ICS

TECHNICAL REPORT

Interface Synthesis at Behavioral RTL

Dongwan Shin, Pei Zhang and Daniel Gajski

Technical Report ICS-01-07
February 15, 2001

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, pzhang, gajski}@ics.uci.edu

Information and Computer Science
University of California, Irvine

Interface Synthesis at Behavioral RTL

Dongwan Shin, Pei Zhang and Daniel Gajski

Technical Report ICS-01-07
February 15, 2001

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, pzhang, gajski}@ics.uci.edu

Abstract

This report describes the interface synthesis methodology at behavioral RTL, based on handshaking protocol using the the SpecC system level design language, which has been developed at CAD Lab., UC Irvine. We use the parity encoder with two communicating behaviors as example. To synchronize two communicating behaviors, we show the methodology to generate the handshaking protocol and transducer.

Contents

List of Figures

Interface Synthesis at Behavioral RTL

Dongwan Shin, Pei Zhang and Daniel Gajski
Center for Embedded Computer System
Information and Computer Science
University of California, Irvine

Abstract

This report describes the interface synthesis methodology at behavioral RTL, based on handshaking protocol using the the SpecC system level design language, which has been developed at CAD Lab., UC Irvine. We use the parity encoder with two communicating behaviors as example. To synchronize two communicating behaviors, we show the methodology to generate the handshaking protocol and transducer.

1. Introduction

The SpecC methodology[GZD+00] uses SpecC system-level design language to implement a system from the specification model to manufacturing-ready implementation model.

The SpecC methodology has 4 levels of abstraction: specification, architecture, communication, and implementation model. The specification model is a pure behavioral description, in which communication between the behavioral blocks are implemented by using global variables rather than channels since up to now no concurrency and synchronization characteristic is specified.

The architecture model is an refined model from the specification model by partitioning the hardware and software. The concurrency and synchronization relationships are explicitly described by substituting the global variables with the channels, which will finally be encapsulated into global busses. The communication between the software blocks running on the same processor is still implemented by using the global variables.

The communication model is the same as the architecture model in that the blocks are mapped to the same components. However, the protocol descriptions for the inter-block communication is refined into the timing-accurate description. The timing-accurate model can be described without knowing the internal structure of the hardware if the I/O protocols between these communicating blocks are clearly defined. But if the protocol is not explicitly given for a block, the RTL description must be analyzed to generate the corresponding I/O protocol. If protocols for the communicating blocks are not compatible, transducer has to be inserted to synchronize them.

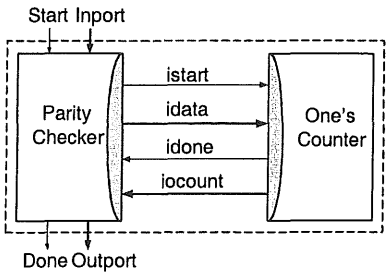

Figure 1. Parity encoder with two communicating behaviors

Finally, the hardware blocks are refined into cycle-accurate description, which is the lowest level of abstraction in the SpecC methodology. The implemenation model has two views: a behavioral RTL view and a structural RTL view. The behaviroal RTL specifies the operations performed in each clock cycle with explicitly modelling the units in the component's datapath and is obtained by scheduling the operations in the C code into clock cycles. However, due to the interdependence of scheduling, allocation, and binding, behavioral RTL requires all three steps to be preformed, and should be refined into the structural RTL. Therefore, the structural RTL view of the implemenation model explicity models the allocation of RTL components, the scheduling of register transfers into clock cycles, and the binding of operations, variables and assignments to functional units, register/memoryies and components busses.

In this report, we mainly focus on synthesizing the communication model into implementation model(behavioral RTL view) of parity encoder, which is written at 2 different levels of abstraction in the SpecC methodology. The rest of the report is organized as follows: section 2 shows the parity encoder algorithm at 2 levels of abstraction. Sec-

