PIG Programming: From Combinatorial Circuits to Self-Replicating and Differentiating Structures

Nicholas J. Macias

March 16, 1999

Chapter 1

INTRODUCTION

This paper discusses techniques for programming the *Processing Integrated Grid* (PIG). While these techniques are applicable to external sources which load programs into the cells, the main focus of this paper is the question of *self-programmable systems*, in particular those which aid in the realization of an entirely self-replicating structure. This focus was choosen for three reasons. First, having a system which can replicate (and differentiate) itself may be useful for large scale applications, especially where the description and layout of its *macro cells* (collections of single cells which function together as a single, larger cellular unit) is defined or modified after the system is already running. Second, having a central focus provides a good road map for deciding which topics to discuss.

The third reason for this focus is proof of concept. There has been much discussion about the dual modes of these cells, and how that allows data and code to be interchanged, allowing cells to read, modify and write other cell's programs. While this makes sense in theory, there has previously been no proof that such things could be achieved. In fact, duplication of one adjacent cell into another was the only demonstration of such techniques, and that required careful control of an external strobe line, synchronized to the system clock, held HI for exactly 128 pulses, and then returned to LO. The question of even these simple operations being controlled from inside the grid has remained open. A grid structure which can read its own programs, transmit them elsewhere in the grid, and program new cells with them, thereby replicating itself, will provide adequate proof-of-concept.

This paper is organized as follows. Chapter 2 gives an overview of basic cell operations, revieving necessary terminology and describing the combinatorial behavior of single cells. Chapter 3 discusses the duplication of one adjacent cell into another, under control of the grid itself. The issue of dealing with a cell which may be trying to program its neighbor(s) is also discussed, and the concept of a *hardware library* is also introduced. Chapter 4 discusses *random*

cell duplication techniques for copying an essentially arbitrary source [1] to an essentially arbitrary destination. This involves the notion of *wire-building sequences*, which are used in the construction of communication channels from one point to another. Chapter 5 introduces the *sequencer*, which is used to generate particular wire-building sequences in any desired order (based on a stored list of sequence numbers, or *sequence stream*. A design for a hi-capacity memory system is also introduced. Chapter 6 discusses replication of the sequencer. It describes the necessary additional hardware, and outlines the appropriate sequence stream. Chapter 7 describes how this self-replicating structure can be used as a substrate to carry along additional cell structures which we wish to duplicate. A technique for differentiating one replicated structure from another is described, and the use of this to specialize different sections of the grid based on a global *grid map* is discussed.

[1] *essentially arbitrary* means as long as you don't care what you step on to get there, and you don't encounter anything which is designed to be unchangable

Chapter 2

BASIC CELL OPERATIONS

The purpose of this chapter is to review the basic operations of the cells, to fix some terminology, and to introduce two peculiar uses of the cells.

2.1 Single Cell Behavior

In discussing the behavior of a single cell, it is important to distinguish between two modes of operation: *program* (or sometimes *code, control* or *C*) mode, and *data* (or *D*) mode.

2.1.1 Data Mode

In data mode, a cell acts as a basic combinatorial unit. Figure 1 illustrates the layout of such a cell.

Basically, this cell functions as a general 4-input, 8-output logic unit. The inputs are supplied on the four D_{in} [1] lines, and the outputs on 4 D_{out} and 4 C_{out} lines. The mapping from inputs to outputs is governed by an internal 128 bit *program*, describing the truth table for the cell. Thus, depending on the program, a basic cell can operate as a wire, a simple gate, a 1-bit adder, a 1-bit multiplexer, etc. Figure 2 shows a sample truth table, and the corresponding 128-bit program for a 1-bit adder.

Of course, combinations of cells may be used to build more complicated cirtcuits, including memory circuits such as flip flops. A three cell SET/RESET

[1] There are some general conventions used for refering to inputs and outputs.For outputs,the first letter is D or C for DATA or CONTROL, and the second is NSEW for North, South, East or West. For inputs, only N, S, E or W are specied; the D input is understood unless otherwise stated.

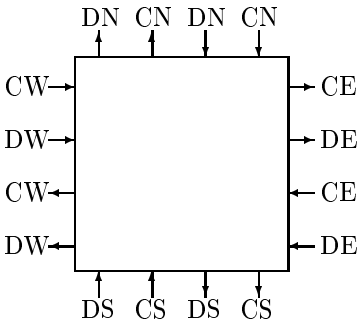


Figure 1
Basic Cell Layout

toggle flip flop is illustrated in Figure 3.

The programs for each of these cells can be written as a set of Boolean equations, as follows:

CELL[0,0]: $\{DE = E$
$DS = E\}$
CELL[0,1]: $\{DE = E\bar{N} + \bar{W}N$
$DW = E\bar{N} + WN\}$
CELL[0,2]: $\{DW = W\bar{N} + S\}$

At the end of this chapter, we will introduce a shorthand notation for specifying grid configurations.

In general, all techniques of traditional digital circuit design apply, though these cells are capable of far more by virtue of their alternate operating mode.

2.1.2 Control Mode

The other mode of operation for a cell is control mode. This is the mode used to read and write a cell's program, and is activated by placing C input HI. When a cell is in control mode, the following events occur:

(1) All C outputs are placed in the LO state

(2) The D outputs on any side with C_{in} LO are latched to their current values.

(3) The internal program is shifted one bit on each tick [2] of the external system clock. The bit being shifted off the end is presented on the D_{out} line for each side with C_{in} HI, and the bit being shifted into the program is the OR of the D_{in} lines for each side with C_{in} HI.

[2] a clock tick is a single four-phase cycle

INPUTS	OUTPUTS							
ENSW	CE	CN	CS	CW	DE	DN	DS	DW
0000	0	0	0	0	0	0	0	0
0001	0	0	0	0	0	0	0	1
0010	0	0	0	0	0	0	0	0
0011	0	0	0	0	0	0	0	1
0100	0	0	0	0	0	0	0	1
0101	0	0	0	0	0	0	1	0
0110	0	0	0	0	0	0	0	1
0111	0	0	0	0	0	0	1	0
1000	0	0	0	0	0	0	0	1
1001	0	0	0	0	0	0	1	0
1010	0	0	0	0	0	0	0	1
1011	0	0	0	0	0	0	1	0
1100	0	0	0	0	0	0	1	0
1101	0	0	0	0	0	0	1	1
1110	0	0	0	0	0	0	1	0
1111	0	0	0	0	0	0	1	1

```
0000000 00000001 00000000 00000001
0000001 00000010 00000001 00000010
0000001 00000010 00000001 00000010
0000010 00000011 00000010 00000011
```

Figure 2
Sample Truth Table for Adder
128 Bit Program Shown At Bottom
(Spaces Added for Readability)

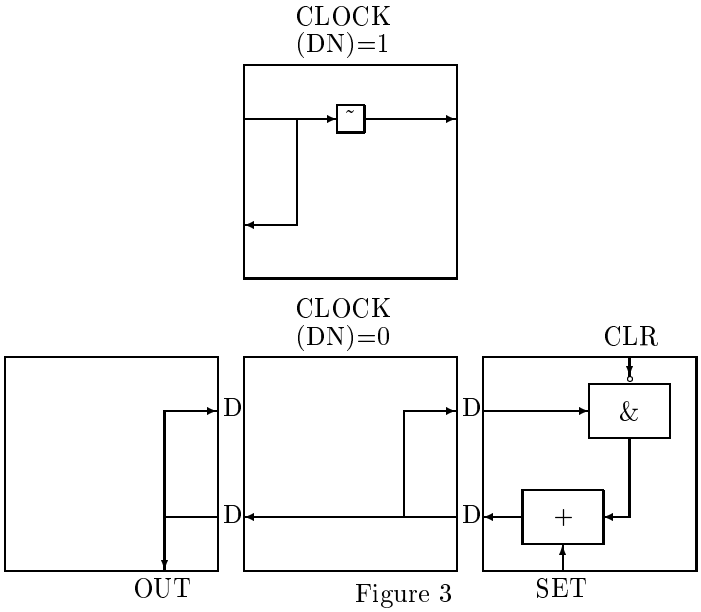


Figure 3
Toggle Flip Flop

Thus it is possible to load a new program into a cell by asserting the C_{in} line on any given side, and placing the truth table, bit by bit (synchronized with the system clock), onto the D_{in} line on the same side.

