Georgia Institute of Technology
School of Electrical and Computer Engineering

elRoy

A Systolic Processor Array

CompE 4510 Senior Design

Winter 1995

Darrell Stogner
Craig Ulmer

Table of Contents

Background, Purpose, and Status

Traditionally, the rule of thumb for creating high speed computation bound programs falls to the saying DM/ND: Don't Multiply, Never Divide. Even with optimizations in hardware computation algorithms, the ALU's slowest operation is inevitably a form of multiplication or division. Since multiplications and divisions are necessary for a large percentage of computational problems, it would be beneficial to arrange the slowest operations in a way that the events could take place in parallel in order to minimize delay.

Systolic architectures allow such parallelism by breaking up computations, and arranging their execution in a way that costly operations occur at the same time. Although the amount of complexity in controlling the overall system increases largely, the benefits of parallel designs are worthy enough to warrant the added difficulty.

The purpose of this project is to exploit systolic architecture to find a faster means for working through high computation mathematical functions. The applications at which this project is geared towards are generally found within the Digital Signal Processing (DSP) domain of problems. The goals for this project include three specific applications that prove the array's design and advantages:

1. Discrete Convolution operation
2. Matrix and Vector Multiplication
3. Matrix and Matrix Multiplication

The systolic array of specialized processors designed in this project should be thought of as a side system much like a specialized coprocessor. For an actual implementation of this project, a main controller processor would perform normal computer tasks, while delegating specific computation tasks to the systolic array. This project uses a "simple" multi-cycle processor to handle the flow of operation and control of the systolic array.

All components of the system have been tested and synthesized to hardware. Due to the size of the project, the hardware emulators were not able to hold enough of the design to physically test our implementation. Since we were mainly interested in the theory of the design, we performed several rigorous test programs in the Synopsys VHDL environment to determine if our design theories were correct. While the tests were not gate timing simulations, we were able to prove that our theories of design were correct and valid. While it would have been desirable to see the design fully running in hardware, we consider the concept to be more important than a physical implementation.

Fast Statistics

Overall

System:	elRoy
Form:	Systolic Processor Array, Additional multi-cycle control processor
Target Applications:	High Computational Programming - DSP, Convolutions, Matrix
Data Width:	16-bit
Instruction Width:	32-bit
VHDL Design:	Structurally - Almost entirely to gate level
Synthesis Status:	Entirely synthesized
Hardware Status:	Too large to fit anything of interest in emulator boards
Testing Status:	VHDL Tested, All application goals assembled and tested

Control Processor

Function:	Flow Control, State Machine, Normal Ops
Register file:	8 16-bit registers
ALU:	Standard ops - Add,Sub,Stack,Or,Xor,Cmp,Copy
PC OPs:	Jmp, Jsr, Rts, Ja,Je

Systolic Array

Array Dimension:	Linear
Flow:	Uni-directional
Array Form:	Adjustable -- Contains mutated Moving Results and mutated Fan Out
Maximum Cells:	32 (Limited due to 16-bit architecture)
Data Inputs:	Serial Load and Broadcast Load
Cell Control:	Single Microcode Instruction, Control Signals
Features:	Easily expandable

Cell

Features:	Flexible datapath based on Microcode Instruction Adjustable FIFO Queue for inserting bubbles into array pipeline Cells are all identical structurally - easier to mass produce

I. Theory

Systolic Array Theory

There are several mathematical operations that require multiple calculations for each result. These calculations are often similar and can be done in parallel for a faster result. A systolic processor array takes advantage of this fact and breaks up the monotonous task of handling data by allowing several specialized processors to munch on individual components of the data at the same time.

Perhaps the easiest example to view the beauty of a systolic array comes from the calculation of a matrix-vector multiplication. For each answer, several values must be multiplied and accumulated. A simple approach to building a system that implements this view is the fanout design (figure 1). For each result we multiply an entire row of the matrix by the vector's column, then add everything together with a specialized adder. Clearly, all of the multiplications for the particular result occur at the same time, minimizing the overall system's multiplication delay.

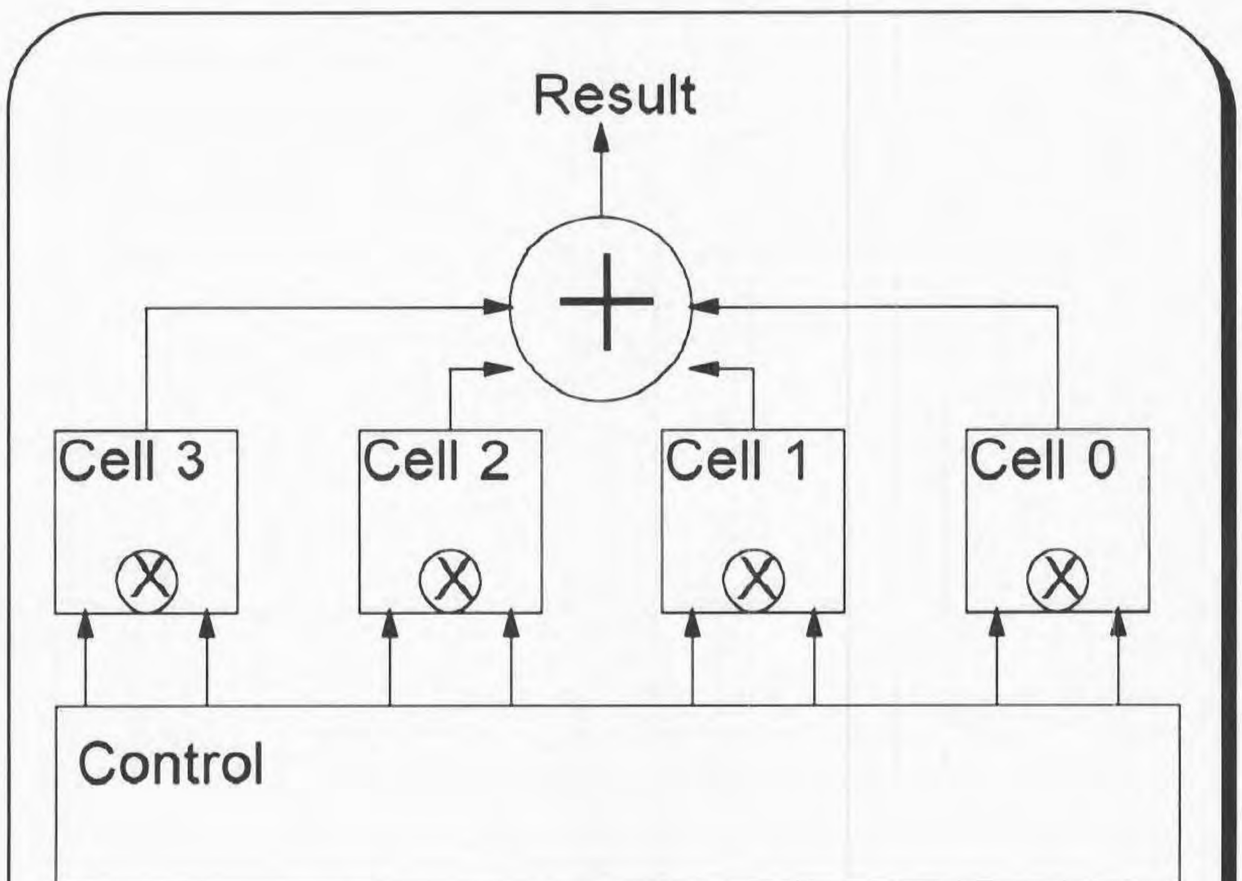


Figure 1: Fanout Design

Although easier to build and track, the fan out approach suffers from a weakness in the result adder:

1. The adder will turn into a large amount of logic since it must add more than two numbers together.

2. The array will have to have a fixed number of cells. Since this prevents expandability, the number oc cells cannot be tailor picked to fit job requirements.

The next strategy for designing the array consists of breaking the adder up into small stages that can be implemented within each cell. In essence, this pipelines the cell array, allowing results to be generated as they work their way through the array. This topology is refered to as a moving results system(figure 2), since results move with input data through the cell array.

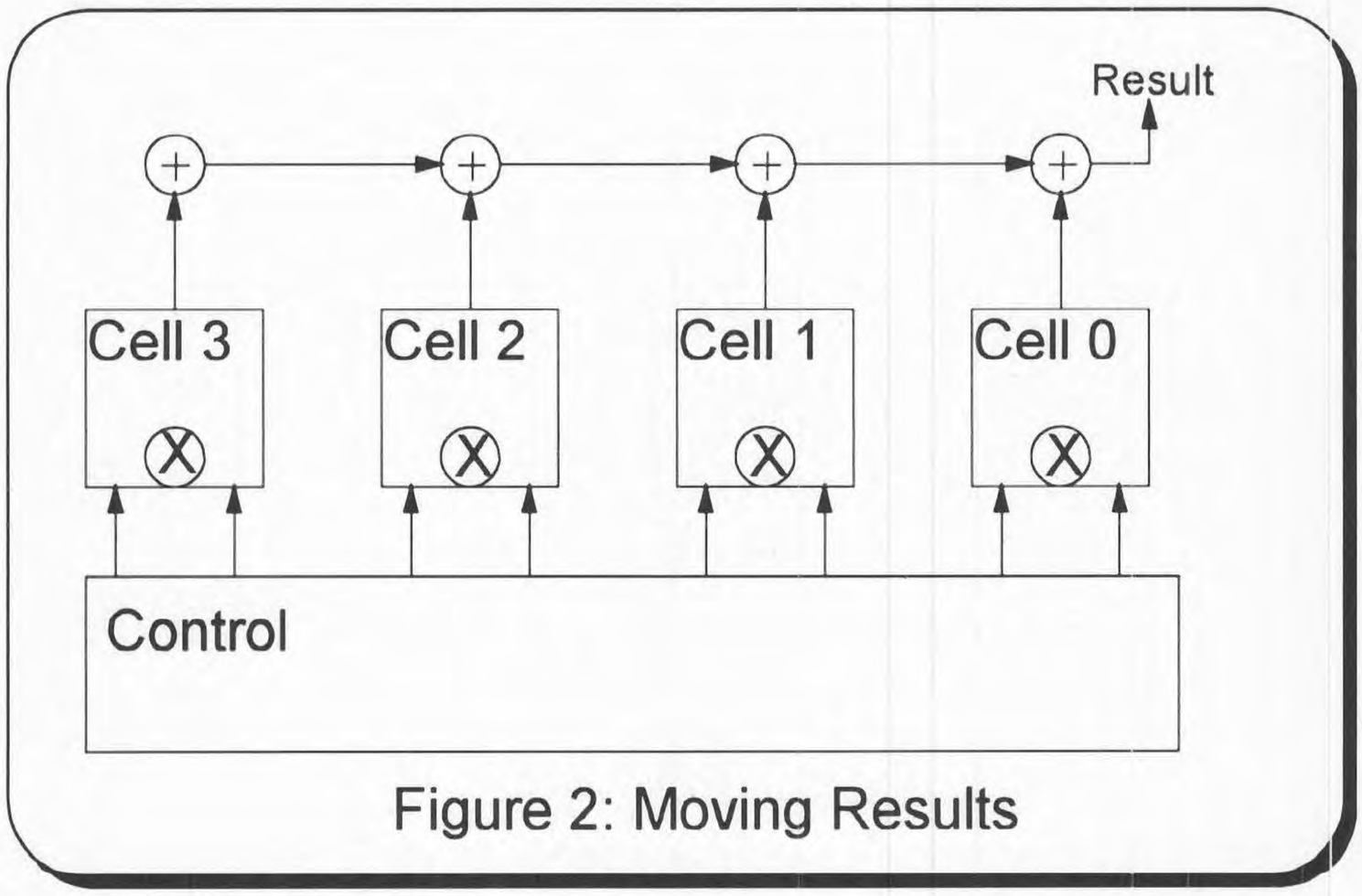


Figure 2: Moving Results

This topology requires more control effort, but removes the problems associated with the fan out approach. By manipulating how values are passed to the cells, we can implement a system that works as a long pipe and generates an output for each input. In this manner, we can easily expand the array to as many or as few cells as we require. An evolved version of the Moving Results implementation serves as the heart of elRoy's computational heart.

Other topologies for array configurations involve multidimensional arrangements of cells. Many of these designs involve careful timing models with bi-directional communication links between cells. Although the key to improving systolic array performance is to increase the dimension of the array, the practical features of parallel processing can be exploited with the simpler one dimensional array.

Matrix-Vector Multiplication Theory

The multiplication of a matrix by a vector is significantly sped up by the use of a systolic array of processors. The parallelization of the process is limited by the number of processors in the link, and typically by the speed of the interface. The design chosen here lends itself to a particularly fast computational procedure, and is therefore limited only by the number of processors. In short, elRoy's architecture is highly suitable to matrix vector operations.

The first step is to understand how the matrix will be stored in the machine. The typical initial reaction is to store it row by row in an array. After consideration of the multiplication process, it was discovered that if the numbers were stored in the array column by column, instead of row by row, the multiplication could be computed with architecture compatible with the convolution operation. Once this had been determined, the algorithm was obvious:

Note: This assumes an M x N matrix, and an N length vector. See the Vector Multiplication Diagrams.

Initialization: All accumulators and all registers must be set to zero.

1) The k^{th} element of the vector is parallel-loaded into the Cell register.
2) The k^{th} column of the matrix is loaded into the inputs of the systolic array sequentially.
3) The product of the two terms is calculated, and the result is added into the value held in the accumulator.
4) If (k == N) goto 5, else goto 1.
5) Note that at this point the values in the each accumulator correspond to one of the elements in your output vector. They are propagated through the accumulator into memory.

Note that the algorithm above assumes that you have at least M systolic processors.

In short, the vector multiplication algorithm works by accumulating answers in each cell. The procedure is illustrated in figure 3. The arrangement of loading the cells is described in figure 4.

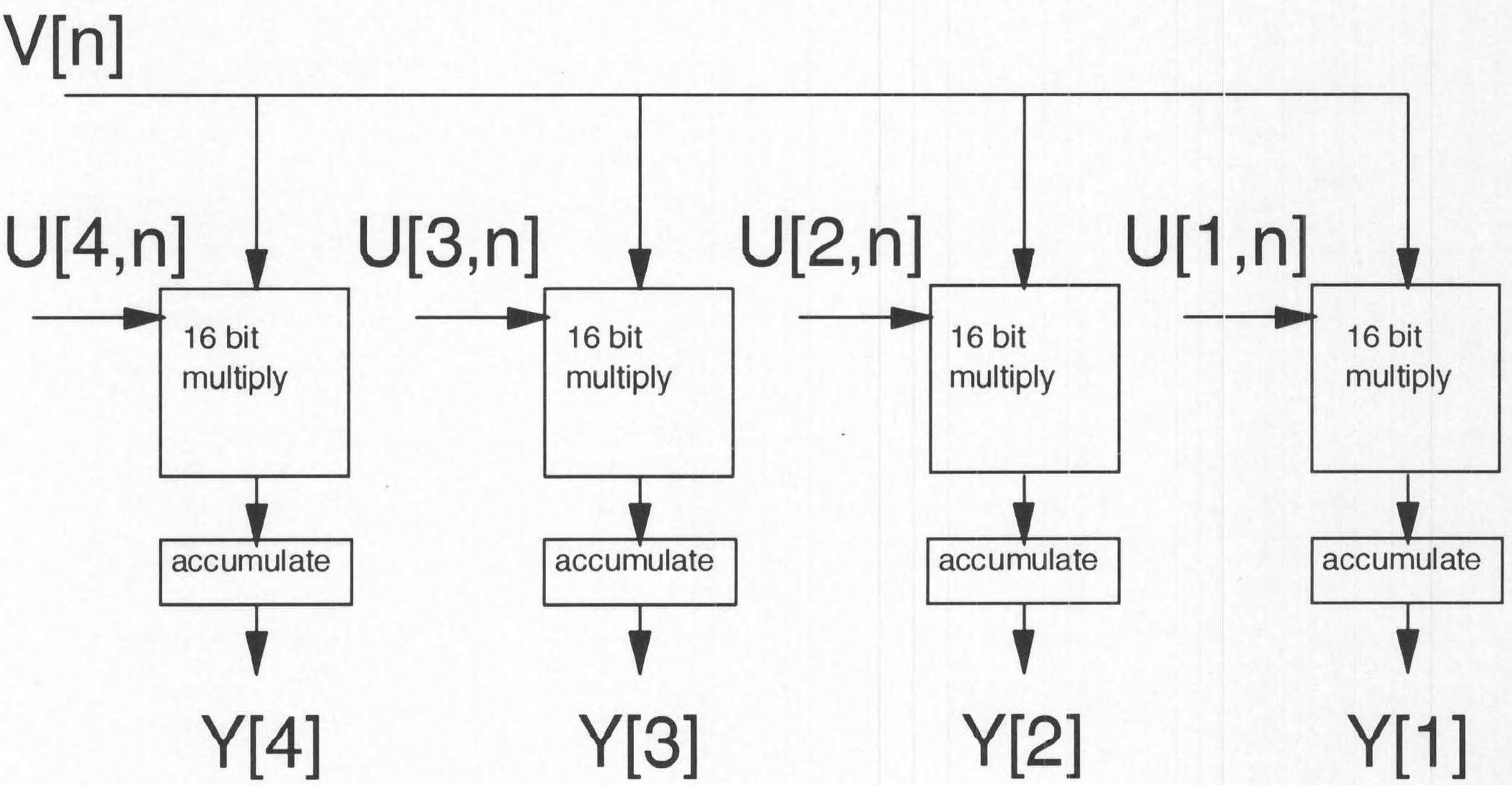
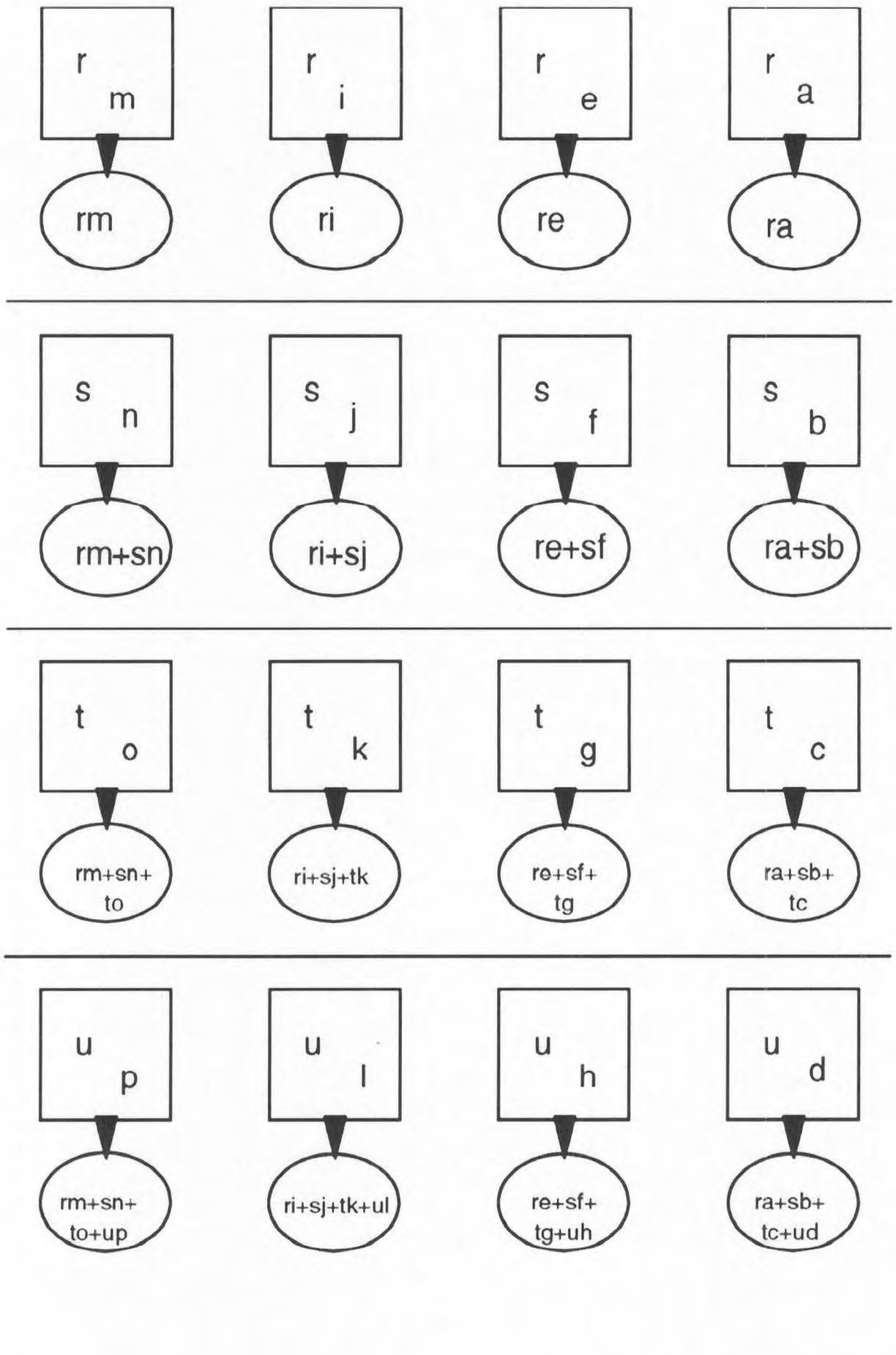


Figure 3: Vector Multiplication Configuration



Vector Multiplication

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} r \\ s \\ t \\ u \end{bmatrix}$$

Figure 4: Vector Multiplication

If the number of rows in the matrix exceeds the total number of processors, then a control process must exist to process each column in chunks. This can be conveniently handled by the operating system. It will be necessary for the OS to efficiently break up the columns into blocks of manageable size, and to keep track of where each set of results needs to be stuck in the output vector. Another problem best handled by software is the fact that the final outputs as they are read off will be in reverse order. It is possible to place the results on a stack, and then pop them off one by one, but this is a needless waste of processor time. A good OS can place the output results into the correct memory address. Generally speaking, given a system with X systolic processors, and a matrix with M rows and N columns, the number of calculation steps to obtain the final product is:

$$Trunc(\frac{M}{X}) \bullet N$$

The total speed is also affected by the time required to propagate the column elements through the processor array. This gives a final calculation time of:

$$[(Trunc(\frac{M}{X}) + 1) \bullet N \bullet cycle\_speed] \bullet [M \bullet propagation\_speed]$$

Matrix-Matrix Multiplication

The Matrix-Matrix multiplication operation is essentially the Matrix-Vector multiplication problem placed within a loop. If we treat the columns of the second matrix as individual vectors, we can easily apply the above operation several times over to produce the proper results.

