# COMPILING DIGITAL NETWORKS FOR PARALLEL SIMULATION

by

Eric McCaughrin

Memorandum No. UCB/ERL M94/33

11 May 1994

# COMPILING DIGITAL NETWORKS FOR PARALLEL SIMULATION



by

Eric McCaughrin

Memorandum No. UCB/ERL M94/33

11 May 1994

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# COMPILING DIGITAL NETWORKS FOR PARALLEL SIMULATION



by

Eric McCaughrin

Memorandum No. UCB/ERL M94/33

11 May 1994

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

For high-level verification, the design implementation can be changed if the functional behavior is preserved. This paper discusses techniques for re-structuring circuits at the logic level to increase simulation performance on massively parallel SIMD computers. This approach is different from previous work, which has attempted to improve parallel simulation performance by writing faster simulators. We have written a simulator for the MasPar MP-1 to measure speed-up. While some of the logic synthesis techniques are MP-1 specific, most are sufficiently general that they can be applied to other massively parallel architectures.

# Acknowledgments

I would like to thank my research advisor, Prof. A. Richard Newton, for his guidance and encouragement. My discussions with him focussed my research on the critical issues of the project and kept me from becoming side-tracked. I also thank Professors Alberto Sangiovanni-Vincentelli and David Culler for their suggestions and timely reading of this work. This research was sponsored by the Semiconductor Research Corporation under contract number 93-DC-008 and the Defense Advanced Research Projects Agency under the grant JFBI90-073 *Design and Prototyping of Hard Real Time Systems*. I am grateful for their financial support. I also thank Digital Equipment Corp. for providing excellent computing facilities at the CAD lab, MasPar Computer Corporation for access to their MP-1 and the assistance of Tom Blank of MasPar. Brad Krebs and everyone else on the CAD lab staff were also very helpful and are the best system administrators I have ever met.

Special thanks to Gary York of Cadence Design Systems for many useful discussions on simulation and for helping simulate the benchmark circuits on Verilog. Rick McGeer also provided an enormous amount of information on synthesis and other CAD issues and I thoroughly enjoyed the discussions I had with him. Finally, Kia Cooper and Heather Brown have been incredibly helpful and I would have gotten completely lost at Berkeley without them.

I especially want to thank my friends in Berkeley for their emotional support and for broadening my interests – Janet Pederson (and everyone else at the Chateau Co-Op), Cecilie Birner for some incredible bike rides, and everyone in the Hiking Club. I also wish to thank long time friends from RPI: George Kyriazis, Jean Bordes, Crash Dominus, and Chris Widmann. They are not only great companions, but I learned more from them than I did in the classroom.

I could never repay my family for their love and support over the years. I know that my education has been one of the most important goals of my parents and it thrills me to see them so happy that I am achieving that goal. Having a close family provides so much emotional support and I thank all of them: my parents Greta and Craig and my sisters Natalie, Rebekah, and Lara.

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Simulation is one of the most important VLSI CAD tools. Chip designers rely on simulation to avoid fabricating expensive prototypes. Simulation is also very extremely expensive. Today's big designs require weeks of simulation. A large amount of research has looked at ways of reducing the cost of simulation. The two main areas that have been explored are specialized hardware for simulation and behavioral simulation.

One example of specialized hardware is the Field Programmable Gate Array (FPGA). An FPGA can be programmed with a user-defined logic function with several hundred FPGA's stored in a single chip. While FPGA's have extremely good simulation performance, synthesizing a circuit for FPGA's and wiring the chips takes a lot of time. So, FPGA's are mostly used to verify a design only when it has neared completion.

Other hardware platforms that have been explored include parallel computers. Regardless of the parallel computer architecture (shared memory, massively parallel SIMD, etc.) the major hurdle for parallel simulation has been the large amount of interprocessor communication. Many real-world circuits have insufficient parallelism to overcome the communication overhead. Modest performance improvements have been reported with parallel simulation, but not enough to make it cost-effective.

Given the cost of specialized hardware, behavioral simulation has become increasingly popular. With this methodology, the circuit is specified at a high level. Logic synthesis uses the high level description to synthesize a gate-level design with a given area/speed criterion. Simulating the design at the behavioral level can be considerably faster than gate-level simulation. The high-level description also helps designers conceptualize the circuit.

Behavioral simulation is excellent for applications that are not time-critical (like ASIC's). For time-critical applications (especially microprocessor data paths) the design has few high-level abstractions. In some cases, logic synthesis does not produce the optimum result, so the design is done at the gate-level. In neither case is behavioral simulation very useful.

This paper describes an alternate simulation methodology for these situations. Logic synthesis can be used to synthesize a circuit not for a chip or FPGA implementation, but for simulation on massively parallel computers. This paper describes a logic synthesis tool that optimizes a circuit for parallel simulation by synthesizing parallelism and grouping logic functions in a way that reduces interprocessor communication. To test various synthesis algorithms, a parallel simulator was written for the MasPar MP-1, a massively parallel SIMD computer. On gate-level designs, this methodology outperformed Verilog-XL[TM91] running on a SparcStation-2 by more than a factor of 20. Without synthesis, the MP-1 simulator showed little improvement over Verilog-XL.

The synthesis tool runs under the Sequential Interactive Synthesis System (SIS)[BR+87]. SIS represents the circuit as a multilevel set of logic equations. Various subroutines are provided for manipulating the design. Logic minimization, technology mapping, and decomposition tools are also included.

Although SIS supports sequential circuits, only combinational ones were simulated. In some cases, sequential circuits were made combinational by loop breaking. Currently, SIS supports only single-output logic functions. Because better simulation results can be obtained with multiple-output functions, changes were made to SIS to provide this capability.

The report is organized as follows: Chapter 2 gives some background on circuit simulation. Chapter 3 describes previous efforts in parallel simulation. Chapter 4 is an overview of the MasPar MP-1 computer. In Chapter 5, the simulator written for the MP-1 is discussed. Synthesis techniques for exploiting some features of the simulator are presented in Chapter 6. Chapter 7 lists the experimental results. These results will show that logic synthesis greatly

improves parallel simulation performance. Chapter 8 goes into greater detail of the simulation code. Parallel simulation on the Connection Machine-5, conclusions, and future work are presented in Chapters 9 and 10.

# Chapter 2

# Circuit Simulation Background

Digital systems can be modeled at four different levels: circuit, switch, gate, and behavioral. Different levels trade-off accuracy for simulation speed. Mixed-mode simulators combine different levels of simulation into a single program.

Circuit-level simulation is the most accurate and the most expensive. It models circuits at the transistor, resistor, and capacitor level. Because it is so slow, circuit-level simulation is only possible on designs with fewer than 10,000 transistors. This limit is much smaller than today's designs, which have more than a million transistors. Examples of circuit-level simulators are *SPICE*[Nag75] and *CAzM*[Erd89].

Switch-level simulation models transistors as simple switches. Examples of switch-level simulators are ESIM[Ter83] and MOSSIM[Bry84]. COSMOS[Bry87] preprocesses transistor networks into a set of boolean equations which are then simulated. Chapter 3 will discuss a parallel version of COSMOS.

Gate-level simulation represents the circuit with boolean logic equations (or gates). Timing information can be assigned to gates. Fixed delay simulation is also possible, which allows faster simulation when timing information is unimportant. Examples of gate-level simulators are HILO[Gen85] and THOR[SB87].

While gate-level simulation has been a very popular way of modelling circuits, its performance has not kept pace with larger designs. As a result, RTL and behavioral-level simulation are becoming more popular.

At the RTL and behavioral-level, high-level constructs, like looping and arithmetic operations, can be incorporated in a circuit description. These high-level abstractions allow much faster simulation. Behavioral descriptions are described with high level languages (like

VHDL), which can be compiled and executed like any other computer language. Once the designer is satisfied with the design, logic synthesis can produce a gate-level circuit from the behavioral description.

For switch-level and higher levels of abstraction, there are generally two simulation approaches: oblivious and discrete-event simulation. For each time step, oblivious simulators evaluate every gate. Experiments have shown, however, that for each new input vector, only a very small percentage of gates have a change in output. Discrete-event simulation exploits this behavior by using a queue to schedule simulation only on gates whose input has changed. When a gate is simulated, it calculates the new output, and if the output is different, all the gates on the fanout net are added to the scheduling queue. When there are no more events in the queue, the primary outputs are shipped out and the next input vector is processed.

Another simulation classification is whether the simulator is "interpretive" or "compiled". An interpretive simulator represents the circuit with a data structure containing the circuit elements and their interconnections. For each evaluation on a circuit element, the simulator must read this data structure. On uniprocessors, compiled simulators are much faster because they eliminate the data structure altogether. Instead, the data structure is incorporated in a computer-generated program. Compiled simulation is not possible on data parallel machines like the MP-1 because each processor must execute the same program.

# Chapter 3

# Previous Work in Parallel Simulation

Because simulation is such an important part of VLSI design, there has been much simulation research. This chapter summarizes previous efforts in parallel simulation at the switch and logic levels.

## 3.1 Measuring Parallelism in Circuits

Several papers have reported calculations to determine whether there is sufficient parallelism in real circuits to make parallel simulation worthwhile.

Bailey and Snyder [BS85] measured the amount of parallelism in circuits using a switch-level simulator. Parallelism was defined as the average number of events executed in a timestep. An event occurs for transitions from 0 to 1, 1 to 0, and to an indeterminate state ("X").

This metric was applied to two large circuits. One was a RISC microprocessor and the second was an IIR digital filter. The filter incorporated a 16 x 16 multiplier, 32-bit ripple adder, 9-bit ripple counter, a 17 stage, 16-bit shift register, four 3 stage, 16-bit shift registers, and a PLA. The remaining circuits were computer generated: two multipliers, a shift register, and a 4-to-16 decoder.

For each circuit, the average parallelism, the maximum parallelism, and the percentage of parallelism was measured. The percentage of parallelism is the percentage of nodes that are changing in a time step (or the average parallelism divided by the number of nodes in the circuit). These results are given in Table 3.1.

