Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Systems-on-Chip Design
Course Number:	COE 838
Semester/Year (e.g.F2016)	W2022

Instructor:	Dr. Gul Khan

Assignment/Lab Number:	2a
Assignment/Lab Title:	The Multiplier

Submission Date:	Thursdsay, February 3rd, 2022
Due Date:	Wednesday, February 2nd, 2022

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Pano	Toni	500822828	01	T. P.

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a “0” on the work, an “F” in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Table of Contents

List of Figures

List of Tables

1. Carry Propagate Adder (CPA)

a. Manual Calculation of Test Cases

Table 1. Expected output for each test case of CPA unit.

A	B	Carry In	Carry Out	Sum
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

b. Waveforms of Test Cases

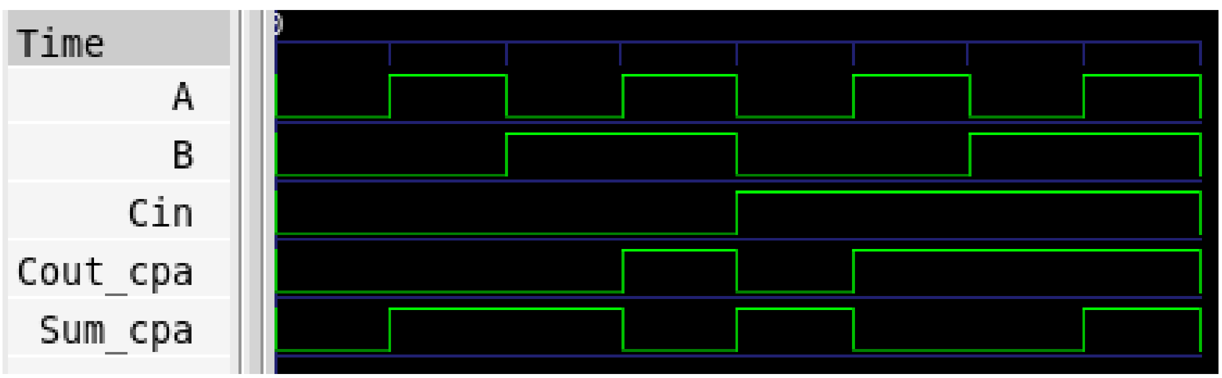

Figure 1. Waveforms of test cases for CPA unit.

2. Carry Save Adder (CSA)

a. Manual Calculation of Test Cases

Table 2. Expected output for each test case of CSA unit.

A	B	Carry In	Sum In	Carry Out	Sum Out
0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	1
0	0	1	0	0	1
1	0	1	0	0	1
0	1	1	0	0	1
1	1	1	0	1	0
0	0	0	1	0	1
1	0	0	1	0	1
0	1	0	1	0	1
1	1	0	1	1	0
0	0	1	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	1	1	1	1

b. Waveforms of Test Cases

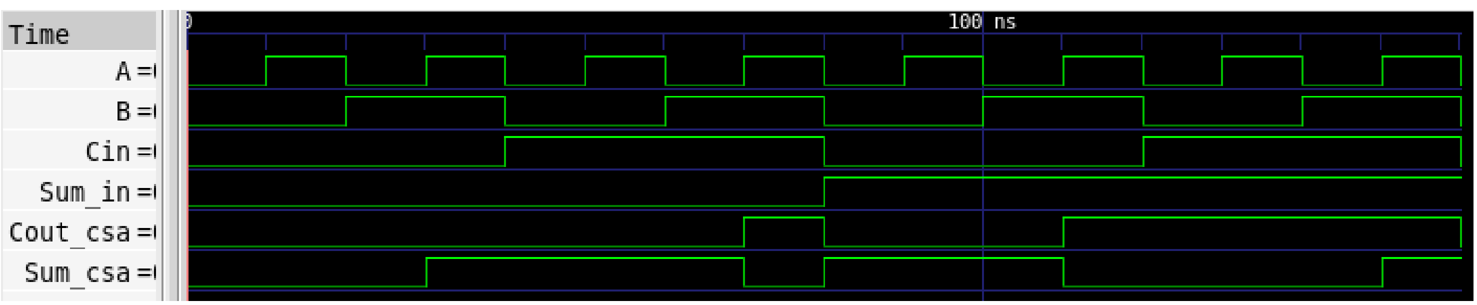

Figure 2. Waveforms of test cases for CSA unit.

