Queue Generation Algorithm for Interface Synthesis

Dongwan Shin and Daniel Gajski

Technical Report CECS-02-12
April 11, 2002

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

{dongwans,gajski}@cecs.uci.edu

Queue Generation Algorithm for Interface Synthesis

Dongwan Shin and Daniel Gajski

Technical Report CECS-02-12
April 11, 2002

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

{dongwans,gajski}@cecs.uci.edu

Abstract

In system-on-a-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. In order to automate design reuse, methods for combining system components with incompatbile communication protocols must be developed because a large portion of integration is devoted to designing the interfaces between interacting components. In this report, we propose the interface architecture in which queues are used for data transfer between components with incompatible protocols and describe the algorithm for generating syntheisizable RTL description of queues using arbitrary memories.

Contents

List of Figures

Queue Generation Algorithm for Interface Synthesis

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

Abstract

In system-on-a-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. In order to automate design reuse, methods for combining system components with incompatbile communication protocols must be developed because a large portion of integration is devoted to designing the interfaces between interacting components. In this report, we propose the interface architecture in which queues are used for data transfer between components with incompatible protocols and describe the algorithm for generating syntheisizable RTL description of queues using arbitrary memories.

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. This increasing complexity of VLSI design and time to market pressures on the complex system-on-chip(SOC) have forced designers to consider reuse of Intellectual Property(IP) blocks [Mis01]. Since most IPs are proivided by different vendors, and they have different interface schemes, and different data rates, the combining these components is an error-prone task and the most important part of system integration.

The basic goal of an interface syntheis is to generate interfaces between incompatible components. Data could be transferred at different bit width, operating frequency, data rates. In this report, we propose novel queue-based interface scheme, which is general enough to accomodate any component protocols.

In order to implement queue-based interface architecture, the **canonical model** of a queue must be defined. It will reduce the design space of implementing various queues and interfaces. To define the canonical queue model which can contain various memories, we take a look at at timing constraints of various memory organization. Also, we introduce an algorithm which generates synthesizable RTL description from the protocol specification using the canonical queue model.

The rest of this report is organized as follows: section 2 describes our interface architecture and canonical model of queue. Section 3 takes a closer look at the queue architecture for different types of memories and queue generation algorithm. Section 4 shows the generated SpecC codes by queue generation tool. Section 5 concludes this report with a brief summary and future work.

2. The 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(FIFO queue) to smoothen the burst data transfer requests and two FSMDs(Finite State Machine with Data) to queue and unqueue data.

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 are operating at different frequencies.

2.1. Communication Scheme

Since all transactions between system components are performed through queues which is 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 write data to queue to which the producer(system component which sends data) sends data

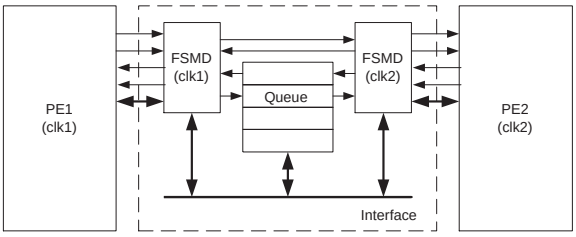


Figure 1. The interface architecture of two imcompatible components

which the consumer(system component which receives the data which the producer sends) needs.

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.

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 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 register. The bit width of internal register will be maximum bit width of two communication parties(same as bit width of the queue, bw_Q), which reduces the number of data transfers between 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 interface communication scheme between the state machines and a queue with a single I/O port. When two state machines share the same queue and data can be transfered bi-directionally, the handshaking between them is essential, because they share the same queue and must have a way to resolve any contention. So, the producer must generate a signal to let the consumer read the data in the queue. Also, consumer must send signal to let the producer know that it has read data. If two queues are used for storing data, or data transfers occur in one direction, the hand-shaking protocol is no longer needed which is shown in Figure 3. In addition, if queue with separate I/O ports is selected for storing data, the handshaking between FSMDs is not needed.

