

1988

Floating point bit-sequential arithmetic units /

David Mark Blaker
Lehigh University

Follow this and additional works at: <https://preserve.lehigh.edu/etd>



Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Blaker, David Mark, "Floating point bit-sequential arithmetic units /" (1988). *Theses and Dissertations*. 4909.
<https://preserve.lehigh.edu/etd/4909>

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

**Floating Point Bit-Sequential
Arithmetic Units**

by

David Mark Blaker

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

1988

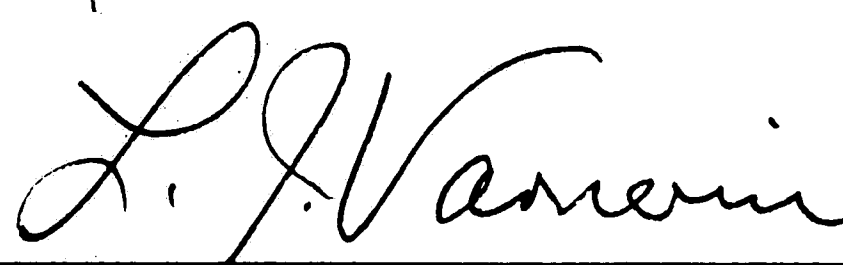
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

JULY 15, 1988

(date)



Professor in Charge



Chairman of Department

Acknowledgements

I am profoundly grateful to my wife, Polly, and to my children, Sarah Elizabeth and Nathan Isaac, for their support and understanding while this thesis was created. They were generous and gracious in making do without me for many evenings and weekends. I could not have finished this job without their love. I would like to thank Dr. Wagh for his guidance and numerous suggestions, without which this thesis would have been incomplete. Thanks are also due to my managers and colleagues at AT&T Bell Laboratories, who have generously allowed me the time and resources to do this work. Finally, I wish to express my gratitude to my parents for raising me and guiding me.

Table of Contents

Abstract	1
1. Introduction	2
1.1 Parallel Processing and VLSI	2
1.2 Fixed Point and Floating Point	3
1.3 Thesis Outline	3
2. On the Suitability of Bit-Sequential Architectures for VLSI	5
2.1 Introduction	5
2.2 Wire Delays in VLSI	6
2.3 Bit-sequential architectures	7
2.4 Fixed Point Bit-Sequential Arithmetic Units	9
3. Floating Point Bit-sequential Multiplier	15
3.1 Introduction	15
3.2 Multiplication Strategy	16
3.3 Detailed Implementation	17
3.4 Conclusion	22
4. Floating Point Bit-sequential Adder	23
4.1 Introduction	23
4.2 Addition Strategy	23
4.3 Detailed Implementation	24
4.4 Conclusion	29
5. Applications for Bit-Sequential Floating-Point Arithmetic Units	30
5.1 Introduction	30
5.2 Matrix-Vector Multiplication	30
5.3 FFT Calculation	32
6. Conclusion	35
6.1 Discussion	35
6.2 Future Directions	35
References	37
Appendix A. Floating Point Bit-Sequential Multiplier Schematics	38
Appendix B. Floating Point Bit-Sequential Adder Schematics	55
Vita	73

List of Figures

Figure 2.1. Bit-Sequential Full Adder	10
Figure 2.2. Semi-Systolic Bit-Sequential Multiplier	11
Figure 2.3. Kaiser, Jackson and McDonald Bit-Sequential Multiplier Section	12
Figure 2.4. High Performance Bit-Sequential Multiplier Section	13
Figure 5.1. Matrix-Vector Multiplier	31
Figure 5.2. Butterfly Processor	33
Figure 5.3. FFT Calculator	34
Figure A.1. FPMPY.1	39
Figure A.2. MANMPY.1	40
Figure A.3. MPY0.1	41
Figure A.4. MPY0.2	42
Figure A.5. MPY.1	43
Figure A.6. MPY.2	44
Figure A.7. MPY23.1	45
Figure A.8. MPY23.2	46
Figure A.9. EXPFMT.1	47
Figure A.10. EXPFMT.2	48
Figure A.11. EXPFMT.3	49
Figure A.12. EXPFMT.4	50
Figure A.13. EXPFMT.5	51
Figure A.14. EXPFMT.6	52
Figure A.15. EXPFMT.7	53
Figure A.16. EXPFMT.8	54
Figure B.1. FPADD.1	56
Figure B.2. FPADD.2	57
Figure B.3. FPADD.3	58
Figure B.4. FPADD.4	59
Figure B.5. FPADD.5	60
Figure B.6. FPADD.6	61
Figure B.7. FPADD.7	62
Figure B.8. FPADD.8	63
Figure B.9. FPADD.9	64
Figure B.10. FPADD.10	65

List of Figures (Cont.)

Figure B.11. FPADD.11	66
Figure B.12. FL1P3AX.1	67
Figure B.13. SW2X2.1	68
Figure B.14. DNRMCTR.1	69
Figure B.15. DNRNCTR.2	70
Figure B.16. EXPCTR.1	71
Figure B.17. RNRMCTR.1	72

Abstract

Computational architectures must be reevaluated in the light of VLSI. In the past, processors and even arithmetic units were constructed from many microelectronic components, interconnected on some substrate. Today, arithmetic units and even whole processors are integrated into a single silicon die, whether in CMOS or bipolar technology. A fundamental problem as VLSI scales to smaller features and higher densities is that the delay associated with wires is becoming a larger and larger portion of the total delay, and threatens continued advances in system speed. At least for a certain class of problems, the answer is to switch to bit-sequential architectures. In these architectures, wire lengths are kept very short, and there is almost no global communications, aside from the clock. For many problems in this class, floating point number representation is necessary, but most of the published work in bit-sequential arithmetic uses the fixed point format. This thesis describes the novel design of a floating point bit-sequential multiplier and adder. 0.33 Mflops addition, and 0.9 Mflops multiplication are achieved with 12,192 and 9,024 transistors, respectively, using a 1.25μ CMOS technology.

Chapter 1

Introduction

1.1 Parallel Processing and VLSI

Recent advances in integrated circuit manufacturing have made custom Very Large Scale Integrated (VLSI) circuits relatively inexpensive and widely available. Computational architectures must be reevaluated in the light of VLSI advances. In the past, processors and arithmetic units used many discrete components placed on a substrate. The current VLSI technology allows arithmetic units and even whole processors to be integrated into a single silicon die, both in CMOS and bipolar circuits. There is even the possibility of integrating multiple processors, at least simple ones, onto a single die. Certainly many processors can be integrated onto a common substrate, and into a common system. However, the design decisions made in that previous era are not necessarily the correct ones in the present era of VLSI.

A fundamental problem as VLSI scales to smaller features and higher densities is that the delay associated with wires is becoming a larger and larger portion of the total delay, and threatens continued advances in system speed. Barring any breakthrough in interconnection technology, such as

superconducting interconnects on VLSI chips, it is necessary to find a way to exploit millions of transistors on a single silicon die, while shrinking the lengths of the wires being used. At least for a certain class of problems, the answer is to switch to bit-sequential architectures. In these architectures, wire lengths are kept very short, and there are almost no global communications, aside from the clock. Another problem in VLSI is the limitation of package pin counts. The advantage of bit-sequential architecture can be more than an order of magnitude reduction in pin counts.

1.2 Fixed Point and Floating Point

There are many times in signal processing when fixed point arithmetic is inadequate, due to limited dynamic range. In these cases, floating point arithmetic is necessary. To date there has been a plethora of papers published about fixed point bit-sequential arithmetic units^{[1] [2] [3]}, and a dearth of papers about floating point bit-sequential arithmetic units. This thesis attempts to redress that lack by reporting the design of two fundamental floating point bit-sequential arithmetic units, a multiplier and an adder.

1.3 Thesis Outline

Chapter 2 explores the argument for bit-sequential arithmetic in more detail, and reviews some examples of fixed point bit-sequential arithmetic units. In chapter 3, the design of a novel floating point bit-sequential multiplier is

reported in detail. Chapter 4 reveals the design of a novel floating point bit-sequential adder. Both these processors accept two floating point operands in VAX format bit-sequentially, and produce their results in the same format. In chapter 5, two applications are described for the arithmetic units developed in this thesis. Chapter 6 states the conclusions.

Chapter 2

On the Suitability of Bit-Sequential Architectures for VLSI

2.1 Introduction

In programming a function on a digital computer, it is always possible to make tradeoffs between the time to run the program, and the space, or amount of memory, which the program uses. A similar, but more complicated situation exists with regard to designing digital hardware, especially with regard to VLSI. Again, a tradeoff can be made between time and space, or in this case amount of hardware. However, in the case of the hardware implementation of a function, there is a cost of communications which is not apparent in the case of software. In the case of hardware, any information which must be passed between two or more separate physical entities must pass over some physical medium. Let us limit our discussion to the case of conducting wires.*

* The advent of optical communication may change the constraints in the near future, but for now, optics is not available to the VLSI designer.

2.2 Wire Delays in VLSI

Wires take up both area and time. VLSI is essentially a planar technology, therefore we discuss the area of wires embedded in a plane. It is a simple matter to extend the argument to the $2\frac{1}{2}$ dimensional case of multiple planes which can communicate only by orthogonal contacts. The cost of a wire is its area in the plane, which is proportional to its length and width.^[4] There is another cost, which is more subtle. Since wires are embedded in a plane or planes, and have finite width, then the perimeter of a module which must connect to n wires is proportional to n . If the module is square, then the minimum area it can possibly have is proportional to $\left(\frac{nP}{4}\right)^2$. It can be seen, then, that it is very important to balance the area of wires and the area of the modules which they connect, lest the VLSI device be overwhelmed by the area of wires. In fact, many VLSI devices today have about half of their area devoted exclusively to wiring.

The other cost of wires is in their delay. As VLSI technology continues to scale to smaller and smaller dimensions, the capacitance of the wiring per unit length does not decrease. This is due to the fact the the capacitance per unit length is proportional to the ratio of width to height above the substrate. As the technology scales in both directions, this ratio remains approximately fixed. Also, the resistance, which is inversely proportional to the width and to the

thickness of the wire, goes up as the square of the scaling. Therefore, the RC time constant of the wire goes up quadratically with the scaling. Now, it is possible to ignore the wire delay, and regard it as a lumped capacitor, if the RC delay of the wire is significantly shorter than the gate delay.^[5] Otherwise, the wire delay becomes a significant part of the total delay. The delay of the gates themselves, however, does scale down. Therefore, we see that the RC time constant is going up approximately quadratically, while the gate delay is going down as the channel lengths.

The result of this analysis is that the length of a "delayless" wire is decreasing. Note, however, that the size of the chips does not decrease. As the technology continues to scale down, designers keep the chip sizes the same, and put in more devices. This is the point: if the architecture of the device is such that there are wires which are in the path which determines the cycle time of the device, then we will soon reach the point where further scaling, while it may increase the density of the device, cannot improve its speed. What can we learn from this argument? In order to gain increases in throughput which are commensurate with the gains in the transistors themselves, it is necessary to reduce the lengths of the wires in the critical paths of the architecture.

2.3 Bit-sequential architectures

The logical conclusion of the above discussion is to use a bit-sequential architecture. In a bit-sequential architecture, data is processed word-parallel-

bit-serial, rather than word-serial-bit-parallel, as in a microprocessor. A bit-sequential architecture is also the ultimate conclusion of a pipelined architecture. In systolic architectures, multiple processing units operate in lockstep with each other, and only communicate with nearest neighbors. In a semi-systolic architecture, this constraint is relaxed somewhat to allow some globally broadcast signals. By processing data one bit at a time, in systolic or semi-systolic architectures, a VLSI device can be expected to run at a speed which more closely tracks the fundamental limits of the technology than a parallel architecture can.

There are other reasons for wishing to use a bit-sequential architecture. Chief among these is the high degree of regularity in designing a larger function as a collection of identical cells. This has the effect of reducing design time, which is critical for VLSI. The problem with having a million transistors on a single die is defining what to do with them, and designing them. Reusing a small collection of modules in large regular arrays is one way of addressing these issues. Another point is that not having to drive large loads means that most of the transistors in a serial architecture will be small. This increases the transistor density of the device. Also, since there are almost no global wires, the density goes up again. Further, there are many fewer transistors, because data is being processed only one bit at a time, rather than all at once. Therefore the area of a bit-serial implementation is much reduced from a parallel

implementation. In the domain of delay, the critical paths are limited to very few gates and very short wires, so the clock rate goes up. Balanced against this higher clock rate and lower area is a much lower throughput.

An important figure of merit for VLSI is the ratio of throughput to area. If the reduction of area counterbalances the decrease in throughput in equal or greater proportion, then it is advantageous to use a bit-sequential architecture. This is especially true in building parallel arrays of computational units to perform computationally intensive calculations, i.e., systolic arrays.

In all fairness and honesty, one must look at when it is not a good idea to use bit-sequential architecture. These seem to be the two extreme cases: when the required throughput is so low that a general purpose processor is more than a match, and when the data rate approaches the maximum clock rate of the technology. When the data rate approaches the maximum clock rate, bit-serial computational units must be interleaved to keep up. At that point, it may become simpler to use a pipelined parallel implementation.

2.4 Fixed Point Bit-Sequential Arithmetic Units

At this point, it is useful to survey some existing bit-sequential implementations. Since this thesis is limited to a floating point multiplier and adder, the discussion will be limited to fixed point multipliers and adders. A bit-sequential full adder is simplicity itself (see Figure 2.1).

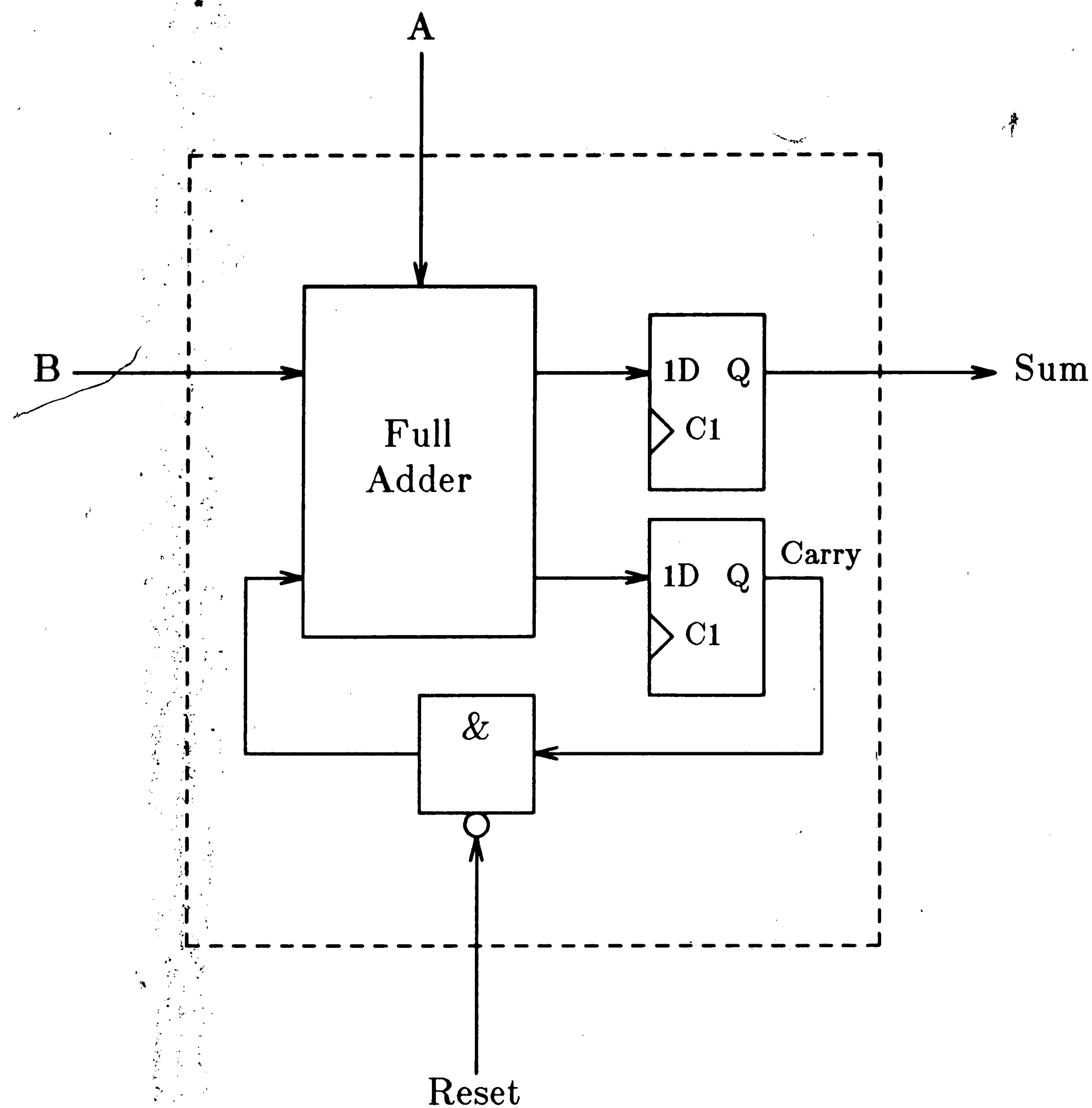


Figure 2.1. Bit-Sequential Full Adder

It consists of a full adder, two D flip-flops and an AND gate. Both operands are input one bit at a time, least significant bit first. The output also appears one bit at a time, least significant bit first, delayed from the input by one clock cycle.

There are many bit-sequential multipliers in the literature. Two fundamentally different approaches are described here. One is purely systolic, and accepts both operands in bit-sequential form. The other is semi-systolic,

and is actually a parallel-serial multiplier. The semi-systolic multiplier receives one of its operands in parallel, and one in serial form^[6] (see Figure 2.2).

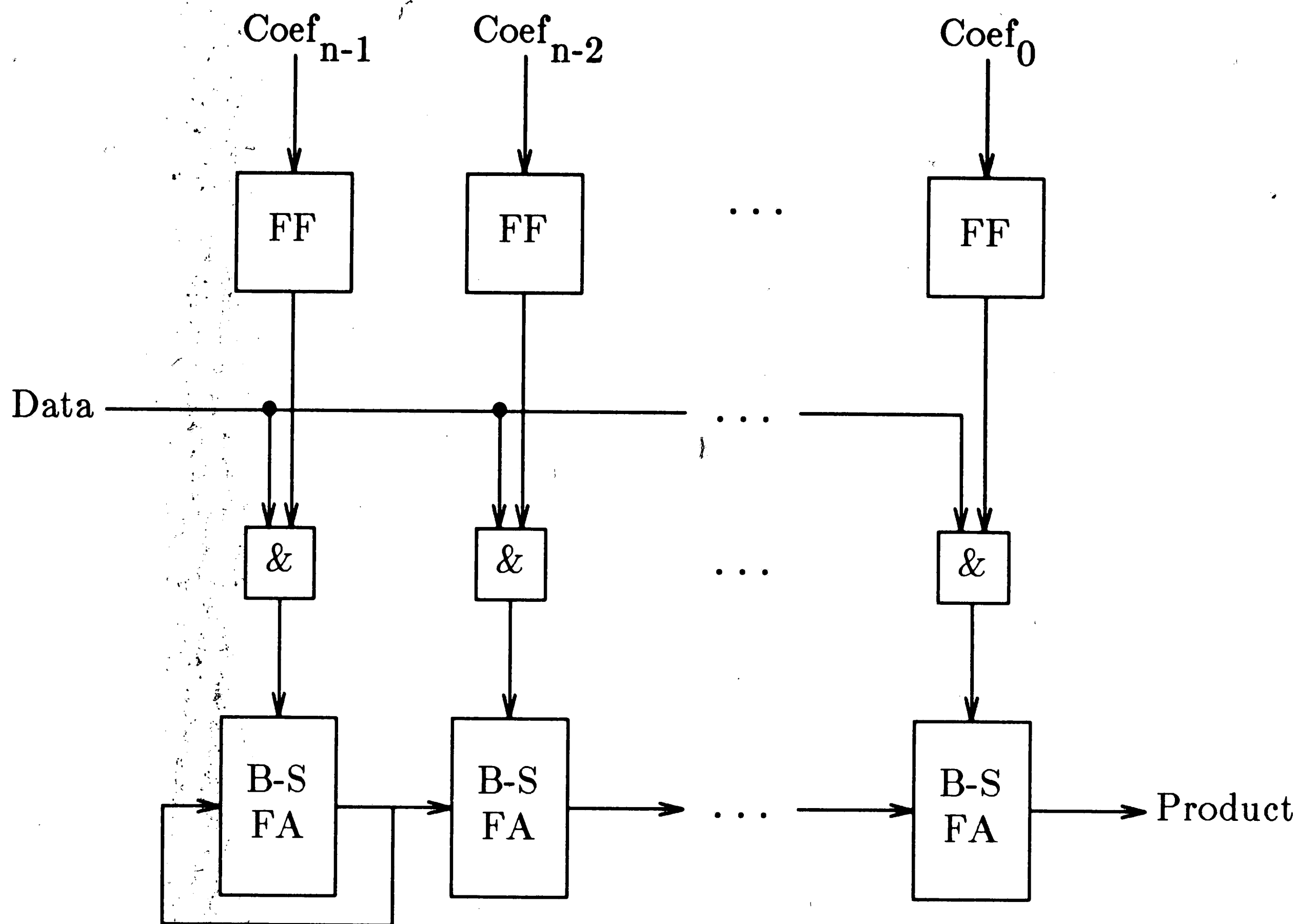


Figure 2.2. Semi-Systolic Bit-Sequential Multiplier

The multiplier multiplies two n -bit numbers, producing a $2n$ -bit result every $2n$ clock cycles. The delay is 1 clock cycle. It is semi-systolic because the serial input is broadcast to every cell.

