

A Field-Programmable Prototyping Board: XC4000 BORG User's Guide

Pak K. Chan*

UCSC-CRL-94-18
April 1994 (6/27/95, 12/11/98 revised)

Board of Studies in Computer Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

The XC4000 BORG board is a PC-based prototyping board with two “user” FPGAs, two “routing” FPGAs, and a fifth FPGA which implements the glue logic for the PC bus. The BORG board is a reusable educational tool intended for a variety of classes; the BORG board, its toolset, and the reprogrammability of the FPGAs further reduce the time/cost of constructing prototypes using FPGAs. This report documents the design, implementation, and the use of BORG: A Field-Programmable Prototyping Board.

*Development of the XC4000 prototyping board is supported in part by National Science Foundation Grant MIP-9111607 and Xilinx Inc.

Contents

1. Introduction	5
1.1 Field-Programmable Prototyping Boards	5
1.2 What BORG Is?	5
1.3 Xilinx XC4000 FPGA parts	8
1.4 Limits on the number of connections between the FPGAs	9
1.5 About this User's Guide	9
2. Installation	10
2.1 What Do You Need?	10
2.2 Software Retrieval and Installation	10
2.3 Hardware Installation	11
2.4 Testing the BORG Board	14
3. Simple Demonstrations	17
3.1 A Tetris Machine	17
3.2 A Maze Solver Machine	18
4. Principle of Operation	21
4.1 Status indicators	21
4.2 Stand-alone BORG board	21
4.3 BORG board as a Peripheral Device of the PC/XT	22
4.4 Put the BORG Board Inside or Outside the PC?	22
4.5 I/O Address Mapping	22
4.6 Memory Mapping	25
4.7 Hardware Interrupt Channel	26
4.8 DMA Channel	27
4.9 Configuring the controller X0 FPGA	27
4.10 Programming the R1, X1, R2 and X2 FPGAs	28
4.11 Global Reset	30
4.12 Readback	30
4.13 JTAG Boundary Scan	30
4.14 System Clock and Single Step	30
4.15 On-board SRAM and arbitration	32
4.15.1 8K×8 SRAM	32
4.15.2 Dual-port SRAM arbitration	33
4.16 Limits on the Number of Connections Between the FPGAs	34

5. Software	37
5.1 Memory related programs <code>mtest</code> and <code>inspect</code>	37
5.2 Board Wiring test program <code>Scan</code>	38
5.3 Pin assignment program <code>assign</code>	38
5.3.1 Projects, Demos and their MCS files	38
6. Design flow	40
6.1 Introduction	40
6.2 Details	40
7. ASSIGN (Ver 3.0) A Pin Assignment Program for BORG Prototyping Board	47
7.1 Preface to earlier versions	47
7.2 Assign as a Pin Assignment Program	47
7.2.1 Place in the design process	47
7.2.2 Command Line Arguments	49
7.2.3 An Environment Variable	50
7.2.4 Alias Files	50
7.2.5 <code>Rx.info</code>	52
7.2.6 Examples of using <code>assign</code>	55
7.2.7 Xilinx XC3000 Series Design	55
7.2.8 XC4000 Series Design	55
7.3 I/O Specification File	55
7.4 BORG Wiring File	57
7.5 Theory of ASSIGN	57
7.6 Problem Description	57
7.7 Graph Reduction	58
7.8 Augmentation	59
7.9 Main Program Loop	60
7.10 Performance	61
7.11 BORG wiring connections	64
7.11.1 XC3000-series BORG wiring connections	64
7.11.2 XC4000-series BORG wiring connections	69
8. Using the Protoboard and Schematic Drawings	73
8.1 Proto-area, Common Anode LEDs	73

9. Guide to Some Laboratory Experiments	78
9.1 Creating user I/O ports in R1	78
9.2 Hardware Interrupt and Interrupt Service Routine	82
9.3 Synchronization Problem	87
9.4 Music Lab	90
9.5 DMA Lab	96
9.6 Boundary Scan Lab	103
9.7 Possible Term Project Description	103
9.8 Initialization of the Bottle	104
9.8.1 Pill encodings	105
9.9 Initialization of the Dr. Mario Machine	105
9.10 Handshake and Timing	105
9.11 Project	106
9.12 Design of a Dr. Mario player	107
9.13 The game environment	107
9.14 What will be finalized later?	107
9.15 Evaluation	108
9.16 Your responsibilities	108
9.17 Suggestion	109
9.18 Initialization of the Bottle	109
9.18.1 Pill encodings	110
9.19 Initialization of the Dr. Mario Machine	110
9.20 Handshake and Timing	110
10. Maze Runner project report	115
10.1 Maze Runner Specifications	115
10.2 Hardware Requirement	115
10.3 Host Program	115
10.4 Design and Implementation	116
10.4.1 Algorithm	116
10.4.2 Implementation	116
10.4.3 R1: The I/O Port	117
10.4.4 X1: The Brain	117
10.4.5 Finder Box	117
10.4.6 Mover	123
10.4.7 Memory Controller Signals	123
10.4.8 Selector	124
10.4.9 Status	124
10.4.10 Direction Processing Logic	124
10.5 R2: The Memory Controller	124
10.5.1 Memory I/O	124

10.5.2	6-Bit Up/Down Counter (C64BUDRD)	132
10.5.3	5-Bit Up/Down Counter (C32BUDRD).	132
10.5.4	Counter Control Logic	132
10.5.5	I/O pads, buffers, and tri-state buffers	134
10.6	Testing and Verification	134
10.7	Timing and Chip Utilization	134
10.8	Credits	135
11.	Troubleshooting	136
12.	Acknowledgements	138

1. Introduction

1.1 Field-Programmable Prototyping Boards

Field-Programmable Gate Arrays (FPGAs) provide a medium to accelerate the process of prototyping digital designs. For designs incorporating multiple FPGAs, the bottleneck is now the process of wire-wrapping, bread-boarding, constructing a printed circuit board, or constructing a multi-chip module. In addition to being time consuming, these processes cannot be carried out until all FPGA designs have been completed (placed and routed), since locking or preassigning I/O pins often prevent FPGA place-and-routers from completing the routing.

To circumvent this bottleneck, FPGAs can be used as re-programmable interconnection chips. The BORG, as shown in Fig. 1.1, is a PC-based prototyping board that contains two *user* FPGAs, two *routing* FPGAs; a fifth FPGA implements the glue logic to the PC bus.¹ To facilitate the design process using the BORG board, algorithms and tools have been developed to aid in the configuration of the routing FPGAs.

The BORG board, its toolset, and the reprogrammability of the FPGAs further reduce the time/cost of constructing prototypes using FPGAs. There are two versions of the BORG boards. Twenty five XC3000 BORG boards were built in 1992, and the XC4000 boards were manufactured in March 1994. This document describes the XC4000 BORG board. It documents the design, implementation, and the use of BORG: A Field-Programmable Prototyping Board.

1.2 What BORG Is?

The BORG board is a reusable PC-based educational tool intended for classes such as logic design, advanced logic design, processor design, and introduction to ASIC design. The BORG board uses the XC4000 family Field-Programmable Gate Arrays (FPGAs). The XC4000 FPGAs are reprogrammable, so one BORG board can be shared by more than one group at the same time. With one XC4002A FPGA on the board, the BORG board can support a 1,000 gate-count design. When it is populated with four XC4010D FPGAs, it can accommodate a 40,000 gate-count design. However, the BORG board is *not* a supercomputer nor a high-performance “generic” processor. Production of 100 BORG boards in March 1994 is generously supported by Xilinx Inc. Half of the boards produced have been (or will be) distributed for free.²

¹P. K. Chan, M. Schlag, and M. Martin, “BORG: A reconfigurable prototyping board using Field-Programmable Gate Arrays,” in *Proceedings of the 1st International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, (Berkeley, California, USA), pp. 47–51, Feb. 1992.

²The manufacturing cost of a populated XC4000 BORG board is US\$250.00 as of March 1994. Contact d1am@xilinx.com for details.

You can install the BORG board internally to a PC with XT/ISA bus; it will occupy one 8-bit XT expansion slot. This is not the most convenient way to use the BORG board. With the help of the protozone adapter card³ which extends the XT bus signal to a 50-pin ribbon cable, the BORG board can be used *externally* to a PC.

The BORG board has 5 programmable FPGAs, and all of them can be programmed by a user. There are two *user* FPGAs, two *routing* FPGAs, and a fifth FPGA (X0) that implements the glue logic to the PC bus as illustrated in Fig. 1.2.

The glue logic FPGA (X0) is programmed by a serial PROM on power-up. With the appropriate setting of one jumper and dip switches on the BORG board, you can also program X0 with the Xilinx `xchecker`. The rest of the FPGAs can be programmed directly from the PC or by the `xchecker` hardware and software (see Section 4.2 of Chapter 4).

The PC and the FPGAs can communicate using port I/O, interrupts, the shared memory on the BORG, or DMA transfers. Port I/O is the simplest and fastest, while DMA is the most complicated and surprisingly slow. Just as any other I/O expansion card (disk controller, parallel port, serial port), you need to map the BORG board I/O ports, interrupt channels, DMA channels into the PC's valid I/O space, memory space, or channel numbers. Section 9.1 of Chapter 9 describes the procedure for constructing your own I/O ports in the FPGAs, and Section 4.7 illustrates the basic interrupt structure.

There is a built-in dual-ported 8K × 8 SRAM on the BORG board. The SRAM is shared between the FPGAs and the PC. Naturally, it is mapped into the PC's memory address space. Access to the SRAM by the PC and user FPGAs is arbitrated by X0. The arbitration can be performed under program control as detailed in Chapter 4.

Different designs run at different speeds. With the XC4000-6 speed grade part on the board, a typical design runs at 8MHz. A 8 MHz TTL clock is supplied on the board as the system clock. This clock can be further divided down to accommodate lower speed designs, refer to Chapter 4 for details.

With multiple-FPGA designs, connecting the signals between the FPGAs is an additional task that must be incorporated into the design flow. User FPGAs are placed and routed individually, and the I/O (pin) assignments of the individual FPGAs do not ordinarily match the constraints on the board. You can use the tool `assign` to match up the pin assignments so that the signals between the FPGAs are correctly connected. `Assign` is described in Chapter 7, and multiple-chip design flow is in Chapter 6.

You will have design projects that need components which are not on the BORG board. For example, you will need operational amplifiers and a digital-to-analog converter in conjunction with an FPGA to build a frequency analyzer; or you will need a piezoelectric buzzer and some transistors to build a digital music synthesizer. A protoarea on the left-hand side of the prototyping board is there to accommodate any extra components.

³Developed by Stanford University, Professor Abbas El Gamal's group. Available from — Proto Tools, 3500 Granada Avenue #156, Santa Clara, CA 95051, Attn: Kalon Goodrich. email: kalon@cup.portal.com

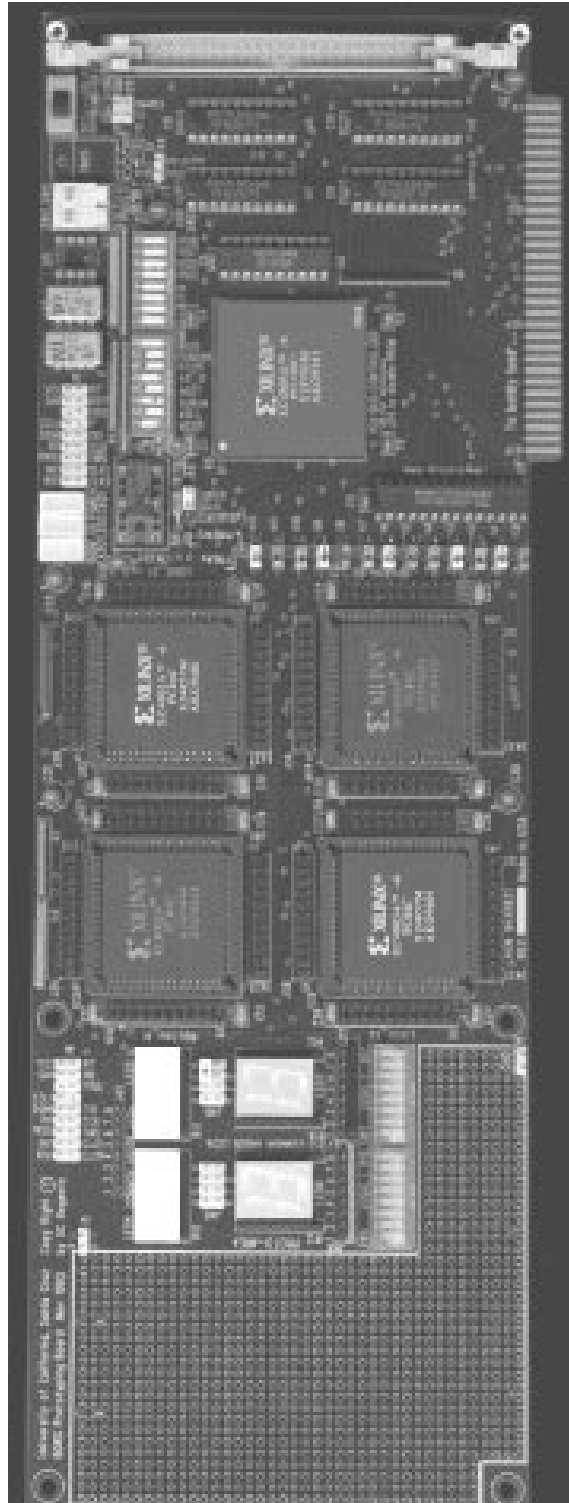


Figure 1.1: A portrait of the XC4000 BORG board.

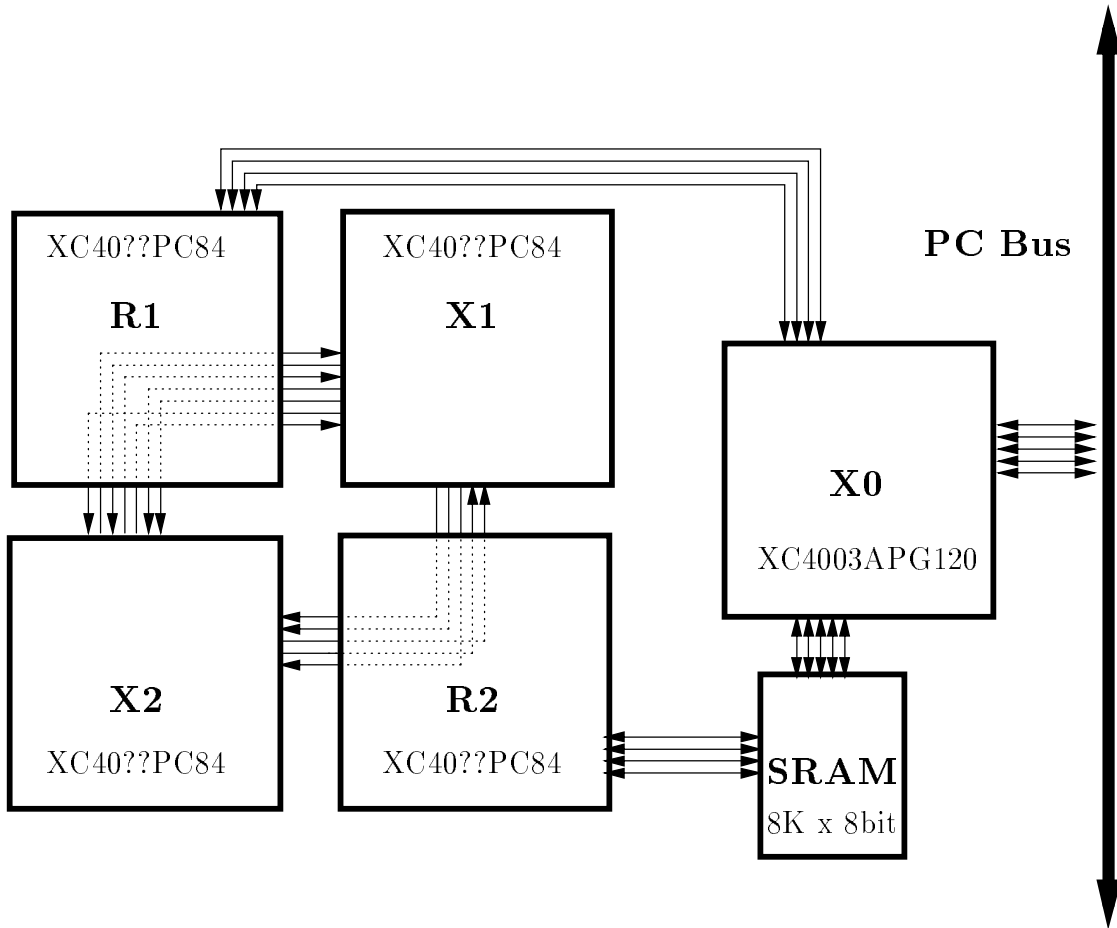


Figure 1.2: Connections between the user FPGAs, **X0** and the PC

Some simple laboratory experiments are presented in Chapter 9 to illustrate some uses of the BORG board. Projects which have used the BORG board in the past include Tetris machine, Dr. Mario machine, and a mazer runner.

1.3 Xilinx XC4000 FPGA parts

The XC4000 BORG board can be populated with 2 *user* Xilinx XC4000 family FPGAs **X1** and **X2** and 2 *routing* FPGAs **R1** and **R2**. **R1** and **R2** connect the two *user* FPGAs together electronically and also provide connections to the 8K×8 dual-port SRAM, the PC bus (via **X0**), and other devices. Figure 1.2 illustrates the basic concept. We shall refer to **R1**, **X1**, **R2**, **X2** collectively as the *ASICs*.

The ASICs can be any one of the XC4000 FPGAs in a 84-pin PLCC package, for example, XC4002PC84, XC4002APC84, XC4003PC84, XC4003APC84, XC4004PC84, XC4005PC84, and XC4010DPC84 with either -5 or -6 speed grade parts. These PLCC packages are pin-to-pin compatible.

For introductory-level classes, you may not need all the ASICs. The ASICs can be extracted from the BORG board using a PLCC-chip extraction tool.

1.4 Limits on the number of connections between the FPGAs

Some of the I/O pads on **R1** and **R2** are used to support the dual-ported SRAM and port I/O communications with the PC. Thus although the number of user pads available on a 84-pin PLCC package is 54, the **maximum** number of connections between **X1** and **X2** which can be realized with **R1** and **R2** is **38**, with the plastic jumpers of J11-J23 on the left side.

With the plastic jumpers of J11-J23 on the right side, the **maximum** number of connections between **X1** and **X2** which can be realized with **R1** and **R2** is **32**. Section 4.16 elaborates this limitation further.

The BORG board has been tested on 8MHz, 11MHz, and 13MHz buses (note: bus speed not CPU speed) with 386/486 DX-33, DX-40, and DX-50 CPUs; it has not been tested with a 33MHz PC bus.

1.5 About this User's Guide

This user's guide consists of the following chapters:

Chapter 2 describes how to install the software and hardware for the first time users, and a 4-step procedure to test the BORG board.

Chapter 3 demonstrates two multiple-FPGA designs: a Tetris machine and a maze solver machine.

Chapter 4 describes the detailed operation of the BORG board and its controller interface with the PC.

Chapter 5 describes some utility programs.

Chapter 6 describes the complete design flow using multiple FPGAs, and the software tools that you will need to use the BORG board with multiple FPGAs.

Chapter 7 details the pin assignment program `assign` that is essential for designing multiple FPGAs.

Chapter 8 describes the bits and pieces that are needed to use the BORG board from a "hardware" perspective.

Chapter 9 suggests a range of projects of varying degree of difficulties.

2. Installation

This chapter describes how to install the BORG board inside or outside a PC/XT. The hardware and software you will need is listed in Section 2.1. Sections 2.2 and 2.3 guide you step-by-step through the installation (and retrieval) of the software, and installation of the BORG board, respectively. After the installation, in Section 2.4 you will test the functionality of the BORG board. Although the BORG boards were tested by the manufacturer (BAT PC Technology of Milpitas, CA) before shipment, you may want to test your BORG board one more time just to be sure.

2.1 What Do You Need?

In addition to a PC/XT, you will need internet access to retrieve the software package and this user's guide(!). You need the following hardware and software to use the BORG board:

1. Xilinx XC4000 FPGA core implementation tools.
2. An `xchecker` cable.
3. An IBM compatible Personal Computer (PC/XT), with 1 Mbyte of available storage space, and an available 8-bit expansion slot.

This machine will be used as a prototyping machine.

4. Some vacant I/O port addresses on the PC/XT.

The default address is `0x30X` (`0x300` to `0x30F`). See Fig. 2.6 for other options.

5. Some vacant 8K-byte memory addresses on the PC/XT.

The default base address is `0xd0000h`. See Fig. 2.6 for other options.

Only items #3-5 are required to test the BORG board.

2.2 Software Retrieval and Installation

You need to have internet `ftp` access. All the software are available by `ftp` to the internet depository `ftp@cse.ucsc.edu`(128.114.134.19). Login as `anonymous` and use `yourname@your.host.name` as the password (for our records).

```
% ftp ftp@cse.ucsc.edu
ftp > user anonymous
Connected to ftp.
220 ftp FTP server (Version wu-2.1c(13) Fri Feb 18 10:49:37 PST 1994) ready.
ftp> Name: anonymous
ftp> Password: yourname@your.host.name
ftp> cd pub
ftp> cd borg
ftp> binary
ftp> get borg.zip
ftp> get pkunzip.exe
ftp> quit
```

At this point you have obtained the BORG distribution `borg.zip` in zip format, and a public domain program `pkzip` to unpackage the distribution. Transfer both files to your PC. Now assuming the files you ftp'ed are on drive A:, on your PC do

```
C:> mkdir borg
C:> cd borg
C:> copy a:pkzip.exe
C:> copy a:borg.zip
C:> set borg=0x300
C:> pkunzip -d borg.zip
```

Don't forget the "-d" option. Compare the result of the directory listing below.

```
C:> dir/w
```

with the following files and directory

```
[.]          [..]          BD.EXE       SCAN.EXE     ASSIGN.EXE
MTEST.EXE   TESTME.BAT    INSPECT.EXE  CLOCK.EXE    MAZE.EXE
ARBIT.EXE   SETASSIG.BAT  [DESIGN]     #README     PORTEST.EXE
[EMPTY]     BSCAN.EXE    [MCS]        ISR.COM      INTPC.EXE
CLEAR.EXE   [ASSIGN]     [SRC]        DEFAULT.EXE  TETRIS.EXE
           25 file(s)      ?????? bytes
```

Congratulations, you have successfully installed the package if there are no discrepancies.

2.3 Hardware Installation

Figure 2.1 illustrates the location and function of the BORG board components. For this installation, you need to locate jumpers J3, J11-J23 and J24, and the red dip switches SW1 and SW2.

If you DO NOT have a protozone adapter card, then you will install the BORG board in **add-in mode** as follows:

1. Turn the PC power off.
2. Set the dip switches SW1 and SW2 on the BORG board according to Fig. 2.5.
3. Place the plastic jumpers at locations J11-J23 and J24 on the two left pins (the two pins closest to the proto-area) as in Fig. 2.5.
4. Plug the BORG board card into a PC expansion slot as shown in Fig. 2.4.
5. Turn the PC power on.
6. Go to Section 2.4.

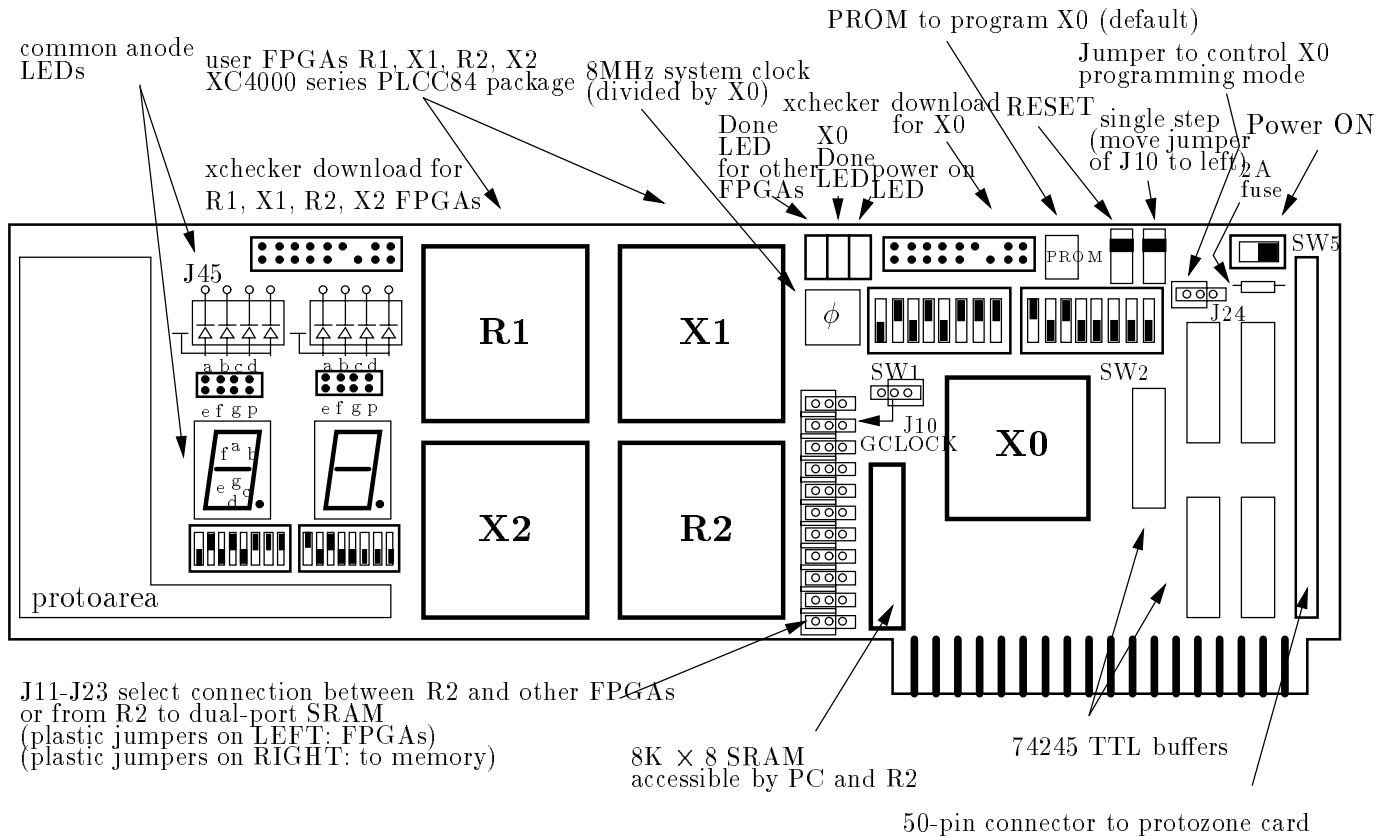


Figure 2.1: BORG board and some of its features.

If you DO have a protozone adapter card, then you can install the BORG board in **host mode** as follows:

1. Turn the PC power off.
2. Set the dip switches SW1 and SW2 on the BORG board according to Fig. 2.3.
3. Place the plastic jumpers at locations J11-J23 and J24 on the two left pins (the two pins closest to the proto-area) as in Fig. 2.3.
4. Plug the protozone adapter card into a PC expansion slot.
5. Connect the protozone adapter card to jumper J3 of the BORG board using the 50-pin flat ribbon cable accompanying the protozone card as illustrated in Fig. 2.2.
6. Turn the PC power on.
7. Go to Section 2.4.

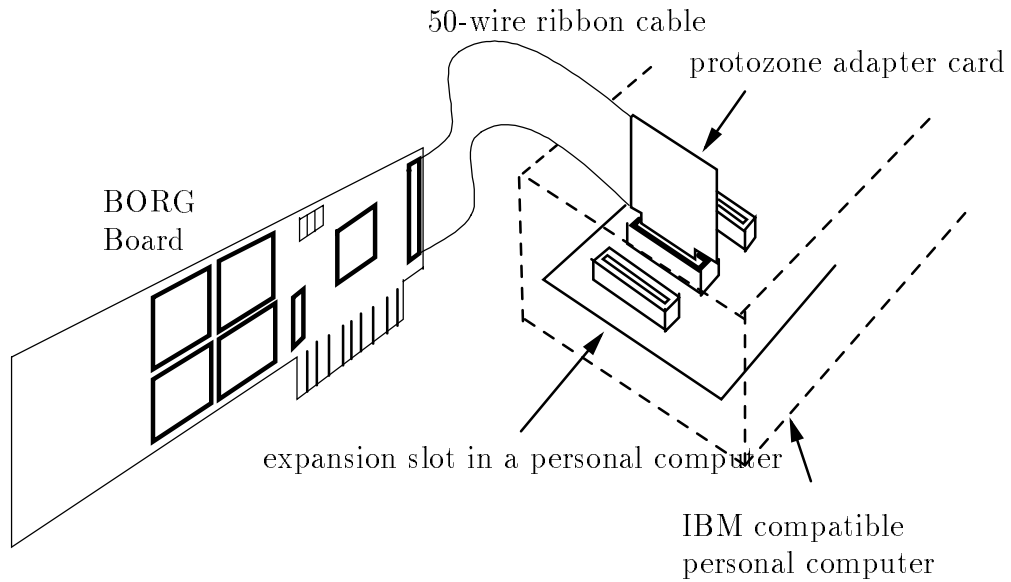


Figure 2.2: Using the BORG board in host mode

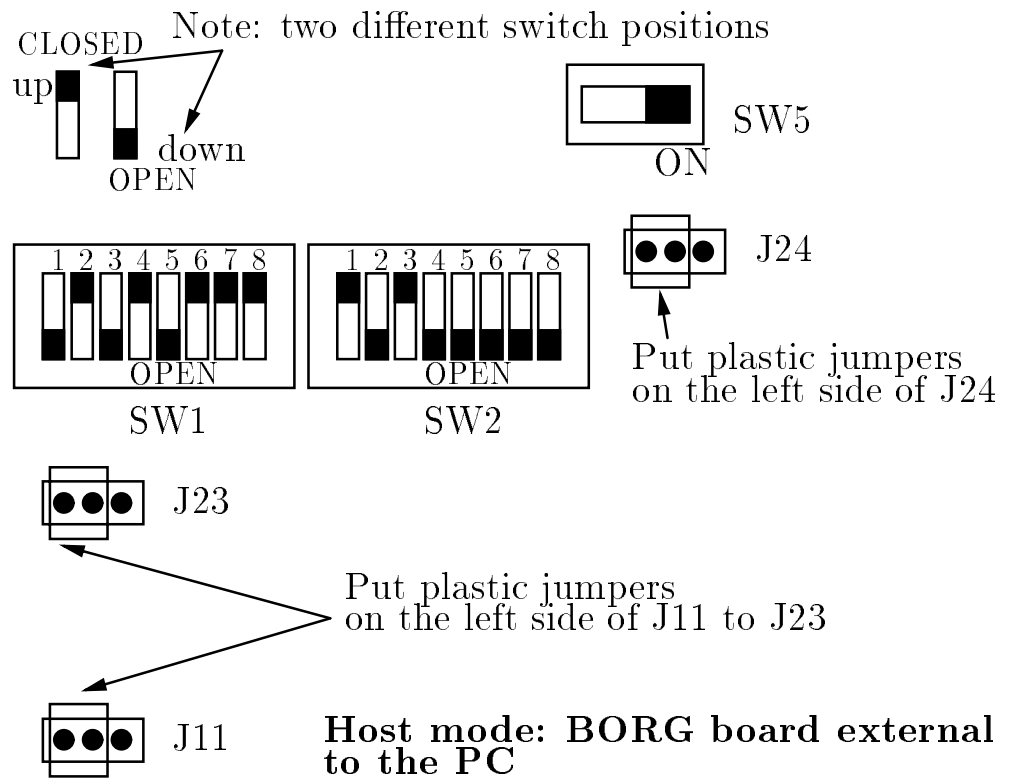


Figure 2.3: Setting for testing BORG board (host mode) with port address 0x30X and memory based address 0xd0000h.

2.4 Testing the BORG Board

These tests require:

- I/O port addresses: `0x30X` (`0x300` to `0x30F`) must be vacant. These are the default I/O port addresses. See Fig. 2.6 for other options.
- Memory address: Also the 8K-byte memory addresses with base address `0xd0000h` must be vacant. See Fig. 2.6 for other options.

Now, take the following steps:

1. Slide SW5 to ON to supply power to BORG board
2. LED1 & LED2 of BORG board should turn ON, and LED3 should be OFF. If not, proceed to the diagnostics in Chapter 11 after checking that the jumper J24 is correctly positioned.
3. Run the `bd` program as shown below:

```
C:> bd mcs\scan.mcs
```

Wait for LED3 to turn ON (this will take a few seconds and all three LEDs LED1, LED2, and LED3 will be ON). If not, proceed to the diagnostics in Chapter 11.

4. Run the `scan` program as shown below:

```
C:> scan
```

It should report:

```
Board scan test done.
Datain -> 0
Board test passed. Accept BORG board.
```

If not, proceed to the diagnostics in Chapter 11.

5. Run the memory test program as shown below:

```
C:> mtest
```

It should report:

```
Finished 8192 bytes. Total errors 0.
```

If program does not report 0 errors, then proceed to the diagnostics in Chapter 11 after checking that jumpers J11-23 are correctly positioned.

The tests which you have just completed exercise all of the connections between the FPGAs and most (but not all) of the components on the BORG.

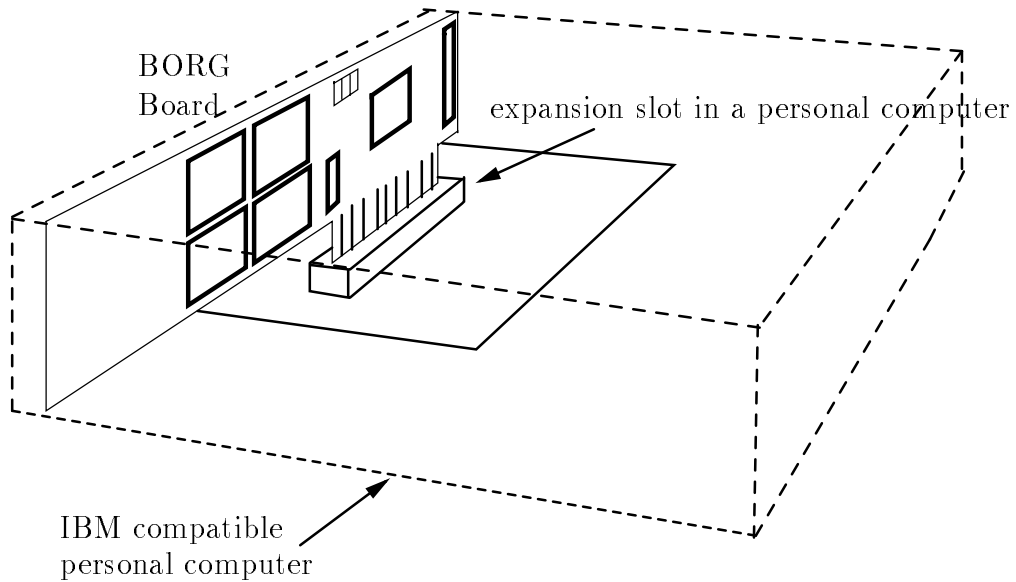


Figure 2.4: Using the BORG board in add-in mode.

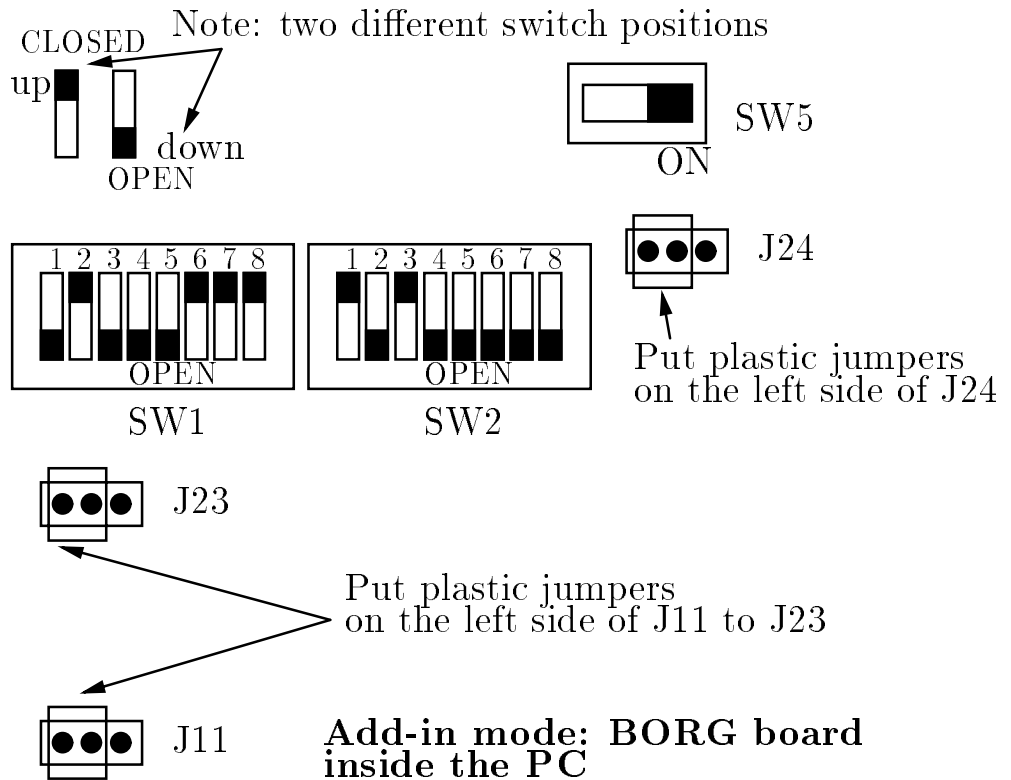


Figure 2.5: Setting for testing BORG board (add-in mode) with port address 0x30X and memory based address 0xd0000h

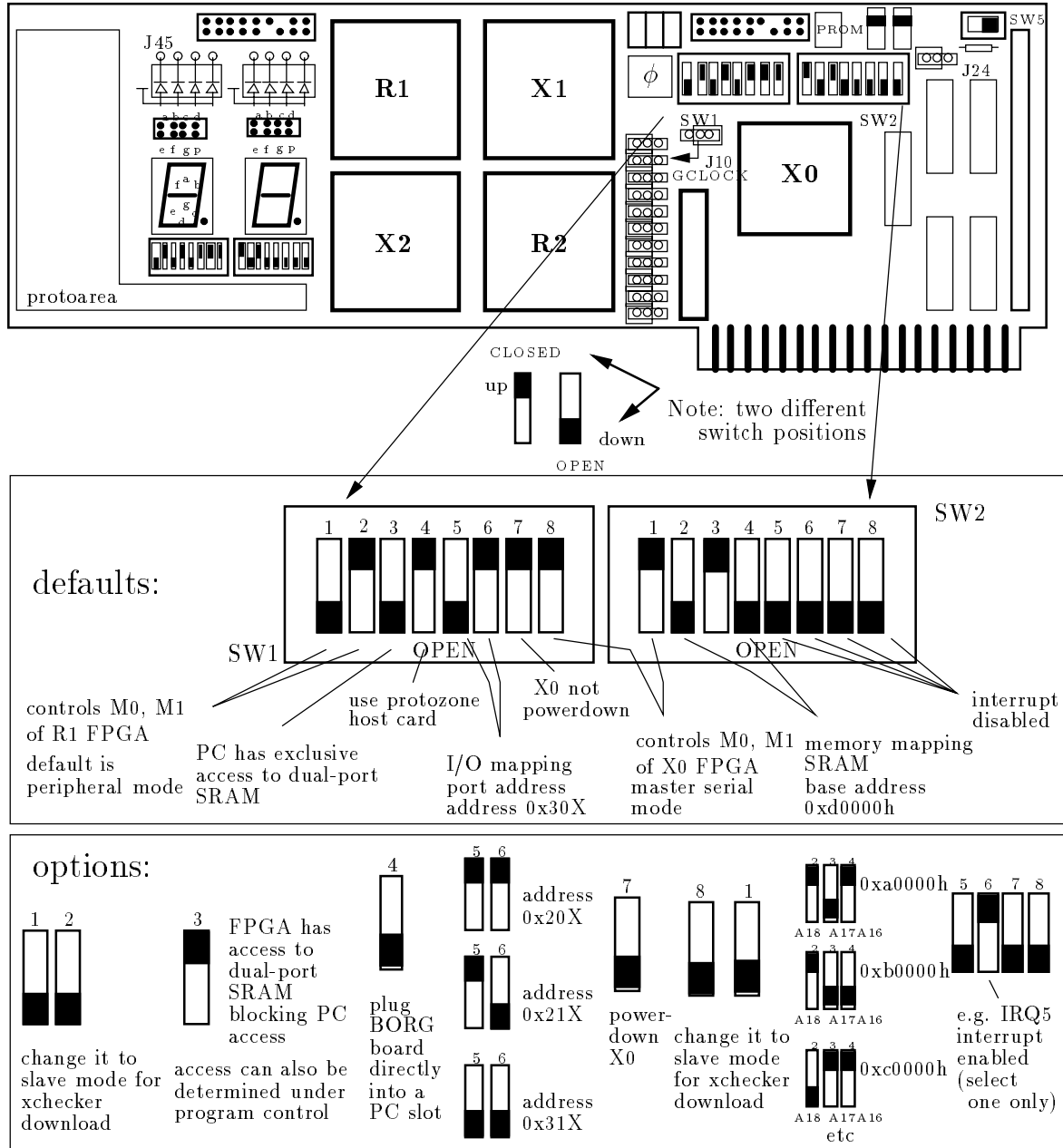


Figure 2.6: Defaults and Options of the BORG board.

