VLSI Design for Freshmen and Sophomores

by

David Harris

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

April 1994

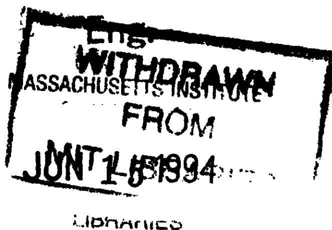
Copyright David Harris 1994. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce
and to distribute 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
April 11, 1994

Certified by ________________________________
William J. Dally
Thesis Supervisor

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



VLSI Design for Freshmen and Sophomores

by

David Harris

Submitted to the Department of Electrical Engineering and Computer Science

April 1994

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

Abstract

Very Large Scale Integration (VLSI) design has advanced in recent years to the point that a deep knowledge of semiconductor physics and a penchant for black magic is no longer necessary to design functional integrated circuits. David Harris and Professor William Dally taught a class on VLSI design to freshmen and sophomores who had an elementary background in digital electronics. Over three and a half weeks during 1994 Independent Activities Period, ten students learned to design and layout Complementary Metal Oxide Semiconductor (CMOS) circuits and implemented an eight bit microprocessor on a MOSIS TinyChip. Major results of this project are: (1) the design of the simple eight bit microprocessor, dubbed the Unintel Sexium, (2) development of CAD tools and resources to facilitate VLSI design at MIT, (3) proof that undergraduates are capable of mastering VLSI design, and (4) that a class on VLSI design would be a tremendously educational and rewarding new undergraduate laboratory subject.

Thesis Supervisor: William J. Dally
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Many people contributed in many ways to this thesis and I would like to thank them.

First of all, thanks to all of my students. Without you, this thesis never would have happened, and without your tremendous ability and hard work, the Unintel Sexium would never have been completed. You are the ones who motivated me to put in my best effort for the class. Dan Hartman deserves special credit for sacrificing 4 full Saturdays after the term had started to tracking down insidious leftover LVS errors.

Thanks moreover to my thesis advisor, William Dally. Our discussions at the freshman picnic about the possibility of teaching VLSI to freshmen and sophomores triggered the class. Your tremendous technical expertise in the field, your time preparing class lectures, and your support in all of my crazy teaching projects made the class a success. Your clever design of the Sexium architecture made it possible to fit the entire processor on a single chip. Thanks also to Professor Jon Allen, MIT's Czar of VLSI education.

Many people at the Artificial Intelligence Laboratory were a great help. Thanks to Andrew Chang for all the system administration work and for saving our files when the hard disk crashed. Thanks to Larry Dennison, Duke Xanthopoulos, and the rest of the Reliable Router group for teaching me the toolset, developing Cadence infrastructure, and teaching me a great deal about chip design. Thanks to Mike Bolotski, the God of Cadence, for more help bringing up the tools, and to Fritz Herrmann and Jeff Gealow for connecting the AI lab to the Building 39 Versatek plotter. And thanks to everybody in the CVA group who patiently when Cadence crashed and sucked CPU cycles from their machines.

MIT deserves recognition for its support in turning wild ideas into wildly successful classes. Thanks to the Edgerton Center and the IAP office for funding the class. Also, thanks to Pawan Sinha and Christopher Doerr for their cartoons that liven up this thesis.

Finally, a special thanks to my parents. Thanks for putting up with me as the most obnoxious baby that ever jumped out of his crib, for giving me support and freedom, and for sending me to MIT for some of the best years of my life! Thanks to my father, Doodyhead, for reminding me to always finish a job right.

Table of Contents

1. Introduction **8**

2. Background **9**

3. Course Description **10**
- 3.1 Course Overview 10
  - 3.1.1 The Experiment 10
  - 3.1.2 Teaching Objectives 11
  - 3.1.3 Teaching Responsibility 12
  - 3.1.4 Administrative issues 12
  - 3.1.5 Further Objectives 14
- 3.2 Teaching Philosophy 14
- 3.3 Lectures 15
- 3.4 Problem Sets 16
- 3.5 Sexium Project 17
- 3.6 Reflections on Teaching 18
  - 3.6.1 Teaching Challenges 18
  - 3.6.2 Effective Teaching Techniques 19
  - 3.6.3 Student Evaluations 20

4. Microprocessor Design **22**
- 4.1 Overview 22
- 4.2 Instruction Set Architecture 23
- 4.3 External Interface 25
- 4.4 Microarchitecture 26
- 4.5 Verilog Model 30
- 4.6 Schematics 31
- 4.7 Floorplan & Layout 31
  - 4.7.1 Layout Design Rules 31
  - 4.7.2 Floorplan 32
  - 4.7.3 Datapath Connectivity Diagrams 33
- 4.8 Cell Descriptions 34
  - 4.8.1 Top Level 34
  - 4.8.2 alubox 34
  - 4.8.3 regbox 36
  - 4.8.4 pcmabox 37
  - 4.8.5 control 38
  - 4.8.6 Pads 38
- 4.9 Timing 39
  - 4.9.1 Signal Timing 39
  - 4.9.2 Tau Model 40
  - 4.9.3 Critical Paths 41
- 4.A Verification 43
  - 4.A.1 Schematic Simulation 44
  - 4.A.2 Layout vs. Schematics 44
  - 4.A.3 Layout Simulation 44
  - 4.A.4 Final Verification 45

5. Tool Development **46**
5.1 Sexium Architecture Tools 46
5.1.1 msim 46
5.1.2 Regression Suite 46
5.2 Cadence Tools 47
5.2.1 mosis2n Library and Technology File 47
5.2.2 Standard Cell Library 47
5.2.3 PLA Generator 48
5.2.4 Pad Frame 48
5.2.5 Setting up Cadence for another class 49

6. Conclusions **50**
6.1 Educational Results 50
6.2 Project Results 51
6.2.1 Unintel Sexium Summary 51
6.2.2 Tool Summary 51
6.3 Summary 52
6.4 Recommendations 52

Bibliography **55**

Appendix A: Course Handouts **56**
Syllabus & Administrivia 57
Class Roster (Revised) 59
Machine Assignments 60
Course Evaluation 61
Cadence Reference
Part 1 63
Cadence CAD Reference
Part 2 66
Cadence CAD Reference
Part 3 68
Sexium Design Rules 71
Standard Cell Reference 75
Unintel Sexium 76
Problem Sets 102
Scribe Notes 135

Appendix B: Student Evaluations **174**

Appendix C: Sexium Design Files **195**
Sexium Schematics 196
Sexium Layout 233
Verilog Model 242
Microcode PLA Equations 256
HSPICE Simulations 260
PLA Simulation 261
I/O Pad Simulation 263

Appendix D: Tools **266**
Locating tools on-line 267
msim 269
regress.asm 285
plagen.il 291

1. Introduction

This thesis describes an educational experiment, bringing VLSI design to freshmen and sophomores at MIT during the 1994 Independent Activities Period. The experiment involved preparing an IAP class, developing lectures, problem sets, and notes, creating and documenting CAD tools, and designing and implementing an 8 bit microprocessor for the class project. David Harris co-taught with Professor William Dally; Professor Dally was responsible for four of the lectures, the overall processor architecture, and for guidance and a VLSI education philosophy throughout the project. David Harris and the ten students carried out the remainder of the work, detailed in this document.

The educational experiment produced overwhelmingly positive results. Hence, we contend in this thesis that VLSI ought to be integrated into the MIT undergraduate curriculum. Chapter 2 presents background about events motivating and leading to the VLSI Chip Design class. Chapter 3 describes the course in more detail, beginning with the objectives and teaching philosophy and showing how they manifest in the lectures, problem sets, and class project. Chapter 4 discusses the class project, the design and implementation of the Unintel Sexium 8 bit accumulator-based microprocessor. This project was a significant engineering project in itself, and the process flow, from Verilog to schematics through layout and verification, is examined. Chapter 5 documents several of the CAD tools developed to support the project; much of the work should be reusable in other VLSI classes at MIT. Chapter 6 reports the results of the educational experiment and concludes with the case for bringing VLSI into the undergraduate Electrical Engineering and Computer Science curriculum.

2. Background

Very Large Scale Integration (VLSI) Application-Specific Integrated Circuits (ASICs), including gate arrays and custom chips, have nearly replaced traditional TTL logic for implementing today's ever more complex digital systems. Moreover, the design of VLSI systems has become much simpler in the last decade; most of the black magic is gone. Nevertheless, digital electronics education continues to revolve around TTL based systems. While TTL chips do offer the advantage of being very concrete and easy for novices to use, it would seem that VLSI should be taught as well because a good understanding of the capabilities and limitations of the VLSI medium is critical to being a real-world digital designer.

MIT has pioneered VLSI education in the past, especially via Mead and Conway's revolutionary class in 1978, and presently offers a variety of courses on VLSI design at the graduate level. 6.371 is a very popular introductory graduate course; in recent semesters, due to the influx of Masters' students under the new Master of Engineering program, it has been severely oversubscribed and undergraduates have little hope of enrolling. A modest variety of more advanced coursework in VLSI is also offered, but nothing is available for the bulk of undergraduate students.

This author became interested in teaching VLSI after spending three semesters teaching digital electronics to freshmen at MIT. The freshman course, unofficially titled "6.007: License to Hack," seeks to give freshmen a taste of electrical engineering and enough practical electronics background to make the student comfortable designing real-world digital circuits and working with soldering irons, breadboards, and various discrete components. It is taught with a sequence of design projects and hands-on labs, culminating with a digital tape recorder and a student-designed robot.

In the fall of 1993, David Harris and Professor Dally, the faculty advisor to 6.007, decided to experiment with teaching VLSI to freshmen and sophomores with only basic digital electronics background. They developed a 9-unit course, "6.008," and taught it to a class of ten students during Independent Activities Period, January 1994. Seven of the students were 6.007 graduates: two freshmen and five sophomores. One of the remaining students was a sophomore who had other digital background; the last two were juniors with extensive background including 6.004 and 6.313.

3. Course Description

Teaching is the half of learning.
— Confucius, in Record on the Subject of Education

This chapter describes the VLSI Chip Design course. A detailed description of the Unintel Sexium microprocessor is deferred to Chapter 4. Copies of the handouts prepared for the class appear in Appendix A.

3.1 Course Overview

3.1.1 The Experiment

This class can be viewed as an experiment in undergraduate education. The hypothesis is that freshmen and sophomores with a basic knowledge of digital electronics are capable of learning VLSI design, especially circuit design and layout. A three-part metric is used to measure success:

- Student performance on problem sets
- Student performance implementing complete chip
- Student enjoyment of the material

The method of instruction was a sequence of lectures, problem sets, and the final chip design. Teaching took place over three and a half weeks during MIT's Independent Activities Period. Students attended class every Monday, Wednesday, and Friday[1] from 2-4 PM in the Athena Electronic Classroom, for a total of 11 lectures. Between classes, students were expected to complete 6-12 hours of outside work.

3.1.2 Teaching Objectives

The objective of the class was to teach the students the following:

- Static CMOS circuit design
- CMOS layout technique
- Higher-level design: floorplanning, standard cells, datapaths, and regular arrays
- Managing complexity through hierarchy, modularity, and regularity
- Team design

Note that issues such as speed and power dissipation were intentionally omitted; although these are necessary for real-world design, they require a background in E&M and device physics not available to most freshmen and sophomores. Moreover, these issues can be learned from a textbook or job after the basic concepts have been mastered. The first three topics are typical of any VLSI class; the latter two are especially important because this class may be the first exposure of freshmen and sophomores to complex engineering design. Hands-on experience doing an actual complex design is the best way to teach appreciation of engineering methodology.

To meet these objectives, the class followed the syllabus below:

M3 Introduction. Administrivia. Switch-based circuits. MOSFETs. Examples.
W5 Chip Fabrication. Design Rules. Layout. Low-level examples.
F7 High-level layout. Floorplanning.
M10 Flip-flops in VLSI. Counter example.
W12 Regular arrays: RAMs, ROMs, PLAs. PLA examples.
F14 Class project. Overview of microprocessors. Programming Unintel Sexium.
T18 Microarchitecture of Sexium. Trace of program execution.
W19 Class project issues. Project lab.
F21 Project lab continued.
M24 Pads, electromigration, and other Deep Dark Secrets.
W26 Fabricating chips. Future directions. End-of-class party.

[1] Monday, January 17 was a holiday, so that class was rescheduled for Tuesday, January 18.

3.1.3 Teaching Responsibility

Responsibility for teaching the class was divided between Professor Dally and David Harris. The two jointly brainstormed and wrote the syllabus. Professor Dally was responsible for the overall Sexium architecture, especially making it an accumulator machine built from a single bus, allowing an area-efficient implementation, and for specifying the datapaths and writing the RTL description of the microcode. Professor Dally also delivered formal lectures introducing major concepts on Monday the 3rd, Wednesday the 5th, Monday the 10th, and Wednesday the 12th. He reviewed the problem sets and the Sexium implementation. Unfortunately, he was on travel during the second portion of the month, but he was able to continue with design reviews over electronic mail.

David Harris was responsible for the remaining work. He wrote the class proposal and obtained funding and a listing in the IAP guide. He brought up the CAD tools and technology files required for the class. He developed the Sexium processor, from an assembler and Instruction Set Architecture (ISA) simulator written in C, to a Verilog model, to gate and transistor level schematics and a floorplan. He wrote and graded the problem sets, and delivered the remaining lectures. He also produced extensive documentation of the tools and project.

The class proved to require more preparation than anyone had anticipated, but was tremendously educational and enjoyable for all involved.

3.1.4 Administrative issues

"6.008" was officially listed as 6.090, Special Subjects in Electrical Engineering and Computer Science, and named VLSI Chip Design in the IAP guide. Anne Hunter in EECS department provided the number; the IAP office provided the listing. It was first listed for 6 units of credit, but was upgraded to 9 units midway through the month because the students were investing significantly more than 6 units of work. To receive pass/fail credit, students were expected to do the following:

- Attend at least 9 of the 11 classes
- Complete 5 of the 6 problem sets

- Contribute to final project
- Act as scribe, recording one lecture

There were more students enrolled than lectures to scribe because some of the later sessions proved just to be design labs, so some students did not scribe.

Class was scheduled in room 1-115, an Athena Electronic classroom. The classroom has approximately 30 color workstations, one at each seat, as well as an instructor's workstation connected to a projection display. The display was very useful for illustrating the CAD tools and layout technique. The classroom was reserved by sending email to eclass@mit.edu; more information is available by adding the "info" locker on Athena.

The estimated budget for the course is listed below:

Fabricating MOSIS TinyChip	$420
13 copies of chip plot	$130
Design Project Ice Cream prizes	$23
Final party food & utensils	$10
Total	**$583**

The course received funding from three sources. Course 6 refused to provide funding for IAP activities, so the Edgerton Center sponsored the seminar and provided $250. The IAP office provided an additional $250. Professor Dally offered to cover the remainder through his discretionary funds.

3.1.5 Further Objectives

In addition to experimenting with VLSI education and teaching the students, David had two other objectives. One was to build an infrastructure for using the Cadence tools in other classes at MIT. The second was to design and build a microprocessor.

MIT has used the Mentor Graphics GDT tools in the past. Unfortunately, Mentor has been doing a poor job maintaining the tools and keeping them up to date with recent technology; thus many of the laboratories, including the Artificial Intelligence Laboratory and Microsystems Technology Lab, have shifted to the Cadence toolset that was donated to MIT in 1993. The 6.371 teaching staff ultimately intends to shift to the Cadence tools, but was unable to get them ready for use in the fall of 1993. The tools have a very steep learning curve and require an extensive set of technology files to work properly.

David attempted to develop enough infrastructure around the Cadence tools so that they could be used by other classes without requiring as much knowledge and effort on the part of the teaching staff. This involved developing a technology file to support the mosis2n 2 μm N-well process, properly importing a good pad frame (pad frame problems have chronically plagued 6.371), and writing a PLA generator general enough to be used by other classes. It also involved documenting the tools at a level accessible to novices.

The remaining objective was more personal: ever since taking 6.004, David had been fascinated by designing microprocessors. Teaching this class was a fabulous opportunity to do a complete design, from architecture down to layout and verification. It was also a chance to learn from the mistakes he had made on the Reliable Router project and to apply his experience with the Cadence toolset.

3.2 Teaching Philosophy

Both David and Professor Dally had strong opinions about how a class of this nature should be taught. Major points include:

- Conveying excitement about VLSI design and engineering in general
- Learning through hands-on work and original design
- Intuitive, rather than mathematical, approach to fundamental issues
- Provide practical background for UROPs / summer jobs

- Never teach false models that must be unlearned later

To keep the class accessible to freshmen who may not have even studied electricity and magnetism, we carefully restricted the topics covered. In particular, we avoided issues requiring a quantitative understanding of timing or device characteristics. We selected an edge-triggered clocking discipline with a locally generated clkbar signal to avoid issues of skew. We used minimum-sized devices in all places. We limited circuit design to fully restoring static circuits so that students could not get themselves in serious trouble with analog effects; however, as the students developed a better understanding of the design rules, we pointed out circumstances under which the rules could be violated to reduce critical area, especially by using NMOS-only transmission gates and pseudo-NMOS PLA circuitry.

Another challenge that arose was the varied experience levels of the students. Many of the students had taken little or no electronics beyond the basic 6.007 class; however, several had mastered 6.004 and even 6.313. Properly targeting the lectures such that the advanced students learned new material and remained interested while the less experienced students could still understand was very challenging, especially for many of the early lectures that addressed fundamental principles of digital electronics. The most effective answer to this experience differential was providing extra credit problems on the homework that allowed students to do original design and optimization. The best solution on each extra credit problem was rewarded with a Toscinini's ice cream gift certificate.

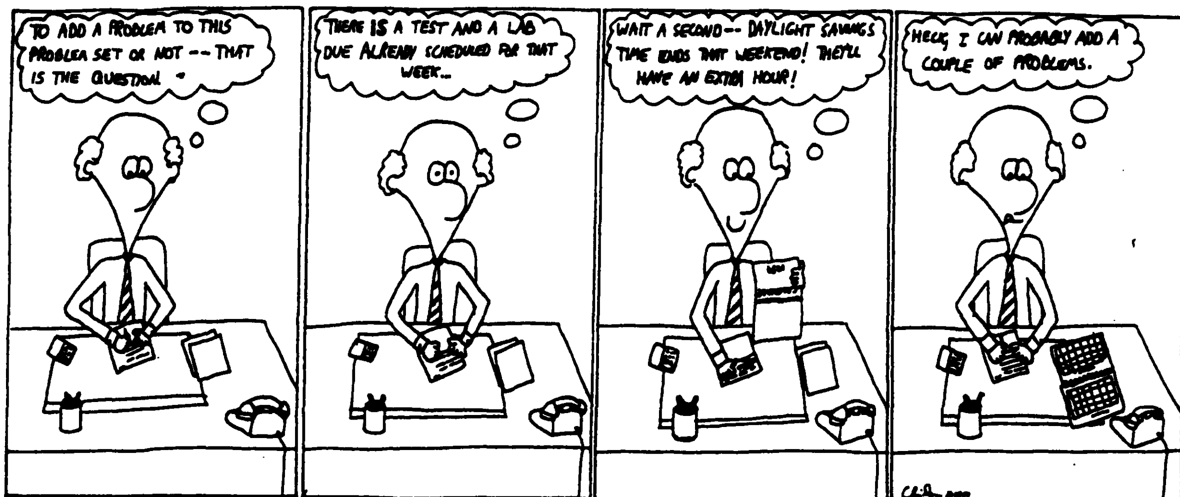
3.3 Lectures

The first seven lectures were typical of formal MIT lectures. They tended to be paced rapidly, occasionally faster than the less-experienced students could follow. The remaining four class periods focused on project-related issues and had little or no formal lecture time. The syllabus in section 3.1.2 summarizes the material that was covered.

For each of the first six lectures, a student volunteer acted as the scribe, recording the lecture, typing it, and distributing copies to class members. This had three purposes: to provide a reference on the lecture to the other students, to provide a written record of the lecture in the event that the class might be taught again in the future, and to help the scribe learn the material in more depth than usual. The scribe's notes were reviewed and

edited on the evening before the succeeding class to catch conceptual and linguistic errors and to answer any questions the scribe may have developed while writing; the corrected notes were then photocopied and distributed. Copies of these notes appear in Appendix A.

3.4 Problem Sets



At each class, a problem set related to the lecture material was assigned, due two days later at the next class. The problem sets were design and layout-oriented; while the first problem set primarily involved design of logic circuits using MOS transistors on paper, the remaining problem sets all required students to log into computers at the Artificial Intelligence laboratory and run the Cadence layout software or the msim simulator developed for the class. After Problem Set 6, the students switched to laying out the Sexium processor and began working as teams instead of doing individual assignments. Problem sets were graded loosely on a √- / √ / √+ system; the primary goal was to help students master the material by actually doing it. A copy of all of the problem sets appears in Appendix A.

Problem Set 4 involved the design of standard cells by the students. Seventeen of the most frequently used cells were selected for the standard cell library; they were assigned such that two students designed each of the cells. Given transistor-level schematics and guidelines on standard cell design rules, the students produced layout. The most compact

versions of each cell were chosen for the library; in cases where neither cell was correctly laid out, the student with the most nearly correct design was asked to revise the cell. The flip-flop was a special case; it was offered as an extra-credit design project to create the most compact flip-flop. A great competition ensued and a very compact flip-flop was produced by Matt Sakai. Individual design reviews were held afterward with each student to clear up confusion about well contacts, DRC rules, standard cell design, and so forth.

3.5 Sexium Project

The most exciting part of the class for many students was the final project, in which the class implemented the Unintel Sexium microprocessor. A technical description of the processor appears below in Chapter 4; this section describes the work the students carried out.

The students were presented with a description of the Sexium architecture and a draft of the floorplan and schematics; it was emphasized that these documents were tentative and that any optimizations students found were highly encouraged. On Wednesday, January 19th, students were divided into four teams, responsible for the control, alubox, regbox, and pcmabox. Each team was led by a captain who had learned the Layout vs. Schematic (LVS) verification tools and who had shown mastery of the course. The teams were each assigned a set of datapath cells and expected to layout a single bitslice of each cell.

On Friday the 21st, the completed bitslices were examined and it was found that the datapath was 2500 lambda wide, much wider than the 1800 lambda available; moreover, it was found that the control logic would occupy most of the upper half of the chip, precluding a large second row of datapath cells. The class analyzed the factors leading to the excessive area and brainstormed solutions. Several students found ways to eliminate a few lambda on many cells; the greatest area savings was achieved by incorporating multiplexor cells directly into the modules they were driving, eliminating the two inverters used to restore the logic levels. Since multiplexors were so common in the logic design, this provided a significant area savings. Students continued with the design by assembling the bitslices into full 8 bit datapaths and placing a row of standard cells atop the datapath to provide factored control signals (e.g. clkbar, or enables on the latches).

By Monday the 24th, nearly all of the datapath cells passed LVS and DRC checks and the control team had written the equations for the PLA, adapting them from the microcode David had written in Verilog. The datapath proved to be about 40 lambda wider than the chip after extensive optimization, so the relatively small regbox was moved to the upper half of the chip, still leaving enough room for the control PLA and standard cells. While the control team finished its design, the datapath was placed in the pad frame and wired together; power and ground busses were also run across the center of the chip. Dan Hartman learned to run the Verilog simulator and assisted with the Verilog simulation of the schematics; he and other students also drew schematics for the control box and test multiplexors.

On Wednesday the 26th, the last day of class, the datapath was complete and wired up. The control was also nearly complete; a last minute bug had been located and corrected in the microcode and the PLA was regenerated. A nifty Sexium logo was designed and the global wiring was nearly finished.

Verification and final modifications took longer than expected: many modules had incorrectly placed pins that were difficult to locate; the Cadence extractor also had difficulty with the different scale on the pad frame, which had been read in from CIF (Caltech Interchange Format). Dan Hartman and David Harris spent each Saturday in the lab hunting down bugs and produced final CIF output on February 24, 1994.

3.6 Reflections on Teaching

3.6.1 Teaching Challenges

There were several challenges involved in teaching VLSI to freshmen and sophomores. The greatest challenge was balancing the level of presentation to keep the more experienced students interested while retaining clarity of explanation for the students with little background in electronics. Teaching enough VLSI for students to do layout and use the tools required a rapid pace; however, it was more important that the students understand the material than that every facet of VLSI design be covered. From the student performance and overwhelmingly favorable evaluations received, one freshman felt he did not master the material; the other students felt reasonably comfortable and Dan Hartman learned enough to proceed to the advanced 6.372 class in the spring.

Another challenge was simply developing the course. Designing an entire microprocessor in a month was a reasonable effort; writing and grading problem sets, maintaining notes, and so forth was a mammoth effort. Mmuch of this work will not have to be repeated, should this course be offered next year.

3.6.2 Effective Teaching Techniques

Several teaching techniques proved valuable in this class and are summarized below. They include the hands-on focus, the highly motivational project, extra credit problems on each assignment, design reviews, scribe assignments, and the electronic classroom.

The key to successfully teaching VLSI in such a short time to novices was the hands-on emphasis. Some students learn well from lecture; others learn from textbooks; however, everybody seems to learn well by doing. There is no substitute for actual design work when teaching design. The first problem set involved paper designs; on every subsequent assignment, students did most of their work on the computer actually doing layout, programming, or simulation and verification. Problem sets were graded informally, but allowed David to catch students who were having conceptual difficulties and fix misunderstandings before they became serious.

The second key was the highly motivational project of designing and building the Unintel Sexium. Many of the students, especially team leaders, acquired a personal sense of responsibility for the project and worked into the wee hours of the nights searching for optimizations and bugs and producing extremely tight layout. For the students, having completed a chip and hanging a plot of it on the wall builds confidence as engineers and excitement about design. As a teacher, the project kept the class exciting throughout, despite the tremendous amount of effort required.

A mid-month design review of each of the students was critical to prepare everybody for the project.. By Problem Set 4, students had learned everything they needed to know about layout and developed cells for the standard cell library. Many of the students had minor misconceptions about layout: some didn't include substrate contacts, pins, or labels, others did not push design rules aggressively enough and ended up with enormous layouts. An individual conference with each student in which the standard cells were evaluated and corrected where necessary cleared up these problems and prepared the students to produce correct layout for the final project.

As noted earlier, the disparity among experience levels presented a challenge for teaching. The most effective approach to the problem was offering extra-credit problems on the problem sets. This allowed the problem set to focus on the basic issues for the students with little experience, while giving the advanced students a chance to do interesting original design and optimization. Ice cream certificates as prizes added a bit of motivation to the assignments for less than 5% of the total class budget.

Scribe assignments were another experimental technique that worked well. Traditionally, only a few graduate classes have employed student scribes. In 6.008, a volunteer was chosen at each meeting to record the lecture and type it for distribution at the next class. Many students found that they better understood the lecture after trying to type it up and asking questions where they found that something from lecture was unclear. The notes served as reference material for the other students. As a first time class, the notes will also give a record of the material covered in each lecture to make future classes easier. One issue of scribing was difficult: as there were only six formal lectures in which the scribes took notes, and ten students, it was to the advantage of students not to volunteer, in the hopes that they would never have to scribe. A solution to this problem was giving the scribe responsibility to any student who arrived late when nobody else would volunteer!

The electronic classroom was an interesting and useful teaching resource. As such a hands-on class, there were many opportunities where the students could try layout in class with help and supervision; even more importantly, the tools and layout styles could be illustrated on the projector during class. The disadvantages of the electronic classroom are that students tend to be logged in, sending zephyrs or playing computer games during lecture, and that when the lights are dimmed to use the projection screen, it is easy to fall asleep.

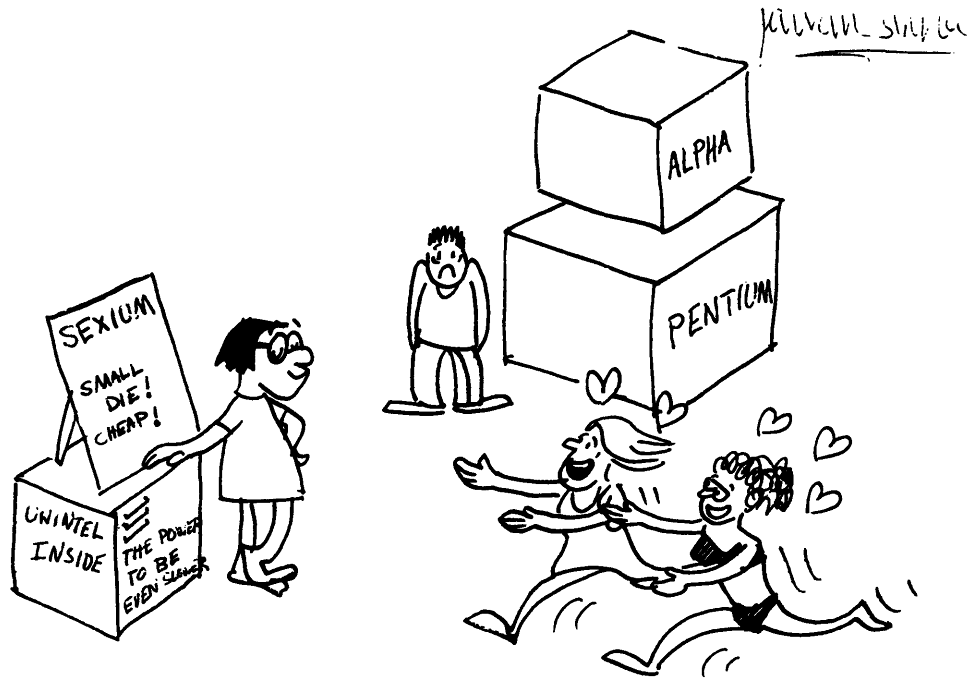
MIT has been seeking to improve the quality of teaching in recent years. These results should remind teachers of some of the proven techniques they have at their disposal.

3.6.3 Student Evaluations

On the last day of class, students were requested to evaluate the class in excruciating detail. The evaluation forms appear in Appendix B; they are summarized below:

Students reported between 20 and 30 hours of work per week for the class, increasing toward the end of the month as the project kicked in. Problem sets were considered long but generally worthwhile and necessary to master the material. The lectures got a mix of comments; many complained that they were too fast, while some felt lectures were too slow. This can be attributed to the wide range of student backgrounds. The microprocessor project was a universal favorite part of the class and was very motivational to many students. Design problems with a chance for optimization and extra help sessions outside of class to review layout were also considered valuable. In the future, students would like to see the chip actually functioning (perhaps through simulation) and would like to spread the class over a longer period of time to reduce the workload. Although one freshman felt overwhelmed by the pace and uncomfortable with his mastery of the material by the end of the term, the others reported fair to thorough mastery and interest in follow-on activities.

4. Microprocessor Design



This section describes the design of the Unintel Sexium microprocessor. Much of the material is taken from the design document developed for the class that appears in Appendix A; however, the design document tends to take more of a tutorial approach.

4.1 Overview

The project selected for the class had to meet several design constraints:

- Exciting and motivational for students
- Capable of being partitioned into work for ten individuals
- Small enough to fit on 4 $M\lambda^2$ MOSIS TinyChip
- To be designed in one month by David Harris
- To be implemented in one and a half weeks by freshmen and sophomores

The most exciting project brainstormed that satisfied the remaining design criteria was an eight bit microprocessor, dubbed the Unintel Sexium. The Sexium was an accumulator machine with a single bus. The registers and ALU fit belong in a single datapath; the microcoded control, implemented with a PLA and state counter, sat above the ALU. The

instruction set was very simple. The program counter and memory address registers were 16 bits each, giving the machine a 64K address space.

The Unintel Sexium is described in detail in the remaining sections of this chapter.

4.2 Instruction Set Architecture

The Sexium instruction set was chosen to provide a minimal Turing-universal set of operations, plus a small number of additional instructions required for efficiently implementing certain constructs (e.g. walking through an array or calling a subroutine).

Visible to the programmer were the accumulator, four general purpose registers (R0-R3), and a 16 bit program counter (PCH : PCL) and memory address register (MAH : MAL).

A table with the complete instruction set appears below:

Instruction	Effect	Comments
Arithmetic / Logical		
`ADD reg`	$A \leftarrow A + reg$	**Add** A to register
`AND reg`	$A \leftarrow A \,\&\, reg$	Bitwise **AND** of A and reg
`NOT`	$A \leftarrow \neg A$	Bitwise complement (**NOT**)
`SHR`	$A \leftarrow A >> 1, A_7 = 0$	**Shift right**
`ROR`	$A \leftarrow A >> 1, A_7 = A_{0\text{-old}}$	**Roll right**
`PUT reg`	$reg \leftarrow A$	**Put** A into register
`GET reg`	$A \leftarrow reg$	**Get** A from register
`TST reg`	$A_0 \leftarrow$ carry (A+reg) $A_1 \leftarrow$ zero (A+reg)	**Test** A + register and set bits of A accordingly
Memory		
`LDA`	$A \leftarrow$ Mem[MA]	**Load A** from memory
`LDI`	$A \leftarrow$ Mem[MA] $MA \leftarrow MA+1$	**Load** A from memory and **i**ncrement MA
`LDM const`	$A \leftarrow$ const	**Load A immediately**
`STA`	Mem[MA] $\leftarrow A$	**St**ore **A** to memory
`STI`	Mem[MA] $\leftarrow A$ $MA \leftarrow MA+1$	**St**ore A to memory and **i**ncrement MA
Control		
`JMP high low`	$PC \leftarrow$ high:low	**Jump** to absolute address
`BRA const`	$PC \leftarrow PC +$ signed const	**Branch** to relative address
`CAL high low`	$R1{:}R0 \leftarrow PCH{:}PCL+3$ $PC \leftarrow PC +$ const	**Call** subroutine and save return address
`RTN`	$PCH{:}PCL \leftarrow R1{:}R0$	**Return** from subroutine
`SKZ`	If $(A = 0)$ $PC \leftarrow PC + 2$	**Skip** ahead 2 if A is **z**ero

Note that SKZ is the only conditional instruction. Instructions are either 1, 2, or 3 bytes long. The lower 5 bits of the first byte specify the operation; the upper 3 bits may name one of the eight registers. Some instructions take an additional one or two bytes from the instruction stream. A table of instruction codings appears below.

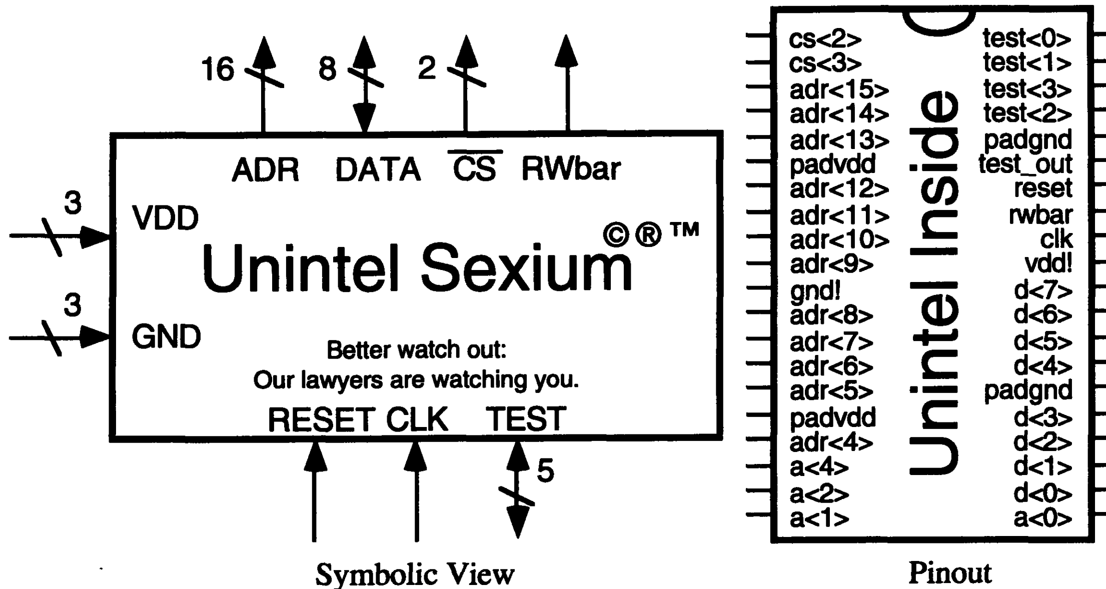
Instruction	Byte 1	Byte 2	Byte 3
ADD	r e g 0 1 0 0 0		
AND	r e g 0 1 0 0 1		
NOT	0 0 0 0 1 1 0 0		
SHR	0 0 0 0 1 1 0 1		
ROR	0 0 0 0 1 1 1 0		
PUT	r e g 0 1 0 1 1		
GET	r e g 0 1 1 1 1		
TST	r e g 0 1 0 1 0		
LDA	0 0 0 0 0 0 0 0		
LDI	0 0 0 0 0 0 0 1		
LDM	0 0 0 0 0 1 0 0	c o n s t	
STA	0 0 0 0 0 0 1 0		
STI	0 0 0 0 0 0 1 1		
JMP	0 0 0 1 0 0 0 0	h i g h	l o w
BRA	0 0 0 1 0 0 1 0	c o n s t	
CAL	0 0 0 1 0 0 0 1	h i g h	l o w
RTN	0 0 0 1 0 1 0 0		
SKZ	0 0 0 1 0 0 1 1	c o n s t	

Register Codes

Code	Register
000:	R0
001:	R1
010:	R2
011:	R3
100:	PCL
101:	PCH
110:	MAL
111:	MAH

4.3 External Interface

The Sexium is packaged in a 40 pin ceramic DIP from MOSTS. In the standard MOSIS pad frame, four pins are dedicated to power and ground for the pad frame; an additional two are used for power and ground to the internal logic, leaving 34 pins for general purpose I/O. The Sexium pinout is shown below:



Symbolic View

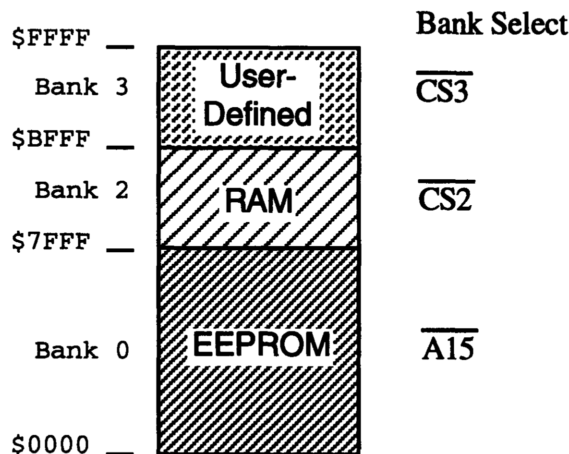
Pinout

RESET and CLK are fed into the chip; CLK goes directly in without passing through any buffering stages, allowing better shaping of the clock from off-chip. TEST has four select signals and one output; it allows the user to probe internal signals by muxing one of sixteen signals, according to the table below: The ring oscillator is a seven-inverter oscillator. Unfortunately, based on the tau-model developed in section 4.9.2, the oscillator is predicted to have a period of 3.5 ns, well beyond the bandwidth of the output pads. The T flip-flop is toggled on the rising edge of the clock.

0000:	resetbar	0001:	ring osc.	0010:	T flop	0011:	F bit
0100:	s<0>	0101:	bus<0>	0110:	bus<1>	0111:	bus<2>
1000:	bus<3>	1001:	bus<4>	1010:	bus<5>	1011:	bus<6>
1100:	bus<7>	1101:	ysel	1110:	ir<0>	1111:	neg

RWbar is high for read operations and is pulsed low for writes to control external memory. The DATA pins are an 8 bit bidirectional data bus; the ADR pins are a 16 bit address bus. The CS pins decode the top bits of the address, dividing memory into banks

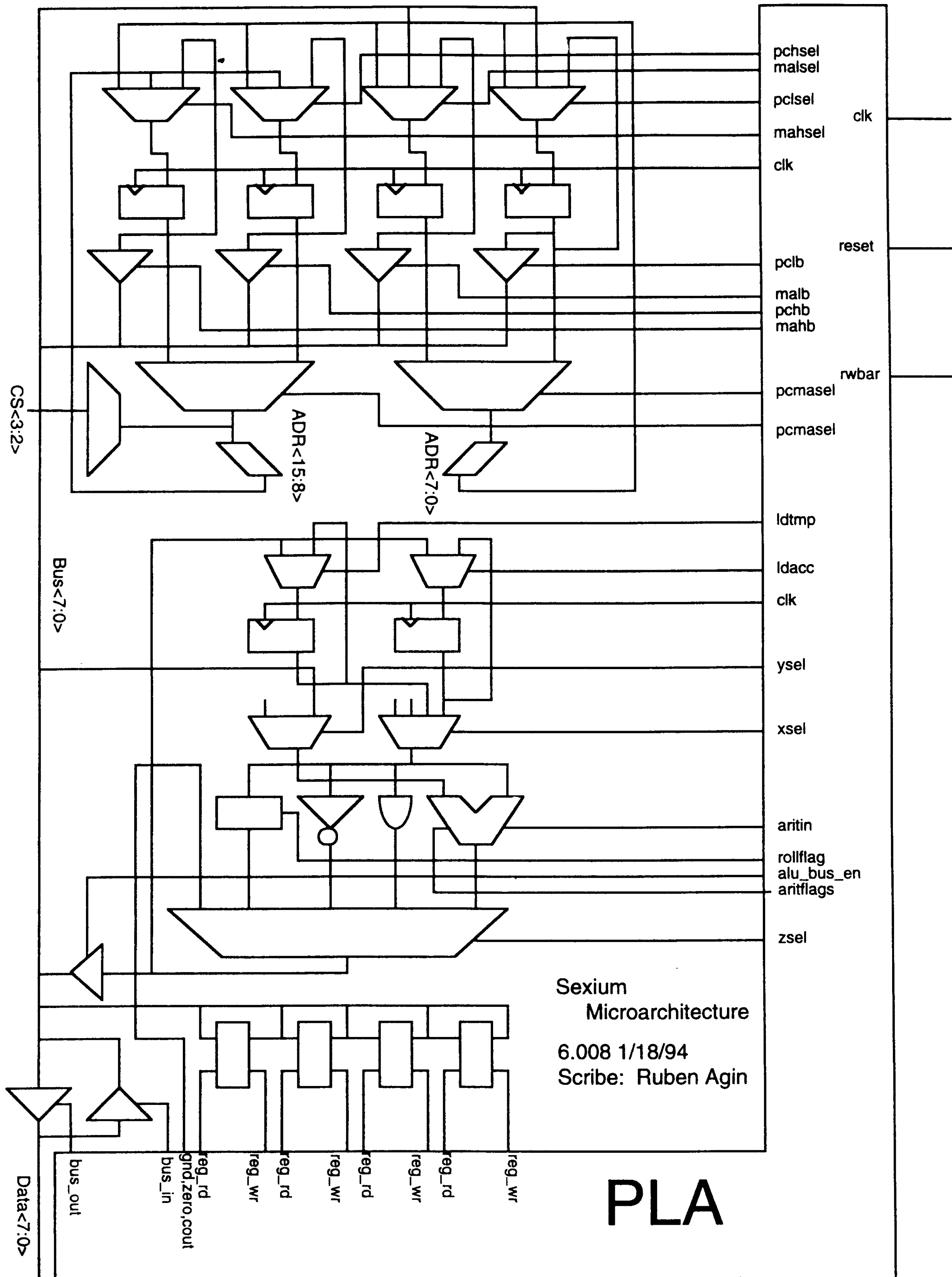
and allowing a computer to be built with no external glue logic between the CPU and memory. The two CS pins and the most significant address pin, A15, decode the address space as follows:



4.4 Microarchitecture

In order to pack the entire microprocessor onto a TinyChip, close interplay between the microarchitecture and floorplan was unavoidable. The chip was partitioned into a datapath and a control unit. The 8-bit datapath was built around a single bus; it was in turn partitioned into the alubox, the regbox, and the pcmabox (program counter and memory address). In addition, the 8 bit instruction register (ir) and the bidirectional data bus drivers were located in the datapath. The control module sat on top of the datapath. It produced control signals that enabled the various modules of the path. A diagram of the datapath, drawn by Ruben Agin when he was scribe, appears below:

One important piece of state in the microarchitecture not visible to the programmer is the TMP registers. Located in the alubox, it is used to store temporary results while performing BRA, CAL, and JMP instructions.



pchsel
malsel
pclsel
mahsel
clk
clk
reset
pclb
malb
pchb
mahb
rwbar
pcmasel
pcmasel
CS<3:2>
ADR<15:8>
ADR<7:0>
ldtmp
ldacc
clk
Bus<7:0>
ysel
xsel
aritin
rollflag
alu_bus_en
aritflags
zsel
Sexium
Microarchitecture
6.008 1/18/94
Scribe: Ruben Agin
Data<7:0>
bus_out
bus_in
gnd,zero,cout
reg_rd
reg_wr
reg_rd
reg_wr
reg_rd
reg_wr
reg_rd
reg_wr
PLA

The control unit interprets instructions using simple microcoded sequences. It is implemented as a Finite State Machine with several flip-flops, a state counter, and a large PLA containing random logic. The FSM has fourteen bits of state:

`ir<7:0>`	instruction register, holds current opcode
`s<2:0>`	state counter, indicates current step in multi-step instructions
`f`	fetch, indicates fetching next instruction into ir
`carry`	carry bit from ALU, used in BRA command to add 16 bit numbers
`neg`	negative bit from ALU, used in BRA command for sign extension

In addition to the state, the control FSM takes five other inputs:

`reset`	reset processor, clear PC to 0 and fetch first instruction
`zero`	zero out from ALU (used for SKZ)
`cout`	carry out from ALU (used for BRA)
`negative`	MSB of Z output is set, indicating negative 2's complement value
`regopin`	intermediate signal fed back through PLA indicating register op

Note that feeding regopin back through the PLA significantly reduces the area of the PLA at the expense of doubling the delay through the PLA for outputs triggered by regopin.

The FSM produces 43 outputs, summarized below:

ALU Control

`xsel<3:0>`	controls the xmux selecting the input to the ALU 0001 = A 0010 = TMP 0100 = $00 1000 = $FF
`ysel`	controls the ymux selecting the input to the ALU 0 = bus 1 = $00
`zsel<4:0>`	controls the zmux selecting the output of the ALU 00001 = adder 00010 = ander 00100 = neger 01000 = shifter 10000 = flags
`alu_bus_en`	enables tristate of Z output to bus
`rollflag`	controls bit 7 of shifter output 0 = shift (load 0 into bit 7) 1 = roll (load old bit 0 into bit 7)
`ldacc`	loads the A register from the Z output
`ldtmp`	loads the TMP register from the Z output
`cin`	sets cin on adder

REG Control

`reg_wr<3:0>`	write general purpose register from bus
`reg_rd<3:0>`	read general purpose register onto bus

PCMA Control

`pcma_rd<3:0>`	read PC / MA onto bus
`pcma_in<7:0>`	write PC / MA from bus or incrementor (four 2-bit codes) 00 = keep old value 01 = bus 10 = incrementor
`pcma_select`	choose PC or MA to drive address pins 0 = PC 1 = MA

I/O Control

`bus_out`	drive bus onto data pins
`rwbar`	read or write external memory 1 = read 0 = write (pulsed low during second half of cycle)

FSM Control

`clrs`	clear S counter
`newf`	set f bit
`latchcarry`	latch CARRY bit
`latchneg`	latch NEG bit
`regop`	fed back straight to regopin

The following microcode is used to implement each of the instructions. All operations listed on one line occur in parallel during one clock and results are not visible until the next rising clock edge.

Reset (RESET = 1)

S000: PC ← 0; S ← 0; F ← 1;

Fetch (F = 1)

S000: IR ← M[PC]; S ← 0; F ← 0; PC ← PC + 1

LDA / LDI

S000: A ← M[MA]; (if inc MA ← MA + 1); S ← 0; F ← 1;

STA / STI

S000: M[MA] ← A; (if inc MA ← MA + 1); S ← 0; F ← 1;

LDM

S000: A ← M[PC]; PC ← PC + 1; S ← 0; F ← 1;

2-Op (e.g. ADD, AND, TST, GET) or 1-Op (e.g. NOT, SHR, ROR)

S000: A ← A op REG; S ← 0; F ← 1;

PUT

S000: REG ← A; S ← 0; F ← 1;

JMP

S000: TMP ← M[PC]; PC ← PC + 1;

S001: PCL ← M[PC];

S010: PCH ← TMP; S ← 0; F ← 1;

BRA

S000: TMP ← M[PC];
S001: TMP ← PCL + TMP;
S010: PCL ← TMP;
S011: TMP ← PCH + CARRY - NEG
S100: PCH ← TMP;

CAL

S000: TMP ← M[PC]; PC ← PC + 1;
S001: R1 ←M[PC]; PC ←PC + 1;
S010: R0 ← PCL;
S011: PCL ← R1;
S100: R1 ← PCH;
S101: PCH ← TMP; S ← 0; F ← 1;

RET

S000: PCL ← R0
S001: PCH ← R1; S ← 0; F ← 1;

SKZ

S000: (if A = 0 PC ← PC +1)
S001: PC ← PC + 1; s ← 0; (if A = 0) then F ← 1;
else PC ← PC + 1; F ← 1;

The logic equations for the PLA are listed in the Appendix B.

4.5 Verilog Model

Once an assembler and interpreter had been developed to test the ISA (see section 5.1), David wrote a Verilog model of the Sexium to flush out the microarchitecture. This model identified all of the datapath cells required. Care was taken to exploit regularity and modularity to minimize the number of unique cells that students had to design, layout, and especially test. It also contained equations implementing the sequencer and microcode of the control logic.

The Verilog model was capable of loading memory with a program assembled by msim and executing it, showing the contents of the registers and busses over time. Running the regression suite allowed David to debug the microcode and prove the microarchitecture was complete.

The complete Verilog model appears in Appendix C.

4.6 Schematics

Once the Verilog model was complete, it was a simple matter to create schematics. At first, only the top level schematics were drawn in Cadence; the mid-level modules such as the alubox were left as functional views containing the Verilog modules. Once the chip simulated correctly at that level, the functional views were replaced by schematics going all the way down to the transistor level. The general hierarchy of the chip is shown below:

Level	Example Cell Name
Chip with external memory	computer
Chip	sexium
Major module	alubox
8 bit datapath cell	adder
1 bit datapath slice	adderbit

As the design progressed and some modularity had to be sacrificed for area efficiency, several of the schematic cells were changed from the original Verilog model. For example, the model contained a 3 input multiplexor connected to a D flop-flop; the schematics eventually contained a single cell, flop3, with the multiplexor and flip-flop integrated tightly together.

All of the schematics appear in section 4.8 along with the cell descriptions.

4.7 Floorplan & Layout

This section describes the physical implementation of the Sexium microprocessor. Appendix C contains plots of the layout of the chip and each of the major modules.

4.7.1 Layout Design Rules

In order for the large group of students to rapidly produce a chip that would fit together correctly, carefully defined layout design rules were essential. The physical design rules were simple: layout had to obey the MOSIS design rules. In addition, there were special rules specific to the project:

Standard Cell Design Rules

- Power and ground run horizontally in metal 1
- Inputs, outputs, and control run vertically in poly or metal 2
- All cells are 60 λ tall, measured from center of ground to center of power busses
- Every cell has substrate contact, every well has well plug
- Top half of cell reserved for PMOS devices, bottom for NMOS
- Pins and labels on all inputs and outputs

Datapath Design Rules

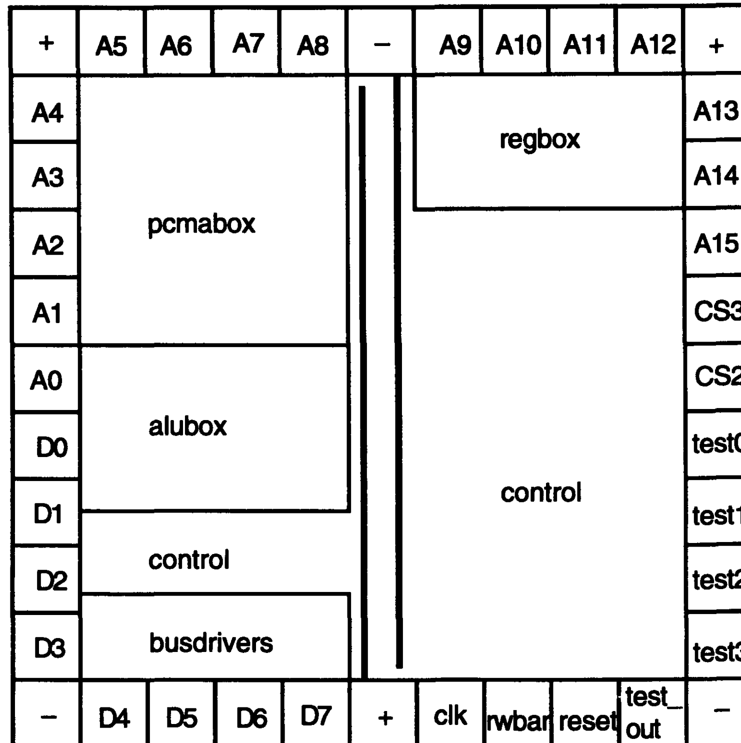
- Bitlines run horizontally in metal 2, constrained to lie in ten horizontal tracks
- Power, ground, and control run vertically in metal 1
- All cells are 80 λ tall
- Every cell has substrate contact, every well has well plug
- Pins and labels on all inputs and outputs

Circuit Design Rules

- Use fully restoring logic except in specially approved cases
  (e.g. single-transistor muxes, pseudo-NMOS PLAs)
- Generate clkbar locally to avoid skew problems
- Use static logic

4.7.2 Floorplan

A rough floorplan of the chip appears on the next page:



Floorplan: 1 inch = 600 λ

The datapath cells, except the regbox that did not fit, were lined up on one side of the chip, while control occupied the other side. A row of standard cells above the datapath generated some control signals (e.g. clkbar); the power and ground busses of the cells were widened to 20 λ each to supply power to the entire chip. The PLA, state counter, test multiplexors, and logo were jammed into the control section as they fit best.

4.7.3 Datapath Connectivity Diagrams

One of the challenges of the datapath design was to position bitlines of datapath cells in such a way that routing channels between cells to connect inputs and outputs were minimized. The problem was further complicated by the fact that several cells were used in multiple modules, so any metal 2 interconnect running over a cell at any of the higher level modules had to be placed such that it did not interfere with the wiring in the cell. This was most serious when the design was modular and one multiplexor was used in many places; as modularity was sacrificed to reduce area, planning the wiring also became simpler.

To manage this complexity, David drafted "datapath connectivity diagrams" illustrating the placement of the datapath cells and the usage of the metal tracks. The diagrams were updated on a regular basis as students found tricks that would eliminate more and more unnecessary routing channels. The connectivity diagrams for the various datapath modules appear with the cells in section 4.8.

4.8 Cell Descriptions

For each cell in the design, this section offers a brief description, area and transistor count information, and a datapath connectivity diagram where appropriate. Schematic and layout views of each cell are located in Appendix C. Also, Appendix C contains a higher-level Verilog model of the entire microprocessor.

4.8.1 Top Level

computer

Instantiates instances of Sexium processor and memory, used to simulate complete system.

sexium

Contains the entire chip: datapath, control, and pads.

Area:	$2220\ \lambda \times 2250\ \lambda = 5.0 \bullet 10^6\ \lambda^2$
Transistors:	5319
Area / Transistor:	$940\ \lambda^2$

The transistor count is broken up as follows:

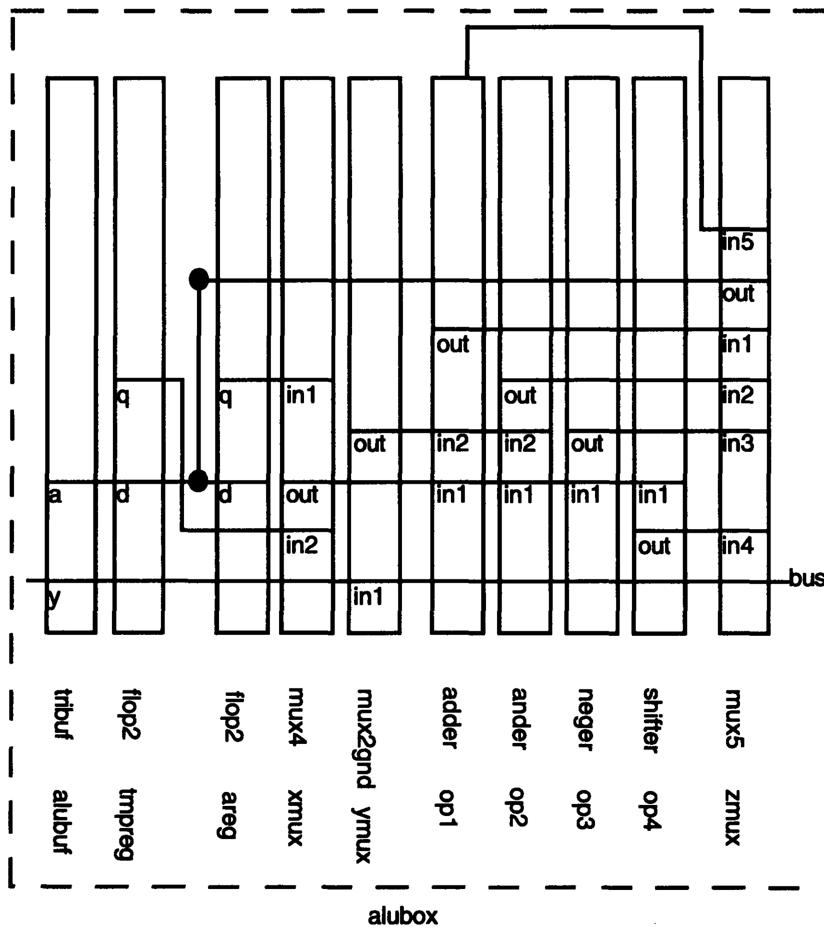
Module	Transistors	Area (λ^2)	Area/Transistor (λ^2)
alubox	844	$3.0 \bullet 10^5$	356
regbox	1074	$5.2 \bullet 10^5$	484
pcmabox	1176	$6.3 \bullet 10^5$	536
control	1233	distributed	
pads	922	distributed	

4.8.2 alubox

Most Sexium operations move data through the alubox. Two multiplexors select the X and Y inputs to the ALU; a third chooses the output of one of the functional units to be the Z output. The module contains ten cells: two flip-flops with the A and TMP registers, three multiplexors to chose X, Y, and Z, an adder, ander, neger, and shifter, and

a tristate buffer for driving Z onto the bus. The adder, ander, and neger functions could have been combined into a single cell at the expense of a potentially less efficient adder design.

