Rochelle
Williams
end of cor

Georgia Institute of Technology

School of Electrical and Computer Engineering

elRoy

A Systolic Processor Array

CompE 4500 Senior Design

Fall 1994

Darrell Stogner
Craig Ulmer

Table of Contents

Purpose	1
Systolic Array Theory	2
Vector Multiplication Theory	4
Discrete Convolution Theory	8
System Design and Architecture	13
Testing elRoy	18
Programming	19
Assembly Instruction Format	20
Assembly Language Instructions	21
Cell Microcode Format	22
The Convolution Operation	23
Writing an Assembly Convolution Program	24
Example Convolution Assembly Program	25
System Software	26
Speed Comparisons	27
Tool Evaluation	28
Conclusions	29
Appendix A: TIM Assembly	
Appendix B: Test Programs and Timing Diagrams	
Appendix C: Circuit Schematics	
Appendix D: Figures List	
Appendix E: Bibliography	

Purpose

The purpose of this project is to explore the use of systolic array architectures to find a better way of computing high computation mathematical functions. Typically this kind of design allows large scale calculations to be determined in a small percentage of the time a traditional serial processor would take. The obvious applications are in fields such as Digital Signal Processing or higher order mathematics jobs.

The project implements an algorithm that would normally be found as a coprocessor on a larger system. However, to test the algorithm a small scale processor is required to feed the systolic array proper information and handle outside computing responsibilities. The overall system is required to have a good deal of flexibility to allow the design to adapt to different challenges. Additionally, the system must be realizable in hardware under uncertain specifications. ?

While this project is intended for only small scale use, it could easily have it's functionality increased simply by expanding it's busses. Since expandability is a key issue, future upgrades were considered with a great amount of priority.

Systolic Array Theory

There are several mathematical operations that demand 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 divides the monotonous task of multiplying parallel data at the same time with several simple processors.

At a glance, the easiest way for implementing a systolic processor array for an operation such as vector multiplication would be to use a fan out design, where each processor cell is assigned two numbers to multiply and the results of all the cells are added together with a large adder.

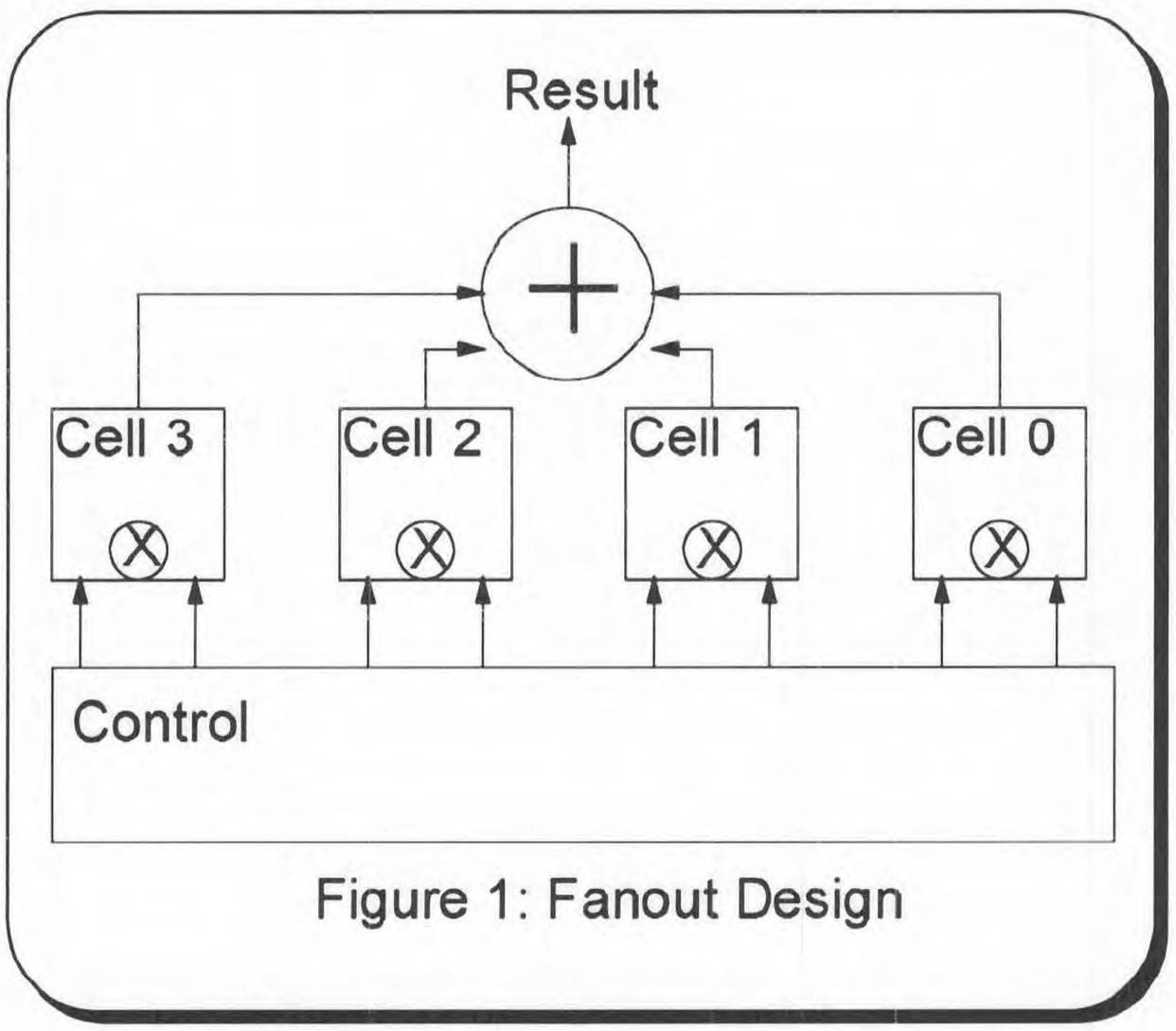


Figure 1: Fanout Design

Although easier to build and understand, the fan out approach (figure 1) suffers a weakness in the result adder. In addition to being difficult to build an adder with several inputs, it is almost impossible to expand the processor array with additional cells without having to redesign the adder unit.

The next strategy for designing the array consists of breaking the adder up into small stages that can be implemented within the cells. To build a system that has a result pipeline, connections must be made between the cells to pass along and modify results.

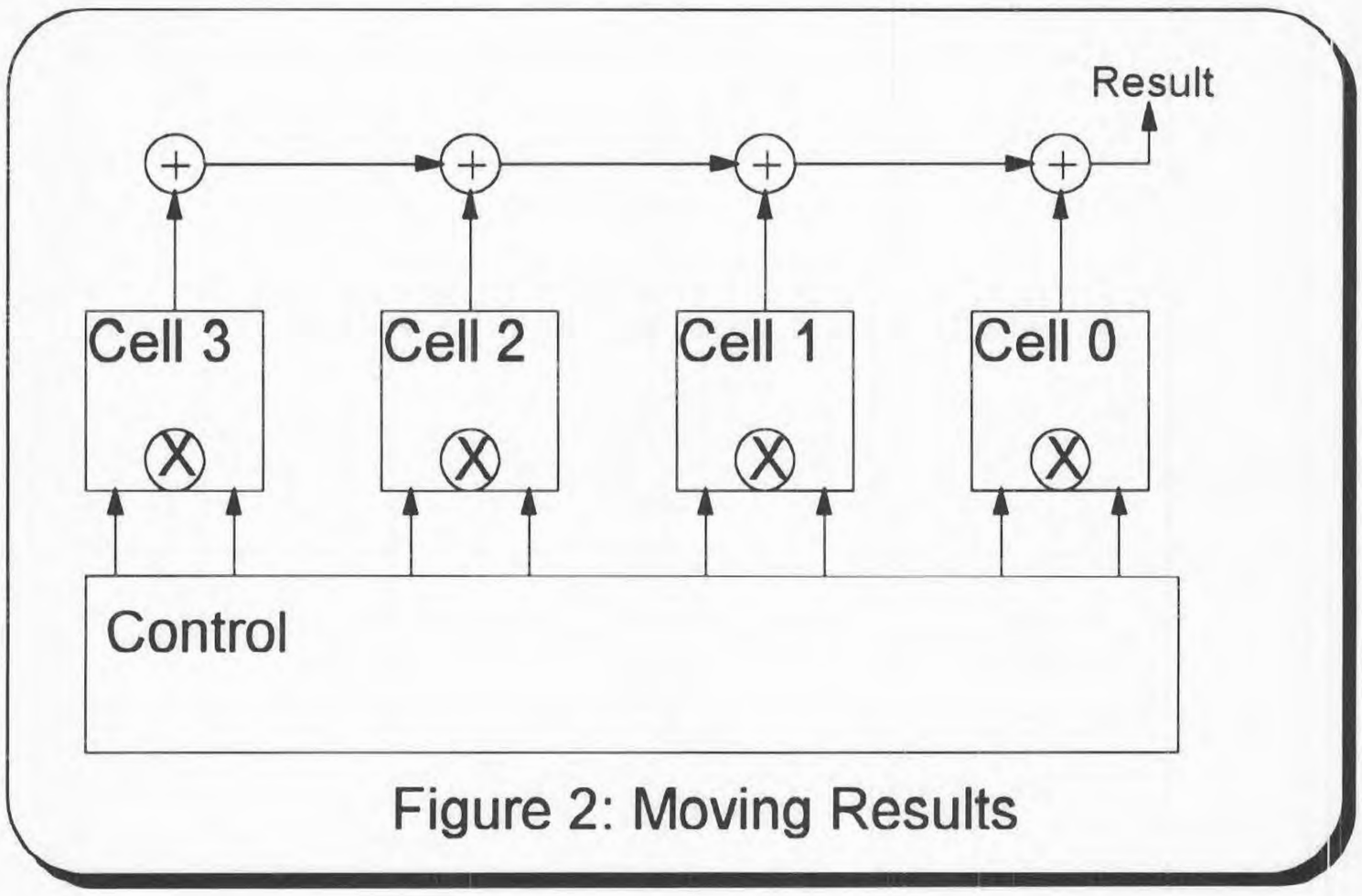


Figure 2: Moving Results

The implementation where inputs are broadcasted, results move, and weights stay (figure 2) allows easy expansion of the array. The moving results configuration of cells provided the fundamental design in the implementation of elRoy.

Other topologies for array configuration involve designs with 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.

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 kth element of the vector is parallel-loaded into the Cell register.
2) The kth 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.

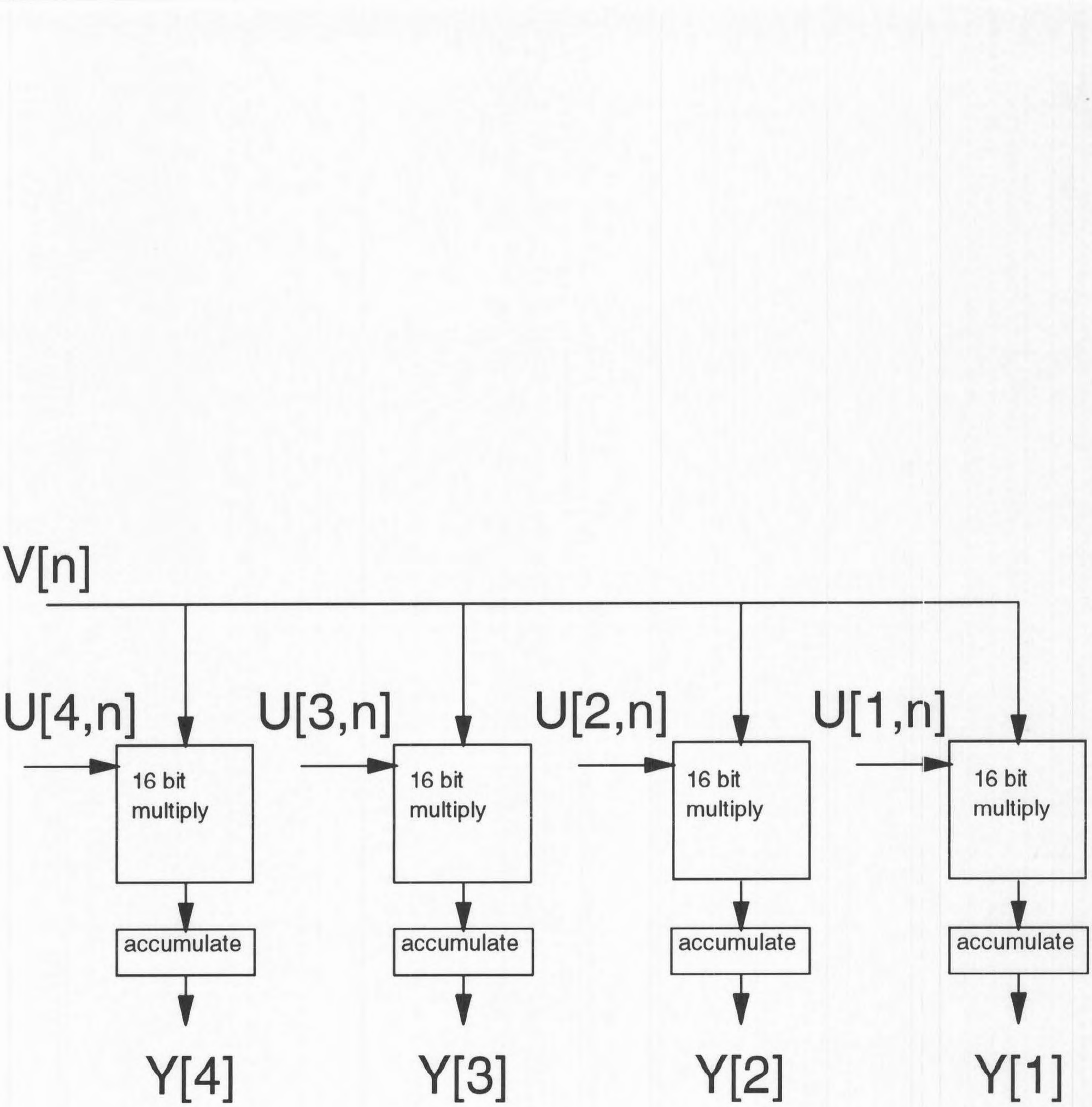
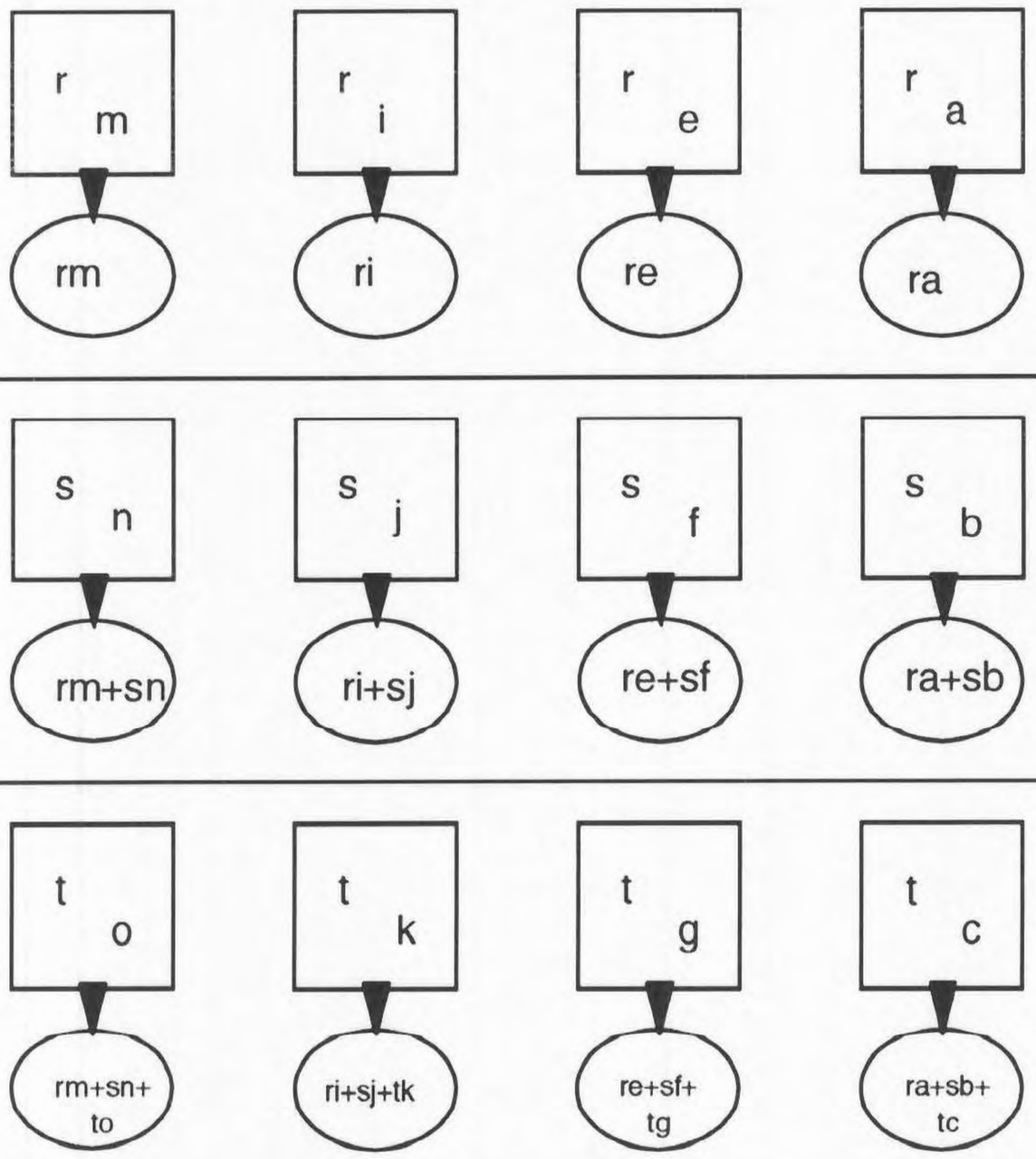


Figure 3: Vector Multiplication Configuration



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 calculations ~~needed~~ 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:

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

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. In order to eliminate gaps in the H equation. 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.

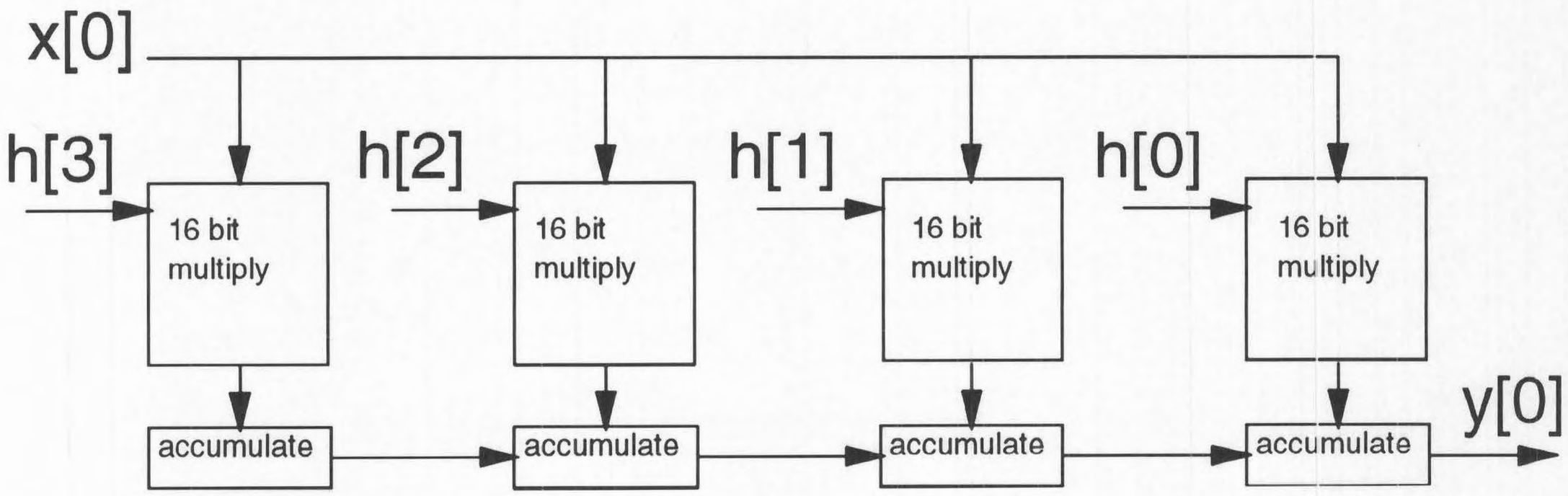


Figure 5:Discrete Convolution Configuration

Discrete Convolution

[a 0 b c d]*[e f g h]

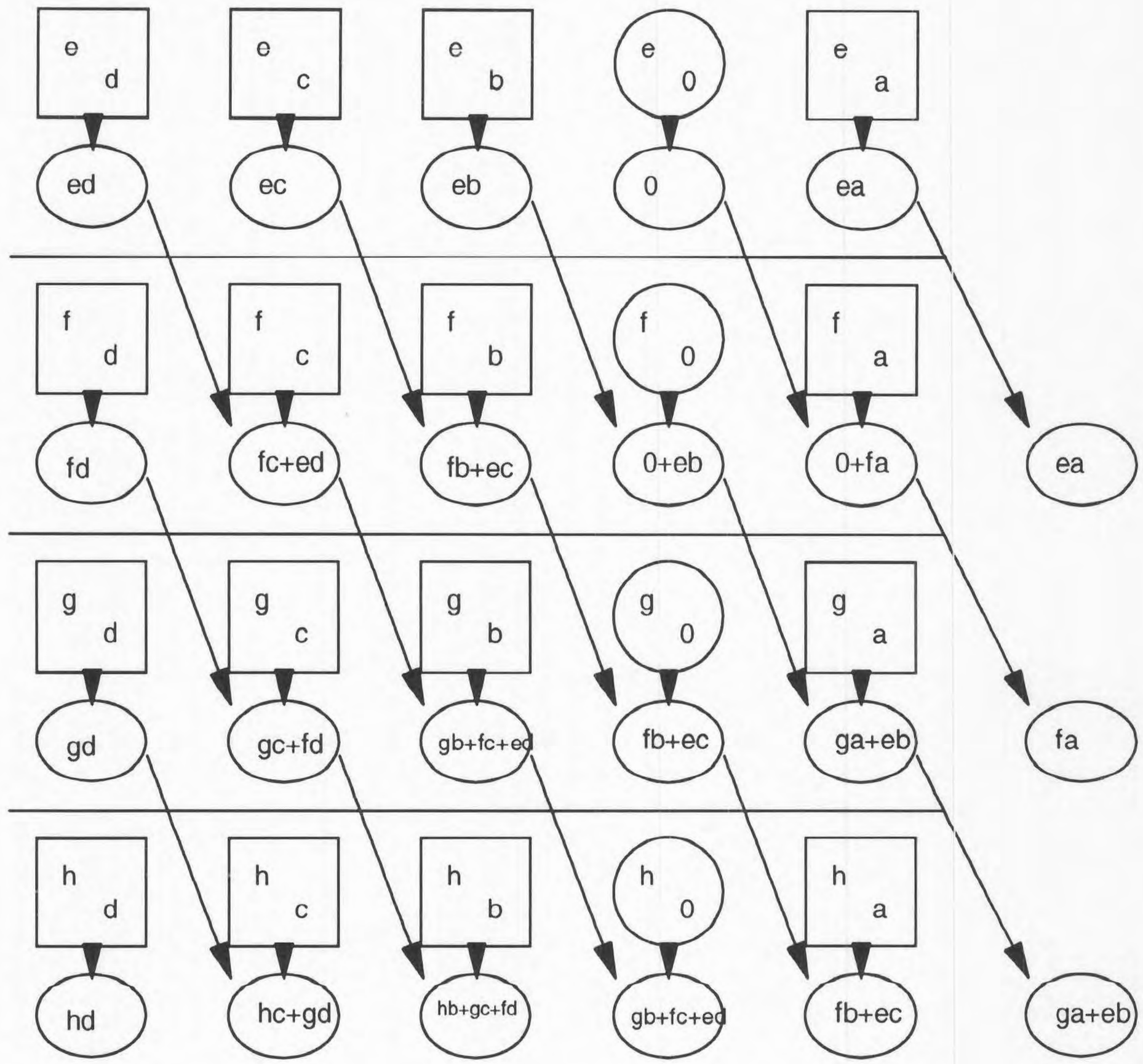


Figure 6: Convolution Pipe

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.

System Design and Architecture

elRoy is a systolic processor array, and by definition it's architecture (figure 7) is built around the multiply-accumulate cell. The overall array *is* controlled by a main processor, but the actual cell is where the main work is done.

The cell (figure 8,10) is basically composed of four components: decode, multiply, accumulate, and FIFO . There is a block used to decode the instruction received from the main processor. This was designed down to the gate level. It basically uses combinational logic to run the reset and enable lines for all the cell components. The second component is the multiply unit. It takes in two 16-bit sign-magnitude numbers and multiplies them using a simple shift-add algorithm. The 32-bit output is converted to twos complement representation. The third block is the accumulate unit. It adds the output of the multiplier to either the value calculated on the previous cycle or a new accumulator value passed in from the previous cell in the array. The final component is the FIFO queue. It is used to hold accumulates in the cell for a certain number of clock cycles. This feature is used to eliminate wasted cycles due to multiplication by zero.

The other main block in elRoy is the main control unit (figure 9) used to handle the piping of data and instructions to the array. It contains a single processor with a limited instruction set. The processor consists of an ALU, program counter, state machine and memory bus. The ALU has a register file inside as well as several different components for computing sums, differences and several boolean operations. The program counter is used to fetch instructions from memory. It has a stack and does support function calls. The memory bus is used to fetch instructions and data from memory, and to write data back to memory. The state machine is a complex VHDL description of the processor behavior. It is currently only partially coded.

The cells are arranged in a linear sequence. Theoretically the array can be of any size. DIP switches are set outside each cell. The instruction sent from the main processor has a cell address field. The cells compare the address field with the DIP settings to see if it should load in the new instruction. The main cell has a parallel data load to all cells and a sequential data load to the first cell in the array.

