

# A Virtual Workbench for Electric Lego Labkits

by

Jay M. Grabeklis

Submitted to the Department of Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

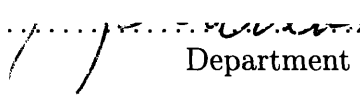
at the

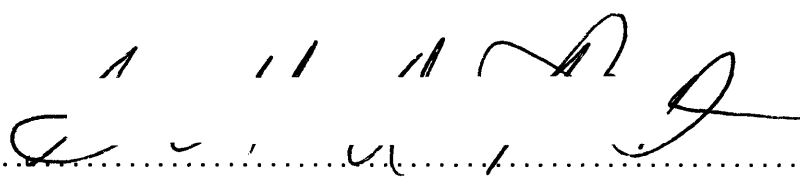
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

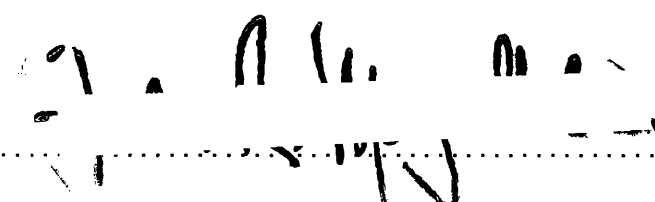
September 1996

© Jay M. Grabeklis, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly  
paper and electronic copies of this thesis document in whole or in part, and to grant  
others the right to do so.

Author  .....  
Department of Electrical Engineering and Computer Science  
August 26, 1996

Certified by  .....  
Gill A. Pratt  
Associate Professor  
Thesis Supervisor

Accepted by  .....  
F. R. Morgenthaler  
Chairman, Department Committee on Graduate Theses  
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Eng.

MAR 21 1997

LIBRARIES

# **A Virtual Workbench for Electric Lego Labkits**

by

Jay M. Grabeklis

Submitted to the Department of Electrical Engineering and Computer Science  
on August 26, 1996, in partial fulfillment of the  
requirements for the degree of  
Master of Engineering in Computer Science and Engineering

## **Abstract**

Current digital prototyping technology has been far out-paced by recent advances in micro-processor technology. In order to build a modern RISC processor a new technology is required. The Electric Lego uses Field Programmable Gate Arrays along with a novel clock distribution scheme to provide a fast, scalable alternative to protoboard and wire circuits. In order to effectively utilize this advance, however, a programming environment must be designed and built which closely simulates the experience of physically building a circuit. At present, commercially available software and FPGA programming methods are much too slow to deliver this kind of performance. In this paper I discuss the design and implementation of an FPGA programming environment which will closely simulate the physical construction of a circuit. Included in the environment are schematic entry and logic placement tools which interact with a fast router as well as a parallel port interface which far exceeds the download performance of traditional serial port interfaces.

Thesis Supervisor: Gill A. Pratt

Title: Associate Professor

# Contents

<b>1</b>	<b>Current Digital Prototyping Technology</b>	<b>5</b>
<b>2</b>	<b>A New Approach to Prototyping</b>	<b>8</b>
2.1	The FPGA . . . . .	9
2.2	Electric Lego Supporting Circuitry . . . . .	9
2.3	The Lab Kit . . . . .	10
2.4	Advantages of Electric Legos . . . . .	11
<b>3</b>	<b>Programming FPGAs</b>	<b>13</b>
3.1	Design Entry . . . . .	13
3.2	Down Loading Designs . . . . .	14
3.2.1	Serial FPGA Programming . . . . .	15
3.2.2	Byte FPGA Programming . . . . .	15
<b>4</b>	<b>The Virtual Workbench</b>	<b>17</b>
4.1	Primitive Part Table . . . . .	18
4.2	Input File Formats . . . . .	20
4.2.1	The Xilinx Netlist File Parser . . . . .	21
4.3	The Main Data Structure . . . . .	23
4.3.1	Part Types and Instances . . . . .	23
4.3.2	Adding Parts to the Structure . . . . .	24
4.3.3	Adding Connections to the Structure . . . . .	25

4.4	The Schematic Capture Program . . . . .	26
4.4.1	The Schematic Program's View . . . . .	26
4.4.2	Select Mode . . . . .	27
4.4.3	Part Mode . . . . .	27
4.4.4	Wire and Bus Modes . . . . .	28
4.5	The Logic Placer . . . . .	28
4.6	The Router . . . . .	29
<b>5</b>	<b>The Host/Lab Kit Interface</b>	<b>32</b>
5.1	The Parallel Port Interface . . . . .	32
5.1.1	The Enhanced Parallel Port Protocol . . . . .	33
5.1.2	The Labkit Parallel Port Hardware Interface . . . . .	35
5.1.3	The Gateway Chip . . . . .	37
5.2	The Parallel Port Software Interface . . . . .	41
<b>6</b>	<b>Conclusion and Possible Extensions</b>	<b>45</b>
<b>A</b>	<b>Central Data Structure Classes</b>	<b>47</b>
<b>B</b>	<b>Routines for Building the Central Data Structure</b>	<b>49</b>
<b>C</b>	<b>Parallel Port Programmable Array Logic Specification</b>	<b>51</b>

## Chapter 1

# Current Digital Prototyping Technology

In many current EECS lab classes the base technology for constructing any kind of circuit is the “proto-board”. A proto-board is a plastic plate with a grid of holes overlying connectors that allows several wires to be connected at each node. A digital circuit can be created by plugging MSI chips into the board and connecting their inputs and outputs appropriately with wires. Proto-boards are inexpensive and they provide a student with a physical representation of the circuit that is easy to wire and debug. Unfortunately, there are several inherent drawbacks in using proto-boards for circuit construction.

The first problem with wire proto-board-and-wire circuits is their electrical characteristic. Proto-board sockets have relatively high parasitic inductances and capacitances and the long wires between nodes have large, uncontrolled impedances. Either of these problems alone would impose severe speed limitations on a digital circuit, but together they force a design to be clocked at a snail’s pace. In 6.004, for example, the lab series includes the implementation of a simple architecture called the “Maybe”. The limitations of the proto-boards limit the Maybe’s clock frequency to approximately 1 MHz.

A second limitation is the physical bulk of the wires. In the past, many real-world architectures had 8- or 16-bit wide data paths. It was not unreasonable to expect a stu-

dent to wire an 8-bit bus by hand and such an implementation would be an acceptable approximation of a real microprocessor. Over the past few years, however, the standard microprocessor has adopted 32-bit wide data paths. It is difficult to reliably build such a data path by hand with wires on a proto-board substrate. First, there is the physical limitation of the number of sockets on a proto-board. A few 32-bit paths would likely take up a majority of the available sockets on a board. Unfortunately, not all of the connections would be reliable. When you take the students' sanity into account the thought of implementing a computer this way seems out of the question. Debugging and maintaining a data path like the one described above would be a full-time job.

Another issue related to the problem of wider data paths is one of finding components for students to use as the building blocks of their computer. For instance, students of course do not build an ALU from scratch. For an 8-bit implementation it is possible to get an 8-bit ALU off-the-shelf that will plug straight into the proto-board. Finding such a part in a 32-bit version is more difficult.

When the current proto-board technology was introduced over a decade ago, its capabilities were adequate. The Maybe machine mentioned above, for example, is an 8-bit bus-based architecture and can be directly implemented on a proto-board. This was a decent approximation of the average commercially available microprocessor in the early-to-mid 1980's. The Maybe ran much more slowly than a real processor, of course, but students were directly implementing the architecture they were trying to learn about. Since then, however, advances in microprocessor technology have greatly out-paced the capabilities of proto-board and wire implementations. Today's state-of-the-art microprocessor is a 32 or even 64-bit pipelined design that directly implements a RISC-like instruction set. 6.004 in its current form attempts to expose students to this class of processor by running a virtual 32-bit RISC machine on top of the 8-bit Maybe. The problem with this is that in order to simulate a 32-bit architecture on an 8-bit machine there must be interpretive layers between the virtual and physical machines. In the case of the Maybe there are two such layers. In the first layer, the RISC instructions are interpreted via a microcode ROM into Maybe "nanocode". This nanocode is then interpreted into Maybe machine instructions which,

because of the electrical problems mentioned above, can be executed at a rate of only 1000 per second. The effect of this can be a loss of focus from the important topics of the course. The idea behind building the RISC machine is to teach students the fundamentals of modern microprocessors. Students tend to concentrate on trying to understand the layers of interpretation rather than on internalizing the basic concepts of the modern microprocessor. Even though understanding the use of microcode is part of understanding how a processor works, given the movement of modern architecture towards the direct hardware implementation of instruction sets more emphasis should be placed on those concepts. The 6.004 lab kit, in its current form, cannot meet this goal. The lab kits dictate that the physical machine can be no better than an 8-bit architecture. The requirement of teaching the basics of a modern microprocessor means that students will inevitably get lost in the layers of interpretation. With the likelihood of needing to teach the basics of parallel machines in the not-too-distant future, it is clear that a new technology is needed.

