General Transducer Architecture

D. Gajski, Hansu Cho, Samar Abdi

Technical Report CECS-05-08
August 8, 2005

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

{gajski,hscho,sabdi}@ics.uci.edu

Abstract

IP reuse is gaining importance in modern system design in order to reduce both cost and time to market. However, incompatibility of IP interface protocols is a serious hindrance to their reuse. Communication between two IP components, with different interface protocols, requires an extra component that must translate one protocol to another. This component is referred to as a transducer. In this paper we propose a template for a general transducer that can be parameterized to translate between any two protocols. Using this template, a transducer can automatically be generated to allow communication between IPs with incompatible interfaces. As a result, different IPs can easily be used in a system, without the manual and error prone task of designing custom transducers.

Contents

List of Figures

General Transducer Architecture

Hansu Cho, Samar Abdi, Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

IP reuse is gaining importance in modern system design in order to reduce both cost and time to market. However, incompatibility of IP interface protocols is a serious hindrance to their reuse. Communication between two IP components, with different interface protocols, requires an extra component that must translate one protocol to another. This component is referred to as a transducer. In this paper we propose a template for a general transducer that can be parameterized to translate between any two protocols. Using this template, a transducer can automatically be generated to allow communication between IPs with incompatible interfaces. As a result, different IPs can easily be used in a system, without the manual and error prone task of designing custom transducers.

1 Introduction

The design complexity of System-on-chip (SoC) is rising while the time to market for a new product is shrinking. Consequently, designers are forced to implement more functionality in a shorter period of time. The most widely used approach for tackling this problem is IP reuse. However, different IPs, typically, have different interface protocols. Therefore, communication from one IP to another might not be possible using a common bus. This communication must be routed through an additional component known as a transducer[1].

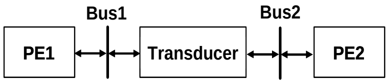


Figure 1. Top level Block Diagram of Transducer, PE1, and PE2

The transducer is a component with two interfaces as shown in Figure 1. We have two different Processing Elements(PEs), namely PE1 and PE2, using different protocols on Bus1 and Bus2, respectively. We also have a transducer component with two interfaces, one connected to Bus1 and the other to Bus2. Assume that PE1 wants to sent data to PE2. Since PE2 does not support the protocol of Bus1, the communication must go through the transducer component. The data is first sent from PE1 to transducer using Bus1. The transducer locally buffers the data and then sends it to PE2 using Bus2. Thus the transaction is completed.

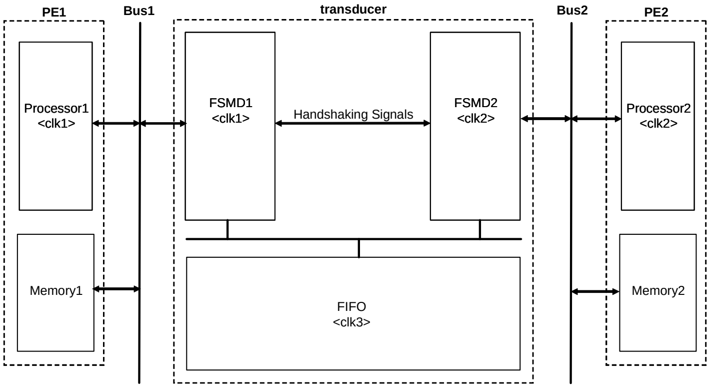


Figure 2. Block Diagram of Transducer

Figure 2 shows the transducer in greater detail. It is composed of three different subcomponents, namely FSMD1, FSMD2, and FIFO. FSMD1 implements the protocol of Bus1 and enables PE1 to read(write) to(from) the FIFO. If the protocol on Bus1 is synchronous, then FSMD1 must run at the same clock speed as PE1. FSMD2 implements the protocol of Bus2 and enables PE2 read(write) from(to) the FIFO. As in the previous case, if the protocol on Bus2 is synchronous, then the clock speed for FSMD2 must be the same as that of PE2. The FIFO consists of a local memory and a controller that communicates with FSMD1 and FSMD2 using double handshake protocol. The asynchronous handshake between the FIFO and the two FSMDs allows different clock speeds for FSMD1, FSMD2 and the FIFO. As a result, the transducer is capable of interfacing PEs running at different clock speeds.

The rest of the paper is organized as follows. Section 2 describes the functionality and structure of different parts of the transducer. In Section 3, we explain the operation of each subcomponent of the transducer. The details of communication between transducer and the PEs are illustrated in Section 4. Section 6 describes the sequence of operations for transferring data from PE1 to PE2. In section **??**, we present the methodology for automatic transducer generation.

2 Transducer

The transducer has three components - FSMD1, FSMD2, and FIFO. And some prameters are used to decide the datapath and controller of the transducer. In first subsection, the parameters are summarized. In following sections, the function and structure of each component is described. Also, architectures for bidirectional communication is explained after that.

2.1 Parameters

In this section, we summarize the parameters to describe the transducer
1. Timing diagram and signal direction for Bus1, Bus2 and FIFO memory
2. Clock of Bus1(clk1), Bus2(clk2), and Memory(*clk3*)
3. Bus1 data width(k) , Bus2 data width(n), and FIFO width(m)
4. Size of each field in message header(PEID, NameID, Size)
5. Master/Slave information of each PE
5. Timing constraints for the momory in FIFO (t_{WP}, t_{OE}, t_{DH} in Figure 6(b), Figure 8(b))

2.2 FIFO

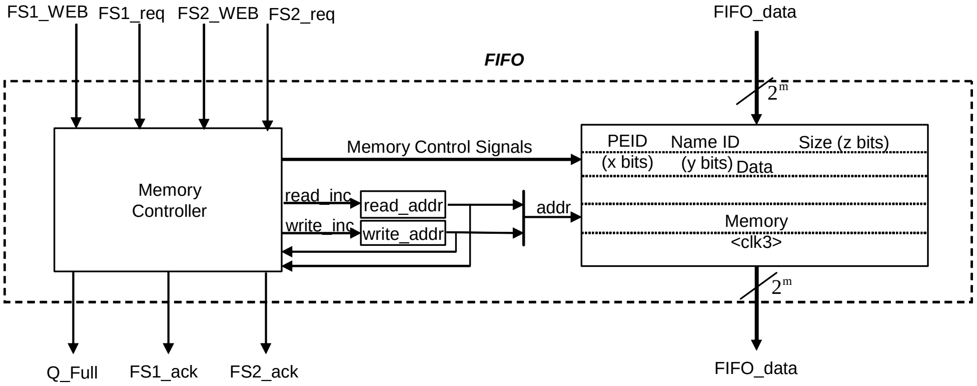


Figure 3. Block Diagram of FIFO

Figure 3 shows the structure of the FIFO. It has memory, memory controller, and address registers. Memory is used to store the temporary data since PE1 and PE2 read and write the data in different rate. Memory receives and send data to FSMD1 or FMSD2 using FIFO_Data bus. Each data stored in the memory should have three fields - PEID, NameID, Size. PEID represents the destination of the data. Name ID represents the identity and source of the data. Size represents the size of data.

Memory controller generates memory control signals and communicate with FSMDs. It communicates with FSMD1 and FSMD2 using double handshake protocol. No clock is used between them. This double handshake protocol enables PE1 and PE2 to read and write at different speed regardless of the clock. Since memory controller generates momory control signals for the memory from given parameters, any type of memory can be used.

Address registers indicate read and write address of the memory. After each read or write operation, memory controller increase their value by one. These registers are used to generate *Q_Full* signal. If FIFO is full, FSMD1 and FSMD2 should not write to FIFO till it has space. So FIFO has to send *Q_Full* signal to FSMD1 and FMSD2. *Q_Full* signal is aseerted if read and write address are the same after write operation.