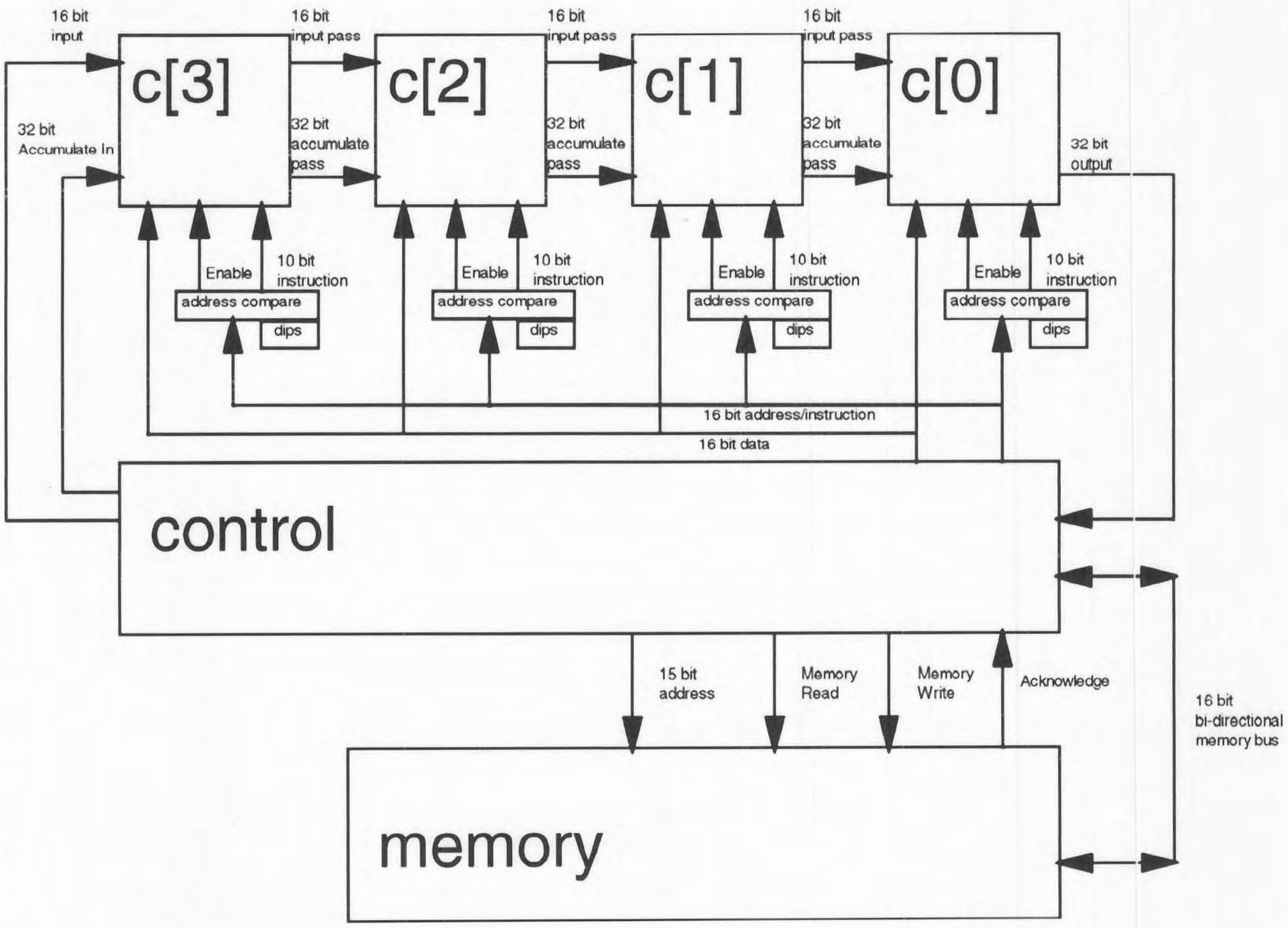


Figure 7: elRoy Flow Diagram

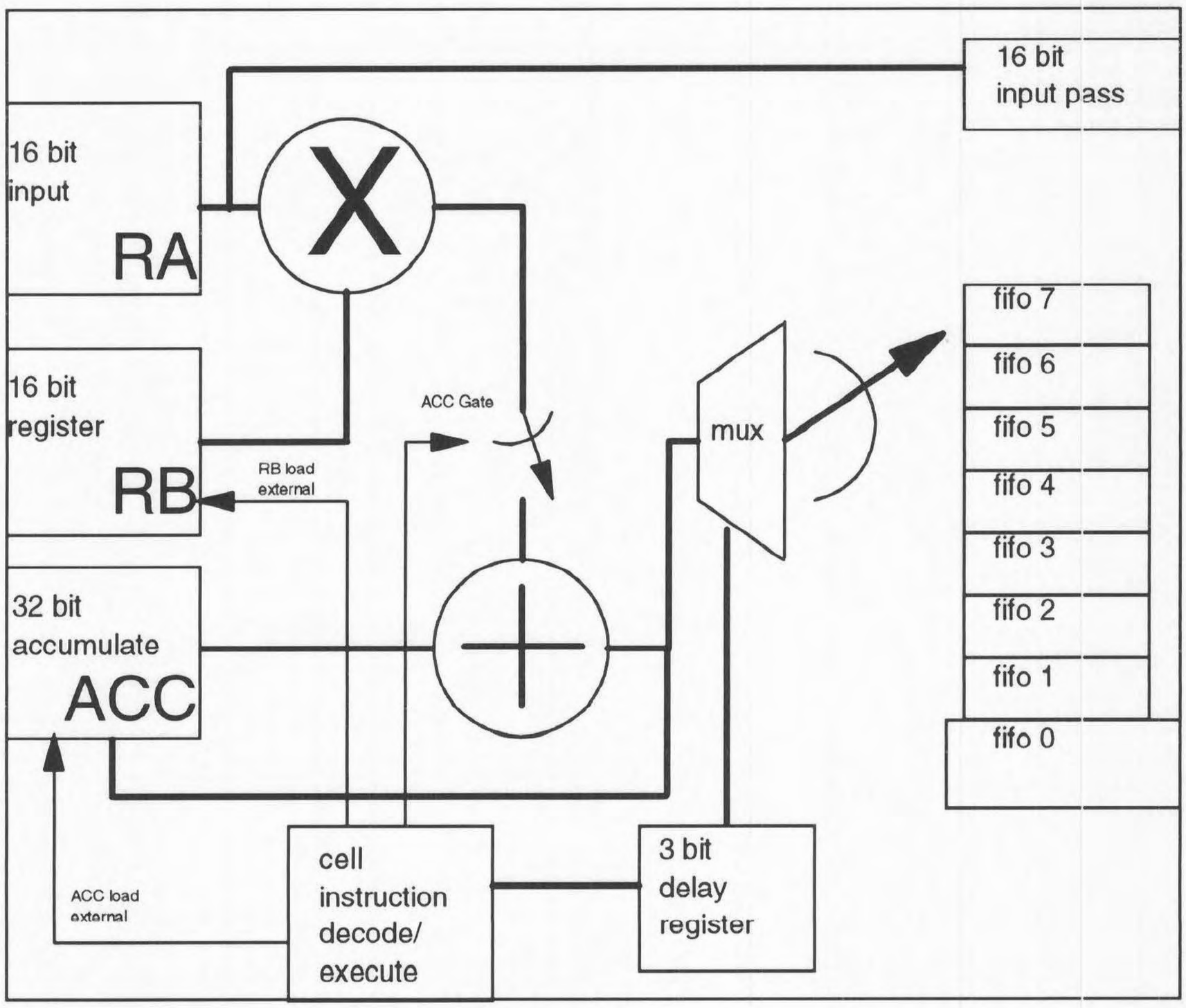


Figure 8:Individual Cell

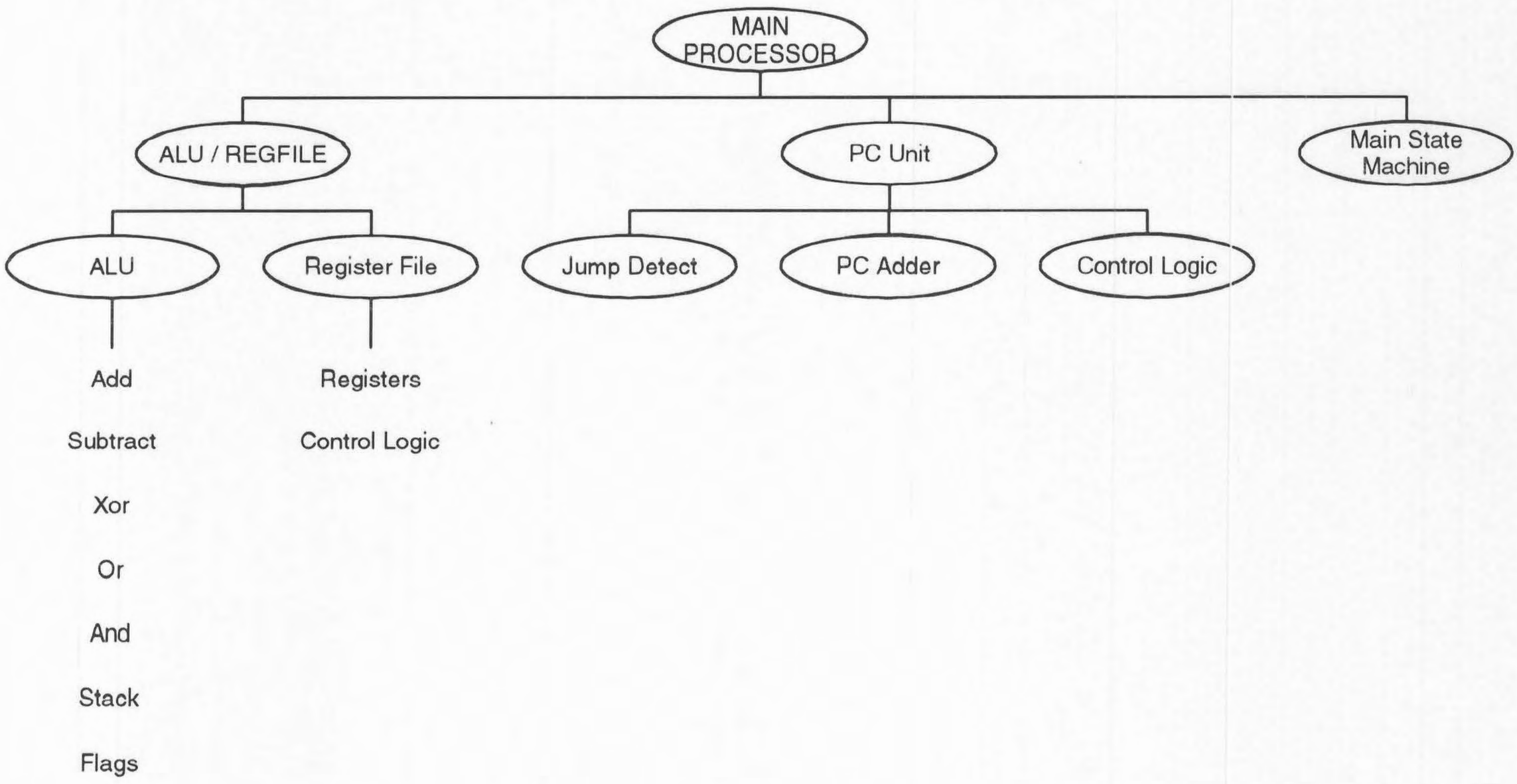


Figure 9: Main Processor Components

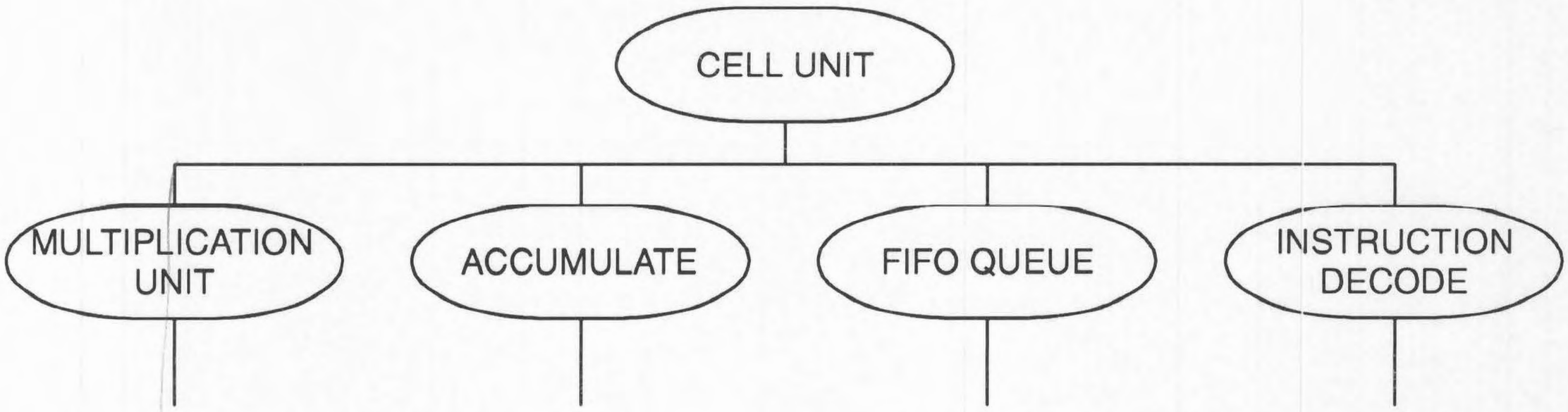


Figure 10: Cell Components

Testing

Testing elRoy was a major undertaking. The process was done using a bottom-up methodology. Individual components such as registers, adders and muxes were tested in the Synopsys simulator. The basic process was to create a test block schematic for each part and run the analyzer on it. A test file was then edited with a sequence of instructions used to test all the different cases for the element. Equivalence partitioning was used to divide the valid inputs up into major sections, and then representative examples of each equivalence instance were run through the test.

Once the bottom level elements were tested we bumped up one level in the design hierarchy and tested these components. The same process was used, although due to the complexity of the higher level parts, more vectors of conditions had to be tested. An example of components at this level would be the ALU, the FIFO queue and the multiplier.

This process culminated in the testing of the multiply-accumulate cell. The control signals used by the main processor to control the cell were simulated. The first stage in cell testing was to ensure that the control signals could fully manipulate the elements of the cell (the multiplier, the accumulator and the FIFO queue). Once this was accomplished testing proceeded to a full scale multiply-accumulate sequence. Minor bugs were encountered and fixed. The final stage of testing involved feeding the cell the inputs and controls it would see if it were taking part in a convolution of two arrays of size four.

The vectors [5 3 6 4] and [1 2 3 4] were convolved by hand. It was calculated that the end cell of a four processor array would have '4' parallel loaded into one register. It would have 5 - 3 - 6 - 4 passed to it as inputs to the other register on every multiply-accumulate cycle. Furthermore, 25 - 26 - 12 - 0 would be the accumulates passed into the cell. The control signals were inputted manually, and the output sequence [45 38 36 16] was generated, which is the correct output.

This last test was particularly illustrative. The exact timing necessary to propagate the data through the cells was difficult to get down, but after several attempts the data flow was seamless. The timing chart can be used to write the state machine for the convolution instruction directly.

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 consists of two main processor components, the main processor (or central control unit) and the array of cells. Within the main processor is a small scale ALU, a register file, and the system control logic. Actual program execution occurs in the main processor and particular computations are delegated out to individual cells as the main processor interprets instructions. The main processor communicates with the cells by issuing microcode words to the cells and sending control signals as the appropriate times. A cell interprets the last microcode word received and executes it's instruction when the proper signals are sent from the main processor.

The assembly language provides a means for controlling cells through specialized instructions. However, if the user needs to access cells directly, he or she can write specific microcode words and send them to the cells through a load operation with a cell as an address. Although the microcode words are difficult to program, either the assembly language can be modified or macros can be defined in C to adjust for frequently used cell commands.

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	(Write only)	:Loads the Accumulate In pipe with the data value specified
ACCOUT:	Accumulate Out	(Read Only)	: Reads the Accumulate Out value (ie,result) from the cell pipe
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 1)	-	(source 2)
MOVR:	(destination)	=	(source 1)		
ANDR:	(destination)	=	(source 1)	AND	(source 2)
OR:	(destination)	=	(source 1)	OR	(source 2)
NOP:					

Memory Operations:

LOAD:	(destination)	[source 1]	Loads destination register from address in source register
LOADD:	(destination)	16 bit value	Loads destination register with the 16 bit data value
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.
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

The Convolution Operation

In order for the convolution operation to be successful, the user must set up the system properly. The process is broken into three phases.

Phase 1: Initialize Cells

Each cell must be properly configured. This is done by clearing all of the cells, loading each cell with the specified amount of delay, and loading the input2 (i.e. h[n]) values to the RA registers in the cells. The procedure in assembly is described below.

```
CCLEAR                          ; Clears out all of the cell values
CSETDEL    CELL0    B#000       ; Set Delay in Cell 0 to 0
CSETDEL    CELL1    B#010       ; Set Delay in Cell 1 to 2
        [. . .]
CLRA
CLOAD      R5                   ; Load value at address R5 into
                                ;  the RA pipe
CLOAD      R6                   ; Load value at address R6 into
                                ;  the RA pipe
        [. . .]
```

Phase 2: Initialize Main Processor Registers

The main processor registers hold important information about the convolution operation and must be initialized. For example, a convolution could be set up using the following registers.

R0: Beginning address of array input1
R1: Beginning address of the destination array
R2: Length of the input1 (length(x))
R3: Length of the input2 - 1 (length(h) - 1)
R4: -1 (decrements length counters)
R5: +2 (increments array positions)

Phase 3: Performing the actual Convolution

Once the system has been initialized, all that is left to do is use a loop that loads data to cells, maintains array pointers, and writes the result to memory.

Writing an Assembly Convolution Program

The following program illustrates the general flow of programming elRoy to compute a convolution sum. The following information was used for this example.

x[n] = [1 2 3 4 5 6 7]

h[n] = [1 0 0 2 0 0 0 3 4]

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

starting address of x[n] = $1000
starting address of h[n] = $2000
starting address of y[n] = $3000

For this simple assembly program, the work of computing the delays and zeros is placed on the programmer. The programmer must install the delays into the cells by hand, and the h[n] array must be packed in memory, omitting the zero terms(i.e., $h_{mem}[n]$ = [1 2 3 4]).

For more advanced and flexible programs, the programmer should construct loops that count zeros in the h[n] array and place the delay into the cells automatically.

Example of Convolution Assembly Program

```
; Example Assembly Program : Compute a Convolution Sum
; Set up some array pointers
        LOADD     R6,       H#2000          ; Set h[n]'s beginning address
        LOADD     R7,       H#0002          ; Set R7 to +2
; Set up the delays in the cells
        CCLEAR                              ; Clears out all of the cell values
        CSETDEL   CELL1,    B#010           ; Set Delay in Cell 1 to 2
        CSETDEL   CELL2,    B#011           ; Set Delay in Cell 2 to 3
; Set up and fill the RA pipe
        CLRA                                ; Next cell data values fill RA pipe
        CLOAD     R6                        ; Load value at $2000 into RA pipe
        ADD       R6,R6,R7                  ; Increment h[n] pointer
        CLOAD     R6                        ; Load value at $2002 into RA pipe
        ADD       R6,R6,R7                  ; Increment h[n] pointer
        CLOAD     R6                        ; Load value at $2004 into RA pipe
        ADD       R6,R6,R7                  ; Increment h[n] pointer
        CLOAD     R6                        ; Load value at $2006 into RA pipe
; Set up registers
        LOADD     R0,       H#1000          ; Set x[n] position
        LOADD     R1,       H#3000          ; Set y[n] position
        LOADD     R2,       H#0007          ; Set length of x[n]
        LOADD     R3,       H#0008          ; Set length of h[n] - 1
        LOADD     R4,       H#8001          ; Set decrementer to -1
        LOADD     R5,       H#0002          ; Set incrementer to +2
; Start Convolution
CLOOP:
        CLOAD     R0                        ; Load next x[n] into pipe
        WRITE     R1,ACCOUT                 ; Write result to y[n]
        ADD       R0,R0,R5                  ; Increment x[n] pointer
        ADD       R1,R1,R5                  ; Increment y[n] pointer
        ADD       R2,R2,R4                  ; Decrement loop counter
        JA        CLOOP                     ; If Counter positive, keep looping
; Pass out values that are still in pipe now that x[n] is done
PLOOP:                                      ; Done with x[n], empty pipe
        CPASS                               ; Tell cells to pass out a result
        WRITE     R1,ACCOUT                 ; Write result to y[n]
        ADD       R1,R1,R5                  ; Increment y[n] pointer
        ADD       R3,R3,R4                  ; Decrement loop counter
        JA        PLOOP                     ; If Counter positive, keep looping
end
```

System Software

Since the heart of elRoy's purpose is to implement an algorithm for specialized mathematical functions, the system software is oriented for computations rather than complex user interaction. Since elRoy would ideally be implemented as a specialized co-processor unit to a full scale processor, the operating system components within this implementation have been minimized.

For this implementation, mathematical libraries are greatly beneficial to elRoy's function. The system can work with a wide array of computation algorithms. Through linear algebra, it is not difficult to use elRoy to implement Fourier transforms, derivatives and integrals of functions, scalings, rotations, projections, inverses, and other linear transforms. The C compiler serves as a means for building up a large resource of functions that transform the task of programming the processor to a job of working with mathematical tools.

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 a shift-add 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.

Tool Evaluation

The two main tools used by our group this quarter were TIM and Synopsys. With only minimal experience in TIM, learning how to use the tools was a difficult challenge.

TIM was used to generate the machine code run by elRoy. The usefulness of this tool was limited for several reasons. The primary reason is elRoy does not require complicated sequences of assembly instructions to perform his various tasks. Although a definition file was created and sample programs were assembled, when testing elRoy the instructions were invariably assembled by hand. Another area where TIM could use some refinement is the actual process itself (from start to finish). Atomization would greatly enhance this tools capabilities.

Synopsys was the tool used to generate the schematics for elRoy and to test the VHDL code generated. Although the schematic editor was found wanting in terms of visual appeal, it was a good tool overall. The interface was very intuitive (especially if you were familiar with how the schematics were transformed into VHDL code) and the hierarchical approach made for a very simple design approach. The learning curve for the tool was fast. Only a small amount of toying with the tutorial and productive work could begin.

The testing capabilities of Synopsys were also top-notch. The unix-like interface to the hierarchy made for easy switching between parts. The tracing and subsequent timing diagrams generated was all that was needed to test elRoy and his many parts. There are some complaints. For instance, the assignment of variables had to be in binary. This could get annoying when the user had to input two 16 bit inputs, a 16 bit instruction and a 32 bit accumulate-in to a cell on every 15th clock cycle.

Overall Synopsys was the best tool used by either of us in our previous projects (including Orcad, Autocad, Viewlogic and Logic Works). Newer versions of Synopsys should probably concentrate on improving the schematic editor as that is the only part found to be lacking.

Conclusions

The systolic array offers an impressive arsenal of computational power for mathematically oriented applications. While elRoy is a specific implementation limited by course equipment, it has successfully demonstrated the possibilities of such architectures. The heart of elRoy could easily be extracted and placed in a device oriented as a special purpose math co-processor. With the growing market for DSP chips, elRoy presents a competitive angle in computation power.

The actual hardware implementation of this project will dramatically change elRoy's identity. The unit was designed assuming a worst case hardware implementation phase, resulting in several gate level components. As documentation for the hardware becomes available, it is obvious that some key parts, such as the multiply unit, have already been built in optimized designs. To take advantage of these resources, elRoy will have to be redesigned. Although it is disheartening to have to change a project this far along, the actual workload is not expected to be outrageous. A large amount of the design has fallen to learning the tools, but revision time has dropped as our experience has grown.

It is difficult to say what would be done differently in this project if it had to be started again from day one. With a clearer list of tools available in the hardware design phase, construction could possibly have gone smoother with less concern over the design tools' synthesis process. We believe that day zero will come again in January when elRoy is adjusted to fit the new libraries.

Appendix A: TIM Assembly

SAT Nov 2, 1994

```
ITLE    ASSEMBLY LANGUAGE DEFINITION FILE FOR EXAMPLE COMPUTER DESIGN
ORD     32
IDTH    72
INES    50
*************************************************************************
REGISTER ASSIGNMENTS - REGISTER DIRECT ADDRESSING
*************************************************************************
0:          EQU     B#0000
1:          EQU     B#0001
2:          EQU     B#0010
3:          EQU     B#0011
4:          EQU     B#0100
5:          EQU     B#0101
6:          EQU     B#0110
7:          EQU     B#0111
*************************************************************************
 SPECIAL FUNCTION REGISTERS
*************************************************************************
CCIN:       EQU     B#1000          ; ACCUMULATE IN  - WRITE ONLY
CCOUT:      EQU     B#1001          ; ACCUMULATE OUT - READ ONLY
INST:       EQU     B#1010          ; CELL INSTRUCTION
A:          EQU     B#1011          ; DATA FOR RA
B:          EQU     B#1011          ; DATA FOR RB
ONVR:       EQU     B#1100          ; EXECUTE CONVOLUTION SEQUENCE
ULTR:       EQU     B#1101          ; EXECUTE MULTIPLICATION SEQUENCE
*************************************************************************
 CELL FUNCTION EQUATES
*************************************************************************
ELL0:        EQU     B#000000
ELL1:        EQU     B#000001
ELL2:        EQU     B#000010
ELL3:        EQU     B#000011
ELLP:        EQU     B#100000        ; PARALLEL LOAD ALL CELLS
*************************************************************************
 INSTRUCTION OPCODE LABELS - MUST BE 4-BITS
*************************************************************************
OR:         EQU     B#0000
XOR:        EQU     B#0001
ADD:        EQU     B#0010
SUB:        EQU     B#0011
AND:        EQU     B#0100
MOV:        EQU     B#0101
PUSH:       EQU     B#0110
POP:        EQU     B#0111
 JUMP INSTRUCTIONS
JMP:        EQU     B#1000
JA:         EQU     B#1001
JE:         EQU     B#1010
CMP:        EQU     B#1011
RTS:        EQU     B#1100
JSR:        EQU     B#1101
 LOAD INSTRUCTIONS
LOAD:       EQU     B#1110
WRITE:      EQU     B#1111
 SETUP
ULL:        EQU     B#0000                  ;4-BIT ZERO VALUE
PCODE:      SUB     4VLMOV                  ;4-BIT OPCODE FIELD
LANK16:     EQU     16H#0000                ;16-BIT FIELD
W:          DEF     16VH#0000,16VH#0000     ;32-BIT DATA DIRECTIVE
*************************************************************************
```

ASSEMBLY LANGUAGE INSTRUCTIONS

```
**************************************************************************
ORR:        DEF     LXOR,4VH#0,4VH#0,4VH#0,BLANK16
DDR:        DEF     LADD,4VH#0,4VH#0,4VH#0,BLANK16
UBR:        DEF     LSUB,4VH#0,4VH#0,4VH#0,BLANK16
OVR:        DEF     LMOV,4VH#0,4VH#0,NULL,BLANK16
NDR:        DEF     LAND,4VH#0,4VH#0,4VH#0,BLANK16
RR:         DEF     LOR,4VH#0,4VH#0,4VH#0,BLANK16
 MEMOPS
OAD:        DEF     LLOAD,4VH#0,4VH#0,NULL,BLANK16    ; LOAD TO(R) FROM[R]
RITE:       DEF     LWRITE,NULL,4VH#0,4VH#0,BLANK16   ; WRITE TO[R] FROM(R)
OADD:       DEF     LLOAD,4VH#0,NULL,H#F,16VH#0000    ; LOAD TO(R) FROM DATA
 BRANCHING
MP:         DEF     LCMP,NULL,4VH#0,4VH#0,BLANK16
E:          DEF     LJE,NULL,NULL,NULL,16VH#0000
A:          DEF     LJA,NULL,NULL,NULL,16VH#0000
SR:         DEF     LJSR,NULL,NULL,NULL,16VH#0000
ETURN:      DEF     LRTS,NULL,NULL,NULL,BLANK16
ONV:        DEF     LLOAD,CONVR,NULL,NULL,BLANK16     ; SET OFF CONV
MUL:        DEF     LLOAD,MULTR,NULL,NULL,BLANK16     ; SET OFF MULT
OP:         DEF     LOR,NULL,NULL,NULL,BLANK16        ; OR R0 WITH ITSELF
**************************************************************************
 CELL ASSEMBLY LANGUAGE INSTRUCTIONS
**************************************************************************
 CELL INSTRUCTION FORMAT (16 BITS):
       HIGH
          6 : CELL#        : 1 -- CELL PARALLEL LOAD
                           : 5 -- CELL ADDRESS
          1 : RA           : 1 -- LOAD EXTERNAL
          2 : RB           : 1 -- LOAD EXTERNAL
                           : 1 -- SET TO ZERO
          2 : ACCUMULATE   : 1 -- LOAD EXTERNAL
                           : 1 -- SET TO ZERO
          1 : ACC GATE     : 1 -- ADD MULTIPLY RESULT TO ACCUMULATE
          4 : DELAY        : 1 -- LOAD NEW DELAY VALUE
                           : 3 -- DELAY VALUE
        LOW

 SET DELAY IN CELL
                                                CELL#      LOAD       DELAY
SETDEL:     DEF     LLOAD,CINST,NULL,NULL,6VB#000000,B#0000001,3VB#000
 LOAD RA PIPE -- HOLDS ON TO PREVIOUS ACCUMULATE, CLEARS RB
                     LOAD  DEST                   CELL#
LRA:        DEF     LLOAD,CINST,NULL,NULL,CELLP,B#1010000000
 LOAD RB AND ACCUMULATE
                     LOAD  DEST                   CELL#
LAA:        DEF     LLOAD,CINST,NULL,NULL,6VB#000000,B#0100010000
 PASS OUT ACCUMULATES -- GIVES ANSWERS
                     LOAD  DEST                   CELL#
PASS:       DEF     LLOAD,CINST,NULL,NULL,CELLP,B#0011000000
 CELL CLEAR -- WIPES OUT DELAY AND ACCUMULATE, SETS RB TO ZERO
                     LOAD  DEST                   CELL#
CLEAR:      DEF     LLOAD,CINST,NULL,NULL,6VB#000000,B#0010101000
ND
```

```
Addr            Line TEST PROGRAM 1 FOR ELROY

                   1 TITLE  TEST PROGRAM 1 FOR ELROY
                   2 LIST F,W
                   3 LINES 50
                   4 ;TEST XORS
0000 10010000      5 XORR  R0,R0,R1
0001 11010000      6 XORR  R1,R0,R1
0002 E00FAAAA      7 LOADD R0,H#AAAA
0003 E10FFFFF      8 LOADD R1,H#FFFF
0004 E20F0002      9 LOADD R2,H#0002
0005 E30F0003     10 LOADD R3,H#0003
0006 14010000     11 XORR  R4,R0,R1
0007 15230000     12 XORR  R5,R2,R3
0008 56400000     13 MOVR  R6,R4
0009 47120000     14 ANDR  R7,R1,R2
000A 02120000     15 ORR   R2,R1,R2
000B E0200000     16 LOAD  R0,R2
000C F0120000     17 WRITE R1,R2
000D E70FAFAF     18 LOADD R7,H#AFAF
000E B0120000     19 CMP   R1,R2
000F A000F000     20 JE    H#F000
0010 B0340000     21 CMP   R3,R4
0011 9000FAAA     22 JA    H#FAAA
0012 D0003EEE     23 JSR   H#3EEE
0013 C0000000     24 RETURN
0014 EC000000     25 CONV
0015 ED000000     26 MMUL
0016 00000000     27 NOP
0017 EA00040A     28 CSETDEL CELL1,B#010
0018 EA008280     29 CLRA
0019 EA008110     30 CLAA CELLP
001A EA0080C0     31 CPASS
001B EA0004A8     32 CCLEAR CELL1
                  33
```

ACCIN	A 00000008	ACCOUT	A 00000009	ADDR	D	ANDR	D
CELL1	A 00000001	CELL2	A 00000002	CELL3	A 00000003	CELLP	A 00000020
CMP	D	CONV	D	CONVR	A 0000000C	CPASS	D
JE	D	JSR	D	LADD	A 00000002	LAND	A 00000004
LJMP	A 00000008	LJSR	A 0000000D	LLOAD	A 0000000E	LMOV	A 00000005
LPOP	A 00000007	LPUSH	A 00000006	LRTS	A 0000000C	LSUB	A 00000003
MOVR	D	MULTR	A 0000000D	NARG	A 00000000	NOP	D
R0	A 00000000	R1	A 00000001	R2	A 00000002	R3	A 00000003
R7	A 00000007	RA	A 0000000B	RB	A 0000000B	RETURN	D

ssembly Phase complete.
0 error(s) detected.

