CASE STUDY IN PROTOTYPING WITH RAPIDE: SHARED MEMORY MULTIPROCESSOR SYSTEM

Alexandre Santoro

Technical Report: **CSL-TR-93-564**
(Program Analysis and Verification Group Report No. 61)

March 1993

This research has been supported by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research Grant N00014-92-J-1928 and under TRW, subcontract FZ2394LK1S-04.

Case Study in Prototyping with Rapide: Shared Memory Multiprocessor System

Alexandre Santoro

Technical Report: CSL-TR-93-564
Program Analysis and Verification Group Report No. 61

March 1993

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Rapide is a concurrent object-oriented language designed for prototyping distributed systems. This paper describes the creation of such a prototype, more specifically a shared memory multiprocessor system. The design is presented in an evolutionary manner, starting with a simple CPU + memory model. The paper also presents some simulation results and shows how the partially ordered event sets that Rapide produces can be used both for performance analysis and for an in-depth understanding of the model's behavior.

Key Words and Phrases: prototyping, simulation, event-based modelling, formal constraints, performance measurements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This model is the result of the author's introduction to prototyping with Rapide-0.2. In order to gain some "hands on" experience with the language, it was decided to attempt the modelling of a system that was simple, but not trivial. This would provide a better understanding of Rapide's power and shortcomings, as well as the issues involved in creating a prototype.

It was chosen to model a multiprocessor system in which several processor units, each one consisting of a CPU and a cache, share one bus through which they access the main memory. Not only is this a well understood system, but it also has the advantage of scalability. The prototype could evolve slowly, starting with one CPU and the main memory, then being refined by the addition of a cache, a bus arbiter and finally by replicating the number of processors to create the multiprocessor environment.

The following chapters attempt to describe the evolution of the system, starting from the initial model and explaining how it changed. This will be followed by a performance analysis of the multiprocessor model. Finally, it will present an evaluation of the prototyping experience and comments on Rapide-0.2 and the toolset.

There are two appendixes at the end of this report. The first one contains contains a complete listing of the cpl files used for model creation. It is supposed that the reader is acquainted with the Rapide-0.2 language. If not, reading [Bry92] and [Hsi92] is suggested. The second appendix contains the code for the same system written in μRapide, an architecture definition language.

The author would like to thank David Luckham, Doug Bryan, James Vera, Larry Augustin and Walter Mann for their help and patience in helping the author understand Rapide and create the model.

Chapter 2

The First System

In order to get things up and running as quickly as possible, the first system consisted of only a CPU and main memory, as may be seen in figure 2.1. In it, the big rectangles represent the design units while the small ones with the rounded corners represent the associated actions. The lines indicate connections between events. That is, each pair of rounded rectangles connected by a line corresponds to a connect statement in the Rapide-0.2 code.

The main memory unit performs only two functions, namely read and write. When it receives a *MemRead* event it fetches the data from the address supplied (stored in the array variable **MemBlock**) and returns it through a *ReadEnd* event. When it receives a *MemWrite* event it stores the supplied data in the proper address and returns a *WriteEnd* event, signaling the completion of the task. It has a simple body, described by the following '**when** block':

```
<< memory_read_cycle >>
when MemRead(?address) then
   ReadEnd(MemBlock[?address]);
end when;

<< memory_write_cycle >>
when MemWrite(?address,?data) then
```

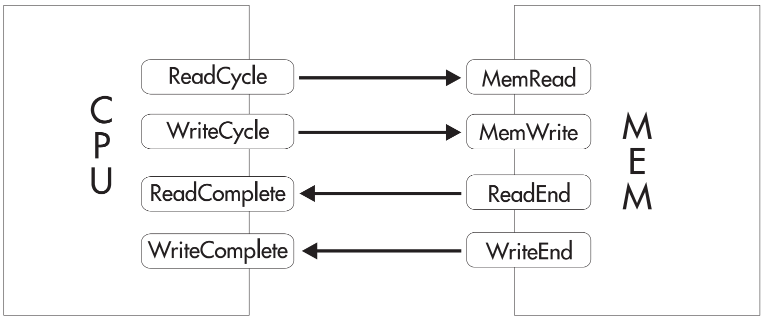

Figure 2.1: Architecture of first system

```
      MemBlock[?address] := ?data;
      WriteEnd;
   end when;
```

From the system's point of view, the CPU is nothing but a black box that generates read and write requests to the main memory. Accordingly, the CPU designed was simple, with two main **when** bodies. The first one, triggered by an internal *Execute* event, generates a *ReadCycle* or *WriteCycle* event; the second one waits for a *ReadComplete* or *WriteComplete* event before issuing an *Execute* event to initiate another cycle. The code for these two **when** blocks can be found below. An internal counter, **it**, has been added to the code in order to limit the number of cycles that the CPU will issue.

```
<< executes_commands >>
when Execute then
   it := it-1;
   op := Random(2);
   add := Random(10);
   if op=1 then
      ReadCycle(add);
      end if;
   if op=2 then
      dt  := Random(99);
      WriteCycle(add,dt);
      end if;
end when;

<< request_end >>
when ReadComplete(?data) or WriteComplete then
   if it /= 0 then
      Execute;
   end if;
end when;
```

The file describing the system as a whole consists of the connects between the CPU and the memory indicated in figure 2.1, plus a **when** body for initializing the CPU. In this **when** body the user is asked the number of cycles that the CPU is to perform. That data is then passed to the CPU through a *StartCpu* event. When the CPU receives that event it does nothing more than store that value in **it** and generate an *Execute* event. The body for the system's **when** process is shown below.

```
<< start_activity >>
when Start then
   put("Number of memory accesses->");
   get_line(AccCnt);
   Proc::ExecStart(AccCnt);
end when;
```

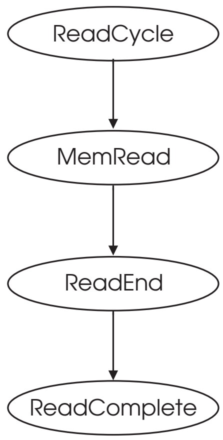

Figure 2.2: Poset for model 1

Compilation and execution of the above model yields a simple partially ordered event set (poset). A small example is shown in figure 2.2. As can be seen, it shows the behavior of the system during a read cycle. The CPU generates a request (*ReadCycle*), the main memory receives the request (*MemRead*), it returns the desired data (*ReadEnd*) and the CPU receives it (*ReadAck*). The graph is totally ordered, with only one causality edge connecting two events. This was expected since this is not a parallel system and, therefore, cannot have two events happening simultaneously.

Chapter 3

The Second System

The next step in the evolution of the system was the introduction of a memory hierarchy through the addition of a cache memory. As figure 3.1 shows, the cache and CPU were encapsulated in one design unit called processor. Figure 3.2 shows the architecture of the resulting system. There was no need to modify the code for the CPU, nor for the main memory design units.

The cache implemented was write-through and fully associative. A fully associative cache is one in which any data/tag combination may be stored in any position. A write-through cache is one in which the CPU writes simultaneously to the cache and the main memory, freeing the cache from having to monitor the bus and update the main memory itself. These cache policies simplify considerably the cache design.

The behavior of the cache is quite simple. During a read cycle, the processor generates a *CacheRead* event, furnishing the address to be read. The cache compares this with the contents of its tag memory and if a match is found, it returns a *Hit* event. When no matching address is found a *MemReadReq* is generated and the desired data is fetched from the main memory. When the memory returns the requested value, the cache stores its own copy of it. The code for implementing that is shown below.

```
<< read_request_processing >>
when CacheRead(?address) then
   Found := 0;
   for i in 1..CACHESIZE loop
      if CacheMem[i].add = ?address then
         Hit(CacheMem[i].dt) pause CACHEDELAY;
         Found :=1;
         exit;
         end if;
      end loop;
   if Found = 0 then
      MainAdd := ?address;
      MemReadReq(?address) pause CACHEDELAY;
      end if;
end when;
```