Area:	$524\,\lambda \times 757\,\lambda = 3.0 \cdot 10^5\,\lambda^2$
Transistors:	844
Area / Transistor:	$356\,\lambda^2$

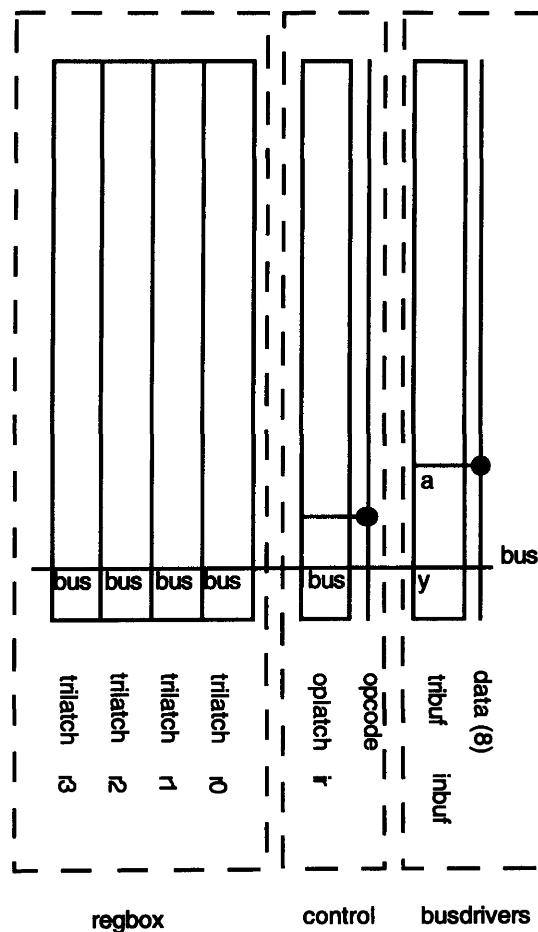


Cell	Width	Copies	Notes
flop2	80	2	mux2 integrated into flip-flop to save area
tribuf	37	1	8-bit tri-state bus driver
mux2gnd	25	1	unrestoring, single NFET mux
mux4	55	1	unrestoring, single NFET mux
mux5	68	1	restoring mux
adder	100	1	8-bit adder
ander	29	1	8-bit and function
neger	16	1	8-bit not function
shifter	wire only	1	8-bit shift or roll
Total:	524		including wiring tracks

4.8.3 regbox

The regbox is the simplest of the datapath modules. It contains four tristate latches, R0-R3, which can latch the bus or drive their contents onto the bus. While instruction register and busdrivers are technically not part of the regbox, they are described in this section. The busdriver cell drives input from the pads onto the data bus; a separate tri-state driver built into the pads drives the bus out to the data pins.

Area:	$\sim 687\ \lambda \times 757\ \lambda = 5.2 \cdot 10^5\ \lambda^2$
Transistors:	1074
Area / Transistor:	$484\ \lambda^2$

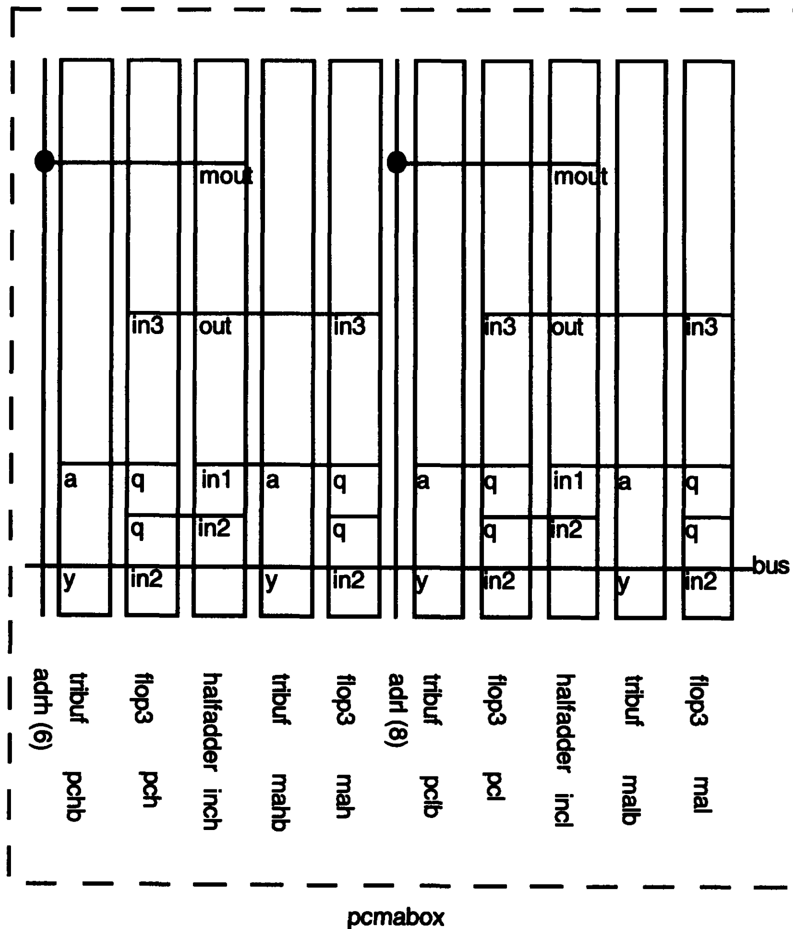


Cell	Width	Copies	Notes
trilatch	69	4	transparent latch with tri-state output
oplatch	119	1	like trilatch, but always outputs to control
tribuf	69	1	8-bit tri-state bus driver
Total:	687		includes wide busses

4.8.4 pcmabox

The pcmabox contains the four pcma registers (PCH, PCL, MAH, MAL), along with a 16 bit incrementor and a multiplexor to drive either PC or MA onto the address bus. In addition, each register has a tristate buffer to drive its contents onto the bus. The 16 bit registers are partitioned into 8-bit low and high halves. Each half shares an 8 bit incrementor between the PC and MA registers and total area is minimized by exploiting locality.

Area:	$832\lambda \times 757\lambda = 6.3 \bullet 10^5 \lambda^2$
Transistors:	1176
Area / Transistor:	$536\lambda^2$



pcmabox

Cell	Width	Copies	Notes
flop3	94	4	flip-flop with 3-input mux
tribuf	37	4	8-bit tri-state bus driver
halfadder	76	2	half-adder with 2-input mux
Total:	832		includes wide busses

4.8.5 control

The control unit contains the microcode sequencer, a bit of random logic used to factor the microcode, and the test multiplexors that bring test signals out to the pads. The microcode sequencer is implemented with a large PLA, flip-flops, and a 3 bit state counter to maintain the current state. Due to its irregular shape, area is not tabulated.

Transistors: 1233

control_pla

The control PLA was created with the PLA generator. It is a pseudo-NMOS NOR/NOR design. The logic equations used to generate control_pla are located in appendix C.

Area: $801\,\lambda \times 775\,\lambda = 6.2 \cdot 10^5\,\lambda^2$
Transistors: 829
Area / Transistor: $747\,\lambda^2$

testc

This testc block is a set of standard cells that implements the 16 input test multiplexor and a bit of test logic to help debug the chip.

Area: $422\,\lambda \times 259\,\lambda = 1.1 \cdot 10^5\,\lambda^2$
Transistors: 138
Area / Transistor: $792\,\lambda^2$

counter

The counter is a 3 bit counter built from standard cells that sequences through the microcode when interpreting multiple-step instructions

Area: $403\,\lambda \times 111\,\lambda = 4.5 \cdot 10^4\,\lambda^2$
Transistors: 90
Area / Transistor: $497\,\lambda^2$

4.8.6 Pads

The pads from MOSIS were studied and a schematic model was created so that the pads could be included in the chip-level simulation and checked with LVS. This should help eliminate the common silly mistakes of incorrectly connecting signals to the pads.

frame_v4

The MOSIS pad frame contains 40 pads. The four corner pads are dedicated to VDD or GND to power the output drivers on the frame itself. An additional pair of pads provide VDD and GND to the chip interior. The remaining 34 pads are I/O pads, described below.

v4io

The I/O pads supplied by MOSIS feature a tri-state output driver and unbuffered, inverted, and buffered versions of the input. The enable pin, when high, turns on the tri-state driver and makes the pad function as an output pad. ESD protection is done with 600/3 field oxide transistors, a 150 ohm ndiff resistor / diode, and tri-state output drivers acting as a pair of diode clamps. Two power and two ground pins provide isolated power to the pad frame; an nwell isolates substrate currents of the NMOS output drivers from the substrate of the chip interior. According to MOSIS, each pad can sink or source 11mA of current and can drive a 50 pF load.

Section 4.9.4 includes data from HSPICE simulations of pad rise and fall times.

4.9 Timing

4.9.1 Signal Timing

The Sexium uses a simple timing methodology for ease of design, interfacing, and testing. Signals are divided into two classes: edge-triggered and level-sensitive.

The vast majority of signals are edge-triggered. On the rising edge of the clock, new values are loaded into the control state and PC / MA registers. The PLA evaluates and datapath performs its operations in a purely combinational fashion henceforth with two exceptions.

The two level-sensitive exceptions are R0-R3 and memory writes. When the clock falls low, the level-sensitive registers open. The registers were implemented in this way because it saves area while remaining race-free. Memory writes also occur when the clock is low. The data to write is driven onto the data bus and the rwbar output falls low. At the rising edge of the clock, rwbar returns high. Since the two gates driving rwbar are

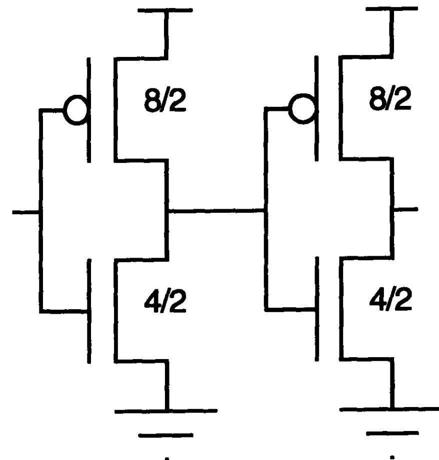
much faster than the evaluation time for the PLA, the write shuts off before the data bus could change and corrupt memory.

4.9.2 Tau Model

The Tau Model is a simple method of estimating delays through CMOS circuits. It models the RC delay of transistors driving a gate load capacitance, source-drain parasitics, and any additional wiring parasitics. In this section, the Tau Model is described; in the subsequent section, the model is used to predict critical paths through the microprocessor.

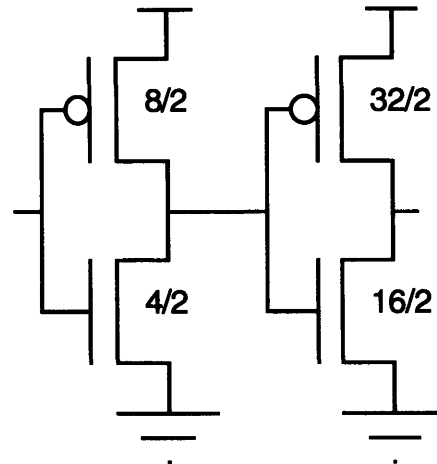
In the Tau Model, τ_c is defined as the RC product of a unit-sized (4 λ / 2 λ) NMOS transistor loaded with a unit-sized gate capacitance. PMOS transistors are normally assumed to have twice the effective resistance of NMOS devices. Source or drain capacitance and capacitance of local interconnect is assumed to be approximately equal to one gate capacitance. Consider an inverter with a unit-size NMOS device and a double-sized PMOS device driving an identical inverter. The effective resistance is 1; the effective capacitance is 6:

NMOS load gate	1
PMOS load gate	2
NMOS driver drain	1
PMOS driver drain	2



Thus the product is $6\tau_c$. This is approximately the propagation delay through the first inverter. Similarly, an inverter driving a 4x inverter has a delay of $15\tau_c$: the resistance is one and the capacitance is 15.

NMOS load gate	4
PMOS load gate	8
NMOS driver drain	1
PMOS driver drain	2



In the mosis2n 2 μm N-well process that we are using, we can compute τ_c and the gate capacitance C_g as follows[2]:

$$C_g = \frac{\varepsilon_{ox} W \bullet L}{t_{ox}} = \frac{3.45 \bullet 10^{-13}\,\text{F/cm} \bullet 4 \bullet 10^{-4}\,\text{cm} \bullet 2 \bullet 10^{-4}\,\text{cm}}{400 \bullet 10^{-8}\,\text{cm}} = 6.9\,\text{fF}$$

$$\tau_c = \frac{2\,V_{DD}^2}{(V_{DD} - V_T)^2} \bullet \frac{L^2\,\varepsilon_{ox}}{2\,K_p\,t_{ox}\,V_{DS}} = \frac{25}{16} \bullet \frac{(2 \bullet 10^{-4}\,\text{cm})^2 \bullet 3.45 \bullet 10^{-13}\,\text{F/cm}}{26.3 \bullet 10^{-6}\,\mu\text{A/V}^2 \bullet 400 \bullet 10^{-8}\,\text{cm} \bullet 5} = 41\,\text{ps}^3$$

HPSICE simulation of a unit sized inverter driving a 4x inverter gives a propagation delay of 560 ps falling and 620 ps rising, indicating an actual average τ_c of 39 ps, in very close agreement with the model!

According to MOSIS process parameters, approximately a minimum width strip of metal approximately 80 λ long or a strip of poly approximately 100 λ long adds C_g of capacitance. Similarly, a 200 λ strip of poly adds one unit of resistance (about 8 KΩ). PMOS devices have roughly 1/3 the drive strength of NMOS devices.

4.9.3 Critical Paths

Although speed was not a primary objective for the Sexium design, the probable critical paths were briefly considered. Three paths are described below.

[2]Derivation in 9/27/93 lecture notes of 6.845, Concurrent VLSI Architecture, Professor Dally, MIT.
[3]Transistor parameters taken from 1992 mosis2n run.

The add instruction is the longest strictly combinational critical path. A signal must propagate through the PLA twice to decode the register read. The register must be driven onto the bus and trickle through an 8-stage ripple-carry adder before finally being latched into the accumulator. Simply propagating through the PLA twice requires over 1000 τ_c (the most significant part of this delay is driving the input lines, which requires twice as long as optimal because of the significant delay charging up the input line before the input inverter switches and begins discharging the complement of the input line; the IR input lines are especially slow due to the distributed RC delay of an 800 λ line driving 51 loads). This corresponds to more than 40 ns merely decoding the instruction and suggests maximum operating speed would be around 10 MHz if a similar amount of time is required for instruction execution.

Module	**Cell**	**Path**
control	ir	clk → Q
control	control_pla	IR → add product
control	control_pla	add product → regop out
control	control_pla	regop → r0rd product
control	control_pla	r0rd → reg_rd[0]
regbox	r0	reg_rd[0] → bus
alubox	ymux	bus → y
alubox	adder	y → sum
alubox	zmux	sumt → q
alubox	amux	in2 → out
alubox	areg	d setup time

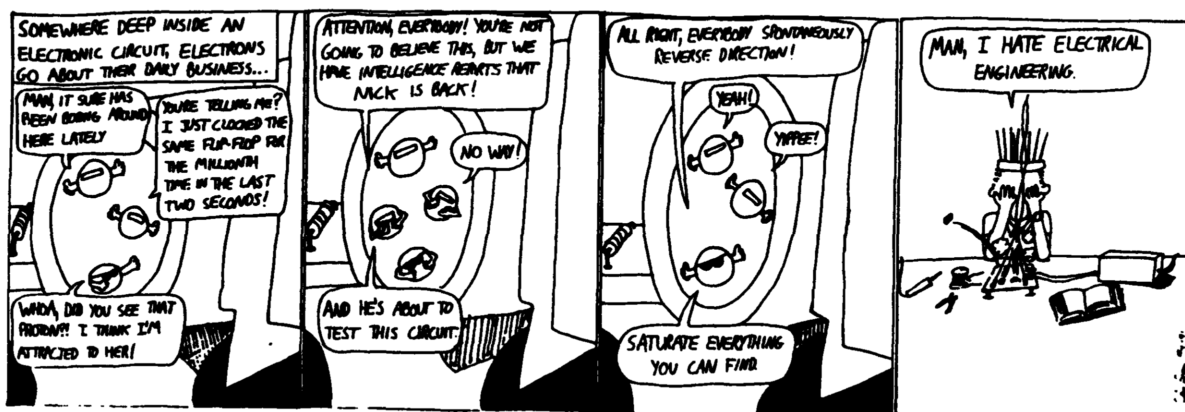
The STA instruction is a second critical path. During the first half of the cycle, the instruction is decoded and read onto the bus. During the second half of the cycle, the bus is driven out onto the data pads and written to memory. Hence, the decode and bus drive time must be less than half of the clock period.

Module	**Cell**	**Path**	
control	ir	clk → Q	
control	control_pla	IR → sta product	
control	control_pla	sta product → xsel	
alubox	xmux	xsel → x	
alubox	adder	x → sum	
alubox	zmux	sum → z	
alubox	alubuf	z → bus	
			(wait for clk low)
control	nand	clk → rwbar	
control	inv	rwbar → wrbar	
padframe	v4io	enable → out	
external ram		write time	

Memory read instructions, including FETCH cycles, trigger the third critical path. In this path, the instruction must be decoded. The appropriate address is then driven out through the pad frame; MOSIS reports approximately 13 ns delay through the output pad driving a modest load. The memory must respond by driving its value back onto the data bus and into the Sexium. In most cases, this path is expected to limit the system clock because a typical EPROM containing boot code will require about 120 ns to access. When decoding and pad propagation delays are considered, this sets a practical system speed of about 5 MHz. This critical path can be eliminated with faster memory, at a considerably higher total system cost.

Module	Cell	Path
control	ir	clk → Q
control	control_pla	IR → lda product
control	control_pla	lda product → pcmasel
pcma	muxl	pcmasel → a
padframe	v4io	a → adr
external memory		adr → data
padframe	v4io	data → d
busdrivers	inbuf	d → bus
alubox	ymux	bus → y
alubox	adder	y → sum
alubox	zmux	sum → z
alubox	alubuf	z → bus
alubox	ymux	bus → y
alubox	adder	y → sum
alubox	zmux	sumt → q
alubox	amux	in2 → out
alubox	areg	d setup time

4.A Verification



4.A.1 Schematic Simulation

The first part of verification was simulation at the schematic level to ensure a correct design. The design was netlisted to Verilog form, then simulated with Verilog XL-grx. The first simulation used functional descriptions of each of the major blocks; later simulations eliminated functional views and simulated all of the circuitry except the PLA at the switch level. The PLA layout is generated directly; hence, no schematic view is available and a behavioral view output by the PLA generator must be used instead.

4.A.2 Layout vs. Schematics

The Layout vs. Schematic (LVS) tool was immensely valuable for verification. Students learned the tool and checked their work against the schematic to catch most layout errors. Cadence's LVS tool produces very cryptic error messages and is difficult to use, but with practice students mastered it and the more advanced students were able to run LVS on modules higher in the hierarchy, verifying the complete chip. Dan Hartman was responsible for most of the final verification and debugging.

4.A.3 Layout Simulation

Critical circuits had to be verified from layout using the analog capabilities of HSPICE. In particular, the PLA and I/O pads were simulated.

On the PLA, it was found that a single on NFET in the NOR / NOR array was sufficient to pull an output down to 0.5 volts, below the threshold of any subsequent N devices. Simulation showed that each pullup, when on, drew approximately 300 μA. With a total of 43 outputs and 56 product terms, the PLA will draw a maximum of 30 mA. Fortunately, most of the outputs are off most of the time; during a typical instruction, such as ADD R0, simulation shows that the PLA draws 16 mA of current. Power rails were drawn 20 λ wide to accommodate such current levels.

A crude simulation neglecting the significant wire capacitances and polysilicon resistances suggested a rise-time for worst-case signals that must propagate through the PLA twice to be about 5 ns and a fall time of 7 ns. With actual parasitics, the propagation delays are predicted to increase to as long as 40 ns (primarily due to distributed polysilicon resistance). See the plots of the PLA timing, voltage swings, and current drain in Appendix C.

The I/O pads simulated correctly in HSPICE. Switching waveforms are also available in Appendix C.

4.A.4 Final Verification

The final verification proceeded as follows:

- Flat-file DRC of entire chip
  Automatic joining of VDD! and GND! nets was turned off to check connectivity
- Hierarchical extraction of entire chip
- LVS of entire chip
- Verilog simulation of chip at schematic level
- Output of chip to CIF format
- Run fixcif on CIF output
  The Cadence CIF output is buggy; fixcif adds some missing statements
- Read CIF into Magic
  The Magic layout editor has a known good DRC function
- Run DRC on CIF using Magic

The flat-file DRC reports one error: a short between gnd! and padgnd. This is because the padframe and internal logic both have substrate contacts. While an N-well actually isolates the internal logic from the noisy substrate beneath the pads, DRC believes the entire substrate is one net.

Hierarchical extraction and LVS report no errors. The Verilog simulation shows that multiple signals drive the external data bus for a short period during write cycles; this is acceptable and is ignored. Magic no DRC errors using a set of design rules that have been used on working chips.

When verification was complete (February 24, 1994), the chip was submitted to MOSIS.

5. Tool Development

Developing the proper set of tools was critical for the success of this class and design of the microprocessor. This chapter describes the various tools, both for simulating and validating the instruction set and for building the chip in Cadence.

5.1 Sexium Architecture Tools

5.1.1 msim

msim is an assembler and interpreter for the Sexium ISA. It accepts a file of Sexium assembly language instructions and assembles them, producing a binary file and outputting versions that can be read into the Verilog simulator and into an EPROM burner. It then enters simulation mode. In this mode, msim allows the user to run, trace, and single-step programs, disassemble, read, and modify memory, print the state of the sexium registers, and set or change breakpoints.

msim is documented in Appendix A in the simulator reference manual. It was written in portable C code and runs on the AI lab Sparcstations, on Athena, and on the Macintosh environment. It is approximately 900 lines long; complete source code is listed in appendix D.

5.1.2 Regression Suite

It was also important to verify that all of the Sexium instructions work correctly. To do this, a regression suite that exercises each of the instructions was developed. The program uses each of the instructions, triggering a break if any instruction performs incorrectly.

The regression suite tests both the msim simulator, the Verilog model, and the Verilog netlist extracted from the chip schematics. Source code appears in appendix D.

5.2 Cadence Tools

5.2.1 mosis2n Library and Technology File

Cadence requires information specific to the fabrication process in order to properly extract HPSICE netlists and display the proper set of layers. This information resides in the transistor library and in the process technology file.

For this class, a library with mosis2n (a 2 micron n-well process fabricated by MOSIS) transistor parameters and a technology file with the proper layers for mosis2n were developed. They were adapted from the HP26 files developed by Larry Dennison and Mike Bolotski.

These libraries should provide especially useful because 6.371 usually works in the mosis2n process and requires tested technology files to migrate to the Cadence tools.

In addition to the standard nmos and pmos transistors, weaknmos and weakpmos devices were defined. These are identical from the layout perspective, but are modeled as resistive devices by the Verilog netlister. These allow flip-flops and other cells such as SRAM with weak feedback to simulate properly.

5.2.2 Standard Cell Library

A second resource developed in this class was a library of seventeen standard cells. The cells are built on a 60 lambda pitch and designed such that any two cells can be abutted without interference. Power and ground are run horizontally in metal 1; inputs and outputs come vertically in poly or metal 2. The cells are hand-tuned for area and use minimum size (4/2) devices, trading away performance for minimal area.

The table below lists the cells, the student(s) responsible for the layout, and the width and area / transistor of each cell. All cells pass LVS and DRC. Note that the mx multiplexors produce inverted outputs, that flop is a static positive edge-triggered D flip-flop, latch is a static positive level-sensitive latch, and aoi3 is a three-input and-or-invert gate.

Cell	Creator	Width	Area / transistor
and2	Dan	30	300
aoi3	Jeff / Dan	29	290
flop	Matt	69	230
inv	Ethan / Dan	11	330
invtri	Dan	19	285
latch	Matt	36	216
mx2	Nehal	40	400
mx3	Bayard	88	660
mx4	Bayard	112	672
nand2	Nehal	18	270
nand3	Ruben	32	320
nand4	Matt	30	225
nor2	Nehal	18	270
nor3	Aarati	24	240
nor4	Matt	30	225
or2	Dan	30	300
xor2	Jeff	36	216

5.2.3 PLA Generator

MIT has had a chronic problem with reliable PLA generation in recent years. For this class, David wrote a simple PLA generator that accepts logic equations in sum of products form and produces compact layout, with a width of 16 λ / input + 12 λ / output and a height of 12 λ / product term (all dimensions neglect the modest sized input and output inverters and the weak pullup devices). The PLA is a NOR / NOR architecture built with psuedo-NMOS pullups for simplicity and area efficiency. Modification to support clocked precharge circuitry would be simple.

The PLA generator is written in SKILL, Cadence's LISP-like scripting language. The first portion is somewhat kludgey, reading the file of Boolean equations and producing an internal representation. The second part processes the internal representation, producing layout. The generator produces both layout and a Verilog behavioral description.

The PLA generator is documented fully in Part 3 of the Cadence Tool Reference (appendix A) and source code appears in Appendix D.

5.2.4 Pad Frame

6.371 also has had difficulty in recent years with pad frames. MOSIS supplies a CIF file with a pad frame for the 2.25 mm square TinyChip; unfortunately, the conversion from

CIF to Mentor Graphics' L language or Cadence's layout database is trickier than it should be.

David imported the TinyChip pad frame, fixed the problems introduced by the conversion, and added pins and schematic views to use for verification. The pad frame is being fabricated with the Sexium project; if it works correctly, it will be a resource available for future classes. The pads are documented in section 4.8.6.

5.2.5 Setting up Cadence for another class

In order to set up files to run Cadence for another class, follow the steps below:

1) Create a class directory
   In the case of this class, the directory was /home/cva2/6090user

2) Place a copy of the technology files in the class directory.
   Files may be copied from /home/cva2/6090user/techfiles. It is also possible to continue using the files in the 6090user directory without making a copy; in such a case, be very careful that any modifications do not disturb other techfile users.

3) Create a library directory
   The library directory will hold the libraries for the students and class project. In the case of this class, the library directory was /home/cva2/6090user/libs

4) Create a project directory
   A special project directory must be located in the Cadence filesystem to define information unique to the project. When Cadence is invoked with the command

   ```
   startCds -p sexium
   ```

   it looks in the directory /cds/local/skill/projects/SEXIUM for the project-startup.il file, which has information unique to each project, including the library search paths, default technology file path, etc. It should be edited to contain proper information for the particular project. When creating the project directory, be sure to copy all files, including the invisible .simrc.

5) Create user accounts
   The last step is to create accounts for each user. The .paths be set correctly to run Cadence from the Concurrent VLSI Architecture Sparcstations.

6. Conclusions

6.1 Educational Results

The Institute's *Better Teaching at MIT* colloquium raised awareness of the need for continually improving undergraduate education. The VLSI Chip Design class was an educational experiment motivated by such a goal. It proved that VLSI design is no longer the sole province of silicon wizards who have endured years of semiconductor physics; now, the exciting and very important field is accessible to freshmen and sophomores with a bare minimum of experience.

Students entered with a variety of backgrounds, but the least common denominator was elementary digital electronics that many students had taken as a freshman seminar. Students left knowing how to do the following:

- Implement Boolean functions using MOSFETs or switches
- Take advantage of compact multiplexor structures to implement functions
- Layout leaf cells in standard cell and datapath design styles
- Rapidly examine layout tradeoffs using stick diagrams
- Combine leaf cells into higher level modules
- Implement flip-flops, counters, and sequential logic in CMOS
- Understand design and use of RAM, ROM, and PLAs
- Understand the architecture of a micro-coded accumulator machine
- Write assembly language code for the Sexium
- Work in a team to rapidly layout a complex chip
- Verify cells using Design Rule Checker and Layout vs. Schematic comparison

Student evaluations, included in Appendix B, were overwhelmingly positive. Nine of the ten students felt that they had a good command of the material. One exceptionally strong student skipped over 6.371 and is now doing well in 6.372; another freshman is doing VLSI design as a UROP in the Concurrent VLSI Architecture group.

6.2 Project Results

6.2.1 Unintel Sexium Summary

The Unintel Sexium was nearly complete by the final day of class. Verification required four subsequent Saturday sessions; the final layout has been submitted to MOSIS and is queued for the next fabrication run.

Statistics about the chip are listed below:

Area:	$2220\,\lambda \times 2250\,\lambda = 5.0 \bullet M\lambda^2$
Transistor count:	5319
Average area / transistor:	$940\,\lambda^2$
Static power dissipation:	~100 mW

Speed was intentionally neglected in order to complete the design in a small amount of time and area. Nevertheless, the longest critical path is expected to be dominated by EPROM access time when 120 ns EPROMs are employed, limiting system speed to approximately 5 MHz.

6.2.2 Tool Summary

The tools developed for the project are summarized in this section. They include the msim assembler / interpreter, the regression suite, the mosis2n libraries, the PLA generator, and the pad frame.

The msim assembler / interpreter and the regression suite are very useful tools for the Sexium project. They will be used again this spring as students continue the project with a software effort (see section 6.4). They are also excellent examples of where a day's effort building tools can greatly enhance confidence in the validity of an architecture.

The Cadence-based tools, including the mosis2n technology information, the PLA generator, and the pad frame, were designed to not only be used in this class, but also to support other classes, such as 6.371, when they shift to the Cadence CAD software. The PLA generator is a remarkably efficient way to implement a large set of arbitrary logic equations, saving both time and area in comparison to standard cells. The pad frame is

necessary on any MOSIS TinyChip produced at MIT; when it is proven reliable, it will eliminate one worry every designer in a class or research group presently faces.

6.3 Summary

In summary, this thesis project had three primary benefits: proof-of-concept of teaching VLSI at an earlier level, education of ten students, and development of the Cadence infrastructure.

The class proved that freshmen and sophomores were capable of mastering elementary VLSI circuit design and layout. It was shown that teaching VLSI was an excellent medium for introducing ideas of managing complex systems as a team, of applying digital electronics to a real system, and of teaching computer design and assembly language programming.

The ten students who took the course learned a variety of subjects at many levels. They learned about transistors, methods of building digital circuits, and layout and verification techniques. They also learned to program in simple assembly language and to understand how computers work from the transistor level all the way up to the software level. They had the experience of building a complicated VLSI system, an experience that should motivate and build confidence in the young students. There have already been requests from the Concurrent VLSI Architecture group for 6.008 students to work as UROPs doing VLSI design; the students' experience should provide many opportunities for exciting UROPs at MIT and summer jobs in industry.

Finally, the technology files, pads, and PLA generator developed in this class should help other MIT classes begin using the Cadence tools. Cadence is significantly more powerful than the old Mentor Graphics tools; MIT will benefit when a significant number of VLSI designers are competent with the new CAD tools.

6.4 Recommendations

Based on the completion of the Sexium microprocessor and the response of the students, "6.008: VLSI Chip Design" was a fabulously exciting and successful educational experiment. This author hopes to see the course impact MIT education over several different time scales.

This spring, David is teaching a successor course, in which 6.008 students are building a complete computer around the Sexium microprocessor. The class, already dubbed 6.009, is being taught through the Edgerton Center as an informal seminar. Students have breadboarded the computer, built an emulator card using a Xilinx Field-Programmable Gate Array (FPGA) that fits in the socket intended for the Sexium, and debugged the system so that it successfully passes the regression suite. They are presently developing a serial interface and beginning the software project.

There are several ways that 6.008 can shape future education at MIT. It is possible that the class could be repeated next IAP; these notes should make the teaching job substantially easier. More significantly, 6.008 has proven that MIT undergraduates can learn VLSI design and that it is an exciting and motivational enhancement of the curriculum, as well as a skill much in demand by today's employers. Hence, there are at least two ways that VLSI can move into the undergraduate experience.

The most straightforward is teaching VLSI as an undergraduate laboratory subject, in much the same way that 6.111 is presently taught. VLSI is an ideal medium for illustrating the issues of complex engineering systems. Background in VLSI design is becoming ever more important and would be a valuable skill for undergraduates seeking summer and permanent employment. As we have demonstrated, only a elementary knowledge of digital circuits is required; indeed, having no background at all may be helpful because there will be few lessons from TTL design that have to be unlearned as the implementation medium changes.

This new undergraduate laboratory should and easily can be designed to satisfy the Institute Lab requirement. Licenses to run Cadence on Athena should be obtained. TAs well-versed with the Cadence tools are vital; this would be difficult at first, but would become easier as the critical mass of VLSI experts at MIT increases. The class is clearly a major undertaking and would require a professor extremely dedicated to undergraduate education, but would give MIT the lead in innovative teaching of VLSI at the university setting.

A second place where VLSI should enter the curriculum is through better-integrated core subjects. The bulk of real-world digital design is no longer based on 74-series TTL chips; MIT should update its courses to reflect this by introducing VLSI in the context of

digital electronics (6.004 and 6.111). PALs, FPGAs, gate arrays, standard cell designs, and full custom chips offer a wide spectrum of tradeoffs between design time and circuit efficiency that a good engineer should understand. Similarly, since the major application of semiconductors is in fine-line CMOS processes, 6.012 should better emphasize MOSFETs and device tradeoffs and limitations that affect performance of digital circuits. With so many faculty members versed in so many different applications of VLSI, MIT should be able to build a consensus for change and modernize the core curriculum to reflect the CMOS revolution that we are experiencing.

Bibliography

Merrill Brooksby and Patricia Castro, "University and Industrial Cooperation for VLSI," Hewlett-Packard Journal, June 1981.

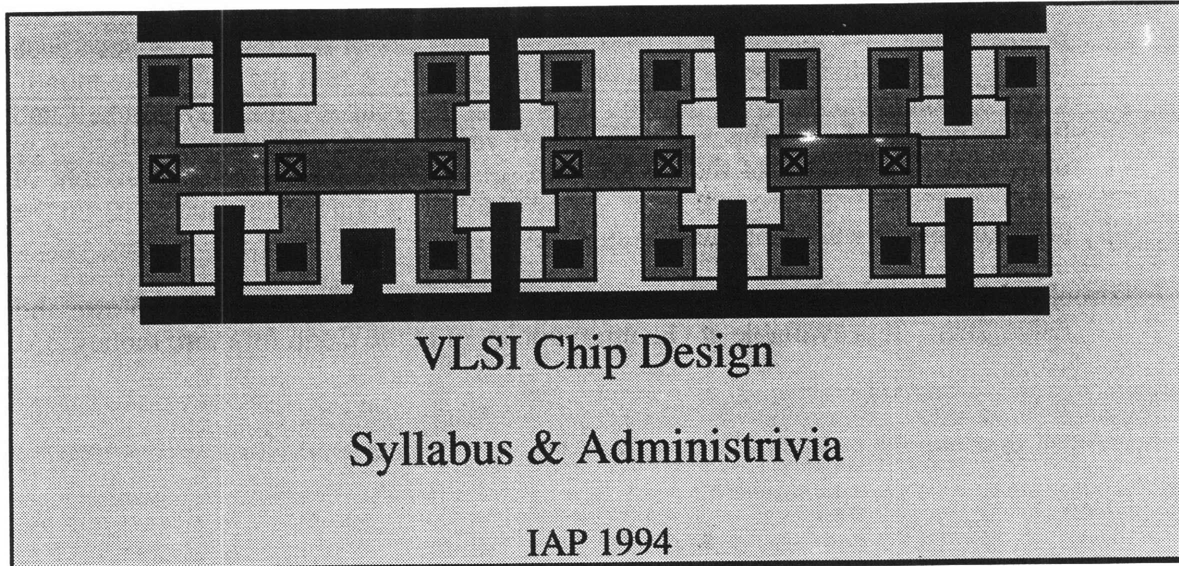
William Dally, "6.845 Lecture Notes," MIT, Cambridge, Mass, September 27, 1993.

David Harris, "Underground Guide to Chip Design," MIT Concurrent VLSI Architecture Memo #45, August 1992.

Neil Weste and Kamran Eshraghian, "Principles of CMOS VLSI Design," Reading, Mass.: Addison-Wesley, 1993.

Appendix A: Course Handouts

This appendix includes all of the handouts produced as part of the 6.008 course, including administrative material, documentation for the CAD tools, problem sets, the Sexium design reference, and the notes taken by student scribes. It also includes the evaluations of the class provided by the students.



General: Chip design was once a black art requiring years of obscure study, but now is possible for anyone with a basic knowledge of digital electronics. In this hands-on Edgerton Seminar, we will learn to design chips from the ground up. The first two weeks introduce the fundamental ideas of circuit design in CMOS and of chip layout. In the second half, we study microprocessors, then build an Unintel Sexium processor. At the end of the month, we will send the class design out for fabrication.

Syllabus:

M3	Introduction. Administrivia. Switch-based circuits. MOSFETs. Examples.
W5	Chip Fabrication. Design Rules. Layout. Low-level examples.
F7	High-level layout. Floorplanning. Simulation.
M10	Flip-flops in VLSI. Counter example.
W12	Special-purpose circuits: RAMs, ROMs, PLAs. PLA examples.
F14	Class project. Overview of microprocessors. Programming Unintel Sexium.
T18	Microarchitecture of Sexium. Trace of program execution.
W19	Class project issues. Project lab.
F21	Project slack time. Tour or guest lecture.
M24	Pads, electromigration, and other Deep Dark Secrets.
W26	Fabricating chips. End-of-class party.

Lectures are held 2-4 PM in 1-115. Attendance is mandatory; if you miss two classes without making advance arrangements, you will be asked to drop. Extra meeting time for the project may be scheduled if there is demand later this month.

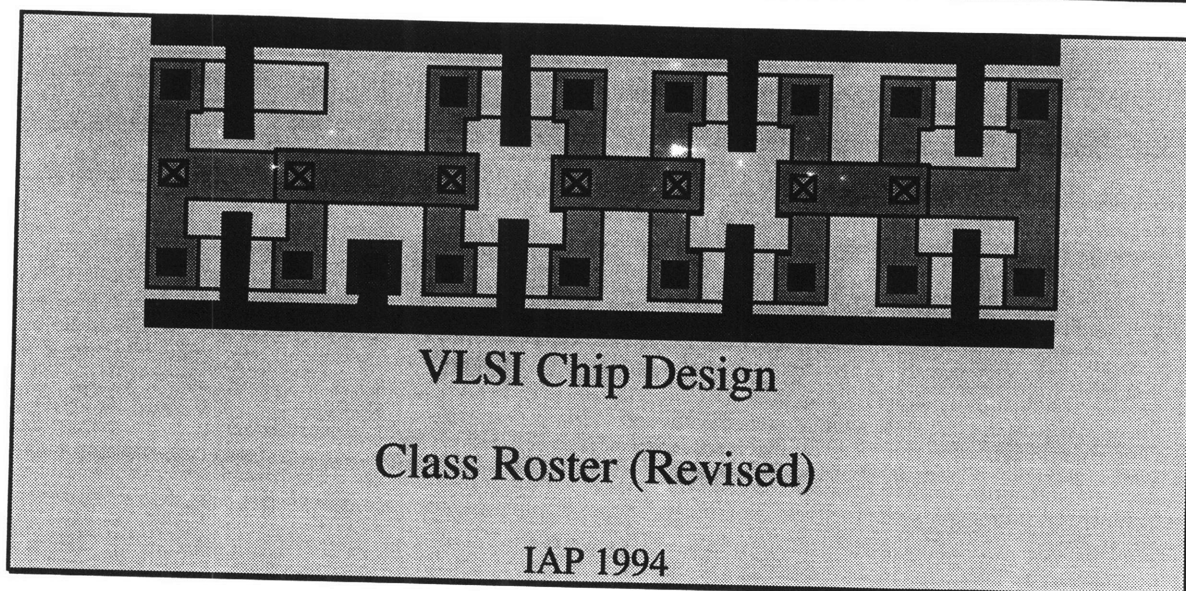
Teaching Staff:

Professor Bill Dally	NE43-620A	x3-6043	billd@ai.mit.edu
David Harris	East Campus	x5-6373	harrisd@mit.edu

Office hours for Professor Dally are after class 4-5 and by appointment. David Harris will be on call between 11 AM and 11 PM to help with questions, homework, etc. If he is not at home, finger him on Athena and at ai.mit.edu.

Homework Policy: Most of the learning in this class will occur while working on the problem sets. We anticipate an average of five hours of homework between each lecture, tending to be less near the beginning and more near the end of the month. Cooperation is encouraged, but you must write up your solutions by yourself and list the names of the students with whom you worked. To pass the class, you must turn in at least 8 of the 10 problem sets and show a reasonable amount of effort on each. Feel free to contact David Harris if you have any questions on the problem sets or want hints getting started.

Textbook: The recommended text is *Principles of CMOS VLSI Design*, by Weste and Eshraghian. It is available at Quantum books, or at the Coop for more money.

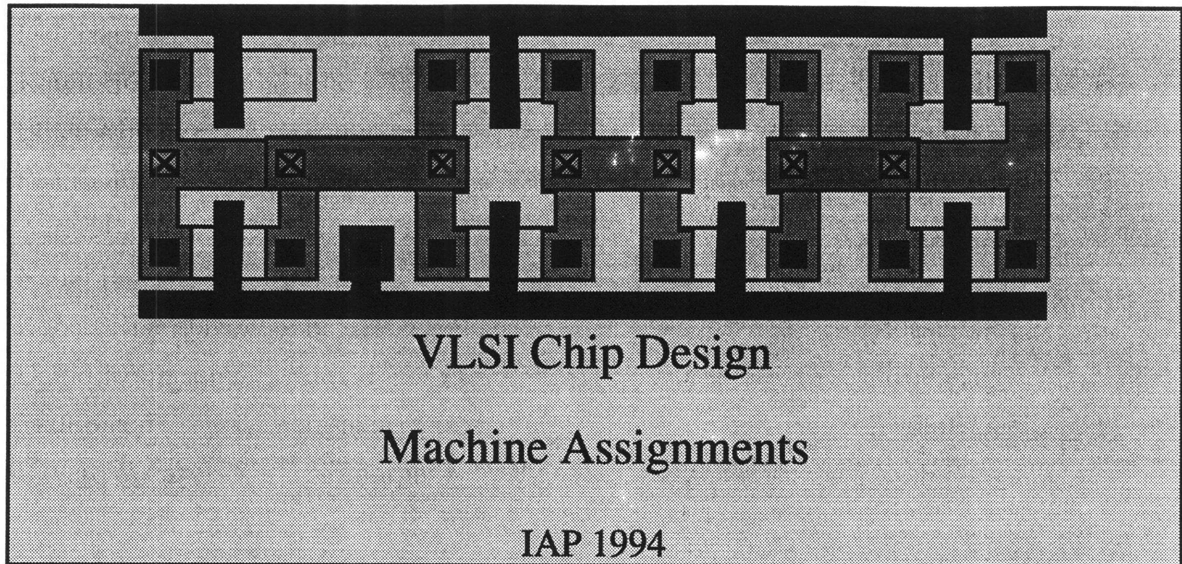


Teaching Staff:

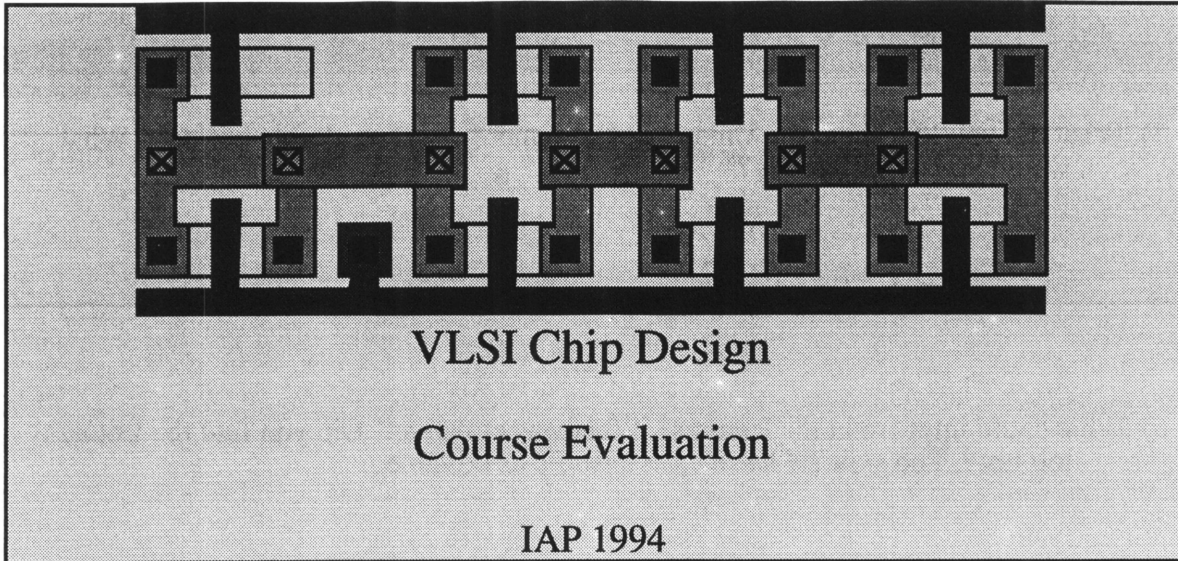
Bill Dally		billd@ai.mit.edu	x3-6043
David Harris		harrisd@mit	x5-6373

Students:

Ruben Agin	'96	ragin@mit	x5-7693
Jeff Bowers	'97	jbowers@mit	x5-9303
Dan Hartman	'95	omv@mit	x5-7563
Karen Ho	'94	kho@mit	x5-8815
Ethan Mirsky	'95	eamirsky@mit	x5-7574
Peter Orondo	'95	podo@mit	492-5722
Aarati Parmar	'96	aaratip@mit	x5-8588
Nehal Patel	'96	nehal@mit	x5-8771
Robert Ristroph	'96	rgr@mit	247-0506
David_Robinson	'97	david_r@mit	x5-9202
Matt Sakai	'96	msakai@mit	x5-6487
Bayard Wenzel	'96	biomorph@mit	262-5090



Student	Username	Password	Machine
Ruben Agin	ragin		alf
Jeff Bowers	jbowers		bc
Dan Hartman	omv		caffeine
Karen Ho	kho		cold-milk
Ethan Mirsky	eamirsky		flapjack
Peter Orondo	podo		fruity-pebbles
Aarati Parmar	aaratip		grits
Nehal Patel	nehal		jelly
Robert Ristroph	rgr		toast
David Robinson	david_r		tropicana
Matt Sakai	msakai		wom
Bayard Wenzel	biomorph		wsb



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

2) What did you learn?

3) What were the best and worst parts of the class? What would you do differently in the future?

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

5) What did you like about the Unintel Sexium microprocessor design project? What would you do differently?

6) What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.

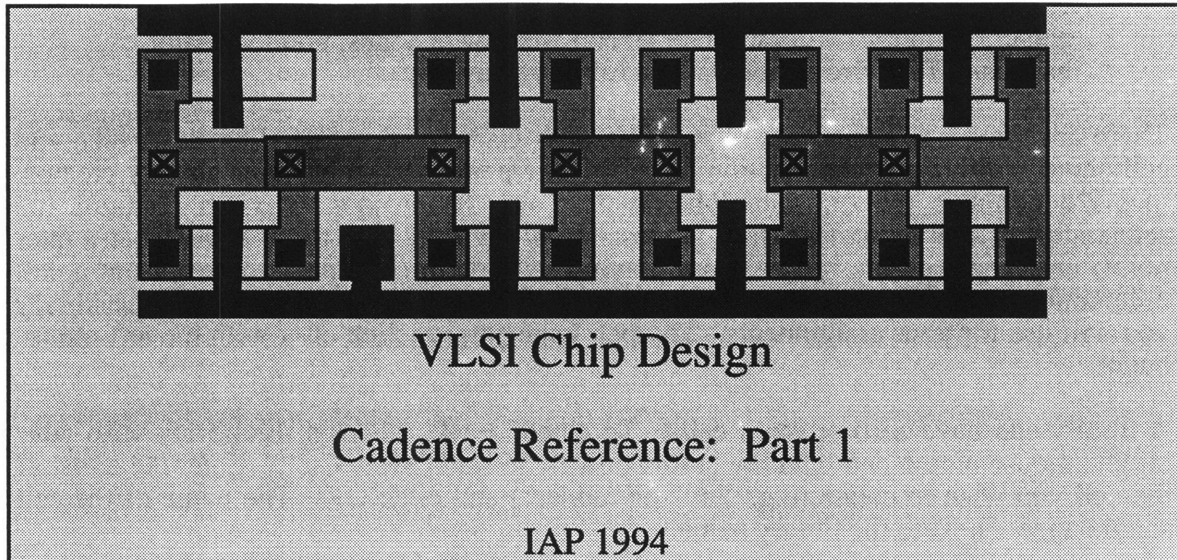
7) How much time did you spend outside of class between each lecture?

8) Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?

9) Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?

10) The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.

11) If this class were to be offered again next year, would you be interested in TAing?



The Cadence Design System is a powerful Computer-Aided Design (CAD) tool set that we will be using for VLSI layout. It has zillions of fancy features that we won't be using in this class. We will start as simple as possible and work our way through the tools over the next few assignments. If you can't hold in your curiosity about a tool, feel free to ask after class.

You will be running the tools on Sparcstation computers at the Artificial Intelligence lab via a remote login from Athena. You will be assigned a machine at the AI lab, a username, and a password. The Cadence tools burn lots of computer cycles on the machines that you will be using, so please be polite. You are a guest on a machine that is probably being used by a graduate student, and you may be asked to log out if the machine is overly loaded. If so, contact David Harris for another machine that you can use. Also, disk space is limited, so don't keep more than one backup copy of your work and throw out old files when you no longer need them.

To run the Cadence tools, use the following steps.

1) Log into a **color** Athena workstation using your normal account.
   Color workstations are available in W20-575, 1-142, 2-225, 4-167, 10-500, 10-600, 11-113, 11-116, 16-034, 37-312, 37-318, 37-322, and E51-007.
2) Type `xhost+`
3) Type `telnet machine.ai`
   machine is the name of your assigned machine
   Enter your AI lab username and password at the prompt
4) If this is your first time logging in, please change your password by typing `passwd`
   You will be prompted to enter your old password, then a new one.
   Please choose a password that is not in the dictionary or otherwise obvious.
4) Type `setenv DISPLAY hostname:0.0`
   hostname is the name of your Athena workstation (e.g. m11-666-13)
   The hostname should be printed on a sticker attached to the workstation
5) Type `startCds -p sexium`
   This starts the Cadence tools running on the sexium project
6) Place the Command Interpreter Window (CIW), bogus window, and Library Browser
   Ghostly images of windows will appear three times. Click the middle button to place them on the screen. The first window is the

Command Interpreter Window. Cadence prints information, warnings, and errors in this window. The second window is bogus; nothing really happens. The third window is the Library Browser.

There are three basic concepts to working with Cadence designs: libraries, cells, and cellviews. A library is a collection of cells. There are several libraries already created. mosis2n contains the transistors being used for the project. sexium contains the schematics of the Unintel Sexium processor. We will discuss these libraries at a later point; until then, please refrain from playing with them (it is easy to check out a cell unintentionally). In addition, there is a library with your username. This is the library you will use for your assignments. To open your library, click on it with the left mouse button.

A library can have zero or more cells. To create a cell, click on the library with your middle mouse button and hold the button down. Select "Create Cell." Enter the name of the cell you wish to create (e.g. "inv" or "adder") and click OK. The name of the cell should appear next to the library name.

Each cell may have several cellviews, containing various representations of the cell. The most important view in this class is the "layout" cellview. In the layout cellview, you draw rectangles representing the masks for metal, poly, diffusion, etc. Other cellviews include "schematic" used for a schematic-level representation of a circuit, and "extracted" used for running simulations and verification of your layout. We will use these cellviews in later assignments.

To create a "layout" cellview, click on the cell with the middle mouse button and select "Create Cellview" from the menu. Type "layout" as the view and click OK.

To edit your layout, click on the cell with the right button to select it if it is not already selected. Then click on the cellview with the middle button and choose "Edit" from the menu. Two windows should appear. One is a palette with the various layers from which to choose. The other is the layout window.

The menus in the layout window offer lots of commands to help create your layout. The most commonly used ones have handy keyboard shortcuts. Some of the important commands are described below (shortcut listed in parentheses):

Rectangle (r)
Draws a rectangle in the currently selected layer.
Change layers by clicking on the palette

Move (m)
Moves the currently selected rectangle(s)

Copy (c)
Copies the currently selected rectangle(s)

Delete (delete key)
Deletes the currently selected rectangle(s)

Undo (u)
Undo the last action, if you didn't want to delete that rectangle...
Undo can be repeated up to five times.

Stretch (s)
Stretches the selected edge of a rectangle

Save
Saves your work. Do this often

Ruler (r)

Measures the length in lambda
Use Shift - r to remove rulers

Zoom (z)
Zooms in on a portion of the design

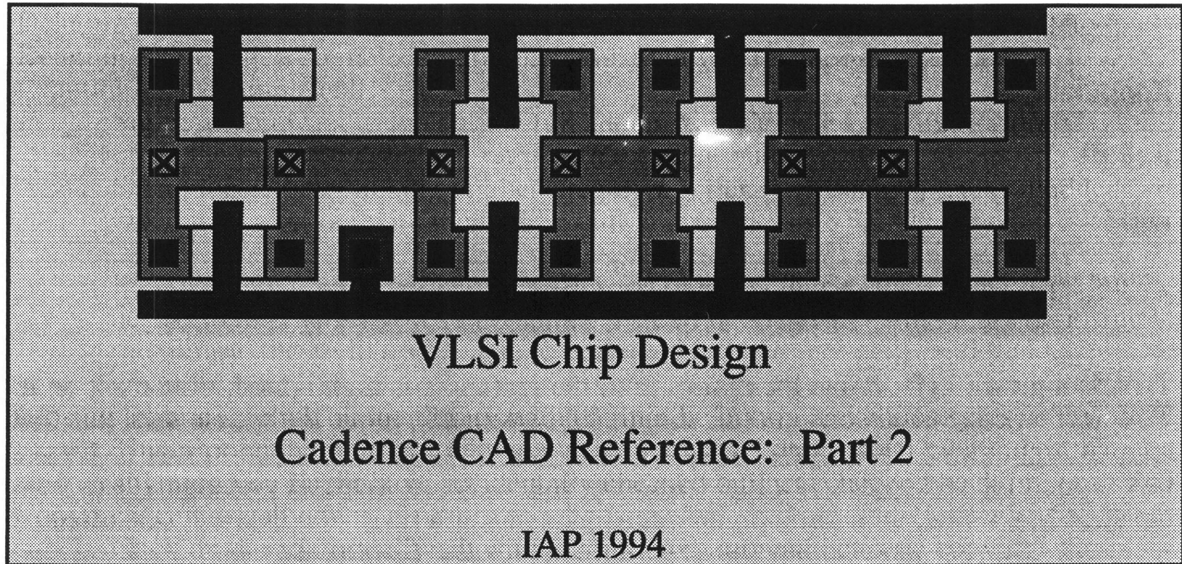
Full (f)
Zooms out to show your full design

DRC
The Design Rule Checker, under the Verify menu
This scans your design, looking for design rule violations
Use the Verify: Markers: Explain command to explain any violations

To select a rectangle, move the mouse until the rectangle is highlighted, then click on it. To select an edge of a rectangle (for the stretch command), move the mouse until just that edge is highlighted, then click. To select multiple rectangles, drag the mouse to form a box around the rectangles you like. Another way to select multiple rectangles is to hold the shift key down, then click on the rectangles one at a time. To deselect one selected rectangle without deselecting the rest, hold down the Control key and click on that rectangle. To get out of an editing mode (like drawing rectangles, or moving), type Control-C.

Save your work often. Cadence has the annoying habit of crashing from time to time. This usually involves slowing your machine to a crawl, then having all of the windows disappear and losing your work. If Cadence crashes, start it up again with startCds command. If the problem persists, contact David.

To quit Cadence, choose Quit from the Open menu in the Command Interpreter window.



This manual documents more handy Cadence commands

Managing your Cadence files

Deleting a cellview: click on the cellview with the middle button and choose delete
Deleting a cell: click on the cell with the middle button and choose delete
Examining an example cell:

Suppose you want to look at the full adder example in the examples library.
Open the library by clicking on it with the right button.
Select the cell by clicking on the category LECTURE3, then the cell fulladder
Open it by clicking with the middle button and choosing READ

Please do not open other people's files for EDIT. When you do, you check out access to the file. If you do not check the cell back in by closing it, then closing the library, they will not be able to edit their cell and they will send you nasty email.

