Communication Design for No Instruction Set Computer

Jelena Trajkovic and Daniel Gajski

Technical Report CECS-05-09
July 25, 2005

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{jelenat,gajski}@cecs.uci.edu

Abstract

Increasing application demands flexible and fast components for their implementation. The No Instruction Set Computer (NISC) is one such component that provides reconfigurable data path and programmable control to drive the data path. This report addresses implementation of communication between NISC-style components. Our communication architecture contains of a FIFO queue that buffers data being sent from one NISC to another. We present the design of the FIFO and the data transfer methods used by the NISCs to communicate using this FIFO. Our design serves as a template for implementation of communication in NISC based systems.

Contents

1. Introduction and Motivation **1**

2. System Architecture **1**

2.1 Data Format 2
2.2 FIFO 2
2.3 NISC 3

3. Data Transfer Operations **4**

3.1 Data Send 4
3.2 Data Receive 5

4. Conclusions **5**

List of Figures

1 Top level view of system consisting of arbitrary number of NISCs 2
2 Connection details for system with two NISCs 3
3 Data format 3
4 Detailed system architecture 6
5 FIFO FSM 7
6 Data Send Operation Algorithm (NISC) 8
7 Timing diagram for Data Send Operation 9
8 Data Receive Operation Algorithm (NISC) 10
9 Timing diagram for Data Receive Operation 11

Communication Design for No Instruction Set Computer

Jelena Trajkovic and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

Increasing application demands flexible and fast components for their implementation. The No Instruction Set Computer (NISC) is one such component that provides reconfigurable data path and programmable control to drive the data path. This report addresses implementation of communication between NISC-style components. Our communication architecture contains of a FIFO queue that buffers data being sent from one NISC to another. We present the design of the FIFO and the data transfer methods used by the NISCs to communicate using this FIFO. Our design serves as a template for implementation of communication in NISC based systems.

1. Introduction and Motivation

With the rise in complexity and performance demands of modern applications, designers look forward to new options for their implementation. The ideal implementation would be one that combines the flexibility of software with the performance of hardware. One such attractive implementation option is the No Instruction Set Computer (NISC). It provides a fully customizable and reconfigurable data path and programmable control to drive the data path. Thus, each implementation is hardware like in performance, yet debuggable like software.

In order to satisfy application needs, designers may compose their system using several NISCs. Generally, different NISCs may have different clock speeds. Therefore, communication elements are required to allow exchanging data amongst such NISCs. To address this problem, we propose a simple yet effective solution: using a First-In-First-Out (FIFO) queue for sending and receiving data from one NISC to another. In this report, we present a solution for a system with any number of NISC processors. We also define requirements for each NISC processor in order to support the proposed communication architecture. Similar work has been discribed in [3], where the architecture of a general transducer and methodology for its automatic genaration was proposed.

Figure 1 shows the top level view of a system consisting of N NISCs communicating through a FIFO queue. Each NISC is connected to a common bus and its respective arbiter. Assume that NISC1 wants to send data to NISC2. The I/O Bus is used to transfer data from NISC1 to FIFO, using double handshake protocol. Before the data transfer, NISC1 needs to obtain bus from the arbiter. In order to identify the receiver, a header is prepended to the data containing the address of the recepient, i.e. NISC2. Once the data has been propagated to top of the FIFO, the receiver(NISC2) is notified. Upon notification, NISC2 initiates reading data from the top of FIFO. The details of the system components, and the send and receive procedures will be described in the following sections. In Section 2 we will give the system overview, describe data format and components used. Section 3 presents details of data send and receive operations, and we conclude this report with a summary in Section 4.

2. System Architecture

Details of the communication architectures are shown on the example of a system consisting of two NISC processors, shown in Figure 2. In order to access the bus, *NISCi* sends request to bus arbiter using *Reqi* line. Any centralized bus arbiter can be used. If bus access is granted to the particular *NISCi*, *Arbiter* sets value on the *Granti* line. The line stays high during the entire time *NISCi* holds the bus. Once the data transfer has been completed, *NISCi* drops a request by putting 0 on *Reqi* line, and the *Arbiter* replies by dropping the *Granti* and it releases the bus.