The original bit-serial multiplier seems to have been described by Jackson, Kaiser and McDonald^[7] (see Figure 2.3). n of these sections may be concatenated to form an n bit multiplier. In this case, both operands are input serially, and there are no broadcast signals. Two n -bit numbers are multiplied

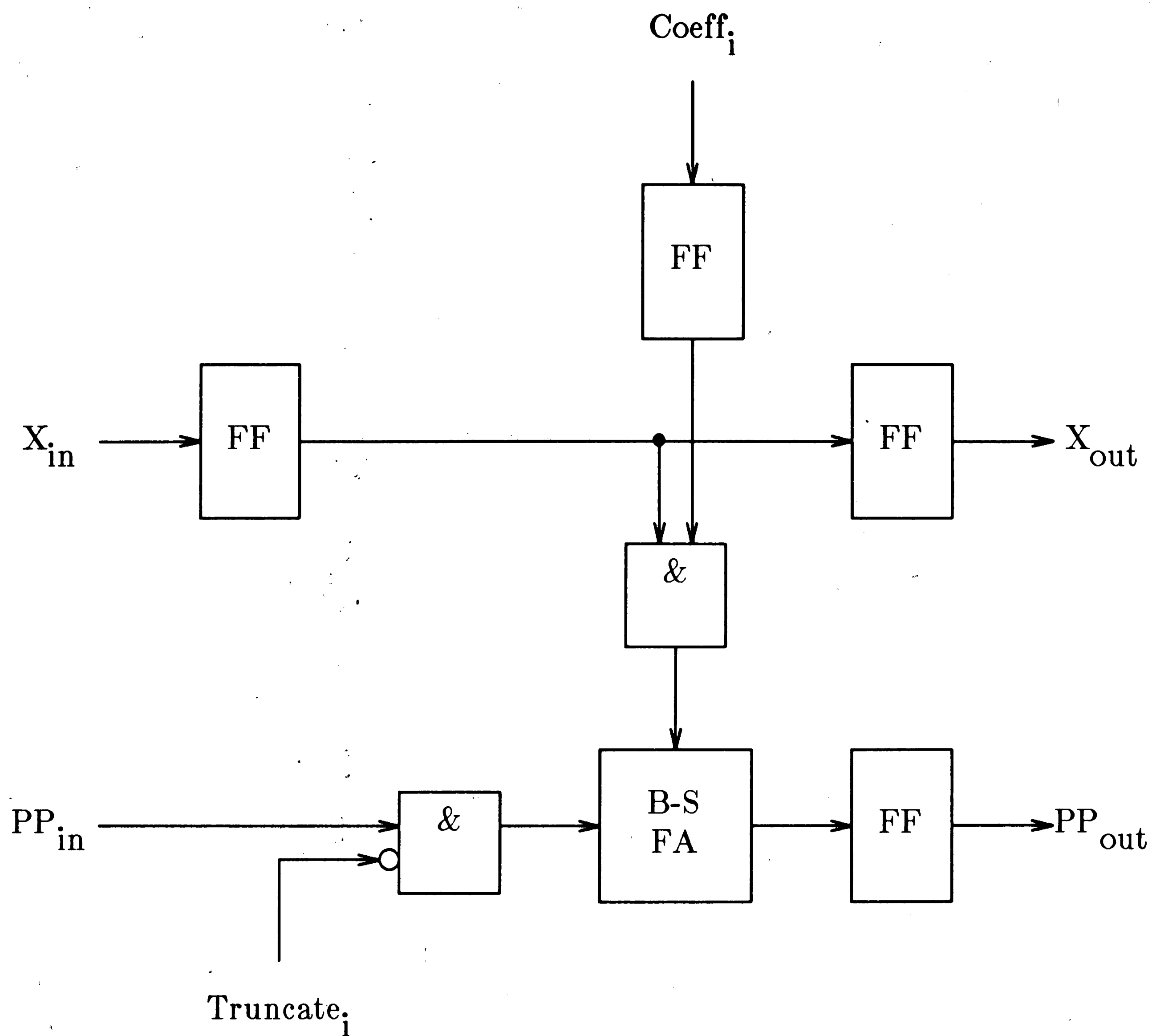


Figure 2.3. Kaiser, Jackson and McDonald Bit-Sequential Multiplier Section to produce a truncated, n -bit product every n clock cycles. The truncate input is the same signal as the reset to the bit-sequential full adder. The delay of each section is two clock cycles. The truncation has the effect of right shifting the partial product produced in each cell to align it with the partial product in the next cell, much as is done in manual multiplication.

The fixed point multiplier which is used in this thesis is a variation of the Kaiser, Jackson, McDonald multiplier, which was described by Scanlon and Fuchs^[8] at the 1986 IEEE International Conference on Computer Design (see Figure 2.4).

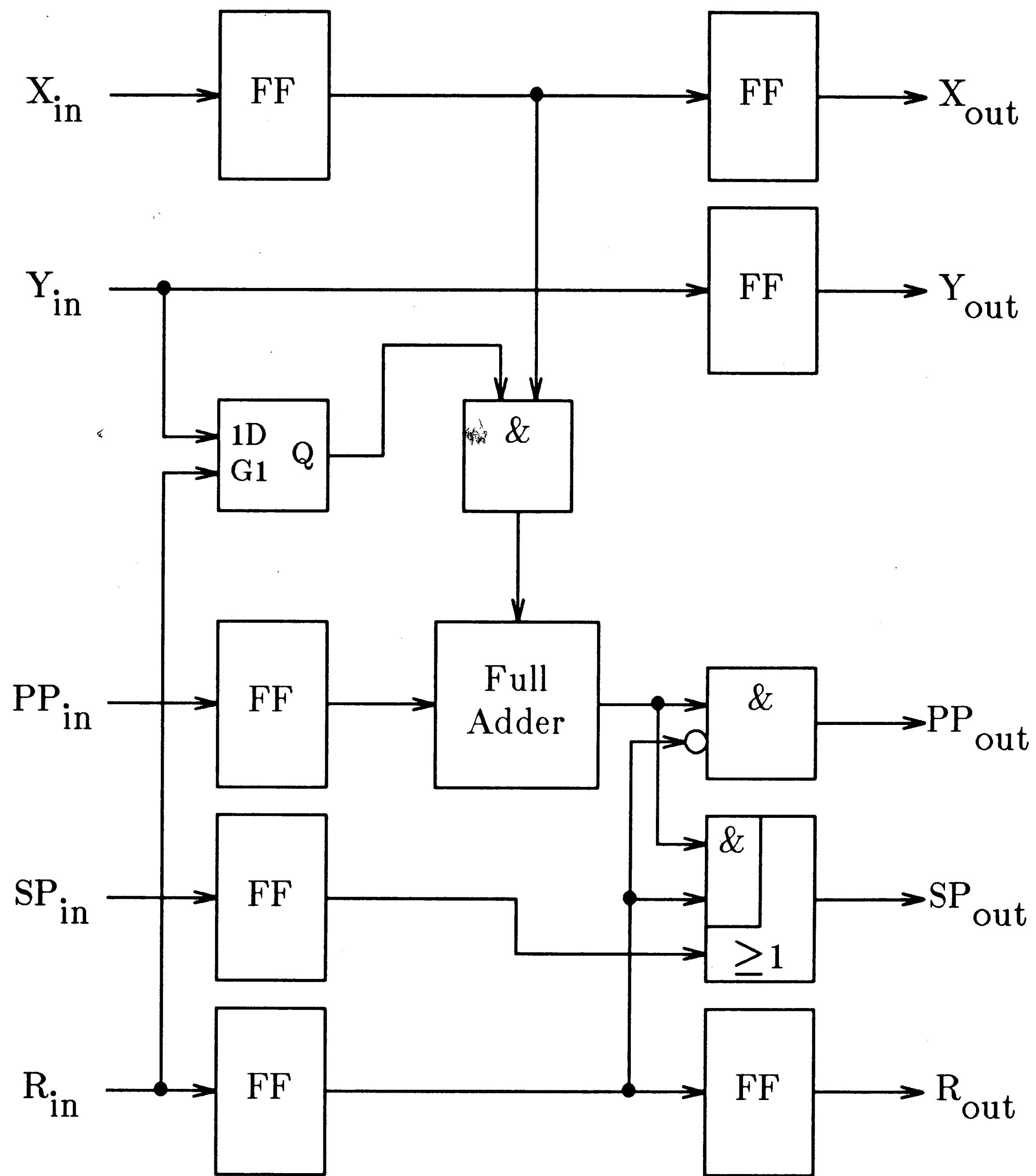


Figure 2.4. High Performance Bit-Sequential Multiplier Section

This multiplier, which differs from that of Jackson, et al, only in the addition of one D flip-flop and one 3-input complex gate per bit, produces the $2n$ -bit product of two n -bit twos complement numbers every $2n$ clock cycles. The Y input corresponds to the coefficient input in the previous multiplier, which has been serialized. The PP signals are the partial products corresponding exactly to the PP signals in the previous multiplier. The SP signals are the least significant n bits of the partial product, which were lost in the previous multiplier. Having the complete $2n$ bit product will be important in the next chapter on floating point multiplication.

Chapter 3

Floating Point Bit-sequential Multiplier

3.1 Introduction

There are cases where the dynamic range and precision of fixed-point representation are inadequate. Examples are large matrix operations, large FFT's, etc. Also, many algorithms are initially designed and simulated in software, using floating point arithmetic, on large general-purpose computers. However, parallel floating point arithmetic units are large and very expensive. Putting together many of them would be prohibitively costly in area, power and dollars. For these reasons, a floating point bit-sequential multiplier, and a floating point bit-sequential adder have been designed. This chapter describes the multiplier, and the next chapter describes the adder.

The floating point number representation chosen for this multiplier is in VAX F format. The mantissa is a 24 bit binary string. The exponent is an 8 bit binary string in two's-complement format, biased by +128. The sign is a single bit which is asserted for negative values. The value of the number is $(-S) \times (2^{E-128}) \times (M \times 2^{-23})$. For example, -32 would be represented by $S=1, E=10000101, M=100000000000000000000000$. 1 would be represented by

24 clock cycles, the corresponding exponent and sign are received on the exponent wires (MCDEXP and MPYEXP). The eight bits of the exponent are entered least significant bit first, followed by the sign bit, followed by 15 bits which are undefined and ignored. There is also a reset input, which must be entered coincident with the lsb's of the mantissa and the exponent. A block diagram of the multiplier can be found in Figure 3.1.

3.3 Detailed Implementation

The multiplier is broken into two parts: a mantissa multiplier, and an exponent/reformatter. The mantissa multiplier is a completely systolic implementation, as described in the previous chapter. The delay from the appearance of the lsb of the input to the lsb of the output is 25 clock cycles. A full 48 bit result can be produced every 24 clock cycles. The mantissa multiplier is fully pipelined, so that a multiplication can be performed every 24 clock cycles.

The exponent/reformatter (EXPFMT) section takes the full-precision mantissa product, and the two exponent and sign inputs, and formats the outputs. Within the EXPFMT, there are two basic operations: finding the sign and exponent of the result, and rounding and adjusting the output mantissa and exponent. The exponent of the result is found with a simple bit-sequential adder. However, since each of the two operands exponents are biased by plus 128, one of the biases must be subtracted before the result exponent (XP) is

correct. The sign is found by just XORing together the signs of the two input operands to obtain the sign of the result (NEG). Once the result exponent is found, it is necessary to detect certain conditions.

The first condition is overflow. If the resulting unbiased exponent is greater than +127, then it is outside the domain of the representation, and a flag is set to indicate overflow (OVF). Since the multiplier pipelines three operations, the flag is pipelined. A related condition is maximum exponent (MXP). If the unbiased exponent is exactly +127, then an overflow results if either the mantissa product is $\geq 2^1$, or rounding the mantissa forces a carry out of the most significant bit. This flag is also pipelined.

The next condition is underflow. If the resulting unbiased exponent is less than -128, then it is outside the domain of the representation in the other direction, and the underflow flag (UNF) is set. However, an underflow only occurs if the result is finite and too small to be represented in the dynamic range of the representation, but not if the result is exactly zero. If either of the two input exponents is exactly 0 (biased representation), then that input is 0, and the result is exactly zero, and not an underflow. Therefore, if that case is detected the zero flag (ZERO) is set, and the UNF flag is not set. A related condition is minimum exponent minus one (MNP). Notice that although the representable range of unbiased exponents is [-127,+127], that UNF is not asserted unless the resulting unbiased exponent is less than -128. If the

resulting unbiased exponent equals -128, then an underflow might or might not occur, depending on the mantissa product. If the mantissa product overflows either before or after rounding, then the exponent will be incremented by 1 to -127, and no underflow will occur. If, however, no mantissa overflow occurs, then an underflow will be signaled.

In summary, the result of the first stage of the EXPFMT is the sign bit (NEG), the eight bit biased exponent (XP), and five flags: overflow (OVF), maximum exponent (MXP), underflow (UNF), minimum exponent minus one (MNP), and zero (ZERO). Since the most significant bit of the mantissa product does not occur for 72 clock cycles, or three words, each of these fourteen bits is pipelined three times, 24 clock cycles apart.

The last stage of the EXPFMT, and of the bit-sequential floating point multiplier, is rounding the product of the mantissas, and formatting the result. Recall that the mantissa format is:

$$2^0 . 2^{-1} \dots 2^{-23}$$

and that the mantissa is a positive magnitude. Also, except for zero, the mantissa must be normalized, i.e. the mantissa must lie in the range $2^0 \leq \text{mantissa} \leq 2^1 - 2^{-23}$. Multiplying two 24 bit numbers results in a 48 bit number, whose format is:

$$2^1 \ 2^0 \ . \ 2^{-1} \ \dots \ 2^{-46}$$

and the range of the result is $2^0 \leq \text{mantissa product} \leq 2^2 - 2^{-21} + 2^{-46}$. If the most significant bit of the result is set, then the mantissa overflows, the mantissa is right shifted one bit, and the exponent is incremented by 1. If the MXP flag has been set for this operation, then OVF is asserted, and the mantissa and exponent are set to all ones. The sign of the result is unaffected.

The mantissa must be rounded to a 24 bit result. Since it is impossible to know which will be the least significant bit until the most significant bit is output by the mantissa multiplier, the 24 most significant bits of the result, a guard bit (GRD), and two different rounding bits are saved: round if the most significant product bit is asserted (RNDMSPT), and round if the most significant product bit is negated (RNDMSPC). The fair rounding scheme requires always rounding to an even result if the full-precision result is exactly halfway between two results, i.e., the round bit is set, and all lower order bits are zero. In that case, only round (increment the mantissa), if the mantissa is odd. In the case that the mantissa product is $\geq 2^1$, then the least significant bit of the product will be 2^{-22} , and the result will be rounded up only if the 2^{-23} bit is set, and if either the least significant bit is set or any bit at all to the right of the 2^{-23} bit is set. In other words, only if the round bit is set, and the result is odd, so round to even in any case or the round bit is set and the bits to be

truncated are $>1/2$ the least significant bit. Similarly, in the case that the mantissa product is $<2^1$, then the least significant bit of the product will be 2^{-23} , and the result will be rounded up only if the 2^{-24} bit is set, and if either the least significant bit is set, or if any bit at all to the right of the 2^{-24} bit is set.

It is important to know whether rounding the mantissa will overflow the mantissa, because that forces the exponent to be incremented, and may cause an overflow if true, or an underflow if false. If it were necessary to wait to detect this condition until after the complete result were obtained, then the multiplier would have to add another pipeline stage. Fortunately, it is possible to detect an overflow from rounding even before the rounding is done. It is only necessary to detect that the 24 bits of the mantissa, assuming that the mantissa product will not have the most significant bit asserted, and the round bit are all equal to one. This is the only case which can force an overflow from rounding. This case is detected, and if true, forces the mantissa to be right shifted and the exponent to be incremented by one, with the possible consequences previously described.

It then remains only to perform the actual rounding with a simple bit-sequential half adder, and to increment the exponent with another bit-sequential half adder. If an overflow occurs, the mantissa and exponent are set to all ones, without affecting the sign bit. If an underflow occurs, or the result

is exactly zero, the mantissa, exponent and sign are all set to zero. If the result does not exactly match the full-precision result, either because of truncating the full-precision mantissa product, or because of an overflow or underflow, an inexact flag (INX) is asserted. A new result appears on two wires (PRDMAN and PRDEXP) every 24 clock cycles or less frequently. There is an output flag (RST74), which indicates the least significant bit of a new result, and negative (NEG), zero (ZRO), overflow (OVF), underflow (UNF) and inexact (INX) flags which are also output.

3.4 Conclusion

The bit-sequential floating point multiplier is designed with 1.25μ CMOS standard cells from the AT&T library.^[11] That library has two sets of cells: an area optimized set which uses 5μ transistors, and a performance optimized set which uses 20μ transistors. This design used the area optimized set. It uses 381 flip-flops and 556 gates, or 9,024 transistors. The deepest path between two flip-flops is 5 gates deep. Its maximum clock rate at 5.0V, 25 °C and nominal processing is 21.8MHz. With a latency of 24 clock cycles, that is equivalent to 0.9 MFLOP. The delay from the least significant bit of the input to the least significant bit of the output is 74 clock cycles. Since the multiplier only has 14 I/O's, it could be packaged in a 16-pin plastic dip; that is a very cheap MFLOP. The complete schematics of the multiplier are included as Appendix A, Figures A.1-A.16

Chapter 4

Floating Point Bit-sequential Adder

4.1 Introduction

The floating point bit-serial adder uses the same format as the multiplier. It has two additional inputs: a subtract augend input (SUBTRG), and a subtract addend input (SUBTRD). When these signals are asserted, their respective operands are negated by inverting their sign bits. The adder can therefore implement $a+b$, $a-b$, $-a+b$ and $-a-b$. The adder does, however, have a slightly longer delay than the multiplier: 76 clock cycles instead of 74. The adder is also not purely systolic. The exponent difference, mantissa denormalization, arithmetic and renormalization are bit-serial, but the exponent adjustment is done by a parallel counter, as the mantissa result is being generated. For this reason, the floating point adder is the speed limiting component in a bit-sequential system, rather than the multiplier.

4.2 Addition Strategy

The adder is not as easily broken into mantissa and exponent parts as the multiplier. The algorithm for floating point addition is somewhat more convoluted than for multiplication:^[10]

1. find the difference in the exponents, select the larger
2. denormalize the smaller exponent mantissa (right shift)
3. add or subtract the mantissas
4. mantissa normalization and rounding, adjust the exponent
5. detect overflow or underflow

4.3 Detailed Implementation

The first step is to find the difference in the exponents, and to select the larger number. The difference is found with one's complement arithmetic, implemented with a bit-serial adder, a shift register, and a bit serial half adder and XOR. This implementation augments the algorithm by also finding the larger number even when the exponents are equal. The smaller number, or the AUGEND input if the numbers are exactly equal, is the one chosen to be denormalized. This means that if a subtraction is performed, the result is guaranteed to be non-negative. This will be important later. If the unbiased result exponent (EXP) is +127, then the maximum exponent flag (MXP) is asserted. If the rounded sum of the mantissas overflows, then the exponent will be incremented by 1, and an overflow will occur.

The signs of the operands, after negating if the respective subtract inputs are asserted, are XORed together. If they are different, the mantissa of the smaller number is subtracted from the mantissa of the larger number after the smaller number is denormalized. The sign of the result (NEG) is the sign of the larger number.

The next step is to denormalize the mantissa of the smaller number. This is easily accomplished in the bit-serial format by loading the mantissa into a parallel-in-serial-out register which also has a clock enable. The exponent difference is loaded into a down counter (DNRMCTR), and counts down. As long as the count is ≥ 0 , the shift register does a logical right shift. The serial output of the register goes into a three bit register, which consists of the guard (GRD), round (RND) and sticky (STK) bits. The sticky bit is formed by ORing together all the bits which are shifted into it. The purpose of the GRD bit is to become the least significant bit if the mantissa is left shifted when the result is normalized, and to be the round bit otherwise. The purpose of the RND bit is to round the result if the mantissa is left shifted. The purpose of the STK bit is more subtle. No matter how many bits are shifted past the RND bit, it is only necessary to OR them together into the STK bit to get the correct result. If the denormalized number is subtracted, then all the bits, including the STK bit, are inverted, and one is added to the STK bit. The resulting STK bit will be the OR of the result even if the subtraction were performed with full-precision. The resulting STK bit is also necessary to decide if the portion of the mantissa to be truncated is exactly equal to one half the least significant bit. However, since the adder is pipelined to accept a new operand every 24 clock cycles, only shifts of 0 to 23 places can be made bit-serially. It is necessary to add a small parallel shifter which shifts the least significant bit of the denormalized

operand, the guard bit the round bit and the sticky bit, between the output of the denormalizing shift register and the next register in the pipeline.

The third step is to perform the actual addition or subtraction. There are only 24 clock cycles, but the denormalized mantissa is 27 bits long, including the guard, round and sticky bits. Again, it is necessary to include a small parallel adder to obtain the result for the least significant result bit, and the guard, round and sticky results. The result of the summation is stored in a shift register. It is possible for the result of the summation to overflow one bit, or for there to be any number of leading zeroes from 0 to 24. In a parallel floating point adder, the number of leading zeroes in the result is encoded, the mantissa is renormalized and the exponent is decremented by that amount. In the bit-serial case, it is possible to do these operations on-line, as the result is being calculated.

As the result appears, one bit at a time, an exponent counter (EXPCTR) decrements the exponent every time a zero appears in the result, and reloads the original exponent every time a one appears, or if there is an overflow out of the most significant bit. This is why it was necessary to select the numbers so that the result could be guaranteed to be non-negative, else this simple scheme wouldn't work. At the same time a renormalization counter (RNRMCTR), counts the leading zeroes. Every time a one is encountered, or if there is an overflow out of the most significant bit, RNRMCTR is cleared to zero. At the

same time, it is necessary to know if the result will overflow after rounding, so that the exponent can be adjusted, and any overflow or underflow can be detected. This condition is harder to detect in the adder than in the multiplier. The mantissa in the multiplier could only overflow, not underflow. However, it is possible to detect those results which will overflow after rounding, even before performing the rounding.