Discrete Convolution Theory

To provide efficiency in computation as well as compatibility with other processor functions, the philosophy behind the discrete convolution operation is to treat the array of cells as a pipeline. Each cell focuses on a particular stage in the pipeline, while data is fed into the array serially. The result is an output for every cycle once the pipeline is fully initialized. In order to build the convolution, the fan out implementation was first examined.

In the fan out design, each of the cells is preloaded with a static value of H, and input X is fed serially from the left. In each cell, the shifting value X is multiplied by the cell's static value of H. The results of all cells are added together to produce one value of the output.

The range of H for this implementation is unfortunately limited to the number of cells in the system, assuming the convolution is a one pass algorithm. As well, for each zero value in the H equation, an entire cell must be wasted on a calculation that will always result in zero. Take for example the following equation.

$$h[n] = A\delta[n] + 0\delta[n-1] + 0\delta[n-2] + B\delta[n-3]$$

Implementing it on the initial design, the first cell would be loaded with A, the second 0, the third 0, and the fourth B. The zero coefficients make the second and third cells perform operations in which the result is already known.

In order to make better use of the cells, it is more efficient to simply place bubble stages in the data stream so zero terms do not waste cell computation time. To do the bubbling, a FIFO queue is placed in each cell that would delay the input from passing on to the next cell before the proper time has passed. The queue would have to be setup for each time the overall convolution is performed, but allows for a greater range of flexibility in the system. This idea is similar to placing a variable amount of NOP's in microprocessor code in order to fix timing sequences.

An acceptable approach to the problem of upgradability is to do the summations in small doses as the terms become available. elRoy's convolution strategy is to pass a running sum of terms through each stage and provide each stage with the appropriate values at the correct times.

The following equations illustrate the convolution process.

$$x[n] = A\delta[n] + 0\delta[n-1] + B\delta[n-2] + C\delta[n-3] + D\delta[n-4]$$

$$h[n] = E\delta[n] + F\delta[n-1] + G\delta[n-2] + H\delta[n-3]$$

$$y[n] = x[n]*h[n]$$

Since h[n] of the convolution can be thought of as a device that shifts and scales the output by h[n]'s terms, the following table represents the output.

y[0] =	EA						
y[1] =	E0	+	FA				
y[2] =	EB	+	F0	+	GA		
y[3] =	EC	+	FB	+	G0	+	HA
y[4] =	ED	+	FC	+	GB	+	H0
y[5] =			FD	+	GC	+	HB
y[6] =					GD	+	HC
y[7] =							HD

From this table, it is clear that there involves a good bit of symmetry in the output of the convolution. There are two basic keys to building the system. The first fact is that each column of the table has the same corresponding coefficient of h[n]. The second fact is that the coefficients of x[n] are found in order vertically for each column. This brings about the idea that the h[n] values may be serially loaded, and that if the operations were timed correctly, the values of x[n] could be broadcast to all of the cells.

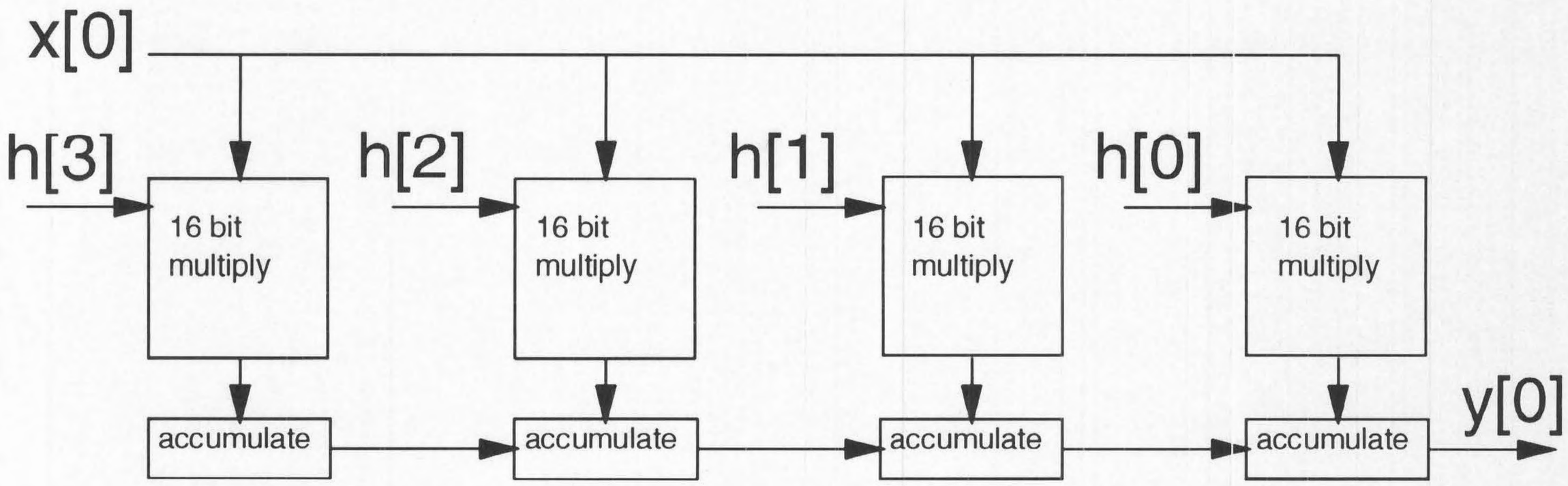


Figure 5: Discrete Convolution Configuration

Discrete Convolution

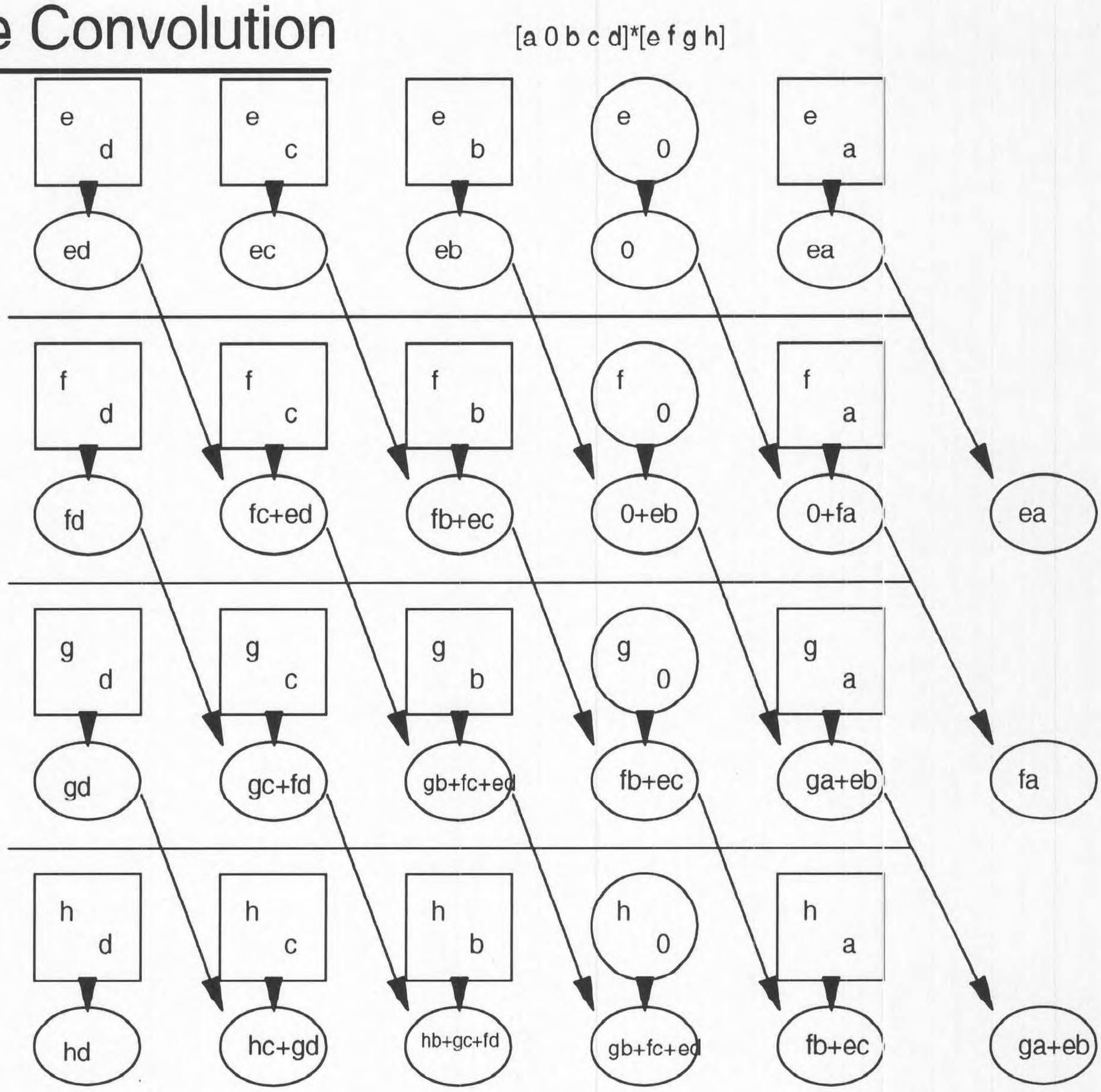


Figure 6: Convolution Pipe

We apply the facts of the convolution to the following procedure, as illustrated in figures 5 and 6.

1) Zero all of the accumulators.

2) Slide x[n] into the dynamic register of the cells so that the cells will appear with A in the far right cell, and D in the far left cell. This is done by serial insertion.

3) Set up the zero bubbles. For this example this is done by instructing the second processor from the right to add one extra wait state (via the fifo queue) before it puts out and output.

4) Load the first value of h[n] (or E in this case) in parallel to all of the cell's static register.

5) Multiply the static register by the dynamic register in each cell, and add it to the accumulator of each cell.

6) Shift all of the accumulator values one step to the right. (i.e., into an accumulator or fifo)

7) Repeat from step 4 until all operations are done. This should be monitored by the control unit. The number of results = length(x) + length(h) - 1.

Since x[n]*h[n] = h[n]*x[n], the sequences h[n] and x[n] could also exchange roles in the above scenario. Ideally, the sequence that is longer, but still smaller than the number of cells should be held in the cells (x[n] in the previous scenario).

The largest benefit of the system is that an output is cranked out every cycle without a need for one large adding system. Additionally, the hardware is compatible with the other implemented matrix operations.

II. Architecture

System Design and Architecture

The architecture used to implement elRoy consists of a main controlling processor, a systolic linear array of cells, and a memory unit. Data values within the system are all 16-bit, two's compliment, integer numbers.

As mentioned earlier, the cells in the systolic array follow a form of the moving results architecture. However, since different operations require different architectures, elRoy's cells must allow a flexible and controllable datapath. The elRoy system achieves this by passing microcode words to cells as changes in the cell control are required. The main processor controls not only the configuration of the cells, but also the entire data flow for the overall system.

Main Processor(figure 9)
The main processor is a small scale processor that interprets code and controls the whole system. It contains an ALU, an register file with 8 registers, a state machine that controls the system, and logic to handle bus arbitration and data flow. All components in the main processor were designed down to the gate level, except for the state machine. The state machine is a simple model with the normal Fetch,Decode,and Execute stages, along with additional stages for some cell operations.

Cell Array
The cell array is a linear arrangement consisting of up to 32 identical cells. Each cell has a bank of dip switches determining the cell's local 5-bit address. An extra parallel load bit in the address indexing scheme allows the cells to all accept information from the same source.

Individual Cell(figure 8,10)
An individual cell contains the logic to determine if it is being spoken to by the main processor, as well as the ability to interpret micro-code instructions. The cell contains two registers RA and RB that hold the 16-bit data values that are to be multiplied. The RA register serves as a means of serial loading from the previous register, while the RB register reads 16-bit data values broadcast to all cells. The result of the multiplication can be accumulated, added to the previous cell's accumulator value, or simply passed along to the next cell. A FIFO queue can additionally be used to add delay stalls to the passing of results. The FIFO queue has an adjustable value of up to 7 stalls between cells. Components for the cell were all originally written exclusively to the gate level. To optimize the synthesis process, the adder and multiplier units were changed to simple VHDL models.

Memory
Memory consists of two 32-bit data pathways to the main processor, as well as some standard memory control signals. While the original model consisted of a signal 32-bit bi-directional data pathway, the model was changed to two uni-directional 32-bit pathways for simplicity and clarity. The memory can only access even addresses, since elRoy only deals in 16-bit data values.

