Design and Implementation of the Integer Unit Datapath of the MAP Cluster of the M-Machine

by

Parag Gupta

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

© Parag Gupta, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author .....
Department of Electrical Engineering and Computer Science
May 18, 1996

Certified by .....
William J. Dally
Professor
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Design and Implementation of the Integer Unit Datapath of the MAP Cluster of the M-Machine

by

Parag Gupta

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

Abstract

This thesis presents the design and implementation of the integer unit datapath found in the execution stage of the MAP cluster pipeline. It begins with a discussion of design flow and techniques before presenting the integer unit specification. Next, the integer unit architecture is developed. Finally, the design and implementation of each module within the integer unit is presented. Issues ranging from architecture to logic to circuits to layout are presented.

Thesis Supervisor: William J. Dally
Title: Professor

Acknowledgments

There are many people without whom this thesis would not be possible. First of all, I would like to thank Professor Bill Dally for giving me the opportunity to work in his group. The past year and a half have been the most grueling but most rewarding time for me at MIT. I learned so much by working on this real world type project. Many thanks go to Andrew Chang for being so patient with me and all of my questions. I know I was a pain in the neck and I really appreciate all of your help. The same goes to Dan Hartman for always being around to answer yet more stupid questions.

I would also like to thank my good friend Frank Chan for sharing all of his thesis horror stories with me. He made sure I started work on this thesis early, thus making the weeks before thesis due date unstressful.

Most of all, I would like to thank my family. I would like to thank them for encouraging me to go to MIT. It was my mother's dream that I go to MIT. That probably helped with my decision. I would like to thank my family for bearing the financial burden of an MIT education, allowing me to concentrate on academics. I love you very much.

Contents

1 Introduction **8**

1.1 M-Machine Architectural Overview 8

1.2 Scope 12

1.3 Thesis Outline 13

2 Design Flow and Methodology **14**

2.1 Architectural Design 14

2.2 Module Design and Implementation 18

2.3 Testing 20

2.3.1 Functional Testing 20

2.3.2 Timing Verification 21

2.3.3 Variations in Verification Styles 23

2.4 Summary 24

3 Integer Unit Architecture **26**

3.1 Integer Unit Specifications 26

3.2 The Major Modules 27

3.3 Putting It All Together 32

4 Integer Unit Submodules **37**

4.1 Adder/Subtractor 37

4.1.1 PGK Block 39

4.1.2 Group PGK 42

4.1.3 Global Carry Chain 49

4.1.4 Local Carry Chain 51
4.1.5 Final Cell 52
4.2 Zero Detect 54
4.2.1 Precharged Gate 54
4.2.2 Pseudo-NMOS Gate 55
4.2.3 Static Gates 55
4.2.4 Cascode Voltage Swing Logic 56
4.2.5 Push/Pull CVSL 57
4.3 Shifter 58
4.4 Selector 60
4.4.1 Extracts 60
4.4.2 Inserts 61
4.4.3 Immediates 61
4.4.4 Shift/Rotate Operations 61
4.4.5 Implementation 62
4.5 Mask Generator 63
4.6 Segment Checker 65
4.7 Boolean Unit 66
4.8 Send Unit 68
4.9 Errval Generator 69
4.10 Latches 69
4.11 Muxes 71
4.12 Demultiplexor 71

5 Conclusions **73**

5.1 Summary 73
5.2 Suggestions For Further Study and Improvement 74

A M-Machine Instruction Set Architecture **75**

A.1 Arithmetic Operations 76
A.1.1 ADD 76

A.1.2 ADDU - add unsigned 76
A.1.3 SUB - subtract 76
A.1.4 SUBU - subtract unsigned 77
A.1.5 ASH - arithmetic shift 77
A.1.6 LSH - logical shift 77
A.1.7 ROT - rotate 78
A.1.8 AND - bitwise logical and 78
A.1.9 OR - bitwise logical or 78
A.1.10 XOR - bitwise logical exclusive or 78
A.1.11 NOT - bitwise logical negation 79
A.2 Byte Manipulation 79
A.2.1 EXTB - extract byte 79
A.2.2 EXTH - extract halfword 79
A.2.3 INSB - insert byte 80
A.2.4 INSH - insert halfword 80
A.3 Comparison Operations 80
A.3.1 ILT - integer less than 80
A.3.2 ILE - integer less than or equal 81
A.3.3 ULT - unsigned less than 81
A.3.4 ULE - unsigned less than or equal 82
A.3.5 INE - integer not equal 82
A.3.6 IEQ - integer equal 82
A.3.7 CCAND - condition code logical and 83
A.3.8 CCOR - condition code logical or 83
A.3.9 CCNAND - condition code logical nand 83
A.4 Data Movement 84
A.4.1 MOV - move register 84
A.4.2 EMPTY - unset register presence bits 84
A.4.3 CCEMPTY - unset condition register presence bits 84
A.5 Control Flow Operations 85

A.5.1 BR - branch 85
A.5.2 JMP - jump 85
A.5.3 ILL - illegal instruction 86
A.6 Address Calculation 86
A.6.1 LEA - load effective address 86
A.6.2 LEAB - load effective address from base 87
A.6.3 SETPTR - set pointer bit 87
A.6.4 ISPTR - test pointer bit 87
A.6.5 ISERR - test for *Errval* 88
A.7 Immediate Operations 88
A.7.1 SHORU - shift & or unsigned 16-bit immediate 88
A.7.2 IMM - create 16-bit immediate 88
A.8 Configuration Space Operations 89
A.8.1 IGTLB - GTLB Access 89
A.9 Communication Operations 89
A.9.1 ISND0 - send priority 0 message, *user level* 89
A.9.2 ISND0O - send priority 0 message, *user level* 90
A.9.3 ISND0P - send priority 0 message physical 90
A.9.4 ISND0PO - send priority 0 message physical, ordered 91
A.9.5 ISND0PNT - send priority 0 message physical, no throttling . 91
A.9.6 ISND0PNTO - send priority 0 message physical, no throttling, ordered 92
A.9.7 ISND1PNT - send priority 1 message physical, no throttling . 92
A.9.8 ISND1PNTO - send priority 1 message physical, no throttling ordered 93

B Schematics **94**

List of Figures

1-1 M-Machine Architecture 9

1-2 MAP Chip Architecture 10

1-3 MAP Chip Cluster Architecture 11

1-4 Relationship of V-Threads to H-Threads 12

2-1 Initial Architecture of Example Unit - Module Identification 15

2-2 Intermediate Architecture of Example Unit - Sharing 16

2-3 Architecture of Example Unit - Latches 17

2-4 Comparison of Two *And* Gates 19

2-5 Design and Testing Flow 23

2-6 Integer Unit Design Flow 24

3-1 Modules needed to execute *leab* 30

3-2 Intermediate integer unit architecture 32

3-3 Intermediate integer unit architecture with output mux 33

3-4 Intermediate integer unit architecture with result driver 34

3-5 Intermediate integer unit architecture with latches 35

3-6 Final integer unit architecture 36

4-1 Block diagram of adder internals 39

4-2 Block diagram of pgk block internals 40

4-3 Schematic of pass gate version of pgk block 41

4-4 Schematic of static gate version of pgk block 42

4-5 Schematic of non-*xor* version of pgk block 43

4-6 Illustration of a typical link in the carry chains 44
4-7 Group pgk module with a chain of 8 unbroken links 46
4-8 Group pgk module with a broken chain of 8 links 47
4-9 Group pgk module with speculative chain 48
4-10 Global carry chain schematic 50
4-11 Local carry chain schematic 52
4-12 Final cell schematic 53
4-13 CVSL Zero Detect Slice Schematic 56
4-14 Push/Pull CVSL Zero Detect Slice Schematic 57
4-15 Shifter stages and interconnections 58
4-16 Internals of bit stage of shifter 59
4-17 Layout strategy for the selector 62
4-18 Mask generator basic cells and implementation for low 2 bits 64
4-19 Schematic of first stage of segment checker 66
4-20 Boolean unit schematic 67
4-21 Block diagram of send unit internals 68
4-22 Schematic of a latch 69
4-23 Block diagram of recirculating p-latch 70
4-24 Internals of the tri-state driver 71

Chapter 1

Introduction

This thesis presents architectural and circuit design techniques used in the design and implementation of the integer execution unit used in the M-Machine. The M-Machine is an experimental multicomputer developed by the Concurrent VLSI Architecture Group in the Artificial Intelligence Lab at the Massachusetts Institute of Technology. The M-Machine will be used to investigate different mechanisms for parallelism and to investigate architectures and design techniques that make efficient use of 1995 technology [1].

1.1 M-Machine Architectural Overview

The M-Machine consists of computing nodes that are connected by a 3-D mesh network, where each channel of the network operates at 800 megabytes per second and can transmit in both directions simultaneously. Each of the computing nodes consists of a multi-ALU processor (MAP) chip plus external DRAM. See Figure 1-1 for an illustration of this. The MAP chip itself contains four execution clusters, a memory subsystem, and a communications subsystem. The memory subsystem consists of four cache banks and an external memory interface, while the communications subsystem consists of network interfaces and a network router. There are two crossbar switches that connect the four execution clusters, memory subsystem, and communications subsystem together. One of these switches, called the M-switch, is used to send

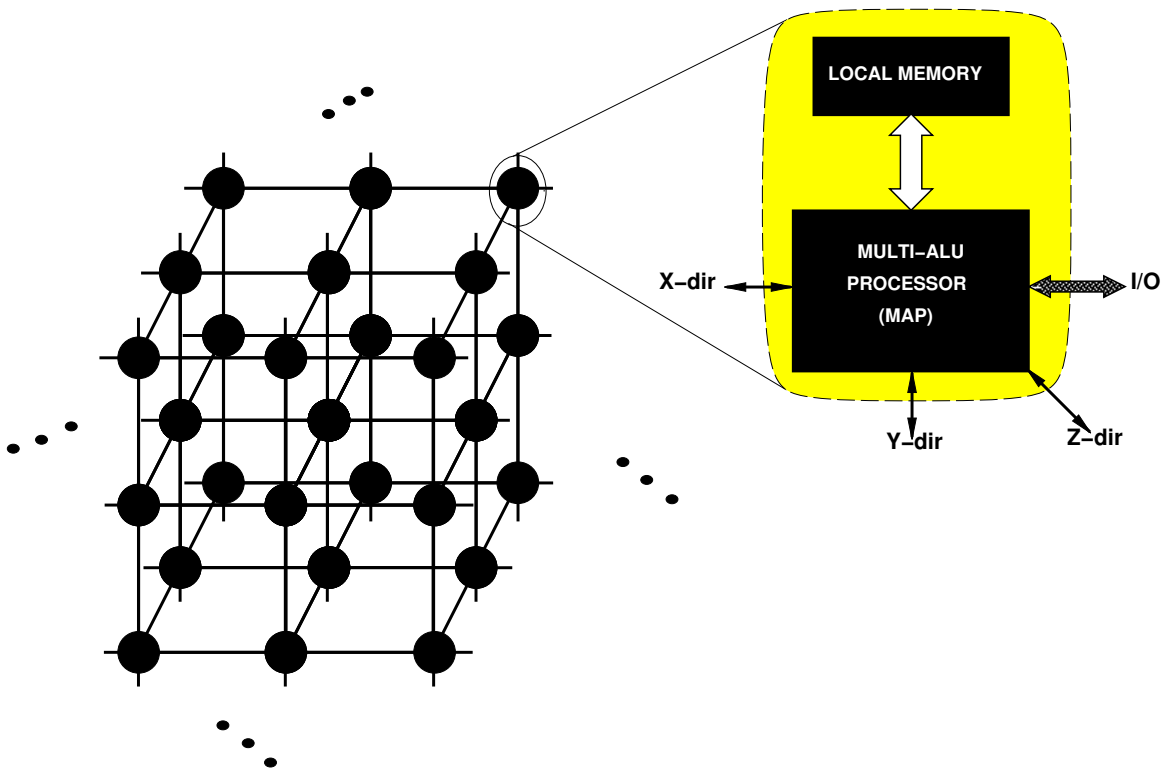


Figure 1-1: M-Machine Architecture

requests from the clusters to the memory subsystem, while the othe switch, called the C-switch, is used to send data from the memory subsystem to the clusters, to send data between the four execution clusters, and to send/receive data via the internode communications subsystem. See Figure 1-2 for an illustration of the architecture of the MAP chip [2].

Each of the four clusters is a 64-bit, three-issue, fully pipelined microprocessor. The three functional units found on each cluster are an integer unit, floating point unit, and a memory unit. The memory unit is primarily responsible for sending out load/store requests to the memory subsystem, but it also has the capability to handle many integer operations. Also found in each cluster is an integer register file, a floating point register file, and an instruction cache. One instruction for a cluster consists of up to three operations, one for each functional unit. Each of these operations are issued at the same time but can complete out of order as different

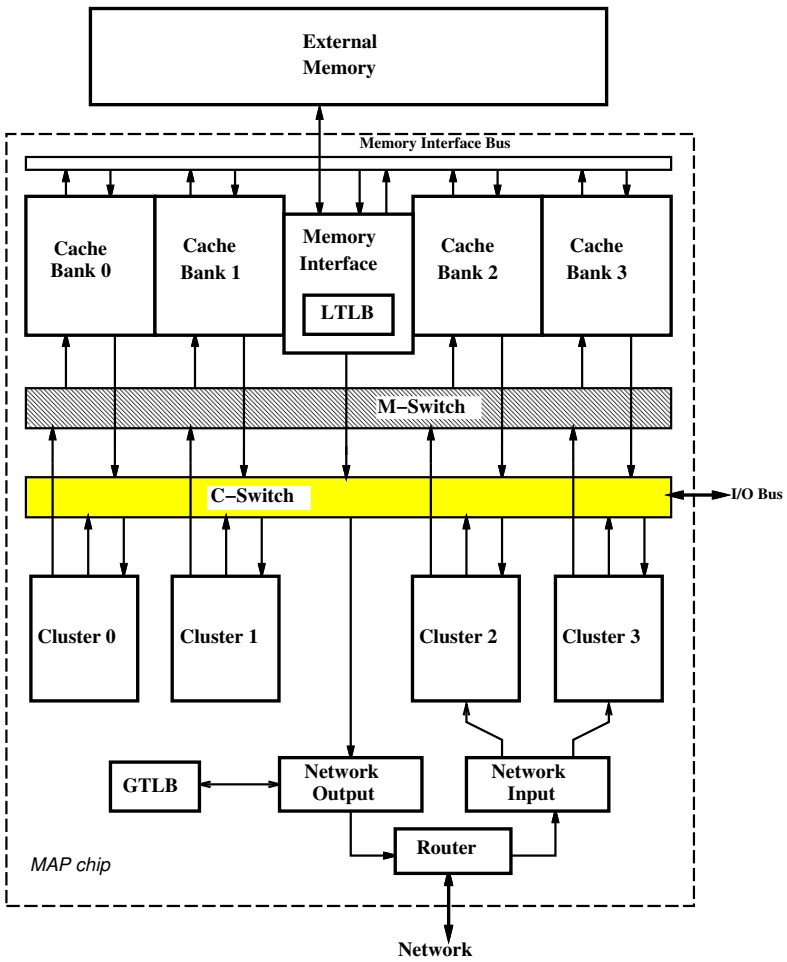


Figure 1-2: MAP Chip Architecture

operations may have different latencies. See Figure 1-3 for a block diagram of the high level architecture of a cluster [2].

The M-Machine provides the hardware that allows many threads to run concurrently on a node. There are enough resources for six V-Threads, each of which consists of four H-Threads, one H-Thread per cluster. Four of the six V-Threads are user threads, while the other two are for handling exceptions and events. H-Threads can communicate through registers, with each having the capability to write values to the register files of the other H-Threads within the same V-Thread. The H-Threads in a V-Thread can execute as independent threads with different control flows for

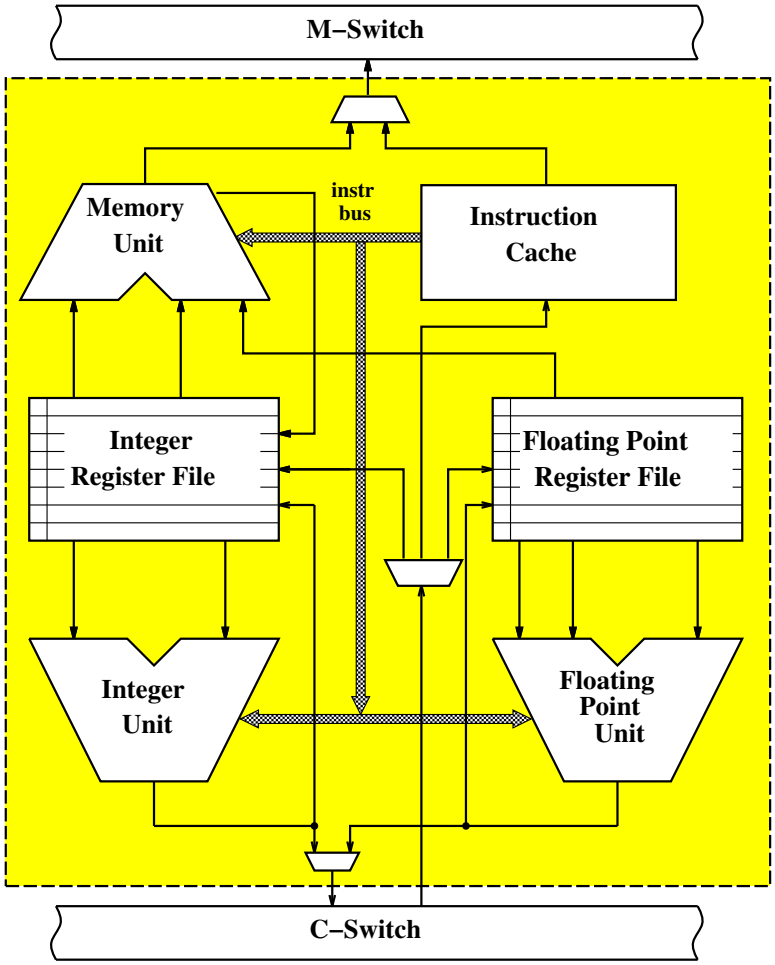


Figure 1-3: MAP Chip Cluster Architecture

exploiting loop or thread-level parallelism. On the other hand, the H-Threads can be scheduled statically as a unit to exploit instruction level parallelism, as in a VLIW machine. The V-Threads are interleaved at run-time over the clusters on each cycle. The synchronization pipeline stage handles this by holding the next instruction to be issued from each V-Thread and only issuing an instruction when all of its operands and resources are ready. Since these V-Threads can be interleaved with no delay, switching between threads to mask any pipeline or memory delays can be done without any time penalty. This allows the M-Machine to efficiently execute programs with varying granularities of parallelism [1]. See Figure 1-4 for an illustration of the

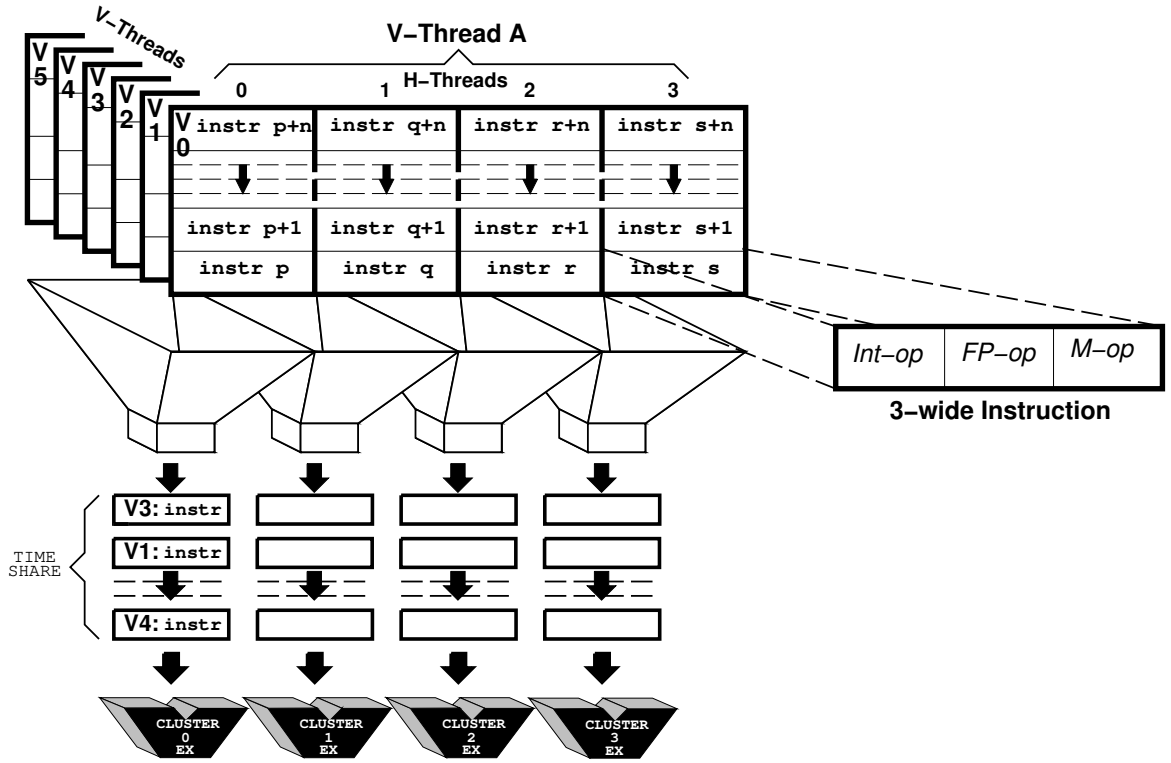


Figure 1-4: Relationship of V-Threads to H-Threads

relationship between V-Threads and H-Threads.

1.2 Scope

The MAP pipeline consists of five stages. These stages are instruction fetch, register read, synchronization, execution, and write back. This thesis will discuss the design and implementation of the execution stage of the integer unit datapath. The execution stage is where the actual integer unit operation is carried out and a result produced. The synchronization stage provides the operands to the integer unit and the instruction information to the integer unit control. The integer unit datapath sends its result to the writeback stage of the pipeline, where the result is written back to the proper destination, as determined by integer unit control.

In developing the design of the integer unit datapath, the instruction set will be

presented as it relates to the integer unit. Much of the thesis will be spent discussing the circuits that were used in the implementation of the integer unit as well as the functionality of the various modules that make up the integer unit datapath. More in-depth circuit studies of Manchester carry chains and zero detection units are presented. Tradeoffs, circuit techniques, alternate designs, and testing strategies will be discussed where relevant.

1.3 Thesis Outline

This thesis is organized as follows. The next chapter will present the architectural design process, the module design and implementation process, and the testing methodology that was used. Following that will be a chapter that presents the architectural design based on the MAP instruction set architecture. Then comes a chapter on the actual design and implementation of each of the modules that make up the integer unit. Here, various designs are discussed along with any issues/tradeoffs associated with the different designs. Then the implementations and circuit issues are presented. Finally any testing/verification issues are discussed. The chapter after this presents the testing/verification involved in the integration of the integer unit modules. The final chapter serves as the conclusion to this thesis. The appendices contain such items as the integer unit ISA, some verilog and spice test decks, and the integer unit schematics.

Chapter 2

Design Flow and Methodology

This chapter discusses the process of designing the integer unit datapath. It begins with a section on high level architectural design, followed by a section on individual module design and implementation, and then a section on testing methodology. The chapters following this one present the use of these techniques in actually building the integer unit datapath.

2.1 Architectural Design

When designing a large circuit, such as a functional unit in a microprocessor, the first step is to understand what the unit is supposed to do, how much time it is supposed to do it in, and what its datapath inputs and outputs need to be. In order to figure out what the unit is supposed to do, a good place to start is with the instruction set architecture (ISA). This will give a good idea of what the different tasks are that are expected of the unit to be designed. The amount of time that the unit has to do whatever it is supposed to do is largely determined by the clock rate, the time at which its inputs are available, and the time at which the outputs are needed. The datapath inputs and outputs are largely determined based on the pipeline design of the processor. In the case of the execution stage of the pipeline, the datapath inputs and outputs consist of operands and results.

The first step in looking over the ISA is to determine what submodules are needed

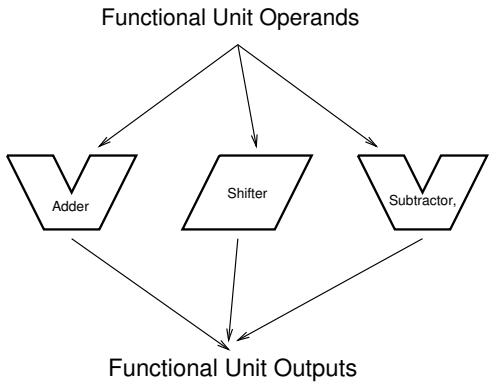


Figure 2-1: Initial Architecture of Example Unit - Module Identification

within the functional unit for each instruction. For example, suppose there is an *add* instruction. This tells you that the unit will need an adder with a path from the unit's inputs to the adder's inputs. It also tells you that you will need a path to the unit's output. Now suppose you come across a *shift* instruction. This tells you you need a shifter with paths from the unit's input to the shifter's input as well as a path from the shifter's output to the unit's output. Additionally, assume there is a *subtract* instruction. This means some kind of a subtraction unit is needed as well. After going through this process for each instruction, you now have an idea of what the major modules within the unit are going to be. See Figure 2-1 for an illustration of this initial architecture.

The next step is to figure out which of the modules can be shared across many instructions and also where muxes are going to be needed. For example, suppose there is an *add* and *subtract* instruction. Instead of having a separate adder and a separate subtractor, the adder can easily be turned into an adder/subtractor by adding a little bit of logic and also adding a control signal indicating whether to do an add or subtract. Similarly, suppose an adder and shifter need a path to the output. This implies that the outputs of the adder and shifter can be fed into a multiplexor. The output of this multiplexor could then go to the output. The decision as to whether to select the adder or shifter output can be based on the current instruction and

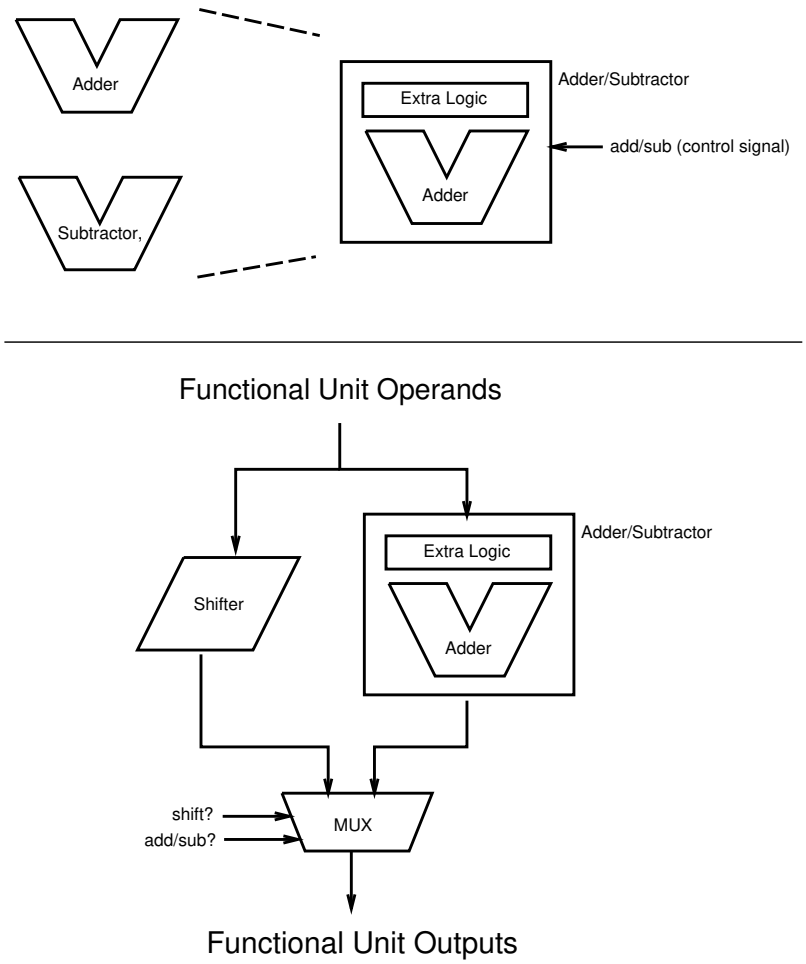


Figure 2-2: Intermediate Architecture of Example Unit - Sharing

be conveyed to the mux as a control signal. Therefore, this step reduces the total number of modules needed since sharing can occur. This step also helps with the determination of what control signals are going to be needed. See Figure 2-2 for the new architecture with latches.

The final step in coming up with a high level design is to decide on latch placement. To do this, a good idea of the delay through each of the modules in the unit is needed, so this step may be put off until such estimates exist. Generally, the idea is to place the n and p latches in places such that the delay through the logic between each set of latches is about equal. Another factor to keep in mind in deciding where to place the

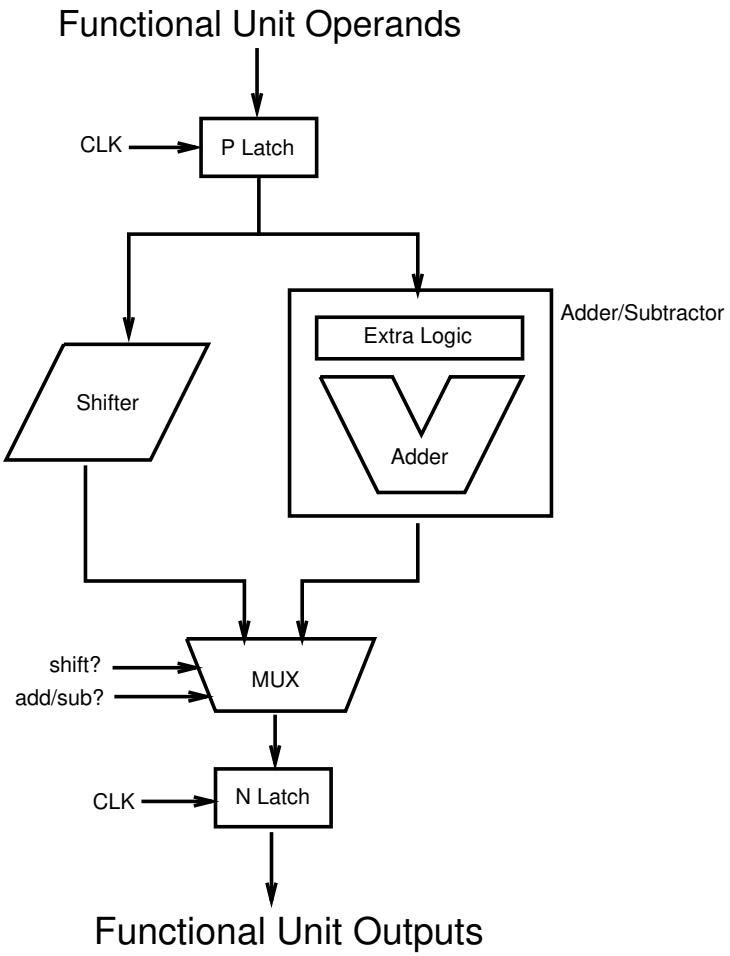


Figure 2-3: Architecture of Example Unit - Latches

latches is the number of bits that need to be latched at each stage. Latches should be placed in locations such that fewer bits need to be latched, thus decreasing the actual number of one bit latches that will be needed. Therefore, there may be a tradeoff between the number of bits to be latched and the amount of logic between latches. See Figure 2-3 for the architecture with latches place such that the number of bits that need to be latched is minimized.

After going through these steps, a high level architectural diagram of the unit being designed should be possible to draw. The diagram should show major modules such as adders and shifters, control modules such as muxes, wires between these

modules, and control signals needed for these modules, just as in Figure 2-3. The next step is to design and implement each of the modules within the unit.

2.2 Module Design and Implementation

At this point, each module's interface has been defined. The next step is to design and implement each of these modules. Depending on the size of the module in question, further division of the module into submodules may be needed, requiring a procedure similar to the one in the previous section. If the module is simple, such as a mux or latch, implementation is probably the next step. Implementation means the building of circuits at the transistor level.

In designing a module, there are certain things that must be known about the module. These include the inputs and outputs to the module, the speed at which the module must run, and the function that the module is supposed to implement. This information should be enough information to allow for the module to be implemented such that it works correctly and executes at the correct speed.

However, knowing some other things about the "environment" in which the module will be placed can allow for a more "efficient" implementation in which the circuit may occupy less space, the outputs may have better rise and fall times, and the circuit may dissipate less power. One such factor is knowing the load that this module will have to drive. This is important for a few reasons. The larger the load that needs to be driven, the larger the drive on the output signal has to be in order to have reasonable rise and fall times. Also, the larger the drive needed on the output, the larger the number of stages of buffering that are needed on the output to achieve the desired drive. Since propagating a signal through buffers takes time, the time to carry out the rest of the function is actually reduced as output load goes up. Knowing this information will help save future design iterations. Knowing the load on the output implies that the module to which the output goes has already been designed and/or the length of the output wires are known. For this reason, it is often a good idea to design the module at the end of the unit first and then work backwards towards the

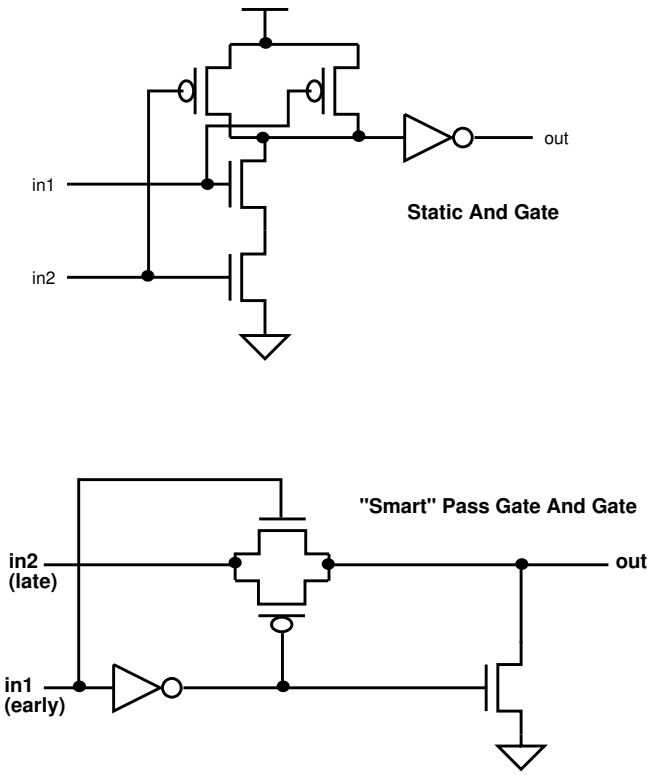


Figure 2-4: Comparison of Two *And* Gates

front of the unit. In addition, since large output loads means more buffering in the module being designed, it is often advisable to keep the input load low such that the module that drives this one does not have to do much buffering.

Another factor that is helpful to know which can lead to a "better" circuit is the time at which input signals will be valid. If it is known that some signal will arrive earlier than some others, than it may be possible to do some "precomputation" based on the signals that arrive early. When the late arriving signals finally are valid, there could then be less logic for those signals to propagate through, thereby possibly leading to a faster module. For example, consider a 2-input *and* gate which consists of a *nand* gate and an inverter. A static version of the gate could be used if it is not known when the inputs arrive. However, if it is known that one input arrives before another, a pass gate version could be used such that the pass gates are set for the late arriving input. The critical path is therefore decreased. This is a very simple

example but can be applied in many places and many ways. See Figure 2-4 for an illustration of these two styles of *and* gates.

2.3 Testing

Once a module has been designed, functional verification should be carried out before sizing transistors. This will save time especially as functional bugs are found that cause the circuit to have to change. After verifying functionality, then transistor sizing can take place. At this point, timing verification needs to take place. The following sections discuss both functional and timing verification techniques.

2.3.1 Functional Testing

The first step in verifying a module is to make sure that functionally the circuit is doing what it is supposed to do, i.e. the outputs of the circuit for each given set of inputs matches what the outputs should be. In doing functional verification, the delay through the module is not the main concern.

For a circuit to be truly functionally verified, every possible set of inputs would need to be run on the circuit. Often this is not practical, such as when trying to verify a 64 bit adder, due to the extremely large number of possible input combinations. In these cases, the following procedure, once completed, should leave the tester reasonably confident that the circuit is functionally correct.

The first step in the functional verification procedure is generally to run hand-crafted test cases through a module, making sure that the module passes these tests. These hand-crafted cases should at the very least test every part of the circuit. This is called glass-box testing, since the tester has to know what is inside the "box" in order to test all parts of it. By testing every part of the circuit, it is meant that for each part of the circuit there is at least one set of inputs such that the output depends on the proper operation of that part of the circuit. In this way, every part of the circuit is "exercised". This kind of testing is often useful for determining such things as where connections are missing, where connections are switched, and where

there are logical errors.

Running hand crafted cases often only tests a very small percentage of all of the possible input combinations. The more input combinations that are tested, the more confident a tester can be that the circuit is logically correct. Therefore, the next step in functionally verifying a circuit is to run a randomized test. Here, a program is written that generates random inputs and feeds them into the circuit. The outputs can be checked versus the results of running the same inputs through a behavioral model of the circuit. If the results differ, then the errors should be logged so the tester can go back to the circuit and figure out the problem. By letting this simulation run for a long time (several days, depending on the number of possible input combinations), many more inputs can be run through the circuit than by doing only glass-box testing. This kind of testing is more of a black-box strategy, since the tester does not necessarily need to know the actual implementation of the circuit in order to test it. In glass-box testing, the tester must know the circuit in order to test each part of it.

Doing glass-box and black-box testing will not guarantee that a circuit is functionally correct if all possible inputs are not tested. However, it will leave the tester more confident of the circuit. This confidence level goes up as more testing is done.

2.3.2 Timing Verification

Having verified functionality of a circuit, the next steps are to size the circuit and run timing tests to see that the circuit meets the timing constraints that have been placed on it. The timing tests for the integer unit were run using HSPICE. HSPICE is a tool which basically takes in a netlist representing the circuit to be tested, a model representing the process to be used, and a stimulus file which contains the inputs to be run through the circuit. The output can be viewed graphically using a tool such as GSI. The output, when viewed using GSI, consists of the waveforms at various nodes in the circuit, including the inputs and outputs. From these waveforms, the delay through a circuit to various nodes and the rise and fall times of signals at various nodes in the circuit can be determined.

Since simulations using HSPICE often take a relatively long time to run as compared to simulations for functionality using something like Verilog, a significantly smaller number of timing simulations can be run. It is important to simulate each path through a module. If that is not practical, then it is more important to simulate those paths which potentially may have the longest delays. These long paths can often be determined from looking at the circuit schematics.

Since these simulations should indicate what the delay through the circuit will actually be when the chip is fabricated, it is important that the simulation take into account such things as wire length, parasitics, and output loads. In order to estimate these, the tester needs to have an idea of where the module being tested will be sitting in the overall floorplan and how large the various modules will be. The tester also needs to know what the outputs will be driving. From this information, wire lengths and loading can be estimated and added into the stimulus file. Without adding this information, simulations would underestimate the amount of delay through a module. It is generally better to overestimate, as opposed to underestimate, loads, thus leading to overestimated delays. That way, if the circuit can meet the timing constraints with the overestimate, it can definitely meet the timing constraints when the actual fabricated circuit is used. However, if underestimated loads are used in the simulation resulting in underestimated delays, it is possible that the fabricated circuit may not meet the timing constraints.

Therefore, timing verification consists of determining wire, gate, and parasitic loads, adding these numbers into the spice deck, simulating the critical paths through the circuit, checking to make sure the delays meet timing constraints, and making sure that the rise and fall times of all signals are reasonable. If the timing numbers are not reasonable, that may mean going back and resizing the circuits and then resimulating again. If it is determined that the timing constraints cannot be met using the current design, then that may entail redesigning, reverification of functionality, and reverification of timing.

Once satisfied with the verification process, the circuit is layed out. On completing layout, more precise numbers for wire, gate, and parasitic loads become available.

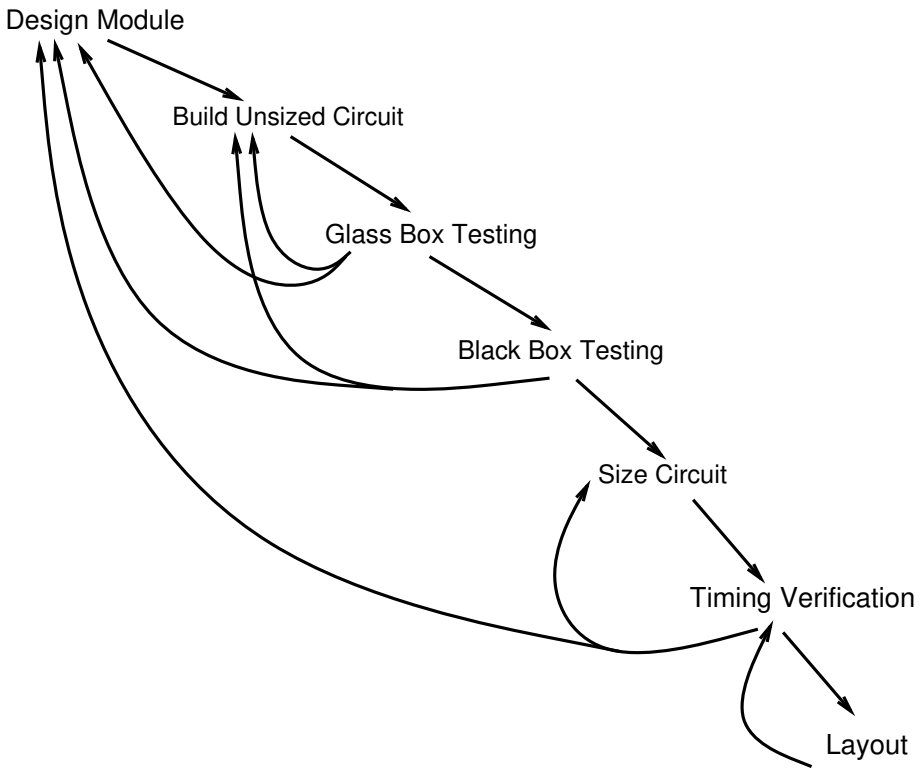


Figure 2-5: Design and Testing Flow

These numbers should be corrected in the spice deck that was previously created, and timing verification should be redone to make sure that the circuit is still operating correctly and is still fast enough to meet constraints.

See Figure 2-5 for an illustration of the design and testing flow. As the figure shows, there may be several iterations of many of the steps.

2.3.3 Variations in Verification Styles

Due to the possibility of having to redesign and reverify functionality after finding that the circuit is not meeting timing constraints, it may seem that a lot of needless time was spent in functional verification before timing verification was even reached. For this reason, it may be more reasonable to follow a different procedure in testing a module.

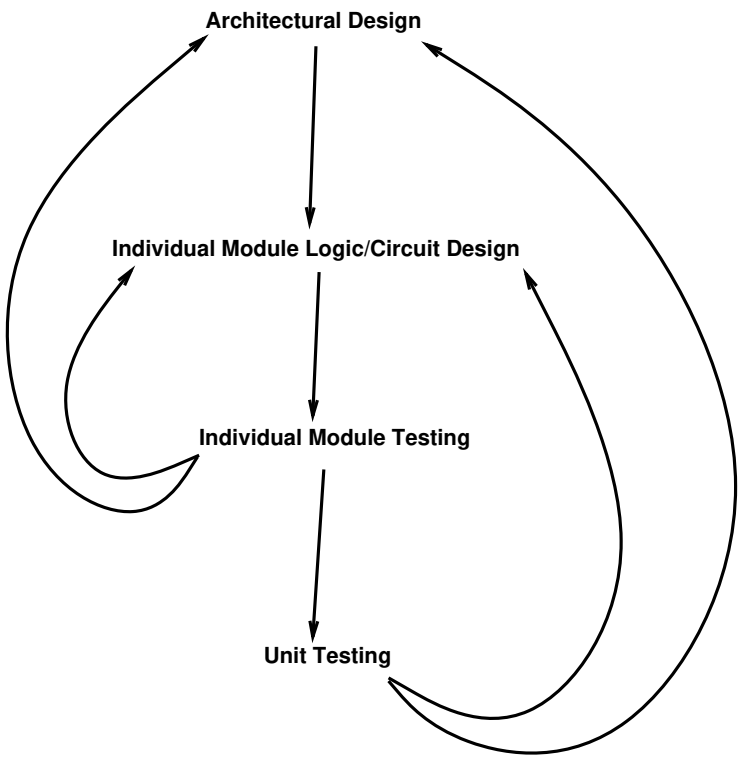


Figure 2-6: Integer Unit Design Flow

This better procedure would be to do glass-box testing first, as before, and then move on to timing tests after sizing the circuits. At this point, if it seems that timing will not be a problem, then the functional verification can be continued at a more thorough level in parallel with the timing verification. This may save time since a more unreasonable design from a timing viewpoint can be detected earlier, thereby saving the time it would take to do the full functional verification.

2.4 Summary

In summary, the design of the integer unit datapath began with design of the overall architecture. In this step, the necessary modules and interconnections were determined. Next, design of each module was done, determining the kind of logic each module would require, followed by circuit design of each module. Finally came

testing of each individual module, testing of groups of modules, and then testing of the whole integer unit. See Figure 2-6 for an illustration of the design flow.

Chapter 3

Integer Unit Architecture

This chapter presents the high level design of the integer unit. It develops the architecture by going through the ISA and then determining what modules are needed, what modules can be shared, and what control signals are required. At that point, the high level architecture of the integer unit datapath will be complete. The first section presents a high level specification for the integer unit.

3.1 Integer Unit Specifications

The integer unit is one of three functional units in each cluster of the MAP chip. The integer unit is part of the execution pipeline stage, which follows the synchronization pipeline stage and precedes the writeback pipeline stage. Since the chip has a 100 MHz clock, the integer unit has 10ns to do whatever it has to do[1]. The integer unit datapath takes in two source operands and the instruction pointer from the synchronization stage. It outputs the result to the writeback stage, whcih in turn sends the result to the integer register file. The result and source operands are 64 bit two's complement numbers.

The addresses are actually 54 bits wide with the remaining 10 bits having special meanings. The top four bits indicate what kind of pointer the address is. The next

[1]If the integer unit needs more time, it may be able to borrow time from an adjacent pipeline stage

six bits are used to indicate the size of the memory segment containing that address. These six bits are used for such things as doing segment boundary violations for address calculation.

3.2 The Major Modules

In this section, the ISA will be used to determine what major modules will be needed in the integer unit. See the appendix for a complete listing of the integer unit instruction set.

The first batch of instructions in the integer unit ISA is the arithmetic operations. These operations include *add*, *add unsigned*, *subtract*, *subtract unsigned*, *arithmetic shift*, *logical shift*, *rotate*, *and*, *or*, *xor*, and *not*. The *add* operations imply the need for an adder that takes in two 64-bit inputs and outputs the sum to the writeback stage. The adder also outputs some overflow information that should go to control. The *subtract* operations imply the same thing as the *add* operations, except we need a subtractor instead of an adder. Recognizing that a subtractor can be implemented as an adder that inverts its second operand and adds one to the output, the adder can be used to serve as a subtractor as well. This is possible when the operands are two's complement values. Therefore the adder would need a control signal to indicate when to subtract. When this signal is asserted, the adder should invert the second operand and set an internal carryin to high.

The *shift* and *rotate* operations imply the need for some kind of a shifter that takes in one 64-bit input along with control information about how much to shift, in what direction to shift, and whether to wrap around in the case of *rotate*. It outputs a new 64-bit value that should go on to the writeback stage.

The boolean operations imply the need for some sort of boolean unit that takes in two 64 bit operands and control information representing the function to carry out. The output would be sent on to the writeback pipeline stage.

The next batch of instructions are the byte manipulation operations. These include *extract byte*, *extract halfword*, *insert byte*, and *insert halfword*. The *extract*

operations consist of taking a particular byte or halfword from one source and putting it in the low byte or halfword of the result. This implies the need for shifting to move the byte or halfword to the appropriate place in the result and a need for some sort of selector to pass either the byte, halfword, or zero. The *insert* operatons consist of moving the byte or halfword in the low bits of one of the sources to a particular location in the other source. Again, this implies the need for a shifter and selector. The shifter can be the same as that used for the *shift* operations. The selector can be placed at the output of the shifter. It would need such control signals as whether to pass the shifter output, the source input, or zero depending on the operation being performed.

The comparison operations compare the two operands checking for such things as equality, greater than, or less than. A way to do comparison operations is to subtract the two operands and run the results through a zero detector. The result of the zero detector in combination with the overflow information that comes from the adder/subtractor can then be used in control to decide the result of the comparison. Therefore, comparison operations require an adder/subtractor, which was already needed, along with a zero detector that takes in the result of the adder/subtractor. The output of the zero detector would go to control. No additional control inputs are needed for the adder, and no control inputs are needed for the zero detect. The result of the comparison operation, which will come from control, needs a path to the output of the integer unit. This result will go to a condition code register and will be referred to as *cc*.

The next class of instructions is the data movement instructions. The integer unit does not have to do much with these instructions except to pass values from the input to the output. Therefore, no special modules are needed to handle these instructions. There just has to be a way to pass inputs straight through the integer unit unchanged. This can be done in many ways such as passing the input through the shifter unchanged or passing the input through the boolean unit unchanged.

The only control flow operation that requires any kind of computation is the *branch* instruction, whereby *IP* and *src2* have to be added together. The adder that

is already needed for other operations can again be reused. However, now since the operands for the adder depend on what instruction is being executed, a multiplexor will be needed in front of the first input to the adder in order to select between the *IP* and *src1*. A control signal for the multiplexor will select the proper operand based on the instruction.

The address calculation operations include *lea*, *leab*, and some pointer operations which write condition codes. The *lea* operation consists of adding an immediate to an address and making sure that the resulting address is within the same segment as the original address. This implies the need for an adder and for some way to check that the new and old address are within the same segment. One way to check this is to generate some kind of mask that indicates which bits of the old and new address must be the same, based on the 6 segment size bits from the original address. This mask, in conjunction with the old and new address can then be used to determine if there is a segment violation. Therefore, an adder, mask generator, and segment checker are needed. The adder already exists.

The *leab* instruction is similar to *lea* except that an immediate is added to the segment base of a given address. Then a segment check is done on the new address. Therefore, some way is needed to generate the segment base. This can be done by masking out the bits from the original address that are allowed to change and still result in an address that is in the same segment. Having done that, the integer to be added to the base can be logically or'd with the segment base, thus resulting in the new address. To do the segment check, a mask, old address, and new address is needed. The mask can be generated by the mask generator, the new address would be the immediate to add to the segment base, and the old address could be zero. If any bit in the new address outside the mask area is not zero, the segment checker would detect this and signal an error. Therefore, this instruction requires two boolean units, a mask generator, and a segment checker. However, the address inputs to the segment checker for *leab* are different that those for *lea*, so muxes would be needed at the address inputs to the segment checker. Also, a path would be needed from the mask generator to the first boolean unit. The other input to the first boolean unit

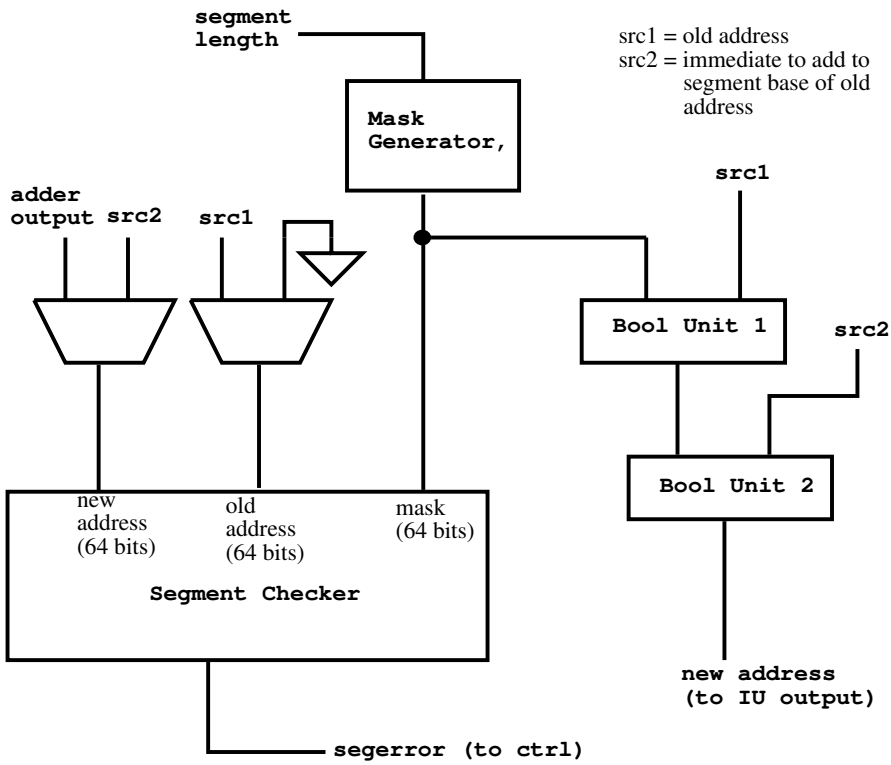


Figure 3-1: Modules needed to execute *leab*

would be the first source input to the integer unit. The second boolean unit would take in the output of the first boolean unit and the second source input to the integer unit. Boolean operations could still be done by passing the source input through the first boolean unit and carrying out the actual boolean operation in the second boolean unit. See Figure 3-1 for an illustration of the modules needed to execute an *leab* instruction.

The pointer operations in the address calculation class of operations simply require a place for a condition code to enter the datapath and be passed through to the output. No computation module is needed in the datapath for these operations.

There are two immediate operations. The create immediate operation requires a pass through path, which is already needed by some previous instructions. The other operation, called the shift & or unsigned 16-bit immediate, requires one input to be shifted 16 bits and then an immediate to be placed in the low 16 bits. This means a

shifter and some kind of inserting mechanism is required, both of which already exist as a shifter and selector. Therefore, no new modules are needed for the immediate class of operations.

The configuration space operations consists of only the GTLB access instruction. This operation is similar in function to the operations in the communications class of operations, which is the final class of operations. The general idea behind the communication operations is to send a series of values onto the C-switch, which is the switch on which clusters can communicate with each other and on which data from memory is returned to the clusters. The sources of the values to send over the C-switch are both of the integer unit source inputs followed by data off of the writeback line that goes to the writeback stage. The writeback line is actually an input/output line over which data is sent to and from the register file. Therefore, to execute these instruction requires a mechanism for sending these values, one at a time on each clock cycle. This implies muxes and latches, all of which will need to be placed in a separate computation module that takes in the two source operands as well as the writeback line. The output of this "send" unit would need to go to the C-switch.

