INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700   800/521-0600
The functional memory approach to the design of custom computing machines

Halverson, Richard Peyton, Jr., Ph.D.

University of Hawaii, 1994
THE FUNCTIONAL MEMORY APPROACH TO THE DESIGN OF CUSTOM COMPUTING MACHINES

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMMUNICATION AND INFORMATION SCIENCES

AUGUST 1994

By

Richard Peyton Halverson, Jr.

Dissertation Committee:

Art Lew, Chairperson
W. Wesley Peterson
William E. Remus
Dan J. Wedemeyer
Edward J. Weldon, Jr.
ACKNOWLEDGEMENTS

I would like to thank the members of my committee for their service. I would also like to thank the others who made this dissertation possible. Ray Panko provided my primary financial support throughout my years as a Ph.D. student. Dan Watanabe provided the initial funding for the hardware. Dominic McCarthy gave me the ideas for Chapter 7. On a personal level, I would like to thank my father Richard Sr., my late mother Sau Hung Young Halverson, and my wife Christine David.

I would also like to thank my chair Art Lew for helping me out as much as he did, especially with this final document. The detail, comprehensiveness, sense and time he spent was much appreciated.
ABSTRACT

This dissertation describes a software system and related hardware architecture in which high level language programs are compiled into gate level logic circuitry that is configured specifically to execute the compiled program. A system whose processor can be dynamically reconfigured to suit different applications is known as a custom computing machine (CCM). We have designed a new class of CCMs based on the concept of functional memory (FM), which we construct by connecting field programmable gate arrays (FPGAs) in parallel with conventional random access memory (RAM). FM is used by the processor for computing the (possibly multi-operand) expressions of the high level language program in the combinational logic provided by the FPGAs. When all program expressions are computed in FM, the necessary processor instruction set reduces to a minimal number of moves and jumps.

Our functional memory computer (FMC) is a four FPGA FM prototype with a fifth FPGA programmed as the minimal processor. The language we adopted as the high level source language for programming the FMC is a decision-table (DT) variation of standard Pascal. DT programs for a shortest path and two sorting algorithms were translated, executed, and analyzed on the FMC. The second sorting program demonstrated a nondeterministic array selection function. An analysis for the shortest path program showed that memory load/store counts remained comparable for FMC and von Neumann implementations. However, with the FMC, a 35% reduction in total execution steps occurred because all computation steps are performed in parallel on the FMC.

The problem of compiling high level DTs to low level FMC object code is more complex than for conventional machines because each single expression in the source program can translate into several tens of lines of FPGA circuit definition code. The
Windows based system developed for this purpose includes a compiler that translates source programs into intermediate assembly language modules, and an operating system that invokes system routines for assembling, linking, placing and routing, and loading the FPGA machine level object code into the minimal processor and functional memory.
# Table of Contents

Acknowledgements ................................................................................ iv
Abstract ................................................................................................. v
List of Tables ......................................................................................... xiii
List of Figures ......................................................................................... xiv
Preface ....................................................................................................... xviii

Chapter 1. Introduction ........................................................................... 1
  1.1 Decision Table Programming Model ............................................. 2
  1.2 Hardware Background ................................................................. 4
    1.2.1 HPC Cache-Only Multiprocessor .......................................... 5
  1.3 Levels of Programmability ............................................................ 7
    1.3.1 Von Neumann Machines ....................................................... 7
    1.3.2 Microprogrammable Machines ........................................... 8
    1.3.3 Custom Computing Machines .......................................... 11
  1.4 Survey of Custom Computing Machines ................................. 12
    1.4.1 DEC's Perle-0 ................................................................. 13
    1.4.2 PRISM-II ................................................................. 13
    1.4.3 Reconfigurable Processor Unit ........................................ 13
    1.4.4 MoM-4 Xputer ............................................................ 14
    1.4.5 Splash 2 ................................................................. 14
    1.4.6 AnyBoard ................................................................. 15
    1.4.7 Chameleon ................................................................. 15
  1.5 The Functional Memory Approach ............................................. 16
    1.5.1 Functional Memory ......................................................... 16

vii
4.3 Conclusions ................................................................. 97

Chapter 5. Analyses of Execution ........................................... 99

5.1 Time Comparisons .......................................................... 99
  5.1.1 Shortest Path Execution Times ................................... 100
  5.1.2 Deterministic Bubble Sort Times ............................... 101
  5.1.3 Nondeterministic Bubble Sort Times ......................... 103

5.2 Cycle Count Comparisons ............................................... 104
  5.2.1 Shortest Path Cycle Counts .................................... 105
  5.2.2 Deterministic Bubble Sort Cycle Counts .................... 107
  5.2.3 Nondeterministic Bubble Sort Cycle Counts ............... 108

5.3 Load–Store Comparisons ............................................... 110
  5.3.1 Shortest Path Load–Store Analysis ........................... 111

5.4 Conclusion .................................................................. 114

Chapter 6. The Design and Implementation of a Compiler for a Functional
Memory Computer .............................................................. 116

6.1 The Compiling Process .................................................. 116
  6.1.1 Generate Intermediate Text (1.0) ............................... 121
  6.1.2 Generate Memory Map (2.0) .......................... 123
  6.1.3 Generate Microcode Source Module (3.0) .................. 125
  6.1.4 Generate FPGA Source Modules (4.0) ....................... 125
    6.1.4.1 Expand Function Macros (4.2.2) ...................... 128
    6.1.4.2 Generate Condition Stub Logic (4.2.3) .......... 129
    6.1.4.3 Parsing Expressions (4.2.3.1) ..................... 130
    6.1.4.4 Generate Rule Address Logic (4.2.4) .......... 133
    6.1.4.5 Generate Action Stub Logic (4.2.5) ............ 134
6.1.4.6 Generate Input Register Logic (4.2.6) .......................... 135
6.1.4.7 Generate Output Multiplexer Logic (4.2.7) .............. 136
6.1.4.8 Generate Address Select Logic (4.2.8) ..................... 137

6.2. The User Interface ........................................................................ 138
   6.2.1 File Menu ............................................................................. 139
   6.2.2 View Menu ........................................................................... 140
   6.2.3 Generate Menu ...................................................................... 142
   6.2.4 Compile Menu ....................................................................... 145

Chapter 7. Application: Examples in Image Processing ..................... 147
  7.1 Convolution ............................................................................... 148
    7.1.1 Image Magnification .............................................................. 150
    7.1.2 Pyramid Special Function Implementation ....................... 151
    7.1.3 Magnification Program Example ............................................ 154
    7.1.4 Magnification Execution Comparison ................................... 156
  7.2 Histograms .................................................................................. 157
    7.2.1 Character Classification ........................................................ 157
    7.2.2 Row-Column Sum Special Function ................................. 158
    7.2.3 Histogram Program Example ............................................... 160
    7.2.4 Row Column Histogram Computation Comparison ............ 161

Chapter 8. Conclusions ........................................................................ 162
  8.1 Decision Table Computers ............................................................ 163
  8.2 Custom Computing Machines ..................................................... 163

Appendixes. Program Listings .............................................................. 165
  Appendix 1. Bubble Sort Compiler Listing (BUBBLE.LIS)........... 165
  Appendix 2. Bubble Sort Minimal Processor Code (BUBBLE.ASM) . 167
LIST OF TABLES

Table 2.1. Minimal Processor Instruction Set..........................52
Table 2.2. Microinstructions for Implementing mP Operations........56
Table 3.1. Bubble Sort Functional Memory Map...........................65
Table 3.2. Bubble Sort Execution Table..................................66
Table 3.3. Bubble Sort Rule Selection Logic..............................67
Table 3.4. FPGA Compilation Statistics - Deterministic Bubble Sort....74
Table 3.5. Shortest Path Functional Memory Map...........................77
Table 3.6. Shortest Path Execution Table ..................................78
Table 3.7. Shortest Path Rule Address Map ................................79
Table 3.8. Shortest Path FPGA Compilation Statistics .....................84
Table 4.1. Nondeterministic Bubble Sort Functional Memory Map........95
Table 4.2. Nondeterministic Bubble Sort Execution Table................95
Table 4.3. Nondeterministic Bubble Sort FPGA Compilation Statistics...97
Table 5.1. Shortest Path Execution Times ................................100
Table 5.2. Bubble Sort i486 Execution Times ..............................102
Table 5.3. Deterministic Bubble Sort FMC Execution Times ...............103
Table 5.4. Nondeterministic Bubble Sort FMC Execution Times ..........104
Table 5.5. Shortest Path Cycle Counts ....................................106
Table 5.6. Bubble Sort Cycles Comparison (Random Array) ...............110
Table 5.7. Shortest Path Load/Store Count Comparison ...................115
LIST OF FIGURES

Figure 1.1. Decision Table Representation of a Program.............................1
Figure 1.2. Decision Table Execution.....................................................3
Figure 1.3. Decision Table Execution Model .............................................4
Figure 1.4. Hawaii Parallel Computer Block Diagram.................................6
Figure 1.5. Compiler for a von Neumann Computer ......................................8
Figure 1.6. Compiler for a Microprogrammable Processor .............................9
Figure 1.7. Burroughs Small Reconfigurable Processor.................................10
Figure 1.8. FPGA Based Reconfigurable Processor .....................................12
Figure 1.9. Functional Memory Concept....................................................17
Figure 1.10. Functional Memory Computer and Instruction Set .....................18
Figure 1.11. Compiling for a FMC............................................................20
Figure 1.12. Categorizing Hybrid Data Flow Computers ............................21
Figure 1.13. System for Demonstrating the Functional Memory Approach .......24
Figure 2.1. Implementing Functional Memory – Multiplexer Method ............27
Figure 2.2. Implementing Functional Memory – Logical OR Method ................28
Figure 2.3. Functional Memory FPGA Interface Circuit Shell ......................30
Figure 2.4. Address Select PALASM.........................................................33
Figure 2.5. Input Register PALASM...........................................................33
Figure 2.6. Output Multiplexer PALASM..................................................34
Figure 2.7. $A \neq B$ Test PALASM.............................................................36
Figure 2.8. $A = 5$ Test FPGA Equations..................................................37
Figure 2.9. $A < B$ PALASM.................................................................38
Figure 2.10. $D = A + B$ Equations.........................................................39
Figure 2.10 (continued). \( D = A + B \) Equations.................................40
Figure 2.11. Condition Entries Selecting Rule and Rule Starting Addresses......41
Figure 2.12. Rule Jump Address Calculation Example ................................42
Figure 2.13. Eight Bit \( A = A + B0 + B1 \) Accumulator PALASM ...............44
Figure 2.14. 16 Bit Auto Shift Register – Shift Left Eight Bits .....................45
Figure 2.15. Functional Memory Computer Board Block Diagram .................46
Figure 2.16. Architecture of the Minimal Processor ..................................48
Figure 2.17. Microinstruction Control Register .........................................54
Figure 2.18. Minimal Processor - Detailed Block Diagram ............................57
Figure 2.19. System Processor - Minimal Processor Interface ........................58
Figure 2.20. System Monitor - User Interface ...........................................59
Figure 3.1. Bubble Sort Program .............................................................62
Figure 3.2. Bubble Sort Flow Chart .........................................................63
Figure 3.3. Bubble Sort Program for FM ..................................................64
Figure 3.4. Deterministic Bubble Sort Expression Graphs ............................68
Figure 3.5. FPGA Operator Macros Used for Deterministic Bubble Sort .......70
Figure 3.6. FPGA Code Generation for \( j+1 \) ..............................................71
Figure 3.7. Completion of the Compilation Process ....................................73
Figure 3.8. Shortest Path Program ............................................................75
Figure 3.9. Shortest Path Rule Address Computation .................................81
Figure 3.10. Shortest Path Expression Computation .................................82
Figure 4.1. \( O(1) \) Set Operators ..............................................................88
Figure 4.2. \( O(1) \) Minimum Array Element Selection ................................90
Figure 4.3. Nondeterministic Bubble Sort Program ....................................92
Figure 4.4. Nondeterministic Bubble Sort Flow Chart ...............................92
Figure 4.5. NBS Exchange Address and Rule Selection .............................93
In 1986 an engineer friend of mine named Richard Shaffer and I stopped at a local vendor in Minneapolis to pick up some PALs, when the representative asked us into the back room to show us something. XILINX had just introduced the first RAM-based field programmable gate array (FPGA), which was the first very large scale integrated (VLSI) circuit chip where the circuit is “written into it” not at the factory but after the power is turned on. In the back room the rep showed us a XILINX 2018 FPGA development system. He asked, “Can you think of a good application for these things?” My reply was, “Only if you had a reason to change the circuit as often as you change RAM, like if the chip could somehow be reprogrammed from a high-level language compiler.” As my friend and I were driving back, I discussed with him how interesting it would be to write a compiler which would specify the logic circuitry of the machine that the program would execute on. In 1991 this became my dissertation topic.
CHAPTER 1. INTRODUCTION

This dissertation began as a challenge to build a machine designed expressly for decision table (DT) execution. One advantage of representing a program as a DT [CODASYL, 1982] is that all the program's conditional statements are consolidated as "condition stubs" in upper left quadrant of the table with the idea that they can all be evaluated simultaneously. (See Figure 1.1.) Based on the evaluation of all these condition stubs, a "rule" corresponding to one of the columns is chosen. The rule contains a list of statements to execute, which are made up of "action stubs" in the lower left quadrant of the table.

![Figure 1.1. Decision Table Representation of a Program](image)

A machine designed for DT execution would be one where the amount of time it takes to select a rule is independent of the number of condition stubs and rules. Such a machine would also be able to compute arithmetic expressions in constant time (within practical limits). This means that theoretically, the right sides of assignment statements would also evaluate in unit time, independent of the number of variables or operands. In the end, theoretical execution time for a program on such a machine would depend only
on (a) the number of iterations of the rules it must execute, plus (b) the number of variable changes (i.e., assignment statements) each rule contains.

If "constant time" is on the order of tens or hundreds of microseconds, then such a machine would only have theoretical value. If this time is on the order of tens or hundreds of nanoseconds, then the machine would have practical value today. Also, the machine would have only theoretical value if the sizes of implementable programs are too small to be of any significance, although "practical limits" of today are much different than what they will be five years from now.

For this dissertation, we have built a machine whose "constant times" are in the tens to hundreds of nanoseconds, and whose "practical limits" do not impair it from potentially impacting important applications (e.g., image processing) over the next five years. This machine utilizes a relatively new technology, that of field programmable gate array (FPGA [Brown, et al., 1992]) chips, to extend RAM (memory) to what we call "functional memory" (FM), which has attached expression computation (function processing) elements.

1.1 DECISION TABLE PROGRAMMING MODEL

The decision table is used as the high-level programming language to compile because its structure conveniently separates and consolidates the conditional expressions for program control into a single expression calculation. Since it has been shown previously that any computer program can be expressed in a decision table form [Lew, 1982], decision tables are general purpose. In fact, decision tables support multiway branching, making it even more expressive and efficient than conventional languages. For convenience, we will assume limited-entry unambiguous DTs in our treatment here.
As Figure 1.2 illustrates, a decision table consists of four quadrants. The upper left contains condition stubs, which are expressions that can be evaluated all at once. The upper right quadrant lists the condition entries, which define columns of possible expression result combinations. Multiple 'T' entries in a column indicate logically ANDed condition stubs which must be true for the "rule" (column) to "fire" (i.e., be selected for execution). The lower right quadrant contains the action entries that indicate row by row with X's, which action stub statements (in the lower left quadrant) are to be executed when the rule fires. Note that the right half of the table (the entry table) is simply an AND-OR array, containing boolean inputs and outputs. The process of translating any program (i.e., flowchart) into a decision table is mechanical and explained in [Lew, 1982].

![Decision Table Execution Diagram](image)

**Figure 1.2. Decision Table Execution**

A decision table program executes by first evaluating all the condition stubs simultaneously. The results feed the entry table logic which selects a unique rule. The
selection of the rule defines a subprogram entry point address. There, code for all the action stubs in a selected entry column is executed. The whole process repeats (starting with the reevaluation of the condition stubs) until a selected rule causes the program to terminate.

Figure 1.3 illustrates the decision table execution model. A rule is provided for the processor by the FPGA chip in the form of a jump address to code to execute for that rule. The code to execute involves simple fetching expression results from FM and storing them back into FM. When the processor is complete, the address of the next rule to execute is immediately available to be read and executed.

1.2 HARDWARE BACKGROUND

With a $1,000 department grant, the “Hawaii Parallel Computer” was designed and the parts were ordered. Construction quality was not compromised, but the absolute
cheapest most practical integrated circuit components were used, including the $3.95 Intel MCS-51 8031 microprocessor (i8031). With all these conditions to evaluate simultaneously, the use of a field programmable gate array (FPGA) seemed like a natural component to the solution. Evaluating any sort of array of boolean variables, such as the right half of a decision table, is inherently easier to implement in boolean logic than sequentially in assembly language on a microprocessor. Therefore the XILINX 3042 4,200 gate FPGA was ordered [XILINX, 1993], which at $125 was by far the most expensive single component.

The machine evolved over a period of two years. In the original design, the FPGA would evaluate only the right half of the decision table, with an array of microprocessors feeding it boolean inputs and reading a boolean vector output. The machine resembled a cache-only multiprocessor, because local two-port shared memories on each microprocessor board allowed the memory array as a whole to emulate a concurrent read exclusive write (CREW) Parallel Random Access Machine (PRAM) [Kumar, et al., 1994]. Our use of FPGAs also made our design significantly different from the only other “DT computer” [Pawlak 1986] of which we are aware. That system considers only the entries, and ignores condition and action stubs.

1.2.1 HPC Cache-Only Multiprocessor

The original HPC architecture featured an array of i8031 microprocessors connected to two-port RAMs implementing a “cache-only” multiprocessor. (See Figure 1.4.) A unique wire-ORed bus facilitated the cache update by the main microprocessor. The subordinate i8031 “von Neumann processor” array evaluated the condition stubs of the decision table. A special Boolean Processor (BP), which was simply the XILINX 3042 FPGA, “executed” the entry table. Finally, the “system processor” would execute the action stubs according to the BP output.
The BP produces an action stub vector from the condition stub results, indicating which action stubs to execute. Output bits (programmed as boolean expressions of input bits) are computed simultaneously, nondeterministically (asynchronously) and in dataflow fashion (using combinational logic). Execution time is instantaneous (tens or hundreds of nanoseconds) and (ideally) independent of number of input bits. The BP with I/O bits are implemented in the single XILINX FPGA and programmed using the Programmable Array Logic Assembly Language (PALASM).

When it came time to wire the XILINX board, five more, higher capacity FPGAs became available as samples from XILINX. With four 9,000 gate XC3090 and one 6,400 XC3064 FPGA, we were able to expand the portion of the program computed in the FPGA to all the condition expressions as well. (Up to this point, the condition expressions were to be evaluated using the array of microprocessors with their true/false results being fed to the FPGA.)
The HPC was first described publicly where the condition expressions and rule selection were computed in the FPGA array, and all the rule statements (action stubs) were executed in the main microprocessor [Halverson and Lew, 1994-1]. Our primary focus now assumes the case where the expressions on the right sides of the assignment statements are also computed in the FPGA array [Halverson and Lew, 1994-2].

1.3 LEVELS OF PROGRAMMABILITY

When programming machines, high level language statements are translated down into the native "ones and zeros" language codes of the machine. These machine codes tell the hardware what to do. The granularity of the specification detail of these machine language codes, in effect, define the hardware/software boundary and the level of programmability of the machine. Few bits per machine cycle (eight or 16) for von Neumann microprocessors are coarse grained. Microprograms often use less than 100 bits per cycle therefore are medium grained. FPGA specifications can require hundreds or thousands of bits defining a single cycle computation therefore would be fine grained. The high level language translation tools become increasingly complex the more fine grained the hardware/software boundary.

1.3.1 Von Neumann Machines

Digital computers as we know them today were conceived in the late 1940's. They were unique from computing machines of the past in that their programming involved setting switches electronically, using the same storing and fetching mechanism as the program itself uses for data as it is executing (i.e., program memory was the same as data memory). This meant that programs could now modify themselves, but more importantly, it meant that a program could create another program and execute it, without user intervention. Memory now contained either processor instruction codes or data, depending on the reference point of a particular program. One program's data output can
itself be a program, to be executed at a different time. Figure 1.5 illustrates the most common program whose output is itself a program: a compiler.

On a von Neumann processor, the instruction repertoire is a fixed set of operations that the processor is "wired" to carry out. The granularity of the program instructions involve for example (a) moving a word from the memory into a processor register, or (b) adding two registers together inside the processor. (A register is a one word memory element inside the processor. Processors usually have 8, 16 or 32 general purpose registers for storing values temporarily for calculations.) These program instructions may take multiple machine cycles to complete.

![Diagram of a von Neumann Computer with compiler](image-url)

**Figure 1.5. Compiler for a von Neumann Computer**

### 1.3.2 Microprogrammable Machines

In the 1970s, semiconductor memory prices dropped to a level where microprogrammable computers became feasible. At a lower level, microprogram instruction codes involve the multiplexing of signals between processor registers and through arithmetic and logic units to perform mathematical functions (e.g., addition, shifting, multiplication). The memory that contains the microprogram is usually separate...
from the memory that contains the program data, but not necessarily. The microinstruction codes themselves are generally longer (i.e., consist of more bits) than von Neumann processor instruction codes, because sets of bits are generally controlling several finer grained operations at once. Most machines are designed so microinstructions take exactly one machine cycle to complete. An example of a microcode crosscompiler like the one shown in Figure 1.6 would be one that would, for example, translate Pascal code into microcode for a particular microprogrammable processor where the compiler itself runs on a different computer, such as an IBM or a Mac.

![Microcode Crosscompiler Diagram]

Figure 1.6. Compiler for a Microprogrammable Processor

A microprogrammable processor can be microprogrammed to emulate a von Neumann computer that executes a particular instruction set (often called a "macroinstruction" set at this level). To the microprogram, the von Neumann instruction codes are simply input data (stored in the data memory) that define a particular sequence of microinstructions to execute. A microinstruction fetches a von Neumann (macro)instruction code, and depending on what it is, several more microinstructions execute to carry out the operation.
In the early 1970s, Burroughs (now Unisys) built the Small Reconfigurable Processor which took advantage of microprogramming for loading in different microcode for different instruction sets before each user (macro) program executed [Burroughs, 1973]. COBOL programs, for example, could be compiled into a different macroinstruction set than FORTRAN programs. If a program that was compiled by the FORTRAN compiler were executed immediately following a program compiled by the COBOL compiler, the FORTRAN macroinstruction set microcode would have to be loaded before the processor would be able to "understand" (i.e., execute) the FORTRAN program. Figure 1.7 illustrates this.

![Figure 1.7. Burroughs Small Reconfigurable Processor](image)

It is possible to build a microprogrammable computer that is microprogrammed to emulate an Intel 486 (i486) processor, and then later change the microprogram so it emulates a Motorola 68030 processor. Such a computer could execute either an IBM or a Macintosh program depending on the microprogram it is executing.
In a microprogrammable computer, the total number of registers, as well as all the circuits for performing all the operations are still "hardwired" and controlled by a fixed format microinstruction. At one time, a particular physical register for example may be used as the EAX register in the i486, whereas another time, that same register may be used as the D0 register in the 68030. Still, however, the microprogrammable processor has a fixed set of registers, multipliers, arithmetic and logic units and data paths which are controlled by a fixed microinstruction format.

1.3.3 Custom Computing Machines

In 1985 the first reprogrammable "field programmable gate array" (FPGA) chip was introduced [Brown, et al., 1992]. At this even lower level, FPGA programs specify not only the gating of data between registers and through functional units, but the connection of a fixed number of gates that actually make up the registers and functional units. A term used for this very low level of program instructions is "nanocode." The FPGA (or connected set of FPGAs) cannot be called a "processor" until it is nanoprogrammed to be one. FPGA processors that are designed to be reconfigured in some fashion based on the application are referred to in the integrated circuit technology and design automation community as custom computing machines (CCMs).

The nanocode of a CCM consists of bytes of ones and zeros, however, these ones and zeros effectively define the interconnection of a fixed set of gates and flip-flops within the FPGA(s). These components can be interconnected to implement any number of registers and different types of functional units within practical limits. It is evident that a computer designed based on FPGAs would implement the different processor instruction sets more efficiently than a microprogrammable based processor. First, in microprogrammable processors, bus architectures are often used between registers and functional units to maintain generality and to allow microoperation encoding. Point to
point data paths between registers and functional units are much easier to implement at the
gate level when the code allows gate interconnect to be defined specifically. Also, there
would be no wasted gates implementing registers or functions that a particular emulated
macroinstruction set might not require. The exact number of registers are connected the
most direct way, to functional units which implement the exact operations necessary for
the particular processor being emulated.

1.4 SURVEY OF CUSTOM COMPUTING MACHINES

Figure 1.8 illustrates the components and tools of a FPGA based processor.
Employing reprogrammable FPGAs for custom computing is an area several other
researchers are pursuing. However, these others do not share our design objective of
permitting programming of the FPGAs in a general-purpose high-level programming
language. Others focus on specialized functions to the point of limiting general
applicability in all areas. All, however, are successful in demonstrating the concepts and
potential for fine grained granularity of functions at the hardware/software machine
boundary.

Figure 1.8. FPGA Based Reconfigurable Processor
1.4.1 DEC’s Perle-0

Digital Equipment, Paris Research Laboratories introduced their concept of a “programmable active memory” (PAM) which in their implementation of Perle-0 connected 25 XILINX 3020s (totaling 50,000 equivalent gates) to a 512K byte RAM on a single VME Sun 3 board [Bertin, et al., 1989]. Since the FPGAs connect directly to the bus, it may be configurable as functional memory. Their focus, however, was oriented towards implementing specialized hardware coprocessors which are loaded with data and execute specialized functions for specialized applications. Early projects using PAM investigated massively parallel processors designed to operate on large operands (e.g., 150 decimal digit modular multiplication, data compression and image processing) with the goal of augmenting the CPU for a particular application. Operands are written into the PAM by the Sun and the results read back out upon completion.

1.4.2 PRISM-II

Another example is the PRISM-II platform, which contains an Am29050 main processor with slots for several triple XILINX 4010 boards (totaling 30,000 equivalent gates each) for custom coprocessing. So far from [Wazlowski, et al., 1993], they have reported quite good results on “single-pass” functions without loops. Single pass functions pose a problem because the main processor must transfer the operands and results back and forth from memory which may increase memory transactions overall. A goal of PRISM-II is to implement all the loop constructs of C which will increase the grain size and surely reduce overall the number of memory transactions.

1.4.3 Reconfigurable Processor Unit

A Reconfigurable Processor Unit (RPU) described in [Guccione and Gonzalez, 1993] is an array of reprogrammable FPGAs attached to a memory. Their overall goal of
compiling a subset of C into FPGA code appears similar to PRISM's but Guccione and Gonzalez instead seem to have focused on implementing specific parallel models with limited and specialized constructs for looping. Unlike PRISM, it appears that a RPU is capable of fetching its own data from memory and writing results back which eliminates any extra load-store transactions by the main processor. The RPU as described appears it too may be configurable as FM.

1.4.4 MoM-4 Xputer

The MoM-4 Xputer architecture is one result of the work by Hartenstein described in [Ast, et al., 1993]. Their data-procedural paradigm emphasizes "data sequencing" as opposed to control flow sequencing as in a von Neumann computer. Implementable as a reprogrammable FPGA attached to the main memory, multi-operand computations are performed in combinational logic connected to an internal cache. Functional memory appears to be equivalent to the f- and h-functors along with the programmable interconnect within the rALU of the MoM-4. When functional memory is combined with a "minimal" (ALU-less) processor and cache in the same FPGA, it is quite similar to an Xputer. It is the goal of our project, however, to implement high-level languages directly, therefore our control flow mechanism, which is based on a decision table, is more general purpose.

1.4.5 Splash 2

Splash 2 contains one or more boards each with an array of 16 well connected XILINX 4010 chips [Gokhale and Minnich, 1993]. The architecture does an excellent job supporting pipelined and SIMD processor configurations. Splash 2, for example, can be programmed in dbC, which is a superset of C used on other SIMD computers. The dbC preprocessor produces C that runs on the Sun and VHDL which define SIMD processors with an instruction set tailored to the application, one or more of which fit into
each XILINX chip. When the actual program executes, looping is still handled in the
Sun, which transmits SIMD instructions to the Splash 2 board(s).

1.4.6 AnyBoard

An Anyboard [Van den Bout, 1993] is a six XILINX chip highly configurable board
which plugs into an IBM PC. Since Anyboard is for prototyping hardware, their
SOLDER language (similar to C) does provide if-then constructs but other program
control constructs of C would have limited value. This is because when programming in
the Anyboard environment, the user thinks in terms of designing hardware whereas in the
other compile-to-FPGA projects (including ours), the goal is to translate high level
language programs (where the user is thinking about writing software). Anyboard’s
design mapping tool for partitioning a design across many chips, however, would be
useful in any compile-to-FPGA project where a compiled function may consume more
than one FPGA.

1.4.7 Chameleon

Chameleon is a workstation [Heeb and Pfister, 1993] based on LSI Logic’s LR33000
32 bit RISC processor that has a Configurable Array Logic (CAL) array of more than
6,000 gates attached to the system bus. The CAL array can be configured as a
coprocessor with its own memory and I/O. The Debora language used to program the
logic array is C like, but intended for describing the state transitions of sequential logic.
All statements execute in parallel except those “guarded” using the IF construct. Except
for the IF statement, there are no other traditional language constructs for defining control
flow.
1.5 THE FUNCTIONAL MEMORY APPROACH

The functional memory approach involves implementing all expression computation in FPGAs connected to the RAM, which together is called functional memory. The processor is left only with the task of copying expression results into destination memory variables.

1.5.1 Functional Memory

Functional memory is a simple extension of the "boolean processor" (BP) concept discussed in Section 1.1. Instead of condition result bits being captured in the FPGA, variables are captured in registers as they are written to main memory. Mapping the registers into main memory relieves the system processor of having to explicitly copy variables into FPGA registers. Instead of only outputting a vector stating which action stubs to execute or which rule to execute, the FPGA can be programmed to compute arithmetic expressions that use the registered variables as operands. A "low-tech" analogy is formula cells in a spreadsheet, as illustrated in Figure 1.9. Like a spreadsheet, locations can be programmed to be the calculated result of an expression containing other memory locations as arguments. Similar to a BP, when functional memory is written, expressions will be recomputed simultaneously using combinational logic. Also similar to a BP, execution time is instantaneous and independent of the number of input locations.

When the system processor executes an assignment statement, it stores a value into a variable location in the memory. If this variable is used in an expression, it will be automatically captured in a functional memory register. Expression results must also be given their own memory locations, just as formulas are given their own cell addresses in a spreadsheet. The analogy extends to the human spreadsheet user being the processor.
When the user changes a spreadsheet cell, all the formula cells that refer to that changed cell themselves change instantaneously.

![Figure 1.9. Functional Memory Concept](image)

Functional memory is used in conjunction with the DT programming model to implement a FM Computer.

### 1.5.2 A Functional Memory Computer (FMC)

A functional memory computer contains functional memory in addition to (or in place of) conventional data RAM. The gate array is programmed using the PALASM logic-gate programming language. Expression computation is performed not in the processor but in the functional memory. In “pure” functional memory computing, no arithmetic or logic operations are performed in the processor. The necessary processor instruction set reduces to move and jump instructions, as illustrated in Figure 1.10.
1.5.3 High Level Language FMC Programming

Our FM approach to custom computers is quite different from those discussed in the last section. Since FPGAs still offer minimal reprogrammable gate counts (=10,000), in many cases, the added parallel processing with the custom coprocessor approach is still necessarily fine-grained. We observe that several of these projects have made great progress towards synthesizing logic from C-like blocks within loops of user programs, however, one problem with these finer-grained systems is that when they don’t loop, the resulting program often requires an increase in the number of transactions across the system bus because operands still must be passed back and forth every iteration. The direct use of functional memory would help here.

![Diagram](image)

Figure 1.10. Functional Memory Computer and Instruction Set
When these coprocessors do implement internal looping, none that we know of are able to implement all varieties of looping mechanisms found in today's high level programming languages. Many are specialized constructs for specific parallel processing applications. Functional memory can also be used to compute jump addresses from the variable operands of the conditional expressions in a program, by intermediately representing the program as a decision table. With our technique, we can implement the looping control for any high level language program in a reprogrammable FPGA.

Figure 1.11 illustrates what tools would be necessary to provide functional memory computing to high level language programmers. The compiler first would have to produce the macroinstruction code consisting of move instructions and jumps, which would be stored in the program memory. Additionally, the compiler would have to produce additional nanocode for implementing each expression computation found in the user's source program. Nanocode formats are specific to the particular FPGA chip manufacturer so the compiler would most likely produce a FPGA specification in a common circuit design specification language, which would further be compiled using the manufacturer's design tools.

The most novel part of the FMC compiler is obviously the part which extracts the specific expression information from the high level language program and produces the circuit design specification which computes the expressions. Expressions on the right sides of assignment statements are allocated memory locations and the logic which computes the expression is generated. Also, logic for evaluating the conditional expressions which lead to jump addresses must also be generated. The main goal of this dissertation is to demonstrate the feasibility of a functional memory computer and its compiler.
1.6 OTHER RELATED LITERATURE

Prior to the FPGA custom computing machine literature of the 1990s, data flow computers in the 1980s showed quite a bit of promise for fast, nondeterministic execution of expressions. Data flow computers [Dennis, 1980] however were usually implemented using some type of processor that executes machine cycles at its most granular level, whereas custom computing machines execute combinational logic at the most granular level.

1.6.1 Hybrid Data Flow Computers

Hybrid data flow architectures can be categorized using dimensions similar to a model developed in [Carlson and Fortes, 1987]. As the graph in Figure 1.12 on the left shows, the vertical dimension reflects the order scheduling scheme of computational steps. Zero (0) represents a control-sequencing scheme, where computational steps are scheduled in a "centralized" fashion, having nothing to do with the availability of operands (only that they are guaranteed), as in von Neumann computers. One (1) on the vertical axis
represents the data-driven scheme, where operations are scheduled in a "decentralized" fashion, based purely on the independent availability of operands.

The horizontal axis represents the granularity of computational steps. Smaller values represent smaller granules of space and time, such as required only for simple binary (two operand) operations and expressions. Further to the right, parallelism becomes increasingly coarser grained with clusters of assignment statements, basic blocks, blocks with loops, and tasks.

![Diagram](image)

**Figure 1.12. Categorizing Hybrid Data Flow Computers**
(Adapted from Carlson and Fortes, 1987)

Most "pure" data flow computers (i.e., those that are not hybrids) implement their most elementary program operations on a set of one or more primitive independent processors. A "processor" would imply sequencing, at least in the "finite state machine" sense. For this reason, we show data flow computers in the graph on the right in Figure 1.12 using a "control-sequencing" scheme at the most finest levels of granularity (e.g., \(a+b\)). Beginning at the simple expression (two-operand) level, operations can become data-driven (e.g., \(a+b\) is computed upon \(a\) and \(b\) both being available). Some hybrid computers may maintain von Neumann style computations for expressions but
assignments within basic blocks are scheduled in data flow fashion. Some hybrids may consist of von Neumann processors executing tasks with task scheduling itself being data-driven [Buehrer and Ekanadham, 1987].