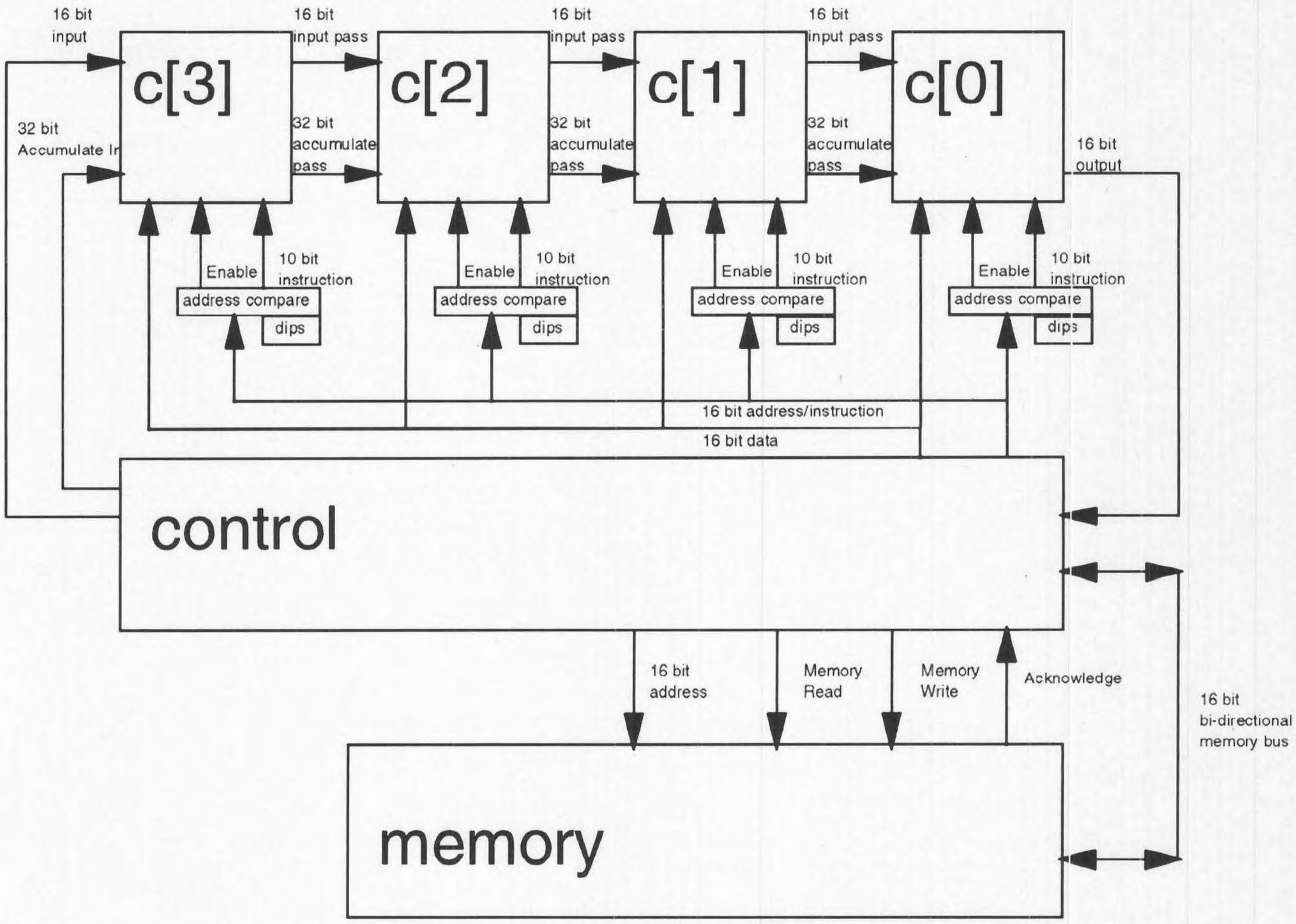


Figure 7: elRoy Flow Diagram

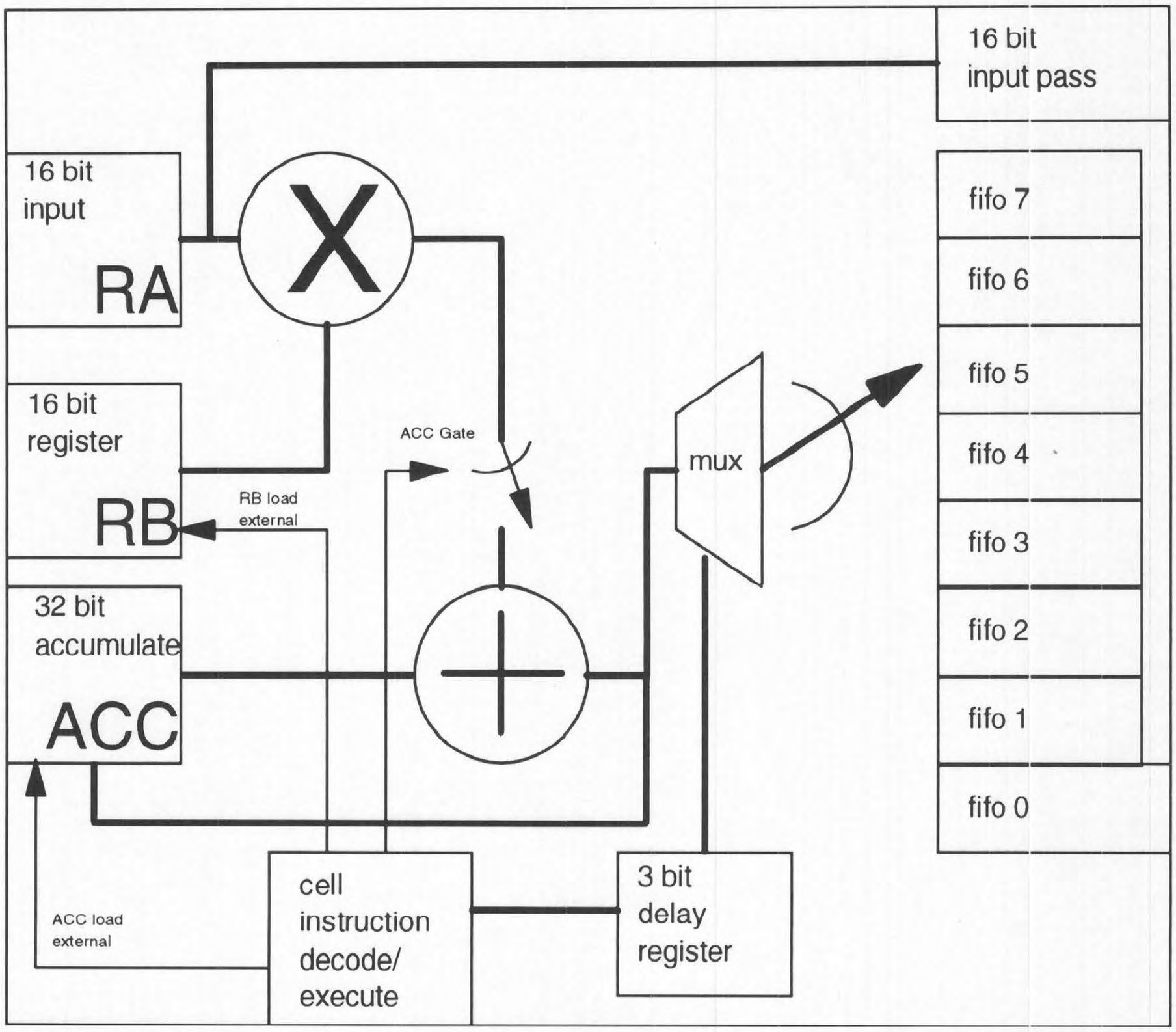


Figure 8: Individual Cell

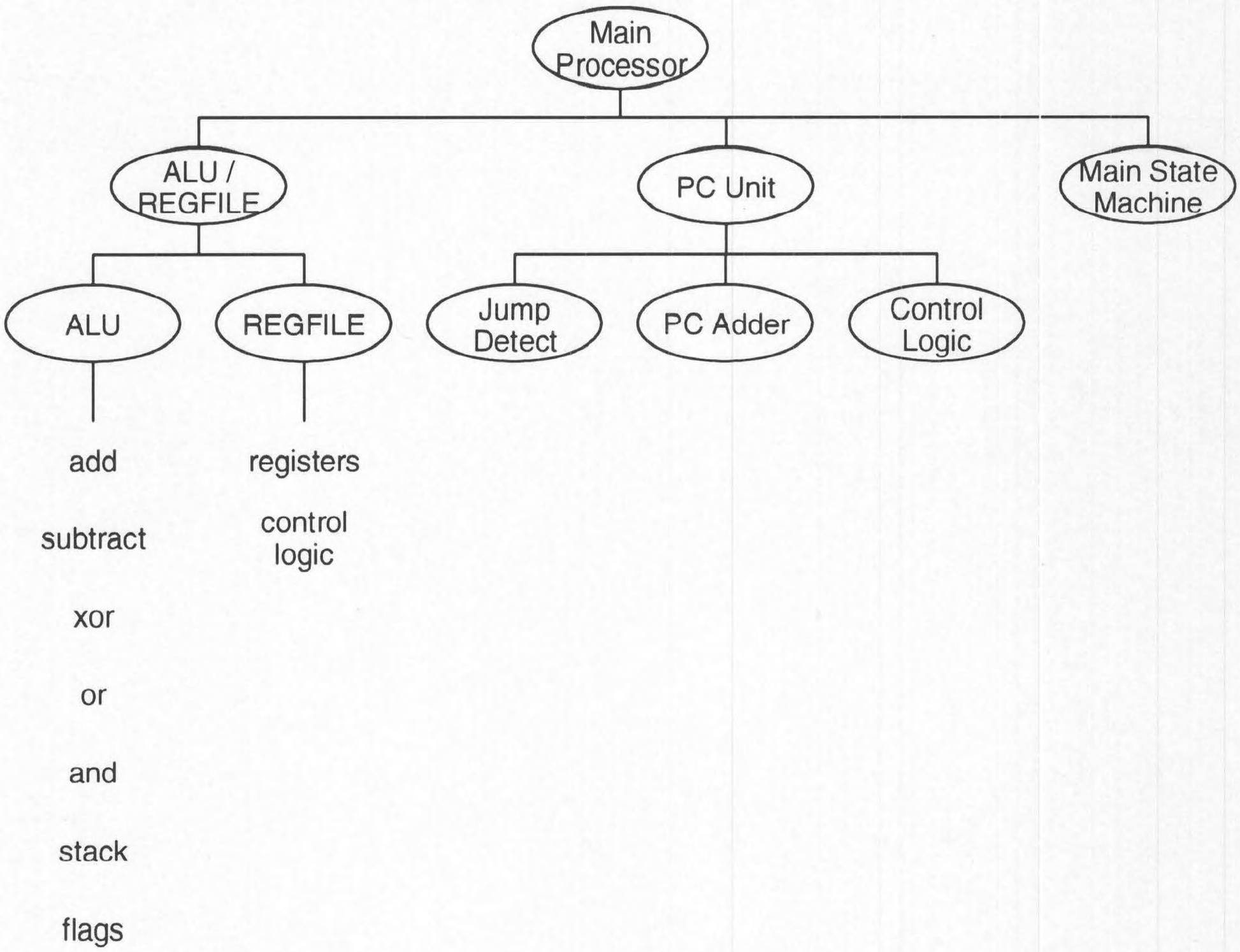


Figure 9: Main Processor Components

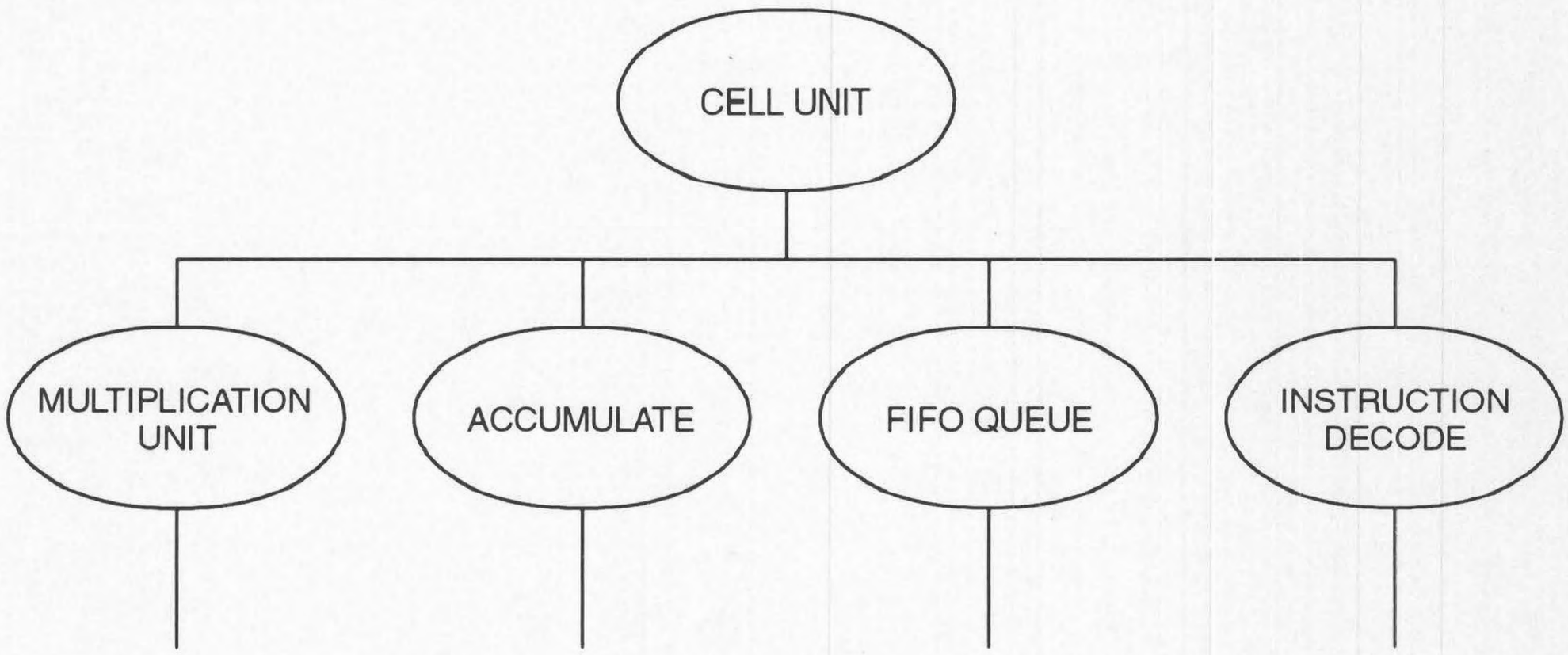


Figure 10: Cell Components

III. Synthesis

Synthesis and Partitioning

The synthesis process involves taking a high level VHDL description of an ASIC design and translating it into gate-level netlists. The gate-level netlists are modeled in various technology libraries. These net-lists can then be exported to hardware for the purpose of testing. The Synopsys VHDL Compiler can be used to synthesize ASICs.

The synthesis can be optimized for a variety of factors such as speed, area and power consumption. Constraints are used to control which factors the compiler emphasizes. Speed or timing constraints are used to specify maximum delay through a particular path (usual the critical path). The compiler will usually attempt to get as close as possible to the specified goal. Area constraints specify the maximum amount of space a design is allowed to take up. This is usually given in term of a total gate count. Power constraints refer to the maximum amount of power the ASIC can dissipate.

Synthesis libraries contain information that the compiler needs in order to best create a netlist for a given design. Technology libraries contain much information, including the area, propagation delay and power consumption of a given cell. This information allows the compiler to make the correct choices based on the operating constraints.

In elRoy, the synthesis process was *relatively* straightforward. Our initial design was in most places taken down to the gate level. The one exception, the shift-add multiplier, was replaced by the synthesis multiplier for speed purposes. In most cases this made synthesizing elRoy easier. Still, since this was such a new process problems arose that required resynthesizing.

After running through the synthesis tutorial, we started off by synthesizing the lower level components using the design compiler. We worked our way up the design slowly until the full design had been synthesized. At this point, much time was spent attempting to translate the synthesized netlists back into the *sge* environment. Unfortunately the symbol library for the Xylinx parts was corrupted.

It was discovered that the xblocks synthetic library was not supported yet by the partitioning software, so the design was resynthesized. It was then discovered that using the FPGA compiler would generate better results. Unfortunately an old .synopsys_dc.setup was used and the xblocks synthetic library was still in the file. The design was synthesized yet again using the FPGA compiler with no xblocks.

At this point a week was spent mopping up the original design (a state machine had been added, and much debugging had taken place since the beginning of the quarter). The new design was then resynthesized. The partitioning software had not yet been installed, so some time was spent generating *.mra* and *.sim* files from the synthesized design, and then testing these files. Only limited testing was done before the partitioning software was installed (i.e. the multiplier and adder in the cell).

Another unfortunate problem was found at this point. The naming convention used by Xylinx for the gate level parts was found to be the exact same naming convention used by us (part name, inputs and outputs). Using search and replace on all our VHDL files fixed this problem, and the design was resynthesized yet another time.

We then used silicon concepts partitioning software. The software essentially takes a netlist description of gates and translates it into an internal format. The netlist is then run through prepartitioning and partitioning in order to isolate all of the structures in the hardware that require special handling. You then 'create hardware' which pops up a description of the zycad box. The partitioned hardware can then be assigned to zycad cards. The design is then routed, and finally compiled.

MANY problems were encountered with the partitioning hardware. The most crippling problem encountered was the translator, which had trouble reading in the design. Slightly over two weeks were spent fixing problem after problem, until the thing would finally read in. Unfortunately at this point we realized just how large our design had become. The full array was out of the question. In fact only the main processor OR one cell could fit at any given time. Furthermore, the main processor used tri-states, which the router did not seem to like. Also, the size of the multiplier was too large for one chip. If, however, the design was broken down, it broke into gates. This meant too many I/Os. We were finally able to get the thing to actually fit into one chip. The routing and compiling were done overnight in the background, and the necessary files to go to the zycad boxes sit on our account, which is where the project stands now.

In general the synthesis tools were excellent. Synopsys did an excellent job. The compiler required a large address space, but that was not a problem once our accounts on the Suns were activated. The silicon graphics software was rather poor. We got the impression that we were working with a beta version (for instance, multiple error messages that differed only in grammar would be returned). The translator was simply awful. The software looked like it was written for a Macintosh.

IV. Programming

Programming elRoy

Since the elRoy system contains multiple processors, there are several issues that users must face when writing programs to control elRoy. For the average user, elRoy provides a minimal assembly instruction set that handles the challenging task of concurrency within array tasks. However, elRoy also provides a means of programming at a lower system level for advanced users with specific needs. This allows users both flexibility and security in using elRoy today and in the future.

Overall Strategy

The elroy system was designed to appear as a single unit to the programmer. While program execution is sometimes handed over to the processor array, execution all appears to take place within the main processor. The user has standard programming hardware available such as a register file, an ALU, a stack, and subroutine support. Additionally, the main processor can be used to set up the systolic processor array. Cells in the array are configured through microcode words routed from the main processor.

Since configuring the cells in the systolic array is often a tedious and challenging task, commonly used cell commands are predefined in the assembly language to aid programmers. However, the user still has the ability to configure the cells by hand as necessary. While the microcode words are generally difficult to program, it is important that users be able to manipulate the hardware in order to allow elRoy to be adapted to fit future challenges.

Assembly Instruction Format

The assembly instruction is broken into two 16 bit fields: the instruction resides in the top 16 bits and any data values exist in the lower 16 bits.

Instruction Fields

Op Code	Destination	Source 1	Source 2	Data Value
4 bits	4 bits	4 bits	4 bits	16 bits

The destination and source fields represent both the registers in the main processor as well as data paths to particular cell busses. There are eight registers R0 through R7 within the main control unit. These registers are general purpose read/write registers that the ALU and memory have access to.

There are also five special registers that control cell functions.

ACCIN:	Accumulate In	(Read only)	: Uses the result of the array as the source for writing to memory
ACCOUT:	Accumulate Out	(Write Only)	: Loads the Accumulate Out data register with data value specified
CINST:	Cell Instruction	(Write Only)	: Writes data value to the cell instruction register
RA:	RA Bus	(Write Only)	: Pipe data value to RA input of left cell
RB:	RB Bus	(Write Only)	: Pipe data value to RB bus

Assembly Language Instructions

ALU / Register Operations:

XORR:	(destination)	=	(source 1)	XOR	(source 2)
ADDR:	(destination)	=	(source 1)	+	(source 2)
SUBR:	(destination)	=	(source 2)	-	(source 1)
COPYR:	(destination)	=	(source 1)		
ANDR:	(destination)	=	(source 1)	AND	(source 2)
ORR:	(destination)	=	(source 1)	OR	(source 2)
NOP:					

ALU / Data Value Operations

XORD:	(destination)	=	(source 1)	XOR	Data Value
ADDD:	(destination)	=	(source 1)	+	Data Value
SUBD:	(destination)	=	(source 1)	-	Data Value
ANDD:	(destination)	=	(source 1)	AND	Data Value
COPYD:	(destination)	=	Data Value		
ORD:	(destination)	=	(source 1)	OR	Data Value

Memory Operations:

LOAD:	(destination)	[source 1]	Loads destination register from address in source register
WRITE:	[destination]	(source 1)	Writes to the value in source register to the address in destination register

Branch Operations:

JMP:	16 bit value			Jump to 16 bit address
CMP:	(source 1)	-	(source 2)	Subtract but do not store, setting flags.
CMPD	(source 1)	-	data	Subtract but do not store, setting flags
JE:	16 bit value			Jump to 16 bit address if Zero flag set
JA:	16 bit value			Jump to 16 bit address if Positive flag set
JSR:	16 bit value			Jump to subroutine at 16 bit address
RETURN:				Return from subroutine

Cell Instructions:

CSETDEL:	(cell) 3 bit value	Sets delay in selected cell to 3 bit value
CLRA:		Load RA into pipe, holding previous accumulates, and clearing RB's.
CLAA:	(cell)	Load RB and accumulate
CPASS:		Pass accumulates out / produce answers
CCLEAR:	(cell)	Clear all cell settings (RA/RB/ACC/Delay)
CLOAD:	(source 1)	Give cell data lines the data value from the address in source register
CLOADD:	16 bit data value	Give cell data lines the 16 bit data value

Cell Microcode Format

The microcode instruction that is sent to a cell consists of 16 bits. The description is as follows.

16 bit Cell Instruction

Cell ID	RA Attributes	RB Attributes	Accumulate Attributes	Delay
6 bits	1 bit	2 bits	3 bits	4 bits

Cell ID Field

Parallel Load	Cell Address
1 bit	5 bits

RA Attributes

Load External on Clock
1 bit

RB Attributes

Load External on Clock	Zero Value
1 bit	1 bit

Accumulate Attributes

Load External on Clock	Zero Value	Sum with Multiply Result
1 bit	1 bit	1 bit

Delay

Load from Microcode Word	Delay Value
1 bit	3 bits

Translations Using the TIM Assembler

The TIM assembler is used to translate elRoy assembly code into a machine language form that the system can run. While the TIM assembler provides an acceptable translation of instruction to hexadecimal values, it lacks some of the required addressing details needed by elRoy.

Since the assembly programs had to be run in VHDL, they needed to be translated from hexadecimal to a form that Synpopsys could understand. A C program (see appendix) was written to translate the assembly listing files into code that could be easily imported into Synopsys. One of the problems with the TIM assembler in its use with elRoy is that TIM assumes that the program counter fetches values in 32-bit chunks. Unfortunately, elRoy was designed to address memory in a logical 8-bit fashion due to the needs of data flow. Where TIM counts its memory locations by ones, elRoy needs values to be counted by fours.

The C program addresses this problem by adjusting every memory reference by multiplying it by four. While it doesn't make much logical sense to have to essentially compile an assembly program and then parse it again to fix errors, there was no adequate way to force TIM to address memory the way elRoy needed.

The assembly definition file is listed in the appendix.

Assembly Programs

For the three program goals of the project, specific assembly test programs were coded to prove the theory of design. All source code is listed in the appendix.

Convolution

For the convolution program, the system needed to load an H array into the cells and then present an X array through broadcasting. Additionally, the non-trivial task of writing assembly to pack the H array was also tackled. Cell utilization was maximized by having the assembly program examine the H array and determine if it could figure out a bubble system to manage zeros in the H array. In doing this, the system proved that the assembly language could set the cell delays without human interaction.

Sample data was applied in several versions of the assembly to prove that the algorithm and system did in fact do a proper convolution. On one run, a square wave was chosen as the X array, and a single square pulse was chosen for H. As expected, the square wave was convolved into a triangle wave (a pulse convolved with a pulse of equal width results in a triangular pulse with scaled height). The same square wave was then applied to a first difference filter ([1 -1]), resulting in spikes corresponding to the changes in the input. Other test runs were performed with arrays designed to take advantage of the packing algorithm, and the results were verified with MATLAB.

Matrix-Vector Multiplication

The matrix-vector program explored an alternate configuration for the cell array. The accumulates for each cell were set to load and accumulate internally, and inputs were slided in serially after every multiplication. The algorithm outlined in the theory section of this report was applied to the assembly language. After allowing the program to run, memory was examined and the proper values were discovered in the proper locations.

Matrix-Matrix Multiplication

The matrix-matrix assembly was created by adding particular code around the matrix-vector algorithm. Specifically, looping was implemented to achieve the desired result. The stack had to be used for temporary space during this operation, due to the lack of registers. The sample case chosen for this operation was a 4x9 matrix times a 9x4 matrix. The results were again found to be correct after verifying the results with MATLAB. For a time, larger matrices were considered as candidates for the multiplication, but the burden of time involved in verifying 81 or more answers allowed the 4x9 case to be adequate enough to prove the architecture.

V. Testing

Testing

Testing the elRoy system was taken very seriously due to the theoretical nature of the project. While testing the hardware implementation was not possible due to size constraints in the emulator boxes, the Synopsys environment was found to provide an accurate means of verifying the design's concepts. Testing was broken up into two stages within Synopsys: component testing and overall system testing.

Since a large portion of the design was written in low level logic, it was necessary to test all of the individual parts that would normally be represented by behavioral VHDL code. Tests were performed to cover as many parts as possible without compromising the accuracy of the tests. Several examples of the tests are included in the appendix. While the tests were tedious, they proved that elements worked as they were designed. The next stage of testing investigated whether what was designed was what really needed to be done.

The largest amount of debugging time was a result of the overall system testing. While elements of the design often worked as individual parts, it was discovered that the communication between each part was sometimes mismatched. Testing the overall system came late in the project due to the need for all parts to be defined and working. Additionally, the proper assembly code had to be written to test the operation of the machine. Inevitably, several unknowns had to be tested at the same time.

The first and most important test program to be run on the machine was a simple convolution operation (listed in the appendix). Since the algorithm is fairly complex and operation intensive, the majority of the errors in the state machine and datapath were found after several sessions of debugging. The convolution assembly was shown to produce the correct answers as well as pack arrays as designed. A great deal of time was then spent on verifying the Matrix-Vector and Matrix-Matrix multiplications. After modifications were made within the TIM assembler and the overall machine, all three algorithms were determined to produce correct results. All answers were verified in MATLAB.

Since the three program goals were met with great success, it was determined that our theories of machine design as well as machine implementation were correct.