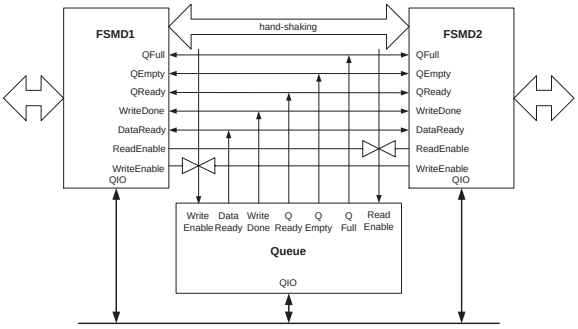


Figure 2. The interface communication scheme using one queue

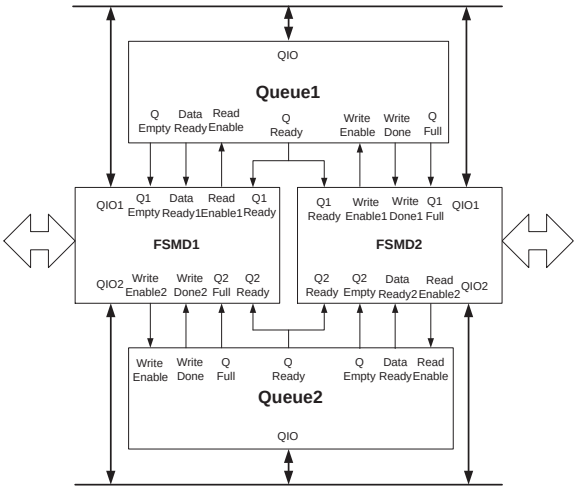
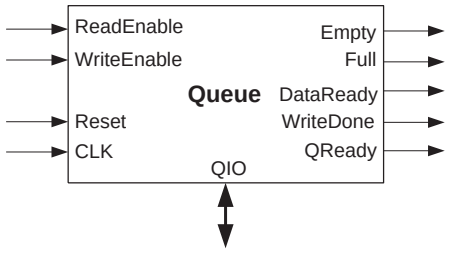


Figure 3. The interface communication scheme using two queues

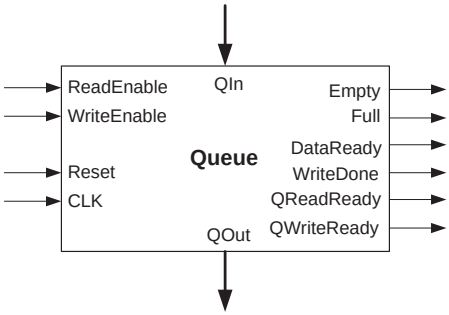
2.2. Queue Model

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 in which it was written in. System components like processors, ASICs which send data to each other, in the sense that when the 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



(a) Queue with a single I/O port



(b) Queue with two I/O ports

Figure 4. Queue block diagram

make our own queues which are general enough to include the specific queues, distributed by various vendors. Our implemenation of a queues is shown in Figuire 4. In our implemenation, 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 the longest, 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) starts writing(reading) the data in the queue, `QReady` become 1, which warns the other can't use the queue. 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 amount of 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 has 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.

2.3. Memory Timing

Generally, a queue contains memory to store data internally [Gaj97]. The operation of the queue is determined by memory organization and timing(Figure 5). There are many memory timing parameters in a memory datasheet. For example, the address lines will be set at t_0 then followed by `CS` at t_1. Consequently, the memory data will become available at t_2. The delay time $t_2 - t_0$ is called the memory access time(T_{acc}), since it would take $t_2 - t_0$ to obtain data from memory. The delay time $t_2 - t_1$ is called the ouput-enable time(T_{oe}), since it represents the delay in enabling the output drivers. After the value of the address lines has changed at t_3, the valid data will be available until time t_5. This time interval $t_5 - t_3$ is called the output hold time(T_{oh}). Finally the time difference $t_5 - t_4$ is called the output-disable time(T_{od}), since it represents the delay in the disabling of output data. For write-cycle timing, the address line must be set somewhat earlier. The delay $t_1 - t_0$ is called the address setup time(T_{as}). Data should be stable for some time before and after the falling edge of the `CS` to ensure proper operation. These times are called data setup time(T_{ds}) and data hold time(T_{dh}) and are defined by the time intervals $t_3 - t_2$ and $t_4 - t_3$. `CS` or `RW` signals have to be asserted for a duration equal to or longer than the write-pulse width(T_{wpw}), defined by the time $t_3 - t_1$. Furthermore, an address must stay valid for some time after the falling edge of `CS` or `RW`. This time, called address-hold time(T_{ah}), is defined by time interval $t_5 - t3$.