All of the instructions have now been accounted for. However, the integer unit also needs to take care of passing exception and "errval" identifiers. An "errval" is basically a deferred exception. Generally, an errval is created by passing the IP to the writeback stage, where the IP pointer type has been modified to reflect an errval and some information about the type of errval has been added in the six bits usually associated with segment length. Therefore, a path is needed from the IP to a module which composes the errval to be sent to the output. Finally, exception identifiers are simply 10 bits that also need an entry point to the datapath from control and then a path to the output whenever an exception occurs. However, these 10 bits do not need to be combined with any other piece of data, unlike the errval.

Therefore, the major modules that will be needed include an adder/subtractor, shifter, selector, zero detector, two boolean units, mask generator, segment checker, send unit, and errval generator. See Figure 3-2 for an illustration of this temporary

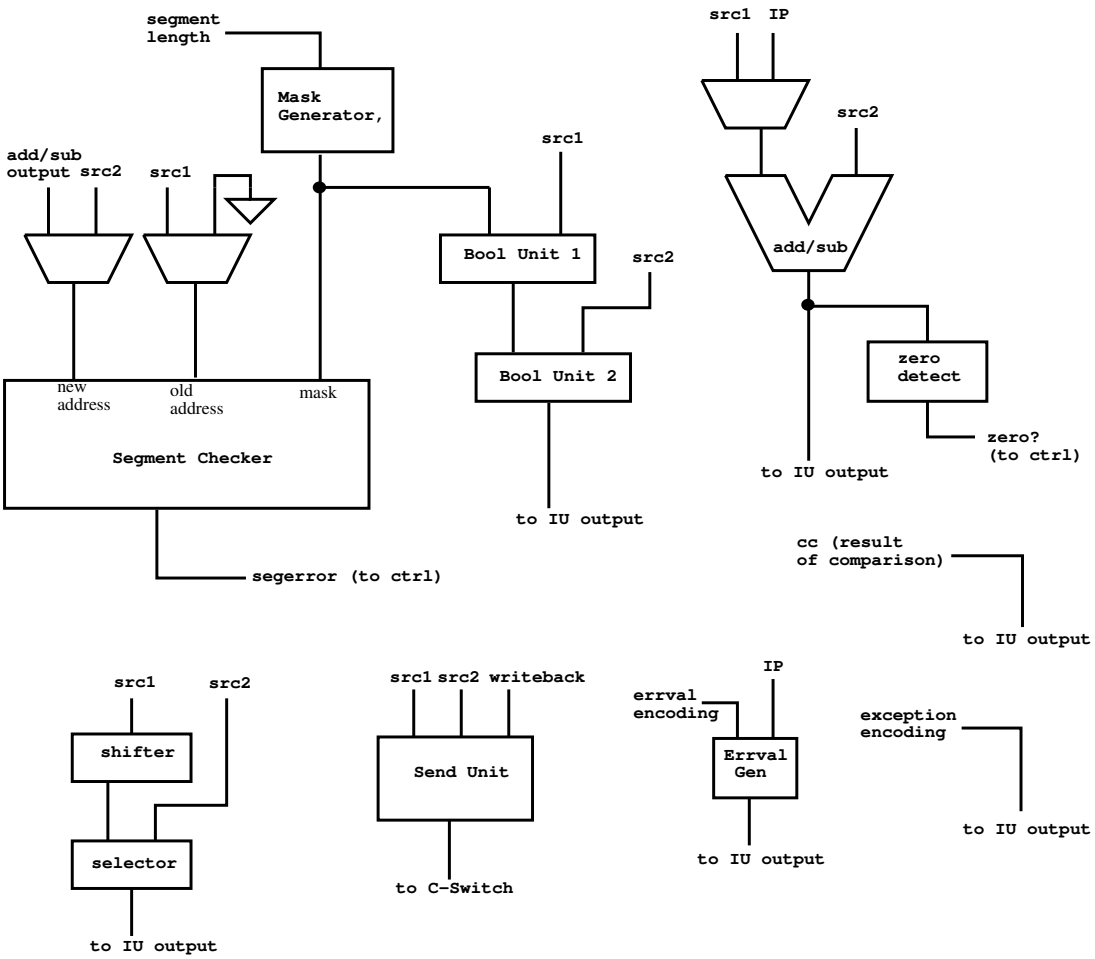


Figure 3-2: Intermediate integer unit architecture

architecture. In the next section, these modules are put together to form the integer unit datapath.

3.3 Putting It All Together

Now that all of the needed major modules have been identified, the next step is to put them together along with muxes, latches, and drivers, in order to form the integer unit datapath.

Many of the major modules share the same inputs, so running the inputs to each of these modules needs to be done. Also, many of these modules want to send their

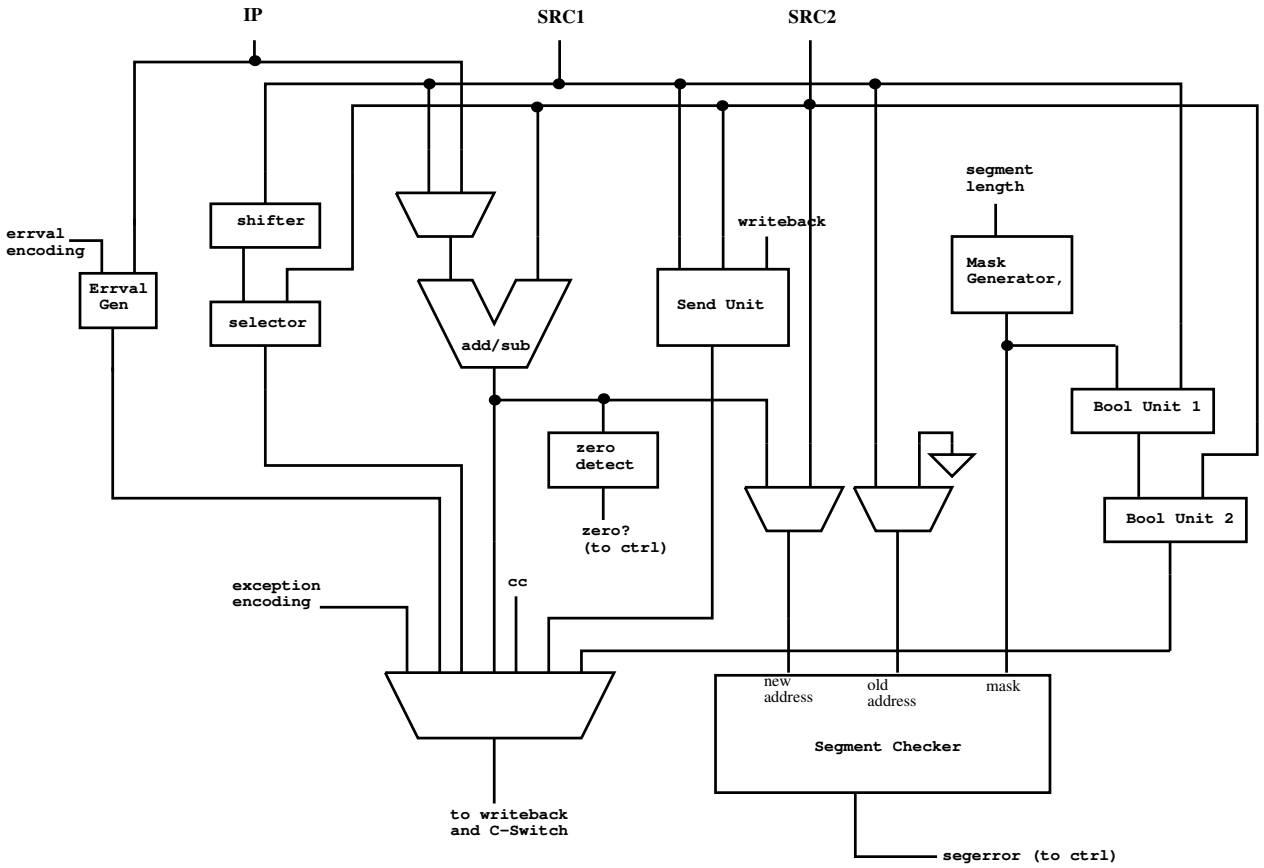


Figure 3-3: Intermediate integer unit architecture with output mux

outputs to the integer unit output. A mux is needed on the output line to select between the various modules that are driving the output. The line to select would be based on things such as the operation being performed and whether there was an exception or an errval to pass. See Figure 3-3 for the updated integer unit architecture.

The results of the integer unit instructions either go to the C-switch or the integer register file. However, the line to the integer register file drives data in both directions, so the module driving the output has to have a tri-state capability. Therefore, a tri-state driver is need to drive the outputs of the integer unit to the register file and C-switch. See Figure 3-4 for the updated architecture.

At this point, the integer unit architecture is nearly complete. However, the n-latches and p-latches need to be added. The strategy here is to place latches such

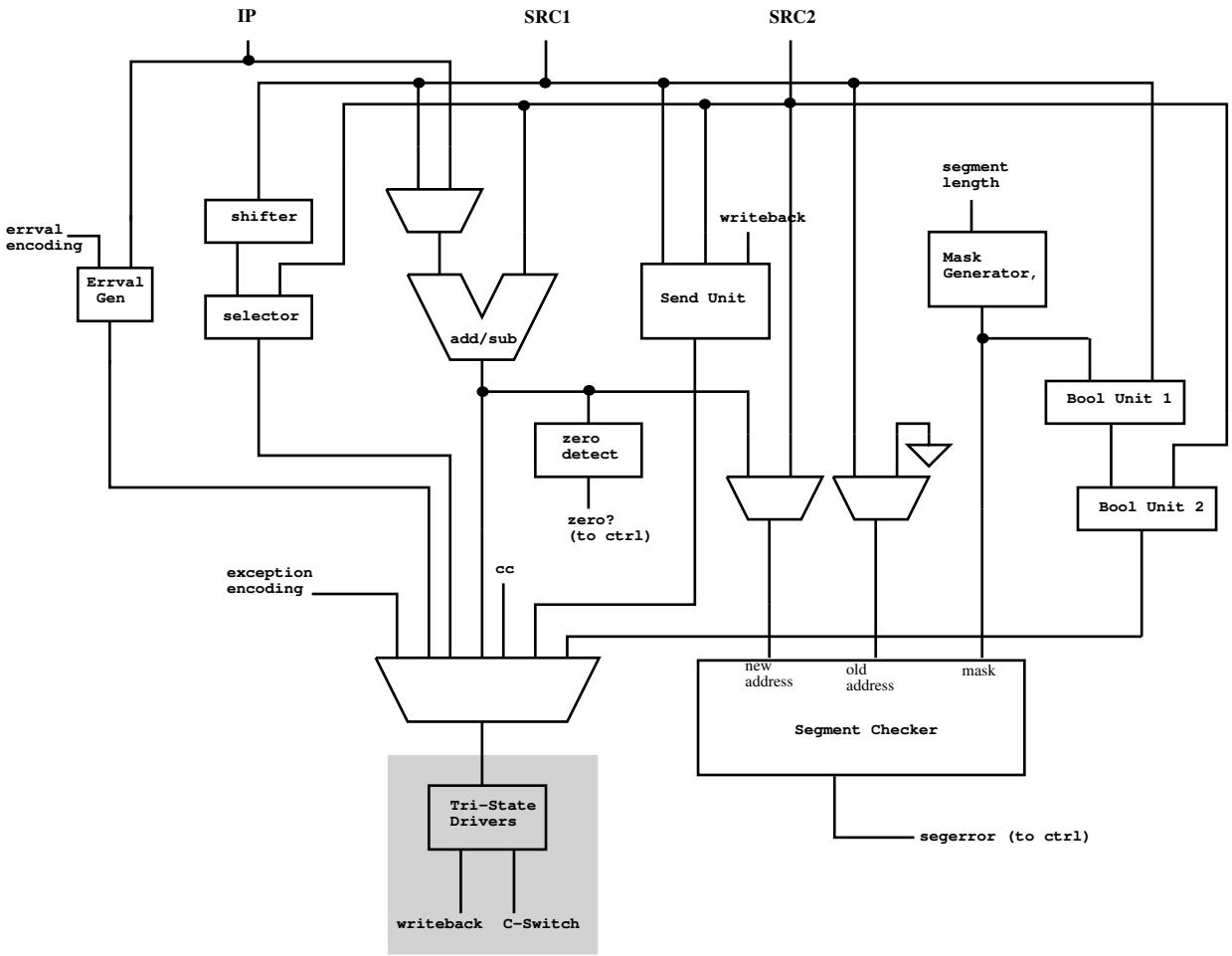


Figure 3-4: Intermediate integer unit architecture with result driver

that few bits have to run across the latch and such that the amount of logic between latches takes up about half of a clock cycle. Without knowing the actual delays through each of the major modules and looking more at having a low number of bits needing to be latched, it appears that the best latch placement would be p-latches at the inputs to the integer unit and n-latches after the mux that selects the output to go to the driver. This will probably result in a larger amount of logic between the p-latches and n-latches than between the n-latches and the next set of p-latches in the next pipeline stage. However, the amount of logic between the n-latches and p-latches will hopefully have a delay that is low enough such that there is no timing

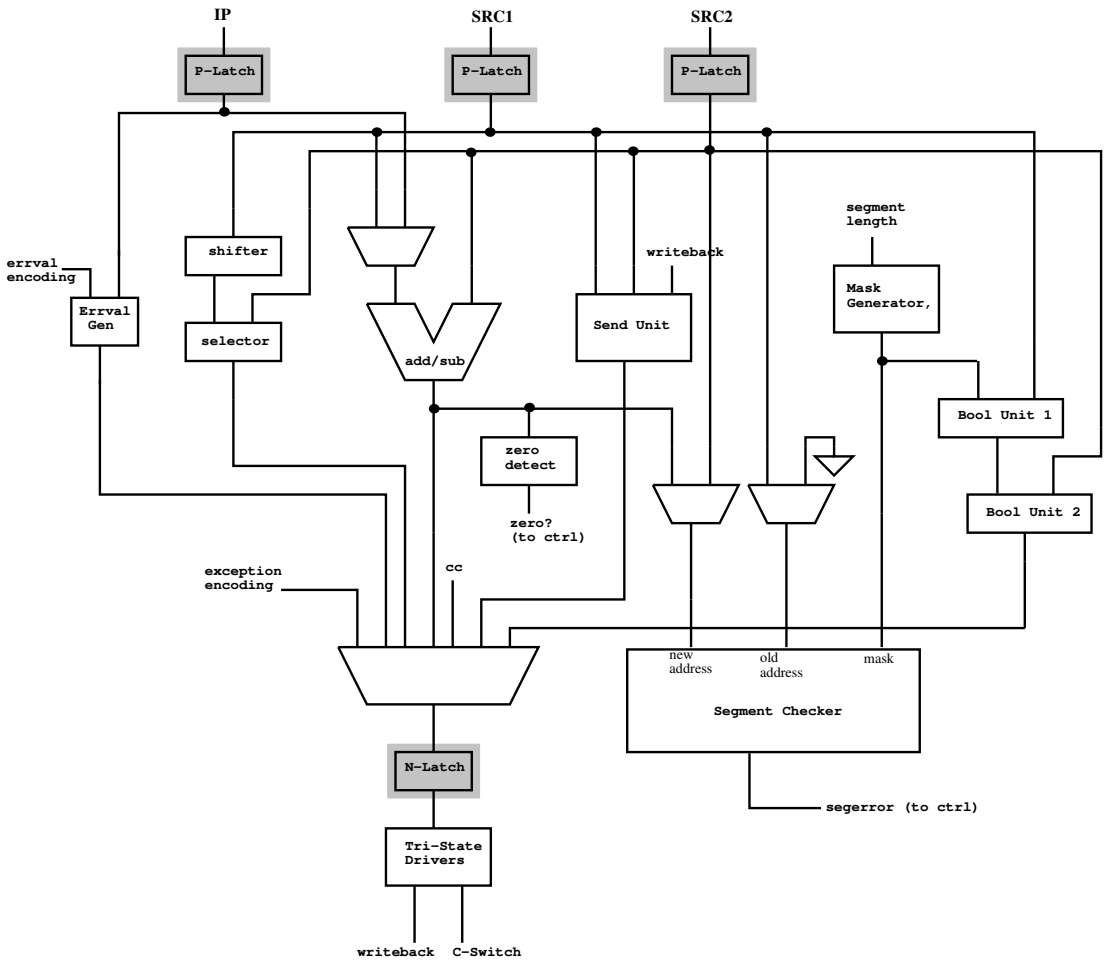


Figure 3-5: Intermediate integer unit architecture with latches

issues here. If not, then some time can be borrowed from the logic that follows the n-latch. See Figure 3-5 for an illustration of the latch placement in the integer unit.

One final issue has to do with the timing of deciding when to pass an errval. In some cases, the decision to pass an errval may not occur until after the second half of the clock cycle. It would probably be beneficial to place the errval as close to the output as possible such that it has to propagate through less logic in the case that it is selected. This can be done by placing a 2-input multiplexor between the n-latch and the driver, where the two inputs would be the latched output of the previous multiplexor and the errval line. Since the errval has no n-latch in its path, another

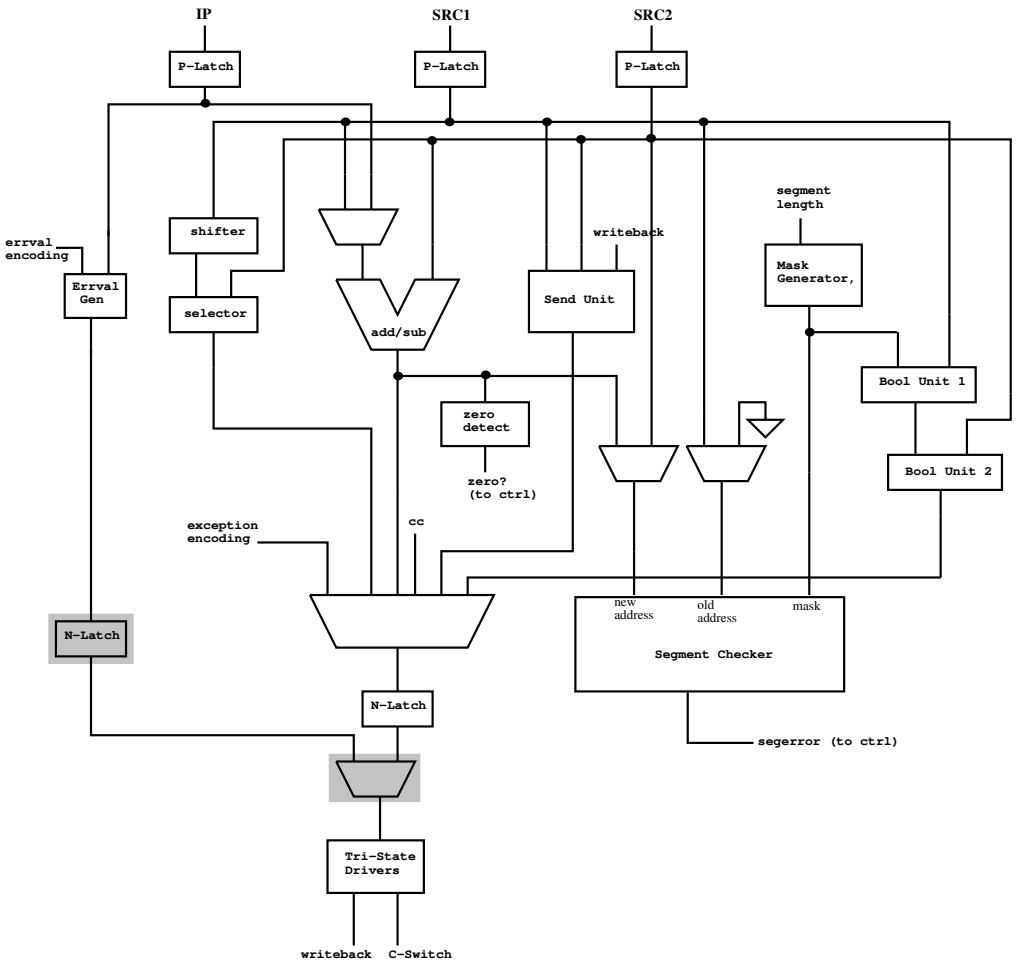


Figure 3-6: Final integer unit architecture

n-latch would have to be added on the errval line. At this point, the integer unit datapath architecture is complete. See Figure 3-6 for the integer unit architecture.

Chapter 4

Integer Unit Submodules

The following sections provide descriptions of each of the major submodules within the integer unit. The purpose, design, and implementation of each module is presented. Also, any issues/tradeoffs that led to a particular implementation are discussed.

4.1 Adder/Subtractor

The adder/subtractor that has been implemented is a hybrid carry lookahead/carry select adder. It takes in two 64-bit operands, along with *sub* and *carryin*, which are control signals. In the rest of this section, *a* and *b* will refer to the two operands. The output is another 64-bit value, which will be referred to as *out*, along with two bits which go to control. These two bits are called *gc8* and *gc7* and are used by the integer unit control to detect overflow.

In order to understand how the adder works, first consider the addition of two one-bit numbers with a *carryin* input and a *carryout* output. Without knowing what the *carryin* signal is, it is known that *carryout* will be 1 if both operands are 1, *carryout* will be 0 if both operands are 0, and *carryout* will equal *carryin* if one of the operands is 1 and the other 0. These three cases are called *generate*, *kill*, and *propagate*, respectively and are easily computed through use of an and gate, nor gate, and xor gate, respectively.

Knowing *generate*, *kill*, and *propagate* is helpful when *carryin* is a late arriving

signal, since it is possible to set up a chain structure that allows *carryin* to propagate up through the chain for each pair of bits very quickly. This in turn allows for quick *sum* calculations once *carryin* is known. To calculate the *sum* bit, all that needs to be done is to xor the *carryin* and *propagate* bits for each pair of input bits.

A very slow adder could be built which creates a chain going across all 64 bits. This would be slow since *carryin* would potentially have to propagate through at least 64 transistors. The adder implemented in the integer unit instead does some speculative computation of sum bits and then selects the appropriate results, such that the longest chain that any *carryin* has to propagate through is a chain of 8 transistors.

Basically, after computing *generate*, *kill*, and *propagate* for each pair of bits, *generate*, *kill*, and *propagate* are computed for each of 8 groups of 8 bits. From these group signals, group carries can be computed for each group of 8 bits. While the computation of these group signals is going on, local carries for each of the 64 pairs of bits are computed in groups of 8 bits assuming *carryin* of both 0 and 1, thus resulting in 128 local carry signals. These 128 signals are then xor'd with the corresponding *propagate* bits to generate 128 *sum* bits. The proper 64 bits to output are chosen based on the group carry signals, since these indicate whether the correct *carryin* for each local group of 8 bits was a 0 or 1.

See Figure 4-1 for a block diagram of the adder architecture. The pgkblock computes 64 sets of *generate*, *kill*, and *propagate* signals, which go to the group pgk block and to the local carry chains. The group pgk block calculates *group generate*, *group kill*, and *group propagate* for each group of 8 bits, sending its results to the global carry chain, which calculates group carries. The local carry chains, of which there are 2 for each group of 8 bits, computes local carry signals assuming carryins of 0 and 1. The outputs of both the global carry chain and local carry chains, along with individual *propagate* signals, go to the final cell where sum bits are computed and then selected based on the group carries.

The worst case delay through the adder is about 3.7ns. This occurs when *carryin* is 0 and each pair of bits being added together has a 0 and a 1. The following sections

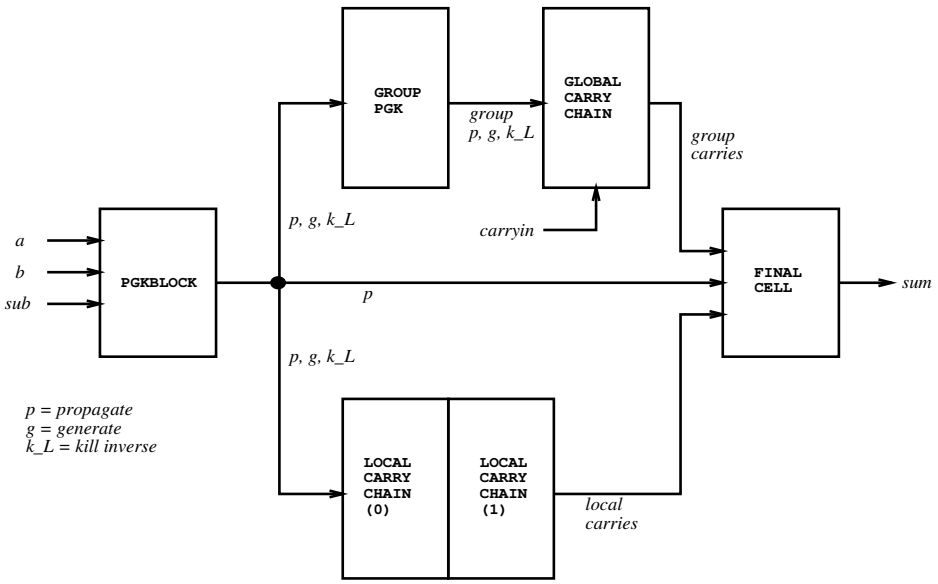


Figure 4-1: Block diagram of adder internals

describe each of the modules in detail.

4.1.1 PGK Block

The pgk block is the module into which the 64-bit operands flow. It also takes in the *sub* control signal. Recall, when doing a subtract, b needs to be inverted and *carryin* needs to be set to 1. The pgk block does the necessary inversion in the case where a subtract is being done. The purpose of the pgk block is to compute *propagate*, *generate*, and *kill_L*, for each of the 64 pairs of input bits as follows:

$$propagate = a \oplus bnew$$

$$generate = a \& bnew$$

$$kill\_L = a | bnew$$

where $bnew$ is equal to b if a normal add is being done, else $bnew$ is equal to the inverse of b in the case of a subtract operation. See figure 4-2 for a block diagram of

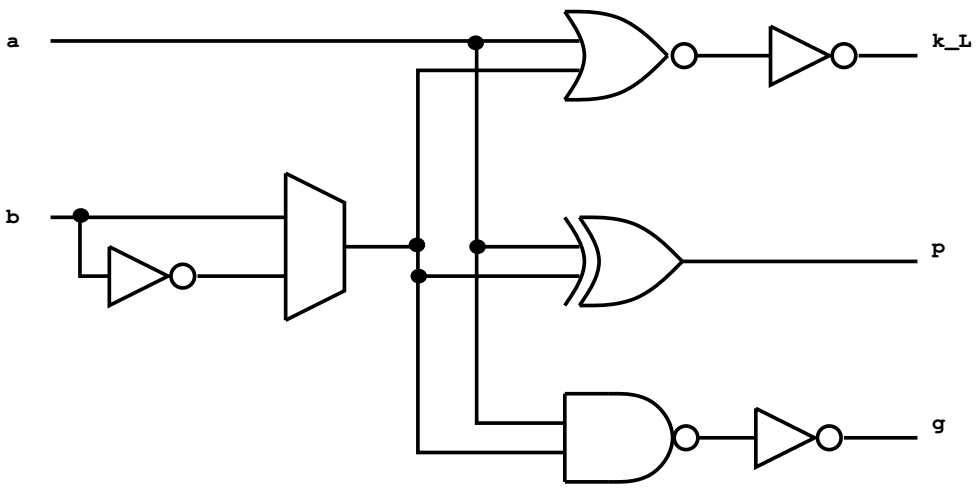


Figure 4-2: Block diagram of pgk block internals

the pgk block. The adder contains 64 instances of this module, one module for each pair of bits.

Of the three operations that need to be done in this block (*xor, and, or*), *xor* is the most time consuming and is therefore on the critical path through the pgk block. Therefore, different xor gates were looked at before one was selected. The options that were tested include a static branch based version and a pass gate version. Additionally, one other experiment was tried for evaluation of *propagate*. By noting that $propagate = generate \& kill\_L$, the *xor* gate can be eliminated in favor of an *and* gate.

See Figure 4-3 for the schematic of the pass gate version of the pgkblock. Note that each transistor has its relative width printed next to it in this and all schematics. The pass gate version consists of two n-only pass gates which pass a or $\hat{a}$ depending on the value of b. The value of b will be known earlier than that of a since there is a mux in front of the adder on the a input, as can be seen in the integer unit architecture. This knowledge is useful since b can then be used to select the pass gate such that the pass gates will be ready to pass the a input when it becomes valid. Following the pass gates is a low threshold inverter to equalize the rise and fall times on the *propagate* output. The worst case delay through this version of pgk block is 700ps.

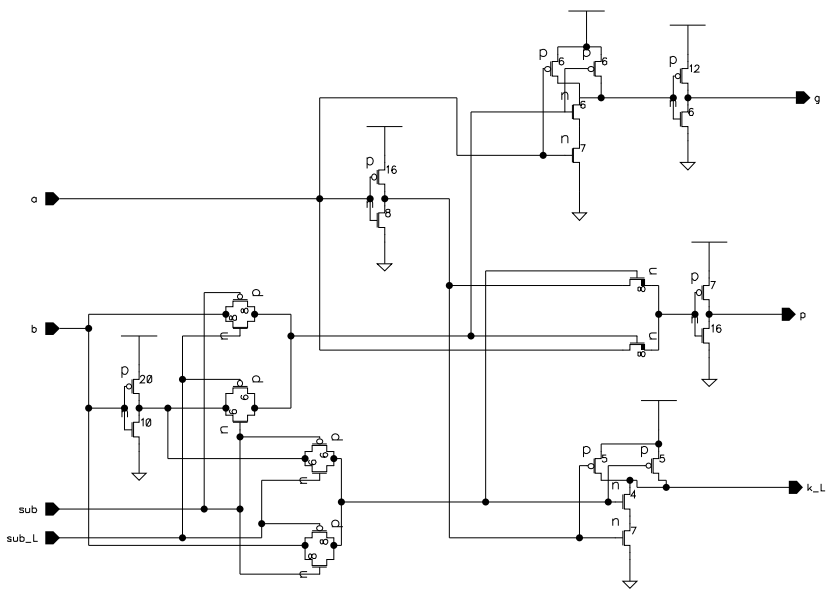


Figure 4-3: Schematic of pass gate version of pgk block

This occurs when a and sub are high with b going from high to low.

See Figure 4-4 for the schematic of the static gate version of the pgkblock. The static gate version of an *xor* consists of 8 transistors, requiring both senses of a and b at the inputs to the *xor* gate. In the worst case, the output needs to be pulled up through two PFETs. The worst case delay through the pgk block using this gate was 760ps. This happens when a is low, sub is high, and b transitions from low to high. This circuit takes up more area than the pass gate version and also has rather large transistors in the static xor gate. This in turn causes the load on the xor input lines to be large compared to the pass gate version. Given that the load on the pgk block inputs is kept the same for both the pass gate and static gate versions of the pgk block, the delay through the static gate version will be larger due to the higher internal load.

See Figure 4-5 for the schematic of the pgkblock which does not use an *xor* gate. The final version, which does not use an *xor* gate, simply gates the *generate* and *kill_L* signals together to derive *propagate*. The worst case delay through this circuit is 920ps, occurring when a and sub are low with b going from high to low. The nice

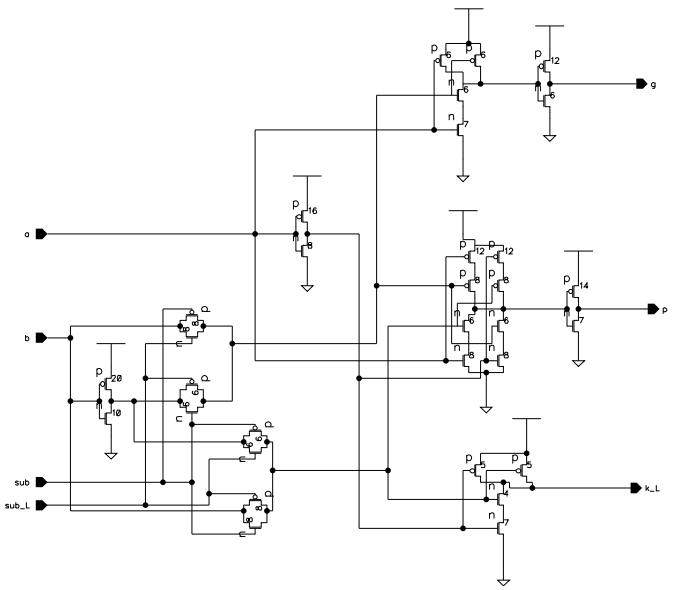


Figure 4-4: Schematic of static gate version of pgk block

thing about this circuit is the reduced load on the internal a and b lines. However, the evaluation of the *propagate* signal cannot begin until *generate* and *kill_L* are known, thus causing the large worst case delay.

From these experiments, it is clear that the best version to use is the pass gate *xor* version. Not only is this the fastest of the three, but also it uses the least area.

4.1.2 Group PGK

The group pgk module is responsible for taking in eight sets of *propagate*, *generate*, and *kill_L* signals, corresponding to a group of 8 bits. It uses these 24 input signals to compute group generate, group propagate, and group kill_L, which will be referred to as *gg*, *gp*, and *gk_L*, respectively. The purpose of computing these signals is to allow for group carries to be computed in the next stage. These group carries will be used to select the proper local carries in the final stage. There are 8 of these modules in the adder, thus accounting for all 64 bits.

When *gp* is asserted, this indicates that the carry output from this set of 8 bits should be whatever the carry input is. This signal should be asserted if all of the

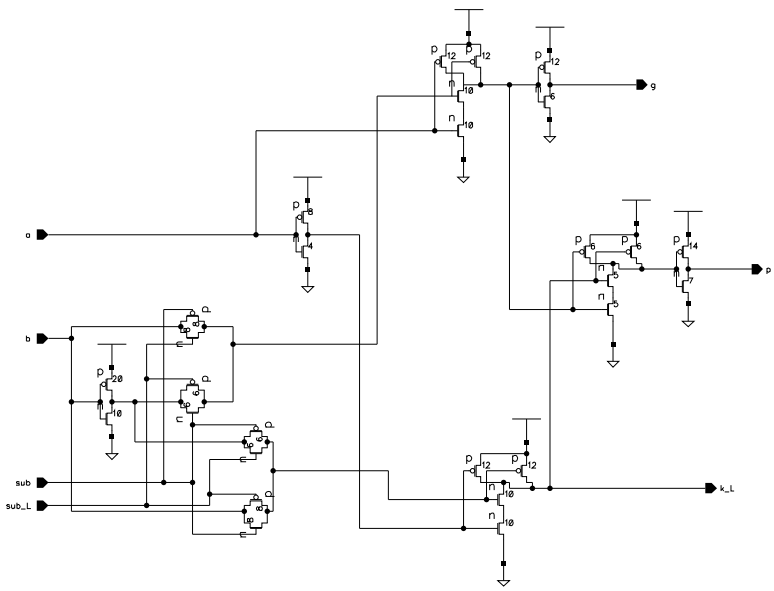


Figure 4-5: Schematic of non-*xor* version of pgk block

8 input *propagate* bits are asserted. This requires an 8 input *and* gate. The actual implementation is two 4-input *nand* gates followed by a 2-input *nor* gate. Using an 8-input gate is not reasonable due to the long chain of transistors that would result. The delay through a long chain of transistors can become very high, and the resulting output waveform can have very bad rise and fall times. In addition, transistor sizes can get very large when trying to achieve reasonable delays.

When *gg* is asserted, this indicates that the carry output for this set of 8 bits should be asserted no matter what the carry input is. The circuit that is used to compute this is a Manchester carry chain. Based on the values of the input *propagate*, *generate*, and *kill_L* signals, *gg* can be computed. Each "link" of the chain basically consists of three transistors, each controlled by one of 8 sets of *propagate*, *generate*, and *kill_L*. The idea is to pass a 0 up to the next "link" in the chain if the carry out for the bit being looked at is a 0, and to pass a 1 if the carry out is a 1. At the top of the chain, if the output is a 1, then that indicates that the carry output for the group is a 1. Therefore, for a particular "link", if *generate* is asserted, then a 1 should be passed on to the next chain. If *kill_L* is asserted, then a 0 should be passed on. Otherwise,

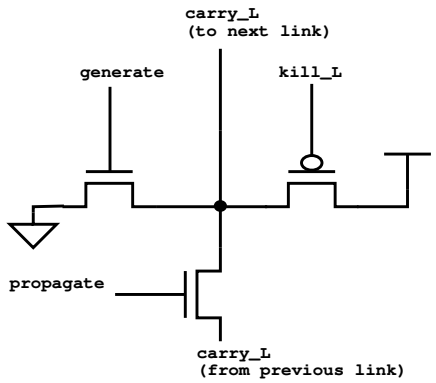


Figure 4-6: Illustration of a typical link in the carry chains

if *propagate* is asserted, then the value being passed from the previous "link" should be passed on to the next link. This indicates a "link" that looks like that shown in Figure 4-6. Note that in this figure, the inverse of carry is passed up the chain, meaning if *generate* is asserted, it would pass a 0, and vice versa for *kill_L*. See the actual schematic later in this section.

Since there are 8 bits in a group, the chain would consist of 8 of these links. Eight of these links together means there would be 8 transistors in series. This may or may not be a problem in terms of the size these transistors would have to be and the speed at which this circuit would run. If it turns out there is a size of speed problem, one thing to try would be putting a buffer in the middle of the chain. Another idea is splitting the chain, making the top part of the chain speculative, and using the result of the bottom part to select the proper results from the top part of the chain.

Before going on to present the different ways for setting up the carry chain, the evaluation of *gk_L* will now be discussed. Actually, all that is need to evaluate *gk_L* is to realize that $gk\_L = gp|gg$. Therefore, *gk_L* is not evaluated until *gp* and *gg* have been determined. However, since *gp* will be ready earlier than *gg* in the worst cases, the gate used to generate *gk_L* is set up to expect *gp* to be early, thus saving some time. This is done by placing the *gg* input to the *nor* gate at the transistors closest to the output of that gate, which is a standard technique for taking advantage of early

arriving signals. See the schematic later in this section for the actual implementation.

The first chain that was tested for evaluation of *gg* was the chain of 8 with no breaks. The sizing of the transistors along the carry chain was initially done based on Elmore delays, where the chain is modeled as an RC network [4]. The result of these calculations indicate that the transistors should be large at the bottom of the chain and smallest at the top of the chain, where the load that has to be driven is much smaller. See Figure 4-7 for the schematic. Simulations indicate a worst case delay through the group pgk using this type of chain to be about 1.5ns, with nearly equal delays through each stage of the chain.

The second style that was tested simply inserts a buffer halfway through the chain. The purpose of doing this is to reduce the load on the bottom parts of the chain. This allows the size of the transistors to get smaller while maintaining the same delay through the circuit. The circuit can even become faster with careful sizing of the transistors and still be smaller than the original version. Inserting a buffer halfway up the top part of the chain means that the signal being propagated up the chain will be the inverse of what it used to be. Therefore, any generates and kills on the top half of the chain should now pass the opposite of what they normally passed. Also, the output *gg* now has to be inverted to maintain the correct sense. The circuit that uses this style has a maximum transistor size that is one-third the size of the largest transistor in the chain without the buffer. Not only that, but also the new circuit is about 100ps faster. Therefore, there is no good reason to use the chain without a buffer. See Figure 4-8 for the schematic.

The third style is similar to the second style in that the chain is split. However, in this third style, the top part of the chain is replicated twice, with one of the chains taking an input at the bottom of the chain of 0, and the other taking in a 1. Also, the logic to evaluate *gk_L* is replicated twice, one for each of the two replicated chains. The results of both of these chains is sent to a multiplexor and to the *gk_L* logic, which itself sends its results to a multiplexor, where the correct result is selected by the value coming out of the bottom part of the carry chain. The idea here is to do parallel computation to make efficient use of time. See Figure 4-9 for the schematic.

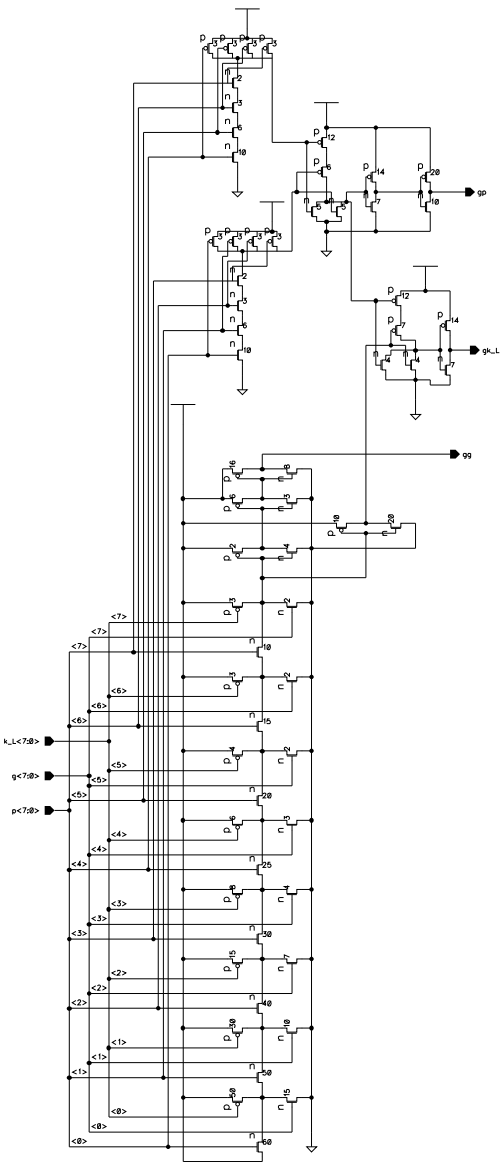


Figure 4-7: Group pgk module with a chain of 8 unbroken links

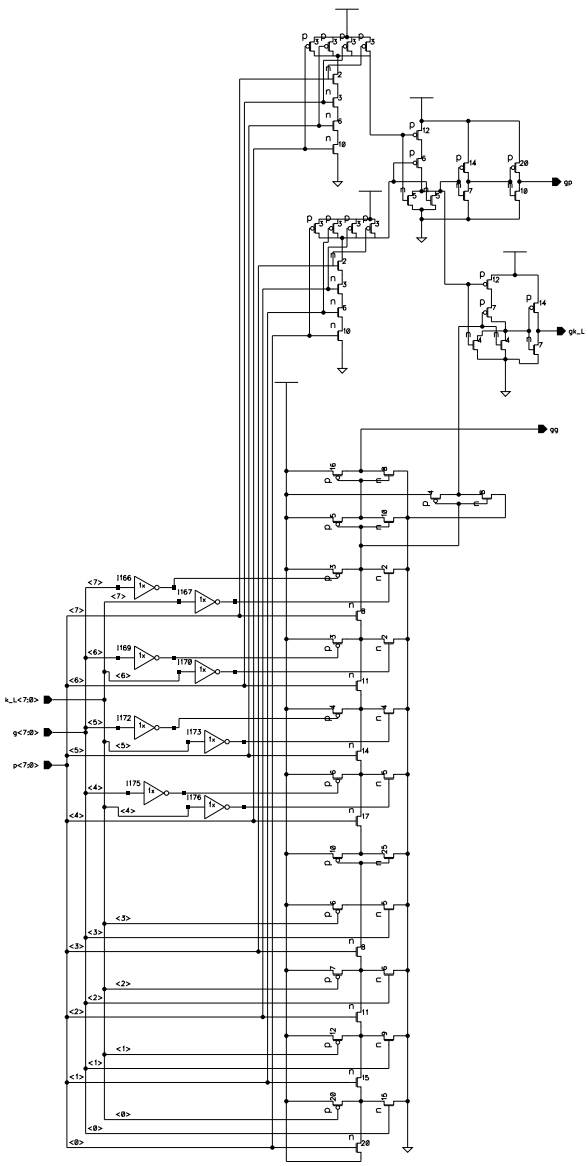


Figure 4-8: Group pgk module with a broken chain of 8 links

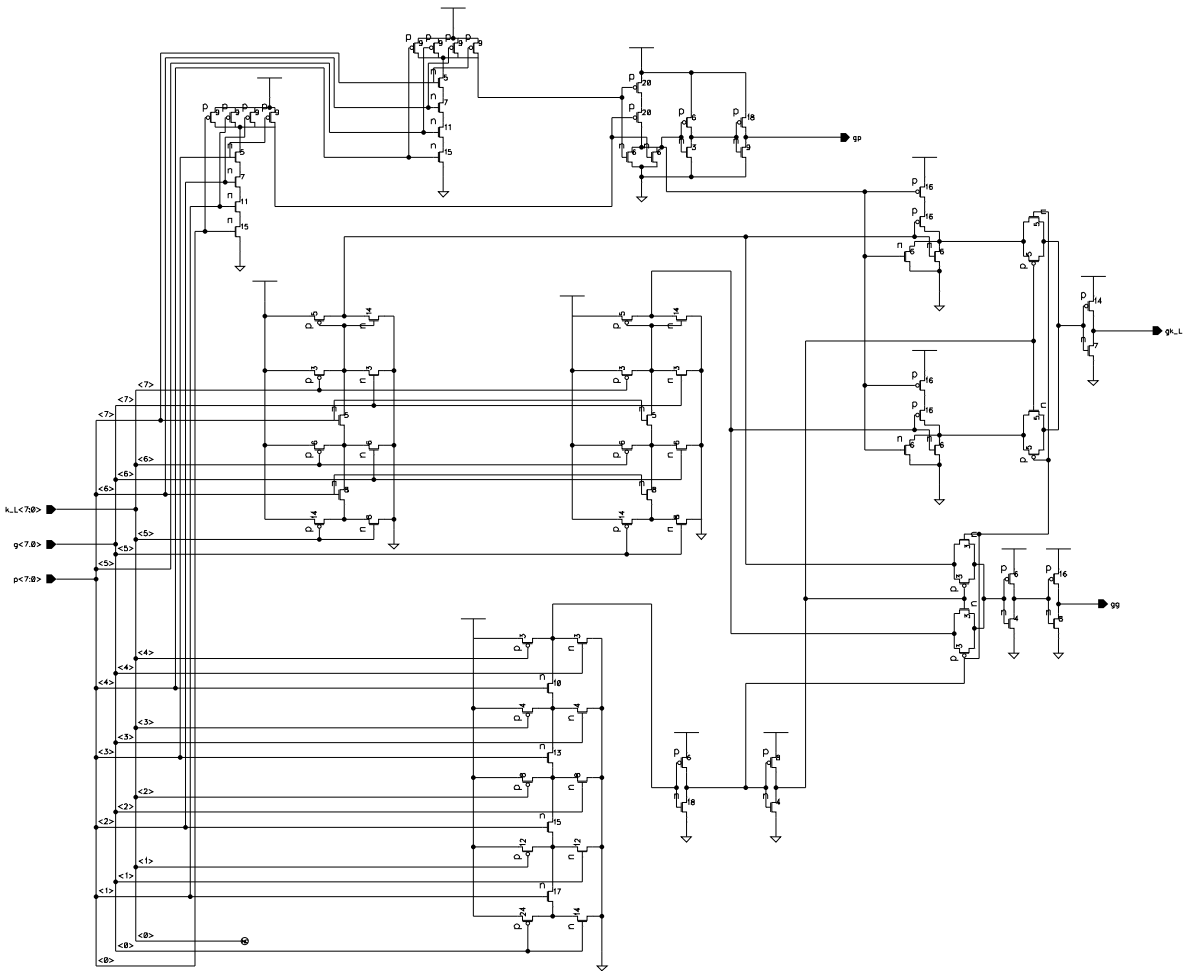


Figure 4-9: Group pgk module with speculative chain

One issue to consider is where to split the chain. It would be nice to have the mux select signal ready at about the same time the mux inputs are ready, that way minimizing the amount of "wasted" time. As it turns out, simulations indicate that this optimal split occurred with a 5-3 split, with the bottom chain consisting of 5 links and the top two speculative chains consisting of 3 links each. As it turns out, the delay through 5 links is a little less than the delay through 3 links and the *gk_L* logic. Simulations shows that the delay through the group pgk is now 1.15ns, which is about a 25-30% speedup over the original chain and about a 20% speedup over the buffered chain. However, due to the replication of the top part of the chain, the area used up by this new circuit is greater than the area of the second circuit. Which one to choose depends on the constraints placed on the adder. For the integer unit, speed was more important. Therefore, this third style is used.

Ideally, the third style should see delay cut in half plus the time through a multiplexor. However, the circuit did not quite achieve that. One reason for this may be the uneven split of the chain. Had there been no *gk_L* logic, then the chain may have been split evenly resulting in delays closer to ideal levels. Another reason for the non-ideal results may be the extra buffering required at the end of the bottom chain in order to drive the signals to the mux selects.

4.1.3 Global Carry Chain

The global carry chain determines the carry out for each group of eight bits. To do this, it takes in group generates, propagates, and kills from the group pgk module. It uses a carry chain, as in the group pgk, to determine the eight carry bits that are the outputs. Another input to the global carry chain is the *carryin* input to the adder. The carry bits which are the outputs of the global carry chain are used in the final cell to determine which of the sum bits from the local carry chains to drive to the output. There is only one instance of this module in the adder.

The global carry chain is very similar to the carry chain used in the group pgk, but not exactly the same. In the group pgk, there was only one output from the top of the chain. In the global carry chain, there will be 8 outputs, one after each "link"

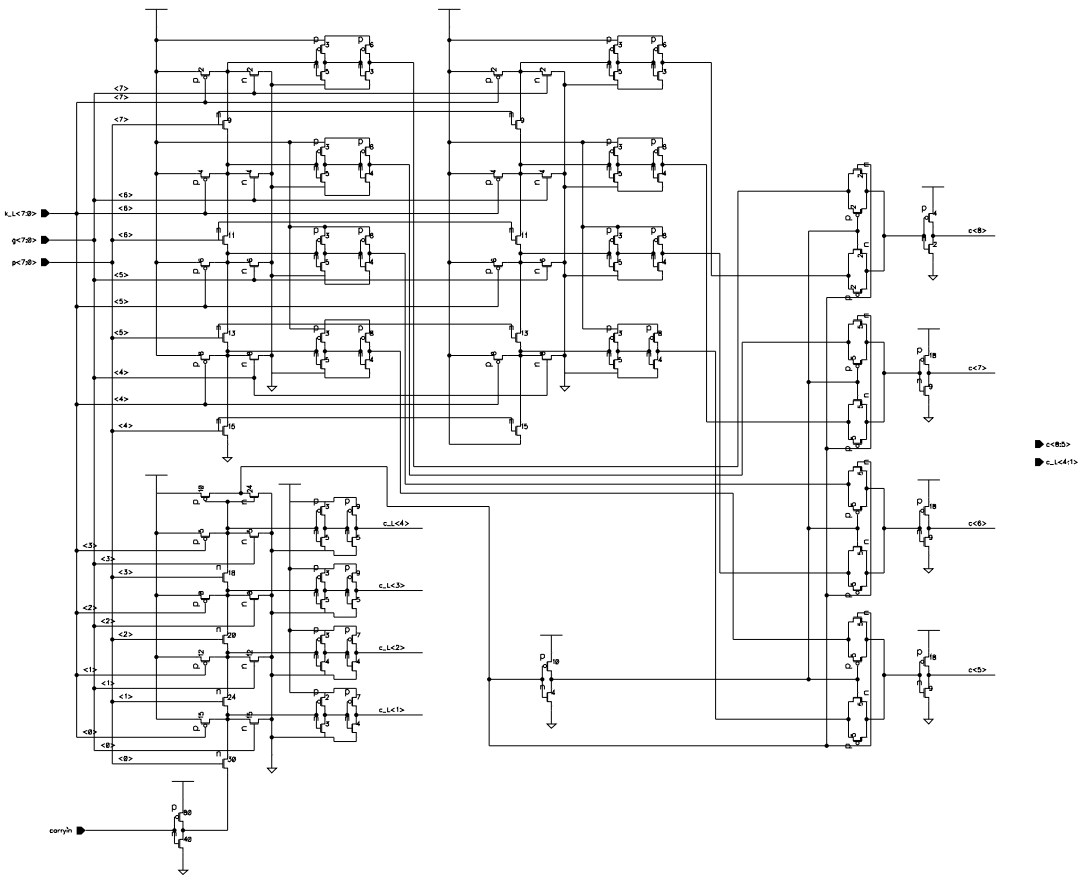


Figure 4-10: Global carry chain schematic

of the chain. This means that there will be additional loads on each of the links of the chain, in turn meaning that the transistors that go up the chain may have to be larger in order to achieve the same delay as that of the chain in the group pgk. One other difference is in the input at the bottom of the carry chain. The global carry chain takes in the *carryin* input to the adder. The group pgk takes in a 0 at the bottom of the chain since the chain is used there to determine whether or not to generate a carry on the output. Other than these differences, the global carry chain will operate exactly the same as the carry chain in the group pgk. Therefore, the issues in design of the chain are basically the same.

One thing to consider in designing the speculative version of the carry chain is where the split should occur. In the global carry chain case, the outputs from each of

the speculative chains do not have to go through any additional logic. They go straight to the mux that will be selected by the output of the bottom part of the carry chain. As it turns out, an even split of the chain results in the best timing. See Figure 4-10 for the schematic. The critical path through the carry chain is propagation of *carryin* up the chain to the mux selects, where the results of the speculative chain are then sent to the output. Note that the low 4 outputs (which come from the bottom part of the chain) are inverse carry signals. The delay through the global carry chain is 1.1ns in the worst case, which occurs when all of the propagates flip at once from low to high and *carryin* is 1. The largest transistor in the chain is 30 microns, which is slightly larger than the biggest link in the carry chain found in the group pgk.

4.1.4 Local Carry Chain

The local carry chains are used to speculatively determine what the carry outs for each of the 64 bit positions will be. The local carry chains, like the global carry chain and the carry chain in the group pgk, are chains of 8 links. Each module contains two chains, one which uses a carry in of 0 and the other which uses a carry in of 1. Each chain takes in the *propagate*, *generate*, and *kill_L* signals from the pgk block. Each chain outputs 8 carry out signals. Since the real carry in is not known until the results of the global carry chain are ready, by precomputing carry outs for both possible values of carry in, all that is needed to decide the correct carry out is a mux selected by the global carry chain outputs. The data inputs to this mux would be the outputs of the local carry chain. Since each group of eight bits requires two carry chains, their are a total of 16 instances of this module.

The local carry chains operate in parallel with the group pgk and global carry chains. Therefore, the local carry chains need to complete computation in the time it takes to propagate through the group pgk and global carry chain. This means the speed of the local carry chain can be much less than the speed of the other chains that have been presented. This in turn means that the transistors can be downsized to save on space, without affecting adder timing.

