An FPGA-Based Minimal Instruction Set Computer

Richard Halverson, Jr. and Art Lew
Information and Computer Sciences Department
University of Hawaii at Manoa
Honolulu, Hawaii, 96822 (USA)
richardh@hawaii.edu, artlew@hawaii.edu

Memory mapped field programmable gate arrays (FPGAs) can be used to offload processor computations in microprocessor-based systems. Multi-operand expressions can be computed in combinational logic eliminating computation cycles. When *all* computations take place in the FPGAs, the processor no longer needs an arithmetic/logic unit and a RISC system whose instruction set consists of only moves and jumps results. This FPGA supported minimal instruction set approach can be used to implement small fast microprocessor systems that can be customized for a varying set of specialized applications. This paper describes the design and implementation of such a system.

1 INTRODUCTION

Reprogrammable FPGAs allow changes or revisions of "hardware" at the gate level, in circuit, in a couple of hundred milliseconds, hence, FPGA circuitry can be different for different user programs. Reprogrammable FPGAs have been available since 1986 and are now used in a variety of applications [1]. Since FPGAs still offer minimal reprogrammable gate counts (~20,000), in many cases, the added parallel processing is still necessarily fine-grained. With the clock rates of off-the-shelf microprocessors exceeding 100 MHz with 1–2 cycle instruction executions, it is difficult for reprogrammable FPGAs to compete in sequential computations. Computing expressions with more than two terms on a microprocessor involves a sequence of arithmetic and/or logical operations typically with assignments being made to temporary variables. FPGAs can increase system performance when computations can be performed in parallel in a reasonable amount of custom combinational logic.

Employing reprogrammable FPGAs for "custom computing" is an area other researchers are pursuing. PRISM-II is an Am29050 based proprietary architecture which contains an expandable

set of three-FPGA reconfigurable platforms for custom coprocessing [2]. PRISM-II uses a configuration compiler which accepts a user program written in C as input, and generates hardware and software "images" as output. The hardware image contains information that is necessary for synthesizing hardware for the reconfigurable platform corresponding to the time critical sections of code, which are identified a priori by the programmer. Critical sections can be iterative procedures. The main processor passes operands into the FPGAs and copies out the results. A limitation of PRISM-II is that global and static variables that must be accessible from outside the FPGA must be reimplemented separately in the form of explicit arguments to the critical section.

The Anyboard is a five FPGA logic emulator platform on an IBM PC plug compatible circuit board [3]. Designed as a lower cost alternative to hardware prototyping/emulation/hardware simulation systems (e.g., Quickturn, Zycad), logic circuits can be specified using the SOLDER hardware description language, and the resulting circuit will be compiled and partitioned for implementation on the Anyboard. Since Anyboard is for prototyping hardware, SOLDER 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 PRISM-II (as well as our system), the goal is to translate high level language software programs. 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. Its FPGA logic, however, is not memory mapped but interfaces with the microprocessor via the I/O bus.

The Virtual Computer [4] is an S-bus compatible single FPGA logic emulator platform with optional memory expansion. Similar to PRISM-II, it was designed for implementing custom hardware for software execution support. Engineers use existing hardware design tools (e.g., ViewLogic, OrCad) to implement logic circuits in the FPGA. 256 bytes of memory mapped register space is available for the programmer interface. After the final placement and routing of the FPGA chip, Virtual Computer software tools create a hardware object which can be inserted into C program applications. Software can then read and write the registers from a C program. These downloadable "circuits" as well as their memory mapped interface to the program are

defined by a hardware engineer using FPGA design tools and separately from the applications software which utilizes the board.

Our approach is similar to PRISM's in that the FPGA configuration is derived from expressions found within the application program. Whether right sides of assignment statements, array element address computations or program jump addresses, expressions are extracted from the program by the high level language compiler and coded automatically into a text based hardware description language for configuring the FPGAs (thus eliminating the need for the hardware engineer). Unlike PRISM, however, our variables can be shared by both the FPGA and microprocessor without having to implement separate explicit arguments for passing data. Our approach memory maps the shared variables, however, unlike a Virtual Computer, the number of variables and their locations are completely determined by the high level language compiler. Therefore, unlike any other FPGA based custom coprocessor that we are aware of, our approach does not require defining explicit variables for passing arguments to the FPGA. With FPGA custom coprocessing still necessarily fine-grained, this feature alone can result in a substantial performance advantage.

Connecting programmable memory mapped FPGAs in parallel with the RAM provides the functionality that a spreadsheet program adds to a manual spreadsheet. A *manual spreadsheet* (e.g., paper and pencil), that is, one that can only "store" numbers written into its cells by the user, is analogous to the conventional RAM of the microprocessor. With a *spreadsheet program*, cells (i.e., memory locations) can not only be written for storage, but they can be programmed with formulas to compute expressions based on values in other cells. Our approach is to connect FPGAs in parallel with RAM to provide this spreadsheet functionality effect. RAM with one or more FPGAs memory mapped in this fashion we call *functional memory*. Functional memory (FM) allows program expressions to be computed in parallel in combinational logic as operand variables are assigned new values, just as formula cells of a spreadsheet recompute "instantaneously" when operand cells are changed. When *all* expression computation is performed in the functional memory, then the necessary processor instruction set reduces to a simple set of moves and jumps. This allows the processor to be small and streamlined to less than

what is necessary for a typical RISC processor because no general purpose operand registers or ALU is needed.

This paper describes the design of our prototype functional memory computer (FMC). A FMC consists of a processor connected to a memory via a standard address–data bus. In a FMC, some or all of an executing program's expression computation is performed in the FM. Using our "minimal" processor, which contains no ALU or internal operand registers, all the program's expression computation can be performed in the FM. Section 2 describes how to implement a FM and how it is programmed. Section 3 describes how logic for computing expressions and program addresses are implemented in combinational logic of the FPGAs using the PALASM hardware description language. Section 4 describes the architecture and implementation of our minimal processor and shows sample object code for it as output from our compiler using a binary search program example. Section 5 briefly describes the compiler. Section 6 concludes with implications of this research and our future plans.

2 CONSTRUCTING A FUNCTIONAL MEMORY

The design of a functional memory can be accomplished in several ways. Figure 1 illustrates how one can 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 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. When data are stored into the RAM, operands used in expression computations are captured simultaneously in FPGA registers. When expression results are read by the microprocessor, the FPGA drives the data bus. Bidirectional FPGA data lines and restricting expression outputs to locations absent of RAM simplifies the circuit. Other techniques are described in [5], [6], and [7].