Since some parameters are less than the others, some

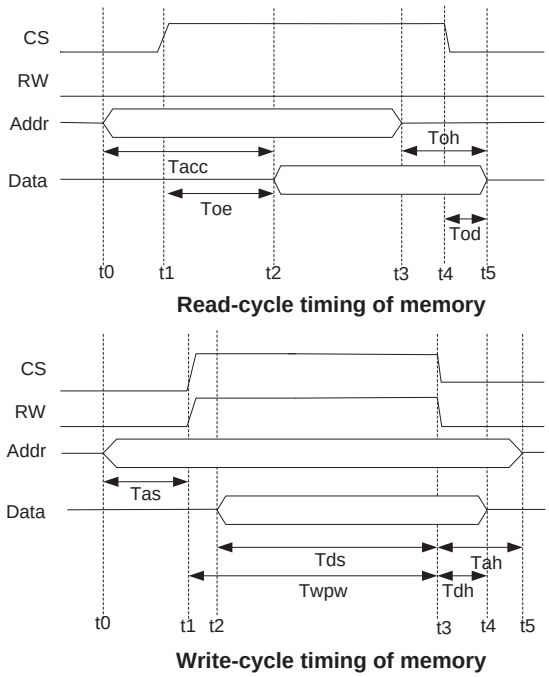


Figure 5. Timing diagram of a memory

parameters can be discarded. For example, output enable time(T_{oe}) is less than memory access time(T_{acc}) for read-cycle operation, output enable time can be ignored. In the same way, we can obtain 5 parameters, T_{acc}, T_{oh}, T_{as}, T_{wpw}, and T_{ah}, which are needed to be considered. Usually T_{as} and T_{ah} have 0 value, but we will consider them in order to generalize the memory timing model.

Until now, we have considered only asynchronous implementation of a memory. For the synchronous memory, all timing is synchronized by the clock and read/write operation can be performed in one cycle. Our queue will be synchrous implementation, the memory clock will be same as that of queue, the generated queue will perform read/write operation in a single cycle.

2.4. 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 the clock period of the queue. Given timing constraints of the memory and the clock period of the queue, queue generation can be considered as the task of generating a state machine which implements the 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 based on the granularity given by the queue clock. Finally the queue description will be generated for integration in interface synthesis. Figure 6 shows the read/write timing diagram of queue with a single I/O port, which contains a single port memory, which is shown in Figure 4(a). In this figure the `MEM/*` denotes ports of memory. For example `MEM/Addr` means the Addr port(Address port) of the memory. Figure 7 shows the read/write timing diagram of queue with two I/O ports which contains two port memory which is shown in Figure 4(b).

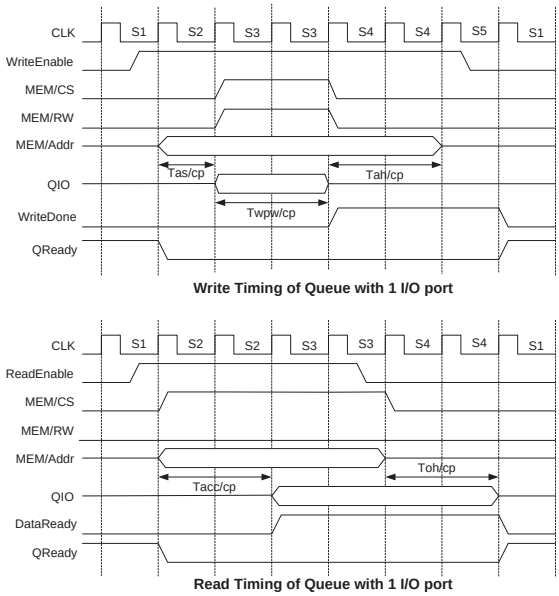


Figure 6. Timing diagram of a Queue with a single I/O port

2.5. Queue Implementaion

Figure 8 shows the internal architecture of a queue with a single read/write port. The queue incorporate two counters, `Front` and `Back`, pointing at the front and the back of the queue. The `Front` counter contains the address of the earliest written data. Whenever a read operation is requested, the data in location addressed by the `Front` counter is read to the I/O bus and the counter is incremented. The `Back` counter contains the address of the first empty location in the queue, and whenever a write operation is requested, the data is written into empty location addressed by the `Back` counter, at which point the counter is incremented. If data is being read from the queue and the `Front` counter points to the same location as the `Back` counter, it means the queue is empty. On the other hand, if data is being written into the queue, the `Back` counter points to the same location as the `Front` counter, this means the queue is full.