3. Multiplier (mult)

a. Manual Calculation of Test Cases

Table 3. Expected output for test cases of multipler unit.

A	B	P
0	0	0
0	1	0
0	27	0
0	254	0
0	255	0
1	0	0
1	1	1
1	27	27
1	254	254
1	255	255
27	0	0
27	1	27
27	27	729
27	254	6858
27	255	6885
254	0	0
254	1	254
254	27	6858
254	254	64516
254	255	64770
255	0	0
255	1	255
255	27	6885
255	254	64770
255	255	65025

b. Waveforms of Test Cases

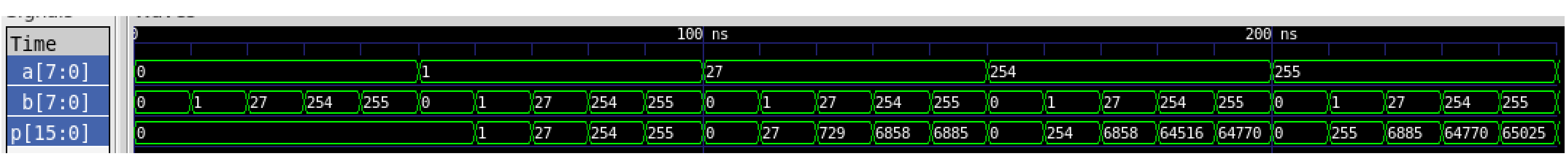

Figure 3. Wavefroms of test cases for multiplier unit.

Appendix A: Main File for Testing Multiplier Unit

sc_main4.cpp

```
#include <iostream>
#include <math.h>
#include <systemc.h>

//import modules
#include "cpa.h"
#include "csa.h"
#include "mult.h"

void test_cpa_csa();
void test_mult();

int sc_main(int argc, char* argv[]){
	//set up multipler unit
	const int bits = 8; //min size is 2 bits
	sc_signal< sc_uint<bits> > a("a");
	sc_signal< sc_uint<bits> > b("b");
	sc_signal< sc_uint<2*bits> > p("p");

	mult<bits> m("mult");
	m.a(a);
	m.b(b);
	m.p(p);

	//set up waveform trace file
	sc_trace_file *tf_mult;
	tf_mult = sc_create_vcd_trace_file("trace_file");	//create trace file called
"trace_file_mult.vcd"
	tf_mult -> set_time_unit(1, SC_NS); //set unit time of traces to 1 ns
	sc_trace(tf_mult, a, "a");
	sc_trace(tf_mult, b, "b");
	sc_trace(tf_mult, p, "p");

	//simulation of test cases
	int cases = 5;
	int arr[5] = {0,1,27,254,255};
	int i, j;
	for(i = 0; i < cases; i++){
		for(j = 0; j < cases; j++){
			a.write(arr[i]);
			b.write(arr[j]);
			sc_start(10, SC_NS);
			cout << "a * b = p: " << a.read() << " * " << b.read() << " = " <<
p.read() << endl;
		}
	}

	//pass end with 10 more seconds, otherwise last test case is hard to see in waveform file
	a.write(0);
	b.write(0);
	sc_start(10, SC_NS);
	sc_close_vcd_trace_file(tf_mult);
}
```

Makefile

```
#
CC=/usr/bin/g++ -std=c++11
ARCH := $(shell arch)
SYSTEMC_HOME=/usr/local/SystemC-2.3.0

# 64bit or 32bit libaries to link to
LINUXLIB := $(shell if [ ${ARCH} = "i686" ]; \
                    then \
                         echo lib-linux; \
                    else \
                         echo lib-linux64; \
                    fi)

INCLUDES = -I$(SYSTEMC_HOME)/include -I.

LIBRARIES = -L. -L$(SYSTEMC_HOME)/$(LINUXLIB) -lsystemc -lm

RPATH = -Wl,-rpath=$(SYSTEMC_HOME)/$(LINUXLIB)

PROGRAM = test.x
SRCS    = cpa.h cpa.cpp csa.h csa.cpp splitter.h joiner.h mult.h sc_main4.cpp
OBJS    = cpa.o csa.o sc_main4.o

all : $(PROGRAM)

$(OBJS) : $(SRCS)
	$(CC) $(INCLUDES) -c $(SRCS)

$(PROGRAM) : $(OBJS)
	$(CC) $(INCLUDES) $(LIBRARIES) $(RPATH) -o $(PROGRAM) $(OBJS)

clean:
	@rm -f $(OBJS) $(PROGRAM) *.cpp~ *.h~
```

