# 3

# THE ORDERED TRANSACTION STRATEGY

The self-timed scheduling strategy in Chapter 1 introduces synchronization checks when processors communicate; such checks permit variations in actor execution times, but they also imply run time synchronization and arbitration costs. In this chapter we present a hardware architecture approach called Ordered Transactions (OT) that alleviates some of these costs, and in doing so, trades off some of the run time flexibility afforded by the ST approach. The ordered transactions strategy was first proposed by Bier, Lee, and Sriram [Lee90][Bier90]. In this chapter we describe the idea behind the OT approach and then we discuss the design and hardware implementation of a shared-bus multiprocessor that makes use of this strategy to achieve a low-cost interprocessor communication using simple hardware. The software environment for this board is provided by the Ptolemy system developed at the University of California at Berkeley [Buck94][Ptol94].

## 3.1 The Ordered Transactions strategy

In the OT strategy we first obtain a fully-static schedule using the execution time estimates, but we discard the precise timing information specified in the fully-static schedule; as in the ST schedule we retain the processor assignment ($\sigma_p$) and actor ordering on each processor as specified by $\sigma_t$; in addition, we also retain the order in which processors communicate with one another and we enforce this order at run time. We formalize the concept of transaction order below.

Suppose there are $k$ inter-processor communication points $(s_1, r_1)$, $(s_2, r_2)$, …, $(s_k, r_k)$ — where each $(s_i, r_i)$ is a *send-receive* pair — in the FS schedule that we obtain as a first step in the construction of a self-timed schedule. Let $R$ be the set of *receive* actors, and $S$ be the set of *send* actors (i.e. $R \equiv \{r_1, r_2, \ldots, r_k\}$ and $S \equiv \{s_1, s_2, \ldots, s_k\}$ ). We define a **transaction order** to be a sequence $O = (v_1, v_2, v_3, \ldots, v_{2k-1}, v_{2k})$, where $\{v_1, v_2, \ldots, v_{2k-1}, v_{2k}\} \equiv S \cup R$ (each communication actor is present in the sequence $O$). We say a transaction order $O$ (as defined above) is **imposed** on a multiprocessor if at run time the *send* and *receive* actors are forced to execute in the sequence specified by $O$. That is, if $O = (v_1, v_2, v_3, \ldots, v_{2k-1}, v_{2k})$, then imposing $O$ means ensuring the constraints: $end(v_1, k) \leq start(v_2, k)$, $end(v_2, k) \leq start(v_3, k)$, …, $end(v_{k-1}, k) \leq start(v_k, k)$ ; $\forall k \geq 0$.

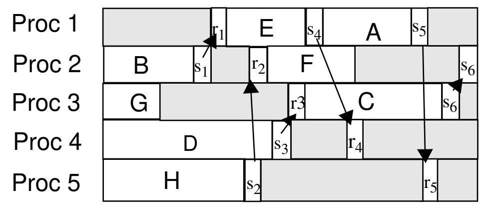
Thus the OT schedule is essentially an ST schedule with the added transaction order constraints specified by $O$.

After an FS schedule is obtained using the execution time estimates, the transaction order is obtained from the $\sigma_t$ function of the FS schedule: we simply set the transaction order to $O = (v_1, v_2, v_3, \ldots, v_{2k-1}, v_{2k})$, where

$$\sigma_t(v_1) \leq \sigma_t(v_2) \leq \ldots \leq \sigma_t(v_{2k-1}) \leq \sigma_t(v_{2k}) .$$

The transaction order can therefore be determined by sorting the set of communication actors ($S \cup R$) according to their start times $\sigma_t$. Fig. 3.1 shows an example of how such an order could be derived from a given fully-static schedule. This FS schedule corresponds to the HSDFG and schedule illustrated in Chapter 1 (Fig. 1.2).

The transaction order is enforced at run time by a controller implemented in hardware. The main advantage of ordering inter-processor transactions is that it




Transaction order: $(s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6)$

Figure 3.1. One possible transaction order derived from the fully-static schedule

allows us to restrict access to communication resources statically, based on the communication pattern determined at compile time. Since communication resources are typically shared between processors, run time contention for these resources is eliminated by ordering processor accesses to them; this results in an efficient IPC mechanism at low hardware cost. We have built a prototype four processor DSP board, called the Ordered Memory Access (OMA) architecture, that demonstrates the ordered transactions concept. The OMA prototype board utilizes shared memory and a single shared bus for IPC — the sender writes data to a particular shared memory location that is allocated at compile time, and the receiver reads that location. In this multiprocessor, a very simple controller on the board enforces the pre-determined transaction order at run time, thus eliminating the need for run time bus arbitration or semaphore synchronization. This results in efficient IPC (comparable to the FS strategy) at relatively low hardware cost. As in the ST scenario, the OT strategy is tolerant of variations in execution times of actors, because the transaction order enforces correct sender-receiver synchronization; however, this strategy is more constrained than ST scheduling, which allows the order in which communication actors fire to vary at run time. The ordered transactions strategy, therefore, falls in between fully-static and self-timed strategies in that, like the ST strategy, it is tolerant of variations in execution times and, like the FS strategy, has low communication and synchronization costs. These performance issues will be discussed quantitatively in the following chapter; the rest of this chapter describes the hardware and software implementation of the OMA prototype.

## 3.2 Shared bus architecture

The OMA architecture uses a single shared bus and shared memory for inter-processor communication. This kind of shared memory architecture is attractive for embedded multiprocessor implementations owing to its relative simplicity and low hardware cost and to the fact that it is moderately scalable — a fully interconnected processor topology, for example, would not only be much more expensive than a shared bus topology, but would also suffer from its limited scalability. Bus bandwidth limits scalability in shared bus multiprocessors, but for medium throughput applications (digital audio, music, etc.) and the size of the machine we are considering, a shared bus is ideal. We propose to solve the scalability problem by using multiple busses and hierarchy of busses, for which the ideas behind the OMA architecture directly apply. We refer to Lee and Bier [Lee90] for how the OMA concept is extended to such hierarchical bus structures.

From Fig. 1.3 we recall that the self-timed scheduling strategy falls naturally into a message passing paradigm that is implemented by the send and receive primitives inserted in the HSDFG. Accordingly, the shared memory in an architecture implementing such a scheduling strategy is used solely for message passing: the send primitive corresponds to writes to shared memory locations, and receive primitives correspond to reads from shared memory. Thus the shared memory is not used for storing shared data structures or for storing shared program code. In a self-timed strategy we can further ensure, at compile time, that each shared memory location is written by only one processor (one way of doing this is to simply assign distinct shared buffers to each of the send primitives, which is the scheme implemented in the Ptolemy environment); as a result, no atomic test-and-set instruction needs to be provided by the hardware.

Let us now consider the implementation of IPC in self-timed schedules on such a shared bus multiprocessor. The sender has to write into shared memory, which involves arbitration costs — it has to request access to the shared bus, and the access must be arbitrated by a bus arbiter. Once the sender obtains access to shared memory, it needs to perform a synchronization check on the shared memory location to ensure that the receiver has read data that was written in the previous iteration, to avoid overwriting previously written data. Such synchronization is typically implemented using a semaphore mechanism; the sender waits until a semaphore is reset before writing to a shared memory location, and upon writing that shared memory location, it sets that semaphore (the semaphore could be a bit in shared memory, one bit for each send operation in the parallel schedule). The receiver on the other hand busy waits until the semaphore is set before reading the shared memory location, and resets the semaphore after completing the read operation. It can easily be verified that this simple protocol guarantees correct sender-receiver synchronization, and, even though the semaphore bits have multiple writers, no atomic test-and-set operation is required of the hardware.

In summary the operations of the sender are: request bus, wait for arbitration, busy wait until semaphore is in the correct state, write the shared memory location if semaphore is in the correct state, and then release the bus. The corresponding operations for the receiver are: request bus, wait for arbitration, busy wait on semaphore, read the shared memory location if semaphore is in the correct state, and release the bus. The IPC costs are therefore due to bus arbitration time and due to semaphore checks. Such overhead consumes of the order of tens of instruction cycles if no special hardware support is employed for IPC. In addition, semaphore checks consume shared bus bandwidth.

An example of this is a four processor DSP56000 based shared bus system designed by Dolby labs for digital audio processing applications. In this machine, processors communicate through shared memory, and a central bus arbiter resolves bus request conflicts between processors. When a processor gets the bus it performs a semaphore check, and continues with the shared memory transaction if the semaphore is in the correct state. It explicitly releases the bus after completing the shared memory transaction. A receive and a send together consume 30 instruction cycles, even if the semaphores are in their correct state and the processor gets the bus immediately upon request. This translates to 8% of the 380 instructions per processor in the example of Chapter 1, section 1.4, that considered processing samples of a high-quality audio signal at a sampling rate of 44 KHz on processors running on a 60ns clock. Such a high cost of communication forces the scheduler to insert as few interprocessor communication nodes as possible, which in turn limits the amount of parallelism that can be extracted from the algorithm.

One solution to this problem is to send more than one data sample when a processor gets access to the bus; the arbitration and synchronization costs are then amortized over several data samples. A scheme to "vectorize" data in this manner has been proposed by [Zivo94], where the authors use retiming [Lei91] to move delays in the HSDFG such that data can be moved in blocks, instead of one sample at a time. There are several problems with this strategy. First, retiming HSDFGs has to be done very carefully: moving delays across actors can change the initial state of the HSDFG causing undesirable transients in the algorithm implementation. This can potentially be solved by including preamble code to compute the value of the sample corresponding to the delay when that delay is moved across actors. This, however results in increased code size, and other associated code generation complications. Second, the work of Zivojinovic *et. al.* does not apply uniformly to all HSDFGs: if there are tight cycles in the graph that need to be partitioned among processors, the samples simply cannot be "vectorized" [Messer88]. Thus presence of a tight cycle precludes arbitrary blocking of data. Third, vectorizing samples leads to increased latency in the implementation; some signal processing tasks such as interactive speech are sensitive to delay, and hence the delay introduced due to blocking of data may be unacceptable. Finally, the problem of vectorizing data in HSDFGs into blocks, even with all the above limitations, appear to be fundamentally hard; the algorithms proposed by Zivojinovic *et. al.* have exponential worst case run times. Code generated currently by the Ptolemy system does not support blocking (or vectorizing) of data for many of the above reasons.

Another possible solution is to use special hardware. One could provide a full interconnection network, thus obviating the need to go through shared memory. Semaphores could be implemented in hardware. One could use multiported memories. Needless to say, this solution is not favourable because of cost, especially when targeting embedded applications.

A general-purpose shared bus machine, the Sequent Balance [Patt90] for example, will typically use caches between the processor and the shared bus. Caches lead to increased shared memory bandwidth due to the averaging effect provided by block fetches and due to probabilistic memory access speedup due to cache hits. In signal processing and other real time applications, however, there is a stringent requirement for deterministic performance guarantee as opposed to probabilistic speedup. In fact, the unpredictability in task execution times introduced due to the use of caches may be a disadvantage for static scheduling techniques that utilize compile time estimates of task execution times to make scheduling decisions (we recall the discussion in section 1.2.3 on techniques for estimating task execution times). In addition, due to the deterministic nature of most signal processing problems (and also many scientific computation problems), shared data can be deterministically prefetched because information about when particular blocks of data are required by a particular processor can often be predicted by a compiler. This feature has been studied in [Mouss92], where the authors propose memory allocation schemes that exploit predictability in the memory access pattern in DSP algorithms; such a “smart allocation” scheme alleviates some of the memory bandwidth problems associated with high throughput applications.