The final issue to think about is the style of carry chain that should be used. In

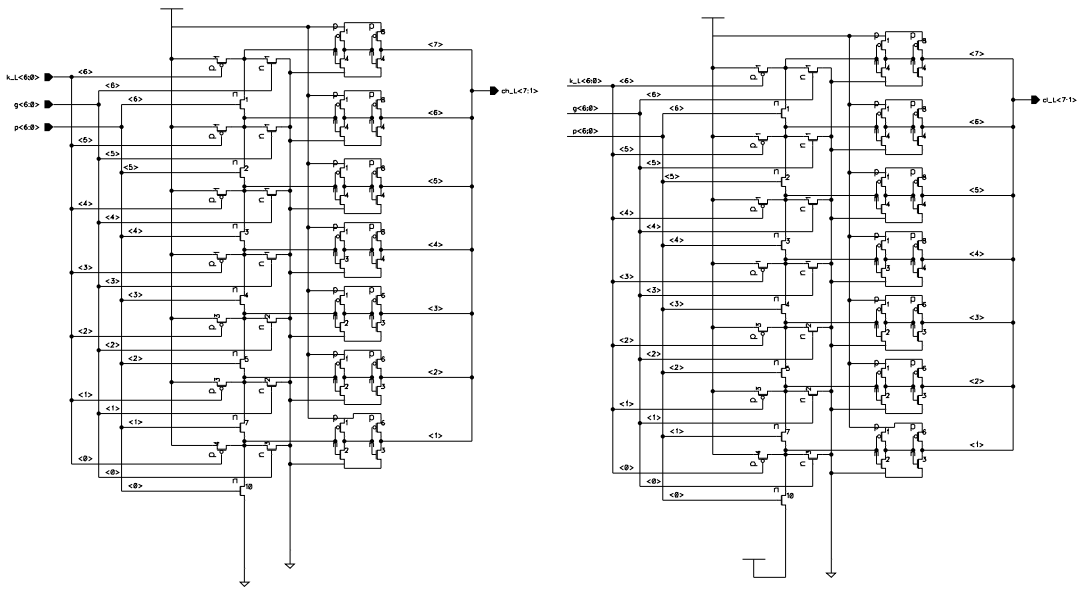


Figure 4-11: Local carry chain schematic

the case of the local carry chains, since the transistors can be small and still meet timing constraints, there is no need to switch to one of the other styles. Therefore, the local carry chains are simply chains of 8 links with no speculation or breaking up of the chain. See Figure 4-11 for the schematic. The worst case delay through the chain was 1.5ns. This delay is not much more than that of the global carry chain, but the size of the transistors is much smaller and layout is much easier.

4.1.5 Final Cell

The final cell takes in the outputs of both the local and global carry chains and the *propagate* output of the pgkblock. It uses these signals to determine what the sum bit will be. First, note that $sum = propagate \oplus localcarry$. Therefore, both local carries are sent through *xor* gates along with the *propagate* signal corresponding to that position. Then, the global carry bit is used to determine which of the two sum bits is the correct one. If the global carry bit is high, then the sum bit corresponding to the local carry bit from the chain with a carry input of 1 should be selected, else the other one should be selected. There are 64 instances of this module, one for each

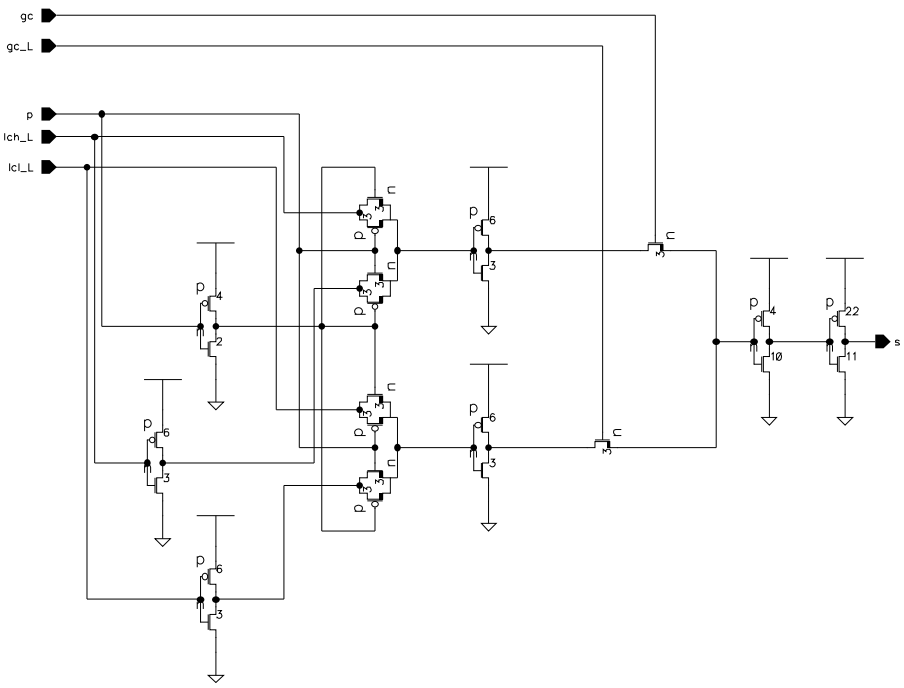


Figure 4-12: Final cell schematic

sum bit.

Since the local carries are early arriving signals, the *xor* need not be optimized for speed, but can instead be made small. A pass gate version is used, where the transistors are relatively small. The mux that follows these *xor*s is also a pass gate version, but it is an n-only pass gate version. The reason for this is to decrease the load on the global carry chain outputs. Note that each output of the global carry chain has to go to eight final cells. Thus, it is important to keep the gate load on the global carry signals as low as possible. Adding PFETs to the mux would cause a significant increase in the load. This in turn would be mean the buffers driving out of the global carry chain would have to be larger, in turn meaning there would be higher loads on each link of the chain, etc. See Figure 4-12 for the schematic.

The worst case delay through the final cell occurs when the global carry bit transitions from high to low resulting in the same transition on the output. The worst case delay is 700ps.

4.2 Zero Detect

This section covers the design and implementation of the zero detect unit. The zero detect unit takes in a 64 bit number and outputs a 1 if all 64 bits are 0, else it outputs a 0. There are many possible styles of zero detect that can be used, each having their own advantages and disadvantages. There are 5 main styles of zero-detect that were investigated. These are a precharged gate, static gates, CVSL, pseudo-NMOS gates, and push/pull CVSL. Within each of these styles, there are yet more designs that must be chosen from. In the end, a static gate version was chosen. In the following sections, these 5 styles are discussed. They are presented in the order in which they were considered.

4.2.1 Precharged Gate

This circuit would consist of one large gate with 65 NFETs and 1 PFET. The 64 NFETs would receive the 64 inputs at their gates while the PFET would be connected to the precharge signal. The remaining NFET would also be connected to the precharge signal at the gate. Its drain would be connected to the sources of each of the other 64 NFETs, while its source would be connected to ground. The precharge signal is typically a clock signal which can be used to tell when the 64 input bits have settled to their final values. First pass simulations showed that this style circuit would run fast (less than 1.3ns) and could also be quite small, where each of the 64 NFETs would be 6 microns wide.

However the reason this circuit is not used has to do with the timing of the precharge signal. The inputs to the zero-detect are not synchronized with the clock, therefore the clock signal could not be used as the precharge signal. Since there is no simple way of telling when the inputs to the zero-detect have settled, no precharge signal could be generated. Therefore, this circuit could not be used.

4.2.2 Pseudo-NMOS Gate

This circuit is structurally exactly the same as the precharged version. The only differences are that the PFET is permanently on, but it is small in size compared to the NFETs, and there is no 65th NFET between the other NFETs and ground. This circuit works but is slower than the precharged version due to the fight that occurs when an NFET is turned on. Due to the unwanted static power dissipation that occurs with this circuit, this style was not too seriously considered.

4.2.3 Static Gates

There are many ways a zero-detect can be implemented using static gates such as inverters, nands, and nors. Each of these ways differs in the number of stages, type of gates, and the radix of each stage. For example, one version with only three stages of logic is the nor-nand-nor version. Another version would be an inverter-and-nand-nor circuit containing five levels of logic. Of course, the number of levels goes up if you include some buffering. Each of these versions can have different radices for each stage. In the end, the inverter-and-and-and version with a 4-4-4 radix turned out to be a circuit that was fast and small compared to the others that were tested.

In testing each of the different versions, wire load is a major concern since there are interconnect wires running across many bit slices. With each bit being 25 microns wide, the wires can get to be very long and the wire capacitance can not be ignored. Calculations show that versions with 3 stages of logic had a wire load equivalent to 30 microns of gate between the first two stages, 70 microns of gate between the second two stages, and 100 microns of gate on the output. This is due to the increasing number of bit slices that wires have to traverse as the number of gates goes down at each level. In general, tested versions had 3 levels of logic, where each level may have some buffering in addition to a nand/nor gate. Fewer levels of logic with higher radices resulted in gates with long chains of transistors. More levels with smaller radices meant that the delay would become too large due to the amount of extra logic that has to be traversed.

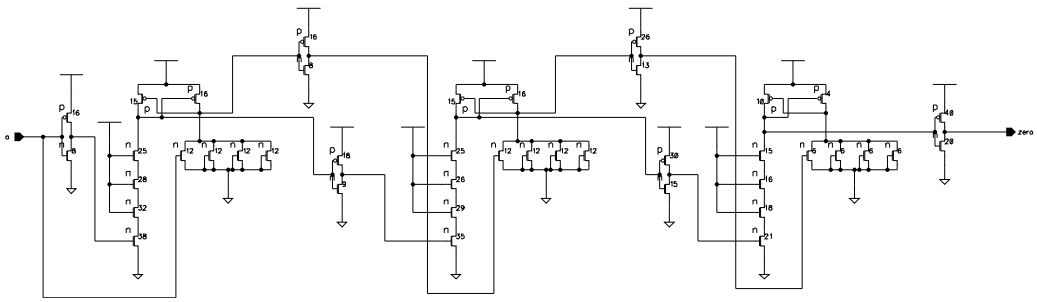


Figure 4-13: CVSL Zero Detect Slice Schematic

It appears that the 4-4-4 *and* gate version is the best primarily because it has no nor gates in it. The problem with nor gates is that it has chains of PFETs. Since the delay through PFETs is larger than through NFETs, PFETs must in general be very large compared to NFETs in order to get similar delays. The larger they get, the more of a load they put on the transistors driving it. Therefore, by eliminating these chains of PFETs, theoretically the circuit should be smaller and faster. This appears to be the case. Another reason this appears to be the best version is because each stage has a buffer, since an *and* gate consists of a *nand* gate and an inverter. This means the load on each *nand* gate is smaller since they will be driving inverters instead of the next stage *nand* plus wire load.

A first pass at layout of this version indicates that the transistors in every stage have room to grow if need be. Increasing the size of the transistors would not increase the area that the zero detect is taking up. The delay through this circuit is 1.5ns.

4.2.4 Cascode Voltage Swing Logic

The next style is the CVSL version. In CVSL, each gate takes in both senses of the input signal and outputs two signals, which are the inverse of each other. The load on each input signal is just one PFET and the same number of NFETs as would be found in the equivalent static gate. As a result, it is believed that since the load of a CVSL gate is smaller than that of an equivalent static gate, this version may run faster. The problem with this version is that fights occur between the PFET and its

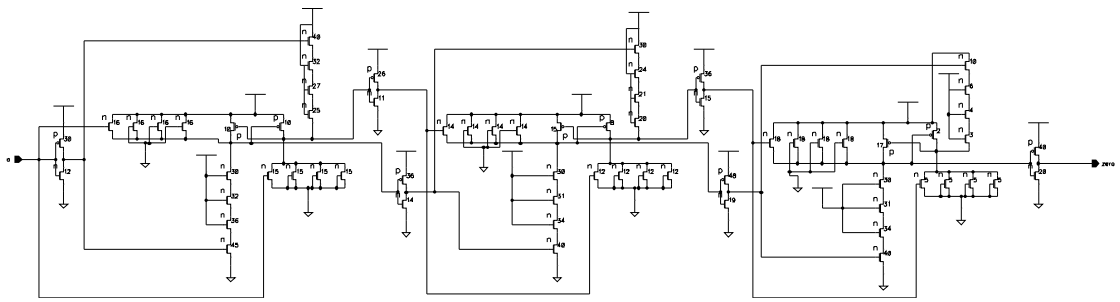


Figure 4-14: Push/Pull CVSL Zero Detect Slice Schematic

corresponding NFET structure thereby possibly offsetting any speed increase from the reduced load and also introducing increased power dissipation.

The first CVSL version to test has 3 *and* gates. See Figure 4-13 for a schematic of a slice of this version of the zero detect. At first pass, it runs in 2.3ns and is already rather big. The reason it is probably so slow is the load on each stage is large due to the lack of buffering. Therefore, buffering between each stage was introduced in the form of inverters. This brought the speed down to about 2ns. It appears that introducing the buffers does not cause a large speedup since the number of stages of logic was doubled. However, the signals are much sharper, having faster rise and fall times.

4.2.5 Push/Pull CVSL

The final version uses a modified CVSL version as was rediscovered by Dan Hartman of the Concurrent VLSI Architecture Group at the MIT Artificial Intelligence Lab. This idea was originally proposed by DEC. Under this modified version, each NFET structure is replicate on the opposing PFETs side, thereby helping the PFET pull up faster when the opposite NFET structure is pulling down. Theoretically, this should reduce the fight that occurs. Though the load on each stage goes up compared to normal CVSL, the load compared to static gates is smaller, since NFETs are being used on the pullup side instead of PFETs. However, this style has almost double the number of transistors as the CVSL version, therefore meaning it takes up nearly

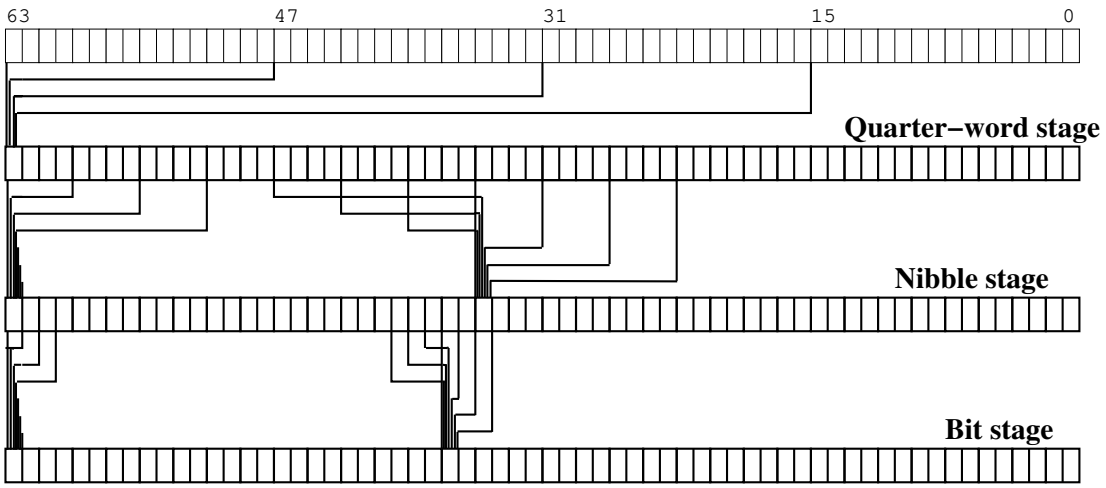


Figure 4-15: Shifter stages and interconnections

twice as much space [3]. See Figure 4-14 for a schematic of a slice of this zero detect. As it turns out, this circuit runs faster than the CVSL version and at about the same speed as the static version. It runs at 1.5ns. However, the transistors are rather large. Due to the more complicated nature of this circuit compared to static gates, interconnections between and within gates is more complex. In addition, first cut layout shows that just laying out the transistors (not worrying about interconnect) takes up much more space than the static version. Adding in the interconnect makes things even worse. For these reasons, the static gate version of the zero detect is the one that is used in the integer unit.

4.3 Shifter

The shifter that is used in the integer unit was designed and implemented by Jeff Bowers of the Concurrent VLSI Architecture Group in the MIT Artificial Intelligence Lab. The high level algorithm and design are presented here.

The shifter takes in a 64 bit quantity along with many control signals indicating if the operation is a rotate, arithmetic shift, or logical shift. Other control signals include the sign extension bits and amount to shift.

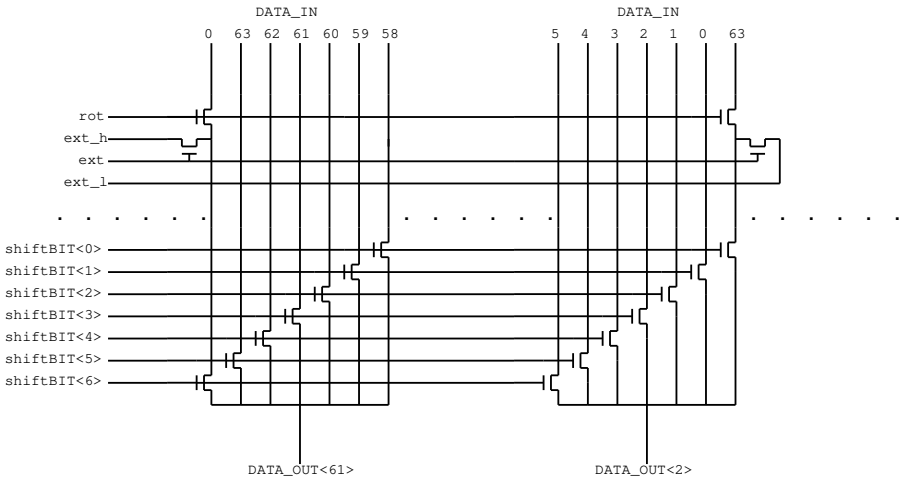


Figure 4-16: Internals of bit stage of shifter

The shifter uses a radix 4 shift algorithm. There are three logical stages in the shifter: the quarterword shift stage, nibble shift stage, and bit shift stage. Note that a quarterword is 16 bits and a nibble is 4 bits. At each stage, a particular bit can be set to any one of 7 different bit positions from the previous stage, depending on the amount and direction of the shift/rotate to be performed. See Figure 4-15 for an illustration of this. This implies muxes in front of each bit for each stage. These muxes are implemented as n-only pass gates to reduce loading. There are seven select signals for each of the three stages of logic. These signals come from control. See Figure 4-16 for an illustration of this mux.

The lengths of wires, particularly between the input and quarterword stage can become very long. Therefore, buffering is necessary between stages to drive this load. This results in some added delay through the shifter. Had the shifter not needed to rotate, the length of the longest wire could be much less, meaning less buffering would be needed.

The shifter also has the capability to perform sign extension on both the high and low ends of the word in the case of logical and arithmetic shifts. Based on the operation to perform, control sets the extension bits to their proper values. These values are then muxed into the correct bit position when a rotate is not being done.

See Figure 4-16 for an illustration of this logic.

The area of the shifter is constrained by the number of wires involved in the shifter. More than 150 M2 tracks are needed by the shifter, thus resulting in a large area of about 285 microns. The worst case delay through the shifter is 2.8ns for a rotate instruction, where signals have to be driven across long wires.

4.4 Selector

The selector is needed for proper execution of the byte manipulation instructions and immediate operations, as defined in the ISA of the M-Machine. These operations include inserts, extracts, IMM (create immediate), and SHORU. The selector takes in *insrc*, which is really the second source input to the integer unit, and *inshift*, which is the output of the shifter. The input control signals are *extract*, *extract_L*, and an 8 bit signal called *sel*. The output is a 64 bit number which will go to the result mux.

The selector basically consists of 64 3-input muxes, with the 3 inputs being ground, *insrc*, and *inshift*. The mux is controlled based on the control inputs. Each bit of *sel* is used in the control of 8 of the 3-input muxes. Basically, if *sel* is asserted, then *inshift* is passed. If *extract* is asserted and *sel* is not asserted, then a 0 is passed. Finally, if *extract* and *sel* are not asserted, the *insrc* is passed.

4.4.1 Extracts

The extract instructions include extract byte and extract halfword, meaning a particular byte or halfword needs to be put in the least significant bits of the output, with the most significant bits being set to zero. The shifter is responsible for shifting the input such that the byte or halfword to be extracted is in the low part of the word. The selector should then take in the shifted word and output the correct value. By asserting *extract* and *sel[0]* in the case of extract byte, or *extract* and *sel[3:0]* in the case of extract halfword, the selector will do the correct thing.

4.4.2 Inserts

The insert instructions include insert byte and insert halfword. These instructions specify the byte or halfword to be inserted in the first operand and the word into which something should be inserted in the second operand to the integer unit. The instruction also specifies which byte or halfword the first operand should be placed in. The shifter is responsible for shifting the byte or halfword into the correct position where it will be inserted. The selector then takes in the shifter output and value into which the byte or halfword should be inserted, selecting the proper values to pass to the output. The *extract* input should not be asserted. The *sel* signal(s) corresponding to the byte/halfword into which something will be inserted should be asserted, thus causing the shifter output to pass through to the selector output. For all other bytes, the *sel* signal should not be asserted, causing *insrc* to be passed through to the selector output.

4.4.3 Immediates

The immediate instructions include shift & or unsigned 16-bit immediate (SHORU) and create 16-bit immediatie (IMM). When SHORU is being executed, the low 16 bits of the second operand to the integer unit should be shifted into the low end of the first operand. Therefore, the shifter is responsible for shifting its input 16 bits to the left, while the selector is responsible for passing the low 2 bytes of *insrc* and the high 6 bytes of *inshift*. This is done by setting *extract* to 0, *sel[7:2]* to 1, and *sel[1:0]* to 0.

In the case of IMM, the immediate to be passed comes in to the selector at *insrc*. This signal should simply pass the through the selector unchanged. This is easily done by making sure *extract* and all 8 bits of *sel* are not asserted.

4.4.4 Shift/Rotate Operations

Whenever a shift or rotate operation is being performed, the selector must pass *inshift* unchanged through to the output. This is easily done by asserting all of the *sel* signals.

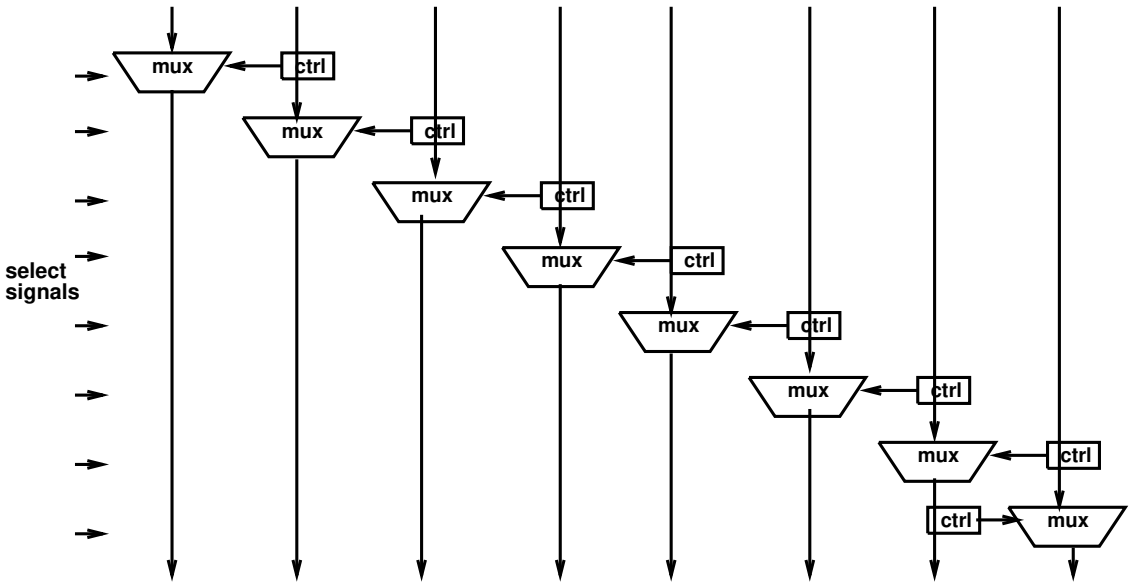


Figure 4-17: Layout strategy for the selector

4.4.5 Implementation

Implementation of the selector is straightforward. The mux consists of 3 n-only pass gates that feed some buffering. The select signals for these muxes come from control and from some local nor gates, whose inputs come from control. Since each group of 8 bits potentially receives different mux control signals, each group of muxes cannot be aligned with any other group of muxes, thus potentially leading to an inefficient layout. The reason all the mux select signals are not generated in control is due to the high number (24) of control wires that would need to run from control to the selector. By running only 10 control lines to the selector and locally generating mux select signals, space could be saved. This would be done by placing local logic in the space created by having to have each group of muxes unaligned. See Figure 4-17 for an illustration of this.

The worst case delay through the selector is only 550ps, which is the delay from data input to output, since the mux should be set up by the time the data arrives.

4.5 Mask Generator

The purpose of the mask generator is to serve as an aid in determining segment violations. It takes in the 6 bit *length* of the segment in which the original address is located. It outputs a 64 bit *mask* indicating which bit positions in the new address can differ from the corresponding bit positions in the old address without causing a segment violation. The *mask* goes to the segment checker, which also takes in the new and old addresses. A mask bit of 0 indicates that the new address and old address must have the same bit at that bit position, else there is a segment violation. A mask bit of 1 indicates that the new and old addresses can differ in that bit position without there being a segment violation. Consider the case where *length* is 001010. This means that the low 10 bits of the mask should be 1's, while the top 54 bits should be 0.

The design of the mask generator was done by Andrew Chang of the Concurrent VLSI Architecture Group of the MIT Artificial Intelligence Laboratory. A high level description of the mask generator is presented here. Each bit of the mask is computed independently of the others. A bit is basically computed by a chain of 6 cells, one cell for each length bit. Each cell takes in two signals from the previous stage (the first stage takes in power and ground) and a length bit. They output two signals to the next stage or one signal if the cell is in the last stage. The cell internals basically consists of two pass gate muxes, which are selected based on the *length* bits. The difference between each of the cells is how the inputs to these internal muxes are arranged. By combining these cells in different ways, the appropriate outputs can be attained for each of the 64 bit positions.

Consider bit position 10. If the MSB of *length* is asserted, then without knowing any of the other bits of *length*, the mask at that position should be 1. In the implementation of the mask generator the first of the six cells in bit position 10 would pass out two signals which are both 1. Since all future cells can only pass through what arrives at its inputs, the output of the sixth cell is guaranteed to be a 1. Now supposed the MSB was not asserted, then determination of the mask bit happens

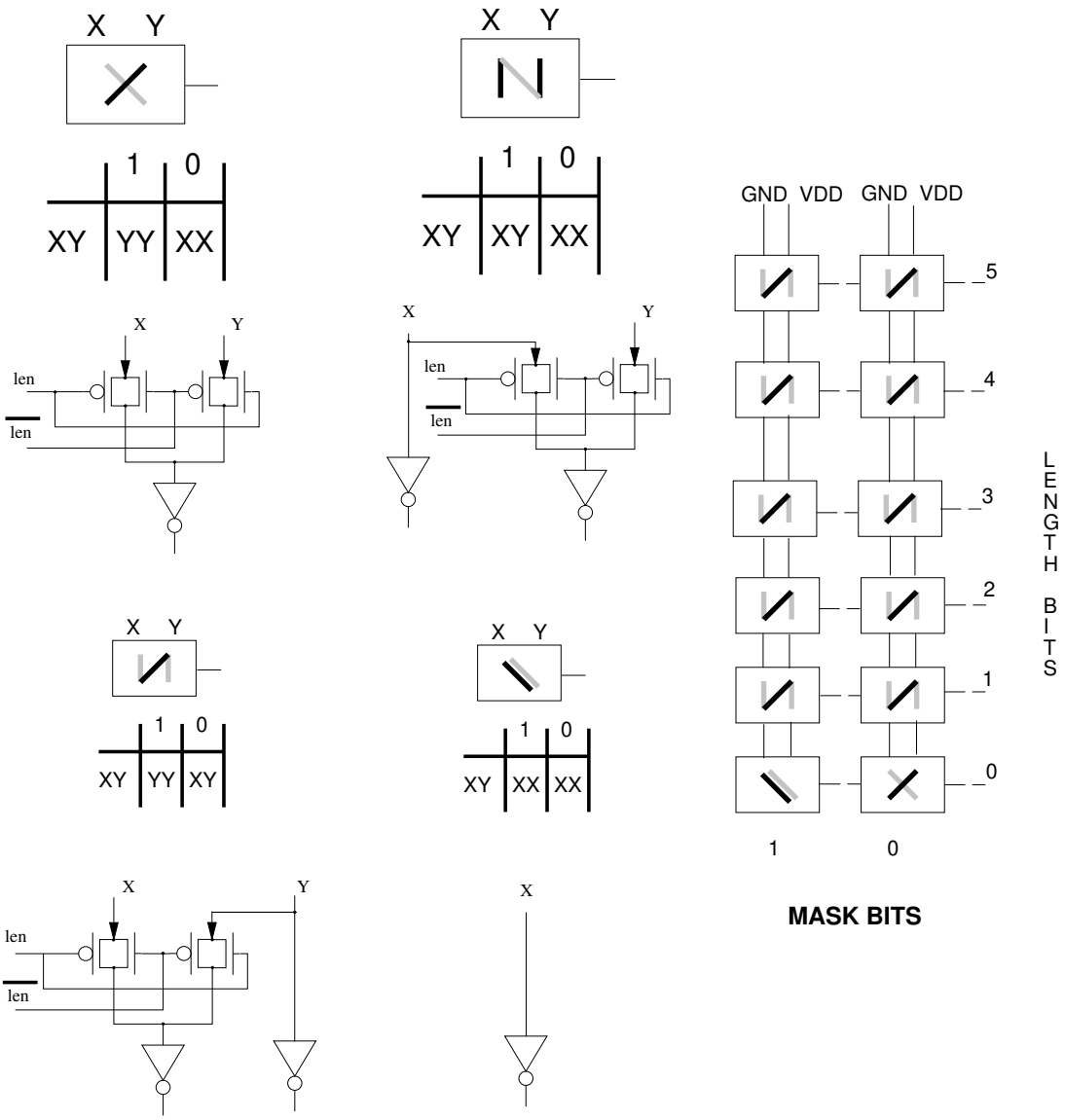


Figure 4-18: Mask generator basic cells and implementation for low 2 bits

in one of the later cells. In this way, the mask generator can use the length bits to determine the mask at each of the 64 bit positions. See Figure 4-18 for an illustration of the basic cells in the mask generator and sample implementations for two of the mask bits.

In the implementation of the mask generator, buffers are placed after the third cell in each bit position in order to break up the chain of six transistors that would have otherwise resulted. Also, note that an address is only 54 bits, with bits 55 to 64 encoding information such as the length of the segment and the pointer type. Therefore, the mask for these top ten bit positions should be a 0, indicating that any change in these bit positions is an error. The delay through the mask generator is 1.55ns in the worst case.

4.6 Segment Checker

The segment checker is used to detect segment violations. It takes in the 64 bit quantites *old address*, *new address* (which is to be checked for a violation), and *mask* from the mask generator. It outputs a one bit signal *segerror*, which is asserted if there is a segment violation. If any bit x of *mask* is 0 and bit x of *old address* does not equal bit x of *new address*, then *segerror* should be asserted.

The first stage of the segment checker is basically a 3-input mux. If the *mask* bit is 1, then this stage outputs a 0, thus indicating that there was no error at this bit. If *mask* is a 0 and *old* is a 0, then *new* is passed. This works, since if *new* was a 1, then there is an error and a 1 should be passed. If *new* was a 0, then there is no error and a 0 should be passed. If *mask* is a 0 and *old* is a 1, then the inverse of *new* should be passed. This works, since if *new* was a 0, then there is an error and a 1 should be passed. If *new* was a 1, then there is no error and a 0 should be passed. There are no tricks in the circuitry here. See Figure 4-19 for the schematic of this stage.

The second stage of the segment checker is simply a copy of the zero detection unit that was presented earlier. If there were no errors in any of the bit positions, then the zero detection unit will detect this and pass a 1 to an inverter, which will

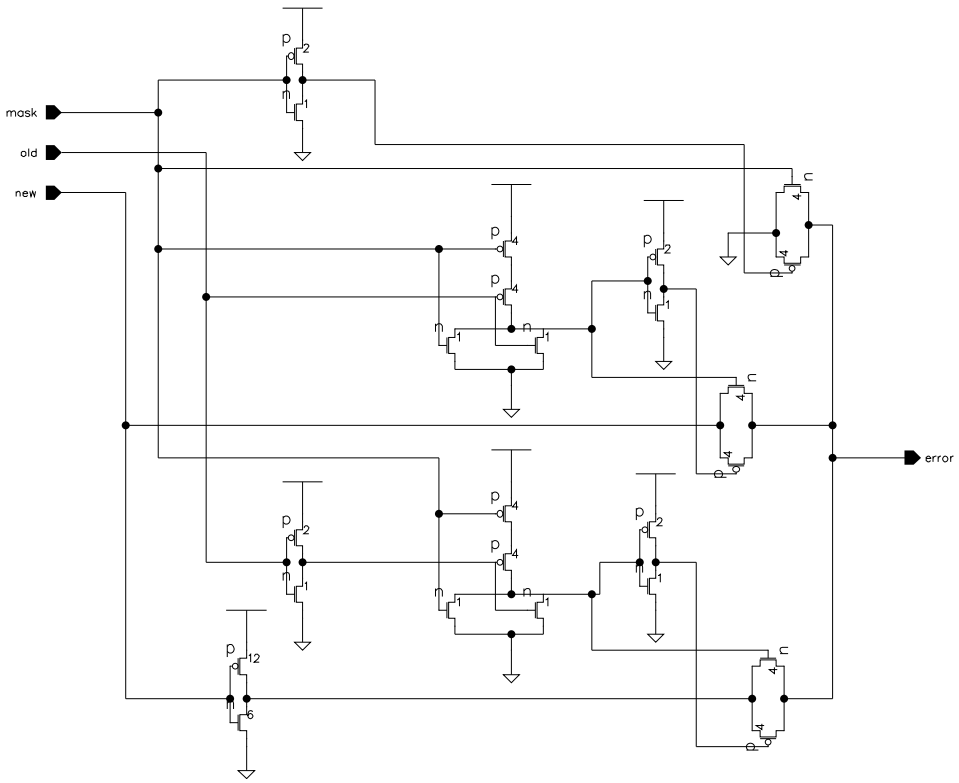


Figure 4-19: Schematic of first stage of segment checker

drive *segerror* to 0. If any of the inputs to the zero detect is a 1, thus indicating there was an error, then the output of the segment checker will be a 1.

The delay through the segment checker is 1.75ns in the worst case.

4.7 Boolean Unit

The boolean unit takes in two 64-bit operands, a and b, and 5 function bits, f. It outputs the 64-bit result of the boolean operation. See Figure 4-20 for the schematic of the boolean unit. The function bits indicate what boolean operation should be executed. These function bits go through two of four possible pass gates. These pass gates are selected by a, the inverse of a, b, and the inverse of b. For example, *f[0]* goes through pass gates controlled by a and b, while *f[1]* goes through pass gates controlled by a and the inverse of b. The circuit is set up such that one of the first four f bits

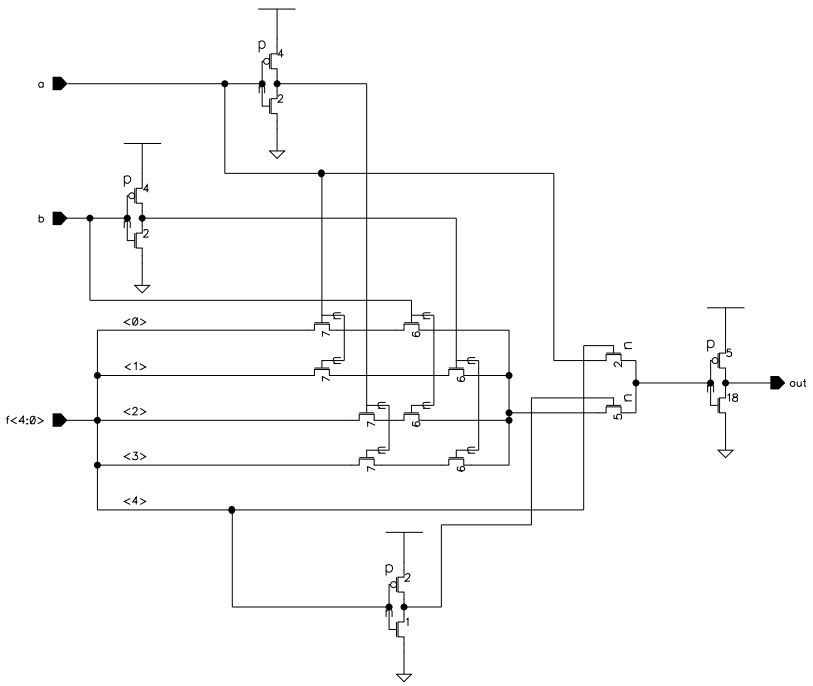


Figure 4-20: Boolean unit schematic

will propagate to a two-input mux controlled by *f[4]*. The other input to this mux is *a*. The output of the mux goes to an inverter which drives the output of the boolean unit. Therefore, if *f[4]* is asserted, then the inverse of *a* is passed to the output, else one of the first four *f* bits is inverted and passed to the output.

To determine how to choose a particular function, one must know the truth table for the function to be executed. Consider the *and* function. Here, if both *a* and *b* are asserted, then the output should be a 1, else it should be a 0. Therefore, *f[0]* and *f[4]* should be a 0, while bits 1, 2, and 3 of *f* should be ones. That way, if *a* and *b* are asserted, then *f[0]* will pass through the mux, be inverted, and be sent to the output, resulting in an output of 1. Otherwise, the output will be a 0. This is the type of process used in determining the function codes for each of the boolean functions that may be needed.

The worst case delay through the boolean unit is .5ns.

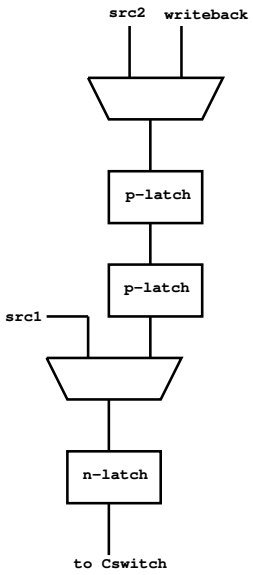


Figure 4-21: Block diagram of send unit internals

4.8 Send Unit

The send unit is responsible handling communication operations. It takes in both source operands to the integer unit and the writeback line, along with control signals. It outputs the value that should be sent across the C-switch in the current cycle. This value goes to the result mux.

In the first cycle of a communication operation, the first two values to send over the C-switch are present as the two source operands. Since only the first operand can be sent on the first cycle, the second operand has to be kept around in the send unit for the second cycle, at which point it will be driven to the C-switch. Following that, the data coming in on the writeback line will be passed to the C-switch. This all implies the need for p and n latches between the second operand and the mux which selects what to drive out of the send unit. Since the writeback line also needs to be held across cycles, there is a mux at the beginning of the send unit that selects between the second source operand and the writeback line. Prior to being driven to the output of the send unit, all data goes through an additional p latch, since the

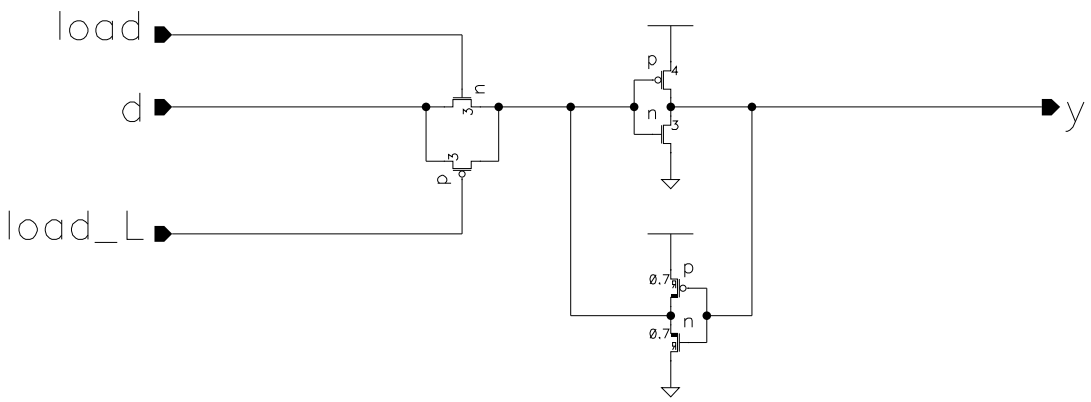


Figure 4-22: Schematic of a latch

next latch encountered on the route to the integer unit output is an n-latch. See Figure 4-21 for an illustration of the send unit internals.

There are no circuit tricks pulled here. The send unit just consists of latches, muxes, and inverters. The delay through the send unit is 750ps from the source 2 input to the output.

4.9 Errval Generator

The errval generator module is used to synthesize an errval that may need to be written back to the register file in the case of a deferred exception. It takes in 54 bits of the instruction pointer and ten bits that encode the pointer type of an errval and the type of errval being passed. The errval generator simply passes these through to the output with one level of buffering. The delay through this module is just 300ps. The output goes to an n-latch, which goes to the writeback mux. The errval is selected to go to the integer unit output by the writeback mux when a newly synthesized errval needs to be passed.

4.10 Latches

There are p latches and n latches in the integer unit. The p latch is transparent

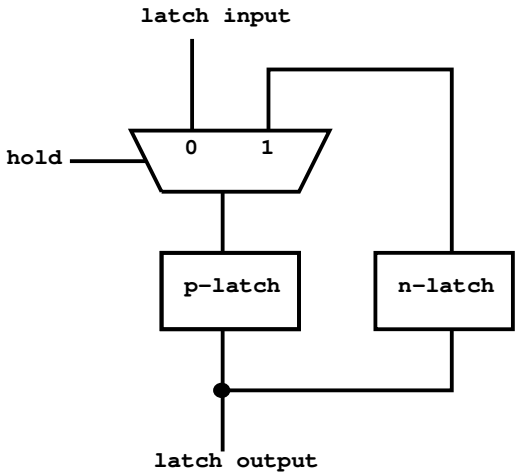


Figure 4-23: Block diagram of recirculating p-latch

when clock is high, while the n latch is transparent when clock is low. The n latch is simply implemented as a complementary pass gate that goes to an inverter. There is a feedback inverter to the input of the first inverter. Since the input to this feedback inverter is the output of the n latch, the designer has to make sure that there is no way for the output net to be unintentionally switched, thus causing the state of the n latch to possibly change. The delay through the n latch from clock to output is 370ps with a setup time of only 200ps. See Figure 4-22 for a schematic of a latch. By changing the way clock is hooked up to the load pins, this can be either a p latch or an n latch.

The p latch is a recirculating latch. It has the capability of holding its output across many clock cycles even though the input may change over those clock cycles. The recirculating p latch is implemented through use of a 2-input mux followed by a p latch with the same structure as the n latch. There is also a feedback n latch which goes from the output of the p latch to one of the mux inputs. The other mux input is the recirculating p latch input. The feedback n latch has no keeper in it, thereby making it dynamic. Since this latch is clocked every cycle, this should not be a problem. When the latch should hold a value, the mux selects the feedback signal, else the input signal is selected. The output of the internal p latch is also the output

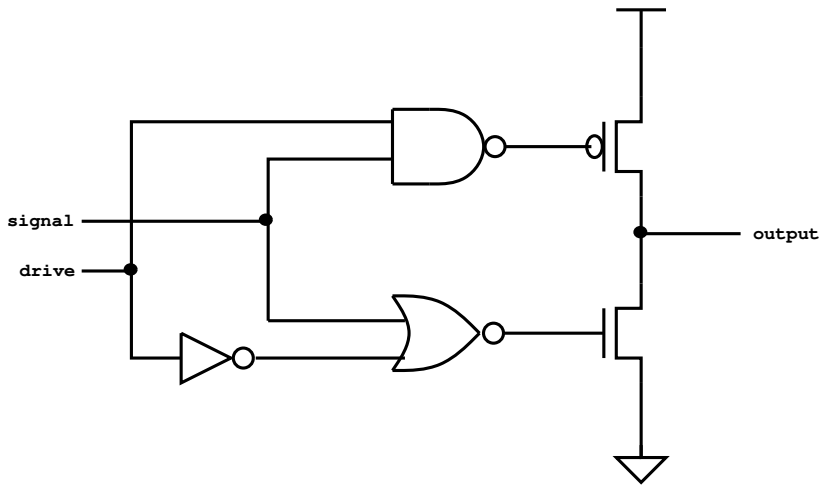


Figure 4-24: Internals of the tri-state driver

of the recirculating p latch. See Figure 4-23 for a block diagram of the recirculating p latch. The delay from clock to output is 850ps with a setup time of 250ps.

4.11 Muxes

The muxes used in the integer unit are all pass gate muxes. Only the 6-input mux is a non-complementary pass gate version. This is due to the fact that 12 control signals would otherwise be needed to control a complementary version of the 6-input mux. Both the 2-input mux and 4-input mux are complementary. There are no circuit tricks in any of the muxes.

4.12 Demultiplexor

The demultiplexor is responsible for driving the 64-bit result of the integer unit to either the C-switch or the register file. The demultiplexor internally consists of two tri-state drivers designed by Dan Hartman. Control signals indicate which driver, if any, should drive the result. These drivers are very large in order to drive the large load on the output lines. They work by gating the drive signals with the data signals, the output of which turns on the proper transistors that drive the output lines. If

the drive signal is not asserted, nothing is driven. See Figure 4-24 for an illustration of the driver logic. The delay through each driver is 700ps.

Chapter 5

Conclusions

5.1 Summary

This thesis presented the design and implementation of the integer unit datapath, which is found in the execution pipeline stage of the MAP chip cluster. The MAP chip can be found at each node of the M-Machine. First, the design flow and some design and implementation techniques were presented. Next, a high level specification of the integer unit was presented. From this we found out what the integer unit is supposed to do, what its inputs and outputs are, and how much time it has to execute an instruction.

Knowing this, we went on to develop the architecture of the integer unit. First, we determined what were the major modules needed in the integer unit in order to carry out all of the instructions. Then we went on to connect these modules together with muxes and latches to arrive at our final integer unit architecture.

The next chapter went on the describe each of the modules in more detail, discussing the design and implementation of each of the modules. The adder posed the greatest design challenge. A study of carry chains was done in the adder section. The zero detect section presented a study of various circuit styles that could have been used. Each section ended with some timing information about each module.

5.2 Suggestions For Further Study and Improvement

Due to the time constraints involved in designing a chip, the amount of time spent investigating more alternatives for various circuits was limited. In designing the adder, only one type of adder was worked on: the carry select/lookahead hybrid adder. Studies of other types of adders would be an interesting study to do. Another interesting related study would be building an adder using dynamic logic or CVSL, and then doing speed comparisons versus the static implementation.

Given more time, improvements could have been made in the space/time tradeoff. In the middle of this project, the timing constraints were loosened since it was decided that simulations should be done at the typical/typical low voltage corner, as opposed to the slow/slow corner. Many of the modules had already been designed and implemented by this time, including the adder and zero detect. These are very large modules that are used through the chip. However, due to time and manpower constraints, these modules could not be reimplemented as smaller modules with slightly longer delays. This could have led to much more efficient modules. However, sacrifices have to be made if a chip is to be fabricated quickly so as to still be competitive in the marketplace.

Appendix A

M-Machine Instruction Set Architecture

A.1 Arithmetic Operations

A.1.1 ADD

- Description: Integer signed addition. The value in *src2* can either be a register name or a short immediate.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← *src1* + *src2*

A.1.2 ADDU - add unsigned

- Description: Integer unsigned addition. The value in *src2* can either be a register name or a short immediate. Returns a value that is congruent mod 2^n of the actual value on overflow.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← *src1* + *src2*

A.1.3 SUB - subtract

- Description: Integer signed subtraction. The value in *src2* can either be a register name or a short immediate.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← *src1* − *src2*

A.1.4 SUBU - subtract unsigned

- Description: Integer unsigned subtraction. The value in *src2* can either be a register name or a short immediate. Returns a value that is congruent mod 2^n of the actual value on overflow.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← *src1* − *src2*

A.1.5 ASH - arithmetic shift

- Description: Arithmetic shift: Direction is determined by the sign of the src2 (or immediate). A positive sign results in a left shift. A negative sign results in a right shift. During right shifts sign extension is performed on the high order bits.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← ashift(*src1*, *src2*)

A.1.6 LSH - logical shift

- Description: Logical shift: Direction is determined by the sign of the src2 (or immediate). A positive sign results in a left shift. A negative sign results in a right shift. Zeroes are shifted into the MSB during right shifts and into the LSB during left shifts.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← lshift(*src1*, *src2*)

A.1.7 ROT - rotate

- Description: Rotate: Direction is determined by the sign of the src2 (or immediate). A positive sign results in a left shift. A negative sign results in a right shift.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← rotate(*src1*, *src2*)

A.1.8 AND - bitwise logical and

- Description: Bitwise logical AND. If one/both of the inputs is/are *pointer* type the output is still the bitwise logical AND of the operands, but the pointer bit is cleared.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← *src1* & *src2*

A.1.9 OR - bitwise logical or

- Description: Bitwise logical OR. If one/both of the inputs is/are *pointer* type the output is still the bitwise logical OR of the operands, but the pointer bit is cleared.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← *src1* | *src2*

A.1.10 XOR - bitwise logical exclusive or

- Description: Bitwise logical XOR. If one/both of the inputs is/are *pointer* type the output is still the bitwise logical XOR of the operands, but the pointer bit is cleared.

- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  $dest \leftarrow src1 \oplus src2$

A.1.11 NOT - bitwise logical negation

- Description: Bitwise logical NOT. If the input is *pointer* type the output is still the bitwise logical NEGATION, but the pointer bit is cleared.
- Errval/Exceptions:
  - Generates an Errval if the source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  $dest \leftarrow \neg src$

A.2 Byte Manipulation

A.2.1 EXTB - extract byte

- Description: Extracts a byte from *src1* and places it in the low byte of *dest*. The position of the byte is specified by the low 3 bits of *src2*. The rest of the bits in *dest* are zeroed out.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  tmp$\leftarrow src1 \gg (src2 \ll 3)$
  $dest \leftarrow$ tmp & 0xff

A.2.2 EXTH - extract halfword

- Description: Extracts a halfword (4 bytes) from *src1* and places it in the low four bytes of *dest*. The position of the halfword is specified by the third least significant bit (bit 2) of *src2*. The rest of the bits in *dest* are zeroed out.
- Errval/Exceptions:
  - Generates an Errval if either source operand is an Errval.

- Causes *no* exceptions.

- Operation:
  tmp← *src1* ≫ ((*src2* & 0x4) ≪ 3)
  dest← tmp & 0xffffffff

A.2.3 INSB - insert byte

- Description: Inserts the byte located in the low bits of the register *src2* into a byte location of the word in register *src1*. The byte location is specified in bits 32–34 (a number from 0–7) in register *src2*. Note: *src2* can NOT be an immediate.

- Errval/Exceptions:

  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.

- Operation:
  tmp1← ((*src2* ≫ 32) & 0x7) ≪ 3
  tmp2← *src1* & ¬(0xff ≪ tmp1)
  dest← tmp2 | ((*src2* & 0xff) ≪ tmp1)

A.2.4 INSH - insert halfword

- Description: Inserts the halfword located in the 32 bits of the register *src2* into a halfword location of the word in register *src1*. The halfword location is specified in bit 34 (0, 1) in register *src2*. Note: *src2* can NOT be an immediate.

- Errval/Exceptions:

  - Generates an Errval if either source operand is an Errval.
  - Causes *no* exceptions.

- Operation:
  tmp1← ((*src2* ≫ 32) & 0x4) ≪ 3
  tmp2← *src1* & ¬(0xffffffff ≪ tmp1)
  dest← tmp2 | ((*src2* & 0xffffffff) ≪ tmp1)

A.3 Comparison Operations

A.3.1 ILT - integer less than

- Description: Compares the signed integer in register *src1* with that in *src2*, placing a "1" (TRUE) in the destination condition register if *src1* < *src2* and placing a "0" (FALSE) there otherwise. The destination may be a local condition register or one of the writable global condition registers.

- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest← *src1* < *src2*

A.3.2 ILE - integer less than or equal

- Description: Compares the signed integer in register *src1* with that in *src2*, placing a "1" (TRUE) in the destination condition register if *src1* ≤ *src2* and placing a "0" (FALSE) there otherwise. The destination may be a local condition register or one of the writable global condition registers.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest← *src1* ≤ *src2*

A.3.3 ULT - unsigned less than

- Description: Compares the unsigned integer in register *src1* with that in *src2*, placing a "1" (TRUE) in the destination condition register if *src1* < *src2* and placing a "0" (FALSE) there otherwise. The destination may be a local condition register or one of the writable global condition registers.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest← *src1* < *src2*

A.3.4 ULE - unsigned less than or equal

- Description: Compares the unsigned integer in register *src1* with that in *src2*, placing a "1" (TRUE) in the destination condition register if *src1* $\leq$ *src2* and placing a "0" (FALSE) there otherwise. The destination may be a local condition register or one of the writable global condition registers.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest$\leftarrow$ *src1* $\leq$ *src2*

A.3.5 INE - integer not equal

- Description: Compares the signed integer in register *src1* with that in *src2*, placing a "1" (TRUE) in the destination condition register if *src1* $\neq$ *src2* and placing a "0" (FALSE) there otherwise. The destination may be a local condition register or one of the writable global condition registers.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest$\leftarrow$ *src1* $\neq$ *src2*

A.3.6 IEQ - integer equal

- Description: Compares the signed integer in register *src1* with that in *src2*, placing a "1" (TRUE) in the destination condition register if *src1* = *src2* and placing a "0" (FALSE) there otherwise. The destination may be a local condition register or one of the writable global condition registers.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest$\leftarrow$ *src1* == *src2*

A.3.7 CCAND - condition code logical and

- Description: Performs the logical "and" of the contents of condition registers *src1* and *src2*. The source operands may be any local or global condition registers. The destination may be a local or a writable global condition register.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest← *ccsrc1* && *ccsrc2*

A.3.8 CCOR - condition code logical or

- Description: Performs the logical "or" of the contents of condition registers *src1* and *src2*. The source operands may be any local or global condition registers. The destination may be a local or a writable global condition register.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest← *ccsrc1* || *ccsrc2*

A.3.9 CCNAND - condition code logical nand

- Description: Performs the logical "nand" of the contents of condition registers *src1* and *src2*. The source operands may be any local or global condition registers. The destination may be a local or a writable global condition register.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes pointers in either source operands into integers.
  - Causes an exception if either source operands are Errvals.
- Operation:
  ccdest← ¬(*ccsrc1* && *ccsrc2*)

A.4 Data Movement

A.4.1 MOV - move register

- Description: Moves value from *src* to *dest*. *Preserves* the RSZ bit.
- Errval/Exceptions:
  - Generates an Errval if the source operand is an Errval.
  - Causes *no* exceptions.
- Operation:
  dest← *src*

A.4.2 EMPTY - unset register presence bits

- Description: Sets the register presence bits to empty. The low 16 bits of *src* are a bitmask indicating which registers to empty. A "1" in position i means that register i is to be emptied. The eighteen bits of immediate are split between the *val* field and the *src* field.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Demotes any pointer, except Errval, in *src* operand to an integer.
  - Causes an exception if source operand is Errval.
- Operation:

```
tmp←src & 0xffff
for(i=0; i<16; i++)
   if(tmp & 0x1 == 1)
      REG(i).status ← 0
   tmp←tmp ≫ 1
```

A.4.3 CCEMPTY - unset condition register presence bits

- Description: Sets the condition register presence bits to empty. The low 16 bits of *src* are a bitmask indicating which registers to empty. A "1" in position i means that register i is to be emptied. The local condition registers specified in the low 4 bits, while the writable global condition registers are specified in bits 8 and 9. The eighteen bits of immediate are split between the *val* field and the *src* field. The eighteen bits of immediate are split between the *val* field and the *src* field.
- Errval/Exceptions:
  - Generates *no* Errval's.