If the two input exponents differed by more than one, then the largest possible left shift to renormalize the result is one. If, however, the two input exponents were equal, or differed only by one, then the largest possible left shift is 24. In this case, only the GRD bit will be set; the RND and STK bits must be zero, because the denormalization shift was ≤ 1 . Therefore, the mantissa can only overflow after rounding if the 24 most significant bits are all one, and the GRD bit is one, or if the most significant bit is zero, the next 23 bits are one, and the GRD and RND bits are one. It is impossible for the two leading bits to be zero if both the RND and STK bits are set. Therefore, there are no more than these two ways for the 25 consecutive most significant bits of the normalized result to be all ones.

Again, there is the problem that there are only 24 clock cycles to renormalize and output the result, but there are 26 different possible realignments of the result: right shift one (mantissa overflow), or left shift 0 to 24 places. This problem was solved by combining a shift register with three pointers. If the

mantissa overflowed, or if the most significant bit would overflow after rounding, then a shift right must be performed, and a pointer (SHR) is set which points to the bit to the left of the lsb. This bit becomes the least significant bit of the output of the result register. If the mantissa didn't overflow, and the most significant bit is asserted, and the 24 most significant bits and the GRD bit are *not* all ones, then no shifts are performed, and a pointer (SH0) points to the least significant bit of the result, which becomes the least significant bit of the output. If neither of these pointers is asserted, then the least significant bit of the output is taken from the bit to the right of the least significant bit of the result. At the same time that the pointers are formed, the round flag (RND) is calculated based on which flag is asserted.

It now remains to renormalize the result, round it, and output the result. This is all done simultaneously, only 2 clock cycles delayed from producing the most significant bit of the result. As in the multiplier, if overflow or underflow occurs, the output is reformatted. The adder has a sum mantissa output (SUMMAN), sum exponent and sign output (SUMEXP), negative flag (NEG), zero flag (ZRO), overflow flag (OVF), underflow flag (UNF), and inexact flag (INX). It also outputs a signal indicating the least significant bit of the output (RST76).

4.4 Conclusion

The adder is also designed with area optimized standard cells from the AT&T library. It uses 432 flip-flops and 965 gates, or 12,192 transistors. The deepest path between two flip-flops is 10 gates deep. Its maximum clock rate at 5.0V, 25 ° C and nominal processing is 8.0MHz. With a latency of 24 clock cycles per operation, the resulting throughput is 0.33 MFLOPS. The delay from the least significant bit of the inputs to the least significant bit of the output is 76 clock cycles; that is two more than for the multiplier. The deepest path in the adder is twice as long as in the multiplier, and the adder frequency is less than half. That is reasonable, since the adder is not a purely systolic system like the multiplier. Further, no attempt was made to optimize the timing of the unit. It is reasonable to expect a large improvement with more work, maybe as large as a factor of two, simply by adding registers in the appropriate places, and buffering heavily loaded wires, especially the RST wires, which control the operation of the adder. Experience shows that often a very small number of long paths can slow down an otherwise much faster circuit. The complete schematics of the adder are included as Appendix B, Figures B.1-B.17.

Chapter 5

Applications for Bit-Sequential Floating-Point Arithmetic Units

5.1 Introduction

The arithmetic units described in the two previous chapters are most useful when applied to a heavily pipelined, functionally parallel architecture. This is because the large delays of the units (>3 words), and the large number of clock cycles per input (24), would make using them unreasonable otherwise. On the other hand, they are very cheap in absolute terms, so that if a very low performance, cost-sensitive application existed that required floating-point arithmetic and a higher performance than possible in a software implementation, they would be applicable. Applications such as those described occur frequently in digital signal processing. This chapter describe two particular uses for these units: in a generic matrix-vector multiplier, and in a parallel-pipelined FFT calculator.

5.2 Matrix-Vector Multiplication

In matrix-vector multiplication, an $n \times n$ matrix multiplies a length n input column vector to produce a length n output column vector. Each row of the

output vector is the inner-product of a row of the matrix with the input vector. Hence the total computation requires n^2 multiplications and $n(n-1)$ additions. By providing n multipliers and $(n-1)$ adders in a systolic array as shown in Figure 5.1, it is possible to do matrix-vector multiplication in $O(n)$ time.

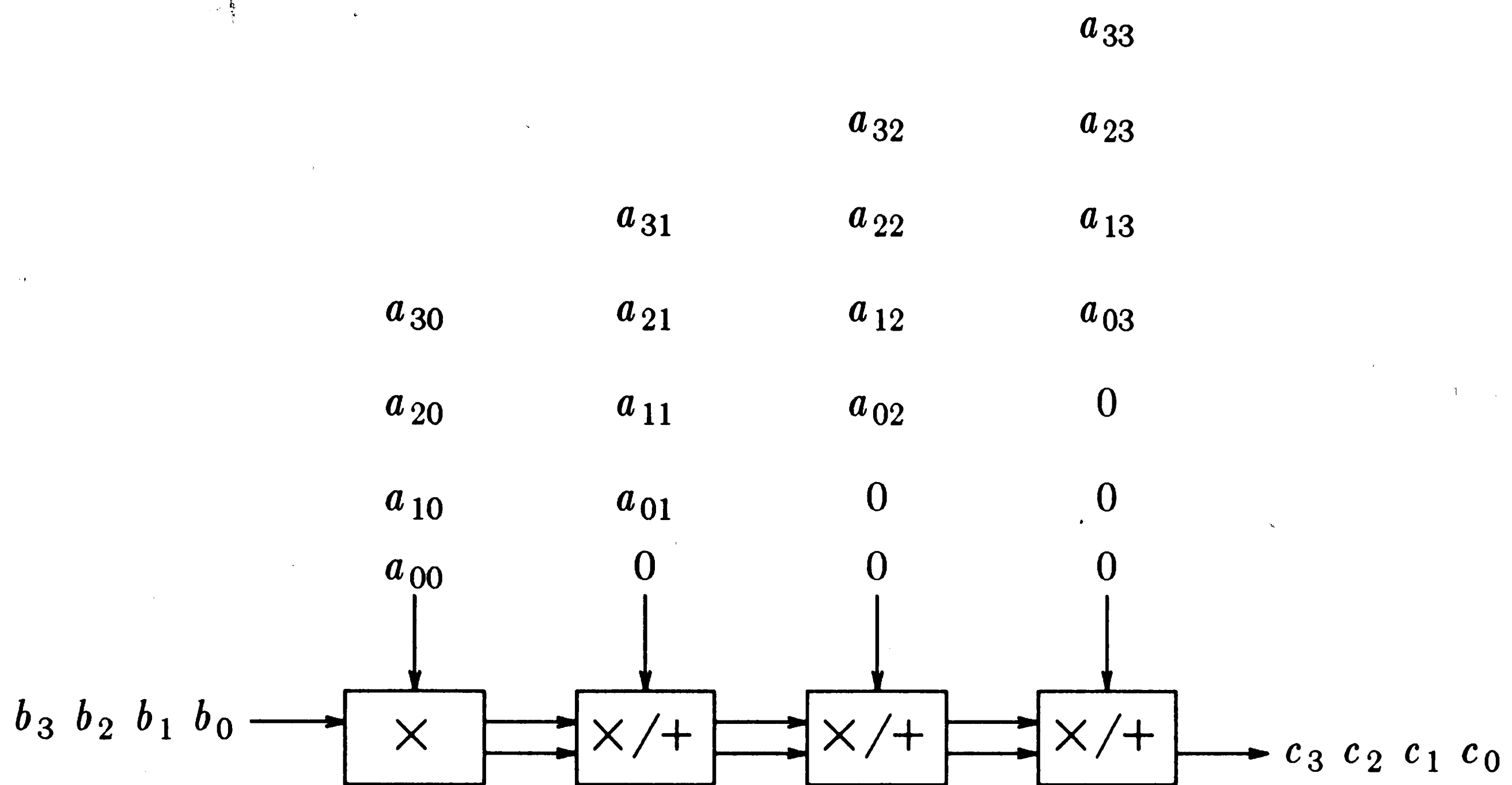


Figure 5.1. Matrix-Vector Multiplier

If the multiplier and adder modules described in the previous chapters are used in this application, then a new data point can start every 24 clock cycles. Therefore the latency is $24P$, where P is the clock period. The throughput is $1/24P$. The multiplications occur in parallel, so that only the multiplication in the first cell adds to the delay. Since the multiplier delay is 74 clock cycles, and the adder delay is 76 clock cycles, the total delay is $(74+76(n-1))P$. The floating point bit-sequential multiplier and adder described in the previous

chapters are particularly well-suited to such an application. They have very limited pincounts, compared with a parallel implementation, so that many more units can be included on one chip without being limited by packages. Also, since they are so small, it is easy to imagine putting redundant units on a chip, with switches to bypass a faulty unit.

5.3 FFT Calculation

The Fourier transform is an example of matrix-vector multiplication. However, Cooley-Tukey and others have shown how to do this special case in $O(n \log n)$ time. One could use four multipliers and six adders developed in chapters 3 and 4 to form a butterfly processor (see Figure 5.2). $(n/2) \log n$ of these butterfly processors can be combined to form a parallel transform processor (see Figure 5.3). Again, the latency of the FFT calculator would be $24P$, and the throughput would be $1/24P$. In this case, however, the delay of the array is $(74 + 2 \times 76) \log n$. Since there are 3 words input to the butterfly and 2 words output from the butterfly, there are only 10 data wires, 1 clock wire, 1 reset in wire, 1 reset out wire, and 109,248 transistors per butterfly. Again, it should be possible to package a 32-bit floating point butterfly processor in an inexpensive 16-pin package. A special memory chip can be designed which cycles through the appropriate coefficients every 24 clock cycles. Its pincount would also be extremely low, since all the coefficients are only two wires wide.

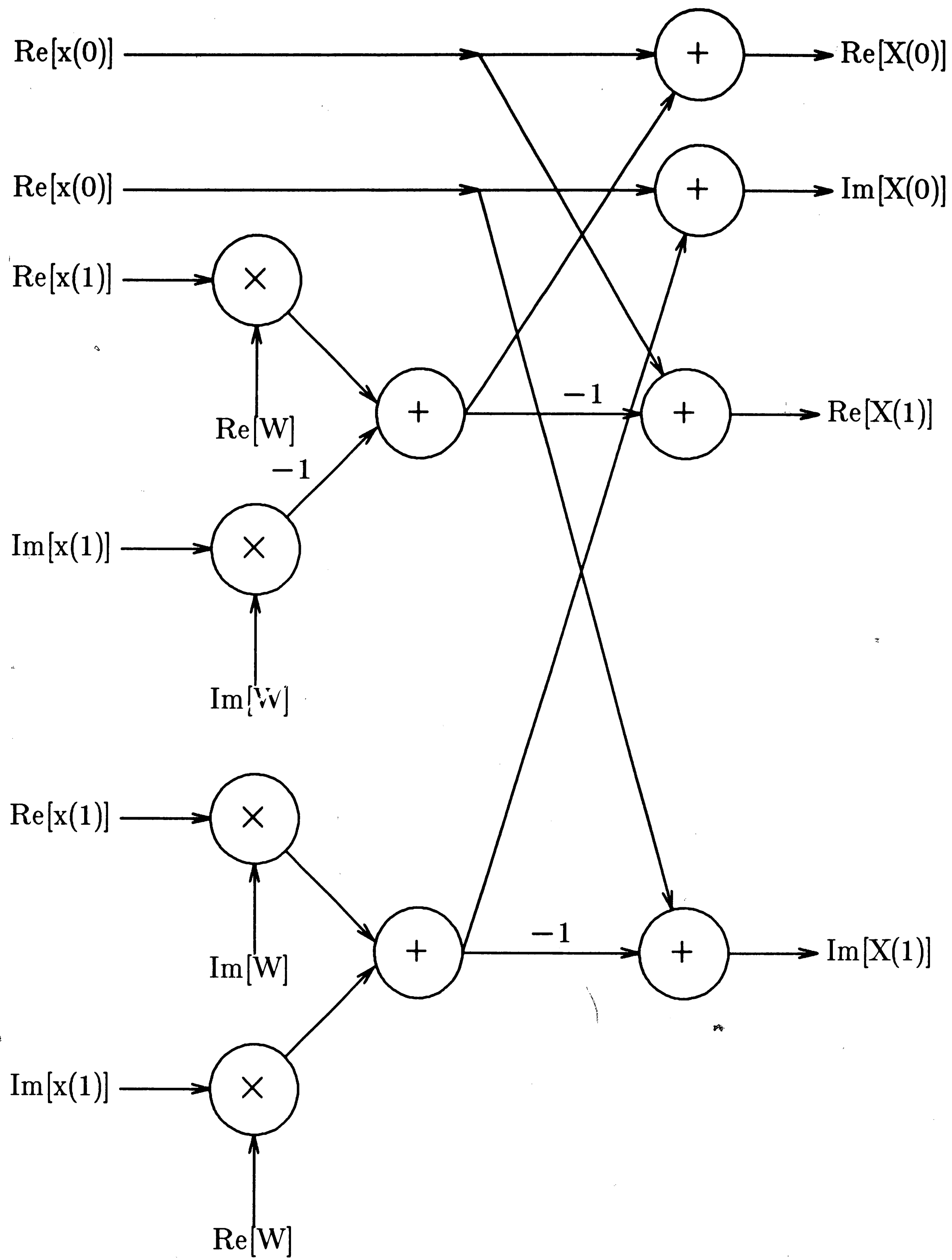


Figure 5.2. Butterfly Processor

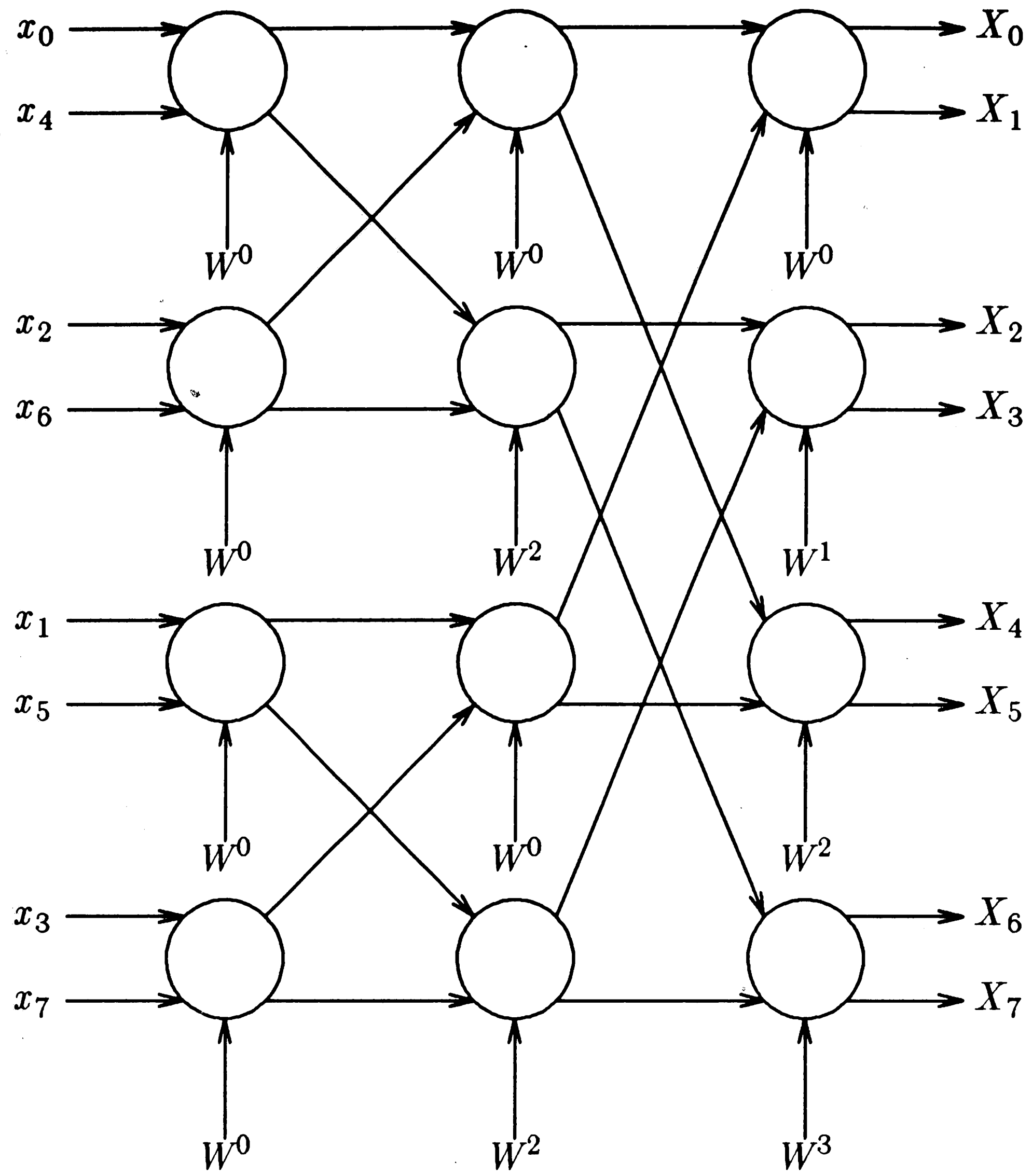


Figure 5.3. FFT Calculator

Chapter 6

Conclusion

6.1 Discussion

This thesis makes an argument for the utility of bit-sequential architectures in pipelined-parallel functions. Most of the work reported in the literature deals only with fixed point number representation. The major contribution of this thesis is the design of two completely sequential floating point processors for the first time. The design of a floating point bit-sequential multiplier reported here runs at 0.9 Mflops. The adder, the slower of the two, runs at least at 0.33 Mflops. An array of these units will have a throughput which is an integer multiple of 0.33 MFlops. One should note that this performance is not optimized, and can be improved with more careful implementation. The applicability of these processors is demonstrated by single-chip implementations of a matrix-vector multiplier, and an FFT butterfly operator.

6.2 Future Directions

The floating point bit-sequential multiplier and adder described in the preceding chapters are useful by themselves. A number of directions suggest themselves for expanding on what has been described here. One area of study

would be to implement these units, optimizing their area-throughput, to see what results might be obtained compared with the estimates made here. Another area of study would be to make a more complete set of operators, including divide, square root, fixed \leftrightarrow floating convert, etc. Yet another area of study would be to improve the units described here by making them completely IEEE754-1985 compatible, including denormalized numbers, NaN's and infinity. Of course, there is the whole subject of various applications, particularly those which take advantage of the small size of each individual unit to incorporate fault tolerance into an array of operators.

REFERENCES

1. I-Ngo Chen and Robert Willoner. "An $O(n)$ Parallel Multiplier with Bit-Sequential Input and Output", *IEEE Trans on Computers*, vol. C-28, no. 10, Oct 1979.
2. R. Gnanasekaran. "A Fast Serial-Parallel Binary Multiplier", *IEEE Trans on Computers*, vol. C-34, no. 8, Aug 1985.
3. Tom Rhyne and Noel R. Strader, II. "A Signed Bit-Sequential Multiplier", *IEEE Trans on Computers*, vol. C-35, no. 10, Oct 1986.
4. Charles E. Leiserson. *Area-Efficient VLSI Computation*. Cambridge, MA: The MIT Press, 1983.
5. Neil Weste and Kanran Eshraghian. *Principles of CMOS VLSI Design: a Systems Perspective*. Reading, MA: Addison-Wesley Publishing Company, 1985.
6. Paul M. Chau, Kay C. Chew and Walter H. Ku. "A Bit-Serial Floating-Point Complex Multiplier-Accumulator For Fault-Tolerant Digital Signal Processing Arrays", in *1987 International Conference on Acoustics, Speech, and Signal Processing*, pp483-486. Piscataway, NJ: IEEE, 1987.
7. Leland B. Jackson, James F. Kaiser and Henry S. McDonald. "An Approach to the Implementation of Digital Filters", *IEEE Transactions on Audio and Electroacoustics*, vol. AU-16, no. 3, pp413-421, Sept 1968.
8. Joseph T. Scanlon and W. Kent Fuchs, "High Performance Bit-Serial Multiplication", *Proc IEEE International Conference on Computer Design: VLSI in Computers*, pp114-117. Piscataway, NY: IEEE, 1986.
9. ANSI/IEEE Std 754-1985: *An American National Standard IEEE standard for Binary Floating-Point Arithmetic*. New York: The Institute of Electrical and Electronics Engineers, Inc., 1985.
10. Paul M. Chau and Walter H. Ku. "A VLSI Floating Point Signal Processor". in Sun-Yuan Kung, Robert E. Owen and J. Greg Nash, eds. *VLSI Signal Processing, II*. New York: IEEE Press, 1986.
11. 1.25 μ CMOS Cell Library. AT&T, 1987.

Appendix A

Floating Point Bit-Sequential Multiplier Schematics

This appendix includes the complete schematics for the floating point bit-sequential multiplier described in chapter 3.