Processors with caches can cache semaphores locally, so that busy waiting can be done local to the processor without having to access the shared bus, hence saving the bus bandwidth normally expended on semaphore checks. Such a procedure, however, requires special hardware (a snooping cache controller, for example) to maintain cache coherence; cost of such hardware usually makes it prohibitive in embedded scenarios.

Thus, for the embedded signal processing applications that we are focussing on, we argue that caches do not have a significant role to play, and we claim that the OT approach discussed previously provides a cost effective solution for minimizing IPC overhead in implementing self-timed schedules.

### 3.2.1 Using the OT approach

The OT strategy, we recall, operates on the principle of determining (at compile time) the order in which processor communications occur, and enforcing that order at run time. For a shared bus implementation, this translates into determining the sequence of shared memory (or, equivalently, shared bus) accesses at compile time and enforcing this predetermined order at run time. This strategy, therefore, involves no run time arbitration; processors are simply granted the bus according to the pre-determined access order. When a processor obtains access to the bus, it performs the necessary shared memory transaction, and releases the bus; the bus is then granted to the next processor in the ordered list.

The task of maintaining ordered access to shared memory is done by a central *ordered transaction controller*. When the processors are downloaded with code, the controller too is loaded with the pre-determined access order list. At run time the controller simply grants bus access to processors according to this list, granting access to the next processor in the list when the current bus owner releases the bus. Such a mechanism is robust with respect to variations in execution times of the actors; the functionality of the system is unaffected by poor estimates of these execution times, although the real-time performance obviously suffers.

If we are able to perform accurate compile time analysis, then each processor would obtain access to the shared bus whenever it needed it. No arbitration needs to be done since there is no contention for the bus. In addition, no semaphore synchronization needs to be performed, because the transaction ordering constraints respect data precedences in the algorithm; when a processor accesses a shared memory location and is correspondingly allowed access to it, the data accessed by that processor is certain to be valid. As a result, in the ideal scenario, a shared bus access takes no more than a single read or write cycle on the processor, and the overall cost of communicating one data sample is two or three instruction cycles.

The performance of this scheme depends on how accurately the execution times of the actors are known at compile time. If these compile time estimates are reasonably accurate, then an access order can be obtained such that a processor gains access to shared memory whenever it needs. Otherwise, a processor may have to idle until it gets a bus grant, or, even worse, a processor when granted the bus may not complete its transaction immediately, thus blocking all other processors from accessing the bus. This problem would not arise in normal arbitration schemes, because independent shared memory accesses would be dynamically reordered.

We will quantify these performance issues in the next chapter, where we show that when reasonably good estimates of actor execution times are available, forcing a run time access order does not in fact sacrifice performance significantly.

## 3.3 Design of an Ordered Memory Access multiproces-

## sor

### 3.3.1 High level design description

We chose Motorola DSP96002 processors for the OMA prototype. Although the OMA architecture can be built around any programmable DSP that has built-in bus arbitration logic, the DSP96002 is particularly suited to our design because of its dual bus architecture and bus arbitration mechanism. In addition these processors are powerful DSPs with floating point capability [Moto89].

A high level block diagram of such a system is depicted in Fig. 3.2. Each DSP96002 is provided with a private memory that contains its program code; this local memory resides on one of the processor busses (the “A” bus). The alternate “B” bus of all processors are connected to the shared bus, and shared memory resides on the shared bus. The transaction controller grants access to processors using the bus grant (BG) lines on the processor. A processor attempts to perform a shared memory access when it executes a communication actor (either send or receive). If its BG line is asserted it performs the access, otherwise it stalls and waits for the assertion.

After a processor obtains access to the shared bus, it performs the shared memory operation and releases the bus. The transaction controller detects the release of the bus and steps through its ordered list, granting the bus to the next processor in its list.

The cost of transfer of one word of data between processors is 3 instruction cycles in the ideal case; two of these correspond to a shared memory write (by the sender) and a shared memory read (by the receiver), and an extra instruction cycle is expended in bus release by the sender and bus acquisition by the receiver.

Thus for the example of Chapter 1, less than 1% of the available 380 instructions per sample are required per transaction. This is of course in the ideal scenario where the sender and the receiver obtain access to the shared bus upon request. Such low overhead interprocessor communication is obtained with the

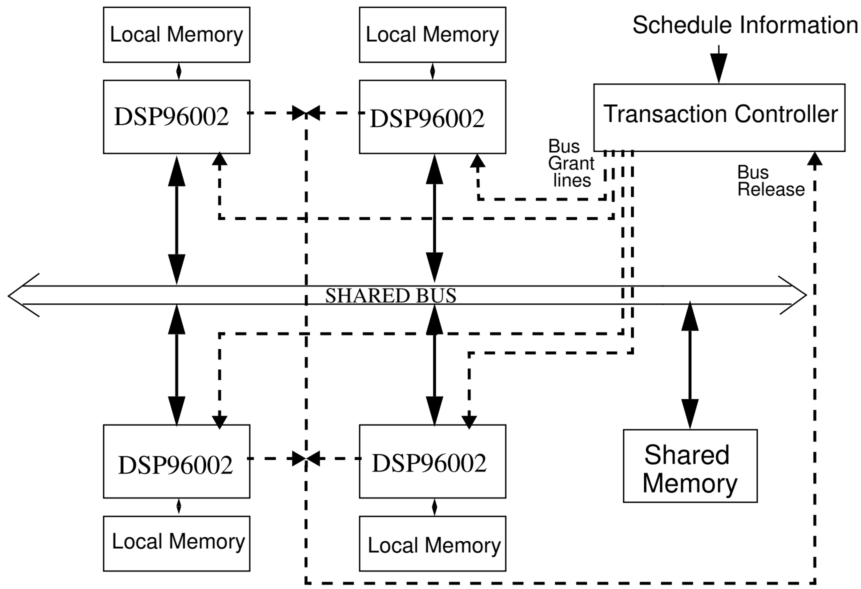



Figure 3.2. Block diagram of the OMA prototype

transaction controller providing the only additional hardware support. As described in a subsequent section, this controller can be implemented with very simple hardware.

### 3.3.2 A modified design

In the design proposed above, processor to processor communication occurs through a central shared memory; two transactions — one write and one read — must occur over the shared bus. This situation can be improved by distributing the shared memory among processors, as shown in Fig. 3.3, where each processor is assigned shared memory in the form of hardware FIFO buffers. Writes to each FIFO are accomplished through the shared bus; the sender simply writes to the FIFO of the processor to which it wants to send data by using the appropriate shared memory address.

Use of a FIFO implies that the receiver must know the exact order in which data is written into its input queue. This, however, is guaranteed by the ordered

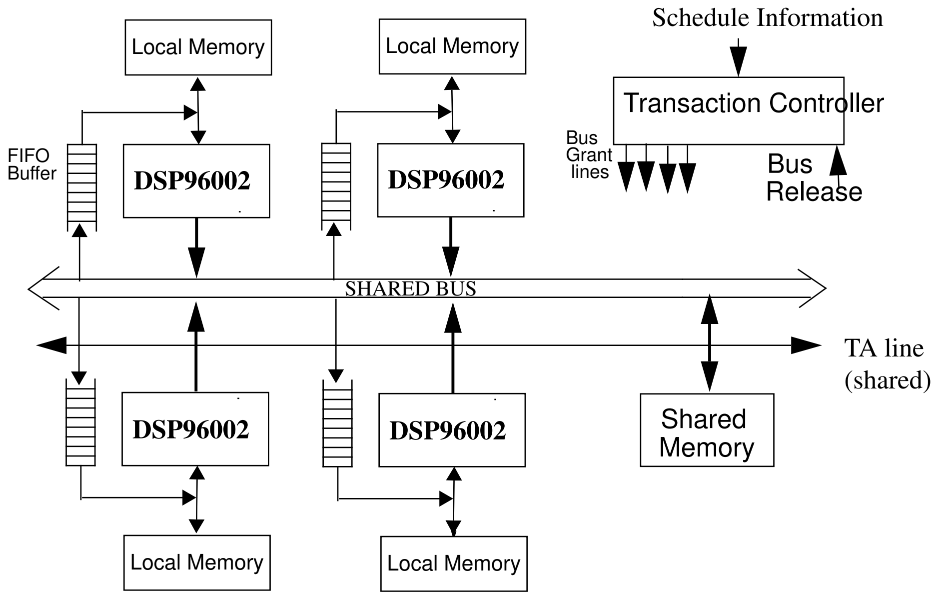



Figure 3.3. Modified design

transaction strategy. Thus replacing a RAM (random access memory) based shared memory with distributed FIFOs does not alter the functionality of the design. The sender need only block when the receiving queue is full, which can be accomplished in hardware by using the 'Transfer Acknowledge (TA)' signal on the DSP96002; a device can insert arbitrary number of wait states in the processor memory cycle by de-asserting the TA line. Whenever a particular FIFO is accessed, its 'Buffer Full' line is enabled onto the TA line of the processors (Fig. 3.4). Thus a full FIFO automatically blocks the processor trying to write into it, and no polling needs to be done by the sender. Receiver read is local to a processor, and does not consume shared bus bandwidth. The receiver can be made to either poll the FIFO empty line to check for an empty queue, or we one can use the same TA signal mechanism to block processor reads from an empty queue. The TA mechanism will then use the local ("A") bus control signals ("A" bus TA signal, "A" address bus etc.). This is illustrated in Fig. 3.4.

Use of such a distributed shared memory mechanism has several advantages. First, the shared bus traffic is effectively halved, because only writes need to go through the shared bus. Second, in the design of Fig. 3.2, a processor that is granted the bus is delayed in completing its shared memory access, all other processors waiting for the bus get stalled; this does not happen for half the transactions in the modified design of Fig. 3.3 because receiver reads are local. Thus there is more tolerance to variations in the time at which a receiver reads data sent to it. Last, a processor can broadcast data to all (or any subset) of processors in the system by simultaneously writing to more than one FIFO buffer. Such broadcast is not possible with a central shared memory.

