

**Floating Point Multiply/Add Unit For the
M-Machine Node Processor**

by

Daniel K. Hartman

Submitted to the Department of Electrical Engineering and
Computer Science

in partial fulfillment of the requirements for the degrees of

Master of Engineering

and

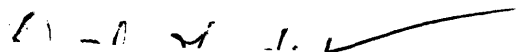
Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

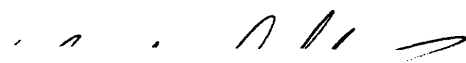
© Massachusetts Institute of Technology 1996. All rights reserved.



Author

Department of Electrical Engineering and Computer Science

May 18, 1996

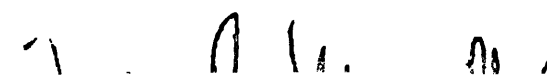


Certified by

William J. Dally

Associate Professor

Thesis Supervisor



Accepted by

F. R. Morgenthaler

Chairman, Departmental Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 11 1996

Barker Eng

**Floating Point Multiply/Add Unit For the M-Machine
Node Processor**

by

Daniel K. Hartman

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis covers the design of a floating-point multiplier/adder unit for the M-Machine node processor. The design includes the overall architecture, the algorithms used for multiplication/addition, complete schematics and general layout guidelines for the datapath, and the logic equations to control rounding.

Thesis Supervisor: William J. Dally
Title: Associate Professor

Acknowledgments

Without a doubt, the experience of being part of the M-Machine project has been the most valuable of my educational experience by far. The opportunity to do design work for a project such as this seems to be very rare; certainly I would have never had the opportunity to build an entire floating point unit in industry without years of experience. If anyone has doubts as to whether cutting processors can come out of academia, I believe this project will demonstrate it is possible.

A big thank you goes to Professor Bill Dally for giving me the opportunity to tackle such a big project, as well as giving lots of circuit advice and being an incredibly large bank of knowledge.

Special thanks goes to Steve Keckler for giving design advice, and generally looking over my shoulder.

I would also like to express my appreciation to the rest of the M-Machine hardware team, including:

Jeff Bowers UROP Student (Responsible for Bidirectional Pads and Floating Point Divide/Square-root Unit)

Andrew Chang Research Scientist (General Manager and many other things)

Nick Carter Phd. Student (Memory Subsystem Guru)

Parag Gupta MEng Student (Circuit Designer for many units)

Whay Lee Phd. Student (Network Interface Guru)

Working with a group of highly competent people is a reward of its own.

Contents

1	The M-Machine and its Floating Point Requirements	10
2	The Instruction Set	13
2.1	FMUL: Double Precision Multiply	14
2.2	FADD: Double Precision Addition	15
2.3	FMULA: Double Precision Multiply and Add	16
2.4	FTOI: Double Precision to Integer Conversion	17
2.5	ITOF: Integer to Double Precision Conversion	17
3	Hardware Needed to Support Operations	19
3.1	Double Precision Multiplication	19
3.1.1	Multiplication Techniques	20
3.1.2	Alternative Rounding Techniques	22
3.2	Double Precision Addition	24
3.2.1	Alternate rounding techniques	25
4	Possible Architectures for Multiply/Add	28
4.1	Separate Pipelines for Multiply and Add	28
4.2	Shared resources between Multiply and Add units	29
4.3	Multiply followed by Add	29
4.4	Multiply with no rounding followed by Add	32
4.5	Chosen Architecture for Implementation	33
5	Schematic Implementation	34

5.1	Pipeline Design	34
5.2	Multiplier Design	35
5.2.1	Booth Encoding	35
5.2.2	Multiplication Arrays	36
5.2.3	Timing Considerations	39
5.2.4	Array Combination	42
5.2.5	Multiplier Rounding	42
5.3	Addition	43
5.3.1	Alignment Shifter	43
5.3.2	Addition and Rounding	44
5.3.3	Post-normalization	45
5.4	Support for other operations	45
5.4.1	Immediate Support	45
5.4.2	Send Operations	45
5.4.3	Err-vals	46
5.4.4	Conversion Operations	46
6	Circuit Design for the Multiplier Array	47
6.1	Evaluation	53
7	Results	55
8	Conclusion	60
8.1	Lessons Learned	60
A	IEEE Double Precision Format	62
B	An Alternative Multiplication Scheme for Eliminating Sign Extension	64
C	Draft HDCVSL Logic Paper for JSSC	67
C.1	Differential Logic Families	68
C.2	Analysis	69

C.3 XOR Folding	70
C.4 Conclusion	71
D Schematics	74
D.1 Immediate Path and Front End	75
D.2 Stage 0	77
D.2.1 Booth Encoding Cells	78
D.3 Stage 2	83
D.3.1 Multiplier Array and Subcells	84
D.3.2 Array Combination and Subcells	104
D.4 Stage 3	112
D.5 Stage 4	113
D.5.1 Shift Sticky (Mask Generator and subcells)	114
D.6 Stage 6	124
D.6.1 Stage 6 Subcells	126
D.7 Stage 7	129
D.8 Send Unit	130
D.9 Drivers and Misc Cells	131
D.10 Shared Library Cells Used	134
D.10.1 Latches	134
D.10.2 Muxes	139
D.10.3 Leading Zero Detector	141
D.10.4 Shifters	146
D.10.5 1 bit Adders	157
D.10.6 Zero Detector	158
D.10.7 Adders	162

List of Figures

1-1	<i>Floating Point Pipeline</i>	11
4-1	<i>Separate Multiplier and Adder Units</i>	29
4-2	<i>Shared Add/Round Logic</i>	30
4-3	<i>Shared Add/Round and Shifter Logic</i>	31
4-4	<i>Fused Multiply/Add Units sharing the shifter</i>	32
4-5	<i>Fused Multiply/Add Units with no separate Multiplier rounding</i>	33
5-1	<i>Booth Encoder Cell - Top Level</i>	37
5-2	<i>Booth Encoder Cell - XOR Inputs</i>	37
5-3	<i>Booth Encoder Cell - Select 1,3</i>	38
5-4	<i>Booth Encoder Cell - Select 0,2,4</i>	38
5-5	<i>Timing for Evaluate Signals for Domino Logic</i>	40
5-6	<i>Interface Latch for static logic from domino signals</i>	41
5-7	<i>Array Combination using 3 stages of Full Adders</i>	42
6-1	<i>Static CMOS Full Adder (Output Buffers not shown)</i>	48
6-2	<i>Pass Transistor Full Adder (Output Buffers not shown)</i>	49
6-3	<i>Differential Cascode Voltage Switch Logic Full Adder</i>	49
6-4	<i>Hybrid DCVSL Full Adder</i>	50
6-5	<i>Differential Domino Full Adder</i>	51
6-6	<i>Example of a circuit where charge sharing can occur</i>	52
7-1	<i>Current Floorplan for the Multiple ALU Processor</i>	59

C-1	<i>2 Input XOR gates for DCMOS, DCVSL, and HDCVSL families . . .</i>	72
C-2	<i>Folded 2 Input XOR gates for DCMOS, DCVSL, and HDCVSL families</i>	73

List of Tables

3.1	<i>3-2 Adder</i>	21
3.2	<i>Radix 4 Booth Encoding Example</i>	22
3.3	<i>Multiplication Rounding Detection</i>	23
3.4	<i>Possible Rounding Results</i>	25
3.5	<i>Post-rounding Shift Detection</i>	26
3.6	<i>Effect of rounding to the right of S bit</i>	27
5.1	<i>Booth Encoding Outputs</i>	36
6.1	Test Vectors for Static Logic	54
7.1	<i>Critical Path Timing</i>	56
A.1	<i>IEEE Double Precision Examples</i>	63
B.1	<i>Typical Full Adder</i>	65
B.2	<i>First Recoded Adder</i>	65
B.3	<i>Second Recoded Adder</i>	66
C.1	<i>Results for 2 Input XOR Comparison</i>	69
C.2	<i>Results for Folded 2 Input XOR Comparison</i>	70

Chapter 1

The M-Machine and its Floating Point Requirements

The M-Machine project [3] is an attempt to explore various ways of utilizing parallelism at different granularities, using custom processors hooked up in a 3-D mesh network. Each node of the network contains 8 megabytes of local memory and a Multiple ALU Processor (designated the MAP chip).

The MAP chip is designed to exploit finer grain parallelism while still providing performance at a level roughly equivalent to today's microprocessors. Each map chip contains 4 *clusters*, each of which is roughly analogous to a RISC uP core. The clusters each contain an instruction cache, a register file, an integer unit, a memory unit ¹, and a floating-point unit. Other features include:

- 100mhz target clock frequency
- 128Kbyte on-chip L1 cache
- Support for multi-threaded operation with zero-time switching penalty
- Network router and communication ports on-chip

¹The memory unit can execute many of the operations that the integer unit can, along with load and store instructions

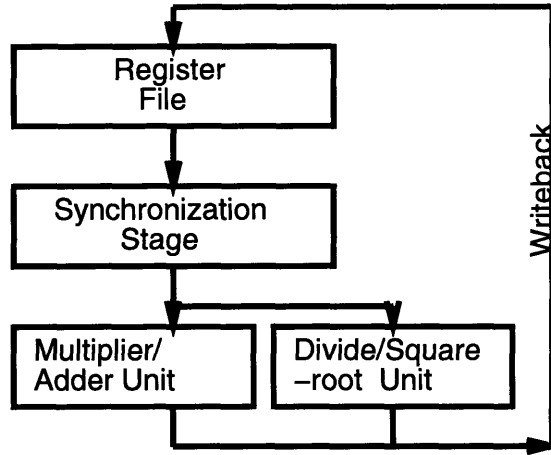


Figure 1-1: *Floating Point Pipeline*

The floating point unit consists of a 15 word by 64 bit register file, a multiplier/adder unit, and a divide/square-root unit. The pipeline model for the floating point unit is shown in figure 1-1. The first stage of the pipeline consists of a register read from the register file. Next, the synchronization stage ensures all operands are valid ². Then the operands are dispatched to either the multiplier/adder unit or the divide/square-root unit. Finally, output values from the two units are written back to the register file.

My work concentrated on the datapath portion of the multiplier/adder unit (designated the FPU-MULA unit), along with the rounding equations for the control logic. Because of its position in the global design, there were many design constraints on the multiplier/adder:

- Support for IEEE double-precision multiplication, addition, multiply-and-add, comparison, and conversion operations.
- Target clock period of 10ns
- Fully pipelined with target latency of 4 clock cycles

²This is accomplished using a scoreboard in the control logic

- Fixed Bit width of 25.2 microns (total 2mm width) and approximate total height of 3.8mm
- Ability to stall pipeline and bypass stages.

Chapter 2

The Instruction Set

The MAP Instruction set is outlined in detail in M-Machine Instruction Set Reference v1.4 [4]. Only a small subset of these instructions is executed through the FMULA unit; these instructions are

FADD Floating Point Addition

FSUB Floating Point Subtraction

FMUL Floating Point Multiply

FMULA Floating Point Multiply and Add

IMUL Integer Multiply (low word)

HMUL Integer Multiply (high word)

MOV Move Register

ITOF Signed Integer to Floating Point Conversion

FTOI Floating Point to Signed Integer Conversion

FTOIU Floating Point to Unsigned Integer Conversion

FLT Floating Point Less Than

FLE Floating Point Less Than or Equal

FEQ Floating Point Equal

FNE Floating Point Not Equal

FIMM Create 16 Bit Immediate

FSHORU Shift & OR Unsigned 16 Bit Immediate

FSND0 Send Priority 0 Message, user level

FSND0O Send Priority 0 Message, ordered, user level

FSND0P Send Priority 0 Message, physical

FSND0PO Send Priority 0 Message, physical, ordered

FSND0PNT Send Priority 0 Message, physical, no throttling

FSND0PNT0 Send Priority 0 Message, physical, no throttling, ordered

FSND1PNT Send Priority 1 Message, physical, no throttling

FSND1PNT0 Send Priority 1 Message, physical, no throttling, ordered

Many of these instructions are simply variants of each other, with simple control-side changes. Therefore, the datapath has been designed to accommodate the 12 *unique* instructions: FADD, FMUL, FMULA, IMUL, HMUL, MOV, ITOF, FTOI, FTOIU, FIMM, SHORU ¹.

2.1 FMUL: Double Precision Multiply

This operation performs an IEEE compliant double precision ² multiplication on two input arguments, A and B. The pseudo-code for the operation looks like:

1. $\text{res.exp} = \text{a.exp} + \text{b.exp}$;

¹The Send instructions are implemented using a send unit directly copied from the Integer unit. See [5] for details

²See Appendix A for a short discussion of the IEEE double precision floating point format

2. `res.mant = a.mant * b.mant;`
3. `if (res.mant >= 2) res.mant=res.mant >> 1, res.exp++;`
4. `if (res.mant[-1] == 1) res.mant+= lsb;`
5. `if (res.mant[-2:-n] == 0) res.mant[0] = 0;`
6. `if (res.mant >= 2) res .mant=res.mant >> 1, res.exp++;`

To summarize: Add the exponents, multiply the mantissa's. If the mantissa overflows, right-shift mantissa by 1 to normalize, and increment exponent. If the bit 1 to the right of the LSB of the mantissa is 1, increment mantissa (fractional portion $\geq .5$). If all the bits more than 1 to the right of the LSB of the mantissa are 0, force LSB of mantissa to be 0 (when fractional part $= .5$, round to nearest even). Finally, check mantissa for overflow, if so right-shift mantissa and increment exponent.

Checks can be easily done in parallel for invalid input arguments and overflow conditions. Gradual underflow conditions are a bit more complicated; the calculation is done on the underflowed input value as normal ³, and the final result is normalized as necessary. This ends up being an additional two steps: right-shift if result exponent is too small, and left-shift if mantissa underflows but exponent is larger than minimum value ⁴.

1. `while (res.exp < minval) res.exp++, res.mant >> 1;`
2. `while ((res.exp > minval) && (res.mant < 1)) res.exp--, res.mant << 1;`

2.2 FADD: Double Precision Addition

This operation performs an IEEE-compliant double precision addition on two input arguments, A and B. The pseudo-code for the operation looks like:

1. `if (a.exp > b.exp) less=b,more=a else less=a,more=b;`

³Some implementations, such as the SPARC, normalize the input values before calculation

⁴The minimum exponent is -1022

2. `res.exp = more.exp;`
3. `while (less.exp < more.exp) less.exp++,less.mant=less.mant >> 1;`
4. `res.mant = less.mant + more.mant;`
5. `while ((res.mant < 1) && (res.exp > minexp)) res.mant=res.mant << 1, res.exp-`
`;`
6. `if (res.mant >= 2) res.mant=res.mant >> 1, res.exp++;`
7. `if (res.mant[-1] == 1) res.mant++;`
8. `if (res.mant[-2:n] == 0) res.mant[0] = 0;`
9. `if (res.mant >= 2) res.mant=res.mant >> 1, res.exp++;`

The pseudo-code here first determines which of the arguments has the larger and smaller exponents; the result will have the larger exponent. The smaller number is then right shifted until the binary points are aligned. When this is achieved, the mantissas are added together. Then the result is normalized, and overflow is checked. Rounding is done as in the multiplication operation, and overflow is again checked after rounding.

Overflow and illegal numbers are again handled by the control logic. Gradual underflow is also dealt with correctly - underflowed inputs will work correctly, and the post-normalization stage will not normalize past the underflow mark.

2.3 FMULA: Double Precision Multiply and Add

This operation performs an IEEE-compliant double precision multiplication on two input arguments, A and B, than an IEEE-compliant double precision addition on the result of the multiplication and a third input argument C. This result of this operation should be indistinguishable from a separate FMUL and FADD instruction pair.

2.4 FTOI: Double Precision to Integer Conversion

This operation converts from the double-precision floating point format to a 2's complement integer result. The pseudo-code for the operation looks like:

1. `res.exp = a.exp - 52;`
2. `res.mant = a.mant;`
3. `while (res.exp < 0) res.exp++,res.mant=res.mant >> 1;`
4. `while (res.exp > 0) res.exp--,res.mant=res.mant << 1;`

The exponent is first corrected for the shift in the binary point, and the mantissa is copied. If the resultant exponent is negative, right-shift mantissa and increment exponent until 0. If the exponent is positive, left-shift mantissa and decrement exponent until 0. If one desires the conversion to be rounded as well, the following steps can be added (which are equivalent to the rounding in the addition and multiplication, except with no overflow correct).

1. `if (res.mant[-1] == 1) res.mant++;`
2. `if (res.mant[-2:n] == 0) res.mant[0] = 0;`

2.5 ITOF: Integer to Double Precision Conversion

This operation converts from a 2's complement integer to a double-precision floating point result. This pseudo-code for the operation looks like:

1. `res.exp = 52;`
2. `res.mant = a.mant;`

3. while (res.mant[63:53] > 0) res.exp++, res.mant=res.mant >> 1;

4. while (res.mant < 1) res.exp--, res.mant=res.mant << 1;

The exponent is first set to 52, on the assumption that the mantissa is correctly normalized from the given integer argument. If the high bits of the mantissa are non-zero (the bits beyond bit 52), the mantissa is right-shifted and the exponent is incremented. If the mantissa is not normalized, the mantissa is left-shifted and the exponent decremented.

Chapter 3

Hardware Needed to Support Operations

3.1 Double Precision Multiplication

The pseudo-code, and what's needed to implement it:

res.exp = a.exp + b.exp; Exponent calculation will be handled in the control logic.

res.mant = a.mant * b.mant; Mantissa multiplication will require a multiplication array of some sort; this is discussed below.

if (res.mant >= 2) res.mant=res.mant >> 1, res.exp++; This requires a single mux to select between the mantissa and the mantissa right-shifted by 1 place.

if (res.mant<-1> == 1) res.mant+= lsb; This requires an incrementer and a mux to select between the mantissa and the incremented mantissa.

if (res.mant<-2:-n> == 0) res.mant[0 = 0;] A single bit mux is needed to select between bit 0 and 0.

if (res.mant >= 2) res.mant=res.mant >> 1, res.exp++; This requires a single mux to select between the mantissa and the mantissa right-shifted by 1 place.

3.1.1 Multiplication Techniques

The most obvious algorithm for doing multiplication is that of the long-multiplication (similar to multiplying in decimal). Here, one adds a weighted version of the multiplicand whenever a bit in the multiplier is one. A simple example of a 4x4 bit multiply is shown below.

```

    0110 = 6
x   0111 = 7
-----
00000110 = 6
00001100 = 12
00011000 = 24
00000000 = 0
-----
00101010 = 42

```

To handle 2's complement signed arithmetic, things become a bit rough. Both the multiplicand and the multiplier need to be sign extended out to the full number of bits. Two examples are shown below.

```

    1101 = -3      0010 = 2
x   0110 = 6      x   1101 = -3
-----          -----
00000000 = 0      00000010 = 2
11111010 = -6     00000000 = 0
11110100 = -12    00001000 = 8
00000000 = 0      00010000 = 16
00000000 = 0      00100000 = 16
00000000 = 0      01000000 = 16
00000000 = 0      10000000 = 16
00000000 = 0      00000000 = 16

```

In0	In1	In2	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3.1: 3-2 Adder

----- -----
11101110 = -18 11111010 = -6

For a 64x64 bit multiply, this results in 128 partial products of 128 bits each. 128 adders, each 128 bits wide would take a considerable amount of silicon area to implement, as well as have a very long latency, both of which are unacceptable in this application.

One way around this is to not calculate the complete sum at every stage, but instead to calculate a redundant carry/sum form. Basically, at each bit, 3 bits will be summed together, forming two output bits, the carry and the sum (a 2 bit result) – see table 3.1

These outputs will be fed into the next stage (the sum bits being shifted right by 1), which will add another bit into the running sum and continue on. After all partial products have been added together, a 128 bit adder will be used to add the final two carry/sum numbers together.

This still leaves 128 stages of logic to pass through, giving quite a significant area and delay. One solution is to go to a higher radix at each stage. For radix n, each stage will add a precomputed multiple between 0 and n-1 of the multiplier, depending on log₂n bits of the multiplicand. For example, radix-4 will select between 0, 1x, 2x, and 3x the multiplicand, using 2 bits of the multiplier, and reducing the number of stages to 64.

Another technique is to again use redundant encoding, this time encoding the multiplier bits. Booth encoding uses log₂n bits of the multiplier at each stage, and

Bit 1	Bit 0	Last MSB	Total	Next Stage	Last Stage	Final
0	0	0	0	0	0	0
0	0	1	1	0	-1	0
0	1	0	1	0	0	+1
0	1	1	2	0	-1	+1
1	0	0	-2	+4	0	+2
1	0	1	-1	+4	-1	+2
1	1	0	-1	+4	0	+3
1	1	1	0	+4	-1	+3

Table 3.2: Radix 4 Booth Encoding Example

selects a multiple of the multiplicand between $-n+1$ and $n-1$. Negative multiplicands are easy to form from their positive counterpart¹, so only half the number of multiplicands need to be kept around. In addition, the sign extension bits of the multiplier can be tossed away, halving the number of partial products.

Booth encoding allows an error of up to $1/2 * \log_2 n$ that can be tolerated, provided it is corrected for in the next stage. This is accomplished by examining the MSB of the previous stage, and planing on the next stage using the current MSB. By assuming the next stage will add $\log_2 n * \text{the multiplicand}$ is the MSB is 1, we can subtract this from the current expected multiplicand. An example table for radix 4 is shown in table 3.2.

The first two columns represent the bits that are being used. Column 3 is the MSB from the last stage, which will need to be corrected for. Column 4 shows what multiple to add in for this stage. Columns 5 and 6 show what the next and previous stage will be adding in. Column 7 shows the final answer with all correction factors, indicating the correct expected value from just columns 1 and 2.

3.1.2 Alternative Rounding Techniques

The pseudo-code for multiplication has two adds occuring. The first add occurs after the multiplier array in order to produce the 128 bit result from the redundant carry/sum result. The second add occurs in the rounding stage to increment the

¹Invert and add 1 at the LSB

Rs	Rc	Rin	Ov	Rnd	Result	Pre-Incr	Final
0	0	0	0	1	1	0	1
0	0	0	1	1	2	0	2
0	0	1	0	1	2	0	2
0	0	1	1	1	3	0	3
0	1	0	0	1	2	1	0
0	1	0	1	1	3	1	1
0	1	1	0	1	3	1	1
0	1	1	1	1	4	1	2
1	0	0	0	1	2	1	0
1	0	0	1	1	3	1	1
1	0	1	0	1	3	1	1
1	0	1	1	1	4	1	2
1	1	0	0	1	3	1	1
1	1	0	1	1	4	1	2
1	1	1	0	1	4	1	2
1	1	1	1	1	5	1	3

Table 3.3: *Multiplication Rounding Detection*

mantissa.

By using a somewhat different strategy [8], the two adds can be combined. As noted above, rounding can be accomplished by adding 1 to the bit to the right of the LSB of the result mantissa, henceforth called the R bit. The other factors in calculating this bit are the carry/sum inputs (Rs and Rc), and the carry-in bit (Rin). Finally, if the mantissa overflows, the correct place to add the rounding 1 is the LSB; this can be accomplished by adding an additional 1 to the R position. Table 3.3 illustrates the possible inputs to the R position, and the resultant R bit.

The result bit shows that the possible values of the R bit will be in the range of 1-5. This will result of overflows into the LSB bit of 0, 1, or 2. By calculating mant, mant+1, and mant+2 in parallel, one can do select the proper result at the end, and avoid having two adds sequentially. However, calculating three possible results is computationally expensive.

With a simple technique, one can reduce the range of possible answers by observing that Rs and Rc will be calculated early. By taking the logical ‘or’ of this value, and adding it to the LSB position, the final range is reduced to 0-3, as shown in the last two columns. Now, the possible final results are mant and mant+1; by a

simple modification of the 64 bit adder, one can calculate mant and mant+1 with considerably less than the area of two 64 bit adders.

This strategy only implements IEEE round-to-nearest; if the fractional part is less than .5, round down, if greater than or equal to .5, round up. The preferred rounding style is IEEE round-to-nearest-even; here rounding when the fractional part is .5 will occur in the direction of the closest even number ².

To implement round-to-nearest-even, the only change necessary is the ability to force the LSB to 0. The reason this is sufficient is that the two rounding schemes differ ONLY when the fractional part is .5 and the integer part has an LSB of 0. Round-to-nearest will round up, resulting in a integer part with LSB of 1 (but no carry into the next significant bit), round to nearest will result in a integer part with LSB of 0.

To detect when the forcing of the LSB is necessary, the value of the bits beyond the the 1/2 bit must be considered. By performing a logical OR of all these bits (resulting in the sticky bit), one can determine if the fractional part is exactly .5 (sticky bit is 0), or >.5 (sticky bit is 1).

3.2 Double Precision Addition

The pseudo-code, and what's needed to implement it:

if (a.exp > b.exp) less=b,more=a else less=a,more=b; Two 2 input muxes
to select between mantissa.

res.exp = more.exp;

while (less.exp < more.exp) less.exp++,less.mant=less.mant >> 1; Right shifter
to shift mantissa

res.mant = less.mant + more.mant; 64 bit adder to add mantissa's together

²This is an arbitrary decision, the main incentive is to split which way rounding occurs more evenly. Whether this strategy is at all valid is beyond the scope of this thesis

Subtract Operation	Negative Result	Non-Rounded Result	Rounded Result
0	-	A+B	A+B+1
1	0	A+B+1	A+B+1+1 = A+B+2
1	1	(A+B+1)	(A+B+1)+1 = (A+B)

Table 3.4: *Possible Rounding Results*

while ((res.mant < 1) && (res.exp > minexp)) res.mant=res.mant << 1, res.exp-

Leading zero detector and left shifter to normalize mantissa.

if (res.mant >= 2) res.mant=res.mant >> 1, res.exp++; Mux to select between mantissa and right shifted mantissa

if (res.mant<-1> == 1) res.mant++; 64 bit incrementer to generate rounded version and 2 input mux to select incremented and non-incremented version.

if (res.mant<-2:n> == 0) res.mant[0 = 0;] 1 bit 2 input mux for forcing LSB to 0.

if (res.mant >= 2) res.mant=res.mant >> 1, res.exp++; Mux to select between mantissa and right shifted mantissa

3.2.1 Alternate rounding techniques

As in the multiplication algorithm, the add and round can be combined into one step. This technique consists of two separate parts; the first determines what the possible values from the addition are, the second selects between the non-rounded and rounded result.

The adder inputs will always be two positive mantissa values³. If the signs of the two input arguments vary, the operation to be done becomes a subtract. Subtraction is accomplished by inverting one of the arguments and adding 1. Finally, if a subtraction operation occurs, and the result is negative, the 2's complement of the result must be taken. Table 3.4 illustrates the possible operations.

³IEEE double precision format is sign-magnitude rather than 2's complement

Bit 53	Bit 52	Bit 51	Post-normalization	Effective R position	Round Vector
1	-	-	Right shift by 1 position	Bit 0 = L bit	1000
0	1	-	No Shift	Bit -1 = R bit	0100
0	0	1	Shift Left by 1	Bit -2 = G bit	0010
0	0	0	Shift Left by 2 or more	Bit -3 = S bit	0001

Table 3.5: *Post-rounding Shift Detection*

Not counting logical inverses, there are three distinct possible values: $A+B$, $A+B+1$, and $A+B+2$. By generating these three values, and selecting the correct two for the non-rounded and rounded cases, the rounding logic simply needs to determine which of the two remaining values to choose.

The problem remains of whether the final value needs to be incremented or not. Normally, three bits to the right of the LSB of the mantissa are kept track of. These bits, the R bit ($\text{mant}[-1]$), the G bit ($\text{mant}[-2]$) and the S bit ($\text{---mant}[-3:\text{inf}]$)⁴ keep additional precision. Because only one of the input arguments is shifted, the R, G, and S bits will come only from the shift⁵.

To determine rounding, one normally adds 1 to the R bit position and checks for overflow occurring. However rounding normally occurs after post-normalization, the R position may actually be a different position than where it is before post-normalization. To fold the rounding stage in with the addition, one needs to determine where the R position will be.

To do this, a 3 bit leading zero detect is done on the resultant mantissa. This will specify how much the result will be normalized; the possible cases are listed in the table 3.5.

The key point to note is that a shift of more than 2 to the right is equivalent to a shift of 2. This is because the S bit is copied when left shifting. When adding a rounding bit to any location right of the S bit, the exact same result will happen as if it were added to the S position, due to the copying of the S bit when shift occurs (see table 3.6). The only difference will be in the value of the bit at the rounded position,

⁴The S bit is actually the logical 'OR' of all the bits that are shifted beyond the G position

⁵The R,G, and S bits of the other input will always be 0, since IEEE double precision format does not contain these values

S bit	Round at S RGS	Rounded LRGS	Round right of S RGS...	Rounded RGS
0	000	001	000...00	000...01
1	111	000+	111...11	000...00+

Table 3.6: *Effect of rounding to the right of S bit*

which will not be truncated for the final result.

The other 'feature' of this form of rounding is that all bits right of the S bit, and including the s-bit, are guaranteed to be zero after rounding. Therefore, the normalization shifter can safely shift in 0's after the R and G positions, eliminating the high fanout from the S bit.

To accomplish rounding, first a 4-bit vector is formed from the R-G-S bits - this is equal to $4 * R + 2 * G + S$, or $R * (R) + 2 * (G) + S + 1$ in the case of a negative mantissa result when the two's complement. needs to be computed. To this vector is added the rounding vector, and the MSB of this 4 bit result is checked. A 1 here indicates an overflow into the LSB of the mantissa, and the rounded result should be selected.

```

bitvec = negative ? 4*( R ) + 2*( G ) + S + 1 : 4*R + 2*G + S;
bitvec += rndvec;
result = bitvec[3] ? rneded : nonrneded;

```

To accomplish round-to-nearest-even, forcing is done on the bit to the left of the rounding bit if the rounding bit position is 1, and all bits to the right are 0. This gives 4 different bit positions that could be forced, L1, L0, R, and G.

Chapter 4

Possible Architectures for Multiply/Add

Given the techniques discussed for implementing multiplication and addition, there are several possibilities for implementing a combined Multiply/Add instruction that calculates $A*B + C$.

4.1 Separate Pipelines for Multiply and Add

Figure 4-1 shows two independent pipelines compute the multiply and add separately. This has the advantage of giving minimal latency for the multiply instruction and the add instruction, but has a large latency for the multiply-add instruction, no better than that of a multiply followed by an add.

Given a super-scalar processor which can issue multiple instructions, this would have the advantage of being able to schedule multiply and add instructions simultaneously. Unfortunately, the M-Machine can only issue 1 floating point instruction per cycle.

Furthermore, there can be no sharing of resources needed for the multiply and add; this implementation uses the maximal amount of silicon area.

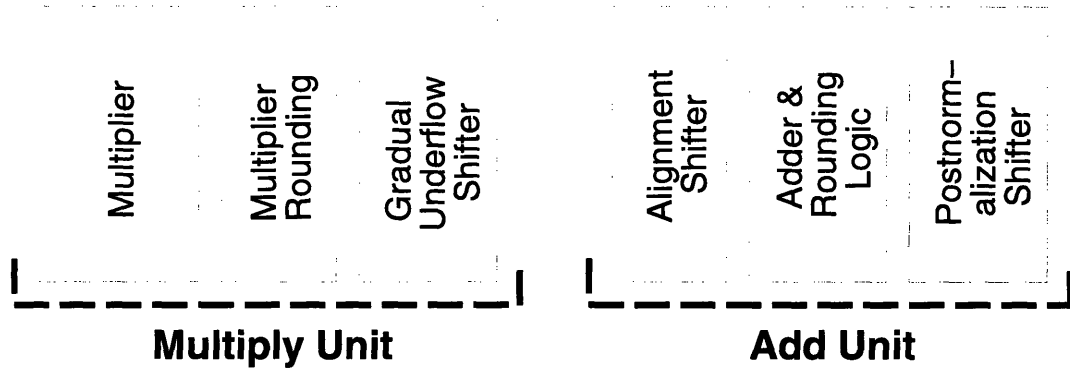


Figure 4-1: *Separate Multiplier and Adder Units*

4.2 Shared resources between Multiply and Add units

In the previous architecture, there is no resource sharing between the Multiply and Add units. However, one could easily share the adder and rounding logic between the multiplier and adder pipelines. The resultant pipeline is shown in figure 4-2.

By performing the gradual underflow shift *before* the multiply, the alignment shifter and gradual underflow shifter could be shared, as shown in figure 4-3.

Both of these approaches have the side effect of increasing the latency of the instructions due to the pipeline stages that don't quite overlap. Given a somewhat clever scheduling algorithm and the ability to feed-through stages, this effect could be reduced. However, the bigger problem is that the fused multiply-add is no longer a fully pipelined operation; two passes through the pipeline are necessary, and therefore multiply-add instructions can not be retired every cycle.

4.3 Multiply followed by Add

Figure 4-4 shows an architecture with a single multiply/add pipeline. Here, the gradual underflow shifter and the alignment shifter have been merged into one unit, otherwise the architecture is similar to the separate units placed end-to-end.

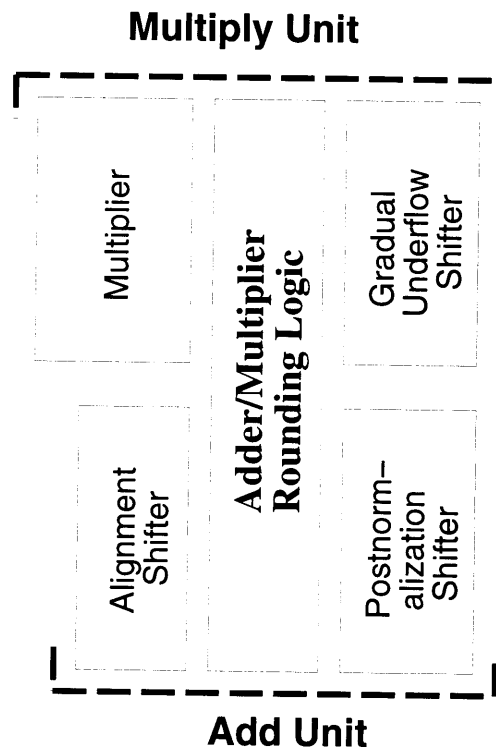


Figure 4-2: *Shared Add/Round Logic*

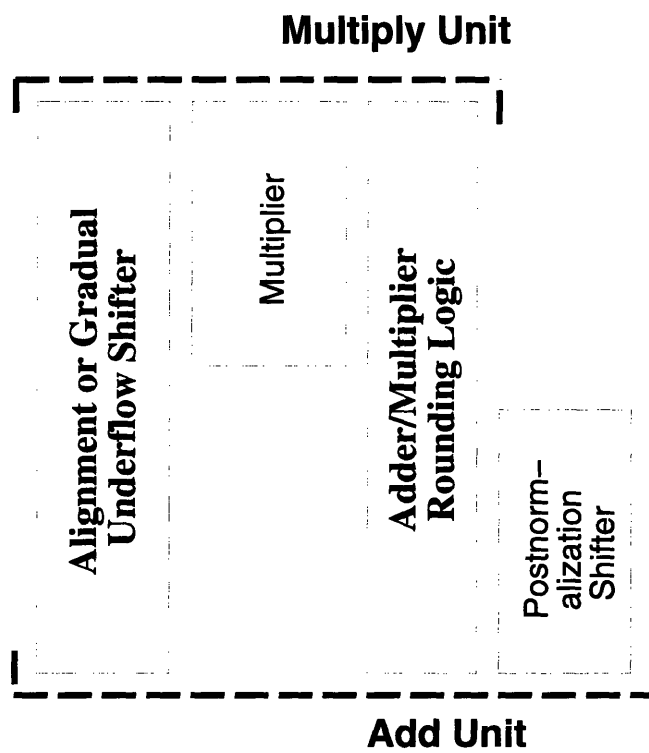


Figure 4-3: *Shared Add/Round and Shifter Logic*

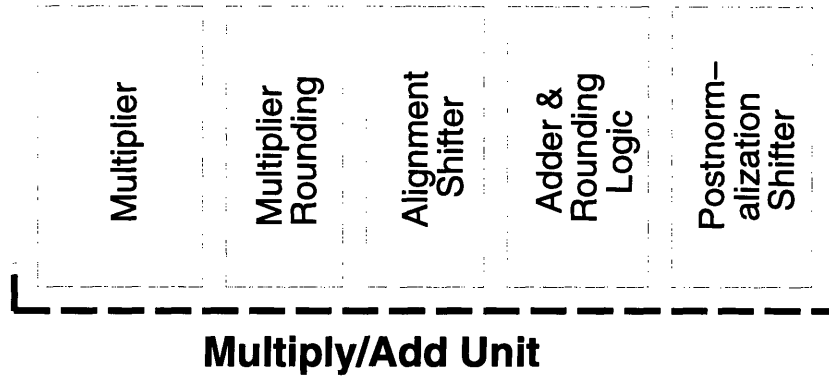


Figure 4-4: *Fused Multiply/Add Units sharing the shifter*

Fusing the two pipelines gives a smaller latency for the fused multiply-add operation, as well as some sharing of resources. However, the latency of both the multiply and the add operations has been increased to that of the full multiply-add. Some recovery of this time could be accomplished by allowing operations to drop-through the pipeline, though the scheduler must be careful to avoid two instructions retiring on the same cycle. In addition, the scheduler must be careful to avoid overlapping usage of the shared shifter.

4.4 Multiply with no rounding followed by Add

By removing the rounding stage from the multiplication operation and passing the UN-rounded sum and carry portions to the adder, the latency of the multiply-add operation can be further reduced (see figure 4-5). The main disadvantage of this over using the intermediate rounding stage is that the fused multiply/add instruction can give different results than the non-fused operations. Another disadvantage is that twice the number of wires need to be sent to the adder stage from the multiplier stage.

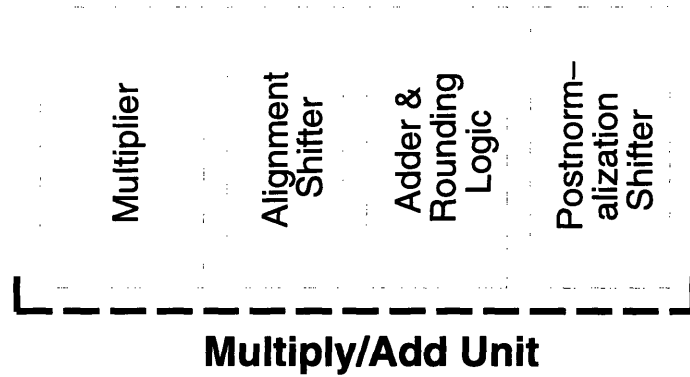


Figure 4-5: *Fused Multiply/Add Units with no separate Multiplier rounding*

4.5 Chosen Architecture for Implementation

The fused pipeline with intermediate rounding was chosen as the architecture for the floating point unit in the M-Machine.

The major reasons include:

- Ability to dispatch a fused multiply-add operation every clock cycle
- Relatively Low latency for the fused multiply-add operation
- Equivalent results from fused and non-fused multiply-add operations

Although other architectures have lower latencies for the fused multiply-add, they do don't give the same results as the non-fused operation, and therefore are unsatisfactory for this implementation.

Chapter 5

Schematic Implementation

5.1 Pipeline Design

The actual schematic implementation of the floating point multiplier-adder unit requires a 4 cycle pipeline divided into 8 stages each one-half cycle in length. These stages are:

1. Booth Encoding
2. Multiplication Arrays - First half
3. Multiplication Arrays - Second half
4. Multiplication Rounding
5. Addition Alignment - First half
6. Addition Alignment - Second half
7. Addition and Rounding
8. Post-normalization

Because of the use of transparent latches, stages can borrow time from their neighbors. The actual times for each stage, and the direction of time borrowing is shown below.

5.2 Multiplier Design

The multiplier is required to support both floating point and integer operations. To accomplish this, the entire 64x64->128 multiplication is computed, and appropriate rounding or selection of the output bits is done.

To make rounding easier, floating point inputs are aligned such that the binary point of the result will occur between bit positions 116 and 115, and the least significant bit will occur at position 64. To accomplish this, the B argument is shifted up by 12 places, putting its binary point between bits 64 and 63.

5.2.1 Booth Encoding

As mentioned earlier, a radix-8 booth encoding scheme is used to do the multiplication. Radix 8 encoding requires the partial products -4A, -3A, -2A, -1A, 0A, 1A, 2A, 3A, 4A. Generating all but the 3A and -3A partial product can be done with simple shifts and inversions; calculation of 3A requires a full adder, which occurs during this stage.

Also, the B argument will be booth-encoded into the select and invert signals for the partial product muxes. Table 5.1 shows the desired outputs for the different input vectors.

The equations used to implement this are:

$$t2 = \text{Bit2 XOR Bit1}$$

$$t1 = \text{Bit2 XOR Bit0}$$

$$t0 = \text{Bit2 XOR Bit-1}$$

$$\text{SelInv} = \text{Bit2}$$

$$\text{Sel4} = t2 \text{ AND } t1 \text{ AND } t0$$

$$\text{Sel3} = t2 \text{ AND } (t1 \text{ XOR } t0)$$

$$\text{Sel2} = t2 \text{ XOR } (t1 \text{ AND } t0)$$

$$\text{Sel1} = t2 \text{ AND } (t1 \text{ XOR } t0)$$

$$\text{Sel0} = t2 \text{ AND } t1 \text{ AND } t0$$

Bit 2	Bit 1	Bit 0	Bit -1	Partial Product	Select Invert	Sel 4	Sel 3	Sel 2	Sel 1	Sel 0
0	0	0	0	0	-	0	0	0	0	1
0	0	0	1	A	0	0	0	0	1	0
0	0	1	0	A	0	0	0	0	1	0
0	0	1	1	2*A	0	0	0	1	0	0
0	1	0	0	2*A	0	0	0	1	0	0
0	1	0	1	3*A	0	0	1	0	0	0
0	1	1	0	3*A	0	0	1	0	0	0
0	1	1	1	4*A	0	1	0	0	0	0
1	0	0	0	-4*A	1	1	0	0	0	0
1	0	0	1	-3*A	1	0	1	0	0	0
1	0	1	0	-3*A	1	0	1	0	0	0
1	0	1	1	-2*A	1	0	0	1	0	0
1	1	0	0	-2*A	1	0	0	1	0	0
1	1	0	1	-A	1	0	0	0	1	0
1	1	1	0	-A	1	0	0	0	1	0
1	1	1	1	0	-	0	0	0	0	1

Table 5.1: *Booth Encoding Outputs*

Schematics implementing these functions are shown in 5-1. Figure 5-2 shows the circuitry for calculating t2, t1, and t0 by performing an XOR of each of the low bits with the high bit. Figure 5-4 and Figure 5-3 show the circuitry to calculate Sel0/Sel2/Sel4 and Sel1/Sel3 respectively ¹.

5.2.2 Multiplication Arrays

In order to sum up the 22 partial products, two arrays are used to each add 11 partial products each. The results from these arrays are then combined and passed to the rounding stage.

As mentioned early, a 3 input 1 bit binary adder is used to produce 2 output bits; this way another partial product can be added for every stage of adders. Because the first stage can take in 3 products, this leads to a total of 8 stages to compute the sum of 11 partial products.

Another strategy is to use a 7 input 1 bit adder that produces 3 bits of output;

¹The ability to factor common transistors resulted in the somewhat strange grouping circuit topologies, instead of having 5 independent circuits

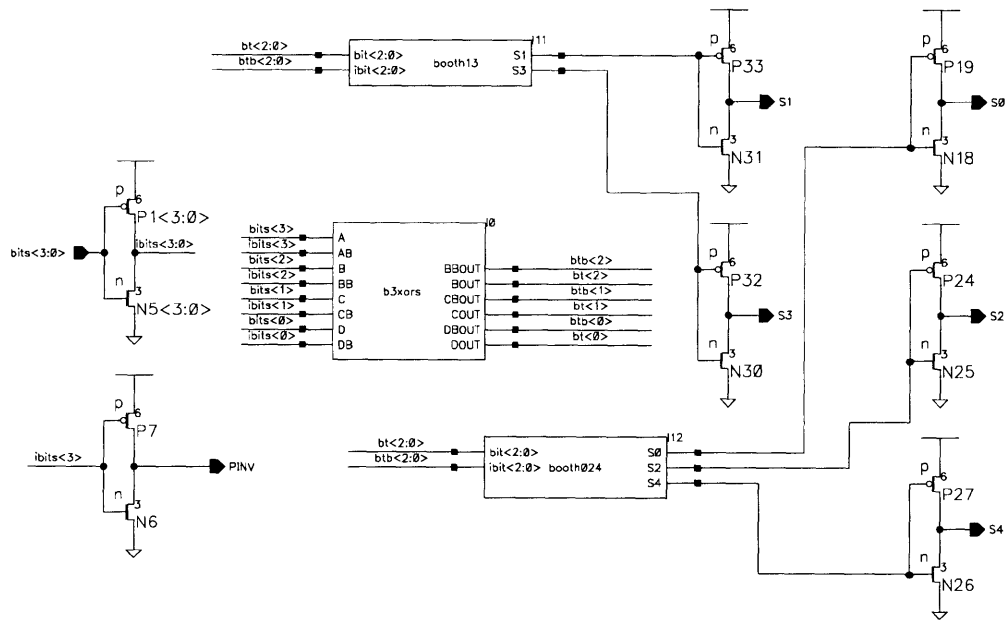


Figure 5-1: Booth Encoder Cell - Top Level

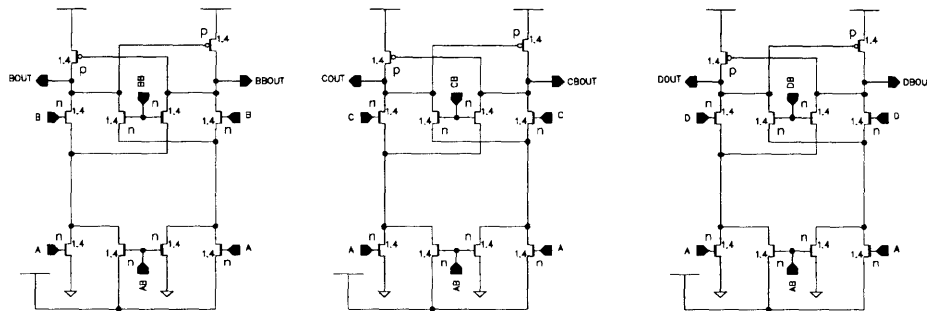


Figure 5-2: Booth Encoder Cell - XOR Inputs

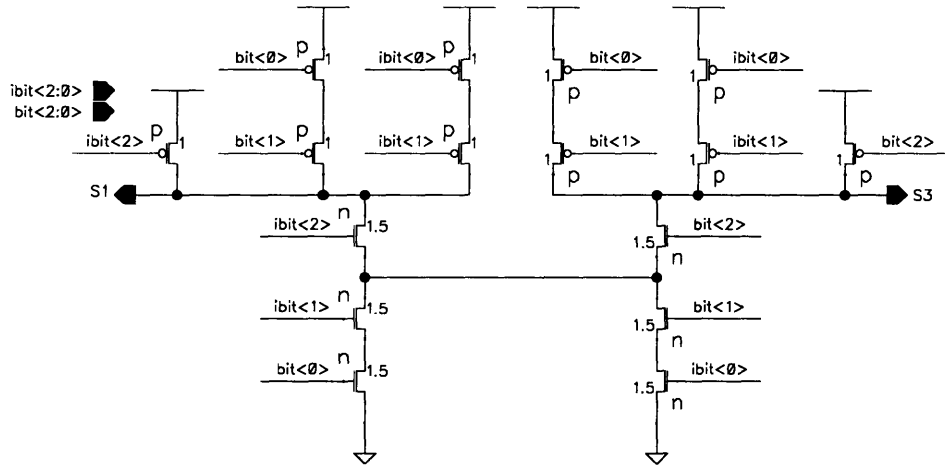


Figure 5-3: Booth Encoder Cell - Select 1,3

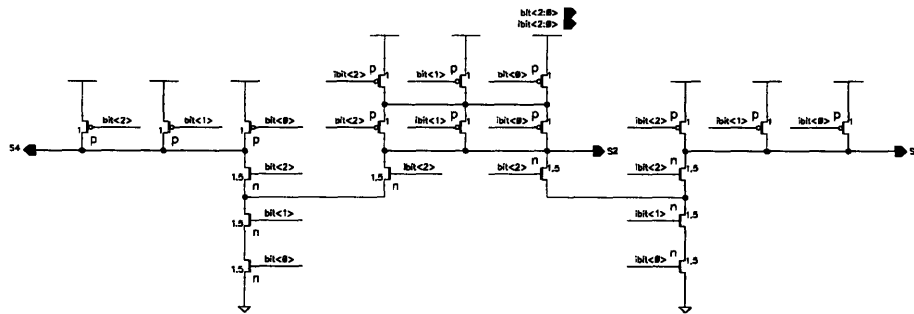


Figure 5-4: Booth Encoder Cell - Select 0,2,4

the potential is then that 4 partial products can be added at each stage, and only two stages would be necessary - or the whole 22 partial products could be added up in 5 stages. However, the circuit complexity is greatly increased for the 7-4 adder over the 3-2 adder; in addition, many more wiring tracks are needed, and using the 3-2 adder all the available wiring tracks are already taken.

Rather than compute a 128 bit sum every stage, only the 66 bits affected by the current partial product are summed. This allows the A and 3A wires (the inputs to the partial product muxes) to pass straight down the multiplication array, as well as reduces the area and power since redundant calculation is eliminated. However, the outputs of each stage must be sign extended since negative results are possible. This means that the load on the MSB of each addition stage has a load 3x normal ²; in the case of the 7-4 adder this would be 12x normal, another reason not to use it.

A differential domino full-adder was selected as the base cell of the array – see figure 6-5 (the next chapter has a comparison of different circuit designs). Because of this choice, timing of the inputs to the array and the precharge/evaluate signals to the pieces becomes critical. In addition, a domino-style mux is used for the input to the full-adders.

5.2.3 Timing Considerations

Figure 5-5 shows the timing methodology for the domino portion of the arrays. A delayed-precharge timing style, similar to that used in the Intel Pentium Pro ([7]), is used. Because the isolation inverters in the domino logic are resized to provide a faster rising edge than falling edge, more than half the cycle is allocated for precharge. This is accomplished by starting the precharge of the first stage when the clock rises, halfway into the evaluation cycle, and continuing through part-way of the next cycle. The muxes will precharge right after the clock's falling edge for a short pulse, then start evaluating.