To read a cell's program, the C_{in} line is raised, and the D_{out} line is monitored to read the program. Though this read is inherently destructive, it can also be done non-destructively, as will be explained in Chapter 3.

2.2 Additional Cell Operations

There are two additional uses of cells which go beyond their basic combinatorial functionality. They involve the use of the internal cell program for something other than holding a truth table. Both of these are examples of *live* [3] cells.

2.2.1 Crystal Oscillators

By loading a particular pattern of 1's and 0's into a cell's program, placing it permenantly in C mode, and recycling the shifted data back into the cell's program, a cell can be used as a *crystal*. The effect is for the cell to output a fixed pattern of 1's and 0's repeatedly, generally at some integral fraction of the system clock's frequency. For example, if you alternate between 1's and 0's in the program, then the output of the crystal changes once for every system clock tick. If you load up eight 1's, eight 0's, eight 1's, etc., then for every eight system clock ticks, the crystal output changes state once.

A particularly useful crystal which we will encounter later is one which outputs a single pulse for every 128 system clock ticks. As we will see in Chapter 3, this is the key to internal cell duplication operations.

[3] a *live* cell is one which is (or may be) asserting a C_{out} line. The issue of dealing with such cells will be discussed in Chapter 3.

Note that a crystal is always accompanied by some sort of controller, to prevent its outputs from programming nearby cells. Because of the nature of C mode, if a crystal is allowed to run unchecked for even a single clock tick, the controller may be placed in C mode, preventing it from regaining control of the crystal itself. This can start a chain of rampant cell reprogramming, which is generally difficult to stop and impossible to reverse.

2.2.2 Memory Cells

While cells can be used to build memory units such as flip flops, we can take advantage of the architecture of a single cell to store far more information. Recall that each cell has a built in capability for storing 128 bits of truth table information. However, if we never intend to use a cell in D mode, we can load whatever we desire into the cell's program, as long as we're careful about how we handle it. A circuit which uses this technique to realize a hi capacity memory subsystem is discussed in Chapter 5.

2.3 A Little Shorthand

Eventually, we must be able to talk about cells without the benefit of pictoral descriptions, in particular when trying to specify the layout of a grid (for simulation purposes for example). For this reason, it is convienent to have a shorthand for decribing cells and their relationship to one another. We introduce that shorthand here, which is actually designed to be valid input to the PIG compiler.

2.3.1 Cell Specification

Cell specification begins with a *.def* command, and ends with *.end* Everything in between is part of the definition of a single cell. Each line has the form $< OUTPUT >=< EXP >$, where EXP is a standard boolean expression involving the symbols:

```
& | + ~ ! ^ 0 1 N S W E ( )
```

The symbols N S W and E are taken to mean DATA inputs; the output is one of DN, DS, DW, DE, CN, CS, CW and CE. So for example, the cells of the toggle flip flop of Figure 3 could be described with the commands:

```
.def FF1 ; Output cell
DE=E
DS=E
.end
```

```
.def FF2 ; middle cell (clocking)
DE=(E&(~N)) + (~W&N)
DW=(E&(~N)) + ( W&N)
.end

.def FF3 ; Loopback with SET and CLR
DW=(W&(~N)) + S;
.end
```

When such a definition appears in the input to the compiler, a truth table file is created with the given name. This ASCII file contains the progam for the given cell, and is used in realizing the layout map.

2.3.2 Cell Layout

Once a set of cells has been defined, their relative positions within the PIG must be specified. This is done by drawing a *layout map* using the symbols defined by the *.def* commands. So for the flip flop, the layout map would be as shown below:

```
.name toggle ; This is the name for the binary file
.size 1x3 ; height x width

FF1 FF2 FF3 ; Place the cells as shown
```

FF1, FF2 and FF3 are the names of files which should contain the necessary truth table for each cell. While these are generally created with *.def* commands, this is not strictly necessary.

2.3.3 Macros

Specifying the layout of a number of cells can quickly get confusing, especially when a large number of cells are involved. Things can be made much clearer by allowing the use of symbolic cell names, especially for cells whose only function is routing data from one side to another. For this reason, the compiler also contains a macro facility. This allows for single character substitutions, and is intended primarily to make more readable layout maps. The specification of such substitutions might appear as follows:

```
.macro
- ../mac/ew.we
| ../mac/ns.sn
\ ../sw.ws
+ ../mac/ew.ns.sn.we
/ ../mac/nw.wn
```

```
U ../mac/ew.nn.we ; used for feedback
.end
```

The left hand side contains the macro symbol, and the right hand side the name of the cell (file) to substitute. A layout which allows access to the flip flop's inputs could then be specified as:

```
.name better
.size 4x3
 -     \ ; This line drives the clock
 -     +     \ ; This is CLEAR
FF1   FF2   FF3
 -     -     / ; and this is SET
```

Note that macros are substituted wherever they occur, **even in the middle of a word**.

We will use this sort of shorthand in the future, except for cases where seeing a symbolic picture of the cells is particularly enlightening (as in wire building).

Two Final Notes

Two comments should be made here, more for historical reasons than anything else. First, we sometimes switch to RPN (postfix) in writing our cell definitions. The first compiler worked directly in RPN, and there was no point in continually writing two versions of the source code. Second, you will sometimes see cell names such as "RnCm" for a cell which (at least at some time) was placed at row n, column n (always indexed from 0). Before the compiler existed, cells were generally created using the debugger, and existed only in binary form. The source code for such structures has generally been produced by a *decompiler*, which analyzed the binary image and produced the Boolean expressions for each cell. These cells were named RnCm, to allow their easy placement in a layout map

Chapter 3

ADJACENT CELL DUPLICATION: HARDWARE LIBRARIES

We now present the basic concept of *cell duplication*. This is the most likely technique for reprogramming a cell, since constructing a truth table from scratch can be quite difficult. We will focus on non-destructively reading a nearby cell's program and duplicating it in another nearby cell, paying attention to questions of timing and live cells. We will also describe a *hardware library*, which is a general structure for reading one of several possible source cells and duplicating it in some fixed location.

3.1 Basic Cell Duplication

Ordinarily, reading a program from a cell is a destructive process, since it is read by shifting the program and reading the last bit as its shifted off the end. To preserve the program, this last bit must be continually re-entered into the program, and a complete 128-tick program cycle must be completed. This is in fact quite easy to accomplish, given the timing of the programming operations.

This system clock is actually divided into four phases. During Phase 1, the last bit of the program is already on the D_{out} line, available for processing and reloading back onto the D_{in} line. Phase 2 latches D_{in} while leaving D_{out} unchanged. Phase 3 shifts the program and updates D_{out}, and Phase 4 unlatches D_{in}. Hence the outgoing bit is presented on the output before the shift occurs, and the incoming bit is loaded into the program before the outgoing bit is lost.

This staggered timing allows a cell's program to be examined, potentially modifed, and loaded back into itself (and copied elsewhere) bit by bit. After 128

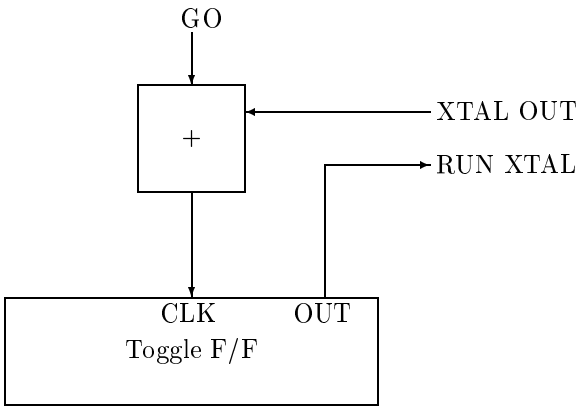


Figure 4
128 Tick Counter

clock ticks, a complete copy of the program has been obtained, and the source cell is unchanged.

