Modeling Guidelines for ASIC Reuse

Gaurav Aggarwal
Daniel D. Gajski

Technical Report UCI-ICS-98-03
March, 1998

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425.
Phone: (714) 824-8059

gaurav@ics.uci.edu
gajski@ics.uci.edu

Abstract

In this report, we discuss the various issues and problems associated with ASIC reuse. We describe the different models of communication between components and the essential issues in interfacing ASICs that use different communication protocols. We come up with guidelines that help in modeling for reuse. We also propose a new HDL, SpecC, that has the desirable characteristics for co-designing systems. This language is suited for ASIC reuse and overcomes the limitations of VHDL.

Contents

List of Figures

Modeling Guidelines for ASIC Reuse

Gaurav Aggarwal **Daniel D. Gajski**

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425.
Phone: (714) 824-8059

Abstract

In this report, we discuss the various issues and problems associated with ASIC reuse. We describe the different models of communication between components and the essential issues in interfacing ASICs that use different communication protocols. We come up with guidelines that help in modeling for reuse. We also propose a new HDL, SpecC, that has the desirable characteristics for co-designing systems. This language is suited for ASIC reuse and overcomes the limitations of VHDL.

1 Introduction

In the last decade, digital electronic systems have been growing in complexity and functionality very rapidly. Explorations must be done at the system level to reduce the number of objects that have to be dealt with. The design-turnaround time and design costs can be reduced by *reusing* existing designs. However, reusing application-specific ICs (ASICs) is not an easy task. Components need to communicate with other components in the system for data transfer and synchronization. ASICs may have to be redesigned when reuse is employed because of communication protocol mismatches. It is now obvious that ASICs must be modeled in a way that supports ease of reuse. In this report, we outline the guidelines for modeling ASICs with a goal of achieving reusability.

The report is organized as follows. We first describe the reuse problem and the *plug-and-play* capability in Section 2. This section also defines the concept of a channel and emphasizes the dichotomy of communication and computation. Section 3 gives an overview of the experiments performed to understand the issues involved in the reuse problem. We then present our guidelines for modeling in Section 4. Our critique of VHDL as a modeling language leads us to present SpecC as a preferred language for reuse in Section 5. Finally, we conclude in Section 6.

2 The Reuse Problem

The reuse problem is well illustrated by the *plug-and-play* feature. Consider a system shown in Figure 1 where it is desired to replace ASIC `B` with another ASIC `E` that provides the same functionality. This task is non-trivial in the general case because the communication protocols of `B` and `E` might be different.

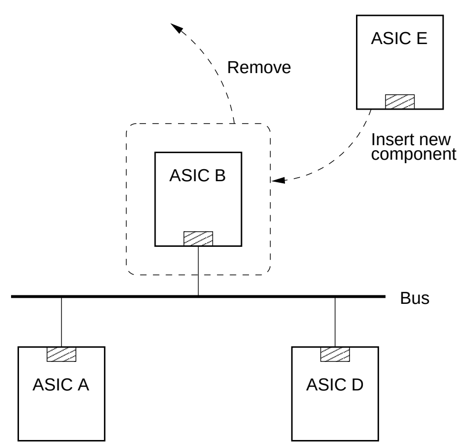

Figure 1: *Plug-and-play*: change an ASIC

Similarly, in Figure 2, `Bus 1` that connects ASIC `A` to ASIC `B` has to be replaced with a new bus, `Bus 2`. Again, this is not easy since the ASICs `A` and `B` used the `Bus 1` protocol earlier and now need to use the

`Bus 2` protocol.

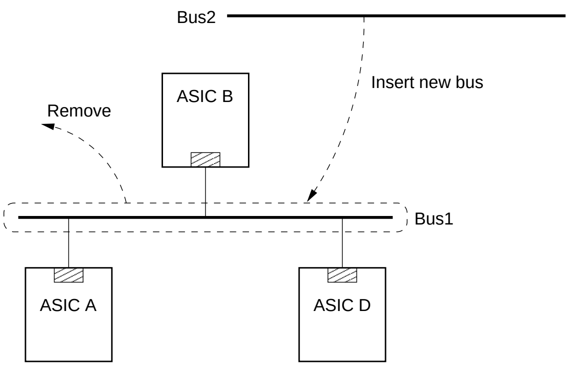

Figure 2: *Plug-and-play*: change a bus

The *plug-and-play* capability essentially means ease of replacing a component with a new component that has the same functionality but different implementation. A system configuration consisting of three ASICs, (`A, B, D`), is shown in Figure 3. It is desired that ASIC `B` be replaced with a new ASIC `E`. The new ASIC has the same functionality as `B` but possibly a different design, e.g. it may be pipelined, or may be area/power optimized. The new ASIC, in general, will use a different communication protocol. Hence, it cannot be directly connected in the existing system. A *protocol transducer* is required to translate the protocols. Thus, whenever an ASIC is replaced and there is a protocol mismatch, a new transducer has to be used.

Similarly, there may be protocol mismatches when the bus itself is replaced, e.g., a VME bus might be replaced by a PCI bus. A system configuration with three ASICs connected through a bus is shown in Figure 4. `Bus 1` is replaced with `Bus 2`. The new bus, `Bus 2`, uses a different communication protocol. Hence, transducers must be inserted for each component since the components used the older `Bus 1` protocol. This is shown in Figure 4 where three new transducers are required to connect the ASICs to the new bus. This is in contrast to replacing an ASIC where only one transducer might be modified or where only one new transducer has to be inserted. When a bus is replaced then all the components connected to the bus need new transducers.

2.1 Communication vs. Computation

The biggest obstacle to reuse is the communication protocol mismatch problem. As a first step towards reuse, the distinction between *computation* and *communication* must be emphasised. In an ASIC design model, **the communication portion needs to be separated from the computation portion**. If communication is mixed with the computation in a component then every time there is a mismatch in the communication protocols, the components have to be redesigned to reflect the new protocol. However, if communication is separated from the computation, then the basic core that defines the ASIC functionality remains the same. Only the communication block needs to be modified for the new protocol. This dichotomy between computation and communication is achieved naturally if *communication channels* are used and modeling is done at higher abstraction levels as explained below.

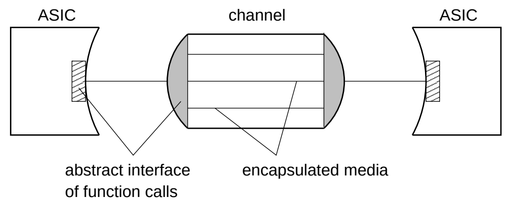

Figure 5: A generic communication model with a channel

A generic communication model [3] using a channel is shown in Figure 5. The main idea is to disassociate the detailed communication protocol Like the bus widths, number of control signals, timing and so forth from the ports of the component. The computation in the ASIC component makes calls to abstract function calls in order to read/write variables and exchange data. It does not even need to know what kind of channel will be used eventually for the actual communication. All that is required is that the channel conforms to the port interface. The port interface of the ASIC consists of function calls while the actual communication media is encapsulated in the channel. The data transport methods in the channel do the actual implementation of the protocol.

The port interfaces of ASICs would be comprised of function calls whose primary purpose is to transfer *information* across the port. It is upto the communicating ASICs to give it a meaning and interpret the information being transferred in different ways de-

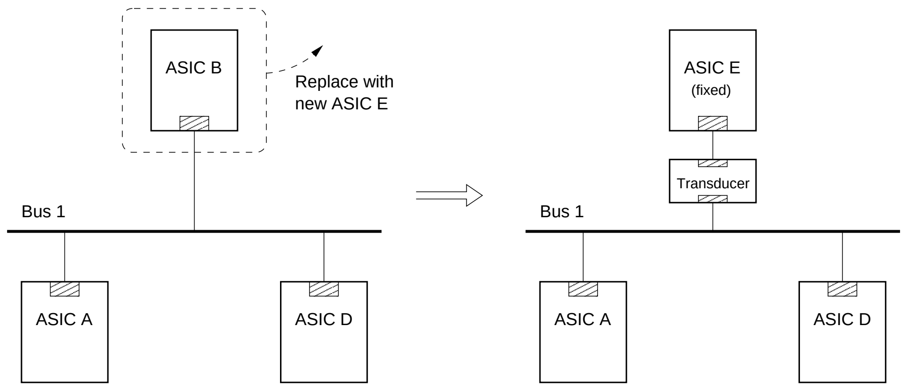

Figure 3: Effect of replacing an ASIC in a system

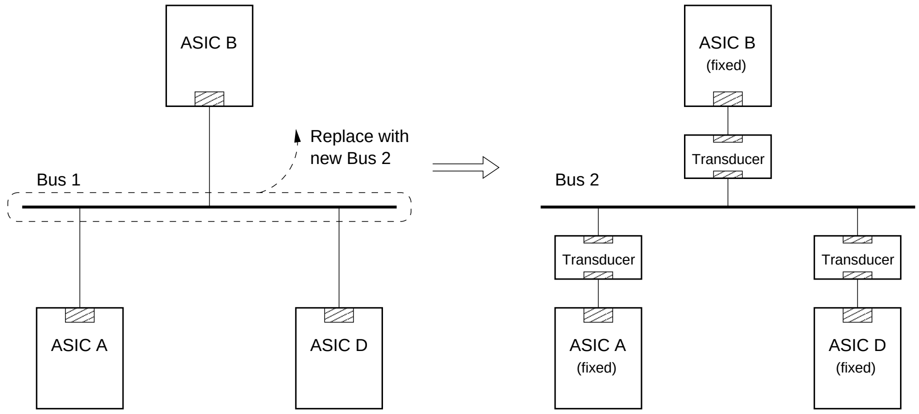

Figure 4: Replacing a bus necessitates change of transducers

pending on the protocol. Typical function calls would include `read()` and `write()`. An example is shown in Figure 6. A skeletal code fragment and port interface declaration is shown in Figure 6(a). The port interface consists of detailed bit level signals and the code includes the detailed timing. However, compare this to the code fragment shown in Figure 6(b). The port interface now consists of an abstract function call called `write()`. The tranfer protocol is encapsulated in this function call. The computation process in the ASIC does not use detailed timing protocol. It just invokes the function call provided in the port interface as shown on the line "`write (local);`" in Figure 6(b).

```
....
port ( ....
     clk    : in   bit;
     dout : out integer;
     start : out bit );
....

begin
     ....
     start <= '1';
     wait until clk = '1';
     dout <= local;
     start <= '0';
     ....
end;
```

(a)

```
....
port ( ....
     function write (data : integer);
     );
....

begin
     ....
     write (local);
     ....
end;
```

(b)

Figure 6: ASIC Port Interfaces: (a) Detailed Bit-level (b) Abstract Function calls

Generic function calls such as `read_word()` and `write_word()` can be used to transfer any kind of information. Thus, the sender and receiver of words can transfer data of various widths, control signals, exceptions etc. using such function calls. There is another option. The port interfaces can be modeled differently to actually reflect the kind of transaction occurring across the port. If such is the case, then the port interface will consist of function calls like `request_and_wait()`, `data_size()`, `component_exception()`, `terminate_operation()`, `priority_trigger()`, `component_idle()` etc.

The port interfaces do not include bit-widths and timing details. It is, therefore, easier to develop the protocol transducers at a behavioral level rather than at detailed bit level. It may even be possible to generate the transducers automatically. Secondly, generic transducers that do translations between typical protocols can be designed and stored in a library. Furthermore, customization of a generic transducer is simpler if the models are at a higher level and do not include timing details.

We next look at the implementation issues for systems containing channels. This will help us to determine the effect of using channels on the *plug-and-play* problem. Consider a typical system of two synthesizable ASICs as shown in Figure 7. ASICs `A` and `B`

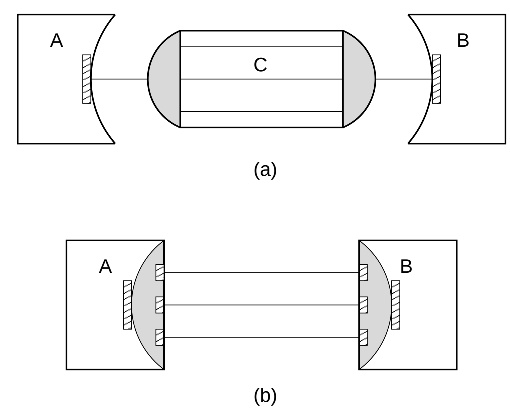

Figure 7: Channel inlining for synthesizable ASICs: (a) before inlining (b) after inlining

are synthesizable components and use a channel `C` for communication. When the system is implemented, the methods of the channel are **inlined** into the connected ASICs. Figure 7(a) shows the model at the higher level before the system is implemented. The channel methods are moved into the ASICs during implementation and the resulting model is shown in Figure 7(b). The bus wires that were originally encapsulated by the channel are now exposed.

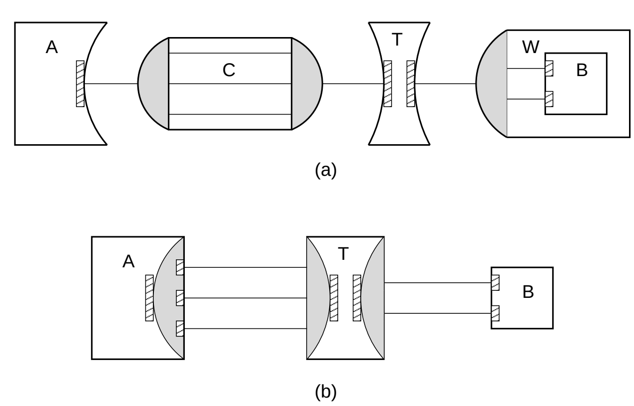

Figure 8: Channel inlining for a synthesizable ASIC and a fixed component: (a) before inlining (b) after inlining

When B in Figure 7 is replaced with a fixed component, we get a system as shown in Figure 8, where a synthesizable ASIC A communicates with B. We encapsulate B in a wrapper W since B is not synthesizable and has a low-level port interface. Thus, The function of the wrapper is to provide an abstract port interface for the fixed component, a port interface that consists of function calls instead of the detailed bit-level interface that a fixed component has. This model before system implementation is shown in Figure 8(a). The protocol of B as encapsulated by wrapper W may be different from that used by the channel C. A protocol transducer T will then have to be inserted between the channel C and the wrapper W. If wrapper W and channel C use the same protocol then the transducer will basically be an *identity* component or one that has just a memory element. The primary function of this transducer is to translate all communication transactions between the fixed component B and ASIC A. When the system is implemented, the methods on the side of the synthesizable ASIC are moved into the ASIC. The methods of the channel on the side of the fixed component are joined with the wrapper and implemented together with the protocol transducer as shown in Figure 8(b).

Thus, the process of IP reuse involves modeling components at a higher level and declaring port interfaces using function calls. ASICs with such generic interfaces are stored in a library. Any ASIC with the desired functionality can be chosen and plugged into the system model. An appropriate generic transducer is picked from the library and modified for doing protocol translation between this particular ASIC and the system being designed. This higher level system model is then refined and the channel methods are inlined into the ASIC, as discussed above. The transducer is also refined during system implementation along with other components.

Similarly, the channel can be replaced with any other bus channel in the library as long as it provides the abstract function calls used inside the component and specified by the port interface, e.g., a channel might encapsulate a VME bus and provide an ASIC with methods for reading and writing word a from memory. This channel can be replaced by another that encapsulates a PCI bus for instance but provides the same interface that consists of the read/write methods. In addition, the channel can also be incrementally refined. It can provide communication methods using simple function calls and shared variables in the earlier phases of specification and partitioning. Later the channel can be refined into a detailed timing model that reflects an actual physical bus protocol.

2.2 Communication Implementations

Another important issue for reuse is the prescence of a rigorous methodology to design the system from a high level specification. The methodology will consist of well defined transformations and intermediate models. This provides documentation and helps in reuse since reuse is difficult both at the final implementation level which is too detailed as well as the high level specification which is too abstract.

A generic methodology for hardware-software codesign for ASICs [1] is shown in Figure 9. The design steps include *allocation, partitioning, scheduling* and *communication refinement*, which form the synthesis flow of the methodology. The task of allocation determines the number and types of the system components, such as processors, ASICs and busses, used to implement the system behavior. The task of partitioning maps the sub-behaviors in the specification to the system component. Scheduling determines the ordering of execution of the sub-behaviors on a sequential processor. The task of communication refinement selects the appropriate protocols and resources to implement the abstract communications between the sub-behaviors. It also generates protocol transducers used for interfacing components with different protocols.