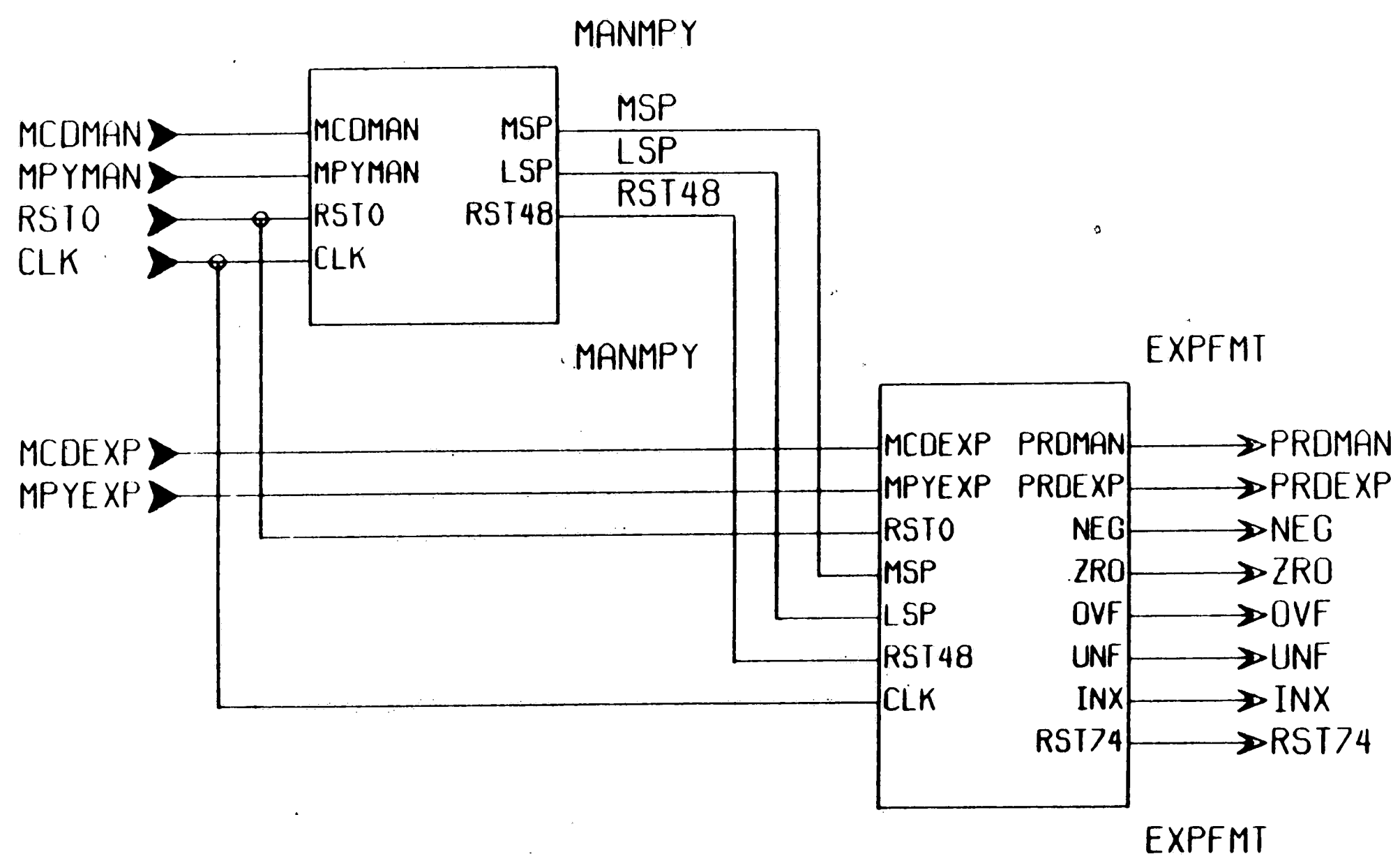
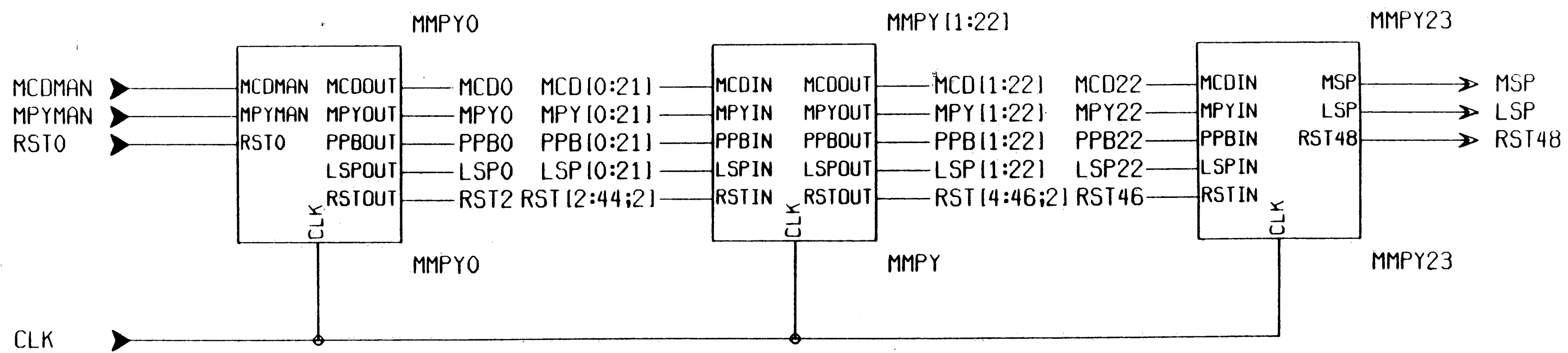