The effect of circuit size on parallelism was determined by generating larger instances of the computer generated circuits. The results are summarized in Tables 3.2-3.4. The

amount of parallelism did increase in larger circuits, but the percentage of parallelism decreased. Other authors have measured the amount of parallelism in small circuits to extrapolate the amount of parallelism in large circuits [WF87]. Because the percentage of parallelism decreases, such predictions may be incorrect.

| Circuit | Transistors | Nodes | Percent Parallelism | Average Parallelism | Maximum Parallelism |
|---|---|---|---|---|---|
| 32-bit RISC | 24,068 | 10,500 | 0.06% | 6.3 | 140 |
| IIR Digital Filter | 27,360 | 14,399 | 0.04% | 6.4 | 280 |
| 8x8 Baugh-Wooley Multiplier | 2,162 | 1,083 | 0.26% | 2.8 | 22 |
| 8x8 Booth Multiplier | 2,013 | 1,088 | 0.31% | 3.4 | 41 |
| 8-stage, 16-bit Shift Register | 1,536 | 1,048 | 2.4% | 25 | 69 |
| 4-to-16 Decoder | 208 | 110 | 2.9% | 3.2 | 11 |

Table 3.1: Circuit Parallelism [BS88]

| Size (n x n) | Transistors | Percent Parallelism |
|---|---|---|
| 4 x 4 | 594 | 0.54% |
| 8 x 8 | 2,162 | 0.26% |
| 16 x 16 | 8,178 | 0.18% |
| 24 x 24 | 18,034 | 0.16% |
| 32 x 32 | 31,730 | 0.15% |

Table 3.2: Baugh-Wooley Multiplier with Random Inputs [BS88]

| Size (n x n) | Transistors | Percent Parallelism |
|---|---|---|
| 8 x 8 | 2,013 | 0.31% |
| 16 x 16 | 6,867 | 0.21% |
| 24 x 24 | 14,665 | 0.19% |
| 32 x 32 | 25,407 | 0.17% |

Table 3.3: Booth Multiplier with Random Inputs [BS88]

| Bits | Stages | Transistors | Percent Parallelism |
|---|---|---|---|
| 8 | 8 | 768 | 2.5% |
| 16 | 4 | 768 | 2.4% |
| 16 | 8 | 1,536 | 2.4% |
| 16 | 16 | 3,072 | 2.4% |
| 32 | 8 | 3,072 | 2.4% |

Table 3.4: Shift Register with Random Inputs [BS88]

One concern with these measurement is the 0.1 ns timestep. A larger timestep yields greater parallelism, but may affect simulation accuracy. By using a unit-delay model, the amount of parallelism is up to an order of magnitude greater than the values obtained from the 0.1 ns timestep.

Kravitz et al [KBR89] also measured parallelism with a switch-level simulator, called CM_COSMOS. They determined the effective parallelism to be from 100 to 3000.

CM_COSMOS is the parallel version of the COSMOS switch-level simulator. COSMOS preprocesses MOS circuits into equivalent boolean formulas. Because the preprocessor and simulator work at the boolean level, their work is applicable to logic simulation. CM_COSMOS runs on the Connection Machine-2 (CM2). Like the MasPar MP-1, the CM2 is a

massively parallel SIMD architecture, but has only 1-bit processors[Hil86].

CM_COSMOS supports 3-valued logic -- 1, 0, and X. The preprocessor partitions the circuit into channel-connected subnetworks. Subnetworks are a function of the inputs and the previous output. They are repeatedly evaluated until a steady state is reached. Each subnetwork is compiled into an equivalent boolean model of AND and OR operators. Operators are mapped to processors. Each processor has two phases of operation -- a compute step where the result of the operator is found and a communication step where the node output is sent across the network to the next operator. On a SIMD machine, the time required to send outputs over the network is proportional to the largest fanout. For this reason, CM_COSMOS reduces large fanout operators with fanout trees. Operators were limited to two fanouts.

Parallelism was defined as the average number of boolean operators that are evaluated concurrently. Two circuits were measured. One was an industrial bus controller and the other was a full custom data path circuit. Depending on the number of processors used, the effective parallelism ranged from 100 to 3000. Over 2 million boolean functions were evaluated per second.

## 3.2 Parallel Logic Simulators

Several approaches have been proposed for parallel simulation. The simplest is a parallel version of a unit-delay compiled mode simulator. This type of simulation uses no event queue; rather, every element is evaluated at each time step. [SB88] found the speed-up over the uniprocessor version to be 6-13 on a 16 processor machine. However, this method is generally not as effective as parallel event-driven simulation because most real circuits have very low activity rates.

Central Concurrency Control [PS88] is one type of parallel event-driven simulation. It manages the event queue from a central control processor. This method suffers significant communication bottlenecks because all the other processors must simultaneously interact with the control processor to send and receive events. When implemented on a general-purpose shared-

memory computer, [SB88] only achieved a maximum speedup of 2 using eight processors.

The Chandry-Misra algorithm[CM81] avoids the central event queue altogether. It partitions circuit elements into *Logical Processes* (LP's) that have their own event queues. LP's read time-stamped event messages on their inputs. When all the inputs on a circuit element are available, the output can be calculated. If the output has changed, a time-stamped message is sent. Each LP has its own local clock. The local clock is advanced when an output is calculable.

The algorithm has two phases of operation: the compute phase (as described above) and deadlock resolution. Deadlock occurs when every element is waiting for at least one input. Elements do not evaluate their output when at least one input is not known at the current local time. Deadlock is a direct result of the event-driven nature of the algorithm because outputs are not propagated when they do not change. Deadlock is resolved by finding the unprocessed event with the minimum time-stamp and updating the valid time of all inputs with no events to that time. Deadlock resolution is a major bottleneck. [SG89] found that 19-58% of the simulation time was spent resolving deadlocks in their benchmark circuits.

[SG89] measured the average concurrency in real circuits using the Chandry-Misra algorithm. Concurrency is defined as the number of elements evaluated in one cycle. All elements are evaluated in the same amount of time. Element evaluation can activate a whole new set of elements, which are evaluated in the following cycle. This metric assumes an infinite number of processors and ignores deadlock and communication costs. The results are summarized in Table 3.5. Ardent-1 is the vector control unit for the Ardent Titan supercomputer, H_FRISC is a RISC generated by the HERCULES high-level synthesis system, Multiplier is a 16x16 bit integer multiplier, and 8080 is a TTL board implementation of the 8080 microprocessor.

| Circuit | Representation | Element Count | Average Concurrency |
|---|---|---|---|
| Ardent-1 | gate/RTL | 13,349 | 107 |
| H_FRISC | gate | 8,076 | 111 |
| Mult-16 | gate | 4,990 | 45 |
| 8080 | RTL | 281 | 10 |

Table 3.5: Average Concurrency in Real Circuits [SG89]

Like the Chandry-Misra algorithm, the Time-Warp algorithm[JD85] also maintains a separate event queue and simulation clock at each circuit element. The clock is allowed to advance independently at each element. Time-Warp differs from Chandry-Misra in that if an input at a circuit element does not have a time stamp as recent as the local clock, the circuit element will be evaluated anyway without waiting for a new input to arrive. While this avoids the deadlock problem, it is possible that event precedence becomes lost.

When an element receives an event with a time stamp before the current local time, the element backtracks to a state before events were processed out of order and sends "Anti-events" to cancel spurious events. [A86] implemented a switch-level simulator using Time-Warp and obtained a speed-up of 4 over the uniprocessor version on a 6 processor computer.

Finally, Encore Computer Company implemented an event-driven functional simulator for the Multimax computer[W86]. They reported a speed-up of 3 on a 5 processor system. Adding more processors did not improve performance.

Chapter 4

# Design of the MP-1 Parallel Simulator

This chapter describes the MP-1 architecture and ways of implementing logic simulation on the MP-1.

## 4.1 Overview of the MasPar MP-1

The MP-1 is relatively slow compared to newer machines. Nonetheless, it is a good platform for testing synthesis strategies.

The MP-1 consists of a two dimensional array of processing elements (PE's). The system can accommodate from 1024 to 16384 PE's. The Array Control Unit (ACU) controls the PE's, sending instructions to each PE and handling serial operations. The PE array and ACU are collectively referred to as the Data Processing Unit (DPU).

### 4.1.1 The Processing Element

Each PE has a 1.8 MIPS processor, forty 32-bit registers, and 64 KBytes of RAM. Each PE is connected to its eight nearest neighbors and to a global router which allows a PE to communicate with any PE in the system. The processor has a 4-bit integer ALU and accesses local memory through a 4-bit bus.

### 4.1.2 The Array Control Unit

The ACU is a 14 MIPS processor with thirty-two 32-bit registers and 128 KBytes of data memory. The ACU broadcasts instructions to PE's. All PE's execute the same instruction

stream, but have independent storage through their local memory. The ACU also computes addresses and scalar data values, issues control signals to the PE array, and performs diagnostics on the system.

### 4.1.3 The Data Network

PE's have two methods of communication: the X-net and the global router. The X-net connects each PE to its eight nearest neighbors in the directions north, south, east, west, northeast, northwest, southeast, and southwest. The system has toroidal wrapping, so PE's on the edge of the physical array still have eight neighbors.

The global router is more flexible because it allows a PE to communicate with any PE in the system. Sometimes the global router is slower than the X-net.

PE's are organized into 4x4 clusters. While the global router can communicate with all clusters simultaneously, only one PE per cluster can be accessed at a time. One outgoing and one incoming message per cluster is permitted simultaneously. A message within a cluster consumes both the incoming and outgoing channels. When more than one message in a cluster attempts to use a channel, the messages are serialized.

### 4.1.4 Programming the MP-1

MasPar provides a parallel versions of Fortran and C. Both include new data types and control structures for parallel operations on data. The extensions to C will be discussed in Chapter 8. A function library is included for transferring data between the DPU and the UNIX front-end. The DPU and the front-end may communicate asynchronously. Each can perform other tasks while the other transfers data.

## 4.2 Simulation Strategies for SIMD Machines

An effective synthesis algorithm derives from the design of a fast simulator. Clearly, the synthesis algorithm must partition the gates in the circuit among the processor nodes in a way that minimizes communication between partitions. Furthermore, each processor must evaluate its set of gates in an efficient manner.

### 4.2.1 SIMD Simulation

For compiled simulation, a technology mapper can map logic functions to logic opcodes supported by the computer running the simulation. Unfortunately, compiled simulation is not possible on a SIMD computer like the MP-1 because all processing elements share the same instruction stream. Instead, each processor must store its set of logic functions in its private data memory. The simulation code consists of a small simulation loop that reads the logic function from memory each time an output needs to be calculated.

One way to represent logic functions is by encoding gate types. The simulation loop "switches" on the gate type as follows:

```
switch (node.type) {
  case AND:
    node.output = node.input0 & node.input1;
    break;
  case OR:
    node.output = node.input0 | node.input1;
    break;

  etc.
}
```

While this method is memory efficient, it runs slowly on a SIMD computer. Since there is a single instruction stream, *case* blocks execute serially. It is better to represent the logic function as a truth table. Calculating an output is a simple matter of accessing an element in an array:

$$node.output = node.table[node\_input_0\ OR\ node\_input_1\ OR\ ...\ node\_input_n];$$

For this method to work, the gate input bits must be arranged to not overlap when OR'ed together.

Each processor must evaluate a set of gates. The fastest possible calculation is by collapsing the subnetwork into a single combinational logic block (CLB). The entire subnetwork can be calculated by a single table look-up. Given that a processing node can address $k$ bits of local memory, the synthesis algorithm must partition the gates such that each CLB has fewer than $k$ inputs.