3.2 Timing Issues: the 128 Tick Counter

The first demonstrations of cell duplication were done with a 4×3 cMOS cell array attached to a Tektronix Logic Analyzer. The system took approximately 30 minutes to hook up, and the analyzer was programmed, by hand, to raise the desired C_{in} line, output 128 clock ticks (placing the appropriate signals on the appropriate phase lines) and then drop the C_{in} line. This level of control is indicative of the critical timing involved in even a very basic duplication process. The controller must be aware of the system clock, and must synchronize its operations to it.

Achieving this with a grid-based circuit is actually quite simple. A 6-cell circuit call the *128 Tick Counter* [1] can accomplish this. Its basic function is to raise its output line to the HI state for exactly 128 ticks of the system clock, and then drop it to LO. Figure 4 illustrates the layout of the 128 Tick Counter.

Basically, when the GO signal is raised and lowered, the flip flop changes state from 0 to 1. This starts the crystal running, and produces the HI output for 128 system clock ticks. At the beginning of the 128^{th} tick, the crystal output goes HI, which raises the clock input of the flip flop. At the end of the tick, the output drop, the flip flop toggles to the 0 state, the crystal stop running, and the output goes LO.

For practice, Listing 1 contains the cell specifications and layout map for the 128 Tick Counter.

[1] often called the "128er"

```
;
; 5-cell 128-tick counter
; (Big-time test of compiler...)
;
; The top row is the crystal and controller,
; the bottom row a Toggle F/F.
; When CTRL[DN] is toggled, CTRL puts out a pulse
; to the F/F clock, which sets the F/F
; and starts the system. The F/F output tells CTRL
; to run the crystal, and generates the system output.
; After 128 ticks, the crystal output goes HI, which
; is OR'd with CTRL[DN] to pulse the F/F.
; At the end of the tick, the F/F clock goes LO and the
; F/F is cleared...this stops the system.
;
.def 0 ; space holder
.end

.def XTAL
DW=E&N&S&W
.end

.def CTRL
CE=S ; F/F output starts crystal
DE=E ; Preserve crystal contents
DS=N+E ; XTAL or DN can drive F/F
.end

.def FF1 ; Flip Flop feedback
DE=E
DS=E ; Output here
.end
.def FF3
DW=W
.end

.def FF2 ; Flip Flop clocking
DN=W ; always take output from left feedback cell
DW=(N & W) + (N & ~E) ; If clk=1, DW=W else DW=E
DE=(N & ~W) + (~N & E) ; If clk=1, DE=~W  else DE=E
.end
```

Listing 1
128 Tick Counter

```
; Now we layout the cells in the map

.name 128.bin
.size 2x3

;       +--- This is the GO line
;       |
;       V
0      CTRL    XTAL
FF1    FF2     FF3
;|
;+--> This it the output
;
```

Listing 1 (cont.)
128 Tick Counter

3.3 Handling Live Cells: Guard Cells

While the ability of cells to program neighboring cells is key to the PIG's power, this ability can be disasterous in the wrong circumstances. Once a cell has asserted its C_{out} line to any neighboring cell, it must fully program that cell before returning C_{out} to LO. Failure to do so will most likely ruin the neighboring cell. Moreover, if the programming cell tends to output any 1's while programming its neighbor, it is likely that the neighbor will assert one or more of *it's* C_{out} lines when it returns to data mode. The eventual corruption of the grid is generally irreversible. At best, you can hope to isolate the damaged part to prevent its further spread.

Cells whose program specifies certain input combinations for which one or more of the cell's C_{out} lines will go HI are called *live* cells; they are characterized by their potential for modifying neighboring cell's programs. In general, in a well designed system, such cells are carefully controlled and pose no problem.

However, when we begin talking about hardware libraries and other notions which involve moving around *individual* cells, we need to pay particular attention to these cells. For example (as we will see in Chapter 4), when building a communication channel, we will have to transmit a live cell to the end of the channel, in order to build an extension. Once this cell is a part of the channel, it's C_{out} lines are controlled by neighboring cells. But to transmit this cell down the channel, we need to have a stored copy elsewhere in the grid (in a hardware library actually). This copy must be controlled by some other means. In fact, we would like a way to handle all such live cells, regardless of their particular configuration. Hence we introduce the concept of a *guard cell*.

Roughly speaking, a guard cell is any cell whose function is to somehow prevent a live cell from getting out of control. We will describe three types of guard cells.

3.3.1 Passive Guard Cells

One of the most common type of live cell is one which asserts a C_{out} line but keeps the corresponding D_{out} line LO (perhaps because it's routing some other D input which happens to be LO). In this case, the cell may erase its neighbor's program, but there is no danger of creating a new live cell.

For these types of live cells, a *passive guard cell* can be used. This is nothing more than a blank cell which is placed next to the live cell, so that no other cell is reprogrammed. Passive guard cells are ideal for such low-activity live cells.

3.3.2 Protected Passive Guard Cells

Sometimes a live cell is doing more than just programming a neighbor with all 0s. If the neighboring cell were **always** in control mode, then a passive guard cell would still suffice. But we will eventually want to read the live cell, which means placing *it* in contol mode. When this happens, the live cell's C_{out} lines go LO, and the guard cell switches to data mode, and may become live itself.

In these cases, a modified passive guard cell can be used. Since the only problem is having the guard cell enter data mode, we can associate *another* cell with the guard, whose function is to continually assert a C_{in} line of the guard cell. Hence the guard cell doesn't depend on the live cell to keep it in control mode.

Such a configuration is called a *protected passive guard cell.* It is sufficient for most purposes, as long as we understand the live cell's behavior and can handle it from a side where C_{out} is always LO.

3.3.3 Active Guard Cells

Sometimes, a cell may be asserting C_{out} on all four sides. In this case, we could contain it with passive guard cells, but then we'd have no way of accessing the cell's program (remember that passive guard cells are generally kept in control mode). Additionally, we may have a fixed organization to a cell library, requiring that the cells be accessed from their Northern side. This means that an arbitrary live cell may or may not be compatible with this library, depending on which C_{out}'s it's asserting. For these cases, we need an alternative strategy for controlling the cells.

The solution is to *place the live cell in control mode, and keep it there forever.* In this way, the cell can never assert any of its C_{out} lines, and therefore will not reprogram any neighboring cells. Still, since it is in control mode, the contents of its program are continually being transmitted through a D_{out} line, and can

be processed from their. We thus need a cell which asserts C_{in} on the live cell, recirculates the cell's program, and presents its contents when required. Such a cell is called an *active guard cell*. In fact, we have already encountered an example of an active guard cell, namely the control cell used in a crystal circuit.

Active guard cells represent the most general mechanism for handling live cells of an unknown nature. There are two main considerations to their use:

(1) The cell's contents must be accesed in multiples of 128 ticks of the system clock.

(2) The guard cell must always preceed the live cell when replicating it elsewhere on the grid.

Neither of these pose a major problem if care is used in the design process.

3.4 Semi-Random Cell Duplication:Hardware Libraries

A *hardware library* is a structure which contains a number of *source* cells, a selection mechanism, and a 128-tick counter. A cell number can be entered on a set of inputs (ther *select lines*), and a GO line pulsed. The system then reads the corresponding source cell, passes its program down a channel, and replicates it in a fixed destination. When the copy operation is completed, a DONE line is raised.

Other than the actual reading and writing of the program, this is nothing more than a simple combinatorial circuit. The desired cell number is sent through a number of *match columns*, one of which will output a 1 from the last row. This is used to signal an active guard cell that it should pass its live cell's data back to the destination cell.