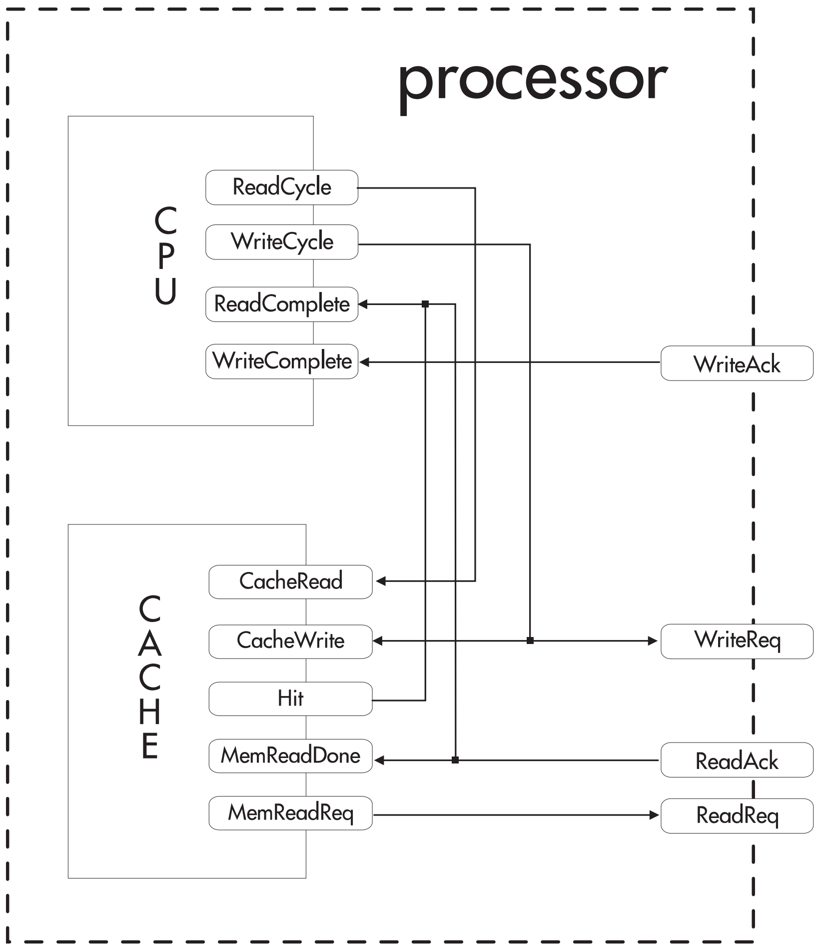

Figure 3.1: Architecture for processor design unit

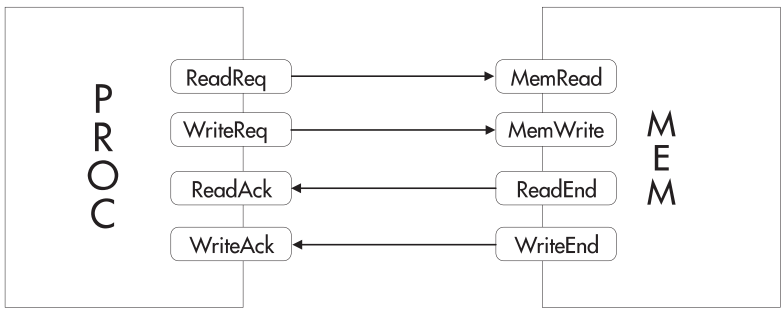

Figure 3.2: Architecture for model 2

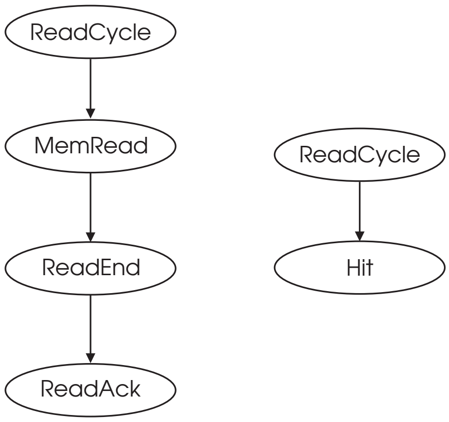

Figure 3.3: Posets for model 2

```
when MemReadDone(?data) then
   TempAdd := GetIndex(MainAdd);
   CacheMem[TempAdd].add := MainAdd;
   CacheMem[TempAdd].dt := ?data;
   end when;
```

During a write cycle there is no interaction between the cache and the main memory. *WriteCycle* events generated by the CPU are sent to both independently. The cache simply copies the corresponding address and data to its own memory. The code for doing that is shown below.

```
<< write_store >>
when CacheWrite(?address,?data) then
   TempAdd := GetIndex(?address);
   CacheMem[TempAdd].add := ?address;
   CacheMem[TempAdd].dt := ?data;
end when;
```

In the code for the cache presented above can be found two references to the function **GetIndex**. This function was added to the model to simplify the coding. It just returns the index to the cache memory position containing the specified address or, if one is not found, the index of a free slot. When the address is not present and there is no free slot, **GetIndex** will randomly generate an index to any position in the cache memory. The code for **GetIndex** can be found in the appendix, as part of the cache design unit definition.

Figure 3.3 shows some reduced posets (i.e., not all events are shown) that occur when the system is executed. The difference between them and the one of the first model is that some of the *MemRead-ReadEnd* event pairs of the latter are substituted by a single *Hit* event, when a match is found in the cache. Note that again, the poset is practically a straight line due to the nonexistence of any parallelism in our model.

Chapter 4

The Third System

Once system 2 is up and running, the next step is to add another processor and turn it into a multiprocessing system. To do so requires two additional modifications. First, since there will be two processors trying to use the same bus, there must be some way to determine which one gets access and when, in order to avoid contention. This requires the use of a bus arbiter and the definition of a bus access protocol.

The second modification is concerned with cache coherency. With two processors it is possible for any of them to modify the data in main memory. Thus, each cache has to monitor the bus and update its entry whenever some processor tries to write to a main memory address that the cache has stored. Figure 4.1 shows the resulting system.

The modification of the cache unit was simple. An **in** action, *ExternWrite*, was added, connected to any write accesses to the main memory. A **when** body was created to treat these accesses by checking if the cache has that address in its banks. If so, it updates its contents with the data available in the bus. The code for this **when** process is shown below.

```
<< external_update >>
when ExternWrite(?address,?data) then
   for i in 1..CACHESIZE loop
      if CacheMem[i].Add = ?Address then
         CacheMem[i].dt  := ?data;
          end if;
      end loop;
end when;
```

The bus arbiter implements a simplified version of the VME-bus protocol. Basically, each processor has three actions associated with bus use: *BusReq*, *BusAck* and *FreeBus*. Whenever a processor wants to acquire the bus, it issues a *BusReq* event. The Bus Arbiter receives this event and, when the bus is free, gives control of it to the processor through the *BusAck* event. When the processor has completed its use of the bus it relinquishes it through a *FreeBus* event.

The VME arbiter recognizes up to 4 request levels, each one indicated by a different request line. Requests in lines with lower numbers have higher priority. That is, no level i request will be acknowledged until all pending requests of a level smaller than i have been processed. Though at first this suggests that only 4 different bus masters (units that want to drive the bus) can be

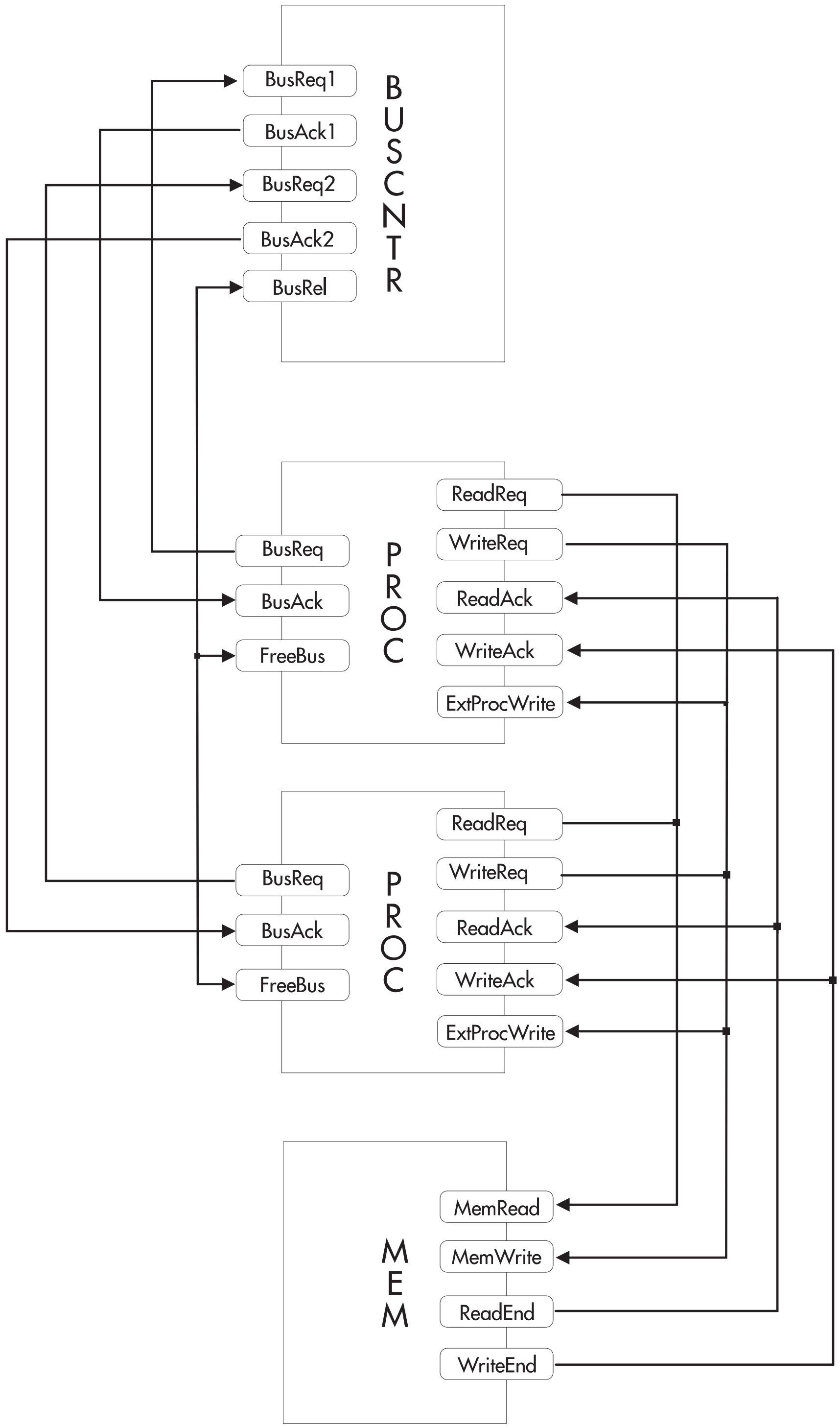

Figure 4.1: Architecture for model 3

connected to the system, that is not true. More than one master can share one request line by using a daisy-chain for each level.