## Chapter 2

# A New Approach to Prototyping

The limitations of wire-and-protoboard circuits suggest the need for a new technology for constructing digital circuits. The complexity of the modern microprocessor makes any kind of physical wiring an unlikely candidate. To directly implement a RISC machine, wide data paths are a must and it is impractical for a student to wire a 32-bit data path by hand. A logical alternative is some kind virtual workspace where components can be arranged into a circuit and down loaded onto programmable logic arrays. Programmable logic arrays provide enough resources to implement a simple RISC machine and would solve all of the physical problems associated with wire and protoboard circuits. The workspace for programming the arrays would include schematic entry, logic placement, and routing tools giving the look and feel of working with a physical circuit.

The new prototyping technology to be used in 6.004 has been dubbed the “Electric Lego”. An electric Lego module consists of a field programmable gate array (FPGA), supporting circuitry, and connectors on a 3-by-3 inch printed circuit card designed to be part of a three dimensional diamond lattice. The Electric Lego design is based on the NuMesh architecture and provides a very scalable framework for “plugging together” the FPGAs[3].



## 2.1 The FPGA

The FPGA is meant to provide the power and resources to directly implement today's state-of-the-art microprocessors as well as the flexibility to adapt to whatever innovations future computer science students might need to study. The FPGA used in the 6.004 Electric Lego design is the Xilinx 4013 chip. This chip consists of a 24x24 array of Configurable Logic Blocks (CLBs) and a routing network for communication between them. Each CLB consists of two four-to-one function generators on the input side, one function generator to combine the outputs of the others, two output flip-flops, and a network of multiplexers to generate appropriate control signals for the function generators and flip-flops. CLBs are capable of implementing any logic function of up to 9 inputs to 2 outputs with a propagation delay of only 7 nanoseconds[4].

Connecting this array of CLBs is an interconnect network consisting of three kinds of wires connected by switch matrices which consist of programmable n-channel pass transistors to make connections between wires. The first type of wire is called a "single-length" wire. These lines run horizontally and vertically and enter a switch matrix between each CLB. Next are the "double-length" wires which run parallel to the single-length wires, but only enter a switch matrix between every other block. Finally, there are "long-lines" which run the entire length of the chip. These are intended for timing-critical signals which might be skewed or unnecessarily delayed by running through the switch matrices. The three different types of wires are designed to minimize the parasitic inductances and capacitances for the average signal path while providing rich routing resources. With the ability of each of the CLBs to implement a 9-input function and the powerful routing between the blocks each Xilinx 4013 is capable of implementing over 13,000 logic gates.

## 2.2 Electric Lego Supporting Circuitry

The key to the Electric Legos' scalability is the supporting clock circuitry on each board. This circuitry allows an arbitrary number of Legos to have synchronized clock signals. The

clock distribution scheme used by the Legos is based on an idea proposed by Gill Pratt and John Nguyen[2]. In traditional clock distribution networks the signal is distributed by a tree-like structure which must be designed for a particular system configuration. These types of trees are not scalable. This is unsuitable for the Electric Legos because part of their usefulness stems from the ability to assemble them into any desired lattice.

Pratt and Nguyen's idea centers around generating a local clock at each node which is synchronized to its neighbors' clock signals. If all nodes in a network behave this way, the phase of all clocks in a network will converge to a global average. This simple averaging scheme, however, can result in stable, or mode-locked, configurations where the separate clocks in a network do not have equal phase. Pratt and Nguyen solve this problem by using the output of a triangle-wave phase detector to control the output of a voltage-controlled crystal oscillator. This phase detector ensures that the only stable state is when the phases of all clocks are equal.

The clock synchronizing system used for the Lego lattice is slightly different than the one mentioned above. Because the Legos form a diamond lattice instead of a two or three-dimensional cartesian mesh, the triangle-wave phase detector cannot eliminate all mode-locks. The Legos eliminate mode-lock by synchronizing incrementally. Instead of all of the nodes trying to synchronize at once, one new node at a time is added to the set of synchronized nodes. Because mode-lock is impossible both in a two-node network and when adding a single node to an already synchronized set, this incremental approach guarantees that the lattice will be synchronized.

## **2.3 The Lab Kit**

The Electric Legos would not be very useful without a system for programming them and communicating with them once they have been programmed. The system that will serve this purpose is a lab kit that will provide physical support and communication circuitry for the Legos and a host machine which will provide a software environment for designing and compiling circuits which can be down loaded onto the Legos.

The lab kit itself will be similar to the old 6.004 lab kits with a few modifications. The main section on the new kits will be an area for plugging in the Legos. The first Lego will straddle a “gateway” chip which facilitate communication between the lab kit and the host. The gateway will be described in detail in a later section. The rest of the Legos will build out from the first. Adjacent to the Lego area will be a small breadboard. This board will be used for simple labs early in the course as well as for monitoring signals in the lattice.

The host machine will run a software environment for creating circuits. The environment will provide students with the ability to graphically design and modify circuits, place the logic in the desired FPGA by hand, and automatically route signals between logic elements. In addition, the environment will manage the interface between the host and the lab kit and down load compiled designs to the Lego Lattice.

## **2.4 Advantages of Electric Legos**

The Electric Lego approach has many advantages over proto-board and wire circuits and the software environment, if carefully designed, will provide the look and feel of physically wiring a circuit without the cumbersome chore of plugging hundreds of wires into a proto-board. It might be argued that using a virtual workbench as opposed to chips and wires might take away from the lab experience. The point of the lab, however, is to visualize a circuit and build it from its component parts. What a student sees and connects will be precisely controlled in the virtual environment. Proto-board lab kits invariably turn into jumbled masses of wires which students pray will not completely come apart before the term is over.

The electric Lego approach will also solve the performance problems associated with proto-boards. Without the parasitics of the protoboards and the inductances of foot-long wires, designs can be clocked at much more realistic speeds. Students will also be able to directly implement the 32-bit wide data paths of today’s common RISC machines. Currently students are distracted by the interactions between two layers of microcode and extremely crude hardware that mysteriously result in the execution of a load-store instruction set.

With electric Legos, students will string 32-bit data paths between the components of a pipelined RISC machine and watch instructions being directly executed by the virtual hardware they have created. Additionally, without being forced to spend a considerable amount of time on seldom-used microcoding concepts, perhaps students can spend time on more interesting aspects of architecture. These might include the implementation of performance enhancements like pipeline bypassing, multi-level caches, and branch prediction.

## Chapter 3

# Programming FPGAs

One of the main challenges in building the new 6.004 lab kits is to provide students with an environment where all of the “hands-on” feel of the old protoboard technology is not lost. There is something about building a system yourself, as opposed to reading about it or simulating it in software, that makes for a deeper understanding. The main goal of the software environment is to convert a design from a concept in a student’s head to an implementation on a Lego FPGA seamlessly and with the illusion that all that is happening is that wires are being moved around from module to module. There are many excellent software packages around today that go a long way toward accomplishing this goal. These packages contain separate programs for schematic entry, placement of logic into CLBs, and routing of signals between CLBs. Unfortunately, they also have drawbacks that make them less than ideal.

### 3.1 Design Entry

As mentioned above, the illusion of physically wiring a circuit is critical to the 6.004 lab experience. When a wire is inserted or moved in a physical circuit, it can be immediately tested and debugged. This is not the case with commercially available software environments for programming FPGAs. When using Aldec’s Foundation software, for example, the process is as follows. First, a design is entered into a schematic capture program. This

program provides graphical representations for parts and the wires between them. When the user feels that a design is correct, the netlist (a textual description of the circuit) must be exported to a file. This process takes 15 to 20 seconds for the simplest designs containing on the order of 10 logic gates. For complex designs like the ones students will manipulate in 6.004 this process will take as much as 5 minutes or more. Next the netlist must be translated into a program that does the placement and chip-level routing for the design. This is a faster process than exporting the netlist from the schematic capture program, but still takes a significant amount of time. Once the netlist has been translated into this program the real problems begin.

The design passes through three stages: optimization, placement and routing, and bit stream generation. The bit stream is the binary file that is sent to the FPGA telling it how to program itself. Since the optimization step can usually be disabled we will only worry about the placement, routing, and bit stream generation. These processes are extremely time consuming even for the simplest designs. One design, an Enhanced Parallel Port interface which will be discussed in detail later, has 15 flip-flops, a 3-bit address decoder, and 6 3-input AND gates. It takes fifteen minutes to complete the placement, routing, and bit stream generation process. For the kinds of designs that 6.004 students will be implementing, this delay will be several hours. This is clearly unacceptable. Compounding the problem is the fact that every time the slightest change is made to a design the entire process from netlist exporting to bit stream generation must be completed again. This means that in the most involved labs every time the slightest change is made to a design, a student will have to wait several hours before the change can be downloaded to the FPGA for testing.

## **3.2 Down Loading Designs**

In addition to the circuit compilation issues presented above, there is the problem of getting the bit file onto an FPGA. The standard commercially available interface is called the “XChecker” cable and connects to the serial port of a host machine. The host programs

the FPGA by sending the bit file one bit at a time through the XChecker cable. Using this standard interface would add to the design loop delays and further impair the ability to provide students with the illusion of manipulating a physical circuit.