The modified design, however, involves a significantly higher hardware cost than the design proposed earlier. As a result, the OMA prototype was built around the central shared memory design and not the FIFO based design. In addition, the DSP96002 processor has an on-chip host interface unit that can be used as

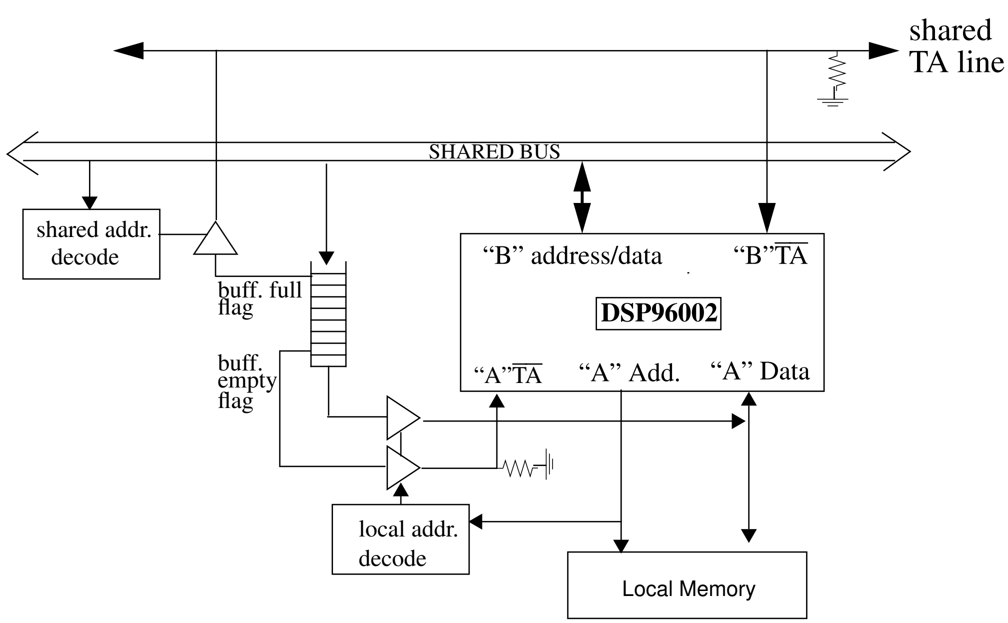



Figure 3.4. Details of the "TA" line mechanism (only one processor is shown).

a 2-deep FIFO; therefore, the potential advantage of using distributed FIFOs can still be evaluated to some degree by using the chip host interface even in the absence of external FIFO hardware.

Simulation models were written for both the above designs using the Thor hardware simulator [Thor86] under the Frigg multi-processor simulator system [Bier89]. Frigg allows the Thor simulator to communicate with a timing-driven functional simulator for the DSP96002 processor provided by Motorola Inc. The Motorola simulator also simulates Input/Output (I/O) operations of the pins of the processor, and Frigg interfaces the signals on the pins to with the rest of the Thor simulation; as a result, hardware associated with each processor (memories, address decoding logic, etc.) and interaction between processors can be simulated using Frigg. This allows functionality of the entire system to be verified by running actual programs on the processor simulators. We did not use this model for performance evaluation, however, because with just a four processor system the cycle-by-cycle Frigg simulation was far too slow, even for very simple programs. A higher-level (behavioral) simulation would be more useful than a cycle-by-cycle simulation for the purposes of performance evaluation, although we did not attempt such high-level simulation of our designs.

The remainder of this chapter describes hardware and software design details of an OMA board prototype.

## 3.4 Design details of a prototype

A proof-of-concept prototype of the OMA architecture has been designed and implemented. The single printed circuit board design is comprised of four DSP96002 processors; the transaction controller is implemented on a Xilinx FPGA (Field Programmable Gate Array). The Xilinx chip also handles the host interface functions, and implements a simple I/O mechanism. A hierarchical description of the hardware design follows.

### 3.4.1 Top level design

This section refers to Fig. 3.5. At the top level, there are four "processing element" blocks that consist of the processor, local memory, local address decoder, and some glue logic. Address, data, and control busses from the PE blocks are connected to form the shared bus. Shared memory is connected to this bus; address decoding is done by the "shared address decoder" PAL (programmable array logic) chip. A central clock generator provides a common clock signal to all processing elements.

A Xilinx FPGA (XC3090) implements the transaction controller and a simple I/O mechanism, and is also used to implement latches and buffers during bootup, thus saving glue logic. A fast static RAM (up to 32K x 8) stores the bus access order in the form of processor identifications (IDs). The sequence of processor IDs is stored in this "schedule RAM", and this determines the bus access order. An external latch is used to store the processor ID read from the schedule RAM. This ID is then decoded to obtain the processor bus grants.

A subset of the 32 address lines connect to the Xilinx chip, for addressing the I/O registers and other internal registers. All 32 lines from the shared data bus are connected to the Xilinx. The shared data bus can be accessed from the external connector (the "right side" connector in Fig. 3.5) only through the Xilinx chip. This feature can be made use of when connecting multiple OMA boards: shared busses from different boards can be made into one contiguous bus, or they can be left disconnected, with communication between busses occurring via asynchronous "bridges" implemented on the Xilinx FPGAs. We discuss this further in section 3.4.7.

Connectors on both ends of the board bring out the shared bus in its entirety. Both left and right side connectors follow the same format, so that multiple boards can be easily connected together. Shared control and address busses are

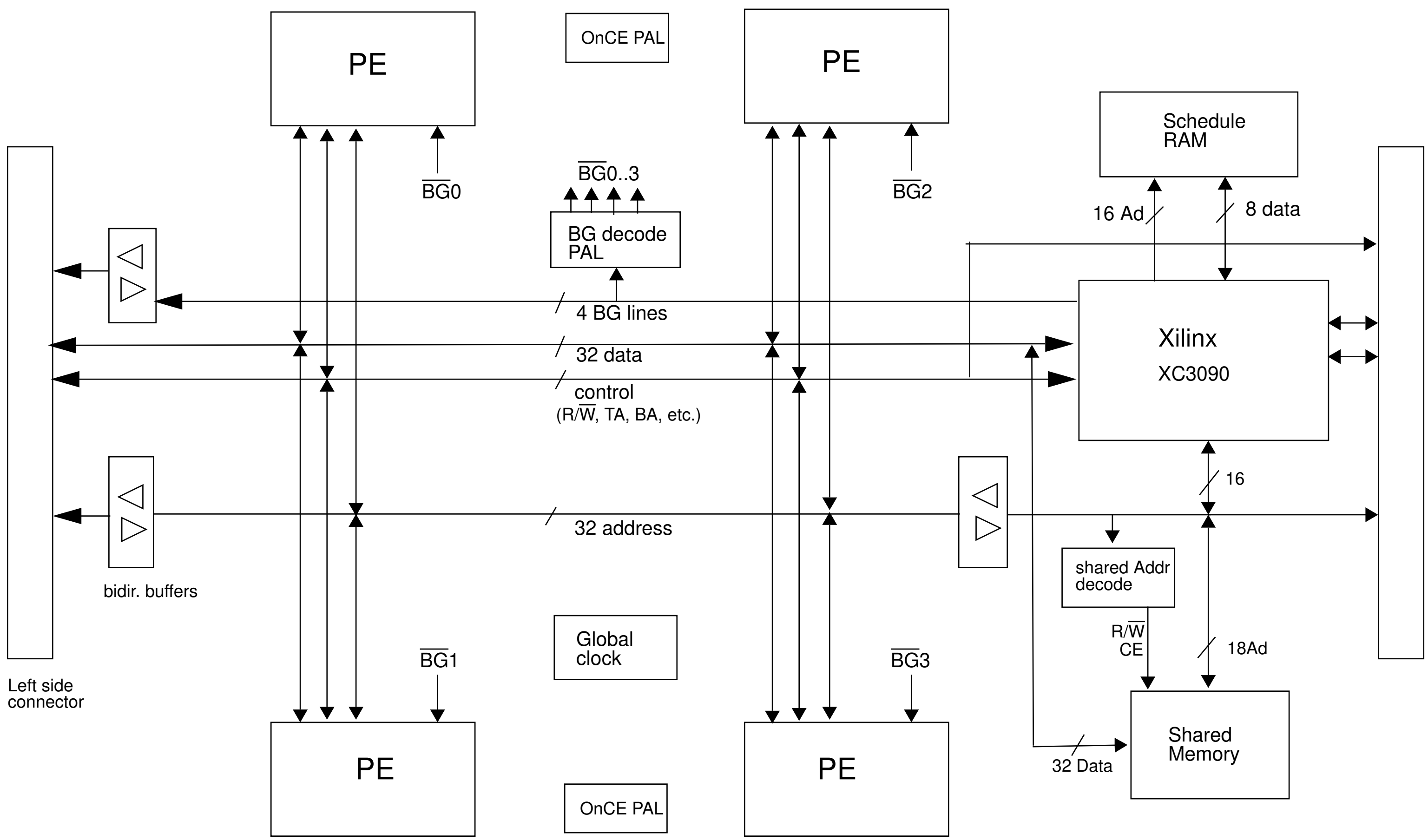



Figure 3.5. Top-level schematic of the OMA prototype

buffered before they go off board via the connectors, and the shared data bus is buffered within the Xilinx.

The DSP96000 processors have on chip emulation ("OnCE" in Motorola terminology) circuitry for debugging purposes, whereby a serial interface to the OnCE port of a processor can be used for in-circuit debugging. On the OMA board, the OnCE ports of the four processors are multiplexed and brought out as a single serial port; a host may select any one of the four OnCE ports and communicate to it through a serial interface.

We discuss the design details of the individual components of the prototype system next.

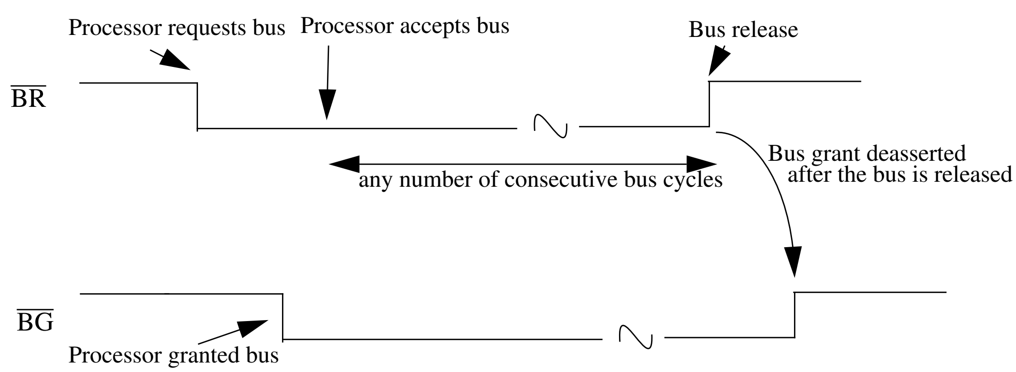
### 3.4.2 Transaction order controller

The task of the transaction order controller is to enforce the predetermined bus access order at run time. A given transaction order determines the sequence of processor bus accesses that must be enforced at run time. We refer to this sequence of bus accesses by the term **bus access order list**. Since the bus access order list is program dependent, the controller must possess memory into which this list is downloaded after the scheduling and code generation steps are completed, and when the transaction order that needs to be enforced is determined. The controller must step through the access order list, and must loop back to the first processor ID in the list when it reaches the end. In addition the controller must be designed to effectively use bus arbitration logic present on-chip, to conserve hardware.