Among the goals of the Piecewise Data Flow computer (PDF) [Requa and McGraw, 1983] is to decrease the latency of execution of a single program and to support most existing applications without reprogramming (only recompiling). These are two of our goals as well. The PDF is a heterogeneous multiprocessor which can (a) perform parallel array instructions, (b) overlap independent scalar operations within a basic block, and (c) overlap basic blocks. The scheduling of these basic operations is data-driven. Loops and tasks, however are control sequenced which allows programs written in conventional languages to be mechanically translated for execution on the PDF.

As the graph shows, a FMC is most similar to the Piecewise architecture, hence is not only a hybrid, but also is able to support easily applications written in conventional languages while potentially reducing single program execution latency. (This chart differs from Carlson and Fortes [1987] in that “Simple Expressions” and below are shown implemented in control sequenced fashion. We have indicated this distinction because at some level, most (if not all) pre-FPGA era data flow computers implement expression computations in control-sequenced fashion, one operation at a time.) What makes a FMC different is the fact that a FMC doesn’t use primitive processors to compute functions, but instead uses functional memory. FMCs exhibit functional parallelism with true expression level granularity in which expressions are computed in data flow fashion in combinational logic. Assignment statements, basic blocks, blocks with loops and coarser granules are sequenced in the von Neumann style on our FMC.
1.6.2 Functional Programming

Functional programming languages offer an alternative to the von Neumann style of one word-at-a-time, one statement-at-a-time computing [Backus, 1978], and inspired the design of data flow languages [Ackerman, 1982]. Since functional memory computers incorporate data flow concepts, they also represent a new class of functional programming systems. Functional memory programming shares with functional/dataflow programming a focus on values (of expressions) instead of memory addresses (of variables).

At a higher level, associated with function procedures rather than expressions, functional memory computers can also be characterized as a class of “Applicative State Transition” systems [Backus, 1978] in which decision tables define ‘large’ transformations between ‘whole’ states. When these transformations can be implemented not by primitive processors but with combinational logic, an enormous potential for performance improvement exists.

The practicality of these ideas is limited by the capacities of FPGAs. As FPGA gate counts increase, we expect functional memory computers will become more and more suited to functional programming. In the meantime, we will direct our attention to what can be done with present-day technology.

1.7 Overview of This Dissertation

This dissertation demonstrates the feasibility of the functional memory approach to custom computing machines by describing the implementation of a system like the one shown in Figure 1.13. The difference between this and the one shown in Figure 1.11 is that the processor program is microcode in a microprogram memory different than the data RAM. This simplifies the design while still fully demonstrating the concept. In fact,
a microinstruction operation code register of just one byte emphasizes the minimal necessary functionality of the processor.

Chapter 2 of this dissertation describes the hardware implementation of our functional memory computer. It begins with how to implement a functional memory and how to program functional memory to compute expressions. Chapter 2 ends with a description of the minimal processor and the system processor interface. Chapter 3 describes the translation of decision table bubble sort and shortest path programs into minimal processor microcode and FPGA assembly language source code. Chapter 4 discusses the implementation of nondeterministic language constructs using a nondeterministic bubble sort example. Chapter 5 analyzes the shortest path and bubble sort examples shown so far by comparing their execution with von Neumann equivalent programs executing on an Intel 486 (i486) processor. Comparisons are made of execution times, cycle counts and load/store counts.

![Diagram](image)

**Figure 1.13. System for Demonstrating the Functional Memory Approach**

Chapter 6 describes the implementation of our FMC compiler, which automatically generates the FPGA source code and minimal processor microcode from decision table source programs. The decision table language syntax is very similar to standard Pascal
with only minor differences. Finally, the applicability of our functional memory approach towards the implementation of a real world application is investigated in Chapter 7. Image processing was chosen as the application because the processing requirements of many image processing functions appear to be well suited to very large scale integrated (VLSI) circuit solutions. FPGA VLSI is especially useful as so many similar but different functions are used. Chapter 7 describes the implementation of a program to perform image magnification and another program used to compute histograms for character recognition. Chapter 8 concludes with a summary and an indication of our further research plans.
CHAPTER 2. THE DESIGN AND IMPLEMENTATION OF A FUNCTIONAL MEMORY COMPUTER

This chapter describes the design of our prototype functional memory computer (FMC). A FMC consists of a von Neumann processor connected to a memory via a standard address-data bus. Some or all of the memory can be functional memory (FM). In a FMC, some or all of an executing program's expression computation is performed in the FM. Section 2.1 describes how to implement FM and how it is programmed. FMs are constructed using field programmable gate array (FPGA) chips. Programmed inside the FPGA are input registers, combinational logic and output multiplexers. The input registers are written when the processor writes to memory. Programmed combinational logic computes expressions from the input registers and feeds the output multiplexers. The output multiplexers are read when the processor reads memory. Section 2.2 describes the input and output interface logic. Section 2.3 shows how several basic infix operators are implemented in combinational logic. Section 2.4 describes how to generate the combinational logic for computing program jump addresses. In some cases, substantial performance gains can be achieved when registers are given an automatic increment or shift capability. Section 2.5 describes how this can be provided. When all the computations are performed in the functional memory, the necessary processor instruction set minimizes to a set of moves and an indirect jump. Section 2.6 concludes this chapter with a description of our minimal processor, and its interface to the system processor.

2.1 CONSTRUCTING A FUNCTIONAL MEMORY

A functional memory (FM) is a random access memory (RAM) connected in parallel with one or more reprogrammable field programmable gate arrays (FPGAs). FM allows
program expressions to be computed in parallel in combinational logic as opposed to sequentially by the main processor. Reprogrammable FPGAs allow changes or revisions of hardware at the gate level, in circuit, in a matter of milliseconds, hence, FPGA logic can be different for different user programs. With FM, the FPGA is programmed to contain registers which can be written when RAM locations are written. The FPGA is programmed to output when certain addresses are read. The design of a FM is straightforward. As Figure 2.1 illustrates, one can be constructed by simply connecting a FPGA in parallel with a RAM and multiplexing the outputs under control of the FPGA. When data are stored into the RAM, they may be also be captured simultaneously in FPGA registers if they are operands used in an expression computation.

![Figure 2.1. Implementing Functional Memory – Multiplexer Method](image)

As Figure 2.2 illustrates, functional memory can also be implemented by connecting a field programmable gate array in parallel with a conventional RAM and logically ORing the outputs. When data are written into the RAM, they may be captured in registers in the FPGA if they are to be used in an expression computation. Expression results (e.g., the right sides of assignment statements) are assigned exclusive addresses with zero values stored in the respective RAM locations. Since the RAM and FPGA outputs are ORed, when reading data from the RAM only, the FPGA outputs zero hence only the RAM is read. When reading expression results, the RAM outputs zero so only the FPGA outputs
are read. When reading "RAM-only" data, the FPGA outputs zero. Multiple FPGAs can
attach to a single RAM as long as only one FPGA drives its outputs on any given read
(just as with multiple RAMs). This way the OR function can be nearer (or within) the
processor.

![Diagram of FPGA and RAM connections](image)

**Figure 2.2. Implementing Functional Memory – Logical OR Method**

The programs which configure the FPGAs are produced by compiler from a high-
level programming language, therefore it is easiest to use a text based boolean equation
method for defining the FPGA logic. Several hardware description languages exist for
VLSI designs. We chose PALASM because it is a fully functional and flexible yet simple
language, which was developed nearly 20 years ago for designing field programmable
medium scale integrated circuit chips. In order to provide a standard PALASM program
format for different chip sizes and pin configurations, a FPGA interface circuit "shell" is
used as illustrated in Figure 2.3. This interface circuit contains all the specific pin
assignment details for a particular FPGA part number (and circuit). Within it is defined a
PALASM module which contains the program for computing the expressions.

### 2.1.1 Programming the FPGA

As Figure 2.3 suggests, each supported FPGA chip type (and pin assignment) has its
own Viewlogic [XILINX, 1993] logic schematic which specifies the in-circuit pin
configuration (i.e., the interface circuit shell). All incoming and outgoing signals are buffered with assigned specific pin numbers. When the processor reads a memory location in which no FPGA output becomes active, then the FPGA outputs must read zero (for the logical-OR with RAM function to work properly). This is accomplished by using active low data outputs and using a resistor network to pull the outputs high (to zero) when no FPGA output is addressed. Since the resistor pull-ups are necessary, the FPGA data output buffers can be wired to switch to a high impedance state when the output is driven high. As Figure 2.3 shows, this is implemented by connecting the data output driver inputs to its high impedance control. This allows the option of more than one FPGA to drive different bits of the same address (e.g., one FPGA can drive the lower byte of a 16 bit word while another drives the upper byte).

Within the Viewlogic interface circuit shell is defined a standardized PALASM macro with pins defined as shown using the PALASM "PDS" file format on the top. Address, control, data in and data out pins are defined, in order, using the same names. Following the EQUATIONS statement appear the compiler generated boolean equations which define the address select logic, input registers, rule address generation logic, expression logic, and output multiplexers for the particular user program. As we shall see later in this chapter, the PALASM language syntax is similar to conventional high level languages. Combinational gates are defined using the "=" assignment symbol whereas flip-flop contents use the "::=" assignment. Bit expression operators include "*" for logical AND, "+" for logical OR and ":+:" for exclusive-OR. Unary inversion is indicated by preceding the signal name or expression with a "!" character.

The optional daisy-chain signal shown in Figure 2.3 can be used to implement large expressions which span several chips. In Chapter 4, the daisy-chain is used in the nondeterministic sorting program to implement a function to provide the indexes of out-
of-order array element pairs. In the following sections we will describe what the combinational logic consists of for implementing expressions in FPGAs of the functional memory.

Figure 2.3. Functional Memory FPGA Interface Circuit Shell
2.2 Interface

The interface logic implemented inside the FPGA consists of registers for capturing input and output multiplexers for gating out the results. There is a register for each operand of an expression. Also, for each expression implemented, there is a set of inputs to the output multiplexer. Each operand corresponds to a program variable which has a designated memory address. Each different expression is also allocated its own memory address. Therefore, address selection logic must be generated for each input register and expression output address.

Our FMC was implemented using XILINX 3090 [XILINX, 1993] parts. Each configurable logic block (CLB) is limited to, at most, five inputs in its combinational expression. Our design assumes a 64K byte memory space. The data bus is 16 bits and address lines A15 to A1 are used to decode byte address pairs for reading or writing. Note that address line A0 is not needed because odd bytes are not addressed separately, but are read and written simultaneously with the even address just below as 16-bit words.

The following three sections describe how the address select logic, input register logic, and output multiplexer logic are generated, making up the processor interface. Expression logic and rule address generation will be discussed in Sections 2.3 and 2.4.

2.2.1 Address Select

The address select logic must produce a select line for each input register and each output expression. Each will correspond to a unique (compiler allocated) address. Because of the nature of the CLBs, it is convenient to group the address lines by hexadecimal digit and decode only those digits that are used. The address selects can then be constructed by logically ANDing the decoded digits.
For example, suppose our program contained input registers at locations 0000 and 0028, and expressions to be output when locations 0002 and 006A are read. As depicted in Figure 2.4, we use an intermediate signal naming convention for each hex digit value where the address selects begin with ‘AS’, followed by an X, H, M, or L corresponding to address lines A15-A12, A11-A8, A7-A4, and A3-A0 respectively (although recall that A0 is not decoded). The remaining hex digit identifies what hex value the signal can be used to select. Therefore, for each group of four address lines, we can first generate the actual input register and output multiplexer select lines, keeping track of which hex digits we need decoded. We see that the select line for address 0028, for example, requires A15-A12 to equal 0 (ASX0), A11-A8 to equal 0 (ASH0), A7-A4 to equal 2 (ASM2) and A3-A0 to equal 8 (ASL8). Once all the select equations have been generated, we know which specific hex digits also must decoded. In this example, we see only hex digit 0 must be decoded for lines A15-A12 (ASX0), as well as for lines A11-A8 (ASH0). For lines A7-A4, digits 0, 2 and 6 need to be decoded (ASM0, ASM2, ASM6). Finally, for lines A3-A0, hex digits 0, 2, 8 and A are used (ASL0, ASL2, ASL8, ASLA).

2.2.2 Input Registers

Each memory location that is involved in an expression computation must be captured in an input register whenever the processor initializes or updates the value. Figure 2.5 lists the PALASM equations for implementing a 16-bit register which captures anything written to location 0028. "reg_028_0" is the name of the input flip-flop, into which data is clocked directly from the data input bus bit 0. The lower byte flip-flops are clocked with the “wrlc” (WRite Lower Clock) line while the high order byte is clocked using the “wrhc” signal. Clocks are enabled by the “sel_028” select line shown generated in Figure 2.4.
When the total number of different input signals into a pair of equations or flip-flops is four or less, they can be combined into one CLB. Therefore, the 16-bit register capturing writes to address 0028 (and 0029) require only eight CLBs.

\[
\begin{align*}
\text{sel}_0 &= \text{ASX0}*\text{ASH0}^*\text{ASM0}^*\text{ASL0} & \text{Select for lambda input} \\
\text{sel}_2 &= \text{ASX0}*\text{ASH0}^*\text{ASM0}^*\text{ASL2} & \text{Select for @Rule output} \\
\text{sel}_{028} &= \text{ASX0}*\text{ASH0}^*\text{ASM2}^*\text{ASL8} & \text{Select for j input} \\
\text{sel}_{06A} &= \text{ASX0}*\text{ASH0}^*\text{ASM6}^*\text{ASLA} & \text{Select for @char[j] output} \\
\text{ASX0} &= /A12^*/A13^*/A14^*/A15 & \text{A15-A12 hex digit 0} \\
\text{ASH0} &= /A8^*/A9^*/A10^*/A11 & \text{A11-A8 hex digit 0} \\
\text{ASM0} &= /A4^*/A5^*/A6^*/A7 & \text{A7-A4 hex digit 0} \\
\text{ASL0} &= /A1^*/A2^*/A3 & \text{A3-A1 hex digit 0} \\
\text{ASL2} &= A1^*/A2^*/A3 & \text{A3-A1 hex digit 2} \\
\text{ASM2} &= /A4^*/A5^*/A6^*/A7 & \text{A7-A4 hex digit 2} \\
\text{ASL8} &= /A1^*/A2^*/A3 & \text{A3-A1 hex digit 8} \\
\text{ASM6} &= /A4^*/A5^*/A6^*/A7 & \text{A7-A4 hex digit 6} \\
\text{ASLA} &= A1^*/A2^*/A3 & \text{A3-A1 hex digit A} 
\end{align*}
\]

Figure 2.4. Address Select PALASM

; j input register at address 028
reg_028_0 := di0 ; j input bit 0
reg_028_0.clk = wrlc ; Write Lower Byte Clock
reg_028_0.ce = sel_028 ; Clock Enable

... reg_028_7 := di7 ; j input bit 7
reg_028_7.clk = wrlc ; Write Lower Byte Clock
reg_028_7.ce = sel_028 ; Clock Enable
reg_028_8 := di8 ; j input bit 8
reg_028_8.clk = wrhc ; Write Upper Byte Clock
reg_028_8.ce = sel_028 ; Clock Enable

... reg_028_15 := di15 ; j input bit 15
reg_028_15.clk = wrhc ; Write Upper Byte Clock
reg_028_15.ce = sel_028 ; Clock Enable

Figure 2.5. Input Register PALASM
2.2.3 Output Multiplexers

Every output from the FPGA is the result of some expression computation. We have chosen a naming convention which indicates the type of expression for each set of signals which will be multiplexed out of the chip when a particular location is read by the processor. Figure 2.6 shows a multiplexer example for three expressions. The microprogram address for a rule to be executed (@rule), an array element address (char[j]) and the result of an expression which appeared on the right side of an assignment statement (j+1). For data output bit 0 (do0), either rule_0 is selected when address 0002 is read, adr_06A_0 is selected when address 006A is read, or exp_06E_0 is read when address 006E is read. By using three intermediate gate levels by convention, we can accommodate up to 2×5×4 = 40 expressions in one chip. Notice that the read signal (/RDC) should be ANDed last to minimize chip access time.

\[
\begin{align*}
do0t1 &= \text{rule}_0 \times \text{sel}_02 + \text{adr}_06A_0 \times \text{sel}_06A \quad ; \quad @\text{rule}, \ char[j] \\
do0t3 &= \text{exp}_06E_0 \times \text{sel}_06E \quad ; \quad j+1 \\
do0a1 &= do0t1 + do0t3 \\
do0 &= /(/RDC \times (do0a1)) \quad ; \quad \text{Output bit 0} \\
\ldots \\
dol5t1 &= \text{rule}_15 \times \text{sel}_02 + \text{adr}_06A_15 \times \text{sel}_06A \quad ; \quad @\text{rule}, \ char[j] \\
dol5t3 &= \text{exp}_06E_15 \times \text{sel}_06E \quad ; \quad j+1 \\
dol5a1 &= dol5t1 + dol5t3 \\
dol5 &= /(/RDC \times (dol5a1)) \quad ; \quad \text{Output bit 15}
\end{align*}
\]

Figure 2.6. Output Multiplexer PALASM

In many cases, not all the bits of an input register or output to the multiplexer need be implemented. Variables used for index variables only need to be the size of the largest index value. With array address computations, for example, the lowest order bit (A0) is always zero hence need not exist. (As indicated earlier, this is because the data path to memory for our FMC is 16 bits but we use the byte addressing convention.) It is
worthwhile implementing only those bits that are necessary because it conserves configurable logic blocks which are the scarcest resource in a FMC.

2.3 IMPLEMENTATION OF EXPRESSION OPERATORS

Combinational logic to compute expressions from input registers is built up from PALASM modules that implement each operator. Conditional operators we wish to implement are $A \neq B$ and $A < B$, where $A$ and $B$ are numbers ranging from one to 16 bits wide. By simple input operand reversal and/or output inversion, we easily derive $A = B$ by inverting the output of $A \neq B$. $A \geq B$ is obtained by inverting $A < B$, and $A > B$ is derived by reversing the $A$ and $B$ operands. Finally, $A \leq B$ is obtained by reversing $A$ and $B$ and inverting the output.

With an Add (with carry in) function, subtraction can be implemented by inverting the operand to be subtracted and setting the carry (assuming a 2’s complement notation).

When comparing or adding a constant, it is best to build the constant into the expression so expensive constant registers need not be used. Zero bit values can be implemented using a $(B \text{ AND NOT } B) = 0$ identity and one values can use $(B \text{ OR NOT } B) = 1$.

2.3.1 $A \neq B$ Test

Figure 2.7 shows how the $A \neq B$ test can be implemented. Intermediate signals exclusive-ORing bits from $A$ and $B$ in each position are ORed all together at a second level. If any pair of bits were different in any bit position, a one value will prevail. The $A \neq B$ test consumes 11 CLBs. Two 16-bit integers are tested in three gate delays.
;compute bit C where if A <> B then C=1 else C=0

;first level of not equal to comparator

C_eq0_1 = A0:::B0 + A1:::B1
C_eq2_3 = A2:::B2 + A3:::B3
C_eq4_5 = A4:::B4 + A5:::B5
C_eq6_7 = A6:::B6 + A7:::B7
C_eq8_9 = A8:::B8 + A9:::B9
C_eq10_11 = A10:::B10 + A11:::B11
C_eq12_13 = A12:::B12 + A13:::B13
C_eq14_15 = A14:::B14 + A15:::B15

;second level of comparator

C_eq0_7 = C_eq0_1 + C_eq2_3 + C_eq4_5 + C_eq6_7
C_eq8_15 = C_eq8_9 + C_eq10_11 + C_eq12_13 + C_eq14_15

;third level of comparator

C = C_eq0_7 + C_eq8_15

--- Figure 2.7. A ≠ B Test PALASM ---

Figure 2.8 shows how basically the same circuit can be used to test, in this example, if A is equal to five. Five is equal to 0000 0000 0000 0101, so for each bit i, the AND of Ai and not Ai will provide a zero value, and the OR of Ai with not Ai will give a one. Each Bi term in Figure 2.7 is replaced with either (Ai + /Ai) for a one and (Ai*/Ai) for a zero. The final "C =" equation is the OR of all the intermediate exclusive-OR sums, and is inverted because here we are implementing the '=' function instead of '≠'.

Minimization is possible for the equations in Figure 2.8 in terms of CLB count without algebraic manipulation. Five Ai terms will fit in one equation, which would reduce the number of logic levels by one and cut the number of CLBs by more than one half to five.
; compute bit C where if A = 5 then C=1 else C=0

;first level of comparator

C_eq0_1 = A0++:(A0 + /A0) + A1++:(A1*/A1)
C_eq2_3 = A2++:(A2 + /A2) + A3++:(A3*/A3)
C_eq4_5 = A4++:(A4*/A4) + A5++:(A5*/A5)
C_eq6_7 = A6++:(A6*/A6) + A7++:(A7*/A7)
C_eq8_9 = A8++:(A8*/A8) + A9++:(A9*/A9)
C_eq10_11 = A10++:(A10*/A10) + A11++:(A11*/A11)
C_eq12_13 = A12++:(A12*/A12) + A13++:(A13*/A13)
C_eq14_15 = A14++:(A14*/A14) + A15++:(A15*/A15)

;second level of comparator

C_eq0_7 = C_eq0_1 + C_eq2_3 + C_eq4_5 + C_eq6_7
C_eq8_15 = C_eq8_9 + C_eq10_11 + C_eq12_13 + C_eq14_15

;third level of comparator

C = / (C_eq0_7 + C_eq8_15)

Figure 2.8. A = 5 Test FPGA Equations

2.3.2 Magnitude Comparison Tests

In order to be able to compare two values, a magnitude comparison test function must be provided. Ours is implemented using a basic A–B subtract circuit, where the borrow out of the highest order bit indicates if A is less than B. This comparison-only operation is somewhat simpler to implement than the full subtract function because the difference result bits themselves need not be generated.

Figure 2.9 shows a basic A<B combinational logic function for comparing two 16-bit operands. As with the not-equal operation, only three gate delays are needed to complete the comparison. By reversing A and B, the A>B can be compared. By inverting the C output, whether A≥B can be determined. By both reversing the inputs and inverting the output, A≤B can be computed. The intermediate carry notation is that C_It/j/k = carry
into bit $i$ if the carry into bit $j$ ($j < i$) is $k$. ($k = 0$ or 1). For example, \( C_{lt10\_6\_0} = \text{carry into bit 10 if carry into bit 6 is 0} \). The A<B PALASM equations consume 13 CLBs.

;compute bit $C$ where if $A < B$ then $C = 1$ else $C = 0$

;1st level of 16 bit comparator:
\[
C_{lt2} = A1\_B1 + A1 + B1 * A0 + B0
\]
\[
C_{lt4\_2\_0} = A3\_B3 + (A3 + B3) * A2 + B2
\]
\[
C_{lt4\_2\_1} = A3\_B3 + (A3 + B3) * A2 + B2
\]
\[
C_{lt6\_4\_0} = A5\_B5 + (A5 + B5) * A4 + B4
\]
\[
C_{lt6\_4\_1} = A5\_B5 + (A5 + B5) * A4 + B4
\]
\[
C_{lt8\_6\_0} = A7\_B7 + (A7 + B7) * A6 + B6
\]
\[
C_{lt8\_6\_1} = A7\_B7 + (A7 + B7) * A6 + B6
\]
\[
C_{lt10\_8\_0} = A9\_B9 + (A9 + B9) * A8 + B8
\]
\[
C_{lt10\_8\_1} = A9\_B9 + (A9 + B9) * A8 + B8
\]
\[
C_{lt12\_10\_0} = A11\_B11 + (A11 + B11) * A10 + B10
\]
\[
C_{lt12\_10\_1} = A11\_B11 + (A11 + B11) * A10 + B10
\]
\[
C_{lt14\_12\_0} = A13\_B13 + (A13 + B13) * A12 + B12
\]
\[
C_{lt14\_12\_1} = A13\_B13 + (A13 + B13) * A12 + B12
\]
\[
C_{lt16\_14\_0} = A15\_B15 + (A15 + B15) * A14 + B14
\]
\[
C_{lt16\_14\_1} = A15\_B15 + (A15 + B15) * A14 + B14
\]

;2nd level of 16 bit comparator:
\[
C_{lt6} = C_{lt6\_4\_0} + C_{lt6\_4\_1} * (C_{lt4\_2\_0} + C_{lt4\_2\_1} * C_{lt2})
\]
\[
C_{lt12\_6\_0} = C_{lt12\_10\_0} * C_{lt12\_10\_1} + C_{lt10\_8\_0} * C_{lt12\_10\_1} + C_{lt12\_6\_0} * C_{lt12\_10\_1} + C_{lt12\_6\_1} * C_{lt12\_10\_1}
\]
\[
C_{lt12\_6\_1} = C_{lt12\_10\_0} * C_{lt12\_10\_1} + C_{lt12\_6\_1} * C_{lt12\_10\_1}
\]
\[
C_{lt16\_12\_0} = C_{lt16\_14\_0} + C_{lt16\_14\_1} * C_{lt14\_12\_0}
\]
\[
C_{lt16\_12\_1} = C_{lt16\_14\_0} + C_{lt16\_14\_1} * C_{lt14\_12\_1}
\]

;3rd level of 16 bit comparator:
\[
C = C_{lt16\_12\_0} + C_{lt16\_12\_1} * C_{lt12\_6\_0} + C_{lt16\_12\_1} * C_{lt12\_6\_1} * C_{lt6}
\]

Figure 2.9. A < B PALASM
2.3.3 Adding and Subtracting

Figure 2.10 shows the basic A+B combinational logic function. By inverting each B bit and setting the carry in (C0) equal to 1, two’s complement subtraction can easily be implemented. The intermediate carry notation is the same as in the A<B function,
\[ Dc_{i,j,k} = \text{carry into bit } i \text{ if the carry into bit } j (j < i) \text{ is } k. (k = 0 \text{ or } 1) \]
For the intermediate sum bits, \( Ds_{i,j,k} = \text{sum bit } i \text{ if carry into bit } j (j < i) \text{ is } k. (k = 0 \text{ or } 1) \)
For example, \( Ds_{9,8,1} = \text{sum bit 9 if carry into bit 8 is 1} \).

---

\[
D = A + B + C0, \text{ } D, \text{ } A, \text{ } B \text{ are 16 bits, } C0 \text{ is carry bit in}
\]
\[
D0 = A0 + B0 + C0
\]
\[
D1 = A1 + B1 + (A0 \times B0 + (A0 + B0) \times C0)
\]
\[
Dc2 = A1 \times B1 + (A1 + B1) \times (A0 \times B0 + (A0 + B0) \times C0)
D2 = A2 + B2 + Dc2
\]
\[
D3 = A3 + B3 + (A2 \times B2 + (A2 + B2) \times Dc2)
\]
\[
Dc4_2_0 = A3 \times B3 + (A3 + B3) \times (A2 \times B2)
Dc4_2_1 = A3 \times B3 + (A3 + B3) \times (A2 + B2)
D4 = A4 + B4 + (Dc4_2_0 + Dc4_2_1 \times Dc2)
\]
\[
Ds5_4_0 = A5 + B5 + (A4 \times B4)
Ds5_4_1 = A5 + B5 + (A4 + B4)
D5 = Ds5_4_0 \times (Dc4_2_0 + Dc4_2_1 + Dc2) + Dc5_4_1 \times (Dc6_4_0 + Dc6_4_1 + Dc2)
\]
\[
Dc6_4_0 = A5 \times B5 + (A5 + B5) \times (A4 \times B4)
Dc6_4_1 = A5 \times B5 + (A5 + B5) \times (A4 + B4)
Dc6 = Dc6_4_0 + Dc6_4_1 \times (Dc4_2_0 + Dc4_2_1 + Dc2)
D6 = A6 + B6 + + Dc6
\]
\[
D7 = A7 + B7 + (A6 \times B6 + (A6 + B6) \times Dc6)
\]
\[
Dc8_6_0 = A7 \times B7 + (A7 + B7) \times (A6 \times B6)
Dc8_6_1 = A7 \times B7 + (A7 + B7) \times (A6 + B6)
D8 = A8 + B8 + + (Dc8_6_0 + Dc8_6_1 + Dc6)
\]
\[
Ds9_8_0 = A9 + B9 + (A8 \times B8)
Ds9_8_1 = A9 + B9 + (A8 + B8)
D9 = Ds9_8_0 \times (Dc8_6_0 + Dc8_6_1 + Dc6) + Ds9_8_1 \times (Dc8_6_0 + Dc8_6_1 + Dc6)
\]

---

Figure 2.10. \( D = A + B \) Equations

39
\[ \begin{align*}
D_{10,8_0} &= A_9B_9 + (A_9B_9)(A_8B_8) \\
D_{10,8_1} &= A_9B_9 + (A_9B_9)(A_8B_8) \\
D_{10,6_0} &= D_{10,8_0} + D_{10,8_1} + D_{8_6} \\
D_{10,6_1} &= D_{10,8_0} + D_{10,8_1} + D_{8_6} \\
D_{10} &= A_{10}B_{10} + (D_{10,6_0} + D_{10,6_1} + D_{6}) \\
\end{align*} \]

\[ \begin{align*}
D_{11,10_0} &= A_{11}B_{11} + (A_{10}B_{10}) \\
D_{11,10_1} &= A_{11}B_{11} + (A_{10}B_{10}) \\
D_{11} &= D_{11,10_0} + (D_{10,6_0} + D_{10,6_1} + D_{6}) \\
\end{align*} \]

\[ \begin{align*}
D_{12,10_0} &= A_{12}B_{12} + (A_{11}B_{11}) \\
D_{12,10_1} &= A_{12}B_{12} + (A_{11}B_{11}) \\
D_{12} &= D_{12,10_0} + (D_{10,6_0} + D_{10,6_1} + D_{6}) \\
\end{align*} \]

\[ \begin{align*}
D_{13,12_0} &= A_{13}B_{13} + (A_{12}B_{12}) \\
D_{13,12_1} &= A_{13}B_{13} + (A_{12}B_{12}) \\
D_{13} &= D_{13,12_0} + (D_{12,6_0} + D_{12,6_1} + D_{6}) \\
\end{align*} \]

\[ \begin{align*}
D_{14,12_0} &= A_{14}B_{14} + (A_{13}B_{13}) \\
D_{14,12_1} &= A_{14}B_{14} + (A_{13}B_{13}) \\
D_{14} &= D_{14,12_0} + (D_{12,6_0} + D_{12,6_1} + D_{6}) \\
\end{align*} \]

\[ \begin{align*}
D_{15,14_0} &= A_{15}B_{15} + (A_{14}B_{14}) \\
D_{15,14_1} &= A_{15}B_{15} + (A_{14}B_{14}) \\
D_{15} &= D_{15,14_0} + (D_{14,6_0} + D_{14,6_1} + D_{6}) \\
\end{align*} \]

Figure 2.10 (continued). \( D = A + B \) Equations

Notice that the equations in Figure 2.10 are grouped by bit. If the actual bit widths of the operands are known, only the PALASM up to the widest operand needs to be generated, instead of defaulting to some arbitrary large number. Two 16-bit integers can be added in three levels of delay, consuming 34 CLBs.

2.4 Computing the Rule Address

The most general form of conditional jumping is a multi-way computed branch where the addresses can be derived from any conditional expression which depends on any number of variables and their states. This is the general case used in a decision table, which we have chosen as our high level language to implement.
The most complex expression which must be implemented in the FPGAs is often the expression for computing the starting jump address for each rule. Figure 2.11 shows the condition entries portion of an example decision table with five rule columns and four condition stubs rows. The entry at the intersection of a row and column shows the state that the condition stub must be in for that rule to fire (that is, for the actions associated with that rule to be selected for execution). When all four condition stub states match in a particular rule column, then that rule fires. On the FMC, the states of the condition stubs are first used to generate a signal corresponding to each rule indicating when it fires. These intermediate signals are then used to generate each address bit of the starting location of the action code which executes that particular rule.

<table>
<thead>
<tr>
<th>Condition Stubs</th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
<th>Rule 4</th>
<th>Rule 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda = (L1, L0)</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>Cond. Expression (C2)</td>
<td>-</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Cond. Expression (C3)</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Cond. Expression (C4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>0004</td>
</tr>
<tr>
<td>Rule 2</td>
<td>003C</td>
</tr>
<tr>
<td>Rule 3</td>
<td>0044</td>
</tr>
<tr>
<td>Rule 4</td>
<td>0074</td>
</tr>
<tr>
<td>Rule 5</td>
<td>00B4</td>
</tr>
</tbody>
</table>

Figure 2.11. Condition Entries Selecting Rule and Rule Starting Addresses

Figure 2.11 on the right provides hypothetical rule starting addresses for this example. Rule 1 starts at location 0004. Rule 2 starts at 003C, Rule 3 at 0044, Rule 4 at 0074 and Rule 5 starts at location 00B4. From these addresses, we can see that, for example, address bits 0 and 1 are always zero, hence they need not be implemented. Address bit 2 is a one when any rule is selected. Address bit 3 is a one only when Rule 2
(003C) is selected. Address bits 4 and 5 are ones when Rule 2 (003C), Rule 4 (0074) and Rule 5 (00B4) are selected. The remaining high-order address bits are determined in like fashion.

Figure 2.12 shows the PALASM code for implementing this example. First are listed those signals which are true for each rule when selected (Rule 1, ..., Rule 5). Note that they depend on the value of the lambda variable (L1, L0), as well as the bit states of the other three condition stubs.

```plaintext
; Rule bits from L0, L1, C2, C3, C4
Rule1 = /L1*/L0 ; Starting at Address 004
Rule2 = /L1*L0*C2 ; Starting at Address 03C
Rule3 = /L1*L0*/C2*C3 ; Starting at Address 044
Rule4 = /L1*L0*/C2*/C3*C4 ; Starting at Address 074
Rule5 = /L1*L0*/C2*/C3*/C4 ; Starting at Address 084

rule_2 = Rule1 + Rule2 + Rule3 + Rule4 + Rule5
rule_3 = Rule2
rule_4 = Rule2 + Rule4 + Rule5
rule_5 = Rule2 + Rule4 + Rule5
rule_6 = Rule3 + Rule4
rule_7 = Rule5
```

Figure 2.12. Rule Jump Address Calculation Example

The second set of signals in Figure 2.12 are the bit output values of the starting address for the rule to fire. For each output bit rule_i (i = 2, ..., 7), then for each rule Rule_j (j = 1, ..., 5), if the i-th bit of the starting address for Rule_j is a one, then it is included in the rule_i equation.

In this example, only those non-zero bits of the address output were implemented, which is most efficient. We see that the rule address can be computed from the condition stubs results in less than ten CLBs.
2.5 AUTO-CLOCKING REGISTERS

There are many instances where adding an internal accumulator or shifting capability within an expression can help to eliminate processor clock cycles. In this section we give the PALASM implementations of an automatic bit-summing function and an automatic 8-bit shift register function.

2.5.1 Automatic Bit Summing

In this example, two bits (B1 and B0) are added to A when A is written. A is written with all zeros if C is written. In other words, if both B1 and B0 contain zeros, nothing is added to A when written. If either B1 or B0, but not both is equal to one, then A is incremented by one. If both B1 and B0 are 1, A is incremented by two. This function is useful for computing diagonal histograms for character recognition. It can be reduced somewhat if automatic single incrementing is all that is desired.

