NISC Double-Handshake Communication Interface

Bita Gorjiara, Mehrdad Reshadi, Daniel Gajski

Technical Report CECS-06-05

March 2006

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

{ bgorjiar, reshadi, gajski}@cecs.uci.edu

NISC Double-Handshake Communication Interface

Bita Gorjiara, Mehrdad Reshadi, Daniel Gajski

Technical Report CECS-06-05

March 2006

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

{ bgorjiar, reshadi, gajski}@cecs.uci.edu

Abstract

To enable communication to a NISC component, three basic elements should be supported by NISC: interrupt handling, the interface mechanism, and the proper software drivers. In this document, we describe the NISC interface and the corresponding software drivers for three specific double-handshake protocols: one-word double handshake controlled by one FSM, one-word double handshake with two FSMs and burst-mode double-handshake protocols. The protocols are designed for communication over a shared bus.

Table of Contents

1 NISC communication basics 4
2 One-word double-handshake NISC interface with one FSM 4
3 One-word double-handshake NISC interface for shared buses with two FSMs 7
4 Burst double-handshake NISC interface for shared bus with two FSMs 9

NISC Double-Handshake Communication Interface

Bita Gorjiara, Mehrdad Reshadi, Daniel Gajski

1 NISC communication basics

To enable communication to a NISC component, three basic elements should be supported by NISC: interrupt handling, the communication interface (CI) unit, and the proper software drivers. In this document, we describe the NISC interface and the corresponding software drivers for three specific double-handshake protocols: one-word double handshake controlled by one FSM, one-word double handshake with two FSMs and burst-mode double-handshake protocols. The protocols are designed for communication over a shared bus (Figure 1).

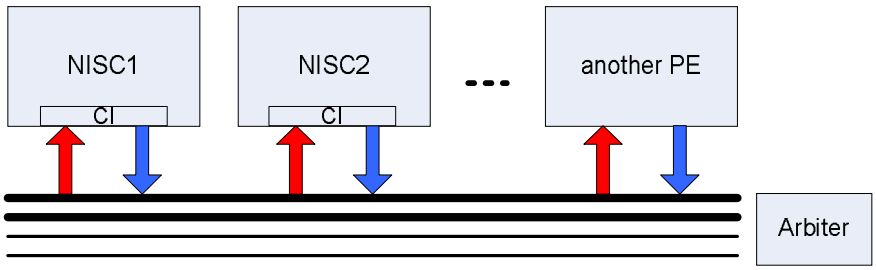

Figure 1. NISCs communicating to a shared bus through Communication Interfaces (CI)

2 One-word double-handshake NISC interface with one FSM

In this section, the implementation of CI for a simple double-handshake protocol is presented. In this protocol one word is transmitted per bus transaction. Figure 6 shows a simple NISC architecture and the corresponding CI. NISC provides two register (i.e. *Addr*, *Din*) to temporarily store address and data that must be sent. Furthermore, NISC controls the interface using two control signals *StartSend* and *ReceiveDone*. The interface has one internal register (i.e. *Dout*) to store the received data. Note that *Din* and *Dout* need to be separated in order to avoid conflict between NISC and the interface. The interface, also, provides feedback to the NISC about its internal status through signals *SendBusy* and *Interrupt*. As shown in Figure 2, the handshaking protocol on the shared bus is implemented by an FSM.

Figure 3 shows the pseudo code that runs on NISC and acts as a driver API for sending a word to a target component identified by *receiverAddr*. Once a program calls the *Send_Driver*, the driver checks the *SendBusy* signal to make sure that the interface is not busy with sending a previous message (line 3). Then, the driver writes the data and address of receiver to the *Din* and *Addr* registers (line 4, 5). These registers are controlled by the CW register of NISC. Next, the driver issues *StartSend* for the FSM (line 6) and returns to the calling program.

Figure 5 shows the state and timing diagrams of the FSM. As soon as the StartSend becomes 1, the Send FSM sets the *SendBusy* signal to make sure that NISC will not request another send until the current one is finished. The Send FSM also sets the *Request* signal and waits for the bus grant (state S1). Once the *Grant* signal becomes 1, it puts the data and address on the bus, raises the *Ready* signal and waits for the acknowledgement from the receiver (state S2). When the *Ack* signal becomes 1, the Send FSM lowers the *Ready* signal (state S3) and then goes back to state S0 and releases the bus by lowering the *Request*. It also resets the *SendBusy* signal to allow NISC to send another word.