Closing a library: click on the library with the middle button and choose close

This is polite to do after looking at somebody else's library. It guarantees that anything you might have accidentally checked out gets checked back in.

Copying a cell or cellview:

Click on the cell or cellview with the middle button and choose copy
Enter the name of the library and cell where you want the copy to go

Layout Editor Commands

To merge several rectangles into one polygon, select the rectangles and choose Merge from the Tools submenu of the Edit menu.

To add a label to a wire, select the labels layer from the palette. Use the Labels command (keyboard shortcut = l) to bring up a labels box. Enter the name of the label, then click where you want the label to go. Labels must be attached to a piece of layout (i.e. on top of the layout). If a label is on several layers, it names metal 2 if that is present, then metal1 if that is present, or else poly if no metal is present.

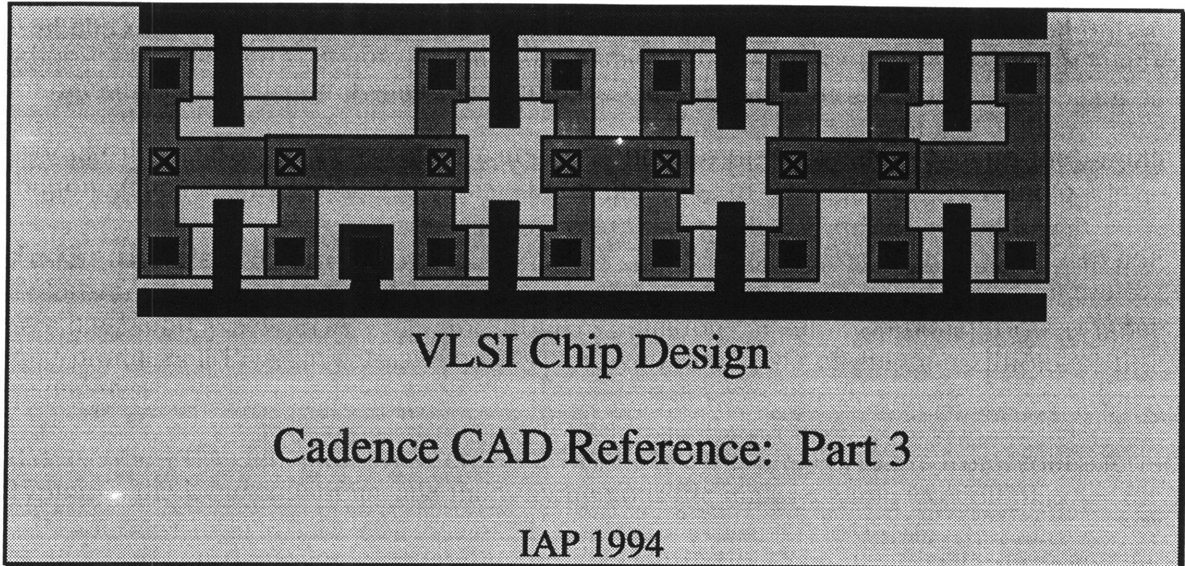
To place pins on a layout, use the Pins command (p). Enter the name of the pin and indicate if it is an input or output. Make sure you have selected the palette layer to be the same as the wire to which you are attaching the pin. Then draw the pin;

it should be 4 λ x 4 λ, overlapping the wire by 2 λ. (Exception: Poly pins can be 4 λ x 2 λ.)

To place an instance of a cell, choose the Instance (i) command. Enter the name of the cell. Click to place the instance.

To create a rotated or flipped copy of a piece of layout, select the rectangles or instances. Choose copy. Click the right button to rotate, or hold the shift key and click the right button to flip. Click the left button to place the copy.

You may wish to poke around the Design, Window, Create, Edit, and Misc menus. Don't explore too much in the Tools or Verify menus; some tools may not be correctly installed and may crash Cadence.



Creating Cell Categories

You may be sick of having a zillion little cells littering your library. I'm certainly sick of looking at them. You can solve this problem by grouping your cells into a "Cell Category." To create a cell category, click on the library with the middle button and choose "Create Cell Category." Enter the name of the category (e.g. PS3) and the name of one of the cells that you want to put into the category (e.g. fulladder).

To add the rest of the cells to the category, click on the cell category with the middle button. Choose Add Cell to Category. Enter the name of the cell that you want to add.

Using the PLA Generator

We have a nifty new PLA generator. It takes equations in sum of product form as inputs and produces a PLA layout as output, saving you the hassle of drawing zillions of little rectangles.

To use the PLA generator, first create a PLA file by typing `emacs` from your xterm, typing in the logic equations, and then saving your file under a name like jolt.pla.

A PLA file can have logic equations, product definitions, input order definitions, and comments. Comments are lines that begin with #; they are ignored. It is a good idea to put a comment at the beginning of your PLA file saying what the file is, who created it, when it was created, and what your inputs and outputs mean, so other people (like me) can understand your code. An example is listed below. Logic equations must be in sum of products form with the single quote indicating inversion.

Product definitions name a product of various inputs. If a particular product is to be used more than once, it should be defined, so that the same product line is used in all of the outputs. Product definitions also tend to make code more readable. A product definition is a line with the name of the product, followed by the < sign, followed by a product of inputs.

Finally, input order definitions are used to indicate what order the inputs appear in from left to right. Input order definitions are a line beginning with the word "inputorder"

followed by one or more names of inputs. If inputs are not ordered using the inputorder command, they appear in the order that they are defined. Product terms and outputs also appear in the order they are defined.

```
# Adder.pla
# Written 1/8/94 by David Harris
# This PLA adds a, b, and cin to produce sum and cout.

cout = cin*a + cin*b + a*b
sum = cin*a'*b' + cin'*a*b' + cin'*a'*b + cin*a*b
```

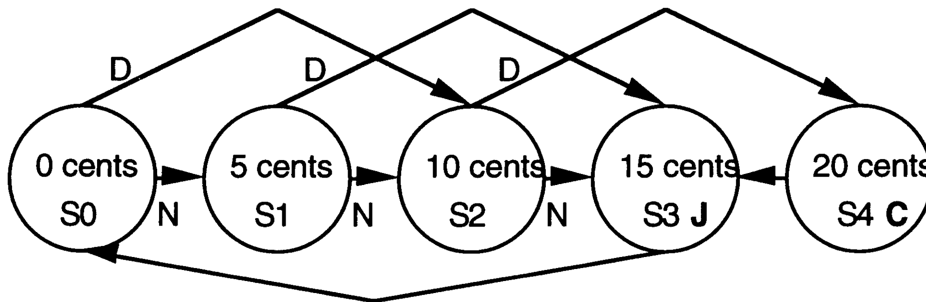
Once you have created your PLA file, go to the Command Interpreter Window. Type

```
(plagen "name.pla" "library" "cellname")
```

where name.pla is the name of the PLA file, library is the library where the PLA should be created, and cellname is the name of the cell that you want to create. For example, if Jeff wants to create an adventure game PLA in his library, he might type

```
(plagen "adventure.pla" "jbowers" "adventure")
```

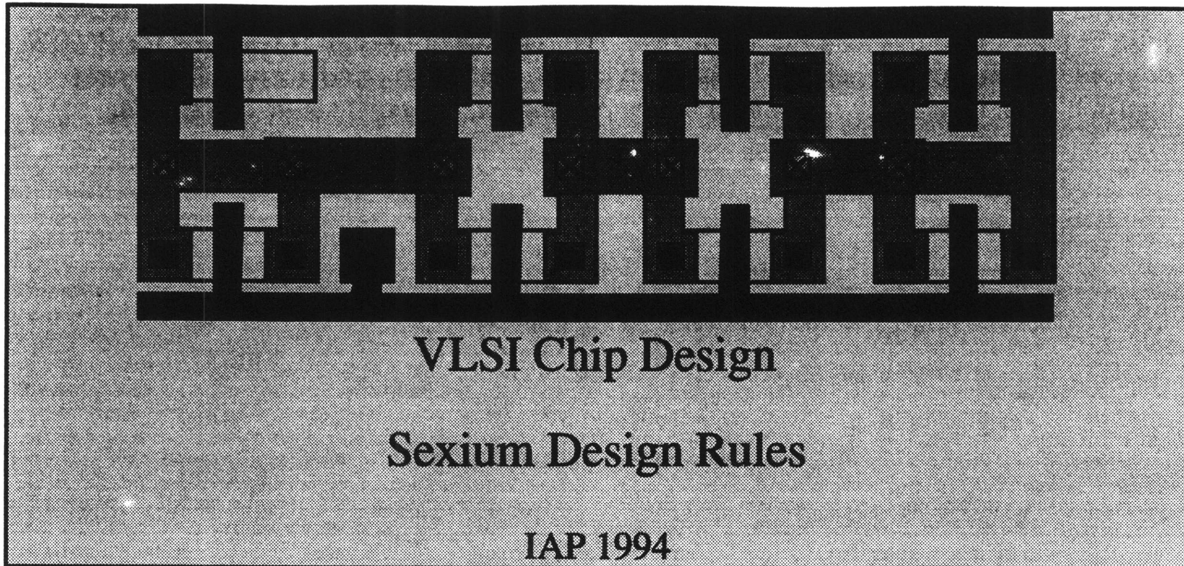
A more complicated PLA example using the product definitions and inputorder commands is shown below. It is a model of a Jolt machine that accepts nickels (N) and dimes (D) and will give you a dose of sugar and caffeine (J) when you insert 15 cents, along with change (C) if you insert 20 cents. The state transition diagram appears first. Three flip-flops, S[0], S[1], and S[2], encode the present state; the PLA generates the next state and outputs as a function of the present state and inputs.



Current State			Inputs		Next State			Outputs	
S[2]	S[1]	S[0]	N	D	$S_{new}[2]$	$S_{new}[1]$	$S_{new}[0]$	J	C
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	0	0
0	0	0	0	1	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0
0	0	1	1	0	0	1	0	0	0
0	0	1	0	1	0	1	1	0	0
0	1	0	0	0	0	1	0	0	0
0	1	0	1	0	0	1	1	0	0
0	1	0	0	1	1	0	0	0	0
0	1	1	0	0	0	0	0	1	0
1	0	0	0	0	0	1	1	0	1

```
# Jolt Machine FSM
# David Harris   January 1994
# Note:  no checking is done that the user
#        doesn't put in coins at the wrong time

# Define input order
inputorder N D S[0] S[1] S[2]

# Product terms
S1dime   = S[2]' + S[1]' + S[0]  * D
S2nickel= S[2]' + S[1]  + S[0]  * N
S4       = S[2]  + S[1]' + S[0]'

# Outputs
# (Note:  output definitions must each appear on one line,
#  not spread over several the way they are printed here)
Snew[2] = S[2]' * S[1]  * S[0]' * D
Snew[1] = S[2]' * S[1]' * S[0]' * D +
          S[2]' * S[1]' * S[0]  * N +
          S[2]' * S[1]  * S[0]' * N' * D' +
          S1dime + S2nickel + S4
Snew[0] = S[2]' * S[1]' * S[0]' * N +
          S[2]' * S[1]' * S[0]  * N' * D' +
          S1dime + S2nickel + S4
J       = S[2]' * S[1]  * S[0]
C       = S4

# Note:  Snew[2] can't be written as S2 * D
# because nothing can multiply a product term.
```

If you have any problems using the PLA tools, please contact David Harris right away.



Physical Design Rules

Level	Minimum Width	Minimum Spacing
poly	2	2
diffusion	4	4
metal1	4[4]	3
metal2	4	4
contact	2 x 2 exactly	2
via	2 x 2 exactly	3

Figure 1 a more detailed summary of lambda-based design rules. They allow width 3 metal1 and diffusion. In most cases, this is not very important, while our rules tend to keep lines on a 4 lambda grid, especially when a minimum spacing of 4 is used for diffusion and metal1.

Datapath Design Rules

Datapaths are designed in a funny way. The individual cells are first drawn with a fixed width of 80. Metal 1 lines for control and power are run with a preferred horizontal direction; metal 2 lines for the bit lines run vertically, along with poly for gates and local interconnect. Figure 2 is a plot of a full adder done in the datapath style.

When datapath cells get assembled into a full-fledged data path, they are rotated to have a fixed height of 80 / bit and a variable width. Then they are stacked as shown in Figure 3, to produce an 8 bit tall data path.

Standard Cell Design Rules

Standard cells are designed more straighforwardly. Metal 1 lines are again used for control and power with a preferred horizontal direction. GND is placed at the bottom of the cell and VDD at the top; a spacing of exactly 60 from middle of GND to middle of

[4]Technically, metal 1 and diffusion may be 3 lamda wide. However, they must be 4 lambda wide whereever a contact is present, so the 3 lambda width seldom is helpful.

VDD is required. Poly for gates and inputs is run vertically; metal 2 should be used as little as possible and is also run vertically. The top half of the cell is reserved for PMOS devices and the bottom half for NMOS devices so that wells of adjacent cells do not conflict; thus, nwell may not extend below 30 lambda, nor ndiff above 25 lambda.

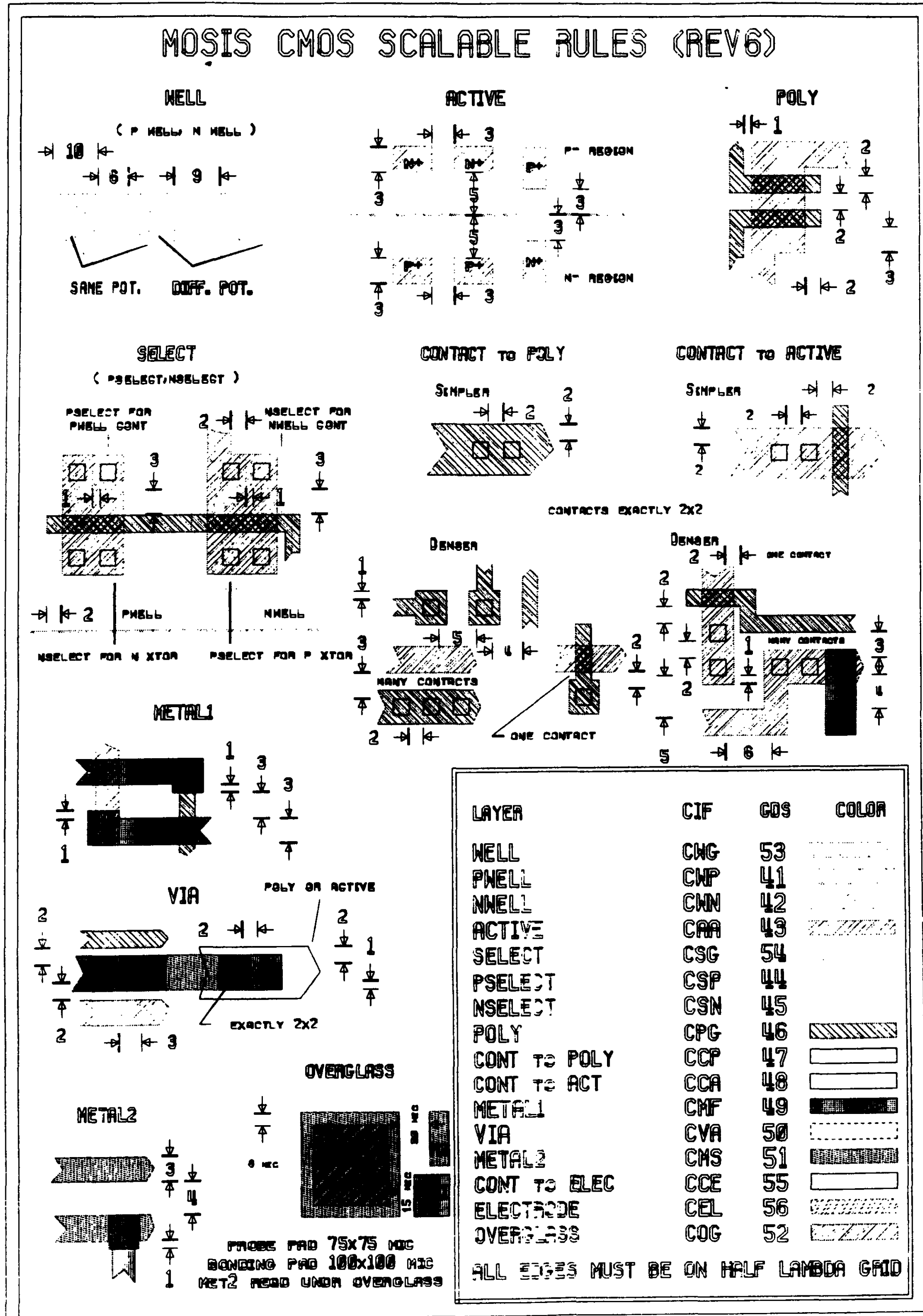


Figure 2: Datapath full adder

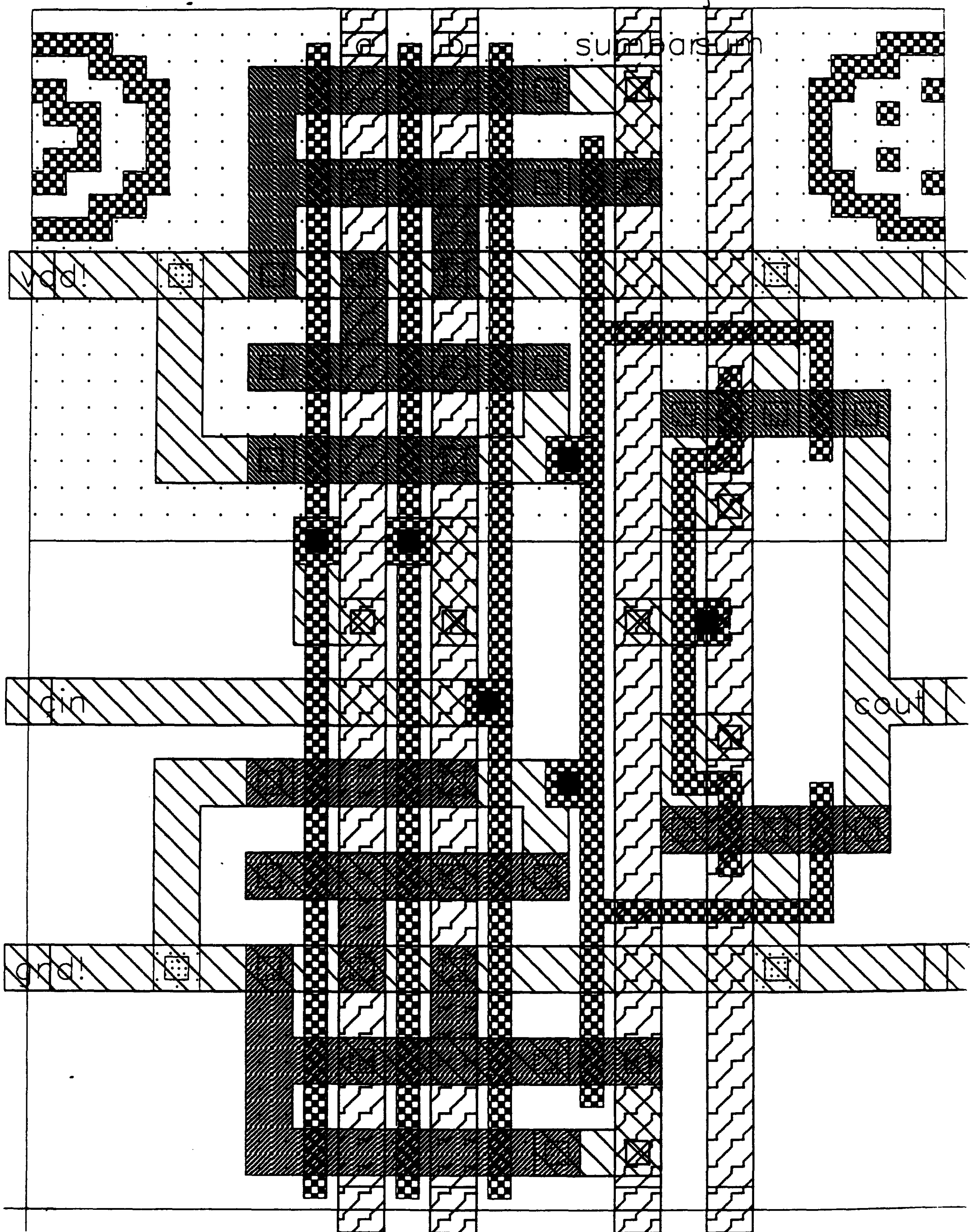
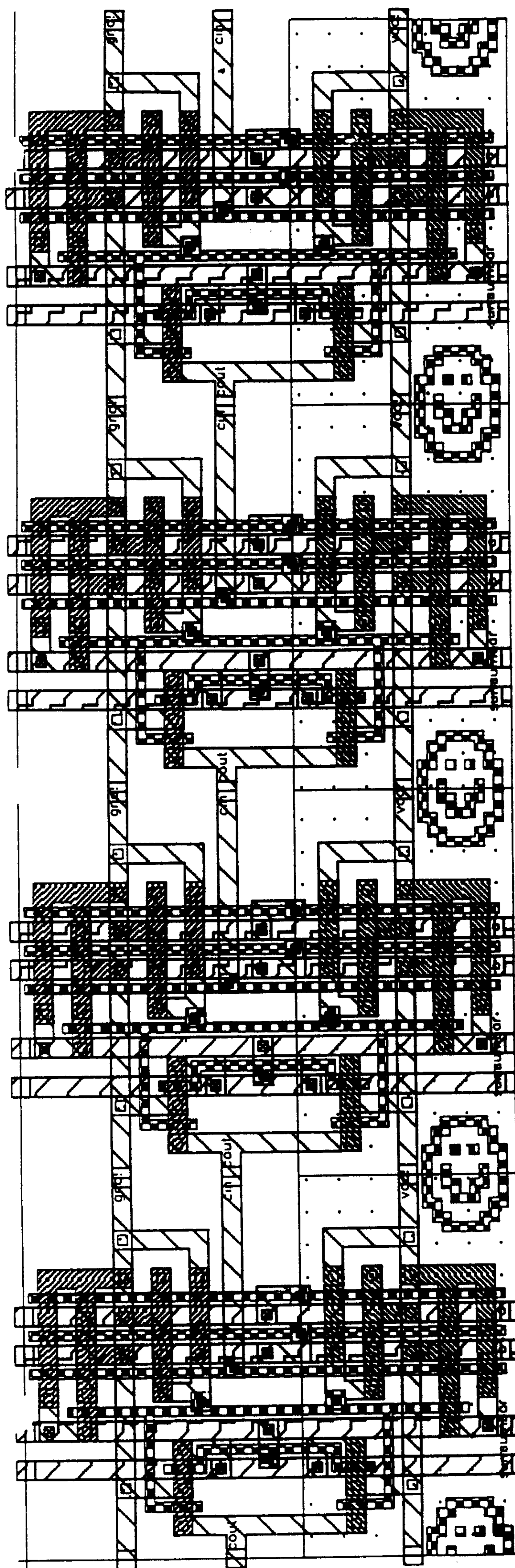
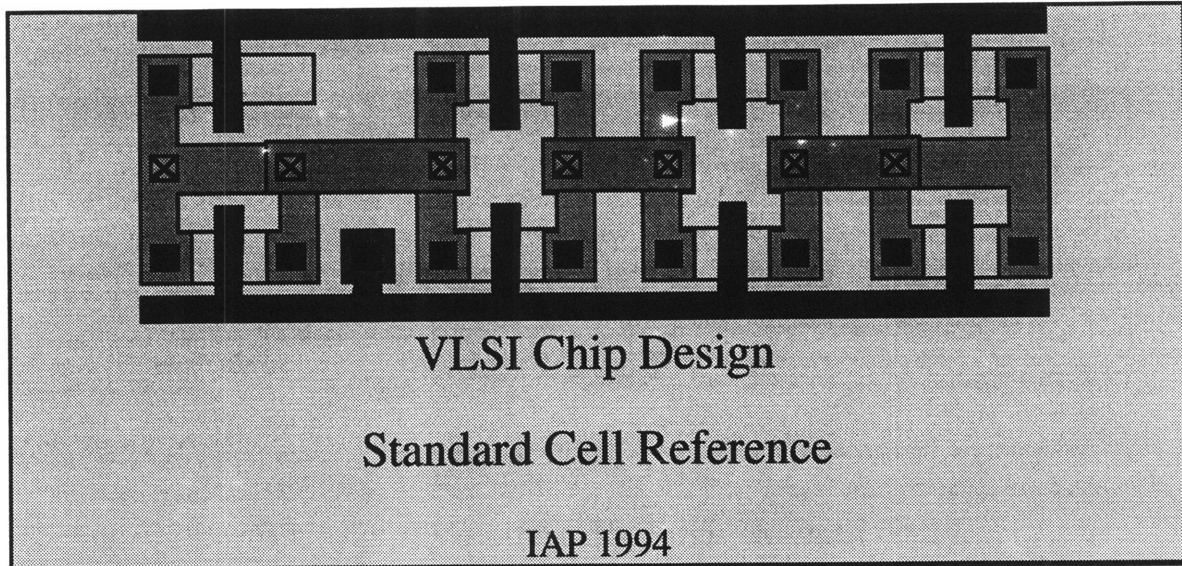


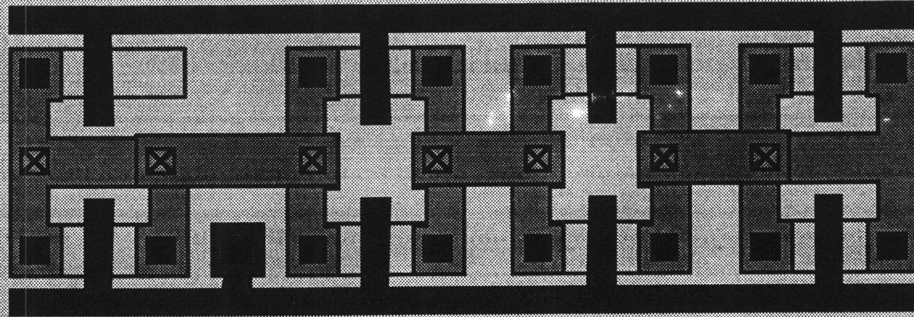
Figure 3: 8 bit datapath





The standard cell library is complete and all of the cells have passed the layout versus schematic check. The following table summarizes the cells available in the library, the width of the cells and the average area per transistor of each cell.

Cell	Creator	Width	Area / transistor
and2	Dan	30	300
aoi3	Jeff / Dan	29	290
flop	Matt	69	230
inv	Ethan / Dan	11	330
invtri	Dan	19	285
latch	Matt	36	216
mx2	Nehal	40	400
mx3	Bayard	88	660
mx4	Bayard	112	672
nand2	Nehal	18	270
nand3	Ruben	32	320
nand4	Matt	30	225
nor2	Nehal	18	270
nor3	Aarati	24	240
nor4	Matt	30	225
or2	Dan	30	300
xor2	Jeff	36	216



VLSI Chip Design

Unintel Sexium Instruction Set Architecture, Microarchitecture & Simulator Reference

David Harris

1/17/94

1 Introduction **77**

2 Instruction Set Architecture **78**
- 2.1 Registers 78
- 2.2 Memory 79
- 2.3 Operations 80
- 2.4 Operation Coding 81

3 Microarchitecture & Floorplan **83**
- 3.1 Top Level Architecture 83
- 3.2 alubox 84
- 3.3 regbox 85
- 3.4 pcmabox 86
- 3.5 control 87

4 Sexium External Interface **90**
- 4.1 Pinout 90
- 4.2 Complete System Schematic 91

5 Simulator Reference **92**
- 5.1 Assembly Language Syntax 92
- 5.2 Programming Example 93
- 5.3 Using MSIM 94
- 5.4 Creating EPROMs 95

Appendix A: Sexium Schematics **96**

1 Introduction

The Unintel Sexium is an 8 bit microprocessor designed for the "6.008:" VLSI Chip Design class. It is intended to be very simple, both so that is easy to understand and so that it can be built in the space available on a MOSIS TinyChip.

The Unintel Sexium features a minimal set of instructions that are sufficient to write any program (as long as enough memory is available). Section 2 describes the Sexium registers and instructions.

Section 3 delves into the microarchitecture, showing the interconnections between the various modules of the processor. Section 4 discusses the pinout of the Sexium chip, including how to hook it up to external circuitry to build a functional computer.

Section 5 is a reference manual for the Sexium simulator program (MSIM), that allows the programmer to write, assemble, and test code on workstations. It also contains example programs illustrating Sexium assembly language.

2 Instruction Set Architecture

The Unintel Sexium is a type of microprocessor called an accumulator machine. To understand the principle of an accumulator machine, imagine adding a list of forty-two numbers as follows:

1) Let A be the first number
2) Add the second number to A
3) Add the third number to A
...
42) Add the forty-second number to A

In this example, we use the variable A to hold (or to accumulate) the sums throughout the computation. In an accumulator machine, there is a register called the accumulator, or A for short, that keeps partial results. Arithmetic operations act on A and one other piece of data and store their result in A.

2.1 Registers

In addition to the accumulator, there are eight other registers that the Sexium programmer can use. Four of the registers are general purpose, while two are dedicated to the current memory address and two hold the program counter.

The four general purpose registers are named R0-R3. They are useful for holding temporary results; arithmetic operations are generally of the form A <- A op Rx where op is the operation and $0 \leq x \leq 3$. For example, consider the following set of instructions:

```
LDM $96         # load the number $96 into A
PUT R0          # put a copy of A into register R0
LDM $04         # load the number $01 into A
ADD R0          # Add R0 to A (A should get $9A)
```

These instructions are written in a form called assembly language. In Sexium assembly language, the $ sign indicates that a constant is written in hexadecimal (base 16). The program adds the numbers $96 + $04 and stores the result ($9A) in the accumulator. LDM stands for **LoaD** **iM**mediate. It loads the hexadecimal number specified into the A register. PUT puts a copy of the accumulator into the register specified. ADD adds the accumulator to the register specified and stores the results in the accumulator.

The memory address registers, called MAH and MAL (**M**emory **A**ddress **H**igh and **M**emory **A**ddress **L**ow), are special. The Sexium can access 65536 = 2^{16} different memory locations. Thus, it needs 16 bits to store an address in memory. MAL holds the 8 least significant bits, while MAH holds the 8 most significant bits. The following program reads a byte of data from memory address $0666.

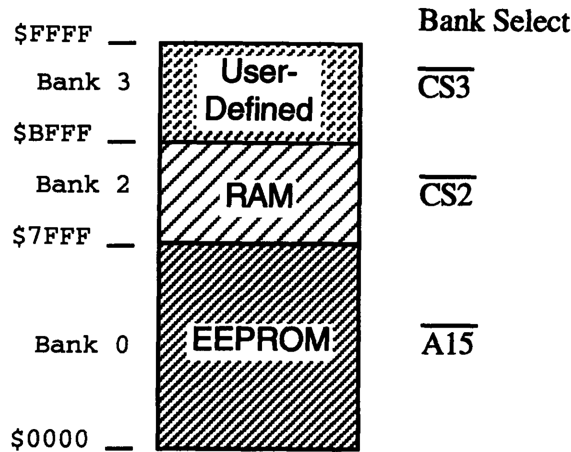
```
LDM $06          # load the high part of the address
PUT MAH          # and set the memory address high
LDM $66          # load the low part of the address
PUT MAL          # and set the memory address low
LDA              # load the accumulator from memory
```

In this example, the LDA reads the memory location specified by MAH and MAL and stores the value in the accumulator.

Finally, there are two registers used to store the program counter, called PCH and PCL (**P**rogram **C**ounter **H**igh and **P**rogram **C**ounter **L**ow). They are very similar to the memory address registers, holding a 16 bit number specifying which instruction the Sexium is currently executing. The programmer seldom directly manipulates the program counter registers (because suddenly the instruction being executed would change), but other jump and branch instructions do change the program counter. After each instruction is executed, the program counter is automatically advanced to point to the next instruction.

2.2 Memory

As mentioned before, the Unintel Sexium has 16 bit addresses, supporting 64K (kilobytes) of memory. This memory is partitioned into 3 banks, one of 32K and two of 16K. Bank 0 is normally reserved for ROM and Bank 2 is, by convention RAM. Bank 3 is presently undefined, but could be used for input / output devices (like keyboards, screens, printers, disk drives, and so forth) or for more RAM or ROM. The memory map below illustrates this partitioning of memory:



When the Sexium is first powered up or reset, the program counter is set to $0000. Therefore, it is important that a program be stored starting at location $0000; otherwise the Sexium would wander off into never-never land executing random garbage. This occurrence is called a "feature."

2.3 Operations

The Unintel Sexium has a minimal instruction set. This makes the construction simpler, but can cause programming to be tedious. The Sexium instructions are summarized in the table below.

Instruction	**Effect**	**Comments**
Arithmetic / Logical		
`ADD reg`	$A \leftarrow A + reg$	**Add** A to register
`AND reg`	$A \leftarrow A \& reg$	Bitwise **AND** of A and reg
`NOT`	$A \leftarrow \neg A$	Bitwise complement (**NOT**)
`SHR`	$A \leftarrow A >> 1, A_7 = 0$	**Sh**ift **r**ight
`ROR`	$A \leftarrow A >> 1, A_7 = A_{0\text{-old}}$	**Ro**ll **r**ight
`PUT reg`	$reg \leftarrow A$	**Put** A into register
`GET reg`	$A \leftarrow reg$	**Get** A from register
`TST reg`	$A_0 \leftarrow$ carry (A+reg) $A_1 \leftarrow$ zero (A+reg)	**Test** A + register and set bits of A accordingly
Memory		
`LDA`	$A \leftarrow Mem[MA]$	**L**oad **A** from memory
`LDI`	$A \leftarrow Mem[MA]$ $MA \leftarrow MA+1$	**L**oad A from memory and **i**ncrement MA
`LDM const`	$A \leftarrow const$	**L**oa**d** **A** im**m**ediately
`STA`	$Mem[MA] \leftarrow A$	**St**ore **A** to memory
`STI`	$Mem[MA] \leftarrow A$ $MA \leftarrow MA+1$	**St**ore A to memory and **i**ncrement MA
Control		

`JMP high low`	PC ← high:low	**Jump** to absolute address
`BRA const`	PC ← PC + signed const	**Branch** to relative address
`CAL high low`	R1:R0 ← PCH:PCL+3 PC ← PC + const	**Call** subroutine and save return address
`RTN`	PCH:PCL ← R1:R0	**Return** from subroutine
`SKZ`	If (A = 0) PC ← PC + 2	**Skip** ahead 2 if A is **zero**

In this table, MA is the 16 bit memory address and PC is the 16 bit program counter. Mem[MA] indicates the data in memory at the address specified by MA. `reg` indicates a register name; it may be `R0`, `R1`, `R2`, `R3`, `MAH`, `MAL`, `PCH`, or `PCL`. `high`, `low`, and `const` indicate byte-sized constants.

Some of the instructions are worthy of extra attention. The TST instruction tests the sum of the accumulator and another register. This sum of two eight bits may produce a 9 bit number. If the 9th bit is a 1, we say that there is a carry. If the eight low bits are all 0, we say that the result is zero. TST sets bit 0 of the accumulator if there is a carry and bit 1 if there is a zero. It makes all other bits of the accumulator 0.

JMP jumps to any address in the program. CAL is like JMP, but stores the return address in R1 and R2; the RTN command fetches this address and returns. BRA is a relative branch; it adds an eight bit two's complement number to the current program counter. This number may be either positive or negative, resulting in a branch of -128 to 127 from the present location. SKZ is the only conditional instruction on the Sexium. It tests if the accumulator contains a zero, and if so, skips the next two bytes in the instruction stream. Normally, a BRA instruction should follow the SKZ; since the BRA instruction requires two bytes, everything works out cleanly. If a one-byte instruction follows SKZ, an extra one-byte instruction must be added to pad out the instruction stream or SKZ will execute at the wrong place. Three byte instructions (i.e. JMP or CAL) may not follow a SKZ because the skip might end up in the middle of the instruction and crash a horrible death.

2.4 Operation Coding

Instructions are stored as byte-sized codes. Five bits specify the operation; the three remaining bits specify one of eight registers on instructions that require a register. Some instructions must be followed by one or more additional bytes; for example, LDM must be followed by one byte to load into the A register.

The following table shows the coding of each instruction:

Instruction	Opcode byte	Second byte	Third byte
ADD	r e g 0 1 0 0 0		
AND	r e g 0 1 0 0 1		
NOT	0 0 0 0 1 1 0 0		
SHR	0 0 0 0 1 1 0 1		
ROR	0 0 0 0 1 1 1 0		
PUT	r e g 0 1 0 1 1		
GET	r e g 0 1 1 1 1		
TST	r e g 0 1 0 1 0		
LDA	0 0 0 0 0 0 0 0		
LDI	0 0 0 0 0 0 0 1		
LDM	0 0 0 0 0 1 0 0	const	
STA	0 0 0 0 0 0 1 0		
STI	0 0 0 0 0 0 1 1		
JMP	0 0 0 1 0 0 0 0	high	low
BRA	0 0 0 1 0 0 1 0	const	
CAL	0 0 0 1 0 0 0 1	high	low
RTN	0 0 0 1 0 1 0 0		
SKZ	0 0 0 1 0 0 1 1	const	

Register Codes

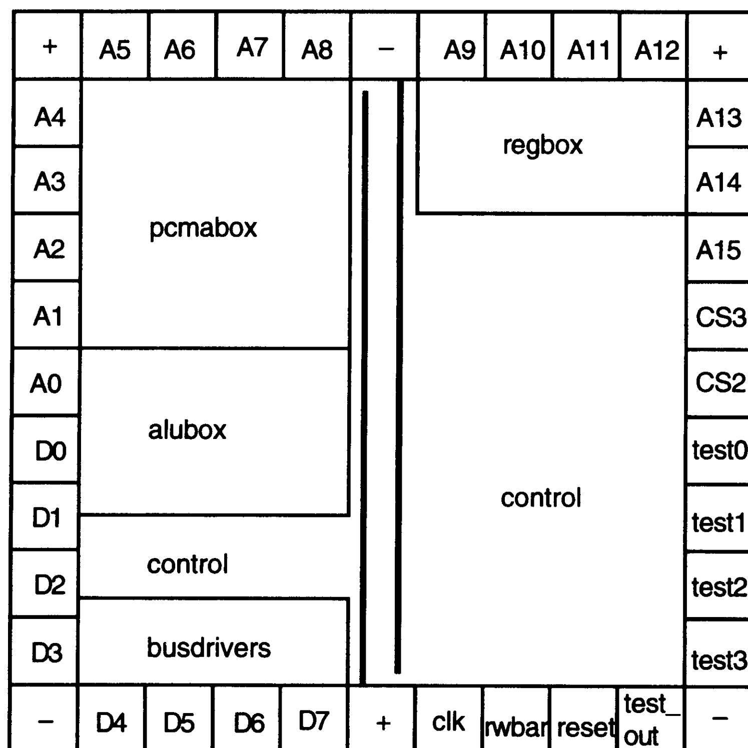
Code	Register
000:	R0
001:	R1
010:	R2
011:	R3
100:	PCL
101:	PCH
110:	MAL
111:	MAH

3 Microarchitecture & Floorplan

The microarchitecture of a digital system is the arrangement of modules and busses to implement the functionality. We will examine the Unintel Sexium microarchitecture, starting with a floorplan of the entire chip, then delving into each of the four major blocks: control, regbox, alubox, and pcmabox.

3.1 Top Level Architecture

The Unintel Sexium will be fabricated on a Mosis TinyChip, 2200 λ on a side. It will be packaged in a 40 pin ceramic DIP; pinouts are described in chapter 4. Each pad is 200 λ square, leaving 1800 λ per side for the actual logic, as shown in the floorplan below:



Floorplan: 1 inch = 600 λ

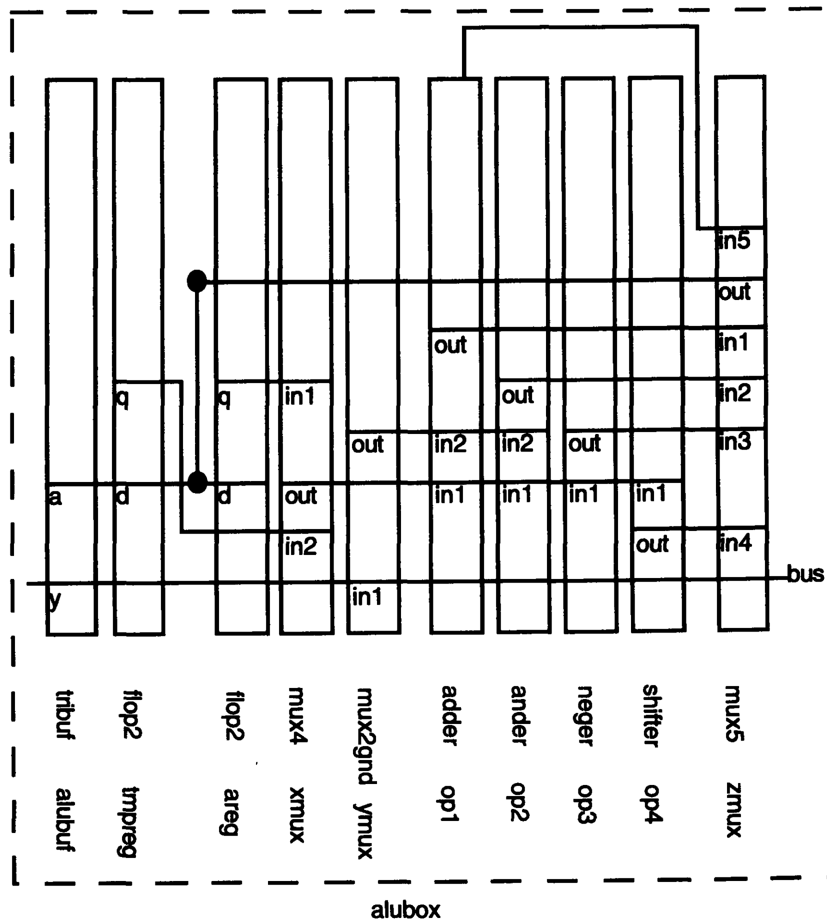
The bottom portion of the chip is the 8 bit datapath. It is based on a single bus that connects the ALU and the eight registers. Each datapath cell is 80 lambda tall, for a total datapath height of 640; above the datapath are power and ground busses and a row of

standard cells to generate control signals. The regbox extends into the top half. A block diagram of the datapath modules at a finer level of detail is located in the appendix.

The top portion of the chip is devoted to the control logic. This control is implemented as a finite state machine, built from a large PLA and some standard cell flip-flops. Multiplexors to send out internal signals to the test pins are also in the control box. The control extends into the datapath for the eight bit instruction register (IR).

3.2 alubox

The alubox contains the A and TMP registers, multiplexors to select the X and Y inputs to the function units, a multiplexor to select the appropriate function and place it on the Z output, and a tristate buffer to drive Z onto the bus. The TMP register is accessed in the same way as the accumulator and is used by various branching instructions to store intermediate sums. The schematics of the alubox are shown in the appendix.

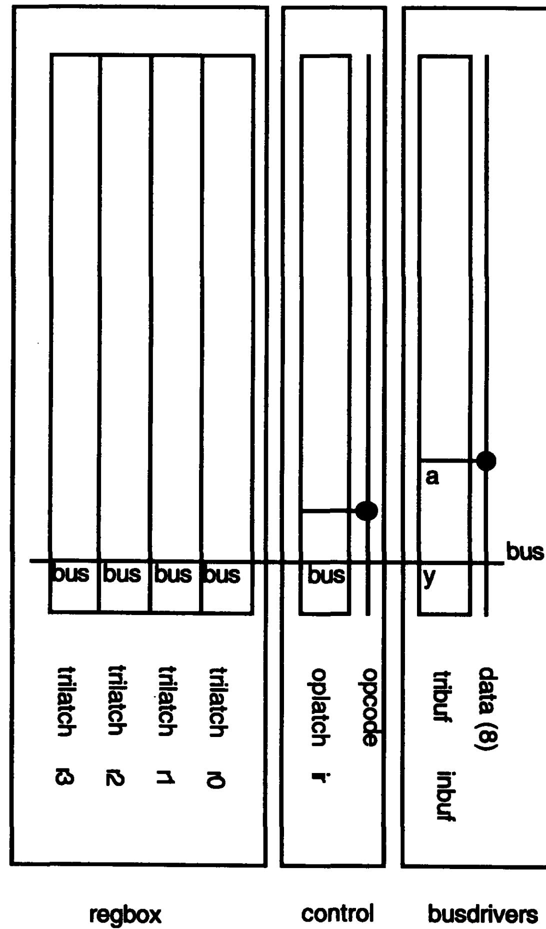


The figure above illustrates the arrangement of cells in one bitslice of the datapath. Metal 2 bitlines run horizontally; there are ten metal tracks available. Vertical power, gnd, and control inputs to the various cells are not shown.

3.3 regbox

The regbox contains the four general purpose registers. They are implemented as transparent latches, open while the clock is high. Schematics of this module are located in the appendix.

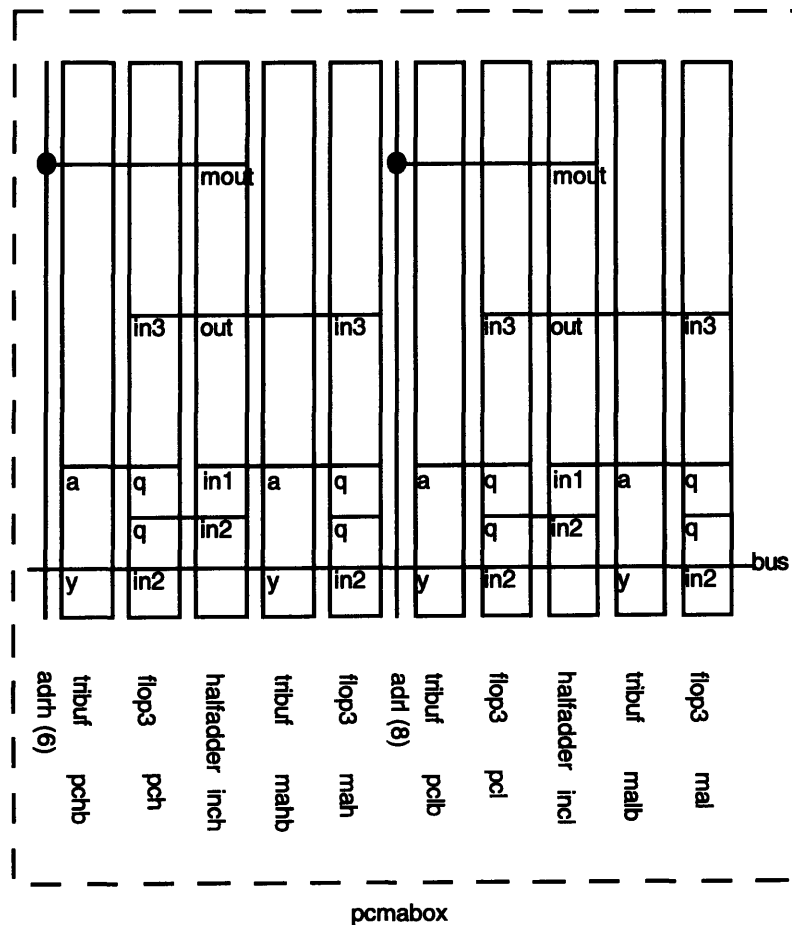
The figure below shows the arrangement of cells in one bitslice of the regbox, along with the adjacent opcode register and data bus tristate drivers / receivers.



3.4 pcmabox

The pcmabox contains some slightly messier logic used for the program counter and memory address registers. In addition to being read and written from the bus, the PC and MA registers must be able to be driven to the address pins and to be incremented. A multiplexor is used to choose either PC or MA to drive the 16 address pins. This address is also fed into a 16 bit incrementor, which may be selected to be loaded back into the registers. The appendix contains schematics of the pcmabox.

The figure below shows the usage of metal 2 tracks and the connections between cells for the PCH and MAH registers. An identical copy is required for the PCL and MAL registers.



pcmabox

3.5 control

The Unintel Sexium is controlled by a large FSM that drives the various select and enable lines in the datapath to sequence the proper set of operations to execute instructions. The FSM has fourteen bits of state.

ir<7:0>	instruction register, holds current opcode
s<2:0>	state counter, indicates current step in multi-step instructions
f	fetch, indicates fetching next instruction into ir
carry	carry bit from ALU, used in BRA command to add 16 bit numbers
neg	negative bit from ALU, used in BRA command for sign extension

In addition to the state, the control FSM takes four other inputs:

reset	reset processor, clear PC to 0 and fetch first instruction
zero	zero out from ALU (used for SKZ)
cout	carry out from ALU (used for BRA)
negative	MSB of Z output is set, indicating negative 2's complement value

Note: cout and negative are just conditionally latched into the carry and neg flip-flops; they do not feed into the FSM directly.

The FSM produces many outputs, summarized below:

ALU Control

xsel<3:0>	controls the xmux selecting the input to the ALU 0001 = A 0010 = TMP 0100 = $00 1000 = $FF
ysel	controls the ymux selecting the input to the ALU 0 = bus 1 = $00
zsel<4:0>	controls the zmux selecting the output of the ALU 00001 = adder 00010 = ander 00100 = neger 01000 = shifter 10000 = flags
alu_bus_en	enables tristate of Z output to bus
rollflag	controls bit 7 of shifter output 0 = shift (load 0 into bit 7) 1 = roll (load old bit 0 into bit 7)
ldacc	loads the A register from the Z output
ldtmp	loads the TMP register from the Z output
cin	sets cin on adder

REG Control

reg_wr<3:0>	write general purpose register from bus
reg_rd<3:0>	read general purpose register onto bus

PCMA Control

pcma_rd<3:0>	read PC / MA onto bus
pcma_in<7:0>	write PC / MA from bus or incrementor 00 = keep old value 01 = bus 10 = incrementor
pcma_select	choose PC or MA to drive address pins 0 = PC 1 = MA

I/O Control

bus_out	drive bus onto data pins
rwbar	read or write external memory 1 = read 0 = write (pulsed low during second half of cycle)

FSM Control

clrs	clear S counter
newf	set f bit
latchcarry	latch CARRY bit
latchneg	latch NEG bit

The following microcode is used to implement each of the instructions. All operations listed on one line occur in parallel during one clock and results are not visible until the next rising clock edge.

Reset (RESET = 1)

S000: PC ← 0; S ← 0; F ← 1;

Fetch (F = 1)

S000: IR ← M[PC]; S ← 0; F ← 0; PC ← PC + 1

LDA / LDI

S000: A ← M[MA]; (if inc MA ← MA + 1); S ← 0; F ← 1;

STA / STI

S000: M[MA] ← A; (if inc MA ← MA + 1); S ← 0; F ← 1;

LDM

S000: A ← M[PC]; PC ← PC + 1; S ← 0; F ← 1;

2-Op (e.g. ADD, AND, TST, GET) or 1-Op (e.g. NOT, SHR, ROR)

S000: A ← A op REG; S ← 0; F ← 1;

PUT

S000: REG ← A; S ← 0; F ← 1;

JMP

S000: TMP ← M[PC]; PC ← PC + 1;

S001: PCL ← M[PC];

S010: PCH ← TMP; S ← 0; F ← 1;

BRA

S000: TMP ← M[PC];

S001: TMP ← PCL + TMP;
S010: PCL ← TMP;
S011: TMP ← PCH + CARRY - NEG
S100: PCH ← TMP;

CAL

S000: TMP ← M[PC]; PC ← PC + 1;
S001: R1 ←M[PC]; PC ←PC + 1;
S010: R0 ← PCL;
S011: PCL ← R1;
S100: R1 ← PCH;
S101: PCH ← TMP; S ← 0; F ← 1;

RET

S000: PCL ← R0
S001: PCH ← R1; S ← 0; F ← 1;

SKZ

S000: (if A = 0 PC ← PC +1)
S001: PC ← PC + 1; s ← 0; (if A = 0) then F ← 1;
else PC ← PC + 1; F ← 1;

The logic equations for the FSM are listed in the appendix.

In addition, the control unit includes a 16 input multiplexor to test the internal signals on the external test pins. Test<3:0> selects one of the following signals to be muxed out to the testout pin.

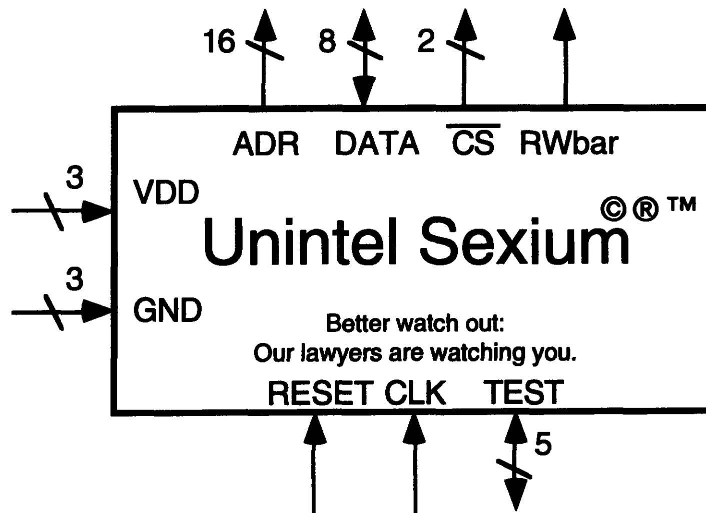
0000:	resetbar	0001:	ring osc.	0010:	T flop	0011:	F bit
0100:	s<0>	0101:	bus<0>	0110:	bus<1>	0111:	bus<2>
1000:	bus<3>	1001:	bus<4>	1010:	bus<5>	1011:	bus<6>
1100:	bus<7>	1101:	ysel	1110:	ir<0>	1111:	neg

4 Sexium External Interface

The Unintel Sexium is not a very good computer by itself. It needs to be able to communicate with the outside world and read and write memory to do anything interesting. This section describes the pinout of the Sexium processor and shows a schematic of the processor hooked up to memory.

4.1 Pinout

Like the 6502 (in the Apple II and Commodore computers) and the Z80 microprocessor (in the hideous old CP/M microcomputers[5]), the Sexium is packaged in a 40 pin DIP (Dual Inline Package). This is less expensive than larger packages and should be sufficient for our needs. The pinout diagram is shown below.



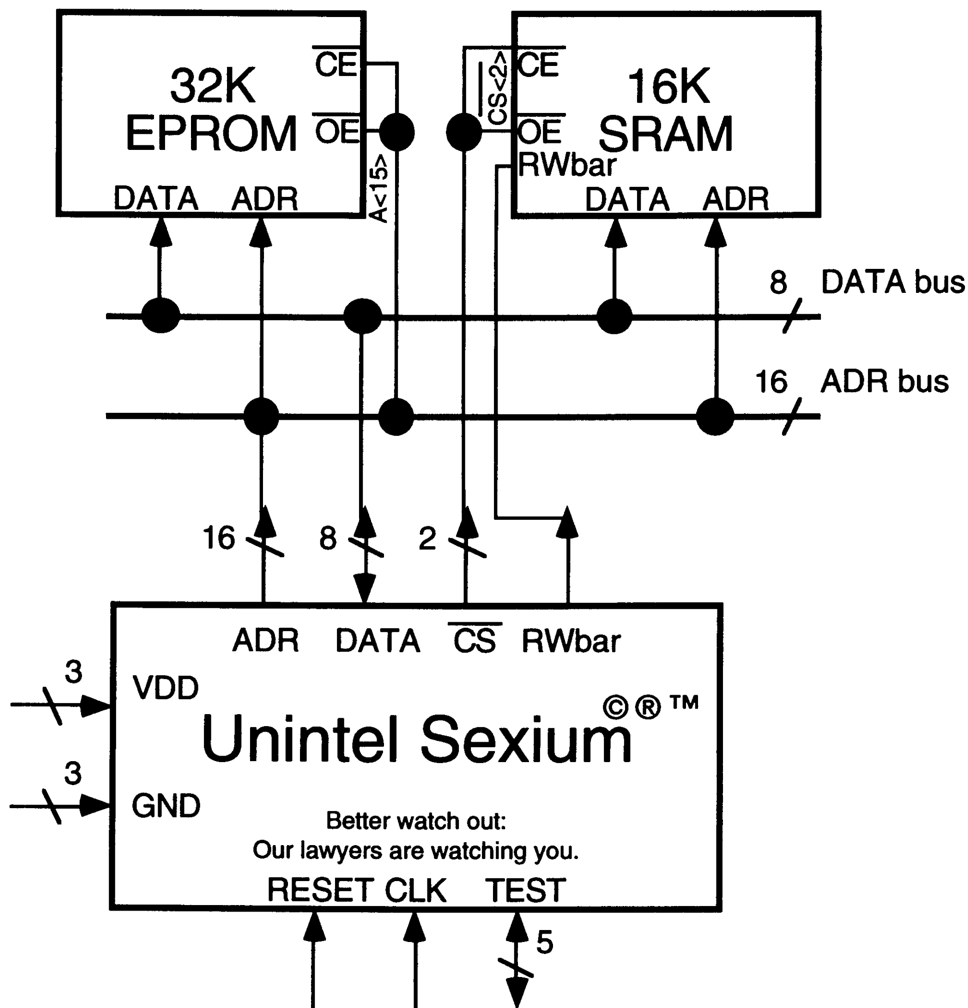
Getting a good, stable VDD and GND is always difficult, so three pins are devoted to each. RESET initializes the Sexium, clearing the PC to zero and making it start execution of instruction 0 in memory. CLK is the clock input. ADR is a 16 bit address bus, indicating which memory location to read or write. DATA is an 8 bit bidirectional data bus; on reads, data flows from external chips into the Sexium; on writes, data flows out of the Sexium to memory. The CS lines are the decoded version of the two most significant address bits; they are used to select one of the three banks of the address

[5]CP/M was an early user-unfriendly operating system that ran on Z80 microprocessors. According to popular legend, CP/M stood for Conspiracy to Protect the Ministry.

space. RWbar is high for a read, but is pulsed low during the second half of the clock cycle for a write.

