3-26-1993

A Systolic Simulation and Transformation System

Ronald I. Greenberg
Rgreen@luc.edu

H.-C. Oh

EE-TR-93-027

A Systolic Simulation and Transformation System

Ronald I. Greenberg and H.-C. Oh [*]
Department of Electrical Engineering
University of Maryland
College Park, MD 20742
rig@eng.umd.edu and ohc@eng.umd.edu

March 26, 1993

Abstract

This paper presents a CAD tool, SystSim, to ease the design of systolic systems. Given a high-level, functional description of processors, and a high-level description of their interconnection, SystSim will perform simulations and provide graphical output. SystSim will also perform transformations such as retiming, which eases use of the methodology of Leiserson and Saxe of designing a system with broadcasting and then obtaining a systolic system through retiming.

[*]Work supported in part by NSF grant CCR-9109550.

1 Introduction

This paper presents a CAD tool for designing systolic (and semisystolic or synchronous) systems. Since systolic arrays were first proposed by Kung and Leiserson[4], their suitability to the VLSI technology has attracted many researchers' interests. However, the design and analysis of systolic arrays is difficult and error prone, so that various design methodologies[1, 5, 8, 10] and CAD tools[2, 11, 13, 14] have been proposed.

This paper discusses a CAD tool to support the design methodology proposed by Leiserson and Saxe[8, 9] as well as additional transformations that may be used to optimize systolic systems [3, 7]. The basic methodology of Leiserson and Saxe is to first design a *semisystolic* or *synchronous* system that is not necessarily systolic. While a systolic system may be viewed as a collection of processors (combinational logic) connected by communication edges, with each edge containing at least one register (clocked memory), a semisystolic system may, in general, include edges that contain no registers. (A semisystolic system must still meet the requirement that there is no *cycle* in the communication graph that is entirely without registers.) It is often easier to design a system with a specified functionality that is merely semisystolic, because such systems can implement a *broadcast*, an essentially simultaneous communication of a datum to all processors in the system. That is, despite staying within the usual systolic model that limits each processor to a constant number of "local" connections, independent of system size, broadcasts can be accomplished in a semisystolic system by having signals ripple through the system from one processor to another unimpeded by registers. Once the semisystolic system has been designed, it can be automatically transformed to a systolic system by the technique of *retiming* [8, 9].

SystSim is geared towards two major goals. First, it provides tools for describing arbitrary semisystolic systems in a concise hierarchical fashion and then simulating their behavior. Instead of using a standard logic simulator and building a system up from switch-level or gate-level descriptions, it is possible to start from a higher-level functional description of modules and compose new subsystems from previously described subsystems. Tools are provided to ease the composition of regular arrays of processors, but less homogeneous systems may also be composed. As an important aid to the simulation process, SystSim provides some graphical output capabilities that allow the user to view the data as it passes through the system from one time step to the next. Second, in addition to the tools for describing and simulating semisystolic systems, various automated transformations are provided. While some of these capabilities are still being programmed for inclusion in the final version of this paper, SystSim currently will perform retiming to meet a specified clock period when possible. It also supports the simple technique of *slowdown*, which is often useful in conjunction with retiming. The final version of the paper will also implement such operations as *hold-up*, *interlacing*, *coalescing*, and the use of the basic techniques for such purposes as maximizing clock rate, maximizing throughput, and minimizing latency. [3, 7, 8, 9].

Section 2 of this paper gives a general description of the capabilities of SystSim, Section 3 provides a simple example, and Section 4 provides concluding remarks.

2 SystSim Capabilities

This section describes the capabilities of SystSim. We begin with a discussion of how the user inputs a system specification. Then we discuss the simulation and transformation capabilities.

2.1 Specification of User System

As mentioned in the introduction, the user inputs a system description in a concise hierarchical fashion. The general form for describing a type of bottom-level processing element (PE) is as follows:

(`defpe` *pe_type input_list output_list delay compute_def self_loops*) ,

where *pe_type* is a name for the type of PE, *input_list* is the list of the input ports of the PE, *output_list* is the list of the output ports of the PE, *delay* specifies the time delay of the PE, *compute_def* is a list of lisp forms that compute the PE outputs from the inputs, and *self_loops* is described below. In the current implementation, it is assumed that all the internal paths from input ports to output ports of a PE have the same delay. This delay can be taken as the maximum delay occurring in the PE, at the cost of a bit of conservatism in the constraints on the clock period.

(As an aside to LISP experts, **defpe**, like **defglue** and **deflinarray** below are "macros", i.e., the arguments are not evaluated. This is of significance in terms of the exact syntax used in the example system description of Section 3, but we will avoid getting bogged down in a COMMON LISP tutorial. Any doubts about the syntax for system specification should be cleared up by the example; in this paper, additional knowledge about COMMON LISP should be necessary only to understand the exact form of the functional specification for the processors.)

It should be noted that our general approach is to define types of PEs and to build up definitions of larger subsystems from previously defined PEs or subsystems. After a subsystem has been defined, instances (copies) may be created and named by using the general function **create** in the following form:

(**create** *sys_name sys_type*) .

The last argument to **defpe**, *self-loops*, is the name of a function for creating communication edges that link an input and output port of the same instance of the PE. Such functions are defined in the same way as functions to create communication edges between two different subsystems. The general syntax for these "gluing functions" is

(**defglue** *glue_name glue_def*) ,

where *glue_name* is a name for the glue function. At present, *glue_def* is to be just a list of invocations of the function **edge**; later we will allow it to be more general LISP code using the **edge** function. An invocation of the function **edge** is in the following form:

(**edge** *tail_sys tail_port head_sys head_port num_regs*) .

Such an invocation creates an edge directed from the port named *tail_port* on the subsystem *tail_sys* to the port named *head_port* on the subsystem *head_sys*, but **defglue** is designed so that the "gluing function" can be applied to many different pairs of subsystem instances. **defglue** creates a function of two arguments to be known as **subsys1** and **subsys2** within *glue_def*. (When a gluing function is given as the *self-loops* argument of **defpe**, the two arguments it will later be applied to will be the same PE.) Hence, an edge from the first of the two subsystems being connected together to the second can be written as

(**edge subsys1** *tail_port* **subsys2** *head_port num_regs*) .

(The syntax for defining communication edges is slightly verbose with respect to the example system definition in Section 3, but it will allow implementation of a potentially useful capability for accessing ports that have not been assigned distinct names within the top-level subsystems being connected. For example, it may be desirable to access a port that would be described in English as "the port named *foo* on the 5th element of the *subsys1*", where *subsys1* is a linear array of other subsystems.)

Once subsystems and gluing functions have been defined, subsystems can be connected in various ways to define larger subsystems. For example, we may connect several into a linear array, square array, or hexagonal array; or we may simply connect arbitrary pairs of subsystems in building up an irregular structure. We will complete implementation of all of these connection methods and possibly others for the final paper; at present we restrict attention to the composition of linear arrays, which has already been implemented. In order to describe a linear array, it suffices to say

(**deflinarray** *sys_type subsys_type glue_fn N*) ,

where *subsys_type* is the type of subsystem the linear array is built from, *sys_type* is a name for this type of linear array, *glue_fn* is the gluing function to use for connecting adjacent subsystems, and *N* is the number of elements in the array.