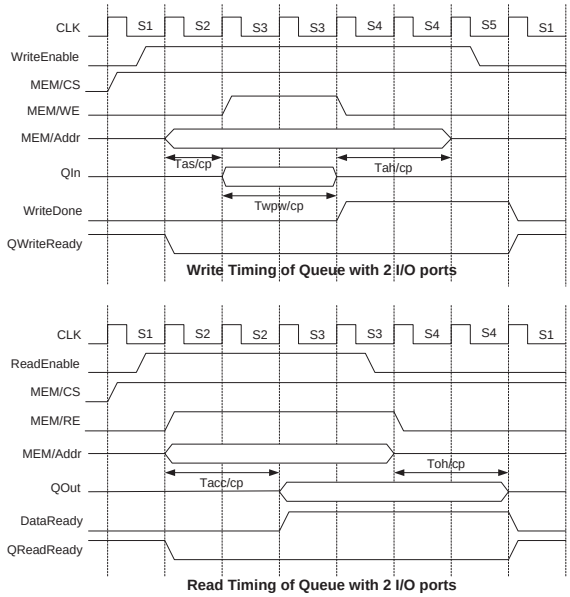


Figure 7. Timing diagram of a Queue with two I/O ports

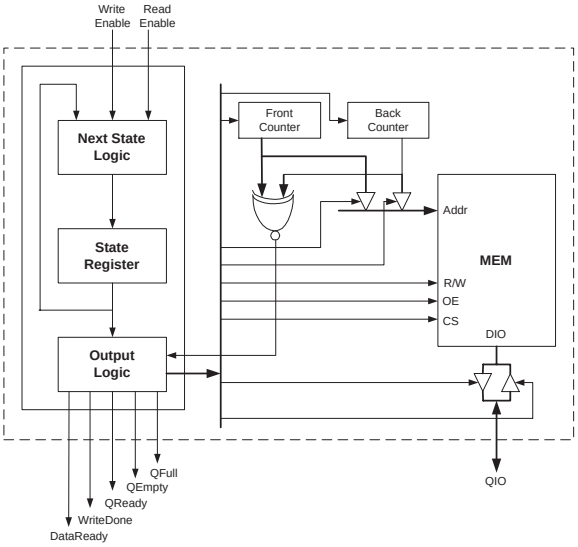


Figure 8. Internal architecture of a Queue

Table 1. clock selection for a queue

timing parameters		clock period			
		2 ns	5 ns	8 ns	15 ns
T_{acc}	15 ns	8	3	2	1
T_{oh}	8 ns	4	2	1	1
T_{as}	7 ns	4	2	1	1
T_{wpw}	23 ns	12	5	3	2
T_{ah}	5 ns	3	1	1	1
sum	58 ns	62ns	65ns	64ns	70ns
overhead		4ns	7ns	6ns	12ns

2.6. Clock Selection for Queue

As already mentioned, the depth of the queue can be determined by the total amount of data sent and ratio of sending rate and receiving rate of data in the producer and consumer respectively. The clock selection for the queue is also important and difficult to determine. During the clock selection, the memory timing parameters and the clock of communication parties(producer and consumer) should be considered.

If the queue has two different clocks for read and write cycle operation and the read/write clock period is long enough to accomodate the memory timing, then the clock of read operation will be that of the producer, and the clock of write operation will be that of the receiver, which makes interfaces between the queue and state machines synchronous. But dual clock scheme could not be applied to our queue implementation because it has only one clock.

The interface between the queue and the state machines may be asynchrous, if the clocks of the state machines are not harmonic to each other. Therefore, our queue implementation has the `DataReady` and `WriteDone` to accomodate asynchronous write/read operation between them, which means any clock period can be selected for the queue.

Based on the selection of the clock, read/write latency of the queue will be determined. If we select long clock period, it will lead to a larger idle time per clock cycle. For example, Table 1 shows the variation of the clock overheads with various clock selection. In this table, the larger is the clock period, the larger is the overhead. However, the shorter clock period result in large number of states. Trade-off between clock overhead and states becomes necessary while selecting a clock.

3. The Queue Generation Algorithm

As we already mentioned, we have to schedule the timing constraints based on given the clock period of the queue in order to generate a queue model from the memory timing constraints. Therefore, the FSMD implementation selects instances of the given timing ranges based on the granularity given by the queue clock. Then we can make general form of state machine of the queue as shown in Figure 9 and Figure 10. In Figure 9, state machine for queue with 1 read/write port which contains the 1 port memory is shown. The number of states are dependent on the 5 memory timing parameters as earlier described. In the same way, general form of state machine for the queue with 2 separate read/write port can be defined as shown in Figure 10.