In this model, the arbiter was implemented by the BusControl_c design unit. There are four independent request events (*BusReq1*, *BusReq2*, *BusReq3* and *BusReq4*), as well as four acknowledge lines (*BusAck1*, *BusAck2*, *BusAck3* and *BusAck4*). Each *BusReq* event generates an internal *Dispatch* event that triggers the Dispatcher **when** body, passing the number of the request line as a parameter. The Dispatcher determines if the bus is free or not. If the bus is free, a *BusAck* event is sent to the respective processor, which can then proceed with its bus access, If the bus is already in use, an internal *Hold* event that has as a parameter the number of the *BusReq* line that generated the call to the dispatcher is issued. When the arbiter receives a *BusRelease* event (which generates an internal *Dispatch* event with parameter 5) it checks if any requests are on hold and, if they are, choses one for using the bus and generates an internal *Free* event. If no requests are pending the bus remains free and the arbiter idles. The **when** bodies for implementing these functions are shown below.

```
<< req_analysis_1 >>
when BusReq1 then
   Dispatch(1);
   end when;

 << gen_bus_ack >>
when Hold(?id) and Free(?id) then
   case ?id is
      when 1 => BusAck1;
      when 2 => BusAck2;
      when 3 => BusAck3;
      when 4 => BusAck4;
      when others => null;
      end case;
   end when;

<< dispatcher >>
when Dispatch(?cd) then
   if BusBusy=0 and ?cd/=5 then
      BusBusy := 1;
      Hold(?cd);
      Free(?cd);
   elsif BusBusy/=0 and ?cd/=5 then
      Req[?cd] := 1;
      Hold(?cd);
   elsif Req[1]=1 then
      Req[1]:=0;
      Free(1);
   elsif Req[2]=1 then
      Req[2]:=0;
```

```
      Free(2);
   elsif Req[3]=1 then
      Req[3]:=0;
      Free(3);
   elsif Req[4]=1 then
      Req[4]:=0;
      Free(4);
   else
      BusBusy:=0;
      end if;
   end when;
```

At this point it was also decided to add clocking to the system. This consisted of defining a global clock and adding pause statements to the CPU, the cache and the main memory. The pause statement in the CPU was added to make it wait a random time (between 1 and 5 units) before starting the next read or write cycle. This was a way to simulate the varying time the CPU takes to execute different instructions. For the cache, all that was done was add one unit delay between *CacheRead* and *Hit*, to simulate the read delay found in most memories. Likewise, delays of 3 units were introduced to *ReadEnd* and *WriteEnd* in order to simulate the slower main memory.

Compiling and executing this model results in posets like the one shown in figure 4.2. As can be seen, there are now two "threads," one to each processor. Edges connect the threads at *BusAck* nodes, which depend on a *BusReq* and a *BusRel* in order to proceed with the bus access.

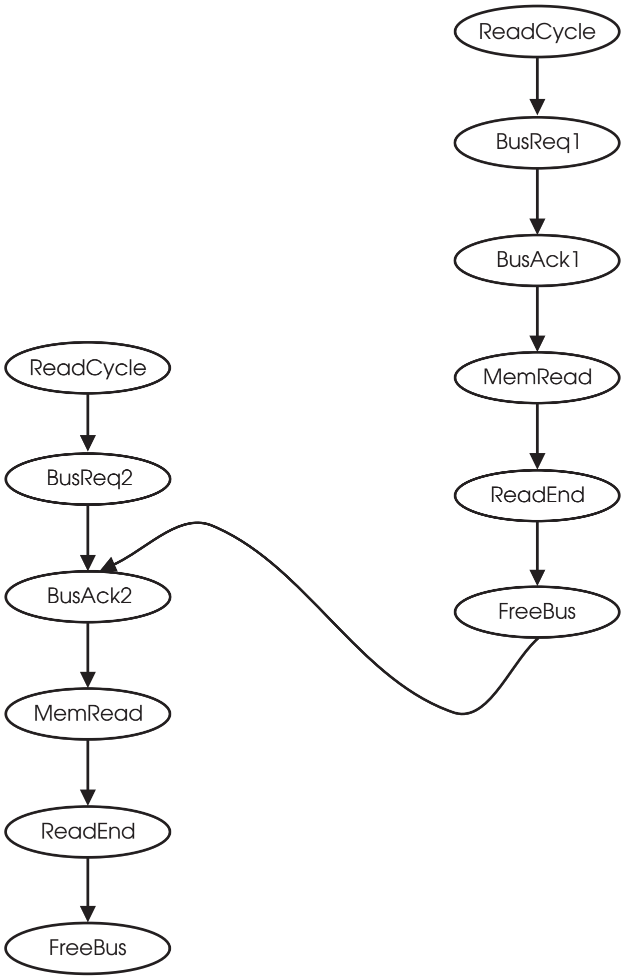

Figure 4.2: Poset for model 3

Chapter 5

Constraints

Constraints are dealt with in a separate chapter because they are not part of the model per se. Instead, they are a mechanism provided by Rapide-0.2 through which one can check a model and guarantee that it is behaving according to specification. In this chapter, each design unit will be analyzed separately and its proper behavior determined. The corresponding constraints will then be defined.

The first unit to be studied is the CPU. As has been said previously, from the system's point of view it is just a black box that creates read and write requests. Thus, to ensure its proper behavior, all that is necessary is to guarantee that it completes a read or write cycle before starting a second one. This leads to the following two constraints:

```
<< complete_read >>
when ReadCycle(?address) then
    ReadComplete(?data)
before Execute;

<< complete_write >>
when WriteCycle(?address,?data) then
    WriteComplete
before Execute;
```

For the cache, only constraints related to read events will be built since cache writes do not generate any event other than storing the data in the memory. Thus, there will be two constraints. The first one checks that a *CacheRead* is really processed (by generating either a *Hit* or *MemReadReq* event) and not ignored. The second one checks that when a read to the main memory is issued the cache will receive the result before processing anything else. The code for these constraints is shown below.

```
<< check_read_processed >>
when CacheRead(?add1) then
   Hit(?dt1) or MemReadReq(?add1)
before CacheRead(?add2) or CacheWrite(?add2,?dt2);
```

```
<< check_read_complete >>
when MemReadReq(?add1) then
   MemReadDone(?dt1)
before CacheRead(?add2) or CacheWrite(?add2,?dt2);
```

The main memory, like the CPU, is the simplest one to build constraints for. One must check only that it does not process a cycle before having completed the previous one. The code for that is:

```
<< check_read >>
when MemRead(?add1) then
   ReadEnd(?dt1)
before MemRead(?add2) or MemWrite(?add2,?dt2);

<< check_write >>
when MemWrite(?add1,?dt1) then
   WriteEnd
before MemRead(?add2) or MemWrite(?add2,?dt2);
```

The bus arbiter is concerned with avoiding conflicts and deadlocks in the bus access mechanism. Thus, it should have constraints to guarantee that no two processors are using the bus at the same time. It should also guarantee that no processor which did not request the bus is getting a *BusAck* event (which would cause deadlock, since that unit would never release the bus). Finally, it should guarantee that the priority mechanism is being respected.

The first is guaranteed by ensuring that there is always a *BusRel* between any two *BusAck* events. The second is guaranteed by declaring that no *BusAck* should happen that is not preceded by the proper request. Priorities are guaranteed by making sure that once there is a *BusReq* at one level, no *BusAck* of higher level is issued before that one is treated. The code for these constraints is shown below.

```
<< free_bus_check >>
when BusAck1 or BusAck2 or BusAck3 or BusAck4 then
   BusRelease
before BusAck1 or BusAck2 or BusAck3 or BusAck4;

<< bus_req_check >>
not BusAck1 before BusReq1;
not BusAck2 before BusReq2;
not BusAck3 before BusReq3;
not BusAck4 before BusReq4;

<< priority_1 >>
when BusReq1 then
   BusAck1
before BusAck2 or BusAck3 or BusAck4;
```

```
<< priority_2 >>
when BusReq2 then
   BusAck2
before BusAck3 or BusAck4;

<< priority_3 >>
when BusReq3 then
   BusAck3
before BusAck4;
```

In the processor unit several constraints have to be verified, some dealing with the memory access, others dealing with the bus protocol. For example, a processor should not release a bus it never acquired, and it should own the bus before issuing any write or read request. On the other hand, it should not release the bus before it has completed the memory access cycle it requested. These conditions are reflected in the constraints shown below.

```
<< guarantee_release >>
when BusReq then
   BusAck
before FreeBus;

<< guarantee_mem_access >>
when BusAck then
   WriteAck or ReadAck(?data)
before FreeBus;

<< bus_permission_for_write >>
when CpuChip::WriteCycle(?address,?data) then
   BusAck
before WriteReq(?address,?data);

<< bus_permission_for_read >>
when CacheChip::MemReadReq(?address) then
   BusAck
before ReadReq(?address);
```

Finally, there is the system level. Most of the system's behavior is already being verified by constraints on its components, so not much needs to be done at this level. The only part of the behavior that could not be checked at a lower level was cache coherency, mainly because it involves the interaction of several units at the system level.

Unfortunately, checking cache coherence is not trivial, because Rapide-0.2's constraint language does not have access to the data and events inside a design unit, and because there is no way to express patterns of the form "the last A causally preceding B." To overcome these problems the code had to be modified.