- Demotes any pointer, except Errval, in *src* operand to an integer.
- Causes an exception if source operand is Errval.

- Operation:

```
tmp←src & 0xffff
for(i=0; i<4; i++)
   if(tmp & 0x1 == 1)
      CCREG(local, i).status ← 0
   tmp←tmp ≫ 1
tmp←tmp ≫ 4
for(i=0; i<2; i++)
   if(tmp & 0x1 == 1)
      CCREG(global, i).status ← 0
   tmp←tmp ≫ 1
```

A.5 Control Flow Operations

A.5.1 BR - branch

- Description: Control flow change. Branch target is IP relative and the number specified in *src* is the relative number of bytes. Note that it is the programmer's (or assembler's) responsibility to ensure that all branch targets are on halfword boundaries and are to the beginning of a valid instruction. The eighteen bits of immediate are split between the *val* field and the *src* field. The M-Machine architecture has three (3) branch delay slots.

- Errval/Exceptions:
  - Generates an Errval under the following conditions
    1. Result of post-increment is out of segment.
    2. *src* is an ErrVal
  - Demotes any pointer, except Errval, in *src* operand to an integer.
  - Causes an exception if the source operand is an Errval.

- Operation:

  IP← IP + *src*

A.5.2 JMP - jump

- Description: Control flow change to computed jump target. The jump target is contained in *src*. Note: *src* can NOT be an immediate. The M-Machine architecture has three (3) delay slots after a jump.

- Errval/Exceptions:

- Generates *no* Errval's.
- Causes an exception under the following conditions
  1. The source operand is an Errval.
  2. Permission violation.
  3. Source operand is an integer.

- Operation:

  IP$\leftarrow$ *src*

A.5.3 ILL - illegal instruction

- Description: Causes an illegal instruction exception when executed.
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Causes an exception on excecution.
- Operation:

  Illegal instruction fault

A.6 Address Calculation

A.6.1 LEA - load effective address

- Description: Creates a new pointer by adding the integer contents of the register *src2* to the address of the pointer in *src1*, storing the result into register *dest*. A fault is signalled if the operation creates a pointer outside the segment of the original pointer.
- Errval/Exceptions:
  - Generates an Errval if
    1. Result of post-increment is out of segment.
    2. *src1* is an ErrVal, Key, Enter.
    3. *src2* is an ErrVal.
  - Propagates an Errval if *src1* operand is an Errval.
  - Demotes any pointer, except Errval, in *src2* to an integer.
  - Causes *no* exceptions.
- Operation:
  dest$\leftarrow$ *src1* + *src2*

A.6.2 LEAB - load effective address from base

- Description: Creates a new pointer by adding the integer contents of the register *src2* to the segment base of the pointer in *src1*, storing the result into register *dest*. A fault is signalled if the operation creates a pointer outside the segment of the original pointer.
- Errval/Exceptions:
  - Generates an Errval if
    1. Result of post-increment is out of segment.
    2. *src1* is an ErrVal, Key, Enter.
    3. *src2* is an ErrVal.
  - Propagates an Errval if *src1* operand is an Errval.
  - Demotes any pointer, except Errval, in *src2* to an integer.
  - Causes *no* exceptions.
- Operation:
  dest← Ptr_base(*src1*) + *src2*

A.6.3 SETPTR - set pointer bit

- Description: Sets the pointer bit of the contents of register *src* and places the result in register *dest*. A SETPTR performed on an existing pointer results in a pass through. This operation is privileged and may only be executed in privileged mode.
- Errval/Exceptions:
  - Generates an Errval if
    1. The source operand is an Errval
    2. The integer bit pattern results in an Errval.
  - Causes an exception if IP is not execute-system.
- Operation:
  tmp← *src1*
  tmp.ptr← 1
  dest← tmp1

A.6.4 ISPTR - test pointer bit

- Description: Checks the pointer bit of the value in register *src*, placing "1" in *ccdest* if *src* is a pointer, and "0" otherwise.
- Errval/Exceptions:

- Generates *no* Errval's.
- Causes *no* exceptions.

- Operation:
  ccdest← *src*.ptr

A.6.5 ISERR - test for *Errval*

- Description: Checks the pointer bit and permission fields of the value in register *src*, placing "1" in *ccdest* if *src* is an *Errval*, and "0" otherwise.

- Errval/Exceptions:
  - Generates *no* Errval's.
  - Causes *no* exceptions.

- Operation:
  if((*src*.permission == *Error Value*) && (*src*.ptr))
    ccdest← 1
  else
    ccdest← 0

A.7 Immediate Operations

A.7.1 SHORU - shift & or unsigned 16-bit immediate

- Description: Left shift (*Logical*) the value in *dest* 16 bits and "bitwise OR" the 16-bit immediate specificed in *val* into the 16 lowest bits of *dest*. No sign extension is performed on the result.

- Errval/Exceptions:
  - Generates an Errval if the source operand is an Errval.
  - Causes *no* exceptions.

- Operation:
  tmp← left lshift(*dest*)
  dest← *tmp* | *val*

A.7.2 IMM - create 16-bit immediate

- Description: Create the 16-Bit immediate specified by *val* and place into *dest*. Sign extension *is* performed.

- Errval/Exceptions:

- Generates *no* Errval's.
- Causes *no* exceptions.

- Operation:
  dest← SEXT(*val*)

A.8 Configuration Space Operations

A.8.1 IGTLB - GTLB Access

- Description:

  Atomically perform a transaction on the GTLB - read, write or probe.

- Operation:
  if(*src2*.permission ≠ *execute message*)
    Fault(Illegal_code_pointer)
  else
    Message_buffer← GTLB_Translate(*src1*)
    Message_buffer← *src2*
    for(count = 0; count < *length*; count++)
      Message_buffer← REG(int, count+4)
  if((GTLB_Translate(*src1*) == HIT) && (count = length))
    ccdest← 1
  else
    ccdest← 0

A.9 Communication Operations

A.9.1 ISND0 - send priority 0 message, *user level*

- Description:

  Atomically sends a priority 0 message from the message composition registers. The message destination address (a virtual address) is specified in *src1* and a pointer to the code at the remote end is specified in *src2*. The *src1* value is translated into a node identifier automatically by the GTLB. If *src2* is not in the *enter message* state, a fault will occur. The body of the message is specified by the *length* field. The length field is split between *lenmsb* and *len*. The *bank* field is superfluous. The integer register file contains the message body and the *length* (0–9 for user threads, 0–11 for the Event V-Thread) field specifies the number of words in the message body, beginning from register i4.

- Errval/Exceptions:

  - Generates *no* Errval's.

- Causes an exception if
  1. *src1* (ie. Virtual Addr) is an ErrVal.
  2. *src2* (ie. dest IP) is not *Enter Message* type.

- Operation:

```
if(src2.permission ≠ enter message)
   Fault(Illegal_code_pointer)
else
   Message_buffer← GTLB_Translate(src1)
   Message_buffer← src2
   for(count = 0; count < length; count++)
      Message_buffer← REG(int, count+4)
if( (GTLB_Translate(src1) == HIT) && (count = length))
    ccdest← 1
else
    ccdest← 0
```

A.9.2 ISND0O - send priority 0 message, *user level*

- Description: Same as above, except ordered delivery is guranteed.

A.9.3 ISND0P - send priority 0 message physical

- Description:

  Atomically sends a priority 0 message from the message composition registers. The message destination address (a *physical* node number) is specified in *src1* and a pointer to the code at the remote end is specified in *src2*. If *src2* is not in the *enter message* state, a fault will occur. The body of the message is specified by the *length* field. The length field is split between *lenmsb* and *len*. The *bank* field is superfluous. The integer register file contains the message body and the the *length* (0–9 for user threads, 0–11 for the Event V-Thread) field specifies the number of words in the message body, beginning from register i4.

  Note: this is a system level send operation.

- Errval/Exceptions:
  - Generates *no* Errval's.
  - Causes an exception if
    1. *src1* (ie. Virtual Addr) is an ErrVal.
    2. *src2* (ie. dest IP) is not *Enter Message* type.
    3. IP is not execute-system.

- Operation:
  if(*src2*.permission $\neq$ *enter message*)
    Fault(Illegal_code_pointer)
  else
    Message_buffer$\leftarrow$ *src1*
    Message_buffer$\leftarrow$ *src2*
    for(count = 0; count $<$ *length*; count++)
      Message_buffer$\leftarrow$ REG(int, count+4)
  if(count = length)
    ccdest$\leftarrow$ 1
  else
    ccdest$\leftarrow$ 0

A.9.4 ISND0PO - send priority 0 message physical, ordered

- Description: Same as above, except ordered delivery is guranteed.

A.9.5 ISND0PNT - send priority 0 message physical, no throttling

- Description:

  Atomically sends a priority 0 message from the message composition registers. The outgoing message counter is not automatically decremented. The message destination address (a *physical* node number) is specified in *src1* and a pointer to the code at the remote end is specified in *src2*. If *src2* is not in the *enter message* state, a fault will occur. The body of the message is specified by the *length* field. The length field is split between *lenmsb* and *len*. The *bank* field is superfluous. The integer register file contains the message body and the *length* (0–9 for user threads, 0–11 for the Event V-Thread) field specifies the number of words in the message body, beginning from register i4.

  Note: this is a system level send operation.

- Errval/Exceptions:

  - Generates *no* Errval's.
  - Causes an exception if
    1. *src1* (ie. Virtual Addr) is an ErrVal.
    2. *src2* (ie. dest IP) is not *Enter Message* type.
    3. IP is not execute-system.

- Operation:
  if(*src2*.permission $\neq$ *enter message*)
    Fault(Illegal_code_pointer)

else
Message_buffer← *src1*
Message_buffer← *src2*
for(count = 0; count < *length*; count++)
Message_buffer← REG(int, count+4)
if(count = length)
ccdest← 1
else
ccdest← 0

A.9.6 ISND0PNTO - send priority 0 message physical, no throttling, ordered

- Description: Same as above, excepted ordered message delivery guranteed.

A.9.7 ISND1PNT - send priority 1 message physical, no throttling

- Description:

  Atomically sends a priority 1 message from the message composition registers. The message destination address (a *physical* node identifier) is specified in *src1* and a pointer to the code at the remote end is specified in *src2*. If *src2* is not in the *enter message* state, a fault will occur. The body of the message is specified by the *length* field. The length field is split between *lenmsb* and *len*. The *bank* field is superfluous. The integer register file contains the message body and the *length* (0–9 for user threads, 0–11 for the Event V-Thread) field specifies the number of words in the message body, beginning from register i4.

  Note: this is a system level send operation.

- Errval/Exceptions:
  - Generates *no* Errval's.
  - Causes an exception if
    1. *src1* (ie. Virtual Addr) is an ErrVal.
    2. *src2* (ie. dest IP) is not *Enter Message* type.
    3. IP is not execute-system.

- Operation:
  if(*src2*.permission ≠ *enter message*)
  Fault(Illegal_code_pointer)
  else
  Message_buffer← *src1*
  Message_buffer← *src2*

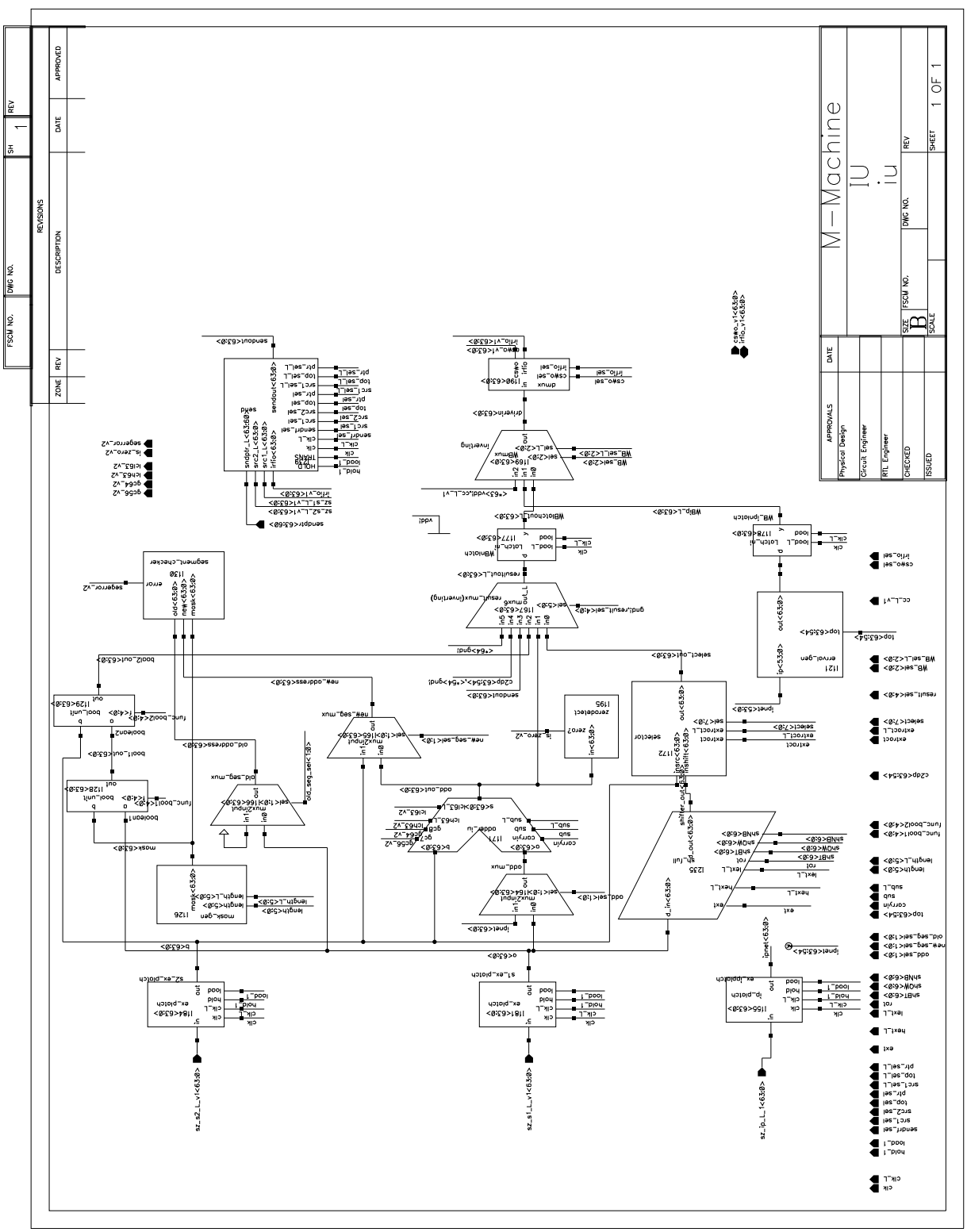
```
    for(count = 0; count < length; count++)
        Message_buffer← REG(int, count+4)
if(count = length)
    ccdest← 1
else
    ccdest← 0
```

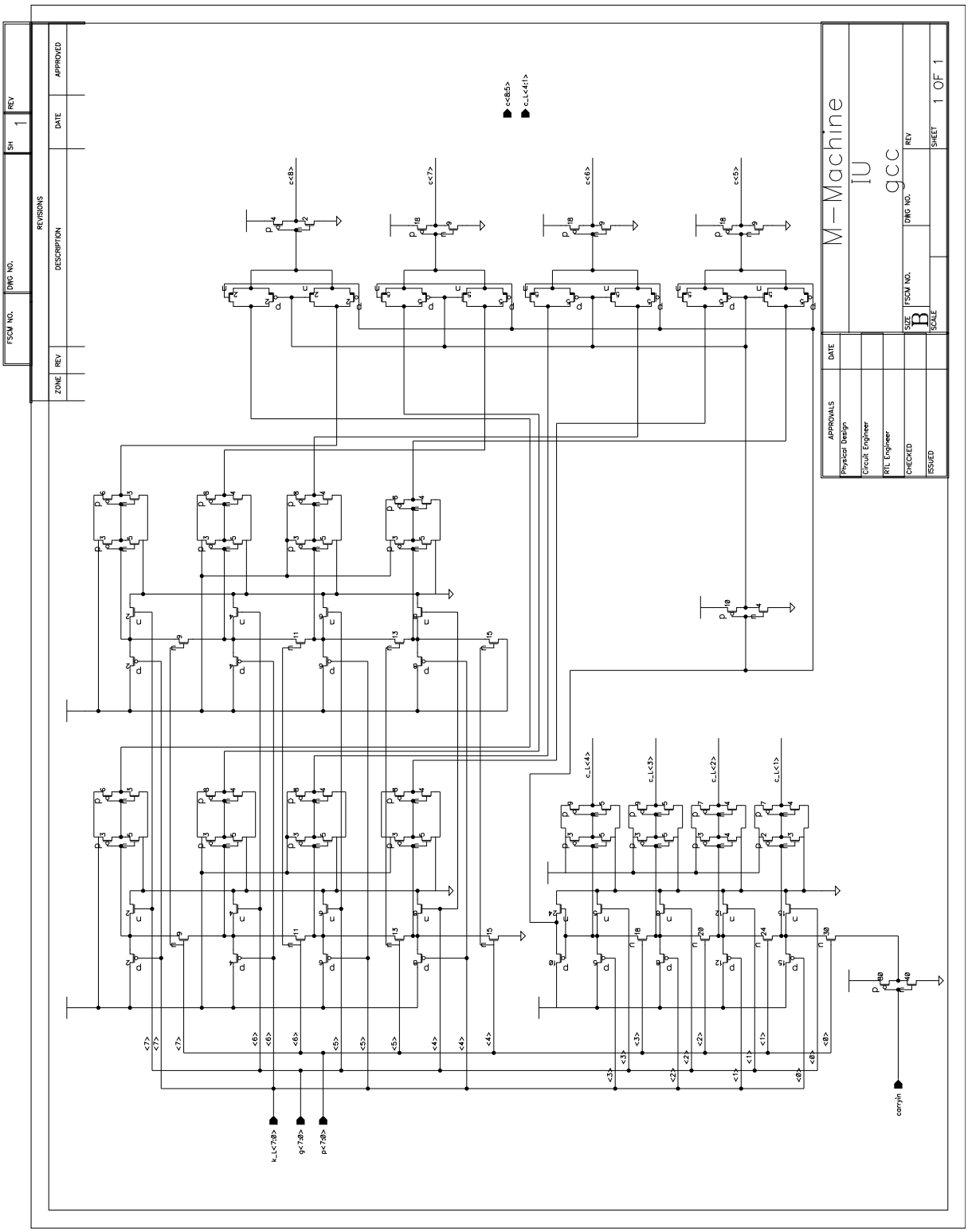
A.9.8 ISND1PNTO - send priority 1 message physical, no throttling ordered

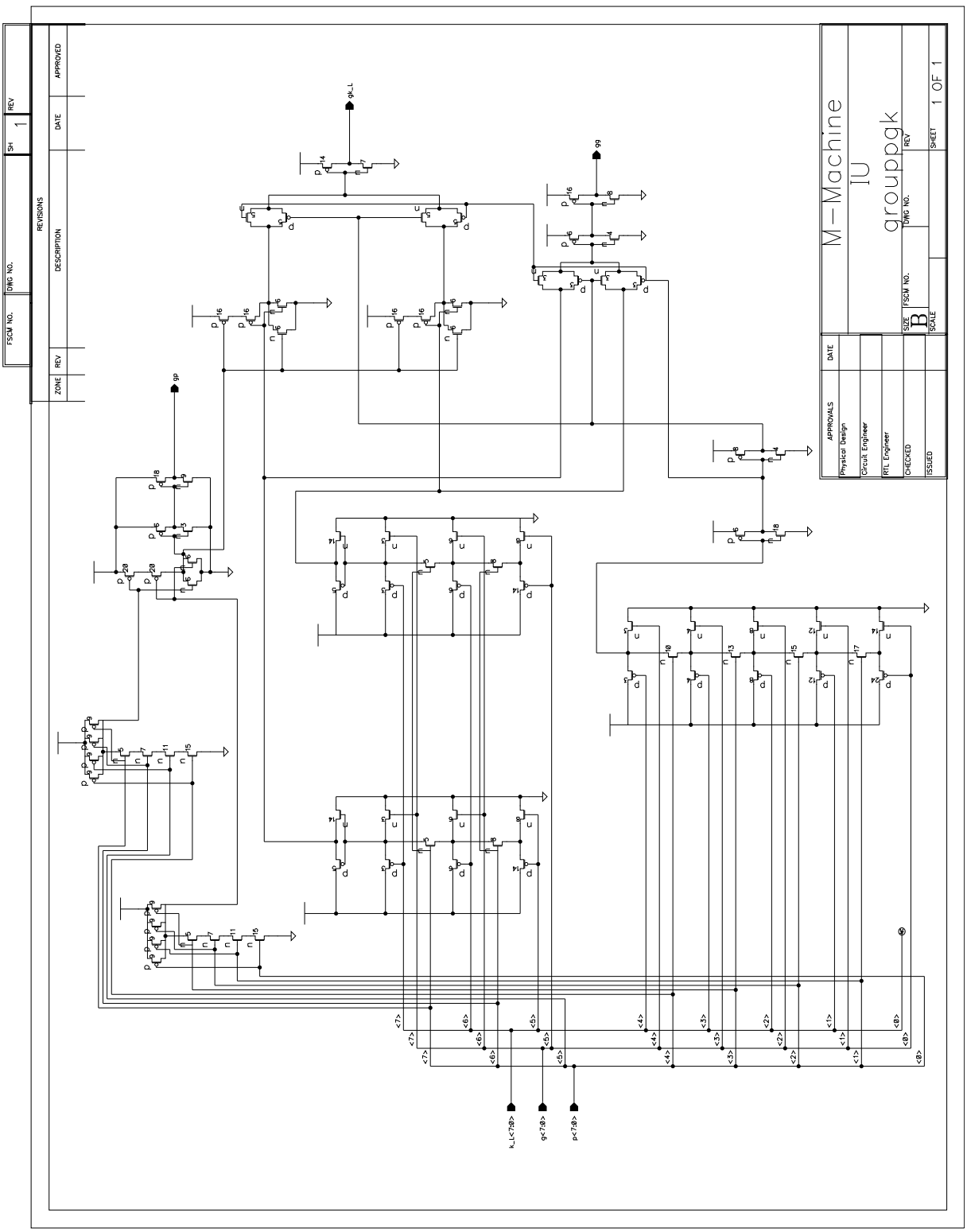
- Description:
- Errval/Exceptions:
  - Generates *no* Errval's.
  - Causes an exception if
    1. *src1* (ie. Virtual Addr) is an ErrVal.
    2. *src2* (ie. dest IP) is not *Enter Message* type.
    3. IP is not execute-system.

Appendix B

Schematics

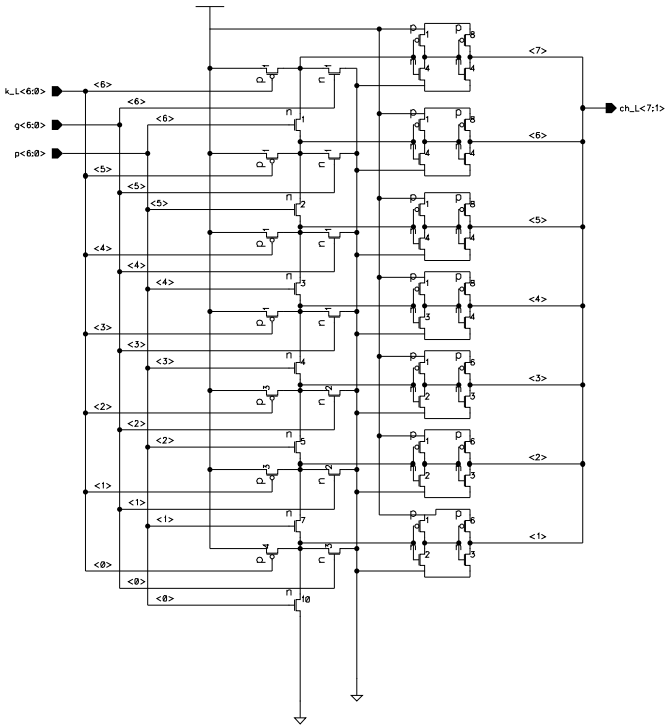

M-Machine
IU
iu
FSCM NO.
DWG NO.
REV
SIZE
B
SCALE
SHEET 1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH 1
REV

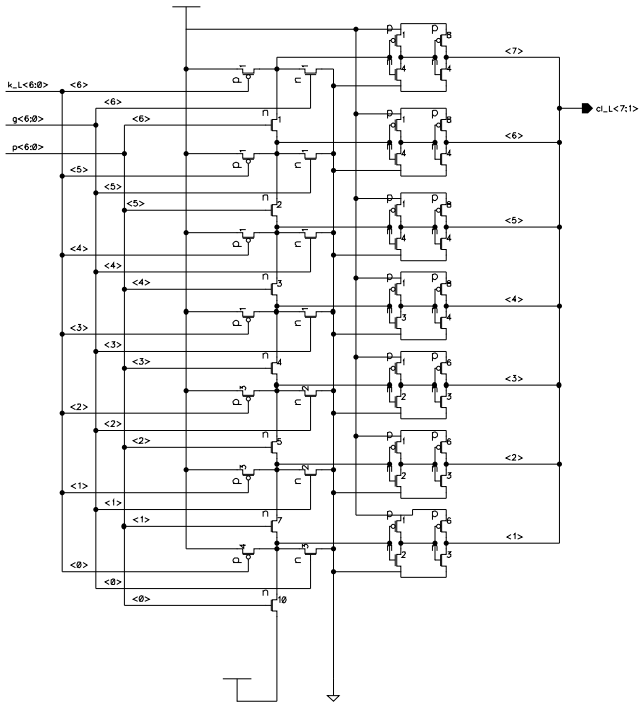

M-Machine
IU
gcc
SIZE
B
SCALE
DWG NO.
REV
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
1
REV
c<8:5>
c_L<4:1>
c<8>
c<7>
c<6>
c<5>
c_L<4>
c_L<3>
c_L<2>
c_L<1>
k_L<7:0>
g<7:0>
p<7:0>
carryin


M-Machine
IU
gruppgk
REVISIONS
DESCRIPTION
DATE
APPROVED
ZONE
REV
FSCM NO.
DWG NO.
SH 1
SIZE B
SCALE
SHEET 1 OF 1
APPROVALS
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
gk_L
gp
99
k_L<7:0>
g<7:0>
p<7:0>

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED





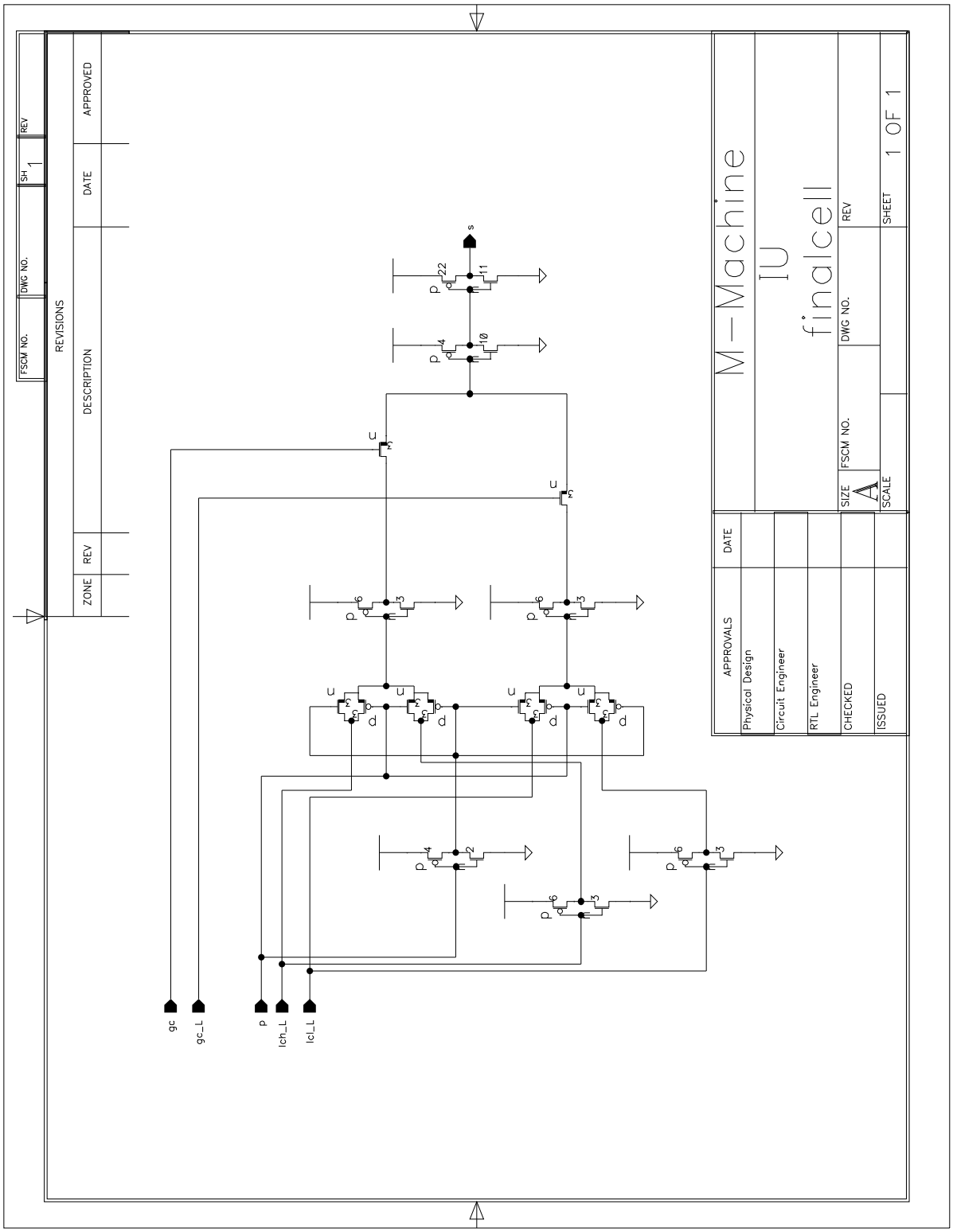
APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

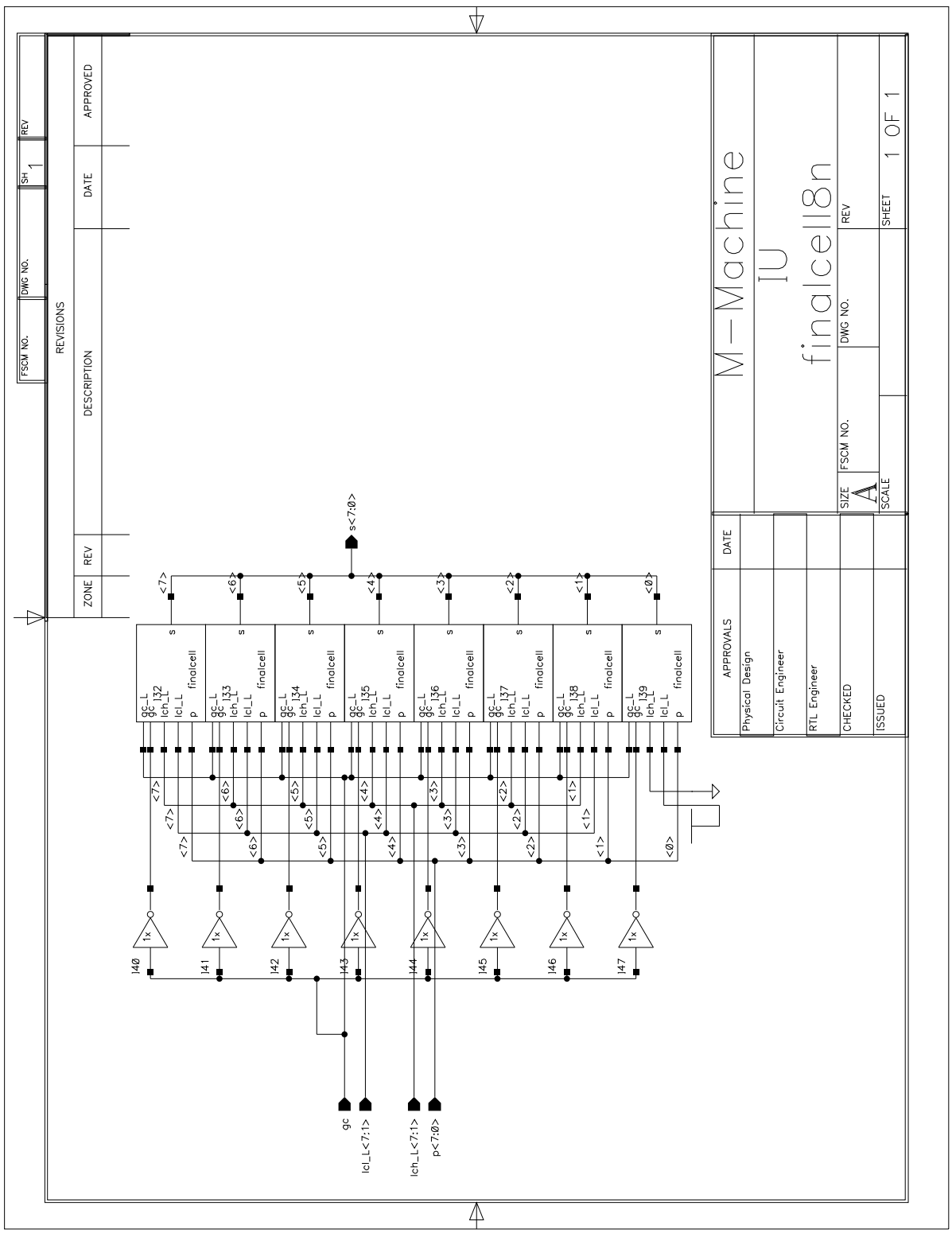
M—Machine

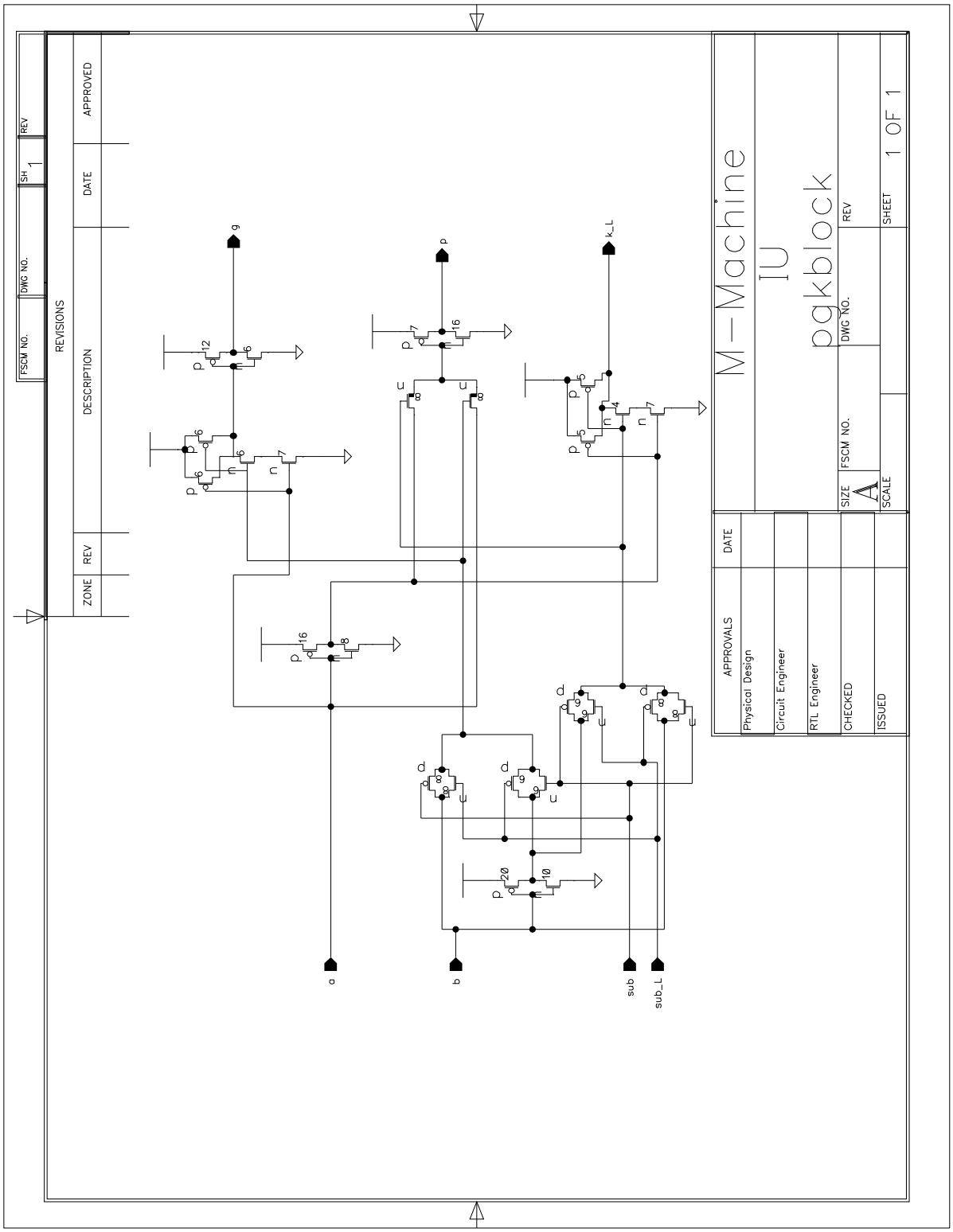
IU

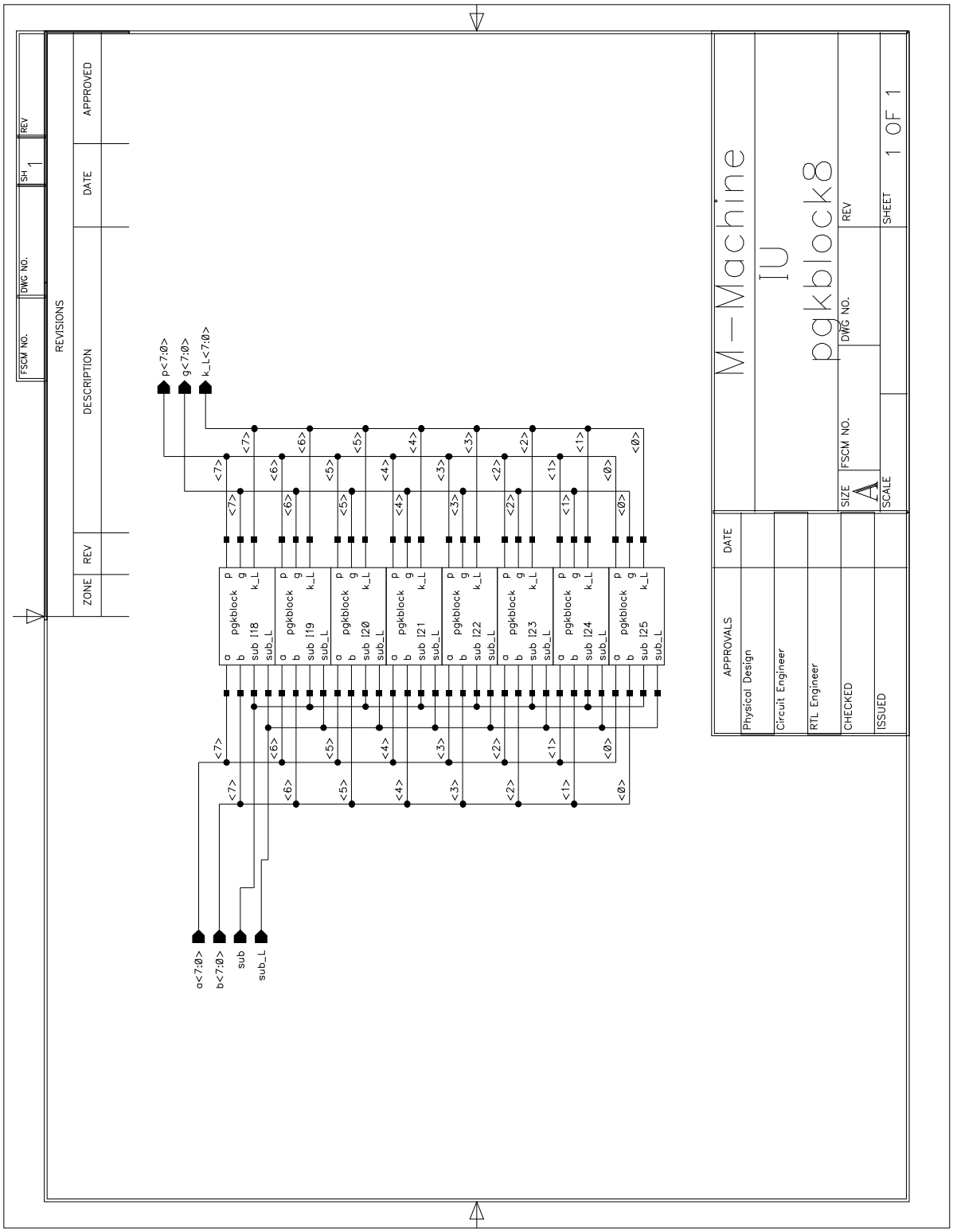
Icc

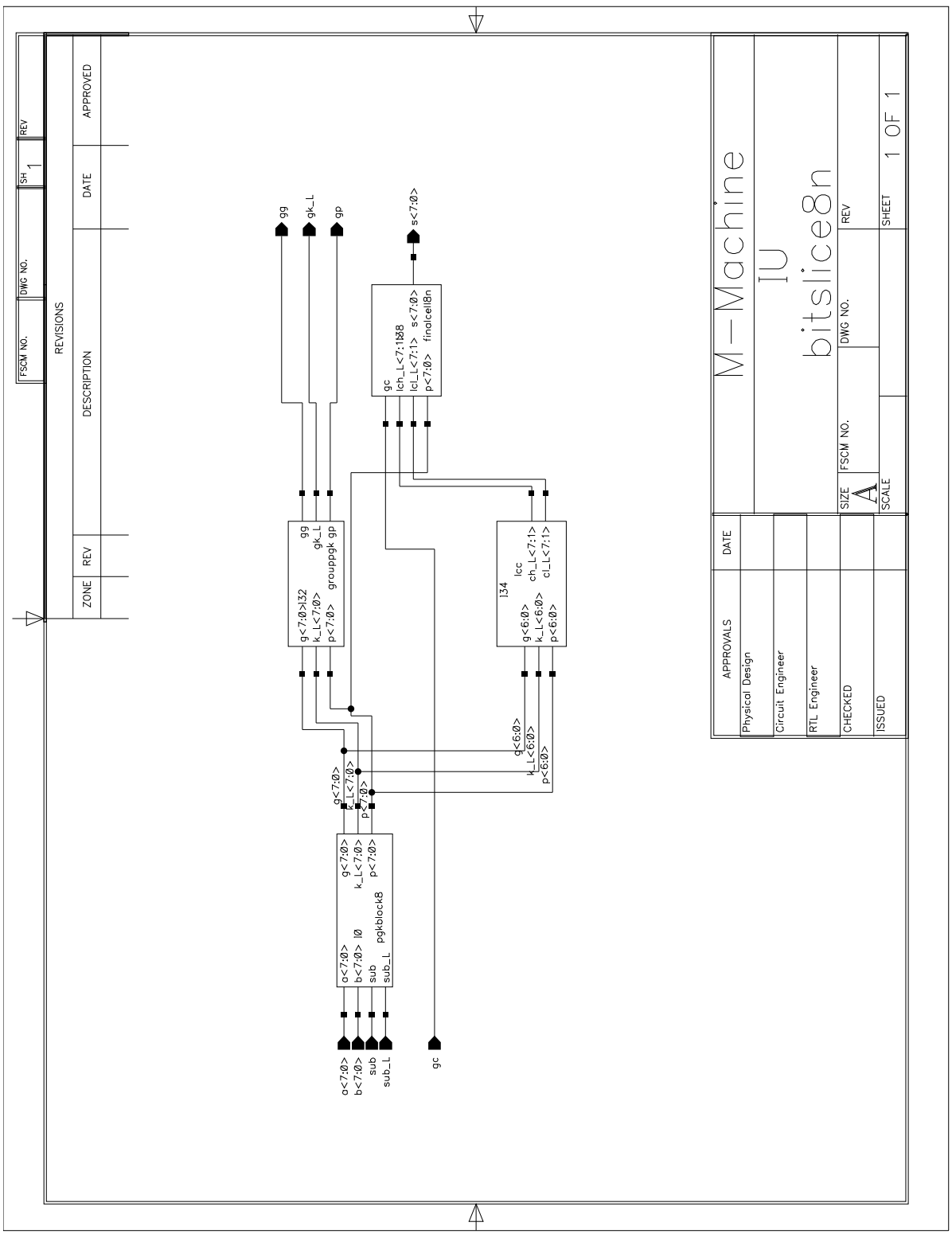
SIZE	FSCM NO.	DWG NO.	REV
B			
SCALE			SHEET 1 OF 1

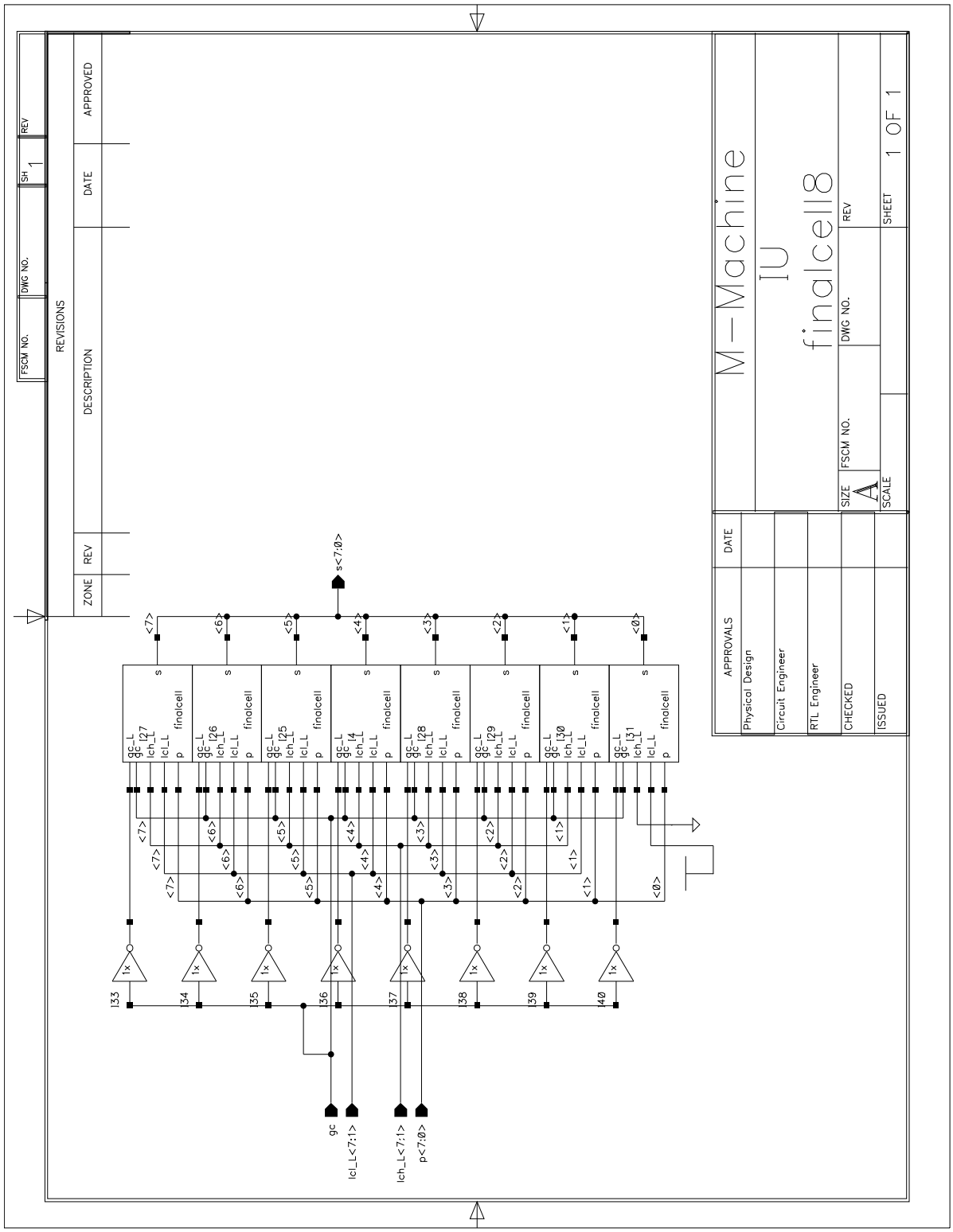

M-Machine
IU
finalcell
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
SIZE
SCALE
SHEET 1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
s
gc
gc_L
p
lch_L
lcl_L

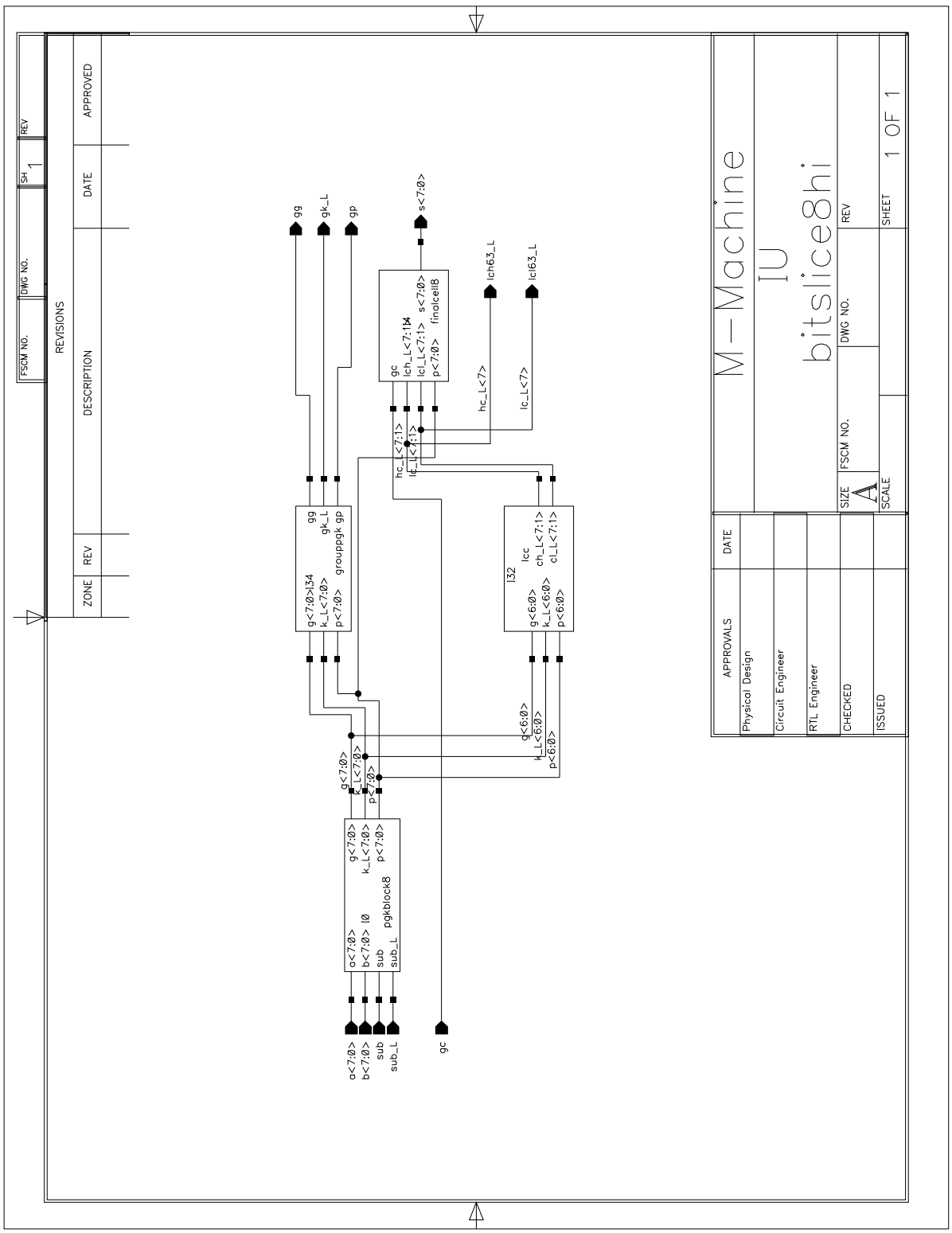

M-Machine
IU
finalcell8n
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A1
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH 1
REV
s<7:0>
gc
lcl_L<7:1>
lch_L<7:1>
p<7:0>
I40
I41
I42
I43
I44
I45
I46
I47
I32
I33
I34
I35
I36
I37
I38
I39
finalcell
1x