Theoretical Speed Comparisons

A few 'C' programs were written to benchmark the multipliers on the Pentium. These were tested on the machines in the VLSI lab (60 MHz). The program took 38.2 seconds to calculate 100 million multiplies of two variables of type int. This comes out to 22 clock cycles per multiply. elRoy currently takes 18 clock cycles for a multiply-accumulate chunk. Given two functions with lengths M and N, the Pentium would take (N)(M)(22)(clock period). elRoy, with an array of size X would take (N)(M)(18)(clock period) / X. The table below charts out the ratio of calculation times required for various values of X and the two clock rates (in MHz):

Ratio (Pentium/elRoy)	Number of Cells	elRoy Clock Rate	Pentium Clock Rate
0.15	4	2	66
0.3	4	2	33
0.74	4	10	66
2.37	64	2	66
4.74	64	2	33
11.85	64	10	66
47.41	256	10	66
94.81	512	10	66
625.76	1,024	33	66

The gain is remarkable. Also note that elRoy currently uses an unoptimized algorithm for the multiplies. A faster algorithm would greatly reduce the clock cycles required per multiply and enhance the speedup factor. Also, these factors due not include the pipelining introduced by the FIFO queue in each cell. For very specialized input vectors, the time savings approaches another factor of seven per cell. The 652.67 in the last example becomes a staggering 4,485,447.68. Note also that these numbers do not reflect memory reads and writes.

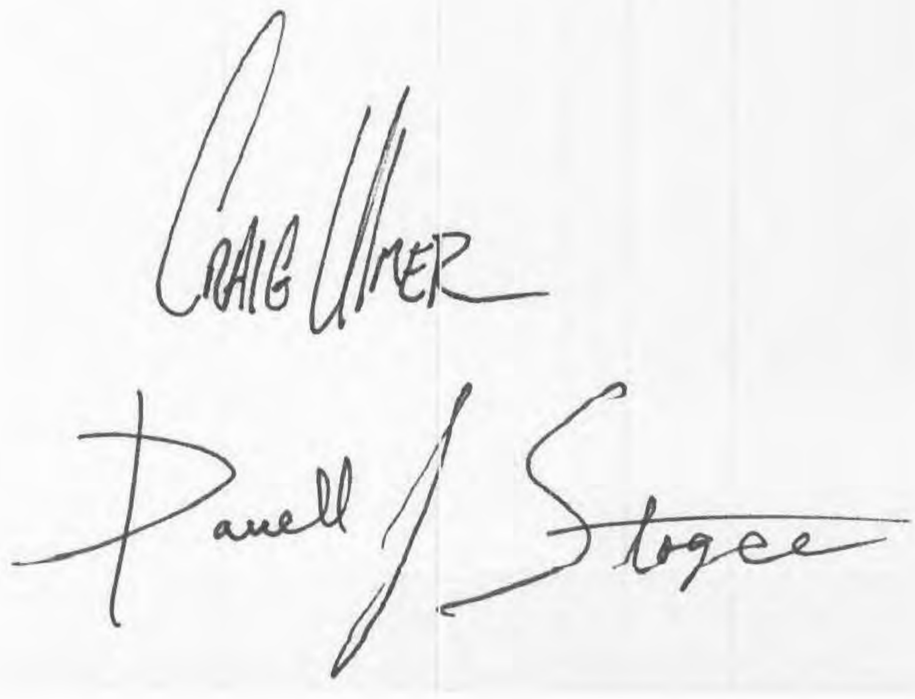
Conclusions

The elRoy system provides a great deal of insight into the amount of thought required in building parallel systems. While the benefits for building such systems are obvious in special high calculation jobs, the level of complexity jumps in every level of computer usage, from the design to the programming language. However, once these issues are dealt with, their lessons can be applied to several similar applications.

Although the project was too large to fit into the hardware emulators, we were still able to verify the theory of design through software emulation. Since the implementation was elaborated down to the gate level, the simulations generated results that correspond to those expected with the actual hardware implementation.

Overall, elRoy was a large challenge for the design team because it journeyed into an area with no obvious guidelines or models. The model for the system was constructed bottom up as we blindly felt our way through often unstable requirements. As a result, the design underwent several revisions, each adding new characteristics to the overall system. We feel that it is nearly impossible to design an experimental system such as elRoy without redesigning the machine as you work along.

The end product satisfies our original goals. The system is easily expandable and allows a flexible architecture to perform several high computation algorithms. All three specified mathematical programs were assembled, tested, and verified for our architecture. While elRoy's use in the real world is limited to education, it serves as a convenient model for the advantages of a mixture of various architectures. In the least, it has served as a back bone for our design team, and its design can never be fully documented.



Appendix A: TIM Assembly Definitions

```
HALE Listing: asm.src                    Mar 12 11:52:13 1995        Page   1

 Line  ASSEMBLY LANGUAGE DEFINITION FILE FOR ==>elRoy<==

    1  TITLE    ASSEMBLY LANGUAGE DEFINITION FILE FOR ==>elRoy<==
    2  WORD     32
    3  WIDTH    72
    4  LINES    50
    5
    6  ;////////////////////////////////////////////////////////////////
    7  ;  File: ASM.SRC  Purpose: Assembly definitions for the elRoy pr
    8  ;  Craig Ulmer / Darrell Stogner                          CompE 45
    9  ;  No Modifications without Authors' consent                   Mar
   10  ;////////////////////////////////////////////////////////////////
   11
   12  NUMCELLS:   EQU     H#0004   ; Number of cells in the system
   13
   14  ;*****************************************************************
   15  ;Standard REGISTER ASSIGNMENTS
   16  ;*****************************************************************
   17  R0:         EQU     B#0000
   18  R1:         EQU     B#0001
   19  R2:         EQU     B#0010
   20  R3:         EQU     B#0011
   21  R4:         EQU     B#0100
   22  R5:         EQU     B#0101
   23  R6:         EQU     B#0110
   24  R7:         EQU     B#0111
   25  ;*****************************************************************
   26  ; SPECIAL FUNCTION REGISTERS
   27  ;*****************************************************************
   28  ACCIN:      EQU     B#1000        ; ACCUMULATE IN  - WRITE ONLY
   29  ACCOUT:     EQU     B#1001        ; ACCUMULATE OUT - READ ONLY
   30  CINST:      EQU     B#1010        ; CELL INSTRUCTION
   31  RA:         EQU     B#1011        ; DATA FOR RA
   32  RB:         EQU     B#1011        ; DATA FOR RB
   33  CDELINT:    EQU     B#1100        ; DELAY INTERNAL
   34  EXTDATA:    EQU     B#1101        ; USE A 16BIT DATA VALUE
   35
   36  ;*****************************************************************
   37  ; CELL FUNCTION EQUATES
   38  ;*****************************************************************
   39  CELL0:      EQU     B#000000
   40  CELL1:      EQU     B#000001
   41  CELL2:      EQU     B#000010
   42  CELL3:      EQU     B#000011
   43  CELLP:      EQU     B#100000       ; PARALLEL LOAD ALL CELLS
   44  ;*****************************************************************
```

```
HALE Listing: asm.src                    Mar 12 11:52:13 1995        Page   2

 Line  ASSEMBLY LANGUAGE DEFINITION FILE FOR ==>elRoy<==

   45  ; INSTRUCTION OPCODE LABELS - MUST BE 4-BITS
   46  ;*****************************************************************
   47  LOR:        EQU     B#0000
   48  LXOR:       EQU     B#0001
   49  LADD:       EQU     B#0010
   50  LSUB:       EQU     B#0011
   51  LAND:       EQU     B#0100
   52  LCOP:       EQU     B#0101
   53  LPUSH:      EQU     B#0110
   54  LPOP:       EQU     B#0111
   55  ; JUMP INSTRUCTIONS
   56  LJMP:       EQU     B#1000
   57  LJA:        EQU     B#1001
   58  LJE:        EQU     B#1010
   59  LCMP:       EQU     B#1011
   60  LRTS:       EQU     B#1100
   61  LJSR:       EQU     B#1110
   62
   63  ; LOAD INSTRUCTIONS
   64  LLOAD:      EQU     B#1101
   65  LWRITE:     EQU     B#1111
   66
   67  ; SETUP
   68  NULL:       EQU     B#0000                ;4-BIT ZERO VALUE
   69  OPCODE:     SUB     4VLCOP                ;4-BIT OPCODE FIELD
   70  BLANK16:    EQU     16H#0000              ;16-BIT FIELD
   71  DW:         DEF     16VH#0000,16VH#0000   ;32-BIT DATA DIRECTIVE
   72  ;*****************************************************************
   73  ;ASSEMBLY LANGUAGE INSTRUCTIONS
   74  ;*****************************************************************
   75  ; Register to Register ALU ops
   76  XORR:       DEF     LXOR,4VH#0,4VH#0,4VH#0,BLANK16
   77  ADDR:       DEF     LADD,4VH#0,4VH#0,4VH#0,BLANK16
   78  SUBR:       DEF     LSUB,4VH#0,4VH#0,4VH#0,BLANK16
   79  COPYR:      DEF     LCOP,4VH#0,4VH#0,NULL,BLANK16
   80  ANDR:       DEF     LAND,4VH#0,4VH#0,4VH#0,BLANK16
   81  ORR:        DEF     LOR,4VH#0,4VH#0,4VH#0,BLANK16
   82  ; Register Ops with 16 bit data
   83  XORD:       DEF     LXOR,4VH#0,EXTDATA,4VH#0,16VH#0000
   84  ADDD:       DEF     LADD,4VH#0,EXTDATA,4VH#0,16VH#0000
   85  SUBD:       DEF     LSUB,4VH#0,EXTDATA,4VH#0,16VH#0000
   86  COPYD:      DEF     LCOP,4VH#0,EXTDATA,NULL,16VH#0000
   87  ANDD:       DEF     LAND,4VH#0,EXTDATA,4VH#0,16VH#0000
   88  ORD:        DEF     LOR,4VH#0,EXTDATA,4VH#0,16VH#0000
```

Assembly Definition File

```
 Line  ASSEMBLY LANGUAGE DEFINITION FILE FOR ==>elRoy<==

   89  CMPD:       DEF     LCMP,NULL,EXTDATA,4VH#0,16VH#0000
   90
   91  ; MEMOPS
   92  LOAD:       DEF     LLOAD,4VH#0,4VH#0,NULL,BLANK16      ; LOAD T
   93  WRITE:      DEF     LWRITE,NULL,4VH#0,4VH#0,BLANK16     ; WRITE
   94  LOADD:      DEF     LLOAD,4VH#0,EXTDATA,NULL,16VH#0000  ; Load D
   95
   96  ; BRANCHING
   97  JMP:        DEF     LJMP,NULL,NULL,NULL,16VH#0000
   98  CMP:        DEF     LCMP,NULL,4VH#0,4VH#0,BLANK16
   99  JE:         DEF     LJE,NULL,NULL,NULL,16VH#0000
  100  JA:         DEF     LJA,NULL,NULL,NULL,16VH#0000
  101  JSR:        DEF     LJSR,NULL,NULL,NULL,16VH#0000
  102  RETURN:     DEF     LRTS,NULL,NULL,NULL,BLANK16
  103
  104  ; Other stuff
  105  NOP:        DEF     LOR,NULL,NULL,NULL,BLANK16       ; OR R0 WIT
  106  TIMSUCKS:   DEF     16VH#0000,16VH#0000              ; For data
  107
  108  ;*****************************************************************
  109  ; CELL ASSEMBLY LANGUAGE INSTRUCTIONS
  110  ;*****************************************************************
  111  ; CELL INSTRUCTION FORMAT (16 BITS):
  112  ;       HIGH
  113  ;          6 : CELL#       : 1 -- CELL PARALLEL LOAD
  114  ;                          : 5 -- CELL ADDRESS
  115  ;          1 : RA          : 1 -- LOAD EXTERNAL
  116  ;          2 : RB          : 1 -- LOAD EXTERNAL
  117  ;                          : 1 -- SET TO ZERO
  118  ;          2 : ACCUMULATE  : 1 -- LOAD EXTERNAL
  119  ;                          : 1 -- SET TO ZERO
  120  ;          1 : ACC GATE    : 1 -- ADD MULTIPLY RESULT TO ACCUMULA
  121  ;          4 : DELAY       : 1 -- LOAD NEW DELAY VALUE
  122  ;                          : 3 -- DELAY VALUE
  123  ;        LOW
  124
  125  ;.................................................................
  126  ; SET DELAY IN CELL
  127  ;    User specifies cell and delay value
  128  ;                                              CELL#      LOAD
  129  CSETDEL:     DEF     LLOAD,CINST,EXTDATA,NULL,6VB#000000,B#00000
  130
  131  ;.................................................................
  132  ; SET DELAY IN CELL BASED ON INTERNAL VALUES
```



```
 Line  ASSEMBLY LANGUAGE DEFINITION FILE FOR ==>elRoy<==

  133  ;           RA holds the cell ID
  134  ;           RB holds the cell's delay
  135  CSETDELI:    DEF     LLOAD,CDELINT,4VH#0,4VH#0,BLANK16
  136
  137  ;.................................................................
  138  ; LOAD RA PIPE -- HOLDS ON TO PREVIOUS ACCUMULATE -- RB is Zero
  139  ;                       LOAD  DEST              CELL#
  140  CLRA:        DEF     LLOAD,CINST,EXTDATA,NULL,CELLP,B#1000100000
  141
  142  ;.................................................................
  143  ; LOAD RB AND ACCUMULATE EXTERNALLY - FOR CONVOLUTION, Acc from
  144  ;                       LOAD  DEST              CELL#
  145  CLAAE:       DEF     LLOAD,CINST,EXTDATA,NULL,CELLP,B#0101010000
  146
  147  ;.................................................................
  148  ; LOAD RB AND ACCUMULATE INTERNALLY - FOR MULTIPLICATION Acc fro
  149  ;                       LOAD  DEST              CELL#
  150  CLAAI:       DEF     LLOAD,CINST,EXTDATA,NULL,CELLP,B#0100010000
  151
  152  ;.................................................................
  153  ; LOAD ACCUMULATOR - RA Constant,RB Zero,Load external Acc, Pass
  154  ;                       LOAD  DEST              CELL#
  155  CLACC:       DEF     LLOAD,CINST,EXTDATA,NULL,CELLP,B#0011010000
  156
  157  ;.................................................................
  158  ; PASS OUT ACCUMULATES -- GIVES ANSWERS
  159  ;                       LOAD  DEST              CELL#
  160  CPASS:       DEF     LLOAD,CINST,EXTDATA,NULL,CELLP,B#0011010000
  161
  162  ;.................................................................
  163  ; CELL CLEAR -- WIPES OUT DELAY AND ACCUMULATE, SETS RB TO ZERO
  164  ;                       LOAD  DEST              CELL#
  165  CCLEAR:      DEF     LLOAD,CINST,EXTDATA,NULL,6VB#000000,B#00101
  166  END
```

Symbol Table: asm.src Page 5

ACCIN	A	00000008	ACCOUT	A	00000009	ADDD	D		ADDR	D	
ANDD	D		ANDR	D		BLANK16	A	00000000	CCLEAR	D	
CDELINT	A	0000000C	CELL0	A	00000000	CELL1	A	00000001	CELL2	A	00000002
CELL3	A	00000003	CELLP	A	00000020	CINST	A	0000000A	CLAAE	D	
CLAAI	D		CLACC	D		CLRA	D		CMP	D	
CMPD	D		COPYD	D		COPYR	D		CPASS	D	
CSETDEL	D		CSETDELI	D		DW	D		EXTDATA	A	0000000D
JA	D		JE	D		JMP	D		JSR	D	
LADD	A	00000002	LAND	A	00000004	LCMP	A	0000000B	LCOP	A	00000005
LJA	A	00000009	LJE	A	0000000A	LJMP	A	00000008	LJSR	A	0000000E
LLOAD	A	0000000D	LOAD	D		LOADD	D		LOR	A	00000000
LPOP	A	00000007	LPUSH	A	00000006	LRTS	A	0000000C	LSUB	A	00000003
LWRITE	A	0000000F	LXOR	A	00000001	NOP	D		NULL	A	00000000
NUMCELLS	A	00000004	OPCODE	D		ORD	D		ORR	D	
R0	A	00000000	R1	A	00000001	R2	A	00000002	R3	A	00000003
R4	A	00000004	R5	A	00000005	R6	A	00000006	R7	A	00000007
RA	A	0000000B	RB	A	0000000B	RETURN	D		SUBD	D	
SUBR	D		TIMSUCKS	D		WRITE	D		XORD	D	
XORR	D										

Definition Phase complete.
0 error(s) detected.