The first modification had to with detecting cache hits. The only events that the processor sends to the system are requests for the bus and memory read and write events. There is no event visible at the system level that indicates the occurrence of a *ReadCycle* followed by a *Hit*. Thus it was necessary to add two new out actions to the processor design unit, *IntHit* and *IntReadCycle*, and connect the internal *Hit* and *ReadCycle* actions to them. This way each internal event would generate an equivalent external one visible to the system.

Second, a property variable, **LastVal**, was defined and a **when** body created at system level to act as a shadow memory. This **when** body is triggered by any write to main memory and stores in **Lastval**(address) the same data that is stored in main memory. Thus, **Lastval**(address) always contains an exact copy of the respective main memory address, but visible to the system. This when body is shown below.

```
<< memory_monitor >>
when ProcBoard[?id]::Writereq(?address,?data) then
   LastVal(?address) := ?data;
   end when;
```

For the constraint itself, basically it has to state that whenever there is a *ReadCycle* followed by a *Hit*, the data returned by the latter event should be the same as the one currently in the main memory. In order to match each *IntReadCycle* to the corresponding *IntHit*, it was necessary to use the fact that these two events must differ exactly by one time unit, the cache hit delay. The constraint can thus be expressed as:

```
<< cache_coherency_check >>
 not ProcBoard[?id]::IntReadCycle(?address,?t1)=>
       ProcBoard[?id]::IntHit(?data,?t2)
 where
     ?data /= LastVal(?address) and ?t2-?t1=1;
```

Chapter 6

Performance Analysis

Defining, compiling and running a model in Rapide-0.2 results in a poset, which can be represented as a directed acyclic graph. The first use of such a graph is to verify the correctness of the model, by checking its behavior and any possible constraint violations. The poset, though, does not furnish only causality information, but also timing, making it possible to also use it for performance analysis.

To collect data for such an analysis, three distinct models were compiled, differing only in the number of processors. Proc1, Proc2 and Proc3 had each, respectively, one, two and three processors. The compiled models were then run, with the number of cycles per processor in Proc1, Proc2 and Proc3 set to 240, 120 and 80 respectively. The resulting posets, representing the simulation of systems in which the same total work load is shared between different number of processors was analyzed. All the data was obtained by counting events and looking at time stamps. The relevant collected data is shown in table 6.1.

The first, simplest performance data that can be obtained is the speedup of the model as more processors are added. Speedup is defined as the ratio between "execution time with one processor" and "execution time with n processors." In this case,

$$Speedup = \frac{ET_1}{ET_n}$$

where ET_n stands for the execution time of the system with n processors. The execution time can be obtained directly from the posets. They were 1389, 771 and 678 for Proc1, Proc2 and Proc3, which leads to speedups of 1.00, 1.80 and 2.05 respectively. The results can be seen in graphical form in figure 6.1.

A second set of data that can be obtained from the posets is cache performance. It is trivial to count the number of *ReadCycle* events generated by each processor, as well as the number of *Hit*

MODEL	ET	MEM	READS	HITS
Proc1	1389	199	105	41
Proc2	771	185	131	55
Proc3	678	204	119	36

Table 6.1: Data for system analysis

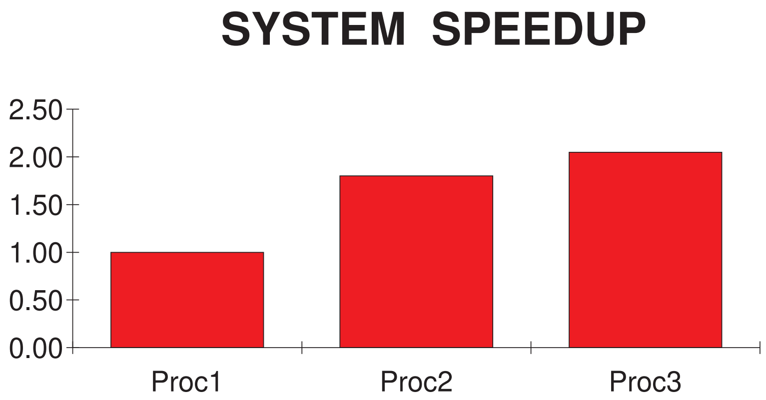

Figure 6.1: Performance analysis - system speedup

MODEL	SPEED.	BAND.	HIT
Proc1	1.00	42.98	39.05
Proc2	1.80	71.98	41.98
Proc3	2.05	90.27	30.25

Table 6.2: Simulation data for performance analysis

events of their corresponding caches. The hit ration is then given by:

$$HitRatio = 100 * \frac{COUNT_{Hits}}{COUNT_{ReadCycles}}$$

Proc1, Proc2 and Proc3 had *Hit* counts of 41, 55, and 36 respectively. Their *ReadCycle* counts were of 105,131 and 119, which results in Hit Ratios of 39.05%, 41.98% and 30.25%. Note that these ratios are close to the expected value of 40% (dictated by the cache's size of 4 entries, the main memory's size of 10 entries and the fact that the read addresses are generated randomly).

Finally, the utilization of the bus bandwidth can also be measured. By bandwidth utilization we mean the ratio of the time that the Bus was actually used to the total execution time. This can be expressed as

$$utilization = 100 * \frac{3 * (COUNT_{MemRead} + COUNT_{MemWrite})}{ET_n}$$

The factor of three multiplying the total number of read and write events is the duration of one such cycle (i.e. the main memory delay). From the data in the test run the resulting bandwidth for each system was 42.98%, 71.98% and 90.27%. This data is shown graphically in figure 6.2.

Table 6.2 summarizes the results presented in this chapter.

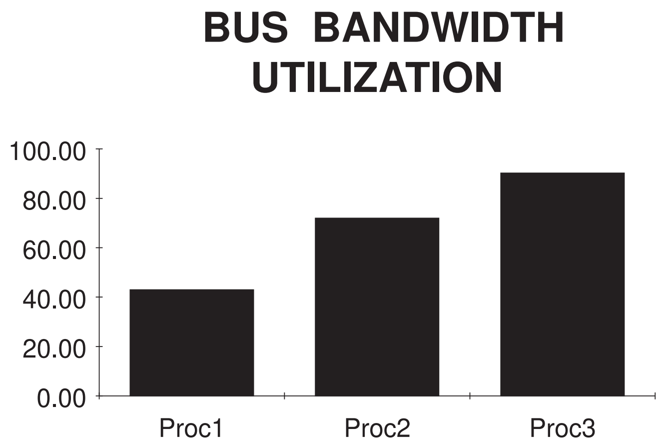

Figure 6.2: Performance analysis - bus bandwidth utilization

Chapter 7

Comments

As a tool for prototyping and modelling systems Rapide-0.2 has many strong points. First of all, event based modelling is fast. With a programming language such as C, one would have to worry about how to, for example, indicate a *Hit* to the processor. Variables would have to be defined, set at the appropriate points and tested. In Rapide-0.2 all that is necessary is to define the event *Hit*, specify the proper connections and the work is done. Whoever is using the system is freed from worrying about several implementation details, making the modelling job proceed at a much faster pace.

Another exciting aspect of the language is the constraint concept. Defining the constraints and coding it is simple job and the resulting posets make verifying the behavior of the model trivial. One does not have to spend long hours looking through linear traces to check if the events happened in the correct order and for the correct reason.

The main advantage of Rapide-0.2, though, is its use of posets. By indicating the causality relation between events they provide much more information than can be gotten from a simple linear trace, and this information has several uses. As will be seen below, this information aids in debugging and provides a much better understanding of the system.

Together with the constraints and the generated *Inconsistent* events, the causality relations provide a powerful and efficient mechanism for tracing the execution and debugging the code. By providing at the same time the final event sequence and information about how one got there, it practically eliminates the need to step through an execution in order to find out what went wrong.

The greatest strength of causality relations, though, is in helping one to understand the system. By tracing back through the graph one can visually identify which of many possible units generated an observed event (such as which processor generated the BusRelease signal). This extra information simplifies the analysis of the system by facilitating the task of grouping events according to the components who caused them.

Another good point of causality relations is that they provide a better, more thorough understanding of the system. For example, examine figure 7.1. It shows a poset of the system with 3 processors when processor 1 has gained control of the bus, while processors 2 and 3 wait. When processor 1 releases the bus, processor 2 is granted access.

The edge between *BusReq3* and *BusAck2* was totally unexpected. After all, why should the fact that processor 3 requested the bus have anything to do with processor 2 getting it? *BusAck2* would occur anyway, even if *BusReq3* had never happened. It certainly didn't depend on it!

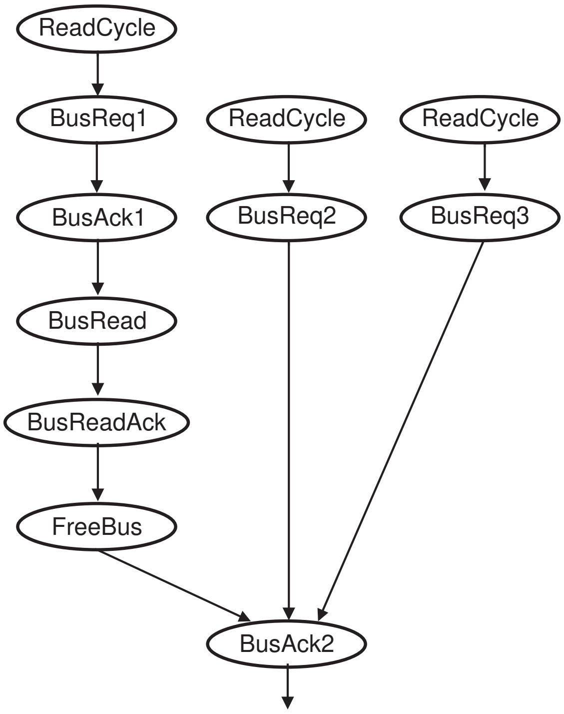

Figure 7.1: Bus access poset for Proc3 with “extra” edge

This extra edge made the author think he had coded the bus arbiter incorrectly, so he rewrote it. No matter how he changed the code, if the system was working correctly (i.e., no constraints were violated), that edge always appeared. This made the author sit down and think about why that edge kept appearing, He had envisioned the arbiter as being mostly a combinational circuit, with one flip-flop to indicate the state of the bus as being in use or not. Any bus requests would propagate through the combinational circuit and the proper acknowledgement would be sent when the bus was free. Since this would imply that an acknowledgment depended only on who requested the bus and who was releasing it, the extra edge made no sense. It turns out that such an implementation would lead to race conditions were two or more requests to arrive simultaneously. The author's vision of how the arbiter was to be implemented was wrong.