2.2 Simulations and Transformations

After a semisystolic system has been created, built-in functions of SystSim can be invoked to initialize the system (by giving all the data values a special value representing an undefined state) and to tick the clock. Each time the clock is ticked, all the data values are updated. (A topological sort of the portion of the communication graph involving edges with no registers is performed. Then the processor outputs can be computed in an appropriate order to yield valid results.) It is also straightforward to set values of input ports of the system and read values of output ports. The progress of the simulation can also be viewed graphically as described in Section 2.3.

The most interesting of the basic techniques for transforming systems is retiming. Retiming involves rearranging the registers in the system without affecting the input/output behavior of the system as a whole. Retiming is the main tool in transforming a nonsystolic system to a systolic system under the methodology of Leiserson and Saxe[8, 9]. Often, to obtain a systolic system, it is also necessary to apply the technique of slow-down, which simply involves duplicating (or triplicating, etc.) every register in the system. It is also possible to consider more general tasks than transforming a nonsystolic system to a systolic one. Specifically, we might like to find that retiming of a semisystolic system that minimizes the clock period, where the clock period must be long enough to accommodate the maximum sum of processing delays among communication paths that contain no registers. In SystSim, we have implemented a function **cp** to compute the smallest allowable clock period without retiming and a function **meetcpspec** that retimes to meet a specified clock period if that specification can be met. The relatively trivial operation of slow-down has also been implemented. SystSim allows specification of arbitrary delays for processing elements, though the simple example in Section 3 uses only elements of delay 1. The algorithms implemented in SystSim are efficient algorithms for the general-delay case as derived by Leiserson and Saxe[9]. Soon, we will implement additional operations such as *hold-up*, *interlacing*, *coalescing*, and higher-level operations for such purposes as minimizing clock period, maximizing throughput, minimizing latency, and exploring the tradeoffs among these goals [3, 7, 8, 9].

2.3 Graphical Output

Viewing snapshots of the system is a natural way to arrange the movement of data and verify the behavior of systolic (and semisystolic) systems. However, it is tedious and error prone to track down the snapshots of complex systems with a pen and a paper, so SystSim automates this task.

Screen 1 shows a typical view of the computer screen which the user of SystSim works on. In the window at lower left, the user initiates the lisp interpreter, loads SystSim, and loads or types definitions of semisystolic systems. Then he can create and simulate systems, obtaining the snapshot display in the large, upper window. The snapshot window displays the current data values of all the ports. (Some of the data values in Screen 1 are truncated for lack of space, but there is also a capability for zooming in part of the display.) Registers are shown as small black boxes on the communication edges.

Any simulation, transformation, or viewing commands are generally initiated by typing in the window at lower left. For example, the user can type commands to tick the clock or try to meet a clock period specification, or he can enter arbitrary LISP code invoking elementary functions of SystSim. Functions we did not yet discuss since they relate specifically to the graphical output are **(show-elem)** and **(zoom)**. After invoking **(show-elem)**, the user can select an element in the display with the mouse and see the names of the ports for that element, as is also shown in Screen 1. After **(zoom)** is invoked, the mouse is used to select a rectangular region of the display for enlargement, as illustrated in Screen 2. (Screen 2 also illustrates the selection of a different element via the **(show-elem)** command.) When the display has been zoomed, the mouse can also be used to scroll the display or zoom back out. (Screen 2 shows a scroll bar for horizontal scrolling and a button for zooming out. Scrolling has not been completely implemented yet, but the final implementation will include horizontal and vertical scrolling.)

3 An Example: A Priority Queue

This section illustrates the capabilities of SystSim with the example of a priority queue. We begin by explaining the design and the description for SystSim of a semisystolic priority queue (which is not systolic). Then we show the results of some simulations and transformations, including making the system systolic.

3.1 Priority Queue Description

A priority queue is a data structure which supports the following two operations:

- INSERT(x) : Insert a key x into the queue.
- EXTRACT : Extract the smallest key from the queue.

In our example, we will use character strings for the keys, so that smallest means alphabetically earliest.

It is relatively easy to come up with a convincing design for a semisystolic priority queue in the form of a linear array, as sketched by Leiserson[6, Sec. 3.2.1]. In the design described here, priority queue processing elements are connected as shown in Screen 1. (The hosts at the ends of the arrays are used to control the external interfaces of the priority queue.) The smaller picture of a single PE shows the names of the input and output signals. The basic organization is to have each processor store a key as internal state (hence the self loop with ports `cout` and `cin`, which necessarily contains a register) and to also allow processors to pass a character to the left (via `bout`) or right (via `aout`). In addition, there is a signal (represented by `extractin` and `extractout`) passed through the system that is 1 (true) to tell a processor to do its part for an EXTRACT operation and 0 (false) to tell it to do its INSERT behavior. (We will assume that the queue never idles, since idling can be achieved by inserting an element larger than the maximum allowed for true data. In our example, we use "ZZZ" for this special "infinite" data value.)

It is a great convenience, to be able to broadcast the signal telling processors which operation to perform, so that they can all perform a single queue operation together, rather than figuring out how to orchestrate the behavior of processors to perform parts of different queue operations at any given time. Hence, `extractin` is simply passed straight through to `extractout` by each processor. When the operation is extract, the behavior of the processors is particularly simple. Given that the semisystolic system will maintain the invariant that the stored keys of the processors as given by the `cin` values are in sorted order, we simply need each processor to pass its stored key to the processor on its left and to replace its stored key with the one being passed in from the right. Since keys only need to be passed one step to the left, we put a register on the leftgoing communication edges. For insertion, it is again helpful to use rippling logic. The key to be inserted is input at the left end of the array, and each processor passes to the right the larger of its stored key and its received key and stores the smaller of the two.

In total, the operations performed by each processor are as follows. (We include here a slight additional twist to improve performance. Instead of waiting for an extract operation to pass data to the left we finish any operation by placing the newly computed stored key on the leftgoing communication edge as well, so that it will be immediately available to the processor on the left at the beginning of the next cycle. Viewing the graphical simulations discussed below should give the reader fuller confidence in the correctness of this approach.)

```
$extractout \leftarrow extractin$
**if** $extractin = 1$ **then**
    $cout \leftarrow bin$
**else**
    $cout \leftarrow \min\{ain, cin\}$
    $aout \leftarrow \max\{ain, cin\}$
**endif**
$bout \leftarrow cout$
```

```
(defglue PQPESELFGLUE
        (edge subsys1 'cout  subsys1 'cin  1)
)

(defglue PQPEGLUE
        (edge subsys1 'extractout subsys2 'extractin  0)
        (edge subsys1 'aout       subsys2 'ain        0)
        (edge subsys2 'bout       subsys1 'bin        1)
)

(defpe PQPE (ain bin extractin) (aout bout extractout) 1
          ( (setf extractout extractin)
            (cond ( (eq 1 extractin)
                    (setf cout bin)    )
                  ( (eq 0 extractin)
                    (cond ( (string-lessp ain cin)
                            (setf cout ain)
                            (setf aout cin)       )
                          ( t
                            (setf cout cin)
                            (setf aout ain)       )
            (setf bout cout)
          )
          'PQPESELFGLUE
)

(deflinarray  'PQ  'PQPE  'PQPEGLUE  6)
```

Figure 1: The SystSim description of a semisystolic priority queue of length 6

Based on the design, we have just given for a semisystolic priority queue, the complete SystSim description for a priority queue of length 6 is as in Figure 1.