4.2 Complete System Schematic

The following schematic shows an Unintel Sexium connected to an EPROM, and an SRAM. The other CS line could drive a multiplexor to decode I/O devices, such as A/D or D/A converters, joysticks, keyboards, serial printers, etc.



5 Simulator Reference

The MSIM program is an assembler and simulator for the Unintel Sexium microprocessor. It allows the user to assemble a program (convert it from instruction names into 1's and 0's) and to simulate the program for easy debugging. It also produces a file with the program in binary form that can be burnt into an EPROM.

5.1 Assembly Language Syntax

Sexium assembly language supports five kinds of information: comments, instructions, label definitions, labels, and constants. The assembler is case-insensitive; it does not care if one uses upper or lower case.

Comments are just for the human reader; they are ignored by the assembler. Any text following a # sign is treated as a comment and ignored. It is a good idea to carefully comment your code, including your name, the date of creation, what registers and memory locations are used for what data, and what sneaky hacks you use in your program. This will help other people (and maybe eve you!) understand your code.

In addition to the standard Sexium instructions (ADD, AND, NOT, SHR, ROR, PUT, TST, LDA, LDI, LDM, STA, STI, JMP, BRA, CAL, RTN, and SKZ), the MSIM program supports the BRK (**break**) instruction. The BRK instruction tells the simulator to stop and print out the current state of all of the registers. This is useful to end your program and to help debug.

Label definitions are lines that just have a label (any piece of text up to 24 characters) followed by a colon (:). They are used to mark points in your code where branches or jumps should go. It is illegal to put an instruction on the same line as a label definition. Each label can be defined only once in the program.

Labels refer to label definitions elsewhere in the code. They are used with the JMP, CAL, or BRA instructions. Constants are hexadecimal numbers preceded by a $ sign; they are primarily for LDM instructions, but can be used instead of a label on JMP, CAL, or BRA instructions.

5.2 Programming Example

The following program is an example of code written in Sexium assembly language, using labels, comments, and a variety of instructions.

```
# Fib.asm
#
# Written 1/14/93 by David Harris
#
# This program computes the Nth
# fibbonacci numbers, where N is the
# value stored in R0 (1 <= N <= 13)
#
# Fib. numbers 0-6 are 0, 1, 1, 2, 3, 5, 8
#
# Some C code to do this would be:
##    int fib(int N)
##    {
##          int cur, prev, next;
##
##          prev = 0; /* Zeroth fib number */
##          cur = 1;  /* First fib number */
##          N=N-1;
##          while (N != 0) {
##                next = prev+cur;
##                prev = cur;
##                cur = next;
##                N = N-1;
##          }
##          return cur;
##    }
#
# Register Use
# R0:  Number of numbers remaining to compute
# R1:  Temporary storage
# R2:  Current fib number
# R3:  Previous fib number

      LDM $06             # Compute 6th fib number
      PUT R0
Start:
      LDM $00             # 0, the 0th fib number
      PUT R3              # store 0th fib number in R3
      LDM $01             # 1, the 1st fib number
      PUT R2              # store first fib number in R2
      BRA Check           # Check to see if we are done
Loop:
      GET R2              # Compute next = R2 + R3
      ADD R3
      PUT R1
      GET R2              # Make previous number = current
      PUT R3
      GET R1              # And current number = next
      PUT R2
Check:
      LDM $FF             # = -1
```

```
ADD R0              # Computer R0 <= R0-1
PUT R0
SKZ                 # Have we counted down to 0?
BRA Loop            # No:  Continue computing
BRK                 # Yes: All done!
```

Two longer examples are available on-line. Mult.asm, by Matthew Sakai, multiplies the numbers stored in R2 and R3 and produces a 16 bit result. It demonstrates loops and 16 bit addition. Regress.asm tests all of the instructions in the Sexium instruction set. It is useful for verifying that the simulator works correctly; it is also a useful test vector for a schematic-level model of the microprocessor. Both of these files are in the /home/cva2/6090user/tools/msim directory.

5.3 Using MSIM

To use MSIM, create your program and save it with a name like prog.asm. The .asm suffix is required. Then run MSIM using the command:

```
msim prog
```

MSIM will first assemble your program. If it catches any mistakes, correct them. When it assembles correctly, you will be presented with a menu of choices:

```
[G]o  [T]race  [S]tep  [M]emory  [B]reakpoint  [R]eset  [Q]uit:
```

Press the first letter of the command you want. Go runs the program until it encounters either a BRK instruction or the end of the code. Trace is similar, but prints out the registers and instruction being executed at each instruction. Step executes a single instruction, then prints out the registers and waits for your next command. This is useful for debugging your code. Memory allows you to view memory. It brings up a new menu with the choices:

```
[D]isassemble  [R]ead  [W]rite:
```

You can use the disassemble option to view part of your code, the read option to print out part of memory in hexadecimal, and the write option to modify a byte in memory.

Breakpoint allows you to set a breakpoint in your program. It replaces the instruction of your choice with a BRK, so that you can run or trace to that point, then stop. To get the

original instruction back, set a breakpoint at instruction -1. Finally, Reset clears all of the registers to 0 to restart the simulation.

5.4 Creating EPROMs

To burn an EPROM with the Sexium program you have written, follow this sequence of steps:

1) Assemble your program (e.g. prog.asm)
2) Copy the prog.dat file to Athena using ftp
3) Convert the file format by attaching the 6.111 locker and running
   dat2ntl prog.dat > prog.ntl
4) Burn the EPROM in the 6.111 lab using the promdio command at a programmer

Appendix A: Sexium Schematics

This appendix contains schematics for the Unintel Sexium.

A1: Datapath Block Diagram

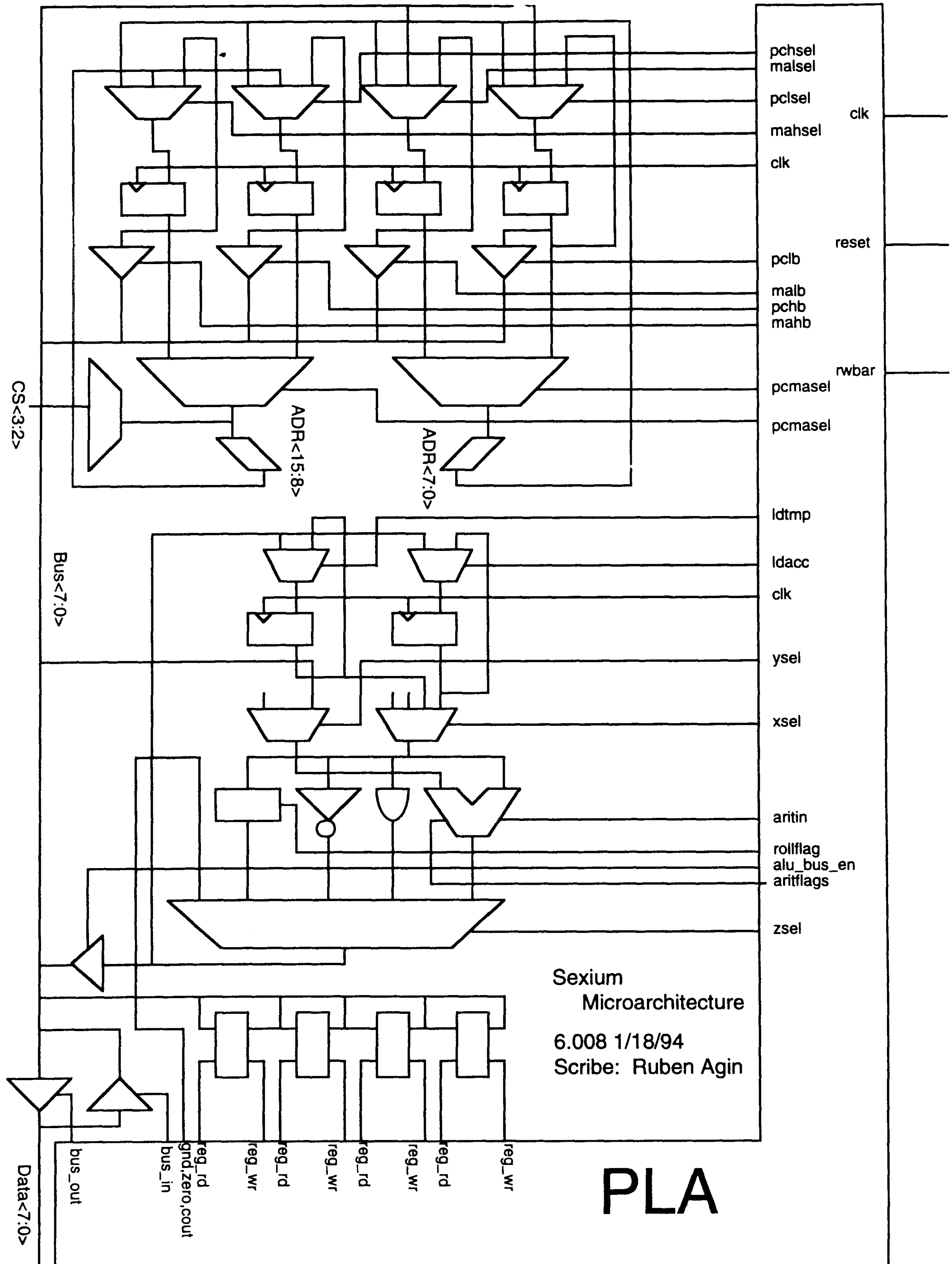
A2: sexium

A3: alubox

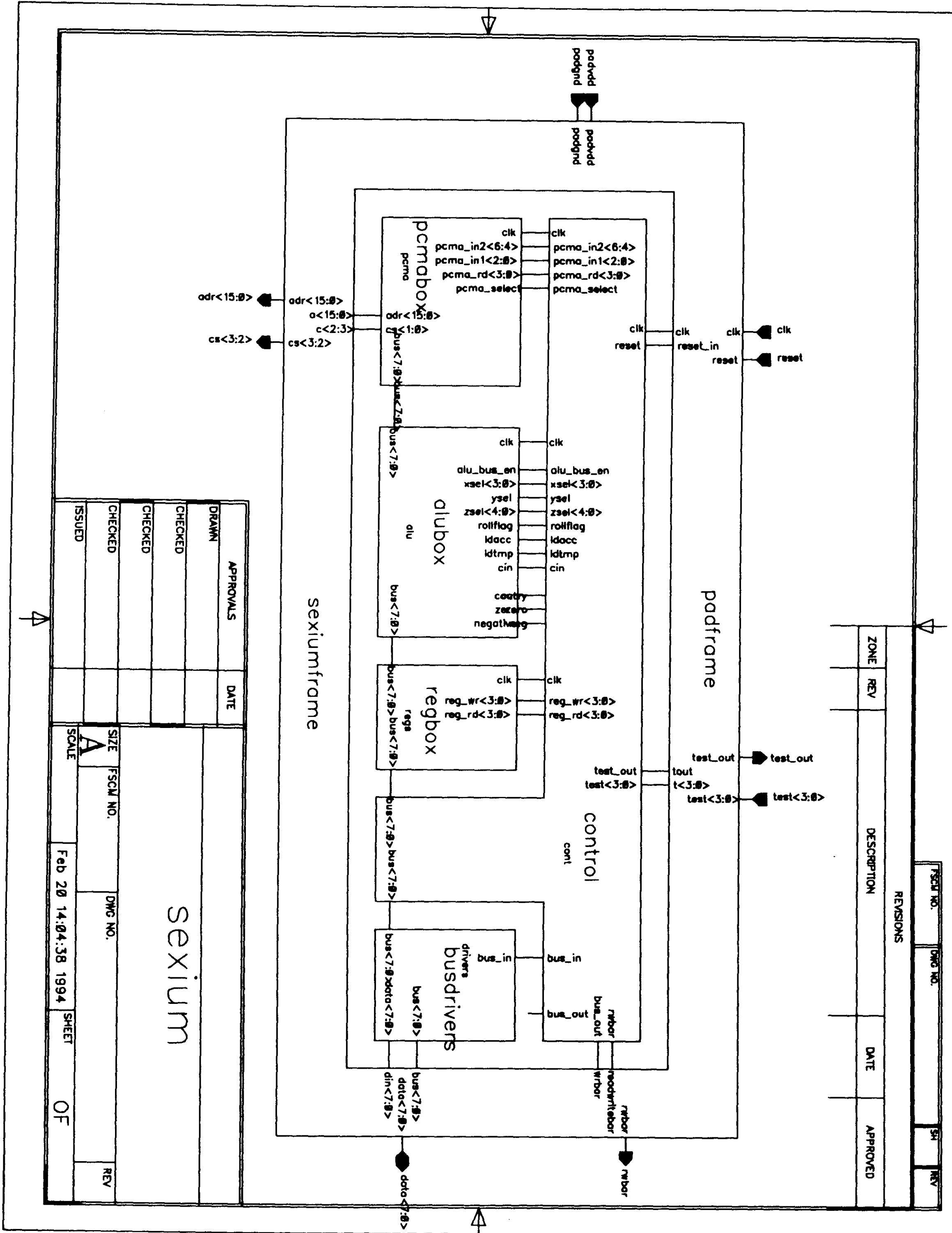
A4: pcmabox

A5: regbox

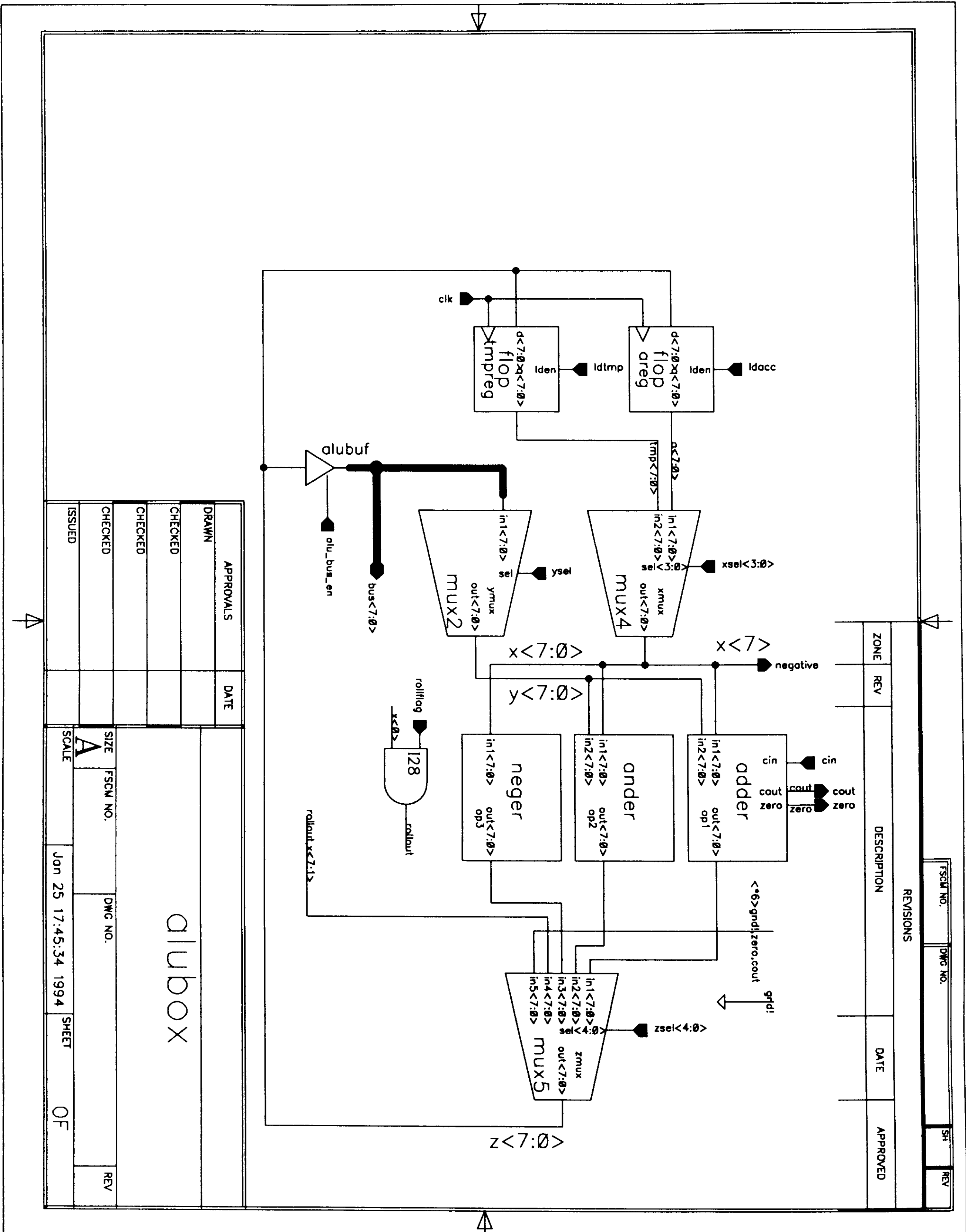
A1:datapath



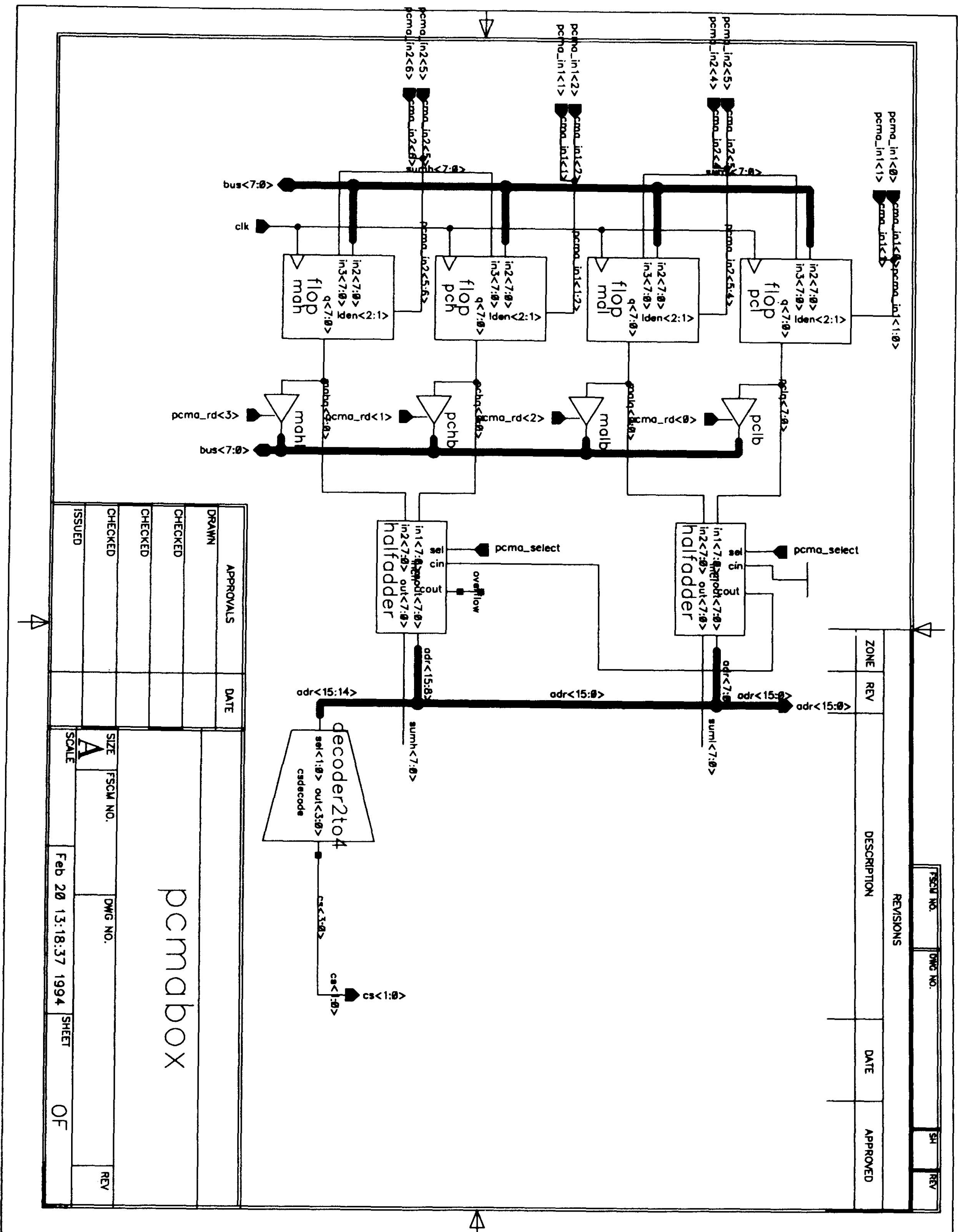
A2: sexium



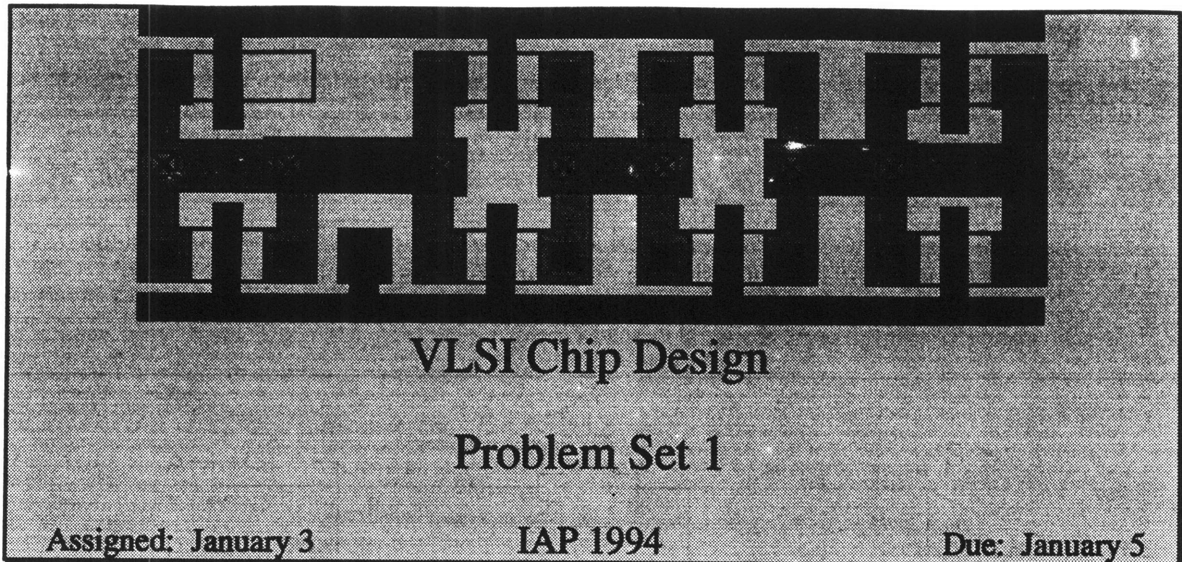
A3: alubox



A4: pcmabox



A5: regbox

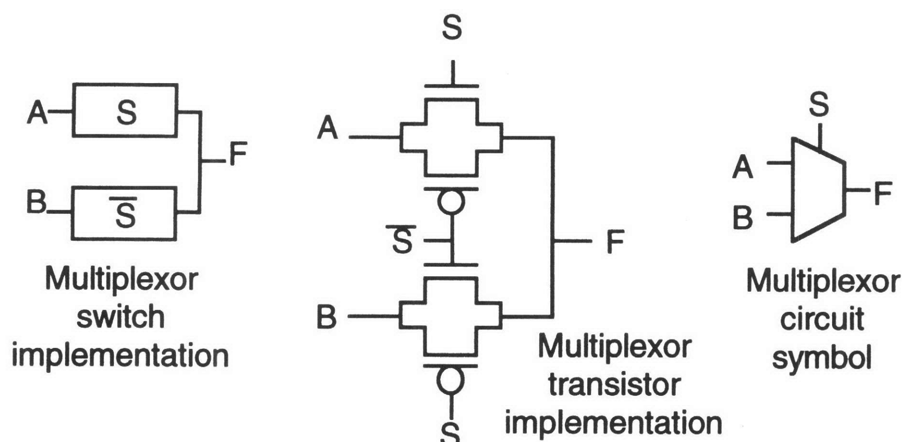


Try to do these questions in order. They tend to build concepts. You may do a design in a later section that is more efficient than your earlier design.

1) **Warm-up**: Draw NOT, NAND, NOR, AND, and OR gates using switch logic. Do the thing using MOSFETs.

2) Design an XOR gate using switches. How many switches do you need? Repeat with MOSFETs. How many transistors do you need? Also, design an XOR gate using logic gates. How many total switches would the logic gate implementation require?

3) Consider the two-input multiplexor shown below. The multiplexor (also called a MUX) has two data inputs, A and B, a control input S, and an output F. If S = 1, F = A; otherwise F = B.



(a) Design a two-input multiplexor using logic gates. How many total switches would the logic gate implementation require? How many transistors? How do these numbers compare?

(b) Design a four-input multiplexor using switches. It should have four data inputs, A through D, and two control inputs, S1 and S2.

(c) Design 2-input NAND, NOR, AND, OR, and XOR gates using only the four-input multiplexor from part (b).

(d) Design 3-input NAND, NOR, AND, and OR gates using only a four-input multiplexor from part (b).

These problems should demonstrate that multiplexors are an extremely powerful and efficient way to design switch logic and are also frequently good for transistor logic. When possible, think in terms of multiplexors, not gates.

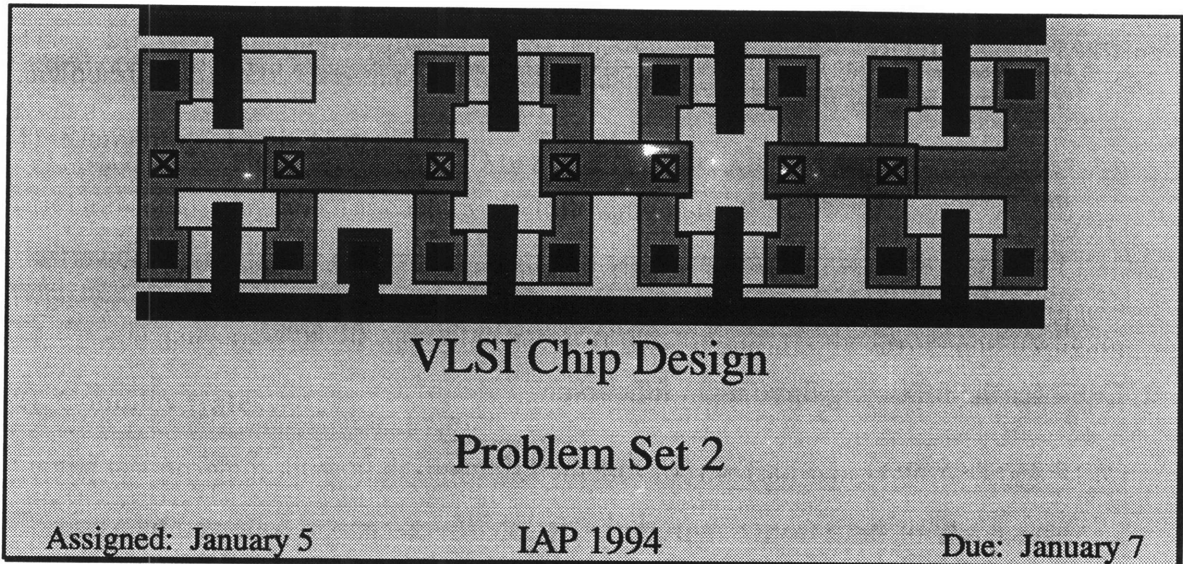
4) Consider the following three input function:

$$F = \overline{A \bullet (B + C)}$$

This could be built from a two-input OR gate (6 transistors) followed by a two-input NAND gate (4 transistors) for a total of 10 transistors. Implement this function with 6 transistors.

This gate, called OR-AND-INVERT, is one of many interesting gates efficiently implemented in CMOS. **Extra credit:** What other three and four input gates can be implemented with 6 and 8 transistors, respectively?

5) Secret Agent 6.007 has a terrible memory for dates and can never remember if a month has 31 days. Help him design a switch circuit that produces a 1 if and only if the month has 31 days. The circuit has four inputs, representing the month in binary (January = 0001, February = 0010, ..., December = 1100). **Design contest:** how few switches do you really need? How few transistors?



Most of these problems involve doing layout with the Cadence tools. Do the layout in the library of your name and hand in the names of the cells. On all layouts assume that the inputs are available in poly running vertically and that the outputs should emerge in vertical poly or metal2. The VDD and GND lines run horizontally in metal 1.

There will be an optional help session on Thursday at 2pm at the Edgerton Center. Bring any questions or problems that you have had using the Cadence tools.

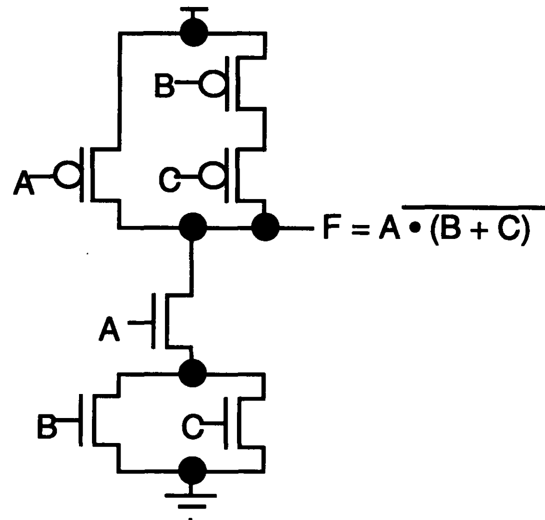
1) Elementary Logic Gates

(a) Draw stick diagrams for each of the following gates:
NOR, 3-input NAND, AND
(you may wish to draw the schematics first)

(b) Using Cadence, layout each of the cells from part (a). Run DRC and verify that the gates pass the design rule check. If they do not, fix the errors and try again.

2) Complex Gates

On the last problem set, you designed schematics for the following OR-AND-INVERT gate:



(a) Draw a stick diagram of this gate.
(b) Using Cadence, layout this gate and check it with DRC.

3) Multiplexor

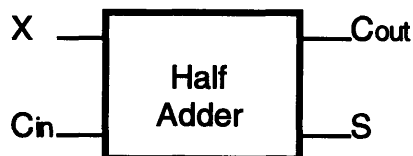
(a) Draw a stick diagram of a 2-input multiplexor. You may assume that you have both S and Sbar available.
(b) Layout the multiplexor in Cadence. Check with DRC.

4) Half Adder (Design Project)

The Bursar's office is having difficulty issuing tuition bills because the fees are getting larger than the largest number their computer system can handle. They have called in Secret Agent 6.007 to build a new system that can handle the larger numbers.

One of the key features of the system is the ability to add 1 to the current tuition amount. A monkey sits in the back of the office, repeatedly pressing the add 1 button. At the beginning of each year, the office checks how high the monkey has reached and charges that fee.

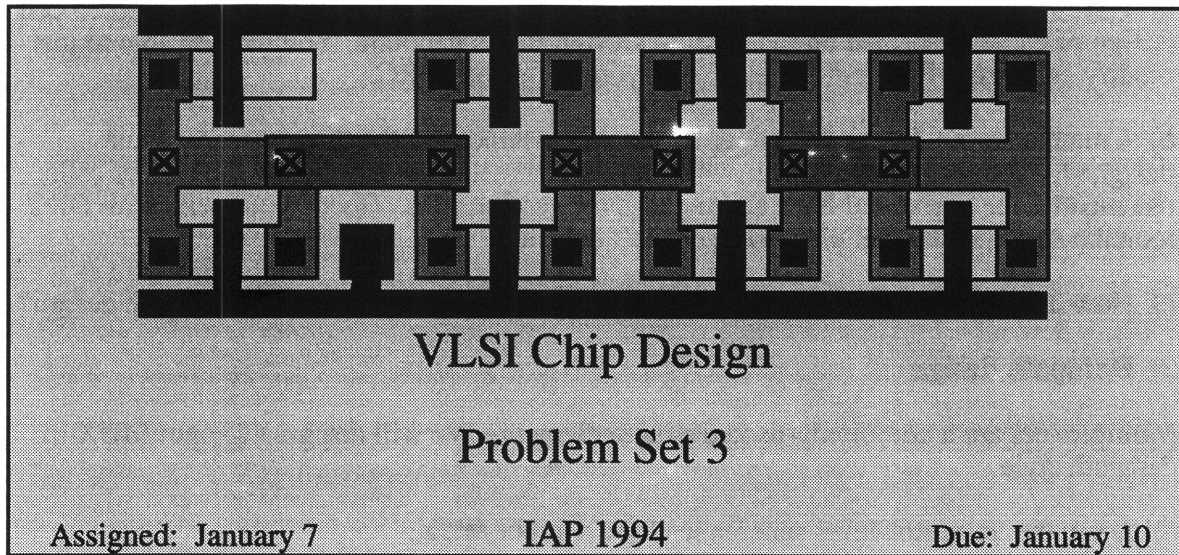
Secret Agent 6.007's first task is to design a half adder. The half adder takes two input, X and C_{in} (the carry in). It adds C_{in} to X and produces two bits of output, a sum S (the 1's digit) and a carry out C_{out} (the 2's digit).



(a) Write a truth table for the half adder. Draw a transistor-level schematic.
(b) Draw a stick diagram for the half adder.
(c) Layout the half adder in Cadence. Check with DRC.
Design Contest: How small can you make your layout?

5) Feedback

How long did you spend on this problem set? What did you like? What would you change?

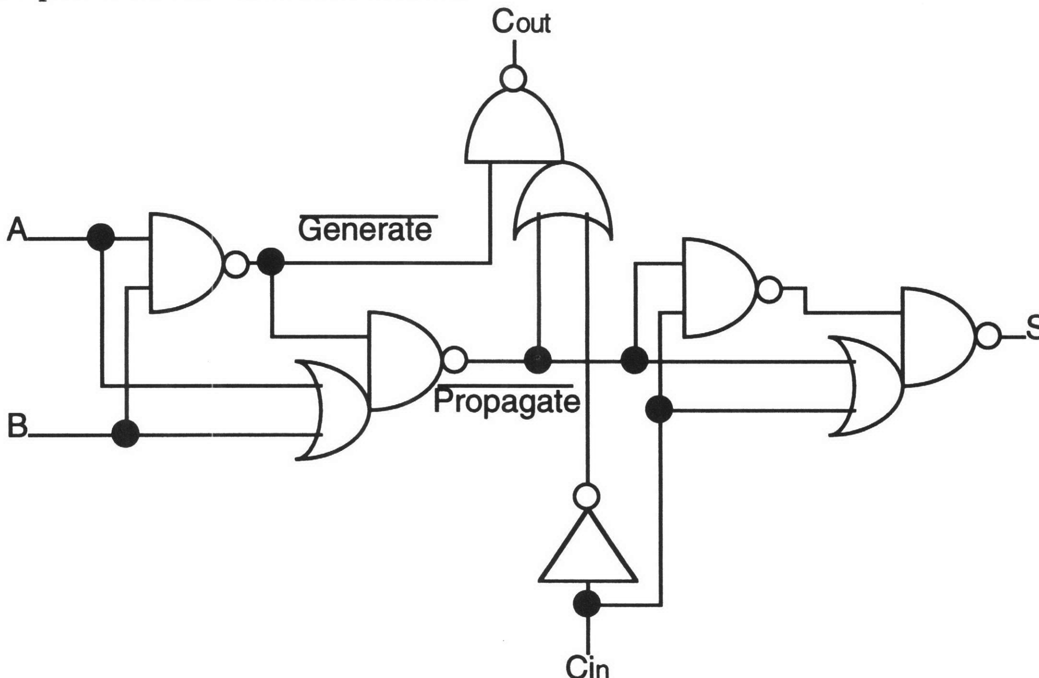


Save your solutions in your Cadence library and hand in the names of the cells.

1) Standard Cell Design

Standard cells are a relatively fast way to design a large, messy set of gates. However, they are seldom as efficient as carefully drawn custom logic. In this problem, we will compare the sizes of a full adder implemented as a datapath element and as standard cells.

In class, we looked at a datapath implementation of a full adder. It was 80 λ x 102 λ, for a total of 8160 λ^2. Below are schematics for an alternative full adder design, better for implementation with standard cells.



Note that the full adder is built from one inverter, two NAND gates, and three OR-AND-INVERT gates.

a) Layout a NOT gate, a NAND gate, and an OR-AND-INVERT gate using standard cell design rules, as described on the Sexium Design Rules handout. You may be able to just modify cells from Problem Set 2. Check your cells with DRC.

b) Create a layout for the full adder by placing instances of the three gates you just designed. Connect them together with routing channels above or below the row of cells; use metal 1 for horizontal lines and metal 2 for vertical runs. Check your cells with DRC to make sure that the cell placement doesn't conflict.

c) How large is your standard cell full adder? How does it compare to a datapath design?

2) Datapath design

Multiplexors are a commonly-used datapath element. We will design a 4-input MUX in datapath style.

a) Draw the transistor-level schematics for a 4-input MUX.

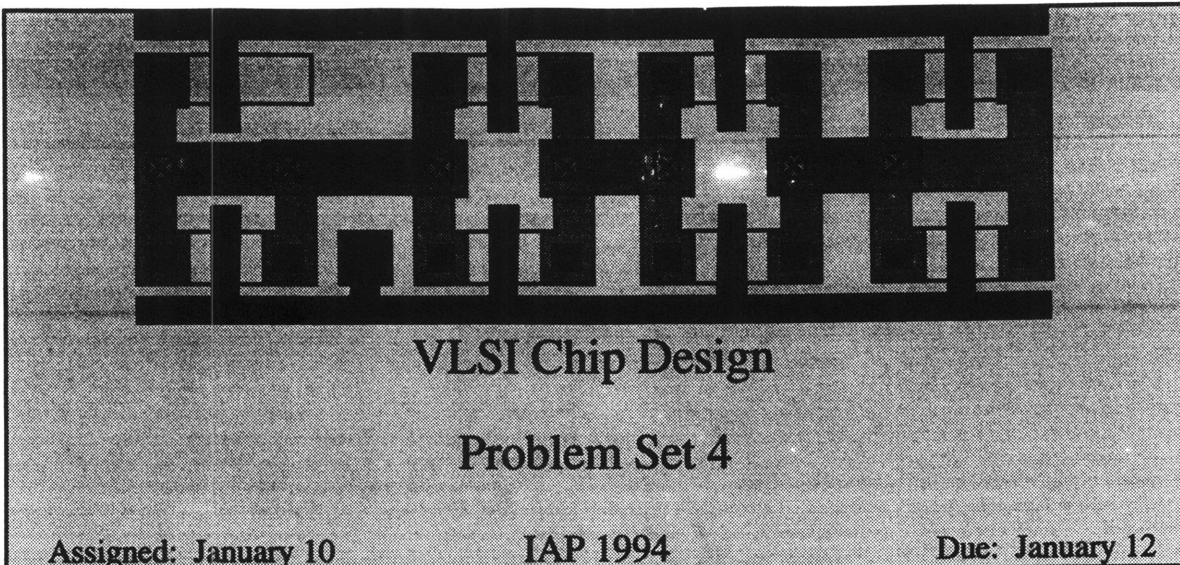
b) Draw a stick diagram of the MUX.

c) Using the datapath design style, lay out your MUX. It should have 5 metal 2 lines running vertically: inputs A-D and the output. It should have VDD, GND, and two select lines (to choose one of the four inputs) running horizontally in metal 1. Check your design with DRC.

d) Build an 8-bit datapath by placing 8 rotated instances of your multipelxor. The VDD, GND, and control lines should run vertically through all 8 instances; the 40 bit lines (5 from each of the 8 instances) should run horizontally.

3) Feedback

How long did you spend on this problem set? What did you like? What would you change?



1) Standard Cell Library

We will be needing a good standard cell library for future problem sets and for the Unintel Sexium project. Each person is assigned several cells to layout. Make your best effort to minimize area; we will be very tight on chip space. Refer to the Standard Cell Design Rules handout for dimensions. The best version of each cell will be selected for the 6.008 Standard Cell library.

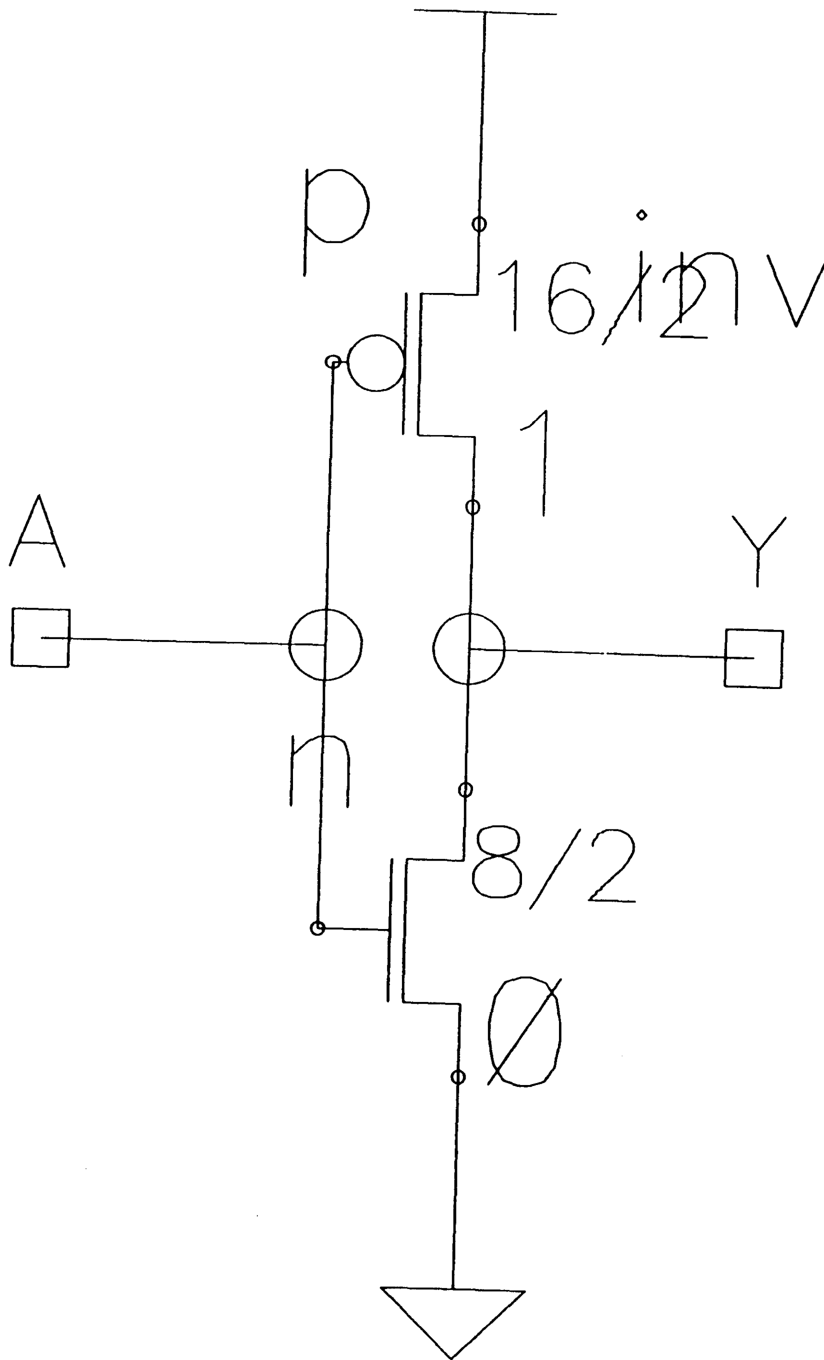
Cell Name	Designers
inv	Ethan, David
invtri	Dan, David
nand2	Ruben, Nehal
nand3	Ruben, Robert
nand4	Ruben, Matt
nor2	Aarati, Nehal
nor3	Aarati, Robert
nor4	Aarati, Matt
and2	Dan, David
or2	Dan, Ethan
xor2	Jeff, Ethan
aoi3	Jeff, Robert
mx2	Bayard, Nehal
mx3	Bayard, Jeff
mx4	Bayard, Jeff
latch	Matt, Ethan
flop	*** Extra Credit ***

2) 4-bit counter

In this problem, you will design and layout a 4 bit counter.

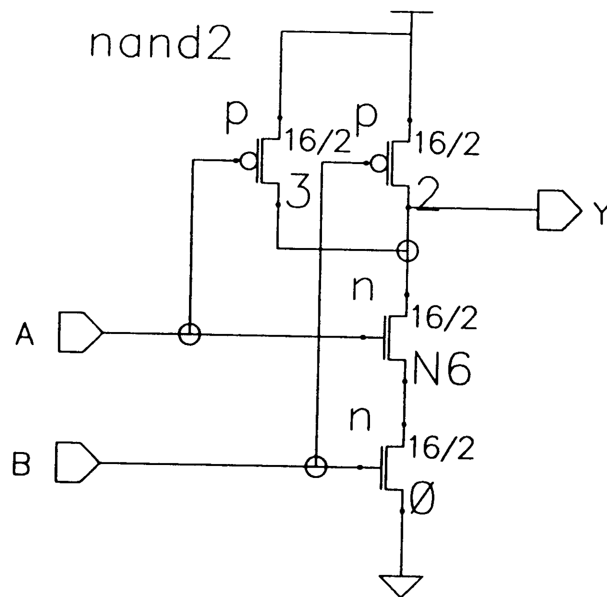
a) Sketch schematics for a 4 bit counter using 4 D flip-flops and whatever other combinational gates you need.
b) Using the flip-flop from the stdcells library and your standard cells from Problem Sets 3 and/or 4, layout a 4 bit counter. What is the area / bit for your counter?