### **3.2.1 Serial FPGA Programming**

The details for the rest of this section refer to the programming methods for Xilinx FPGAs. These chips, specifically the Xilinx 4013, are the main feature of each Lego module. Programming one of these parts serially involves the manipulation of three signals: PROG, DIN, and CCLK. In order to prepare the chip for programming the PROG line must be held low for at least 10 milliseconds and then raised and held high for the rest of the time the chip is in use. Once the part has been initialized the configuration bits can be sent. These bits are sent by setting DIN to the appropriate value and strobing CCLK. In order to program an FPGA serially, the host has to shift bits out to DIN one at a time while generating an appropriate pulse on CCLK for each bit. This involves at least 3 clock cycles per bit. One cycle is needed to set DIN. A second cycle raises CCLK. The third cycle clears CCLK completing the transmission of one bit. This method is obviously unsuited for the purposes of the Lego lab kits. Luckily, Xilinx offers another way to program its chips.

### **3.2.2 Byte FPGA Programming**

Xilinx's Asynchronous Peripheral Mode (APM) is perfectly suited for the Lego kits. What is required by the Electric Legos kit is a fast, simple, bi-directional interface to get bit files from the host machine to the lattice. APM enables the design of an interface that fits these criteria.

In APM, one byte can be presented to the FPGA at a time. Upon receiving a pulse on the -WS pin the chip latches the byte into an internal shift register. Then the bits are shifted out on the rising edges of an internal oscillator signal instead of relying on an external source to generate CCLK. While the chip is busy shifting bits it holds the RDY/-BUSY signal low to prevent the host from sending another byte. As soon as the FPGA is

ready for the next data byte it raises RDY/-BUSY. During download, the Xilinx -INIT pin is held high. If an error occurs at any time during download, the -INIT signal is pulled low and no more configuration bits are shifted into the chip. Another signal, DONE, is held low during programming. After a successful download this signal is released. If the signal is pulled high through a resistor externally, a high on the DONE line signals a successful programming of the FPGA.

AMP makes the download bit rate higher by allowing almost one bit per internal clock cycle to be shifted into the chip. The rate is not exactly one because some cycles are missed if the shift register is waiting for the next byte. In addition, the number of strobes generated by the host is reduced by a factor of eight, further lowering the per-bit programming overhead. The interface for the Lego kits is designed to take advantage of this programming method.

The facts presented above suggest that the use of FPGA technology in a setting such as 6.004 necessitates a better development environment capable of more closely approximating the experience of physically building a circuit. Also helpful would be a host-to-kit interface that would take advantage of the faster FPGA programming modes and allow for bi-directional communication between the design environment and the Lego lattice. The design and construction of such a system will be described in the following sections.



## Chapter 4

# The Virtual Workbench

The key to improving upon the already existing software environments is to create an environment where the component parts are more tightly coupled. Included in this environment will be a router designed at MIT by Randy Sargent and Carl Witty for the 6.004 lab kits as well as schematic capture and logic placement programs designed as part of this thesis. The router has the ability to make incremental changes to a design. For example, if a student needs to replace a regular flip-flop with one with an output enable input the router can un-instantiate the old flip-flop and instantiate the new one without having to compile the rest of the circuit. The 6.004 development environment is designed to capitalize on this capability.

The flaw in the commercially available packages is that the schematic entry, logic placement, and signal routing facilities are similar to stages of an assembly line. The output of one stage is the input of the next. A better model would be one central representation that can be acted upon simultaneously by any of the components. Each of the components (schematic capture, logic placement, signal routing) could be thought of as a filter which shows a different view of the circuit. The schematic capture shows the schematic shapes of the parts and the wires between them and the logic placer shows the shapes of the parts in terms of CLBs and where they are located on the FPGA. Using this approach, the user is given the illusion that they are wiring a virtual circuit in the same manner that a physical

one would be wired. That is, it appears that a wire is being plugged in and the circuit is ready for testing immediately (or at least shortly) thereafter. This is made possible by eliminating the cumbersome interfaces between the component programs. By using the central data structure, changes made by the schematic capture or logic placement program can immediately be seen by the router. Upon detecting the change, the router can then insert it into the design incrementally and create a new bitfile. The new software environment will provide a seamless flow from a concept for a circuit all the way through bitfile generation. Figure 4-1 shows the entire environment from design entry to FPGA programming.

## 4.1 Primitive Part Table

Before the Schematic Capture, Placement, and Routing environment (SCPR) can do anything, it needs to know the names of the primitive parts to be used in a particular lab session. This is because non-primitive parts (those made up of other parts) have to be handled differently. This will be discussed in a later section. In addition to the part names, the SCPR needs to know the shapes of the parts for both schematic and logic placement views and how to instantiate the part in the router. The shapes for the schematic capture and logic placer programs are just lists of lines and dimensions in terms of CLBs, respectively. The instantiation information is just a string describing the part to be sent to the router.

All of this information is read into the Primitive Part Table on startup of the SCPR. When SCPR starts up, it looks for a file with a “.ppt” extension in the directory containing the library parts for that particular lab. Each part in a .ppt file is specified using the format shown in Figure 4.1.

The “lines” field specifies the straight lines used in drawing a part in the schematic capture program. The format is “x1 y1 x2 y2”. Next, the “arcs” field describes the curves to be drawn in the schematic capture program. The first two numbers are the x and y coordinates of the upper-left corner of a rectangle and the second two are the coordinates of the lower-right corner. The arc to be drawn will be an ellipse centered on the center of this rectangle and extending to its sides. The “router spec” field is the string that has to

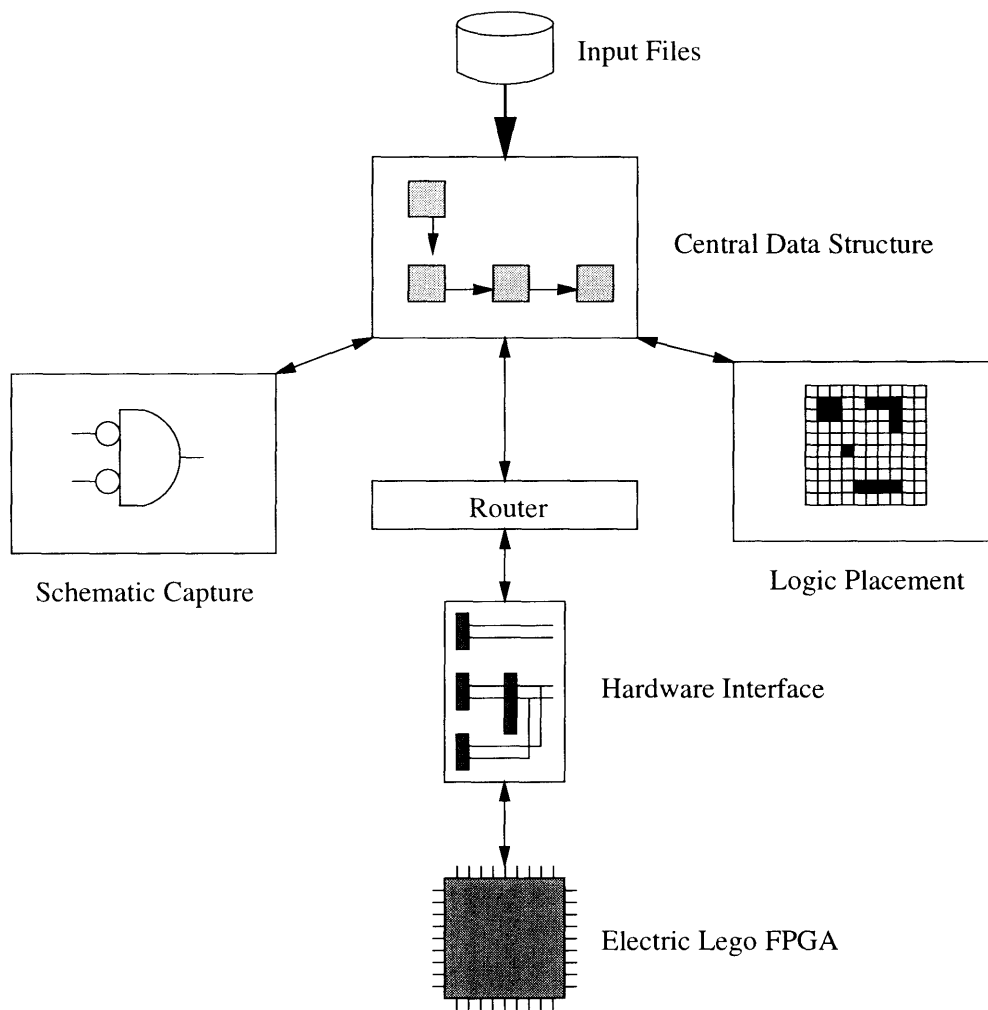


Figure 4-1: Overview of the Electric Lego programming environment

```
#name:  
OR2  
#schematic shape:  
#lines:  
1 2 3 4  
2 5 3 1  
#arcs:  
1 3 4 5  
#router spec:  
OR2(I0, I1 -> O) size(1,1)
```

Figure 4-2: Input format for Primitive Part Table construction

be sent to the router to instantiate the part. This will be discussed in more detail in a later section. The primitive part table is useful in all phases of design creation. The schematic capture program uses it to access information for adding and drawing parts and the logic placer uses it to find the CLB dimensions of parts. The original purpose of the primitive part table, however, was to provide SCPRs input file parsing routine with the names of all of the primitive parts.