3. Simple Demonstrations

3.1 A Tetris Machine

In this demonstration we shall download a Tetris machine which is a multiple-chip design. It uses the X1 and X2 FPGAs for logic, R1 and R2 FPGA for routing. This Tetris machine is realized with approximately 150 XC4000 CLBs. A program running on the PC displays the Tetris bucket (Fig. 3.1) and communicates with the Tetris machine running in the ASICs using port I/O. The program randomly draws a tile type and presents it to the Tetris machine. The Tetris machine determines how to rotate and move the tile before the tile drops. The Tetris machine uses the XC4000 “on-chip” RAM for keeping track of the Tetris bucket; it is not using the dual-ported SRAM on the BORG board.

For this demonstration, the BORG board can be either installed in the **add-in mode** or **host-mode** with the default settings as given in Fig. 2.4 or Fig. 2.2, respectively. *If the required settings are not as prescribed for your installation mode, please set them as described in Section 2.3 now.* This demonstration requires I/O port addresses `0x30X` (`0x300` to `0x30F`) to be vacant. These are the default I/O port addresses. See Fig. 2.6 for options to change the I/O port mapping.

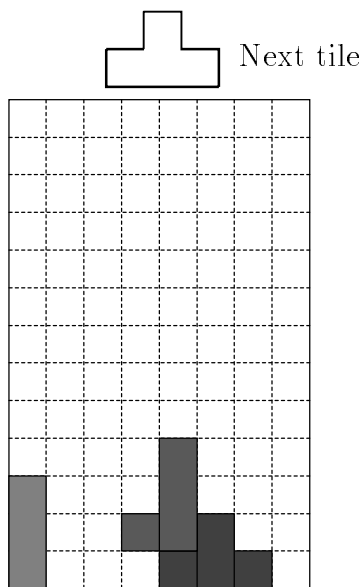


Figure 3.1: A Tetris bucket and some of its tiles.

Important: This Tetris demo requires that your PC is preloaded with the `ansi.sys` device driver. If this is not the case, the problem can be corrected by including this line in your `config.sys` file, and rebooting your machine.

```
DEVICE=C:\DOS\ANSI.SYS
```

1. Download the `mcs` file of the Tetris machine, by typing

```
C:> bd mcs\martine.mcs
```

Observe that the DONE indicator LED3 should turn off and then ON again, indicating all ASICs are programmed.

2. Exercise the Tetris machine by typing

```
C:> tetris
```

Terminate the program with `^C` and clean up the screen by using the supplied program `clear`. If your screen is all messed up now, this means that your PC was not running the `ansi.sys` device driver.

3.2 A Maze Solver Machine

The mazer machine is a multiple-chip design which solves a maze. The machine uses the R1 and R2 FPGAs for logic, and X1 and X2 FPGAs for routing (not a mistake). This maze machine is realized with approximately 120 XC4000 CLBs. It uses 2K bytes of the on-board (dual-ported SRAM) SRAM for keeping track of the maze.

For this demonstration, the BORG board can be installed either in **add-in mode** or **host-mode** with the required settings as given in Fig. 3.2 or Fig. 3.3, respectively. *If the required settings are not as prescribed in these figures, please set them this way now.* Note that jumpers J11-J23 are set to the right which is **not** the default setting that was given in Section 2.3. This demonstration requires I/O port addresses `0x30X` (`0x300` to `0x30F`) to be vacant. These are the default I/O port addresses. See Fig. 2.6 for options to change the I/O port mapping.

Important: This mazer demonstration requires that your PC is preloaded with the `ansi.sys` device driver. If this is not the case, the problem can be corrected by including the following line in your `config.sys` file, and rebooting your machine.

```
DEVICE=C:\DOS\ANSI.SYS
```

Important: You need to block the PC's access to the dual-ported SRAM by using the program

```
C:> arbit xilinx
```

This gives the R2 FPGA exclusive access to the dual-ported SRAM.

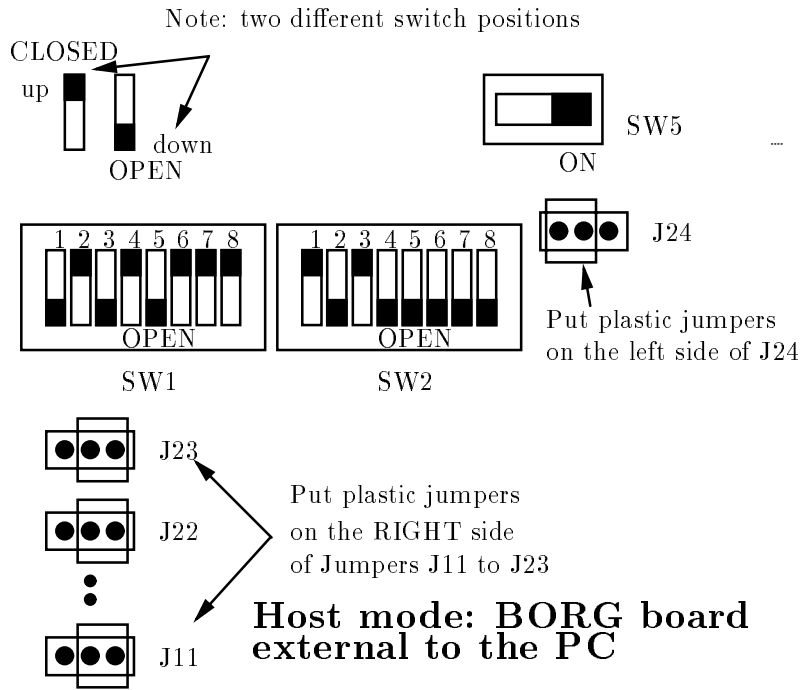


Figure 3.2: Setting for running Maze machine with the BORG board in host mode, with port address 0x30X and memory based address 0xd0000h

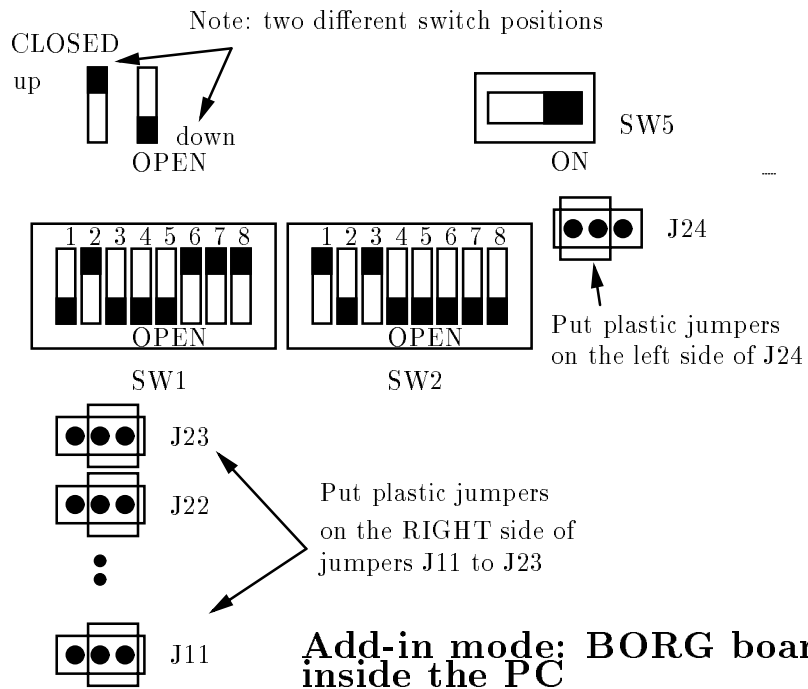


Figure 3.3: Setting for running Maze machine with the BORG board in add-in mode, using port address 0x30X and memory based address 0xd0000h

Please follow the given steps:

1. Download the `mcs` file of the maze machine, by typing

```
C:> bd mcs\maze.mcs
```

Observe that the DONE indicator LED3 should turn off and then ON again, indicating all FPGAs are programmed.

2. You can exercise the maze machine by typing

```
C:> maze
```

This program displays a randomly generated maze with one exit (character `%`). Starting from a randomly chosen location (the origin), the mazer (`@`) runs the maze in two passes. In the first pass, the mazer traverses and explores the maze. When the mazer reaches the exit, it is teleported back to the origin. On the second run the mazer tries to reach the exit in record time.

```

-----
      |      |      ^
- | | -----
| | | | | | | |
| | | | | --- | --- ----- |
|%.@| | | | | | |
-----

```

Level 2 maze. Total moves 108

You may terminate the program with `^C` and clean up the screen by using the supplied program `clear`. If your screen is all messed up now, this means that your PC was not running the `ansi.sys` device driver.

4. Principle of Operation

4.1 Status indicators

There are three LEDs on the BORG board which indicate the status of the FPGAs and the board.

POWER This LED (LED1 rightmost LED on the top) indicates that the BORG board has power.

X0 This LED (LED2) indicates that the PC/XT bus controller FPGA **X0** is configured.

DONE The DONE pins of the user FPGAs **R1**, **X1**, **R2**, **X2** are tied together to the DONE LED (LED3) to indicate that the four user FPGAs (ASICS) are configured.

There are also two common-anode seven segment displays and two common-anode four-bar LEDs in the proto-area that can be used to monitor additional signals.

4.2 Stand-alone BORG board

You can use the BORG board in the same way as the Xilinx XC4000 demo board. This is the simplest but not the best way to use the BORG board. In this mode, you can use the four user XC4000 FPGAs. To use the BORG board as a stand-alone board, you must

1. set position PDWDWN of the BORG board to open, this disables (power downs) the X0 controller.
2. connect an **xchecker** cable to jumper J8,
3. set position M0R1 of DIP switch SW1 to open,
4. set position M1R1 of DIP switch SW1 to open, and
5. supply power (+5V) to the board via jumper J5.

Steps 3 and 4 have just put R1 into slave mode. For programming the FPGAs, use the **xchecker** program and cable. The FPGAs are daisy-chained in the following order:

R1 -> X1 -> R2 -> X2

This means the Dout (Data out program pin) of the first FPGA R1 is connected to the Din of the second FPGA X1 and so forth so on. Their DONE pins are tied together. LED3 turns to red if the four FPGAs are successfully programmed.

If you need only one FPGA, you must use the R1 FPGA. You can either extract the rest of the FPGAs with a PLCC extractor tool made by a company called AUGAT, or download the rest of the FPGAs with “empty” bit streams. You can find null bit streams for the individual FPGA types in the distribution package under the directory

`empty`:

`em4002a.bit` `em4003a.bit`

Important: You need to “concatenate” the bit streams of the individual FPGAs for download, by using the Xilinx **makeprom** program.

```
makeprom -o design.mcs -u 0 myr1.bit em4003a em4002a em4003a
```

This example assumes that your design bit stream is in the bit file `myr1.bit`.

4.3 BORG board as a Peripheral Device of the PC/XT

The BORG board is just like any other PC/XT peripheral cards; it interfaces with the PC/XT via port I/O, memory map, interrupt, and DMA. The next few sections will guide you to map the BORG board into the PC vacant and valid I/O address space and memory address space, interrupt and DMA channels.

Also, the BORG board draws its power from the PC's power supply. You don't have to worry, because most PCs have 150 Watt to 250 Watt power supply. The BORG board consumes approximately 5W of power. There is also a 3-Ampere fuse on the BORG board, just in case.

For now, you should study Fig. 2.6 to identify the locations of jumpers, switches and reference designators on the BORG board.

4.4 Put the BORG Board Inside or Outside the PC?

The BORG board has two modes of installation. You can install the BORG board inside or outside a PC; we refer the first option as *add-in mode* and the latter as *host mode*.

Add-in mode The BORG can be plugged into a PC/XT expansion slot, as illustrated in Fig. 2.4. This has the disadvantage that the FPGAs' signals are inaccessible. But you can use a PC/XT signal extension card to accommodate the BORG card. The extender card is recommended since it allows easier access to signals on the BORG board.

Host mode Alternatively, with the *Protozone*¹ host card in an PC/XT expansion slot and a 50-wire flat ribbon cable from the protozone host card plugged into connector J3, the BORG board can be used outside the PC, as illustrated in Fig. 2.2.

4.5 I/O Address Mapping

Minimally, the BORG board must be mapped into some vacant locations in the PC/XT's I/O address space. The BORG board's controller X0 has four predefined I/O ports for maintaining the vital communication with the PC to support downloading bitstreams. We call these I/O ports X0ports.

You can build *additional* I/O ports to support your design in the R1 FPGA. In a "typical" PC configuration, you will find that the I/O addresses from 0x300 to 0x30F are vacant. Examples of occupied I/O address locations are `0x378` and `0x2F8` which are the printer port LPT1 and serial port COM2, respectively. There are

¹A. El Gamal, "Protozone: The PC-Based ASIC Design Frame, User's Guide," Tech. Rep. SISL90-???, Stanford Information Systems Laboratory, Stanford University, Aug. 1990.

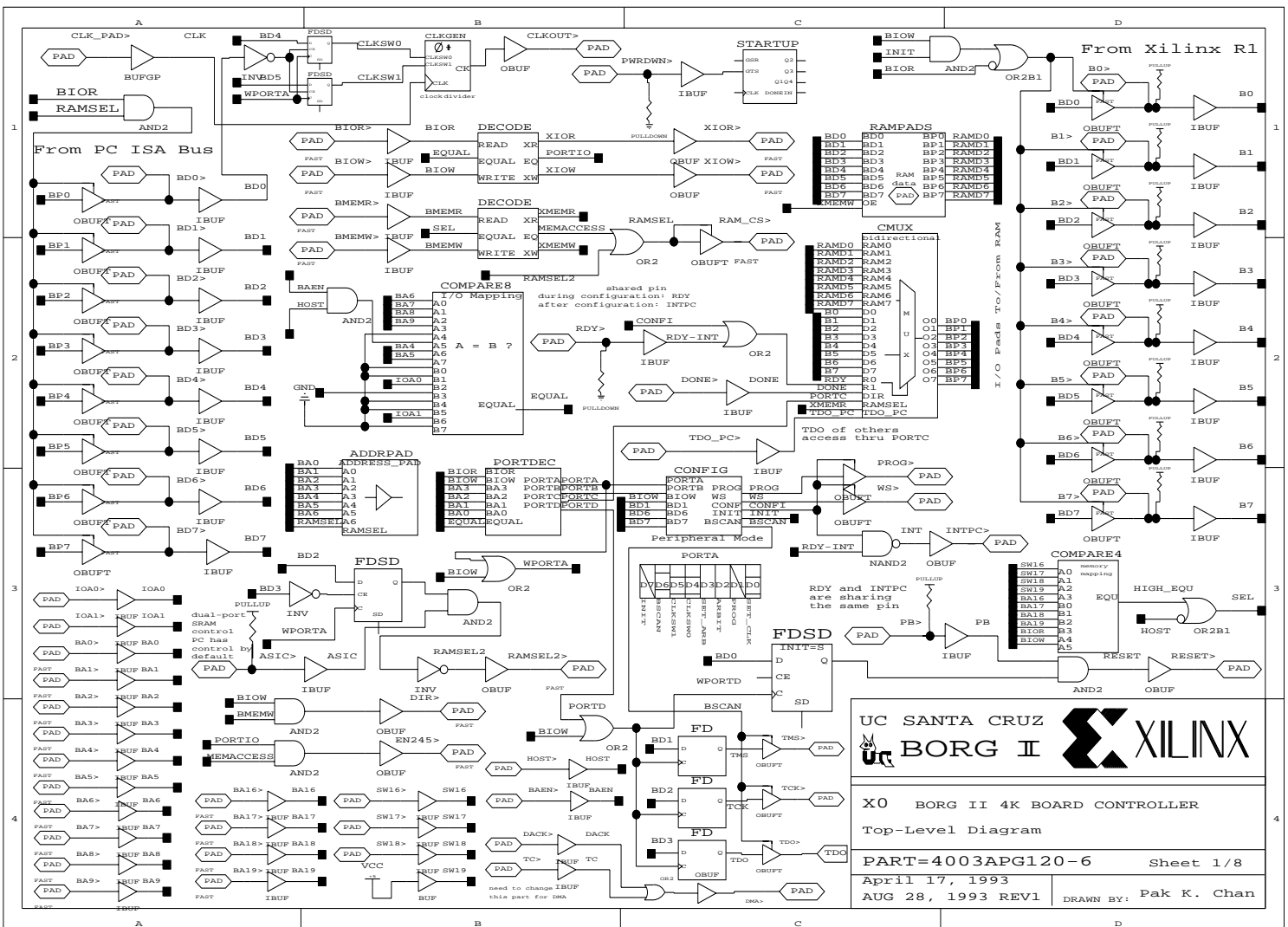


Figure 4.1: X0 Top-level schematic drawing of the BORG board.

provisions to modify the I/O mapping to suit your needs. Referring to Fig. 4.1 for the top-level schematic of the X0 controller. You will find that the module COMPARE8 decodes address A4-A9 and the settings of the DIP switch SW2 positions IOA0 and IOA1 to determine the I/O mapping. The XT bus active-low signal `baen` is used only in add-in mode (`host=1`), as illustrated in Fig. 4.2.

```

compare ---> 1 IOA0 0 0 0 IOA1 0 0          X X X X
with         | | | | | | | |
I/O         ---> BA9 BA8 BA7 BA6 BA5 BA4 0 (host & baen) BA3 BA2 BA1 BA0
addresses

```

Figure 4.2: I/O Address Decoding in X0.

So the the comparator's output is asserted when address lines BA8 and BA4 match the setting of positions IOA0 and IOA1 of DIP switch SW2. The least significant four address lines BA0-BA3 are decoded in X0, but only the lower 4 I/O locations are taken by X0 controller. The I/O mapping is listed in Table 4.1. Also, the address lines BA0-BA3 are provided as inputs in R1, and *must* be fully or partially decoded in R1 to avoid conflict with the ports in the X0 controller. You should consult Section 9.1 for further information on building your own I/O ports in the R1 FPGA.

IOA0	IOA1	addresses
0	0	0x20X
0	1	0x21X
1	0	0x30X
1	1	0x31X

Table 4.1: I/O mappings of BORG board (note: IOA=0 means switch is closed, IOA=1 means switch is open, and X is a don't-care).

Important: Referring to Fig. 4.1, the BORG board's controller X0 has four pre-defined I/O ports defined in the module PORTDEC for maintaining the vital communication with the PC to facilitate downloading bitstreams. We call them **X0ports**. So depending on the settings of positions IOA0 and IOA1 of DIP switch SW1, X0ports' port addresses in X0 are given in Table 4.2. The functions of the X0ports are given in Table 4.3.

I/O Ports	IOA0	IOA1	addresses
PORTA,B,C,D	0	0	0x200 to 0x203
PORTA,B,C,D	0	1	0x210 to 0x213
PORTA,B,C,D	1	0	0x300 to 0x303
PORTA,B,C,D	1	1	0x310 to 0x313

Table 4.2: Occupied I/O addresses in X0.

X0port	Function
PORTA	set control functions of other ports and SRAM arbitration
PORTB	download bit streams
PORTC	read port (contains a zero)
PORTD	boundary scan and global reset

Table 4.3: Functions of I/O ports (X0ports) in X0.

As shown in Table 4.4, the I/O signals - IOR, IOW, A0, A1, A2, A3, D0-D7 are available to the **R1** FPGA. Port I/O is the simplest way for the BORG board to communicate with the PC. The **C** library functions

```
inportb(port)
outportb(port, byte)
```

can be used for reading and writing the ports, respectively.

The I/O read and write signals: XIOR and XIOW have already been decoded by **X0** to ensure that the I/O signals IOR and IOW are directed towards the BORG Board. (The decoding is controlled by switch positions IOA0 and IOA1 of the DIP switch SW1.) Four of the 16 available ports are used by **X0** as described. This leaves 12 port addresses available for the R1 FPGA to communicate with the PC.

Signal	Pin # of R1 FPGA
INTERRUPT	70
A0	83
A1	81
A2	82
A3	80
XIOR	51
XIOW	50
D0	71
D1	69
D2	67
D3	65
D4	61
D5	59
D6	58
D7	56
Global Clock	13
Global RESET	10

Table 4.4: System signals available to **R1**.

4.6 Memory Mapping

The dual-ported SRAM (U2) can be accessed by your PC/XT if the SRAM is properly mapped into the PC/XT's vacant memory address space.

In the *host mode*, the mapping is determined by the setting dip switch SW2 of your protozone adapter card, please consult your Protozone adapter card user's guide.

In the *add-in mode*, you can control the mapping with switch positions A18, A17, and A16 of DIP switch SW2 (on the BORG board) which set the equality comparison with the PC address lines A19, A18, A17, A16. In either case, for dual-port access, the 8K dual-port SRAM 6116 (U2) must be mapped into a block of locations in your PC upper memory area (UMA). UMAs are higher than 640K and less than 1024K in the memory address space.

Finding vacant locations is tricky. Typically, this can be either locations with base memory address 0xd0000 or 0xe0000. Table 4.5 shows a typical high memory map in DOS.

A19,A18,A17,A16	Typical usage
F	System BIOS (ROM)
E	probably not used ?
D	probably not used ?
C	Network Adapter, Video ROM, HD controller
B	Video RAM
A	Video RAM

Table 4.5: Typical UMA address map in a PC computer.

If your PC is using DOS 5.0 or higher, there may also be a problem if the memory manager is using some of the upper memory area to accommodate your device drivers (e.g., mouse, ansi.sys etc). You can avoid memory conflicts by commenting “DOS=HIGH” out from your `config.sys`, and also avoiding the use of “loadhi” commands. At any rate, do the following in DOS 5.0 (or higher) to display a memory map and find an area that is vacant to accommodate the 8K dual-port RAM.

```
C:> mem /p
```

or

```
C:> mem /c
```

You should consult Section 4.15 for further information on arbitrating the dual-port SRAM.

4.7 Hardware Interrupt Channel

Pin 70 of the R1 FPGA is connected to hardware interrupt channel of your PC/XT. The IBM PC AT and PC/XT computers have different channel assignments, so be careful. Table 4.6 shows a typical hardware interrupt channel in a PC AT computer.

You can enable an interrupt channel by the DIP switch SW2 on the BORG board. If you are in add-in mode, you can select either IRQ3, or IRQ5, or IRQ7, or IRQ9 by the DIP switch SW2 to enable interrupt; or none to disable an interrupt. Make sure that the channel you chose is not in conflict with other devices in your system, for example, a serial mouse uses IRQ4; and IRQ5 may be used by a printer in LPT2.

Hardware Interrupt	Vector	Description
IRQ0	0x08	System Timer
IRQ1	0x09	Keyboard Interrupt
IRQ2	0x0A	unused connect to another 8259A chip
IRQ3	0x0B	serial port COM2
IRQ4	0x0C	serial port COM1
IRQ5	0x0D	parallel port LPT2 in PC/AT (hard disk in PC/XT !)
IRQ6	0x0E	floppy disk controller
IRQ7	0x0F	parallel port LPT1
IRQ8	0x70	real time clock
IRQ9	0x71	(0x0A) rerouted to IRQ 2
IRQ10-IRQ15		PC/AT only

Table 4.6: Typical hardware interrupt channel in a PC AT computer.

If you are in host mode, you need to select the interrupt channel in the protozone adapter card. You can use a lab given later in Section 9.2 as a guide to write interrupt service routine, and the use the hardware interrupt feature.

4.8 DMA Channel

You need to change the default design of the controller X0 to practice DMA transfer using the BORG board, and you must use the protozone adapter card in order to use DMA. The protozone adapter card's DMA channel is designed for an PC/AT computer. Also, you need to select the proper DMA channel in the protozone adapter card.

Three DMA related signals: terminal count expire (TC), DMA request (DMA), DMA acknowledge (DACK) are available in X0 for you to build your own DMA controller.

You can follow a lab given later in Chapter 9 as a guide to use the DMA feature.

4.9 Configuring the controller X0 FPGA

Master serial mode: By default, the controller X0 (U1) is programmed by a small serial PROM xc1765D (in U3) using the master serial mode. To set X0 to this mode:

1. shunt J24 on the left side with a plastic jumper,
2. set position M0X0 of dip switch SW1 to closed, and
3. set position M1X0 of dip switch SW2 to closed.

Slave mode: Alternatively, customize your own controller by programming X0 in the slave mode using the Xilinx **xchecker** cable via J9. To set X0 to this mode:

1. shunt jumper J24 on the right side with a plastic jumper,
2. set position M0X0 of dip switch SW1 to open, and
3. set position M1X0 of dip switch SW2 to open.

In either case, the light emitting diode LED2 turns to green when X0 is successfully programmed.

4.10 Programming the R1, X1, R2 and X2 FPGAs

For programming purpose, the FPGAs R1, X1, R2, and X2 are daisy-chained, which means the Dout of the first FPGA R1 is connected to the Din of the second FPGA X1 and so forth so on. Their DONE pins are tied together.

The R1 FPGA can be programmed either in peripheral mode or slave mode; the other three X1, X2, R2 FPGAs are always configured in the slave serial mode. Since, the mode pins M0, M1 and M2 pins of **X1**, **R2**, and **X2** are tied to VCC, this puts them into daisy chained slave programming mode with the **R1** FPGA as the master. Remember:

R1 -> X1 -> R2 -> X2

This means the Dout of the first FPGA R1 is connected to the Din of the second FPGA X1 and so forth so on. Their DONE pins are tied together.

Important: You need to “concatenate” the bit streams of the individual FPGAs for download. You do so by using the Xilinx `makeprom` program, see the next two paragraphs.

If you need only one FPGA, you must use the R1 FPGA. You can either extract the rest of the FPGAs with a PLCC extractor tool made by a company called AUGAT, or fill the rest of the FPGAs with “empty” bit streams. You can find null bit streams for each of the individual FPGA types in the distribution package under the directory `empty`:

`em4002a.bit em4003a.bit`

Use them to generate a single `mcs` file of your design along with the bit stream of your design in the R1 FPGA (say: `myr1.bit`) using the Xilinx `makeprom` utility:

```
makebits myr1
makeprom -o design.mcs -u 0 myr1.bit em4003a em4002a em4003a
```

To use the R1 FPGA in the peripheral mode, you set both positions M0R1 and M1R1 of DIP switch SW1 to open and closed, respectively. The bit streams to configure the FPGAs are downloaded via the 8-bit PC databus sent by the supplied download program `bd`. LED3 (DONE) turns to red if the FPGAs are successfully programmed.

```
c:> bd design.mcs
```

To use the R1 FPGA in the standalone mode, refer to Section 4.2.

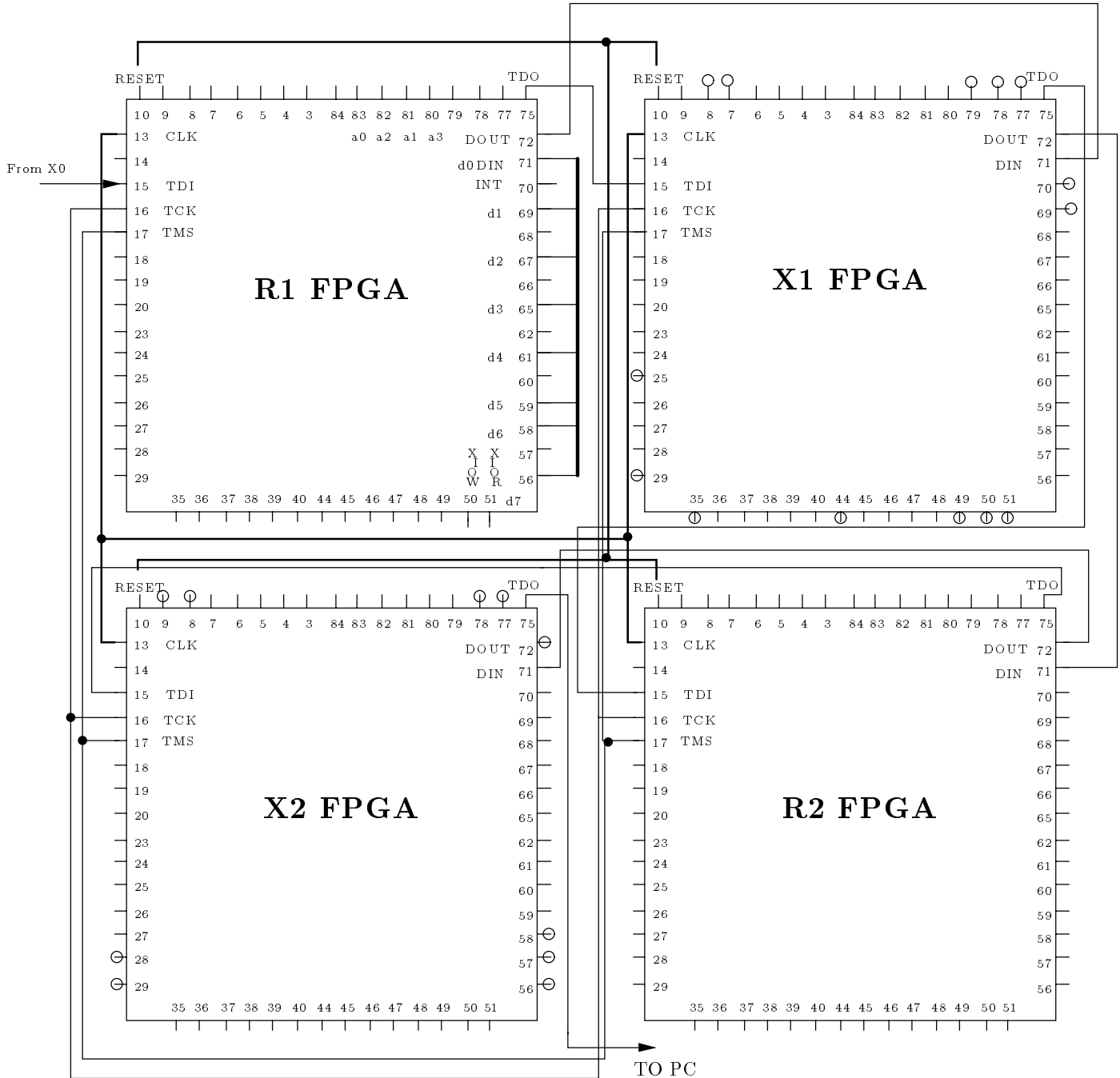


Figure 4.3: User FPGAs and Global Signals.

4.11 Global Reset

You can reset the R1, X1, R2 and X2 FPGAs manually by depressing the push button SW4. This global reset can be also initiated under (port I/O) program control. It is connected to Pin 10 of all user FPGAs, as illustrated in Fig. 4.3. As mentioned earlier in Section 4.5, the BORG board's controller X0 has four predefined I/O ports. Bit 0 of PORTD is used for global reset.

4.12 Readback

Only the R1 and X0 FPGAs are available for readback using the `xchecker` program and cable. The mode pins of the other FPGAs are tied to VCC, so readback is not possible.

4.13 JTAG Boundary Scan

You can only use R1, X1, R2 and X2 FPGAs for boundary scan. X0 is the controller of the boundary scan chain. As mentioned, the BORG board's controller X0 has four predefined I/O ports. The three JTAG boundary scan pins: TMS, TCK, TDI of the R1, X1, R2 and X2 FPGAs are connected to bit 1 to bit 3 of PORTD of X0 to boundary scan the user FPGAs under port I/O program control. X0 reads the TDO from the user FPGAs via the TDO_PC pin.

Warning: Since bit 0 of PORTD is reserved for global reset (active low), don't write a zero to bit 0 of this port unless you really mean to.

4.14 System Clock and Single Step

You may find the on-board (default 8 MHz) TTL-crystal clock generator useful. Place the plastic jumper on the right side of J10 to use the crystal clock. It is divided internally by a counter in the X0 controller (if X0 is not powered down). The clock divisor can be selected by the `clock` program. For example, you use

```
c:> clock turbo
```

for a divided by 1 clock (default 8 MHz), and

```
c:> clock slow
```

for a divided by 8 clock.

The `clock` utility loads 2 bits to select the desired divisor that resides in bits 4 and 5 of PORTA of X0port inside X0 (see Section 4.5).

You can toggle the system manually by placing the plastic jumper on the left side of J10 and use the push button for single stepping. The global clock is broadcast to Pin 13 of all user FPGAs, as illustrated in Fig. 4.3.

A listing of the `clock` utility is given on the next page.

```

#include<stdio.h>
#include<dos.h>
#include<stdlib.h>

main(int argc, char *argv[]) /* clock speed selection */
{
    unsigned int PortA;
    char * portenv;
    setcbrk(1);

    printf("\nCLOCK Ver. #1.0\n");
    printf("UC SANTA CRUZ, COMPUTER ENGINEERING, August 1993\n");
    printf("(c) Copyright 1993 UC Regents. All rights reserved\n\n");

    if(argc==1) {
        printf("  Function: Set BORG Protoboard global CLOCK speed\n");
        printf("  Usage:   clock [ slow | quick | fast | turbo ]\n\n");
        printf("                /8    /4    /2    /1\n\n");
        exit(1);
    }
    portenv = getenv("BORG");

    /* Control Port in X0 */
    if(!strcmp(portenv,"0x300"))
        PortA = 0x300;
    else if(!strcmp(portenv,"0x200"))
        PortA = 0x200;
    else if(!strcmp(portenv,"0x210"))
        PortA = 0x210;
    else if(!strcmp(portenv,"0x310"))
        PortA = 0x310;
    else {
        printf(" Wrong PORT address\n");
        printf(" Please specify PORT address\n e.g. set BORG=0x300%s\n");
        exit(1);
    }

    printf(" >BORG PORT address is %s\n",portenv);

    if(argc==2){
        switch(argv[1][0])
        {
            case 's': outportb(PortA, 0xce);
                    printf(" >Global clock is now slow \n");
                    break;
            case 'q': outportb(PortA, 0xde);
                    printf(" >Global clock is now quick \n");
                    break;
            case 'f': outportb(PortA, 0xee);
                    printf(" >Global clock is now fast \n");
                    break;
            case 't': outportb(PortA, 0xfe);
                    printf(" >Global clock is now turbo \n");
                    break;
            default: printf(" Error: flag not recognize '%s'\n", argv[1]);
                    printf(" Usage:   clock [ slow | quick | fast | turbo ]\n\n");
        }
        exit(0);
    }
}

```


pin of SRAM	pin of R2 PC84 package FPGA
D0	R2.46
D1	R2.45
D2	R2.44
D3	R2.40
D4	R2.39
D5	R2.38
D6	R2.37
D7	R2.36
A0	R2.47
A1	R2.48
A2	R2.49
A3	R2.50
A4	R2.51
A5	R2.56
A6	R2.57
A7	R2.58
A8	R2.61
A9	R2.62
A10	R2.67
A11	R2.66
A12	R2.59
CS	R2.30
OE	R2.65
WE	R2.60

Table 4.7: Memory signals from R2 to dual port SRAM; the connections to memory addresses A0-A12 depend on jumpers J11-J23. To use the pin assignment tool `assign` you need to use the appropriate *wiring file* and flag (see `assign` command option in Chapter 7) to reflect the status of the jumpers.

4.15 On-board SRAM and arbitration

4.15.1 8K×8 SRAM

If your design requires only a wide but shallow amount of memory, it is much better to use the XC4000 on-chip RAM. If you need deep but narrow memory, the on-board 8K×8 SRAM can be useful.

As indicated on the BORG board, pin 2 of jumpers J11-J23 are the (A0-A12) address lines to the SRAM coming out from the R2 FPGA. You can move the plastic jumpers of J11-J23 to the right side to use all the on-board 8K×8 SRAM. In this case, you have less connections available between X1 and X2 FPGAs, as illustrated in Fig. 4.4.

In Fig. 4.4, you will find that the SRAM is connected to the R2 FPGA, the pin assignment of R2 FPGA is given in Table 4.7. All the memory access signals (8-bit data lines, 13-bit address lines, R/W, OE-, and CS-) of the user FPGAs have to go through **R2** before reaching the SRAM (see also Figure 1.2). In particular, pin 30 of **R2** is the chip select (CS-). This signal is tri-stated and is in wire-AND configuration with the **RAMSEL** signal of **X0**. You need to use the special MD1 symbol in your schematic drawing to use this pin. This active-low signal is normally pulled high by a 4.7K resistor. Figure 4.5 illustrates the memory *write* timing as the SRAM is under tested.

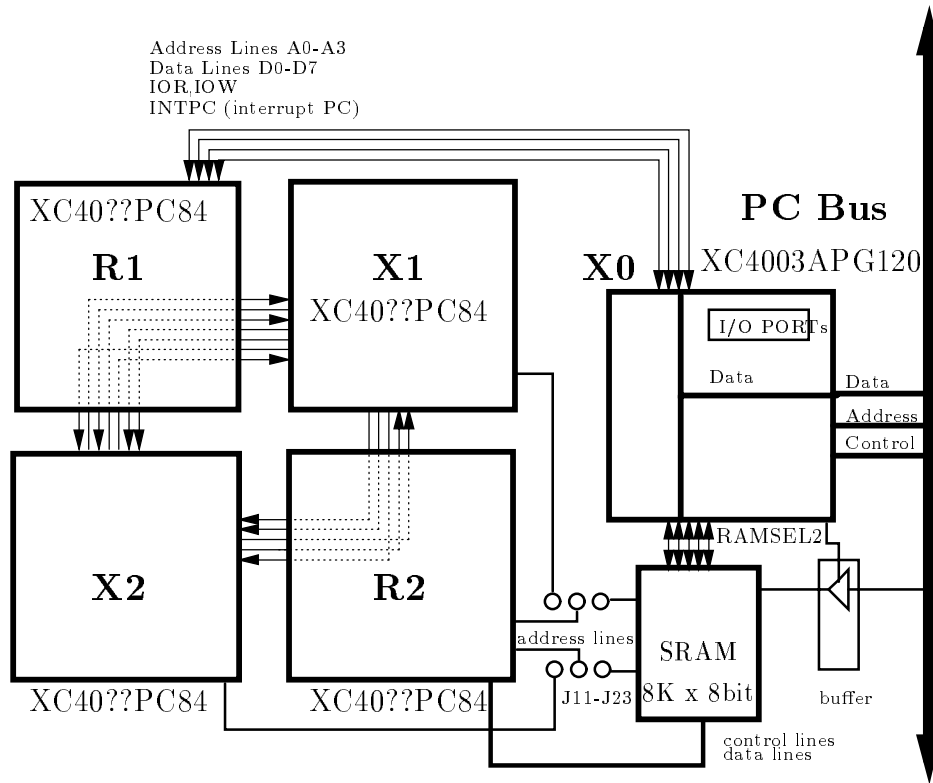


Figure 4.4: SRAM and the rest of the FPGAs.