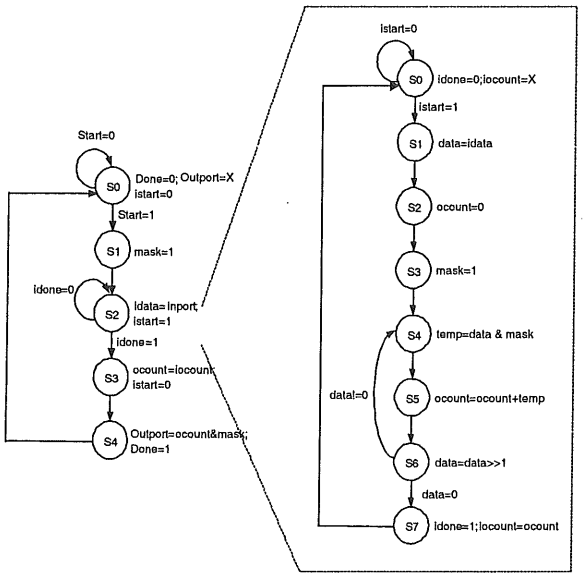

Figure 2. Parity encoder with two communicating behaviors at behavioral RTL

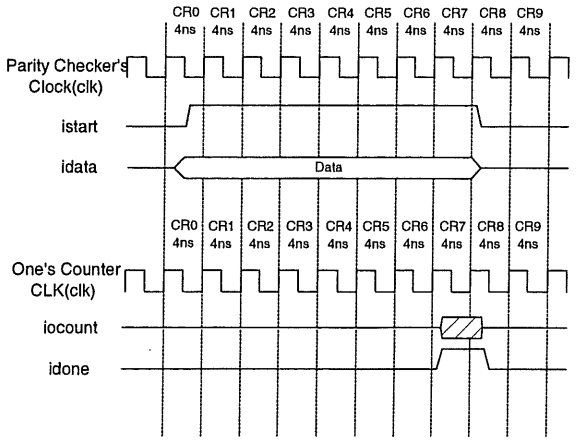

Figure 3. Timing diagram with two communicating behaviors

tion 3 describes transducer synthesis methodology for two communicating behaviors with different clocks for the parity encoder. We conclude the report in section 4.

2. Synthesis for Parity Encoder

Parity encoder is utilized as error detection/correction coding in data communication community. We select it to find the problems which occur in refining from communication model with communicating behaviors into implementation model. The operation of parity encoder is shown in Fig. 1. The parity encoder is activated by `start` signal, and gets data as input, and generates done signal and even parity output. The parity encoder is composed of two behaviors: one's counter and even parity checker. The first computes the number of one for data, and the last generates even parity bit by examining the output of one's counter. These two behaviors communicate through signals such as `istart`, `idone`, `idata`, and `iocount`. The one's counter is actived when `istart` signal is asserted to high and `idone` is asserted to high when the number of one's is calculated. Appendix A.1 represents the communication model for the parity encoder.

2.1. Parity encoder with single clock

Assuming that the delay of all data processing operations is one clock cycle and ALU and shifter unit are only used as datapath unit, the parity encoder is designed into behavioral RTL model in FSMD represenation. Fig. 2 shows the behavioral RTL description for parity encoder with one's counter which is sub-FSMD of parity encoder. The even parity checker has 5 states and one's counter 8 states. In Fig. 2, dotted line represents that the one's counter is the sub FSMD of parity checker in state `S2`. Because the one's counter has internal loop between `S4` and `S6`, the latency of the parity encoder is not known before its execution. We know just the maximum/minimum latency of the parity encoder. Fig.3 shows the timing diagram between parity checker and one's counter. In our implementation, the period of clock for even parity checker is selected *6ns* and one's counter, *4ns*. The behavioral RTL SpecC codes for parity encoder is shown in Appendix A.2.

2.2. Parity encoder with internal different clocks

In our implementation, handshaking protocol is utilized to implement the parity encoder. Generally handshaking protocol solved the synchronization problem for communicating behaviors with different clocks. The implemenation which is shown in section 2.1 is working when the clock period of parity checker is larger than that of one's counter. But one's counter goes to state `S0` because `istart` of one's counter is 1 after state `S7` while even parity checker waits that `idone` becomes 1, and then it results in generating incorrect output. This synchronization problem can be fixed by moving `ocount = iocount` in state `S3` into state `S2` of parity checker and merging state `S3` and state`S4`. Fig. 4 shows the FSMD model for the parity encoder with different clocks. The timing diagram with communicating behaviors with two different clocks is shown in Fig. 5 and the modified code of parity checker is shown in Appendix A.3