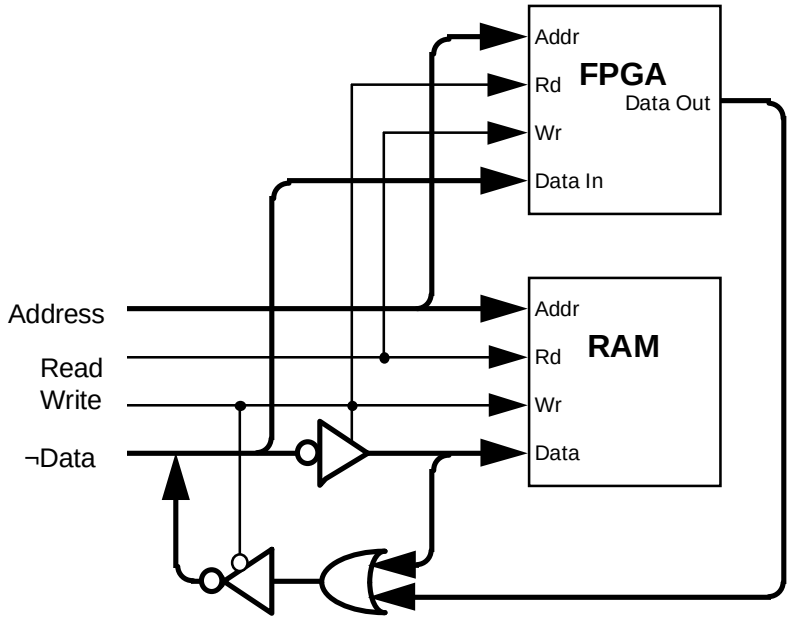

Figure 1. Functional Memory - Logical OR Method

2.1 Programming the FPGA

With functional memory, input and output locations as well as the expressions for computation are configured when the user program is loaded for execution because the configuration will not only depend on what expressions the user includes in the program, but also where the compiler allocates the memory locations of the variables used in those expressions. The FPGA configuration programs are produced by compiler from a high-level programming language, therefore it is easiest to use a text based hardware description language for defining the FPGA circuitry. Several such languages exist for VLSI designs. We chose PALASM because of our experience, availability, and because it is similar to but simpler than other hardware description languages such as Xilinx ABEL [8]. 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. 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.

As Figure 2 suggests, each supported FPGA chip type (and pin assignment) has its own interface circuit shell schematic which specifies the in-circuit pin configuration. All incoming and outgoing signals have assigned specific pin numbers and are buffered. 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 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 3 indicates, this is accomplished by connecting the data output driver inputs to its high impedance output 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).

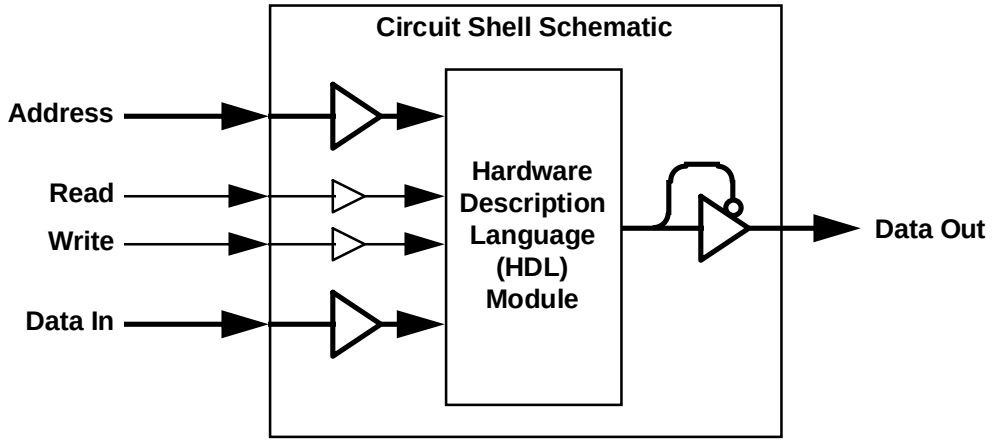

Figure 2. Circuit Shell and the HDL Module

Within the interface circuit shell is defined a standardized hardware description language module. Figure 3 shows the interface "pins" defined using the PALASM ".PDS" file format. Address, control, data in and data out pins are defined, in order, using the same names as used in the circuit shell schematic. 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 with all

HDLs, the PALASM language syntax is similar to conventional high level languages. Combinational outputs are defined using the "=" assignment symbol whereas flip-flop contents are defined using the ":=" assignment. Bit expression operators include "∗" for logical AND (often denoted mathematically as "∧"), "+" for logical OR (∨) and ":+:" for exclusive-OR (⊕). Unary inversion (¬) is indicated by preceding the signal name or expression with a "/" character.

```
TITLE    BINSRCH
AUTHOR   Mr. D. T. Compiler
DATE     12-20-1994 at 14:10:08

CHIP     1BINSRCH   LCA

;ADDRESS INPUT PINS (18)
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15
RDC WRLC WRHC ;READ, WRITE LOW BYTE, WRITE HIGH BYTE
;DATA INPUT PINS (16)
DI0 DI1 DI2 DI3 DI4 DI5 DI6 DI7 DI8 DI9 DI10 DI11 DI12 DI13 DI14 DI15
GCLK ZERO ;GLOBAL CLOCK, LOGICAL ZERO
;DATA OUTPUT PINS (16)
DO0 DO1 DO2 DO3 DO4 DO5 DO6 DO7 DO8 DO9 DO10 DO11 DO12 DO13 DO14 DO15

EQUATIONS
 ;Expression Logic
   ...
 ;Input Registers
   ...
 ;Output Multiplexer
   ...
 ;Address Select Logic
   ...
 ;Rule Address Generation Logic
   ...
;END
```

Figure 3. HDL (PALASM) Declarations and Format

3 EXPRESSION COMPUTATION

This section describes how FPGAs can be configured to perform expression computations. Section 3.1 describes the implementation of arithmetic and logical operations. Section 3.2 describes program address calculation. Section 3.3 describes the input and output interface.

3.1 Implementing 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 $B \wedge \neg B = 0$ logical equivalence and one values can be obtained using $B \vee \neg B = 1$.

3.1.1 $A \neq B$ Test

Figure 4 shows how the $A \neq B$ test can be implemented. Equations are listed in order of increasing bit position. This is because the compiler keeps track of how many bits are necessary for each variable, and only implements that which is necessary for the program. Intermediate signals exclusive-ORing bits from A and B in each position are all ORed together at a second level. If any pair of bits were different in any bit position, a one value will prevail. The same circuit can also be used to test if two operands are equal by inverting the final C output. The $A \neq B$ test for two 16-bit integers consumes 11 configurable logic blocks (CLBs) in a Xilinx 3090 and can be compared in three gate delays [8].

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

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
;
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
C = C_eq0_7 + C_eq8_15
```

Figure 4. $A \neq B$ Test PALASM

3.1.2 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 5 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 determined. By inverting the *C* output, *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_lt*i*_*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 = 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

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_lt6 = C_lt6_4_0 + C_lt6_4_1*(C_lt4_2_0 + C_lt4_2_1*C_lt2)
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_lt12_6_0 = C_lt12_10_0+C_lt12_10_1*C_lt10_8_0+C_lt12_10_1*C_lt10_8_1*C_lt8_6_0
C_lt12_6_1 = C_lt12_10_0 + C_lt12_10_1*C_lt10_8_0 +
C_lt12_10_1*C_lt10_8_1*C_lt8_6_1
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)
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
;===
C = C_lt16_12_0 + C_lt16_12_1*C_lt12_6_0 + C_lt16_12_1*C_lt12_6_1*C_lt6
```

Figure 5. *A* < *B* PALASM

3.1.3 Adding and Subtracting

Figure 6 shows the basic *A*+*B* combinational logic function. By inverting each *B* bit and setting the carry in (C_0) 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* = carry into bit *i* if the carry into bit *j* (*j* < *i*) is *k*. (*k* = 0 or 1). For the intermediate sum bits, Ds*i_j_k* = sum bit *i* if carry into bit *j* (*j* < *i*) is *k*. (*k* = 0 or 1.) For example, Ds9_8_1 = sum bit 9 if carry into bit 8 is 1.