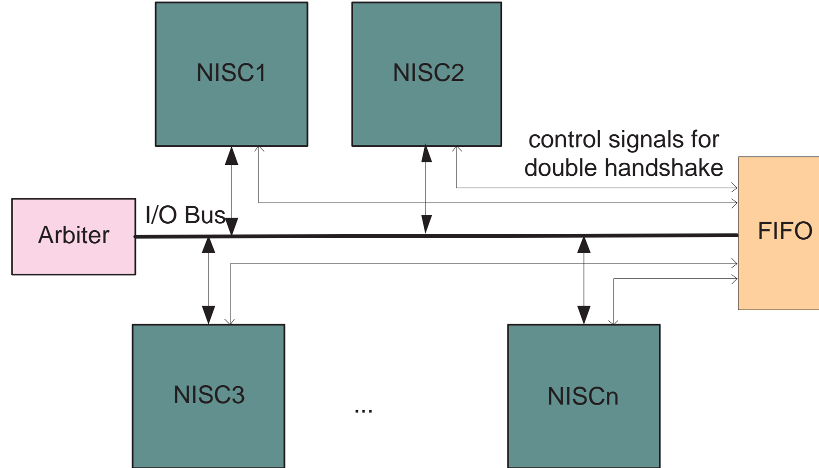

Figure 1. Top level view of system consisting of arbitrary number of NISCs

Once the bus is obtained, *NISCi* signals to the FIFO that it is ready to start the data transfer by putting *Readyi* high and 0 or 1 on *R/W* in order to read to or write from the *FIFO*. *FIFO* consists of a controller (*FIFO FSM*) and a data path (*FIFO Data Path*). Ready and R/W signals are input in the (*FIFO FSM*), as well as *Empty*, *Full*, *Size* and *Q_avail* that come from *FIFO Data Path*. *Empty* and *Full* show the state of memory in the data path, while *Q_avail* gives size (in data words) of the available memory space. *Size* is the number of data words that are to be sent or received. FIFO FSM generates read or write request to the queue by asserting (0,1) or (1,1) on (*Rd/Wr*, *Enable*) lines. *FIFO* also connects to the *I/O Bus* and the *ID Bus*. The former one is used for data transfer, and the latter one is used to broadcast the receiver's address. Detailed explanations of send and receive operation modes are given in Sections 3.1 and 3.2.

2.1 Data Format

Before sending, data has been formatted into the packet as shown in Figure 3. A one word long header is prepended to the *Data value*. The header consists of three fields: number of words in the packet (*Size*), receiver's address (*Addr*) and the data identifier (*ID*). The sender is responsible for assembling the packet before starting the send procedure, and the receiver is responsible for disassembling the packet upon receiving. The details of assembling/disassembling are out of the scope of this report. The packet header is also used by the FIFO. *Size* is checked to determine if there is sufficient space available in the queue's memory while receiving and to count the number of words transferred.

2.2 FIFO

Details of FIFO and NISC architectures are shown in Figure 4. As mentioned above, FIFO consists of a controller (*FIFO FSM*) and a data path (*FIFO Data Path*). *FIFO Data Path* consists of a RAM memory module (*Mem*), that stores the data, and up/down (n+1) bit counters, *Front* and *Back*, that point to the head and tail of the queue respectively. When data is to be written to the RAM, the *Rd* signal enables incrementing *Front* counter and selects (using *Selector*) the output of the *Front* counter (n lower bits) as the RAM's address. In case of read access, *Wr* signal and *Back* counter are used instead of *Rd* and *Front*. External *Enable* and *Rd/Wr* signals are supplied to the RAM's *CS* (chip select) and *RWS* (read write signal). Data is supplied to/from *I/O Bus*. The comparator output, together with the most significant bit of *Front* and *Back* counters, are used to determine if the FIFO queue is full or empty, thus generating signals *Full* and *Empty*. *Size* and *Q_avail* are also up/down counters. *Size* is the size of the data packet being written or read, and is decremented each time a new data word is transferred. *Q_avail* is the number of free words in memory, and is incremented/decremented depending on the nature of data transfer. *Size, Addr* and *ID* all compose a storage for the data header. More on FIFO design and its operation can be found in [1].

The operation of *FIFO FSM* is shown in Figure 5. The in and out signals are described in Section 2. FIFO and NISC communicate using double handshake protocol. If there is no request from any NISC, or if the queue can not satisfy the request, FSM is in the *idle* state. Once a write request comes in, and if the queue is not full, the receive operation is started. The first data word is stored into *"Size, Addr, ID"* register, and the data size is checked against the available queue space.