### 4.2.2 The MP-1 Simulator

Our MP-1 simulator is interpretive, event-driven, and handles only combinational circuits. Each processor performs the following steps per CLB:

```
Foreach input vector {
    OR together the CLB inputs to form the table index
    Foreach CLB Fanout {
        Look-up the output from the table belonging to the fanout
        If the output has changed, send it to the appropriate CLB
    }
}
```

The simulator allows multiple-output CLB's with multiple fanouts. Each fanout sends one or more bits, which must be ordered properly so that they are OR'ed correctly with other fanouts in the correct sequence. Since bit extraction and re-ordering is too slow, fanouts have separate look-up tables. Each fanout may use only a portion of the output variables. As such, the simulator checks for a change in output for each fanout.

The MP-1 has three kinds of parallelism. First, it provides bitwise parallelism at the processor level by evaluating a set of gates through a table look-up. The second type is levelized parallelism. CLB's are evaluated in topological order from the inputs to the outputs. For a given input vector, all CLB's at a level are evaluated simultaneously. Third, the simulator pipelines the input vectors. While processors at level $l_n$ simulate input vector $i_m$, processors at level $l_{n-1}$ simulate input vector $i_{m+1}$.

Because the synthesis algorithm changes the circuit, timing analysis cannot be performed. No timing information is associated with CLB's. Although simulation accuracy is reduced, it allows us to avoid the event precedence problem tackled by the Time Warp and Chandry-Misra algorithms.

Chapter 5

# Synthesis Algorithms

This chapter gives two synthesis algorithms for improving parallel simulation. Both are divided into the following steps:

1. Gates with more than $k$ fanins are split using AND-OR decomposition.
2. The resulting network is grouped into multiple-output CLB's.
3. CLB's with a disproportionate number of fanouts are duplicated.
4. A placement algorithm assigns CLB's to processors in a way that exploits the communication network of the target machine.

The difference in the two algorithms comes in step 2 -- the grouping step. The first approach uses a bin packing algorithm derived from FPGA synthesis. The second preserves spatial locality when combining gates into CLB's.

## 5.1 Decomposition of Infeasible Nodes

If the maximum size of a CLB is $k$ inputs, then nodes with more than $k$ inputs must be split in order to fit in a CLB. There are a variety of decomposition methods. AND-OR decomposition was used for both synthesis algorithms.

Infeasible nodes can be split using any partition of the inputs. Infeasible nodes are represented as a sub-graph of AND and OR nodes, which are recursively decomposed. For example, a 3-input function $z = ab + ac + bc$ can be decomposed into 2-input functions $v=ab$, $w=ac$, $x=bc$, $y=v+w$, and $z = y+x$.

## 5.2 Algorithm 1 -- Bin Packing Approach

The MP-1 simulator is an implementation of a table look-up (TLU) architecture. Much work has already been done on logic synthesis for TLU in the area of Field Programmable Gate Arrays (FPGA's). An FPGA is a programmable chip that implements a user-specified logic function of $k$ inputs. The Xilinx(3000) is a typical FPGA, with $k$=5[Xil].

The goal of FPGA synthesis is to find the minimum number of CLB's to represent a circuit. A popular FPGA synthesis algorithm was modified to solve the grouping problem in step 2. One major change is that the MP-1 allows CLB's to have up to 16 outputs. FPGA's are limited to one or two outputs.

The bin packing algorithm[GJ79] is very effective at solving the FPGA synthesis problem. The goal of bin packing is to find the minimum number of fixed width bins into which a set of variable-sized boxes can be packed. The bins represent CLB's and the boxes are logic functions.

There are several algorithms for solving the bin packing problem. The First Fit Decreasing (FFD) algorithm starts with an empty list of bins and orders boxes in decreasing order of their size. As each box is visited, it places the box in the first bin in which it fits. If the box does not fit in any of the bins, a new bin is created containing only that box. This bin is added to the end of the list of bins. The Best Fit Decreasing (BFD) also visits boxes in decreasing order, but tries to place a box in the bin that leaves the most inputs unused.

The first synthesis algorithm used a modified version of the BFD algorithm. It tried packing each gate into the bin that added the fewest number of inputs to the bin. For instance, if a bin already uses signals A, B, C, then adding the gate B AND D would only add one input (D). A gate was not added to a CLB if it caused the resulting circuit to become acyclic. The gates were visited not in decreasing order, but in breadth-first order, starting from the primary inputs and working toward the primary outputs. Experiments showed this ordering gave better results than visiting gates by size.

The problem with the FPGA approach is that it is optimized for reducing the total number of CLB's before fanout optimization. In step 3, CLB's with a large number of fanouts are duplicated. Because functions were added to CLB's without regard to communication costs, fanout optimization caused an explosion in circuit size. The second algorithm combines gates that are physically close together in the circuit. As a result, the circuit does not grow as large during fanout optimization.

## 5.3 Algorithm 2 -- Grouping with Spatial Locality

The second algorithm is a greedy heuristic that merges functions into larger, multiple-output CLB's until the number of inputs to the CLB reaches a user-specified limit $k$ (usually determined by the memory capacity of the simulator and the maximum number of fanouts). The algorithm proceeds as follows. The nodes are visited in topological order, from the second level to the outputs. For each node $v$, all possible pairs of nodes ("input nodes") fanning out to $v$ are examined. If $v$ and any pair of input nodes has a total unique number of inputs less than $k$, the group of nodes are merged into a "super-node" provided the network remains acyclic. If $v$ cannot be merged with any pair of nodes, it is combined with the input node that produces the fewest number of inputs while keeping the network acyclic. Failing this, $v$ is not combined with any nodes.

The following example demonstrates the algorithm when $k=3$. Given the sample network in Figure 5.1, we begin at the second level with node 3. The input nodes are 1 and 2. The number of inputs to nodes 1, 2, and 3 is less than $k$, so they are combined. Node 5 is also at the second level and it is combined with node 4, its only fanin node. Figure 5.2 shows the results after the second level pass.

Visiting node 6 next, we cannot combine its two input gates because that would result in 4 inputs, which is too many. However, node 6 can be combined with either {1,2,3} or

{4,5} -- we arbitrarily pick the former. Node 8 is merged into {4,5}.

All that is left is node 7. While merging node 7 into {4,5,8} would not exceed $k$, it would create a loop. Placing node 7 into {1,2,3,6} would exceed $k$, so node 7 is not merged. Figure 5.3 shows resulting circuit.

Figure 5.4 gives pseudocode for the grouping algorithm. In the step following grouping (duplication), nodes with too many fanouts are split. One variation on this grouping algorithm is to limit the number of output variables a CLB may accumulate before the duplication step. For some circuits, this reduced the overall number of CLB's. Another variation is to limit the number of messages a CLB may receive from fanin CLB's. This can improve load balancing by ensuring that no processing element has to OR together an excessive number of input vectors.

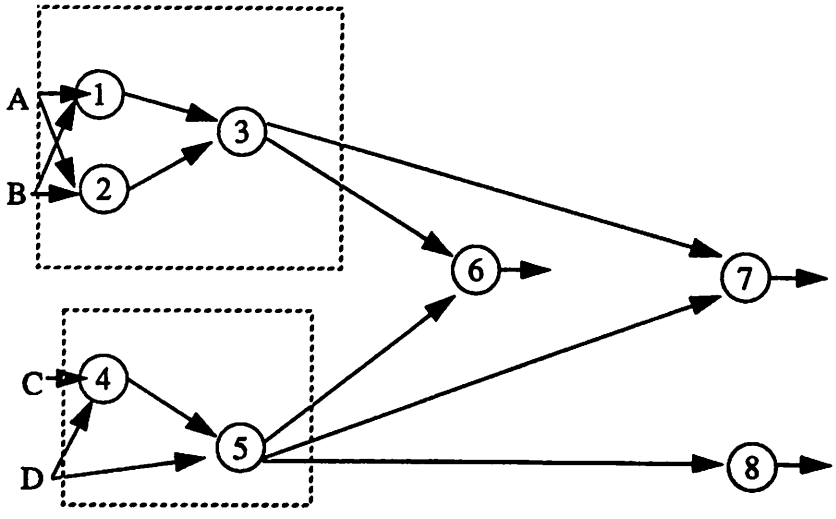


Figure 5.1: Sample Circuit

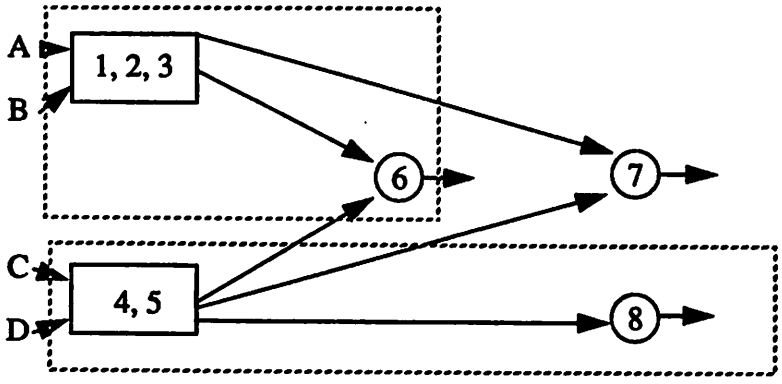


Figure 5.2: Sample Circuit After Second Level Pass

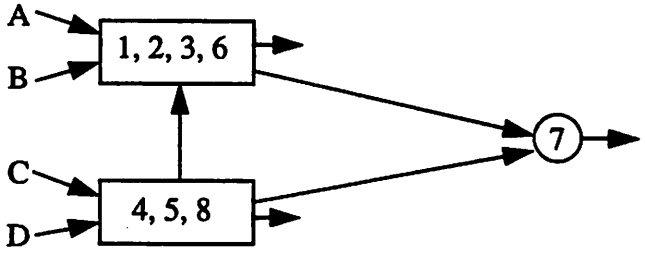


Figure 5.3: Grouped Circuit

```
for(level=1; level<num_levels; level++) {
   foreach(v, nodes at level) {
      for(i=0; i<num_fanin(v); i++) {
         current = fanin(v, i);
         for(j=i+1; j<num_fanin(v); j++) {
            add = fanin(v, j)
             if (num_fanin(current, add, v) <= k &&
                dag(current, add, v)) {
                   merge_nodes(current, add, v);
                   Mark v as assigned
             }
         }
      }
   }
   foreach(v, unassigned nodes at "level") {
      If possible, merge v into a fanin node of v that produces the fewest
      inputs (less than k) while keeping the circuit acyclic
   }
}
```

Figure 5.4: Grouping Algorithm Pseudocode. The function *num_fanin* returns the total unique number of signals coming into its list of gates and *merge_nodes* combines a list of gates into a single logic function. The *i'th* fanin of *v* is returned by *fanin(v, i)*. The depth of the network is *num_levels* and *k* is the fanin limit of a gate.