On the other hand, if you need more connections between the user FPGAs X1 and X2, you may move the plastic jumpers of J11-J23 to the left side (this is the default configuration, see Fig. 2.6); and the on-board 8K×8 SRAM is *inaccessible*.

4.15.2 Dual-port SRAM arbitration

The 8K dual-port SRAM can be accessed either by the PC or the R2 FPGA. The X0 controller provides some simple arbitration logic. There are three mechanisms for arbitration.

First, you can control the default dual-port SRAM access by setting position 3 of DIP switch SW1 (DURAM). If this switch position is open, the PC has exclusive access to the SRAM. By the same token, you can make the SRAM inaccessible to the PC by closing this switch.

Second, you can arbitrate the dual-port SRAM access under program control, overwriting the default set by DIP switch SW1 (DURAM). Bit 2 and bit 3 of PortA of X0 port arbitrates the memory access, as illustrated by the `arbit` utility on the next page.

Third, jumper J1 is connected to the ASIC pin of the X0 controller. This *active-low* signal can be used to block the PC access to the dual-port SRAM by tristating the data and address buffers surrounding the dual-port SRAM on the PC side. The

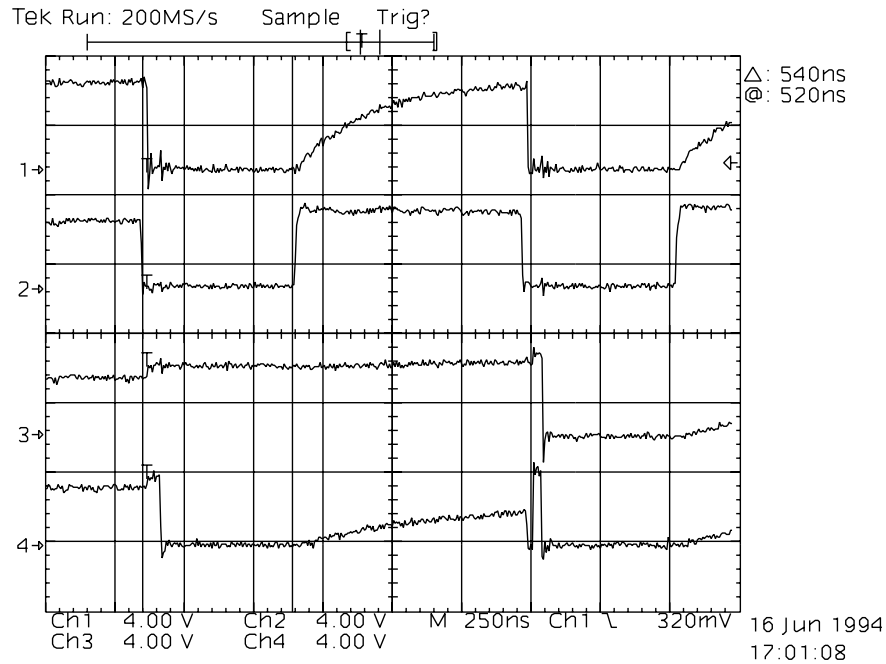


Figure 4.5: SRAM write timing of two consecutive write cycles. Channel 1 is the chip select **CS** signal. Channel 2 is the write **WE** signal. Channel 3 is address line **A0**, a ‘1’ on the first write cycle, and then a ‘0’ on the second one. Channel 4 is the data line **D0**, zeros for both cycles.

static RAM 6264 is of 70ns speed grade. We have tested the BORG board using 150ns RAM without problems. We use the 70ns speed grade because it is available and cheap.

4.16 Limits on the Number of Connections Between the FPGAs

Some of the I/O pads on **R1** and **R2** are used to support the dual-ported SRAM and port I/O communications with the PC. Thus, although the number of user pads available on a 84-pin PLCC package is 54, the **maximum** number of connections between **X1** and **X2** which can be realized with **R1** and **R2** is **38**, with the plastic jumpers of J11-J23 on the left side, and using TDI and TDO pins.

With the plastic jumpers of J11-J23 on the right side, the **maximum** number of connections between **X1** and **X2** which can be realized with **R1** and **R2** is **28**. Figure 4.6 shows the pin distribution between the FPGAs. There are some unconnected pin in the **X1** and **X2** FPGAs are indicated with a small circle on their pins in Fig. 4.3. They can be used for probing/debugging purposes.

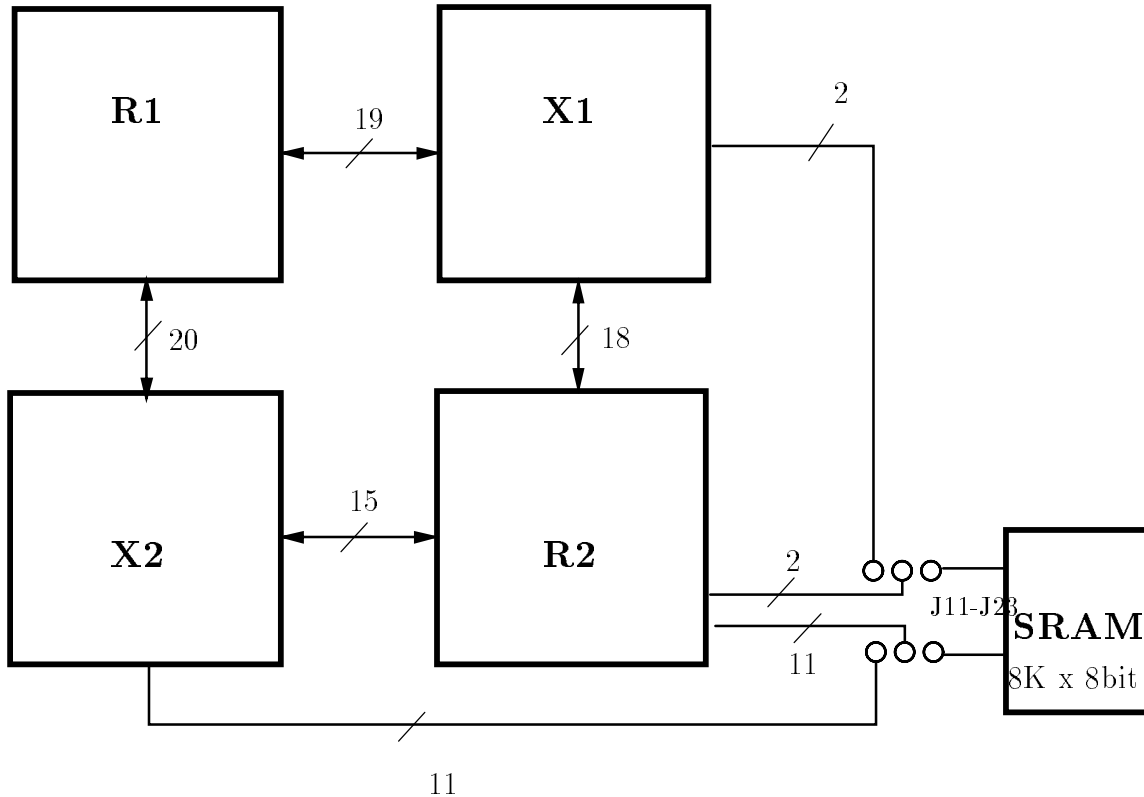


Figure 4.6: Pin Distribution between the FPGAs.

```

/* program arbit
   dual-port SRAM arbitration */
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>

main(int argc, char *argv[])
{
    unsigned int PortA;
    char * portenv;
    setcbrk(1);

    printf("\nRAM ARBITER Ver. #1.0\n");
    printf("UC SANTA CRUZ, COMPUTER ENGINEERING, August 1993\n");
    printf("(c) Copyright 1993 UC Regents. All rights reserved\n\n");

    if(argc==1) {
        printf("  Function: Arbitrate BORG II Protoboard's RAM\n");
        printf("  Usage:   arbit [ xilinx | pc ]\n\n");
        exit(1);
    }
    portenv = getenv("BORG");

    /* Control Port in X0 */
    if(!strcmp(portenv,"0x300"))
        PortA = 0x300;
    else if(!strcmp(portenv,"0x200"))
        PortA = 0x200;

```

```

else if(!strcmp(portenv,"0x210"))
    PortA = 0x210;
else if(!strcmp(portenv,"0x310"))
    PortA = 0x310;
else {
    printf(" Wrong PORT address\n");
    printf(" Please specify PORT address\n e.g. set BORG=0x300%s\n");
    exit(1);
}
printf(">BORG PORT address is %s\n",portenv);

if(argc==1) {
printf(" Function: Arbitrate BORG II Protoboard's RAM\n");
printf(" Usage: arbit [ xilinx | pc ]\n\n");
exit(1);
}
if(argc==2){
switch(argv[1][0])
{
case 'x': outportb(PortA, 0xf3);
printf(">BORG Xilinx's has exclusive access to the RAM\n");
break;
case 'p': outportb(PortA, 0xf7);
printf(">PC has exclusive access to the RAM\n");
break;
default:
printf(" Error: unknow flag '%s'\n", argv[1]);
printf(" Usage: arbit [ xilinx | pc ]\n\n");
exit(1);
}
printf("\n Warning: RAM access can be hardwired by a\n");
printf(" : switch position 3 (DURAM) of DIP SW#1\n");
printf(" : Open: PC access closed: FPGA access\n");
exit(0);
}
}

```

5. Software

The software programs and subdirectories in the distribution package Ver 1.0 are described in Table 5.1.

file name	description
arbit	arbitrates dual-port SRAM access
assign	pin assignment program to connect multiple FPGAs (need a 386)
bscan	boundary scan program (unsupported!)
bd	downloads an mcs file to the BORG board (runs on XT compatible)
clear	clear a messy screen
clock	program to change the system clock rate
default	print out the default DIP switch settings
inspect	list content of dual-port SRAM
isr.com	interrupt service routine for interrupt lab
intpc	interrupt generator for the interrupt lab
maze	maze runner driver project example
mtest	checks (by writing after reading) the 8k dual-port SRAM 6264 on the BORG Board
portest	lab example to show building I/O ports in R1
setassign.bat	example bat file to set an environment variable for the program assign
scan	scan test to check all the I/O on BORG board
tetris	driver for the Tetris project (runs on XT compatible)
src\	subdirectory containing the source code
designs\	subdirectory with the LCA files for the project examples
mcs\	subdirectory with the mcs files for the design examples
empty\	subdirectory with null bit-streams for XC4003a and XC4002a 84PLCC packages
X0\	subdirectory viewdraw schematic of the X0 controller
assign\	subdirectory supporting files

Table 5.1: Contents of Software Distribution.

5.1 Memory related programs `mtest` and `inspect`

The memory test `mtest` program checks whether the dual-port SRAM is accessible from the PC. The `inspect` program displays the entire contents of the 8K dual-port SRAM.

Before running these programs, you need to disable any access to the dual-port SRAM from the user FPGAs, by closing position 3 DURAM of SW1. You need to download a “null” bit stream into **R2**. You can use the supplied bit stream `portest.mcs` or `scan.mcs` that are in this distribution. Both of these `mcs` files have the necessary bit stream to tristate the I/O pins of **R2**). You should make sure that the PC has exclusive access to the memory, do

```
c:> arbit pc
```

before running either programs.

5.2 Board Wiring test program Scan

The design file `scan.mcs` contains bit streams that chain up most of the I/O pads of the user FPGAs to be a shift register. The program `scan` shifts a zero into the chain and checks whether the zero successfully arrives after certain number of clock cycles.

5.3 Pin assignment program assign

`Assign` is a C++ program which assigns pads on the routing FPGAs to connect the two user FPGAs. You must run `assign` on 32-bit 386/486 machines. Both its source code and executables are included. `Assign` has been compiled with the g++ (DJ) public domain compiler. You should read the section on the options available with `assign` in Chapter 7.

5.3.1 Projects, Demos and their MCS files

The directory `designs\` contains the projects and their LCA files. Their mcs files are in the `mcs\` directory.

tetris4 - Martine Schlag's Tetris project in Aug 1991, the original design used one XC3020 and one XC3042. I have converted the XC3000 design to XC4000 for the purpose of this distribution.

x1tet4f.lca - the controller of the Tetris machine design

x2tet4f.lca - the datapath of the Tetris machine design

R1tet4f.lca - the 1st routing FPGA design

R2tet4f.lca - the 2nd routing FPGA design

tetris4.mcs the bit stream of the complete design

amazer - Jason Y. Zien and David Van Brink's maze runner project in Fall 1992 (CMPE 225 UC Santa Cruz). Their maze runner machine used the XC3000 Borg board for development of the project in Fall 1992. I have converted the XC3000 design to XC4000 for the purpose of this distribution.

R1newg.lca - the 1st FPGA design

R2newg.lca - the 2nd FPGA design

amazerg.lca - the 3rd FPGA design

amazer4.mcs - the bit stream of the complete design

randmaze - David Van Brink's "random" maze runner project in Fall 1992 (CMPE 225 UC Santa Cruz). I have converted the XC3000 design to XC4000 for the purpose of this distribution.

randmaze.lca - the single FPGA design.

This Maze Solver was designed in **XACT** and uses 15 CLBs to make random, but legal, moves through the maze. For each move, it takes into account the mouse's previous direction, and the state of the walls around it, and a random bit, to decide what move to make next. Essentially, it has 50% chance of following a right-hand rule, and 50% chance of following a left hand rule.

The system is clocked by the falling edge of the PC's **YourMove** signal line. The random element comes from the system clock on the Borg board, toggling a flip-flop. Since the BORG's clock is independent from the PC's clock, this seems to work well enough.

portest - testing parallel I/O ports configured in **R1** FPGA

sch schematic drawing of the design in viewdraw

portest.lca - 4 I/O ports in **R1**

portest.mcs the bit stream of the design

intpc - hardware interrupt demo using the **R1** FPGA

sch schematic drawing of the demo in viewdraw

intpc.lca - one I/O port in **R1** by generating hardware interrupt

intpc.mcs - the bit stream of the design

intpc.exe - a driver to trigger the generation of an interrupt

isr.com - a interrupt service routine for the demo

asylab - synchronization failure lab demo using the **R1** FPGA

sch schematic drawing of the demo in viewdraw

asylab.mcs - the bit stream of the design

asylab.exe - the driver to demonstrate synchronization failure

music - frequency synthesizer demo using the **R1** FPGA, you need a digital-to-analog converter and a small transistor amplifier to "listen" to this lab

sch schematic drawing of the demo in viewdraw

music.mcs - the bit stream of the design

music.exe - a driver to use the keyboard to control the frequency of sine wave generated by the FPGA

6. Design flow

6.1 Introduction

The essence of the design process using the BORG board for a multiple FPGA design can be summarized in the following steps.

1. Place and route **X1** and **X2** (the 2 *user* FPGAs), letting the placement and routing program `ppr` (or `apr`) choose the pad assignments.
2. Re-arrange the pad assignments of **X1** and **X2** with the `assign` utility to conform to the hardwired constraint of the BORG printed circuit board.
3. Place and route the **X1** and **X2** again using the incremental place-and-route flags of `ppr` (or `apr`).
4. Place and route **R1** and **R2** (the *routing* FPGAs).
5. Generate the bit streams of **R1**, **X1**, **R2**, and **X2** using `makebits` and concatenate them using `makeprom`.

Note: in principle, you can also treat **X1** and **X2** as the routing chips, and use **R1** and **R2** for logic; or even use all four FPGAs for logic. `Assign` is able to handle these situation, but you have to read Chapter 7.

6.2 Details

In greater details, suppose that you have two cooperating XC4003a LCA designs, the following steps illustrate the process of using the tool set to connect the two LCA designs electronically on the BORG board.¹

1. Hand partition your design into two XC4003aPC84 FPGAs.
2. Place and route the FPGA designs without imposing any constraints on the pad assignments. You should let `ppr` determine the pad assignments of your LCA designs. Say, the two (routed) LCA design files are called `X1a.LCA` and `X2a.LCA`; and their `XNF` files are called `X1a.XNF` and `X2a.XNF`, respectively.

```
C:> ppr X1a
C:> ppr X2a
```

3. Run “`assign`” with an “`alias.file`” to obtain an *interconnection map* `Rx.info`.

```
C:> assign -1 X1a.LCA -2 X2a.LCA -a alias.file
        -x1 X1a.cst -x2 X2a.cst -r1 R1.cst -r2 R2.cst -i
```

¹You may use an XC4002, XC4003, XC4004, XC4005, or XC4010D in place of any user FPGAs currently on your BORG board. This distribution provides two XC4003a as the user FPGAs, and two XC4002a as the routing FPGAs.

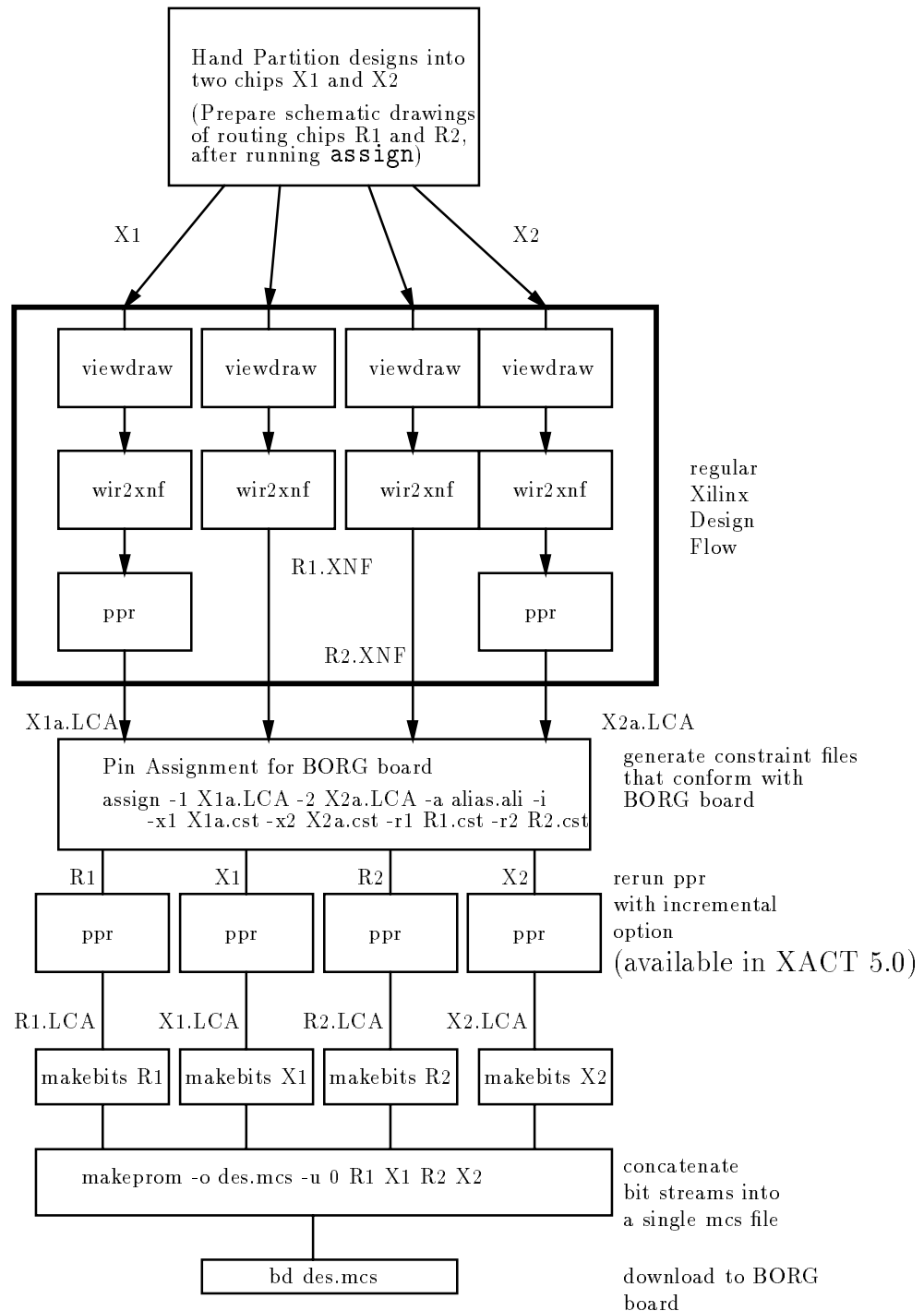


Figure 6.1: Using `Assign` to augment the Xilinx Design Flow for multiple-chip design. Draw the schematics of R1 and R2 after using `assign`, not before.

The `alias.file` is used to match nets which are to be connected between **X1** and **X2** which (may) have different names. Ideally, you created designs for **X1** and **X2** in which all nets that are to be interconnected have the same name. However, if for some reason, you gave different names to the signals, for example “Select” on **X1** and “select_data” on **X2**, an alias in the `alias.file` will cause these signals to be matched. This is particularly useful if you want to use the memory chip or PC-bus. You **MUST** alias those signals to the names given in the *wiring file* (refer to Chapter 7 for details). Some of these special signal names are:

PC Bus Data Lines:

```
&&BusData_0, &&BusData_1, &BusData_2, &&BusData_3,
&&BusData_4, &&BusData_5, &&BusData_6, &&BusData_7
```

PC Bus Address Lines:

```
&&BusAddress_0, &&BusAddress_1, &&BusAddress_2, &&BusAddress_3
```

PC Bus Control Lines:

```
&&BusControl_0, &&BusControl_1
```

Memory Data Lines:

```
&&MData_0, &&MData_1, &&MData_2, &&MData_3
&&MData_4, &&MData_5, &&MData_6, &&MData_7
```

Memory Address Lines:

```
&&MAddress_0, &&MAddress_1, &&MAddress_2, &&MAddress_3,
&&MAddress_4, &&MAddress_5, &&MAddress_6, &&MAddress_7
```

Memory Control Lines:

```
&&M_WE, &&M_OE, &&M_CS
```

Forced Nets:

```
&&R1, &&R2
```

The alias file itself contains pairs of net names that are to be matched. One example is:

```
;
; a sample alias file
; comments started with a semicolon
;
memaddr0 &&MAddress_0
memaddr1 &&MAddress_1
memaddr2 &&MAddress_2
memaddr3 &&MAddress_3
start Start_Machine
```

which illustrates forcing some nets to be used as memory address signals.

Another example is:

```
Prot<1>      &&R1
Prot<2>      &&R1
PLateral<1> &&R1
PLateral<2> &&R1
```

```

Plateral<3> &&R1
PMoveready &&R1
IOB1      &&R1
PYourmove &&R1
Pnewtile<1> &&R1
Pnewtile<2> &&R1
Pnewtile<3> &&R1
PSavcol<1> &&R1
PSavcol<2> &&R1
PSavcol<3> &&R1
- Pclk

```

which illustrates forcing some nets that must go to the **R1** FPGA.

The interconnection map `Rx.info` suggests a consistent way of connecting the user FPGAs **X1** and **X2** using the routing FPGAs **R1** and **R2**, hopefully.² A sample interconnection map is given below:

NET NAME	NET ALIAS	COST	SRC	DEST
-----	-----	----	---	----
PSavcol<3>	&&R1	[0]	X1.38	-> R1.48 O_PAD
PSavcol<3>	&&R1	[0]	X0	-> R1
Pcol<3>	Pcol<3>	[0]	X1.37	-> R2.5 I_PAD
Pcol<3>	Pcol<3>	[0]	X2.6	-> R2.28 O_PAD
Pc<13>	Pc<13>	[0]	X1.4	-> R2.4 I_PAD
Pc<16>	Pc<16>	[0]	X1.26	-> R1.60 I_PAD
Pc<16>	Pc<16>	[0]	X2.79	-> R1.49 O_PAD
Pcond<2>	Pcond<2>	[0]	X1.40	-> R1.44 O_PAD
Pnewtile<2>	&&R1	[0]	X2.81	-> R1.47 O_PAD
Pc<9>	Pc<9>	[0]	X1.23	-> R2.9 I_PAD
Pc<9>	Pc<9>	[0]	X2.27	-> R2.47 O_PAD
Prot<1>	&&R1	[0]	X0	-> R1
Prot<1>	&&R1	[0]	X2.7	-> R1.37 I_PAD
Pcond<7>	Pcond<7>	[0]	X2.70	-> R2.62 I_PAD
Pc<17>	Pc<17>	[0]	X1.36	-> R1.36 I_PAD
Pc<17>	Pc<17>	[0]	X2.14	-> R1.35 O_PAD
PYourmove	&&R1	[0]	X1.59	-> R1.24 O_PAD
PYourmove	&&R1	[0]	X0	-> R1
PMoveready	&&R1	[0]	X1.28	-> R1.57 I_PAD
PMoveready	&&R1	[0]	X0	-> R1
Pc<3>	Pc<3>	[0]	X1.5	-> R1.79 I_PAD
Pc<3>	Pc<3>	[0]	X2.26	-> R1.26 O_PAD
Pcond<4>	Pcond<4>	[0]	X1.19	-> R2.8 O_PAD
Pcond<4>	Pcond<4>	[0]	X2.23	-> R2.68 I_PAD
Pc<20>	Pc<20>	[0]	X1.68	-> R2.69 I_PAD
Pc<20>	Pc<20>	[0]	X2.68	-> R2.18 O_PAD

The first column is the PAD (net) name, the second is the PAD's alias name, the third column is the cost, and the fourth column is the source FPGA's pad number, and the last column is the destination FPGA's pad number and are connections that need to be made inside **R1** and **R2**.

²There may not be a consistent assignment and this problem is NP-complete.

`Assign` will also generate two constraints files `X1a.cst` and `X2a.cst`. Use these two files to route `X1a.LCA` and `X2a.LCA` with `ppr` again. You should use the incremental option of (`apr -g` for the XC3000 designs) `ppr` (available in XACT 5.0 in May 1994) to guide the new placement and routing processes using the old designs, and the new constraints files `X1a.cst` and `X2a.cst`. For example,

```
C:> ppr X1a outfile=X1
C:> ppr X2a outfile=X2
```

Now, you have two new LCA files `X1.LCA` and `X2.LCA` with the pad assignments determined by `assign`.

4. With the I/O map generated by `assign`, draw a schematic diagram for each of the routing chips, **R1** and **R2**, using `viewdraw`. The constraint files for the routing chips have also been generated by `assign`.

Figure 6.2 illustrates a rather typical schematic drawing of the R1 routing chip.

Notice that there is actually some logic in the “routing chips.” Please generate the routed LCA files of the routing chips using the Xilinx ADI software `wir2xnf` and `ppr` (or `xmake`, if you like).

```
C:> wir2xnf r1
C:> wir2xnf r2
C:> ppr r1
C:> ppr r2
```

Now you have two routed LCA files: `R1.LCA` and `R2.LCA`.

5. You generate the bit files for all the LCA files:

```
C:> makebits X1
C:> makebits X2
C:> makebits R1
C:> makebits R2
```

Now you put these bit files together into a single `mcs` file. Use `makeprom`, and set the `promsize` to 64K, set the file format to Intel `mcs`, and load the bit files in the upward direction starting from location 0. Gather the bit files and concatenate them into a single `mcs` file, say `design.mcs`, by loading the bit files in the following order

```
makeprom -o design.mcs -u 0 R1.bit X1.bit R2.bit X2.bit
```

The order is important since it corresponds to the order in which the FPGAs are daisy-chained on the BORG board.

6. Download the `mcs` file using the program `bd`.

```
C:> bd design.mcs
```

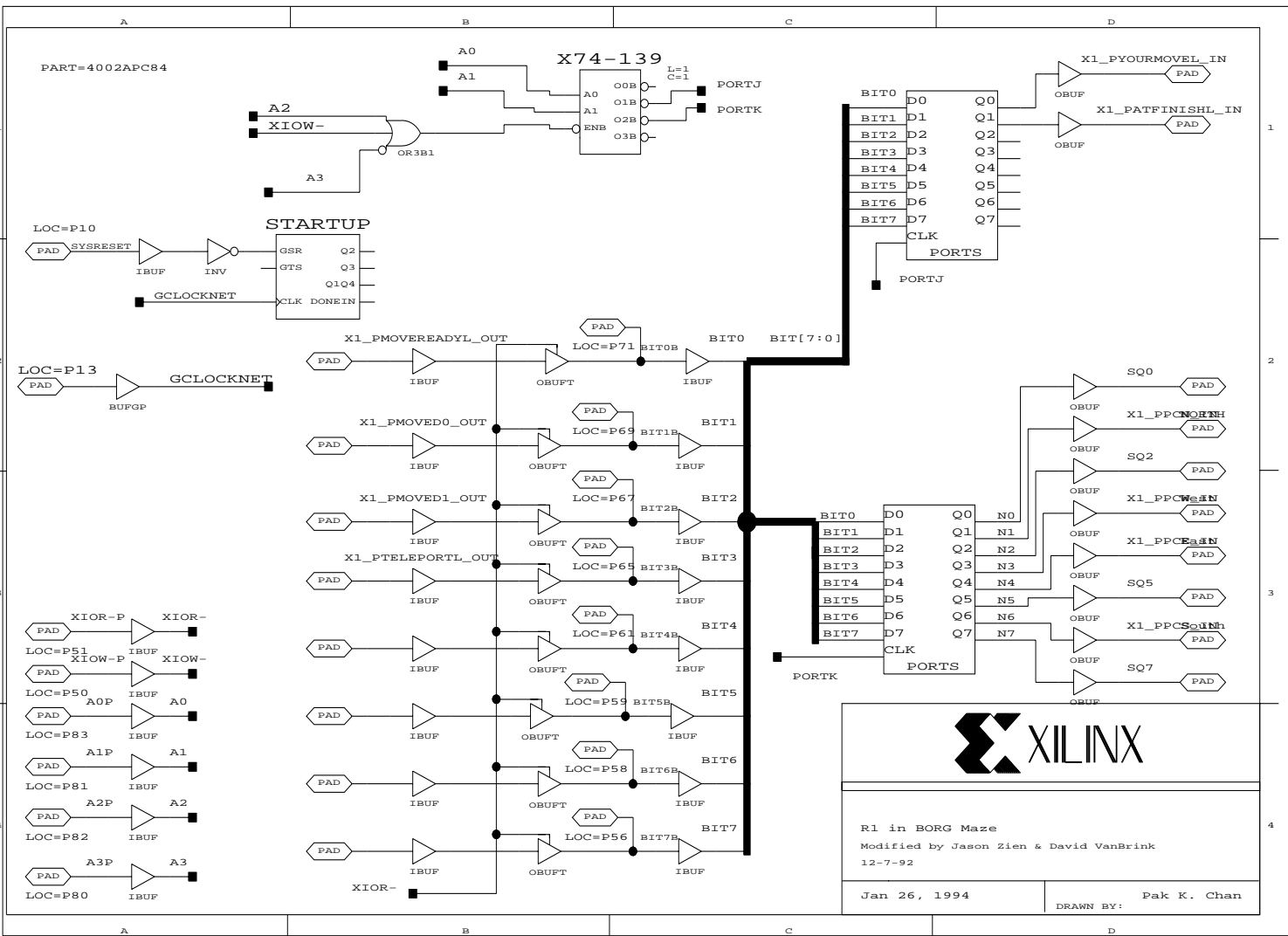


Figure 6.2: Maze runners' top-level schematic drawing of an R1 routing chip using the BORG board.

7. ASSIGN (Ver 3.0) A Pin Assignment Program for BORG Prototyping Board

7.1 Preface to earlier versions

¹ **Assign 3.0** may not necessarily be compatible with earlier versions of **assign**. This new version generates pin assignment for connections involving *one or two* user FPGAs (earlier versions are restricted to connection between two user FPGAs). Although the algorithms used are deterministic, they are dependent upon the ordering of the pads read in from the LCA files. The ordering of the pads is randomly changed after each iteration, that is why multiple iterations of the algorithm can be run. Therefore, minor changes to the LCA files may yield very different output from **assign**. **Assign 3.0** has been tested with Xilinx **apr 3.2**, **apr 3.3** (with incremental place and route version), and **ppr 1.31** (*without* incremental place and route). Incremental placement and routing is necessary for the *efficient* use of **assign**. **Assign** supports XILINX XC3020, XC3030, XC3042, XC3064, XC3090, XC4002, XC4003, XC4004, and XC4005 PC84-package FPGAs used in the BORG I and II prototyping board. The **X1** and **X2** user FPGAs are assumed to be of the same type.

7.2 Assign as a Pin Assignment Program

Locking (constraining) I/O pins down during placement and routing is known to be harmful. Not only that it increases the time taken to place and route a design, but locking down I/O pins also reduces your chances of having a successful placed and routed designs. **Assign** is a pin (I/O pad) assignment program which will increase the chance of successful placement and routing runs even under the given BORG board level constraint.

Assign does so in two steps. First, use the placement and routing program (**ppr** or **apr**) to place and route your designs *without constraints*. In other words, let **ppr** or **apr** choose the initial pin assignments freely. Next, **assign** will then perturb the initial pin assignments to satisfy the board level constraint. The designs are rerouted using the incremental placement and routing option.

7.2.1 Place in the design process

Assign is a program that produces consistent pin assignments for the BORG prototyping board. **Assign** takes two user LCA files which are intended to be downloaded to BORG, and produces two corresponding constraint files which can then be used by **apr** or **ppr** to generate a valid pin assignment.

BORG is a rapid prototyping board for PC-based machines. It contains two user-programmable XILINX FPGAs (X1 and X2) and two dedicated routing FPGAs (R1 and R2) as shown earlier in Fig. 1.2. Wires going from each user chip to each routing

¹ **Assign** is written by Jason Y. Zien

chip follow *roughly* an alternating pattern (wire i of X1 goes to R1, wire $i+1$ of X1 goes to R2, wire $i+3$ of X1 goes to R1, ...). Since BORG contains hardwired connections between the user FPGAs and routing FPGAs, the initial pin assignments generated by the XILINX tools (which have no knowledge of these board-level pin assignment constraints), must be rearranged to produce a correct, usable circuit. The advantage of having hardwired connections is the elimination of wire-wrapping a circuit, which can be extremely time consuming and tedious.

The typical design process for BORG has several steps. `Assign` fits in the middle of that process. The major steps in the design process are:

1. Draw schematics for X1 and X2 FPGAs.
2. Draw schematics for R1 and R2 FPGAs.
3. Create the unconstrained LCA files using `apr` or `ppr`.
4. Create the alias file for `assign` to match up nets with different names or to force nets to go to a specific routing chip.
5. Run `assign` on the X1 and X2 LCA files.
6. Edit the R1 and R2 schematics so that nets which pass through these chips are connected. These net names must match the incoming X1 or R1 net name, with the `X1_` or `X2_` prefix attached, depending on the source of the net.
7. Rerun `apr` or `ppr` on each LCA file *using the incremental placement and routing option*. For `apr`, use the `-c [file.cst]` option so that the constraint file generated by `assign` will lock the pads to the proper places. If running `ppr [file.xnf]`, it will automatically read in a constraint file named `[file.cst]`.
8. Use `makebits` to create the `bit` files.
9. Use `makeprom` to group together the `bit` files into one `.mcs` file for downloading.

IMPORTANT: The constraint files generated for R1 and R2 prepend either `X1_` or `X2_` to a net name depending on the source of the net. This is necessary because if matched nets in X1 and X2 have the same name, that would cause a name clash on the routing chip through which the net pass. Example: suppose nets `neta` on X1 and `bnet` on X2 are to be forced through R1. The net adjacent to the pad in which `neta` enters/leaves must be named `X1_neta` while the net adjacent to the pad in which `bnet` leaves/enters must be named `X2_bnet`. This only applies if one uses the constraint files generated by `assign`. Of course, one may choose to not use these net names, and directly set the pad locations in the schematic based on the information in `Rx.info`.

IMPORTANT: The user must exercise extreme care in making sure that nets which are NOT to be matched have different names. In particular, one needs to be careful of such things as `CLOCK` nets. `assign` may inadvertently match the clock signals on both user chips. See Section 7.2.4.

7.2.2 Command Line Arguments

`assign` takes a number of command line arguments. Its usage is as follows:

```
assign [options ...]
```

Options (and their descriptions, which follow the ';' symbol) include:

```
-1 x1file.lca ; x1file.lca=name of the X1 lca file
-2 x2file.lca ; x2file.lca=name of the X2 lca file
-a aliasfile  ; aliasfile= file that gives aliases to nets for matching
-u           ; flag, use memory connections
-s num      ; num=starting window size
-e num      ; num=ending window size
-m num      ; num=maximum solutions allowed outside the window
-x1 x1file.cst ; x1file.cst=name of the constraint file for chip X1
-x2 x2file.cst ; x2file.cst=name of the constraint file for chip X2
-r1 r1file.cst ; r1file.cst=name of the constraint file for chip R1
-r2 r2file.cst ; r2file.cst=name of the constraint file for chip R2
-i          ; run single and pairwise swap improvement phase
-g          ; run greedy graph reduction
-c          ; Output a CLB Locking constraint file (for apr ver 3.3)
```

If none of the constraint file output options (`-x1 -x2 -r1 -r2`) are specified, then by default, the program writes out constraint files `x1.cst`, `x2.cst`, `r1.cst`, and `r2.cst`. The constraint file output format is chip-specific. That is, the constraint files for Xilinx XC3000 series FPGAs differ from XC4000 series FPGAs. It is assumed that XC3000 series designs will be placed and routed using `apr` while XC4000 series designs will be placed and routed using `ppr`. The output constraint files are generated to be compatible with the corresponding place and route program.

The `-u` option allows `assign` to use special lines from R2 to the on-board memory. Because of pin limitations of the FPGA packages used, and due to the large number of memory address lines, these lines are selectively activated or not activated by some switches on the BORG board. If the memory lines are not used, then extra connections between the routing chips and user chips are available for general use. However, if the memory lines are used, then these connections are unavailable for general-purpose use. This option affects the use of all memory address lines for the 4K borg, but only the upper address lines (bits 8-10) of the 3K BORG. *** BE SURE THAT THE BORG DIP SWITCHES which affect the memory lines are set properly, or your design might not work!!!! *****

Due to a change in the way `apr ver 3.3` handles the locking of blocks, the `-c` option of `assign` should be used to speed up the placement phase of `apr`. When `-c` is used, two files, `x1clb.cst` and `x2clb.cst` are created and the line `'Include x1clb.cst;'` is included at the end of `x1.cst` and `'Include x2clb.cst;'` is included at the end of `x2.cst`. The files `x1clb.cst` and `x2clb.cst` lock all of the CLBs which were found in the input LCA files.

Previously, the recommended usage of constraint files generated by assign was:

```
% assign -1 x1.lca -2 x2.lca -a file.ali
% apr -1 -c x1.cst x1.lca x1new.lca
% apr -1 -c x2.cst x2.lca x2new.lca
```

The `-c` option does not do anything when the chips are Xilinx XC4000 series FPGAs.

Now, for `apr ver 3.3` (and later versions) we recommend:

```
% assign -1 x1.lca -2 x2.lca -a file.ali -c
% apr -q -c x1.cst x1.lca x1new.lca
% apr -q -c x2.cst x2.lca x2new.lca
```

By default, `assign` uses the augmentation algorithm. It has been experimentally noted that using the default mode tends to produce better results for very large, dense I/O designs, while the greedy graph reduction heuristic (`-g` option) tends to produce better results for small, sparse designs.