The only feasible implementation of the arbiter would be as a state machine, polling each request line at a time and using that to decide what to do. In this case the extra edge would make sense, because *BusReq3* had to be processed before *FreeBus* (it occurred earlier in time) and any event generated by the latter (*BusAck2*, would then depend on the former. Thus, the presence of the "extra" edge clarified for the author what the structure of the arbiter should be like, giving him a deeper understanding of the system.

As a prototyping tool, though, Rapide-0.2 is still not perfect. First of all, it needs a more powerful constraint language. The need to create new events, change interfaces and add **when** bodies just so that the cache coherence constraint could be defined was unsettling. Access to the data internal to a design unit and more powerful pattern constructs would be more than welcome.

There were also some minor inconveniences in the language, but nothing that one would not expect in an experimental one. The author missed the ability to define global constants. Another concept he missed was the one of "exclusive events" for triggering **when** bodies (i.e., once an event triggers one **when** body it cannot be used to trigger another).

As for supporting tools. the partial order browser proved to be a powerful instrument in analyzing posets. Some support tools were missing, though, mainly one to count how many time each type of event occurred in the simulation. An even more interesting tool would have been some sort of architecture editor, a graphical interface that would enable the user to define connect statements by drawing lines between rectangles, rather than having to go through the process of typing it in as connect statements.

These are not major problems and some of them have even been dealt with in Rapide-1.0. The author's main concern, though, is that Rapide is not part of the usual design cycle. That is, once the model is up and running the user has to start again from scratch in building the real system in whatever language (s)he is going to use. This discourages the lazy user from using Rapide and going directly to the final language, despite all of Rapide's advantages in ease of use and speed.

In short, Rapide is a powerful and fast tool for generating models of systems. It is fast because it uses event processing, and powerful because of the causality and timing information provided by the posets. With a more complete accompanying toolset and when it is made part of the design cycle by enabling the user to attach routines written in other programming languages to it, it will be a superb tool for system prototyping and development.

Bibliography

[Bry92] Doug Bryan. Rapide–0.2 language and tool-set overview. Technical Note CSL–TN–92–387, Computer Systems Lab, Stanford University, February 1992.

[Hsi92] Alexander Hsieh. Rapide-0.2 examples. Technical Report CSL–TR–92–510, Computer Systems Lab, Stanford University, February 1992.

Appendix A

Rapide-0.2 Code for Multiprocessor System

This appendix contains the complete Rapide-0.2 code for all the design units making up system 3. Each design unit is specified in one independent file.

A.1 Memory

```
design  memory_c is clocked

   in action MemRead(address : integer);
   in action MemWrite(address : integer; data: integer);
   out action ReadEnd(data : integer);
   out action WriteEnd;

end memory_c;

design body memory_c is

    MEMDELAY : constant integer := 3;

    MemBlock : array [1..10] of integer;

   ?address : integer;
   ?data    : integer;
   ?add1    : integer;
   ?add2    : integer;
   ?dt1     : integer;
   ?dt2     : integer;

-- CONSTRAINTS
```

```
   -- complete read cycle
   << check_read >>
   when MemRead(?add1) then
      ReadEnd(?dt1)
   before MemRead(?add2) or MemWrite(?add2,?dt2);

   -- complete write cycle
   << check_write >>
   when MemWrite(?add1,?dt1) then
      WriteEnd
   before MemRead(?add2) or MemWrite(?add2,?dt2);

begin

   << memory_read_cycle >>
   when MemRead(?address) then
      ReadEnd(MemBlock[?address]) pause MEMDELAY;
   end when;

   << memory_write_cycle >>
   when MemWrite(?address,?data) then
      MemBlock[?address] := ?data;
      WriteEnd pause MEMDELAY;
   end when;

end memory_c;
```

A.2 CPU

```
design cpu_c is clocked

   out action ReadCycle(address : integer);
   out action WriteCycle(address : integer; data : integer);
   in action  ReadComplete(data : integer);
   in action  WriteComplete;
   in action  ExecStart(count : integer);

end cpu_c;

design body cpu_c is

   add : integer;
   dt  : integer;
   op  : integer;
```

```
   it  : integer;

   ?data    : integer;
   ?count   : integer;
   ?address : integer;

   action Execute;

-- CONSTRAINTS

  -- check that a read finishes before starting another cycle
  << complete_read >>
  when ReadCycle(?address) then
      ReadComplete(?data)
  before Execute;

  -- check that a write finishes before starting another cycle
  << complete_write >>
  when WriteCycle(?address,?data) then
      WriteComplete
  before Execute;

begin

   << start_cpu >>
   when ExecStart(?count) then
      it := ?count;
      Execute;
   end when;

   << executes_commands >>
   when Execute then
      it := it-1;
      op := Random(2);
      add := Random(10);
      if op=1 then
         ReadCycle(add);
         end if;
      if op=2 then
         dt  := Random(99);
         WriteCycle(add,dt);
         end if;
   end when;
```

```
   << request_end >>
   when ReadComplete(?data) or WriteComplete then
      if it /= 0 then
         Execute pause Random(5);
      end if;
   end when;

end cpu_c;
```

A.3 Cache

```
design cache_r is clocked

   in action CacheRead (address:integer);
   in action cacheWrite (address:integer; data:integer);
   in action ExternWrite (address:integer; data:integer);
   in action MemReadDone (data:integer);

   out action Hit (data:integer);
   out action MemReadReq (address:integer);

end cache_r;



design body cache_r is

-- constant declarations

   CACHEDELAY : constant integer := 1;
   CACHESIZE  : constant integer := 4;

-- type declarations

   type cacheitem is record
      add   : integer;
      dt    : integer;
      end record;

-- variable declarations

   CacheMem  : array [1..CACHESIZE] of cacheitem;
   TempAdd   : integer;
   MainAdd   : integer;
```

```
   Found     : integer;

-- placeholder declarations

   ?address : integer;
   ?data    : integer;
   ?add1    : integer;
   ?add2    : integer;
   ?dt1     : integer;
   ?dt2     : integer;

-- CONSTRAINTS

   -- checks that a read is executed
   << check_read_processed >>
   when CacheRead(?add1) then
      Hit(?dt1) or MemReadReq(?add1)
   before CacheRead(?add2) or CacheWrite(?add2,?dt2);

   -- checks that it got requested data before receiving
   -- another request
   << check_read_complete >>
   when MemReadReq(?add1) then
      MemReadDone(?dt1)
   before CacheRead(?add2) or CacheWrite(?add2,?dt2);

-- function definition

function GetIndex(ReqAdd:integer) return integer is
begin
   for i in 1..CACHESIZE loop
      if CacheMem[i].add = ReqAdd then
         return i;
         end if;
      end loop;

   for i in 1..CACHESIZE loop
      if CacheMem[i].add = 0 then
         return i;
         end if;
      end loop;
   return Random(CACHESIZE);
end GetIndex;
```

```
begin

-- initialization block
   when Start then
     for i in 1..CACHESIZE loop
         CacheMem[i].add := 0;
         end loop;
      end when;

-- read control block:
   << read_request_processing >>
   when CacheRead(?address) then
      Found := 0;
      for i in 1..CACHESIZE loop
         if CacheMem[i].add = ?address then
            Hit(CacheMem[i].dt) pause CACHEDELAY;
            Found :=1;
            exit;
            end if;
         end loop;
      if Found = 0 then
         MainAdd := ?address;
         MemReadReq(?address) pause CACHEDELAY;
         end if;
   end when;

-- memory read control block:
   when MemReadDone(?data) then
      TempAdd := GetIndex(MainAdd);
      CacheMem[TempAdd].add := MainAdd;
      CacheMem[TempAdd].dt := ?data;
   end when;

-- write control block:
   << write_store >>
   when CacheWrite(?address,?data) then
      TempAdd := GetIndex(?address);
      CacheMem[TempAdd].add := ?address;
      CacheMem[TempAdd].dt := ?data;
   end when;

   << external_update >>
   when ExternWrite(?address,?data) then
      for i in 1..CACHESIZE loop
```

```
         if CacheMem[i].Add = ?Address then
            CacheMem[i].dt  := ?data;
             end if;
         end loop;
      end when;

end cache_r;
```