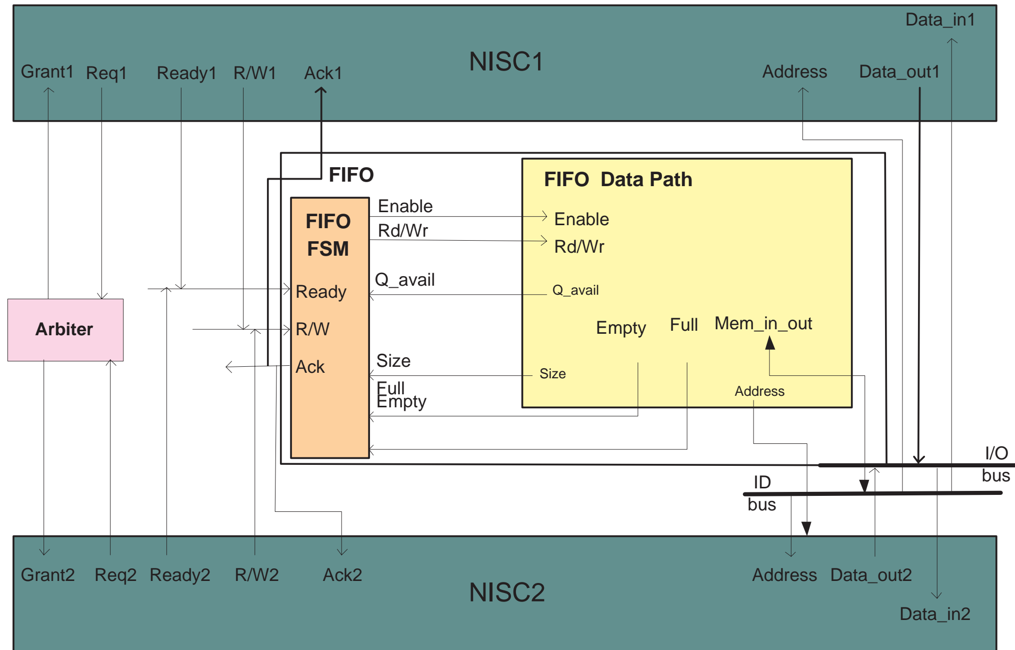

Figure 2. Connection details for system with two NISCs

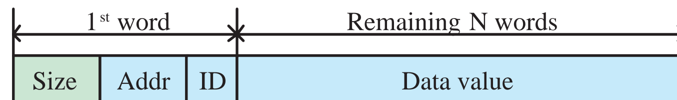

Figure 3. Data format

In case there is not enough space to accommodate the entire data packet, FSM transitions back to the *idle* state. Otherwise, the received data word is stored into the memory and an acknowledgment (*Ack*) is sent. The *Ack* signal stays high until the sender drops *Ready* signal. *Size* is decremented to show the number of data words left for transfer. If there are more data words to be transferred, FSM will wait till *Ready* becomes high again and then repeat the transfer (starting from *R2*). Once the entire packet has been received, FSM transitions to either *idle* or *S00a* state. It transitions to the *idle* state if sending has been previously initialized. If not, states *S00a* and *S00b* read out the first data word from the top of the stack. The first data word (see Figure 3) will be placed into *"Size, Addr, ID"* register and the address of the receiver will be broadcasted over *ID Bus*.

Upon recognizing its own address, the receiver will initiate receive operation (see Section 3.2). Once FSM receives the read request (*Ready = 1* and *R/W = 0*), it enables the first word to the *I/O Bus*, asserts *Ack* signal and waits for *Ready* signal to drop (state "S0"). *Size* is checked against 1, in order to continue the transfer of another word. Both *Size* and *Q_avail* are updated and a new word is read out in case where there are more data words to be transferred. Again, when transfer of the entire packet is completed, the first word of the following data packet is read out into *"Size, Addr, ID"* register, if the queue is not empty.

The order of data transfer requests can be arbitrary. A scenario where several NISCs one after another send data before any NISC receives data is possible. An extreme case of this scenario may block the entire system: no one can send, because the queue is full, until the data packet from the top of the queue is delivered. Note that the opposite scenario, where queue is empty, blocks only NISCs that are waiting for data.

2.3 NISC

Figure 4 also shows details of two NISC processors, namely NISC1 and NISC2. For the purpose of this work, these two processors are identical, where in general case they may be different. Still each NISC needs to have certain elements that will

allow it to communicate with the FIFO queue. The required elements will be defined in this and the following sections after, a brief introduction about NISC.

In contrast to a traditional, RISC-style processor, NISC does not have a decoding unit. The application program is compiled directly into a set of control signals, called Control Words (CW), that are stored into the program memory (*CMem* in Figure 4). Therefore an instruction set does not exist, allowing us to have fully customized data path. The control unit (on the left hand side of the NISC block diagram), consists of an address generator (*AG*), a program counter (*PC*) and a control word memory (*CMem*). All these elements perform the same duties as in a standard processor, except that a word that is read out of *CMem*, is supplied directly to a data path (on the right hand side of the NISC block diagram). The data path shown here is very simple; it consists of the register file (*RF*), the arithmetic-logic unit (*ALU*), the memory (*Mem*) and the pipeline registers (*R1, R2, R3, Addr, Data_i* and *Data_o*). In general the NICS data path may be non-pipelined or pipelined, have data forwarding and any number of pipeline stages. It also may have any number or type of functional units, such as alu, multiplier, multiplier/divider, multiply-and-accumulate or a custom functional unit. For more information on NISC, refer to [2, 4].