The code for such a hardware library is illustrated in Listing 2

The hardware library will form a central part of the *sequence generator*, which will be described at the end of the next chapter.

```
;
; Sample code for hardware library
; For recreational purposes only
;
; 7-cell hardware library.
; The desired cell is indicated in
; DW[0,0] (MSB) - DW[2,0] (LSB)
; The copy begins by RAISING then CLEARING GO,
; DW[4,0]
; The cell is copied into [3,0]
; Completion is indicated by DW[4,0]
;
; First we define the match cells
; These read W, and if it's the right bit,
; pass N to S, else they pass 0.
; (The top cell in each column passes "1" or 0)
;
.def m0 ; Match a "0" from W
DS=W~N& ; W=0, so pass N
DE=W ; and pass along data
.end
.def m1
DS=WN& ; Match a "1" from W
DE=W
.end

.def tm0 ; Top-row
DS=W~ ; W=0, so pass 1
DE=W
.end
.def tm1
DS=W ; Match a "1" from W
DE=W
.end

; Next we have the readers at the bottom
; of each match column. While DW=1, we are
; doing a transfer (so they do DE=W). If DN=1,
; they read their cell & pass its data to W
; If they get "0" from N, just route E to W

.def rdr
DE=W ; Pass "reading" signal
CS=NW& ; read *your* cell
DS=S ; non-destructively
; If N=1, pass S->DW
;    else pass E->DW
DW=EN~& SN& +
.end
```


Listing 2
Sample Hardware Library

```
; Now we define the 128-er cells
; (some of these are modified from before)

.def xtal ; crystal
CE=E~N~W~S~&&& ; Start with "1" in 1st spot
.end

.def ctrl
CE=S ; F/F output starts crystal
DE=E ; Preserve crystal contents
DS=WE+ ; XTAL or DW (GO) can drive F/F
.end

.def FF1 ; Flip Flop feedback
DE=E
DN=E ; Output on North
.end
.def FF3
DW=W
.end

.def FF2 ; Flip Flop clocking
DN=W ; always take output from left feedback cell
DW=NW& N~E& + ; If clk=1, DW=W else DW=E
DE=NW~& N~E& + ; If clk=1, DE=~W  else DE=E
.end

; need one funky routing cell
.def dctl ; Destination Control
CW=S
DW=E ; Data from row of rdr cells
DE=S
.end

; Now a few macros
.macro
. ../mac/0 ; just a place holder
- ../mac/ew.we ; Standardized notation
; means DW=E and DE=W
; (always alphabetize)
+ ../mac/sn.sw.we ; Pass RUNNING N and W
; pass GO from W to E
| ../mac/ns.sn
.end
```


Listing 2
Sample Hardware Library (cont.)

```
; and now we place the cells in the grid

.name hwlib.bin
.size 7x9
;
; Each match column is composed of three match cells
; specifying whether a "1" or "0" on that row is
; considered a match. If all three rows match, then
; the bottom match cell outputs a HI on its DS, thus
; activating its rdr

-   -   tm0  tm0  tm0  tm1  tm1  tm1  tm1 ; Sel MSB
-   -    m0   m1   m1   m0   m0   m1   m1
-   -    m1   m0   m1   m0   m1   m0   m1 ; Sel LSB
. dctl  rdr  rdr  rdr  rdr  rdr  rdr  rdr  ; DST
.   |    S1   S2   S3   S4   S5   S6   S7 ; Sources
-   +  ctrl  xtal..... ; GO/~DONE
.  FF1  FF2  FF3.....
```

Listing 2
Sample Hardware Library (concluded)

Chapter 4

RANDOM CELL DUPLICATION: WIRE BUILDING

While a hardware library is useful for reading one of several cell progams, it has the fundamental limitation that the source must be one of a pre-selected set of fixed cells, and the destination is always another fixed cell. To be able to truly synthesize new circuits from within the grid, we need the ability to read a cell from basically any location, and copy its contents into a new cell at any other location. This chapter will explain how to do this. The central idea will be that of *wire building*, by which we can construct a commumication channel from one point to another. Such channels are built up cell by cell, using very specific sequences of program streams. We will describe what these sequences look like in general, and then outline several such sequences which are necessary for structure replication. Finally, we will describe the *sequence generator*, which handles the generation of such sequences for the purpose of general wire building.

4.1 Basic Task

By a *wire* we mean a line of cells which allow us to control a non-adjacent *target* cell. In this context, "control" means we must be able to access its C_{in} and D_{in} lines on at least one side, presumably because we wish to program this remote cell (if we wish to access more than one side, we assume this will be done by constructing more than one wire). While this is directly possible for an adjacent cell, a cell which is non-adjancent is harder to control, because you can not directly transmit a C_{out} signal the way we do with D_{out} (i.e., asserting

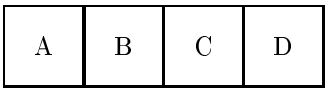


Figure 5
Single-Cell-Wide Wire

a C line always directs the **adjacent** cell to enter C mode). There are two ways to solve this problem. One is to let a cell which **is** adjacent to the target cell do the controlling. This is ideal for a **very** short wire, but is not very practical for longer ones.

The other solution is to use a wire which is more than one cell wide, composed of multiple *channels*. Having more than one path through the wire allows one to transmit the desired C_{out} values. The construction of such wire is trickier than the single-cell-wide case, but the resulting structure is far more flexible.

4.2 Skinny Wires and Towers of Hanoi

We can directly program an adjacent cell, because each cell's truth table lets us set a neighbor's C_{in}. What about a nearly adjacent cell, say one which is two cells away? Let's call our cells "A" "B" etc. and lay them out as shown in Figure 5.

Suppose we have direct control over cell A, and we wish to load a program into cell B. Well, since A can directly control B, and we can control A, we can tell A how we want B controlled. So we have a 3 step procedure:

Step 1: Program A to pass DW_{in} to DE_{out} and set CE_{out} HI
Step 2: Send our program for B into A[DW_{in}]
Step 3: Program A to pass DW_{in} to DE_{out} and set CE_{out} LO (to return B to data mode)

Suppose now we wish to load a program into cell C. We repeat the above procedure twice, as follows:

Steps 1-3: Program B to pass *its* DW_{in} to DE_{out} and set CE_{out} HI
Step 4: Send our program for C into A[DW_{in}]
Steps 5-7: Program B to pass DW_{in} to DE_{out} and set CE_{out} LO

This 7 step procedure allows full control over cell C. Well, you can tell what's gonna happen next. To program cell D, we need 7 steps to tell C how to control D, one step to send in D's program, and 7 more steps to tell C to stop programming D and just pass data to it.

PC_0	PC_1	$\cdots$	PC_N	T
CC_0	CC_1	$\cdots$	CC_N	B

Figure 6
Multi-Channel Wire

In general, we need on the order of 2^n steps to build such a wire of length n. If we wish to access cells more than a few cells away, this approach if prohibitively slow (a wire a mere 64 cells long could essentially never be completed). Still, this "Tower Of Hanoi" solution is ideal for very short wires, and in fact will be used constantly in what follows.

4.3 Multi Channel Wires

In order to control both the C and D lines of a target cell, we can construct a wire with more than one channel along its length. For example, we can run two channels side by side, as shown in Figure 6.

The bottom row is called the *control channel* (CC), and its cells are called CC cells. The top row is called the *program channel* (PC), and its cells are called PC cells. The last cells (CC_N and PC_N) are called *head cells*, the previous CC and PC cells are called *routers*. The cell labeled T is the target cell, and B is unused for now. The cell being controlled by the wire is said to be at the *head of the wire*.

So the idea is, the string of PCs will carry a stream of data to the target, while the string of CCs carry a HI or LO signal, which will be sent to the C_{in} of the target cell T. So each CC_i and $PC_i (i < N)$ simply pass DW_{in} to DE_{out}. However, **the head cells must be different from the others in the wire**. In particular, CC_N passes its data (from DW_{in}) to PC_N (via $CC_N[DN_{out}]$). PC_N still passes its DW_{in} to its DE_{out}, but **also takes its DS_{in} (from CC_N) and passes it to CE_{out}**. This means that by placing a HI on the control channel, the data sent down the program channel will be entered into the target as a new program. Thus we have complete control over the target cell T.

4.4 Extending the Wire

Now for a great confession...**the thing we do most often with wire is use it to build more wire.** The process of extending a wire involves programming the B and T cells at the head of the wire to function as new CCs and PCs.

