Interface Synthesis from Protocol Specification

Dongwan Shin and Daniel Gajski

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

{dongwans,gajski}@cecs.uci.edu

Interface Synthesis from Protocol Specification

Dongwan Shin and Daniel Gajski

{dongwans,gajski}@cecs.uci.edu

Abstract

In system-on-chip design, automating design reuse is one of the most important issues. Since most Intellectual Properties(IP) are provided by different vendors, they have different interface schemes, and different data rates. In order to automate design reuse, methods for combining system components with incompatbile I/O and interface protocols, must be developed. Furthermore design interfaces between interacting components is an error-prone task. In this report, we propose the interface architecture in which queues are used for data transfers between components with incompatible protocols. We also describe a method to generate system interface from the protocol specification.

2

Contents

List of Figures

Interface Synthesis from Protocol Specification

Dongwan Shin and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

In system-on-chip design, automating design reuse is one of the most important issues. Since most Intellectual Properties(IP) are provided by different vendors, they have different interface schemes, and different data rates. In order to automate design reuse, methods for combining system components with incompatbile I/O and interface protocols, must be developed. Furthermore design interfaces between interacting components is an error-prone task. In this report, we propose the interface architecture in which queues are used for data transfers between components with incompatible protocols. We also describe a method to generate system interface from the protocol specification.

1. Introduction

Advances in the VLSI industry and design methodology have allowed the complexity of a single chip to contain more than millions of transistors. The increasing complexity of VLSI design and market pressures in the complex system-on-chip(SOC) have forced designers to consider reuse of Intellectual Property(IP) blocks [Mis01]. Since most IPs, however, are proivided by different vendors, and they have different interface scheme, and different data transfer rates, composition of these components should be developed. It is error-prone task and one of the most important part of system integration.

The basic purpose of an interface syntheis is to generate interfaces between incompatible components. Data could be transferred at different bit widths, operating frequencies, transfer rates and so on. In this report, we propose novel queue-based interface scheme, which general enough to accomodate any component protocols

The rest of this report is organized as follows: section 2 gives a brief description of previous and related work. Section 3 describes our interface architecture and canonical model of queue. Section 4 gives an overview of our protocol specification model. We will takes a closer look at synthesis step for out interface architecture. Section 6 shows the synthesis results by our method using examples. Section 7

concludes this report with a brief summary and future work.

2. Related work

The problem of interface synthesis between system components with incompatible protocols has been addressed in the literature. In [BK87], the event graph was introduced to establish the correct synchronization and data sequencing. The limitation of this approach is that the two protocols should be made compatible by assigning the labels manually to the data on both sides, since the protocol specification is given at low level of abstraction using waveforms.

In [LV94], signal transition graph was introduced for protocol specification and the hardware interface is synthesized with asynchronous logic. In [NG95], the protocol specification is decomposed into five basic operations(data read/write, control read/write, time delay), while the protocol is represented as an ordered set of relations whose execution is guarded by a condition or by a time delay. Finally, relations between two protocols are grouped into a set of relation groups which transfer the same amount of data.

In [SM98], the interface architecture provides a mechanism for implementing communication through the standard interface and enables the composition of synchronous blocks and provides hooks for optimizing system performance by prioritizing component communication. They have assumed that communicating components utilize the same data types. Interfaces between blocks with multiple busses can be generated when the control of these busses is separate. Futhermore, by communicating data through input and output buffers with separate read and write functionality, components can be connected that operate at different frequencies.

In [PRSV98], the two protocols are described using regular expressions and are translated into corresponding deterministic finite automata. Then interface protocol can be synthesized as an FSM by production computation algorithm. In this approach, the correspondence between pieces of data on the two sides is automatically resolved. However, the limitation of this approach is that the two communicating parties are driven by the same clock. In [PCPK00],

this limitation is overcome by inserting additional states and edges in the FSM and introducing a queue in the protocol converter.

3. Interface architecture

Our interface architecture is basically composed of synchronous system interfaces as shown in Figure 1. The system components(PE1 and PE2) may operate at different frequencies and at different data rates. Our interface architecture includes a buffer(queue) to smoothen the burst data transfer requests and two state machines(FSMD) to transform incompatible protocols between system components and the buffer. Futhermore, by communicating data through the state machines with queue, the system components which operate at different bit widths and at different frequencies can be easily connected.

In our interface architecture, system components(PE1 and PE2) in Figure 1 are directly connected to its corresponding state machines and will transfer data to other component through the state machines. The state machines are responsible for receiving(sending) data from(to) the corresponding system components and writing(reading) the data to(from) the queues. The operating frequency of the state machine will be the same as the corresponding system component so as to reduce synchronization overhead of protocols which occurs by operating at different frequency.
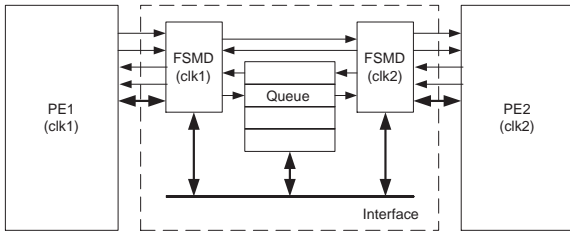

Figure 1. The interface architecture for components with incompatible protocols

3.1. Communication scheme

Since all transactions between system components are performed through queues which are controlled by state machines, we have to consider two inteface protocols, the protocol between state machines and queues and the protocol between system components and state machines. In other words, state machines should handle two interface protocols: one for system components and the other for queues. The state machines will perform following:

1. receive data from the producer(system component sending data) and store them into an internal buffer

2. write data from internal buffer into queue at proper time.

3. read data from the queue and store into it internal buffer to send them to the consumer(system component receiving data which the producer sent)

4. send data to the consumer

The bit width and depth of the queue should be determined for lossless data transfer. In order to reduce the number of transfers between state machines and the queue, the bit width of the queue is determined as follows:

$$bw_Q = \max[bw_s, bw_r]$$

where bw_s is bit width of the producer and bw_r is bit width of the consumer. If the bit width is selected to less than bw_Q, the state machines must write and read data more often than that of bw_Q.

The depth of queue will be determined to minimize the size of queue by following formula:

$$Q_n = \max(0, Q_{n-1} + (P_n - C_n))$$

where Q_n is the depth of queue in time n. P_n represents the amount of produced data in time n and C_n represents the amount of consumed data in time n. The depth of queue will be the maximum of Q_n.

The state machines will be responsible for merging and slicing the data to make them suitable for the queue. During the transfer, part of data will be temporarily stored in the state machines, which means the state machines should contain the internal buffer. The bit width of internal buffer will be maximum bit width of two communicating parties(same as bit width of the queue, bw_Q), which reduces the number of data transfers between the state machines and queue.

The interface protocol between state machines and queues will be fixed because the queue interface is predefined. But the interface protocol between system components and state machines will be varied depending on the protocol of system components.

Figure 2 shows the interface communication scheme between the state machines and a queue with a single I/O port. When two state machines share one queue and the data can be transfered bidirectionally, the handshaking between them is needed. Then, the producer must generate a signal to let the consumer read the data in the queue. Also, comsumer must send signal to let the producer know that he has read data. If two queues are used for storing data, or data transfers occur only in one direction, handshking protocol is no longer needed as shown shown in Figure 3. In addion, if the queue with separate I/O port is selected for storing data, the handshaking between state machines is not needed.
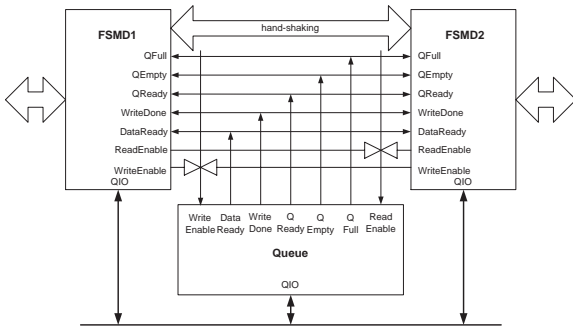
Figure 2. The interface communication scheme using one queue

Figure 3. The interface communication scheme using two queues

3.2. Queue model

Queue is frequently used to smoothen bursts in the requests for a service [Gaj97]. It stores the surplus data which will eventually be read in the same order it was written in. System components like processors, ASICs which send data to each other, in the sense that when data production momentarily exceeds the data consumption, queue must be inserted between the producer and the consumer. Of course, in such cases the data production rate cannot exceed the consumption rate indefinitely, since that would require an infinite queue. On the contrary, both rates on an average, must be the same. However, production and consumption bursts do occassionally occur, and the size of the queue determines how large a burst can be tolerated.