Such a methodology with well defined design steps can go a long way in keeping the design modular and reducing debug/test time. Each design step generates a more refined model, preferably in the same language. This provides a standard documentation and improves communication between the designers. The well defined models and transformations also provide a good base for formal verification of the refined models. The intermediate models also make the design more manageable and maintainable for future upgrades, thus, encouraging IP reuse.

The different components in a system may communicate using different architecture implementations. We next describe the different communication models. Communication between two components would use one of these styles. It is, thus, important to describe the models so that abstract interfaces may be defined for them. Later, this information can be used to generate generic transducers for interfacing to these

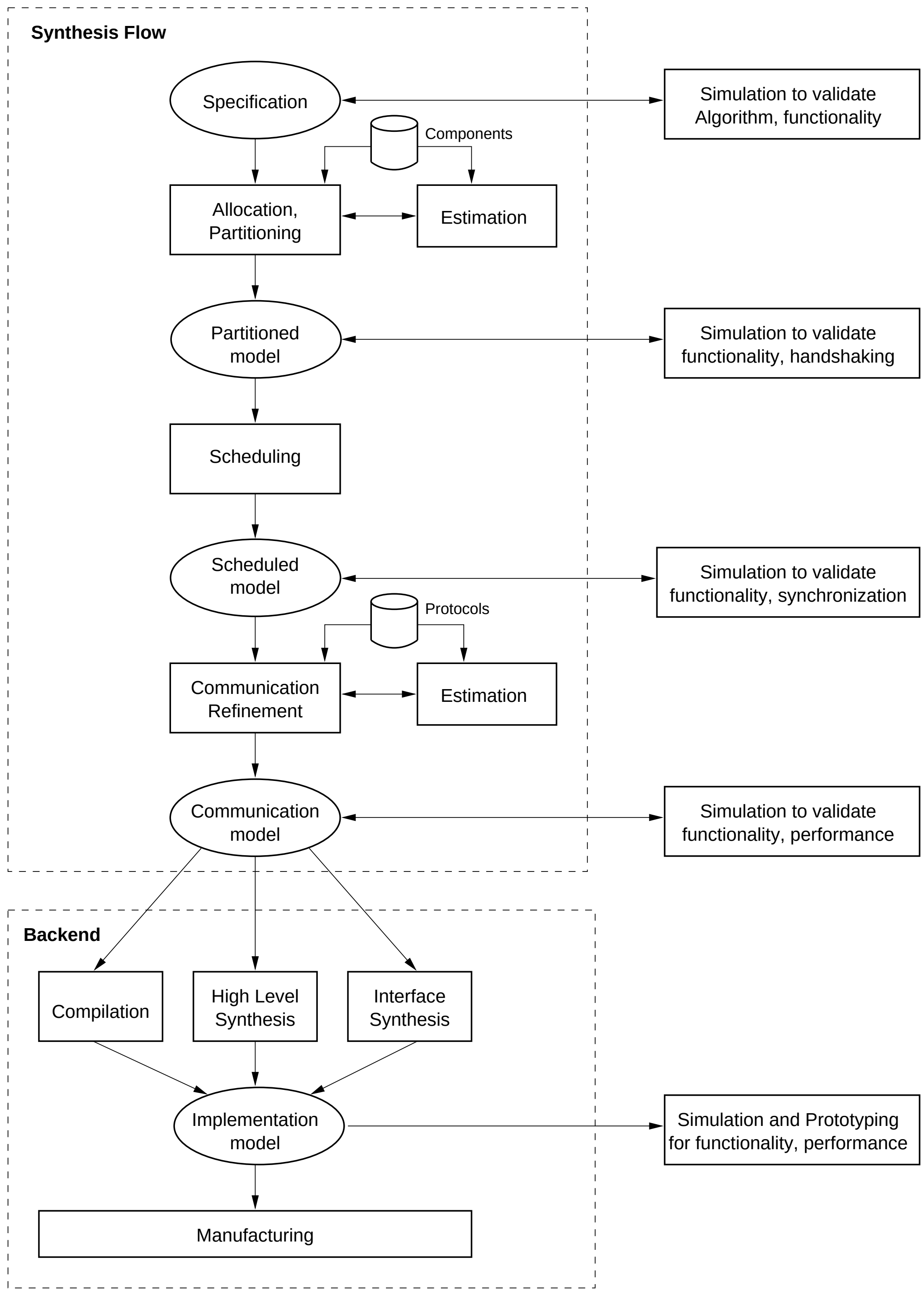

Figure 9: A generic co-design methodology

communication models.

Direct Connection using a Bus: In the simplest case, the two ASICs communicate using the same protocol. The components can then be di-

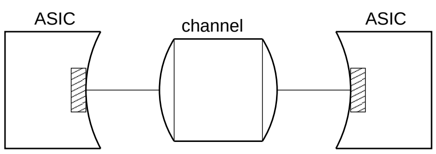

Figure 10: Direct connection of two ASICs

rectly connected to each other using a channel as shown in Figure 10. This channel encapsulates a communication bus like the VME or the PCI or an application specific bus.

Protocol Translation: If the ASICs have different protocols for communication, then the protocols needs to be translated. This protocol translation is done by a *transducer* as shown in Figure 11. The transducer is basically a finite state machine. It uses a few registers that may be required for protocol translation, e.g., one communication protocol uses a 16-bit wide data bus while the other uses only an 8-bit wide bus. The transducer then uses a register to first read the 16-bit word and then sends out one byte at a time. Another example might be that of a parallel to serial convertor using a shift register.

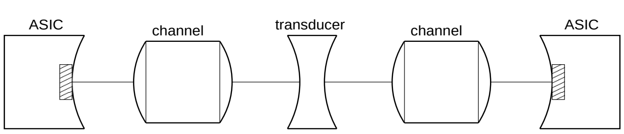

Figure 11: Simple protocol translation

Proper synchronization of the communication components must be ensured for correct results. The rate of data production and consumption may not be identical. If the system specification and implementation guarantee equal production and consumption rates then no translation effort is required. However, if this is not assured then handshaking must be used, e.g., if the producer generates data at a faster rate than used by the consumer, then the producer is restricted to generate the next set of data until it gets an `acknowledge` signal. This signal is given when the consumer reads the data. Similarly, a faster consumer will wait for a data `valid` signal from the producer that denotes prescence of a new data word. If handshaking signals cannot be used, the data will have to be buffered using other schemes as discussed below.

FIFO queue: The ASICs may produce and consume data in a burst mode. In such a case, storing the data in an intermediate buffer may be a better solution than operating the system at the speed of the slowest block. Data can be transferred between the functional blocks using a First-In First-Out (FIFO) queue as shown in Figure 12. There can be two types of interfaces: *blocking*

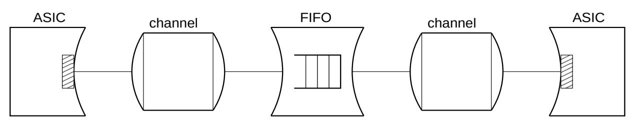

Figure 12: Communication using a FIFO queue

and *non-blocking*. The blocking interface uses `full` and `empty` lines for synchronization. Thus, the producer writes data into the buffer only if the `full` line is low. When the FIFO becomes full, the `full` line is asserted. Similarly, the consumer reads data from the buffer only when the `empty` line is low. This protocol then ensures that the slower component is not overwhelmed with data by the faster one and that no data is lost.

The non-blocking FIFO buffer does not use any extra signals to flag the current status of the buffer. Such an interface is used if the producer-consumer specifications ensure that they synchronize in such a manner that neither is overwhelmed by the other. If this is not possible, the FIFO will need to be large enough. In either case, however, data will be lost if too many burst requests occur.

Memory buffer: The inter-component communication can also be done using a random access memory in the interface as shown in Figure 13. An advantage of using memories over FIFO buffers is that they provide random access to the data stored. In the case of FIFOs the consumer must necessarily consume data in the same order as supplied by the producer. The data is removed from the FIFO queue as it is being read

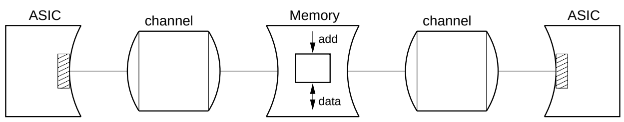

Figure 13: Communication using a memory buffer

by the consumer. Hence, if the consumer needs to use a data-set again, it must be stored internally. On the other hand, the data continues to reside in the memory even after being read so data may be read and written in different order.

The downside of using a memory is the added complexity of addressing. In addition, an arbiter may be required since both components access the memory. The arbiter needs to sequentialize requests that occur at the same time. Multi-ported memories may also be used to service more memory access requests at the same time. A memory is less restrictive than a FIFO because it can be used for bi-directional data transfer. A shared memory can also be used to exchange data amongst more than two ASICs and for multicasting data.

3 Modeling DCT as an example

We next describe modeling of Discrete Cosine Transform (DCT) as an illustrative example. Discrete Cosine Transform [2] is widely used in DSP applications for image compression in both still and motion picture standards. The DCT problem can be expressed as follows (details in Appendix A).

$$OutBlock = CosBlock \times InBlock \times CosBlock^T$$

where $InBlock$ is the input 8×8 block of pixels, f. $OutBlock$ is the output matrix in the frequency domain F and $CosBlock$ is defined above. The DCT can, thus, be modeled as two 8×8 matrix multiplications. These matrix multiplications (MM) can be serialized in time.

$$TempBlock = InBlock \times CosBlock^T \quad \text{(MM1)}$$

and

$$OutBlock = CosBlock \times TempBlock \quad \text{(MM2)}$$

The DCT transformation can then be modeled as two processes. The first process completes the first matrix multiplication and generates the 8×8 $TempBlock$ matrix. The results of this matrix multiplication is then used by the second process that generates the final output matrix, $OutBlock$. Both processes have an internal copy of the $CosBlock$ matrix.

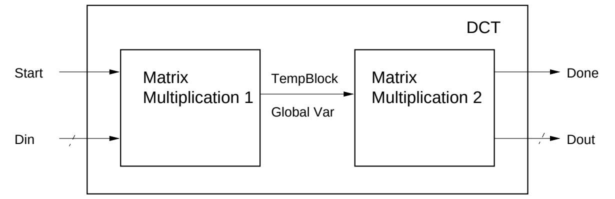

Figure 14: DCT modeled as two serialized matrix multiplications

The DCT can be modeled as two communicating processes as shown in Figure 14. The design of this ASIC, then, includes specifying the protocol of communication between the two processes and synthesizing the communication interface. The high-level *specification* of the problem is given in Appendix B. This high-level specification is partitioned into two processes each performing a single matrix multiplication. This model is shown in Figure 15. At this level of modeling, communication is done through *global variables*. The complete VHDL model is given in Appendix C.

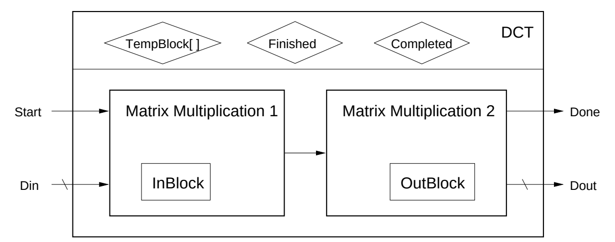

Figure 15: Global variables in the partitioned model

Process 1 reads the input data and stores the results of the first matrix multiplication in the global variable `TempBlock` and makes a flag `Finished` high to denote that it is done with its computation. This flag is a global variable and hence accessible to the second process also. The second process polls the `Finished` flag and when it sees that the first process is over with its computation, it goes ahead and does the second matrix multiplication, sets the `Completed` flag and finally puts the results on the output signal pins. Process 1 proceeds only after the second process has read the `TempBlock` matrix, i.e. it waits for the `Completed` flag to be set. The *InBlock* and the *OutBlock* matrices are stored in local memories of the first and second processes respectively.

The DCT specification is partitioned into two communicating processes. The first process generates the 8×8 `TempBlock` matrix. The 64 bytes of this matrix have to be transferred to the second process for the second matrix multiplication. There are various possible schemes for communicating this set of data. We next explore some of the different styles of model implementation and how components may be reused.

3.1 Protocol 1: Handshaking for each byte

We first look at direct communication between the two components without any intermediate storage elements. The two components can communicate using a dedicated bus with handshaking on every byte as shown in Figure 16. This protocol uses four control signals and separate address and data lines. The first process stores the result of the matrix multiplication, $TempBlock$, in local memory. When it is done with the computation, it sends a transmit request signal on TxReq. The second process responds with a TxAck signal. The 64 bytes of data are then transferred over the data bus with handshaking for each byte. The detailed protocol is described below. MM1 is matrix multiplication 1, MM2 is matrix multiplication 2, P1 is process 1 and P2 is process 2.

1. MM1 complete, P1 raises TxReq
2. P2 acknowledges when ready with a high on TxAck
3. P1 raises DReq and waits for ack
4. P2 sends DAck when ready to receive a byte
5. P1 places the address and data and pulls DReq low
6. P2 reads the address and data when DReq goes low
7. P2 pulls DAck low to signal completion of transfer of 1 byte
8. steps 3–7 repeated for a total of 64 times
9. P1 lowers TxReq
10. P2 lowers TxAck

The VHDL model for this protocol is given in Appendix D. This protocol dedicates 4 control signals for synchronization between the processes. The $TempBlock$ matrix is duplicated in the two processes. Even though memory is wasted due to duplication and extra time is required for transferring the entire matrix from one block to the other, this might be the only option in certain cases. If the producer and consumer produce and consume data at the same rate, then the consumer need not acknowledge receipt of data. Data may be strobed on every low→high transition of a Req signal or clock. However, if such assumptions about the relative rates cannot be made then the system must be operated at the speed of the slowest component by using request and acknowledge signals.

3.2 Protocol 2: Handshaking for every byte-pair

A dedicated bus may use protocols for communication that differ from the one described above. The various parameters in the protocol specification include the number of control wires, if the protocol is synchronous or asynchronous, the clock rate for synchronous protocols, type/width of data/address busses and the exact timing specification. Another protocol for communication over a dedicated bus is shown in Figure 17.

The first process stores the result of the matrix multiplication ($TempBlock$) in local memory. When it is done with the computation, it initiates data transfer with a request on the Req signal. The second process responds with an Ack signal. The 64 bytes of data are transferred over an 8-bit bus with handshake for every two bytes. The protocol can be specified as follows. MM1 is matrix multiplication 1, MM2 is matrix multiplication 2, P1 is process 1 and P2 is process 2.