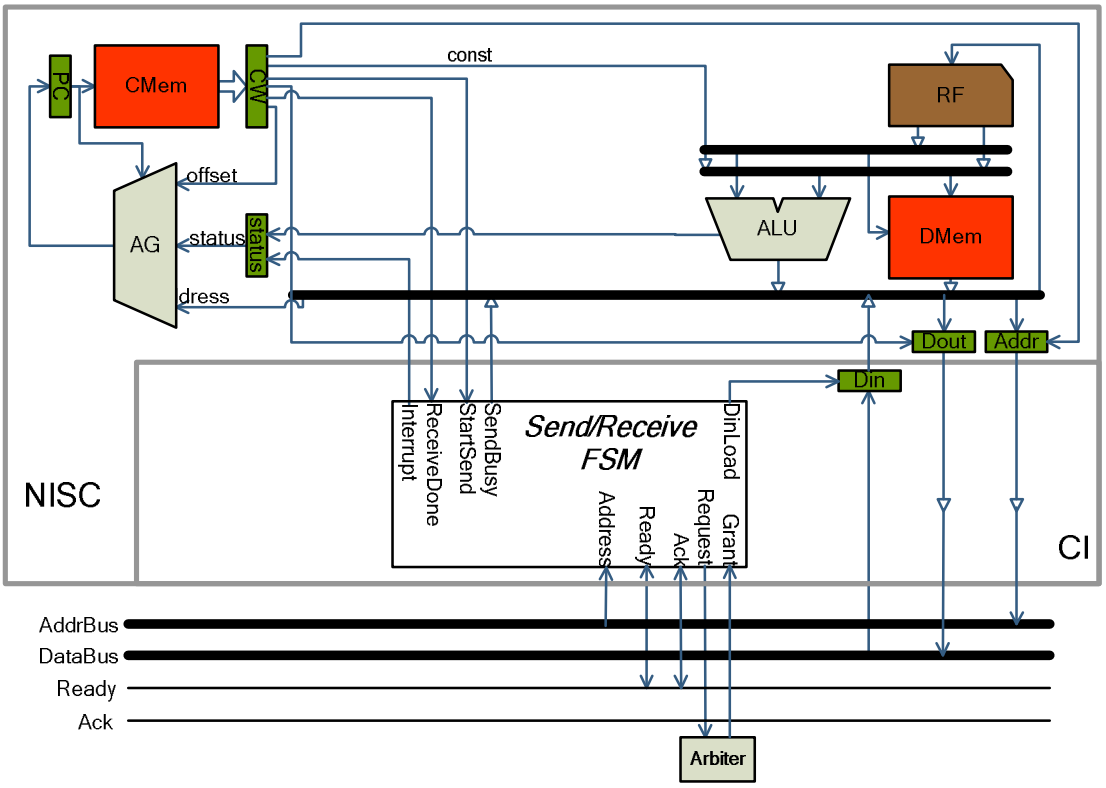

Figure 2. One-word double-handshake interface with one FSM

```
1   void Send_Driver(data, receiverAddr){
2     //wait while until the FSM is ready
3     while(SendBusy() ==1);
4     Addr = receiverAddr;
5     Dout = data;
6     StartSend ();
7   }
```

Figure 3. One-word DH send driver code

```
1   interrupt Receive_Interrupt(){
2     disable interrupt
3     some_buff = Din;
4     ReceiveDone();
5     enable interrupt
6   }
```

Figure 4. One-word DH receive driver code

On the receiver side a similar FSM (shown in Figure 5) monitors the address bus and the *Ready* signal to find out when to start reading data (State R0). If the address matches with its own address, and the *Ready* signal is set, then the FSM stores the data into *Din* register, and interrupts the NISC by raising the *Interrupt* signal (state R1). At this point, the Receive FSM sets the *Ack* signal to high and waits for the *Ready* signal to become 0 (state R2). Afterwards, in state R3, the FSM resets *Ask* signal and waits for the *ReceiveDone* to become 1, which indicates that the NISC has finished reading the *Din*. Once *ReceiveDone* signal become one, the FSM goes back to the state S0 and waits to send or receive another word.