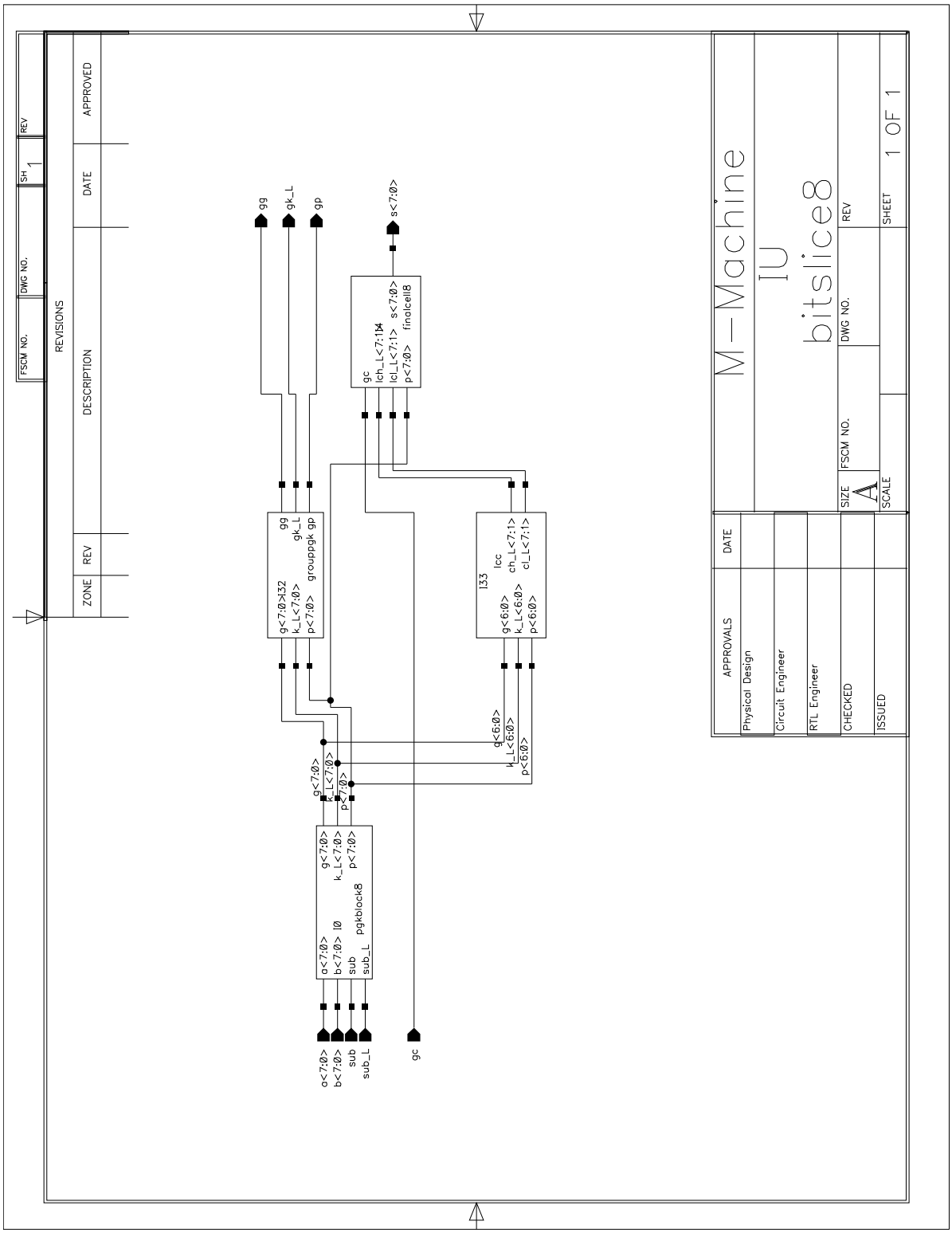

M-Machine
IU
pgkblock
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
APPROVALS
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
g
p
k_L
a
b
sub
sub_L

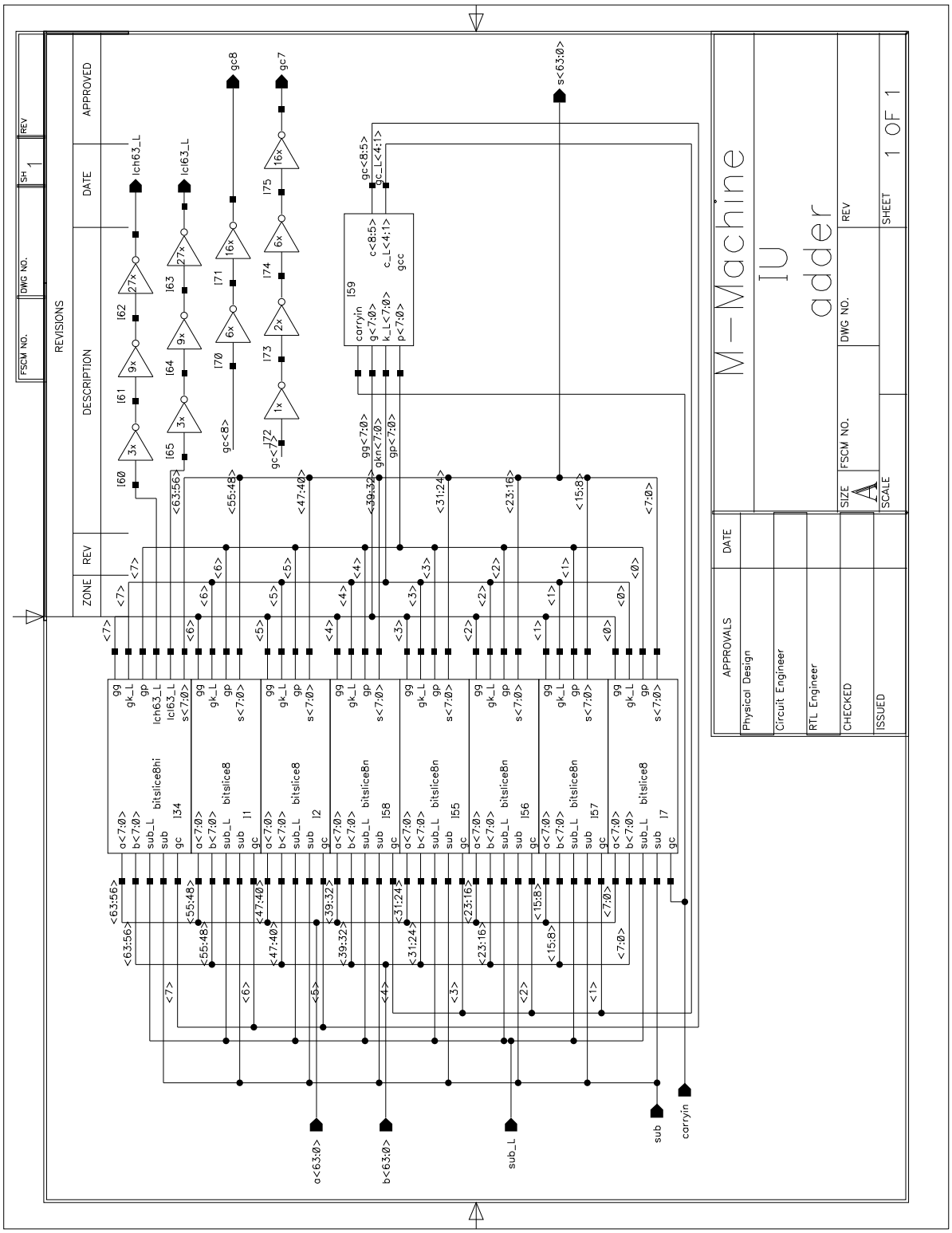

M-Machine
IU
pgkblock8
REVISIONS
DESCRIPTION
DATE
APPROVED
ZONE
REV
FSCM NO.
DWG NO.
SH 1
REV
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE A
SCALE
SHEET 1 OF 1
p<7:0>
g<7:0>
k_L<7:0>
o<7:0>
b<7:0>
sub
sub_L
pgkblock
sub 118
sub 119
sub 120
sub 121
sub 122
sub 123
sub 124
sub 125

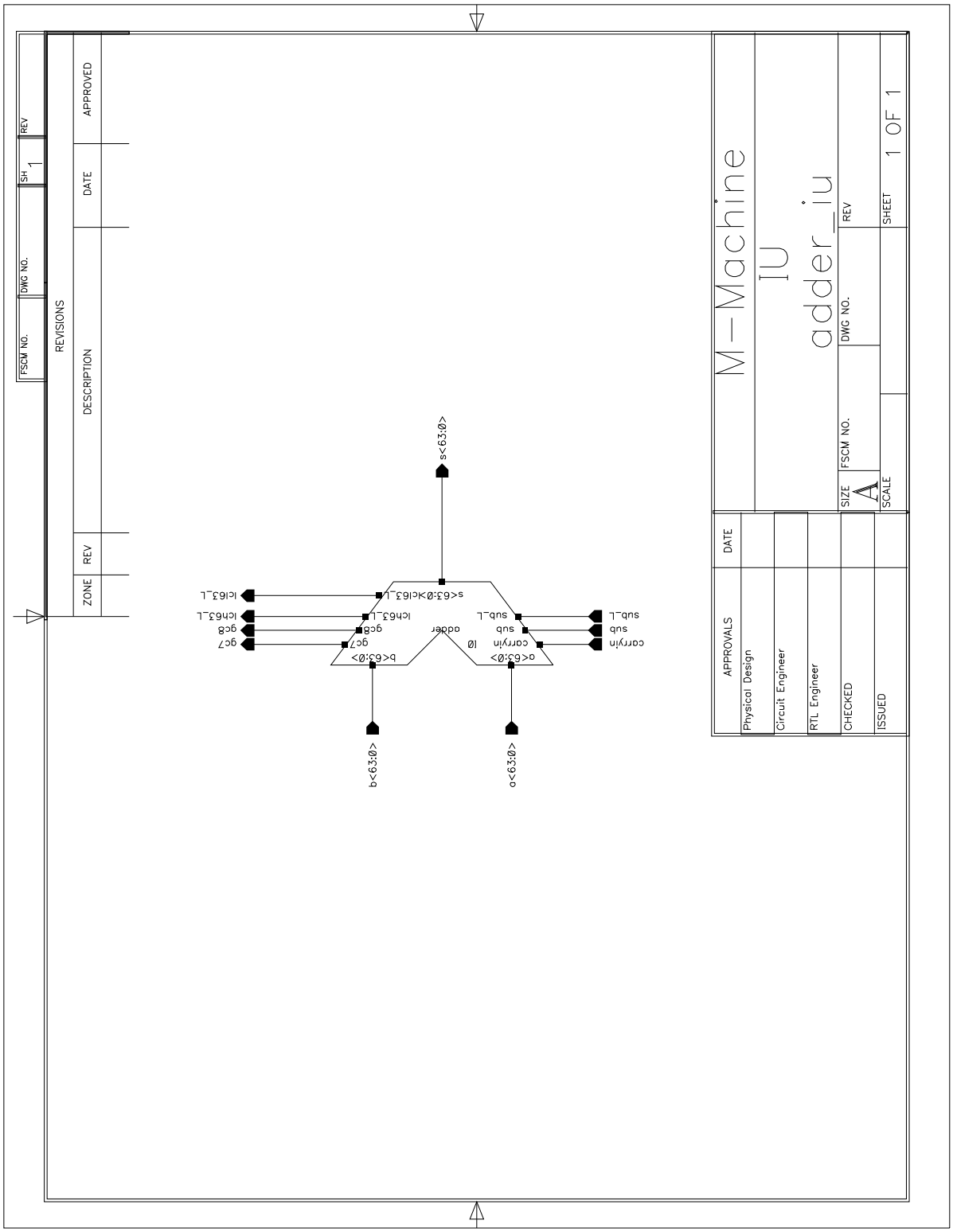

M-Machine
IU
bitslice8n
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH 1
REV
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
gg
gk_L
gp
s<7:0>
gc
ich_L<7:1>
icl_L<7:1>
p<7:0>
s<7:0>
finalcell8n
g<7:0>
k_L<7:0>
p<7:0>
groupgk
134
lcc
g<6:0>
k_L<6:0>
p<6:0>
ch_L<7:1>
cl_L<7:1>
a<7:0>
b<7:0>
sub
sub_L
I0
pgkblock8
sub_L

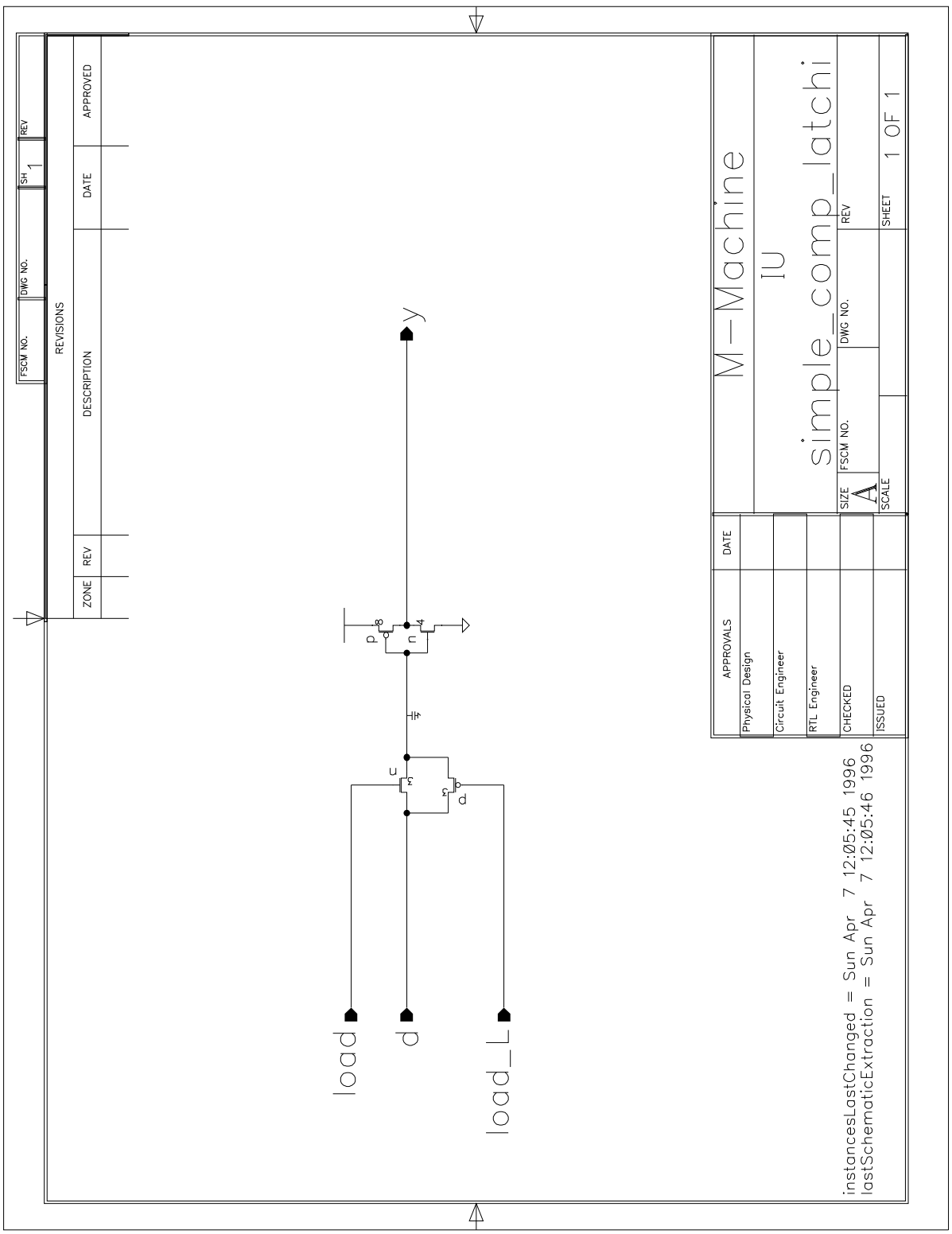

M-Machine
IU
finalcell8
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
s<7:0>
gc
lcl_L<7:1>
lch_L<7:1>
p<7:0>

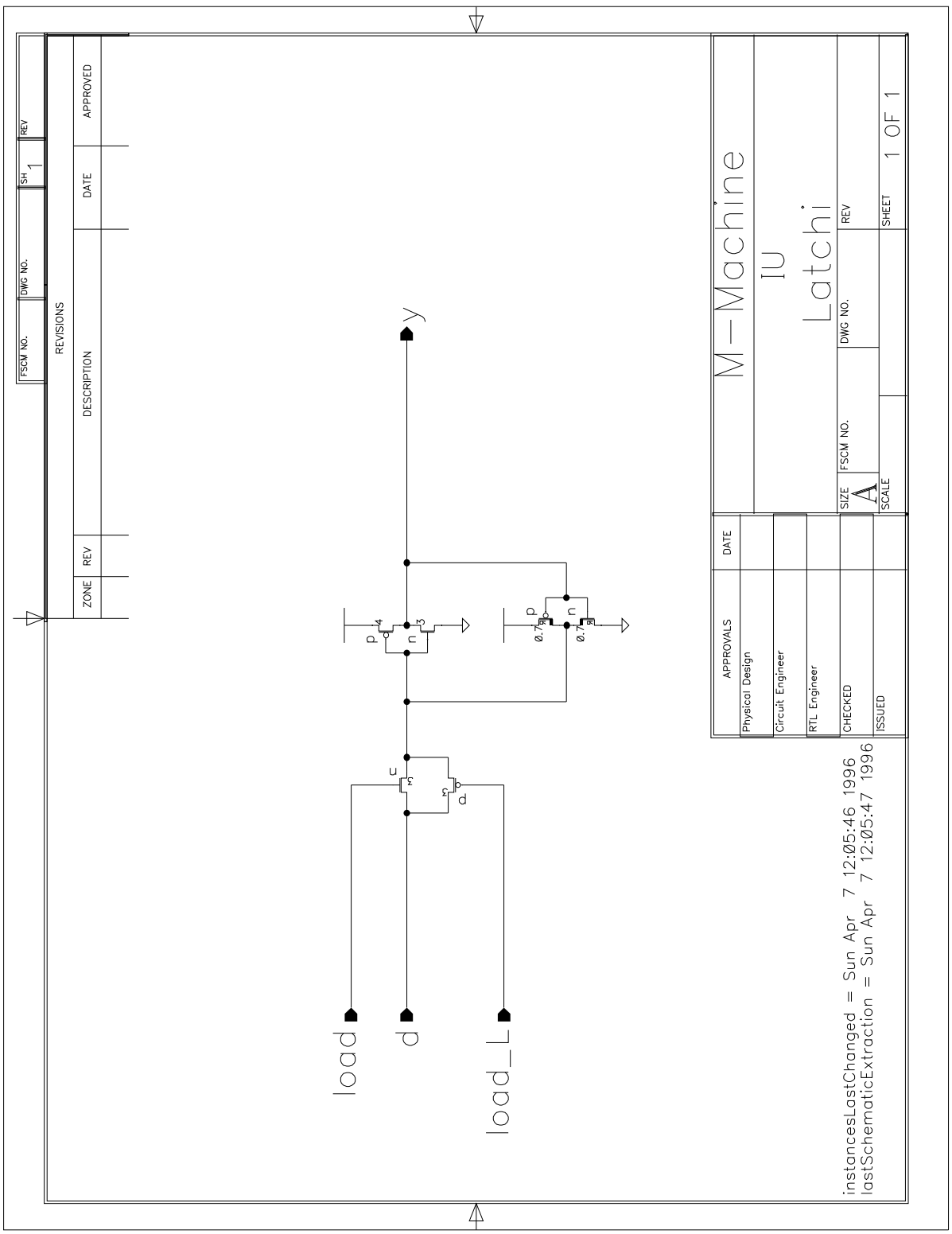

M-Machine
IU
bitslice8hi
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
gg
gk_L
gp
s<7:0>
lch63_L
lcl63_L
hc_L<7>
lc_L<7>
gc
lch_L<7:1>
lcl_L<7:1>
p<7:0>
finalcell8
hc_L<7:1>
lc_L<7:1>
g<7:0>:134
k_L<7:0>
p<7:0>
grouppgk
l32
lcc
ch_L<7:1>
cl_L<7:1>
g<6:0>
k_L<6:0>
p<6:0>
g<7:0>
k_L<7:0>
p<7:0>
a<7:0>
b<7:0>
l0
sub
sub_L
pgkblock8
sub


M-Machine
IU
bitslice8
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A1
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH 1
REV
gg
gk_L
gp
s<7:0>
gc
ich_L<7:1>
icl_L<7:1>
p<7:0>
finalcell8
g<7:0>
k_L<7:0>
p<7:0>
grouppgk
133
icc
ch_L<7:1>
cl_L<7:1>
g<6:0>
k_L<6:0>
p<6:0>
o<7:0>
b<7:0>
sub
sub_L
10
pgkblock8


M-Machine
IU
adder
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
bitslice8hi
bitslice8
bitslice8n
carryin
a<63:0>
b<63:0>
sub_L
sub
carryin


M-Machine
IU
adder_iu
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
1
REV
s<63:0>
s<63:0>lcl63_L
lcl63_L
lch63_L
gc8
gc7
b<63:0>
adder
I0
a<63:0>
carryin
sub
sub_L
b<63:0>
a<63:0>


y
load
d
load_L
p
n
c
d
M-Machine
IU
simple_comp_latchi
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A1
SCALE
SHEET 1 OF 1
instancesLastChanged = Sun Apr 7 12:05:45 1996
lastSchematicExtraction = Sun Apr 7 12:05:46 1996

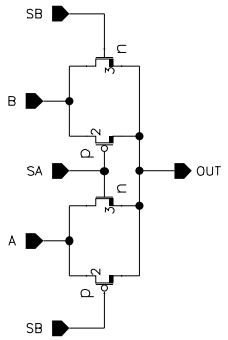


M-Machine
IU
Latchi
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
1
REV
y
load
d
load_L
instancesLastChanged = Sun Apr 7 12:05:46 1996
lastSchematicExtraction = Sun Apr 7 12:05:47 1996

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



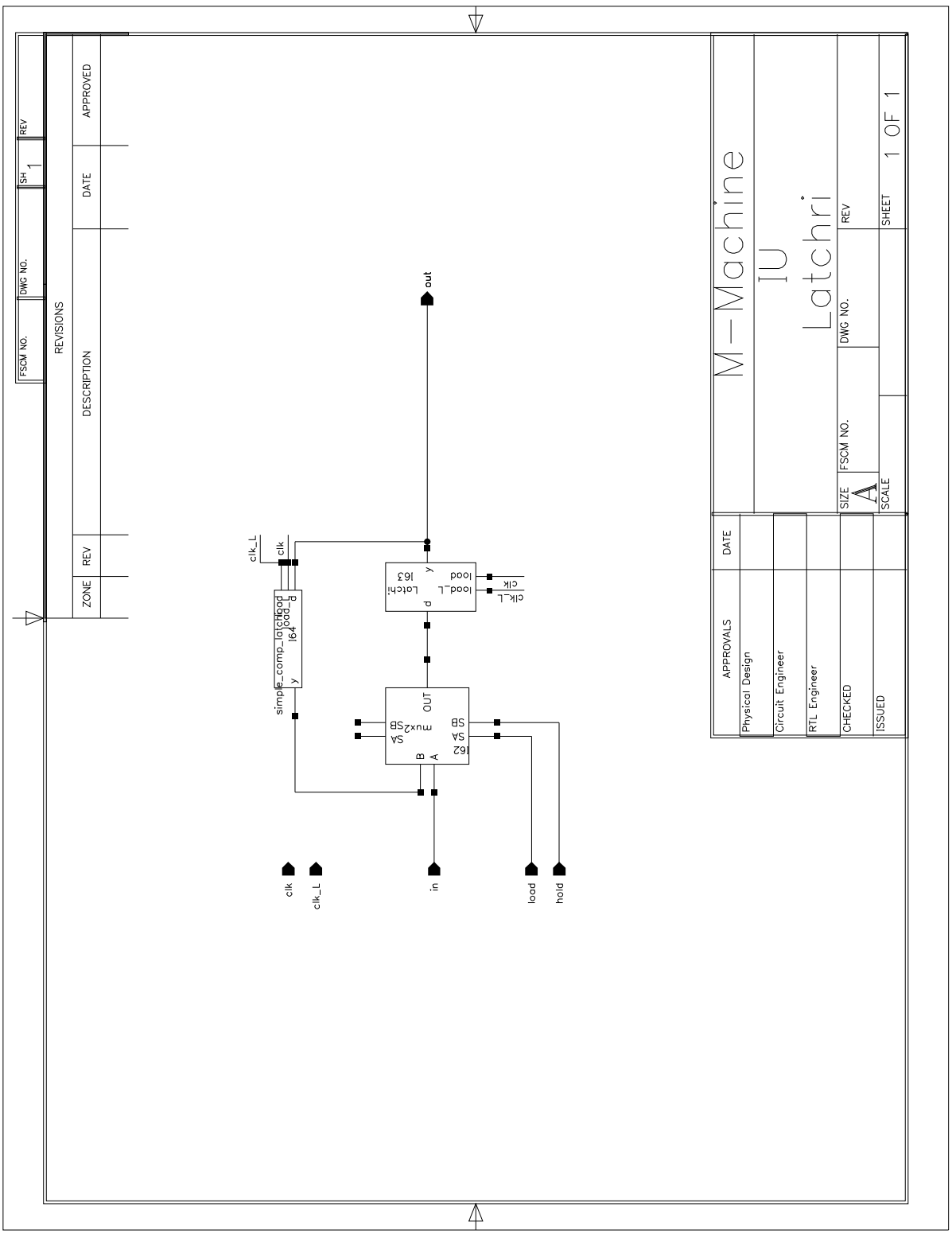
APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

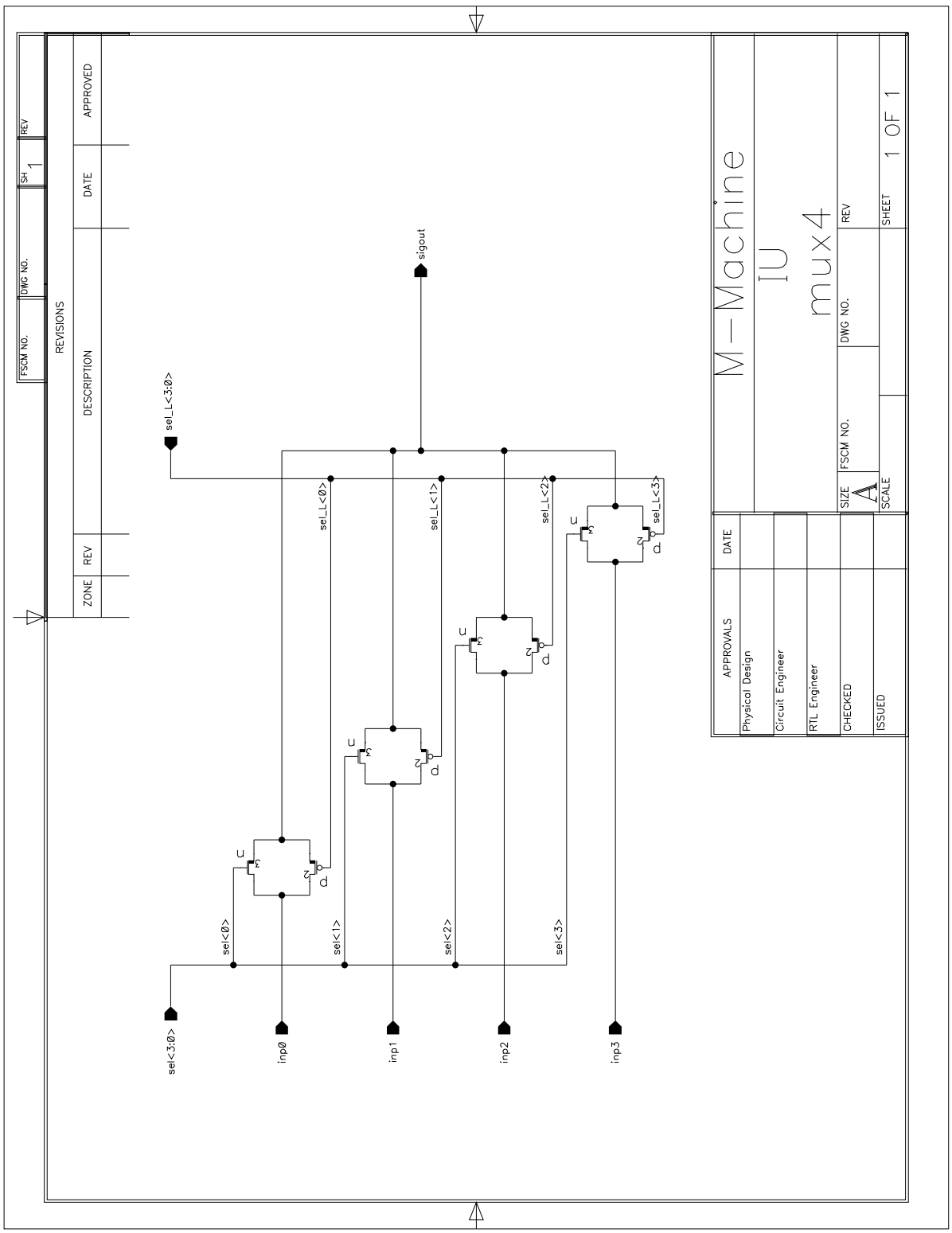
M-Machine

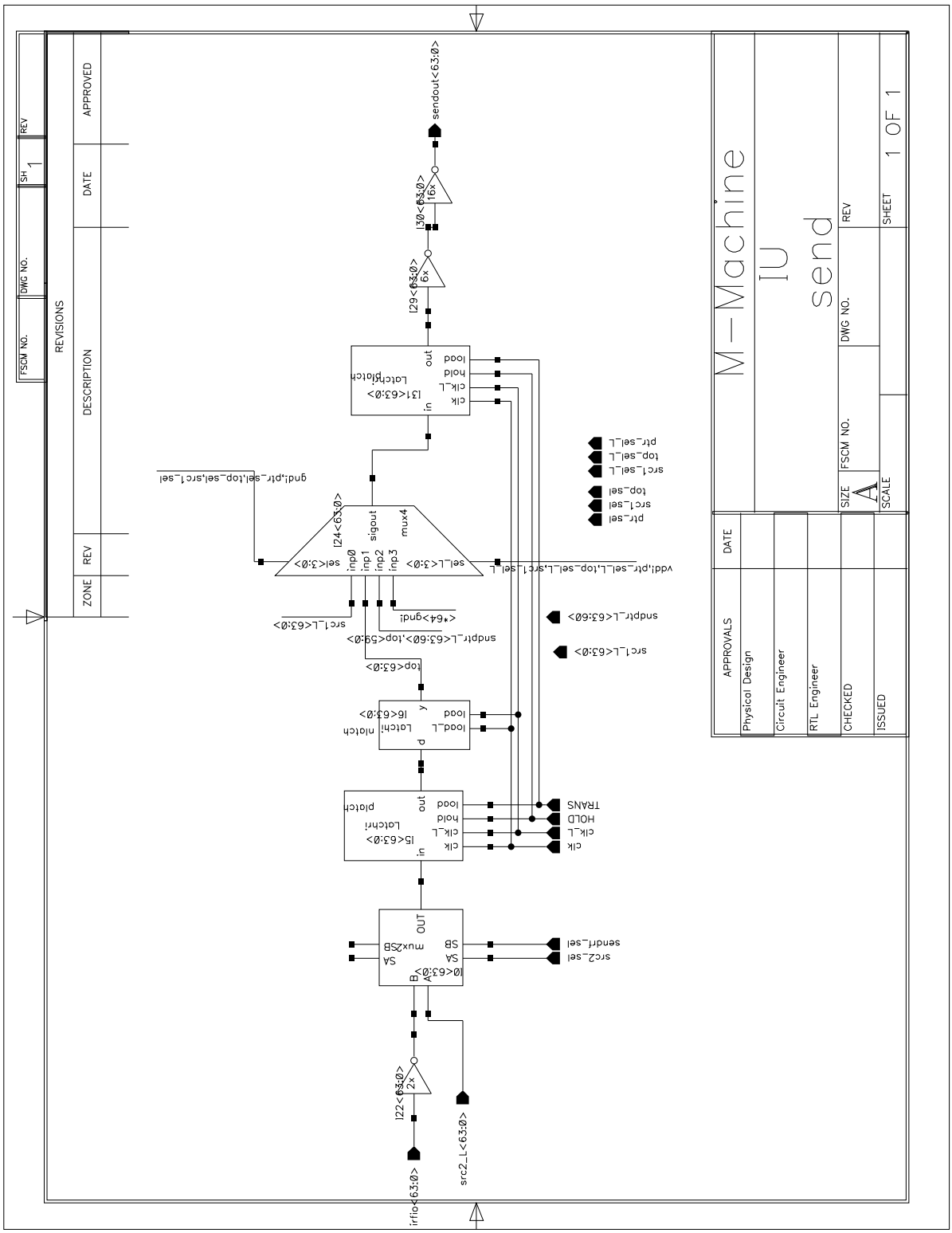
IU

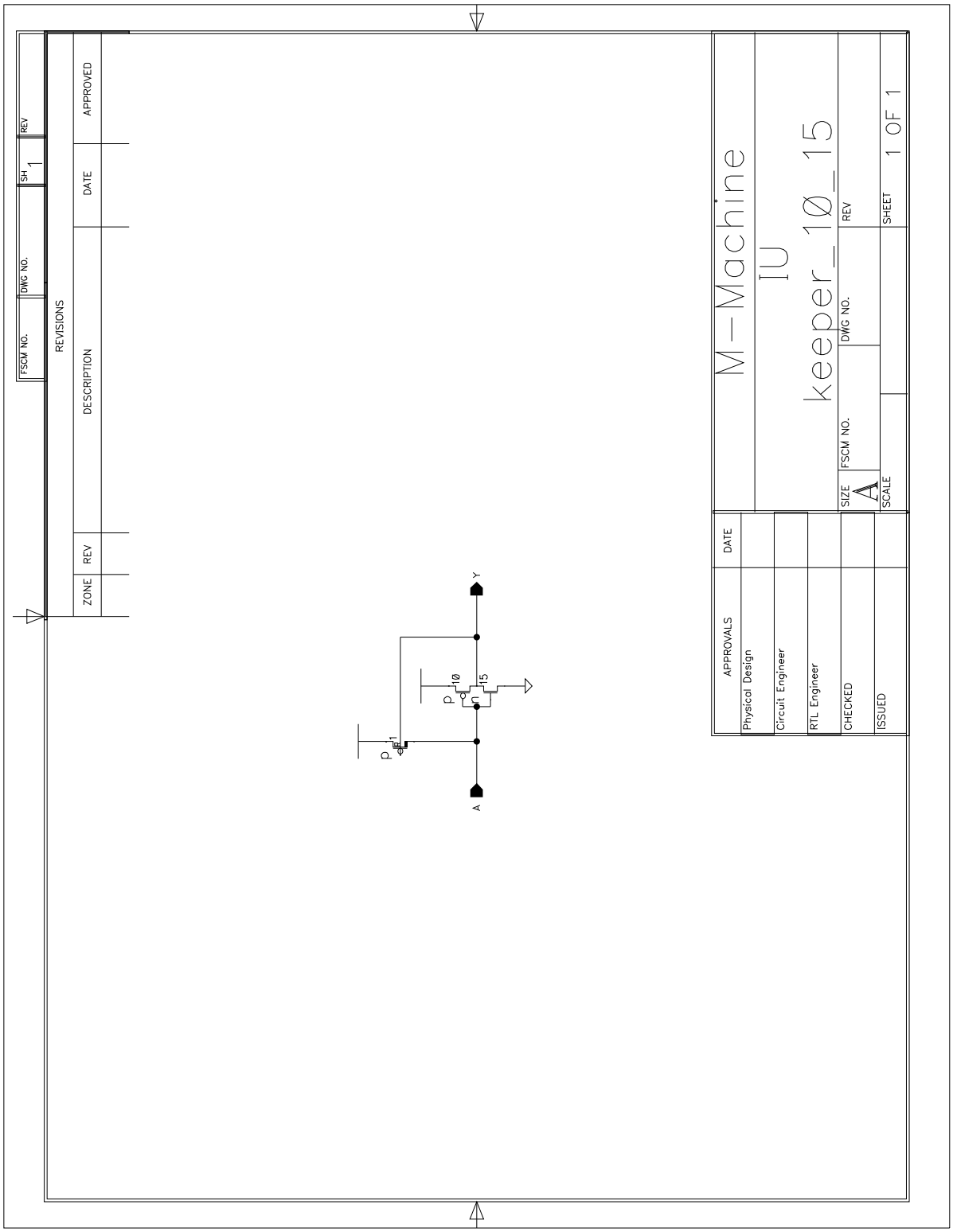
mux2

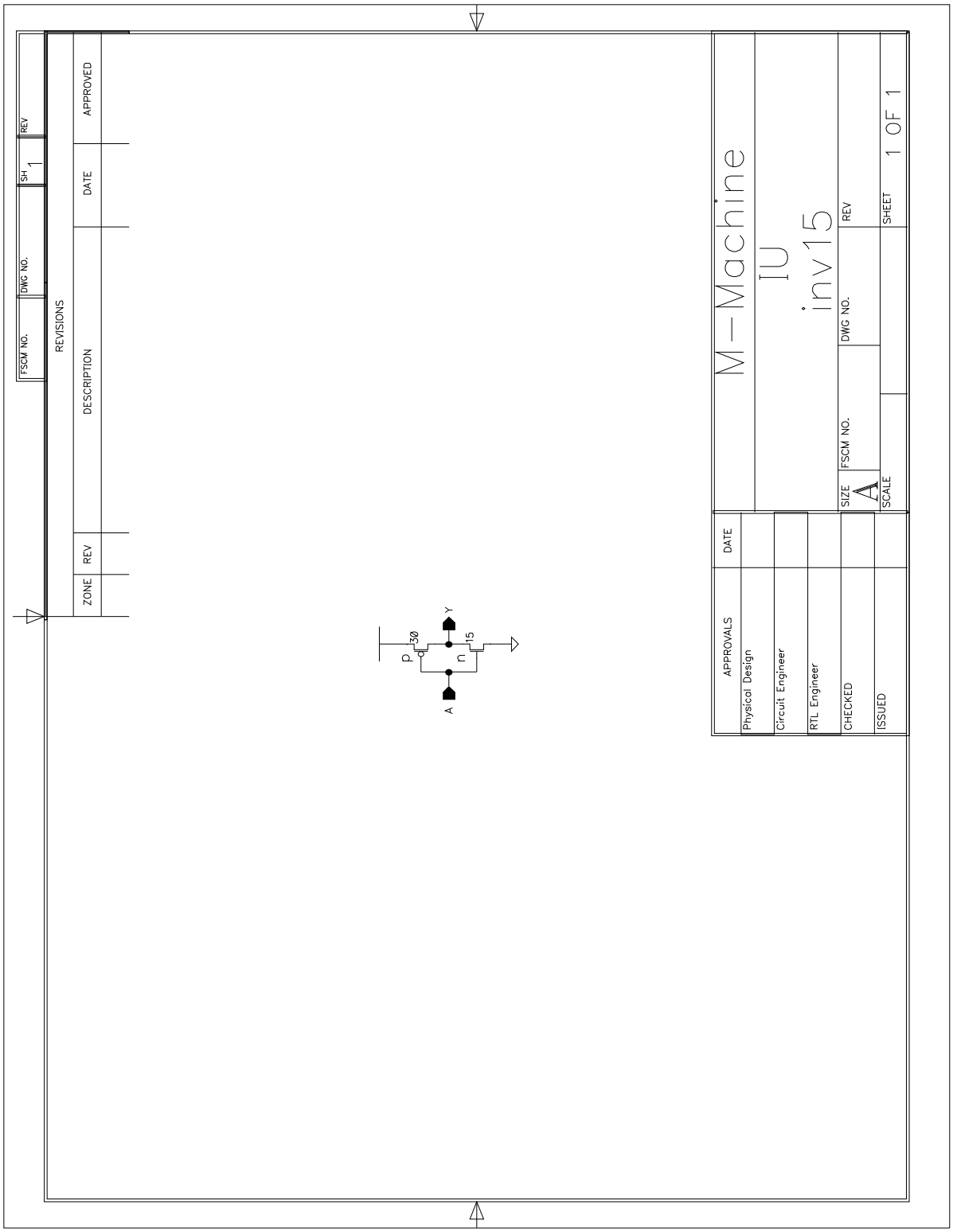
SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE			SHEET 1 OF 1


M-Machine
IU
Latchri
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
1
REV
out
clk_L
clk
simple_comp_latchload
load_d
I64
y
Latchi
I63
load_L
load
d
y
clk_L
clk
OUT
mux2
SB
SA
B
A
I62
SB
SA
clk
clk_L
in
load
hold


sigout
sel_L<3:0>
sel_L<0>
sel_L<1>
sel_L<2>
sel_L<3>
sel<3:0>
sel<0>
sel<1>
sel<2>
sel<3>
inp0
inp1
inp2
inp3
M-Machine
IU
mux4
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
SCALE
SHEET 1 OF 1


M-Machine
IU
send
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
1
REV
sendout<63:0>
I30<63:0>
16x
I29<63:0>
6x
I31<63:0>
Latchi
platch
out
in
load
hold
clk_L
clk
gnd!,ptr_sel,top_sel,src1_sel
I24<63:0>
sigout
mux4
sel<3:0>
inp0
inp1
inp2
inp3
sel_L<3:0>
vdd!,ptr_sel_L,top_sel_L,src1_sel_L
ptr_sel_L
top_sel_L
src1_sel_L
top_sel
src1_sel
ptr_sel
src1_L<63:0>
<*64>gnd!
sndptr_L<63:60>,top<59:0>
sndptr_L<63:60>
src1_L<63:0>
top<63:0>
I6<63:0>
Latchi
nlatch
load_L
load
I5<63:0>
Latchri
platch
TRANS
HOLD
clk_L
clk
I0<63:0>
mux2SB
SA
SB
OUT
sendrf_sel
src2_sel
I22<63:0>
2x
src2_L<63:0>
irfio<63:0>

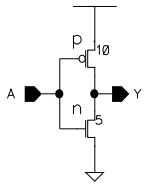

M-Machine
IU
keeper_10_15
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
1
REV
A
Y
p
1
p
10
n
15


M-Machine
IU
inv15
REVISIONS
DESCRIPTION
DATE
APPROVED
ZONE
REV
FSCM NO.
DWG NO.
REV
SH 1
SIZE
A1
SCALE
SHEET 1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
Y
A
P
n
30
15

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

M—Machine

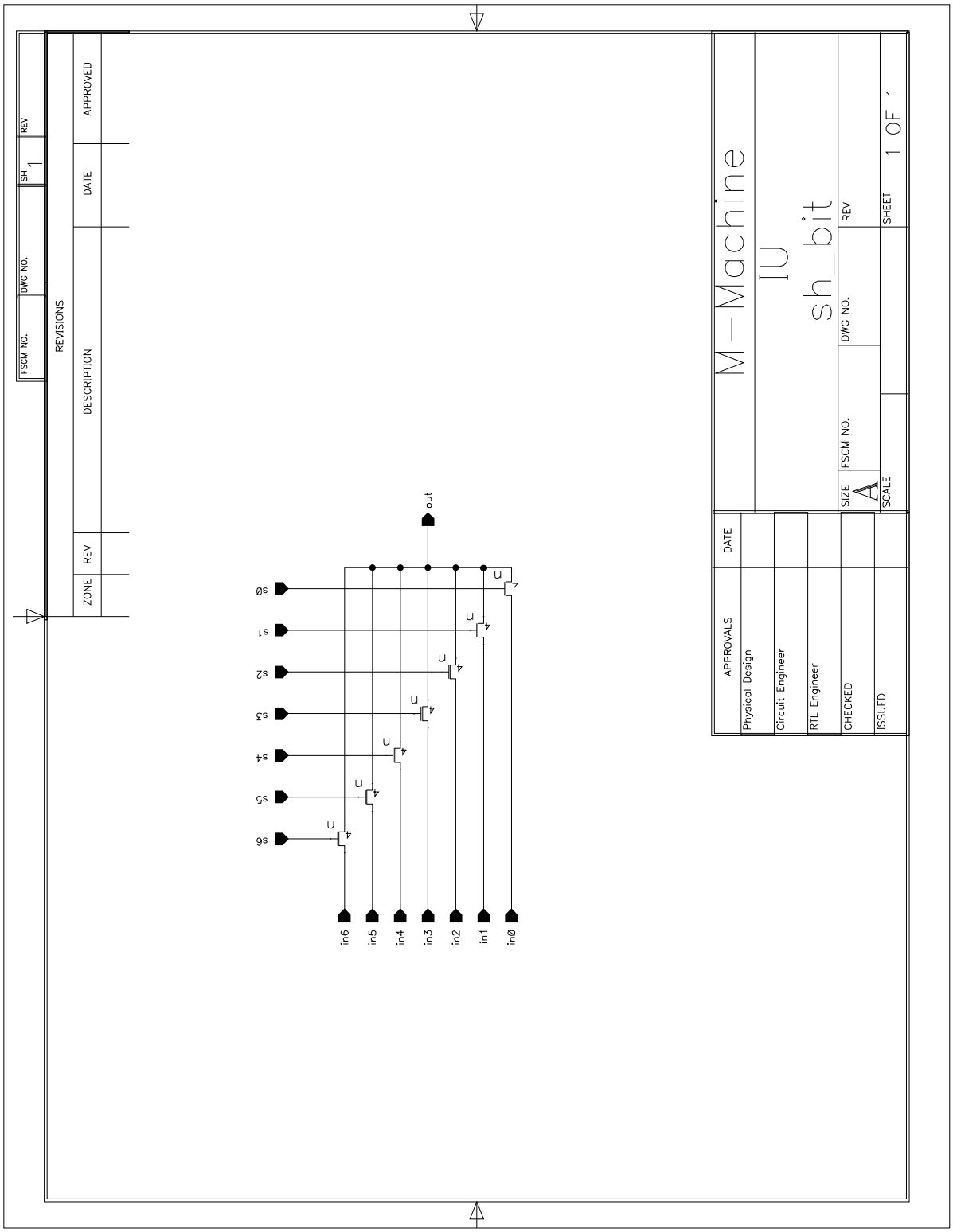
IU

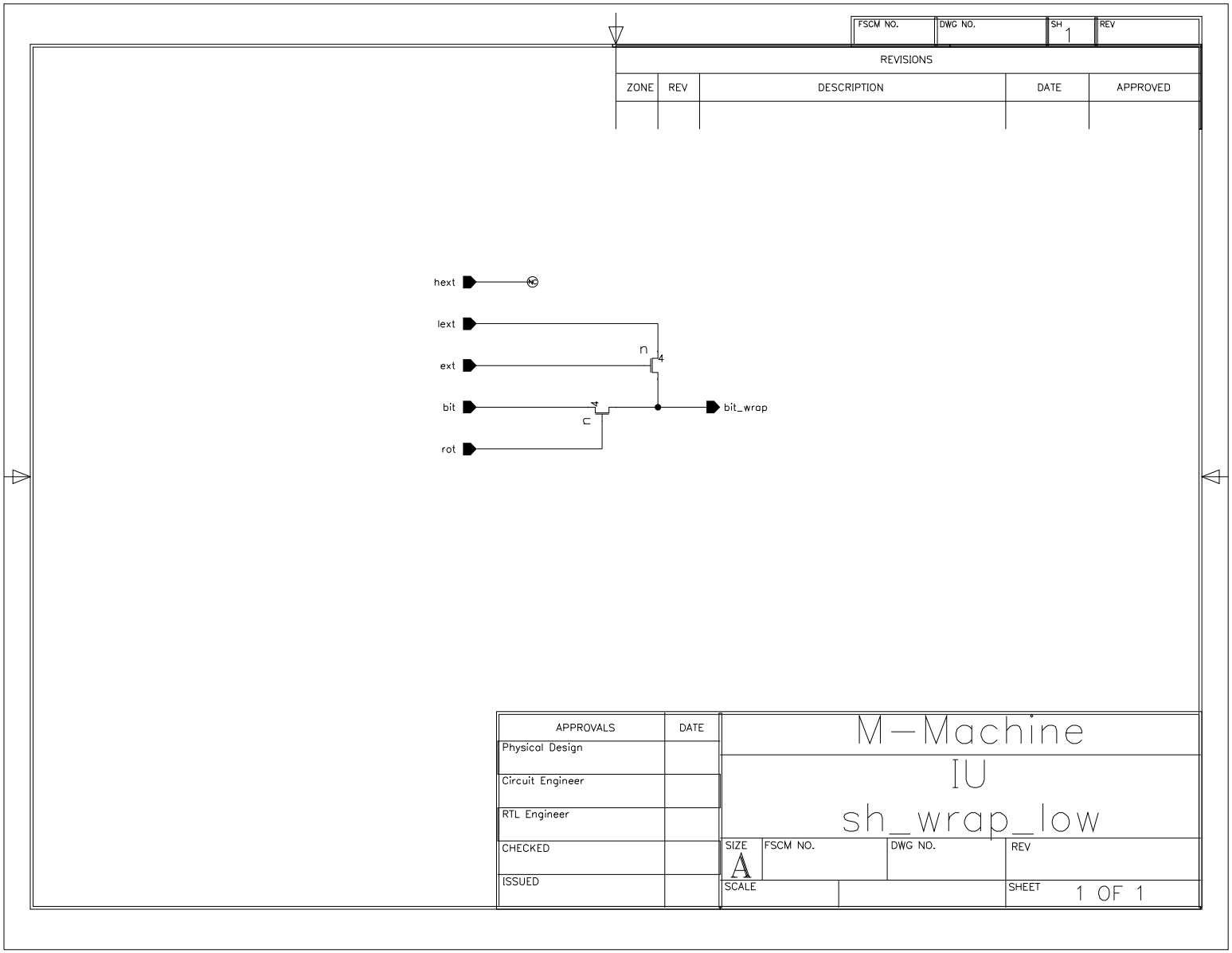
inv5

SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE

SHEET 1 OF 1

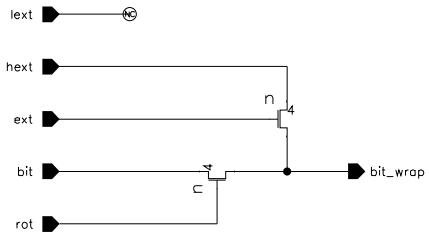

M-Machine
IU
sh_bit
REVISIONS
DESCRIPTION
DATE
APPROVED
ZONE
REV
FSCM NO.
DWG NO.
REV
SH 1
SIZE
A
SCALE
SHEET 1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
out
s0
s1
s2
s3
s4
s5
s6
in6
in5
in4
in3
in2
in1
in0


FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
hext
lext
ext
bit
rot
n
4
n
4
bit_wrap
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
M-Machine
IU
sh_wrap_low
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1

FSCM NO.	DWG NO.	SH 1	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



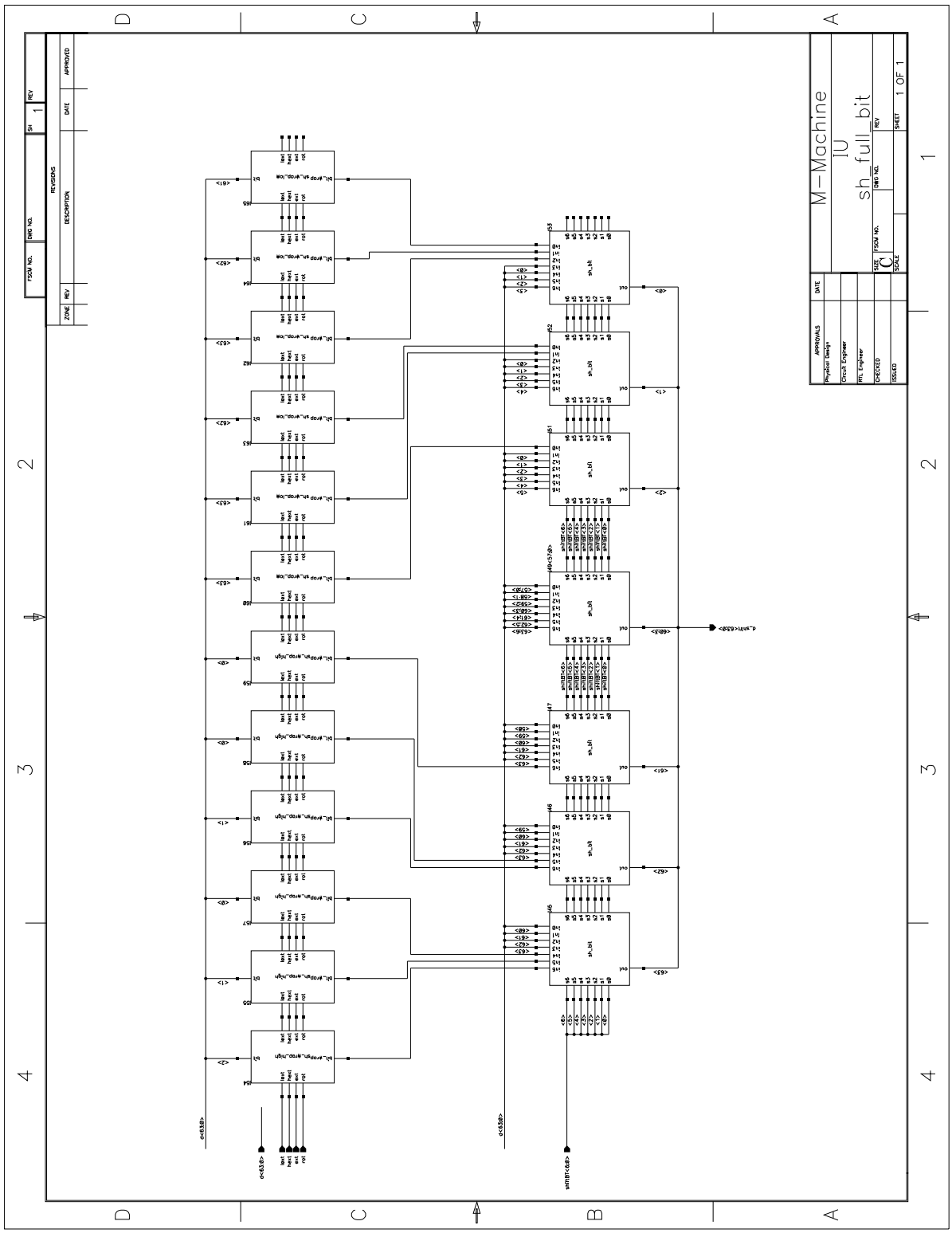
APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