inv schematic

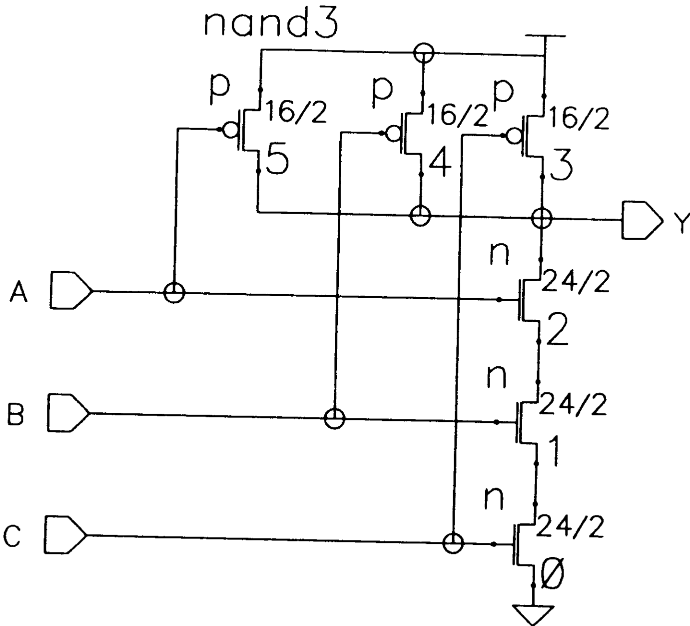


invtri schematic

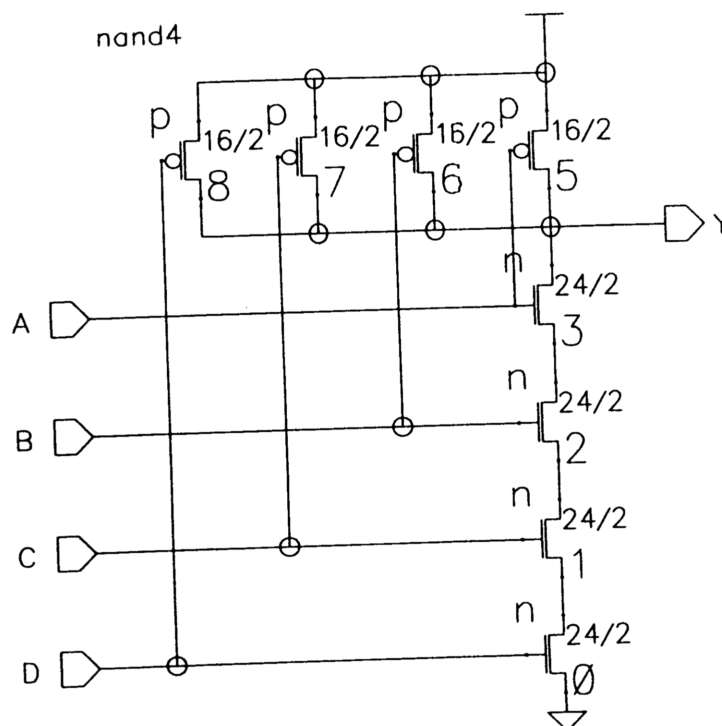
nand2 schematic



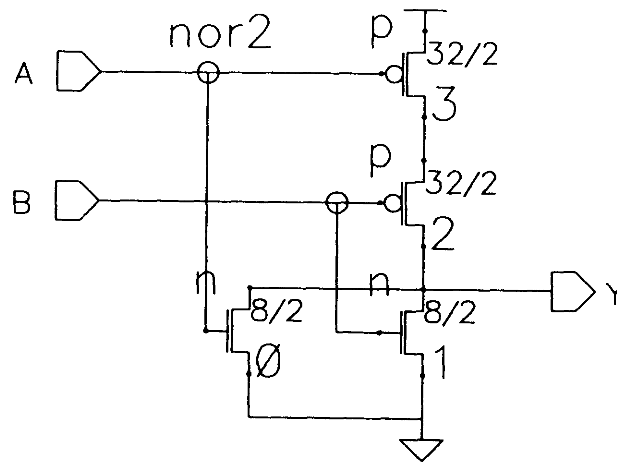
nand3 schematic



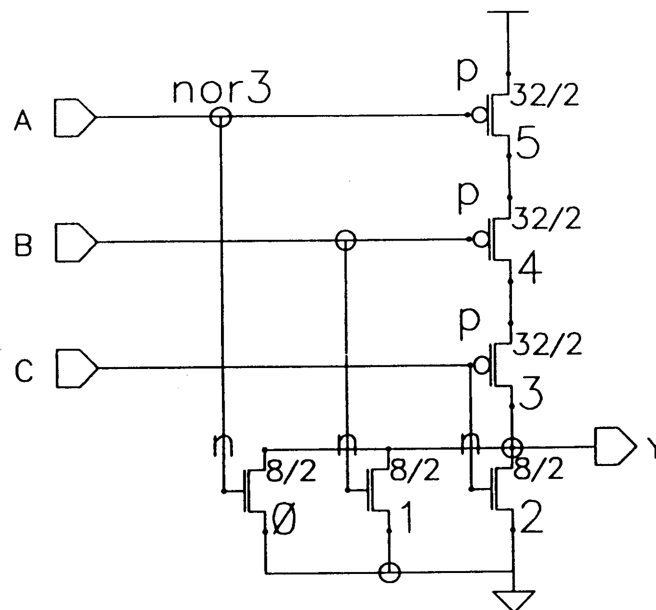
nand4 schematic



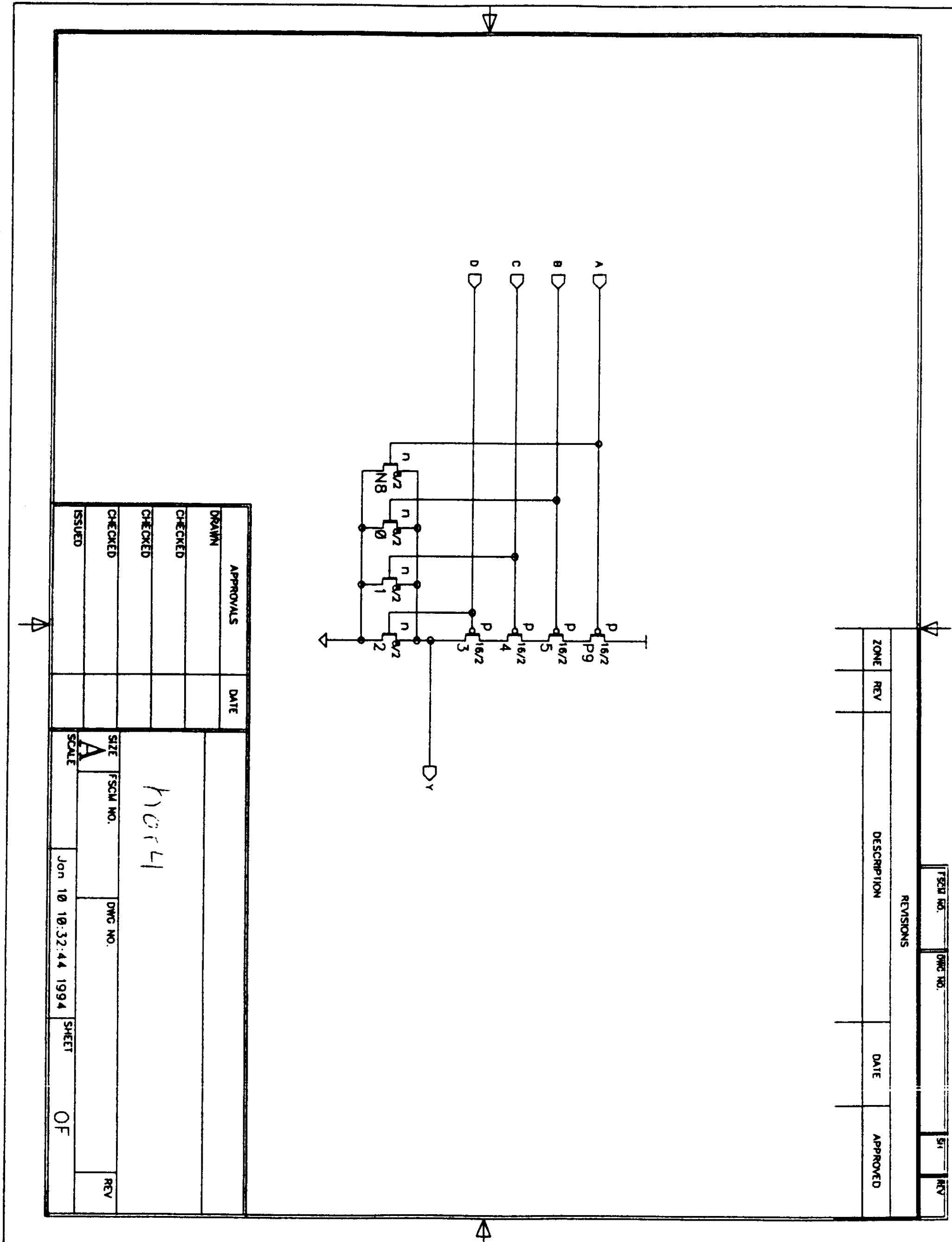
nor2 schematic



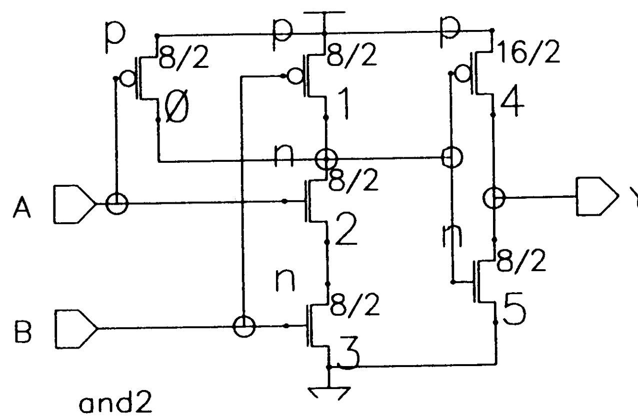
nor3 schematic



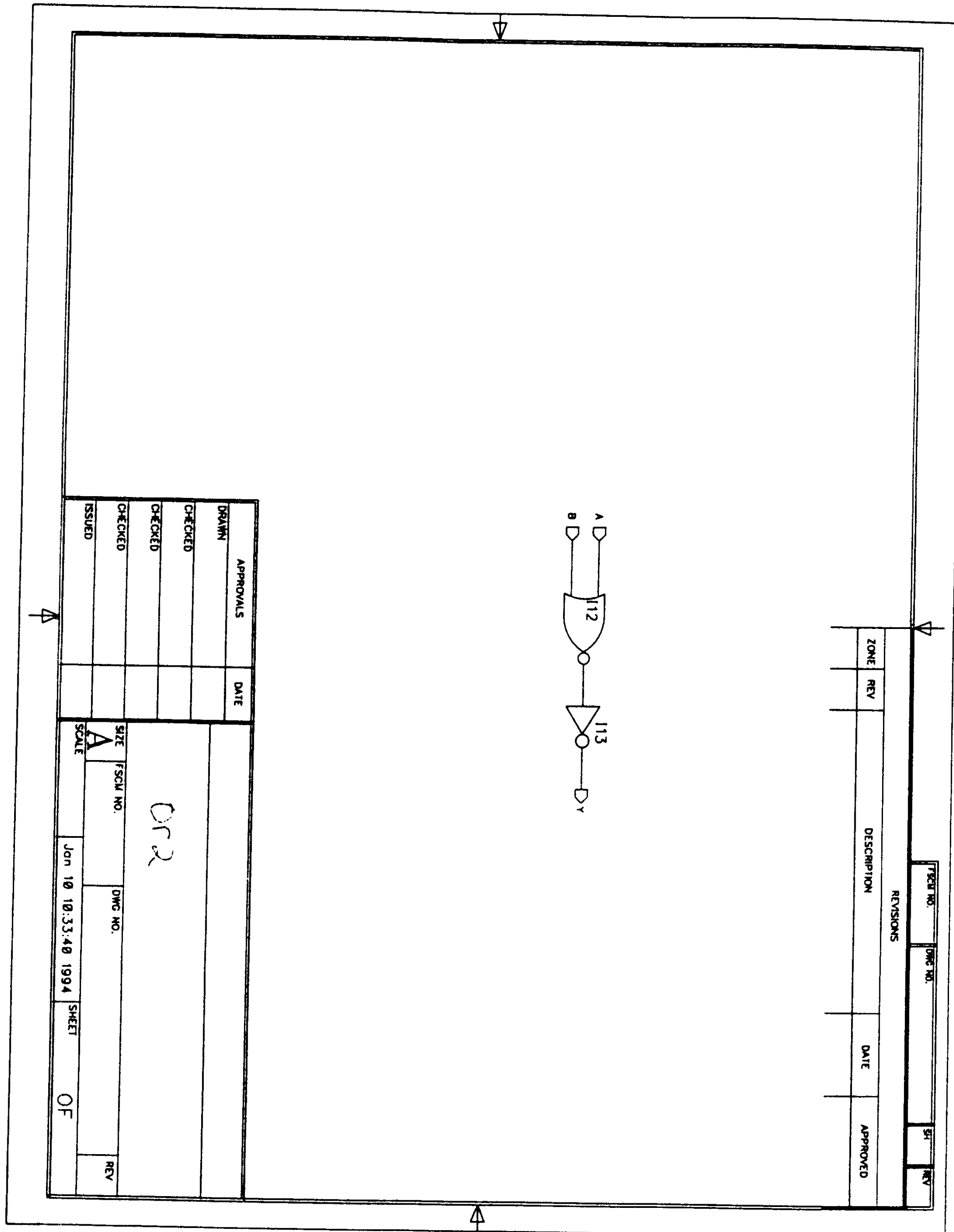
nor4 schematic



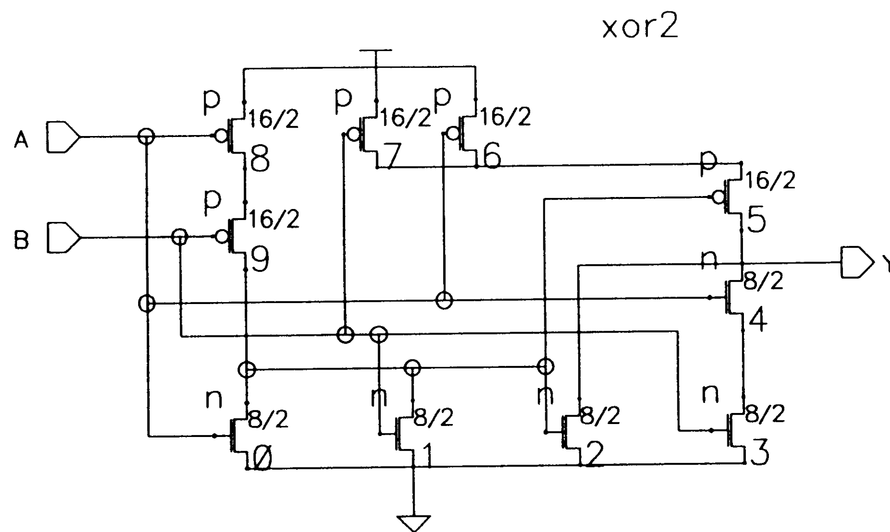
and2 schematic



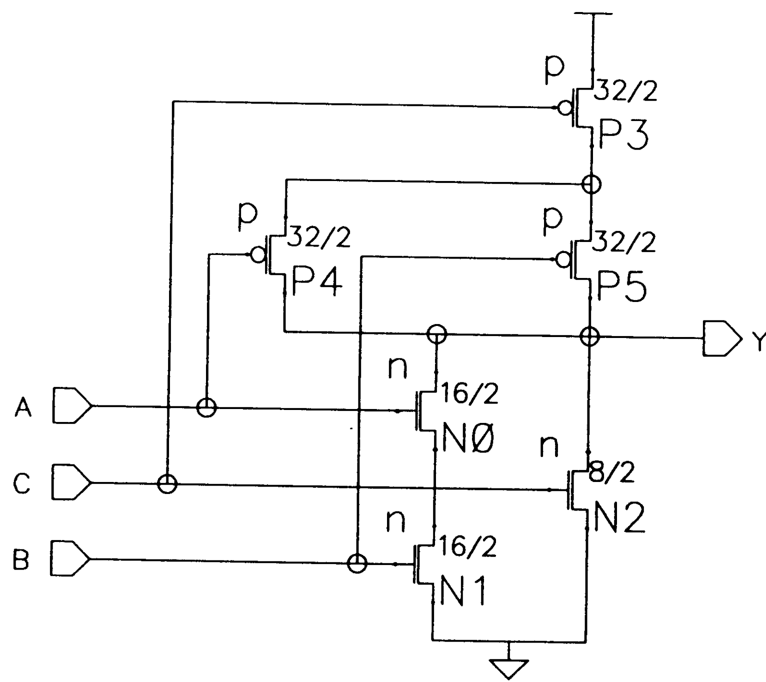
or2 schematic



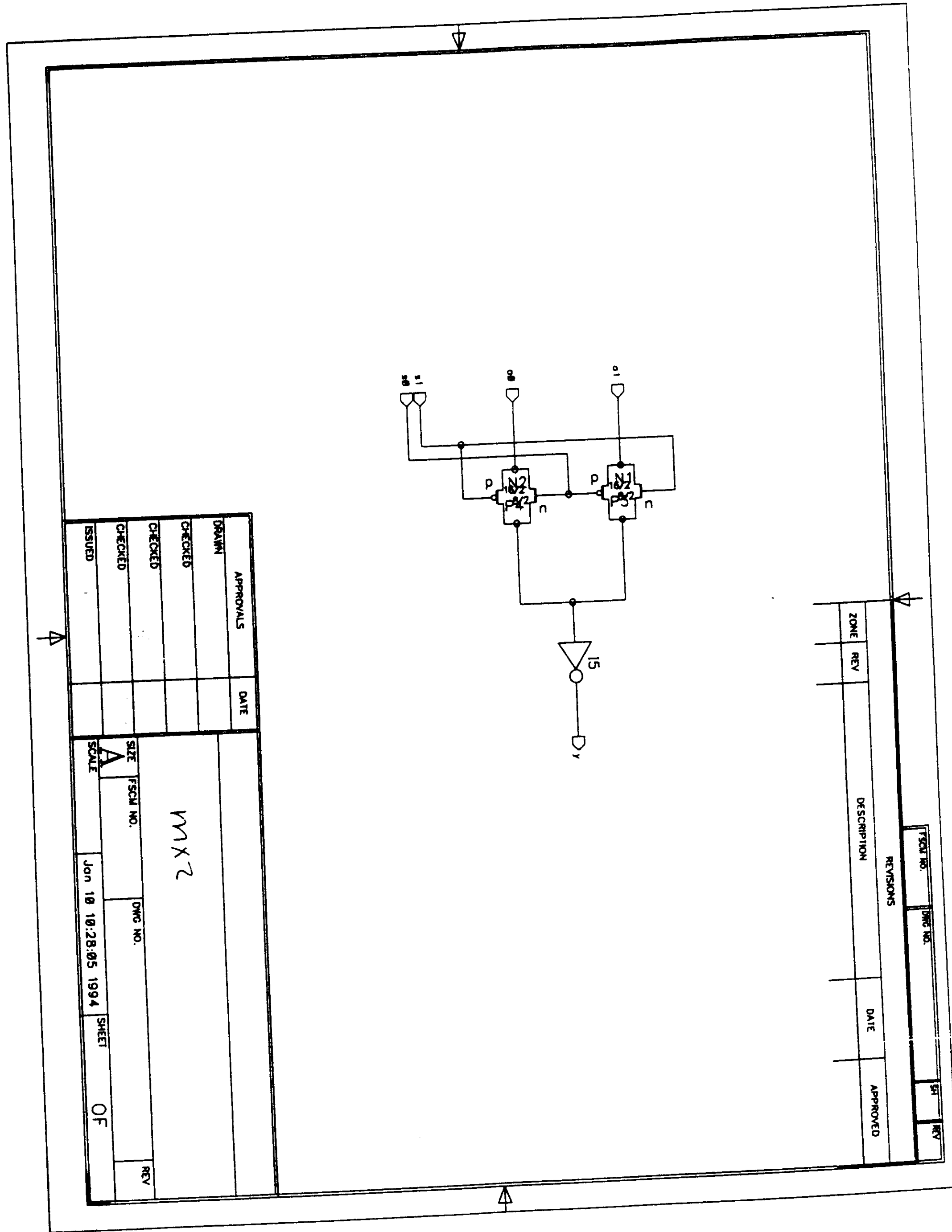
xor2 schematic



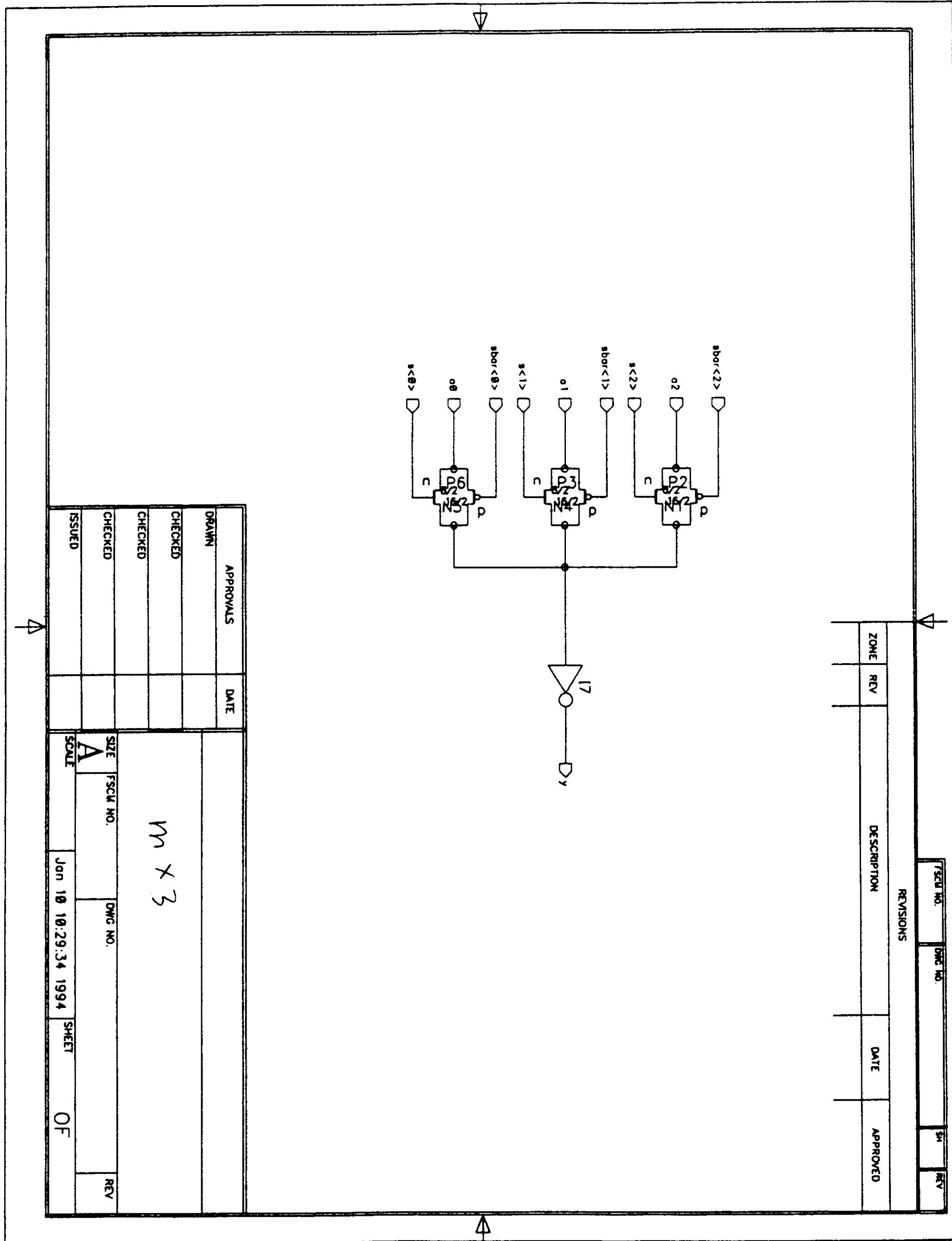
aoi3 schematic



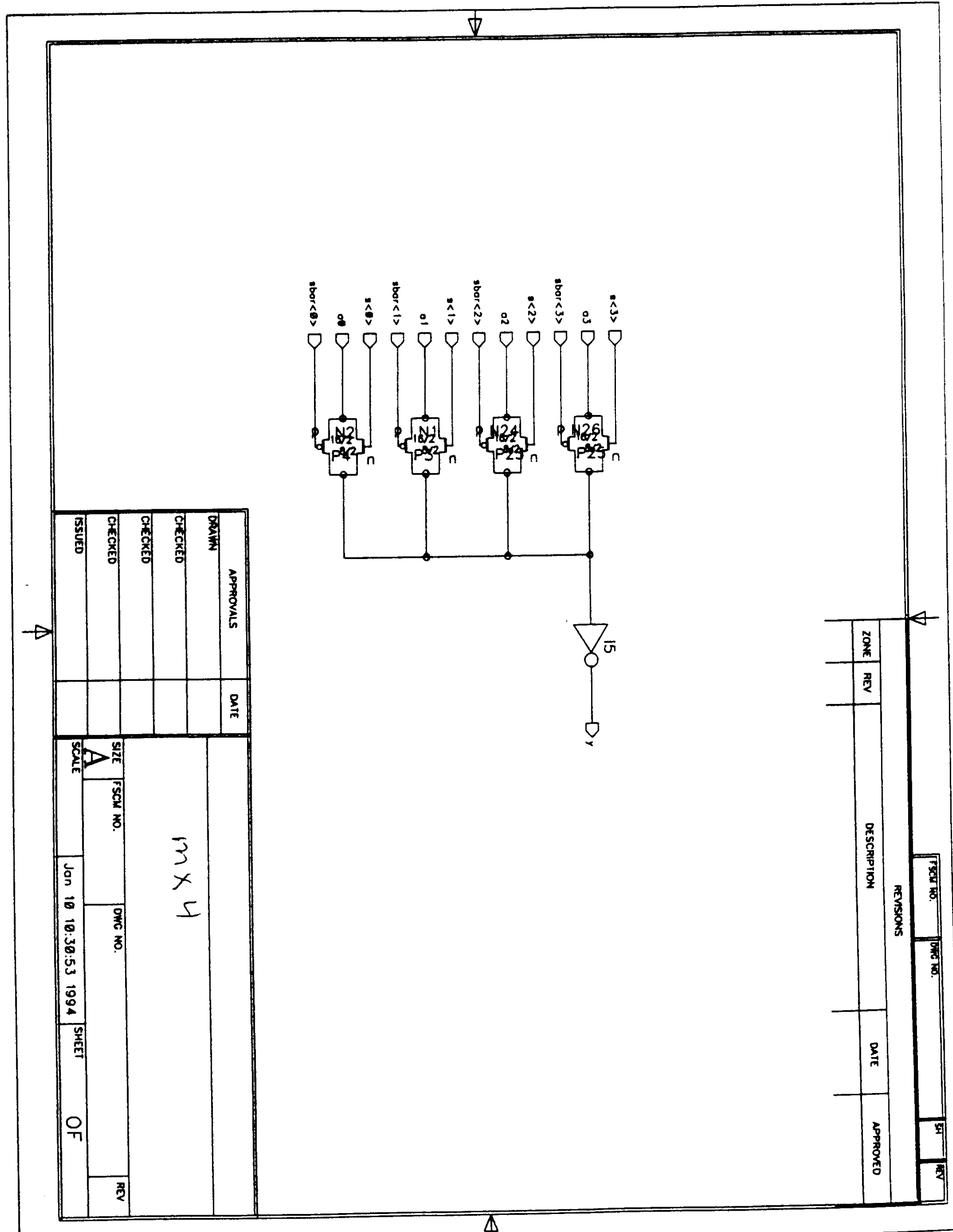
mx2 schematic



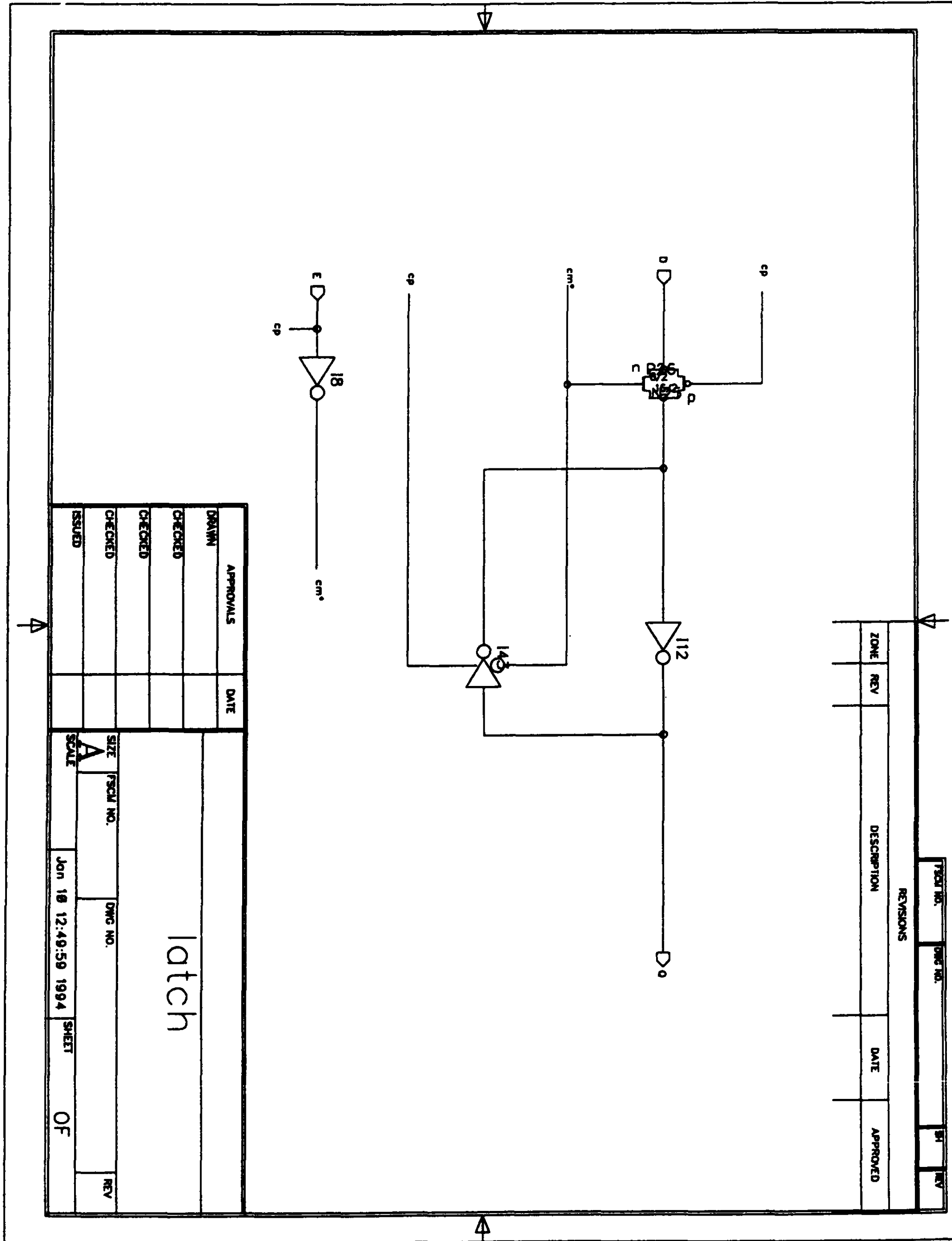
mx3 schematic



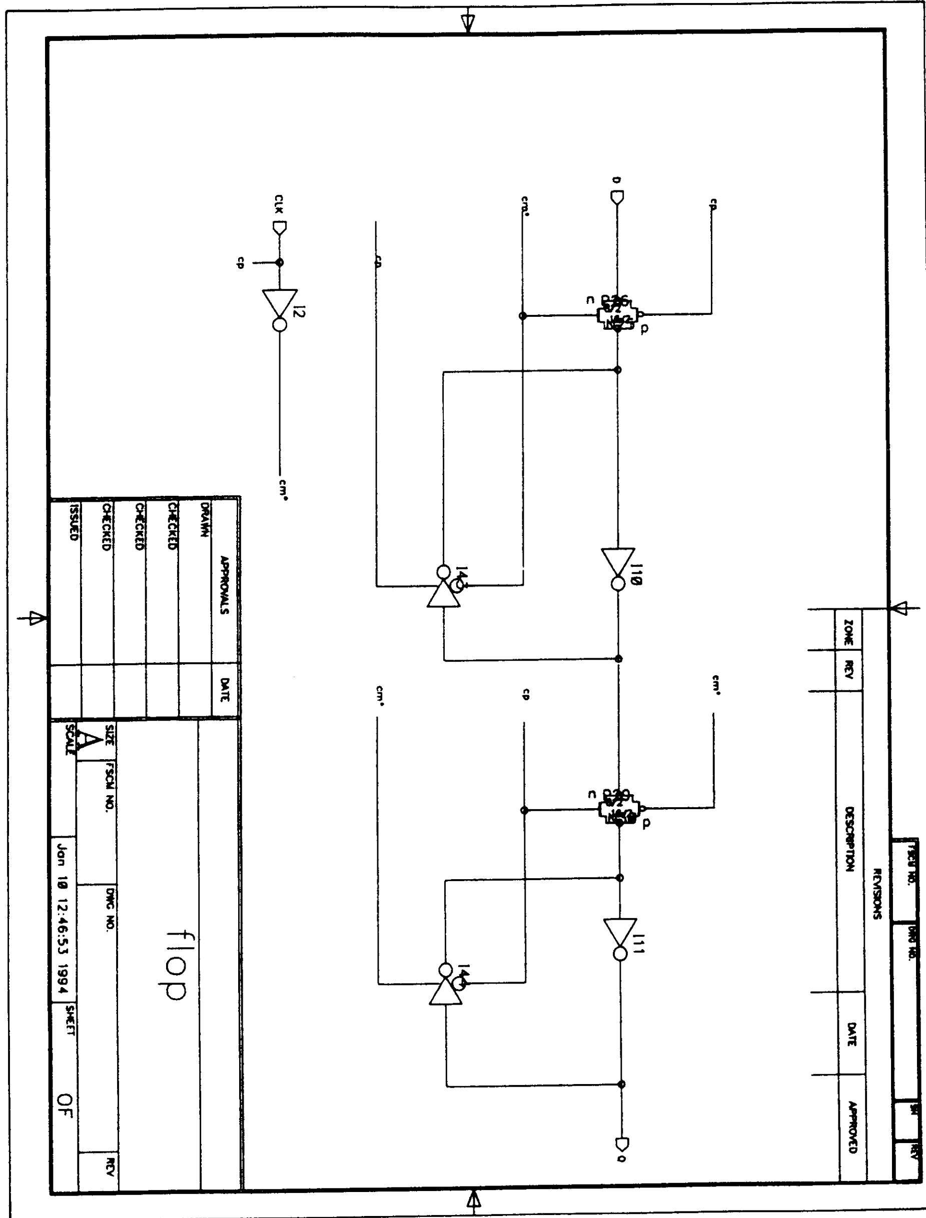
mx4 schematic

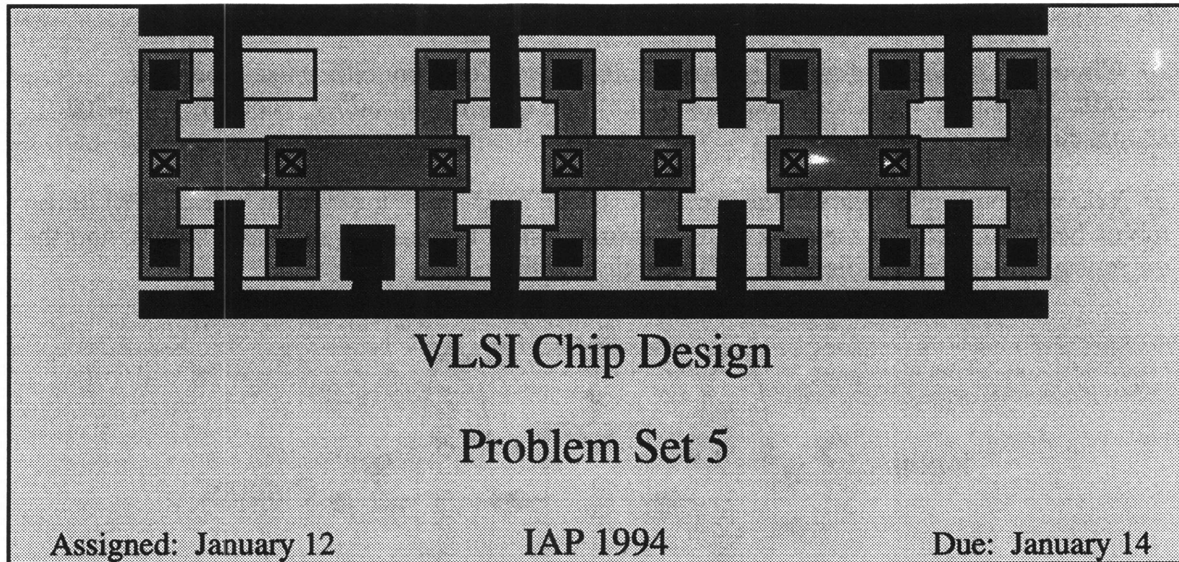


latch schematic



flop schematic

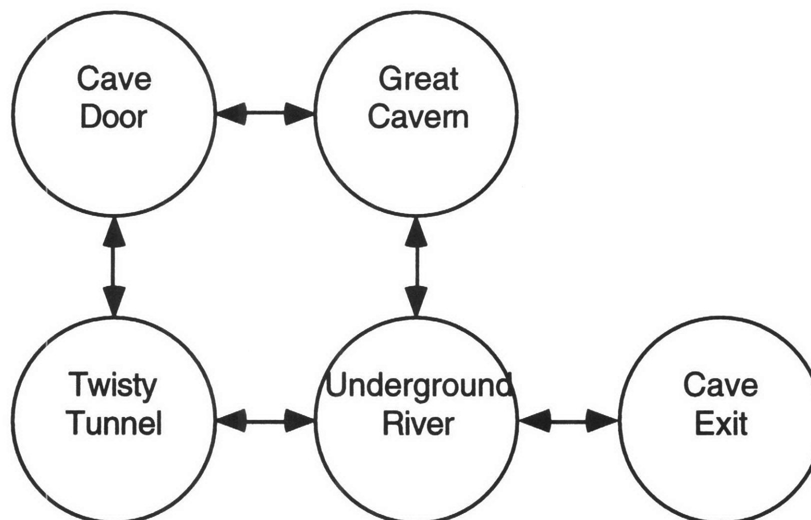




1) Finite State Machine Design

Secret Agent 6.007 has been hired as a consultant for the Assassins' Guild to build an adventure game on a chip. The adventure game should be implemented as a finite state machine.

For the first version, this game will be very simple. It will have five rooms: the Cave Door, the Great Cavern, the Twisty Tunnel, the Underground River, and the Cave Exit. The rooms are connected as shown below. The goal of the game is to get from the Cave Door to the Cave Exit. The Assassins' Guild plans to sell the game to Hahvahd students, who are predicted to get weeks of enjoyment from the game.

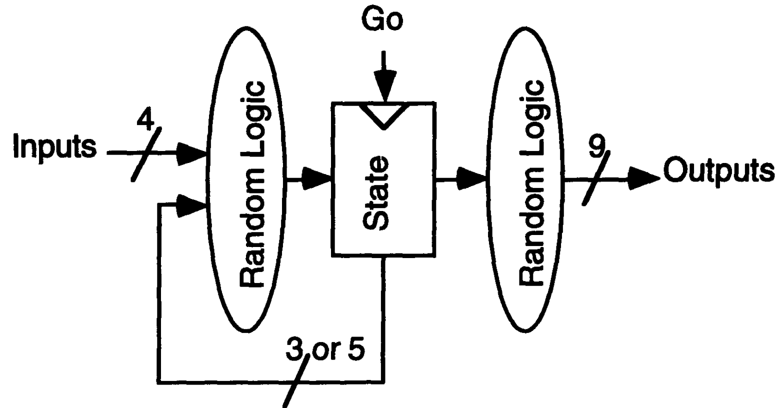


The game has five controls (**N**orth, **S**outh, **East**, **W**est, and **R**eset) and a **G**o button. The player holds down the desired control and presses Go to clock the circuit. There are nine outputs: four indicating if there are exits to the North, South, East, or West, and five indicating if the player is in the **D**oor, **C**avern, **T**unnel, **U**nderground river, or e**X**it. When the Reset control is high and Go is pressed, the player is teleported to the Cave Door to begin the game.

a) Draw a state transition diagram.

b) Choose a mapping of states to binary numbers. You may either use a packed encoding (3 bits to represent one of 5 states) or a one-hot encoding, according to what seems easiest.

c) You will be implementing the game as shown on the block diagram below. Fill in the tables below indicating the next state as a function of the current state and inputs, and the current room and available exits as a function of the current state.



Current State	N	E	W	S	R	New State

Cur State	N out	E out	W out	S out	D	C	T	U	X

d) Write logic equations in sum-of-products form for the new state as a function of the current state and inputs and for the room and exits as a function of the current state.

2) Standard Cell Implementation

Implement your FSM design from part 1 using standard cells from the stdcells library. You will probably want to arrange your standard cells in two or more rows instead of one really horribly long row.

3) PLA Implementation

Implement your FSM design using a PLA and standard-cell flip-flops. Note that you only should need one PLA; it will take the current state and N, S, E, W, R as inputs and will produce the next state, the directions available, and the room as outputs.

4) Comparison

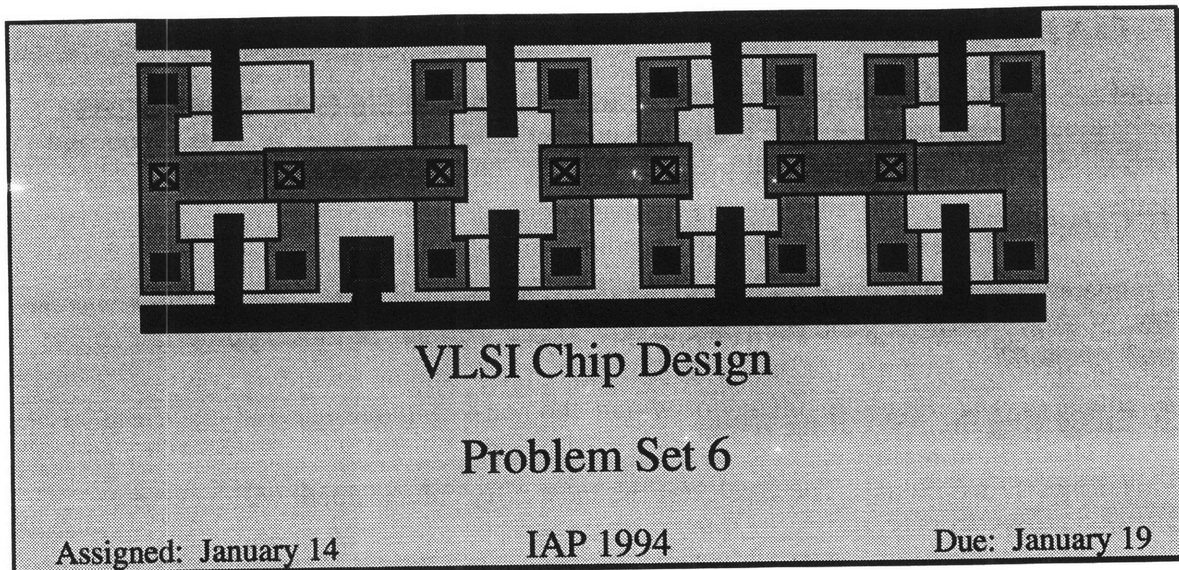
Compare your implementations of the finite state machine in parts 2 and 3. What was the area of each? How long did each take to do? When would it be most appropriate to use each method?

5) Enhancing the Adventure Game

This problem is optional. The creator of the most nifty enhancement will receive a Toscii's gift certificate.

Thing of a feature to add to your adventure game. Examples would be adding an Three Headed Troll (or a grumpy graduate student) who you could not pass until pressing a kill button or a bottomless pit that puts you in the dead state with no exit save reset. Be creative!

Describe the new feature and draw your new state diagram. Write your new set of state equations. Then implement the FSM in the method of your choice.



This problem set is intended to give you practice programming the Unintel Sexium. If you have never programmed in assembly language before, you may find this problem set rather challenging; if you get stuck, please ask David or a fellow student for help getting started. Write your code in an emacs window in your directory at the AI lab and test and debug using the MSIM simulator. When your code works, turn in a sheet of paper with the names of your two programs.

1) The Electronic Accountant

Secret Agent 6.007 is posing as an accountant to infiltrate the enemy headquarters. His exciting and rewarding job as an accountant is to add up lists of 64 numbers. Secret Agent 6.007 wants to program his Sexium microprocessor to add the numbers, then let it do his work while he sneaks around the headquarters.

The numbers are stored in memory addresses $1000-$103F. Each is a single byte. The sum should be a two byte quantity and should be left in R2 and R3 (R2 containing the most significant byte) at the end. The code below solves this problem. It is available in /home/cva2/6090user/tools/msim/sum.asm.

```
# Sum.asm
#
# Written 1/14/93 by David Harris
#
# This program sums 64 numbers, and
# leaves the results in R2 and R3.
# (R2 holds most significant byte of sum)
# The numbers are located in memory
# addresses $1000-$103F.

Start:
      LDM $10           # Set MA to $1000
      PUT MAH
      LDM $00
      PUT MAL
      PUT R2            # Clear R2 and R3 to 0
      PUT R3
Loop:
```

```
        LDI                     # Load next number X to add
        PUT R1                  # Save a copy in R1
        TST R3                  # Check if X + R3 > 255
        PUT R0                  # Temporarily store result of test

        GET R1                  # Recall number to add
        ADD R3                  # Add X to R3
        PUT R3                  # and put it back in R3

        GET R0                  # Recall result of test
        LDM $01                 # mask off carry but of test
        AND R0
        SKZ                     # Skip if carry is false
        BRA Increment           # if carry, increment most significant byte
        BRA Check               # Else check if done
Increment:
        LDM $01                 # Otherwise
        ADD R2                  # Add 1 to R2
        PUT R2
Check:
        LDM $C0                 # = -40
        TST MAL                 # Check if we've gotten to $1040
        PUT R0                  # Mask off Z bit
        LDM $02
        AND R0
        SKZ                     # Skip if zero
        BRA Done                # Otherwise quit
        BRA Loop                # If not done, continue looping
Done:
        BRK
```

Unfortunately, while Secret Agent 6.007 is still out, a new list of 4 numbers comes in. They are stored in addresses $1108-$110b. Help Secret Agent 6.007 modify his program to handle the new list before his boss comes looking. To get some practice with Sexium assembly language, make a copy of the code in your own directory. Change it so that it properly adds these numbers; this time, store the results in R0 and R1 instead. Test your code by using the Memory Write command to store four different numbers in $1108-$110b, then running the code.

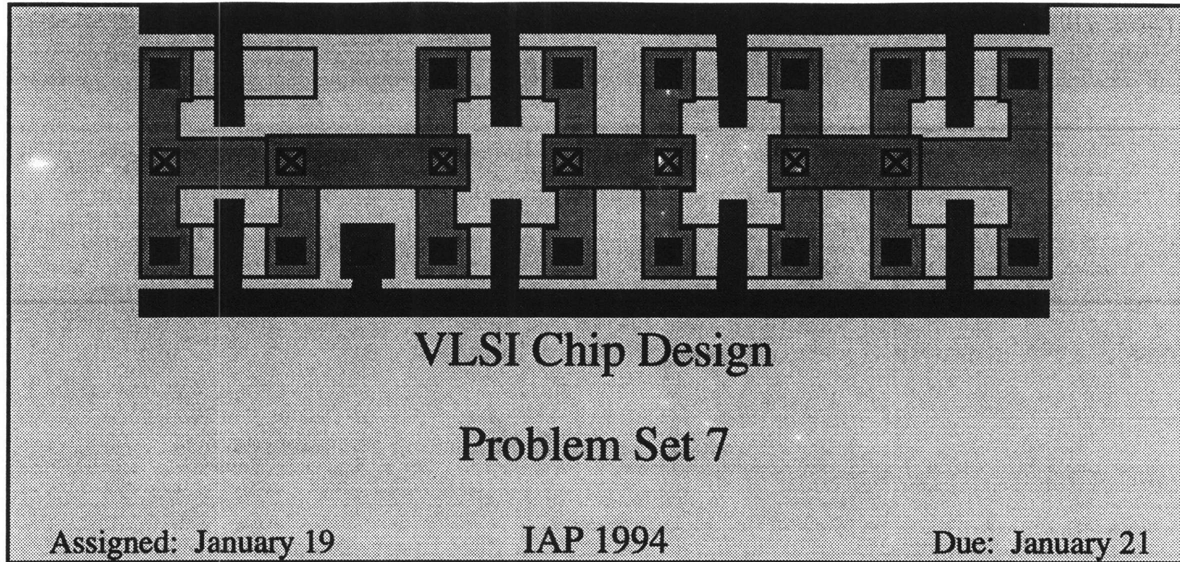
2) Insertion sort

The insertion sort is a method of sorting a list of N numbers. It is not an efficient sorting algorithm, but works well for short lists and is the easiest sorting algorithm to program. The code below is an implementation of insertion sort in C.

```
void sort(int N, int data[])
{
        int i, j, swap;

        for (i = 0; i < N; i = i + 1)
                for (j = i; j < N; j = j + 1)
                        if (data[i] > data[j]) {
                                swap = data[i];
                                data[i] = data[j];
                                data[j] = swap;
                        }
}
```

Assume that N is initially stored in register R0 and that the N numbers are located in memory addresses starting at $1000. Test your code by setting N=6 and putting 6 different numbers in memory addresses $1000-$1005.

A Toscii's certificate will be given to the hacker who produces the shortest insertion sort code. If there is a tie, the programmer with the better documentation (e.g. most clear without being overly verbose) wins.



We have now begun the Sexium layout. In this problem set, you will be working as teams to layout each of the leaf cells in the Sexium. In the next problem set, you will be assembling the leaves into complete modules. In the final problem set, we will be assembling the modules into a full chip.

The following teams have been established:

alubox	pcmabox	regbox	control
Ruben	**Dan**	**Jeff**	**Ethan**
Matt	Bayard	Aarati	Nehal
Robert	David		

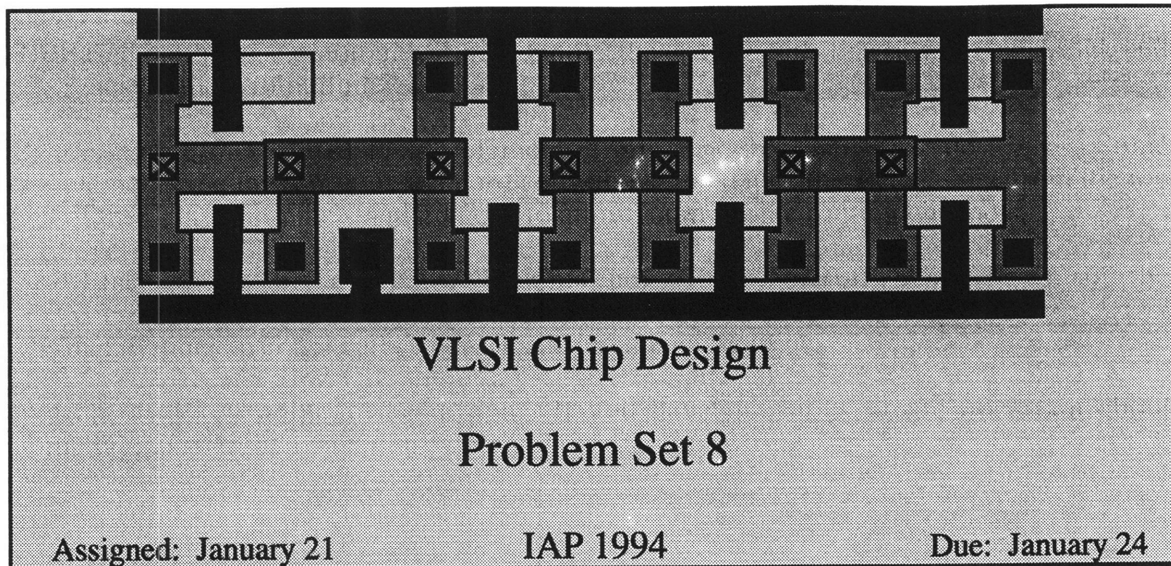
Each team has several leaf cells to layout by Friday. The cells should be designed in the unintel library. The teams should divide up the labor and design the cells according to the datapath design rules (except the control team, which should use PLAs and standard cells). Run DRC and LVS to make sure that the cells are correct. Pay close attention to where the metal 2 tracks are supposed to be placed. Several of the leaf cells are used in many modules; they have been divided among the various groups.

alubox
mux4bit
mux5bit
adderbit
anderbit
negerbit
shifterbit

pcmabox
halfadderbit
flopbit
mux2bit
mux3bit

regbox
triregbit
tribufbit

control
enter equations for PLA and generate it
construct following cells from standard cells:
- decoder2to4
- counter



In this problem set, we will be optimizing the cells that we have designed in Problem Set 7 and assembling them into complete modules. We have time reserved in 1-115 on Saturday from 12-6 and all day Sunday; it should be an intense but exciting weekend getting the modules assembled, optimized to fit, and verified!

1) Based on the discussion in lecture, identify the critical cells that need to be shrunk and redesign them to minimize area.

2) Assemble the 8 bit datapath elements from the 1 bit slices you designed. Some elements require a row of standard cells above the datapath. Leave 25 λ from the top of the datapath to the standard cells to allow room for the routing channels.

3) Place the 8 bit datapath elements next to one another as shown in the Microarchitecture Reference and connect the appropriate wires. Run lines extra metal2 lines over the cells where needed to connect bitlines.

We are at a key point in the design. If you have questions or are uncertain about something, ask right away; don't delay or risk doing it wrong.

Scribe Notes

The following sets of notes were taken by students and handed out at subsequent lectures.

6.008 Notes
Monday, January 3d 1994
Lecturer - Bill Dally
Scribe - Robert Ristroph

General.

Two recommended texts -- _Principles of CMOS VLSI Design_ by Weste and Eshraghian, and "The Art of Electronics" by Horowitz and Hill. _The Art of Electronics_ is recommended by Dave as a good general reference; _Principles of CMOS VLSI Design_ covers a lot of the basics in Chapter 1, which you should read if you get the book.

Note: Contest for a good 6.008 logo, due Friday; use either Mac format or PostScript on Athena.

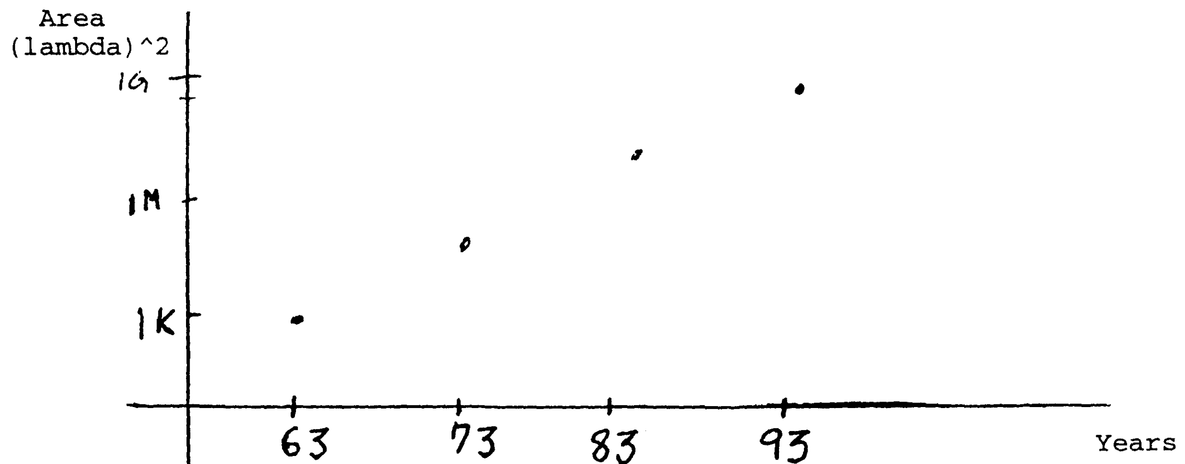
Introduction.

We will be designing an 8-bit microprocessor on a Tiny Chip. One of the opened chips that was passed around was a Tiny Chip; 8-bit means the processor handles numbers 8 bits in length.

A "one-micron process" means that the transistors produced have a gate with the width of one micron. We will be using a two-micron process.

Bill Dally showed us an eight-inch wafer, which contained many individual chips on it; about 35% of the chips on it were bad, which is about average. Each chip costs about $15 to make exclusive of packaging and testing; after breaking up the wafer and putting the chips in ceramic or plastic packages, the chips cost around $20.

The rate of progress in VLSI (Very Large Scale Integration) is very fast. A measure of area on a chip is , where lambda is the length of the gate. We have a nice graph from showing the rise in useful area on a chip against time:



Useful area is about quadrupled every three years, because the gates get smaller and the chips get bigger.
1000 = 1 bit static RAM (SRAM, or Static Random Access Memory)
10,000 = 1 bit full adder
100 to 200 = 1 bit DRAM cell (Dynamic Random Access Memory)
Some historical tidbits ---
-- Bob Noyce and Jack Kilby were people who built amplifiers cheaply on a single chip, instead of making transistors

separately and then hooking them up
<--- 1st microprocessor built by Ted Hoff at Intel in 1970 called the 8008

A "Data Book Engineer" picks out components with nice characteristics out of data books. In contrast a VLSI engineer has more freedom, and is less constrained -- but this means the engineer has to know more about various areas, and bring them all together.

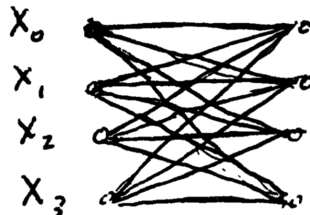
- Architecture
- Logic design (higher level than just AND and OR gates)
- Floor Plan
- Cell Design
- Circuit
- Layout

When building a circuit out of components, it is often appropriate to approximate the circuit size by the number of components, while wire is mostly ignored. In VLSI, a more appropriate approximation of circuit size comes from the amount of wire.

Here is an example of a barrel shifter.
A 4-bit barrel shifter takes in a four bit number and an instruction telling it how many places to shift it. The binary number 1100 would look like the following if shifted:

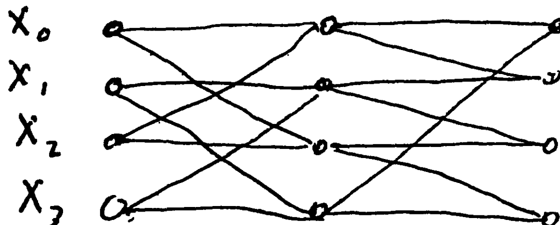
Shifted by 0 : 1100
Shifted by 1 : 1001
Shifted by 2 : 0011
Shifted by 3 : 0110
Shifted by 4 : 1100

We could make a shifter with each input bit connected to every output bit and a four-way (or N-way, to be general) switch to decide where it would go. But this would leave with on the order of n squared wires, which would take up too much room if we made a 16-bit or 32-bit shifter.

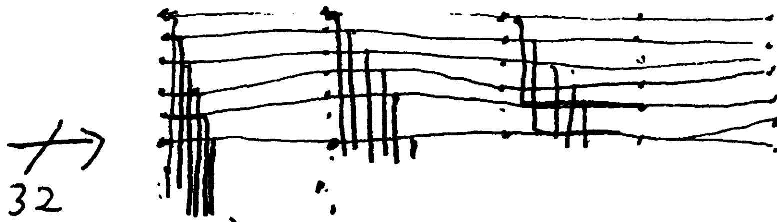


$n^2 = 16$ wires
4 switches

Instead we can build the shifter in stages, with each stage either shifting the bit down or sending it straight accross. The first set of stage eeither sends the bit straight across or shifts it down by N/2 (N is the number of bits the shifter handles), the second stage either shifts it down by N/4 or sends it straight across, etc. Here is the four-bit shifter:



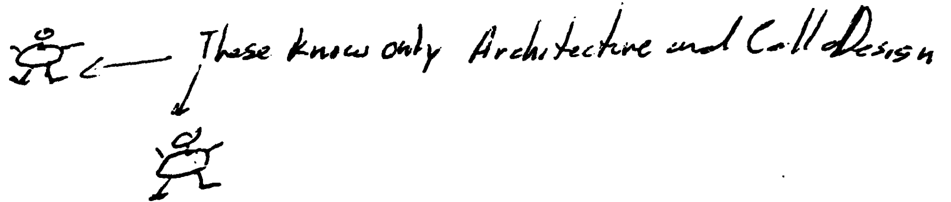
Note that there are log base 2 of N stages. Since there are N bits at each stage, we only need N log2 (N) switches. With 32 bits this is 160, not a large number; but notice how the band of wires running down again takes up a lot of room.



Thick bands of wires show how wires still dominate space

VLSI engineers try to be relatively knowl=dgeable about all of the areas listed above, so that they are a "tall thin man" who knows a little about each each rather than a "short fat man" who only knows one area well, but maybe very well.

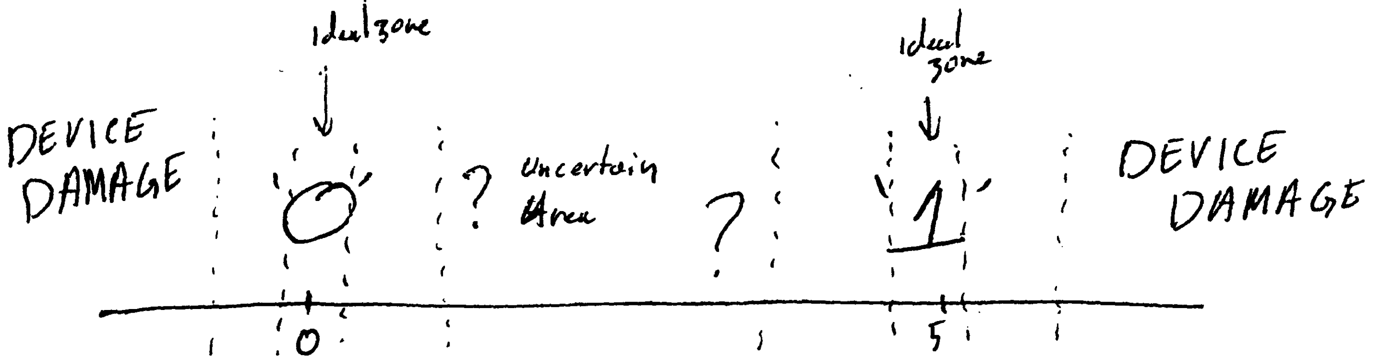
-Architecture
-Logic Design
-Floor Plan
-Cell Design
-Circuit
-Layout



a little bit of him everywhere

Signals.

Our processor will manipulate digital signals. A Signal can be represented by a wire or line on the page. Because are using digital signals, we want out signal to hold either the value of "0" or "1". In reality, the voltage (our signal level will be indicated by a voltage) should hopefully be in the indicated region:



Too high or too low a voltage will destroy the transistor or other device. The ideal region is narrower than what will be interpreted as a "1" or "0" because we want to allow for some swing due to noise. 5 volts was once common as "1", but now 3.3 volts is becoming more common, and it will drop further as gate distance shrinks.

In order to do logic we need a switch, which we will represent as

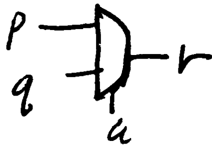
where the switch is closed (conducts) when a is true.

(Nice Historical Note: At Aiken Labs they have an old computer made of electro-mechanical relays for computing Bessel functions in relation to ballistics work.)

We now start to develop a system of logic with switches.

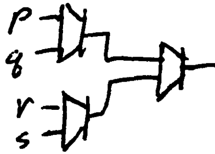
The following circuit will choose between two inputs (a' is the compliment of a, or NOT a: "0" when a is "1", "1" when a is "0"):

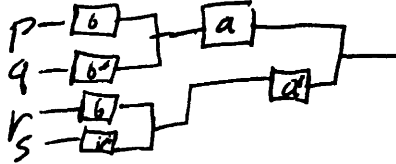
Abstracted to a simpler picture, we call this the two-input multiplexor. It allows us to choose between two values. Also, here is a truth table for the multiplexor.



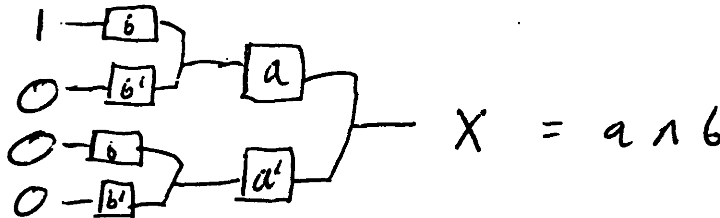
a	p	q	r
1	x	x	p
0	x	x	q

Here is a four-input multiplexor, drawn with two-input multiplexors and with just switches.

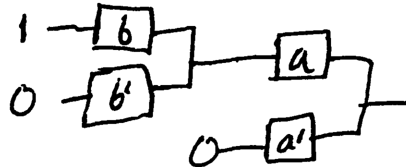




We can represent "a and b" or a^b with multiplexors by setting the input so the multiplexor chooses an input of "1" only when both a and b are true.



This diagram is partly redundant, and can be made more efficient:

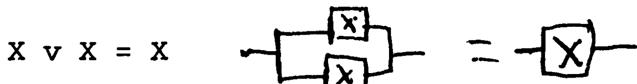


Since we are going to be using switches to do Boolean Algebra, a quick review is helpful.

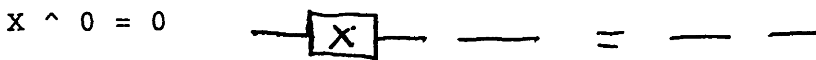
Boolean Algebra.

Basic Axioms:

X ^ X = X

X v X = X 

"^" is "and" and "v" is "or"

X ^ 0 = 0 

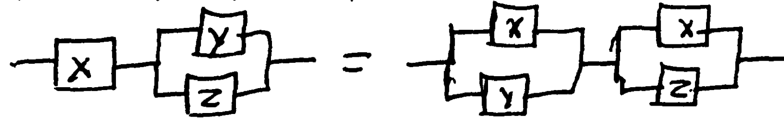
X v 1 = 1

The last to above are duals of each other. To obtain the dual of an equation, change all 0s to 1s, 1s to 0s, and's to or's and or's to and's.

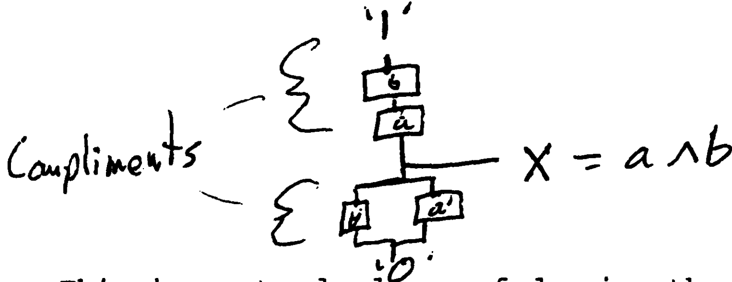
X ^ Y = Y ^ X

X ^ (Y ^ Z) = (X ^ Y) ^ Z

X v (Y ^ Z) = (X v Y) ^ (X v Z)

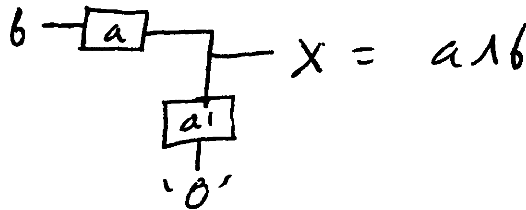


Using these laws, we can make our a ^ b diagram better yet.



This is a standard way of drawing these logic equations -- with "1" at the top and "0" at the bottom. Note that the expressions in the top and bottom branches should be compliments of each other. We would like to avoid ever having a short from top to bottom, because too much current can cause the migration of metals in the device, which can ruin it.

There is a simpler structure yet:

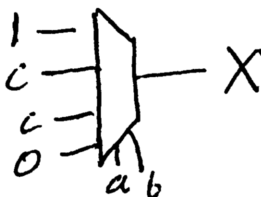


However, this does not have have the property that it restores the signal. Since b itself is passed through a gate, the voltage may drop or become contaminated with noise. Doing this more than once can bring you out of the region that will be read as a "1". We will try to use only restoring circuits.