Appendix B: Test Programs and Timing Diagrams

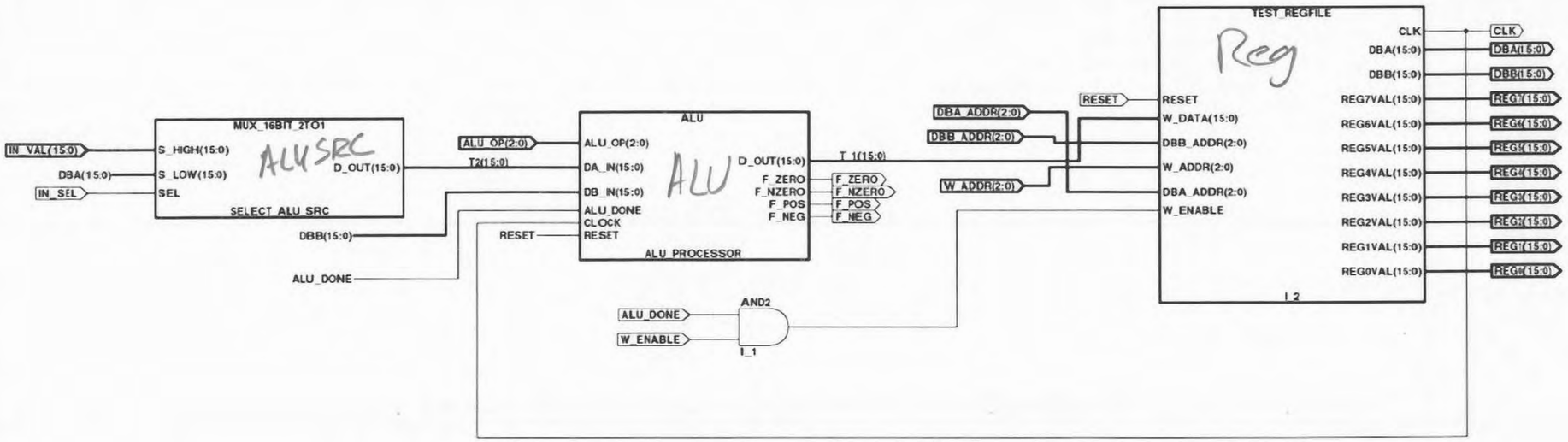
```
cd /e/uut
trace alu_done
trace alu_op
trace dba_addr
trace dbb_addr
trace in_sel
trace in_val
trace reset
trace w_addr
trace w_enable
trace clk
trace dba
trace dbb
trace f_neg
trace f_pos
trace f_zero
trace f_nzero
trace reg7
trace reg6
trace reg5
trace reg4
trace reg3
trace reg2
trace reg1
trace reg0

cd /e/uut
assign ('1') reset
assign ("000") alu_op
assign ('0') alu_done
assign ("000") dba_addr
assign ("000") dbb_addr
assign ('1') in_sel
assign ("1111111111111111") in_val
assign ("000") w_addr
assign ('0') w_enable

run 25

cd /e/uut
assign ('0') reset
assign ("101") alu_op
assign ('1') alu_done
assign ("000") dba_addr
assign ("000") dbb_addr
assign ('1') in_sel
assign ("1000000000000000") in_val
assign ("000") w_addr
assign ('1') w_enable
run 20

cd /e/uut
assign ("1000000000000001") in_val
assign ("001") w_addr
assign ('1') w_enable
run 20

cd /e/uut
assign ("1000000000000010") in_val
assign ("010") w_addr
assign ('1') w_enable
run 20

cd /e/uut
assign ("1000000000000011") in_val
assign ("011") w_addr
assign ('1') w_enable
run 20

cd /e/uut
assign ("1000000000000100") in_val
assign ("100") w_addr
assign ('1') w_enable
run 20

cd /e/uut
assign ("1000000000000101") in_val
assign ("101") w_addr
assign ('1') w_enable
run 20

cd /e/uut
assign ("1000000000000110") in_val
assign ("110") w_addr
assign ('1') w_enable
run 20

cd /e/uut
assign ("1000000000000111") in_val
assign ("111") w_addr
assign ('1') w_enable
run 20

cd /e/uut
```

```
assign ("101") alu_op
assign ('1') alu_done
assign ("000") dba_addr
assign ("000") dbb_addr
assign ('1') in_sel
assign ("1111111111111111") in_val
assign ("000") w_addr
assign ('0') w_enable
run 20

cd /e/uut
assign ("010") alu_op
assign ('0') alu_done
assign ("110") dba_addr
assign ("111") dbb_addr
assign ('0') in_sel
assign ("1111111111111111") in_val
assign ("000") w_addr
assign ('0') w_enable
run 60

cd /e/uut
assign ('1') alu_done
assign ('1') w_enable
run 20

cd /e/uut
assign ('0') alu_done
assign ('0') w_enable
assign ("000") dba_addr
assign ("111") dbb_addr
run 60

cd /e/uut
assign ("011") alu_op
assign ('0') alu_done
assign ("111") dba_addr
assign ("110") dbb_addr
assign ('0') in_sel
assign ("1111111111111111") in_val
assign ("001") w_addr
assign ('0') w_enable
run 20

cd /e/uut
assign ('1') alu_done
assign ('1') w_enable
run 20
```

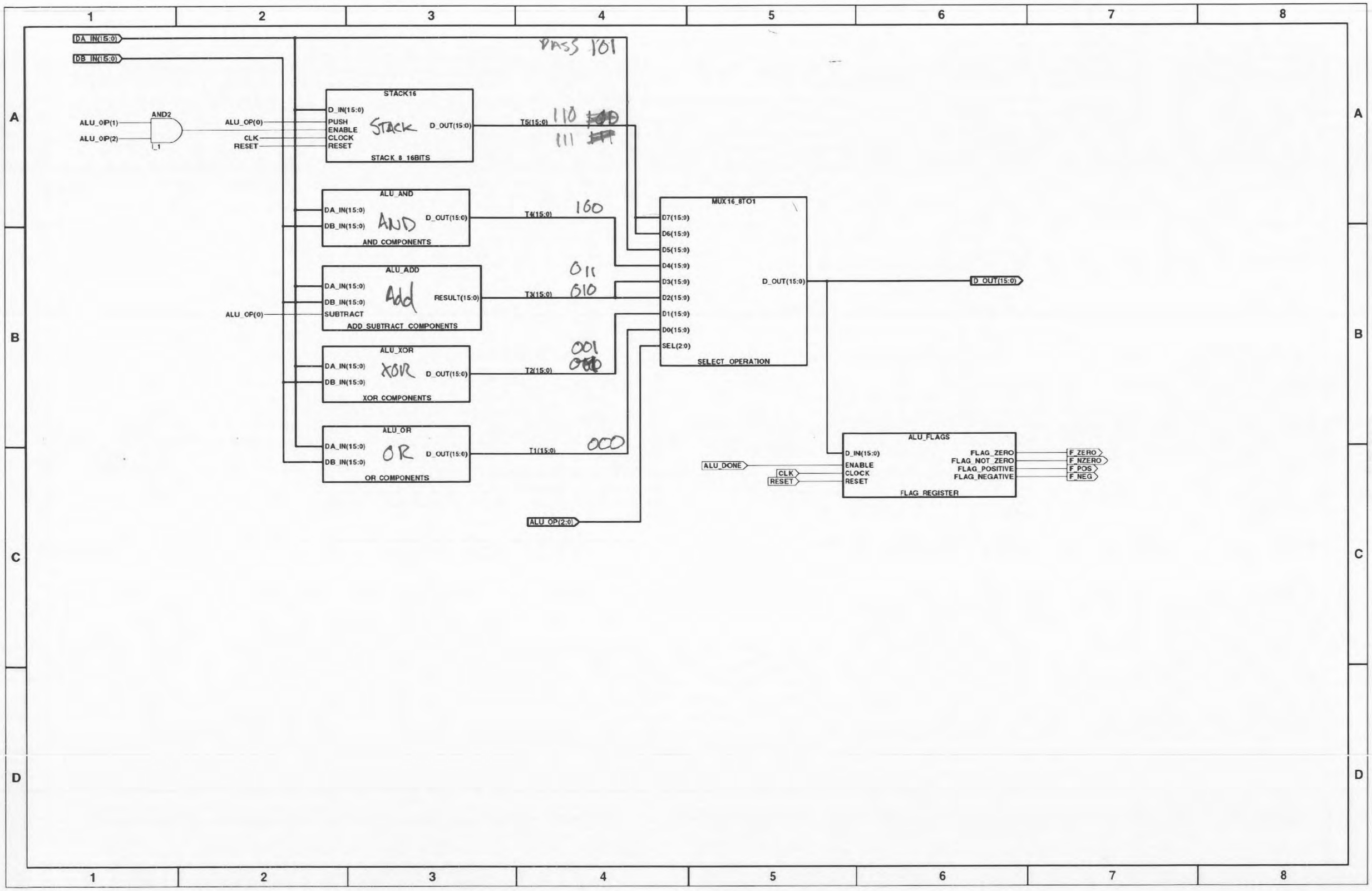
```
cd /e/uut
trace dba_addr
trace dbb_addr
trace dba
trace dbb
trace reset
trace w_addr
trace w_data
trace w_enable
trace clk
trace reg0val
trace reg1val
trace reg2val
trace reg3val
trace reg4val
trace reg5val
trace reg6val
trace reg7val

cd /e/uut
assign ("000") dba_addr
assign ("000") dbb_addr
assign ('0') w_enable
assign ('1') reset
assign ("000") w_addr
assign ("1000000000000000") w_data
run 25

cd /e/uut
assign ('0') reset
run 20

cd /e/uut
assign ("111") w_addr
assign ('1') w_enable
assign ("1000000000000111") w_data
run 20


cd /e/uut
assign ('0') w_enable
assign ("1110000000000110") w_data
run 20


cd /e/uut
assign ('1') w_enable
assign ("110") w_addr
assign ("1000000000000110") w_data
run 20

cd /e/uut
assign ('1') w_enable
assign ("101") w_addr
assign ("1000000000000101") w_data
run 20

cd /e/uut
assign ('0') w_enable
assign ("101") w_addr
assign ("1110000000000101") w_data
run 20

cd /e/uut
assign ('1') w_enable
assign ("101") w_addr
assign ("1000000000000101") w_data
run 20

cd /e/uut
assign ('1') w_enable
assign ("100") w_addr
assign ("1000000000000100") w_data
run 20

cd /e/uut
assign ('1') w_enable
assign ("011") w_addr
assign ("1000000000000011") w_data
run 20

cd /e/uut
assign ('1') w_enable
assign ("010") w_addr
assign ("1000000000000010") w_data
run 20

cd /e/uut
assign ('1') w_enable
assign ("001") w_addr
assign ("1000000000000001") w_data
run 20

cd /e/uut
```

```
assign ('1') w_enable
assign ("000") w_addr
assign ("1000000000000000") w_data
run 20

cd /e/uut
assign ('0') w_enable
assign ("110") w_addr
assign ("1111000000000110") w_data
run 20

cd /e/uut
assign ("110") dba_addr
assign ("001") dbb_addr
assign ("1111000000000110") w_data
run 20

cd /e/uut
assign ("010") dba_addr
assign ("100") dbb_addr
assign ("1111000000000110") w_data
run 20
```

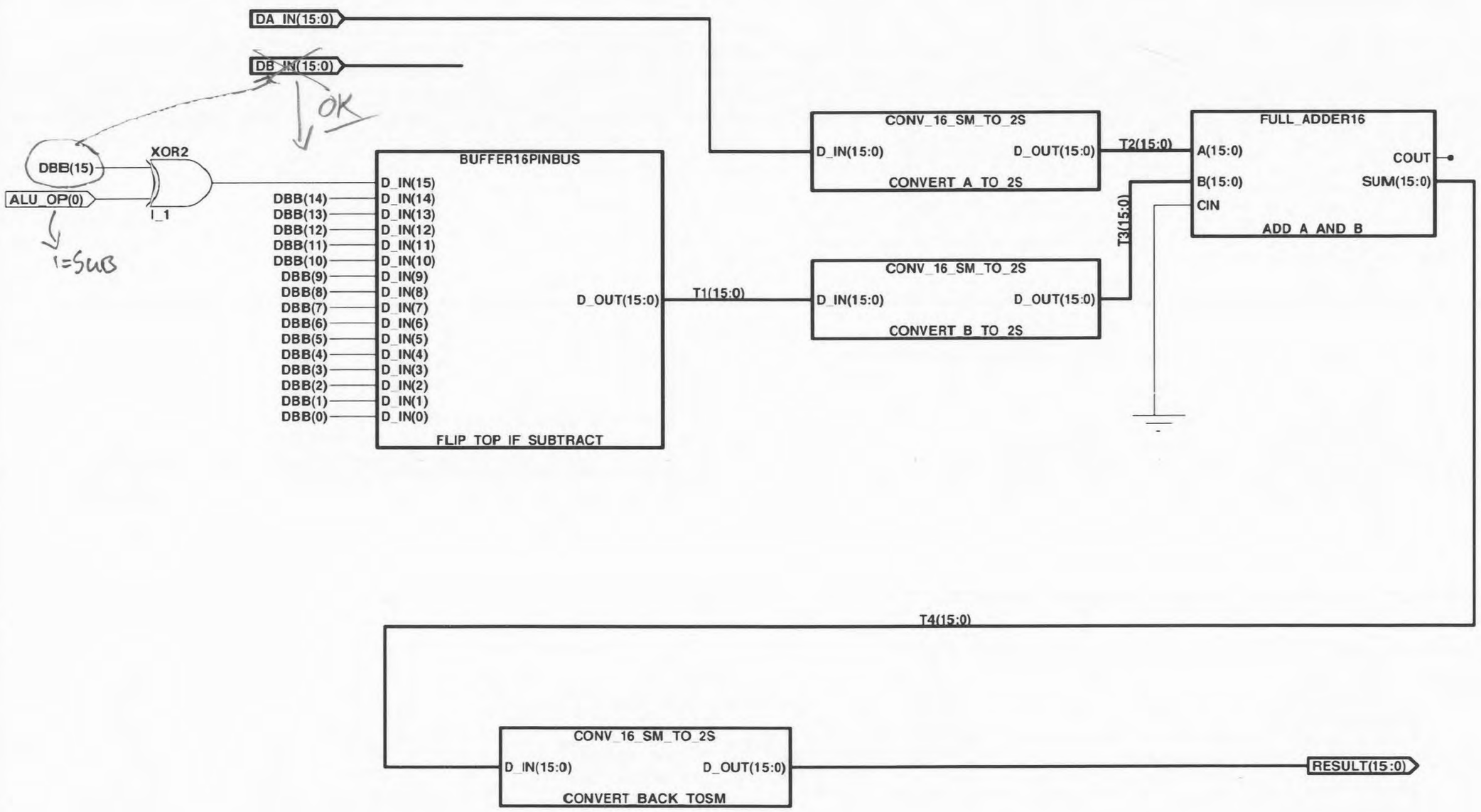
ALU + Reg Test File 11:20 pm Jun 17



ALU → JAN 27, 11:50pm



Add/SUB operation

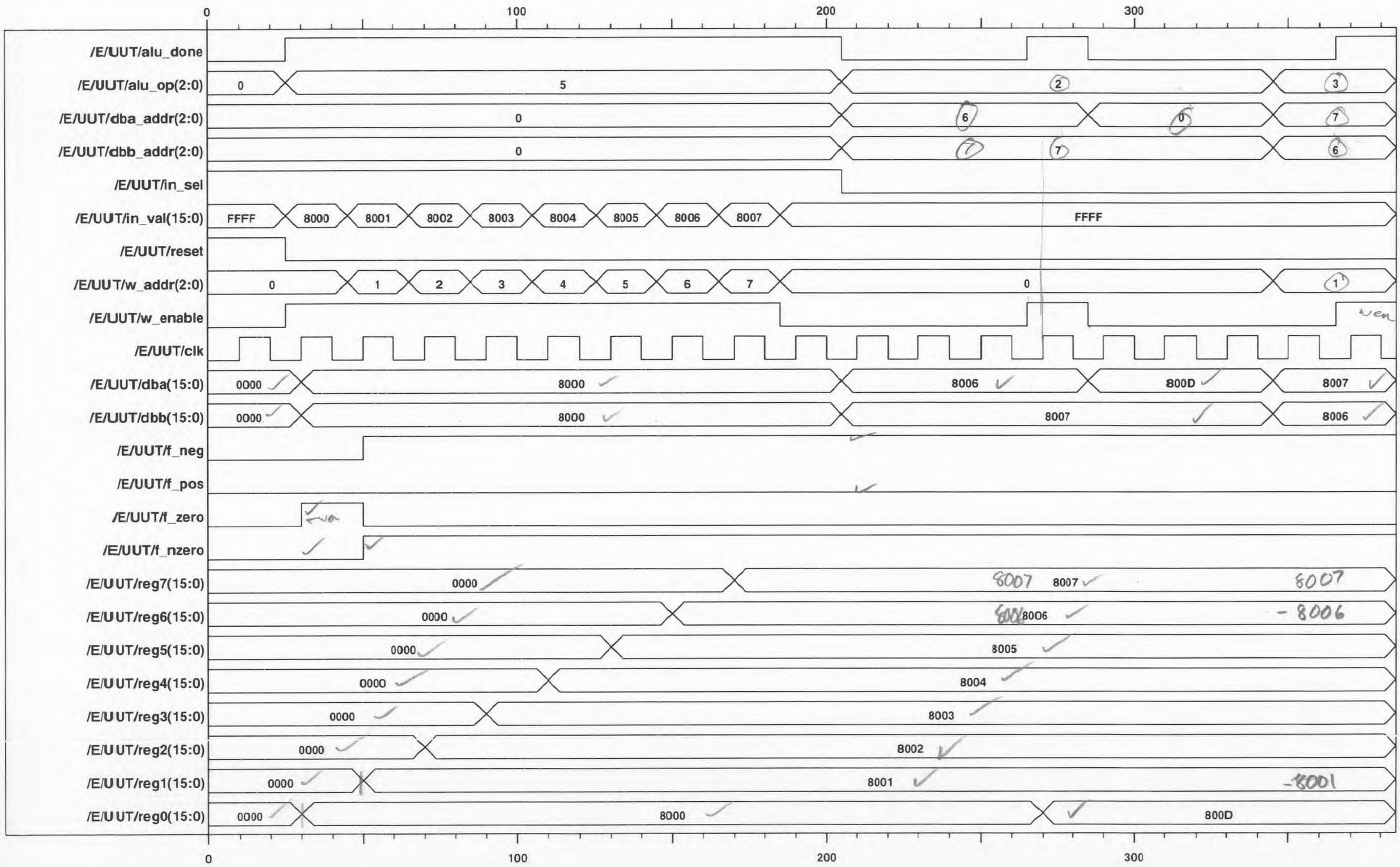


```
cd /e/uut
trace da_in
trace db_in
trace subtract
trace result

assign ("0000O000000001010") da_in
assign ("0000O000000000111") db_in
assign ('0') subtract
```

(1)

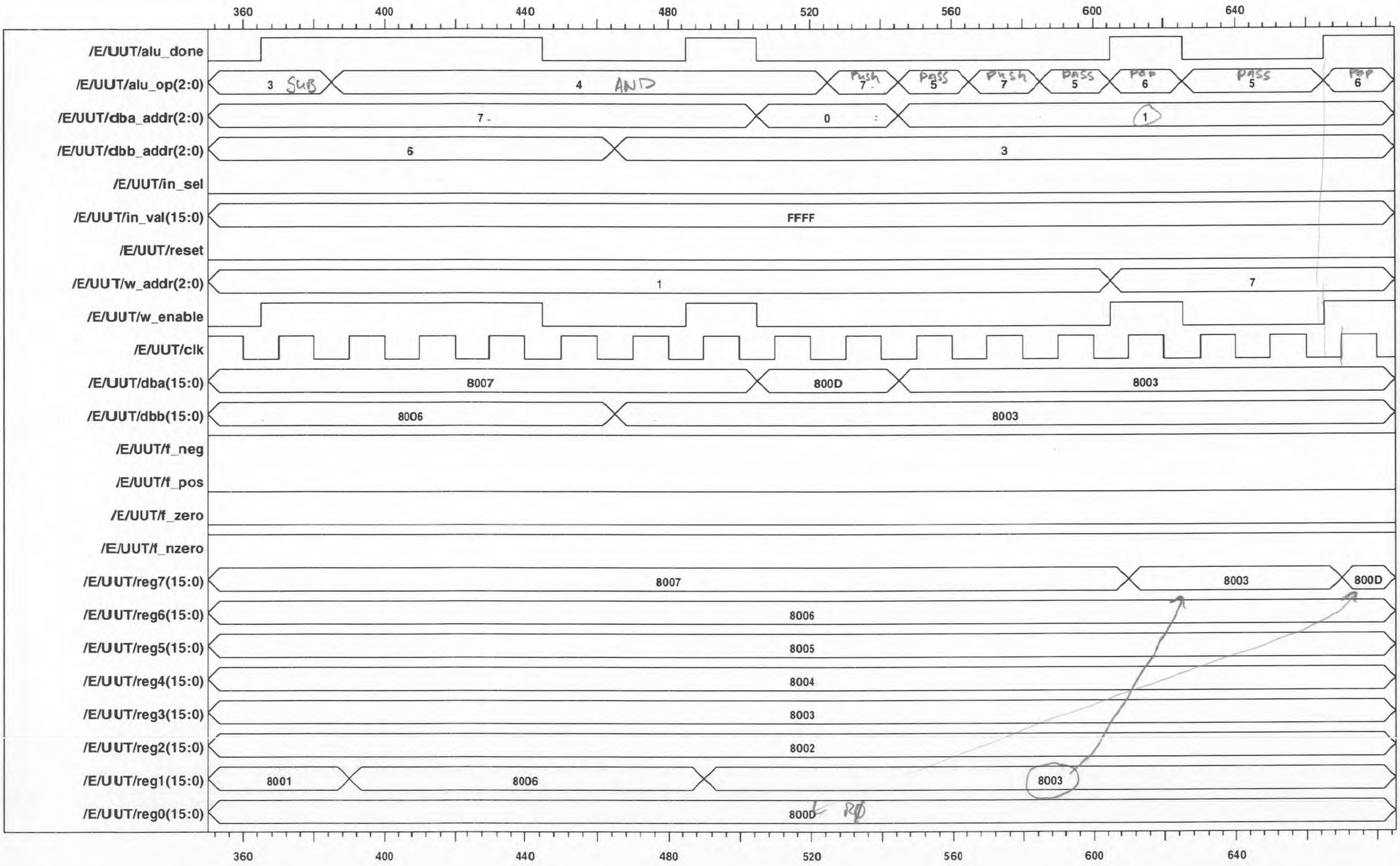
Appears to generally work

XOR →
OR →
PASS →



Test for Overall ALU OPS
(1/2)

(cont)~ Push/Pop → Appear to work fine - pushed 2 values, poped off 2, got right back

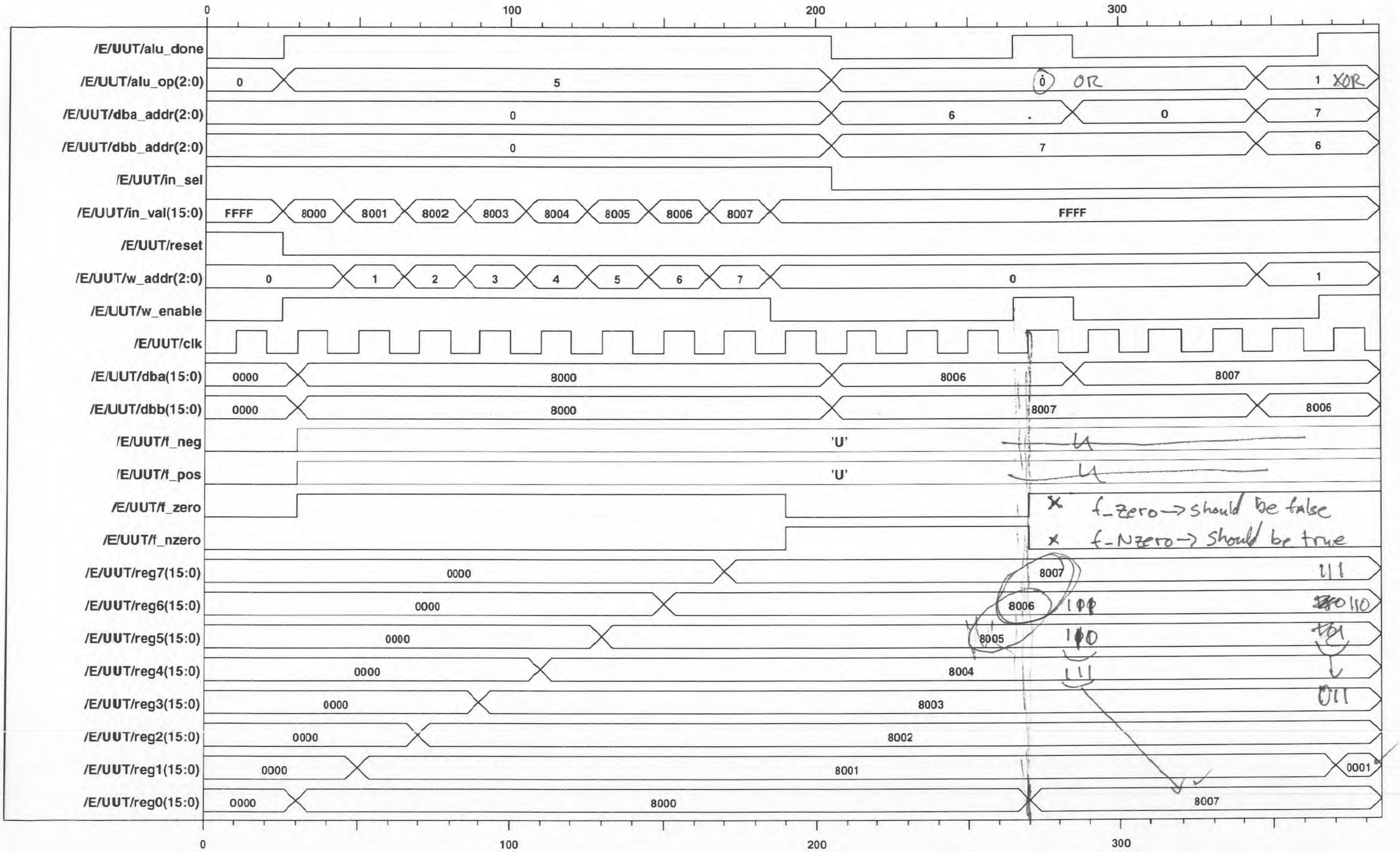


Test for Overall ALU ops
(2/2)

ALU OR, XOR ⇒ first glance - OK

Test 1

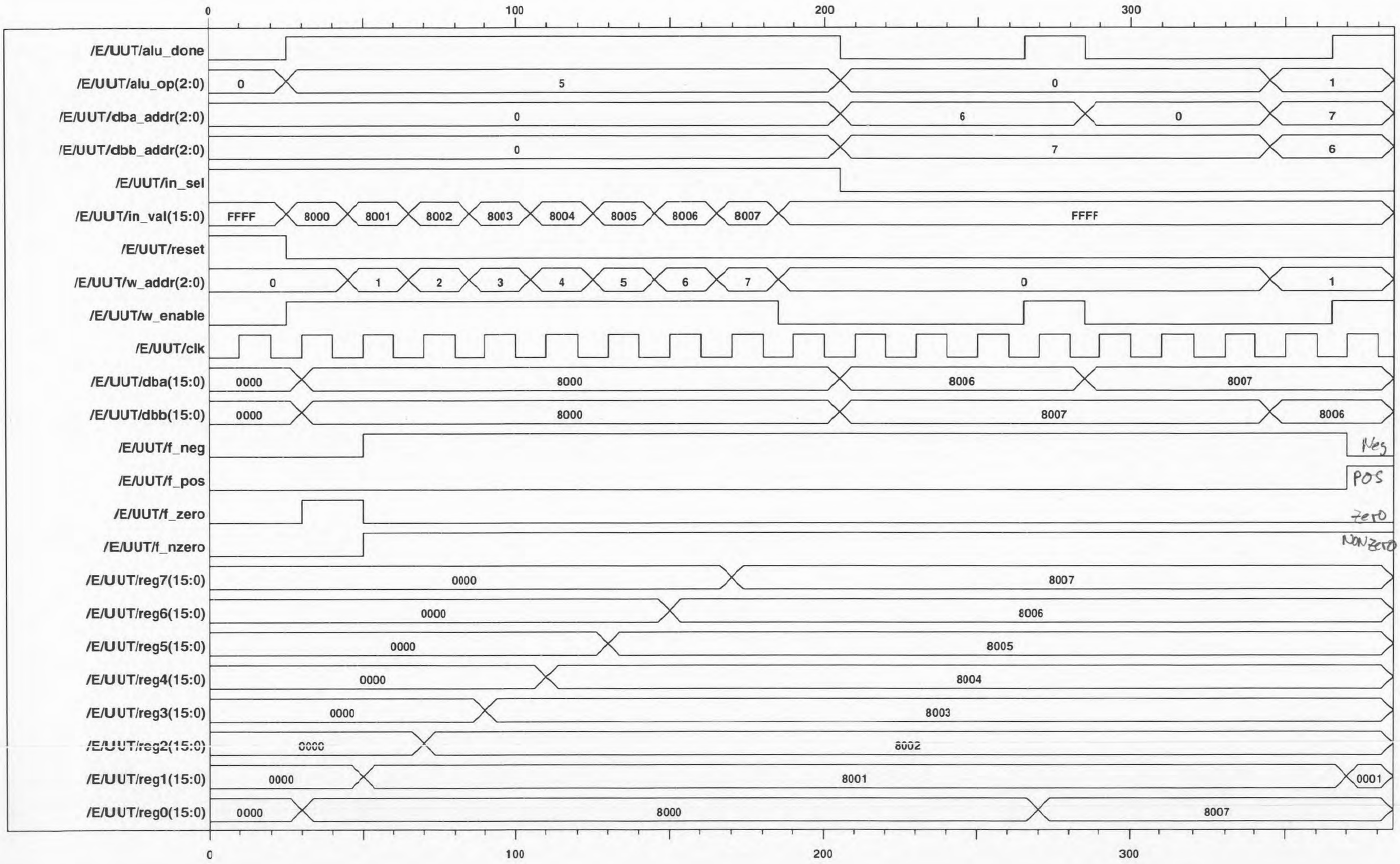
Sign Register is wrong



XOR/OR Tests (Bad flags)

(1/1)

ALU OR, XOR → fixed FLAGS Registers

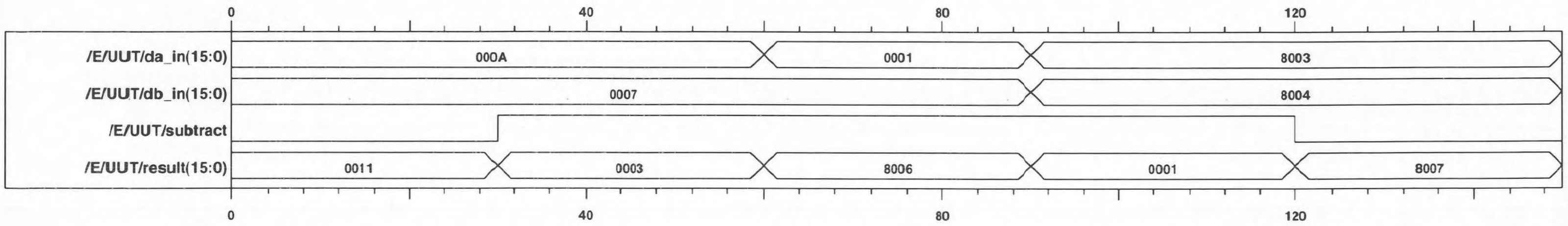


XOR/OR Tests – FLAGS Fixed

(1/1)