To facilitate data transfer between system components,

we make our own queues which are general enough to include the specific queues, distributed by various vendors. Our implemenation of a queue is shown in Figuire 4. In our implementation, a queue can have one or two I/O ports(`QIO` or `QIn` and `QOut`) for data. It also has several input control signals: `ReadEnable`, `WriteEnable`, and `Reset`. When `ReadEnable` is equal to 1, the queue will output the data which has been stored for the longest time, taking it from the front of queue. Similarly, when `WriteEnable` is equal to 1, the data will be added to the back of the queue. `ReadEnable` and `WriteEnable` are never equal to 1 at the same time.

Queue also has several control outputs which are used to control the producer and the consumer. When the queue is full, the signal `Full` will have a value of 1, which will warn the producer that any further data sent to the queue will be discarded. When `Empty` becomes 1, it warns the consumer that no data has yet arrived. When the producer(consumer) writes(read) the data in the queue, `QReady` become 1, which warns the other can not use the queue, because someone is using it. For queue with separate read and write port, `QReadReady` and `QWriteReady` are needed to prevent mulitple producers from accessing the queue at the same time.
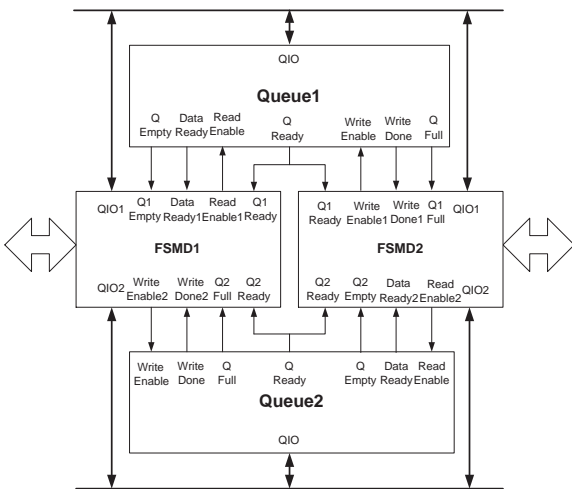
Our queue is implemented with a memory to store large data. The clock period of the queue is frequently less than the memory read access time. In this case, the consumer does not know when it can read data from the queue. To tackle this problem, `DataReady` signal is implemented. When `DataReady` is equal to 1, the queue generate the read data for the consumer. Similarly, when memory write time is longer than the clock period of the queue, the producer does not know when it must deassert control signal for writing data to queue. For this, `WriteDone` signal is implemented. When the `WriteDone` is equal to 1, the data are written to queue, and the consumer can deassert control signals. If read/write operation can be performed in one clock cycle, `DataReady` and `WriteDone` are not needed for implementation because they are useful for only multicycle read/write operation.

3.3. Queue Timinig Diagram

In order to generate a queue model from the memory timing constraints, we have to schedule the timing constraints based on given clock period of the queue. Given timing constraints of the memory and the clock period of the queue, queue generation reduces to the task of generating a state machine that implements the given queue functionality and satisfies the timing constraints. This requires scheduling of memory timing constraints into clock cycles such that no constraint is violated. Therefore, the FSMD implementation selects instances of the given timing ranges
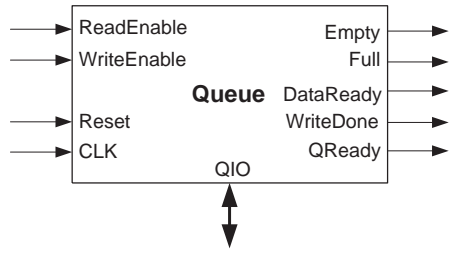
(a) Queue with a single I/O port
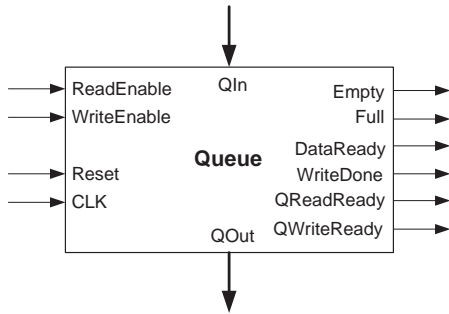
(b) Queue with two I/O ports

Figure 4. Queue block diagram

Write Timing of Queue with 1 I/O port

Read Timing of Queue with 1 I/O port

Figure 5. Timing diagram of a Queue with a single I/O port

based on the granularity given by the queue clock. Finally the queue description will be generated for integration in interface synthesis.

Figure 5 shows the read/write timing diagram of queue with a single I/O port, which contains 1 port memory, which is shown in Figure 4 (a). In this figure the MEM/ denotes the signals are ports of memory. For example MEM/Addr means the Addr port(Address port) of the memory. Figure 6 shows the read/write timing diagram of queue with 2 I/O ports which contains 2 port memory which is shown in Figure 4(b). We developed queue generation algorithm [SG02] to generate FSMD model for queue from timing specification of the given memory, which is beyond the scope of this report and refer to this report [SG02].

4. Protocol specification

In order to generate the interface automatically, the tool has to accept the information about protocols. The timing diagram information is informal and sometimes ambiguous for a tool to understand its meaning. So in the first place, protocols must be described in a form usable by the synthe-

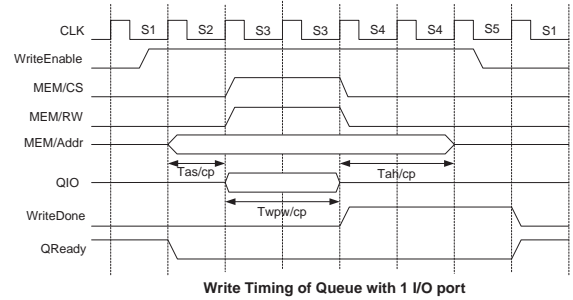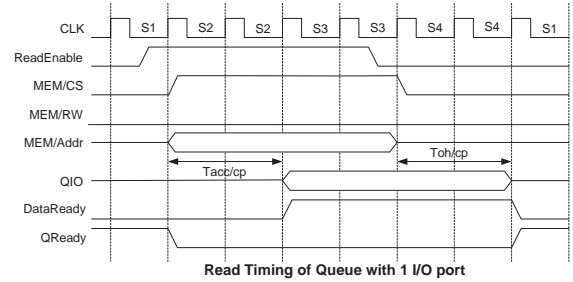sis tool and also suitable for simulation.

Both system component interface and bus protocol must be described in a form which provides complete information to match the protocols. This form would not only include the temporal information of signals but also the causal relationships between them with associated delay. At least, three aspects have to be included: interface port or bus wire information(name, bit width, direction), protocol sequeuece, and timing.

We propose a formal model, called *Protocol Sequence Graph*, PSG from now on, which can be used by synthesis algorithm, that captures the minimal necessary set of features representing the interface and its associated communication protocol.

4.1. Protocol sequence graph

Protocol sequence graph is designed for describing the detailed data transfer and synchronization operations of a communication protocol. In PSG, a protocol will be decomposed to several transactions, which are atomic communication operations. Each transaction can be represented by a PSG. Usually, an interface communication protocol involving two parties should be represented by two complementary PSGs: one for sending and the other is receiving. The interface protocol specified by a PSG may formally be described in terms of communication actions, which are data changes of signals:

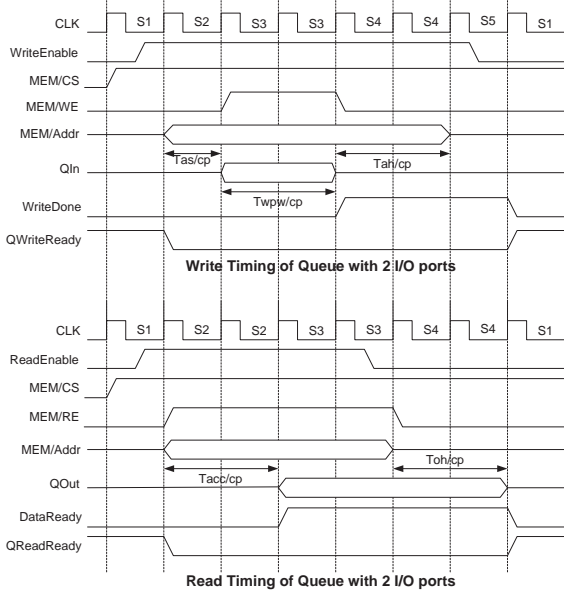**Definition 1** Let *A* be a basic action defined as:

4

Figure 6. Timing diagram of a Queue with two I/O ports

$$A = s \odot v$$

where s is signal, $\odot$ is operator, v is a value to be transferred

The value v can be one of followings:

- 0(level high), 1(level low), $\sim$(level toggling), x(don't care) and z(high impedence)

- +(rising edge) and −(falling edge)

- #(valid address/data signal) and ∗(invald address/data signal)

To represent relationships(conditions for input signals or assignments for output signals) between signals and values, we use four operators as shown:

- ?: check value of an input signal

- !: assign value to an output signal

- →: read address/data from a bus

- ←: write address/data to a bus

Each operator has its *dual* which can be viewed as a complementary operation. The *duality* of any of these actions can be found by obtaining the inverse relation between assignment statement and waiting statement of signal events.