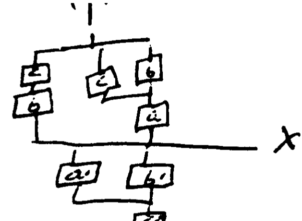
Example: suppose we wish to make a function that will return a "1" if 2 or 3 of its 3 inputs are "1", and a "0" otherwise. We might start by drawing a truth table of our function:

a	b	c	X	
0	0	0	0	0
0	0	1	0	
0	1	0	0	c
0	1	1	1	
1	0	0	0	c
1	0	1	1	
1	1	0	1	1
1	1	1	1	

We could then reduce this to a non-restoring circuit using a multiplexor:



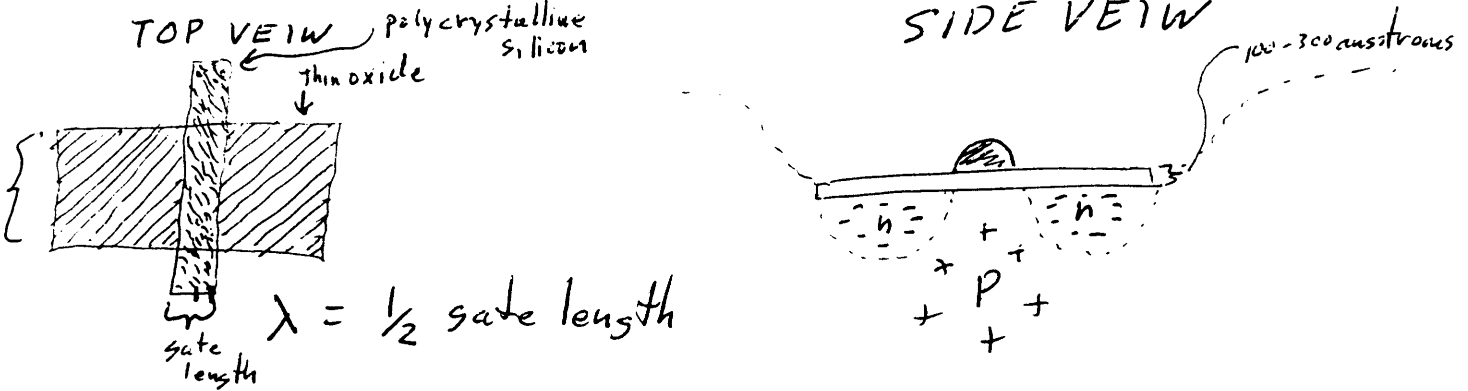
Or we could make a restoring circuit with our method of paths to "1" and "0":



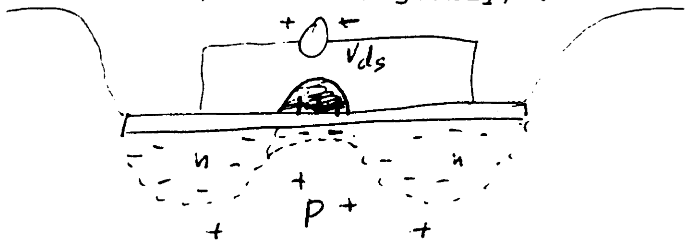
Now that we have a basic way of making things happen with switches, let's shift our focus down to a different level and look at how to make the switches themselves.

Switches.

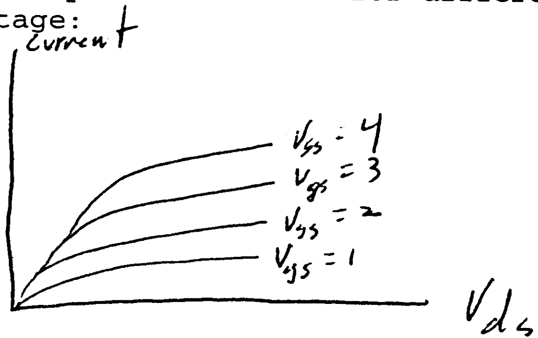
Imagine a wide, thin piece of metal oxide laying on a silicon substrate. This is crossed by a narrower strip of polycrystalline silicon conductor, and two separated regions of n-type substrate are created underneath the oxide:



This is a transistor. (N-type material moves charge by moving electrons, p-type material moves charge by moving electron holes.) A positive charge on the gate will repel the positive holes underneath, creating a thin n-type region connecting the two sides (which are called the source and drain, interchangeably) :



So if the gate is more positive than the more negative of the source and drain, then the source and drain will conduct. In the following diagram, different I-V plots are shown for different voltages of drain-to-source voltage:

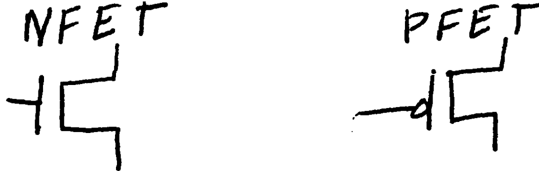


This only makes part of a switch because it can't reliably pass a "1". It has a tendency to turn itself off because if the drain and n-channel underneath the gate have too high a voltage, they stop repeling the electron holes so the n-channel stops conducting.

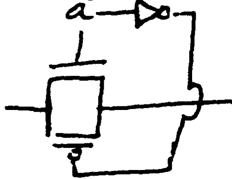
However, a p-type transistor will pass a "1" but not a "0". A PFET is the opposite of an NFET:

NFET stands for N-channel Field Effect Transistor, and PFET stands for P-channel Field Effect Transistor. MOSFET means Metal Oxide Semi-conductor Field Effect Transistor.

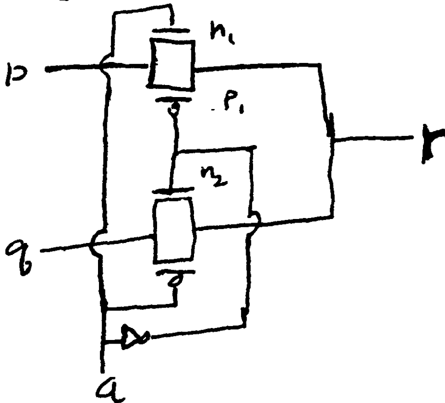
We represent the two switches by the following symbols:



Here is diagram of a good switch, that will pass both "0" and "1":

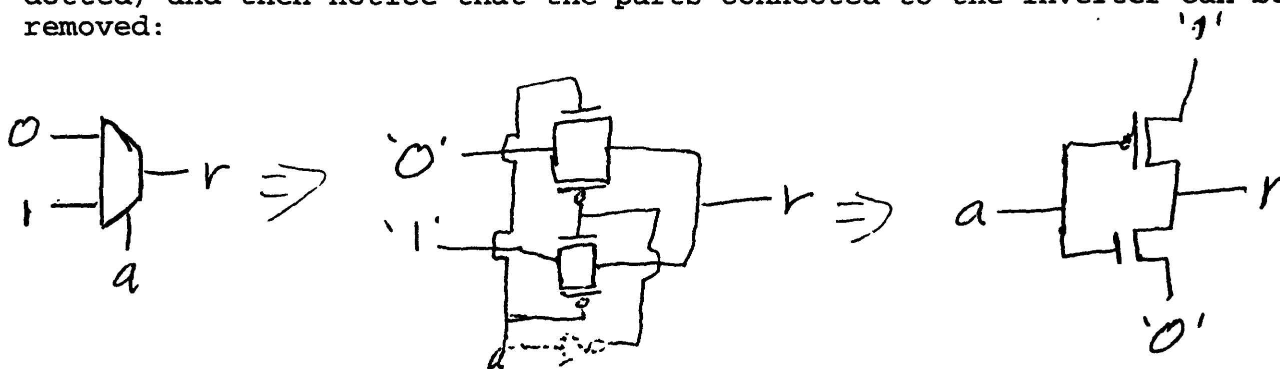


Note that it uses an inverter. We have not yet built an inverter, but we will. Here is a circuit of a multiplexor. Note the truth table off to the side in which we check that in each state there is a path to the appropriate value through the transistor type that can reliably carry that value.



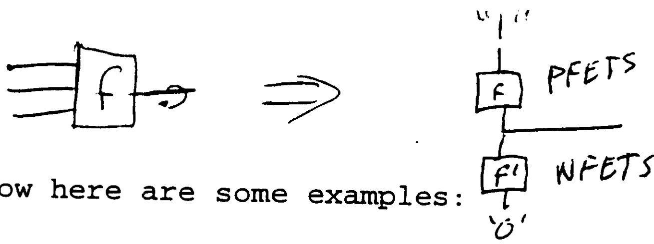
a	p	q		
0	x	0	0	n_2 p_2
0	x	1	1	n_2 p_2
1	0	x	0	n_1, p_1
1	1	x	1	n_1, p_1

Now we want to make an inverter, to complete our system. Start by making a multiplexor (which recursively has an inverter in it, drawn dotted) and then notice that the parts connected to the inverter can be removed:



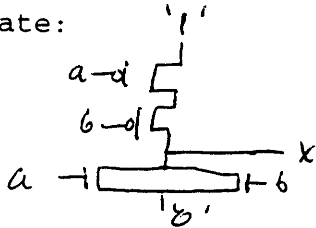
Now that we have seen how to build switches and use them to produce logic, here is a brief summary:

A logic gate can be made into two paths, each of which is connected to "0" or "1" and which are composed of PFETs or NFETs, as is appropriate for the signal they carry.



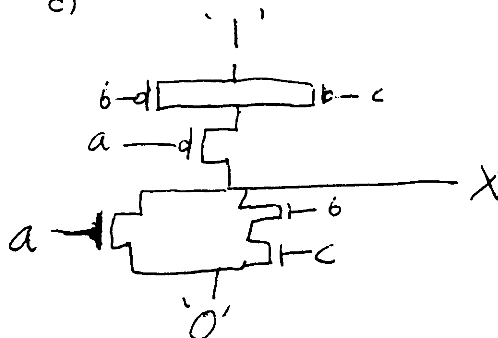
Now here are some examples:

Nor gate:



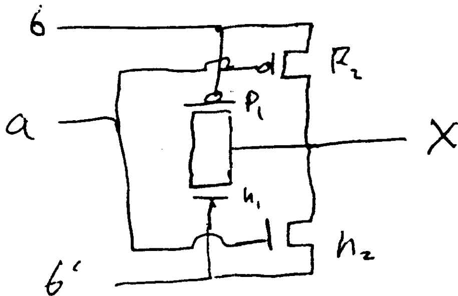
f = (a v (b ^ c))' = a' ^ (b' v c')

f' = a v (b ^ c)



Exclusive Or:

X = (a ^ b') v (b ^ a')



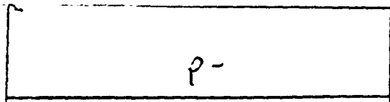
a	b	
0	0	n_1 p_1 p_2
0	1	p_2
1	0	n_1 n_2 p_1
1	1	n_2

44 6.008 Lecture Notes #2 by Nehal Patel
Lecturer: Bill Dally
1/6/94

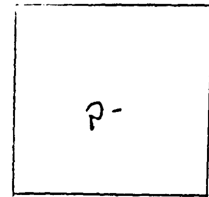
Fabrication:

The fabrication of CMOS devices involves steps that add various layers of material onto a initial substrate. To illustrate this fabrication process, the fabrication of a CMOS inverter will be described.

1) Start with an initial silicon wafer. This wafer is called the SUBSTRATE. The substrate can be doped type p or type n. For this example we will use a p-type substrate.

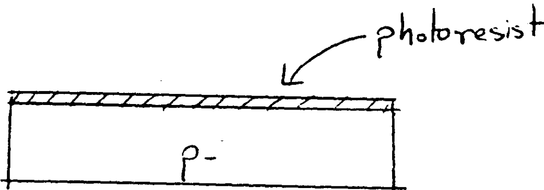


Cross-section of substrate

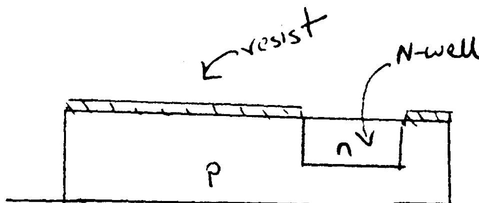


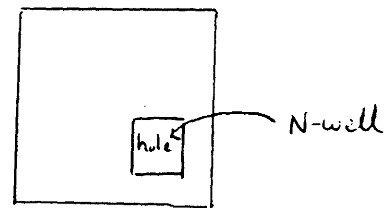
Top view of substrate

2) The n-well process. Our inverter will need both a p-type MOSFET and a n-type MOSFET. P-type mosfets require two p-doped regions submerged in an ambient n-doped region. To provide this ambient n-doped region, we make an N-WELL in our p-substrate (for a n-substrate, use a P-WELL for the NMOS.) To make this well, first we cover the substratewith a PHOTORESIST. This photoresist can be etched away by acid after development under UV light.

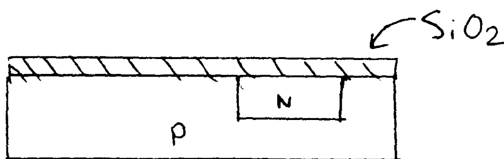


Then we design a mask that specifies the desired shape of the n-well (Essentially when we design a layout on Cadence, we are designing the various masks used in the fabrication process.) Through this mask we shine UV light to develop the photoresist where the n-well will be. Then we shoot ions thru the mask to n-dope the region. Now we have an n-well.

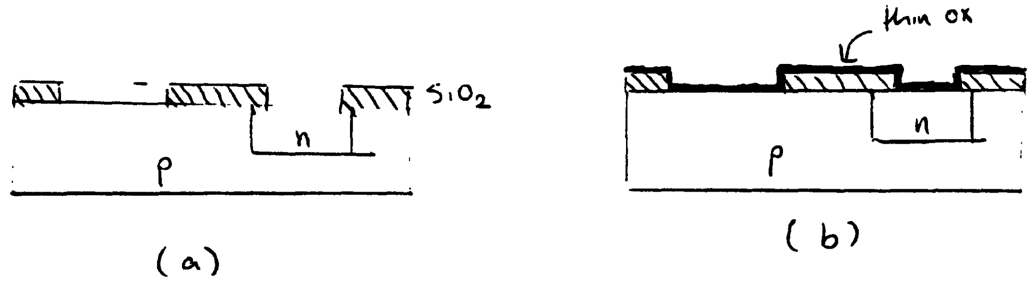




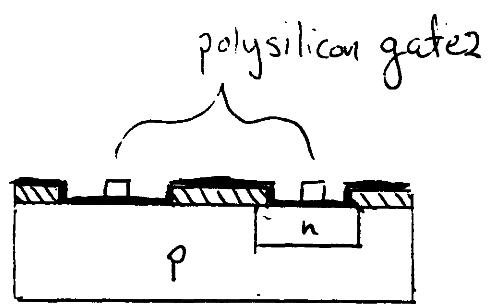
3) To insulate various materials on the wafer from each other, silicon dixoide is used. A layer of SiO2 can be grown on the wafer by placing it into a special high temperature oven. First, a nice thick layer of SiO2 is grown on the wafer.

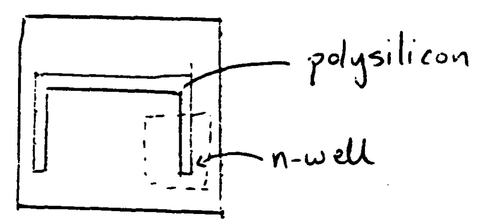


4)Next the SiO2 above the sites where the transistors will be located is etched away (a). Then a thin layer of SiO2 is grown on the whole wafer(b). This THIN OX will insulate the gate of the transistors from the source and the drain.



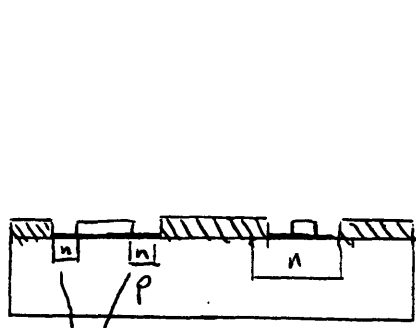
5) Now we make the gates to the transistors. Gates are made from POLYSILICON. Polysilicon is polycrystaline Si which can conduct reasonably well. The polysilicon is grown onto the wafer by using a special oven and then the desired shape of the gates is etched out.



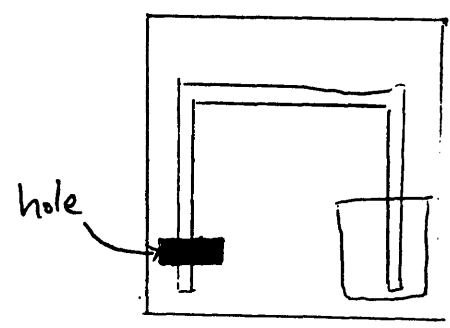


Final shape of the polysilicon layer
(TOP view)

6)Now we add the source and drain for the NMOS transistor. Using a mask, we beam ions into the wafer to dope it n-type. Notice that although the mask is rectangular, two disjoint n-doped regions(i.e the source and the drain) are formed. This is because the polysilicon gate blocks part of the hole in the mask.



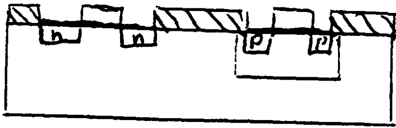
These are called
N-Diffusion regions



Top view of the mask

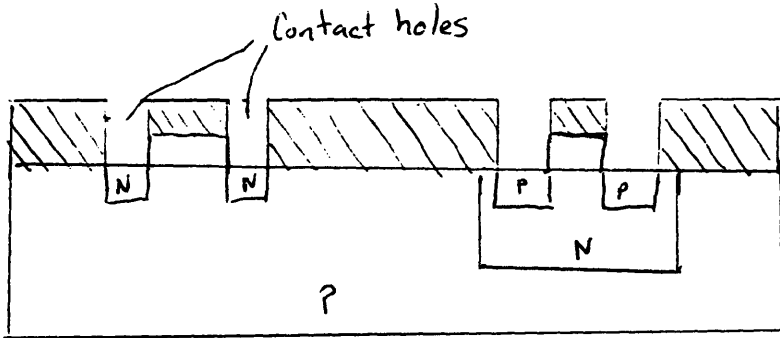
7) A similar process is carried for the PMOS transistor to form its source and drain.

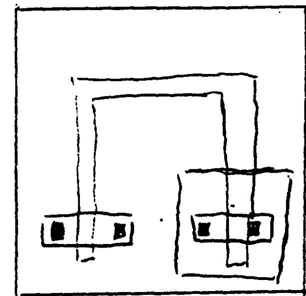
(figure below)



(figure for step 7)

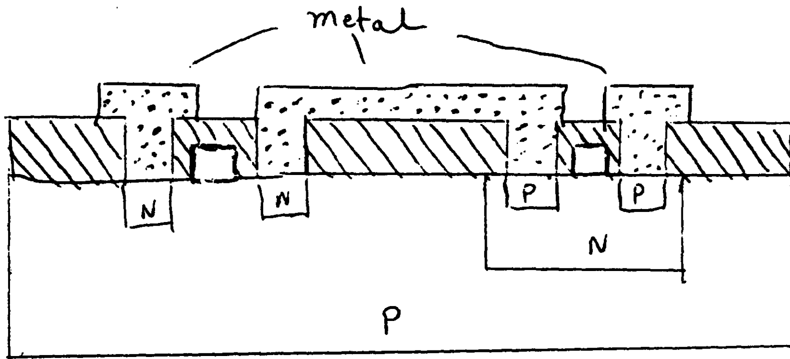
8)Now we are done with the ACTIVE LAYER (polysilicon,n-diffusion, p-diffusion). Now we will add a layer of metal on the wafer. First, to insulate this metal from the rest of the chip, we grow layer of SiO2 on the wafer. The CONTACTS are places where the metal will touch the active layer. Contacts are formed by etching away holes in the SiO2 currently covering the wafer.

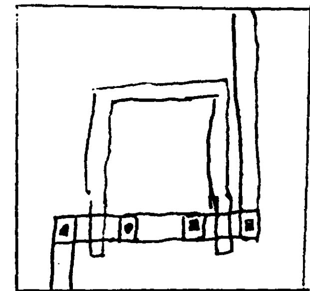




Contact Mask (TOP View)

Now, using a mask, metal is placed onto the wafer.





(TOP View)

9) The final step is for the wafer to be PASSIVATED to protect the surface from contaminants.

Since an inverter is a relatively simple device, only one layer of metal was needed. For more complex circuits, an additional layer of metal is convenient. This is formed by placing a layer of SiO2 on the first metal layer, cutting holes into the SiO2 for contacts, and then adding the second layer of metal thru a mask. We will refer to the first layer of metal as metal1 and the second layer as metal2. Notice that metal2 can only be directly connected to metal1 (the contacts between metal1 and metal2 are referred to as VIAs). In order to connect metal2 to the active layer, a via is used to connect it to a stub of metal1 and then a normal contact connects the stub to the active layer. If 2 layers of metal are not sufficient, polysilicon can be used as an INTERCONNECT(wire), although this is less desirable due to the higher resistivity of polysilicon.

Circuit Design:

Topology conventions: When designing a LEAF CELL try to observe to following layout conventions:

1)Try to orient p-diffusion and n-diffusion regions horizontally, with the p-regions above the n-regions.

2)Orient the polysilicon vertically.

3)Run metal1 horizontally (doesn't have to be followed too strictly).

4)Run metal2 vertically " "

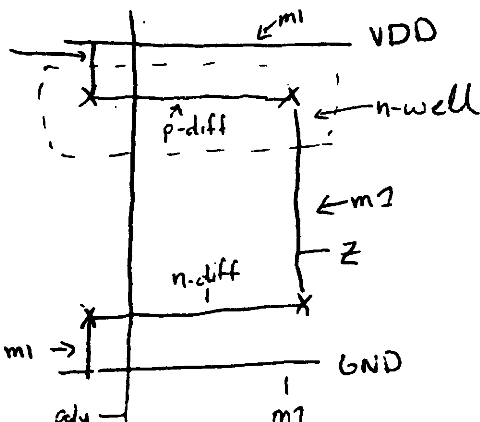
5)Run the power bus(VDD) horizontally at the top of the cell in metal1.

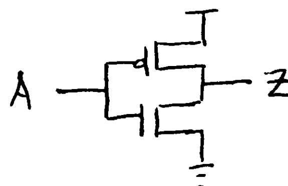
6)Run the GND bus horizontally at the bottom of the cell in metal1.

Stick Figures: To get an indication of the topology of a layout, stick figures may be used. A stick figure uses colored lines to indicate various regions on the wafer. X's indicate connections. We have adopted the following color convention:

n-diffusion: green
p-diffusion: pink/orange
polysilicon: red
metal1 : blue
metal2 : brown/purple

For example, the following is an example of a stick figure of an inverter which adheres to the adopted topology conventions (Note: this is NOT the stick figure for the inverter shown in the fabrication process example above. That inverter does not even adhere to the topology conventions we have adopted.)





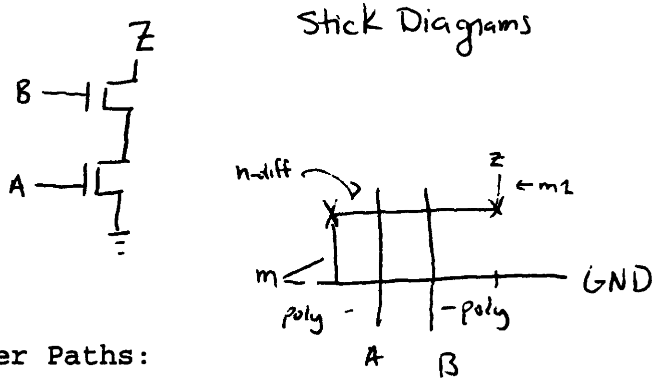
(Represents the topview of a leaf cell).

Design Strategies:

18

P-Diffusion Segregation: As we will see later n-well regions take up a large portion of the leaf cell. Since p-diffusion regions must be placed in n-well, they should try to be kept together so that they may be placed in the same well.

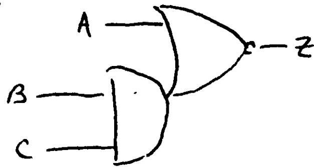
When then source/drain of a transistor is to be connected to the source/drain of another transistor (both transistors must be of the same type), then one long diffusion region might be used instead of two short diffusion regions:



Euler Paths:

There is a systematic approach to determining the fewest number of diffusion regions needed for your circuit. First, on the schematic of your circuit identify the NODES. Nodes are the sources/drain conections. Divide the nodes into two categories, those that are associated with p-type transistors and those that are associated with n-type transistors. For each set of nodes try to find a EULER PATH. An Euler path visits each transistor exactly once by jumping from node to node. If a Euler path cannot be found, find the longest path which jumps from node to node while visiting each transistor at most once. All the transistors on this path may be constructed with one diffusion region. Continue this process for the remaining transistors.

Example:



Euler Path 1:

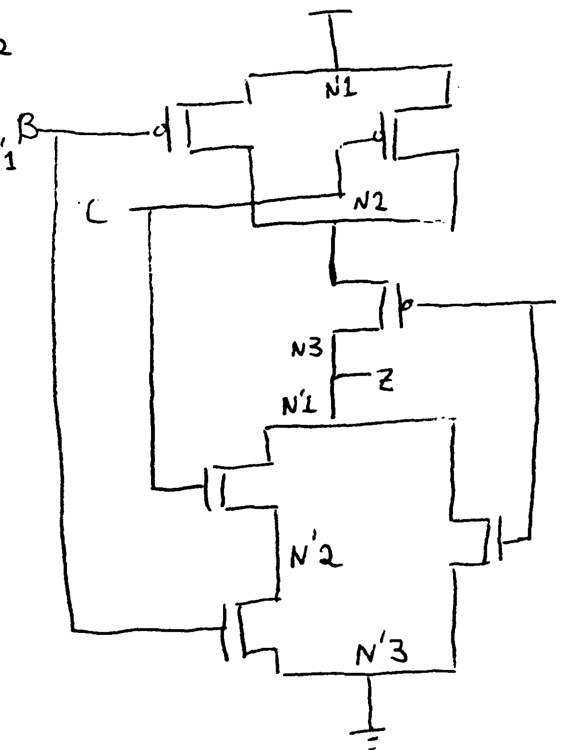
N3 $\xrightarrow{a}$ N2 $\xrightarrow{B}$ N1 $\xrightarrow{c}$ N2

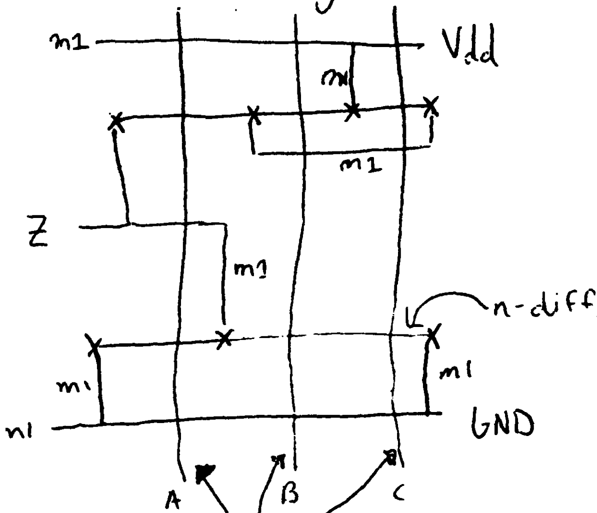
Euler Path 2

N'1 $\xrightarrow{a}$ N'3 $\xrightarrow{B}$ N'2 $\xrightarrow{c}$ N'1

Thus only 1 p-diff region and 1 n-diff region are needed

Schematic w/ Nodes





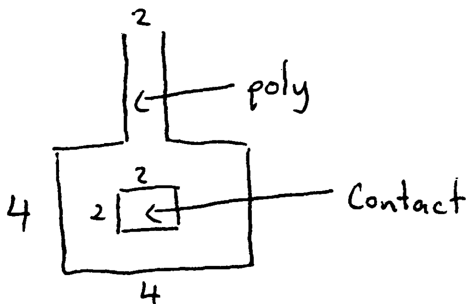
Geometric Design Rules:

Stick figures give an indication of the topology of a layout. Of course in reality, the regions depicted in the diagrams must have non zero width. The following are guidelines indicating the minimum width of each type of material, and the minimum distance between two regions of the same type:

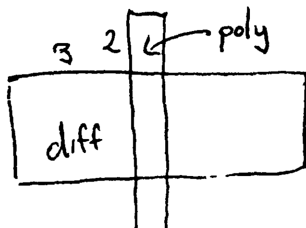
(All values have units of lambda)

Material	Width	Separation
Polysilicon	2	2
Diffusion	4	4
Metal1	4	3
Metal2	4	4

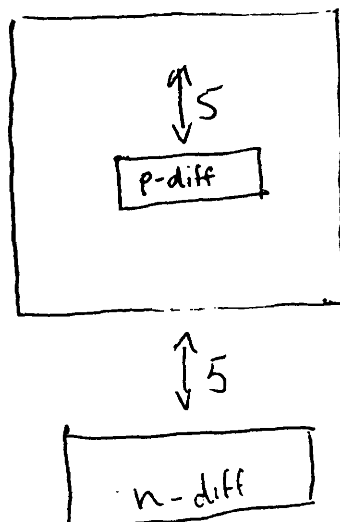
Contacts and vias are 2x2. Contacts must be kept 2 apart, vias 3 apart. When applying a contact to polysilicon, the polysilicon must be extended to width 4:



Polysilicon must overextend diffusion by at least 2, diffusion must overextend by 3:



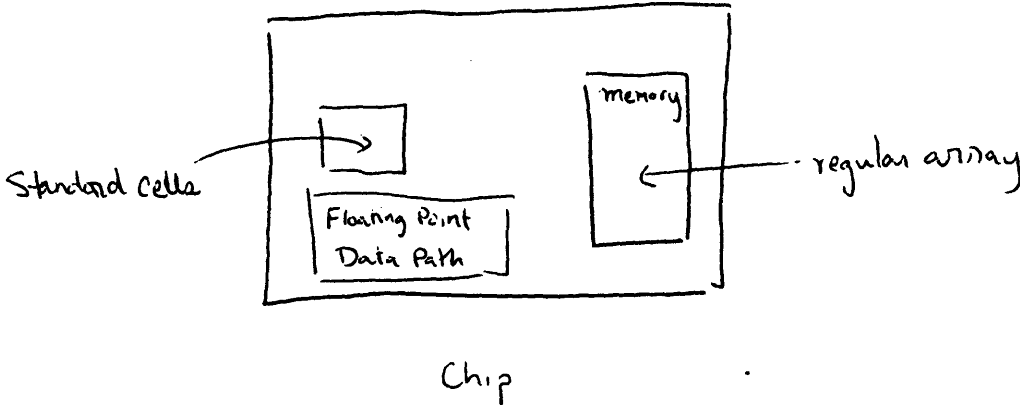
Wells must be 5 lambda away from any diffusion not contained in the well. Also wells must overextend its diffusions by 5 lambda:



Higher Level Architecture:

50

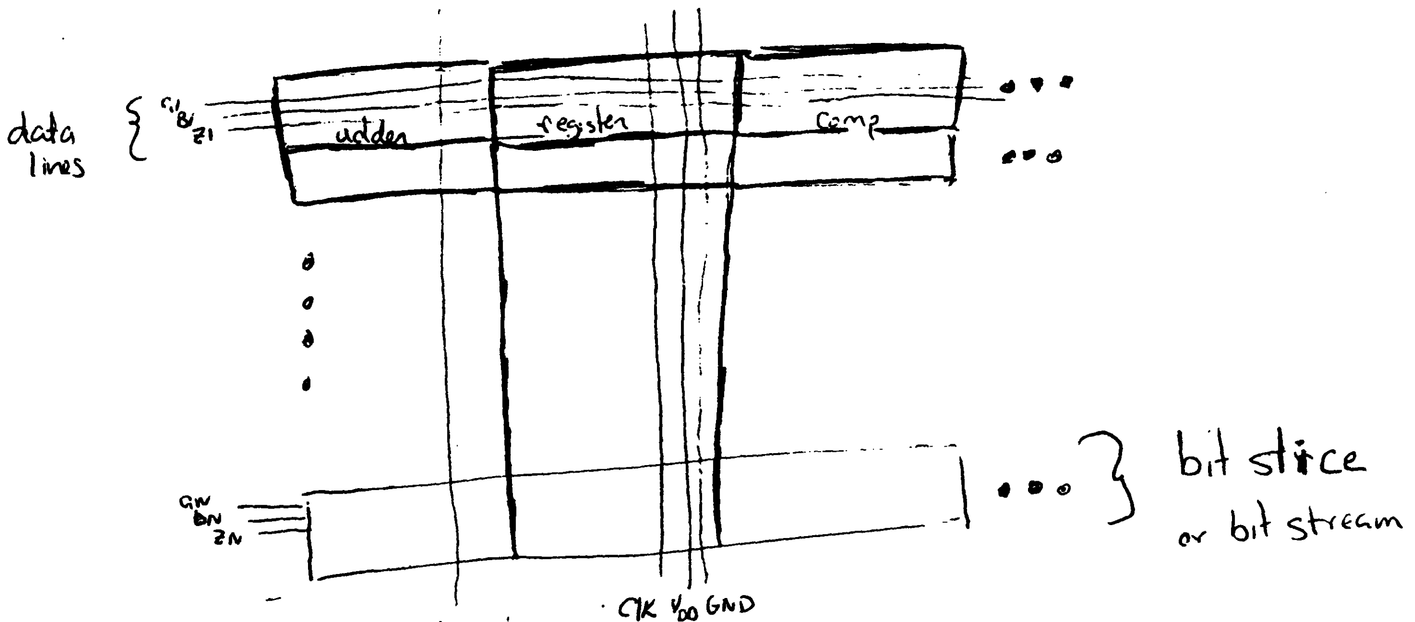
A chip can be decomposed into various units such as memory, the control unit, register files, etc. These various units are made up of the leaf cells that we have been discussing up to now. There are certain patterns that are used when connecting up leaf cells to make these larger units. Three common patterns are REGULAR ARRAYS, DATA PATHS, and STANDARD CELLS. Which pattern is used depends on the unit that is being designed :



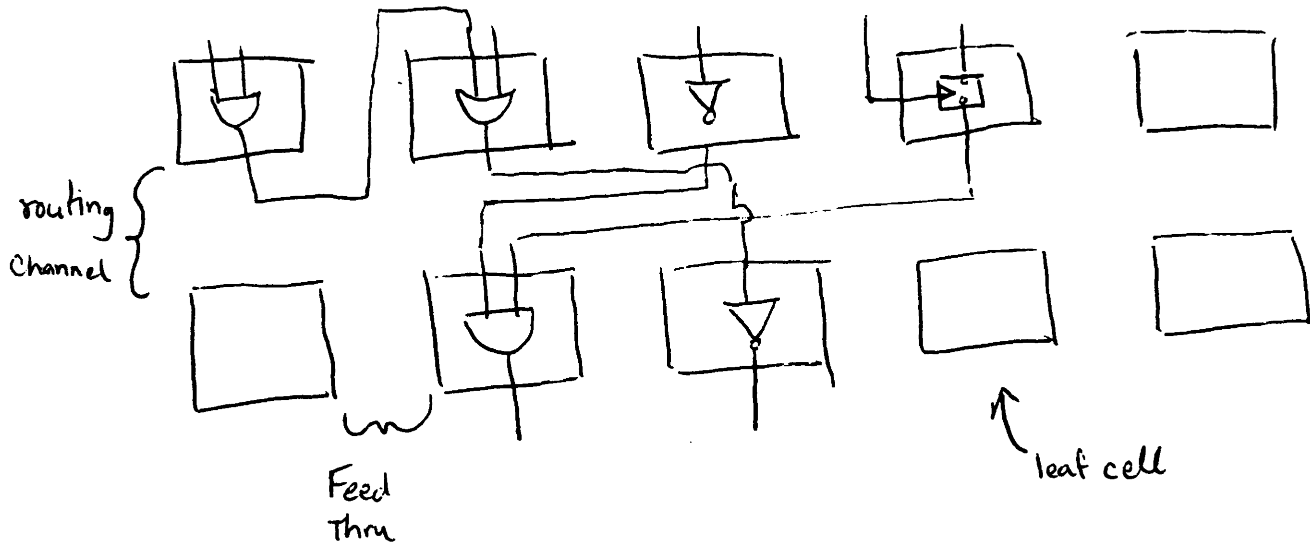
Regular Array:

Regular Arrays are the simplest of the three patterns. It consists of laying down many identical leaf cells in a 2D array, and connecting the leaf cells in a regular way. Regular arrays are found in structures that have highly repetitive nature such as RAM, ROM, PLA (Programmable Logic Unit).

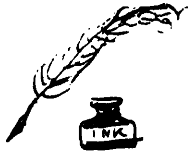
Data Paths:
An example of a data path might be a simple floating point unit. An n-bit chip will have data paths that are n BIT STREAMS high. A bit stream is a row of leaf cells that is responsible for producing the output for a particular bit. For example, a bit stream might consist of an adder, then a register, followed by a comparison operator, etc. Data wires run parallel to the data path (horizontal). Data that is needed in a different bitstream is passed through vertical CONTROL wires. CLK, GND, VDD also run vertical through the bit streams:



Standard Cells:
Standard cells are often used to code messy logic functions. These functions could be coded compactly as data paths, but in order to speed up the design process, the less efficient standard cell approach is used. The standard cell approach consists of laying out the needed logic gates in a grid pattern and then connecting the various gates together:



6.008 Lecture Notes 3
Friday, January 7, 1994
Lecturer: David Harris
Scribe: Aarati Parmar

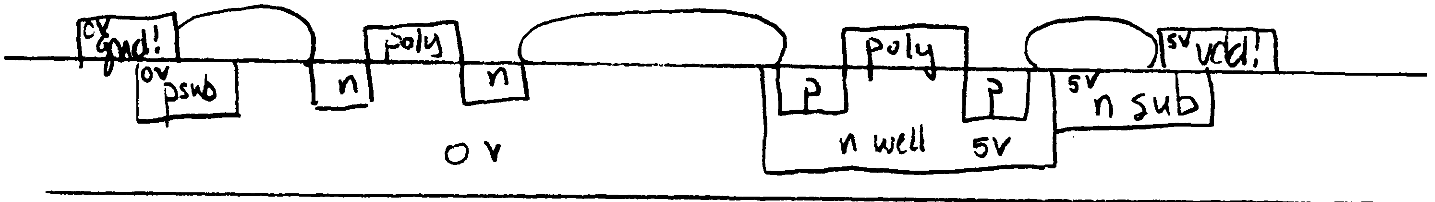


Reading:
6.1
6.2
8.1
8.2.1.1-8.2.1.2

Well and substrate contacts:

for these transistors to work, you need to keep the p-substrate (for the n-fets) at ground (0 volts) and the n-well (the p-fets) at vdd (5 volts)

ex : inverter

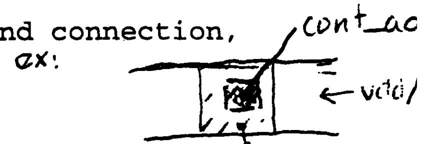


To recap:

p-sub at 0 volts
n-wells at 5 volts

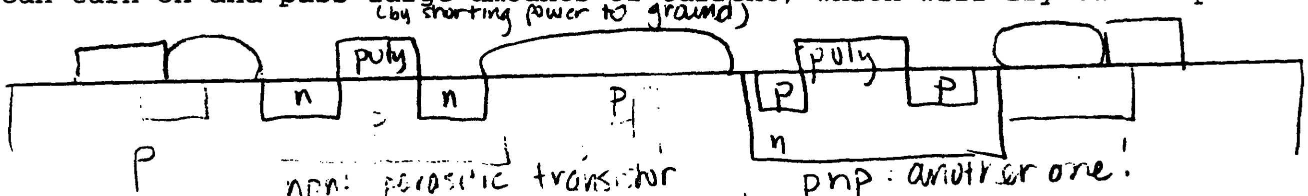
an example of this is under examples, std_ nand2

note how the n-well is tied to 5v by the n-sub, and the p-substrate is grounded by the psub: the n-well overlaps the vdd! metal, and then a 4x4 square of nsub is laid down over the intersection, and a normal 2x2 contact of active area (cont_aa) is used to connect the whole thing together. in the case of the psub, a 4x4 square of psub is place on the the ground connection, and a 2x2 square of contact is put over that. ex:



you put at least one n/p contact for each well.

latchup: is something that results in power being shorted to ground, which causes a huge current to flow through, frying the chip. this is a higher order effect that has to do with "parasitic transistors" (don't worry we students are immune to them). these parasitic transistors are bipolar transistors (npn or pnp) that result from the sandwiching of a layer between it s "opposites". anyway, what happens is that at a high enough voltage, these little transistors can turn on and pass large amounts of current, which will fry the chip. (by shorting power to ground)

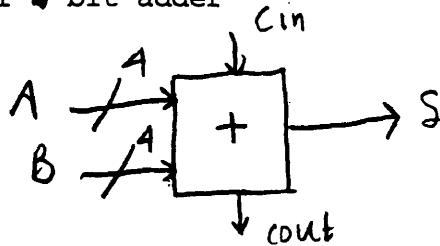


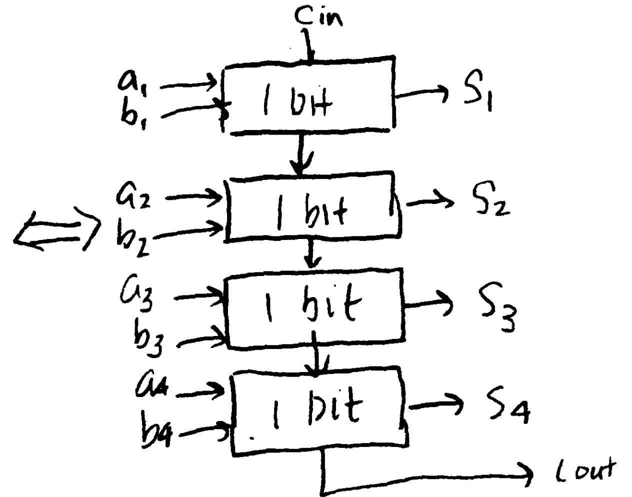
to keep this from happening, make sure that no transistor is more than 50 lambda from a contact. also tie the substrate to ground to ensure that these little parasitic transistors can't be incarnated.

some basic rules about psub and nsub:
- they must be 4 away from n or p diffusion, and 5 away from gates (polysilicon) which cross diffusion.

4 bit adder case study:

logic design for 4 bit adder

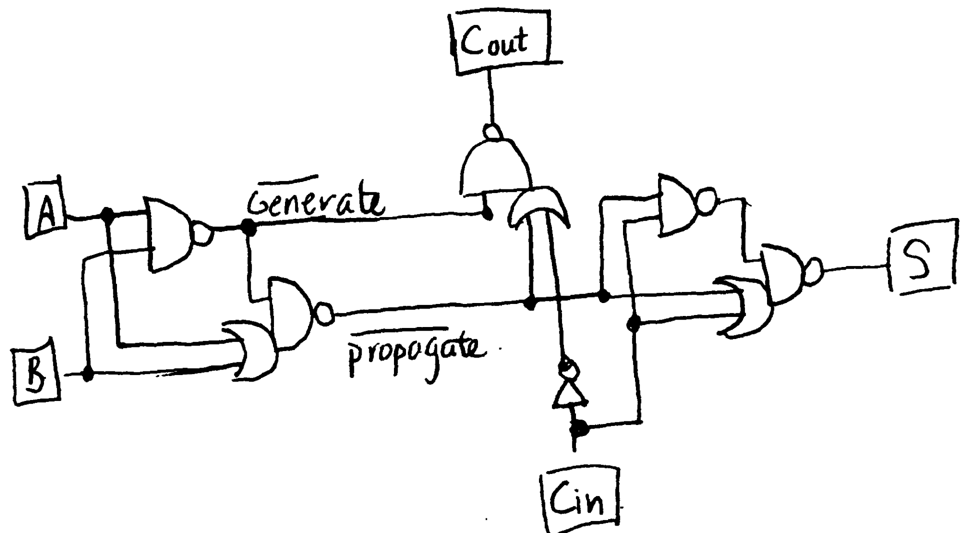




truth table for adder:

a	b	cin	s	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

cout = ab + bc + ac
s = abc + (a+b+c)cout'



some explanations:

generate (a carry): true when cout will be forced high independent of cin

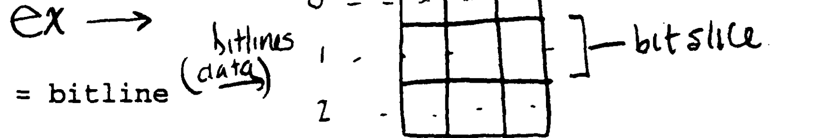
propagate (a carry): true when cout is true if and only if cin is true

maybe a truth table will clarify things a bit:

a	b	g	p
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

g (generate) = a & b
p (propagate) = a xor b

looking at secret agent 6.007 full adder datapath - example of implementation of adder as a datapath. one cell is repeated four times to make a four bit adder.



rules for datapath :

1) databits flow through metal 2 = bitline

2) power, ground, control (e.g. cin, select, on a mux) through metal 1 = wordline

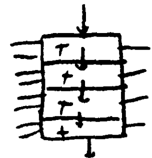
3) each cell has to pass on its values to next cell, etc.

4) must be 80 wide

to see a fine example of datapaths, check at your neighborhood cadence library for the fulladder cell under examples, lecture3
and it's only 8160 lambda squared! what a bargain!

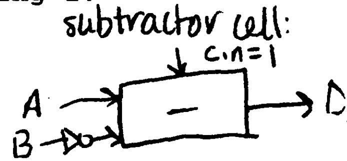
some new ideas:

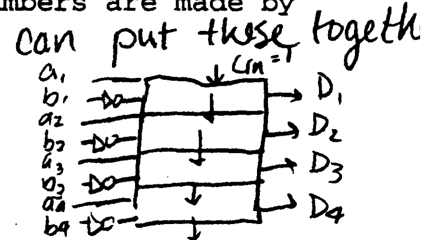
1. hierarchy : to build up large structure out of small pieces
- example - out little 4 bit adder



- or, say you wanted to build a to build a subtractor - one way to do it would be to build a subtractor cell, and then put them together, to make an n bit subtractor, etc. an easier and better way would be to realize that in bit arithmetic, negative numbers are made by inverting a bit and then adding 1.

so, a - b
 = a + -b
 = a + (b' + 1)
 ↑ those are pluses, not ors.
wow! it's that simple!





2. regularity: make cells reusable - like make one multiplexer that everyone can use

3. modularity: make modules that can plug together - interchangeable parts, like legos. you could plug any two and they would fit together.
in this case, you would make all cells the same height, but this standardization has its down side, because you could end up with inefficient cells - those that don't use all the space that is availiable to them.

4. locality: keep information that works together physically close together

using the tools :

pins:

pins help incorporate the ideas of hierarchy and regularity.
they show where levels should be connected.
there should be one pin for each input and output. (pins on both ends)

how to make a pin:
1.- under create of cadence - create pin , or just push p.
2. pick a terminal name - use lowercase to be consistent because cadence is case sensitive
3. draw 4x4 rectangle - make sure it's in the same layer as the terminal for which you are making a pin

properties: can find out also about connectivity
to use this, click on a pin, hit q (but not too hard), and pick connectivity.
cadence will tell you what it's connected to.

labels:

to make a label:
1. select labels layer in pallette
2. hit l for label
3. pick a name - for vdd and gnd the standards are vdd! and gnd! - make sure the name agree with the one assigned to the pin
4. height - choose a good one for readability - usually 2

something of interest to note:
the way cadence assigns labels is that if a label covers several different layers, the drc checker would first check to see if a metal2 exists for the label to be linked with, and then metal 1 and then the poly.

you can also make instances of cells:
instances:

see under examples, lecture3, adder4, layout

an instance of a cell is just like a copy of it. you can rotate it, flip it, paste it, etc.
when using them, if you even get confused, just look at the bottom of the window, and the cadence will show you some things you can do with the mouse buttons.

to use instances:
choose copy and select cell.
the right mouse button rotates the cell.
shift plus the right mouse button flips it.

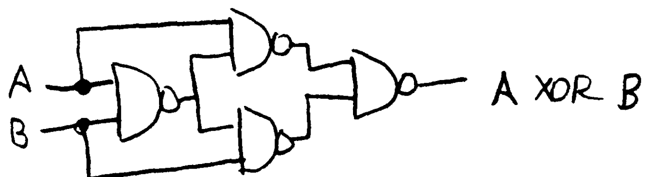
- when sticking instances together, overlap the pins so that they look like one pin.

standard cells

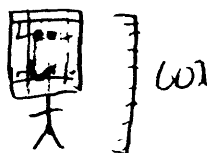
neat fact: nand is a universal function - from this, you can make anything!!

wow!

xor from nand;



a standard cell has a height 60



<u>to stick cells together:</u>
1. create instance - std_nand2
2. stick pins together

crtl c gets out of mode

6.008 Lecture 4: Memory
Monday, January 10, 1994
Lecturer: Bill Dally
Scribe: Ethan Mirsky

The Latch

All electric circuits have a parasitic capacitance associated with them. A transmission gate (Figure 1), can be modeled with a parasitic capacitor (Figure 2).

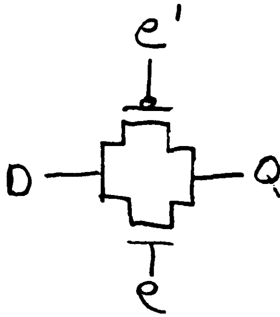


Figure 1

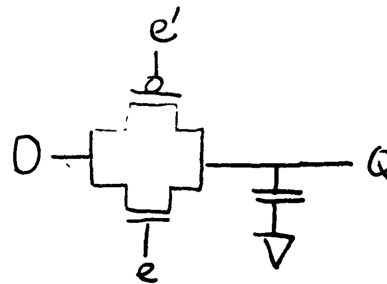


Figure 2

When the capacitor is charged up (when D and e are driven high) and then disconnected (e is driven low), the capacitor will retain its charge for anywhere between 1 ms and 1 minute, depending on the size and composition of the materials used in the circuit as well as the temperature of the circuit. For the time that the capacitor is holding its charge, it can be said to be remembering the value of D. However, in order to prevent the decaying charge from losing its value, the charge must be reinforced. Figure 3 shows a circuit which will perform this function.

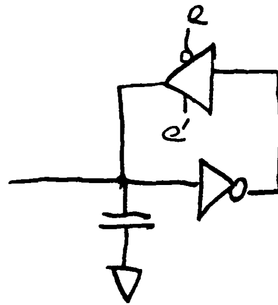


Figure 3

The top device in Figure 3 is a tristate inverter. A diagram for it is shown in Figure 4:

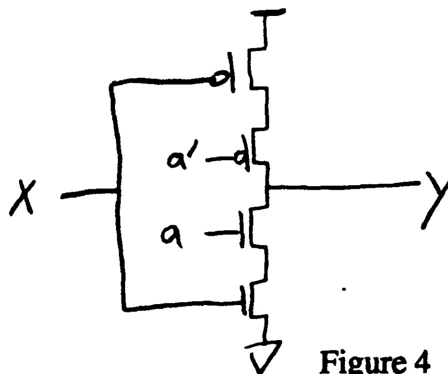


Figure 4

When a is high (a' low), Y becomes X'. When a is low, Y is disconnected from X. When the feedback loop of Figure 3 is attached to the transmission gate the circuit becomes a static latch (Figure 5).

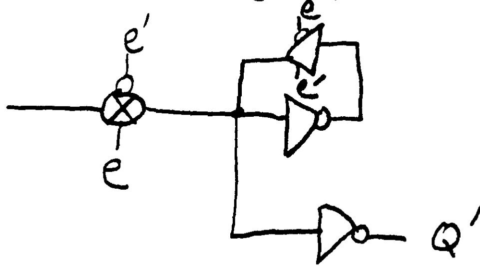


Figure 5

When e is high, the value at D is driven to Q. The tristate inverter is turned off, so that there is no conflict between the new value and the old value. When e is low, the D is disconnected from Q, but the value on the capacitor is maintained through the feedback loop, and thus the circuit "remembers" the value of D. The output inverter is used to isolate the circuit from the devices it is driving so that the capacitor does not get drained by a large load. Figure 6 shows a timing diagram for this process.

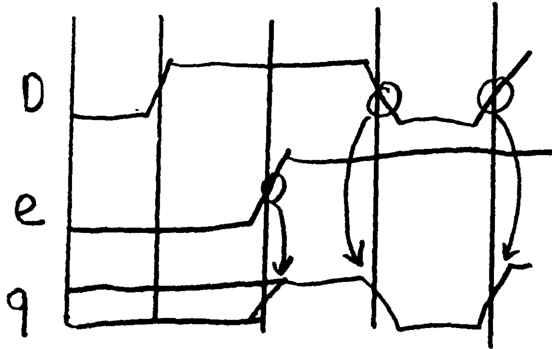


Figure 6

The actual storage is done with energy and it is necessary for the circuit to have energy storage (e.g. capacitance), even if it is only parasitic (there is no explicit capacitor built in).

In order to reset a latch (to set its value to low without forcing the input to zero), the following circuit can be used. Note: the symbol ⊗ refers to a transmission gate.

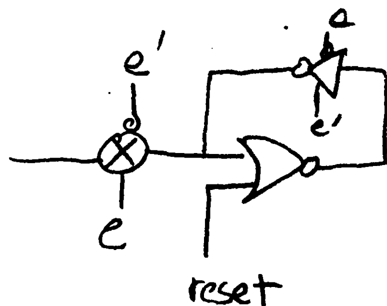


Figure 7

The Flip-Flop

In order to make a sequential circuit (one that works in response to a clock) a memory element needs to have two latches built in, as shown in Figure 8. Normally such a circuit is represented as shown in Figure 9.

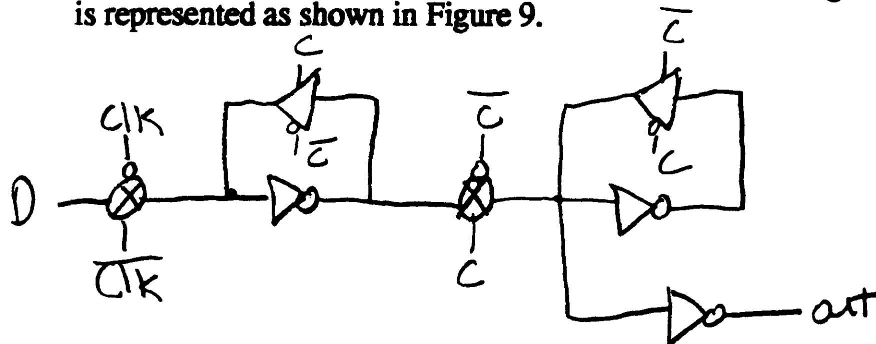


Figure 8

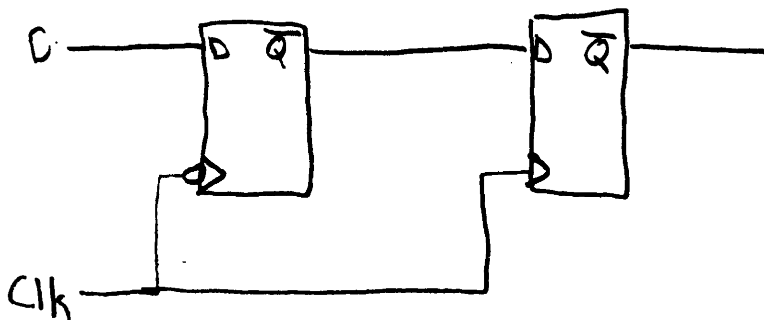


Figure 9

When the clock is low, the master latch is open, but the slave is still outputting the old value it is holding. When the clock goes high, the master closes, continuing to output the old value of D. The slave opens, and the old D is being saved and can be read on the output. This circuit is static on both clock high and clock low which means that the clock may be stopped either high or low without the output changing. If the tristate feedback loops are removed, the circuit will still work as long as the clock is maintained at a high enough frequency. This saves 10 transistors (four each from the tristate inverters and 2 from the output inverter.

In order to reset a flip-flop, add a nor gate before the final latch, as shown in Figure 10. This reset is synchronous (e.g. it happens only at the next clock edge).

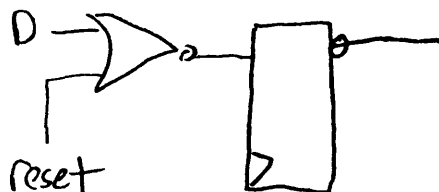


Figure 10

Timing Constraints for Sequential Circuits

Flip-flops have setup and hold times which constrain the timing of their operations. If a input signal is changed after the setup time before the clock edge or before the hold time (see Figure 11), the output of the flip-flop may become meta-stable. The meta-stable state occurs when the output is stuck on the threshold between logic high and low. When an output is meta-stable it will be corrected on the next clock cycle but, since its output could be read by other circuits as either a high or a low circuits following the meta-stable one will often do the wrong things. This situation must be avoided.

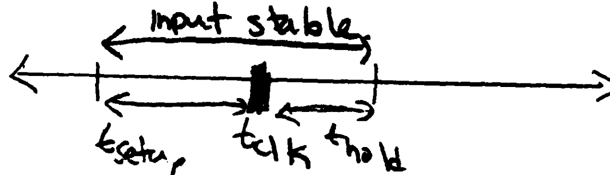


Figure 11

A common cause for this problem is clock skew. Clock skew occurs when the clock signal arrives at one device significantly before or after another. If these devices are supposed to work simultaneously, this could result in signals changing inside the setup time of the second device. The following constrain equation must be obeyed:

$$t_{skew} + t_{hold} < t_c + t_{pd}$$

where:

t_{skew}	is the difference between the arrival times of the two clock signals.
t_{hold}	is the hold time of the flip-flops involves.
t_c	is the contamination delay of the flip-flops. This is the delay after the clock edge before the out put of the flip-flop starts changing.
t_{pd}	is the propagation delay of the flip-flops. This is the delay after the clock edge before the output of the flip-flop stops changing.