ALU_add_SUBTRACT Unit

looks like it works



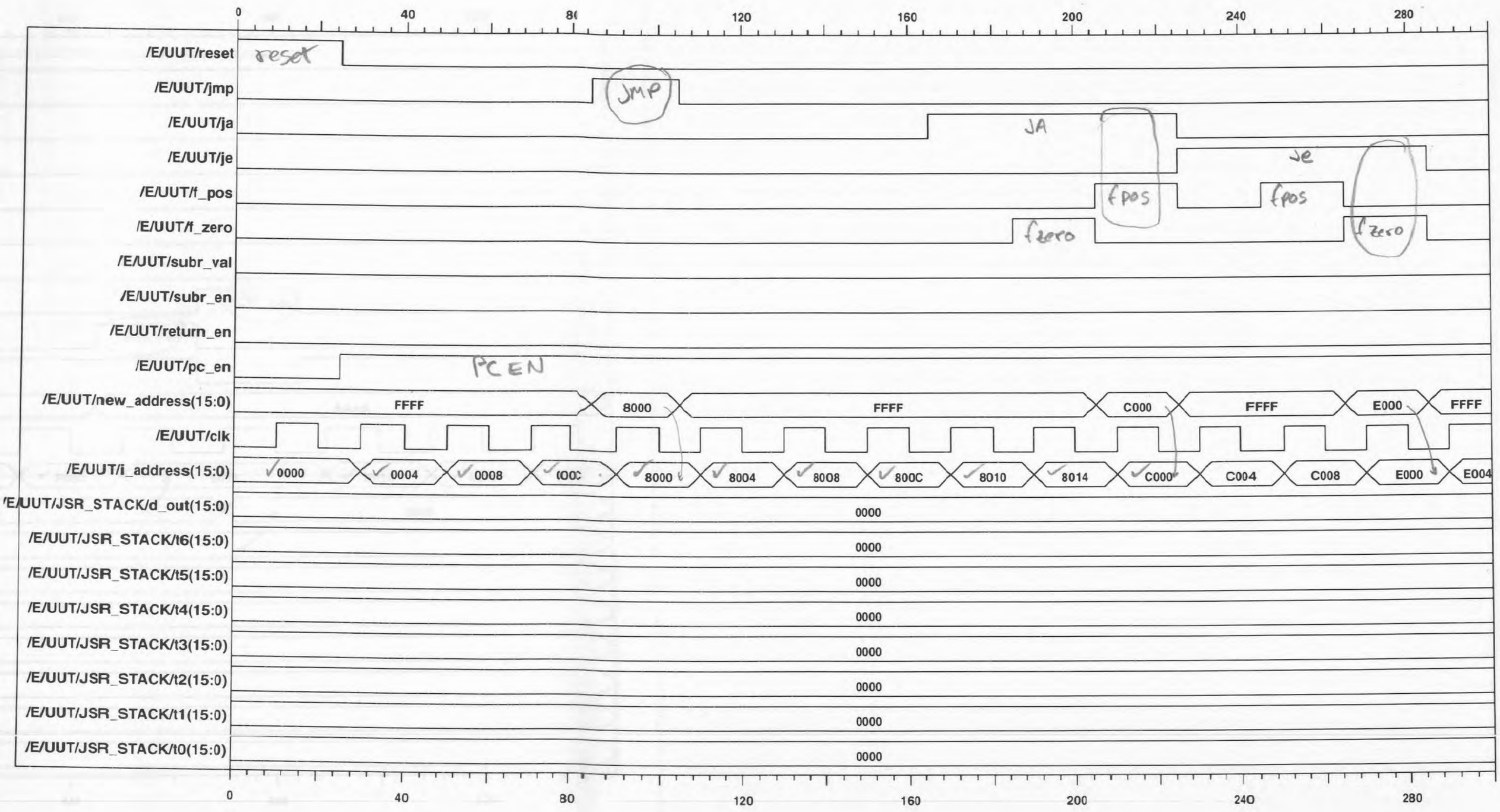
ALU Add/SUB unit

```
cd /e/uut
trace reset
trace jmp
trace ja
trace je
trace f_pos
trace f_zero
trace subr_val
trace subr_en
trace return_en
trace pc_en
trace new_address
trace clk
trace i_address

trace /E/UUT/JSR_STACK/d_out
trace /E/UUT/JSR_STACK/t6
trace /E/UUT/JSR_STACK/t5
trace /E/UUT/JSR_STACK/t4
trace /E/UUT/JSR_STACK/t3
trace /E/UUT/JSR_STACK/t2
trace /E/UUT/JSR_STACK/t1
trace /E/UUT/JSR_STACK/t0

assign ('1') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('0') pc_en
assign ("1111111111111111") new_address

run 25

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 60

assign ('0') reset
assign ('1') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1000000000000000") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 60

assign ('0') reset
assign ('0') jmp
assign ('1') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 20
```

```
assign ('0') reset
assign ('0') jmp
assign ('1') ja
assign ('0') f_pos
assign ('0') je
assign ('1') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('1') ja
assign ('1') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1100000000000000") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('1') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 20


assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('1') f_pos
assign ('1') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('1') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('1') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1110000000000000") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('1') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 60

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('1') subr_val
assign ('1') subr_en
assign ('0') return_en
```

```
assign ('0') pc_en
assign ("1111000000000000") new_address

run 20

assign ('0') reset
assign ('1') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111000000000000") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1111111111111111") new_address

run 60

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('1') return_en
assign ('1') pc_en
assign ("1010101010101010") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('1') subr_en
assign ('0') return_en
assign ('0') pc_en
assign ("1010101010101010") new_address

run 20

assign ('0') reset
assign ('0') jmp
assign ('0') ja
assign ('0') f_pos
assign ('0') je
assign ('0') f_zero
assign ('0') subr_val
assign ('0') subr_en
assign ('0') return_en
assign ('1') pc_en
assign ("1010101010101010") new_address

run 60
```

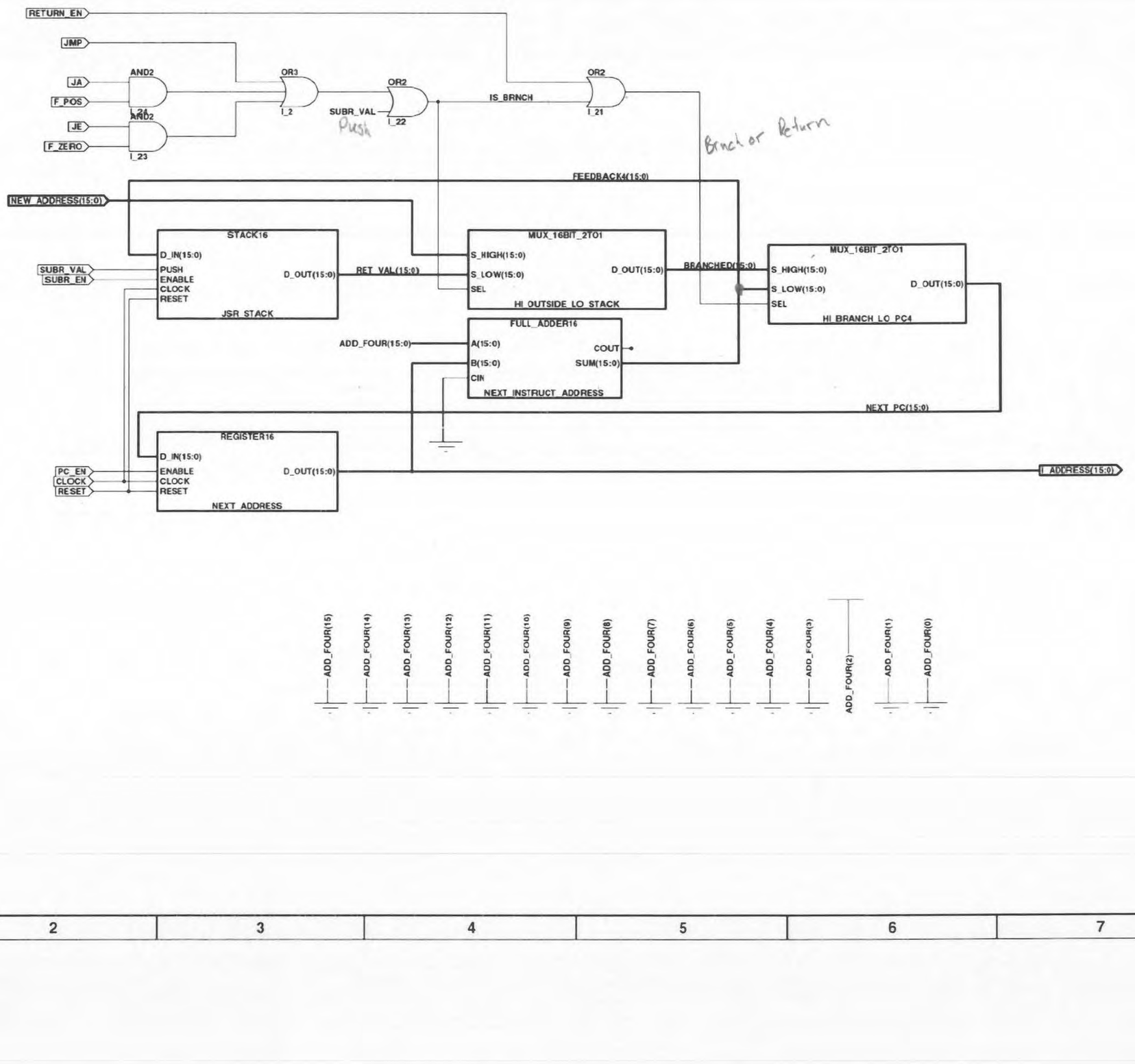
fixed problem with +4. JSR's, ~~RET~~; JE, Ret Correctly



PC_unit

(1/2)

Sheet 1 of 1


RETURN_EN
JMP
JA
F_POS
JE
F_ZERO
AND2
I_24
AND2
I_23
OR3
I_2
OR2
SUBR_VAL
I_22
IS_BRNCH
OR2
I_21
Push
Brnch or Return
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
FULL_ADDER16
ADD_FOUR(15:0)
A(15:0)
B(15:0)
CIN
COUT
SUM(15:0)
NEXT_INSTRUCT_ADDRESS
NEXT_PC(15:0)
REGISTER16
D_IN(15:0)
PC_EN
CLOCK
RESET
ENABLE
CLOCK
RESET
D_OUT(15:0)
NEXT_ADDRESS
I_ADDRESS(15:0)
ADD_FOUR(15)
ADD_FOUR(14)
ADD_FOUR(13)
ADD_FOUR(12)
ADD_FOUR(11)
ADD_FOUR(10)
ADD_FOUR(9)
ADD_FOUR(8)
ADD_FOUR(7)
ADD_FOUR(6)
ADD_FOUR(5)
ADD_FOUR(4)
ADD_FOUR(3)
ADD_FOUR(2)
ADD_FOUR(1)
ADD_FOUR(0)

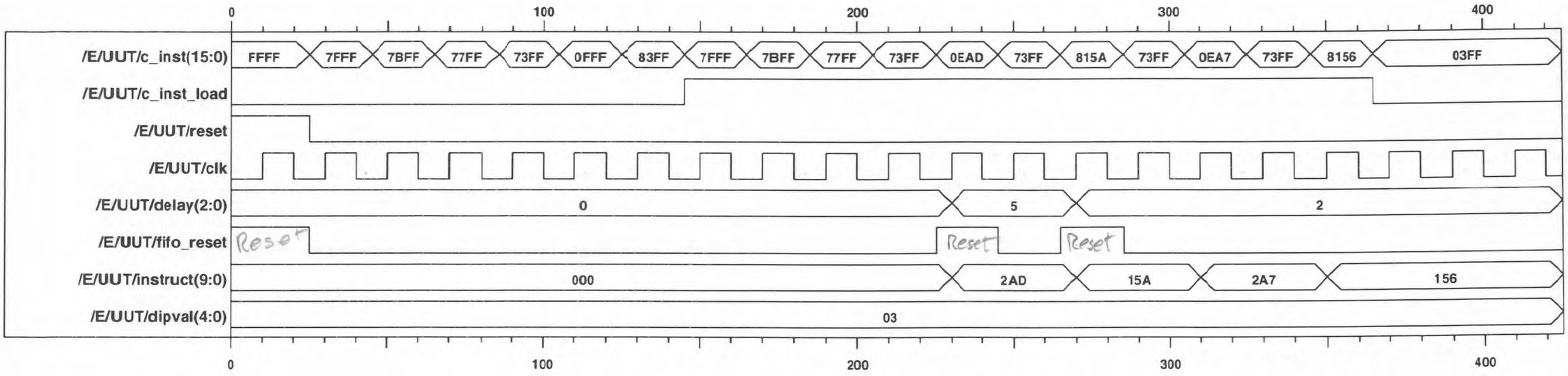
```
cd /e/uut
trace c_inst
trace c_inst_load
trace reset
trace clk
trace delay
trace fifo_reset
trace instruct
trace dipval

assign ('1') reset
assign ("1111111111111111") c_inst
assign ('0') c_inst_load

run 25

assign ('0') reset
assign ("0111111111111111") c_inst
run 20

assign ("0111101111111111") c_inst
run 20

assign ("0111011111111111") c_inst
run 20

assign ("0111001111111111") c_inst
run 20

assign ("0000111111111111") c_inst
run 20

assign ("1000001111111111") c_inst
run 20

echo try with bad address and a c_load

assign ('1') c_inst_load
assign ("0111111111111111") c_inst
run 20

assign ('1') c_inst_load
assign ("0111101111111111") c_inst
run 20

assign ('1') c_inst_load
assign ("0111011111111111") c_inst
run 20

assign ('1') c_inst_load
assign ("0111001111111111") c_inst
run 20

assign ('1') c_inst_load
assign ("0000111010101101") c_inst
run 20

echo bad address and cload

assign ('1') c_inst_load
assign ("0111001111111111") c_inst
run 20

echo parallel address and cload

assign ('1') c_inst_load
assign ("1000000101011010") c_inst
run 20

echo bad address and cload

assign ('1') c_inst_load
assign ("0111001111111111") c_inst
run 20

echo good address, cload, no delay load

assign ('1') c_inst_load
assign ("0000111010100111") c_inst
run 20

echo bad address and cload

assign ('1') c_inst_load
assign ("0111001111111111") c_inst
run 20

echo parallel address and cload

assign ('1') c_inst_load
assign ("1000000101010110") c_inst
run 20
```

```
assign ('0') c_inst_load
assign ("0000001111111111") c_inst
run 60
```

Cell Address Decode and Hold (v.2)

Appears to work fine. Added fifo Reset.

See next page for validation on design before Reset Fifo added



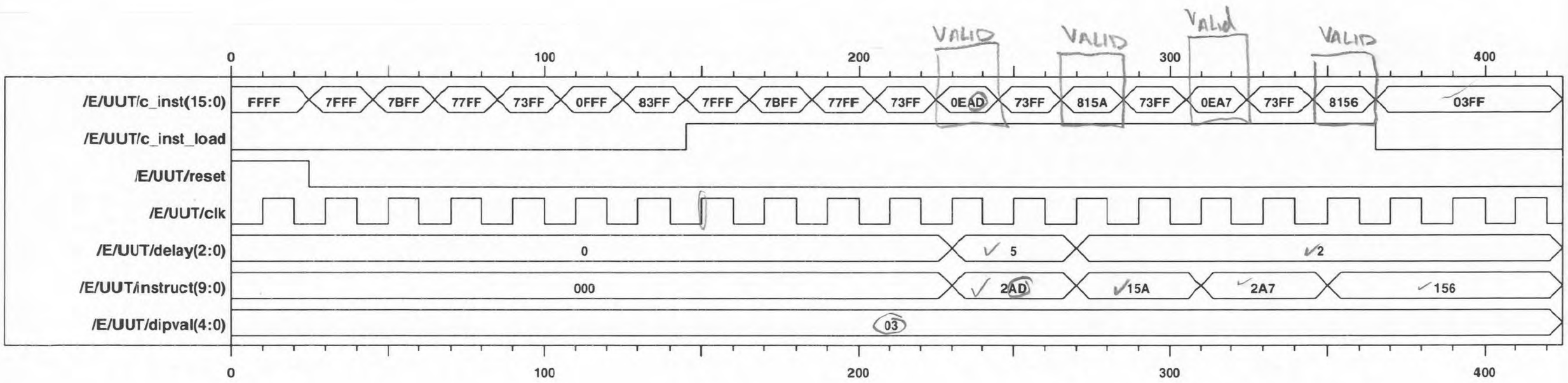
Cell Address Decode + Hold (V.1)

OLD - this has been updated on previous page

The cell address decode appears to work fine.

However → if we change the delay, we need to flush the fifo

Add a delay set pin to out of Add dec, or it to Reset of fifo.



0EA7

0000 | 1110 | 1010 | 0111 → no load

8156 → 0110 → no load

03FF → 000000|11 → no load

VALID IDS

000011

1 xxxxx

815A

1000 0001 0101 1010

100000 | 010101 | 1010

1 5 A

Delay=2

0EAD

0000 1110 (10) 13 = 1010

0 | 00011 | 101010 | 1101

2 A D

8421

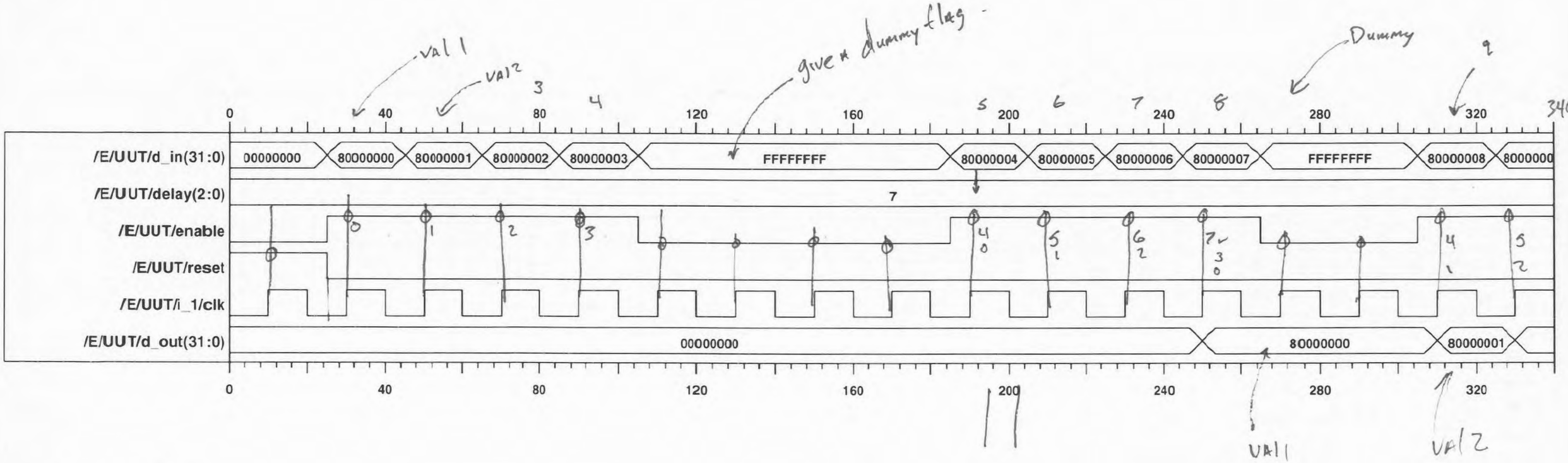
1|101 ⇒ 5 delay

```
cd /test_stack
trace d_in
trace push
trace enable
trace reset
trace clk
trace d_out
trace t6
trace t5
trace t4
trace t3
trace t2
trace t1
trace t0

assign ("1111000011110000") d_in
assign ('1') push
assign ('0') enable
assign ('1') reset
run 25

assign ('0') reset
assign ('1') enable
assign ("1000000000000001") d_in
run 20

assign ("1000000000000010") d_in
run 20

assign ('0') enable
run 20

assign ('1') enable
assign ("1000000000000011") d_in
run 20

assign ('0') push
assign ("iiiiiiiiiiiiiiii") d_in
run 20

assign ('1') push
assign ("1000000000000100") d_in
run 20

assign ("1000000000000101") d_in
run 20

assign ("1000000000000110") d_in
run 20

assign ("1000000000000111") d_in
run 20

assign ("1000000000001000") d_in
run 20

assign ("1000000000001001") d_in
run 20

assign ('0') push
run 180
```

Queue diagram

Works fine <u>if</u> enable comes when <u>clock = 0</u>

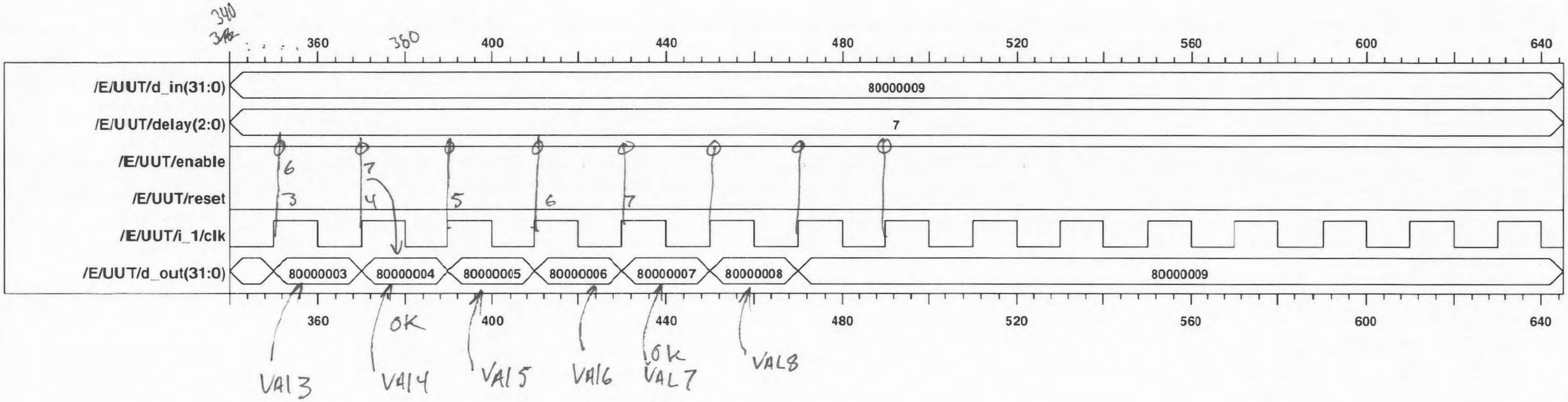
otherwise added edges invalid data

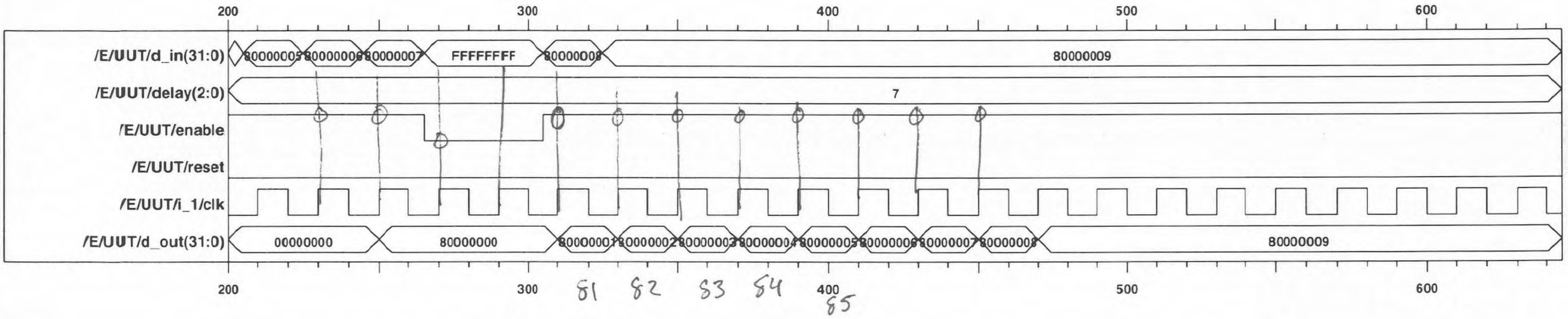


Fifo-8 timing diagrams

Nov 21 10:30pm

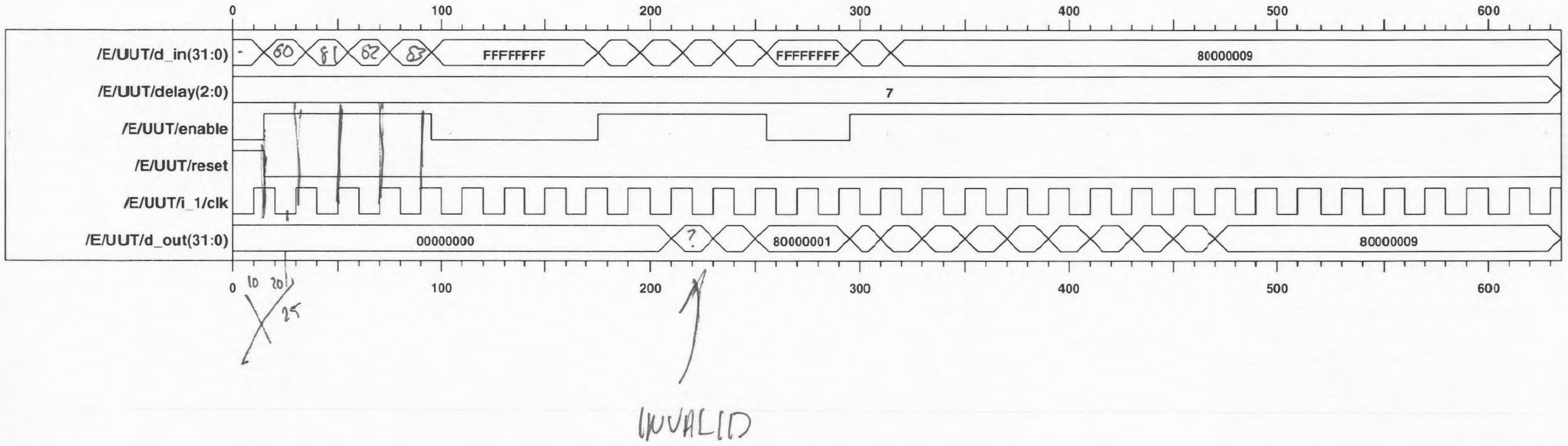
Result: enable allowed ONLY when clock=0


340
360
400
440
480
520
560
600
640
/E/UUT/d_in(31:0)
80000009
/E/UUT/delay(2:0)
7
/E/UUT/enable
/E/UUT/reset
/E/UUT/i_1/clk
/E/UUT/d_out(31:0)
80000003
80000004
80000005
80000006
80000007
80000008
80000009
6
7
3
4
5
6
7
OK
OK
VAL3
VAL4
VAL5
VAL6
VAL7
VAL8


200
300
400
500
600
/E/UUT/d_in(31:0)
80000005
80000006
80000007
FFFFFFFF
80000008
80000009
/E/UUT/delay(2:0)
7
/E/UUT/enable
/E/UUT/reset
/E/UUT/i_1/clk
/E/UUT/d_out(31:0)
00000000
80000000
80000001
80000002
80000003
80000004
80000005
80000006
80000007
80000008
80000009
200
300
400
500
600
S1
S2
S3
S4
S5