2.3 FSMD1

FSMD1 receives the data from PE1 and write it to FIFO or vice versa. For these purpose, FSMD1 generates control signals for PE1 and match the different data width between Bus1 data and FIFO. Figure 4 is the internal structure of FSMD1. FSMD1 is composed of three main components - Controller, Data registers, and status flags.

The controller makes appropriate control signals for PE1 and data registers in FSMD1. Therefore PE1 can write data to registers in FSMD1 and read from it. Then the controller sends FIFO control signals to read(write) from(to) data registers.

FSMD1 has two status flags - *FSMD1_ready* and *Data2_ready*. *FSMD1_ready* represents data1 registers are ready(='1') or not(='0'). Whenever PE1 tries to write to data registers, it checks *FSMD1_ready* flag and writes only if it is ready. *Data2_ready* is used when FSMD1 reads data from FIFO. When FSMD1 receives *ready2* signal from FSMD2 then it checks *Data2_ready* flag and starts reading from FIFO if and only if *Data2_ready* is ready('1').

The data registers and its load(*ld* in Figure 4) and enable(*en* in Figure 4) signals deals with different data width between Bus1 and FIFO. In Figure 4(a), Bus1 data width(2^k) is smaller than FIFO width(2^m). So, each data register has the size of bus1 data width(2^k). The total size of data registers is equal to the FIFO width(2^m). In this case, PE1 write to each data for $2^{|k-m|}$ times and the controller generate load signal(*ld1*,*ld2*,...) for corresponding data register. After PE1 write to all data registers then enable signal(*en*) is assered to all data registers that it write to the FIFO at once. Figure 4(b) shows opposit case. Bus1 data width(2^k) is larger than FIFO width(2^m). So, each data register size is equal to Queu width and total size of the data register is Bus1 data width. In this case, data registers read at once and write to the FIFO $2^{|k-m|}$ times.

interrupt signal is used to send an interrupt to PEs when transducers runs in slave mode. *tx_interrupt* signals is used to receive the interrupt from the slave devices when transduer runs in master mode.

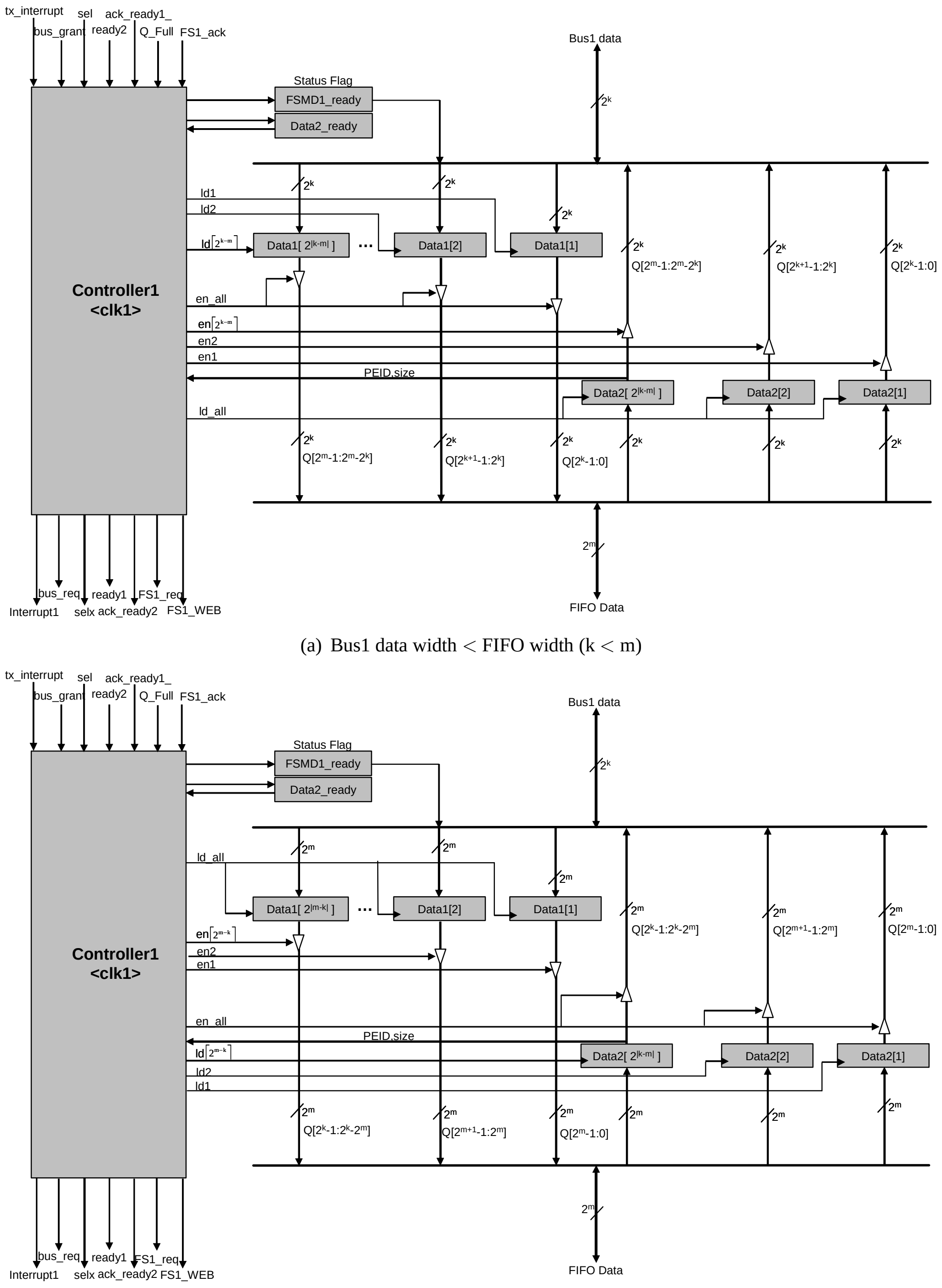
bus_req and *bus_grant* signals are used for arbitration. When multiple PEs are attached to the bus, then arbiter is attached to the bus and it communicate with PEs using these signals. *sel* and *selx* signals are used when one transducer writes data to the other transducer. Also, *selx* signals is used to write data to slave devices. More explanation is on section 4.

2.4 FSMD2

The functionality and internal structure of FSMD2 is basically the same with FSMD1. The only difference is that it deals with PE2.

2.5 Bidirectional communication

The transducer provides bidirectional communication between PE1 and PE2. In other to do that, the transducer have to solve two problems. First, it should handle the direction of the data flow. In Figure 4, data1x and data2x registers are used for this purpose. Data1x registers are used to receive data from PE and data2x registers are



(a) Bus1 data width < FIFO width (k < m)

(b) Bus1 data width > FIFO width (k>m)

Figure 4. Block diagram of FSMD1

used to send data to PE. Second, the transducer should prevent simultaneous writing to any of the storage device, because data are comming from two directions. It has three storage devices - Data1 register, Data2 register, and FIFO Memory. Data1x and Data2x registers are dedicated to one way of data flow. So, simultaneous writing never happens. *FS_ack* signals prevent the probelm with FIFO Memory. If FIFO receives two *FS_req* signals from both FSMDa and FSMD2, then it replies to only one request. That, the other FSMD waits till the operation ends.

3 Transducer Operation

In this section, The internal operation of the FIFO is explained. Section 3.1 shows how FSMD write data to FIFO. Section 3.2 explains how FSMD reads data from the FIFO. FSMD1 and FSMD2 are the masters and FIFO is the slave during these operations.