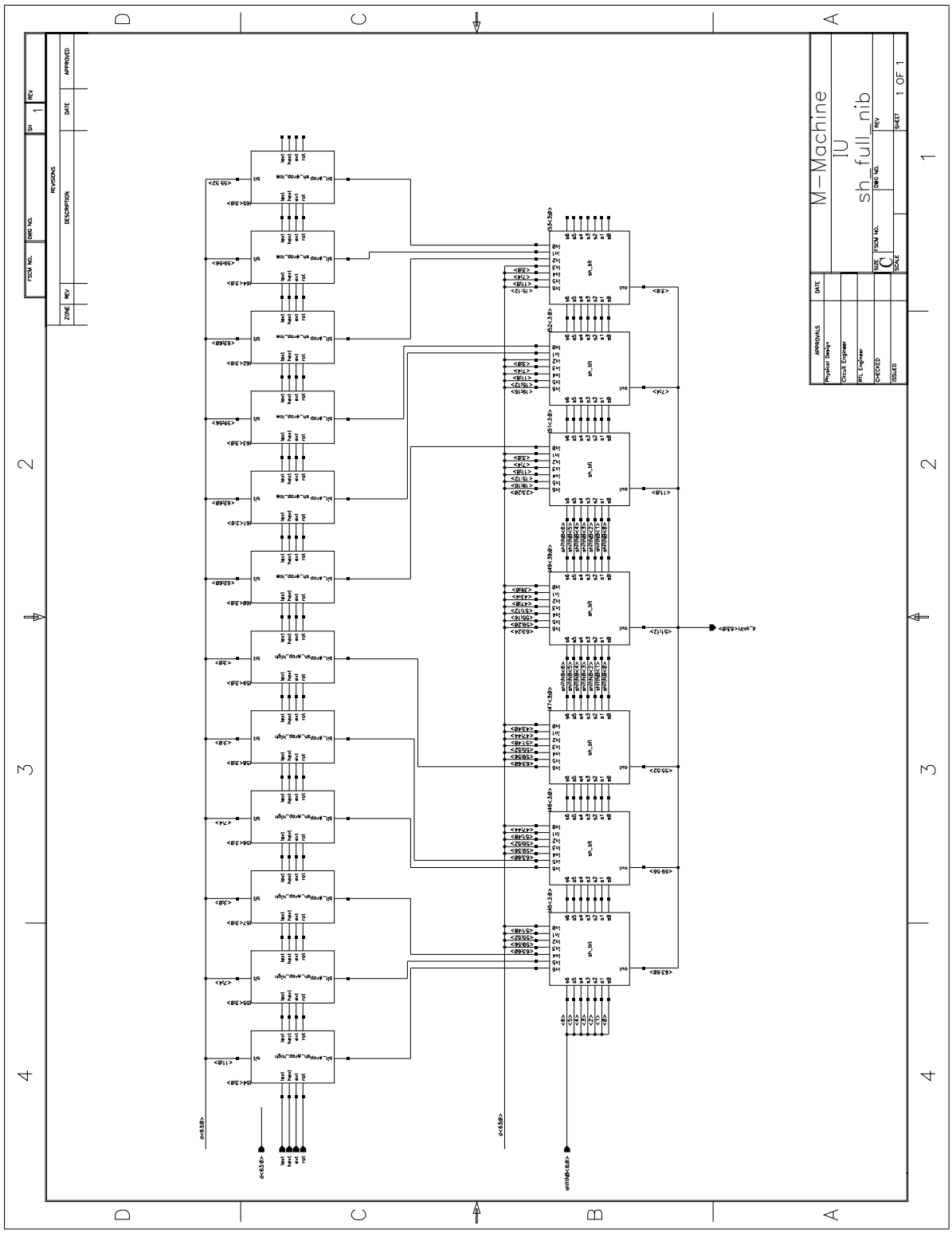
M-Machine

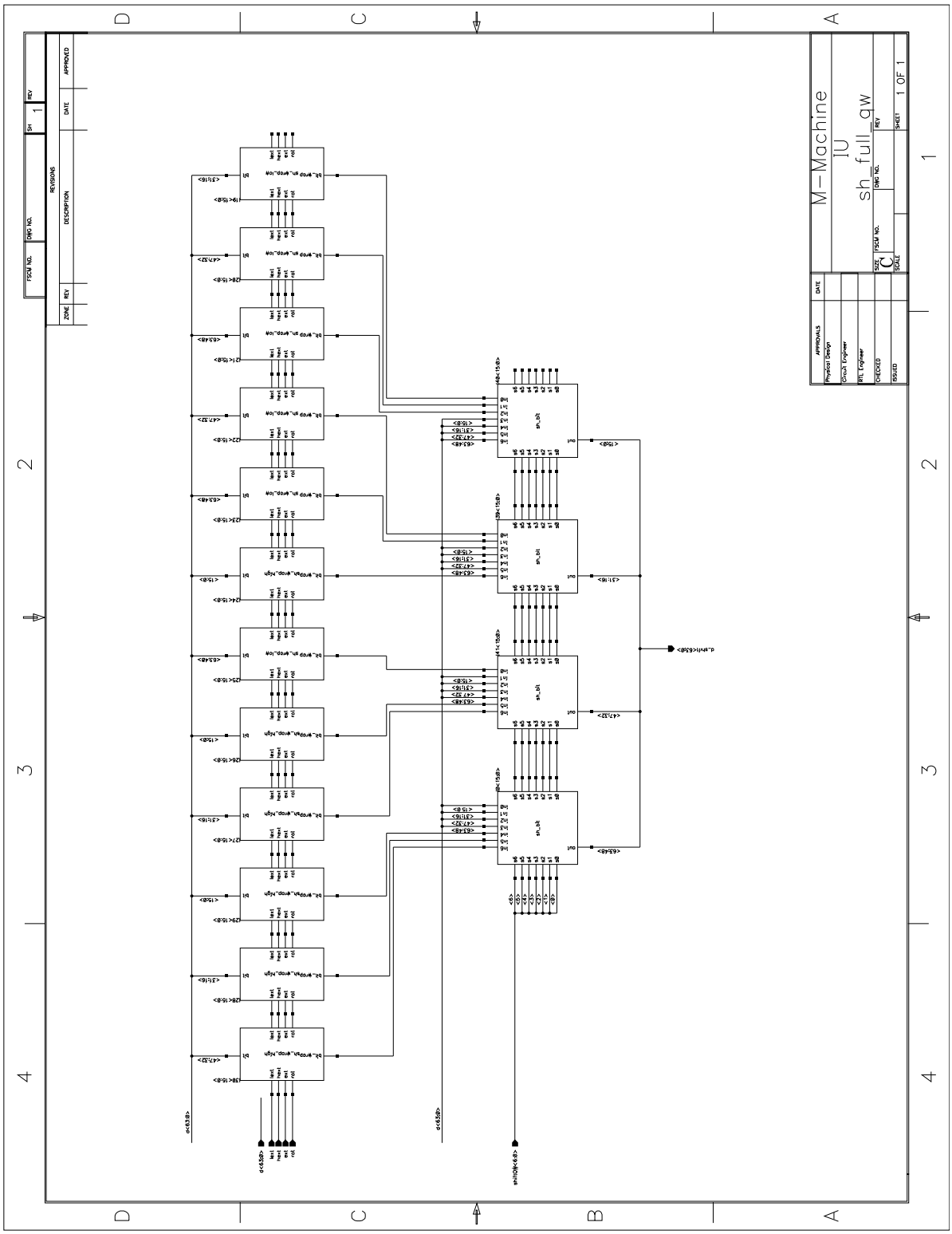
IU

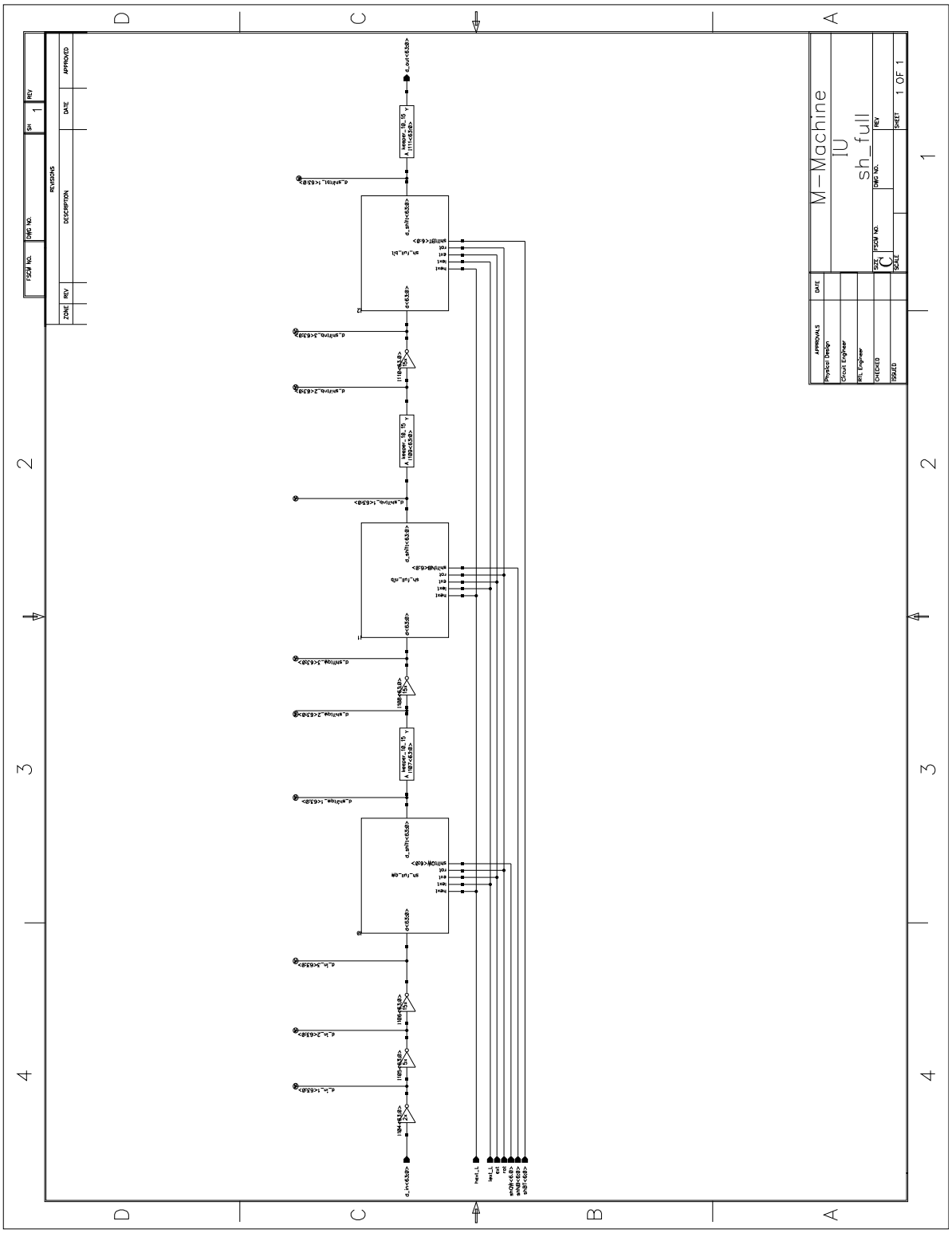
sh_wrap_high

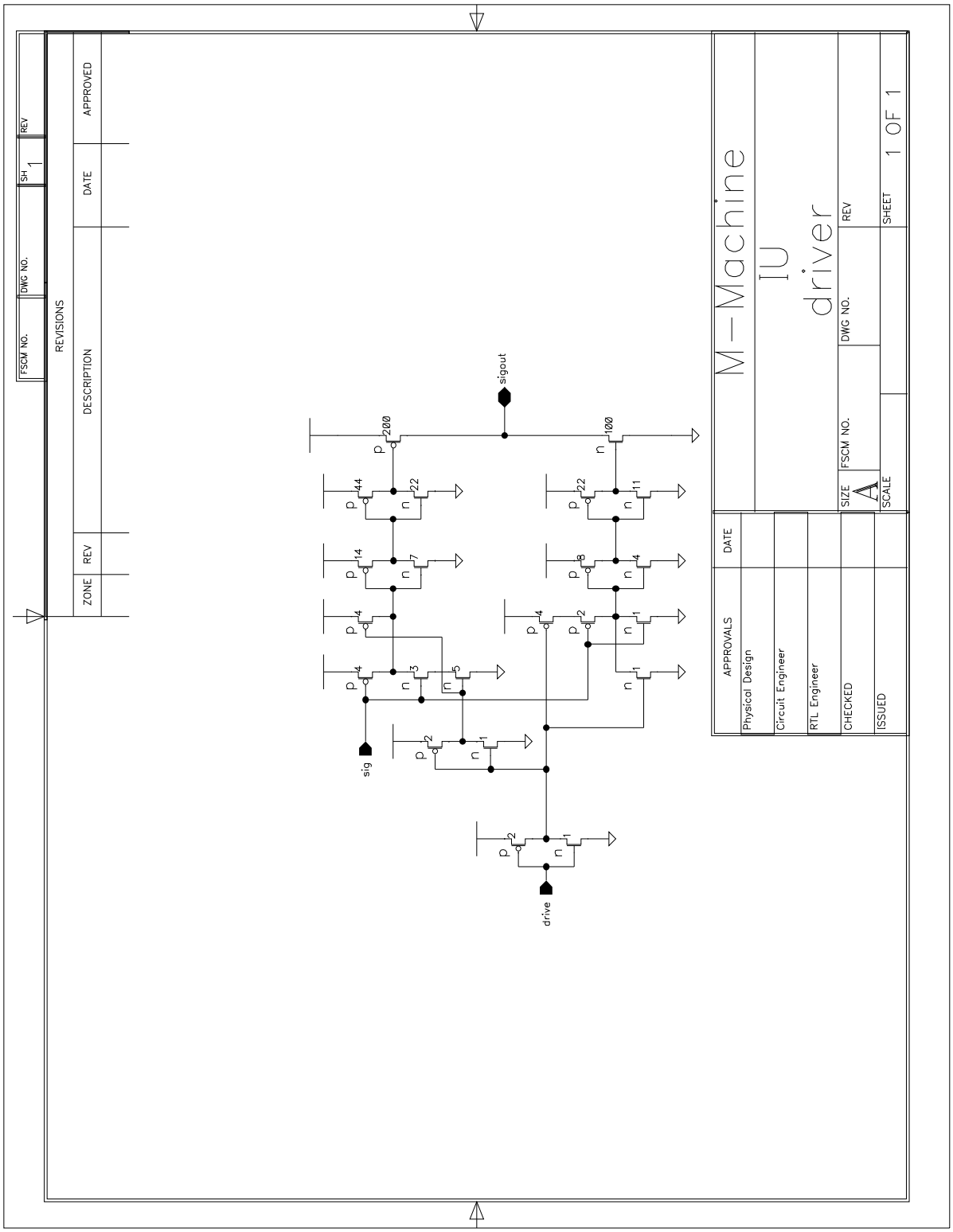
SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE			SHEET 1 OF 1

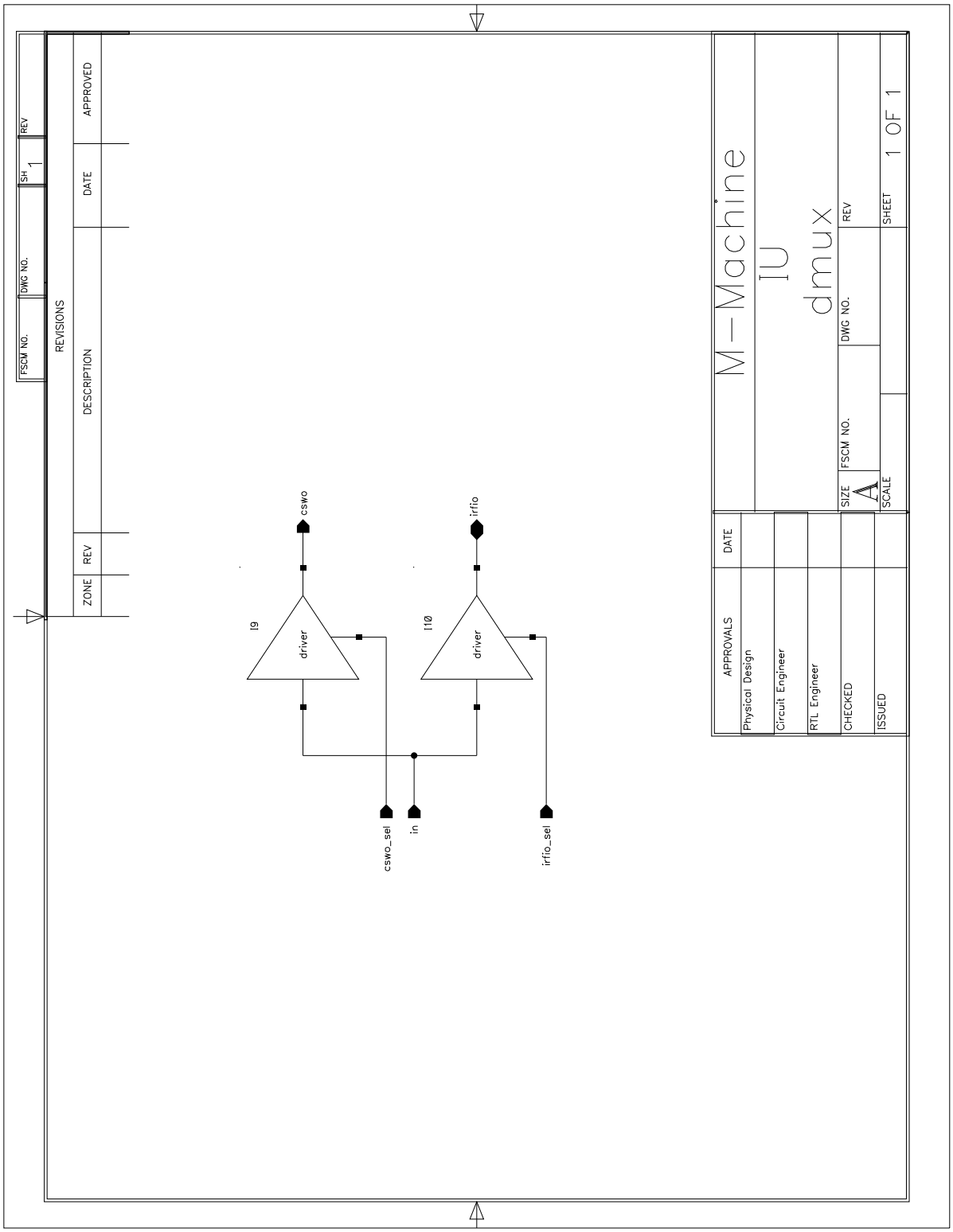

M-Machine
IU
sh_full_bit
1 OF 1

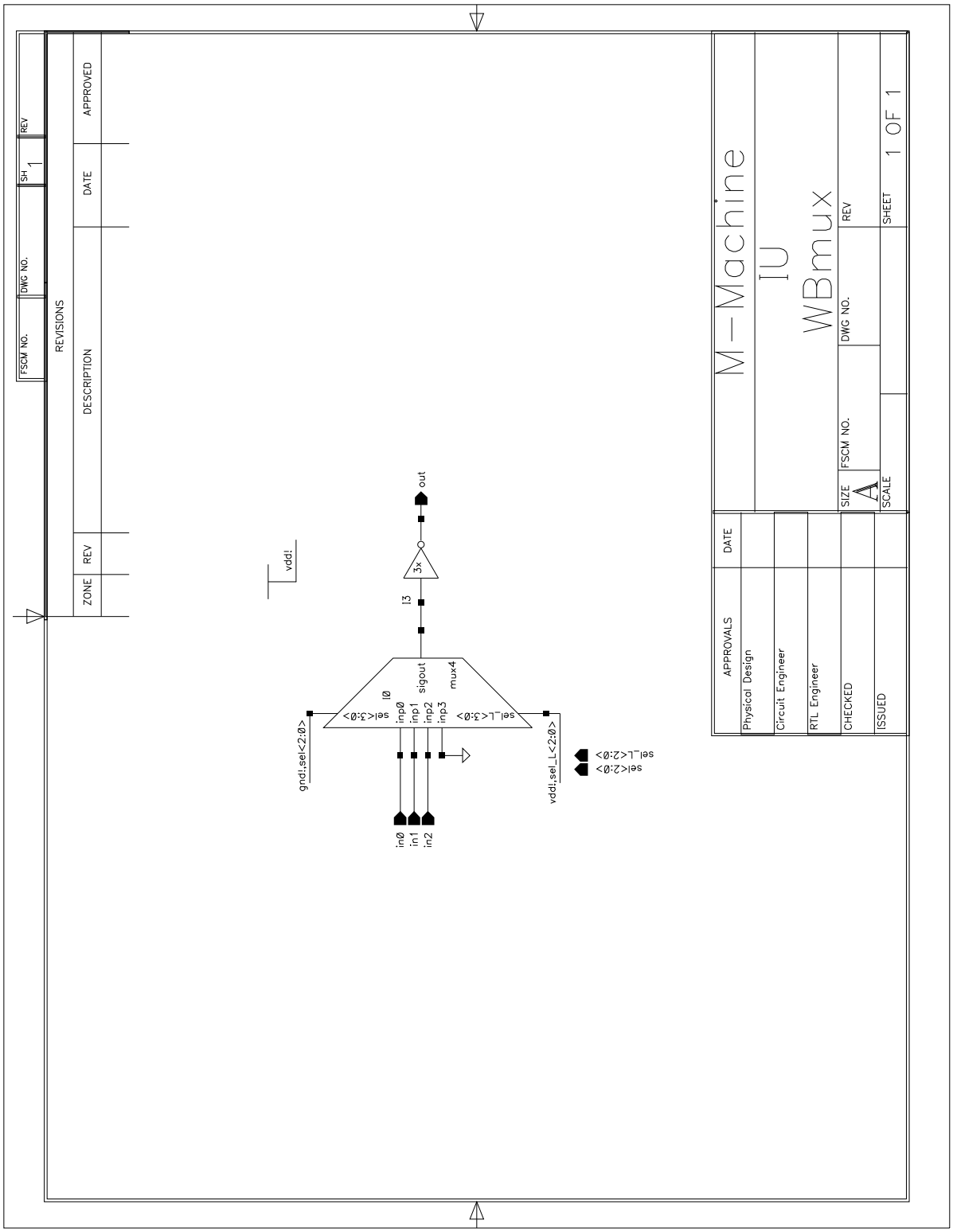

M-Machine
IU
sh_full_nib
1 OF 1

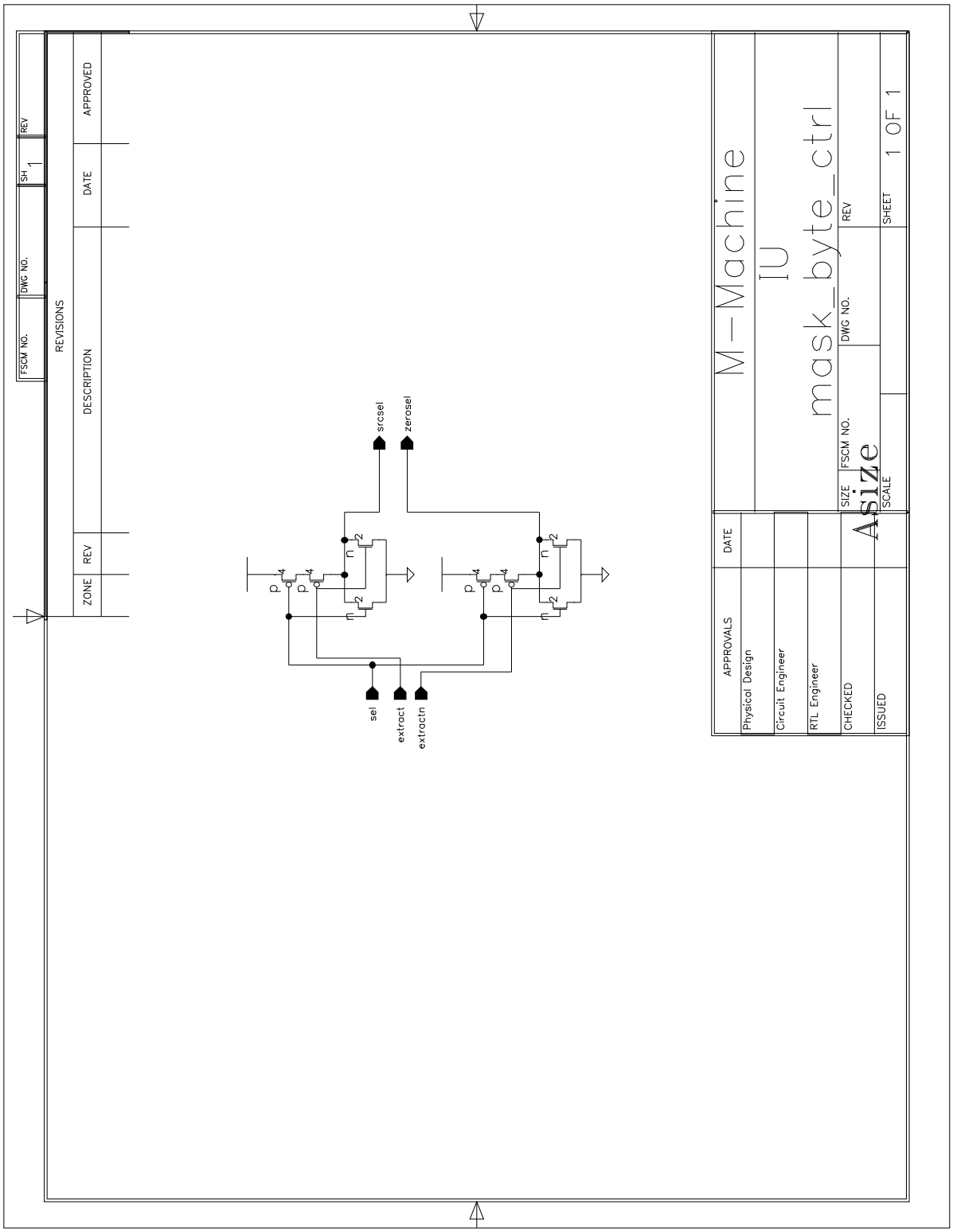

M-Machine
IU
sh_full_gw
REVISIONS
DESCRIPTION
APPROVALS
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
C
SHEET
1 OF 1

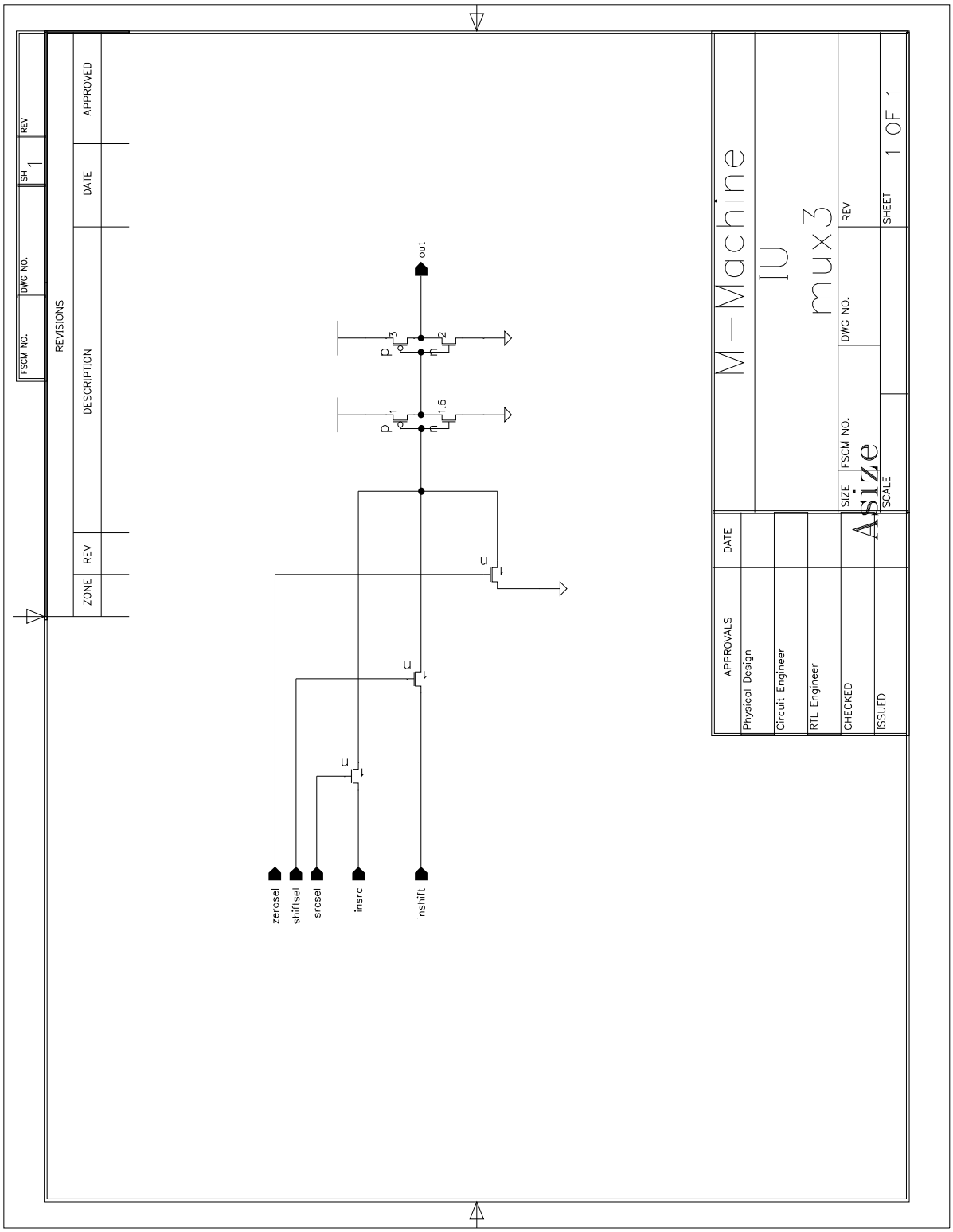

M-Machine
lU
sh_full
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
APPROVALS
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
DATE
SIZE
C
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1

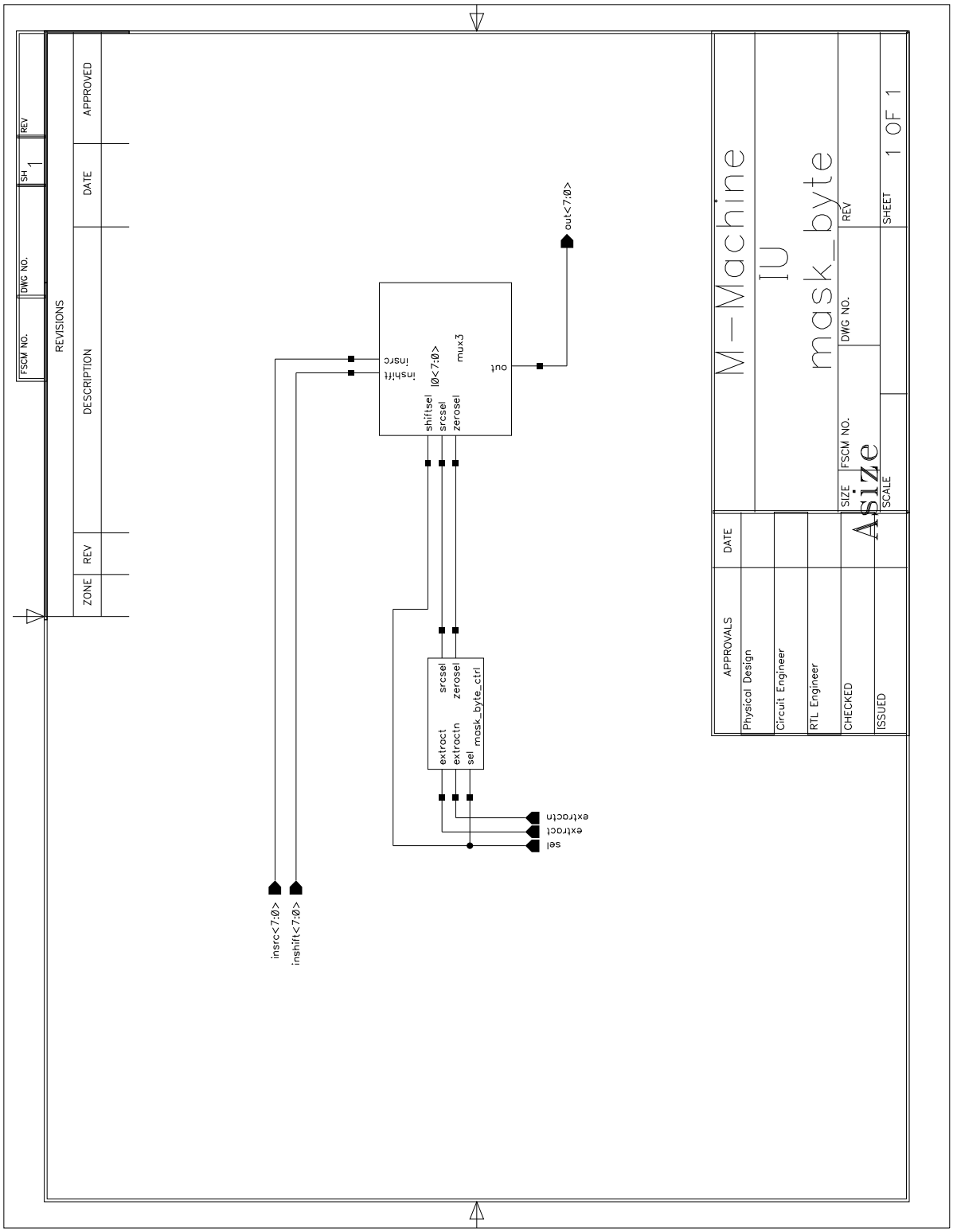

M-Machine
IU
driver
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE A
SCALE
SHEET 1 OF 1
sigout
sig
drive

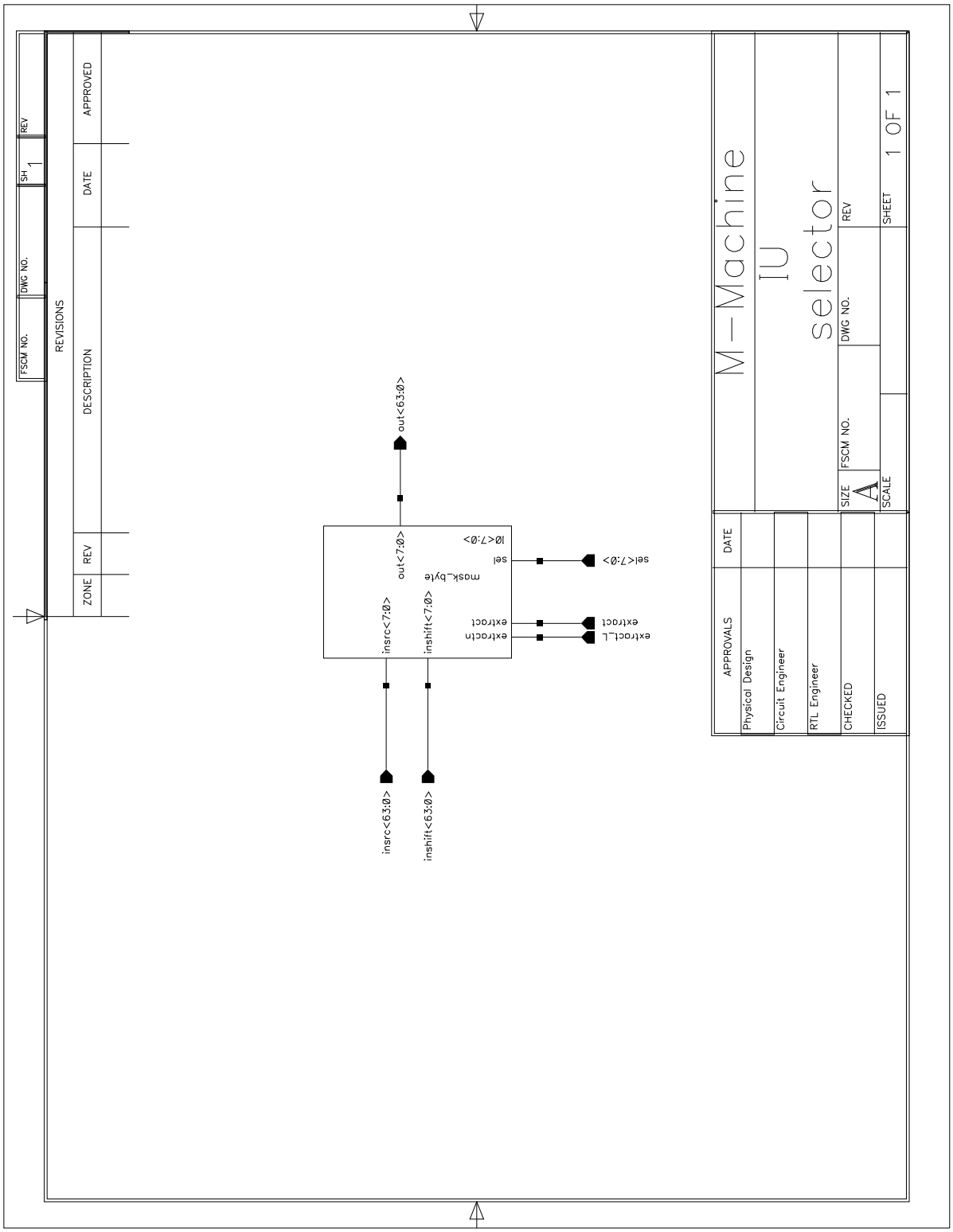

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
dmux
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
cswo
irfio
I9
I10
driver
driver
cswo_sel
in
irfio_sel

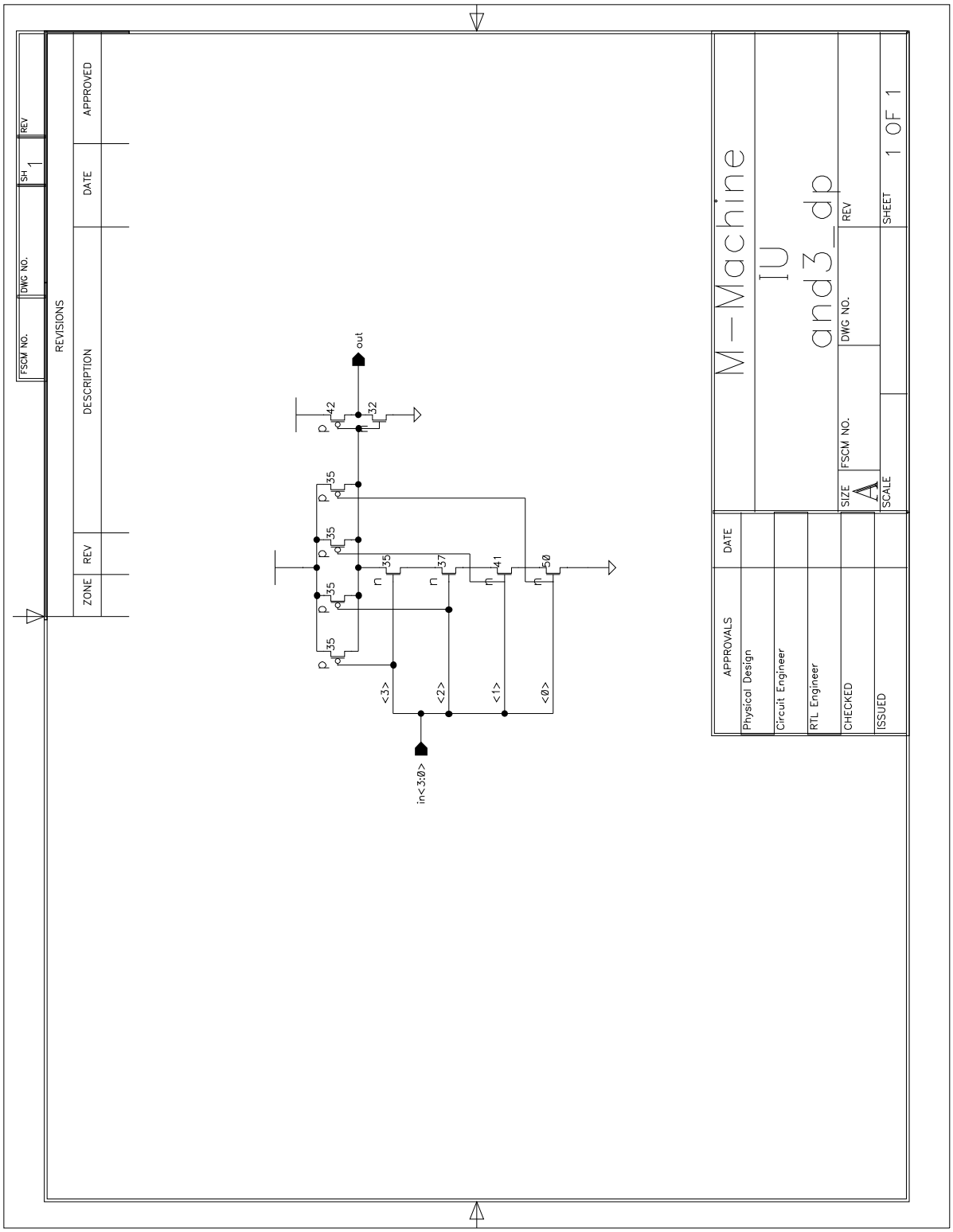

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
WBmux
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
vdd!
out
3x
I3
I0
sigout
mux4
inp0
inp1
inp2
inp3
sel<3:0>
sel_L<3:0>
gnd!:sel<2:0>
vdd!:sel_L<2:0>
in0
in1
in2
sel<2:0>
sel_L<2:0>

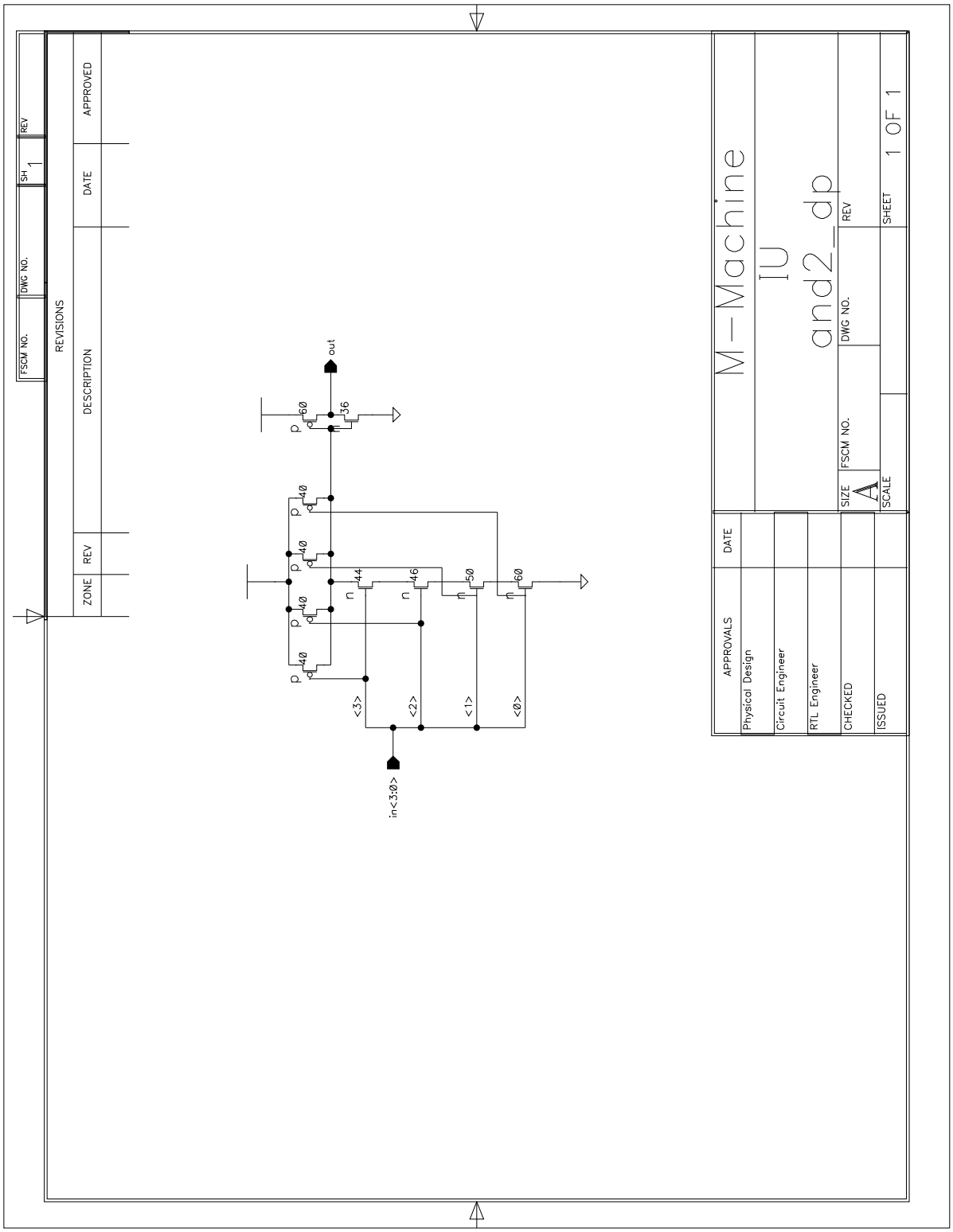

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
srcsel
zerosel
sel
extract
extractn
p 4
n 2
M-Machine
IU
mask_byte_ctrl
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A size
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1

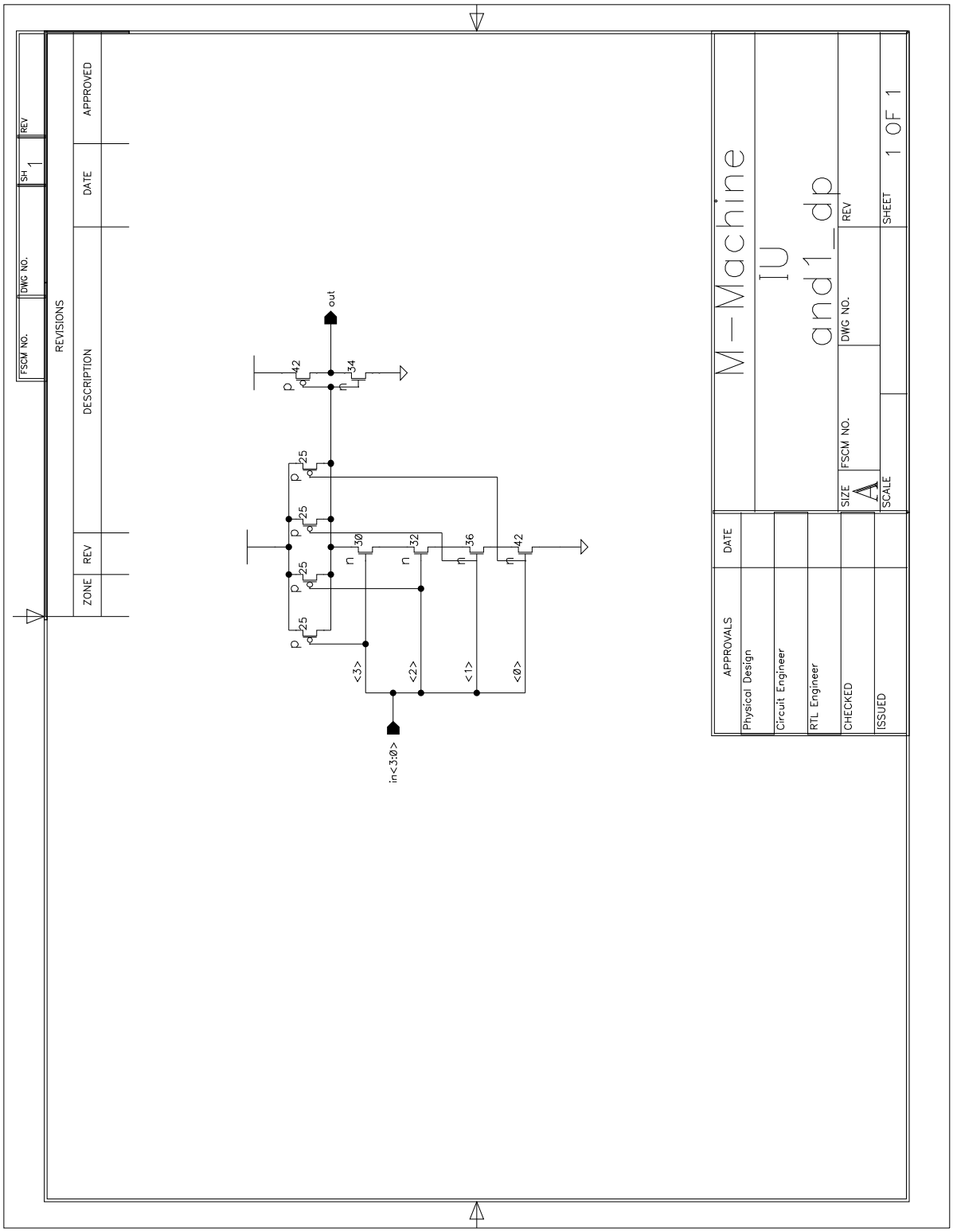

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
mux3
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
FSCM NO.
DWG NO.
REV
A size
SCALE
SHEET 1 OF 1
out
zerosel
shiftsel
srcsel
insrc
inshift

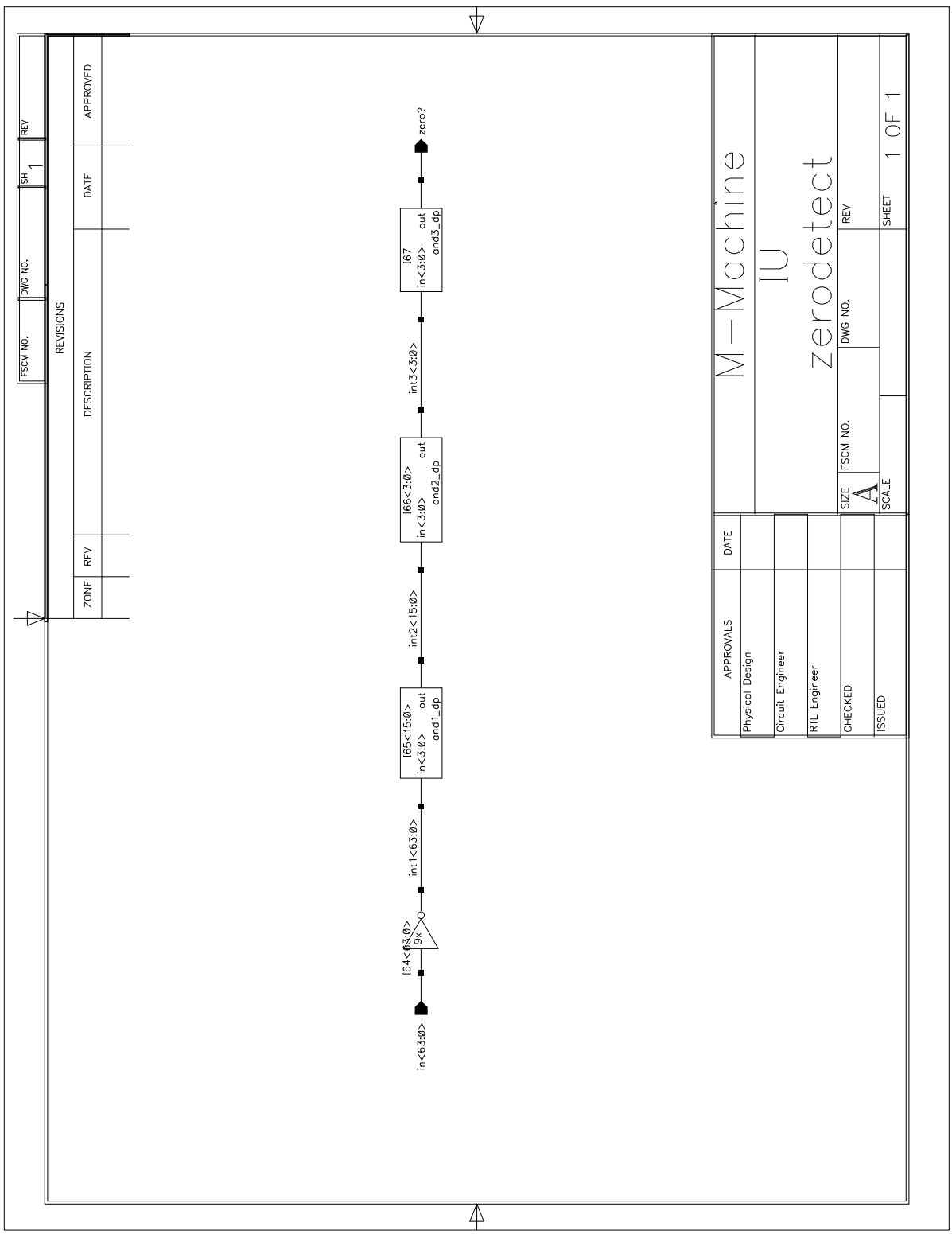

M-Machine
IU
mask_byte
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH 1
REV
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A size
SCALE
FSCM NO.
DWG NO.
REV
SHEET 1 OF 1
mux3
inshift
insrc
I0<7:0>
shiftsel
srcsel
zerosel
out
out<7:0>
mask_byte_ctrl
extract
extractn
sel
srcsel
zerosel
insrc<7:0>
inshift<7:0>
sel
extract
extractn