The operator ? is dual to operator !, and operator → to operator ←. Specifically, actions can be divided into control related actions(?, !) and data related actions(→, ←).

Communication actions can be grouped if they have no timing relationships and can be executed at the same time. These communication actions are called *composite actions* which makes one node in PSG.

Definition 2 *Protocol Sequence Graph G(A, E)* is directed graph, where vertices *A* are *composite actions*, which are labeled with signal names combined with associated value and arcs *E* represent the causal relations and temporal relations between vertices. Each vertex of PSG is either labeled with conditions for input signals or assignments for output signals.

The PSG has two special no-action nodes: *source* and *sink*. The PSG becomes polar graph by introducing the source and sink node. The polar graph the single entry and single exit property, which makes easy to find start and end of protocol.

Usually specific timing information must be annotated in arcs besides the causal relationship information. The annotation arcs are used to denote the timing relations between two vertices of the PSG. The timing relations must include the propagation delay and timing constraint between two vertices and clock information. All timing annotations are denoted by (*min*, *max*) to express the interval between the minimum and the maximum time constraint or delay. The annotation used on input signal vertices can be interpreted as (*setup*, *hold*) to denote the setup and hold time constraints for the input condition relative to the local clock. For output signal vertices the annotations denote the propagation delays relative to the local clock.

We demonstrate how PSG can be used to model interface protocol using two examples: one is double handshaking protocol and the other is Queue interface protocol.

4.1.1 Double handshaking protocol

In this section, we will show how to make PGS from protocol specification of asynchronous protocol, double handshaking protocol, whose block diagram and timing diagram is shown in Figure 7 and in Figure 8 respectively. Write operation begins with the master(producer) initiating the transfer by putting `Addr` and `Data` out through the address bus and data bus, while asserting the `Req` signal and waiting for the `Ack` signal from the slave(consumer). As soon as the slave detects the `Req` signal, it loads the `Addr` and `Data` and asserts the `Ack`. The master can then deassert the `Req` signal while the `Addr` and `Data` signal can be invalidated. To begin next transaction, the slave has to deassert its `Ack` signal.

For read operation, communication begins with the master initiating the transfer by putting `Addr` out through the
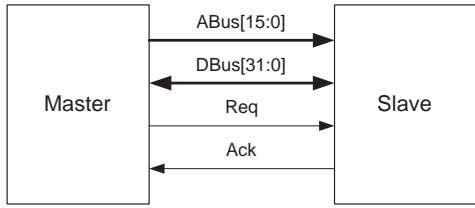
Figure 7. Block diagram of 4-phase double handshaking protocol

write cycle

Figure 8. Timing diagram of 4-phase double handshaking protocol

address bus, while asserting the `Req` signal and waiting for the `Ack` signal and valid data from the slave. As soon as the slave detect the `Req` signal, it writes data through data bus and asserts the `Ack`. After The master reads the data from the data bus, it deasserts the `Req` signal while the `Addr` and `Data` signal can be invalidated. To begin next transaction, the slave has to deassert its `Ack` signal.

The protocol specification and master side PSG of the double handshaking protocol is shown in Figure 9. The method `msend()` describes the master write protocol in SpecC, and the method `mrecv()` does the master read protocol. The corresponding PSGs are shown on the right in Figure 9. Timing constraints are specified for the events on the wires by enclosing the code driving and sampling the wires in a `do-timing` construct. The constraints are specified as ranges between the labels in the code sequence. For example, according to the protocol timing diagram the range between T1 and T2 is from 5 to 8 time units. The dot node at the top of PSG is source node and sink node is at the bottom of PSG. Similar to the master side of the protocol, the slave side can be specified as shown in Figure 10). The protocol of the slave side is dual of that of master side because the slave side will be the counterpart of the master interface. Therefore, the slave side protocol will be generated automatically through interface synthesis.

4.1.2 Queue interface protocol

In this section, we will show how to make PGS from protocol specification of synchronous protocol, queue interface protocol which will be used in our interface architecture, whose timing diagram is shown in Figure 5. The queue is slave during transaction between system components.

Write operation begins with the master(producer) initiating the transfer by putting the data through `QIO` while asserting the `WriteEnable` signal and waiting for the `WriteDone` signal from the queue(consumer). As soon as the queue detect the `WriteEnable` signal, it loads the data through data bus and asserts the `WriteDone`. The master can then deassert the `WriteEnable` signal while the data bus can be invalidated. To begin next transaction,
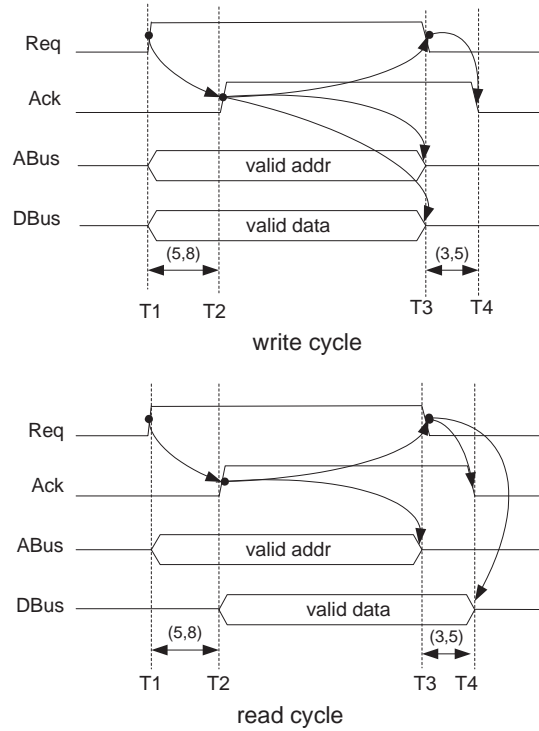
the queue has to deassert its `WriteDone` signal.

For read operation, communication begins with the master initiating the transfer by while asserting the `ReadEnable` signal and waiting for the `DataReady` signal and valid data from the queue. As soon as the queue detects the `DataReady` signal, it writes the data through data bus and asserts the `DataReady`. After The master reads the data from the data bus, it deasserts the `ReadEnable` signal. To begin next transaction, the queue has to deassert its `DataReady` signal.

The protocol specification for synchronous protocol is represented by FSMD construct in SpecC. Each state has actions and transition depending on the conditions of inputs and current state, which is shown in Figure 11. The method `Qwrite()` describes the queue write protocol in SpecC, and the method `Qread()` does the queue read protocol in terms of the queue. The corresponding PSGs are shown on the right in Figure 11. Similar to the Queue interface protocol, the master side of the queue interface protocol can be specified as shown in Figure 12. The protocol of the master side is dual of that of the queue because the queue interface is counterpart of the master interface. Therefore, the master side protocol will be generated automatically from our queue model through interface synthesis.
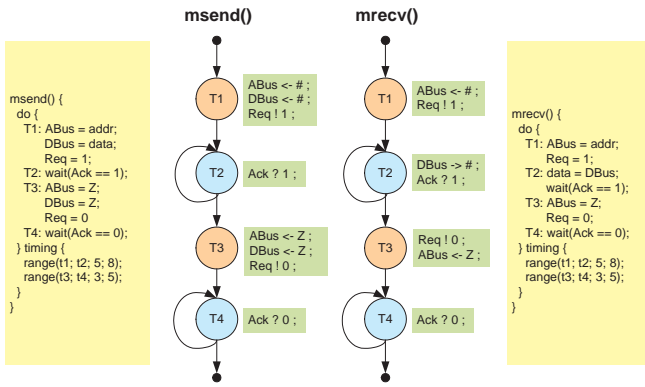
Figure 9. Protocol specification for double handshaking protocol(master)

Figure 11. Protocol specification for Queue interface protocol

Figure 10. Protocol specification for double handshaking protocol(slave)
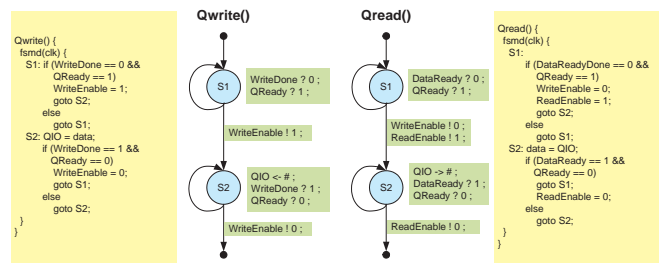
Figure 12. Protocol specification for Queue interface protocol(master)

5. Interface synthesis

Interface synthesis process is intended to insert interface transducers between system components with different protocols. The interface formed when duals are found for each action in original protocol specifications and are used to form the new interface behavior, first for the sending protocol, then, for the receiving protocol. In this way, we allow the capture of data from one component, the FIFO queue of transmitted data and the transmission to the other component.