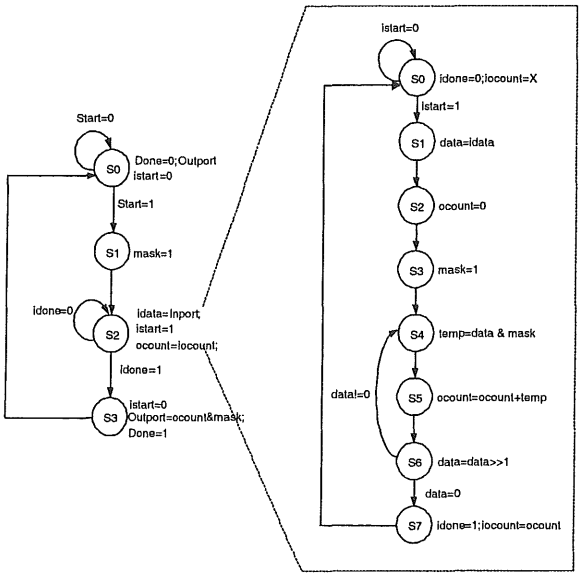

Figure 4. FSMD diagram with two communicating behaviors with different clocks

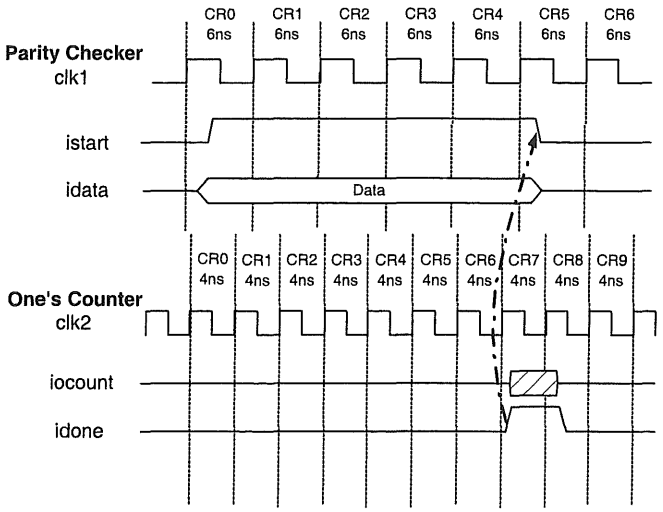

Figure 5. timing diagram with two communicating behaviors with different clocks

2.3. Parity encoder with acknowledge signals

As mentioned so far, the communication protocol between parity checker and one's counter don't have acknowledge signal `ack`, and the `istart` signal and `idata` should be maintained through execution of one's counter. If the protocol has acknowledge signals, `iack_start` for `istart` and `iack_done` for `idone`, the `istart` and `idata` don't need to be maintained during the execution of one's counter any longer. Fig. 6 shows the modified FSMD for parity encoder with acknowledge signals. Timing diagram is shown in Fig. 7

3. Transducer design for parity encoder

The functionality of transducer includes: 1) repacket the data to the type which can be recognized by the other communicating block; 2) adjust the data arrival and leaving time such that data is safely transferred. Because implemenation of parity encoder doesn't have application layer communication protocols such as slicing, we don't need to consider repacketing the data into the other communicating block. Now we consider just adjusting the data synchronization. Fig. 8 represents the parity encoder which is composed of even parity checker and one's counter, communicating with different clock period. The even parity checker is operated by `clk1` and one's counter `clk2`. The clock of transducer is selected between minimum of `clk1` and `clk2`, and 3rd clock(greatest common divisor of the clock periods). The selection of clock period of transducer is important. If clock period is larger than those of two behavior, overall latency of parity encoder will increase. Otherwise, latency is will decrease but design may have the timing violation problem due to clock skew. In our implemenation, the clock of transducer is selected to minimum of that of parity checker and one's counter.

3.1. Transducer synthesis methodology

A protocol specification is typically composed of 5 atomic operations(data read/write, control wait/assign, time delay)[NG95]:

1. waiting for an event on an input control line
2. assigning a value to an output control line
3. reading a value from input data line
4. assigning a value from output data line, and
5. waiting for a fixed time interval

Becasuse time delay operation is not implemented by hardware synthesis tool and all operations are synchronized by event, we don't need to consider time delay operation any longer.

The first step in interface synthesis is to represent each of the two communicating protocols as ordered set of *relations*, which is a set of assignments to output control and data lines and the reading of a value from input data lines, upon the occurrence of a certain condition. The *condition* could be an event on an input control line or fixed delay with respect to some previous event. The protocol of even parity checker and one's counter should be extracted from their

Atomic operation	SpecC equivalent	Dual operation in transducer
waiting for event	signal.waitval(1)	signal.assign(1)
assign control line	signal.assign(1)	signal.waitval(1)
read data line	var = data	data = temp_var
assign data line	data = var	temp_var = data
fixed delay	waitfor(10)	waitfor(10)

Table 1. Dual of atomic protocol operations

behavioral description as ordered set of relations, which is shown in Fig. 9(a). Having derived the set of relations for the two protocols, we now need to group the relations in the two protocols into a set of relation groups, which are ordered subset of the set of relations that represents a unit of data transfer between the two processes. The relation groups are created in such a manner that the size of the data generated by the relations in the group from one protocol is identical to that expected by the relations in the group from the other protocol. Fig. 9 shows the partitioning the relations of parity encoder.

Having combined the relations into a set of relations groups, we now generate the interface process to make the two protocols compatible. The set of operations in the relation groups taken in order represents the sequence of atomic operations across the two protocols. The interface process can be obtained by simply replacing them with their exact *dual* or complementary operations which are shown in Table 1. Fig. 10 shows the transducer behavior with dual of operations in parity encoder. Two lines of codes in circle will be omitted because `istart.waitval(0)` is followed by `istart.waitval(1)` and the parity checker doesn't wait for `istart` to be low. The transducer has FSMD with the 4 states which is shown in Fig. 11 and the datapath for transducer is shown in Fig. 12. The timing diagram for Transducer implementation is shown in Fig. 13. The detail codes is shown in Appendix A.5.