Because the first stage will not start evaluating until shortly after the clock falls,

²See Appendix B for a technique to alleviate this extra loading

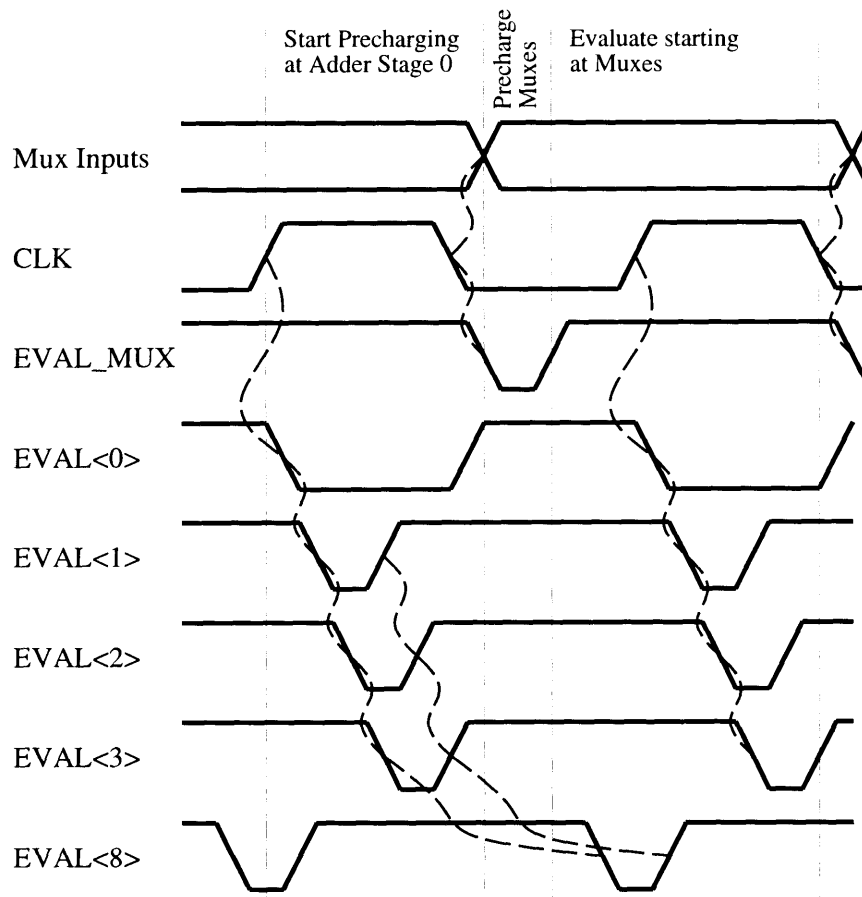


Figure 5-5: *Timing for Evaluate Signals for Domino Logic*

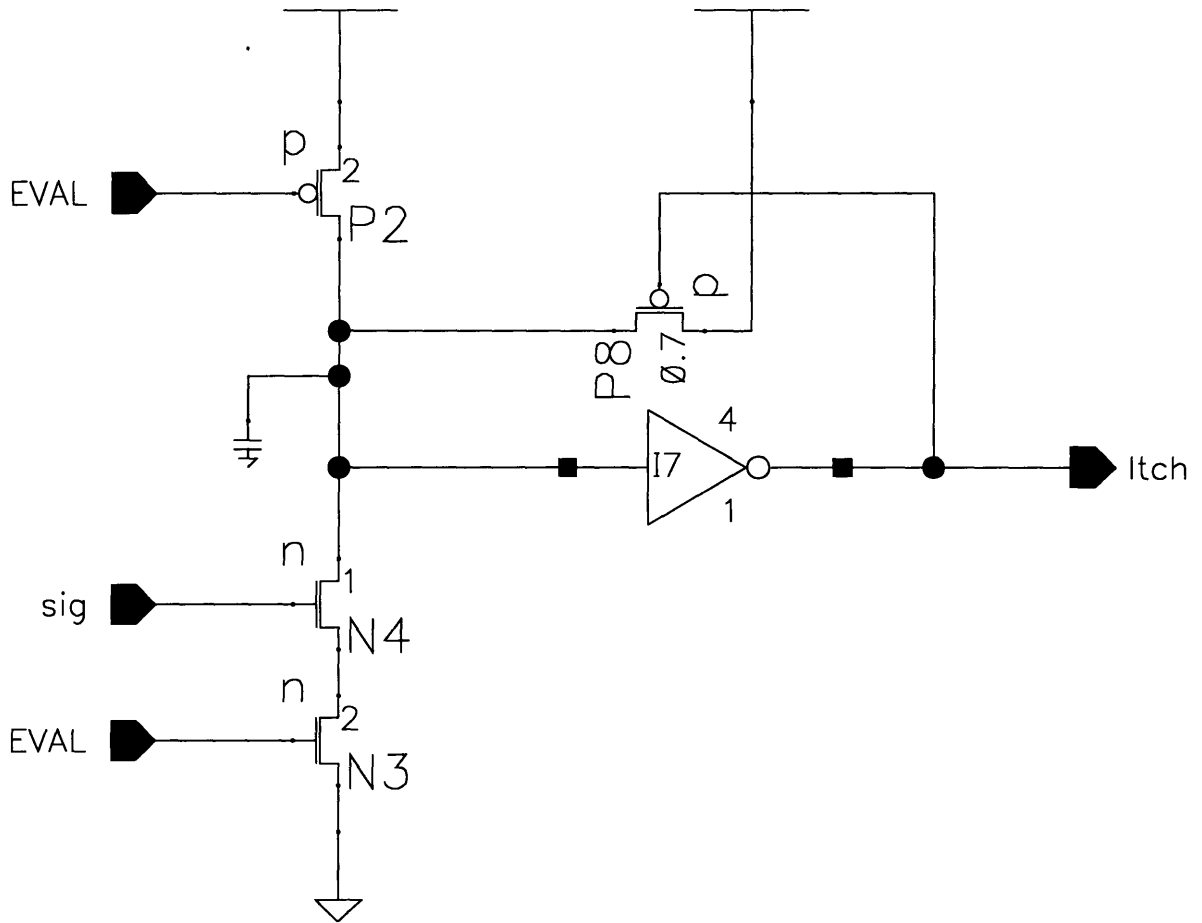


Figure 5-6: *Interface Latch for static logic from domino signals*

two of the three inputs into the next stage will not reach an assertion state (where either the signal or the complementary version goes high) until the first stage muxes evaluate. This will prevent all the rest of the next stage from evaluating, and likewise none of the latter stages will evaluate.

As a result of the staggered precharge strategy, the some of the output bits (those generated by the bits shifted off in the early stages) will not remain valid until the end of the evaluation cycle. Therefore an interface circuit needs to be added to allow seem-less integration with static cmos. For this purpose, we use a version of an RS latch (see figure 5-6) that sets on valid signal and resets on clock-low. This allows the rising edge of the output to Trigger the latch, while the falling edge caused by the precharging of the array will be ignored until the next cycle.

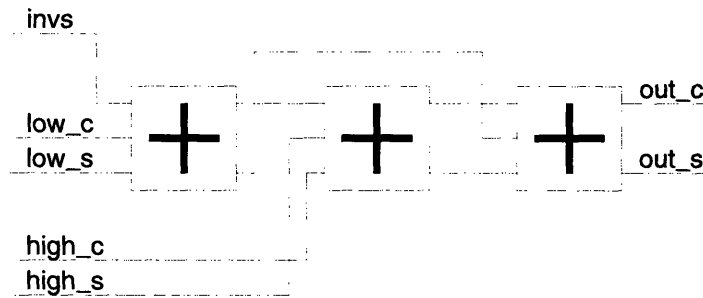


Figure 5-7: *Array Combination using 3 stages of Full Adders*

5.2.4 Array Combination

Before rounding can occur, the four 96 bit outputs from the two arrays must be combined into two 128 bit outputs. A fifth input vector consisting of the inversion selects ³ needs to be summed into the final output. This is accomplished using three stages of 1 bit adders as shown in figure 5-7. In order to simplify timing, these adders will be static hdcvsl adders rather than the domino adders inside the arrays.

5.2.5 Multiplier Rounding

The last stage of the multiplication involves combining the two 128 bit vectors from the array combination stage, adding them together, and rounding the result. Rounding is performed as discussed in Section 3.1.2 to prevent the need for two separate sequential adds.

The schematic implementation uses the shared library 64 bit adder to compute the low 64 bits of the multiplication, and a zero detector ⁴ on the result to calculate the sticky bit for rounding. The high 64 bits of the input vectors are first routed to a set of full-adders that add in the pre-rounding bit, then the result is fed to a specialized adder ⁵ that computes $a+b$ and $a+b+1$. The two outputs are then muxed

³By simply doing adding in an inverted version of the partial product rather than a 2's complement negative version, the time to generate these partial products is greatly reduced. However, the extra 1 still needs to be added into the final result, and hence occurs here.

⁴The zero detector is a shared-library element designed by Parag Gupta [5] for the Integer Unit

⁵This is the same as the normal 64 bit adder, except with 2 global carry chain; one calculates the sum with carry in of 0, one calculates the sum with carry in of 1.

by the control and the LSB is again muxed to allow forcing of it to 0 for support of round-to-nearest-even.

For integer multiplication, the pre-rounding bit is forced to 0, and the the overflow of the low 64 bit add is used to do the selection between the possible high-word outputs; the results in a simple 128 bit add, rather than a round.

5.3 Addition

The second half of the pipeline performs the IEEE double-precision add and subtract. This requires an alignment shifter, an adder and round unit, and a post-normalization stage.

5.3.1 Alignment Shifter

The alignment shifter performs the alignment shift on the argument with the lower exponent. A 2 input mux selects between the inputs to be shifted, and a 64 bit right shifter ⁶ to perform the actual shift.

To support rounding, the sticky bit needs to be calculated as well. This accomplished using a generator detector circuit that passes bits that are shifted off and forces the rest of the bits to zero. The output of the mask generator is then fed to a zero-detector, and the output becomes the inverted version of the sticky bit.

The mask generator itself uses a radix 4 scheme and 4 customized cells. Each cell has two datapath inputs (inx and iny), two datapath outputs (outx and outy), and four control (sel<3:0>). The control inputs represent a shift amount as a one-hot encoding.

Each cell is effectively a comparator against a fixed constant. When the the comparison results in equally, inx and iny pass out to outx and outy without modification. In the case of the shift amount being greater than the constant, both outx and outy are set to inx. In the case of the shift amount being less than the constant, both outx

⁶The shifter used is a shared library shifted designed by UROP Jeff Bowers for the Floating Point Divide/Square Root unit [1]

and outy are set to iny. By cascading cells, an arbitrary length comparison can be made placing the cells for the MSB's first and feeding the outputs to successively less significant cells.

The entire generator is built by tiling a 2 dimensional array of these cells where each column has a comparator with a constant equal to its column number. By feeding in the value to be shifted in the x inputs, and ground into the y inputs, the y outputs at the bottom of the comparator will be equal to the inputs only for those values shifted off ⁷.

The non-shifted input is then muxed between itself and an inverted version; this is done to support addition in the case of the inputs having different signs (or subtractions with the same sign).

5.3.2 Addition and Rounding

The addition and rounding are a combined operation using the technique described in section 3.2.1. Three sums are computed: $A+B$ and $A+B+1$ (using the dual carry chain adder) and $A+B+2$ (using a stage of 1 bit full-adders and the normal 64 bit adder). The results of these operations are fed into two 4 input muxes that select the two possibilities for the result (depending on if the result needs to round up or not); in addition, these muxes will invert the output in the case of negative results.

Actual rounding calculation is done inside the control logic using the two LSB's of the shift output and the sticky bit calculated. A 3 bit leading zero detector determines the rounding position ⁸. A two input mux is then used to select between the rounded and non-rounded cases. Finally, another two-input mux is used to implement round-to-nearest even for forcing the LSB to be 0.

⁷In this case, the function implemented is greater-than. By altering the values places in the top inputs, less-than, less-or-equal, and greater-or-equal functions can easily be created

⁸This leading zero detector is implemented in datapath logic for speed reasons; given a slightly more relaxed cycle time, it would be preferable to implement in standard cells

5.3.3 Post-normalization

To complete the addition operation, the result needs to be normalized. A 64 bit leading zero detector is used to determine how much shifting is necessary. This result is passed to the control logic ⁹ and then fed to another shifter to perform the actual normalization.

The leading zero detector uses a structure similar to the 64 bit adders. 8 local chains each look at 8 bit of the input vector, and perform a leading zero detect. The results are then passed to a global chain, which determines which local chain detects the first 1. The global chain asserts one of the drive signals to a local carry chain, and that chain drives the correct bit pattern onto the output lines.

5.4 Support for other operations

The pipeline discussed supports many of the required operations. Some addition hardware is needed to support immediate instructions, send instructions, and conversion instructions. In addition, a bank of registers are used in the generation of error-vals which are generated in place of exceptions ¹⁰.

5.4.1 Immediate Support

A separate set of 4 registers is used to support immediate and move operations. These registers are wired back-to-back with muxes to allow feed-through when the pipeline is empty. A 2 input mux on the input is used to support the SHORU instruction ¹¹.

5.4.2 Send Operations

Send operations are supported by a separate send unit, which is a duplicate of the send unit used in the integer unit [5].

⁹In the case of gradual underflow, the result should not be normalized so much that it results in an exponent $j - 1023$.

¹⁰See M-Machine Exception Document...

¹¹SHORU A, B, C implements $C = (A \ll 16) + (B \& 0x3F)$

5.4.3 Err-vals

Like the immediate registers, a set of 4 registers is used to support err-val generation. When an errval is detected as one of the inputs to an operation, that value is placed in the err-val path. Otherwise, the instruction pointer along with some status bits are written into the err-val path. When the operation completes, if an err-val was detected, or a condition occurs generation an err-val (such as a $0 * \text{infinity}$ operation), the err-val is written back instead of the result.

5.4.4 Conversion Operations

FTOI and FTOUI conversion operations are fairly straightforward, and only require a change in the control logic to correctly post-normalize the number. ITOF, on the other hand, requires a 11 bit leading zero detect on the input integer to determine if any right-shifting is necessary. Originally, this was intended to be done in the datapath logic, but for now is being implemented in standard cells ion the control logic.

Chapter 6

Circuit Design for the Multiplier Array

Because of the large number of stages of addition in the multiplication array, considerable effort was placed on making the 3-2 adder circuit as fast as possible. 5 different designs were produced and considered, all optimized for two late arriving inputs and one early input, since the partial product is available at the start of the cycle.

Static CMOS

The first is a standard static CMOS full-adder circuit, taken from *Principles Of CMOS VLSI Design* [6] ¹ - see figure n. The main disadvantage to this design is that three stages of logic separate the sum output from the inputs; one to calculate carry, one to calculate sum, and the last to invert sum. This could be reduced to 2 by using alternating between inverted-input adders and non-inverted-input adders to eliminate the final inverter ²

Pass Transistor CMOS

The second style has the same carry calculation circuitry, but uses a 4 transistor XOR to calculate $A \text{ XOR } B$ and a final pass-transistor XOR to calculate $(A \text{ XOR } B) \text{ XOR}$

¹The second edition circuit has a few errors

²actually, the inverted-input/output and noninverted-input/output adders are equivalent circuits

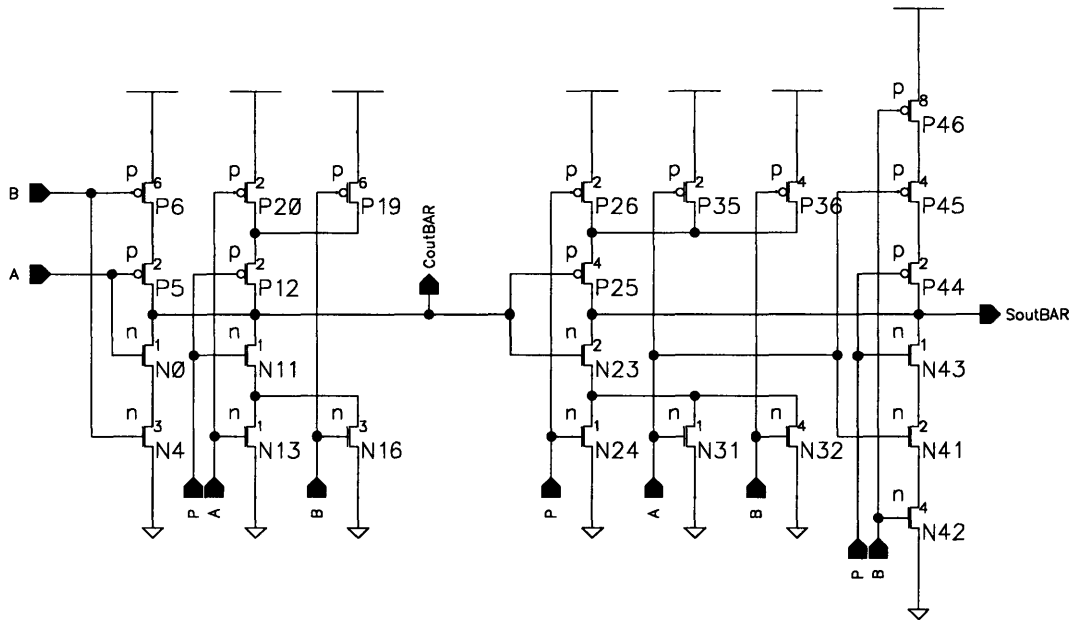


Figure 6-1: *Static CMOS Full Adder* (Output Buffers not shown)

C. See figure n

One quirk is that this version requires both true and complemented versions of input A, and produces true and complemented versions of the sum bit. This allows the number of stages to calculate the sum bit to be reduced to effectively 1, speeding up the circuit greatly.

Differential CVSL

The third style uses different CVSL circuitry to calculate both the sum and carry bits. Because it is differential, both normal and complemented forms of the inputs are available for computation, making the sum bit computation much easier. Also, the CVSL requires only pulldown circuitry, hence the input loading is much less than normal logic.

Due to the cross-coupled PFETs on the DCVSL circuit, there is an inherent difference in edge times. Because the pulldown circuit must always pull below $V_{dd} - V_t$ before the pullup circuitry on the other side engages. Therefore, the falling edge will always be faster than the rising edge.

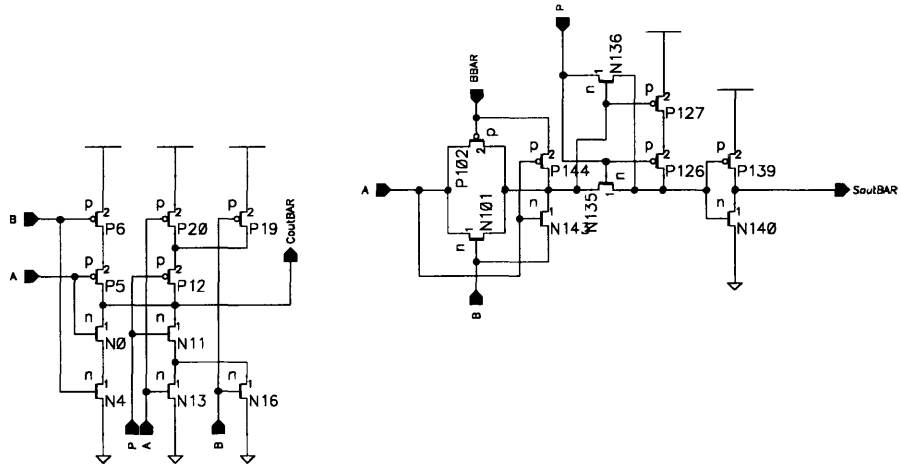


Figure 6-2: *Pass Transistor Full Adder (Output Buffers not shown)*

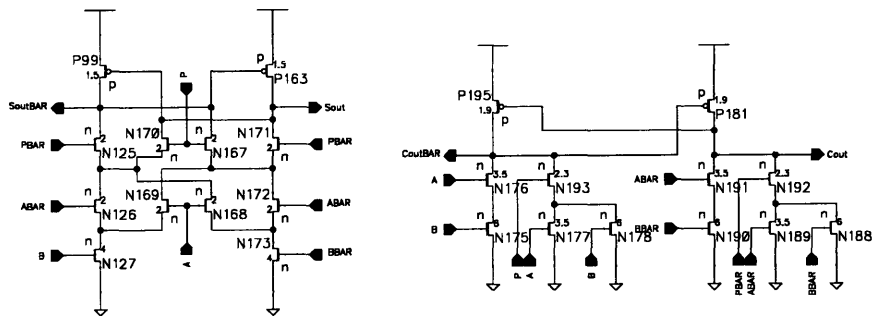


Figure 6-3: *Differential Cascode Voltage Switch Logic Full Adder*

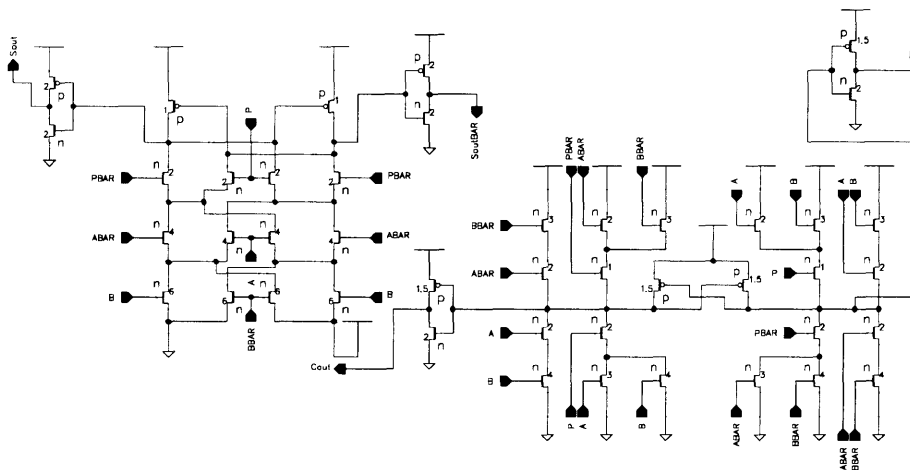


Figure 6-4: *Hybrid DCVSL Full Adder*

Given a number of consecutive stages of DCVSL circuitry, though, the problem does not become as bad. This is due to the fact that after the first stage, the falling edge will be occurring before the rising edge, and therefore the pulldown tree on one side will turn off before the pullup tree on the other side engages. Therefore, the effective worst-case delay on stages after the first will be less than the worst-case rising edge delay.

One potential concern is the sizing of the pullup pfet's versus the pulldown nfets. Given process variation with weaker nfets and stronger pfets, its possible the nfets might not be able to overcome the pfets, and cause the circuit to fail.

Hybrid Differential CVSL

The forth style is a modification of the CVSL circuitry to pull up as well as down, except using nfets exclusively. The main incentive for this is to eliminate the fight between the nfets and pfets, while still retaining reduced input loading from that of static cmos. Because of the pullup tree consisting of nfets, the two cross-coupled pfets still need to exist to bring the node voltage all the way to Vdd. However, the nfets on the pullup side can be much smaller than the pfets in static cmos.

Because of the structure of the XOR tree in the sum bit calculation, adding the circuitry to pullup consists of only two additional transistors; this greatly increases

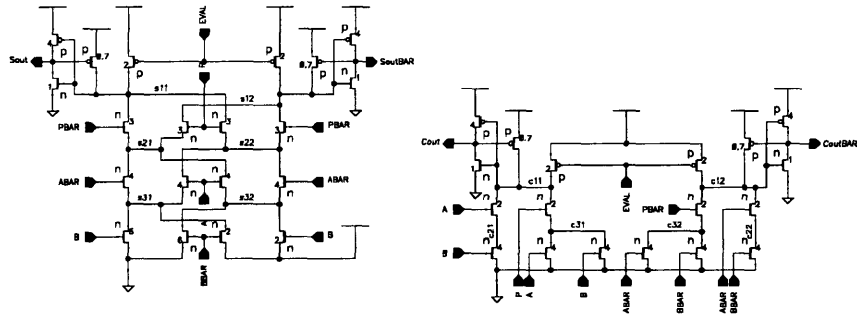


Figure 6-5: *Differential Domino Full Adder*

the speed of the sum bit calculation while only minimally impacting the input loading. Unfortunately, a full tree must exist for pulling up when calculating the carry bit, but the sizes are smaller than that of the pulldown tree.

Differential Domino

The final style analyzed is that of differential domino - a precharge/evaluate logic style that uses nfet pulldown trees and an ratio-ed inverter to isolate the precharged node. Before calculation starts, the node is charged via a single pfet. Then, the pfet is turned off, and the evaluation begins.

Again, the use of nfets only allows much reduced input capacitance. By appropriately ratioing the nfet and pfet in the isolation inverter, one can change the threshold point of the inverter, and cause it to trip earlier, and decrease the total delay through the circuit.

Unfortunately, precharge-evaluate logic families logic domino introduce a whole new set of concerns. The first is a restriction on when the inputs are valid; once evaluation starts, the signals may only be monotonically increasing. If a signal undergoes a high-to-low transition, the output signal will not necessary De-assert itself.

The second concern is that one must build a timing circuit to generate the precharge and evaluate signals at the appropriate time.

The rest of the concerns deal with the fact that the evaluate node is not necessarily driven; it is possible that once the node is left floating high, it will incorrectly be

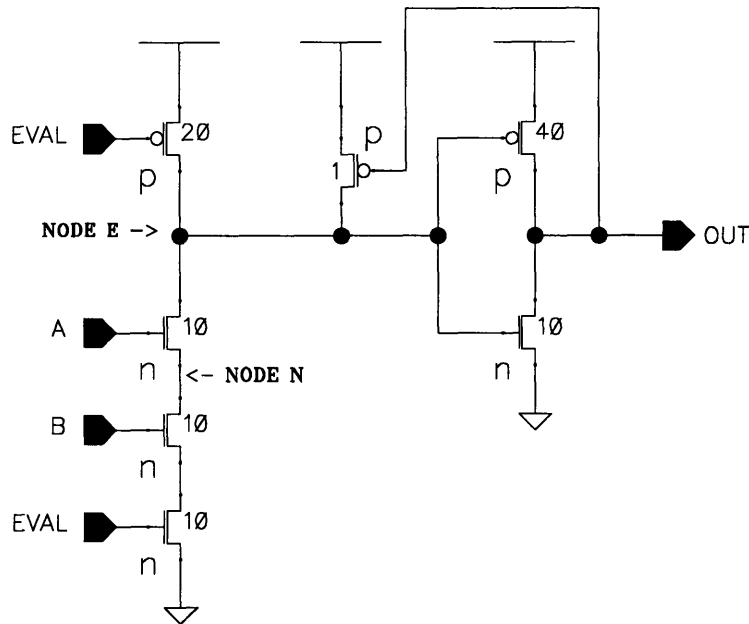


Figure 6-6: *Example of a circuit where charge sharing can occur*

discharged enough to trip the isolation inverter.

This can occur simply by charge leaking off the evaluation node. A simple way to handle this is to put a very weak P device on the evaluation node, driven by the output of the isolation inverter. This will tend to restore any high voltages on the evaluation node if its starts leaking away, but not enough to significantly slow down a proper evaluation.

Another way the node can be discharged is by charge sharing. Consider the circuit in figure 6-6. On the first cycle, signals A and B will be high during evaluate; besides discharging the evaluate node (E), this will also discharge the parasitic capacitance at node N. During the second cycle, A will be high, but B will be low. This will cause node N and E to share their charge, equalizing the voltages at these nodes. Because node N was discharge the cycle, the voltage on node E will tend to drop, possibly enough to incorrectly trip the isolation inverter; after this happens, the weak P pullup device does not help.

Finally, crosstalk capacitance between the evaluation node and some other control signals can cause premature discharge. Given a sharp pulse on the control wires, the

voltage on the evaluate node might be driven higher or lower; in the case of it being driven lower, the isolation inverter might incorrectly trip.

In order to deal with this problem, the XOR tree for the sum bit calculation was made to pull both directions, as in the Hybrid CVSL version. This ensures that the evaluate node is always driven during evaluation, and eliminates the charge sharing problems from all the intermediate nodes.

Adding the pullup circuitry on the carry bit circuit would be quite costly, however. Fortunately, charge sharing is less of an issue, since there are only 2 stacked nfets. In addition, the crosstalk capacitance needed to cause failure was determined, and shown to be much greater than the actual amount of crosstalk capacitance that will be present in the physical layout.

6.1 Evaluation

To evaluate the worst case edges for each static logic family, twenty-four test vectors were generated, as given in table 6.1. These vectors represent every possible transition of the late arriving inputs (A and B) that can occur.

Worst case delays for these families show a wide range when driving a 4x Load. Differential Domino has the best performance, approximately 550ps. HDCVSL has the best time for a static family, at 730ps. DCVSL does considerably worse with delays over 1ns, and the rest of the families are even worse.

For most families, the best and worst case edges have very little separation; whenever possible, sacrifices on the best edge were made to speed up the worst edge. For DCVSL, the rising edges are always slower, and these edges were optimized as much as possible.

Vector	A Input	B Input	P Input
1	Rise	0	0
2	Fall	0	0
3	0	Rise	0
4	Rise	1	0
5	1	Fall	0
6	1	Rise	0
7	Fall	Fall	0
8	Rise	Rise	0
9	Fall	1	0
10	Rise	Fall	0
11	Fall	Rise	0
12	0	Fall	0
13	Rise	0	1
14	Fall	0	1
15	0	Rise	1
16	Rise	1	1
17	1	Fall	1
18	1	Rise	1
19	Fall	Fall	1
20	Rise	Rise	1
21	Fall	1	1
22	Rise	Fall	1
23	Fall	Rise	1
24	0	Fall	1

Table 6.1: Test Vectors for Static Logic

Chapter 7

Results

Verification of the design was done for both timing and functional issues. Timing verification was accomplished using the HSPICE circuit simulator to test individual pieces and obtain worst case delays for the individual units. Table 7.1 shows the worst case time-path ¹ for the entire multiply add, run with a vdd of 3.0 ² and Typical models for the transistors.

Functional verification was done by extracting the entire design to a Verilog netlist and simulating with Verilog-XL. A set of inputs designed to do path-complete testing were run through the design, along with a large number of random vectors. The results were compared against the results from the C library routines running on a Sparc system. All vectors matched exactly, except for multiplications with gradual underflow inputs, for reasons discussed earlier.

The physical layout has been completed only recently; unfortunately, the area used is 50estimated areas are shown below.

Width = 66 datapath cells, 1995.84 microns

Booth Encoding	Height	Cumulative
Adder	270 micron	270

¹Because of the use of transparent latches, the stages are allowed to borrow time from each other, and need not be less than 5ns.

²The voltage was reduced to give some margin in case of unaccounted for delays, such as local wiring capacitance

Unit	Subunit	Latency (TTL)
Select inputs		1 ns
Stage 0	CLA Adder	4.7 ns
	Latch	3.7 ns
	Latch	.5 ns
	Latch	.5 ns
Stage 2	Domino Mux	7.44 ns
	Domino Full Adder	1 ns
	HDCVSL Full Adder	.46 ns * 9
	Latch	.6 ns * 3
	Latch	.5 ns
Stage 3	AddBoth	7.8 ns
	Zero Detector	4 ns
	Control Logic	1.3ns
	Latch	2 ns
	Latch	.5 ns
Stage 4	Mask Generator	3.3 ns
	Latch	1.5 ns
	Zero Detector	.5 ns
	Latch	1.3 ns
	Latch	.5 ns
Stage 6	AddBoth	7.5 ns
	Control Logic	4 ns
	Latch	3 ns
	Latch	.5 ns
Stage 7	LZDetect	6.3 ns
	Control Logic	2.7 ns
	Shift Left	2 ns
	Latch	1.5 ns
	Latch	.5 ns
WB mux and driver		1 ns
Total		39.5 ns

Table 7.1: *Critical Path Timing*

	Booth Encoding Cells	25 micron	295
Multiplier Array			
2x	Latch Cell	10 micron	315
16x	Domino fadders w/ muxs	50 micron	1115
3x	CVSL fadders	30 micron	1205
	Latch	10 micron	1215
	Routing Tracks	230 micron	1446
Multiplier Rounding			
	Adder	270 micron	1715
	AdderBoth	400 micron	2115
	CVSL fadders	30 micron	2145
3x	mux2	5 micron	2160
Alignment			
4x	mux2	5 micron	2180
	uni-dir shifter	150 micron	2330
6x	Latch	10 micron	2390
	mask gen	75 micron	2465
	zero detect	27 micron	2492
Add and Round			
2x	CVSL fadder	30 micron	2552
2x	addboth	400 micron	3352
2x	mux4i	10 micron	3372
Post-Normalization			
	uni-dir shifter	150 micron	3522
	LZ Detect	125 micron	3647

Other Stuff

LZ Detect	125 micron	3772
WB driver	50 micron	3822

Area bloats were divided between the multiplier array (where the desired density was hard to achieve due to local wiring constraints) and the adder cells. Because of the large scale use of shared-library datapath cells, optimal density was not achieved; unfortunately, time and monetary constraints restricted a more custom approach.

Figure 7-1 shows the current floorplan of the Multiple ALU Processor³. Three FPU-MULA units are shown; due to area constraints, one of the clusters (along with its FPU-MULA unit) were eliminated. These three units are located at the right side of the die, with the data bits flowing left-to-right.

³The FPU-MULA unit has the dubious distinction of being the largest single block on the MAP chip

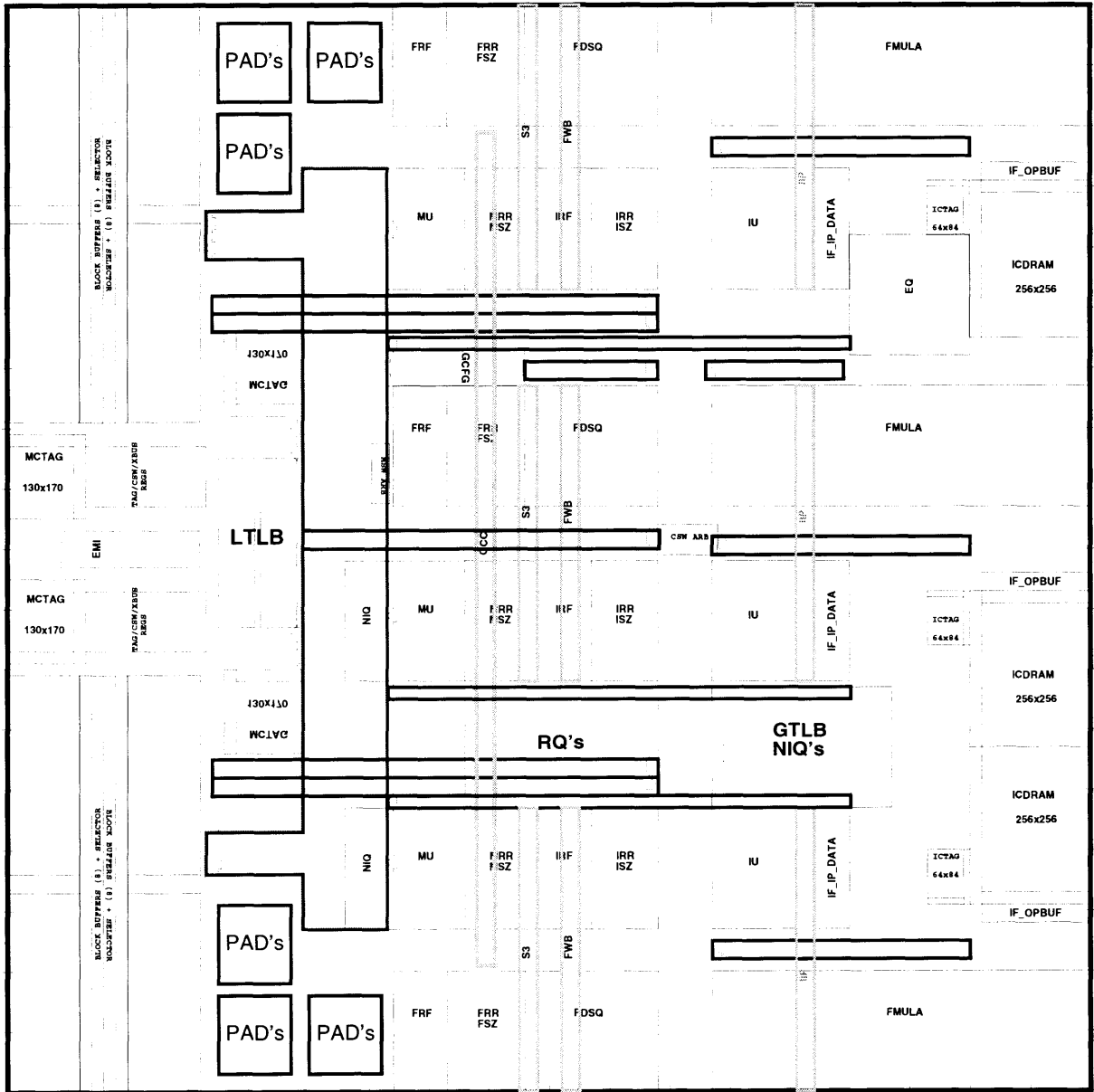


Figure 7-1: *Current Floorplan for the Multiple ALU Processor*

Chapter 8

Conclusion

Other than the original estimated area, all constraints were met. The FPU-MULA has a fairly decent 4 cycle latency, compared to the IBM PowerPC 601 ¹ which has a latency of 7 cycles at a similar clock speed.

The multiply operation itself has been implemented in faster circuits than the 20ns latency used here. Many such implementations use a tree-style multiplier rather than array multiplier; however, tree-style multipliers require a bigger area because of the large number of long wires.

Another impact on timing was the use of static circuits in all areas except the actual array. Given free usage of precharge/evaluate logic style, and some of the techniques discussed in appendix B, A latency of 10ns could probably be accomplished on the given process without resorting to a tree-style multiplier.

The Multiplier/Adder unit should be instantiated in silicon when the Multiple ALU Processor enters fabrication in early 1997.

8.1 Lessons Learned

Finally, here's a brief list of lessons I've learned from the project.

¹a RISC processor done on the same fabrication process

1. Drain capacitances can have a large impact; large, single stage muxes will be slow.
2. There is a significant more-than-linear growth of transistor strengths when increasing from minimal size.
3. Being aware of the layout constraints is key, both at a large floorplan level and a small circuit level. A good number of circuit designers seem unaware of this.
4. Precharge/evaluate style logic is very fast, though the engineering effort required to engineer it correctly and validate operation is large.
5. If one is attempting a completely new project, the estimated schedule will be WAY off.
6. A little algorithm hacking can go a long way - see alternate rounding techniques in Chapter 3, as well as Appendix C
7. As Tom Knight likes to frequently state, the wires are a very big problem indeed!

Appendix A

IEEE Double Precision Format

The Double Precision floating point format [2] fits in a 64bit word. The MSB, bit 63, represents the sign of the number; a 1 here signifies a negative number. The next 11 bits, bits 62-52, contain the exponent + 1023. Finally, the last 52 bits contain the mantissa without the leading one. The table A.1 shows some examples, plus the representations for some special types.

Bit 63 Sign	Bits 62:52 Exponent	Bits 51:0 Mantissa	Represented Value
0	0x3FF	0x00000000000000	$1.0 * 2^0 = 1.0$
1	0x3FF	0x80000000000000	$-1.5 * 2^0 = -1.5$
0	0x400	0x40000000000000	$1.25 * 2^1 = 2.5$
0	0x000	0x00000000000000	0.0
1	0x000	0x00000000000000	-0.0
0	0x000	0x80000000000000	$0.5 * 2^{-1022}$
1	0x000	0x40000000000000	$0.25 * 2^{-1022}$
0	0x7FF	0x00000000000000	+infinity
1	0x7FF	0x00000000000000	-infinity
0	0x7FF	0x8nnnnnnnnnnnnnn	NAN = not a number

Table A.1: *IEEE Double Precision Examples*

Appendix B

An Alternative Multiplication Scheme for Eliminating Sign Extension

As mentioned earlier, the critical path through the multiplier array is through the most significant bit of the sub-arrays; this is due to the extra loading on the outputs of the cell. Since the output words are shifted right by 3 bits for the sum and 2 bits for the carry, a sign extension of 4x and 3x respectively is required. This 4x loading greatly increases the delay. After implementation of the FPU-MULA unit was completed, an alternative technique was discovered that reduces this critical path delay.

The technique relies on a recoding of the output and input bits into the most significant bit, and uses the fact that the inputs of an adder are interchangeable.

For a given adder, there are 3 inputs, one which come from the partial product generated, two which come from the previous stage. These inputs and the results they generate are listed in table B.1 below.

The key observation is that Bits 65, 64, 63, and 62 will all receive the same Sout bit, and that Bits 65, 64, and 63 will all receive the same Cout bit; furthermore, because the inputs to Bits 65, 64, and 63 all come from the MSB cell, the order of the Sin and Cin inputs can be permuted. Therefore, a recoded Sout and Cout shown

PP	Cin	Sin	Cout	Sout
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table B.1: *Typical Full Adder*

PP	Cin	Sin	CoutR	SoutR
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table B.2: *First Recoded Adder*

in table B.2 will give valid results when added in to the next stage ¹.

As shown, the truth table is now *much* simpler and can be implemented with a simple 3 input AND and a 3 input OR gate. These gates will run considerably faster than the 3 input XOR and 3 input majority gate in the normal adder, and can be scaled up to drive the excessive load much faster than the normal adder, though probably slower than the unloaded normal adder, and thus still in the critical path.

However, the recoding can be applied yet again for additional gain! Here, the observation is that for stages after the first, the MSB adder will be given two of the inputs as recoded inputs. Table B.3 shows the truth table for this new recoded adder.

This new table can now be implemented with a 2 input OR and a 2 input AND, both of which use the partial product input and one of the sum/carry inputs. This circuit can run MUCH faster than the normal adder, probably in comparable time to the unloaded normal adder when driving a 4x load.

¹Bit 62, however, takes Sout and Cout from different adders, and therefore needs to compute the *real* Cout from the MSB.

PP	Cin	Sin	CoutR	SoutR
0	0	0	0	0
0	0	1	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	1	1	1

Table B.3: *Second Recoded Adder*

One thing to note is that the Bit 62 full adder takes as inputs Sout from the MSB and Cout from the MSB-1 adder. Therefore, the normal Sout must still be generated from the MSB adder.

This technique can also be easily applied to higher radix multiplication where even more sign extension is required, such as Radix 8 (where a 8x load occurs).

Another possible area for improvement is recoding the partial product input², allowing the use of adders bigger than the 3-2 adder used here. Using a 7-3, 6-3, or even 5-3 adder would allow additions of 4, 3, or 2 partial products each stage respectively. Unfortunately, not enough time was available to do substantial analysis, though a 5-3 adder with recoded partial product

²This input will arrive simultaneously at all stages, and therefore for latter stages of the multiplier, will arrive *much* earlier than the other inputs. Therefore, even a very complex encoding could be done on the latter stages

Appendix C

Draft HDCVSL Logic Paper for JSSC

Below is a preliminary draft of a paper intended for eventual submission to the *IEEE Journal of Solid-State Circuits*. Although Digital Equipment Corporation patented a circuit family of this type 2 years ago, the results were never published, and I 'rediscovered' the circuit family before becoming aware of this.

Unfortunately, power numbers were not available, and thus the discussion section on power is blank.

The most commonly used logic family among processes having both NFET's and PFET's is, of course, that of static CMOS. Its main advantage over other possibilities is sheer simpleness of design. Static CMOS gates dissipate no static power, have rail-to-rail output voltages, and will function across all process corners without careful attention to device sizings. However, power, area, and delay considerations may warrant using a different logic style.

Differential Logic families can provide increased performance while still maintaining untimed operation ¹. Three different differential logic families are presented and compared.

¹Generally, precharge-evaluate type circuits will outperform any untimed circuit, but the extra engineering effort required to engineering the timing circuitry and validate the circuits operation is enough to discourage use

C.1 Differential Logic Families

Differential CMOS (DCMOS) can be used to reduce remove inverter stages that would correct the polarity of the output. Being a differential logic style, complementary inputs are required to generate the complementary outputs. Compared to static CMOS, the total input capacitance will be about double, due to the extra set of transistors for computing the complementary output.

Differential Cascode Voltage Switch Logic ² is one such family. By only using the pulldown chain from differential CMOS, the input loading is reduced by roughly a third. Pullup is accomplished by a solitary PFET that has its gate connected to the the output of the other pulldown chain. Switching occurs when one pullup tree turns on; when this occurs, the pulldown chain overpowers the PFET and begins to reduce the voltage on this node. As this occurs, the other pulldown chain turns off, and the PFET associated with it has its gate start to turn on as the node from the first pulldown chain reduces in voltage. This action turns off the other PFET, and eventually both nodes switch.

DCVSL inherently has a slower rise time than fall time, due to the gates of the pulldown logic directly being connected to the inputs, while gates of the pullup logic is indirectly connected through pulldown logic of the other side ³. Another problem is that incorrect ratioing of the PFET's and NFET's can cause failure of the circuit; if the pulldown chain can not overcome the pullup PFET, the outputs will not switch. Finally, the short circuit current is much higher than that in static CMOS.

A hybrid of DCVSL, HDCVSL⁴, overcomes most of these problems by adding a pullup chain to each output. The pullup chains are constructed out of NFET's rather than PFET's of differential CMOS; this provides a pullup to $V_{dd}-V_t$ which is enough to seriously weaken the PFET on the side being pulled down. As the one output is pulled to ground, the PFET on the other side will complete the pullup to V_{dd} .

²Patented by International Business Machines, Inc.

³This effect is lessened when one DCVSL stage drives another, as the early arriving falling edge turns off the pulldown chain before the rising edges activate the pullup chain

⁴Patented by Digital Equipment Corporation

HDCVSL alleviates most of the problems inherent in CVSL. Rise and fall times can be made approximately equal, since both actions are now directly coupled to the inputs. Incorrect ratioing can no longer cause a functional malfunctioning; pulling down no longer consists of a NFET chain overcoming a PFET pullup - instead, the PFET pullup will be shut off by the NFET pullup chain on the other output. Finally, less short circuit current will flow, due to the pullup PFET being turned off early.

C.2 Analysis

Differential logic families work best when building symmetric gates – where the number of transistors and circuit topology is approximately the same between the active-high and active-low output. This gives approximately equal propagation delays for both outputs.

Two common circuits that are symmetric are the majority gate and the parity gate, both used in 1-bit adders. The two input parity (or XOR gate) has been chosen as a test circuit for comparison.

The schematic implementations of the XOR gate are shown in figure C-1. The test circuit measures the delay from input crossover to output crossover driving a fanout of 3x. Input Loading is in terms of microns of gate width. Power measurements are for only for the middle XOR (accounting for short-circuit current and output loading).

Logic Family	Tpd A Input	Tpd B Input	Input Load A	Input Load A	Power A Trans	Power B Trans	Gate Area
DCMOS	364ps	492ps	60u	60u			240u
DCVSL	456ps	579ps	20u	20u			100u
HDCVSL	410ps	456ps	40u	40u			190u

Table C.1: *Results for 2 Input XOR Comparison*

As table C.1 shows, DCMOS and HDCVSL have approximately the similar delays (DCMOS being slightly faster for the A input, HDCVSL faster for the B input), but both are considerably faster than DCVSL. DCMOS does surprisingly well – even with its high input loading – because less short circuit current is dissipated than HDCVSL

and DCVSL. With lighter loads, the performance of DCMOS relative to the other two is even better.

****Insert discussion about power****

In terms of gate area ⁵, DCVSL uses considerably less, asymptotically approaching 1/3 of DCMOS. HDCVSL is in the middle, with asymptotically 2/3rd the gate area of DCMOS.

C.3 XOR Folding

The XOR network used can be arranged in a slightly different topology, giving the circuits shown in figure C-2. This can be generalized, allowing one to implement a n-input pulldown tree (or pullup tree) using $4n-2$ devices by building a crossover stage for every input.

HDCVSL allows one more degree of folding, resulting in a pullup AND pulldown tree in only $4n$ devices (versus $8n-4$ devices for the DCMOS pullup and pulldown tree). In fact, any boolean expression of the form $A \text{ xor } F(A,B,C,...)$ can be folded to use the same pulldown tree as HDCVSL plus two additional devices.

Logic Family	Tpd A Input	Tpd B Input	Input Load A	Input Load A	Power A Trans	Power B Trans	Gate Area
DCMOS	318ps	364ps	60u	60u			180u
DCVSL	364ps	466ps	20u	20u			80u
HDCVSL	230ps	277ps	40u	40u			110u

Table C.2: Results for Folded 2 Input XOR Comparison

As shown in table C.2, the HDCVSL XOR is the clear winner in terms of delay; this should give similar advantages for any such foldable circuits. The DCMOS and HDCVSL have slightly less delays than the non-folded version, but are still approximately proportionate.

⁵This is at best a rather crude estimate of the area required to implement the circuit. However, both HDCVSL and DCVSL will have a clear with only needing two PFETS, and thus a very small well region for NWELL processes. DCVSL has the additional advantage of only having 1 complicated network, and could very easily have half the area of DCMOS

*** Say something about power ***

In terms of gate area ⁶ HDCVSL is now very close the DCVSL, and both have an area asymptotically approaching 1/3 DCMOS.

C.4 Conclusion

DCMOS remains a very good basic assembly circuit for general useage. When area is extremely critical, DCVSL can save up to 1/3 the area, but at a considerable delay penalty. HDCVSL provides a compromise between these two families.

For circuits with an XOR component, HDCVSL is the clear winner in all areas except size, where DCVSL is only marginally better. When building XOR trees, the folded HDCVSL XOR should be the circuit of choice.