This example, shown in Figure 2.13, only implements eight bits. The outputs of the register feed back to the input. The register is clocked when either sel_A or sel_C is written. When sel_C is selected, zeros are forced into the register inputs clearing them. When sel_A is selected, A plus B1 plus B0 is fed to the register inputs. The trailing write clock edge (wrlc) clocks the new data into the A register.

2.5.2 Auto-Shifting

An internal shift register is useful in applications where operands are less than 16 bits and more than one may be packed into a 16-bit word. In these cases, it is useful if an input register can either be written directly, or shifted some number of bits if a different variable is written.

Figure 2.14 shows the PALASM implementation of a 16-bit register (A) which can be written directly from D when A is written, or is written by a different value if C is
written. As shown, the alternative C value inputs are the lower eight bits of register A which are stored into the upper eight bits. The upper eight bits of a different register (B) are stored into the lower eight bits of register A. This function is useful in image processing applications with byte-wide data stored two pixels per 16-bit word.

```
;A := A + B1 + B0, A is 8 bits, B1 and B0 are each 1 bit
; write to A adds B1 and B0 to A
; write to C clears A
A0 := /sel_C*(A0 :+: (B1:+:B0))
A0.clkf = wr1c
A0.ce = sel_A + sel_C
A1.clkf = wr1c
A1.ce = sel_A + sel_C
Ac2 = A1*(B1*B0) + (A1 + (B1*B0))*A0*(B1:+:B0)
A2.clkf = wr1c
A2.ce = sel_A + sel_C
As3 = A3 :+: (A2*Ac2)
A3 := /sel_C*As3
A3.clkf = wr1c
A3.ce = sel_A + sel_C
As4 = A4 :+: (A3 + A3*A2*Ac2)
A4 := /sel_C*As4
A4.clkf = wr1c
A4.ce = sel_A + sel_C
As5 = A5*/(A3 + A3*A2*Ac2) + (A5:+:A4)*(A3 + A3*A2*Ac2)
A5 := /sel_C*As5
A5.clkf = wr1c
A5.ce = sel_A + sel_C
Ac6 = A5 + A5*A4*(A3 + A3*A2*Ac2)
A6.clkf = wr1c
A6.ce = sel_A + sel_C
As7 = A7 :+: (A6*Ac6)
A7 := /sel_C*As7
A7.clkf = wr1c
A7.ce = sel_A + sel_C
```

---

Figure 2.13. Eight Bit A = A + B0 + B1 Accumulator PALASM

44
;when sel_A then A[15:0] := D[15:0]

A0 := sel_A*D0 + sel_C*B8 ;sel_A gates D0, sel_C gates B8
A0.clkf = wrlc
A0.ce = sel_A + sel_C

A1 := sel_A*D1 + sel_C*B9 ;sel_A gates D1, sel_C gates B9
A1.clkf = wrlc
A1.ce = sel_A + sel_C

... 

A7 := sel_A*D7 + sel_C*B15 ;sel_A gates D7, sel_C gates B15
A7.clkf = wrlc
A7.ce = sel_A + sel_C

A8 := sel_A*D8 + sel_C*A0 ;sel_A gates D8, sel_C gates A0
A8.clkf = wrhc
A8.ce = sel_A + sel_C

A9 := sel_A*D9 + sel_C*A1 ;sel_A gates D9, sel_C gates A1
A9.clkf = wrhc
A9.ce = sel_A + sel_C

... 

A15 := sel_A*D15 + sel_C*A7 ;sel_A gates D15, sel_C gates A7
A15.clkf = wrhc
A15.ce = sel_A + sel_C

---

Figure 2.14. 16 Bit Auto Shift Register - Shift Left Eight Bits

These two small examples demonstrate how registers, multiplexers and operators can be combined to provide other interesting parallel functions and save processor instruction cycles at the same time. We will see these two PALASM examples (and others) utilized in Chapter 7.

2.6 MINIMAL PROCESSOR

A block diagram of the 4.5" by 6.5" FMC board that we have implemented is illustrated in Figure 2.15. The board plugs into the system bus of a parallel computer
platform we have built which emulates a cache-only memory architecture (see Figure 1.4). Current hardware allows a mixture of Intel MCS-51 8031 (i8031) microcomputer boards and FMC boards to execute in parallel. Executing single programs on a hybrid configuration is discussed in [Halverson and Lew, 1994-1]. Here we are concerned only with programs executing completely on a single FMC.

The FMC board contains a microprogrammable minimal processor (mP) implemented using a FPGA with three 8,192 byte microprogram RAMs (for a 24-bit microinstruction format). The mP is connected to 16,384 bytes of data RAM (configured as 8,192 16-bit words) and four FPGAs which make up the functional memory. Notice that the RAM–FPGA OR function is performed inside the mP.

The system processor in our implementation is an 11.059 MHz i8031, which connects to an IBM PC serially at 19,200 bits per second. The FMC is initialized and
controlled by the system processor. The FPGA code is loaded first, which includes the mP specification and the four FM programs. This process is discussed in slightly more detail below in Section 2.6.5.

After the FPGAs are configured, the system processor loads the program microcode. The system processor can then place the mP in a "transparent" mode with the μPAccess signal and access the FM directly for initialization or test if necessary. The —START/DONE handshake signals begin program execution and notify the system processor when execution terminates. The use of these signals is also explained further in Section 2.6.5.

2.6.1 The Architecture of the Minimal Processor

The minimal processor (mP) is designed only to move memory words from location to location. It is also able to load its program counter with a memory location. As Figure 2.16 illustrates, all that is required is a memory address register (MAR) and a data output register (DOR) which can be loaded immediately (from the program) or from a memory location. This allows for indirect addressing necessary for accessing array elements. A program counter can be incremented by one, or loaded from the program or memory. This minimal processor contains fewer components than conventional processors as we see; absent are temporary operand registers and an arithmetic logic unit.

We chose a microprogrammed design in order to be able to parallelize the program and data fetches and to save having to design in an instruction fetch unit. The microprogram address on top selects the three byte microinstruction in the microprogram RAMs, which is clocked into the microinstruction register as shown. The first byte is the op (operation) code field which initiates FM reads and writes, controls the bus multiplexers and selects which internal data bus (IBUS) register is clocked. It also contains the DONE flag used for indicating to the system processor that the microprogram
has terminated. The microinstruction register also contains a two byte constant field which can supply an FM address, FM data or a microprogram jump address.

Our mP contains four other registers: (1) the data output register (DOR) for latching data to be written into FM, (2) the memory address register (MAR) for latching a FM address for reading or writing FM, (3) the jump address register (JAR) which stores a program jump address and (4) the incrementor register (INC) which stores the current microprogram counter value plus one for program sequencing. The IBUS can be driven from either the microinstruction constant field or the logical OR of the RAM output and the FPGA output when reading FM. IBUS data can be clocked into the DOR, MAR or JAR.

![Architecture of the Minimal Processor](image)

**Figure 2.16. Architecture of the Minimal Processor**

The output of the DBUS multiplexer drives the data bus when writing FM. It can select either from the DOR or the microinstruction constant field. Similarly, the ADDR multiplexer drives the FM address lines and can select from either the MAR or the
microinstruction constant field. The PC multiplexer defines the microprogram counter which drives the microprogram RAM address selecting the location of the next microinstruction to execute. The PC multiplexer can gate either the contents of the JAR for a jump or the INC register for sequencing.

The microinstruction format of our mP is a fixed format and length and is addressed as a four byte word, with the first word defining the opcode and the second providing the value in the constant field. (The upper byte of the opcode is unused so only three byte-wide RAMs are necessary for storing the microprogram.) This two-word fixed format allows us to use any microprocessor assembly language assembler that provides an "equate" and "define word" pseudo-instruction in its repertoire.

2.6.2 Minimal Processor Instruction Set

Program assignment statements consist of a term on the right side (of the ':=' ) whose value must be fetched or computed, and a variable on the left side that indicates where the right side value is to be stored. The right side term may be a constant, a variable, or the result of a computed expression, which in FM is accessed just like other variables (by reading a unique address). Array element addresses must also be computed before the particular element can be accessed. The FM computes the address of the element and the processor uses that address to access the element. Therefore, the processor must not only be able to store constants, access FM via a specific address, but it also must be able to read and write locations indirectly.

For program control, the processor minimally must be able to sequence, jump to a location computed in FM and halt. Table 2.1 describes how our mP provides a minimal set of move and control instructions for implementing any program.
The "DC" instruction is used to store a constant into a scalar variable location. A scalar variable is distinguished from an array element as a scalar's address is known at compile-time. First it loads the constant (from the microinstruction constant field) into the data output register (DOR). Next, with the microinstruction constant field now specifying the variable address, a write transaction is initiated, storing the contents of the DOR into the variable location. As the last column of the table indicates, the "DC" instruction takes two cycles to complete.

The "DD" and "DE" instructions are used to store the contents of a scalar variable or computed expression (respectively) into another scalar variable. First the DOR is loaded by specifying the source variable or expression address in the microinstruction constant field and initiating a FM read. Next, with the destination variable address specified in the microinstruction constant field, the contents of the DOR are written into FM.

A "DI" instruction is used for storing the contents of an array element into a scalar variable. First the location containing the array element address is read and stored into the memory address register (MAR). Next, the contents of the location specified in the MAR is read and stored in the DOR. DOR now contains the array element value. Finally, the contents of the DOR are stored into the destination location. This instruction is also used in cases where an array element is required for an expression calculation, but the array itself isn't stored in the FPGA. In this case, a separate location is allocated for each array reference appearing in an expression. If the array element or a variable used in its address specification changes, it must be updated, which can be accomplished using a "DI" instruction.

An "IC" instruction is used to load an array element with a constant. First the location containing the array element address is read and stored into the MAR. Next, the constant
value (specified in the microinstruction constant field) is written into the address specified by the MAR.

The "ID" and "IE" instructions store the value of a scalar variable or a computed expression (respectively) into an array element. First the MAR is loaded with the array element address from FM. Next, the DOR is loaded with the scalar variable or expression value from FM. Finally, the value in the DOR is written in the address specified in the MAR.

The "II" command is used for moving one array element to another. First the MAR is loaded with the address of the source element read from FM. Second, the source element, as specified in the MAR, is read from FM and loaded into the DOR. Third the MAR is loaded with the address of the destination element from FM. Fourth, with the DOR containing the source value and the MAR containing the address of the destination element, a FM write is initiated.

For program control, when any of the move instructions are being executed, the next microinstruction is implied to be the one following the one being executed. For program branching, a processor must be able to jump to program locations where the address is computed. The "JI" instruction loads the microprogram counter with the contents of the FM location whose address is specified in the microinstruction constant field. In our implementation, a microinstruction pipeline register is used to eliminate the delay associated with reading the microprogram RAMs which allows the instruction period to be reduced. This results in a one cycle delay for jumps to be taken. In certain circumstances, the first microinstruction at the target address can be moved to replace the NOP following the jump microinstruction.
<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Operand #1</th>
<th>Operand #2</th>
<th>μCode Instructions</th>
<th># Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>( i := 1 )</td>
<td>Direct Address e.g., ( i )</td>
<td>Constant e.g., 1</td>
<td>DOR ( \leftarrow 1 ) (( i )) ( \leftarrow ) DOR</td>
<td>2</td>
</tr>
<tr>
<td>DD</td>
<td>( i := j )</td>
<td>Direct Address e.g., ( i )</td>
<td>Direct Address e.g., ( j )</td>
<td>DOR ( \leftarrow (j) ) (( i )) ( \leftarrow ) DOR</td>
<td>2</td>
</tr>
<tr>
<td>DE</td>
<td>( i := j + k )</td>
<td>Direct Address e.g., ( i )</td>
<td>Direct Address e.g., ( j + k )</td>
<td>DOR ( \leftarrow (j + k) ) (( i )) ( \leftarrow ) DOR</td>
<td>2</td>
</tr>
<tr>
<td>DI</td>
<td>( i := a[j] )</td>
<td>Direct Address e.g., ( i )</td>
<td>Indirect Address e.g., ( @a[j] )</td>
<td>MAR ( \leftarrow (@a[j]) ) DOR ( \leftarrow (\text{MAR}) ) (( i )) ( \leftarrow ) DOR</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>&quot;a[j+1]&quot; := ( a[j+1] )</td>
<td>Direct Address e.g., &quot;a[j+1]&quot;</td>
<td>Indirect Address e.g., ( @a[j+1] )</td>
<td>MAR ( \leftarrow (@a[j+1]) ) DOR ( \leftarrow (\text{MAR}) ) (&quot;a[j+1]&quot;) ( \leftarrow ) DOR</td>
<td>3</td>
</tr>
<tr>
<td>IC</td>
<td>( a[i] := 1 )</td>
<td>Indirect Address e.g., ( @a[i] )</td>
<td>Constant e.g., 1</td>
<td>MAR ( \leftarrow (@a[i]) ) (( \text{MAR} )) ( \leftarrow 1 )</td>
<td>2</td>
</tr>
<tr>
<td>ID</td>
<td>( a[i+1] := j )</td>
<td>Indirect Address e.g., ( @a[i+1] )</td>
<td>Direct Address e.g., ( j )</td>
<td>MAR ( \leftarrow (@a[i+1]) ) DOR ( \leftarrow (j) ) (( \text{MAR} )) ( \leftarrow ) DOR</td>
<td>3</td>
</tr>
<tr>
<td>IE</td>
<td>( a[i+1] := j + k )</td>
<td>Indirect Address e.g., ( @a[i+1] )</td>
<td>Direct Address e.g., ( j + k )</td>
<td>MAR ( \leftarrow (@a[i+1]) ) DOR ( \leftarrow (j + k) ) (( \text{MAR} )) ( \leftarrow ) DOR</td>
<td>3</td>
</tr>
<tr>
<td>II</td>
<td>( a[i] := a[i+1] )</td>
<td>Indirect Address e.g., ( @a[i] )</td>
<td>Indirect Address e.g., ( @a[i+1] )</td>
<td>MAR ( \leftarrow (@a[i+1]) ) DOR ( \leftarrow (\text{MAR}) ) MAR ( \leftarrow (@a[i]) ) (( \text{MAR} )) ( \leftarrow ) DOR</td>
<td>4</td>
</tr>
<tr>
<td>JI</td>
<td>GOTO (\textit{location})</td>
<td>Indirect Address e.g., \textit{location}</td>
<td>N/A</td>
<td>μPC ( \leftarrow ) (\textit{location}) NOP*</td>
<td>2</td>
</tr>
<tr>
<td>EX</td>
<td>HALT</td>
<td>N/A</td>
<td>N/A</td>
<td>μPC ( \leftarrow ) μPC μPC ( \leftarrow ) μPC−4*</td>
<td>2</td>
</tr>
</tbody>
</table>

* - necessary because of pipelining

52
Finally, the "EX" (for "exit") instruction in our implementation causes the processor to halt by entering an endless loop with the "DONE" signal held active, indicating to the system processor that the program has terminated.

The last column in Table 2.1 lists the number of cycles each mP instruction takes. These values can be divided by the FMC oscillator megahertz (MHz) to determine how long each instruction takes to execute. For example, a "DC" instruction on a 4 MHz FMC takes 500 nS to execute (whereas on a 40 MHz FMC it would take 50 nS). An "II" instruction on a 4 MHz FMC would take 1 μS. In later chapters we use these figures to compute and verify execution times for sample programs executing on our FMC.

2.6.3 Microinstruction Set Implementation

A microinstruction set can be built up from the operation codes necessary for performing the "μCode Instructions" listed in fifth column of Table 2.1. The registers and data paths in Figure 2.16 are controlled by the "Op Code Register" shown again in Figure 2.17. There are seven relevant bits which provide signals for gating multiplexer paths for moving operands between the registers and the functional memory.

As summarized in Figure 2.17, the "Write FM" activates the write signal to the functional memory. The data to be written originates either from the microinstruction constant field or the data output register (see Figure 2.16), depending on the state of the "DBUS Mux" bit. The address to be written originates from either the microinstruction constant field or the memory address register, depending on the state of the "ADDR Mux" bit.

The "IBUS Mux" bit specifies the contents of the minimal processor's internal data bus, which can contain either the contents of the microinstruction constant field or the contents of the external RAM/XILINX data bus. The contents of the internal data bus can
be clocked into either the data output register, which occurs when the Destination Register Select (DRS) bits are 01. When the DRS bits are 10, the memory address register gets clocked, and when DRS = 11, the program counter gets the contents of the internal data bus.

**Op Code Register**

<table>
<thead>
<tr>
<th>Bit State</th>
<th>Operation Accomplished</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write FM = 1</td>
<td>RAM and possibly a XILINX register are written</td>
</tr>
<tr>
<td>DBUS Mux = 0</td>
<td>Microinstruction constant field is written</td>
</tr>
<tr>
<td>DBUS Mux = 1</td>
<td>Contents of the data output register is written</td>
</tr>
<tr>
<td>ADDR Mux = 0</td>
<td>Microinstruction constant field defines the address</td>
</tr>
<tr>
<td>ADDR Mux = 1</td>
<td>Memory address register defines the address</td>
</tr>
<tr>
<td>IBUS Mux = 0</td>
<td>Microinstruction constant field drives internal data bus</td>
</tr>
<tr>
<td>IBUS Mux = 1</td>
<td>RAM/XILINX output drives internal data bus</td>
</tr>
<tr>
<td>Destination Reg Select = 01</td>
<td>Data output register is written</td>
</tr>
<tr>
<td>Destination Reg Select = 10</td>
<td>Memory address register is written</td>
</tr>
<tr>
<td>Destination Reg Select = 11</td>
<td>Program counter is written</td>
</tr>
<tr>
<td>Done = 1</td>
<td>System processor can see mP is done executing</td>
</tr>
</tbody>
</table>

**Figure 2.17. Microinstruction Control Register**

To facilitate the easy translation of action stubs into the ones and zeros of the microinstruction for gating the operands through the processor, microinstruction mnemonic codes were defined for each bit pattern used for accomplishing the operations listed in Figure 2.17. Table 2.2 lists the 10 microinstruction codes used to implement the minimal processor (macro) instruction set.

The system processor (MCS-51) assembler is used to translate equated microinstruction op-codes, followed by 16 bit constants into a HEX file of four byte
microinstructions. This file is downloaded directly into the microprogram memory of the minimal processor for execution.

2.6.4 Minimal Processor Implementation

Implementing the minimal processor was similar to the approach used for implementing the functional memory FPGAs. All the registers and multiplexers, as shown in a detailed block diagram in Figure 2.18, were specified using PALASM equations. The detail of this figure reveals more than Figures 2.15 and 2.16. When the board is selected (\(\neg\)BoardSel=0), the \(\mu\)PAccess line from the system processor controls the set of outer multiplexers used to switch the processor into “transparent mode” allowing the system processor to access directly either the microprogram RAMs (FM/\(\mu\)C=0) or the functional memory (FM/\(\mu\)C=1). This signal is necessary for loading the microprogram and initializing data values and arrays in functional memory.

Figure 2.18 also shows logically how the opcode register is decoded. The first (leftmost) opcode register bit controls the write signal to the functional memory. The second bit controls the multiplexer that selects whether the data bus lines to the functional memory (RAM and FPGAs) are driven by the data output register (DOR) or the \(\mu\)nstruction constant register. The third opcode register bit selects whether the address bus to the functional memory is driven by the memory address register (MAR) or the constant register. The fourth opcode bit selects which source drives the internal bus, either the constant register or the external data bus input from the functional memory. The fifth and sixth bits are decoded to select which register (if any) is clocked with new data from the internal bus. If the fifth and sixth bits are 01, 10 or 11, the DOR, MAR or jump address register (JAR) clock is enabled, respectively.

The minimal processor was specified as a generic module using PALASM equations. This was done for convenience and to allow the same minimal processor design to be
incorporated into other designs in the future, such as a single chip which contains both a minimal processor and functional memory. In our current implementation, the minimal processor PALASM module resides in its own 175-pin XC 3090. It interfaces directly to the system processor as shown on the left side of the diagram in Figure 2.18 (at the bottom of the page).

Table 2.2. Microinstructions for Implementing mP Operations

<table>
<thead>
<tr>
<th>µI Op Code</th>
<th>OpCode Register</th>
<th>Example Usage</th>
<th>µCode Implementation Example (see Table 2.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC</td>
<td>0000 0100</td>
<td>LDC, 1</td>
<td>DOR ← 1. Load the Data output register with a constant. Used in the DC instruction</td>
</tr>
<tr>
<td>LDA</td>
<td>0001 0100</td>
<td>LDA, j</td>
<td>DOR ← (j). Load the Data output register with the contents of the Address specified. Used in the DD, DE, ID and IE instructions</td>
</tr>
<tr>
<td>LDM</td>
<td>0011 0100</td>
<td>LDM, 0</td>
<td>DOR ← (MAR). Load the Data output register with the contents of the address specified in the Memory address register. Used in the DI and II instructions</td>
</tr>
<tr>
<td>LMA</td>
<td>0001 1000</td>
<td>LMA, @a[i]</td>
<td>MAR ← (@a[i]). Load the Memory address register with the contents of the Address specified. Used in the DI, IC, ID, IE and II instructions</td>
</tr>
<tr>
<td>WMD</td>
<td>1110 0000</td>
<td>WMD, 0</td>
<td>(MAR) ← DOR. Write the memory address specified in the Memory address register, with the contents of the Data output register. Used in the ID, IE and II instructions</td>
</tr>
<tr>
<td>WAD</td>
<td>1100 0000</td>
<td>WAD, i</td>
<td>(i) ← DOR. Write the Address specified (in the constant field) with the contents of the Data output register. Used in the DC, DD, DE, and DI instructions</td>
</tr>
<tr>
<td>WMC</td>
<td>1010 0000</td>
<td>WMC, 1</td>
<td>(MAR) ← 1. Write the memory address specified in the Memory address register, with a Constant value. Used in the IC instruction</td>
</tr>
<tr>
<td>JPI</td>
<td>0001 1100</td>
<td>JPI, @Rule</td>
<td>µPC ← (@Rule). Jump to the location specified Indirectly. Used following each rule to jump to the next rule</td>
</tr>
<tr>
<td>HALT</td>
<td>0000 1101</td>
<td>HALT, $</td>
<td>µPC ← µPC. Jump to the specified location and set the &quot;Done&quot; flag. Used to terminate execution and to notify the system processor</td>
</tr>
<tr>
<td>NOP</td>
<td>0000 0000</td>
<td>NOP, 0</td>
<td>NOP. Do nothing for one cycle. Used following the JPI for pipelining</td>
</tr>
</tbody>
</table>
Figure 2.18. Minimal Processor – Detailed Block Diagram
2.6.5 System Processor Interface

The system processor is an 11.059 MHz Intel MCS-51 8031 (i8031) microprocessor. The minimal processor control bits are connected directly to the system processor’s Port 1 register, as indicated in Figure 2.19. All the bits are set to a ‘1’ state when the machine is reset. Bit 1 connects to the \( \neg \text{START} \) signal of the mP. When \( \neg \text{START} = 1 \), the mP’s \( \mu \text{ProgAddr} \) bus is held at zero. Setting \( \neg \text{START} = 0 \) allows the mP to execute.

\( \mu \text{PAccess} = 1 \) disconnects the mP from the microprogram RAM and FM, allowing the system processor direct access. If \( \text{FM/\mu C} = 1 \), then the system processor can access the FM directly, and when \( \text{FM/\mu C} = 0 \), then the system processor can access the microprogram RAMs. \( \mu \text{PAccess} = 0 \) disconnects the system processor and allows the mP normal access to the microprogram RAM and FM. RED/GRN controls the color of the light emitting diode (LED) on the system processor board. DONE is connected directly to bit 0 of the microinstruction opcode register and is used to signal the system processor when the mP has finished executing a program. Since this signal is “open-collector,” the processor should always place a ‘1’ in this bit position when writing to the Port 1 register.

![System Processor Port 1 Register](image)

Figure 2.19. System Processor - Minimal Processor Interface

The system processor interfaces to the user’s terminal emulation software through a 19.2K bits per second serial port. Figure 2.20 shows the opening screen of the System
Monitor we have implemented as the operating system for the system processor. A menu
presents a list of 14 user commands for loading and debugging programs. The ‘X’
command is used to download a XILINX .MCS file into the FPGAs. The ‘L’ command
is used to download microcode to the FMC board or download system processor 8031
programs. Both the ‘X’ and ‘L’ command must have a two hex digit board identifier
immediately following. (Our system can accommodate up to four parallel FMC boards.)
A ‘G’ followed by a four digit start address is used to execute system processor “driver”
programs for testing FMC programs. The ‘T’ command is the same as ‘G’ except the
start, ending and elapsed execution times from the system clock are also reported.

Figure 2.20. System Monitor - User Interface

After the .MCS FPGA module has been loaded using the ‘X’ command, and the mP
microcode has been loaded using the ‘L’ command, the ‘P’ command can be used to set
and clear bits in the Port 1 register described in Figure 2.19. The lower 32K of the system processor RAM maps directly into the FMC's functional memory and therefore can be accessed and changed (using the 'D' and 'C' commands) by first entering 'PFF' to set Port 1 register bits 4 (μPAccess) and 5 (FM/μP-RAM) to '1'. To execute a program, entering 'PED' sets the mP having access to the microprogram and FM (bit 4 = 0) and starts the mP (bit 1 = 0). Entering 'P' alone reports the value of the Port 1 register allowing the user to observe when the minimal processor has completed executing the program (bit 7 = 1).

The system monitor presented here proved quite useful for debugging, executing and testing the FMC programs described in Chapters 3, 4, 5, 6 and 7. For each example, "driver" programs were written in MCS-51 assembly language to initialize the arrays, execute the FMC program (by directly manipulating the Port 1 register) and verify the results.
In this chapter we introduce decision table programming on a FMC by explaining the process of translating high level language programs into the minimal processor microcode and the FPGA PALASM nanocode for execution on our FMC. One complication when writing high level language programs for a FMC is that array references within expressions cannot be implemented "as expressed" unless the entire array can be captured in FPGA input registers. If the array is too large, then temporary scalar variables (which can be captured in FPGA registers) must be declared for each array reference used in an expression (including condition stubs). In the next chapter on nondeterminism we explore the advantages of capturing the entire array in FPGA input registers. In this chapter, however, we will allocate temporary scalar locations instead.

To illustrate the basic ideas, we will translate a bubble sort and a shortest path program into FMC code. These two examples both contain array references. In the bubble sort example, we find it is to the programmer's advantage to declare explicitly the temporary scalar variables replacing the array references in expressions and to manage their updating. Normally, the exchange operation (see Figure 3.1) would be performed by allocating a single temporary variable which would be used for exchanging two out-of-order array elements. Here, however, the programmer saves execution time by taking advantage of the fact that temporary variables already exist for the two array elements that need to be exchanged.

In the shortest path example, we will treat the allocation of temporary variables as a "compiler problem." The compiler would be responsible for replacing each array reference in the expression with a newly allocated scalar. The compiler would also
determine when the scalar had to be updated, based on the possibility that the array element itself or its index has changed in value.

3.1 BUBBLE SORT PROGRAM EXAMPLE

Figure 3.1 shows both an ordinary Pascal and decision table version of a deterministic bubble sort algorithm. Recognize that the multi-way branch feature with decision tables allows us to reduce the level complexity from four to two [Lew, 1982] simplifying the implementation by reducing the level complexity. In effect, the two-nested loop structure in un-nested, as illustrated in Figure 3.2.

A. Pascal

```pascal
var a : array[0..50] of byte;
    n : byte;
procedure DetBubSort;
{Sort a[1..n] in ascending order}
var j, k : byte;
begin
    for k := n downto 2 do
        for j := 1 to k-1 do
                exchange(A[j],A[j+1])
end
```

B. Decision Table (reduced)

<table>
<thead>
<tr>
<th>Rule</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>k = 1</td>
<td></td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>j = k</td>
<td></td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Figure 3.1. Bubble Sort Program

As we see in the decision table, variable k is initialized at n and j is initialized at 1 when lambda=0. This corresponds to the top box of the flow chart in Figure 3.2. The single test node corresponds to the remaining condition stub and entry portion of the decision table, from which either Rule 2, 3, 4 or 5 will be selected to execute. If either Rules 3, 4 or 5 execute, the node is tested again and a new rule is selected. If Rule 2 executes, the program terminates.
Rules 3, 4 and 5 sort the array by bubbling the largest element to the top of the array. By each time decrementing the "array-to-be-sorted" index by one, the entire array becomes completely sorted after \( n - 1 \) bubbling passes (for a size \( n \) array). Rule 3 starts a bubbling pass by initializing \( j \) (index of bubbling element) and \( k \) (final resting position for bubbled element for this pass). Rules 4 and 5 perform the bubbling action by exchanging elements if necessary.

\[
\begin{align*}
k &:= n \\
j &:= 1
\end{align*}
\]

\[\begin{array}{c}
k := k - 1 \\
j := j + 1
\end{array}\]

Figure 3.2. Bubble Sort Flow Chart

Figure 3.3 shows the bubble sort program modified for execution with functional memory. Notice variables \( t \) and \( t1 \) are used to hold the current values of \( a[j] \) and \( a[j+1] \) respectively, because they are needed in the \( a[j] > a[j+1] \) conditional expression.

Normally, the exchange(\( a[j], a[j+1] \)) operation would be implemented by using a single temporary variable \( t \) by setting \( t := a[j] \), \( a[j] := a[j+1] \) and \( a[j+1] := t \). The fact that \( t \) and \( t1 \) exist, however, allows the exchange to be implemented more efficiently in two array references instead of four.

In the next two sections we describe how the minimal processor microcode and the FPGA nanocode are produced for the decision table bubble sort program in Figure 3.3.
3.1.1 Bubble Sort Functional Memory Map

The first step in translating the bubble sort program to execute on a FMC is to generate the Functional Memory Map shown in Table 3.1. Entered are both the left and right sides of the assignment statements as names of memory locations. Rows 1 and 2 are the lambda and @rule variables necessary for the execution of the decision table. lambda is a two bit input register in the FPGA and @rule is an expression output providing the microprogram address for the code of the next rule to execute. Row 3 contains n, which is not used in any expression therefore a FPGA register does not need to be allocated for it. Row 4 contains the a array allocation. Its address, 0006, will be used as a constant value for computing references to elements. Rows 5 through 8 list the four scalar variables used as operands in expressions, therefore, input registers inside the FPGA chip will be allocated for each. Note that since our FMC has a 16-bit data path between the processor and memory, and operands j, k, t and t1 are only 8 bits, each is actually is stored in the odd byte location immediately following the one listed (e.g., j

\[
\begin{array}{ll}
\text{lambda} &= 1 2 3 4 5 \\
\text{k} &= 0 1 1 1 1 \\
\text{j} &= - T F F F \\
\text{t} &= - - T F F \\
\text{t1} &= - - - T F \\
\end{array}
\]

\[
\begin{array}{ll}
k := n & X - - - - \\
j := 1 & X - X - - \\
k := k-1 & - X - - \\
a[j] := t1 & - - X - \\
a[j+1] := t & - - X - \\
j := j+1 & - - X X \\
t := a[j] & X - X X X \\
t1 := a[j+1] & X - X X X \\
extit & - X - - - \\
\text{lambda} := 1 - - - - \\
\end{array}
\]

Figure 3.3. Bubble Sort Program for FM
actually is stored in 006D). Rows 9 through 12 list the four output expressions for the program. \( k-1 \) appears at location 0074 and is updated immediately when \( k \) at location 006E is written. The addresses of \( a[j] \) and \( a[j+1] \), denoted with the ' \(@\) ' symbol preceding them, appear at locations 0076 and 0078 respectively. \( j+1 \) appears at location 007A. \( @a[j], @a[j+1] \) and \( j+1 \) all update immediately when \( j \) at location 006C is written.

<table>
<thead>
<tr>
<th>Name</th>
<th>Address (hex)</th>
<th>Dimension</th>
<th>Operand Width</th>
<th>Functional Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda</td>
<td>0000</td>
<td>-</td>
<td>2 bit</td>
<td>input register</td>
</tr>
<tr>
<td>@rule</td>
<td>0002</td>
<td>-</td>
<td>7 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>n</td>
<td>0004</td>
<td>-</td>
<td>8 bit</td>
<td>-</td>
</tr>
<tr>
<td>a</td>
<td>0006</td>
<td>[0..50]</td>
<td>8 bit</td>
<td>address constant</td>
</tr>
<tr>
<td>j</td>
<td>006C</td>
<td>-</td>
<td>8 bit</td>
<td>input register</td>
</tr>
<tr>
<td>k</td>
<td>006E</td>
<td>-</td>
<td>8 bit</td>
<td>input register</td>
</tr>
<tr>
<td>t</td>
<td>0070</td>
<td>-</td>
<td>8 bit</td>
<td>input register</td>
</tr>
<tr>
<td>tl</td>
<td>0072</td>
<td>-</td>
<td>8 bit</td>
<td>input register</td>
</tr>
<tr>
<td>k-1</td>
<td>0074</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@a[j]</td>
<td>0076</td>
<td>-</td>
<td>9 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@a[j+1]</td>
<td>0078</td>
<td>-</td>
<td>9 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>j+1</td>
<td>007A</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
</tbody>
</table>

### 3.1.2 Bubble Sort Execution Table

All the information that the compiler must consolidate before code generation is summarized in the Execution Table shown as Table 3.2. Each row represents one minimal processor instruction. The first column identifies the starting entry point of the code for each rule. Column 2 lists the action stub statements collected for each rule. The minimal processor instruction type code is listed in column 3. Processor instructions have one or two operands (one identifying the destination address and one specifying the source operand), which are identified in columns 4 and 5. The minimal processor
microcode can be generated directly from columns 3, 4 and 5. The number of cycles each instruction consumes is listed in column 6. From this column the actual execution time for each rule can be calculated.

Table 3.2. Bubble Sort Execution Table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Statement</th>
<th>mP</th>
<th>Inst</th>
<th>Des Addr</th>
<th>Source</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule1</td>
<td>k := n</td>
<td>DD</td>
<td>006E</td>
<td>0004</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>j := 1</td>
<td>DC</td>
<td>006C</td>
<td>0001</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>lambda := 1</td>
<td>DC</td>
<td>0000</td>
<td>0001</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>t := a[j]</td>
<td>DI</td>
<td>0070</td>
<td>0076</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>t1 := a[j+1]</td>
<td>DI</td>
<td>0072</td>
<td>0078</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>goto @rule</td>
<td>JI</td>
<td>0002</td>
<td>-</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Rule2</td>
<td>EXIT</td>
<td>EX</td>
<td>-</td>
<td>-</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Rule3</td>
<td>j := 1</td>
<td>DC</td>
<td>006C</td>
<td>0001</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>k := k-1</td>
<td>DE</td>
<td>006E</td>
<td>0074</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>t := a[j]</td>
<td>DI</td>
<td>0070</td>
<td>0076</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>t1 := a[j+1]</td>
<td>DI</td>
<td>0072</td>
<td>0078</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>goto @rule</td>
<td>JI</td>
<td>0002</td>
<td>-</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Rule4</td>
<td>a[j] := t1</td>
<td>ID</td>
<td>0076</td>
<td>0072</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>a[j+1] := t</td>
<td>ID</td>
<td>0078</td>
<td>0070</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>j := j+1</td>
<td>DE</td>
<td>006C</td>
<td>007A</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>t := a[j]</td>
<td>DI</td>
<td>0070</td>
<td>0076</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>t1 := a[j+1]</td>
<td>DI</td>
<td>0072</td>
<td>0078</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>goto @rule</td>
<td>JI</td>
<td>0002</td>
<td>-</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Rule5</td>
<td>j := j+1</td>
<td>DE</td>
<td>006C</td>
<td>007A</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>t := a[j]</td>
<td>DI</td>
<td>0070</td>
<td>0076</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>t1 := a[j+1]</td>
<td>DI</td>
<td>0072</td>
<td>0078</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>goto @rule</td>
<td>JI</td>
<td>0002</td>
<td>-</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Since the FPGA must calculate rule addresses (when computing the @rule), the microcode starting addresses must be known before the PALASM nanocode can be generated. Table 3.3 summarizes the rule selection logic for generating the rule address @rule. The table shows that Rule 1 starts at μcode hex location 04. This equals 0000 0100 in binary. Rule 1 fires when lambda L1L0=00. Rule 2 starts at hex location 3C,
which in binary is 0011 1100. Rule 2 fires when \( L_1 L_0 = 01 \) and condition stub \( C_2 \) is true. Rule 3 starts at \( \mu \) code address 44 when \( L_1 L_0 = 01 \), \( C_2 \) is false and \( C_3 \) is true. Rules 4 and 5 are also expressed.