## 4.2 Input File Formats

The SCPR package needs two separate files to specify a design. One file will be in the Xilinx Netlist File (XNF) format and the other will be a format designed specifically for the Schematic Capture program. The XNF format describes the design in terms of part and pin names and connections of pins to “nets”, or wires. This format was chosen for maximum compatibility with commercial software. The second file, called a “.sch” file specifies more superficial information necessary for the schematic capture program. It contains information about part placement on the virtual desktop and how wires and busses are to be drawn from one port to another.

### **4.2.1 The Xilinx Netlist File Parser**

Compatibility with a standard netlist format is important because it ensures that the SCPR package can manipulate designs created by other software suites and vice versa. Designs created with SCPR will be completely portable to other environments which support the XNF format. XNF compatibility also allows a user to “mix and match” design tools. For instance, if a user prefers the SCPR schematic capture program but prefers the optimization capabilities of another software package, he can generate an XNF file from the SCPR package and import it into the logic placer of another design environment.

#### **A Completely General XNF Parser**

The first pass at the XNF parser was a GNU Lex/Yacc generate shift/reduce parser. This parser read in tokens from the xnf file and pushed them onto a stack. Anytime a sequence of tokens on the stack matched a pattern defined by the parser, the tokens were removed and replaced by the name of the pattern. This type of parser proved to be less than optimal for our purposes for a few reasons.

First was the problem of limited lookahead. The shift/reduce parser could only look two tokens ahead. This property proved to be inconvenient for the xnf format. In several common cases a certain definition could be specified by many different token patterns. Some of these patterns differed by more than two tokens, making the artificially imposed look-ahead limit of the shift/reduce parser a disadvantage.

Another reason that using an automatically generated shift/reduce parser was not the best decision was the nature of the information to be extracted from the xnf files. The router we are using only needs part of the information that is possible for an xnf to contain. This information reduced the size of the set of tokens we needed to parse to a very small number. Because of this, it was much easier to hard code this small set of tokens into a simple scanner/parser than to specify an automatically generated shift/reduce parser that might not be fully compatible with our platform. The Lex/Yacc parser did, in fact, cause some compilation problem in the Microsoft Windows environment which is to be used for

the host machines.

Another problem with the shift/reduce parser is that it is difficult to keep state information around. This limitation showed itself in the attempt to parse the names for part instances and the nets which connect them. In an XNF both the part instances (e.g. AND gate “A1”) and the net names (e.g. wire “N1”) are chosen from the same set of possible names. Because of the way tokens are handled in the shift/reduce parser this presented a problem. For example, in an XNF file the way tokens following a name are handled differs according to whether the name specifies a part instance or a wire instance. With the possibility of recursion in a hierarchical XNF file, Lex and Yacc provided no mechanisms for keeping enough state to properly parse all designs.

### **A Custom XNF Parser**

The final version of the parser solved all of these problems. Instead of the separate tokenizing routine passing only one token at a time, the parser included in SCPR gets all of the tokens at once. This allows the parser to enter a state (e.g “parsing part” or “parsing net”) with all of the necessary information to specify a design element instead of having to pass control back to a lexer. With GNU Lex and Yacc this passing of control would cause the parser to lose any state information. The limited look ahead problem also went away. The SCPR parser knows that all relevant information is in the group of tokens in its possession. A stack with artificial size limitations is unnecessary.

Once the parser has the tokens, a simple case statement is all that is necessary to convert them into a meaningful representation of the part they specify. There are only three types of elements that the parser needs to look for: parts (ANDs, ORs, etc.), pins, and connections to the I/O pins on the chip (externals). Therefore only three different types of tokens need to be considered as the first token in a group. Any line that does not begin with one of these can be disregarded. Once one of the legal tokens is encountered, the type and sequence of tokens that can legally follow is very well specified. All the parser has to do is confirm that the tokens form a legal sequence and create a representation for the element. This representation is then added to the data structure that is the centerpiece of the SCPR

package.

Since the custom parser is a single procedure as opposed to a complex shift reduce system, it is also much easier to handle the hierarchical nature of XNF files. As was mentioned in the primitive part table section above, it is important for the parser to know the which parts are primitive. This is because non-primitive parts are simply place holders for collections of simpler logic. When the parser comes across a part that is not primitive it knows to look for another XNF file that describes the part. With the Lex/Yacc parser stacks to save data for use upon returning from parsing a hierarchical file have to be manipulated manually. With the custom parser a simple recursive call automatically takes care of all of this state information. This recursion also provides a very natural way to build the main SCPR data structure.

## **4.3 The Main Data Structure**

The SCPR package's main speed advantage derives from its use of one data structure from which the schematic capture, placement, and routing components can glean information. A change made by one is instantly available to the other. The data structure consists of a list of part types and a tree of part instances.

### **4.3.1 Part Types and Instances**

Each node in the part type list contains information that is common to all parts of that type. There is only one node per part type in the design. For instance, no matter how many AND gates there are in a design there will only be one AND node in the part type list. Part types contain information such as the shape of a part and the names and locations of its input and output ports.

The second part of the central data structure is a tree of part instances. This structure must be a tree as opposed to a list in order to accommodate the hierarchical nature of XNF files. Each node in this tree contains information that is unique to each instance of a part type. This includes a unique identifier, locations for the instance on both the schematic

workbench and CLB array, and a list of pins and the signals to which they connect. Each node also contains a pointer to the part type which it instantiates. Appendix A shows the data members of the PartType and PartInstance classes.

### 4.3.2 Adding Parts to the Structure

Parts are added to the structure either by reading them from an XNF file or by adding them in the schematic capture program. The same routine is used in either case. The only difference is where the placement information comes from. There are two different locations associated with each part: CLB location in the placer and position on the virtual workspace in the schematic capture program. When parts are added via an XNF file these positions are read from files in the same directory. The CLB locations are read from an “.xnf” file and the workspace locations are read from a “.sch” file. These are simple text files which associate part instance names with locations. The workspace locations are a requirement and will automatically be included for all designs created by the SCPR schematic program. The CLB locations, however, are optional. If they are not present the parts are marked as not placed and can be placed later. When parts are added in the schematic capture program the schematic location information is taken from where on the workspace the part is dropped. Parts added this way are marked as unplaced in terms of CLBs and must be placed before routing.

There are two routines which are called when a part is added. The first is called AddType. This routine takes two strings and two booleans as arguments and returns a pointer to the part type it added. The two strings are the name of the part (not the unique identifier) and the name of the file that contains its XNF description. The second string is only relevant if the part is non-primitive. The first boolean is a pointer. AddType uses his memory location to inform the caller whether the type has been previously added. The second boolean tells AddType whether or not the part is primitive. AddType does not always add a type, however. As mentioned earlier, the central data structure only contains one entry for each part type no matter how many times the type is instantiated. If the part



type currently being added is already in the list then AddType just returns a pointer to that node. Otherwise the type is added and a pointer is returned.

The second routine, AddPart, creates a part instance. AddPart takes five arguments and returns a pointer to the part just instantiated. The first argument is a pointer to a list of part instances. This specifies where in the hierarchy to add the instance. Argument two is a boolean which tells AddPart whether the part has been placed in the CLB matrix. If the instance has been placed the third argument specifies its location. The fourth and fifth arguments are the name of the part and its unique identifier. A hierarchical design can be created by adding parts to the subpart lists of other parts. In the current version of SCPR hierarchical designs can be read in from XNF files, but cannot be created by the schematic capture program. This was a conscious choice for the early versions of SCPR. Hierarchical capability was simple to incorporate into the XNF parser and was added for generality. Creating hierarchical designs in the capture program is considerably more difficult. Because the 6.004 labs can be set up to use a small number of primitive parts, the first-pass capture program does not include hierarchical capabilities. These parts will not be primitive in terms of the logic they implement, but in the sense that the capture and placement programs can view them as a block and not be concerned with their substructure. Creation of these parts will be described in a later section.

### **4.3.3 Adding Connections to the Structure**

Connections between parts are always established through “nets”. Nets are just wires that carry certain signals. Each pin on a part is connected to a net. Pins on different parts are connected by attaching them to the same net. In SCPR these connections are made by a routine called AddPin. AddPin takes a list of pins, a pin name, a direction (either input, output, bi-directional), and a net name as parameters and returns a pointer to the pin just added. A pin is added to a certain part by calling AddPin with the part’s pin list and the appropriate group of strings.

All of the routines for adding to the central data structure, AddType, AddPart, and

AddPin rely heavily on strings for input and create structures containing a lot of strings. This may not be an efficient way to encode the parts, but it is a by-product of the input and output interfaces available to SCPR. Both the XNF format and the router interface are completely text-based. Strings are needed to save modified circuits back into XNFs and to instantiate parts in the router.