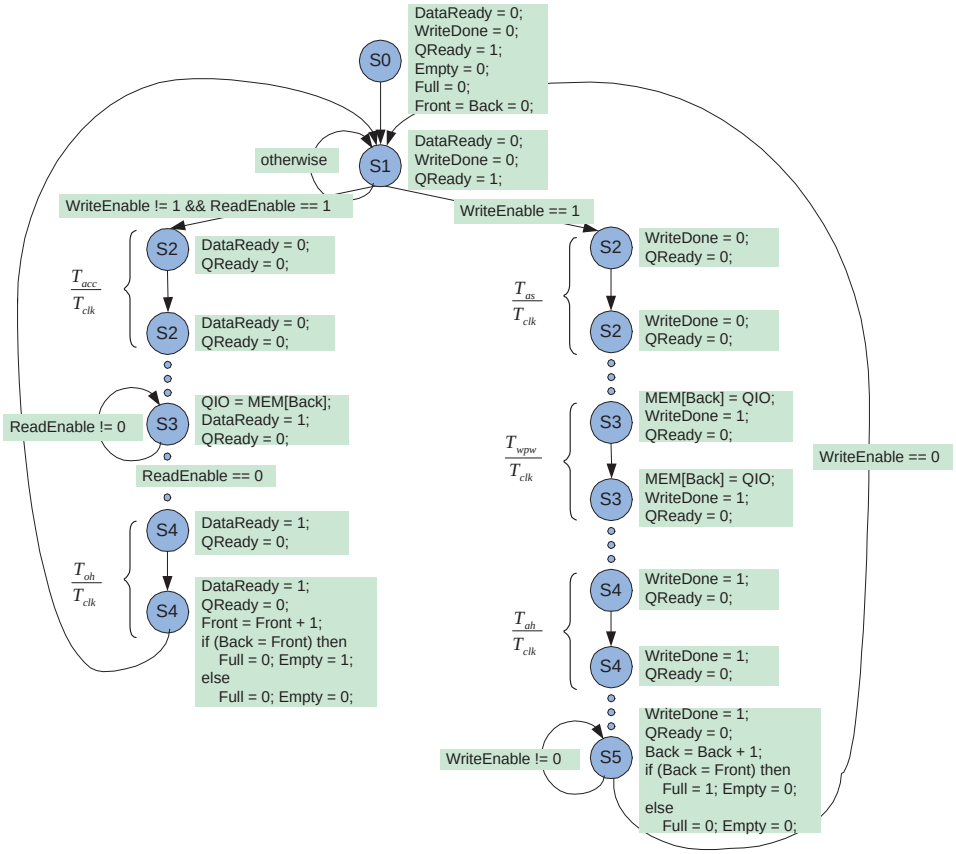


Figure 9. FSMD of a Queue with a single I/O port

3.1. Problem Definition

Given:

1. Timing parameters T_{acc}, T_{oh}, T_{as}, T_{wpw}, and T_{ah} for selected memory
2. Size(bit width and depth) of the selected memory
3. Clock period T_{clk} of the queue

Determine:

1. Finite state machine with data model for the queue.

Condition:

1. memory timing constraints are satisfied.

Algoritm 1 describes the queue generation algorithm from given memory timing constraints and clock period of the queue. In this algorithm, there are generation function calls like `GenerateResetState()`, `GenerateInitialState()`, and so on, which generates FSMD description of each state in Figure 9 and Figure 10. The function `AddState(FSMD, S)` adds state S into state machine(FSMD). First, function `GenerateResetState()` generates reset state(S0), in which every output singal and internal variables for the memory and counters are initialized. Whenever reset is asserted, the state of queue is in this state. Function `GenerateInitialState()` generates initial state(S1), in which all output signals are deasserted until `ReadEnable` or `WriteEnable` gets asserted by external producer and consumer.

For read cycle operation, memory access state(S2) is generated according to memory access time T_{acc} in function `GenerateMemAccessState()`, and data ready state(S3) by function `GenerateDataReadyState()`. Finally, function `GenerateMemOutputHoldState()` generates memory output hold state(S4) based on output hold time T_{oh}.

For write cycle operation, memory address setup state(S2) is generated according to memory address setup time T_{as} in function `GenerateMemAddressSetupState()`. Function `GenerateMemWriteState()` generates the memory

