ICS

TECHNICAL REPORT

Design of A Transducer for Parity Encoder

Haobo Yu
Qiang Xie
Daniel D. Gajski

Technical Report ICS-01-08
Feb 14, 2000

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

{ haoboy, qxie, gajski}@ics.uci.edu
http://www.ics.uci.edu/~haoboy

Information and Computer Science
University of California, Irvine

Contents

1 Introduction 1

2 The parity encoder 1

3 Parity encoder with transducer 2

4 Coldfire Master Bus 2

5 Transducer design 3

5.1 The first transducer 3

5.2 The second transducer 4

6 Conclusion 5

References 5

Appendix A: SpecC code for the first transducer 6

A.1 bus.sc..... 6

A.2counter.sc..... 8

A.3 tb.sc..... 10

A.4 transducer.sc..... 11

Appendix B: SpecC code for the second transducer 13

B.1queue.sc..... 13

B.2 transducer.sc 15

List of Figures

1 Communication Model with bus 2

2 Communication Model : bus inlined..... 2

3 Parity Encoder with transducer..... 3

4 FSMD for the first transducer 4

5 Block diagram for the second transducer 4

6 FSMD1 and FSMD2 for the second transducer 4

Design of A Transducer for Parity Encoder

Haobo Yu, Qiang Xie, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

This report describes the design of a transducer for parity encoder using SpecC RTL methodology. We first begin with the introduction of parity encoder. Then the bus protocol of ColdFire processor is introduced and we present two ways to implement the transducer. The source codes are also included in the Appendix.

1 Introduction

The goal of this project is to explore the ways of synthesizing the communication model into implementation model in SpecC. We do this by showing the design of a parity encoder using SpecC methodology.

The SpecC methodology [GZDG00][GERS00] has four different levels of abstraction: specification, architecture, communication and implementation model.

The specification model is a pure behavior description. The communications between the behavioral blocks are implemented by using global variables rather than channels since up to now no concurrency and synchronization is specified.

The architecture model is a refined model from the specification model with the partition of hardware and software. The concurrency and synchronization relationships resulting from this partitioning are explicitly described by substituting the global variables with the channels, which will finally be encapsulated into a global bus. The communication between 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 description for the inter-block communication is refined into the timing-accurate description.
The implementation model is the result of scheduling the functionality mapped to the components (computation and communication functionality) into register transfers per clock cycle. Therefore, the implementation model is a cycle-accurate model at the register-transfer level.

The implementation model supports two views of the components in the design: a behavioral RTL view and a structural RTL view. In both cases, the steps of allocation, binding and scheduling are required to derive the implementation model. The behavioral RTL does not explicitly represent the datapath architecture and the binding information. The structural RTL view, on the other hand, explicitly describes the structure of data path plus control unit.

In this report, we describe the design of a transducer for parity encoder using SpecC Methodology[GZDG00][GERS00]. The rest of the report is organized as follows: section 2 shows the parity encoder communication model, section 3 describe the parity encoder design with a transducer. Section 4,5,6 describes the design of a transducer for ColdFire and one's counter in the implementation of a parity encoder.

2 The parity encoder

The parity encoder is used in many areas as error detection/correction. In our project, we use it to illustrate the communication model synthesis.

The parity encoder consists of two parts: the parity encoder main part and the one's counter. The parity encoder main part use the result from one's counter to calculate the parity encoder bit.

In our communication model for one's counter, the main part of parity encoder and one's counter communicates using a bus. We implemented the parity encoder communication model both with the bus as separate behavior and with the bus inlined as shown in Figure1 and Figure2.

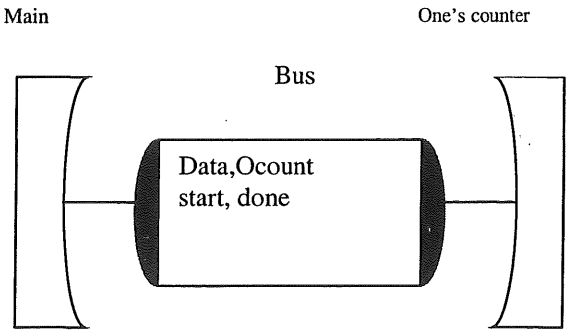


Figure1.Communication Model with bus

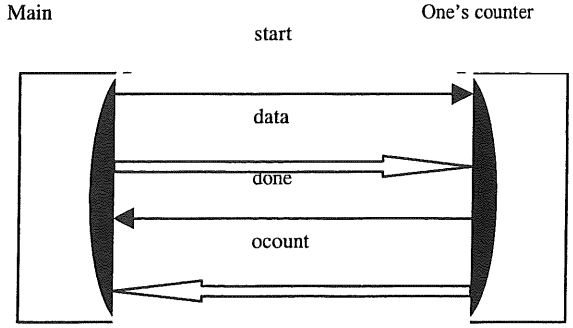


Figure 2.Communication Model: bus inlined

3 Parity encoder with transducer

In the above implementation, both the main part of parity encoder and one's counter are implemented in hardware. Now, we implement the parity encoder main part in software using ColdFire as processor. We need a transducer to connect the ColdFire with the one's counter. Figure 3 shows the structure of our parity encoder with transducer

