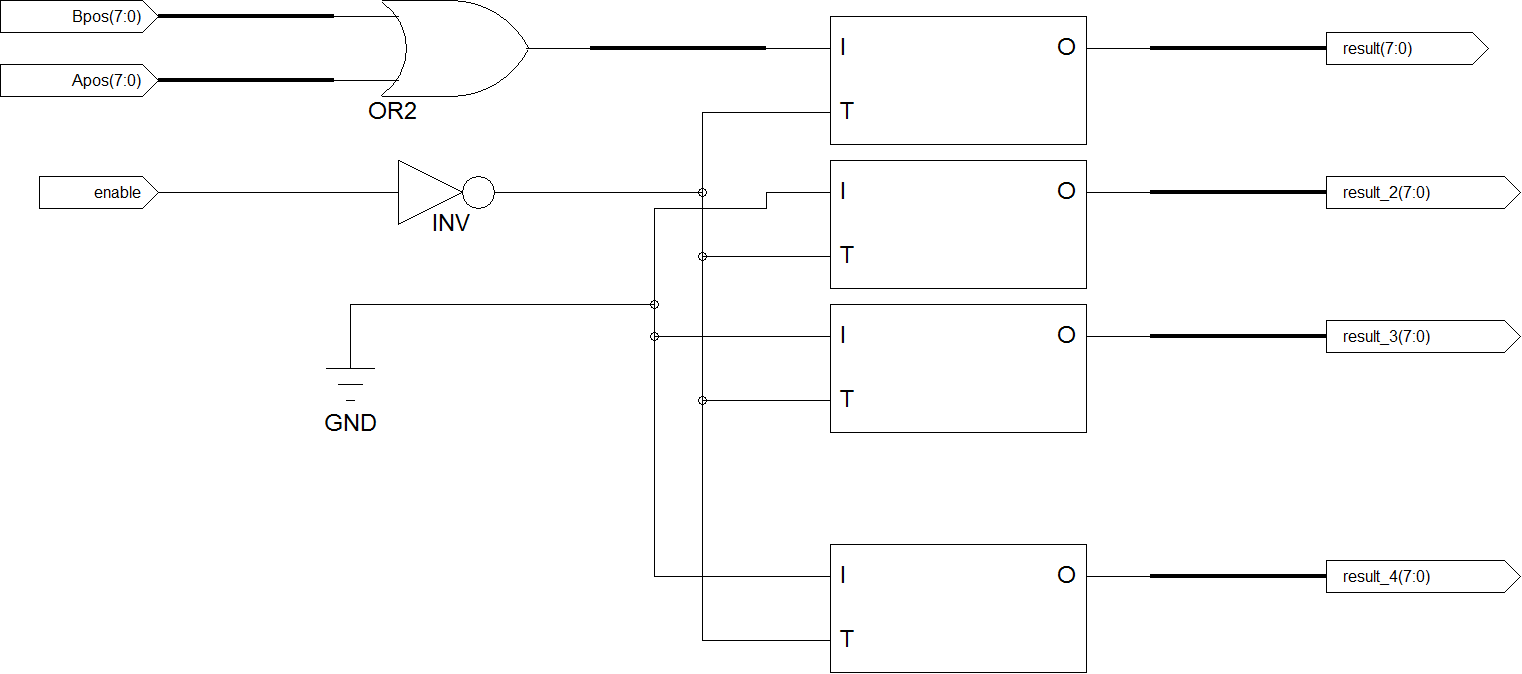
**Machine (Arbiter)**

****