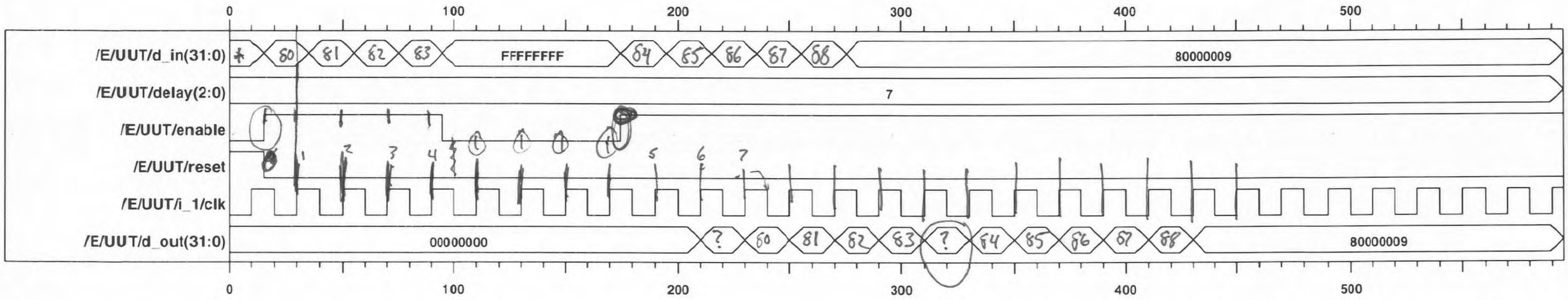
for BAD

When enABLE comes when clock = 1

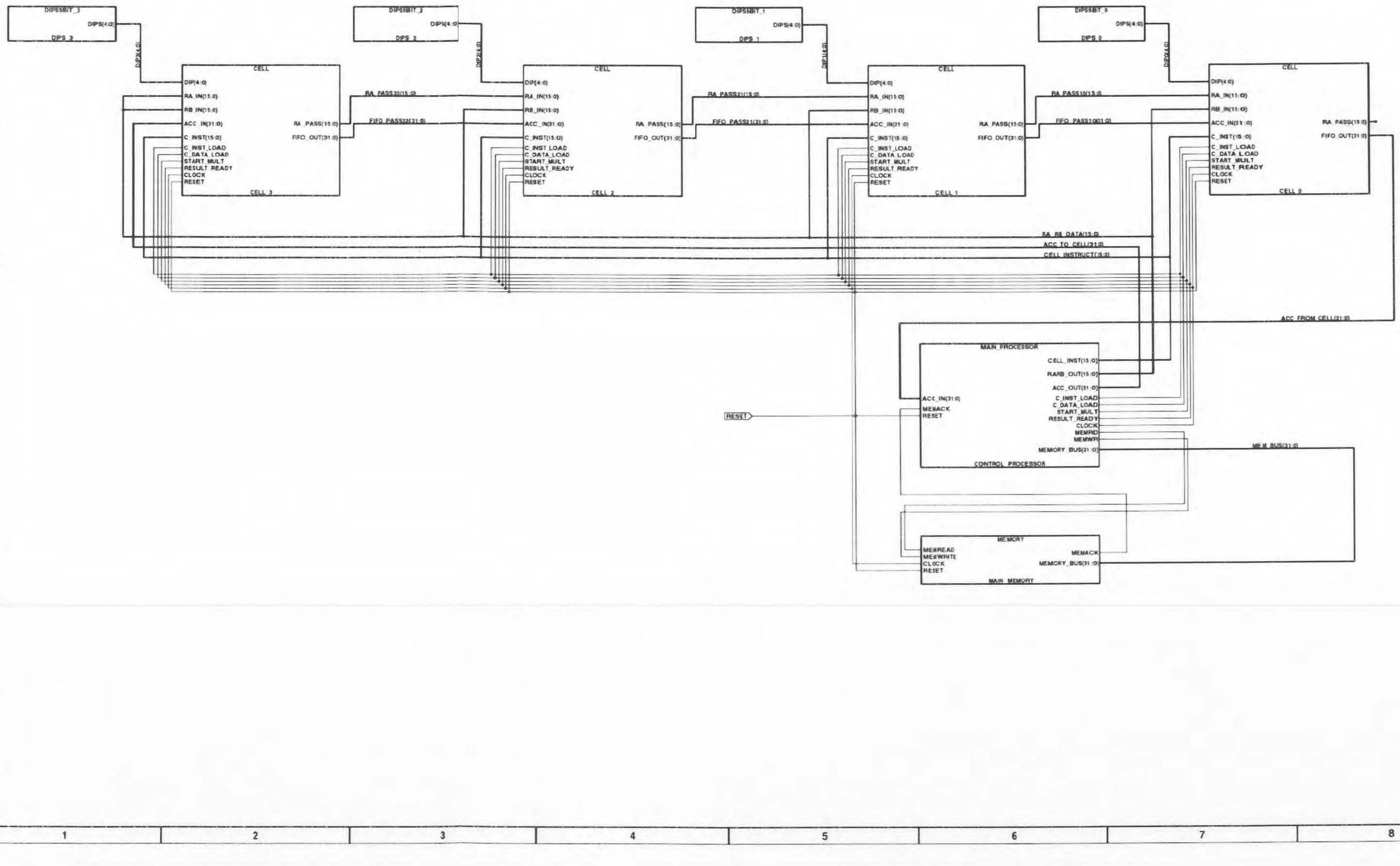


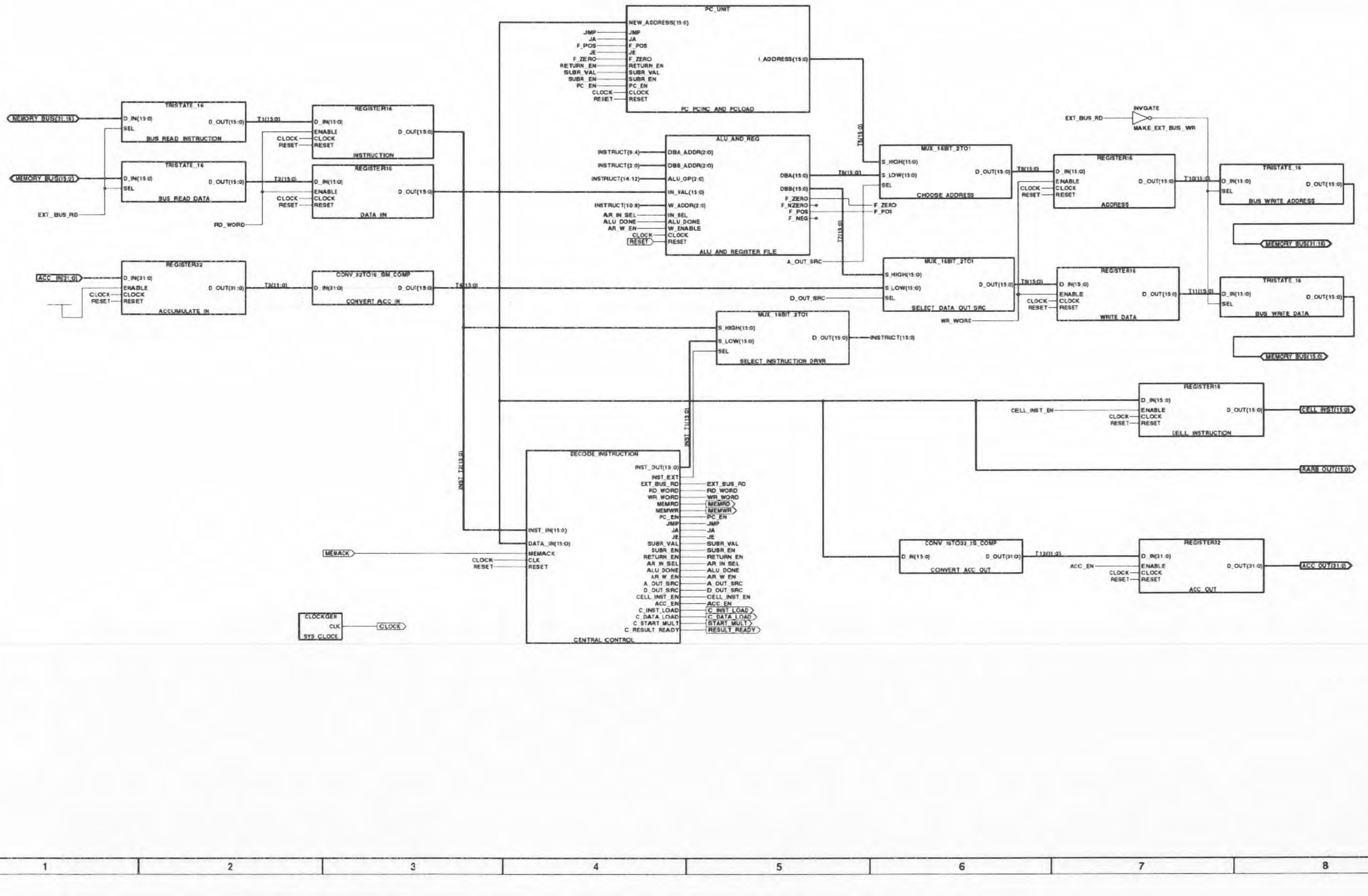
INVALID

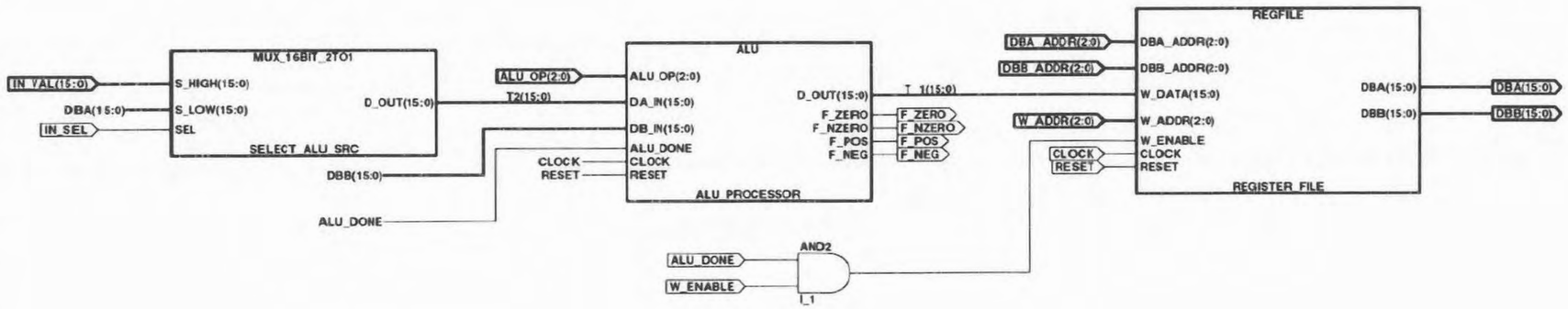
MORE BAD

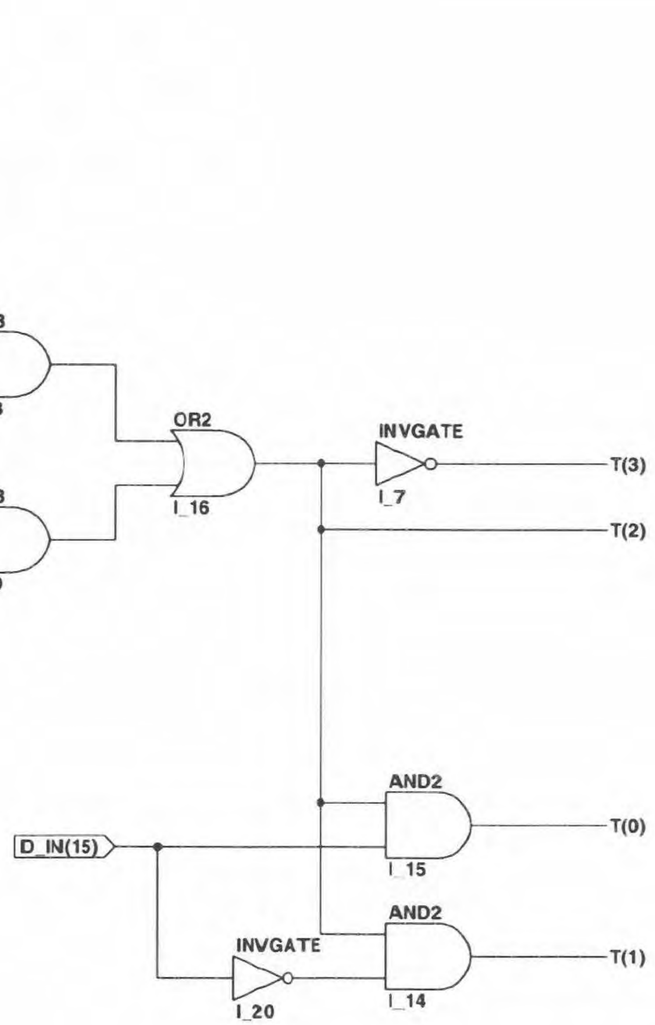

0
100
200
300
400
500
/E/UUT/d_in(31:0)
80
81
82
83
FFFFFFFF
84
85
86
87
88
80000009
/E/UUT/delay(2:0)
7
/E/UUT/enable
/E/UUT/reset
/E/UUT/i_1/clk
/E/UUT/d_out(31:0)
00000000
?
80
81
82
83
?
84
85
86
87
88
80000009

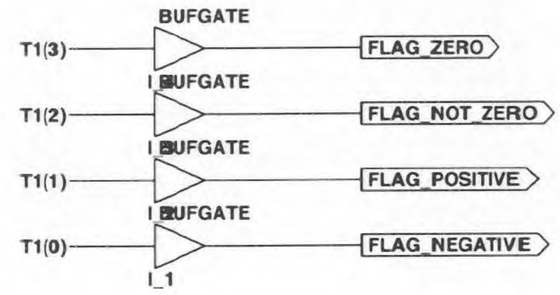
Appendix C: Circuit Schematics

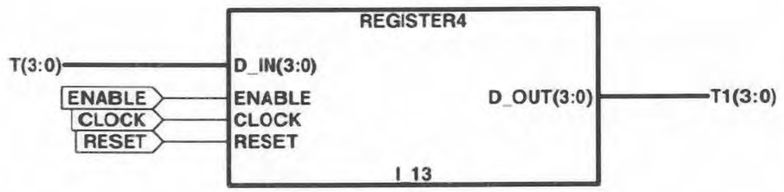

DIPS5BIT_3
DIPS(4:0)
DIPS_3
DIP3(4:0)
DIPS5BIT_2
DIPS(4:0)
DIPS_2
DIP2(4:0)
DIPS5BIT_1
DIPS(4:0)
DIPS_1
DIP1(4:0)
DIPS5BIT_0
DIPS(4:0)
DIPS_0
DIP0(4:0)
CELL
DIP(4:0)
RA_IN(15:0)
RB_IN(15:0)
ACC_IN(31:0)
C_INST(15:0)
C_INST_LOAD
C_DATA_LOAD
START_MULT
RESULT_READY
CLOCK
RESET
RA_PASS(15:0)
FIFO_OUT(31:0)
CELL_3
CELL_2
CELL_1
CELL_0
RA_PASS32(15:0)
FIFO_PASS32(31:0)
RA_PASS21(15:0)
FIFO_PASS21(31:0)
RA_PASS10(15:0)
FIFO_PASS10(31:0)
RA_RB_DATA(15:0)
ACC_TO_CELL(31:0)
CELL_INSTRUCT(15:0)
ACC_FROM_CELL(31:0)
MAIN_PROCESSOR
CELL_INST(15:0)
RARB_OUT(15:0)
ACC_OUT(31:0)
ACC_IN(31:0)
MEMACK
RESET
C_INST_LOAD
C_DATA_LOAD
START_MULT
RESULT_READY
CLOCK
MEMRD
MEMWR
MEMORY_BUS(31:0)
CONTROL_PROCESSOR
MEM_BUS(31:0)
RESET
MEMORY
MEMREAD
MEMWRITE
CLOCK
RESET
MEMACK
MEMORY_BUS(31:0)
MAIN_MEMORY

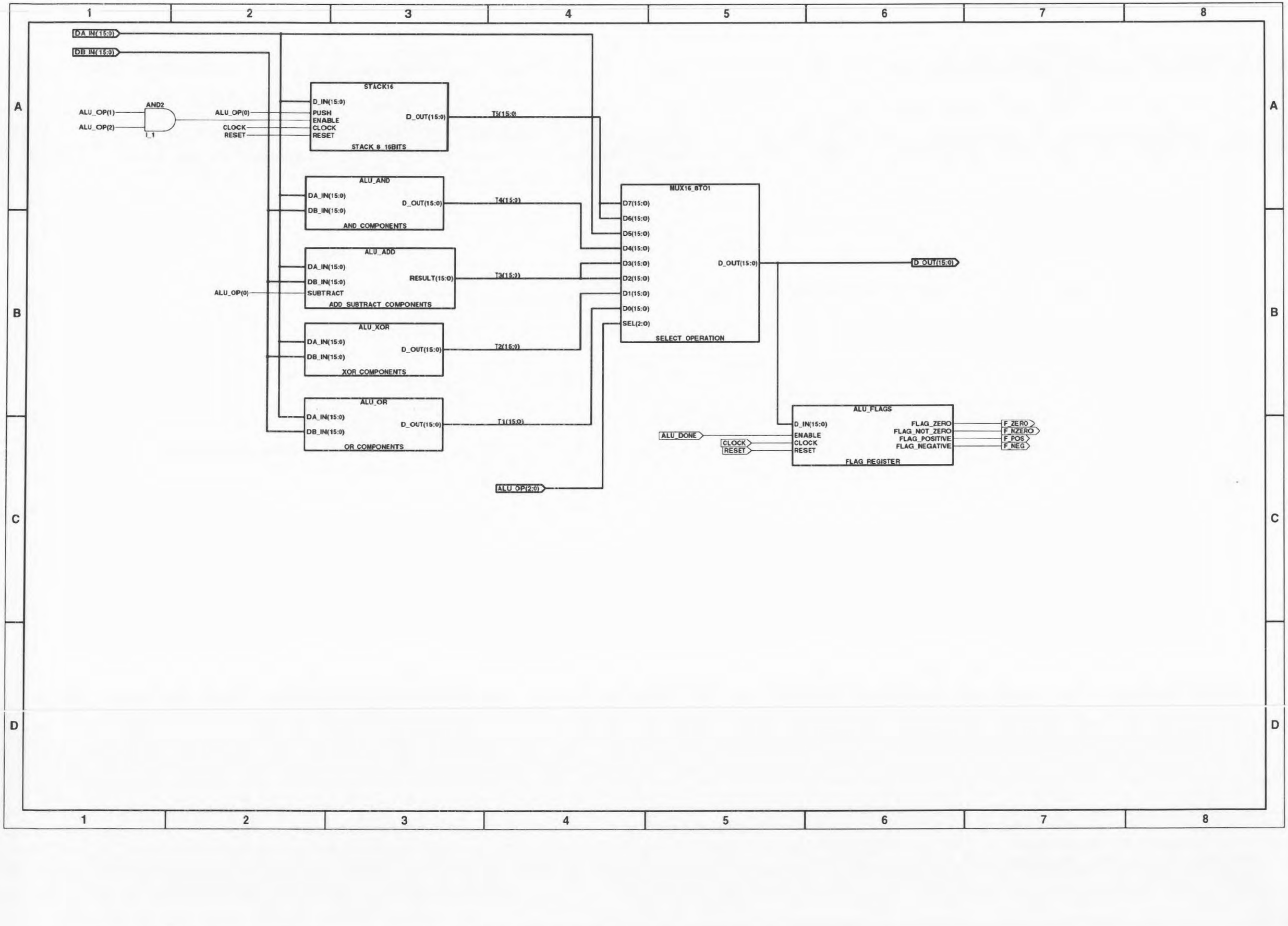

PC_UNIT
NEW_ADDRESS(15:0)
JMP
JA
F_POS
JE
F_ZERO
RETURN_EN
SUBR_VAL
SUBR_EN
PC_EN
CLOCK
RESET
I_ADDRESS(15:0)
PC_PCINC_AND_PCLOAD
MEMORY_BUS(31:16)
TRISTATE_16
D_IN(15:0)
SEL
D_OUT(15:0)
BUS_READ_INSTRUCTION
T1(15:0)
REGISTER16
ENABLE
INSTRUCTION
MEMORY_BUS(15:0)
BUS_READ_DATA
T2(15:0)
DATA_IN
EXT_BUS_RD
RD_WORD
ACC_IN(31:0)
REGISTER32
D_IN(31:0)
D_OUT(31:0)
T3(31:0)
ACCUMULATE_IN
CONV_32TO16_SM_COMP
T4(15:0)
CONVERT_ACC_IN
ALU_AND_REG
INSTRUCT(5:4)
INSTRUCT(2:0)
INSTRUCT(14:12)
INSTRUCT(10:8)
AR_IN_SEL
ALU_DONE
AR_W_EN
DBA_ADDR(2:0)
DBB_ADDR(2:0)
ALU_OP(2:0)
IN_VAL(15:0)
W_ADDR(2:0)
IN_SEL
W_ENABLE
DBA(15:0)
DBB(15:0)
F_NZERO
T5(15:0)
T7(15:0)
ALU_AND_REGISTER_FILE
A_OUT_SRC
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
CHOOSE_ADDRESS
T8(15:0)
INVGATE
MAKE_EXT_BUS_WR
ADDRESS
T10(15:0)
BUS_WRITE_ADDRESS
MEMORY_BUS(31:16)
SELECT_DATA_OUT_SRC
D_OUT_SRC
WR_WORD
T9(15:0)
WRITE_DATA
T11(15:0)
BUS_WRITE_DATA
MEMORY_BUS(15:0)
SELECT_INSTRUCTION_DRVR
INSTRUCT(15:0)
CELL_INSTRUCTION
CELL_INST_EN
CELL_INST(15:0)
RAAR_OUT(15:0)
DECODE_INSTRUCTION
INST_OUT(15:0)
INST_EXT
INST_IN(15:0)
DATA_IN(15:0)
MEMACK
CLK
INST_T2(15:0)
INST_T1(15:0)
CENTRAL_CONTROL
CONV_16TO32_TS_COMP
T12(31:0)
CONVERT_ACC_OUT
ACC_EN
ACC_OUT
ACC_OUT(31:0)
CLOCKGEN
SYS_CLOCK
CLOCK

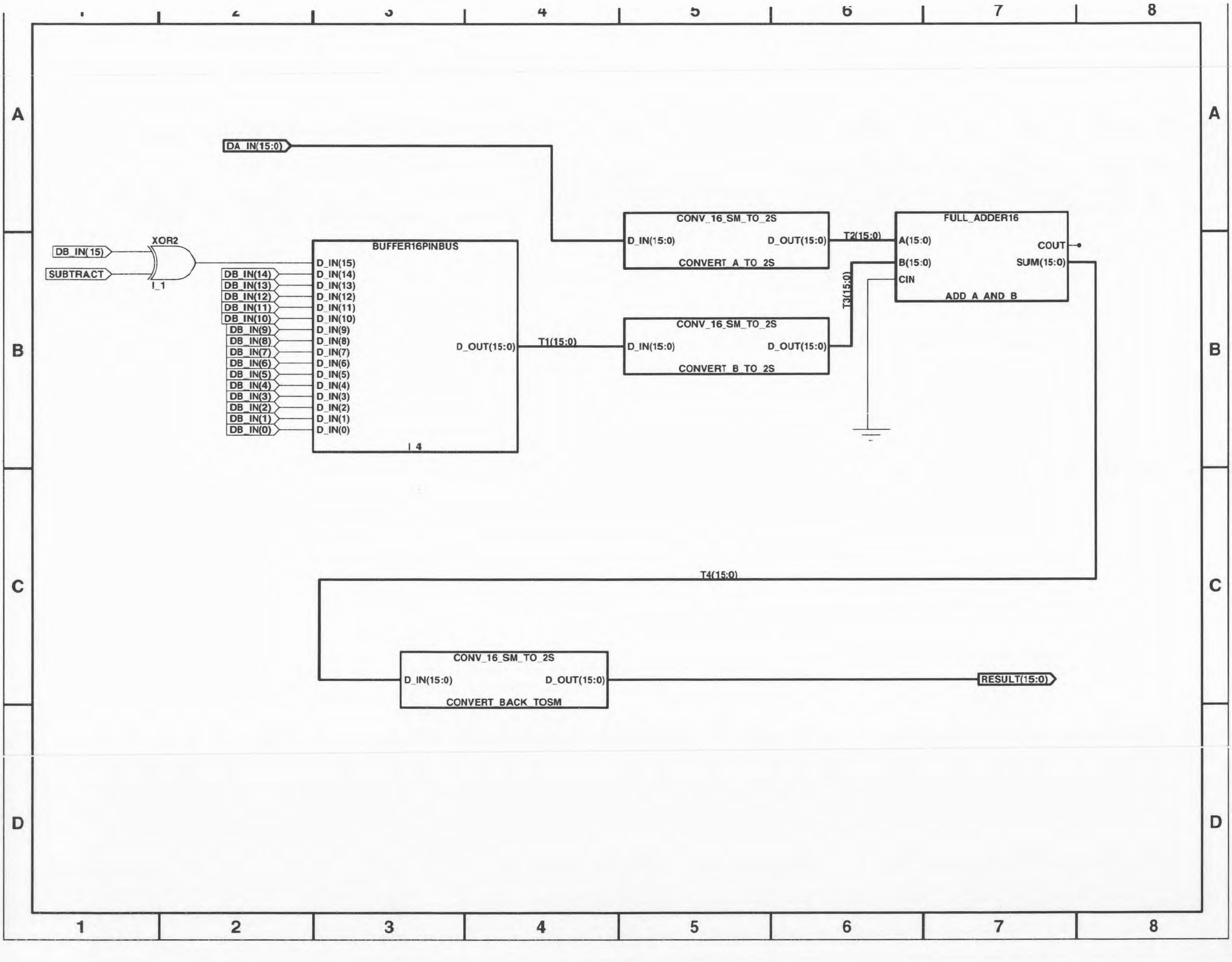

MUX_16BIT_2TO1
IN_VAL(15:0)
S_HIGH(15:0)
DBA(15:0)
S_LOW(15:0)
IN_SEL
SEL
D_OUT(15:0)
SELECT_ALU_SRC
ALU_OP(2:0)
T2(15:0)
DBB(15:0)
ALU_DONE
CLOCK
RESET
ALU
ALU_OP(2:0)
DA_IN(15:0)
DB_IN(15:0)
ALU_DONE
CLOCK
RESET
D_OUT(15:0)
F_ZERO
F_NZERO
F_POS
F_NEG
ALU_PROCESSOR
T_1(15:0)
REGFILE
DBA_ADDR(2:0)
DBB_ADDR(2:0)
W_DATA(15:0)
W_ADDR(2:0)
W_ENABLE
CLOCK
RESET
DBA(15:0)
DBB(15:0)
REGISTER_FILE
AND2
I_1

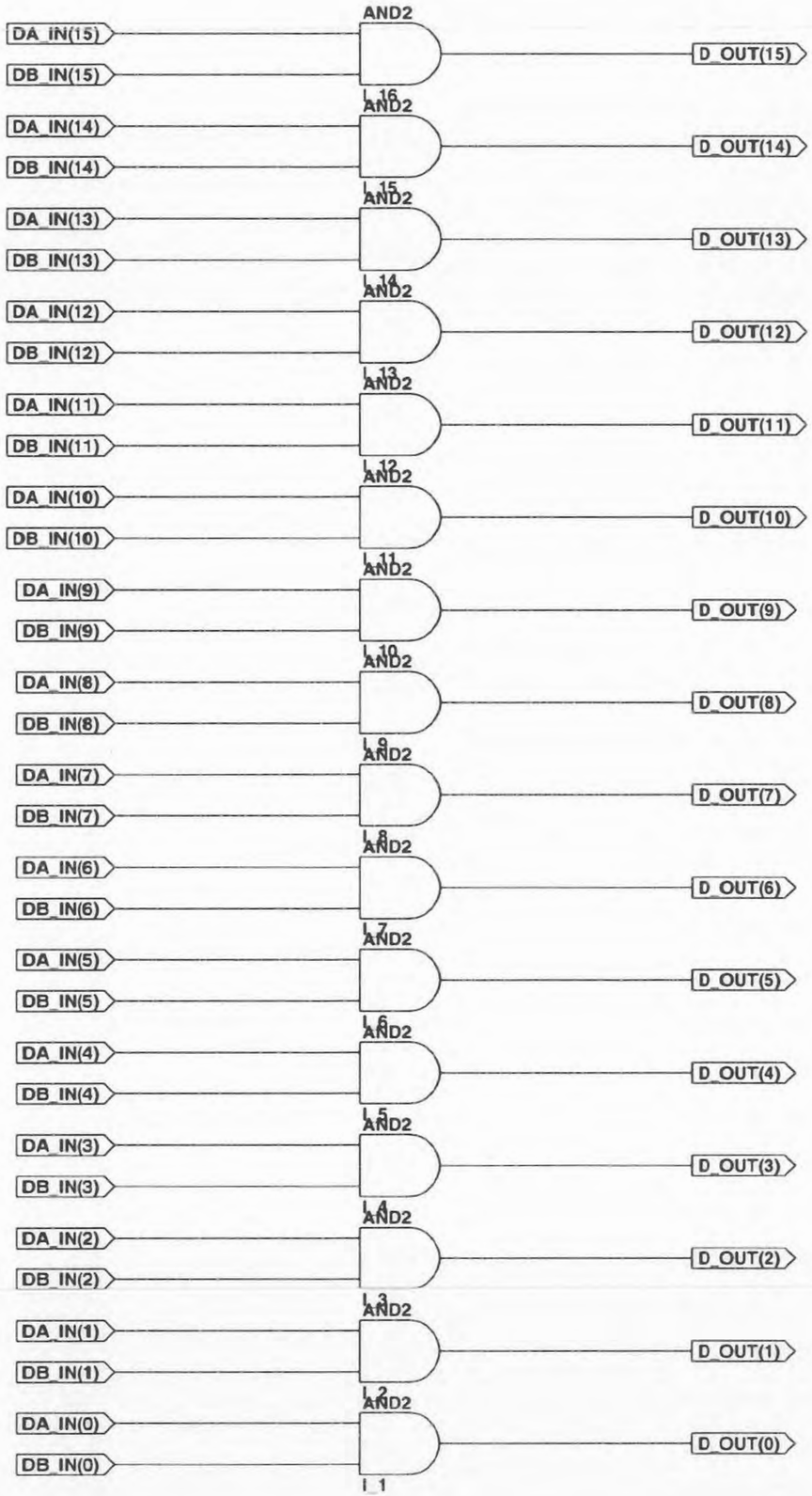

D_IN(14)
D_IN(13)
D_IN(12)
D_IN(11)
D_IN(10)
D_IN(9)
D_IN(8)
D_IN(7)
D_IN(6)
D_IN(5)
D_IN(4)
D_IN(3)
D_IN(2)
D_IN(1)
D_IN(0)
OR8
I_18
OR8
I_19
OR2
I_16
INVGATE
I_7
T(3)
T(2)
AND2
I_15
T(0)
D_IN(15)
INVGATE
I_20
AND2
I_14
T(1)