For a more detailed explanation of this constraint, refer to Computational Structures, by Ward and Halstead, section 4.4.

The Counter

A simple device made out of flip-flop is a counter. A counter will count, in binary, using one flip-flop per bit. Each bit changes if all of the bits so far (starting at the least significant bit) have been ones. A synchronous counter is designed with each bit having the following circuit:

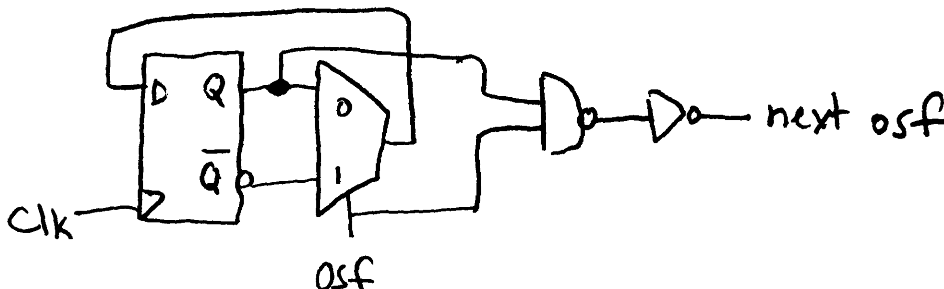


Figure 12

The first osf (one so far) bit is always high. The timing on such a circuit looks like:

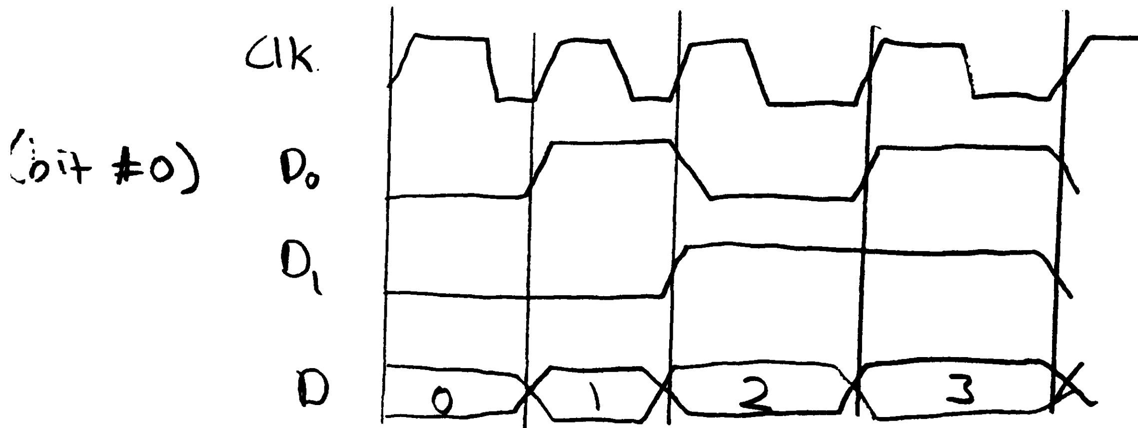


Figure 13

The Coke Machine (A Finite State Machine)

A state machine is device which can perform almost any function. It works by saving a "state" in flip-flops. The values in these flip-flops and the inputs can be used to compute outputs and the next values in the flip-flops (the next state). As an example we will make a coke machine. It will take as inputs lines representing coin inputs: "nickel" and "dime", and a clock. Its outputs will be "dispense coke" and "nickel out". A coke costs $0.15. The state diagram looks like this:

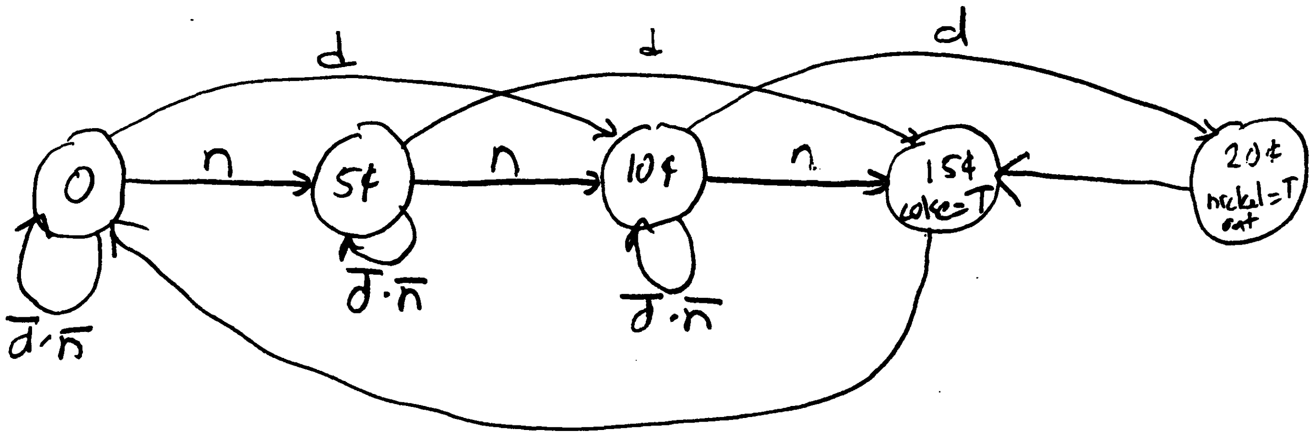


Figure 14

A direct implementation of this can use one flip-flop to represent each state. This is called "One-hot". It looks like:

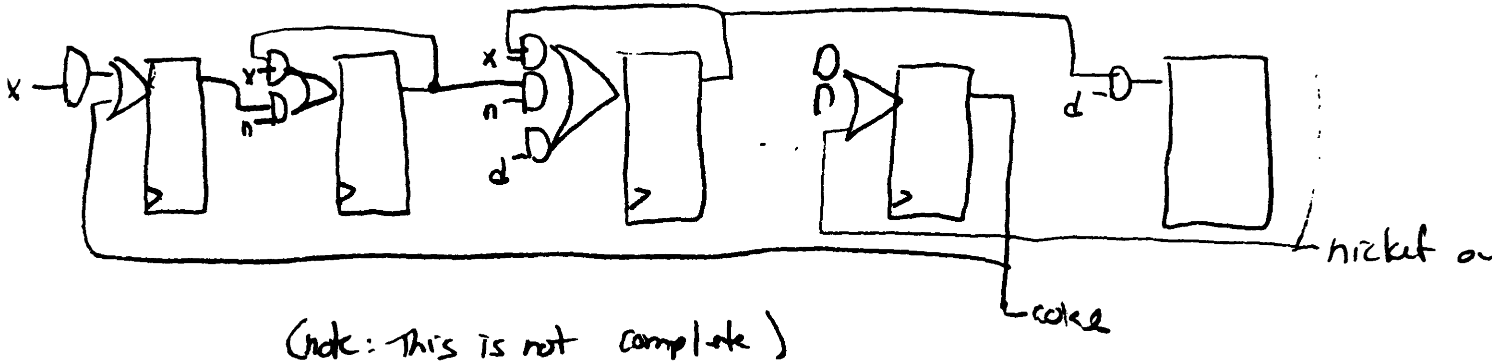


Figure 15

However, there is a better way. Each state can be encoded, so that only three flip-flops are needed. For example, we can encode the states as follows:
\$0.00 = 000   \$0.05 = 001   \$0.10 = 010   \$0.15 = 100   \$0.20 = 101

All the information in the state diagram can be encoded in this table:

	NEXT (based on input)			OUT	
current	none	nickel	dime	coke	nickel out
000	000	001	010	0	0
001	001	010	100	0	0
010	010	100	101	0	0
100	000	---	---	1	0
101	100	---	---	0	1

This information can also be encoded into Karnough Maps:

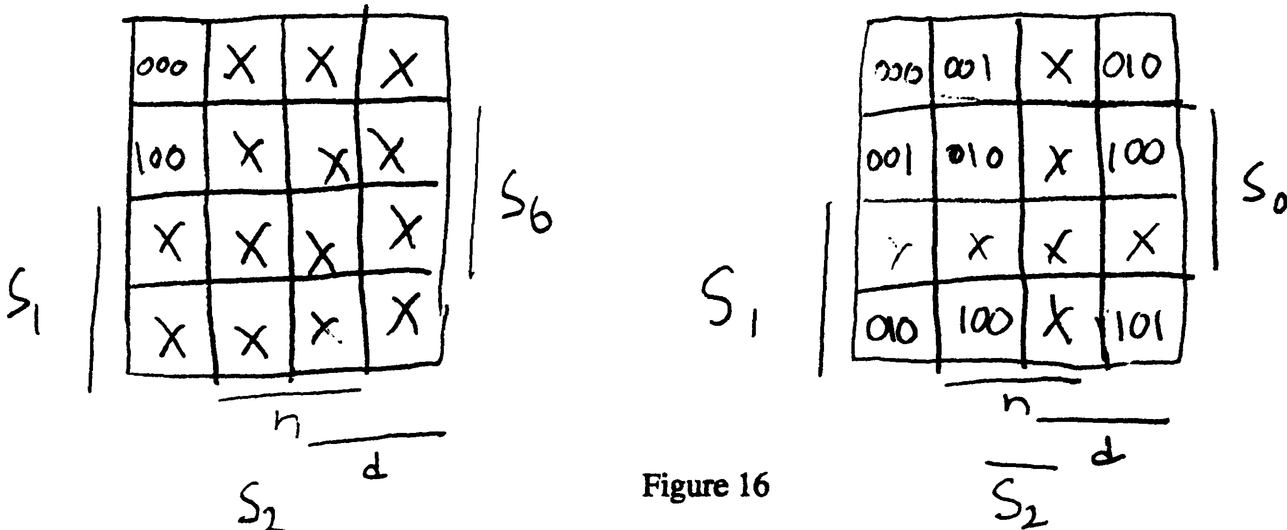


Figure 16

For an explanation of Karnough Maps refer to Computational Structures, by Ward and Halstead, section 3.3.

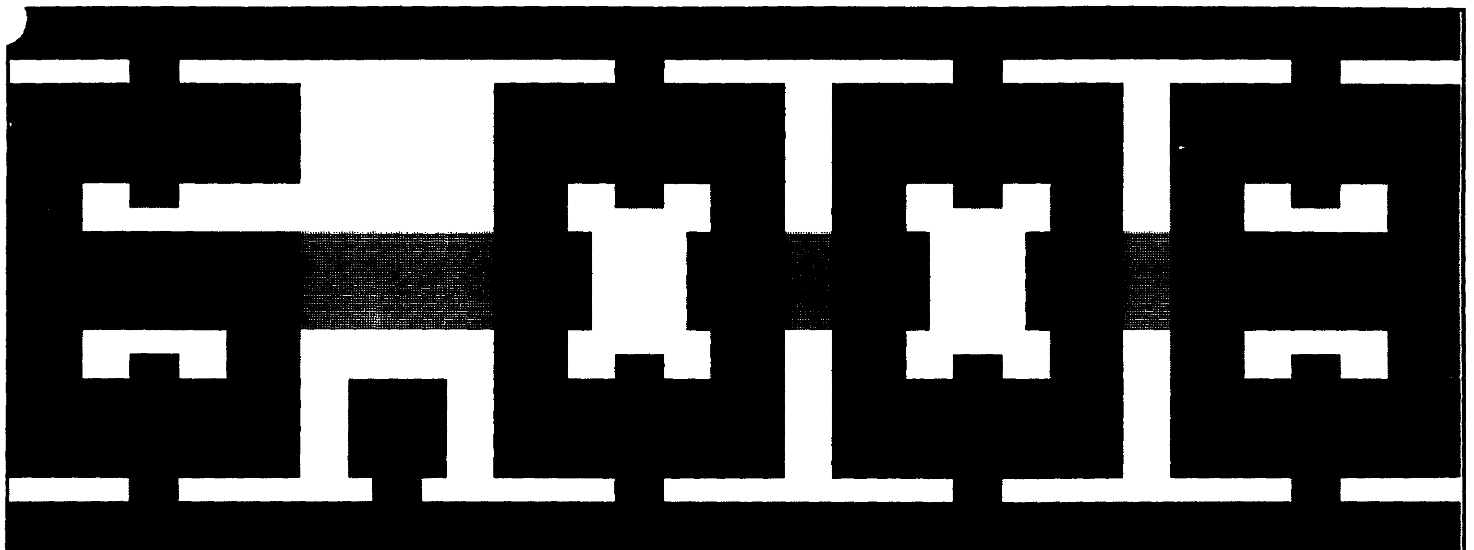
Taking each bit of state (one at a time), the terms can then be grouped into adjacent block. Each group must be a power of two. From this it is possible to write the logic equations for the states:

S0 (new) = S0*S2'*x + S0'*S1'*n + S1*d
S1 (new) = S0'*S1'*d + S1*x + S0*n
S2 (new) = S0*d + S1*d + S1*n + S0*S2

These are in the "Sum of Products" form. This form is easy to implement in PLA's (Programmable Logic Devices). However, to implement this logic in CMOS gates directly, the terms of S2 could be combined:

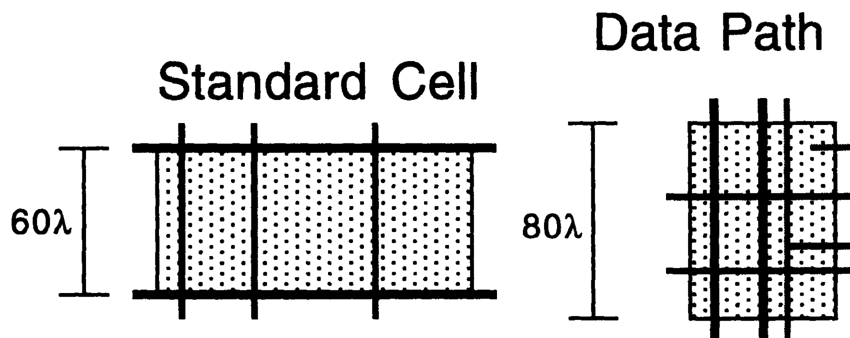
S2 (new) = S0*(d + S2) + S1*(d + n)

Next time: PLA's!



ARRAYS!

So far, we've learned everything there is to know about standard cells and datapaths.[1] To review, standard cells are laid out with horizontal power strips separated by 60 λ with inputs and outputs running vertically; this makes it straightforward to hook cells together to form arbitrary logic circuits. Datapaths, on the other hand, are 80 λ high, have vertical power strips and inputs coming from one side and outputs out the other; this is useful for designing dense circuitry which gets reused on multiple bits of information in which only local communication is necessary[2].

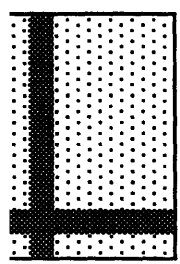


---

[1] Okay, so that's a *slight* exaggeration!

[2] Like the datapaths of a arithmetic unit

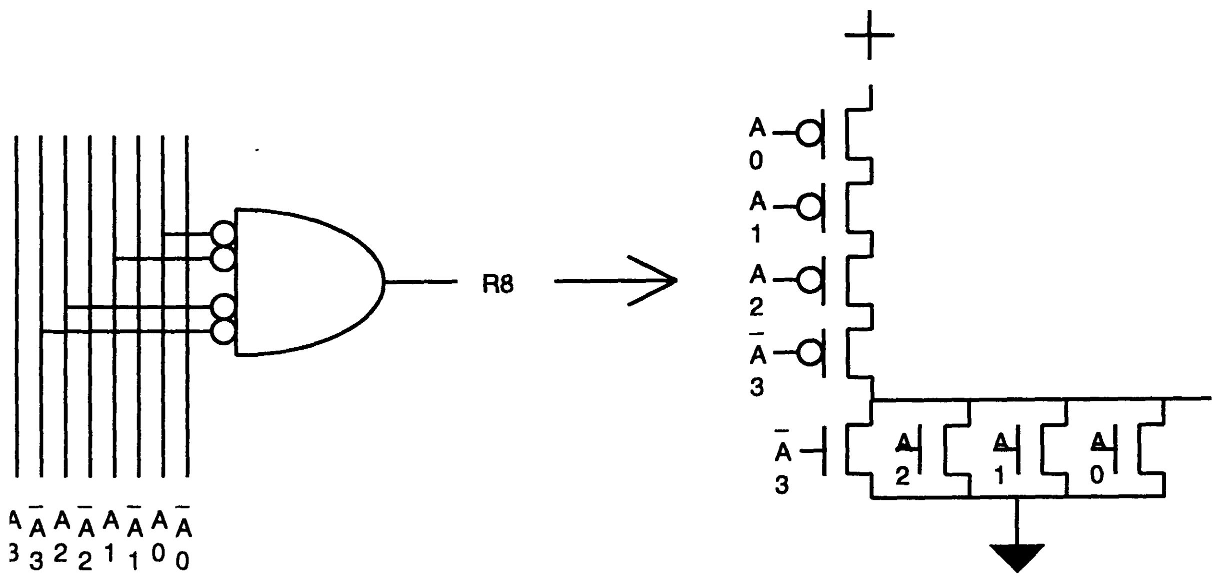
There are, of course, other systematic ways of designing circuitry. One common type of logic circuit is the so-called Read Only Memory. ROM sizes are usually indicated by something like 10 words by 7 bits, where the 10 represents the number of separate addressable locations (requiring $\lceil \log_2(\text{words}) \rceil$, or in this case 4 address lines) and 7 represents the number of output signals coming out of the array. Because of this structure, arrays are an excellent candidate for laying out in a grid fashion.

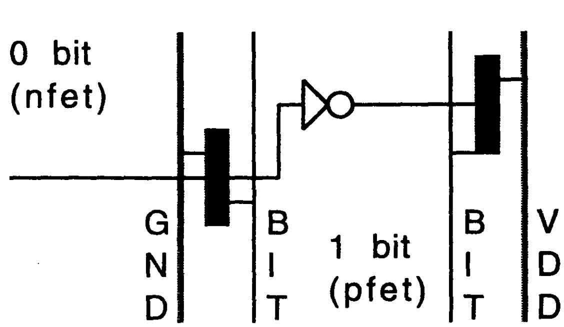


ROM Array for 7 segment decoding.

The row represents the number to display (the word), and the column contains the information for each segment (a bit line)

Row selection is done by a decoder matrix which runs vertically next to the array of bits. At each row, a horizontal row select signal is generated by a inverted-input AND gate (in this case, a 4 input inverted AND, or NOR4). The AND's inputs are connected to vertically running lines of the address and inverted address bits.

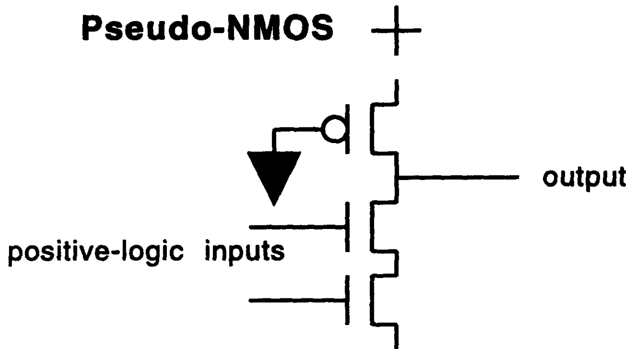




Then, at the intersection of the bit lines (running vertically), and the row select lines (running horizontally) is a transmission gate that allows a value to be driven to the bit line. Actually, a full transmission gate isn't necessary - only a nfet for driving a 0, or a pfet (with an invertor on the row select line) for driving a 1.

Unfortunately, this arrangement is pretty sparse - a pfet can only be so close to an nfet (there has to be a nwell overlapping the pfet by 5 λ in each direction, and a 5 λ spacing between ndiff and nwells). So, in order to more densely pack this array, pseudo-nmos logic is used. This is equivalent to using open collector devices and making

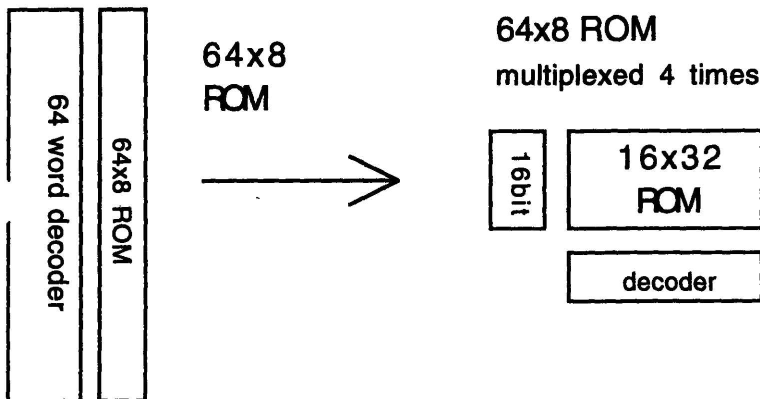
a wire or. Essentially, only the nfet is used (the absence of a nfet means a 1), and a weak pfet is placed at the top of each bitline, 'pulling' it high when no nfets are pulling it low. The astute observer will notice that when an nfet *is* on, a voltage divider circuit is formed. However, Dave has assured us that a voltage no higher than .6v will occur, well under the CMOS threshold voltage[3].



Of course, if it works once, why not use the same trick again? So, the decoder NORs will be implemented using pseudo-nmos, thus allowing the decoders to be compacted as well.

It may not seem too important to make the decoders small (after all, you just need a decoder for every address, while you need a bit for the number of addresses times the number of bits), but suppose you have a 64 word by 8 bit ROM. Using the above technique, the decoder logic would take a significant portion of the total space. Another solution is necessary. The trick is to break the ROM into pieces, and select the bit line from the the pairs.

For example, using 2 of the address bits in the 64x8 example to do bitline selection will reduce the decode width to 8 lines wide (rather than 12), divide the height of the circuit by 4 [4], and multiply the width of the bit array by 4.



The optimal solution is generally a square array of bits. The example ROM shown in class demonstrated two other techniques. One was that the pullup pfet does not need to be on the same end of the row line was the decoding pull-down logic (the gate can be physically distributed). Another was that you can get away without a pullup pfet on the bit lines by putting it after the multiplexor. I'll leave this as an exercise to the reader if its not intuitively obvious why this is so.

There are a few problems with pseudo-nmos, though. Because a voltage-divider circuit is formed, there is static power dissipation (this is why people use CMOS instead, which only has dynamic power dissipation). In fact, it was claimed each pfet will draw .25 milliamps - if there were a 1000 bitlines, and there's a even chance of a bit being on or off, that's over 1/2 watt of power dissipation.

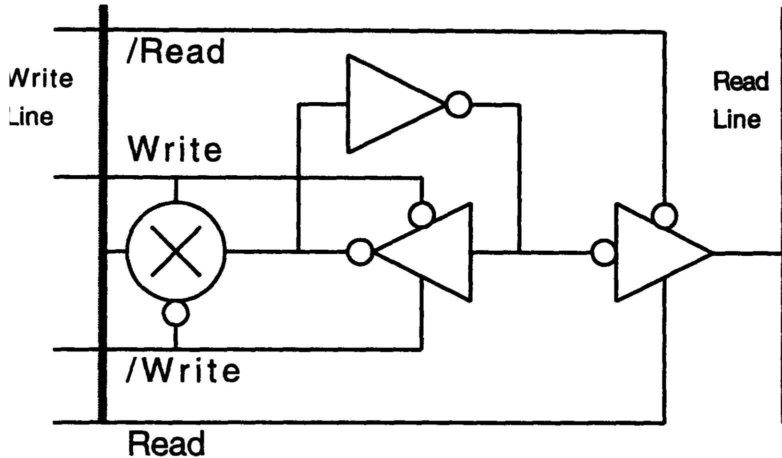
---

3 Of course, if he's wrong, we'll start by pulling out his fingernails...

4 Actually, some room at the bottom needs to be reserved for the multiplexors, but this is relatively small compared to the savings in decoder logic for this example.

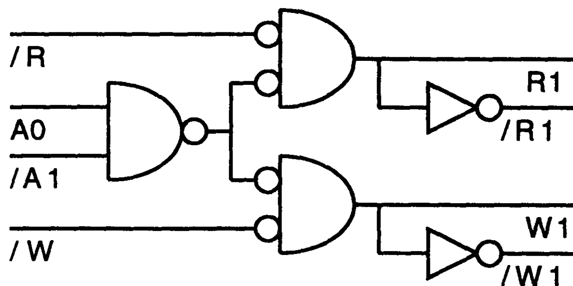
Another problem is that every row line except the selected one will be at a 0 level, thus drawing power. This can be easily fixed by using NAND gates instead of NOR gates at the decoder level, and then inverting the output using a CMOS invertor (with no static dissipation) to generate the 0's. One has to remember that now the inputs to the decoder gates must be the uncomplemented versions rather than the complemented ones[5].

Now that we have Read Only Memories, it would be nice to have memories that can be written, the poorly named Random Access Memories[6]. The basic bit cell consists of a latch with a tristate invertor on the output; a tristate inverter must be used rather than the multiplexor, or the current drain from all the attached gates will probably screw up the value stored in the feedback loop.



RAMs are described in more detail in Section 8.3.1 of Principles of CMOS VLSI Design by Weste and Eshraghian

One notices that now four row lines are needed for every row - Read, /Read, Write, and /Write. These can be generated fairly easy with a n-input NAND gate (where n is $\lceil \log_2 ROWS \rceil$ - the number of address bits) feeding two inverted-input AND2 gates, which then feed two inverters.



One problem with RAM is that the method of chopping up the array to save decoder space isn't quite as simple as it is in a ROM. The problem is you end up writing a whole row, which will consist of more than one word. One strategy is to require a read cycle before a write cycle, storing the whole word in a buffer row then only changing the bits necessary.

The final array type discussed is that of a Programmable Logic Array[7]. Basically, one takes the idea of a ROM array, and makes a few observations. First, the decoder unit is actually acting like an AND gate - by hooking it up to some set of the address lines, one can form any type of product term (like A*/B*C). Second, the array is acting like a NOR gate, pulling the bit low when any of the row lines that are hooked up are asserted.

---

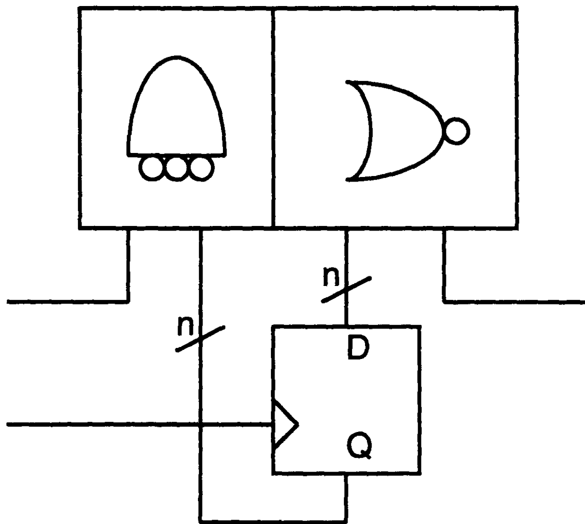
[5] ROMs are described in more detail in Section 8.3.3 of Principles of CMOS VLSI Design by Weste and Eshraghian

[6] Technically speaking, ROMs are random access - what they meant by the name is beyond me.

[7] This sounds like a misnomer to me - maybe automatic logic generation? You certainly can't reprogram the chip once its programmed!

Using these two observations, one can make any sum of products terms that one wants (NOTE: they will be in the form of an inverted sum of products). The number of inputs required determines the number of decoder lines. The number of outputs required determines the number of columns of the array. Finally, the number of separate implicants used determines the number of rows of the NOR array.

One useful application of PLAs is in the making of Finite State Machines. To make a quick and dirty FSM, one determines the sum of produces necessary to implement the state bits and output signals, puts them into a file, and runs Dave's nifty program to spit out a layout. Then, you simply hook the PLA generated to a set of Flip-Flops, add a clock, and you're done!



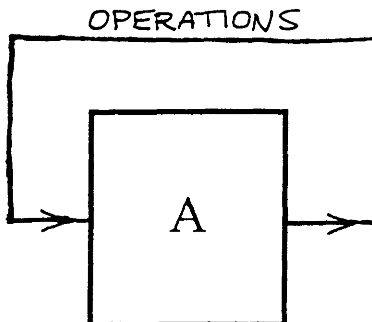
PLAs are described in more detail in Section 8.4.2 of Principles of CMOS VLSI Design by Weste and Eshraghian

6.008
Lecture #6:
THE UNINTEL SEXIUM
or...
FUN THINGS TO DO WITH REGISTERS

Lecturer: David Harris
Dutiful Scribe: Jeff Bowers

1 The accumulator

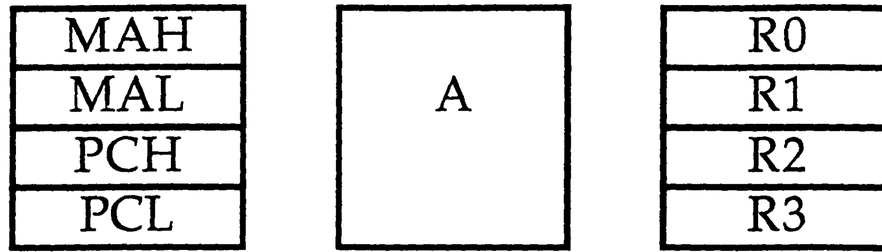
The Unintel Sexium is an accumulator machine, i.e. it uses a device called an accumulator to perform all of its basic operations. Most basic operations input the value stored in the accumulator, perform some operation, then output a new value to the accumulator. This can be illustrated graphically as follows:



For example, when you want to add two numbers, you add one number to another which is already located in the accumulator, and the sum is then placed in the accumulator.

2 Registers

A register is a device used by the Sexium to store information which is being processed. The accumulator is one example of such a register. The Sexium will include eight other registers: four general purpose registers labelled R0 through R3, two memory address registers MAH and MAL, and two program counter registers PCH and PCL. These are illustrated graphically as follows:



The two memory address registers are used to specify some location in memory. The Sexium can then read from or write to that location in memory. MAL is an eight-bit register containing the eight least significant bits, and MAH contains the six most significant bits. Because 14 bits therefore specify the memory address, there can be 2^{14} (= 16,384) different memory locations.

The program counter registers specify which instruction the Sexium is currently executing. After an instruction is executed, the program counter automatically advances to the next instruction (unless the programmer uses a BRA or JMP command, as will be explained later). PCL contains the 8 least significant bits of the program counter, and PCH contains the 6 most significant bits.

The accumulator and the four general purpose registers are also eight-bit registers. The general purpose registers are commonly used by the programmer to store temporary information and perform arithmetic operations.

3 Memory

The Sexium will have 2^{14}, or 16,384, bytes of memory. This memory will be partitioned into four "banks" each containing about 4K of memory. The first bank is reserved for ROM and must contain instructions for the processor; otherwise the processor will have nothing to process when it is reset (the program counter is set to $0000 when the Sexium is reset, and therefore the processor will look for instructions in memory location $0000). The second bank of memory will be RAM, and the third and fourth banks can be chosen to fit the user's purposes (they are presently undefined).

4 Programming the Sexium

In order to specify a set of instructions for the Sexium to perform, the programmer must write a program in assembly language. The instructions available to the programmer are minimal, but adequate for most needs. When programming in assembly language, constants are represented by

hexadecimal numbers (base 16). In base 16, the digits are from 0-F (0-9 and A-F). For example, the number 31 in base 10 would be written as $1F in base 16 (the number is preceded by a dollar sign ($) in order to signify that the number is in base-16). Also, negative numbers are expressed by taking the so-called "two's complement" of the equivalent positive number. This involves inverting each bit of the number and then adding 1. For example, the number $2F can be written as

00101111.

The number -$2F is found by inverting each bit and then adding 1, with the result that

-$2F = 11010000 + 1 = 11010001 = $D1

We call this number -$2F because $2F + $D1 = 256, which is zero on an eight-bit adder.

We can separate the instructions into three different types of operations: arithmetic operations, memory operations, and control operations.

Arithmetic Operations

ADD reg

This instruction adds the number in "reg" to the current value in the accumulator A, and stores the sum in A.

AND reg

This instruction performs a bitwise AND of A and "reg".

NOT

This instruction performs a bitwise NOT of A.

SHR

"Shift right": Divides A by 2 and drops the remainder. Can also be thought of as shifting each digit in the binary number over by one, and discarding the least significant digit, e.g.,

	SHR	
00100101	==>	00010010
($25)		($12)

ROR

"Roll right": Same as SHR, except that the least significant digit replaces the most significant digit:

	ROR	
00100101	==>	10010010
($25)		($92)

PUT reg
Stores the current value of A in "reg".

GET reg
Stores the current value of "reg" in A.

TST reg
Adds "reg" to accumulator, checks if there is a carry out, and checks if the sum is zero. A0 becomes true if there is a carry, and A1 becomes true of the sum is zero (modulo 256).

Memory Operations

LDA
Loads A with the value stored in memory location pointed to by MA

LDI
Loads A with the value stored in memory location pointed to by MA and advances MA by 1.

LDM const
Loads A with "const"

STA
Puts the current value of A in the memory location pointed to by MA

STI
Puts the current value of A in the memory location pointed to by MA and advances MA by 1.

Control Operations

JMP high low
Sets PC (program counter) to value specified in "high" "low".

BRA const
Advances or reduces value of PC by "const". Two's complement is used to move program counter backwards.

SKZ

Advances PC by 2 if A = 0.

BRK

Ends the program.

Instruction Sizes

Most of the instructions have a length of 1 byte. The exceptions are: LDM and BRA, which each have a length of 2 bytes, and JMP, which has a length of 3 bytes. The SKZ command will skip over an instruction of length 2 (or two instructions of length 1). If the SKZ command is followed immediately by a JMP command, then the program will crash because the program counter will encounter the argument of the JMP instruction rather than another instruction.

Some Examples

Suppose that you want to add the numbers $96 and $04. Here is a instruction sequence which would do this:

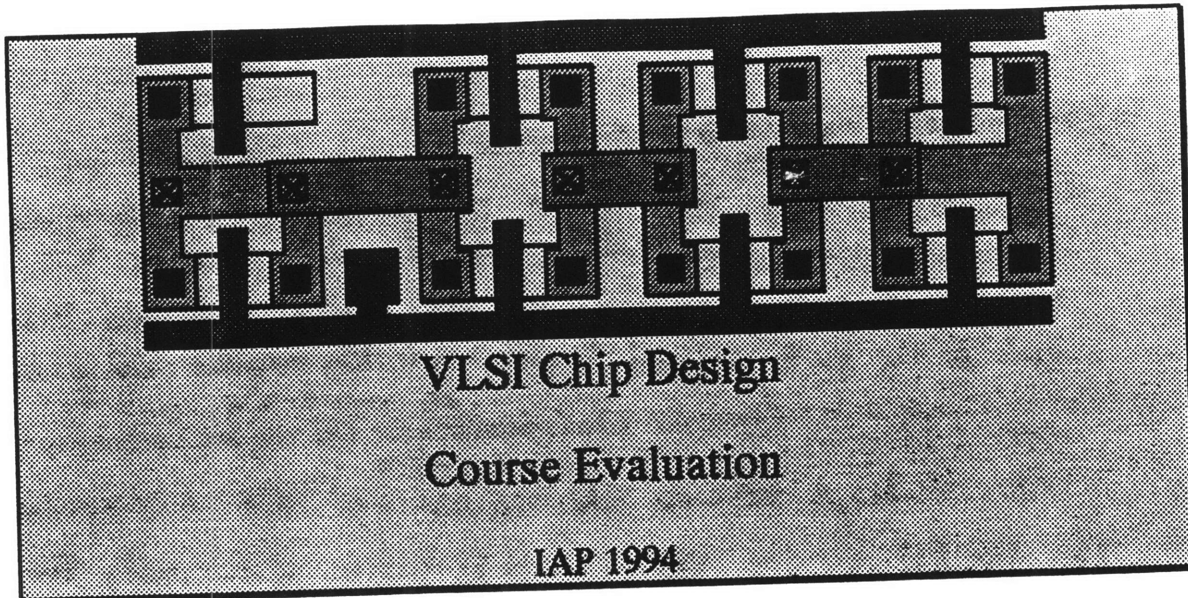
```
LDM $96    #Load the accumulator with the number $96
PUT R0     #Place this value in register R0
LDM $04    #Load the accumulator with $04
ADD R0     #Add the value in R0 to the value in the accumulator
```

Suppose that you want to load the value in memory address $0666 into R2. Here is an instruction sequence:

```
LDM $06    #Load the accumulator with $06
PUT MAH    #Set Memory Address High to $06
LDM $66
PUT MAL    #Set Memory Address Low to $66
LDA        #Load accumulator with value in memory address $0666
PUT R2     #Place this value in register R2
```

Appendix B: Student Evaluations

This appendix contains evaluations completed by the students on the last day of class.



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

It ~~should~~ seemed like a logical continuation of the things I had learned so far. I wanted to see what all that stuff in chip layouts really was.

2) What did you learn?

I learned how chips were manufactured, ~~what~~ how all those different components of a computer and an interface were fit into a small space, how transistors were made, how the wiring is done, how ~~com~~ powerful all the design software was and how involved and tedious building a chip can be.

3) What were the best and worst parts of the class? What would you do differently in the future?

The class was very well-prepared for a new class. The computer seemed to be pretty well-planned. The worst part was that there was so little time and everything had to be learned really quickly, and those with more experience seemed to dominate. More explanation of the software tools, common mistakes, misconcept Perhaps m can simul

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

The FSM problem set was the best next to the actual design. The least valuable was the programming one because we're not going to actually be doing that until the chip is working and somehow interfaced to the real world.

5) What did you like about the Unintel Sexium microprocessor design project? What would you do differently?

I liked having the freedom to actually build something like the Pentium scaled down several orders. Perhaps to actually do more explanation of the overall behavior, how each of the parts interact to make a working computer, and an emphasis on importance of regularity.

6) What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.

More linkage between the issues and the microprocessor. The lectures on chip fabrication were really too fast.

7) How much time did you spend outside of class between each lecture?

6-10 hrs.

8) Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?

I feel like I have a decent command of the material. I would be comfortable in a UROP. More about the physical issues and in that respect, I suppose it was less like a graduate course.

9) Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?

Definitely.

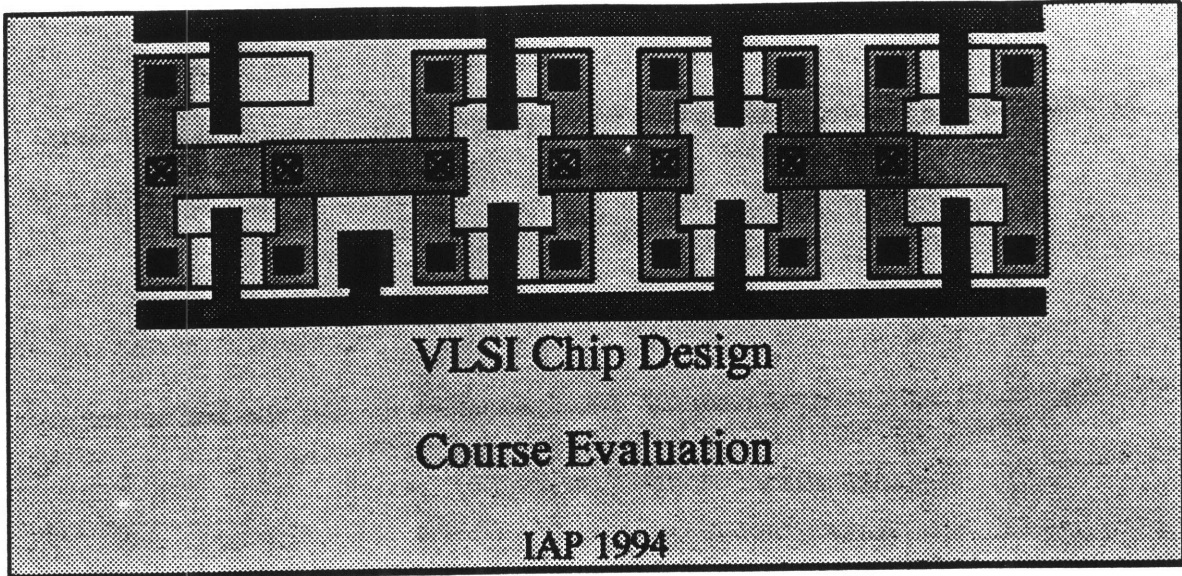
10) The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.

Designing ~~circuit~~ and making PC boards and perhaps something on interfacing computers to the outside world. I still don't see enough of that. May be some programmable logic also.

11) If this class were to be offered again next year, would you be interested in TAing?

Yes.

5x7
metal 1



VLSI Chip Design

Course Evaluation

IAP 1994

We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

Because it seems like good experience, good stuff to know for course 6, it was a sequel to 6.007 & Dave was teaching it. :)

2) What did you learn?

A lot of stuff: cells, layout, thinking neat ways, assembly stuff (which remind me of Basic), & how a mp works.

3) What were the best and worst parts of the class? What would you do differently in the future?

Best - learning stuff.
Worst - nothing really, I didn't mind spending a lot of time on the homework, I just felt bad cause I wanted to d[o] other th[ings]

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

I don't know, they were all kind of neat. It'd be fun to see the circuits change, I don't

178

5) **What did you like about the Unintel Sexium microprocessor design project? What would you do differently?** I can't believe we built a microprocessor! WOW!

6) **What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.**

the Lectures were fast, but it was okay, considering we barely had a month. David's follow helped a lot, as well as not having to write notes.

7) **How much time did you spend outside of class between each lecture?**

I don't know, ~ [illegible] - 12 hours (depending on the prob set.)

8) **Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?**

I guess so. I ~~'m not sure~~ understand how the whole thing & components work, but I wish could've seen them put it together, then I would felt ~~more~~ most comfortable w/ the material. I guess

9) **Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?**

Sure.

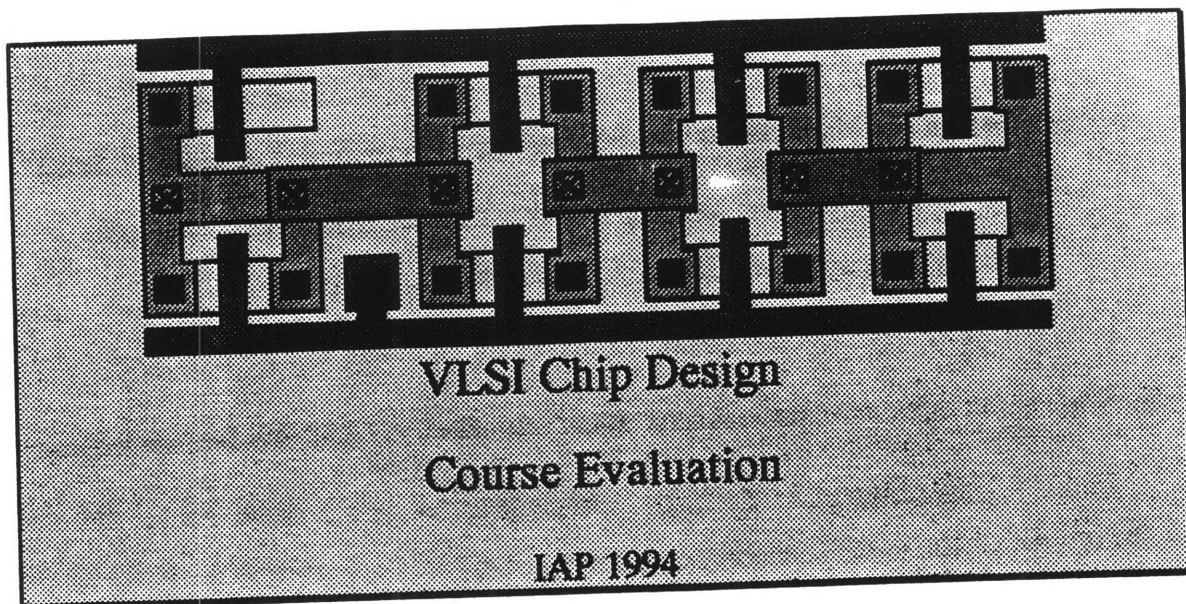
10) **The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.**

Oh, I can't think of anything cool.

11) **If this class were to be offered again next year, would you be interested in TAing?**

If I had time.

I might could it again, I don't know I can't think of anything more to u[illegible] maybe how exactly the [illegible] is incorporated to make a real computer?



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

I wanted to learn about microprocessors, and actually gain experience in designing them. I took the 6.007 seminar and it was really awesome, and this also looked really cool.

2) What did you learn? Lots! How to use Cmos transistors to implement logic, How to layout circuits using ~~[illegible]~~ Cadence, the basic parts of a microprocessor and how it functions.

3) What were the best and worst parts of the class? What would you do differently in the future?

The class was challenging and I really liked the design problems. I was sort of sad that I had to stay up until 3am working on prob sets during IAP

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

The standard cell library was my favorite. I liked trying to design the tighest cell possible. My least favorite was the adventure game I was pretty tedious.

180

5) **What did you like about the Unintel Sexium microprocessor design project? What would you do differently?**

I was amazed that we were doing something so complic yet everything was explained and there wasn't much that was left a mystery.

6) **What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.**

Sometimes I felt the class was a little fast. The best thing was to see other people's designs and also watching people as they desi a cell. This helped me get rid of some bad habits I picked up i the beginning.

7) **How much time did you spend outside of class between each lecture?**

6-7 hrs / problem set

8) **Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?**

I don't think I could say I mastered the material, but I feel pretty comfortable with it and I certainly would like to learn more

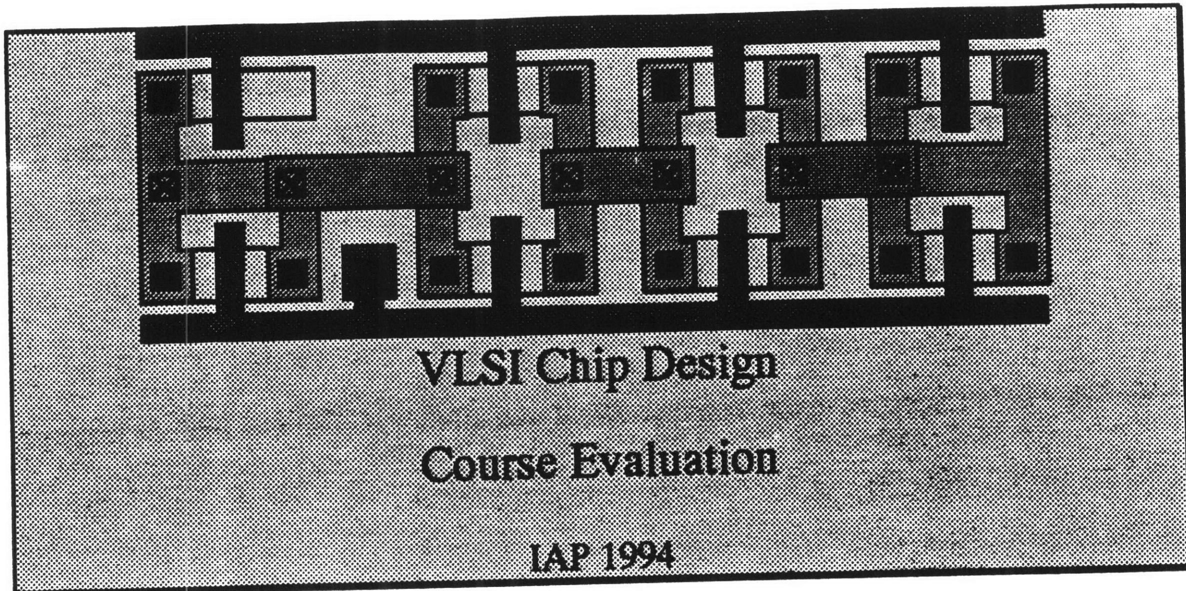
9) **Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?**

Sure.

10) **The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.**

11) **If this class were to be offered again next year, would you be interested in TAing?**

Sure, if I had time.



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

Because there wasn't enough room 6.371, so I got punted.

2) What did you learn?

Everything about VLSI except making it fast.

3) What were the best and worst parts of the class? What would you do differently in the future?

Best: Laying out the session, in fact, it was all great!

Worst: ______

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

Best: All pretty good

Worst: Adventure - just took too [illegible]

182

5) What did you like about the Unintel Sexium microprocessor design project? What would you do differently?

Make it RISC, nuke the PLA

6) What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.

Probably too fast for many except the seniors

7) How much time did you spend outside of class between each lecture?

~20 hrs/week, more at end

8) Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?

Definitely understand it all - speed would be nice to know but thats why 6.372 exists.

9) Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?

Maybe

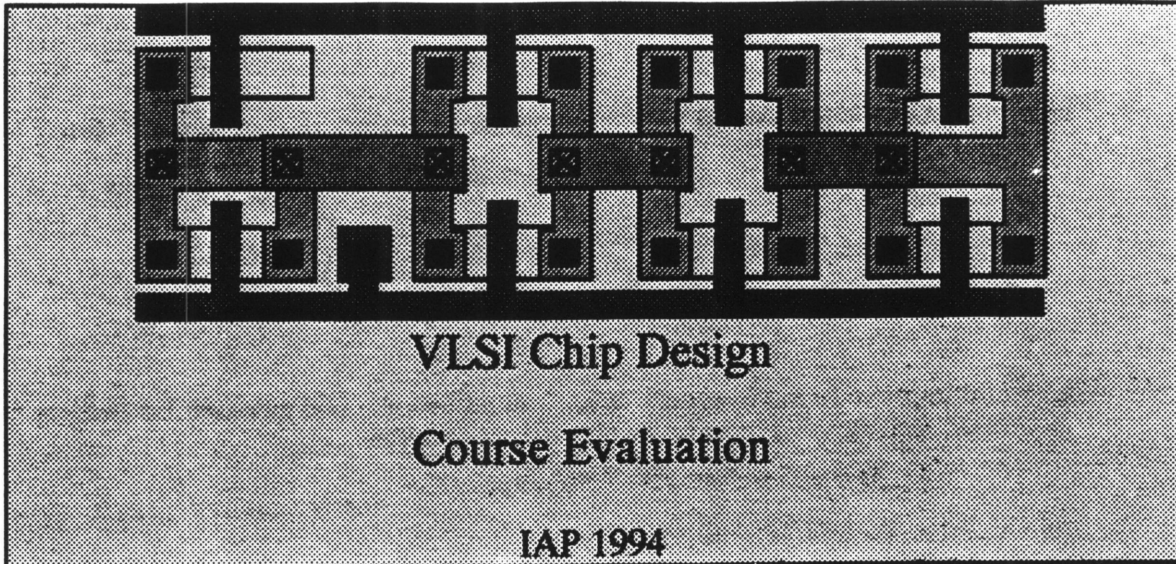
10) The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.

Not yet.

11) If this class were to be offered again next year, would you be interested in TAing?

Most Definitely

Rob Restoeph [handwritten]



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class? I was interested in the subject matter, and believed that knowing this stuff might lead to a urop.

2) What did you learn? The basics of:
- transistors
- chip integration processes
- logic
- microprocessor stuff

3) What were the best and worst parts of the class? What would you do differently in the future? best — working with other people
seein pla logic work
worst — it took up a lot of time —

4) Which problem set was your favorite? Which was least valuable / interesting? Why?
favorite — finite state machine —
least valuable — I'm not sure

184

5) What did you like about the Unintel Sexium microprocessor design project? What would you do differently? Maybe use a bigger chip? (joke)

Maybe a extra handout on physics of transistors, on optimization for speed?

6) What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.

Lectures were pretty good - no suggestions

7) How much time did you spend outside of class between each lecture?

about 6 hrs, average

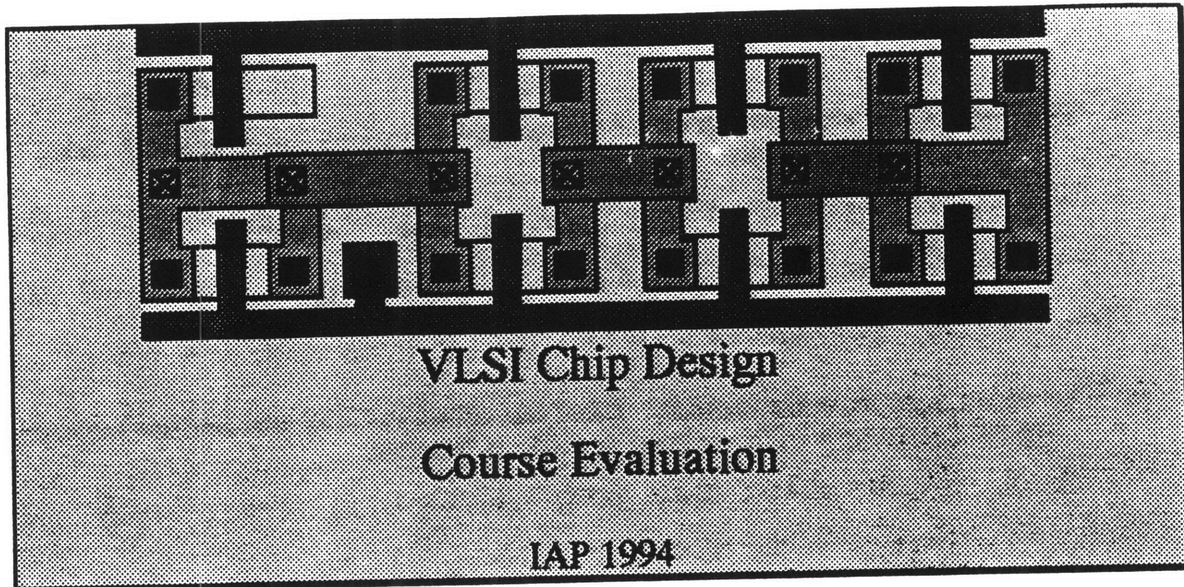
8) Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?

I felt comfortable, but the PLA stuff still seems a little magical. It all felt like

9) Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium? Yes

10) The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.

11) If this class were to be offered again next year, would you be interested in TAing? Yes



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

I wanted to learn how to do VLSI design.

2) What did you learn?

How to use Cadence Design tools, Constraints and technique for doing VLSI design.

3) What were the best and worst parts of the class? What would you do differently in the future?

Best parts: Doing Design, getting to see an actual project coming together

Worst part: Exceptionally long nights / work sessions

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

5) What did you like about the Unintel Sexium microprocessor design project? What would you do differently?

Have the complete design (specs) done before the layout begins. Leave more time for it. Maybe get a bigger chip and do a more modern architecture.

6) What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.

Having 6.004, 6.111 and 6.313 background made most of the lectures too slow. Maybe make 6.004 a prereq and teach more about VLSI rather than basic architecture.
↳ and advanced architecture

7) How much time did you spend outside of class between each lecture?

For the problem sets: 6-8 hours.
For the project: 1020 hours

8) Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?

I feel very confident doing VLSI design now, at UROP or at a company. This was like a grad subject, except that the really advanced stuff (like speed) were left out → because of the project-oriented approach

9) Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?

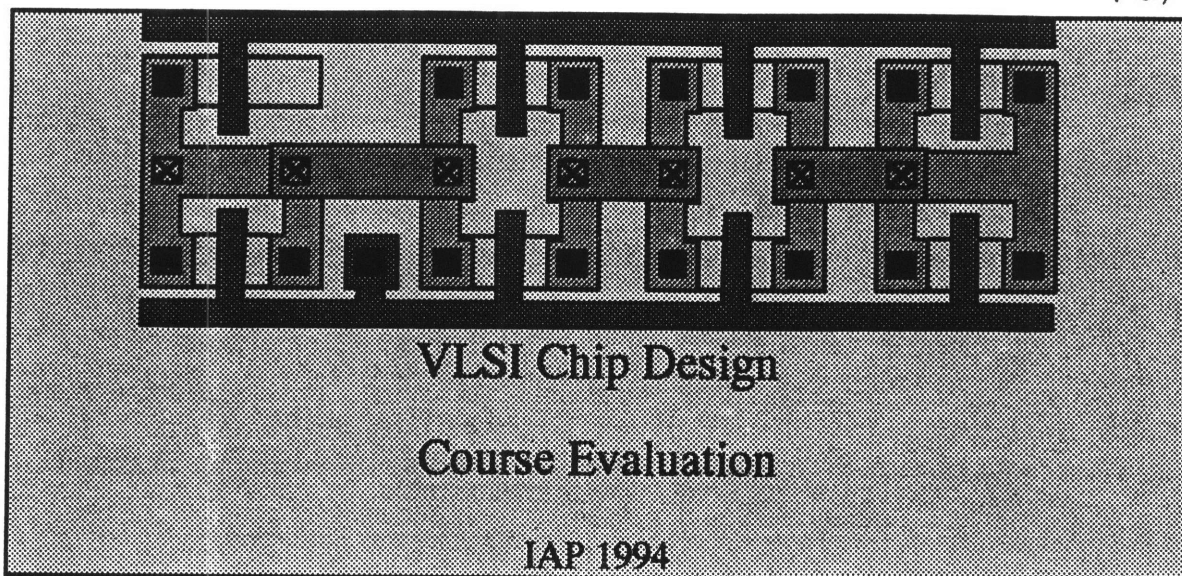
Yes, except that I have a heavy term up ahead...

10) The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.

11) If this class were to be offered again next year, would you be interested in TAing?

Maybe... again, its a time question. I would be interested

堺 高司



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

I did it to learn something about chip layout & manufacturing and also because I had nothing else to do over IAP.

2) What did you learn?

I learned that the pursuit of perfection is a dangerous and taxing endeavor.

3) What were the best and worst parts of the class? What would you do differently in the future?

The worst parts, I think, were lectures, since it was difficult for them to be "appropriate & relevant" to the divers skill levels in the class, and they tended to err towards the low side. The best part was the perpetual challenge of layout to be minimized in area.