1. MM1 complete in P1
2. P1 raises Req
3. P2 acknowledges when ready with a high on Ack
4. P1 places the address and pulls Req low at next clock
5. P2 reads the address when Req goes low
6. P1 places data byte on bus at next clock
7. P2 reads the data byte
8. P1 places the next data bye on bus at next clock

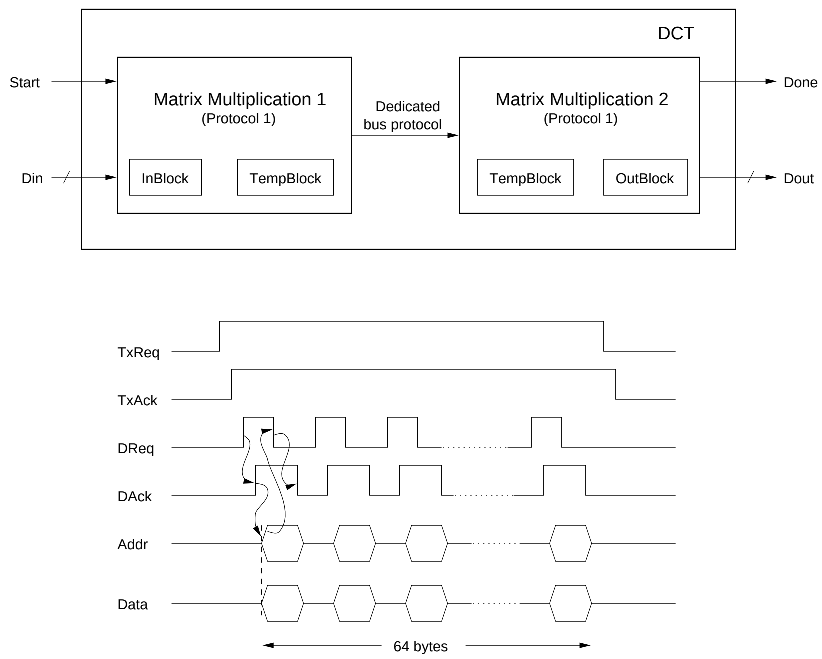

Figure 16: Bus with handshaking for each transmitted byte

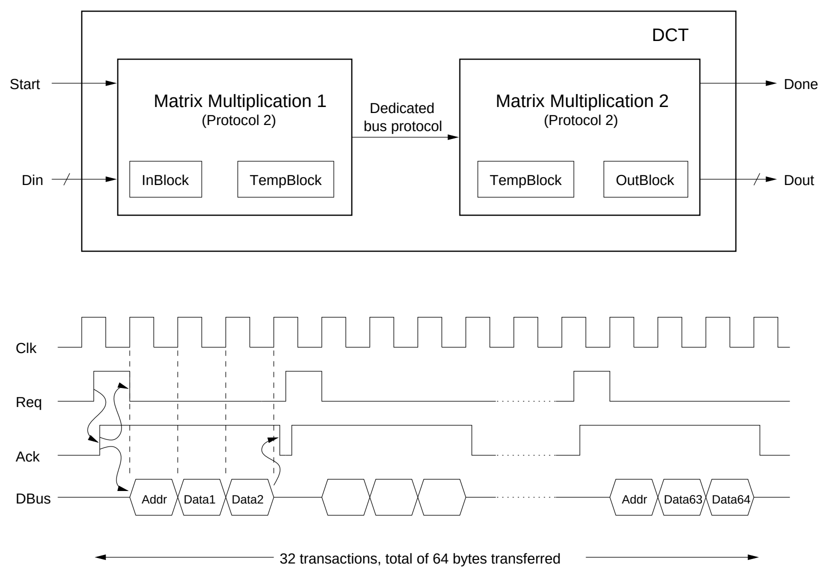

Figure 17: Bus with handshaking for every byte-pair

9. P2 read the data byte and pulls the Ack low
10. steps 2–9 repeated for a total of 32 times

The VHDL model for this protocol is given in Appendix E. This protocol multiplexes address and data on a single bus. The first process places only the even addresses on the bus. It sends out 2 data bytes for each address that it puts on the bus. The model uses only 2 control signals but the data transfer is synchronized with a common clock edge. The second process must count the number of bytes received since there is no transmission request signal. It interprets the first data byte to be for the address supplied and the second data byte for the next contiguous location.

3.3 Reusing an ASIC

The various components of a system may not have the same communication protocol. It may not be possible to synthesize the components to obey the protocol of the other component. Communication between components with different protocols for data transfer is a major hurdle in ASIC reuse. If protocols of processes that need to communicate with each other do not match then a proper interface must be generated that translates the protocols. We next describe experiments done on this problem. Consider the DCT system made up of two components that communicate using Protocol 1 (described in Section 3.1). Now, Process 2 which does the second matrix multiplication is removed and replaced by a component that uses Protocol 2 (described in Section 3.2). This is similar to the generic example shown in Figure 3.

Protocol 1 uses 4 control signals and a separate address and data bus. It is a completely asynchronous protocol. Protocol 2, however, uses only 2 control signals and a single bus. The address and data are multiplexed on this bus and two data bytes are transferred for each address put on the bus. After the data transfer cycle has been initiated using the request and acknowledge signals, the two data bytes are strobed on consecutive clock edges. A transducer is required between the two blocks since they do not use the same protocol. The DCT model composed of processes with different protocols is shown in Figure 18. The transducer (interface) needs to generate the proper control signals and do the protocol translation. The exact sequence of steps for the transducer can be listed as follows.

1. wait for the TxReq signal from process 1
2. send the TxAck signal
3. wait for a DReq signal from process 1
4. send the DAck signal
5. wait for DReq to go low
6. read the address and the first data byte from process 1
7. lower the DAck signal
8. wait for the next DReq signal
9. send the DAck signal
10. wait for DReq to go low
11. read the address and the second data byte from process 1
12. send the Req signal to process 2
13. wait for Ack signal from process 2
14. put the first address on bus at the rising edge of clock
15. put the first data byte at the next clock edge
16. put the second data byte at the next clock edge
17. wait until Ack from process 2 goes low
18. lower the DAck signal for process 1
19. repeat steps 3–18 while TxReq is high
20. lower TxAck signal to process 1

The interface, thus, first gets the address and data from the first process according to its protocol. It stores them internally and does not give the final acknowledge signal to the first process. It returns this acknowledgment only when it completes the data transfer to the second process. This way, it needs limited buffering. Other implementations of the interface are also possible. The interface can be modeled as an FSMD as shown in Figure 19. The actions are associated with the edges which represent the state transitions. The complete VHDL code is given in Appendix F.

3.4 Replacing the Bus

In the previous section, we discussed replacement of a component with another that has the same functionality but different communication protocol. The communication bus itself may be changed with another that has a different protocol. Consider the DCT

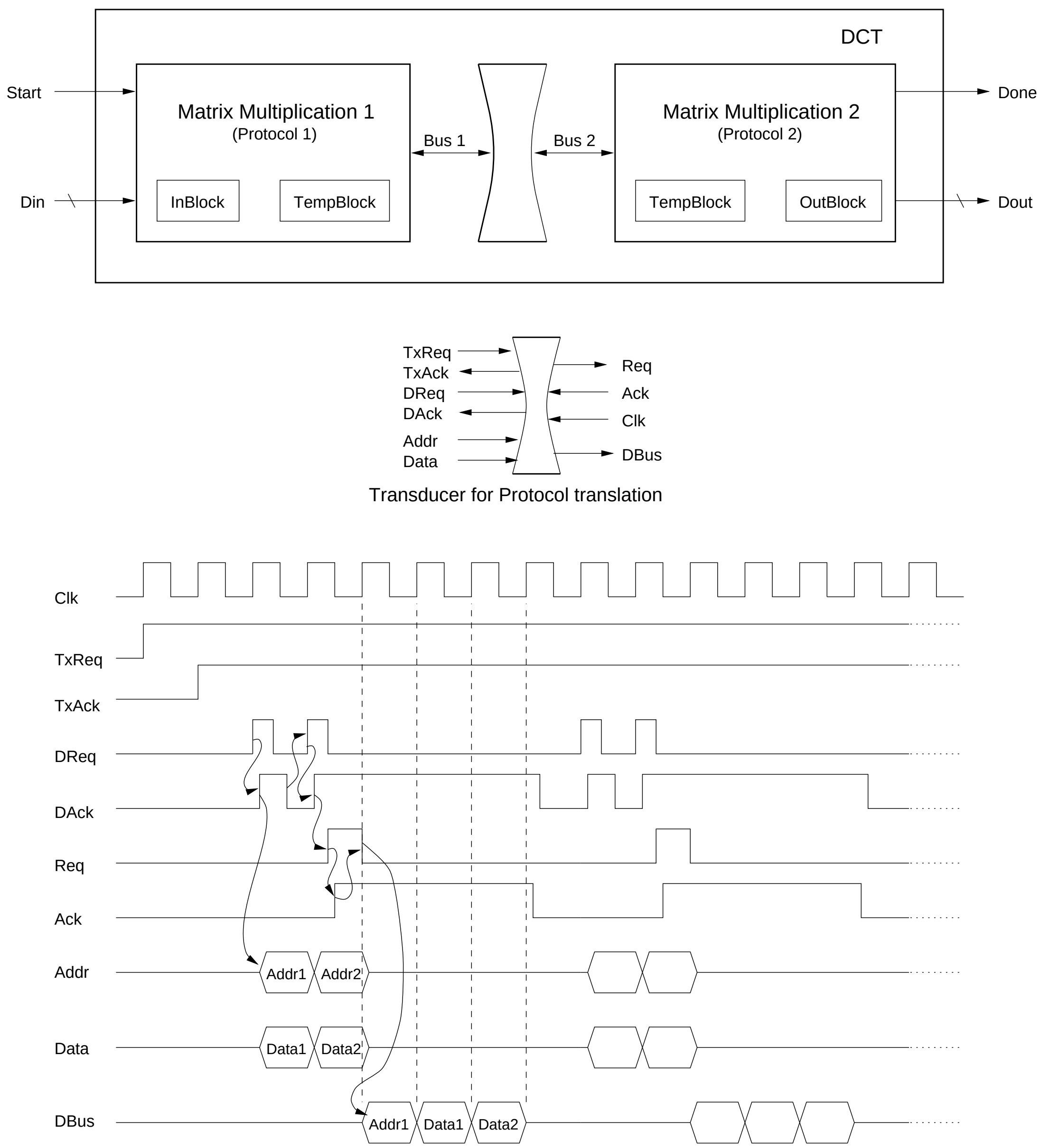

Figure 18: Replacing a component with different protocol

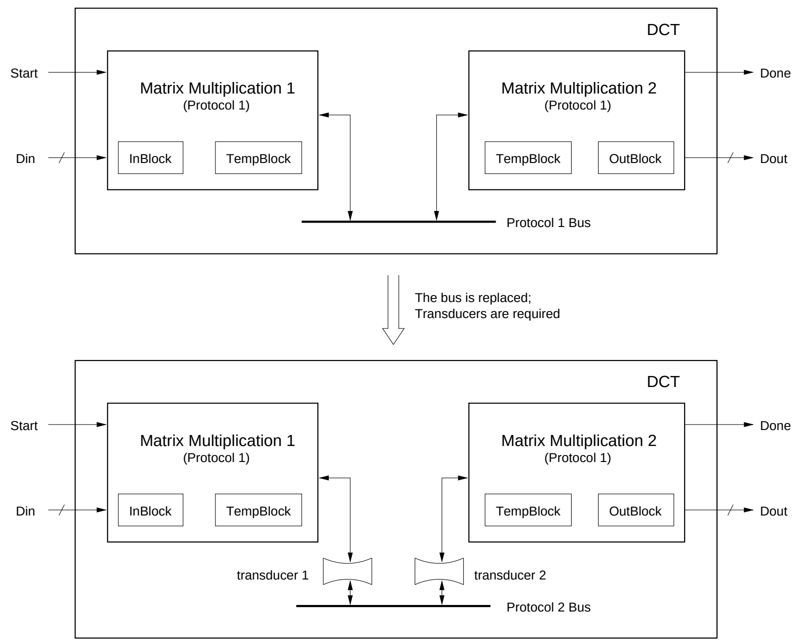

Figure 20: Replacing Protocol 1 bus with Protocol 2 bus in DCT system

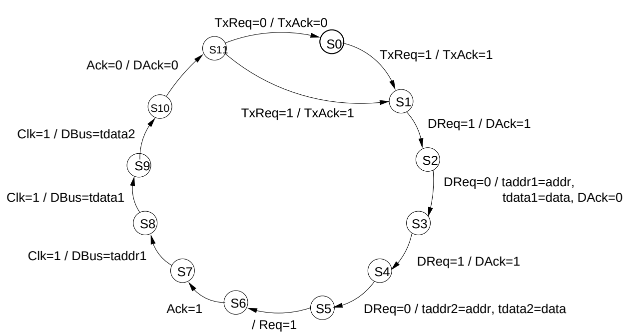

Figure 19: FSMD for protocol translation

example where both components use Protocol 1 as described in Section 3.1. The bus used for communication between them is also a Protocol 1 bus. Hence, transducers are not required for connecting the components to the bus as shown in Figure 20. The protocol 1 bus which uses 4 control signals and handshakes on every byte is then replaced by the protocol 2 bus that has only 2 control signals and transfers two data bytes for every handshaking transaction. Two new transducers are needed to translate the protocols and connect the components to the new bus as shown in Figure 20. The complete VHDL model is given in Appendix G.

3.5 Communication using a FIFO

In the previous protocols the *TempBlock* matrix was duplicated. Both the matrix multiplication units had a copy of the matrix. The first process, MM1, stored the computed results in a local memory and then transferred the matrix to the second process, MM2, which first saved a copy in local memory and then proceeded with the computation. However, the memory duplication overhead can be avoided by sharing the storage between the two components.

For this example and similar problems, bounded First-in First-out (FIFO) queues can be used as means of communication. FIFOs also avoid the data duplication problem. The advantage of using a FIFO is that the consumer and producer can operate concurrently. It is especially useful when the producer and consumer data at approximately the same rate. The producer and consumer do not require extra bits for addressing. The arbiter which is required for shared memory access (discussed in Section 3.6) is also not required. FIFOs, however, do not provide the random access capability that memories provide. The consumer must read the data in the order in which the producer supplies it. Thus, if the producer does not need to reuse the data it produces, it can immediately transfer it to the queue without storing it in local memory and then

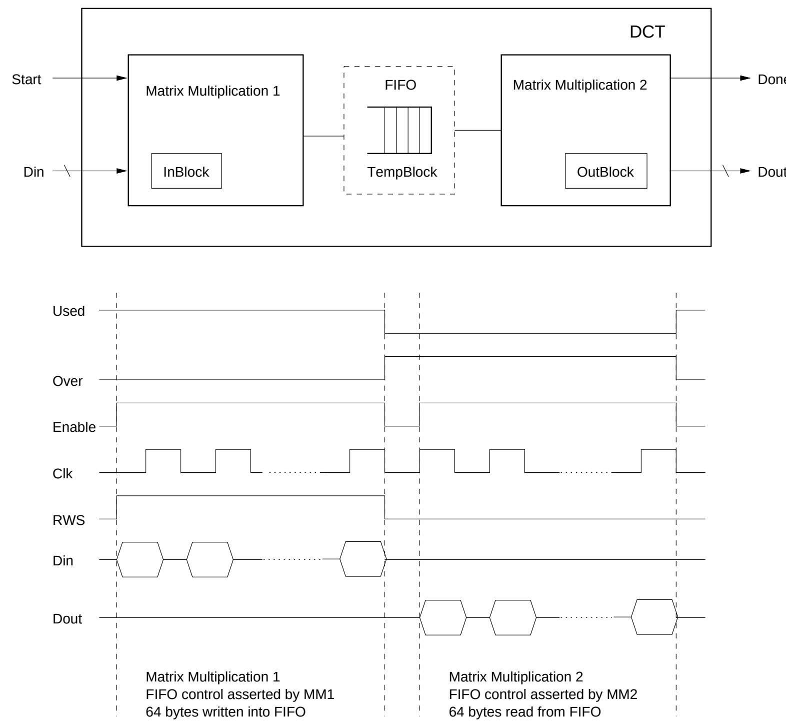

Figure 21: Communication using a FIFO queue

at a later stage reading from there and transferring it. The consumer can read the data from the queue in the first-in first-out discipline.

The protocol for communication using a FIFO is shown in Figure 21. The complete VHDL code is given in Appendix H. The operation is similar to the shared memory protocol. MM1 computes the first matrix multiplication and stores the 64 bytes of data in the FIFO. After the computation is over, it sends an `Over` signal to the other process. MM2 then reads the 64 bytes of data from the FIFO and does the second matrix multiplication. When it is done with the computation, it sends a `Used` signal to MM1 which denotes that MM1 can proceed ahead and do another matrix multiplication.

FIFOs are an efficient way of communicating between two processes. However, reading the data removes it from the queue. Thus, if the consumer needs to access some data repeatedly, it must store it inside. This is not the case for random access memories. The second matrix multiply unit, MM2, needs to use a $TempBlock$ byte repeatedly for the matrix multiplication. It must either buffer the bytes internally or use another matrix multiplication algorithm. This algorithm will read a value from the FIFO (an element of the $TempBlock$ matrix), performs all the multiplications which require this value and then discards this value. This approach has been adopted in our case. If the size of FIFO is the same as the amount of data to be transferred, then dedicated control lines between the two matrix multiply units are not necessary. The *full* and *empty* lines from the FIFO can be used to synchronize the two processes. If dedicated lines for synchronization are used in conjunction with the *full* and *empty* lines both the blocks may operate concurrently.

3.6 Communication using a Memory

The duplication of data and memory in the two components can also be avoided by sharing a memory between the two components. This achieves the same goal as the FIFO in the previous section. In the general case, random access memory and FIFO provide different benefits and have their own limitations.