Appendix B: Main File for Testing CSA and CPA Units

sc_main.cpp

```
#include <iostream>
#include <math.h>
#include <systemc.h>

//import modules
#include "cpa.h"
#include "csa.h"

const int bits = 3; //min size is 2 bits

int read_sig_arr(sc_signal<bool> sig[bits]){
	int i;
	int sum = 0;
	cout << "N = ";
	for(i = bits-1; i >= 0; i--){
		cout << sig[i].read();
		sum += (sig[i].read() << i);
	}
	cout << " = " << sum << endl;

	return sum;
}

void write_sig_arr(sc_signal<bool> sig[bits], int N){
	int i;
	//cout << "N = " << N << " = ";
	for(i = bits-1; i >= 0; i--){
		//cout << ((N & (1 << i)) > 0);
		sig[i].write((N & (1 << i)) > 0);
	}
	//cout << endl;
}

void replace_char(char * str, int pos, char c){
	str[pos] = c;
}

int sc_main(int argc, char* argv[]){

	//signals
	sc_signal<bool> A[bits];
	sc_signal<bool> B[bits];
	sc_signal<bool> P [2*bits];
	sc_signal<bool> C_out;

	//initialize carry save adder modules (CSA)
	csa* csa_arr [bits][bits];
	sc_signal<bool> csa_Co [bits][bits];
	sc_signal<bool> csa_So [bits][bits];
	int i,j;
	char csa_name [9] = "csa_A#B#";
	cout << "1" << endl;

	//connect A B inputs and So Co outputs for each CSA unit
	for(i = 0; i < bits; i++){
		for(j = 0; j < bits; j++){
			replace_char(csa_name, 5, (i+'0'));
			replace_char(csa_name, 7, (j+'0'));
			csa_arr[i][j] = new csa(csa_name);
			csa_arr[i][j] -> A(A[i]);
			csa_arr[i][j] -> B(B[j]);
			if(i > 0){
				csa_arr[i][j] -> So(csa_So[i][j]);
			}
			csa_arr[i][j] -> Co(csa_Co[i][j]);
		}
	}
```

```
	cout << "2" << endl;

	//S = 0 at edge of matrix B[0]
	sc_signal<bool> csa_S_B0 [bits];
	for(i = 0; i < bits; i++){
		csa_arr[i][0] -> S(csa_S_B0[i]);
		csa_S_B0[i].write(0);
	}
	cout << "3" << endl;

	//S = 0 at edge of matrix A[MSB]
	sc_signal<bool> csa_S_AMSB[bits-1];
	for(j = 1; j < bits; j++){
		csa_arr[bits-1][j] -> S(csa_S_AMSB[j-1]);
		csa_S_AMSB[j-1].write(0);
	}
	cout << "4" << endl;

	//S = S of previous B bit and next A bit: for B[1] to B[MSB] and A[0] to A[MSB-1]
	for(i = 0; i < bits-1; i++){
		for(j = 1; j < bits; j++){
			csa_arr[i][j] -> S(csa_So[i+1][j-1]);
		}
	}
	cout << "5" << endl;

	//C = 0 at edge of matrix B[0]
	sc_signal<bool> csa_C_B0 [bits];
	for(i = 0; i < bits; i++){
		csa_arr[i][0] -> C(csa_C_B0[i]);
		csa_C_B0[i].write(0);
	}
	cout << "6" << endl;

	//C = (C of previous B bit) in each A row
	for(i = 0; i < bits; i++){
		for(j = 1; j < bits; j++){
			csa_arr[i][j] -> C(csa_Co[i][j-1]);
		}
	}