### Table 3.3. Bubble Sort Rule Selection Logic

<table>
<thead>
<tr>
<th>Label Entry Point</th>
<th>( \mu ) Code Hex Address</th>
<th>( \mu ) Code Address Bits</th>
<th>( L_1 )</th>
<th>( L_0 )</th>
<th>( C_2 )</th>
<th>( C_3 )</th>
<th>( C_4 )</th>
<th>Expression Causing Rule To Fire</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>04</td>
<td>0000 0100</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(/L_1*/L_0)</td>
</tr>
<tr>
<td>Rule 2</td>
<td>3C</td>
<td>0011 1100</td>
<td>0</td>
<td>1</td>
<td>T</td>
<td>-</td>
<td>-</td>
<td>(/L_1<em>L_0</em>C_2)</td>
</tr>
<tr>
<td>Rule 3</td>
<td>44</td>
<td>0100 0100</td>
<td>0</td>
<td>1</td>
<td>F</td>
<td>T</td>
<td>-</td>
<td>(/L_1<em>L_0</em>/C_2*C_3)</td>
</tr>
<tr>
<td>Rule 4</td>
<td>74</td>
<td>0111 0100</td>
<td>0</td>
<td>1</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>(/L_1<em>L_0</em>/C_2*/C_3*C_4)</td>
</tr>
<tr>
<td>Rule 5</td>
<td>B4</td>
<td>1011 0100</td>
<td>0</td>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>(/L_1<em>L_0</em>/C_2*/C_3*/C_4)</td>
</tr>
</tbody>
</table>

### 3.1.3 Bubble Sort Expression Graphs

Expression graphs are a useful notation to describe the combinational logic flow for computing each expression. For computing the address of the next rule to execute (i.e., @rule at location 0002), we begin by evaluating the condition stubs as shown in Figure 3.4A. The input registers on top of each graph capture and store values when written by the processor to the addresses indicated. Operators (shown rectangular with rounded corners) consist of combinational logic and have no storage capability. The output of each of the ‘=1’, ‘=’ and ‘>’ operators is a boolean value used along with \( \lambda \) to calculate the rule address. The equations defining each @rule bit (\( R_0, R_1, ..., R_7 \)) are shown. From Table 3.3, we know when each bit should be a one. \( R_0 \) and \( R_1 \) are always zero and \( R_2 \) is always one no matter which rule is selected. \( R_3 \) is one only when Rule 2 is selected, therefore \( R_3 \) equals the boolean expression for Rule 2 (\( \lambda = 01 \) and condition stub \( C_2 \) is true). \( R_4 \) and \( R_5 \) are one when Rules 2, 4 and 5 should fire.
therefore they both equal the logical OR of these three boolean expressions. \( R_6 \) is one when Rules 3 or 4 should fire. \( R_7 \) is one when Rule 5 is to fire.

A. Rule Address Calculation

\[
\begin{align*}
R_0 &= 0 \\
R_1 &= 0 \\
R_2 &= 1 \\
R_3 &= (/L_1 \times L_0 \times C_2) \\
R_4 &= (/L_1 \times L_0 \times C_2) + (/L_1 \times L_0 \times /C_2 \times /C_2 \times /C_4) \\
R_5 &= (/L_1 \times L_0 \times C_2) + (/L_1 \times L_0 \times /C_2 \times /C_2 \times /C_4) \\
R_6 &= (/L_1 \times L_0 \times /C_2 \times /C_2 \times /C_4) \\
R_7 &= (/L_1 \times L_0 \times /C_2 \times /C_2 \times /C_4)
\end{align*}
\]

B. Array Address Calculation

C. Assignment Expressions

Figure 3.4 also shows the expression graphs for the rest of the expression computations in the bubble sort program. Figure 3.4B shows the computation of \( \text{addr}[j] \) (the address of \( a[j] \)) and \( \text{addr}[j+1] \). Figure 3.4C shows the only two expressions from the right sides of assignment statements. \( k-1 \) is computed by connecting a decrement
operator to a \( k \) input register. \( j+1 \) is computed using an increment operator connected to a \( j \) input register.

### 3.1.4 Bubble Sort FPGA Code Generation

The expression graphs in Figure 3.4 are helpful for visualizing the combinational logic for computing expressions but these must be translated into a language the FPGA design tools can read. Figure 3.5 shows the "operator macros" used for the five different operators required in our bubble sort program. Each was derived from the more general PALASM equations described in Chapter 2 for Add and Not-Equal. Figure 3.5A shows the equations for testing if an 8-bit value represented by bits K7, ..., K0 is equal to one. If true, then \( C=1 \), otherwise \( C=0 \). When generating the PALASM file the bit name letter prefixes are replaced with signal names that are derived from a unique unit number assigned to each operator.

Figure 3.5D shows the increment by one operator used for implementing the \( j+1 \) expression graph in Figure 3.4C. Figure 3.6 shows all the PALASM code that is generated (i.e., required) when implementing this expression graph. Box 1 on top shows the address select for writing the \( j \) input register as well as the decoding and output enable for the \( j+1 \) multiplexer output gates. Signal \( \text{sel}_06C \) will be true when a 006C appears on the address bus from the processor. When \( \text{sel}_06C \) is logically ANDed with the write signal \( /WRLC \), the processor is storing a value into \( j \). Signal \( \text{sel}_07A \) is true when 007A appears on the address bus. When \( \text{sel}_07A \) is true with the read signal \( /RDC \), the processor is attempting to read the result of \( j+1 \) from functional memory.

The address bit input lines A15, A14, ..., A1 correspond to the pin signal definitions for the FPGA program shell shown in Figure 2.3. They are logically ANDed in nibbles. \( \text{ASX}0 \) is true when the highest order address nibble (A12, A13, A14 and A15) are all
zero. ASH0 is true when A8, A9, A10 and A11 are all zero. Signal sel_06C also needs ASM6, indicating A4, A5, A6 and A7 equals hexadecimal 6, and ASLC, indicating A0, A1, A2 and A3 equals hexadecimal C. sel_07A also needs ASM7 and ASLA to be decoded.

### A. 8 Bit Equal To 1

<table>
<thead>
<tr>
<th>C = K=1  {2 CLBS}</th>
<th>C = J=K  {5 CLBS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>;--1st level of K=1</td>
<td>;--1st level of Equal To</td>
</tr>
<tr>
<td>C73 = /K7*/K6*/K5*/K4*/K3</td>
<td>C76 = (J7++:K7) + (J6++:K6)</td>
</tr>
<tr>
<td>;--2nd level of K=1</td>
<td>C54 = (J5++:K5) + (J4++:K4)</td>
</tr>
<tr>
<td>C = C73*/K2*/K1*K0</td>
<td>C32 = (J3++:K3) + (J2++:K2)</td>
</tr>
<tr>
<td></td>
<td>C10 = (J1++:K1) + (J0++:K0)</td>
</tr>
<tr>
<td></td>
<td>;--2nd level of Equal To</td>
</tr>
<tr>
<td></td>
<td>C = /(C76 + C54 + C32 + C10)</td>
</tr>
</tbody>
</table>

### B. 8 Bit Equal To Comparison

<table>
<thead>
<tr>
<th>C = A+1  {6 CLBS}</th>
<th>C = A-1  {6 CLBS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>;--1st level of B=A+1</td>
<td>;--1st level of B=A-1</td>
</tr>
<tr>
<td>C4 = A3<em>A2</em>A1*A0</td>
<td>C4 = A3*/A2*/A1*/A0</td>
</tr>
<tr>
<td>B0 = /A0</td>
<td>B0 = /A0</td>
</tr>
<tr>
<td>B1 = A1:::A0</td>
<td>B1 = /(A1:::A0)</td>
</tr>
<tr>
<td>B3 = A3::(A2<em>A1</em>A0)</td>
<td>B3 = /(A3::(A2<em>A1</em>A0))</td>
</tr>
<tr>
<td>;--2nd level of B=A+1</td>
<td>;--2nd level of B=A-1</td>
</tr>
<tr>
<td>B4 = A4:::C4</td>
<td>B4 = /(A4:::C4)</td>
</tr>
<tr>
<td>B5 = A5::(A4*C4)</td>
<td>B5 = /(A5::(A4*C4))</td>
</tr>
<tr>
<td>B6 = A6::(A5<em>A4</em>C4)</td>
<td>B6 = /(A6::(A5<em>A4</em>C4))</td>
</tr>
<tr>
<td>B7 = A7::(A6<em>A5</em>A4*C4)</td>
<td>B7 = /(A7::(A6<em>A5</em>A4*C4))</td>
</tr>
</tbody>
</table>

### C. 8 Bit Unsigned Comparison

<table>
<thead>
<tr>
<th>C = J&gt;N  {6 CLBS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>;--1st level of 8 bit comparator</td>
</tr>
<tr>
<td>C2 = /N1<em>J1+/(N1+J1)</em>/N0*J0</td>
</tr>
<tr>
<td>C420 = /N3<em>J3+/(N3+J3)</em>/(N2*J2)</td>
</tr>
<tr>
<td>C421 = /N3<em>J3+/(N3+J3)</em>/(N2+J2)</td>
</tr>
<tr>
<td>C640 = /N5<em>J5+/(N5+J5)</em>/(N4*J4)</td>
</tr>
<tr>
<td>C641 = /N5<em>J5+/(N5+J5)</em>/(N4+J4)</td>
</tr>
<tr>
<td>C860 = /N7<em>J7+/(N7+J7)</em>/(N6*J6)</td>
</tr>
<tr>
<td>C861 = /N7<em>J7+/(N7+J7)</em>/(N6+J6)</td>
</tr>
<tr>
<td>;--2nd level of 8 bit comparator</td>
</tr>
<tr>
<td>C6 = C640+C641*(C420+C421*C2)</td>
</tr>
<tr>
<td>;--3rd level of 8 bit comparator</td>
</tr>
<tr>
<td>C = C860+C861*C6</td>
</tr>
</tbody>
</table>

### D. Increment by 1

<table>
<thead>
<tr>
<th>B = A+1  {6 CLBS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>;--1st level of B=A+1</td>
</tr>
<tr>
<td>C4 = A3<em>A2</em>A1*A0</td>
</tr>
<tr>
<td>B0 = /A0</td>
</tr>
<tr>
<td>B1 = A1:::A0</td>
</tr>
<tr>
<td>B2 = A2::(A1*A0)</td>
</tr>
<tr>
<td>B3 = A3::(A2<em>A1</em>A0)</td>
</tr>
<tr>
<td>;--2nd level of B=A+1</td>
</tr>
<tr>
<td>B4 = A4:::C4</td>
</tr>
<tr>
<td>B5 = A5::(A4*C4)</td>
</tr>
<tr>
<td>B6 = A6::(A5<em>A4</em>C4)</td>
</tr>
<tr>
<td>B7 = A7::(A6<em>A5</em>A4*C4)</td>
</tr>
</tbody>
</table>

### E. Decrement by 1

<table>
<thead>
<tr>
<th>B = A-1  {6 CLBS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>;--1st level of B=A-1</td>
</tr>
<tr>
<td>C4 = A3*/A2*/A1*/A0</td>
</tr>
<tr>
<td>B0 = /A0</td>
</tr>
<tr>
<td>B1 = /(A1:::A0)</td>
</tr>
<tr>
<td>B2 = /(A2::(A1*A0))</td>
</tr>
<tr>
<td>B3 = /(A3::(A2<em>A1</em>A0))</td>
</tr>
<tr>
<td>;--2nd level of B=A-1</td>
</tr>
<tr>
<td>B4 = /(A4:::C4)</td>
</tr>
<tr>
<td>B5 = /(A5::(A4*C4))</td>
</tr>
<tr>
<td>B6 = /(A6::(A5<em>A4</em>C4))</td>
</tr>
<tr>
<td>B7 = /(A7::(A6<em>A5</em>A4*C4))</td>
</tr>
</tbody>
</table>

Figure 3.5. FPGA Operator Macros Used for Deterministic Bubble Sort
Box 2 in Figure 3.6 shows the code implementing the \( j \) input register. Each bit (\( \text{reg}_06C_0 \) through \( \text{reg}_06C_7 \)) represents a storage flip-flop, clocked on the trailing edge of the write clock from the processor, and enabled when the address lines from the processor select location 006C. Box 4 shows the generic Increment-by-1 macro from
Figure 3.5D expanded to compute $j+1$. Notice that ‘A’ is replaced by ‘reg_06C’, ‘C’ is replaced by ‘s_015c’ and the output ‘B’ is replaced by ‘s_015_’.

Box 4 in Figure 3.6 shows the output multiplexer for this particular FPGA. The output signals DO0 through DO7 correspond to those defined in the FPGA program shell in Figure 2.3. The outputs s_015_0,...,s_015_7 contain the $j+1$ value to be gated out when sel_07A is true (when the processor reads location 007A).

3.1.5 Bubble Sort Compilation Statistics

Once the microcode source code and PALASM source code programs have been built, they can be further compiled or assembled into executable object modules using their respective and appropriate off-the-shelf tools. The instruction format of the minimal processor was designed so any conventional assembler that supports label-equates and define-word pseudo instruction scan be used to translate microinstruction mnemonics into a microcode load module. We use the same assembler chosen for the system processor on our FMC platform which is an Intel MCS-51 based processor. As summarized in Figure 3.7, the minimal processor load module typically is produced in less than 10 seconds.

The PALASM source program is assembled into a .XNF file which then must be merged with the proper .XNF circuit interface shell. FPGA chips 1, 2 and 3 are 175-pin XC3090s and FPGA chip 4 is a 132-pin XC3064. Once merged into a single .XNF file, it is then “placed” and “routed.” As shown in Figure 3.7, placement and routing is by far the longest step in the compilation process using presently available design tools. Placement involves binding the gates and latches as configurable logic blocks (CLBs) to physical CLBs in the chip. This step is complex because if done poorly, not all the interconnections may be routable.
The bubble sort program is small enough to fit into one FPGA. As Table 3.4 shows, the expressions consumed 193 gates and 41 latches in 75 CLBs, taking 1 hour and 51
minutes to place using an Intel 486 25 MHz PC. Gates here may be 2, 3, 4 or 5 input, of any type. There were 557 pins with 133 nets taking 6 minutes and 35 seconds to route. Table 3.4 also shows the program uses a maximum number of 15 levels of combinational logic.

<table>
<thead>
<tr>
<th>FPGA</th>
<th># Gates</th>
<th># Latches</th>
<th>Max Level</th>
<th># CLBs</th>
<th># Pins</th>
<th># Nets</th>
<th>Placement Time (hh:mm:ss)</th>
<th>Routing Time (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSCI</td>
<td>193</td>
<td>41</td>
<td>15</td>
<td>75</td>
<td>557</td>
<td>133</td>
<td>1:51:42</td>
<td>0:06:35</td>
</tr>
</tbody>
</table>

As Figure 3.7 shows, once the FPGA has been placed and routed, a “.BIT” file is produced which is linked with the .BIT file for the minimal processor FPGA specification into one load module. With the processor FPGA wired for peripheral program load mode connecting with the four functional memory FPGAs daisy-chained for serial loading, the system processor can load the single FPGA object module (containing all FPGA programs), byte by byte similar to loading a program into memory, except all the bytes are written to the same location (see [XILINX, 1993] for details).

### 3.2 Shortest Path Program Example

For our second example of the translation and execution of a high level language program on a FMC, a shortest path program which also appeared in [Lew and Halverson, 1994] will be used. In Figure 3.8, on the left is shown a Pascal version and on the right is an equivalent DT version of a shortest path program for a topologically sorted graph with distances stored in a two-dimensional array $d$. Some transformations, as described in [Lew, 1982] and shown in [Halverson and Lew, 1994-2], were made to simplify the DT, but it essentially implements the same underlying dynamic programming algorithm: starting from the target node (node $n$), working back node by node towards the starting
node (node 1), at each stage the shortest path from a node to the target is calculated; see [Bellman and Dreyfuss, 1962] for details.

**Pascal**

```
var d : array[1..80, 1..80] of datatype;
  f : array[1..80] of datatype;
  t : array[1..80] of integer;
  n : integer;
procedure findShortestPath;
{find shortest path from node 1 to n}
var i, j, k : integer;
  min : datatype;
  ptr : integer;
begin
  fen := 0;
  for i := n - 1 downto 1 do
    begin
      min := maxint;
      ptr := n + 1;
      for j := i + 1 to n do
        begin
          if d[i,j] + f[j] < min then
            begin
              min := d[i,j] + f[j]
              ptr := j
            end; {if}
        end; {for j}
      f[i] := min;
      t[i] := ptr;
    end; {for i}
end
```

**Decision Table (reduced)**

<table>
<thead>
<tr>
<th>Rule</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda =</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i &gt; 0</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>j &gt; n</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>d[i,j] + f[j] &lt; min</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

```
f[n] := 0;
for i := n - 1 downto 1 do
  begin
    min := maxint;
    ptr := n + 1;
    for j := i + 1 to n do
      begin
        if d[i,j] + f[j] < min then
          begin
            min := d[i,j] + f[j]
            ptr := j
          end; {if}
      end; {for j}
    f[i] := min;
    t[i] := ptr;
  end; {for i}
```

![Figure 3.8. Shortest Path Program](image)

In the program, array references $d[i,j]$ and $f[j]$ appear in two expressions, so the two scalar variables (denoted “$d[i,j]$” and “$f[j]$” in subsequent tables) must be allocated for each of these references. The expressions would then refer to these scalars, which must be updated whenever the referenced array values or their index values $i$ or $j$ change. Compiling programs with such scalars into efficient code in which updates are only performed as needed is a problem we will not address here.
3.2.1 Shortest Path Functional Memory Map

The first step in translating the shortest path program to execute on a FMC is to generate the Functional Memory Map shown in Table 3.5. Entered are both the left and right sides of the assignment statements as names of memory locations. Rows 1, 2 and 3 identify arrays \( d, f \) and \( t \). \( d \) is a two dimensional array \([1..80,1..80]\) of 16 bit unsigned integers. Since these array addresses are calculated in the functional memory, their offset addresses will appear as constants in the FPGA program. \( t \) (row 3) is an array of 8 bit unsigned integers. Our functional memory is configured as 16 bit words, so the array takes up 80 (16 bit) words of address space, with the even (upper) bytes unused (e.g., \( t[1] \) is an eight bit unsigned integer located at byte hexadecimal address 32A1).

\( n, i, j, \text{min}, \lambda, \) "\( d[i,j] \)" and "\( f[j] \)" (rows 4, 5, 6, 8, 10, 12 and 14) are stored in the RAM but also are operands to expressions therefore must be latched in registers in one or more FPGAs. Note that "\( d[i,j] \)" and "\( f[j] \)" are names for memory locations 3350 and 3354. These locations must be loaded (i.e., updated) with their correct values whenever they, \( i \) or \( j \) change before they are used by any expression. As we see, this is the case for the first four rules because each of them modifies \( i \) or \( j \) or both.

@\textit{rule} (row 11) provides the processor with the address of the code for the rule to be executed. This function is derived after the mP instructions have been assembled and mapped into \( \mu \)instruction RAM so the starting addresses for each rule are known. @\( d[i,j], @f[j], @f[n], @f[i] \) and @\( t[i] \) are outputs of the FPGA logic providing the array element addresses of \( d[i,j], f[j], f[n], f[i] \) and \( t[i] \). \( d[i,j]+f[j], n+1, n-1, i-1, i+1 \) and \( j+1 \) are also FPGA outputs providing values based on the current values of the operands.
Table 3.5. Shortest Path Functional Memory Map

<table>
<thead>
<tr>
<th>Name</th>
<th>Address (hex)</th>
<th>Dimension</th>
<th>Operand Width</th>
<th>Functional Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>0000</td>
<td>[1..80,1..80]</td>
<td>16 bit</td>
<td>address constant</td>
</tr>
<tr>
<td>f</td>
<td>3200</td>
<td>[1..80]</td>
<td>16 bit</td>
<td>address constant</td>
</tr>
<tr>
<td>t</td>
<td>32A0</td>
<td>[1..80]</td>
<td>8 bit</td>
<td>address constant</td>
</tr>
<tr>
<td>n</td>
<td>3340</td>
<td>-</td>
<td>8 bit</td>
<td>input register</td>
</tr>
<tr>
<td>i</td>
<td>3342</td>
<td>-</td>
<td>8 bit</td>
<td>input register</td>
</tr>
<tr>
<td>j</td>
<td>3344</td>
<td>-</td>
<td>8 bit</td>
<td>input register</td>
</tr>
<tr>
<td>k</td>
<td>3346</td>
<td>-</td>
<td>8 bit</td>
<td>-</td>
</tr>
<tr>
<td>min</td>
<td>3348</td>
<td>-</td>
<td>16 bit</td>
<td>input register</td>
</tr>
<tr>
<td>ptr</td>
<td>334A</td>
<td>-</td>
<td>8 bit</td>
<td>-</td>
</tr>
<tr>
<td>lambda</td>
<td>334C</td>
<td>-</td>
<td>2 bit</td>
<td>input register</td>
</tr>
<tr>
<td>rul</td>
<td>334E</td>
<td>-</td>
<td>3 bit</td>
<td>expression register</td>
</tr>
<tr>
<td>&quot;d[i,j]&quot;</td>
<td>3350</td>
<td>-</td>
<td>16 bit</td>
<td>input register</td>
</tr>
<tr>
<td>@d[i,j]</td>
<td>3352</td>
<td>-</td>
<td>16 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>&quot;f[j]&quot;</td>
<td>3354</td>
<td>-</td>
<td>16 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@f[j]</td>
<td>3356</td>
<td>-</td>
<td>16 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@f[n]</td>
<td>3358</td>
<td>-</td>
<td>16 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@f[i]</td>
<td>335A</td>
<td>-</td>
<td>16 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@t[i]</td>
<td>335C</td>
<td>-</td>
<td>16 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>d[i,j]+f[j]</td>
<td>335E</td>
<td>-</td>
<td>16 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>n+1</td>
<td>3360</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>n-1</td>
<td>3362</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>i-1</td>
<td>3364</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>i+1</td>
<td>3366</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>j+1</td>
<td>3368</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
</tbody>
</table>

3.2.2 Shortest Path Execution Table

After the FM Map has been generated, the mP Execution Table shown in Table 3.6 can be produced. This table is used to generate the mP microcode that performs the move operations for carrying out the program assignment statements. The entry points for each of the rules are indicated in the first column of rows 1, 10, 19, 25 and 29. For each rule,
the second column begins with a list of the assignment statements, which are derived
directly from the action stubs.

**Table 3.6. Shortest Path Execution Table**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Statement</th>
<th>mP</th>
<th>Inst</th>
<th>Des Addr</th>
<th>Source</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rule1</td>
<td>f[n] := 0</td>
<td>IC</td>
<td>3358</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Rule1</td>
<td>min := maxint</td>
<td>DC</td>
<td>3348</td>
<td>0FFFF</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Rule1</td>
<td>ptr := n+1</td>
<td>DE</td>
<td>334A</td>
<td>3360</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Rule1</td>
<td>i := n-1</td>
<td>DE</td>
<td>3342</td>
<td>3362</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Rule1</td>
<td>j := i+1</td>
<td>DE</td>
<td>3344</td>
<td>3366</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Rule1</td>
<td>lambda := 1</td>
<td>DC</td>
<td>334C</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Rule2</td>
<td>UPDATE “d[i,j]”</td>
<td>DI</td>
<td>3350</td>
<td>3352</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Rule2</td>
<td>UPDATE “f[j]”</td>
<td>DI</td>
<td>3354</td>
<td>3356</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>Rule2</td>
<td>GOTO (@rule)</td>
<td>JI</td>
<td>334E</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Rule3</td>
<td>min := min</td>
<td>ID</td>
<td>335A</td>
<td>3348</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>Rule3</td>
<td>t[i] := ptr</td>
<td>ID</td>
<td>335C</td>
<td>334A</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>Rule3</td>
<td>min := maxint</td>
<td>DC</td>
<td>3348</td>
<td>0FFFF</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>Rule3</td>
<td>ptr := n+1</td>
<td>DE</td>
<td>334A</td>
<td>3360</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>Rule3</td>
<td>i := i-1</td>
<td>DE</td>
<td>3342</td>
<td>3364</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>Rule3</td>
<td>j := i+1</td>
<td>DE</td>
<td>3344</td>
<td>3366</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>Rule3</td>
<td>UPDATE “d[i,j]”</td>
<td>DI</td>
<td>3350</td>
<td>3352</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>Rule3</td>
<td>UPDATE “f[j]”</td>
<td>DI</td>
<td>3354</td>
<td>3356</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>Rule3</td>
<td>GOTO (@rule)</td>
<td>JI</td>
<td>334E</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>Rule4</td>
<td>min := d[i,j]+f[j]</td>
<td>DE</td>
<td>3348</td>
<td>335E</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>Rule4</td>
<td>ptr := j</td>
<td>DD</td>
<td>334A</td>
<td>3344</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>Rule4</td>
<td>j := j+1</td>
<td>DE</td>
<td>3344</td>
<td>3368</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>Rule4</td>
<td>UPDATE “d[i,j]”</td>
<td>DI</td>
<td>3350</td>
<td>3352</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>Rule4</td>
<td>UPDATE “f[j]”</td>
<td>DI</td>
<td>3354</td>
<td>3356</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>Rule4</td>
<td>GOTO (@rule)</td>
<td>JI</td>
<td>334E</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>25</td>
<td>Rule5</td>
<td>j := j+1</td>
<td>DE</td>
<td>3344</td>
<td>3368</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>Rule5</td>
<td>UPDATE “d[i,j]”</td>
<td>DI</td>
<td>3350</td>
<td>3352</td>
<td>3</td>
</tr>
<tr>
<td>27</td>
<td>Rule5</td>
<td>UPDATE “f[j]”</td>
<td>DI</td>
<td>3354</td>
<td>3356</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>Rule5</td>
<td>GOTO (@rule)</td>
<td>JI</td>
<td>334E</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>Rule5</td>
<td>EXIT</td>
<td>EX</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>
At the end of the action stub code, we have added the updates for "di,j" and "f[i,j]" that must be performed before they are used in the expression for computing the address of next rule to execute. At the end of each rule is the indirect jump instruction which starts the execution of the next rule, except rule 5 terminates execution with an EXIT instruction.

Columns 3, 4 and 5 of the mP Execution Table are used to generate the mP microcode. Each mP instruction expands into the microcode instructions shown in Table 2.1. The first and second operands will appear in constant fields in the microinstructions. As this expansion takes place, the starting addresses for each rule becomes known. This information is reflected in the Rule Address Map shown in Table 3.7. With the rule starting addresses known, the boolean expressions for computing the @rule address can be determined. For the nine binary address bits shown in Table 3.7, there will be nine boolean equations. For each bit, we know when it should be a '1' value, which depends on which rule is to fire, which depends on the condition stub values. The expressions for determining the rule address will be shown in the next section.

Table 3.7. Shortest Path Rule Address Map

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Start Address (Hexadecimal)</th>
<th>Start Address (Binary)</th>
<th>Execution Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule1</td>
<td>004</td>
<td>0000 0000 0100</td>
<td>20</td>
</tr>
<tr>
<td>Rule2</td>
<td>054</td>
<td>0000 0101 0100</td>
<td>22</td>
</tr>
<tr>
<td>Rule3</td>
<td>0AC</td>
<td>0000 1010 1100</td>
<td>14</td>
</tr>
<tr>
<td>Rule4</td>
<td>0E4</td>
<td>0000 1110 0100</td>
<td>10</td>
</tr>
<tr>
<td>Rule5</td>
<td>10C</td>
<td>0001 0000 1100</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.7 also shows the number of cycles each rule takes to execute. Each cycle takes exactly one clock period therefore, if it is known how many times each rule fires,
then the exact time it takes for the program to execute can be determined. These figures will be used in Chapter 5 for computing the execution time of the program.

### 3.2.3 Shortest Path Expression Graphs

The expression graphs serve as an intermediate text when hand-generating the PALASM equations. The condition stub expressions evaluate to true/false results and are used to generate the address of the rule to be executed. As Figure 3.9 illustrates, these expressions require variables \( \lambda, i, j, n, \) "\( d[i,j] \)”, “\( f[j] \)” and \( \text{min} \). The boxes on top represent these as input registers. The FM address is also indicated above each. At the bottom of each expression is shown a box which represents a combinational input to the chip's output multiplexer. Below each box identifies the FM address for reading the expression result. Also, below the addresses of the output boxes is shown in which FPGA chip the expression is implemented.

In this example we will also show unit numbers assigned to each operator node. As indicated in the figure, nodes U2, U3, U9 and U10 calculate the condition stubs. To store in "\( d[i,j] \)” and “\( f[j] \)” the array addresses of these elements (which depend on \( i \) and \( j \)) must be calculated. Nodes U4 through U8 calculate the address of \( d[i,j] \) while nodes U11 and U12 calculate the address of \( f[j] \). Note that \( j-1 \) must be calculated for computing \( @f[j] \) in SPC1, and separately for \( @d[i,j] \) in SPC2. The address of the rule microcode (\( @\text{rule} \)) is calculated by U1.

The remaining expressions are shown in Figure 3.10. Once “\( d[i,j] \)” and “\( f[j] \)” (locations 3350 and 3354) have valid values, location 335E will contain the \( d[i,j]+f[j] \) sum. Whenever \( n \) changes (i.e., location 3340 is written), locations 3360 and 3362 will contain the results of expressions \( n+1 \) and \( n-1 \). Whenever \( i \) changes, \( i+1 \) and \( i-1 \) immediately and simultaneously become available. Whenever 3344 (\( j \)) is written, 3368 (\( j+1 \)) immediately contains the value stored in 3344 incremented by 1. The expressions
for the addresses of destination array elements $f[i], f[i]$ and $q[i]$ are also shown in Figure 3.10.

Figure 3.9. Shortest Path Rule Address Computation
Notice in Figure 3.9 that for $d[i,j]$, $i$ must be decremented and multiplied by the size of the first dimension (which in our case is 80). This expression is by far the most complex in our example because it contains an 8 bit by 8 bit input–16 bit output multiplier, U5. The expression consumes more than half of one chip in our implementation (SPC2). This, however, may be where the most improvement for this example is realized over conventional processor implementations.

![Figure 3.10. Shortest Path Expression Computation](image)

### 3.2.4 Shortest Path FPGA Code Generation

The next step in implementing the shortest path program is use the expression graphs to generate the "EQUATIONS" section of the PALASM source code file for each FM FPGA chip (see Figure 2.3). As we have seen, the EQUATIONS section of each chip contains five subsections: (1) the address select logic, (2) the input registers, (3) the rule address generation logic, (4) the expression logic and (5) the output multiplexer logic.
The address select logic includes the clock enables for the input registers and the unit node output enables for those which feed the output multiplexers. Every scalar variable used in an expression, including condition stubs, must be captured in an input register. The rule address generation logic begins with the evaluation of the condition stubs, followed by the generation of the rule address bits. The microcode starting addresses must be known before these equations can be generated. Figure 3.6 of Section 3.1.4 shows an example of the translation of a j+1 expression graph, denoted by U20 in Figure 3.10, into the appropriate PALASM for computing the expression.

### 3.2.5 Shortest Path FPGA Compilation Statistics

After each FPGA PALASM source program is built, it must be linked with its circuit interface shell (see Figure 2.3), then “placed” and “routed” separately. Figure 3.7 illustrates the complete process. Placement and routing is by far the longest step in the compilation process using presently available design tools. Placement involves binding the gates and latches as configurable logic blocks (CLBs) to physical CLBs in the chip. This step is difficult because if done poorly, not all the interconnections may be routable. As Table 3.8 shows, the expressions assigned to the first FPGA (SPC1) consumed 696 gates and 90 latches in 182 CLBs, taking 10 hours and 44 seconds to place. There were 316 nets taking 1 hour and 22 minutes to route. SPC2 used 953 combinational gates and 40 latches in 196 CLBs taking 11 hours and 54 minutes to place. 353 different signal nets took 1 hour and 13 minutes to route. SPC3 held 784 gates and 48 latches in 160 CLBs taking 10 hours and 29 minutes to place, with 261 nets taking only 16 minutes to route.

83
Table 3.8 also shows the maximum number of levels of combinational logic in each chip. Notice that SPC2, which implements the multiplier for calculating $d[i,j]$ used the most gates and took the most number of levels.

When all three FPGAs have been placed and routed, a "BIT" file is produced for each. The .BIT files are linked with the .BIT file for the mP FPGA specification into one load module. With the mP FPGA wired for peripheral program load mode on the system bus and the other four FPGAs daisy-chained off of it, the system processor can load the FPGA object module (containing the five FPGA programs), byte by byte similar to loading a conventional program into ordinary memory, except all the bytes are written to the same location (see [XILINX, 1993] for details).

<table>
<thead>
<tr>
<th>FPGA</th>
<th># Gates</th>
<th># Latches</th>
<th>Max Level</th>
<th># CLBs</th>
<th># Nets</th>
<th>Placement Time (hh:mm:ss)</th>
<th>Routing Time (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPC1</td>
<td>696</td>
<td>90</td>
<td>25</td>
<td>182</td>
<td>316</td>
<td>10:00:44</td>
<td>1:22:47</td>
</tr>
<tr>
<td>SPC2</td>
<td>953</td>
<td>40</td>
<td>37</td>
<td>196</td>
<td>353</td>
<td>11:54:46</td>
<td>1:13:06</td>
</tr>
<tr>
<td>SPC3</td>
<td>784</td>
<td>48</td>
<td>16</td>
<td>160</td>
<td>261</td>
<td>10:29:03</td>
<td>0:16:44</td>
</tr>
</tbody>
</table>

3.3 CONCLUSIONS

In this chapter we introduced the process that must take place for translating high level language programs into the minimal processor microcode and FPGA PALASM nanocode. The subject of compiling and executing high level language programs for our FMC will be taken up again in Chapter 6, where we will describe our compiler for producing the two load modules automatically from a decision table source program.

In this chapter we investigated two approaches for handling the problem of array elements in functional memory expressions. For the bubble sort example, it was to the
programmer's advantage to explicitly allocate the temporary scalar variables because execution cycles were saved by implementing a non-standard exchange operation, which took advantage of the fact that the elements to be exchanged were already stored in temporary locations.