Figure A.1. FPMPY.1
39

Figure A.2. MANMPY.1
40



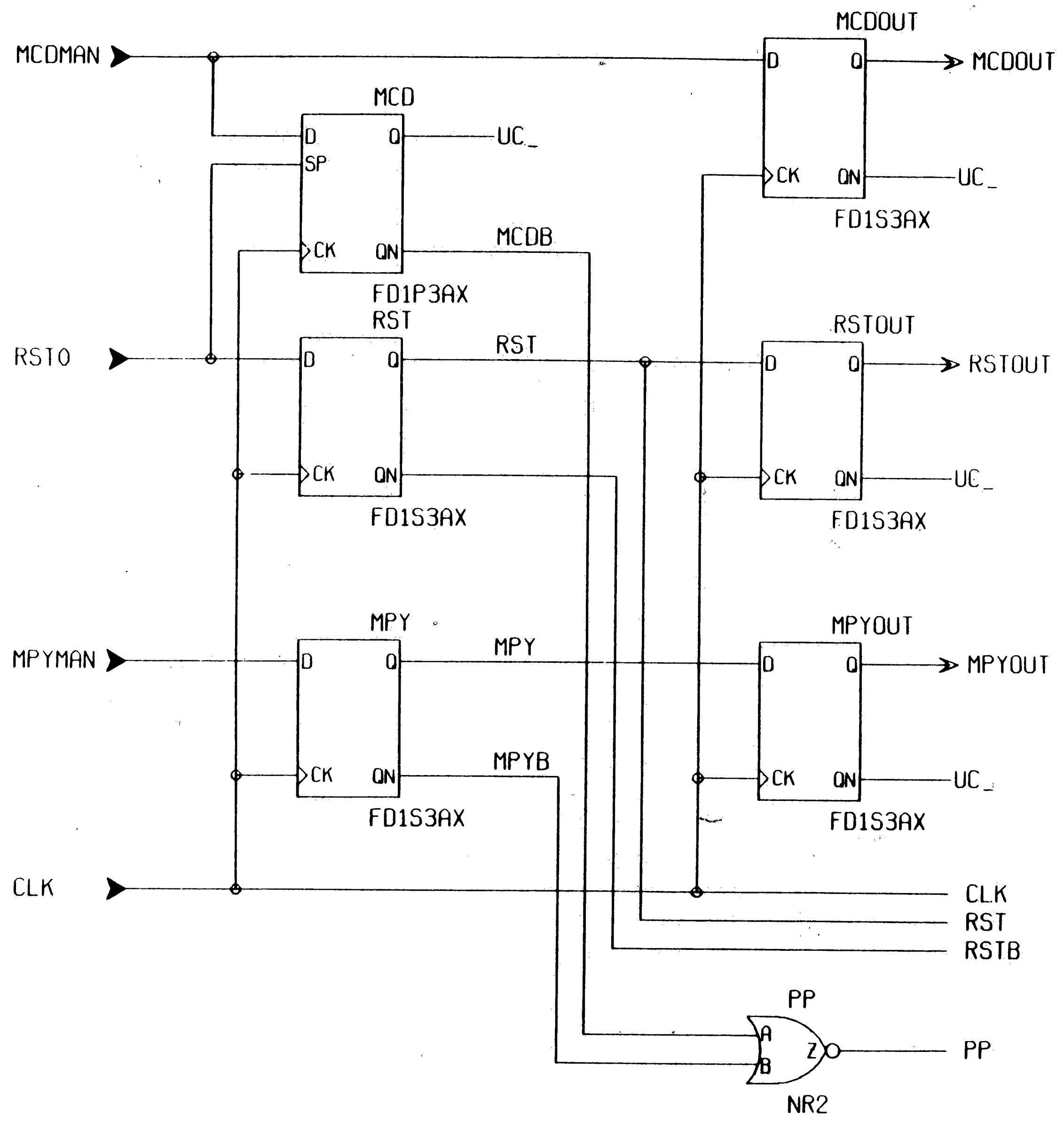


Figure A.3. MPY0.1
41

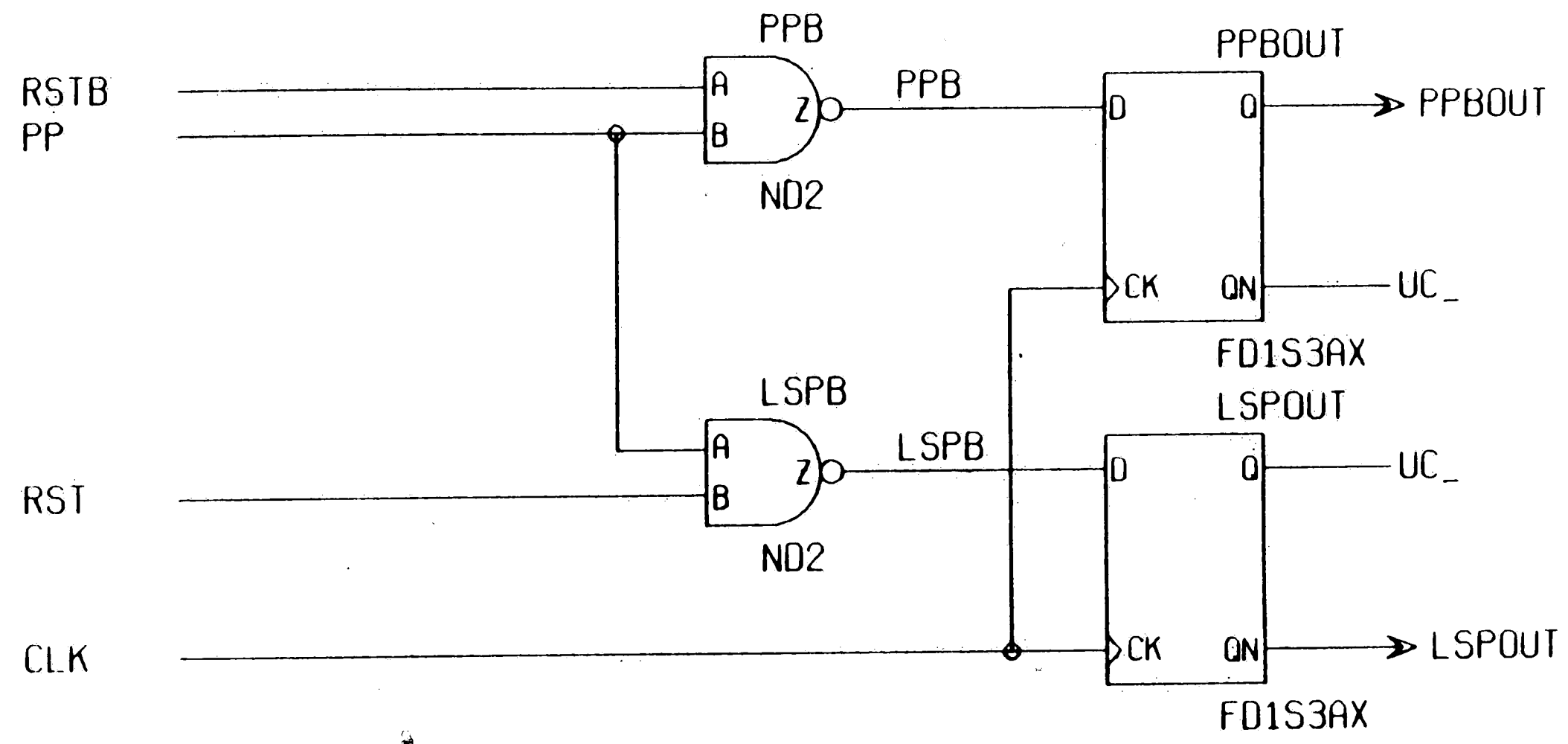


Figure A.4. MPY0.2
42

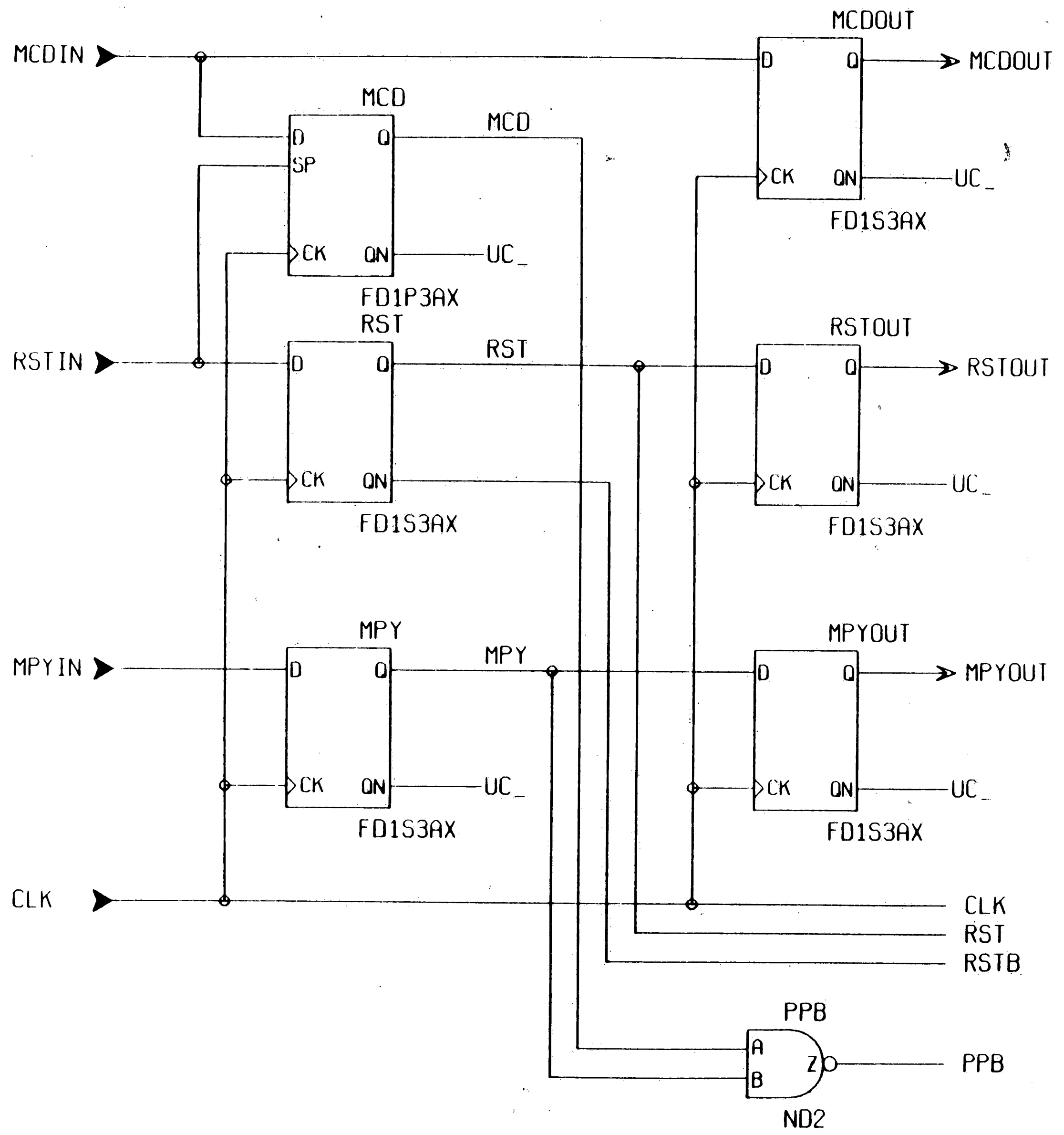


Figure A.5. MPY.1
43

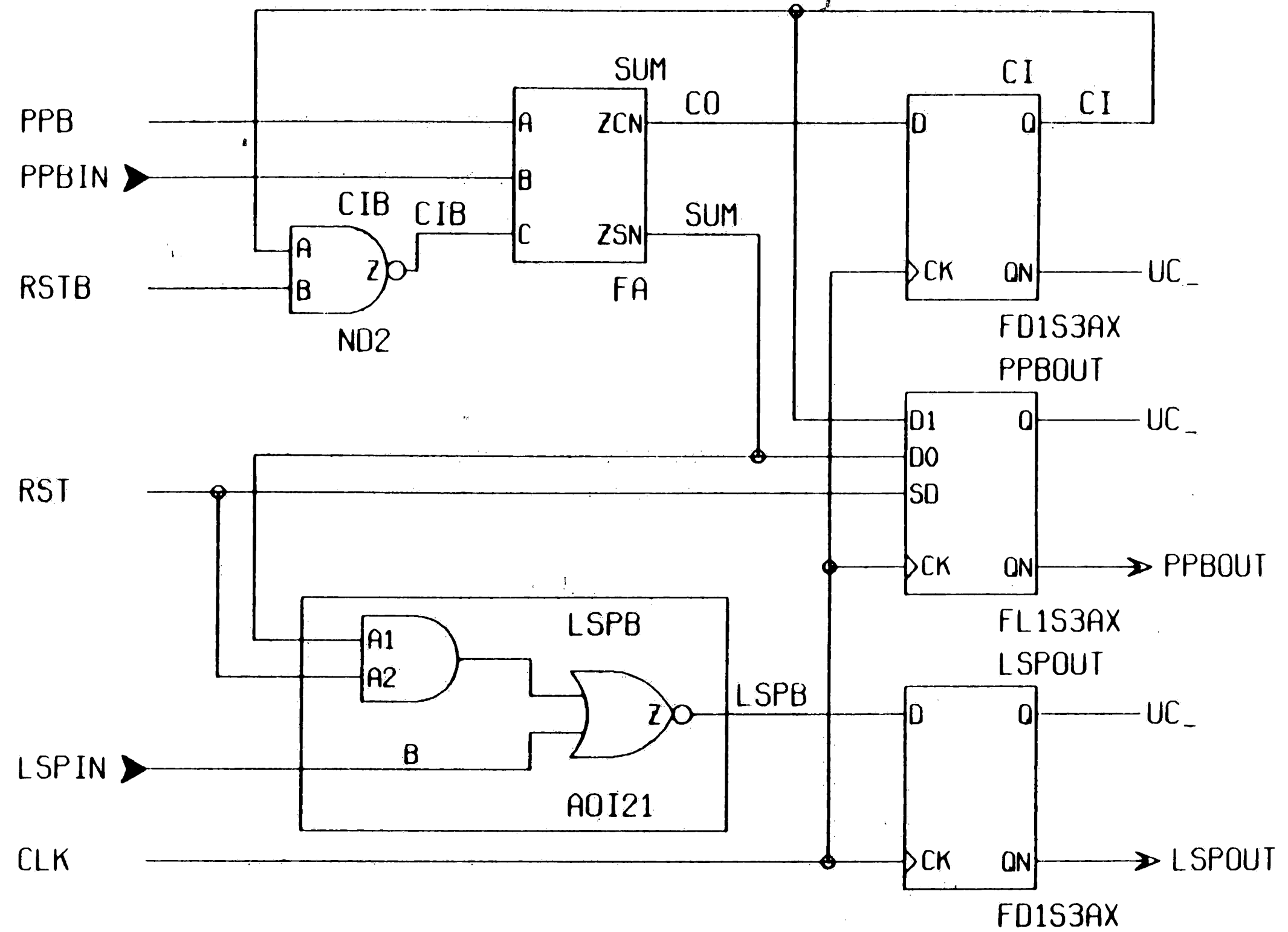


Figure A.6. MPY.2
44

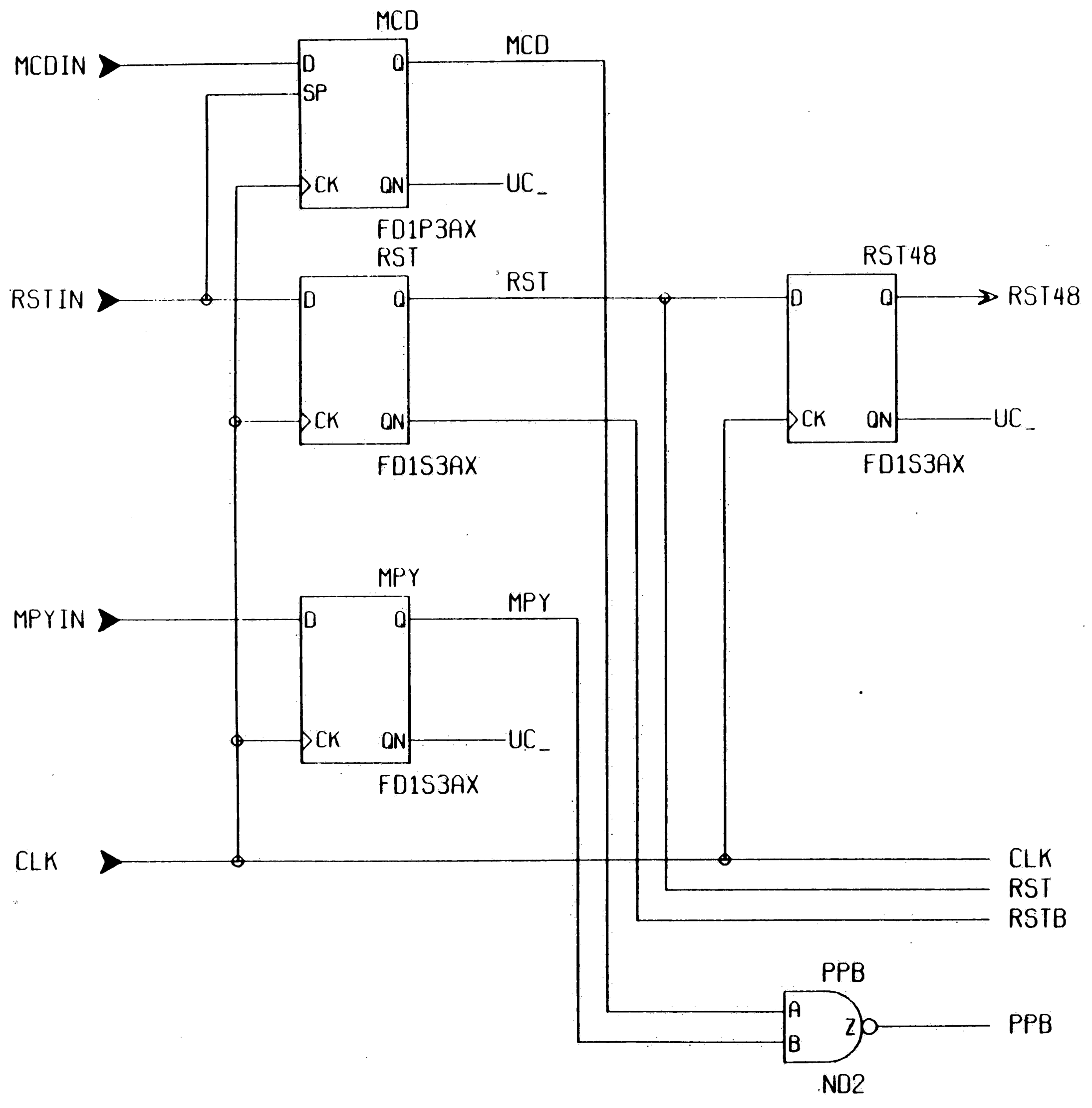


Figure A.7. MPY23.1
45

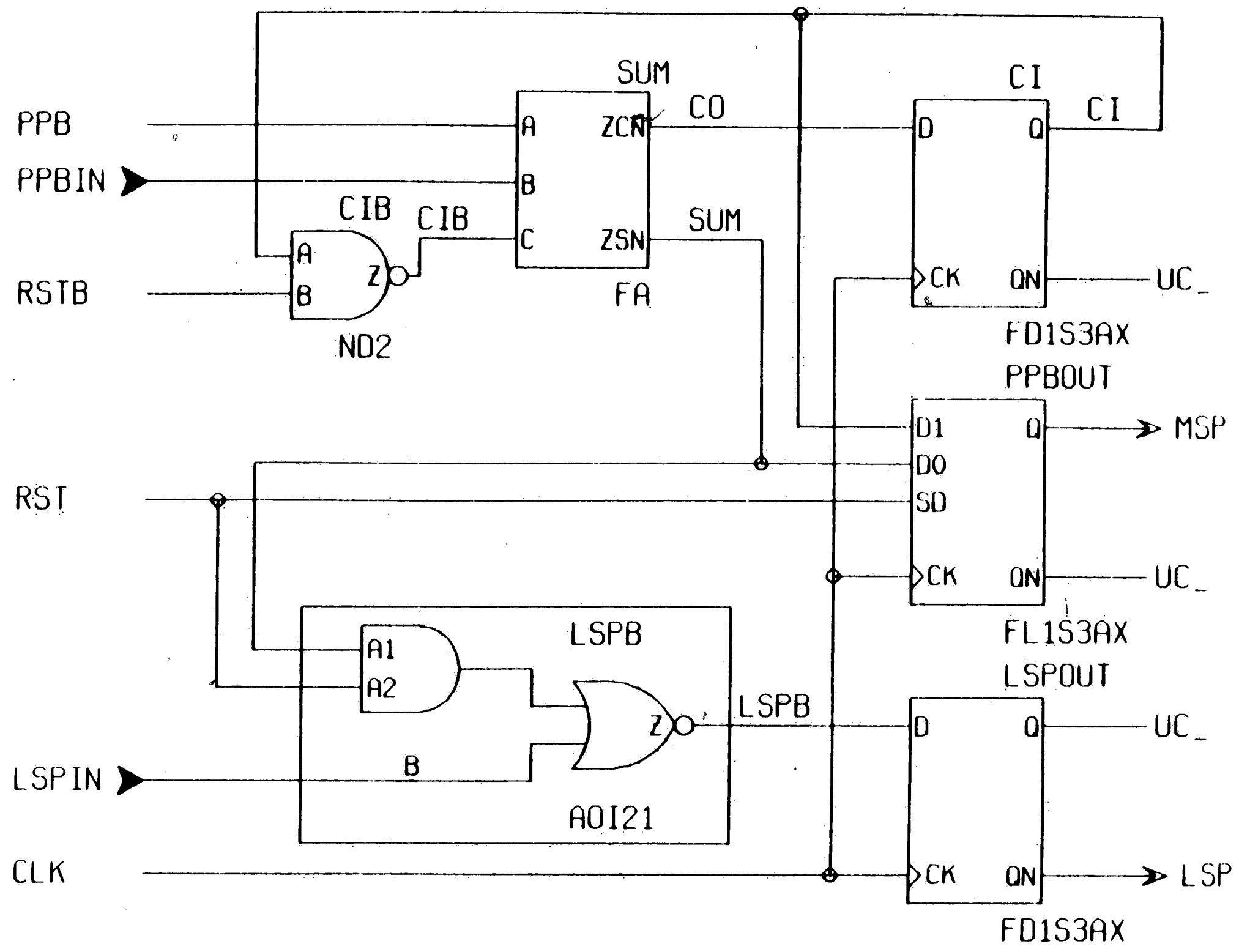


Figure A.8. MPY23.2

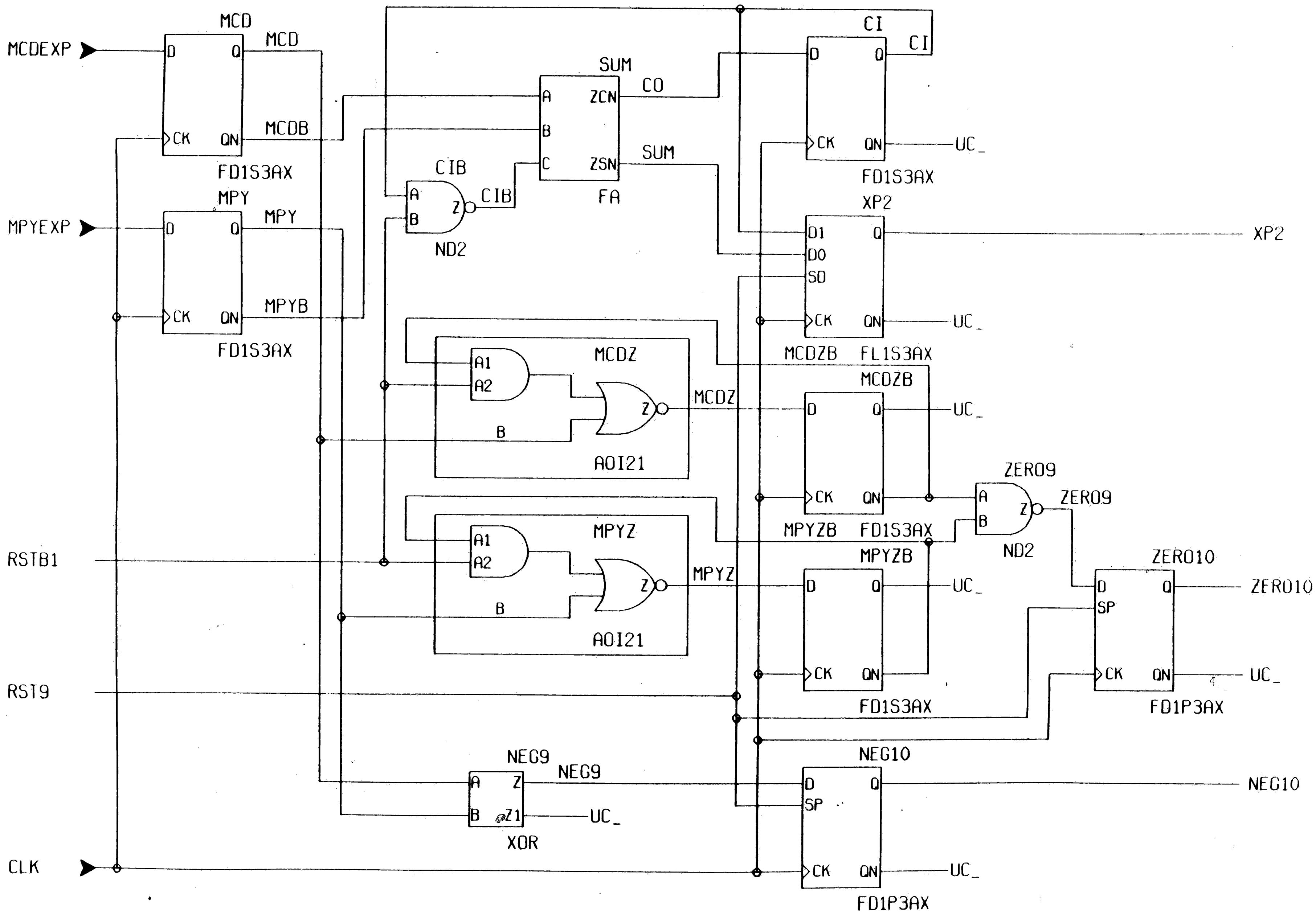


Figure A.9. EXPFMT.1
47

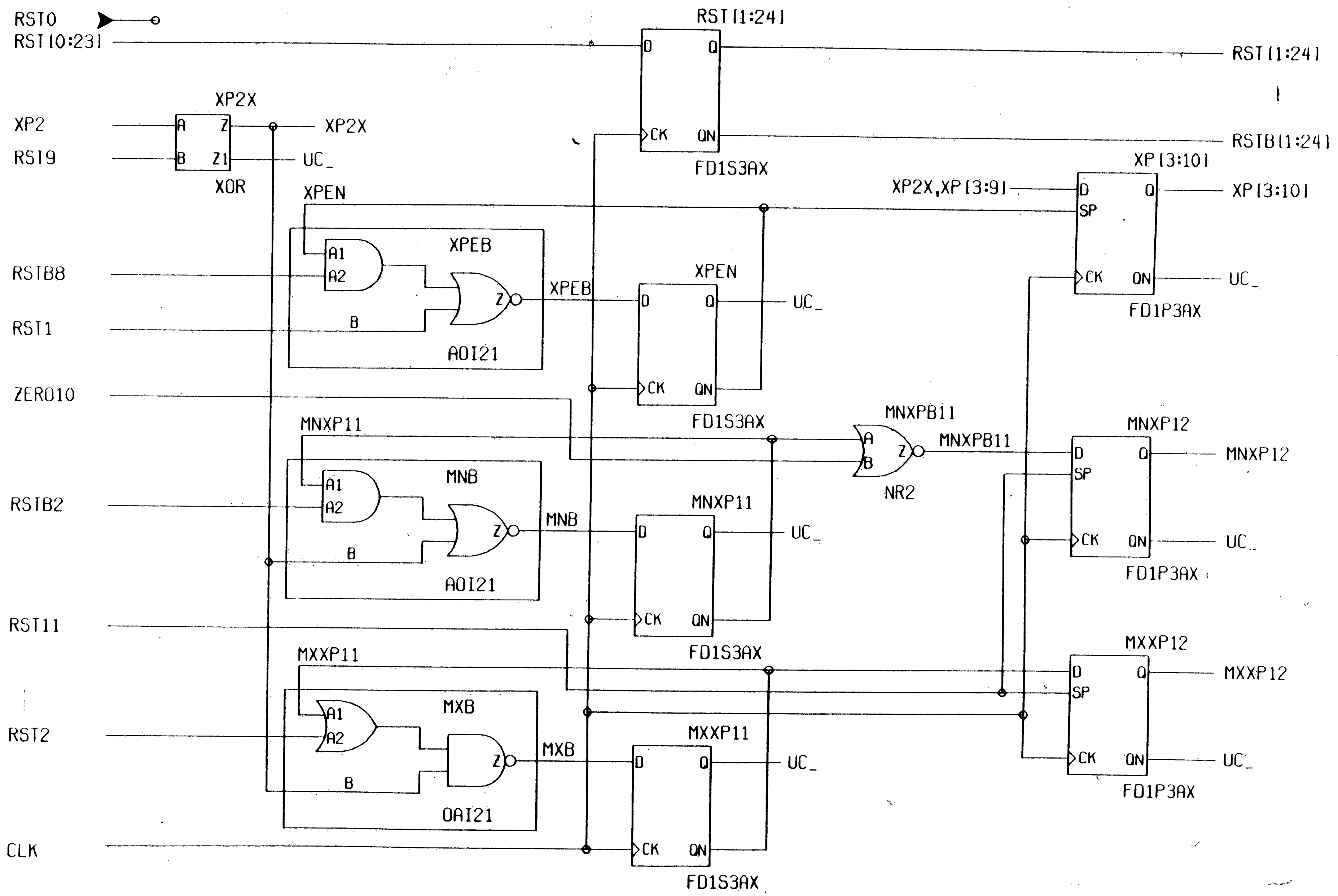


Figure A.10 EXPFMT.2
48

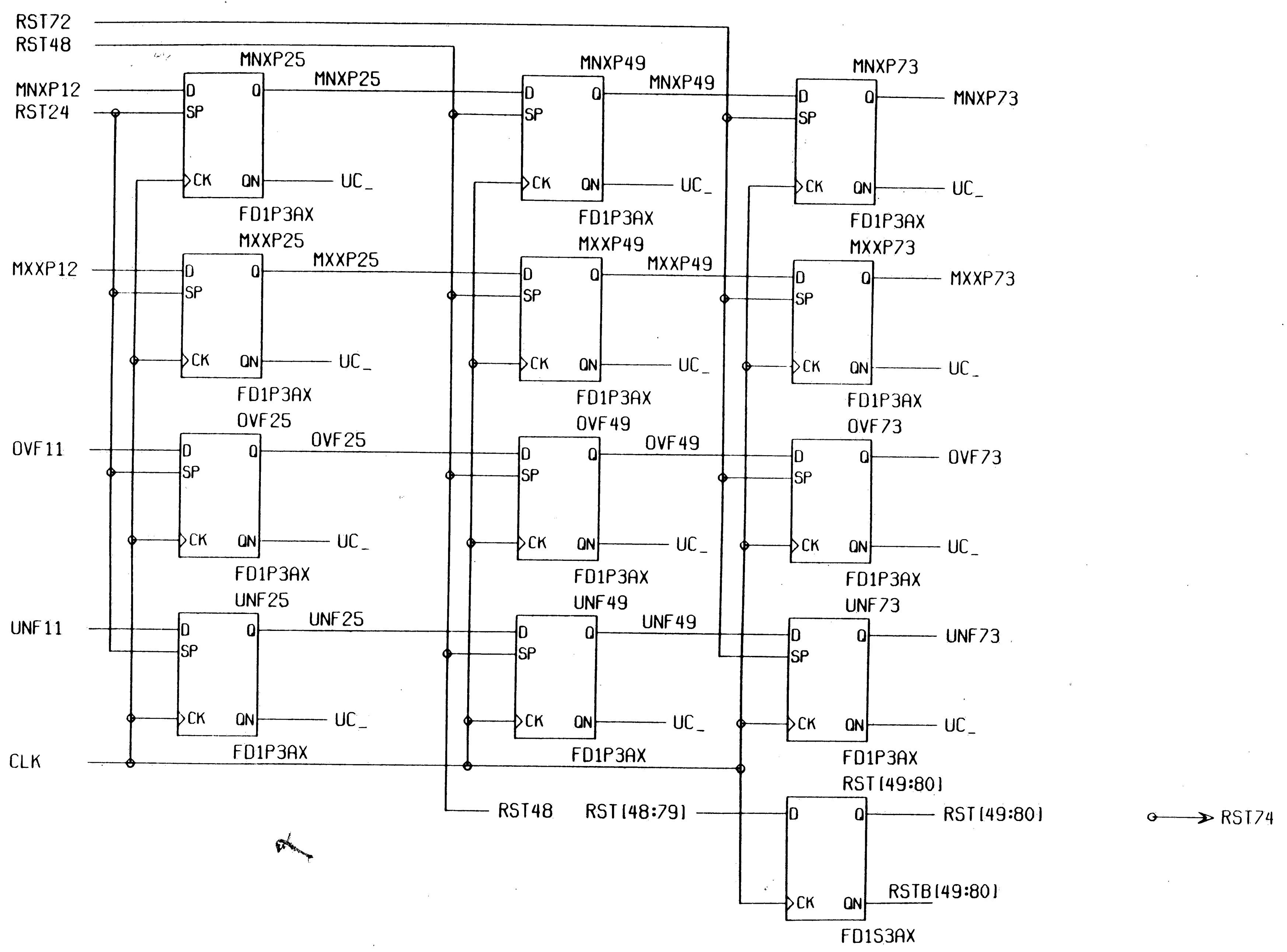