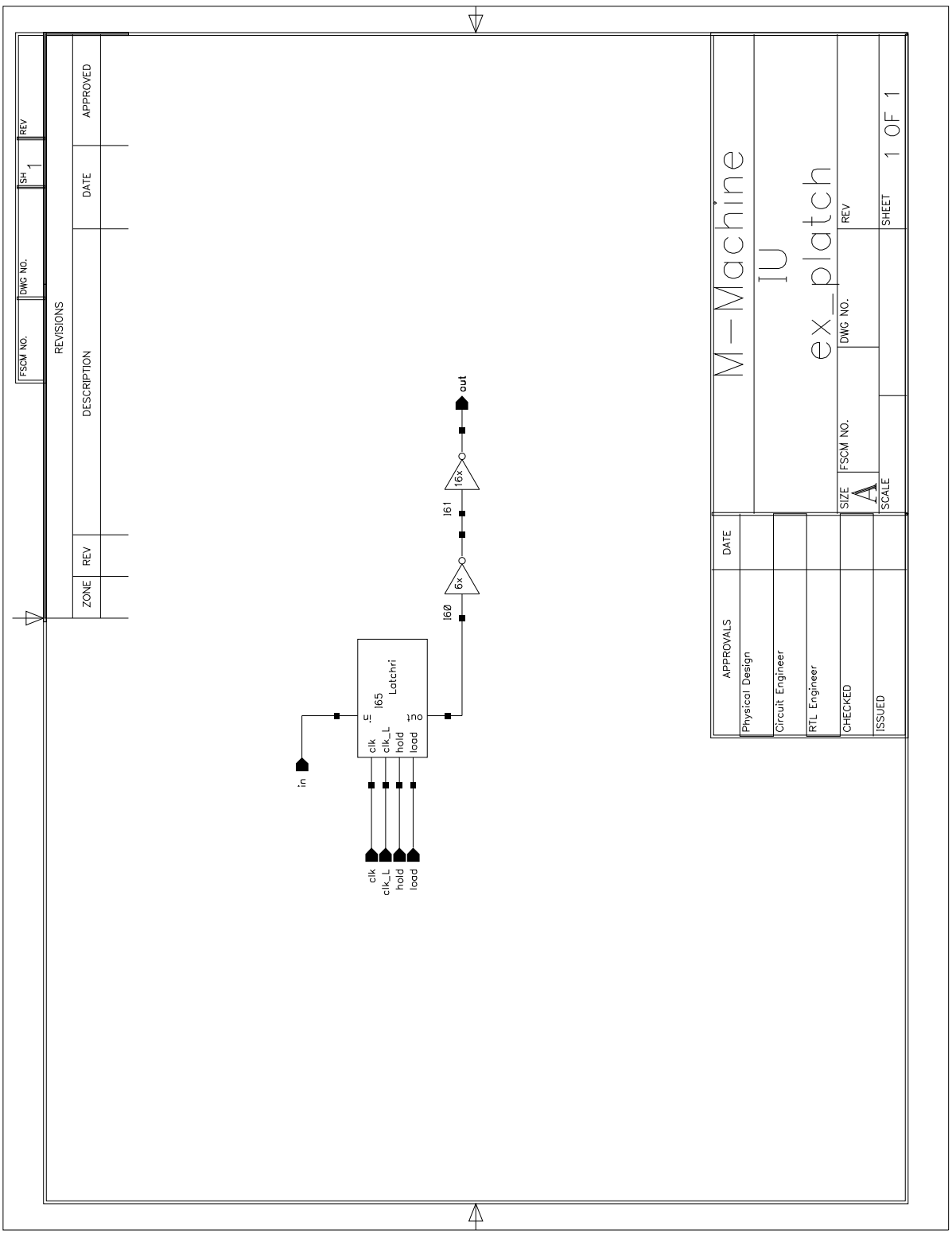

M-Machine
IU
selector
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE A
SCALE
SHEET 1 OF 1
mask_byte
insrc<7:0>
inshift<7:0>
sel
extract
extractn
out<7:0>
insrc<63:0>
inshift<63:0>
sel<7:0>
extract
extractL
out<63:0>

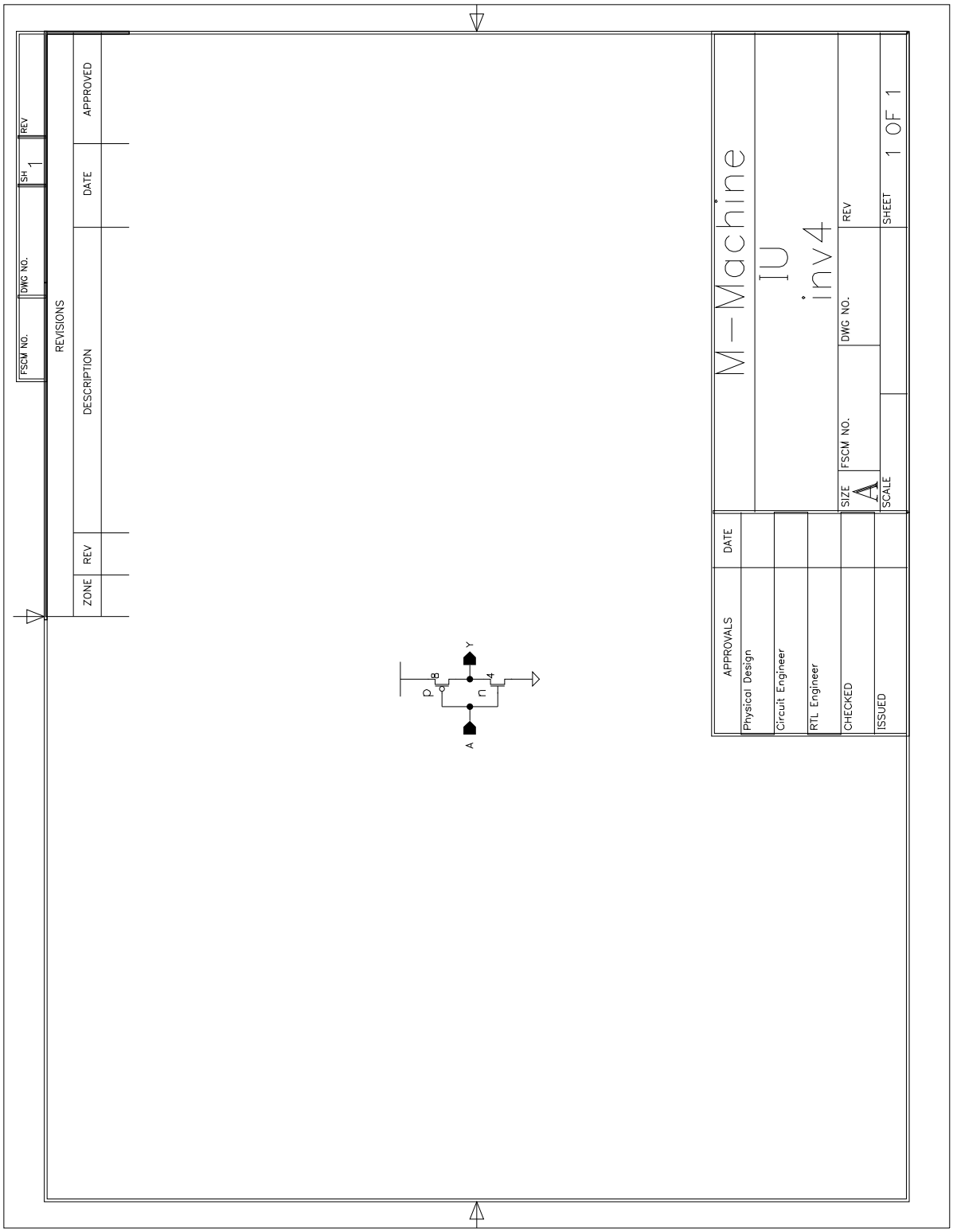

FSCM NO.
DWG NO.
REV
SH 1
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
out
in<3:0>
<3>
<2>
<1>
<0>
M-Machine
IU
and3_dp
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
SCALE
FSCM NO.
DWG NO.
REV
SHEET 1 OF 1

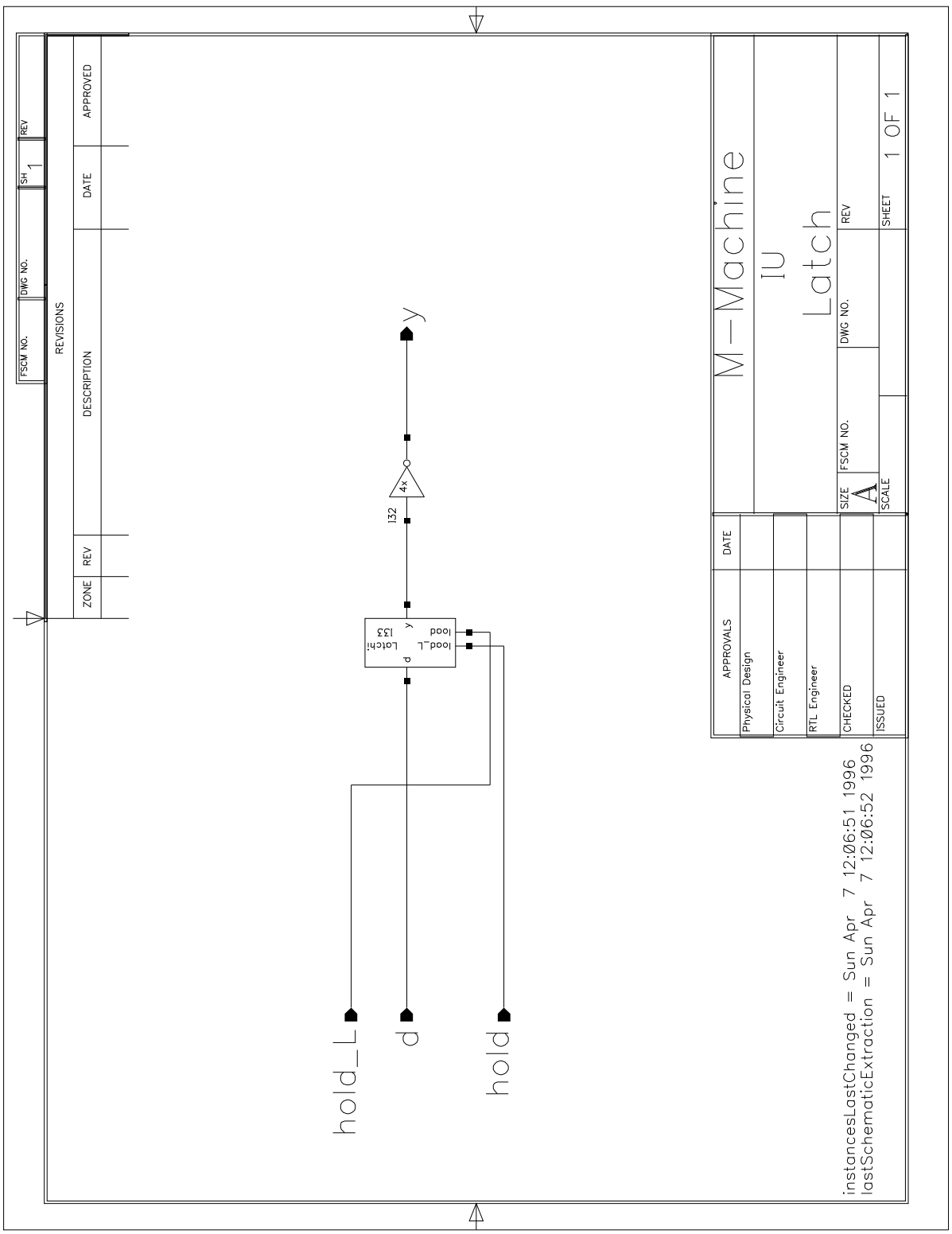

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
and2_dp
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
out
in<3:0>
<3>
<2>
<1>
<0>
p
n
60
36
40
44
46
50
60


REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
and1_dp
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A1
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
out
in<3:0>
<3>
<2>
<1>
<0>
p
n
25
30
32
36
42
34


FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
M-Machine
IU
zerodetect
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
in<63:0>
I64<63:0>
9x
int1<63:0>
I65<15:0>
in<3:0>
out
and1_dp
int2<15:0>
I66<3:0>
in<3:0>
out
and2_dp
int3<3:0>
I67
in<3:0>
out
and3_dp
zero?


REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
ex_platch
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
out
16x
I61
6x
I60
I65
Latchri
clk
clk_L
hold
load
in
out
clk
clk_L
hold
load


M-Machine
IU
inv4
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A1
SCALE
SHEET 1 OF 1
Y
A
p
n
8
4

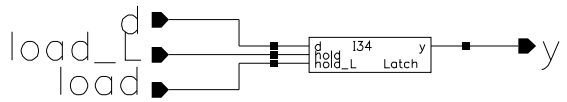


M-Machine
IU
Latch
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
instancesLastChanged = Sun Apr 7 12:06:51 1996
lastSchematicExtraction = Sun Apr 7 12:06:52 1996
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH 1
REV
hold_L
d
hold
Latchi
133
load_L
load
d
y
132
4x
y

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



instancesLastChanged = Sun Apr 7 12:06:52 1996
lastSchematicExtraction = Sun Apr 7 12:06:53 1996

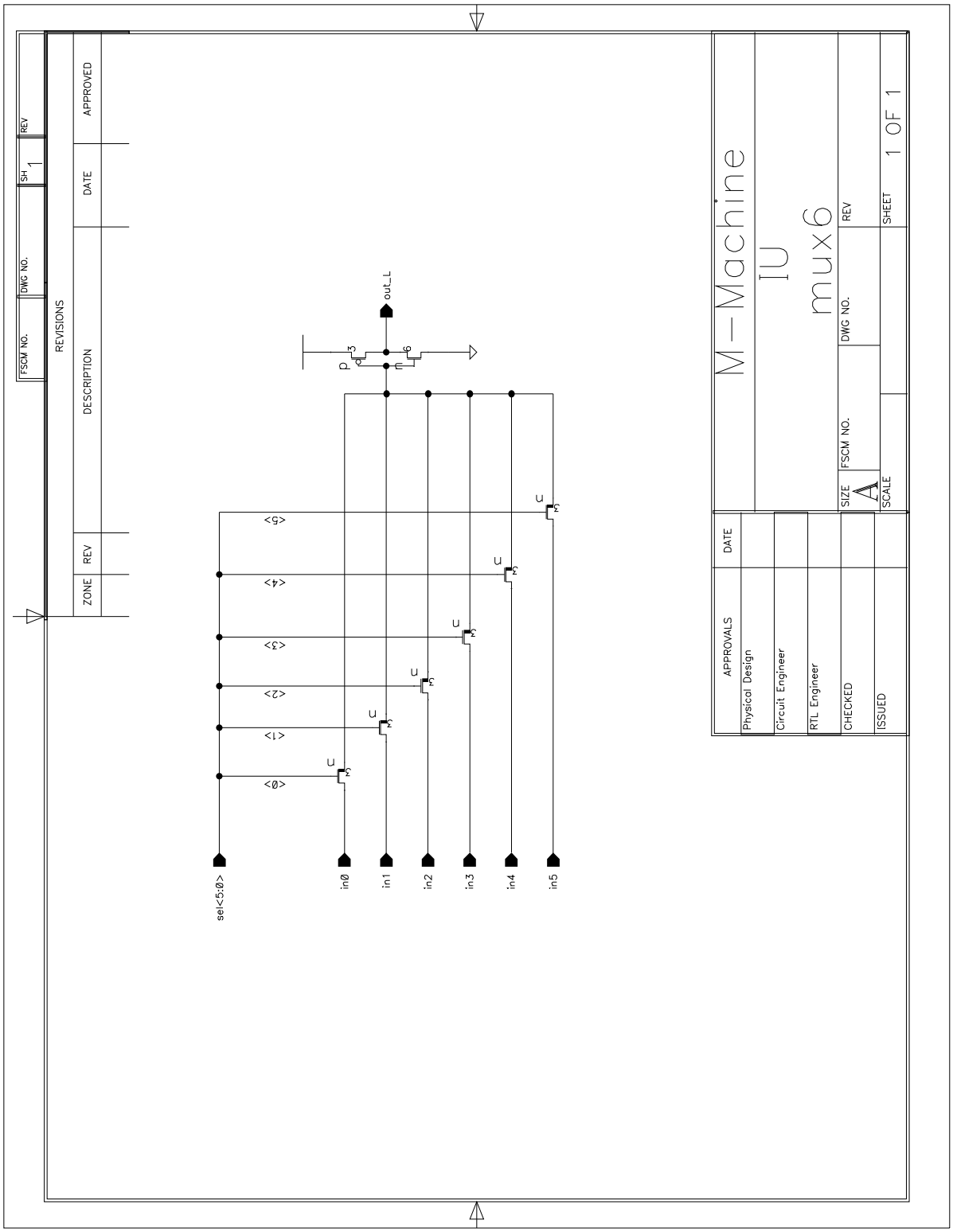
APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

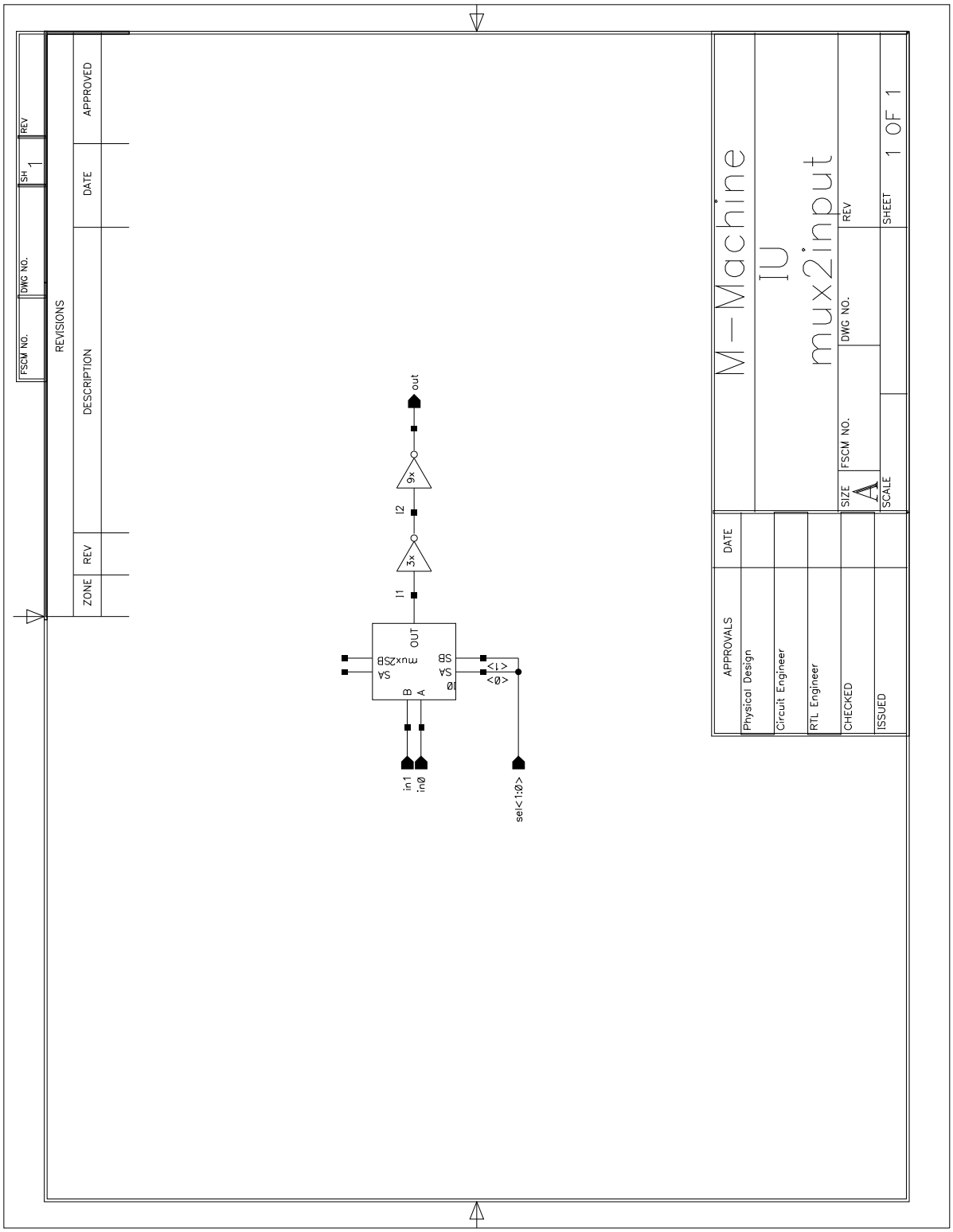
M—Machine

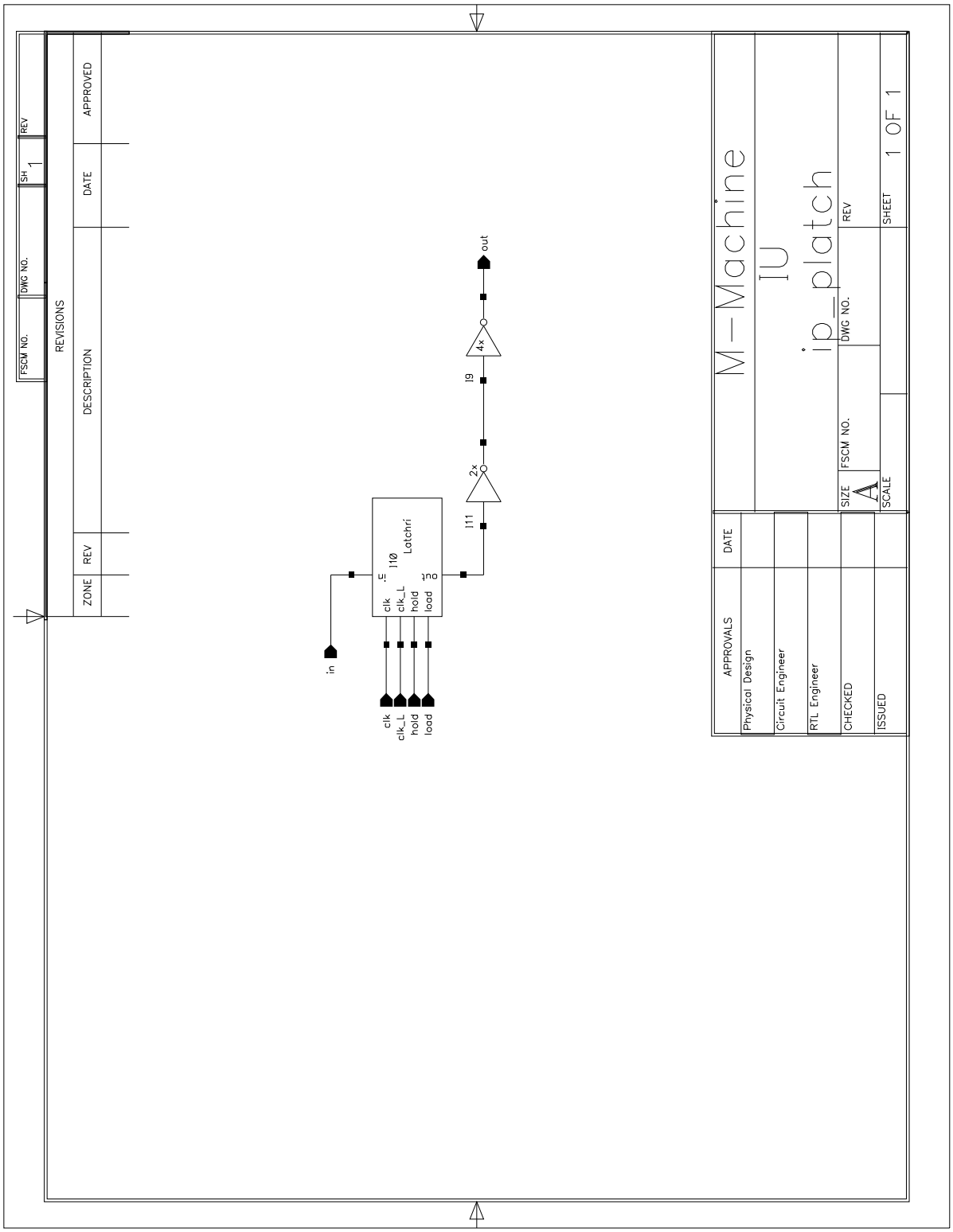
IU

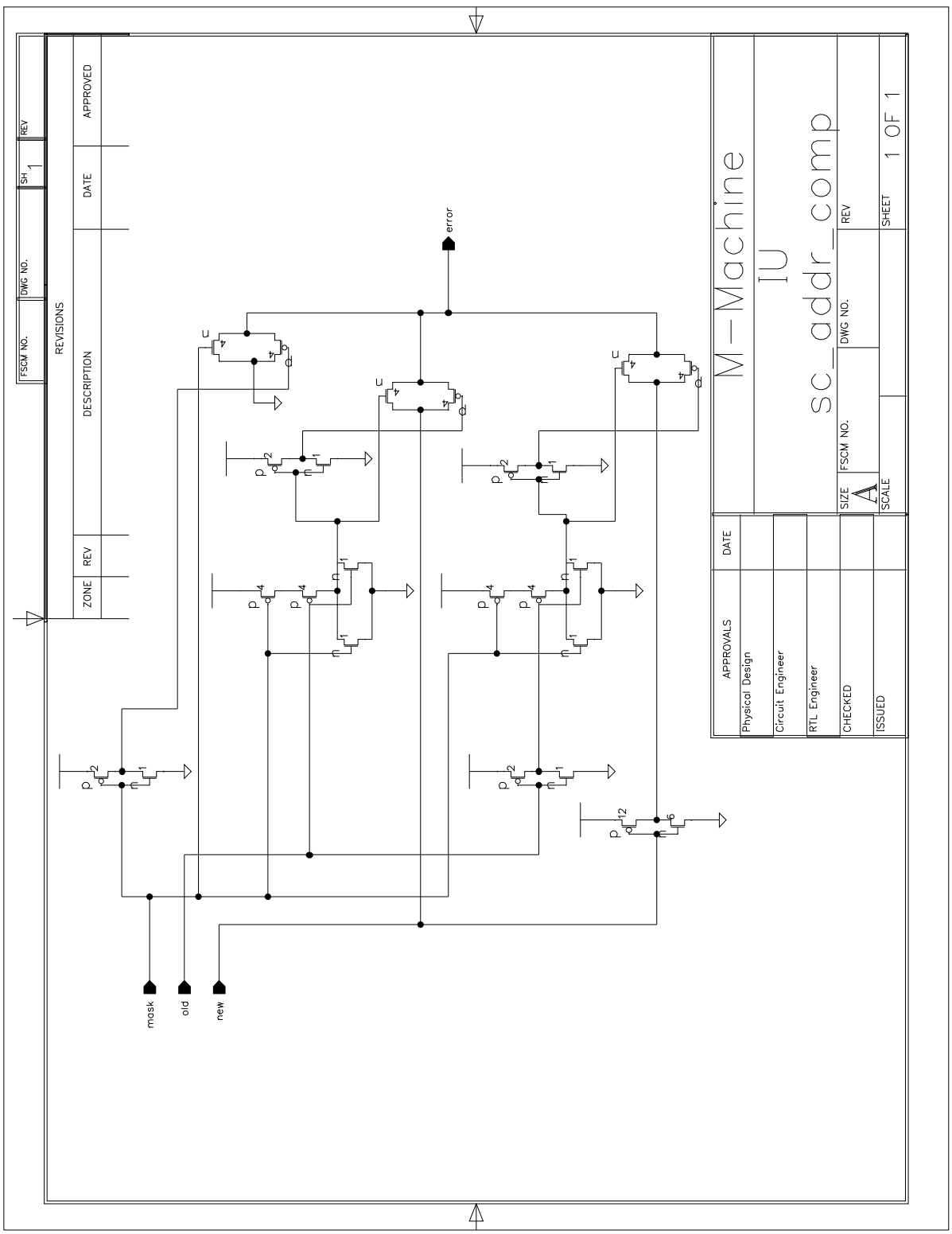
Latch_ni

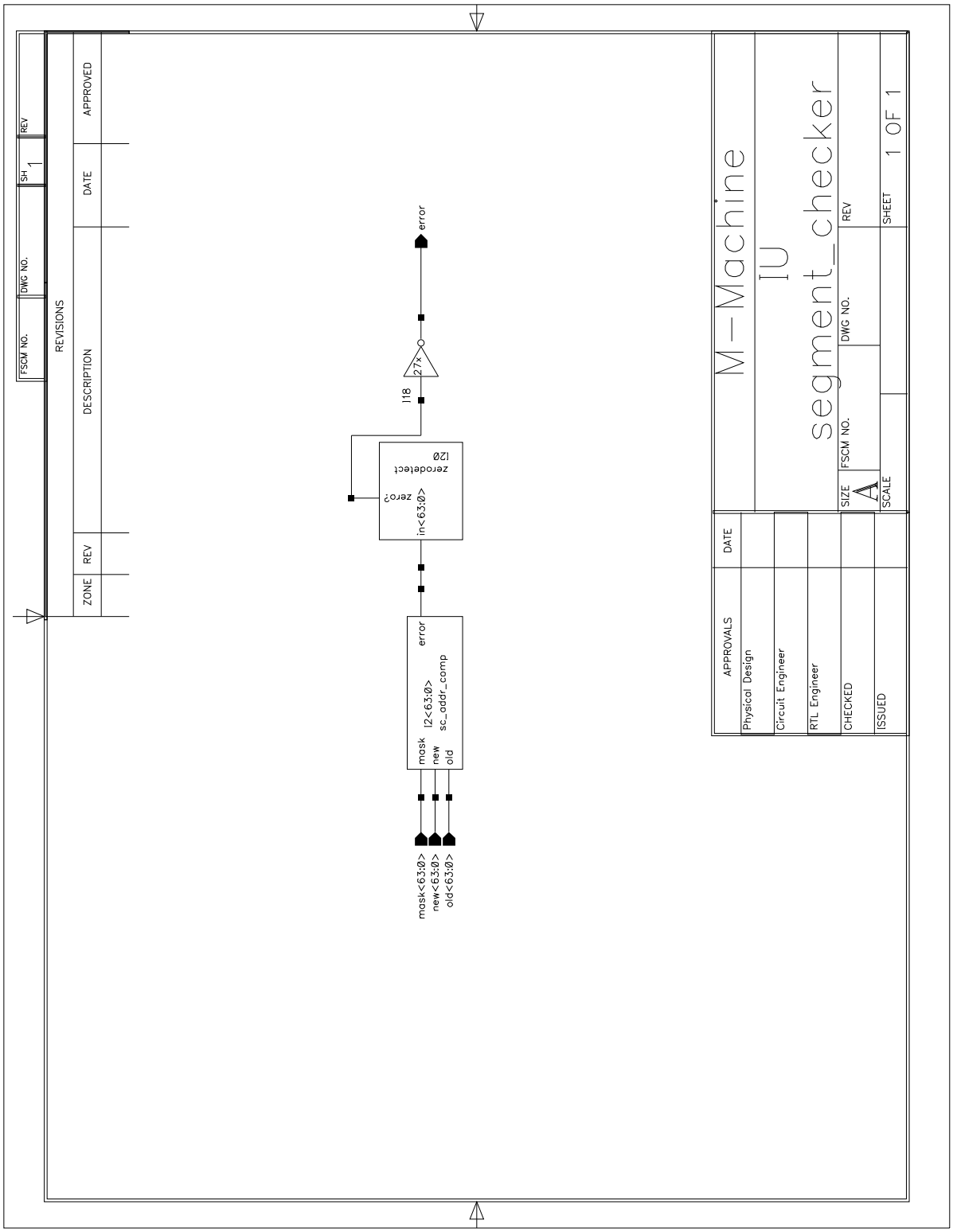
SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE			SHEET 1 OF 1

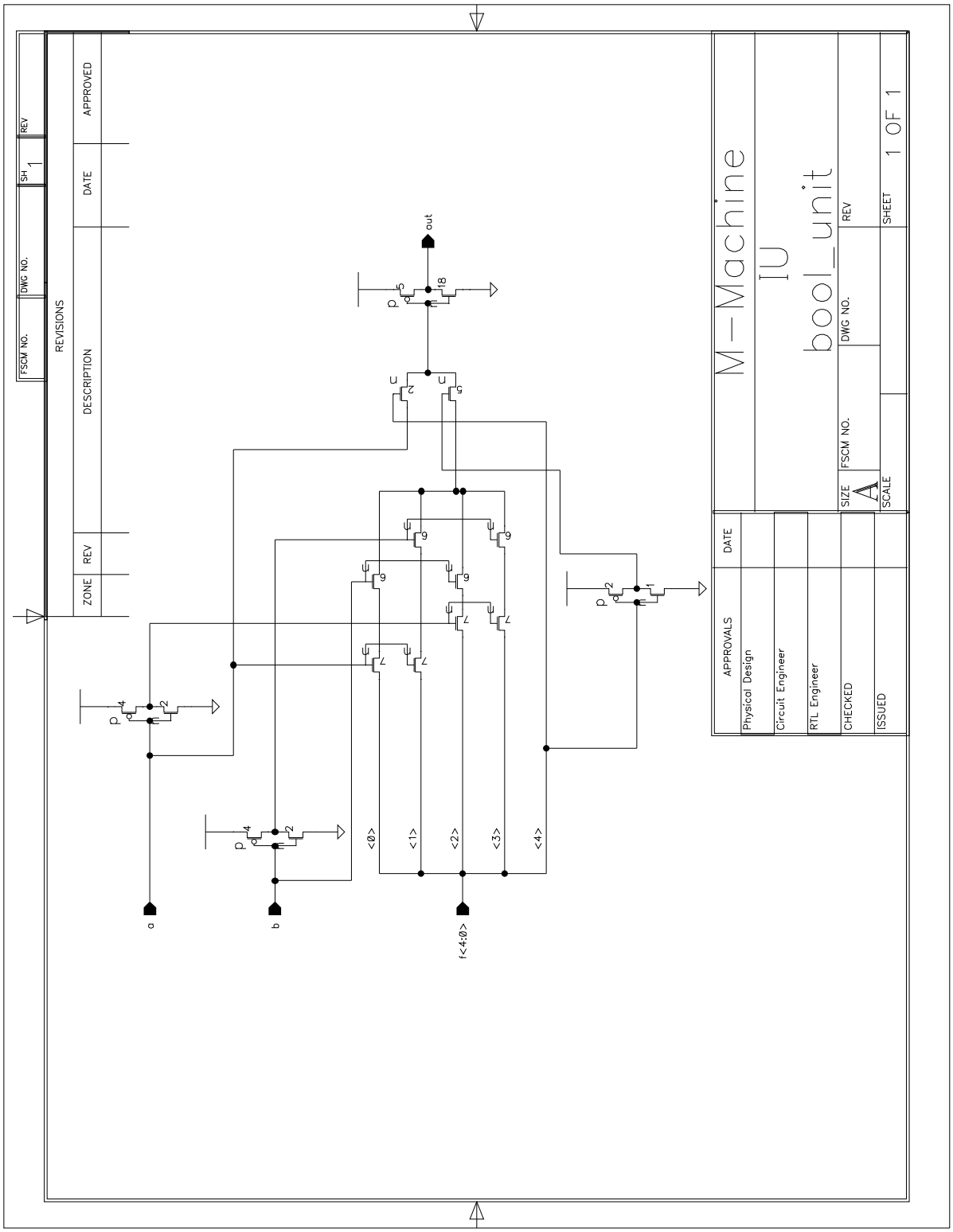

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
mux6
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
out_L
sel<5:0>
in0
in1
in2
in3
in4
in5
<0>
<1>
<2>
<3>
<4>
<5>

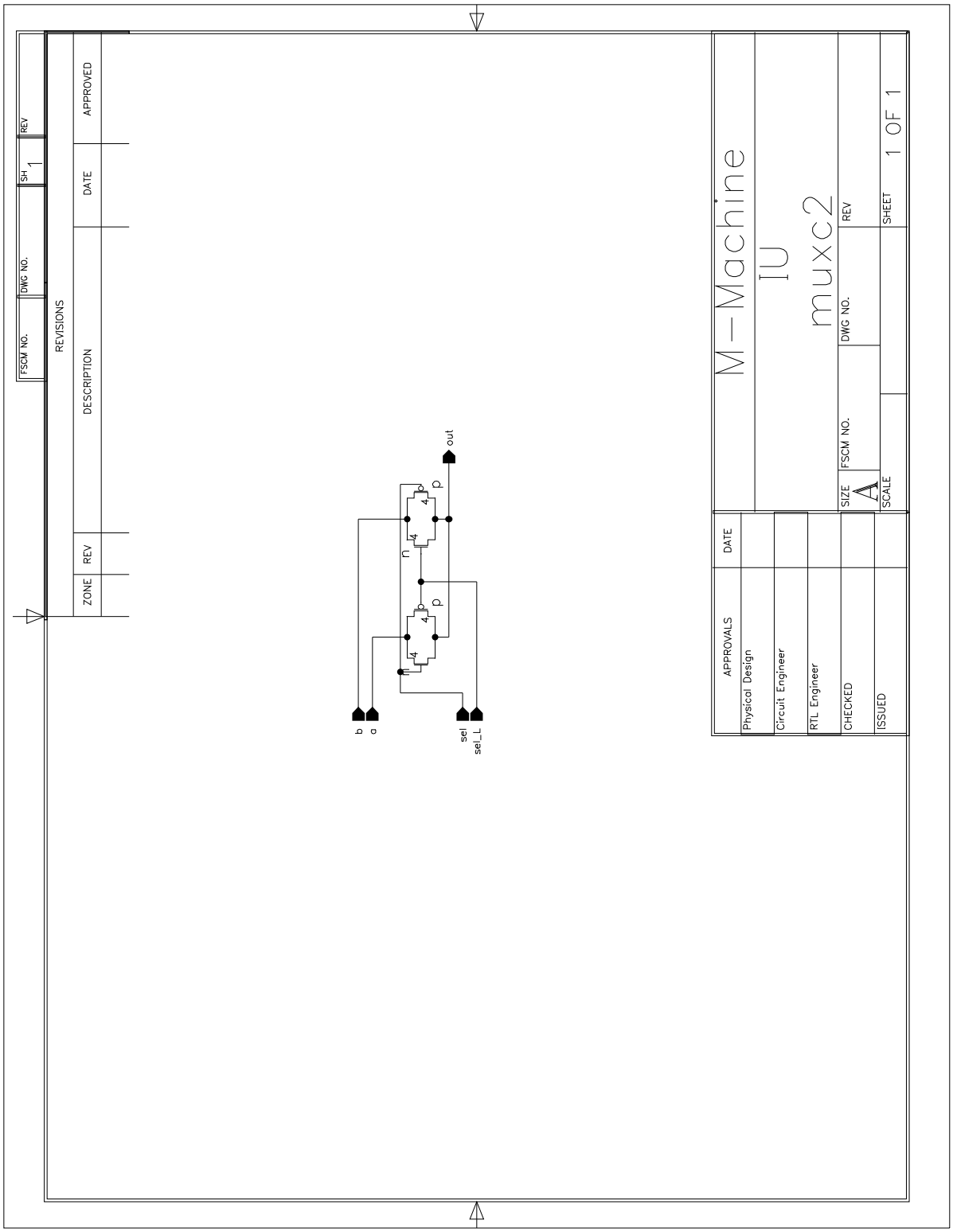

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
out
9x
I2
3x
I1
mux2input
OUT
B
A
S
in1
in0
sel<1:0>
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
M-Machine
IU
mux2input
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1

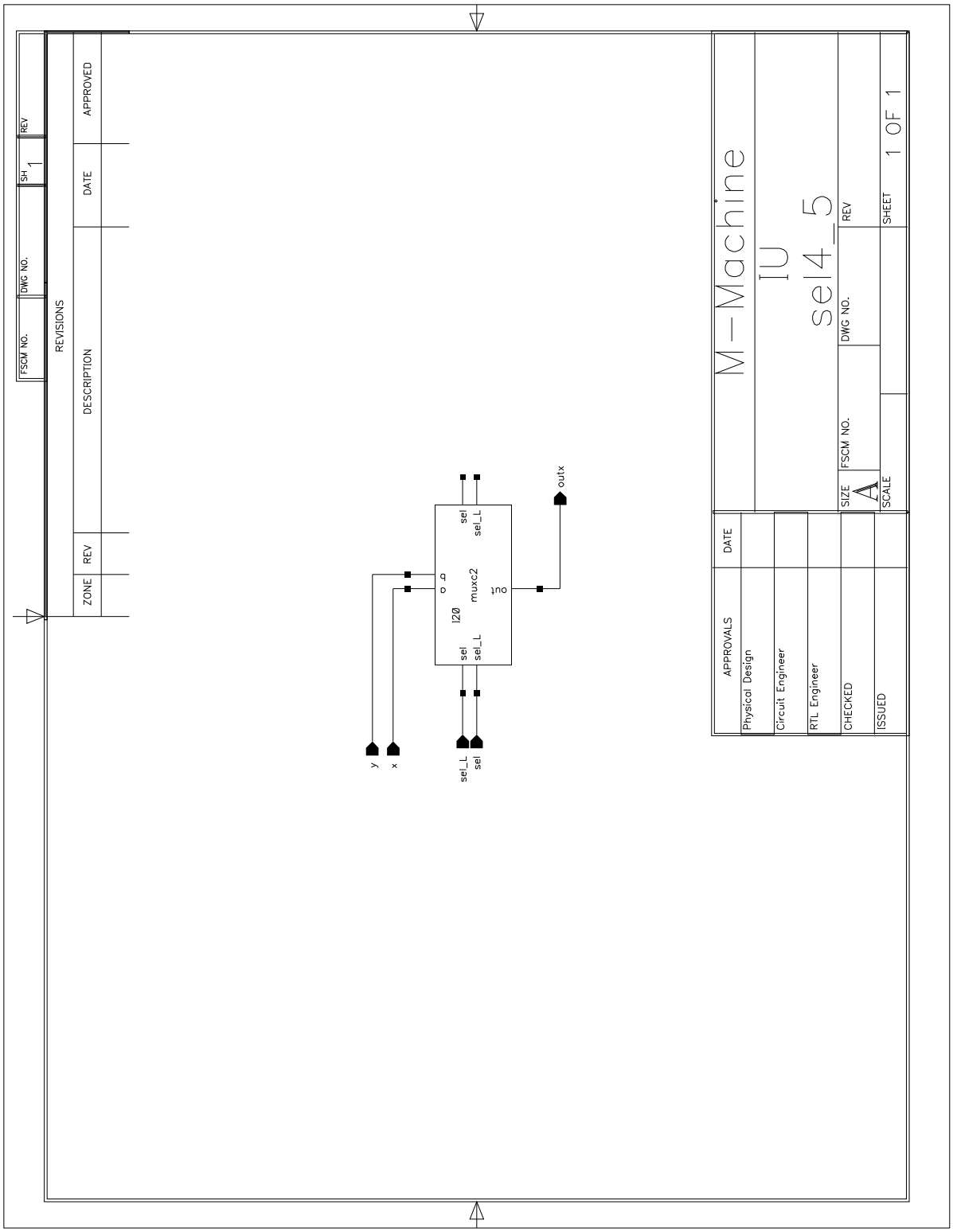

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
in
clk
clk_L
hold
load
I10
Latchri
d
q
I11
2x
I9
4x
out
M-Machine
IU
ip_platch
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1

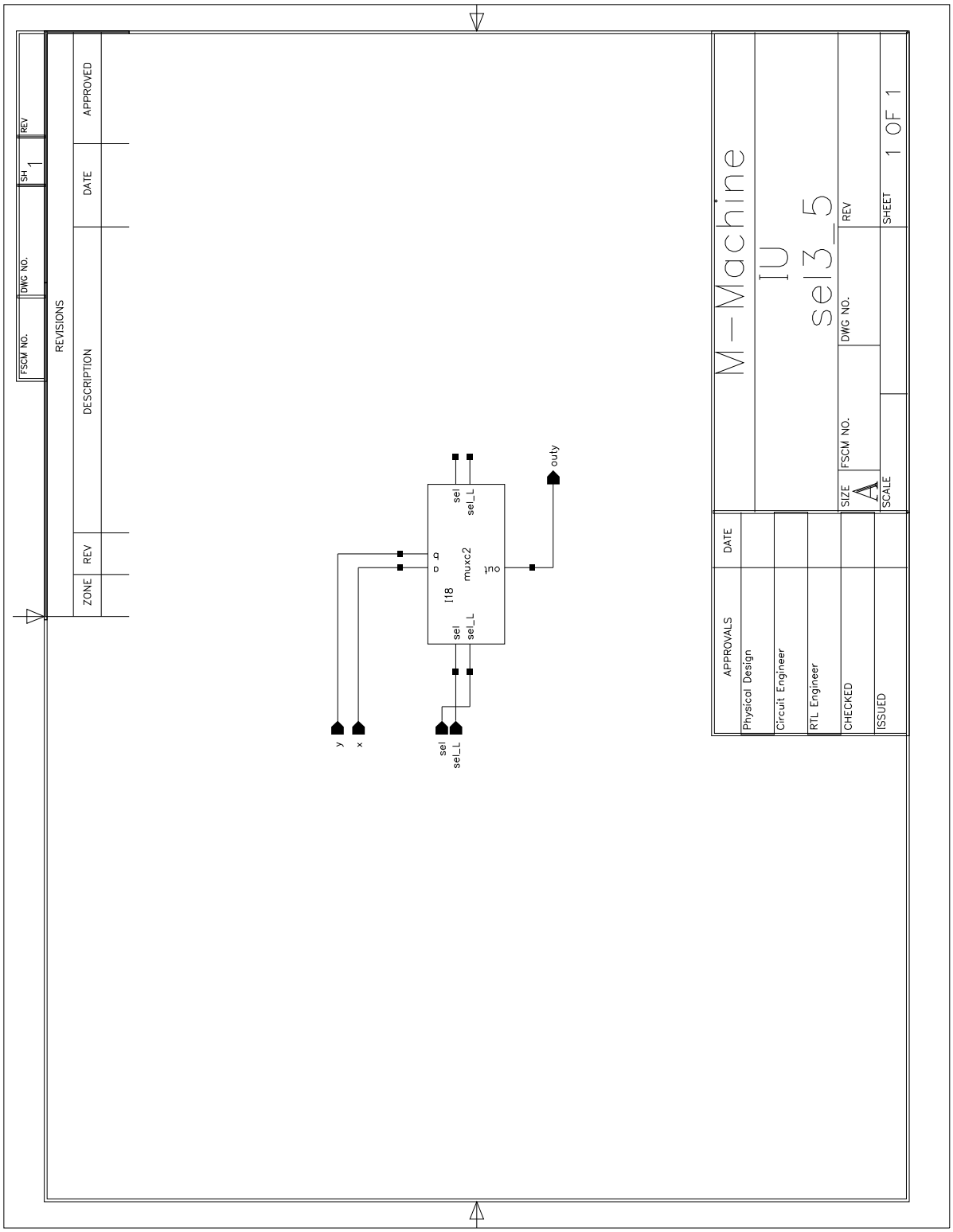

M-Machine
IU
sc_addr_comp
error
mask
old
new

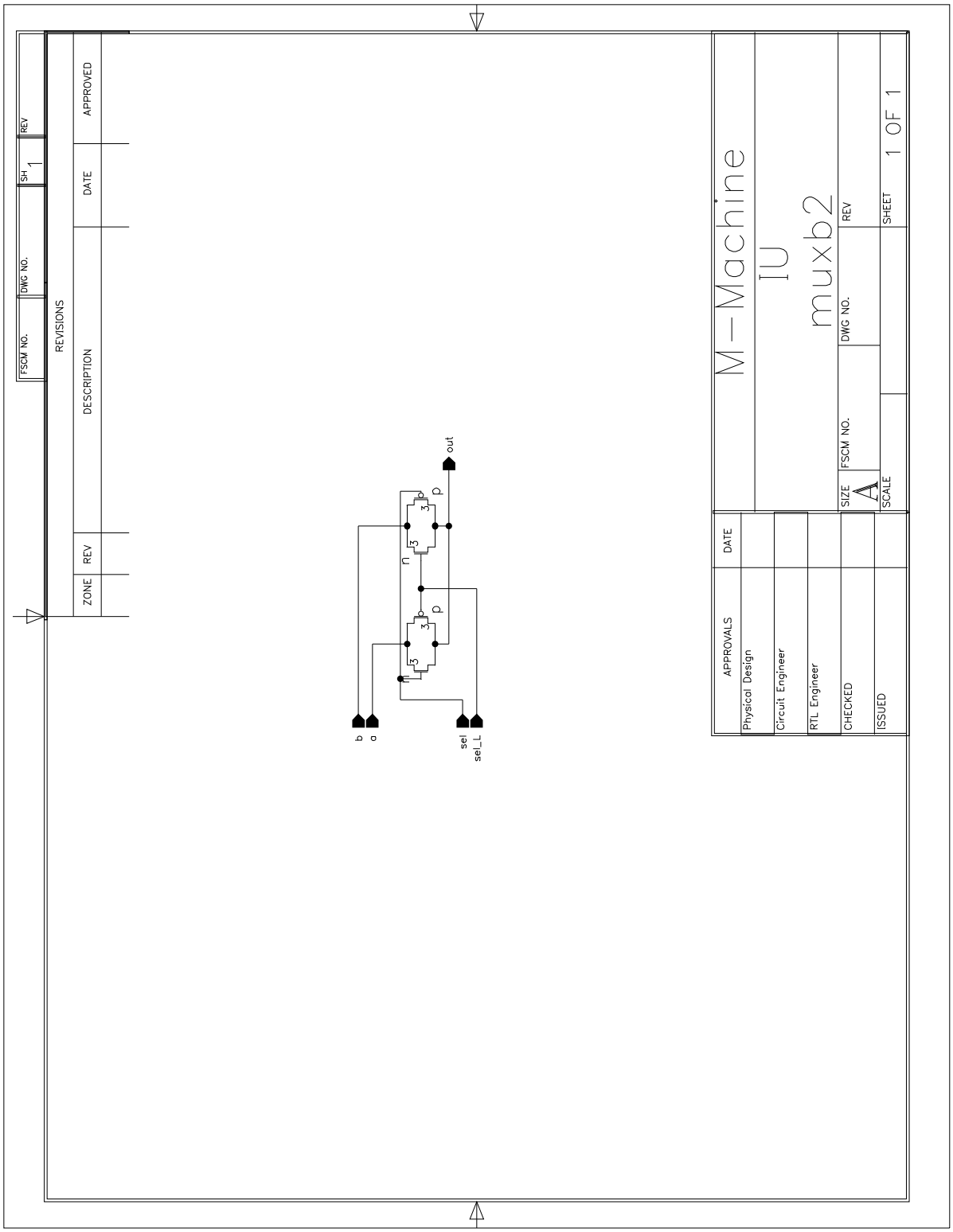

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
error
27x
118
zerodetect
120
zero?
in<63:0>
error
mask
new
old
12<63:0>
sc_addr_comp
mask<63:0>
new<63:0>
old<63:0>
M-Machine
IU
segment_checker
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A1
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1

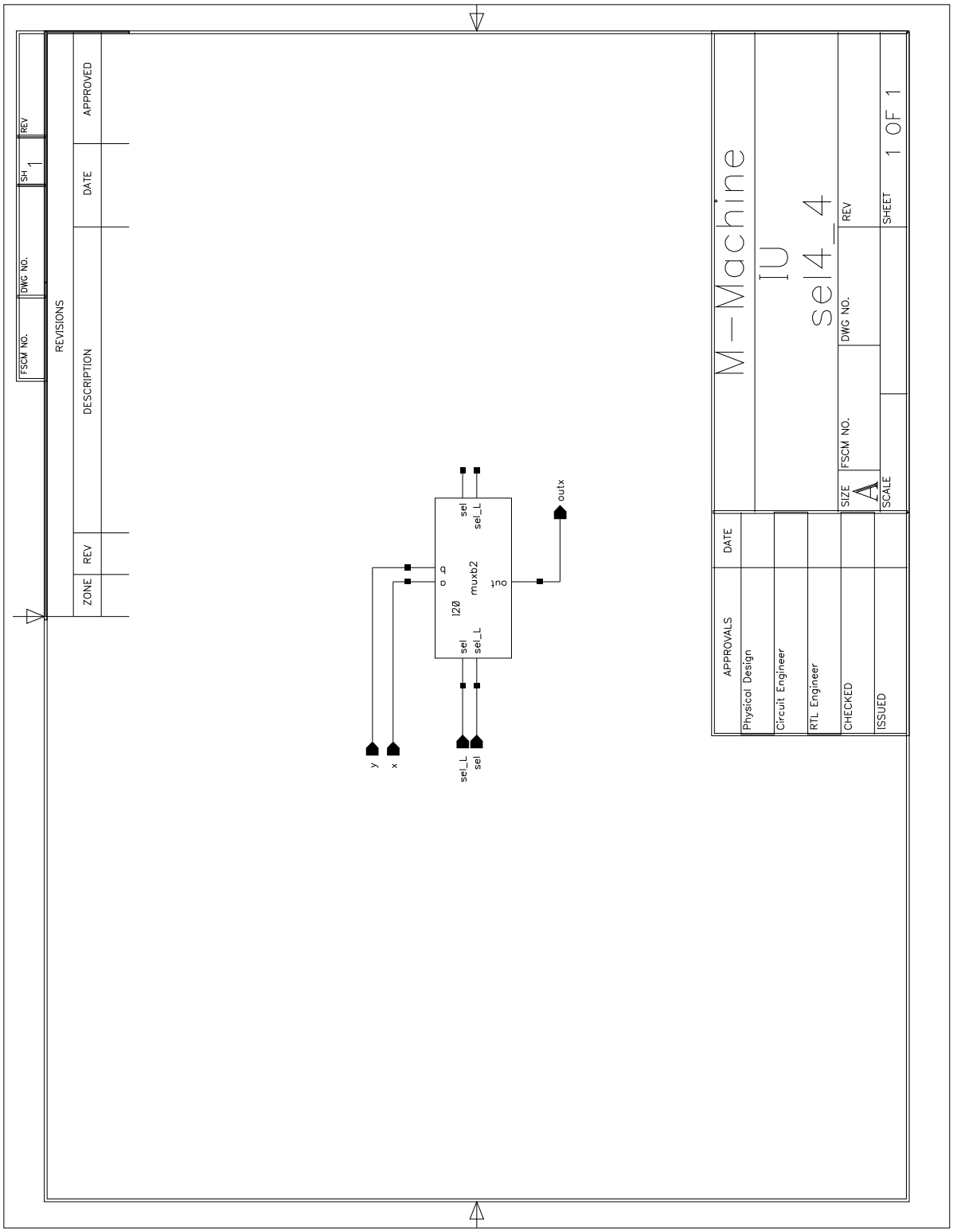

M-Machine
IU
bool_unit
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
SIZE
SCALE
SHEET 1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
out
a
b
r<4:0>
<0>
<1>
<2>
<3>
<4>