Registers *Data_in1* and *Data_out1* and the corresponding selectors are used to store data word for receive and send, respectively. The address decoder is a simple combinational circuit, that generates *Data_ready* signal after the FIFO broadcasts the address of the data packet on the top of the queue. This signal, together with *Grant1, Ack1*, *Timeout*, *More* and *ALU_status*, form the content of a *Status* register. Bits from the *Status* register are used by the address generator to compute "next PC" value. *Cnt* and *Size* are down counters with parallel load. Both values are loaded at the same time the first data word is loaded, just that *Cnt* is loaded only on data send. We determine initial value of *Cnt* such that we allow enough time for FIFO to transition from "idle" to "R3" state and generate *Ack*. Counter *Cnt* generates signal *Timeout* once it reaches 0. The hardware for loading *Size* is not shown due to the lack of space. The initial value of it is the same as upper M bits of Data_out on send or Data_in on receive; where M is bit-width of size field in the data header. The *Size* is decremented every time a transfer of the new word is initiated. If the size of remaining data is greater than 0, *More* signal is generated.

Details of how NISC executes data send and receive are discussed in the following section.

3. Data Transfer Operations

To be able to send or receive data, the original data needs to be prepended a header as explained in Section 2.1. For the purpose of this work, we will assume that the data is already been formated and stored in the memory (*Mem* in Figure 4). We will also assume that memory access takes one cycle; we will point out what needs to be changed in the other case. In this section, we will describe an example where *NISC1* sends data to *NISC2*. Algorithms for send and receive are shown in Figures 6 and 8. Send FSM and Receive FSM correspond to a sequence of control words that NISC processor executes. FSM representation is chosen for two reasons. First, it is intuitive and easy to follow. Second, the I/O operations, may be separated from the "basic" NISC core; i.e. a separate FSM or "NISC controller" may be responsible for the I/O. Independent I/O core will be discussed in our future work.

NISC may start the send or receive operation once any other data manipulation is finished. This means that the other parts of NISC are idle while I/O is being executed. This is the simplest way of dealing with nondeterministic duration of I/O operation in static scheduling. The other solution would be to assume some initial delay, then make a schedule for the best case scenario, and stall the processor if assumed latency is violated. This optimization will be a topic of our future work.

3.1 Data Send

The timing diagram for the Data Send operation is shown in Figure 7. Before we start executing control words specified by Figure 6, a memory read access needs to be initiated. That means, the memory address needs to be in the *Addr* register and memory read needs to be specified on the memory control inputs ("NS00a" and "NS00b"). After that is done, *NISC1* loads the first data word in the *Data_out1* register, sets *Req1* and waits for the *Grant1* from the *Arbiter*. Upon receiving bus grant, *NISC1* sets *Ready1* and *R/W* signals and enables data from register *Data_out1* to the bus. That way it signals to the FIFO that it wants to send the data. FIFO FSM (Figure 5) transitions form "idle" state to "R0" where it loads the first data word. If there is not enough space in the queue, *NISC1* never receives an acknowledgment, and upon generating the timeout, it restarts the operation ("NS2" and "NS9" in Figure 6).

The timing diagram shows the case where enough space is available. In that case, FIFO FSM stores data into RAM (FIFO state "R2") and in the following state sets *Ack1* high. *NISC1* is waiting for the acknowledgment ("NS2" in Figure 6). Once it is received, it resets *Ready1* and disables data output to the bus. After *Ack1* has been set back to zero (FIFO state "R4"),

NISC1 initializes new memory read (corresponds to “NS4”). In the case where there are more data words to be sent, the data is loaded to *Data_out1*, and the transfer is repeated (“NS5” to “NS7” in Figure 6). If memory *Mem* takes N cycles to respond, “NR4” needs to be executed N times. Control words that correspond to “NS0” and “NS6” may be used to load the next memory address.

Once all data words are transferred, *NISC1* drops the request and waits for the grant to drop, and that finishes the receive operation.

3.2 Data Receive

As described in Section 2.2, once FIFO has received data, it broadcasts the receiver’s address throughout *ID Bus*. Address decoder of *NISC2* recognizes its own address and generates *Data_ready* signal. In our example, shown in Figure 9, *Data_ready* is generated before the start of the receive sequence. In general, it may happen that *NISC2* has to wait (“NR0” in Figure 8) for *Data_ready*. *NISC2* than sends the bus request (*Req2*) and waits for *Grant2*.

After NISC2 receives the grant, it signals to the FIFO (“NR2” in Figure 8) that it is ready to receive the data. FIFO replies by setting *Ack1* high and providing the data to the bus. Upon receiving acknowledgment, *NISC2* enables data load into *Data_in2* register, and in the following cycle it drops the request and waits for the acknowledgment to drop (“NR4” in Figure 8). “NR3” and “NR5” are responsible for loading memory address and initiating the access, respectively. If memory *Mem* takes N cycles to respond, “NR5” needs to be executed N times.

If more words need to be transferred, *NISC2* restarts receive (“NR6,” “NR7,” “NR4” and “NR5”). Otherwise, as shown in the timing diagram, it drops *Req2* and waits for *Ack2* to drop in order to proceed with the program execution.

4. Conclusions

In this paper we present a simple and effective solution for communication between several NISCs implemented in a given system. We used a single FIFO queue to buffer the data being communicated. We defined and explained the communication architecture and methods needed to exchange data between NISCs via the FIFO queue.Our definition of the FIFO queue and the double handshake protocol serves as a general template for communication design of any NISC based system. As a result, we can quickly produce a communication architecture and methods for a given application running on multiple NISCs.