In this section, we will demonstrate the interface synthesis algorithm which generates our proposed interface architecture.

5.1. Problem definition

Given:

1. Protocol descriptions of two communicating parties(producer and consumer)

2. Bit width and size for the selected memory

3. Clock period T_{Qclk} of the queue

Determine:

1. FSMDs for state machines

2. FSMD for the queue

Conditions:

1. Timing constraints are met.

5.2. Algorithm for interface synthesis

Algorithm 1 shows the interface synthesis algorithm from given protocol specifications and clock period of the selected queue. In this algorithm, there are two major tasks: MakeDual() and Schedule(). The method MakeDual transforms the protocol sequence graph of original protocol specification to the corresponding dual protocol sequence graph, which can be done by replacing the operators in actions with their duals.

The method Schedule() generates the FSMD from the PSG based on the producer and the consumer clocks.

Actually, the PSG from the synchronous protocol specification is not needed to be scheduled because its PSG is already a FSMD. Therefore, the PSG from the asynchronous protocol specification will be scheduled based on the selected producer and consumer clocks. In other word, the scheduling of PSG selects instances of the given timing ranges based on the granulairty given by the component clock such that no timing constraint is violated. During the scheduing of PSG, some action nodes will be collapsed into a complex action node based on causal relationship. For example, in `msend()` in 9, two nodes `T1` and `T2` can be collapsed into one state because the actions in `T1` will not be changed until `T2`. In the same way, two nodes `T3` and `T4` can be collapsed into one state.

The method `GenerateQueue()` will generate the queue based on the selected memory and the clock of the queue [SG02]. The generated producer interface FSMD, consumer interface FSMD and queue interface FSMD should be collapsed into a single FSMD to obtain interface FSMD. According to the bit width of the producer/consumer interface and bit width of the selected queue, the producer/consumer interface FSMD will be replicated by $\lceil \frac{bw_Q}{bw_S} \rceil$.

The method `AddFSMD()` will collapse the producer and queue interface FSMDs($FMSD_{Si}$, $FSMD_{Qi}$) into the transducer interface FSMD for the producer($FSMD_{TS}$). In the same way, the consumer and queue interface FSMD($FSMD_{Ri}$, $FSMD_{Qi}$) will collapse into the transducer interface FSMD for the consumer. Finally we have two FSMDs for transducer: the producer interface FSMD and the consumer interface FSMD in the tranducer.

6. Examples

We select the parity encoder as an example to show our interface synthesis approach. The parity encoder is utilized as error detection and error correction coding in data communication. The operation of parity encoder is shown in Figure 13. 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(`Ones`) and even parity checker(`Even`). The former computes the number of ones for the 32 bit wide data, and the later generates even parity bit by examining the output(`ocount`) of one's counter. These two behaviors communicate through the channel(`MasterBus` and `SlaveBus`), which contains the interface protocol specification for even parity checker and one's counter.

This communication model will be refined into RTL model through protocol inlining and RTL synthesis of the behaviors and interface generation for channels [GZD$^+$00]. The communicating behaviors in the parity encoder can be

---

Algorithm 1 GenerateInterface($PSG_S(A,V)$, $PSG_R(A,V)$, T_{Qclk})

    // Generate Queue FSMD
    $PSG_Q(A,V)$ = GenerateQueue(T_{Qclk});
    // get the dual of producer PSG
    $PSG_{Si}(A,V)$ = MakeDual($PSG_S(A,V)$);
5: $PSG_{Ri}(A,V)$ = MakeDual($PSG_R(A,V)$);
    $PSG_{Qi}(A,V)$ = MakeDual($PSG_Q(A,V)$);
    // schedule the PSG based on the clock of the producer.
    $FMSD_{Si}$ = Schedule(PSG_{Si});
    // add the producer FSMD($FSMD_{Si}$) to interface FSMD
10: **for** i = 1 to $\lceil \frac{bw_Q}{bw_S} \rceil$ **do**
      AddFSMD($FSMD_{TS}$, $FSMD_{Si}$);
    end for
    // schedule the PSG based on T_{Qclk}
    $FMSD_{Qi}$ = Schedule(PSG_{Qi}, T_{Qclk});
15: // add the queue FSMD($FSMD_{Qi}$) to interface FSMD
    AddFSMD($FSMD_{TS}$, $FMSD_{Qi}$);
    // add the queue FSMD($FSMD_{Qi}$) to interface FSMD
    AddFSMD($FSMD_{TR}$, $FMSD_{Qi}$);
    $FMSD_{Ri}$ = Schedule(PSG_{Ri});
20: // add the consumer FSMD($FSMD_{Ri}$) to interface FSMD
    for i = 1 to $\lceil \frac{bw_R}{bw_Q} \rceil$ **do**
      AddFSMD($FSMD_{TR}$, $FSMD_{Ri}$);
    end for

---

implemented by various system components with different protocols. We selected 4 protocols: double handshaking protocol, ColdFire processor interface, ARM9TDMI processor interface and TMS320C50 DSP processor interface. The parity encoder will be implemented by the composition of 4 protocols which will be explained in following sections. The abstract FSMD description of the synthesized parity encoder will be shown in Figure 14. In this FSMD description, each node corresponds to interface method of the selected protocol which will be described in more detail in following sections. The FSMD is generated through protocol inlining and scheduling of RTL synthesis from non-inlined communication model. The FSMD for the even parity checker(`Even`) is composed of 2 execution parts(`EX1` and `EX2`), a master send part(`msend`) to send data through the bus and a master receive part(`mrecv`) to receive data from the bus.

In the same way, the FSMD for one's counter consists of a execution part(`EX1`) and a master send part(`msend`) and a master receive part(`mrecv`). The generated transducer will be composed of 2 communicating FSMDs with one queue: the one(`FMSD1`) is for interfacing with the even parity checker and the other(`FSMD2` is for interfacing with the one's counter. The `FSMD1` consists of a slave receive part(`srecv`) to get data from the master, a slave send part to
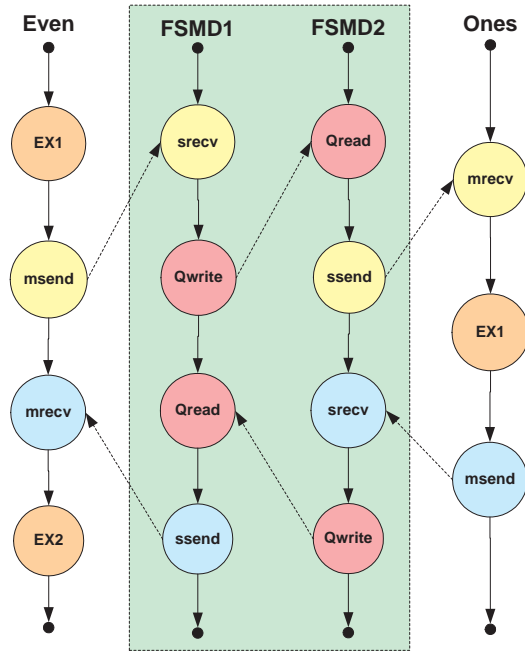
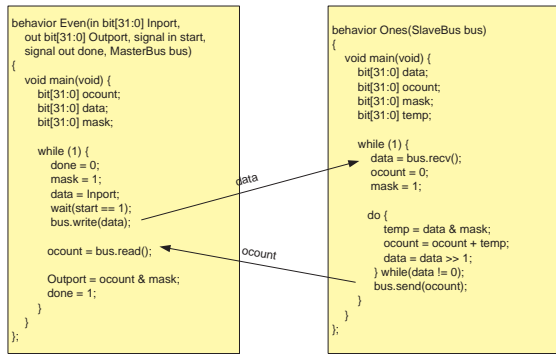Figure 14. FSMD for parity encoder(double handshaking protocol)

Figure 13. Non-inlined communication model of parity encoder

send data to master(ssend), a queue write part(Qwrite) to store data from the master to the queue and a queue read part(Qread) to get data from the queue. The FSMD2 has the same composition as the FSMD1 except for their ordering.