The protocol for communication using a random access memory with implicit arbitration is demonstrated in Figure 22. The two units do not have local mem-

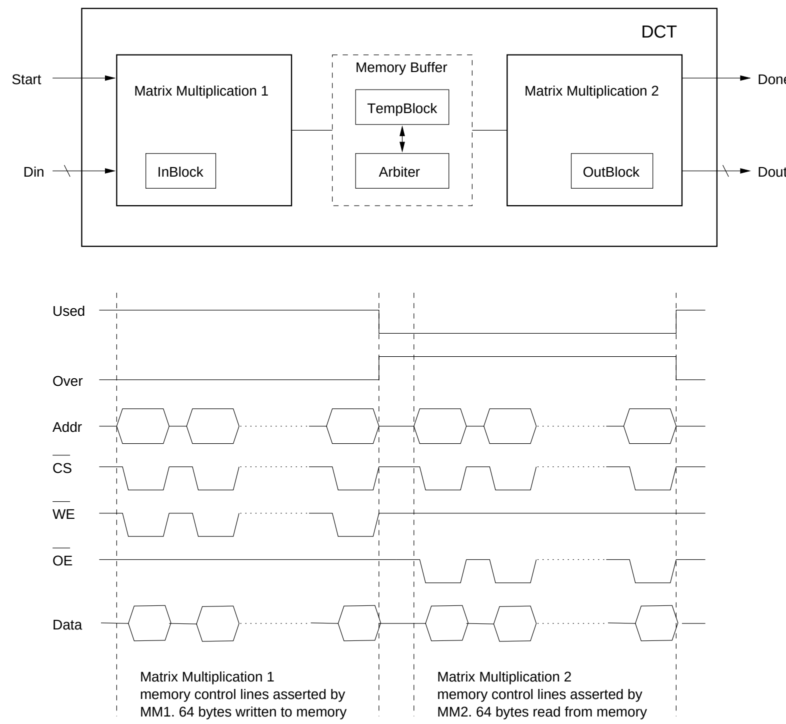

Figure 22: Communication through random access memory

ory for the *TempBlock* matrix. They use a random access memory for storing the matrix. This memory is shared between the two components. During the first half of the computation, the first matrix multiply unit writes into the memory. During this time the address, data and control lines of the memory are asserted by the first matrix multiply unit. MM1 gives the address of the data byte being written and asserts the $\overline{CS}$ and $\overline{WE}$ control lines according to the timing diagram of the memory. MM1 send a computation complete signal on the over line to MM2 and relinquishes control of the address and data busses. In the next phase, the second multiply unit, MM2, controls the random access memory after receiving the over signal from MM1. MM2 supplies the address and asserts the $\overline{CS}$ and $\overline{OE}$ lines to read the matrix values for the computation. When the computation is over, MM2 relinquishes control of the address bus and floats the CS and OE lines. It then sends a used signal to MM1 which denotes that the data from the random access memory has been read and that MM1 can start the next computation.

A NEC 1 M-bit (128K by 8-bit) CMOS static RAM [5] (part number $\mu PD431008L$) has been used for modeling the random access memory between the two processes. It has a read-write cycle time of 17 ns. The complete VHDL code is given in Appendix I.

In this case, the arbiter is very simple and its job is to turn around the data busses. The two components do not try to access the memory buffer at the same time. They communicate between themselves using dedicated control lines to make sure that only one unit accesses the memory at a time. This is possible since the matrix multiply processes do not have any interleaved memory accesses. MM2 reads the memory only after MM1 has written all the data into the memory. However, a priority resolution method and recovery method in case of clashes will need to be determined for the general case of accesses to a random access memory. In this example, however, there is no need for the arbiter to acknowledge successful grant of the busses since it is guaranteed that the processes synchronize between themselves and the arbiter does not need to provide this functionality.

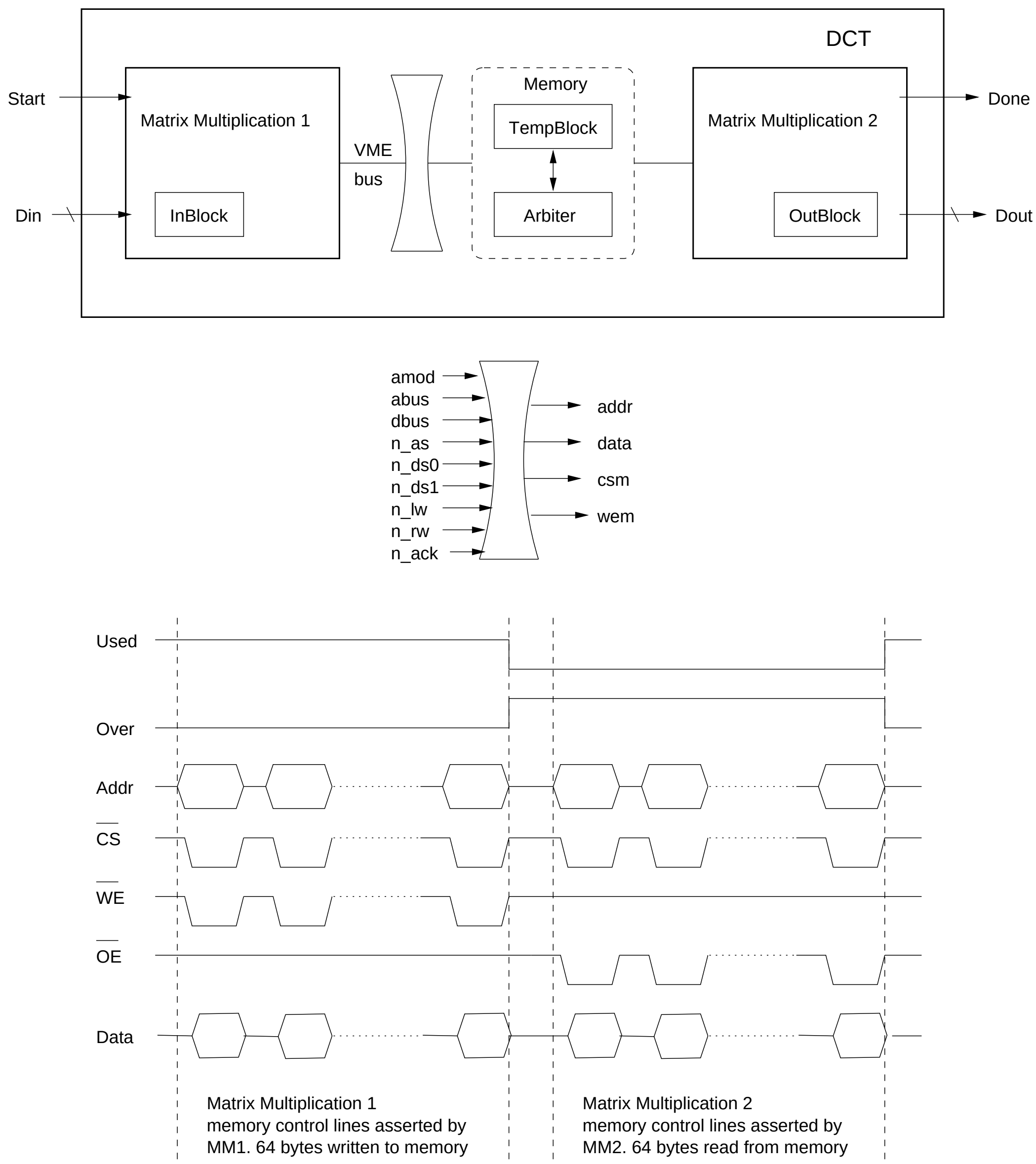

Figure 23: Communication using a Memory and VMEbus

3.7 Communication using a Memory buffer and VMEbus

In the previous sections, we have looked at designs that all use dedicated application specific resources. In particular, the communication busses have been all configured to best suit the example in terms of number of control signals and bus widths. However, this kind of facility may not always be available. One or more of the components might have been defined for some particular protocol. There may not be the flexibility of re-designing the entire component for the particular communication scheme. Otherwise, system considerations might force the designer to choose some standard bus for communication even if the components are being synthesized. This section discusses use of a VMEbus [6] as means of communication between a component and the memory.

As shown is Figure 23, the component MM1 is a VME Master component and communicates in the VMEbus protocol. The component writes the *TempBlock* matrix data bytes in the VMEbus protocol. The shared memory, however, is not a VME slave component. Hence, an interface is used between the VMEbus and the SRAM. The interface unit in this case is minimal since the VME master only does single byte transfers on the lower 8 bits of the data bus. There are no double, quad byte or burst mode transfers. It generates the chip select signal from the address strobe signal of the bus and sends an acknowledgment based on the timing characteristics of the SRAM.

Industry standard busses are commonly used for communication between components. If the components cannot be synthesized for the particular bus protocol then transducers (interfaces) must be created for protocol translation. The interface can be quite complicated with internal buffering along with a state machine for generating the proper control signals.

Experimentation with the DCT example has demonstrated the issues in modeling for reuse. We explored different communication schemes between the two components of DCT. We were able to look at the IP reuse problem by changing an ASIC with another that uses a different communication protocol. We also looked at the effect of changing the bus on the system models. These experiments lead us to propose some guidelines for modeling in VHDL which we present next.

4 Modeling for Reuse

In this section, we discuss some of the essential issues in modeling for reuse. A system consists of various components that communicate amongst themselves for exchanging data and synchronizing computation. In a hardware system, a component refers to any of the processors, ASICs, FIFOs, random access memories that comprise the system. Thus, it is important to ensure that there is no protocol mismatch amongst the various components. Some of the important issues regarding modeling and implementation of a system are as follows.

- Whenever there is a protocol mismatch between two components or between a component and a bus, a *transducer* must be used for protocol translation. If the ASIC is a synthesizable component then the transducer is not a separate entity but can be integrated with the ASIC during the synthesis process. Otherwise, the transducer will exist as a separate component in the system.
- Whenever a component (ASIC, memory, FIFO) is replaced with another that has a different communication protocol, the existing transducer has to be redesigned. If there was no transducer initially, then a new transducer must be inserted.
- The components might communicate using a common bus. Every time a bus is replaced, all the transducers that are used to connect ASICs to this particular bus will need to be redesigned.

Reuse is encouraged if the ASIC does not have the detailed communication protocol in its description. It is extremely difficult to modify a system model if communication is interleaved with computation. Changing the communication protocols then amounts to rewriting the entire model since the communication steps are distributed all over the model. In addition, it is hard to determine what constitutes internal computation and what constitutes external communication. Thus, reuse is encouraged if communication is separated from computation and is moved outside the component model. If this is done then the port interface of the component model consists of function calls instead of bit signals. This is helpful because the components are generic and not restricted to a particular protocol. They can be stored in a library and generic interfaces also facilitate automatic generation of transducers which is rather hard to do if the interfaces specify timing on bit signals.

4.1 Using VHDL as a modeling language

The above mentioned guidelines though highly recommended cannot be put into practice easily when VHDL is used as a modeling language. VHDL is inherently *flat* which means layered protocol specification is not possible. VHDL requires that the `entity` declaration of models be at the bit-level and hence, communication must necessarily be modeled at the physical level, i.e., transitions on signals [7]. But if the communication has to be removed from the component model, then the `entity` interface to the external world will consist of function calls. However, in VHDL an `entity` cannot consist of function calls. It must use the standard data types like `bit, integer` or `std_logic`. The model then also defines the timing specification on these signals. Thus, the interface is *too* detailed and cannot be used for incremental refinement and reuse at the behavioral model.

However, communication can still be separated from computation by using a separate `process` inside the behavioral VHDL `architecture` of the ASIC for just communication. This process encapsulates the entire communication protocol and the computation is done in other processes of the same architecture. The computation processes do not access the `ports` of the `entity`. They request the `communication process` for all read/write requests. This is done using global variables which are essentially `signal` declarations inside the `architecture` of the model. A typical model looks like the one shown in Figure 24.

port interface
of the ASIC

```
entity ASIC is
  port ( clk   : in bit;
         data : in integer);
end ASIC;

architecture behav of ASIC is
    signal sync, ready : bit;
    signal local : integer;
begin
```

compute process(es)
perform only
computations.
No communication

```
    compute : process
        variable a : integer;
    begin
        ....
        sync <= '1';
        wait until ready='1';
        a <= local;
        sync <='0';
        ....
    end process compute ;
```

communicate
process does all
the communication
with the external
world

```
    communicate : process
    begin
        ....
        wait until sync='1';
        local <= data;
        ready <='1';
        ....
    end process communicate;
end behav;
```

Figure 24: Separate communication and computation `processes` in VHDL

Now consider replacing an ASIC model with another that uses a different communication protocol. Typically, the entire model for this new ASIC will need to be rewritten to reflect the different protocol. However, now the communication is clearly demarcated and separated from all the computation since it is in a separate process. Thus, only the `communication process` needs to be redesigned and the computation processes do not need to be modified. This modeling technique is illustrated in Appendix D, E.

When an ASIC is replaced with a new ASIC, the VHDL models need to be modified if the communication protocol of the new ASIC is different from the earlier protocol. The `communication process` of either of the two ASICs between which there is a protocol mismatch needs to be modified. The changes must reflect the protocol of the other ASIC. This might also necessitate modifications in the `port` declarations of the ASIC `entity`. However, if none of the ASIC is synthesizable then the models cannot be changed and a transducer needs to be inserted. This transducer is a separate `entity` which interfaces to the two ASICs. The `architecture` of the transducer has a model for the finite state machine that does the protocol translation as described for the DCT example in Figure 19. When a bus is replaced, then the same procedure needs to be applied for each of the components connected to the bus. If they are not synthesizable then transducers need to be developed and inserted in the top level `entity` that encapsulates all the different components.

The inability to model at a higher level in VHDL leads us to propose the use of the new language, SpecC. SpecC is an executable hardware modeling language suited for co-design and component reuse. In the next section, we discuss the benefits of SpecC and how it supports ASIC reuse.

5 Modeling in SpecC

There is a modeling need to separate the computation from communication, as discussed in Section 2.1. The ports of entities can then be made abstract in terms of number and width of signals. A port of a particular type will then be specified by the *function calls* it will support, e.g., `read_word()` and `write_word()`. This port will be resolved to a real channel that has the actual details at the later time of component instantiation. The SpecC language supports these features and makes modeling for reuse a natural and easy task. It is able to separate the computation specification in a component from the communication protocols by using the concept of channels and the software modeling techniques of data abstraction and information hiding.

5.1 Modeling Refinement in SpecC

A key feature of the SpecC language is that the designer can model the system at a behavioral level and the models can then be refined for system implementation. We next present how modeling refinement is done in the SpecC methodology.

5.1.1 Global Variables

At the highest level, the models are composed of system behaviors that communicate using *global variables*. This kind of modeling is closer to the conceptual model of the system. There is no notion of ports of sub-behaviors or protocol transfers. It is the simplest model that captures the algorithm being used for solving the problem. The designer is not burdened with communication protocols and target architecture at this stage and uses such a model for verification of the algorithms and functionality of the system under design.

A typical system model at this level is shown in Figure 25. The system behaviors see the communication variables as any other local variable since they are declared as global in the specification.

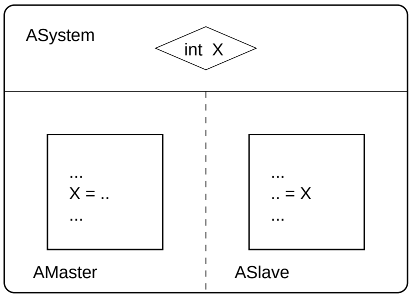

Figure 25: Global variables in a SpecC model

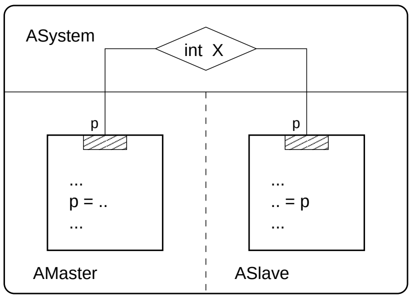

Figure 26: Global variables through ports in a SpecC model

5.1.2 Global variables through ports

The behaviors may also access the global variables through their ports. This model is shown in Figure 26. This model is in essence the same as the earlier one that uses only global variables. The only difference is that in this model, the behaviors explicitly declare the global variables as ports. They access the global variables only through the formal port variable names.

The high level behavioral specification of the system can be composed using either of the two models. They are both at a high level and do not include any notion of communication protocol and channels. This model with ports can be generated automatically from the previous model in a pre-processing step. It is useful to have this model as it clearly defines the interface of a behavior and what variables are internal and what are accessed from outside the system behavior.