Figure 6. *D* = *A* + *B* Equations

```
;D = A+B+C0, D, A, B are 16 bits, C0 is carry bit in

D0 = A0:+:B0:+:C0
;---
D1 = A1:+:B1:+:(A0*B0+(A0+B0)*C0)
;---
Dc2 = A1*B1+(A1+B1)*(A0*B0+(A0+B0)*C0)
D2 = A2:+:B2:+:Dc2
;---
D3 = A3:+:B3:+:(A2*B2+(A2+B2)*Dc2)
;---
Dc4_2_0 = A3*B3+(A3+B3)*(A2*B2)
Dc4_2_1 = A3*B3+(A3+B3)*(A2+B2)
D4 = A4:+:B4:+:(Dc4_2_0+Dc4_2_1*Dc2)
;---
Ds5_4_0 = A5:+:B5:+:(A4*B4)
Ds5_4_1 = A5:+:B5:+:(A4+B4)
D5 = Ds5_4_0*/(Dc4_2_0+Dc4_2_1*Dc2)+Ds5_4_1*(Dc4_2_0+Dc4_2_1*Dc2)
;---
Dc6_4_0 = A5*B5+(A5+B5)*(A4*B4)
Dc6_4_1 = A5*B5+(A5+B5)*(A4+B4)
Dc6 = Dc6_4_0+Dc6_4_1*(Dc4_2_0+Dc4_2_1*Dc2)
D6 = A6:+:B6:+:Dc6
;---
D7 = A7:+:B7:+:(A6*B6+(A6+B6)*Dc6)
;---
Dc8_6_0 = A7*B7+(A7+B7)*(A6*B6)
Dc8_6_1 = A7*B7+(A7+B7)*(A6+B6)
D8 = A8:+:B8:+:(Dc8_6_0+Dc8_6_1*Dc6)
;---
Ds9_8_0 = A9:+:B9:+:(A8*B8)
Ds9_8_1 = A9:+:B9:+:(A8+B8)
D9 = Ds9_8_0*/(Dc8_6_0+Dc8_6_1*Dc6)+Ds9_8_1*(Dc8_6_0+Dc8_6_1*Dc6)
;---
Dc10_8_0 = A9*B9+(A9+B9)*(A8*B8)
Dc10_8_1 = A9*B9+(A9+B9)*(A8+B8)
Dc10_6_0 = Dc10_8_0+Dc10_8_1*Dc8_6_0
Dc10_6_1 = Dc10_8_0+Dc10_8_1*Dc8_6_1
D10 = A10:+:B10:+:(Dc10_6_0+Dc10_6_1*Dc6)
;---
Ds11_10_0 = A11:+:B11:+:(A10*B10)
Ds11_10_1 = A11:+:B11:+:(A10+B10)
D11 = Ds11_10_0*/(Dc10_6_0+Dc10_6_1*Dc6)+Ds11_10_1*(Dc10_6_0+Dc10_6_1*Dc6)
;---
Dc12_10_0 = A11*B11+(A11+B11)*(A10*B10)
Dc12_10_1 = A11*B11+(A11+B11)*(A10+B10)
Dc12_6_0 = Dc12_10_0+Dc12_10_1*Dc10_8_0+Dc12_10_1*Dc10_8_1*Dc8_6_0
```

```
Dc12_6_1 = Dc12_10_0+Dc12_10_1*Dc10_8_0+Dc12_10_1*Dc10_8_1*Dc8_6_1
D12 = A12:+:B12:+:(Dc12_6_0+Dc12_6_1*Dc6)
;---
Ds13_12_0 = A13:+:B13:+:(A12*B12)
Ds13_12_1 = A13:+:B13:+:(A12+B12)
D13 = Ds13_12_0*/(Dc12_6_0+Dc12_6_1*Dc6)+Ds13_12_1*(Dc12_6_0+Dc12_6_1*Dc6)
;---
Dc14_12_0 = A13*B13+(A13+B13)*(A12*B12)
Dc14_12_1 = A13*B13+(A13+B13)*(A12+B12)
Ds14_12_0 = A14:+:B14:+:Dc14_12_0
Ds14_12_1 = A14:+:B14:+:Dc14_12_1
D14 = Ds14_12_0*/(Dc12_6_0+Dc12_6_1*Dc6)+Ds14_12_1*(Dc12_6_0+Dc12_6_1*Dc6)
;---
Ds15_14_0 = A15:+:B15:+:(A14*B14)
Ds15_14_1 = A15:+:B15:+:(A14+B14)
Ds15_12_0 = Ds15_14_0*/Dc14_12_0+Ds15_14_1*Dc14_12_0
Ds15_12_1 = Ds15_14_0*/Dc14_12_1+Ds15_14_1*Dc14_12_1
D15 = Ds15_12_0*/(Dc12_6_0+Dc12_6_1*Dc6)+Ds15_12_1*(Dc12_6_0+Dc12_6_1*Dc6)
```

Figure 6 (continued). *D* = *A* + *B* Equations

Notice that the equations in Figure 6 are also grouped by bit. As with the comparison tests, 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.

3.2 Generating Program Addresses

The most general form of conditional jumping is a multi-way computed branch similar to the one shown in Figure 7. Jump addresses are derived from one or more conditional expressions which depend 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. It has been shown [9] that any flow chart for any program can be transformed into a single multi-way branch like the one in Figure 7 therefore can be expressed using a single iterating decision table so our choice of language is not restrictive. Multi-way branches implemented on typical von Neumann architectures are implemented by sequentially examining each branch condition taking the first one found to be true. With functional memory, a single expression can be derived to compute the jump address of a multi-way branch in a single step. To demonstrate this we will use the binary search algorithm shown in Figure 8.

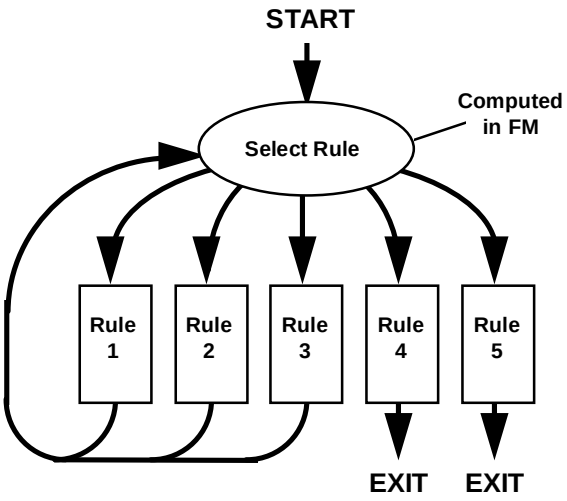

Figure 7. Multi-way Branch

```
1.  procedure BinSearch(n, v, var index);  {Binary Search Procedure returns index}
2.  var i, l, r : integer;
3.  begin
4.      l := 1; r := n;                  {initialize left and right search range}
5.      repeat                                                              {steps}
6.          i := (l+r) div 2;            {point to middle of range}          {4}
7.          if v<a[i].key                {test if value too low or too high} {7}
8.              then r := i–1            {too low so decrement right boundary} {3}
9.              else l := i+1;           {too high so increment left boundary}
10.     until (v=a[i].key) or (l>r);     {loop until index found or value not present} {6}
11.     if v=a[i].key                    {test if value found or not present}
12.         then index := i              {value found so set return index}
13.         else index := n+1            {not found so return next available location}
14. end {BinSearch}
```

Figure 8. Binary Search from Sedgewick [10]

Figure 9 shows a decision table version of the binary search algorithm expressed in Figure 8. The condition stub section in the upper left of the table contains all the conditional expressions of the program that are used to determine jump addresses. (The variable *lambda* here is used for initialization but for more complex programs it can be used to keep track of control flow.) The decision table has five rules corresponding to the multi-way branch diagram shown in Figure 7. The condition entry at the intersection of a condition stub row and a rule column shows the

necessary state of that condition for that rule to fire (that is, for the actions associated with that rule to be selected for execution). When the condition stub states match in a particular rule column, then that rule fires, causing action stubs in the lower left corner to execute if an X appears in the action stub row and rule column intersection. Action stubs are executed in order from top to bottom.