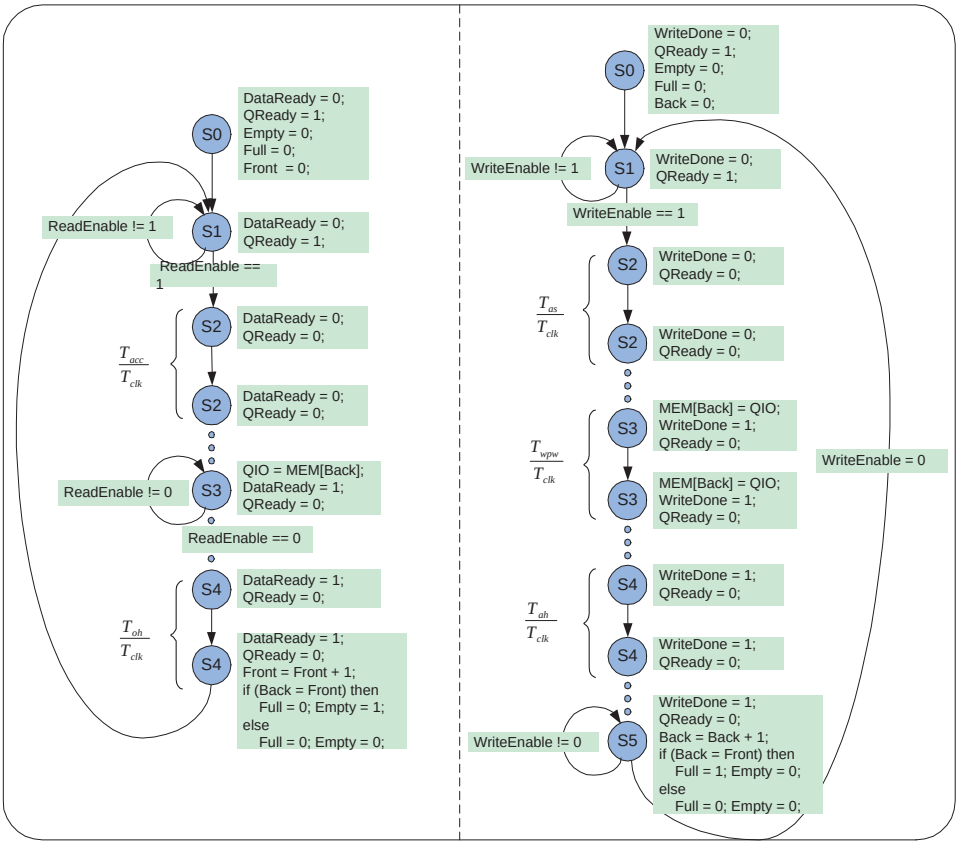


Figure 10. FSMD of a Queue with two I/O port

write state(S3) according to write pulse width time T_{wpw}. Function `GenerateMemAddressHoldState()` generates memory address hold state(S4), based on output hold time T_{ah}. Finally, the memory write done state(S5) is generated by function `GenerateMemWriteDoneState()`.

4. Examples

We implemented the queue generation algorithm in SpecC system. The tool takes the port description of the queue and timing parameter description of a memory as input and generates the SpecC [GZD+00] description for the queue model. We will use two memory types for generating the queues. The one is 1 port asynchronous memory, which has the following timing constraints: $T_{clk} = 2.0$, $T_{acc} = 5.0$, $T_{oh} = 1.0$, $T_{as} = 1.0$, $T_{wpw} = 3.0$, $T_{ah} = 1.0$, and the other is 2 port asynchronous memory, which has $T_{clk} = 2.0$, $T_{acc} = 2.0$, $T_{oh} = 0.0$, $T_{as} = 0.0$, $T_{wpw} = 2.0$, $T_{ah} = 0.0$. The size of the first is 8x32 and the other is 32x32. The generated queue models in SpecC are shown in Appendix A.1 and Appendix A.2.

5. Conclusion and Future Works

This report has shown our queue-based interface architecture, which is general enough to accomodate any target interface. In our architecture, queues are used to smoothen the burst data transfer requests and to interface incompatible protocols. The proposed queue generation algorithm can deal with any types of memories and timing constraints. Using two memory models with different I/O ports and timing constraints, queue models has been generated. In the future, we expect to develop an algorithm to synthesize our interface architecture from protocol description. The described queue algorithm will then be integrated into the interface architecture.