Figure 4 shows the interrupt handler for receiving one word from the bus. First, it disables the interrupt, and then reads the content of *Din* and stores it into a local memory or register file. Next, it notifies the FSM by calling *ReceiveDone*() function which raises the *ReceiveDone* signal. Finally, it enables the interrupt again.

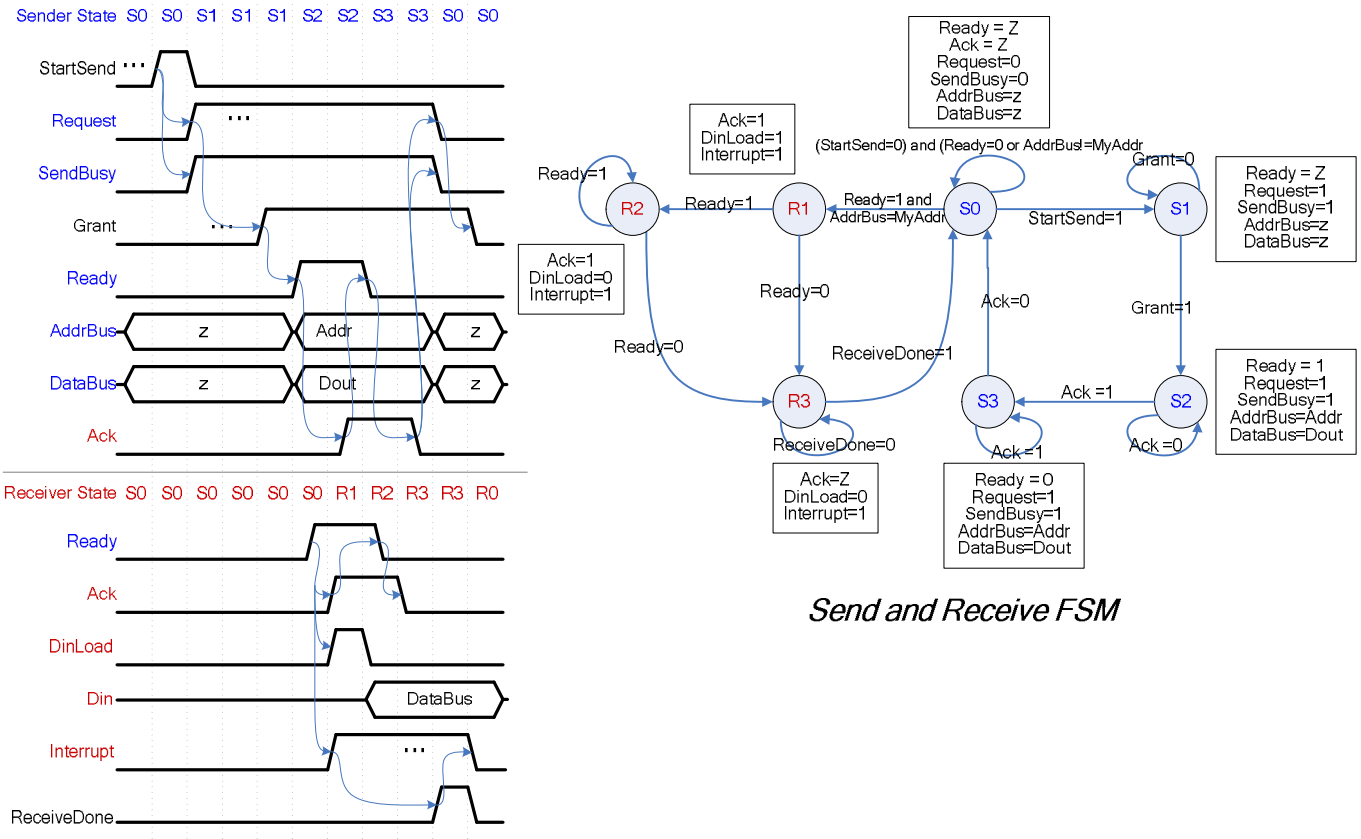

Figure 5. Send/Receive FSM and timing diagrams for a one-word DH protocol

3 One-word double-handshake NISC interface for shared buses with two FSMs

In implementation of CI, we assume the protocol stays the same, however, the FSM is partitioned to two FSMs: one for send and one for receive. The advantage with this approach is more parallelism, and hence better performance. Figure 6 shows the block diagram of NISC and the CI. Similar to the previous implementation, NISC provides two registers to store address and