	Rule:	*1*	*2*	*3*	*4*	*5*	
1	*lambda* =	0	1	1	1	1	
2	*v* < "*a*[*i*].*key*"		T	F	F		
3	*v* = "*a*[*i*].*key*"		F	F	T		
4	*l* > *r*		F	F	F	T	*steps*
5	*l* := 1	X					2
6	*r* := *n*	X					2
7	*r* := *i*–1		X				2
8	*l* := *i*+1			X			2
9	*index* := *i*				X		2
10	*index* := *n*+1					X	2
11	*i* := (*l*+*r*) **div** 2	X	X	X			2
12	"*a*[*i*].key" := *a*[*i*].*key*	X	X	X			3
13	**exit**				X	X	2
14	*lambda* :=	1					2

Starting	*address*
Rule 1	0004_{16}
Rule 2	0038_{16}
Rule 3	$005C_{16}$
Rule 4	0080_{16}
Rule 5	0090_{16}

Number	*of steps*
Rule 1	13
Rule 2	9
Rule 3	9
Rule 4	4
Rule 5	4

Figure 9. Binary Search Decision Table and Rule Starting Addresses

Figure 9 on the right provides the actual rule starting addresses for this example for our FMC. Rule 1 starts at location 0004_{16}. Rule 2 starts at 0038_{16}, Rule 3 at $005C_{16}$, Rule 4 at 0080_{16} and Rule 5 starts at location 0090_{16}. From these addresses, we can see that, for example, address bits 0 and 1 are always zero, hence need not be implemented. Address bit 2 is a one when Rule 1 or Rule 3 is selected. Address bit 3 is a one when Rule 2 or Rule 3 is selected. Address bit 4 is a one when Rules 2, 3 or 5 is selected. Address bit 5 is a one when Rule 2 is selected. Address bit 6 is a one when Rule 3 is selected and address bit 7 is a one when either Rule 4 or 5 is selected. The remaining address bits in this example are zero therefore need not be implemented.

Figure 10 shows the PALASM code for implementing this example. First are listed those signals which are true for each rule when selected (Rule1, ..., Rule5). 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, $C2 = (v <$ "$a[i].key$"), $C3 = (v =$ "$a[i].key$") and $C4 = (l > r)$.

```
;@Rule bits from L0, L1, C2, C3, C4

Rule1 = /L1*/L0             ;Starting at Microprogram Address 004
Rule2 = /L1*L0*C2*/C3*/C4   ;Starting at Microprogram Address 038
Rule3 = /L1*L0*/C2*/C3*/C4  ;Starting at Microprogram Address 05C
Rule4 = /L1*L0*/C2*C3*/C4   ;Starting at Microprogram Address 080
Rule5 = /L1*L0*C4           ;Starting at Microprogram Address 090

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

Figure 10. Rule Jump Address Calculation

The second set of signals in Figure 10 are the bit output values of the starting address for the rule chosen to be executed. 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. The rule address can be computed from the condition stubs results in fewer than ten CLBs.

3.3 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 after their locations have been determined.

Our FMC was implemented using Xilinx 3090 [8] 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 *A*15 to *A*1 are used to decode byte address pairs for reading or writing. Address line *A*0 is not needed because odd bytes are not addressed separately, but are read and written simultaneously with the preceding even address just below as 16-bit words.

3.3.1 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 variable. Figure 11 lists the PALASM equations for implementing the 13-bit register mapped to address location 0006 which captures a new value for the search key *v* in the binary search program. "*reg*_06_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*_06" select line shown generated in Figure 13.

When the total number of different input signals into a pair of equations or flip-flops is four or fewer, they can be combined into one CLB. Therefore, the 13-bit register capturing writes to address 0006 (and 0007) require only seven CLBs.

3.3.2 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 12 shows a multiplexer example for a total of six expressions. For data output bit 0 (*DO*0), either the value of *i*–1 is selected when address $07E4_{16}$ is read, *i*+1 is selected when address $07E4_{16}$ is read, *n*+1 is selected when address $07E8_{16}$ is read, or (*l*+*r*) **div** 2 is selected when address $07EA_{16}$ is read. For output bit 2, the next rule program address (*@Rule*) is added and is output when address 0002_{16} is read, and the address of array element *a*[*i*] is output when address $07EC_{16}$ is read. By using three intermediate gate levels by convention, we can accommodate up to 2x5x4 = 40

expressions in one chip. 200 output expressions can be accommodated by adding a fourth gate level. Notice that the read signal (/RDC) should be ANDed last to minimize chip access time.

```
;v input register at address 06
reg_06_0     := DI0      ;              v input bit 0
reg_06_0.CLKF = WRLC     ;Write Clock
reg_06_0.CE   = sel_06  ;Clock Enable
reg_06_1     := DI1      ;              v input bit 1
reg_06_1.CLKF = WRLC     ;Write Clock
reg_06_1.CE   = sel_06  ;Clock Enable
reg_06_2     := DI2      ;              v input bit 2
reg_06_2.CLKF = WRLC     ;Write Clock
reg_06_2.CE   = sel_06  ;Clock Enable
...
reg_06_11     := DI11      ;               v input bit 11
reg_06_11.CLKF = WRHC     ;Write Clock
reg_06_11.CE   = sel_06  ;Clock Enable
reg_06_12     := DI12      ;               v input bit 12
reg_06_12.CLKF = WRHC     ;Write Clock
reg_06_12.CE   = sel_06  ;Clock Enable
```

Figure 11. Input Register PALASM

```
DO0t1 = s_09_0*sel_07E4 + s_0B_0*sel_07E6 ;  i-1, i+1
DO0t3 = s_0D_0*sel_07E8 + s_0F_0*sel_07EA ;  n+1, (l+r)div2
DO0a1 = DO0t1 + DO0t3
DO0  = /(/RDC*(DO0a1))
...
DO2t1 = rule_2*sel_02 + s_09_2*sel_07E4 ;  @Rule, i-1
DO2t3 = s_0B_2*sel_07E6 + s_0D_2*sel_07E8 ;  i+1, n+1
DO2t5 = s_0F_2*sel_07EA + a_013_2*sel_07EC ;  (l+r)div2, @a[i]
DO2a1 = DO2t1 + DO2t3 + DO2t5
DO2  = /(/RDC*(DO2a1))
...
DO11t1 = a_013_11*sel_07EC ;  @a[i]
DO11a1 = DO11t1
DO11  = /(/RDC*(DO11a1))
```

Figure 12. 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 (*A*0) is always zero hence need not appear. (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 CLBs which are the scarcest resource in a FMC.

3.3.3 I/O Address Select Logic

The address select logic must produce a select line for each input register and each output expression in that particular FPGA. Each will correspond to a unique (compiler allocated) memory address location. Because of the nature of the Xilinx 3000 series 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 the binary search example, our program specifies seven FPGA input registers mapped to locations 0000, 0004, 0006, 07DC, 07DE, 07E0, and 07E2, and six expressions to be output when locations 0002, 07E4, 07E6, 07E8, 07EA and 07EC are read. As shown in Figure 13, 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 *A*15-*A*12, *A*11-*A*8, *A*7-*A*4, and *A*3-*A*0 respectively (although recall that *A*0 is not decoded). The remaining hex digit identifies what hex value for this group of four address lines the signal is 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 when the processor reads *i*–1 (presumably to assign to the variable *r*), the select line for address 07E4 would activate with *A*15-*A*12 equal to 0 (*ASX*0), *A*11-*A*8 equal to 7 (*ASH*7), *A*7-*A*4 equal E_{16} (*ASM*E) and *A*3-*A*0 equal to 4 (*ASL*4). Once all the select equations have been generated, we know which specific hex digits also must decoded (each corresponding to a group of four address lines. In this example, 14 groups must be decoded to generate the select lines for 13 input and output addresses.