Appendix B: Test Programs

Convolution Program

This Program Analyzes AN H Array, figures out the delays AND Cell weights, AND Convolve the H AND X Arrays. The Complexity of this program is due to the Array packing Algorithm. The Packing Algorithm maximizes cell usage by determining bubbles in the Convolution Pipeline.

```
HALE Listing: convolve.asm              Mar 12 14:56:45 1995          Page   1

 Addr              Line CONVOLUTION Program

                     TITLE  CONVOLUTION Program
                     LIST F,W
                     LINES 50
                     ;/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/
-/-/-/-/-
                     ; Convolution Assembly Program for use with elRoy Systolic Array Arc
hitecture
                     ; Craig Ulmer / Darrell Stogner                                  COMPE
4500/4510
                     ;
                     ; No modifications or duplications without the authors' consent
                     ;/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/
-/-/-/-/-

00000 DAD000A8       BEGINCODE: CCLEAR                       ; clear out the cells of all v
alues

                     ;"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""
                     ; Find Delays
                     ;   This algorithm finds out what the delays should be in each cell
by
                     ;   scanning through H and looking for zeros.
                     ;     Registers:  R6: # elements we've counted
                     ;                 R5: Address of new RA values
                     ;                 R4: Temp load value of H
                     ;                 R3: # delay for current
                     ;                 R2: Current Cell number
                     ;                 R1: # elements in H array
                     ;                 R0: H array address
                     ;"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""

00001 50D0004C         COPYD    R0,CONSTHPOS%:      ; Loads the location of H
00002 D1000000         LOAD     R1,R0               ; Loads H Length into R1
00003 21D10001         ADDD     R1,R1,H#0001        ; Offset the H length by 1, for loopin
g
00004 20D00002         ADDD     R0,R0,H#0002        ; Set H pointer to first element
00005 52D00000         COPYD    R2,H#0000           ; Current Cell is #0
00006 55D0004E         COPYD    R5,RAVALS%:         ; Set R5 to next RA write address
00007 56D00000         COPYD    R6,H#0000           ; Set elements counted to 0

00008 53D00000       ACTIVELOOP:COPYD    R3,H#0000  ; Set the number of zeros counter to z
ero

00009 31D10001       NEXTEL: SUBD     R1,R1,H#0001  ; Decrease the number of elements by o
ne
0000A A000001B         JE       DONEELE%:           ; If no more elements, go on to RA ins
ertion
0000B 26D60001         ADDD     R6,R6,H#0001        ; Increase the number of elements coun
ted
0000C D4000000         LOAD     R4,R0               ; Load the next value of H into R4
0000D 20D00002         ADDD     R0,R0,H#0002        ; Increase the pointer of H
0000E B0D40000         CMPD     R4,H#0000           ; See if element of H is zero
0000F A0000017         JE       FOUNDZERO%:         ; If zero, deal with it
```

```
HALE Listing: convolve.asm              Mar 12 14:56:45 1995        Page   2

 Addr           Line CONVOLUTION Program


                  43
                  44                                ; Current element was non-zero
00010 F0540000    45 SETDEL:  WRITE    R5,R4        ; Remember this RA value, place on sc
ratch pad
00011 25D50002    46   ADDD     R5,R5,H#0002        ; Point to the next RA scrap pad
00012 DC230000    47   CSETDELI R2,R3               ; Set the current cell delay
00013 22D20001    48   ADDD     R2,R2,H#0001        ; Set to the next cell
00014 B0D20004    49   CMPD     R2,NUMCELLS         ; See if we've hit the last possible
cell
00015 A0000026    50   JE       DORAS%:             ; If yes, do a truncated convolution
00016 80000008    51   JMP      ACTIVELOOP%:        ; If not, start the loop again
                  52
                  53                                ; Found a zero, add it to the list
00017 B0D30007    54 FOUNDZERO:  CMPD  R3,H#0007    ; Check to see if we've gotten all of
our delay
00018 A0000010    55   JE       SETDEL%:            ; Too many Zeros, we must use an RA as
 0
00019 23D30001    56   ADDD     R3,R3,H#0001        ; Otherwise, increase the delay by one
0001A 80000009    57   JMP      NEXTEL%:
                  58
                  59                                ; We've finished with the h elements,
check
0001B 50D0004C    60 DONEELE: COPYD R0,CONSTHPOS%: ; Reload H position to get length
0001C D1000000    61   LOAD     R1,R0               ; Get length of H again
0001D B0160000    62   CMP      R1,R5               ; Compare with elements we counted
0001E A0000026    63   JE       DORAS%:             ; If equal, then we filled array perfe
ctly
                  64
0001F 54D00000    65   COPYD    R4,H#0000           ; Set the RA write value to zero
                  66
                  67                                ; We did not completely fill the array
,pad out
00020 F0540000    68 PADZEROS:  WRITE    R5,R4      ; Write to the next RA scrap place

00021 25D50002    69   ADDD     R5,R5,H#0002        ; Point to the next RA scrap place
00022 22D20001    70   ADDD     R2,R2,H#0001        ; Point to the next cell
00023 B0D20004    71   CMPD     R2,NUMCELLS         ; See if we've hit the last cell yet
00024 A0000026    72   JE       DORAS%:             ; Yes, load the RA pipe
00025 80000020    73   JMP      PADZEROS%:          ; No, continue padding
                  74
                  75
00026 DAD08220    76 DORAS:  CLRA                  ; Begin loading the RA pipe
00027 52D00000    77   COPYD    R2,H#0000           ; Begin at first cell
00028 55D0004E    78   COPYD    R5,RAVALS%:         ; Set first address or RA scratch pad
00029 DB500000    79 LOADRASLOOP: LOAD     RA,R5    ; Load in the next RA to a cell
0002A 25D50002    80   ADDD     R5,R5,H#0002        ; Increment pointer to next RA scratch
0002B 22D20001    81   ADDD     R2,R2,H#0001        ; Look at next cell
0002C B0D20004    82   CMPD     R2.NUMCELLS         ; See if we've loaded all
0002D A000002F    83   JE       DOCONV%:            ; If so, Begin the convolution
0002E 80000029    84   JMP      LOADRASLOOP%:       ; If not, keep looping
```

```
HALE Listing: convolve.asm              Mar 12 14:56:45 1995        Page   3

 Addr           Line CONVOLUTION Program


                  85
                  86
                  87 ;*************************************************************
**********
                  88 ; Do Convolution
                  89 ;     This section of the code actually performs the convolution ope
ration
                  90 ;     Registers:  R7: Length of Y
                  91 ;                 R6:--- # elements we've counted
                  92 ;                 R5:--- Address of new RA values
                  93 ;                 R4:--- Temp load value of H
                  94 ;                 R3: Y Position
                  95 ;                 R2: X Position
                  96 ;                 R1: Length of H-1
                  97 ;                 R0: Length of X
                  98 ;*************************************************************
**********
                  99
                 100                          ; Convolution Assembly
0002F 53D00057   101 DOCONV:  COPYD R3,CONSTYPOS%: ; Load Loaction of Y Array (Result)
00030 52D00047   102   COPYD    R2,CONSTXPOS%:     ; Load location of X array
00031 51D0004C   103   COPYD    R1,CONSTHPOS%:     ; Load Location of H array
00032 D0200000   104   LOAD     R0,R2              ; Get the length of X
00033 22D20002   105   ADDD     R2,R2,H#0002       ; Point to first element of X
00034 D1100000   106   LOAD     R1,R1              ; Get the length of H
00035 31D10001   107   SUBD     R1,R1,H#0001       ; Set to H-1
00036 59D00000   108   COPYD    ACCOUT,H#0000      ; Set the Acc to always be zero
00037 27100000   109   ADDR     R7,R1,R0           ; Add length X + length H - 1
00038 F0370000   110   WRITE    R3,R7              ; Write length Y to first Y address
00039 23D30002   111   ADDD     R3,R3,H#0002       ; Increase Y pointer
                 112
0003A DAD08150   113   CLAAE                       ; Set up for the convolution operation
                 114
                 115                               ; There are still X values to send thr
ough
0003B DB200000   116 STILLX:  LOAD     RB, R2      ; Load the next x val into RB
0003C F0380000   117   WRITE    R3,ACCIN           ; Write the result to next y
0003D 23D30002   118   ADDD     R3,R3,H#0002       ; Increase y pointer
0003E 22D20002   119   ADDD     R2,R2,H#0002       ; Increase x pointer
0003F 30D00001   120   SUBD     R0,R0,H#0001       ; Decrease x counter
00040 9000003B   121   JA       STILLX%:           ; If not zero, keep churning
                 122
00041 DBD00000   123 STILLH:  LOADD    RB,H#0000   ; Load zero into the RB pipe
00042 F0380000   124   WRITE    R3,ACCIN           ; Write result out to next y
00043 23D30002   125   ADDD     R3,R3,H#0002       ; Increase the y pointer
00044 31D10001   126   SUBD     R1,R1,H#0001       ; Decrease the h counter
```

```
HALE Listing: convolve.asm              Mar 12 14:56:45 1995        Page   4

 Addr            Line CONVOLUTION Program

00045 90000041   127   JA       STILLH%:          ; If not zero, keep churning
                 128
00046 80000046   129 DONE:  JMP      DONE%:
                 130
00047 00090001   131 CONSTXPOS: TIMSUCKS  H#0009,H#0001
00048 00020003   132            TIMSUCKS  H#0002,H#0003
00049 00040005   133            TIMSUCKS  H#0004,H#0005
0004A 00060007   134            TIMSUCKS  H#0006,H#0007
0004B 00080009   135            TIMSUCKS  H#0008,H#0009
0004C 00030001   136 CONSTHPOS: TIMSUCKS  H#0003,H#0001
0004D 00020003   137            TIMSUCKS  H#0002,H#0003
0004E 00000000   138 RAVALS:    TIMSUCKS
0004F 00000000   139            TIMSUCKS
00050 00000000   140            TIMSUCKS
00051 00000000   141            TIMSUCKS
00052 00000000   142            TIMSUCKS
00053 00000000   143            TIMSUCKS
00054 00000000   144            TIMSUCKS
00055 00000000   145            TIMSUCKS
00056 00000000   146            TIMSUCKS
00057 00000000   147 CONSTYPOS: TIMSUCKS
00058 00000000   148            TIMSUCKS
00059 00000000   149            TIMSUCKS
0005A 00000000   150            TIMSUCKS
0005B 00000000   151            TIMSUCKS
0005C 00000000   152            TIMSUCKS
0005D 00000000   153            TIMSUCKS
0005E 00000000   154            TIMSUCKS
0005F 00000000   155            TIMSUCKS
00060 00000000   156            TIMSUCKS
00061 00000000   157            TIMSUCKS
                 158
```

```
Symbol Table: convolve.asm                                        Page   5

 ACCIN     A 00000008 ACCOUT    A 00000009 ACTIVELO A 00000008 ADDD      D
 D           ANDD     D          ANDR     D
 BEGINCOD A 00000000 BLANK16  A 00000000 CCLEAR   D           CDELINT   A 0000000C CELL0
 A 00000000 CELL1    A 00000001 CELL2    A 00000002
 CELL3     A 00000003 CELLP    A 00000020 CINST    A 0000000A CLAAE     D
 D           CLACC    D          CLRA     D
 CMP       D          CMPD     D          CONSTHPO A 0000004C CONSTXPO A 00000047 CONSTYPO
 A 00000057 COPYD    D          COPYR    D
 CPASS     D          CSETDEL  D          CSETDELI D          DOCONV    A 0000002F DONE
 A 00000046 DONEELE  A 0000001B DORAS    A 00000026
 DW        D          EXTDATA  A 0000000D FOUNDZER A 00000017 JA        D          JE
 D           JMP      D          JSR      D
 LADD      A 00000002 LAND     A 00000004 LCMP     A 0000000B LCOP      A 00000005 LJA
 A 00000009 LJE      A 0000000A LJMP     A 00000008
 LJSR      A 0000000E LLOAD    A 0000000D LOAD     D          LOADD     D          LOADRASL
 A 00000029 LOR      A 00000000 LPOP     A 00000007
 LPUSH     A 00000006 LRTS     A 0000000C LSUB     A 00000003 LWRITE    A 0000000F LXOR
 A 00000001 NARG     A 00000000 NEXTEL   A 00000009
 NOP       D          NULL     A 00000000 NUMCELLS A 00000004 OPCODE    D          ORD
 D           ORR      D          PADZEROS A 00000020
 R0        A 00000000 R1       A 00000001 R2       A 00000002 R3        A 00000003 R4
 A 00000004 R5       A 00000005 R6       A 00000006
 R7        A 00000007 RA       A 0000000B RAVALS   A 0000004E RB        A 0000000B RETURN
 D           SETDEL   A 00000010 STILLH   A 00000041
 STILLX    A 0000003B SUBD     D          SUBR     D          TIMSUCKS  D          WRITE
 D           XORD     D          XORR     D

Assembly Phase complete.
0 error(s) detected.
```

MATRIX-Vector Multiplication Program

Sets up the Array to work as a Vector multiplication program. The Matrix Components are loaded in serially, while the vector is broadcast to the cells. Results in only one multiply operation per matrix Column.

```
HALE Listing: vector.asm                    Mar 13 14:15:28 1995         Page   1

                        Vector
 Addr            Line  MATRIX Program
                            Vector
                  TITLE  MATRIX Program
                  LIST F,W
                  LINES 50
                  ;/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/
-/-/-/-/-
                  ; Matrix Mult Assembly Program for use with elRoy Systolic Array Arc
hitecture
                  ; Craig Ulmer / Darrell Stogner                              COMPE
4500/4510
                  ;
                  ; No modifications or duplications without the authors' consent
                  ;/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/
-/-/-/-/-


                  ;""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""
                  ; Multiply Matrx X by Vector V
                  ;
                  ;     Registers:  R7: Constant Rows of X
                  ;                 R6: X Last Address of a loop = X Starting + #Cols*
2
                  ;                 R5: X starting address
                  ;                 R4: # Columns of X * 2 = Rows of V * 2

                  ;                 R3: RA Loop        Rows = Rows of V
                  ;                 R2: Multiply Loop Rows = Rows of X
                  ;                 R1: Current V Address
                  ;                 R0: Current X Address
                  ;""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""

00000 DAD000A8    BEGINCOD: CCLEAR                    ; clear out the cells of all values

00001 50D0002D    INITCODE: COPYD   R0,CONXPOS%:      ; R0=X starting Address
00002 51D00040              COPYD   R1,CONVPOS%:      ; R1=V starting Address
00003 D2000000              LOAD    R2,R0             ; R2=Rows of X
00004 D3100000              LOAD    R3,R1             ; R3=Rows of V / Cols of X
00005 20D00002              ADDD    R0,R0,H#0002      ; R0=First X data
00006 21D10002              ADDD    R1,R1,H#0002      ; R1=First V data
00007 57300000              COPYR   R7,R3             ; R7=Temp counter of X columns
00008 54D00000              COPYD   R4,H#0000         ; R4=Col*2 offset begins at zero
00009 24D40002    INITLOOP: ADDD    R4,R4,H#0002      ; One more element-> offset by +two

0000A 37D70001              SUBD    R7,R7,H#0001      ; Decrease columns counted
0000B 90000009              JA      INITLOOP%:        ; If still a column, loop
0000C 55000000              COPYR   R5,R0             ; R5=First X data address
0000D 26540000              ADDR    R6,R5,R4          ; R6=Address of 2nd row,1st column
0000E 35D50002              SUBD    R5,R5,H#0002      ; Kludge R5 for looping
0000F 57200000              COPYR   R7,R2             ; R7=Number of Rows in X
```

```
HALE Listing: vector.asm                Mar 13 14:15:28 1995        Page   2

 Addr           Line MATRIX Program

00010 DAD08290    43 ADDRA:     CLRA                    ; Set to load RA pipe, holding ACC
S
00011 25D50002    44            ADDD    R5,R5,H#0002    ; Increase X origin by one spot
00012 B0560000    45            CMP     R5,R6           ; See if we've hit the last place
00013 A0000024    46            JE      PASSOUTS%:      ; If so, go to the results pass
00014 50500000    47            COPYR   R0,R5           ; if not, load the next stating X
pos
00015 53700000    48            COPYR   R3,R7           ; Reset the row counter
00016 DB000000    49 ADDLOOP:   LOAD    RA,R0           ; load next X into the RA pipe
00017 20040000    50            ADDR    R0,R0,R4        ; Move one row down
00018 33D30001    51            SUBD    R3,R3,H#0001    ; Decrease the row counter
00019 90000016    52            JA      ADDLOOP%:       ; If not last one, keep looping
                  53
0001A 50D00004    54 PADCELLS:  COPYD   R0,NUMCELLS     ; Get the number of cells
0001B 30700000    55            SUBR    R0,R7,R0        ; R0=Num Cells - Rows
0001C A0000020    56 PADLOOP:   JE      DOMULT%:        ; Perfect fit, do the mult
0001D DBD00000    57            LOADD   RA,H#0000       ; Load a dummy into the RA pipe
0001E 30D00001    58            SUBD    R0,R0,H#0001    ; Decrease counter
0001F 8000001C    59            JMP     PADLOOP%:       ; Keep looping
                  60
00020 DAD08110    61 DOMULT:    CLAAI                   ; Set for Multiply
00021 DB100000    62            LOAD    RB,R1           ; RA loaded, Load the next part of
 V
00022 21D10002    63            ADDD    R1,R1,H#0002    ; Point to the next value of V
00023 80000010    64            JMP     ADDRA%:         ; Do the next column
                  65
00024 50D00045    66 PASSOUTS:  COPYD   R0,CONYPOS%:    ; Start at beginning Y
00025 F0070000    67            WRITE   R0,R7           ; Write # rows to first position
00026 DAD080D0    68            CPASS                   ; Set to pass out answers
00027 20D00002    69 PASSLOOP:  ADDD    R0,R0,H#0002    ; Point to first Y data value
00028 F0080000    70            WRITE   R0,ACCIN        ; Write current result out
00029 DBD00000    71            LOADD   RB,H#0000       ; Force a value to pop out
0002A 37D70001    72            SUBD    R7,R7,H#0001    ; Decrease the counter
0002B 90000027    73            JA      PASSLOOP%:
0002C 8000002C    74 DONE:      JMP     DONE%:
                  75
0002D 00040001    76 CONXPOS:   TIMSUCKS  H#0004,H#0001
0002E 00020003    77            TIMSUCKS  H#0002,H#0003
0002F 00040005    78            TIMSUCKS  H#0004,H#0005
00030 00060007    79            TIMSUCKS  H#0006,H#0007
00031 00080009    80            TIMSUCKS  H#0008,H#0009
00032 00110012    81            TIMSUCKS  H#0011,H#0012
00033 00130014    82            TIMSUCKS  H#0013,H#0014
00034 00150016    83            TIMSUCKS  H#0015,H#0016
00035 00170018    84            TIMSUCKS  H#0017,H#0018
```

```
HALE Listing: vector.asm                Mar 13 14:15:28 1995        Page   3

 Addr           Line MATRIX Program

00036 00190021    85            TIMSUCKS  H#0019,H#0021
00037 00220023    86            TIMSUCKS  H#0022,H#0023
00038 00240025    87            TIMSUCKS  H#0024,H#0025
00039 00260027    88            TIMSUCKS  H#0026,H#0027
0003A 00280029    89            TIMSUCKS  H#0028,H#0029
0003B 00310032    90            TIMSUCKS  H#0031,H#0032
0003C 00330034    91            TIMSUCKS  H#0033,H#0034
0003D 00350036    92            TIMSUCKS  H#0035,H#0036
0003E 00370038    93            TIMSUCKS  H#0037,H#0038
0003F 00390000    94            TIMSUCKS  H#0039,H#0000
00040 00090001    95 CONVPOS:   TIMSUCKS  H#0009,H#0001
00041 00020003    96            TIMSUCKS  H#0002,H#0003
00042 00040005    97            TIMSUCKS  H#0004,H#0005
00043 00060007    98            TIMSUCKS  H#0006,H#0007
00044 00080009    99            TIMSUCKS  H#0008,H#0009
                 100
                 101
00045 00000000   102 CONYPOS:   TIMSUCKS
00046 00000000   103            TIMSUCKS
00047 00000000   104            TIMSUCKS
00048 00000000   105            TIMSUCKS
00049 00000000   106            TIMSUCKS
0004A 00000000   107            TIMSUCKS
0004B 00000000   108            TIMSUCKS
0004C 00000000   109            TIMSUCKS
0004D 00000000   110            TIMSUCKS
0004E 00000000   111            TIMSUCKS
0004F 00000000   112            TIMSUCKS
00050 00000000   113            TIMSUCKS
                 114
```

```
Symbol Table: vector.asm                                        Page   4

 ACCIN    A 00000008 ACCOUT   A 00000009 ADDD     D          ADDLOOP  A 00000016 ADDR
  D          ADDRA    A 00000010 ANDD     D
 ANDR     D          BEGINCOD A 00000000 BLANK16  A 00000000 CCLEAR   D          CDELINT
  A 0000000C CELL0    A 00000000 CELL1    A 00000001
 CELL2    A 00000002 CELL3    A 00000003 CELLP    A 00000020 CINST    A 0000000A CLAAE
  D          CLAAI    D          CLACC    D
 CLRA     D          CMP      D          CMPD     D          CONVPOS  A 00000040 CONXPOS
  A 0000002D CONYPOS  A 00000045 COPYD    D
 COPYR    D          CPASS    D          CSETDEL  D          CSETDELI D          DOMULT
  A 00000020 DONE     A 0000002C DW       D
 EXTDATA  A 0000000D INITCODE A 00000001 INITLOOP A 00000009 JA       D          JE
  D          JMP      D          JSR      D
 LADD     A 00000002 LAND     A 00000004 LCMP     A 0000000B LCOP     A 00000005 LJA
  A 00000009 LJE      A 0000000A LJMP     A 00000008
 LJSR     A 0000000E LLOAD    A 0000000D LOAD     D          LOADD    D          LOR
  A 00000000 LPOP     A 00000007 LPUSH    A 00000006
 LRTS     A 0000000C LSUB     A 00000003 LWRITE   A 0000000F LXOR     A 00000001 NARG
  A 00000000 NOP      D          NULL     A 00000000
 NUMCELLS A 00000004 OPCODE   D          ORD      D          ORR      D          PADCELL
S A 0000001A PADLOOP  A 0000001C PASSLOOP A 00000027
 PASSOUTS A 00000024 POP      D          PUSH     D          R0       A 00000000 R1
  A 00000001 R2       A 00000002 R3       A 00000003
 R4       A 00000004 R5       A 00000005 R6       A 00000006 R7       A 00000007 RA
  A 0000000B RB       A 0000000B RETURN   D
 SUBD     D          SUBR     D          TIMSUCKS D          WRITE    D          XORD
  D          XORR     D

Assembly Phase complete.
    0 error(s) detected.
```

```
 Addr              Line MATRIX Program

                          TITLE  MATRIX Program
                          LIST F,W
                          LINES 50
                          ;/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-
/-/-/-/-/-
                          ; Matrix Mult Assembly Program for use with elRoy Systolic Array Ar
chitecture
                          ; Craig Ulmer / Darrell Stogner                                COMPE
 4500/4510
                          ;
                          ; No modifications or duplications without the authors' consent
                          ;/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-/-
/-/-/-/-/-


                          ;""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""
                          ; Multiply Matrx X by Matrix M
                          ;   Note: this will generate a transposed version of Y
                          ;
                          ;     Registers:  R7: Constant Rows of X
                          ;                 R6: X Last Address of a loop = X Starting + #Cols
*2
                          ;                 R5: X starting address
                          ;                 R4: # Columns of X * 2 = Rows of V * 2

                          ;                 R3: RA Loop       Rows = Rows of V
                          ;                 R2: Columns of M counter
                          ;                 R1: Current V Address
                          ;                 R0: Current X Address
                          ;""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""""""""

                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
                          ; Initialize the M Matrix counters
                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
00000 51D0004A            INITCODE: COPYD   R1,CONMPOS%:     ; R1=M starting Address
00001 D3100000                      LOAD    R3,R1            ; R3=Rows of M / Cols of X
00002 21D10002                      ADDD    R1,R1,H#0002     ; R1=Address of M's Columns
00003 D2100000                      LOAD    R2,R1            ; R2=Columns of M
00004 21D10002                      ADDD    R1,R1,H#0002     ; R1=Address of first M data

                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
                          ; Write out Y's ROWS,COLS values
                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
00005 57D0005D            INITY:    COPYD   R7,CONYPOS%:     ; Get the location of Y
00006 56D00037                      COPYD   R6,CONXPOS%:     ; Get the location of X
00007 D6600000                      LOAD    R6,R6            ; Get the # rows from X
00008 F0760000                      WRITE   R7,R6            ; Write the # rows to Y
00009 27D70002                      ADDD    R7,R7,H#0002     ; Point to address of Y's Cols
```