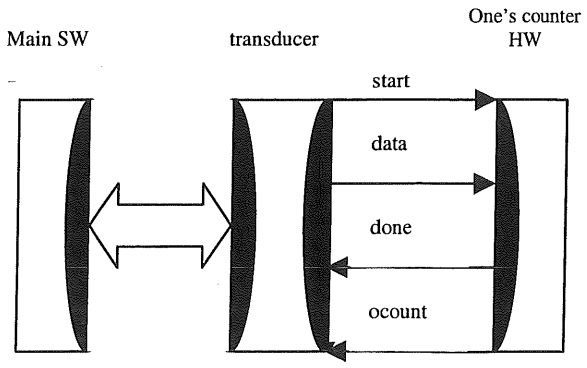


Figure 3.Partiy Encoder with transducer

4 Coldfire master bus

In our parity encoder example, the ColdFire protocal is employed for the software part to input/output the data. The protocol for the ColdFire and the one's counter are not compatible, so a transducer is inserted.

The communication between the ColdFire processor and an external device is based on the ColdFire Master Bus protocol. A subset of the master bus signals used in our model are listed below.

Master Address Bus (MADDR[31:0]). If these output signals are used, the memory space of the external device is implicitly mapped into the memory space in the ColdFire processor. So the ColdFire processor can explicitly specify the target address of the external device during the I/O process.

Master Read/Write (MRWB). This output signal indicates a data read/write operation for the current bus communication. A high level '1' indicates a read and a low level '0' indicates a write.

Master Transfer Start (MTSB). This output signal indicates the start of a bus transfer when it is asserted to be '0'.

Master Read/Write Data Bus (MRDATA[31:0] and MWDATA[31:0]). These input/output bus signals provide the datapaths for the data I/O. We can choose to use 8, 16, 32 bits of the data bus per data bus transfer.

Master Transfer Acknowledge (MTAB). This input signal is asserted to '0' by the slave to indicate the successful completion of a bus transfer. It is sampled by the ColdFire Processor at the end of each clock cycle. If it is not asserted, the ColdFire inserts one or more wait states.

Master Write Data Output Enable (MWDATAOE). when asserted to '1', this output signal indicates that the ColdFire is driving the master write data bus. This is used to control optional bi-directional data bus three-state drivers.

For the ColdFire read I/O, in the first clock cycle (CR0), the address (MADDR) and control information (MTSB, MRWB) are driven onto the bus. In the next cycle (CR1), the address MADDR remains on the bus. The MRWB is still asserted. But MTSB is deasserted to '1'. At this

time, one wait states may be inserted if the slave is not ready to send the data. In this case, the slave will keep (deserting) MTAB as '1'. The ColdFire samples the MTAB at the end of the current clock cycle. Because it finds that the MTAB is deasserted ('1'), the ColdFire will insert one wait state (CR1 wait). In other words, the ColdFire will stay one more clock cycle at this state (CR1). In the next cycle (the ColdFire is still at state CR1), if the slave is ready to send the data, it will assert the MTAB to '0'. At the end of this clock cycle, the ColdFire samples the MTAB. It finds that the MTAB is asserted to '0', so at the end of this clock cycle, the data at the data bus MRDATA is latch into the ColdFire data buffer. If there are more I/O requests suspending, the ColdFire will immediately go back to the state CR0 and start the next data I/O. Otherwise, if there is no more I/O requests, the ColdFire will go to the IDLE state

For the ColdFire write I/O, in the first clock cycle (CW0), the ColdFire asserts the MTSB to '0', drives the address to address bus MADDR, asserts the MRWB to '0' indicating a write I/O. The MWDATAOE is kept deasserted as '0' indicating that data will be not ready on the MWDATA bus until the next clock cycle. In the next clock cycle (CW1), the data is driven onto the data bus by the ColdFire. Also in this clock cycle, the address remains at the MADDR. The MRWB is still asserted as '0'. But the MTSB is deasserted to '1', and the MWDATAOE is asserted to '1' indicating that data is now on the data bus MWDATA. If the slave is not ready to latch the data, the MTAB will not be asserted, that is, the MTAB is '1' in this clock cycle(CW1). At the end of the current clock cycle (CW1), the ColdFire samples the MTAB, and if the MTAB is still deasserted ('1'), one more wait state (CW1 wait) will be inserted. In the next clock cycle (now the ColdFire is still in the state CW1), if the slave is ready to latch the data (which is not shown in the timing diagram), it will assert the MTAB to '0'. At the end of this clock cycle (CW1), the ColdFire samples the MTAB, and it finds that the MTAB is asserted to '0', so it removes the data from MWDATA and jumps out of the wait state (CW1). If there is more I/O requests suspending, the ColdFire will immediately go to the state CW0 and starts a new data transfer. Otherwise, it will go to the IDLE state. busses.