```
sel_00 = ASX0*ASH0*ASM0*ASL0   ;Select for lambda input
sel_02 = ASX0*ASH0*ASM0*ASL2   ;Select for @Rule output
sel_04 = ASX0*ASH0*ASM0*ASL4   ;Select for n input
sel_06 = ASX0*ASH0*ASM0*ASL6   ;Select for v input
sel_07DC = ASX0*ASH7*ASMD*ASLC   ;Select for i input
sel_07DE = ASX0*ASH7*ASMD*ASLE   ;Select for l input
sel_07E0 = ASX0*ASH7*ASME*ASL0   ;Select for r input
sel_07E2 = ASX0*ASH7*ASME*ASL2   ;Select for "a[i]" input
sel_07E4 = ASX0*ASH7*ASME*ASL4   ;Select for i-1 output
sel_07E6 = ASX0*ASH7*ASME*ASL6   ;Select for i+1 output
sel_07E8 = ASX0*ASH7*ASME*ASL8   ;Select for n+1 output
sel_07EA = ASX0*ASH7*ASME*ASLA   ;Select for (l+r)div2 output
sel_07EC = ASX0*ASH7*ASME*ASLC   ;Select for @a[i] output

ASX0 = /A12*/A13*/A14*/A15    ; Highest &H0
ASH0 = /A8*/A9*/A10*/A11      ; High &H0
ASM0 = /A4*/A5*/A6*/A7        ; Middle &H0
ASL0 = /A1*/A2*/A3            ; Low &H0
ASL2 = A1*/A2*/A3             ; Low &H2
ASL4 = /A1*A2*/A3             ; Low &H4
ASL6 = A1*A2*/A3              ; Low &H6
ASH7 = A8*A9*A10*/A11         ; High &H7
ASMD = A4*/A5*A6*A7           ; Middle &HD
ASLC = /A1*A2*A3              ; Low &HC
ASLE = A1*A2*A3               ; Low &HE
ASME = /A4*A5*A6*A7           ; Middle &HE
ASL8 = /A1*/A2*A3             ; Low &H8
ASLA = A1*/A2*A3              ; Low &HA
```

Figure 13. Address Select PALASM

4 THE MINIMAL PROCESSOR

A block diagram of the 4.5” by 6.5” FMC board that we have implemented is illustrated in Figure 14. The board plugs into the system bus of a parallel computer platform that uses an Intel MCS-51 8031 (i8031) microcomputer as the system processor. 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. The RAM–FPGA OR function (see Figure 1) is performed inside the mP.

The system processor connects to an IBM PC serially at 19,200 bits per second. The FMC is initialized and controlled by the system processor through its Port 1 register as shown in Figure 15. The FPGA configuration program is loaded first, which includes the mP specification and the four FM programs.

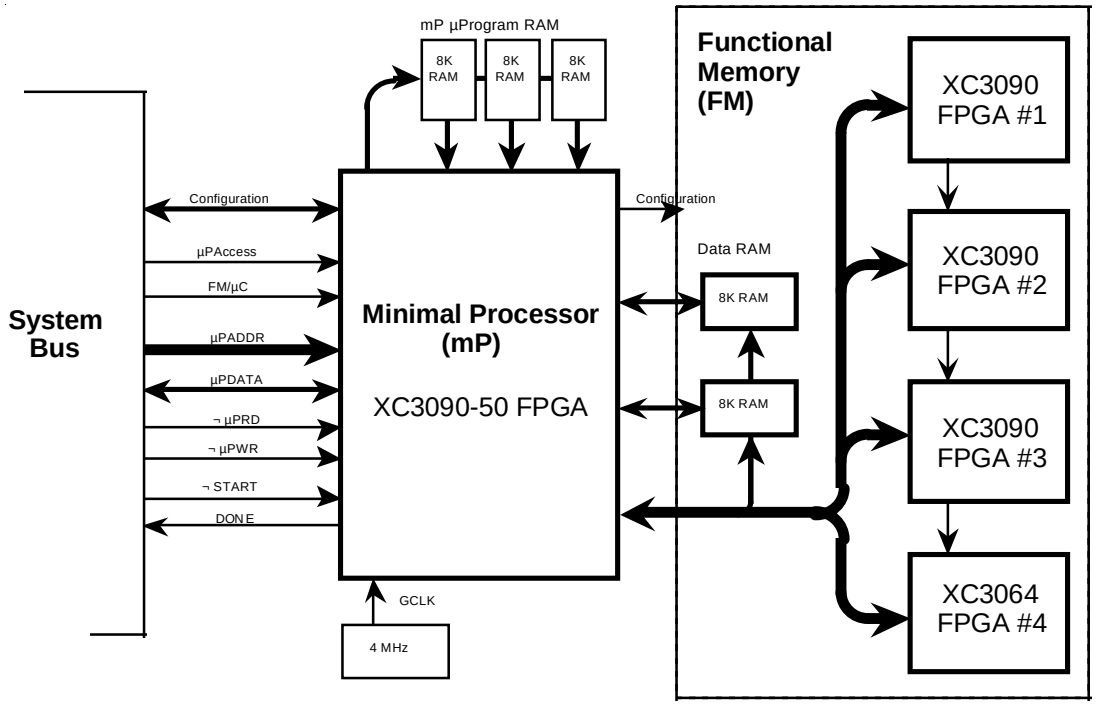

Figure 14. Functional Memory Computer Board Block Diagram

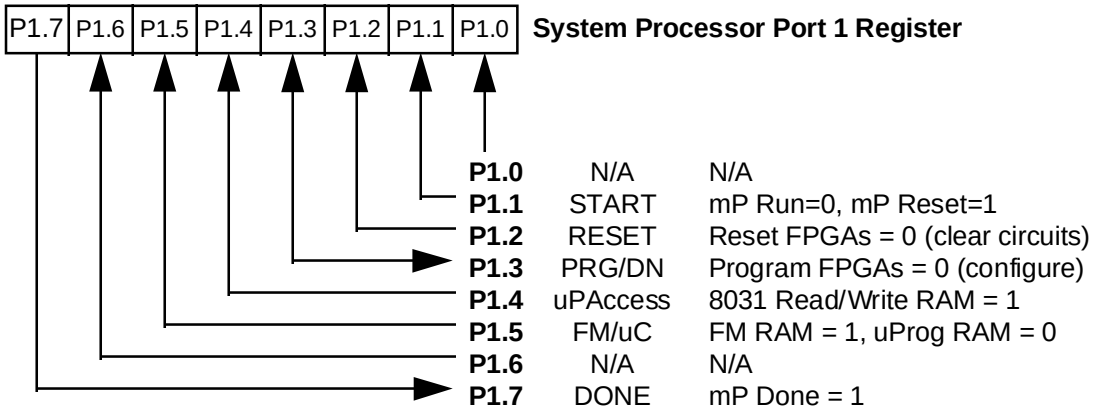

Figure 15. System Processor - Minimal Processor Interface

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* and *DONE* handshake signals begin program execution and notify the system processor when execution terminates.

4.1 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 or from a memory location. As Figure 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 the 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; in particular, there are no temporary operand registers and no arithmetic logic unit.

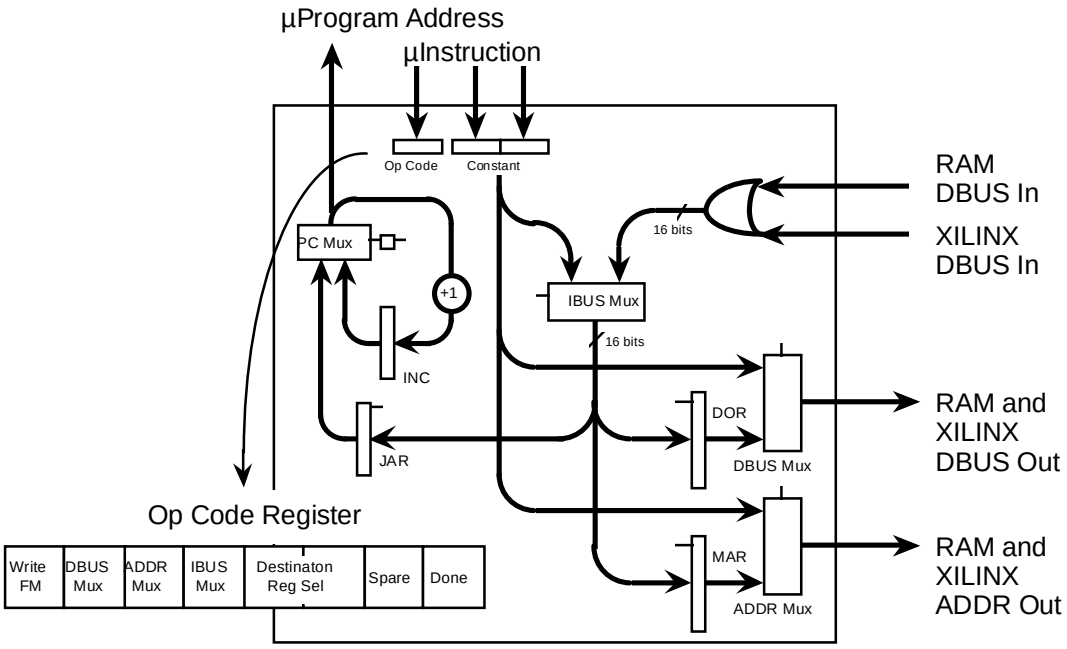

Figure 16. Architecture of the Minimal Processor

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*.

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 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.

4.2 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 and 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 1 describes how our mP provides a minimal set of move and control instructions for implementing any program. Instruction mnemonics are defined with the first letter indicating the destination addressing mode (**D**irect addressing, **I**ndirect addressing or **J**ump address register) and the second letter indicating the source (**C**onstant, **D**irect address, **E**xpression, **I**ndirect address).

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. 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 1 lists the number of cycles each mP instruction takes. These values can be divided by the FMC oscillator frequency 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.

4.3 Microinstruction Set Implementation