5.1.3 Variable inside a channel

The global variable model is refined automatically to a model that includes channels. Each global variable is encapsulated in a channel and the channel defines interfaces for accessing this variable. The behaviors use function calls that are part of the port interface to read and write these variables, as shown in Figure 27.

There is no timing information in either the behavior or the channel. Transfers take place by simply

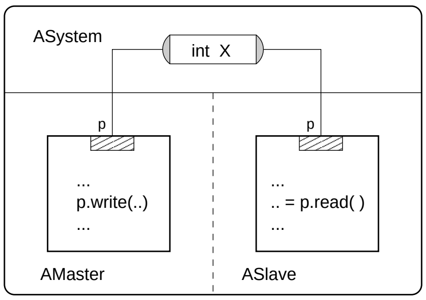

Figure 27: *One-shot transfer* using channels in a SpecC model

reading or writing into the variable inside the channel using the function calls of the interface. This model is used for performing the *allocation, partitioning* and *scheduling* operations required for system implementation as shown in the co-design methodology of Figure 9.

5.1.4 Protocol inside a channel

Allocation and partitioning determine how the system behaviors are mapped and what is the connectivity between components. Channels between behaviors that get mapped to the same component become local variables. The channels then get discarded. The channels going across behaviors on different components are mapped to the common bus that connects the components. A protocol is selected for this bus. The bus is encapsulated in a channel that includes the detailed protocol for the bus. The system behaviors still use function calls for communication as shown in Figure 28.

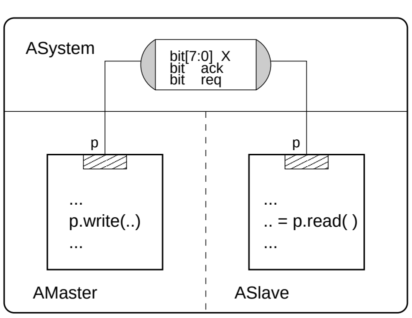

Figure 28: *Protocol transfer* using channels in a SpecC model

The channel has the different bit level signals for shared variable access like `req`, `ack` and defines the bit-widths and timing of data transfer. These channels encapsulate a particular bus protocol and can be stored in a library. Both the ASICs and the busses can be replaced by new components with different protocols. Reuse is performed at this level and behavioral transducers may have to be introduced.

5.1.5 Inlined channel

After the different options for the components have been explored on the channel model, communication synthesis is done. This process inlines the channel functions to synthesizable ASICs or along with the wrappers as separate components. The refined model again consists of ports that are connected to global variables, as shown in Figure 29.

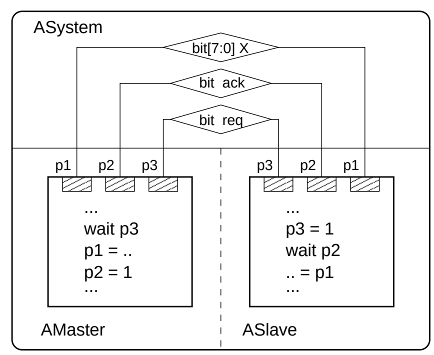

Figure 29: Inlined channel in a SpecC model

This model is, thus, similar to the *global variable through ports* model. However, the important difference is that the model is refined and the ports are now bit signals instead of abstract data types like `int`, `float` etc. The behaviors, too, have the detailed communication protocol that was introduced in them as a result of the inlining process. This model basically consists of a component netlist and can be synthesized using traditional back-end tools.

We next describe some examples in SpecC [4] and how models can be written with desirable features for reuse. We first describe a simple shared variable model and then describe communication over a synchronous bus.

5.2 Shared Memory Channel

A shared memory channel which can be accessed by concurrent processes is shown in Figure 30. For the

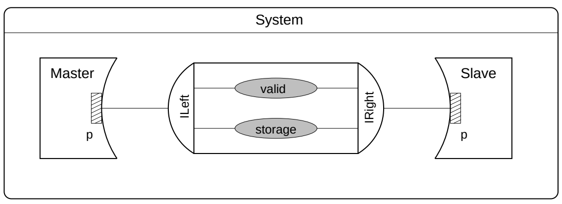

Figure 30: A shared memory channel

sake of simplicity, consider the case when only one of the processes writes into the shared variable and the other reads from it. A typical system using such a shared memory system can be modeled in SpecC as follows (SpecC keywords are in boldface).

```
interface ILeft (void) {
    void write (int val);
};

interface IRight (void) {
    int read (void);
};

channel CShared (void) implements ILeft, IRight {
    int storage;
    bool valid;

    void write (int val) {
        storage = val;
        valid = 1;
    }

    int read (void) {
        while (!valid) ;
        return storage;
    }
};

behavior Master (ILeft p) {
    int local;

    void main (void) {
        ....
        p.write(local);
        ....
    }
};

behavior Slave (IRight p) {
    int local;

    void main (void) {
        ....
        local = p.read();
        ....
    }
};

behavior System () {
    CShared var;

    Master X (var);
    Slave  Y (var);

    void main (void) {
        par {
            X.main();
            Y.main();
        }
    }
};
```

The actual variable, `storage`, is encapsulated by the channel CShared which also has the synchronization `valid` bit. The write operation stores the value and sets the valid bit and the read operation spin-waits on the valid bit. This simple example brings out the concept of information hiding and how computation is distinguished from the communication. The processes (behaviors) themselves do only the computation and just make function calls to `write()` and `read()` operations. The details of the shared variable access are described in the functions encapsulated in the channel.

The behavior entities have ports in the form of *interfaces*. A channel relates to interfaces by the *implements* keyword by which it is guaranteed that the channel will implement the methods declared in the interfaces that it *implements*. Thus, any other channel that implements the same interfaces `ILeft` and `IRight` can be used in place of the `CShared` channel. This kind of modeling is obviously conducive to reuse of components.

5.3 Synchronous Bus Channel

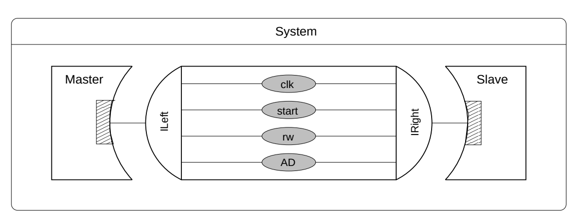

Figure 31: A synchronous bus channel

As a more complex example than shared memory channel consider a synchronous bus channel as shown in Figure 31. This example models read and write of memory over a bus using a simple synchronous bus protocol which is detailed in Figure 32.

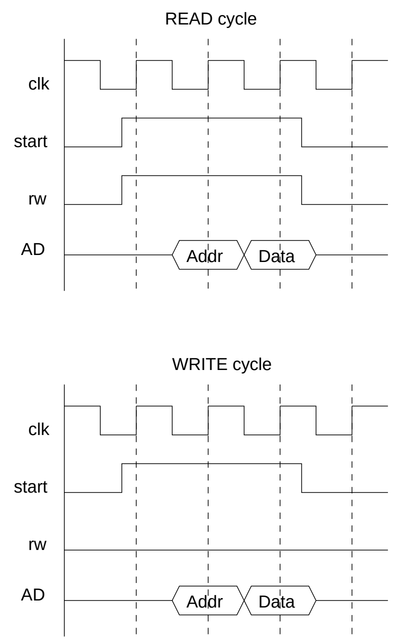

Figure 32: A simple bus protocol

Such a system can be modeled in SpecC as follows (SpecC keywords are in boldface).

```
interface ILeft (void) {
    void read (word addr, word *d);
    void write (word addr, word d);
};

interface IRight (void) {
    void monitor(
        void (*grab)(word addr, word *d),
        void (*deliver)(word addr, word d));
};

channel CBus (void) implements Ileft, IRight {
    clock          clk;
    signal<bit>    start;
    signal<bit>    rw;
    signal<word>   AD;

    void read (word addr, word *d) {
        start=1, rw=1,     clk.tick();
        AD=addr,           clk.tick();
        *d=AD, start=0,    clk.tick();
    }

    void write (word addr, word *d) {
        start=1, rw=1,     clk.tick();
        AD=addr,           clk.tick();
        AD=d, start=0,     clk.tick();
    }

    void monitor (
      void (*grab)(word addr, word *d),
      void (*deliver)(word addr, word *d)) {
        word a, d;

        while (start==0)  clk.tick();
        if (rw==1) {
            clk.tick();
            a = AD,                    clk.tick();
            (*grab)(a, &d), AD=d,      clk.tick();
         } else {
            clk.tick();
            a = AD,                    clk.tick();
            d=AD, (*deliver)(a, d),    clk.tick();
        }
    }
};

behavior Master (Ileft bus) {
    word local;

    void main (void) {
        ...
        bus.read(0x10, &local);
        ...
        local++;
        bus.write(0x10, local);
        ...
    }
};

behavior Slave (IRight bus) {
    word storage[0x100];

    void my_grab (word addr, word *d) {
        *d = storage(addr);
    }
    void my_deliver (word addr, word *d) {
        storage(addr) = d;
    }

    void main (void) {
        for (;;) {
            bus.monitor(my_grab, my_deliver);
        }
    }
};

behavior System (void) {
    CBus     bus;
    Master   master(bus);
    Slave    slave(bus);

    void main (void) {
        par {
            master.main();
            slave.main();
        }
    }
};
```

This example again brings out the distinction between computation and communication achieved in SpecC models. The channel `CBus` encapsulates the wires in the simple bus, viz., `clk, start, rw` and `AD`. The interface ports of the components (behaviors) consist of only abstract methods for communication. These methods just specify the behavior that is used by the components. The actual procedure of these methods is moved to the methods in the channel. Thus, the component only needs to make a functional call to methods like `read()` and `write()`. The detailed timing of `clk, start` and `rw` signals as specified by the protocol is moved inside the channel.

The ports of the behaviors can be mapped to the channel when they are instantiated. In this way, *late binding* can be achieved. Thus, the behavior need not know what channel it will eventually be mapped to and hence the communication details are removed from the functional specification of the component. This also permits *reuse* because any other channel can be used. The channel can use not only a different timing specification but even with different number of wires. The only requirement is that it should implement the interfaces of the ports of the components.

5.4 Abstraction Levels

The co-design methodology given in Figure 9 uses models at different levels of abstraction. At the highest level, there is a *behavioral specification* of the system in the SpecC language. Allocation and partitioning on this model lead to the *partitioning model*. Scheduling is performed on this model which generates the *scheduling model*. Finally, communication refinement leads to the *communication model*. These abstraction levels are helpful for co-design since the design can be debugged, validated and refined incrementally. Each abstraction level refines the level that is higher in the hierarchy. The different levels are essential for reuse since the final implementation model is too detailed and cannot be easily reused. The rigorous synthesis flow provides models with documentation at higher levels of abstraction which can then be reused.

However, the models at the different level of abstraction have different accuracy. The performance metrics of these models may be quite far from that of the final implementation model. The initial specification is only functionally accurate. The SpecC model can be simulated and validated for correctness of results. There is no implementation detail in this model. At the next level, the partitioning model includes the software and hardware components of the system. The different behaviors, however, execute concurrently on the processing element on which they are allocated. The performance accuracy at this abstraction level depends on the accuracy of the estimation tools. The performance of the hardware component can be computed approximately but there are no interface models for communication between the components. It is also difficult to get the complete performance estimates for software too because all the software behaviors execute concurrently, contrary to the actual final implementation. However, good estimates for each behavior may be computed separately.

The partitioning model is scheduled which serializes the behaviors on the processing elements. Thus, the software too can be estimated quite accurately. However, the models at this level still do not include the communication details. The communication refinement step introduces protocol transducers, communication primitives and inlines the channel methods. The communication model is, thus, complete in terms of specification of the final implementation. The performance is still not accurate because there is scope for optimizations during high level synthesis and compilation.

The software component of the communication model is compiled and the hardware component is synthesized using HLS tools and techniques. Cycle-based simulators and instruction set simulators can be used to validate the design and estimate the performance. The design may also be prototyped using FPGAs. The implementation model is then quite close to the final manufactured component and the performance of the prototype with appropriate estimators may be accurate enough for most purposes.

We note that even though the performance accuracy of models at higher abstraction levels is not high, the hierarchy is still acceptable. This is because the fidelity [8] of the estimators is usually high. Thus, experience with the estimator tools can be used to get better performance accuracy than suggested by preliminary estimates.

5.5 Proposed "Co-design Explorer"

We propose to develop and implement a co-design system, the *"Co-design Explorer"* [9]. This system will be based on the new SpecC language [3]. SpecC is able to model mixed abstraction levels as discussed earlier. It can also capture the characteristic features of embedded systems [8], such as concurrency, state transitions, structural and behavioral hierarchy, exception handling, timing, communication and synchronization.

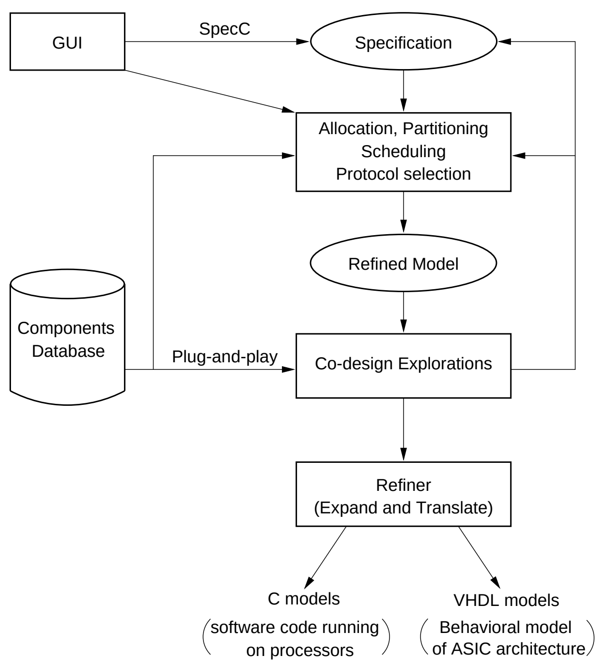

Figure 33: Proposed *Co-design Explorer* System

This system will be based on the co-design methodology presented in Figure 9 and use the *channel* concept illustrated in Figure 5, 7, 8. An overview of the system is given in Figure 33. A graphical user interface (GUI) is used to generate the specification of the system without learning SpecC. The GUI helps in specifying hierarchical and concurrent behaviors. It can be used for specifying the state transition functions and connectivity of behaviors. The GUI generates a specification of the system in SpecC. This model uses global variables for communication as shown in Figure 25. It is internally pre-processed to convert global variables into channels. This model is then be used for manual allocation, partitioning and scheduling with assistance from estimator tools. *Plug-and-play* is performed on the refined model using components from a database. The database also stores generic transducers and busses encapsulated in channels. Explorations are done to meet the performance and cost requirements.

Finally, a *Refiner* tool is used to expand and translate the model generated after explorations. This step includes channel inlining (as discussed in Section 2.1), generation of C code that will run on processors and generation of VHDL behavioral models for the hardware architecture. These low level models can then be compiled and synthesized using traditional techniques to get the implementation model. The advantage of using this methodology is that it uses SpecC which provides a minimal and complete set of constructs required to model embedded systems at various abstraction levels. Furthermore, the designer does not have to know the intricacies of hardware modeling as the graphical user interface can be used to specify hierarchy, concurrency, timing and state transitions.

6 Conclusion

In this report, we have outlined a generic co-design methodology that supports component reuse. We have talked about the various steps in the methodology and discussed the step of communication synthesis in detail. We stress that computation needs to be separated from communication. This distinction needs to be made to generate reusable components. This is because then only the communication part needs to be re-designed. Separation can be achieved by declaring the ports of the component at levels higher than bit signals. The protocol can be abstracted away into the channels used for connecting components.

We have also described different communication models and done explorations using different communication protocols between two components that as a system compute the Discrete Cosine Transform. It is important that all communicating processes use the same protocol. If a component uses a different protocol then a transducer must be used between the components. This transducer will do the protocol translation. The transducer will, in general, be complicated and may have internal buffering of data and ensure correct timing of signals.

Using this exploratory example, we have come up with a set of guidelines for modeling in VHDL. Our critique of VHDL led us to propose the use of the new co-design language, SpecC. We have described some examples from SpecC which support its use for ASIC reuse and incremental refinement. We need to do more research to solve the open problems related to ASIC reuse and use of SpecC. As future work, we need to implement the DCT example at various abstraction levels in SpecC to evaluate the effectiveness of proposed modeling techniques. We also need to compile a list of all possible transaction level function calls in order to study their effect on development of channels and generic transducers. Finally, we propose to develop a new *Co-design Explorer* using the concepts developed.