In order to run `assign`, the user first needs to have *at least one* LCA file which he/she intends to download to X1 and X2 of the BORG prototyping board. Also, an alias file may be created so that nets having different names in the two LCA files can be matched (or prohibited from being matched). These net names **MUST BE** adjacent to I/O pads. `Assign` can not match nets which are not adjacent to I/O pads. `Assign` is **NOT** case sensitive with respect to net names, however, the special alias names which will be described below are case-sensitive.

7.2.3 An Environment Variable

You need to set an environment variable before `assign` can be run. In the UNIX environment, the following line must be placed in the user's `.cshrc` file.

```
setenv BORG_ASSIGN <Directory_Where_Assign_Resides>/
```

In the MS-DOS environment, the following lines must be added to the `autoexec.bat` file:

```
set BORG_ASSIGN=<Directory_Where_Assign_Resides>/
```

where `<Directory_Where_Assign_Resides>` is the full path to the directory in which the `assign` program has been installed and which also contains the three data files: `xc3020.io`, `xc3042.io`, `alt3042.wir`. Also, the directory contains several pin mapping files used internally, which are: `3020.map`, `3030.map`, `3042.map`, `3064.map`, `3090.map`, `4002.map`, `4003.map`, `4004.map`, and `4005.map`.

7.2.4 Alias Files

By default, `assign` matches ALL nets in X1 and X2 which have the same name (insensitive to case). An alias file is used to match nets which are to be connected between two user FPGA chips which have different names. In the ideal case, the user has created their design for the two user FPGAs X1 and X2 such that all nets which are to be interconnected have the same name.

The alias file itself contains pairs of net names that are to be matched. The first column should be the X1 net name. The second column can contain the X2 net name, or one of the special reserved names given above. If the first and second column are X1 and X2 nets respectively, then the third column may contain one of the special reserved names to force both other nets to go through a particular routing chip.

A special name is the - symbol. If the - symbol is the first name, then the next string name signifies a net that is not to be matched by `assign`. This may be useful for example, when a net such as the CLOCK net appears in both user chips, but have already been given fixed locations which should not be modified by `assign`.

Another situation which requires the use of aliases is if for some reason, the user gave different names to the signals, for example `Select` on X1 and `select_data` on X2, an alias in the alias file will still allow the signals to be matched. This is particularly useful if the user wants to use the memory chip or PC-bus. The user MUST alias those signals to the names given in the wiring file. Those special signal names are given below.

Forced Nets (nets forced to either R1 or R2):

```
&&R1, &&R2
```

PC Bus Data Lines:

```
&&BusData_0, &&BusData_1, &BusData_2, &&BusData_3,  
&&BusData_4, &&BusData_5, &&BusData_6, &&BusData_7
```

PC Bus Address Lines:

```
&&BusAddress_0, &&BusAddress_1, &&BusAddress_2, &&BusAddress_3
```

PC Bus Control Lines:

```
&&BusControl_0, &&BusControl_1
```

Memory Data Lines:

```
&&MData_0, &&MData_1, &&MData_2, &&MData_3 &&MData_4,  
&&MData_5, &&MData_6, &&MData_7
```

Memory Address Lines:

```
&&MAddress_0, &&MAddress_1, &&MAddress_2, &&MAddress_3,  
&&MAddress_4, &&MAddress_5, &&MAddress_6, &&MAddress_7
```

Memory Control Lines:

```
&&M_WE, &&M_OE, &&M_CS
```

In practice, it is sufficient to force nets using just `&&R1` and `&&R2`. The other aliases are included for backward compatibility with previous versions of `assign`. For example, using `&&M_WE` is equivalent to `&&R2`. An example of an alias file is given below.

```

; some single forced nets
Dir0 &&BusData_0
Dir1 &&BusData_1
Dir2 &&BusData_2

ROTS &&BusData_3
finish &&BusData_4
Startin &&BusData_5

tile_0 &&BusAddress_0
tile_1 &&BusAddress_1
tile_2 &&BusAddress_2

; some alias matching plus forced nets
CE0 clken0 &&R1
CE1 clken1 &&R1
CE2 clken2 &&R1
CE3 clken3 &&R1
CE4 clken4 &&R1
CE5 clken5 &&R1
CLK_in CLKin &&R2
FIT Fit_in &&R2

; some matching aliases
ROT_IN ROTs
T0 CS0
T1 CS1
T2 CS2
T3 CS3
T4 CS4
T5 CS5
clken0 CEO

; some nets with same name that SHOULD NOT be matched by assign
- GlobalClock
- GlobalReset

```

7.2.5 Rx.info

The `Rx.info` file contains information necessary to generate the routing chips LCA files for downloading (see Fig. 7.1). The first column is the pad (net) name, the second is the pad (net) alias name, the third column is the cost (distance in usable pads from its original pad position), the fourth column is the source chip and pin, and the last column is the destination chip and pin.

NET NAME	NET ALIAS	COST	SRC	DEST
-----	-----	----	---	----
Fit_in	&&R2	[1]	X1.84 ->	R2.2 ?_PAD
Fit_in	&&R2	[1]	X0 ->	R2
CE5	&&R1	[0]	X0 ->	R1
CE5	&&R1	[0]	X2.84 ->	R1.84 O_PAD
CS4	T4	[0]	X1.9 ->	R2.8 I_PAD
CS4	T4	[0]	X2.37 ->	R2.51
CS0	T0	[1]	X1.83 ->	R1.2 I_PAD
CS0	T0	[1]	X2.45 ->	R1.40
ROT_IN	&&BusData_3	[1]	X1.47 ->	R1.42 O_PAD
ROT_IN	&&BusData_3	[1]	X0 ->	R1
Col	COL	[1]	X1.70 ->	R2.82 O_PAD
Col	COL	[1]	X2.48 ->	R2.59 I_PAD
clken1	&&R1	[3]	X1.71 ->	R1.77 I_PAD
clken1	&&R1	[3]	X0 ->	R1
clken5	&&R1	[2]	X1.72 ->	R1.73 I_PAD
clken5	&&R1	[2]	X0 ->	R1
tile_1	&&BusAddress_1	[0]	X1.63 ->	R1.63 O_PAD
tile_1	&&BusAddress_1	[0]	X0 ->	R1
Clkin	&&R2	[3]	X1.3 ->	R2.4 O_PAD
Clkin	&&R2	[3]	X0 ->	R2
CE3	&&R1	[0]	X0 ->	R1
CE3	&&R1	[0]	X2.3 ->	R1.3 O_PAD
CS2	T2	[2]	X1.81 ->	R1.83 I_PAD
CS2	T2	[2]	X2.20 ->	R1.18
CE2	&&R1	[0]	X0 ->	R1
CE2	&&R1	[0]	X2.82 ->	R1.82 O_PAD
decall	DECALL	[1]	X1.16 ->	R2.10 I_PAD
decall	DECALL	[1]	X2.63 ->	R2.68 O_PAD
CS1	T1	[0]	X1.77 ->	R2.84 I_PAD
CS1	T1	[0]	X2.66 ->	R2.70
FIT	&&R2	[0]	X2.83 ->	R2.83 I_PAD
CE1	&&R1	[0]	X0 ->	R1
CE1	&&R1	[0]	X2.78 ->	R1.78 O_PAD
clken2	&&R1	[0]	X1.66 ->	R1.68 I_PAD
clken2	&&R1	[0]	X0 ->	R1

Figure 7.1: A sample Rx.info file.

There may be some extraneous rows generated in `Rx.info`. These are output for informative purposes and the user need not use the information in any way.

`Assign` infers the pad type based on the `file1.lca` and `file2.lca` pads. Currently, it only supports `I_PAD` and `O_PAD` types, and all other pads output in the `Rx.info` file are marked `?_PAD`. The user must determine the pad type in those cases.

54 7. ASSIGN (Ver 3.0) A Pin Assignment Program for BORG Prototyping Board

```
Place Block clken4 P63;
Place Block Clkin P16;
Place Block Dir1 P2;
Place Block CS1 P17;
Place Block CS4 P3;
Place Block finish P71;
Place Block ROTS P39;
Place Block zero P44;
Place Block Dir0 P4;
Place Block CS2 P15;
Place Block clken5 P72;
Place Block Startin P21;
Place Block ROT_IN P30;
Place Block decall P60;
Place Block clken0 P56;
Place Block Dir2 P19;
Place Block Fit_in P84;
Place Block CS5 P77;
Place Block clken2 P66;
Place Block clken3 P61;
Place Block CS0 P9;
Place Block tile_2 P26;
Place Block CS3 P8;
Place Block tile_0 P28;
Place Block Col P18;
Place Block clken1 P68;
Place Block tile_1 P24;
;
; Comment out next line if CLB locking is not desired
Include x1clb.cst;
```

Figure 7.2: A Sample XC3000-series Constraint File.

```
all: chip.ali eval2.lca brains2.lca

brains2.cst, eval2.cst:    brains2.xnf eval2.xnf chip.ali
    # run apr once without constraints to generate lca files for assign
    # - the next 2 lines may be unnecessary in subsequent design runs
    apr brains2.lca
    apr eval2.lca
    assign -1 brains2.lca -2 eval2.lca -a chip.ali \
        -x1 brains2.cst -x2 eval2.cst -r1 r1.cst -r2 r2.cst -i -g

brains.lca:    brains2.cst
    apr -q -c brains2.cst brains2.lca brains.lca
    makebits brains2

eval.lca:    eval2.cst
    apr -q -c eval2.cst eval2.lca eval.lca
    makebits eval2
```

Figure 7.3: A Sample Makefile for XC3000 Series FPGAs.

7.2.6 Examples of using assign

Assign tries to generate a consistent pad assignment that matches all pads of the same name between the two LCA files. **Assign** produces up to five output files, (four `.cst` constraint files – one per chip) and a summary file, `Rx.info` (on DOS machines `Rx.inf`). The constraint files are then used by **apr** (for XC3000 series FPGAs) or **ppr** (for XC4000 series FPGAs) to force the pin assignments of the appropriate nets. First, let us assume that the user already has generated the **XNF** files for his/her design. In order to complete the design, the user must create unconstrained LCA files, run **assign** and then create constrained LCA files.

7.2.7 Xilinx XC3000 Series Design

The **Makefile** in Fig. 7.3 shows the process of generating a XC3000 series design and Fig. 7.2 shows an example of a constraint file. The constraint file consists of two parts. The first part locks all the IOBs, and the second part locks the CLBs, if the `-c` option was used. In the rare event that **apr** can't complete the routing process, unlocking the CLBs by commenting out the last line

```
Include x1clb.cst;
```

of the constraint file should help. Note that you must also create the routing chips and place and route them before the final design can be downloaded.

7.2.8 XC4000 Series Design

The design flow for XC4000 parts is very much like that of XC3000-series parts, except you use **ppr** instead of **apr**; except that the current version (April 1994) of **ppr** has no incremental placement and routing option. We shall update **assign** as soon as the incremental place and route option is available with **ppr**.

The constraint files generated thus conform to the syntax expected by **ppr**, and also have the same pre-extension name as the **XNF** file to be placed and routed. Figure 7.5 shows a **Makefile** for running **assign**. In Fig. 7.4, we have shown the constraint file generated for a routing chip. Notice that there are **X1_** and **X2_** prefixes to the normal net names, indicating which user chip the nets come from. The same prefixes are used in XC3000-series routing chip constraint files.

7.3 I/O Specification File

There are two special files used by **assign**. These are `xc3020.io` and `xc3042.io`. These files contain information about the physical pin locations on the chip (which is 84 pin PLCC package) and the usable pins. You should not change these files. The commands contained in the files include:

```
; a semicolon in the first column of a line denotes a comment MAP
<pin# start: pin# end> -> ( start_x:start_y, end_x:end_y) IO
<startpin:endpin> <startpin:endpin> .... CIO <startpin:endpin>
<startpin:endpin> ...
```



```

Place instance X1_PSavcol<3>: P48;
Place instance X2_PLateral<2>: P46;
Place instance X2_Pnewtile<3>: P39;
Place instance X1_Pc<12>: P62;
Place instance X2_Pc<12>: P14;
Place instance X2_Pnewtile<2>: P47;
Place instance X2_Prot<1>: P37;
Place instance X1_Pc<17>: P36;
Place instance X2_Pc<17>: P35;
Place instance X1_PYourmove: P24;
Place instance X1_PMoveready: P57;
Place instance X1_Pc<3>: P79;
Place instance X2_Pc<3>: P26;
Place instance X1_PSavcol<2>: P18;
Place instance X2_Prot<2>: P29;
Place instance X2_Pnewtile<1>: P28;
Place instance X2_PLateral<1>: P4;
Place instance X1_Pcond<1>: P72;
Place instance X2_Pcond<1>: P25;

```

Figure 7.4: A Sample XC4000 Series Constraint File.

```

all:    amazerg.lca r2newg.lca r1newg.lca
        makeprom -o amazer4 -u 0 r1newg amazerg r2newg e4003a

amazerox.cst:    amazerox.ali amazerox.xnf
        # run ppr once without constraints to generate amazerg.lca
        # - the next line may be unnecessary in subsequent design runs
        ppr amazerox.xnf outfile=amazerg
        assign -l amazerg.lca -a amazerox.ali -s 1 -x1 amazerox.cst\\
                -r1 r1.cst -r2 r2.cst -i -u

amazerg.lca:    amazerox.cst
        ppr amazerox outfile=amazerg logfile=amazerg
        makebits amazerg

r2newg.lca:    amazerox.cst
        ppr r2 outfile=r2newg logfile=r2newg
        makebits r2newg

r1newg.lca:    amazerox.cst
        ppr r1 outfile=r1newg logfile=r1newg
        makebits r1newg

```

Figure 7.5: A Sample Makefile for XC4000 series FPGAs (non-incremental place and route version).

Use MAP to specify the relation between the actual pin number and the logical coordinate of the pin, taking the upper left corner of the chip to be (x=0, y=0) and the lower right to be (x=22, y=22).

IO specifies the list of usable pins on the particular chip. Finally, CIO specifies the list of possibly usable pins (pins which are used in configuration mode, but may be used later).

7.4 BORG Wiring File

The `alt3042.wir` file contains a net list of physical wires on the XC3000 BORG board. The `4k.wir` file contains a net list of physical wires on the XC4000 BORG board. The file specifies how your X1 and X2 FPGAs are connected to the routing (R1, R2) chips. The BORG wiring configuration is hardwired, so this file should NOT be changed by the user.

The connections are specified by:

```
<source>.<pin#> -> <dest>.<pin#> [&&alias_name]
```

where source \in X1, X2, X0, M1 and dest \in R1, R2. A comment is denoted by a ';' semicolon at the start of a line. The X0 chip is an on-board chip of BORG which contains logic to interface to the PC bus. The M1 chip is the memory chip. The optional

```
[&&alias_name]
```

parameter is ONLY used with X0 and M1 mapping in order to specify the alias name for these *forced nets*. The actual wiring configuration is listed in Section 7.11.

7.5 Theory of ASSIGN

The pin assignment problem is formulated as a graph problem, which we call the *two-color assignment problem*. The goal of the two-color assignment problem is to find a consistent, minimum weight node assignment. I describe my solution to the problem, which uses two methods, called *graph reduction* and *augmentation*².

7.6 Problem Description

The problem is formally defined as follows: Graph $G(V, E)$ consists of three sets of vertices, P, Q, and N, which are connected by a set of edges such that every edge has one endpoint in $P \cup Q$, and the other endpoint in N. The N vertices represent the nets which need to be matched on the user-programmable chips. The P vertices represent the X1 pads to which the nets may be assigned, and the Q vertices represent the X2 pads to which the nets may be assigned.

²The augmentation algorithm was created and implemented first by Professor Martine Schlag

$(P \cup Q, N)$ is a bipartition of G . Each vertex $v \in P \cup Q$, has a color, $c(v)$ =red or green. These colors correspond to choosing a routing path through a routing chip R1 or R2, so the color of the vertex in P and the color of the vertex in Q of a matched pair must be the same. It is because of these colors that a standard graph matching algorithm cannot be used. A valid assignment consists of two one to one functions, f_p and f_q , which map a vertex from N to either P or Q .

$f_p: N \rightarrow P, f_q: N \rightarrow Q$

The weight of an assignment is the sum of all of the edge weights in the assignment. Ideally, one would like to find the graph assignment of minimum weight. Edge weights in the graph represent the distance of the chosen pad from the original pad assigned by **apr**. It is beneficial to perturb the pad positions as little as possible so that **apr** may be able to re-route the design WITHOUT re-placing the design, saving a substantial amount of time.

In general, $|N| \leq |P|$ and $|N| \leq |Q|$. There is one further constraint where certain nets (such as those which go to the memory or PC bus) must be assigned to a specific color (routing chip). These nets are called *forced nets*. The corresponding pads associated with these forced nets are called *forced-net-pads*.

7.7 Graph Reduction

The first method for generating consistent pin assignments is called graph reduction. The graph reduction heuristic works as follows:

1. Remove edges from the graph that are impossible to match.

These are the pads of some color c in set P which have no corresponding pads of the same color in set Q , or vice versa. Repeat this step until there are no more impossible edges to remove.

2. Find and remove forced pads.

A *forced pad* is one which some net MUST choose because it has no other unmarked pads to choose from. These forced pads are NOT ONLY forced-net-pads (defined above), but also pads which are forced due to vertex removal done in the next step. The pad is marked as part of the solution set. Repeat this step until no more forced pads remain.

3. Remove one vertex from the graph.

The edge removed depends on the current operating mode of the algorithm. In GLOBALLY GREEDY mode, the edge chosen for removal is the largest weight edge remaining in the graph. In LOCALLY GREEDY mode, the vertex removed is the largest weight edge of the net at the head of the queue containing unassigned nets. In RANDOM mode, the vertex removed is the vertex being considered when a random number exceeds a threshold value (varied from 50% to 90%). Vertices are considered based on their order in the list of vertices connected to a particular net in N .

4. While there are still edges in G , loop back to the first step.

This algorithm is fairly fast (polynomial time), and, if it finds a solution, it is likely to be very close to the ideal solution since high weight edges are removed. The main problem with this heuristic is that incomplete solutions may be generated since a greedy vertex removal might cause some nets to become unassignable. Thus, after the entire algorithm has completed, two more solution-searching phases are used: Find_Last, and Augment2() (described in the next section).

The Find_Last phase looks at every unassigned net and searches for any vertex (pad) which is unused and which the net can use. These are vertices that may have been discarded in the greedy graph reduction. If one is found, the assignment is made.

7.8 Augmentation

There are two augmentation algorithms used: Augment1() and Augment2(). Both algorithms search for alternating paths in the N, P and N, Q subgraphs. A breadth first search is done on the graph starting with an incompletely assigned net vertex. The algorithm recursively searches for a net which can choose some other pad for its solution. In the Augment1() algorithm, the net looks only at pads of the same color as its current solution for possible swapping. This is a standard augmenting path algorithm consisting of only N and pads in P of the same color. In the Augment2() algorithm the net also checks to see if a net can swap its solution with pads of the opposite color.

The simplest way to describe the algorithm is with an example. Figure 7.6 illustrates how the Augment1() procedure works. In the figure, the dashed lines show pads which a net may choose, provided that no other net has chosen to use that pad. Solid lines represent a pad that a net has chosen as part of its matching. The O's inside the vertices represent routing chip R1, the X's represent routing chip R2. Net **a** is currently incompletely assigned. So, net **a** looks at all of the other nets which have a solution that it can use. In this case, net **b** is the only one. Now, net **b** checks to see if it can pick some other pad so that it can give its solution to net **a**. It cannot, so it looks at all nets which have a solution that it could possibly use. In this case, it looks at net **c**. Net **c** cannot choose any other pads for its solution, so we recur once again, and check if net **d** can choose some other pad for its solution. It can. So, net **d** takes the unassigned pad, and then returns the pad it gave up, so net **c** can take that pad and return its previous solution to net **b**, which finally gives up its previous solution to net **a**.

The Augment2() procedure is nearly the same as that of Augment1(). In fact, Augment1() is called as a subroutine from Augment2(), and if no solution is found by Augment1(), then the algorithm searches for pads of the opposite color which a net can take as its solution. Note that forced nets cannot be considered because they can not change colors (routing chips). Figure 7.7 illustrates how the Augment2() procedure works. Starting at net **a**, we consider all nets that have a solution net **a** could use. Nets **b** and **d** are the only ones. Net **b** cannot pick solutions of the opposite color, so we recursively check all nets which could give up its solution to net **b**. Net **c** is such a net. Now, net **c** can pick a solution pair of the opposite color, so it does. Net **b** can then pick a pair of solutions of the opposite color. Finally, net **a** can

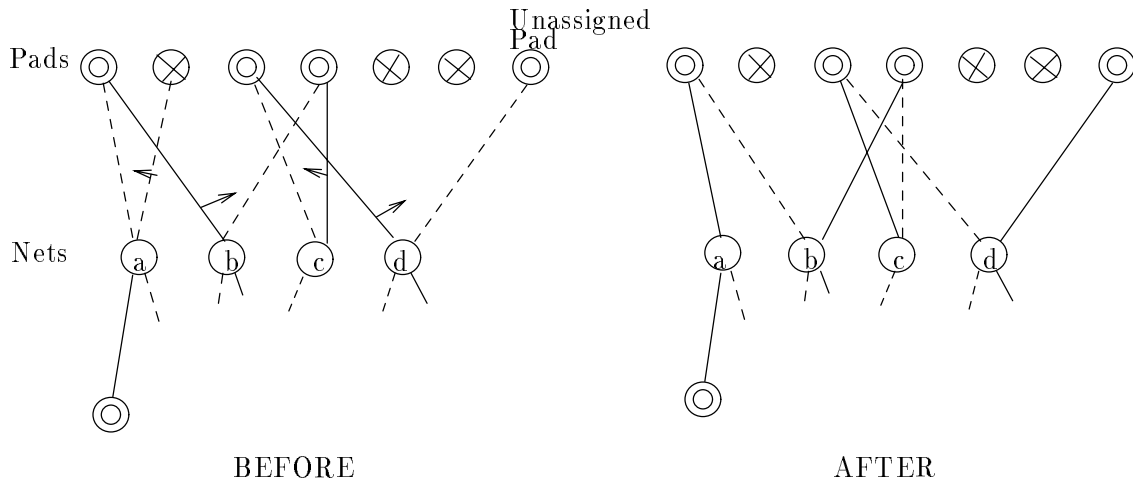


Figure 7.6: Example of the `Augment1()` Algorithm

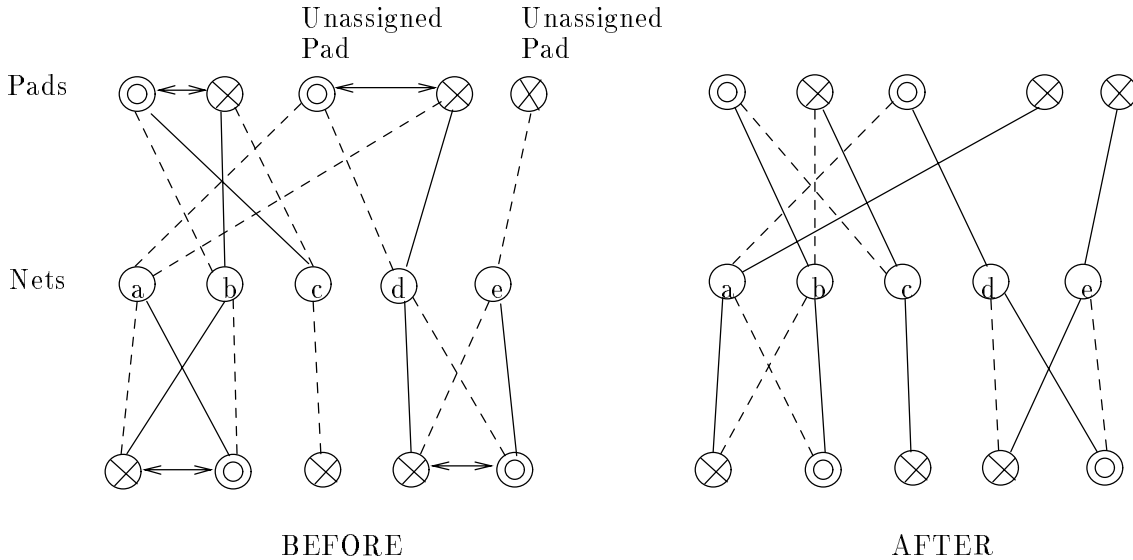


Figure 7.7: Example of the `Augment2()` Algorithm

be assigned a new solution pair. The algorithm is then executed from the beginning again, starting at net **e**, since it does not have a complete assignment, and a solution is eventually found for it.

The pseudocode for each of the two algorithms is nearly identical, so we shall only provide the code for `Augment2()` in Fig. 7.8. Many of the details of the algorithm have been left out so that the general idea of the algorithm would not be overwhelmed by the particular implementation details.

7.9 Main Program Loop

The augmentation and graph reduction algorithms are the major components of `assign`, but it is also useful to see how they are used in the overall scheme of the

```

Augment2(NetNodes,SOLUTION_DESIRED) {
    /* Recursive Breadth-First-Search */
    for each unassigned node 'cnn' {
        mark cnn;

        /* check if net cnn can pick a pad of opposite color for
           its solution which some other net wants */
        pad = swap_if_available2(cnn,SOLUTION_DESIRED);
        if (!pad) pad = Augment1(NetNodes,SOLUTION_DESIRED);

        if (pad) return(pad);

        for each unmarked pad 'p' connected to cnn {
            for each unmarked node 'nn' connected to p {
                if ('nn' has a solution that cnn is looking for using pad p) {
                    mark p;
                    put nn onto NextQ;
                }
            }
        }
        // recursive call
        pad = Augment2(NextQ,SOLUTION_DESIRED);

        if (pad) {
            find the node 'cnn' which wants to use pad for its solution;
            rpad = swap(pad, cnn);
            return(rpad);
        }
    }
}

```

Figure 7.8: The `Augment2()` Algorithm.

program. Figure 7.9 shows the pseudocode for the main program loop, and for the `Solve()` procedure called by the main loop.

7.10 Performance

Let n be the number of nets to be pairwise assigned, p be the maximum number of pads each net can be assigned to, and w be the number of window sizes spanned.

The default `Augment2()` algorithm runs in $O(n^2p^2)$ time. This is because of the particular implementation of the breadth first search algorithm, which looks at every node, and every pad connected to every node. One would expect that the algorithm

```

Solve(mode, parameter) {

    /* ----- */
    /* first run graph reduction if the command line switch was set */
    /* ----- */
    if (switch option '-g') {
        graph_reduction(mode,parameter);
    }

    /* ----- */
    /* run improvement step, if the command line switch was set */
    /* ----- */
    if (switch option '-i') {
        Singular_Improvements();
        Pair_Wise_Swap_Improvements();
    }

    /* ----- */
    /* Run the Augment2() and Find_Last() procedures */
    /* ----- */
    do {
        foundsolution=0;
        for each remaining unmatched net {
            foundsolution += Find_Last();
            foundsolution += Augment2();
        }
    } while (foundsolution);

    /* ----- */
    /* run improvement step, if the command line switch was set */
    /* ----- */
    if (switch option '-i') {
        Singular_Improvements();
        Pair_Wise_Swap_Improvements();
    }

    Save_Solution_If_Better();
}

main() {

    /* try as many window sizes as is necessary */
    for (windowsize=start; windowsize<=end; windowsize++) {
        if (the option '-g' was used) {
            Solve(option,GLOBALLY_GREEDY,NULL);
            Solve(option,LOCALLY_GREEDY,NULL);
            for (i=50; i<=90; i+=10) {
                Solve(option,RANDOM,i);
            }
        }
        /* check for exit condition */
        if (Complete_And_Consistent_Solution_Found) exit_and_output_solution;
    }
}

```

Figure 7.9: The Main Program of Assign.

Design	Blocks	Nets	Pads Assigned	Switch Options	Total Weight	Runtime
Rb	71+125	77+92	29+19	—	18	7.2s
Rb	71+125	77+92	29+19	-i	17	13.9s
Rb	71+125	77+92	29+19	-g	25	13.1s
Rb	71+125	77+92	29+19	-g -i	25	18.3s
Mcl	71+132	72+118	26+21	—	21	7.2s
Mcl	71+132	72+118	26+21	-i	21	12.9s
Mcl	71+132	72+118	26+21	-g	23	12.1s
Mcl	71+132	72+118	26+21	-g -i	21	18.2s
Mtn	205+99	230+107	51+39	—	175	93.6s
Mtn	205+99	230+107	51+39	-i	158	282.7s
Mtn	205+99	230+107	51+39	-g	245	546.2s
Mtn	205+99	230+107	51+39	-g -i	185	933.6s

Table 7.1: Assign Performance

takes $O(np)$ time, but because of the call to `Augment1()` within `Augment2()`, the total execution time is $O(n^2p^2)$.

The greedy reduction algorithm runs in $O(np^2)$ time. This comes from the fact that at most p edges must be removed before the algorithm terminates, and for every edge removed, it takes $O(np)$ to find all of the forced pads and all of the pads which are unmatchable.

Running the improvement phase takes $O(n^2)$ time. So, the overall program performance is $O(wn^2e^2)$.

Table 7.1 shows the actual performance of the program on three designs. All tests were run with an initial window size of one, and were executed on a Sun Sparcstation 1+.

7.11 BORG wiring connections

7.11.1 XC3000-series BORG wiring connections

```

; Jan 29, 1992 (Pak K. Chan)
;
; X1
;
X1.83 -> R1.2
X1.84 -> R2.2
X1.2 -> R1.4
X1.3 -> R2.4
X1.4 -> R1.8
X1.5 -> R2.6
; X1.6 -> R1.6
; X1.7 -> R2.6
X1.8 -> R1.10
X1.9 -> R2.8
; X1.10 -> R1.10
; X1.11 -> R2.8

; pin 13 is for GCLK input
; X1.14 -> R1.14      R1.14 is connected to X0
X1.15 -> R1.15
X1.16 -> R2.10
X1.17 -> R1.17
X1.18 -> R2.36
X1.19 -> R1.19
X1.20 -> R2.41
X1.21 -> R1.21
X1.23 -> R2.18
X1.24 -> R1.24
X1.25 -> R2.20
X1.26 -> R1.26
X1.27 -> R2.48
X1.28 -> R1.28
X1.29 -> R2.50
X1.30 -> R1.30
; pin 33 is M2
;
X1.37 -> R2.52
X1.39 -> R1.34
X1.40 -> R2.56
X1.42 -> R1.36
X1.44 -> R2.58
;
;X1.38 -> R1.41
;
X1.45 -> R1.39
X1.46 -> R2.60
; X1.41 -> R1.41
X1.47 -> R1.42
X1.48 -> R2.62
X1.49 -> R1.45

```

```
X1.52 -> R2.65
X1.53 -> R1.47
X1.56 -> R2.67
X1.57 -> R1.49
; X1.50 ->
; X1.51 -> R2.50
X1.58 -> R2.69
X1.59 -> R1.53
X1.60 -> R2.71
X1.61 -> R1.59
X1.62 -> R2.76
X1.63 -> R1.63
X1.65 -> R2.78
X1.66 -> R1.68
X1.67 -> R2.80
X1.68 -> R1.75
X1.70 -> R2.82
X1.71 -> R1.77
X1.72 -> R1.73
; X1.72 and X1.73 can also be used as user I/O pins
; X1.73 -> R2.72
;
X1.77 -> R2.84
;
; end of 25 pins
; extra pins
X1.78 -> R1.81
X1.81 -> R1.83
; X1 extra pins for XC3030s
;
;X1.38 -> R1.38
;X1.41 -> R1.41
;X1.50 -> R1.50
;X1.51 -> R1.51
; one X2 pin for XC3030s
;X2.6 -> R2.7
;
X2.71 -> R2.79
;
; X2
; X2 north east face
;
X2.2 -> R2.3
X2.4 -> R2.5
;
X2.8 -> R2.9
X2.15 -> R2.11
; X2 north west face
X2.83 -> R2.83
X2.81 -> R2.81
X2.77 -> R2.77
X2.75 -> R2.75

; west face
; special addresses - BORG jumpers affect which lines are usable
; *** *REF1* The following 3 nets are not allowed when the memory
```

66 7. ASSIGN (Ver 3.0) A Pin Assignment Program for BORG Prototyping Board

```
; is used, otherwise, if the -u option is used in assign, then
; these lines are unavailable. See *REF2*

; X2.68 -> R2.15
; X2.70 -> R2.16
; X2.17 -> R2.17
;
;

X2.19 -> R2.42
X2.24 -> R2.19
X2.27 -> R2.21
X2.29 -> R2.49
; south face of X2
X2.37 -> R2.51
X2.40 -> R2.53
; extra from memory address A_11
X2.44 -> R2.14
;
X2.46 -> R2.57
X2.48 -> R2.59
; east face of X2
X2.57 -> R2.61
X2.59 -> R2.63
X2.61 -> R2.66
X2.63 -> R2.68
X2.66 -> R2.70
; end of 24 pins
;
; connection to R1
X2.3 -> R1.3
X2.5 -> R1.5
X2.9 -> R1.9
X2.16 -> R1.11
; west face
X2.18 -> R1.16
X2.20 -> R1.18
X2.23 -> R1.23
X2.25 -> R1.20
X2.26 -> R1.25
X2.28 -> R1.27
X2.30 -> R1.29
; south face
X2.35 -> R1.33
X2.39 -> R1.35
X2.42 -> R1.37
X2.45 -> R1.40
X2.47 -> R1.46
X2.49 -> R1.44
X2.52 -> R1.48
; east face
X2.58 -> R1.52
X2.60 -> R1.57
X2.62 -> R1.61
X2.65 -> R1.66
X2.67 -> R1.71
```

```

; east north face
X2.72 -> R2.73
X2.76 -> R1.76
X2.78 -> R1.78
X2.82 -> R1.82
X2.84 -> R1.84
; end

; force nets
; R1 force nets

; data bits

X0.1 -> R1.72 &&BusData_0
X0.1 -> R1.70 &&BusData_1
X0.1 -> R1.67 &&BusData_2
X0.1 -> R1.65 &&BusData_3
X0.1 -> R1.62 &&BusData_4
X0.1 -> R1.60 &&BusData_5
X0.1 -> R1.58 &&BusData_6
X0.1 -> R1.56 &&BusData_7

; address bits
X0.8 -> R1.79 &&BusAddress_0
X0.9 -> R1.80 &&BusAddress_1
X0.10 -> R1.69 &&BusAddress_2
X0.11 -> R1.14 &&BusAddress_3

; io control bits

X0.12 -> R1.6 &&BusControl_0
X0.13 -> R1.7 &&BusControl_1

; R2 forced nets
; memory data pins D0-D7
M1.9 -> R2.23 &&MData_0
M1.10 -> R2.24 &&MData_1
M1.11 -> R2.25 &&MData_2
M1.13 -> R2.26 &&MData_3
M1.14 -> R2.27 &&MData_4
M1.15 -> R2.28 &&MData_5
M1.16 -> R2.29 &&MData_6
M1.17 -> R2.30 &&MData_7

; memory address pins A0-A7
M1.8 -> R2.37 &&MAddress_0
M1.7 -> R2.38 &&MAddress_1
M1.6 -> R2.39 &&MAddress_2
M1.5 -> R2.40 &&MAddress_3
M1.4 -> R2.44 &&MAddress_4
M1.3 -> R2.45 &&MAddress_5
M1.2 -> R2.46 &&MAddress_6
M1.1 -> R2.47 &&MAddress_7

; special addresses - BORG jumpers affect which lines are usable

```

68 7. ASSIGN (Ver 3.0) A Pin Assignment Program for BORG Prototyping Board

```
; *** *REF2*
; The following 4 nets are usable when the -u option of assign is
; used. Otherwise, the *REF1* lines will be allowed.

M1.23 -> R2.17 &&MAddress_8
M1.22 -> R2.16 &&MAddress_9
M1.19 -> R2.15 &&MAddress_10
M1.19 -> R2.14 &&MAddress_11

; memory control pins WE OE CS
M1.21 -> R2.35 &&M_WE
M1.20 -> R2.34 &&M_OE
M1.18 -> R2.33 &&M_CS

; Dummy pins used by assign to generate forced nets for R1 and R2
X0.1 -> R1.1 &&R1
X0.1 -> R2.1 &&R2

; end
```

7.11.2 XC4000-series BORG wiring connections

```

;
;
; Oct 26, 1993 (Pak K. Chan)
; Jan 20, 1994 (Jason Y. Zien) Added memory address, data, control lines
;
; BORG II XC4000-PC84 wiring file
; wiring file for BORG II
;

; Dummy pins used by assign to generate forced nets for R1 and R2
X0.1 -> R1.1 &&R1
X0.1 -> R2.1 &&R2

; -----

; **** Memory lines ***
;   These are not present in 4knomem.wir
;   these lines are allowed when the -u command-line option
;   of assign is used, otherwise, 4knomem.wir is used if the
;   -u option is not used.
;   BORG dip-switch settings affect which set is physically active.

; mem. address lines
M1.1 -> R2.47 &&MAddress_0
M1.2 -> R2.48 &&MAddress_1
M1.3 -> R2.49 &&MAddress_2
M1.4 -> R2.50 &&MAddress_3
M1.5 -> R2.51 &&MAddress_4
M1.6 -> R2.56 &&MAddress_5
M1.7 -> R2.57 &&MAddress_6
M1.8 -> R2.58 &&MAddress_7
M1.9 -> R2.61 &&MAddress_8
M1.10 -> R2.62 &&MAddress_9
M1.11 -> R2.67 &&MAddress_10
M1.12 -> R2.66 &&MAddress_11

; mem. data lines
M1.12 -> R2.46 &&MData_0
M1.13 -> R2.45 &&MData_1
M1.14 -> R2.44 &&MData_2
M1.15 -> R2.40 &&MData_3
M1.16 -> R2.39 &&MData_4
M1.17 -> R2.38 &&MData_5
M1.18 -> R2.37 &&MData_6
M1.19 -> R2.36 &&MData_7

; mem. control lines
M1.20 -> R2.65 &&M_OE
M1.21 -> R2.60 &&M_WE
M1.22 -> R2.30 &&M_CS

```

70 7. ASSIGN (Ver 3.0) A Pin Assignment Program for BORG Prototyping Board

```
; *** end of Memory lines ***

; -----
; The following lines are not usable when the memory is used
; (not usable when -u option of assign is set).
; They are used by default.
; BORG dip-switch settings affect which set is physically active.
;
; A0
; X2.27 -> R2.47
; A1
; X2.35 -> R2.48
; A2
; X2.39 -> R2.49
; A3
; X2.40 -> R2.50
; A4
; X2.45 -> R2.51
; A5
; X2.50 -> R2.56
; A6
; X2.51 -> R2.57
; A7
; X2.47 -> R2.58
; A12
; X2.46 -> R2.59
; A8
; X2.37 -> R2.61
; A9
; X2.70 -> R2.62
; A11 and A10
; X1.56 -> R2.66
; X1.58 -> R2.67
; end

; -----

;
; R1
;
; R1 force nets to PC
;
X0.0 -> R1.71  &&BusData_0
X0.1 -> R1.69  &&BusData_1
X0.2 -> R1.67  &&BusData_2
X0.3 -> R1.65  &&BusData_3
X0.4 -> R1.61  &&BusData_4
X0.5 -> R1.59  &&BusData_5
X0.6 -> R1.58  &&BusData_6
X0.7 -> R1.56  &&BusData_7

; 4 address lines

X0.8 -> R1.83  &&BusAddress_0
X0.9 -> R1.81  &&BusAddress_1
```

```
X0.10 -> R1.82  &&BusAddress_2
X0.11 -> R1.80  &&BusAddress_3

; 3 io control lines
; xior
X0.12 -> R1.51  &&BusControl_0
; xiorw
X0.13 -> R1.50  &&BusControl_1
; interrupt
X0.14 -> R1.70  &&BusControl_3
;

X2.44 -> R1.14
X1.67 -> R1.18
X1.65 -> R1.19
X1.61 -> R1.20
X2.38 -> R1.23
X1.59 -> R1.24

X2.36 -> R1.25
X2.26 -> R1.26
X2.24 -> R1.27
X2.20 -> R1.28
X2.18 -> R1.29
X1.81 -> R1.3
X2.14 -> R1.35
X1.36 -> R1.36
X2.7  -> R1.37
X2.69 -> R1.4
X1.46 -> R1.40
X1.40 -> R1.44
X2.3  -> R1.45
X2.83 -> R1.46
X2.81 -> R1.47
X2.79 -> R1.49
X2.67 -> R1.5
X1.28 -> R1.57
X2.65 -> R1.6
X1.26 -> R1.60
X1.20 -> R1.66
X1.18 -> R1.68
X2.61 -> R1.7
X1.71 -> R1.72
X1.5  -> R1.79
X2.59 -> R1.8
X1.83 -> R1.84
X2.48 -> R1.9
;
; TDO
;R1.75 -> X1.15
X1.24 -> R1.62
X1.3  -> R1.78
X1.38 -> R1.48
;
X1.48 -> R1.38
```


72 7. ASSIGN (Ver 3.0) A Pin Assignment Program for BORG Prototyping Board

```
X2.5 -> R1.39
;
; R2
;
X1.27 -> R2.14
; X1.75 is TD0
; R2.15 -> X1.75
X2.68 -> R2.18
X2.66 -> R2.20
X2.62 -> R2.24
X1.14 -> R2.7
X1.62 -> R2.70

X2.84 -> R2.25
X2.4 -> R2.26
X2.60 -> R2.27
X2.25 -> R2.29
X2.49 -> R2.35
X2.41 -> R2.41
X1.72 -> R2.71
X2.71 -> R2.72
X1.60 -> R2.77
;
X2.19 -> R2.79
X1.80 -> R2.80
X1.19 -> R2.8
X1.23 -> R2.9
X1.37 -> R2.5
X1.39 -> R2.3
X1.4 -> R2.4
X1.45 -> R2.83
X1.47 -> R2.81
X1.6 -> R2.6
X1.66 -> R2.78
X1.68 -> R2.69
X1.82 -> R2.82
X1.84 -> R2.84
;R2.75 -> X2.15 TD0 cannot be used okay ???
X2.6 -> R2.28
X2.80 -> R2.19
X2.82 -> R2.23

;
X2.23 -> R2.68
```

8. Using the Protoboard and Schematic Drawings

8.1 Proto-area, Common Anode LEDs

The proto-area is on the left-hand-side of the protoboard. Each I/O pad of the XC4000 FPGAs can only supply 3 mA of current, which is not sufficient to drive most LEDs. The author is certainly aware of the availability of the miniature HP 2 mA LEDs, unfortunately, they are not available as 7-segment displays. Therefore, the 7-segment LEDs are common ANODE LEDs, with headers J48 and J49 providing the access to the segments.