References

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

[GZD+00] Daniel Gajski, Jiwen Zhu, Rainer Domer, Andreas Gerstlauer, and Suqing Zhao. *SpecC:*

Algorithm 1 GenerateQueue(T_{clk}, T_{acc}, T_{oh}, T_{as}, T_{wpw}, T_{ah})

```
1:  S0 = GenerateResetState();
2:  S1 = GenerateInitialState();
3:  AddState(ReadStates, S0);
4:  AddState(ReadStates, S1);
5:
6:  // generate Read States
7:  for i = 1 to ⌈Tacc/Tclk⌉ do
8:      S2 = GenerateMemAccessState();
9:      AddState(ReadStates, S2);
10: end for
11: S3 = GenerateDataReadyState();
12: AddState(ReadStates, S3);
13: for i = 1 to ⌈Toh/Tclk⌉ do
14:     S4 = GenerateMemOutputHoldState();
15:     AddState(ReadStates, S4);
16: end for
17:
18: // generate Write states
19: for i = 1 to ⌈Tas/Tclk⌉ do
20:     S2 = GenerateMemAddressSetupState();
21:     AddState(WriteStates, S2);
22: end for
23: for i = 1 to ⌈Twpw/Tclk⌉ do
24:     S3 = GenerateMemWriteState();
25:     AddState(WriteStates, S3;
26: end for
27: for i = 1 to ⌈Tah/Tclk⌉ do
28:     S4 = GenerateMemAdressHoldState();
29:     AddState(WriteStates, S4);
30: end for
31: S5 = GenerateWriteDoneState();
32: AddState(WriteStates, S5);
```

Specification Language and Methodology. Kluwer Academic Publishers, January 2000.

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

A. Queue model in SpecC language

A.1. Queue with shared read/write port using 1 port asynchrous memory

```
/*******************************************************
* SpecC code generated by 'genQ'
* Date: Wed Apr 10 10:49:32 2002
* User: dongwans
*******************************************************/



behavior queue(in event clk, in bit[0:0] rst, in bit[0:0] ReadEnable,
    in bit[0:0] WriteEnable, out bit[0:0] DataReady, out bit[0:0] WriteDone,
    out bit[0:0] QReady, out bit[0:0] Empty, out bit[0:0] Full,
    inout bit[31:0] QInOut)
{

    bit[2:0] Front, Back;
    bit[31:0] Mem[8];

    void main(void)
    {
        enum state  { S0, S1, R1, R2, R3, R4, R5, W1, W2, W3, W4, W5 }  state;
        while (1)
        {
            wait(clk);
            if (rst)
            {
                state = S0;
            }
            switch (state)
            {
                case S0 :
                {
                    DataReady = 0b;
                    WriteDone = 0b;
                    QReady = 1b;
                    Empty = 1b;
                    Full = 0b;
                    Front = Back = 0;
                    state = S1;
                    break;
                }
                case S1 :
                {
                    DataReady = 0b;
                    WriteDone = 0b;
                    QReady = 1b;
                    if (WriteEnable == 1b)
                    {
                        state = W1;
                    }
```

```
            else if (WriteEnable != 1b && ReadEnable == 1b)
            {
                state = R1;
            }
            else
            {
                state = S1;
            }
            break;
        }
        case R1 :
        {
            DataReady = 0b;
            QReady = 0b;
            state = R2;
            break;
        }
        case R2 :
        {
            DataReady = 0b;
            QReady = 0b;
            state = R3;
            break;
        }
        case R3 :
        {
            DataReady = 0b;
            QReady = 0b;
            state = R4;
            break;
        }
        case R4 :
        {
            QInOut = Mem[Front];
            DataReady = 1b;
            QReady = 0b;
            if (ReadEnable == 0b)
            {
                state = R5;
            }
            else
            {
                state = R4;
            }
            break;
        }
        case R5 :
        {
            DataReady = 1b;
            QReady = 0b;
            Front = Front + 1;
            if (Front == Back)
            {
```

```
                Empty = 1b;
                Full = 0b;
            }
            else
            {
                Empty = 0b;
                Full = 0b;
            }
            state = S1;
            break;
        }
        case W1 :
        {
            WriteDone = 0b;
            QReady = 0b;
            state = W2;
            break;
        }
        case W2 :
        {
            Mem[Back] = QInOut;
            WriteDone = 1b;
            QReady = 0b;
            state = W3;
            break;
        }
        case W3 :
        {
            Mem[Back] = QInOut;
            WriteDone = 1b;
            QReady = 0b;
            state = W4;
            break;
        }
        case W4 :
        {
            WriteDone = 1b;
            QReady = 0b;
            state = W5;
            break;
        }
        case W5 :
        {
            WriteDone = 1b;
            QReady = 0b;
            Back = Back + 1;
            if (Front == Back)
            {
                Empty = 0b;
                Full = 1b;
            }
            else
            {
```

```
                        Empty = 0b;
                        Full = 0b;
                    }
                    if (WriteEnable == 0b)
                    {
                        state = S1;
                    }
                    else
                    {
                        state = W5;
                    }
                    break;
                }
            }
        }
    }
};
```