⁶Again, this is a rather questionable benchmark. However, in terms of wiring complexity, HDCVSL now is very close to DCVSL, and both are much better than DCMOS. DCVSL still has a slight advantage in that VDD only connects to PFETS, which might simplify layout a bit

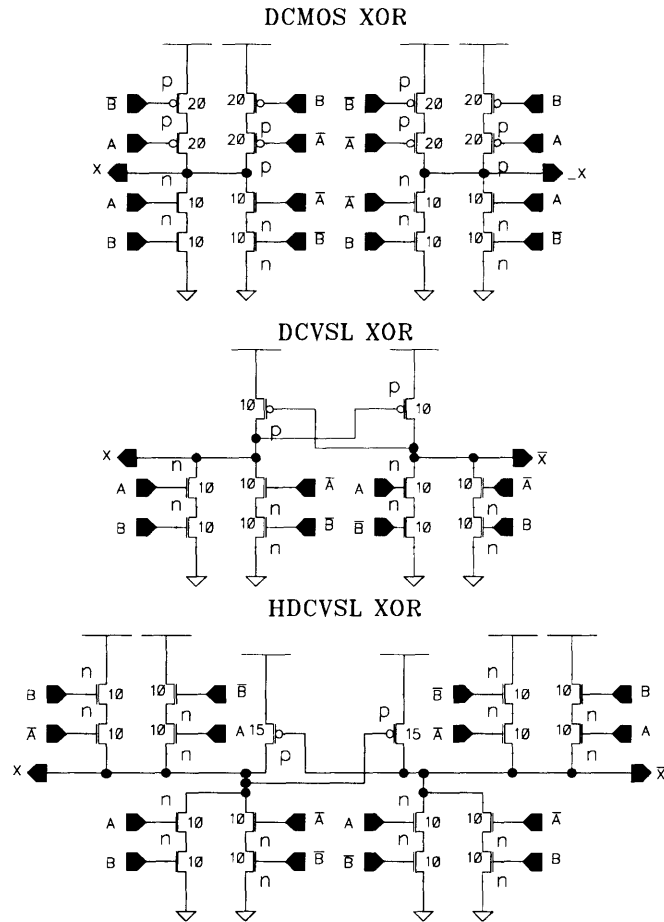


Figure C-1: 2 Input XOR gates for DCMOS, DCVSL, and HDCVSL families

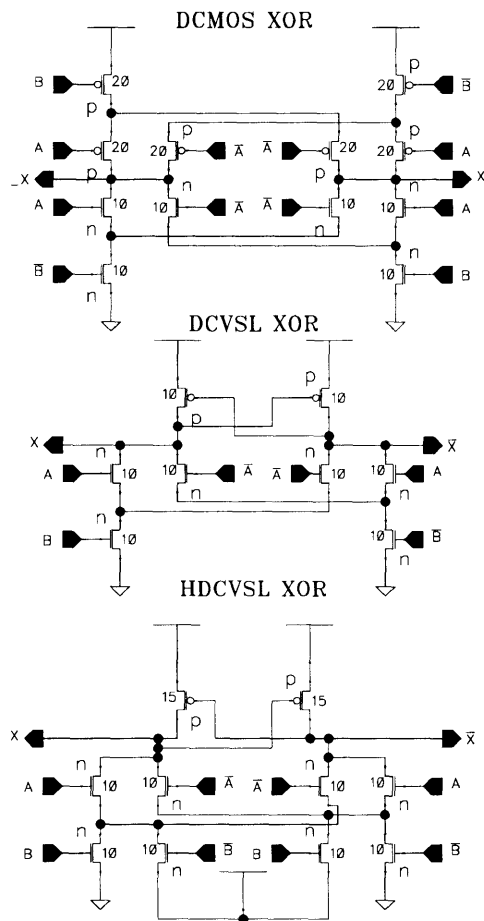
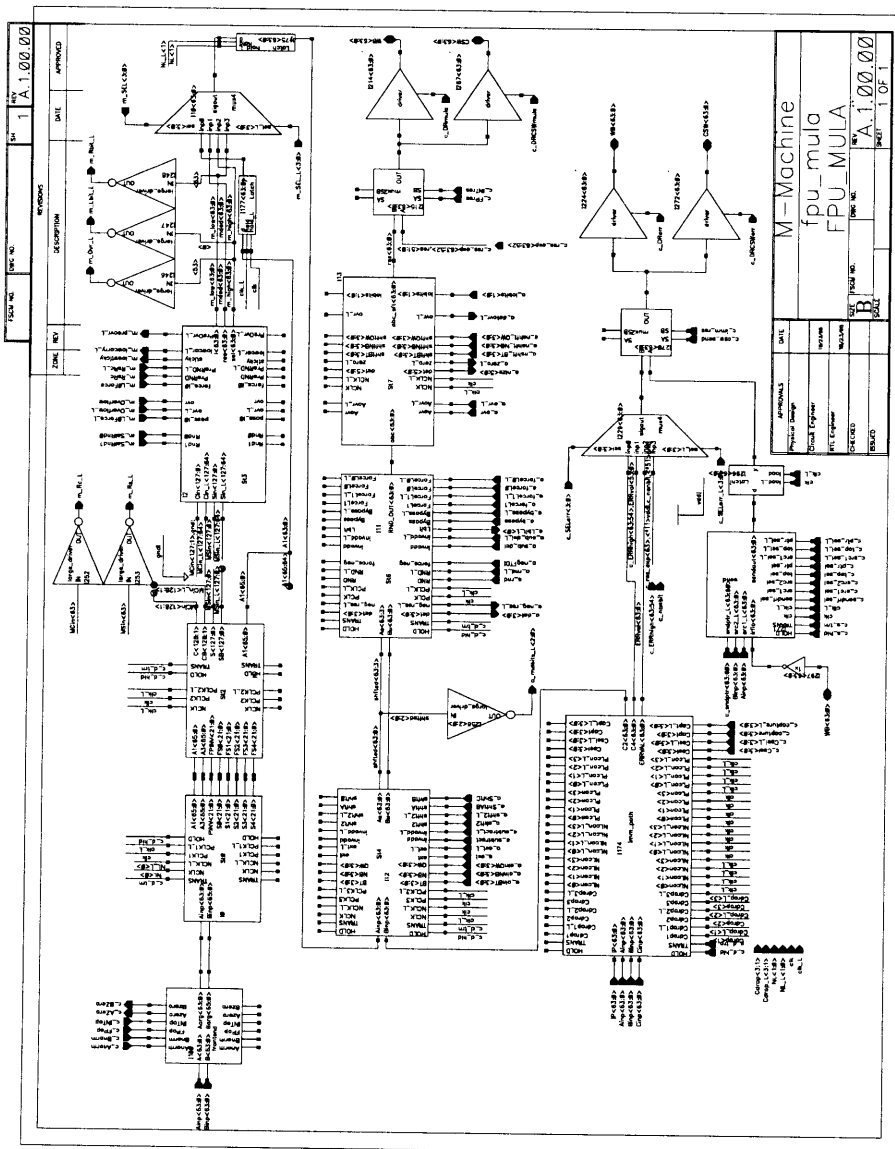


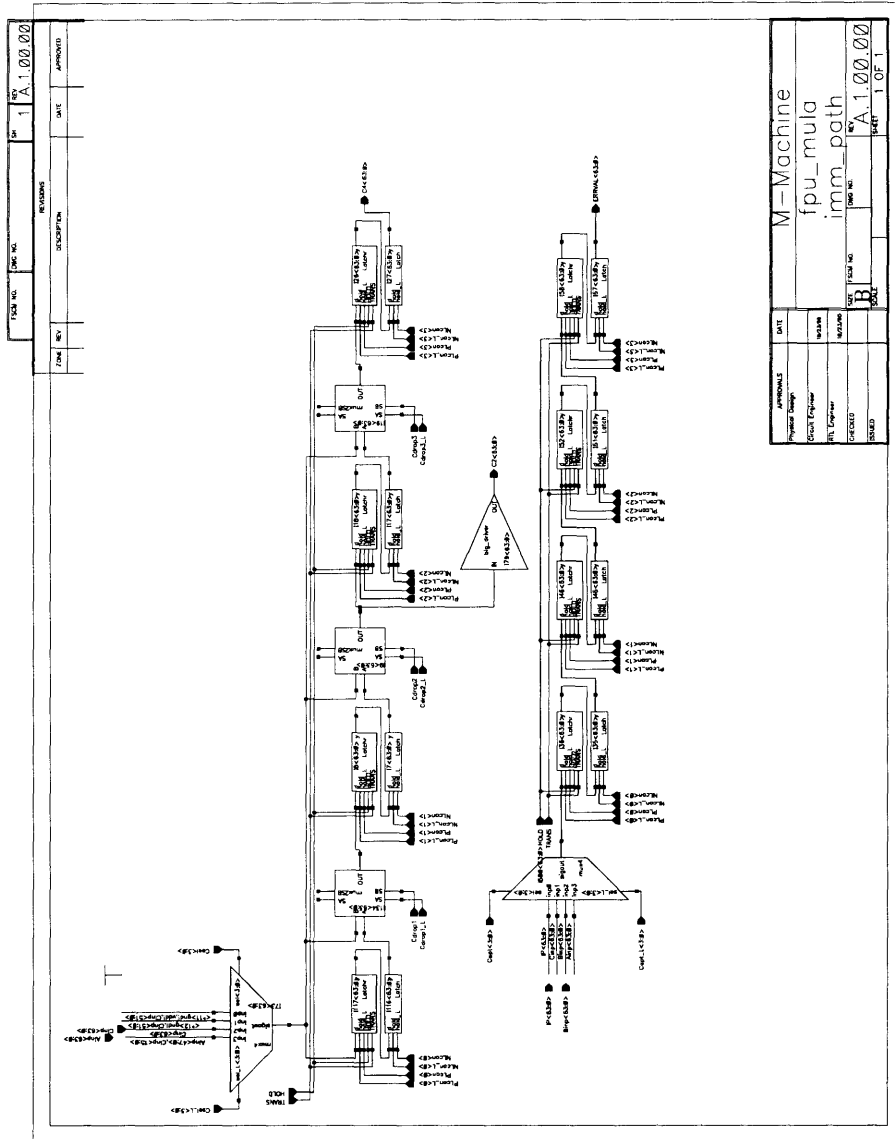
Figure C-2: *Folded 2 Input XOR gates for DCMOS, DCVSL, and HDCVSL families*

Appendix D

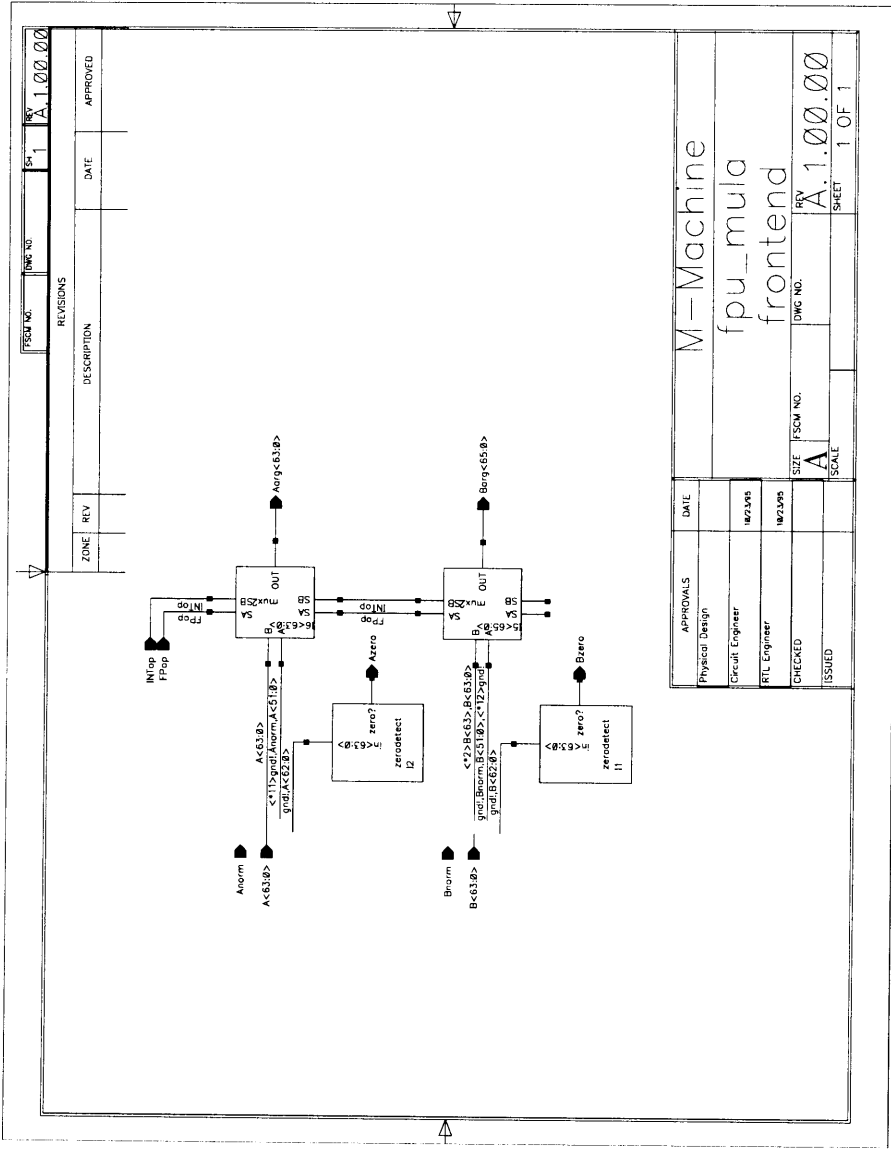
Schematics



D.1 Immediate Path and Front End



APPROVALS	DATE
Project Design	
Circuit Engineer	
Test Engineer	
Checked	
Drawn	
Doc No.	
Rev	
Doc No.	A.1.00.00
Sheet	1 OF 1



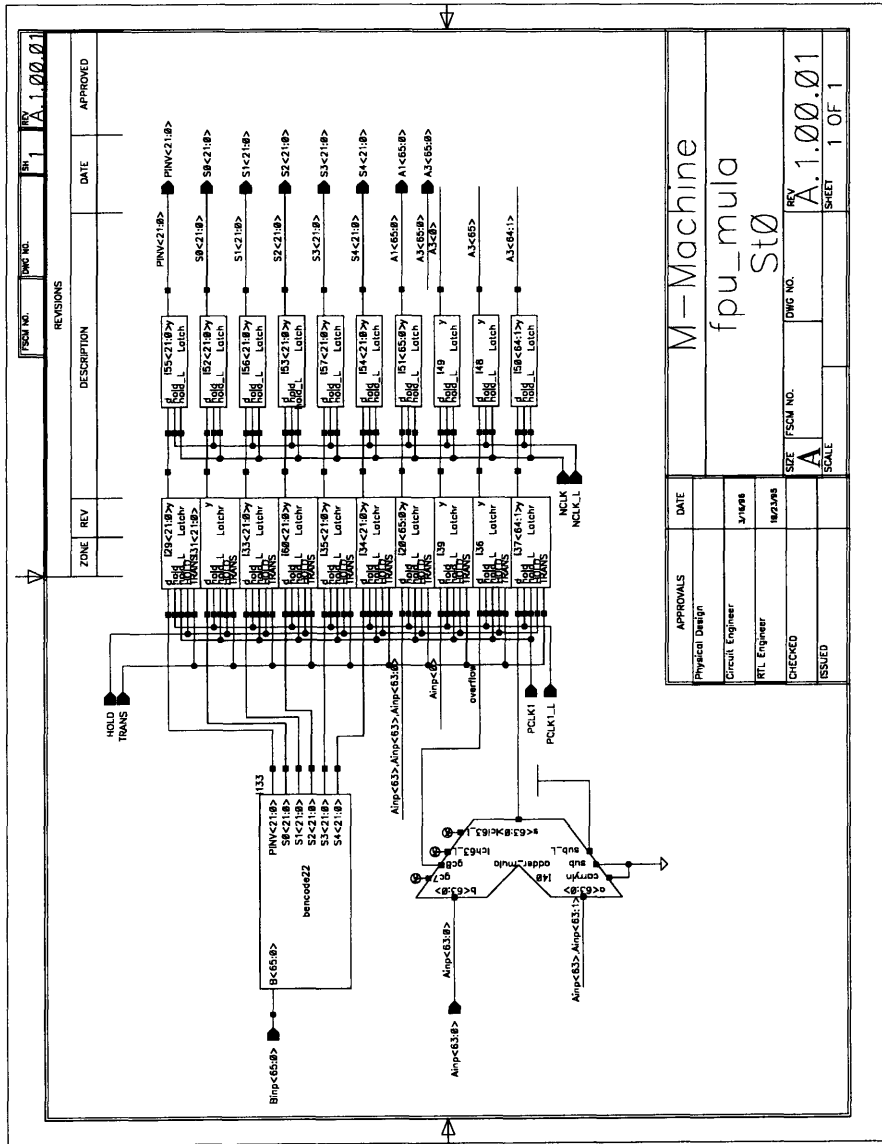
REVISIONS	
ZONE	REV

DESCRIPTION	DATE	APPROVED

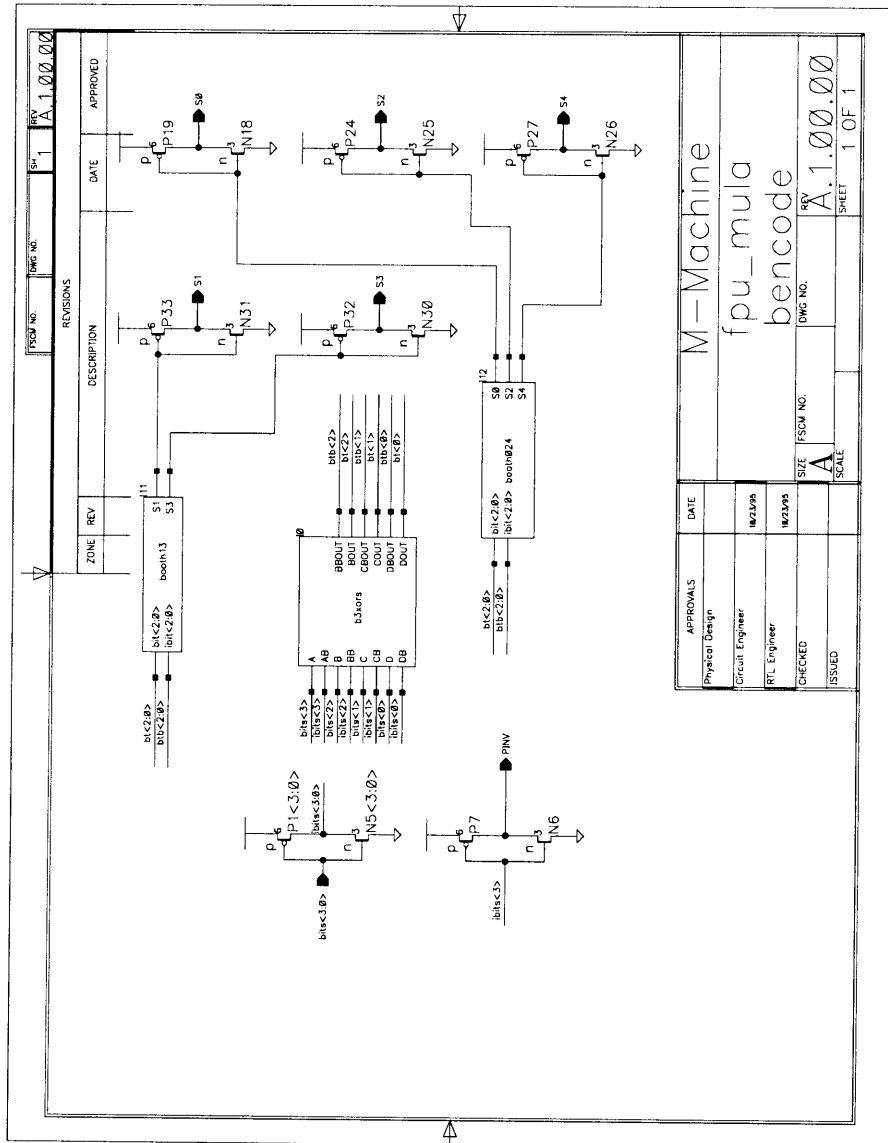
APPROVALS	DATE
Physical Design	
Circuit Engineer	10/25/95
RTL Engineer	10/25/95
CHECKED	
ISSUED	

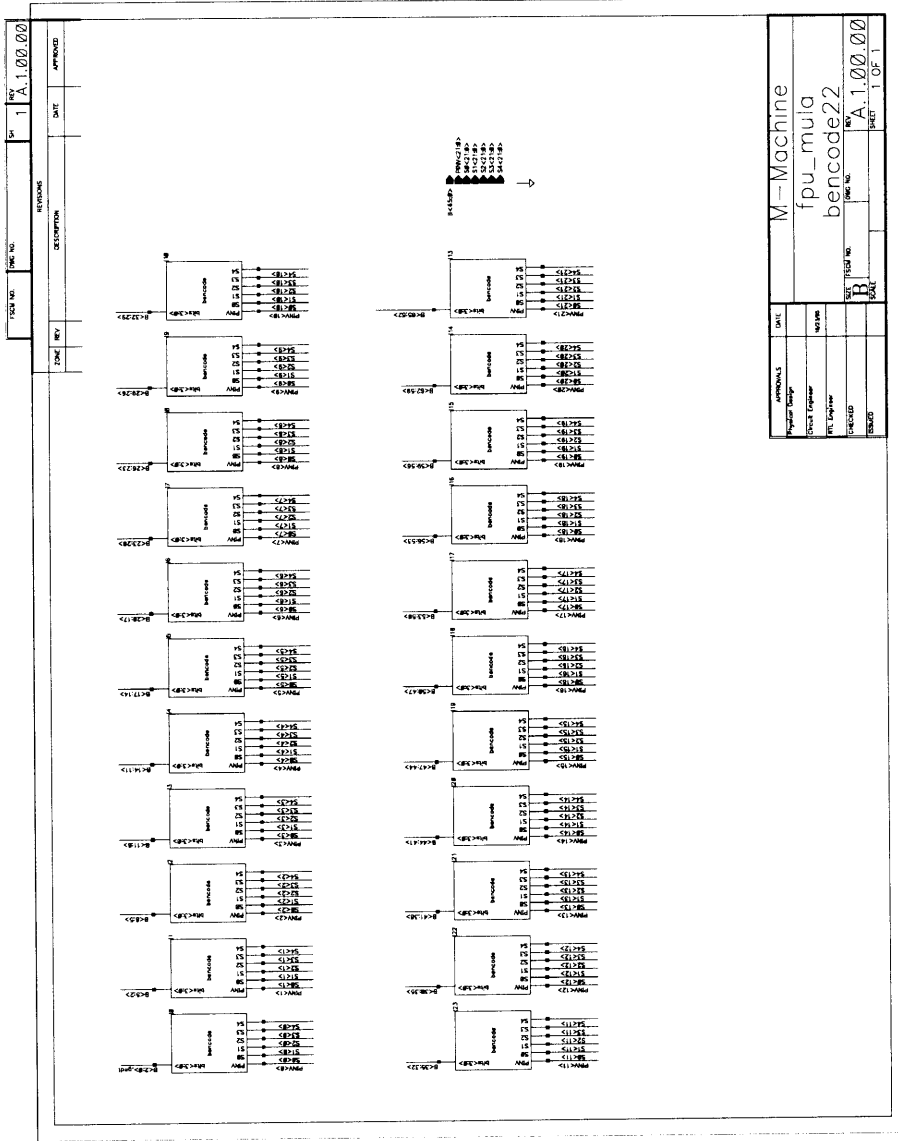
M-Machine	
fpu_mula	
frontend	
SIZE	SCALE
1000	A
DWG NO.	REV
	A.1.00.00
SHEET 1 OF 1	

D.2 Stage 0

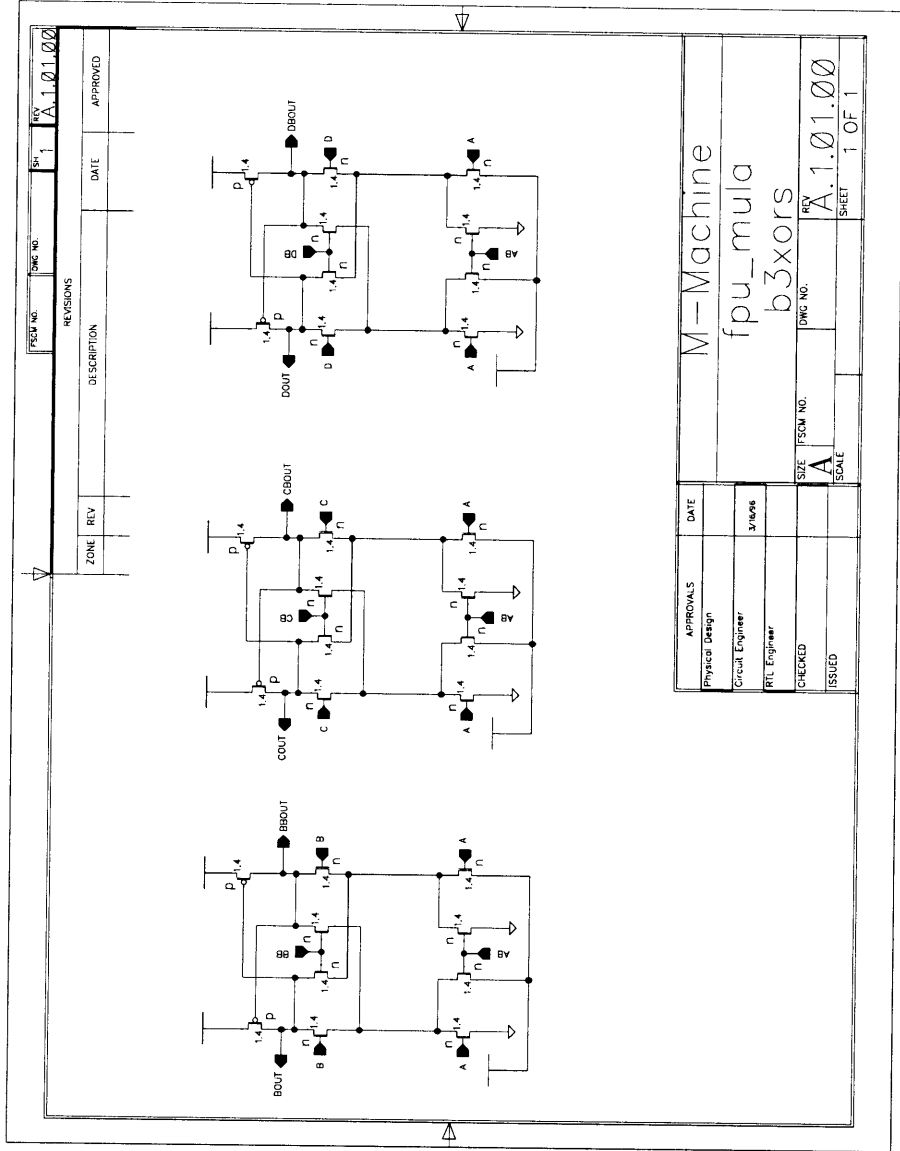


D.2.1 Booth Encoding Cells





M-Machine	
fpu_muia_bencode22	
REV: A.1.00.00	PAGE: 1 OF 1

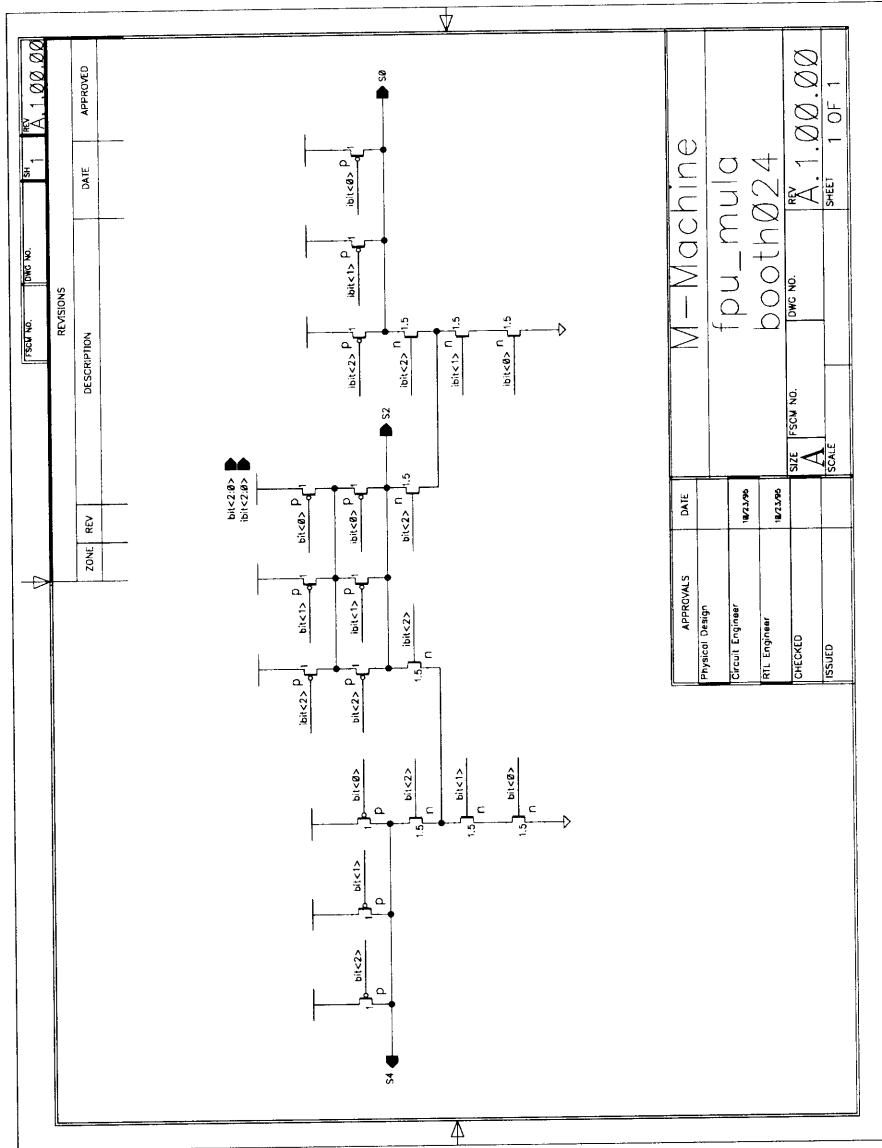


FSOM NO.	REV.	DATE	APPROVED
	1		A.1.01.00
REVISIONS			
ZONE	REV	DESCRIPTION	DATE

APPROVALS		DATE
Physical Design		
Circuit Engineer		2/10/96
RTL Engineer		
CHECKED		
ISSUED		

M-Machine
 fpu_mula
 b3xors

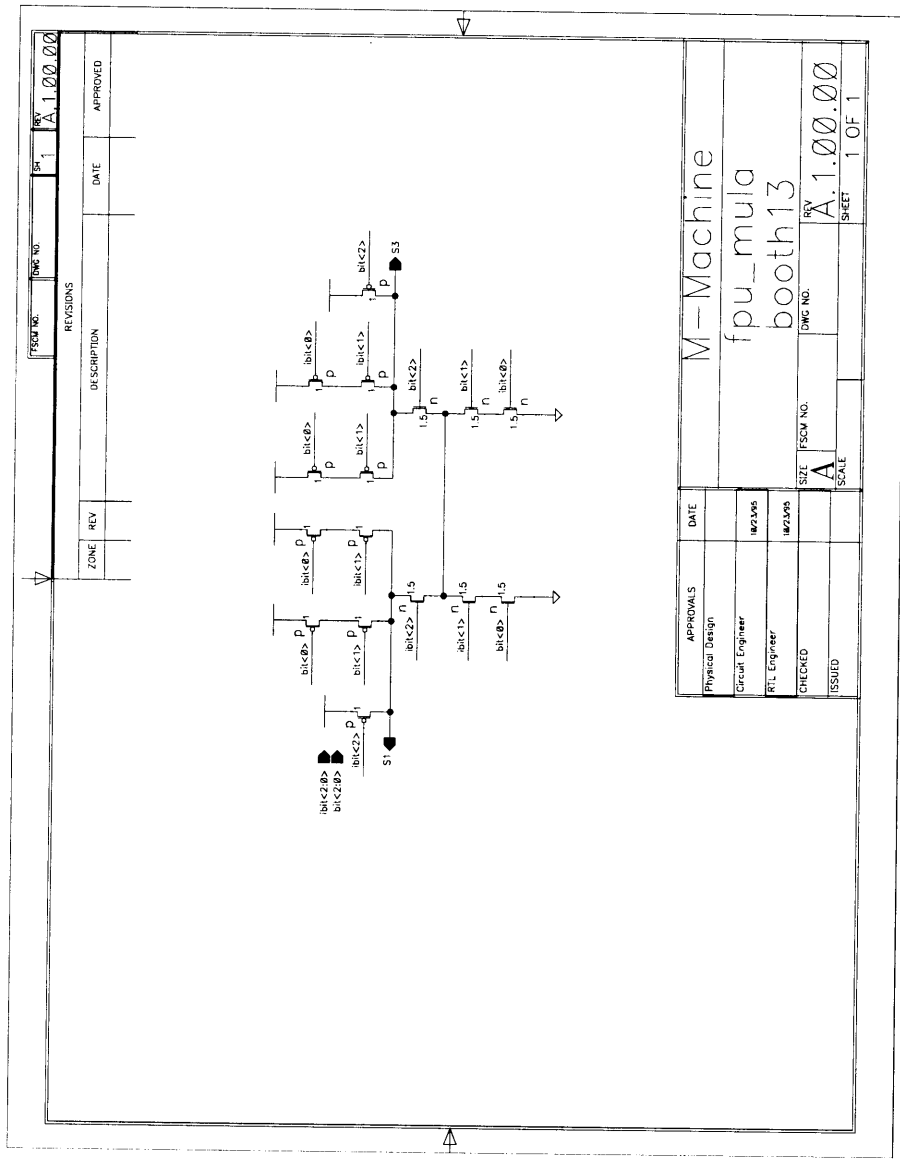
SIZE: A
 FSOM NO.:
 DWG NO.: A.1.01.00
 SCALE: 1 OF 1



FSCM No. A.1.00.00		REV. 1	
ZONE		DATE	
REV		APPROVED	
DESCRIPTION			
REVISIONS			

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/90
RTL Engineer	10/23/90
CHECKED	
ISSUED	

M-Machine	
fpu_mula	
booth024	
SIZE	DWG NO.
A	A.1.00.00
SCALE	SHEET
	1 OF 1



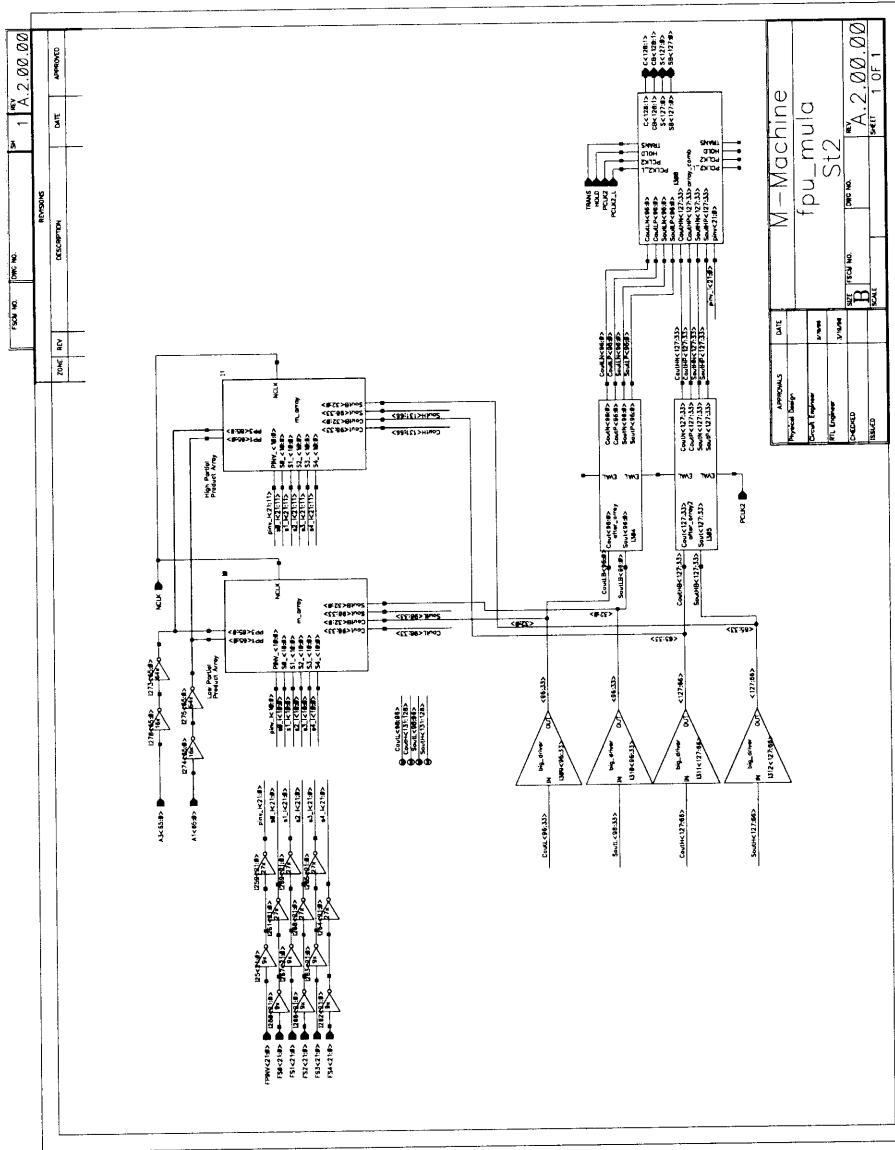
FSCM NO.		DWG NO.		REV		DATE		APPROVED	
		DWG 1		A.1.00.00					

REVISIONS		
ZONE	REV	DESCRIPTION

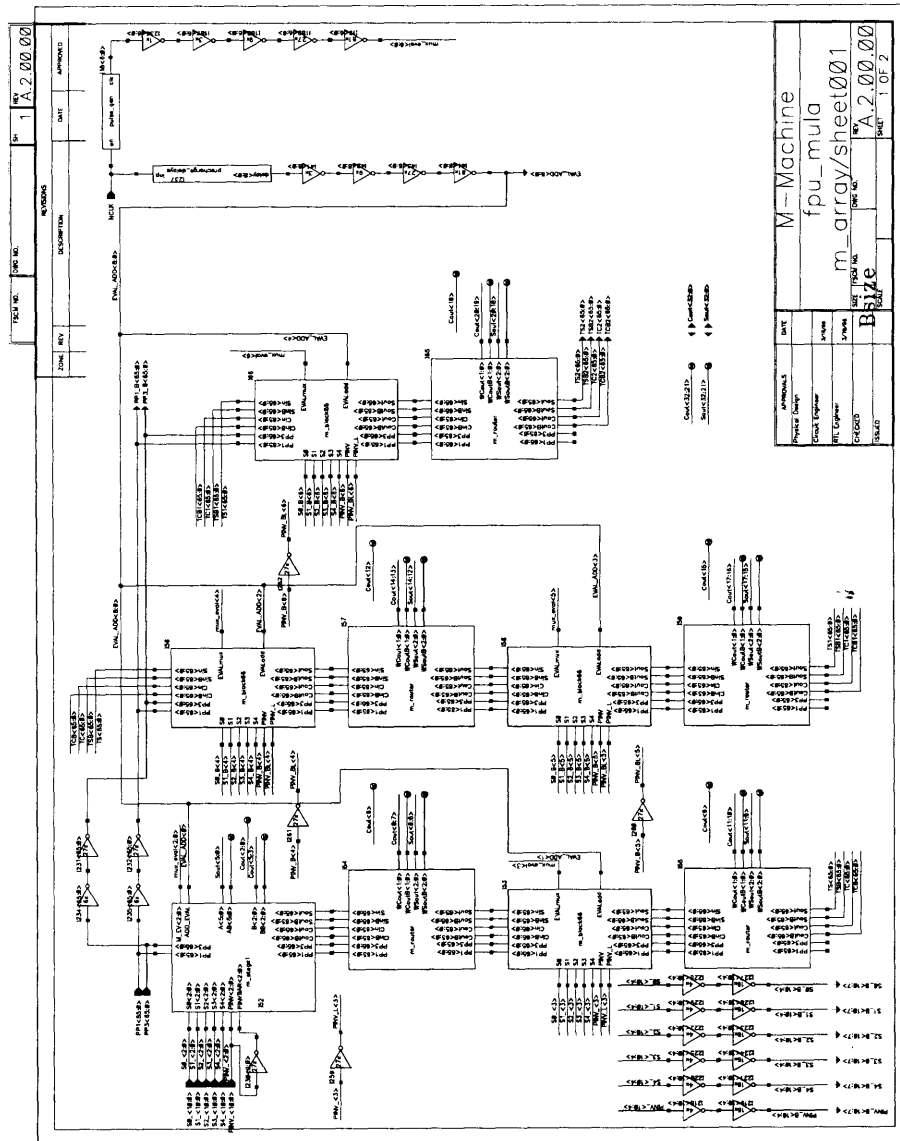
APPROVALS	DATE
Physical Design	
Circuit Engineer	10/2/95
RTL Engineer	10/2/95
CHECKED	
ISSUED	

M-Machine	
fpu_mula	
booth13	
SIZE	FSCM NO.
A	
SCALE	DWG NO.
	A.1.00.00
	REV
	A.1.00.00
	SHEET
	1 OF 1

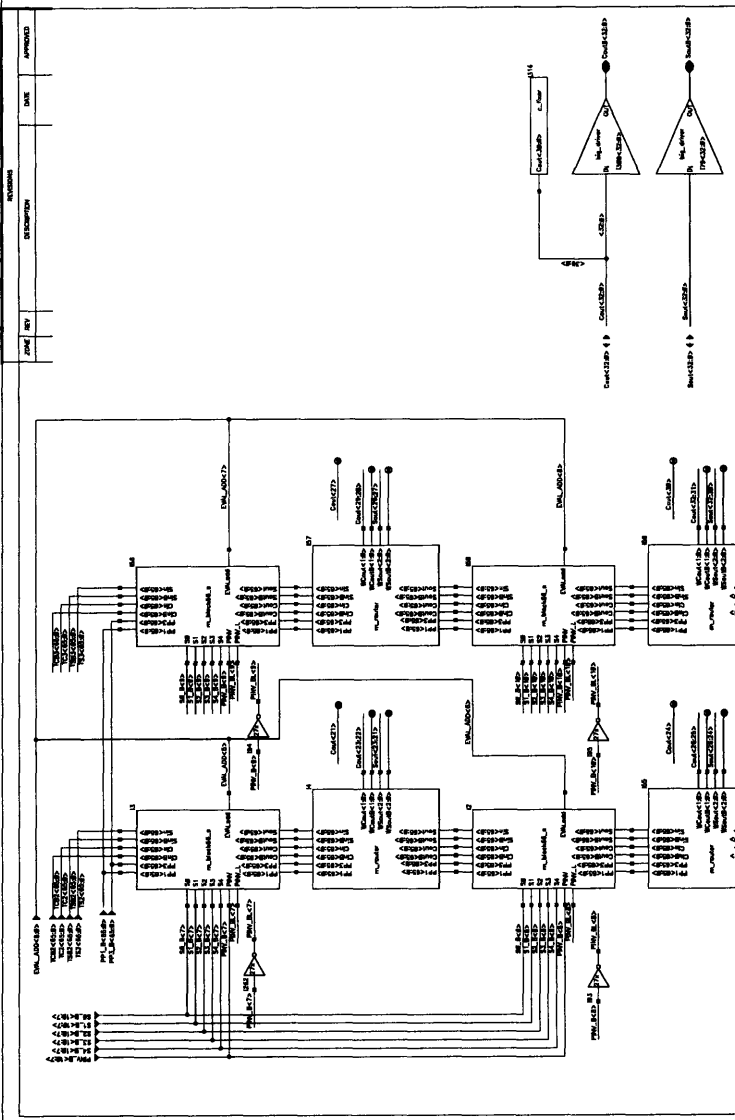
D.3 Stage 2



D.3.1 Multiplier Array and Subcells

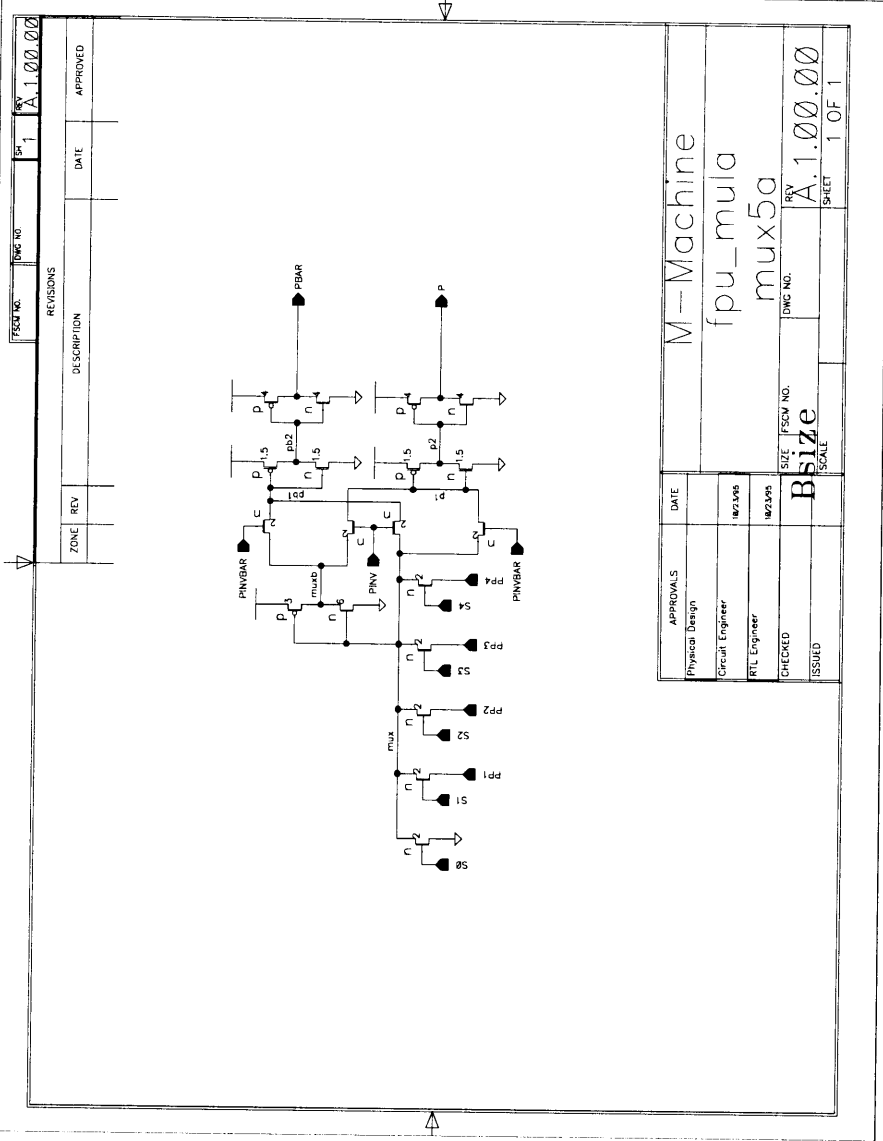


DATE	REV	BY	APPROVED
	2	A.2.00.00	
DATE	REV	BY	APPROVED



DATE	REV	BY	APPROVED
DATE	REV	BY	APPROVED

M-Machine
fpu_mula
m_array/sheet002
A.2.00.00
DATE

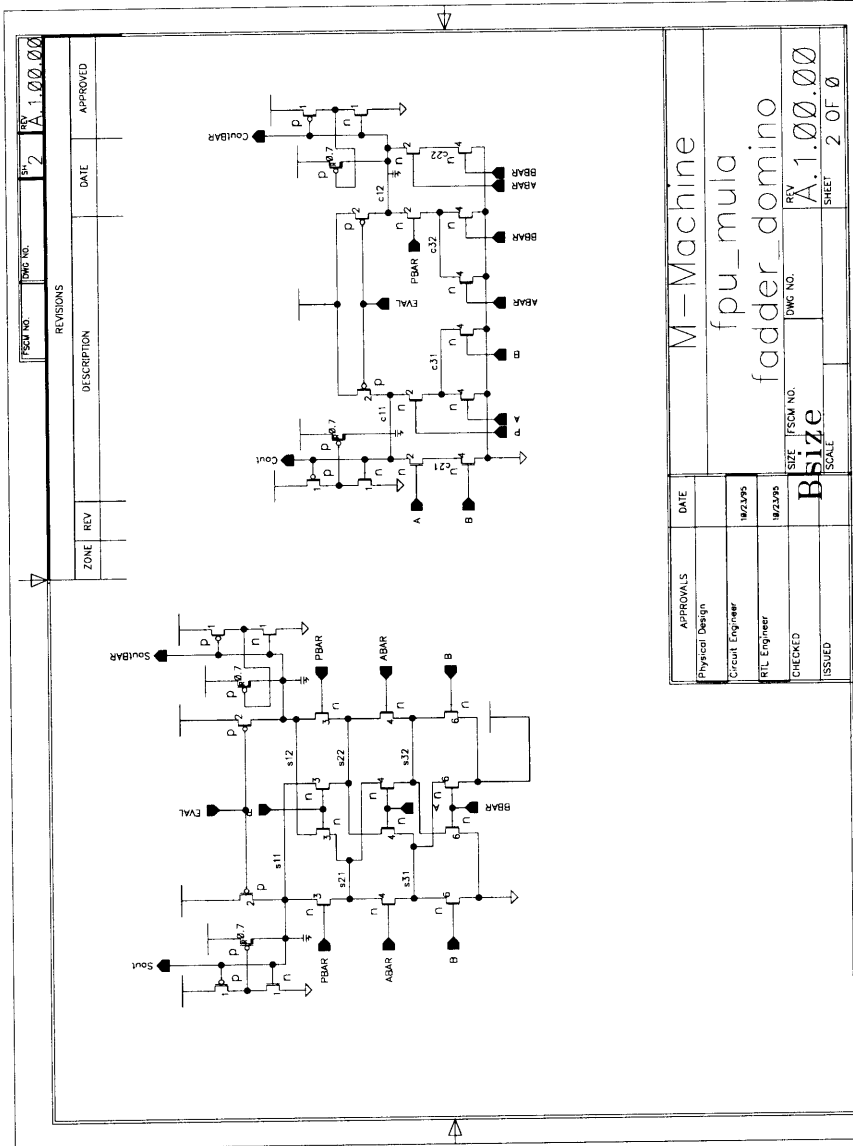


ZONE		REV		DESCRIPTION		DATE		APPROVED	

FIG. NO. DWG. NO.
 SHEET 1 OF 1
 REV. A.1.00.00

APPROVALS	DATE
Physical Design	
Circuit Engineer	14/2/96
RTL Engineer	14/2/96
CHECKED	
ISSUED	

M - Machine
 fpu_mux
 mux5a
 SIZE: FIG. NO. DWG. NO. REV. A.1.00.00
 SCALE: **Bsize** SHEET 1 OF 1