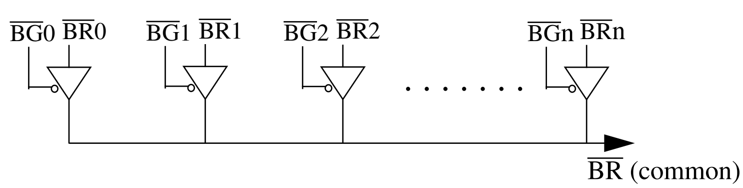
#### 3.4.2.1. Processor bus arbitration signals

We use the bus grant ($\overline{\mathrm{BG}}$) signal on the DSP chip to allow the processor to perform a shared bus access, and we use the bus request ($\overline{\mathrm{BR}}$) signal to tell the controller when a processor completes its shared bus access.

Each of the two ports on the DSP96002 has its own set of arbitration signals; the $\overline{\mathrm{BG}}$ and $\overline{\mathrm{BR}}$ signals are most relevant for our purposes, and these signals are relevant only for the processor port connected to the shared bus. As the name suggests, the $\overline{BG}$ line (which is an input to the processor) must be asserted before a processor can begin a bus cycle: the processor is forced to wait for $\overline{BG}$ to be asserted before it can proceed with the instruction that requires access to the bus. Whenever an external bus cycle needs to be performed, a processor asserts its $\overline{BR}$ signal, and this signal remains asserted until an instruction that does not access the shared bus is executed. We can therefore use the $\overline{BR}$ signal to determine when a shared bus owner has completed its usage of the shared bus (Fig. 3.6 (a)).




(a)




(b)

Figure 3.6. Using processor bus arbitration signals for controlling bus access

The rising edge of the $\overline{BR}$ line is used to detect when a processor releases the bus. To reduce the number of signals going from the processors to the controller, we multiplexed the $\overline{\mathrm{BR}}$ signals from all processors onto a common $\overline{\mathrm{BR}}$ signal. The current bus owner has its $\overline{\mathrm{BR}}$ output enabled onto this common reverse signal; this provides sufficient information to the controller because the controller only needs to observe the $\overline{\mathrm{BR}}$ line from the current bus owner. This arrangement is shown in Fig. 3.6 (b); the controller grants access to a processor by asserting the corresponding $\overline{\mathrm{BG}}$ line, and then it waits for an upper edge on the reverse $\overline{\mathrm{BR}}$ line. On receiving a positive going edge on this line it grants the bus to the next processor in its list.

#### 3.4.2.2. A simple implementation

One straightforward implementation of the above functionality is to use a a counter addressing a RAM that stores the access order list in the form of processor IDs. We call this counter the *schedule counter* and the memory that stores the processor IDs is called the *schedule RAM*. Decoding the output of the RAM provides the required $\overline{\mathrm{BG}}$ lines. The counter is incremented at the beginning of a processor transaction by the negative going edge of the common $\overline{\mathrm{BR}}$ signal and the output of the RAM is latched at the positive going edge of $\overline{\mathrm{BR}}$, thus granting the bus to the next processor as soon as the current processor completes its shared memory transaction. The counter is reset to zero after it reaches the end of the list (i.e. the counter counts modulo the bus access list size). This is shown in Fig. 3.7. Incrementing the counter as soon as $\overline{\mathrm{BR}}$ goes low ensures enough time for the counter outputs and the RAM outputs to stabilize. For a 33MHz processor with zero wait states, $\overline{\mathrm{BR}}$ width is a minimum of 60 nanoseconds. Thus the counter incrementing and the RAM access must both finish before this time. Consequently, we need a fast counter and fast static RAM for the schedule memory. The width of the counter determines the maximum allowable size of the access list (a counter width of size $n$ implies a maximum list size of $2^n$); a wider counter, however, implies a slower counter. If, for a certain width, the counter (implemented on the Xilinx part in our case) turns out to be too slow — i.e. the output of the schedule memory will not stabilize at least one latch set up period before the positive going edge of $\overline{\mathrm{BR}}$

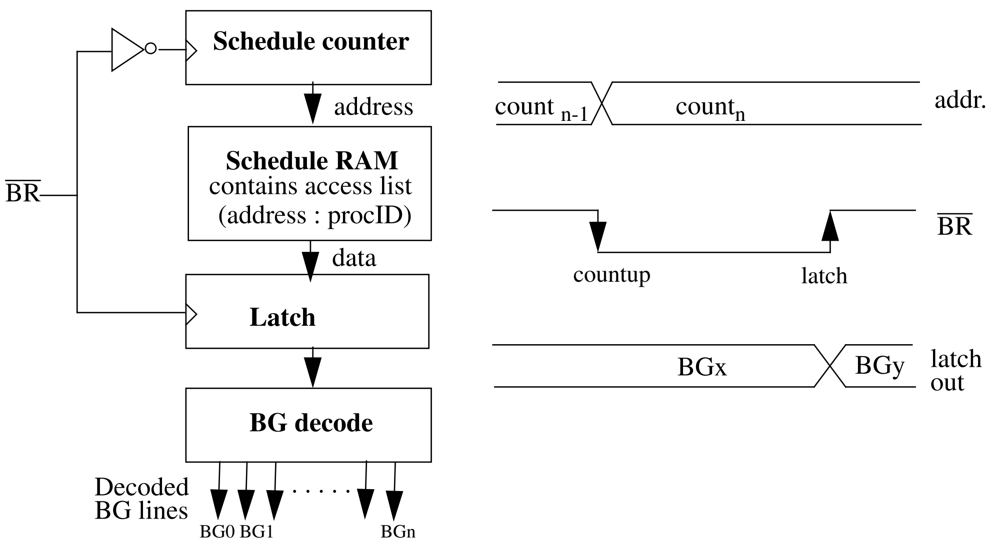



Figure 3.7. Ordered Transaction Controller implementation

arrives — wait states may have to be inserted in the processor bus cycle to delay the positive edge of $\overline{BR}$. We found that a 10 bit wide counter does not require any wait states, and allows a maximum of 1024 processor IDs in the access order list.

### 3.4.2.3. Presettable counter

A single bus access list implies we can only enforce one bus access pattern at run time. In order to allow for some run time flexibility we have implemented the OMA controller using a presettable counter. The processor that currently owns the bus can preset this counter by writing to a certain shared memory location. This causes the controller to jump to another location in the schedule memory, allowing the multiple bus access schedules to be maintained in the schedule RAM and switching between them at run time depending on the outcome of computations in the program. The counter appears as an address in the shared memory map of the processors. The presettable counter mechanism is shown in Fig. 3.8.

An arbitrary number of lists may, in principle, be maintained in the schedule memory. This feature can be used to support algorithms that display data

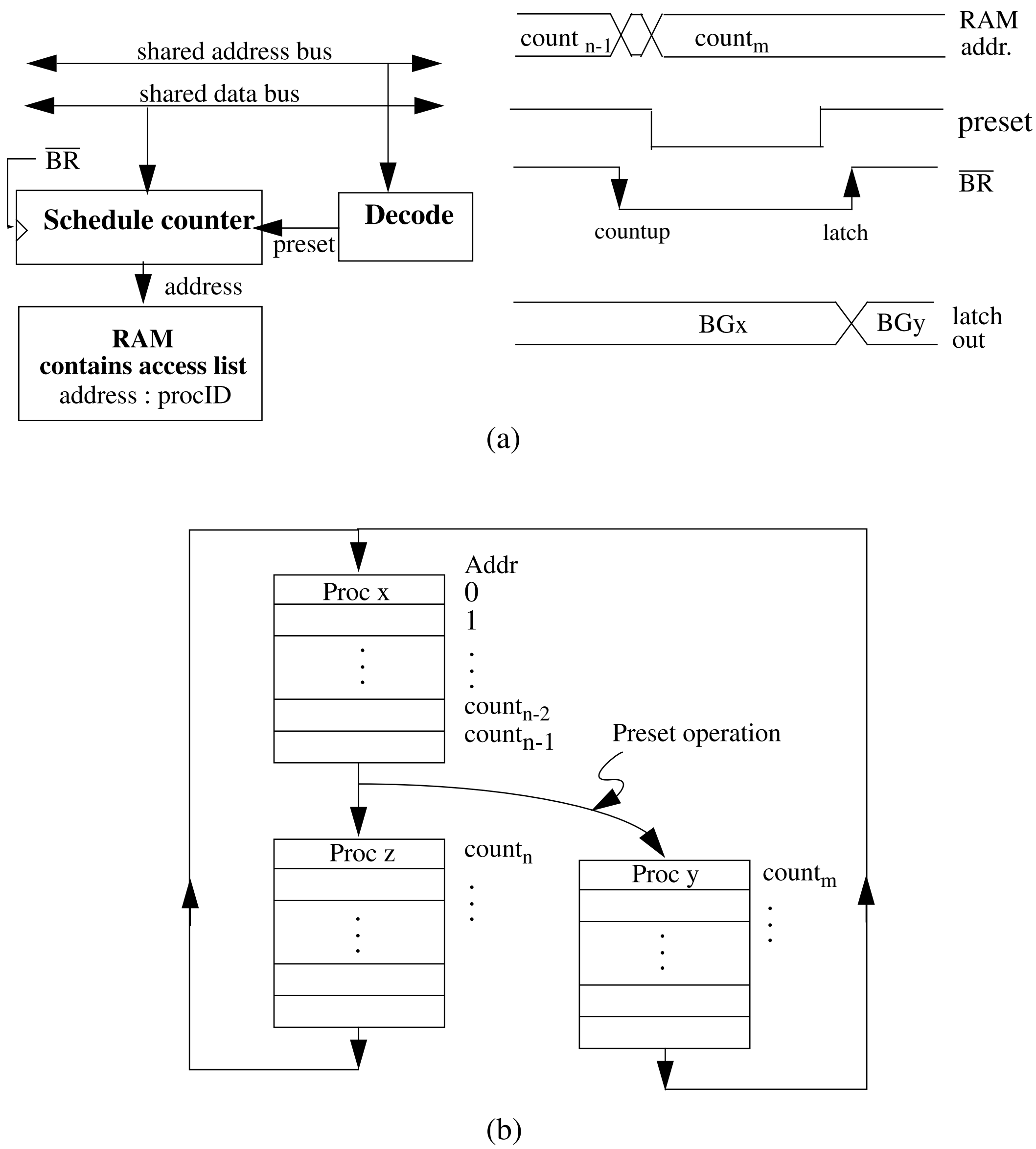



Figure 3.8. Presettable counter implementation

dependency in their execution. For example, a dataflow graph with a conditional construct will, in general, require a different access schedule for each outcome of the conditional. Two different SDF graphs are executed in this case, depending on the branch outcome, and the processor that determines the branch outcome can also be assigned the task of presetting the counter, making it branch to the access list of the appropriate SDF subgraph. The access controller behaves as in Fig 3.8 (b).

We discuss the use of this presettable feature in detail in Chapter 6.

### 3.4.3 Host interface

The function of the host interface is to allow downloading programs onto the OMA board, controlling the board, setting parameters of the application being run, and debugging from a host workstation. The host for the OMA board connects to the shared bus through the Xilinx chip, via one of the shared bus connectors. Since part of the host interface is configured inside the Xilinx, different hosts (32 bit, 16 bit) with different handshake mechanisms can be used with the board.

The host that is being used for the prototype is a DSP56000 based DSP board called the S-56X card, manufactured by Ariel Corp [Ariel91]. The S-56X card is designed to fit into one of the Sbus slots in a Sun Sparc workstation; a user level process can communicate with the S-56X card via a unix device driver. Thus the OMA board too can be controlled (via the S-56X card) by a user process running on the workstation. The host interface configuration is depicted in Fig. 3.9.