data as well as two control signals *ReceiveDone* and *StartSend*. The communication interface contains an internal register (*Din*), and provides feedback to the NISC architecture through *SendBusy* and *Interrupt* signals.

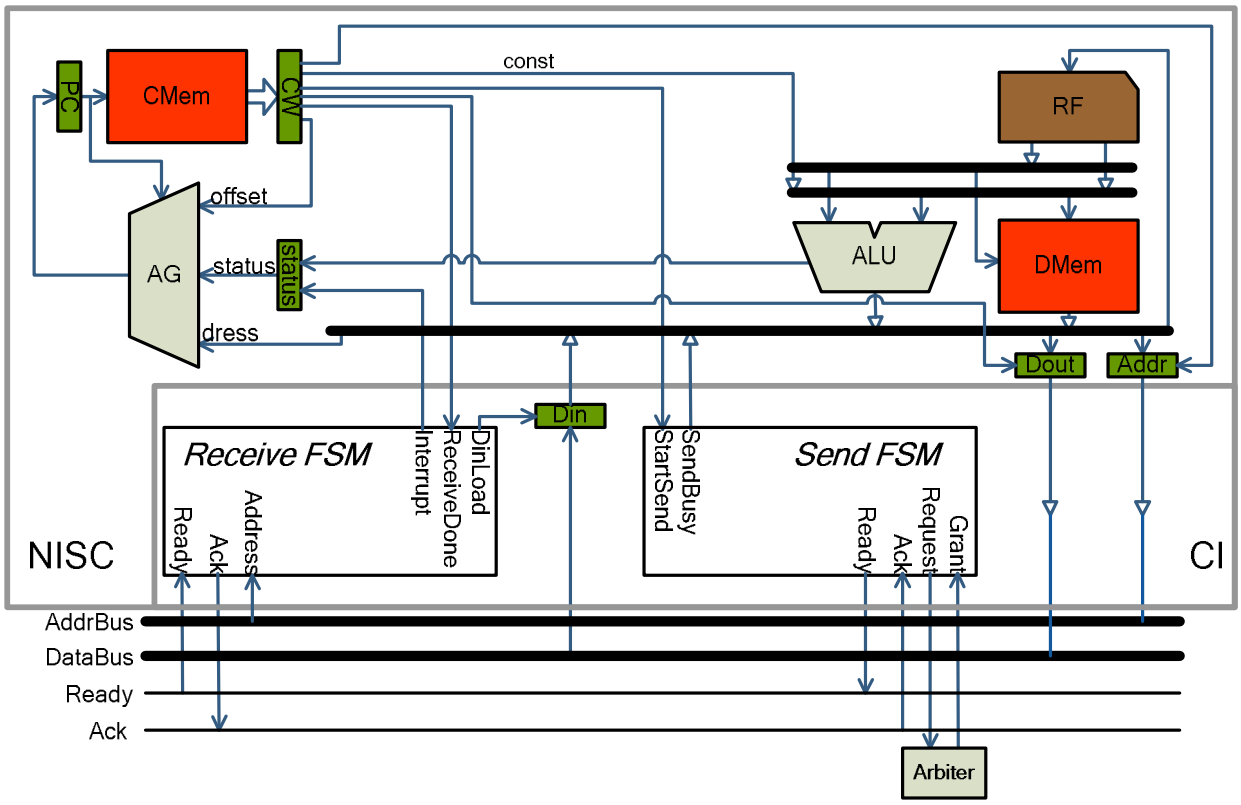

Figure 6. One-word double-handshake interface with two FSMs

Figure 7 shows the pseudo code that runs on NISC and acts as a driver API for sending a word to a target component identified by *receiverAddr*. Once a program calls the *Send_Driver*, the driver checks the *SendBusy* signal to make sure that the interface is not busy with sending a previous message (line 3). Then, the driver writes the data and address of receiver to the *Din* and *Addr* registers (line 4, 5). These registers are controlled by the CW register of NISC. Next, the driver issues *StartSend* for the Send FSM (line 6) and returns to the calling program.