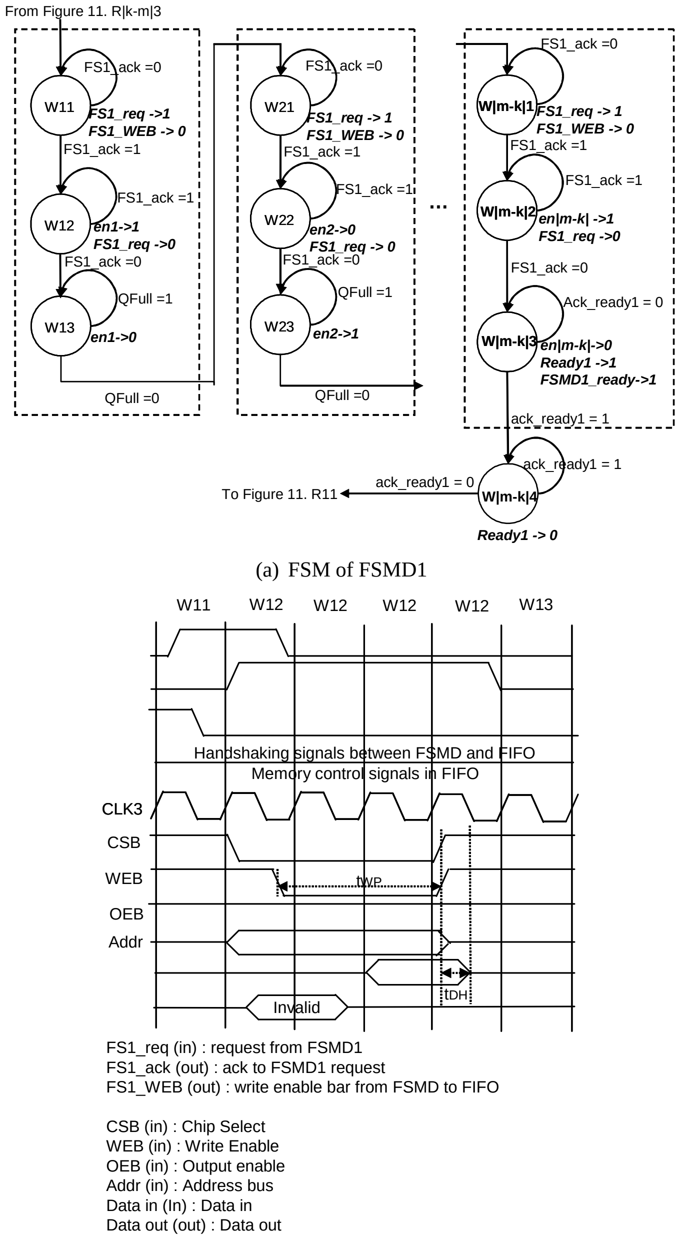
3.1 FSMD1 to FIFO

FSMD1 and FIFO are communicating using double handshaking protocol to deal with different read/write speed between PE1 and PE2. The double handshaking protocol uses three signals. Request(*FS1_req*), acknowledge(*FS1_ack*), read/write(*FS1_WEB*). The operation has four steps. First, FSMD1 reqeusts to FIFO by setting *FS1_req* to '1'. Read/write operation is also indicated using *FS1_WEB* in this step. Second, FIFO replies when it is ready by setting *FS1_ack* to '1' and starts reading or writing. Third, FIFO rests *FS1_ack* to '0' to tell the end of read/write to FSMD1. Finally, FSMD1 resets *FS1_req* to '0' and all operation is over.

Figure 5 is given to describe the operation from FSMD1 to FIFO. Figure 5(a) is the FSM of the controller in FSMD1 for writing operation. Figure 5(b) shows the timing diagram between FSMD and FIFO. Figure 6(a) is the corresponding FSM in FIFO.

In W11 state in Figure 5(a), FSMD1 sends *FS1_req* to the FIFO. If FIFO replies *FS1_ack*, then FSMD1 writes data to FIFO by giving enable signals to its data register(*en* in Figure 4) in W12 state. In state W13, after writing to FIFO and receiving *FS1_ack* signal, it check *Q_Full* signal. If FIFO is full, then waits till it has some space. If FIFO has space, then FSMD1 write the the data in next data register again and again. After FSMD1 writes all data registers to FIFO then, FSMD1 sets *FSMD1_ready* flag in Figure 4 and send *ready1* signal to FSMD2.

While FSMD1 is running according to FSM in Figure 5(a), FIFO runs according to FSM in Figure 6. In Figure 6(b) has two signal groups. One for FSMDs and the other for Memory. *FS1_req* and *FS1_ack* is the double handshaking signals for FSMDs. *FS1_WEB* signal decides read/write operation. The other signals below *clk3* is memory control signals. If FIFO receives *FS1_req* signal and *FS1_WEB = '0'* from FSMD1 at S0 state, then its memory controller replies *FS1_ack* to FSMD1 and puts '0' to *CSB* and *WEB* to write data to the memory in S1 state. In S2 state, it waits till data is written to the memory. Then in state S3, memory controller set *CSB* and *WEB*. After finishing writing data to memory, memory controller increases write address in S4 and reset *FS1_ack* to '0' to notify the end of operation to FSMD1. If FSMD1 reset *FS1_req* to '0' in S5, then all write cylce is done and FIFO waits for another request.



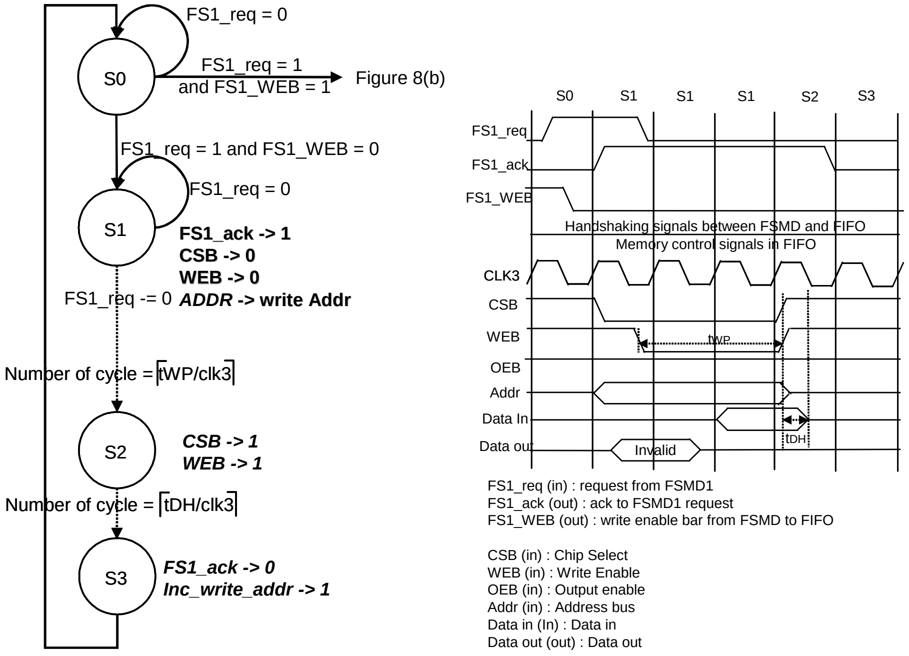
(a) FSM of FSMD1

FS1_req (in) : request from FSMD1
FS1_ack (out) : ack to FSMD1 request
FS1_WEB (out) : write enable bar from FSMD to FIFO

CSB (in) : Chip Select
WEB (in) : Write Enable
OEB (in) : Output enable
Addr (in) : Address bus
Data in (In) : Data in
Data out (out) : Data out

(b) Timing Diagram of FSMD1

Figure 5. FSMD1 to FIFO - FSMD1 FSM and Timing Diagram



(a) FSM of FIFO

(b) Timing Diagram of FIFO (= Figure 5(b))

Figure 6. FSMD1 to FIFO - FIFO FSM and Timing Diagram

Two parameters are used to describe write memory timing in Figure 6(b). t_{WP} represents the length of write period. t_{DH} is the data hold time after write enable sigal*WEB* is set to '1'. If FIFO uses some special memory then, protocol of the memory used in FIFO shoud be given instead of parameters.