At present, we have also done a bit of ad hoc coding for the two independent hosts at the end of the array, which simulate a single host that has two buffers, one at each side of the queue. (Some of the host generation will probably be automated in a later implementation.) Host2 inserts the special "infinity" element of the input data set into the queue so that the queue can be placed into a well-defined state by initially performing a series of EXTRACT operations. Host1 inserts and extracts data and controls the operation of the queue by changing the output signal of its **extractout** port. We have also defined two LISP functions **extract** and **insert** for convenience. The former performs the number of EXTRACT operations specified by the argument to the function. It sets **extractout** of Host1 to 1, and runs the appropriate number of clock cycles. The latter function takes an argument that is placed on **aout** of Host1, sets **extractout** of Host1 to 0, and runs the simulation. For convenience, we have coded **extract** and **insert** to look up the amount of slowdown applied to the system and to run the corresponding number of clock cycles. We have also arranged that the **bin** value for Host1 is printed after every clock cycle under the label "OUTPUT FROM PRIORITY QUEUE TO HOST1".

3.2 Priority Queue Simulation and Transformation

The initial priority queue created in the Section 3.1 is one with broadcasting and a clock period of 6 (Screen 1). The clock period of 6 is required by the need for signals to ripple through 6 processors on a single clock cycle. To begin the simulation of the initial priority queue, the queue was reset with EXTRACT operations. ("ZZZ" was used as a queue entry which is to be larger than every other one.) Then data were inserted and extracted as follows:

```
(insert 'usa)
(insert 'korea)
(insert 'brazil)
(insert 'china)
(insert 'spain)
(insert 'japan)
(extract 6)
```

As indicated earlier, each INSERT and EXTRACT operation corresponds to one clock cycle. At the end of each simulation cycle, the output from the queue to the left host is the current minimum queue entry. This is the crucial thing to look at in verifying valid operation. Screen 1 shows the outputs of SystSim after 6 data were inserted. As mentioned before, some of the data are truncated, but the full values can be seen by zooming in as in Screen 2. Screen 3 shows the outputs of SystSim after the final 6 EXTRACT operations. (Data items represented by "." are undefined, i.e., floating. We could have continued to place some data item on the **ain** and **aout** lines during extraction, but this would just confuse the picture.) In the window at lower left, the output from the queue to Host1 after each EXTRACT operation is displayed. It was observed that the host computer accessed the data at each clock tick correctly, but the clock period of 6 is undesirably long.

In order to improve the clock period, the queue was retimed to meet a clock period specification of 2. We initialized the system and did

```
(meetcpspec 2.)
```

Then the same simulation procedure as used above was performed again. In this second simulation, the queue intervals look different but the same behavior was obtained from Host1's point of view. Screen 4 shows how the registers were rearranged by the retiming and the results of the second simulation. Each operation still took one clock tick. It can be observed that the outputs from the queue to Host1 did not change, but the required clock period had been reduced to 2 units.

As it turns out, 2 is the best clock period that can be obtained without applying slow-down, but we can see what happens if we try for a shorter clock period by inputting:

```
(meetcpspec 1.)
```

This clock period specification cannot be met, but some retiming is done in the process of trying as shown in Screen 5. SystSim signaled the failure as shown in the window at lower left. This system was also simulated, and again, the queue internals look different, but its behavior was all the same to Host1.

Finally, the system is made 2-slow (Screen 6), and the system is successfully retimed to obtain a clock period of 1 by inputting:

```
(slowdown  2)
(meetcpspec 1.)
```

Finally, the systolic system we have just obtained was simulated. Screen 7 shows part of the system after 6 date were inserted. The data extracted by Host1 during the 6 EXTRACT operation is shown in Screen 8. Two clock cycles are now run for each queue operation, and again Host1 sees the same thing (except that it takes twice as long, so that only every other output is relevant).

4 Conclusion

A CAD tool for designing systolic systems has been implemented and proved to be useful in the example of designing a systolic priority queue. One can verify that the systolic priority queue operates correctly, but it would have been more difficult if an attempt had been made to design it directly instead of going through the approach of transforming a semisystolic system. The contrast is even more marked with the inclusion of additional operations such as deletion or the use of variable-length keys[6]. In fact, even the semisystolic design becomes more convincing after simulation.

There are many capabilities still to be added to SystSim. We intend to include facilities for defining interconnection topologies other that linear arrays, and to include other means of manipulating system descriptions. For example, there should be functions to return the length of a system that is a linear array of subsystems or to refer to an element in the array, just as is the case for ordinary arrays. We also intend to include additional capablities for system analysis and transformation. Finally, it may be desirable to implement the more general delay model touched upon by Leiserson and Saxe [9], in which the delay in a processor may be different for each input-output path.

References

[1] Peter R. Cappello and Kenneth Steiglitz. Unifying VLSI array designs with geometric transformations. In H. J. Siegel and Leah Siegel, editors, *Proceedings of the 1983 International Conference on Parallel Processing*, pages 448–457. IEEE Computer Society Press, 1983.

[2] Bradlley R. Engstrom and Peter R. Cappello. The SDEF systolic programming system. In Sartaj K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 645–652. The Pennsylvania State University Press, 1987.

[3] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.

[4] H. T. Kung and Charles E. Leiserson. Systolic arrays (for VLSI). In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings 1978*, pages 256–282. Society for Industrial and Applied Mathematics, 1979. An earlier version appears in [12, Section 8.3].

[5] Sun-Yuan Kung. *VLSI Array Processors*. Prentice-Hall, 1988.

[6] C. E. Leiserson. *Area-Efficient VLSI Computation*. MIT Press, Cambridge, Massachusetts, 1983.

[7] Charles E. Leiserson. Systolic and semisystolic design. In *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers*, pages 627–632. IEEE Computer Society Press, 1983.

[8] Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, 1983.

[9] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991. Earlier versions as MIT Lab for Computer Science technical memo MIT/LCS/TM-309 and as "Optmimizing synchronous circuitry by retiming" (with Flavio M. Rose) in *Third Caltech Conference on Very Large Scale Integration*.