A.2. Queue with separate read/write port using 2 port asynchrous memory

```
/*******************************************************
* SpecC code generated by 'genQ'
* Date: Thu Jan 10 13:01:24 2002
* User: dongwans
*******************************************************/
behavior QRead(in event Clk, in bit[0:0] RST, in bit[0:0] ReadEnable,
    in bit[0:0] WriteEnable, out bit[0:0] DataReady, out bit[0:0] WriteDone,
    out bit[0:0] QReadReady, out bit[0:0] QWriteReady, out bit[0:0] Empty,
    out bit[0:0] Full, in bit[31:0] QIn, out bit[31:0] QOut, inout bit[4:0] Front,
    inout bit[4:0] Back, inout bit[31:0] Mem[32])
{
    void main(void)
    {
        enum state  { R0, R1, R2, R3 }  state;
        while (1)
        {
            wait(clk);
            if (rst)
            {
                state = R0;
            }
            switch (state)
            {
                case R0 :
                {
                    DataReady = 0b;
                    QReadReady = 1b;
                    Empty = 1b;
                    Full = 0b;
                    Front = Back = 0;
                    state = R1;
                    break;
                }
                case R1 :
```

```
                {
                    DataReady = 0b;
                    QReadReady = 1b;
                    if (ReadEnable == 1b)
                    {
                        state = R2;
                    }
                    else
                    {
                        state = R1;
                    }
                    break;
                }
                case R2 :
                {
                    DataReady = 0b;
                    QReadReady = 0b;
                    state = R3;
                    break;
                }
                case R3 :
                {
                    QOut = MEM[Front];
                    DataReady = 1b;
                    QReadReady = 0b;
                    Front = Front + 1;
                    if (Front == Back)
                    {
                        Empty = 1b;
                        Full = 0b;
                    }
                    else
                    {
                        Empty = 0b;
                        Full = 0b;
                    }
                    state = R1;
                    break;
                }
            }
        }
    }
};

behavior QWrite(in event Clk, in bit[0:0] RST, in bit[0:0] ReadEnable,
    in bit[0:0] WriteEnable, out bit[0:0] DataReady, out bit[0:0] WriteDone,
    out bit[0:0] QReadReady, out bit[0:0] QWriteReady, out bit[0:0] Empty,
    out bit[0:0] Full, in bit[31:0] QIn, out bit[31:0] QOut, inout bit[4:0] Front,
    inout bit[4:0] Back, inout bit[31:0] Mem[32])
{
    void main(void)
    {
        enum state  { W0, W1, W2 }  state;
```

```
        while (1)
        {
            wait(clk);
            if (rst)
            {
                state = W0;
            }
            switch (state)
            {
                case W0 :
                {
                    WriteDone = 0b;
                    QWriteReady = 1b;
                    Empty = 1b;
                    Full = 0b;
                    Front = Back = 0;
                    state = W1;
                    break;
                }
                case W1 :
                {
                    WriteDone = 0b;
                    QWriteReady = 1b;
                    if (WriteEnable == 1b)
                    {
                        state = W2;
                    }
                    else
                    {
                        state = W1;
                    }
                    break;
                }
                case W2 :
                {
                    MEM[Back] = QIn;
                    WriteDone = 1b;
                    QWriteReady = 0b;
                    Back = Back + 1;
                    if (Front == Back)
                    {
                        Empty = 0b;
                        Full = 1b;
                    }
                    else
                    {
                        Empty = 0b;
                        Full = 0b;
                    }
                    if (WriteEnable == 0b)
                    {
                        state = W1;
                    }
```

```
                    else
                    {
                        state = W2;
                    }
                    break;
                }
            }
        }
    }
};

behavior queue(in event Clk, in bit[0:0] RST, in bit[0:0] ReadEnable,
    in bit[0:0] WriteEnable, out bit[0:0] DataReady, out bit[0:0] WriteDone,
    out bit[0:0] QReadReady, out bit[0:0] QWriteReady, out bit[0:0] Empty,
    out bit[0:0] Full, in bit[31:0] QIn, out bit[31:0] QOut)
{
    bit[4:0] Front, Back;
    bit[31:0] Mem[32];

    U00 QRead(Clk, RST, ReadEnable, WriteEnable, DataReady, WriteDone, QReadReady,
        QWriteReady, Empty, Full, QIn, QOut, Back, Front, Mem);
    U01 QWrite(Clk, RST, ReadEnable, WriteEnable, DataReady, WriteDone, QReadReady,
        QWriteReady, Empty, Full, QIn, QOut, Back, Front, Mem);

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