## 5.4 Function Duplication

Step 3, function duplication, moves from the outputs to the inputs, duplicating CLB's that have too many fanouts. Figure 5.5 shows the sample network after duplication. Without this step, the computational load is not well distributed as some processors must send a substantially large share of the fanouts. To maximize parallelism, each CLB should have a single fanout. This is impractical for most circuits because the number of CLB's would be too large. Instead, there is a tunable parameter, *R*, for selecting the maximum number of fanouts, which ranged from 2 to 4 in our benchmarks. Table 5.1 gives the number of CLB's for different values of *R* for circuits taken from the ISCAS benchmark suit. The circuit *s38417* was originally sequential, but only the combinational portion was used. "NA" indicates there was insufficient memory to expand the circuit. All circuits in the benchmark suit are gate-level descriptions.

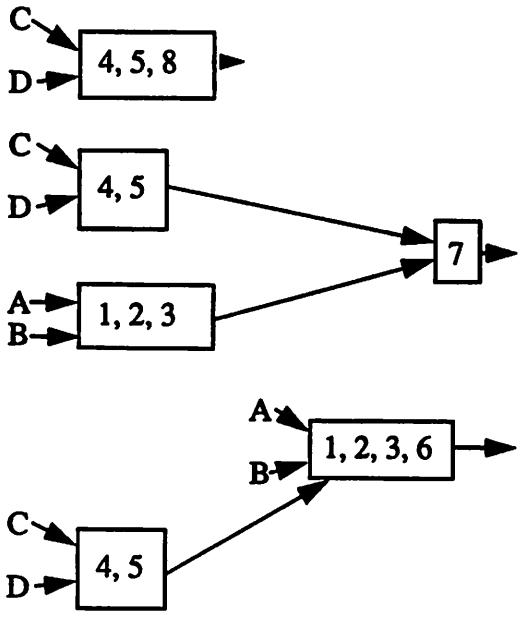


Figure 5.5: Sample Circuit After Duplication

| Circuit | Gates | R=1 k=14 | R=2 k=13 | R=3 k=13 | R=4 k=14 |
|---|---|---|---|---|---|
| c432 | 160 | 9459 | 315 | 104 | 64 |
| c499 | 202 | 1753 | 215 | 147 | 94 |
| c1908 | 880 | 5388 | 426 | 189 | 135 |
| c2670 | 1161 | 2190 | 393 | 289 | 239 |
| c3540 | 1667 | NA | 1766 | 604 | 431 |
| c5315 | 2290 | 13033 | 1473 | 962 | 720 |
| c6288 | 2416 | NA | 1126 | 443 | 313 |
| c7552 | 3466 | 16216 | 1539 | 803 | 589 |
| s38417 | 22397 | NA | 10679 | 5779 | 2960 |

Table 5.1: CLB Count

Table 5.2 compares the CLB count of the two synthesis algorithms. All circuits used *k=13, R=2* except for s38417, which used *k=12, R=4*. These were the parameters used for the actual simulation. The data show that bin packing performs very well on small circuits whereas the second algorithm generally performs better on the larger ones. The second algorithm also has considerably shorter runtime.

| Circuit | Gates | Algorithm 1 | Algorithm 2 |
|---|---|---|---|
| c432 | 160 | 136 | 315 |
| c499 | 202 | 76 | 215 |
| c1908 | 880 | 403 | 426 |
| c2670 | 1161 | 647 | 393 |
| c3540 | 1667 | 2759 | 1766 |
| c5315 | 2290 | 2047 | 1473 |
| c6288 | 2416 | 3742 | 1126 |
| c7552 | 3466 | 2946 | 1539 |
| s38417 | 22397 | 2867 | 2960 |

Table 5.2: Comparison of CLB Counts

## 5.5 Placement

The MasPar MP-1 has both a global and local communication network. The local network (X-net) connects each PE to its eight nearest neighbors in the directions north, south, east, west, northeast, northwest, southeast, and southwest.

The global network allows a PE to communicate with any other PE in the system. PE's are organized into 4x4 clusters. While the system can communicate with all clusters simultaneously, only one PE per cluster can be accessed at a time. One outgoing and one incoming mes-

sage per cluster is permitted simultaneously. A message between two PE's within a cluster consumes both the outgoing and incoming channel.

Despite several attempts to utilize the X-net, relying exclusively on the global network gave better results. The problem with the X-net is that all processors must communicate in the same direction simultaneously. One processor cannot transmit *north* while another transmits *south*. It is also difficult to position CLB's so that they border all their fanin and fanout CLB's.

Figure 5.6 gives the placement algorithm that was used. It attempts to assign CLB's to processors in a way that evenly distributes the communication load among PE clusters. When more than one PE in a cluster attempts to send or receive data, the messages are serialized. If the number of inputs to a cluster is $I$ and the number of outputs is $O$, $MAX(I, O)$ is the number of *pending messages*. The algorithm greedily picks clusters with the fewest pending messages.

The placement algorithm assumes CLB's have roughly equivalent activity rates. If some CLB's have significantly higher activity rates, communication is not evenly distributed. It is possible to record activity rates at run-time, but the data can be input dependent.

```
Initialize C.inputs = C.ouptuts = 0 for all clusters C
foreach CLB F {
   C = cluster where MAX(C.inputs, C.outputs) is the minimum
   Place F in C
   C.inputs = C.inputs + F.fanins
   C.outputs = C.outputs + F.fanouts
}
```

Figure 5.6: Placement Algorithm

Chapter 6

# Experimental Results

To simulate a circuit using our technique, the following steps occur: 1. The circuit is compiled, 2. The compiled circuit is uploaded to the processing elements of the MP-1, and 3. The simulation is executed on a file of input vectors. The next two tables summarize the performance of step 3, the actual simulation.

## 6.1 Simulation Performance

Table 6.1 compares the simulation speed of the original and re-synthesized circuits. The figures were obtained by measuring the simulation time of 5,000 random vectors on a MasPar MP-1 with 8192 processing elements. The time required to display outputs was excluded. The circuit s38417 was originally a sequential circuit, but only the combinational portion was simulated. All synthesized circuits used $R=2, k=13$, except for s38417 which used $R=4, k=12$. All circuits were originally gate-level representations.

Table 6.2 compares the performance of our parallel simulator (with re-synthesized circuits) against Verilog-XL, a popular commercial simulator[TM91]. Verilog ran on a SparcStation 2 with 32 MBytes of physical RAM and a local disk. Table 6.3 summarizes the compilation and data upload times (steps 1 and 2). Both sets of figures were obtained on a DecStation 5000.

| Circuit | Gates | Unmodified (patt/sec) | Re-synthesized (patt/sec) | Speedup |
|---|---|---|---|---|
| c432 | 160 | 379.6 | 888.3 | 2.3 |
| c499 | 202 | 298.7 | 827.4 | 2.7 |
| c1908 | 880 | 208.4 | 863.7 | 4.1 |
| c2670 | 1161 | 270.2 | 836.6 | 3.1 |
| c3540 | 1667 | 222.3 | 537.6 | 2.4 |
| c5315 | 2290 | 213.1 | 669.1 | 3.1 |
| c6288 | 2416 | 242.1 | 736.9 | 3.0 |
| c7552 | 3466 | 233.5 | 698.3 | 3.0 |
| s38417 | 22397 | 32.2 | 397.5 | 12.3 |

Table 6.1: Performance Results

| Circuit | Gates | Verilog-XL (patt/sec) | MasPar (patt/sec) | Speedup |
|---|---|---|---|---|
| c432 | 160 | 735.3 | 888.3 | 1.2 |
| c499 | 202 | 526.3 | 827.4 | 1.6 |
| c1908 | 880 | 256.4 | 863.7 | 3.4 |
| c2670 | 1161 | 152.9 | 836.6 | 5.5 |
| c3540 | 1667 | 170.7 | 537.6 | 3.1 |
| c5315 | 2290 | 86.2 | 669.1 | 7.8 |
| c6288 | 2416 | 5.1 | 736.9 | 144.5 |
| c7552 | 3466 | 54.9 | 698.3 | 12.7 |
| s38417 | 22397 | 15.5 | 397.5 | 25.6 |

Table 6.2: Verilog vs. MP-1

## 6.2 Performance Characterization

The main simulation loop comprises a series of steps: 1. Reading the input vectors from disk, 2. Calculating CLB outputs, and 3. Transmitting outputs over the communication network. Table 6.3 summarizes the amount of time spent in each step.

| Circuit | Gates | Reading Inputs | Computation | Communication |
|---|---|---|---|---|
| c432 | 160 | 2.2% | 68.8% | 28.8% |
| c499 | 202 | 1.5% | 70.5% | 27.9% |
| c1908 | 880 | 1.9% | 69.1% | 28.9% |
| c2670 | 1161 | 6.3% | 66.7% | 27.0% |
| c3540 | 1667 | 2.7% | 47.3% | 49.9% |
| c5315 | 2290 | 4.8% | 59.2% | 36.3% |
| c6288 | 2416 | 2.1% | 64.9% | 33.1% |
| c7552 | 3466 | 6.0% | 61% | 33.0% |
| s38417 | 22397 | 2.5% | 86.5% | 10.7% |

Table 6.3: Performance Analysis

For smaller circuits, roughly one third of the simulation time was spent in the communication step. Previous studies in parallel simulation have concluded that the communication step is the major performance bottleneck for simulation. When using synthesis with simulation, there is no single bottleneck. There are a number of competing factors that affect simulation performance. The job of the circuit compiler is to choose a good balance between trade-offs for optimal simulation performance.

One important trade-off is parallelism vs. communication costs. An improvement in parallelism can reduce the amount of computation, but increase the amount of communication. The amount of parallelism is set by the number of inputs and outputs to the CLB. Each CLB has a

loop for OR'ing together its inputs and another loop for sending its outputs. Reducing these loops improves computation time, but also increases the size of the circuit, leading to more communication.

Another factor to consider is the CLB table size. Larger CLB's tend to produce smaller circuits, but can increase the activity level. With a greater number of inputs to the CLB, the likelihood of a change in the outputs is greater, leading to more events and higher communication rates. It is difficult for the compiler to adjust for activity rates because runtime data is required, which can be very dependent on the input vectors.

The settings that provided the best results on an 8k-processor MP-1 kept the number of CLB's fewer than 2048. The MP-1 global router is capable of communicating with each PE cluster simultaneously. On an MP-1 with 8k PE's, there are 512 clusters (each cluster has 16 PE's). Keeping the number of CLB's fewer than 2048 placed at most 4 CLB's in each cluster. The activity rates of the benchmark circuits were such that 4 CLB's sharing a single communication channel did not overwhelm the global router.

## 6.3 Effective Simulation Performance

Table 6.4 summarizes the MP-1 "overhead" -- the amount of time to compile a circuit and to upload the circuit information to the processing elements of the MP-1. The figures were obtained on a DecStation 5000.