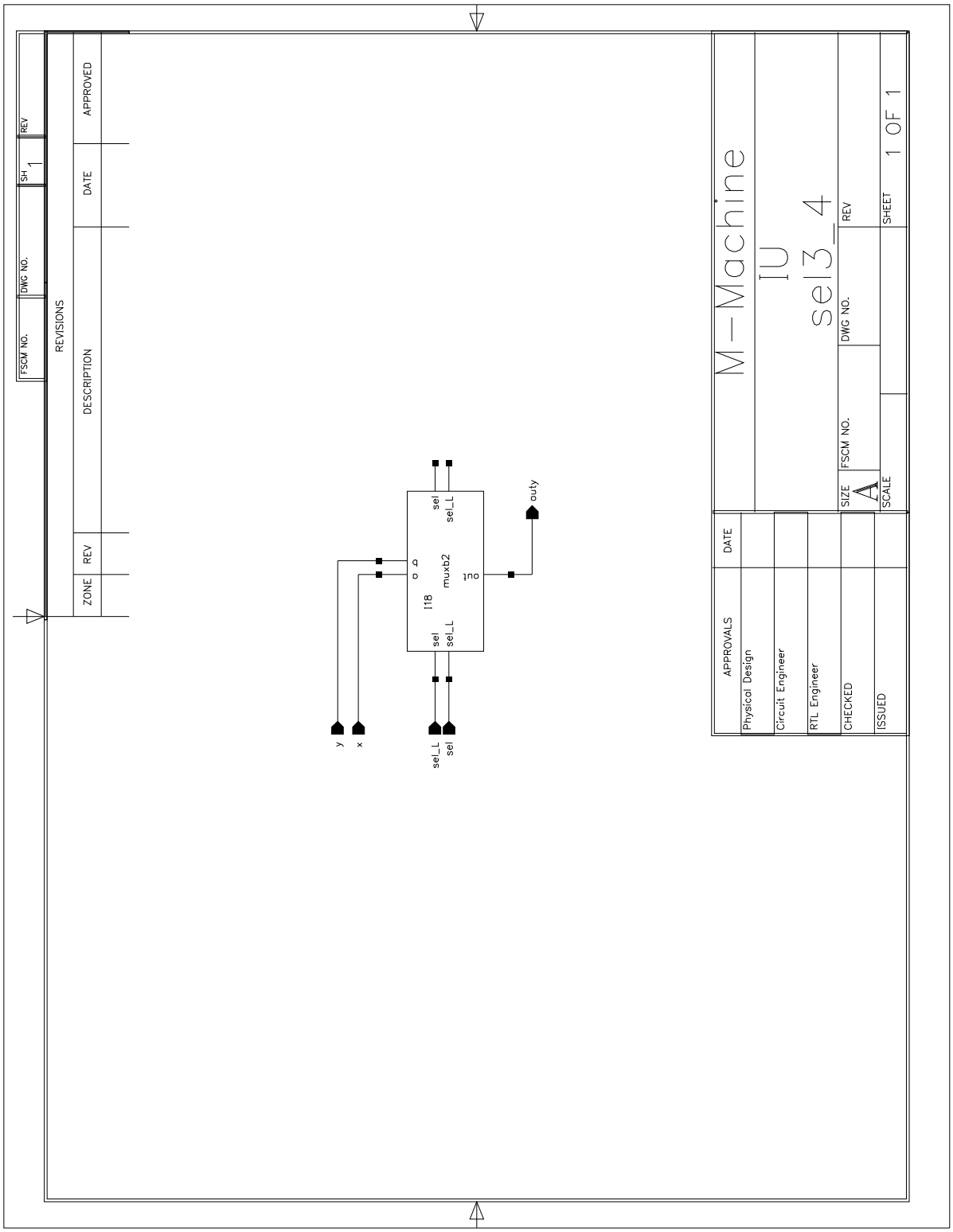

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
muxc2
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
out
p
p
n
p
4
4
4
4
b
a
sel
sel_L

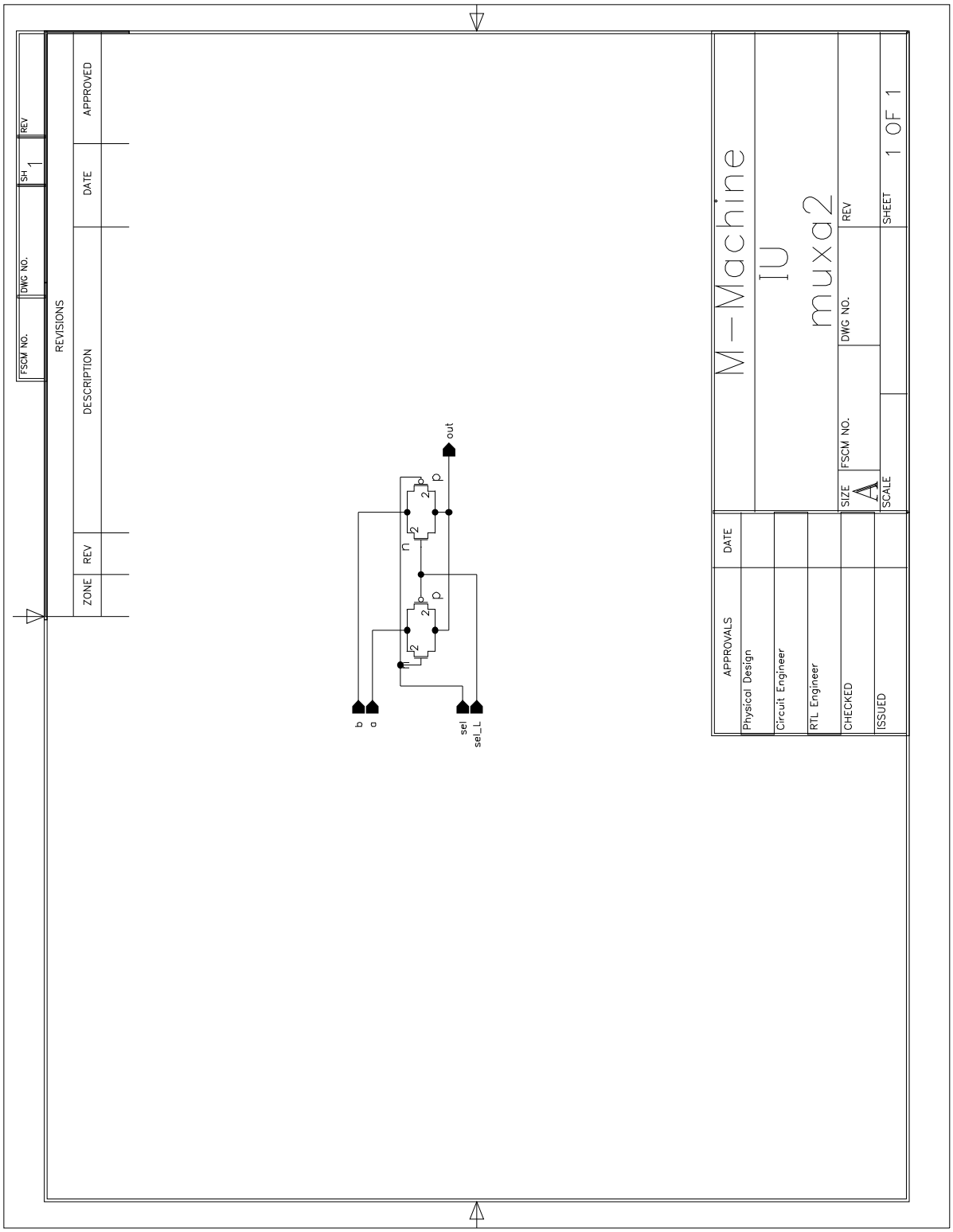

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
sel4_5
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
I20
mux2
sel
sel_L
out
outx
x
y

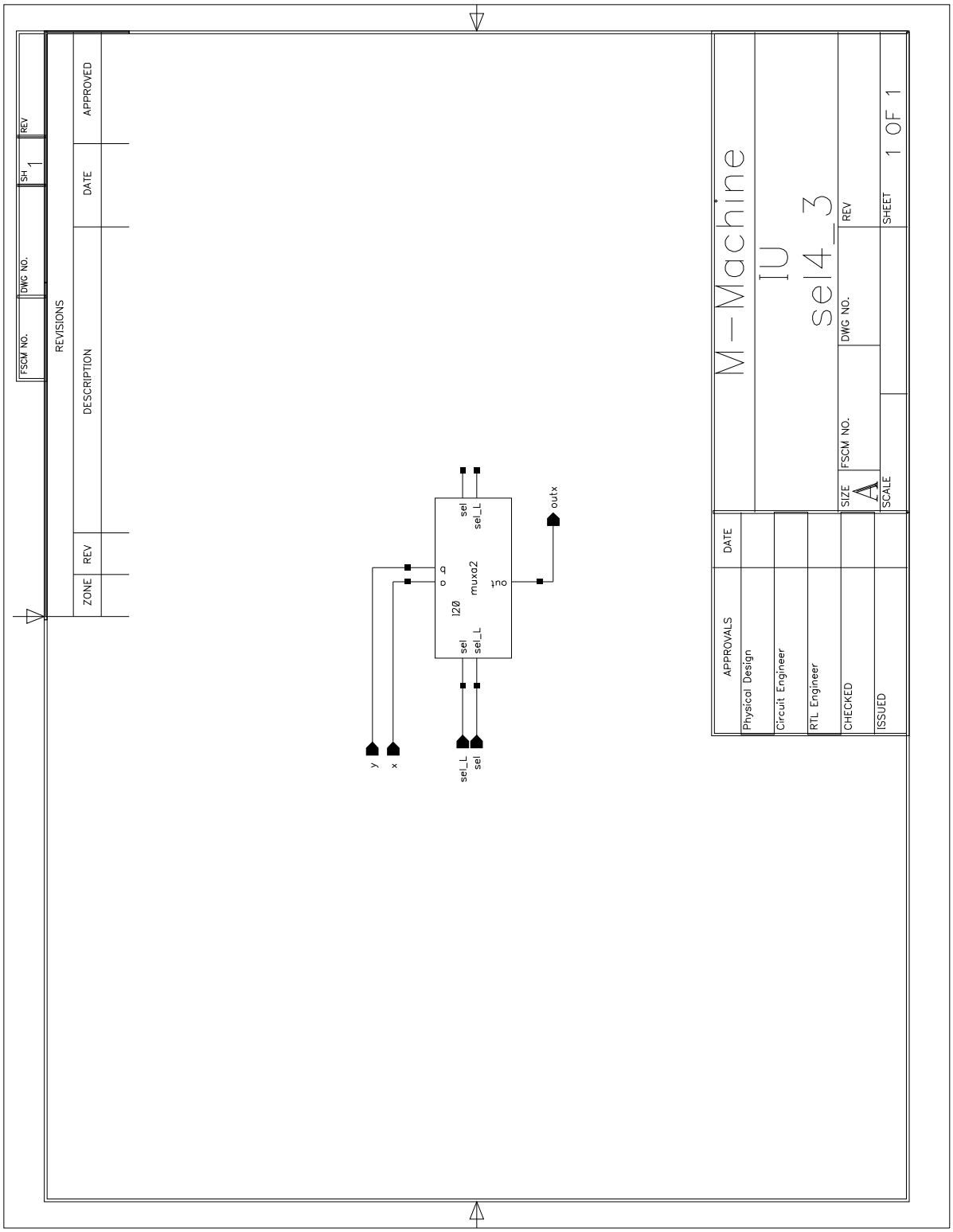

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
sel3_5
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
I18
muxc2
sel
sel_L
out
outy
y
x


REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
muxb2
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
out
b
a
sel
sel_L

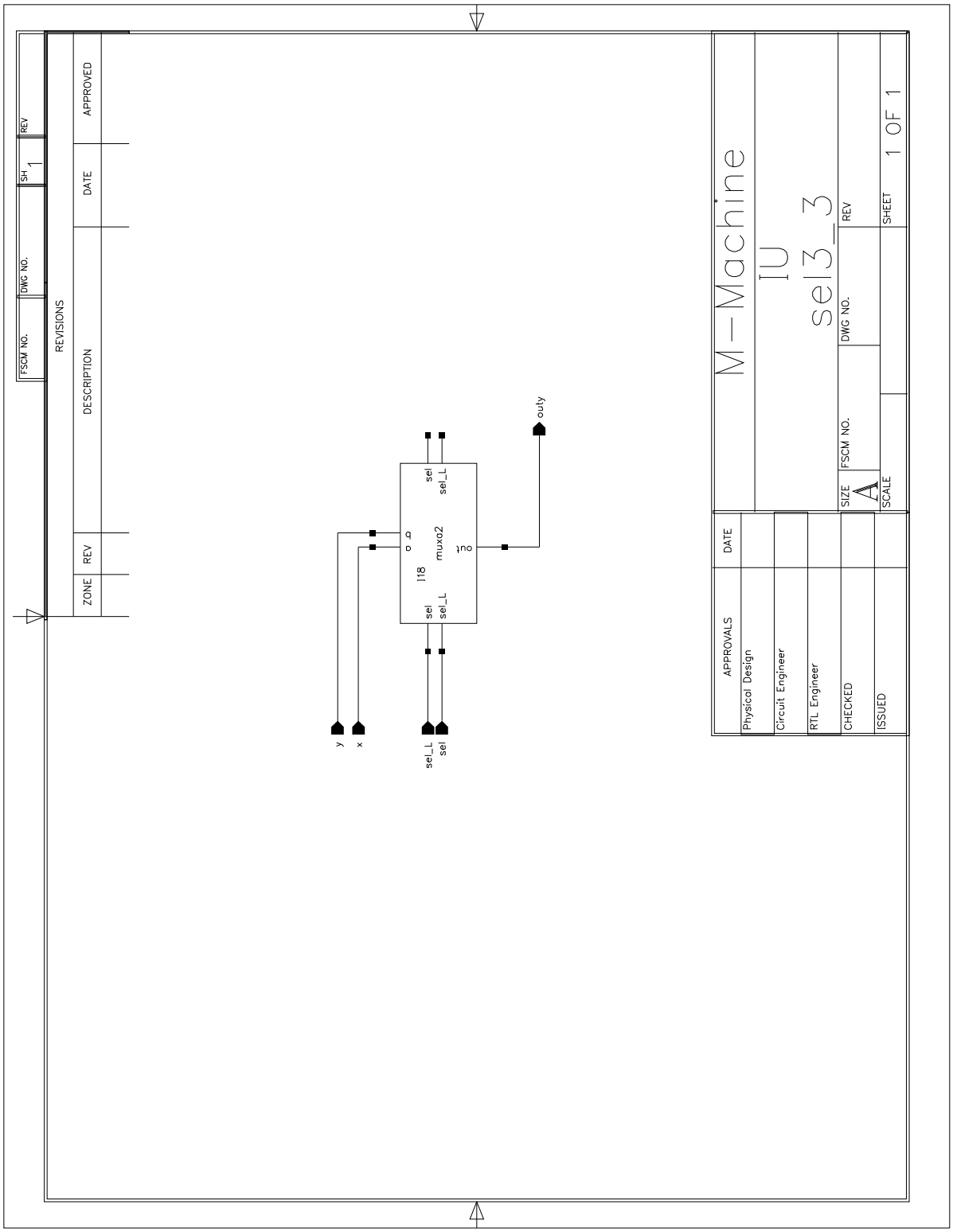

M-Machine
IU
sel4_4
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
1
REV
I20
muxb2
sel
sel_L
out
outx
y
x


REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
sel3_4
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
I18
muxb2
sel
sel_L
d0
d1
out
outy
y
x
sel_L
sel

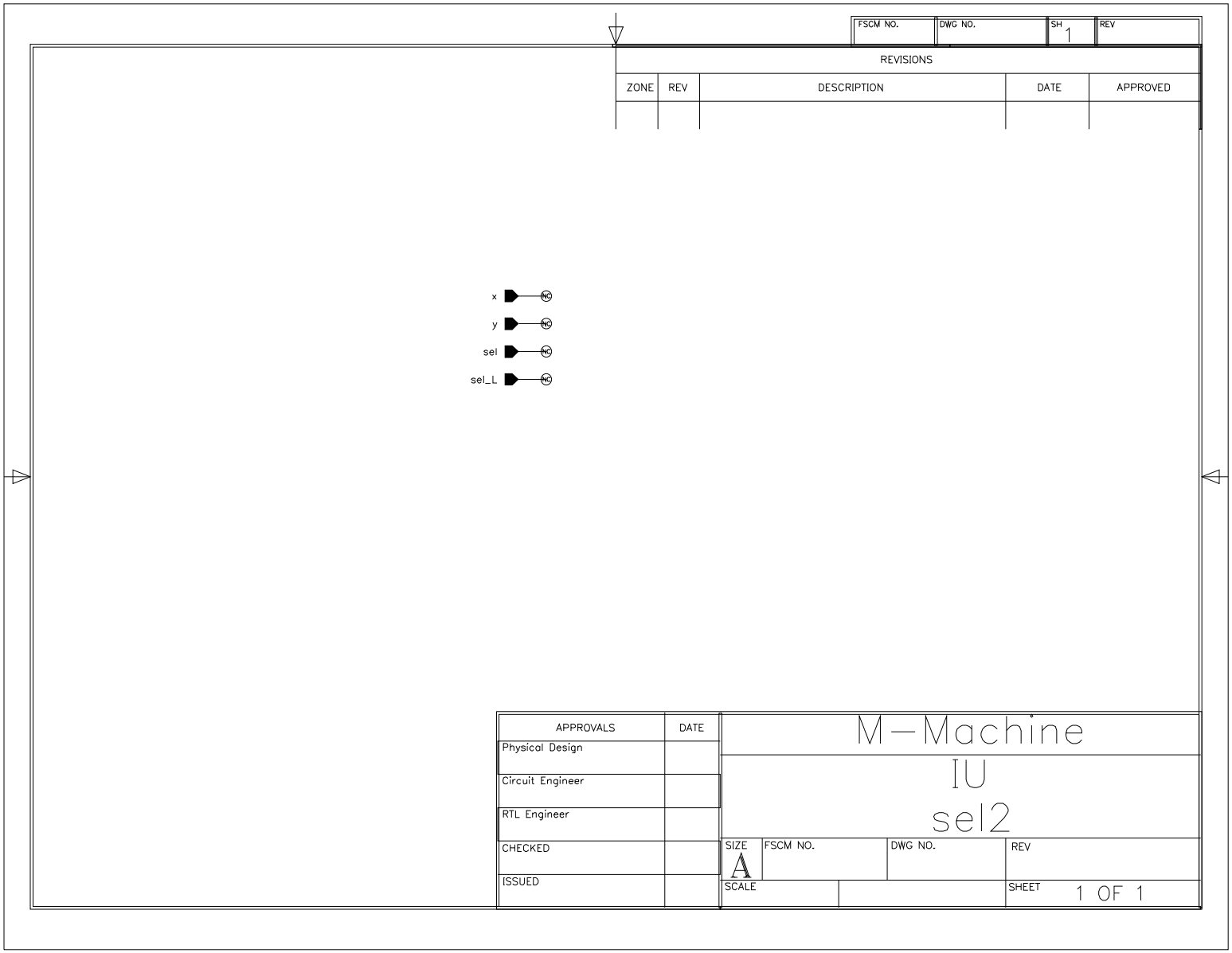

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
M-Machine
IU
muxa2
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
out
b
a
sel
sel_L

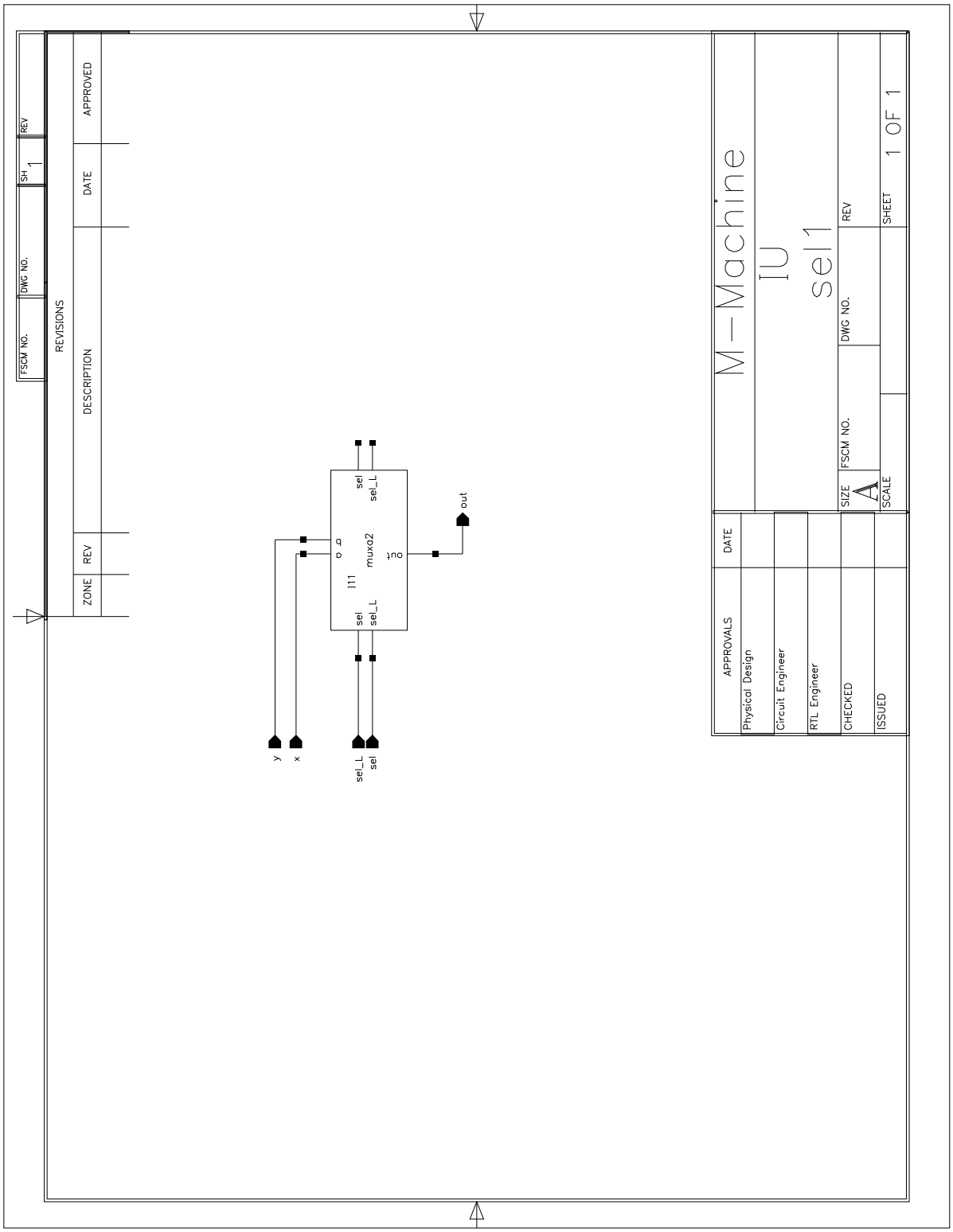


FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
M-Machine
IU
sel4_3
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
I20
muxa2
sel
sel_L
out
outx
y
x



FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
M-Machine
IU
sel3_3
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
I18
muxa2
sel
sel_L
y
x
outy

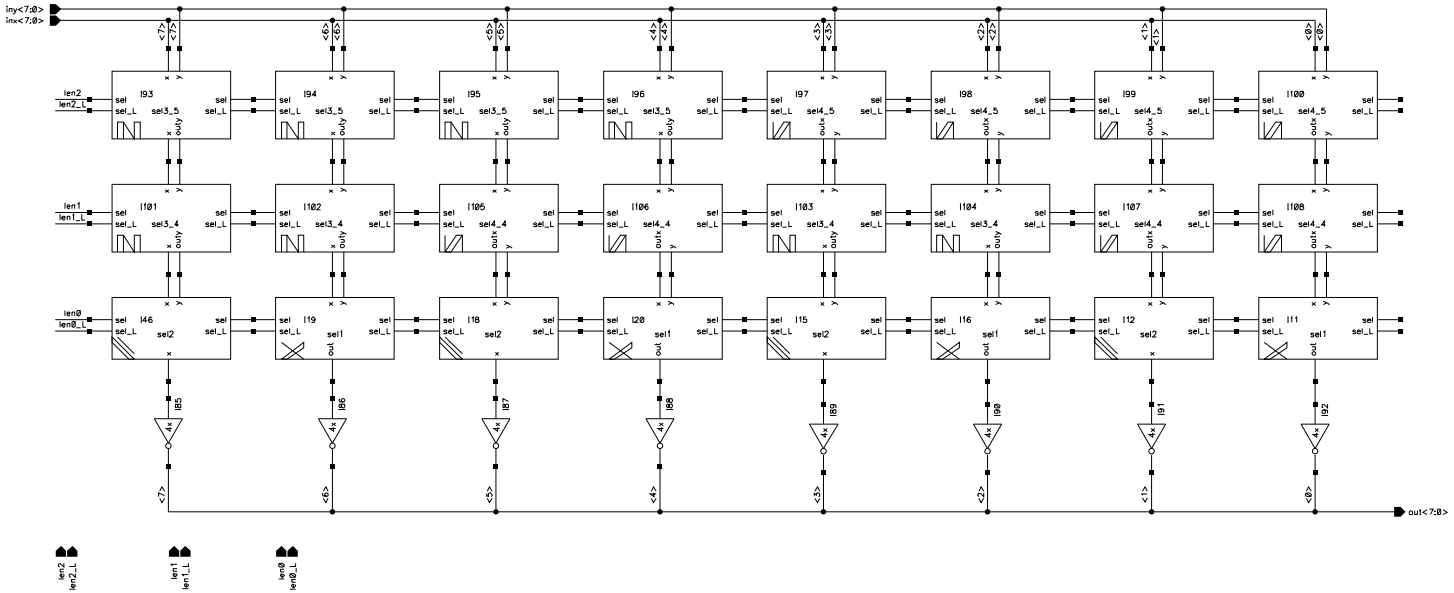

FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
x
y
sel
sel_L
NC
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
M—Machine
IU
sel2
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1



FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
M-Machine
IU
sel1
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
l11
muxa2
sel
sel_L
out
y
x

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



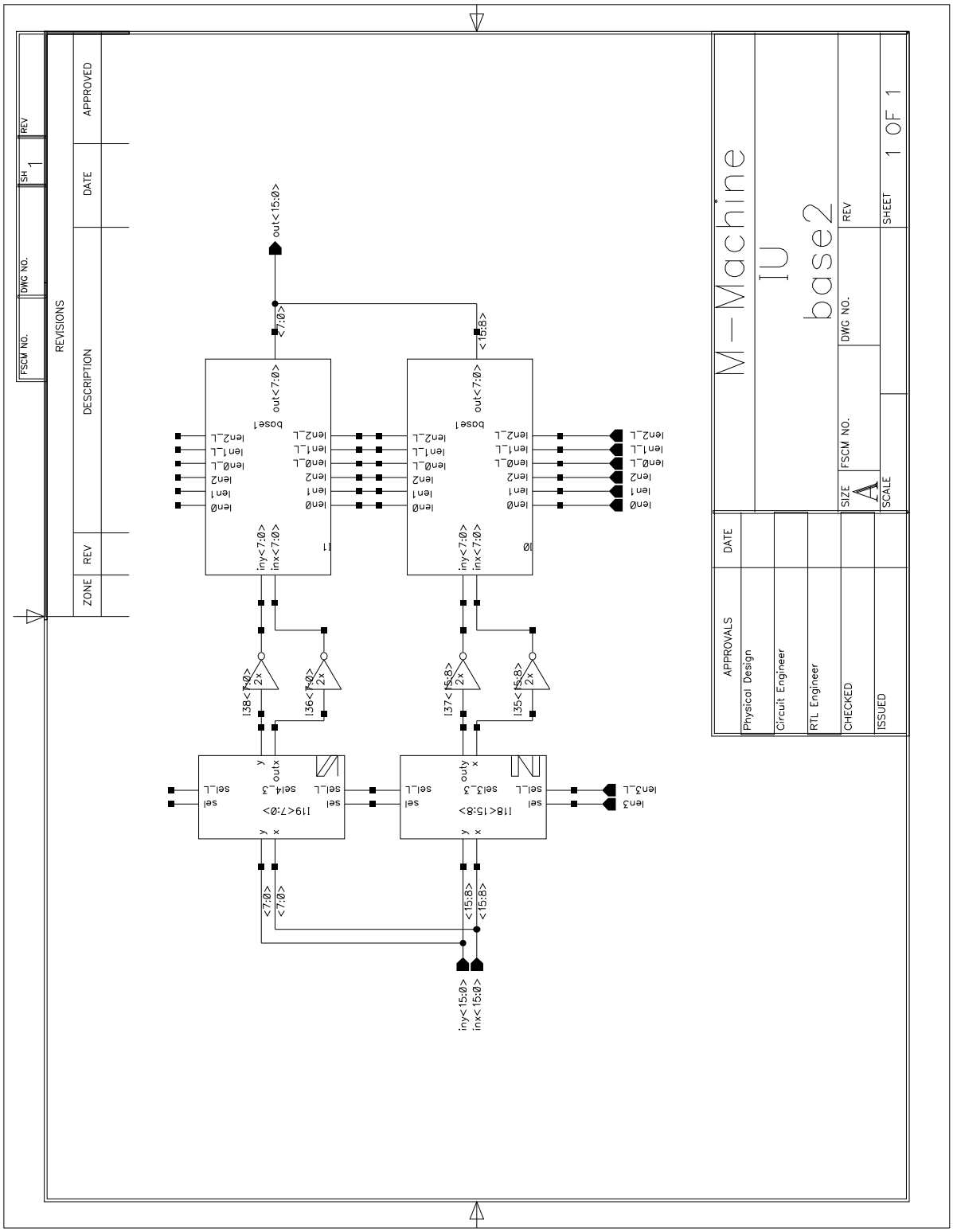
APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

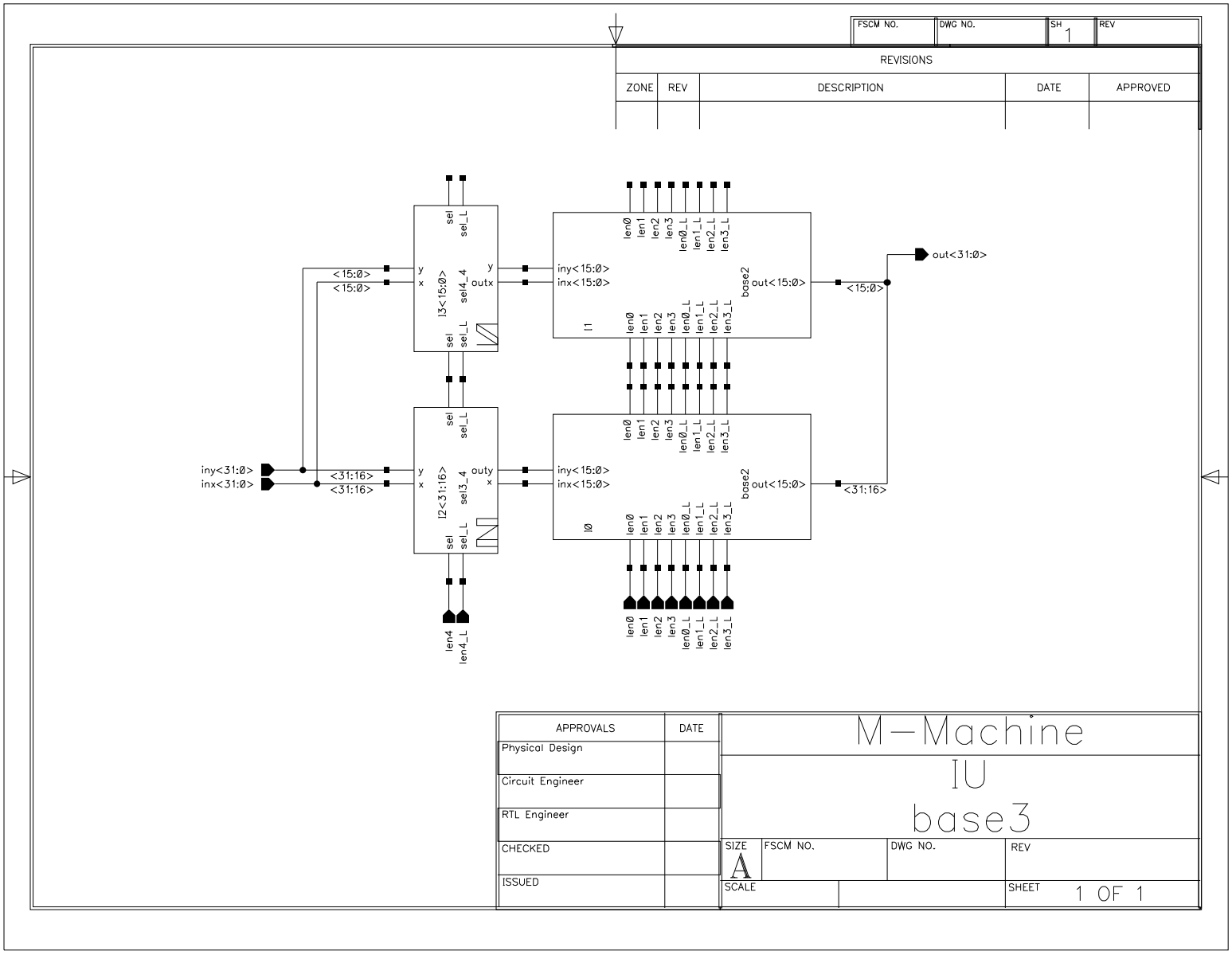
M-Machine

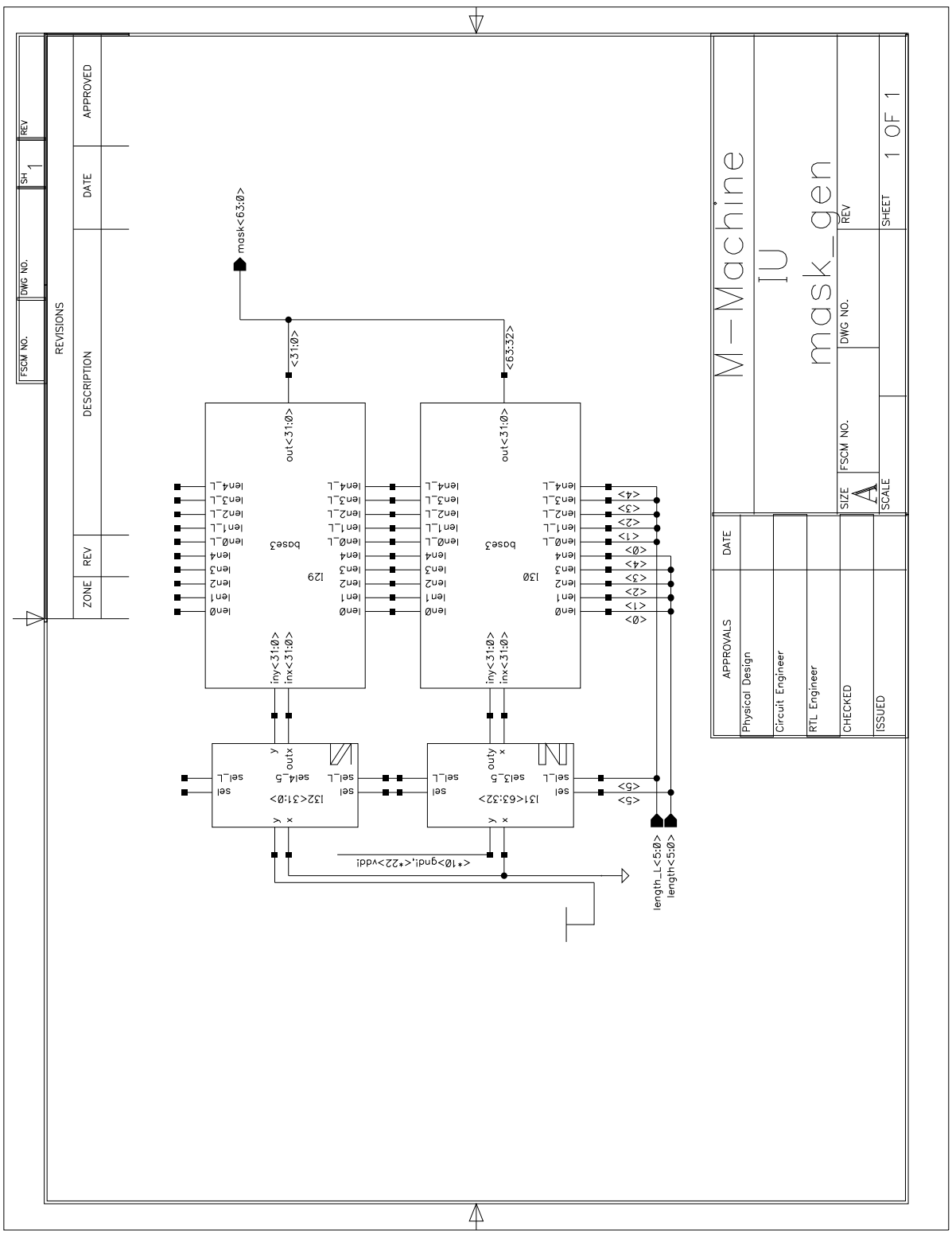
IU

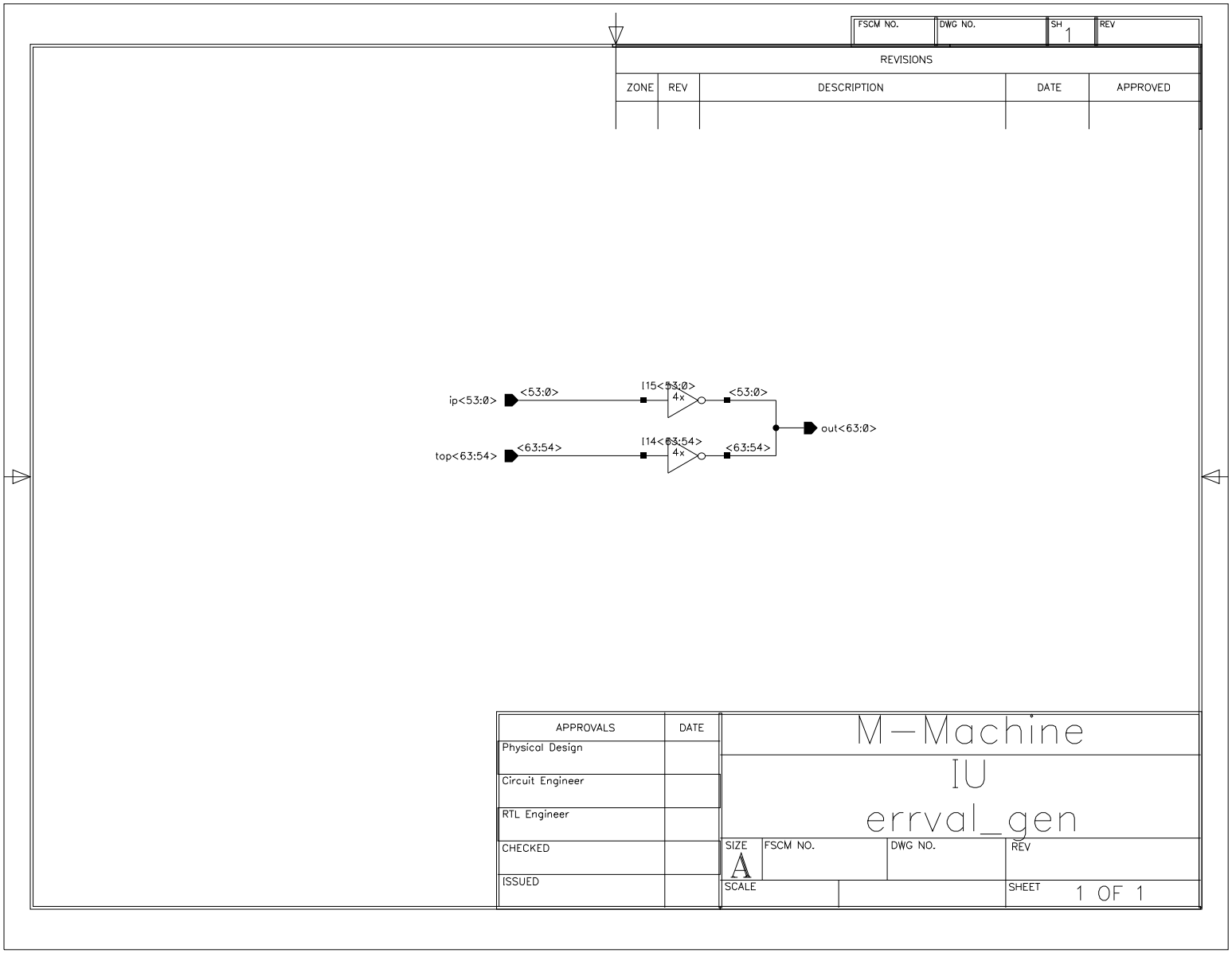
base1

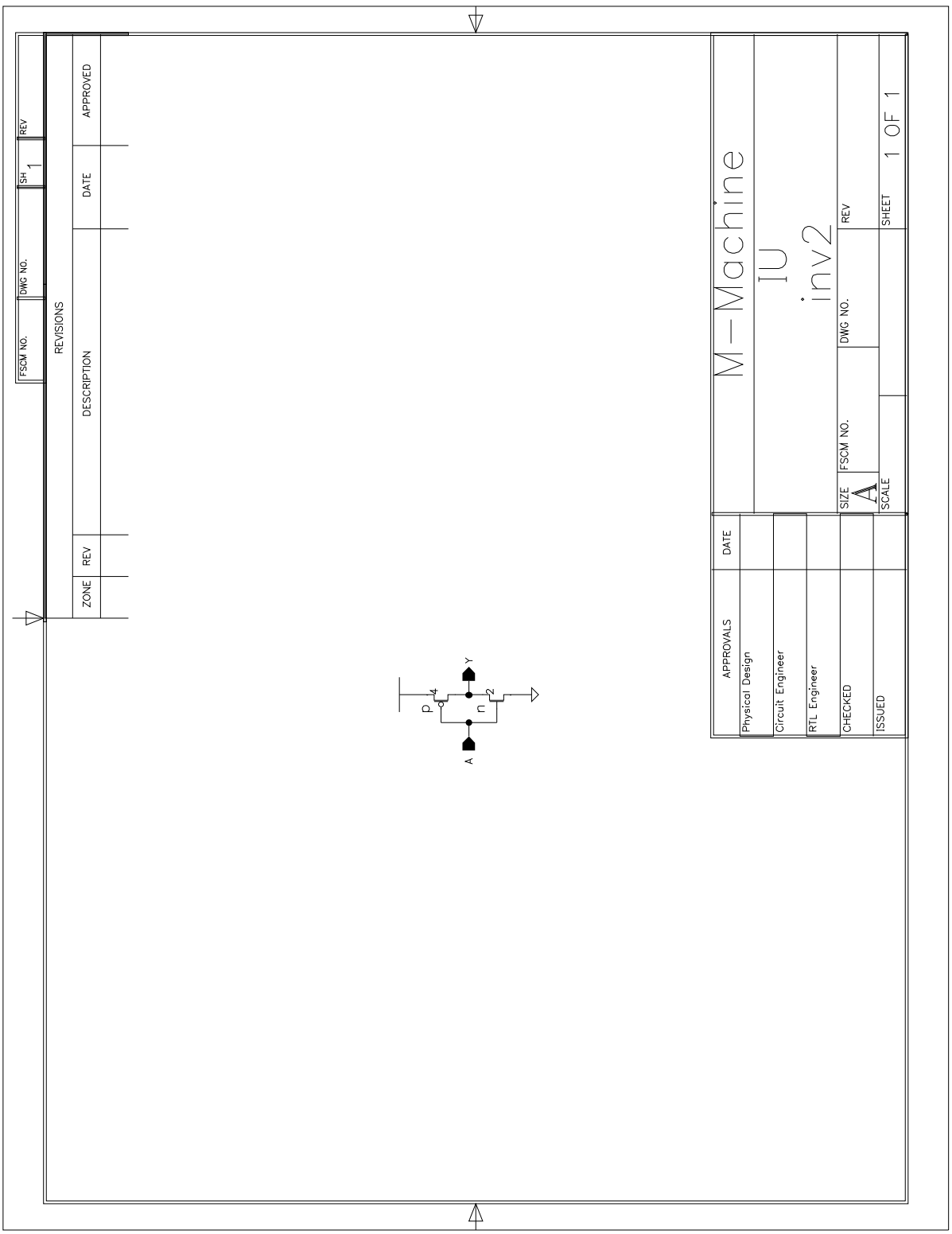
SIZE	FSCM NO.	DWG NO.	REV
B			
SCALE			SHEET 1 OF 1

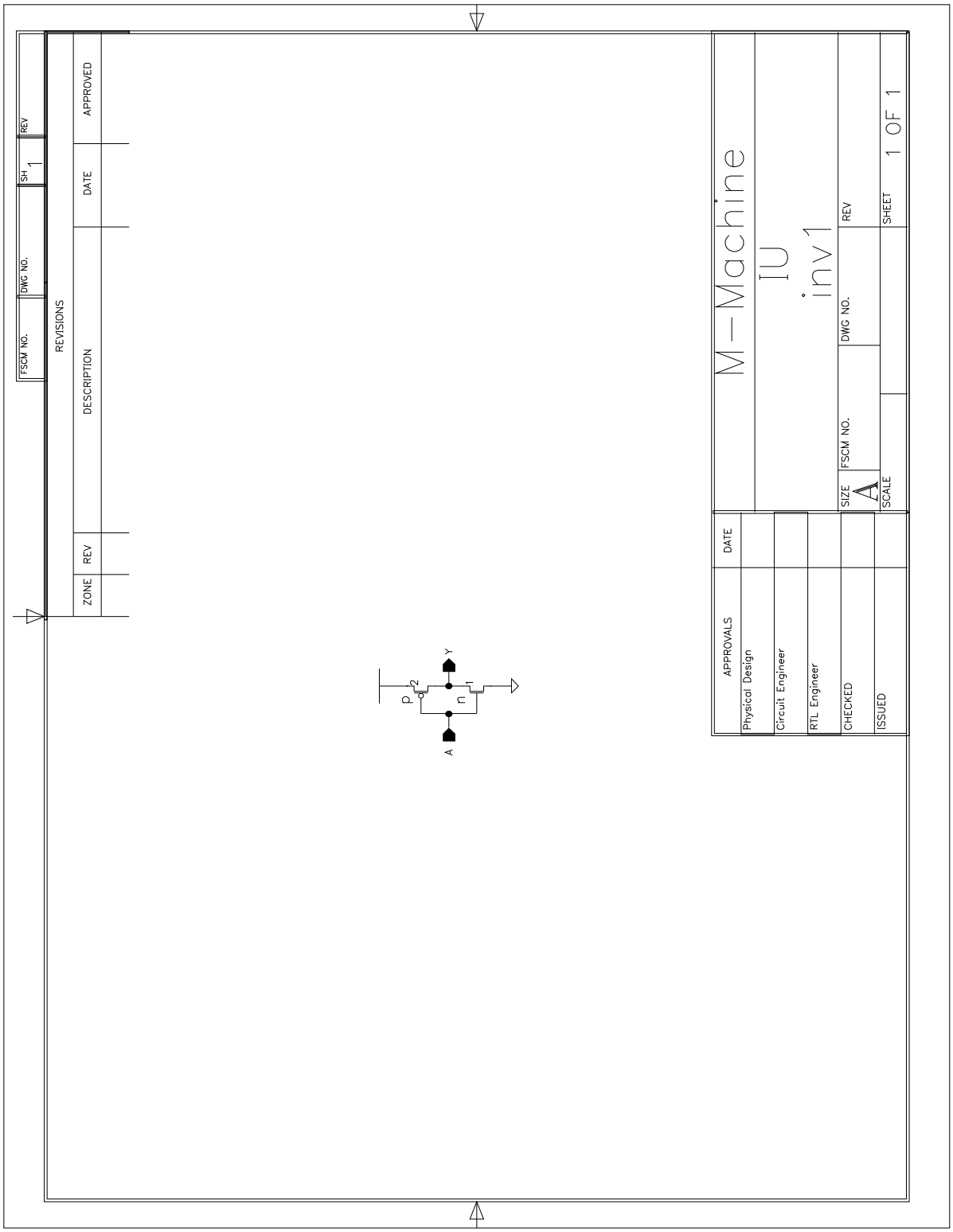

M-Machine
IU
base2
REVISIONS
DESCRIPTION
DATE
APPROVED
ZONE
REV
FSCM NO.
DWG NO.
SH 1
REV
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE A
SCALE
FSCM NO.
DWG NO.
REV
SHEET 1 OF 1
out<15:0>
<7:0>
<15:8>
out<7:0>
bose1
iny<7:0>
inx<7:0>
len0
len1
len2
len0_L
len1_L
len2_L
I38<7:0>
I36<7:0>
I37<15:8>
I35<15:8>
2x
I19<7:0>
I18<15:8>
sel
sel_L
sel4_3
sel3_3
xino
yino
xouty
len3
len3_L
iny<15:0>
inx<15:0>


FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
sel
sel_L
y
x
<15:0>
<15:0>
I3<15:0>
sel4_4
outx
iny<15:0>
inx<15:0>
out<15:0>
base2
I1
len0
len1
len2
len3
len0_L
len1_L
len2_L
len3_L
out<31:0>
<15:0>
iny<31:0>
inx<31:0>
<31:16>
<31:16>
I2<31:16>
sel3_4
outy
I0
<31:16>
len4
len4_L
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
M-Machine
IU
base3
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1


M-Machine
IU
mask_gen
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH 1
REV
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
SCALE
SHEET 1 OF 1
mask<63:0>
<31:0>
<63:32>
out<31:0>
base3
len0
len1
len2
len3
len4
len0_L
len1_L
len2_L
len3_L
len4_L
iny<31:0>
inx<31:0>
129
130
sel
sel_L
xout
yout
132<31:0>
131<63:32>
sel4_5
sel3_5
<0>
<1>
<2>
<3>
<4>
<5>
<*10>gnd!,<*22>vdd!
length_L<5:0>
length<5:0>


FSCM NO.
DWG NO.
SH 1
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
ip<53:0>
<53:0>
I15<53:0>
4x
<53:0>
out<63:0>
top<63:54>
<63:54>
I14<63:54>
4x
<63:54>
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
M-Machine
IU
errval_gen
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET
1 OF 1

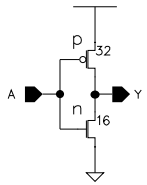

M-Machine
IU
inv2
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
p
n
4
2
A
Y


M-Machine
IU
inv1
REVISIONS
DESCRIPTION
DATE
APPROVED
ZONE
REV
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
FSCM NO.
DWG NO.
REV
SH 1
P
n
Y
A
2
1

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

M—Machine

IU

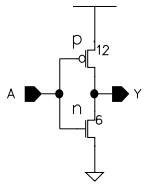
inv16

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

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

M—Machine

IU

inv6

SIZE	FSCM NO.	DWG NO.	REV
A			

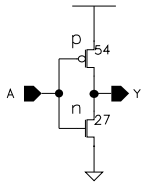
SCALE

SHEET 1 OF 1

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

M—Machine

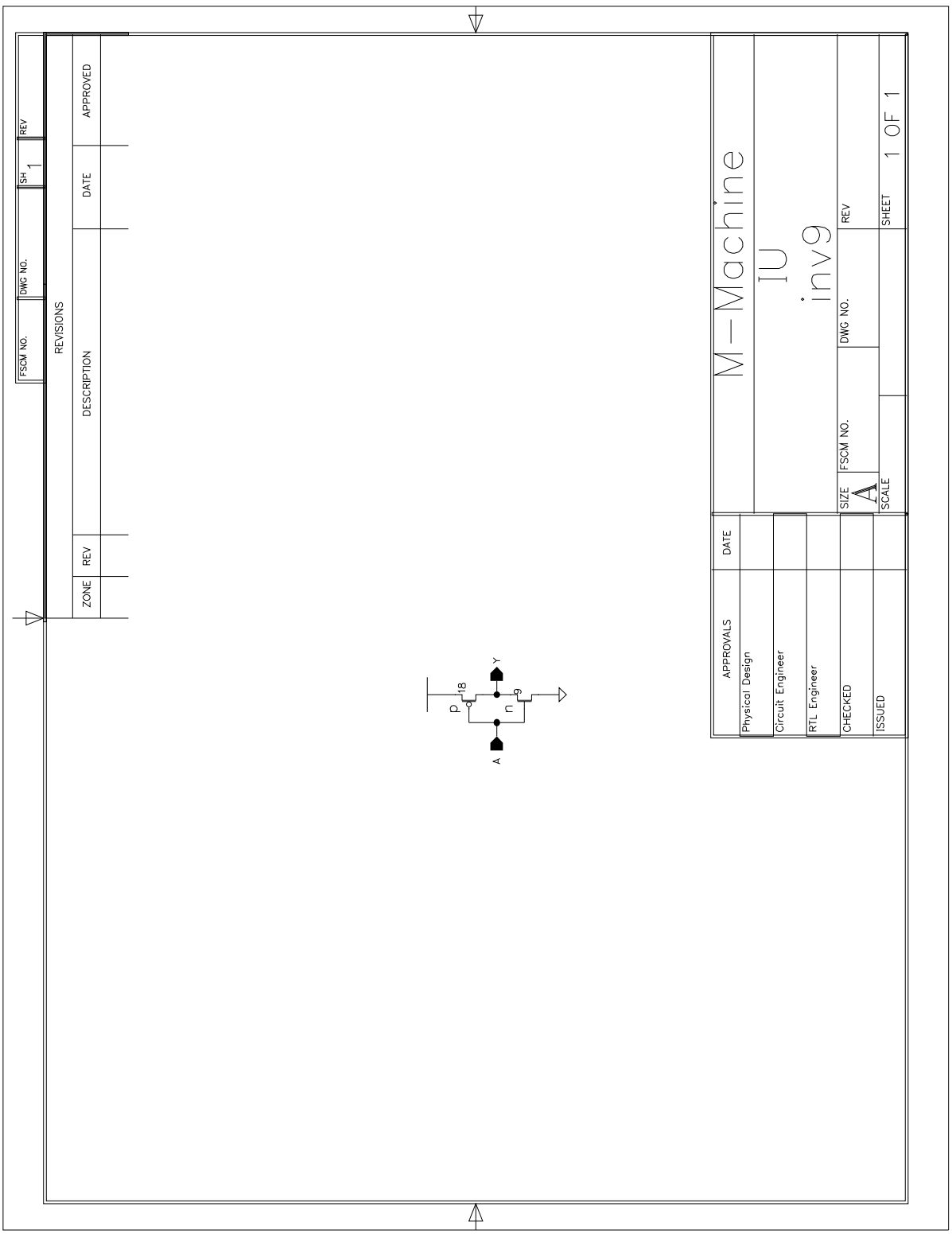
IU

inv27

SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE

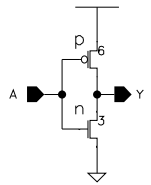
SHEET 1 OF 1


M-Machine
IU
inv9
REVISIONS
DESCRIPTION
DATE
APPROVED
ZONE
REV
FSCM NO.
DWG NO.
REV
SH 1
APPROVALS
DATE
Physical Design
Circuit Engineer
RTL Engineer
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
SHEET 1 OF 1
p
n
18
9
A
Y

FSCM NO.	DWG NO.	SH	REV
		1	

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
Physical Design	
Circuit Engineer	
RTL Engineer	
CHECKED	
ISSUED	

M—Machine

IU

inv3

SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE		SHEET
		1 OF 1

Bibliography

[1] William Dally, Stephen Keckler, Nick Carter, Andrew Changand 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.

[2] Marco Fillo, Stephen Keckler, William Dally, Nick Carter, Andrew Chang, Yevgeny Gurevich, and Whay Lee. The m-machine multicomputer. Submitted to the *22nd annual international symposium on computer architecture*, Massachusetts Institute of Technology, MIT/AI Lab, December 1994.

[3] Daniel Hartman. Design and implementation of the fpu. Master's project, Massachusetts Institute of Technology, EE/CS Department, May 1996.

[4] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*, section 4.5.4.3, page 219. Addison-Wesley, Reading, Massachusetts, second edition, 1993.