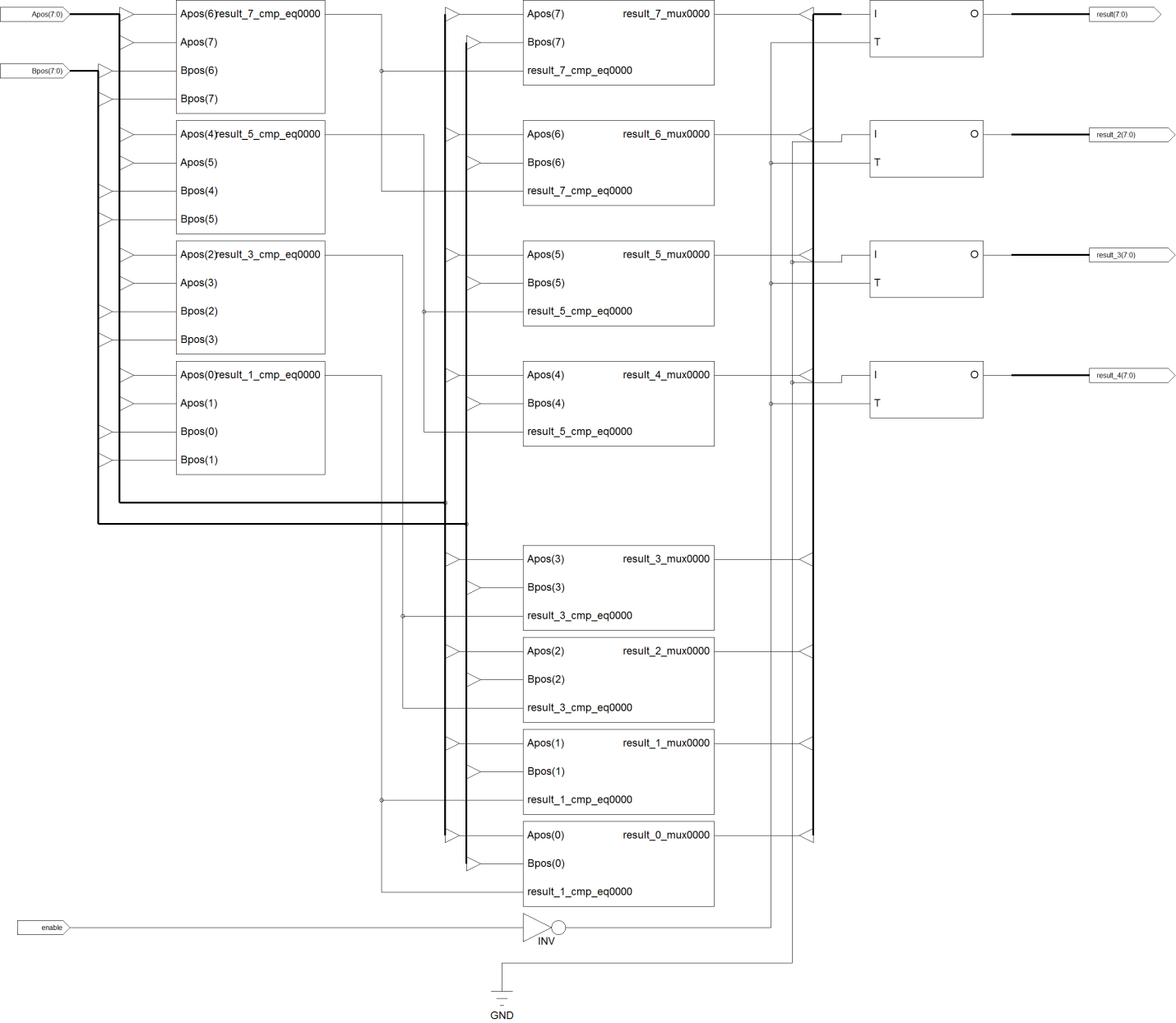
**Intersection**

****