5. Transducer design

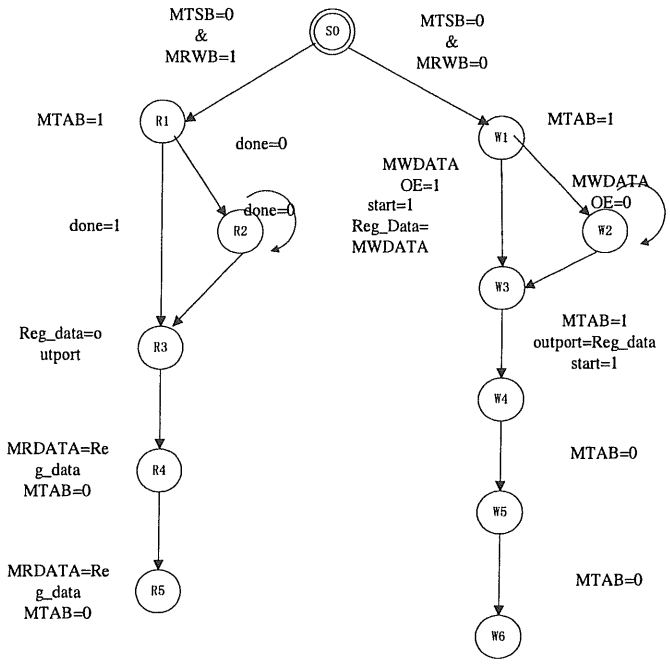


Figure4. FSMD for the first transducer

The functionality of a transducer includes

- Re-pack the data to the type which can be recognized by the other communicating block;
- Adjust the data arrive and leaving time such that data is safely transferred

In this section, we implemented the transducer for parity encoder in two ways.

5.1 The first transducer

The clock for ColdFire bus is 6ns and the one's counter use the 4ns clock. So in our transducer, we set the clock as 4ns.Figure 4 is the FSMD model of our transducer

When the MRWB bit of ColdFire is '1',that is for the ColdFire to read one's counter result. In R1 state, the MTAB is not asserted (that is '1'). At the end of W1, the transducer samples the done signal from the one's counter. If the done signal is asserted to '1', the transducer goes to R3. If the done signal is not asserted (that is '0'), the transducer will stay at R2 state and keeps sampling the done signal. When the done signal is assigned to '1' by the one's counter, the transducer will enter R3. During R3, the one's

counter drives data from outport to transducer internal storage unit Reg_data. In R4, the data is driven onto the ColdFire's master read data bus (MRDATA[31:0]). At the same time, the MTAB