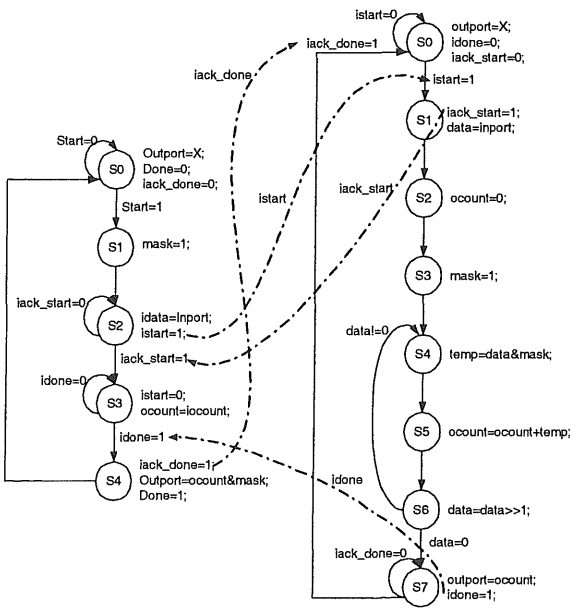

Figure 6. FSMD diagram with acknowledge signals

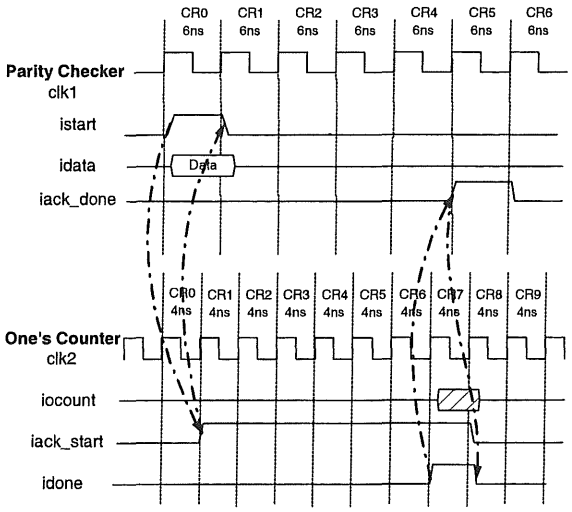

Figure 7. Timing diagram with acknowledge signals

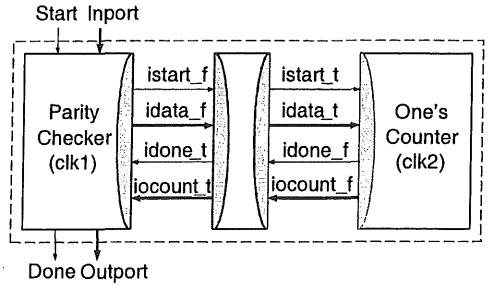

Figure 8. Parity encoder with transducer by different clock period

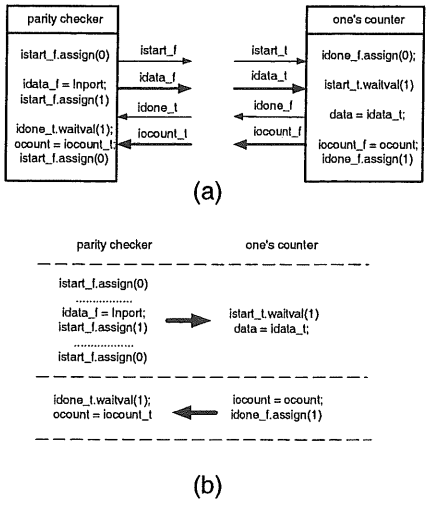

Figure 9. (a) two protocal in SpecC language (b) partititong the relations of parity encoder