In this figure, the dotted line represents the transaction of data between system components and interface FSMDs. For example, the data(data) sent from the even parity checker to ones counter will be transfered through msend in Even, srecv and Qwrite in FSMD1, Qread and msend in FSMD2, finally srecv in Ones. The data ocount from one's counter to even parity checker will be transferred

through the same path in the reverse direction.

6.1. ColdFire processor interface

The even parity checker(Even) is implemented by Cold-Fire processor [Inc01] and the one's counter(Ones) is implemented by ASIC using double handshaking protocol as shown in Figure 15. Though ColdFire processor has 32 bit wide data bus, we reduced it to 8 bit wide in order to generate transducer with different bit widths for experiment. Since they are connected by incompatible interface protocols, the transducer should be introduced to transfer data. ColdFire Processor interface protocol, whose timing diagram is shown in Figure 16. The bus transaction in ColdFire processor interface protocol is split into two phases: address phase and data pahse. During address phase, the address(MADDR) and attribute signal(MRWB) is driven. The MRWB output signal is provided to indicate whether the current cycle is a read(low) or a write(high). The address phase signal(MAPB) is asserted to show that the bus is in address phase. During the data phase, data phase signal(MDPB) is asserted until the bus cycle terminates with a transfer acknowledge(MTAB). On a write cycle, the write data bus(MWDATA) is driven for the entire data phase. On a read cycle, the bus master samples the read data bus(MRDATA) concurrently with MTAB at the rising clock edge.

For memory accesses which requires more than one cycle, the processor can be halted using MTAB. This signal
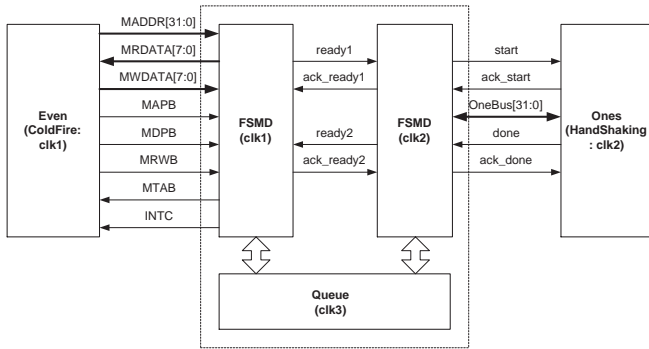
Figure 15. Transaction between ColdFire processor and handshaking protocol

halts the processor during the data phase. The MTAB signal should be driven high at the end of data phase if read/write operation is done.

The protocol specification for synchronous protocol is represented by FSMD construct in SpecC. Each state has actions and transition depending on the conditions of inputs and current state, which is shown in Figure 17. The method mwrite() describes the write protocol in SpecC, and the method mread() does the read protocol. The corresponding PSGs are shown on the right in Figure 17. Through the interface synthesis, the counterpart of the ColdFire processor interface can be derived by extracting the dual of original protocol, which is shown in Figure 18. The ColdFire processor interface protocol is synchronous, then scheduling of the PSG is to make each action become a node in the PSG to corresponding state in FSMD. Generated interface FSMDs are shown in Figure 19. In this figure, two FSMDs(FSMD2 and Ones) are not changed because Ones uses the same double handshaking interface protocol as the example in previous section. The state S1 and S2 is Cold-Fire master bus interface protocol in Even FSMD. The corresponding states in transducer are S0 and S1 in FSMD1. The state R1 and R2 in FSMD1 are dual states of R1 and R2 in Even FSMD.

6.2. ARM9TDMI processor interface

The even parity checker(Even) is implemented by ASIC using double handshaking protocol and the one's counter(Ones) is implemented by ARM9TDMI [Inc00] processor as shown in Figure 20. The ARM9TDMI has separate instruction and data interface. This allows concurrent instruction and data accesses, and greatly reduces the CPI of the processor. For optimal performance, single cycle memory accesses for both interfaces are required, although the core can be wait-stated for non-sequential accesses, or
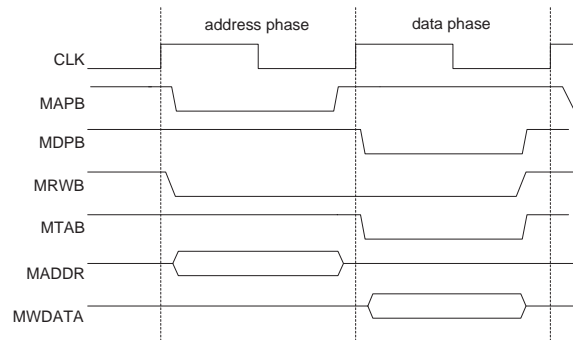
Write Timing
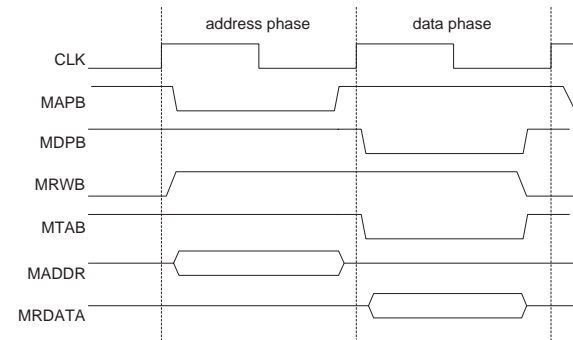
Read Timing

Figure 16. Timing diagram of ColdFire processor interface protocol

slower memory systems.

For both instruction and data interfaces, the ARM9TDMI process core uses pipelined addressing. The address and control signals are generated the cycle before the data transfer takes place, giving any decode logic as much advance notice as possible. All memory access are generated from GCLK.

For each interface there are types of memory access: non-sequential, sequential, internal, coprocessor transfer for the data interface. These accesses are determined by InMREQ and ISEQ for the instruction interface, and by DnMREQ and DSEQ for the data interface.

For memory accesses which requires more than one cycle, the processor can be halted by using nWAIT. This signal halts the processor, including both the instruction and data interface. The nWAIT signal should be driven low by the end of phase 2 in which GCLK is high to stall data interfaces(it is inverted and ORed with GCLK to stretch the internal processor clock). The nWAIT signal must only change during phase 2 of GCLK.

The timing diagram for ARM9TDMI processor interface protocol is shown in Figure 21. Data transfers take place in the memory stage of the pipeline. DnRW indicates the di-
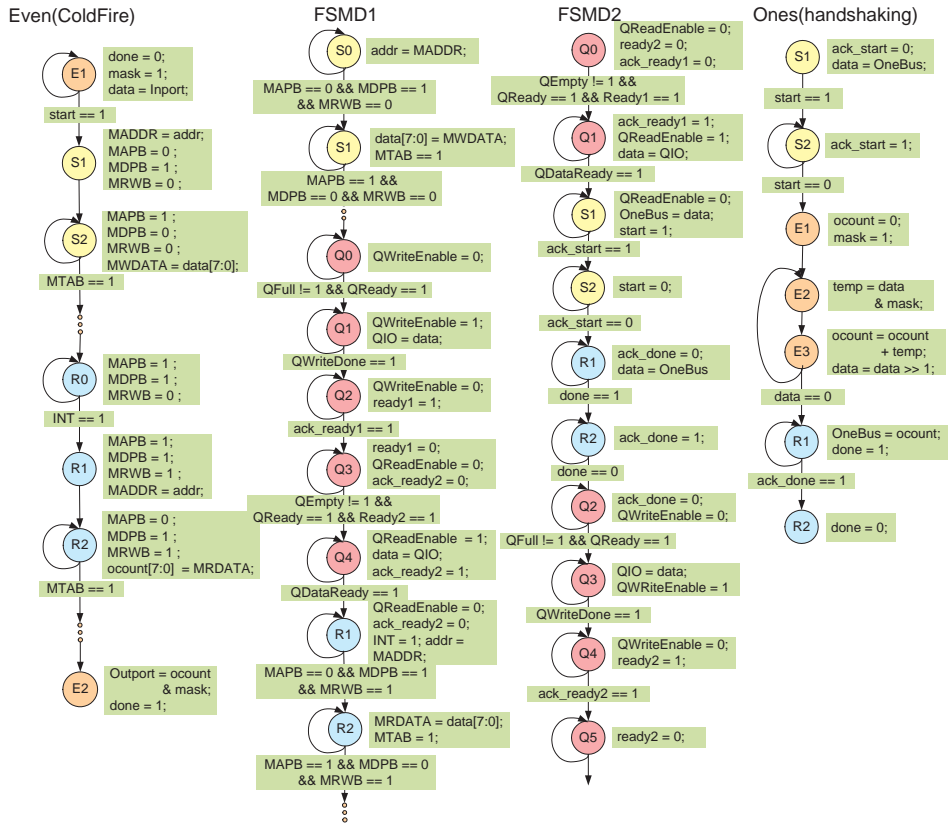
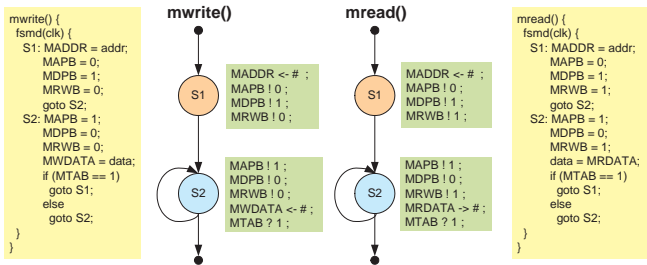**Figure 19. FSMD for parity encoder(ColdFire Processor)**