The overhead costs are very important in the early part of the design cycle when many iterations are needed to debug a circuit. Prohibitive overhead can negate any increased performance from parallel simulation. A graph like the one shown in Figure 6.1 can be constructed from the data of Tables 6.2 and 6.3. With the higher overhead, parallel simulation becomes worthwhile for the circuit s38417 only when more than 100,000 vectors are simulated.

| Circuit | Gates | Compilation (sec) | Upload (sec) | Total (sec) |
|---|---|---|---|---|
| c432 | 160 | 15 | 44 | 59 |
| c499 | 202 | 16 | 33 | 49 |
| c1908 | 880 | 38 | 124 | 162 |
| c2670 | 1161 | 52 | 78 | 130 |
| c3540 | 1667 | 100 | 177 | 277 |
| c5315 | 2290 | 282 | 216 | 498 |
| c6288 | 2416 | 254 | 222 | 477 |
| c7552 | 3466 | 358 | 180 | 539 |
| s38417 | 22397 | 5939 | 607 | 6546 |

Table 6.4: Compilation and Upload Times

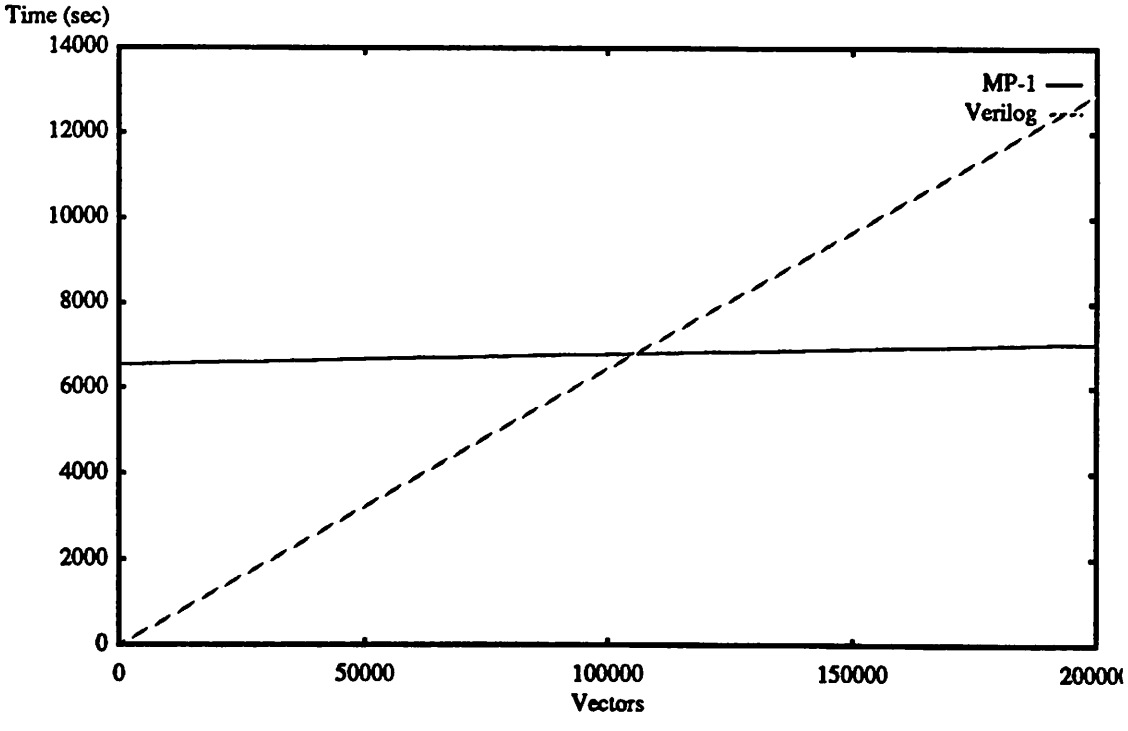


Figure 6.1: Overall Simulation Performance of *s38417*

Chapter 7

# Simulator Internals

In this chapter, the code for the simulator is examined. The full program listing is given in Appendix A. The code performs the following steps:

1. Reads the synthesized circuit and checks for errors
2. Uploads circuit information to the parallel processors
3. Simulates the circuit

A MasPar program has two portions: front-end code and DPU (Data Parallel Unit) code. The front-end code, which runs on the UNIX front end, exchanges data with the DPU and instructs the DPU to execute parallel routines. The parallel (DPU) code is written in MPL, a dialect of C that provides parallel extensions. For the steps listed above, step 1 is performed on the front-end. In step 2, the front-end uploads the data to the DPU. Step 3 is an MPL routine that executes on the DPU. This chapter covers step 3.

## 7.1 Programming in C on the MasPar

As described earlier, the DPU consists of a 2-dimensional array of processing elements (PE's) which are controlled by the Array Control Unit (ACU). The PE's and the ACU have independent address spaces. MPL adds a new type qualifier, *plural*, to differentiate between the two address spaces. The *plural* qualifier allocates storage in PE memory. Variables defined without the *plural* qualifier are "singular" and reside in ACU memory. All PE's store plural variables at the same address.

### 7.1.1 Pointers

Although the DPU has two address spaces, pointers may refer to either space. The following pointer combinations are possible:

- Singular pointer to singular data
- Singular pointer to plural data
- Plural pointer to plural data
- Plural pointer to singular data

A plural pointer to plural data only allows a PE to point to its own local memory. Accessing the memory of another PE is only possible through a communication function.

### 7.1.2 Control Statements

MPL has all the statements of traditional C. However, the plural form of these statements affect which PE's are *active*. Branching and control statements become plural when they refer to a plural expression.

The following *if* block is a plural control statement. The plural variable *iproc* is a special variable that stores a PE's processor number. Hence, iproc's value on processor 0 is 0, its value on processor 1 is 1, etc. This *if* block will increment plural variables *x* and *y* only on processors 0 through 9. The other processors remain idle for the duration of the block.

```
extern plural int iproc;
plural int x, y;

if (iproc < 10) {
      x++;
      y++;
}
```

Plural *if* statements with an *else* clause will make the active set of the *else* block the group of PE's that evaluate the control expression to false. The *if* block and the *else* block do not execute concur-

rently.

### 7.1.3 Looping Statements

Looping statements work similarly to *if* statements. Only PE's that evaluate the controlling expression to true are active. The loop will iterate until there are no more active PE's. The following code block demonstrates a parallel *for* loop.

```
plural int i;

for(i=0; i<iproc; i++) {
        /* code */
}
```

The variable "iproc" is defined as before -- it is a processor's unique identification number. On processor 0, the loop will not execute. Processor 1 performs the loop once, processor 2 performs it twice, and so on. It is important that loops are evenly distributed among processors to exploit parallelism. This loop is not well distributed because a machine with 16k processors will loop 16k times on the last processor. Meanwhile, low-numbered processors are idle.

## 7.2 Simulation Code

The main simulation loop will now be examined. The name of the function is *perform_simulation*, which is called from the front-end after the circuit has been uploaded to the PE array. Figure 7.1 gives the variables used.

| | |
|---|---|
| short num_po; | A function argument containing the number of primary outputs. |
| int d; | A function argument specifying a file pointer to the input vector file. |
| plural unsigned short<br>input_buffer[16][LEVELS]; | Queues the inputs to a CLB. The first range (0..15) is for specifying the network connection to the output of a preceding CLB. Each of these connections can have multiple bits. Clearly, the definition assumes no CLB has more than 16 inputs, which is not possible given the memory capacity of the PE's. The second range (0..LEVELS) stores inputs in a circular queue until they can be processed. LEVELS must be greater than the number of levels in the circuit, or data could be overwritten. |
| plural short<br>prev_output0, prev_output1, ...,<br>prev_outputn; | Stores the previously calculated output for a CLB. There is a separate variable for each output at the CLB. An output is only sent if it differs from the previous output. |
| plural unsigned short<br>prev_inputs[16]; | Stores the last inputs processed at a CLB. |
| plural unsigned char<br>input_vecs[PI_BYTES][VECS]; | Input vectors are read in blocks and stored in this array. VECS is the number of vectors in a block. Input vector bits are packed into bytes. PI_BYTES is the maximum number of bytes a CLB needs for its primary inputs. |
| plural struct node_struct {<br>short num_vectors;<br>short level;<br>Send<br>sender[MAX_FANOUTS];<br>unsigned short<br>table[FANOUTS][TABLE_SIZE];<br>char inputs;<br>} | This data structure contains the lookup tables and netlist information. The field *num_vectors* is the number of input patterns to be simulated, *level* is the CLB's level in the circuit, *sender* contains netlist information, and *table* is the CLB's lookup table. The field *inputs* does not represent the number of bits in the truth table but rather the number of other CLB's that fan-in to the CLB. |
| plural short vectors; | The number of input vectors that have been computed. |

Figure 7.1: Variables used in Main Simulation Loop

### 7.2.1 Initialization

The initialization portion of the simulation loop sets the number of calculated vectors to zero and clears the input buffer.

```
vectors = 0;
for(i=0; i<16; i++) {
        for(j=0; j<LEVELS; j++) input_buffer[i][j] = empty;
}
```

The identifier *empty* is a constant indicating no valid signal has been received by the input buffer. It means the current signal value does not differ from the previous value. The initialization phase also invalidates *prev_output0, prev_output1, ..., prev_outputn* to ensure the first calculated output will generate an event.

```
prev_output0 = prev_output1 ... = prev_outputn = empty;
```

### 7.2.2 Reading the Input Vectors

Following the initialization, the input vectors are read from disk. The input vector file is packed into bytes. The reasons for this are twofold: the IO performance is improved and the inputs must be packed anyway to perform the table lookup. During synthesis, the compiler ensures that no CLB's primary inputs are in more than *PI_BYTES* bytes. For the benchmark circuits, *PI_BYTES* had a value of 4. The following data structure contains information for using the input vectors:

```
struct pi_struct {
        char num_pi_bytes;
        unsigned short pi_mapping[PI_BYTES][256];
        off_t offset[PI_BYTES];
}
```

The field *pi_bytes* is the number of bytes containing primary inputs for the CLB and *offset* points to the file location where each block of vectors resides. The *pi_mapping* field is a lookup table for

re-ordering the byte. When OR'ing together the input vector bytes, some of the bits may overlap. Re-mapping ensures each bit has a unique location.

The code for reading the input vectors is as follows:

```
for(l=0; l<PI_BYTES; l++) {
        if (l < pi_info.num_pi_bytes) {
                pp_lseek(d, pi_info.offset[l], L_SET);
                pp_read(d, input_vecs[l], VECS);
        }
}
```

The functions *pp_lseek* and *pp_read* are parallel versions of the UNIX system calls *seek* and *read*. In the code above, each processor reads VECS number of bytes from the file given by descriptor *d* into the buffer *input_vecs[l]*. Each PE can read from different file positions, set by *pp_lseek*.