A microinstruction set can be designed to implement the µcode instructions listed in fifth column of Table 1. The registers and data paths in Figure 16 are controlled by the *Op Code Register* shown. There are seven relevant bits which provide signals which gate multiplexer paths for moving operands between the registers and the functional memory. 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, 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.

Table 1. Minimal Processor Instruction Set

Type	Example	Operand #1	Operand #2	µCode Instructions	Cycles
DC	*i* := 1	Direct Address e.g., *i*	Constant e.g., 1	DOR ← 1 (*i*) ← DOR	2
DD	*i* := *j*	Direct Address e.g., *i*	Direct Address e.g., *j*	DOR ← (*j*) (*i*) ← DOR	2
DE	*i* := *j*+*k*	Direct Address e.g., *i*	Direct Address e.g., *j*+*k*	DOR ← (*j*+*k*) (*i*) ← DOR	2
DI	*i* := *a*[*j*]	Direct Address e.g., *i*	Indirect Address e.g., @*a*[*j*]	MAR ← (@*a*[*j*]) DOR ← (MAR) (*i*) ← DOR	3
	"*a*[*j*+1]" := *a*[*j*+1]	Direct Address e.g., "*a*[*j*+1]"	Indirect Address e.g., @*a*[*j*+1]	MAR ← (@*a*[*j*+1]) DOR ← (MAR) ("*a*[*j*+1]") ← DOR	3
IC	*a*[*i*] := 1	Indirect Address e.g., @*a*[*i*]	Constant e.g., 1	MAR ← (@*a*[*i*]) (MAR) ← 1	2
ID	*a*[*i*+1] := *j*	Indirect Address e.g., @*a*[*i*+1]	Direct Address e.g., *j*	MAR ← (@*a*[*i*+1]) DOR ← (*j*) (MAR) ← DOR	3
IE	*a*[*i*+1] := *j*+*k*	Indirect Address e.g., @*a*[*i*+1]	Direct Address e.g., *j*+*k*	MAR ← (@*a*[*i*+1]) DOR ← (*j*+*k*) (MAR) ← DOR	3
II	*a*[*i*] := *a*[*i*+1]	Indirect Address e.g., @*a*[*i*]	Indirect Address e.g., @*a*[*i*+1]	MAR ← (@*a*[*i*+1]) DOR ← (MAR) MAR ← (@*a*[*i*]) (MAR) ← DOR	4
JI	GOTO (*location*)	Indirect Address e.g., *location*	N/A	µPC ← (*location*) NOP*	2
EX	HALT	N/A	N/A	µPC ← µPC µPC ← µPC–4*	2

* - necessary because of pipelining

The contents of the internal data bus can be clocked into either the data output register, which occurs when the *Destination Register Select* bits are 01. When the *Destination Register Select* bits are 10, the memory address register gets clocked, and when *Destination Register Select* = 11, the program counter gets the contents of the internal data bus. 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 the *µCode Instructions* column of Table 1. Table 2 lists the 10 microinstruction codes used to implement the minimal processor (macro) instruction set.

Table 2. Microinstructions for Implementing mP Operations

µI Op Code	Op Code Register	Example Usage	µCode Implementation Example (see Table 1)
LDC	0000 0100	LDC, 1	*DOR* ← 1. Load the Data output register with a constant. Used in the DC instruction
LDA	0001 0100	LDA, *j*	*DOR* ← (*j*). Load the Data output register with the contents of the Address specified. Used in the DD, DE, ID and IE instructions
LDM	0011 0100	LDM, 0	*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
LMA	0001 1000	LMA, *@a*[*i*]	*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
WMD	1110 0000	WMD, 0	(*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
WAD	1100 0000	WAD, *i*	(*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
WMC	1010 0000	WMC, 1	(*MAR*) ← 1. Write the memory address specified in the Memory address register, with a Constant value. Used in the IC instruction
JPI	0001 1100	JPI, @Rule	*µPC* ← (*@Rule*). Jump to the location specified Indirectly. Used following each rule to jump to the next rule
HALT	0000 1101	HALT, $	*µPC* ← *µPC*. Jump to the specified location and set the "Done" flag. Used to terminate execution and to notify the system processor
NOP	0000 0000	NOP, 0	No OPeration. Do nothing for one cycle. Used following the JPI for pipelining

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. The assembler output for the binary search decision table program is appears in Listing 1. The

resulting object file is downloaded directly into the microprogram memory of the minimal processor for execution.

Listing 1. Microcode Assembler Output