	cout << "7" << endl;

	//connect So of each B bit in row A[0] to P[0:bits-1]
	for(j = 0; j < bits; j++){
		csa_arr[0][j] -> So(P[j]);
	}
	cout << "8" << endl;

	//initialize carry propagate adder modules (CPA) for each A bit
	cpa* cpa_arr [bits];
	sc_signal<bool> cpa_Co [bits];
	char cpa_name [7] = "cpa_A#";
	for(i = 0; i < bits; i++){
		replace_char(cpa_name, 5, (i+'0'));
		cpa_arr[i] = new cpa(cpa_name);
		if(i < bits-1){
			cpa_arr[i] -> Co(cpa_Co[i]);
			cpa_arr[i] -> B(csa_Co[i][bits-1]);	//connect B of each CPA to Co of each
CSA (in B[MSB] row for each A bit)
		}
		cpa_arr[i] -> So(P[i+bits]);		//connect So of each CPA to each P bit
	}
	cout << "9" << endl;

	//connect Co of prev CPA to C of next CPA
	//connect So of next CSA (in B[MSB] row) to A of prev CPA for CPA 0 and CPA 1
	for(i = 0; i < bits-1; i++){
		if(i < bits-2){
			cpa_arr[i+1] -> C(cpa_Co[i]); //connect
		}
		cpa_arr[i] -> A(csa_So[i+i][bits-1]);
```

```
	}
	cout << "10" << endl;

	//set C of first CPA to 0
	sc_signal<bool> cpa_C0;
	cpa_arr[0] -> C(cpa_C0);
	cpa_C0.write(0);
	cout << "11" << endl;

	//for last CPA, set A to Co of last CSA, set B to Co of prev CPA, set C to 0, and set Co
to C_out
	cpa_arr[bits-1] -> A(csa_Co[bits-1][bits-1]);
	cpa_arr[bits-1] -> B(cpa_Co[bits-2]);
	sc_signal<bool> cpa_CMSB;
	cpa_arr[bits-1] -> C(cpa_CMSB);
	cpa_CMSB.write(0);
	cpa_arr[bits-1] -> Co(C_out);
	cout << "12" << endl;

	//set up waveform trace file
	sc_trace_file *tf;
	tf = sc_create_vcd_trace_file("trace_file"); //create trace file called "trace_file.vcd"
	tf -> set_time_unit(1, SC_NS); //set unit time of traces to 1 ns
	char trace_name [5] = "A[#]";
	for(i = 0; i < bits; i++){
		replace_char(trace_name, 2, (i+'0'));
		sc_trace(tf, A[i], "A"+i);
	}
	trace_name[0] = 'B';
	for(i = 0; i < bits; i++){
		replace_char(trace_name, 2, (i+'0'));
		sc_trace(tf, B[i], "B"+i);
	}
	trace_name[0] = 'P';
	for(i = 0; i < 2*bits; i++){
		replace_char(trace_name, 2, (i+'0'));
		sc_trace(tf, P[i], "P"+i);
	}
	sc_trace(tf, C_out, "C_out");
	cout << "13" << endl;

	//simulation of test cases
	int a, b, p;
	for(i = 0; i < pow(2, bits); i++){
		for(j = 0; j < pow(2, bits); j++){

			write_sig_arr(A, i);
			write_sig_arr(B, j);
			sc_start(10, SC_NS);
			a = read_sig_arr(A);
			b = read_sig_arr(B);
			p = read_sig_arr(P);
			cout << "A * B = C_out P: " << a << " * " << b << " = " << C_out.read() <<
" " << p << endl;
		}
	}

	cout << "14" << endl;
}
```