3.2 FIFO to FSMD2

This section explains how FSMD1 or FSMD2 reads data from FIFO. Figure 7(a) shows the FSM in FSMD2 for the operation. Figure 7(b) shows the timing diagram between FSMD2 and FIFO. Figure 8(a) shows corresponding FSM in FIFO.

If FSMD2 receives *ready1* signal in R11 state in Figure 7(a), then FSMD2 replys *ack_ready1* signal to FSMD1 and send *FS2_req* to FIFO to read the data. The other sequence are pretty much same with section 3.1. The only differentce is the last state. In state R$|n-m|$4,after FSMD2 finishes reading the data from FIFO, it send interrupt to Processor2 to notify that data is ready.

Figure 8 shows the corresponding FSMD and Timing diagram of FIFO. If FIFO receives the *FS2_req* signal at S0, then it replys *FS2_ack* to FSMD2 and starts reading from memory. Memory reading sequence is almost same with the sequence in section 3.1.

4 Operation between Transducer and PE

In this section, communication between PE and Transducer will be shown.

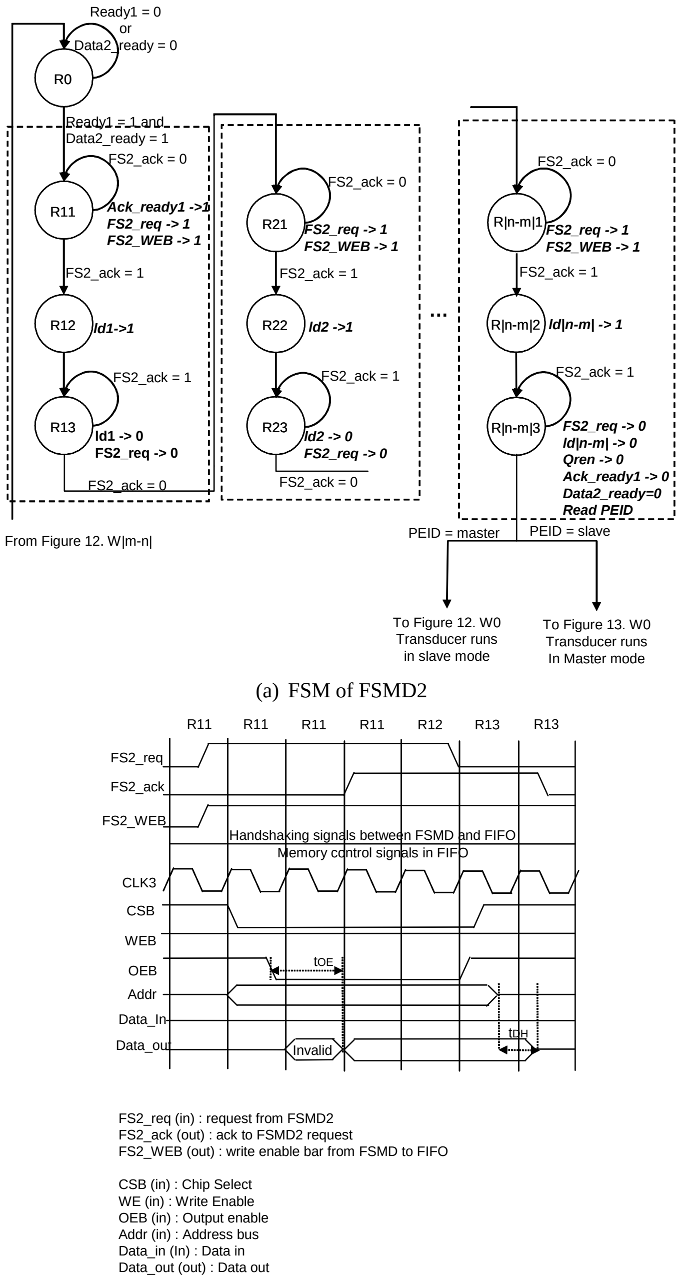
Section 4.1 is the example of synchronous protocol from PE1 to FSMD1. Section 4.2 shows the example of asynchronous protocol from FSMD2 to PE2. Finally, section 5 explains arbitration.

4.1 PE1 to Transducer

Data transfer from PE1 to Transducer can be divided into two cases. In first case, PE1 runs as a master and transducer runs as a slave. In second case, PE1 runs as a salve and transducer becomes master. In first case, the operation is composed of three step. First, PE1 reads the data from Memory1. Second, PE1 checks the *FSMD1_ready* flag in FSMD1. When it is ready, PE1 writes to data registers in FSMD1 as the final step. For each step, PE1 have to acquire the bus using *Bus_req* and *Bus_grant*. The detailed operation is explained in this section.

In second case, PE1 send an interrupt to the transducer then transducer read the data from PE1 using memory read operation. When transducer runs in master mode, it read the size in the message. For every read operation, transducer reduce the size by the amount of data1 register size. Transducer keeps reading till the size reaches zero. Its detailed operation is similar to the operation of the transducer in slave mode in section 4.2 except that transducer is the master and PE is the slave in this case.

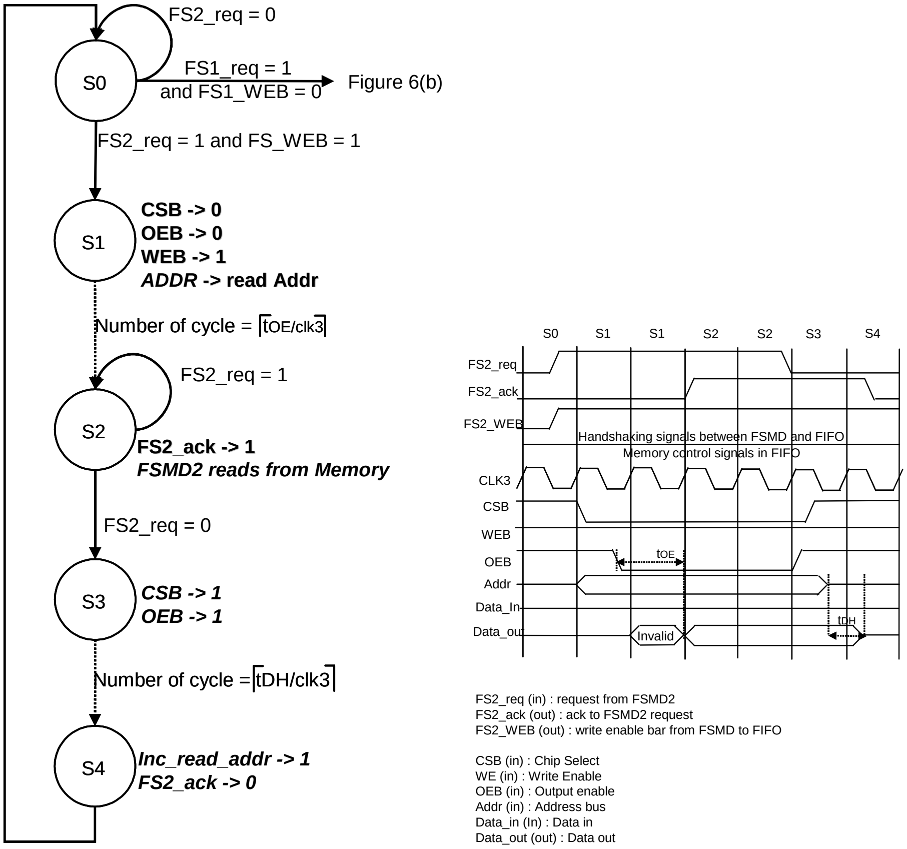
Figure 9 is the first step when PE1 runs as a master. PE1 reads data from memory1 using memory read operation. Memory read/write operation uses write enable bar(*WEB*), address, and data signals in general. In read operation, *WEB* goes down to