In the shortest path example, there was no advantage for the programmer having to explicitly manipulate the temporary scalars, so we would choose for it to be a "compiler problem," to be handled automatically and hidden from the user. The compiler would need to know when to allocate a new temporary scalar, and when to insert code to update it.

In the next chapter, we investigate the possibilities of actually storing arrays in FPGA registers. When FPGA expressions can use array elements as operands directly, we no longer have the problem of allocating and updating temporary scalars. More interestingly, however, we can actually implement nondeterministic algorithms, and give a nondeterministic bubble sort program as an example. In Chapter 5, we will compare the execution profiles of the deterministic bubble sort described in this chapter with the nondeterministic version to be described in Chapter 4 and find they are much different.
CHAPTER 4. IMPLEMENTING NONDETERMINISTIC PROGRAMS ON A FUNCTIONAL MEMORY COMPUTER

When the gate level implementations of functions can be determined at compile time instead of fixed at machine design time, then the amount of logic used to compute the function can vary according to compile time program parameters. In combinational logic systems, added logic can increase functionality without adding program execution cycles. Programs “in the large” conceptually involve “state changes.” Adding combinational logic to help compute larger functions increase what is accomplished within a state change.

In functional memory systems, operands are written into registers which in turn feed combinational logic within the FPGA, which in turn cause outputs to change. It is important that the propagation delay through the combinational logic is tolerable. The amount of time after an operand is written, to when an output will need to be read, can be determined by the compiler. Rarely (if ever) will an expression result be needed in the cycle immediately following when one of its operands is written. Usually it is several instructions later. In our 4 MHz system, a cycle is 250 nS and no instruction is less than 500 nS. Combinational logic gate delays for today’s technology is on the order of tens of nanoseconds. While there are other factors which add to the propagation delay of a combinational logic function, much room exists for adding complexity to single expression computations.

This chapter begins by proposing some nondeterministic operators that can be implemented in functional memory and therefore such constructs can be added to the FMC programming language. Nondeterministic set operators are the most simple because little logic is consumed when storing and operating on set operands in the
FPGAs. We show a minimum array element selection operation can be built-up across several FPGAs if a word-sized daisy-chain connection between chips is available.

In the second section of this chapter we implement a nondeterministic bubble sort program that uses a set selection operation which follows a set of less-than comparators. The less-than comparators connect one to each adjacent array element pair indicating if the pair of elements is out of order. This example demonstrates the power of functional memory systems when there is enough capacity to implement entire arrays in FPGA registers. In our 50 element array sorting example, temporary scalar variables are not needed so 49 comparisons can be performed simultaneously, along with the selection of one of the out-of-order pairs for exchange.

4.1 NONDETERMINISTIC CONSTRUCTS

When functions are written to examine all the members of a set or all the elements of an array simultaneously, we are able to exploit the signal level parallelism of combinational logic. Parallelism in combinational logic is asynchronous in that no order is imposed on the completion of intermediate states hence nondeterministic functions are a natural application. Nondeterministic set selection functions are easily implemented because the component functions for their solution are already traditional building blocks in logic design. When these operations are made available to the programmer, the programmer can implement nondeterministic algorithms. In this section we describe how set and minimum selection operators can be implemented on a functional memory computer.

4.1.1 Nondeterministic Set Operators

Figure 4.1 shows the block diagram implementation of three set functions. Set bit vector inputs may be input registers or outputs from other combinational logic functions
(as in the case with the nondeterministic bubble sort example used in Section 4.2). With one bit for each possible set member, selecting a member, testing for membership and removing or adding an element to the set can be accomplished within one cycle. Note that in the case of adding or removing an element, the new set must still be copied to a destination location. Functions for sets of size over 100 can be implemented in a single FPGA and cascading FPGAs can be accomplished with a single pin daisy-chain.

On conventional computers, sets are implemented either as bits of a vector (which can be one or more words) or as linked lists. For bits in a vector, selecting an element from a set would take at most $O(n)$ for an $n$ element set because element positions would have to be examined one by one and the last element to be examined may be the only element in the set. With linked lists, the function would take $O(1)$. As Figure 4.1A depicts, when implemented in functional memory, every bit position of the set is fed through a selector circuit which allows only one ‘1’ value (indicating that member is present in the set) to pass. That single ‘1’ is input to an encoder which outputs the encoded index position of that specific ‘1’.

When testing for set membership, $n$ bits in a vector representation would take $O(1)$ but the linked list method would take $O(n)$ in the worst case. As shown in Figure 4.1B,
in functional memory, the set vector is fed into a multiplexer and the positional index of
the possible member in question is placed on the select lines. The multiplexer simply
routes the bit value of that position to the output, which will be a ‘1’ if the member is
present and a ‘0’ if not. As we have seen previously, implementations of multiplexers are
very straightforward in gate level logic.

Adding and removing elements can be accomplished in $O(1)$ in most implementations
on conventional machines. Figure 4.1C shows that in functional memory, an “invert bit”
function must be implemented in which case the set input vector would pass through with
only the selected bit inverted. This could be implemented with each input bit feeding one
input of an exclusive-OR gate, with the other exclusive-OR input being fed by the output
of a decoder/demultiplexer circuit. The index selects which particular decoder output is to
be set to a ‘1’, which causes that respective exclusive-OR gate to invert its set input vector
bit. (If it were unknown whether the element being added or removed was initially in the
set, logical-OR would be used for set addition and logical-AND would be used for
removal.)

Selecting a set element is “nondeterministic” in that if more than one member is
present, which one is selected is not specified. Any one will do. When sets are so large
that they require a set bit vector that spans more than one FPGA chip, some interchip
communication is necessary. With each chip managing only one subset, the solution
requires circuits to know whether members exist in one or more other subsets. This
particular problem can be solved very easily with a single inter-FPGA daisy-chain signal
and will be shown in the next section. Set operators for testing membership, addition or
removal can be spread across several FPGA chips by the compiler without any processor
initiated interchip communication.
4.1.2 Minimum Selection

A more difficult nondeterministic function to implement in combinational logic is selecting a minimum element from an array. In this case, the entire array must be stored in FPGA registers, and elements must be compared with each other. The result of each comparison must feed a multiplexer which routes the minimum value of the two to a “next stage” multiplexer. The results of each comparison must also feed logic which encodes the minimum (“winning”) index. Figure 4.2 in the upper left shows a basic two-input building block circuit for determining the index of and minimum of two values. This function can be combined in a binary tree like fashion (shown on the right) to build a circuit which will determine the index of and minimum of several values.

![Diagram of a basic two-input block circuit](image)

**Figure 4.2. O(1) Minimum Array Element Selection**

When the number of values from which a minimum is to be selected will not all fit in one FPGA, then a more sophisticated interchip communication is necessary than in the set selection case. As Figure 4.2 illustrates, each chip is able to determine if it contains a
value smaller than any chip to its left, and if so, it signals all chips to its left as indicated by the “found out” daisy chain. Each chip passes the minimum value to the right through the “value out” daisy chain. If a chip to the right contains a smaller value, the “found in” daisy chain will indicate as such. The “value in” and “value out” daisy chains must be as wide as elements of the array (e.g., 16 bits). The “found in” and “found out” daisy chains can be one bit, as in the case for set selection above.

4.2 Nondeterministic Bubble Sort (NBS)

A bubble sort program from [Lew and Halverson, 1994] will be used to demonstrate the potential advantage of nondeterminism. Bubble sort was chosen because it is extremely simple yet complex enough to show how looping and arrays are handled. The single jump address calculation is simple yet nontrivial, because the program requires a double nested loop. A second nondeterministic bubble sort program is used to demonstrate the dramatic performance potential of nondeterminism. This example also illustrates the implementation of a simple inter-FPGA communication interconnect because every array element must be examined simultaneously and the entire array cannot be contained in a single FPGA.

The bubble sort algorithm sorts an array by exchanging out-of-order adjacent pairs, one pair at time, until none remains. Deterministically, the program must examine \( n(n-1)/2 \) pairs for an \( n \) size array regardless of how sorted or unsorted the array is to begin with. Nondeterministically, the program chooses (i.e., computes in a single step) indexes of adjacent out-of-order pairs, exchanges their contents, and stops when no out-of-order pair remains. Hence the nondeterministic program “examines” only as many pairs as need to be exchanged. Therefore, an array which initially is randomly ordered would take only about half as long to sort as a worst case array. We see a dramatic
difference when sorting an array which is initially ordered, which the nondeterministic program would sort in negligible time regardless of $n$ (see Section 5.2.3), compared to $O(n)$ at best for the deterministic version.

To clarify these ideas, we show in Figures 4.3 and 4.4 a nondeterministic bubble sort which selects an adjacent out-of-order pair for exchange (if one exists) in one operation where the order in which adjacent pairs are examined or selected is not specified in the program (hence is nondeterministic). As long as an out of order adjacent pair exists to be chosen, the process proceeds and pairs are exchanged. The process continues until no more out of order pairs exist.

A. "Pseudo Pascal"

```pascal
const
  N = 50;
var A : array [1 .. N] of byte;
procedure NonDetBubSort
  var j : byte;
  begin
    while {j in 1 .. N-1 | A[j] > A[j+1]} #\emptyset do
      exchange(A[j], A[j+1]);
  end;
```

B. Decision Table

<table>
<thead>
<tr>
<th>Rule:</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>{j in 1 .. N-1</td>
<td>A[j] &gt; A[j+1]} #\emptyset</td>
<td>F</td>
</tr>
<tr>
<td>exchange(A[j], A[j+1])</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>exit</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>

**Figure 4.3.** Nondeterministic Bubble Sort Program

**Figure 4.4.** Nondeterministic Bubble Sort Flow Chart
In our implementation of this, we loop continuously, first finding the set of indices that are out of order. Then we use the set select function to choose an index \( j \), if one exists. From \( j \), the array addresses of the element and its neighbor to the right, at index \( j+1 \) can be located. These three operations are all performed in combinational logic, as shown in Figure 4.5. The array elements at these two computed addresses are then exchanged by the processor.

![Figure 4.5. NBS Exchange Address and Rule Selection](image_url)

Because of limitations on our design tools, up to 14 array elements fit in each FPGA. Comparisons for \( a[1] \) through \( a[14] \) were performed in FPGA #1, \( a[14] \) through \( a[27] \) in FPGA #2, \( a[27] \) through \( a[39] \) in FPGA #3 and comparisons for \( a[39] \) through \( a[50] \) took place in FPGA #4.
The processor must know if any \( j \) exists, and if so, one must be selected. As the figure shows, a chip can only know if any out of order adjacent array elements exist in its own portion of the array. The select/encode logic within the first chip can tell if a \( j \) exists within itself. If one does not exist, this is passed down through a single signal daisy-chain to the second chip. If a \( j \) does not exist in the first or second chips, the second chip passes this down to the third chip. If none exists in the first three chips, this is passed on to the fourth. Whether or not a \( j \) exists determines which rule to execute, whose address is generated by the fourth chip.

When the processor wishes to read the addresses of the chosen \( j \) and \( j+1 \) elements, each chip also uses the daisy-chain to know if it is the leftmost chip containing an out of order adjacent pair. This is to ensure that only one chip gates a \( j \) or \( j+1 \) address on to the data bus when the processor wants to read either of these locations.

### 4.2.1 NBS Functional Memory Map

The first step in translating the nondeterministic bubble sort program of Figure 4.3B to execute on a FMC is to generate the Functional Memory Map shown in Table 4.1. Entered are both the left and right sides of the assignment statements as names of memory locations. Following the array \( a \) allocation, \texttt{@rule} at 0100 (hex) is the expression output providing the microprogram address for the code of which rule to execute. (For convenience, except for the array values themselves which start at 0000 (hex), we chose to begin assigning functional memory locations at 0100 hex.) Rows 3 and 4 contain the array element addresses for \( a[j] \) and \( a[j+1] \). \texttt{temp} and \texttt{temp@} in rows 5 and 6 are temporary locations necessary for the exchange operation.
4.2.2 NBS Execution Table

All the information that must be used to generate the nondeterministic bubble sort microcode is contained in the Execution Table shown in Table 4.2. As with Table 3.2 (for the deterministic bubble sort), each row represents one minimal processor instruction. The first column identifies the starting location of the code for each rule. Column 2 contains the action stubs collected in order for each rule. The minimal processor instructions are listed in column 3. Operands are listed in columns 4 and 5. Column 6 lists the number of cycles each instruction takes to execute. These figures will be used when comparing the execution times between the two bubble sorts in Chapter 5.

**Table 4.1. Nondeterministic Bubble Sort Functional Memory Map**

<table>
<thead>
<tr>
<th>Name</th>
<th>Address (hex)</th>
<th>Dimension</th>
<th>Operand Width</th>
<th>Functional Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0000</td>
<td>[1..50]</td>
<td>8 bit</td>
<td>address constant</td>
</tr>
<tr>
<td>@rule</td>
<td>0100</td>
<td>-</td>
<td>5 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@a[j]</td>
<td>0102</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>@a[j+1]</td>
<td>0104</td>
<td>-</td>
<td>8 bit</td>
<td>expression output</td>
</tr>
<tr>
<td>temp</td>
<td>0106</td>
<td>-</td>
<td>8 bit</td>
<td></td>
</tr>
<tr>
<td>temp@</td>
<td>0108</td>
<td>-</td>
<td>8 bit</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.2. Nondeterministic Bubble Sort Execution Table**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Statement</th>
<th>mP Inst</th>
<th>Des Addr</th>
<th>Source</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule0</td>
<td>goto @rule</td>
<td>JI</td>
<td>0100</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Rule1</td>
<td>EXIT</td>
<td>EX</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Rule2</td>
<td>temp := a[j]</td>
<td>DI</td>
<td>0106</td>
<td>0102</td>
<td>2*</td>
</tr>
<tr>
<td></td>
<td>temp@ := @a[j+1]</td>
<td>DD</td>
<td>0108</td>
<td>0104</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>a[j] := a[j+1]</td>
<td>II</td>
<td>0102</td>
<td>0104</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(temp@) := temp</td>
<td>ID</td>
<td>0108</td>
<td>0106</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>goto @rule</td>
<td>JI</td>
<td>0100</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

* - optimized
From Table 4.2, we see that the exchange operation is implemented by first moving \(a[j+1]\) into \(a[j]\) and then \(a[j]\) into \(a[j+1]\). To implement this, the processor must use a temporary location \(temp\), (location 0106), for temporarily storing the original contents of \(a[j]\) because it is destroyed when \(a[j+1]\) is stored into \(a[j]\). This is just as it would be in a von Neumann implementation. An additional temporary address location, \(temp@\), however, is also needed to store the original \@a[j+1]\) because as soon as a new value is written into \(a[j]\), then \(a[j]>a[j+1]\) immediately changes (in fact indicating they are equal), which immediately changes \(j\), which immediately changes \@a[j+1]\) before it can be used to store in the old value of \(a[j]\). We see in the second to last row in the table, that \(temp\), which holds the original \(a[j]\) value, is stored into the location pointed to by \(temp@\), which is the address of the original \(a[j+1]\).

The first and last processor instruction in Table 4.2 is a “goto” to the location specified in @rule. Normally because of pipelining, it takes one extra cycle on our FMC for the \(\mu\)program counter to be loaded with a new \(\mu\)program address when instructed to do so, therefore, the second \(\mu\)instruction following the one which actually initiates the load is always executed before the jump actually takes place. In most cases (as in the deterministic version), since the destination \(\mu\)program address is not known at compile time, then the second \(\mu\)instruction of the JJ instruction is usually a NOP (no-operation). In this case, however, since the next \(\mu\)instruction following the JJ is always the first \(\mu\)instruction of the DI instruction for performing \(temp:=a[j]\), this \(\mu\)instruction can be placed as the second \(\mu\)instruction of the JJ instruction. This effectively reduces the JJ in this case to a single \(\mu\)instruction operation because the second \(\mu\)instruction of the JJ is actually that which was the first DI \(\mu\)instruction. This eliminates it from the cycle count for the DI instruction on row 3. Since the first \(\mu\)instruction of the DI does not actually
store any data into the RAM, it will not cause any problems on the last iteration when Rule 1 is executed and the program terminates.

4.2.3 NBS FPGA Compilation Statistics

Table 4.3 shows the FPGA compilation statistics for the nondeterministic bubble sort. The implementation required a total of 2344 gates (2, 3, 4 or 5 input), 424 latches (to hold the operands when written to the FPGAs, for a total of 630 CLBs. The table also shows that the 630 CLBs used 4,064 pins and were connected using 1,002 nets (inside the four FPGAs) which took over 26 hours to place and nearly 3 hours to route.

<table>
<thead>
<tr>
<th>FPGA</th>
<th># Gates</th>
<th># Latches</th>
<th>Max Level</th>
<th># CLBs</th>
<th># Pins</th>
<th># Nets</th>
<th>Placement Time (hh:mm:ss)</th>
<th>Routing Time (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBSCI</td>
<td>607</td>
<td>112</td>
<td>19</td>
<td>161</td>
<td>1052</td>
<td>258</td>
<td>7:14:56</td>
<td>0:50:32</td>
</tr>
<tr>
<td>NBSC2</td>
<td>616</td>
<td>112</td>
<td>19</td>
<td>166</td>
<td>1079</td>
<td>262</td>
<td>7:01:11</td>
<td>1:43:26</td>
</tr>
<tr>
<td>NBSC3</td>
<td>580</td>
<td>104</td>
<td>20</td>
<td>159</td>
<td>1036</td>
<td>249</td>
<td>6:51:45</td>
<td>0:11:57</td>
</tr>
<tr>
<td>NBSC4</td>
<td>541</td>
<td>96</td>
<td>20</td>
<td>144</td>
<td>897</td>
<td>233</td>
<td>5:21:03</td>
<td>0:05:09</td>
</tr>
<tr>
<td>Total</td>
<td>2344</td>
<td>424</td>
<td>=30*</td>
<td>630</td>
<td>4064</td>
<td>1002</td>
<td>26:28:55</td>
<td>2:51:04</td>
</tr>
</tbody>
</table>

* estimated with interchip daisy-chain

4.3 CONCLUSIONS

This chapter discussed how some basic nondeterministic operations could be implemented in functional memory with the intent that they eventually be incorporated into the FMC high level programming language. Nondeterministic set operations appear straightforward to implement using functional memory. Nondeterministic algorithms based on sets rather than sequential arrays are especially promising.

To demonstrate the potential of nondeterminism, a nondeterministic bubble sort was implemented that chose out-of-order array elements to be exchanged in $O(1)$ with respect
to \( n \) size of the array. We observed that the size of the array was limited to about 50 elements on our machine. In the next chapter we shall see how nondeterminism can affect the sorting execution times. Future research objectives will include the design of more complex nondeterministic algorithms, such as a divide-and-conquer set selection operation and nondeterministic dynamic programming for finding shortest paths.
CHAPTER 5. ANALYSES OF EXECUTION

In this chapter we will calculate and analyze the shortest path and bubble sort programs we described in the earlier chapters. In the first section we will compare measured execution times for both programs executing on a conventional von Neumann computer (using an Intel 486/SX 25 MHz processor) and our 4 MHz FMC. In the second section we will derive and measure the cycle counts for the two programs for a much fairer comparison between the two machines. The last section concludes with a load/store analysis comparing a shortest path decision table program executing on a FMC to a Pascal version executing on a von Neumann computer (i486).

Measurements were made varying the array \( n \) size. Programs were designed so the array set-up and verification times were constant regardless of \( n \). Programs were executed 1000 times at the varying \( n \) sizes and their times recorded. The program was run with \( n=1 \), which gave just the load and verification times for 1000 executions. This value was then subtracted from the subsequent measurements to obtain an accurate execution time for the program itself, without the array load and verification times.

5.1 TIME COMPARISONS

In this section we measure the execution times of shortest path and bubble sort programs executing on the i486 and on our prototype FMC. Time measurements were made on the i486 using the system clock. Measurements on the FMC were made using a time-of-day circuit (Dallas DS1216) accurate to one-hundredth of a second.
5.1.1 Shortest Path Execution Times

Table 5.1 and Figure 5.1 compare the measured execution times in milliseconds for the shortest path program implemented on our 4 MHz FMC with the same program implemented in "C" on the i486.

Table 5.1. Shortest Path Execution Times

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMC 4 MHz-Worst Case</td>
<td>0.21</td>
<td>0.78</td>
<td>1.68</td>
<td>2.95</td>
<td>4.56</td>
<td>6.52</td>
<td>8.83</td>
<td>11.50</td>
</tr>
<tr>
<td>FMC 4 MHz-Best Case</td>
<td>0.17</td>
<td>0.60</td>
<td>1.28</td>
<td>2.20</td>
<td>3.39</td>
<td>4.81</td>
<td>6.49</td>
<td>8.41</td>
</tr>
<tr>
<td>i486 25 MHz-Worst Case</td>
<td>0.11</td>
<td>0.44</td>
<td>1.04</td>
<td>1.87</td>
<td>2.96</td>
<td>4.28</td>
<td>5.82</td>
<td>7.63</td>
</tr>
<tr>
<td>i486 25 MHz-Best Case</td>
<td>0.11</td>
<td>0.44</td>
<td>1.04</td>
<td>1.81</td>
<td>2.85</td>
<td>4.12</td>
<td>5.65</td>
<td>7.41</td>
</tr>
</tbody>
</table>

From Table 5.1, we see that with the worst case distance array for 10 nodes (to visit), the FMC shortest path program took .21 mS whereas it took only .11 mS for the i486. The best case array of 10 nodes took .17 mS to complete on the FMC and also .11 mS on the i486. For \( n = 80 \), the FMC took 11.5 mS while the i486 took only 7.63 mS in the
worst case. In the best case with $n=80$, the FMC took 8.41 mS while the i486 took 7.41 mS.

As Figure 5.1 illustrates, our 4 MHz FMC does not perform quite as well as the i486. We also see a greater difference between the worst and best case arrays on the FMC than on the i486 which suggests that the assignment of variables $min$ and $ptr$ in Rule 3 consume a greater percentage of the execution time on the FMC than on the conventional i486 microprocessor implementation. On the i486, the greatest percentage of execution time is likely the $d[i,j]$ array address calculation which must be performed in both best and worst cases, which is why the two execution curves are nearly identical.

### 5.1.2 Deterministic Bubble Sort Times

Table 5.2 and Figure 5.2 show the execution times for a bubble sort program written in C and executed on the i486. We will see that the times are superior to the deterministic FMC version and the worst case times are also superior to the nondeterministic FMC version.

Table 5.3 and Figure 5.3 show the observed performance measurements for the deterministic bubble sort program executing on the FMC. Array load and sort verification times were constant regardless of values or $n$-size. For each data point, the program was executed 1,000 times, three times in a row to obtain a “winning” value in seconds and hundredths of seconds. The time to load and verify the array at $n=1$ (1.67 seconds for 1000 iterations) was subtracted from all data points above $n=1$ and reported in Table 5.3 and shown graphically in Figure 5.3.
Table 5.2. Bubble Sort i486 Execution Times

<table>
<thead>
<tr>
<th>(mS) n</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td>0.00</td>
<td>0.05</td>
<td>0.16</td>
<td>0.33</td>
<td>0.54</td>
<td>0.82</td>
<td>1.15</td>
<td>1.53</td>
<td>2.03</td>
<td>2.52</td>
<td>3.13</td>
</tr>
<tr>
<td>Random</td>
<td>0.00</td>
<td>0.05</td>
<td>0.11</td>
<td>0.22</td>
<td>0.38</td>
<td>0.60</td>
<td>0.87</td>
<td>1.15</td>
<td>1.48</td>
<td>1.86</td>
<td>2.31</td>
</tr>
<tr>
<td>Best</td>
<td>0.00</td>
<td>0.05</td>
<td>0.06</td>
<td>0.16</td>
<td>0.27</td>
<td>0.38</td>
<td>0.55</td>
<td>0.71</td>
<td>0.93</td>
<td>1.15</td>
<td>1.43</td>
</tr>
</tbody>
</table>

Figure 5.2. Bubble Sort i486 Execution Times

The difference between the three curves in Figure 5.3 reflect the difference in the number of times Rule 3 executed versus Rule 4. In the worst case, Rule 3 executed \( n(n-1)/2 \) times and Rule 4 never. In the best case, Rule 3 is never executed and Rule 4 is executed \( n(n-1)/2 \) times. Since Rule 3 consumes 6 more cycles than Rule 4, this computes to a \( 6n(n-1)/2 \) cycle difference, or 7,350 when \( n=50 \). Dividing by 4 MHz gives 1.84 mS, which was the observed difference between the two curves for \( n=50 \).
Table 5.3. Deterministic Bubble Sort FMC Execution Times

<table>
<thead>
<tr>
<th>(mS) n</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td>0.00</td>
<td>0.05</td>
<td>0.21</td>
<td>0.46</td>
<td>0.82</td>
<td>1.27</td>
<td>1.83</td>
<td>2.49</td>
<td>3.24</td>
<td>4.09</td>
<td>5.05</td>
</tr>
<tr>
<td>Random</td>
<td>0.00</td>
<td>0.05</td>
<td>0.18</td>
<td>0.41</td>
<td>0.70</td>
<td>1.06</td>
<td>1.53</td>
<td>2.06</td>
<td>2.67</td>
<td>3.33</td>
<td>4.15</td>
</tr>
<tr>
<td>Best</td>
<td>0.00</td>
<td>0.04</td>
<td>0.14</td>
<td>0.31</td>
<td>0.54</td>
<td>0.82</td>
<td>1.18</td>
<td>1.59</td>
<td>2.07</td>
<td>2.61</td>
<td>3.21</td>
</tr>
</tbody>
</table>

Figure 5.3. Deterministic Bubble Sort FMC Execution Times

5.1.3 Nondeterministic Bubble Sort Times

Table 5.4 and Figure 5.4 show the actual measured execution times for the nondeterministic bubble sort program. The difference between the three curves in Figure 5.4 reflect the difference in the number of times Rule 2 executes. In the best case, Rule 2 never executes because no out of order pairs exist, therefore the curve is flat at zero (to the nearest .01 mS) regardless of n. In the worst case, Rule 2 must exchange \( n(n-1)/2 \) pairs, taking nearly 4 mS when \( n=50 \).
Table 5.4. Nondeterministic Bubble Sort FMC Execution Times

<table>
<thead>
<tr>
<th>(mS) n =</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td>0.00</td>
<td>0.04</td>
<td>0.15</td>
<td>0.34</td>
<td>0.62</td>
<td>0.97</td>
<td>1.42</td>
<td>1.93</td>
<td>2.54</td>
<td>3.22</td>
<td>3.98</td>
</tr>
<tr>
<td>Random</td>
<td>0.00</td>
<td>0.02</td>
<td>0.08</td>
<td>0.22</td>
<td>0.35</td>
<td>0.51</td>
<td>0.75</td>
<td>1.00</td>
<td>1.31</td>
<td>1.55</td>
<td>2.04</td>
</tr>
<tr>
<td>Best</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Figure 5.4. Nondeterministic Bubble Sort FMC Execution Times

5.2 CYCLE COUNT COMPARISONS

In this section, the cycle counts for each program running on the different machines is calculated and compared. This provides a fairer comparison between the 25 MHz i486 and the 4 MHz FMC because it assumes both implementations will use relatively the same silicon technology. Our FMC uses the slowest rated FPGA parts (~50s) because these were all that were available in our laboratory. Using parts rated three times faster (e.g., ~150s) would likely allow us to triple our oscillator frequency to 12 MHz. For the FMC programs, the cycle counts are determined from equations derived by analyzing the
decision table. Computed values are then compared with observed measurements by multiplying the execution times by the megahertz of the respective processor.

5.2.1 Shortest Path Cycle Counts

Analyzing the shortest path decision table algorithm given in Figure 3.8, we see that Rules 1 and 5 execute once, Rule 2 executes for \( i \) from \( n-1 \) down to 1 and Rules 3 and 4 together execute a total of \( n(n-1)/2 \) times, depending on how close the distance array \((d)\) is to the best or worst case. In the worst case, \( d[i,j] + f[i] \) is always less than \( \text{min} \) so Rule 3 executes every iteration:

\[
C_{\text{Worst}} = C_{\text{Rule 1}} + (n-1)C_{\text{Rule 2}} + \frac{n(n-1)}{2}C_{\text{Rule 3}} + C_{\text{Rule 5}}
\]

In the best case, however, Rule 3 executes only when \( j=1 \) and Rule 4 executes the rest of the time, therefore:

\[
C_{\text{Best}} = C_{\text{Rule 1}} + (n-1)C_{\text{Rule 2}} + (n-1)C_{\text{Rule 3}} + \frac{(n-1)(n-2)}{2}C_{\text{Rule 4}} + C_{\text{Rule 5}}
\]

For example, we can compute exactly how long a 4 MHz FMC will take to execute the program when \( n=80 \) in the worst case. Plugging in the cycle counts for each rule from Table 3.7 into \( C_{\text{Worst}} \), we see the program takes \( (20 + 79\cdot22 + 80\cdot79/2\cdot14 + 2) = 46,000 \) cycles to complete. At 4,000,000 Hertz (cycles/second), this divides out to 11.5 mS to execute. For the best case, \( C_{\text{Best}} = 33,676 \) for a time of 8.42 mS. This analysis verifies the measured times in Table 5.1.

Table 5.5 and Figure 5.5 show the cycle count comparison (in millions) between the two shortest path implementations. From the table we see that with the worst case array of \( n=10 \) nodes, the FMC program took 800 cycles whereas it took 2,800 cycles for the i486. The best case array of 10 nodes took only 700 to complete on the FMC and also 2,800 on the i486. For \( n=80 \), the FMC took 46,000 cycles while the i486 took 190,800
in the worst case. In the best case with \( n=80 \), the FMC took 33,000 cycles while the i486 took 185,000. These figures were computed by multiplying the execution times in Table 5.1 by the megahertz of the respective processor.

Table 5.5. Shortest Path Cycle Counts

<table>
<thead>
<tr>
<th>('000s)</th>
<th>( n )</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMC-Worst Case</td>
<td>0.8</td>
<td>3.1</td>
<td>6.7</td>
<td>11.8</td>
<td>18.2</td>
<td>26.1</td>
<td>35.3</td>
<td>46.0</td>
<td></td>
</tr>
<tr>
<td>FMC-Best Case</td>
<td>0.7</td>
<td>2.4</td>
<td>5.1</td>
<td>8.8</td>
<td>13.6</td>
<td>19.2</td>
<td>26.0</td>
<td>33.6</td>
<td></td>
</tr>
<tr>
<td>i486-Worst Case</td>
<td>2.8</td>
<td>11.0</td>
<td>26.0</td>
<td>46.8</td>
<td>74.0</td>
<td>107.0</td>
<td>145.5</td>
<td>190.8</td>
<td></td>
</tr>
<tr>
<td>i486-Best Case</td>
<td>2.8</td>
<td>11.0</td>
<td>26.0</td>
<td>45.3</td>
<td>71.3</td>
<td>103.0</td>
<td>141.3</td>
<td>185.3</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.5. Shortest Path Cycle Counts

As Figure 5.5 shows, the FMC takes between 3.5 and 5 times fewer cycles that the i486. This improvement is due to the large decrease in cycles required to compute expressions. The largest decrease was undoubtedly in the calculation of the address of \( d[i,j] \). Programs with more complex expressions would likely see more dramatic decreases in the number of cycles necessary to execute.
We also note that as \( n \) increases, the percentage cycle count improvement of the FMC over the i486 also seems to increase. This may, in part, be due to interrupts on the DOS 486 machine remaining active (e.g., for updating the system clock, etc.) during data collection. This may have a slight inflationary effect on the improvement with the larger readings for \( n \).

5.2.2 Deterministic Bubble Sort Cycle Counts

By examining the decision table for this problem in Figure 3.3, we can derive an equation for the number of cycles required to execute the program depending on the array size \( n \). Rule 1 always executes upon initiation and Rule 2 always executes upon termination. The outer loop variable \( k \) begins at \( n \) and decrements down to 2, causing the inner loop to execute \( n-1 \) times, therefore, Rule 3 executes once each time the inner loop executes, resetting \( j \) to 1 and decrementing \( k \). The body of the inner loop executes \( n(n-1)/2 \) times, alternating between Rules 4 and 5. When \( a[j] \) and \( a[j+1] \) are out of order, then Rule 4 executes to exchange them, after which \( j \) is incremented. Rule 5 executes when no exchange is necessary and just \( j \) is incremented.

We can define \( f(n) \) as the number of times that \( a[j] \) and \( a[j+1] \) must be exchanged. The difference between Rules 4 and 5 (\( C_{Rule4} - C_{Rule5} \)) is exactly the number of cycles it takes to exchange. With the number of cycles in Rule 5 (\( C_{Rule5} \)) being the amount it takes to increment \( j \), then the total number of cycles \( C_{DET} \) to execute the program is:

\[
C_{DET} = C_{Rule1} + C_{Rule2} + (n-1)C_{Rule3} + f(n)C_{Rule4} + \left( \frac{n(n-1)}{2} - f(n) \right)C_{Rule5}
\]

The following expressions for \( f(n) \) are for the worst and best cases of how well sorted the array is before execution begins:
\[
\text{Worst Case} \quad f(n) = \frac{n(n-1)}{2} \\
\text{Best Case} \quad f(n) = 0
\]

For example, with these equations, we can calculate exactly how long our 4 MHz FMC will take to execute the program when \(n=50\) in the worst case. From Table 3.2, we see that Rule 1 takes 14 cycles, Rule 2 takes 2, Rule 3 takes 12, Rule 4 takes 16 and Rule 5 takes 10 cycles. Plugging in the cycle counts with \(n=50\) and worst case \(f(n)\) gives:

\[
C_{\text{DET}} = 14 + 2 + (50-1)12 + \frac{50(50-1)}{2}16 + \left(\frac{50(50-1)}{2} - \frac{50(50-1)}{2}\right)10 = 20,204
\]

To compute the amount of time the program takes to execute at 4 MHz, we divide 20,204 cycles by 4,000,000 Hertz (cycles per second) yielding 5.051 mS. In the best case, we get 12,854 cycles or 3.214 mS.

5.2.3 Nondeterministic Bubble Sort Cycle Counts

As with the deterministic bubble sort problem, by examining the decision table for the nondeterministic bubble sort in Figure 4.3B, we can derive an equation for the number of cycles required to execute the program depending on the array size \(n\). Whenever there is a \(j\) that is greater than its neighbor to the right (hence out of order), then \(a[j]\) and \(a[j+1]\) are exchanged. From Table 4.2, we see that a Rule 0 is placed at the start which determines if Rule 1 or Rule 2 should execute. Rule 1 will execute exactly once, when no out of order pairs remain. Otherwise, Rule 2 executes, as many times as there are out of order pairs to exchange. Therefore, the equation which describes the number of cycles which execute depending on how sorted the array is initially is as follows:

\[
C_{\text{NONDET}} = C_{\text{Rule0}} + C_{\text{Rule1}} + f(n) C_{\text{Rule3}}
\]
with \( f(n) \) defined the same as in the deterministic case. For example, from Table 4.2, we see that Rules 0 and 2 each take 2 cycles and Rule 3 takes 13 cycles. The FMC clock frequency is 4 MHz. To compute the worst case time it would take to sort an array of 50 elements, we have:

\[
T_{\text{ND, worst}} = \frac{2 + 2 + \left(\frac{50(50-1)}{2}\right)13}{4,000,000} = 3.98 \text{mS}.
\]

In the best case, Rule 0 and Rule 1 execute once, taking two cycles each, therefore, only 1 \( \mu \text{S} \) (four cycles) is required.