Unlike the DSP56000 processors, the DSP96002 processors do not have built in serial ports, so the S-56X board is also is used as a serial I/O processor for the OMA board. It essentially performs serial to parallel conversion of data, buffering of data, and interrupt management. The Xilinx on the OMA board implements the necessary transmit & receive registers, and synchronization flags — we discuss the details of the Xilinx circuitry shortly.

The S-56X card communicates with the Sparc Sbus using DMA (direct memory access). A part of the DSP56000 bus and control signals are brought out of the S-56X card through another Xilinx FPGA (XC3040) on the S-56X. For the purpose of interfacing the S-56X board with the OMA board, the Xilinx on the S-56X card is configured to bring out 16 bits of data and 5 bits of address from the

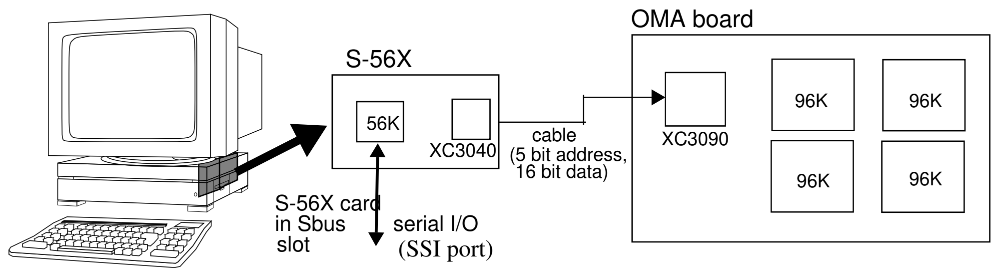



Figure 3.9. Host interface

DSP56000 processor onto the cable connected to the OMA (see Fig. 3.9). In addition, the serial I/O port (the SSI port) is also brought out, for interface with I/O devices such as A/D and D/A convertors. By making the DSP56000 write to appropriate memory locations, the 5 bits of address and 16 bits of data going into the OMA may be set and strobed for a read or a write, to or from the OMA board. In other words, the OMA board occupies certain locations in the DSP56000 memory map; host communication is done by reading and writing to these memory locations.

### 3.4.4 Processing element

Each processing element (PE) consists of a DSP96002 processor, local memory, address buffers, local address decoder, and some address decoding logic. The circuitry of each processing element is very similar to the design of the Motorola 96000 ADS (Application Development System) board [Moto90]. The local address, control, and data busses are brought out into a 96 pin euro-connector, following the format of the 96ADS. This connector can be used for local memory expansion; we have used it for providing local I/O interface to the processing element (as an alternative to using the shared bus for I/O). Port A of the processor forms the local bus, connecting to local memory and address decoding PAL. Each PE also contains address buffers, and logic to set up the bootup mode upon reset and powerup. Port B of the processor is connected to the shared bus.

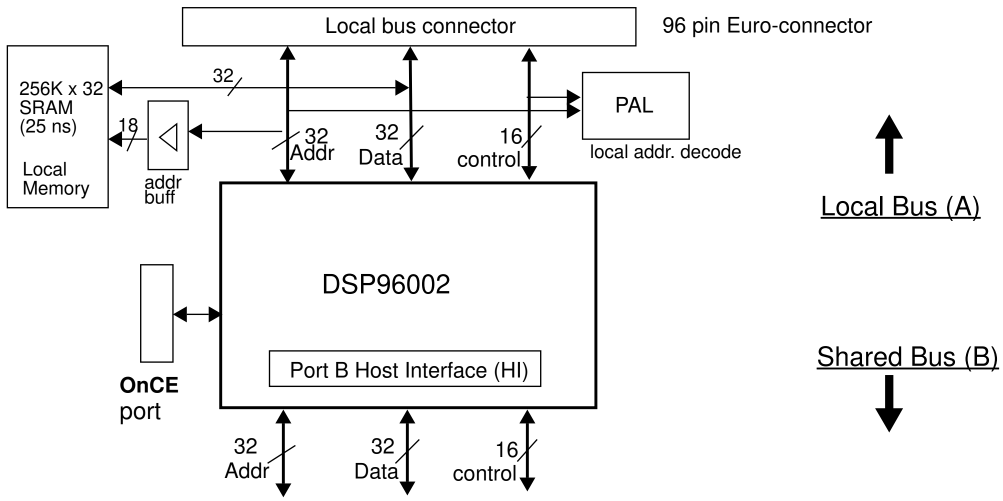



Figure 3.10. Processing element

The DSP96002 processor has a Host Interface (HI) on each of its ports. The port B HI is memory mapped to the shared bus, so that HI registers may be read and written from the shared bus. This feature allows a host to download code and control information into each processor through the shared bus. Furthermore a processor, when granted the shared bus, may also access the port B HI of other processors. This allows processors to bypass the shared memory while communicating with one another and to broadcast data to all processors. In effect, the HI on each processor can be used as a two-deep local FIFO, similar to the scheme in section 3.3.2, except that the FIFO is internal to each processor.

### 3.4.5 Xilinx circuitry

As mentioned previously, the XC3090 Xilinx chip is used to implement the transaction controller as well as a simple I/O interface. It is also configured to provide latches and buffers for addressing the Host Interface (HI) ports on the DSP96002 during bootup and downloading of code onto the processors. For this to work, the Xilinx is first configured to implement the bootup and download related circuitry, which consists of latches to drive the shared address bus and to access the schedule memory. After downloading code onto the processors, and downloading the bus access order into the schedule RAM, the Xilinx chip is reconfigured to implement the ordered transaction controller and the I/O interface. Thus the process of downloading and running a program requires configuring the Xilinx chip twice.

There are several possible ways in which a Xilinx part may be programmed. For the OMA board, the configuration bitmap is downloaded byte-wise by the host (Sun workstation through the S-56X card). The bitmap file, generated and stored as a binary file on a workstation, is read in by a function implemented in the *qdm* software (discussed in section 3.5, which describes the OMA software interface) and the bytes thus read are written into the appropriate memory location on the S-56X card. The DSP56K processor on the S-56X then strobes these bytes into the Xilinx configuration port on the OMA board. The user can reset and reconfigure the Xilinx chip from a Sun Sparc workstation by manipulating the Xilinx control pins by writing to a "Xilinx configuration latch" on the OMA board. Various configuration pins of the Xilinx chip are manipulated by writing different values into this latch.

We use two different Xilinx circuits (`bootup.bit` and `moma1.bit`), one during bootup and the other during run time. The Xilinx configuration during bootup helps eliminate some glue logic that would otherwise be required to latch and decode address and data from the S-56X host. This configuration allows the host to read and write from any of the HI ports of the processors, and also to access the schedule memory and the shared memory on board.

Run time configuration on the Xilinx consists of the transaction controller implemented as a presettable counter. The counter can be preset through the shared bus. It addresses an external fast RAM (8 nanosecond access time) that contains processor IDs corresponding to the bus access schedule. Output from the schedule memory is externally latched and decoded to yield bus grant ($\overline{BG}$) lines (Fig. 3.7).

A schematic of the Xilinx configuration at run time is given in Fig. 3.11. This configuration is for I/O with an S-56X (16 bit data) host, although it can easily be modified to work with a 32 bit host.

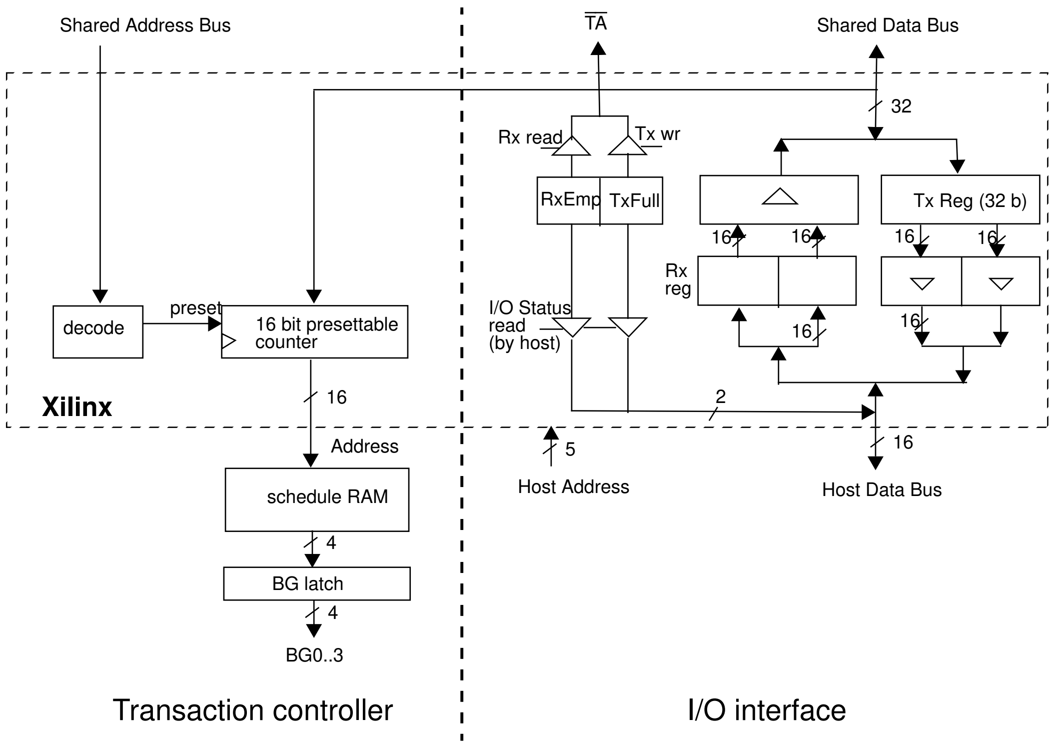



Figure 3.11. Xilinx configuration at run time

### 3.4.5.1. I/O interface

The S-56X board reads data from the Transmit (Tx) register and writes into the receive (Rx) register on the Xilinx. These registers are memory-mapped to the shared bus, such that any processor that possesses the bus may write to the Tx register or read from the Rx register. For a 16 bit host, two transactions are required to perform a read or write with the 32 bit Tx and Rx registers. The processors themselves need only one bus access to load or unload data from the I/O interface. Synchronization on the S-56X (host) side is done by polling status bits that indicate an Rx empty flag (if true, the host performs a write, otherwise it busy waits) and a Tx full flag (if true, the host performs a read, otherwise it busy waits). On the OMA side, synchronization is done by the use of the TA (transfer acknowledge) pin on the processors. When a processor attempts to read Rx or write Tx, the appropriate status flags are enabled onto the TA line, and wait states are automatically inserted in the processor bus cycle whenever the TA line is not asserted, which in our implementation translates to wait states whenever the status flags are false. Thus processors do not have the overhead of polling the I/O status flags; an I/O transaction is identical to a normal bus access, with zero or more wait states inserted automatically.