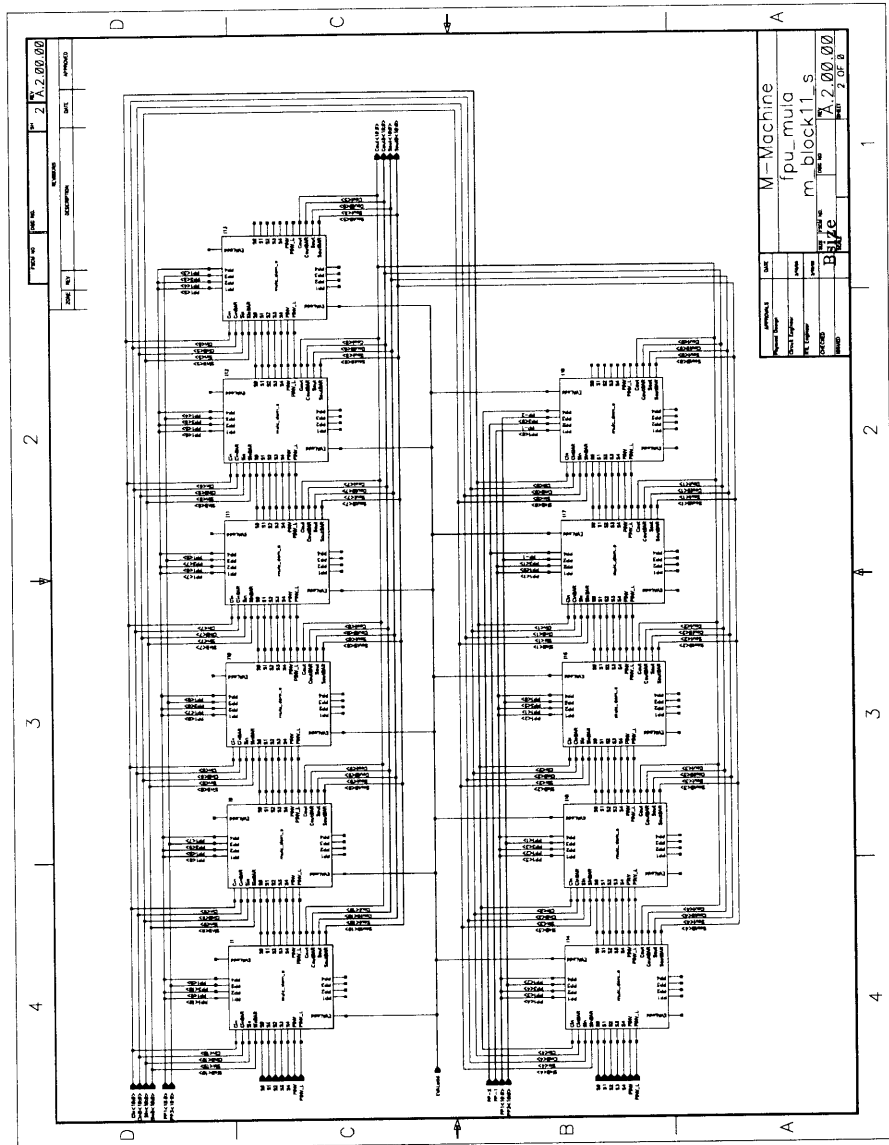


ISSUE NO.	REV.	DATE	APPROVED
	2		
A.1.00.00			

REVISIONS		DATE	APPROVED
ZONE	REV		

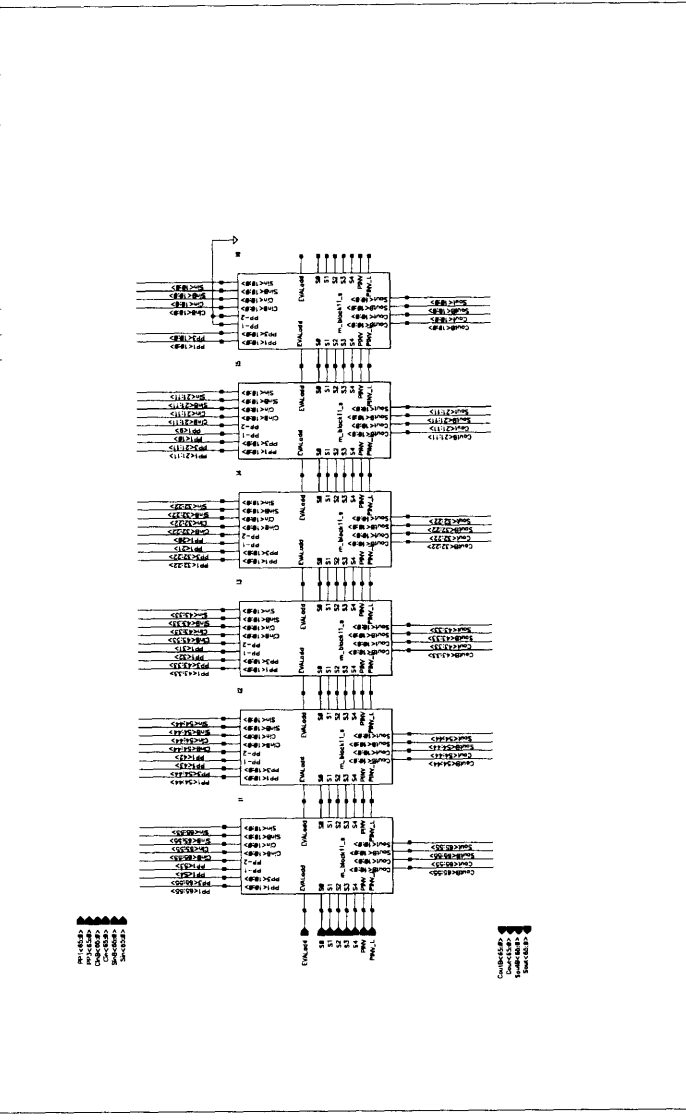
APPROVALS		DATE
Physical Design		
Circuit Engineer		10/23/95
RTL Engineer		10/23/95
CHECKED		
ISSUED		

M-Machine	
fpu_mula	
fadder_domino	
SIZE	ISSUE NO.
Bsize	A.1.00.00
SCALE	SHEET 2 OF 0



DRAWING NO. **21A.2.00.00**
 REVISIONS
 DATE APPROVED

DRAWING NO. **21A.2.00.00**
 REVISIONS
 DATE APPROVED

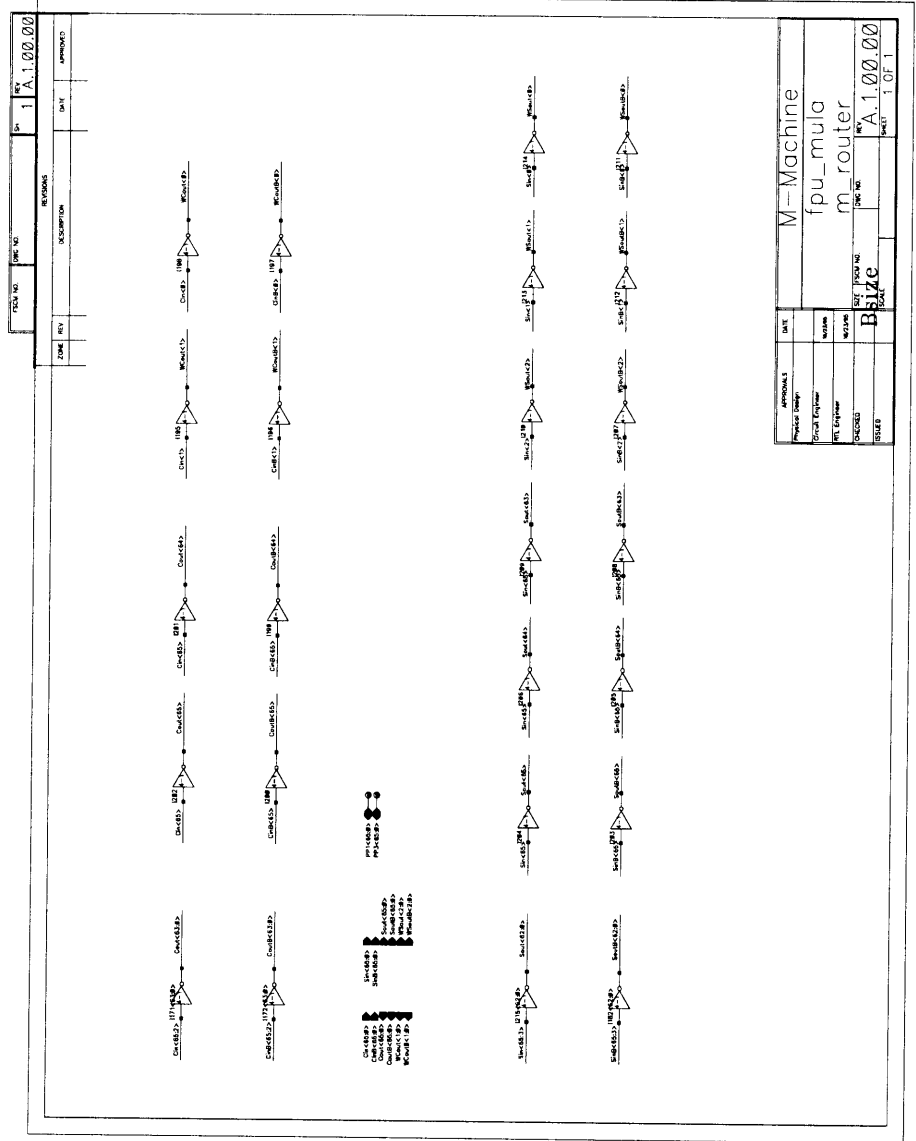


M-Machine
 fpu_mula
 m_block66 s
 DRAWING NO. **21A.2.00.00**
 SHEET **2** OF **0**

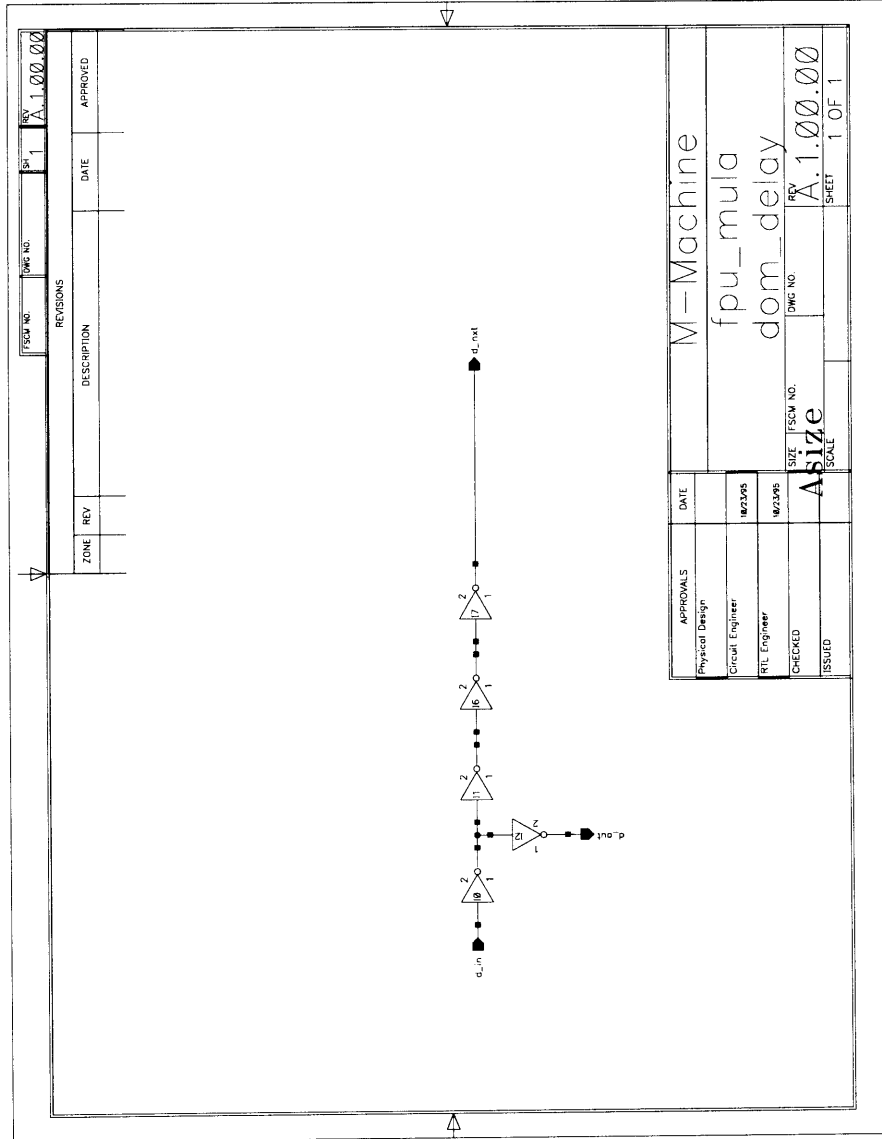
	FSCM NO. 1	DWC NO. 1	REV. 1	A.1.00.00
REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE	M--Machine
Physical Design		share
Circuit Engineer	10/23/95	inv4-1
RTL Engineer	10/23/95	
CHECKED		
ISSUED		

SIZE	FSCM NO.	REV
Bsize		A.1.00.00
SCALE		SHEET 1 OF 1



APPROVALS	DATE	M- Machine	REV.	DATE
Project Manager		fpu_muia		
Project Engineer		m_router		
REV.	DATE	REV.	DATE	REV.
01		01		01
02		02		02
03		03		03
04		04		04
05		05		05
06		06		06
07		07		07
08		08		08
09		09		09
10		10		10
11		11		11
12		12		12
13		13		13
14		14		14
15		15		15
16		16		16
17		17		17
18		18		18
19		19		19
20		20		20
21		21		21
22		22		22
23		23		23
24		24		24
25		25		25
26		26		26
27		27		27
28		28		28
29		29		29
30		30		30
31		31		31
32		32		32
33		33		33
34		34		34
35		35		35
36		36		36
37		37		37
38		38		38
39		39		39
40		40		40
41		41		41
42		42		42
43		43		43
44		44		44
45		45		45
46		46		46
47		47		47
48		48		48
49		49		49
50		50		50
51		51		51
52		52		52
53		53		53
54		54		54
55		55		55
56		56		56
57		57		57
58		58		58
59		59		59
60		60		60
61		61		61
62		62		62
63		63		63
64		64		64
65		65		65
66		66		66
67		67		67
68		68		68
69		69		69
70		70		70
71		71		71
72		72		72
73		73		73
74		74		74
75		75		75
76		76		76
77		77		77
78		78		78
79		79		79
80		80		80
81		81		81
82		82		82
83		83		83
84		84		84
85		85		85
86		86		86
87		87		87
88		88		88
89		89		89
90		90		90
91		91		91
92		92		92
93		93		93
94		94		94
95		95		95
96		96		96
97		97		97
98		98		98
99		99		99
100		100		100

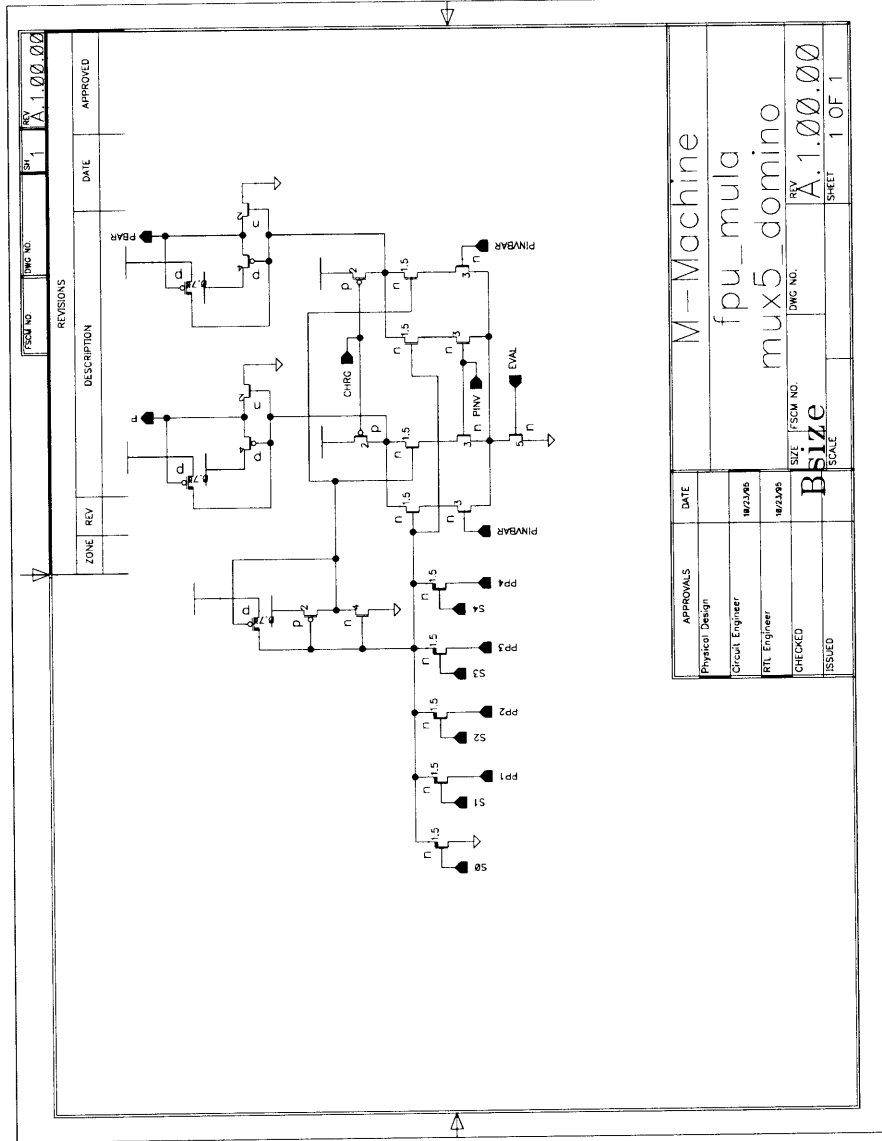


TSCM NO.		DWG NO.		REV	

ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/95
PLC Engineer	10/23/95
CHECKED	
TESTED	

M-Machine	
fpu_mula	
dom_delay	
SIZE	TSCM NO.
Asize	
DWG NO.	REV
	A.1.00.00
SHEET	1 OF 1

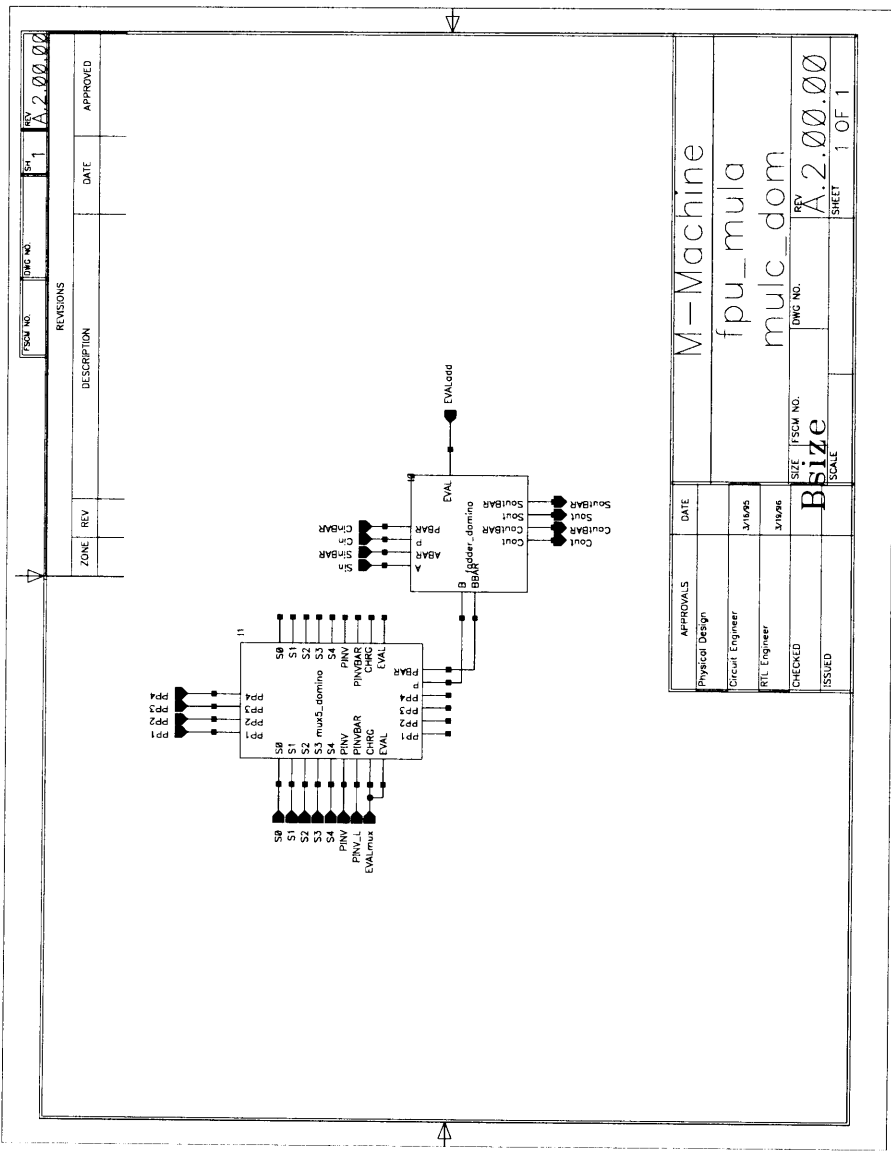


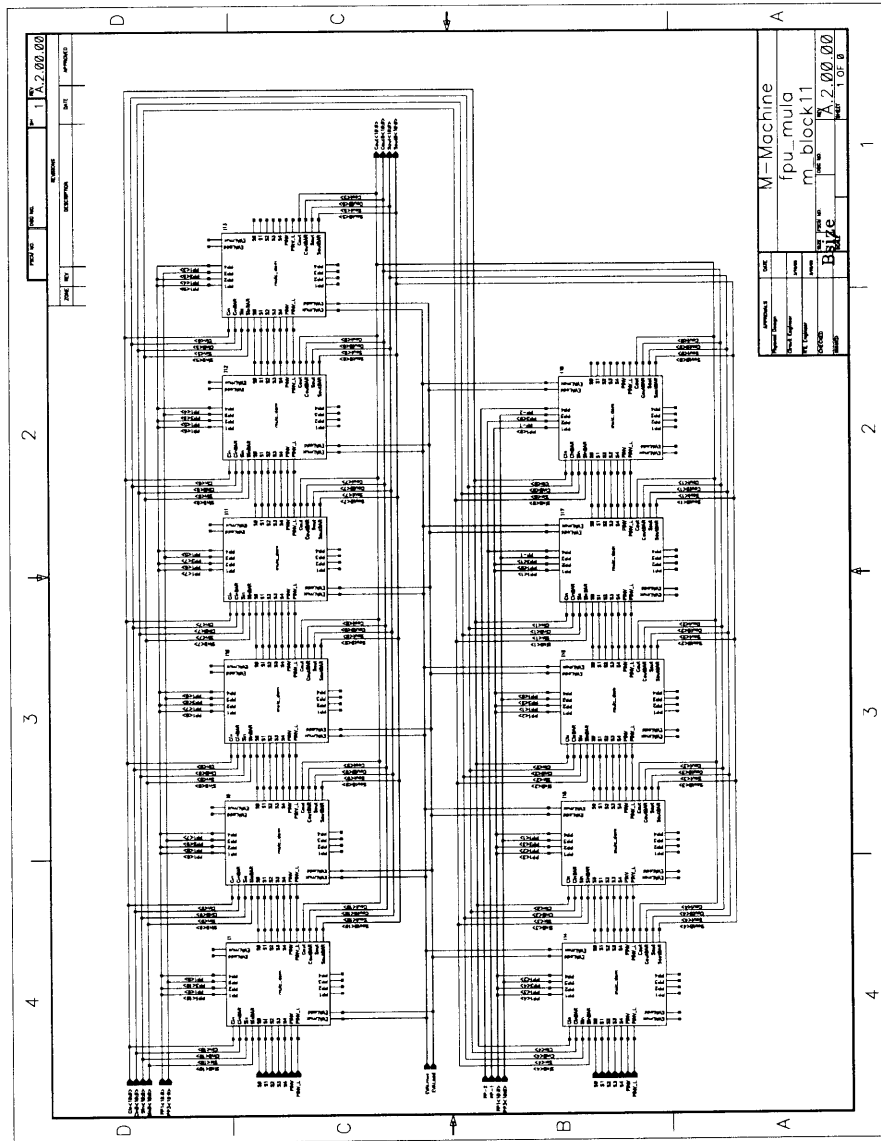
FSCM NO.	SN	REV
	1	A.1.00.00

ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
Physical Design	
Circuit Engineer	11/22/95
RTL Engineer	11/22/95
CHECKED	
ISSUED	

M -- Machine	
fpu_mula	
mux5_domino	
SIZE	FSCM NO.
Bsize	
DWG NO.	REV
	A.1.00.00
SHEET	1 OF 1





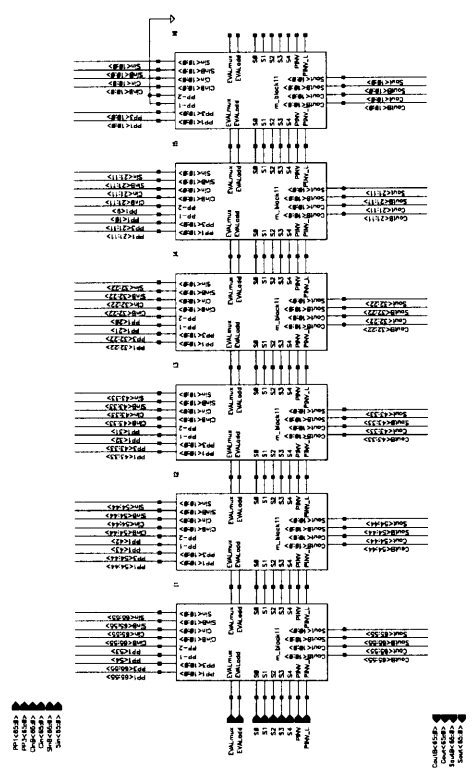
PROJECT NO.	1	A. 2. 000. 000
DATE		
REVISIONS		
APPROVED		

PROJECT NO.	A. 2. 000. 000
DATE	
REVISIONS	
APPROVED	
DESIGNED BY	B. J. C.
CHECKED BY	
DATE	
PAGE	1 OF 8

M-Machine
 rpu_mula
 m_block11

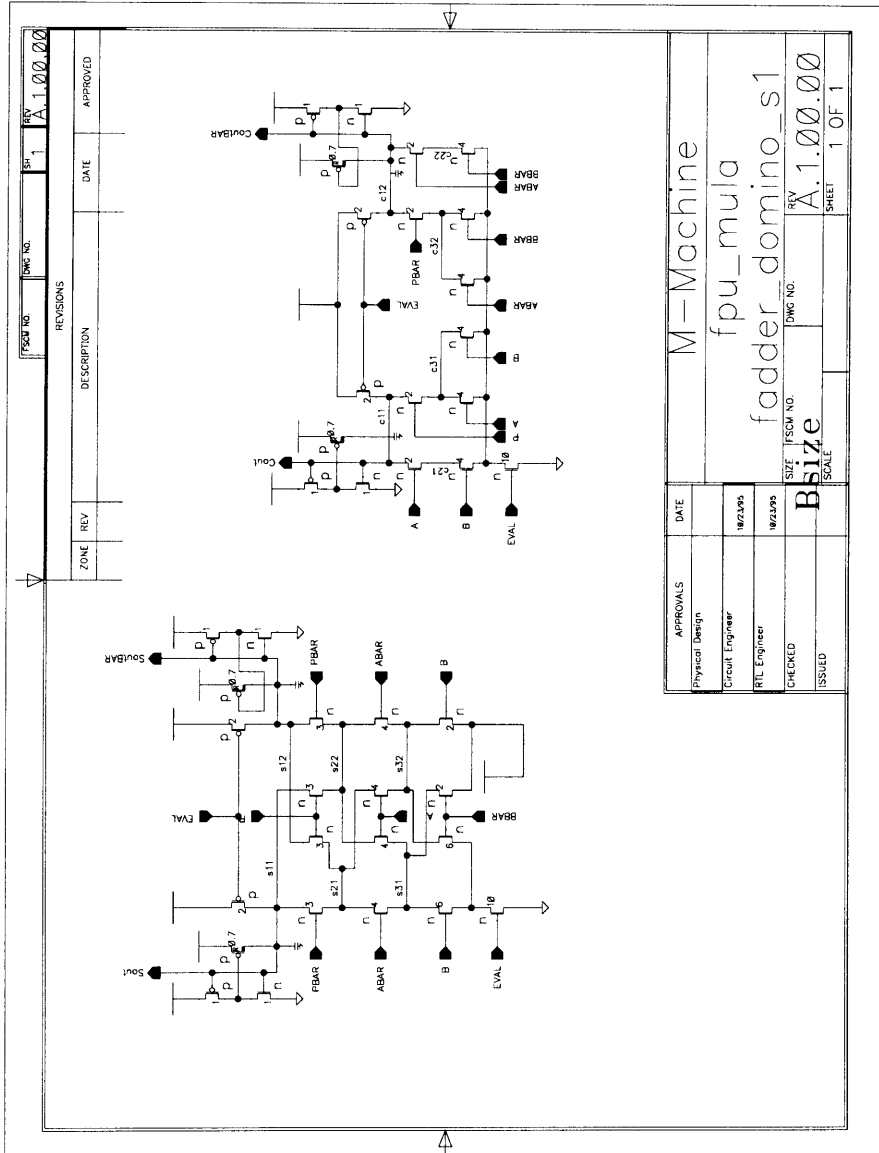
FORM NO.	REV.	DATE	APPROVED
	1	A.2.00.00	

ZONE	REV.	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
Prepared By	
Checked By	
Rev. Checked	
Rev. Approved	
Rev. Released	
Rev. Scaled	
Rev. Size	
Rev. 1 OF 8	

M - Machine
 fpu_mula
 m_block66



REV A.1.00.00
 DWG NO. 1
 FSCM NO.

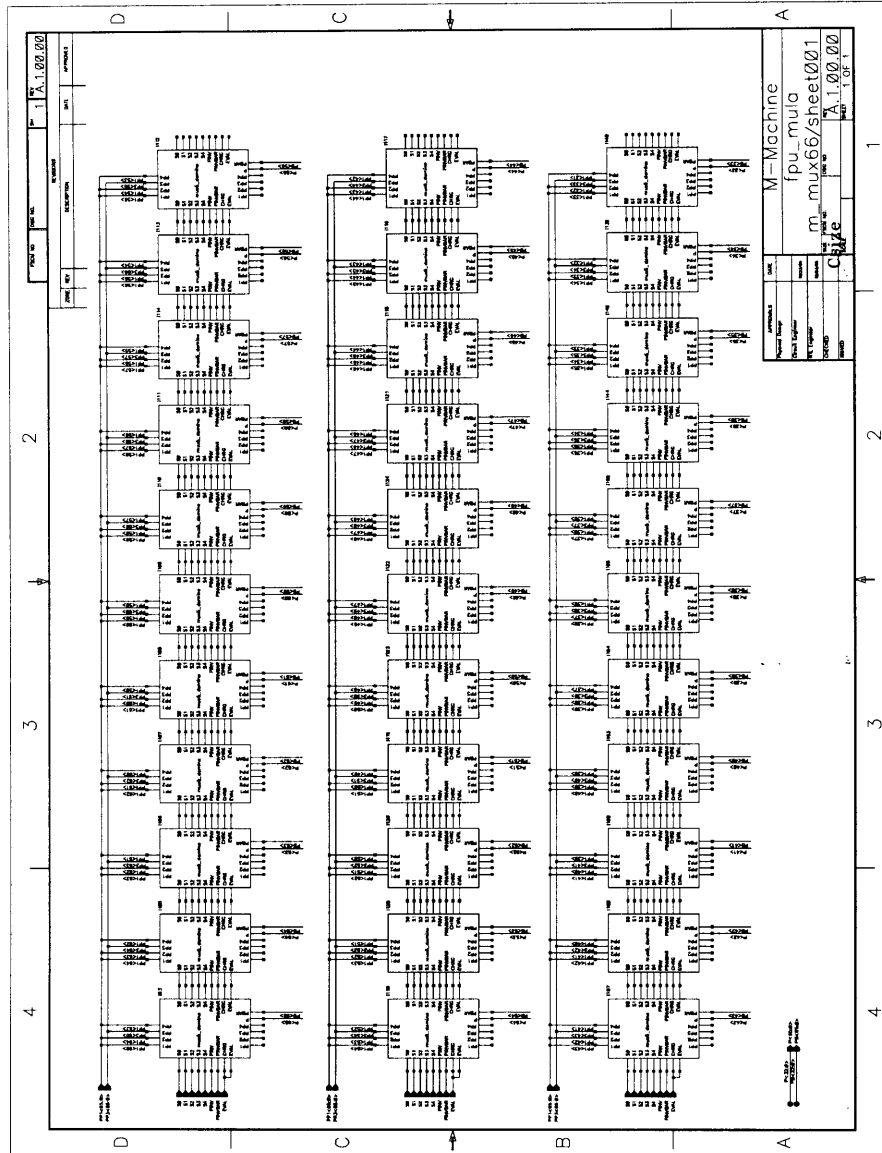
REVISIONS
 DESCRIPTION
 DATE
 APPROVED

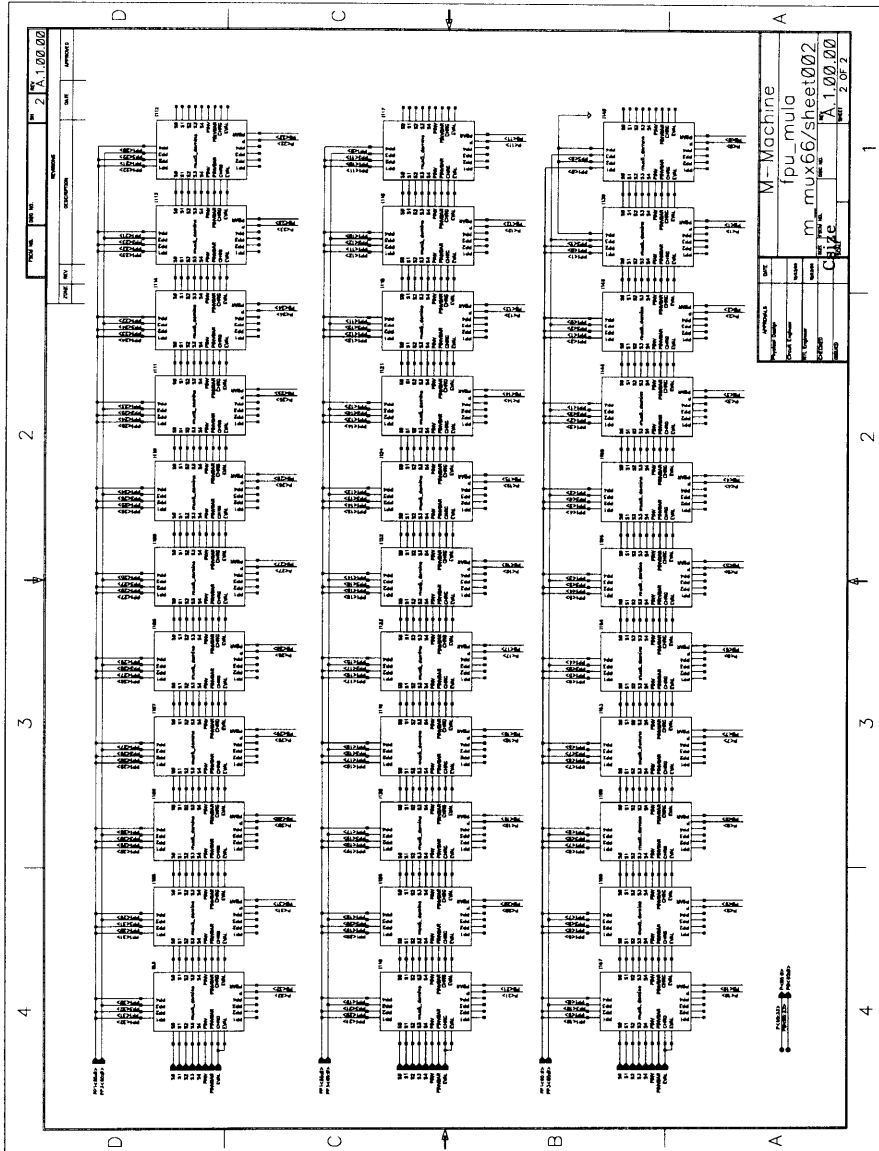
ZONE REV

APPROVALS	DATE
Physical Design	
Circuit Engineer	18/2/2005
RTL Engineer	18/2/2005
CHECKED	
ISSUED	

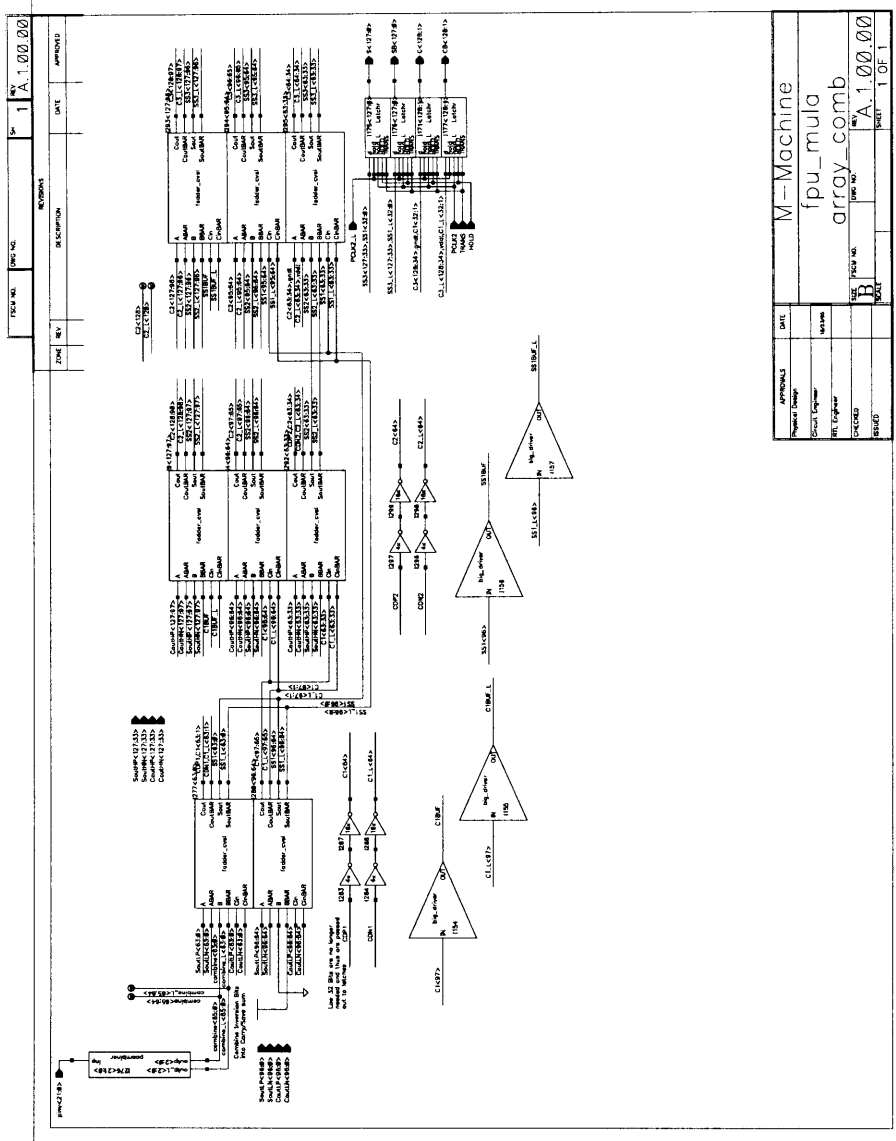
M-Machine
 fpu_mu1a
 fadder_domino_s1
 FSCM NO.
 DWG NO.
 REV A.1.00.00
 SHEET 1 OF 1

SIZE Bsize
 SCALE





D.3.2 Array Combination and Subcells



APPROVALS	DATE
Project Design	
Circuit Engineer	
RTL Engineer	
Checked	
Designed	
M-Machine	
fpu_mula	
array_comb	
SIZE	FIGURE NO.
B	A.1.00.00
SHEET	1 OF 1

FSCM NO.	DWG NO.	REV.	DATE	APPROVED	
		B*			A.1.00.00
REVISIONS					
ZONE	REV	DESCRIPTION	DATE		

The diagram shows a circuit with two input terminals labeled 'ouip' and 'ouip_L'. The 'ouip' input is connected to the base of a transistor. The 'ouip_L' input is connected to the emitter. The collector of the transistor is connected to an output terminal labeled 'ouip'. The emitter is also connected to an output terminal labeled 'ouip_L'. The circuit is powered by a supply 'P'.

M --- Machine					
fpu_mula					
pcombiner2					
FSCM NO.	DWG NO.	REV.	DATE	APPROVED	SHEET
					1 OF 1
SIZE	SCALE				
A	A				

APPROVALS	DATE
Physical Design	
Circuit Engineer	3/18/05
RTL Engineer	3/18/06
CHECKED	
ISSUED	

FROM NO.	DWG NO.	REV.	REV.	REV.	REV.
		1	1	1	1
		A.1.00.00	A.1.00.00	A.1.00.00	A.1.00.00

ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
Physical Design	
Circuit Engineer	J/6/98
RTL Engineer	J/6/98
CHECKED	
ISSUED	

M---Machine	DWC NO.	REV.	REV.
fpu_mula		1	1
pcombiner1		A.1.00.00	A.1.00.00
SIZE	SCALE	SHEET 1 OF 1	
A			

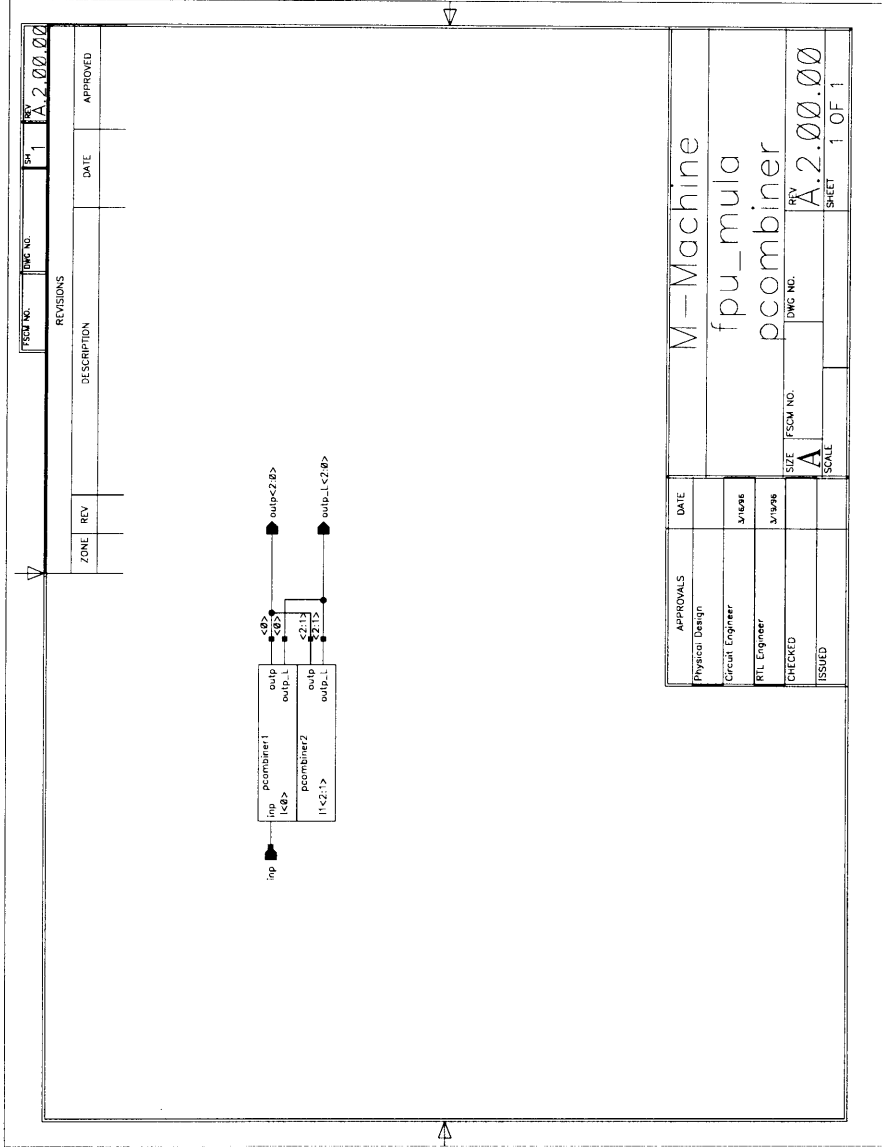
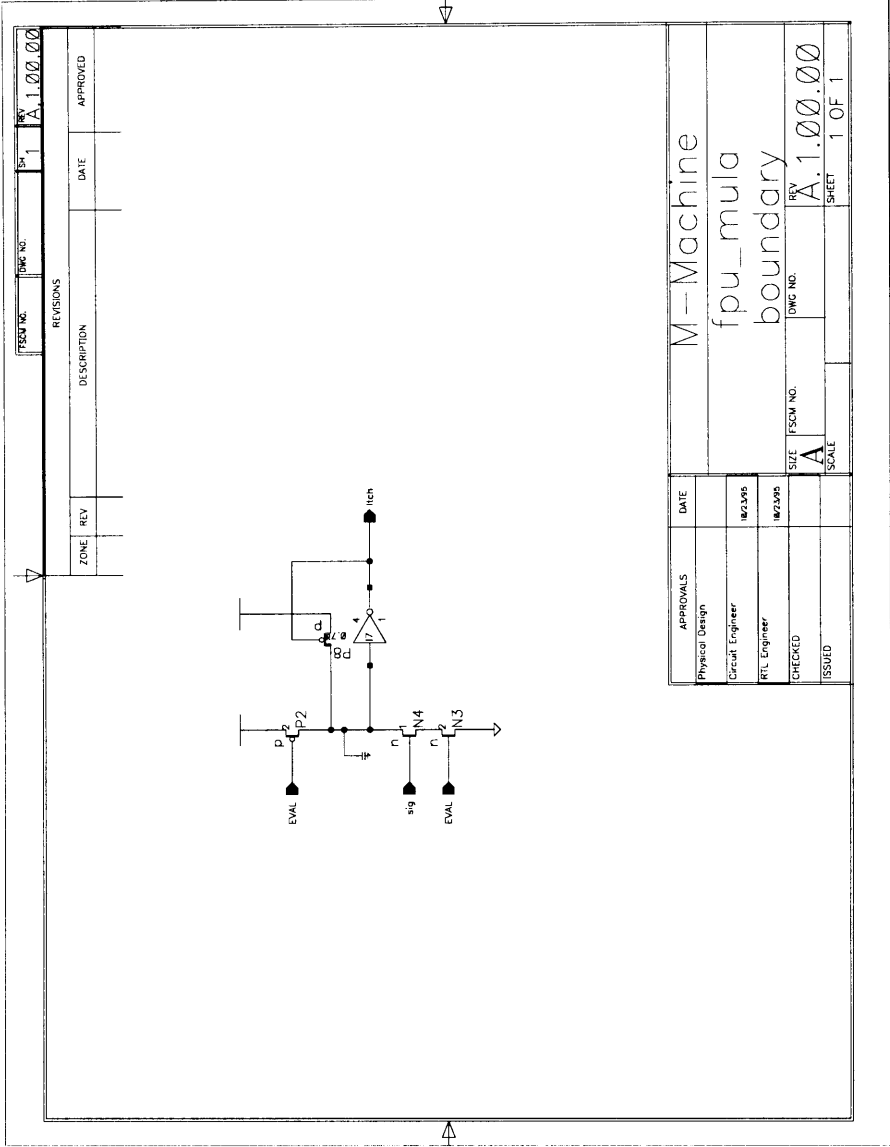


FIG. NO.	DWG. NO.	REV.	DATE	APPROVED
		1		A.2.00.00

REVISIONS		
ZONE	REV	DATE

APPROVALS		DATE
Physical Design		
Circuit Engineer		3/16/98
RTL Engineer		3/19/98
CHECKED		
ISSUED		

M-Machine	
fpu_mula	
pcombiner	
SIZE	FIG. NO.
A	
SCALE	REV.
	A.2.00.00
	SHEET
	1 OF 1