Even(ColdFire)

E1: done = 0; mask = 1; data = Inport;
start == 1
S1: MADDR = addr; MAPB = 0; MDPB = 1; MRWB = 0;
S2: MAPB = 1; MDPB = 0; MRWB = 0; MWDATA = data[7:0];
MTAB == 1
R0: MAPB = 1; MDPB = 1; MRWB = 0;
INT == 1
R1: MAPB = 1; MDPB = 1; MRWB = 1; MADDR = addr;
R2: MAPB = 0; MDPB = 1; MRWB = 1; ocount[7:0] = MRDATA;
MTAB == 1
E2: Outport = ocount & mask; done = 1;

FSMD1

S0: addr = MADDR;
MAPB == 0 && MDPB == 1 && MRWB == 0
S1: data[7:0] = MWDATA; MTAB == 1;
MAPB == 1 && MDPB == 0 && MRWB == 0
Q0: QWriteEnable = 0;
QFull != 1 && QReady == 1
Q1: QWriteEnable = 1; QIO = data;
QWriteDone == 1
Q2: QWriteEnable = 0; ready1 = 1;
ack_ready1 == 1
Q3: ready1 = 0; QReadEnable = 0; ack_ready2 = 0;
QEmpty != 1 && QReady == 1 && Ready2 == 1
Q4: QReadEnable = 1; data = QIO; ack_ready2 = 1;
QDataReady == 1
R1: QReadEnable = 0; ack_ready2 = 0; INT = 1; addr = MADDR;
MAPB == 0 && MDPB == 1 && MRWB == 1
R2: MRDATA = data[7:0]; MTAB = 1;
MAPB == 1 && MDPB == 0 && MRWB == 1

FSMD2

Q0: QReadEnable = 0; ready2 = 0; ack_ready1 = 0;
QEmpty != 1 && QReady == 1 && Ready1 == 1
Q1: ack_ready1 = 1; QReadEnable = 1; data = QIO;
QDataReady == 1
S1: QReadEnable = 0; OneBus = data; start = 1;
ack_start == 1
S2: start = 0;
ack_start == 0
R1: ack_done = 0; data = OneBus
done == 1
R2: ack_done = 1;
done == 0
Q2: ack_done = 0; QWriteEnable = 0;
QFull != 1 && QReady == 1
Q3: QIO = data; QWRiteEnable = 1
QWriteDone == 1
Q4: QWriteEnable = 0; ready2 = 1;
ack_ready2 == 1
Q5: ready2 = 0;

Ones(handshaking)

S1: ack_start = 0; data = OneBus;
start == 1
S2: ack_start = 1;
start == 0
E1: ocount = 0; mask = 1;
E2: temp = data & mask;
E3: ocount = ocount + temp; data = data >> 1;
data == 0
R1: OneBus = ocount; done = 1;
ack_done == 1
R2: done = 0;

Figure 17. Protocol specification for ColdFire processor interface

```
mwrite() {
  fsmd(clk) {
    S1: MADDR = addr;
        MAPB = 0;
        MDPB = 1;
        MRWB = 0;
        goto S2;
    S2: MAPB = 1;
        MDPB = 0;
        MRWB = 0;
        MWDATA = data;
        if (MTAB == 1)
          goto S1;
        else
          goto S2;
  }
}
```

mwrite()
S1: MADDR <- # ; MAPB ! 0 ; MDPB ! 1 ; MRWB ! 0 ;
S2: MAPB ! 1 ; MDPB ! 0 ; MRWB ! 0 ; MWDATA <- # ; MTAB ? 1 ;

mread()
S1: MADDR <- # ; MAPB ! 0 ; MDPB ! 1 ; MRWB ! 1 ;
S2: MAPB ! 1 ; MDPB ! 0 ; MRWB ! 1 ; MRDATA -> # ; MTAB ? 1 ;

```
mread() {
  fsmd(clk) {
    S1: MADDR = addr;
        MAPB = 0;
        MDPB = 1;
        MRWB = 1;
        goto S2;
    S2: MAPB = 1;
        MDPB = 0;
        MRWB = 1;
        data = MRDATA;
        if (MTAB == 1)
          goto S1;
        else
          goto S2;
  }
}
```
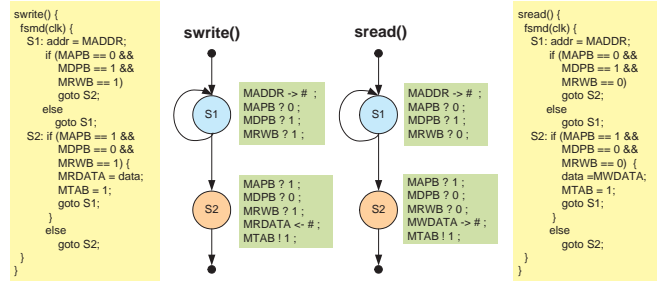
Figure 18. Dual of the ColdFire processor interface protocol

```
swrite() {
  fsmd(clk) {
    S1: addr = MADDR;
        if (MAPB == 0 &&
            MDPB == 1 &&
            MRWB == 1)
          goto S2;
        else
          goto S1;
    S2: if (MAPB == 1 &&
            MDPB == 0 &&
            MRWB == 1) {
          MRDATA = data;
          MTAB = 1;
          goto S1;
        }
        else
          goto S2;
  }
}
```

swrite()
S1: MADDR -> # ; MAPB ? 0 ; MDPB ? 1 ; MRWB ? 1 ;
S2: MAPB ? 1 ; MDPB ? 0 ; MRWB ? 1 ; MRDATA <- # ; MTAB ! 1 ;

sread()
S1: MADDR -> # ; MAPB ? 0 ; MDPB ? 1 ; MRWB ? 0 ;
S2: MAPB ? 1 ; MDPB ? 0 ; MRWB ? 0 ; MWDATA -> # ; MTAB ! 1 ;

```
sread() {
  fsmd(clk) {
    S1: addr = MADDR;
        if (MAPB == 0 &&
            MDPB == 1 &&
            MRWB == 0)
          goto S2;
        else
          goto S1;
    S2: if (MAPB == 1 &&
            MDPB == 0 &&
            MRWB == 0) {
          data = MWDATA;
          MTAB = 1;
          goto S1;
        }
        else
          goto S2;
  }
}
```

rection of transfer, low for reads and high for writes. The signal becomes valid at approximately the same time as the data address bus. For read cycle, DDIN must be driven with valid data for the falling edge of GCLK at the end of phase 2. For write cycle, data will become valid in phase 1 in which GCLK is low, throughout phase 2. Figure 22 shows the protocol specification and PSG for ARM9TDMI processor interface. The method mwrite() describes the write interface protocol in SpecC, and the method mread() does the read interface protocol. The corresponding PSGs are shown