Table 5.6 and Figure 5.6 show the cycle count comparisons between the two FMC and the i486 implementations for a random array. From the table we see that when \( n=10 \), the i486 program took 2,800 cycles whereas the deterministic FMC program took 700 cycles and the nondeterministic version took only 300. When \( n=50 \), the i486 version takes 57,800 cycles whereas the deterministic FMC version takes 16,600 and the nondeterministic one takes only 8,200 cycles. These figures were computed by multiplying the execution times for the random array in Tables 5.2, 5.3 and 5.4 by the megahertz of the respective processor.

As Figure 5.6 shows, the deterministic FMC version uses less than 30% of the number of cycles the i486 requires, while the nondeterministic version uses less than 15%. In the two deterministic versions (deterministic FMC and i486), this improvement is due to the large decrease in cycles required to compute expressions. The largest decrease was associated with the calculation of \( a[j] > a[j+1] \). In addition, the even larger decrease in cycle counts for the nondeterministic FMC version had to do with the fact that not all the adjacent array element pairs needed to be examined sequentially but could be done all at once. The number of actual exchanges that were required was the same. We note that deterministic programs with more complex expressions would likely see more
dramatic decreases in the number of cycles necessary to execute. An additional improvement is certainly achievable for nondeterministic algorithms when functions that deterministically require $O(n)$ steps can be done in one.

### Table 5.6. Bubble Sort Cycles Comparison (Random Array)

<table>
<thead>
<tr>
<th>('000s) n</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>0.0</td>
<td>1.3</td>
<td>2.8</td>
<td>5.5</td>
<td>9.5</td>
<td>15.0</td>
<td>21.8</td>
<td>28.8</td>
<td>37.0</td>
<td>46.5</td>
<td>57.8</td>
</tr>
<tr>
<td>FMC-Det</td>
<td>0.0</td>
<td>0.2</td>
<td>0.7</td>
<td>1.6</td>
<td>2.8</td>
<td>4.2</td>
<td>6.1</td>
<td>8.2</td>
<td>10.7</td>
<td>13.3</td>
<td>16.6</td>
</tr>
<tr>
<td>FMC-Nondet</td>
<td>0.0</td>
<td>0.1</td>
<td>0.3</td>
<td>0.9</td>
<td>1.4</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>5.2</td>
<td>6.2</td>
<td>8.2</td>
</tr>
</tbody>
</table>

![Figure 5.6. Bubble Sort Cycles Comparison (Random Array)](image)

#### 5.3. LOAD-STORE COMPARISONS

Another common method for comparing processors is to perform load–store analysis. A count of the number of times the processor needs to load and store operands across the memory bus provides a measure of efficiency by the processor which also can translate to lower program execution times. We find that functional memory systems save on loads and stores when expression computation primarily involves scalar variables. When many
array references are involved, however, bus transactions increase because temporary scalar variables must be updated.

Counting only loads and stores is somewhat unfair when comparing functional memory systems because execution steps are overlapped with the store operations, so in our analysis, we will also count execution steps.Multiplies and adds will each count one step. We find that large expressions, such as those for computing multidimensional array reference addresses can often be computed in less time with functional memory.

5.3.1 Shortest Path Load–Store Analysis

The top half of Figure 5.7 shows the number of loads, computations and stores which take place when the Pascal-von Neumann version of the shortest path program executes (without FM), depending on the number of nodes (i.e., the array $n$ size). For this analysis, we assume that all variables are stored in memory, but that there are enough registers so all intermediate computations can be stored in internal registers. Operands for each statement are fetched just once, however, no optimization is assumed across more than one statement in a block. A single dimensioned array address computation involves only one addition operation while a two dimensional array address computation involves one multiplication and one addition. Note that $d[i,j]+f[j]$ is calculated just once and stored in $d[i,j]+f[j]$. As indicated by the asterisk (*), the analysis assumes an average case time, where the if predicate is true half the time. As we can see, processor load steps consume the most operations and stores the least.

The bottom half of Figure 5.7 shows the load–store analysis for the decision table executing on a functional memory system. First, since the next rule to execute must be read each decision table iteration, $n(n-1)/2+(n-1)+2$ loads are incurred for this alone. Because operands need no longer be fetched for computation, however, we end up with fewer load steps overall than the Pascal version.
begin
f[n] := 0;
for i := n - 1 downto 1 do
begin
min := maxint;
ptr := n + 1;
for j := i + 1 to n do
begin
temp := d[i,j] + f[j];
if temp < min then
begin
min := temp;
ptr := j;
end;
end;
f[i] := min;
t[i] := ptr;
end;
end;
end

Rule: 1 2 3 4 5
lambda = 0 1 1 1 1
i >= 1 T T T F
j <= n T F F -
*d[i,j]*"f[j"] < min - - T - .5n(n-1)+(n-1)+2 - -

| f[n] := 0 | X - - - - | 1 | - | 1 |
| f[i] := min | - X - - - | 2(n-1) | - | (n-1) |
| t[i] := ptr | - X - - - | 2(n-1) | - | (n-1) |
| min := "d[i,j]+"f[j"] | X X - - - | - | - | 1+(n-1) |
| d[i,j] := d[i,j] | X X X X | 2+(n-1)+n(n-1) | - | 1+(n-1)+.5n(n-1) |

Figure 5.7. Shortest Path Load-Store Analysis
A more obvious improvement comes from the elimination of computational operations that need to be performed by the processor. The Pascal version requires on the order of $3n^2$ computational steps which must be performed sequentially by the processor. We call this the "von Neumann ALU bottleneck." With functional memory, these operations are performed in parallel with the loads and stores.

In an ideal case, functional memory reduces loads and stores as well as eliminating operations. For example, the statement $d[i,j] := k \cdot (m+n)$ executing on a regular von Neumann processor would require (a) five loads for $k$, $m$, $n$, $i$ and $j$, (b) four computations for $m+n$, $k \cdot i \cdot j$ and $+d$, and (c) one store for $d[i,j]$. Functional memory requires only (a) two loads for $k \cdot (m+n)$ and $d[i,j]$ and (b) one store for $d[i,j]$.

As we see from this example, we have a case where functional memory is at a disadvantage, which is when some expression operands are array elements. For each array element in an expression, an extra store is required to store the value back out into memory. The statement $\min := d[i,j] + f[j]$ in Pascal requires four loads for $i$, $j$, $d[i,j]$ and $f[j]$, and one store for $\min$. $d + i \cdot j$, $f + j$ and $d[i,j] + f[j]$ are operations that must be performed by the processor. In a functional memory system, five loads are needed to retrieve $d[i,j], f[j]$ and $d[i,j] + f[j]$ and three stores for "$d[i,j]\$", "$f[j]\$" and $\min$, which is one extra load and two extra stores. Functional memory still, however comes out ahead overall because four sequential computational steps are eliminated. (This assumes each step is the same period, which likely is not the case – multiplication can be 5 to 20 times longer.).

As a result, Figure 5.7 shows that stores for the FM decision table were on the order of $2n^2$, compared to $1.5n^2$ for the Pascal von Neumann version. Overall, if we count loads and stores about equally, both versions require about $6n^2$ loads plus stores to execute.
Table 5.7 shows the step counts for $n=10$ to 80 derived from the load, computation and store equations shown in Figure 5.7, and Figure 5.8 shows these results graphically. Our example was designed to handle an array of size up to $n=80$. The first five rows show the step counts for the Pascal von Neumann version (Pascal vN). Row 4 gives the sum of the loads plus the stores for each array size $n$, while Row 5 totals all the steps. We see that when $n=80$, the Pascal version requires 39,030 loads and stores with an additional 19,357 computation steps, bringing the total number of steps to 58,387. Rows 6, 7 and 8 give the counts for the decision table executing on the FMC (DT-FM). Row 9 sums the total number of steps for the DT-FM version, which at $n=80$ amounts to 36,358 steps, or 22,029 fewer than the Pascal vN version.

Below the double line is shown the percent difference in load plus store counts and steps overall between the two versions. Loads plus stores for the DT-FM version was slightly greater than the Pascal vN version at $n=10$, however, the DT-FM loads plus stores was slightly lower for $n \geq 20$. When counting the computation steps, the DT-FM program shows a 29.79% improvement in total steps over the Pascal von Neumann version at $n=10$, and reaches 37.73% for $n=80$. As the graph illustrates, it is in the elimination of separate computational steps required by the processor of a functional memory system where the potential for savings lie.

5.4 CONCLUSION

Our execution analyses found that for actual execution times, our 4 MHz FMC is slower than a 25 MHz Intel 486/SX, which is not surprising. However, when cycle counts for the different program implementations are compared, the FMC executed the same program in 3 to 5 times fewer cycles. The shortest path load–store analysis showed
our functional memory computer comparable in loads plus stores, but the zero times for computation result in a 35% improvement overall in execution steps.

**Table 5.7. Shortest Path Load/Store Count Comparison**

<table>
<thead>
<tr>
<th>(#)</th>
<th>n = 10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal vN Loads</td>
<td>479</td>
<td>1,864</td>
<td>4,149</td>
<td>7,334</td>
<td>11,419</td>
<td>16,404</td>
<td>22,289</td>
<td>29,074</td>
</tr>
<tr>
<td>Pascal vN Computations</td>
<td>317</td>
<td>1,237</td>
<td>2,757</td>
<td>4,877</td>
<td>7,597</td>
<td>10,917</td>
<td>14,837</td>
<td>19,357</td>
</tr>
<tr>
<td>Pascal vN Stores</td>
<td>191</td>
<td>686</td>
<td>1,481</td>
<td>2,576</td>
<td>3,971</td>
<td>5,666</td>
<td>7,661</td>
<td>9,956</td>
</tr>
<tr>
<td>Pascal vN Loads+Stores</td>
<td>670</td>
<td>2,550</td>
<td>5,630</td>
<td>9,910</td>
<td>15,390</td>
<td>22,070</td>
<td>29,950</td>
<td>39,030</td>
</tr>
<tr>
<td>Pascal vN Total steps</td>
<td>987</td>
<td>3,787</td>
<td>8,387</td>
<td>14,787</td>
<td>22,987</td>
<td>32,987</td>
<td>44,787</td>
<td>58,387</td>
</tr>
<tr>
<td>DT-FM Loads</td>
<td>433</td>
<td>1,568</td>
<td>3,403</td>
<td>5,938</td>
<td>9,173</td>
<td>13,108</td>
<td>17,743</td>
<td>23,078</td>
</tr>
<tr>
<td>DT-FM Computation</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DT-FM Stores</td>
<td>260</td>
<td>920</td>
<td>1,980</td>
<td>3,440</td>
<td>5,300</td>
<td>7,560</td>
<td>10,220</td>
<td>13,280</td>
</tr>
<tr>
<td>DT-FM Total steps</td>
<td>693</td>
<td>2,488</td>
<td>5,383</td>
<td>9,378</td>
<td>14,473</td>
<td>20,668</td>
<td>27,963</td>
<td>36,358</td>
</tr>
</tbody>
</table>

% Diff. Loads+Stores | -3.43% | 2.43% | 4.39% | 5.37% | 5.96% | 6.35% | 6.63% | 6.85% |
% Diff. Total steps  | 29.79% | 34.30% | 35.82% | 36.58% | 37.04% | 37.35% | 37.56% | 37.73% |

**Figure 5.8. Shortest Path Load/Store Count Comparison**
CHAPTER 6. THE DESIGN AND IMPLEMENTATION OF A COMPILER FOR A FUNCTIONAL MEMORY COMPUTER

A high level language compiler for a functional memory computer (FMC) offers several unique challenges to the compiler writer. On conventional machines, statements and expressions are parsed into a sequence of move, arithmetic and logic instructions if the target is von Neumann, or they are parsed into a data flow graph of simple expression assignments if the target is a data flow machine [Ackerman, 1982]. For a FMC, we must do a little of both; statements must be parsed into a sequence of move instructions, and expressions must be parsed into dataflow implementations of the combinational logic functions for computing an expression.

This chapter describes the implementation of our FMC compiler which translates a decision table into the microcode and FPGA source files. These output files are then compiled further (or “assembled”) into execution load modules using existing off-the-shelf tools. The FMC compiler is complete in that the user requires no combinational logic design experience -- only an understanding of how to write decision table programs. The compiler was written in Visual Basic and is fully functional as described herein. It provides a useful graphical user interface with drop-down menus, allowing the intermediate text and tables generated during the compiling process to be examined. The output files can be examined and additional compiler tools can be executed from a drop-down menu. The first part this chapter describes hierarchically the design of the compiler portion of the system. The second part describes the user interface.

6.1 THE COMPILING PROCESS

As Figure 6.1 illustrates, compiling a high-level source program (.SRC) for a FMC involves (1) assembling a language source program (.ASM) for the minimal processor's
(mP) assembler and (2) generating the FPGA source programs (.PDS) for each FPGA in FM. The minimal processor is a "bare-bones" 16-bit microprogrammable processor implementing only the minimal instruction set necessary for program execution. Programs can be up to 8K microinstructions. The compiler also generates FPGA code for up to three XC 3090 and one XC 3064 FPGA, for up to 1,184 configurable logic blocks (CLBs) connected to 16K of RAM.

![Decision Table Diagram](image)

**Figure 6.1. The FMC Compiler Function**

As Figure 6.2 suggests, we have designed our system to take as much advantage of off-the-shelf software as possible. Our minimal processor is microprogrammable, with an instruction format designed so any conventional assembler that supports label-equates and define-word pseudo instructions can be used to translate microinstruction mnemonics into a microcode load module. The MCS-51 assembler used in our system is the same one used for the system processor. It can be executed from a compiler menu selection to
produce the minimal processor load module (.HEX) directly. (See Section 6.2 for details.)

**Figure 6.2. Completion of the Compilation Process Using Off-The-Shelf Tools**
PALASM (Programmable Array Logic ASseMbler) is a flexible but skeletal text-based language for specifying VLSI circuits with a syntax well suited to machine generation. For each chip, the PALASM source file (.PDS) must be further compiled into an FPGA .BIT file, which contains the one-zero bit pattern specifying the connections between and within the CLBs inside the chip. This process shown on the right in Figure 6.2 is greatly simplified to show only the basic steps of which the FMC programmer must be aware. A FMC compiler drop-down menu selects the PDS2XNF assembler tool to execute directly so the user can check that the first stage of the chip compilation process will successfully complete producing the .XNF file. A DOS batch file is also created to complete the creation of the .BIT file because some of the tools used during this stage will not run with Windows. Notice also that this stage of the process can take up to 15 hours for each chip, because the placing of the CLBs within the chip and the routing of the connections between the CLBs is a tedious time consuming process that uses random statistical techniques in order to optimize its chances of finding a routable placement. After a .BIT file has been created for each chip, the MAKEPROM XILINX tool is used to link them all together (along with the minimal processor specification RMC7.BIT) into the FMC XILINX FPGA load module.

The main function of the FMC compiler is to generate the machine level source programs, namely the .ASM and .PDS files, as shown in Figure 6.1. Figure 6.3 shows a breakdown of the compiling stages for producing both the microcode and FPGA source files. We begin by lexically scanning the decision table ASCII source text file and producing several forms of intermediate text. This module (1.0) produces the symbol table as well as intermediate forms for all four quadrants of the decision table. Next, the Generate Memory Map (2.0) module produces a memory table which contains the location, size and type for each variable and expression. With this information, module
3.0 can generate the microcode source file which contains the move instructions necessary for performing the assignment statements for each rule. With this, the starting addresses for each rule are known and Module 4.0 can produce the PALASM source code files.

![Decision Table Compilation Stages](image)

**Figure 6.3. Decision Table Compilation Stages**

The following sections 6.1.1, 6.1.2, 6.1.3, and 6.1.4 describe the implementation of each of these four modules in more detail.
6.1.1 Generate Intermediate Text (1.0)

The main function of module 1.0 is to produce the symbol table and record the stubs and entries in respective arrays. After the symbol table has been initialized with all reserved words and character strings, the decision table file is parsed and an array of symbol IDs (SymProg) is created from ASCII text file. The ConStub array contains the condition stub expression as a string. The ConEnt two dimensional array contains the condition entries by condition stub (rows) and rule (columns). The ActEnt is a similarly dimensioned array containing the action stub entries. The ActStub array contains several fields. The .Statement field contains the full action stub as a string. The .SrcType field contains the data type on the right side of the assignment (:=) symbol (e.g., constant, scalar variable or expression, or an array element). The .SrcExpression field contains the right side expression of the assignment statement. The .DestType field indicates whether the left side variable is a scalar or an array element. The .DestVariable field contains the destination variable symbol.

![Figure 6.4. 1.0 Generate Intermediate Text Hierarchy Chart](image-url)
Parsing the decision table (1.2) relies on a subroutine which strips the next symbol off the decision table text-file (1.2.1) and one which inserts new symbols into the symbol table and retrieves their table index (1.2.2). Figure 6.5 shows the syntax diagram for the basic identifier in a decision table. It has been enhanced slightly from standard Pascal in that arbitrary strings of characters can be identifiers as long as they are surrounded by double quotes. As our compiler does not yet automatically allocate and update temporary scalars for array references in expressions (as in the shortest path example in Chapter 3), double quoting can be used for naming the temporary scalars the same as the reference it replaces (e.g., “a[j]” replaces a[j]). Identifiers can also include underscores (_), exclamation points (!), periods (.) and carets (^).

![Symbol Table Identifier Syntax Diagram](image)

An additional language feature is the ability to assign action stubs to specific FPGA chips. This is useful for large decision tables or decision tables with large operands when the program no longer fits into one FPGA chip. Following the last action entry in a row, a single digit of 1, 2, 3 or 4 will assign the action stub in that row to the FPGA chip designated by the single digit. This feature will be used in Chapter 7.
6.1.2 Generate Memory Map (2.0)

Figure 6.6 shows the hierarchy chart for the module that generates the functional memory map (2.0). The memory map contains the same information that a memory map would contain for a standard von Neumann computer and also the expression information for the FPGA logic. In addition, since CLBs are at a premium, bit lengths are also kept and maintained for each register and expression variable.

The memory map is filled in three basic stages. First the function declaration section of the decision table is parsed (2.1). The syntax of a function declaration is shown in Figure 6.7. A function declaration allows the input and output addresses for a special function to be allocated so the rest of the decision table can reference them normally.
Also, the function is entered separately in the memory table, to be expanded into PALASM code in later in Module 4.0. The decision table may contain more than one function declaration.

Figure 6.7. Syntax Diagrams for Function and Variable Declarations

Module 2.2 parses the variable declaration section of the decision table. Addresses are allocated for each variable declared. Figure 6.7 shows the legal syntax for declaring variables and arrays recognized by Module 2.2. Module 2.3 parses the action stubs statements to allocate locations for array address and assignment statement expression
calculations. Module 2.3 also produces the source and destination values (either addresses or constants) for each action stub.

Figure 6.7 also shows that only singly dimensioned arrays are supported. Multiple dimensioned arrays require multiplication to compute element addresses. Although implemented by hand in the shortest path example in Chapter 3, our compiler, as of yet, does not allow arrays of more than one dimension. Also note that an optional colon followed by a constant can be used in simple type declarations to limit the size of the register and hence operand widths for the operators during PALASM generation.

6.1.3 Generate Microcode Source Module (3.0)

Module 3.0 generates the microcode file containing the assembly language source code for the set of move instructions for each rule. As Figure 6.8 illustrates, Module 3.0 relies on Modules 3.1 for initializing the microcode .ASM file, and Module 3.2 for processing the action entries for each rule. Module 3.1 contains the set of equates defining the microinstruction mnemonics used for implementing the minimal instruction set as defined in Table 2.2 of Chapter 2.

As Figure 6.8 also shows, Module 3.2 expands each minimal processor instruction type into the mnemonic microinstruction codes for implementing that action stub assignment using eight small subroutines, one for each type. With each instruction expanded, the rule starting addresses are recorded in the RuleTable array.

6.1.4 Generate FPGA Source Modules (4.0)

Module 1.0 generated the condition stub array (ConStub) and the condition entry table (ConEnt) which contains the expressions and combination of expressions for determining which rule to execute. Module 2.0 generated the memory address table (MemTable) which contains the input and output address information for the FPGA I/O. It also
contains the action stub array address references and expressions which must be implemented in the FPGA. With the rule starting addresses now known from Module 3.0, Module 4.0 can then generate the FPGA PALASM files.

As Figure 6.9 illustrates, Module 4.0 uses the Memory Table as its primary input. It relies on two main subroutines for generating up to four FPGA .PDS files and the batch file for completing the compilation process for each chip. Module 4.1 executes once initializing the batch file and Module 4.2 executes once for each chip, generating a PALASM .PDS file for that chip.
Module 4.2 relies on nine subroutines for generating the PALASM for one XILINX chip. Module 4.2.1 executes first for each chip, opening a new .PDS file and storing the necessary header information for interfacing to the circuit shell shown in Figure 2.3.

Module 4.2.2 expands any function macros that are declared using a “func” declaration. This feature is demonstrated in Chapter 7. Module 4.2.3 generates the equations for evaluating the condition stubs. This logic is always implemented in the first FPGA chip and feeds the condition stub results to the logic generated in Module 4.2.4, which computes the rule address (@rule). Module 4.2.5 generates the expression logic which computes the right sides of assignment statements and array reference addresses.
As expression logic is being generated for each chip, variables are marked for Module 4.2.6 which generates the input register logic. Module 4.2.7 generates the output multiplexer logic for all the expressions that were implemented in this chip. Finally, Module 4.2.8 generates the address select logic for all the input registers and output multiplexers allocated in this chip.

With the PALASM file for the chip completed and closed, Module 4.2.9 adds its name to the batch file for producing the FPGA load module. In the next seven sections we explain Modules 4.2.2 through 4.2.8 for generating the PALASM equations in further detail.

6.1.4.1 Expand Function Macros (4.2.2)

As illustrated in Figure 6.10, Module 4.2.2 expands any special function declarations the program may contain. Added functions can take advantage of low level parallelism that cannot normally be expressed using standard infix operators, which can appear in expressions (as defined in Section 6.1.4.3), and for which there are built-in operator macros (e.g., as described in Section 3.1.4). Special functions must be declared in a function declaration. The first identifier names a .DEF file containing a PALASM macro to be expanded. The .Expression field of a function declaration in the memory table lists, in order, the input and output variables which are used when the function is expanded.

Module 4.2.2.1 opens the .DEF file for the declared function. The .DEF file contains capital letters (e.g., A, B, ...) that are replaced with the PALASM signal names corresponding to the variables listed in the function declaration. The capital letter A is replaced with the signal name for the first variable. B is replaced with the second one, and so on. A special function can contain up to 26 input/output locations. Module 4.2.2.2 builds the Params array which contains the PALASM signal name to be used for each capital letter in the .DEF file. Module 4.2.2.3 reads each character in the .DEF file
and transfers it over to the PALASM file, except when a capital letter is encountered, in
which case it is replaced using the PALASM signal name contained in the Params array.

![Diagram showing the hierarchy of FPGA Special Function Expansion](image)

**Figure 6.10. 4.2.2 FPGA Special Function Expansion Hierarchy Chart**

Module 4.2.2 can expand more than one consecutively listed function, and continues
to do so until a "var" or a "dtbegin" identifier is encountered.

6.1.4.2 Generate Condition Stub Logic (4.2.3)

Module 4.2.3 generates the combinational logic for evaluating the condition stubs.
Each condition stub consists of one expression which may contain one or more operands
or operators. As shown in Figure 6.11, generating the condition stub logic relies on
Module 4.2.3.1, which is a recursive subroutine that will generate the PALASM for computing an expression as defined by the syntax shown in Figure 6.12.

**Figure 6.11. 4.2.3 Generate FPGA Condition Stub Logic Hierarchy Chart**

For each condition stub listed in the ConStub array, Parse Expression (4.2.3.1) is called to generate the PALASM code. A final equation for each stub provides the signal name input to the rule address logic (Module 4.2.4).

**6.1.4.3 Parsing Expressions (4.2.3.1)**

Figure 6.12 shows the syntax diagrams for valid expressions in our FMC compiler. Implementing the PALASM Not-Equal-To, Less-Than and Add modules described in Chapter 2, the eight operators within the Expression and Simple Expression syntax diagrams were implemented. Also, a bit-by-bit logical-OR operator was added. For Terms, shift-left and shift-right by constants are implemented as "\times 2^N" ("times 2 to the power of") and "\text{div} 2^N" ("div 2 to the power of"). In the future, shifting by variable integer amounts will also be implemented, which will require no more than a 16 position barrel shifter. A bit-by-bit logical-AND operator is also implemented at the Term level.
Factors can be constants, variables or other expressions in parentheses. Variables within an expression can only be scalar in our current implementation. For array elements to be used in an expression, they first have to be copied into scalar variable locations.
Figure 6.13 shows the hierarchy chart for the Parse Expression (4.2.3.1) module. The expression input string is parsed, and at each level, PALASM is generated as needed. PALASM for the bit-by-bit logical-OR and logical-AND operators are both generated using Module 4.2.3.1.1.3, which accepts as a parameter the logical operator symbol to use (i.e., ‘+’ for logical-OR and ‘*’ for logical-AND).

At the lowest level (4.2.3.1.1.1.1), the Parse Factor subroutine either accesses the memory table to identify the address location of a variable, or recursively calls Module
4.2.3.1 again to parse an inner expression. Each variable is flagged as being used in the current chip in the MemTable array so Module 4.2.6 knows to generate an input register and Module 4.2.8 knows to generate select logic for its address.

We note that the input register operand sizes are always known before any operator PALASM is generated. This is because Module 4.2.3.1.1.1 (Parse Factor) always executes before Generate <>, Generate <, Generate + or Generate Shift ever executes to generate PALASM code. These modules generate PALASM for bit widths only as wide as necessary, depending on the widths of the input operands.

6.1.4.4 Generate Rule Address Logic (4.2.4)

Module 4.2.4 generates the combinational logic which calculates the microprogram address of the rule to execute. Figure 6.14 shows the hierarchy chart for the Generate Rule Address Logic module.

Based on the condition entries table and the condition stub expression output signals, Module 4.2.4.1 generates a separate select signal for each rule, recorded in the
Expression field of the RuleTable array. At present, only one CLB is used for each rule select signal, which limits the number of condition stubs and the size of lambda. Adding one CLB level of logic will increase the sum of the number of condition stubs and lambda bits to 25. For each address bit position, Module 4.2.4.2 generates a statement which logically ORs each rule select signal that contains a 1 in that bit position. This also results in a limitation of five rules because each address bit is generated using a single CLB. This can easily be expanded to allow for up to 25 rules by adding one more CLB logic level.

6.1.4.5 Generate Action Stub Logic (4.2.5)

Module 4.2.5 generates the PALASM combinational logic which calculates the expressions found on the right side of the action stub assignment statements, as well as the logic for calculating all the array element addresses used in the program. As the syntax diagram shows at the top in Figure 6.15, an action stub can contain either a scalar variable or an array element on the left side of the assignment operator (:=), and an expression or another array element on the right side.

Module 4.2.5 obtains the expression and array strings to parse from the MemTable .Expression field. For the right side of an assignment statement, Module 4.2.3.1 is called directly to parse the expression, and return the PALASM signal names which contain the computation results. For an array element address, either on the left or right sides of the action stub, Module 4.2.5.1 is called which itself calls 4.2.3.1 to parse the expression for the element index. Whenever a reference to a variable is encountered, it is flagged for inclusion in the current chip by Module 4.2.6, which is discussed next.
6.1.4.6 Generate Input Register Logic (4.2.6)

Once all the expression logic has been generated (by Modules 4.2.2, 4.2.3 and 4.2.5), all the necessary input registers are known and Module 4.2.6 (shown in Figure 6.16) can generate the input register logic. For those variables in the MemTable array flagged as being used for this chip (.Chip), Module 4.2.6 uses the .Addr and .Width fields to generate the signal name connections and the number of input registers. The lambda register is generated separately by Module 4.2.6.1 because lambda is a reserved
word that is treated specially by the rule address logic (generated by Module 4.2.4). In the future, the register will be self modified based on the rule address.

Module 4.2.6.2 generates the input registers for all the remaining input registers. Each register bit requires signal definitions for the write clock (WRLC for the lower order odd address byte, and WRHC for the upper order even address byte), the clock enable (generated later by Module 4.2.8) and the data input bit (defined in the PALASM header generated by Module 4.2.1).

![Diagram: Generate FPGA Input Register Logic Hierarchy Chart]

Figure 6.16. 4.2.6 Generate FPGA Input Register Logic Hierarchy Chart

6.1.4.7 Generate Output Multiplexer Logic (4.2.7)

Module 4.2.7 illustrated in Figure 6.17 generates the output multiplexer logic which is responsible for gating the expression and array address computations out of the chip when the processor reads function memory. For each data bit, DO0 through DO15 (defined in the PALASM Header generated by Module 4.2.1), each memory table entry is examined if it contains a data bit to be gated onto the data bus to the processor when its address is selected. Each data bit therefore must be logically ANDed with the address select line that will be generated in Module 4.2.8. The signal must also be logically ANDed with the /RDC read signal from the processor. Because of the XILINX 3000
series CLB limitation of five inputs, three logic levels are used. First, each expression or array address bit to be gated out is logically ANDed with its select line. Up to two output bits are logically ORed together for a logic level one signal for this data bit. Up to five logic level one signals are ORed together to make a logic level two signal. Up to four logic level two signals are ORed together, and then ANDed with the /RDC signal for the logic level three output signal for the particular address bit. Therefore, up to 40 different output addresses can be supported.

![Diagram](image)

Figure 6.17. 4.2.7 Generate FPGA Output Multiplexer Logic Hierarchy Chart

6.1.4.8 Generate Address Select Logic (4.2.8)

Finally, with the input registers and output multiplexer generated, Module 4.2.8 (shown in Figure 6.18) can generate the address select logic for the input registers and output multiplexer locations used in the chip. For each location in the memory table marked as used in the current chip, an address select equation is generated by Module 4.2.8.1. These equations are precisely the “sel_0**” equations shown in Figure 2.4. Each address nibble (hexadecimal digit) that is used is recorded in the MemAddrNib array. Assuming a 64K byte address space, there are 16 possible hex digit values for each
of the four hexadecimal positions for addressing 64K, therefore the most number of address digits that would have to be decoded is 64. After all the memory select equations have been generated, the equations for decoding only those address nibbles recorded in MemAddrNib are generated by Module 4.2.8.2. These equations are the same as the “AS**” signals also shown in Figure 2.4.

![Figure 6.18. 4.2.8 Generate FPGA Address Select Logic Hierarchy Chart](image)

This concludes the description of the code generation of the compiler. Next we will discuss the user interface.

### 6.2. The User Interface

The compiler was written in Visual Basic to simplify the user interface. Figure 6.19 shows a picture of the opening screen of the FMC decision table compiler. On the left is
a modifiable introduction screen which is used to explain any special instructions to the user. This window can be used for debugging notes when developing decision table programs or compiler extensions. Across the top are four drop-down menus. Each will be discussed in further detail below.

The open window on the right is where a new decision table can be entered or loaded from a ASCII source file. During successive stages of the compilation process, compilation data is indicated when known along the right side of the decision table window. Each of the four drop down menus will now be described.

### 6.2.1 File Menu

As Figure 6.20 shows, the File menu is similar to most other Windows File menus. Users can enter a new table or open an existing one. Decision tables should be saved

![Figure 6.19. FMC Compiler -- Opening Screen](image)
periodically during development using the File Save function. Save As allows users to save decision tables under different names. Shown is the result after opening the bubble.src source file that we will be using as an example through out this discussion.

![FMC DECISION TABLE COMPILER](image)

**Figure 6.20. File Menu**

Notice that all five of the File menu items can be selected using Alt keystrokes. Alt-F-O selects File Open. Alt-F-S saves your work in progress. The compiler should always be terminated using the File Exit command, or keystrokes Alt-F-X. The compiler notifies users if they forget to save their file before exiting or loading a new one.

### 6.2.2 View Menu

The View menu shown in Figure 6.21 lets the user view a variety of different intermediate tables and data in different ways. The first two selections control the width of the left and right windows. The left window shows the various tables, and the right
window the decision table source program. When the DT Full View selection is checked, the decision table window takes up the entire width of the screen, as shown in Figure 6.22.

![Figure 6.21. View Menu](image)

Also from the View menu users can examine the intermediate tables and text that are generated during the compilation process. The Memory Map and the Execution Table are viewable. These two tables are used to generate the PALASM .PDS and .ASM microcode, which is also viewable.

Being able to examine the Symbol Table and Intermediate Text can be very useful when debugging. The Introduction selection reloads the View window with the introduction file in the current directory. This window can be modified and resaved to
keep track of notes during debugging. The Output-to-File switch allows everything viewed in the left window to be stored into the .LIS file.

6.2.3 Generate Menu

The Standard Output selection under the Generate menu shown in Figure 6.23 compiles the decision table, producing a .LIS compiler listing file, the .ASM microcode file and the PALASM .PDS file(s). These three files for the bubble sort program shown in Figure 6.22 are contained in Appendixes 1, 2 and 3 respectively. Generating the Standard Output can also be accomplished by typing ALT-G-O. When the compilation is complete, the resulting statistics appear along the right edge of the screen as shown.
Compiler Modules 1.0, 2.0, 3.0 and 4.0 described earlier in this chapter can also be executed separately in sequence from the Generate menu. This is useful when debugging to observe the intermediate data structures as they are built.