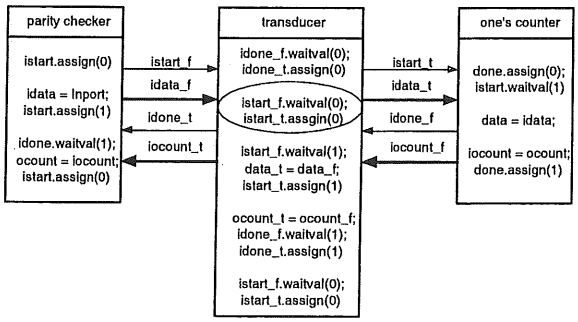

Figure 10. transducer behavior with dual of operations in parity encoder

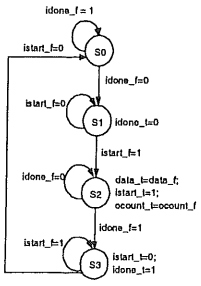

Figure 11. transducer in FSMD

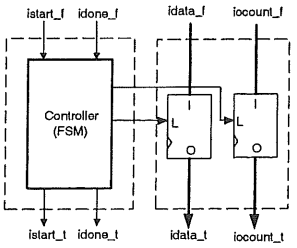

Figure 12. Datapath for transducer

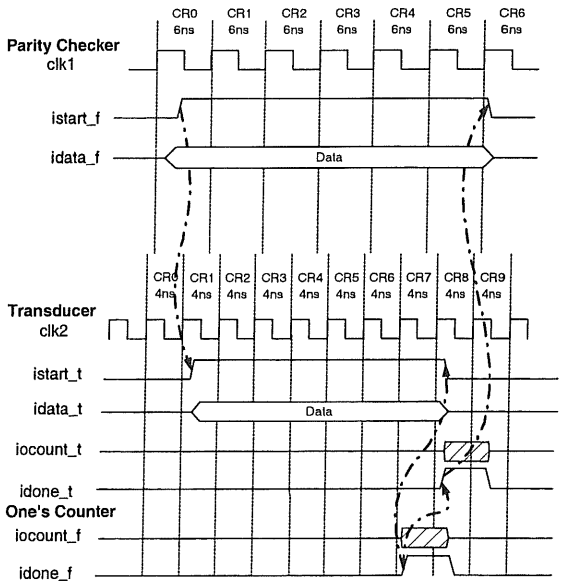

Figure 13. Timing diagram for parity encoder with transducer

4. Conclusion

In this report, we refined communication model into behavioral RTL model in SpecC design methodology. The communication model and behavioral RTL model of the parity encoder was written in SpecC language. To synchronize two communicating behaviors with different clocks, the transducer and the handshaking protocol were implemented in well-defined steps. But we know through this work that behavioral RTL with same protocol doesn't need transducer as though it has internal different clocks.

5. Acknowledgement

The authors would like to thank BroadComm and Conexant of providing fellowship to CECS of UC, Irvine, and Andreas Gerslauer who made many helpful comments for its improvement.

References

[GZD+00] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[NG95] Sanjiv Narayan and Daniel D. Gajski. Interfacing incompatible protocols using interface process generation. In *Proceedings of the Design Automation Conference*, pages 468–473, June 1995.

A. SpecC codes for parity encoder

A.1. Parity encoder: communicaton model

A.1.1 Parity encoder: parity.sc

```
import "ones";
import "even";

behavior parity(in bit[31:0] Inport, out bit[31:0] Outport, iISignal Start,
    iOSignal Done)
{
    bit[31:0] data, ocount;
    cSignal cstart, cdone ;

    even U00(Inport, Outport, Start, Done, data, ocount, cstart, cdone);
    ones U01(data, ocount, cstart, cdone);

    void main (void)
    {
        par {
            U00.main();
            U01.main();
        }
    }
};
```

A.1.2 Even parity checker: even.sc

```
import "bus";

behavior even(in bit[31:0] Inport, out bit[31:0] Outport, iISignal Start,
    iOSignal Done, out bit[31:0] idata, in bit[31:0] iocount, iOSignal istart,
    iISignal idone)
{
    void main(void) {
        bit[31:0] ocount;
        bit[31:0] mask = 0x0001;

        while (1) {
            printf("even: S0\n");
            Done.assign(0);
            Outport = -1;
            Start.waitval(1);

            istart.assign(1);
            idata = Inport; // inlined master bus write
            waitfor(1);
            istart.assign(0);

            printf("even: S1\n");
            idone.waitval(1);   // inlined master bus read
            ocount = iocount; // wait for result of one's counter
```

```
            printf("even:␣S2\n");
            Outport = ocount & mask;     // even parity checker
            Done.assign(1);
            waitfor(1);      // synchronization for testbench
        }
    }
};
```

A.1.3 One's counter: ones.sc

```
import "bus";

behavior ones(in bit[31:0] inport, out bit[31:0] outport, iISignal start,
    iOSignal done)
{
    void main(void) {
        bit[31:0] data;
        bit[31:0] ocount;
        bit[31:0] mask;
        bit[31:0] temp;

        while (1) {
            printf("ones:␣S0\n");
            start.waitval(1); // slave bus read
            data = inport;

            ocount = 0;
            mask = 1;

            while (data != 0) {
                temp = data & mask;
                ocount = ocount + temp;
                data = data >> 1;
            }

            done.assign(1);      // slave bus write
            outport = ocount;
            waitfor(1);
            done.assign(0);
            printf("ones:␣S1\n");
        }
    }
};
```

A.1.4 Testbench: tb.sc

```
import "io";
import "ones";
import "parity";

behavior Main
{
    bit[31:0] inport, outport;
    cSignal start, done;

    IO io(inport, outport, start, done);
```

```
    parity U00(inport, outport, start, done);

    int main (void)
    {
        par {
            io.main();
            U00.main();
        }
        return 0;
    }
};
```

A.1.5 Input/Output for testbench: io.sc

```
import "bus";

// get interger from stdin
behavior IO(out bit[31:0] inport, in bit[31:0] outport, iOSignal start,
    iISignal done)
{
    void main(void) {
        char buf[16];

        while (1) {
            start.assign(0); // maintain start signal low
            printf("Input for parity checker: ");
            gets(buf);
            inport = atoi(buf);

            done.waitval(0);
            start.assign(1);    // start parity checker

            done.waitval(1);
            printf("parity checker output = %d\n", (int)outport);
        }
    }
};
```