## **4.4 The Schematic Capture Program**

The schematic capture program is a point-and-click graphical user interface that allows the user to select parts from a list of primitives and connect them by dragging wires between ports. The set of parts from which the user can choose is specified by a .ppt file as described above. On startup, SCPR reads the appropriate .ppt file to create a database of primitive parts.

For the purposes of the 6.004 design environment, the schematic capture program does not need all of the features of commercial CAD programs. Really all that is needed is for a student to be able to see the parts and wires in a circuit. In addition, the set of parts for any lab session will be a very well-defined set of primitives. For this reason, the first version of the SCPR schematic capture program only provides for adding and deleting parts and wires from a design. All of the capture program's functions are controlled by four buttons on the left side of the screen.

### **4.4.1 The Schematic Program's View**

The schematic capture program's view on a part instance in the central data structure consists of four elements. First, each instance has a bounding box. This region around the part is used in selecting it for moving or deleting. Also included in the capture program's view is a list of arcs and lines for drawing the part. The lines are contained in a linked list of point pairs. The arcs are maintained as a pair of rectangles. Windows provides graphics routines which draw ellipses to fit inside rectangular regions. Drawing an arc as opposed to a closed ellipse, however, requires erasing part of the ellipse. This is the purpose of the

second rectangle. The first one specifies the shape of an ellipse and the second specifies which part of the ellipse is to be erased. The schematic program also needs to know the location on any text describing the part. This information is stored as a point. The final piece of information is the offset of the part from the origin (upper-left hand corner) of the workspace. This gives the schematic capture program the location at which to draw the part. All of the fields described above are translated according to this offset.

#### **4.4.2 Select Mode**

The first button on the left side of the screen puts the program in “select” mode. In this mode, any mouse clicks select the part or wire under the mouse pointer. Once a part has been selected it can either be moved to a new location on the workbench or deleted from the design. In the current version, selected wires can only be deleted.

Selection is done by a simple search of the central data structure. On a mouse event in select mode, the capture program looks at each part instance to see if the mouse pointer is inside its bounding box. If it is the part is marked as selected. The part can then be moved or deleted from the design. If the pointer is found to not be inside any bounding box, the list of wires is searched. If the mouse pointer is collinear with any two points in a given wire, that wire is marked as selected and can then be deleted.

This linear search procedure is not the most efficient, but the expected size of the designs makes it a reasonable choice. Most designs will have only tens of parts and at most a few hundred wires. A linear search on a list of this size will not cause any noticeable delays.

#### **4.4.3 Part Mode**

The second button puts the program in “part” mode. This is the mode which allows parts to be added to a design. When part mode is entered, a dialog box containing all of the primitive parts available appears. The user selects a part from this list with the mouse and clicks a location on the workbench for the part. The part is added to the central data structure with the same procedures used to add an XNF-specified part. Because of the

primitive part table, all that is needed to add a part is its name. This string can be used to index into the table and retrieve the rest of the needed information.

#### **4.4.4 Wire and Bus Modes**

The third and fourth buttons start the “wire” and “bus” modes. These modes allows a user to connect a pin on one part to either a pin on another or to a wire carrying a certain signal. On the first mouse event after wire mode is entered, a wire is started. From then on the user can define the shape of the wire by anchoring certain points with the left mouse button. The wire can be terminated at the appropriate point, either at a port or on a wire, with a click of the right mouse button. Again the same routine used to add a pin when parsing an XNF file can be used here. All that is needed is the name and type of the starting point (i.e. either a port or a wire) and the name and type of the terminating point. If the wire connects one port to another a unique name is generated. If one of the endpoints is another wire the wire being added is give the name of the wire it connects to. Since busses are just groups of wires, the same methods apply for adding them to a design. The only difference is that there must be an iteration of the wire adding routine for each wire in the bus.

### **4.5 The Logic Placer**

Once a design has been specified in terms of parts and connections another important piece of information must be added before a bitfile can be created. This information is the placement of each of the parts in the array of Configurable Logic Blocks that make up the FPGA. In most commercially available environments this is a very time consuming process. It is so time consuming because they attempt to do the placement automatically. This is helpful if there are hundreds or thousands of parts in a design, but for the purposes of 6.004 the designs will not be nearly this complex. Students will be dealing with a dozen or fewer “macro” parts in most cases. The macros will be made up of hundreds of thousands of primitive parts but these will already have their relative placements inside the macro fixed. All that will remain is for the user of the placement tool to place the macro parts. The

subparts will be placed according to their offset in the macro.

Because there will only be a few parts to be placed, it is reasonable to expect a user to be able to do the placement. It is generally much easier for a person to pick locations for shapes on a two-dimensional area than to have an algorithm automatically do the placement. Hand-placement is also a good idea because if signal routing fails due to the location of a part it is easier and more realizable for the user to “eyeball” a new location.

The placer in SCPR is a graphical user interface, created by James Clark at MIT, where a user can drag parts to appropriate locations on the FPGA. It is divided into two panes, one containing a listing of the parts in the design and one containing a 24-by-24 grid showing the location of parts which have already been placed. The placer’s view on the central data structure includes information such as a part’s shape in terms of CLBs, whether or not a part has been placed on the FPGA, and where on the grid it is located. All non-placed parts are drawn in the list pane and all placed parts are drawn at the appropriate coordinates in the grid pane. When a part is dragged from the list pane to the grid pane it is marked as placed and its location on the grid is recorded in the central data structure. Once in the grid pane, parts can be moved to new locations as necessary.

## **4.6 The Router**

The incremental router was a separate project, so all the SCPR package needs to do is interface to it properly. The interface provided by the router is text-based and in the current version of SCPR communication with it takes place through files. Future versions will use some sort of interprocess communication. The routing process produces the bitfile which can then be downloaded onto the FPGA.

Before a design can be routed, all of the part types must be initialized by the router. This will be done when SCPR starts. When the primitive part table is created, each of the primitives will be sent to the router for creation. These primitives could be anything from an AND gate to an ALU. Each part in the .ppt file for a particular lab session will have a corresponding file which completely describes it in the router’s input format. As each part

is entered into the primitive part table this file will be sent to the router. A two-input AND gate would be specified as follows:

```
defmod and2(A,B -> Y) size (1,1)
{
  clb1= create clb "x = f1*f2; y= g1*g2..." at (0,0);
  connect(a1, clb1.f1);
  connect(b1, clb1.f2);
  connect(a2, clb1.g1);
  connect(b2, clb1.g2);
  connect(q1, clb1.x);
  connect(q2, clb1.y);
}
```

The size field tells the router how many CLBs the part occupies. The “create clb...” line contains the information needed to actually configure the function generators in the CLB and the “connect” commands specify the routing of internal signals through the multiplexers in the CLB. This format can also be used to specify parts that take up more than one CLB. By using the “at” field in the “create clb” line the relative locations of the components of multi-CLB parts can be fixed. For 6.004, parts as complex as an ALU will be specified in this format. As mentioned earlier, this will allow the schematic capture and logic placement programs to view even the most complicated parts as primitive blocks.

The next interaction with the router will be when the design is actually routed. The central data structure contains all of the information needed to route the design. Routing is just a matter of instantiating each part instance in the router’s format. This is done by sending a string describing the part instance router to the router. For instance, to create instance “I1” of a 2-input AND gate in the upper-left CLB on the first lego, the following string would be sent to the router:

```
I1 = create AND2(A,B->Y) at (1,1,1);
```

Once all of the parts in a design have been instantiated the communication with the router will, in some cases, not be over. If a signal fails to route, changes have to be made

to the placement of modules around it. This is where the ability to place parts by hand is important. It is much quicker and easier for a person to judge what changes need to be made than to do it automatically. When a signal fails to route, the router will send a message to SCPR telling it which signal needs to be adjusted. This information will be relayed to the user both as text and in graphical form. Upon routing failure the user will receive a message box with the unique identifier of the unroutable wire. In addition the wire will be highlighted in the schematic view and all parts affected by the failure will be highlighted in the placer view. After the moves any logic to correct the problem, the changes will be sent to the router. The incremental nature of the router will be the source of the SCPR package's speed advantage over other design environments. Instead of having to send the whole design again after changes have taken place, SCPR can just unstantiate the old versions of the changed parts and restantiate the new ones. This incremental ability will also make designing faster in other ways. For example, if a user compiles a design in another environment only to discover a wire is missing the entire design must be recompiled after the wire is added. In SCPR, the wire can be added and only the affected parts will have to be reinstantiated. Unfortunately, a working version of the router was not completed in time to incorporate into SCPR. The foundation is set, however, to just plug the router in when it is finished.

## Chapter 5

# The Host/Lab Kit Interface

After a bit file is created by the router, there is one step remaining to complete the illusion that the design on the Lego is being directly manipulated. The bit file must be quickly downloaded to the lattice. If this step is too slow then all of the effort to generate a bit file as quickly as possible is wasted. Most currently available interfaces, such as the Xilinx XChecker cable send configuration bits serially from the host machine to the FPGA. This is not only slow, but it makes the interface between the host and the FPGA much more complicated. In addition, receiving feedback from a peripheral is not possible through a serial port.