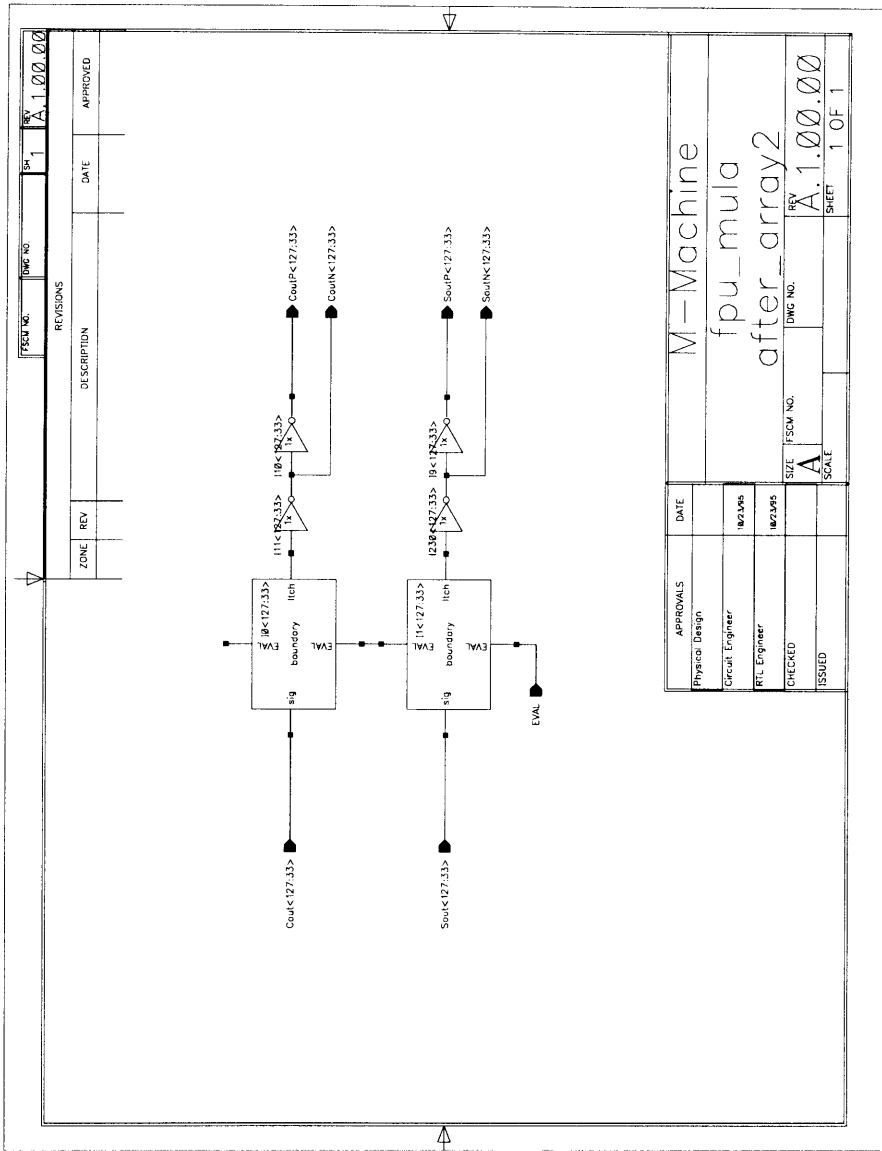
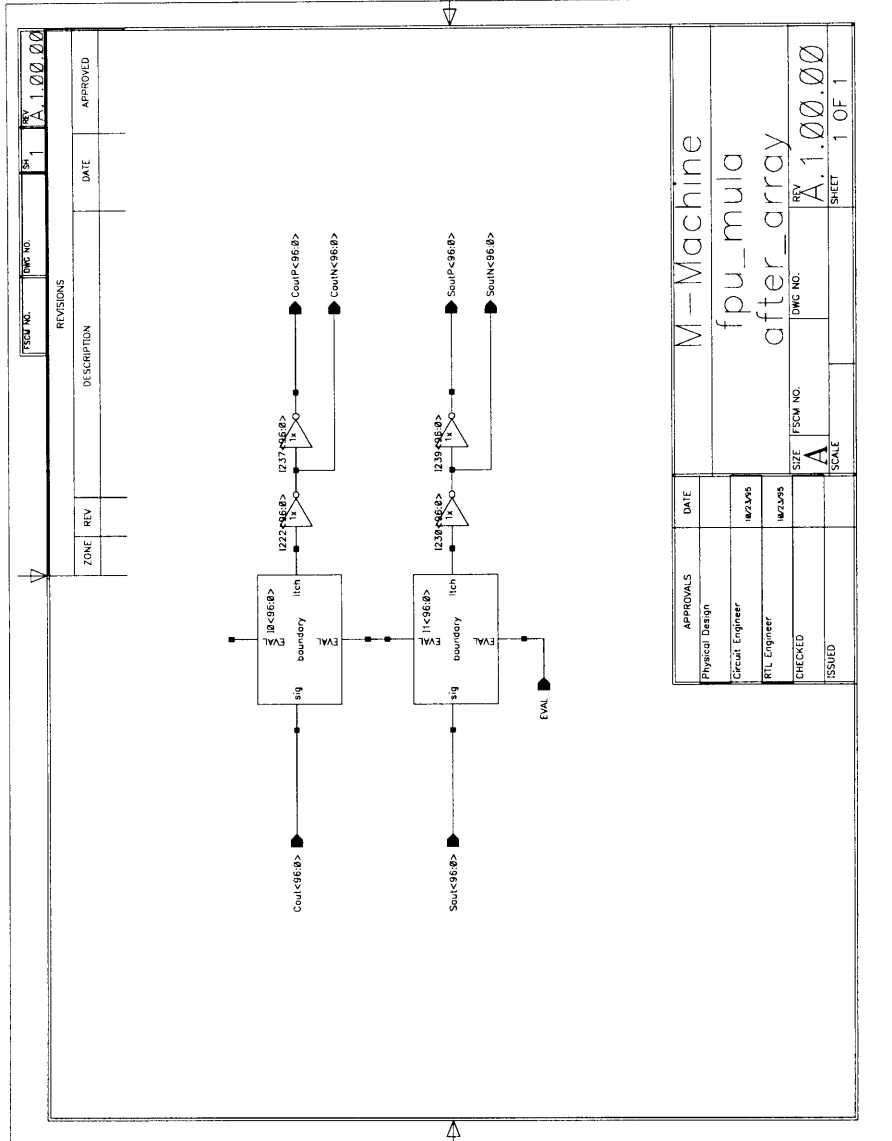


FIGURE NO.	DWG NO.	SM	1	REV	A.1.00.00
------------	---------	----	---	-----	-----------

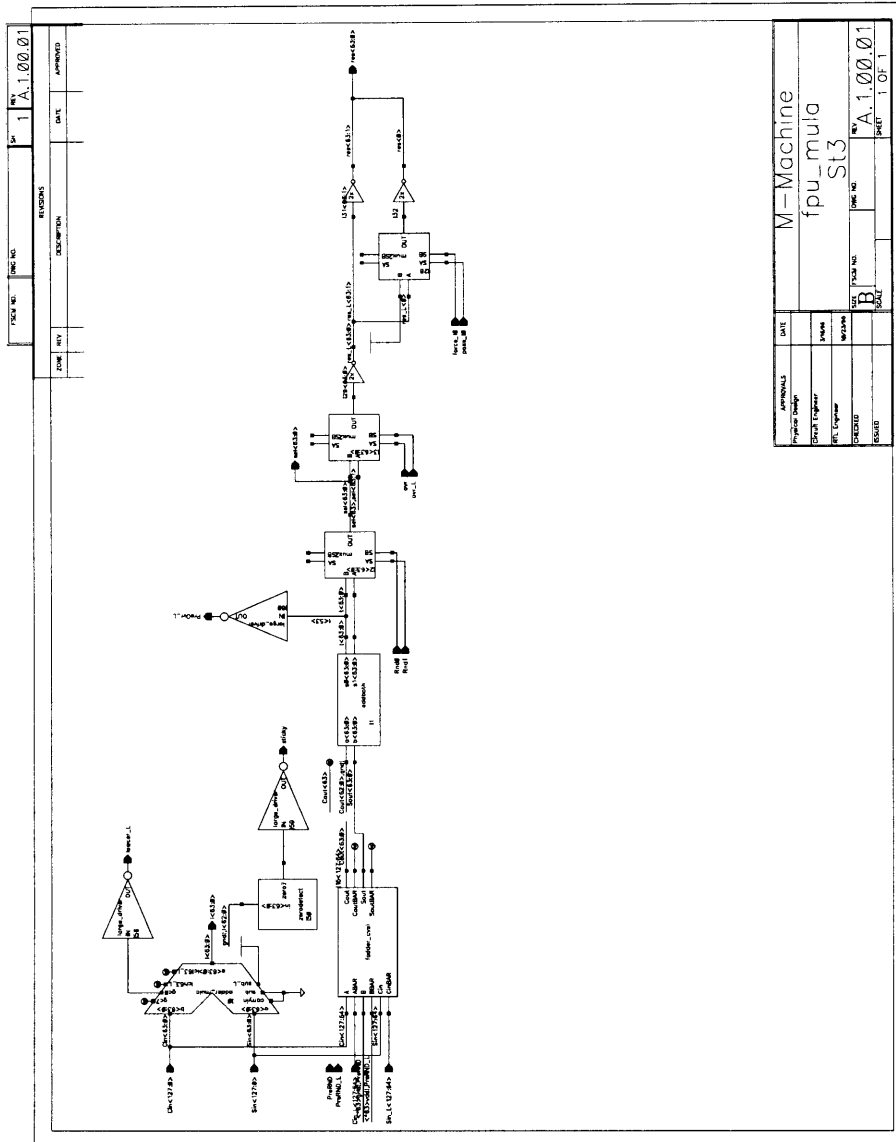
REVISIONS			DATE	APPROVED
ZONE	REV	DESCRIPTION		

APPROVALS	DATE
Physico Design	
Circuit Engineer	16/2/95
RTL Engineer	16/2/95
CHECKED	
ISSUED	

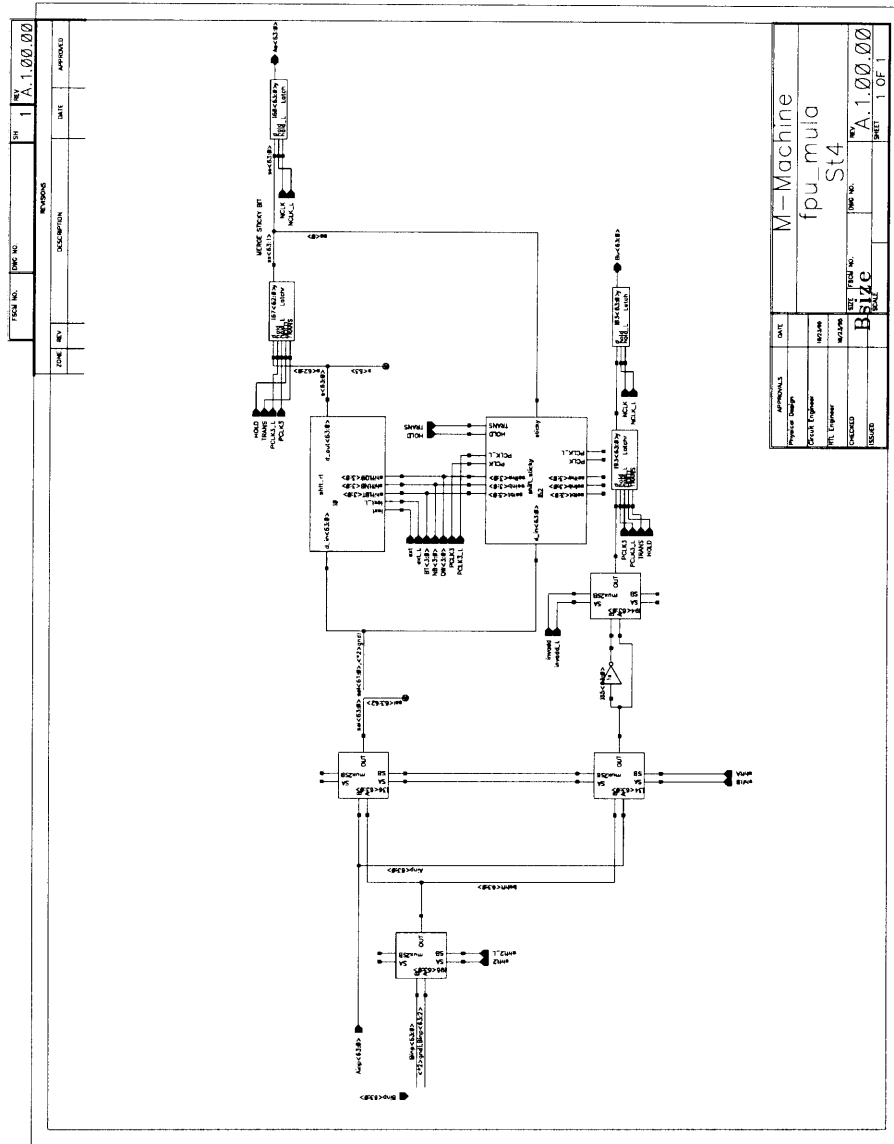
M-Machine	
fpu_mula	
after_array2	
SIZE	FIGURE NO.
A	
SCALE	REV
	A.1.00.00
	SHEET
	1 OF 1



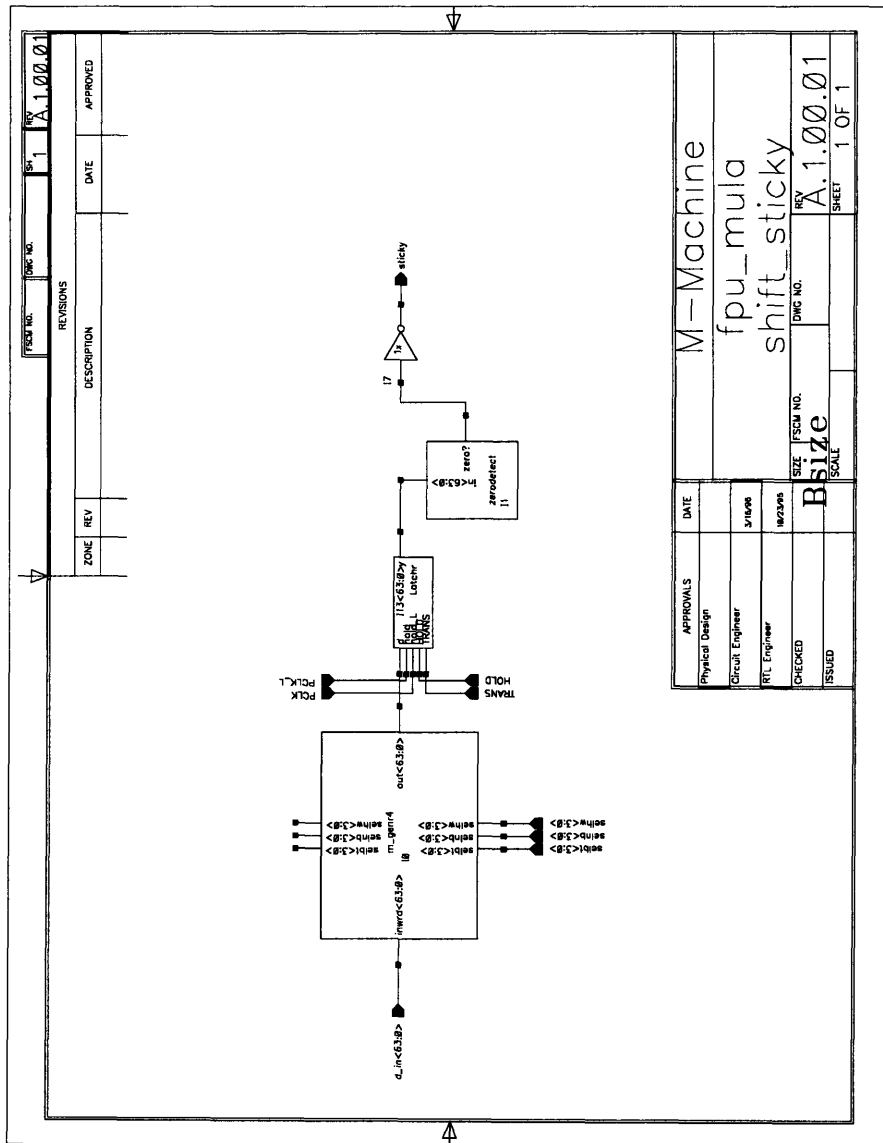
D.4 Stage 3



D.5 Stage 4

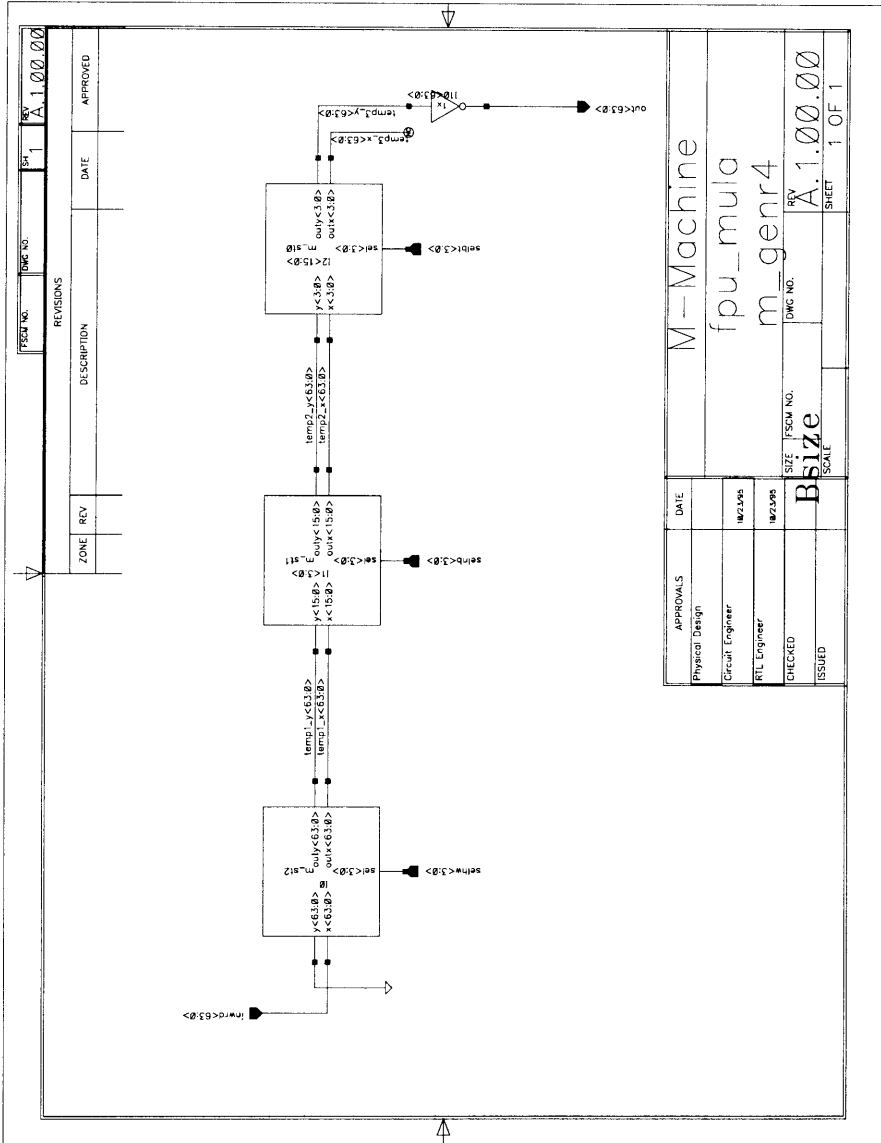


D.5.1 Shift Sticky (Mask Generator and subcells)



APPROVALS	DATE
Physical Design	
Circuit Engineer	3/16/96
RTL Engineer	10/23/95
CHECKED	
ISSUED	

M-Machine	
fpu_muia	
shift_sticky	
SIZE	13<63:0>
SCALE	17
REV	A.1.00.01
DWG NO.	
SHEET	1 OF 1



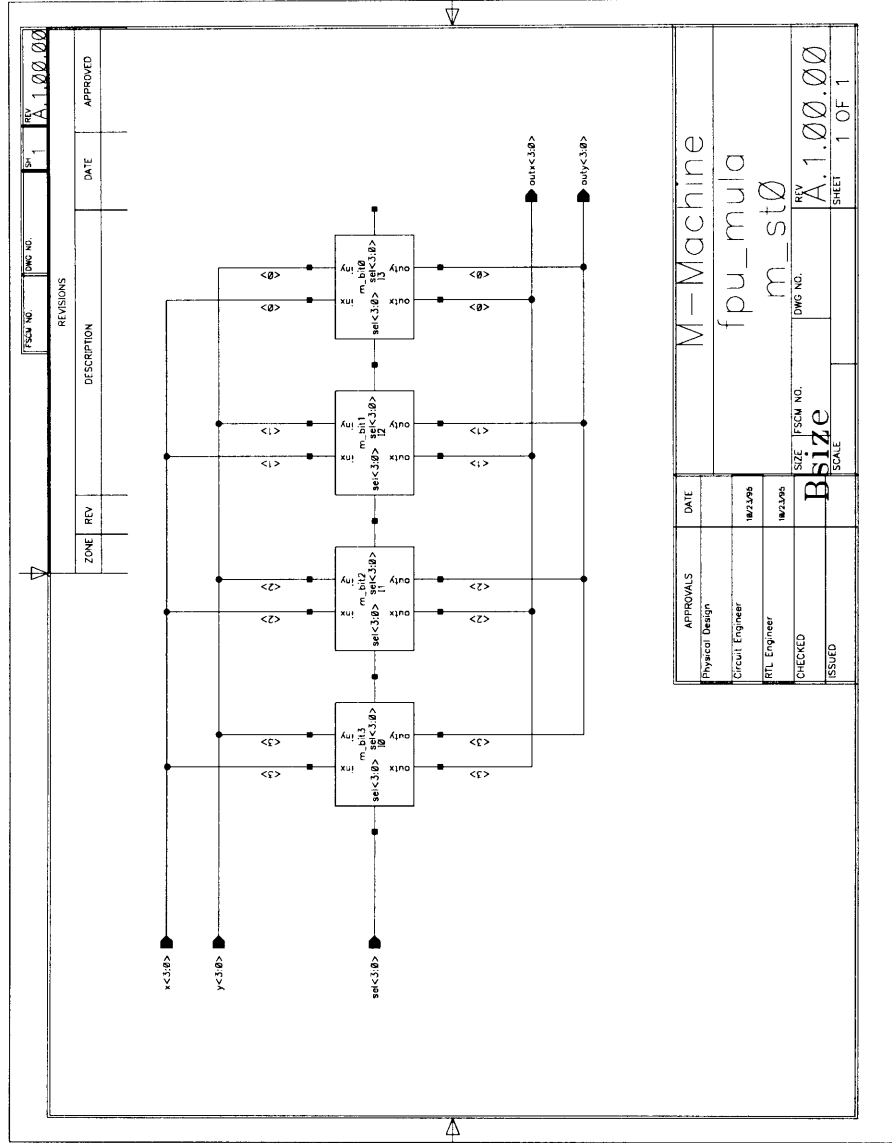
PSUM NO. BMC NO. 1 A.1.00.00

REVISIONS			DATE	APPROVED
ZONE	REV	DESCRIPTION		

APPROVALS		DATE
Physical Design		
Circuit Engineer	10/2/98	
RTL Engineer	10/2/98	
CHECKED		
ISSUED		

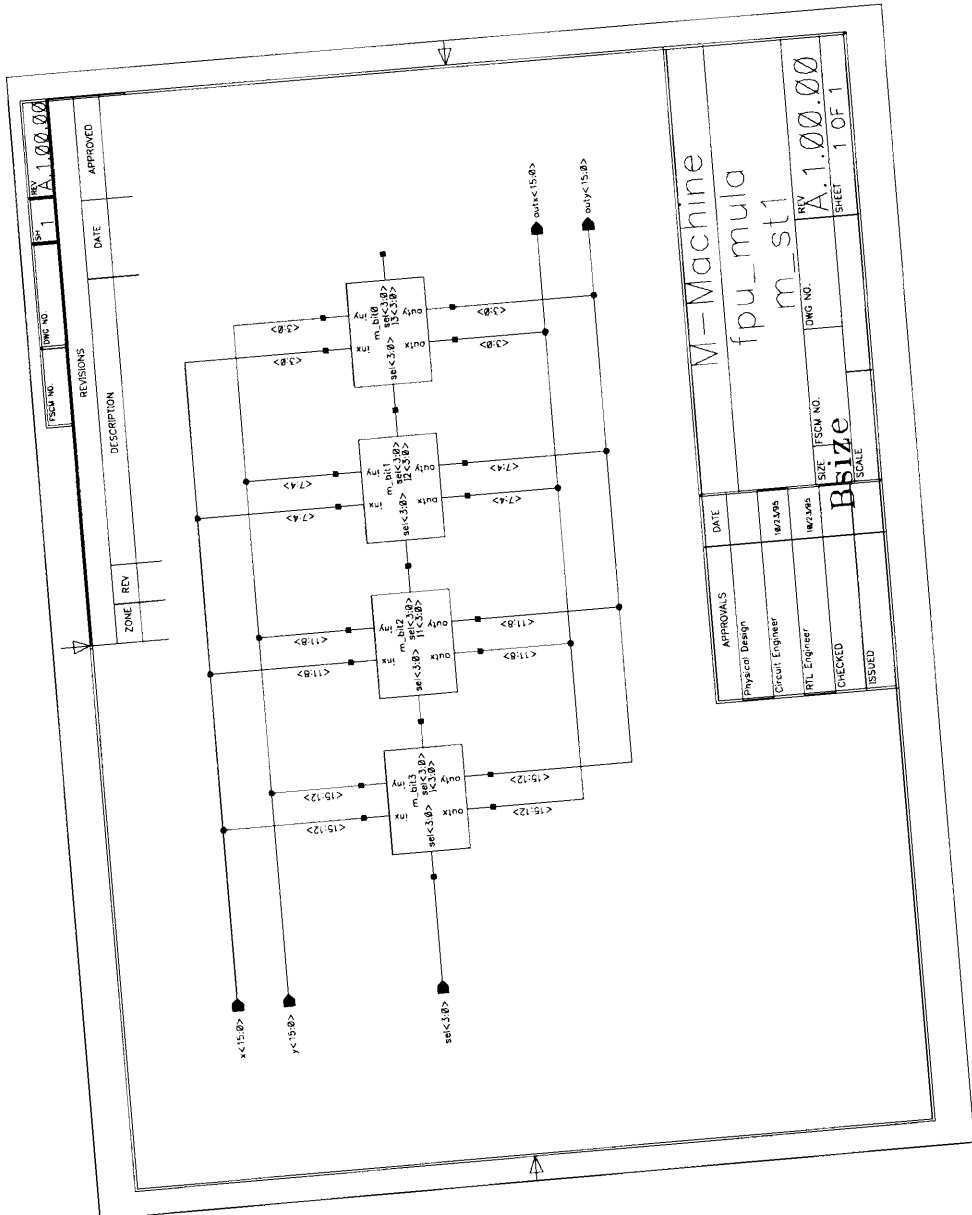
M-Machine
fpu_mula
m_genr4

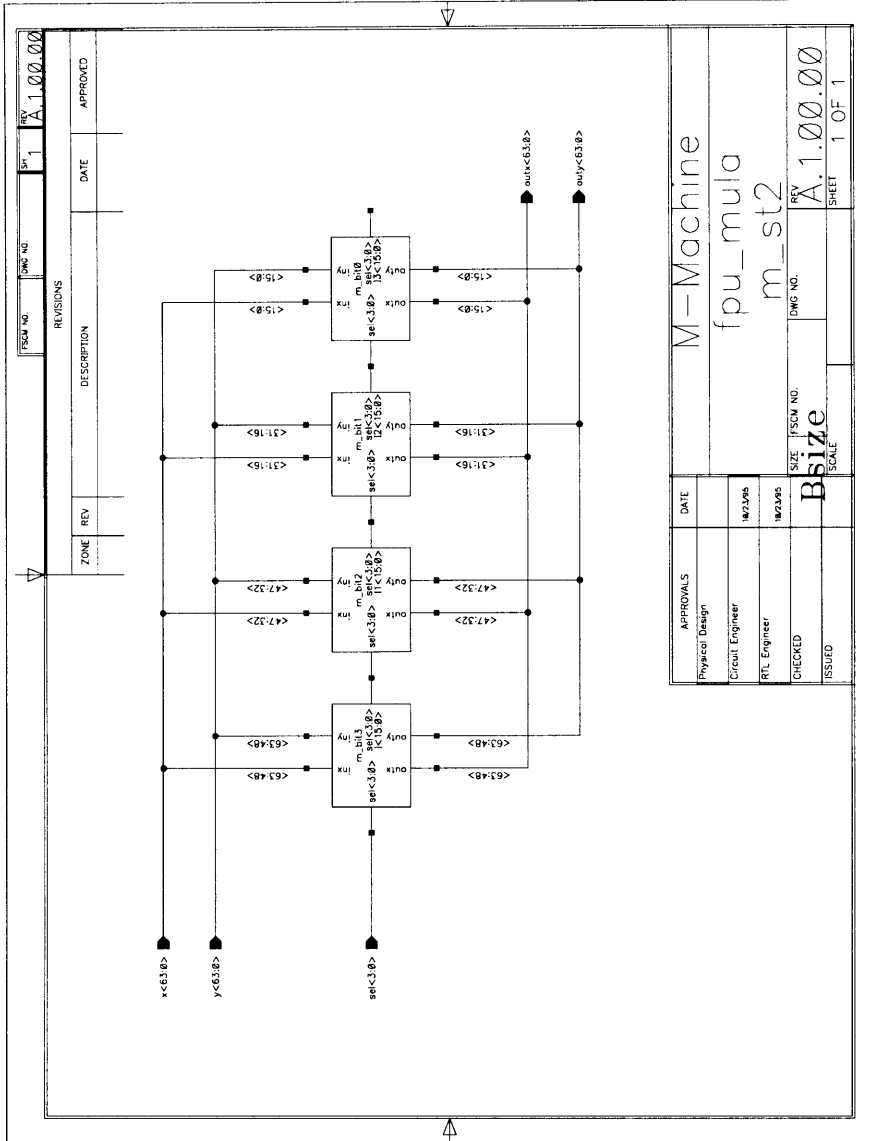
SIZE PSUM NO. BMC NO. REV A.1.00.00
SCALE SHEET 1 OF 1

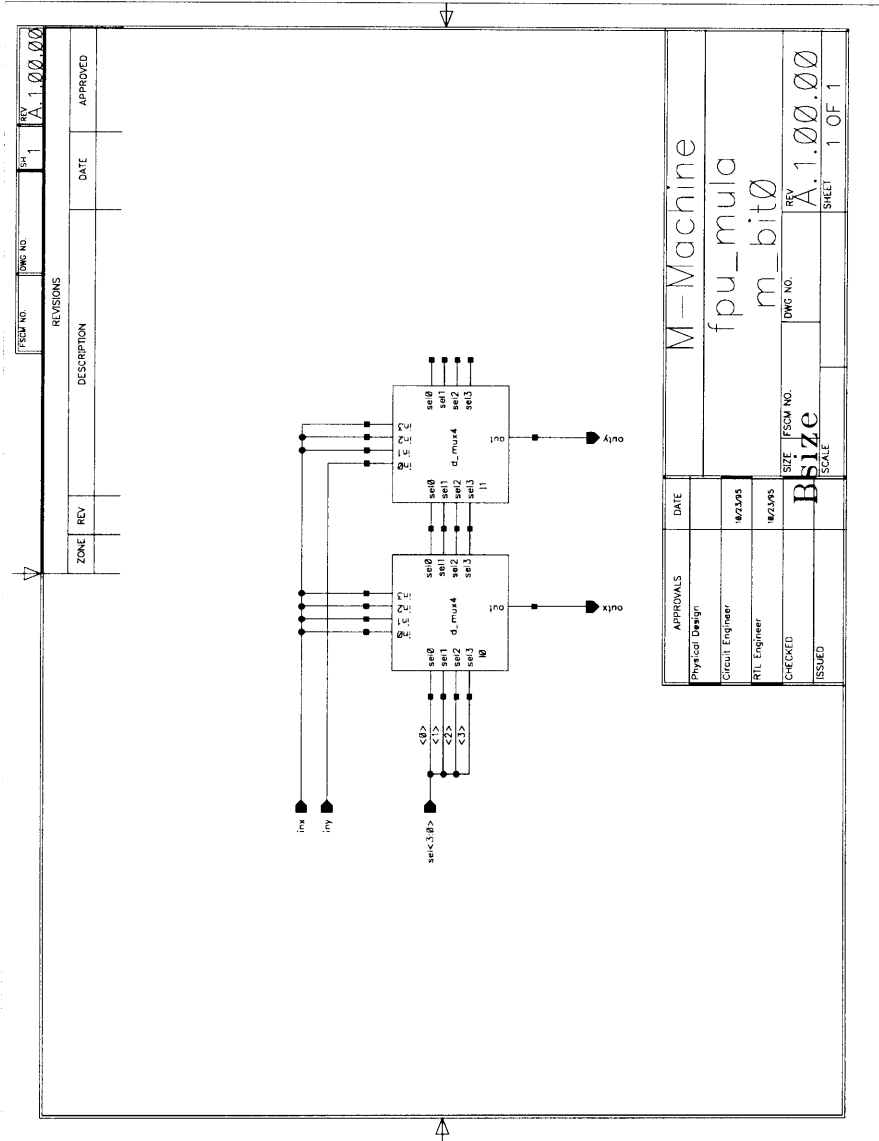


APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/90
RTL Engineer	10/23/90
CHECKED	
ISSUED	

M-Machine	
fpu_mula	
m_st0	
SIZE	DWG NO.
Bsize	A.1.00.00
SCALE	SHEET 1 OF 1





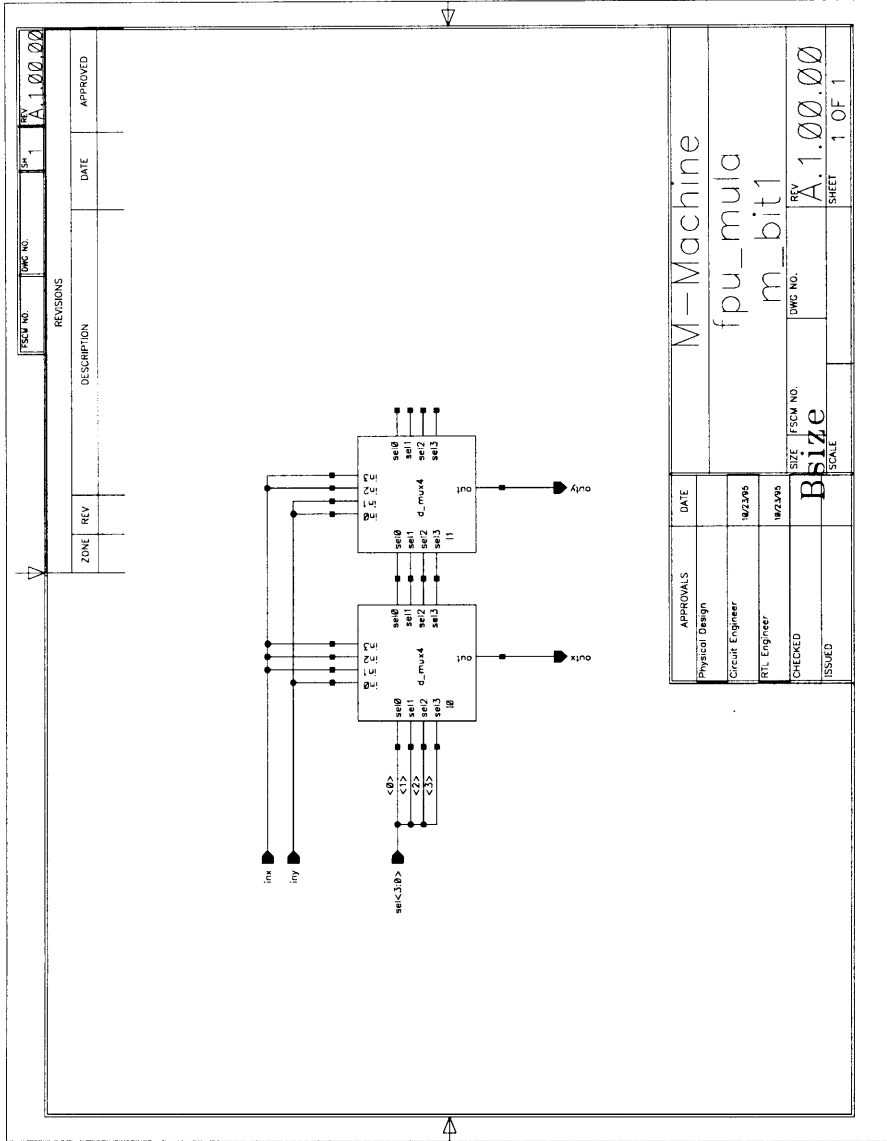


FSSM NO.	DWG NO.	REV.	DATE	APPROVED
		1		

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
Physical Design	
Circuit Engineer	14/2/95
RTL Engineer	14/2/95
CHECKED	
ISSUED	

M-Machine	
fpu_mula	
m_bit0	
SIZE	FSSM NO.
DWG NO.	REV.
A.1.00.00	A.1.00.00
SCALE	SHEET
	1 OF 1

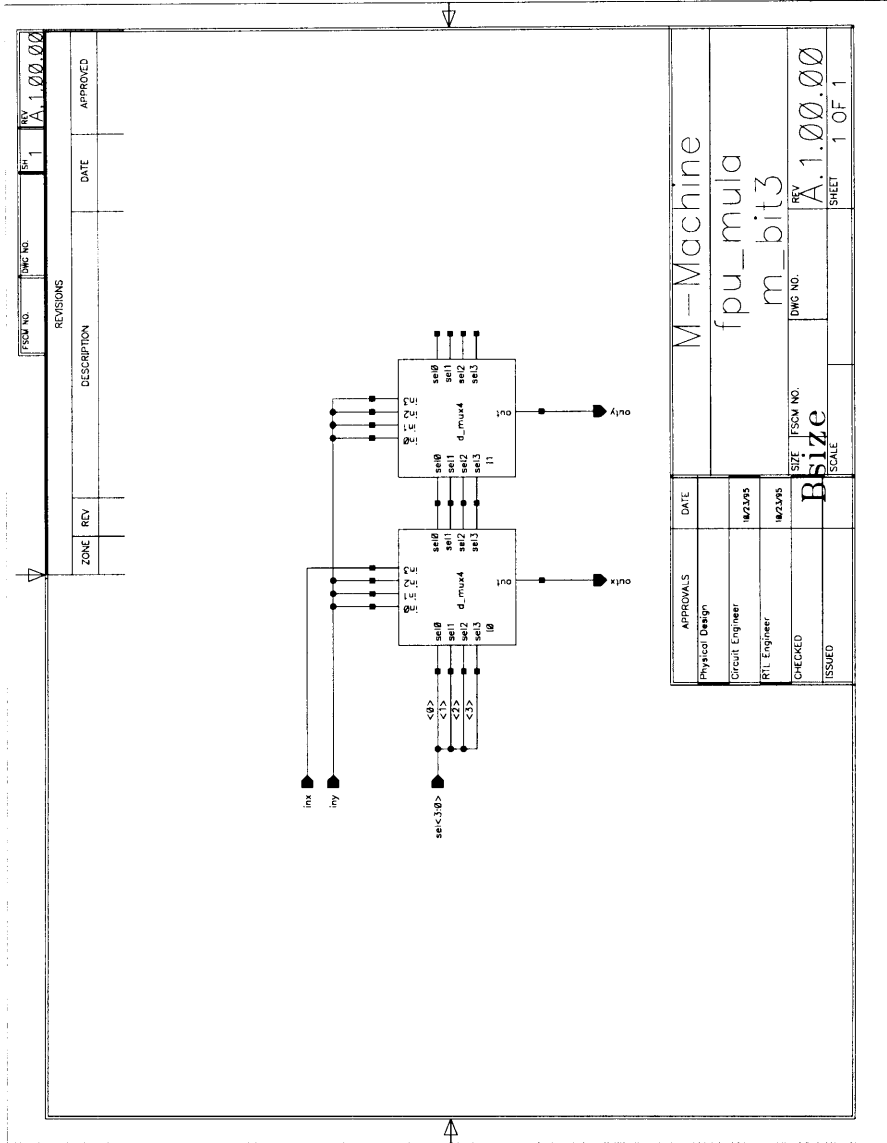


FSCM NO.	DWG NO.	REV	DATE	APPROVED
		A.1.00.00		

REVISIONS		DATE	APPROVED
ZONE	REV		

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/95
RTL Engineer	10/23/95
CHECKED	
ISSUED	

M-Machine	
fpu_mula	
m_bit1	
SIZE	FSCM NO.
Bsize	
DWG NO.	REV
A.1.00.00	A.1.00.00
SHEET	1 OF 1



ZONE	REV	DESCRIPTION	DATE	APPROVED

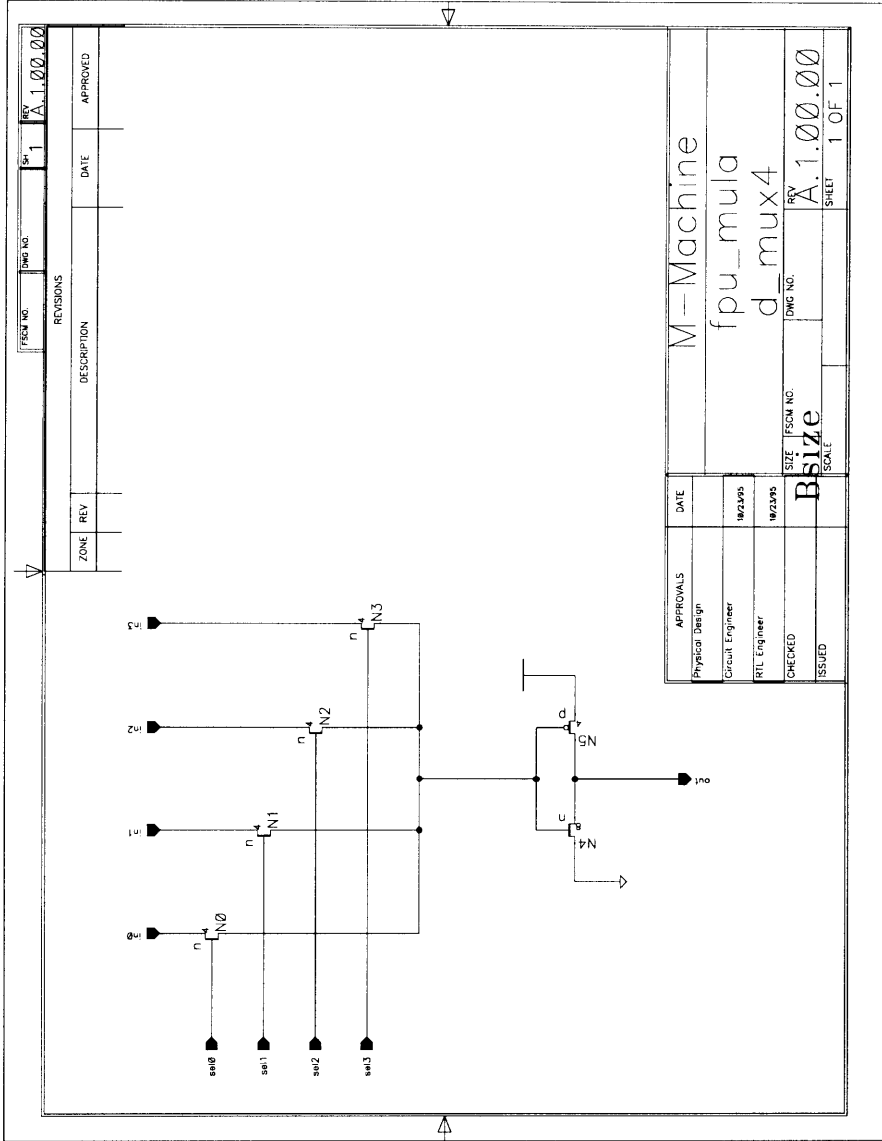
FSOM NO.	DWG NO.	REV
		1

FSOM NO.	DWG NO.	REV
		A.1.00.00

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/95
RTL Engineer	10/23/95
CHECKED	
ISSUED	

SIZE	FSOM NO.	DWG NO.	REV
Bsize			A.1.00.00
SCALE			SHEET 1 OF 1

M--Machine
 fpu_mula
 m_bit3

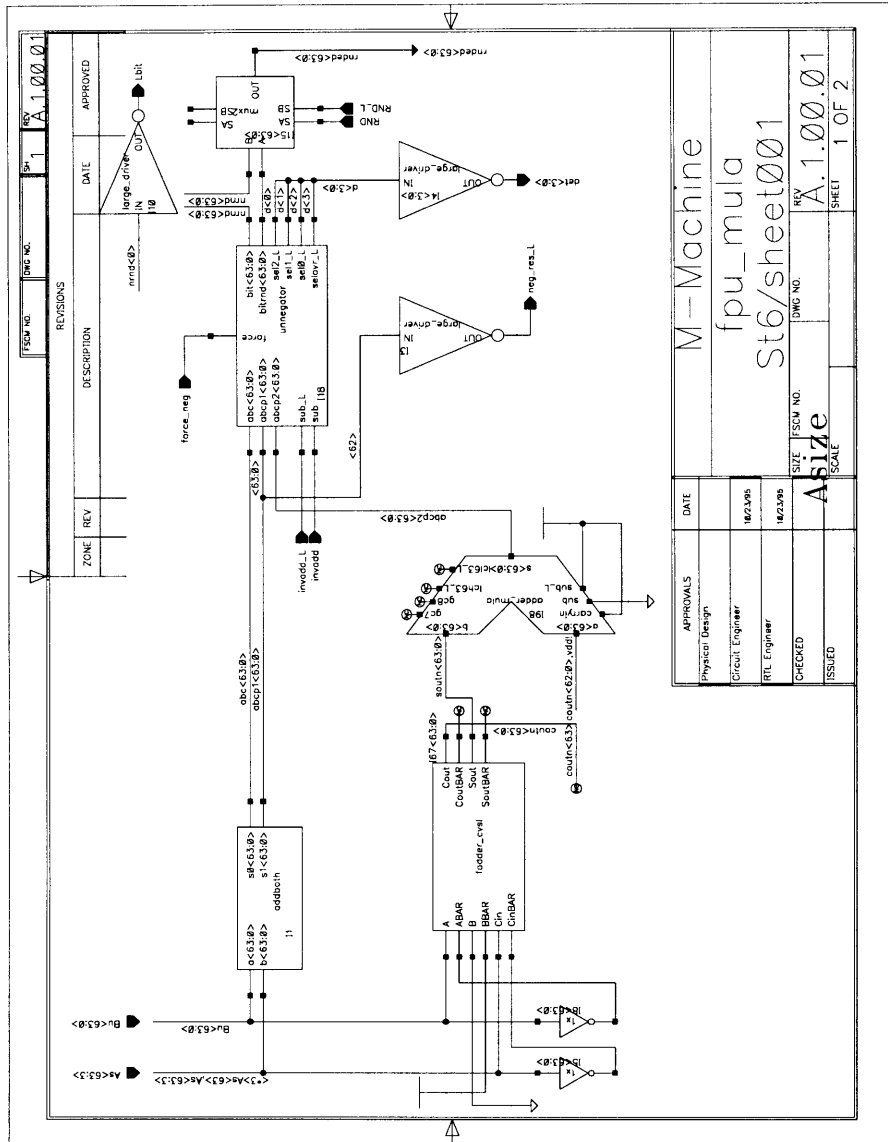


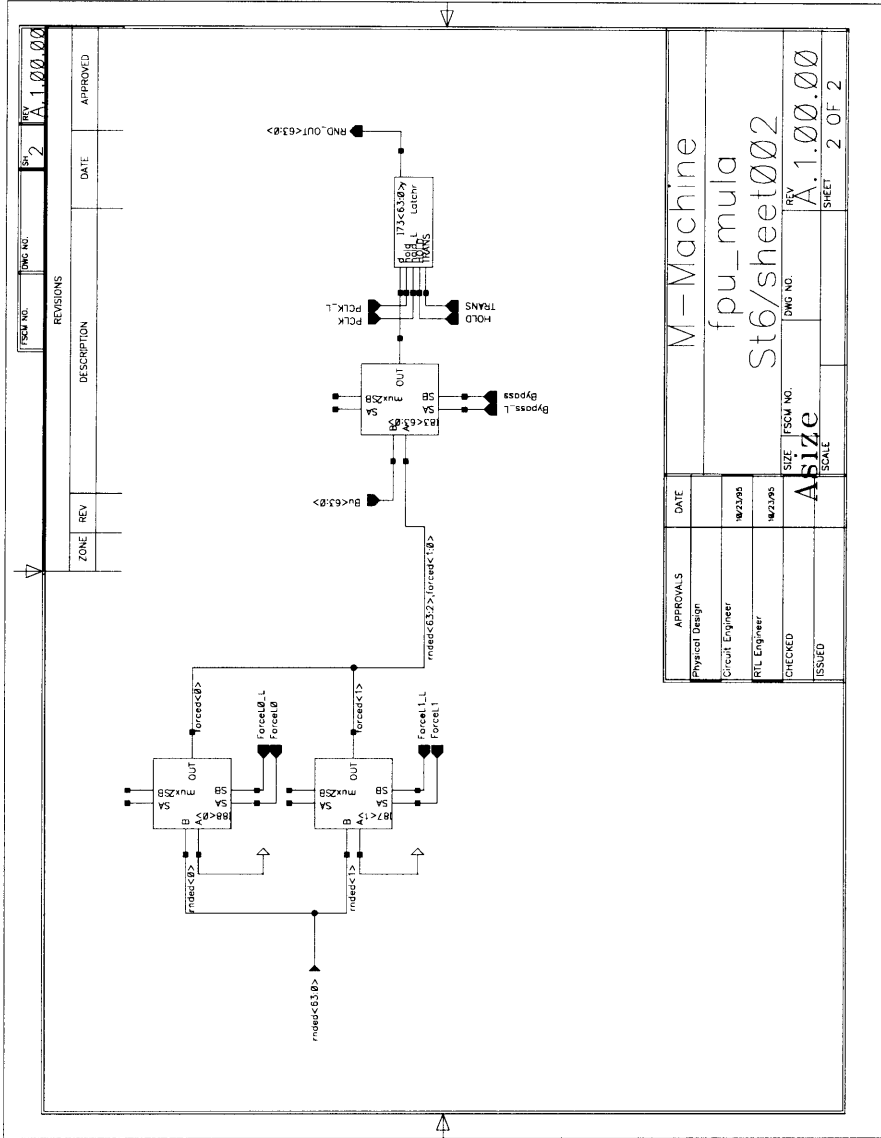
REVISIONS		FORM NO.	DWG NO.	REV.	DATE	APPROVED
ZONE	REV					

APPROVALS		DATE
Physical Design		
Circuit Engineer		18/03/95
RTL Engineer		18/03/95
CHECKED		
ISSUED		