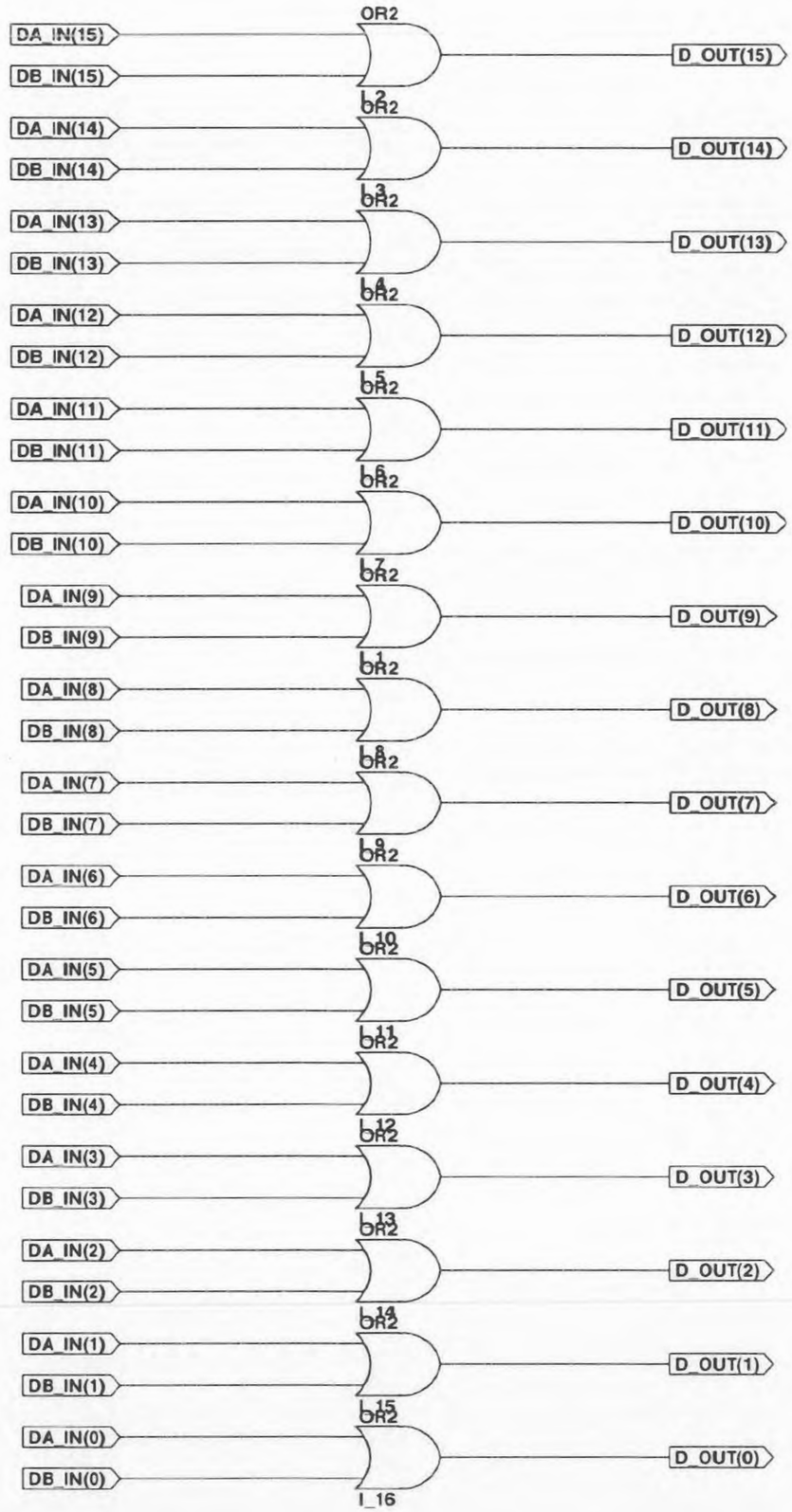

BUFGATE
T1(3)
FLAG_ZERO
I_BUFGATE
T1(2)
FLAG_NOT_ZERO
I_BUFGATE
T1(1)
FLAG_POSITIVE
I_BUFGATE
T1(0)
FLAG_NEGATIVE
I_1

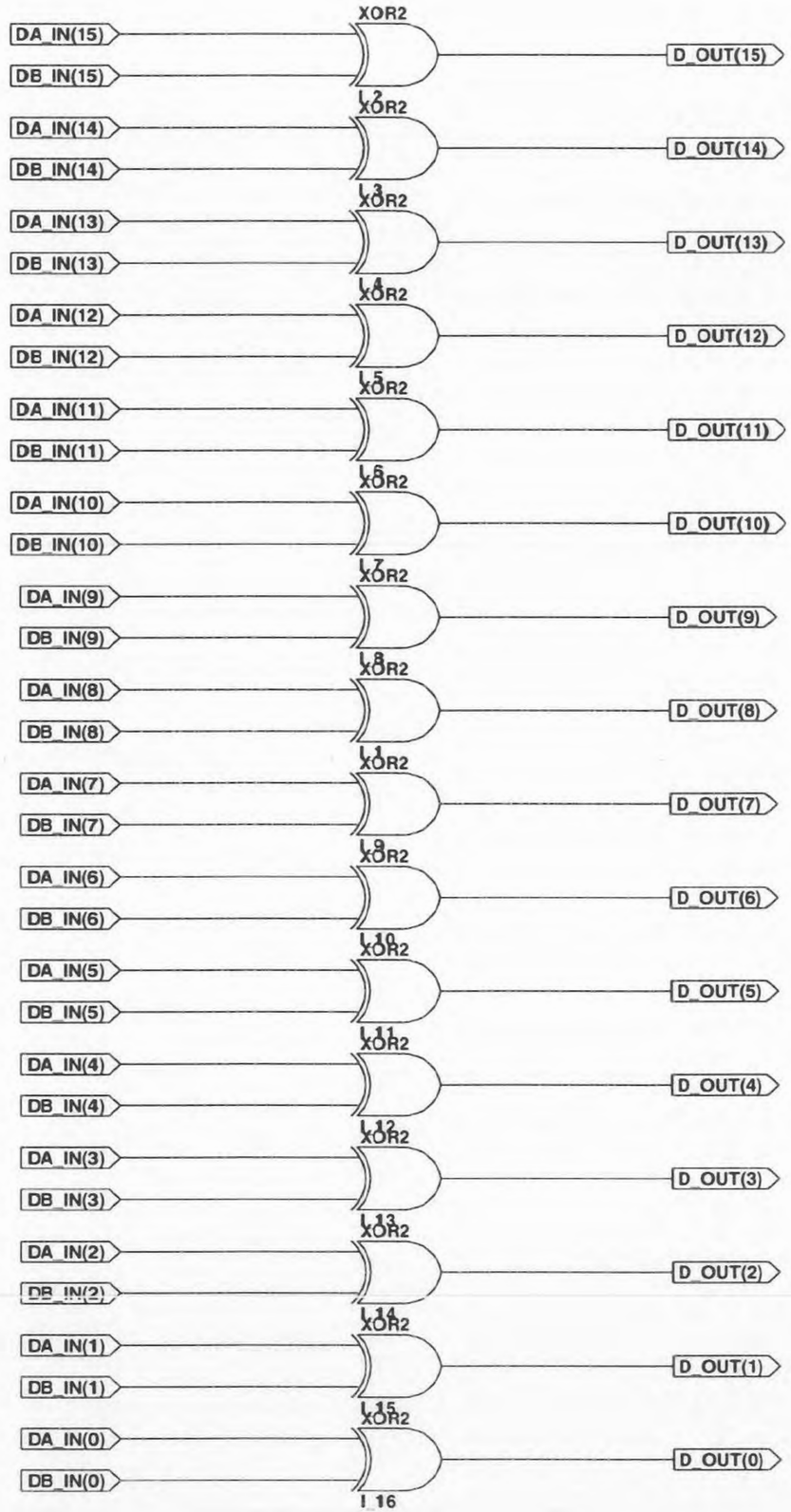

REGISTER4
T(3:0)
D_IN(3:0)
ENABLE
ENABLE
CLOCK
CLOCK
RESET
RESET
D_OUT(3:0)
T1(3:0)
I_13

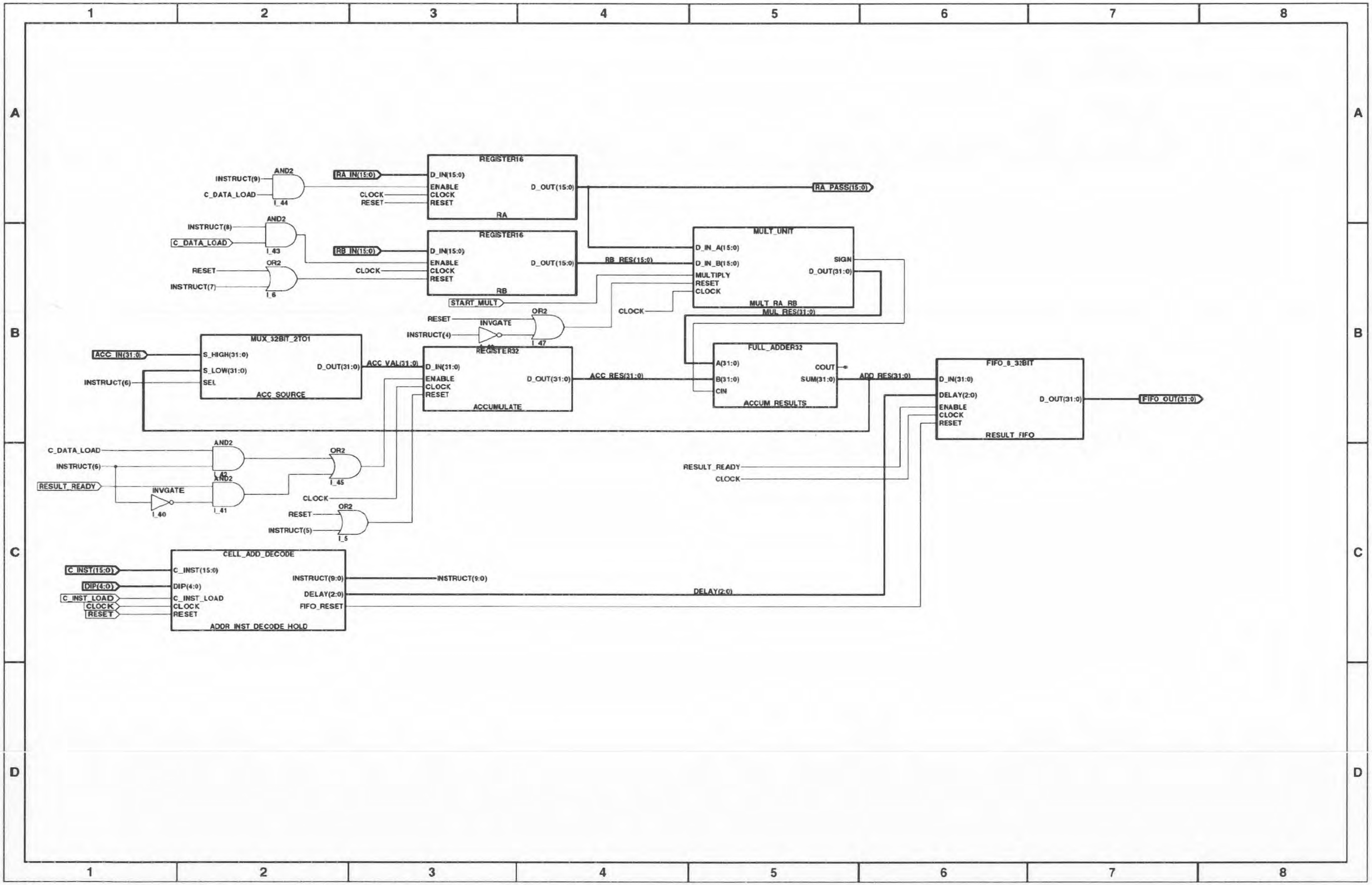

DA_IN(15:0)
DB_IN(15:0)
AND2
ALU_OP(1)
ALU_OP(2)
I_1
ALU_OP(0)
CLOCK
RESET
STACK16
D_IN(15:0)
PUSH
ENABLE
CLOCK
RESET
D_OUT(15:0)
STACK_8_16BITS
T5(15:0)
ALU_AND
DA_IN(15:0)
DB_IN(15:0)
D_OUT(15:0)
AND_COMPONENTS
T4(15:0)
ALU_ADD
DA_IN(15:0)
DB_IN(15:0)
SUBTRACT
RESULT(15:0)
ADD_SUBTRACT_COMPONENTS
T3(15:0)
ALU_XOR
DA_IN(15:0)
DB_IN(15:0)
D_OUT(15:0)
XOR_COMPONENTS
T2(15:0)
ALU_OR
DA_IN(15:0)
DB_IN(15:0)
D_OUT(15:0)
OR_COMPONENTS
T1(15:0)
MUX16_8TO1
D7(15:0)
D6(15:0)
D5(15:0)
D4(15:0)
D3(15:0)
D2(15:0)
D1(15:0)
D0(15:0)
SEL(2:0)
D_OUT(15:0)
SELECT_OPERATION
D_OUT(15:0)
ALU_OP(2:0)
ALU_FLAGS
D_IN(15:0)
ENABLE
CLOCK
RESET
FLAG_ZERO
FLAG_NOT_ZERO
FLAG_POSITIVE
FLAG_NEGATIVE
FLAG_REGISTER
ALU_DONE
CLOCK
RESET
F_ZERO
F_NZERO
F_POS
F_NEG


DA_IN(15:0)
DB_IN(15)
SUBTRACT
XOR2
I_1
BUFFER16PINBUS
DB_IN(14)
DB_IN(13)
DB_IN(12)
DB_IN(11)
DB_IN(10)
DB_IN(9)
DB_IN(8)
DB_IN(7)
DB_IN(6)
DB_IN(5)
DB_IN(4)
DB_IN(3)
DB_IN(2)
DB_IN(1)
DB_IN(0)
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)
D_OUT(15:0)
I_4
T1(15:0)
CONV_16_SM_TO_2S
D_IN(15:0)
D_OUT(15:0)
CONVERT_A_TO_2S
T2(15:0)
CONV_16_SM_TO_2S
D_IN(15:0)
D_OUT(15:0)
CONVERT_B_TO_2S
T3(15:0)
FULL_ADDER16
A(15:0)
B(15:0)
CIN
COUT
SUM(15:0)
ADD_A_AND_B
T4(15:0)
CONV_16_SM_TO_2S
D_IN(15:0)
D_OUT(15:0)
CONVERT_BACK_TOSM
RESULT(15:0)

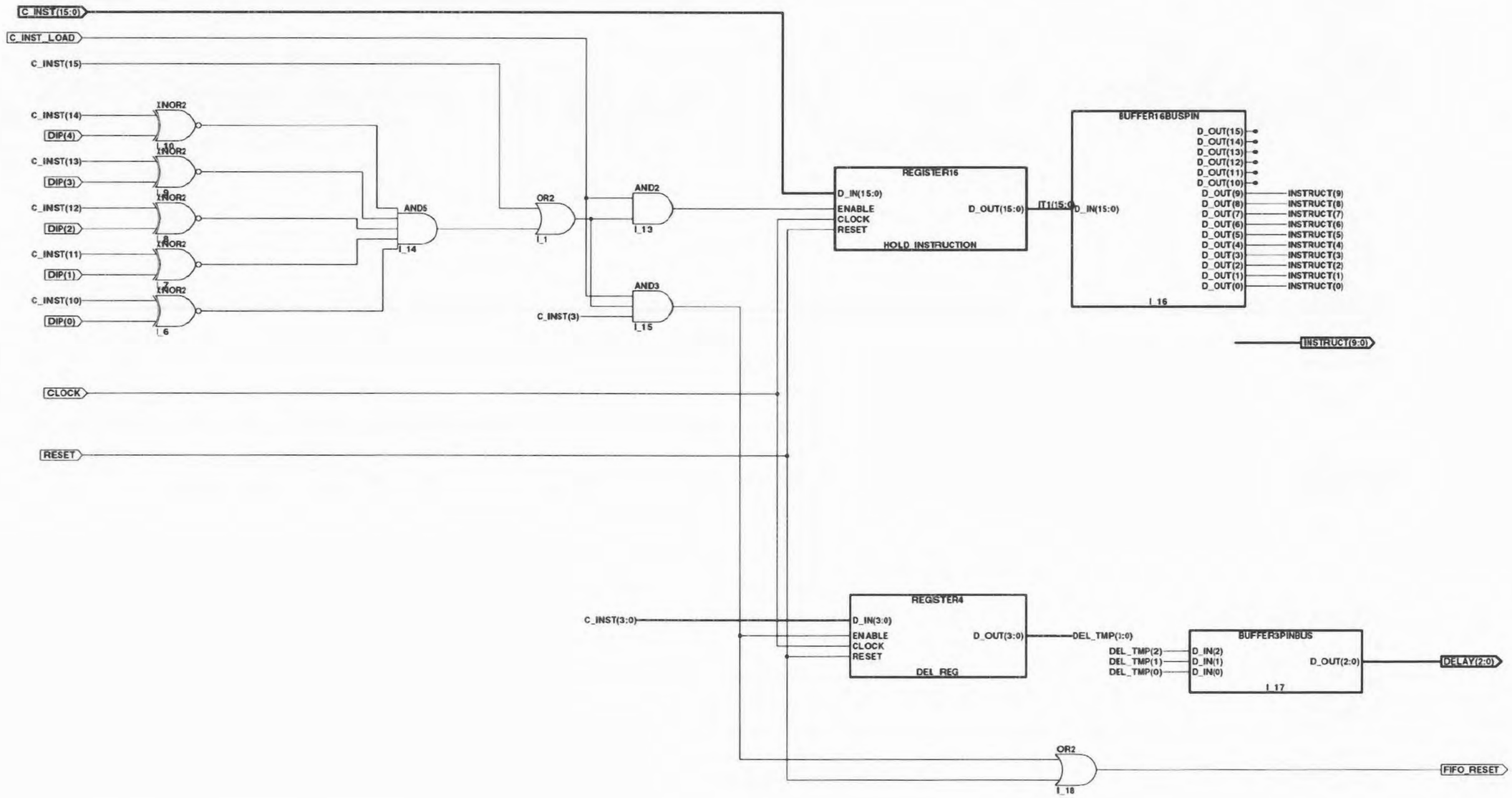

AND2
DA_IN(15)
DB_IN(15)
D_OUT(15)
I_16
DA_IN(14)
DB_IN(14)
D_OUT(14)
I_15
DA_IN(13)
DB_IN(13)
D_OUT(13)
I_14
DA_IN(12)
DB_IN(12)
D_OUT(12)
I_13
DA_IN(11)
DB_IN(11)
D_OUT(11)
I_12
DA_IN(10)
DB_IN(10)
D_OUT(10)
I_11
DA_IN(9)
DB_IN(9)
D_OUT(9)
I_10
DA_IN(8)
DB_IN(8)
D_OUT(8)
I_9
DA_IN(7)
DB_IN(7)
D_OUT(7)
I_8
DA_IN(6)
DB_IN(6)
D_OUT(6)
I_7
DA_IN(5)
DB_IN(5)
D_OUT(5)
I_6
DA_IN(4)
DB_IN(4)
D_OUT(4)
I_5
DA_IN(3)
DB_IN(3)
D_OUT(3)
I_4
DA_IN(2)
DB_IN(2)
D_OUT(2)
I_3
DA_IN(1)
DB_IN(1)
D_OUT(1)
I_2
DA_IN(0)
DB_IN(0)
D_OUT(0)
I_1

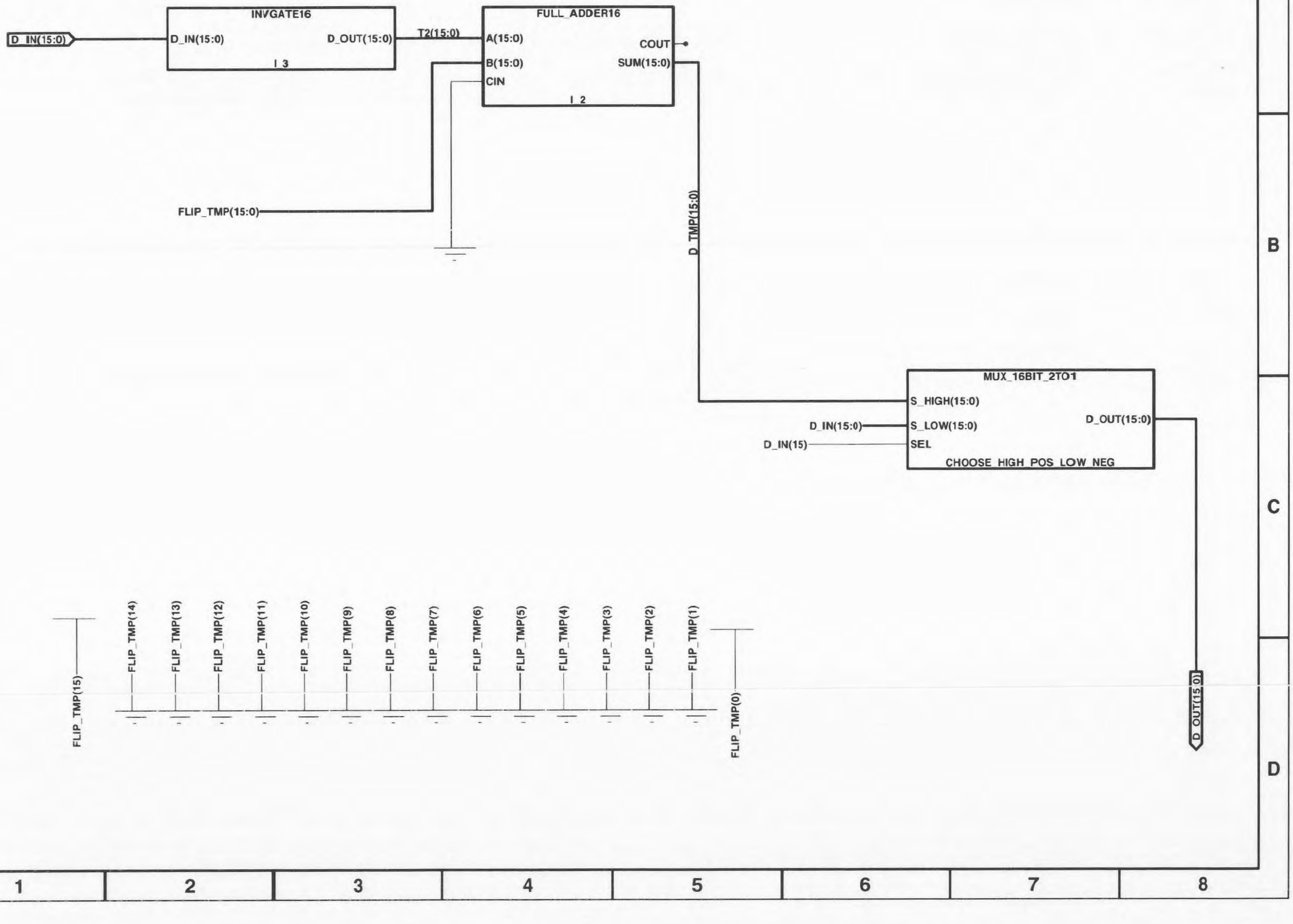

OR2
DA_IN(15)
DB_IN(15)
D_OUT(15)
I_2
DA_IN(14)
DB_IN(14)
D_OUT(14)
I_3
DA_IN(13)
DB_IN(13)
D_OUT(13)
I_4
DA_IN(12)
DB_IN(12)
D_OUT(12)
I_5
DA_IN(11)
DB_IN(11)
D_OUT(11)
I_6
DA_IN(10)
DB_IN(10)
D_OUT(10)
I_7
DA_IN(9)
DB_IN(9)
D_OUT(9)
I_1
DA_IN(8)
DB_IN(8)
D_OUT(8)
I_8
DA_IN(7)
DB_IN(7)
D_OUT(7)
I_9
DA_IN(6)
DB_IN(6)
D_OUT(6)
I_10
DA_IN(5)
DB_IN(5)
D_OUT(5)
I_11
DA_IN(4)
DB_IN(4)
D_OUT(4)
I_12
DA_IN(3)
DB_IN(3)
D_OUT(3)
I_13
DA_IN(2)
DB_IN(2)
D_OUT(2)
I_14
DA_IN(1)
DB_IN(1)
D_OUT(1)
I_15
DA_IN(0)
DB_IN(0)
D_OUT(0)
I_16

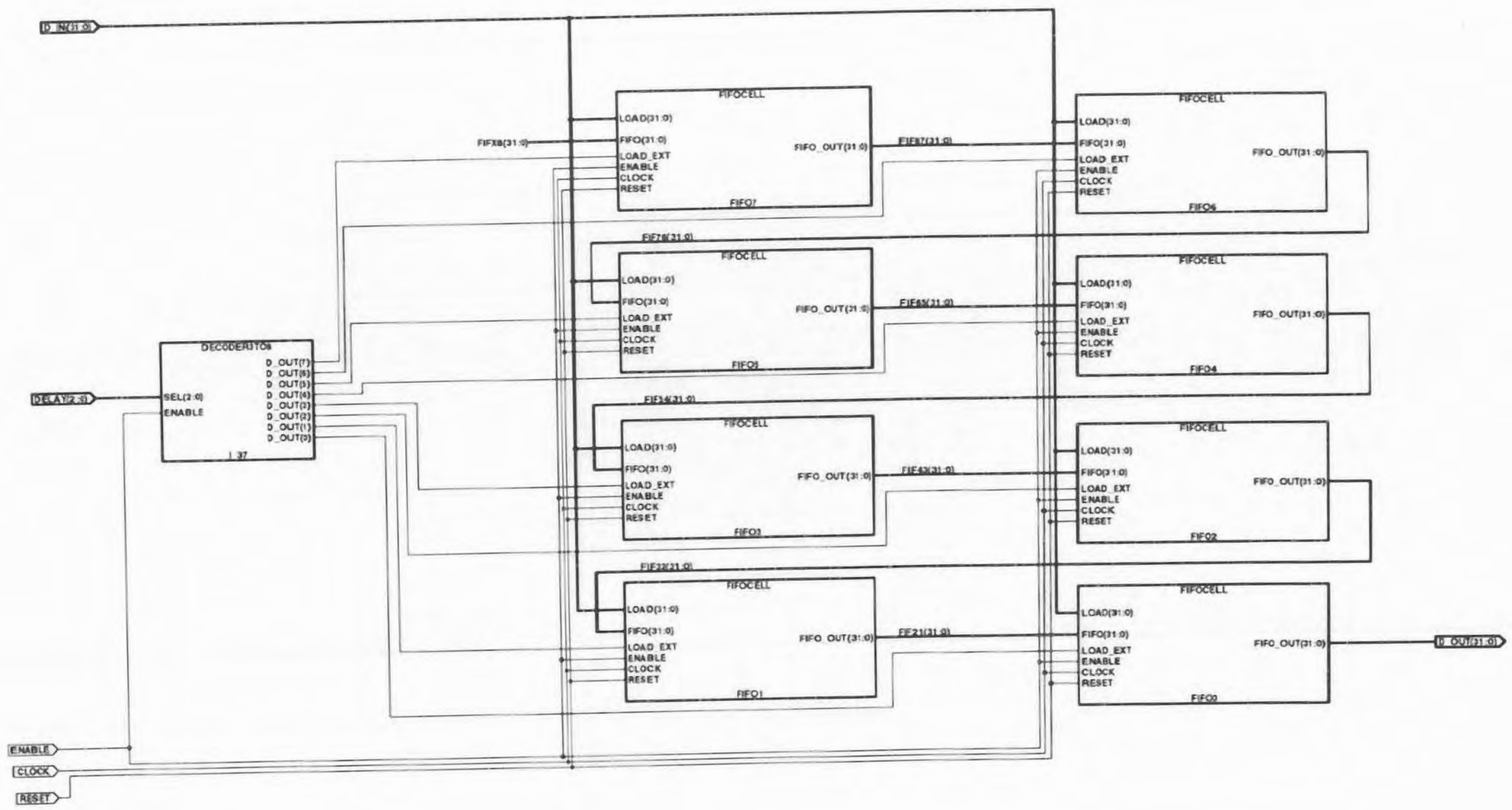

XOR2
DA_IN(15)
DB_IN(15)
D_OUT(15)
I_2
XOR2
DA_IN(14)
DB_IN(14)
D_OUT(14)
I_3
XOR2
DA_IN(13)
DB_IN(13)
D_OUT(13)
I_4
XOR2
DA_IN(12)
DB_IN(12)
D_OUT(12)
I_5
XOR2
DA_IN(11)
DB_IN(11)
D_OUT(11)
I_6
XOR2
DA_IN(10)
DB_IN(10)
D_OUT(10)
I_7
XOR2
DA_IN(9)
DB_IN(9)
D_OUT(9)
I_8
XOR2
DA_IN(8)
DB_IN(8)
D_OUT(8)
I_1
XOR2
DA_IN(7)
DB_IN(7)
D_OUT(7)
I_9
XOR2
DA_IN(6)
DB_IN(6)
D_OUT(6)
I_10
XOR2
DA_IN(5)
DB_IN(5)
D_OUT(5)
I_11
XOR2
DA_IN(4)
DB_IN(4)
D_OUT(4)
I_12
XOR2
DA_IN(3)
DB_IN(3)
D_OUT(3)
I_13
XOR2
DA_IN(2)
DB_IN(2)
D_OUT(2)
I_14
XOR2
DA_IN(1)
DB_IN(1)
D_OUT(1)
I_15
XOR2
DA_IN(0)
DB_IN(0)
D_OUT(0)
I_16



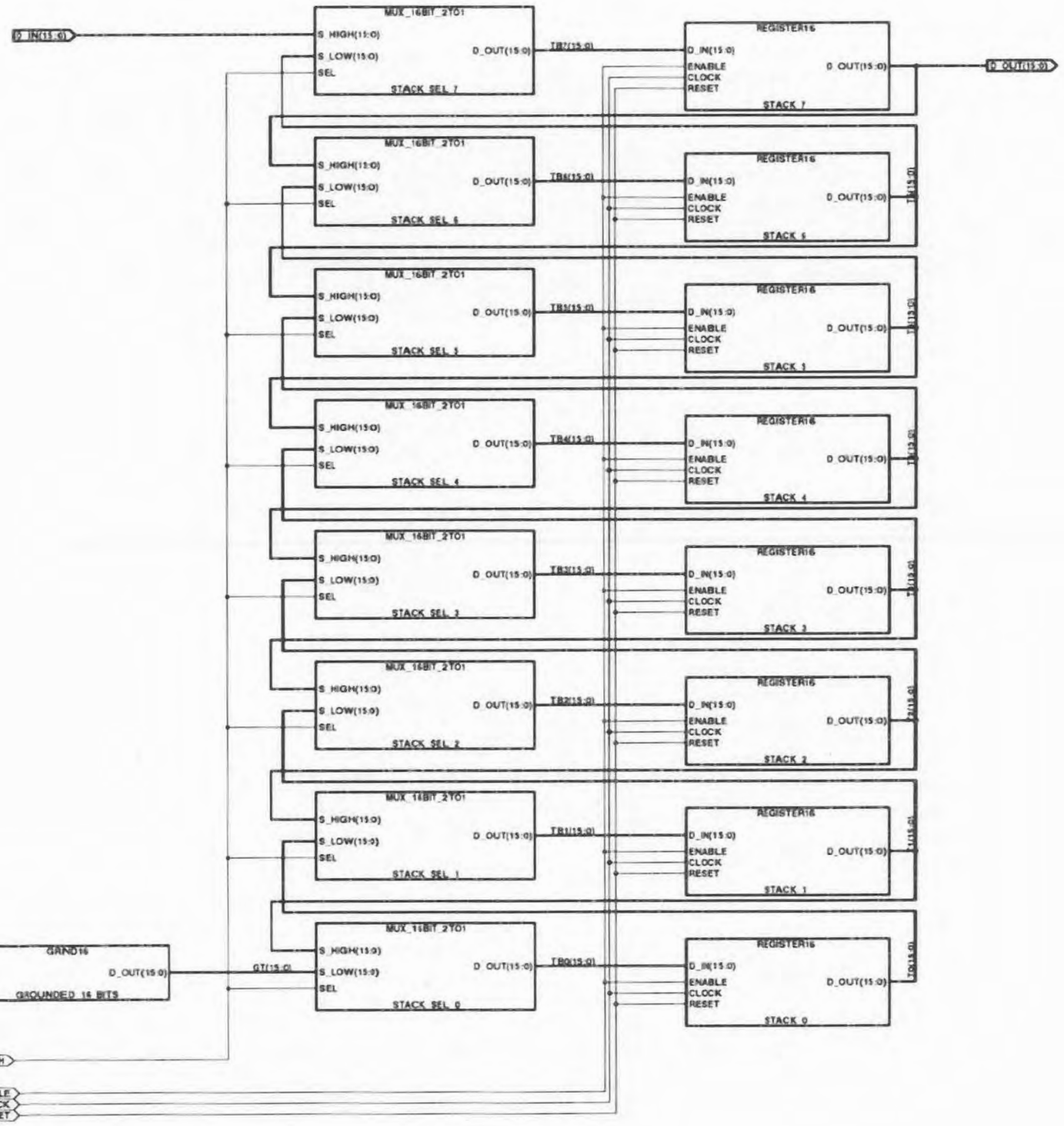
REGISTER16
RA
D_IN(15:0)
ENABLE
CLOCK
RESET
D_OUT(15:0)
RA_IN(15:0)
RA_PASS(15:0)
AND2
INSTRUCT(9)
C_DATA_LOAD
I_44
INSTRUCT(8)
I_43
RB_IN(15:0)
RB
RB_RES(15:0)
OR2
RESET
INSTRUCT(7)
I_6
START_MULT
MULT_UNIT
D_IN_A(15:0)
D_IN_B(15:0)
MULTIPLY
SIGN
D_OUT(31:0)
MULT_RA_RB
MUL_RES(31:0)
INVGATE
INSTRUCT(4)
I_47
MUX_32BIT_2TO1
ACC_IN(31:0)
S_HIGH(31:0)
S_LOW(31:0)
SEL
INSTRUCT(6)
ACC_SOURCE
ACC_VAL(31:0)
REGISTER32
ACCUMULATE
ACC_RES(31:0)
FULL_ADDER32
A(31:0)
B(31:0)
CIN
COUT
SUM(31:0)
ACCUM_RESULTS
ADD_RES(31:0)
FIFO_8_32BIT
DELAY(2:0)
RESULT_FIFO
FIFO_OUT(31:0)
RESULT_READY
I_42
I_45
I_41
I_40
I_5
CELL_ADD_DECODE
C_INST(15:0)
DIP(4:0)
C_INST_LOAD
INSTRUCT(9:0)
DELAY(2:0)
FIFO_RESET
ADDR_INST_DECODE_HOLD