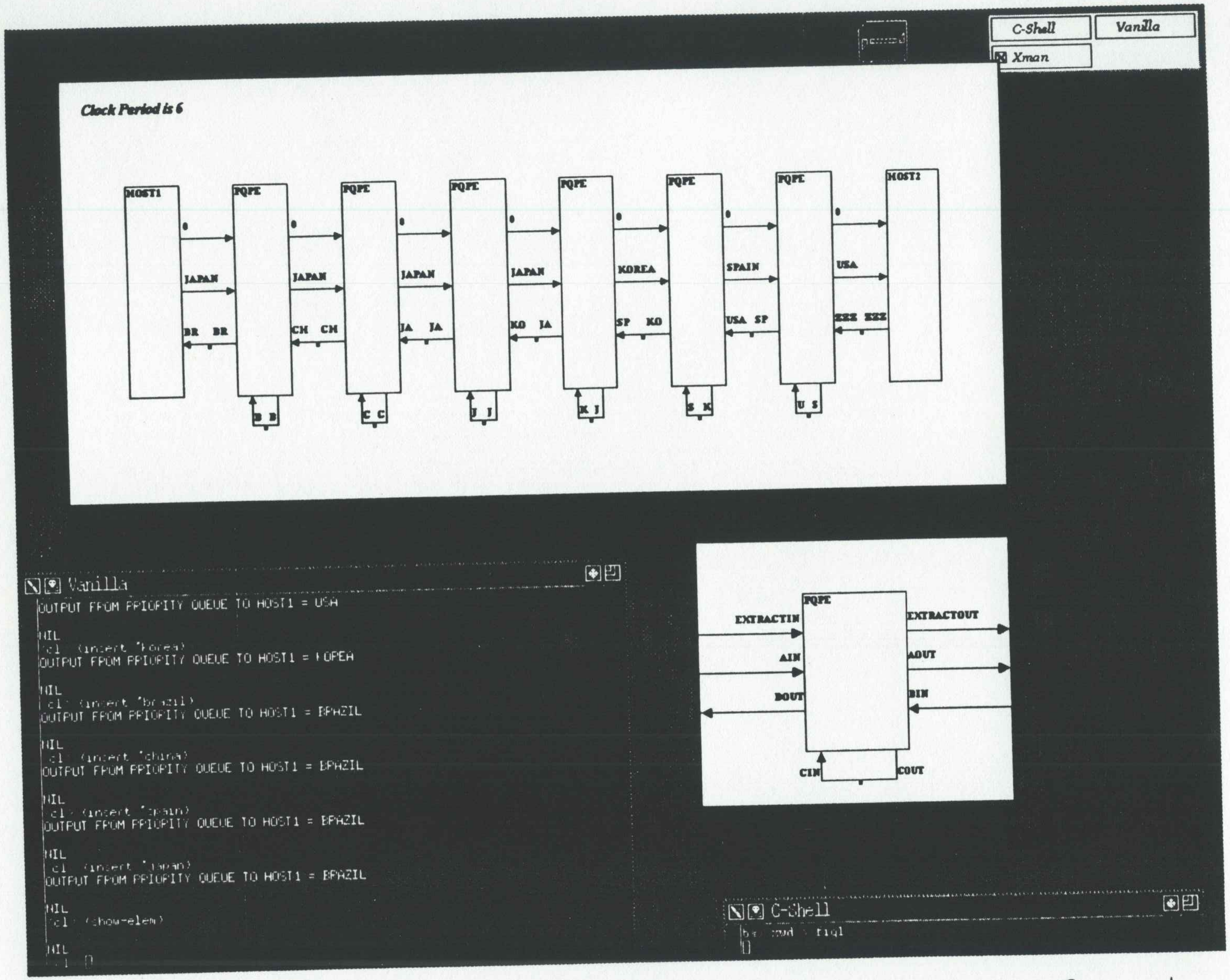
[10] Guo-Jie Li and Benjamin W. Wah. The design of optimal systolic arrays. *IEEE Trans. Computers*, C-34(1):66–77, Jan 1985.

[11] Wayne Luk, Geraint Jones, and Mary Sheeran. Computer-based tools for regular array design. In John McCanny, John McWhirter, and Earl Swartzlander Jr., editors, *Systolic Array Processors*, pages 589–598. Prentice-Hall, 1989.

[12] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980.

[13] Dan I. Moldovan. ADVIS: A software package for the design of systolic arrays. In Robert M. Keller, editor, *Proceedings of the 1984 International Conference on Parallel Processing*, pages 158–164. IEEE Computer Society Press, 1984.

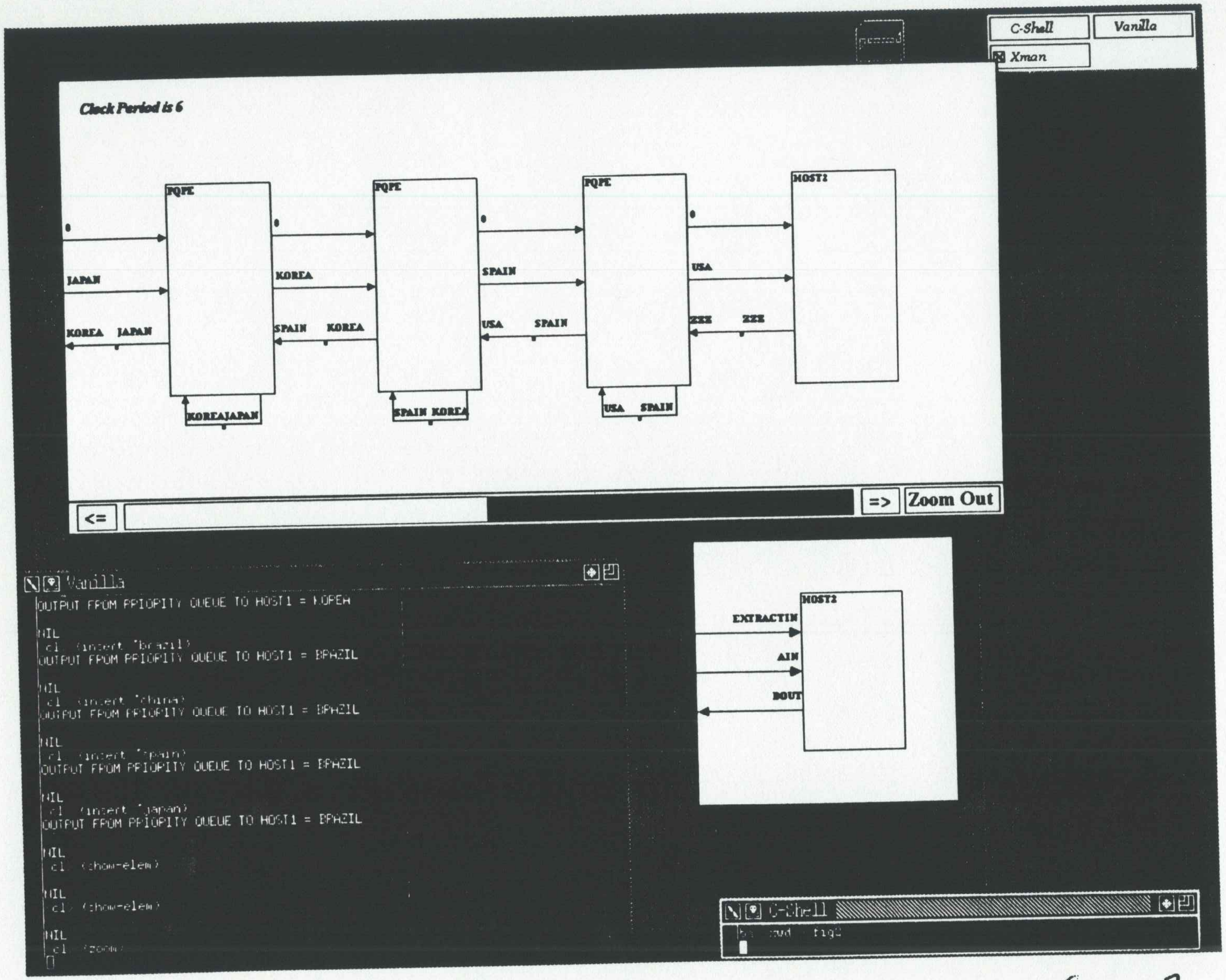
[14] E. Theodore L. Omtzigt. SYSTARS: A CAD tool for the synthesis and analysis of VLSI systolic/wavefront arrays. In Keith Bromley, Sun-Yuan Kung, and Earl Swartzlander Jr., editors, *Proceedings of the 1988 International Conference on Systolic Arrays*, pages 383–391. IEEE Computer Society Press, 1988.



```
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA

NIL
cl> (insert 'korea)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA

NIL
cl> (insert 'brazil)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl> (insert 'china)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl> (insert 'spain)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl> (insert 'japan)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl> (show-elem)

NIL
cl> 
```

```
[illegible]
```