A.1.6 Bus: bus.sc

```
//Signal channel for representation of control signal
interface iOSignal
{
    void assign ( int v ) ;
};

interface iISignal
{
    int val() ;
    void waitval ( int v ) ;
};

channel cSignal()
    implements iISignal, iOSignal
{
    int value=0;
```

```
        event ev;

        void assign ( int v )    // assign a value
        {
            value = v ;
            notify ( ev ) ;
        }

        int val()    // return a value
        {
            return value ;
        }

        void waitval ( int v )  // wait for a value
        {
            while ( value != v )
                wait ( ev ) ;
        }
};

//master bus channle
interface iMasterBus
{
    bit[31:0] read();
    void write(bit[31:0] data);
};

// master bus side of protocol
channel cMasterBus(iOSignal start, iISignal done, out bit[31:0] wdata,
    in bit[31:0] rdata) implements iMasterBus
{

    bit[31:0] read() {
        int data;
        done.waitval(1);    // wait for done signal
        data = rdata;           // sample data bus
        return data;
    }

    void write(bit[31:0] data) {
        start.assign(1);    // assert start signal
        wdata = data;           // write to data bus
        waitfor(1);
        start.assign(0);    // deassert start signal
    }
};

//slave bus channel
interface iSlaveBus
{
    bit[31:0] receive();
    void send(bit[31:0] data);
};
```

```
//slave bus side of protocol
channel cSlaveBus(iISignal start, iOSignal done, out bit[31:0] wdata,
    in bit[31:0] rdata) implements iSlaveBus
{
    bit [31:0] receive() {
        int data;
        start.waitval(1);       // wait for start signal
        data = rdata;               // sample data bus
        return data;
    }

    void send(bit[31:0] data) {
        done.assign(1);     // assert done signal
        wdata = data;           // write to data bus
        waitfor(1);
        done.assign(0);     // deassert done signal
    }
};

#endif
```

A.2. Parity encoder with single clock: RTL model

A.2.1 Parity encoder: parity.sc

```
import "ones";
import "even";

behavior parity(in event clk1, in event clk2, in bit[0:0] rst,
    in bit[31:0] Inport, out bit[31:0] Outport, iISignal Start, iOSignal Done)
{
    bit[31:0] data, ocount;
    cSignal cstart, cdone ;

    even U00(clk1, rst, Inport, Outport, Start, Done, data, ocount, cstart, cdone);
    ones U01(clk2, rst, data, ocount, cstart, cdone);

    void main (void)
    {
        par {
            U00.main();
            U01.main();
        }
    }
};
```

A.2.2 Even parity checker: even.sc

```
import "bus";

behavior even(in event clk, in bit[0:0] rst, in bit[31:0] Inport,
    out bit[31:0] Outport, iISignal Start, iOSignal Done,
    out bit[31:0] idata, in bit[31:0] iocount, iOSignal istart, iISignal idone)
{
```

```
    void main(void) {
        bit[31:0] ocount;
        bit[31:0] mask;
        enum state { S0, S1, S2, S3, S4} state;

        state = S0;

        while (1) {
            wait(clk);
            if (rst == 1b) {
                state = S0;
                Outport = -1;
            }
            switch (state) {
                case S0:
                    Outport = -1;
                    Done.assign(0);
                    if (Start.val() == 1)
                        state = S1;
                    else
                        state = S0;
                    break;
                case S1:
                    mask = 0x0001;
                    state = S2;
                    break;
                case S2:
                    idata = Inport;
                    istart.assign(1);
                    if (idone.val() == 1) // wait for result of one's counter
                        state = S3;
                    else
                        state = S2;
                    break;
                case S3:
                    ocount = iocount;
                    istart.assign(0);
                    state = S3;
                    break;
                case S3:
                    Outport = ocount & mask;     // even parity checker
                    Done.assign(1);
                    state = S0;
                    break;
            }
        }
    }
};
```

A.2.3 One's counter: ones.sc

```
import "bus";

behavior ones(in event clk, in bit [0:0] rst, in bit[31:0] inport,
```

```
    out bit[31:0] outport, iISignal start, iOSignal done)
{
    void main(void) {
        bit[31:0] data;
        bit[31:0] ocount;
        bit[31:0] mask;
        bit[31:0] temp;

        enum state { S0, S1, S2, S3, S4, S5, S6, S7 } state;

        state = S0;

        while (1) {
            wait(clk);
            if (rst == 1b) {
                outport = 0x0000;
                state = S0;
            }
            switch (state) {
                case S0 :
                    done.assign(0);
                    outport = -1;
                    if (start.val() == 1)
                        state = S1;
                    else
                        state = S0;
                    break;
                case S1:
                    data = inport;
                    state = S2;
                    break;
                case S2:
                    ocount = 0;
                    state = S3;
                    break;
                case S3:
                    mask = 1;
                    state = S4;
                    break;
                case S4:
                    temp = data & mask;
                    state = S5;
                    break;
                case S5:
                    ocount = ocount + temp;
                    state = S6;
                    break;
                case S6:
                    data = data >> 1;
                    if (data == 0)
                        state = S7;
                    else
                        state = S4;
```

```
                    break;
                case S7:
                    outport = ocount;
                    done.assign(1);
                    state = S0;
                    break;
            }
        }
    }
};
```