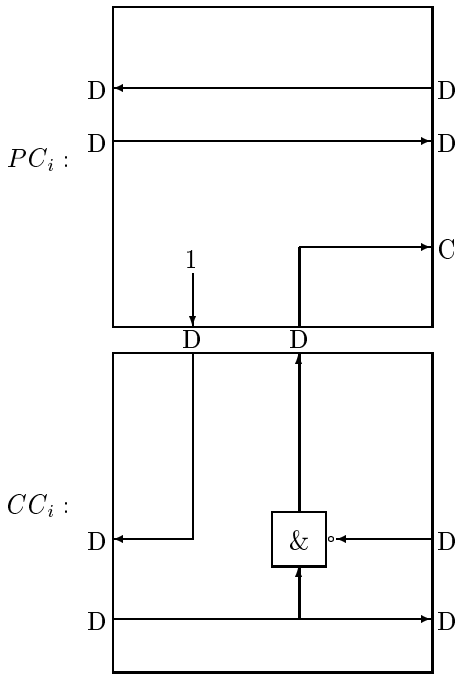


Figure 7
CC and PC Cells

T can be programmed directly, and B can be programmed using the Tower of Hanoi approach outlined above. The only tricky part is that the previous head cells must be turned into normal CC and PC cells (routers), in order to move the head of the wire forward.

The technique for doing this is to pass a signal **backwards** from CC_{N+1} to CC_N to turn CC_N into a router. We can actually leave PC_N as is; as long as it doesn't get a HI on its DS_{in}, it will function as a router only.

The trick is that once CC_N gets its signal, we can no longer program T. So CC_{N+1} and PC_{N+1} must be in place **before** this signal is sent to CC_N. This can be done by using the fact that in control mode, most of the D_{out} lines are latched, and are only updated when the cell is returned to data mode. So the last thing we do is program PC_{N+1} to be a new head cell, and to pass the necessary signal back to CC_N via CC_{N+1}. This signal will not be sent until we finish programming PC_{N+1}. Figure 7 shows the configuration of the CC and PC cells.

This leads to a three[1] step procedure for moving the head of the wire:

[1] actually 3.01...it's important that the CC line be returned to LO after Step 3, at least for a brief instant

STEP 1: Program T to allow programming of B. By sending a HI down CC, we can load any program we wish into T. So in step 1, we program T to take its DW and send it to DS, and to put CS HI. This done, B is ready to be programmed.

STEP 2: Program B as CC_{N+1}. We now set CC to 0 and send the new CC_{N+1} program down the PC line. This programs B to function as a head cell.

STEP 3: Program T as PC_{N+1}. We now set CC HI again and send the new PC_{N+1} program down the PC line.

Normally[2], the CC head cell is getting a 0 from DE_{in}, which means its DW_{in} is sent to its DN_{out}, which allows us to activate the PC head cell's CE_{out}. When PC_{N+1} (and CC_{N+1}) are in place, the new PC head cell sends a 1 out its DS_{out}, which the new CC head cell passes back to CC_N. CC_N thus gets a 1 on its DE_{in}, which blocks its DW_{in} from reaching its DN_{out}, and thus turns CC_N into a router.

The next time the control channel is given a HI signal, it will be passed all the way to CC_{N+1}, which routes it into PC_{N+1}, which will now assert **its** CE_{out}. Thus we have moved the head of the wire over one cell.

Figures 8, 9 and 10 illustrate each of these steps.

4.5 A Little More Shorthand

As the previous example illustrates, the process of wire building can appear quite complex. It turns out, however, that we basically need to specify only two things for each step[3]. First, we must indicate whether the control channel is given a HI or LO signal, and second we must specify what bits are sent down the program channel. The former is simply a single bit value, while the latter is essentially a complete cell program, and can be represented with a list of equations. These are precisely the data which are listed for each step.

Additionally, it is useful to have a pictorial indication of which cell(s) are actually being programmed at each step, to keep track of the leap-frogging which is typical in such recursive procedures. We therefore draw a picture of the cells involved in the sequence, and list inside each the number of any step which modifies that cell's program. Figure 11 shows this for the above example.

[2]This assumes that the cell immediately after CC_N is initially blank, or at least sending 0 out its DW_{out}. This assumption will be removed when we discuss the *pre-clearing step*.

[3]Note that by a *step* we mean a complete 128 tick programming cycle

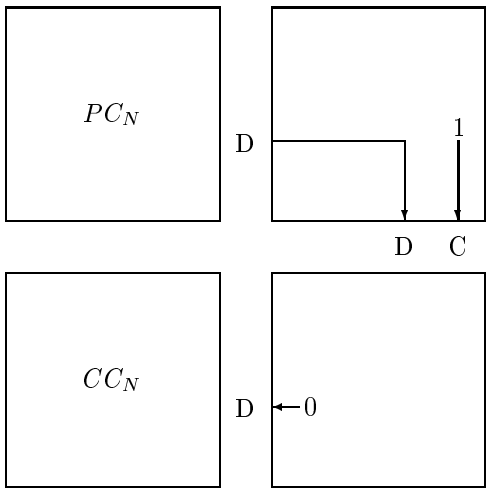


Figure 8
After First Step of
Wire Construction

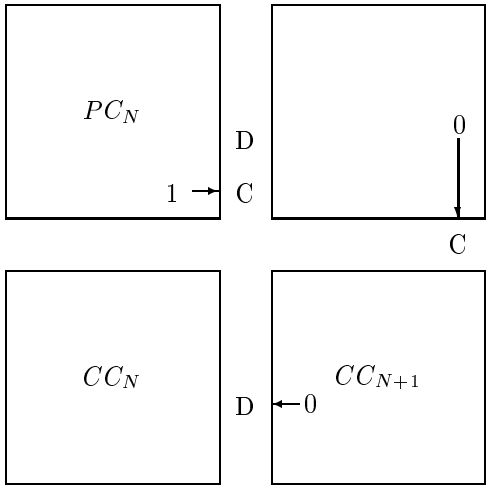


Figure 9
After Second Step of
Wire Construction

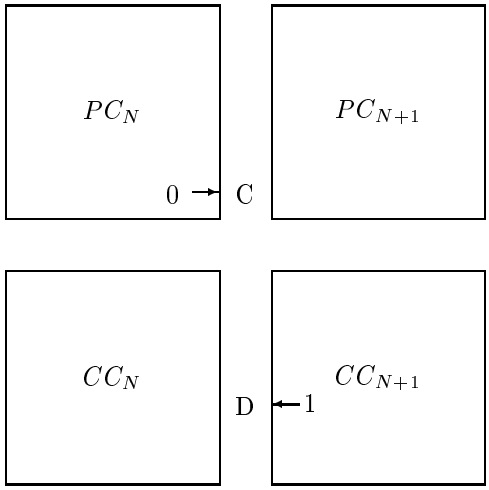


Figure 10
Final Configuration

PC_N	1,3
CC_N	2

Step 1: CC=1,PC=$\{DS = W$
$CS = 1$

Step 2: CC=0,PC=$\{\begin{matrix} DW = N \\ DE = W \end{matrix}\}$
$DN = W \cdot \bar{E}$

Step 3: CC=1,PC=$\{\begin{matrix} DW = E \\ DE = W \\ DS = 1 \\ CE = S \end{matrix}\}$

Figure 11
Sequence for Building EASTERN Wire

CC_N	PC_N
2	1,3

Step 1: CC=1,PC=$\{\begin{matrix} DW = N \\ CW = 1 \end{matrix}\}$

Step 2: CC=0,PC=$\{\begin{matrix} DN = E \\ DS = N \\ DE = N \cdot \bar{S} \end{matrix}\}$

Step 3: CC=1,PC=$\{\begin{matrix} DN = S \\ DS = N \\ DW = 1 \\ CS = W \end{matrix}\}$

Figure 12
Sequence for Building SOUTHERN Wire

4.6 Other Sequences

Next we illustrate two additional wire building sequences. Figure 12 shows how to extend a SOUTHERN wire, and is no different from the EASTERN example (things are just rotated).

Figure 13 lists the steps for building a coner, which changes a SOUTHERN wire into a EASTERN one, and Figure 14 shows the final configuration of cells in the corner.

As noted earlier, the cell in front of the CC head cell should be clear before extending the wire, or the backward "become a router" signal may already be present. This assumption can be avoided by using a *pre-clearing step*. A southern sequence which includes this step is shown in Figure 15.

So as long as we start with clear cells mmediately in front of the head cells, we can use this procedure to keep clearing space in front of us. After each extension, the cell beyond the CC head cell is clear, which means that the CC head cell will truly be acting as a head cell, not as a router.