Makefile

```
#
CC=/usr/bin/g++
ARCH := $(shell arch)
SYSTEMC_HOME=/usr/local/SystemC-2.3.0

# 64bit or 32bit libaries to link to
LINUXLIB := $(shell if [ ${ARCH} = "i686" ]; \
                    then \
                         echo lib-linux; \
                    else \
                         echo lib-linux64; \
                    fi)

INCLUDES = -I$(SYSTEMC_HOME)/include -I.

LIBRARIES = -L. -L$(SYSTEMC_HOME)/$(LINUXLIB) -lsystemc -lm

RPATH = -Wl,-rpath=$(SYSTEMC_HOME)/$(LINUXLIB)

PROGRAM = test.x
SRCS    = cpa.h cpa.cpp csa.h csa.cpp sc_main.cpp
OBJS    = cpa.o csa.o sc_main.o

all : $(PROGRAM)

$(OBJS) : $(SRCS)
	$(CC) $(INCLUDES) -c $(SRCS)

$(PROGRAM) : $(OBJS)
	$(CC) $(INCLUDES) $(LIBRARIES) $(RPATH) -o $(PROGRAM) $(OBJS)

clean:
	@rm -f $(OBJS) $(PROGRAM) *.cpp~ *.h~
```

Appendix C: Multiplier Module Code

mult.h

```
#ifndef MULT_H
#define MULT_H

#include <systemc.h>

#include <string.h>
#include "cpa.h"              //carry propagte adder (CPA)
#include "csa.h"              //carry save adder (CPA)
#include "splitter.h"  //split single int signal into vector of bit signals
#include "joiner.h"           //join vector of bit signals into single int signal

template<int bits>     //min input width for each input into multiplier is at least 2 bits wide
SC_MODULE(mult){
        //inputs and outputs of multiplier
        sc_in< sc_uint<bits> > a;
        sc_in< sc_uint<bits> > b;
        sc_out< sc_uint<2*bits> > p;

        //signals and modules for splitting input bit signals from their input port, and joining
output bit signals to their output port
        sc_vector< sc_signal<bool> > A;
        splitter<bits> sa;

        sc_vector< sc_signal<bool> > B;
        splitter<bits> sb;

        sc_vector< sc_signal<bool> > P;
        joiner<2*bits> jp;

        //carry save adder (CSA) modules and signals
        csa* csa_arr [bits][bits];
        sc_signal<bool> csa_Co [bits][bits];
        sc_signal<bool> csa_So [bits][bits];
        sc_signal<bool> csa_C_B0 [bits];
        sc_signal<bool> csa_S_B0 [bits];
        sc_signal<bool> csa_S_AMSB[bits-1];
        int i,j;
        char csa_name [9];

        //carry propagate adder (CPA) modules and signals
        cpa* cpa_arr [bits];
        sc_signal<bool> cpa_Co [bits];
        sc_signal<bool> cpa_AMSB, cpa_C0, C_out;
        char cpa_name [7];

        SC_CTOR(mult) : sa("splitter_a"), A("A", bits), sb("splitter_b"), B("B", bits), P("P",
2*bits), jp("joiner_p")
        {
                //splitter and joiner blocks linked to signals and input/output ports
                sa.in(a);
                sa.out(A);
                sb.in(b);
                sb.out(B);
                jp.in(P);
                jp.out(p);

                cout << "mult structure:" << endl;

                //initialize carry save adder modules (CSA)
                strncpy(csa_name, "csa_A#B#", 9);
                for(j = 0; j < bits; j++){
                        for(i = 0; i < bits; i++){
                                csa_name[5] = i + '0';
                                csa_name[7] = j + '0';
                                cout << csa_name << " ";
                                csa_arr[i][j] = new csa(csa_name);
                                csa_arr[i][j] -> A(A[i]);
```

```
				csa_arr[i][j] -> B(B[j]);
				csa_arr[i][j] -> Co(csa_Co[i][j]);

				if(i == 0){
					csa_arr[0][j] -> So(P[j]); //connect So of each B bit in
row A[0] to P[0:bits-1]
				}
				else{
					csa_arr[i][j] -> So(csa_So[i][j]);
				}

				//assigning 0 connections to C and S
				if(j == 0){
					csa_arr[i][0] -> C(csa_C_B0[i]);	//C = 0 at edge of
matrix B[0]
					csa_C_B0[i].write(0);
					csa_arr[i][0] -> S(csa_S_B0[i]);	//S = 0 at edge of
matrix B[0]
					csa_S_B0[i].write(0);
				}
				else{
					csa_arr[i][j] -> C(csa_Co[i][j-1]);	//C = (C of previous B
bit) in each A row

					if(i == bits-1){
						csa_arr[bits-1][j] -> S(csa_S_AMSB[j-1]);	//S = 0
at edge of matrix A[MSB]
						csa_S_AMSB[j-1].write(0);
					}
					else{
						csa_arr[i][j] -> S(csa_So[i+1][j-1]); //S = S of
previous B bit and next A bit: for B[1] to B[MSB] and A[0] to A[MSB-1]
					}
				}
			}
			cout << endl;
		}

		//initialize carry propagate adder modules (CPA) for each A bit
		strncpy(cpa_name, "cpa_A#", 6);
		for(i = 0; i < bits; i++){
			cpa_name[5] = i + '0';
			cout << " " << cpa_name << "  ";
			cpa_arr[i] = new cpa(cpa_name);
			cpa_arr[i] -> B(csa_Co[i][bits-1]);	//connect B of each CPA to Co of each
CSA (in B[MSB] row for each A bit)
			cpa_arr[i] -> So(P[i+bits]);		//connect So of each CPA to each
P[4:7] bit

			if(i == 0){
				cpa_arr[0] -> C(cpa_C0);				//for first
CPA, set C to '0'
				cpa_arr[i] -> A(csa_So[i+1][bits-1]); //connect So of next CSA (in
B[MSB] row) to A of prev CPA
				cpa_arr[i] -> Co(cpa_Co[i]);			//connect Co from each
CPA to array of Co signals
				cpa_C0.write(0);
			}
			else if(i == bits-1){
				cpa_arr[i] -> C(cpa_Co[i-1]); //connect Co of prev CPA to C of next
CPA
				cpa_arr[bits-1] -> A(cpa_AMSB);		//for last CPA, set A
to '0'
				cpa_arr[bits-1] -> Co(C_out);		//for last CPA, set Co to
C_out
				cpa_AMSB.write(0);
			}
			else{
				cpa_arr[i] -> C(cpa_Co[i-1]);			//connect Co of prev
CPA to C of next CPA
```

```
                              cpa_arr[i] -> A(csa_So[i+1][bits-1]); //connect So of next CSA (in 
B[MSB] row) to A of prev CPA
                              cpa_arr[i] -> Co(cpa_Co[i]);                 //connect Co from each 
CPA to array of Co signals
                       }
               }
               cout << endl;

               cout << "done setup of mult unit" << endl;
        }
};

#endif
```