A.2.4 Testbench: tb.sc

```
import "io";
import "clock_gen";
import "ones";
import "parity";

behavior Main
{
    bit[31:0] inport, outport;
    bit[0:0] rst;
    event clk1, clk2;
    int clk1_period = 6;
    int clk2_period = 4;
    cSignal start, done;

    IO io(clk1, rst, inport, outport, start, done);
    clock_gen clk_gen01(clk1, clk1_period);
    clock_gen clk_gen02(clk2, clk2_period);
    parity U00(clk1, clk2, rst, inport, outport, start, done);

    int main (void)
    {
        par {
            io.main();
            clk_gen01.main();
            clk_gen02.main();
            U00.main();
        }
        return 0;
    }
};
```

A.2.5 Input/Output for testbench: io.sc

```
import "bus";

// i/o for testbench
behavior IO(in event clk, out bit[0:0] rst, out bit[31:0] inport,
    out bit[31:0] outport, iOSignal start, iISignal done)
{
    void main(void) {
        char buf[16];

```

```
        rst = 1b;       // reset all storage elements in design during 2 clock cycles
        start.assign(0); // maintain start signal low during 2 clock cycle
        wait(clk);
        wait(clk);

        rst = 0b;       // deassign reset

        while (1) {
            wait(clk);
            printf("Input␣for␣one's␣counter:␣");
            gets(buf);
            inport = atoi(buf);

            start.assign(1);    // now, design calculates num. of one in inport
            wait(clk);

            done.waitval(1);
            printf("output␣=␣%d\n", (int)outport);

            wait(clk);
            start.assign(0);
            wait(clk);
            wait(clk);
        }
    }
};
```

A.2.6 Clock generation: clock_gen.sc

```
import "bus";

behavior clock_gen(out event clk, in int hw_clk_period)
{
    void main(void) {
        while (1) {
            waitfor(hw_clk_period);
            printf("\nclock␣event(period:␣%d)!!!\n", hw_clk_period);
            notify(clk);
        }
    }
};
```

A.2.7 Bus: bus.sc

```
//Signal channel for representation of control signal
interface iOSignal
{
    void assign ( int v ) ;
};

interface iISignal
{
    int val() ;
    void waitval ( int v ) ;
};
```

```
channel cSignal()
    implements iISignal, iOSignal
{
    int value=0;
    event ev;

    void assign ( int v )    // assign a value
    {
        value = v ;
        notify ( ev ) ;
    }

    int val()    // return a value
    {
        return value ;
    }

    void waitval ( int v )  // wait for a value
    {
        while ( value != v )
            wait ( ev ) ;
    }
};

//master bus channle
interface iMasterBus
{
    bit[31:0] read();
    void write(bit[31:0] data);
};

// master bus side of protocol
channel cMasterBus(iOSignal start, iISignal done, out bit[31:0] wdata,
    in bit[31:0] rdata) implements iMasterBus
{

    bit[31:0] read() {
        int data;
        done.waitval(1);     // wait for done signal
        data = rdata;            // sample data bus
        return data;
    }

    void write(bit[31:0] data) {
        start.assign(1);     // assert start signal
        wdata = data;            // write to data bus
        waitfor(1);
        start.assign(0);     // deassert start signal
    }
};

//slave bus channel
interface iSlaveBus
```

```
{
    bit[31:0] receive();
    void send(bit[31:0] data);
};

//slave bus side of protocol
channel cSlaveBus(iISignal start, iOSignal done, out bit[31:0] wdata,
    in bit[31:0] rdata) implements iSlaveBus
{
    bit [31:0] receive() {
        int data;
        start.waitval(1);       // wait for start signal
        data = rdata;               // sample data bus
        return data;
    }

    void send(bit[31:0] data) {
        done.assign(1);     // assert done signal
        wdata = data;           // write to data bus
        waitfor(1);
        done.assign(0);     // deassert done signal
    }
};

#endif
```