Figure 6.23. Generate Menu

The standard output included in the .LIS file begins with the decision table text file followed by the compilation statistics listed along the right side of the screen. Next is the Memory Map is included followed by the Execution Table. Figure 6.24 shows both these tables for our bubble sort example using the View Window Full View selection from the View menu. The information provided by our compiler is the same as that listed back in Tables 3.1 and 3.2 for the deterministic bubble sort program described in Chapter 3 with two exceptions.
### Functional Memory Map

<table>
<thead>
<tr>
<th>Assigned Chip</th>
<th>Address or Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>lambda</td>
<td>R 1</td>
</tr>
<tr>
<td>@Rule</td>
<td>P 1</td>
</tr>
<tr>
<td>n</td>
<td>R 0</td>
</tr>
<tr>
<td>a</td>
<td>C 0</td>
</tr>
<tr>
<td>j</td>
<td>R 1</td>
</tr>
<tr>
<td>k</td>
<td>R 1</td>
</tr>
<tr>
<td>&quot;a[j]&quot;</td>
<td>R 1</td>
</tr>
<tr>
<td>&quot;a[j+1]&quot;</td>
<td>R 1</td>
</tr>
<tr>
<td>k-1</td>
<td>H 1</td>
</tr>
<tr>
<td>8a[j]</td>
<td>A 1</td>
</tr>
<tr>
<td>8a[j+1]</td>
<td>A 1</td>
</tr>
<tr>
<td>j+1</td>
<td>H 1</td>
</tr>
</tbody>
</table>

*Types*

- A-Indirect Address
- C-Array Base Address
- D-Function Macro Declaration
- E-Expression Output

### Execution Table

<table>
<thead>
<tr>
<th>Addr</th>
<th>Statement</th>
<th>MP</th>
<th>Dest</th>
<th>Src</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>004</td>
<td>k:=n</td>
<td>DD</td>
<td>006X</td>
<td>0004</td>
<td>2</td>
</tr>
<tr>
<td>00C</td>
<td>j:=1</td>
<td>DC</td>
<td>006C</td>
<td>0001</td>
<td>2</td>
</tr>
<tr>
<td>014</td>
<td>&quot;a[j]&quot;:=a[j]</td>
<td>DI</td>
<td>0070</td>
<td>0076</td>
<td>3</td>
</tr>
<tr>
<td>020</td>
<td>&quot;a[j+1]&quot;:=a[j+1]</td>
<td>DI</td>
<td>0072</td>
<td>0078</td>
<td>3</td>
</tr>
<tr>
<td>02C</td>
<td>lambda:=</td>
<td>DC</td>
<td>0000</td>
<td>0001</td>
<td>2</td>
</tr>
<tr>
<td>034</td>
<td>goto $rule</td>
<td>JX</td>
<td>0002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03C</td>
<td>exit</td>
<td>IX</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>044</td>
<td>j:=l</td>
<td>DC</td>
<td>006C</td>
<td>0001</td>
<td>2</td>
</tr>
<tr>
<td>04C</td>
<td>k:=k-1</td>
<td>DX</td>
<td>006X</td>
<td>0074</td>
<td>2</td>
</tr>
<tr>
<td>054</td>
<td>&quot;a[j]&quot;:=a[j]</td>
<td>DI</td>
<td>0070</td>
<td>0076</td>
<td>3</td>
</tr>
<tr>
<td>060</td>
<td>&quot;a[j+1]&quot;:=a[j+1]</td>
<td>DI</td>
<td>0072</td>
<td>0078</td>
<td>3</td>
</tr>
<tr>
<td>06C</td>
<td>goto $rule</td>
<td>JX</td>
<td>0002</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Rules*

- Rule 1=/L1*/L0: 4 condition
- Rule 2=/L1*O/C2: 55 lines MC
- Rule 3=/L1*O/C2*O: 168 CLBs
- Rule 4=/L1*O/C2*O/C3: 1 chip

---

Figure 6.24. Memory Map and Execution Table
First, the number of bits width (#Bits) for a Memory Map entry reflects the number of bits to implement when generating the PALASM code. Our compiler does not yet handle the implementation of arrays in functional memory, so the number of bits width for arrays defaults to zero. The .Expression field is also included in the Memory Map, which is used by Module 4.0 to generate the PALASM.

6.2.4 Compile Menu

Figure 6.25 shows the microcode source file (selection Microcode .ASM File from the View menu) for the bubblesort decision table in full view. This file can be assembled from the FMC Decision Table Compiler by selecting Assemble Microcode from the Compile menu. We see the file begins with the microcode mnemonic definitions for implementing the minimal processor instruction set.
Compiling the decision table resulted in one FPGA chip as shown under the Compile menu is a PDS2XNF Chip 1 selection for executing the first stage of the chip compile. Figure 6.26 shows the PALASM source file scrolled down to the @rule address generation logic. The equations generated are the same as the logic shown in Table 3.3.

Our compiler (and hardware) can handle decision table programs up to four chips in which case selections PDS2XNF Chip 1, PDS2XNF Chip 2, PDS2XNF Chip 3 and PDS2XNF Chip 4 would also appear in the second group of selections on the Compile menu.

![Figure 6.26. Assemble PALASM .PDS to .XNF](image-url)

146
CHAPTER 7. APPLICATION: EXAMPLES IN IMAGE PROCESSING

In this chapter we examine the efficacy of using functional memory in a practical application. We have chosen image processing and implement two example programs using our FMC compiler. The first example is a 2:1 image magnification program that uses a special convolution operator for producing four output pixels at a time. The second example illustrates how a special row-column summation operator can be used in a character recognition program to calculate horizontal and vertical black pixel histograms in one pass for a 16 by 16 character.

Each program was written, compiled and implemented using the FMC compiler described in Chapter 6. In this scenario, the “configuration” of the compiler (for this particular application) would first involve writing the special function operators. This would likely be performed by the configuration engineer who would need to have logic design and PALASM experience. Once the special function definition (.DEF) files have been debugged, they would be available for the “users” for writing decision table application programs. The decision table programmer therefore need not have any logic design (or PALASM) experience.

There are several reasons why our functional memory approach may provide an effective parallel processing architecture for an image processor. Image processing can take advantage of as much parallelism as available with operands that are often quite small. This makes very large scale integrated (VLSI) circuit solutions attractive. In the range of image processing functions, there are many operations that are similar but not the same. This makes field programmable gate array (FPGA) solutions attractive, because FPGAs are VLSI chips whose circuits could be reprogrammed to perform each slightly differing function separately. When these functions are rarely used simultaneously, then
the total FPGA hardware can be minimal because a smaller amount of logic is needed for any one particular function.

These characteristics favor the FPGA solution because alternative parallel processing architectures are too expensive and inflexible to be able to deliver the same level of processing power as efficiently. Since other custom computing approaches still use the conventional processor-memory division of labor, we believe our FMC approach would be simpler and equally as effective because of how naturally functional memory implements expression level parallelism.

7.1 CONVOLUTION

Convolution is a matrix operation that is useful in many image processing contexts. Pratt [1991] shows examples of convolution being used for image analysis functions such as edge detection, as well as image improvement functions such as noise cleaning, edge crispening and image magnification. Convolution involves multiplying termwise each element of the convolution matrix with a same dimension matrix of pixels values. For example, applying a convolution matrix (on the right) to a matrix of pixel values \((p_{ij}, i = 1,2,3, j = 1,2,3)\) yields a scalar value that is equal to the sum of the element by element multiplications for each position:

\[
\begin{bmatrix}
  p_{1,1} & p_{1,2} & p_{1,3} \\
  p_{2,1} & p_{2,2} & p_{2,3} \\
  p_{3,1} & p_{3,2} & p_{3,3}
\end{bmatrix}
\begin{bmatrix}
  1 & 2 & 1 \\
  2 & 4 & 2 \\
  1 & 2 & 1
\end{bmatrix}
= p_{1,1} + 2p_{1,2} + p_{1,3} + 2p_{2,1} + 4p_{2,2} + 2p_{2,3} + p_{3,1} + 2p_{3,2} + p_{3,3}
\]

Different convolution matrixes are used for different purposes. For edge detection, for example, Laplacian techniques employ convolution to detect spatial changes in the second derivative. Pratt (1991) describes the theory behind deriving the matrix values. The convolution operation is performed on all nine-pixel squares in the image, so each
pixel (except the ones along the edge) are involved in nine convolution operations. Two common Laplacian impulse response arrays are:

\[ H_1 = \frac{1}{4} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \text{and} \quad H_2 = \frac{1}{8} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \]

For continuous noise, such as additive uniform or Gaussian distributed noise, a low-pass filter with impulse response:

\[ H_3 = \left[ \frac{1}{b+2} \right]^2 \begin{bmatrix} 1 & b & 1 \\ b & b^2 & b \\ 1 & b & 1 \end{bmatrix} \]

can be used. \( H_3 \) defines several feasible versions, however,

\[ H_4 = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \]

works best in our implementation because the scaling factor is a power of two.

Convolution can also be used for nonlinear noise cleaning, for example the noise cleaning operator,

\[ H_5 = \frac{1}{8} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \]

can be used for outlier noise when the magnitude of the difference between a particular pixel and its neighbors is greater than some threshold.

Linear edge crispening is useful for medical imaging and can also be performed using discrete convolution with a high-pass impulse array. Three common 3 x 3 high-pass masks are as follows:
H_6 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad H_7 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad H_8 = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix}

Notice the masks have the property that the sum of their elements is unity in order to avoid amplitude bias in the processed image. H_7 has been found to be excellent for edge crispening on chest X-rays [Pratt, 1991].

7.1.1 Image Magnification

When the magnification zoom factor of an image is integer, pixel estimation can also be implemented by convolution [Pratt, 1991]. For example, Figure 7.1 illustrates that for a magnification factor of two, we begin by conceptually interleaving each row and column with zeros. This doubles the horizontal and vertical dimensions of the image. The convolution pass fills in the zeros with interpolated values.

![Figure 7.1. Steps for 2 to 1 Image Magnification](image)

Figure 7.2 lists four interpolation kernels for 2:1 magnification derived in Pratt [1991]. Interpolation error (from using a smaller kernel) results in jaggy line artifacts in the output image. Loss of high spatial frequency detail can also result from using a larger kernel.

The Peg and Pyramid kernels require only additions. Implementing the Peg involves simply summing the three forward neighboring pixels. Using the Pyramid may involve
shifting pixel values left one or two bit positions before the summation. The Bell and
Cubic B-spline kernels require multiplication. Note, however, that no element in either of
the two kernels is a number whose binary representation is more than two bits (e.g.,
$9_{10}=1001_2$, $36_{10}=100100_2$), therefore, multiplying by these values can be performed
using a two-input adder and shifting the inputs.

$$\begin{bmatrix}
1 & 1 \\
1 & 1 \\
\end{bmatrix} \times \begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
\end{bmatrix} \times \begin{bmatrix}
1 & 3 & 3 & 1 \\
3 & 9 & 9 & 3 \\
\end{bmatrix}$$

$$\frac{1}{4} \times \frac{1}{16} \times \frac{1}{64}$$

[Figure 7.2. Interpolation Kernels for 2 to 1 Magnification]

### 7.1.2 Pyramid Special Function Implementation

In this section we will derive a Pyramid special function operator to be used in a
functional memory decision table program. Figure 7.3 shows the calculation of the four
$A_{0,i,j}$ output pixels replacing the original $A_{0}$ input pixel using the Pyramid kernel.

We observe that since most of the pixel matrix elements are zero, the actual
computational effort for calculating the new pixel values is minimal. The upper left pixel
requires summing four values and shifting the result right two bits. The upper right and
lower left pixels require summing two values and shifting right one place.

The block diagram for a functional memory special function implementing this
example is shown in Figure 7.4. It accepts four neighboring pixels as input and provides
four output pixels. We assume eight bit gray-scaled pixels packed two per 16-bit word.
Figure 7.3. New Pixel Computation

Pairs of pixels are input horizontally (up_in and dn_in), two rows at a time. Each pixel is used in two convolution operations, first as a right pixel then as a left one. The function is designed with a shift input variable which when written, clocks either the lower or upper input byte into the lower byte of the staging registers that feed the adders. When shift is written with a zero, the upper staging register byte is written with the value in the lower byte and the lower staging register byte is written with the lower input register byte. When shift is written with a one, the upper staging register byte is also written with the value in the lower byte and the lower staging register byte is written with the upper input register byte.
The PALASM source listing of the Pyramid special function is contained in Appendix 4. When the Pyramid function is used in a program, the capital letters are replaced with the hexadecimal addresses of the variables in the order they appear in the function declaration. The function declaration begins with the “func” keyword followed by the eight character filename of the PALASM.DEF source file which defines the special function. In this example, “func pyramid” identifies “PYRAMID.DEF” as the special
function definition file. In the next section we describe in more detail the use of the Pyramid special function macro.

### 7.1.3 Magnification Program Example

A decision table for magnifying a 32 by 32 8-bit gray-scaled image into a 64 by 64 pixel image is shown Figure 7.5. The "func pyramid" declaration specifies the variable identifiers for use in the program. When the PALASM for the chip is generated, the contents of PYRAMID.DEF is included, except the capital letters in the file are replaced alphabetically by the hexadecimal addresses of the variables in the order they are listed in the declaration.

The input image is a 32 by 32 array of 8-bit pixels packed two per 16-bit word and contained in the array \( \text{in}[0..511] \). The magnified image is written as a 64 by 64 byte image in array \( \text{out}[0..2047] \). \( j \) serves as the input image and \( k \) is the output image index variables. We know that \( j \) will require at most nine bits and \( k \) will require no more than eleven bits for storage, so we can use the "register" data type to specify exactly these widths.

After initializing \( j \) and \( k \) at zero, the pixel pair contained at \( \text{in}[j] \) and the one just below it at \( \text{in}[j+16] \) are loaded into \( \text{up}_\text{in} \) and \( \text{dn}_\text{in} \), which are the \( A \) and \( B \) registers (respectively) of the Pyramid special function logic. Writing an even \( k \) value to \( \text{shift} \) moves the upper (even addressed) bytes of \( \text{up}_\text{in} \) and \( \text{dn}_\text{in} \) into their lower staging registers which feed the adders. The lower staging registers also get shifted into the upper positions when \( \text{shift} \) is written. Each time following a write to \( \text{shift} \), the results of the four convolutions can be copied into the output image array elements \( \text{out}[k] \) and the pair right below, \( \text{out}[k+32] \).
'magnify a 32x32 image to 64x64

func pyramid 'four pyramid convolutions in parallel
up_in, dn_in : register:16; 'input quad
up_2, dn_2 : expression:16; 'shift registers
up_out, dn_out : expression:16; 'output quad
shift : register:1; 'shift mechanism

var j : register:9; k : register:11; 'in, out array indexes
in : array[511] of integer; '16 across, 32 down
out : array[2047] of integer; '32 across, 64 down

dtbegin
lambda = 0 1 1 1 1
k = 2048-32 | - T F F F 'k = 2048?
k and 31 = 0 | - - T F F 'k mod 32 = 0?
k | - - - T F 'k odd?

| j := 0 | X - - - - | 'init input row
| k := 0 | X - - - - | 'init output row
| k := k + 32 | - - - X - | 'skip one output row
| j := j + 1 | - - - X - X | 'inc input row index
| up_in := in[j] | X - X - X 3 | 'get upper byte pair
| dn_in := in[j+16] | X - X - X 3 | 'get lower byte pair
| shift := k | X - X X X | 'shift even/odd k
| out[k] := up_out | X - X X X | 'store upper byte pair
| out[k+32] := dn_out | X - X X X | 'store lower pair
| k := k + 1 | X - X X X | 'next output quad
| exit | - X - - |
| lambda := | 1 - - - |
dtend

Figure 7.5. Decision Table for 2 to 1 Magnification

For each four input pixels that are loaded (into up_in and dn_in), eight output pixels are generated as pixel rows are scanned horizontally. Shifting takes place every time a quad of pixels are stored into the output array, however, loading from the input array takes place every other time, only when k is even. The fourth condition stub ("k") examines this, switching between the fourth rule for odd ks, which only shifts and outputs, and the fifth rule for even ks, which loads up_in and dn_in before shifting and outputting. The third condition stub ("k and 31 = 0") examines when an output row is complete. k must be incremented to skip one row because two rows are generated each
pass. The second condition stub ("k = 2048-32") tests when k has reached the end of the second to the last row, when the program terminates.

### 7.1.4 Magnification Execution Comparison

We chose a 64 by 64 output image because our prototype has limited RAM space. There is no practical reason why the same program couldn't be compiled for a 512 by 512 image output memory, except that the compiler would need to generate operators for 15- and 17-bit operands instead of the 9- and 11-bit ones as for this example. See Appendix 5 for the FMC compiler listing containing the memory map and the execution table for this program.

Rules 1 and 2 of the decision table are executed once. Rules 3, 4 and 5 execute the rest of the time, and each time, k is incremented exactly once and four output pixels are produced. A total of 64 x 64 = 4,096 output pixels are produced four at a time requiring 1,024 iterations, which is exactly the number of input pixels.

To compare the functional memory performance with a von Neumann processor, we can examine the output pixel computations shown back in Figure 7.3. Pixel A01,1 would take five steps to compute, A01,2 and A02,1 would take three steps, and A02,2 would take just one step, therefore, functional memory saves twelve steps for each input pixel. The disadvantage with functional memory is that twice as many bus transactions are needed to move data into up_in and dn_in and out of up_out and dn_out. Pixels are input two at a time, however, each input pixel must be input twice. Output pixels are output two at a time. The number of input memory transactions equals the number of input pixels and the number of output memory transactions equals half of the number of output pixels, which equals twice the number of input pixels. Therefore, with functional memory, we realize a performance savings of $12j^2$ steps at a cost of $j^2$ additional read and $2j^2$ additional store steps.
7.2 HISTOGRAMS

Computing histograms of black or white pixels is an important function in character recognition systems. Most commercially available OCR systems assume that lines of text can be separated by detecting horizontal lines of white space and characters can be isolated within each line by detecting vertical lines of white space [Leedham, 1991]. Characters are isolated by first taking horizontal histograms to separate the lines of text and then separating each line into characters by taking vertical histograms between two horizontal white lines.

7.2.1 Character Classification

Once characters have been isolated, histograms can be used further for recognition. Horizontal, vertical, left diagonal and right diagonal histograms of black pixels in a character can be used for statistical classification. Lettera, et al. [1986] describes a method whereby once the four histograms are obtained for an \( n \)-pixel width character, they are transformed into four eight element vectors roughly corresponding to the histogram of black pixels obtained using an interval width equal to \( n \)-eighths of the pixels. The four vectors are then normalized to give a set of 32 stochastic variables assumed as representative features of the character. These values can then be compared using statistical or fuzzy techniques with tables containing previously learned feature sets for possible characters.

When computing horizontal histograms, functional memory can be useful in eliminating the shifting that must take place when summing the number of ones in an image word. All bits of an image word can be summed in one step. When computing vertical histograms, auto-incrementing functional memory locations can be designed where each bit in the image word controls the incrementing of a separate counter,
therefore, up to as many counters as there are bits in the image word may be incremented simultaneously.

7.2.2 Row-Column Sum Special Function

Figure 7.6 illustrates how an adder circuit can be constructed to sum the number of ones (or zeros) in a 16-bit image word, which would be useful for computing horizontal histograms. The adder at the bottom is used to accumulate the sums for rows longer than 16-bits. The size of the register and the right input of the adder must be large enough to sum all the pixels of a row.

Figure 7.6. 16 Input 1-Bit Adder for Computing Histograms

Figure 7.7 illustrates how a set of auto-incrementing registers can be constructed so that loading one image word can increment as many counters as bits in the image word. With a 16-bit image word, this function can be used to compute vertical histograms 16 at a time. The size of the counters must large enough to sum all the pixels in a column.
When computing histograms across a large area, vertical and horizontal histograms can not be computed simultaneously, therefore, computing both requires two complete screen passes. With functional memory, however, no additional shift or add steps are needed to sum the "1" valued pixels. The disadvantage with functional memory is that twice as many memory transactions are required than with a von Neumann processor.

When horizontal and vertical histograms must be computed in a small area, as when performing character recognition, both the vertical and horizontal histograms can be computed simultaneously. This cuts in half the number of memory transaction required by the functional memory approach, making it even with the von Neumann implementation.

For our programming example, we combine both the 16-input one-bit adder and the 16 bit counters functions into one special function named RowColSum. The first address is where the image word is written. The next address contains the sum of the one-valued bits contained in the first address. The following 16 addresses are for reading the bit counters which are incremented when their respective bit locations equal a one when the
image word is written into the first address. The example in the next section uses
RowColSum to compute the row and column histograms for a 16 by 16 pixel character in
one pass.

7.2.3 Histogram Program Example

A decision table for computing the row and column histograms for a 16 by 16 pixel
character is shown Figure 7.8. The “func RowColSum” declaration specifies the variable
identifiers for use in the program. When the PALASM for the chip is generated, the
contents of ROWCOLSU.DEF are included, except the capital letters in the file are
replaced alphabetically by the hexadecimal addresses of the variables in the order they are
listed in the declaration.

```
func RowColSum 'Row sum/column accumulate function
    InWord : register:16; '16 pixel row input
    RowSum : expression:5; 'sum of row bits
    ColSums : array[15] of expression:5; 'column histogram
var j : register:4;
var char : array[15] of integer; 'character input array
var row : array[15] of integer; 'Row-wise histogram
dtbegin
lambda = | 0 1 1
j = 16 | - F T '16 image words
--------------------+------
ColSums := 0 | X - - 'Zero column sum array
j := 0 | X - - 'loop through 16 words
InWord := char[j] | X X - 'load function input
row[j] := RowSum | X X - 'store sum for this row
j := j + 1 | X X - 'increment row counter
exit | - - X
lambda := | 1 - -
dtend
```

Figure 7.8. Decision Table for Character Histogram Computation

When InWord is written, the sum of the “1” bits appear in RowSum, and ColSums[i]
is incremented if bit 15–i of InWord was written with a “1.” The function is also
designed so that when $ColSums[0]$ is written, all 16 $ColSums[0..15]$ counters are cleared.

The program begins by clearing the column sum array of counters and reading the first character word. The row sum output for the first word ($j=0$) is stored into the first $row$ array element which contains the 16 row histogram values when the program completes. When the program terminates, the $ColSums[0..15]$ array will contain the 16 column histogram values.

**7.2.4 Row Column Histogram Computation Comparison**

See Appendix 6 for the FMC compiler listing containing the memory map and the execution table for this program. From the decision table, we see that Rules 1 and 3 execute once and Rule 2 executes 15 times. To compare the functional memory performance with a von Neumann processor, we first estimate how many computation steps it would take to compute the sum of the “1” bits in a word. Shifting each bit into the carry and adding a constant zero plus the carry with each shift would sum all 16 bits in 32 steps. The $RowColSum$ function accomplishes this task in one step. Summing each bit to a separate accumulator also can be accomplished in three steps per bit by testing each bit and incrementing a separate counter only if its corresponding bit is set. A 16-bit image word could therefore be processed in 48 steps. Each image word processed on a von Neumann processor would therefore require 64 steps whereby the functional memory approach would require only one. Although the row and column histograms may be only a fraction of a character recognition analysis, the savings of 64 computational steps per image word by far out weighs the expense of additional loads and stores with functional memory.
CHAPTER 8. CONCLUSIONS

In this dissertation, we have described the design and implementation of the hardware and software of a new class of parallel processing computer systems based on the idea of functional memory. A functional memory computer makes use (1) of FPGAs, which enable the evaluation of expressions in combinational logic rather than in traditional von Neumann fashion, and (2) of a decision-table based programming language, which permits the separation of program control logic from other computation (and facilitates the specification of large functional/applicative transformations). It is the manner in which we combined these two concepts that led to the central contribution of our dissertation research, namely, the development of a functional memory computer system which is innovative, cost-effective for certain important applications, realizable using existing technology, and potentially of major future value using new technology.

We conclude by noting that the many of our design decisions were made so that we could implement our FMC using off-the-shelf hardware and software and relatively inexpensive (PC-class) technology. For example, our design reflects the capacity of the XILINX chips we had available to us. While our research may have been easier if we had assumed chips of indefinite capacity, or of supercomputers rather than PCs, our results would not have had the same short-term practical value. (In the long run, of course, chips of sufficiently greater size will become available.) In general, with newer, perhaps specially designed technology, there would be fewer restrictions. In any event, even generations from now, there will always be applications which test the limits of available technology, and the ways in which we addressed current limitations may very well be applicable to future ones.
8.1 DECISION TABLE COMPUTERS

Our programming model for the decision table is more powerful than for conventional programming languages because with decision tables, condition expressions are evaluated nondeterministically and multiway branches are not serialized. Their logical nature immediately suggests the use of a field programmable gate array, because logic equations are a FPGA's simplest form of programming. The capacities of FPGAs today are large enough for the condition stub processing and the rule selection all to be implemented in one FPGA. This allowed us quite easily to achieve our goal of constant time condition stub evaluation for realistic programs.

When a decision table executes on a FMC, rule selection consumes just one load cycle. All the condition stubs evaluate in parallel and the rule column is selected by the functional memory in one machine cycle. For the selected rule, the action stubs are executed in order by the processor. When the functional memory is also used to compute the right sides of the assignment statements, the processor activity reduces to a single variable move operation for each assignment statement to be executed. Execution step analyses for a FMC decision table programs involves little more than counting the action stubs for each rule executed. Our goal of designing a "decision table computer" has been realized.

8.2 CUSTOM COMPUTING MACHINES

The use of field programmable gate arrays (FPGAs) not only allowed us to achieve parallel expression computation, but we found our machine competitive in an area of computer architecture research called "custom computing machines" (CCMs). CCMs involve the use of FPGAs to provide customized hardware functions or dedicated processors for flexibility or performance enhancement. Several CCM projects are underway. Most are large scale custom coprocessors or processor emulators with
multiple parallel FPGAs, that replace what would otherwise be a custom VLSI
coprocessor or specialized hardware. Most projects have the goal of programming in a C
type language.

One of our application goals was to replace microprocessor functions with improved
FPGA combinational logic attached to the memory. For a functional memory custom
computing machine to be competitive in performance with today's high speed
microprocessors with internal caches, the functional memory and the minimal processor
must be implemented in the same chip (or wafer).

Portable image processing was found to be an application that had just the right
class characteristics that allowed it to take advantage of the beneficial features of a functional
memory custom computing machine. Functions involve small operands so multiple
operand additions and multiplications require minimal logic. Functions that would have
to be done iteratively on a standard microprocessor can be done in parallel with a single
expression implemented in the functional memory. Finally, since the functions are varied
and use different special operands, and functions are rarely, if ever, performed
simultaneously, reprogrammability would offer a substantial hardware savings.

Our future research will be devoted to improving the cost-effectiveness and usability
of our hardware and software functional memory computer system, and to developing
new classes of applications which can take advantage of this class of systems.
APPENDIXES. PROGRAM LISTINGS

APPENDIX 1. BUBBLE SORT COMPILER LISTING (BUBBLE.LIS)


File: c:\dissert\compiler\bubble\bubble.src

Input File

var n : integer;
  a : array[50] of integer;
  j, k : integer;
  "a[j]", "a[j+1]" : integer;

begin 'bubble sort
lambda = 0 1 1 1 1
k = l - T F F
j = k - - T F F
"a[j]" > "a[j+1]" - - T F
-------------------+----------
 k = n - X -
 j = 1 - X -
 a[j] := "a[j+1]" - - X -
 a[j+1] := "a[j]" - - - X -
 j := j + 1 - - X X
 "a[j]" := a[j] - X X X
 "a[j+1]" := a[j+1] - X X X
 exit - - X -
 lambda := 1 - - -
end. 'bubble sort

Compilation Statistics:
  5 rules, 4 conditions, 10 actions

Functional Memory: 124 bytes
FPGA I/O: 6 inputs, 5 outputs
Microcode: 55 lines MC
FPGA PALASM: 168 CLBs (estimated)
  1 chip

Functional Memory Map

<table>
<thead>
<tr>
<th>Name</th>
<th>*Type</th>
<th>Address or Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda</td>
<td>R</td>
<td>0000 2</td>
</tr>
<tr>
<td>@Rule</td>
<td>P</td>
<td>0002 7</td>
</tr>
<tr>
<td>n</td>
<td>R</td>
<td>0004 8</td>
</tr>
<tr>
<td>a</td>
<td>C</td>
<td>0006 0</td>
</tr>
</tbody>
</table>

165
<table>
<thead>
<tr>
<th>Rule 1</th>
<th>/L1*/L0</th>
</tr>
</thead>
<tbody>
<tr>
<td>004:</td>
<td>k := n</td>
</tr>
<tr>
<td>00C:</td>
<td>j := 1</td>
</tr>
<tr>
<td>014:</td>
<td>&quot;a[j]&quot; := a[j]</td>
</tr>
<tr>
<td>020:</td>
<td>&quot;a[j+1]&quot; := a[j+1]</td>
</tr>
<tr>
<td>02C:</td>
<td>lambda :=</td>
</tr>
<tr>
<td>034:</td>
<td>goto @rule</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 2</th>
<th>/L1*/L0*/C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>03C:</td>
<td>exit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 3</th>
<th>/L1<em>L0</em>/C2*C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>044:</td>
<td>j := 1</td>
</tr>
<tr>
<td>04C:</td>
<td>k := k-1</td>
</tr>
<tr>
<td>054:</td>
<td>&quot;a[j]&quot; := a[j]</td>
</tr>
<tr>
<td>060:</td>
<td>&quot;a[j+1]&quot; := a[j+1]</td>
</tr>
<tr>
<td>06C:</td>
<td>goto @rule</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 4</th>
<th>/L1<em>L0</em>/C2*/C3*C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>074:</td>
<td>a[j] := &quot;a[j+1]&quot;</td>
</tr>
<tr>
<td>080:</td>
<td>a[j+1] := &quot;a[j]&quot;</td>
</tr>
<tr>
<td>08C:</td>
<td>j := j+1</td>
</tr>
<tr>
<td>094:</td>
<td>&quot;a[j]&quot; := a[j]</td>
</tr>
<tr>
<td>0A0:</td>
<td>&quot;a[j+1]&quot; := a[j+1]</td>
</tr>
<tr>
<td>0AC:</td>
<td>goto @rule</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 5</th>
<th>/L1<em>L0</em>/C2*/C3*/C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B4:</td>
<td>j := j+1</td>
</tr>
<tr>
<td>0BC:</td>
<td>&quot;a[j]&quot; := a[j]</td>
</tr>
<tr>
<td>0C8:</td>
<td>&quot;a[j+1]&quot; := a[j+1]</td>
</tr>
<tr>
<td>0D4:</td>
<td>goto @rule</td>
</tr>
<tr>
<td>0DC:</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX 2. BUBBLE SORT MINIMAL PROCESSOR CODE
(BUBBLE.ASM)

.TITLE BUBBLE 07-05-1994 at 12:36:12
;
; MICROINSTRUCTION OPCODES
LDC EQU 000001008  ; DOR <-- constant
LDA EQU 000101008  ; DOR <-- (address)
LDM EQU 001101008  ; DOR <-- (MAR) used LDM, 0
LMA EQU 000110008  ; MAR <-- (address)
WMD EQU 001000008  ; (MAR) <-- DOR
WAD EQU 110000008  ; (address) <-- DOR
WMCE EQU 101000008 ; (MAR) <-- constant
JPI EQU 000111008  ; microPC <-- (address)
HALT EQU 000011018 ; DONE used HALT, $8
NOP EQU 000000008  ; DELAY ONE CYCLE

ORG 0000H
DW NOP, 0

Rule1:  ; 04H
; DD k:=n
DW LDA, 00004H
DW WAD, 0006EH
; DC j:=1
DW LDC, 00001H
DW WAD, 0006CH
; DI "a[j]" :=a[j]
DW LMA, 00076H
DW LDM, 0
DW WAD, 00070H
; DI "a[j+1]" :=a[j+1]
DW LMA, 00078H
DW LDM, 0
DW WAD, 00072H
; DC lambda:=1
DW LDC, 1
DW WAD, 00000H
; Jump to next rule
DW JPI, 02H
DW NOP, 0

Rule2:  ; 03CH
; EX exit
DW HALT, $8
DW HALT, $4

Rule3:  ; 044H
; DC j:=1
DW LDC, 00001H
DW WAD, 0006CH
; DE k:=k-1
DW LDA, 00074H
DW WAD, 0006EH
; DI "a[j]" :=a[j]
DW LMA, 00076H
DW LDM, 0
DW WAD, 00070H
; DI "a[j+1]" := a[j+1]
DW LMA, 00078H
DW LDM, 0
DW WAD, 00072H
; Jump to next rule
DW JPI, 02H
DW NOP, 0

Rule4: 074H
; ID a[j] := a[j+1]
DW LMA, 00076H
DW LDA, 00072H
DW WMD, 0
; ID a[j+1] := a[j]
DW LMA, 00078H
DW LDA, 00070H
DW WMD, 0
; DE j := j + 1
DW LDA, 0007AH
DW WAD, 0006CH
; DI "a[j]" := a[j]
DW LMA, 00076H
DW LDM, 0
DW WAD, 00070H
; DI "a[j+1]" := a[j+1]
DW LMA, 00078H
DW LDM, 0
DW WAD, 00072H
; Jump to next rule
DW JPI, 02H
DW NOP, 0

Rule5: 0B4H
; DE j := j + 1
DW LDA, 0007AH
DW WAD, 0006CH
; DI "a[j]" := a[j]
DW LMA, 00076H
DW LDM, 0
DW WAD, 00070H
; DI "a[j+1]" := a[j+1]
DW LMA, 00078H
DW LDM, 0
DW WAD, 00072H
; Jump to next rule
DW JPI, 02H
DW NOP, 0
END
APPENDIX 3. BUBBLE SORT PALASM SOURCE CODE (1BUBBLE.PDS)

TITLE  BUBBLE
AUTHOR  Mr. D. T. Compiler
DATE  07-05-1994 at 12:36:12
CHIP  1BUBBLE LCA

;ADDRESS INPUT PINS (18)
A1  A2  A3  A4  A5  A6  A7  A8  A9  A10  A11  A12  A13  A14  A15
RDC  WRIC  WRHC ;READ, WRITE LOW BYTE, WRITE HIGH BYTE

;DATA INPUT PINS (18)
DIO  DI1  DI2  DI3  DI4  DI5  DI6  DI7  DI8  DI9  DI10  DI11  DI12  DI13  DI14  DI15

CHIP OUTPUT PINS (17)
;Note that these data output pins are active low
DO0  DO1  DO2  DO3  DO4  DO5  DO6  DO7  DO8  DO9  DO10  DO11  DO12  DO13  DO14  DO15

;DAISY-CHAIN OUTPUT ENABLE
DOEIN ;DATA OUTPUT ENABLE (ACTIVE HIGH)
FROM PREVIOUS CHIP
DOE ;DATA OUTPUT ENABLE (ACTIVE HIGH) TO NEXT CHIP

EQUATIONS
;compute condition stub C2 = (k = 1 )
e_03_ne0_1  =  ((reg_06E_0)/reg_06E_0)+(reg_06E_1)/reg_06E_1)
e_03_ne2_3  =  ((reg_06E_2)/reg_06E_2)+(reg_06E_3)/reg_06E_3)
e_03_ne4_5  =  ((reg_06E_4)/reg_06E_4)+(reg_06E_5)/reg_06E_5)
e_03_ne6_7  =  ((reg_06E_6)/reg_06E_6)+(reg_06E_7)/reg_06E_7)
e_03_0   =  ((reg_06E_0)/reg_06E_0)+(reg_06E_1)/reg_06E_1)+((reg_06E_2)/reg_06E_2)+(reg_06E_3)/reg_06E_3)+((reg_06E_4)/reg_06E_4)+(reg_06E_5)/reg_06E_5)+(reg_06E_6)/reg_06E_6)+(reg_06E_7)/reg_06E_7)
C2  =  e_03_0

;compute condition stub C3 = (j = k )
e_05_ne0_1  =  ((reg_06C_0)/reg_06C_0)+(reg_06C_1)/reg_06C_1)
e_05_ne2_3  =  ((reg_06C_2)/reg_06C_2)+(reg_06C_3)/reg_06C_3)
e_05_ne4_5  =  ((reg_06C_4)/reg_06C_4)+(reg_06C_5)/reg_06C_5)
e_05_ne6_7  =  ((reg_06C_6)/reg_06C_6)+(reg_06C_7)/reg_06C_7)
e_05_0   =  ((reg_06C_0)/reg_06C_0)+(reg_06C_1)/reg_06C_1)+((reg_06C_2)/reg_06C_2)+(reg_06C_3)/reg_06C_3)+((reg_06C_4)/reg_06C_4)+(reg_06C_5)/reg_06C_5)+(reg_06C_6)/reg_06C_6)+(reg_06C_7)/reg_06C_7)
C3  =  e_05_0