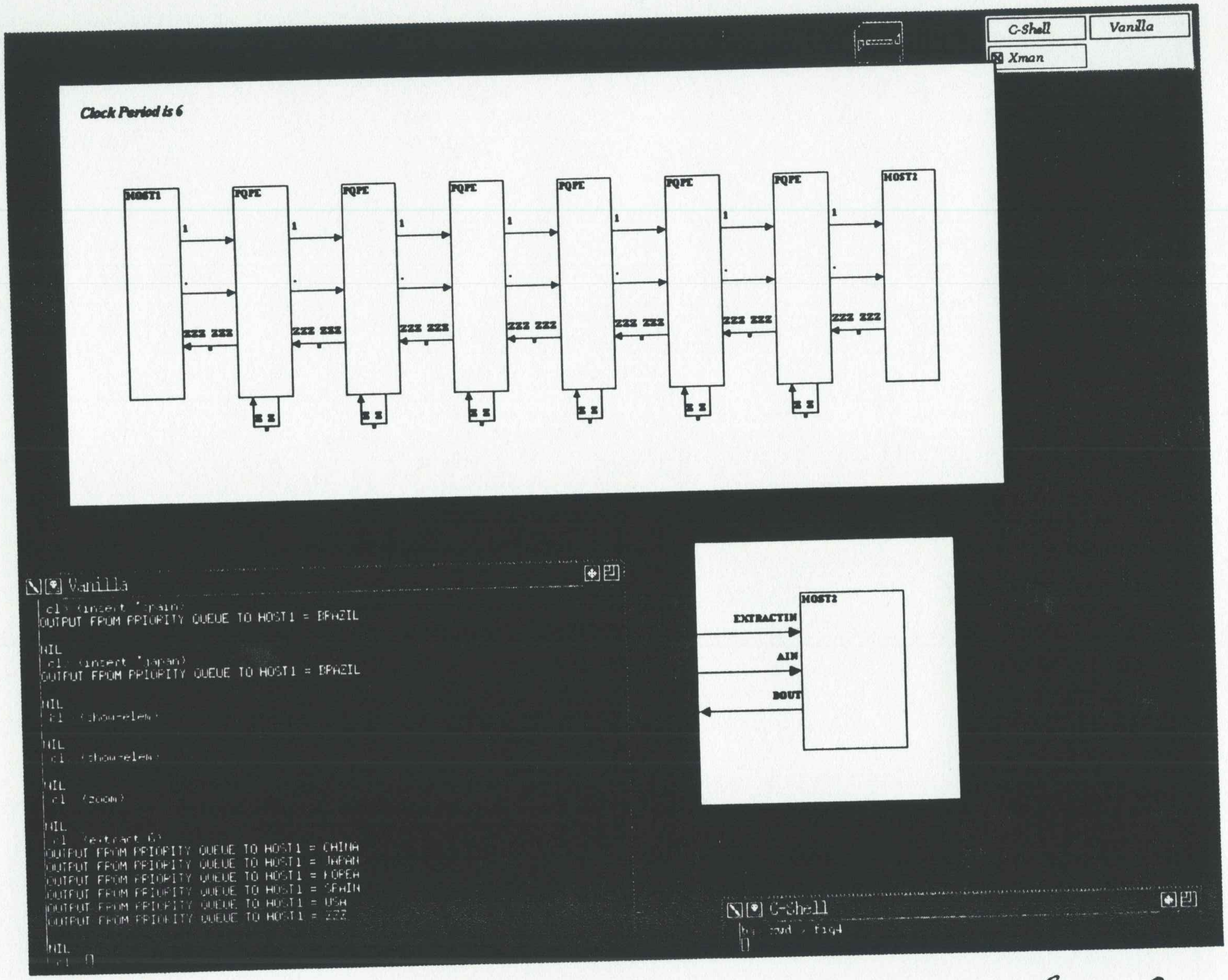
Screen 1



```
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA

NIL
 cl  (insert 'brazil)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 cl  (insert 'china)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 cl  (insert 'spain)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 cl  (insert 'japan)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 cl  (show-elem)

NIL
 cl> (show-elem)

NIL
 cl  (zoom)
```

```
b>  cwd  fig2
```