References

[1] D. Gajski. *Principles of digital design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[2] D. Gajski. Nisc: The ultimate reconfigurable component. Technical report, Technical Report TR 03-28, University of California-Irvine, October 2003.

[3] D. Gajski, H. Cho, and S. Abdi. General transducer architecture. Technical report, Technical Report TR-05-08, University of California-Irvine, July 2005.

[4] D. Gajski and M. Reshadi. Nisc application and advantages. Technical report, Technical Report TR 04-12, University of California-Irvine, May 2004.

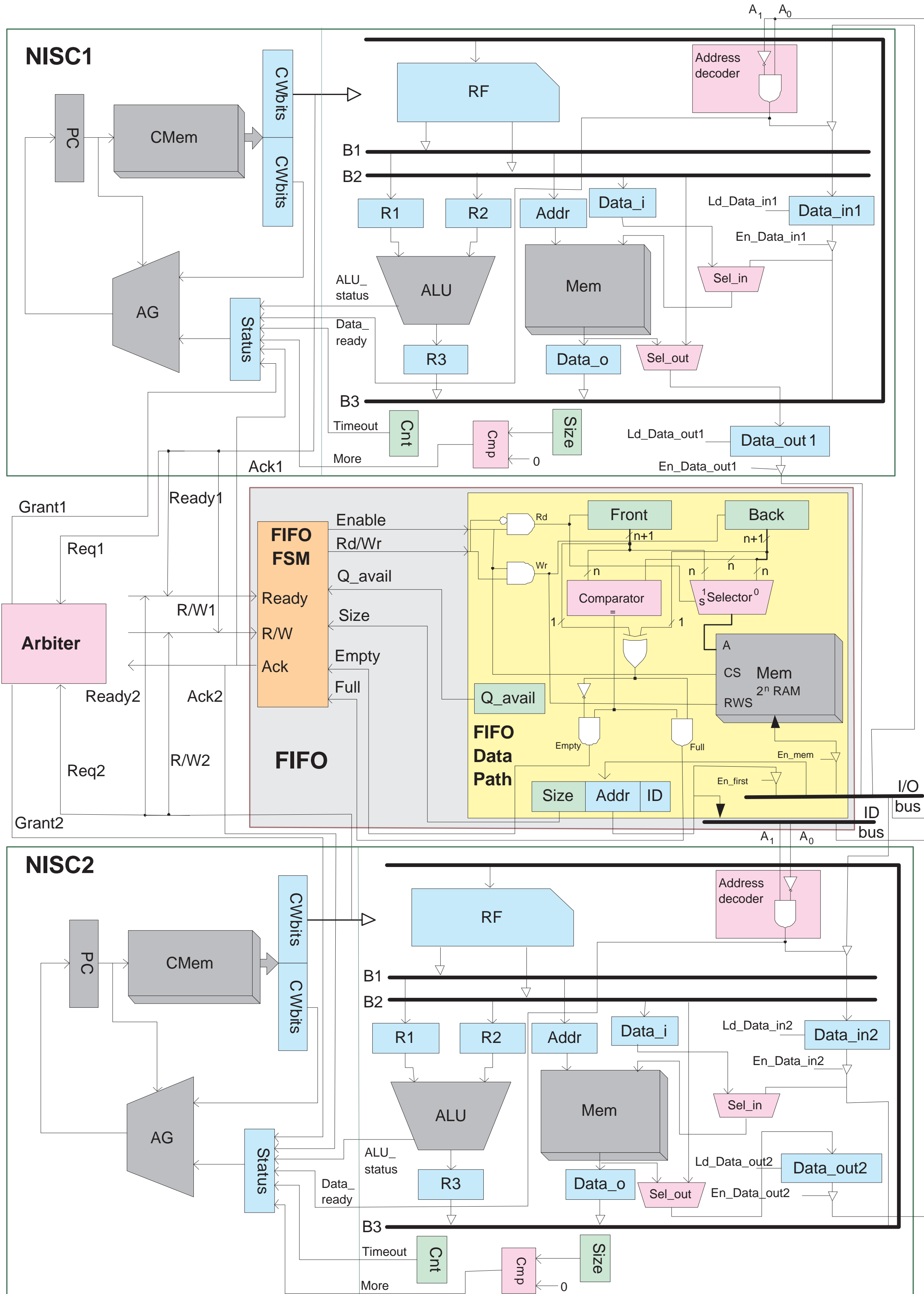

Figure 4. Detailed system architecture

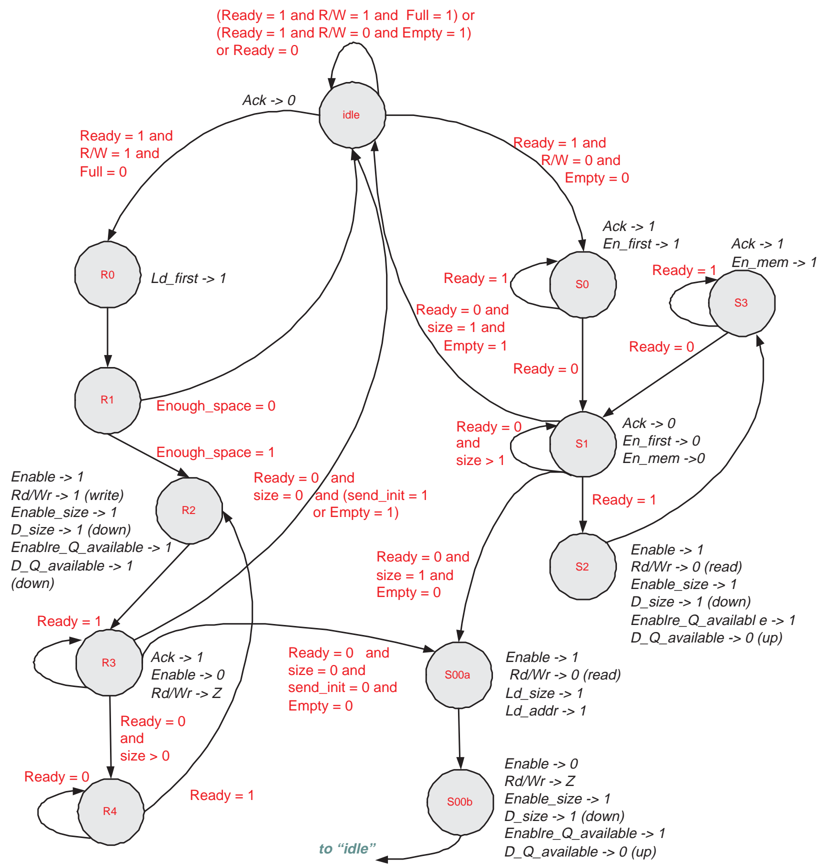

Figure 5. FIFO FSM

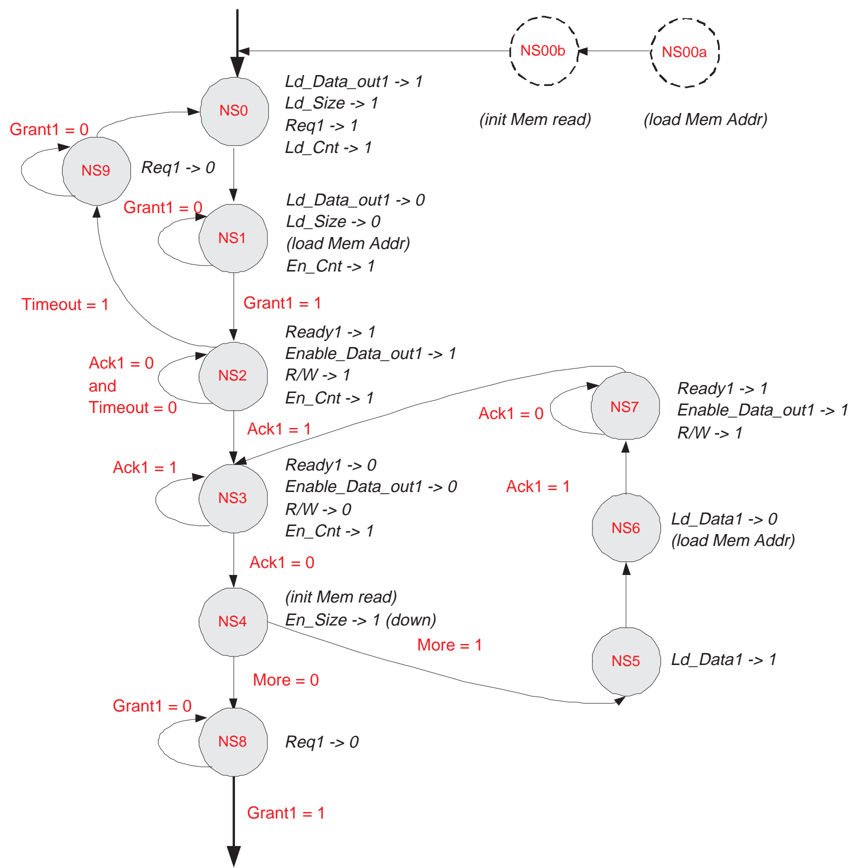

Figure 6. Data Send Operation Algorithm (NISC)