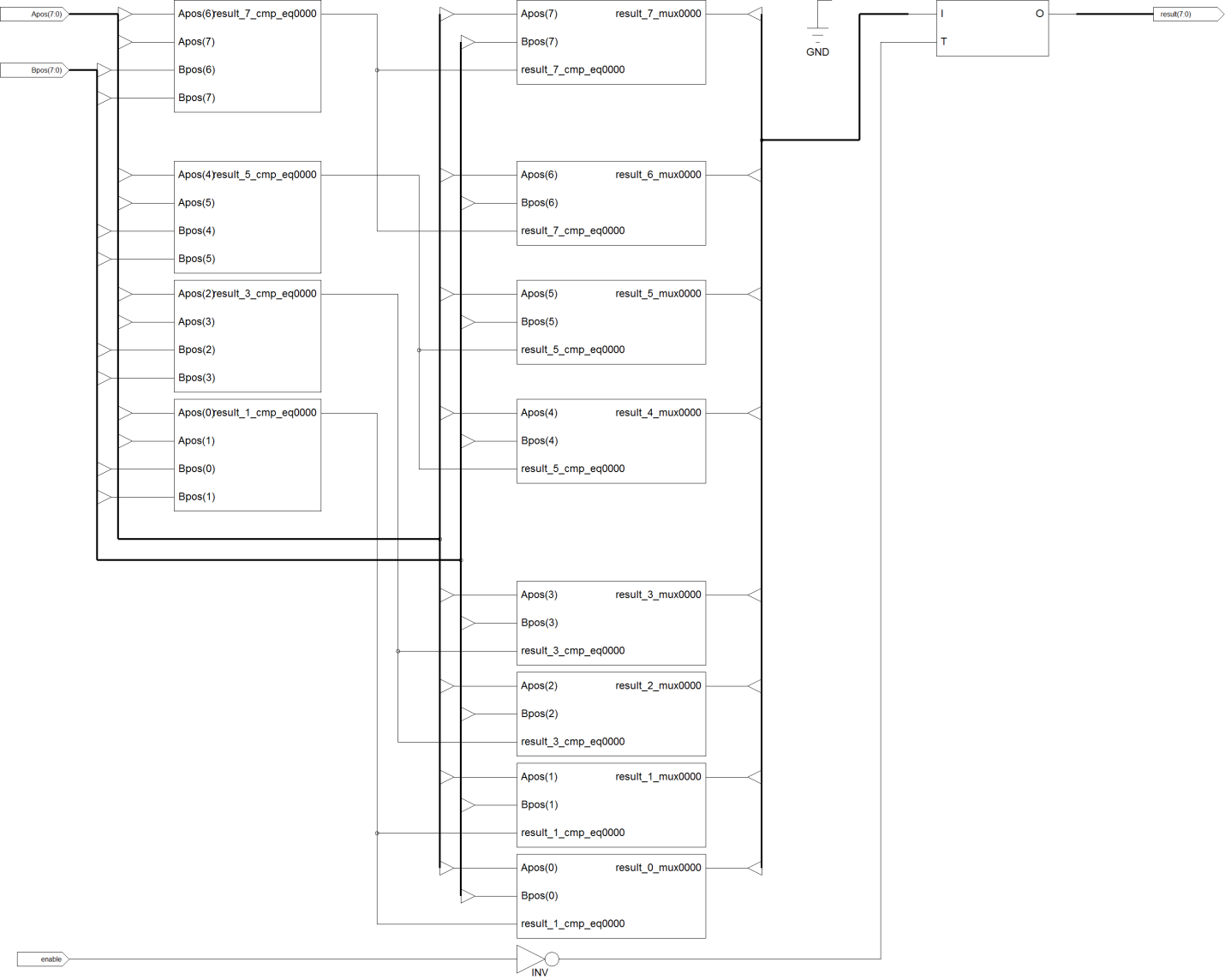
**Supercube**

****