Figure A.12. EXPFMT.4
50

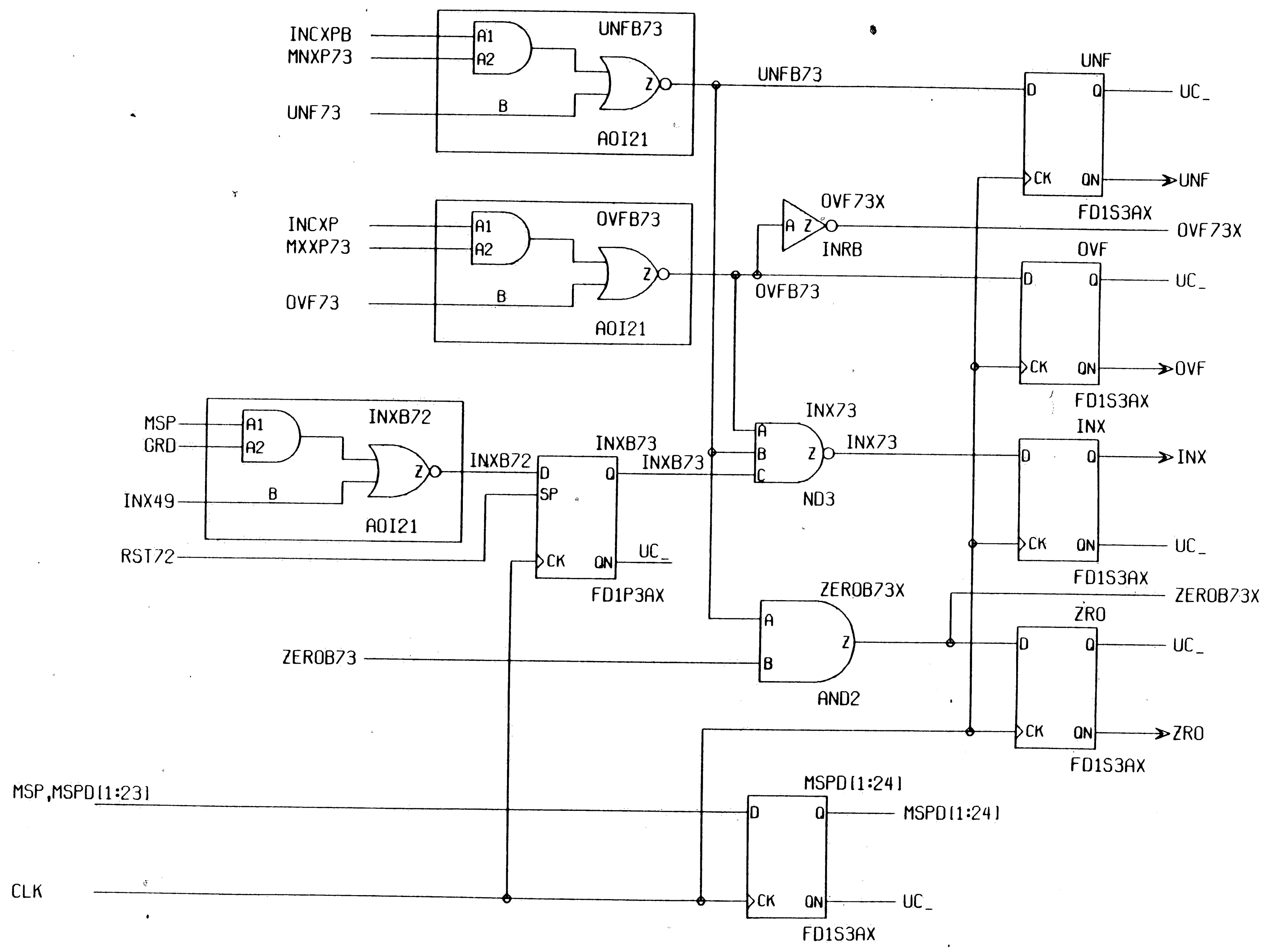


Figure A.14. EXPFMT.6
52

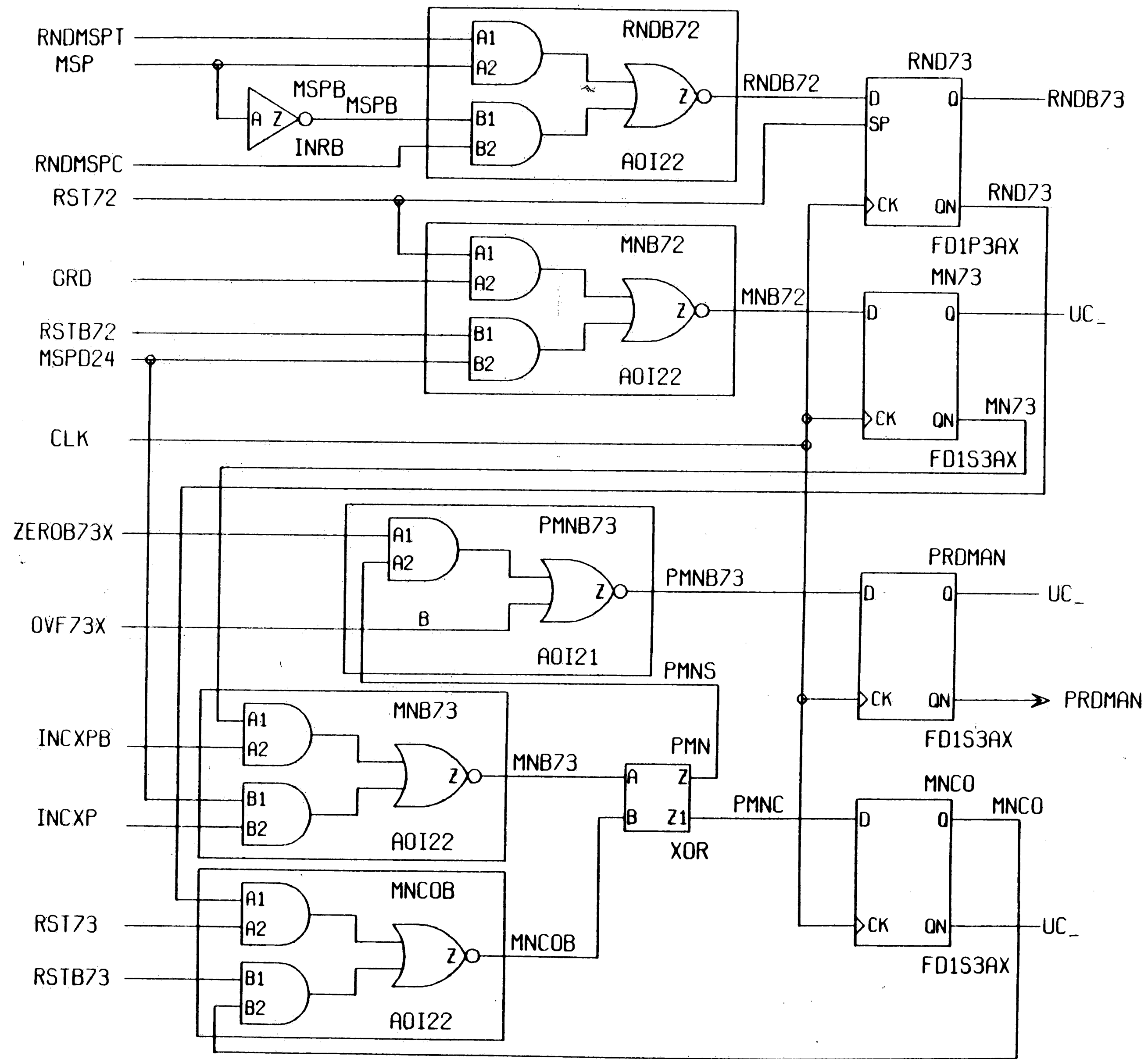


Figure A.15. EXPFMT.7
53

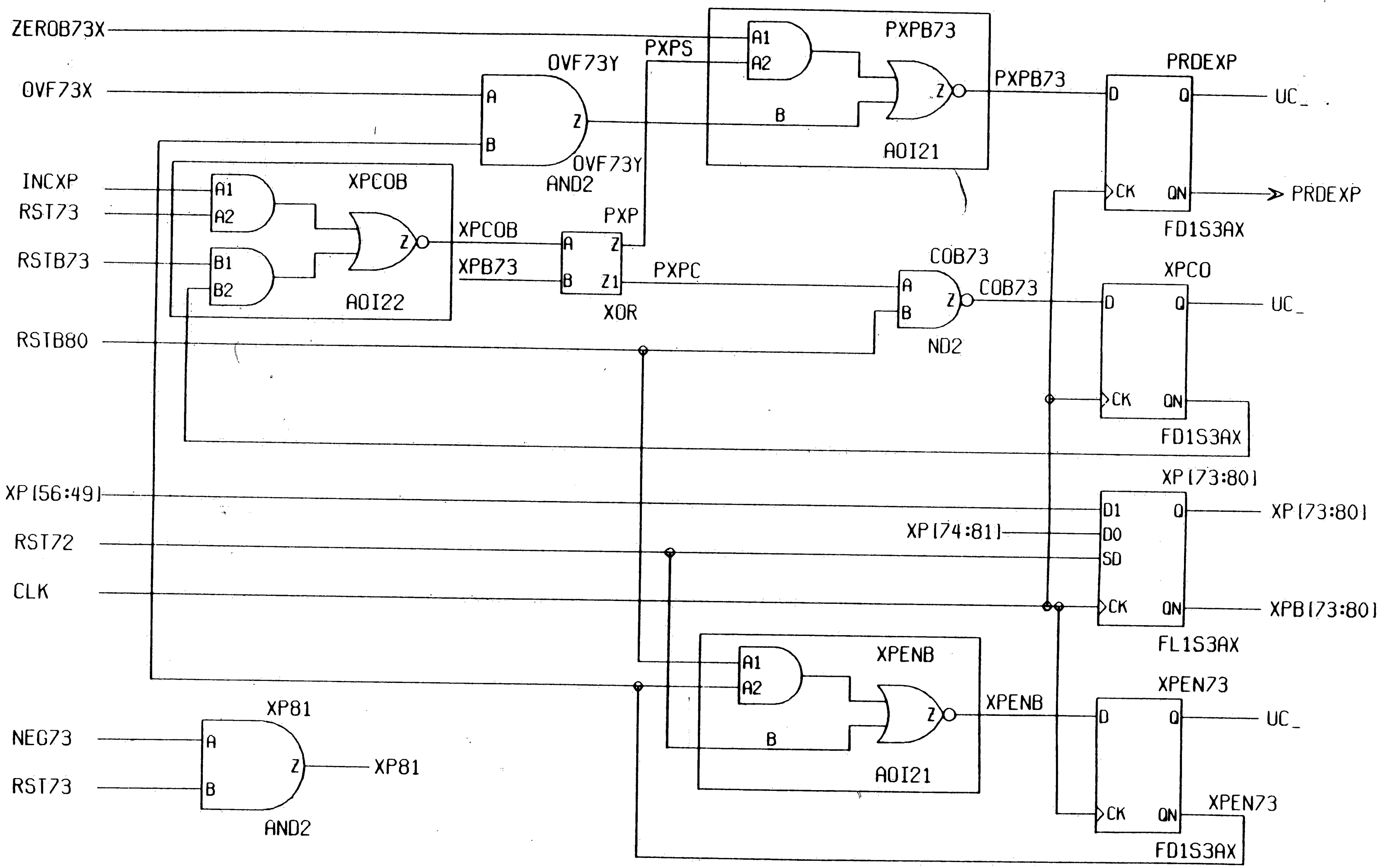


Figure A.16. EXPFMT.8
54

Appendix B

Floating Point Bit-Sequential Adder Schematics

This appendix includes the complete schematics for the floating point bit-sequential adder described in chapter 4.

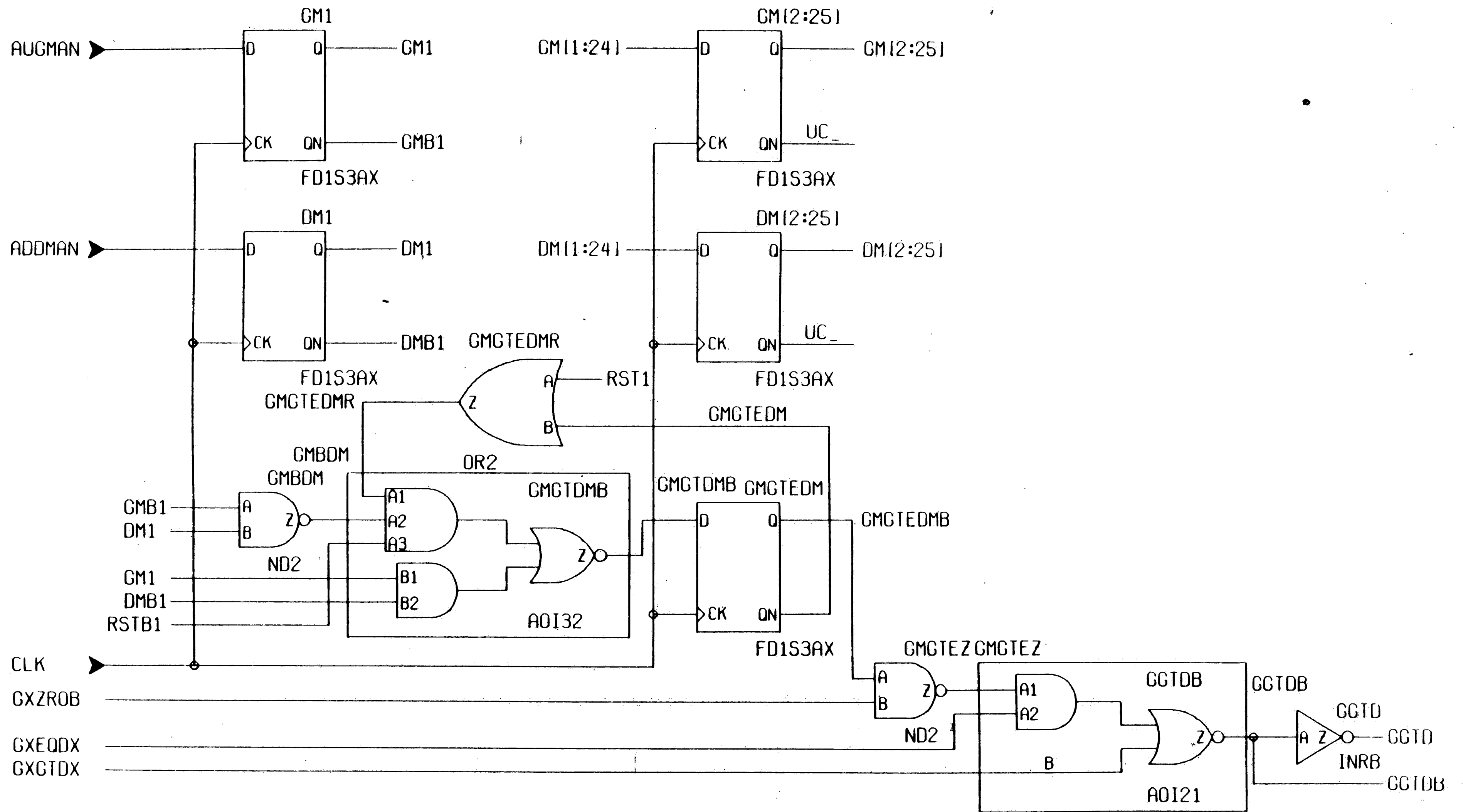


Figure B.1. FPADD.1

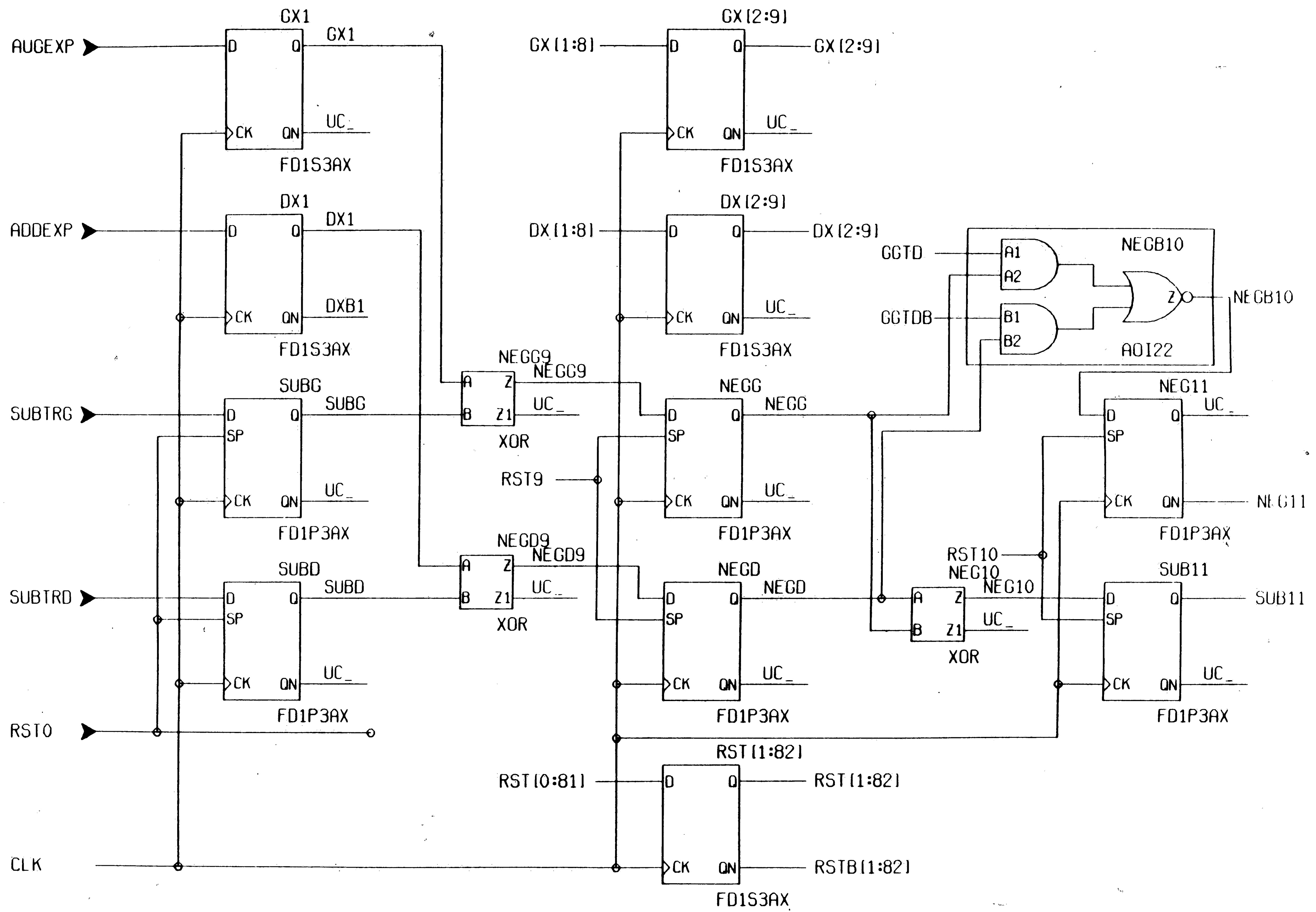


Figure B.2. FPADD.2
57

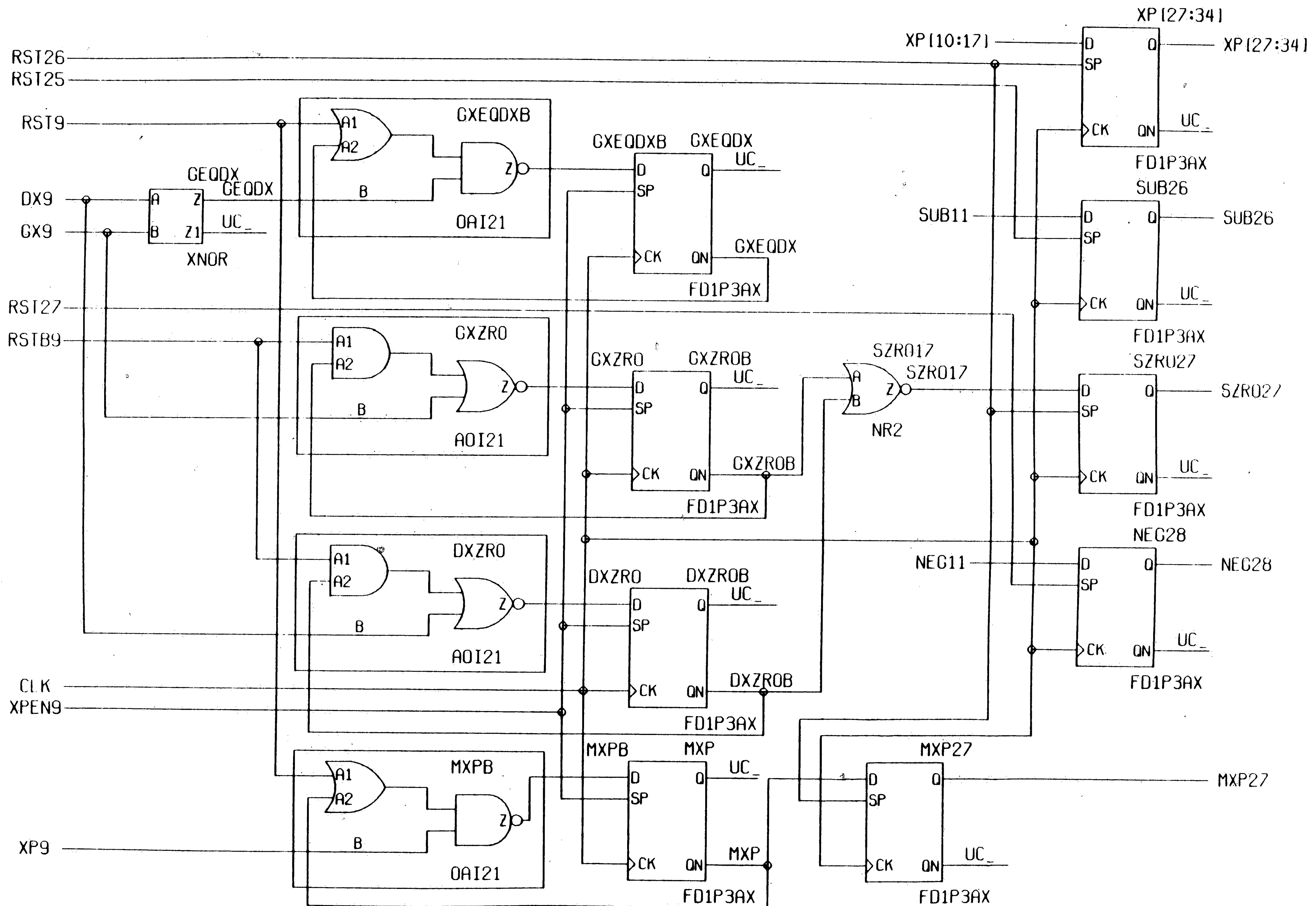
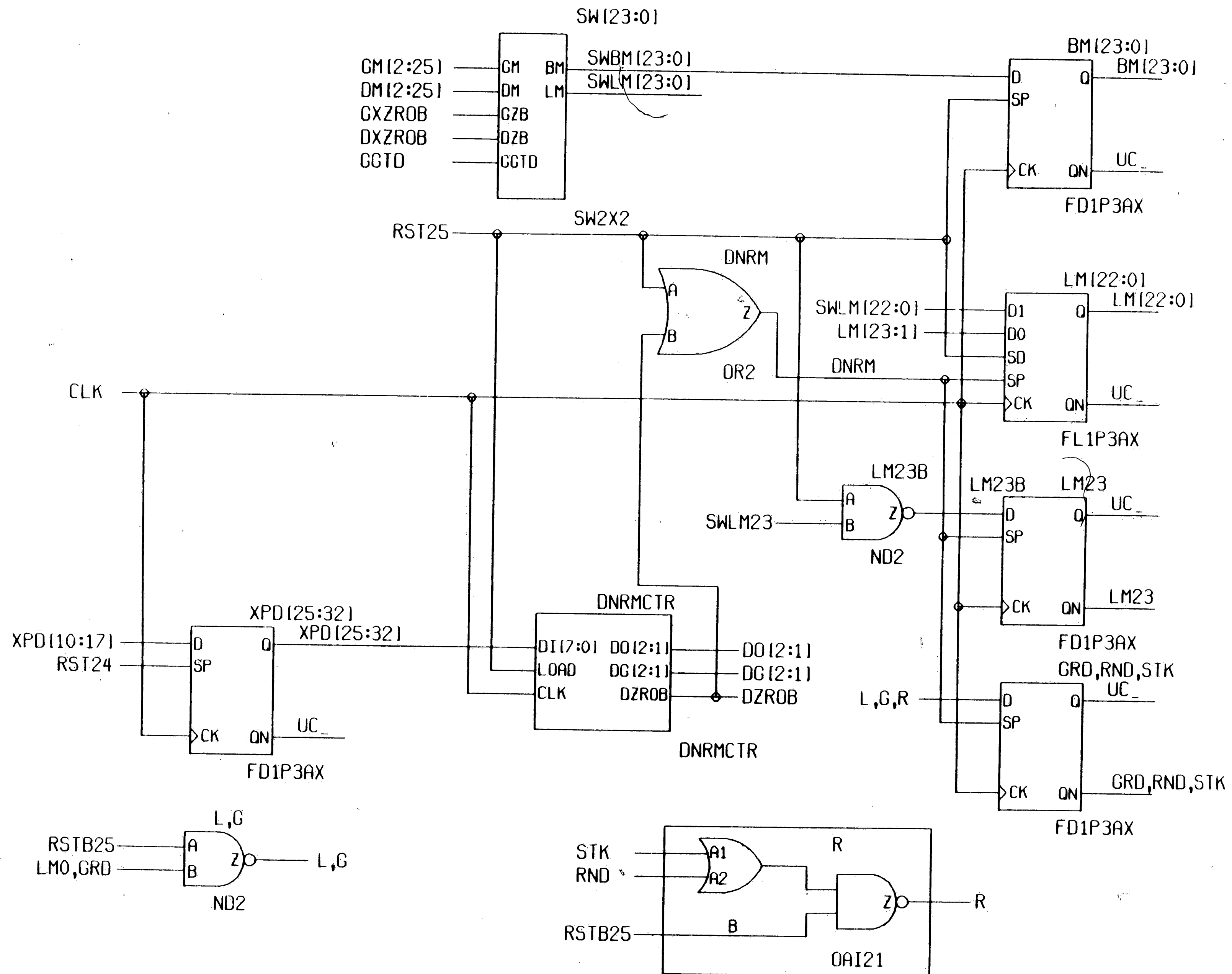