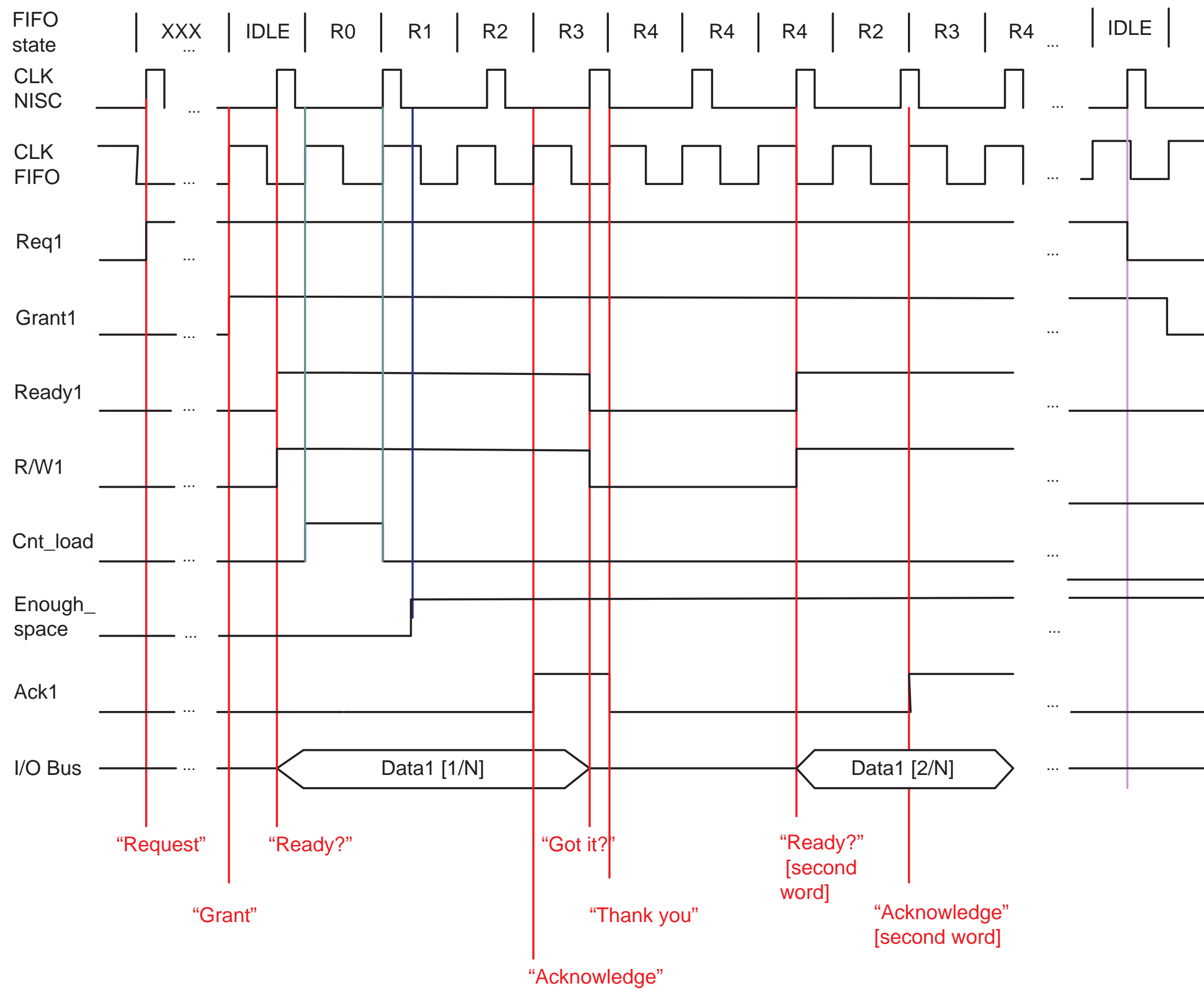

Figure 7. Timing diagram for Data Send Operation

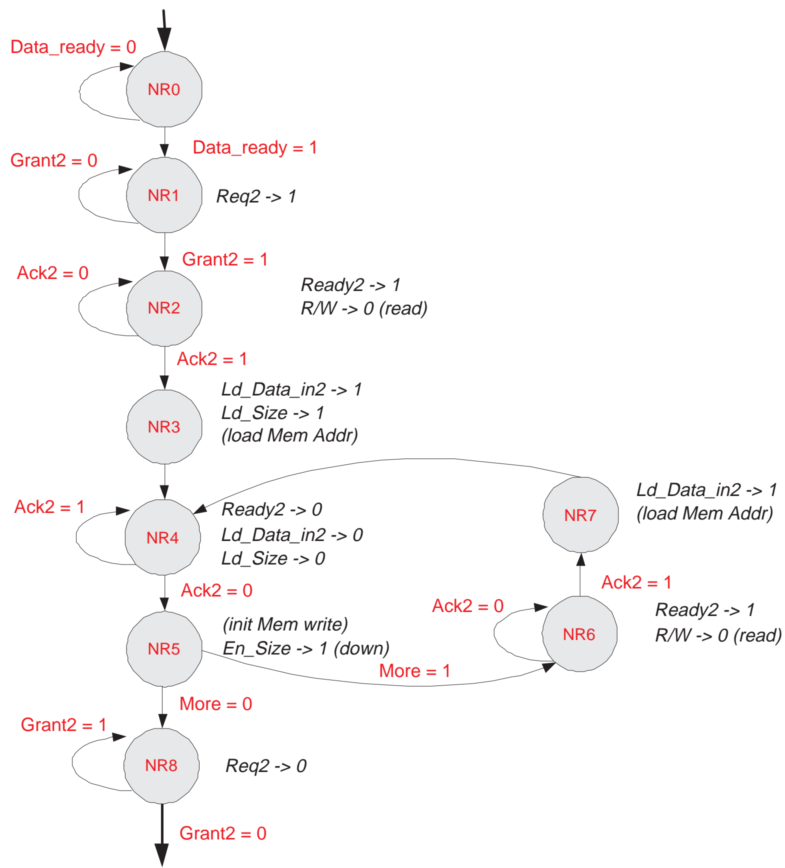