### 7.2.3 Main Simulation Loop

The input vector code reads VECS vectors at a time. The main simulation loop is invoked for each such block of vectors. For each iteration of the loop, the inputs to the CLB are OR'ed together using the variable *vector:*

```
vector = 0;
for (i=0; i<m_inputs; i++) {
        j = input_buffer[i][vec_count];
        if (j==empty)
                j = prev_inputs[i];
        else
                prev_inputs[i] = j;
        input_buffer[i][vec_count] = empty;
        vector = vector | j;
}
```

When a CLB's output is unchanged, it is not propagated. This leaves the corresponding entry in the input buffer of the succeeding CLB *empty*. The loop above checks for empty entries. When it finds one, the previous input is used. When the entry in the input buffer is not empty, *prev_inputs* is updated.

The following loop OR's together *vector* with any required primary inputs:

```
for(j=0; j<PI_BYTES; j++) {
        vector = vector | pi_info.pi_mapping[j][input_vecs[j][vectors]];
}
```

Once *vector* is calculated, it can be used to lookup the output values:

```
output = m_proc_info.table[0][vector];
```

If the output has changed and it is not a primary output, the value is propagated:

```
if (output != prev_output0) {
        pp_rsend(dest, &output, send_adr, sizeof(short));
        prev_output0 = output;
}
```

The function *pp_rsend* copies a block of memory from the local PE to the *dest* PE. In this case, it places the output value into the input buffer of the succeeding CLB. The code for handling outputs is duplicated for each fanout of the CLB. When an output is a primary output, the value is stored. Primary outputs are written out later as a block when the main simulation loop finishes.

Chapter 8

# Overview of the Connection Machine-5

With its limited memory and slow speed, the MasPar MP-1 is obsolete compared to newer machines. The Connection Machine-5 (CM-5), one of the latest to be commercially available, offers much greater computational power. This chapter summarizes the architectural features of the CM-5 and discusses ways of adapting the work done on the MP-1 to the CM-5.

## 8.1 Processing Elements on the CM-5

Like the MP-1, the CM-5 provides a collection of processors, each with their own local memory. The current CM-5 implementation uses the SPARC microprocessor as the processing element with 8-32 MBytes of local memory. This configuration is considerably more powerful than the 1.8 MIPS, 64 KByte processing element used by the MP-1.

The processing elements in the CM-5 may also be equipped with a vector co-processor. In this configuration, memory is organized into four 8 Mbyte banks. Each vector unit has a peak performance of 32 Mops on 64-bit integer operands. The vector units receive instructions from the SPARC, either individually or broadcasted to all four units. The SPARC can perform other tasks while the vector units execute instructions.

Each vector unit has 64 64-bit registers, which can also be used as 128 32-bit registers. A vector mask register allows certain vector elements to be "masked out" for conditional vector processing. Besides addition and multiplication operations, there is support for vectored bitwise logical shift, AND, NAND, OR, NOR, XOR, and NOT.

## 8.2 The CM-5 Networks

The CM-5 has three separate, scalable networks: the Control Network, the Data Network, and the Diagnostic Network. The diagnostic network, used for detecting faulty processors, is not visible to the programmer and will not be discussed here.

The Control Network provides broadcasting, synchronization, and reduction. Reduction combines values from multiple processors to form a single result. The possible reduction operations are: summation, finding the maximum or minimum value, logical OR, exclusive OR, and logical AND.

The Data Network provides simultaneous point-to-point transmission of messages. Unlike the MP-1, the CM-5 Data Network is hierarchical, emphasizing data locality. Processing elements are grouped into four's. Maximum bandwidth between elements within a group of four is 20 Mbytes/sec, 10 Mbytes/sec between groups of 16, and 5 Mbytes/sec for all other messages.

## 8.3 Programming the CM-5

The CM-5 supports both SIMD and MIMD computing. SIMD programming on the CM-5 is very similar to that on the MP-1. Loops and branching constructs restrict which processors are active, implicitly handling synchronization. When multiple execution paths are possible (such as with if-else statements), each block is executed serially, reducing parallelism. Fortunately, such conditions were largely avoided by the MP-1 parallel simulator.

MIMD computing is provided through the control network. The following "two-phase barrier" synchronization method is used: A processor notifies the Control Network that it is ready to enter a barrier. When all other processors have reached the barrier, the network notifies all the processors. While waiting for notification, processors may perform other tasks with their individual instruction streams.

The CM-5 has system calls for switching between SIMD and MIMD processing.

In SIMD mode, the Control Processor broadcasts blocks of instructions to the processing nodes. In MIMD mode, processors independently fetch instructions as needed and synchronization becomes the responsibility of the programmer.

SIMD programs run between two phases: local computation and global communication. By handling communication all at once, synchronization overhead is reduced. It is also easier for compilers to detect common communication patterns. For these reasons, the CM-5 Technical Summary [CM91] recommends using SIMD processing.

## 8.4 Parallel Simulation on the CM-5

The CM-5 has all the architectural features of the MP-1, so the simulation methods for the MP-1 are adaptable to the CM-5. However, it is unclear whether the two machines have comparable "grain" sizes. The synthesis algorithm was tuned for an optimal ratio of compuation and communication on the MP-1. Experimentation is needed to find optimal settings for the CM-5. This is especially true when the CM-5 is equipped with vector hardware. In this configuration, each processing element is a small supercomputer.

Parallel simulation can exploit a number of architectural features of the CM-5 not found on the MP-1 -- a larger address space, vector processing, MIMD processing, and a data network emphasizing data locality.

If the same table lookup methodology were used, the CM-5's address space would allow much larger CLB sizes. When equipped with 8 MBytes of memory, each CLB could have up to 22 inputs. However, experiments performed on the benchmark circuits found that increasing the CLB size beyond 13 did not produce a fewer number of CLB's.

The extra memory in the CM-5 would be useful for handling multiple CLB's at each processing element ("CLB-folding"). Given the cost of a CM-5 processing element, even a machine with several hundred processors is quite expensive. With fewer processors, each node must simulate multiple CLB's.

The processing elements in the CM-5 are fast enough that it might be advantageous to use CLB-folding to save on communication costs, especially on machines with vector units. The computation step of the MP-1 simulator performs two steps: 1. OR the inputs to the CLB together to form a table index and 2. Look up the output values from the tables. Step 1 (the OR loop) can be vectorized to handle multiple CLB's simultaneously. So, CLB-folding does not incur a large parallelism penalty on a CM-5 with vector units.

Unlike the MP-1, the CM-5 provides MIMD computing, allowing each processor to execute an independent instruction and data stream. The current version of the simulator would not benefit greatly from this capability. MIMD processing would be useful for mixed-mode simulation, which would allow simultaneous simulation at the transistor, RTL, or behavioral level.

The CM-5 data network, which is quite different than that in the MP-1, is more amenable to parallel simulation. The MP-1 placement algorithm would do poorly on the CM-5 because it assumes a flat network hierarchy. Instead, the following algorithm is proposed:

1. Assign primary output CLB's to processors such that they are evenly spaced apart.

2. Visit CLB's topologically from the primary outputs to the inputs. Try to place each CLB as close to its fanout CLB's as possible If CLB folding is used, try to place the CLB on the same processor as its fanout. If it does not fit on the same processor, put it in the fanout's group of 4. If it does not fit in the same group of 16, place the CLB in the cluster with the fewest occupied processors.

The major difficulty in step 2 occurs when a CLB belongs to the support of more than one primary output. In this case, it may be impossible to place the CLB near all its fanouts. It is likely that duplicating these CLB's would cause the circuit size to explode. A partial solution is to only duplicate pathologic cases, such as CLB's that cannot be placed in the same group of 16 as one of it's fanouts. This might also require combining the duplication and placement steps of the compiler.

# Chapter 9

# Conclusions and Future Work

A parallel functional simulator was developed for the MasPar MP-1, a massively parallel, SIMD computer. This simulator was used to test optimizations for improving parallel simulation performance.

The results demonstrate the importance of optimizing the circuit. Parallel simulation of the unmodified circuits was only slightly faster than a SparcStation-2 running Verilog-XL. Logic synthesis increased parallel simulation performance by more than an order of magnitude for large circuits.

The data show that the MP-1 is more scalable to larger circuits than a workstation. The circuit s38417, with over 100 times more gates than c432, simulated 47 times slower than c432 under Verilog, but only 2 times slower on the MP-1. Parallel computers have the extra benefit that they can be easily scaled to simulate larger circuits by adding more processors. It is more difficult to scale a workstation without using hardware accelerators. Even increasing the instruction and data cache sizes has limited effect because the simulation code is so large that cache misses are frequent.

It is significant that the MP-1 outperforms the SparcStation-2. The processing element used by the MP-1 is more than ten times slower than the SPARC, yet the MP-1 can outperform the SparcStation-2 by more than a factor of 10. The synthesis algorithm is clearly exploiting parallelism in the circuit. On the MP-1, roughly two-thirds of the simulation time was spent on compuation. On the CM-5 (which uses the SPARC as its processing element), this portion of the simulation would run 10 times faster. Thus, it is likely that the CM-5 could simulate two orders of magnitude faster than a uniprocessor.

# Bibliography


[A86] Jeffrey M. Arnold, "Parallel Simulation of Digital LSI Circuits," *Tech. Report 86-295*, MIT, January 1986

[Bry87] R. E. Bryant, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proc. of the 24th DAC*, 1987.

[BR+87] R. Brayton, R. Rudell *et al*, "MIS: A Multi-Level Logic Optimization System," in *IEEE Transactions on Computer-Aided Design*, Vol. 6, No. 5, November 1987.

[BS85] Mary Bailey and Lawrence Snyder, "An Empirical Study of On-Chip Parallelism," in *Proc. of the 25th DAC*, 1988, pp. 160-165.

[CM81] K. M. Chandry and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *CACM*, April 1981, Vol. 24, No. 11, pp. 198-206.

[CM91] *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corporation, 1991.

[Erd89] D. Erdman, "The Newton Waveform Relaxation Approach to the Solution of Differential Algebraic Systems for Circuit Simulation," Duke University, Durham NC, 1989.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979.

[Hil86] W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1986.

[JD85] D. R. Jefferson *et al*, "Implementation of Time Warp on the Caltech Hypercube," in *1985 Society for Computer Simulation Multiconference*, January 1985.

[KBR89] Saul Kravitz, Randal Bryant, and Rob Rutenbar, "Massively Parallel Switch-level Simulation: A Feasibility Study," in *Proc. of the 26th DAC*, 1989.

[Nag75] L. W. Nagle, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *Technical Report ERL-M520*, University of California, Berkeley, May 1975.