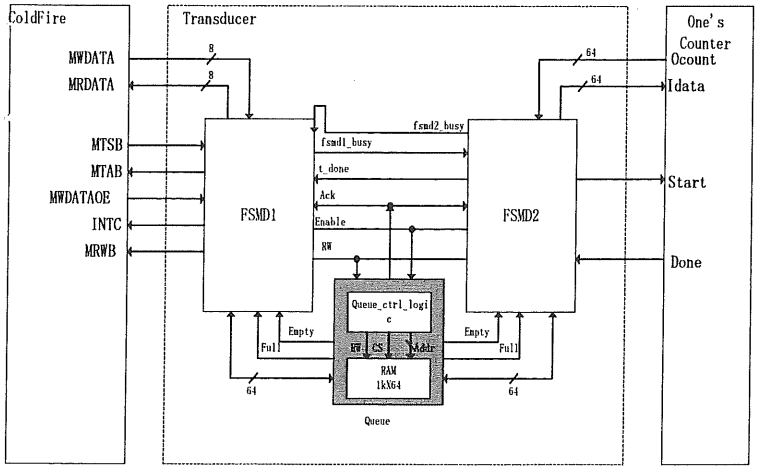


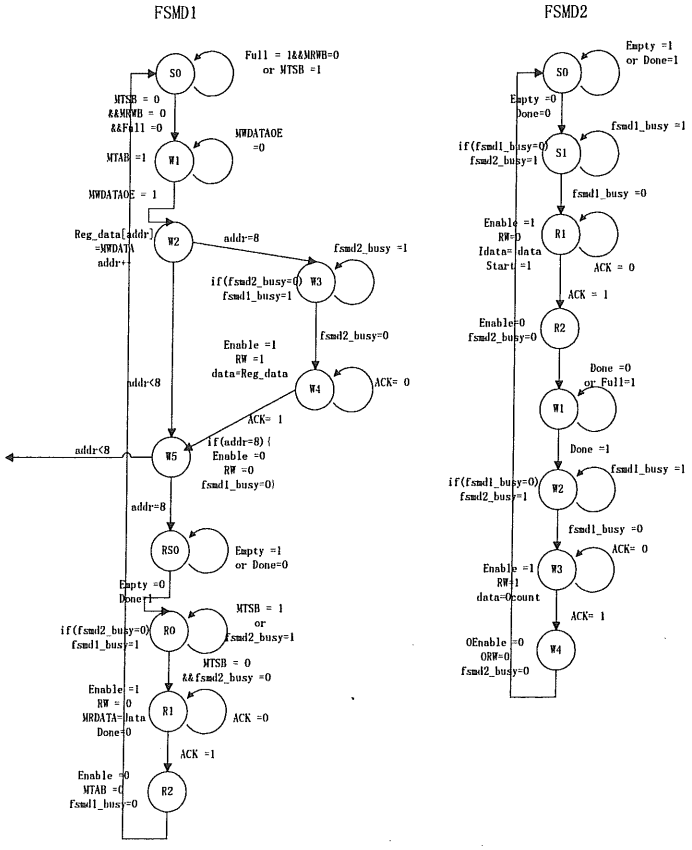
Figure 5: Block diagram for the second transducer

is asserted to '0'. The data remains on the data bus and the MTAB is still asserted '0' in TR5 because the ColdFire's clock is 6ns while our transducer's clock is 4ns. We need to make 2 transducer state (8ns) to ensure ColdFire samples the correct value of the MTAB.

When the MRWB bit of ColdFire is '0',that is for the ColdFire to send start signal and data to one's counter. In W1 state, the MTAB is not asserted (that is '1') .At the end of W1, the transducer samples the MWDATAOE and latches the current value on the MWDATA[31:0]. If the MWDATAOE is asserted to '1', the transducer goes to W3. If MWDATAOE is not asserted (that is '0'), the transducer goes to W2. In W2, the transducer keeps sampling the MWDATAOE and latches the current value on the MWDATA[31:0] until the MWDATAOE is asserted to '1', then it goes to W3.In W3, the transducer get the data from ColdFire bus MWDATA and write to its storage unit Reg_Data. During W4, the transducer asserts the start singal to '1' telling the one's counter to begin its operation and drives the data onto the one's counter. In W5 and W6, the transducer asserts the MTAB to '0' indicating the ColdFire the completion of this bus I/O. Because ColdFire's clock is 6ns while our transducer's clock is 4ns, we need to make 2 transducer state (8ns) to ensure ColdFire samples the correct value of the MTAB.

5.2 The Second Transducer

Now, we try to design the transducer in a general way. This method uses queue to transfer data



Firgure 6 :FSMD1 and FSMD2 for the second transducer

between two FSMD. Figure 5 shows the block diagram of this transducer.

The transducer includes three parts: two FSMD and one data queue. FSMD1 is used to communicate with ColdFire. It will receive the data from ColdFire and put to the queue or get the data from the queue and send to ColdFire. FSMD2 read the data from the data queue and sends the data to one's counter or get the result from one's counter and put into the queue. We selected the queue from standard queues: The data queue includes two parts, the queue controller and a data buffer that can store up to 1k 64-bit data. Figure 6 shows the FSMD diagram of this transducer which illustrate the communication process between the ColdFire and the One's Counter. When ColdFire begins to transfer data, it executes the ColdFire Master bus protocol and send the data out. FSMD1 receives the data and put the incoming data to a data queue. We should notice that the ColdFire send 8bit data per time while FSMD1 converts these 8

bit data to 64bit data using some shift registers and send to the queue. If the data queue is not full(Full = 0), the data will be stored to the data queue. After FSMD2 detects that the data queue is not empty (Empty = 0), it read the data from the queue and send the data to the One's Counter. When the One's Counter finishes the calculation, it set the Done signal and send the result back to FSMD1. FSMD1 receives the result and send it back to the data queue. Then FSMD1 fetch the result and send back to ColdFire by executing the bus protocol.

6. Conclusion

In this report, we designed a transducer for the parity encoder. We explored two ways to design the transducer: the first transducer consists of one FSMD and a register. But it can only work in our example, we just take advantage of the ColdFire protocol and the simple protocol of one's counter. The second transducer is composed of two FSMD and one queue (with 8, 16, 32, 64, 128, 256 bit wide memory). This is a general transducer model. Actually, the design process of a transducer with two FSMD and one queue shows that we can use this general transducer model to connect any two IPs with different protocols. Based on this design, we are further thinking of ways to generate transducer automatically from the given protocols.