Acknowledgements

We would like to thank Rockwell Semiconductor Systems for supporting this work in part through Grant #RSS-24141. This report is the result of stimulating discussions with Christopher Lennard, Nayla Nassif and Robert Larsen. We are grateful to them for their valuable insight into the problem. We thank them for their suggestions and comments. We would also like to extend our thanks to our fellow CadLab members, En-Shou Chang, Rainer Dömer, Jon Kleinsmith, Tadatoshi Ishii and Jianwen Zhu, for their helpful discussions on the co-design methodology and problems in IP reuse.

References

[1] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer. "Essential Issues in Co-design" in *Hardware/Software Co-Design: Principles and Practice* edited by Jørgen Staunstrup, Wayne Wolf. Kluwer Academic Publishers, 1997.

[2] K. R. Rao, P. Yip. *Discrete Cosine Transform.* Academic Press, Inc. 1990.

[3] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer. "The SpecC+ Language". *Technical Report ICS-97-15*, UCI, April 1997.

[4] Jianwen Zhu, Rainer Dömer, Daniel D. Gajski. "Syntax and Semantics of the SpecC+ Language". *Technical Report ICS-97-16*, UCI, April 1997.

[5] "MOS Integrated Circuit, $\mu PD431008L$". *Data Sheet*, NEC Inc., 1996.

[6] Steven Heath *"VMEbus User's Handbook"*. CRC Press, Inc., 1989.

[7] Douglas L. Perry. *VHDL*. 2^{nd} ed., McGraw-Hill, New York, 1994.

[8] Daniel Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong. *Specification and Design of Embedded Systems*, Prentice Hall, New Jersey, 1994.

[9] Daniel D. Gajski, *et al.* "Methodology for Design of Embedded Systems". *Technical Report ICS-98-07*, UCI, March 1998.

A Formal Specification of DCT

The formal specification of the Discrete Cosine Transform (DCT) operation is as follows [2].

$$F_{uv} = \frac{c(m)c(n)}{4} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f_{mn}\, cos\frac{(2m+1)u\pi}{2N} cos\frac{(2n+1)v\pi}{2N}$$

where:

f_{mn} = gray level of pixel at (m, n) in the $N \times N$ image $(0 \leq m, n \leq N-1)$
u, v = discrete frequency variables (0≤u, v≤N-1)
F_{uv} = coefficient at point (u, v) in spatial frequency domain

In typical designs (like the MPEG standard), the image is sub-divided into 8×8 blocks of pixels. We also use a value of $N = 8$ in this example. Furthermore, let $CosBlock$ be a 8×8 matrix defined by

$$CosBlock_{un} = round(factor * (\frac{1}{8} cos\frac{(2n+1)u\pi}{16}))$$

An important property of the cosine transform is that the two summations are separable. Thus, the DCT can be expressed as two matrix multiplications, one after the other.

$$TempBlock = InBlock \times CosBlock^T \qquad \text{(MM1)}$$

and

$$OutBlock = CosBlock \times TempBlock \qquad \text{(MM2)}$$

B High-level Specification of DCT

The high-level specification of the DCT algorithm in VHDL is given below. The incoming matrix is first read into the *InBlock* matrix. The first matrix multiplication generates the *TempBlock* matrix using the *InBlock* and *CosBlock* matrices. The second matrix multiplication generates the *OutBlock* matrix using the *CosBlock* and *TempBlock* matrices. Finally the outgoing transform matrix is written out.

```
--------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Oct 23, 1997
--
-- High Level specification of the DCT component
-- behavioral level model with only one process
--------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity dct is
   port (
     start : in  std_logic;
       clk : in  std_logic;
       din : in  integer;
      done : out std_logic;
      dout : out integer);
end dct;

architecture behavior of dct is
begin
    process
        type mem is array (0 to 7, 0 to 7) of integer;
        variable InBlock, TempBlock, OutBlock: mem;
        variable CosBlock : mem :=
          ((125,  122,  115,  103,  88,   69,   47,   24),
           (125,  103,   47,  -24, -88, -122, -115,  -69),
           (125,   69,  -47, -122, -88,   24,  115,  103),
           (125,   24, -115,  -69,  88,  103,  -47, -122),
           (125,  -24, -115,   69,  88, -103,  -47,  122),
           (125,  -69,  -47,  122, -88,  -24,  115, -103),
           (125, -103,   47,   24, -88,  122, -115,   69),
           (125, -122,  115, -103,  88,  -69,   47,  -24));
        variable a, b, p, sum : integer;
    begin
        wait until start = '1';

        --------------------------------
        -- read the input data matrix
        --------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                InBlock(i, j) := din;
            end loop;
        end loop;

        -----------------------------
```

```
        -- matrix multiplication 1
        --------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := InBlock(i, k);
                    b := CosBlock(j, k);
                    p := a * b;
                    if (k = 0) then
                        sum := p;
                    else
                        sum := sum + p;
                    end if;
                    if (k = 7) then
                        TempBlock(i, j) := sum;
                    end if;
                end loop;
            end loop;
        end loop;

        --------------------------------
        -- matrix multiplication 2
        --------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := TempBlock(k, j);
                    b := CosBlock(i, k);
                    p := a * b;
                    if (k = 0) then
                        sum := p;
                    else
                        sum := sum + p;
                    end if;
                    if (k = 7) then
                        OutBlock(i, j) := sum;
                    end if;
                end loop;
            end loop;
        end loop;

        -------------------------
        -- output the matrix
        -------------------------
        done <= '1';
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dout <= OutBlock(i, j);
            end loop;
        end loop;
        done <= '0';
    end process;
end behavior;
```

C Partitioned model for DCT with global variables

```
--------------------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Oct 25, 1997
--
-- DCT example with 2 components for the 2 matrix multiplications
-- The two components communicate using global variables
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity dct is
   port ( start : in  std_logic;
            clk : in  std_logic;
            din : in  integer;
           done : out std_logic;
           dout : out integer);
end dct;

architecture beh of dct is
   type mem is array (0 to 7, 0 to 7) of integer;
   signal tempblock : mem;
   signal finish : std_logic;
   signal cosblock : mem :=
       ((125,  122,  115,  103,  88,   69,   47,   24),
        (125,  103,   47,  -24, -88, -122, -115,  -69),
        (125,   69,  -47, -122, -88,   24,  115,  103),
        (125,   24, -115,  -69,  88,  103,  -47, -122),
        (125,  -24, -115,   69,  88, -103,  -47,  122),
        (125,  -69,  -47,  122, -88,  -24,  115, -103),
        (125, -103,   47,   24, -88,  122, -115,   69),
        (125, -122,  115, -103,  88,  -69,   47,  -24));
begin
   mm1 : process
        variable inblock : mem;
        variable a, b, p, sum : integer;
   begin
        wait until start = '1';
        finish <= '0';
        -- read the input data matrix
        for i in 0 to 7 loop
           for j in 0 to 7 loop
               wait until clk = '1';
               inblock (i, j) := din;
           end loop;
        end loop;

        -- do the computation
        for i in 0 to 7 loop
           for j in 0 to 7 loop
              for k in 0 to 7 loop
                 a := inblock (i, k);
                 b := cosblock (j, k);
                 p := a * b;
```

```
                if (k = 0) then
                    sum := p;
                else
                    sum := sum + p;
                end if;

                if (k = 7) then
                    tempblock(i, j) <= sum;
                end if;
            end loop;
        end loop;
    end loop;

    -- done with the computation, make finish true
    finish <= '1';
  end process;

  mm2: process
    variable outblock : mem;
    variable a, b, p, sum : integer;
  begin
    wait until finish = '1';
    -- do the computation
    for i in 0 to 7 loop
        for j in 0 to 7 loop
            for k in 0 to 7 loop
                a := tempblock(k, j);
                b := cosblock(i, k);
                p := a * b;

                if (k = 0) then
                    sum := p;
                else
                    sum := sum + p;
                end if;

                if (k = 7) then
                    outblock(i, j) := sum;
                end if;
            end loop;
        end loop;
    end loop;

    -- give the done signal and output the output matrix
    done <= '1';
    for i in 0 to 7 loop
        for j in 0 to 7 loop
            wait until clk = '1';
            dout <= outblock(i, j);
        end loop;
    end loop;
    done <= '0';
  end process;
end beh;
```

D Protocol 1: Handshaking for each byte transfer

```vhdl
--------------------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Oct 30, 1997
--
-- DCT example with 2 components for the 2 matrix multiplications
-- communication over a dedicated bus with handshaking for each byte
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm1 is
   port ( start : in  std_logic;
            clk : in  std_logic;
            din : in  std_logic_vector(7 downto 0);
          txreq : out std_logic;
          txack : in  std_logic;
           dreq : out std_logic;
           dack : in  std_logic;
           addr : out std_logic_vector(5 downto 0);
           data : out std_logic_vector(7 downto 0));
end mm1;

architecture behav of mm1 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal InBlock, TempBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
      variable a, b : std_logic_vector(7 downto 0);
      variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

        -- first matrix multiplication calculation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := InBlock(i, k);
                    b := CosBlock(j, k);
                    p := signed(a) * signed(b);

```

```
                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        TempBlock (i, j) <= sum(15 downto 8);
                    end if;
                end loop;
            end loop;
        end loop;

        -- done with the computation
        over <= '1';
    end process compute;

    communicate: process
    begin
        ready <= '0';
        wait until start = '1';
        txreq <= '0';
        dreq <= '0';

        -- read the input data matrix
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                InBlock (i, j) <= din;
            end loop;
        end loop;
        ready <= '1';

        -- transfer the data
        wait until over = '1';
        ready <= '0';
        txreq <= '1';
        wait until txack = '1';

        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dreq <= '1';
                wait until dack = '1';
                addr <= conv_std_logic_vector(8*i+j, 6);
                data <= tempblock (i, j);
                dreq <= '0' after 3 ns;  -- mem read time
                wait until dack = '0';
            end loop;
        end loop;
        txreq <= '0';
    end process communicate;
end behav;

library ieee;
```

```vhdl
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm2 is
   port (  done : out std_logic;
            clk : in  std_logic;
           data : in  std_logic_vector(7 downto 0);
           addr : in  std_logic_vector(5 downto 0);
          txreq : in  std_logic;
          txack : out std_logic;
           dreq : in  std_logic;
           dack : out std_logic;
           dout : out std_logic_vector(7 downto 0));
end mm2;

architecture behav of mm2 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal TempBlock, OutBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable CosBlock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
       variable a, b : std_logic_vector(7 downto 0);
       variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

        -- second matrix multiplication computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := TempBlock (k, j);
                    b := CosBlock (i, k);
                    p := signed(a) * signed(b);

                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        OutBlock (i, j) <= sum(15 downto 8);
                    end if;
                end loop;
```

```
            end loop;
        end loop;

        -- over with the computation
        over <= '1';
    end process compute;

    communicate: process
        variable tempi, tempj : integer;
    begin
        txack <= '0';
        dack <= '0';
        done <= '0';
        wait until txreq = '1';
        wait until clk = '1';
        txack <= '1' after 1 ns;

        while (txreq = '1') loop
            wait until (dreq = '1' OR txreq = '0');
            if (txreq = '1') then
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                tempi := conv_integer(unsigned(addr)) / 8;
                tempj := conv_integer(unsigned(addr)) mod 8;
                TempBlock (tempi, tempj) <= data;
                dack <= '0' after 8 ns; -- mem write time
            end if;
        end loop;

        wait until clk = '1';
        txack <= '0';
        ready <= '1';

        -- output the computed matrix
        wait until over = '1';
        ready <= '0';

        done <= '1';
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dout <= OutBlock (i, j);
            end loop;
        end loop;
        done <= '0';
    end process communicate;
end behav;

library ieee;
use ieee.std_logic_1164.all;

entity dct is
   port ( start : in  std_logic;
            clk : in  std_logic;
            din : in  std_logic_vector(7 downto 0);
```

```
            done : out std_logic;
            dout : out std_logic_vector(7 downto 0));
end dct;

architecture struct of dct is
    component mm1
        port ( start : in  std_logic;
                 clk : in  std_logic;
                 din : in  std_logic_vector(7 downto 0);
               txreq : out std_logic;
               txack : in  std_logic;
                dreq : out std_logic;
                dack : in  std_logic;
                addr : out std_logic_vector(5 downto 0);
                data : out std_logic_vector(7 downto 0));
    end component;

    component mm2
        port (  done : out std_logic;
                 clk : in  std_logic;
                data : in  std_logic_vector(7 downto 0);
                addr : in  std_logic_vector(5 downto 0);
               txreq : in  std_logic;
               txack : out std_logic;
                dreq : in  std_logic;
                dack : out std_logic;
                dout : out std_logic_vector(7 downto 0));
    end component;

    signal txreq, txack, dreq, dack : std_logic;
    signal data : std_logic_vector(7 downto 0);
    signal addr : std_logic_vector (5 downto 0);
begin
    u1 : mm1
      port map (start, clk, din, txreq, txack, dreq, dack, addr, data);

    u2 : mm2
      port map (done, clk, data, addr, txreq, txack, dreq, dack, dout);
end struct;
```

E Protocol 2: Handshaking for each byte pair

```vhdl
----------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Nov 6, 1997
--
-- DCT example with 2 components for the 2 matrix multiplications
-- components communicate using a dedicated bus and handshaking
-- bus multiplexed between address and data. 2 bytes of data for
-- each address written on bus.
----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm1 is
    port ( start : in  std_logic;
             clk : in  std_logic;
             din : in  std_logic_vector(7 downto 0);
            dreq : out std_logic;
            dack : in  std_logic;
            dbus : out std_logic_vector(7 downto 0));
end mm1;

architecture behav of mm1 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal InBlock, TempBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
        variable a, b : std_logic_vector(7 downto 0);
        variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

       -- first matrix multiplication computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := InBlock(i, k);
                    b := CosBlock(j, k);
                    p := signed(a) * signed(b);

                    if (k = 0) then
```

```
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        TempBlock(i, j) <= sum(15 downto 8);
                    end if;
                end loop;
            end loop;
        end loop;

        -- computation over
        over <= '1';
    end process compute;

    communicate: process
    begin
        ready <= '0';
        wait until start = '1';
        dreq <= '0';
        dbus <= (others => 'Z');

        -- read the input data matrix
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                InBlock(i, j) <= din;
            end loop;
        end loop;
        ready <= '1';

        -- transfer the data
        wait until over = '1';
        ready <= '0';
        wait until clk = '1';
        for i in 0 to 7 loop
            for j in 0 to 3 loop
                dreq <= '1' after 2 ns;
                wait until dack = '1';
                wait until clk = '1';
                dbus <= conv_std_logic_vector(8*i+2*j, 8);
                dreq <= '0';
                wait until clk = '1';
                dbus <= tempblock(i, 2*j);
                wait until clk = '1';
                dbus <= tempblock(i, 2*j+1);
                wait until clk = '1';
                dbus <= (others => 'Z');
                wait until dack = '0';
            end loop;
        end loop;
    end process communicate;
end behav;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm2 is
    port (  done : out std_logic;
             clk  : in  std_logic;
            dbus : in  std_logic_vector(7 downto 0);
            dreq : in  std_logic;
            dack : out std_logic;
            dout : out std_logic_vector(7 downto 0));
end mm2;

architecture behav of mm2 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal TempBlock, OutBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
       variable a, b : std_logic_vector(7 downto 0);
       variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

        -- second matrix multiplication computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := TempBlock(k, j);
                    b := CosBlock(i, k);
                    p := signed(a) * signed(b);

                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        OutBlock(i, j) <= sum(15 downto 8);
                    end if;
                end loop;
            end loop;
        end loop;
```

```
        -- over with computation
        over <= '1';
    end process compute;

    communicate: process
        variable tempi, tempj : integer;
    begin
        dack <= '0';
        done <= '0';
        ready <= '0';

        for i in 0 to 7 loop
            for j in 0 to 3 loop
                wait until dreq = '1';
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                wait until clk = '1';
                tempi := conv_integer(unsigned(dbus)) / 8;
                tempj := conv_integer(unsigned(dbus)) mod 8;
                wait until clk = '1';
                TempBlock (tempi, tempj) <= dbus;
                wait until clk = '1';
                TempBlock (tempi, tempj+1) <= dbus;
                dack <= '0' after 1 ns; -- mem write time
            end loop;
        end loop;
        wait until clk = '1';
        ready <= '1';

        -- output the matrix
        wait until over = '1';
        ready <= '0';
        done <= '1';
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dout <= OutBlock (i, j);
            end loop;
        end loop;
        done <= '0';
    end process communicate;
end behav;

library ieee;
use ieee.std_logic_1164.all;

entity dct is
   port ( start : in  std_logic;
            clk : in  std_logic;
            din : in  std_logic_vector(7 downto 0);
           done : out std_logic;
           dout : out std_logic_vector(7 downto 0));
end dct;
```

```
architecture struct of dct is
   component mm1
   port ( start : in  std_logic;
            clk : in  std_logic;
            din : in  std_logic_vector(7 downto 0);
           dreq : out std_logic;
           dack : in  std_logic;
           dbus : out std_logic_vector(7 downto 0));
   end component;

   component mm2
   port (  done : out std_logic;
            clk : in  std_logic;
           dbus : in  std_logic_vector(7 downto 0);
           dreq : in  std_logic;
           dack : out std_logic;
           dout : out std_logic_vector(7 downto 0));
   end component;

   signal dreq, dack : std_logic;
   signal dbus : std_logic_vector(7 downto 0);
begin
   u1 : mm1
        port map (start, clk, din, dreq, dack, dbus);

   u2 : mm2
        port map (done, clk, dbus, dreq, dack, dout);
end struct;
```