A.4 Processor

```
with cpu_c, cache_r;

design processor_r is clocked

  out action ReadReq (address:integer);
  out action WriteReq (address:integer; data:integer);
  out action BusReq;
  out action FreeBus;
  out action IntHit (data:integer);
  out action IntReadCycle(address:integer);

  in action ReadAck (data:integer);
  in action WriteAck;
  in action StartCpu(count:integer);
  in action BusAck;
  in action ExtProcWrite(address:integer; data:integer);

end processor_r;


design body processor_r is

-- variable declarations

   CpuChip   : cpu_c;
   CacheChip : cache_r;
   BusUse    : protected boolean;
   ReqType   : integer;
   t_add     : integer;
   t_data    : integer;

   ?address : integer;
   ?data    : integer;
```

```
   ?count   : integer;

-- connection declarations

   connect CpuChip::ReadCycle(?address) with
      CacheChip::CacheRead(?address);
      IntReadCycle(?address);
      end connect;

   connect CpuChip::WriteCycle(?address,?data) with
      CacheChip::CacheWrite(?address,?data);
      end connect;

   connect CacheChip::Hit(?data) with
      CpuChip::ReadComplete(?data);
      IntHit(?data);
      end connect;

-- incredible list of constraints

   << guarantee_release >>
   when BusReq then
      BusAck
   before FreeBus;

   << guarantee_mem_access >>
   when BusAck then
      WriteAck or ReadAck(?data)
   before FreeBus;

   << bus_permission_for_write >>
   when CpuChip::WriteCycle(?address,?data) then
      BusAck
   before WriteReq(?address,?data);

   << bus_permission_for_read >>
   when CacheChip::MemReadReq(?address) then
      BusAck
   before ReadReq(?address);

begin

   when StartCpu(?count) then
      CpuChip::ExecStart(?count);
```

```
   BusUse := false;
   ReqType := 0;
end when;

<< bus_control_write_request >>
when CpuChip::WriteCycle(?address,?data) then
   ReqType := 1;
   t_add := ?address;
   t_data := ?data;
   BusReq;
end when;

<< bus_control_read_request >>
when cacheChip::MemReadReq(?address) then
   ReqType := 2;
   t_add := ?address;
   BusReq;
end when;

<< perform_access >>
when BusAck then
   BusUse := true;
   if ReqType=1 then
      WriteReq(t_add,t_data);
   elsif reqType=2 then
      readReq(t_add);
   end if;
end when;

<< end_read_access >>
when ReadAck(?data) then
   if BusUse = true then
      BusUse := false;
      FreeBus;
      CacheChip::MemReadDone(?data);
      CpuChip::ReadComplete(?data);
      end if;
end when;

<< end_write_access >>
when WriteAck then
   if BusUse = true then
      BusUse := false;
      FreeBus;
```

```
         CpuChip::WriteComplete;
         end if;
   end when;

-- This block propagates an external write event to the cache
   << extern_write_propagate >>
   when ExtProcWrite(?address,?data) then
      if BusUse = false then
         CacheChip::ExternWrite(?address,?data);
         end if;
   end when;

end processor_r;
```

A.5 Bus Arbiter

```
design BusControl_c is clocked

   in action BusReq1;
   in action BusReq2;
   in action BusReq3;
   in action BusReq4;
   in action BusRelease;

   out action BusAck1;
   out action BusAck2;
   out action BusAck3;
   out action BusAck4;

end BusControl_c;


design body BusControl_c is

    BusBusy : integer;
    Req     : array [1..4] of integer;

    action Dispatch(code:integer);
    action Hold(code:integer);
    action Free(code:integer);

    ?cd : integer;
    ?id : integer;
```

```
-- incredible list of constraints

   -- checks for free bus before new grant
   << free_bus_check >>
   when BusAck1 or BusAck2 or BusAck3 or BusAck4 then
      BusRelease
   before BusAck1 or BusAck2 or BusAck3 or BusAck4;

   -- checks if there was a request before receiving permission
   << bus_req_check >>
   not BusAck1 before BusReq1;
   not BusAck2 before BusReq2;
   not BusAck3 before BusReq3;
   not BusAck4 before BusReq4;

   -- checks if priorities are being respected
   << priority_1 >>
   when BusReq1 then
      BusAck1
   before BusAck2 or BusAck3 or BusAck4;

   << priority_2 >>
   when BusReq2 then
      BusAck2
   before BusAck3 or BusAck4;

   << priority_3 >>
   when BusReq3 then
      BusAck3
   before BusAck4;

begin

   << initializations >>
   when Start then
      BusBusy := 0;
      for i in 1..4 loop
        Req[i] := 0;
        end loop;
   end when;

   << req_analysis_1 >>
   when BusReq1 then
      Dispatch(1);
```

```
end when;

<< req_analysis_2 >>
when BusReq2 then
   Dispatch(2);
end when;

<< req_analysis_3 >>
when BusReq3 then
   Dispatch(3);
end when;

<< req_analysis_4 >>
when BusReq4 then
   Dispatch(4);
end when;

 << bus_release >>
when BusRelease then
   Dispatch(5);
end when;

 << gen_bus_ack >>
when Hold(?id) and Free(?id) then
   case ?id is
      when 1 => BusAck1;
      when 2 => BusAck2;
      when 3 => BusAck3;
      when 4 => BusAck4;
      when others => null;
      end case;
end when;

<< dispatcher >>
when Dispatch(?cd) then
   if BusBusy=0 and ?cd/=5 then
      BusBusy := 1;
      Hold(?cd);
      Free(?cd);
   elsif BusBusy/=0 and ?cd/=5 then
      Req[?cd] := 1;
      Hold(?cd);
   elsif Req[1]=1 then
      Req[1]:=0;
```

```
         Free(1);
      elsif Req[2]=1 then
         Req[2]:=0;
         Free(2);
      elsif Req[3]=1 then
         Req[3]:=0;
         Free(3);
      elsif Req[4]=1 then
         Req[4]:=0;
         Free(4);
      else
         BusBusy:=0;
      end if;
   end when;


end BusControl_c;
```

A.6 System

```
with processor_r, memory_c, buscontrol_c;

design sys3_r is global clocked
end sys3_r;

design body sys3_r is

   MEMSIZE : constant integer := 10;

   MemBoard  : memory_c;
   ProcBoard : array [1..2] of processor_r;
   BusMaster : BusControl_c;
   AccCnt    : integer;

   ?address : integer;
   ?data    : integer;
   ?id      : integer;
   ?t1,?t2  : time;

   property LastVal(integer) : integer := 0;

-- CONNECTION DECLARATIONS

-- generic connections
```

```
<< connect_read_request >>
connect ProcBoard[?id]::ReadReq(?address) with
   MemBoard::MemRead(?address);
end connect;


<< connect_read_ack >>
connect MemBoard::ReadEnd(?data) with
   ProcBoard[1]::ReadAck(?data);
   ProcBoard[2]::ReadAck(?data);
end connect;

<< connect_write_req >>
connect ProcBoard[?id]::WriteReq(?address,?data) with
   MemBoard::MemWrite(?address,?data);
   ProcBoard[1]::ExtProcWrite(?address,?data);
   ProcBoard[2]::ExtProcWrite(?address,?data);
end connect;

<< connect_write_ack >>
connect MemBoard::WriteEnd with
   ProcBoard[1]::WriteAck;
   ProcBoard[2]::WriteAck;
end connect;

connect ProcBoard[?id]::FreeBus with
   BusMaster::BusRelease;
   end connect;

connect ProcBoard[1]::BusReq with
   BusMaster::BusReq1;
   end connect;

connect BusMaster::BusAck1 with
   ProcBoard[1]::BusAck;
   end connect;

connect ProcBoard[2]::BusReq with
   BusMaster::BusReq2;
   end connect;

connect BusMaster::BusAck2 with
   ProcBoard[2]::BusAck;
```

```
      end connect;

-- CONSTRAINT DECLARATIONS

  -- cache coherency constraint
  << cache_coherency_check >>
   not ProcBoard[?id]::IntReadCycle(?address,?t1)=>
         ProcBoard[?id]::IntHit(?data,?t2)
   where
       ?data /= LastVal(?address) and ?t2-?t1=1;


begin

   << start_activity >>
   when Start then
      put("Number of memory accesses->");
      get_line(AccCnt);
      ProcBoard[1]::StartCPU(AccCnt);
      ProcBoard[2]::StartCPU(AccCnt);
      end when;


    -- monitoring for cache coherency check
   << memory_monitor >>
   when ProcBoard[?id]::Writereq(?address,?data) then
      LastVal(?address) := ?data;
      end when;

end sys3_r;
```

Appendix B

μRapide Code for Multiprocessor System

During the design of this model, it was decided to also code it in μRapide, an architecture description language (ADL). In doing this, there were two goals in mind. First, it was a check to see if the model could be easily coded in an architecture description language. Second, to see if any extra insight could be gained by using such a language.

As it turns out, μRapide did furnish such an insight, due to its separation of architecture and behavior. In the original Rapide-0.2 model each processor consisted of a CPU and a cache, with the bus interface being provided by conditions on the *with* constructs. When using μRapide it became obvious that such conditions constituted a new unit, the bus interface, which was responsible for all communication between the CPU/cache combination and the outside world. Thus, in this new model, the processors consists of three units, as can be seen in figure B.1.

The rest of this appendix contains the actual code for the multiprocessor system model.