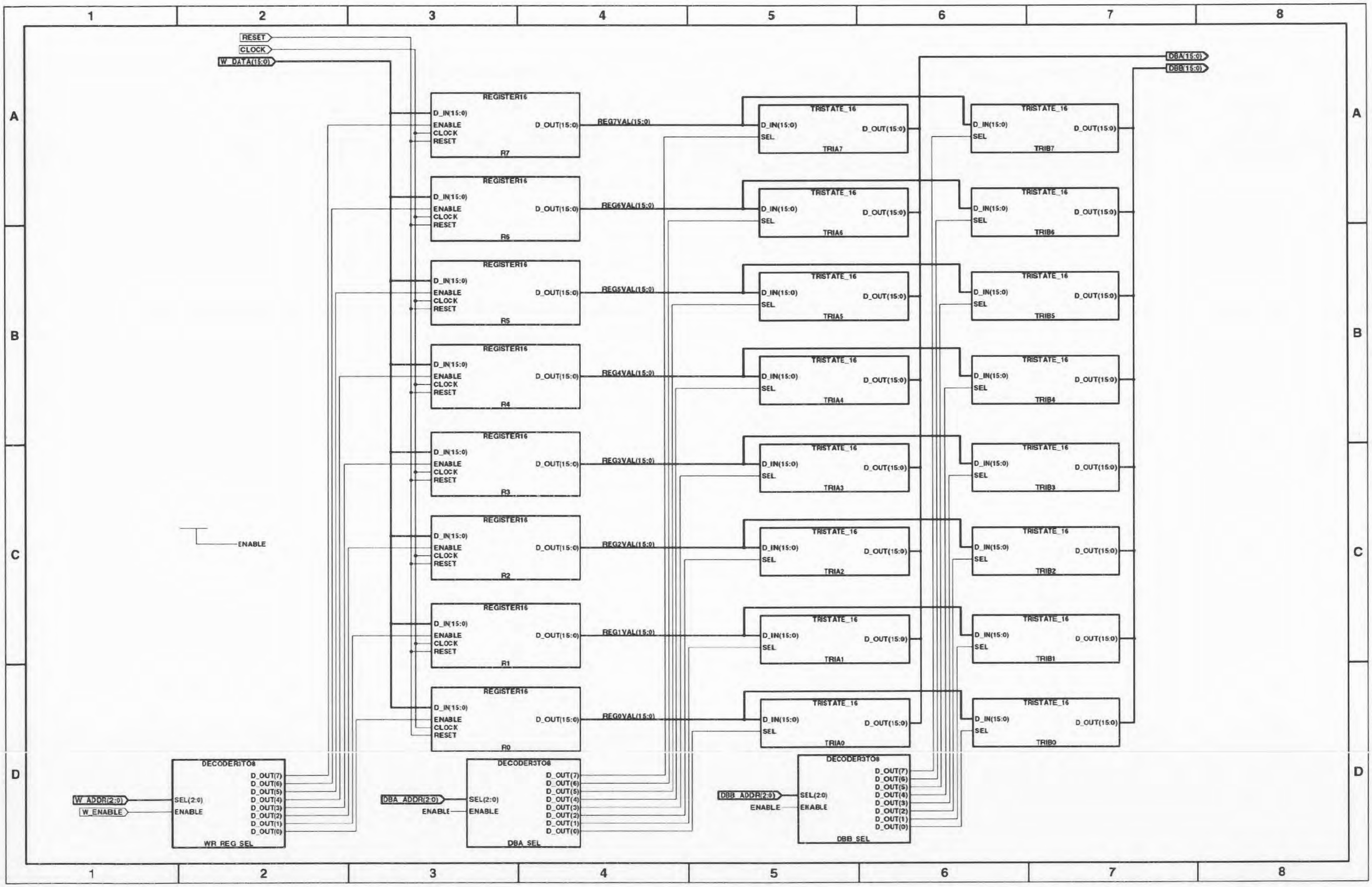

C_INST(15:0)
C_INST_LOAD
C_INST(15)
C_INST(14)
DIP(4)
C_INST(13)
DIP(3)
C_INST(12)
DIP(2)
C_INST(11)
DIP(1)
C_INST(10)
DIP(0)
INOR2
I_10
I_9
I_8
I_7
I_6
AND5
I_14
OR2
I_1
AND2
I_13
AND3
I_15
C_INST(3)
REGISTER16
D_IN(15:0)
ENABLE
CLOCK
RESET
D_OUT(15:0)
HOLD_INSTRUCTION
IT1(15:0)
BUFFER16BUSPIN
D_IN(15:0)
D_OUT(15)
D_OUT(14)
D_OUT(13)
D_OUT(12)
D_OUT(11)
D_OUT(10)
D_OUT(9)
D_OUT(8)
D_OUT(7)
D_OUT(6)
D_OUT(5)
D_OUT(4)
D_OUT(3)
D_OUT(2)
D_OUT(1)
D_OUT(0)
INSTRUCT(9)
INSTRUCT(8)
INSTRUCT(7)
INSTRUCT(6)
INSTRUCT(5)
INSTRUCT(4)
INSTRUCT(3)
INSTRUCT(2)
INSTRUCT(1)
INSTRUCT(0)
I_16
INSTRUCT(9:0)
CLOCK
RESET
C_INST(3:0)
REGISTER4
D_IN(3:0)
ENABLE
CLOCK
RESET
D_OUT(3:0)
DEL_REG
DEL_TMP(1:0)
BUFFER3PINBUS
DEL_TMP(2)
DEL_TMP(1)
DEL_TMP(0)
D_IN(2)
D_IN(1)
D_IN(0)
D_OUT(2:0)
I_17
DELAY(2:0)
OR2
I_18
FIFO_RESET

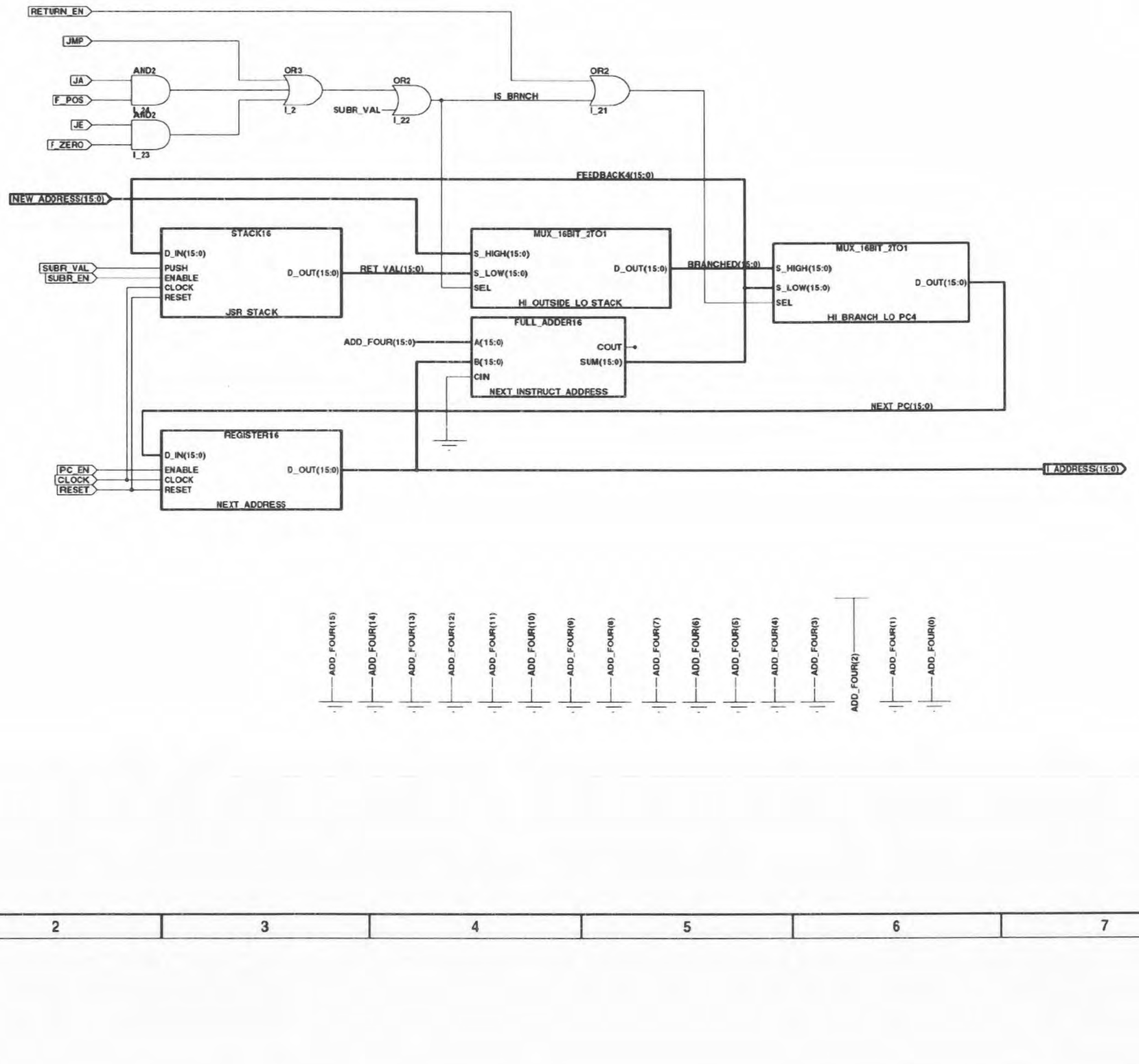

INVGATE16
D_IN(15:0)
D_OUT(15:0)
I_3
T2(15:0)
FULL_ADDER16
A(15:0)
B(15:0)
CIN
COUT
SUM(15:0)
I_2
D_IN(15:0)
FLIP_TMP(15:0)
D_TMP(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
D_IN(15:0)
S_LOW(15:0)
D_IN(15)
SEL
D_OUT(15:0)
CHOOSE HIGH POS LOW NEG
FLIP_TMP(15)
FLIP_TMP(14)
FLIP_TMP(13)
FLIP_TMP(12)
FLIP_TMP(11)
FLIP_TMP(10)
FLIP_TMP(9)
FLIP_TMP(8)
FLIP_TMP(7)
FLIP_TMP(6)
FLIP_TMP(5)
FLIP_TMP(4)
FLIP_TMP(3)
FLIP_TMP(2)
FLIP_TMP(1)
FLIP_TMP(0)
D_OUT(15:0)

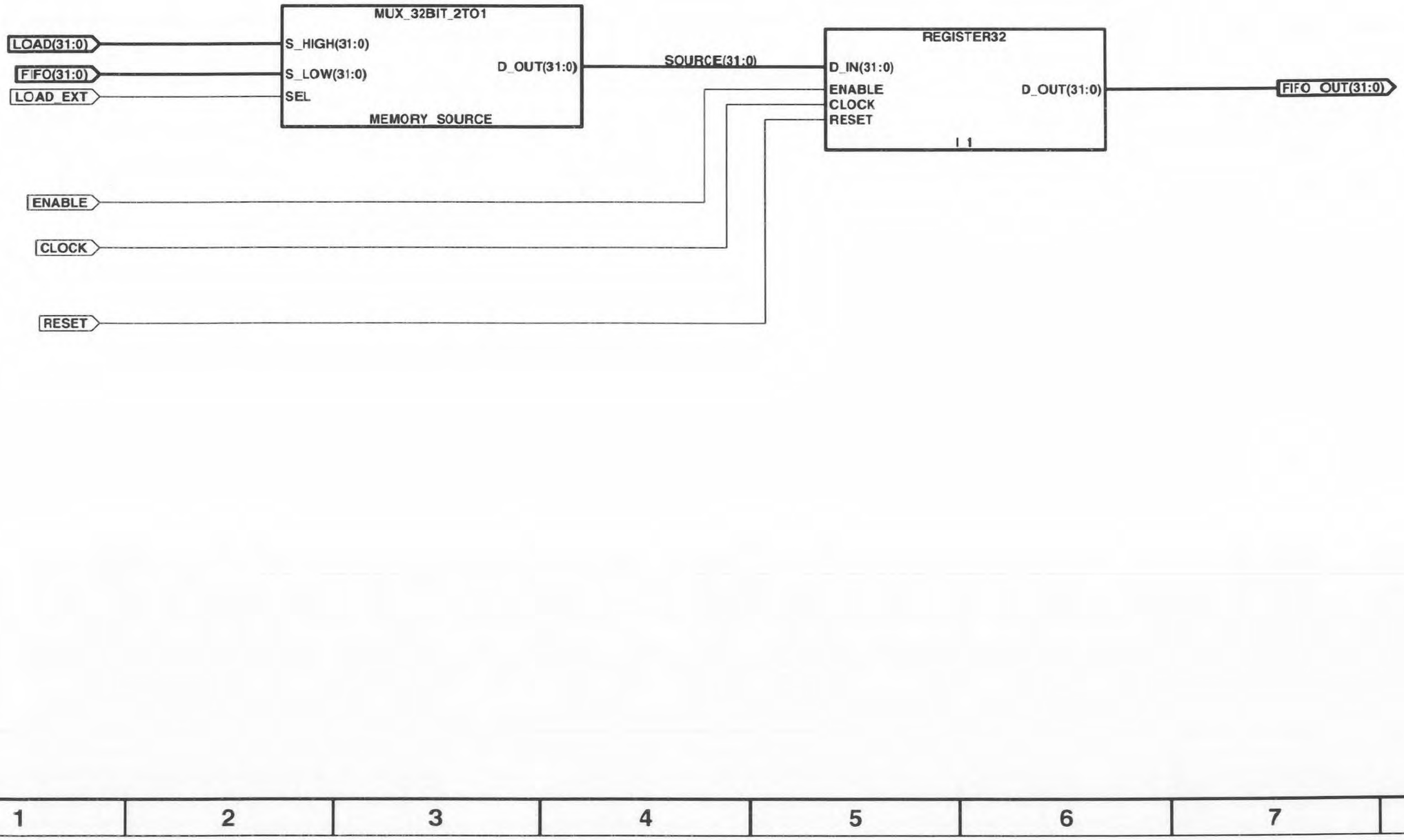

D_IN(31:0)
DELAY(2:0)
ENABLE
CLOCK
RESET
DECODER3TO8
SEL(2:0)
ENABLE
D_OUT(7)
D_OUT(6)
D_OUT(5)
D_OUT(4)
D_OUT(3)
D_OUT(2)
D_OUT(1)
D_OUT(0)
37
FIFOCELL
LOAD(31:0)
FIFO(31:0)
LOAD_EXT
ENABLE
CLOCK
RESET
FIFO_OUT(31:0)
FIFO7
FIFO6
FIFO5
FIFO4
FIFO3
FIFO2
FIFO1
FIFO0
FIFX8(31:0)
FIF87(31:0)
FIF76(31:0)
FIF65(31:0)
FIF54(31:0)
FIF43(31:0)
FIF32(31:0)
FIF21(31:0)
D_OUT(31:0)

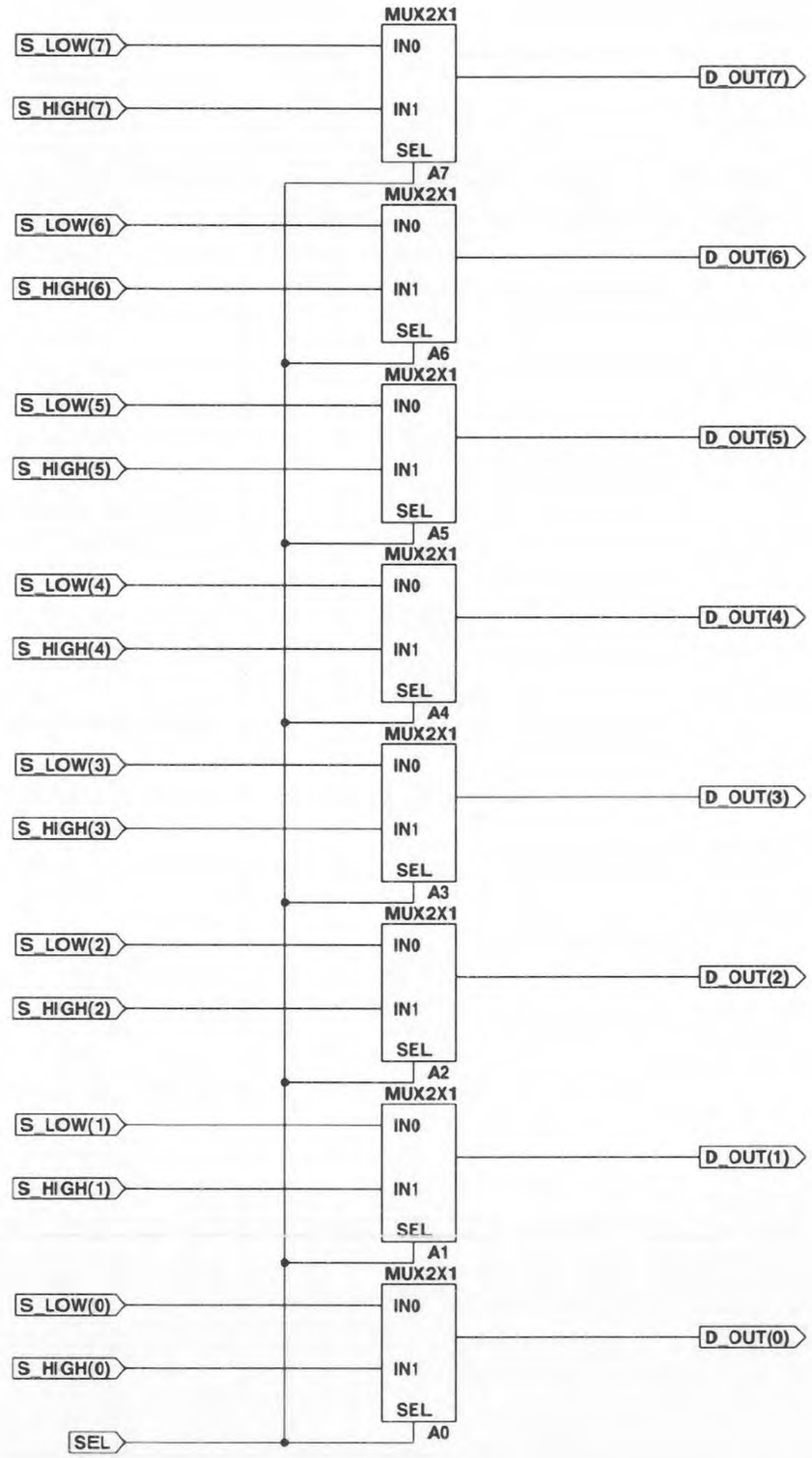
FIFOX8(31)
FIFOX8(30)
FIFOX8(29)
FIFOX8(28)
FIFOX8(27)
FIFOX8(26)
FIFOX8(25)
FIFOX8(24)
FIFOX8(23)
FIFOX8(22)
FIFOX8(21)
FIFOX8(20)
FIFOX8(19)
FIFOX8(18)
FIFOX8(17)
FIFOX8(16)
FIFOX8(15)
FIFOX8(14)
FIFOX8(13)
FIFOX8(12)
FIFOX8(11)
FIFOX8(10)
FIFOX8(9)
FIFOX8(8)
FIFOX8(7)
FIFOX8(6)
FIFOX8(5)
FIFOX8(4)
FIFOX8(3)
FIFOX8(2)
FIFOX8(1)
FIFOX8(0)

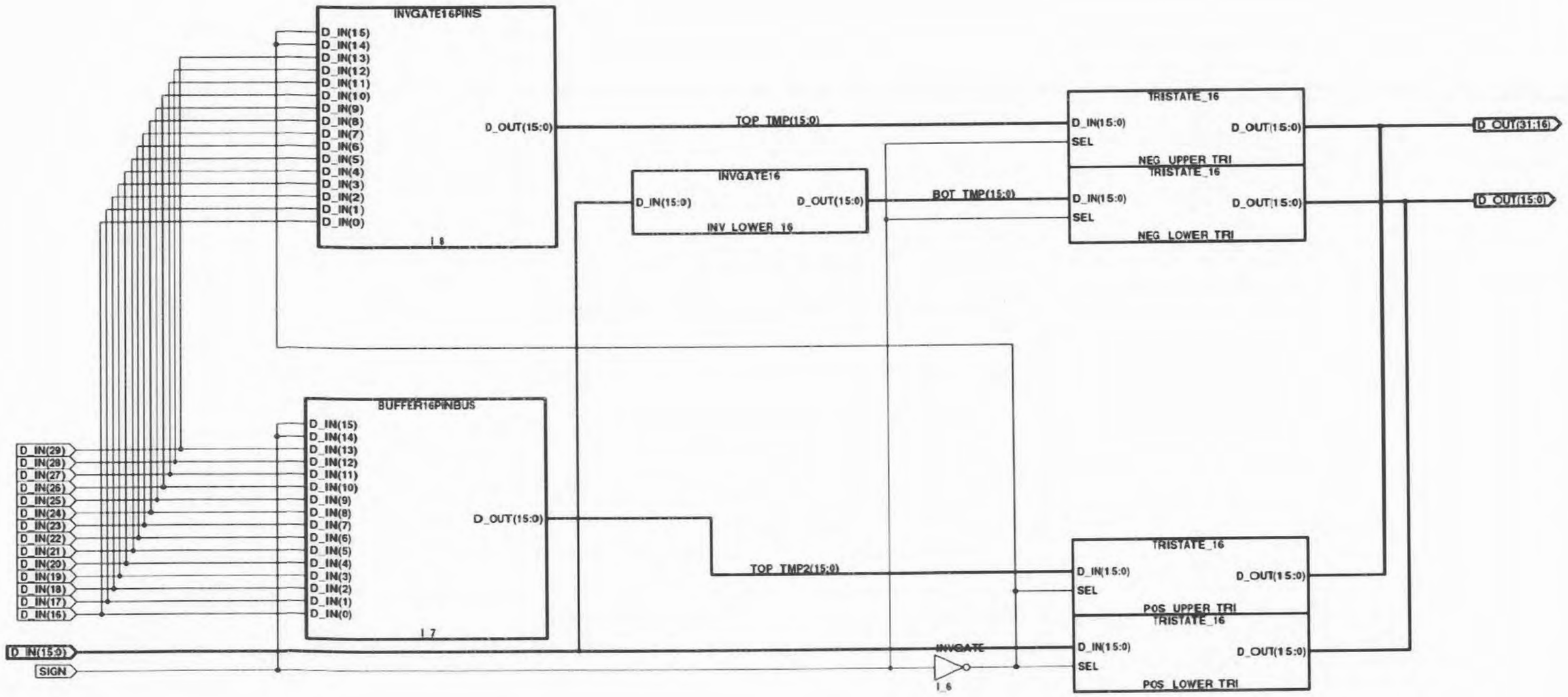

D_IN(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
STACK_SEL_7
TB7(15:0)
REGISTER16
D_IN(15:0)
ENABLE
CLOCK
RESET
D_OUT(15:0)
STACK_7
D_OUT(15:0)
STACK_SEL_6
TB6(15:0)
STACK_6
STACK_SEL_5
TB5(15:0)
STACK_5
STACK_SEL_4
TB4(15:0)
STACK_4
STACK_SEL_3
TB3(15:0)
STACK_3
STACK_SEL_2
TB2(15:0)
STACK_2
STACK_SEL_1
TB1(15:0)
STACK_1
GAND16
GROUNDED_16_BITS
GT(15:0)
STACK_SEL_0
TB0(15:0)
STACK_0
PUSH
ENABLE
CLOCK
RESET

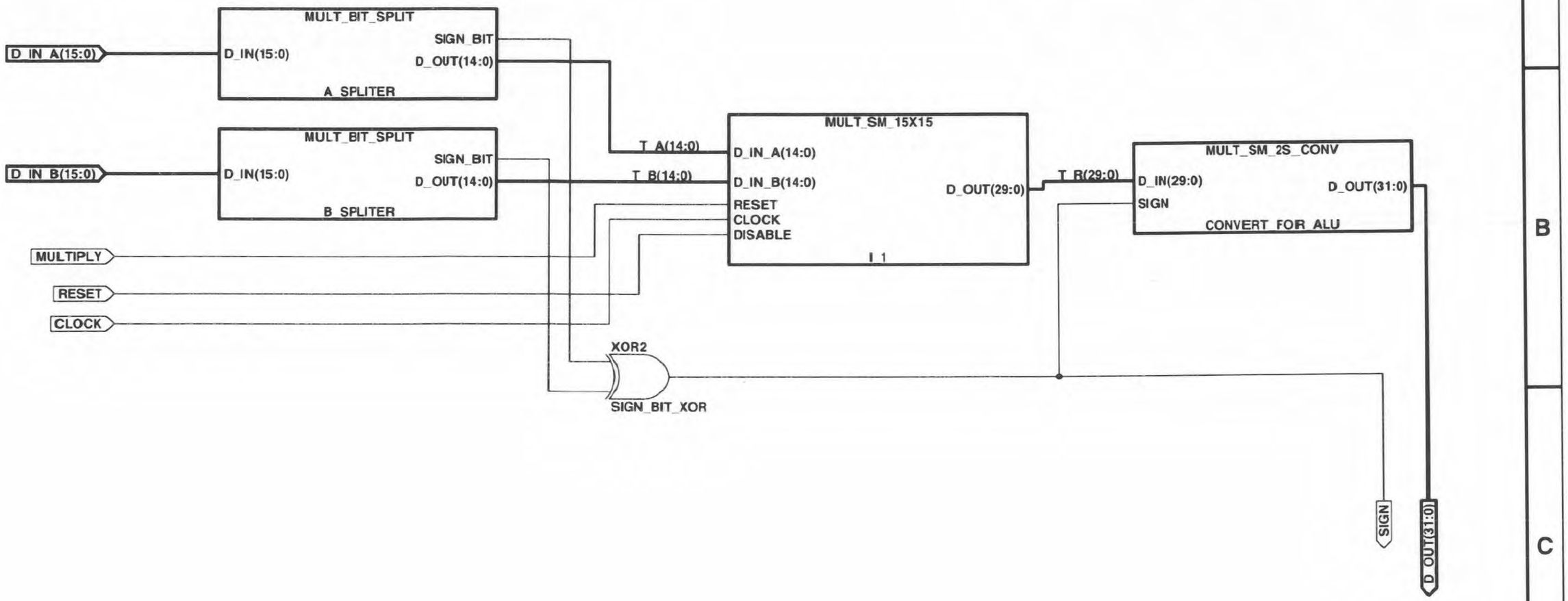

RESET
CLOCK
W_DATA(15:0)
DBA(15:0)
DBB(15:0)
REGISTER16
D_IN(15:0)
ENABLE
CLOCK
RESET
D_OUT(15:0)
R7
R6
R5
R4
R3
R2
R1
R0
REG7VAL(15:0)
REG6VAL(15:0)
REG5VAL(15:0)
REG4VAL(15:0)
REG3VAL(15:0)
REG2VAL(15:0)
REG1VAL(15:0)
REG0VAL(15:0)
TRISTATE_16
SEL
TRIA7
TRIA6
TRIA5
TRIA4
TRIA3
TRIA2
TRIA1
TRIA0
TRIB7
TRIB6
TRIB5
TRIB4
TRIB3
TRIB2
TRIB1
TRIB0
ENABLE
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)
W_ADDR(2:0)
W_ENABLE
WR_REG_SEL
DBA_ADDR(2:0)
DBA_SEL
DBB_ADDR(2:0)
DBB_SEL

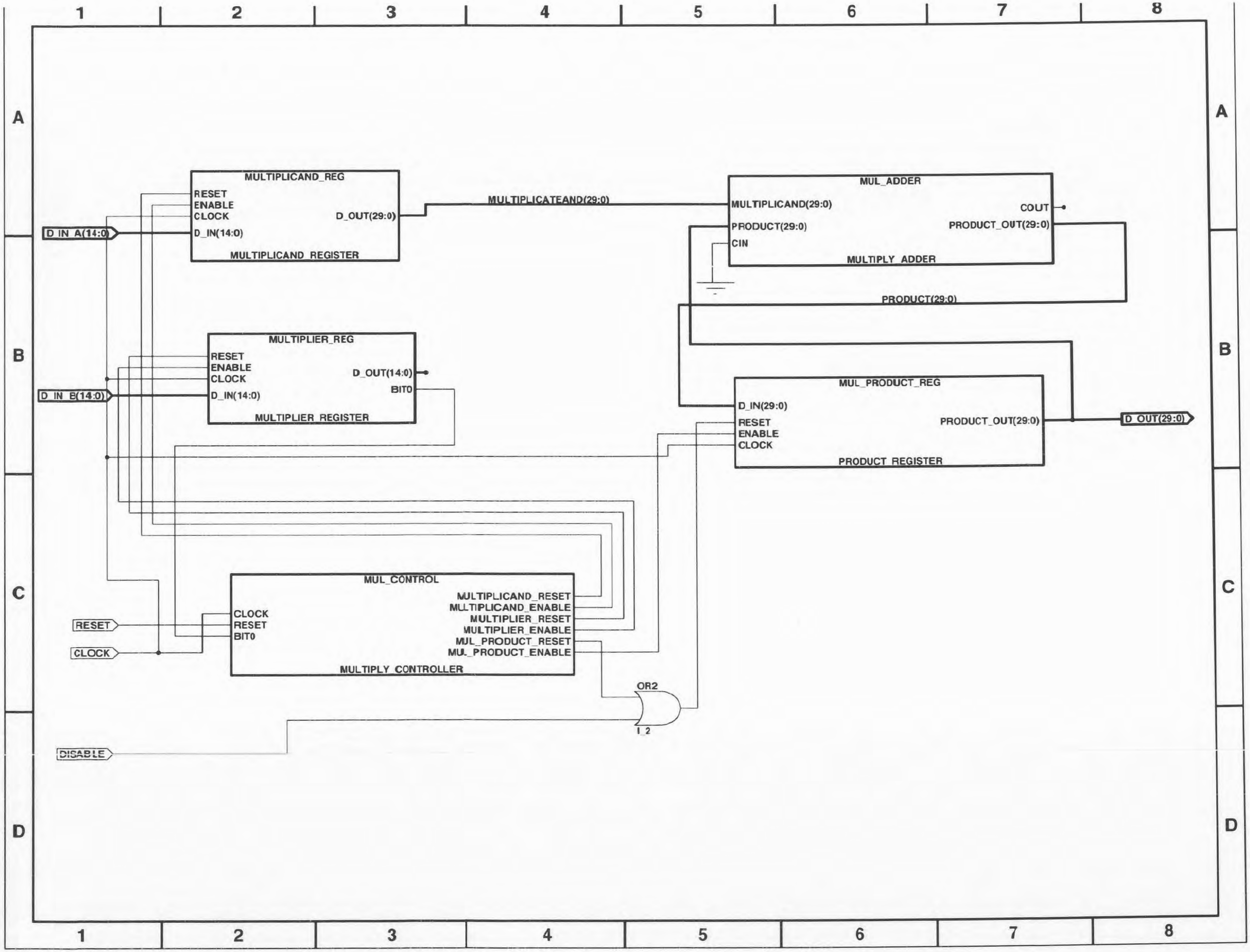

RETURN_EN
JMP
JA
F_POS
JE
F_ZERO
AND2
I_24
AND2
I_23
OR3
I_2
OR2
SUBR_VAL
I_22
IS_BRNCH
OR2
I_21
FEEDBACK4(15:0)
NEW_ADDRESS(15:0)
STACK16
D_IN(15:0)
PUSH
ENABLE
CLOCK
RESET
D_OUT(15:0)
JSR_STACK
SUBR_VAL
SUBR_EN
RET_VAL(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
HI_OUTSIDE_LO_STACK
BRANCHED(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
HI_BRANCH_LO_PC4
FULL_ADDER16
ADD_FOUR(15:0)
A(15:0)
B(15:0)
CIN
COUT
SUM(15:0)
NEXT_INSTRUCT_ADDRESS
NEXT_PC(15:0)
REGISTER16
D_IN(15:0)
PC_EN
CLOCK
RESET
ENABLE
CLOCK
RESET
D_OUT(15:0)
NEXT_ADDRESS
I_ADDRESS(15:0)
ADD_FOUR(15)
ADD_FOUR(14)
ADD_FOUR(13)
ADD_FOUR(12)
ADD_FOUR(11)
ADD_FOUR(10)
ADD_FOUR(9)
ADD_FOUR(8)
ADD_FOUR(7)
ADD_FOUR(6)
ADD_FOUR(5)
ADD_FOUR(4)
ADD_FOUR(3)
ADD_FOUR(2)
ADD_FOUR(1)
ADD_FOUR(0)