Figure B.4. FPADD.4
59

Figure B.5. FPADD.5
60



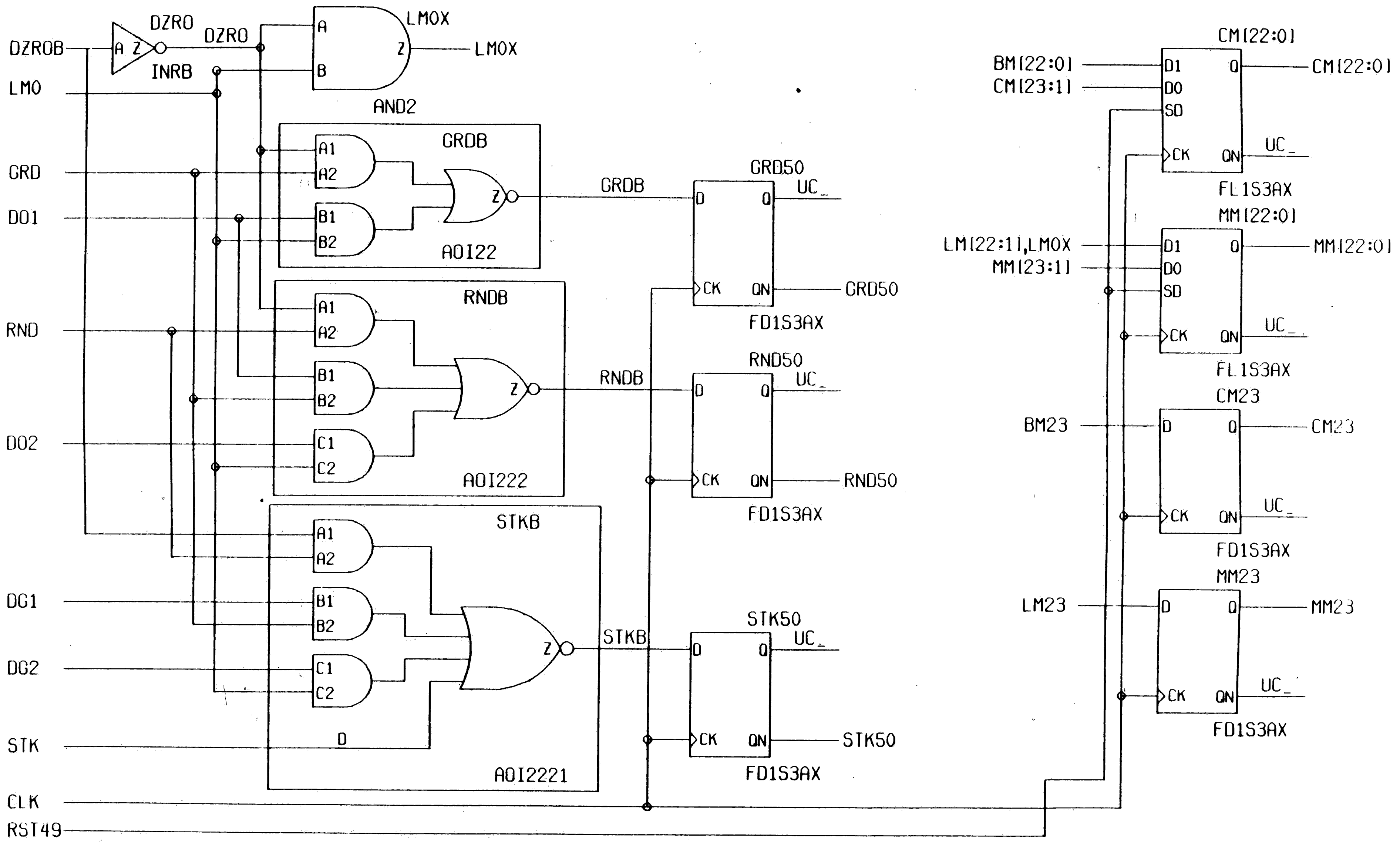


Figure B.6. FPADD.6
61

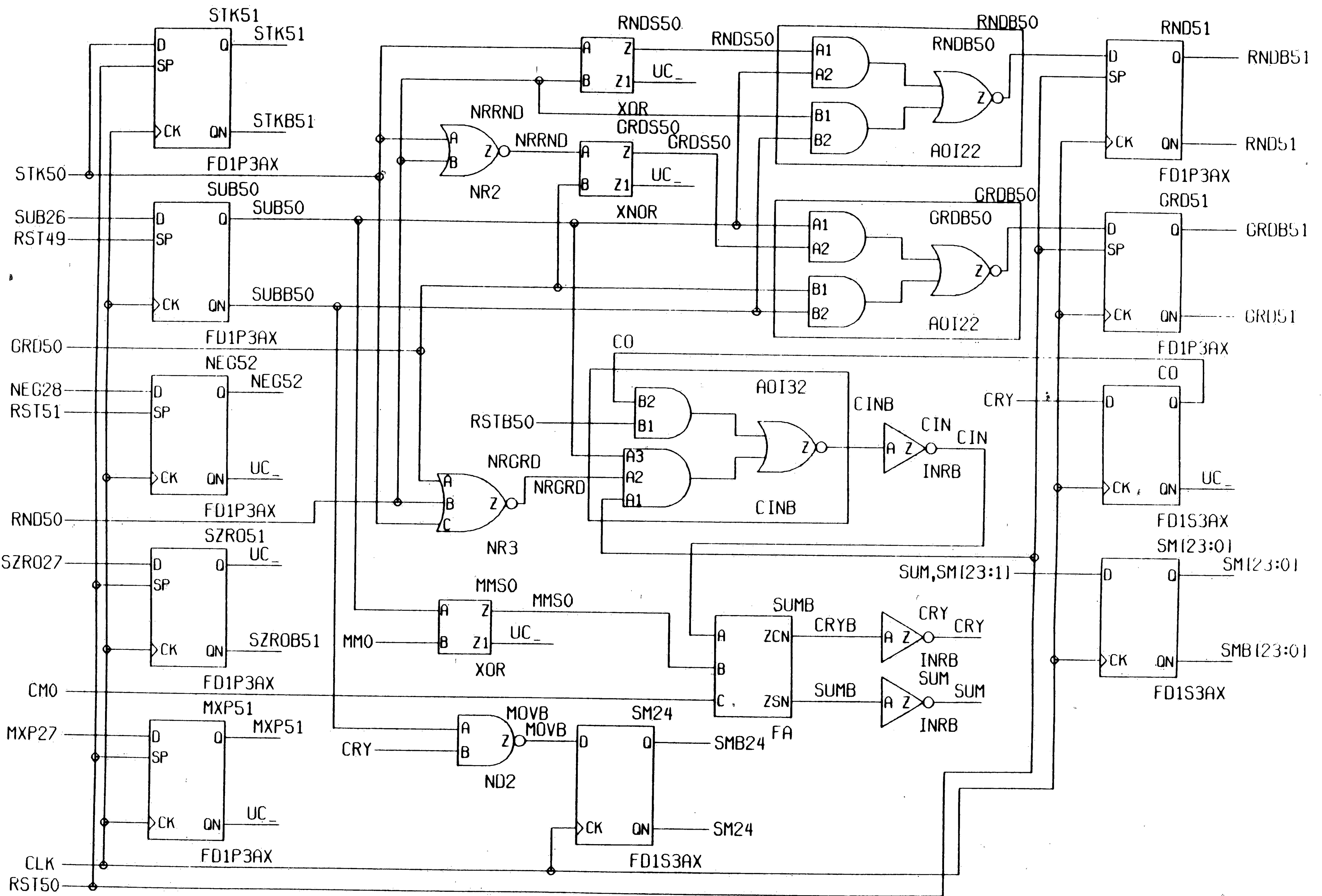


Figure B.7. FPADD.7
62

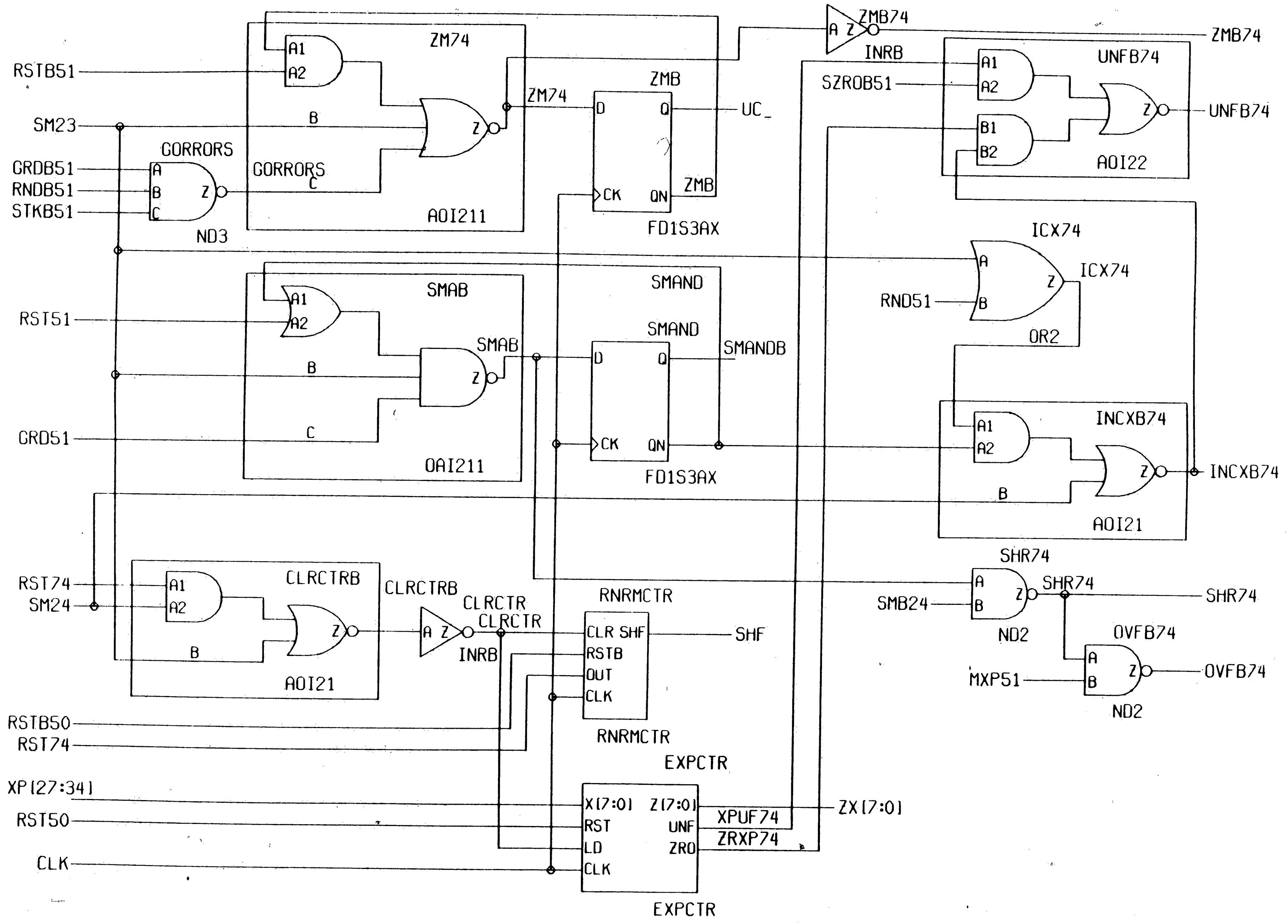


Figure B.8. FPADD.8
63

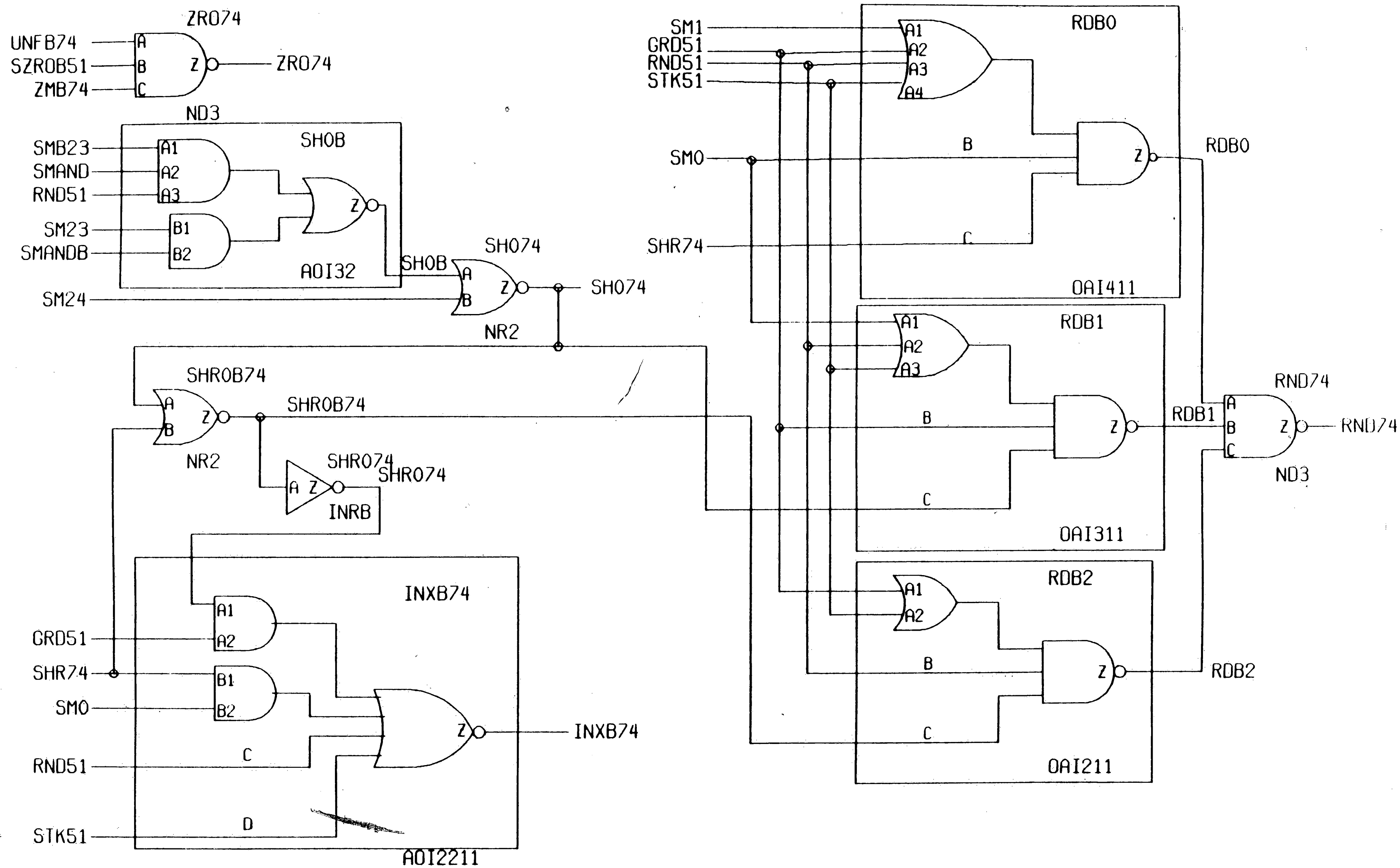


Figure B.9. FPADD.9
64

Figure B.10. FPADD.10
65

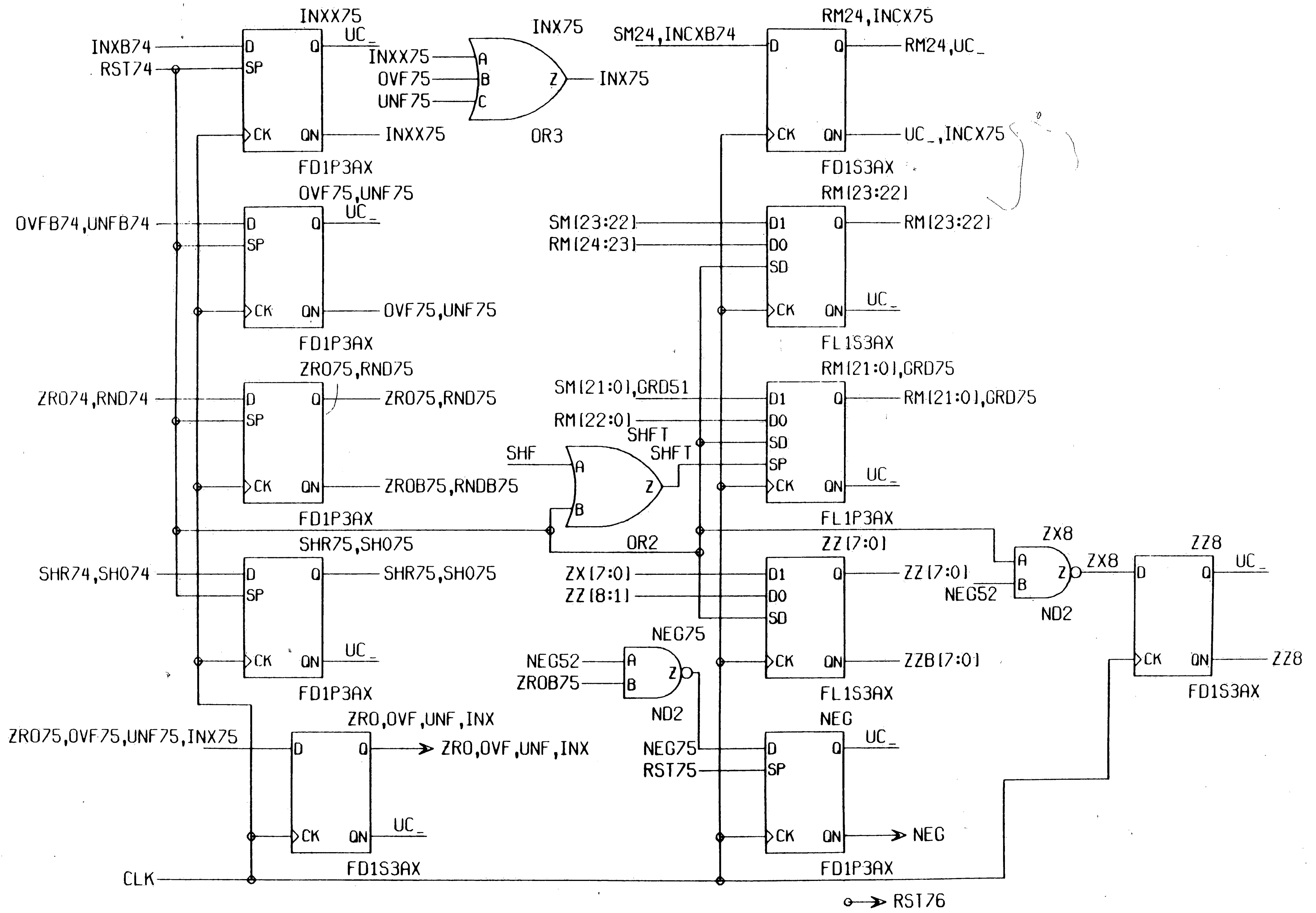
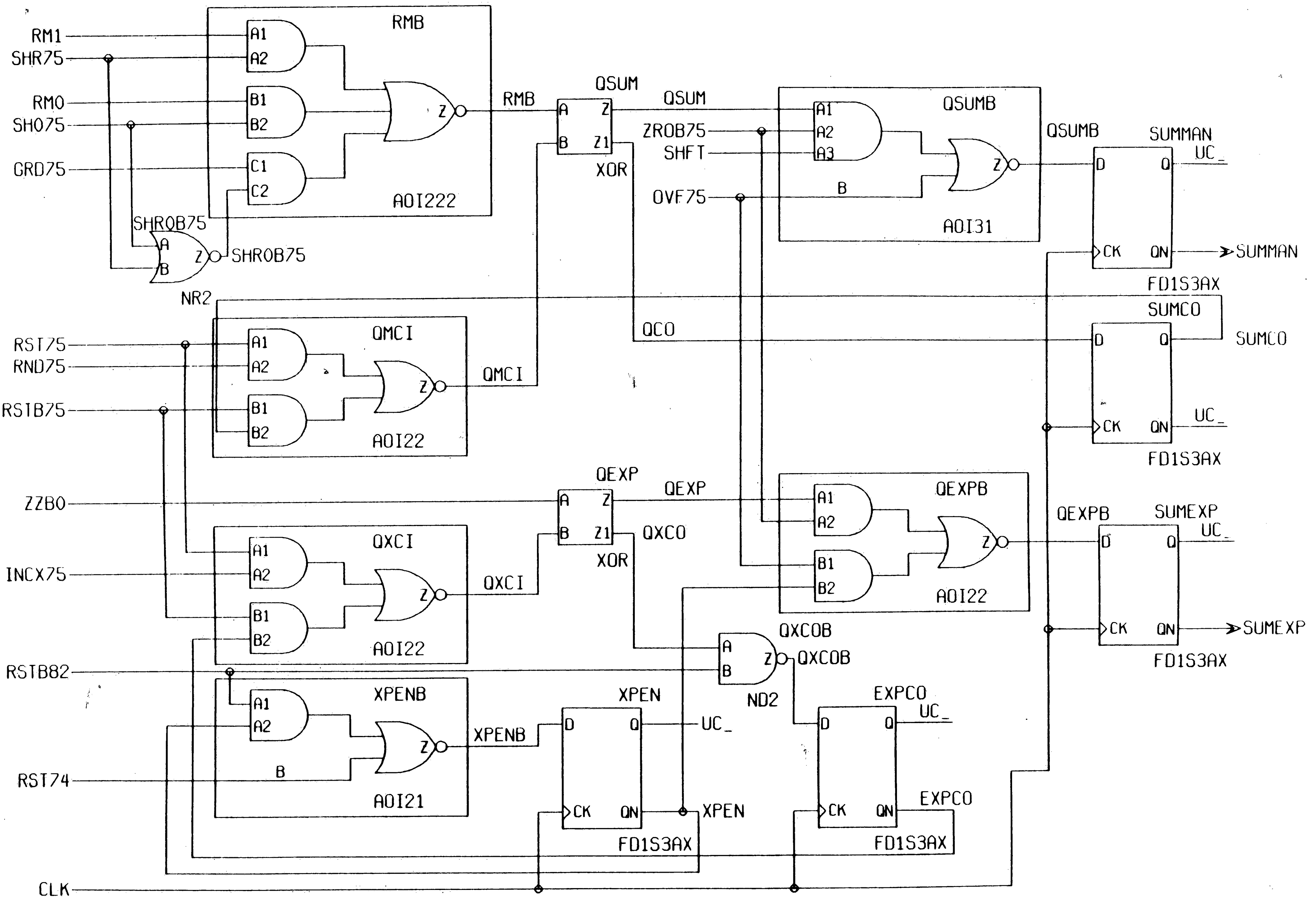