[SB88] Larry Soule' and Tom Blank, "Parallel Logic Simulation on General Purpose Machines," in *Proceedings of the 1988 DAC.*

[SG89] L. Soule' and A. Gupta, "Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation," in *Proc. of the 26th DAC*, 1989.

[PS88] D. L. Pitts and D. L. Smith, "Central Concurrency Control for Distributed Simulations," in *Proc. of the 1988 Summer Computer Simulation Conference*, 1988.

[Ter83] C. Terman, "RSIM -- A Logic-Level Timing Simulator," in *Proc. of the IEEE Conference on Computer Design*, 1983.

[TM91] Donald Thomas and Philip Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1991.

[W86] Andrew Wilson, "Parallelization of an Event Driven Simulator on the Encore Multimax," *Tech. Report ETR 86-005*, Encore Computer, 1986.

[WF87] Ken Wong and Mark Franklin, "Performance Analysis and Design of a Logic Simulation Machine," in *Proc. of the 14th Annual International Symposium on Computer Architecture*, 1987, pp. 49-55.

[Xil] *The Programmable Gate Array Data Book*, Xilinx Inc., 2069 Hamilton Ave. San Jose, CA 95125.

Appendix

# Simulation Code

This appendix gives the MPL simulation code. It is divided into three files: main.c, upload.c, and sim.c. The file upload.c contains code for reading the circuit file. After verifying it has no errors, it is uploaded to the DPU. The the actual simulation is handled in sim.c.

psim.h

```
#include <sys/file.h>

#define FANOUTS 4
#define TABLE_SIZE 4096
#define MAX_FANOUTS 32
#define VECS 2500
#define PI_BYTES 8

typedef struct send_struct {
        short destination;
        char trans;
        short offset;
        short num;
        short vector[32];
        short order[32];
} Send;

typedef struct node_struct {
        short num_vectors;
        short level;
        Send sender[MAX_FANOUTS];
        unsigned short table[FANOUTS][TABLE_SIZE];      /* look-up table */
        char inputs;                                    /* messages expected */
} Node;

typedef struct pi_struct {
        char num_pi_bytes;
        unsigned short pi_mapping[PI_BYTES][256];
        off_t offset[PI_BYTES];
} Pi;


short get_num_input_vectors(), read_primary_inputs(), atob();
short re_order();
```

```
#include <stdio.h>
#include <sys/time.h>
#include "psim.h"

Send *upload_tables();
extern void m_simulate(), m_init();

main(argc, argv)                                                    main
int argc;
char **argv;
{
short       pats, num_pi, num_po, num_clumps, pi_clumps;

            if (argc != 2) {
                        printf("Usage: %s network-file\n", argv[0]);
                        exit(1);
            }

            init();
            pats = VECS;

            /* load tables into PE and grab primary input map */
            upload_tables(argv[1],pats, &num_pi, &num_po, &num_clumps, &pi_clumps);

            callRequest(m_simulate, 2 * sizeof(short), pats, num_po);
}

short get_num_input_vectors(fname)
char *fname;
{
FILE        *fp;
short       num;

            if ( (fp = fopen(fname, "r")) == NULL) {
                        fprintf(stderr,"Error: Could not open %s for reading\n", fname);
                        exit(1);
            }
            fscanf(fp, "%d\n", &num);
            fclose(fp);
            return(num);
}




char *strsav(s)
char *s;
{
char *copy;

            copy = (char *) malloc(strlen(s)+1 * sizeof(char));
            if (!copy) {
                        perror("strsav");
                        exit(1);
            }
            strcpy(copy, s);
            return(copy);
}

enumerate(line, out, table)                                         enumerate
char *line;
short out;
```

*...enumerate*

```
unsigned short *table;
{
char        *copy1, *copy2;
short       value, i=0;
short       j;

            while(line[i]) {
                        if (line[i] == '-') {
                                    copy1 = strsav(line);
                                    copy1[i] = '0';
                                    enumerate(copy1, out, table);
                                    free(copy1);
                                    copy2 = strsav(line);
                                    copy2[i] = '1';
                                    enumerate(copy2, out, table);
                                    free(copy2);
                                    return;
                        }
                        i++;
            }
            value = atob(line);
            table[value] = table[value] | out;
}


short atob(s)
char *s;
{
short       i=0, order = strlen(s)-1, sum=0;

            while(s[i]) {
                        sum += (s[i] - '0') << order;
                        i++; order--;
            }
            return(sum);
}
```

*init*

```
init()
{
            callRequest(m_init, 0);
}
```

```
#include <stdio.h>
#include "psim.h"

extern void m_init_table(), m_init_messages(), m_init_sender();
extern void init_input_buffer(), m_read_primary_inputs(), m_init_num_vectors();
extern void m_set_clump_pindex(), m_set_input_sends(), m_set_proc_info();
extern void m_set_pi_src(), m_get_pi_info(), m_get_levels();

Send *upload_tables(fname, vector_count, num_pi, num_po, num_clumps, pi_clumps)
char *fname;
short vector_count;
short *num_po, *num_pi, *num_clumps;
short *pi_clumps;
{
Pi          *fe_pi_info;
char        m_byte[32], b, off;
FILE        *fp;
char        line[8192], pattern[32], s_value[32];
short       clump, count, order, messages, foo, bar;
short       i, j, inputs, outputs, sop, value, outs, sends, pi_indicies;
short       line_ct=0;
short       p, rec, clump_levels;
unsigned short          table[TABLE_SIZE];
char        primary_output_buffer=0;
Send        sender[MAX_FANOUTS], *primary_inputs;
Node        proc;
int         PI_index[8192];
int         real_inputs;
char        input_ordering[32], index[32];

            if ( (fp = fopen(fname, "r")) == NULL) {
                        fprintf(stderr, "Error: Could not open `%s' for reading.\n",
                                    fname);
                        exit(1);
            }

            fgets(line, 8192, fp);                          /* skip Global: */
            fgets(line, 8192, fp);                          /* skip .clumps: */
            fscanf(fp, ".clump_levels: %hd\n", &clump_levels);
            send_clump_levels(clump_levels);

            fscanf(fp, ".pi %hd\n", num_pi);
            fscanf(fp, ".po %hd\n", num_po);
            line_ct+=3;
            fgets(line, 8192, fp);                          /* skip pi: */
            fgets(line, 8192, fp);                          /* skip po: */

            for(i=0; i<8192; i++) PI_index[i] = -1;

            fscanf(fp, ".pi_index %hd:\n", &pi_indicies);
            line_ct+=3;
            for(i=0; i<pi_indicies; i++) {
                        fscanf(fp, "%hd: %hd\n", &p, &rec);
                        PI_index[p] = rec;
                        line_ct++;
            }

            fscanf(fp, ".records %hd\n", pi_clumps);
            line_ct++;

            primary_inputs = (Send *) calloc((*pi_clumps) + 1, sizeof(Send));
            if (!primary_inputs) Quit("upload_tables()");

            fe_pi_info = (Pi *) calloc((*pi_clumps) + 1, sizeof(Pi));
```

```
if (!fe_pi_info) Quit("upload_tables()");

for(i=0; i < *pi_clumps; i++) {
        fscanf(fp, "record %hd\n", &foo);
        fscanf(fp, ".send %hd\n", &sends);
        line_ct++;
        primary_inputs[i].num = sends;

        for(j=0; j<sends; j++) {
                fscanf(fp, "%hd", &foo);
                primary_inputs[i].vector[j] = foo;
        }
        fscanf(fp, "\n");
        line_ct++;
        for(j=0; j<sends; j++) {
                fscanf(fp, "%hd", &foo);
                primary_inputs[i].order[j] = foo;
        }
        fscanf(fp, "\n");
        line_ct++;
        build_pi_info(&(primary_inputs[i]), &(fe_pi_info[i]), i);
}
fgets(line, 8192, fp);
line_ct++;
if (strcmp(line, ".end global\n"))
        quit(line_ct, "Error in map file — no '.end global'");

send_pi_info(PI_index, fe_pi_info);

/* read in local processor information */
*num_clumps = 0;
while(1) {
        fscanf(fp, "Clump: %hd(%hd)(%hd)\n", &clump, &foo, &bar);
        if (feof(fp)) break;
        proc.num_vectors = vector_count;
        fscanf(fp, "%hd, %hd\n", &foo, &bar);
        fscanf(fp, ".level %hd\n", &(proc.level));
        (*num_clumps)++;
        fscanf(fp, ".outs %hd\n", &outs);
        line_ct+=3;
        if (outs > MAX_FANOUTS) quit(line_ct, "Too many .outs");
        for(i=0; i<outs; i++) {
                fscanf(fp, ".send %hd\n", &sends);
                line_ct++;
                proc.sender[i].num = sends;
                fscanf(fp, "%hd(%hd), %hd: ",
                        &(proc.sender[i].destination),
                        &(proc.sender[i].trans),
                        &(proc.sender[i].offset));
                for(j=0; j<sends; j++) {
                        fscanf(fp, "%hd", &foo);
                        proc.sender[i].vector[j] = foo;
                }
                fscanf(fp, "\n");
                line_ct++;
                for(j=0; j<sends; j++) {
                        fscanf(fp, "%hd", &foo);
                        proc.sender[i].order[j] = foo;
                }
                fscanf(fp, "\n");
                line_ct++;
        }
```

```
                proc.sender[i].destination = -1;                    /* -1 designates PO */
                fscanf(fp, ".PO %hd\n", &sends);
                if (sends && outs)
                        quit(line_ct, "Error: Clump has both po and clump out");

                if ((sends + outs) > MAX_FANOUTS)
                        quit(line_ct, "Too many .outs");
                line_ct++;
                proc.sender[i].num = sends;
                for(j=0; j<sends; j++) {
                        fscanf(fp, "%hd", &foo);
                        proc.sender[i].vector[j] = foo;
                }
                fscanf(fp, "\n");
                line_ct++;
                for(j=0; j<sends; j++) {
                        fscanf(fp, "%hd", &foo);
                        proc.sender[i].order[j] = foo;
                }
                if (sends) proc.sender[i].offset=primary_output_buffer++;
                proc.sender[++i].num = 0;
                fscanf(fp, "\n");
                line_ct++;

                fscanf(fp, ".ci %hd\n", &(proc.inputs));
                fscanf(fp, ".i %hd\n", &inputs);
                fscanf(fp, ".o %hd\n", &outputs);
                if (outputs > 8 * sizeof(unsigned short))
                        quit(line_ct, "Table output size too large");
                fgets(line, 8192, fp);                          /* skip input labels */
                fgets(line, 8192, fp);                          /* skip output labels */
                fscanf(fp, ".p %hd\n", &sop);
                line_ct+=6;

                for(i=0; i<FANOUTS; i++) {
                        for(j=0; j<TABLE_SIZE; j++) proc.table[i][j] = 0;
                }

                for(i=0; i<sop; i++) {
                        fscanf(fp, "%s %s\n", pattern, s_value);
                        line_ct++;
                        value = atob(s_value);
                        enumerate(pattern, value, proc.table[0]);
                        enumerate(pattern, value, proc.table[1]);
                }
                if (outs) Order(&proc);
                fgets(line, 8192, fp);
                line_ct++;
                if (strcmp(line, ".e\n"))
                        quit(line_ct, "Error: No .e found in table");

                send_proc_info(clump, proc);
        }
        fclose(fp);
        return(primary_inputs);
}

build_pi_info(primary_inputs, fe_pi_info, rec)                      build_pi_info
Send *primary_inputs;
Pi *fe_pi_info;
short rec;
{
int         b, i;
```