on the right in Figure 22. Through the interface synthesis, the counterpart of the ARM9TDMI processor interface can be derived by extracting the dual of original protocol, which is shown in Figure 23. The ARM9TDMI processor interface protocol is synchronous, then scheduling of the PSG is to make each action become a node in the PSG to corresponding state in FSMD. Generated interface FSMDs are shown in Figure 24. In this figure, two FSMDs(FSMD1 and Even) are changed into double handshaking protocol
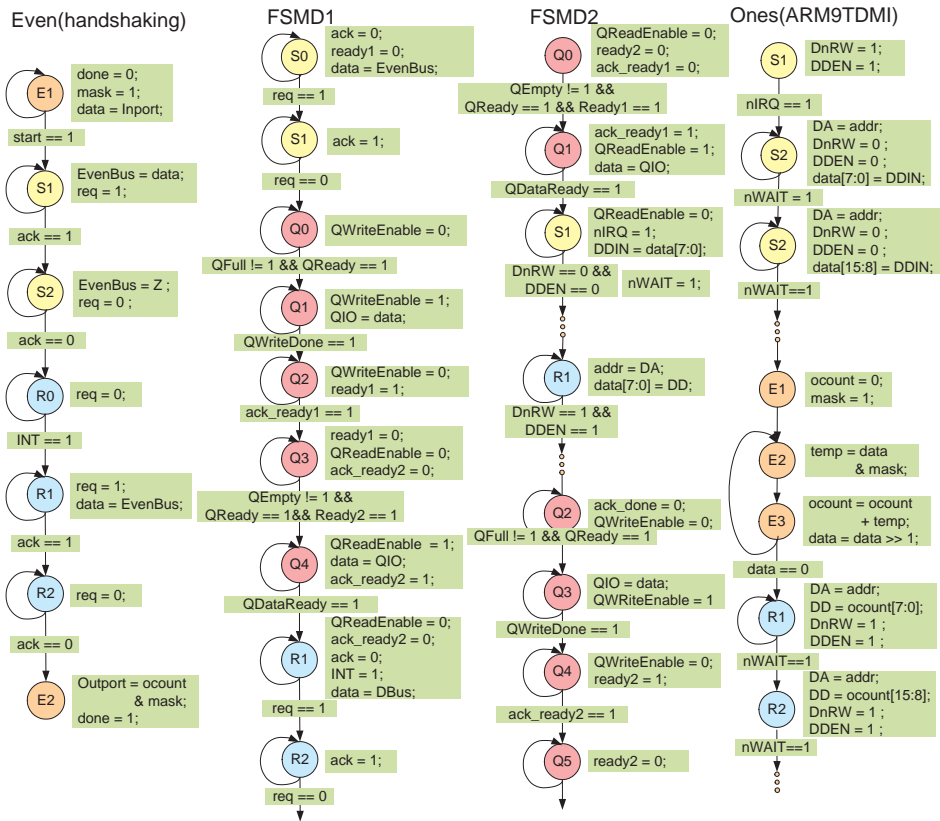
11

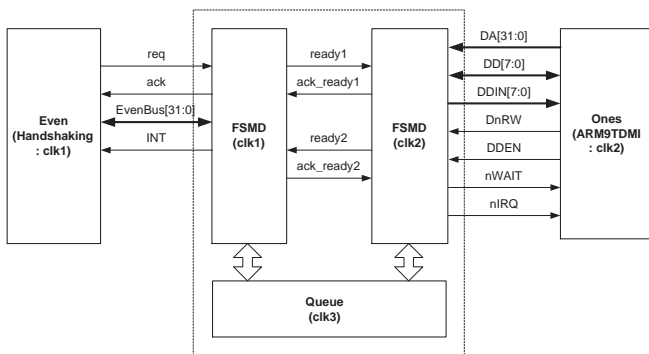Figure 24. FSMD for parity encoder(ARM9TDMI)
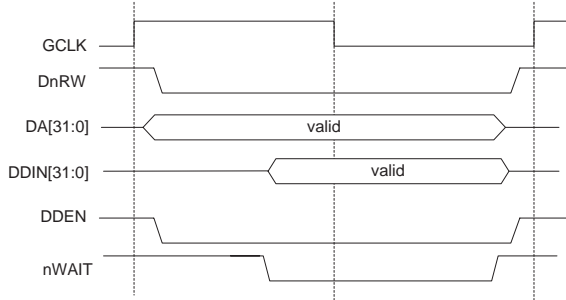
6.3. TMS320C50 DSP processor

The even parity checker(Even) is implemented by TMS320C50 DSP processor [Inc98] and the one's counter(Ones) is implemented by ColdFire processor as shown in Figure 25. The TMS320C50 supports a wide range of system interfacing requirements. Program, data, and I/O address spaces provide interface to memory and I/O, maximizing system throughput. The full 16-bit address and data bus, along with the PS, DS, and IS space select signals, allow addressing of 64K 16-bit words in each of the three spaces. I/O design is simplified by having I/O treated the same way as memory. I/O devices are mapped into the I/O address space using the processor external address and data buses in the same manner as memory-mapped devices.

The TMS320C50 external parallel interface provides various control signals to facilitate interfacing to the device. The timing diagram for TMS320C50 DSP processor interface is shown in Figure 26. The R/W output signal is provided to indicate whether the current cycle is a read or a write. The STRB output signal provides a timing reference for all external cycles. For convenience, the device also provides the RD and the WE output signals, which indi-

Figure 20. Transaction between ARM9TDMI processor and handshaking protocol

with 32-bit wide bus. The state S1 is the read protocol of ARM9TDMI processor interface in Ones FSMD. The corresponding state in transducer are S2 in FSMD2. The state R1 in FSMD2 are dual state of R1 in Ones FSMD.
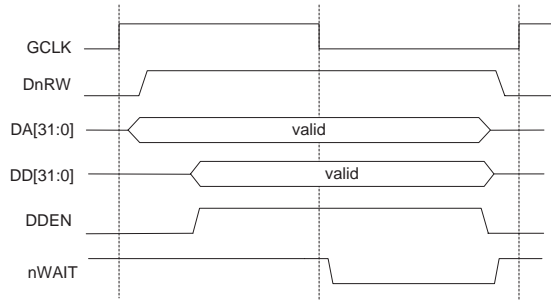
Figure 21. Timing diagram of ARM9TDMI processor interface protocol

Figure 22. Protocol specification for ARM9TDMI processor interface
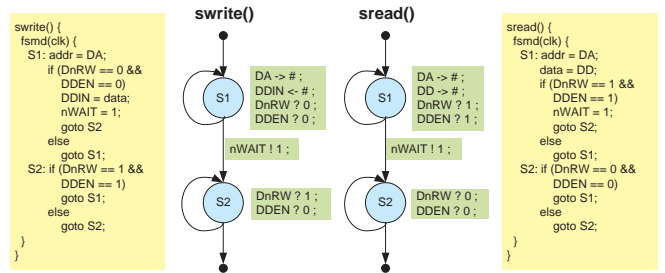
Figure 23. Dual of ARM9TDMI processor interface protocol

cate a read and a write cycle, respectively, along with timing information for those cycles. The availability of these signals minimizes external gating necessary for interfacing external devices to the TMS320C50 interface to memory and I/O devices of varying speeds is accomplished by using the READY line. When transactions are made with slower devices, the TMS320C50 processor waits until the other device completes its function and signals the processor via the READY line. Once a ready indication is provided back to the TMS320C50 from the external device, execution continues. The bus request(BR) signal is used in conjunction with the other TMS320C50 interface signals to arbitrate external global-memory accesses. Global memory is external data-memory space in which the BR signal is asserted at the beginning of the access. When an external global-memory device receives the bus request, the external device responds by asserting the READY signal after the global memory access is arbitrated and the global access is completed.

Figure 27 shows the protocol specification and PSG for TMS320C50 DSP processor interface. The method mwrite() describes the write interface protocol in SpecC, and the method mread() does the read interface protocol. The corresponding PSGs are shown on the right in Figure 27. Through the interface synthesis, the counterpart of the TMS320C50 DSP processor interface can be derived by extracting the dual of original protocol, which is shown
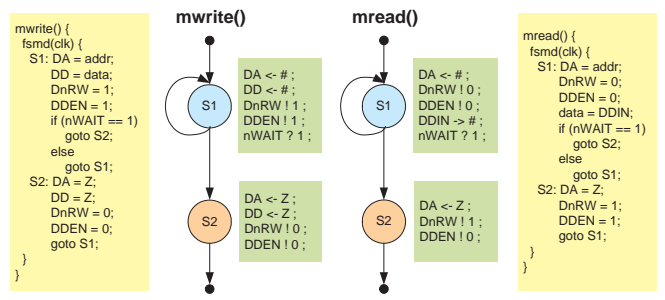
in Figure 28. The TMS320C50 DSP processor interface protocol is synchronous, then scheduling of the PSG is to make each action become a node in the PSG to corresponding state in FSMD. Generated interface FSMDs are shown in Figure 29. In this figure, two FSMDs(FSMD2 and Ones) are changed into ColdFire processor with 32-bit wide bus. The state S1 is the read protocol of ARM9TDMI processor interface in Even FSMD. The corresponding states in transducer are S0 in FSMD1. The state R1 in FSMD1 are dual state of R1 in Ones FSMD.