Figure 9(a) shows the state and timing diagrams of the Send FSM. As soon as the StartSend becomes 1, the Send FSM sets the *SendBusy* signal to make sure that NISC will not request another send until the current one is finished. The Send FSM also sets the *Request* signal and waits for the bus grant (state S1). Once the *Grant* signal becomes 1, it puts the data and address on the bus, raises the *Ready* signal and waits for the acknowledgement from the receiver (state S2). When the *Ack* signal becomes 1, the Send FSM lowers the *Ready* signal (state S3) and then goes back to state S0 and releases the bus by lowering the *Request*. It also resets the *SendBusy* signal to allow NISC to send another word.

```
1   void Send_Driver(data, receiverAddr){
2     //wait while until send FSM is ready
3     while(SendBusy() ==1);
4     Addr = receiverAddr;
5     Dout = data;
6     StartSend ();
7   }
```

Figure 7. One-word DH send driver code

```
1   interrupt Receive_Interrupt(){
2     disable interrupt
3     some_buff = Din;
4     ReceiveDone();
5     enable interrupt
6   }
```

Figure 8. One-word DH receive driver code

The Receive FSM (shown in Figure 9(b)) monitors the address bus and the *Ready* signal to find out when to start reading data (State R0). If the address matches with its own address, and the *Ready* signal is set, then the Receive FSM stores the data into *Din* register, and interrupts the NISC by raising the *Interrupt* signal (state R1). At this point, the Receive FSM sets the *Ack* signal to high and waits for the *Ready* signal to become 0 (state R2). Afterwards, in state R3, the FSM resets *Ask* signal and waits for the *ReceiveDone* to become 1, which indicates that the NISC has finished reading the *Din*. Once *ReceiveDone* signal become one, the FSM goes back to the state R0 and waits to receive another word.

Figure 8 shows the interrupt handler for receiving one word from the bus. First, it disables the interrupt, and then reads the content of *Din* and stores it into a local memory or register file. Next, it notifies the Receive FSM by calling *ReceiveDone*() function which raises the *ReceiveDone* signal. Finally, it enables the interrupt again.