None of the LEDs are connected to the FPGAs, so you need to use jumpers/wires (with sockets on both end) to display your signals. Each segment (in general each LED in the proto-area) can source roughly 4mA to a maximum of 10mA. Header J45 provides the connections to the 4-bar LED4 and LED5 which are also common anode LEDs. SW6 and SW7 are connected to header J46 and J47 respectively; each position is pulled high with a 10K resistor. The header supplies a '1' when the switch is open, and a '0' otherwise.

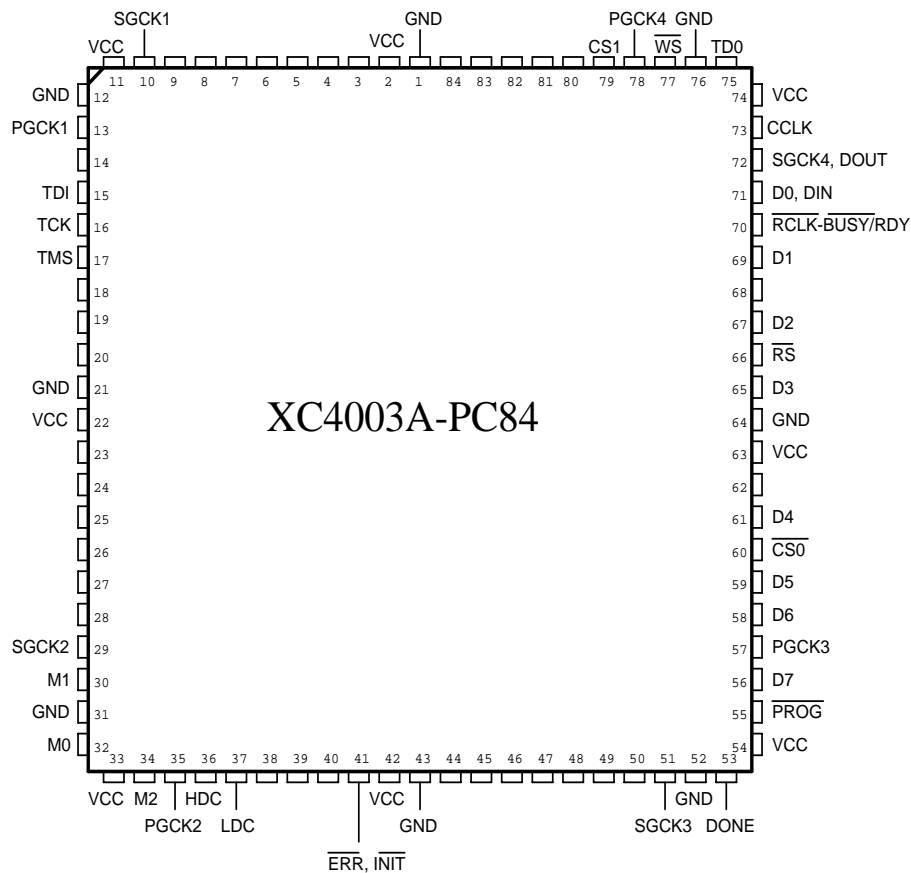


Figure 8.1: XC400?A-PC84 package footprint.

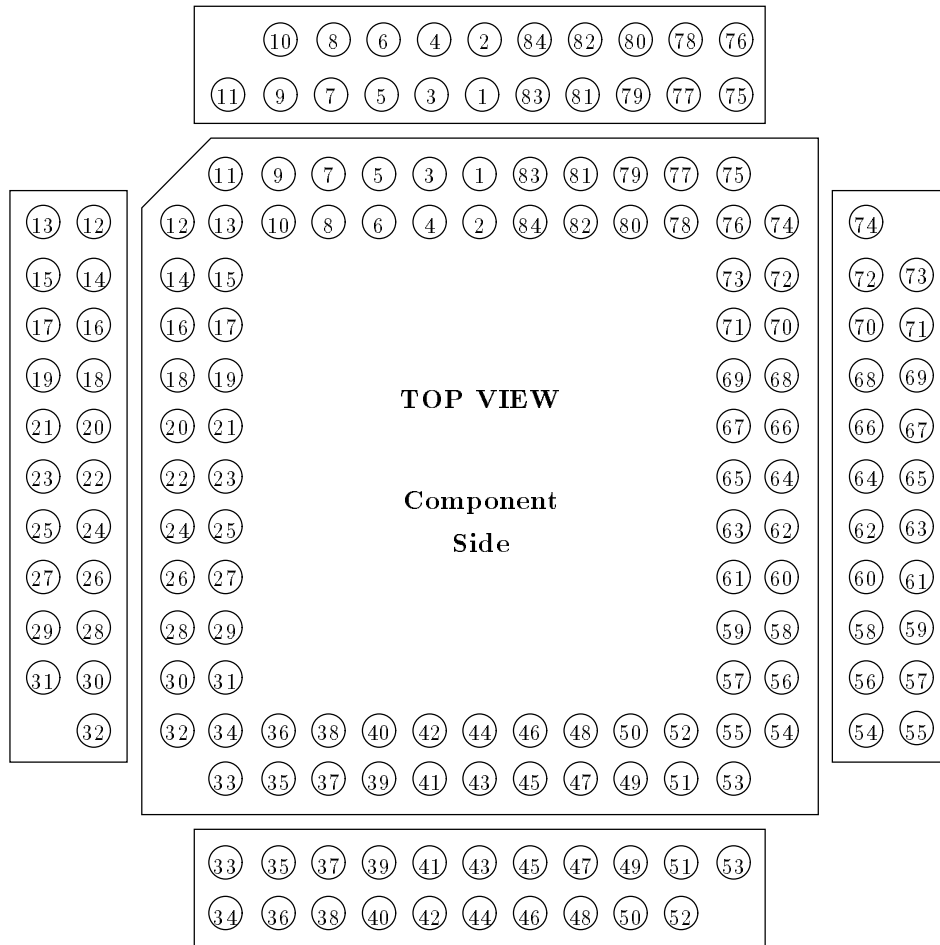


Figure 8.2: 84-pin PLCC Package Footprint and headers, Component Side.

For debugging purposes all the user FPGA pins are connected to the headers for easy signal access. Figures 8.1 and 8.2 provide the 84-pin PLCC footprints and its headers for the “component” side. The assembly drawing of the BORG board with all the reference designators are given in Fig. 8.3. Finally, two sheets of the schematic drawings (drawn with PADS LOGIC) of the BORG board are given in Fig. 8.4 and 8.5 for documentation and debugging purposes.

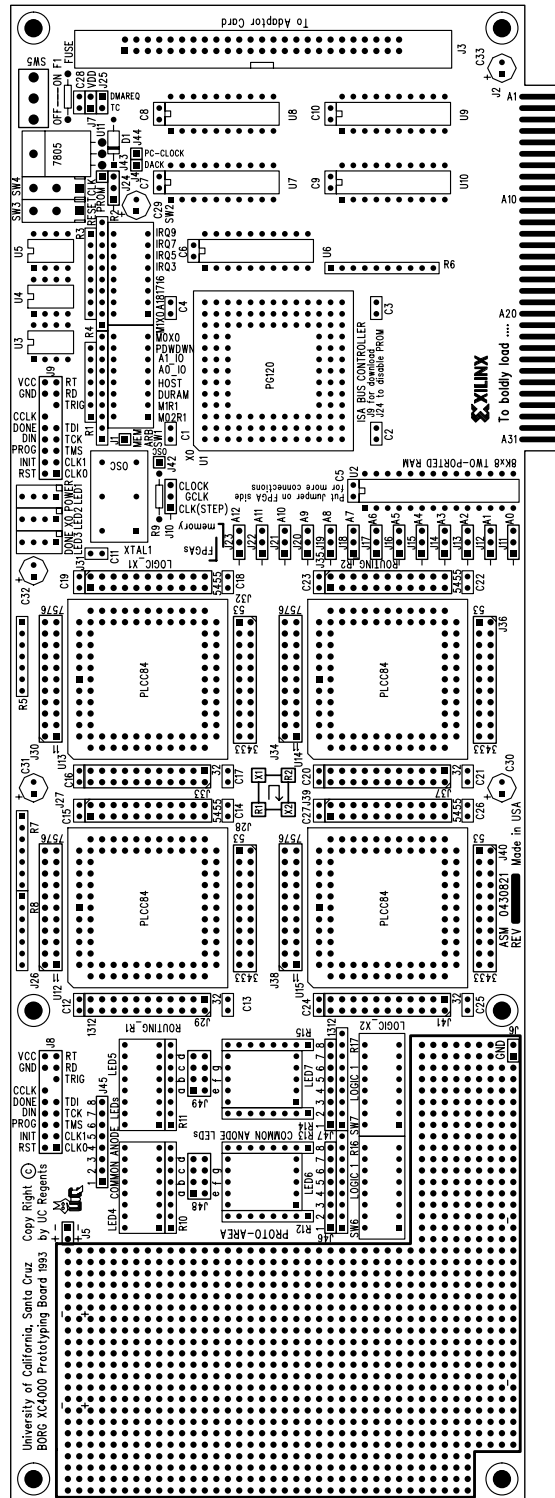


Figure 8.3: The BORG board's assembly drawing with reference designators.

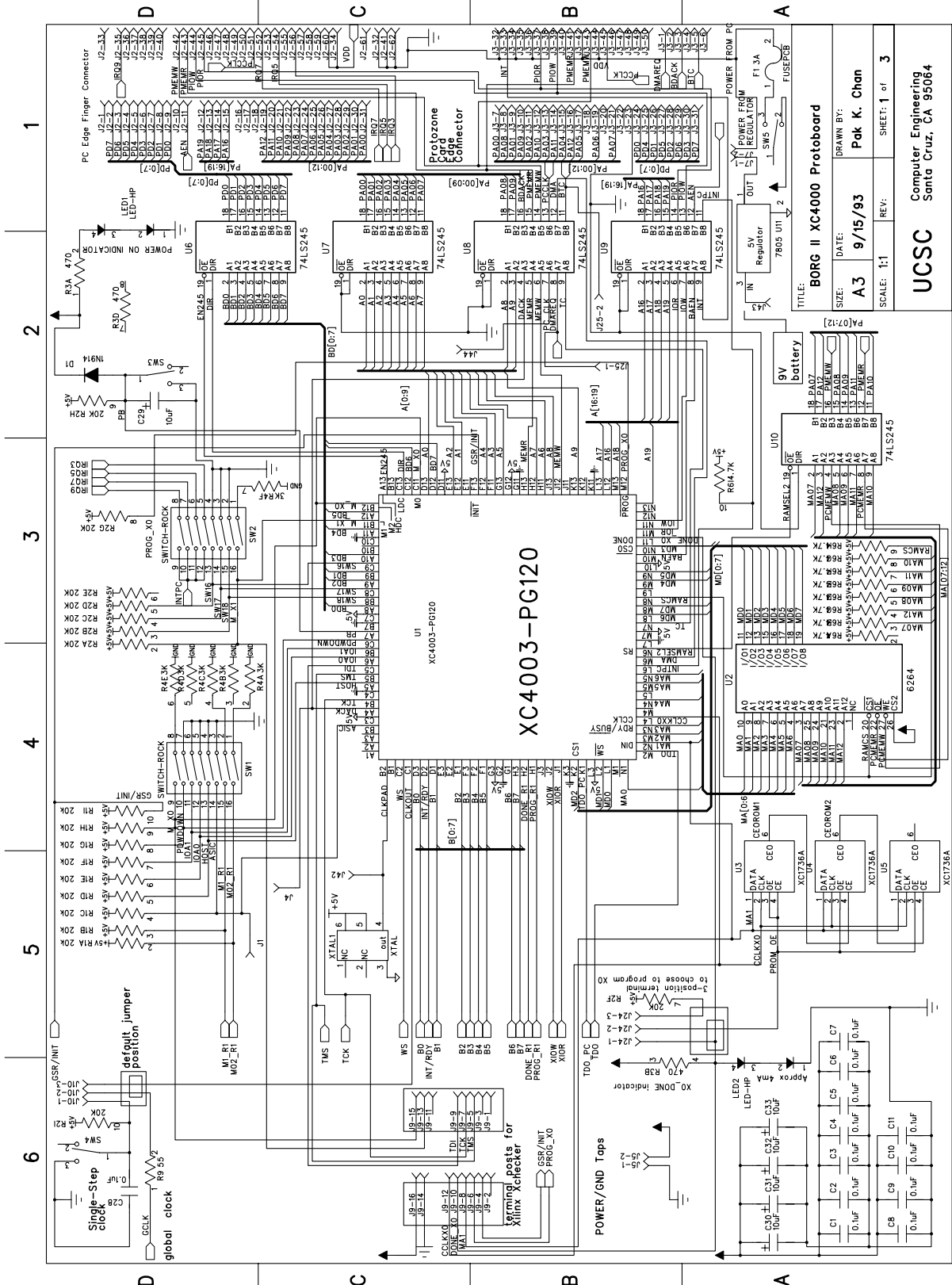


Figure 8.4: Schematic Drawing of the BORG Board (Sheet 1/2).

8.1. Proto-area, Common Anode LEDs

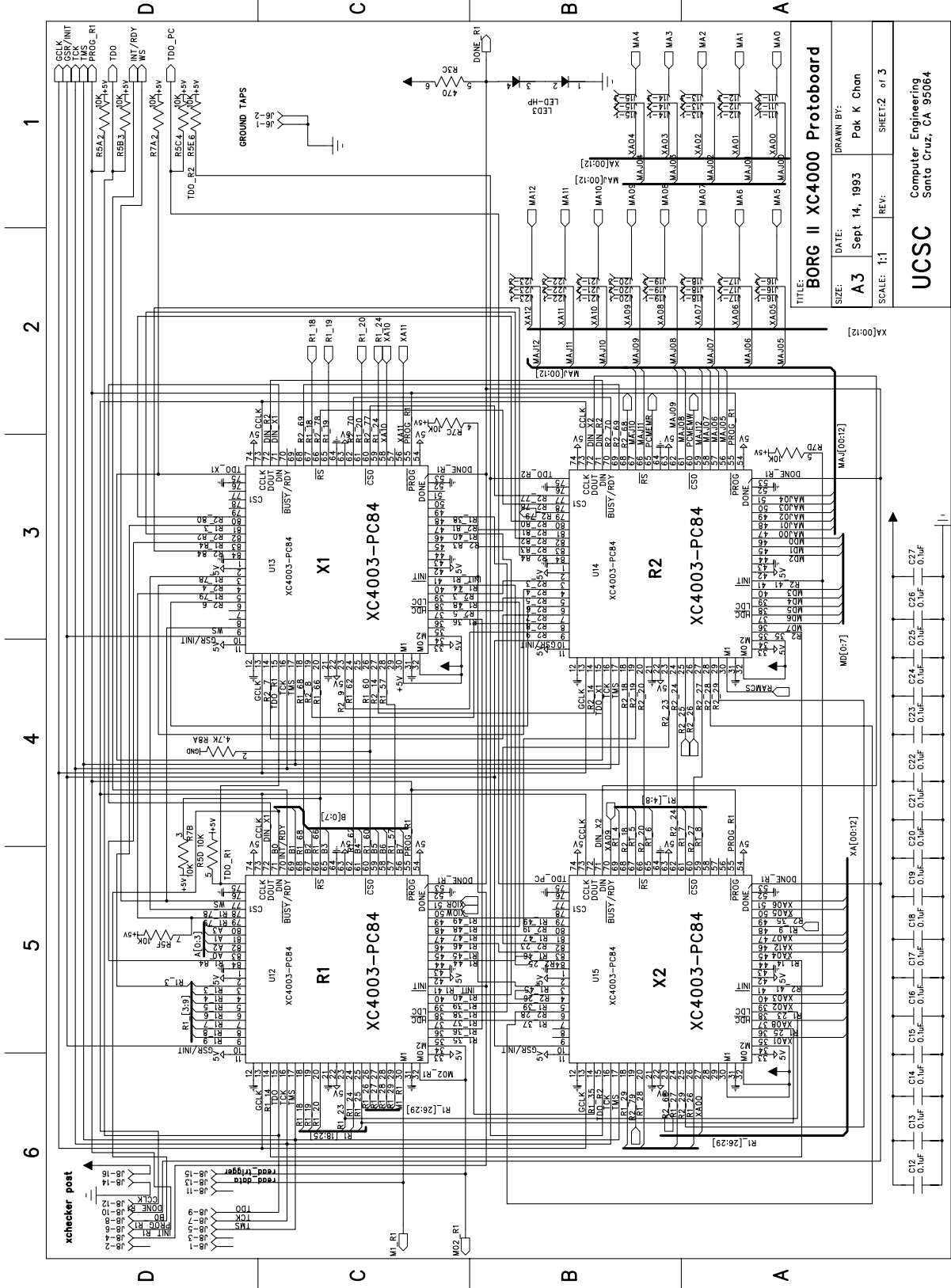


Figure 8.5: Schematic Drawing of the BORG Board (Sheet 2/2).

9. Guide to Some Laboratory Experiments

This chapter serves as a simple guide to use the BORG board. Suggestions for some possible digital design experiments are provided but not elaborated.

9.1 Creating user I/O ports in R1

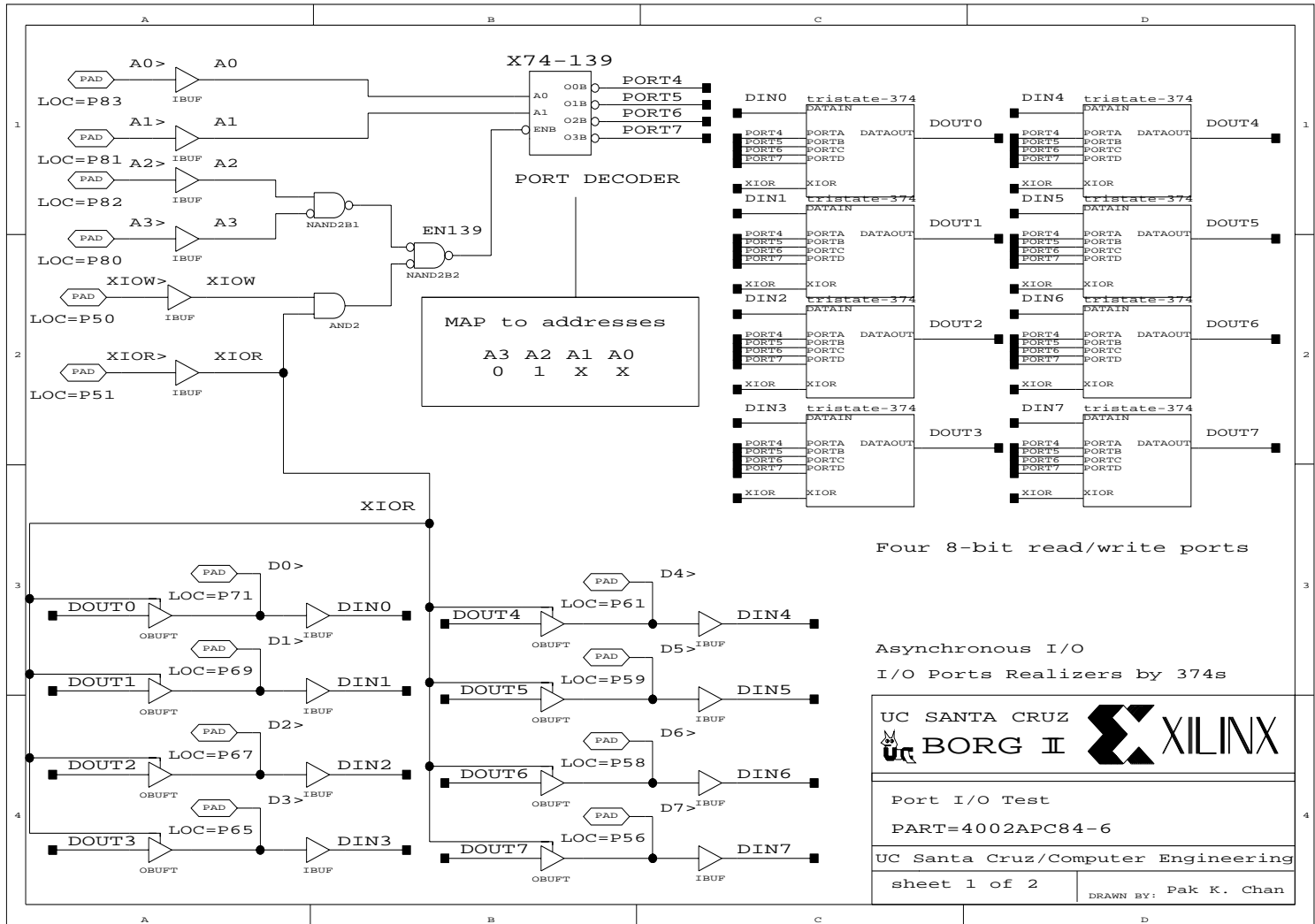
Two sheets of schematic drawings `portest` given in Fig. 9.1 and 9.2 provide the basic idea of implementing I/O ports in **R1** to communicate with the PC. We are creating four user I/O ports. We use a 74139-like part from the Xilinx library for port address decoding. Notice that the outputs of the decoder are active LOW, and the selected output is used to clock the 74374-like (positive edge-triggered) octal registers. The outputs of the octal registers share an 8-bit bus which is tri-stated. The signal `XI0R` is used to control the direction of data on the bus. Note that the I/O pad assignments are provided directly in the schematic in Fig. 9.1.

A simple program `portest.c` which writes and then reads from the I/O ports created in R1 FPGA is given on the next page.

Given that the schematic drawing's file name is `portasy`, you can download this port test demo by taking the following steps:

```
c:> wir2xnf portasy
c:> ppr portasy
c:> makebits portasy
c:> makeprom -o portest.mcs -u 0 portasy.bit em4003a em4002a em4003a
c:> bd portest.mcs
c:> portest
```

Figure 9.1: Building I/O ports in the R1 FPGA.



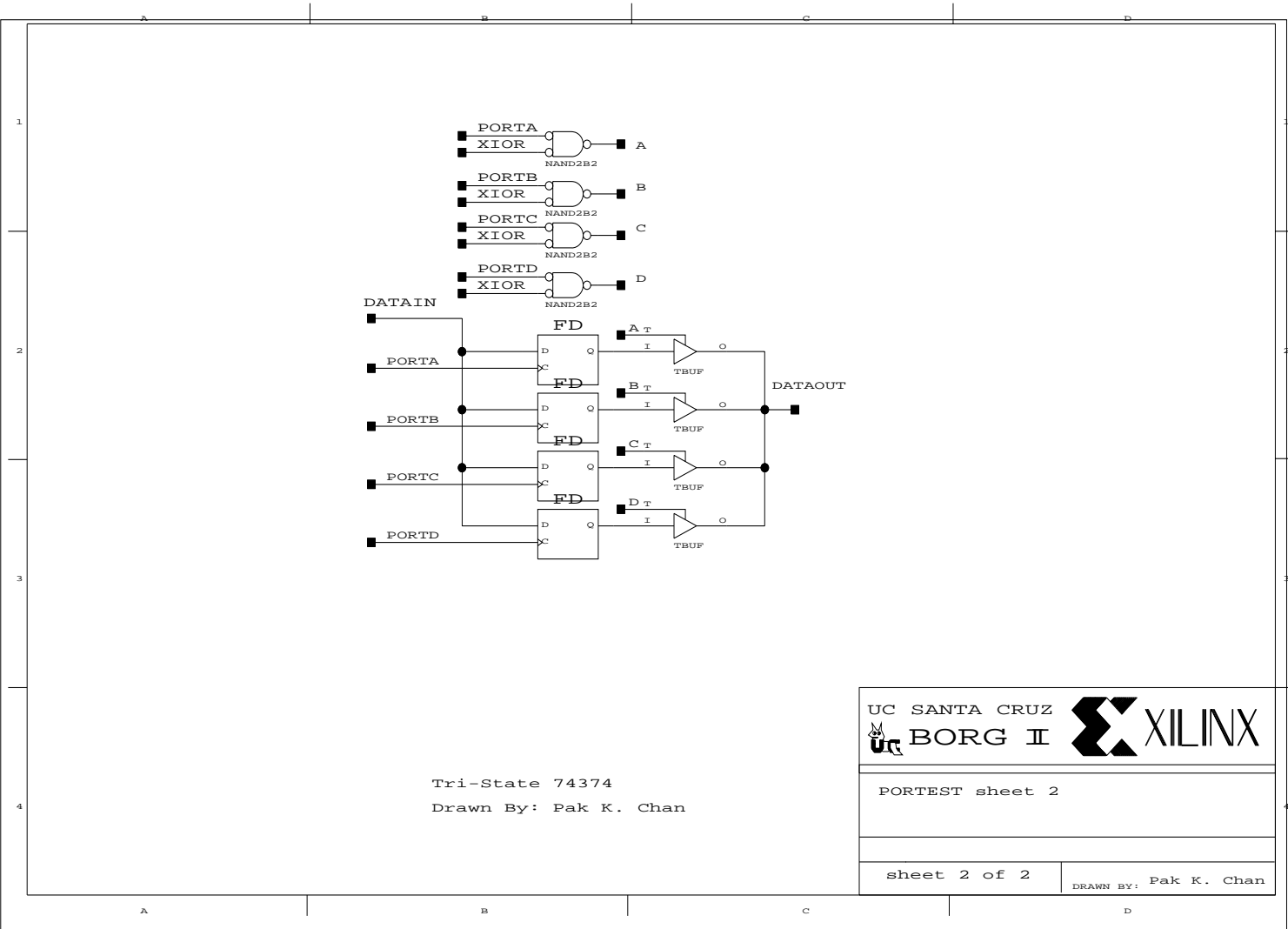


Figure 9.2: Tri-state 74374-like device in the PORTEST schematic.

```

/* portest: write and then read four PORTs in R1*/
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

main ()
{
  unsigned int PORT1, PORT2, PORT3, PORT4;
  unsigned int PortA;

  int i, j;
  float error;
  unsigned char x;
  char * portenv;

  error=0;
  setcbrk(1);

  portenv=getenv("BORG");
  /* Control Port in X0 */
  if(!strcmp(portenv,"0x300"))
    PortA = 0x300;
  else if(!strcmp(portenv,"0x200"))
    PortA = 0x200;
  else if(!strcmp(portenv,"0x210"))
    PortA = 0x210;
  else if(!strcmp(portenv,"0x310"))
    PortA = 0x310;
  else {
    printf(" \n Wrong PORT address\n");
    printf(" Please specify PORT address\n e.g. set BORG=0x300%s\n");
    exit(1);
  }
  printf(" BORG PORT address is %s\n",portenv);

  PORT1=PortA+4;
  PORT2=PortA+5;
  PORT3=PortA+6;
  PORT4=PortA+7;

  for (i =0; i < 15; i++)
  {outportb (PORT1,i);
  outportb (PORT2,i+1);
  outportb (PORT3,i+2);
  outportb (PORT4,i+3);
  printf ("Sent to port   Data   Read Data\n");
  printf ("  PORT1:      %d      %d \n",i,inportb (PORT1));
  printf ("  PORT2:      %d      %d \n",i+1,inportb (PORT2));
  printf ("  PORT3:      %d      %d \n",i+2,inportb (PORT3));
  printf ("  PORT4:      %d      %d \n",i+3,inportb (PORT4));
  printf ("(hit return to continue ..)");
  getchar ();
  }
  printf ("Starting automatic check (read after write)...\n");
  printf ("This will take a minute or so ....\n");

  for (j =0; j < 10000; j++)
  for (i =0; i < 127; i++)
  {outportb (PORT1,i);
  x=inportb (PORT1);
  if( x != i ) ++error;

  outportb (PORT2,i+1);
  x=inportb (PORT2);
  if( x != i+1 ) ++error;

```

```

        outportb (PORT3,i+2);
        x=inportb (PORT3);
        if( x != i+2 ) ++error;

        outportb (PORT4,i+3);
        x=inportb (PORT4);
        if( x != i+3 ) ++error;
    }
    printf ("Total errors %6.0f\n",error);
}

```

9.2 Hardware Interrupt and Interrupt Service Routine

This experiment will illustrate the hardware interrupt feature supported by the BORG board.

The interrupt service routine is called `isr.c`. It indicates that it is serving a hardware interrupt by beeping the PC's speaker. This interrupt service routine counts the number of times that it has been interrupted. It services 10 interrupts and then removes itself. This interrupt service routine is loaded as a memory-resident program, as documented in the code.

The schematic drawing that generates the hardware interrupt (from the BORG board) is `intpc.1`, which is essentially an I/O address decoder connected to a toggle flip-flop. The flip-flop toggles the interrupt request line every time that a predefined I/O address is selected. Now, enable `IRQ9` on your board for this demo.

To load the interrupt generator `intpc`, you do:

```

c:> wir2xnf intpc
c:> ppr intpc
c:> makebits intpc
c:> makeprom -o intpc.mcs -u 0 intpc.bit em4003a em4002a em4003a
c:> bd intpc.mcs
c:> isr

```

We use a small program `intpc.c` which activates the toggle flip-flop to demonstrate the hardware interrupt generation and service processes.

```

c:> intpc
BORG PORT address is 0x300

```

Make sure that you load ISR `isr.com` first.

```

Board Board interrupts PC.
ISR will ring the speaker 10 times.
1 (hit return to continue ..)
2 (hit return to continue ..)
3 (hit return to continue ..)

```

```

#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

main () /* Interrupt PC demo requires schematic drawing INTPC */
{
    unsigned int PORT1, PortA;

    int i, j;
    unsigned char x;
    char * portenv;

    setcbrk(1);

    portenv=getenv("BORG");
    /* Control Port in X0 */
    if(!strcmp(portenv,"0x300"))
        PortA = 0x300;
    else if(!strcmp(portenv,"0x200"))
        PortA = 0x200;
    else if(!strcmp(portenv,"0x210"))
        PortA = 0x210;
    else if(!strcmp(portenv,"0x310"))
        PortA = 0x310;
    else {
        printf(" \n Wrong PORT address\n");
        printf(" Please specify PORT address\n e.g. set BORG=0x300%s\n");
        exit(1);
    }
    printf(" BORG PORT address is %s\n",portenv);

    PORT1=PortA+4;
    printf ("\n Make sure that you load ISR isr.com first.\n");
    printf ("\n Board Board interrupts PC.\n ISR will ring the speaker 10 times.\n");

    for (i = 1; i < 15; i++)
    {
        outportb (PORT1,i); /* toggle the flip-flop inside R1 */
        delay(1);
        outportb (PORT1,i);
        printf (" %2d (hit return to continue ..)", i);
        getchar ();
    }
}

```

```

/* Interrupt Service Routine isr.c
   Modified from and credit to the Protozone User's manual

   A simple interrupt handler example using C without assembly language.
   Code in Borland C.
   This program assumes IRQ9 is used and shows how to handle
   the slave and master Programmable Interrupt Controllers 8259As (PICs)
   We need to take care of both the PICs because IRQ9 is cascaded thru IRQ2.
   The interrupt vector for IRQ2 is 0x0A as defined by the PC

   Note: You need to pull IRQ9 low in order to run this program properly
*/
/*
Compile and execute isr.com with
   tcc -mt -M isr.c
   exe2bin isr.exe isr.com
*/

#include <dos.h>
#include <conio.h>
#include <stdio.h>

#define PIC_master    0x20 /* Programmable Interrupt Controller PIC master */
#define PIC_slave    0xA0 /* Programmable Interrupt Controller PIC slave */

#define EOI          0x20 /* end of interrupt code to send to PICs */
#define IRQ2_mask    0xFB /* interrupt mask to enable interrupt request 2
                           bit 2 is reset */
#define IRQ9_mask    0xFD /* interrupt mask to enable interrupt request 9
                           bit '9' is reset */
#define IRQ9         0x0A /* interrupt number */

#define TIMES      10

void IntRemove();
void interrupt (*oldVector)();
unsigned char oldMask1, oldMask2;
void Install();

void interrupt mybeep(unsigned bp, unsigned di, unsigned si,
                    unsigned ds, unsigned es, unsigned dx,
                    unsigned cx, unsigned bx, unsigned ax)
{
    int i,j;
    static count=0;
    char originalbits, bits;
    unsigned char bcount;

    /* get the current control port of the PIC setting */

    disable();
    /* port for speaker */
    bits = originalbits = inportb(0x61);

    bcount=500;
    for(i=0;i<=bcount; i++){
        outportb(0x61, bits & 0xfc);
        for(j=0;j<=300; j++){
            outportb(0x61, bits | 2);
            for(j=0;j<=200; j++){

```

```

    outport(PIC_slave, EOI);
    if(++count) >= TIMES) IntRemove();
    enable();
}

void Install(faddr, inum)
void interrupt (* faddr)();
int inum;
{
    disable();
    oldVector = getvect(inum);
    setvect(inum, faddr);
    oldMask1 = inportb(PIC_master +1);
    oldMask2 = inportb(PIC_slave +1);
    outportb(PIC_master+1, IRQ2_mask & oldMask1);
    outportb(PIC_slave +1, IRQ9_mask & oldMask2);

    printf("Interrupt Handler installed.\n\n");
    printf("This interrupt handler intercepts 10 interrupts\nand then remove itself.\n");
    enable();
}

void IntRemove()
{
    disable();
    setvect(IRQ9, oldVector);
    outportb(PIC_master+1, oldMask1);
    outportb(PIC_slave +1, oldMask2);
    enable();
    oldVector();
}

main()
{
    char ch;
    Install(mybeep,IRQ9);
    /* check with isr.map
    when compile with
    tcc -mt -M isr.c
    to generate a memory map
        Start Stop Length Name Class
        00000H 01594H 01595H _TEXT CODE
        015A0H 019BBH 0041CH _DATA DATA
        019BCH 019BFH 00004H _EMUSEG DATA
        019C0H 019C1H 00002H _CRTSEG DATA
        019C2H 019C3H 00002H _CVTSEG DATA
        019C4H 019C9H 00006H _SCMSEG DATA
        019CAH 01A15H 0004CH _BSS BSS
        01A16H 01A16H 00000H _BSSEND STACK
    */
    keep(0, 0x01C0); /* make the interrupt handler resident */
}

```

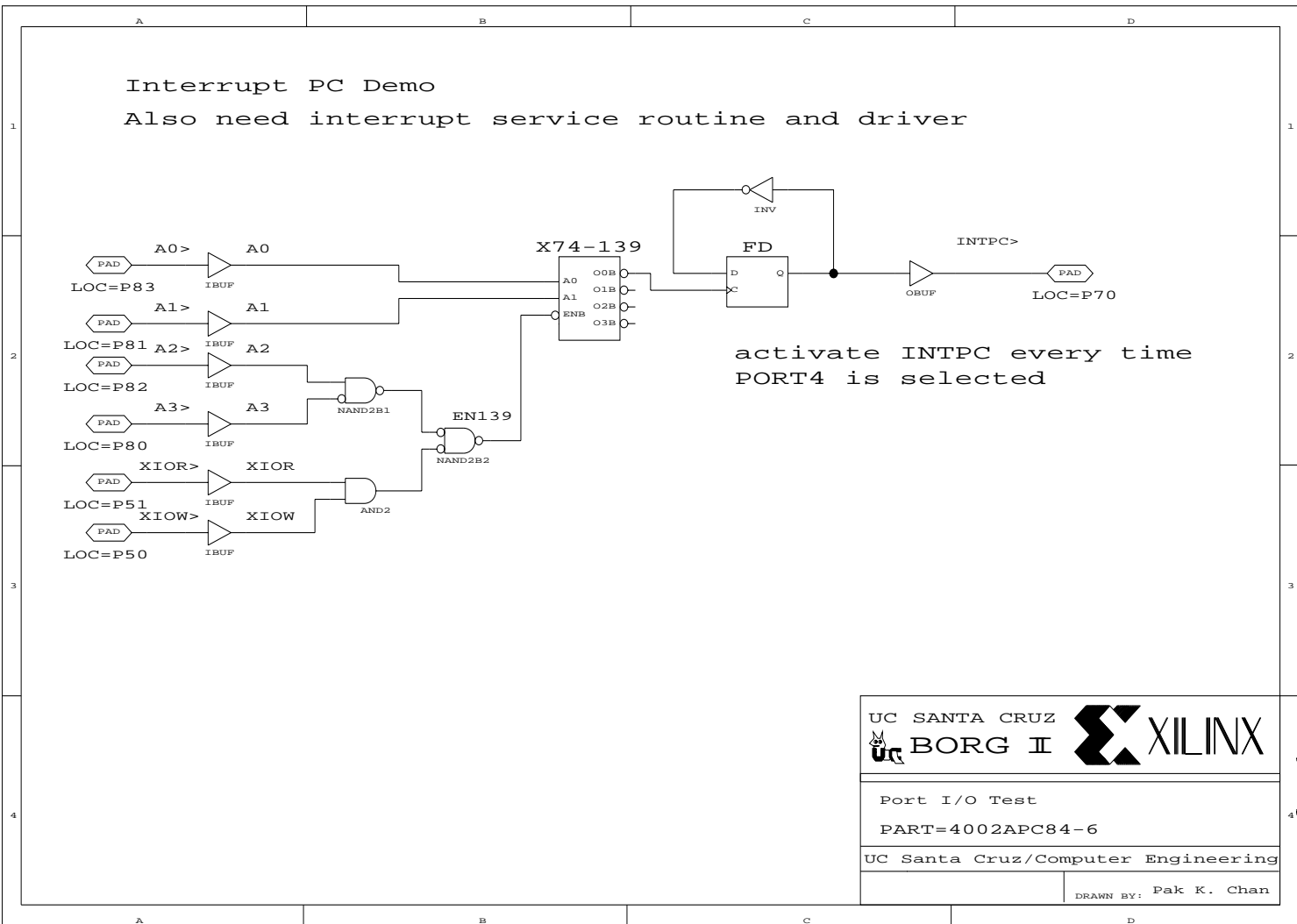


Figure 9.3: Hardware Interrupt Demo. Interrupt Generated by logic in the R1 FPGA.