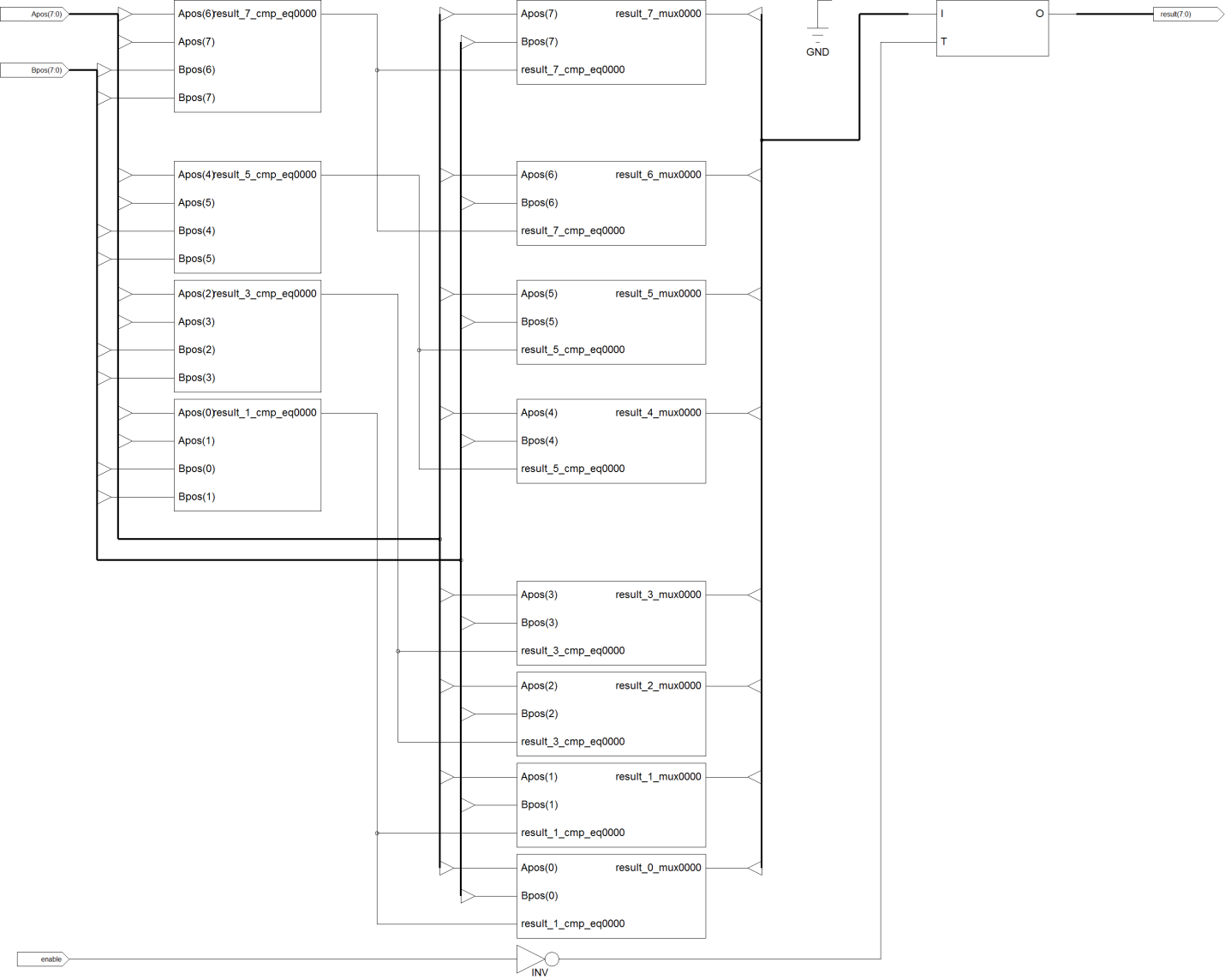
**Prime**

****

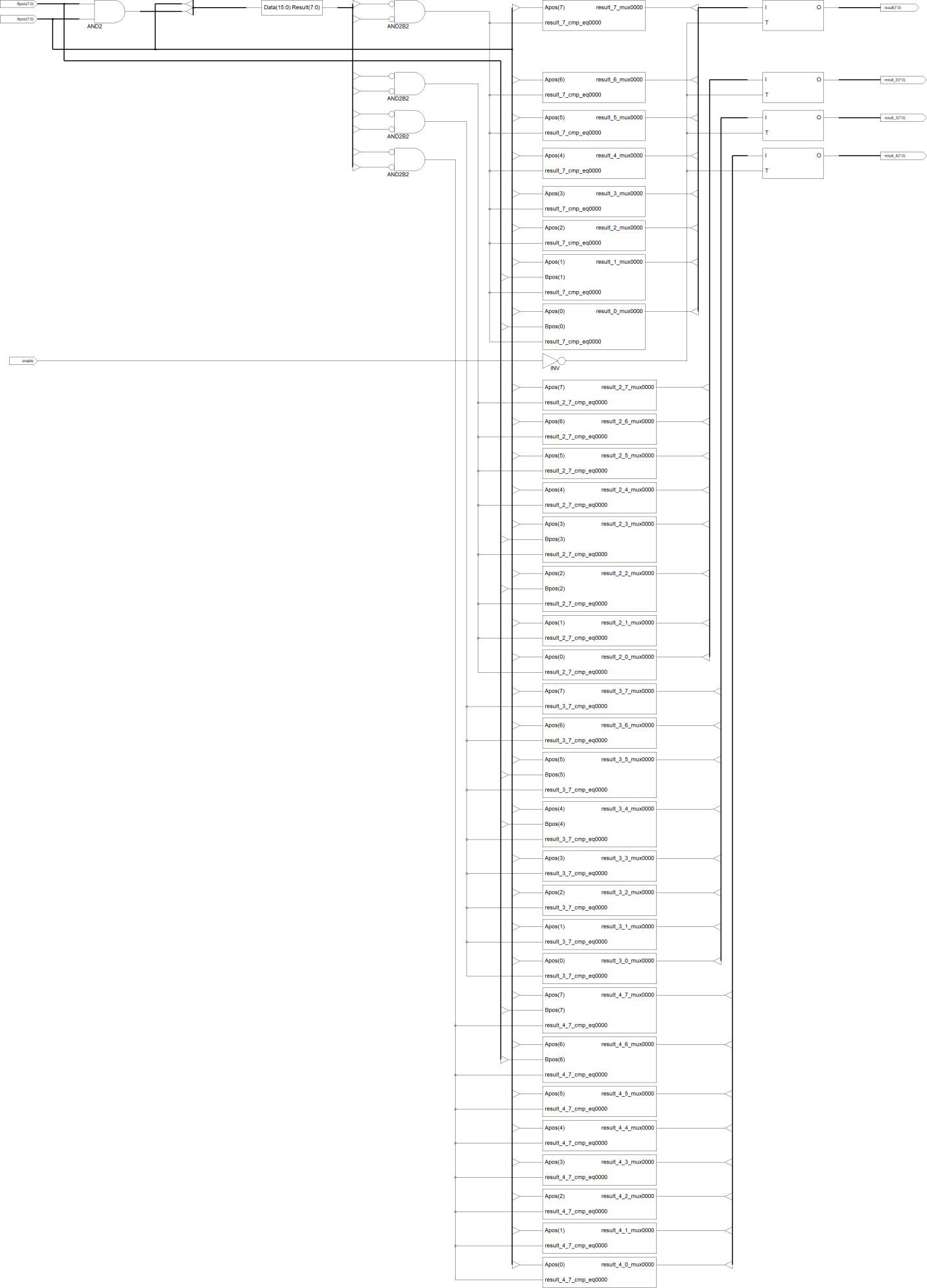