Appendix D: Carry Propagate Adder Module Code

cpa.h

```
#ifndef CPA_H
#define CPA_H

#include <systemc.h>

//carry propagate adder
SC_MODULE(cpa){
	//inputs and outputs
	sc_in<bool> A;		//input bit A
	sc_in<bool> B;		//input bit B
	sc_in<bool> C;		//carry in bit
	sc_out<bool> So;	//sum of A and B
	sc_out<bool> Co;	//carry out

	//behaviour of carry save adder
	void behaviour();

	//constructor
	SC_CTOR(cpa) : A("A"), B("B"), C("C"), Co("Co"), So("So")
	{
		SC_METHOD(behaviour);		//use SC_METHOD to simulate CPA behaviour
		sensitive << A << B << C;	//call CPA behaviour when any of the inputs change
	}
};

#endif
```

cpa.cpp

```
#include <iostream>
#include "cpa.h"

//internal signals
bool a, b, c_in, sum, c_out;
bool a_xor_b;

bool XOR(bool x, bool y);

void cpa :: behaviour(){
	//read inputs
	a = A.read();
	b = B.read();
	c_in = C.read();

	//calculate sum and carry out signals
	sum = XOR(c_in, XOR(a, b));						//C_in
xor A xor B
	c_out = ((!c_in) && (a && b)) + (c_in && (a + b));	//[NOT(C_in) AND (A AND B)] OR [C_in
AND (A OR B)]

	//set outputs
	So.write(sum);
	Co.write(c_out);

	//cout << name() << " [Cin A B] = " << c_in << " " << a << " " << b << " -> [Cout Sum]: "
<< c_out << " " << sum << endl;
}

bool XOR(bool x, bool y){
	return ((!x) && y) || (x && (!y));
}
```