The DSP56000 processor on the S-56X card is responsible for performing I/O with the actual (possibly asynchronous) data source and acts as the interrupt processor for the OMA board, relieving the board of tasks such as interrupt servicing and data buffering. This of course has the downside that the S-56X host needs to be dedicated to "spoon-feeding" the OMA processor board, and limits other tasks that could potentially run on the host.

### 3.4.6 Shared memory

Space for two shared memory modules are provided, so that up to 512K x 32 bits of shared static RAM can reside on board. The memory must have an access time of 25ns to achieve zero wait state operation.

### 3.4.7 Connecting multiple boards

Several features have been included in the design to facilitate connecting together multiple OMA boards. The connectors on either end of the shared bus are compatible, so that boards may be connected together in a linear fashion (Fig. 3.12). As mentioned before, the shared data bus goes to the "right side connector" through the Xilinx chip. By configuring the Xilinx to "short" the external and internal shared data busses, processors on different boards can be made to share one contiguous bus. Alternatively, busses can be "cleaved" on the Xilinx chip, with communication between busses implemented on the Xilinx via an asynchronous mechanism (e.g. read and write latches synchronized by “full” and “empty” flags).

This concept is similar to the idea used in the SMART processor array [Koh90], where the processing elements are connected to a switchable bus: when the bus switches are open processors are connected only to their neighbors (forming a linear processor array), and when the switched are closed processors are connected onto a contiguous bus. Thus the SMART array allows formation of clusters of processors that reside on a common bus; these clusters then communicate with adjacent clusters. When we connect multiple OMA boards together, we get a similar effect: in the “shorted” configuration processors on different boards connect to a single bus, whereas in the “cleaved” configuration processors on different boards reside on common busses, and neighboring boards communicate through an asynchronous interface.

Fig. 3.12 illustrates the above scheme. The highest 3 bits of the shared address bus are used as the “board ID” field. Memory, processor Host Interface ports, configuration latches etc. decode the board ID field to determine if a shared memory or host access is meant for them. Thus, a total of 8 boards can be hooked onto a common bus in this scheme.

## 3.5 Hardware and software implementation

### 3.5.1 Board design

We used single sided through-hole printed circuit board technology for the OMA prototype. The printed circuit board design was done using the ‘SIERA’ system developed at Professor Brodersen’s group at Berkeley [Sriv92]. Under this system, a design is entered hierarchically using a netlist language called SDL (Structure Description Language). Geometric placement of components can be easily specified in the SDL netlist itself. A ‘tiling’ feature is also provided to ease compact fitting of components. The SDL files were written in a modular fashion;

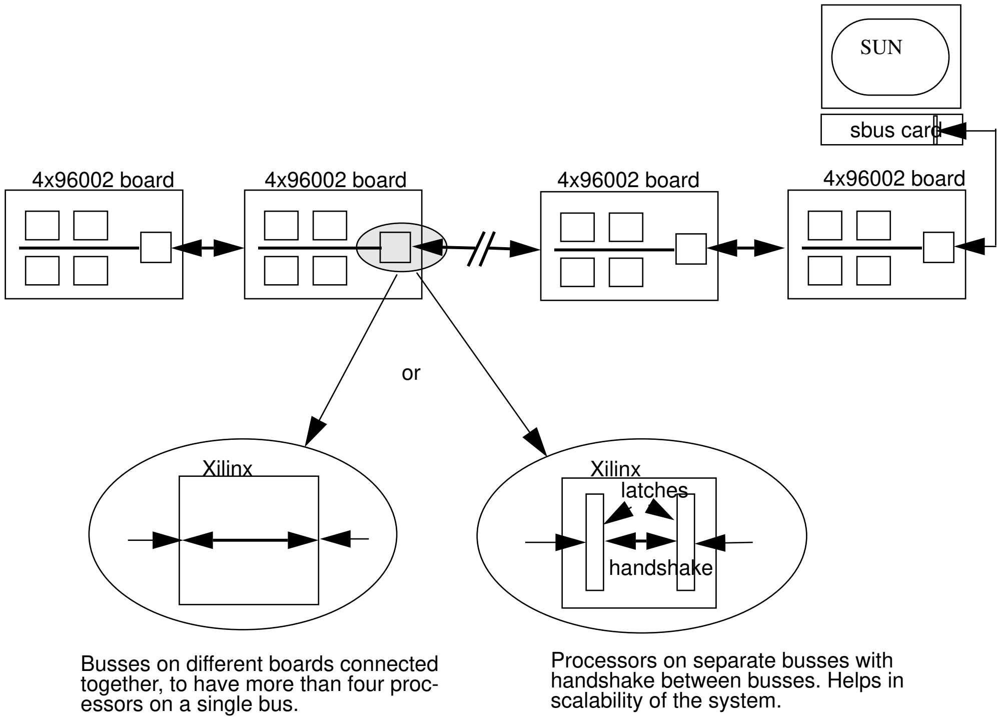



Figure 3.12. Connecting multiple boards

the schematics hierarchy is shown in Fig. 3.13. The SIERA design manager (DMoct) was then used to translate the netlists into an input file acceptable by Racal, a commercial PCB layout tool, which was then used to autoroute the board in ten layers, including one Vcc and one Ground plane. The files corresponding to the traces on each layer (gerber files) were generated using Racal, and these files were then sent to Mosis for board fabrication. Component procurement was mainly done using the FAST service, but some components were ordered/bought directly from electronic component distributors. Table 3.1 lists the salient physical features of the board, and Fig. 3.14 shows a photograph of the board.

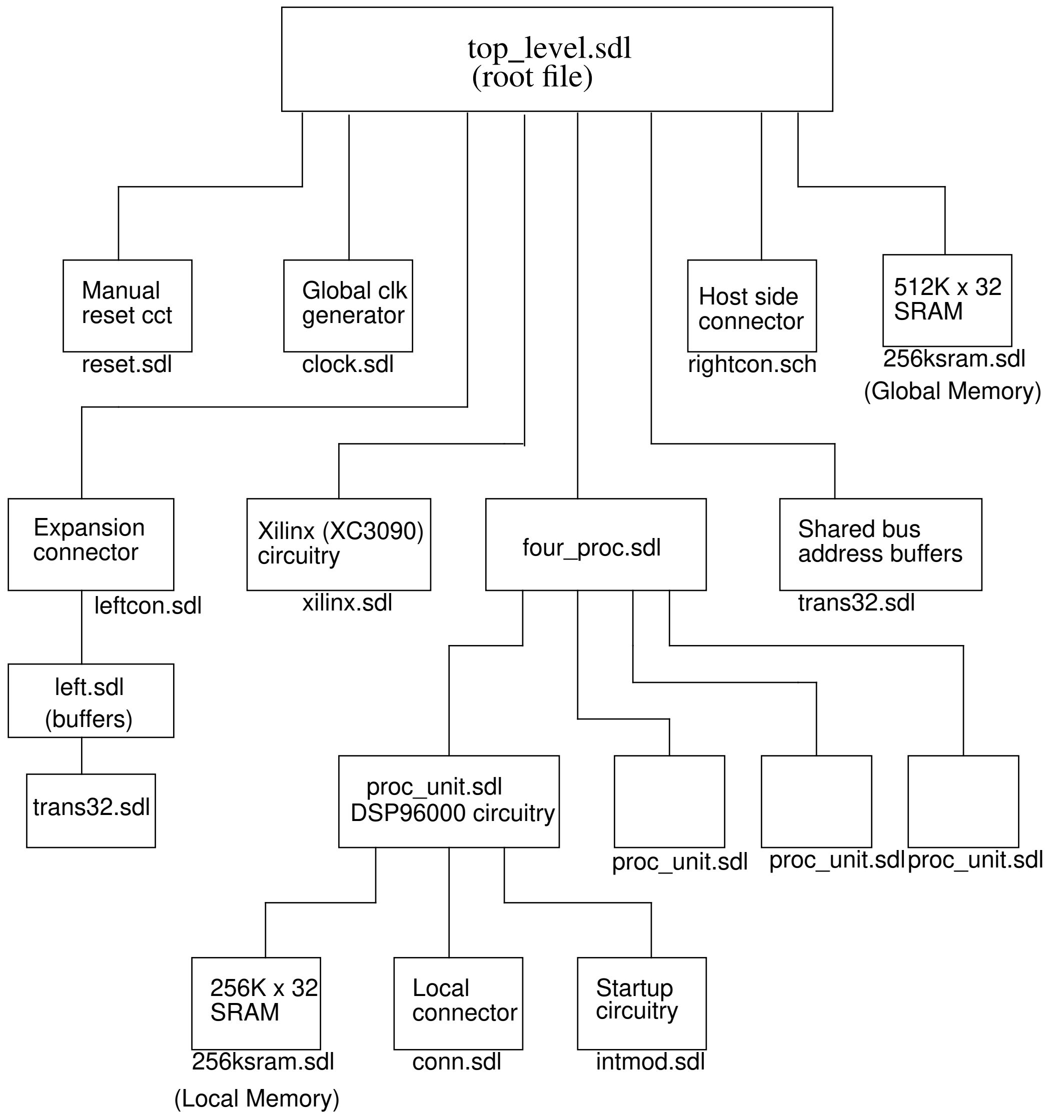



Figure 3.13. Schematics hierarchy of the four processor OMA architecture

Table 3.1. OMA board physical specs

| | |
|---|---|
| Dimensions | 30 cm. x 17 cm. |
| Layers | 10 (including ground and Vcc plane) |
| Number of Components | 230 parts + 170 bypass capacitors |
| *sdl* code | 2800 lines |
| Memory | 512K words shared, 256K words local |

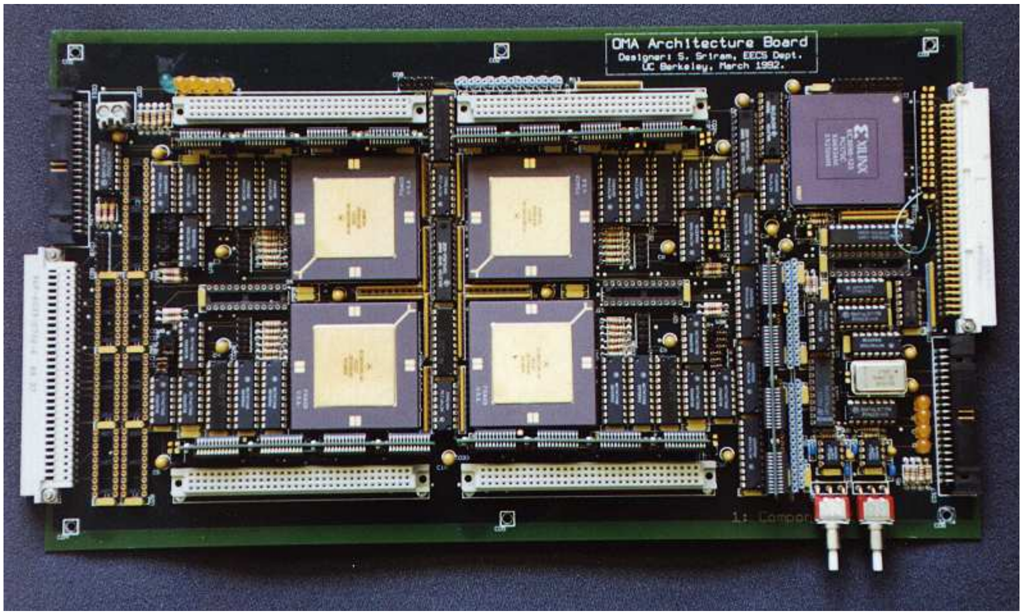



Figure 3.14. OMA prototype board photograph

### 3.5.2 Software interface

As discussed earlier, we use an S-56X card attached to a Sparc as a host for the OMA board. The Xilinx chip on the S-56X card is configured to provide 16 bits of data and 5 bits of address. We use the *qdm* [Laps91] software as an interface for the S-56X board; *qdm* is a debugger/monitor that has several useful built-in routines for controlling the S-56X board, for example data can be written and read from any location in the DSP56000 address space through function calls in *qdm*. Another useful feature of *qdm* is that it uses '*tcl*', an embeddable, extensible, shell-like interpreted command language [Ous94]. Tcl provides a set of built-in functions (such as an expression evaluator, variables, control-flow statements etc.) that can be executed via user commands typed at its textual interface, or from a specified command file. Tcl can be extended with application-specific commands; in our case these commands correspond to the debugging/monitor commands implemented in *qdm* as well as commands specific to the OMA. Another useful feature of tcl is the scripting facility it provides; sequences of commands can be conveniently integrated into scripts, which are in turn executed by issuing a single command.