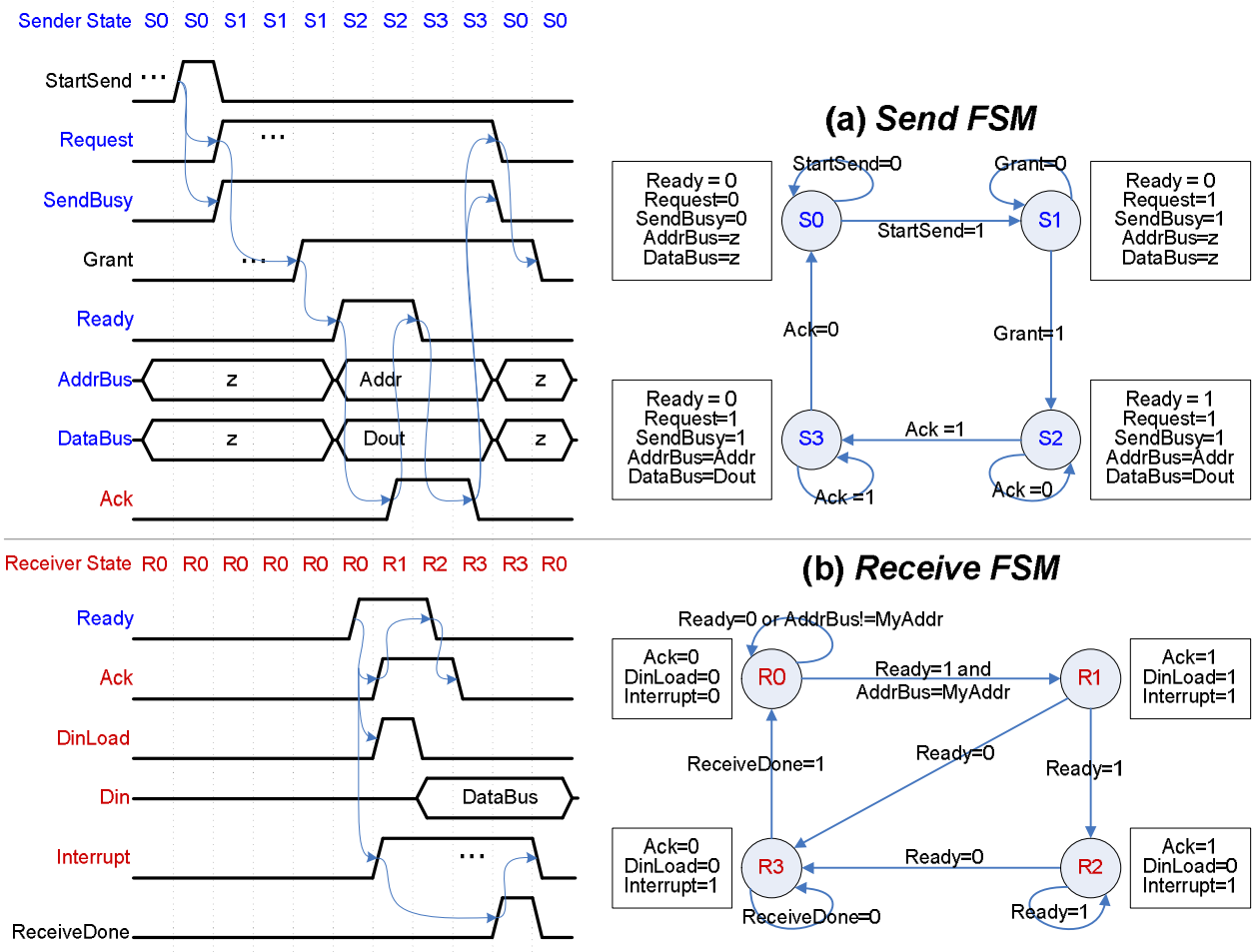

Figure 9. (a) Send FSM and , (b) Receive FSM state and timing diagrams for a one-word DH protocol

4 Burst double-handshake NISC interface for shared bus with two FSMs

In the burst mode protocol described here, we assume that an initial handshaking is done at the beginning of the communication and then *N* words are transmitted in *N* consecutive cycles without any further handshaking. To implement such a burst-mode communication protocol, we need to have two queues with the size of the burst message. Also, the clock cycle of all Send and Receive FSMs must be the same. However, different NISCs may have different clock cycles. Figure 10 shows the NISC architecture and the corresponding CI for burst mode protocol.