Figure B.11. FPADD.11

66



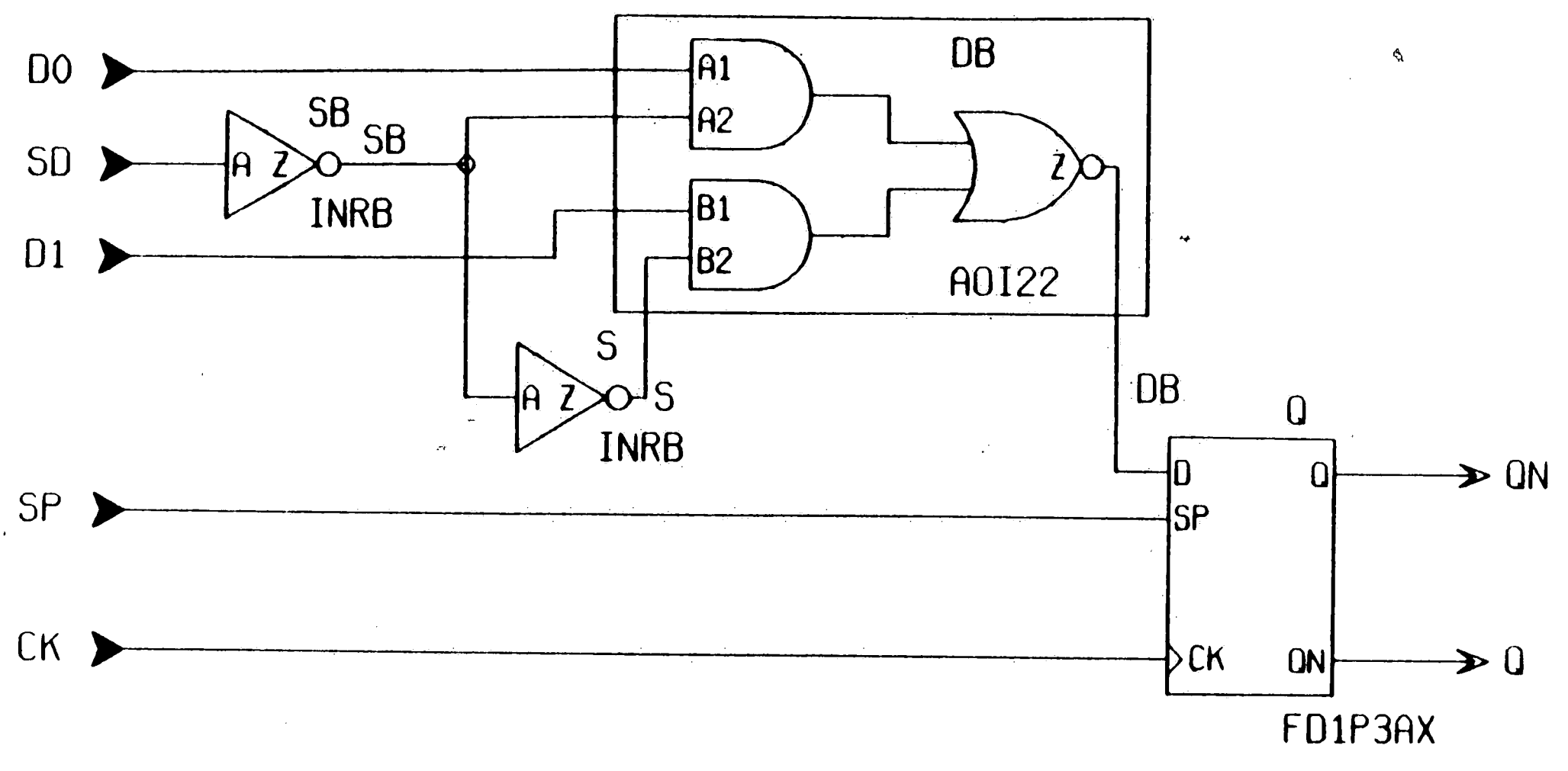


Figure B.12. FL1P3AX.1
67

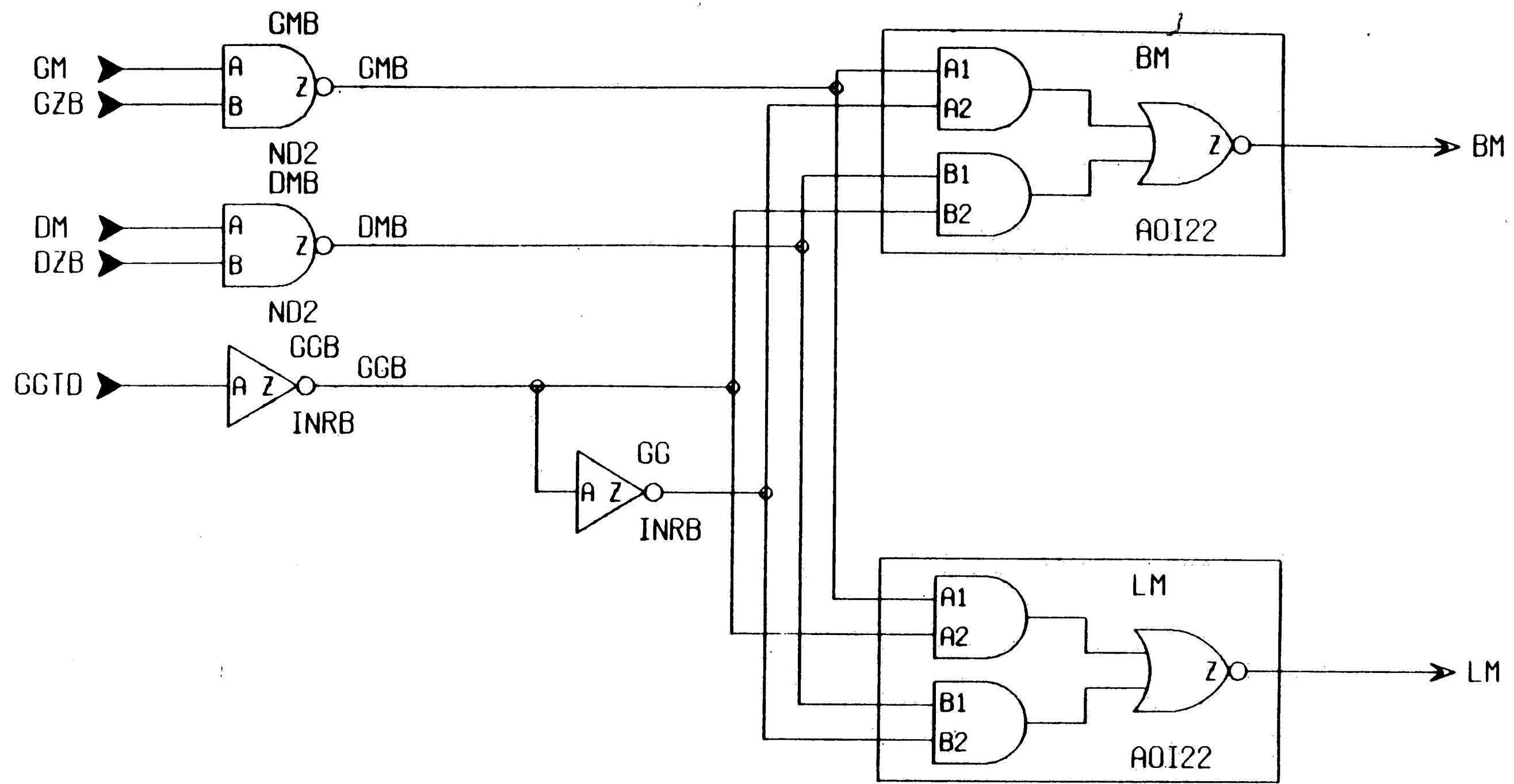


Figure B.13. SW2X2.1
68

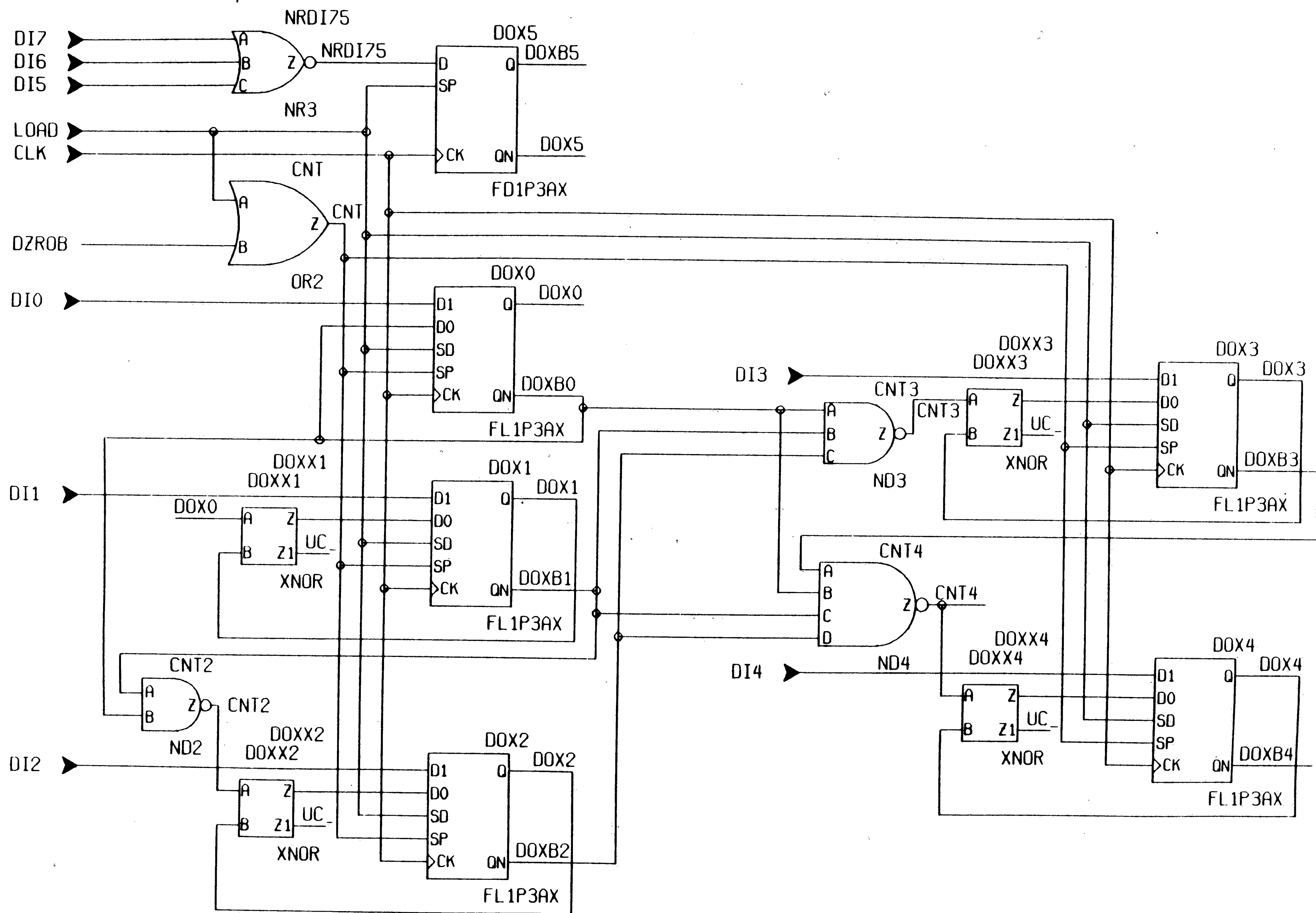


Figure B.14. DNRMCCTR.1
69

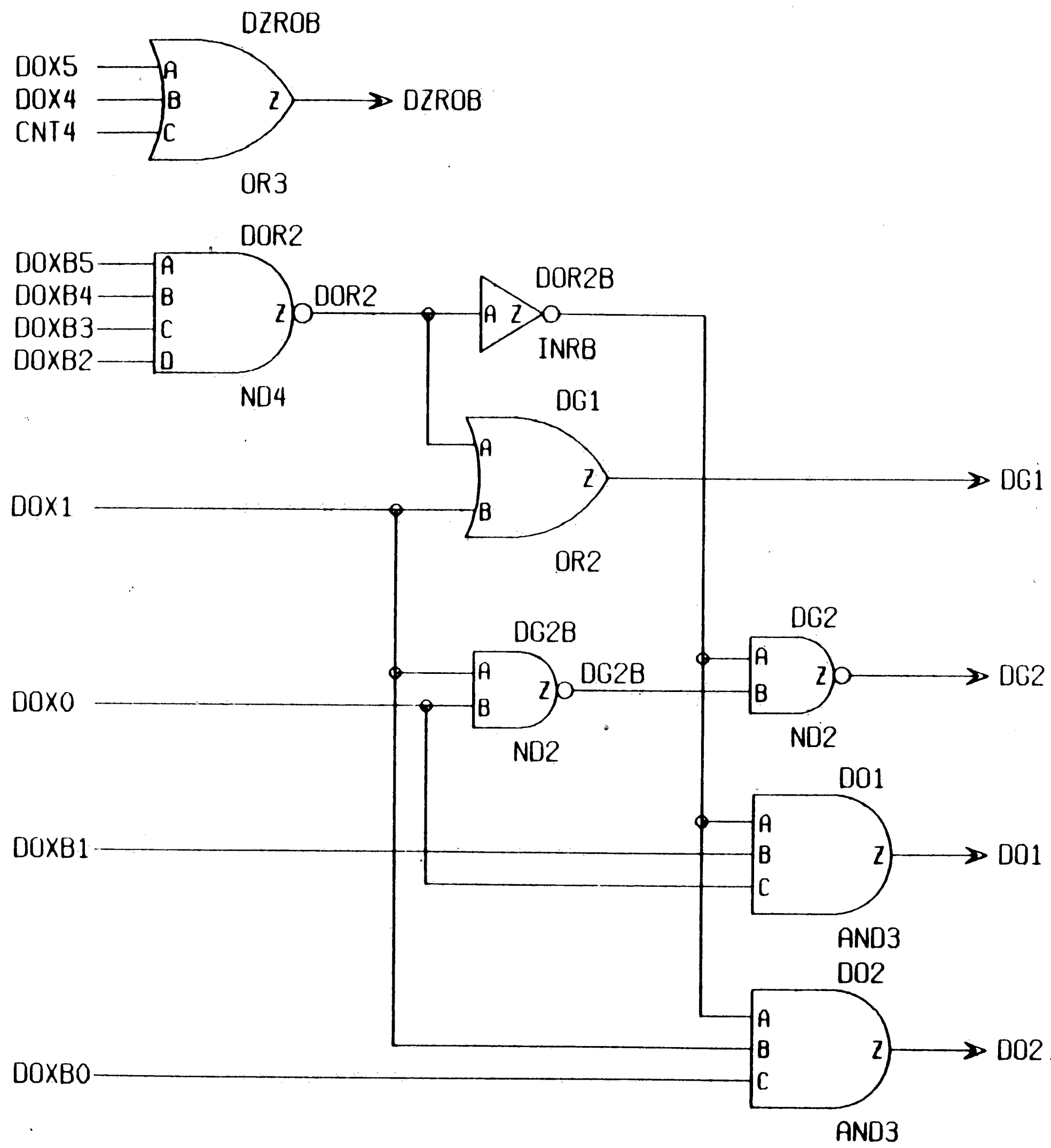


Figure B.15. DNRMCTR.2
70

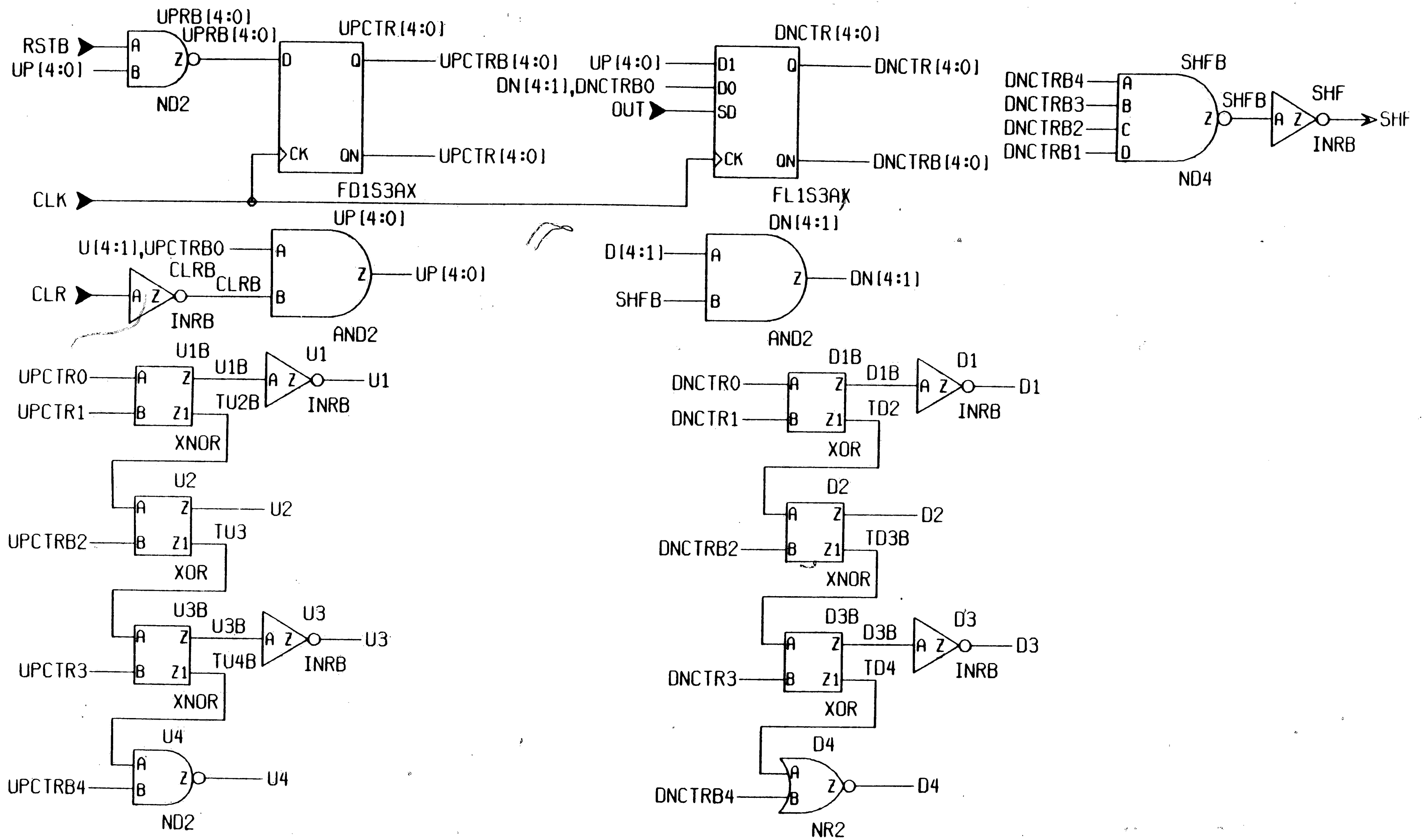


Figure B.17. RNRMCCTR.1
72

Vita

David Mark Blaker was born to Dr. J. Warren Blaker and Mrs. Cynthia Geber Blaker on December 3, 1957, in Boston, Massachusetts. He graduated from Arlington High School, in New York State, in 1974. He then went on to receive the Bachelor of Science degree in Electrical Engineering from the Massachusetts Institute of Technology in 1979. From there he went to the Hewlett-Packard Corporation, in Colorado Springs, Colorado, where he designed portions of timing analyzers and in-circuit emulators. In 1980, he went to ADR Ultrasound in Tempe, Arizona, where he was coinventor of a scan convertor for ultrasound imaging, on which a U.S. patent was issued. He moved to Edge Computer Corporation in 1984, where he designed part of the floating point unit for a high performance work station. In 1985, he moved to AT&T Bell Laboratories in Allentown, Pennsylvania, where he designs VLSI DSP devices, and is responsible for design for testability and built-in self test strategies.

David is married to Polly Jean Blaker (nee Williams), and has two children: Sarah Elizabeth Blaker, aged 2½ years, and Nathan Isaac Blaker, aged ½ year. His personal interests include playing with his children, skiing, walking and reading.