;compute condition stub C4 = ("a[ij]" > "a[ij+1]"

e_07c2  =  ((reg_070_1)/reg_072_1)+(reg_070_1)/reg_072_3)+(reg_070_1)/reg_072_5)+(reg_070_1)/reg_072_7)
e_07c2  =  ((reg_070_1)/reg_072_1)+(reg_070_1)/reg_072_3)+(reg_070_1)/reg_072_5)+(reg_070_1)/reg_072_7)
e_07c2  =  ((reg_070_1)/reg_072_1)+(reg_070_1)/reg_072_3)+(reg_070_1)/reg_072_5)+(reg_070_1)/reg_072_7)
e_07c2  =  ((reg_070_1)/reg_072_1)+(reg_070_1)/reg_072_3)+(reg_070_1)/reg_072_5)+(reg_070_1)/reg_072_7)
e_07c2  =  ((reg_070_1)/reg_072_1)+(reg_070_1)/reg_072_3)+(reg_070_1)/reg_072_5)+(reg_070_1)/reg_072_7)
C4  =  e_07c2

;compute basic from L0, L1, C2, C3, C4
Rule1  =  /L1/*L0 ;Starting at Microprogram Address 004
Rule2  =  /L1/*L0*C2 ;Starting at Microprogram Address 03C
Rule3  =  /L1/*L0*/C2*C3 ;Starting at Microprogram Address 04C
Rule4  =  /L1/*L0*/C2*C3*C4 ;Starting at Microprogram Address 074
Rule5  =  /L1/*L0*/C2*/C3*/C4 ;Starting at Microprogram Address 084

rule_2  =  Rule1 + Rule2 + Rule3 + Rule4 + Rule5
rule_3  =  Rule2
rule_4  =  Rule2 + Rule4 + Rule5
rule_5  =  Rule2 + Rule4 + Rule5
rule_6  =  Rule3 + Rule4
rule_7  =  Rule5

169
\[
\begin{align*}
&L_1 := \text{reg}_\text{OGC} - \text{O.CE} = \text{sel} \text{ OGC} ; \text{Clock Enable} \\
&L_0.\text{CE} = \text{sel} \text{ O0} ; \text{Clock Enable} \\
&L_0.\text{CLKF} = \text{WRLC} ; \text{Write Clock} \\
&\text{Input Registers and Constants} \\
&\text{Input bit: G} \\
&s_{-015} = \ldots \\
&(\text{reg}_\text{OGC} - 1) := \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \\
&(\text{reg}_\text{OGC} - 2) := \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \\
&(\text{reg}_\text{OGC} - 3) := \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \\
&(\text{reg}_\text{OGC} - 4) := \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \\
&(\text{reg}_\text{OGC} - 5) := \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \\
&(\text{reg}_\text{OGC} - 6) := \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \\
&(\text{reg}_\text{OGC} - 7) := \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \odot \text{reg}_\text{OGC} \\
&\text{Input bit: \ldots} \\
\end{align*}
\]
reg_06C 2.CLFK = WRLC ;Write Clock
reg_06C 2.CE = sel_06C ;Clock Enable
reg_06C 3 := D13 ;
j input bit 3
reg_06C 3.CLFK = WRLC ;Write Clock
reg_06C 3.CE = sel_06C ;Clock Enable
reg_06C 4 := D14 ;
j input bit 4
reg_06C 4.CLFK = WRLC ;Write Clock
reg_06C 4.CE = sel_06C ;Clock Enable
reg_06C 5 := D15 ;
j input bit 5
reg_06C 5.CLFK = WRLC ;Write Clock
reg_06C 5.CE = sel_06C ;Clock Enable
reg_06C 6 := D16 ;
j input bit 6
reg_06C 6.CLFK = WRLC ;Write Clock
reg_06C 6.CE = sel_06C ;Clock Enable
reg_06C 7 := D17 ;
j input bit 7
reg_06C 7.CLFK = WRLC ;Write Clock
reg_06C 7.CE = sel_06C ;Clock Enable
reg_06E 0 := D10 ;
k input bit 0
reg_06E 0.CLFK = WRLC ;Write Clock
reg_06E 0.CE = sel_06E ;Clock Enable
reg_06E 1 := D11 ;
k input bit 1
reg_06E 1.CLFK = WRLC ;Write Clock
reg_06E 1.CE = sel_06E ;Clock Enable
reg_06E 2 := D12 ;
k input bit 2
reg_06E 2.CLFK = WRLC ;Write Clock
reg_06E 2.CE = sel_06E ;Clock Enable
reg_06E 3 := D13 ;
k input bit 3
reg_06E 3.CLFK = WRLC ;Write Clock
reg_06E 3.CE = sel_06E ;Clock Enable
reg_06E 4 := D14 ;
k input bit 4
reg_06E 4.CLFK = WRLC ;Write Clock
reg_06E 4.CE = sel_06E ;Clock Enable
reg_06E 5 := D15 ;
k input bit 5
reg_06E 5.CLFK = WRLC ;Write Clock
reg_06E 5.CE = sel_06E ;Clock Enable
reg_06E 6 := D16 ;
k input bit 6
reg_06E 6.CLFK = WRLC ;Write Clock
reg_06E 6.CE = sel_06E ;Clock Enable
reg_06E 7 := D17 ;
k input bit 7
reg_06E 7.CLFK = WRLC ;Write Clock
reg_06E 7.CE = sel_06E ;Clock Enable
reg_070 0 := D10 ;
"a[j]" input bit 0
reg_070 0.CLFK = WRLC ;Write Clock
reg_070 0.CE = sel_070 ;Clock Enable
reg_070 1 := D11 ;
"a[j]" input bit 1
reg_070 1.CLFK = WRLC ;Write Clock
reg_070 1.CE = sel_070 ;Clock Enable
reg_070 2 := D12 ;
"a[j]" input bit 2
reg_070 2.CLFK = WRLC ;Write Clock
reg_070 2.CE = sel_070 ;Clock Enable
reg_070 3 := D13 ;
"a[j]" input bit 3
reg_070 3.CLFK = WRLC ;Write Clock
reg_070 3.CE = sel_070 ;Clock Enable
reg_070 4 := D14 ;
"a[j]" input bit 4
reg_070 4.CLFK = WRLC ;Write Clock
reg_070 4.CE = sel_070 ;Clock Enable
reg_070 5 := D15 ;
"a[j]" input bit 5
reg_070 5.CLFK = WRLC ;Write Clock
reg_070 5.CE = sel_070 ;Clock Enable
reg_070 6 := D16 ;
"a[j]" input bit 6
reg_070 6.CLFK = WRLC ;Write Clock
reg_070 6.CE = sel_070 ;Clock Enable
reg_070 7 := D17 ;
"a[j]" input bit 7
reg_070 7.CLFK = WRLC ;Write Clock
reg_070 7.CE = sel_070 ;Clock Enable
reg_072 0 := D10 ;
"a[j+1]" input bit 0
reg_072 0.CLFK = WRLC ;Write Clock
reg_072 0.CE = sel_072 ;Clock Enable
reg_072 1 := D11 ;
"a[j+1]" input bit 1
reg_072 1.CLFK = WRLC ;Write Clock
reg_072 1.CE = sel_072 ;Clock Enable
reg_072 2 := D12 ;
"a[j+1]" input bit 2
reg_072 2.CLFK = WRLC ;Write Clock
reg_072 2.CE = sel_072 ;Clock Enable
reg_072 3 := D13 ;
"a[j+1]" input bit 3
reg_072 3.CLFK = WRLC ;Write Clock
reg_072 3.CE = sel_072 ;Clock Enable
reg 072.3_CE = sel 072 ; Clock Enable
reg 072.4 := D14 ;
reg 072.4.CLKF = WRLC ; Write Clock
reg 072.4_CE = sel 072 ; Clock Enable
reg 072.5 := D15 ;
reg 072.5.CLKF = WRLC ; Write Clock
reg 072.5_CE = sel 072 ; Clock Enable
reg 072.6 := D16 ;
reg 072.6.CLKF = WRLC ; Write Clock
reg 072.6_CE = sel 072 ; Clock Enable
reg 072.7 := D17 ;
reg 072.7.CLKF = WRLC ; Write Clock
reg 072.7_CE = sel 072 ; Clock Enable

; OE input Multiplexers
D001 = s_09 0^*sel_074 + a_0D 0^*sel_076 ; k-l, @a[j]
D003 = a_013 0^*sel_078 + s_015 0^*sel_07A ; @a[j+1], j+1
D005 = D001 + D003
D00 = (/RDC*(D00a1))
D001 = s_09 1^*sel_074 + a_0D 1^*sel_076 ; k-l, @a[j]
D003 = a_013 1^*sel_078 + s_015 1^*sel_07A ; @a[j+1], j+1
D005 = D001 + D003
D01 = (/RDC*(D01a1))
D02t = rule 2^*sel_02 + s_09 2^*sel_074 ; @Rule, k-l
D02t = a_0D 2^*sel_076 + a_013 2^*sel_078 ; @a[j], @a[j+1]
D025 = s_015 2^*sel_07A ; j+1
D02a = D02t + D023 + D025
D02 = (/RDC*(D02a1))
D031 = rule 3^*sel_02 + s_09 3^*sel_074 ; @Rule, k-l
D033 = a_0D 3^*sel_076 + a_013 3^*sel_078 ; @a[j], @a[j+1]
D035 = s_015 3^*sel_07A ; j+1
D03a = D031 + D033 + D035
D03 = (/RDC*(D03a1))
D04t = rule 4^*sel_02 + s_09 4^*sel_074 ; @Rule, k-l
D04t = a_0D 4^*sel_076 + a_013 4^*sel_078 ; @a[j], @a[j+1]
D045 = s_015 4^*sel_07A ; j+1
D04a = D04t + D043 + D045
D04 = (/RDC*(D04a1))
D051 = rule 5^*sel_02 + s_09 5^*sel_074 ; @Rule, k-l
D053 = a_0D 5^*sel_076 + a_013 5^*sel_078 ; @a[j], @a[j+1]
D055 = s_015 5^*sel_07A ; j+1
D05a = D051 + D053 + D055
D05 = (/RDC*(D05a1))
D06t = rule 6^*sel_02 + s_09 6^*sel_074 ; @Rule, k-l
D06t = a_0D 6^*sel_076 + a_013 6^*sel_078 ; @a[j], @a[j+1]
D065 = s_015 6^*sel_07A ; j+1
D06a = D06t + D063 + D065
D06 = (/RDC*(D06a1))
D07t = rule 7^*sel_02 + s_09 7^*sel_074 ; @Rule, k-l
D07t = a_0D 7^*sel_076 + a_013 7^*sel_078 ; @a[j], @a[j+1]
D075 = s_015 7^*sel_07A ; j+1
D07a = D07t + D073 + D075
D07 = (/RDC*(D07a1))
D08t = rule 8^*sel_02 + s_09 8^*sel_074 ; @Rule, k-l
D08t = a_0D 8^*sel_076 + a_013 8^*sel_078 ; @a[j], @a[j+1]
D08a = D08t
D08 = (/RDC*(D08a1))

; Address Select Logic
sel 00 = ASX0^*ASHO^*ASMO^*ASLO ; Select for lambda input
sel 02 = ASX0^*ASHO^*ASMO^*ASL2 ; Select for @Rule output
sel 06C = ASX0^*ASHO^*AS6^*ASLC ; Select for j input
sel 06E = ASX0^*ASHO^*AS6^*ASLE ; Select for k input
sel 070 = ASX0^*ASHO^*AS7^*ASL0 ; Select for "a[j]" input
sel 072 = ASX0^*ASHO^*AS7^*ASL2 ; Select for "a[j+1]" input
sel 074 = ASX0^*ASHO^*AS7^*ASL4 ; Select for k-l output
sel 076 = ASX0^*ASHO^*AS7^*ASL6 ; Select for @a[j] output
sel 078 = ASX0^*ASHO^*AS7^*ASLA ; Select for @a[j+1] output
sel 07A = ASX0^*ASHO^*AS7^*ASLA ; Select for j+1 output

ASX0 = /A12*/A13*/A14*/A15 ; Highest 4HO
ASH0 = /A8*/A9*/A10*/A11 ; High 4HO
ASMO = /A4*/A5*/A6*/A7 ; Middle 4HO
ASL0 = /A1*/A2*/A3 ; Low 4HO
ASL2 = A1*/A2*/A3 ; Low 4HO
ASM6 = /A4*/A5*/A6*/A7 ; Middle 4HO
ASLC = /A1*/A2*/A3 ; Low 4HC
ASLE = A1*/A2*/A3 ; Low 4HC
ASM7 = A4*/A5*/A6*/A7 ; Middle 4HC
ASL4 = /A1*/A2*/A3 ; Low 4H4
ASL6 = A1*A2*/A3 ; Lw 6i6
ASL8 = /A1*/A2*A3 ; Lw 6H8
ASLA = A1*/A2*A3 ; Lw 6HA
**APENDIX 4. PYRAMID FUNCTION MACRO (PYRAMID.DEF)**

92 estimated CLBs (note: upper case letters are replaced when expanded)

<table>
<thead>
<tr>
<th>Image magnification macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit pixels are packed two per 16-bit word, loading a 2x2 matrix of</td>
</tr>
<tr>
<td>input pixels produces two horizontal 2x2 matrixes of output pixels</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>func pyramid 'magnify a 32x32 image to 64x64</td>
</tr>
<tr>
<td>#up in, #dn in : register:16; 'reg A, reg B</td>
</tr>
<tr>
<td>#up 2, #dn 2 : expression:16; 'exp C, exp D</td>
</tr>
<tr>
<td>#up out, #dn out : expression:16; 'exp E, exp F</td>
</tr>
<tr>
<td>#shift : register:1; 'reg G</td>
</tr>
</tbody>
</table>

| (FD)exp F 15:exp F 8 = (exp C 15:exp C 8 + exp D 15:exp D 8) div2^0 |
| (E1)exp E 7:exp E 6 = (exp C 7:exp C 6 + exp D 7:exp D 6) div2^1 |
| (E0)exp E 15:exp E 8 = (exp F 15:exp F 8 + exp E 7:exp E 6) div2^1 |
| (F1)exp F 7:exp F 6 = exp D 7:exp D 6 |

| Writing a 1 to #reg G shifts registers as follows |
| exp C 15:exp C 8 = reg A 15:reg A 8 |
| exp D 15:exp D 8 = reg B 15:reg B 8 |

| Writing a 2 to #reg G shifts registers as follows |
| exp C 15:exp C 8 = reg A 7:reg A 0 |
| exp D 15:exp D 8 = reg B 7:reg B 0 |

| ====
| t_E 15 = exp F 8 = (exp C 15:exp C 8 + exp D 15:exp D 8) div2^0 |
| t_E 8 = exp C 9:exp D 9 = exp C 8:exp D 8 |

| ====
| t_E 10 = exp C 9:exp D 9 = (exp C 9:exp D 9 + (exp C 8:exp D 8)) |
| t_B 9 = exp C 10:exp D 10 = exp C 8:exp D 8 |

| ====
| t_E 10 = exp C 11 = exp D 11 = (exp C 10*exp D 10 + (exp C 8:exp D 8)) |
| t_B 10 = exp C 11 = exp D 11 = (exp C 10*exp D 10 + (exp C 8:exp D 8)) |

| ====
| t_E 12:0 = exp C 12:0 = (exp C 11:exp D 11 + (exp C 10:exp D 10)) |
| t_E 12:1 = t_E 12:0 = (exp C 11:exp D 11 + (exp C 10:exp D 10)) |
| t_E 12:2 = t_E 12:1 = (exp C 11:exp D 11 + (exp C 10:exp D 10)) |

| ====
| t_E 13:0 = exp C 13:0 = (exp C 12:exp D 12) |
| t_E 13:1 = exp C 13:0 = (exp C 12:exp D 12) |
| t_E 13:2 = exp C 13:1 = (exp C 12:exp D 12) |

| ====
| t_E 14:0 = exp C 13:0 = (exp C 12:exp D 12) |
| t_E 14:1 = exp C 13:0 = (exp C 12:exp D 12) |
| t_E 14:2 = exp C 13:1 = (exp C 12:exp D 12) |

| ====
| t_E 15 = exp C 15:exp D 15 = (exp C 14:exp D 14) |

| ====
| exp E 7:exp E 0 = (exp C 7:exp D 7 | exp C 8:exp D 8) div2^0 |
| exp E 0 = exp C 1:exp D 1 = exp C 0:exp D 0 |
| exp E 1 = exp C 2:exp D 2 = exp C 1:exp D 1 |

| ====
| exp E 2 = exp C 3:exp D 3 = exp C 2:exp D 2 |
| exp E 3 = exp C 4:exp D 4 = exp C 3:exp D 3 |

| ====
| exp E 4:exp E 0 = (exp C 7:exp D 7 | exp E 1:exp E 2) |

175
```c
exp_Ec5_4.0 = exp_C_5 * exp_D_5 * (exp_C_4 * exp_D_4)
exp_Ec5_4.1 = exp_C_5 * exp_D_5 * (exp_C_4 * exp_D_4)
exp_Ec5_4.2 = exp_C_5 * exp_D_5 * (exp_C_4 * exp_D_4)
exp_Ec5_4.3 = exp_C_5 * exp_D_5 * (exp_C_4 * exp_D_4)
exp_Ec5_4.4 = exp_C_5 * exp_D_5 * (exp_C_4 * exp_D_4)

exp_Fc12_10.0 = exp_D_10 * exp_D_10 * exp_Fc10
exp_Fc12_10.1 = exp_D_10 * exp_D_10 * exp_Fc10
exp_Fc12_10.2 = exp_D_10 * exp_D_10 * exp_Fc10
exp_Fc12_10.3 = exp_D_10 * exp_D_10 * exp_Fc10
exp_Fc12_10.4 = exp_D_10 * exp_D_10 * exp_Fc10

exp_Fc13_12.0 = exp_D_12 * exp_D_12 * exp_Fc12
exp_Fc13_12.1 = exp_D_12 * exp_D_12 * exp_Fc12
exp_Fc13_12.2 = exp_D_12 * exp_D_12 * exp_Fc12
exp_Fc13_12.3 = exp_D_12 * exp_D_12 * exp_Fc12

exp_Fc14_12.0 = exp_D_12 * exp_D_12 * exp_Fc12
exp_Fc14_12.1 = exp_D_12 * exp_D_12 * exp_Fc12
exp_Fc14_12.2 = exp_D_12 * exp_D_12 * exp_Fc12
exp_Fc14_12.3 = exp_D_12 * exp_D_12 * exp_Fc12

exp_Fc15 = exp_D_15 * exp_D_15 * exp_Fc15
exp_Fc15_1 = exp_D_15 * exp_D_15 * exp_Fc15
```

176
\begin{verbatim}
;exp F_7:exp_F_0 = exp_D_7:exp_D_0

exp F_0 = exp_D_0
exp F_1 = exp_D_1
exp F_2 = exp_D_2
exp F_3 = exp_D_3
exp F_4 = exp_D_4
exp F_5 = exp_D_5
exp F_6 = exp_D_6
exp F_7 = exp_D_7

;writing to reg_G shifts: exp_C_15:exp_C_8 := exp_C_7:exp_C_0

exp_C_15 := exp_C_7
exp_C_15.clkf = wrlc
exp_C_15.ce = sel_G
;whenever address G gets written

exp_C_14 := exp_C_6
exp_C_14.clkf = wrlc
exp_C_14.ce = sel_G
;whenever address G gets written

exp_C_13 := exp_C_5
exp_C_13.clkf = wrlc
exp_C_13.ce = sel_G
;whenever address G gets written

exp_C_12 := exp_C_4
exp_C_12.clkf = wrlc
exp_C_12.ce = sel_G
;whenever address G gets written

exp_C_11 := exp_C_3
exp_C_11.clkf = wrlc
exp_C_11.ce = sel_G
;whenever address G gets written

exp_C_10 := exp_C_2
exp_C_10.clkf = wrlc
exp_C_10.ce = sel_G
;whenever address G gets written

exp_C_9 := exp_C_1
exp_C_9.clkf = wrlc
exp_C_9.ce = sel_G
;whenever address G gets written

exp_C_8 := exp_C_0
exp_C_8.clkf = wrlc
exp_C_8.ce = sel_G
;whenever address G gets written

;writing to reg_G shifts: exp_D_15:exp_D_8 := exp_D_7:exp_D_0

exp_D_15 := exp_D_7
exp_D_15.clkf = wrlc
exp_D_15.ce = sel_G
;whenever address G gets written

exp_D_14 := exp_D_6
exp_D_14.clkf = wrlc
exp_D_14.ce = sel_G
;whenever address G gets written

exp_D_13 := exp_D_5
exp_D_13.clkf = wrlc
exp_D_13.ce = sel_G
;whenever address G gets written

exp_D_12 := exp_D_4
exp_D_12.clkf = wrlc
exp_D_12.ce = sel_G
;whenever address G gets written

exp_D_11 := exp_D_3
exp_D_11.clkf = wrlc
exp_D_11.ce = sel_G
;whenever address G gets written

exp_D_10 := exp_D_2
exp_D_10.clkf = wrlc
exp_D_10.ce = sel_G
;whenever address G gets written

exp_D_9 := exp_D_1
exp_D_9.clkf = wrlc
exp_D_9.ce = sel_G
;whenever address G gets written
\end{verbatim}
exp_D.9.ce = sel_G  ; whenever address G gets written
exp_D.8 := exp_D.0  ; upper byte bit 8 gets lower byte bit C
exp_D.8.clkf = wrlc  ; write clock
exp_D.8.ce = sel_G  ; whenever address G gets written

;;;;;;

;;;;;writing a 0 to reg G shifts:  exp_C.7:exp_C.0 := reg_A.15:reg_A.8
;;;;;writing a 1 to reg G shifts:  exp_C.7:exp_C.0 := reg_A.7:reg_A.0

exp_C.7 := /di0*reg_A.15 + di0*reg_A.7  ; r15 if /di0=1 + e7 if di0=1
exp_C.7.clkf = wrlc  ; write clock
exp_C.7.ce = sel_G  ; whenever address G gets written

exp_C.6 := /di0*reg_A.14 + di0*reg_A.6  ; r14 if /di0=1 + e6 if di0=1
exp_C.6.clkf = wrlc  ; write clock
exp_C.6.ce = sel_G  ; whenever address G gets written

exp_C.5 := /di0*reg_A.13 + di0*reg_A.5  ; r13 if /di0=1 + e5 if di0=1
exp_C.5.clkf = wrlc  ; write clock
exp_C.5.ce = sel_G  ; whenever address G gets written

exp_C.4 := /di0*reg_A.12 + di0*reg_A.4  ; r12 if /di0=1 + e4 if di0=1
exp_C.4.clkf = wrlc  ; write clock
exp_C.4.ce = sel_G  ; whenever address G gets written

exp_C.3 := /di0*reg_A.11 + di0*reg_A.3  ; r11 if /di0=1 + e3 if di0=1
exp_C.3.clkf = wrlc  ; write clock
exp_C.3.ce = sel_G  ; whenever address G gets written

exp_C.2 := /di0*reg_A.10 + di0*reg_A.2  ; r10 if /di0=1 + e2 if di0=1
exp_C.2.clkf = wrlc  ; write clock
exp_C.2.ce = sel_G  ; whenever address G gets written

exp_C.1 := /di0*reg_A.9 + di0*reg_A.1  ; r9 if /di0=1 + e1 if di0=1
exp_C.1.clkf = wrlc  ; write clock
exp_C.1.ce = sel_G  ; whenever address G gets written

exp_C.0 := /di0*reg_A.8 + di0*reg_A.0  ; r8 if /di0=1 + e0 if di0=1
exp_C.0.clkf = wrlc  ; write clock
exp_C.0.ce = sel_G  ; whenever address G gets written

;;;;;;

;;;;;writing a 1 to reg G shifts:  exp_D.7:exp_D.0 := reg_B.15:reg_B.8
;;;;;writing a 2 to reg G shifts:  exp_D.7:exp_D.0 := reg_B.7:reg_B.0

exp_D.7 := /di0*reg_B.15 + di0*reg_B.7  ; r15 if /di0=1 + e7 if di0=1
exp_D.7.clkf = wrlc  ; write clock
exp_D.7.ce = sel_G  ; whenever address G gets written

exp_D.6 := /di0*reg_B.14 + di0*reg_B.6  ; r14 if /di0=1 + e6 if di0=1
exp_D.6.clkf = wrlc  ; write clock
exp_D.6.ce = sel_G  ; whenever address G gets written

exp_D.5 := /di0*reg_B.13 + di0*reg_B.5  ; r13 if /di0=1 + e5 if di0=1
exp_D.5.clkf = wrlc  ; write clock
exp_D.5.ce = sel_G  ; whenever address G gets written

exp_D.4 := /di0*reg_B.12 + di0*reg_B.4  ; r12 if /di0=1 + e4 if di0=1
exp_D.4.clkf = wrlc  ; write clock
exp_D.4.ce = sel_G  ; whenever address G gets written

exp_D.3 := /di0*reg_B.11 + di0*reg_B.3  ; r11 if /di0=1 + e3 if di0=1
exp_D.3.clkf = wrlc  ; write clock
exp_D.3.ce = sel_G  ; whenever address G gets written

exp_D.2 := /di0*reg_B.10 + di0*reg_B.2  ; r10 if /di0=1 + e2 if di0=1
exp_D.2.clkf = wrlc  ; write clock
exp_D.2.ce = sel_G  ; whenever address G gets written

exp_D.1 := /di0*reg_B.9 + di0*reg_B.1  ; r9 if /di0=1 + e1 if di0=1
exp_D.1.clkf = wrlc  ; write clock
exp_D.1.ce = sel_G  ; whenever address G gets written

178
exp_D_1.ce = sel_G        ; whenever address G gets written

exp_D_0 := /di0*reg_B + di0*reg_B_D  ; if /di0=1 + e0 if e0=1
exp_D_0.clkh = wrih  ; write clock
exp_D_0.ce = sel_G      ; whenever address G gets written

; end of magnify function definition
; reg_A, reg_B, exp_C, exp_D, exp_E, exp_F, reg_G
;
APPENDIX 5. MAGNIFICATION EXAMPLE (MAGNIFY.LIS)


File: c:\dissert\compiler\convolve\magnify.src

Input File

func pyramid  'magnify a 32x32 image to 64x64
    up_in, dn_in : register:16;  'input quad
    up_2, dn_2 : expression:16;  'shift registers
    up_out, dn_out : expression:16;  'output quad
    shift : register:1;  'shift mechanism
var j : register:9;  k : register:11;  'in, out array indexes
in : array[511] of integer;  '16 across, 32 down
out : array[2047] of integer;  '32 across, 64 down
dtbeg
    lambda = | 0 1 1 1 1
    k = 2048-32 | - T F F F   'k = 2048?
k and 31 = 0 | - - T F F   'k mod 32 = 0?
k | - - - T F   'k odd?
-------------------+----------
    j := 0 | X - - -  'init input row
    k := 0 | X - - -  'init output row
    k := k + 32 | - - X - -  'skip one output row
    j := j + 1 | - - X X X 3  'inc input row index
    up_in := in[j] | X - X - X 3  'get upper byte pair
    dn_in := in[j+16] | X - X - X 3  'get lower byte pair
    shift := k | X - X X X  'shift even/odd k
    out[k] := up_out | X - X X X  'store upper byte pair
    out[k+32] := dn_out | X - X X X  'store lower pair
    k := k + 1 | X - X X X  'next output quad
    exit
    lambda := | 1 - - -
dtend

Compilation Statistics:
  5 rules, 4 conditions, 12 actions
Functional Memory: 5156 bytes
FPGA I/O: 6 inputs, 12 outputs
Microcode: 81 lines MC
FPGA PALASM: 493 CLBs (estimated)
  3 chips

Functional Memory Map

<table>
<thead>
<tr>
<th>Assigned Chip</th>
<th>Address or Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>lambda</td>
<td>R</td>
</tr>
<tr>
<td>&amp;Rule</td>
<td>P</td>
</tr>
</tbody>
</table>

180
pyramid ...... D 2 0004 0 up_in dn_in up_2 dn_2 up_out dn_out shift
up_in ...... R 2 0004 16
dn_in ...... R 2 0006 16
up_2 ...... E 2 0008 16
dn_2 ...... E 2 000A 16
up_out ...... E 2 000C 16
dn_out ...... E 2 000E 16
shift ...... R 2 0010 1
j ........... R 3 0012 9
k ........... R 1 0014 11
in .......... C 0 0016 0
out .......... C 0 0416 0
k+32 ...... E 1 1416 11 k + 32
j+1 ...... E 3 1418 9 j + 1
@in[j] ...... A 3 141A 10 in [ j ]
@in[j+16] ... A 3 141C 10 in [ j + 16 ]
@out[k] ..... A 1 141E 12 out [ k ]
@out[k+32] .. A 1 1420 12 out [ k + 32 ]
k+1 ........ E 1 1422 11 k + 1

*Types
A-Indirect Address
C-Array Base Address
D-Function Macro Declaration
E-Expression Output
P-Microprogram Address
R-FPGA Input Register

Execution Table
==============

<table>
<thead>
<tr>
<th>Adr</th>
<th>Statement</th>
<th>mP</th>
<th>Dest</th>
<th>Src</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>----------</td>
<td>----</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>Rule1=/L1*/L0</td>
<td>004: j := 0</td>
<td>DC 0012 0000 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00C: k := 0</td>
<td>DC 0014 0000 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>014: up_in := in[j]</td>
<td>DI 0004 141A 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>020: dn_in := in[j+16]</td>
<td>DI 0006 141C 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>02C: shift := k</td>
<td>DD 0010 0014 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>034: out[k] := up_out</td>
<td>ID 141E 000C 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>040: out[k+32] := dn_out</td>
<td>ID 1420 000E 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>04C: k := k+1</td>
<td>DE 0014 1422 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>054: lambda :=</td>
<td>DC 0000 0001 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>05C: goto @rule</td>
<td>JI 0002 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule2=/L1<em>L0</em>C2</td>
<td>004: exit</td>
<td>EX 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule3=/L1<em>L0</em>/C2*C3</td>
<td>06C: k := k+32</td>
<td>DE 0014 1416 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>074: j := j+1</td>
<td>DE 0012 1418 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>07C: up_in := in[j]</td>
<td>DI 0004 141A 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>088: dn_in := in[j+16]</td>
<td>DI 0006 141C 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>094: shift := k</td>
<td>DD 0010 0014 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>09C: out[k] := up_out</td>
<td>ID 141E 000C 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0A8: out[k+32] := dn_out</td>
<td>ID 1420 000E 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OB4: \( k := k + 1 \)  DE  0014  1422  2
0BC: goto @rule  JI  0002  2

Rule4 = \(/L1*LO*/C2*/C3*/C4\)

OC4: \( \text{shift} := k \)  DD  0010  0014  2
0CC: out\[k\] := up_out  ID  141E  000C  3
0DC: out\[k+32\] := dn_out  ID  1420  000E  3
0E4: \( k := k + 1 \)  DE  0014  1422  2
0EC: goto @rule  JI  0002  2

Rule5 = \(/L1*LO*/C2*/C3*/C4\)

0F4: \( j := j + 1 \)  DE  0012  1418  2
OF5C: up_in := in\[j\]  DI  0004  141A  3
108: dn_in := in\[j+16\]  DI  0006  141C  3
114: \( \text{shift} := k \)  DD  0010  0014  2
11C: out\[k\] := up out  ID  141E  000C  3
128: out\[k+32\] := dn out  ID  1420  000E  3
134: \( k := k + 1 \)  DE  0014  1422  2
13C: goto @rule  JI  0002  2
144:
APPENDIX 6. ROW-COLUMN HISTOGRAM EXAMPLE (ROWCOL,LIS)


File: c:\dissert\compiler\charrec\rowcol.src

Input File

Compute Row and Column Histograms

func RowColSum  'Row sum/column accumulate function
    InWord : register:16;    '16 pixel row input
    RowSum : expression:5;  'sum of row bits
    ColSums : array[15] of expression:5; 'column histogram
var j : register:4;      'array indexes
    char : array[15] of integer; 'character input array
    row : array[15] of integer; 'Row-wise histogram

dtbegin
lambda = | 0 1 1
j = 16    | - F T  '16 image words
----------------------------------------------------------
ColSums := 0    | X --  'Zero column sum array
j := 0          | X --  'loop through 16 words
InWord := char[j]| X X -- 'load function input
row[j] := RowSum| X X -- 'store sum for this row
j := j + 1      | X X -- 'increment row counter
exit           | -- X
lambda :=      | 1 --
dtend

Compilation Statistics:
3 rules, 2 conditions, 7 actions
Functional Memory: 112 bytes
FPGA I/O: 3 inputs, 21 outputs
Microcode: 29 lines MC
FPGA PALASM: 259 CLBs (estimated)
2 chips

Functional Memory Map

<table>
<thead>
<tr>
<th>Assigned Chip</th>
<th>Address or Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type</td>
</tr>
<tr>
<td>lambda</td>
<td>R</td>
</tr>
<tr>
<td>@Rule</td>
<td>P</td>
</tr>
<tr>
<td>rowcolsum</td>
<td>D</td>
</tr>
<tr>
<td>inword</td>
<td>R</td>
</tr>
<tr>
<td>rowsum</td>
<td>E</td>
</tr>
<tr>
<td>colsums</td>
<td>E</td>
</tr>
<tr>
<td>colsums.1</td>
<td>E</td>
</tr>
</tbody>
</table>

183
colsums.2 ... E 2 000C 5
colsums.3 ... E 2 000E 5
colsums.4 ... E 2 0010 5
colsums.5 ... E 2 0012 5
colsums.6 ... E 2 0014 5
colsums.7 ... E 2 0016 5
colsums.8 ... E 2 0018 5
colsums.9 ... E 2 001A 5
colsums.10 .. E 2 001C 5
colsums.11 .. E 2 001E 5
colsums.12 .. E 2 0020 5
colsums.13 .. E 2 0022 5
colsums.14 .. E 2 0024 5
colsums.15 .. E 2 0026 5
j ............ R 1 0028 4
char ........ C 0 002A 0
row ........... C 0 004A 0
@char[j] .... A 1 006A 6 char [ j]
@row[j] ...... A 1 006C 7 row [ j]
j+1 ........... E 1 006E 4 j + 1

*Types
A-Indirect Address
C-Array Base Address
D-Function Macro Declaration
E-Expression Output
P-Microprogram Address
R-FPGA Input Register

Execution Table
===============

<table>
<thead>
<tr>
<th>Addr</th>
<th>Statement</th>
<th>mP</th>
<th>Dest</th>
<th>Src</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule1=/L1*/L0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004: colsums:=0</td>
<td>DC</td>
<td>0008</td>
<td>0000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>00C: j:=0</td>
<td>DC</td>
<td>0028</td>
<td>0000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>014: inword:=char[j]</td>
<td>DI</td>
<td>0004</td>
<td>006A</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>020: row[j]:=rowsum</td>
<td>ID</td>
<td>006C</td>
<td>0006</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>02C: j:=j+1</td>
<td>DE</td>
<td>0028</td>
<td>006E</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>034: lambda:=</td>
<td>DC</td>
<td>0000</td>
<td>0001</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>03C: goto @rule</td>
<td>JI</td>
<td>0002</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

| Rule2=/L1*L0*/C2 |
| 044: inword:=char[j] | DI | 0004 | 006A | 3 |
| 050: row[j]:=rowsum | ID | 006C | 0006 | 3 |
| 05C: j:=j+1 | DE | 0028 | 006E | 2 |
| 064: goto @rule | JI | 0002 | | 2 |

| Rule3=/L1*L0*C2 |
| 06C: exit | EX | | 2 |
| 074: | | | | |
REFERENCES