F Replacing an ASIC

```
--------------------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Nov 13, 1997
--
-- DCT example with 2 components for the 2 matrix multiplications
-- components have different protocols. an interface is used
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm1 is
    port ( start : in  std_logic;
             clk : in  std_logic;
             din : in  std_logic_vector(7 downto 0);
           txreq : out std_logic;
           txack : in  std_logic;
            dreq : out std_logic;
            dack : in  std_logic;
            addr : out std_logic_vector(5 downto 0);
            data : out std_logic_vector(7 downto 0));
end mm1;

architecture behav of mm1 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal InBlock, TempBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
      variable a, b : std_logic_vector(7 downto 0);
      variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

        -- first matrix multiplication computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := inblock (i, k);
                    b := cosblock (j, k);
                    p := signed(a) * signed(b);

```

```
                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        TempBlock (i, j) <= sum(15 downto 8);
                    end if;
                end loop;
            end loop;
        end loop;

        -- computation over
        over <= '1';
    end process compute;

    communicate: process
    begin
        ready <= '0';
        wait until start = '1';
        txreq <= '0';
        dreq <= '0';

        -- read the input data matrix
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                InBlock (i, j) <= din;
            end loop;
        end loop;
        ready <= '1';

        -- transfer the data
        wait until over = '1';
        ready <= '0';
        txreq <= '1';
        wait until txack = '1';

        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dreq <= '1';
                wait until dack = '1';
                addr <= conv_std_logic_vector(8*i+j, 6);
                data <= tempblock (i, j);
                dreq <= '0' after 3 ns;  -- mem read time
                wait until dack = '0';
            end loop;
        end loop;
        txreq <= '0';
    end process communicate;
end behav;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm2 is
    port (  done : out std_logic;
             clk : in  std_logic;
            dbus : in  std_logic_vector(7 downto 0);
            dreq : in  std_logic;
            dack : out std_logic;
            dout : out std_logic_vector(7 downto 0));
end mm2;

architecture behav of mm2 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal TempBlock, OutBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
       variable a, b : std_logic_vector(7 downto 0);
       variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

        -- second matrix multiplication computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := TempBlock(k, j);
                    b := CosBlock(i, k);
                    p := signed(a) * signed(b);

                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        OutBlock(i, j) <= sum(15 downto 8);
                    end if;
                end loop;
            end loop;
        end loop;
```

```vhdl

        -- over with computation
        over <= '1';
    end process compute;

    communicate: process
        variable tempi, tempj : integer;
    begin
        dack <= '0';
        done <= '0';
        ready <= '0';

        for i in 0 to 7 loop
            for j in 0 to 3 loop
                wait until dreq = '1';
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                wait until clk = '1';
                tempi := conv_integer(unsigned(dbus)) / 8;
                tempj := conv_integer(unsigned(dbus)) mod 8;
                wait until clk = '1';
                TempBlock (tempi, tempj) <= dbus;
                wait until clk = '1';
                TempBlock (tempi, tempj+1) <= dbus;
                dack <= '0' after 1 ns; -- mem write time
            end loop;
        end loop;
        wait until clk = '1';
        ready <= '1';

        -- output the matrix
        wait until over = '1';
        ready <= '0';
        done <= '1';
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dout <= OutBlock (i, j);
            end loop;
        end loop;
        done <= '0';
    end process communicate;
end behav;


library ieee;
use ieee.std_logic_1164.all;

entity transducer is
    port (  clk : in  std_logic;
          txreq : in  std_logic;
          txack : out std_logic;
           dreq : in  std_logic;
           dack : out std_logic;
           addr : in  std_logic_vector(5 downto 0);
```

```
            data : in  std_logic_vector(7 downto 0);
            dbus : out std_logic_vector(7 downto 0);
             req : out std_logic;
             ack : in  std_logic);
end transducer;

architecture behav of transducer is
begin
    process
        variable taddr1, taddr2 : std_logic_vector(7 downto 0);
        variable tdata1, tdata2 : std_logic_vector(7 downto 0);
    begin
        txack <= '0';
        req <= '0';
        dack <= '0';

        wait until txreq = '1';
        wait until clk = '1';
        txack <= '1';

        while txreq = '1' loop
            wait until (dreq = '1' OR txreq = '0');
            if (txreq = '1') then
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                -- receive first address and data byte
                taddr1 := "00" & addr;
                tdata1 := data;
                dack <= '0' after 2 ns;
                wait until dreq = '1';
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                -- receive second address and data byte
                taddr2 := "00" & addr;
                tdata2 := data;
                -- ready to send two bytes
                req <= '1';
                wait until ack = '1';
                wait until clk = '1';
                dbus <= taddr1;
                req <= '0';
                wait until clk = '1';
                dbus <= tdata1;
                wait until clk = '1';
                dbus <= tdata2;
                wait until clk = '1';
                dbus <= (others => 'Z');
                wait until ack = '0';
                -- give the ack signal to first process
                dack <= '0' after 2 ns;
            end if;
        end loop;
        txack <= '0' after 2 ns;
    end process;
end behav;
```

```vhdl


library ieee;
use ieee.std_logic_1164.all;

entity dct is
    port ( start : in  std_logic;
             clk : in  std_logic;
             din : in  std_logic_vector(7 downto 0);
            done : out std_logic;
            dout : out std_logic_vector(7 downto 0));
end dct;

architecture struct of dct is
    component mm1
        port ( start : in  std_logic;
                 clk : in  std_logic;
                 din : in  std_logic_vector(7 downto 0);
               txreq : out std_logic;
               txack : in  std_logic;
                dreq : out std_logic;
                dack : in  std_logic;
                addr : out std_logic_vector(5 downto 0);
                data : out std_logic_vector(7 downto 0));
    end component;

    component mm2
        port (  done : out std_logic;
                 clk : in  std_logic;
                dbus : in  std_logic_vector(7 downto 0);
                dreq : in  std_logic;
                dack : out std_logic;
                dout : out std_logic_vector(7 downto 0));
    end component;

    component transducer
        port (  clk : in  std_logic;
              txreq : in  std_logic;
              txack : out std_logic;
               dreq : in  std_logic;
               dack : out std_logic;
               addr : in  std_logic_vector(5 downto 0);
               data : in  std_logic_vector(7 downto 0);
               dbus : out std_logic_vector(7 downto 0);
                req : out std_logic;
                ack : in  std_logic);
    end component;

    signal txreq, txack, dreq, dack, req, ack : std_logic;
    signal data, dbus : std_logic_vector(7 downto 0);
    signal addr : std_logic_vector (5 downto 0);

begin
    u1 : mm1
         port map (start, clk, din, txreq, txack, dreq, dack, addr, data);
```

```
    u2 : mm2
        port map (done, clk, dbus, req, ack, dout);

    u3 : transducer
        port map (clk, txreq, txack, dreq, dack, addr, data, dbus, req, ack);
end struct;
```

G Replacing the bus

```
--------------------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Dec 2, 1997
--
-- replace a bus with another bus that has different protocol
-- two transducer need to be used
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm1 is
   port ( start : in  std_logic;
            clk : in  std_logic;
            din : in  std_logic_vector(7 downto 0);
          txreq : out std_logic;
          txack : in  std_logic;
           dreq : out std_logic;
           dack : in  std_logic;
           addr : out std_logic_vector(5 downto 0);
           data : out std_logic_vector(7 downto 0));
end mm1;

architecture behav of mm1 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal InBlock, TempBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
       variable a, b : std_logic_vector(7 downto 0);
       variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

        -- first matrix multiplication calculation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := InBlock (i, k);
                    b := CosBlock (j, k);
                    p := signed(a) * signed(b);

```

```
                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        TempBlock (i, j) <= sum(15 downto 8);
                    end if;
                end loop;
            end loop;
        end loop;

        -- done with the computation
        over <= '1';
    end process compute;

    communicate: process
    begin
        ready <= '0';
        wait until start = '1';
        txreq <= '0';
        dreq <= '0';

        -- read the input data matrix
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                InBlock (i, j) <= din;
            end loop;
        end loop;
        ready <= '1';

        -- transfer the data
        wait until over = '1';
        ready <= '0';
        txreq <= '1';
        wait until txack = '1';

        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dreq <= '1';
                wait until dack = '1';
                addr <= conv_std_logic_vector(8*i+j, 6);
                data <= tempblock (i, j);
                dreq <= '0' after 3 ns;  -- mem read time
                wait until dack = '0';
            end loop;
        end loop;
        txreq <= '0';
    end process communicate;
end behav;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm2 is
   port (  done : out std_logic;
            clk : in  std_logic;
           data : in  std_logic_vector(7 downto 0);
           addr : in  std_logic_vector(5 downto 0);
          txreq : in  std_logic;
          txack : out std_logic;
           dreq : in  std_logic;
           dack : out std_logic;
           dout : out std_logic_vector(7 downto 0));
end mm2;

architecture behav of mm2 is
    type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
    signal TempBlock, OutBlock : mem;
    signal ready, over : std_logic;
begin
    compute: process
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
        variable a, b : std_logic_vector(7 downto 0);
        variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until ready = '1';
        over <= '0';

        -- second matrix multiplication computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := TempBlock (k, j);
                    b := CosBlock (i, k);
                    p := signed(a) * signed(b);

                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        OutBlock (i, j) <= sum(15 downto 8);
                    end if;
```

```
                end loop;
            end loop;
        end loop;

        -- over with the computation
        over <= '1';
    end process compute;

    communicate: process
        variable tempi, tempj : integer;
    begin
        txack <= '0';
        dack <= '0';
        done <= '0';
        wait until txreq = '1';
        wait until clk = '1';
        txack <= '1' after 1 ns;

        while (txreq = '1') loop
            wait until (dreq = '1' OR txreq = '0');
            if (txreq = '1') then
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                tempi := conv_integer(unsigned(addr)) / 8;
                tempj := conv_integer(unsigned(addr)) mod 8;
                TempBlock (tempi, tempj) <= data;
                dack <= '0' after 8 ns; -- mem write time
            end if;
        end loop;

        wait until clk = '1';
        txack <= '0';
        ready <= '1';

        -- output the computed matrix
        wait until over = '1';
        ready <= '0';

        done <= '1';
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                dout <= OutBlock (i, j);
            end loop;
        end loop;
        done <= '0';
    end process communicate;
end behav;

library ieee;
use ieee.std_logic_1164.all;

-- converts bus 1 protocol (4 control signals) to the
-- bus 2 protocol (2 control signals)
entity transducer1 is
```

```
    port (   clk : in  std_logic;
           txreq : in  std_logic;
           txack : out std_logic;
            dreq : in  std_logic;
            dack : out std_logic;
            addr : in  std_logic_vector(5 downto 0);
            data : in  std_logic_vector(7 downto 0);
            dbus : out std_logic_vector(7 downto 0);
             req : out std_logic;
             ack : in  std_logic);
end transducer1;

architecture behav of transducer1 is
begin
    process
        variable taddr1, taddr2 : std_logic_vector(7 downto 0);
        variable tdata1, tdata2 : std_logic_vector(7 downto 0);
    begin
        txack <= '0';
        req <= '0';
        dack <= '0';

        wait until txreq = '1';
        wait until clk = '1';
        txack <= '1';

        while txreq = '1' loop
            wait until (dreq = '1' OR txreq = '0');
            if (txreq = '1') then
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                -- receive first address and data byte
                taddr1 := "00" & addr;
                tdata1 := data;
                dack <= '0' after 2 ns;
                wait until dreq = '1';
                dack <= '1' after 1 ns;
                wait until dreq = '0';
                -- receive second address and data byte
                taddr2 := "00" & addr;
                tdata2 := data;
                -- ready to send two bytes
                req <= '1';
                wait until ack = '1';
                wait until clk = '1';
                dbus <= taddr1;
                req <= '0';
                wait until clk = '1';
                dbus <= tdata1;
                wait until clk = '1';
                dbus <= tdata2;
                wait until clk = '1';
                dbus <= (others => 'Z');
                wait until ack = '0';
                -- give the ack signal to first process
```

```
                dack <= '0' after 2 ns;
            end if;
        end loop;
        txack <= '0' after 2 ns;
    end process;
end behav;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-- converts bus 2 protocol (2 control signals) to the
-- bus 1 protocol (4 control signals)
entity transducer2 is
    port (   clk : in  std_logic;
           txreq : out std_logic;
           txack : in  std_logic;
            dreq : out std_logic;
            dack : in  std_logic;
            addr : out std_logic_vector(5 downto 0);
            data : out std_logic_vector(7 downto 0);
            dbus : in  std_logic_vector(7 downto 0);
             req : in  std_logic;
             ack : out std_logic);
end transducer2;

architecture behav of transducer2 is
begin
    process
        variable tadd : std_logic_vector (5 downto 0);
        variable data1, data2 : std_logic_vector (7 downto 0);
    begin
        -- wait for protocol 1 to be ready
        txreq <= '1';
        wait until txack = '1';

        for i in 0 to 7 loop
            for j in 0 to 3 loop
                -- first read the addr and two data bytes
                wait until req = '1';
                ack <= '1' after 1 ns;
                wait until req = '0';
                wait until clk = '1';
                tadd := dbus(5 downto 0);
                wait until clk = '1';
                data1 := dbus;
                wait until clk = '1';
                data2 := dbus;
                -- send the two bytes with the addresses
                dreq <= '1';
                wait until dack = '1';
                addr <= tadd;
                data <= data1;
                dreq <= '0' after 3 ns;
```

```
                wait until dack = '0';
                wait until clk = '1';
                dreq <= '1';
                wait until dack = '1';
                addr <= conv_std_logic_vector(conv_integer(unsigned(tadd)) + 1, 6);
                data <= data2;
                dreq <= '0' after 3 ns;
                wait until dack = '0';

                wait until clk = '1';
                ack <= '0' after 1 ns;
            end loop;
        end loop;
        txreq <= '0';
        wait until txack = '0';
    end process;
end behav;


library ieee;
use ieee.std_logic_1164.all;

entity dct is
    port ( start : in  std_logic;
             clk : in  std_logic;
             din : in  std_logic_vector(7 downto 0);
            done : out std_logic;
            dout : out std_logic_vector(7 downto 0));
end dct;

architecture struct of dct is
    component mm1
        port ( start : in  std_logic;
                 clk : in  std_logic;
                 din : in  std_logic_vector(7 downto 0);
               txreq : out std_logic;
               txack : in  std_logic;
                dreq : out std_logic;
                dack : in  std_logic;
                addr : out std_logic_vector(5 downto 0);
                data : out std_logic_vector(7 downto 0));
    end component;

    component mm2
        port (  done : out std_logic;
                 clk : in  std_logic;
                data : in  std_logic_vector(7 downto 0);
                addr : in  std_logic_vector(5 downto 0);
               txreq : in  std_logic;
               txack : out std_logic;
                dreq : in  std_logic;
                dack : out std_logic;
                dout : out std_logic_vector(7 downto 0));
    end component;