9.3 Synchronization Problem

The PC and the BORG board are driven by different clocks. You need to synchronize any information transfer between them to avoid any timing problems. Particularly when you have sequential logic (such as a finite state machine) inside the R1 FPGA, the data transfer from the PC to your sequential logic *must* be synchronized by synchronization registers using the (not the PC) system clock.

Lab: The schematic drawing as shown in Fig. 9.4 has an I/O port located at address `PORT4`. The output of this port feeds two D flip-flops, `FFONE` and `FFTWO`. These two D flip-flops are clocked by the system clock, and these D flip-flops are constrained to be mapped into different CLBs (just to exaggerate the failure rate, you can put them together in the same CLB if you want). The counter registers the number of times that the output of the flip-flops are different.

Questions: What causes the outputs of the D flip-flops to be different? How would you fix the problem?

To load this lab `asylab`, you do:

```
c:> wir2xnf asylab
c:> ppr asylab
c:> makebits asylab
c:> makeprom -o asylab.mcs -u 0 asylab.bit em4003a em4002a em4003a
c:> bd asylab.mcs
c:> asylab
```

A sample driver for this lab is included on the next page.


```

/*****
/* asytab v1.0                               April 5,1994*/
/*****
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>

int main(int argc,char *argv[])
{
    unsigned char loc, oldloc;
    int wait;
    char * portenv;
    unsigned int PORTRESET, PORT4;
    unsigned PortA;

    setcbrk(1);
    portenv = getenv("BORG");

    /* Control Port in IO */
    if(!strcmp(portenv,"0x300"))
    PortA = 0x300;
    else if(!strcmp(portenv,"0x200"))
        PortA = 0x200;
    else if(!strcmp(portenv,"0x210"))
        PortA = 0x210;
    else if(!strcmp(portenv,"0x310"))
        PortA = 0x310;
    else {
        printf(" Wrong PORT address\n");
        printf(" Please specify PORT address\n e.g. set BORG=0x300%s\n");
        exit(1);
    }
    PORTRESET = PortA + 3;
    PORT4 = PortA + 4;

    /* reset the machine */
    outportb(PORTRESET, 0x00);
    outportb(PORTRESET, 0x01);
    delay(1);
    /* read Port 4 until all zeroes */
    wait = 10;
    while((loc=inportb(PORT4)) != 0 && wait !=0)
    {wait--; delay(1);
    printf("Waiting for counter to reset.\n"); }
    /* stop reading */
    if(loc!=0) {printf("Counter in R1 never reset.\n");
    }
    else
    {printf("Counter in R1 set to 0.\n"); oldloc = -1;
    while(1){
        outportb(PORT4, 0x01);
        delay(1);
        loc=inportb(PORT4);
        if(loc != oldloc) {
            printf("Counter --> %d \n",loc); oldloc=loc;
        }
        outportb(PORT4, 0x00);
        delay(1);
        loc=inportb(PORT4);
        if(loc != oldloc) {
            printf("Counter --> %d \n",loc); oldloc=loc;
        }
    }
    }
}

```

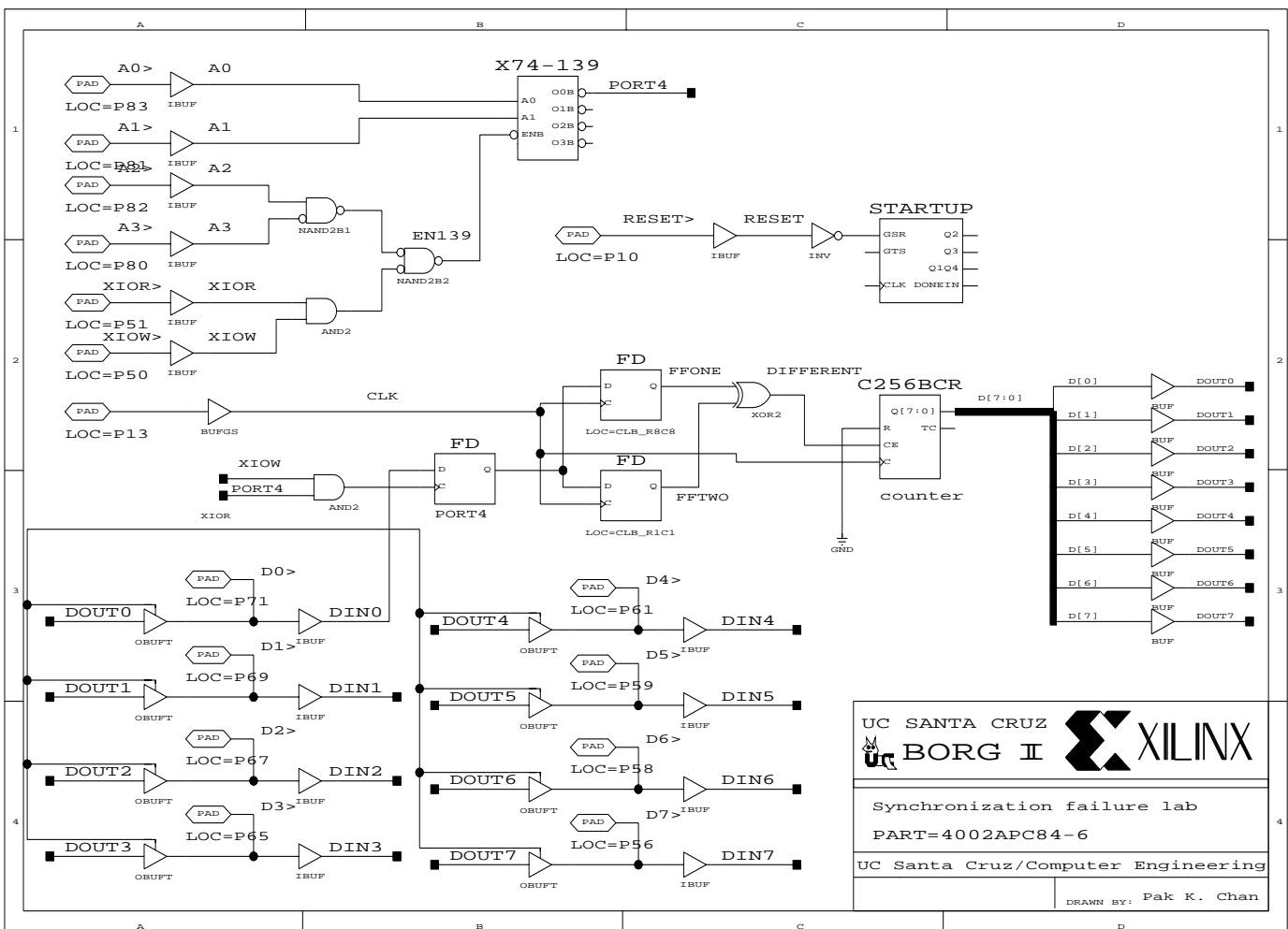


Figure 9.4: Synchronization failure lab. Design in R1 FPGA.

9.4 Music Lab

This frequency synthesizer lab demonstrates the use of XC4000 CLBs as Read-Only-Memories (ROMs). You will also need the following additional components to appreciate this lab.¹

1. one digital-to-analog converter part # TI TLC5602CN
2. one 2N2219A NPN transistor, one 2N2222 NPN transistor
3. some resistors
4. three 10 μ F capacitors
5. one potentiometer
6. an 8 Ω speaker

The DAC yields only one volt dynamic range, so we use some discrete components to build a simple two-stage transistor amplifier with a voltage gain of 2, as shown in Fig. 9.5. You can replace this part with a higher quality amplifier.

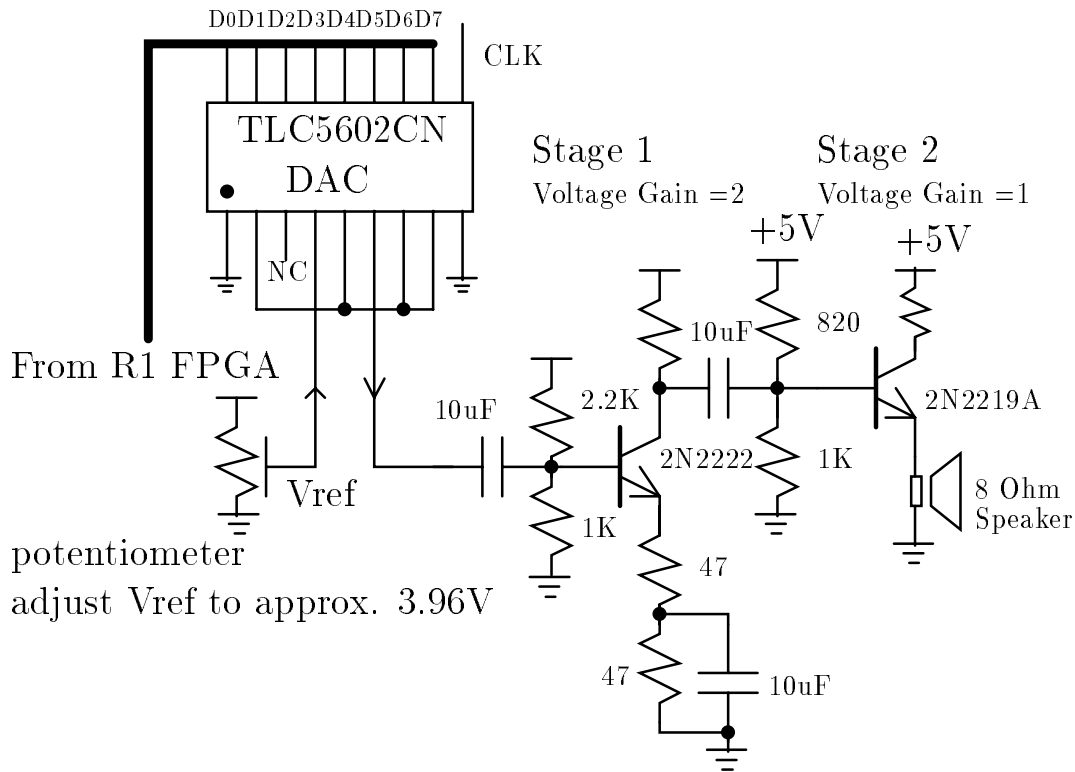


Figure 9.5: Digital-to-Analog Converter and a two-stage transistor amplifier for the “music” lab.

As illustrated in Fig. 9.5, the DAC is used to convert the digital output of the **R1** FPGA to an analog (sine-wave like) signal. The transistors and the rest of the discrete components form a simple two-stage amplifier to drive a small 8 Ω speaker.

¹Credit to Tom W. Geocaris.

Referring to the schematic drawings as shown in Figs. 9.6 and 9.7, the module ROM64W stores a (low fidelity) discretized “sine” wave. The content of the module is initialized by using the Xilinx `memgen` utility on the data file `rom64w.mem`.

```
; =====
; rom64w.mem: A 64-word deep by 8-bit wide ROM memory.
; =====
;
TYPE ROM      ; The memory is a ROM
DEPTH 64      ; The memory is 64 words deep
WIDTH 8       ; Each memory word is 8 bits wide
SYMBOL VIEWLOGIC PINS ; Build a VIEWLOGIC symbol with pin inputs
DATA 10#128#,
10#140#,
10#153#,
10#165#,
10#177#,
10#188#,
10#199#,
10#209#,
10#218#,
10#226#,
10#234#,
10#240#,
10#245#,
10#250#,
10#253#,
10#254#,
10#255#,
10#254#,
10#253#,
10#250#,
10#245#,
10#240#,
10#234#,
10#226#,
10#218#,
10#209#,
10#199#,
10#188#,
10#177#,
10#165#,
10#153#,
10#140#,
10#128#,
10#116#,
10#103#,
10#91#,
10#79#,
10#68#,
10#57#,
10#47#,
10#38#,
10#30#,
10#22#,
10#16#,
10#11#,
```

```

10#6#,
10#3#,
10#2#,
10#1#,
10#2#,
10#3#,
10#6#,
10#11#,
10#16#,
10#22#,
10#30#,
10#38#,
10#47#,
10#57#,
10#68#,
10#79#,
10#91#,
10#103#,
10#116#

```

A 16-bit binary counter CNT16 is used to scan the ROM64W module at different rates to produce sine waves of different frequencies. The scan rate is loadable from the PC's keyboard via two I/O ports located at the **R1** FPGA.

To load this lab `synth`, you do:

```

c:> wir2xnf synth
c:> xnfmerge synth music
c:> ppr music
c:> makebits music
c:> makeprom -o music.mcs -u 0 music.bit em4003a em4002a em4003a
c:> bd music.mcs
c:> music

```

Use the PC's keyboard to change the frequency of the sound! A very primitive driver is included for the purpose of illustration.

```

#include <stdio.h>
#include <math.h>
#define PORT1 0x304
#define PORT2 0x305
#define CLK      8000000
#define BUF_SIZE 64
#define CTRL_C   0x3

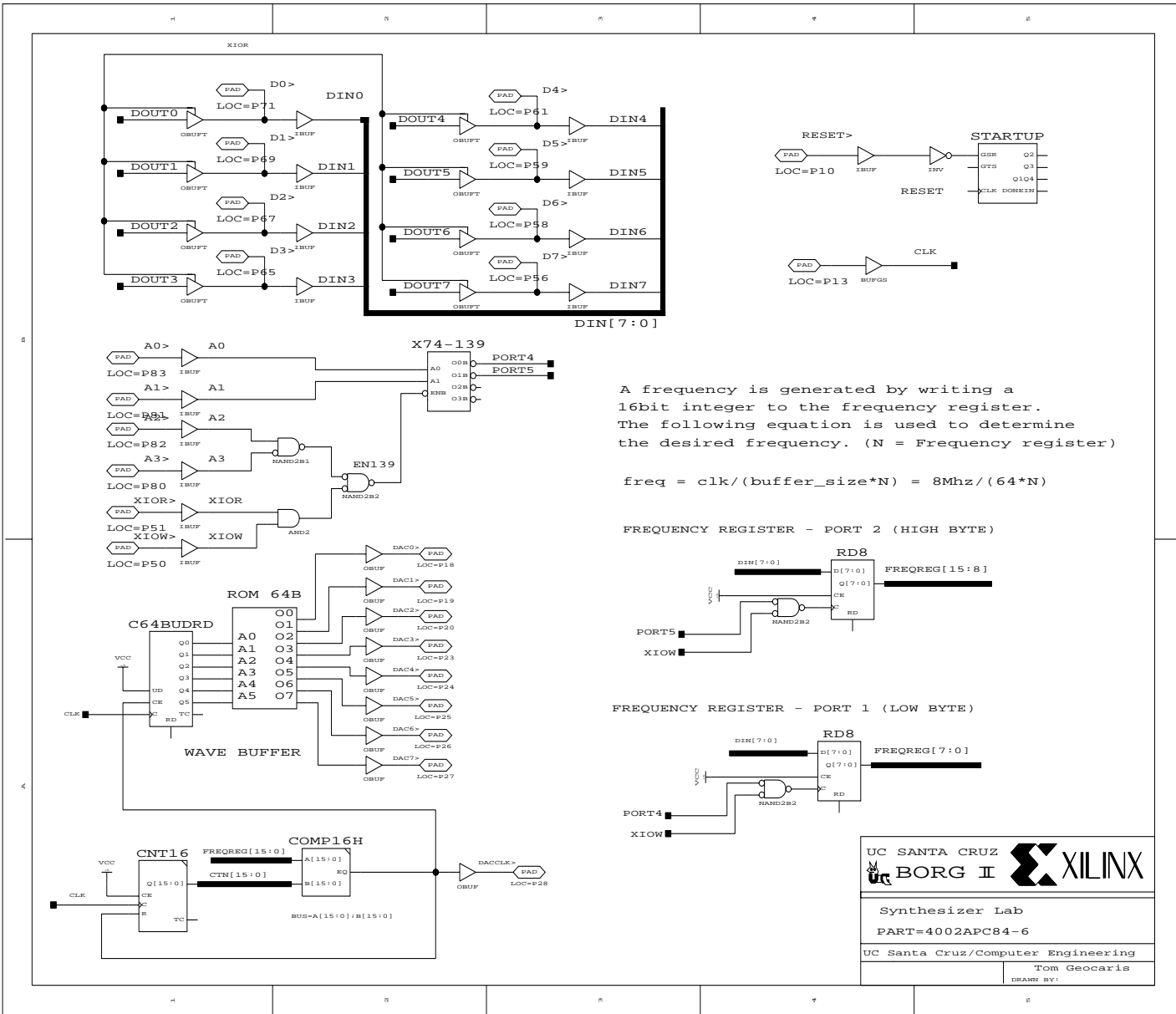
main( int argc, char **argv )
{
    unsigned int n;
    int i;
    char buf[128];

    while ( 1 ) {
        switch ( getch() )
        {

```

```
        case 'q':
            i = 0;
            break;
        case 'w':
            i = 1;
            break;
        case 'e':
            i = 2;
            break;
        case 'r':
            i = 3;
            break;
        case 't':
            i = 4;
            break;
        case 'y':
            i = 5;
            break;
        case 'u':
            i = 6;
            break;
        case 'i':
            i = 7;
            break;
        case 'o':
            i = 8;
            break;
        case 'p':
            i = 9;
            break;
        case '[':
            i = 10;
            break;
        case ']':
            i = 11;
            break;
        case CTRL_C:
            exit(1);
        default:
            continue;
    }
    n=floor(CLK/BUF_SIZE/(440.0*pow(1.0594631,i))+0.5);
    outportb( PORT1, n & 0xff );
    outportb( PORT2, (n & 0xff00) >> 8 );
}
}
```

Figure 9.6: Frequency Synthesizer Lab. Design in R1 FPGA (Sheet 1/2).



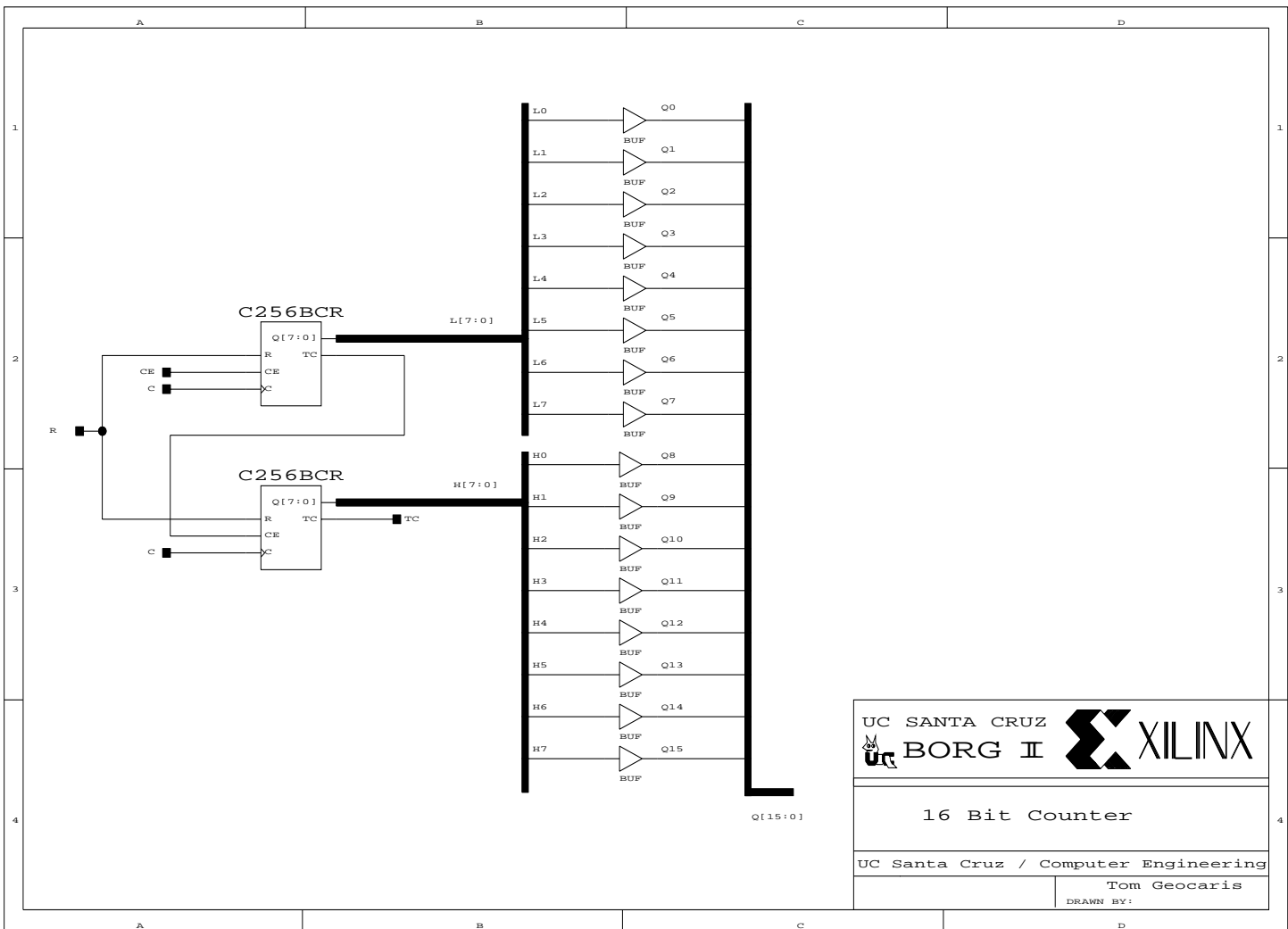


Figure 9.7: Frequency Synthesizer Lab. Design in R1 FPGA (Sheet 2/2).

9.5 DMA Lab

This lab demonstrates transferring data from the BORG board to the PC's memory using DMA. This lab illustrates the steps involved in

- programming the Intel 8237A-5 DMA controller on the PC,
- initiating the DMA request and transfer with the X0 controller of the BORG board

The data to be transferred are generated by a counter in the R1 FPGA on the BORG board. The first data byte has the value 1, the second byte is 2, and the last byte is 256.

You need:

1. a Protozone adapter card, set dip switch positions 6 and 5 of SW1 to ON to enable DMA channel 3,
2. make sure that the 74LS367A TTL on the protozone board is not excessively noisy, use an oscilloscope to observe the terminal count signal (TC) on the BORG board. I have to hand-pick a good 74LS367A buffer for this lab,
3. set the BORG board to host mode, use position 4 of dip switch SW1 of the BORG board,
4. change the BORG board controller X0 by programming X0 in the slave mode using the Xilinx **xchecker** cable via J9. To set X0 to this mode:
 - (a) shunt jumper J24 on the right side with a plastic jumper,
 - (b) set position M0X0 of dip switch SW1 to open, and
 - (c) set position M1X0 of dip switch SW2 to open.

You can download the supplied bit files `x0dma.bit` and `dmaio.mcs` by using the `xchecker` cable.

```
c:> xchecker x0dma    /* download the X0 controller */
c:> bd dmaio.mcs      /* program the R1 FPGA to
                    generate the data for the transfer */
c:> tst               /* program the 8237A-5 DMA controller */
                    /* initiate the DMA transfer */
```

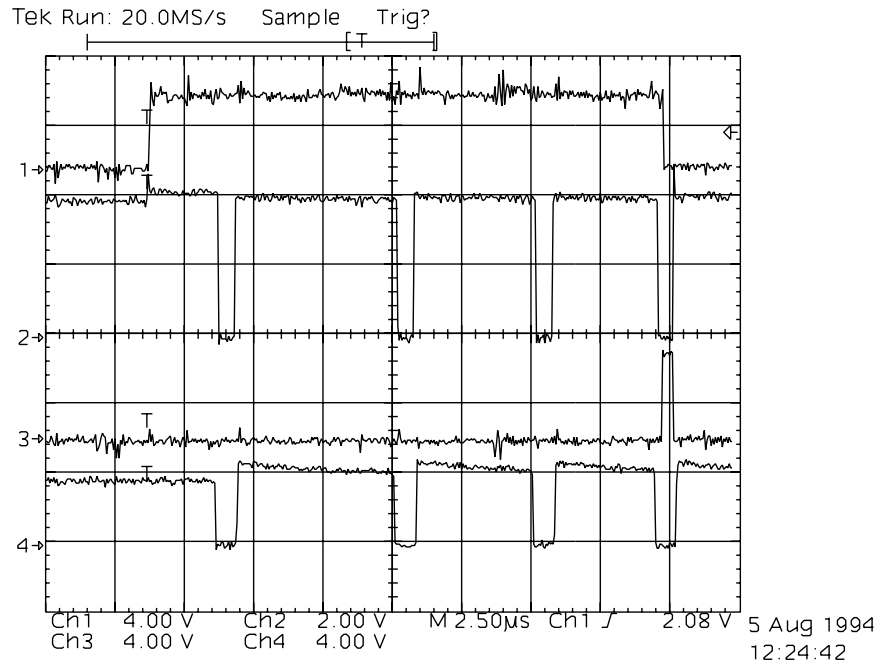


Figure 9.8: DMA transfer timing of four consecutive bytes from the BORG board to PC's memory. Channel 1 is the DMA request **DMAREQ** signal. Channel 2 is the IO read **IOR** signal. Channel 3 is the terminal count **TC** signal on the ISA bus. Channel 4 is the DMA ACK **DACK** signal.

You should see the output:

```
Initialize DMA controller.
Used first half of data area.
Load counters values.
Trigger DMA transfer.
Terminal count now 80.
Terminal count pending 80.
Terminal count has expired 8.
```

```
DMA transfers of 256 bytes completed. Hit to continue.
Used first half of data area.
```

```
0 : 1
16 : 11
32 : 21
48 : 31
64 : 41
80 : 51
96 : 61
112 : 71
128 : 81
144 : 91
160 : a1
176 : b1
192 : c1
208 : d1
```

```

224 : e1
240 : f1

```

It means 256 bytes have been transferred from the R1 FPGA to the PC's main memory using DMA.

Figure 9.8 depicts the DMA transfer timing of four consecutive bytes from the BORG board to PC's memory. Two sheets of schematic drawings `x0dma` and `r1dma` are provided in Figs. 9.9 and 9.10, respectively to illustrate the supporting hardware for this DMA lab.

```

#include <stdio.h>
#include <dos.h>
/*
  DMATST.C: DMA transfer from R1 Xilinx FPGA to PC memory

  Mostly from the protozone manual Prof El Gamal Stanford University
  Modified by Pak K. Chan for the BORG II protoboard 7/15/94

  ** Supporting hardware in the Protozone board: DMA channel 3
  make sure that your PC is not using this channel,
  otherwise this demo may crash your system
  use an oscilloscope to probe the DACK on the borg board
  to be sure

  Supporting hardware in the BORG board:
  X0 with DMA  x0dma.bit
  R1 with DMA  r1.bit
  all four chips with DMA  dmaio.mcs

  Port 0x304 a write triggers DMA transfer from R1's counter to PC's memory

  compile with: tcc -etst dmatst.c dmaini.asm
*/

#define DSIZE  256 /* data set size for DMA transfer */
#define INC 16

/* public variables defined in assembly code dmaini.asm */

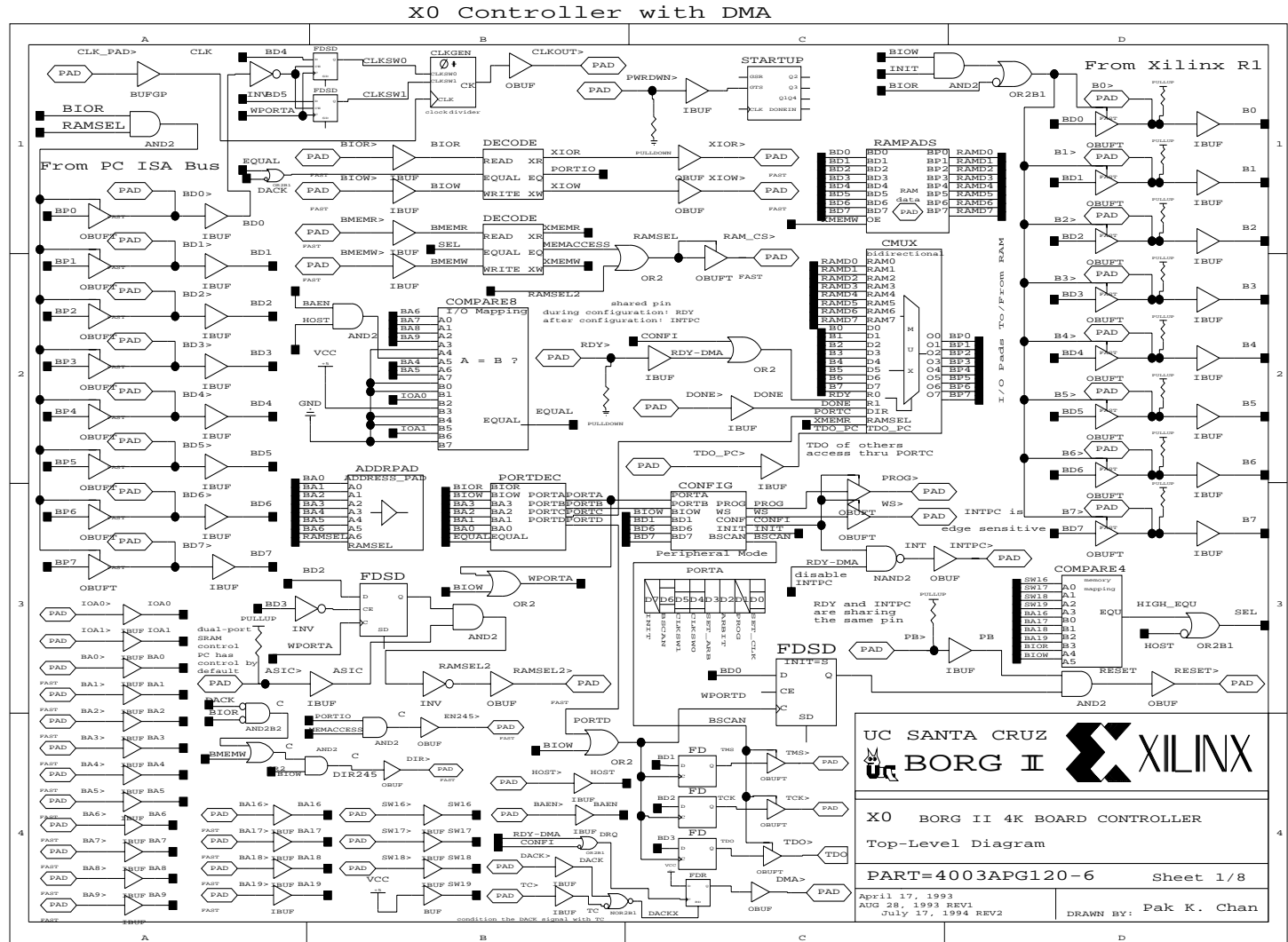
extern unsigned char dbeg[DSIZE];
extern unsigned char dmid[DSIZE];
extern unsigned int usebeg;

#define RESETPORT 0x303
#define DMAPORT  0x304

extern int dmainit();
main()
{
  unsigned int i;
  unsigned char tc;

```

Figure 9.9: DMA Lab. Design in X0 controller FPGA.



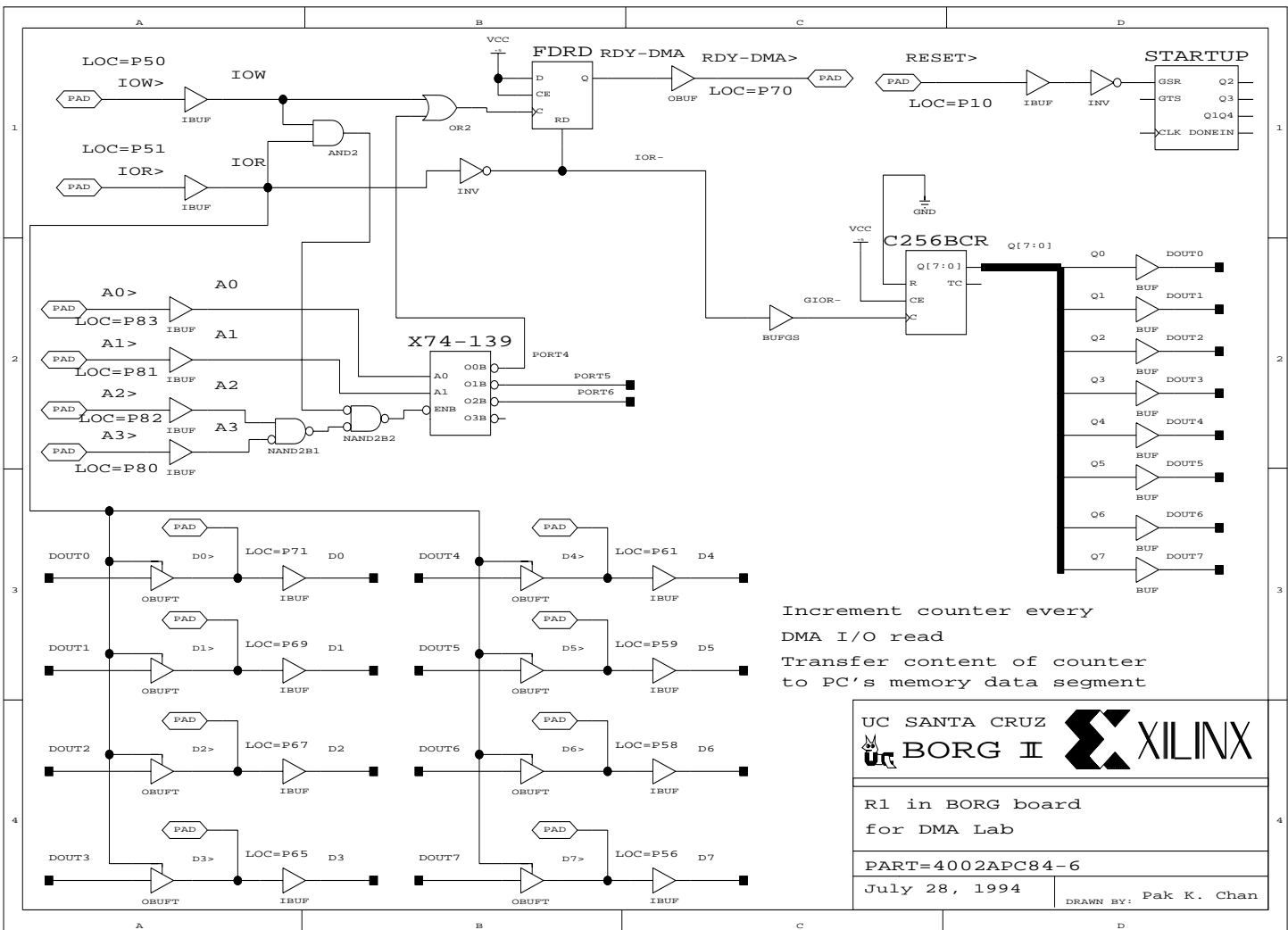


Figure 9.10: DMA Lab. Design in R1 FPGA.

```

/* initialize the DMA controller - dmainit() in assembly language */

setcbrk(1);

printf("Initialize DMA controller.\n");
i = dmainit();
if(usebeg) {
    printf("Used first half of data area.\n");
} else
    printf("Used second half of data area.\n");

printf("Load counters values.\n");
outportb(RESETPORT, 0);
outportb(RESETPORT, 0xff);

printf("Trigger DMA transfer.\n");

outportb(DMAPORT, 0xff); /* send a 1 to request for DMA */

/* check terminal count of DMAC bit 3 of status register */
/* note: status register is destructive read */
tc = inportb(8);
printf(" Terminal count now %3x.\n", tc);

while(((tc=inportb(8)) & 0x08) != 0x08){
    printf(" . Terminal count pending %3x.\n", tc);
    /* if you see this, it means trouble */
}
printf(" Terminal count has expired %3x.\n", tc);
outportb(0x0a, 0x07); /* mask (disable) DMA Channel 3 */
putchar('\n');

printf("DMA transfers of %d bytes completed. Hit to continue.\n",DSIZE);
getchar();

if(usebeg) {
    printf("Used first half of data area.\n");
    for(i = 0; i < DSIZE; i+= INC)
        printf(" %3d : %2x\n", i, dbeg[i]);
} else {
    printf("Used second half of data area.\n");
    for(i = 0; i < DSIZE; i+= INC)
        printf(" %3d : %2x\n", i, dmid[i]);
}
}

```



```

        mov    dx, bx          ; copy the value into dx
        and    dx, 000Fh      ; keep only the lower 4 bits in dx
        inc    dx             ; increment
        and    bx, 0FFF0h     ; kill the low 4 bits in bx
        mov    ax, OFFSET dmid; get the offset of the data area
        add    ax, bx         ; add offset and bx
destok:
        out    dmactl+6, al   ; AX contains the base address - send
        mov    cl, 8         ; the low half followed by the high half
        shr    ax, cl        ; to the appropriate port
        out    dmactl+6, al

        mov    al, 57h       ; set the desired mode of the DMA controller
        out    dmactl+11, al

        out    dmactl+12, al ; clear the byte pointer bit
        mov    al, tcl       ; set the terminal count
        out    dmactl+7, al  ; According to TCL and TCH
        mov    al, tch       ; will be transferred
        out    dmactl+7, al  ; ??? bytes
        mov    al, dl        ; set DMA page register
        out    dmapage+2, al ;
        mov    al, 3
        out    dmactl+10, al ; unmask the channel
        ret
        ENDP
        END

```

9.6 Boundary Scan Lab

I'll fill in this part in the second revision of this user's guide.

9.7 Possible Term Project Description

A little bit of history, I have given this Dr. Mario design as a term project in Advanced Logic Design in Spring 1993. Four out of six groups (two per group) of students finished their projects using the older XC3000 BORG board. A project description is given on the next page.

University of California, Santa Cruz, Fall '97

CMPE 126 P.K.Chan, Oct, 1997

Project Description Part II
CMPE 126: Advanced Logic Design
DR_X. MARIO² Digital Machine (due Dec 4, 1997)

This is part two of the project description.

9.8 Initialization of the Bottle

The host program (PC) writes 8-bit words one at a time to an output port at I/O address 0x0304. There is a one-bit RDY² flag (the least-significant bit) at the output port of I/O address location 0x0305. You can use a simple FSM in the **R1** FPGA to capture TWO successive words from the PC:

```
word1 = DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
word0 = DA7 DA6 DA5 DA4 DA3 DA2 DA1 DA0
```

which represent the encoding of the initial viruses. There will be no more than 8 viruses at any levels, and the viruses are always located at the bottom of the bottle. Here is the virus encoding:

Bit1	bit0	virus
0	0	S
0	1	A
1	0	L
1	1	(no virus)

For example, an initial bottle status such as:

SSA LLSS

from left to right (column 0 to 7), they will be encoded as:

```
Position  0  1  2  3  4  5  6  7
word1 = 0  0  0  1  1  1  0  0 (bit 1)
word0 = 0  0  1  1  0  0  0  0 (bit 0)
```

The PC writes the first word **word0** and then asserts RDY low, the FSM machine reads the output port and saves the word in a bank of 8-bit registers. The PC waits for roughly 1ms, then deasserts RDY to high. It then sends out the second word **word1** and then asserts RDY low. The PC waits for roughly 1ms, then deasserts RDY to high. The FSM machine reads the output port and saves the second word in another bank of 8-bit registers. The RDY signal then becomes the **YourMove**² signal in the game.