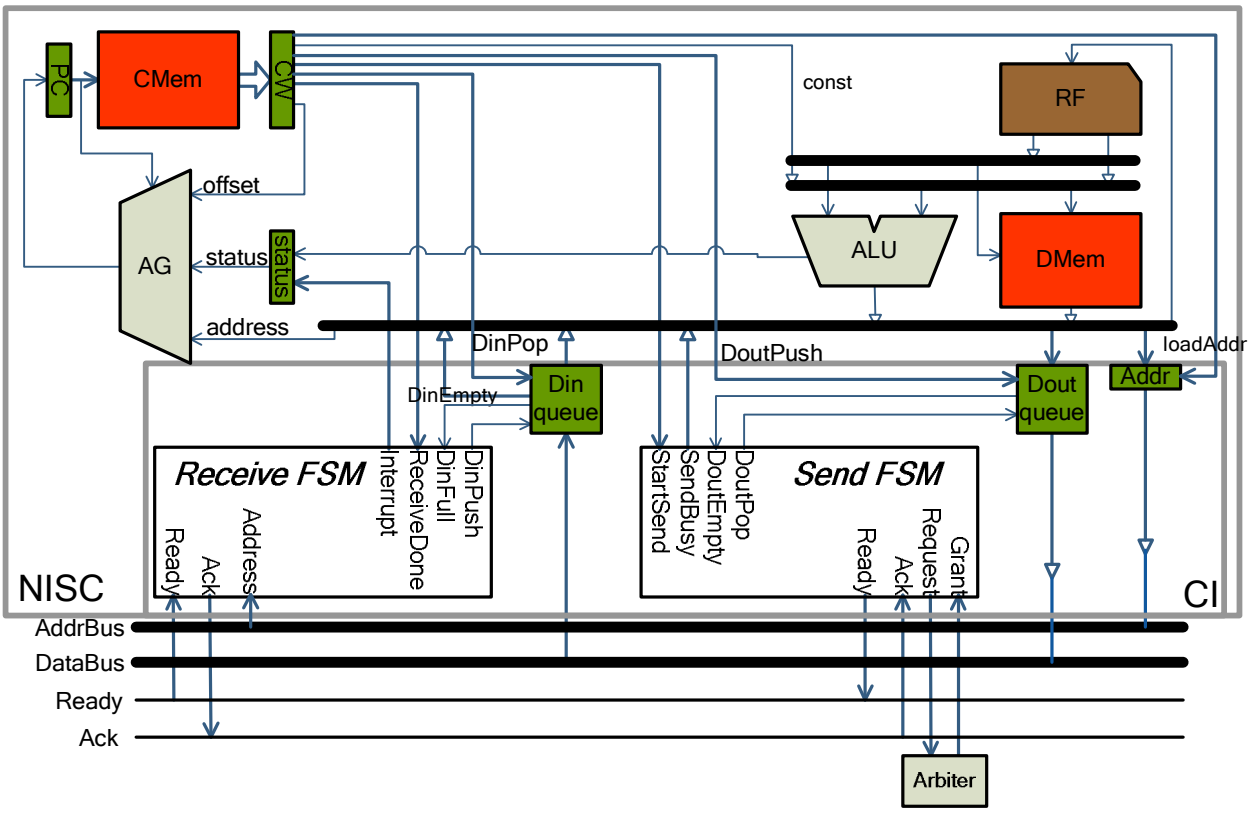

Figure 10. Burst double-handshake interface for NISC

To send data, the program calls the *Send_Driver* from the driver (shown in Figure 11). This function first loads the receiver address (*receiveAddr*) into the *Addr* register (line 3), and then reads multiple words of data from memory stored in *memAddr* and pushes them into the *Dout* queue (line 5). Next, the driver issues the *StartSend* command to the Send FSM (line 6).

The Send FSM (shown in Figure 13(a)) waits for the StartSend command, and then raises the Request signal to get the bus access (state S1). Once the bus is granted (*Grant*=1), the Send FSM puts the address on the bus and raises the *Ready* signal (state S2) and waits for the receivers acknowledgement. Once *Ack* becomes 1, the Send FSM puts the data of *Dout* queue on the bus one word per cycle until the queue becomes empty (state S3). Afterwards, the Send FSM lowers the *Ready* signal and waits for the receiver to lower the *Ack* signal (state S4). Next, it releases the bus and goes to state S0.

```
1  void Send_Driver(n, memAddr, receiveAddr){
2   while(SendBusy() == 1);
3   LoadAddr(receiveAddr);
4   for(i=0; i<n; i++)
5     PushDout(memAddr[i]);
6   StartSend();
7  }
```

Figure 11. Burst DH send driver code

```
1  interrupt Receive_Interrupt(){
2   disable interrupt
3   some_buffer[0] = Top();
4   int i=1;
5   while (DinEmpty()==0){
6     PopDin();
7     some_buffer[i++] = Top();
8   }
9   ReceiveDone();
    enable interrupt
   }
```