Figure 8. Data Receive Operation Algorithm (NISC)

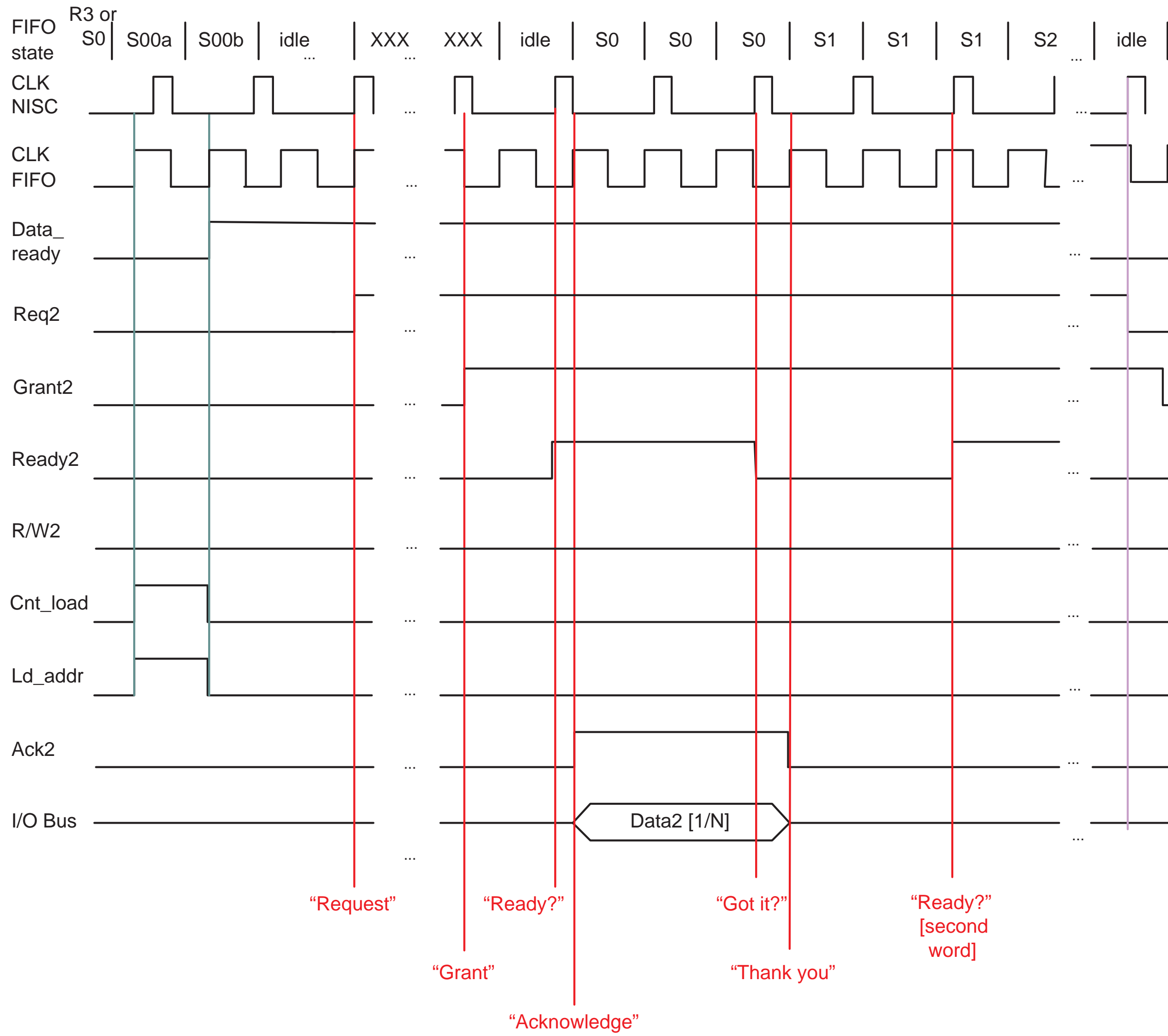

Figure 9. Timing diagram for Data Receive Operation