²DR_X. MARIO is a trademark of Nintendo of America Inc.

9.8.1 Pill encodings

There are six distinct pills, and their encodings are:

	Bit	Bit	Bit
	2	1	0

AA	0	0	0
LL	0	0	1
SS	0	1	0
AL	0	1	1
AS	1	0	0
LS	1	0	1

9.9 Initialization of the Dr. Mario Machine

The host (driver) provides a global reset signal that resets all the flip-flops before each round of the game.

9.10 Handshake and Timing

After initialization and sending the viruses to the ports, the PC communicates with the **DR_X. MARIO** Machine using the following protocol.

PS: your machine is required to register the laterals (column location) and pill rotation (0, 1, 2, or 3 clockwise increment).

The port assignments in the **R1** FPGA are:

```

I/O Address: 0x300  Function:                used by X0
I/O Address: 0x301  Function:                used by X0
I/O Address: 0x302  Function:                used by X0
I/O Address: 0x303  Function: Global Reset- used by X0
I/O Address: 0x304 (from PC to Mario machine)
    
```

```

Bit    7  6  5  4  3  2  1  0
-----
| D7 D6 D5 D4 D3 D2 D1 D0 |
-----
    
```

I/O Address: 0x305 (from PC to Mario machine)

```

Bit    7  6  5  4  3  2  1  0
-----
|           Pill Type| RDY' / |
|           Bit     |       |
|           2  1  0 | YourMove|
-----
    
```

I/O Address: 0x306 (from Dr. Mario machine to PC)

Bit	7	6	5	4	3	2	1	0	

				Rot	Lateral				
				ation					
			1	0	2	1	0	MoveReady	

^									
	bdsyn/verify								viewdraw/viewsim
	evaluation function								
	FSM								
	Design						debug		report
+-----+----->									
	11/18	11/20	11/25	11/27	12/2	12/4	12/11		

This is part one of the project description. There will be one more handout which will specify the interface and hardware protocols.

9.11 Project

You will devise strategies to play **DR_X. MARIO** and implement one of the strategy with two Xilinx **XC4003A-PC84s**, and an 8K-byte SRAM. Your design will interface with a “host” computer that will be responsible for keeping track of the Dr. Mario screen and your machine’s score. The only information provided by the host computer will be the next pill.

DR_X. MARIO is a 2-dimensional color matching game in which the doctor (player) must consume the pills (and possibly viruses) before the pills destroy the patient. Figure 9.11 shows the **DR_X. MARIO** “pill bottle”. There are nine different pills, as shown in Figure 9.12, which are presented one at a time at the top of the bottle. The two halves of the capsules are colored **S**carlet, **A**quamarine, or **L**emon.³ (Actually, there are only six different pills but we are counting the mirror images of the multi-colored pills as different pills.) The doctor must choose how to place the pill within the rectangular 8×16 bottle. The pill can be rotated in units of 90 degrees

³To avoid poisoning color-blind patients the pills are also labeled with **S**’s, **A**’s and **L**’s.

and the pill can be moved left or right to the desired position. The pill then drops to the bottom of the bottle or until it is stopped by other pills already in the bottle. Figure 9.13 shows the bottle after several pills have been placed and the next pill at the top of the bottle is **A****S**.

If the doctor succeeds in placing the pill so that there is a rectangular grid region of size $1 \times n$ where $n \geq 4$ of the same color, then this region vanishes. Note that this may cause the other remaining halves of the pills to fall further down in the bottle, and when they fall, other regions may vanish, and so on. The game continues until no more pills can be placed because the two grid squares in the center columns and the top row are occupied.

Figure 9.14 illustrates an example. Suppose the doctor decides to place the **A****S** pill in the sixth column (from the left) after rotating it so that the **A** is at the bottom. The two regions which vanish are the one in row 8 filled with **S**'s and the one in column 6 filled with **A**'s. But causes the **L****S** pill in row 9 to fall down one row creating a vertical region filled with **L**'s in column 3. After removing these 4 **L**'s, their other halves fall down in columns 2 and 4 as illustrated in the third bottle. Nothing interesting happens in column 2, but in column 4 there is now a vertical region of **S**'s. There is also a horizontal region of **S**'s in row 2. This brings out the point that one side of a capsule may create both horizontal and vertical regions. After removing these two regions we end up with the fourth bottle in Figure 9.14.

One last detail that need to be mentioned is that the bottle might not be empty to begin with. There may be some viruses clinging around at different points. These viral beats look exactly like half-pills and will vanish in the same manner as the pills.

9.12 Design of a Dr. Mario player

As your term project in `cmpe126`, design and debug a digital-DR_X. **MARIO** - player machine using two Xilinx `XC4003A-PC84` and an 8K-byte RAM.

To know and understand the game, a copy of the game is in the Athena `cmpe126` directory called `Mario`. The program is called `bugs` and all the source codes are there. The controls are: `h` for left, `l` for right, `s` for clockwise rotate, `a` for counter-clockwise rotate, `j` for dropping the pill down, and `q` for quitting the game.

9.13 The game environment

Your machine will interface to a host PC that present the pills one at a time. I shall write (provide) the host PC driver. You are also allowed to an 8K-byte RAM as part of your machine. The host maintains the screen, informs the player on the next pill type, processes the player's decision, keeps track of the state of the bucket and the score.

9.14 What will be finalized later?

I reserve the right to modify:

Viruses: whether or not there will be viruses and how they will be given.

Scoring: how the player will be scored.

Interface: protocol with the host PC.

System clock rate: of your machine. The host and your machine may be driven by separate clocks.

I'll be responsible for building the host. When the host is completed in the sixth or seventh week, all the above items will be finalized.

9.15 Evaluation

There will be a (single elimination?) tournament on Dec 4, 1997 in AS 240 (2:00-4:00pm). Also, the quality of your design will be evaluated based on

- a. Your score.
- b. the number of XC4000 LCAs used, and the number of CLBs and IOBs used.
- c. the propagation delay along the critical path(s), in other words, the maximum clock rate of your design.
- d. estimate your machine's scores at different clock rates (8 MHz, 16 MHz, and 20 MHz).
- e. the documentation of your design.

9.16 Your responsibilities

- a. Devise and test at least two basic strategies with a (behavioral) high-level simulation. To examine how good your strategy is: code your strategy in C and integrated into the DR_X. MARIO source code that is supplied to you. **DUE BY Nov 4, 1997.**

Be prepared to present your strategy(ies) to the class.

There is always the danger that the high-level language constructs in C are too powerful and may not be implemented efficiently or directly in hardware. Just keep in mind that your strategy has to be realized in Xilinx FPGA, eventually. Estimate the number of CLBs that is required by your strategy(ies).

- b. Work individually. You MUST have a complete hardware prototype of the project by Dec 4, 1997.
- c. Submit a good quality final report documenting you strategy, design, schematic diagrams, timing diagrams, test plans, simulation results, design files (.lca and viewlogic), and your (logic synthesis) .eqn and .bds files on Athena. DUE BY Dec 11, 1997, 5:00pm. Please place and submit the design files on a floppy disk.
- d. Realize your design either with the BORG prototyping board.

9.17 Suggestion

When devising your strategy to solve this problem, keep the implementation constraints in mind. Students have a tendency to come up with “interesting” strategies which are not easily implementable in hardware. Please start with a VERY simple strategy first, and estimate the hardware resources needed to realize it. You can improve the game strategy later on when you have a better understanding of the constraints and the game.

A successful project requires good planning, step by step documentation, and innovation. Procrastination leads to disaster. Start working on it now.

9.18 Initialization of the Bottle

Like your midterm, an XT/PC writes 8-bit words one at a time to an output port at address 0x0304. There is a one-bit RDY flag (the least-significant bit) at the output port at address 0x0305. Your FSM in the R1 FPGA captures TWO successive words from the PC.

```
wordB = DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
wordA = DA7 DA6 DA5 DA4 DA3 DA2 DA1 DA0
```

which represent the encoding of 8 initial viruses. Here is the virus encoding:

```
Bit
10

00 S
01 A
10 L
```

For example, an initial bottle status such as:

```
SSAALLSS
```

from left to right (column 0 to 7), they will be encoded as:

```
Position   0  1  2  3  4  5  6  7
wordB = 0  0  0  0  1  1  0  0
wordA = 0  0  1  1  0  0  0  0
```

The PC writes the first word `wordA` and then asserts RDY low, the FSM machine reads the output port and saves the word in a bank of 8-bit registers. The PC waits for roughly 1ms, then deasserts RDY to high. It then sends out the second word `wordB` and then asserts RDY low. The PC waits for roughly 1ms, then deasserts RDY to high. The FSM machine reads the output port and saves the second word in another bank of 8-bit registers. The RDY signal then becomes the `YourMove` signal in the game.

9.18.1 Pill encodings

There are six distinct pills, so their encodings are:

	Bit	Bit	Bit
	2	1	0
AA	0	0	0
LL	0	0	1
SS	0	1	0
AL	0	1	1
AS	1	0	0
LS	1	0	1

9.19 Initialization of the Dr. Mario Machine

The host (driver) provides a global reset signal that resets all the flip-flops before each round of the game.

9.20 Handshake and Timing

After initialization and sending the viruses to the ports, the PC communicates with the **DR. MARIO** Machine using the following protocol.

PS. your machine is required to register the laterals and pill rotation.

The port assignments in the **R1** FPGA are:

```
I/O Address: 0x300  Function:          used by X0
I/O Address: 0x301  Function:          used by X0
I/O Address: 0x302  Function:          used by X0
I/O Address: 0x303  Function: Global Reset- used by X0
I/O Address: 0x304 (from PC to Mario machine)
```

```
Bit    7  6  5  4  3  2  1  0
-----
| D7 D6 D5 D4 D3 D2 D1 D0 |
-----
```

I/O Address: 0x305 (from PC to Mario machine)

```
Bit    7  6  5  4  3  2  1  0
-----
|           Pill Type| RDY/  |
|           Bit     |     |
|           2  1  0  | YourMove|
-----
```

I/O Address: 0x306 (from machine to PC)

Bit	7	6	5	4	3	2	1	0

			Rot		Lateral			
			ation					
			1 0		2 1 0		MoveReady	

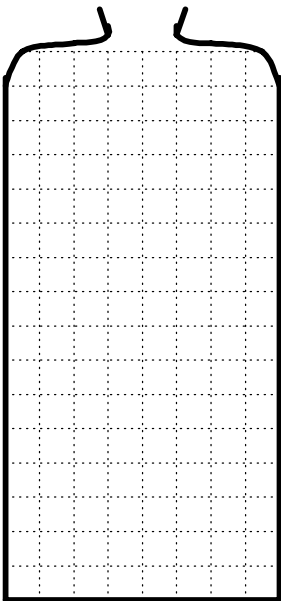


Figure 9.11: DR_X. MARIO (8 × 16) bottle.

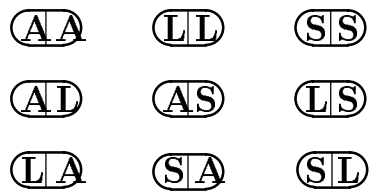


Figure 9.12: DR_X. MARIO pills.

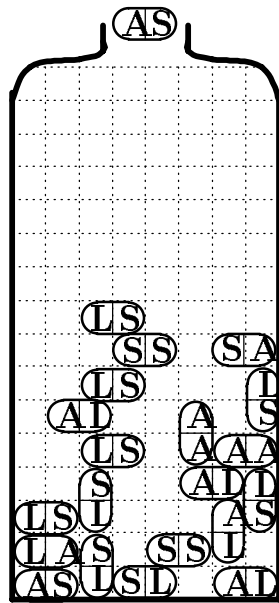


Figure 9.13: A typical game state in DR_{χ} . MARIO .

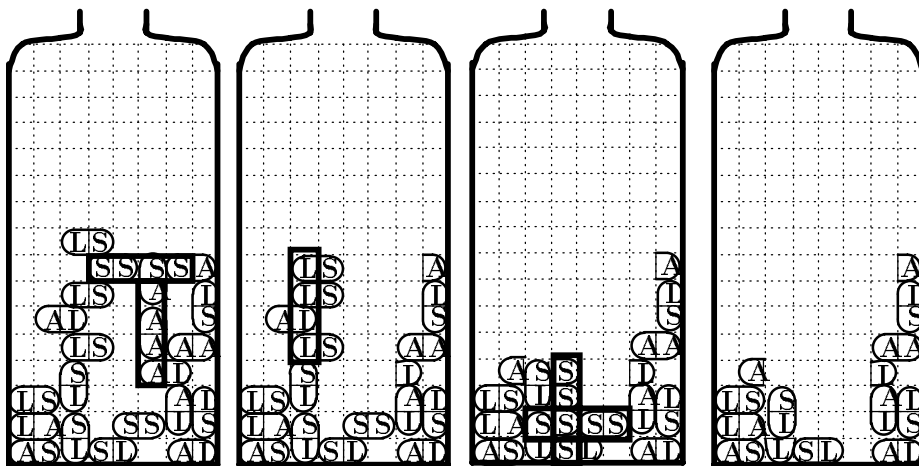


Figure 9.14: A typical game state in DR_{χ} . MARIO

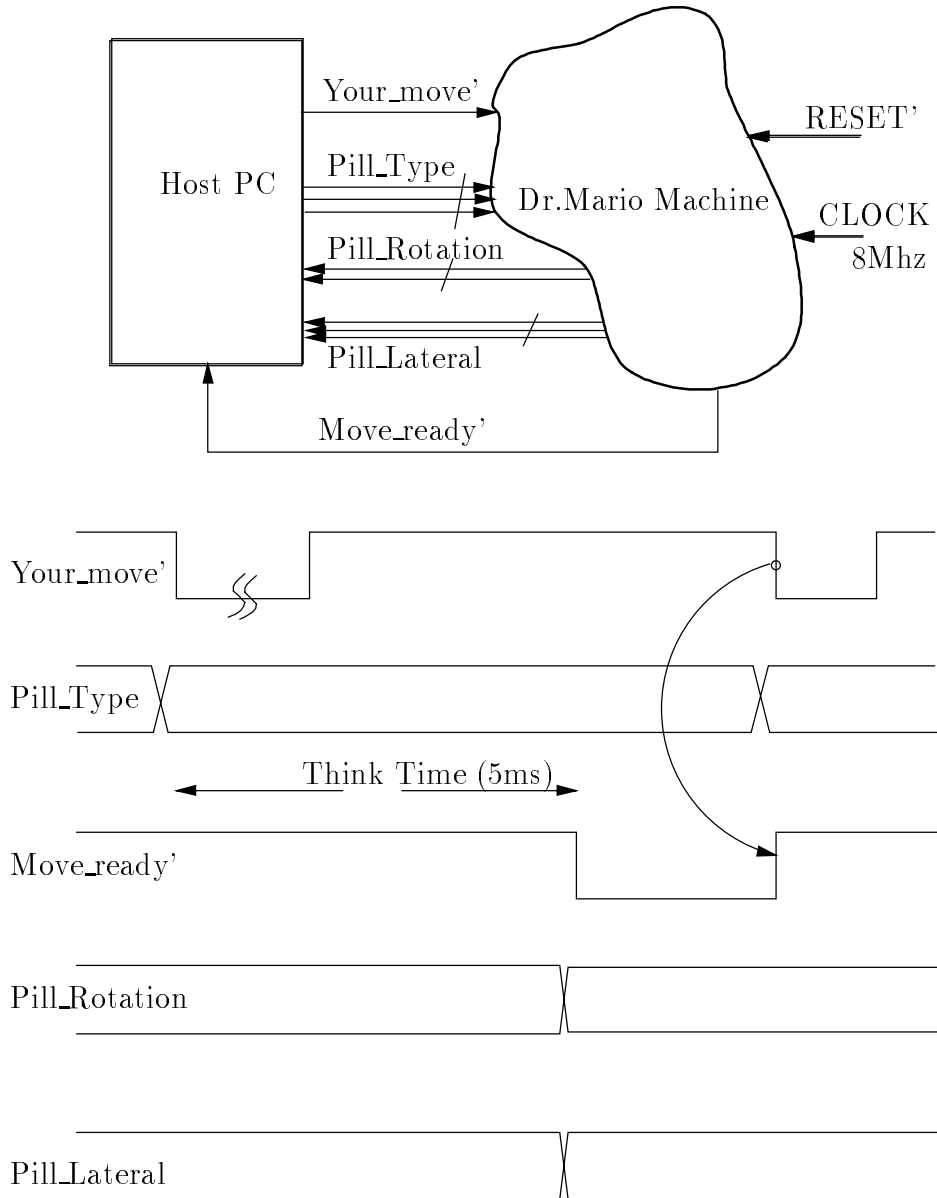


Figure 9.15: Host/ **DR_X**. **MARIO** Machine Handshake, after initialization (Tentative !!) .

10. Maze Runner project report

CMPE126 – Advanced Logic Design
Maze Runner
Spring 1995 UC Santa Cruz
Instructor: Pak K. Chan
Prepared by: Ali Ersheid (ersheido@cats)
Hernan Saab (saab@cats)
Due Date: June 15, 1995

Abstract The Maze Runner is an FPGA-based design that solves mazes running on a PC and interacting with the XC4000 BORG board. This report documents the algorithm used to solve the mazes, the implementation, and the complete design specifications.

10.1 Maze Runner Specifications

The Maze Runner machine is implemented using the XC4000 BORG board. It solves simply connected and multi-connected mazes created randomly by a PC-based host program. After the Maze Runner solves each maze the first time and finds its exit, it asks to be teleported in order to solve the maze one more time. As a rule, solving the maze the second times takes less steps than the first time.

10.2 Hardware Requirement

The Maze Runner Machine was implemented using three of the chips on the XC4000 BORG board. The following table shows the chips used:

Position	Type*	Purpose
R1	4002A	Port Controller.
X1	4003A	State Machine.
R2	4002A	Memory Controller.
X2	4003A	Not used.

* The Xilinx 4002A can be used for all chips.

In addition, an 8K-byte SRAM was also used to keep track of the back pointers and for solving the maze again after discovering it.

10.3 Host Program

The host program is written by Professor Pak Chan in C. It interacts with the BORG board and provides the Maze Runner with the following signals: Eight signals that inform that Maze Runner of the status of its surroundings:

```

0 1 2
3 0 5
6 7 8

```

A 1 indicates that the position is a wall and a 0 indicates that it is a floor or the exit. The Maze Runner uses four of these signals only (North, South, East, and West). A signal to indicate that the program is ready for your next move. A signal that indicates that you are on the exit. A global reset signal.

The Maze Runner provides the following signals to the host program: Two signals to indicate the direction of the movement (North: 00, South: 01, West: 10, East: 11). A signal indicating that the move signals are ready. A signal asking to be teleported.

10.4 Design and Implementation

The Maze Runner was designed using a variation of a depth-first search algorithm to find the exit. While discovering it, an image of the maze is mapped into the first 2K bytes of the SRAM.

10.4.1 Algorithm

The algorithm used to solve the maze is a depth-first search. The search algorithm works as follows: Three bits of memory are used for each cell to indicate whether the cell is visited, blocked, or is a backpointer to another cell. When the hero lands on the first cell, that cell is initialized to 100 to indicate that it has been visited. The hero checks its surrounding cells and moves into the first unvisited cell it finds. In doing the checking, the wall signals from the host program are checked first. If there are now walls, the memory is checked. This minimizes unnecessary memory access. When the hero moves into a new cell, that cell is marked depending on what position the hero came from according to the following code:

```

100 - North
101 - South
110 - West
111 - East

```

The hero continues as in steps 2 and 3 until it finds the exit or it reaches a dead-end. If a dead-end is reached, the first two bits of the cell value are used as back pointers after inverting the first bit. Once a cell belongs to a dead-end, it is marked with the code 011 indicating that it is blocked. If the hero finds the exit, it asks to be teleported. At this point, the Maze Runner runs in a different mode in which it reads the pointers to the exit directly from memory. This algorithm does not find the shortest path, but one that is short.

10.4.2 Implementation

The design for the Maze Runner was implemented using two XC4002A and one XC4003A FPGAs. The same design could have been implement using three 4002A FPGAs or one 4003A and one 4002A FPGAs. The next three sections describe the details of the design for each FPGA.

10.4.3 R1: The I/O Port

Since R1 is the only FPGA on the BORG board that is directly connected to the PC data bus, it had to be used for communicating with the PC. The I/O Port design is very simple and does not require much work. As can be seen in the MAZEPOR diagram on the following page, only the necessary signals are read off of the PC data bus. The I/O port reads from two different addresses from the PC. The first is labeled PORTJ and it reads the two signals Your Move and On Exit, which are indications from the PC that it is the Maze Runner's turn to move and the hero is on the exit, respectively. The second address is labeled PORTK and it reads the eight signals corresponding to the surrounding cells' status. Note that even though the PC host program provides eight signals, the Maze Runner only reads four signals corresponding to North, South, East, and West. As for writing data to the PC host program, the Maze Runner sends four bits which are Move Ready, Move0, Move1, and Teleport. These signals communicate to the host program that the data for the next move is ready, the first bit of the move, the second bit of the move, and the teleport request, respectively. The I/O Port schematic diagram is shown in Fig. 10.1.

10.4.4 X1: The Brain

The choice for placing the brain of the Maze Runner in the X1 FPGA was made because X1 had to be used to connect the I/O Port with the Memory Controller FPGAs. Instead of just using it as a routing chip, it was used as the brain at the same time. The BRAIN FPGA consists of the main FSM driving the Maze Runner machine, I/O buffers and pads, and some logic that is used primarily as an edge catcher which catches a 001 instead of 01. This logic ensures a true active signal and avoids any noise signals. The FSM of the BRAIN is called the BIGONE and it consists of the following parts:

1. Finder Box (FNDRBOX)
2. Mover
3. Memory Controller Signals (TOMEM)
4. Selector
5. Status
6. Direction Processing Logic

The BRAIN and BIGONE schematic diagrams are shown in Figures 10.2 and 10.3.

10.4.5 Finder Box

The Finder Box is simply the FSM that is used to find the exit and to control the various instructions necessary to initialize the memory, move the hero, detect the exit, and teleport. Inside the Finder Box, the Finder part is the combinational logic for the state machine. The mustang description for the Finder is as follows:

```
.i 9
.o 14
.s 5

--1-----  ADD      ADD      00-10010001-10
```

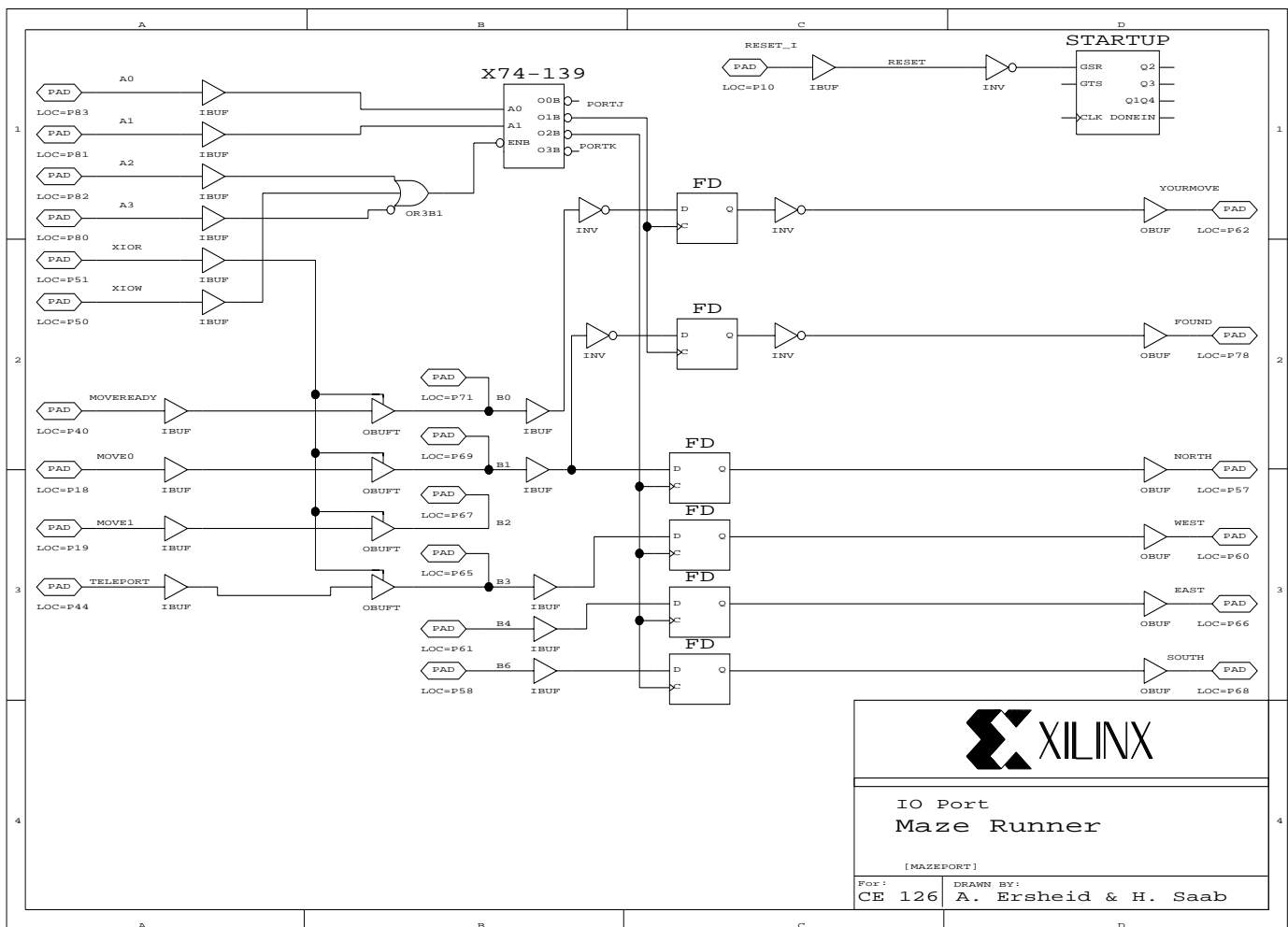


Figure 10.1: I/O port.

XILINX

IO Port
Maze Runner

[MAZEPORT]

For : CE 126	DRAWN BY : A. Ersheid & H. Saab
-----------------	------------------------------------

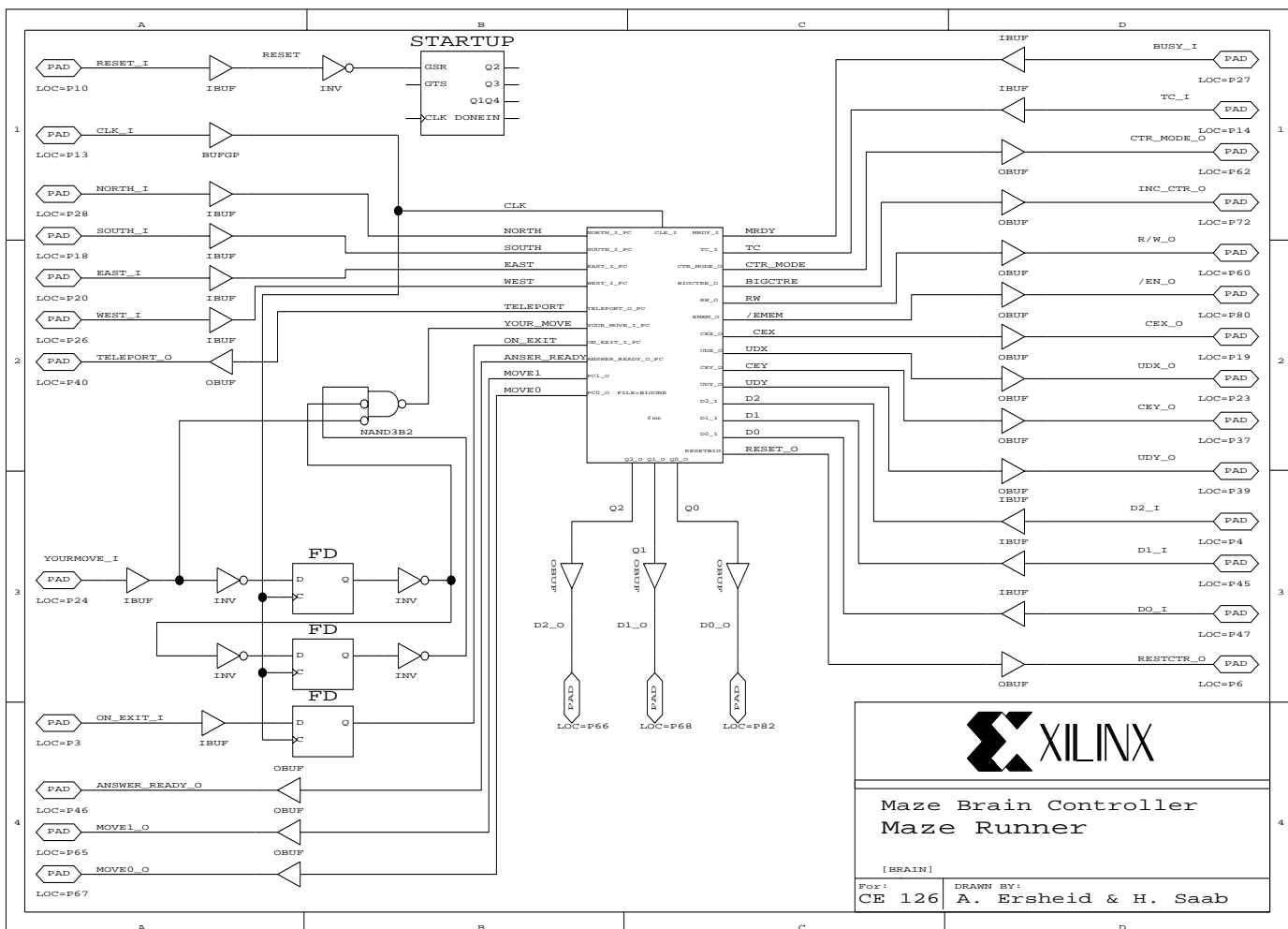


Figure 10.2: Brain.

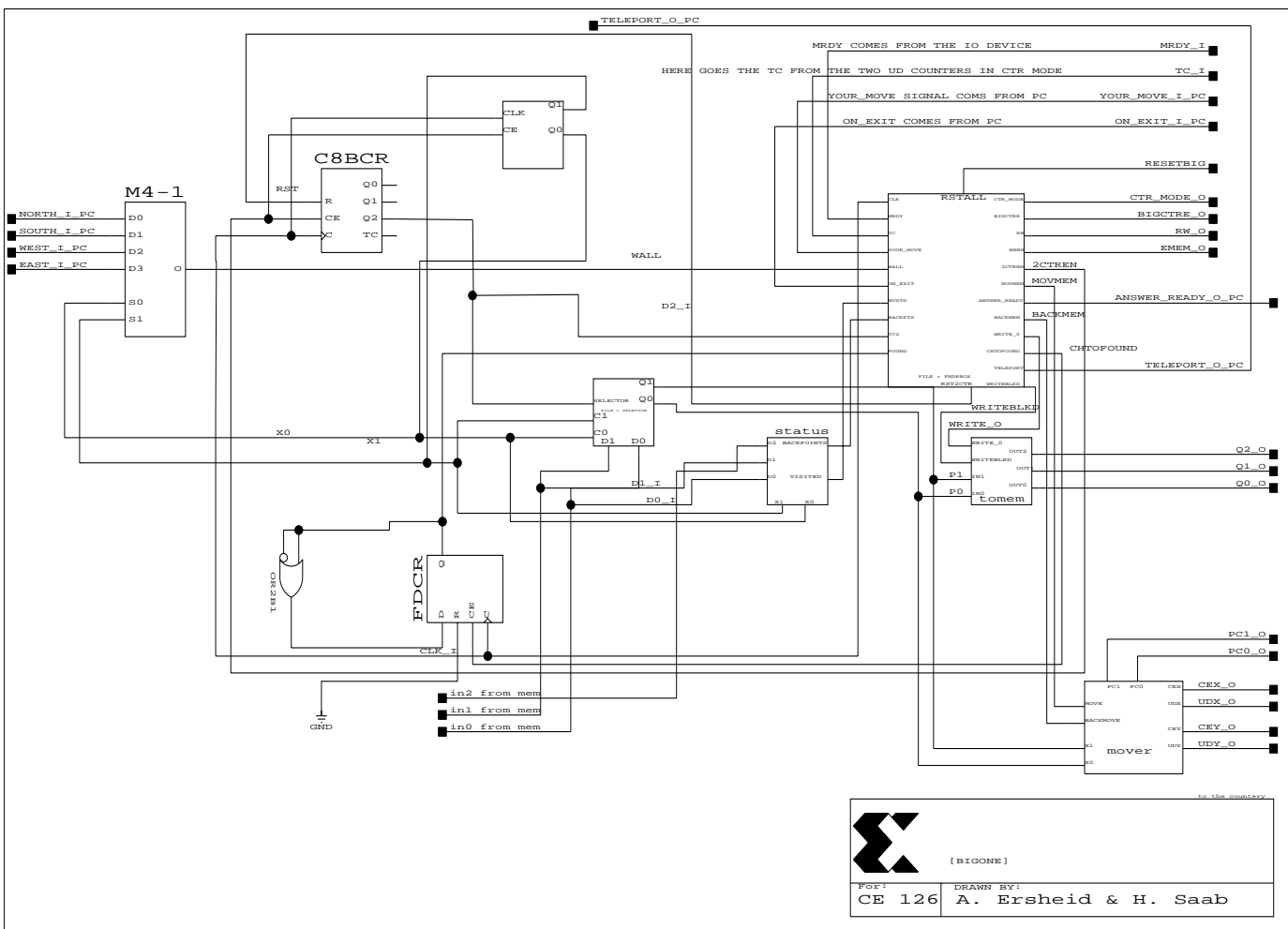



Figure 10.3: Bigone.

	
[BIGONE]	
FOR:	DRANN BY:
CE 126	A. Ersheid & H. Saab

```

--0----- ADD      START  00-10010001-00
-----   START  WAITST  10000010001100
1-----   WAITST WAITST  10010010001100
0-----   WAITST  IN      11010010001000
-----   INWAIT  IN      10000010101-00
00-----  WAITIN  IN      11110010101-00
1-----  WAITIN  WAITIN  10010010101-00
01-----  WAITIN  A1      11-1001-001-00
---1---0-  A1      INC1    00-10010001-00
---0---0-  A1      R1      00-10010001-00
-----1-  A1      R1      00-10010001-00
-----   INC1    A1      00-11010001-00
-----0-  R1      WAITR1  00100110001-00
-----1-  R1      WAITR1  00100010001-00
1-----  WAITR1  WAITR1  00110010001-00
0----1-00  WAITR1  INC1    00110111001-00
0----0-00  WAITR1  W1      00110010001-00
0----1-1  WAITR1  WAITMV  00110010001-00
0----0-1  WAITR1  INC1    00110111001-00
0----1-  WAITR1  W1      00110010001000
-----0-  W1      WAITW1  00000010001000
-----1-  W1      WAITW1  00000010001100
1-----0-  WAITW1  WAITW1  00010010001000
1-----1-  WAITW1  WAITW1  00010010001100
0-----  WAITW1  WAITMV  00-10010001000
--1----- FOUND  FOUND  00-10010010-11
--0----- FOUND  A1      00-10010001-11
--1-1--0-  WAITMV  WAITMV  00-10000001-00
--1-1--1-  WAITMV  WAITMV  00-10001001-00
--0-1--0-  WAITMV  A1      00-10010001-01
--0-1--1-  WAITMV  A1      00-10111001-01
--1-0----  WAIT  MVWAITMV 00-10010001-01
--0-0---0  WAITMV  FOUND  00-10010001-01
--0-0---1  WAITMV  A1      00-10010001-01

```

The state diagram is shown on the next page.

A PLA file was created from the above description using the one-hot assignment option in `mustang` as follows:

```
mustang -l finder > finder.pla
```

The PLA description file was modified to change the names of I/O signals and the states. The resulting PLA description looks like this:

```

.i 22
.o 27
.ilb MRDY TC YOUR_MOVE WALL ON_EXIT MVSTD BACKPTS TC2 FOUND PS12
     PS11 PS10 PS9 PS8 PS7 PS6 PS5 PS4 PS3 PS2 PS1 PS0
.ob  NS12 NS11 NS10 NS9 NS8 NS7 NS6 NS5 NS4 NS3 NS2 NS1 NS0
     CTR_MODE BIGCTRE RW EMEM 2CTREN MOVMEM ANSWER_READY BACKMEM
     WRITE_0 CHTOFOUND TELEPORT WRITEBLKD RST ALL RST2CTR

--1----- 1----- 1000000000000 00-10010001-10
--0----- 1----- 0100000000000 00-10010001-00

```

```

----- -1----- 001000000000 10000010001100
1----- --1----- 001000000000 10010010001100
0----- --1----- 000100000000 11010010001000
----- ---1----- 000010000000 10000010101-00
00----- ----1----- 000100000000 11110010101-00
1----- ----1----- 000010000000 10010010101-00
01----- ----1----- 000001000000 11-1001-001-00
---1---0- ----1----- 000000100000 00-10010001-00
---0---0- ----1----- 000000010000 00-10010001-00
-----1- ----1----- 000000010000 00-10010001-00
----- ----1----- 000001000000 00-11010001-00
-----0- ----1----- 000000001000 00100110001-00
-----1- ----1----- 000000001000 00100010001-00
1----- ----1----- 000000001000 00110010001-00
0---1-00 ----1----- 000000100000 00110111001-00
0---0-00 ----1----- 0000000001000 00110010001-00
0---1-1 ----1----- 0000000000100 00110010001-00
0---0-1 ----1----- 0000001000000 00110111001-00
0---1- ----1----- 0000000001000 00110010001000
-----0- ----1----- 0000000000010 00000010001000
-----1- ----1----- 0000000000010 00000010001100
1-----0- ----1----- 0000000000010 00010010001000
1-----1- ----1----- 0000000000010 00010010001100
0----- ----1----- 0000000000100 00-10010001000
--1----- ----1----- 0000000000001 00-10010010-11
--0----- ----1----- 0000010000000 00-10010001-11
--1-1--0- ----1----- 0000000000100 00-10000001-00
--1-1--1- ----1----- 0000000000100 00-10001001-00
--0-1--0- ----1----- 0000010000000 00-10010001-01
--0-1--1- ----1----- 0000010000000 00-10111001-01
--1-0--- ----1----- 0000000000100 00-10010001-01
--0-0---0 ----1----- 0000000000001 00-10010001-01
--0-0---1 ----1----- 0000010000000 00-10010001-01

```

To create logic equations, the above file was imported into misII. The entire file was collapsed using the `clp` command and an equation file was exported to an `eqn` file. The `eqn` file was then modified manually to make some of the obvious logic reductions. The `eqn` file was brought into misII and the boolean script was run on it to produce the following `eqn` description file:

```

INORDER = MRDY TC YOUR_MOVE WALL ON_EXIT MVSTD BACKPTS TC2 FOUND PS12 PS11 PS10
          PS9 PS8 PS7 PS6 PS5 PS4 PS3 PS2 PS1 PS0;
OUTORDER = NS12 NS11 NS10 NS9 NS8 NS7 NS6 NS5 NS4 NS3 NS2 NS1 NS0 CTR_MODE
          BIGCTRE RW EMEM 2CTREN MOVMEM ANSWER_READY BACKMEM WRITE_O
          CHTOFOUND TELEPORT WRITEBLKD RSTALL RST2CTR;
NS12 = YOUR_MOVE*PS12;
NS11 = !YOUR_MOVE*PS12;
NS10 = MRDY*PS10 + PS11;
NS9 = !MRDY*PS10 + [33];
NS8 = MRDY*PS8 + PS9;
NS7 = !YOUR_MOVE*FOUND*PS2 + !MRDY*TC*PS8 + !YOUR_MOVE*[35] + !YOUR_MOVE*PS0 +
      PS6;
NS6 = WALL*!TC2*PS7 + [27];

```

```

NS5 = TC2*PS7 + !WALL*PS7;
NS4 = MRDY*PS4 + PS5;
NS3 = !MVSTD*!FOUND*[34] + TC2*[34];
NS2 = BACKPTS*FOUND*[34] + !MRDY*PS1 + YOUR_MOVE*PS2;
NS1 = MRDY*PS1 + PS3;
NS0 = !YOUR_MOVE*!ON_EXIT*!FOUND*PS2 + !TELEPORT;
CTR_MODE = PS8 + PS9 + PS10 + PS11;
BIGCTRE = !MRDY*PS8 + !MRDY*PS10;
RW = [33] + PS4 + PS5;
EMEM = !PS11*!PS9*!PS5*!PS3;
MOVMEM = !YOUR_MOVE*TC2*[35] + !TC2*PS5 + [27];
ANSWER_READY = !PS2 + !ON_EXIT + !YOUR_MOVE;
BACKMEM = TC2*[35] + [27];
WRITE_0 = !TC*PS8 + NS8;
CHTOFOUND = !TELEPORT;
TELEPORT = !PS0 + !YOUR_MOVE;
WRITEBLKD = TC2*NS1 + NS10;
RSTALL = NS12 + PS0;
RST2CTR = !ON_EXIT*PS2 + !YOUR_MOVE*PS2 + PS0;
2CTREN = PS6;
[27] = MVSTD*!TC2*!FOUND*[34] + !BACKPTS*FOUND*[34];
[33] = !MRDY*!TC*PS8;
[34] = !MRDY*PS4;
[35] = ON_EXIT*PS2;

```

The eqn file was used to create an xnf file using eqn2xnf:

```
eqn2xnf -4 finder.eqn
```

Finally, the following two programs were run on the xnf file in order to create the Viewdraw schematic:

```

xf2wir finder
viewgen finder

```

The resulting schematic diagram is shown in 10.4.