```

```
    component transducer1
        port (   clk  : in   std_logic;
              txreq  : in   std_logic;
              txack  : out  std_logic;
               dreq  : in   std_logic;
               dack  : out  std_logic;
               addr  : in   std_logic_vector(5 downto 0);
               data  : in   std_logic_vector(7 downto 0);
               dbus  : out  std_logic_vector(7 downto 0);
                req  : out  std_logic;
                ack  : in   std_logic);
    end component;

    component transducer2
        port (   clk  : in   std_logic;
              txreq  : out  std_logic;
              txack  : in   std_logic;
               dreq  : out  std_logic;
               dack  : in   std_logic;
               addr  : out  std_logic_vector(5 downto 0);
               data  : out  std_logic_vector(7 downto 0);
               dbus  : in   std_logic_vector(7 downto 0);
                req  : in   std_logic;
                ack  : out  std_logic);
    end component;

    signal txreq1, txack1, dreq1, dack1, req, ack : std_logic;
    signal txreq2, txack2, dreq2, dack2 : std_logic;
    signal data1, data2, dbus : std_logic_vector(7 downto 0);
    signal addr1, addr2 : std_logic_vector (5 downto 0);
begin
    u1 : mm1
         port map (start, clk, din, txreq1, txack1, dreq1, dack1, addr1, data1);

    u2 : mm2
      port map (done, clk, data2, addr2, txreq2, txack2, dreq2, dack2, dout);

    u3 : transducer1
         port map (clk, txreq1, txack1, dreq1, dack1, addr1, data1, dbus, req, ack);

    u4 : transducer2
         port map (clk, txreq2, txack2, dreq2, dack2, addr2, data2, dbus, req, ack);
end struct;
```

H Communication using a FIFO queue

```vhdl
--------------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Nov 20, 1997
--
-- DCT example with 2 components for the 2 matrix multiplications
-- communication through a 64 byte FIFO queue with full/empty lines
--------------------------------------------------------------------------

-- model of a 64 byte FIFO
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity fifo is
    port ( enable : in  std_logic;
           reset  : in  std_logic;
             rws  : in  std_logic;
             clk  : in  std_logic;
             din  : in  std_logic_vector(7 downto 0);
            dout  : out std_logic_vector(7 downto 0);
            full  : out std_logic;
           empty  : out std_logic);
end fifo;

architecture behav of fifo is
begin
    process (clk)
        type storagetype is array(0 to 63) of std_logic_vector(7 downto 0);
        variable storage : storagetype;
        variable front, back : integer;
        variable isempty, isfull : std_logic := '0';
    begin
        if (clk = '1') then
            -- reset counters and flags
            if (reset = '1') then
                front := 0;
                back := 0;
                isempty := '1';
                isfull := '0';
            else
                -- read from fifo
                if (rws = '0' and isempty = '0' and enable = '1') then
                    dout <= storage(front);
                    front := (front + 1) mod 64;
                    if (front = back) then
                        isempty := '1';
                        isfull := '0';
                    end if;
                end if;

                -- write into fifo
                if (rws = '1' and isfull = '0' and enable = '1') then
                    storage(back) := din;
```

```
                    back := (back + 1) mod 64;
                    if (front = back) then
                        isempty := '0';
                        isfull := '1';
                    end if;
                end if;
            end if;
            empty <= isempty;
            full <= isfull;
        end if;
    end process;
end behav;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm1 is
    port ( start : in  std_logic;
             clk : in  std_logic;
             din : in  std_logic_vector (7 downto 0);
           empty : in  std_logic;
             rws : out std_logic;
          enable : out std_logic;
           reset : out std_logic;
            data : out std_logic_vector (7 downto 0));
end mm1;

architecture behav of mm1 is
   type mem is array (0 to 7, 0 to 7) of std_logic_vector (7 downto 0);
begin
    process
       variable inblock : mem;
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
       variable a, b : std_logic_vector (7 downto 0);
       variable p, sum : std_logic_vector (15 downto 0);
    begin
        wait until start = '1';
        data <= (others => 'Z');

        -- read the input data matrix
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                inblock (i, j) := din;
```

```
            end loop;
        end loop;

        -- reset the fifo
        reset <= '1';
        wait until clk = '1';
        reset <= '0';

        -- do the computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := inblock (i, k);
                    b := cosblock (j, k);
                    p := signed(a) * signed(b);

                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    -- computed one entry of matrix. write to FIFO
                    if (k = 7) then
                        enable <= '1';
                        rws <= '1';
                        data <= sum(15 downto 8);
                        wait until clk = '1';
                        enable <= '0';
                        rws <= '0';
                    end if;
                end loop;
            end loop;
        end loop;
    end process;
end behav;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm2 is
    port (  done : out std_logic;
             clk : in  std_logic;
            data : in  std_logic_vector(7 downto 0);
            full : in  std_logic;
             rws : out std_logic;
          enable : out std_logic;
            dout : out std_logic_vector(7 downto 0));
end mm2;

architecture behav of mm2 is
   type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
   type widemem is array (0 to 7, 0 to 7) of std_logic_vector(15 downto 0);
```

```
begin
    process
        -- this algorithm will use wider memory
        variable outblock : widemem;
        variable cosblock : mem := (
          ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
          ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
          ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
          ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
          ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
          ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
          ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
          ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
          );
        variable a, b : std_logic_vector(7 downto 0);
        variable p, sum, temp : std_logic_vector(15 downto 0);
    begin
        enable <= '0';
        rws <= '0';
        wait until full = '1';
        wait until clk = '1';

        -- do the computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                -- read one entry of matrix
                enable <= '1';
                wait until clk = '1';
                enable <= '0';
                wait until clk = '1';
                a := data;

                -- add the product to all the partial sums
                for k in 0 to 7 loop
                    if (i /= 0) then
                        temp := outblock(k, j);
                    end if;
                    b := cosblock (k, i);
                    p := signed(a) * signed(b);

                    if (i = 0) then
                        sum := p;
                    else
                        sum := signed(temp) + signed(p);
                    end if;
                    outblock (k, j) := sum;
                end loop;
            end loop;
        end loop;

        -- give the done signal and output the output matrix
        done <= '1';
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
```

```
                dout <= outblock (i, j) (15 downto 8);
            end loop;
        end loop;
        done <= '0';
    end process;
end behav;

library ieee;
use ieee.std_logic_1164.all;

entity dct is
    port ( start : in  std_logic;
             clk : in  std_logic;
             din : in  std_logic_vector(7 downto 0);
            done : out std_logic;
            dout : out std_logic_vector(7 downto 0));
end dct;

architecture struct of dct is
    component mm1
        port ( start : in  std_logic;
                 clk : in  std_logic;
                 din : in  std_logic_vector(7 downto 0);
               empty : in  std_logic;
                 rws : out std_logic;
              enable : out std_logic;
               reset : out std_logic;
                data : out std_logic_vector(7 downto 0));
    end component;

    component mm2
        port (  done : out std_logic;
                 clk : in  std_logic;
                data : in  std_logic_vector(7 downto 0);
                full : in  std_logic;
                 rws : out std_logic;
              enable : out std_logic;
                dout : out std_logic_vector(7 downto 0));
    end component;

    component fifo
        port ( enable : in  std_logic;
                reset : in  std_logic;
                  rws : in  std_logic;
                  clk : in  std_logic;
                  din : in  std_logic_vector(7 downto 0);
                 dout : out std_logic_vector(7 downto 0);
                 full : out std_logic;
                empty : out std_logic);
    end component;

    signal rws1, rws2, rws, enable1, enable2, enable : std_logic;
    signal full, empty, reset : std_logic;
    signal d_mm1, d_mm2 : std_logic_vector(7 downto 0);

```

```
begin
    u1 : mm1
         port map (start, clk, din, empty, rws1, enable1, reset, d_mm1);

    u2 : mm2
         port map (done, clk, d_mm2, full, rws2, enable2, dout);

    enable <= enable1 OR enable2;
    rws <= rws1 OR rws2;

    u3 : fifo
         port map (enable, reset, rws, clk, d_mm1, d_mm2, full, empty);
end struct;
```

I Communication using a shared memory

```
--------------------------------------------------------------------------------
-- Rockwell ASIC Reuse Modeling Project
-- Nov 28, 1997
--
-- DCT components communicate using a shared memory
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity mm1 is
    port ( start : in  std_logic;
             clk : in  std_logic;
             din : in  std_logic_vector(7 downto 0);
            used : in  std_logic;
            over : out std_logic;
             csm : out std_logic;
             wem : out std_logic;
            addr : out std_logic_vector(5 downto 0);
            data : out std_logic_vector(7 downto 0));
end mm1;

architecture behav of mm1 is
   type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
begin
    process
       variable inblock : mem;
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
       variable a, b : std_logic_vector(7 downto 0);
       variable p, sum : std_logic_vector(15 downto 0);
    begin
        wait until start = '1';
        data <= (others => 'Z');

        -- read the input data matrix
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                inblock(i, j) := din;
            end loop;
        end loop;

        wait until clk = '1';
```

```
        wem <= '1';
        csm <= '0';     -- enable the chip
        wait until clk = '1';

        -- do the computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    a := inblock (i, k);
                    b := cosblock (j, k);
                    p := signed(a) * signed(b);

                    if (k = 0) then
                        sum := p;
                    else
                        sum := signed(sum) + signed(p);
                    end if;

                    if (k = 7) then
                        -- write the sum to shared memory
                        addr <= conv_std_logic_vector(i*8+j, 6);
                        wait for 1 ns;
                        wem <= '0';
                        wait for 8 ns;           -- tWHZ time
                        data <= sum(15 downto 8);
                        wait for 9 ns;           -- tDW
                        wem <= '1';
                        wait for 1 ns;
                        data <= (others => 'Z');
                        wait for 2 ns; -- next cycle after this
                    end if;
                end loop;
            end loop;
        end loop;

        -- done with the computation
        csm <= '1';
        wait until clk = '1';
        over <= '1';

        -- wait till second unit finishes
        wait until used = '1';
    end process;
end behav;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mm2 is
    port (  done : out std_logic;
             clk : in  std_logic;
             csm : out std_logic;
             oem : out std_logic;
            addr : out std_logic_vector(5 downto 0);
```

```
            data : in  std_logic_vector(7 downto 0);
            used : out std_logic;
            over : in  std_logic;
            dout : out std_logic_vector(7 downto 0));
end mm2;

architecture behav of mm2 is
   type mem is array (0 to 7, 0 to 7) of std_logic_vector(7 downto 0);
   type widemem is array (0 to 7, 0 to 7) of std_logic_vector(15 downto 0);
begin
    process
       variable outblock : widemem;
       variable cosblock : mem := (
         ("01111101", "01111010", "01110011", "01100111", "01011000", "01000101", "00101111", "00011000"),
         ("01111101", "01100111", "00101111", "11101000", "10101000", "10000110", "10001101", "10111011"),
         ("01111101", "01000101", "11010001", "10000110", "10101000", "00011000", "01110011", "01100111"),
         ("01111101", "00011000", "10001101", "10111011", "01011000", "01100111", "11010001", "10000110"),
         ("01111101", "11101000", "10001101", "01000101", "01011000", "10011001", "11010001", "01111010"),
         ("01111101", "10111011", "11010001", "01111010", "10101000", "11101000", "01110011", "10011001"),
         ("01111101", "10011001", "00101111", "00011000", "10101000", "01111010", "10001101", "01000101"),
         ("01111101", "10000110", "01110011", "10011001", "01011000", "10111011", "00101111", "11101000")
         );
       variable a, b : std_logic_vector(7 downto 0);
       variable p, sum, temp : std_logic_vector(15 downto 0);
    begin
        used <= '0';
        -- do not assert the memory control lines
        oem <= '1';
        csm <= '1';
        -- wait for the first matrix mult to finish
        wait until over = '1';
        csm <= '0';
        addr <= (others => '1');

        wait until clk = '1';

        -- do the computation
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                -- get the (i,j) data from memory
                addr <= conv_std_logic_vector(i*8+j, 6);
                wait for 1 ns;
                oem <= '0';
                wait for 17 ns;
                oem <= '1';

                a := data;
                for k in 0 to 7 loop
                    if (i /= 0) then
                        temp := outblock(k, j);
                    end if;

                    b := cosblock (k, i);
                    p := signed(a) * signed(b);

```

```
                    if (i = 0) then
                        sum := p;
                    else
                        sum := signed(temp) + signed(p);
                    end if;
                    outblock (k, j) := sum;
                end loop;
                wait for 3 ns; -- next cycle
            end loop;
        end loop;

        csm <= '1';
        oem <= '1';

        -- give the done signal and output the output matrix
        wait until clk = '1';
        done <= '1';
        for i in 0 to 7 loop
          for j in 0 to 7 loop
              wait until clk = '1';
              dout <= outblock(i, j)(15 downto 8);
          end loop;
        end loop;
        done <= '0';
    end process;
end behav;

library ieee;
use ieee.std_logic_1164.all;

entity arbiter is
   port (   cs1  : in     std_logic;
            cs2  : in     std_logic;
             cs  : out    std_logic;
          addr1  : in     std_logic_vector(5 downto 0);
          addr2  : in     std_logic_vector(5 downto 0);
           addr  : out    std_logic_vector(5 downto 0);
          data1  : in     std_logic_vector(7 downto 0);
          data2  : out    std_logic_vector(7 downto 0);
           data  : inout  std_logic_vector(7 downto 0));
end arbiter;

architecture behav of arbiter is
begin

    process (cs1, cs2, addr1, addr2, data1, data)
    begin
        -- only one component should access memory
        assert (cs2/='0' or cs1/='0')
          report "memory access clash"
          severity warning;

        -- unit 1 only writes to memory
        if (cs1='0' and cs2/='0') then
            cs <= '0';
```

```vhdl
            addr <= addr1;
            data <= data1;
            -- unit 2 only reads from memory
        elsif (cs1/='0' and cs2='0') then
            cs <= '0';
            addr <= addr2;
            data2 <= data;
        else
            cs <= '1';
        end if;
    end process;
end behav;


library ieee;
use ieee.std_logic_1164.all;

entity dct is
   port ( start : in  std_logic;
            clk : in  std_logic;
            din : in  std_logic_vector(7 downto 0);
           done : out std_logic;
           dout : out std_logic_vector(7 downto 0));
end dct;

architecture struct of dct is
    component mm1
        port ( start : in  std_logic;
                 clk : in  std_logic;
                 din : in  std_logic_vector(7 downto 0);
                used : in  std_logic;
                over : out std_logic;
                 csm : out std_logic;
                 wem : out std_logic;
                addr : out std_logic_vector(5 downto 0);
                data : out std_logic_vector(7 downto 0));
    end component;

    component mm2
        port (  done : out std_logic;
                 clk : in  std_logic;
                 csm : out std_logic;
                 oem : out std_logic;
                addr : out std_logic_vector(5 downto 0);
                data : in  std_logic_vector(7 downto 0);
                used : out std_logic;
                over : in  std_logic;
                dout : out std_logic_vector(7 downto 0));
    end component;

    component sram
        port ( nce : in    std_logic;
               noe : in    std_logic;
               nwe : in    std_logic;
                 a : in    std_logic_vector(5 downto 0);
```

```vhdl
                d : inout std_logic_vector(7 downto 0));
    end component;

    component arbiter
        port (  cs1 : in    std_logic;
                cs2 : in    std_logic;
                 cs : out   std_logic;
              addr1 : in    std_logic_vector(5 downto 0);
              addr2 : in    std_logic_vector(5 downto 0);
               addr : out   std_logic_vector(5 downto 0);
              data1 : in    std_logic_vector(7 downto 0);
              data2 : out   std_logic_vector(7 downto 0);
               data : inout std_logic_vector(7 downto 0));
    end component;

    signal csm1, csm2, cs, oem, wem, used, over : std_logic;
    signal addr1, addr2, addr : std_logic_vector(5 downto 0);
    signal data, data1, data2 : std_logic_vector(7 downto 0);

begin
    u1 : mm1
         port map (start, clk, din, used, over, csm1, wem, addr1, data1);

    u2 : mm2
         port map (done, clk, csm2, oem, addr2, data2, used, over, dout);

    u3 : sram
         port map (cs, oem, wem, addr, data);

    u4 : arbiter
         port map (csm1, csm2, cs, addr1, addr2, addr, data1, data2, data);
end struct;
```