### 5.1 The Parallel Port Interface

The interface selected for communication between the host machine and the Lego lattice is the parallel port. One important reason for this choice is that virtually every PC has a parallel port. This makes it possible to use the lab kit with machines from different manufacturers. With some software porting, it would even be possible to use the kits with either PCs, Macs, or workstations. A second reason is the parallel port's simplicity. A prototype for communication through the ISA bus was designed and built, but was deemed inappropriate for the kits. In order to use this interface, the kits needed to contain a motherboard with ISA slots. The mother board then needed an operating system to



SPP Name	EPP Name	DB-25 Pin	I/O Direction
-Strobe	-Write	1	Output
Data[7:0]	Data[7:0]	2-9	Bi-Di
-Acknowledge	-Interrupt	10	Input
Busy	-Wait	11	Input
Paper End	User Defined	12	Input
Select Input	-Adrstroke	13	Input
-Auto Feed	-Datastroke	14	Output
-Error	User Defined	15	Input
-Initialize	-Reset	16	Output
-Select	User Defined	17	Output
Ground	Ground	18-25	-

Table 5.1: Parallel Port Pinouts

control the bus. This greatly increased both the hardware and software complexity of the kits. As will be shown in a later section, the parallel port requires only a few chips which can be mounted on the main board of each kit. Before describing the design of the parallel port interface, it will be helpful to describe the protocol that makes the interface possible.

### 5.1.1 The Enhanced Parallel Port Protocol

The standard parallel port protocol only allows for the transfer of data from the host to the peripheral. This is not useful for the Lego kits because the host needs feedback from the lattice on the status of down loads. Luckily, the IEEE recently approved the Enhanced Parallel Port protocol (EPP), standardizing a bi-directional mode for parallel ports. EPP proved very suitable for use in the new 6.004 kits in more ways than one.

First, EPP provides the host with a transparent, memory-mapped interface for reading from and writing to the parallel port. The standard parallel port (SPP) is mapped to three registers starting at a base address. For most machines this is hex 378 (0x378). The base register is the write-only Data port. Base+1 is the Status register which contains signals such as Paper End which are commonly used by printers. Base+2 is the Control register. This register is used to send strobe and initialization signals. EPP extends this set by

Port Name	Offset	Mode	Read/Write	Function
SPP Data	+0	Both	Write	Standard data port. Manual strobing.
SPP Status	+1	Both	Read	Status bits from peripheral
SPP Control	+2	Both	Write	Control bits to peripheral
EPP Address	+3	EPP	Read/Write	Generates read or write address cycle.
EPP Data	+4	EPP	Read/Write	Generates read or write data cycle.
User Defined	+5 to +7	N/A	-	-

Table 5.2: Parallel Port Register Definitions

adding five extra registers. The two that are useful for communicating with the lattice are base+3 and base+4. These are the EPP Address Port and EPP Data Port, respectively. Unlike standard mode, where the control signals have to be controlled by driver software, in EPP mode all signals are automatically generated by the port hardware. In addition, both ports are bi-directional and are therefore capable of providing feedback from the lattice.

The second aspect of EPP mode that makes it fit well with the needs of the lab kits is the fact that there are two read/write ports. As will be discussed later, it is very convenient to be able to address different registers in the interface between the host machine and the lab kits. In EPP mode the data port can be used to send data to a register selected by values sent to the address port. In order to use these ports, however, the peripheral must understand the important signals in the EPP protocol.

The discussion here will describe the signals used in manipulating the data port. The same principles apply when using the address port. There are three main signals that are important when interacting with a parallel port in EPP mode: -WRITE, -DATASTROBE (or -ADRSTROBE), and -WAIT. These signals are carried on parallel port pins 1, 14 (17), and 10, respectively. -WRITE is an active-low signal that signals that a write is taking place. -DATASTROBE and -ADRSTROBE are active (low) when there is valid data on the parallel port data pins. The -WAIT signal is a handshake signal that tells the host when

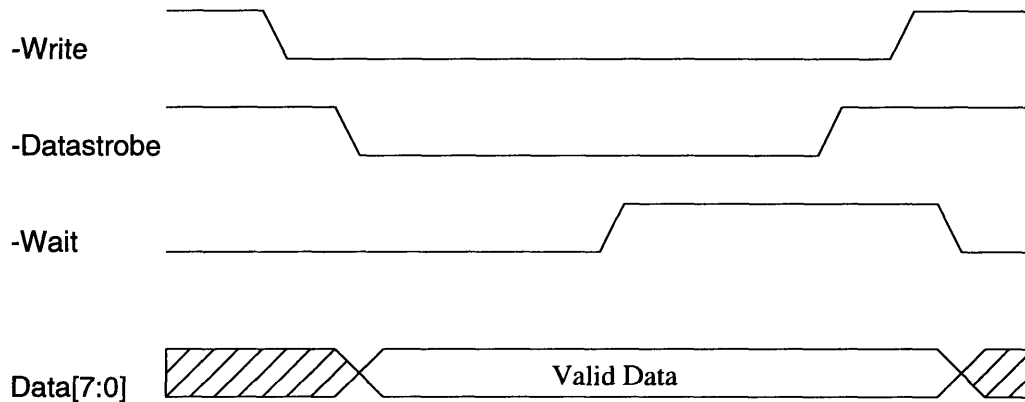


Figure 5-1: Signal Timing for an EPP Data Write Cycle

it can remove the data from the data pins. The host cannot begin a read or write cycle until -WAIT is asserted (low) and cannot end a cycle until -WAIT is high. So when the host writes to the memory location corresponding to the EPP data port the EPP hardware first asserts -WRITE and puts the data on the parallel port data pins. Next, when the data is valid, the -DATASTROBE signal is asserted telling the peripheral that it can now take the data. As soon as the peripheral has the data it deasserts -WAIT allowing the EPP hardware to end the write cycle. After the cycle has ended (i.e. -WRITE and -DATASTROBE are deasserted) the peripheral asserts -WAIT allowing the host to begin another read or write cycle. Figure 5-1 shows the signals generated during a data write cycle.

### 5.1.2 The Labkit Parallel Port Hardware Interface

The parallel port interface is designed for flexibility. There are a few protocols in existence now and may be more in the future. A circuit that would only allow the use of one protocol would prevent the kit from being useful with machines that might not support that particular protocol. Also, if for some reason it becomes necessary to use a different protocol the lab kit should be capable of adapting. The FPGA technology that promises to drastically improve the 6.004 lab experience is also perfect for providing this flexibility.

The parallel port circuit cannot, however, just consist of an FPGA connected directly to the parallel port signals. First, there needs to be some buffering to prevent damage to the FPGA. In addition, there are two modes of communication between the parallel port and the kit. Programming the “gateway” FPGA is slightly different than programming the FPGAs in the Lego lattice. A small amount of logic is needed to switch between these modes.

The parallel port circuit on the main board of the lab kit consists of one Xilinx 4013 FPGA along with three 74LS245 8-bit bi-directional buffers and a 22V10 PAL. The PAL provides the logic for switching between communication modes. The first LS245 buffers the data bits from the parallel port and routes them to Xilinx pins D0-D0. These are the pins that accept data bytes for shifting into the FPGA when programming in Asynchronous Peripheral Mode. This buffer’s direction input is connected to both the PAL and the Xilinx. This allows the PAL to control the direction of data transfer during programming of the gateway chip. After programming is complete, most of the PAL pins, including the one controlling the direction, tri-state, allowing the Xilinx to take control of the parallel port.

The second two LS245s buffer the control and status bits from the parallel port. Since the control and status registers are write- and read-only, respectively, their direction control pins are hardwired. The control bits are routed through the buffer to pins 2 through 5 of the PAL. This allows the PAL to generate the proper signals when the gateway FPGA is being programmed. These signals are also routed to the gateway so that it has access to them after programming is complete. The parallel port status signals are also connected to both the PAL and the FPGA. During gateway programming, the PAL informs the parallel port of the status of the download. After the PAL tri-states, the Xilinx takes control of these bits and uses them to relay the status of the lattice to the host.

The function of the PAL is to translate parallel port signals into programming signals for the gateway Xilinx. There are five signals that the host and the gateway need to communicate to each other during gateway programming: PROG, -WS, -INIT, DONE, and RDY/-BUSY. The PAL routes the parallel port -INITIALIZE signal to the PROG pin on the gateway chip. Holding the -INITIALIZE signal low for 10 ms will prepare the

gateway for programming. Next, the PAL routes the parallel port -STROBE signal to -WS on the FPGA. Configuration bytes are sent to the gateway chip in standard parallel port mode. This choice was made to insure compatibility with all PCs. If a host machine doesn't support EPP it can still communicate with the kit by down loading an appropriate design to the gateway chip. To send a byte, the host simply puts it on the data lines of the parallel port byte writing to the base memory address. A -STROBE pulse generated by writing to the status register (base+2) causes a pulse on the -WS pin of the Xilinx latches the data into the shift register.