**Consensus**

****

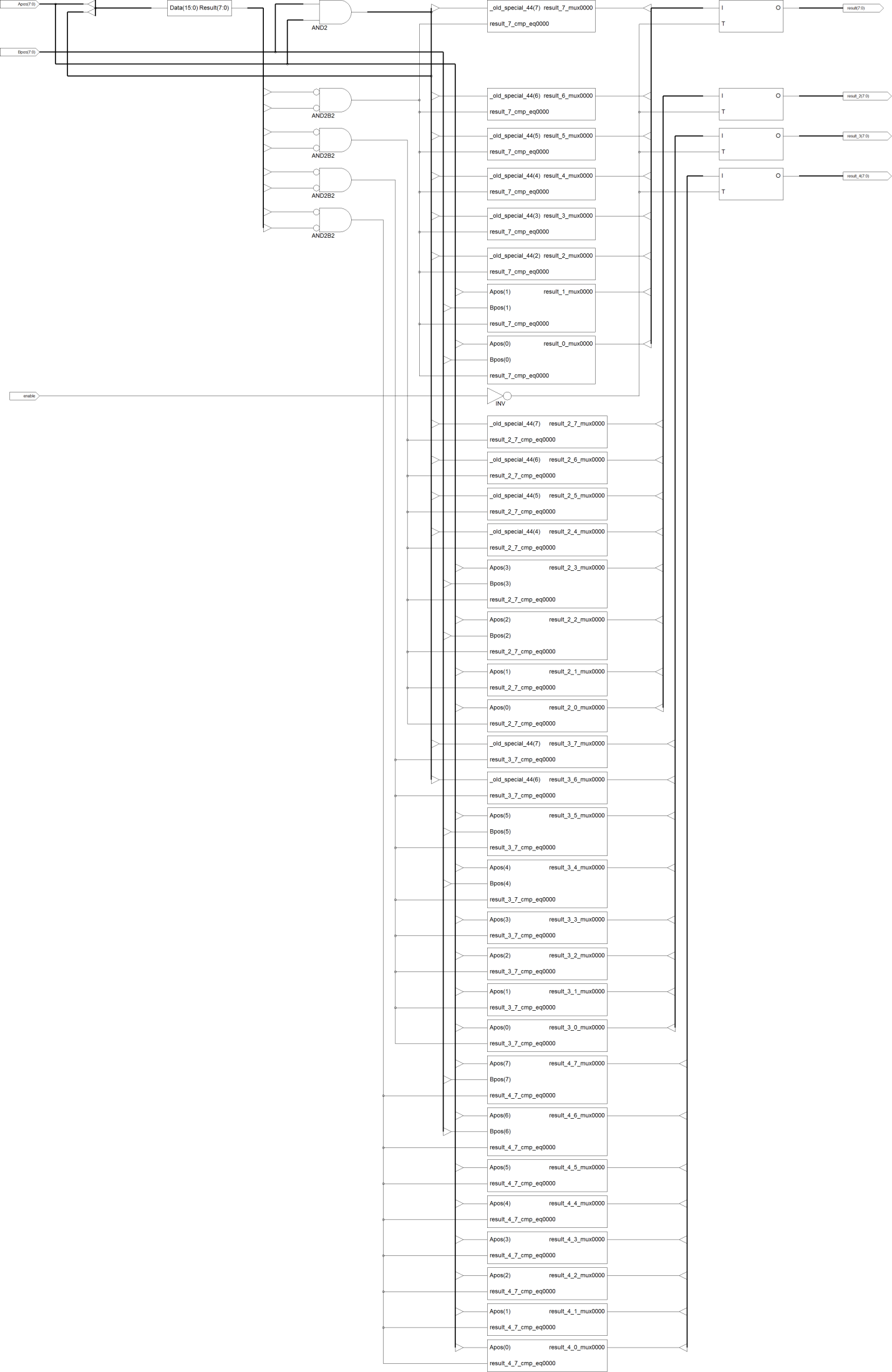
**Cofactor**

****

**Sharp**

****

**Disjoined\_sharp**

****