7. Conclusion and future work

This report shows our queue-based interface architecture, which is general enough to accommodate any target interface. In our architecture, queues are used to smoothen the burst data transfer requests and to transform incompatible protocols. Specifically, protocols were captured in SpecC language, the protocol specification must be refined into transducer and component interface for later synthesis. Then, method to generate system interface from the protocol specification has been described. Future work will include a detailed analysis and experiment with protocol and transducer generation, namely optimization techniques to allow for higher performance during communication. Fi-
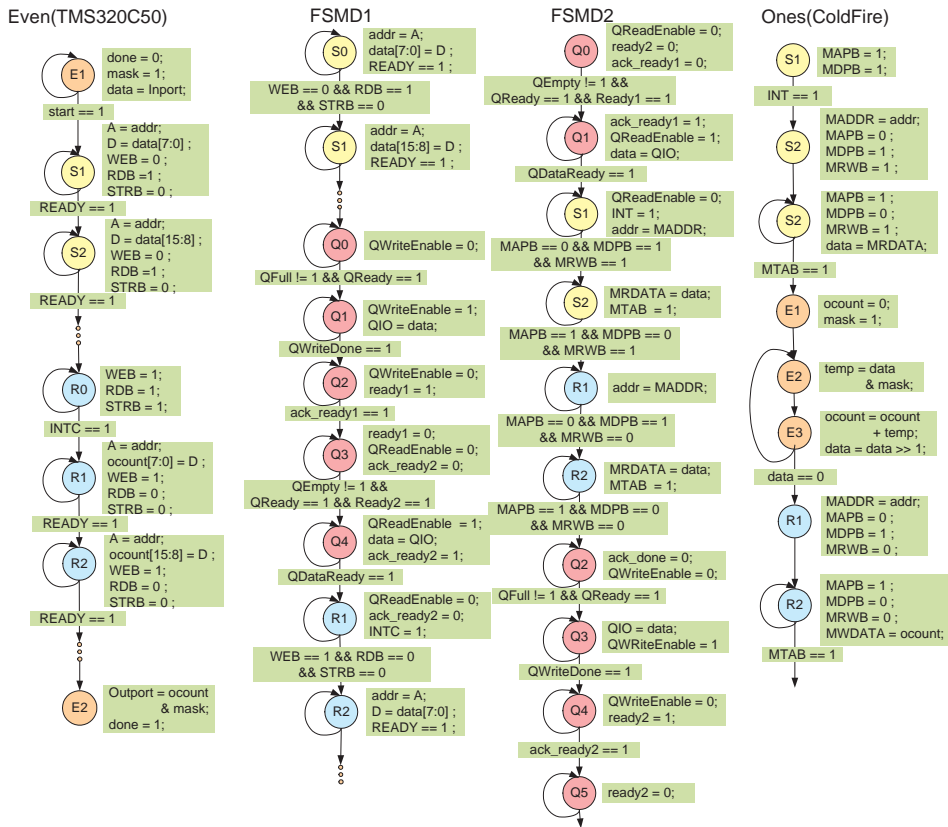
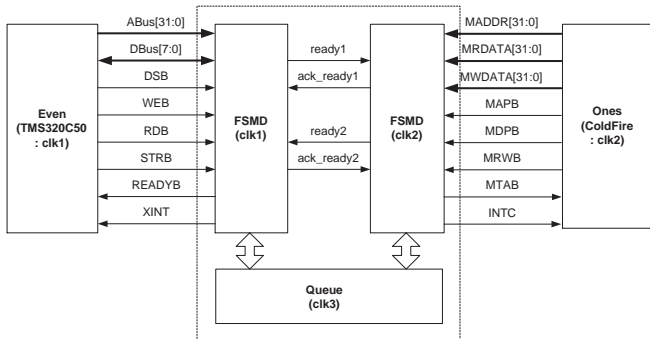Figure 29. FSMD for parity encoder(TMS320C50)

Figure 25. Transaction between TMS320C50 DSP processor and ColdFire processor

nally, this work is intended to be integrated into system design methodology under development at CADLAB of University of California, Irvine.

References

[BK87]   Gastano Borriello and Randy Katz. Synthesis and optimization of interface transducer logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 274–277, November 1987.

[Gaj97]   Daniel Gajski. *Principles of Digital Design*. Prentice Hall, 1997.

[GZD+00] Daniel Gajski, Jiwen Zhu, Rainer Domer, Andreas Gerstlauer, and Suqing Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[Inc98]   Texas Instruments Inc. TMS32C5x User's Guide, June 1998.

[Inc00]   ARM Inc. ARM9TDMI Technical Reference Manual, March 2000.

[Inc01]   Motorola Inc. ColdFire CF4e Core User's Manual, June 2001.
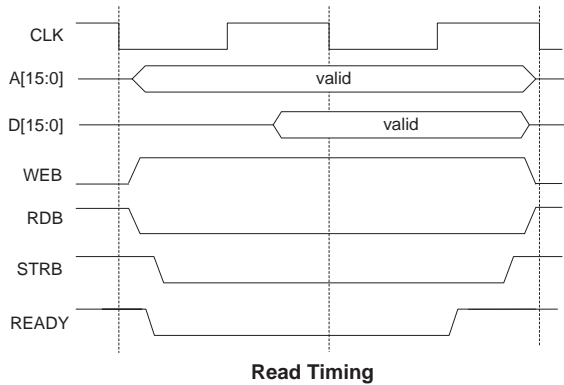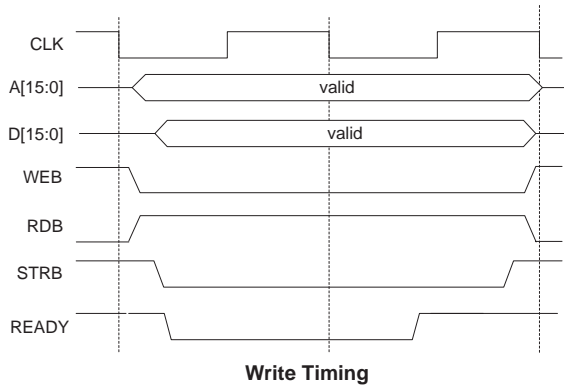
Write Timing

Read Timing

Figure 26. Protocol specification of TMS320C50 DSP interface protocol
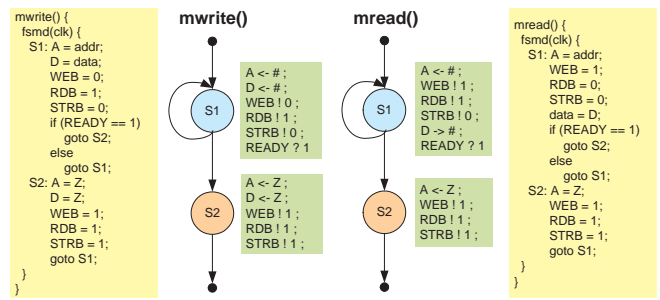

Figure 27. Protocol specification for TMS320C50 DSP interface
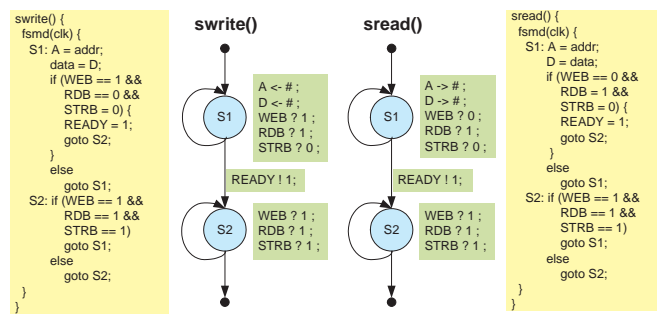

Figure 28. Dual of TMS320C50 DSP interface protocol

[LV94]   Bill Lin and Steven Vercauteren. Synthesis of concurrent system interface modules with automatic protocol conversion generation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 101–108, November 1994.

[Mis01]   International Technology Roadmap for Semiconductors, http://public.itrs.net/, 2001.

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

[PCPK00] Bong-Il Park, Hoon Choi, In-Cheol Park, and Chong-Min Kyung. Synthesis and optimization of interface hardware between ip's operating at different clock frequencies. In *Proceedings of the International Conference on Computer Design*, pages 519–524, June 2000.

[PRSV98] Roberto Passerone, James A. Rowson, and Alberto Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the Design Automation Conference*, pages 8–13, June 1998.

[SG02]   Dongwan Shin and Daniel Gajski. Queue Generation Algorithm for Interface Synthesis. Technical Report ICS-TR-02-03, University of California, Irvine, February 2002.

[SM98]   James Smith and Giovanni De Micheli. Automated composition of hardware components. In *Proceedings of the Design Automation Conference*, pages 14–19, June 1998.