(a) FSM of FSMD2

(b) Timing Diagram of FSMD2

Figure 7. FIFO to FSMD2 - FSMD2 FSM and Timing Diagram



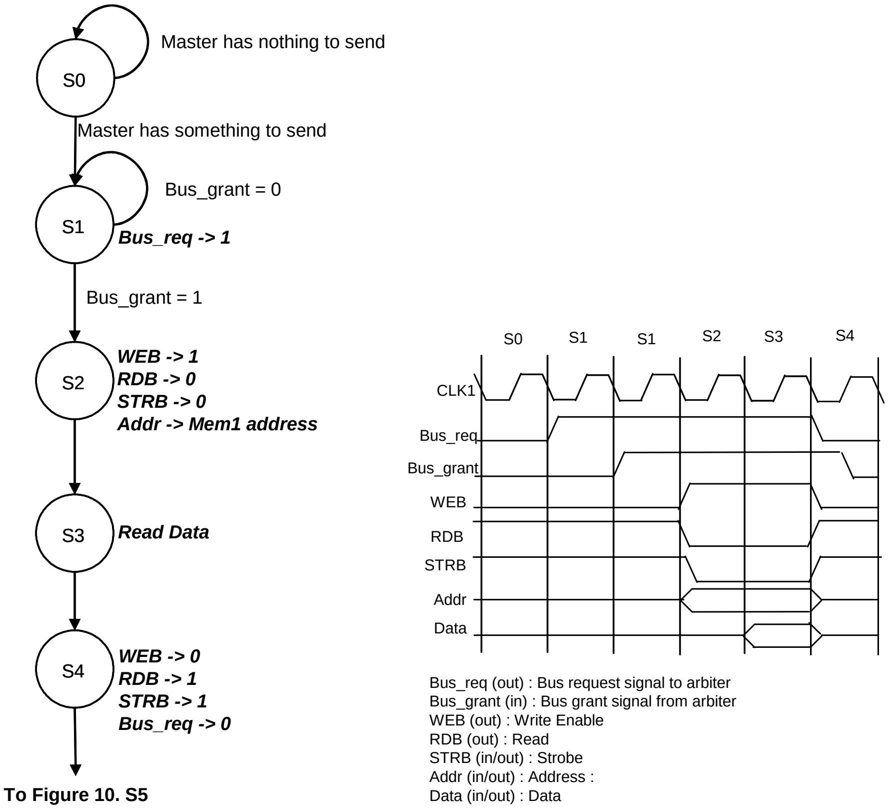
FS2_req (in) : request from FSMD2
FS2_ack (out) : ack to FSMD2 request
FS2_WEB (out) : write enable bar from FSMD to FIFO

CSB (in) : Chip Select
WE (in) : Write Enable
OEB (in) : Output enable
Addr (in) : Address bus
Data_in (In) : Data in
Data_out (out) : Data out

(a) FSM of FIFO

(b) Timing Diagram of FIFO (= Figure 7(b))

Figure 8. FIFO to FSMD2 - FIFO FSM and Timing Diagram



(a) FSM

(b) Timing Diagram

Figure 9. Memory1 to Processor1 - Processor1 FSM and Timing Diagram

’0’ and address are coming at the same cycle. Memory takes at least one clock cycle to interpret the address and give the data at next clock cycle. Every control signals are issued at S1, and data is read in following clock cycle (S2).

Figure 10 shows second step. After reading the data from Memory1, PE1 tries to write the data to FSMD1. PE1 checks the status of FSMD1 before writing, if *FSMD1_ready* flag is ’0’ then it keeps reading the status flag till it is ’1’. Else it writes to data1 register in FSMD1 using memory write operation. We assumed memory mapped I/O that Memory1 and FSMD1 can be read or write using memory read/write operation without overwriting each other.

From S6 to S8, PE1 reads the status register in FSMD1 using memory read operation. Address for status register is loaded in the address bus. Its timing diagram is shown in Figure 10(b). If FSMD1 is ready then PE1 write the data to data1 register in FSMD1 using memory write operation. States from S9 to S11 and its corresponding Timing diagram Figure 10(c) shows its memory write operation. In this case, Data1 address is loaded in address bus.

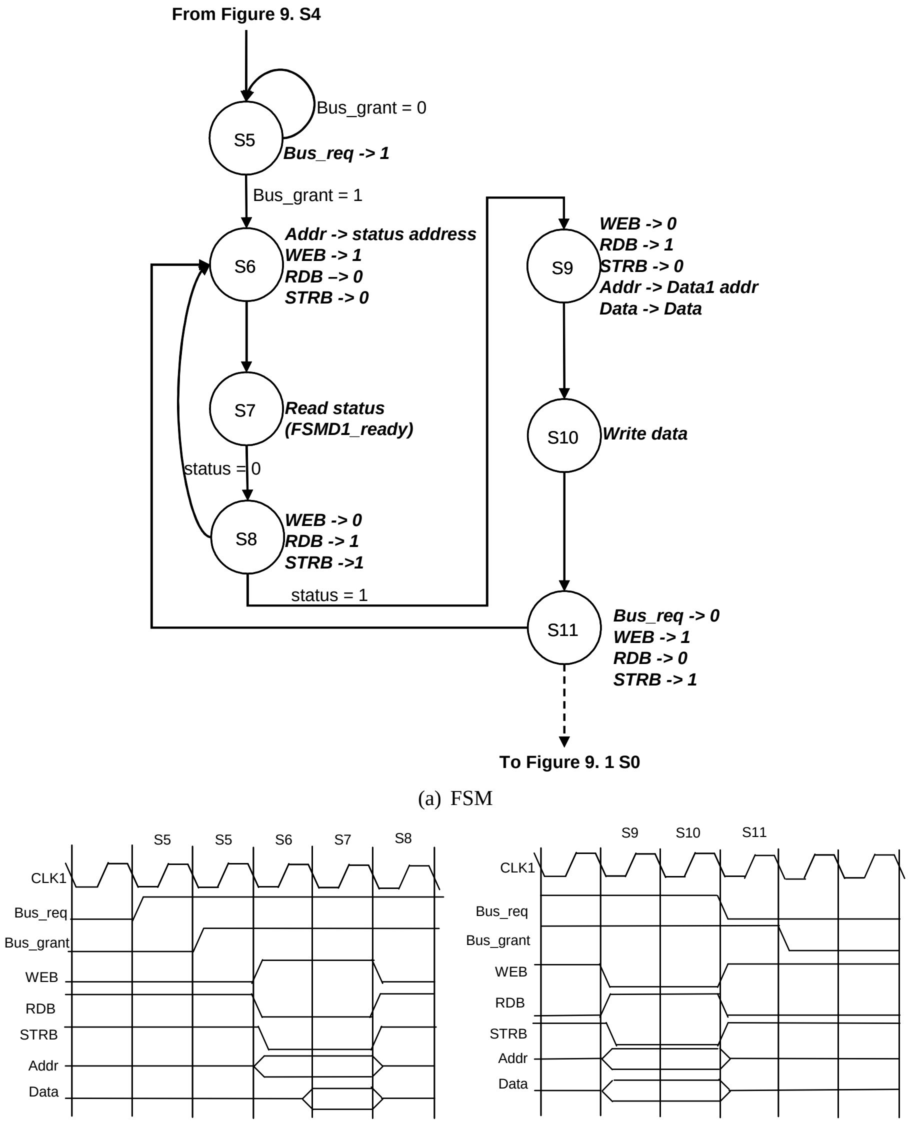
Figure 11 is corresponding FSMD form S9 to S11 in Figure 10(a). Figure 11(a) is FSM for this operation. Memory write operation is running $2^{|k-m|}$ times. For example, assume that PE1 uses 16 bits data and FIFO has 64 bit width, then FSMD1 should have four 16 bits data registers and Processor 1 writes to each register in FSMD1 over four times. In every second state in each reading cycle (R12,R22,...) load signal(*ld1,ld2,...*) for each register is issued. If all of the data registers are written then FSMD1 reset *FSMD1_ready* flag to ’0’ to prevent overwriting and send FIFO write request signals(*FS1_req, FS1_WEB*) to FIFO.