10.4.6 Mover

Activates the signals necessary to move the hero in the host program and it's pointer in memory according to the input signals generated from the Finder FSM. Two of the output signals PC0 and PC1 are connected directly to the host program as the necessary move bits (move0 and move1). The four other output signals are connected to the 5-bit and 6-bit up/down counters in the Memory Controller. Two of the four signals are for chip enable and the other two are for choosing the appropriate up/down signals for the counter. The Mover schematic diagram is in 10.5.

10.4.7 Memory Controller Signals

This part is designed to control the three data bits that go directly to the memory. If the FSM is in the initialization mode, the part deasserts the data lines; otherwise, it generates the proper 3-bit code depending on the direction and the blocked status of the cell. The TOMEM schematic diagram is shown in 10.6.

10.4.8 Selector

This part consists of two muxes that choose between the pointer in memory and the pointers in the FSM according to the selector signal, which is an indication from a counter that all the surrounding cells have been checked. The Selector Schematic Diagram is shown in 10.7.

10.4.9 Status

This part decodes the 3-bit cell data and determines if the cell is visited or if it is a backpointer to another cell. The status schematic diagram is in 10.8.

10.4.10 Direction Processing Logic

The 2-bit counter creates the selector signals for the 4-1 multiplexer. The purpose of the mux is to select one surrounding cell status at a time. The 3-bit counter is reset on every move. Its purpose is to create the circular movement of the hero. The Q2 signal out of this counter is high when all of the four surrounding cells have been checked, indicating the hero should check the memory about their status.

10.5 R2: The Memory Controller

The R2 FPGA was chosen to control the memory because it is the only chip on the BORG board that is directly connected to the 8KB SRAM chip. The function of the memory controller is basically to read and write data to and from the SRAM chip. Its design is simple. The only thing that is of concern in this design is the timing problem. The Memory Controller is comprised of the following parts:

1. Memory I/O (MEMIO).
2. 5-bit up/down counter (C32BUDRD).
3. 6-bit up/down counter (C64BUDRD).
4. Counter control logic.
5. I/O pins, pads, buffers, and tri-state buffers.

The schematic diagram for the Memory controller is shown in 10.9.

10.5.1 Memory I/O

The Memory I/O part was designed using the conventional design tools *mustang*, *misII*, and *Viewdraw*. The *mustang* description for the Memory I/O part is as follows:

```
.i 2
.o 5
.s 5

# Inputs : /EN, R/W
# Outputs: CS, WE, OE, OB, BS

1- S0 S0 11110
```

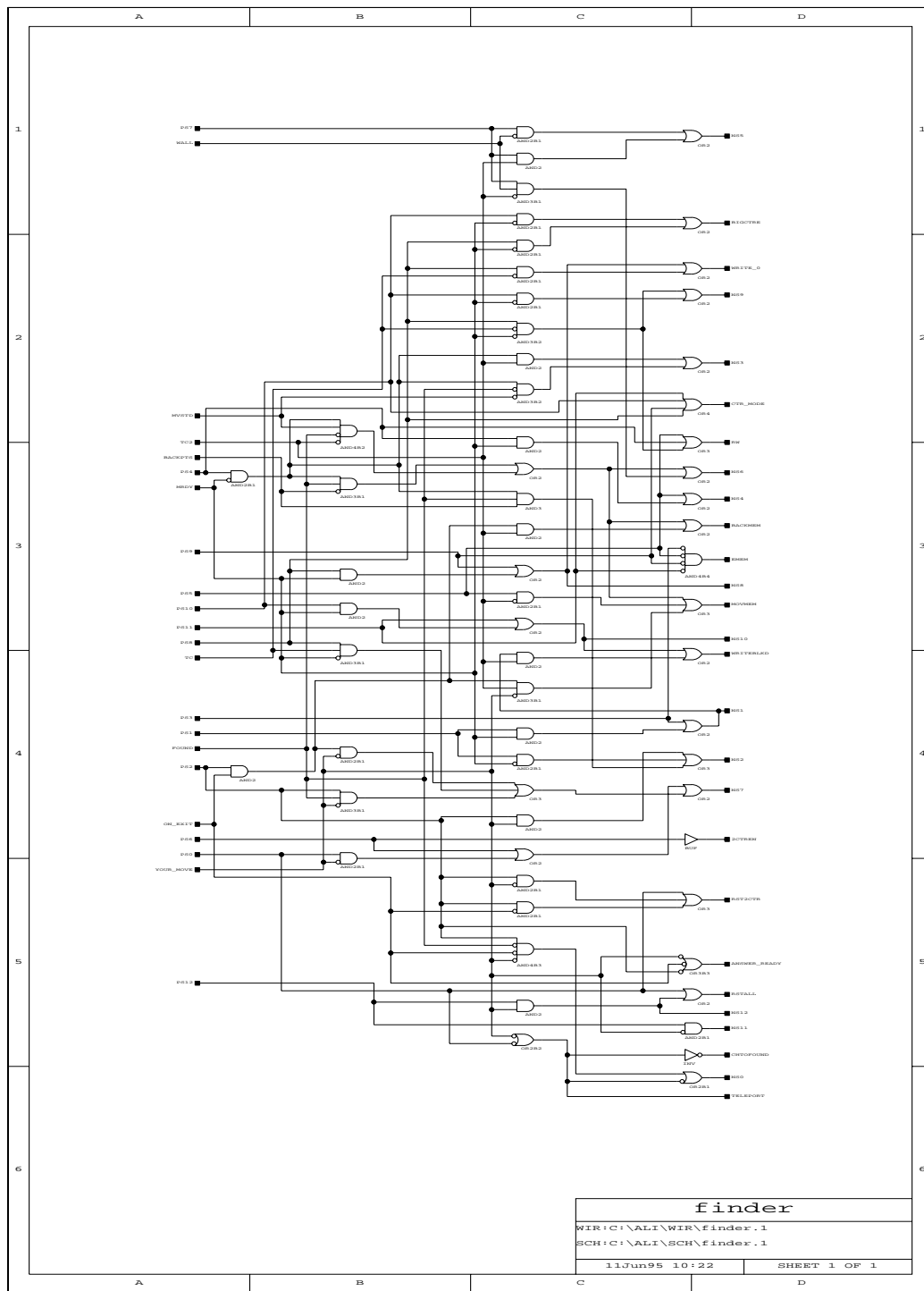


Figure 10.4: Finder.

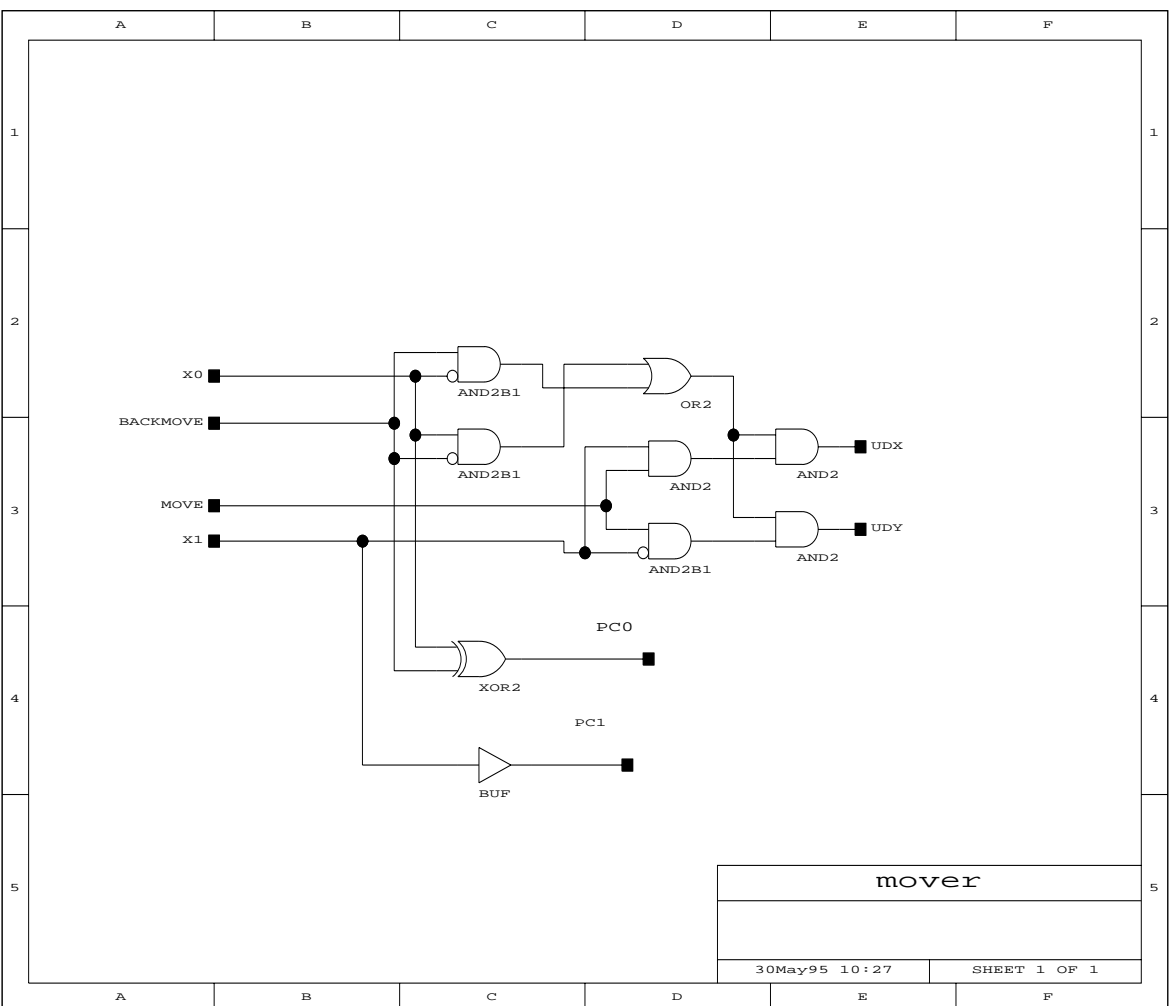


Figure 10.5: Mover.

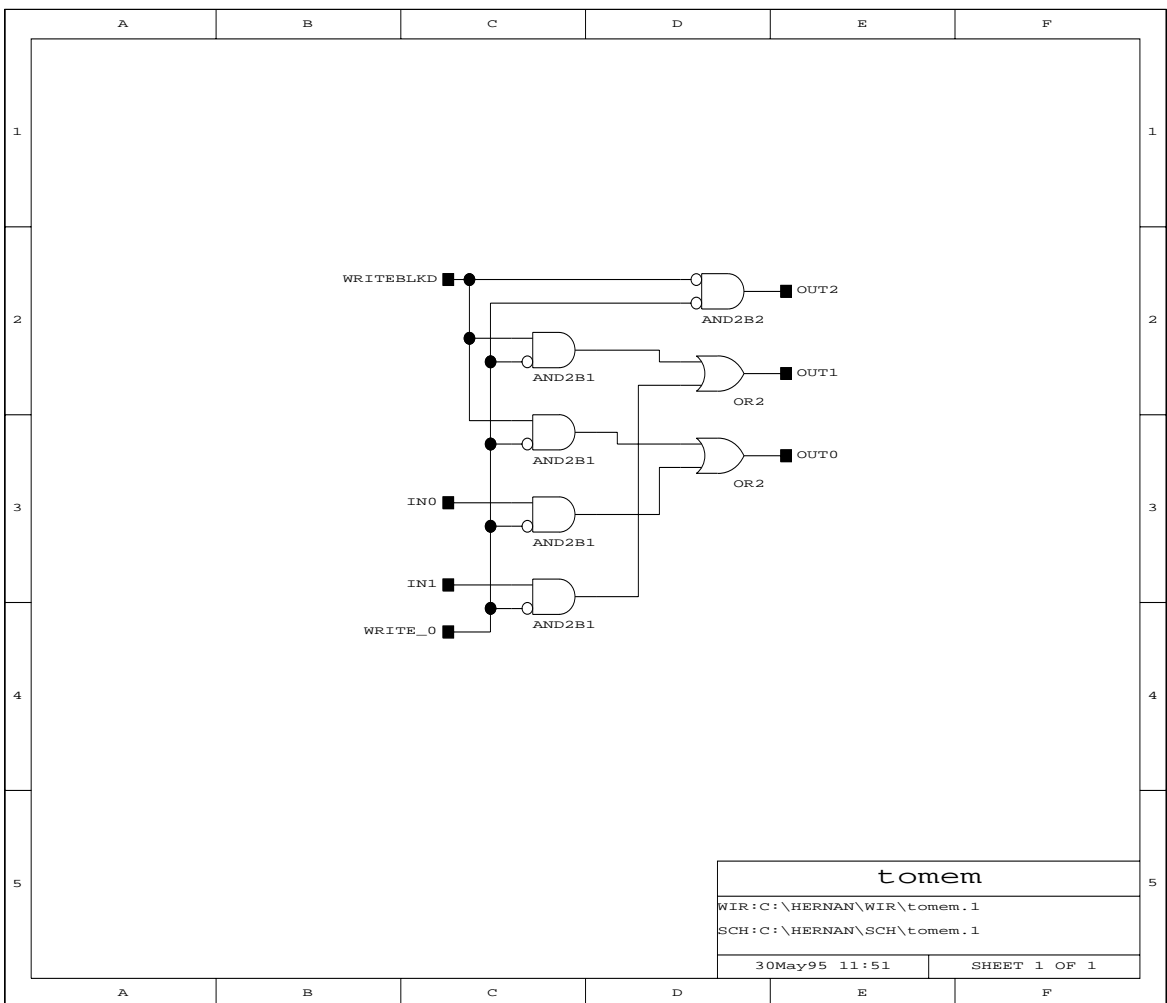


Figure 10.6: Tomem.

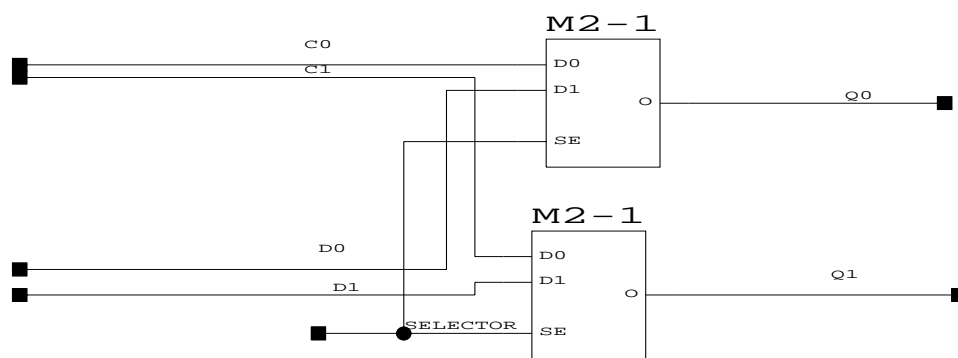


Figure 10.7: Selector.

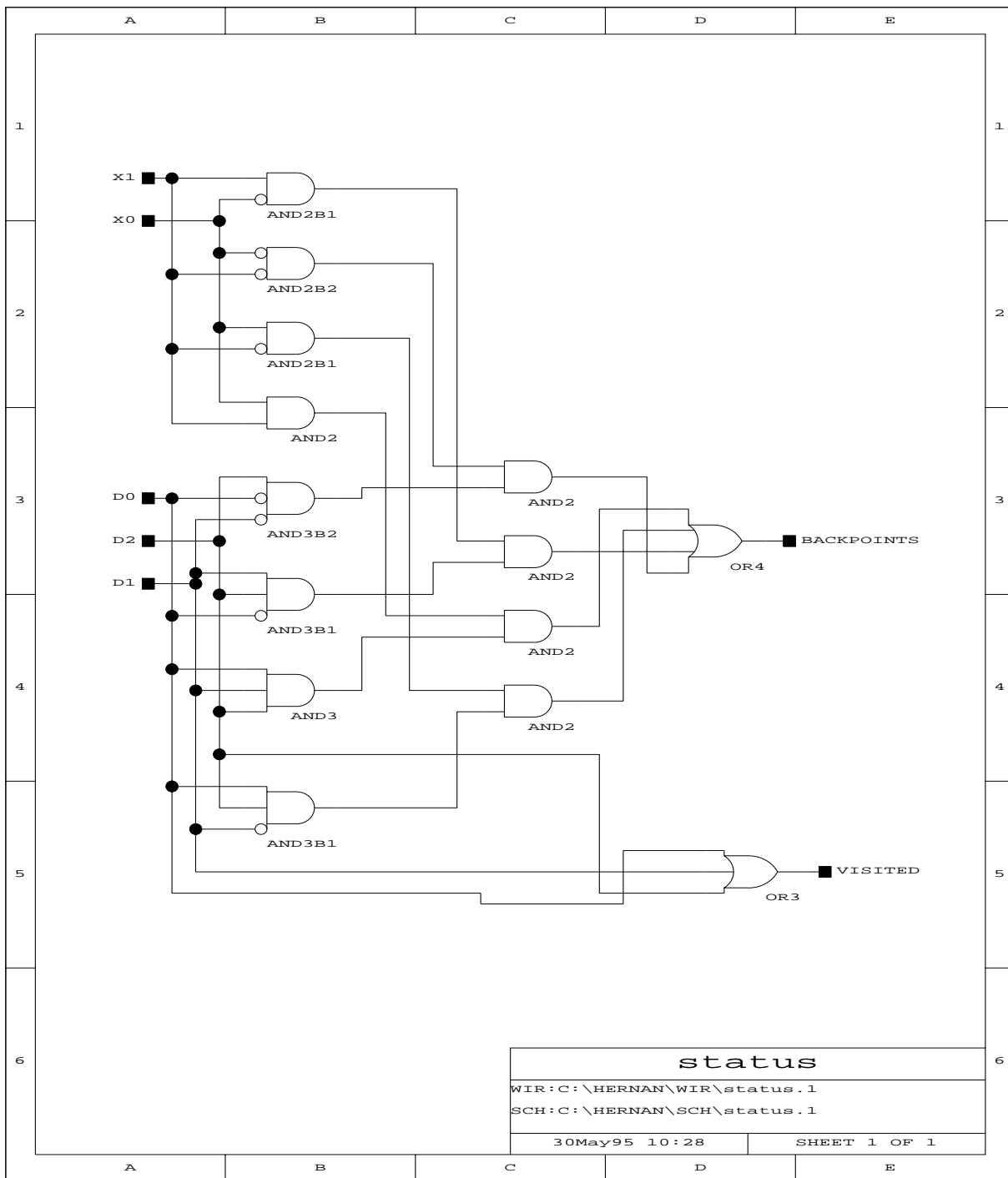


Figure 10.8: Status.

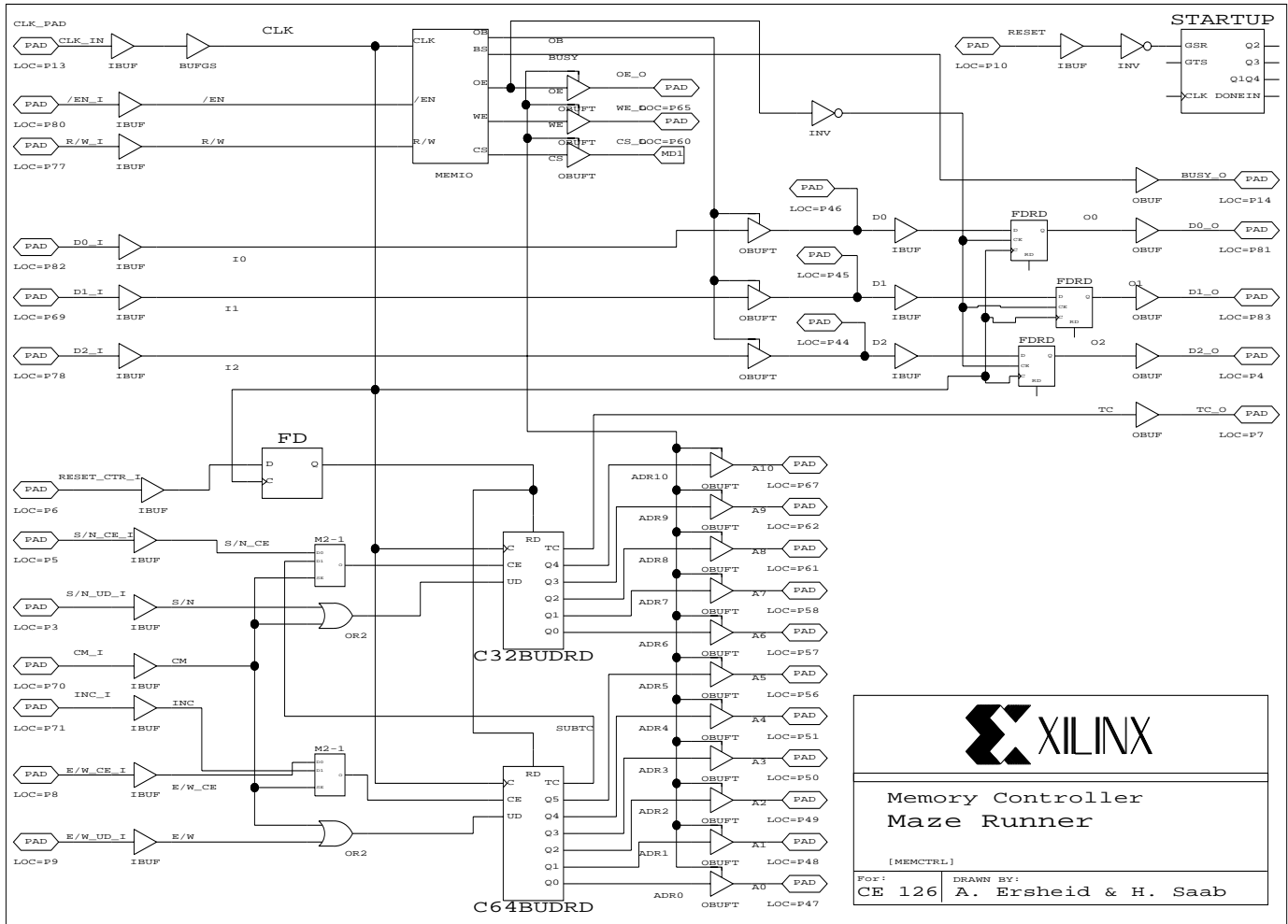


Figure 10.9: Memory Control.

```

00  S0 S1 01101
01  S0 S3 01011
--  S1 S2 00101
--  S2 S0 11110
--  S3 S4 01011
--  S4 S0 11110

```

The above description allows for an two input signals that will directly come from the BRAIN FSM. The two signals are enable and read/write. Once the enable signal is asserted, the Memory I/O reads or writes, depending the R/W signal, until the process is completed regards of the enable signal.

The first three outputs (CS, WE, OE) control the memory chip directly. The output buffer (OB) signal controls the tri-state buffers on the data lines. The busy (BS) signal is asserted while the Memory I/O is reading or writing and is connected to the BRAIN FSM. A PLA file was created from the above description using the one-hot assignment option in mustang as follows:

```
mustang -l memio > memio.pla
```

The PLA file description was modified to change the names of the input and output signals. The resulting PLA description looks like this:

```

.i 7
.o 10
.ilb /EN R/W PS4 PS3 PS2 PS1 PS0
.ob NS4 NS3 NS2 NS1 NS0 CS WE OE OB BS
1- 1---- 1000011110
00 1---- 0100001101
01 1---- 0010001011
-- -1--- 0001000101
-- ---1- 1000011110
-- --1-- 0000101011
-- ---- 11000011110

```

To create logic equations, the above file was imported into misII. The entire file was collapsed using the clp command and an equation file was exported to look as follows:

```

INORDER = /EN R/W PS4 PS3 PS2 PS1 PS0;
OUTORDER = NS4 NS3 NS2 NS1 NS0 z4 z3 z2 z1 z0;

NS4 = /EN*PS4 + PS0 + PS1;
NS3 = !/EN*!R/W*PS4;
NS2 = !/EN*R/W*PS4;
NS1 = PS3; NS0 = PS2;
CS = /EN*PS4 + PS0 + PS1;
WE = PS0 + PS2 + PS1 + PS4;
OE = !R/W*PS4 + /EN*PS4 + PS0 + PS1 + PS3;
OB = R/W*PS4 + /EN*PS4 + PS0 + PS2 + PS1;
BS = !/EN*PS4 + PS2 + PS3;

```

Before implementing the above logic equations in Viewdraw, some of the output equations were reduced manually to look like this:

```

WE = !PS3;
OE = !R/W*PS4 + /EN*PS4 + !PS2*!PS4;
OB = R/W*PS4 + /EN*PS4 + !PS3*!PS4;

```

At this point, the logic equations for the Memory I/O were implemented in Viewdraw manually. The schematic for the Memory I/O is shown in 10.10 and the timing diagram is shown on the page after that.

10.5.2 6-Bit Up/Down Counter (C64BUDRD)

This counter is from the Viewdraw 4000 library. This counter is connected to the lower six bits (5-0) of the memory address lines to control the memory access of the hero's East-West movement. This counter is also used for initializing the first 3 bits of the first 2KB of memory when cascaded with the 5-bit up/down counter (C32BUDRD). The input signals to this counter are described below under the Counter Control Logic section.

10.5.3 5-Bit Up/Down Counter (C32BUDRD).

This counter is from the Viewdraw 4000 library. This counter is connected to the upper five bits (10-6) of the memory address lines to control the memory access of the hero's North-South movement. This counter is also used for initializing the first 3 bits of the first 2KB of memory when cascaded with the 6-bit up/down counter (C64BUDRD). The input signals to this counter are described below under the Counter Control Logic section.

10.5.4 Counter Control Logic

Since the two up/down counters serve two purposes (initialize memory and addressing), their input signals must be controlled to determine their current purpose. The operation mode of the counters is determined by a signal coming from the BRAIN FSM. This signal is called counter mode (CM) and is asserted when the counters are used to initialize the memory and deasserted otherwise. When in counter mode, the two counters are cascaded together to create an 11-bit counter. In this case, the counters operate as follows:

1. The CE input signal to the 5-bit counter is the TC output signal of the 6-bit counter.
2. The CE input signal to the 6-bit counter is the increment (INC) signal from the BRAIN FSM. The INC signal is used to increment the now 11-bit counter. This signal is used only in the counter mode.
3. The U/D input signals to both counters is high, causing both of them to act as up counters.

When not in counter mode, the two counters operate independently. In this case, the two counters' inputs come directly from the BRAIN FSM.

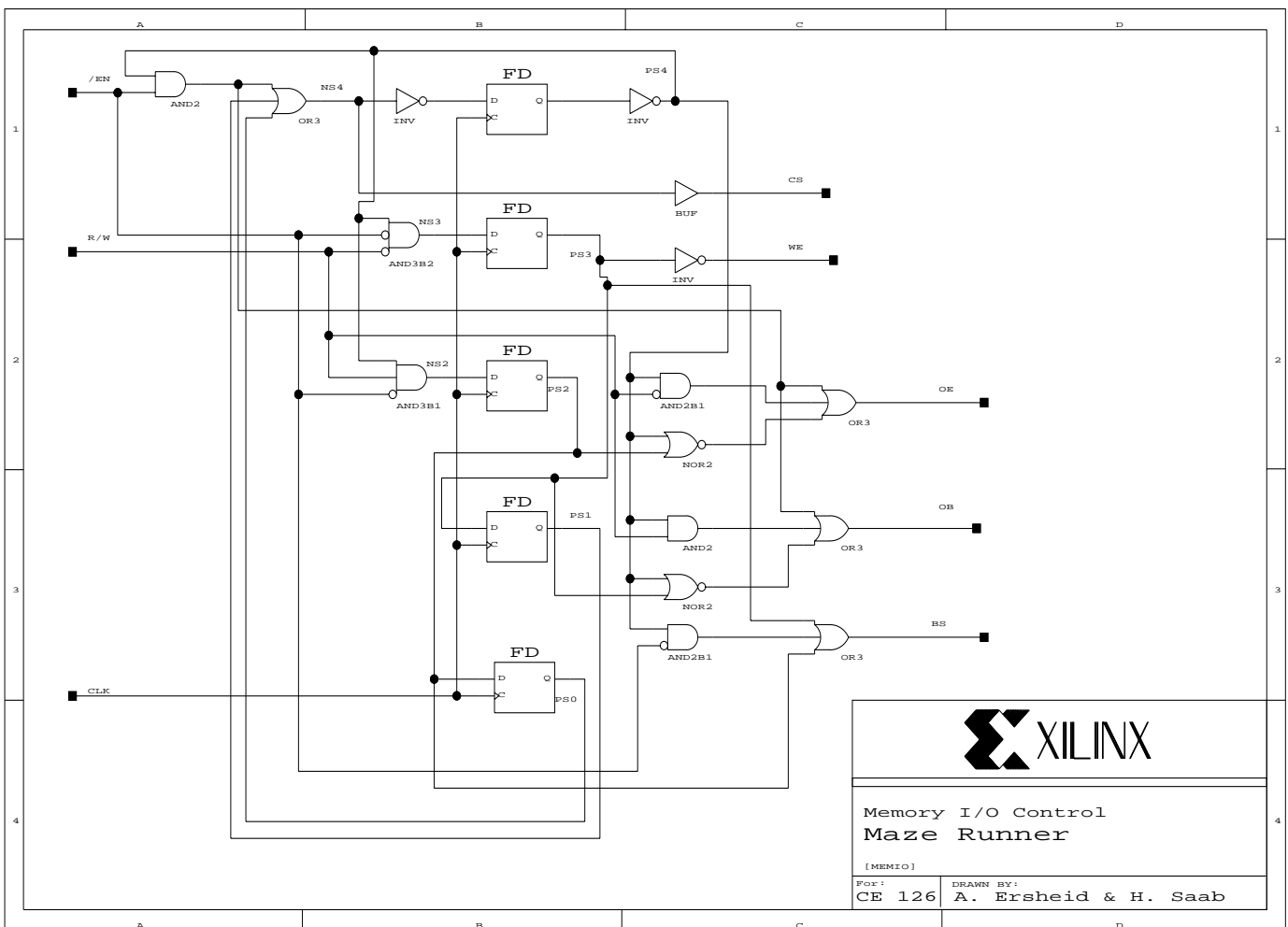


Figure 10.10: Memory I/O.

10.5.5 I/O pads, buffers, and tri-state buffers

Only three data lines and 11 address lines are used in the Memory Controller. Two of the data bits are used to store the back pointer and one for the visited/unvisited flag. The use of only 2 KB of memory limits the maximum size of the maze to 64 columns by 32 rows. The address lines are tri-stated using the CS signal in order to release them when not in use. The data lines are tri-stated using the OB signal in order to control reading and writing using those lines. The use of the I/O pads and buffers is self-explanatory.

10.6 Testing and Verification

Testing and verification for the Maze Runner consisted of downloading to the BORG board and observing its action on the screen. On occasions, the digital oscilloscope had to be used. Viewsim was also used to test the values of the address bits, the data bits, and other functions of the Maze Runner. The only critical timing problem was the timing of the Memory Controller, which was very simple and worked from the first time it was designed. Most of the debugging had to be done in the BRAIN FSM and its components. The design of the BRAIN FSM had a bug that took several days to find. The bug was not marking the first position that the hero lands on as visited. This bug caused the teleport action to do strange things. Since the position was not marked properly, the hero kept visiting that position. After the Maze Runner Machine was completed, it was run for over 1,600 levels. The average for the first try was 417 and for second try is 142.

10.7 Timing and Chip Utilization

According to XDELAY, the maximum clock speed for the Maze Runner machine is 12.2 Mhz. The utilization of the FPGA components is shown in the table below:

Component	R1	X1	R2	Total
Occupied CLBs	5	40	29	74
Packed CLBs	3	32	20	55
Package Pins	23	29	35	87
FG Function Generators	4	64	41	109
H Function Generators	2	11	13	26
Flip Flops	6	22	20	48
Memory Write Controls	0	0	0	0
3-State Buffers	0	0	0	0
3-State Buffer Output Lines	0	0	0	0
Address Decoders	0	0	0	0
Address Decoder Output Lines	0	0	0	0

The Memory Controller has taken up more CLBs than necessary because of the use of built-in parts such as the counters and the muxes.

10.8 Credits

The design, functionality, and algorithm used for this Maze Runner was originated by the authors. However, the design of the strategy for solving the maze has changed greatly since the beginning of the project. Both Hernan and Ali worked on the C program that developed the algorithm. Hernan has designed the BRAIN of this Maze Runner. He spent countless hours, day and night, on debugging it. Ali has helped in the design of the FSM machine of the BRAIN. Ali has taken on the responsibility of creating the Memory Controller, the Maze I/O Port, putting the entire project together (chip layout and pin assignment, etc.), and preparing this report. Hernan has also helped in the preparation of this report. Throughout the entire project both Ali and Hernan have been aware of what the other was doing, from design and implementation to debugging. Hernan spent a lot of time debugging the BRAIN FSM.

11. Troubleshooting

This section may help you isolate the problem and as a result, eliminate the need to contact technical support and allow continued productivity (variations from SONY TV guide).

Most the problems can be corrected with a better understanding of your computer's PC/XT configuration. Use diagnostic software such as QAPLUS to display your PC's configuration. You need to know the occupied port addresses, occupied IRQ channels, occupied DMA channels, and occupied memory address. Do not configure the BORG board in conflict with the occupied resources.

Symptom	Check these items
No LED1	slide switch SW5 to ON, check the conductivity of the fuse with a ohmmeter, an high impedance indicates that the fuse is blown.
computer crashed	are you using a protozone adapter card? If so, check IC 74HCT04 and connect (solder) a 22K Ohm resistor between pin 1 and pin 7 of the IC. This is a known manufacturing bug in the protozone adapter card.
No LED2	This is an indication that X0 is not configured, LED2 is tied to the DONE pin of X0 (xc4003APG120). Check that a PROM is in U3. Check plastic jumper is on the left side of J24 shunting positions 1 and 2. check position 8 of SW1 and position 1 of SW2 are open. This configuration sets X0 in the master serial mode.
bd complains x0 is dead	This may be an indication that X0 is not configured, or the communication between the PC and the BORG board is broken. Check the TTLs one by one.
board fail scan test	Check that the plastic jumpers are on the left side of jumpers J11-J23. If they all are, some of the I/O pins of the user FPGAs are dead.
board fail scan test	Check that the plastic jumpers are on the left side of jumpers J11-J23. If they all are, they might not be making very good contacts with the metal headers, push the plastic jumpers in and see if that improves the situation.

Table 11.1: To be Continued.

board fail memory test	<p>Check that position 3 of SW1 is closed. This enables the PC to access the dual-ported SRAM exclusively. Check that the memory (base) address mapping of the SRAM are matched on both the BORG board (hardware) and the software mtest.exe</p> <p>Consult Fig. 2.6 for the hardware mapping.</p>
No LED3	<p>All the DONE pins of the user FPGAs R1, X1, R2, and X2 are tied to LED3.</p> <p>Check that when you make the mcs file for download, you had all the correct bit stream and the correct part type for the FPGAs.</p>
No LED3	<p>If you are downloading using the bd program, check positions 1 and 2 of SW1 to make sure that R1 is configured to peripheral mode.</p> <p>If you are downloading using the xchecker cable, check positions 1 and 2 of SW1 to make sure that R1 is configured to slave mode.</p>
Can't interrupt PC	<p>If you are using the protozone host adapter card, check the setting of the IRQ requests.</p> <p>If you are using the BORG board in the add-in mode, check positions 5 to 8 of SW2 to select the IRQ channel.</p> <p>Check that the IRQ channel selected has no conflict with other peripheral cards.</p>
DMA not working	<p>You must use the protozone host adapter card for DMA. Check the correct setting of the DMA channel selection. Some DMA channels are only valid with a PC/XT but not a PC/AT.</p> <p>The standard X0 has no DMA mechanism built-in, but you can easily build your own.</p>
design doesn't run properly	<p>Check the maximum clock speed of your design. The default system clock is 8MHz, this may be too fast for some designs. Slow down the system clock by using the CLOCK utility.</p>
design can't access SRAM	<p>Check the logic for the arbitration of the dual-ported SRAM is correct.</p> <p>Check position 3 of SW1 for the favouritism of arbitration. Use the utility <code>arbit</code> to change the default.</p>

Table 11.2: Troubleshooting and diagnostics.

12. Acknowledgements

The development of the BORG board is supported in part by an National Science Foundation Research Initiation Award supplement. The manufacturing of the 100 BORG boards is supported entirely by Xilinx, Inc. for educational purposes. Therefore, I am grateful to Xilinx, Inc. for their support of the BORG project, in particularly to David Lam for his magnificent coordination of the BORG project, and his wonderful ability to pull all the resources together to finish this project. I am also indebted to Xilinx engineering and technical staff: Carol Henley who taught me PCB layout using PADS, Ed Resler who was willing to share his wisdom in manufacturing hardware, and Eric Wright who had given me his expert advice and read the initial draft of this users' guide.

I can't thank Jason Y. Zien enough for finding all sorts of way to improve `assign` and taking the responsibility of coding and supporting two versions of it. I thank Professor Abbas El Gamal of Stanford University for his pioneering work in FPGA education and his inspiration. Finally, special thanks to Martine Schlag for the basic algorithm of `assign` and insisting on designing an additional Tetris machine.