The only case in which this clearing procedure will fail is if we try to program a cell which is already being programmed (in which case we are sharing control of the cell with one or more of its neighbors). With proper grid management, this should never happen.

CC_N	PC_N	
2,6	1,3,5, 7,9	New T
4	8	

Step 1: CC=1,PC=$\{\begin{matrix} DW = N \\ CW = 1 \end{matrix}\}$

Step 2: CC=0,PC=$\{\begin{matrix} CS = 1 \\ DS = S + E \end{matrix}\}$

Step 3: CC=1,PC=$\{\begin{matrix} DW = N \\ CW = 0 \end{matrix}\}$

Step 4: CC=0,PC=$\{DE = N\}$

Step 5: CC=1,PC=$\{\begin{matrix} DW = N \\ CW = 1 \end{matrix}\}$

Step 6: CC=0,PC=$\{\begin{matrix} DS = N \\ DN = E \end{matrix}\}$

Step 7: CC=1,PC=$\{\begin{matrix} CS = 1 \\ DS = N \end{matrix}\}$

Step 8: CC=0,PC=$\{\begin{matrix} DE = W \\ DN = W \cdot \bar{E} \end{matrix}\}$

Step 9: CC=1,PC=$\{\begin{matrix} DW = 1 \\ DE = N \\ CE = S \end{matrix}\}$

Figure 13
Sequence for Turning from SOUTH to EAST

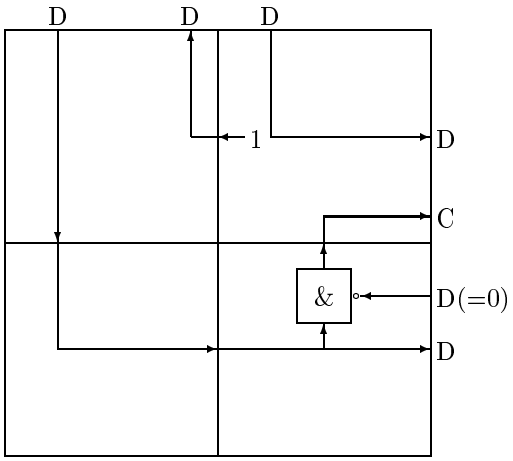


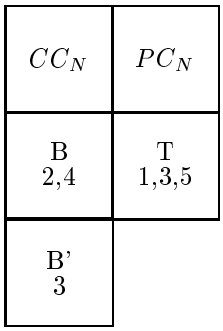
Figure 14
Final Configuration
After Corner Construction

4.7 Breaking A Wire

Generally the wires we are building are intended only for temporary use; if you want a permenant wire, you may as well just program a line of cells when you first load the grid. Our main interest is in being able to build wires from one cell to another, to allow us to replicate a cell. Once this is done, the wire is no longer needed and, in fact, is most likely in the way. How do we move the head of the wire **back** to the beginning? The solution is to use a *break* sequence. This is a special sequence which operates on CC_0, the CC cell at the beginning of the wire. To turn CC_0 and PC_0 back into head cells, we set $CC_0[DW_{in}]$ HI (assuming a SOUTHERN wire here). CC_0 is different from the other CC cells, in that if its DW_{in} goes HI, it outputs 0 on DS_{out} and 1 on CS_{out}. This will clear CC_1, which removes the "become a router" signal and returns CC_0 to its head cell status. Our pre-clearing step will now take care of continually clearing a path in front of the control channel.

4.8 Here We Are: So What?

Well, now we can build a wire to an arbitrary target cell in the grid, and we can control that cell's C and D lines. So what do we do now? Generally, when we are building a wire to some particular section of the grid, it is because we wisht to read or write some cell's program near that point. While we have direct control over the cell to the east of the PC head cell (assuming we're building



Step 1: CC=1,PC=$\{\begin{matrix} DW = N \\ CW = 1 \end{matrix}\}$

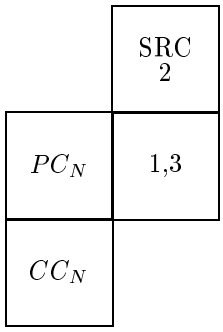
Step 2: CC=0,PC=$\{\begin{matrix} CS = 1 \\ DS = 0 \end{matrix}\}$
This programs B to clear B'

Step 3: CC=1,PC=$\{\begin{matrix} DW = N \\ CW = 1 \end{matrix}\}$
This doesn't change T's program, but it **does** put T into program mode, which drops its C_{out} lines, which allows B to assert **its** CS_{out} to clear B'

Step 4: CC=0,PC=$\{\begin{matrix} DN = E \\ DS = N \\ DE = N \cdot \bar{S} \end{matrix}\}$

Step 5: CC=1,PC=$\{\begin{matrix} DN = S \\ DS = N \\ DW = 1 \\ CS = W \end{matrix}\}$

Figure 15
Sequence for Building SOUTHERN Wire
With Pre-Clearing Step



Step 1: CC=1,PC=$\{\begin{matrix} CN = 1 \\ DN = N \\ DW = N \end{matrix}\}$

Step 2: CC=0; READ FROM PC

Step 3: CC=1,PC=0

Figure 16
NE Read Sequence

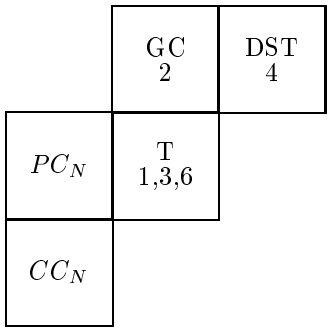
an EASTERN wire), it turns out that if we read and write a cell **to the side** of the wire, rather than one directly in the wire's path, we can extend the wire without disturbing that cell. This means we can read or write a line of cells without having to completely rebuild the wire each time. **This means that the process of reading or writing a line of N cells is truly order N.**

Since we are reading a cell to the side of the one we can directly control, we will have to do a little bit of leap-frogging at the head of the wire, i.e. we will need a sequence. Figure 16 illustrates a sequence for reading the cell to the north east of PC_N (SRC). The actual read occurs during Step 2, when the SRC cell places its data on the program channel. Step 3 is simply clearing out the T cell, but in reality, the wire could be automatically extended after the read, beginning with Step 3.

4.8.1 Guarded Transfers

When transfering cells, we must be concerned about the context in which that cell is programmed to exist, and consider what it will do out of that context. In other words, it's time to worry about live cells again. We adopt the convention [4]

[4] Many of the conventions we will use are not strictly necessary, but are designed with an eye towards the self-replicating structure of Chapter 6.



Step 1: CC=1,PC=$\{\begin{matrix} DN = W \\ CN = 1 \end{matrix}$

Step 2: CC=0,PC=$\{\begin{matrix} DE = E + S \\ CE = 1 \\ DS = E \end{matrix}\}$

Step 3: CC=1,PC=$\{\begin{matrix} DN = W \\ CN = 0 \end{matrix}$

Step 4: CC=0,PC=SOURCE PROGRAM

Step 5: CC=0,PC=0

Step 6: CC=1,PC=0

Figure 17
Guarded Cell Write

that **whenever an active guard cell is being used, it should be located to the east of the cell it's guarding**. We will say more about the nature of these guard cells in Chapter 6, but for now, let's just talk about sending a live cell down the wire and writing it to the northeast of the head of the wire.

There is a six step sequence which will create a guard cell directly to the northeast of the head, and then copy the live cell to the north or east of the guard cell. The idea is to place the guard cell on the grid first, and then load the live cell **using the guard cell as a programmer** as well as a guard. Figure 17 shows this sequence.

Step 1 gives a path (through T) to allow us to program the guard cell (GC), and Step 2 actually loads the guard cell's program (which allows programming of DST by ORing DS_{in} with DE_{in}).

During Step 3, we are programming T, so T's C_{out} is forced low, and it's

DN_{out} will be latched. If a HI value is latched, then the DST will be filled with all 1's (the most feared of all possible live cells!). We need a LO value latched, which will not affect DST. What value will be latched then? It depends on the outcome of a race. If the new CC value arrives before the start of the new PC stream, then we will latch the last bit of the **previous** PC stream. If the PC stream arrives first, it will be the first bit of the current PC.