MATRIX - MATRIX Program


```
 Addr              Line MATRIX Program

0000A F0720000                      WRITE   R7,R2            ; Write the # cols to Y
0000B 60700000                      PUSH    R7               ; Put the Y address on the stack

                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
                          ; Calculate the address offset of one row of X
                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
0000C 57300000            COFFSET:  COPYR   R7,R3            ; R7=Temp counter of X columns
0000D 54D00000                      COPYD   R4,H#0000        ; R4=Col*2 offset begins at zero
0000E 24D40002            COFFLOOP: ADDD    R4,R4,H#0002     ; One more element-> offset by +two

0000F 37D70001                      SUBD    R7,R7,H#0001     ; Decrease columns counted
00010 9000000E                      JA      COFFLOOP%:       ; If still a column, loop

                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
                          ; Main loop for matrix
                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
00011 DAD000A8            MATLOOP:  CCLEAR                   ; Clear out all cells - new vector
00012 50D00037                      COPYD   R0,CONXPOS%:     ; Get X starting address
00013 D7000000                      LOAD    R7,R0            ; R7 = Num Rows of X
00014 20D00004                      ADDD    R0,R0,H#0004     ; Set to first data value of X
00015 55000000                      COPYR   R5,R0            ; Set to first data value of X
00016 26540000                      ADDR    R6,R5,R4         ; R6=address of 2nd row, 1st column
00017 35D50002                      SUBD    R5,R5,H#0002     ; Kludge R5 for looping

                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
                          ; A new vector of M. Must reinitialize the cells and counters
                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
00018 DAD08290            ADDRA:    CLRA                     ; Set to load RA pipe, holding ACCS
00019 25D50002                      ADDD    R5,R5,H#0002     ; Increase X origin by one spot
0001A B0560000                      CMP     R5,R6            ; See if we've hit the last place
0001B A000002C                      JE      PASSOUTS%:       ; If so, go to the results pass
0001C 50500000                      COPYR   R0,R5            ; if not, load the next stating X p
os
0001D 53700000                      COPYR   R3,R7            ; Reset the row counter

                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
                          ; Individual vector loop
                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
0001E DB000000            ADDLOOP:  LOAD    RA,R0            ; load next X into the RA pipe
0001F 20040000                      ADDR    R0,R0,R4         ; Move one row down
00020 33D30001                      SUBD    R3,R3,H#0001     ; Decrease the row counter
00021 9000001E                      JA      ADDLOOP%:        ; If not last one, keep looping

                          ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~
```

```
HALE Listing: matrix.asm              Mar 13 14:12:02 1995        Page   3

 Addr            Line MATRIX Program

                   85 ; Pad out the array if more Cells then # multiplying
                   86 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
00022 50D00004     87 PADCELLS: COPYD    R0,NUMCELLS     ; Get the number of cells
00023 30700000     88           SUBR     R0,R7,R0        ; R0=Num Cells - Rows
00024 A0000028     89 PADLOOP:  JE       DOMULT%:        ; Perfect fit, do the mult
00025 DBD00000     90           LOADD    RA,H#0000       ; Load a dummy into the RA pipe
00026 30D00001     91           SUBD     R0,R0,H#0001    ; Decrease counter
00027 80000024     92           JMP      PADLOOP%:       ; Keep looping
                   93
                   94 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
                   95 ; Do the actual multiplication
                   96 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
00028 DAD08110     97 DOMULT:   CLAAI                    ; Set for Multiply
00029 DB100000     98           LOAD     RB,R1           ; RA loaded, Load the next part of
 M
0002A 21D10002     99           ADDD     R1,R1,H#0002    ; Point to the next value of M
0002B 80000018    100           JMP      ADDRA%:         ; Do the next column
                  101
                  102 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
                  103 ; Write out answers
                  104 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
0002C 70000000    105 PASSOUTS: POP      R0              ; Pop off the Y-1 address
0002D DAD080D0    106           CPASS                    ; Set to pass out answers
0002E 20D00002    107 PASSLOOP: ADDD     R0,R0,H#0002    ; Point to next Y data value
0002F F0080000    108           WRITE    R0,ACCIN        ; Write current result out
00030 DBD00000    109           LOADD    RB,H#0000       ; Force a value to pop out
00031 37D70001    110           SUBD     R7,R7,H#0001    ; Decrease the counter
00032 9000002E    111           JA       PASSLOOP%:      ; Keep looping if not zero
00033 60000000    112           PUSH     R0              ; Push the Y address for storage
                  113
                  114 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
                  115 ; Figure out next Column of M
                  116 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
00034 32D20001    117 MATEND:   SUBD     R2,R2,H#0001    ; Decrease M's Column counter
00035 90000011    118           JA       MATLOOP%:       ; If more columns, keep looping
                  119
00036 80000036    120 DONE:     JMP      DONE%:
                  121
00037 00040009    122 CONXPOS:  TIMSUCKS  H#0004,H#0009
00038 00010002    123           TIMSUCKS  H#0001,H#0002
00039 00030004    124           TIMSUCKS  H#0003.H#0004
0003A 00050006    125           TIMSUCKS  H#0005,H#0006
0003B 00070008    126           TIMSUCKS  H#0007,H#0008
```

```
HALE Listing: matrix.asm              Mar 13 14:12:02 1995        Page   4

 Addr            Line MATRIX Program

0003C 00090011    127           TIMSUCKS  H#0009,H#0011
0003D 00120013    128           TIMSUCKS  H#0012,H#0013
0003E 00140015    129           TIMSUCKS  H#0014,H#0015
0003F 00160017    130           TIMSUCKS  H#0016,H#0017
00040 00180019    131           TIMSUCKS  H#0018,H#0019
00041 00210022    132           TIMSUCKS  H#0021,H#0022
00042 00230024    133           TIMSUCKS  H#0023,H#0024
00043 00250026    134           TIMSUCKS  H#0025,H#0026
00044 00270028    135           TIMSUCKS  H#0027,H#0028
00045 00290031    136           TIMSUCKS  H#0029,H#0031
00046 00320033    137           TIMSUCKS  H#0032,H#0033
00047 00340035    138           TIMSUCKS  H#0034,H#0035
00048 00360037    139           TIMSUCKS  H#0036,H#0037
00049 00380039    140           TIMSUCKS  H#0038,H#0039
0004A 00090004    141 CONMPOS:  TIMSUCKS  H#0009,H#0004
0004B 00010002    142           TIMSUCKS  H#0001,H#0002
0004C 00030004    143           TIMSUCKS  H#0003,H#0004
0004D 00050006    144           TIMSUCKS  H#0005,H#0006
0004E 00070008    145           TIMSUCKS  H#0007,H#0008
0004F 00090011    146           TIMSUCKS  H#0009,H#0011
00050 00120013    147           TIMSUCKS  H#0012,H#0013
00051 00140015    148           TIMSUCKS  H#0014,H#0015
00052 00160017    149           TIMSUCKS  H#0016,H#0017
00053 00180019    150           TIMSUCKS  H#0018,H#0019
00054 00210022    151           TIMSUCKS  H#0021,H#0022
00055 00230024    152           TIMSUCKS  H#0023,H#0024
00056 00250026    153           TIMSUCKS  H#0025,H#0026
00057 00270028    154           TIMSUCKS  H#0027,H#0028
00058 00290031    155           TIMSUCKS  H#0029,H#0031
00059 00320033    156           TIMSUCKS  H#0032,H#0033
0005A 00340035    157           TIMSUCKS  H#0034,H#0035
0005B 00360037    158           TIMSUCKS  H#0036,H#0037
0005C 00380039    159           TIMSUCKS  H#0038,H#0039
                  160
0005D 00000000    161 CONYPOS:  TIMSUCKS
0005E 00000000    162           TIMSUCKS
0005F 00000000    163           TIMSUCKS
00060 00000000    164           TIMSUCKS
00061 00000000    165           TIMSUCKS
00062 00000000    166           TIMSUCKS
00063 00000000    167           TIMSUCKS
00064 00000000    168           TIMSUCKS
```

HALE Listing: matrix.asm Mar 13 14:12:02 1995 Page 5

```
 Addr            Line MATRIX Program

00065 00000000   169          TIMSUCKS
00066 00000000   170          TIMSUCKS
00067 00000000   171          TIMSUCKS
00068 00000000   172          TIMSUCKS
00069 00000000   173          TIMSUCKS
0006A 00000000   174          TIMSUCKS
0006B 00000000   175          TIMSUCKS
0006C 00000000   176          TIMSUCKS
0006D 00000000   177          TIMSUCKS
0006E 00000000   178          TIMSUCKS
0006F 00000000   179          TIMSUCKS
00070 00000000   180          TIMSUCKS
00071 00000000   181          TIMSUCKS
00072 00000000   182          TIMSUCKS
00073 00000000   183          TIMSUCKS
                 184
```

Matrix-Matrix Multiplication

Similar to Matrix-Vector program, but Loops the Algorithm to perform All operations. Results Are stored in transposed form.

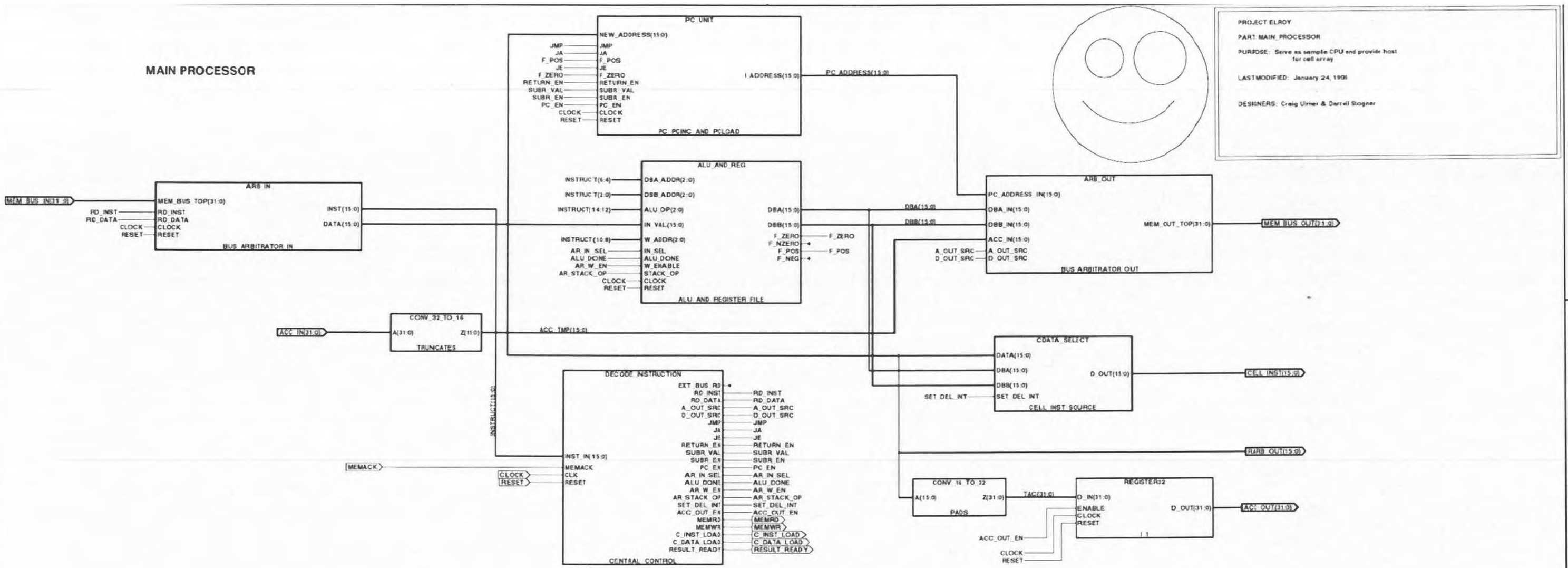
Symbol Table: matrix.asm Page 6

```
 ACCIN    A 00000008 ACCOUT   A 00000009 ADDD     D          ADDLOOP  A 0000001E ADDR
 D          ADDRA    A 00000018 ANDD     D
 ANDR     D          BLANK16  A 00000000 CCLEAR   D          CDELINT  A 0000000C CELL0
 A 00000000 CELL1    A 00000001 CELL2    A 00000002
 CELL3    A 00000003 CELLP    A 00000020 CINST    A 0000000A CLAAE    D          CLAAI
 D          CLACC    D          CLRA     D
 CMP      D          CMPD     D          COFFLOOP A 0000000E COFFSET  A 0000000C CONMPOS
 A 0000004A CONXPOS  A 00000037 CONYPOS  A 0000005D
 COPYD    D          COPYR    D          CPASS    D          CSETDEL  D          CSETDELI
 D          DOMULT   A 00000028 DONE     A 00000036
 DW       D          EXTDATA  A 0000000D INITCODE A 00000000 INITY    A 00000005 JA
 D          JE       D          JMP      D
 JSR      D          LADD     A 00000002 LAND     A 00000004 LCMP     A 0000000B LCOP
 A 00000005 LJA      A 00000009 LJE      A 0000000A
 LJMP     A 00000008 LJSR     A 0000000E LLOAD    A 0000000D LOAD     D          LOADD
 D          LOR      A 00000000 LPOP     A 00000007
 LPUSH    A 00000006 LRTS     A 0000000C LSUB     A 00000003 LWRITE   A 0000000F LXOR
 A 00000001 MATEND   A 00000034 MATLOOP  A 00000011
 NARG     A 00000000 NOP      D          NULL     A 00000000 NUMCELLS A 00000004 OPCODE
 D          ORD      D          ORR      D
 PADCELLS A 00000022 PADLOOP  A 00000024 PASSLOOP A 0000002E PASSOUTS A 0000002C POP
 D          PUSH     D          R0       A 00000000
 R1       A 00000001 R2       A 00000002 R3       A 00000003 R4       A 00000004 R5
 A 00000005 R6       A 00000006 R7       A 00000007
 RA       A 0000000B RB       A 0000000B RETURN   D          SUBD     D          SUBR
 D          TIMSUCKS D          WRITE    D
 XORD     D          XORR     D
```

Assembly Phase complete.
0 error(s) detected.

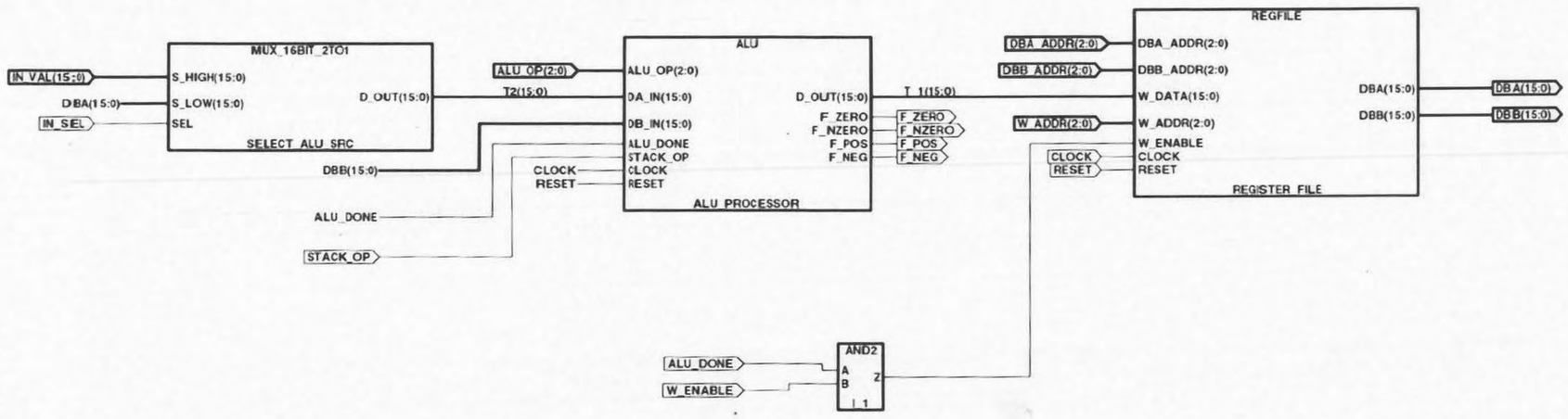
Appendix C: Circuit Schematics

MAIN PROCESSOR

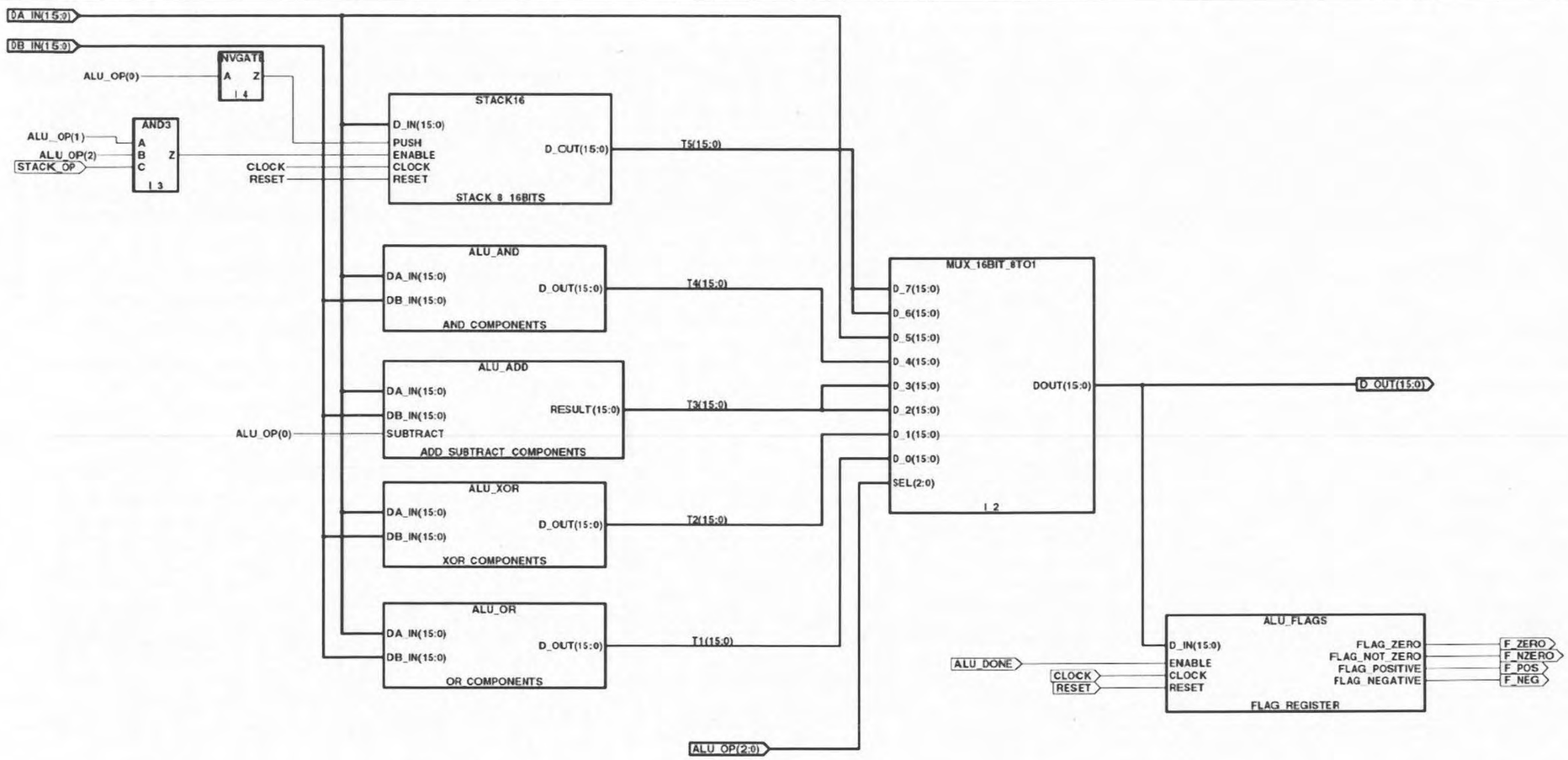


MAIN PROCESSOR

ALU + Regfile


MUX_16BIT_2TO1
IN_VAL(15:0)
DBA(15:0)
IN_SEL
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
SELECT_ALU_SRC
T2(15:0)
ALU_OP(2:0)
ALU
ALU_OP(2:0)
DA_IN(15:0)
DB_IN(15:0)
ALU_DONE
STACK_OP
CLOCK
RESET
D_OUT(15:0)
F_ZERO
F_NZERO
F_POS
F_NEG
ALU_PROCESSOR
T_1(15:0)
DBB(15:0)
ALU_DONE
STACK_OP
REGFILE
DBA_ADDR(2:0)
DBB_ADDR(2:0)
W_DATA(15:0)
W_ADDR(2:0)
W_ENABLE
CLOCK
RESET
REGISTER_FILE
DBA(15:0)
DBB(15:0)
AND2
A
B
Z
1