The PAL also routes the Xilinx DONE, -INIT, and RDY/-BUSY signals to the PAPER END, -ERROR, and BUSY parallel port pins. The PAL inverts RDY/-BUSY to make it compatible with the parallel port. These allow the host to send bytes only when the FPGA is ready and to detect errors or programming completion. Connecting to the parallel port signals mentioned here was chosen in hopes of being as compatible as possible with existing parallel port software drivers. Luckily, because they all pass through the PAL, these connections are completely changeable. Any of the relevant Xilinx signals can be routed to any parallel port status bit and any parallel port control bit can be routed to the Xilinx -PROG and -WS pins. In addition, the PAL even allows for the logical combination of any of these signals should the need arise. The logic functions implemented by the PAL are shown in Appendix A. The final element in the parallel port circuit is a voltage controlled crystal oscillator. This oscillator generates the master clock for the whole lattice.

### **5.1.3 The Gateway Chip**

The key to communicating with the Lego lattice is the design that gets down loaded to the gateway Xilinx. This design allows the parallel port to interact directly with the PROG, DIN, and CCLK pins on the Lego adjacent to the gateway chip. It also allows the parallel port to access the ports on a Lego command module. The Lego command module is a design that is loaded onto a Lego to allow it to program its neighbors.

## The Lego Command Module

A Lego far from the gateway chip is programmed by putting the command module design on each Lego leading up to it and then putting the desired bit file on the target. The command module, designed at MIT by Anne Wright and Andrew Huang, makes this possible by providing a set of registers for programming neighbors and a set of opcodes for reading and writing those registers and turning the Lego into a “pipe” from one of its sides to another.

The five important registers in the command module are used for programming adjacent Legos. Registers 0 through 3 control the PROG, DIN, and CCLK signals for the north, east, south, and west neighbors. Serial programming mode is used due to a limited number of pins on each side of the Lego. To program the north neighbor, for instance, a configuration stream can be sent to register 0 in the manner described in the FPGA programming section. Register 4 controls tri-state enables for each of the programming registers. To enable register 0 bit zero in register 4 is set to 1, and so on. In order to write to these registers one of the instructions in the command module’s four member instruction set must be used.

The command module’s instruction set includes the READ, WRITE, PATH, and NUKE instructions. READ and WRITE access the internal registers. The PATH and NUKE opcodes are the key to programming an arbitrary lattice. When the PATH command is received on one side of a Lego it goes into pipe mode. In this mode all data and opcodes are passed through the Lego unchanged to the side specified by the PATH command. This continues until a NUKE command is issued to the Lego in pipe mode. The Lego passes this opcode through as usual and then exits pipe mode. Programming a Lego at an arbitrary position in the lattice is just a matter of setting up a path. Once a path is established, the gateway chip can program the Lego just like it would any of its direct neighbors. Once the programming is complete the NUKE command is sent and the process can be started again. The structure of these opcodes is shown in Table 5.1.3.

Data and opcodes are read and written to a Lego via a pair of ports on each of the four sides. Each of these ports consists of eight data lines and two control signals. The VALID signal is pulsed when valid data is present on the data lines. The OP signal differentiates

Command	Opcode	Description
Write	0100aaaa	After this command is received, all data accompanying a VALID pulse will go to the register specified by aaaa.
Read	1000aaaa	Read register aaaa. After putting data on its output port, the Lego that received this command pulses its VALID_OUT line.
Path	0001pppp	After receiving this command a Lego routes all data and opcodes to pppp.  0001 North 0010 East 0100 South 1000 West
Nuke	0010xxxx	A Lego receiving this command passes it through in pipe mode then returns to normal mode.

Table 5.3: Lego Command Module Opcodes

between opcodes and regular data bits. A high OP line means that the bits on the data lines are an opcode. The output port on each side connects to the input port of the neighbor on that side and vice versa.

### The EPP Command Module

The EPP command module is the circuit that is down loaded to the gateway FPGA to allow the parallel port to communicate with the rest of the Legos. It contains six addressable registers for programming direct neighbors and communicating with their Lego command module data ports. Because it sits directly on the top board of the kit, the gateway chip only communicates through its north and south sides. Each of these sides uses one register for direct programming of its neighbor and two registers for reading and writing the Lego command module ports. Addressing these registers is where the EPP address port is useful. It can be used to set up reads and writes to a certain registers.

The address register in the EPP command module is used to latch addresses as they come in from the parallel port. This is done by monitoring the -WRITE and -ADRSTROBE signals. When both signals are asserted the address register is enabled and the bits from the data bus are latched. The address is then sent to a three bit address decoder. The outputs of this decoder are used to select the internal registers. To access the north programming register the host just sends a zero to the EPP address port. All subsequent reads and writes to the EPP data port will access the north programming register.

**The EPP Programming Registers** The programming registers have two functions. When written, the three low-order bits correspond to the PROG, DIN, and CCLK signals on the neighboring Lego. When read, the two low-order bits are the neighbor's -INIT and DONE signals. To program a neighbor the host can send the configuration stream serially to the appropriate register. After the stream has been sent the host confirms that the -INIT and DONE bits are both high. If either is low, an error has occurred during programming. The write port on the programming registers is made up of three enable flip-flops. One of the three low-order data bits is attached to each of the flip-flops. The enable input of each is driven by the "prog write" signal. This signal is driven high when the -DATASTROBE, -WRITE, and programming register enable signals are all asserted. The read port is just a tri-state buffer. When the programming register is read the -INIT and DONE signals are driven onto the parallel port data pins.

**The EPP Status Registers** The status registers are used to control the VALID and OP signals for the Lego command modules on the north and south neighbors. Writing a one to the low bit of the status register produces a VALID pulse for the input data port of the neighboring Lego. This is done by passing the low bit through a three-stage pipeline. After a logic 1 enters the second stage of the pipeline the next falling clock edge clears the first flip-flop. The effect is that setting the low bit of the status register high causes a two clock cycle pulse on the VALID output. The second bit in the status register write port generates the OP signal for the neighboring Lego. This signal is not pipelined and reaches



the Lego two clock cycles ahead of the VALID pulse guaranteeing that it will be at a valid logic level. The third bit clears the third bit of the status read port. The reason for this will become clear shortly.

The read port of the status register allows the host to read the VALID and OP values currently in the register and to detect when data has come in from a neighboring Lego command module. The VALID signal from each neighbor comes into the third bit of the status register read port. If the host is expecting a response from the lattice and this bit is high, then a valid response is ready to be read from the data register. After the byte has been received by the host it writes a one to the third bit of the status register clearing the VALID input signal. The host can then request more data from the lattice.

**The EPP Data Registers** The EPP command module data registers provide access to the input and output data ports on the neighboring Lego command module. These registers, in cooperation with the status registers allow the host to communicate with the lattice. Data from the write port is simply passed to the Lego command module read port on the neighboring Lego command module. The read port gives the host machine the data that was present on the neighbor's output data port at the last rising edge of its VALID signal. To write a Lego command module register the host puts the "Lego write" opcode in the data register and writes 0x3 to the status register. This sets the OP bit high and sends the data bits to the Lego as an opcode. Until another opcode is sent, all data accompanied by a VALID pulse will go to the register specified in the Lego write command. To read a Lego register the "Lego read" opcode is sent. The host then waits for bit 3 (VALID\_IN) of the status register to go high. It then reads the EPP module data register and clears the VALID\_IN bit by writing a zero to the third bit of the status register.

## 5.2 The Parallel Port Software Interface

The user controls the Lego lattice through a command line interface. The following commands are available:

- Load0: Down load a design to the gateway FPGA. This is done in Asynchronous Peripheral Mode. All other down loads are done serially.
- Load1: Down load a design to a Lego adjacent to the gateway chip.
- Path: Establish a path in a certain direction. By default the first Lego is the “master” Lego. The path command changes the master. For example, the “path north” command transfers master status to the north neighbor of the current master.
- Nuke: Transfer master status back to the first Lego.
- Lload: “Lego load”. This command down loads a design to one of the current master’s neighbors. “Lload lego.bit north” down loads the design specified by lego.bit to the current master’s north neighbor.
- Lwrite: Write a value to a Lego command module register. “Lwrite 0 0xff” writes 0xff to Lego register 0.
- Lread: Read an internal Lego command module register. “Lread 0” returns the value in register 0.

**The Load0 Command** The load0 command interacts directly with the parallel port hardware on the kit. To begin programming, the -INITIALIZE signal on the parallel port is asserted. This, through the PAL, resets the gateway FPGA by driving the -PROG line low. After 10 ms the host deasserts -INITIALIZE allowing programming to begin. The configuration bits are then sent a byte at a time through the standard parallel port. This is done by writing the byte to the data port (memory location 0x378) and pulsing the -STROBE signal. The PAL routes the -STROBE signal to the -WS pin on the gateway chip. After each byte is sent, the load0 command waits for all of the bits to be shifted into the FPGA. Since the PAL converts the Xilinx RDY/-BUSY signal into the parallel port BUSY signal, the host just needs to wait for the BUSY signal to be deasserted. After all of the configuration bits have been sent the host checks to see if the programming was successful.