A.3. Parity encoder with two different clocks: RTL model

A.3.1 Even parity checker: even.sc

```
import "bus";

behavior even(in event clk, in bit[0:0] rst, in bit[31:0] Inport,
    out bit[31:0] Outport, iISignal Start, iOSignal Done,
    out bit[31:0] idata, in bit[31:0] iocount, iOSignal istart, iISignal idone)
{
    void main(void) {
        bit[31:0] ocount;
        bit[31:0] mask;
        enum state { S0, S1, S2, S3 } state;

        state = S0;

        while (1) {
            wait(clk);
            if (rst == 1b) {
                state = S0;
                Outport = -1;
            }
            switch (state) {
                case S0:
                    Outport = -1;
                    Done.assign(0);
                    if (Start.val() == 1)
```

```
                    state = S1;
                else
                    state = S0;
                break;
            case S1:
                mask = 0x0001;
                state = S2;
                break;
            case S2:
                idata = Inport;
                istart.assign(1);
                ocount = iocount;
                if (idone.val() == 1)
                    state = S3;
                else
                    state = S2;
                break;
            case S3:
                istart.assign(0);
                Outport = ocount & mask;     // even parity checker
                Done.assign(1);
                state = S0;
                break;
        }
    }
}
};
```

A.4. Parity encoder with acknowledge signals

A.4.1 Even parity checker: even.sc

```
import "bus";

behavior even(in event clk, in bit[0:0] rst, in bit[31:0] Inport,
    out bit[31:0] Outport, iISignal Start, iOSignal Done,
    out bit[31:0] idata, in bit[31:0] iocount, iOSignal istart, iISignal idone,
    iISignal iack_start, iOSignal iack_done)
{
    void main(void) {
        bit[31:0] ocount;
        bit[31:0] mask;
        enum state { S0, S1, S2, S3, S4 } state;

        state = S0;

        while (1) {
            wait(clk);
            if (rst == 1b) {
                state = S0;
                Outport = -1;
            }
            switch (state) {
                case S0:
```

```
                    printf("even: S0\n");
                    Outport = -1;
                    Done.assign(0);
                    iack_done.assign(0);
                    if (Start.val() == 1)
                        state = S1;
                    else
                        state = S0;
                    break;
                case S1:
                    printf("even: S1\n");
                    mask = 0x0001;
                    state = S2;
                    break;
                case S2:
                    printf("even: S2\n");
                    idata = Inport;
                    istart.assign(1);
                    if (iack_start.val() == 1)
                        state = S3;
                    else
                        state = S2;
                    break;
                case S3:
                    printf("even: S3\n");
                    istart.assign(0);
                    ocount = iocount;
                    if (idone.val() == 1)
                        state = S4;
                    else
                        state = S3;
                    break;
                case S4:
                    printf("even: S4\n");
                    iack_done.assign(1);
                    Outport = ocount & mask;     // even parity checker
                    Done.assign(1);
                    state = S0;
                    break;
            }
        }
    }
};
```

A.4.2 One's counter: ones.sc

```
import "bus";

behavior ones(in event clk, in bit [0:0] rst, in bit[31:0] inport,
    out bit[31:0] outport, iISignal start, iOSignal done)
{
    void main(void) {
        bit[31:0] data;
        bit[31:0] ocount;
```

```
bit[31:0] mask;
bit[31:0] temp;

enum state { S0, S1, S2, S3, S4, S5, S6, S7 } state;

state = S0;

while (1) {
    wait(clk);
    if (rst == 1b) {
        outport = 0x0000;
        state = S0;
    }
    switch (state) {
        case S0 :
            done.assign(0);
            outport = -1;
            if (start.val() == 1)
                state = S1;
            else
                state = S0;
            break;
        case S1:
            data = inport;
            state = S2;
            break;
        case S2:
            ocount = 0;
            state = S3;
            break;
        case S3:
            mask = 1;
            state = S4;
            break;
        case S4:
            temp = data & mask;
            state = S5;
            break;
        case S5:
            ocount = ocount + temp;
            state = S6;
            break;
        case S6:
            data = data >> 1;
            if (data == 0)
                state = S7;
            else
                state = S4;
            break;
        case S7:
            outport = ocount;
            done.assign(1);
            state = S0;
```

```
                    break;
                }
            }
        }
    };
```

A.5. Parity encoder with transducer: RTL model

A.5.1 Transducer: txducer.sc

```
import "bus";

behavior txducer(in event clk, in bit[31:0] data_f, out bit[31:0] data_t,
    in bit[31:0] ocount_f, out bit[31:0] ocount_t, iISignal istart_f, iOSignal istart_t,
    iISignal idone_f, iOSignal idone_t)
{
    void main(void) {
        enum state { S0, S1, S2, S3 } state;

        state = S0;
        while (1) {
            wait(clk);
            switch (state) {
                case S0:
                    if (idone_f.val() == 0)
                        state = S1;
                    else
                        state = S0;
                    break;
                case S1:
                    idone_t.assign(0);
                    if (istart_f.val() == 1)
                        state = S2;
                    else
                        state = S1;
                    break;
                case S2:
                    data_t = data_f;
                    istart_t.assign(1);
                    ocount_t = ocount_f;
                    if (idone_f.val() == 1)
                        state = S3;
                    else
                        state = S2;
                    break;
                case S3:
                    istart_t.assign(0);
                    idone_t.assign(1);
                    if (istart_f.val() == 0)
                        state = S0;
                    else
                        state = S3;
                    break;
            }
```

```
            }
        }
    };
```