It turns out the former is generally preferable (because of the truth table organization, the first program bit is CE_{out} for N=S=E=W=0, which seems likely to be 0; the last program is DW_{out} for N=S=W=E=1, which seems likely to be 1). Therefore, we rig the race so that the new CC data arrives **before** the new PC data[5].

Step 4 actually sends in the program, which is loaded into DST by the guard cell. In Step 5, we would like to reprogram T, but here the problem of the latched bit arises again. This time, it's the last bit of DST which will be latched and OR'd by GC. Since we can not make assumptions about the nature of the program being copied (remember it's **supposed** to be a live cell), we use a dummy programming cycle, sending out all 0s, so that the last bit out is guarenteed to be LO; this is what step 5 is for. Step 6 again clears out T.

Note that this style of guard cell constantly presents its cell's program on DS_{out}, available for reading by the method described in Figure 16.

4.9 Sequence Generator

We conclude this chapter with a brief description of the *sequence generator*. The concepts presented here will be used extensively in the next chapter, though the particular design described in this section will be modified extensively. I am including this design because it is simpler to understand, and captures the spirit of what the sequence generator should do. The version used for the self replicating structure is enhanced for reasons of practicality, but is more special purpose as a result.

We have seen that things such as wire building involve sequences of PC and CC data, applied in a particular order. Sequences come in different lengths, and the PC and CC values depends entirely on what the sequence is doing. It is therefore handy to have a structure which will generate the appropriate CC and PC data, in the appropriate order, i.e., to generate sequences. It turns out that with a few flip flops and a little control circuitry, we can turn a hardware library in to such a sequence generator.

Basically, we store each step of each sequence in a hardware library, we send a *sequence number* into the most significant bits of the select bus, and we

[5] I'm afraid I know of no elegant general solution to this problem. if the bit value is wrong, one can sometimes work with the inverted data, i.e., send out the complimented PC stream, and have rigged things to invert the data elsewhere. It seems easier to just keep the race in mind and force the last program bit of such sequence steps to be 0

attach a counter to the least significant bits. On an external **GO** command, the counter is cleared, and clocking is enabled. There is a 128-tick counter constantly running, and the next time it ticks, the counter is incremented, signaling **step 1** [6] of the requested sequence. The appropriate PC data is retrieved from the hardware library, and passed down the PC channel of our wire.

After that programming cycle is completed, the 128er tick will increment the counter again, thus signaling **step 2** of the indicated sequence (the post significant bits still indicate the sequence number, so step 2 of **that** sequence will be selected).

Two questions arise. First, what about the control channel data? We need to indicate, for each step, whether to send HI or LO down to CC. Second, how do we know when we've reaced the end of the sequence?

The answer is to pass a *strobe* signal backwards along the rows of the hardware library (recall the layout of the standard hardware library, as shown in Listing 2). The select lines run from west to east, passing through a set of match columns. Each column is composed of a number of match cells, which indicate whether the bit on their row indicates a match for that column. To pass back additional data, we simply have the match cells pass data from DE_{in} to DW_{out}, **unless they are in the matching column**, in which case they send a fixed value (LO or HI), depending on the nature of the cell in that row.

This means that when a column matches the value on the select lines, we can pass back not only the PC data specified by the corresponding source cell, but **up to 8 bits of boolean information specific to that sequence number and that step**.

We thus have the answers to both our questions. We specify that the 6th row will pass back the value for the CC channel, while the 7th row (bottom) will pass back **EOS** (End Of Sequence). The controlling hardware uses this signal to disable the clocking, thus the counter remains fixed at the EOS step, and no further information is sent (assuming the CC and PC data corresponding to the EOS column are specified as 0).

To implement this strobe, we create two new match cells, called **sm0** and **sm1**. When these cells are in a matching row, they pass a HI value out DW_{out}, which is routed all the way back to the west end of the library.

We thus have a system which allows us to specify CC and PC data to be sent down the channels, in any order we desire, for as many steps as we desire[7]. The system accepts a **GO** signal, generates the sequence, and indicates completion by raising its **DONE** line.

Listings 3 and 4 (at thge end of this chapter) show the code for this sequence generator. Listing 5 shows a *load file*, which is used for managing multiple

[6] because the counter may be cleared at any point during the 128 tick programming cycle, we always begin our sequences with step 1

[7] Our sequences can technically only be 15 steps long, but since the sequence number is connected to the counter, the 16^{th} step of sequence N will simply look like Step 0 of Sequence N+1; as long a we don't assert EOS, we can cascade the two sequences together

source files. The load file simply tells where each subgrid should be loaded in a supergrid.

This sequence library is not complete, but contains a few representative sequences. The **GO** line is actually two inputs, DN_{in} on [0,0] and [0,1]. The sequence number is specified on DW_{in}[0,0] (MSB) thru DW_{in}[3,0] (LSB), and the DONE line is output on DW_{out}[9,0]. Additionally, DW_{in}[9,0] is connected to the CC head cell, i.e., it controls C_{in} on the target cell, while DW_{in}[10,0] is connected to the PC head cell, i.e., it controls D_{in} on the target cell. DW_{out}[10,0] delivers any output from the target cell, i.e., is connected to T[D_{out}]. Finally, $DW_[11, 0]$ is the **BREAK** line, and connects to CN_{out} on CC_0. Asserting this line for 128 clock ticks will break the wire, returning the head to CC_0 and PC_0 as described previously.

This completes our discussion of wire building techniques. We are now ready to discuss the actual sequencer, which directs the sequence generator according to commands received from a stored program. Once this mechanism is in place, we need only write the correct sequencing program to copy cells from one point to another. In Chapter 6, we will develop such a program to replicate the entire sequencer and memory system, along with a little [8] additional hardware called the *exploded grid*.

[8] actually "little" is not entirely accurate; the extra structure will account for approximately 85% of the final system!

```
; sequence generator, misc. control hardware

.def plff0 ; pre-loadable flip flop
DE=NW~& EW~& EN~& ++ ; DW=INPUT
DS=N ; DN=LOAD (SET then CLEAR)
.end
;-----

.def plff1
DE=S~W~& NW~& EN~&S& ++ ; DN=LOAD (SET then CLEAR)
DN=E ; DS=CLK in
DS=N
DW=S~W& NW& EW& EN~&S& +++
.end
;-----

.def plff2 ; DE=OUTPUT
DE=W
DS=N
DW=W
.end
;-----

.def plff2s ; special, for 6th column
DE=W
DS=E ; Route EOS
DW=W
.end
;-----
```

Listing 3
Sequence Generator Control
(Select Lines)

```
; Misc. routing here (gross)

.def xtal
CE=E~N~S~W~ &&& ; crystal
; this MUST be loaded ready to run,
; since we use active guared cells
; in the HW library

.end
;-----

.def xctl
CW=1
DN=E~W&
DS=E
DW=W
.end
;-----

.def R9C2
DS=WN+
.end
;-----

.def R10C3
DS=WN+
DW=S
.end
;-----

.def Bcc ; Breakable CC cell
CS=W
DE=NS~&
DN=E
DS=N
.end
;-----

.def pc ; PC cell
CS=W
DN=S
DS=N
DW=1
.end
;-----
```


Listing 3
Sequence Generator Control (cont)
(Routing)

```
.macro
- ../mac/ew.we ; wire
_ ../mac/ew.we.1n ; wire
T ../mac/es.we ; pass EOS into routing block
+ ../mac/ew.ns.sn.we
| ../mac/ns.sn
/ ../mac/es.nw
J ../mac/nw.we
.end

; layout map

.name seq-ctrl.bin
.size 12x4
plff0 plff1 plff2     -
plff0 plff1 plff2     -
plff0 plff1 plff2     -
plff0 plff1 plff2     -
plff0 plff1 plff2     -
plff0 plff1 plff2     -
plff0 plff1 plff2s    -
plff0 plff1 plff2     T
 xtal  xctl     +     /
    _     J  R9C2     | ; _ is special, sends 1^DN
    -     -     + R10C3
    -     -   Bcc    pc
```

Listing 3
Sequence Generator Control (concluded)
(Layout Map and Macros)

```
;
; Sequence library
;
.def E1 ; These are for EASTERN chnnel
       CS=1
       DS=W
.end
;------

.def E2
       DE=W
       DN=E~W&
       DW=N
.end
;------

.def E3
       CE=S
       DE=W
       DS=1
       DW=E
.end
;------
```