Figure 12. Burst DH receive driver code

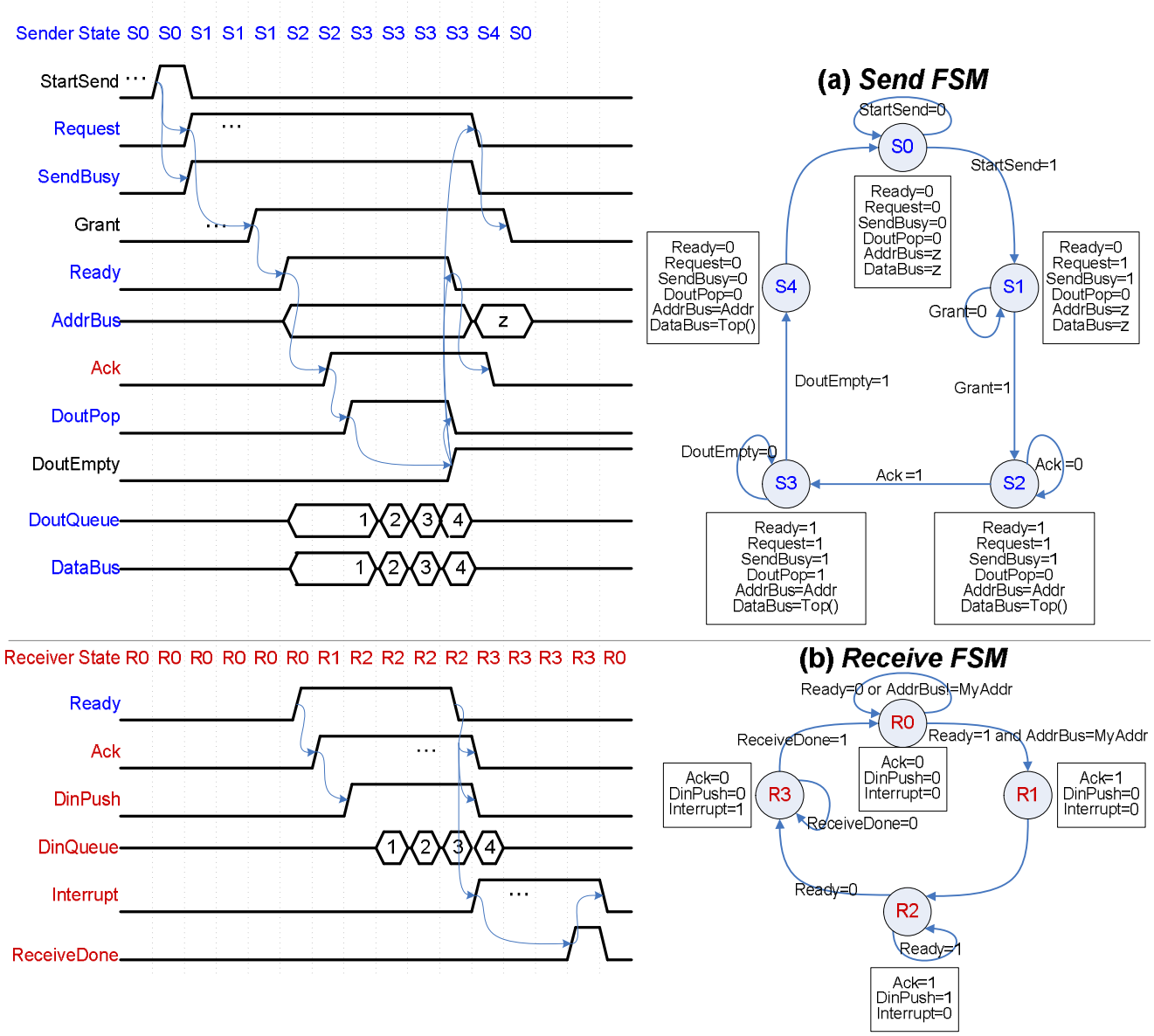

Figure 13. (a) send, (b) receive state diagrams and timing diagrams for a burst communication protocol

On the receiver side, the Receive FSM (shown in Figure 13(a)) monitors the address bus and the *Ready* signal to detect when to start reading data (State R0). If the address matches with its own address, and the *Ready* signal is set, then the Receive FSM raises the *Ack* signal. Then it waits for one cycle for the first data to be placed on the bus and then for the next several cycles it reads from the bus one word per cycle as long as the *Ready* signal remains 1 (state R3). The data is pushed into the *Din* queue in every clock cycle. Once *Ready* becomes 0, the Receive FSM lowers the *Ack* signal (state R4). It also interrupts the NISC and waits until all data are read from the queue. In this way, the receive queue will be empty for the next packet of data.

Figure 12 shows the receive interrupt routine. First, it disables the interrupt and then pops the data from the queue and stores it in a local memory until the queue becomes empty. Finally, it notifies the Receive FSM that the reading is finished and re-enables the interrupt.