4.2 Transducer to PE2

After FSMD2 reads the data from the FIFO, it reads PEID. If PEID represent master, then transducer runs in slave mode and sends an interrupt to PE. PE reads the data from data2 register in FSMD2 using memory read operation. Else, transducer runs as a master device. If Transducer runs in master mode, then its FSM and Timing Diagram is similar with Figure 10(Processor1 to FSMD1). Transducer checks the status of the slave device first, then write the data to the slave device. The transducer send select signal or request signal to the slave device to initiate writing.

Figure 12 is the FSM and timing diagram of the FSMD2 master mode when it uses double handshake protocol. As mentioned above, the transducer tries to acquire the bus using *bus_request* and *bus_grnat* signals in W0 state. Then it checks the status of the slave device from W1 to W2 state. W1 and W2 state does not represent one clock cycle. They are inserted to show the operation sequence. In W11 state, if the bus uses double handshake, then the transducer sends request signal to the slave device. Else, *selx* signal should be asserted in W1 state and should remain till the end of writing operation. As transducer runs as a master in this mode, it reads the size field in the message at first stage, and keeps writing till the size becomes zero.

Figure 13 is the FSM and timing diagram of the FSMD2 when it runs in slave mode. In W0 state, FSMD2 sends interrupt to PE. Then, the master device, PE send request to FSMD2 in Wx1 state. In Wx2 state, FSMD2 replies acknowledge to PE and write to bus using *en* siganl of data2 registers in Wx3 state. In Wx4 state, if PE finishes reading

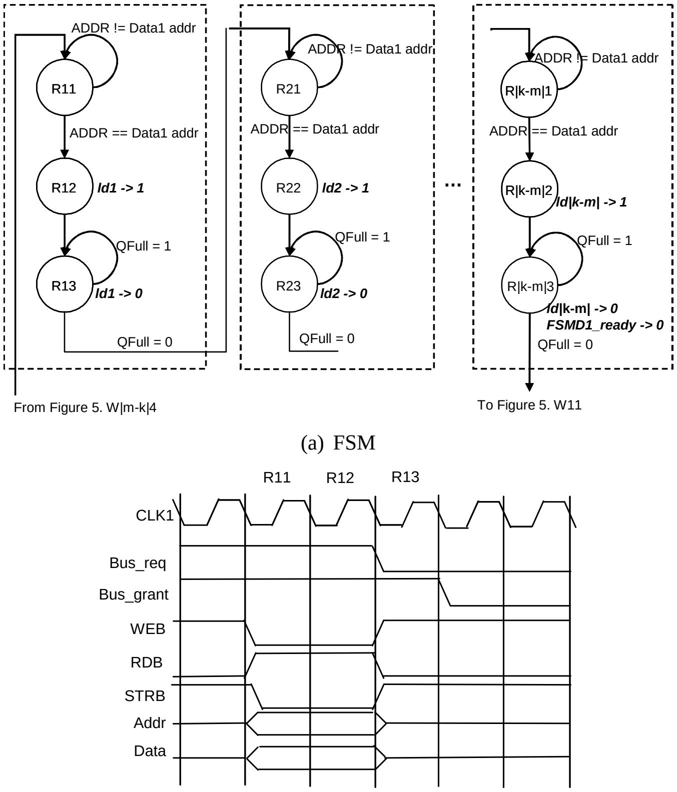


(a) FSM

(b) Timing Diagram - Processor1 checks status

(c) Timing Diagram - Processor1 writes to FSMD1

Figure 10. Processor1 to FSMD1 - Processor1 FSM and Timing Diagram

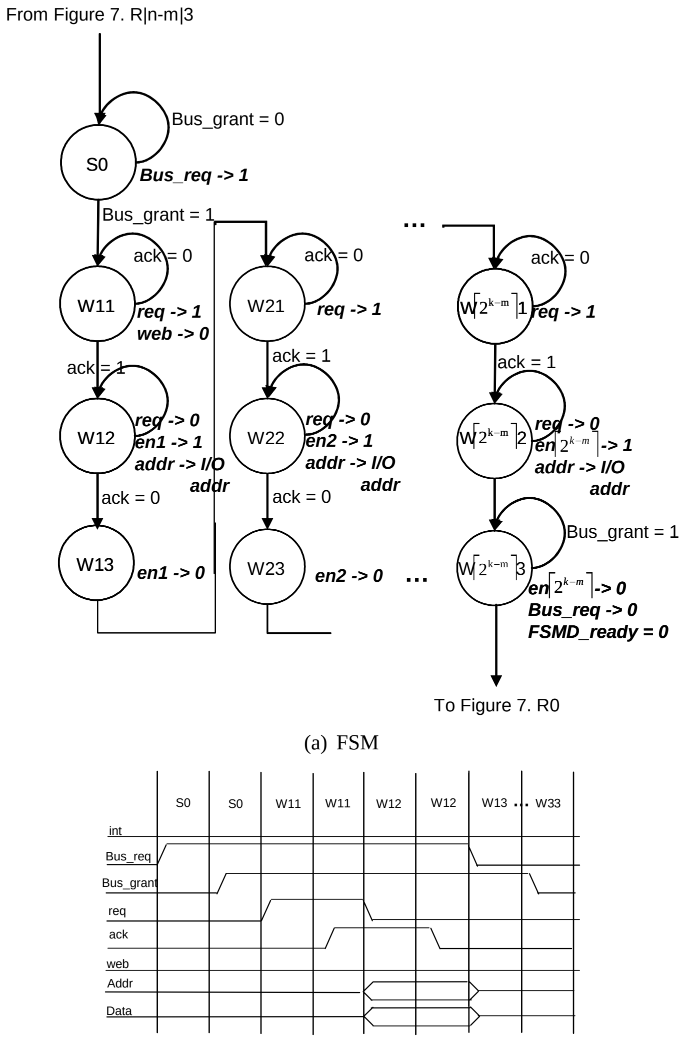


(a) FSM

Bus_req (out) : Bus request signal to arbiter
Bus_grant (in) : Bus grant signal from arbiter
WEB (out) : Write Enable
RDB (out) : Read
STRB (in/out) : Strobe
Addr (in/out) : Address :
Data (in/out) : Data

(b) Timing Diagram (= Figure. 10(c))