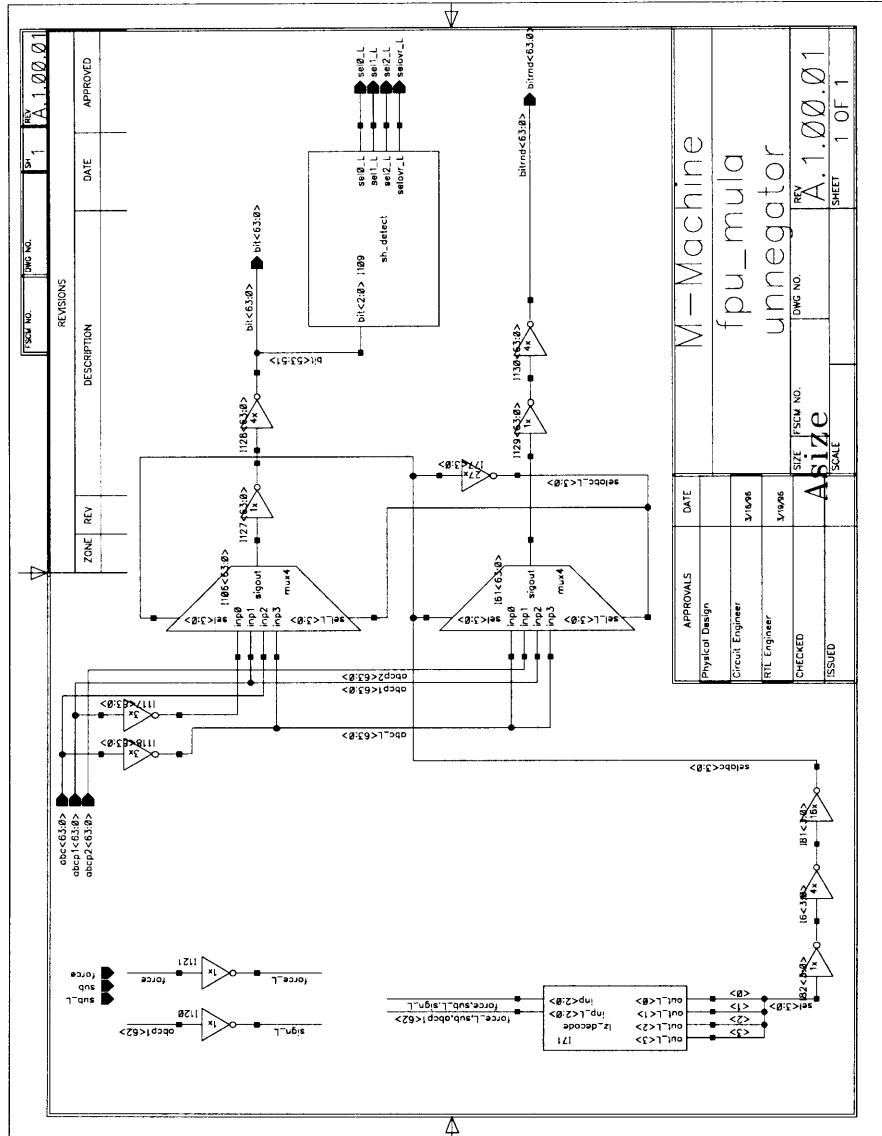
M--Machine	
fpu_mula	
d_mux4	
SIZE	FSCM NO.
Bsize	
SCALE	
DWG NO.	REV.
	A.1.00.00
SHEET	1 OF 1

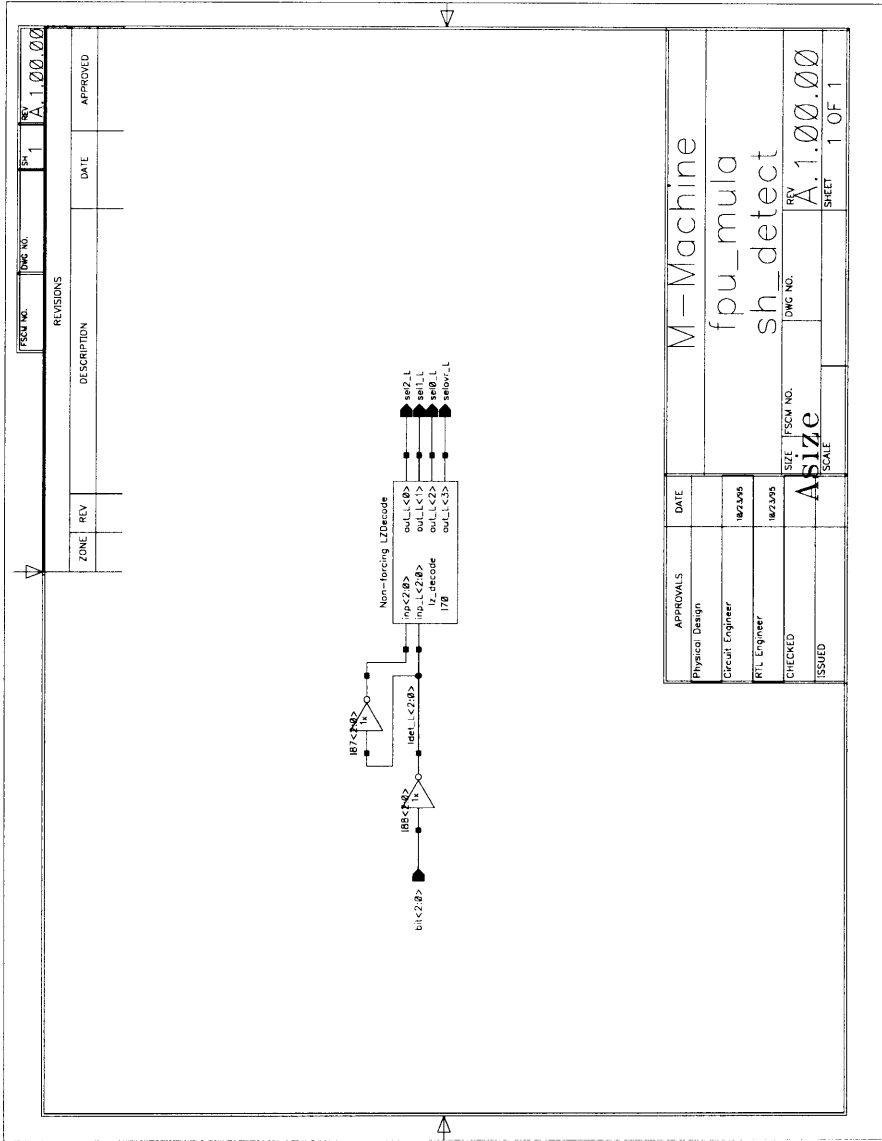
D.6 Stage 6





D.6.1 Stage 6 Subcells



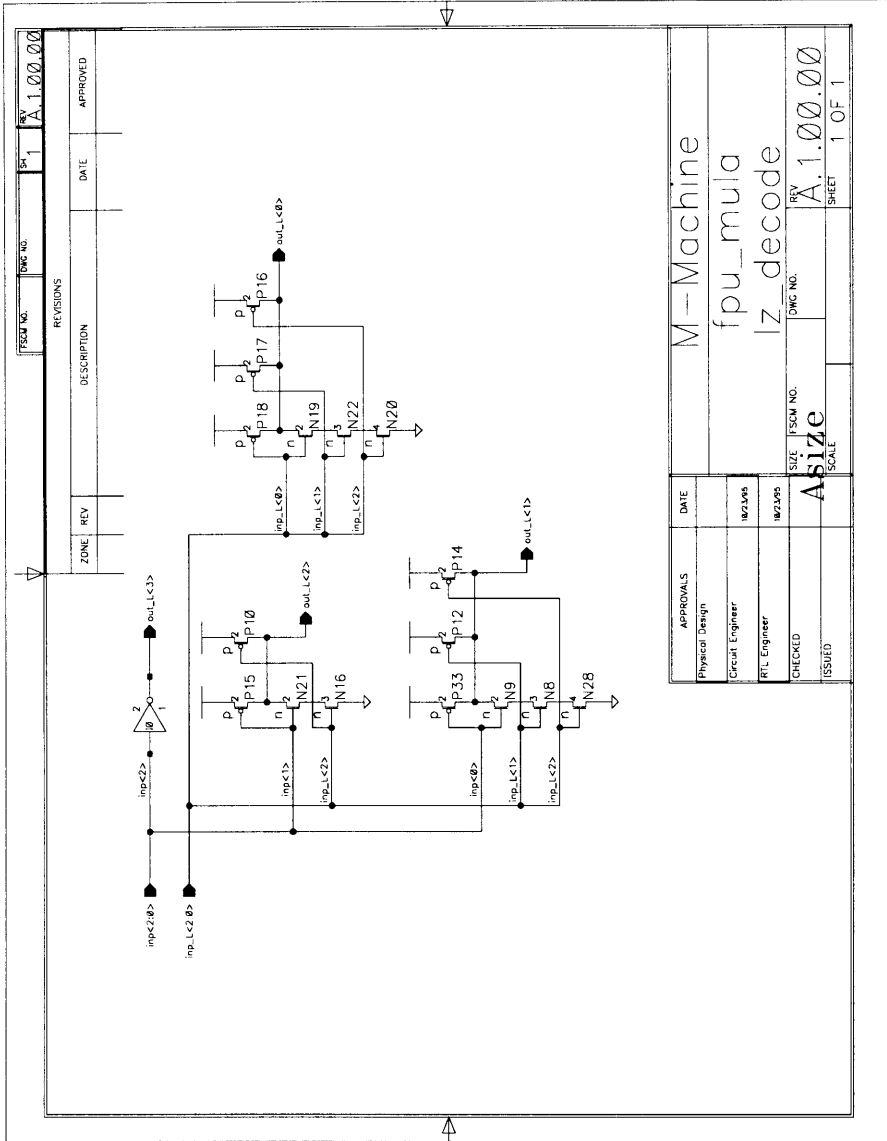


FSCM NO. 1 A.1.00.00
 REV. NO. 1 A.1.00.00

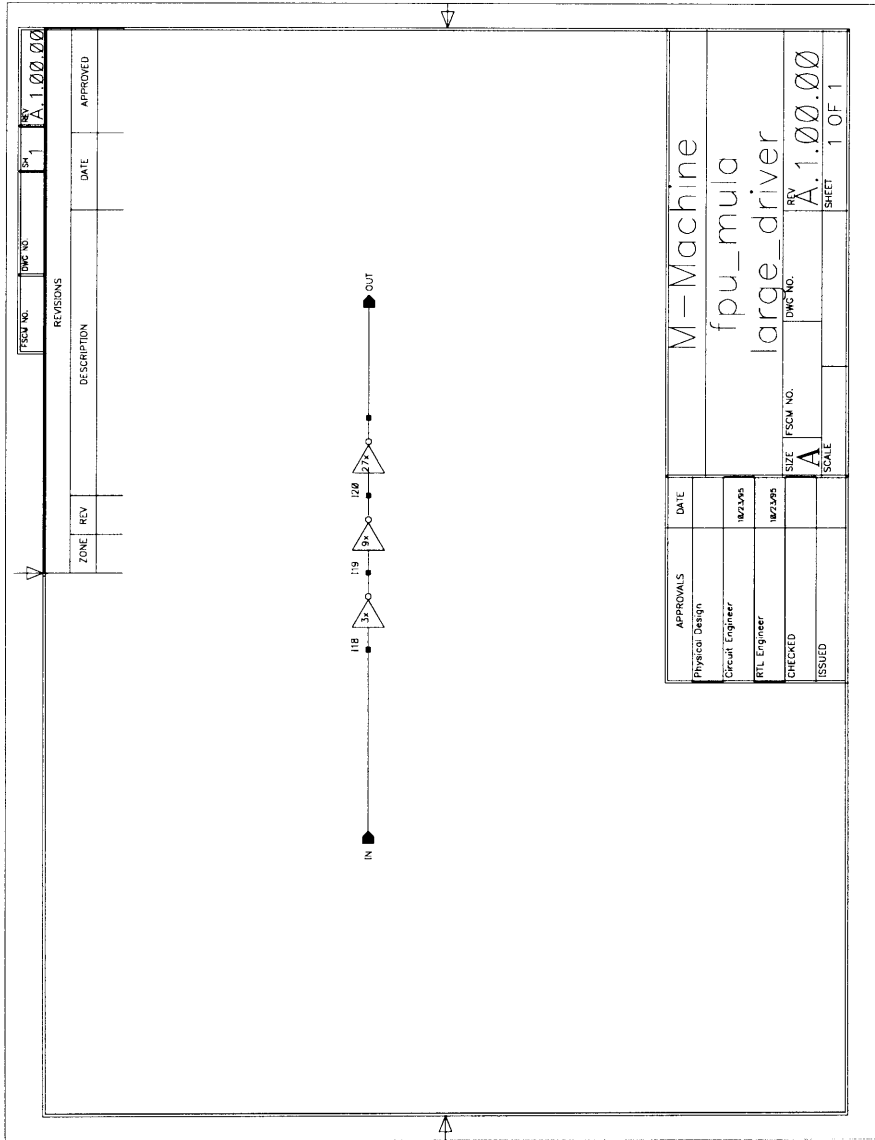
REVISIONS			DATE	APPROVED
ZONE	REV	DESCRIPTION		

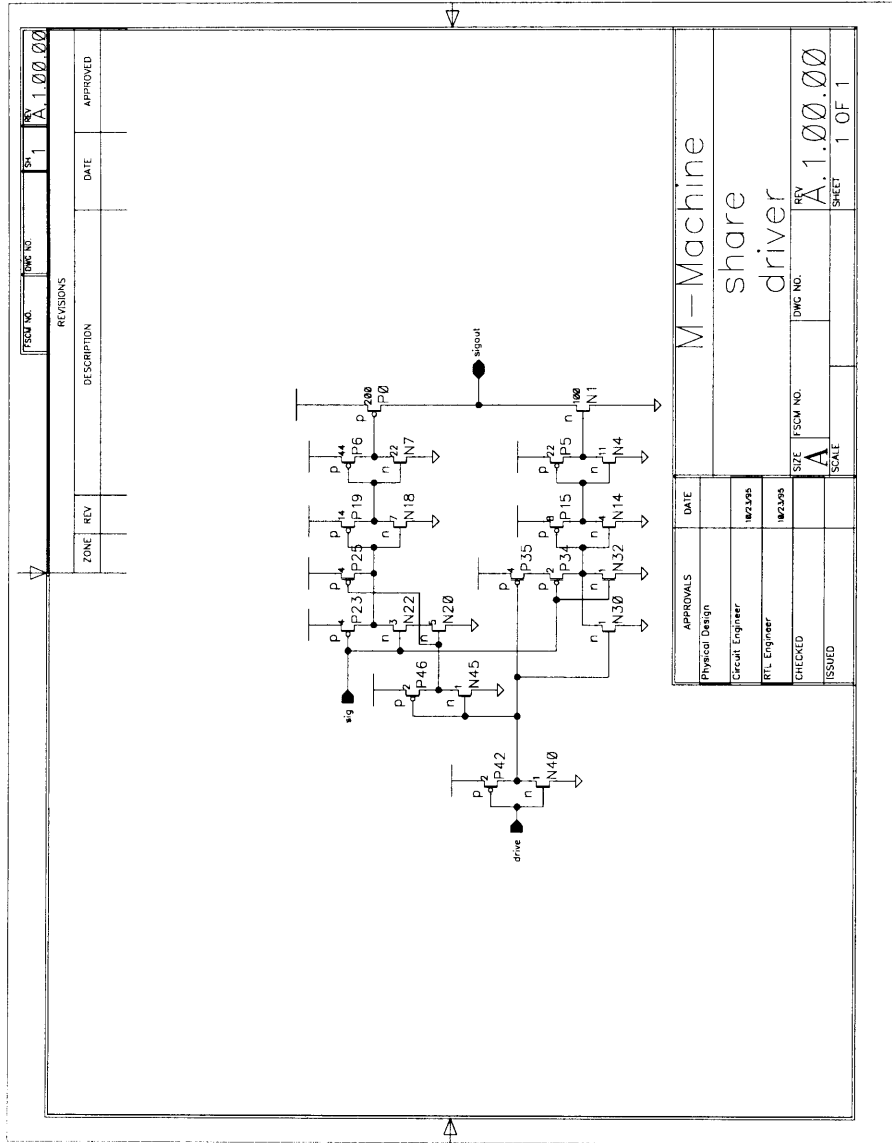
APPROVALS	DATE
Physical Design	
Circuit Engineer	18/2/95
RTL Engineer	18/2/95
CHECKED	
ISSUED	

M-Machine	
fpu_mula	
sh_detect	
SIZE	FSCM NO.
A size	A.1.00.00
SCALE	SHEET
	1 OF 1



D.9 Drivers and Misc Cells



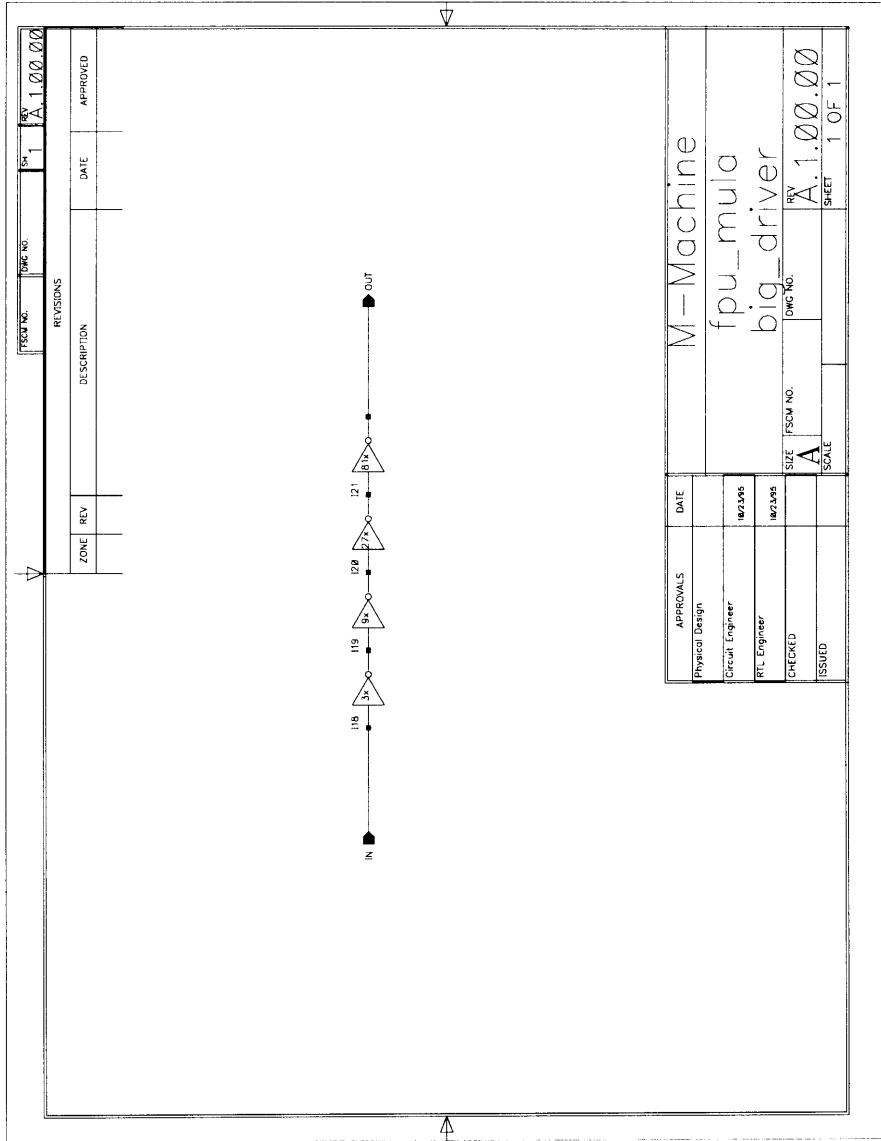


ZONE	REV	DESCRIPTION	DATE	APPROVED

REVISIONS	
REV	DATE
1	A.1.00.00

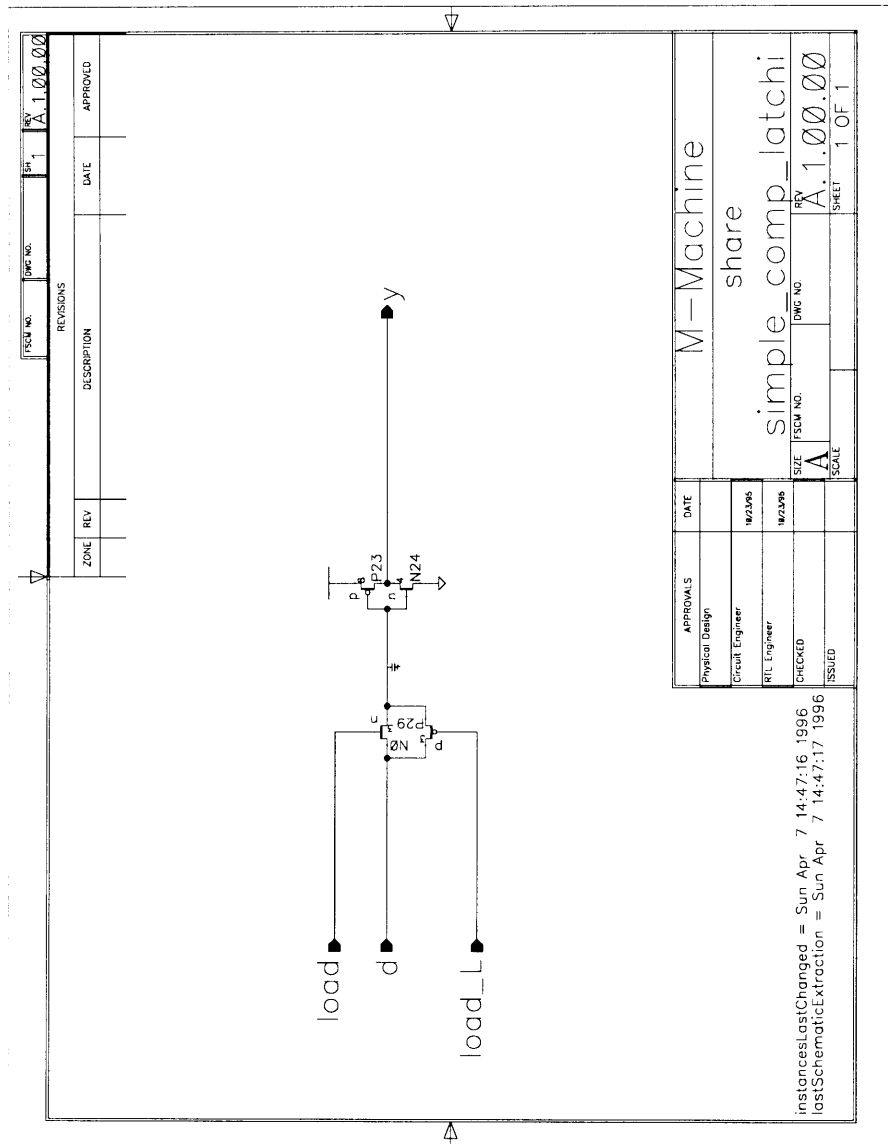
APPROVALS		DATE
Physical Design		
Circuit Engineer	IM2399	
RTL Engineer	IM2399	
CHECKED		
ISSUED		

M-Machine	
share driver	
SIZE	FSCM NO.
A	
SCALE	
REV	DWG NO.
A.1.00.00	
SHEET	1 OF 1



D.10 Shared Library Cells Used

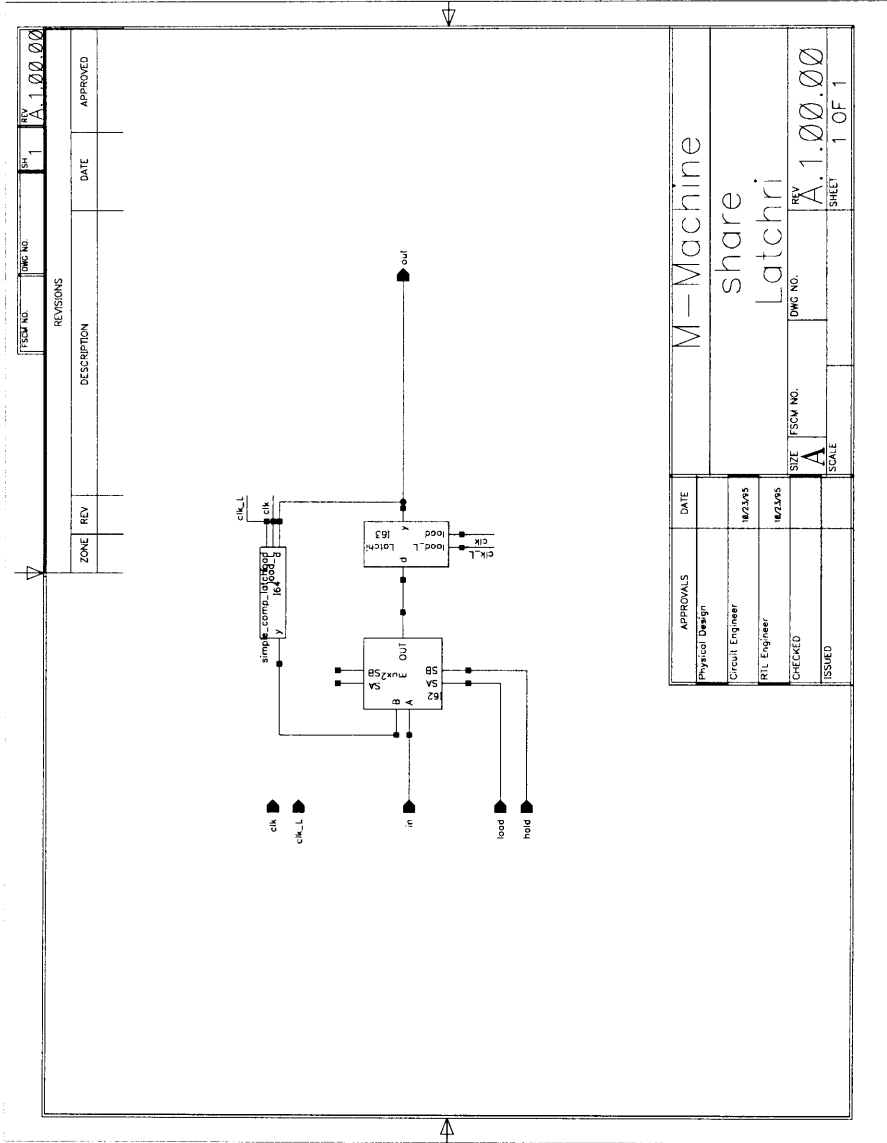
D.10.1 Latches



FSCM NO.	REV	DATE	APPROVED
	1	A.1.00.00	
REVISIONS			
ZONE	REV	DATE	APPROVED

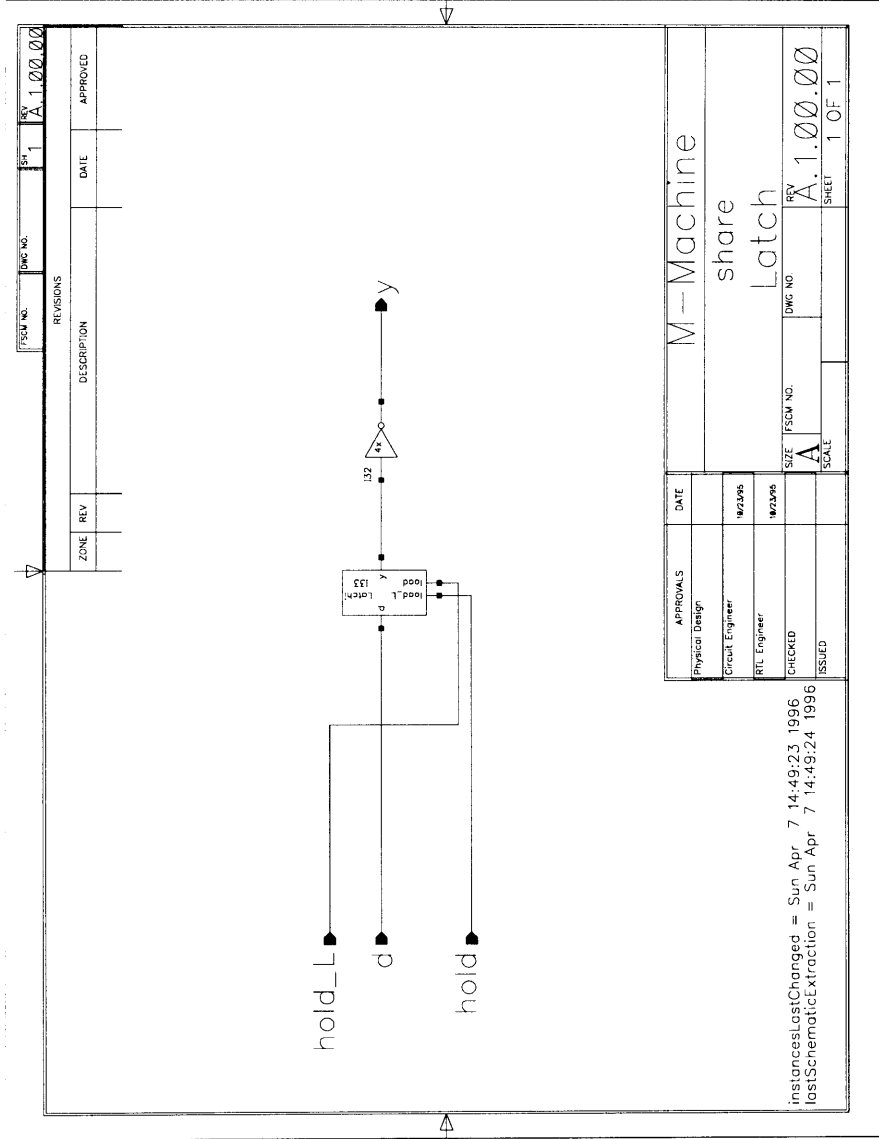
APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/95
RTL Engineer	10/23/95
CHECKED	
ISSUED	

M-Machine
 share
 simple_comp_latchi
 SIZE A FSCM NO. DWG NO. REV A.1.00.00
 SCALE SHEET 1 OF 1



APPROVALS	DATE
Physical Design	
Circuit Engineer	14/2/95
RTL Engineer	14/2/95
CHECKED	
ISSUED	

M-Machine					
share					
Latchri					
SIZE	FSCM NO.	DWG NO.	REV.	SCALE	SHEET
A	A	A.1.00.00	A.1.00.00	1 OF 1	1 OF 1

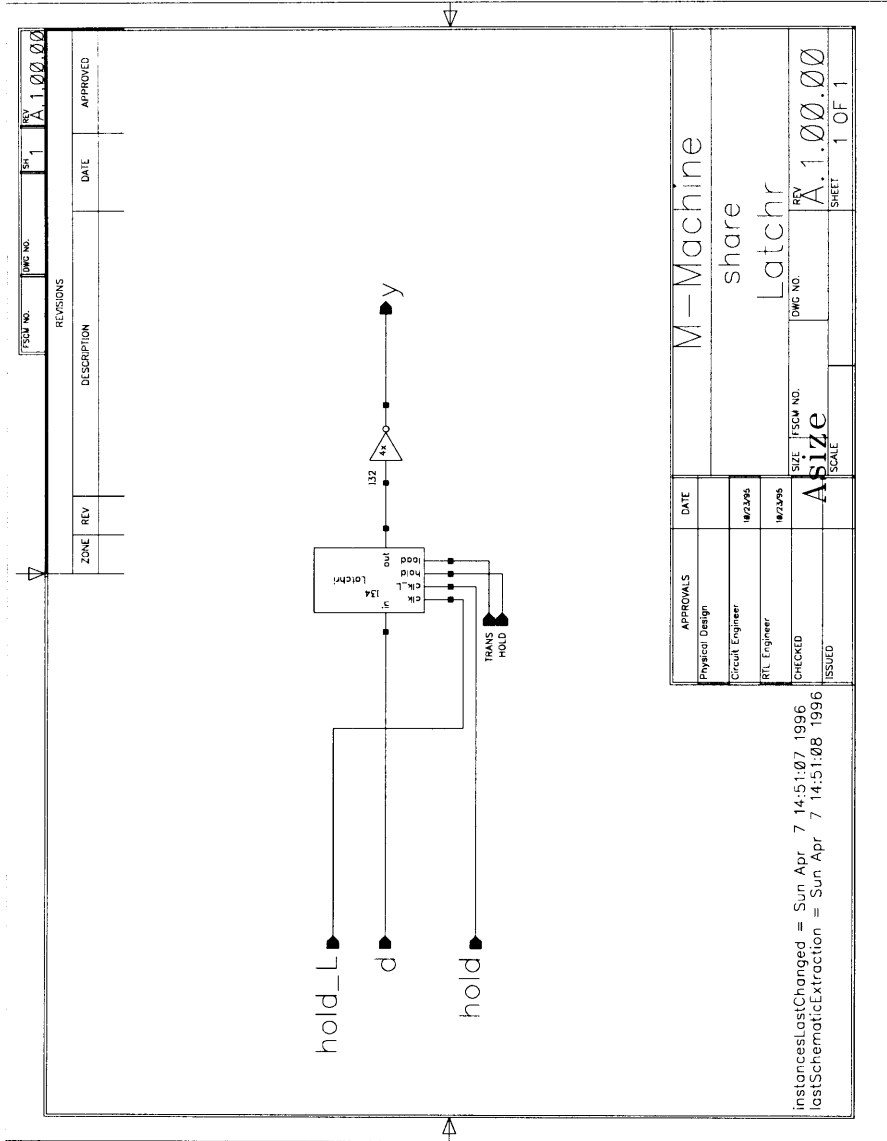


FSCM NO.	1	REV	A.1.00.00
REVISIONS			
ZONE	REV	DATE	APPROVED

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/96
RTL Engineer	10/23/96
CHECKED	
ISSUED	

instancesLostChanged = Sun Apr 7 14:49:23 1996
lostSchematicExtraction = Sun Apr 7 14:49:24 1996

SIZE	FSCM NO.	REV
A		A.1.00.00
SCALE		SHEET 1 OF 1



FSQM No. 1
DWG No. A.1.00.00

REVISIONS		DATE	APPROVED
ZONE	REV		

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/96
RTL Engineer	10/23/96
CHECKED	SIZE FSQM NO.
ISSUED	SCALE

M-Machine
share
Latch

instancesLastChanged = Sun Apr 7 14:51:07 1996
lastSchematicExtraction = Sun Apr 7 14:51:08 1996

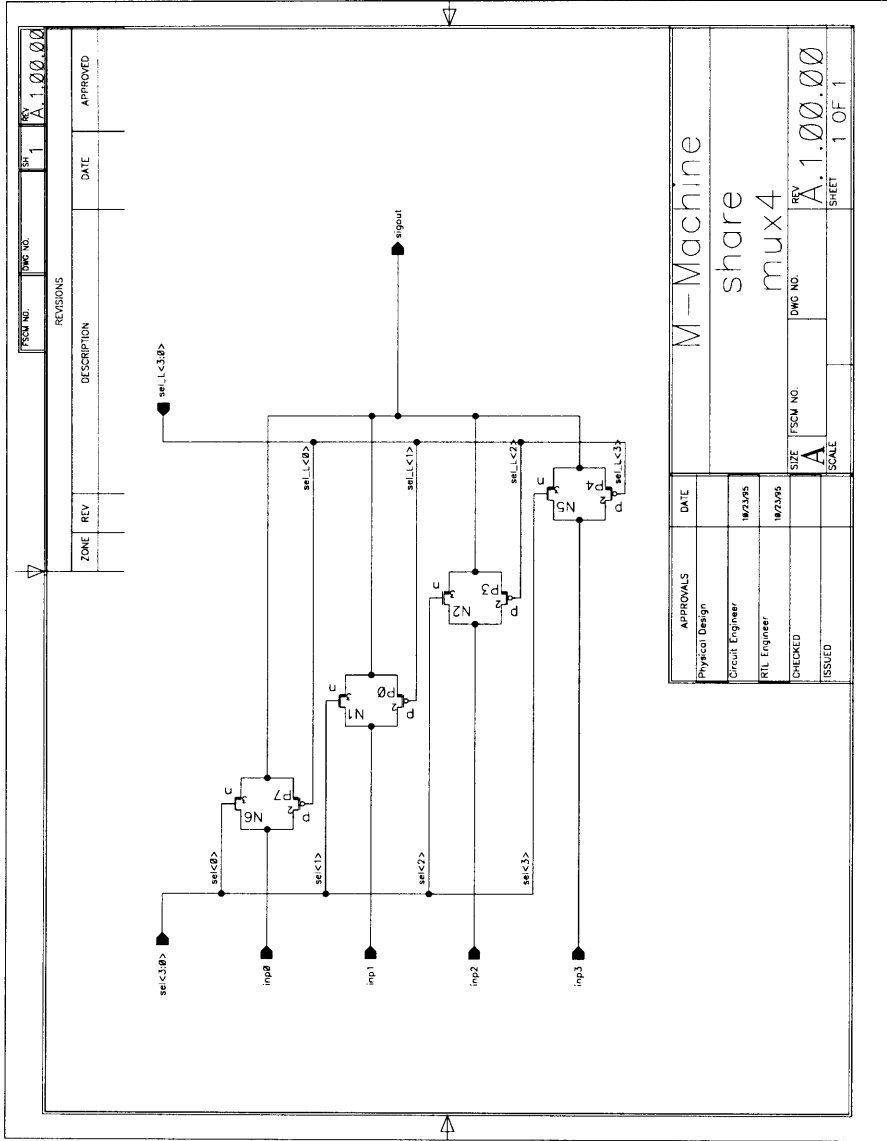
DWG No. A.1.00.00
SHEET 1 OF 1

D.10.2 Muxes

	FSCM NO. 1	REV. A.1.00.00			
REVISIONS		DATE		APPROVED	
ZONE	REV	DESCRIPTION	DATE		

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/2/95
REL Engineer	10/2/95
CHECKED	
ISSUED	

M-Machine	
share	
mux2	
SIZE A	FSCM NO.
SCALE	REV. A.1.00.00
	SHEET 1 OF 1



REVISIONS		DATE	APPROVED
ZONE	REV		

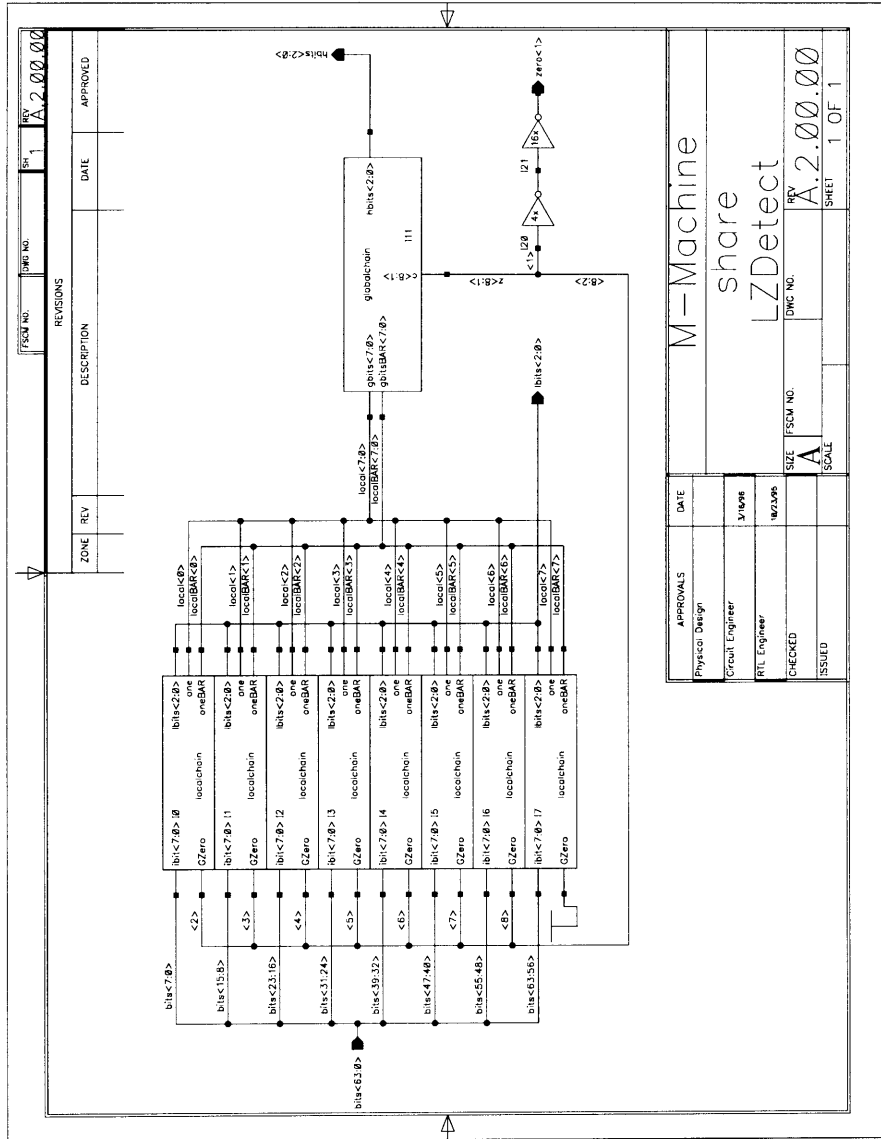
FORM NO.	DWG NO.	REV	REV
		1	A.1.00.00

APPROVALS	DATE
Physical Design	
Circuit Engineer	11/23/95
RTL Engineer	11/23/95
CHECKED	
ISSUED	

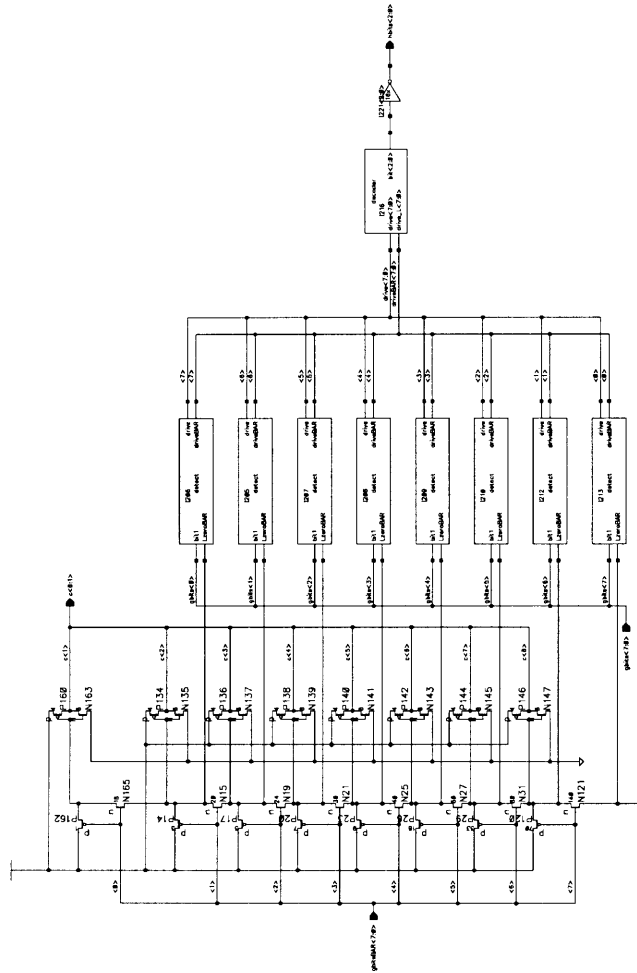
M-Machine
share
mux4

FORM NO.	DWG NO.	REV	REV
		A	A.1.00.00
SIZE	SCALE	SHEET 1 OF 1	

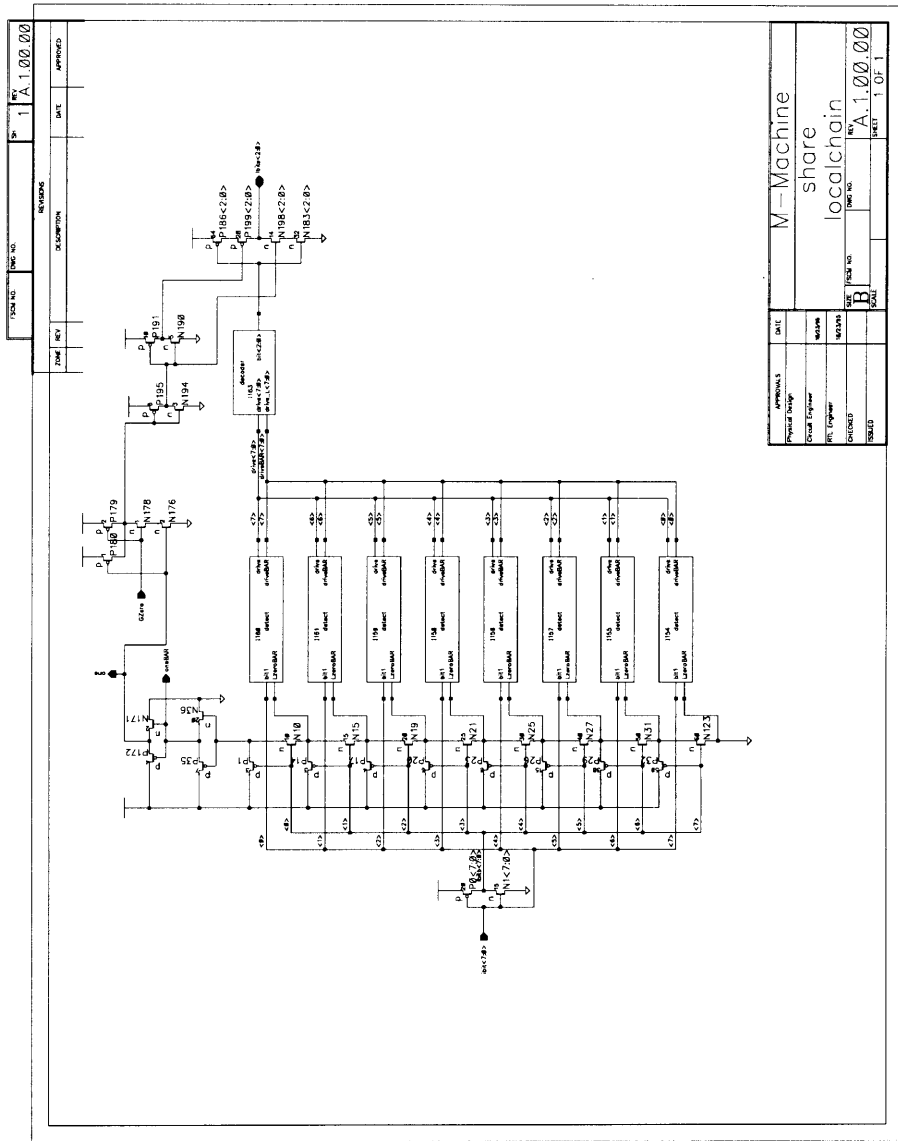
D.10.3 Leading Zero Detector



FORM NO.	REV.	DATE	APPROVED
1	1	A.1.01.00	
DESCRIPTION		REVISIONS	
ZONE	REV.	DATE	APPROVED



APPROVALS	DATE
Project Manager	
Elect. Engineer	
Mech. Engineer	
Checked	
Tested	
M-Machine	
share	
globalchain	
FORM NO.	REV.
1	A.1.01.00
SHEET	1 OF 1

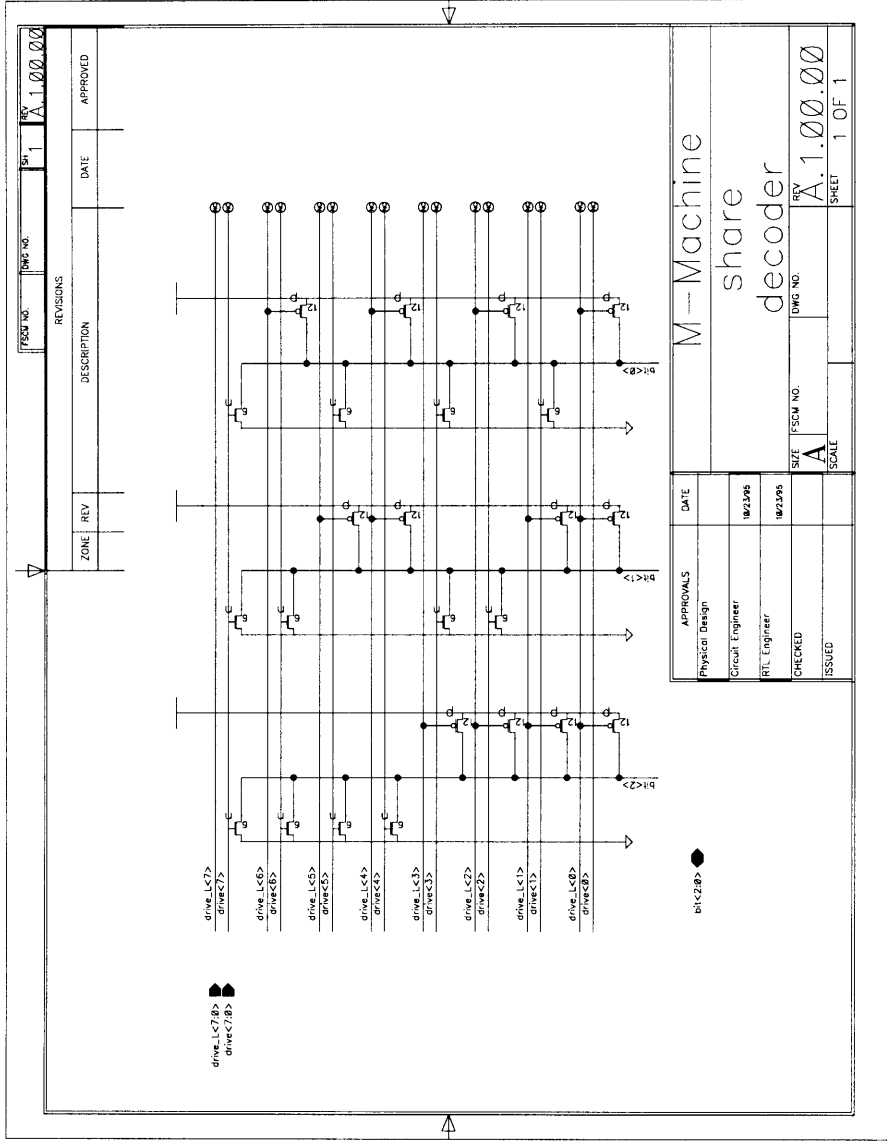


ITEM NO.	REV.	DESCRIPTION	DATE	APPROVED
1	A.1.00.00			

APPROVALS	DATE
Physical Design	
Circuit Engineer	
IC Engineer	
Checked	
Drawn	

PROJECT NO.	REV.
B	A.1.00.00

SHEET	OF
1	1



APPROVALS		DATE
Physical Design		
Circuit Engineer		10/23/06
RTL Engineer		10/23/06
CHECKED		
ISSUED		

M-Machine
share
decoder

DWG NO.