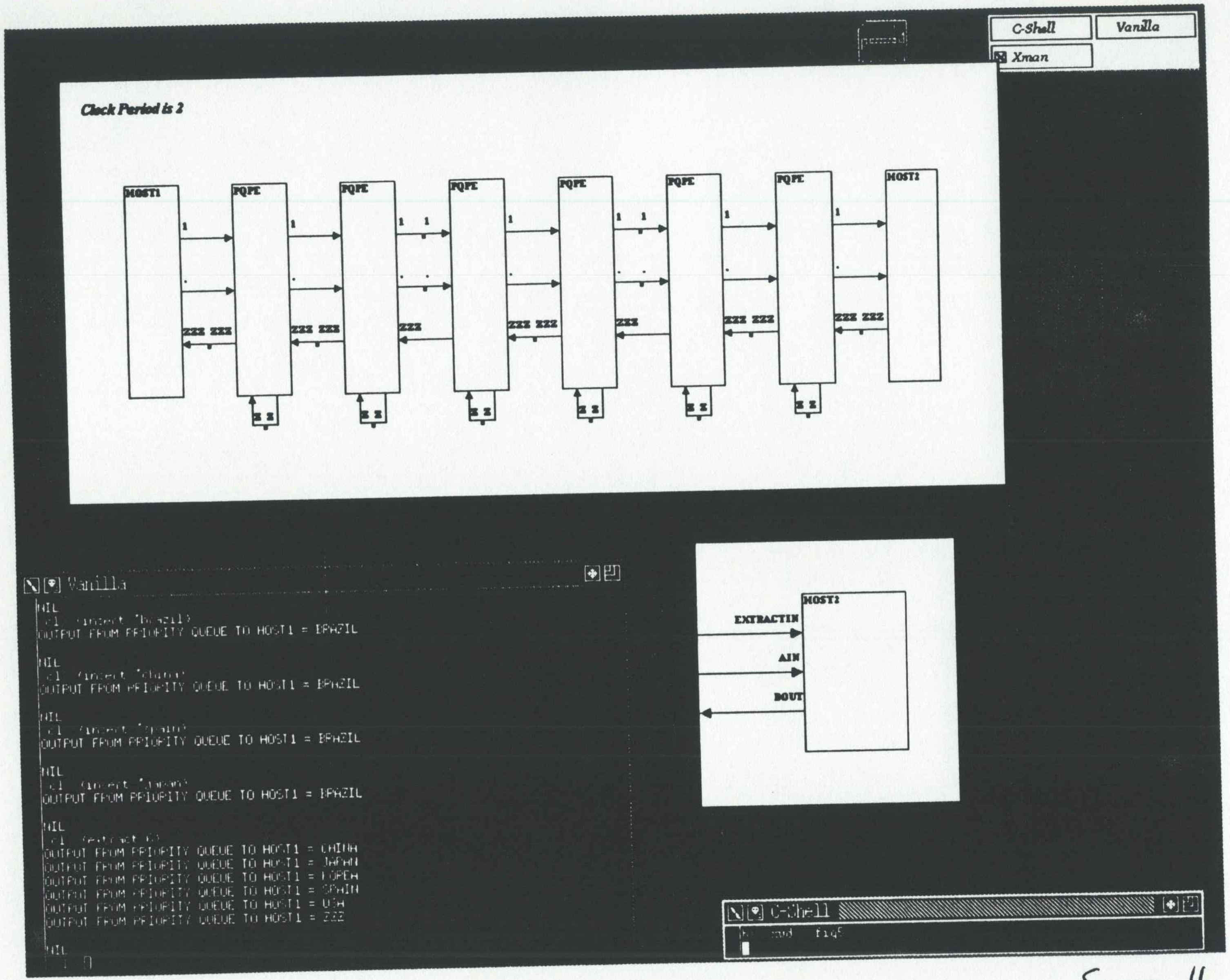
Screen 2



```
cl> (insert 'spain)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl> (insert 'japan)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl> (show-elem)

NIL
cl> (show-elem)

NIL
cl> (zoom)

NIL
cl> (extract 6)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = CHINA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = JAPAN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = SPAIN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = ZZZ

NIL
cl> []
```

```
b> cmd > fig4
[]
```

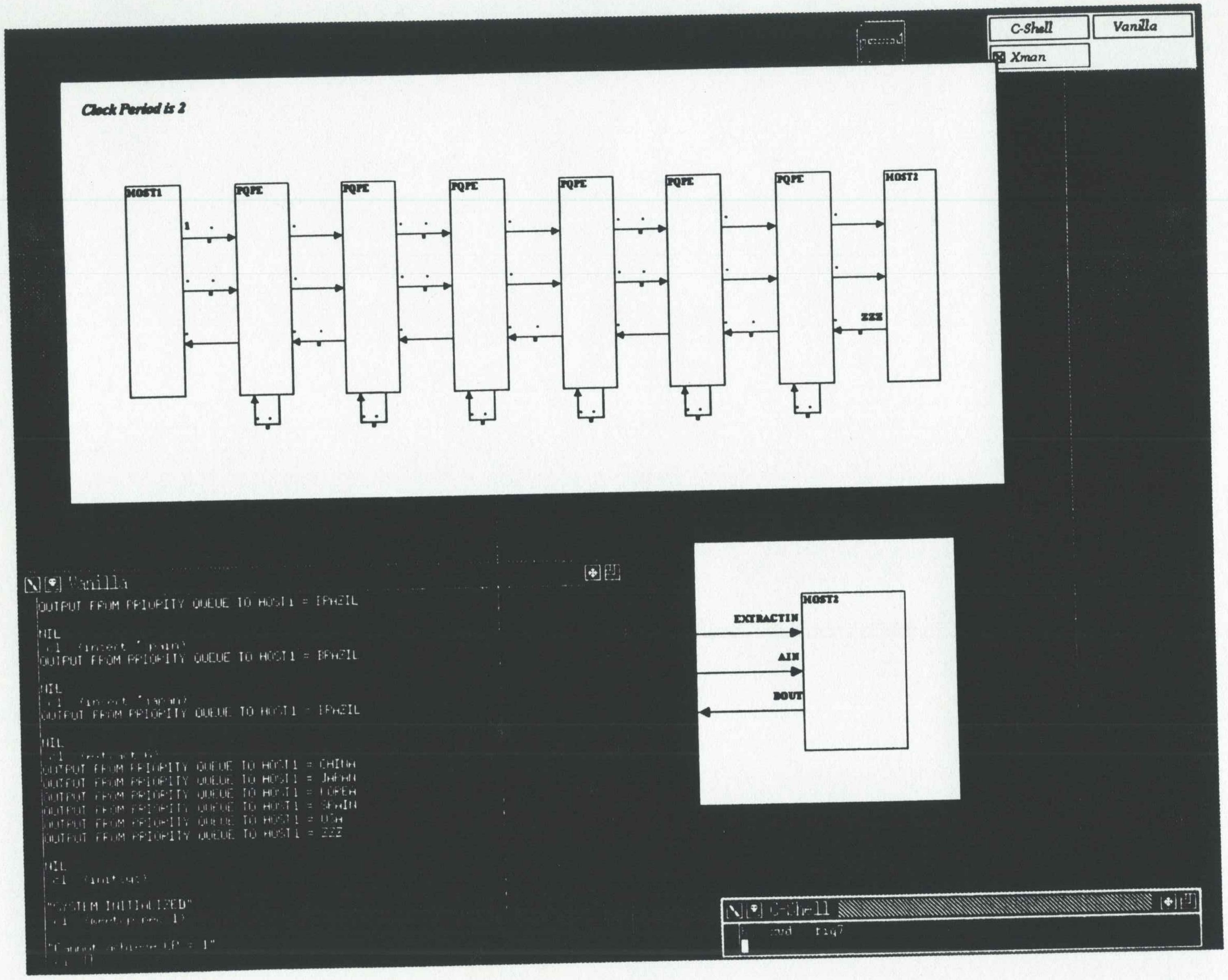
Screen 3



```
Vanilla
NIL
 c1  (insert 'brazil)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 c1  (insert 'china)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 c1  (insert 'spain)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 c1  (insert 'japan)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 c1  (extract 6)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = CHINA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = JAPAN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = SPAIN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = ZZZ

NIL
 c1
```

```
C-Shell
k  cmd  fig5
```