upload.c

*...build_pi_info*

```
int         vec, order;
int         off, byte_offset;
char        m_byte[256];

            for(b=0; b<32; b++) m_byte[b] = 0;
            for(b=0; b<primary_inputs->num; b++) {
                        vec = primary_inputs->vector[b];
                        m_byte[vec/8] = 1;
            }
            off = 0;
            for(b=0; b<32; b++) {
                        if (m_byte[b]) {
                                    fe_pi_info->offset[off] = (off_t) b * VECS;
                                    m_byte[b] = off++;
                                    if (off >= PI_BYTES) {
                                                fprintf(stderr, "Too many PI in rec %d\n", rec);
                                                return;
                                    }
                        }
            }
            fe_pi_info->num_pi_bytes = off;

            for(i=0; i<PI_BYTES; i++) init_pi_table(fe_pi_info->pi_mapping[i]);

            for(b=0; b<primary_inputs->num; b++) {
                        vec = primary_inputs->vector[b];
                        order = primary_inputs->order[b];
                        byte_offset = m_byte[vec/8];
                        mod_table(fe_pi_info->pi_mapping[byte_offset], vec, order);
            }
}
```

*mod_table*

```
mod_table(table, pos, order)
unsigned short *table;
int pos, order;
{
int         vector;
unsigned short val, mask;

            pos = pos % 8;
            for(vector = 0; vector < 256; vector++) {
                        mask = 1 << pos;
                        table[vector] = table[vector] |
                                    (((vector & mask) != 0) << order);
            }
}
```

*send_clump_levels*

```
send_clump_levels(clump_levels)
short clump_levels;
{
            callRequest(m_get_levels, sizeof(short *), &clump_levels);
}
```

*init_pi_table*

```
init_pi_table(table)
unsigned short *table;
{
int         i;

            for(i=0; i<256; i++) table[i] = 0;
}
```

*send_pi_info*

```
send_pi_info(PI_index, fe_pi_info)
int *PI_index;
Pi *fe_pi_info;
{
short       i, p;

            for(i=0; i<8192; i++) {
                        if (PI_index[i] != -1) {
                                    p = PI_index[i];
                                    callRequest(m_get_pi_info, 2*sizeof(short)+sizeof(Pi *),
                                                &(fe_pi_info[p]), i, p);
                        }
            }
}
```

*Order*

```
Order(proc)
Node *proc;
{
short       i, f;
short       j, output;
unsigned short vector;
Send        *send;

    for(f = 0; f < FANOUTS; f++) {
            for(i=0; i < TABLE_SIZE; i++) {
                        if (proc->table[f][i]) {
                                    send = &(proc->sender[f]);
                                    output = proc->table[f][i];
                                    vector = 0;
                                    for(j=0; j<send->num; j++) {
                                                vector = vector | (((output & (1 <<
                                                            send->vector[j])) != 0) <<
                                                            send->order[j]);
                                    }
                                    proc->table[f][i] = vector;
                        }
            }
    }
}
```

*send_proc_info*

```
send_proc_info(clump, proc)
short clump;
Node proc;
{
            callRequest(m_set_proc_info,
                        sizeof(short) + sizeof(Node *), clump, &proc);
}
```

*quit*

```
quit(line, s)
short line;
```

upload.c

*...quit*

```
char *s;
{
        fprintf(stderr, "Line %d: %s\n", line, s);
        exit(1);
}
```

*Quit*

```
Quit(s)
char *s;
{
        perror(s);
        exit(1);
}
```

```
short re_order(buffer, p)
char *buffer;
Send *p;
{
short   i, r=0;

        for(i=0; i<p->num; i++)
                r = r | (buffer[p->vector[i]] << p->order[i]);
        return(r);
}
```

```
#include <stdio.h>
#include <sys/time.h>
#include <mpl.h>
#include "psim.h"

#define empty 10000
#define send_adr &(input_buffer[m_sender->offset][vec_count])
#define LEVELS 32


plural Pi pi_info;
plural short global_m_num_vectors;
plural unsigned short ** plural m_table;
plural char global_m_inputs;
plural Send * plural m_sender;

plural Node m_proc_info;
plural unsigned short input_buffer[16][LEVELS];
plural unsigned char input_vecs[PI_BYTES][VECS];
short network_levels;

void perform_simulation(), m_n.ap();


visible void m_set_proc_info(clump, info)
short clump;
Node *info;
{
short       nx, ny;

            m_map(clump, &nx, &ny);
            blockIn(info, &m_proc_info, nx, ny, 1, 1, sizeof(Node));
}



visible void m_get_pi_info(fe_pi_info, clump, record)
Pi *fe_pi_info;
short clump, record;
{
short       nx, ny;

            m_map(clump, &nx, &ny);
            blockIn(fe_pi_info, &pi_info, nx, ny, 1, 1, sizeof(pi_info));
}


visible void m_get_levels(fe_levels)
short *fe_levels;
{
            copyIn(fe_levels, &network_levels, sizeof(short));
}



void m_map(clump, x, y)
short clump, *x, *y;
{
            *y = clump / nxproc;
            *x = clump % nxproc;
}
```

```
visible void m_init()
{
short       i, j;
plural      short x, y;

            global_m_inputs = -1;
            global_m_inputs = 0;
            global_m_num_vectors = 0;

            for(x = 0; x<PI_BYTES; x++) {
                        for(y=0; y<256; y++) pi_info.pi_mapping[x][y] = 0;
            }

/*****
            for(y=0; y<16; y++) {
                        for(x=0; x<LEVELS; x++) input_buffer[y][x] = 0;
            }
*****/

            pi_info.num_pi_bytes = 0;
}


quit(s)                                                                    quit
char *s;
{
            perror(s);
            exit(1);
}


void kludge()
{

            global_m_num_vectors = m_proc_info.num_vectors;
            m_table = m_proc_info.table;
            global_m_inputs = m_proc_info.inputs;
            m_sender = &(m_proc_info.sender[0]);
}

visible void m_simulate(pats, num_po)
short pats, num_po;
{
short       i;
int         d;
double      itime;
struct      timeval btime, etime;

            /* __routerCount = 0; */
            gettimeofday(&btime, 0);
            kludge();

            d = open("/tmp/inputs", O_RDONLY);
            if (d < 0) quit("open");
            perform_simulation(num_po, d);

            gettimeofday(&etime, 0);
            itime = (etime.tv_sec + 1.0e-6*etime.tv_usec) -
                  (btime.tv_sec + 1.0e-6*btime.tv_usec);
        fprintf(stderr, "Time: %e\n", itime);
            printf("routerCount = %d\n", __routerCount);
}
```

```
void perform_simulation(num_po, d)
short num_po, d;
{
plural register unsigned short i, j;
plural register unsigned short prev_output0, prev_output1;
plural register short m_num_vectors = global_m_num_vectors;
plural register char m_inputs = global_m_inputs;
plural register unsigned short vector;
plural register short vectors=0;
plural register short vec_count=0;
plural register short output;
plural register short sender_num = m_sender->num;
plural register char level;
plural short dest = m_sender->destination;
plural unsigned short outbuf;
plural short po_index = m_sender->order[0];
plural unsigned short prev_inputs[16];
char final_out[LEVELS][256];
char l;
int k, out_count = 0;

        for(i=0; i<16; i++) {
                for(j=0; j<LEVELS; j++)
                        input_buffer[i][j] = empty;
        }
        prev_output0 = prev_output1 = empty;

        for(l=0; l<PI_BYTES; l++) {
                if (l < pi_info.num_pi_bytes) {
                        pp_lseek(d, pi_info.offset[l], L_SET);
                        pp_read(d, input_vecs[l], VECS);
                }
        }

        level = m_proc_info.level;

        while(vectors < m_num_vectors) {
                if (level) {
                        level--;
                        continue;
                }
                vector = 0;
                for(i=0; i<m_inputs; i++) {
                        j = input_buffer[i][vec_count];
                        if (j==empty) {
                                j = prev_inputs[i];
                        }
                        else
                                prev_inputs[i] = j;
                        input_buffer[i][vec_count] = empty;
                        vector = vector | j;
                }

                /* PI evaluation */
                for(j=0; j<PI_BYTES; j++) {
                        vector = vector |
                            pi_info.pi_mapping[j][input_vecs[j][vectors]];
                }

                output = m_proc_info.table[0][vector];
                if (dest == -1) {
                    m_proc_info.table[2][vectors] = m_proc_info.table[0][vector];
              m_proc_info.table[3][vectors] = m_proc_info.table[1][vector];
```

```
                }

                else {
                        outbuf = output;
                        if (outbuf != prev_output0) {
                                pp_rsend(dest,&outbuf, send_adr, sizeof(short));
                                prev_output0 = outbuf;
                        }

                        m_sender = &(m_proc_info.sender[1]);
                        if (m_sender->num) {
                                dest = m_sender->destination;
                                outbuf = m_proc_info.table[1][vector];
                                if (outbuf != prev_output1) {
                                        pp_rsend(dest, &outbuf,
                                                send_adr,sizeof(short));
                                        prev_output1 = outbuf;
                                }

                                m_sender = &(m_proc_info.sender[2]);
                                if (m_sender->num) {
                                        dest = m_sender->dest ination;
                                        outbuf = m_proc_info.table[2][vector];
                                        if (outbuf != prev_output2) {
                                            pp_rsend(dest, &outbuf,
                                                    send_adr, sizeof(short));
                                            prev_output2 = outbuf;
                                        }

                                m_sender = &(m_proc_info.sender[3]);
                                if (m_sender->num) {
                                        dest = m_sender->destination;
                                        outbuf = m_proc_info.table[3][vector];
                                        if (outbuf != prev_output3) {
                                            pp_rsend(dest, &outbuf,
                                                    send_adr, sizeof(short));
                                            prev_output3 = outbuf;
                                        }
                                }
                                }
                        }
                        m_sender = &(m_proc_info.sender[0]);
                        dest = m_sender->destination;
                }

                ++vectors;
                vec_count = ++vec_count % LEVELS;

                if (network_levels == 0) {
                        out_count = ++out_count % LEVELS;
                }
                else network_levels--;
        }
}
```