References

[GZDG00]D. Gajski er al. *SpecC: Specification Language and Design Methodology*, Kluwer Academic Publishers, 2000

[GERS00]A. Gerstlauer *SpecC Modeling Guidelines*, University of California, Irvine, Technical Report ICS-00-xx, September 2000

[YDLG00] H.Yin, H,Du, Lee, D.D. Gajski , Design of a JPEG Encoding System using SpecC Methodology

Appendix A: SpecC code for the first transducer

A.1 bus.sc

```
/*************************************************************************
*
Filename: bus.sc
Description: Bus Definition
*************************************************************************/
#include <stdio.h>
#define HW_CLK1 6
interface ISignal{
      void assign(int v);
      int  val();
      void waitval(int v);
      };
interface OSignal{
      void assign(int v);
      int  val();
      void waitval(int v);
      };
channel CSignal() implements ISignal, OSignal
{
      int value;
      event ev;
      void assign(int v){ // assign a value;
            value = v;
            notify(ev);
      }
      int val(){        // return the value;
            return value;
      }
      void waitval(int v){    //wait for value
            while(value != v) wait(ev);
      }
};
interface iMasterBusProtocol
{
      bit[31:0] read(bit[31:0] addr);
      void write(bit[31:0] addr, bit[31:0] data);
};

// master bus side of protocol
channel cMasterBusProtocol(   out bit[31:0] MWDATA,
                        in bit[31:0] MRDATA,
                        OSignal MTSB,
                        ISignal MTAB,
                        OSignal MWDATAOE,
                        out bit[31:0] MADDR,
                        OSignal MRWB
                    )
      implements iMasterBusProtocol
{
      bit[31:0] read(bit[31:0] addr)
      {
```

```
            int data;
            int val;

            // first clock cycle CR0
            MADDR = addr ;          //  assign address lines
            MTSB.assign(0) ;  //  assert the start of bus transfer
            MRWB.assign(1) ;  //  assert read control
            waitfor ( HW_CLK1 ) ;

            // second clock cycle CR1
            val = MTAB.val() ;      //  sample acknowledge line
            while(val==1)
            {
                  MADDR = addr ;    //  assign address lines
                  MTSB.assign(1) ;//  deassert the start of bus
transfer
                  MRWB.assign(1) ;//  assert read control
                  waitfor(HW_CLK1);
                  val = MTAB.val();//  sample acknowledge line
            }
            data = MRDATA ;         //  sample data bus
            waitfor(HW_CLK1);
            return data;
      }
void write(bit[31:0] addr, bit[31:0] data)
      {
            int val ;

            // first clock cycle CW0
            MADDR = addr ;          //  assign address lines
            MTSB.assign(0) ;  //  assert the start of bus transfer
            MRWB.assign(0) ;  //  assert write control
            MWDATAOE.assign(0) ;    //  deassert write data available
            waitfor ( HW_CLK1 ) ;

            // second clock cycle CW1
            val = MTAB.val() ;      //  sample acknowledge line
            while(val==1)
            {
                  MADDR = addr ;  //  assign address lines
                  MTSB.assign(1) ;//  deassert the start of bus
transfer
                  MRWB.assign(0) ;//  assert write control
                  MWDATAOE.assign(1);//  assert write data available
                  MWDATA = data ; //  drive data outputs
                  waitfor(HW_CLK1) ;
                  //printf("wait for the MTAB value to be 1 ");
                  val = MTAB.val() ; // sample acknowledge line

            }
            waitfor(HW_CLK1);
      }
};

interface iMasterBus
{
      void send(int data, int addr);
```

```
        int recv(int addr);
};

channel cMasterBus (    out bit[31:0] MWDATA,
                        in bit[31:0] MRDATA,
                        OSignal MTSB,
                        ISignal MTAB,
                        OSignal MWDATAOE,
                        out bit[31:0] MADDR,
                        OSignal MRWB,
                        ISignal INTC
                        )
        implements iMasterBus
{

        cMasterBusProtocol  protocol (MWDATA, MRDATA, MTSB, MTAB,
MWDATAOE,
                                      MADDR, MRWB ) ;
        int i ,temp;

        void send(int data, int addr)
        {
                // data transfer
                protocol.write ( addr, data ) ;
        }

        int recv(int addr)
        {
                // high-level synchronization, wait for ready signal
                INTC.waitval ( 0 ) ;
                waitfor(HW_CLK1) ;

                // data transfer
                temp = protocol.read ( addr ) ;
                return(temp);
        }

};
```