First it reads the parallel port -ERROR bit which is driven through the PAL by the Xilinx -INIT. Next, the host checks the parallel port PAPER END signal. This is driven by the Xilinx DONE pin, again through the PAL. If DONE is asserted and -ERROR is not, load0 informs the user that programming was successful. Otherwise, an error is signaled.

**Load1** This command programs the first Lego in the lattice. Having one command for programming all Legos, whether the first or not, would be more elegant. This ability will be added in the future by making the EPP command module a slightly modified Lego command module. The flexibility built into the parallel port makes this possible. For the first version of the parallel port, however, the gateway chip design necessitates a separate command for the first Lego.

This command programs the first Lego by using the programming registers in the EPP command module. First, the appropriate programming register is selected. This is done by writing to the EPP address port (memory location 0x37b). Once this register is selected the configuration stream can be sent serially by writing to the EPP data port (0x37c). After the stream is sent, the programming register is read. If bit 0 (-INIT) and bit 1 (DONE) are both high the programming was successful.

**The Load Command** The load command works almost exactly as the load1 command except that it uses the programming registers of the master Lego command module instead of the programming registers of the EPP command module. First, the appropriate register is selected with the “lego write” opcode. From this point the bits are sent exactly as the bits in the load1 command except that they are sent to the EPP command module data register instead of the programming register. The lego write command ensures that all of configuration data is sent to the proper Lego command module register. After all of the programming information is sent the status of the down load is checked by reading the Lego programming register. The read port on a Lego programming register is exactly the same as its EPP module counterpart. If the DONE and -INIT bits are high, programming was successful.

**Other Commands** The rest of the commands just send the proper opcode to the adjacent Lego. This is done by writing the opcode to the EPP data register and writing a three to the EPP status register. The opcode is sent to the current master Lego for execution.

## Chapter 6

# Conclusion and Possible Extensions

The purpose of the project described in this paper was to lay a foundation on which a complete environment for the new 6.004 lab kits could be built. The current software only provides basic functionality without the advanced features present in commercially available software. The basic framework for a vastly superior FPGA programming environment is present, however. The SCPR package's use of a central data structure for communication between the schematic capture program, the placer, and the router gives it the potential to very closely simulate the experience of physically wiring a circuit. It can drastically reduce the time required to take a circuit from concept to implementation as well as the time to make changes when debugging and tuning a circuit. This ability will be critical in the 6.004 setting where students will be designing a new circuit every week. Design loop delays would be frustrating, discouraging, and would take away from the effectiveness of the labs. As the SCPR environment develops it should provide a fast, powerful environment that is easy to use.

Some improvements might include optimization and router interface routines that run as background processes. These could run constantly while the user is inputting and editing a design. Instead of the user having to explicitly optimize a design or send it to the router,

the background processes could continuously optimize the logic and update the bit file incrementally as the design changes. When the design is ready to down load the bit file will immediately available.

The down load interface can also be greatly improved. The Legos provide a mechanism for determining the shape of the lattice. If the PROG signal for a certain side is driven low and the DONE signal remains high then there is no neighbor on that side. By doing a tree search on the lattice using this technique, the host can determine the topology of the lattice. The host can then present the user with a graphical two- or three-dimensional representation of the lattice. Instead of typing a series of instructions in a command line interface, the user could simply drag a file onto the picture of the desired Lego. The graphical user interface could generate all of the "path" and "load" commands without exposing the circuit designer to this complexity.

The SCPN software ideas along with the parallel port interface provide a solid foundation for the 6.004 Electric Legos environment. The benefits of the Lego approach are not limited to 6.004, however. All digital design classes can be enhanced through the use of this technology. For teaching purposes the virtual environment designed for 6.004 is just as good as building circuits from wires and MSI logic. When circuits become faster and more complex the advantages of FPGA technology far outweigh any benefits obtained from physically wiring a circuit, especially when coupled with a fast programming environment. With the need for rapid prototyping of digital designs outside of the classroom and the gaining popularity of research in reconfigurable computing, the Electric Lego technology and the SCPN environment will prove to be valuable tools in many areas.

## Appendix A

# Central Data Structure Classes

---

*/\* PlacerPart.h : interface of the CPart\* classes \*/*

```
class CPartType : public CObject
{
public:
    CString m_name;
    BOOL m_is_primitive;
    CSchematicPart m_schematic_info;
    CPlacerPart m_placer_info;
};
```

 10

```
class CPartInstance : public CObject
{
public:
    CString m_UID;
    CPoint m_schematic_loc, m_placer_loc;
    CPartType *m_part;
    CList<CPartInstance*, CPartInstance*> m_subparts;
    CList<CPin*, CPin*> m_pins;
};
```

 20

```
class CSchematicPart : public CObject
{
public:
    CRect m_bounding_box;
    CList<CRect*, CRect*> *m_arcs;
    CList<CRect*, CRect*> *m_erasers;
    CList<CLine*, CLine*> *m_lines;
    CList<CPort*, CPort*> *m_ports;
};
```

 30

```
class CPlacerPart : public CObject
{
public:
    CRect m_CLB_dimensions;
    CPoint m_CLB_location;
};
```

40

---



## Appendix B

# Routines for Building the Central Data Structure

---

```
POSITION AddType(char *name, char *UID, BOOL is_primitive)
{
    POSITION pos = root_partlist->GetHeadPosition();
    /* Look through part type list to see if already exists */
    while (pos) {
        CPartType *part = root_partlist->GetAt(pos);
        /* If type already present return pointer */
        if (part->m_name == name) {
            return (pos);
        }
        root_partlist->GetNext(pos);
    }
    /* Otherwise add the type and return a pointer to it. */
    root_partlist->AddTail(new CPartType(name, name, is_primitive));
    pos = root_partlist->GetTailPosition();
    return (pos);
}
```

10

```
POSITION AddPart(CList<CPartInstance*, CPartInstance*> *instances, char *name, char *UID)
{
    POSITION pos = instances->GetHeadPosition();
    /* Search for position for new part (alphabetical order). */
    while (pos && strcmp(instances->GetAt(pos)->m_part->m_name, name) < 0) {
        instances->GetNext(pos);
    }
    /* Create the new part and insert it. */
    CPartInstance *part = new CPartInstance(name, UID);
```

20

```

if (!pos) {
    instances->AddTail (part);
    return (instances->GetTailPosition());
} else {
    instances->InsertBefore (pos, part);
    instances->GetPrev(pos);
    return pos;
}
}

```

30

```

POSITION AddPin(CTypedPtrList<CObList, CPlacerPin*>* pinlist, CString name, char *dir, char *net, char *inv)
{
    enum direction d;

    if (!strcmp(dir,"I")) d=DIR_INPUT;
    else if (!strcmp(dir,"O")) d=DIR_OUTPUT;
    else d=DIR_BOTH;
    pinlist->AddTail( new CPlacerPin( name, d, net, *inv?TRUE:FALSE));
    return pinlist->GetTailPosition();
}

```

40

## Appendix C

# Parallel Port Programmable Array Logic Specification

```
module pport
  TITLE 'pport PAL for Electric Legos
  pport device 'P22V10';
```

```
"Input pins
CLK pin 1;
STROBE_ pin 2;
AUTOFEED_ pin 3;
INITIALIZE_ pin 4;
SELECT_INPUT_ pin 5;
PIN_6 pin 6;
PIN_7 pin 7;
PIN_8 pin 8;
PIN_9 pin 9;
INIT_ pin 10;
DONE pin 11;
RDY_BUSY_ pin 13;
```

```
"Output pins
PIN_14 pin 14;
PIN_15 pin 15;
ERR_ pin 16;
SELECT pin 17;
PAPER_END pin 18;
BUSY pin 19;
ACKNOWLEDGE_ pin 20;
```

```
WS_ pin 21;  
PROGRAM_ pin 22;  
DIR pin 23;
```

EQUATIONS

```
PIN_14.OE = 0;  
PIN_15.OE = 0;  
ERR_.OE = !DONE;  
SELECT.OE = !DONE;  
PAPER_END.OE = !DONE;  
BUSY.OE = !DONE;  
ACKNOWLEDGE.OE = !DONE;  
WS_.OE = !DONE;  
PROGRAM_.OE = 1;  
!ERR_ = !INIT_  
SELECT = 1;  
PAPER_END = 0;  
BUSY = !RDY_BUSY_  
ACKNOWLEDGE := !WS_  
!WS_ := !STROBE_  
!PROGRAM_ = !INITIALIZE_;
```

```
end pport;
```

# Bibliography

- [1] Lazzaro, John P. *The LOG System* 1996: Physics of Computation Group, Caltech  
URL=<http://www.pcmp.caltech.edu/chipmunk/document/log/index.html>.
- [2] Pratt, Gill A. and Nguyen, John *Distributed Synchronous Clocking* Transactions on Parallel and Distributed Systems, vol. 6, pp. 314-328, Mar. 1995.
- [3] Shoemaker, David and Metcalf, Chris and Ward, Steve *NuMesh: A Communication Architecture for Static Routing* 1995: NuMesh Group, MIT LCS.
- [4] Xilinx, Inc. *The Programmable Logic Data Book* 1995: Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124-3400.