Some functions specific to the OMA hardware that have been compiled into *qdm* are the following:

**`omaxiload`** *`fileName.bit`* : load OMA Xilinx with configuration specified by file.bit

**`omapboot`** *`fileName.lod proc#`* : load bootstrap monitor code into the specified processor

**`omapload`** *`fileName.lod proc#`* : load DSP96002 .lod file into the specified processor

**`schedload`** *`accessOrder`* : load OMA bus access schedule memory

These functions use existing *qdm* functions for reading and writing values to the DSP56000 memory locations that are mapped to the OMA board host interface. The sequence of commands needed to download code onto the OMA board and run it is summarized in Fig. 3.15.

```
proc omaDoAll {} {
      xload                              # configure S-56X board Xilinx
      omareset                           # reset OMA board
      omaxiload bootup.bit               # configure OMA Xilinx for booting procs
      foreach i {0 1 2 3} {
          omapboot $i omaMon.lod            # load bootstrap monitor routine
          omapload $i $icode.lod            # load code (0code.lod, 1code.lod etc.)
          schedload access_order.data       # load bus access schedule into
      }                                     schedule memory
      omaxiload moma1.bit                # reconfigure OMA Xilinx to implement
                                         Transaction Controller and I/O
      load host.lod; run                 # run and load I/O interrupt routines on S-
                                         56X board
      synch                              # start all processors synchronously
}
```

Figure 3.15. Steps required for downloading code (*tcl* script omaDoAll)

Each processor is programmed through its Host Interface via the shared bus. First, a monitor program (omaMon.lod) consisting of interrupt routines is loaded and run on the selected processor. Code is then loaded into processor memory by writing address and data values into the HI port and interrupting the processor. The interrupt routine on the processor is responsible for inserting data into the specified memory location. The S-56X host forces different interrupt routines, for specifying which of the three (X, Y, or P) memories the address refers to and for specifying a read or a write to or from that location. This scheme is similar to that employed in downloading code onto the S-56X card [Ariel91].

Status and control registers on the OMA board are memory mapped to the S-56X address space and can be accessed to reset, reboot, monitor, and debug the board. Tcl scripts were written to simplify commands that used are most often (e.g. 'change y:fff0 0x0' was aliased to 'omareset'). The entire code downloading procedure is executed by the tcl script 'omaDoAll' (see Fig. 3.15).

A Ptolemy multiprocessor hardware target (Chapter 12, Section 2 in [Ptol94]) was written for the OMA board, for automatic partitioning, code generation, and execution of an algorithm from a block diagram specification. A simple heterogeneous multiprocessor target was also written in Ptolemy for the OMA and S-56X combination; this target generates DSP56000 code for the S-56X card, and generates DSP96000 multiprocessor code for the OMA.

## 3.6 Ordered I/O and parameter control

We have implemented a mechanism whereby I/O can be done over the shared bus. We make use of the fact that I/O for DSP applications is periodic; samples (or blocks of samples) typically arrive at constant, periodic intervals, and the processed output is again required (by, say, a D/A convertor) at periodic intervals. With this observation, it is in fact possible to schedule the I/O operations within the multiprocessor schedule, and consequently determine when, relative to the other shared bus accesses due to IPC, the shared bus is required for I/O. This allows us to include bus accesses for I/O in the bus access order list. In our particular implementation, I/O is implemented as shared address locations that address the Tx and Rx registers in the Xilinx chip (section 3.4.5), which in turn communicate with the S-56X board; a processor accesses these registers as if they were a part of shared memory. It obtains access to these registers when the transaction controller grants access to the shared bus; bus grants for the purpose of I/O are taken into account when constructing the access order list. Thus we order access to shared I/O resources much as we order access to the shared bus and memory.

We also experimented with application of the ordered memory access idea to run time parameter control. By run time parameter control we mean controlling parameters in the DSP algorithm (gain of some component, bit-rate of a coder, pitch of synthesized music sounds, etc.) while the algorithm is running in real time on the hardware. Such a feature is obviously very useful and sometimes indispensable. Usually, one associates such parameter control with an asynchronous user input: the user changes a parameter (ideally by means of a suitable GUI on his or her computer) and this change causes an interrupt to occur on a processor, and the interrupt handler then performs the appropriate operations that cause the parameter change that the user requested.

For the OMA architecture, however, unpredictable interrupts are not desirable, as was noted earlier in this chapter; on the other hand shared I/O and IPC are relatively inexpensive owing to the OT mechanism. To exploit this trade-off, we implemented the parameter control in the following fashion: The S-56X host handles the task of accepting user interrupts; whenever a parameter is altered, the DSP56000 on the S-56X card receives an interrupt and it modifies a particular location in its memory (call it $M$). The OMA board on the other hand receives the contents of $M$ *on every schedule period,* whether $M$ was actually modified or not. Thus the OMA processors never "see" a user created interrupt; they in essence update the parameter corresponding to the value stored in $M$ in *every* iteration of the dataflow graph. Since reading in the value of $M$ costs two instruction cycles, the overhead involved in this scheme is minimal.

An added practical advantage of the above scheme is that the tcl/tk [Ous94] based GUI primitives that have been implemented in Ptolemy for the S-56X (see "CG56 Domain" in Volume 1 of [Ptol94]) can be directly used with the OMA board for parameter control purposes.

## 3.7 Application examples

### 3.7.1 Music synthesis

The Karplus-Strong algorithm is a well known approach for synthesizing the sound of a plucked string. The basic idea is to pass a noise source in a feedback loop containing a delay, a low pass filter, and a multiplier with a gain of less than one. The delay determines the pitch of the generated sound, and the multiplier gain determines the rate of decay. Multiple voices can be generated and combined by implementing one feedback loop for each voice and then adding the outputs from all the loops. If we want to generate sound at a sampling rate of 44.1 KHz (compact disc sampling rate), we can implement 7 voices on a single processor in real time using the blocks from the Ptolemy DSP96000 code generation library (CG96). These 7 voices consume 370 instruction cycles out of the 380 instruction cycles available per sample period.

Using four processors on the OMA board, we implemented 28 voices in real time. The hierarchical block diagram for this is shown in Fig. 3.16. The resulting schedule is shown in Fig. 3.17. The makespan for this schedule is 377 instruction cycles, which is just within the maximum allowable limit of 380. This schedule uses 15 IPCs, and is therefore not communication intensive. Even so, a higher IPC cost than the 3 instruction cycles the OMA architecture affords us would not allow this schedule to execute in real time at a 44.1 KHz sampling rate, because there is only a 3 instruction cycle margin between the makespan of this schedule and the maximum allowable makespan. To schedule this application, we employed Hu-level scheduling along with manual assignment of some of the blocks.

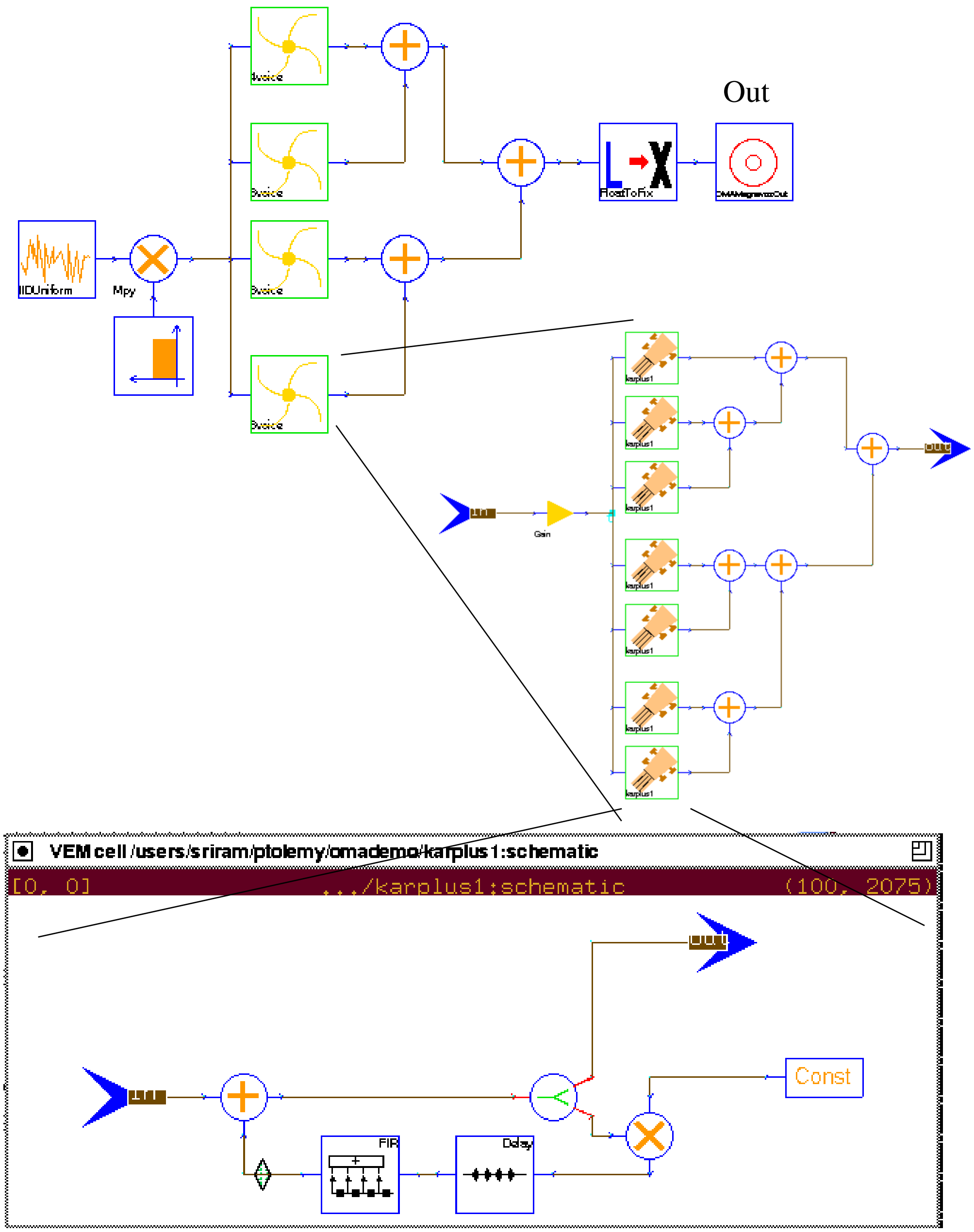



Figure 3.16. Hierarchical specification of the Karplus-Strong algorithm in 28 voices.