Listing 4
Sequence Generator Library
(EASTERN Sequence)

```
.def c1 ; These are for turn from SOUTH to EAST
       CW=1
       DW=N
.end
;------

.def c2
       CS=1
       DS=SE+
.end
;------

.def c3
       DW=N
.end
;------

.def c4
       DE=N
.end
;------

.def c5
       CW=1
       DW=N
.end
;------

.def c6
       DN=E
       DS=N
.end
;------
.def c7
       CS=1
       DS=N
.end
;------
```

Listing 4
Sequence Generator Library (cont)
(CORNER Sequence)

```
.def c8
       DE=W
       DN=E~W&
.end
;------

.def c9
       CE=S
       DE=N
       DW=1
.end
;------
; Basic SOUTHERN channel


.def cc ; CONTROL channel
       DE=NS~&
       DN=E
       DS=N
.end
;------

.def gc1 ; (guard cell)
        CS=1
        DE=W
        DW=E
.end
;-------

.def p1 ; Used to load CC
       CW=1
       DW=N
.end
;------

.def pc ; PROGRAM channel
       CS=W
       DN=S
       DS=N
       DW=1
.end
;------
```

Listing 4
Sequence Generator Library (cont)
(CORNER Sequence (concluded) and SOUTHERN Sequence)

```
.def rl ; Read NORTH EAST
       CE=1
       DE=E
       DN=E
.end
;------

.def g1 ; Sequence 5: GUARDED TRANSFER
       CE=1
       DE=N
.end
;------

.def g2
       CS=1
       DS=WS+
       DW=S
.end
;------

.def g3
       DE=N
.end
;------
;
; blocks for matching seq #/step #, etc
;
.def ctl
        CS=N
        DN=N
        DS=NS
        DW=NS& EN~&  +
.end
;-------
.def gtl ; guarded ctrl cell; keeps target rotating
        CS=1
        DN=N
        DS=S
        DW=NS& EN~&  +
.end
;-------
```

Listing 4
Sequence Generator Library (cont)
(GUARDED READ Sequence and ctl cells)

```
.def m0
       DE=W
       DN=S
       DS=NW~&
       DW=E
.end
;------

.def m1
       DE=W
       DN=S
       DS=NW&
       DW=E
.end
;------

.def tm0
        DE=W
        DN=S
        DS=W~
        DW=E
.end
;-------

.def tm1
        DE=W
        DN=S
        DS=W
        DW=E
.end
;-------
```

Listing 4
Sequence Generator Library (cont)
(Matching Section)

```
.def sm0
        DE=W
        DN=S
        DS=NW~&
        DW=SE+
.end
;-------

.def sm1
        DE=W
        DN=S
        DS=NW&
        DW=SE+
.end
;-------


.macro
- ../mac/ew.we
^ ../mac/sn
V ../mac/ns
| ../mac/ns.sn
+ ../mac/ew.ns.sn.we
. ../mac/0
L ../mac/en.ne.ns
J ../mac/ns.nw.wn
\ ../mac/sw.we.ws
/ ../mac/ns.se
' ../mac/wn
T ../mac/ew.we.ws
[ ../mac/en.ew.ns.we

} ../mac/es.sw
U ../mac/ew.nn.we ; used for feedback
.end
```

Listing 4
Sequence Generator Library (cont)
(Matching Section and Macros)

```
; Sequence library for basic routing
; Seq #0 is dummy EOS
; Seq #1 is SOUTH
; Seq #2 is TURN from SOUTH to EAST
; Seq #3 is EAST (removed)
; Seq #4 is READ SOUTH EAST (removed)
; Seq #5 is GUARDED READ

; The top 8 rows form the match columns, so they're routed
; vertically on the left. The bottom of the match columns
; pass back EOS to the left, so they're routed through the
; feedback paths in the lower blocks along the right.
;
; The match columns have tm0 or tm1 on top, and m0, m1, sm0 or sm1
; down each one. The last digit indicates a piece of the pattern
; that column is matching; for this system, this pattern from top to bot is:
; Seq #(MSB), Seq #(B2), Seq #(B1), Seq #(LSB),
; Step #(MSB), Step #(B2), Step #(B1), Step #(LSB)
; sm vs. m indicates that if that columnm matches, a trobe
; should be sent down the corresponding row. In this system,
; the "ctl" row passes back programming data, the bottom row is the
; END OF SEQUENCE strobe, next is the CONTROL-CHANNEL strobe, and next the
; SPECIAL strobe (for HW transfers)
```

Listing 4
Sequence Generator Library (cont)

```
.size 23x27
.name seq-lib.bin
; here's sequences #2 and #0

 --------T --   tm0 tm0 tm0 tm0   - tm0 tm0 tm0 tm0 tm0 tm0   - tm0    ...
 -------T+ --    m0  m0  m0  m0   -  m0  m0  m0  m0  m0  m0   -  m0    ...
 ------T++ --    m1  m1  m1  m1   -  m1  m1  m1  m1  m1  m1   -  m0    ...
 -----T+++ --    m0  m0  m0  m0   -  m0  m0  m0  m0  m0  m0   -  m0    ...
 ----T++++ --    m0  m0  m0  m0   -  m0  m0  m0  m1  m1  m1   -  m0    ...
 ---T+++++ --    m0  m0  m0  m1   -  m1  m1  m1  m0  m0  m0   -  m0    ...
 --T++++++ --   sm0  m1 sm1  m0   - sm0  m1 sm1  m0 sm0  m1   -  m0 --\
 -T+++++++ --    m1  m0  m1  m0   -  m1  m0  m1  m0  m1 sm0   - sm1 -\|
; ||||||||                                                            ||
 -++++++++ }gc1 ctl gtl ctl ctl gc1 ctl ctl gtl ctl ctl ctl ctl ctl \||
 .VVVVVVVV +   .  c1  c2  c3  c4   .  c5  c6  c7  c8  c9   .   .   . |||
; ||||||||                                                            |||
; ||||||||                                                            |||
; here's seq. 1 & 5                                                   |||
; ||||||||                                                            |||
 .|||||||L +-   tm0 tm0 tm0 tm0  tm0 tm0 tm0 tm0 tm0 tm0 tm0 tm0 tm0^^^
 .||||||L+ +-    m0  m0  m0  m0   m1  m1  m1  m1  m1  m1  m1  m0  m0|||
 .|||||L++ +-    m0  m0  m0  m0   m0  m0  m0  m0  m0  m0  m0  m0  m0|||
 .||||L+++ +-    m1  m1  m1  m1   m1  m1  m1  m1  m1  m1  m1  m0  m0|||
 .|||L++++ +-    m0  m0  m0  m0   m0  m0  m0  m0  m0  m0  m0  m0  m0|||
 .||L+++++ +-    m0  m0  m0  m1   m0  m0  m0  m1  m1  m1  m1  m0  m0|||
 .|||||||/ +-     +   +   +   +    +   +   +   +   +   +   +   +   +++'
 .|L+++++[ +-   sm0  m1 sm1  m0  sm0  m1 sm1  m0  m0 sm1  m1  m0  m0|| .
 .|||||||/ +-     +   +   +   +    +   +   +   +   +   +   +   +   ++' .
 .L++++++[ U-    m1  m0  m1 sm0   m1  m0  m1  m0  m1  m0 sm1 sm1 sm1|  ..
; ||||||||                                                            |  ..
 .|||||||/ --     +   +   +   +    +   +   +   +   +   +   +   +   +'  ..
 .|||||||L -gc1 ctl ctl ctl ctl  gtl gtl ctl gtl ctl ctl ctl ctl ctl   ...
 .VVVVVVVV -   .  p1  cc  pc   .   g1  g2  g3 SRC   .   .   .   .   .   ...
```

Listing 4
Sequence Generator Library (concluded)
(Layout Map)

```
< seq-ctrl.bin 0 0
< seq-lib.bin 0 4
> seq.bin 0 0 23 31
Complete sequencer w/channel building sequences
```

Listing 5
Sequence Generator Load File