A.2 counter.sc

```
/*************************************************************************
*
   Filename: counter.sc
   Description: One's counter model(bus interface not inline)
*************************************************************************/
import "bus";
#ifndef HW_CLK
#define HW_CLK 4
#endif
behavior HW(in bit[31:0] inport,
              out bit[31:0] outport,
              ISignal start,
              OSignal done          )
{
      void main(void)
```

```
{
bit[31:0] data, Mask, temp; //temp. variables;
bit[7:0]  count;
enum state {S0, S1, S2, S3, S4, S5, S6,S7,S8} state;
state = S0;   //initial states;
while( state != S8 ){
      switch(state)
      {
        case S0 :
              done.assign(0);
            if (start.val() == 1)
                  state = S1;
            else
                  state = S0;
            break;

        case S1:
            data = inport;
            printf("data=%d \n" , (int)data);
            state = S2;
            break;

        case S2:
            count = 000b;
            state =S3;
            break;

        case S3:
            Mask = 001b;
            state = S4;
            break;

        case S4:
           temp = data & Mask;
            state = S5;
            break;

        case S5:
            count += temp;
            state = S6;
            break;

        case S6:
            data = data >> 1;
            if(data)
                 state = S4;
            else
                 state = S7;
            break;

        case S7:
            done.assign(1);
            outport=count;   //output data;
            state = S8;
            break;
        }
     waitfor(HW_CLK);
```

```
            }
        }
    };
```

A.3 tb.sc

```
/*********************************************************************
*
    Filename: tb.sc
   Description : One's counter model,TestBench
*********************************************************************/
import "transducer";
behavior SW(out bit[31:0] MWDATA,
          in bit[31:0] MRDATA,
          OSignal MTSB,
          ISignal MTAB,
          OSignal MWDATAOE,
          out bit[31:0] MADDR,
          OSignal MRWB,
          ISignal INTC )
{
cMasterBus  bus(MWDATA, MRDATA, MTSB, MTAB, MWDATAOE,MADDR, MRWB,
INTC);      void main(void)
      {
            int inport,outport;
            printf("Input for one's counter: ");
            scanf("%d",&inport);
            printf("Bus sending %d to one's counter.... ",inport);
            bus.send(inport,0);
            outport=bus.recv(0);
            printf("The one's number is  = %d\n", outport);
      }
};
behavior Main
{
      bit[31:0] MWDATA ;      // coldfire write data bus
      bit[31:0] MRDATA ;      // coldfire read data bus
      bit[31:0] MADDR  ;      // address
      // coldfire-transducer control
      CSignal MTSB, MTAB, MWDATAOE,MRWB, INTC ;
      //transducer-one's counter control
      CSignal start,done;
      bit[31:0] inport,outport;
      SW   sw(MWDATA, MRDATA, MTSB, MTAB, MWDATAOE,MADDR, MRWB, INTC);
      HW   hw(inport,outport,start,done);
      INTERFACE  inter(MWDATA, MRDATA, MTSB, MTAB, MWDATAOE, MADDR,
                  MRWB,inport,outport,start,done);
      int main(void){
            MWDATAOE.assign(0) ;
            printf("********************\n");
            printf(" Begin...\n");
            printf("********************\n");
            par
            {
                  sw.main();
                  inter.main();
```

```
                hw.main();
            }
            printf("******************\n");
            printf(" End...\n");
            printf("******************\n");
            return(1);
            }
};
```

A.4 transducer.sc

```
/*************************************************************************
*
   Filename: transducer.sc
   Stage: transducer for coldfire and one's counter
*************************************************************************/
#ifndef __INTERFACE_
#define __INTERFACE_
import "counter";
#ifndef HW_CLK
#define HW_CLK 4
#endif
behavior INTERFACE(in bit[31:0] MWDATA,
            out bit[31:0] MRDATA,
            ISignal MTSB,
            OSignal MTAB,
            ISignal MWDATAOE,
            in bit[31:0] MADDR,
            ISignal MRWB,
            out bit[31:0] inport,
              in bit[31:0]  outport,
              OSignal start,
              ISignal done )
{
      int Reg_Data ;
   void main(void) {
      enum state {S0, R1, R2, R3, R4, R5, W1,W2,W3,W4,W5,W6} state;
      int val,temp ;
      state=S0;
      while (1) {
      switch(state){
            case S0:
                  MTAB.assign(1) ;
                  val = MTSB.val() ;
                  if(val==0){       // begin a new bus transfer
                        temp=MRWB.val() ;
                        if (temp ==0)
                              state=W1;
                        else
                              state=R1;
                        }
                  else
                        state=S0;
                  break;

            case R1 :
                  MTAB.assign(1) ;
```

```
        state=R2;
        break;

case R2 :
        if (done.val()==1){
                state=R3;
                goto label1;
                }
        else
                state=R2;
        break;

case R3 :
        label1:
        Reg_Data = outport ;
        state=R4;
        break;

case R4 :
        MRDATA = Reg_Data ;
        MTAB.assign(0) ;
        state=R5;
        break;

case R5 :
        MRDATA = Reg_Data ;
        MTAB.assign(0) ;
        state=S0;
        break;

//coldfire write to one's counter
case W1 :
        MTAB.assign(1) ;
        state=W2;
        break;

case W2 :
        val = MWDATAOE.val() ;
        if (val == 0){
                state=W2;
                MTAB.assign(1) ;}
        else {
                state=W3;
                goto label2;
                }
        break;

case W3 :
        label2:
        MTAB.assign(1) ;
        start.assign(0) ;
        Reg_Data = MWDATA ;
        state=W4;
        break;

case W4 :
        MTAB.assign(1) ;
```

```
                inport = Reg_Data ;
                start.assign(1) ;
                state=W5;
                break;

          case W5 :
                MTAB.assign(0) ;
                state=W6;
                break;

          case W6 :
                MTAB.assign(0) ;
                waitfor(HW_CLK) ;
                state=S0;
           }
          waitfor(HW_CLK);
         }
    }
};
#endif
```