Screen 4

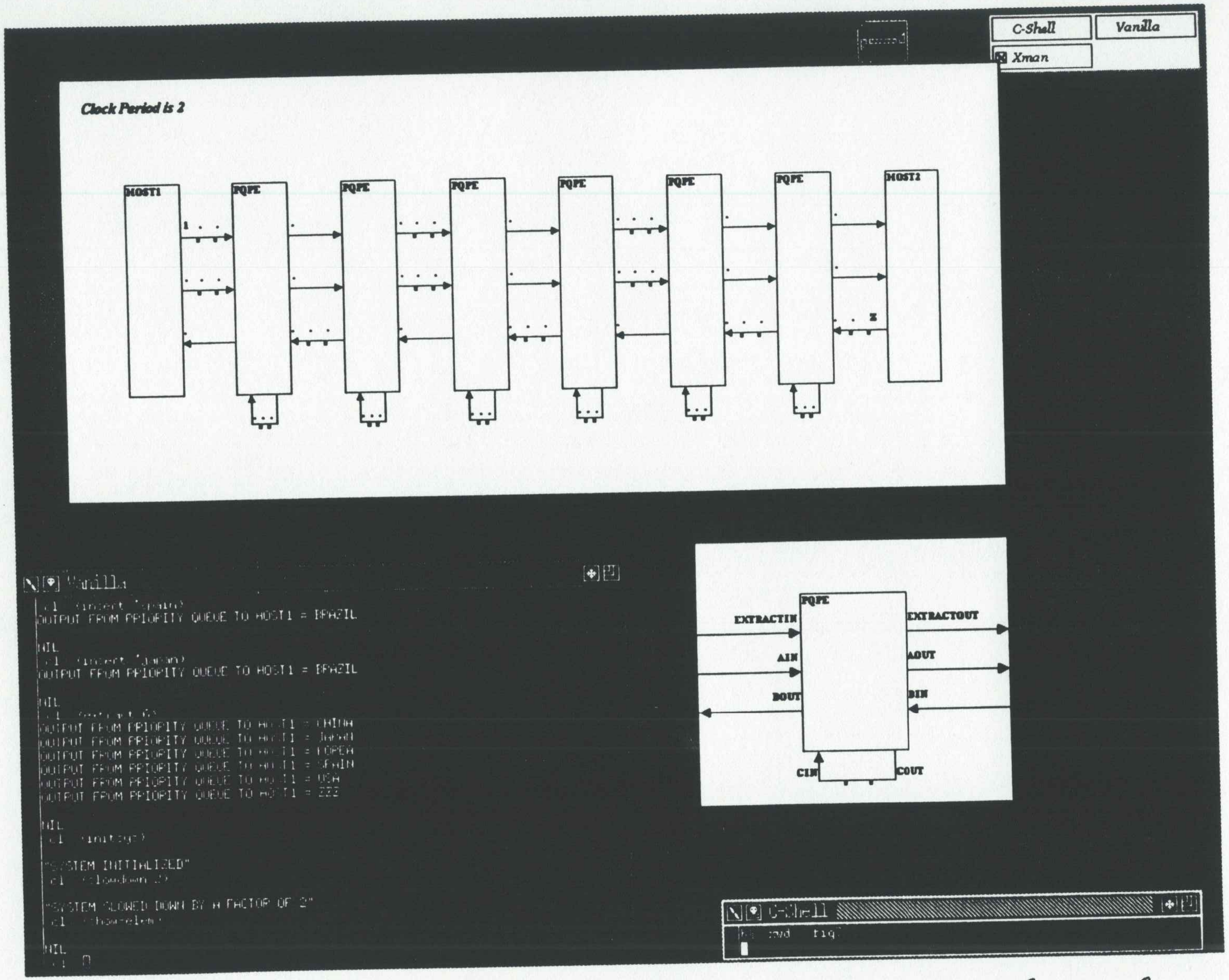


```
OUTPUT FROM PRIORITY QUEUE TO HOST1 = IRAZIL

NIL
<1> (insert 'brain)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
<1> (insert 'japan)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = IRAZIL

NIL
<1> [illegible]
OUTPUT FROM PRIORITY QUEUE TO HOST1 = CHINA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = JAPAN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = SPAIN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = ZZZ

NIL
<1> (init-sys)

"SYSTEM INITIALIZED"
<1> [illegible] 1)

"Cannot achieve CP = 1"
<1> [illegible]
```

```
% cmd fig7
```