Appendix E: Carry Save Adder Module Code

csa.h

```
#ifndef CSA_H
#define CSA_H

#include "cpa.h"

//carry save adder
SC_MODULE(csa){
	//inputs and outputs
	sc_in<bool> A;		//input bit A
	sc_in<bool> B;		//input bit B
	sc_in<bool> C;		//carry in bit
	sc_in<bool> S;		//sum in bit
	sc_out<bool> So;	//sum of A and B
	sc_out<bool> Co;	//carry out

	sc_signal<bool> AB;

	//internal carry propagate adder module
	cpa cpa1;

	//behaviour of carry save adder
	void behaviour();

	//constructor
	SC_CTOR(csa) : A("A"), B("B"), C("C"), S("S"), Co("Co"), So("So"), cpa1("cpa")
	{
		//set up internal module
		cpa1.A(AB);		//output from AND gate is first input bit
		cpa1.B(S);		//sum acts as other input bit
		cpa1.C(C);
		cpa1.So(So);
		cpa1.Co(Co);

		//determine simulation method
		SC_METHOD(behaviour);	//use SC_METHOD to simulate CSA behaviour
		sensitive << A << B;	//call CSA behaviour when any of the inputs change
	}

};

#endif
```

csa.cpp

```
#include <iostream>
#include "csa.h"

bool a_and_b;

void csa :: behaviour() {
	a_and_b = A.read() & B.read();
	AB.write(a_and_b);	//simulate AND gate at input of CPA adder block

	//cout << "\t" << name() << " [Cin A B S] = " << C.read() << " " << A.read() << " " << 
S.read() << " " << B.read() << ", [Cin A*B S] " << C.read() << " " <<  a_and_b << " " << B.read() 
<< " -> " << endl;
}
```

Appendix F: Splitter Module Code

This module splits a single sc_signal bus into an sc_vector that contains a separate sc_signal variable for each bit in the bus.

splitter.h

```
#ifndef SPLITTER_H
#define SPLITTER_H

#include <systemc.h>

//split int into array of ports
template <int BITS>
SC_MODULE(splitter){
	sc_in<sc_uint<BITS> > in;
	sc_vector< sc_out<bool> > out;

	int i, num;

	//splits input into each port for output
	void behaviour();

	SC_CTOR(splitter) : out("out", BITS)
	{
		SC_METHOD(behaviour);
		sensitive << in;
	}
};


template <int BITS>
void splitter <BITS> :: behaviour(){
	num = in.read();

	for(i = 0; i < BITS; i++){
		out[i].write((num & (1 << i)) > 0);
	}
}

#endif
```

splitter.cpp

```
/*#include <iostream>
#include "joiner.h"

//function is never recognized by compiler?
int i, sum;
template <int BITS>
void joiner <BITS> :: behaviour(){

	sum = 0;
	for(i = 0; i < BITS; i++){
		sum += (in[i].read() << i);
	}
	out.write(sum);
}*/
```

Appendix G: Joiner Module Code

This module joins each bit (as an sc_signal variable) within an sc_vector into single sc_signal bus.

joiner.h

```
#ifndef JOINER_H
#define JOINER_H

#include <systemc.h>

//split int into array of ports
template <int BITS>
SC_MODULE(joiner){
	sc_vector< sc_in<bool> > in;
	sc_out< sc_uint<BITS> > out;

	int i, sum;

	//splits input into each port for output
	void behaviour();

	SC_CTOR(joiner) : in("in", BITS)
	{
		SC_METHOD(behaviour);
		for(i = 0; i < in.size(); i++){
			sensitive << in[i];
		}
	}
};

template <int BITS>
void joiner <BITS> :: behaviour(){
	sum = 0;
	for(i = 0; i < BITS; i++){
		sum += (in[i].read() << i);
	}
	out.write(sum);
}

#endif
```

joiner.cpp

```
/*#include <iostream>
#include "joiner.h"

//function is never recognized by compiler?
int i, sum;
template <int BITS>
void joiner <BITS> :: behaviour(){

	sum = 0;
	for(i = 0; i < BITS; i++){
		sum += (in[i].read() << i);
	}
	out.write(sum);
}*/
```