Appendix B: SpecC code for the second transducer

B.1 queue.sc

```
/*****************************************
   Project: RTL Model Implementation
   Stage: One's counter model(with ColdFire)
   Filename: queue.sc
   Last change: 1/11/01
   Author: Qiang Xie
*****************************************/
#define HW_CLK 6
import "bus";

behavior MEM( ISignal CS,
              ISignal RWS,
              inout bit[31:0] Data,
              in bit[3:0] addr)
{
      int mem_data[7];
      void main(void)
      {
      enum State{S0} state;
      state = S0;

      while(1){
        switch(state){
            case S0:
            if((CS.val()==1)&&(RWS.val()==1)) {
                  mem_data[addr] = Data;
            printf("write data: %d, Addr: %d\n", mem_data[addr], (int)
addr);
                  }
            else if((CS.val()==1)&&RWS.val()==0){
```

```
                    Data = mem_data[addr];
                printf("Read data: %d; Addr:  %d\n", (int) Data, (int)
addr);
                    }
                state = S0;
                break;
                }
        waitfor(HW_CLK);
        }
        }

} ;

behavior Queue_Ctrl_Logic(
                      ISignal Enable,
                      ISignal RW,
                      out bit[3:0] addr,
                      OSignal Empty,
                      OSignal Full,
                      OSignal CS,
                      OSignal Ready)
{
      bit[3:0] back_addr, front_addr;
      void main(void)
      {
            enum State{S0, S1}state;
            //initial state
            state = S0;
            back_addr  = 1000b;
            front_addr = 1000b;
            Empty.assign(1);
            Full.assign(0);

            while(1){
                  switch(state){
                  case S0:
                  Ready.assign(0);
                  if(back_addr[2:0] == front_addr[2:0]){
                        if(back_addr[3] == front_addr[3]){
                              Empty.assign(1);
                              Full.assign(0);
                        }else{
                              Empty.assign(0);
                              Full.assign(1);
                              }
                        }
                  else{
                        Empty.assign(0);
                        Full.assign(0);
                        }
                  if(Enable.val())
                  {
                    //select read and write address,
                    if(RW.val() ==1 ) {
                        addr = back_addr&0111b;
                        back_addr++;
                        }
```

```
                    else {
                        addr = front_addr&0111b;
                        front_addr++;
                        }
                    //select memory
                    CS.assign(1);
                    state = S1;
                  }else {
                        CS.assign(0);
                        Ready.assign(0);
                        }
                  break;

                  case S1:
                  Enable.assign(0);
                  Ready.assign(1);
                  state = S0;
                  break;
                  }
              waitfor(HW_CLK);
            }
    }
};


behavior Queue(
            ISignal Enable,
            ISignal RW,
            inout bit[31:0] data,
            OSignal Full,
            OSignal Empty,
            OSignal Ready)
{
            bit[3:0] addr;
            CSignal CS;


            MEM mem(CS, RW, data, addr);
            Queue_Ctrl_Logic q_ctrl(Enable, RW, addr, Empty, Full, CS,
Ready);

            void main(void)
            {
                  par{
                  q_ctrl.main();
                  mem.main();}
            }
};
```