```
             INPUT  FILENAME :  BINSRCH.ASM
             OUTPUT FILENAME :  BINSRCH.OBJ

 1          .TITLE  BINSRCH 12-20-1994 at 14:10:08
 2                                  ;
 3                                  ; MICROINSTRUCTION OPCODES
 4                                  ;DOR <-- constant
 5           00 04                  LDC     EQU     00000100B
 6                                  ;DOR <-- (address)
 7           00 14                  LDA     EQU     00010100B
 8                                  ;DOR <-- (MAR) used LDM,0
 9           00 34                  LDM     EQU     00110100B
10                                  ;MAR <-- (address)
11           00 18                  LMA     EQU     00011000B
12                                  ;(MAR) <-- DOR
13           00 E0                  WMD     EQU     11100000B
14                                  ;(address) <-- DOR
15           00 C0                  WAD     EQU     11000000B
16                                  ;(MAR) <-- constant
17           00 A0                  WMC     EQU     10100000B
18                                  ;microPC <-- (address)
19           00 1C                  JPI     EQU     00011100B
20                                  ;DONE  used HALT,$
21           00 0D                  HALT    EQU     00001101B
22                                  ;DELAY ONE CYCLE
23           00 00                  NOP     EQU     00000000B
24
25   0000                                   ORG   0000H
26   0000   00 00 00 00                     DW    NOP,0
27                                  Rule1:  ; 04H
28                                  ;  DC l:=1
29   0004   00 04 00 01                     DW    LDC, 00001H
30   0008   00 C0 07 DE                     DW    WAD, 007DEH
31                                  ;  DD r:=n
32   000C   00 14 00 04                     DW    LDA, 00004H
33   0010   00 C0 07 E0                     DW    WAD, 007E0H
34                                  ;  DE i:=(l+r)div2
35   0014   00 14 07 EA                     DW    LDA, 007EAH
36   0018   00 C0 07 DC                     DW    WAD, 007DCH
37                                  ;  DI "a[i]":=a[i]
38   001C   00 18 07 EC                     DW    LMA, 007ECH
39   0020   00 34 00 00                     DW    LDM, 0
40   0024   00 C0 07 E2                     DW    WAD, 007E2H
41                                  ;  DC lambda:=1
42   0028   00 04 00 01                     DW    LDC, 1
43   002C   00 C0 00 00                     DW    WAD, 00000H
44                                  ;  Jump to next rule
45   0030   00 1C 00 02                     DW    JPI, 02H
46   0034   00 00 00 00                     DW    NOP, 0
47                                  Rule2:  ; 038H
```

```
   48                                ;  DE r:=i-1
   49   0038   00 14 07 E4                DW    LDA, 007E4H
   50   003C   00 C0 07 E0                DW    WAD, 007E0H
   51                                ;  DE i:=(l+r)div2
   52   0040   00 14 07 EA                DW    LDA, 007EAH
   53   0044   00 C0 07 DC                DW    WAD, 007DCH
   54                                ;  DI "a[i]":=a[i]
   55   0048   00 18 07 EC                DW    LMA, 007ECH
   56   004C   00 34 00 00                DW    LDM, 0
   57   0050   00 C0 07 E2                DW    WAD, 007E2H
   58                                ;  Jump to next rule
   59   0054   00 1C 00 02                DW    JPI, 02H
   60   0058   00 00 00 00                DW    NOP, 0
   61                               Rule3:  ; 05CH
   62                                ;  DE l:=i+1
   63   005C   00 14 07 E6                DW    LDA, 007E6H
   64   0060   00 C0 07 DE                DW    WAD, 007DEH
   65                                ;  DE i:=(l+r)div2
   66   0064   00 14 07 EA                DW    LDA, 007EAH
   67   0068   00 C0 07 DC                DW    WAD, 007DCH
   68                                ;  DI "a[i]":=a[i]
   69   006C   00 18 07 EC                DW    LMA, 007ECH
   70   0070   00 34 00 00                DW    LDM, 0
   71   0074   00 C0 07 E2                DW    WAD, 007E2H
   72                                ;  Jump to next rule
   73   0078   00 1C 00 02                DW    JPI, 02H
   74   007C   00 00 00 00                DW    NOP, 0
   75                               Rule4:  ; 080H
   76                                ;  DD index:=i
   77   0080   00 14 07 DC                DW    LDA, 007DCH
   78   0084   00 C0 00 08                DW    WAD, 00008H
   79                                ;  EX exit
   80   0088   00 0D 00 88                DW    HALT, $
   81   008C   00 0D 00 88                DW    HALT, $-4
   82                               Rule5:  ; 090H
   83                                ;  DE index:=n+1
   84   0090   00 14 07 E8                DW    LDA, 007E8H
   85   0094   00 C0 00 08                DW    WAD, 00008H
   86                                ;  EX exit
   87   0098   00 0D 00 98                DW    HALT, $
   88   009C   00 0D 00 98                DW    HALT, $-4
   89   00A0                            END

*****  S Y M B O L I C   R E F E R E N C E   T A B L E  *****

HALT  = 000D    JPI   = 001C    LDA   = 0014    LDC   = 0004
LDM   = 0034    LMA   = 0018    NOP   = 0000    Rule1   0004
Rule2   0038    Rule3   005C    Rule4   0080    Rule5   0090
WAD   = 00C0    WMC   = 00A0    WMD   = 00E0

    LINES ASSEMBLED :    89          ASSEMBLY ERRORS :    0
```

Listing 1. Microcode Assembler Output (Cont.)

5 COMPILING DECISION TABLES

A goal of the FMC compiler is that the user require no combinational logic design experience. As Figure 17 illustrates, the compiler's task involves inputting a high-level source program (.SRC) and (1) generating an assembly language source program (.ASM) for the minimal processor's (mP) microcode assembler and (2) generating the FPGA source programs (.PDS) for each FPGA in FM as required to perform the expression computations. An example assembly language source file compiled by the assembler for the binary search program appears in Listing 1 in the previous section. Section 3 describes the structure and content of an FPGA source file. Listing 2 contains the compiler listing for the binary search program.

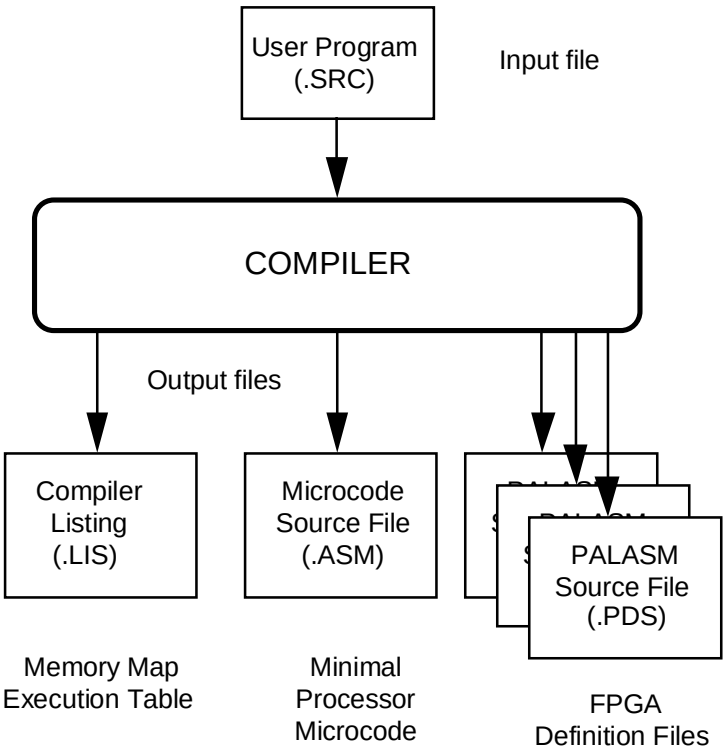

Figure 17. The FMC Compiler

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. A FMC compiler drop-down menu allows selecting 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 executed to complete the process, generating the FMC FPGA load module.

Listing 2. Decision Table Compiler .LIS Output