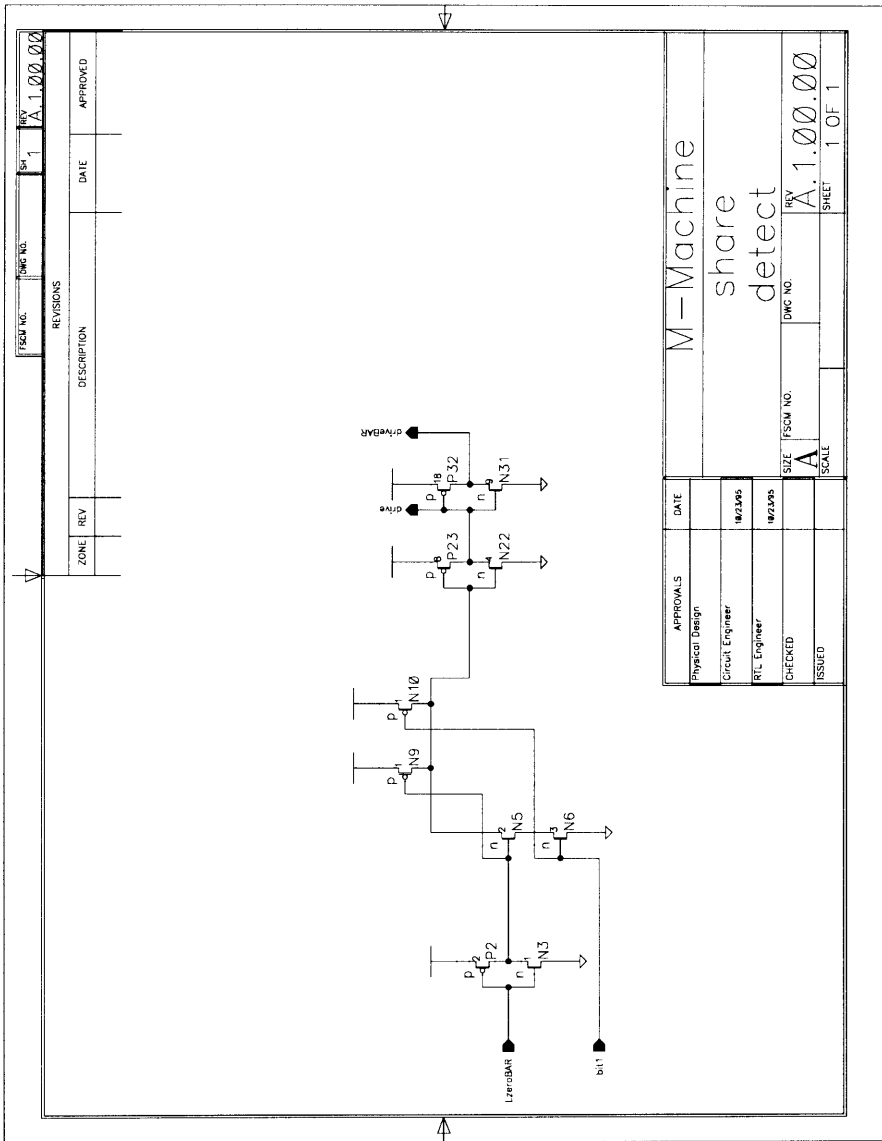
REV A.1.00.00

SIZE A SCALE

SHEET 1 OF 1

REVISIONS		DATE	APPROVED
ZONE	REV		

FIGURE NO.	DWG NO.	REV
		<u>A.1.00.00</u>



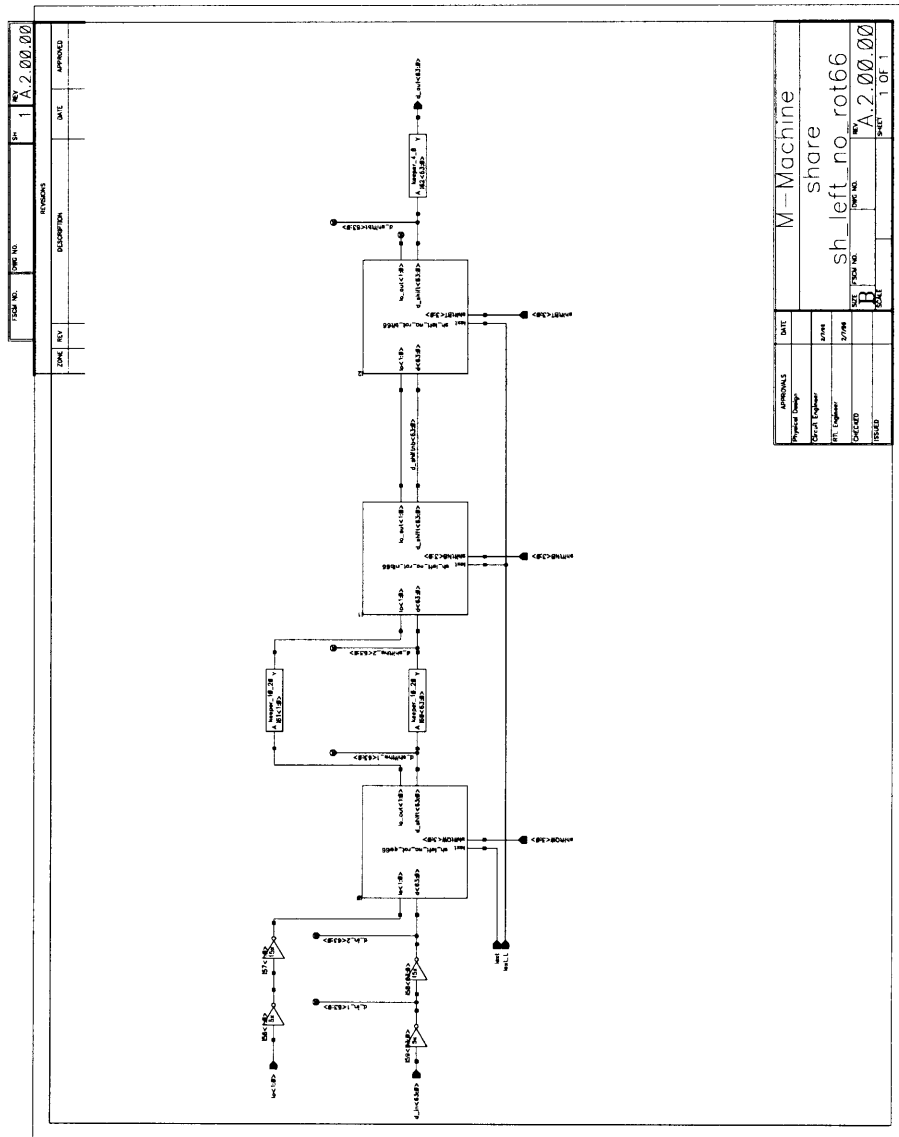
ESD No.	DMC No.	REV	DATE	APPROVED
		1		A.1.00.00

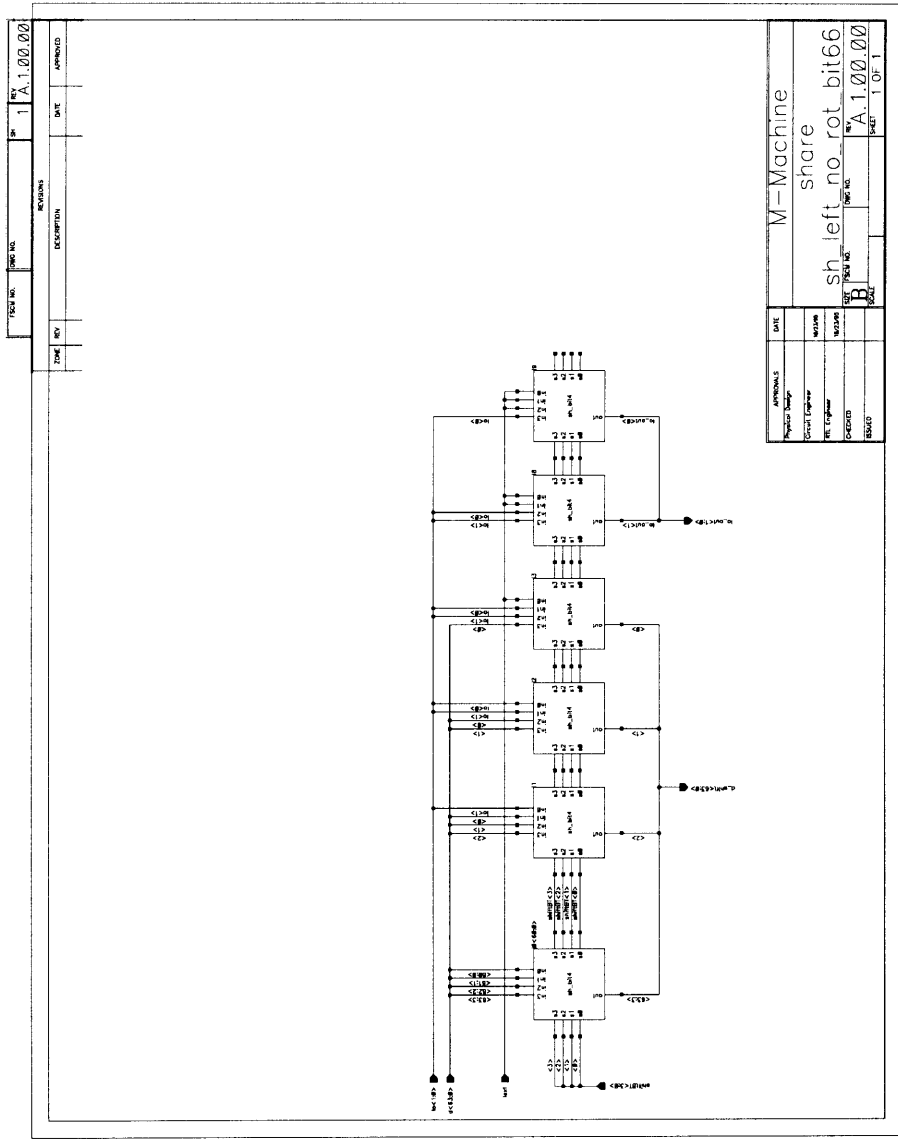
REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

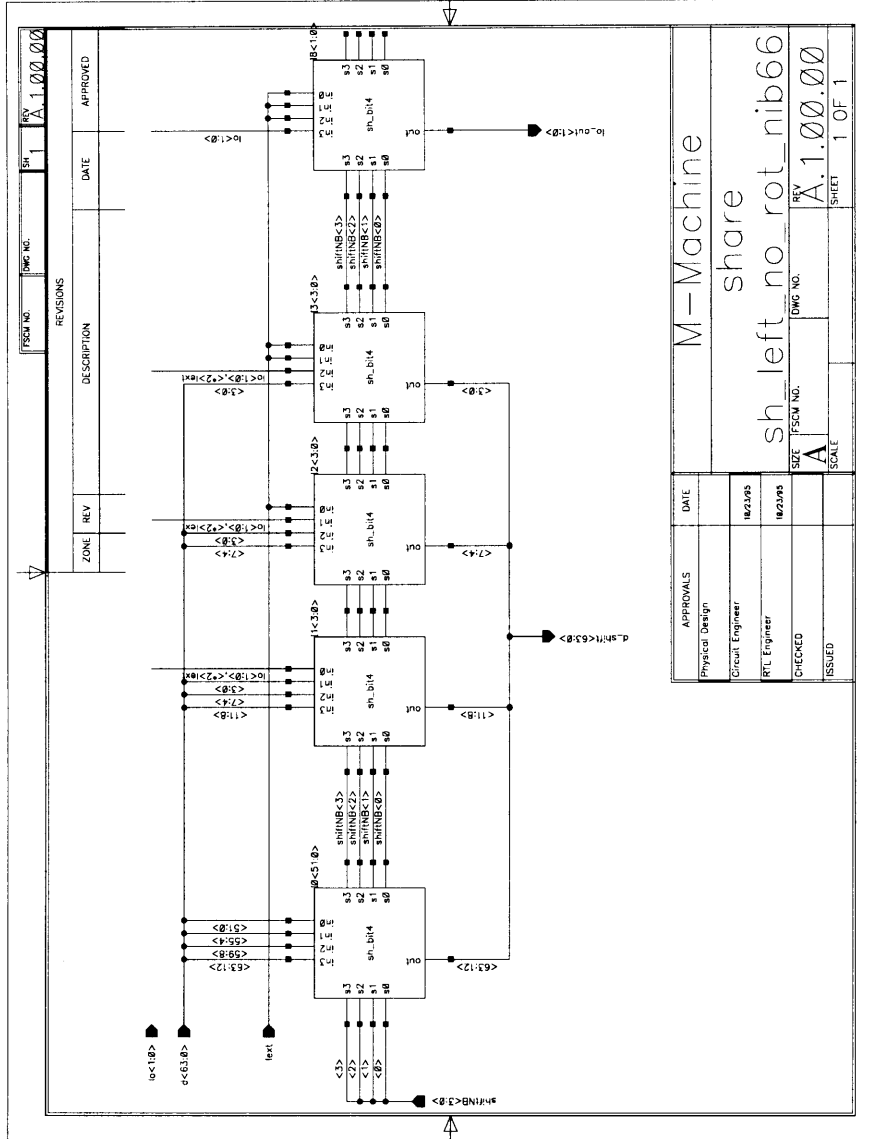
M-Machine	
share	
detect	
APPROVALS	DATE
Physical Design	
Circuit Engineer	10/2/98
RTL Engineer	10/2/98
CHECKED	
ISSUED	
SIZE	DMC NO.
A	
SCALE	REV
	A.1.00.00
	SHEET
	1 OF 1

D.10.4 Shifters

These cells were designed by Jeff Bowers, for use in the Floating Point Divide/Square-root unit.

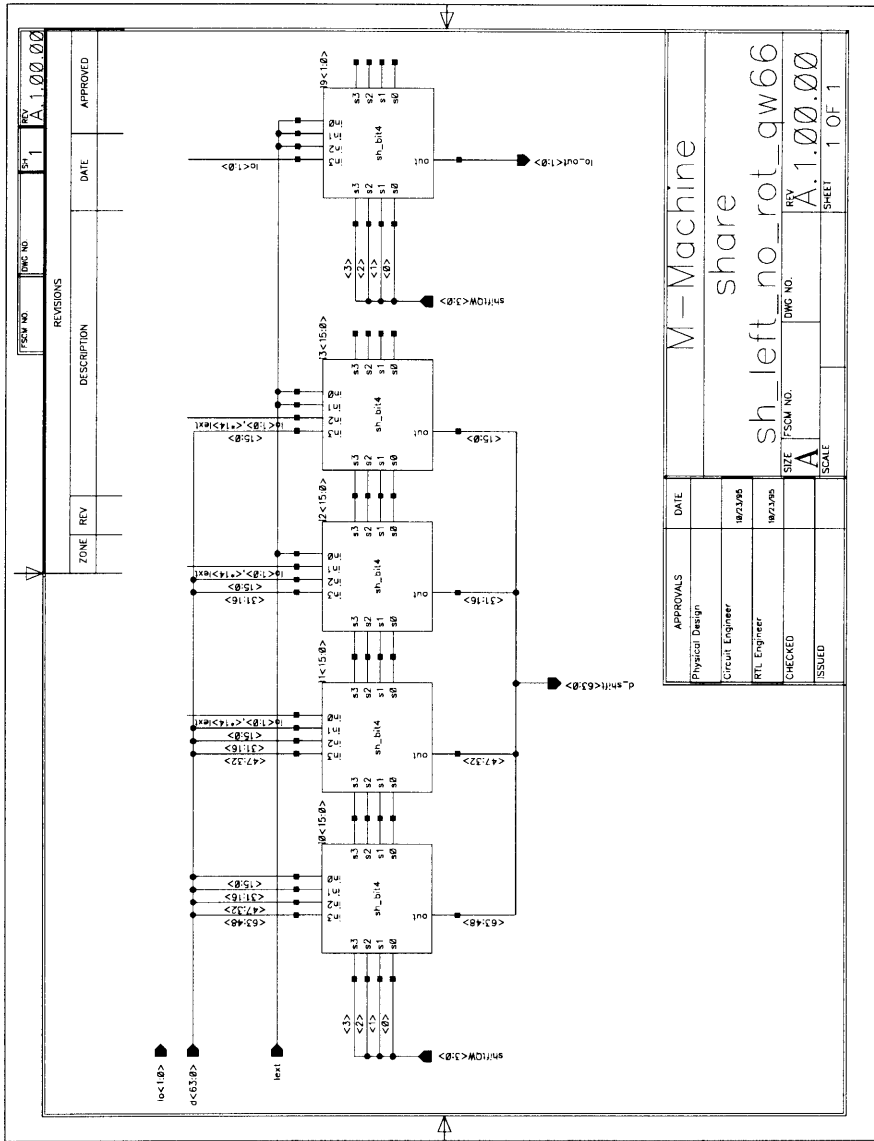


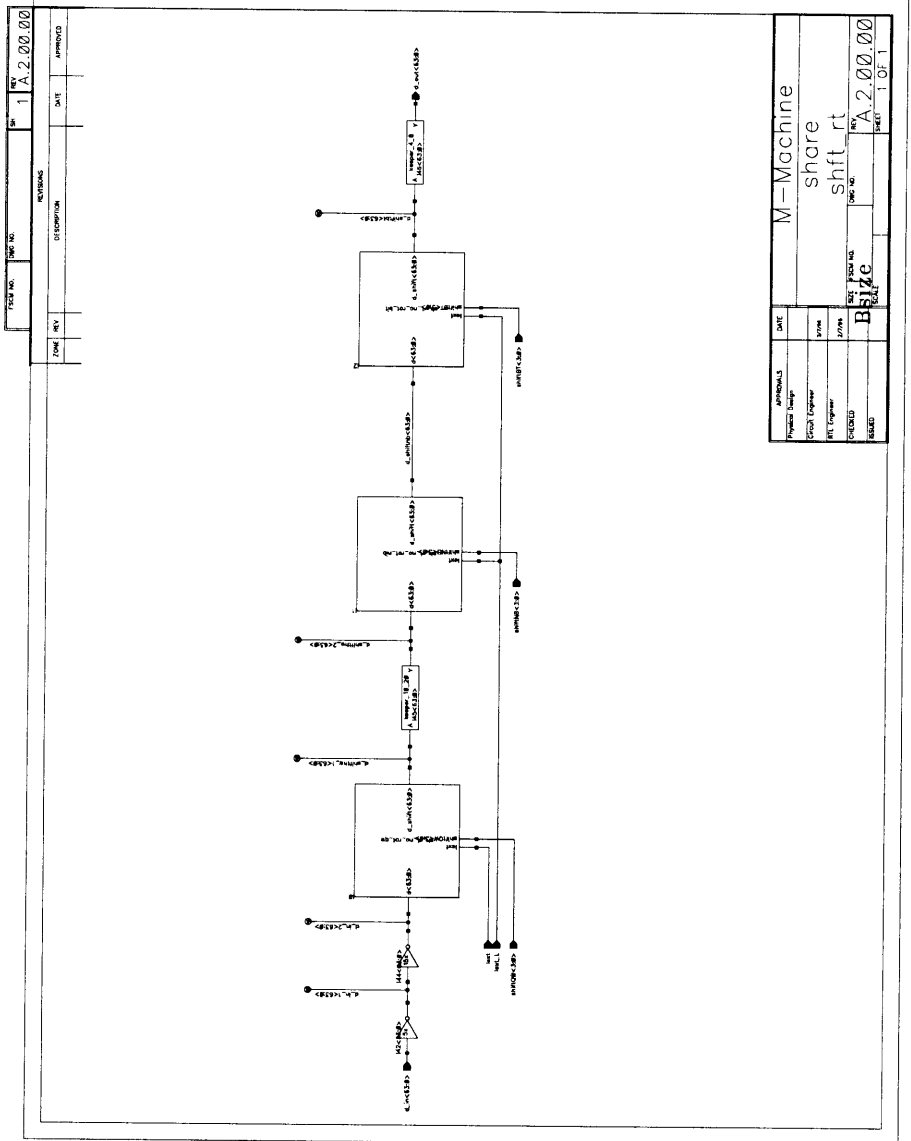




FSCM NO. <u>1</u>		REV. <u>A.1.00.00</u>	
REVISIONS			
ZONE	REV.	DATE	APPROVED
DESCRIPTION			
M-Machine			
share			
sh left no rot_nib66			
SIZE	FSCM NO.	DWG NO.	REV.
A			A.1.00.00
SCALE			SHEET 1 OF 1

APPROVALS	DATE
Physical Design	
Circuit Engineer	18/2/95
RTL Engineer	18/2/95
CHECKED	
ISSUED	



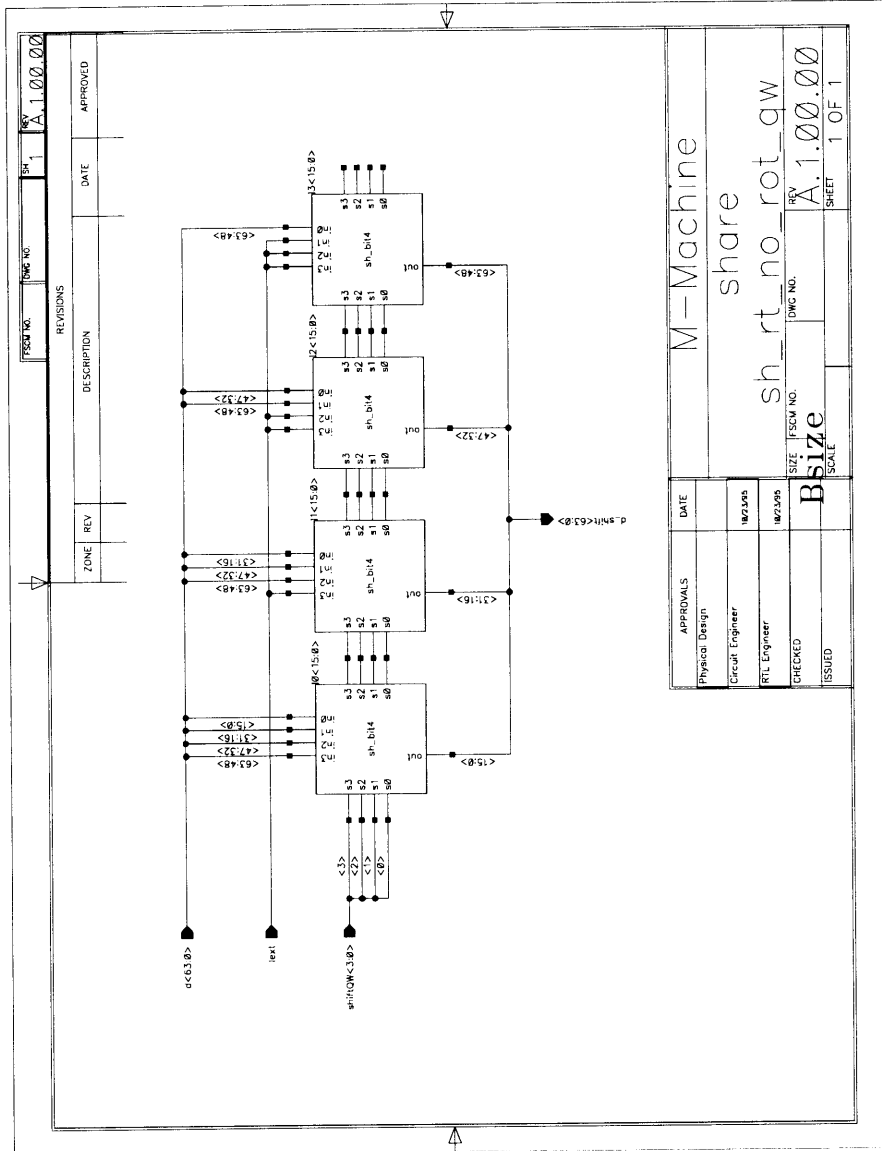


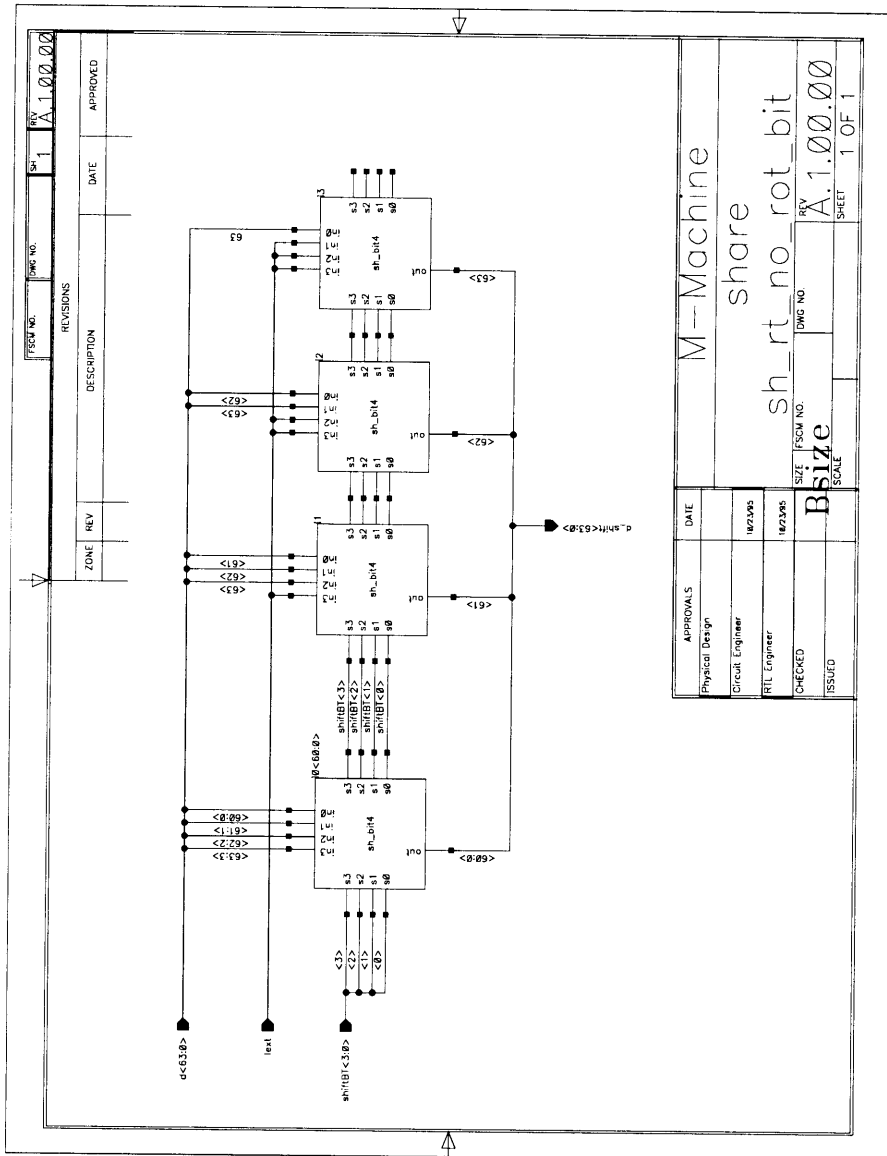
REV	DATE	APPROVED
1	A. 2.00.00	

ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
Project Manager	
Client Engineer	
REV Engineer	1/1/19
DESIGNED	
DRAWN	

M-Machine share shift rt	
REV FROM NO.	REV
1	A. 2.00.00
PAGE 1 OF 1	





REVISIONS		DATE	APPROVED
ZONE	REV		

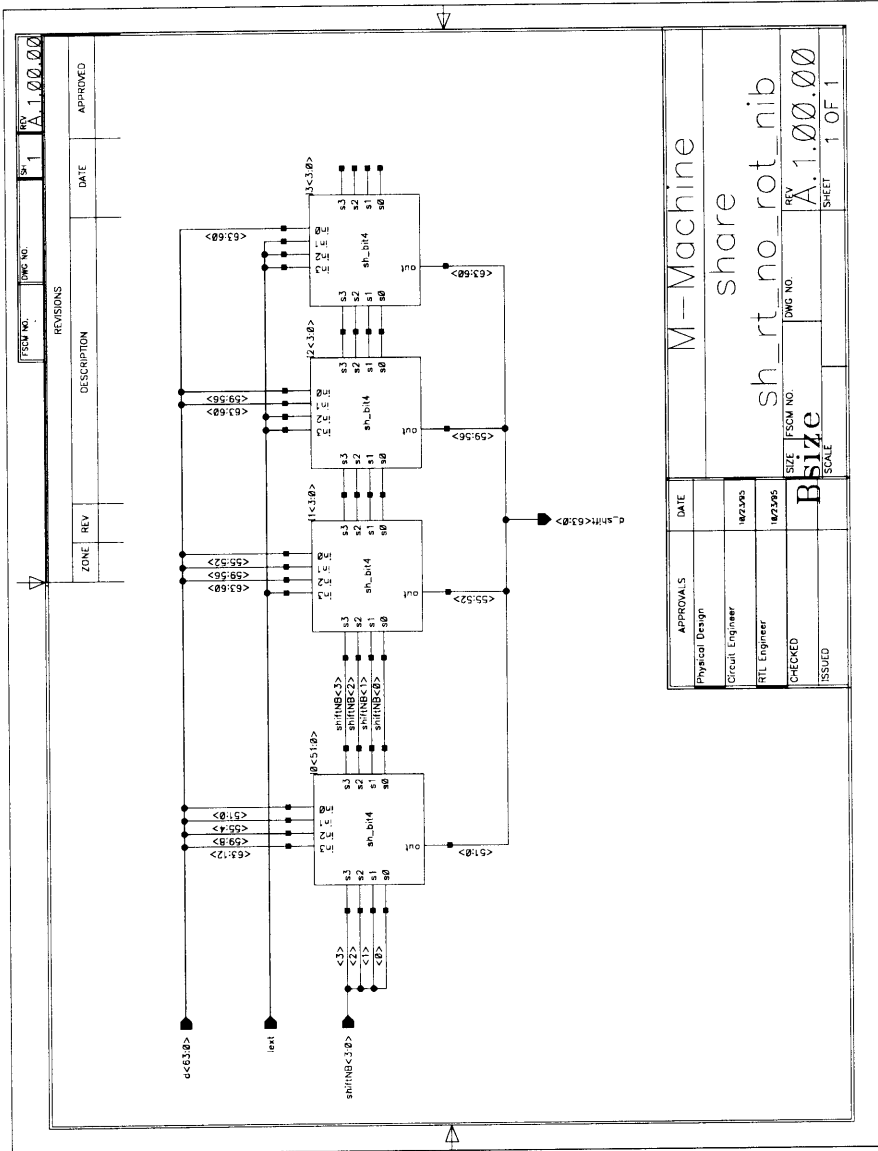
DESCRIPTION	DATE

DATE	APPROVALS
14/03/08	Physical Design
14/03/08	Circuit Engineer
14/03/08	RTL Engineer
	CHECKED
	ISSUED

M--Machine
share
sh_rt_no_rot_bit

SIZE: Bsize
FSCM NO.:
SCALE:

REV: A.1.00.00
DWG NO.:
SHEET: 1 OF 1

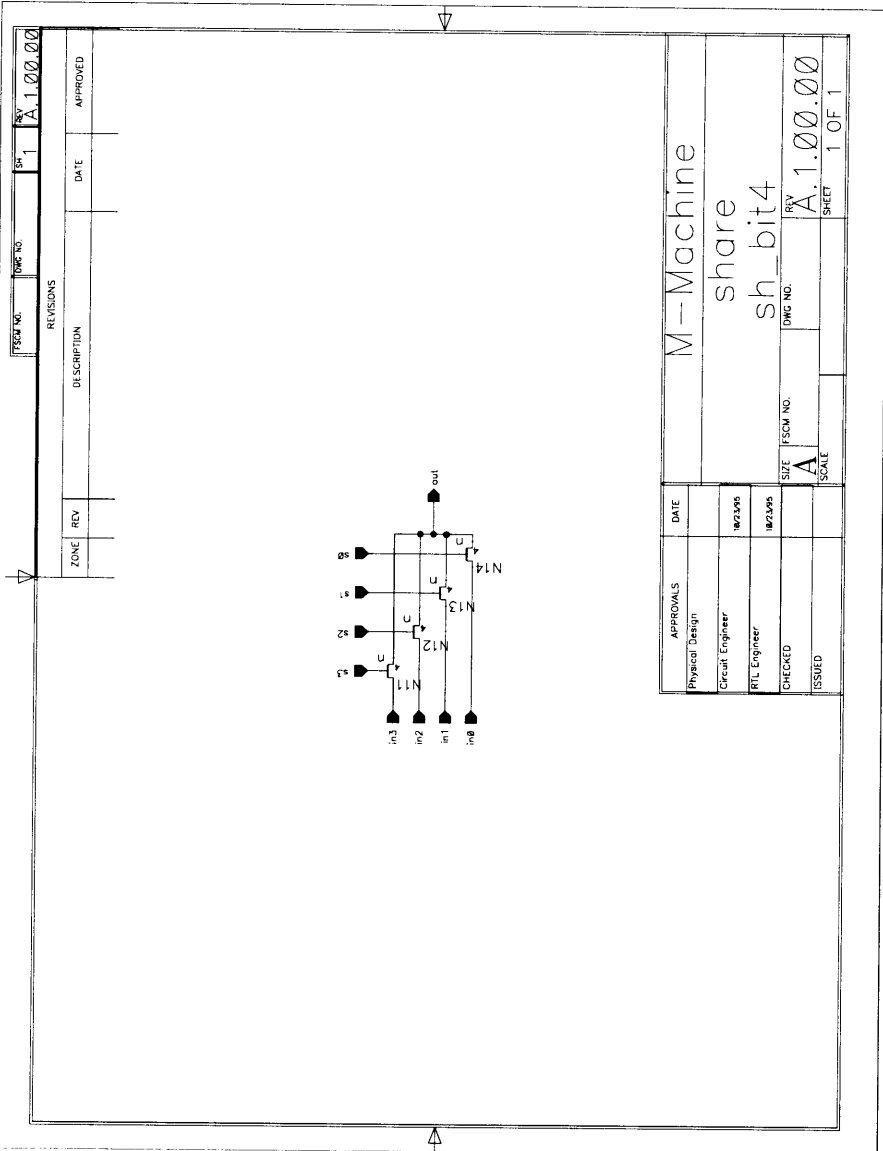


FROM NO.	TO NO.	REV	DATE	APPROVED
		1		

ZONE	REV	DESCRIPTION

APPROVALS		DATE
Physical Design		
Circuit Engineer	18/2/95	
RTL Engineer	18/2/95	
CHECKED		
ISSUED		

M--Machine	
share	
sh_rt_no_rot_nib	
SIZE	FSM NO.
B size	A.1.00.00
SCALE	SHEET 1 OF 1

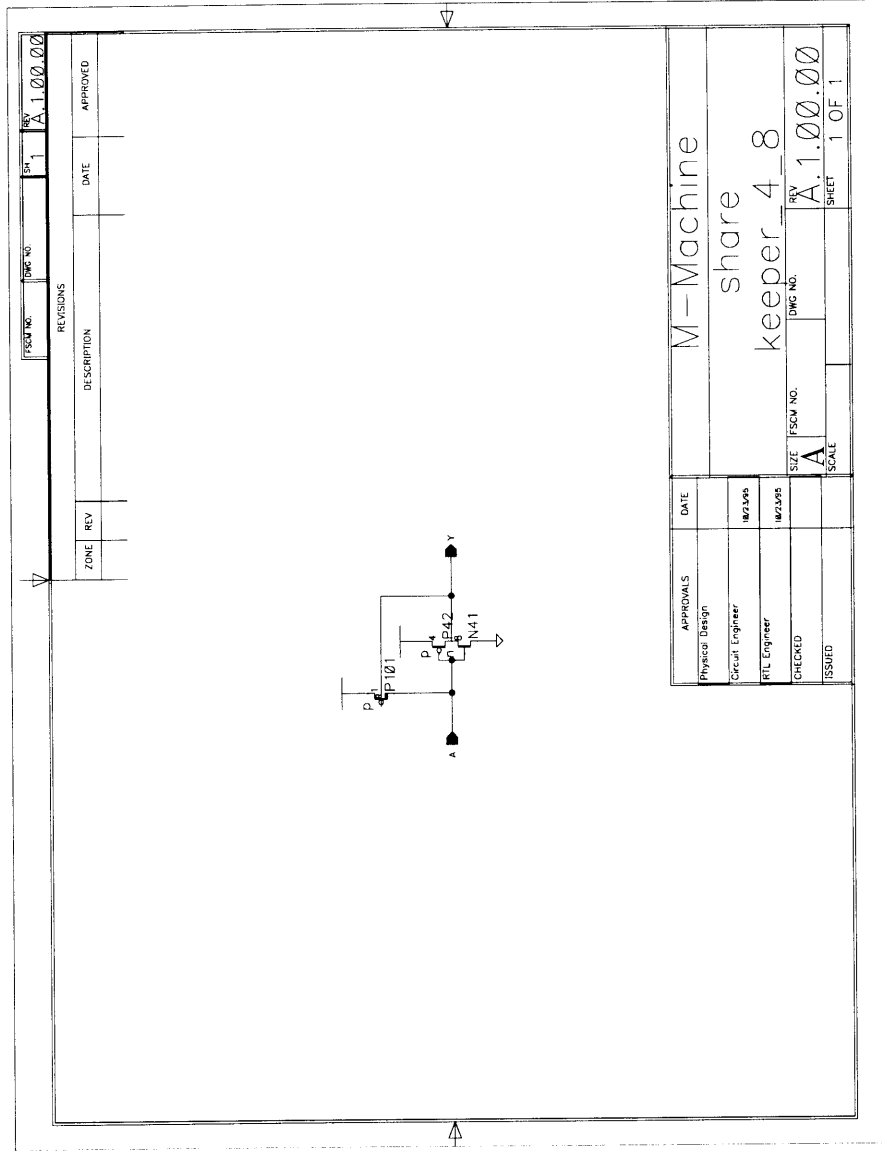


REVISIONS DISCRPTION DATE APPROVED	
ZONE	REV

Dwg No. **A.1.00.00**
 Rev **1**

APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

M--Machine
 share
 sh_bit4
 Dwg No. **A.1.00.00**
 Rev **1**
 Scale **A**
 Sheet **1 OF 1**



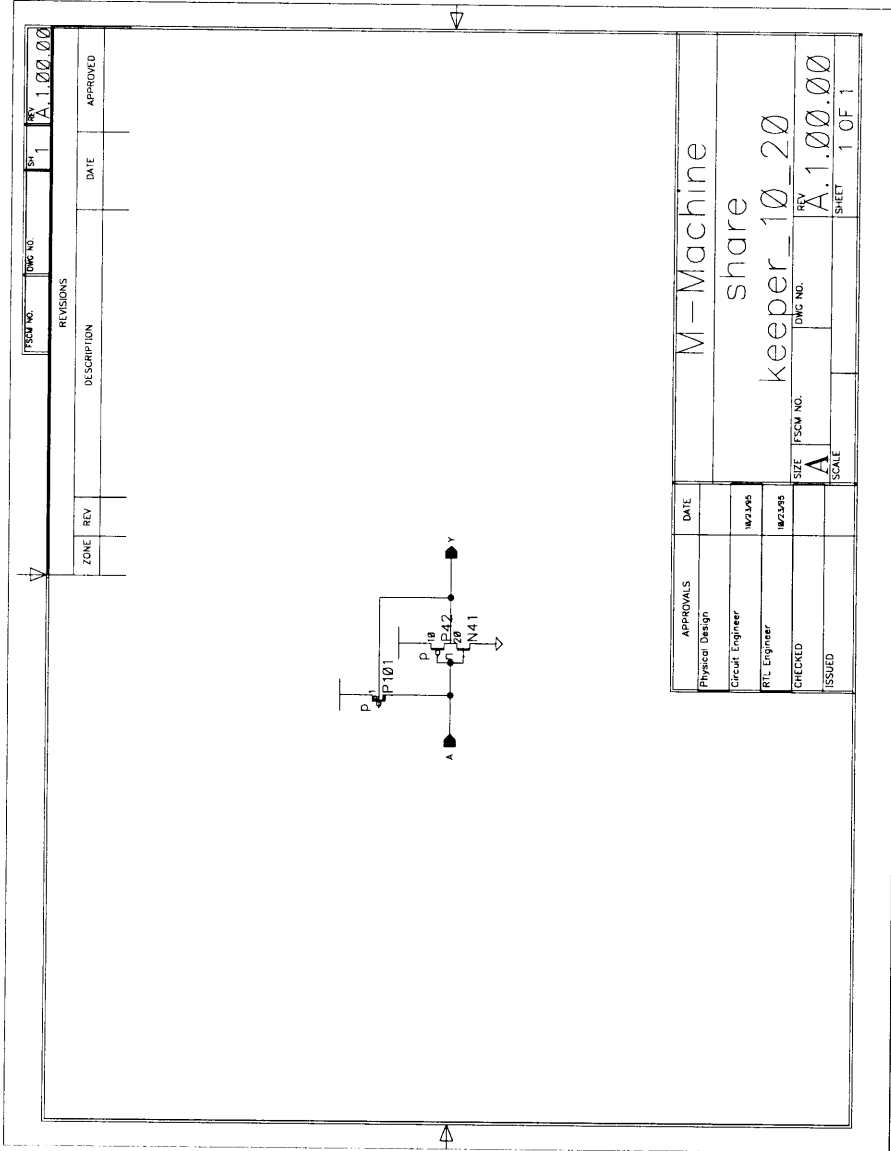
ZONE	REV	DATE	APPROVED

FSCM NO.	DATE	REV
		A.1.00.00

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/23/95
RTL Engineer	10/23/95
CHECKED	
ISSUED	

M-Machine
share
keeper 4 8

SIZE A FSCM NO. DWG NO. REV A.1.00.00
SCALE SHEET 1 OF 1



FSM NO.	REV. NO.	REV.	DATE	APPROVED
		1		A.1.00.00

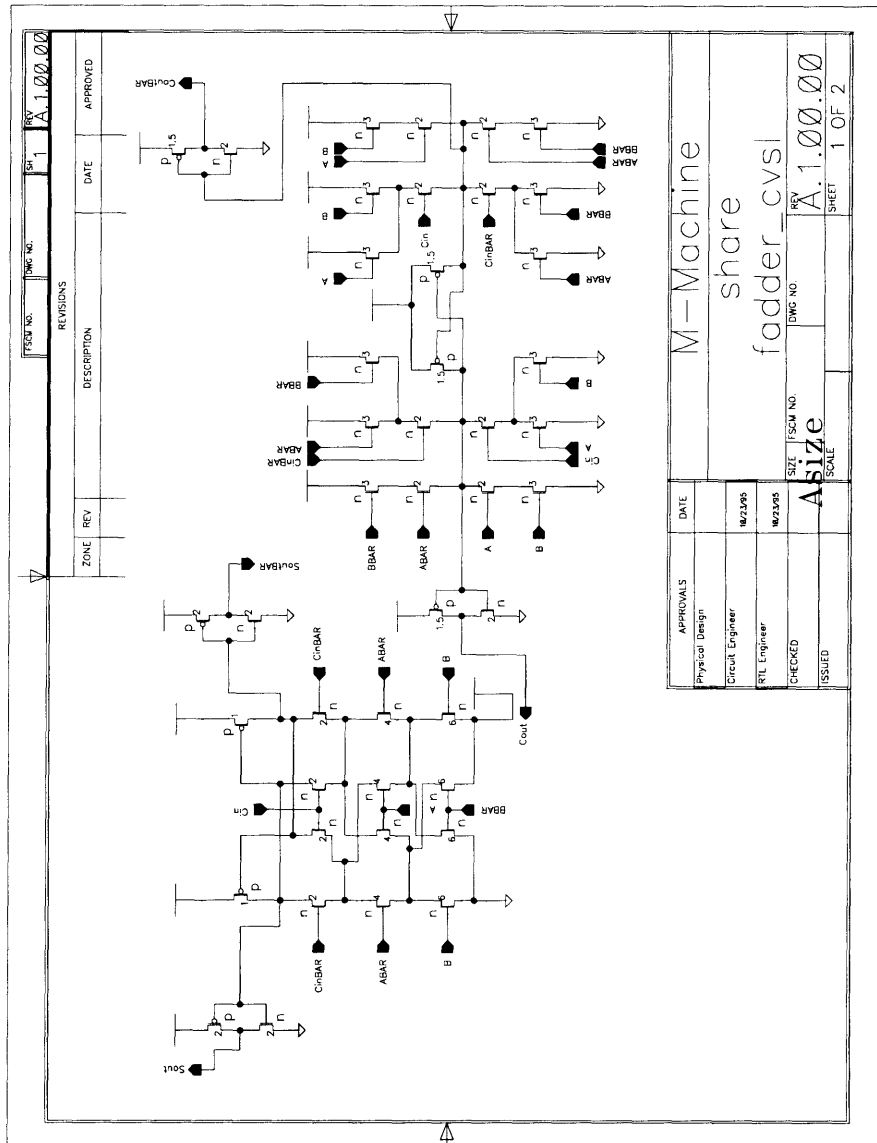
REVISIONS		
ZONE	REV	DESCRIPTION

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/2/06
ATE Engineer	10/2/06
CHECKED	
ISSUED	

M-Machine share keeper	
SIZE	FSM NO.
A	
SCALE	REV
	A.1.00.00
	DRG NO.
	SHEET
	1 OF 1

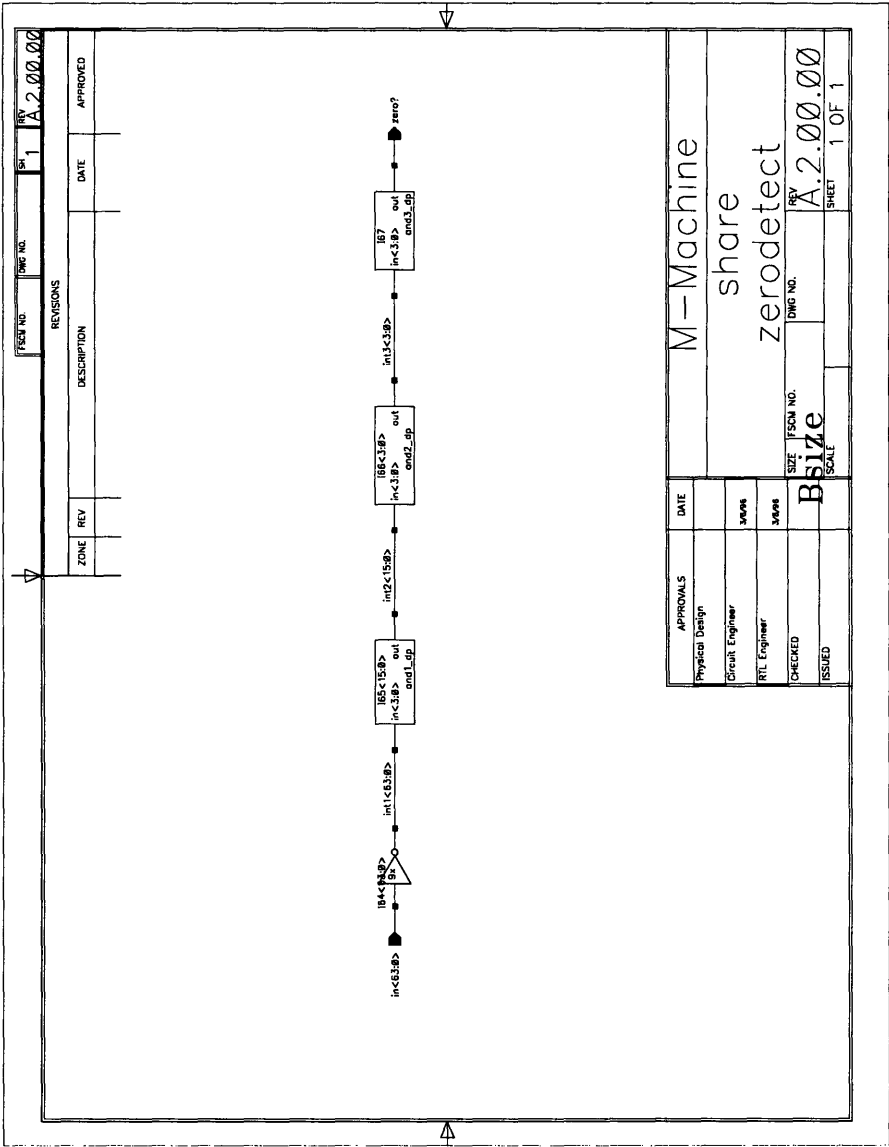
D.10.5 1 bit Adders

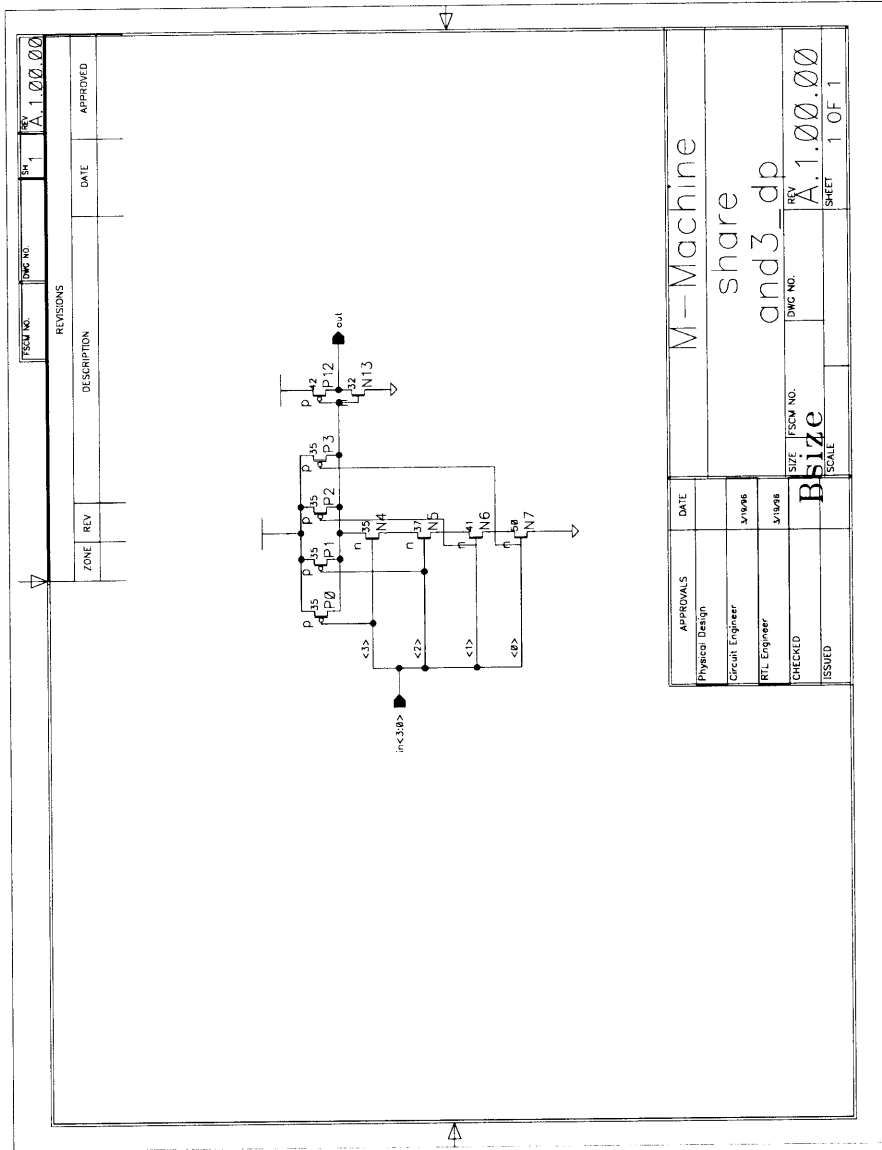
This cell was designed by myself, but modified by Jeff Bowers for use in the shared library.