B.1 CPU

```
interface cpu is

   out action ReadCycle(address : integer);
   out action WriteCycle(address : integer; data : integer);

   in action  ReadComplete(data : integer);
   in action  WriteComplete;
   in action  ExecStart(count : integer);

behavior

   add : integer;
   dt  : integer;
   op  : integer;
   it  : integer;
```

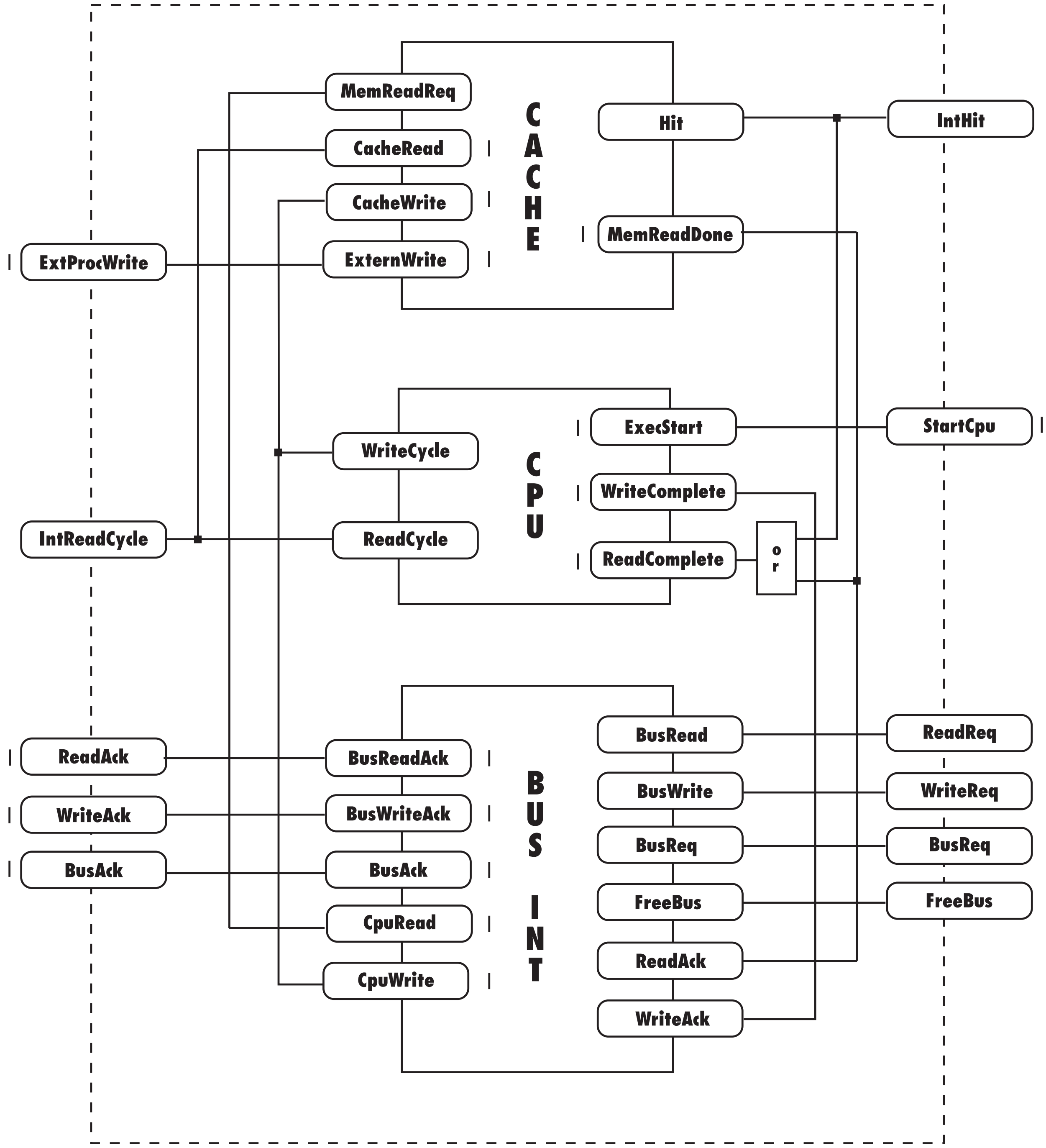

Figure B.1: New architecture for processor unit

```
   ?data    : integer;
   ?count   : integer;
   ?address : integer;

   action Execute(cycle : integer);


-- loads in number of iterations and triggers first execution
ExecStart(?count) => it := ?count;
                     Execute(Random(2)); ;

-- generates a read cycle
Execute(?data) where ?data=1  => it := it-1;
                                 add := Random(10);
                                 ReadCycle(add);;

-- generates a write cycle
Execute(?data) where ?data=2  => it := it-1;
                                 add := Random(10);
                                 dt  := Random(99);
                                 WriteCycle(add,dt);;

-- waits for a random amount of time upon memory access completion
-- and then starts another cycle
( ReadComplete(?data) or WriteComplete) where it/=0  =>
                                 Execute(Random(2)) pause Random(5);;

constraints

 ( Execute(?count) -> ((ReadCycle(?address)->ReadComplete(?data)) or
                       (WriteCycle(?address,?data)->WriteComplete) ))*<

end cpu;
```

B.2 Memory

```
-- This interface describes the main memory unit, which behaves like any
-- decent memory should. It accepts reads and writes and performs the
-- desired operation.

interface memoryc
```

```
   in action MemRead(address : integer);
   in action MemWrite(address : integer; data: integer);

   out action ReadEnd(data : integer);
   out action WriteEnd;

behavior

    MEMDELAY : constant integer := 3;
    MEMSIZE  : constant integer := 10;

    MemBlock : array [1..MEMSIZE] of integer;

   ?add1    : integer;
   ?add2    : integer;
   ?dt1     : integer;
   ?dt2     : integer;
-- received a read request, return data
   MemRead(?add1) => Read ReadEnd(MemBlock[?add1]) pause MEMDELAY;;

-- received a write request, store data and return an acknowledgement
   MemWrite(?add1,?dt1) => MemBlock[?add1] := ?dt1;
                           WriteEnd pause MEMDELAY;;

constraint

  (
    (MemRead(?add1)->ReadEnd(?dt1)) or (MemWrite(?add1,?dt1)->WriteEnd)
  )*<

end memoryc;
```

B.3 Cache

```
-- This file contains the description of the processor cache we are
-- trying to implement. At this moment it is going to be full
-- associative and write-through.

interface cache is

   in action CacheRead (address:integer);
   in action cacheWrite (address:integer; data:integer);
   in action ExternWrite (address:integer; data:integer);
```

```
   in action MemReadDone (data:integer);

   out action Hit (data:integer);
   out action MemReadReq (address:integer);

behavior

-- constant declarations

   CACHEDELAY : constant integer := 1;
   CACHESIZE  : constant integer := 4;
   SPAN       : constant range 1..CACHESIZE

-- type declarations

   type cacheitem is record
      add   : integer;
      dt    : integer;
      end record;

   action IntWrite(address:integer,dt:integer,i:integer);
   action IntRead(address:integer,i:integer);

-- variable declarations

   CacheMem  : array [1..CACHESIZE] of cacheitem;
   TempAdd   : integer;
   MainAdd   : integer;
   Found     : integer;

-- placeholder declarations

   ?address : integer;
   ?data    : integer;
   ?add1    : integer;
   ?add2    : integer;
   ?dt1     : integer;
   ?dt2     : integer;
   ?i       : integer;

-- function definition

function GetIndex(ReqAdd:integer) return integer is
begin
```

```
   for i in 1..CACHESIZE loop
      if CacheMem[i].add = ReqAdd then
         return i;
         end if;
      end loop;

   for i in 1..CACHESIZE loop
      if CacheMem[i].add = 0 then
         return i;
         end if;
      end loop;
   return Random(CACHESIZE);
end GetIndex;

-- initialization, fill cache with zeros
Start =>
   CacheMem[1..MEMSIZE].add := 0;;

-- the next few blocks perform a search through the cache for the
-- requested read data. It performs a loop through three parts, an ini-
-- tialization part, a loop search and an end.

-- loop initialization
CacheRead(?address) =>
   IntRead(?address,1);;

-- loop search, no hit yet
IntRead(?address,?i) where(?i<=CACHESIZE and CacheMem[?i].add/=?address) =>
   IntRead(?address,?i+1);;

-- loop search, found a hit
IntRead(?address,?i) where(?i<=CACHESIZE and CacheMem[?i].add=?address) =>
   Hit(CacheMem[?i].dt) pause CACHEDELAY;;

-- end of loop search, no hit, generate a memory read request
IntRead(?address,?i) where ?i>CACHESIZE =>
   MainAdd := ?address;
   MemReadReq(?address) pause CACHEDELAY;;

-- read complete, store data in cache
MemReadDone(?data) =>
   TempAdd := GetIndex(MainAdd);
   CacheMem[TempAdd].add := MainAdd;
   CacheMem[TempAdd].dt := ?data;;
```

```
-- write requested, store data in cache
CacheWrite(?address,?data) =>
   TempAdd := GetIndex(?address);
   CacheMem[TempAdd].add := ?address;
   CacheMem[TempAdd].dt := ?data;;

-- someone is writing to main memory. Update it if you have the
-- address stored somewhere. Again we use the loop structure

-- initialize loop
ExternWrite(?address,?data) =>
   IntWrite(?address,?data,1);

-- loop search, no hit
IntWrite(?address,?data,?i) where (?i<=CACHESIZE and CacheMem[?i]/=?address) =>
   IntWrite(?address,?data,?i+1);;

-- loop search with hit
IntWrite(?address,?data,?i) where (?i<=CACHESIZE and CacheMem[?i]=?address) =>
   CacheMem[?i].dt  := ?data;;

constraint

   (
      ( CacheRead(?add1) -> (Hit(?dt1) ) or
      ( CacheRead(?add1) -> MemReadReq(?add1) -> MemReadDone(?dt1) ) or
      ( CacheWrite(?add1,?dt1) )
   )*<

end cache;
```