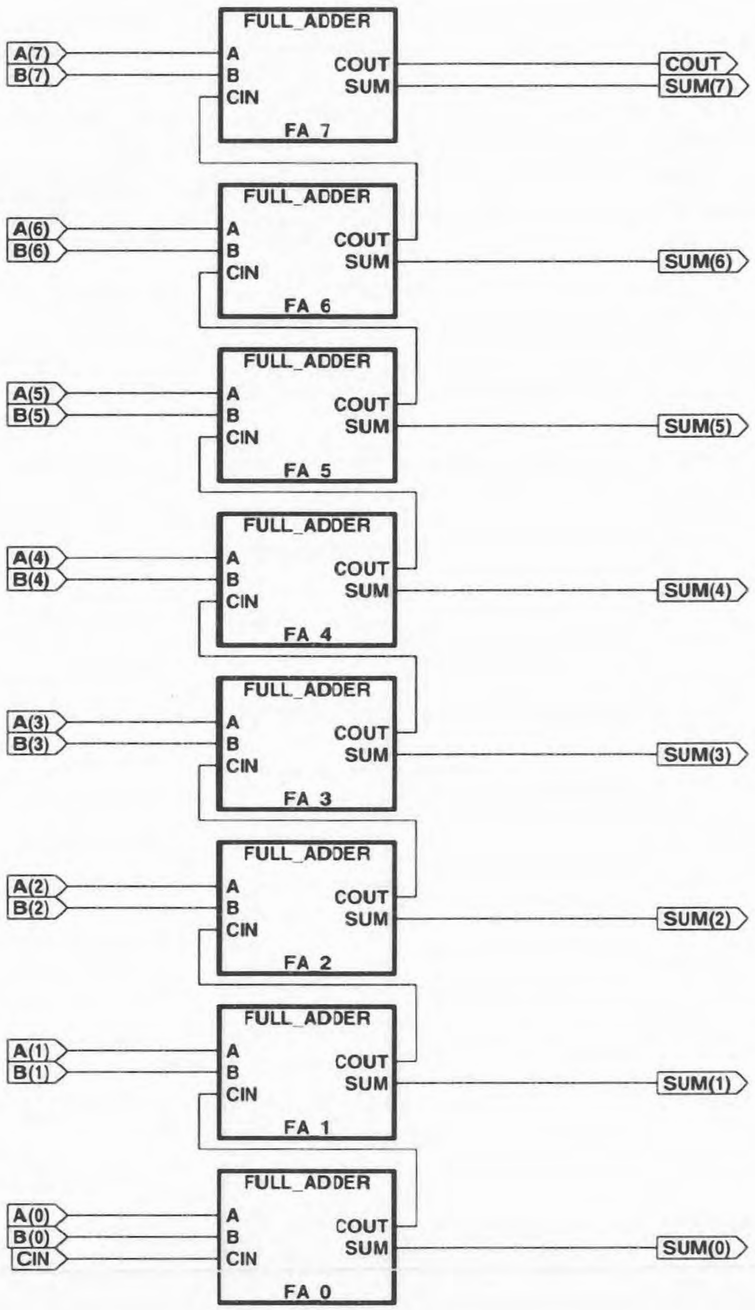

MUX_32BIT_2TO1
LOAD(31:0)
FIFO(31:0)
LOAD_EXT
S_HIGH(31:0)
S_LOW(31:0)
SEL
D_OUT(31:0)
MEMORY_SOURCE
SOURCE(31:0)
REGISTER32
D_IN(31:0)
ENABLE
CLOCK
RESET
D_OUT(31:0)
FIFO_OUT(31:0)
ENABLE
CLOCK
RESET

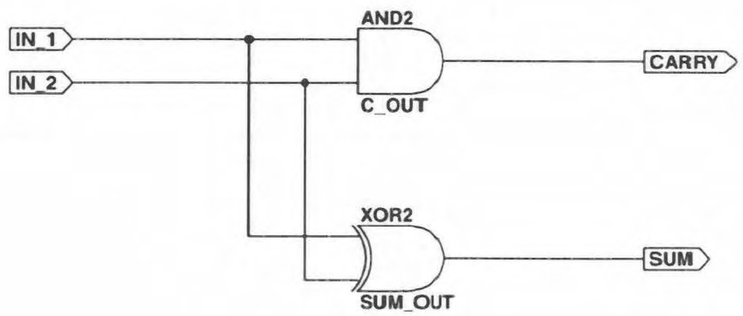

MUX2X1
S_LOW(7)
S_HIGH(7)
IN0
IN1
SEL
D_OUT(7)
A7
MUX2X1
S_LOW(6)
S_HIGH(6)
IN0
IN1
SEL
D_OUT(6)
A6
MUX2X1
S_LOW(5)
S_HIGH(5)
IN0
IN1
SEL
D_OUT(5)
A5
MUX2X1
S_LOW(4)
S_HIGH(4)
IN0
IN1
SEL
D_OUT(4)
A4
MUX2X1
S_LOW(3)
S_HIGH(3)
IN0
IN1
SEL
D_OUT(3)
A3
MUX2X1
S_LOW(2)
S_HIGH(2)
IN0
IN1
SEL
D_OUT(2)
A2
MUX2X1
S_LOW(1)
S_HIGH(1)
IN0
IN1
SEL
D_OUT(1)
A1
MUX2X1
S_LOW(0)
S_HIGH(0)
IN0
IN1
SEL
D_OUT(0)
A0
SEL

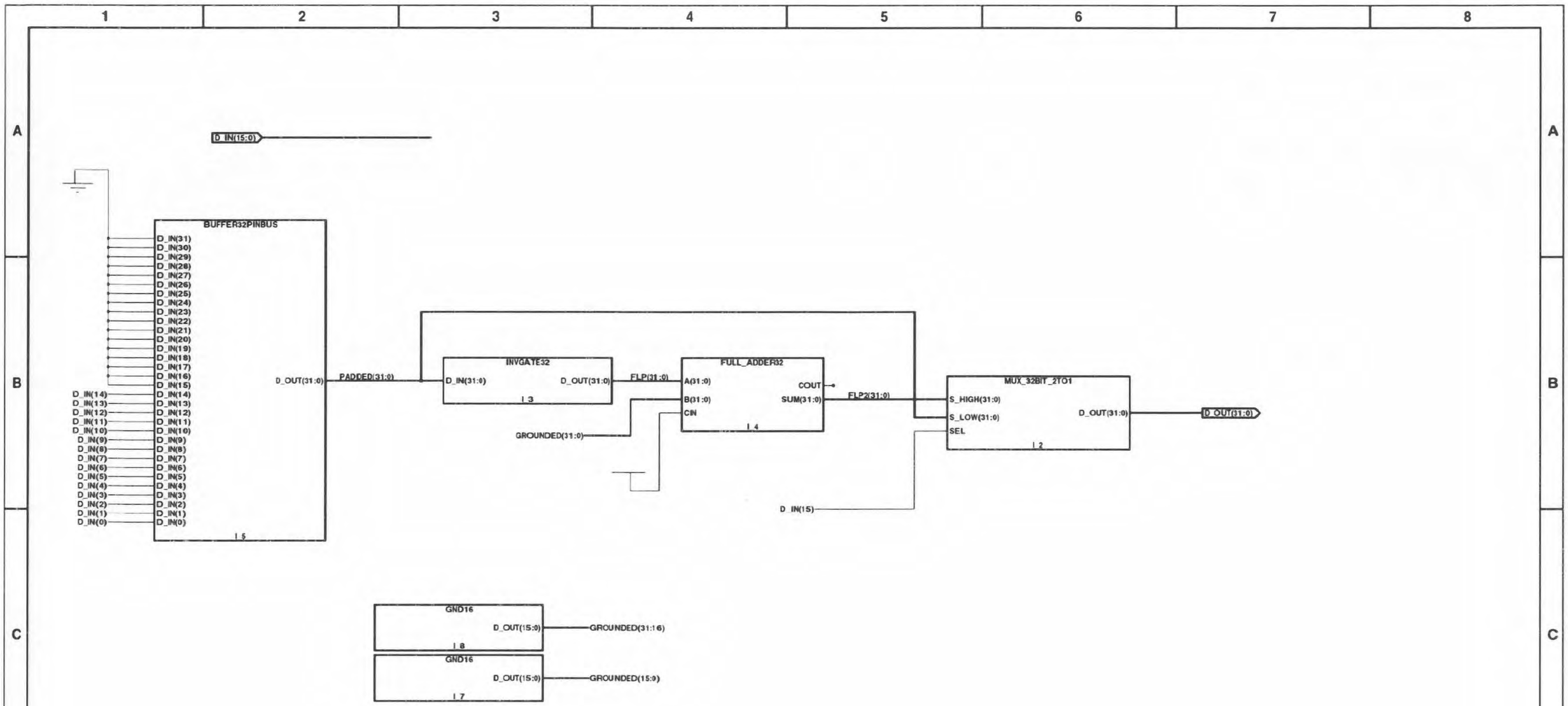

INVGATE16PINS
D_IN(15)
D_IN(14)
D_IN(13)
D_IN(12)
D_IN(11)
D_IN(10)
D_IN(9)
D_IN(8)
D_IN(7)
D_IN(6)
D_IN(5)
D_IN(4)
D_IN(3)
D_IN(2)
D_IN(1)
D_IN(0)
I_8
D_OUT(15:0)
TOP_TMP(15:0)
INVGATE16
D_IN(15:0)
D_OUT(15:0)
INV_LOWER_16
BOT_TMP(15:0)
TRISTATE_16
D_IN(15:0)
SEL
D_OUT(15:0)
NEG_UPPER_TRI
TRISTATE_16
D_IN(15:0)
SEL
D_OUT(15:0)
NEG_LOWER_TRI
D_OUT(31:16)
D_OUT(15:0)
BUFFER16PINBUS
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)
I_7
D_OUT(15:0)
D_IN(29)
D_IN(28)
D_IN(27)
D_IN(26)
D_IN(25)
D_IN(24)
D_IN(23)
D_IN(22)
D_IN(21)
D_IN(20)
D_IN(19)
D_IN(18)
D_IN(17)
D_IN(16)
D_IN(15:0)
SIGN
TOP_TMP2(15:0)
INVGATE
I_6
TRISTATE_16
D_IN(15:0)
SEL
D_OUT(15:0)
POS_UPPER_TRI
TRISTATE_16
D_IN(15:0)
SEL
D_OUT(15:0)
POS_LOWER_TRI

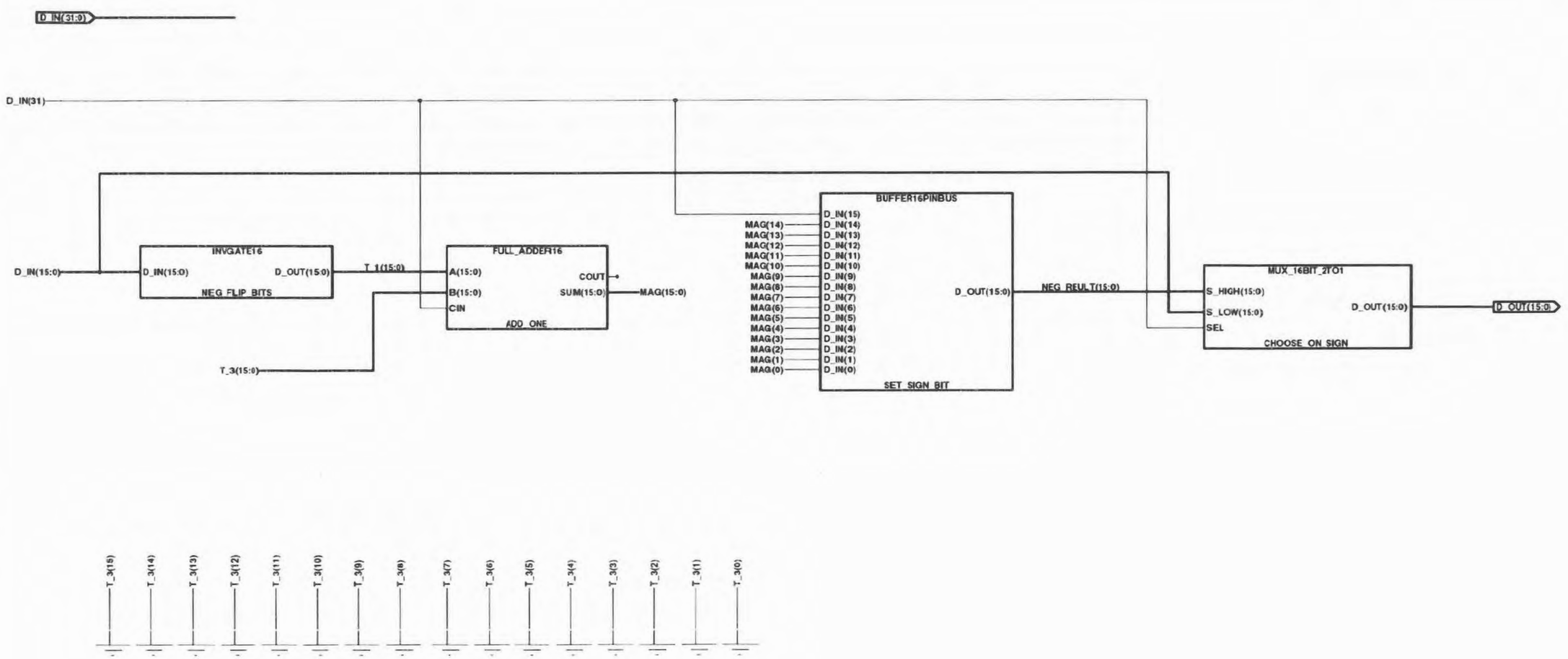

MULT_BIT_SPLIT
D_IN_A(15:0)
D_IN(15:0)
SIGN_BIT
D_OUT(14:0)
A_SPLITER
MULT_BIT_SPLIT
D_IN_B(15:0)
D_IN(15:0)
SIGN_BIT
D_OUT(14:0)
B_SPLITER
T_A(14:0)
T_B(14:0)
MULT_SM_15X15
D_IN_A(14:0)
D_IN_B(14:0)
RESET
CLOCK
DISABLE
D_OUT(29:0)
I_1
T_R(29:0)
MULT_SM_2S_CONV
D_IN(29:0)
SIGN
D_OUT(31:0)
CONVERT_FOR_ALU
MULTIPLY
RESET
CLOCK
XOR2
SIGN_BIT_XOR
SIGN
D_OUT(31:0)

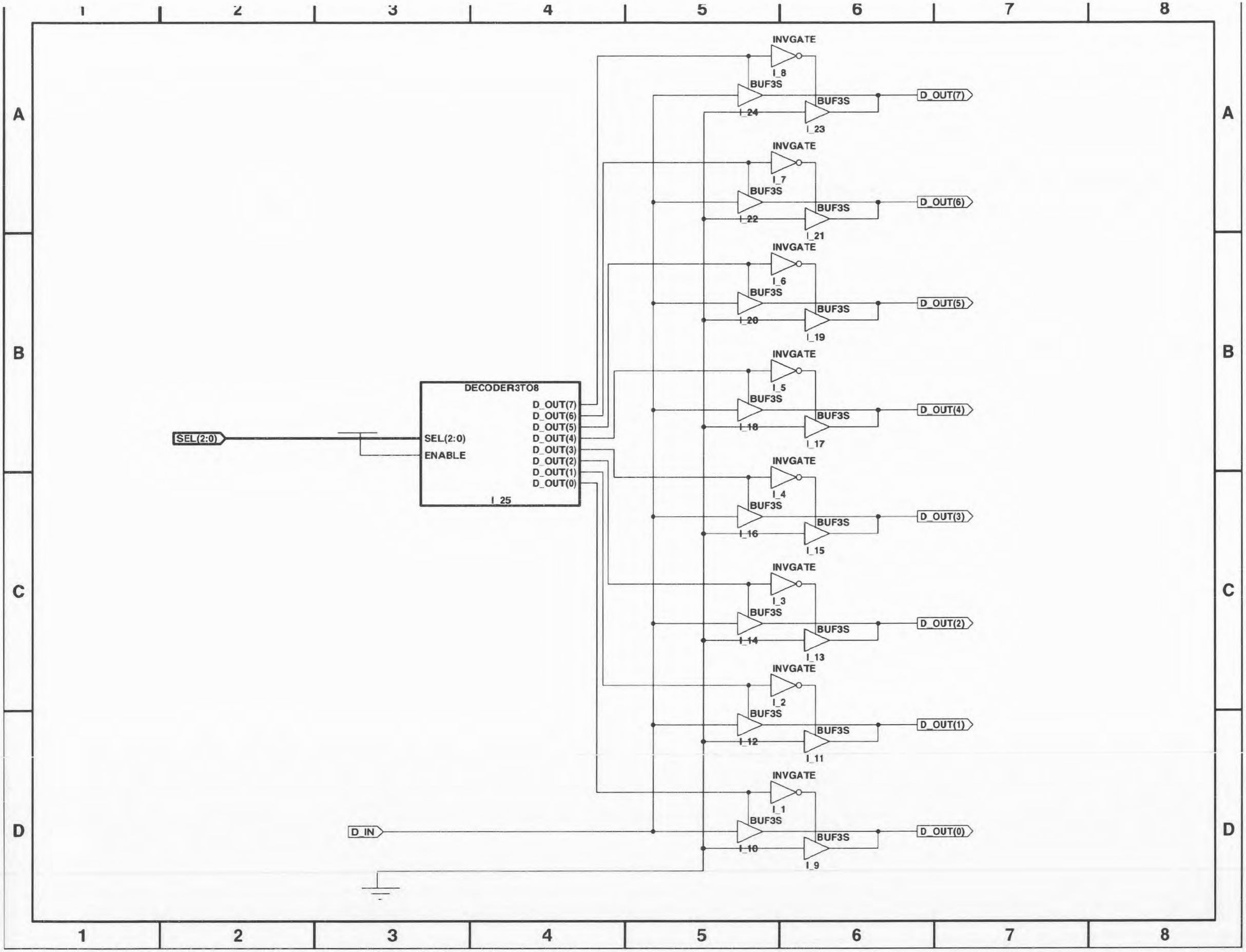

MULTIPLICAND_REG
RESET
ENABLE
CLOCK
D_IN(14:0)
D_OUT(29:0)
MULTIPLICAND_REGISTER
D_IN_A(14:0)
MULTIPLICATEAND(29:0)
MUL_ADDER
MULTIPLICAND(29:0)
PRODUCT(29:0)
CIN
COUT
PRODUCT_OUT(29:0)
MULTIPLY_ADDER
PRODUCT(29:0)
MULTIPLIER_REG
RESET
ENABLE
CLOCK
D_IN(14:0)
D_OUT(14:0)
BIT0
MULTIPLIER_REGISTER
D_IN_B(14:0)
MUL_PRODUCT_REG
D_IN(29:0)
RESET
ENABLE
CLOCK
PRODUCT_OUT(29:0)
PRODUCT_REGISTER
D_OUT(29:0)
MUL_CONTROL
CLOCK
RESET
BIT0
MULTIPLICAND_RESET
MULTIPLICAND_ENABLE
MULTIPLIER_RESET
MULTIPLIER_ENABLE
MUL_PRODUCT_RESET
MUL_PRODUCT_ENABLE
MULTIPLY_CONTROLLER
RESET
CLOCK
OR2
I_2
DISABLE

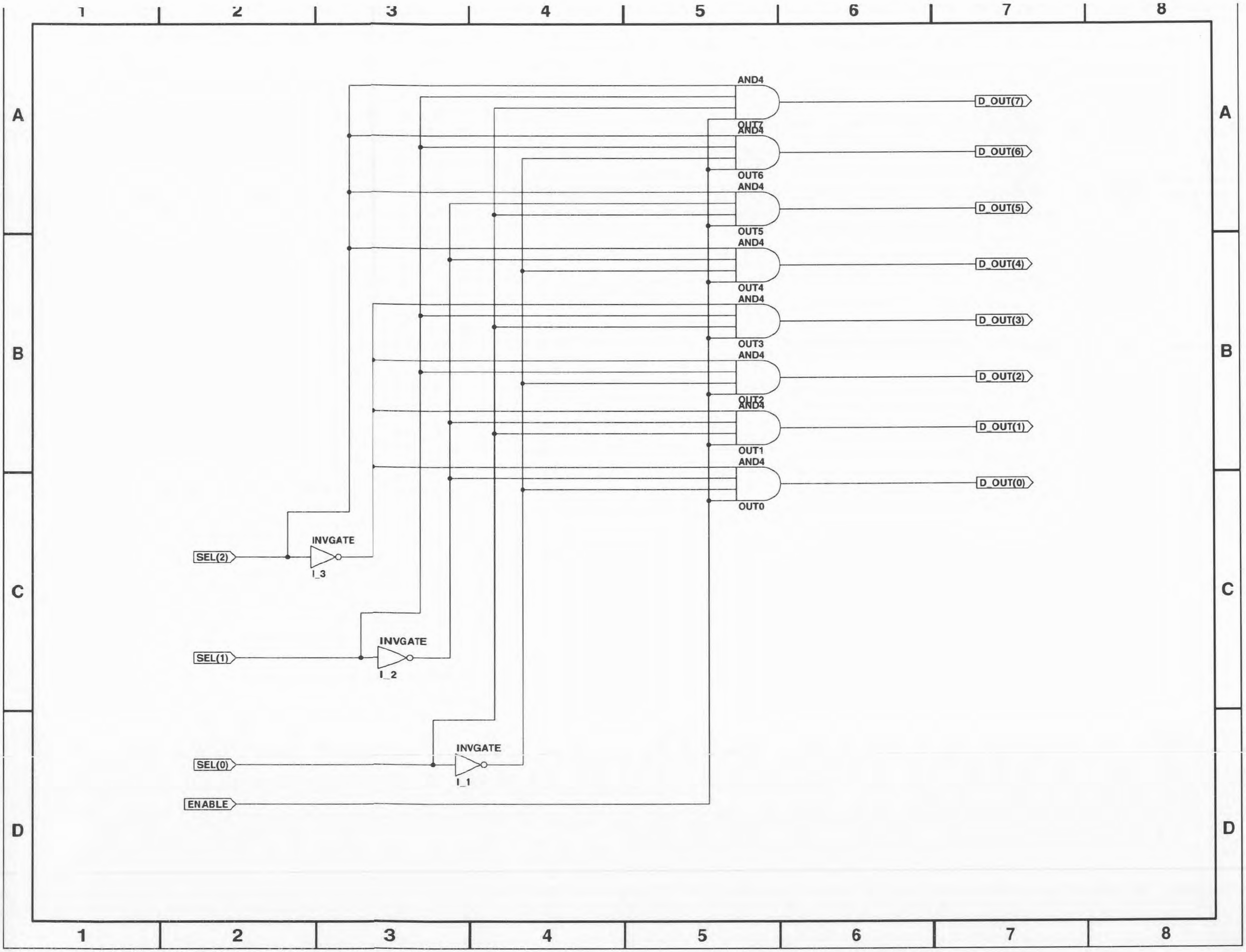

FULL_ADDER
A(7)
B(7)
A
B
CIN
COUT
SUM
COUT
SUM(7)
FA_7
FULL_ADDER
A(6)
B(6)
A
B
CIN
COUT
SUM
SUM(6)
FA_6
FULL_ADDER
A(5)
B(5)
A
B
CIN
COUT
SUM
SUM(5)
FA_5
FULL_ADDER
A(4)
B(4)
A
B
CIN
COUT
SUM
SUM(4)
FA_4
FULL_ADDER
A(3)
B(3)
A
B
CIN
COUT
SUM
SUM(3)
FA_3
FULL_ADDER
A(2)
B(2)
A
B
CIN
COUT
SUM
SUM(2)
FA_2
FULL_ADDER
A(1)
B(1)
A
B
CIN
COUT
SUM
SUM(1)
FA_1
FULL_ADDER
A(0)
B(0)
CIN
A
B
CIN
COUT
SUM
SUM(0)
FA_0

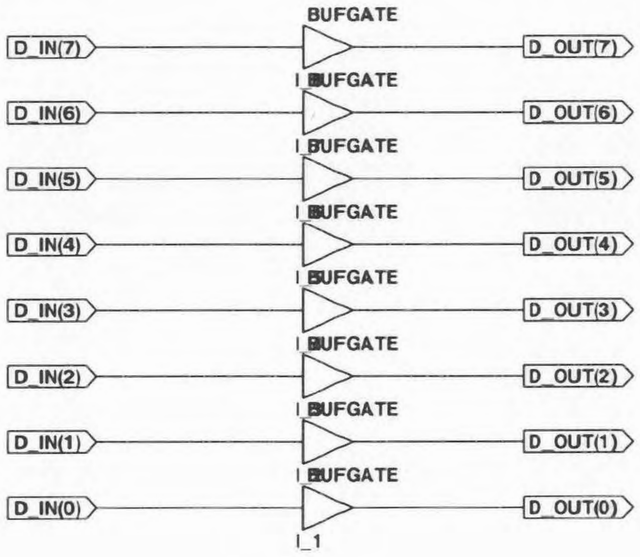

IN_1
IN_2
AND2
C_OUT
CARRY
XOR2
SUM_OUT
SUM


D_IN(15:0)
BUFFER32PINBUS
D_IN(31)
D_IN(30)
D_IN(29)
D_IN(28)
D_IN(27)
D_IN(26)
D_IN(25)
D_IN(24)
D_IN(23)
D_IN(22)
D_IN(21)
D_IN(20)
D_IN(19)
D_IN(18)
D_IN(17)
D_IN(16)
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)
D_OUT(31:0)
I_5
PADDED(31:0)
INVGATE32
D_IN(31:0)
D_OUT(31:0)
I_3
FLP(31:0)
FULL_ADDER32
A(31:0)
B(31:0)
CIN
COUT
SUM(31:0)
I_4
FLP2(31:0)
GROUNDED(31:0)
MUX_32BIT_2TO1
S_HIGH(31:0)
S_LOW(31:0)
SEL
D_OUT(31:0)
I_2
D_OUT(31:0)
D_IN(15)
GND16
D_OUT(15:0)
I_8
GROUNDED(31:16)
GND16
D_OUT(15:0)
I_7
GROUNDED(15:0)


D_IN(31:0)
D_IN(31)
D_IN(15:0)
INVGATE16
D_IN(15:0)
D_OUT(15:0)
NEG_FLIP_BITS
T_1(15:0)
T_3(15:0)
FULL_ADDER16
A(15:0)
B(15:0)
CIN
COUT
SUM(15:0)
ADD_ONE
MAG(15:0)
BUFFER16PINBUS
MAG(14)
MAG(13)
MAG(12)
MAG(11)
MAG(10)
MAG(9)
MAG(8)
MAG(7)
MAG(6)
MAG(5)
MAG(4)
MAG(3)
MAG(2)
MAG(1)
MAG(0)
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)
D_OUT(15:0)
SET_SIGN_BIT
NEG_REULT(15:0)
MUX_16BIT_2TO1
S_HIGH(15:0)
S_LOW(15:0)
SEL
D_OUT(15:0)
CHOOSE_ON_SIGN
D_OUT(15:0)
T_3(15)
T_3(14)
T_3(13)
T_3(12)
T_3(11)
T_3(10)
T_3(9)
T_3(8)
T_3(7)
T_3(6)
T_3(5)
T_3(4)
T_3(3)
T_3(2)
T_3(1)
T_3(0)


1
2
3
4
5
6
7
8
A
B
C
D
DECODER3TO8
SEL(2:0)
ENABLE
D_OUT(7)
D_OUT(6)
D_OUT(5)
D_OUT(4)
D_OUT(3)
D_OUT(2)
D_OUT(1)
D_OUT(0)
I_25
SEL(2:0)
D_IN
INVGATE
BUF3S
I_8
I_24
I_23
I_7
I_22
I_21
I_6
I_20
I_19
I_5
I_18
I_17
I_4
I_16
I_15
I_3
I_14
I_13
I_2
I_12
I_11
I_1
I_10
I_9


AND4
D_OUT(7)
OUT7
AND4
D_OUT(6)
OUT6
AND4
D_OUT(5)
OUT5
AND4
D_OUT(4)
OUT4
AND4
D_OUT(3)
OUT3
AND4
D_OUT(2)
OUT2
AND4
D_OUT(1)
OUT1
AND4
D_OUT(0)
OUT0
INVGATE
SEL(2)
I_3
INVGATE
SEL(1)
I_2
INVGATE
SEL(0)
I_1
ENABLE


BUFGATE
D_IN(7)
D_OUT(7)
BUFGATE
D_IN(6)
D_OUT(6)
BUFGATE
D_IN(5)
D_OUT(5)
BUFGATE
D_IN(4)
D_OUT(4)
BUFGATE
D_IN(3)
D_OUT(3)
BUFGATE
D_IN(2)
D_OUT(2)
BUFGATE
D_IN(1)
D_OUT(1)
BUFGATE
D_IN(0)
D_OUT(0)
I_1

Appendix E: 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.