D.10.6 Zero Detector

These cells were designed by Parag Gupta [5] for the Integer Unit.

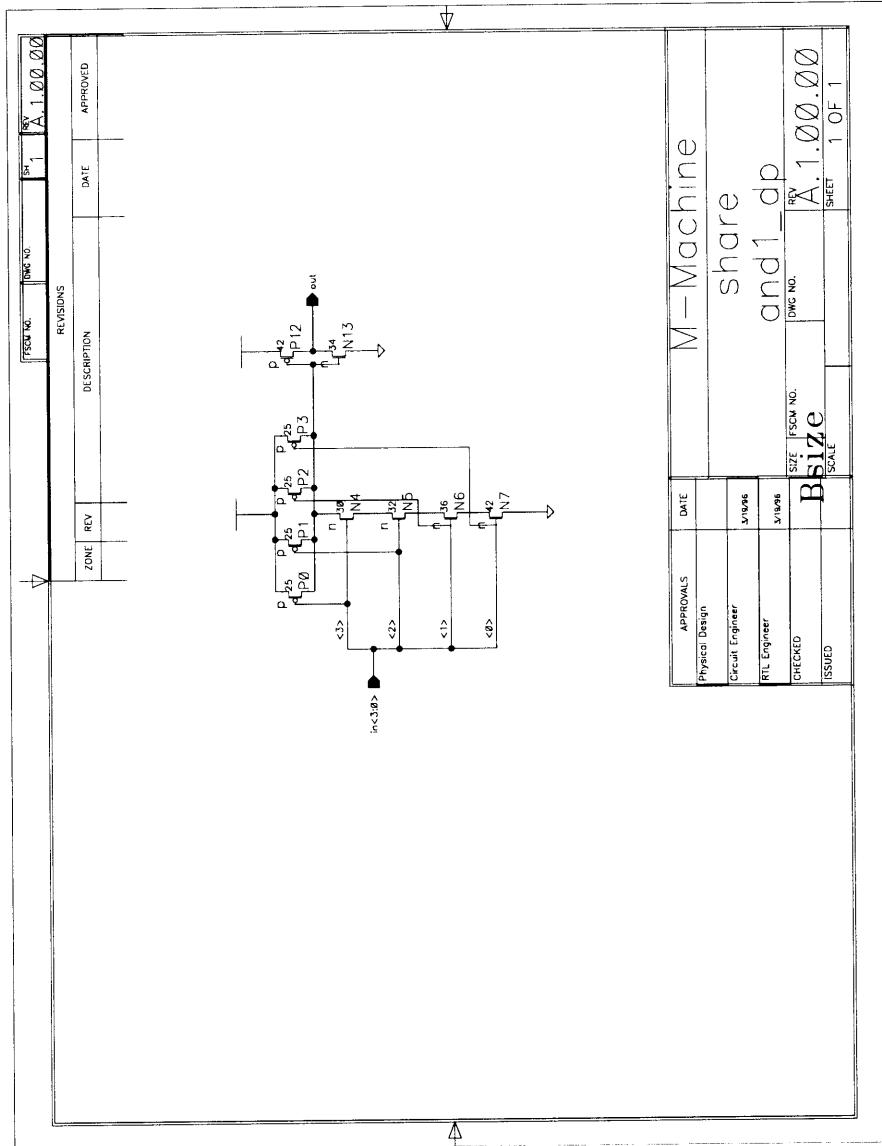




FSCM NO.		DWG NO.		REV		A.1.00.00	
ZONE		REV		DATE		APPROVED	

REVISIONS	
DESCRIPTION	DATE

M-Machine	
share	
and3_dp	
APPROVALS	DATE
Physical Design	
Circuit Engineer	3/18/98
RTL Engineer	3/18/98
CHECKED	
ISSUED	
SIZE	FSCM NO.
Bsize	
SCALE	
DWG NO.	REV
A.1.00.00	A.1.00.00
SHEET	1 OF 1

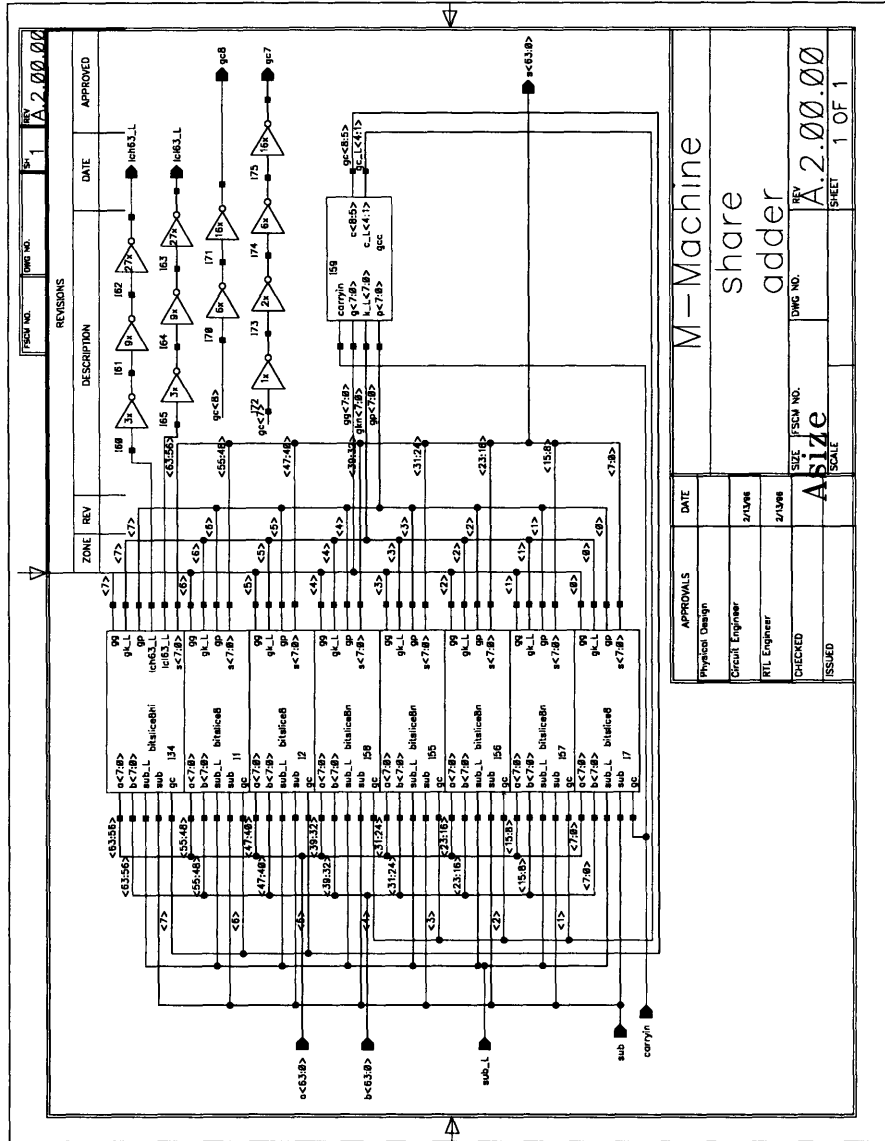


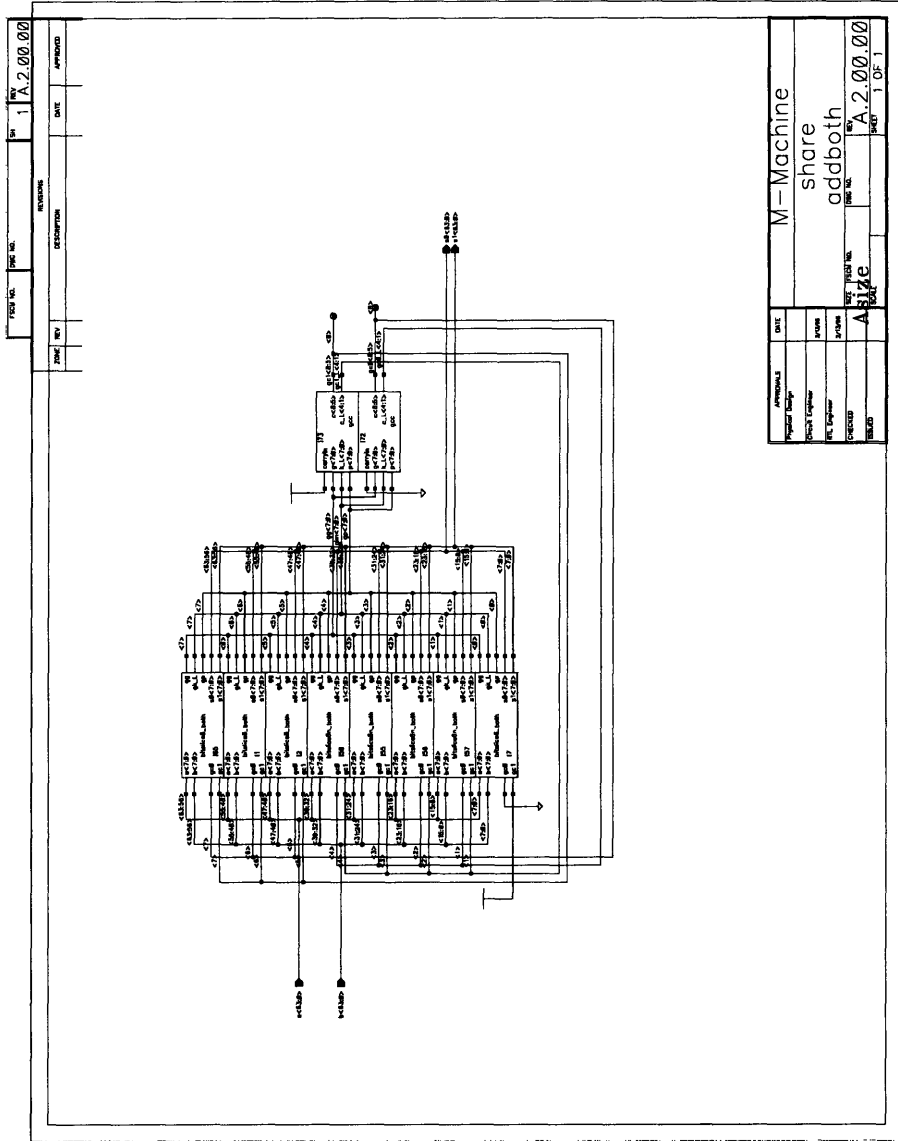
M-Machine	
share	
and1_dp	
	REV A.1.00.00
	SHEET 1 OF 1

APPROVALS	DATE
Physical Design	
Circuit Engineer	3/18/96
RTL Engineer	3/18/96
CHECKED	SIZE FSCM NO.
ISSUED	SCALE

D.10.7 Adders

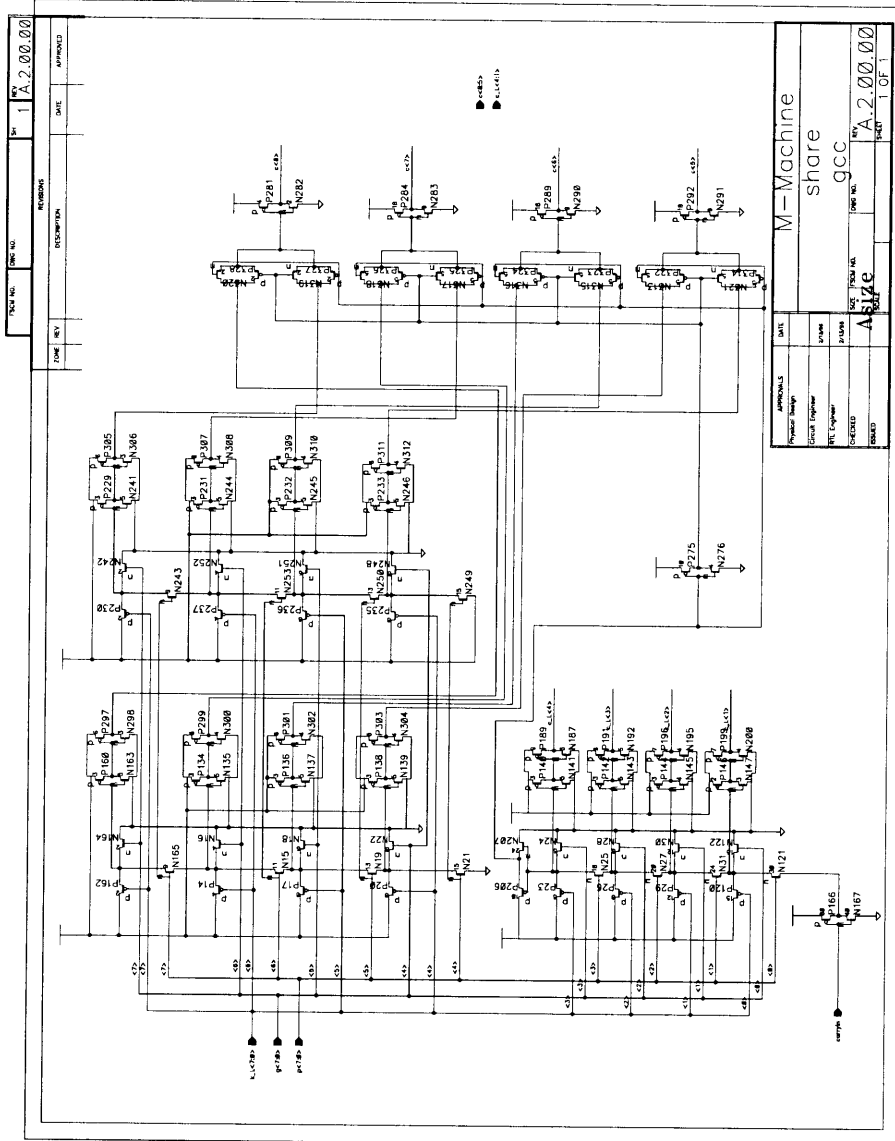
These cells were designed by Parag Gupta [5] for the Integer Unit, with the exception of the adder_both cell.

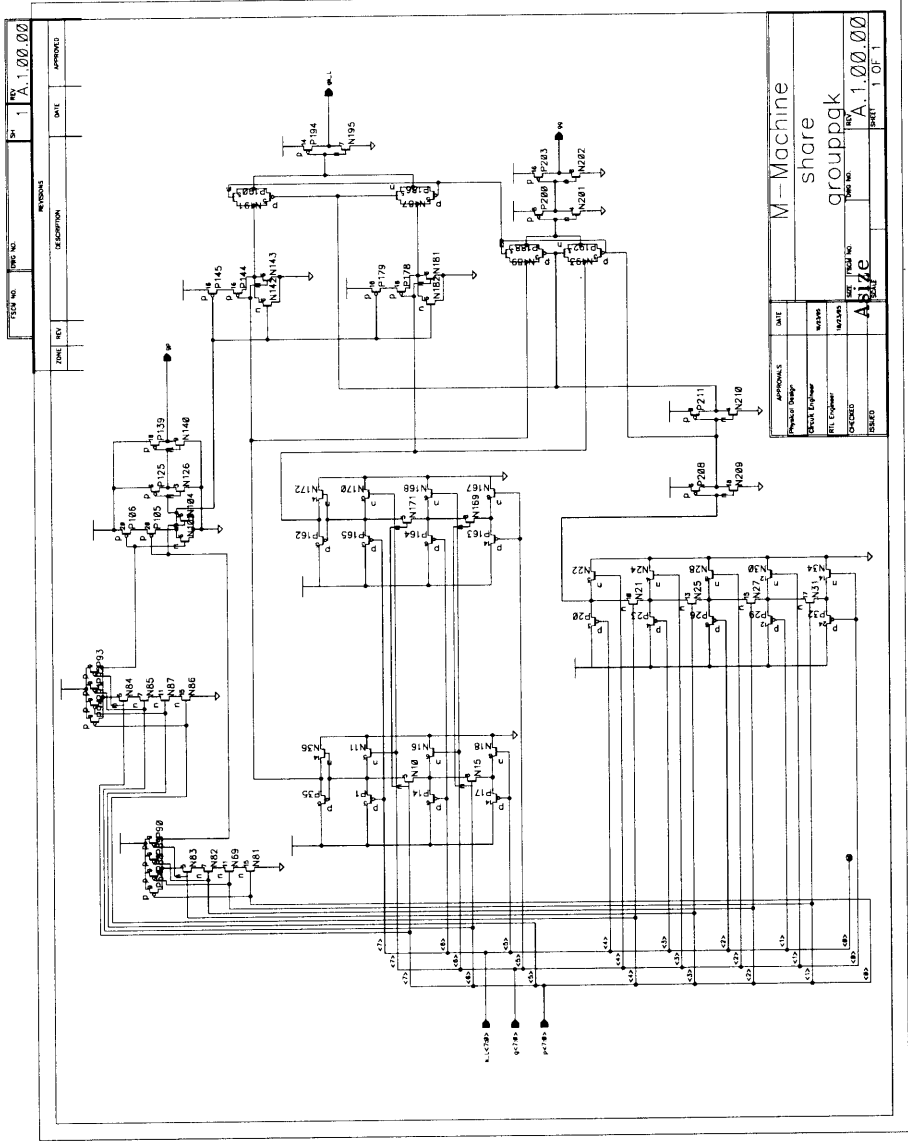




REV	DATE	APPROVED
1	A. 2. 00. 00	

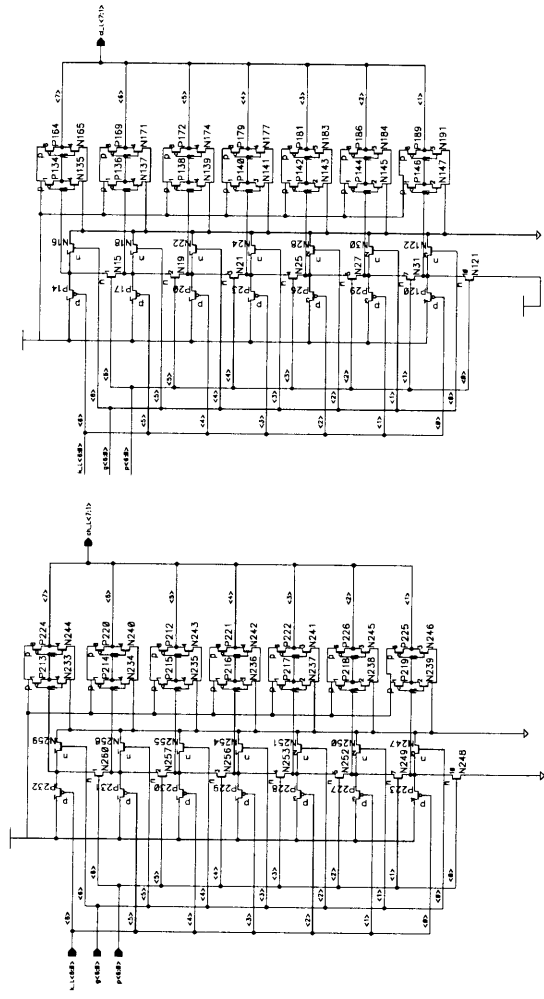
APPROVALS	DATE
Project Engineer	
Check Engineer	
EC Engineer	
DESIGNED	
DATE	
BY	
SCALE	
M-Machine	
share	
adbbath	
REV	DATE
1	A. 2. 00. 00
BY	
SCALE	1:05-1





DATE		M - Machine	
APPROVALS		share	
Project Manager	DATE	group	
Elect. Engineer	DATE	pak	
Bill Engineer	DATE	A.1.00.00	
Checked	DATE	1 OF 1	
Drawn	DATE		
Size	DATE		
Sheet	DATE		

REV	DATE	APPROVED
1	A.3.00.00	
DESCRIPTION		
ZONE	REV	

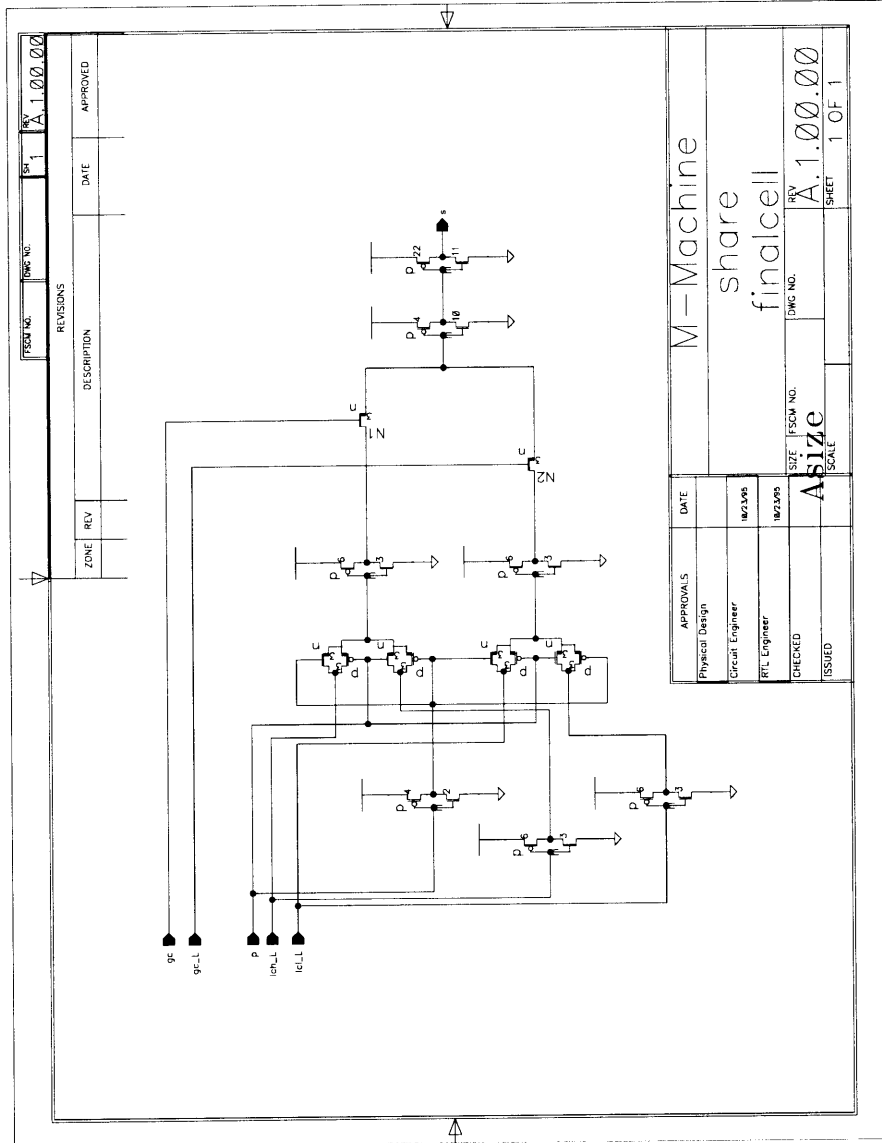


DATE	REV	DATE	APPROVED
DESCRIPTION			
ZONE	REV		

M-Machine
share
ICC

REV: A.3.00.00
PAGE: 1 OF 1

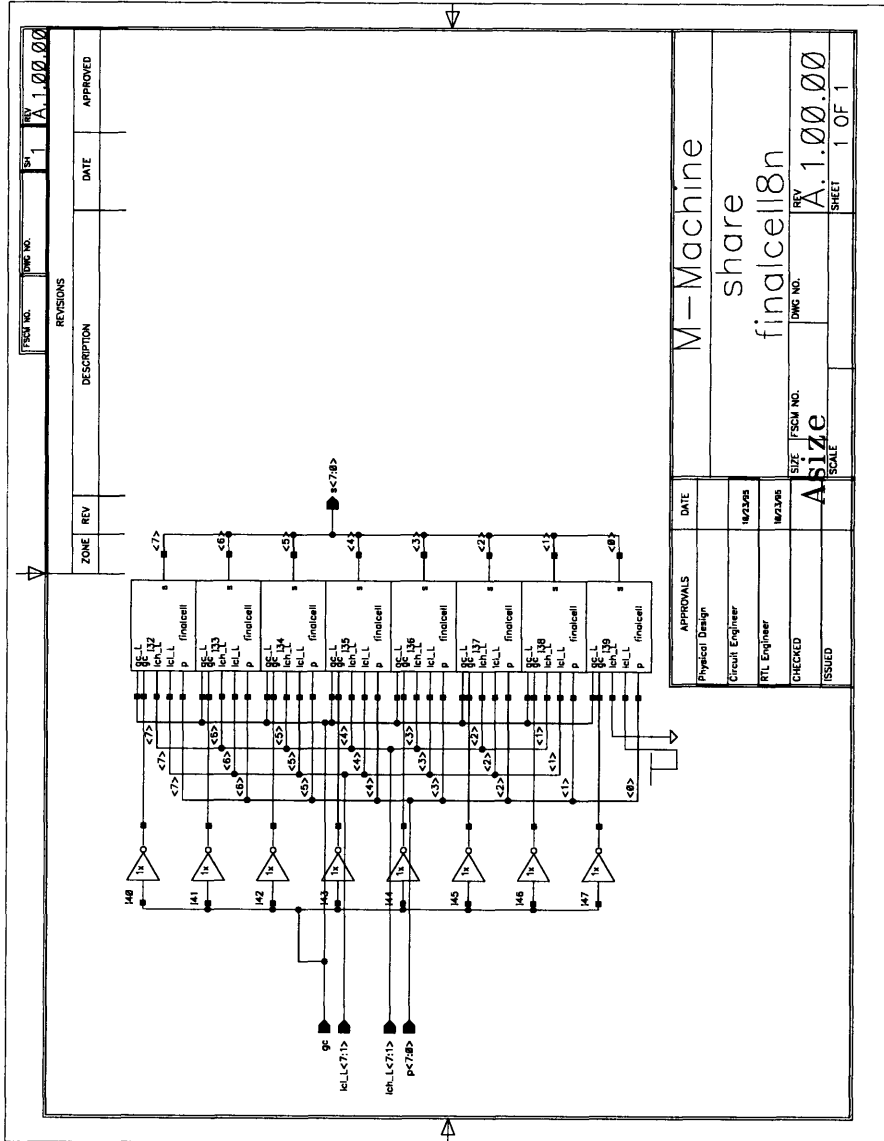
DATE: 3/1/04
DESIGNED: [Name]
CHECKED: [Name]
PHYSICAL DESIGN: [Name]
CIRCUIT ENGINEER: [Name]
FET ENGINEER: [Name]



FSCM NO.	DWG NO.	REV	DATE	APPROVED
		1		

REVISIONS	
ZONE	REV

M-Machine	
share	
finalcell	
APPROVALS	DATE
Physical Design	
Circuit Engineer	18/2/95
RTL Engineer	18/2/95
CHECKED	SIZE
ISSUED	FSCM NO.
	SCALE
	REV
	A.1.00.00
	DWG NO.
	REV
	A.1.00.00
	SHEET
	1 OF 1



REVISIONS		DATE	APPROVED
ZONE	REV	DESCRIPTION	

APPROVALS		DATE
Physical Design		
Circuit Engineer		
RTL Engineer		
CHECKED		
ISSUED		

M-Machine share finalcell8n	
SIZE	SCALE
PSDM NO.	Asize
DWG NO.	A.1.00.00
REV	1
SHEET 1 OF 1	

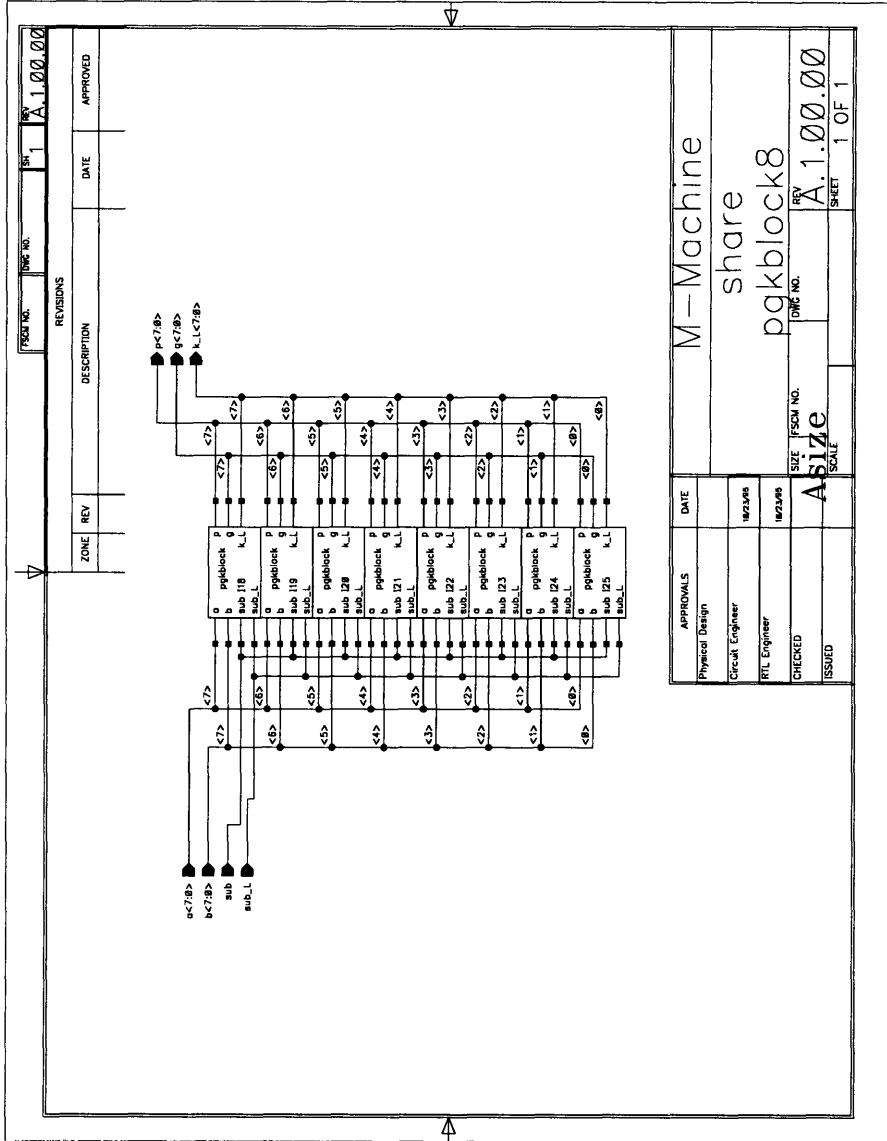


FIG. NO. 1 A.1.00.00

REV. 1

APPROVED

REVISIONS

DESCRIPTION

DATE

ZONE

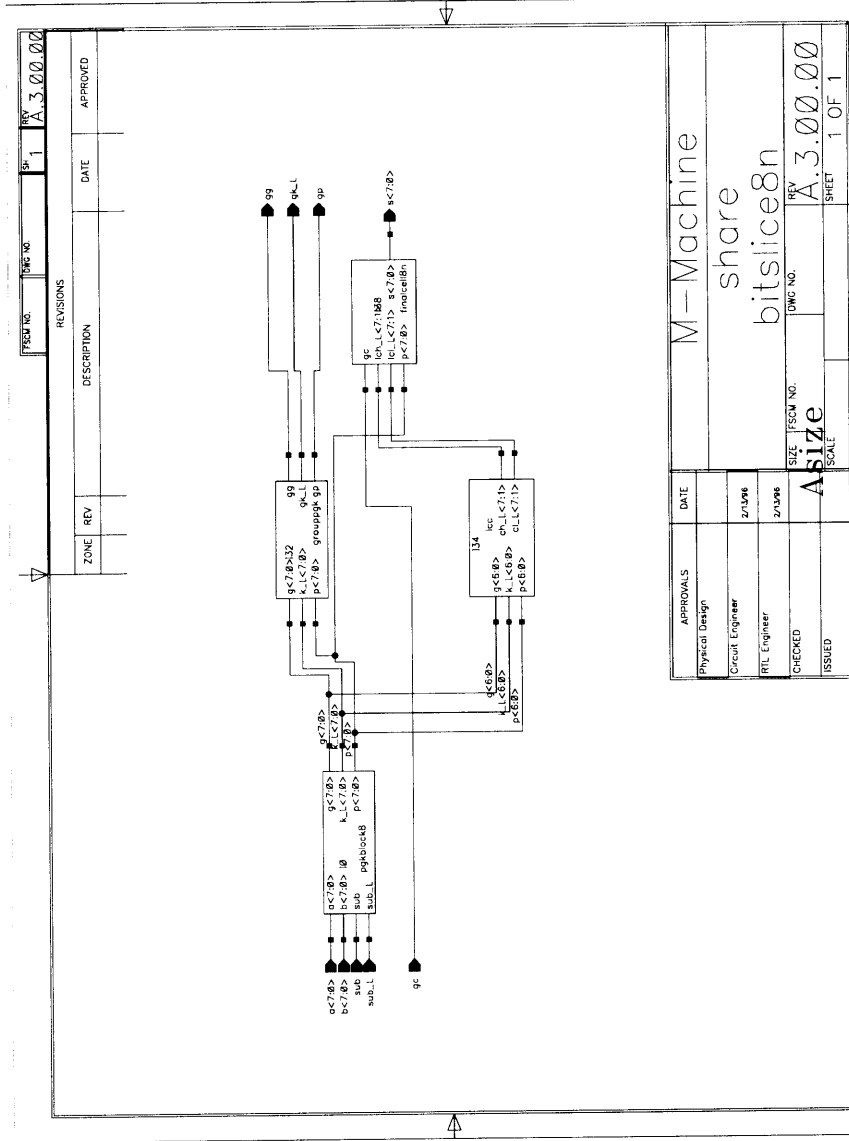
REV

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/21/98
RTL Engineer	10/21/98
CHECKED	
ISSUED	

M-Machine
share
pkgblock8

FIG. NO. 1 A.1.00.00

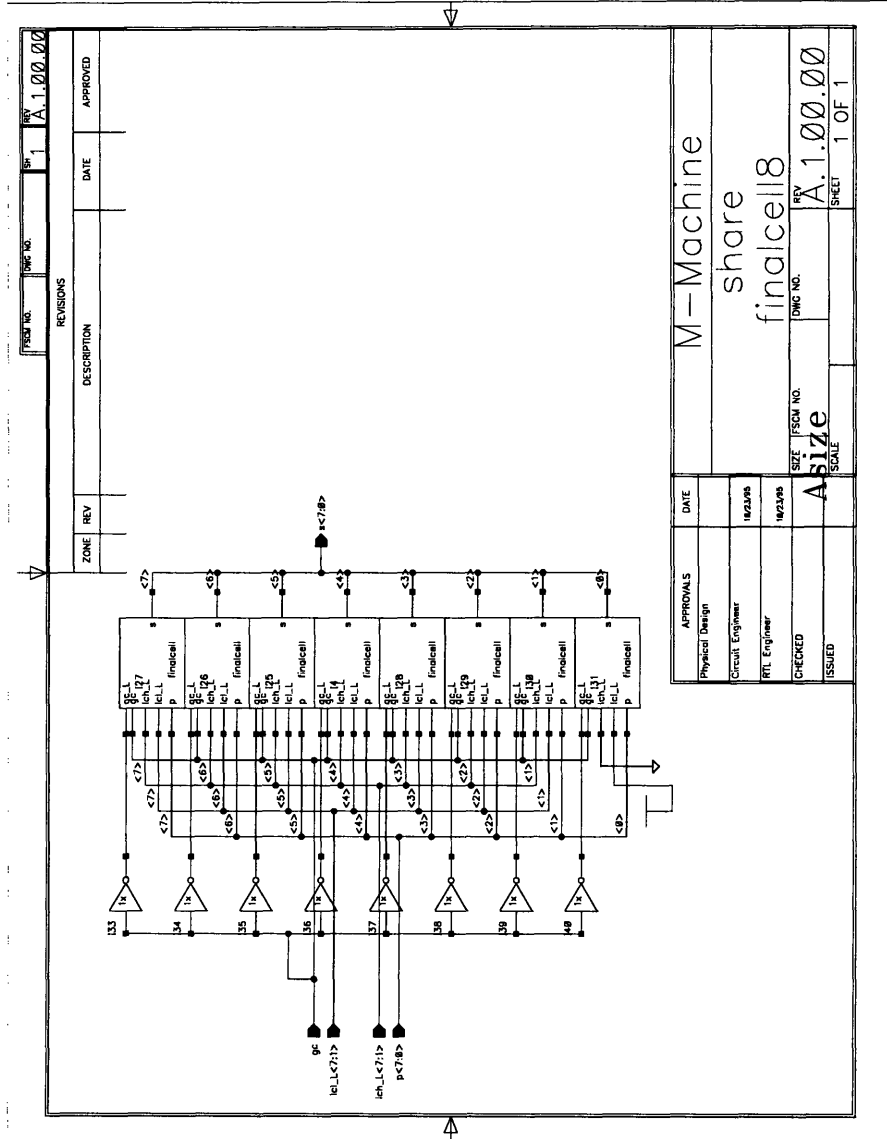
SIZE: A size
SCALE: 1 OF 1



FSM NO.	1	REV	A.3.00.00
ZONE	REV	DATE	APPROVED
REVISIONS			
DESCRIPTION	DATE	APPROVED	

APPROVALS	DATE
Physical Design	
Circuit Engineer	2/1/96
RTL Engineer	2/1/96
CHECKED	
ISSUED	

M-Machine	
share	
bitslice8n	
SIZE	FSM NO.
A.3.00.00	A.3.00.00
SCALE	SHEET
	1 OF 1



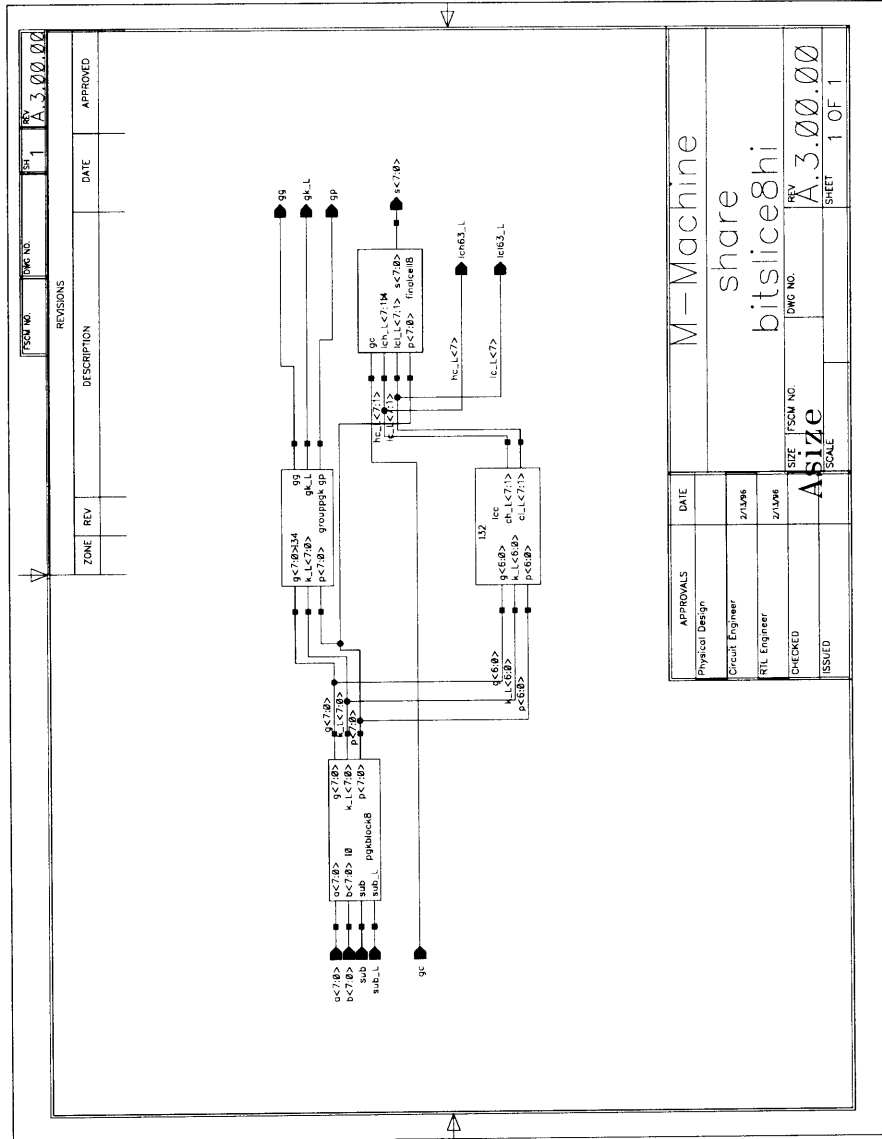
FSCM NO.	REV	DATE	APPROVED
1	A	1.00.00	

REVISIONS		DATE	APPROVED
ZONE	REV		

DESCRIPTION

APPROVALS	DATE
Physical Design	
Circuit Engineer	10/20/00
RTL Engineer	10/20/00
CHECKED	
ISSUED	

M-Machine	SIZE	FSCM NO.	REV
share	A size		A
finalcell8	SCALE	DWG NO.	1.00.00
			1 OF 1

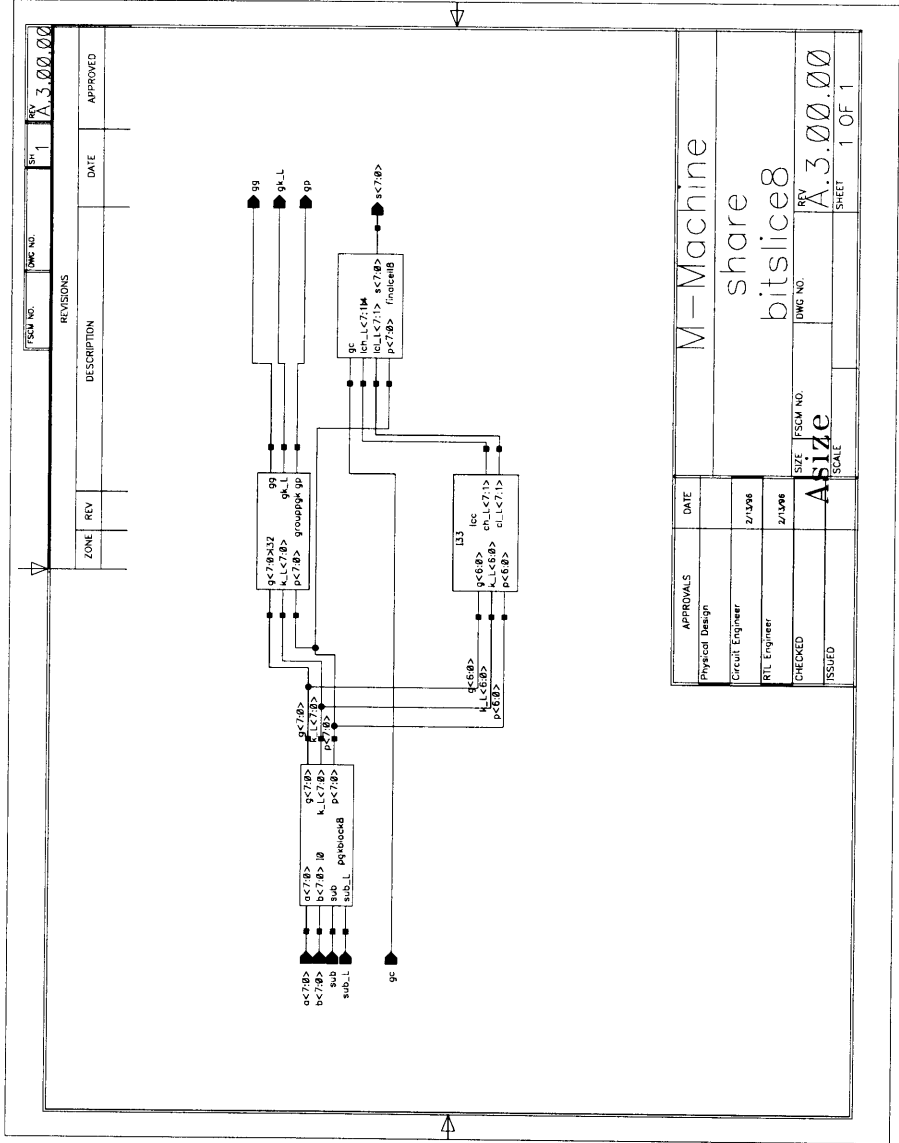


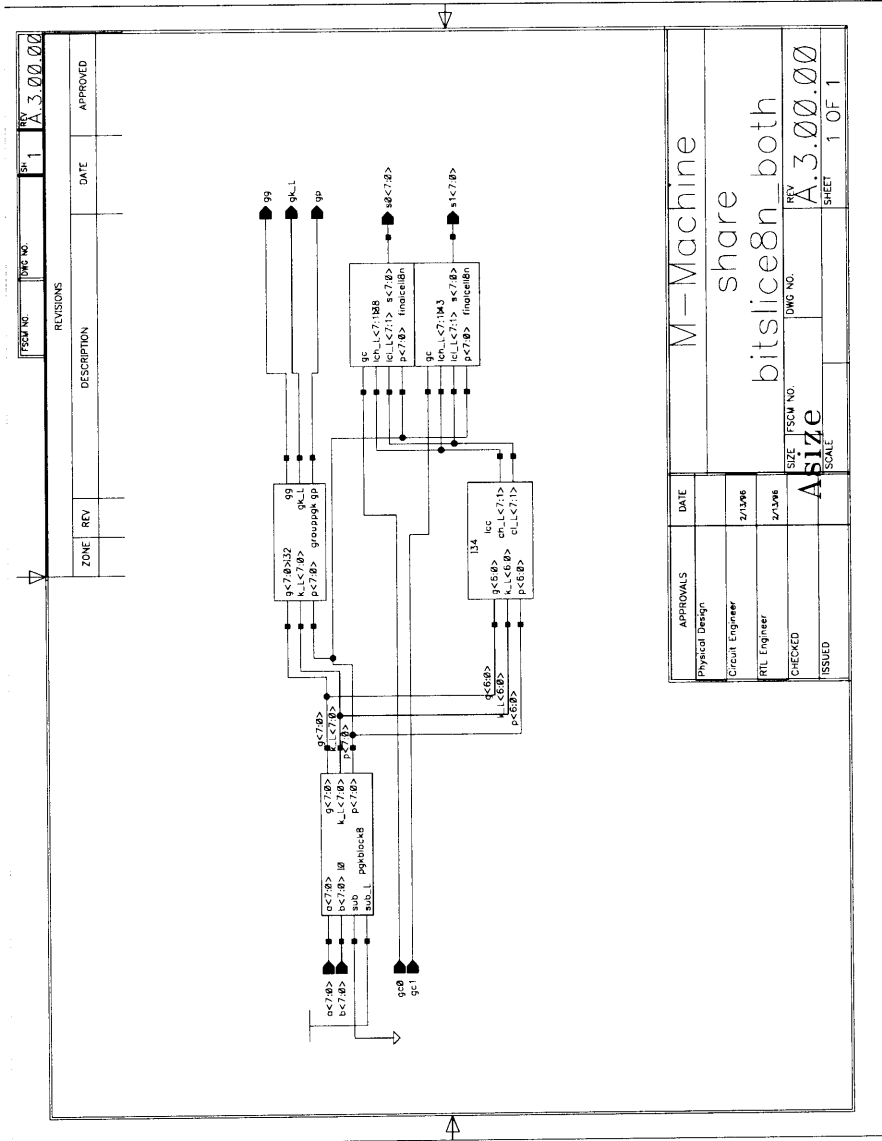
ZONE	REV	DESCRIPTION	DATE	APPROVED

REVISIONS	
1	REV A.3.00.00

APPROVALS	DATE
Physical Design	
Circuit Engineer	2/1/96
RTL Engineer	2/1/96
CHECKED	
ISSUED	

M-Machine	
share	
bitslice8hi	
SIZE	FSCM NO.
ASize	
SCALE	
REV	DWG NO.
A.3.00.00	
SHEET	1 OF 1





FSCM NO.	REV. NO.	REV.	DATE	APPROVED
		1	A.3.00.00	

REVISIONS		DATE	APPROVED
ZONE	REV		

APPROVALS	DATE
Physical Design	
Circuit Engineer	2/13/96
RTL Engineer	2/13/96
CHECKED	
ISSUED	

M-Machine	
share	
bitslice8n_both	
SIZE	DWG NO.
A314	A.3.00.00
SCALE	SHEET
	1 OF 1

Bibliography

- [1] Jeff Bowers. The Floorplan and Layout of the Floating Point Divide/Square-Root Unit. Technical report, Concurrent VLSI Architecture Group, 1988.
- [2] Jerome Coonen. An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic. In *IEEE Computer*. IEEE Computer Society Press, 1980.
- [3] William Dally, Stephen Keckler, Nick Carter, Andrew Chang and Marco Fillo, and Whay Lee. M-machine architecture v1.0. MIT Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, MIT/AI Lab, August 1994.
- [4] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. *The MAP Instruction Set Reference Manual v1.4*. Concurrent VLSI Architecture, MIT A.I. Laboratory, 1996.
- [5] Parag Gupta. Design and Implementation of the Integer Unit Datapath of the MAP Cluster of the M-Machine. Master's thesis, Massachusetts Institute of Technology, 1996.
- [6] Kamran Eshraghian Neil H. E. Weste. *Principles of CMOS VLSI Design, Second Edition*, chapter 8.2. Addison-Wesley, 1993.
- [7] Randy L. Steck Robert P. Colwell. A 0.6um BiCMOS Processor with Dynamic Execution. In *ISSCC Proceedings, February 1995*. Academic Press, 1995.
- [8] M. P. Santoro, G. Bewick, and M. A. Horowitz. Rounding algorithms for IEEE multipliers. In *Proceedings of 9th Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1989.