Figure 11. Processor1 to FMSD1 - FSMD1 FSM and Timing diagram



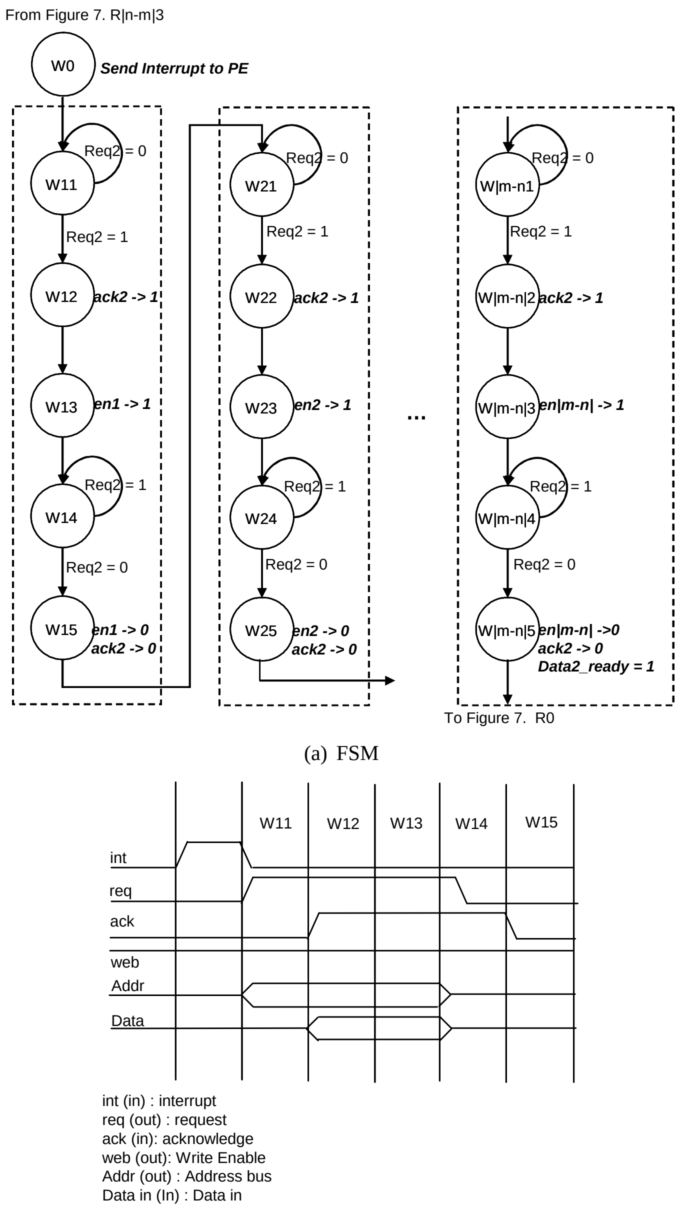
(a) FSM

Transducer write to Slave Device

Int (in) : interrupt
Bus_req (out) : Bus request
Bus_grant(in) : Bus grant
req (out) : request
ack (in): acknowledge
web (out): Write Enable
addr (out): Address
Data (in) : Data

(b) Timing Diagram

Figure 12. FSMD2(Master) to Slave Device - FSMD2 FSM and Timing Diagram

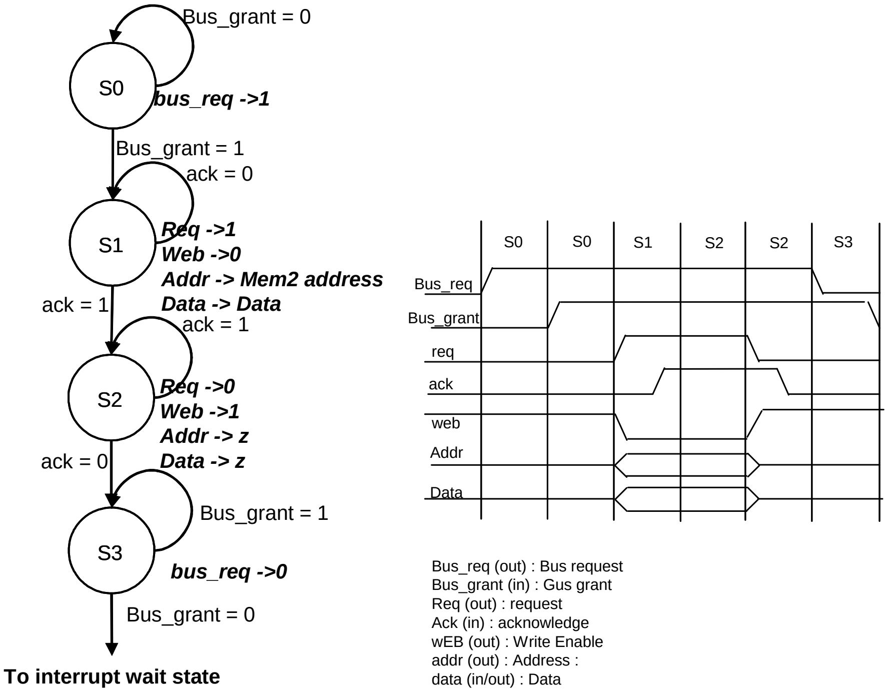


(a) FSM

(b) Timing Diagram

Figure 13. FMSD2(Slave) to Processor2 - FSMD2 FSM and Timing Diagram

then it set *req2* signal to '0' to tell FSMD2 that it read the data. In State Wx5, *ack2* siganl is set to '0' and it means the end of transfer. After one cycle(dotted box) then, PE keep reading next data register till read all data registers in FSMD2. Finally, PE2 write the data to Memory2 using memory write operation. Figure 14 shows its FSMD and timing diagram.



(a) FSMD

(b) Timing Diagram

Figure 14. Processor2 to Memory2 - Processor2 FSM and Timing Diagram

5 System with Transducers

This section deals with multiple transducer and with multiple PEs. But, only two basic configuration is used to compose all possible comfiguration with multiple PEs and multiple transducers.

5.1 General model

Figure 15 shows general case example. Between PE1 and PE2 it has one transducer. PE1 and PE3 have two transducers between them. Any other configuration can be extended using these two basic configurations. Therefore, the transducer model suggested can be a general transducer.

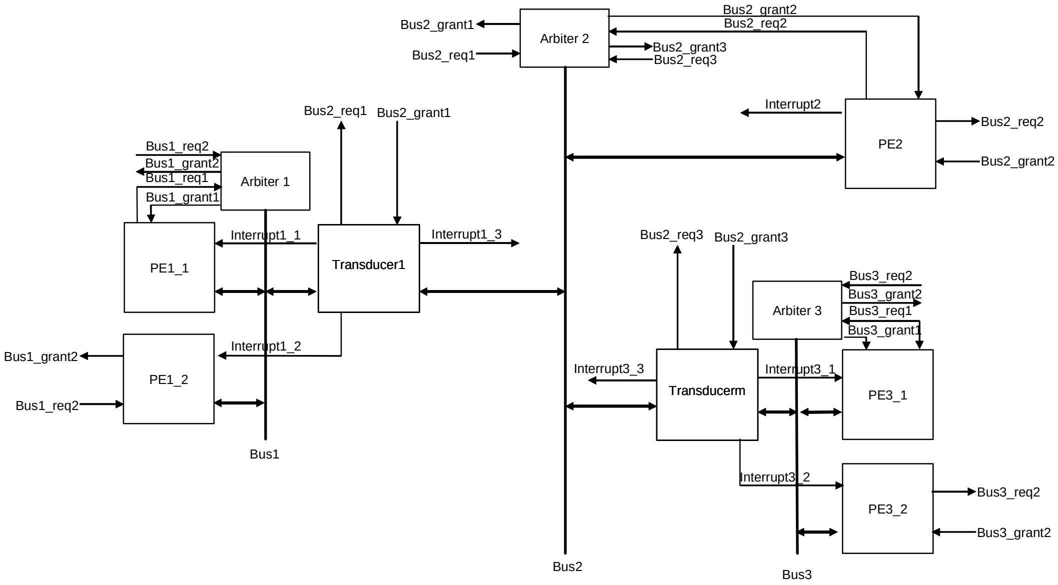


Figure 15. Arbitration - General model

5.2 Bus Transducer

Figure 16 is the first basic configuration. It has two buses and multiple PEs are connected to each bus. An arbiter is attached to each bus and there is only one transducer between buses. This case can be devided into two operation. From PE to transducer and transducer to PE. From PE to transducer, PE is always is master and transducer is always slave. But, from transducer to PE, the role of the transducer depends on PE. If PE is slave then the transducer becomes master. Otherwise, transducer becomes slave.