```
Program: BINSRCH    12-20-1994 at 14:10:09

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

Input File
==========

var n : integer;                'array size
    v : integer;                'search key
    index : integer;            'returned index
    a : array[1000] of integer; 'array to be searched
    i, l, r : integer;          'index, left, right
    "a[i]" : integer;           'temporary scalar
dtbegin
lambda =           | 0 1 1 1 1
v < "a[i]"         | - T F F -   'test to update l or r
v = "a[i]"         | - F F T -   'test if index found
l > r              | - F F F T   'test for loop exit
-------------------+----------
l := 1             | X - - - -   'set left boundary index
r := n             | X - - - -   'set right boundary index
r := i - 1         | - X - - -   'decrement right boundary
l := i + 1         | - - X - -   'increment left boundary
index := i         | - - - X -   'found! set return index
index := n + 1     | - - - - X   'key not found
i := (l+r) div 2   | X X X - -   'calculate new middle
"a[i]" := a[i]     | X X X - -   'update scalar for test
exit               | - - - X X   'exit
lambda :=          | 1 - - - -
dtend.

Compilation Statistics:
  5 rules, 4 conditions, 10 actions
Functional Memory: 2030 bytes
FPGA I/O: 7 inputs, 6 outputs
Microcode: 40 lines MC
FPGA PALASM: 183 CLBs (estimated)
             1 chip

Functional Memory Map
=====================

     Assigned Chip  Address or Value
Name      *Type  |  | #Bits  Expression
-----         |  |  |     |  ----------
lambda ...... R  1  0000  2
@Rule ....... P  1  0002  7
n ........... R  1  0004  13
v ........... R  1  0006  13
index ....... R  0  0008  13
```

```
a ........... C  0  000A  0
i ........... R  1  07DC  13
l ........... R  1  07DE  13
r ........... R  1  07E0  13
"a[i]" ...... R  1  07E2  8
i-1 ......... E  1  07E4  13  i - 1
i+1 ......... E  1  07E6  13  i + 1
n+1 ......... E  1  07E8  13  n + 1
(l+r)div2 ... E  1  07EA  13  ( l + r ) div 2
@a[i] ....... A  1  07EC  14  a [ i ]

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


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

 Adr  Statement        mP  Dest  Src   Cyc
 ---  ---------        --  ----  ----  ---
Rule1=/L1*/L0
 004: l:=1             DC  07DE  0001   2
 00C: r:=n             DD  07E0  0004   2
 014: i:=(l+r)div2     DE  07DC  07EA   2
 01C: "a[i]":=a[i]     DI  07E2  07EC   3
 028: lambda:=         DC  0000  0001   2
 030: goto @rule       JI  0002         2
Rule2=/L1*L0*C2*/C3*/C4
 038: r:=i-1           DE  07E0  07E4   2
 040: i:=(l+r)div2     DE  07DC  07EA   2
 048: "a[i]":=a[i]     DI  07E2  07EC   3
 054: goto @rule       JI  0002         2
Rule3=/L1*L0*/C2*/C3*/C4
 05C: l:=i+1           DE  07DE  07E6   2
 064: i:=(l+r)div2     DE  07DC  07EA   2
 06C: "a[i]":=a[i]     DI  07E2  07EC   3
 078: goto @rule       JI  0002         2
Rule4=/L1*L0*/C2*C3*/C4
 080: index:=i         DD  0008  07DC   2
 088: exit             EX               2
Rule5=/L1*L0*C4
 090: index:=n+1       DE  0008  07E8   2
 098: exit             EX               2
 0A0:
```

Listing 2. Decision Table Compiler Output (Cont.)

6 CONCLUSION

This work has provided us with two important conclusions. First, functional memory can greatly reduce the number of steps in programs where the ratio of computation steps to load-stores is relatively high. For the binary search example, the inner loop of the Pascal version when executed on an efficient microprocessor takes 20 steps whereas on a decision table using functional memory the same is accomplished in only 9 steps. More dramatic results were reported in [11] for an image processing application.

The second important conclusion of this research is the demonstration that when *all* expression computation is performed in the functional memory, then the necessary processor instruction set reduces to a simple set of moves and jumps. This allows the processor to be small and streamlined to less than what is necessary for typical RISCs because no general purpose operand registers or ALU are needed in the processor proper. With such a simple processor (which we call a "minimal" processor), it becomes quite feasible to implement at least some of the expressions within the same chip as the processor when the processor itself is also a FPGA. Moving program address computation into the processor chip would eliminate the jump step required at the end of each rule. Moving array address computations into the processor chip would eliminate the extra step necessary for fetching the array element address from functional memory, before the element itself can be fetched. Moving the remaining expressions into the processor chip would eliminate the load step of an assignment statement.

Our prototype functional memory computer has allowed us to investigate these and other reprogrammable FPGA based architectures. It has been useful in illustrating both the promise and limitations of functional memory coprocessing systems. Many of our present limitations are associated with our use of older generation off-the-shelf technology. We believe the underlying principles behind FM will prove valuable in a variety of ways, and that perhaps FM will benefit from and motivate new technologies. Our research is presently focused on developing a functional memory coprocessor for PCI compatible computers using Xilinx 4,000 series parts, and adapting our Windows based decision table compiler to the new machine.

REFERENCES

[1] S. Trimberger, "A reprogrammable gate array and applications," *Proceedings of the IEEE,* vol. 81, no. 7, pp. 1030-1041, Jul., 1993.

[2] L. Agarwal, M. Wazlowski, and S. Ghosh, "An asynchronous approach to synthesizing custom architectures for efficient execution of programs on FPGAs," *Proc. 23rd International Conference on Parallel Processing*, vol. 2, pp. 290-294, Aug., 1994.

[3] D. Thomae, and D. Van den Bout, "Automatic circuit partitioning in the Anyboard rapid prototyping system," *Microprocessors and Microsystems*, vol. 16, no. 6, pp. 283-290, 1992.

[4] "S-bus-based transformable computing system," *Military & Aerospace Electronics*, PennWell Publishing Company, Oct., 1994.

[5] R. Halverson, Jr., *The functional memory approach to the design of custom computing machines*, unpublished doctoral dissertation (available from UMI), University of Hawaii, Aug., 1994.

[6] R. Halverson, Jr. and A. Lew, "Programming with functional memory," *Proc. 1994 International Conference on Parallel Processing* Vol I, pp 85-92, or Technical Report ICS-TR-94-25, Information and Computer Sciences department, University of Hawaii at Manoa, Aug., 1994.

[7] R. Halverson, Jr. and A. Lew, "Programming the Hawaii parallel computer," *FPGA '94. ACM 2nd Intl. Workshop on FPGAs*, or Technical Report ICS-TR-94-24, Information and Computer Sciences department, University of Hawaii at Manoa, Feb., 1994.

[8] XILINX, *The Programmable Logic Data Book – 1994*, XILINX, Inc.: San Jose CA, 1994.

[9] A. Lew, "On the emulation of flowcharts by decision tables," *Communications of the ACM,* vol. 25, no. 12, pp. 895-905, Dec., 1982.

[10] R. Sedgewick, *Algorithms*, 2nd. ed., Addison-Wesley, Inc.: New York, p.198, 1988.

[11] R. Halverson, Jr., and A. Lew, "FPGAs for Expression Level Parallel Processing," Technical Report ICS-TR-94-26, Information and Computer Sciences department, University of Hawaii at Manoa, Dec. 1994.