ALU



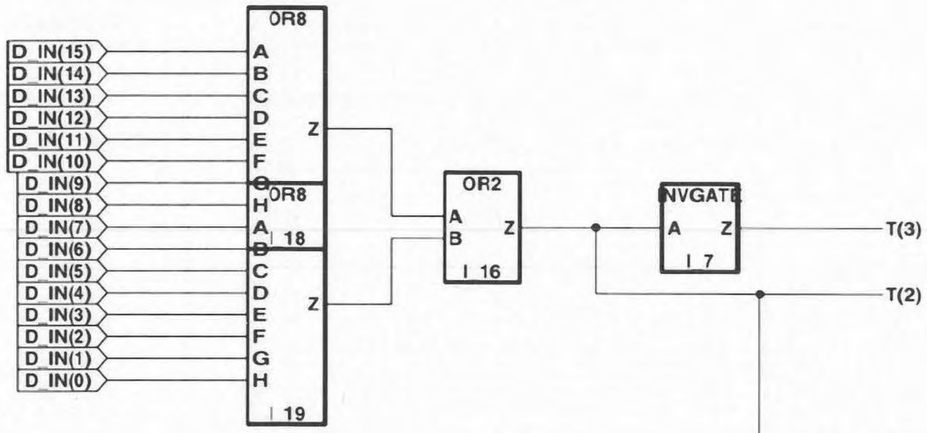
PROJECT ELROY

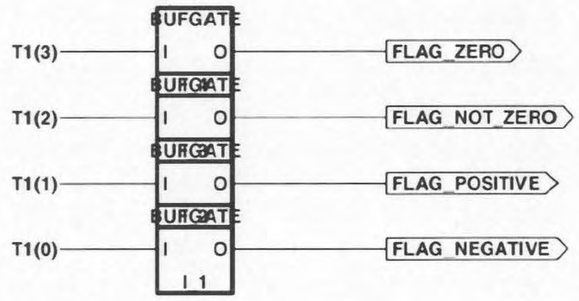
PART: ALU

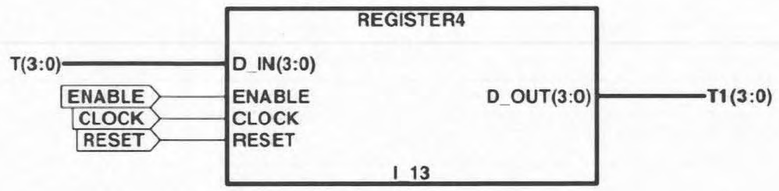
PURPOSE: Overall ALU with Flags

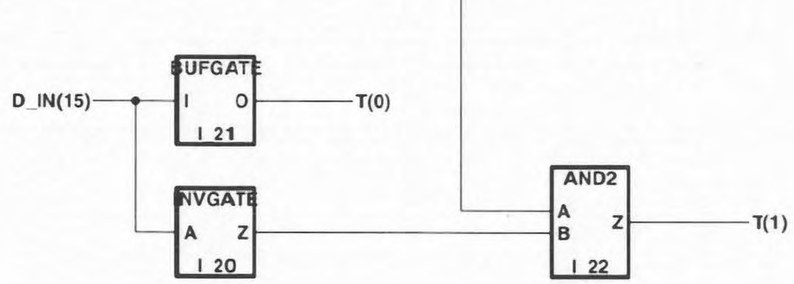
LAST MODIFIED: January 24, 1995

DESIGNERS: Craig Ulmer & Darrell Stogner


OR8
D_IN(15)
D_IN(14)
D_IN(13)
D_IN(12)
D_IN(11)
D_IN(10)
D_IN(9)
D_IN(8)
D_IN(7)
D_IN(6)
D_IN(5)
D_IN(4)
D_IN(3)
D_IN(2)
D_IN(1)
D_IN(0)
A
B
C
D
E
F
G
H
Z
I 19
OR8
I 18
OR2
I 16
INVGATE
I 7
T(3)
T(2)

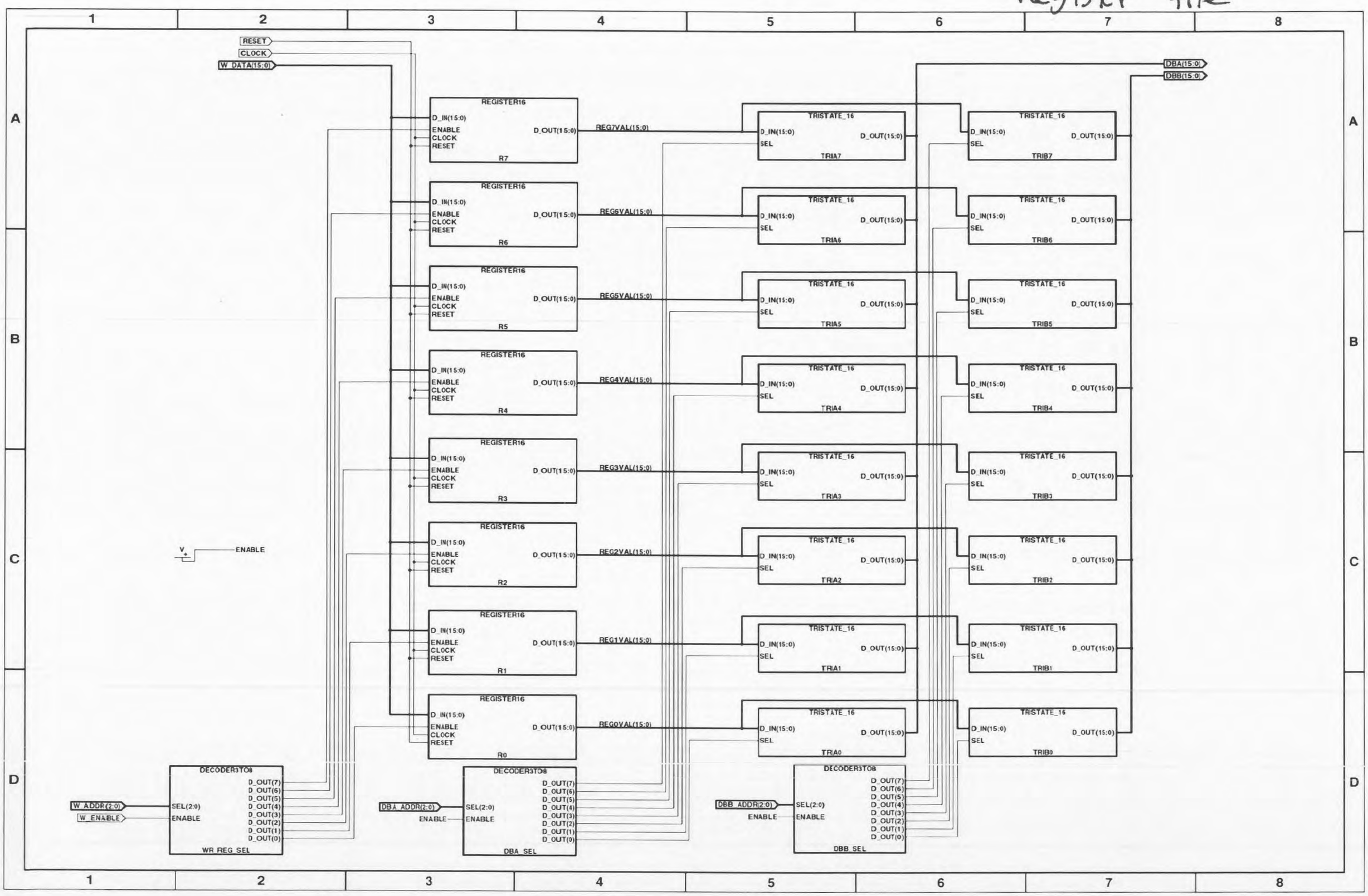

BUFGATE
T1(3)
FLAG_ZERO
T1(2)
FLAG_NOT_ZERO
T1(1)
FLAG_POSITIVE
T1(0)
FLAG_NEGATIVE
I
O
I 1


REGISTER4
T(3:0)
D_IN(3:0)
ENABLE
CLOCK
RESET
D_OUT(3:0)
T1(3:0)
I 13

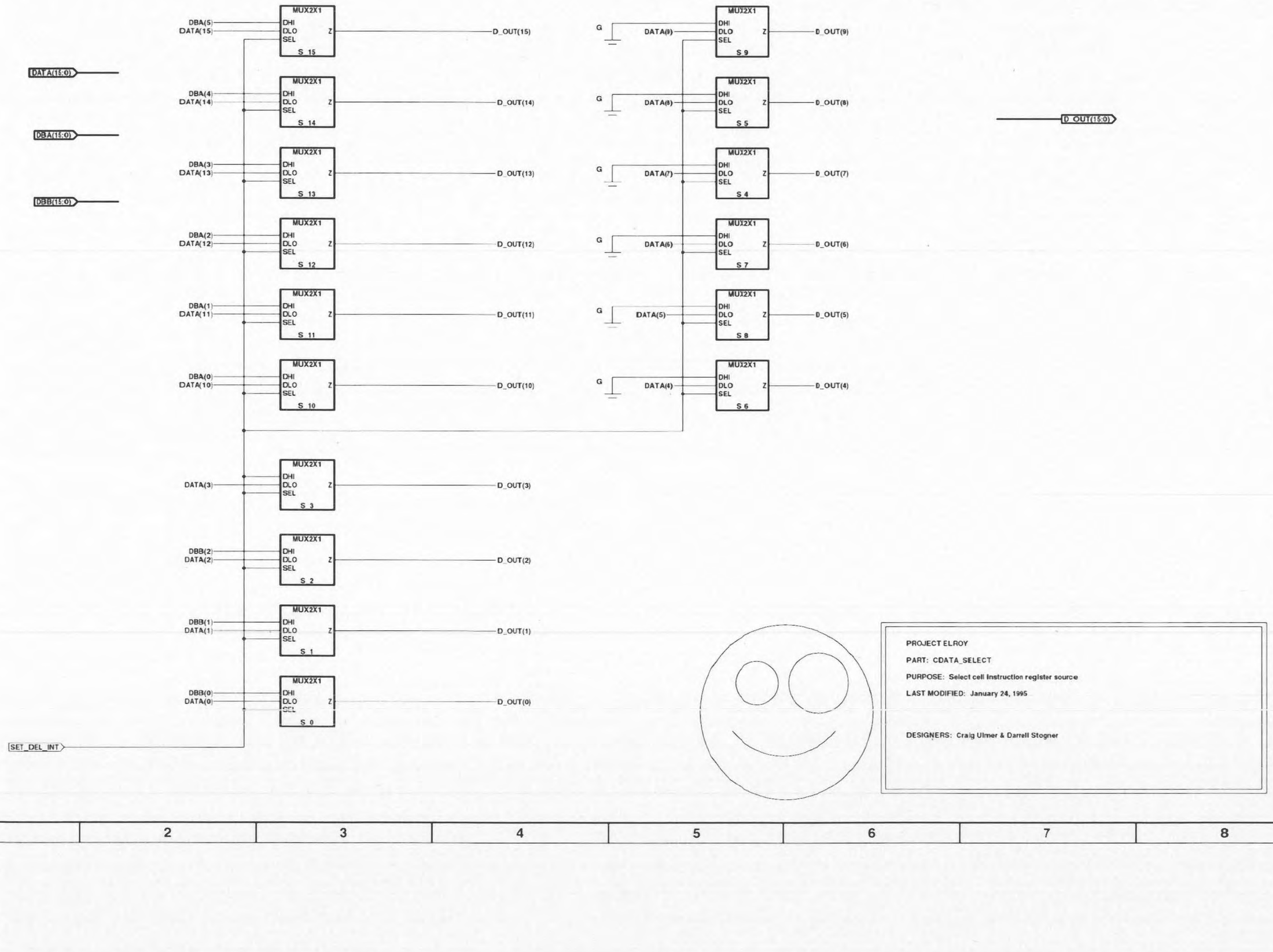

BUFGATE
D_IN(15)
I
O
I 21
T(0)
INVGATE
A
Z
I 20
AND2
B
I 22
T(1)

ALU FLAGS

Register file



CDATA Select



PROJECT ELROY

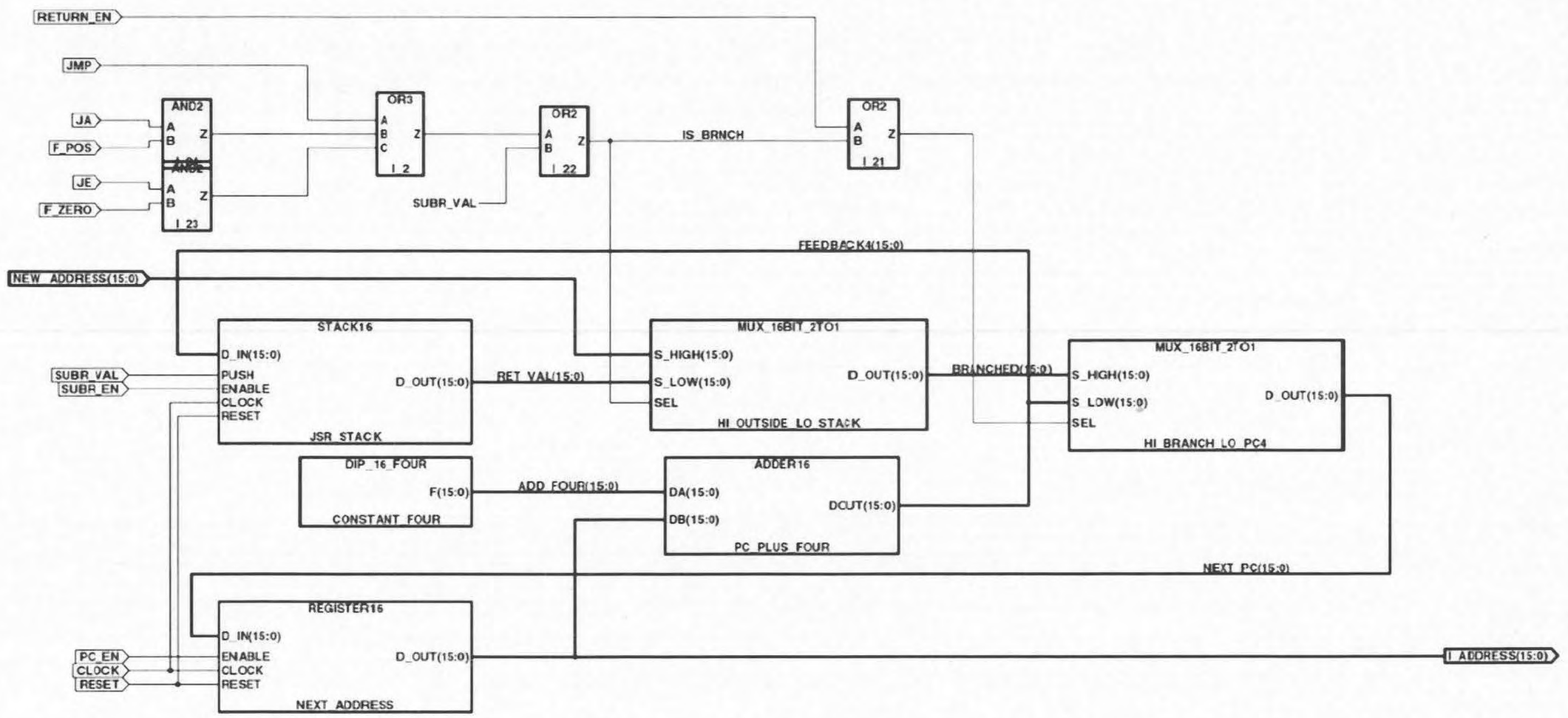
PART: CDATA_SELECT

PURPOSE: Select cell instruction register source

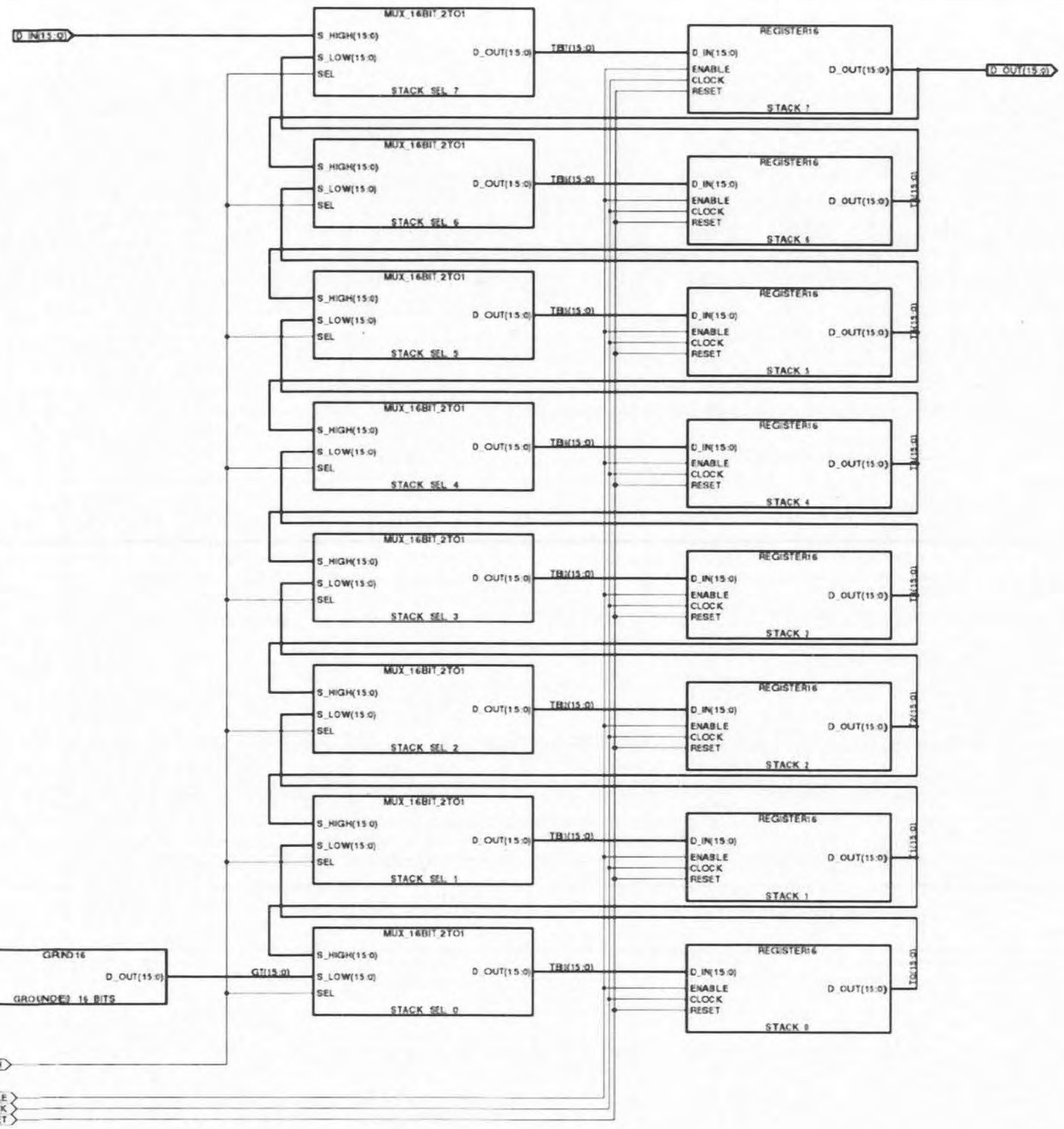
LAST MODIFIED: January 24, 1995

DESIGNERS: Craig Ulmer & Darrell Stogner

PC Unit

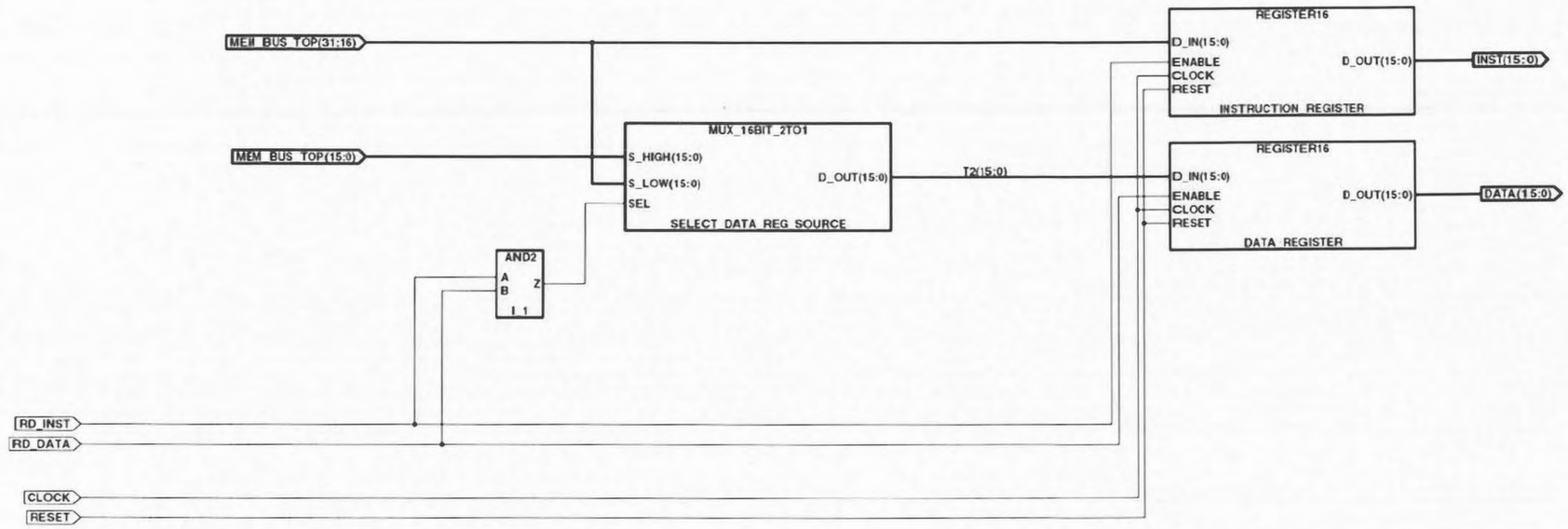

RETURN_EN
JMP
JA
F_POS
JE
F_ZERO
AND2
I_23
OR3
I_2
OR2
I_22
SUBR_VAL
IS_BRNCH
OR2
I_21
FEEDBACK4(15:0)
NEW_ADDRESS(15:0)
STACK16
D_IN(15:0)
PUSH
ENABLE
CLOCK
RESET
D_OUT(15:0)
JSR_STACK
SUBR_VAL
SUBR_EN
RET_VAL(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
HI_OUTSIDE_LO_STACK
BRANCHED(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
HI_BRANCH_LO_PC4
DIP_16_FOUR
F(15:0)
CONSTANT_FOUR
ADD_FOUR(15:0)
ADDER16
DA(15:0)
DB(15:0)
DOUT(15:0)
PC_PLUS_FOUR
NEXT_PC(15:0)
REGISTER16
D_IN(15:0)
ENABLE
CLOCK
RESET
D_OUT(15:0)
NEXT_ADDRESS
PC_EN
CLOCK
RESET
I_ADDRESS(15:0)