5.3 Dedicated Transducers to each PE

Second basic configration is shown is Figure 17 Each PE uses different protocol that each PE has own transducer and the transducer is connected to one main bus. In this case, there are two transducers between PEs. For example, when PE1 send data to PE2, the data starts from PE1.Then, it goes through transducer1 and transducer2. Finally, the data arrives at PE2. In this example basic operation between PEs and transducer is the same with previous configration. But,there are two transducer between PEs. The first transducer becomes master and the second transducer becomes slave. So, transducer1 acquires the bus and send *sel_2* signal to transducer2 with status check signals. If trnasducer2 is ready, then transducer1 writes the data according to the main bus write protocol.

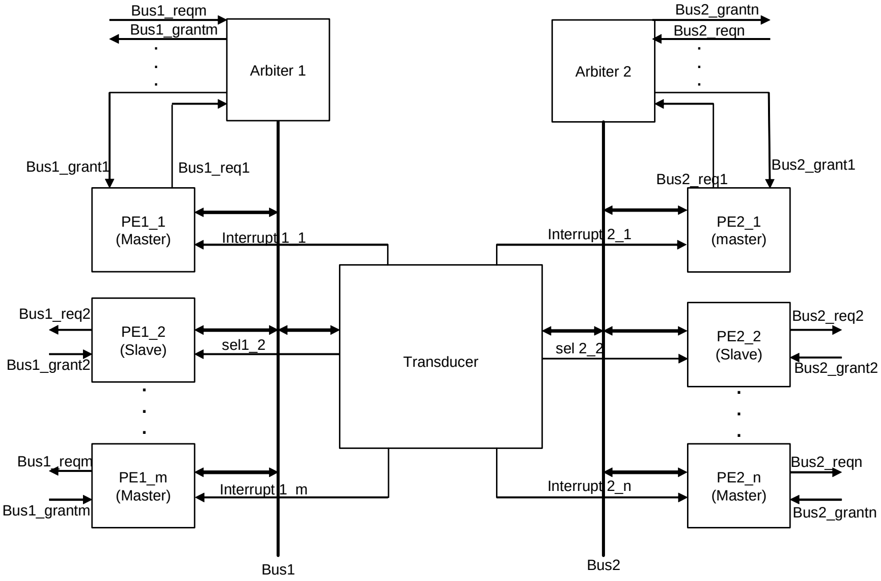


Figure 16. Arbitration - one transducer between two PEs

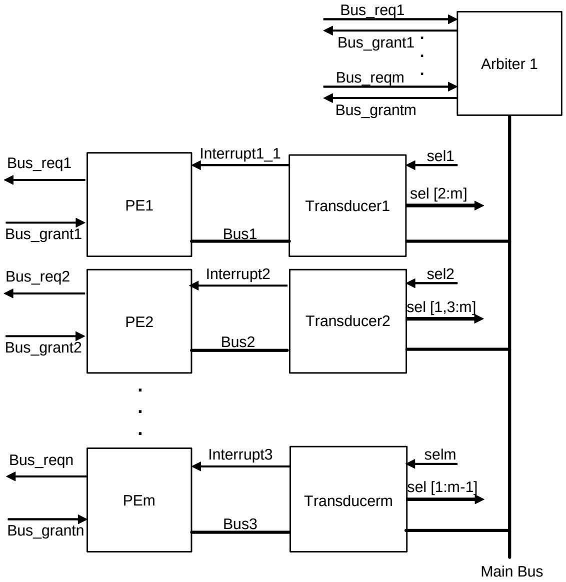
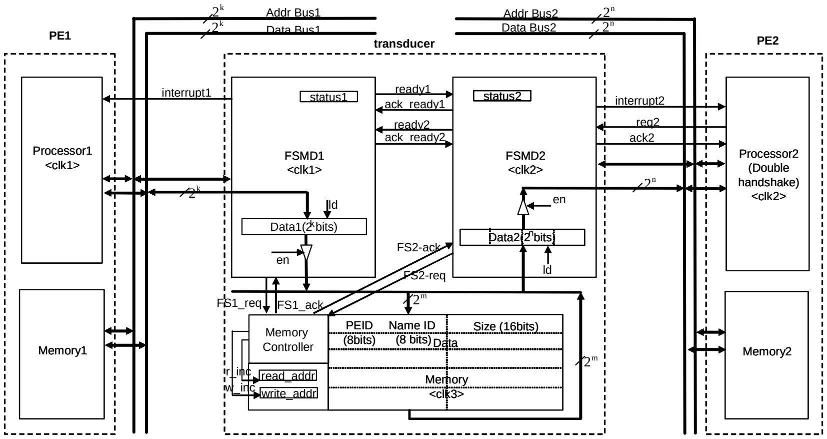


Figure 17. Arbitration - two transducers between PEs



Parameter : Bus1 width = k, Queue width = m, Bus2 width =n
Timing Diagram for Bus1, Bus2
Singal In/Out Direction
Size of each field in message (PEID, NameID, Size)
Master/slave information for each PE
clk(can be given or can be derived from TD + FSMD)
t_{WP},t_{DH},t_{OE} for Memory controller (Timing Diagram 6,7)
(twp: write pulse width, t_{DH} : data hold from write time, t_{OE} :Output enable to vaild output)
Status bits : FSMD Ready, PE_req,Q_Full

Figure 18. Block Diagram of Transducer in Detail

6 Overall Operation from PE1 to PE2

This section summarizes the operation of each block according to data flow sequence. Figure 18 shows the detailed block diagram without arbitration. Its has 7 sub blocks - Memory1, Processor1, FSMD1, FIFO, FSMD2, Processor2, and Memory2. Therefore there are 6 steps present to send a data from Memory1 to Memory2.

6.1 Memory1 toProcessor1

Processor 1 reads data from Memory1 using memory read operation

6.2 Processor1 to FSMD1

After reading the data from Mermoy1, Processor 1 tries to write the data to FSMD1. Before Processor1 starts writing the data to FSMD1, Precessor1 checks the *FSMD1_ready* flag in FSMD1. If it is ready, then Processor1 writes the data using memory write operation, else it keeps reading the *FSMD1_ready* flag. We assumed memory mapped I/O that Memory1 and FSMD1 can be read or write using memory read/write operation without overwriting each other.

6.3 FSMD1 to FIFO

When all data registers are written by Processor1, then FSMD1 reset *FSMD1_ready* flag to '0' to prevent overwriting by PE1 and send write request signal to FIFO. After

FSMD1 writes all of the data to FIFO, then FSMD1 set *FSMD1_ready* flag to '1' and send *ready1* siganl to FSMD2 to notify that FIFO has data for FSMD2.

6.4 FIFO to FSMD2

If FSMD2 receives the *ready1* signal from FSMD1, then it replies *ack_ready1* signal to FSMD1 and send *FS2_req* signal to FIFO.

6.5 FSMD2 to Processor2

After FSMD2 reads the data from the FIFO, it reads PEID. In this example, PE2 is the master device that transducer sends an interrupt to Processor2. Then, Processor2 reads the data from data2 register in FSMD2 using memory read operation.

6.6 Processor2 to Memory2

Finally, Processor2 writes the data to Memory2 using memory writed operation.

7 Conclusion and future work

This report shows our FIFO-based general transducer for any target interface. The transducer can communicate between two different protocols runs in different clock and different data width. We also extracted the parameters need to generate the transducer automatically. Future work will be automatic generation of the transducers from given parameters and routing, interrupt handling and slave device address generation in txducer master mode.

References

[1] D. Shin and D. Gajski. Interface Synthesis from Protocol Specification. Technical Report ICS-TR-02-13, University of California, Irvine, April 2002.