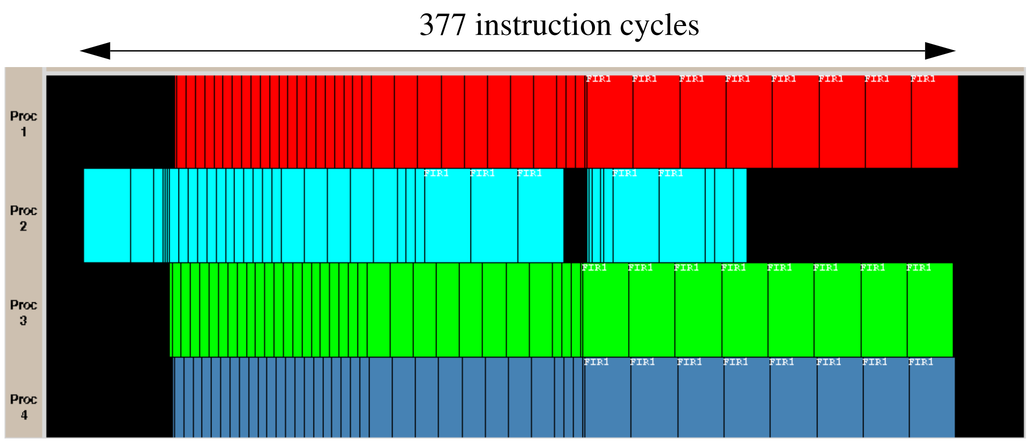


Figure 3.17. Four processor schedule for the Karplus-Strong algorithm in 28 voices. Three processors are assigned 8 voices each, the fourth (Proc 1) is assigned 4 voices along with the noise source.

### 3.7.2 QMF filter bank

A Quadrature Mirror Filter (QMF) bank consists of a set of *analysis* filters used to decompose a signal (usually audio) into frequency bands, and a bank of *synthesis* filters is used to reconstruct the decomposed signal [Vai93]. In the analysis bank, a filter pair is used to decompose the signal into high pass and low pass components, which are then decimated by a factor of two. The low pass component is then decomposed again into low pass and high pass components, and this process proceeds recursively. The synthesis bank performs the complementary operation of upsampling, filtering, and combining the high pass and low pass components; this process is again performed recursively to reconstruct the input signal. Fig. 3.18(a) shows a block diagram of a synthesis filter bank followed by an analysis bank.

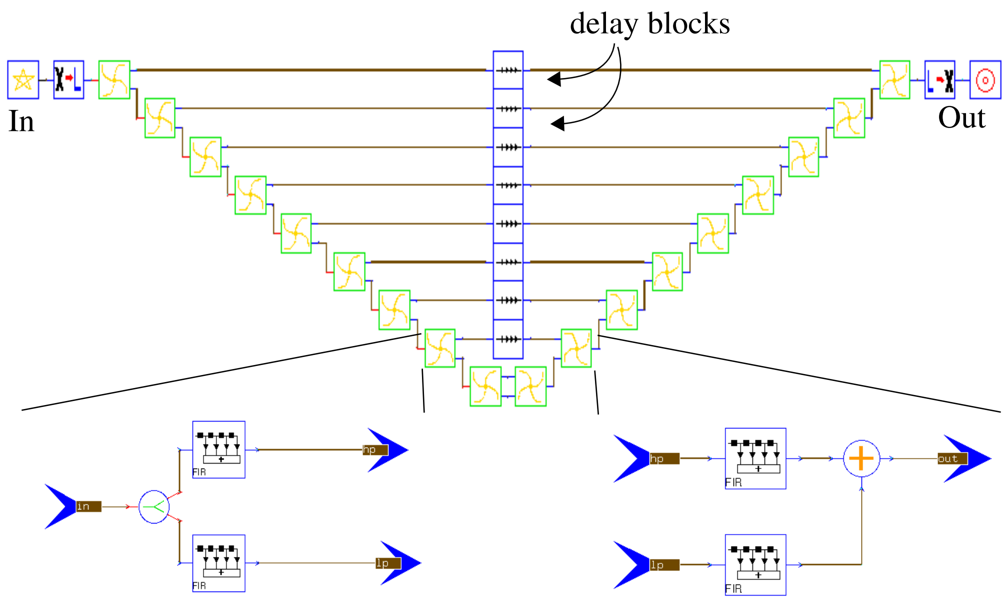
QMF filter banks are designed such that the analysis bank cascaded with the synthesis bank yields a transfer function that is a pure delay (i.e. has unity response except for a delay between the input and the output). Such filter banks are also called *perfect reconstruction* filter banks, and they find applications in high quality audio compression; each frequency band is quantized according to its energy content and its perceptual importance. Such a coding scheme is employed in the audio portion of the MPEG standard.

We implemented a perfect-reconstruction QMF filter bank to decompose audio from a compact disc player into 15 bands. The synthesis bank was implemented together with the analysis part. There are a total of 36 multirate filters of 18 taps each. This is shown hierarchically in Fig. 3.18(a). Note that delay blocks are required in the first 13 output paths of the analysis bank to compensate for the delay through successive stages of the analysis filter bank.

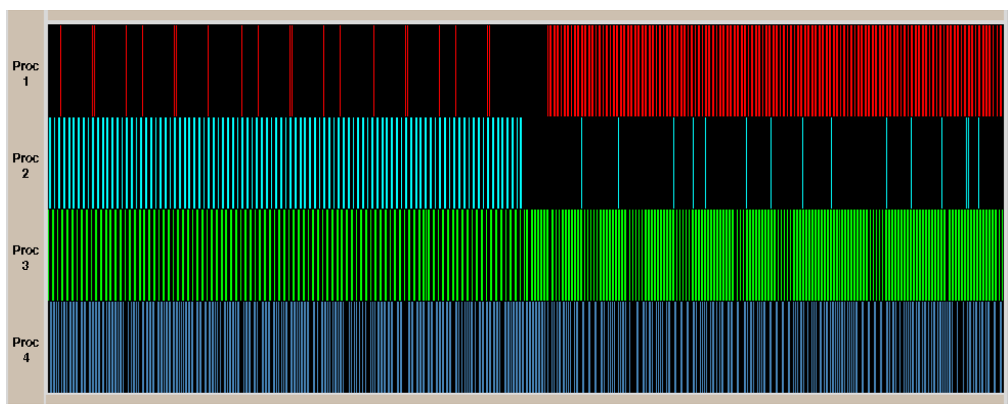
There are 1010 instruction cycles of computation per sample period in this example. Using Sih's Dynamic Level (DL) scheduling heuristic, we were able to achieve an average iteration period of 366 instruction cycles, making use of 40 IPCs. The schedule that is actually constructed (Gantt chart of Fig. 3.18(b)) operates on a block of 512 samples because these many samples are needed before all the actors in the graph fire at least once; this makes manual scheduling very difficult. We found that the DL heuristic performs close to 20% better than the Hu-level heuristic in this example, although the DL heuristic takes more than twice the time to compute the schedule compared to Hu-level.

### 3.7.3 1024 point complex FFT

For this example, input data (1024 complex numbers) is assumed to be present in shared memory, and the transform coefficients are written back to shared memory. A single 96002 processor on the OMA board performs a 1024 point complex FFT in 3.0 milliseconds (ms). For implementing the transform on all four processors, we used the first stage of a radix four, decimation in frequency FFT computation, after which each processor independently performs a 256 point FFT. In this scheme, each processor reads all 1024 complex inputs at the beginning of the computation, combines them into 256 complex numbers on which it performs a 256 point FFT, and then writes back its result to shared memory using bit reversed addressing. The entire operation takes 1.0 ms. Thus we achieve a speedup




(a)




(b)

Figure 3.18. (a) Hierarchical block diagram for a 15 band analysis and synthesis filter bank. (b) Schedule on four processors (using Sih's DL heuristic [Sih90]).

of 3 over a single processor. This example is communication intensive; the throughput is limited by the available bus bandwidth. Indeed, if all processors had independent access to the shared memory (if the shared memory were 4-ported for example), we could achieve an ideal speedup of four, because each 256 point FFT is independent of the others except for data input and output.

For this example, data partitioning, shared memory allocation, scheduling, and tuning the assembly program was done by hand, using the 256 point complex FFT block in the Ptolemy CG96 domain as a building block. The Gantt chart for the hand generated schedule is shown in Fig. 3.19.

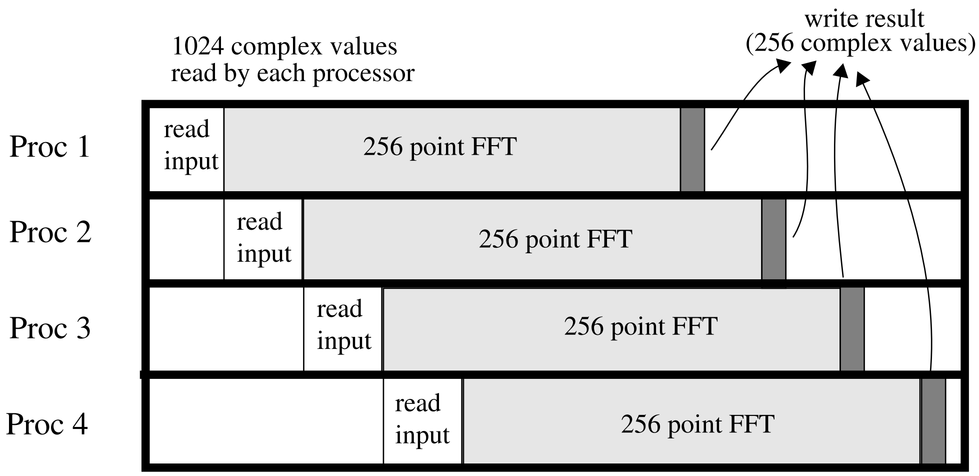



Figure 3.19. Schedule for the FFT example.

## 3.8 Summary

In this chapter we discussed the ideas behind the Ordered Transactions scheduling strategy. This strategy combines compile time analysis of the IPC pattern with simple hardware support to minimize interprocessor communication overhead. We discussed the hardware design and implementation details of a prototype shared bus multiprocessor — the Ordered Memory Access architecture — that uses the ordered transactions principle to statically assign the sequence of processor accesses to shared memory. External I/O and user control inputs can also be taken into account when scheduling accesses to the shared bus. We also discussed the software interface details of the prototype and presented some applications that were implemented on the OMA prototype.