B.4 Bus Interface

```
interface busint is

  out action BusRead (address:integer);
  out action BusWrite (address:integer; data:integer);
  out action BusReq;
  out action FreeBus;
  out action ReadAck(data:integer);
  out action WriteAck;
```

```
  in action BusReadAck (data:integer);
  in action BusWriteAck;
  in action BusAck;
  in action CpuRead(address:integer);
  in action CpuWrite(address:integer,data:integer);

behavior

-- variable declarations

   action IntAck(tp:integer,bu:boolean,data:integer);
   action WantBus(tp:integer,add:integer,data:integer);

   BusUse    : boolean := false;;

   ?address : integer;
   ?data    : integer;
   ?count   : integer;

-- issues bus request when CPU wants to write
   CpuWrite(?address,?data) =>
      WantBus(1,?address,?data);
      BusReq;;

-- issues bus request when CPU wants to read
   CpuRead(?address) =>
      WantBus(2,?address,0);
      BusReq;;

-- waits until it receives a bus it requested and issues a write
-- to main memory
   WantBus(?tp,?address,?data) and BusAck where ?tp=1 =>
       BusUse := true;
       BusWrite(/address,?data);;

-- waits until it receives a bus it requested and issues a read
-- to main memory
   WantBus(?tp,?address,?data) and BusAck where ?tp=2 =>
       BusUse := true;
       BusRead(?address);;

-- a read event has bee received, send it to be processed along with the
-- current state of the bus (if this processor owns it or not)
   ReadAck(?data) =>
```

```
       IntAck(2,BusUse,?data);

-- a write event has bee received, send it to be processed along with the
-- current state of the bus (if this processor owns it or not)
   WriteAck(?data) =>
       IntAck(1,BusUse,0);

-- received the read requested, release the bus and send the data to
-- the CPU
   IntAck(?tp,?bu,?data) where (?tp=2 and ?bu=true) =>
        BusUse:=false;
        FreeBus;
        ReadAck(?data);;

-- received the write requested, release the bus and send the data to
-- the CPU
   IntAck(?tp,?bu,?data) where (?tp=1 and ?bu=true) =>
        BusUse := false;
        FreeBus;
        WriteAck;;

constraints

  (
    ( CpuWrite(?add,?dt) -> BusReq -> BusAck -> BusWrite(?add,?dt) ->
          BusWriteAck -> FreeBus -> WriteAck                         ) or
    ( CpuRead(?add) -> BusReq -> BusAck -> BusRead(?add) ->
          BusReadAck(?dt) -> FreeBus -> WriteAck(?dt)                )
  )*<

end busint;
```

B.5 Processor

```
-- This file describes the processor board of our computer, consisting
-- of the CPU, the cache and the bus interface.

interface processor is

  out action ReadReq (address:integer);
  out action WriteReq (address:integer; data:integer);
  out action BusReq;
  out action FreeBus;
  out action IntHit (data:integer);
```

```
  out action IntReadCycle(address:integer);

  in action ReadAck (data:integer);
  in action WriteAck;
  in action StartCpu(count:integer);
  in action BusAck;
  in action ExtProcWrite(address:integer; data:integer);

behavior

  ?add,?dt : integer;

constraints

  (
    ( BusReq -> BusAck -> ReadReq(?add) -> ReadAck(?dt) -> FreeBus ) or
    ( BusReq -> BusAck -> WriteReq(?add,?dt) -> WriteAck -> FreeBus )
  )*<

end processor

with cpu, cache, busint;

architecture ProcArch
   for Processor is

   P : cpu;
   M : cache;
   C : busint;

   ?address : integer;
   ?data    : integer;
   ?count   : integer;

connections

   P.ReadCycle(?address) = M.CacheRead(?address);
   P.ReadCycle(?address) = IntReadCycle(?address);
   P.WriteCycle(?address,?data) = M.CacheWrite(?address,?data);
   P.WriteCycle(?address,?data) = C.CpuWrite(?address,?data);
   (M.Hit(?data) or C.ReadAck(?data)) => P.ReadComplete(?data);
   M.Hit(?data) = IntHit(?data);
   M.MemReadReq(?address) = C.CpuRead(?address);
   C.BusRead(?address) = ReadReq(?address);
```

```
   C.BusWrite(?address,?data) = WriteReq(?address,?data);
   C.BusReq = BusReq;
   C.FreeBus = FreeBus;
   C.ReadAck(?data) = M.MemReadDone(?data);
   C.WriteAck = P.WriteComplete;
   StartCpu(?count) = P.ExecStart(?count);
   ReadAck(?data) = C.BusReadAck(?data);
   WriteAck = C.BusWriteAck;
   BusAck = C.BusAck;
   ExtProcWrite(?address,?data) = M.ExternWrite(?address,?data);

end ProcArch
```

B.6 Bus Arbiter

```
-- This interface describes the bus arbiter unit, responsible for coor-
-- dinating the access to the data bus by requesting processors. A VME-
-- like protocol is implemented, with priority given to requests on
-- lines with lower numbers

interface BusControl is

   in action BusReq1;
   in action BusReq2;
   in action BusReq3;
   in action BusReq4;
   in action BusRelease;

   out action BusAck1;
   out action BusAck2;
   out action BusAck3;
   out action BusAck4;

behavior

   action IntBus(id:integer, state:boolean);
   action Release(cd:integer);

   BusFree : integer;
   Req     : array [1..4] of integer;
   code    : integer;

   ?id, ?st, ?cd : integer;
```

```
-- translates Bus Request into an internal event with information
-- about current bus state (free or not)
BusReq1 => IntBus(1,BusFree);;
BusReq2 => IntBus(2,BusFree);;
BusReq3 => IntBus(3,BusFree);;
BusReq4 => IntBus(4,BusFree);;

-- adds request to the waiting list if bus is not free
IntBus(?id,?st) where ?st=false => Req[?id]:=1;;

-- gives the bus to whomever requested it if it was free
IntBus(?id,?st) where (?id=1 and ?st=true) => BusFree=false; BusAck1;;
IntBus(?id,?st) where (?id=2 and ?st=true) => BusFree=false; BusAck2;;
IntBus(?id,?st) where (?id=3 and ?st=true) => BusFree=false; BusAck3;;
IntBus(?id,?st) where (?id=4 and ?st=true) => BusFree=false; BusAck4;;

-- has received a bus release signal, now sets the code according to
-- the waiting events. This code is used to determine who next gets
-- access to the bus and uses a system that gives priority to requests
-- with smaller numbers
BusRelease => code := Req[1]*8+Req[2]*4+Req[3]*2+Req[4];
              Release(code);;

-- no one requested the line, set status of bus to free
Release(?cd) where ?cd=0 => BusFree=true;;

-- someone wants to use the line, find out who(according to the code)
-- and give her the proper permission
Release(?cd) where (?cd>=8 and ?cd<16) => Req[1]:=0; BusAck1;;
Release(?cd) where (?cd>=4 and ?cd<8 ) => Req[2]:=0; BusAck2;;
Release(?cd) where (?cd>=2 and ?cd<4 ) => Req[3]:=0; BusAck3;;
Release(?cd) where (?cd>=1 and ?cd<2 ) => Req[4]:=0; BusAck4;;

constraints

   not (BusAck1<BusReq1)
   not (BusAck2<BusReq2)
   not (BusAck3<BusReq3)
   not (BusAck4<BusReq4)

   ( (BusAck1 or BusAck2 or BusAck3 or BusAck4) -> BusRelease )*<

   ( BusReq1 -> BusAck1 )*<
   ( BusReq2 -> BusAck2 )*<
```

```
   ( BusReq3 -> BusAck3 )*<
   ( BusReq4 -> BusAck4 )*<

end BusControl;
```

B.7 System

```
-- architecture of the computer system. It consists of two processors
-- sharing a main memory block, plus a bus arbiter used to solve bus
-- access conflicts.

with processor, memoryc, buscontrol;

architecture system is

   MEMSIZE : constant integer := 10;

   Mem     : memoryc;
   P       : array [1..2] of processor;
   Arbiter : BusControl;
   AccCnt  : integer;

   ?address : integer;
   ?data    : integer;
   ?id      : integer;
   ?t1,?t2  : time;

   LastVal  : array [1..MEMSIZE] of integer;

connections

   P[?id].ReadReq(?address) = Mem.MemRead(?address);
   P[?id].WriteReq(?address,?data) = Mem.MemWrite(?address,?data);
   P[?id].WriteReq(?address,?data) = P[1].ExtProcWrite(?address,?data);
   P[?id].WriteReq(?address,?data) = P[2].ExtProcWrite(?address,?data);

   P[?id].WriteReq(?address,?data) => LastVal(?address):=?data;

   Mem.WriteEnd = P[?id].WriteAck;
   Mem.ReadEnd(?data) = P[?id].ReadAck(?data);

   P[?id].FreeBus = Arbiter.BusRelease;

   P[1].BusReq = Arbiter.BusReq1;
```

```
   P[2].BusReq = Arbiter.BusReq2;

   Arbiter.BusAck1 = P[1].BusAck;
   Arbiter.BusAck2 = P[2].BusAck;

Start => put("Number of memory accesses->");
         get_line(AccCnt);
         ProcBoard[1]::StartCPU(AccCnt);
         ProcBoard[2]::StartCPU(AccCnt);;

constraints

   not (P[?id].IntReadCycle(?address,?t1) -> P[?id].IntHit(?data,?t2))
      where ?data /= LastVal(?address) and ?t2-?t1=1;

end system;
```