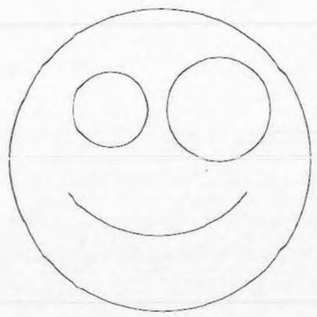
Stack


D_IN(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
STACK_SEL_7
STACK_SEL_6
STACK_SEL_5
STACK_SEL_4
STACK_SEL_3
STACK_SEL_2
STACK_SEL_1
STACK_SEL_0
TRI(15:0)
REGISTER16
D_IN(15:0)
ENABLE
CLOCK
RESET
STACK_7
STACK_6
STACK_5
STACK_4
STACK_3
STACK_2
STACK_1
STACK_0
GRND16
GROUNDED 16 BITS
GTI(15:0)
PUSH
ENABLE
CLOCK
RESET

Bus ARB IN

ARB_IN





PROJECT ELROY

PART: ARB_IN

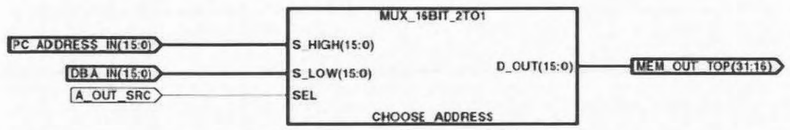
PURPOSE: Handle input registers and memory bus

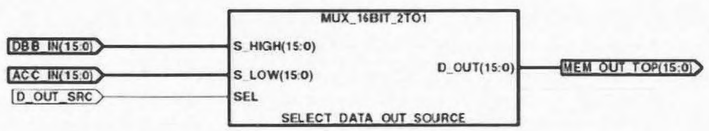
LAST MODIFIED: January 24, 1995

DESIGNERS: Craig Ulmer & Darrell Stogner

Bus Arb Out

ARB_OUT





PROJECT ELROY

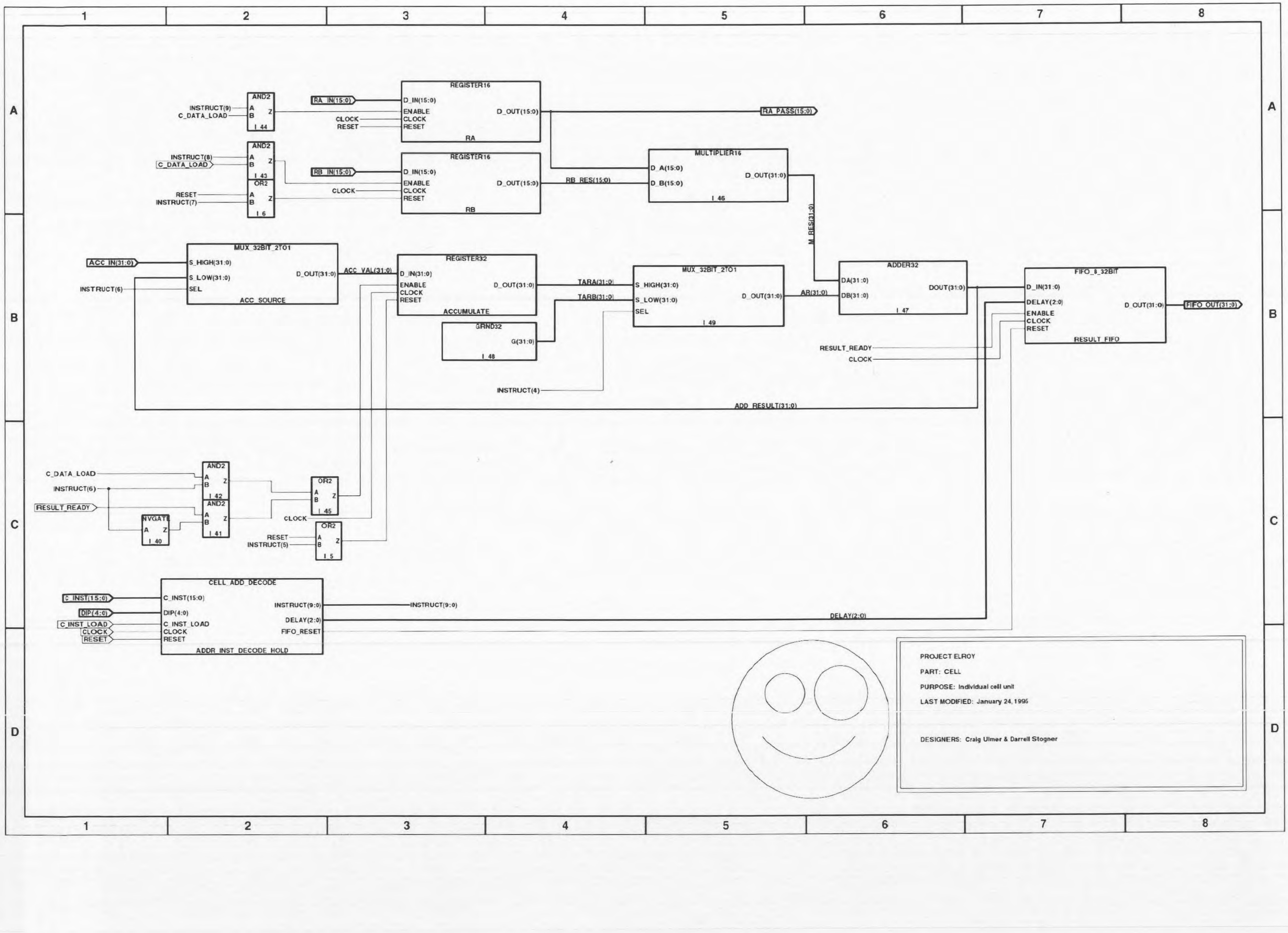
PART: ARB_OUT

PURPOSE: Handle output registers and memory bus

LAST MODIFIED: January 24, 1995

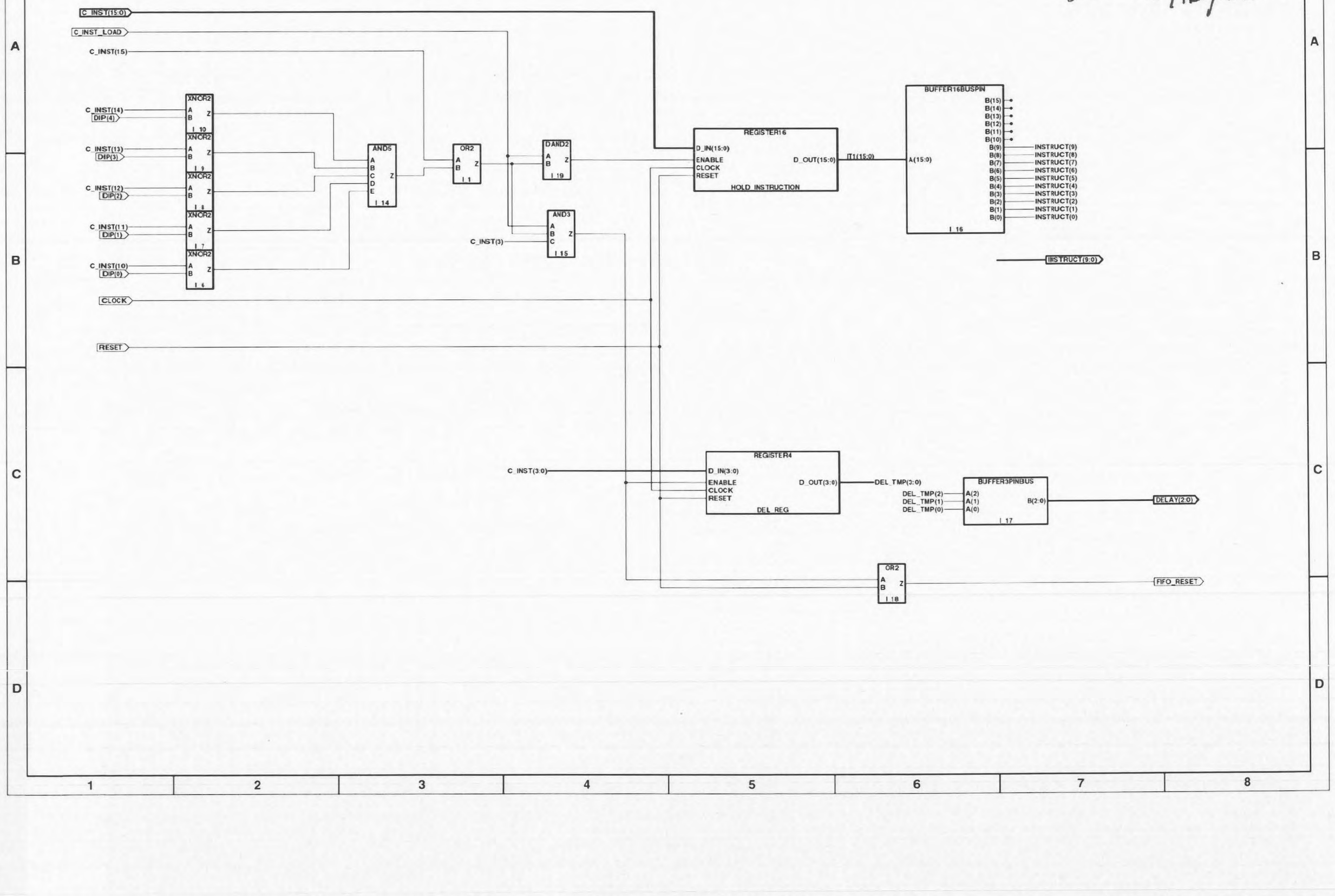
DESIGNERS: Craig Ulmer & Darrell Stogner

CELL

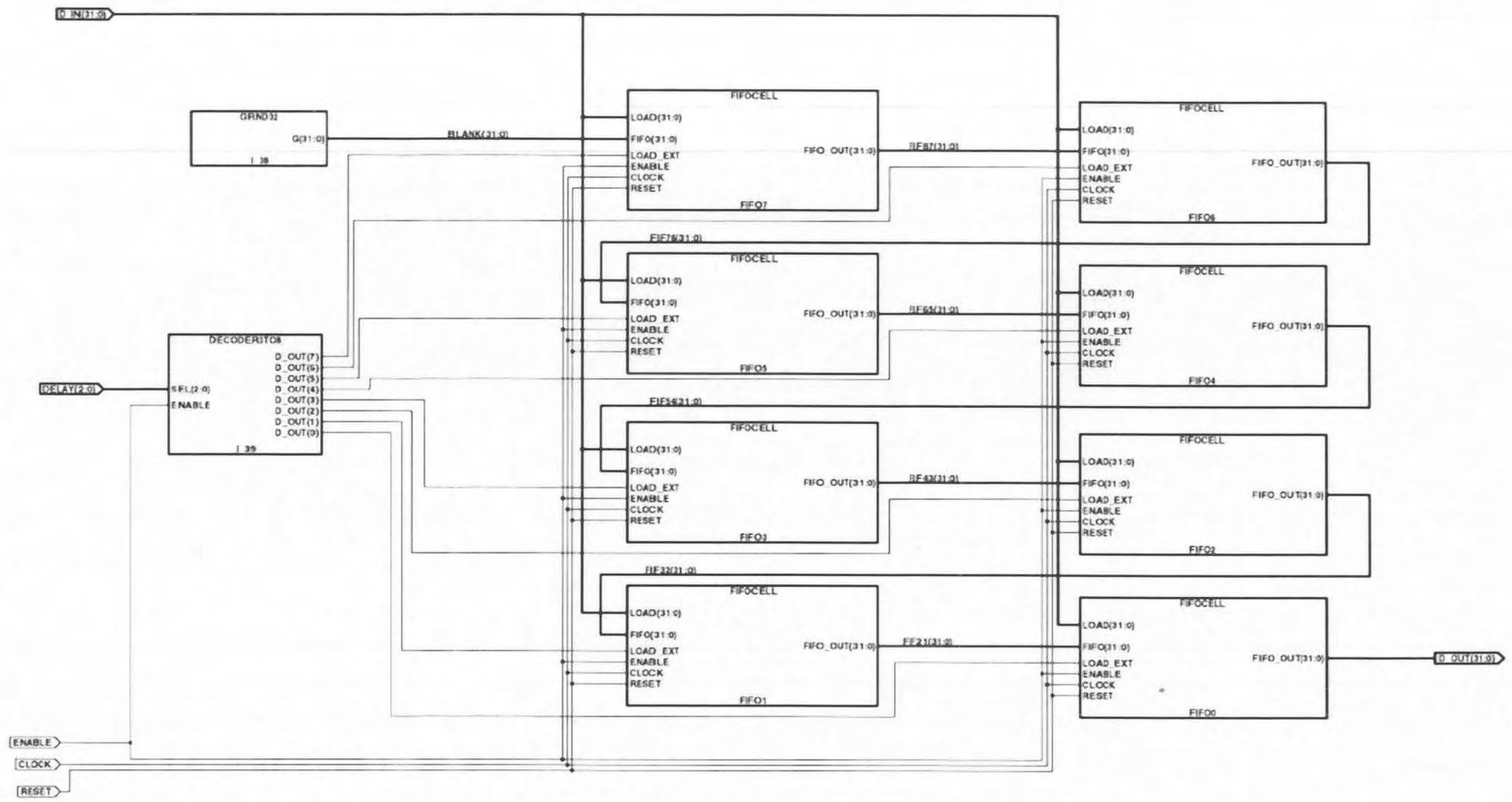

AND2
INSTRUCT(9)
C_DATA_LOAD
I_44
RA_IN(15:0)
REGISTER16
D_IN(15:0)
ENABLE
CLOCK
RESET
D_OUT(15:0)
RA
CLOCK
RESET
RA_PASS(15:0)
AND2
INSTRUCT(8)
C_DATA_LOAD
I_43
OR2
RESET
INSTRUCT(7)
I_6
RB_IN(15:0)
REGISTER16
D_IN(15:0)
ENABLE
CLOCK
RESET
D_OUT(15:0)
RB
CLOCK
RB_RES(15:0)
MULTIPLIER16
D_A(15:0)
D_B(15:0)
D_OUT(31:0)
I_46
M_RES(31:0)
ACC_IN(31:0)
MUX_32BIT_2TO1
S_HIGH(31:0)
S_LOW(31:0)
SEL
D_OUT(31:0)
ACC_SOURCE
INSTRUCT(6)
ACC_VAL(31:0)
REGISTER32
D_IN(31:0)
ENABLE
CLOCK
RESET
D_OUT(31:0)
ACCUMULATE
TABA(31:0)
TABB(31:0)
GRND32
G(31:0)
I_48
MUX_32BIT_2TO1
S_HIGH(31:0)
S_LOW(31:0)
SEL
D_OUT(31:0)
I_49
AR(31:0)
ADDER32
DA(31:0)
DB(31:0)
DOUT(31:0)
I_47
FIFO_8_32BIT
D_IN(31:0)
DELAY(2:0)
ENABLE
CLOCK
RESET
D_OUT(31:0)
RESULT_FIFO
FIFO_OUT(31:0)
RESULT_READY
CLOCK
INSTRUCT(4)
ADD_RESULT(31:0)
C_DATA_LOAD
INSTRUCT(6)
RESULT_READY
AND2
I_42
AND2
I_41
NVGATE
I_40
OR2
I_45
CLOCK
OR2
RESET
INSTRUCT(5)
I_5
CELL_ADD_DECODE
C_INST(15:0)
DIP(4:0)
C_INST_LOAD
CLOCK
RESET
C_INST(15:0)
DIP(4:0)
C_INST_LOAD
CLOCK
RESET
INSTRUCT(9:0)
DELAY(2:0)
FIFO_RESET
ADDR_INST_DECODE_HOLD
INSTRUCT(9:0)
DELAY(2:0)
PROJECT ELROY
PART: CELL
PURPOSE: Individual cell unit
LAST MODIFIED: January 24, 1995
DESIGNERS: Craig Ulmer & Darrell Stogner

Cell Instruction

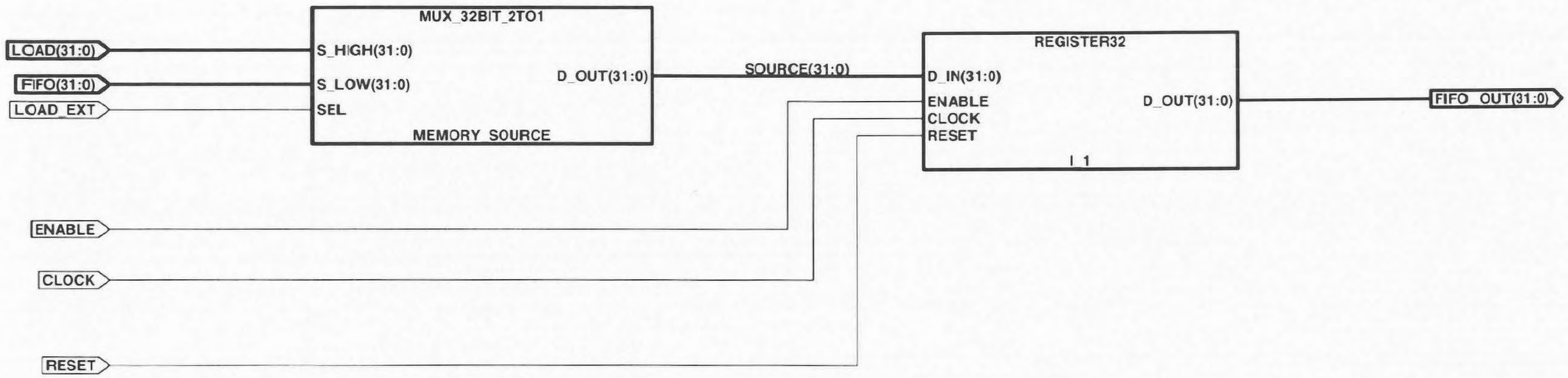
Decode/Register



Fifo Queue


D_IN(31:0)
GRND32
G(31:0)
I_38
BLANK(31:0)
DECODER3TO8
D_OUT(7)
D_OUT(6)
D_OUT(5)
D_OUT(4)
D_OUT(3)
D_OUT(2)
D_OUT(1)
D_OUT(0)
SEL(2:0)
ENABLE
I_39
DELAY(2:0)
FIFOCELL
LOAD(31:0)
FIFO(31:0)
LOAD_EXT
ENABLE
CLOCK
RESET
FIFO_OUT(31:0)
FIFO7
FIFO6
FIFO5
FIFO4
FIFO3
FIFO2
FIFO1
FIFO0
FIF76(31:0)
FIF65(31:0)
FIF54(31:0)
FIF43(31:0)
FIF32(31:0)
FIF21(31:0)
D_OUT(31:0)
ENABLE
CLOCK
RESET

Fifo Cell


MUX_32BIT_2TO1
LOAD(31:0)
FIFO(31:0)
LOAD_EXT
S_HIGH(31:0)
S_LOW(31:0)
SEL
D_OUT(31:0)
MEMORY_SOURCE
SOURCE(31:0)
REGISTER32
D_IN(31:0)
ENABLE
CLOCK
RESET
D_OUT(31:0)
I_1
FIFO_OUT(31:0)
ENABLE
CLOCK
RESET

Appendix D: Bibliography

Bibliography

Ashenden, P. J., *The VHDL Cookbook*, 1st ed., University of Adelaide, South Australia, Department of Computer Science, 1990.

Baker, Louis, *VHDL Programming*, John Wiley & Sons, Inc., New York, New York, 1993.

Kung, H. T., *Warp Experience: We Can Map Computations Onto a Parallel Computer Efficiently*, Carnegie Mellon University, Department of Computer Science, 1988.

Kung, H. T., *Why Systolic Architectures?*, Carnegie Mellon University, Department of Computer Science, 1982.

Mead, C., Conway, L., *Introduction to VLSI Systems*, Addsion- Wesley Publishing, 2nd. ed., 1980.

Patterson, D. A., and Hennessy, J. L., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Mateo, California, 1994.

Schafer, R. W., and Openheim, A. V., *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.

Strang, Gilbert, *Linear Algebra and Its Applications*,3rd ed., Harcourt Brace Jovanovich College Publishers, Orlando, FL, 1986.

Synopsys, Inc., *Synopsys: VSS Family Tutorial*, 1994.

Texas Instruments, *User's Guide: Digital Signal Processing Products*, Texas Instruments Incorporated, 1990.

Wakerly, J. F., *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.