Screen 5

14

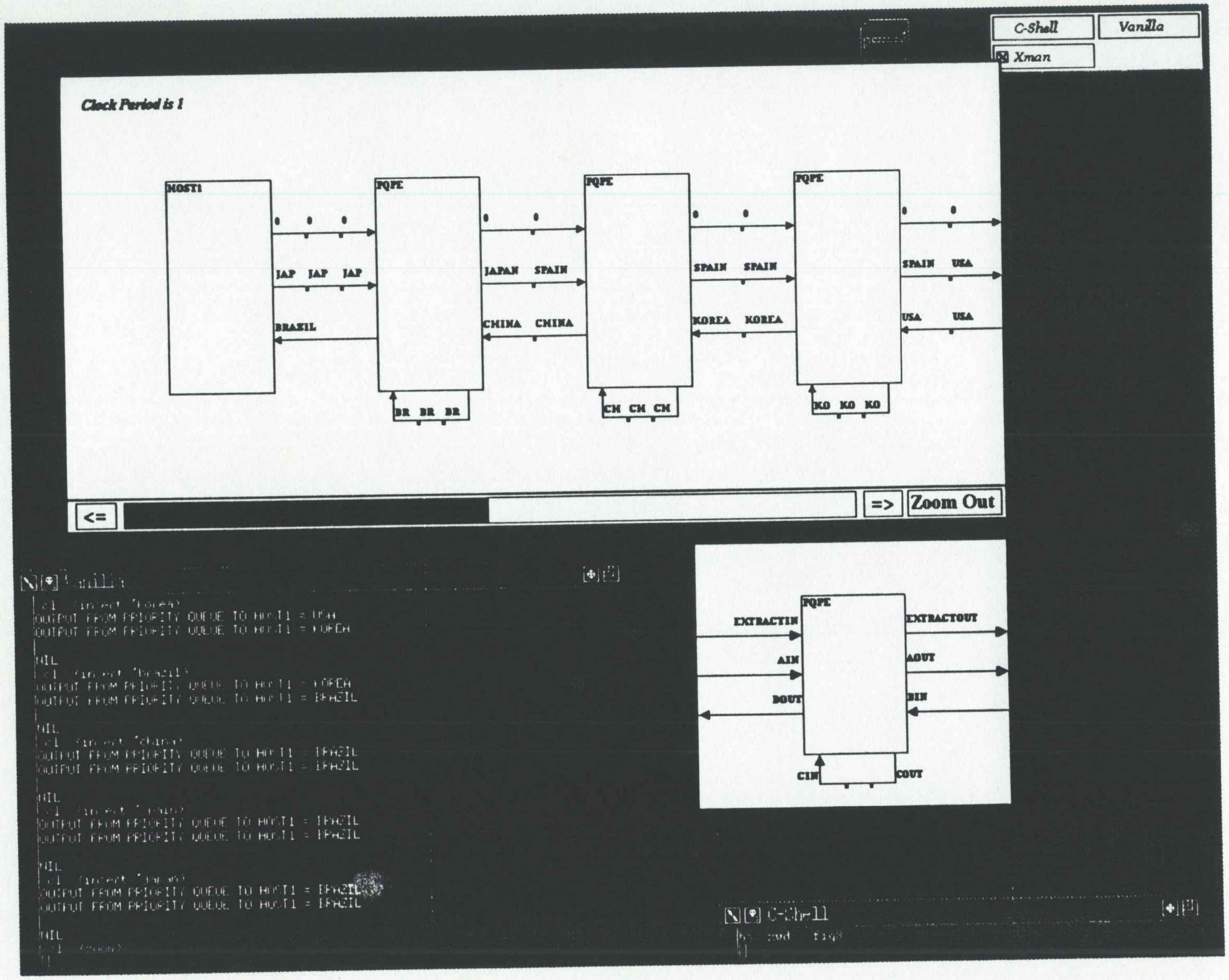


```
 c1  (insert 'spain)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 c1  (insert 'japan)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
 c1  (extract 6)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = CHINA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = JAPAN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = SPAIN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = ZZZ

NIL
 c1  (initsys)

"SYSTEM INITIALIZED"
 c1  (slowdown 2)

"SYSTEM SLOWED DOWN BY A FACTOR OF 2"
 c1  (showelem)

NIL
 c1  
```

```
 % cmd  fig
```

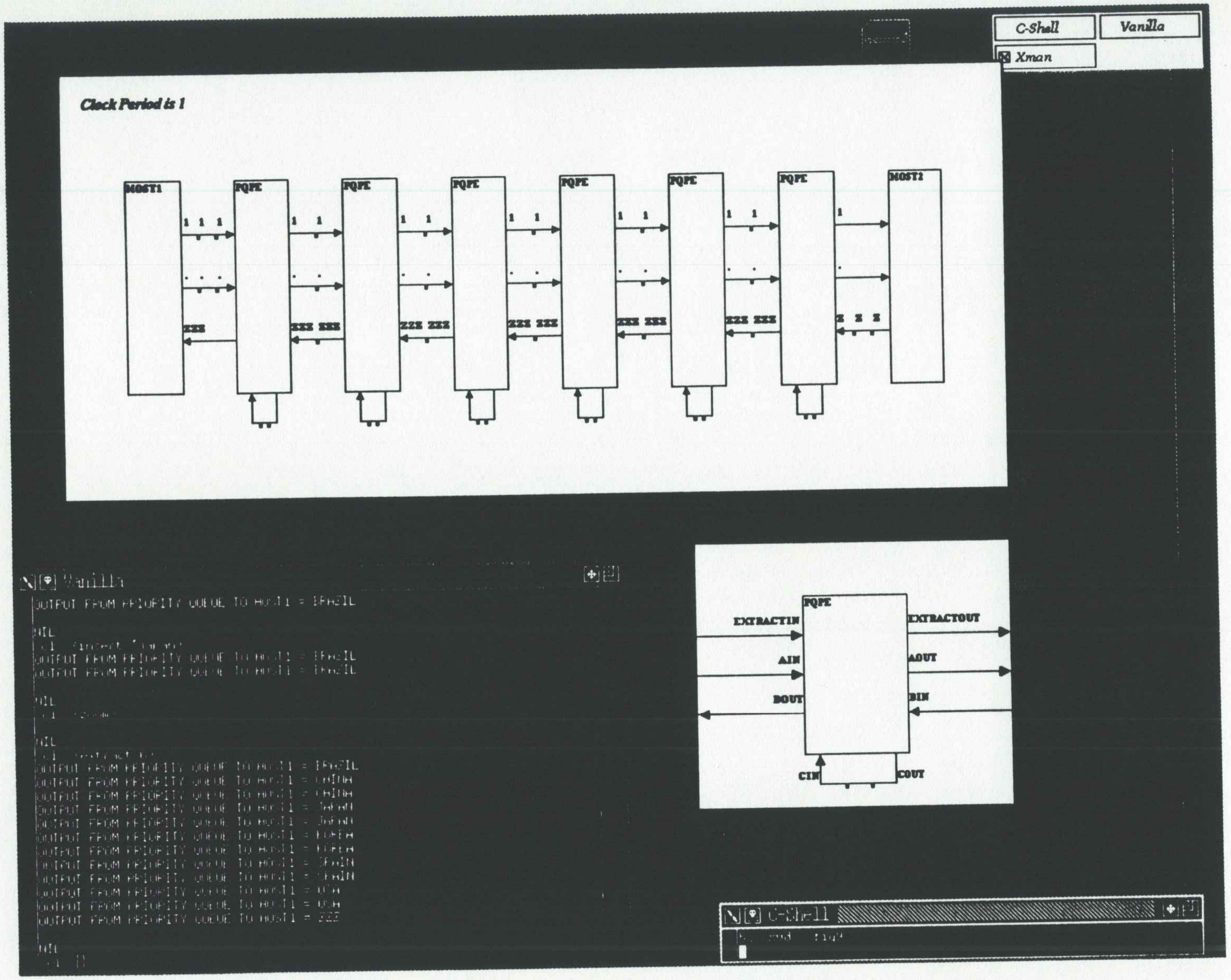
Screen 6

15



```
cl  (insert 'korea)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA

NIL
cl  (insert 'brazil)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl  (insert 'china)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl  (insert 'spain)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl  (insert 'japan)
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
cl  (zoom)
```

Screen 7



```
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
<1> [illegible]
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL

NIL
<1> [illegible]

NIL
<1> [illegible]
OUTPUT FROM PRIORITY QUEUE TO HOST1 = BRAZIL
OUTPUT FROM PRIORITY QUEUE TO HOST1 = CHINA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = CHINA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = JAPAN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = JAPAN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = KOREA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = SPAIN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = SPAIN
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = USA
OUTPUT FROM PRIORITY QUEUE TO HOST1 = ZZZ

NIL
<1> []
```

Screen 8