B.2 transducer.sc

```
/*****************************************
   Project: RTL Model Implementation
   Stage: One's counter model(with ColdFire)
   Filename: transducer.sc
   Last change: 1/10/01
```

```
   Author: Qiang Xie
*****************************************/
#define HW_CLK_1 6
#define HW_CLK_2 4
import "bus";
import "queue";

#ifndef __INTERFACE_
#define __INTERFACE_

behavior FSMD_1(
            in bit[31:0] MADDR,
            in bit[31:0] MWDATA,
            out bit[31:0] MRDATA,
            ISignal MTSB,
            OSignal MTAB,
            ISignal MWDATAOE,
            OSignal INTC,
            ISignal MRWB,
            OSignal Enable,
            OSignal RW,
            ISignal Full,
            ISignal Empty,
            inout bit[31:0] iodata,
            ISignal Ready,
            ISignal Done)
{
      int Reg_data;

    void main(void) {
      enum State{S0, S1,
                 W1, W2, W3,
                 RS0, R0, R1, R2}
            state;
      state = S0;
      //initilize signal
      MTAB.assign(1);
      INTC.assign(1);
      Enable.assign(0);
      RW.assign(0);


      //T0
      while(1){
      switch(state){
            case S0:
            if((MTSB.val()==0)&&(MRWB.val() == 0)) {
                  //write data
                  if(Full.val()==1) {
                  printf("Data buffer full, Please send data
later.\n");
                  MTAB.assign(0);
                  state = S0;}
                  else {
                        MTAB.assign(1);
                        state = S1 ;}
                  }
```

```
            break;

            case S1:
            if(Enable.val() == 0) state = W1;
            break;

            //begin write data to slave bus
            case W1:
            if(MWDATAOE.val()==0)MTAB.assign(1) ;
            else state = W2;
            break;

            case W2:
            if(Ready.val() ==0){
                  Enable.assign(1);
                        RW.assign(1);
                  MTAB.assign(1);
                  iodata = MWDATA;}
            else
                  state = W3 ;
            break;

            case W3:
            Enable.assign(0);
            RW.assign(0);
            MTAB.assign(0);
            state = RS0;
            break;

            case RS0:
            if((Empty.val() == 0)&&(Done.val() == 1)) {
                  INTC.assign(0);
                  state = R0;
                  }
            break;

            case R0:
            if((MTSB.val() == 0)&&(MRWB.val() == 1)&&(Enable.val() ==0)
) state = R1 ;
            break;

            case R1:
            //TR1
            if(Ready.val() == 0){
                  MTAB.assign(1);
                  Enable.assign(1);
                  RW.assign(0);
                  INTC.assign(1);
                  MRDATA = iodata; }
            else
                  state = R2;
            break;

            case R2:
            Enable.assign(0);
            Done.assign(0);
            MTAB.assign(0) ;
```

```
                state = S0;
                break;
                }
                waitfor(HW_CLK_1);
                }

        }
};

behavior FSMD_2(
                inout bit[31:0] DB,
                OSignal Start,
                ISignal Done,
                OSignal Enable,
                OSignal RW,
                ISignal Full,
                ISignal Empty,
                inout bit[31:0] iodata,
                ISignal Ready,
                OSignal t_done
                )
{
                int addr;
                int Reg_data;

                void main(void){
                enum State{S0, S1, R1, R2, W1, W2, W3, W4} state;
                state = S0;
                t_done.assign(0);

                while(1){
                        switch(state){

                        case S0:
                        if((Empty.val()== 0)&&(t_done.val()==0)) state = S1;
                        break;

                        case S1:
                        if(Enable.val() == 0){
                                state = R1;}
                        break;

                        case R1:
                        printf("FSMD2 state R1\n");
                        if(Ready.val() == 0) {
                        Start.assign(1);
                        Enable.assign(1);
                        RW.assign(0);
                        DB = iodata; }
                        else
                        state = R2;
                        break;

                        case R2:
                        Enable.assign(0);
                        state = W1;
                        break;
```

```
                    //wait for result and write result to queue
                    case W1:    // wait Done
                    if((Done.val() == 1)&&(Full.val()==0))
                          state = W2;
                    break;

                    case W2:
                    if(Enable.val() == 0)
                    {
                          t_done.assign(1);
                          state = W3;}
                    break;

                    case W3:
                    if(Ready.val() ==0){
                          Enable.assign(1);
                          RW.assign(1);
                          iodata = DB;}
                    else
                          state = W4;
                    break;

                    case W4:
                    RW.assign(0);
                    Enable.assign(0);
                    Done.assign(0);
                    state = S0;

                    break;
                    }
              waitfor(HW_CLK_2);
              }

}
};


behavior Transducer(
                in bit[31:0] MADDR,
                in bit[31:0] MWDATA,
                out bit[31:0] MRDATA,
                ISignal MTSB,
                OSignal MTAB,
                ISignal MWDATAOE,
                OSignal INTC,
                inout bit[31:0]  DB,
                OSignal Start,
                ISignal Done,
                ISignal MRWB)
{
            bit[31:0] data;
            CSignal Enable;
            CSignal RW;
            CSignal Full, Empty;
            CSignal Ready;
            CSignal t_done;
```

```
            FSMD_1 fsm1(MADDR, MWDATA, MRDATA, MTSB, MTAB, MWDATAOE,
INTC,
                        MRWB, Enable, RW, Full, Empty, data,Ready,
t_done);
            FSMD_2 fsm2(DB, Start, Done, Enable, RW, Full, Empty, data,
Ready, t_done);
            Queue queue(Enable, RW, data, Full, Empty, Ready);

            void main(void){
                  Start.assign(0);
                  Done.assign(0);

                  par{
                  queue.main();
                  fsm1.main();
                  fsm2.main();
                  }
            }
};

#endif
```