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

I really liked the problem sets that involved layout of relatively complex logic gates (like full adder & flip-flop). My least favorite was probably the adventure game, which seemed like a lot of tedium to prove a relatively minor point.

188

5) What did you like about the Unintel Sexium microprocessor design project? What would you do differently?

I think the design project was well thought-out and it went together fairly smoothly. The only complaint I have is that towards the end, not more than 3-4 people could effectively work on the chip, so some people were left with nothing to do. It worked out, but this should be rearranged if possible, should the class be taught again.

6) What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.

Most of the lectures seemed to go at too slow a pace, but I think that was largely because I have seen most of the material presented before this. One major problem was that since lectures were held in an electronic classroom, it was very easy to become distracted from the lecture. The workstations weren't used very much for actual lecture demos.

7) How much time did you spend outside of class between each lecture?

6-16 hours, depending on the nature of the assignment

8) Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?

Hmm... How can I know ~~what~~ about what wasn't covered? I think I mastered the art of compact layout fairly well, but not so much of the higher-level rationales for making the design this way or that. It definitely felt like an undergrad subject. After all the material is not hard at all, to think that it might have been a graduate subject is unbelievable.

9) Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?

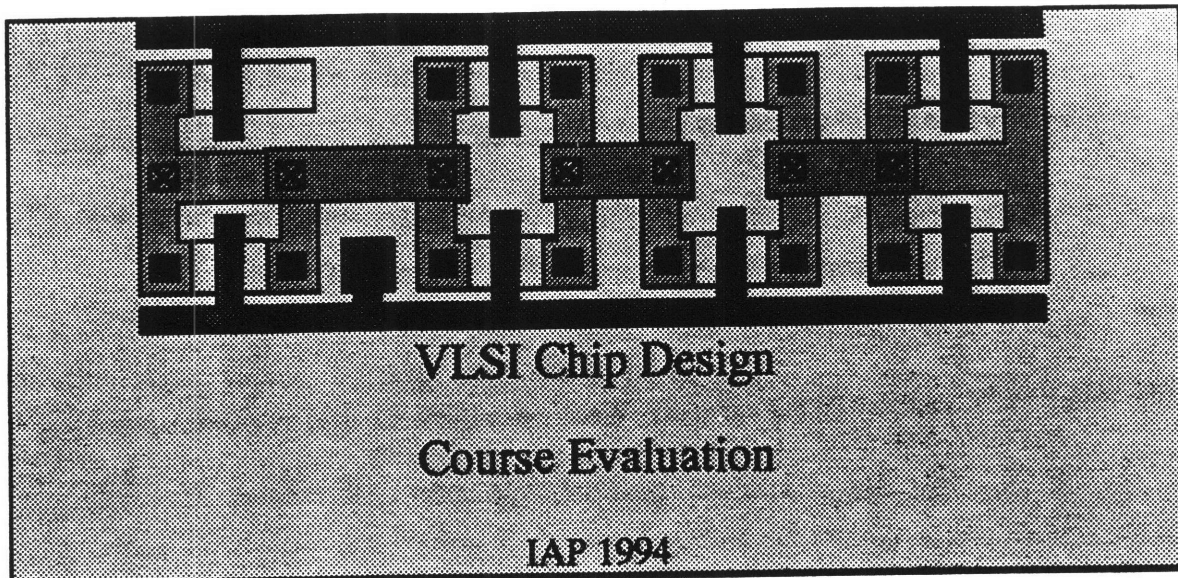
Yes, if I have the time to do it.

10) The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.

Nothing comes to mind...

11) If this class were to be offered again next year, would you be interested in TAing?

Not really, I'm not much of a teacher...



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

I wanted to learn something about how chips are designed and fabricated

2) What did you learn?

I learned that I'd like to study more VLSI, perhaps through a UROP or summer job. I think I think I understand the basics very well, well enough to attempt my own designs.

3) What were the best and worst parts of the class? What would you do differently in the future?

The problem sets were very long and challenging but necessary to understanding the material.

4) Which problem set was your favorite? Which was least valuable / interesting? Why?

I enjoyed the last problem sets when we actually started building the components of the chip. I like the design competitions.

190

5) What did you like about the Unintel Sexium microprocessor design project? What would you do differently?

I would have liked to know more about how all the components work, instead of just the components I designed. For all I know, the control unit is just a big black box and I know about the design of it.

6) What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.

The lectures were very fast but I don't think that can be avoided. It might be nice to prepare lecture notes which could be presented in a bound volume such as in 6.007

7) How much time did you spend outside of class between each lecture?

5-7 hours

8) Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?

I would enjoy doing a UROP or summer job in VLSI design. I feel that I understand the basics of it very well, but I would like to learn more about simulation, programming, and optimization.

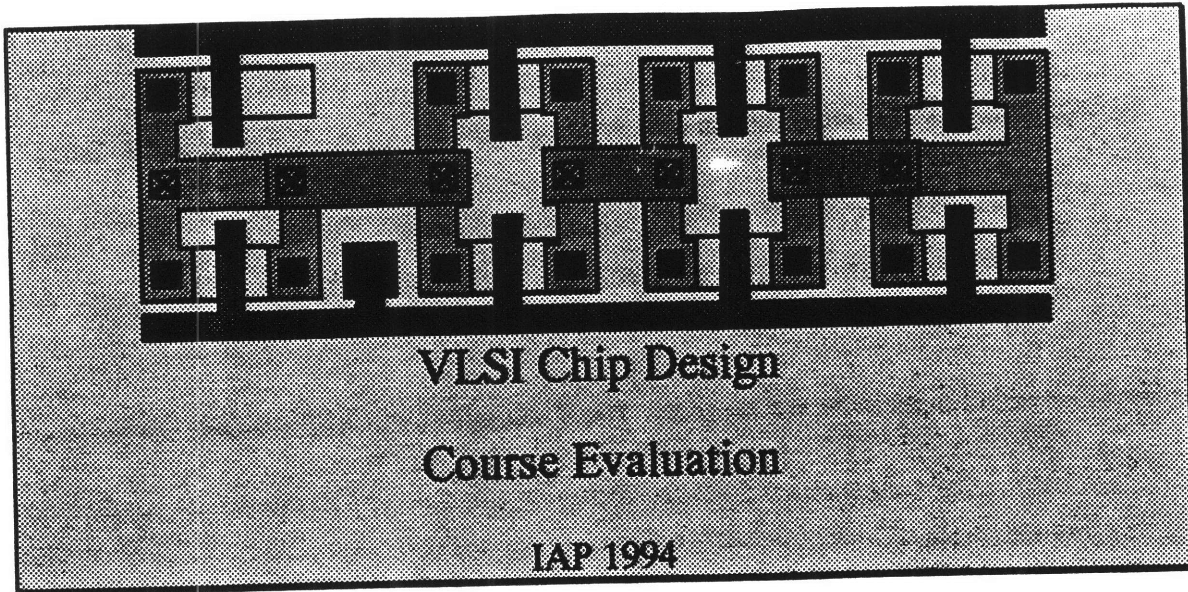
9) Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?

Definitely

10) The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.

11) If this class were to be offered again next year, would you be interested in TAing?

Yes although I'd probably want to have some more experience beforehand — maybe a UROP or summer job.



We have taught this class (6.090, VLSI Chip Design) as an experiment in bringing a traditionally graduate subject to the early MIT experience. We need lots of feedback to learn if the experiment was successful and to make improvements in the future, should we be offering it again. Please fill out the following evaulation in excrutiating detail.

1) Why did you take this class?

I TOOK THE CLASS TO CONTINUE MY 6.007 EDUCATION. IT ALSO SEEMED LIKE SOMETHING INTERESTING TO DO DURING IAP.

2) What did you learn? I LEARNED A GREAT DEAL OF INFORMATION ABOUT TRANSISTORS, CIRCUITS, FABRICATING A CHIP, ETC.

3) What were the best and worst parts of the class? What would you do differently in the future?

THE BEST PART OF THE CLASS WAS USING CADENCE. ONE OF THE WORST PARTS WAS THE PROBLEM SETS, BEING WAY TOO LONG IN THE BEGINNING. THE MAIN FLAW OF THIS CLASS WAS THAT IT ATTEMPTED TO COMPRESS TOO LARGE AN AMOUNT OF MATERIAL INTO A FEW LECTURES, PROBLEM SETS, ETC OVER 3 w

4) Which problem set was your favorite? Which was least valuable / interesting? Why? OF COU

THERE IS REALLY NO OTHER WAY TO GET THIS AMOUNT OF MATERIAL A STUDENT'S HEAD WITHIN A SHORT TIME PERIOD OTHER THAN TO CRAM IT, DOESN'T NECESSARILY GUARANTEE MASTERY. A COURSE SUCH AS THIS BE MORE FEASIBLE OVER THE SEMESTER RATHER THAN DURING IAP

192

5) **What did you like about the Unintel Sexium microprocessor design project? What would you do differently?**

I LIKED THE BASIC IDEA OF DESIGNING A CHIP USING CAD. THERE WOULDN'T BE ANYTHING SIGNIFICANT THAT I WOULD DO DIFFERENTLY.

6) **What could improve in the lectures? What was effective? Did you find the lectures too fast? Too slow? Please be as specific as possible.**

IF YOU PLAN TO TEACH THIS CLASS TO FRESHMEN WHO HAVE HAD ONLY "6.007" AND NOTHING ELSE IN TERMS OF EE, THE LECTURES NEED TO BE SOMEWHAT SLOWER. I FOUND IT DIFFICULT TO MOVE AT THE PACE Bill Daly WAS GOING AT AND OFTEN HAD TO PUT FORTH EXTRA TIME AND EFFORT TO UNDERSTAND THE MATERIAL.

7) **How much time did you spend outside of class between each lecture?**

IN THE BEGINNING OF IAP, I SPENT A LOT MORE TIME THAN I DID TOWARDS THE END, ONE OF THE REASONS BEING THAT IT HAD BECOME LESS ENJOYABLE AND DID NOT SEEM NECESSARY TO.

8) **Do you feel like you mastered the material? Would you feel comfortable with a UROP or summer job doing VLSI design? What would you want to know that wasn't covered? To what extent did we teach VLSI as an undergraduate subject, and to what extent do you feel like you took a graduate subject?**

ALTHOUGH I LEARNED A SUBSTANTIAL AMOUNT OF MATERIAL, I DO NOT FEEL THAT I HAVE "MASTERED" IT THOROUGHLY AND WOULD NOT FEEL VERY COMFORTABLE WITH A SUMMER JOB DOING VLSI.

9) **Would you be interested in a 6-unit follow-on subject this spring developing hardware and/or software for a computer built from the Sexium?**

No.

10) **The Edgerton Center sponsors hands-on seminars like this one, as well as hands-on UROP work. Do you have any ideas for other seminars the Edgerton Center should offer or UROP projects you would like to do? You may be interested in stopping by the Edgerton Center, room 4-409, and seeing what is happening.**

No.

11) **If this class were to be offered again next year, would you be interested in TAing?**

No.

Institute Archives and Special Collections
Room 14N-118
The Libraries
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139-4307

This is the most complete text of the thesis available. The following page(s) were not included in the copy of the thesis deposited in the Institute Archives by the author:

p. 193/194

Telephone: (617) 253-5690 • reference (617) 253-5136

Appendix C: Sexium Design Files

This appendix contains the various design files created for the Sexium project, including the Verilog model, the equations describing the microcode PLA, HSPICE simulations, the schematics, and the layout.

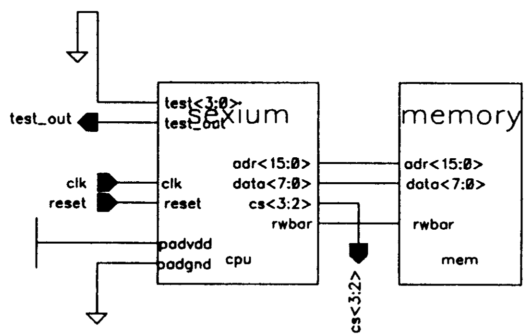
Sexium Schematics

Complete schematics for the Sexium microprocessor appear on the following pages:

computer 197
sexium 198
alubox 199
flop2 200
flop2bit 201
adder 202
adderbit 203
ander 204
anderbit 205
neger 206
negerbit 207
tribuf 208
tribufbit 209
mux5 210
mux5bit 211
mux4 212
mux4bit 213
mux2gnd 214
mux2gndbit 215
regbox 216
trilatch 217
trilatchbit 218
pcmabox 219
flop3 220
flop3bit 221
halfadder 222
halfadderbit 223
control 224
oplatch 225
oplatchbit 226
test 227
counter 228
sexiumframe 229
v4io 230
flopen 231

FSCM NO.	DWG NO.	SH	REV

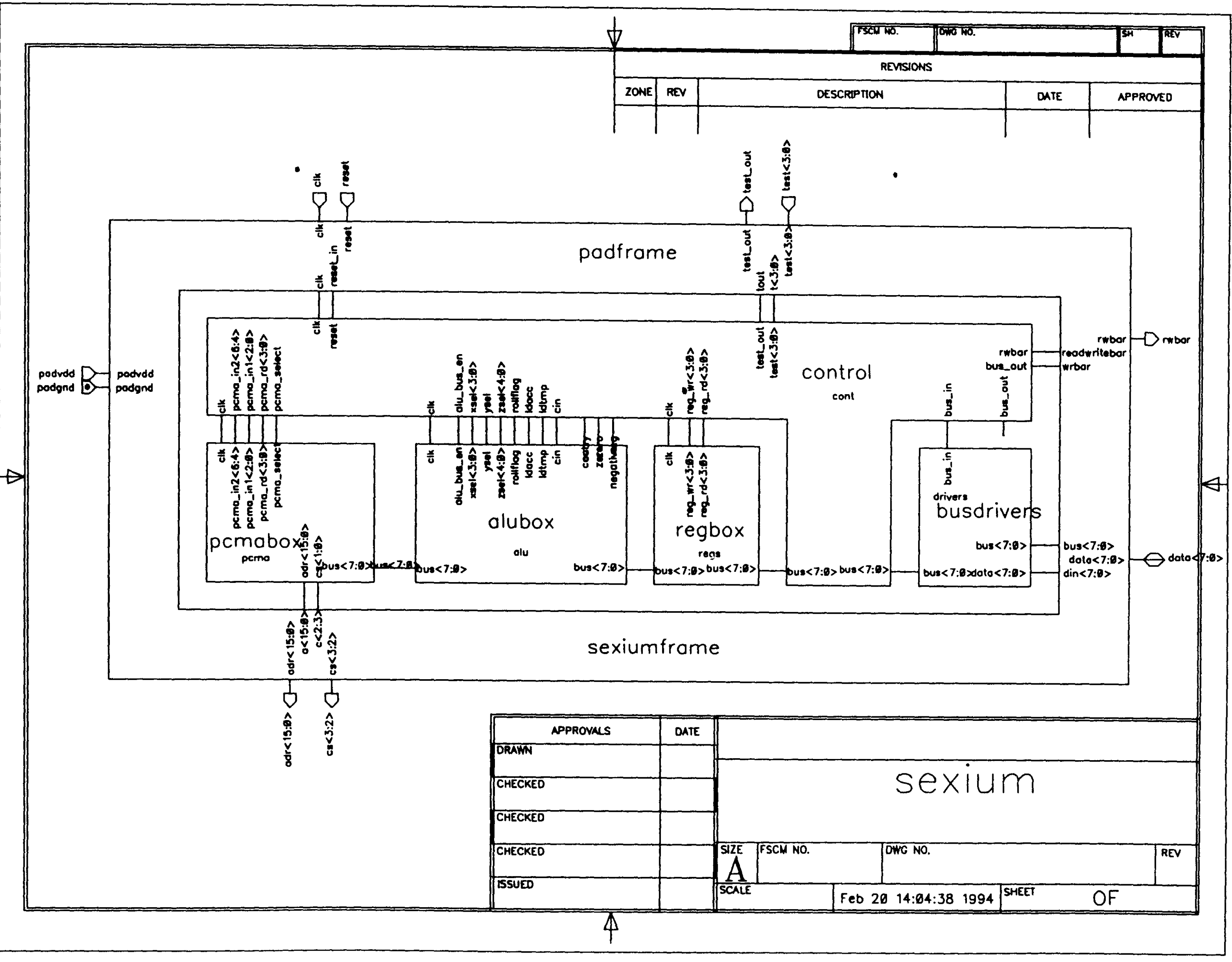
REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



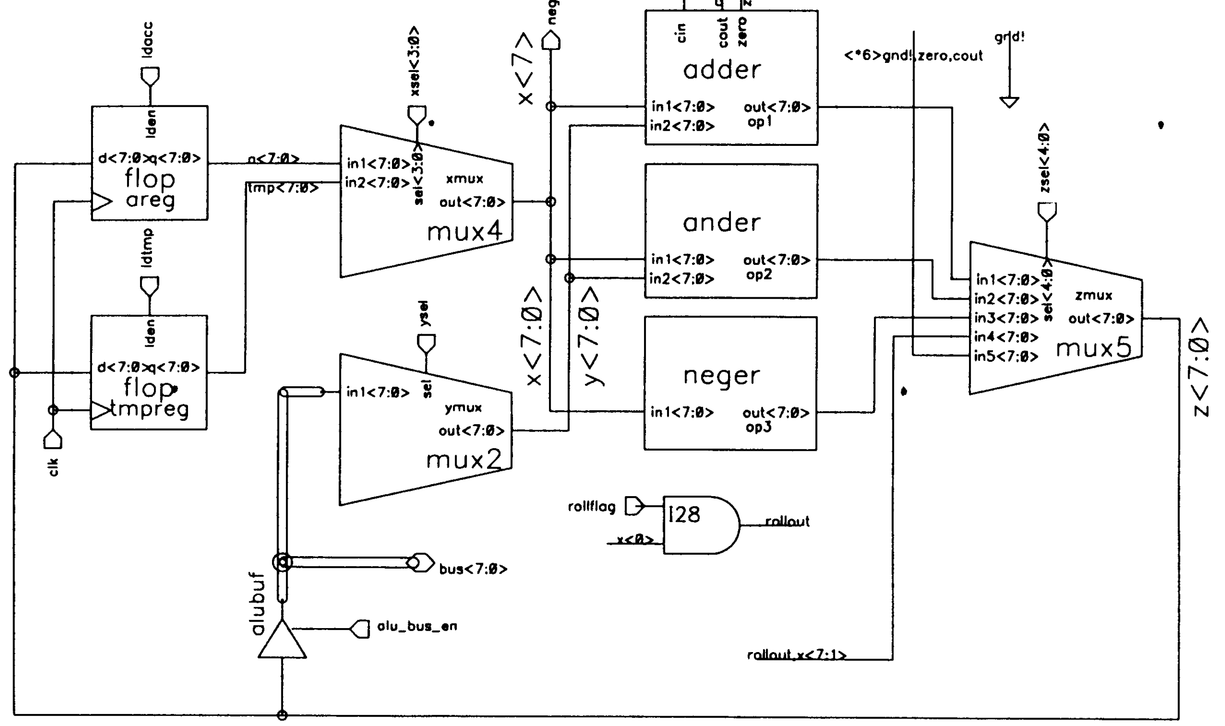
APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

computer

SIZE A	FSCM NO.	DWG NO.	REV
SCALE	Feb 20 22:39:10 1994	SHEET OF	



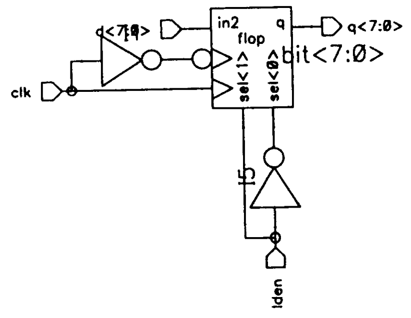
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
REV
padframe
sexiumframe
control
cont
pcmabox
pcma
alubox
alu
regbox
regs
busdrivers
drivers
clk
reset
reset_in
test_out
test<3:0>
tout
t<3:0>
rwbar
readwritebar
wrbar
bus_out
bus_in
padvdd
padgnd
pcma_in2<6:4>
pcma_in1<2:0>
pcma_rd<3:0>
pcma_select
alu_bus_en
xsel<3:0>
ysel
zsel<4:0>
rollflag
ldacc
ldtmp
cin
carry
zero
negative
reg_wr<3:0>
reg_rd<3:0>
bus<7:0>
data<7:0>
din<7:0>
adr<15:0>
a<15:0>
c<1:0>
c<2:3>
cs<3:2>
APPROVALS
DATE
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
sexium
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
Feb 20 14:04:38 1994
SHEET
OF


REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
REV
ldacc
lden
d<7:0>q<7:0>
flop
areg
ldtmp
flop
tmpreg
clk
a<7:0>
tmp<7:0>
xsel<3:0>
in1<7:0>
in2<7:0>
sel<3:0>
xmux
out<7:0>
mux4
ysel
sel
ymux
mux2
alubuf
alu_bus_en
bus<7:0>
negative
x<7>
x<7:0>
y<7:0>
cin
cout
zero
adder
op1
ander
op2
neger
op3
<*6>gnd!,zero,cout
gnd!
zsel<4:0>
in3<7:0>
in4<7:0>
in5<7:0>
sel<4:0>
zmux
mux5
z<7:0>
rollflag
x<0>
I28
rollout
rollout,x<7:1>
APPROVALS
DATE
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
alubox
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
Jan 25 17:45:34 1994
SHEET
OF

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED

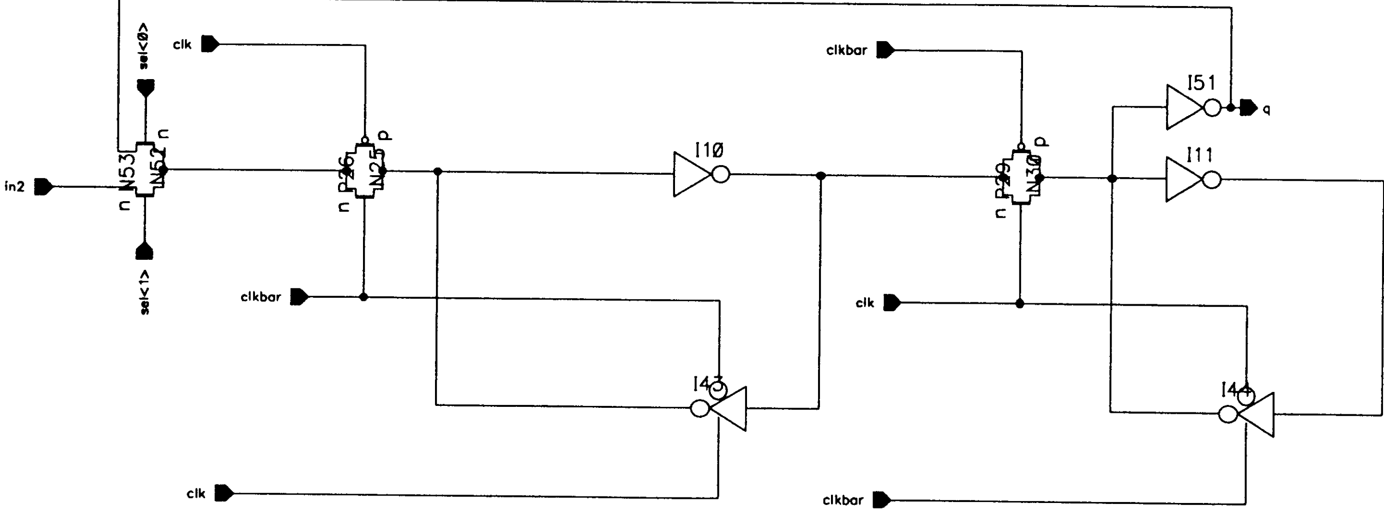


APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

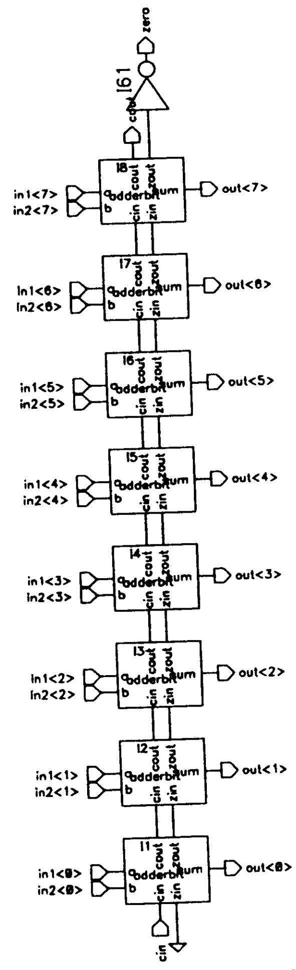
flop2

SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE	Feb 3 20:17:19 1994	SHEET	OF


in2
sel<1>
sel<0>
N53
N52
n
clk
clkbar
P26
N25
p
I10
I43
I44
I11
I51
P29
N30
q
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
APPROVALS
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
flop2bit
SIZE
A
SCALE
Mar 7 16:42:51 1994
SHEET
OF

202



FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

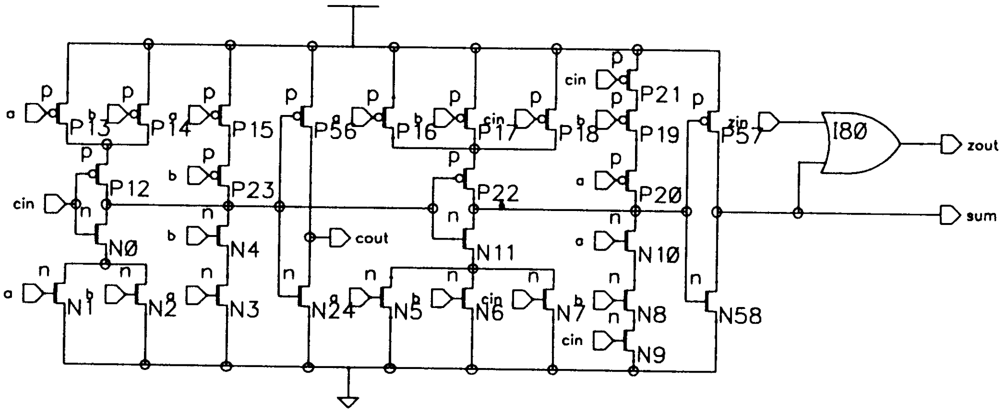
adder

SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE	Feb 20 23:50:30 1994	SHEET OF	

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

adderbit

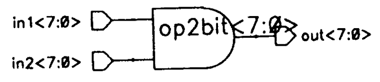
SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE	Feb 20 23:55:20 1994	SHEET	OF

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

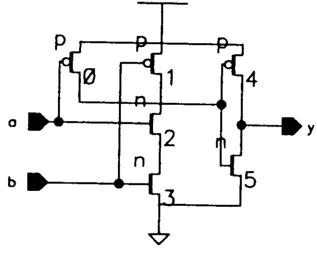
under

SIZE A	FSCM NO.	DWG NO.	REV
SCALE	Jan 21 15:16:12 1994	SHEET OF	

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



and2

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

anderbit

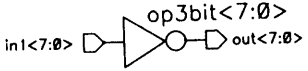
SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE	Feb 21 00:08:01 1994	SHEET	OF

206

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

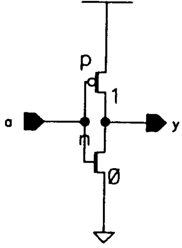
neger

SIZE A	FSCM NO.	DWG NO.	REV
SCALE	Jan 22 16:13:35 1994	SHEET OF	

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



inv

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

negerbit

SIZE	FSCM NO.	DWG NO.	REV
A			

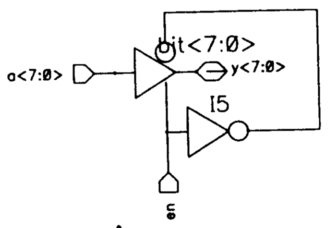
SCALE	Jan 19 15:19:52 1994	SHEET	OF

208

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

~~busd~~ tribuf

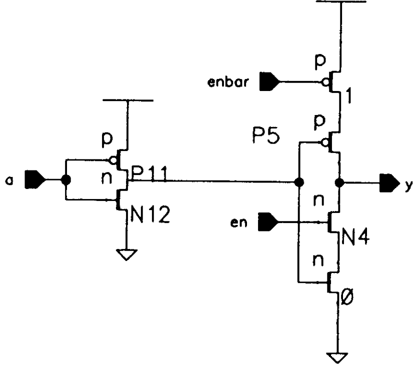
SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE Jan 22 14:42:54 1994 SHEET OF

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



inv

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

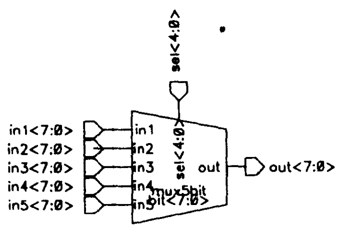
invtri

SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE	Jan 24 09:01:22 1994	SHEET	OF

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

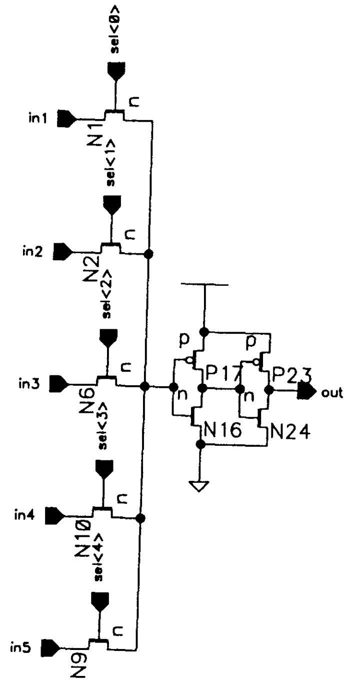
mux5

SIZE A	FSCM NO.	DWG NO.	REV
SCALE	Jan 19 13:34:44 1994	SHEET OF	

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

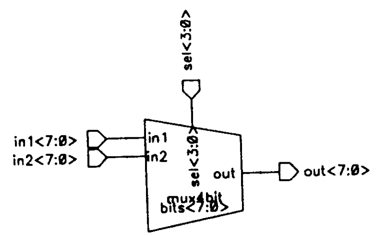
mux5bit

SIZE A	FSCM NO.	DWG NO.	REV
SCALE	Jan 22 16:13:36 1994	SHEET	OF

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



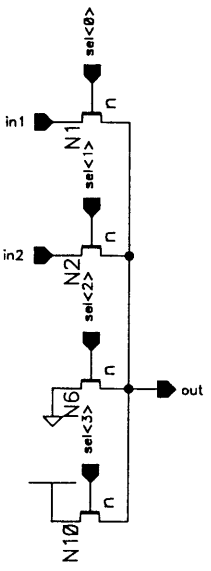
APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

mux4

SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE	Jan 22 14:18:47 1994	SHEET OF	

FSCM NO.	DWG NO.	SH	REV

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

mux4bit

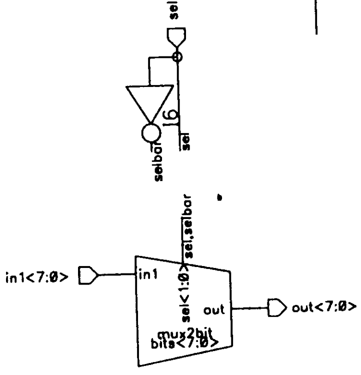
SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE	Jan 23 14:20:50 1994	SHEET	OF

214

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

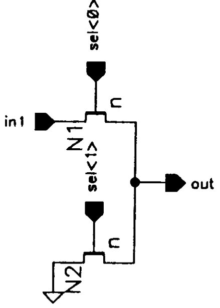
mux2gnd

SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE	Jan 22 14:13:06 1994	SHEET OF	

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED

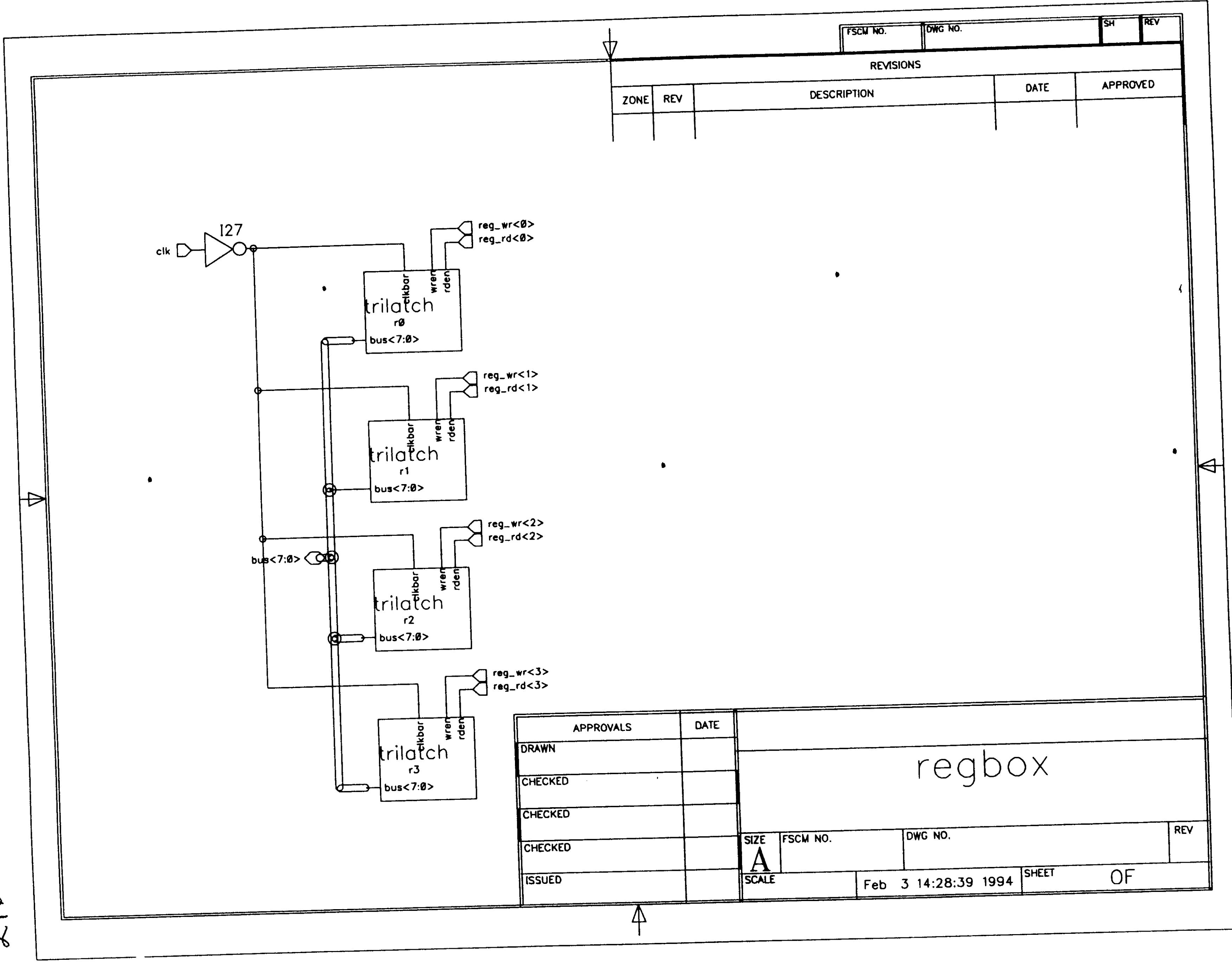


APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

mux2bit
gnd

SIZE	FSCM NO.	DWG NO.	REV
A			

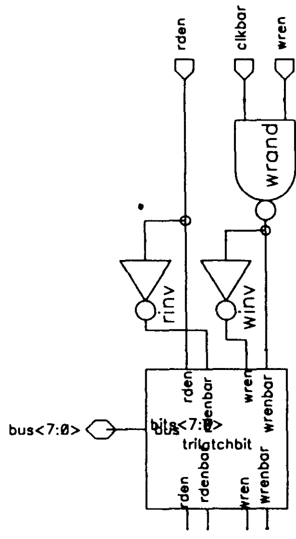
SCALE | Jan 22 14:11:24 1994 | SHEET OF


FSCM NO.
DWG NO.
SH
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
clk
I27
reg_wr<0>
reg_rd<0>
clkbar
wren
rden
trilatch
r0
bus<7:0>
reg_wr<1>
reg_rd<1>
trilatch
r1
bus<7:0>
reg_wr<2>
reg_rd<2>
bus<7:0>
trilatch
r2
bus<7:0>
reg_wr<3>
reg_rd<3>
trilatch
r3
bus<7:0>
APPROVALS
DATE
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
regbox
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
Feb 3 14:28:39 1994
SHEET
OF

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED

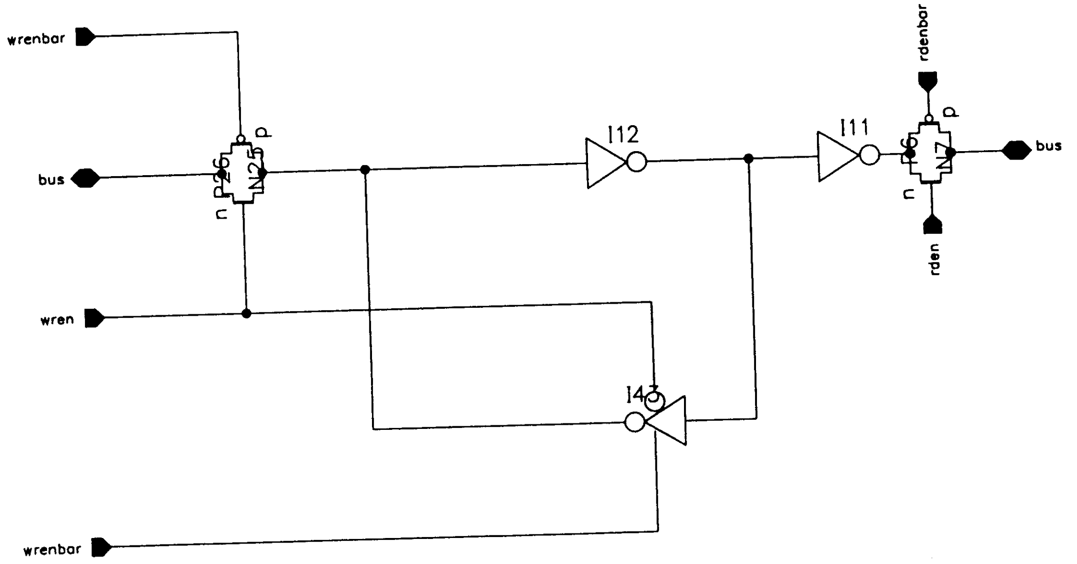


APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

trilatch

SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE | Feb 3 16:21:25 1994 | SHEET OF



REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

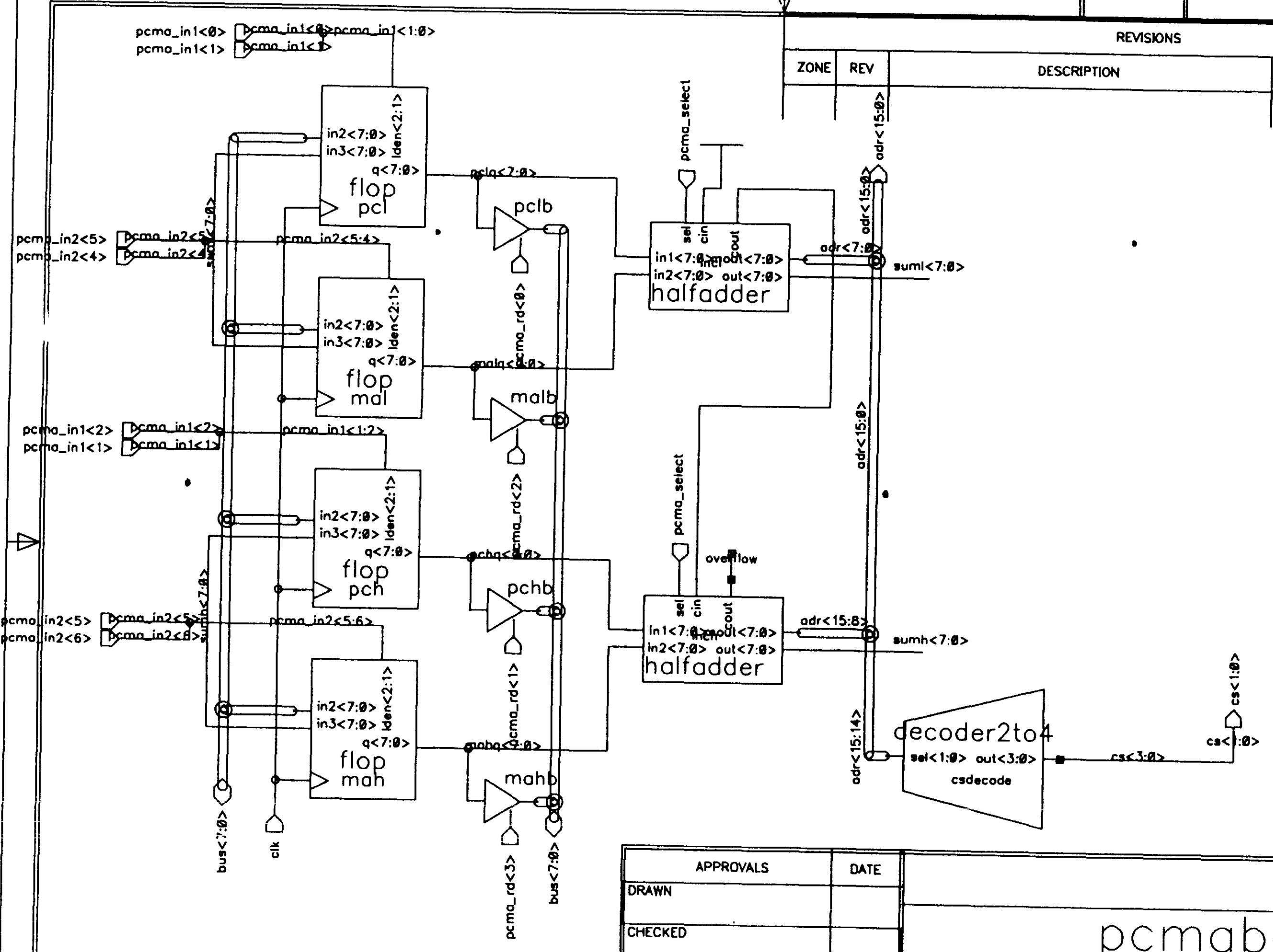
FSCM NO.	DWG NO.	SH	REV

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

trilatchbit

SIZE	FSCM NO.	DWG NO.	REV
A			
SCALE	Feb 20 22:38:55 1991	SHEET	OF

218

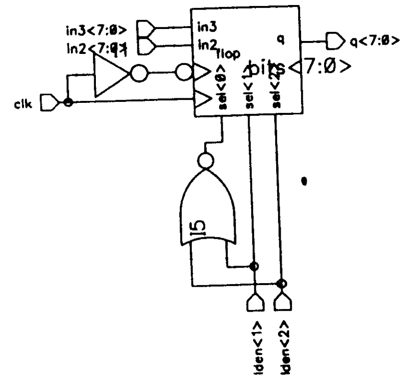

pcma_in1<0>
pcma_in1<1>
pcma_in1<1:0>
pcma_in2<5>
pcma_in2<4>
pcma_in2<5:4>
suml<7:0>
pcma_in1<2>
pcma_in1<1>
pcma_in1<1:2>
pcma_in2<5>
pcma_in2<6>
pcma_in2<5:6>
sumh<7:0>
in2<7:0>
in3<7:0>
lden<2:1>
q<7:0>
flop
pcl
mal
pch
mah
pclq<7:0>
malq<7:0>
pchq<7:0>
mahq<7:0>
pclb
malb
pchb
mahb
pcma_rd<0>
pcma_rd<2>
pcma_rd<1>
pcma_rd<3>
bus<7:0>
clk
pcma_select
sel
cin
cout
in1<7:0>
in2<7:0>
mout<7:0>
out<7:0>
halfadder
overflow
adr<15:0>
adr<7:0>
adr<15:8>
adr<15:14>
suml<7:0>
sumh<7:0>
decoder2to4
sel<1:0>
out<3:0>
csdecode
cs<3:0>
cs<1:0>
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
APPROVALS
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
pcmabox
SIZE
A
SCALE
Feb 20 13:18:37 1994
SHEET
OF

220

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED

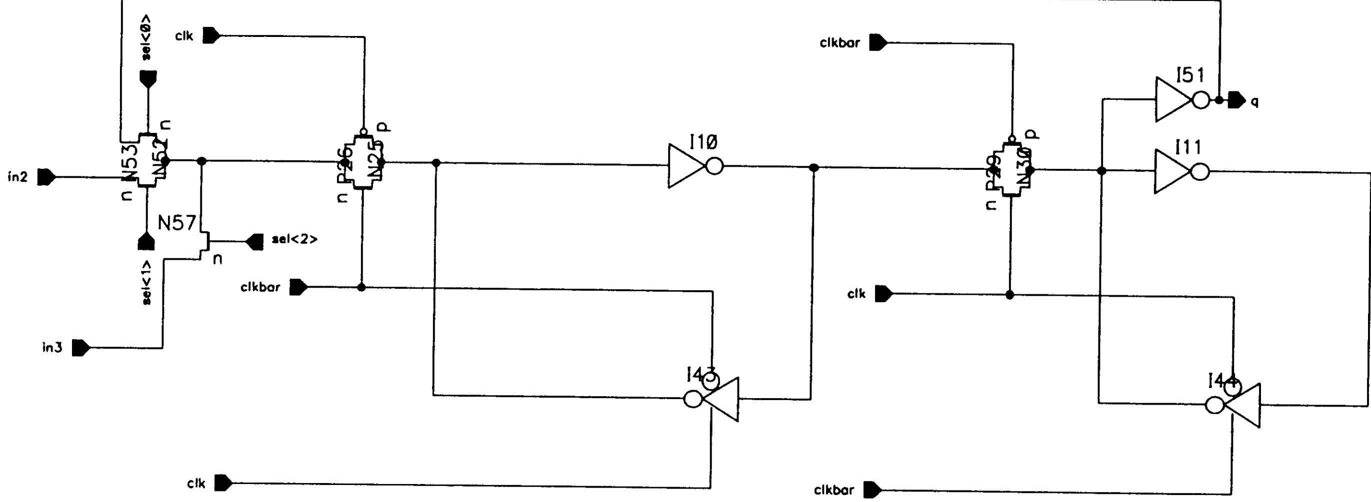


APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

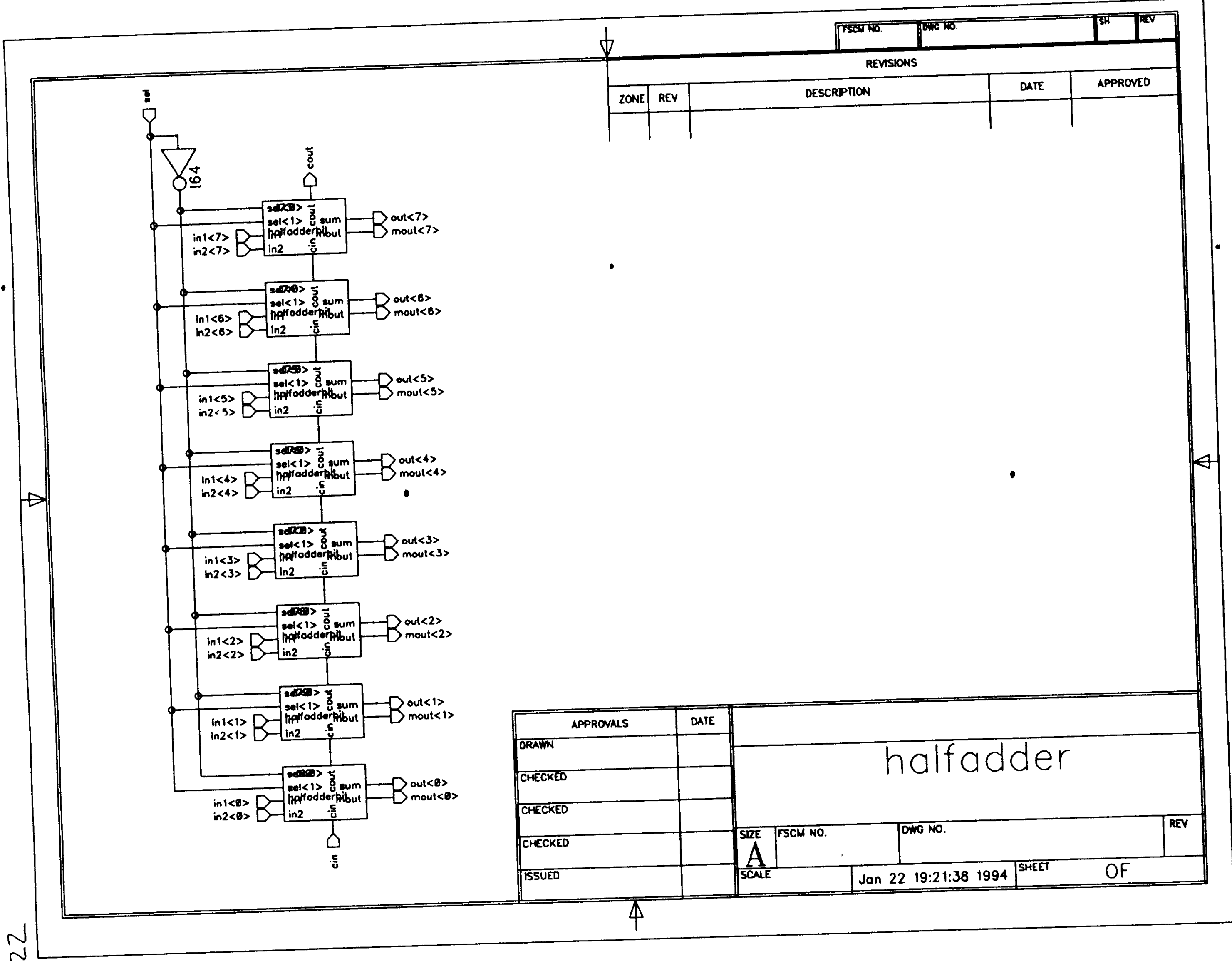
flop3

SIZE	FSCM NO.	DWG NO.	REV
A			

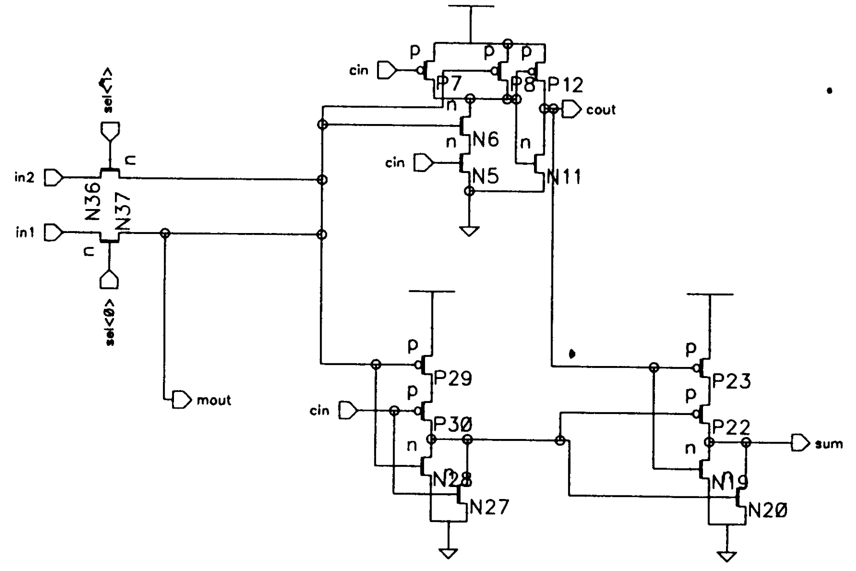
SCALE | Feb 3 20:17:20 1994 | SHEET OF



in2
in3
sel<0>
sel<1>
sel<2>
N53
N52
N57
clk
clkbar
P26
N25
I10
I43
P29
N30
I11
I51
I44
q
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
FSCM NO.
DWG NO.
SH
APPROVALS
DRAWN
CHECKED
ISSUED
flop3bit
SIZE
A
SCALE
Mar 7 17:43:01 1994
SHEET
OF



sel
I64
cout
cin
halfadderbit
sel<1>
in1
in2
sum
mout
in1<0>
in2<0>
out<0>
mout<0>
in1<1>
in2<1>
out<1>
mout<1>
in1<2>
in2<2>
out<2>
mout<2>
in1<3>
in2<3>
out<3>
mout<3>
in1<4>
in2<4>
out<4>
mout<4>
in1<5>
in2<5>
out<5>
mout<5>
in1<6>
in2<6>
out<6>
mout<6>
in1<7>
in2<7>
out<7>
mout<7>
FSCM NO.
DWG NO.
SH
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
APPROVALS
DATE
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
halfadder
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
Jan 22 19:21:38 1994
SHEET
OF



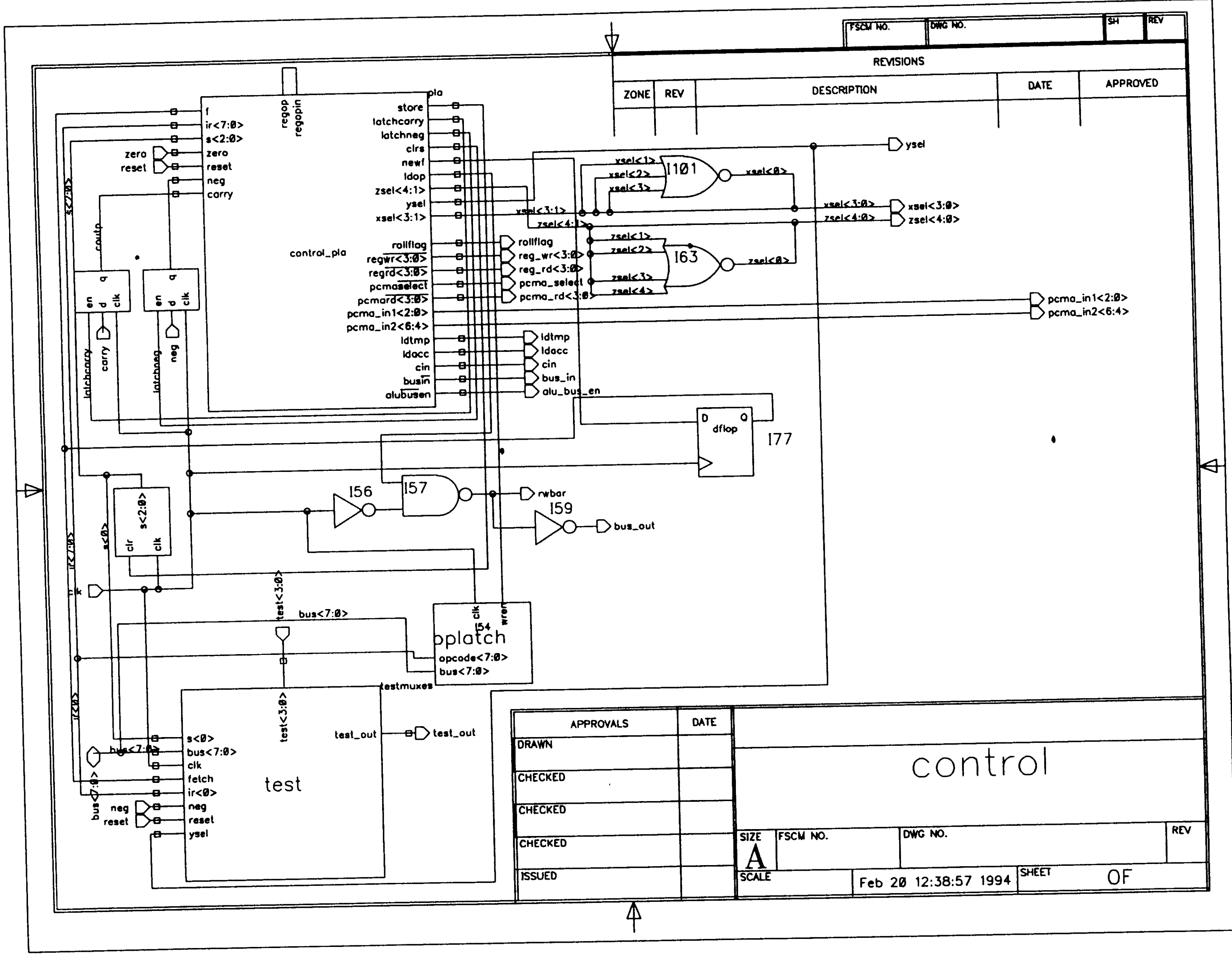
ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

halfaddebit

SIZE	FSCM NO.	DWG NO.	REV
A			

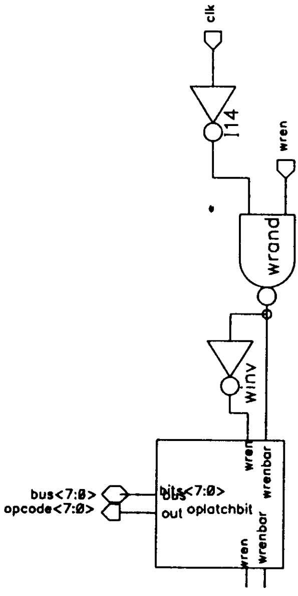
SCALE | Jan 24 08:29:49 1994 | SHEET OF


control
Feb 20 12:38:57 1994
APPROVALS
DATE
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
SIZE
A
FSCM NO.
DWG NO.
SCALE
SHEET
OF
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
control_pla
test
oplatch
dflop
test<3:0>
test_out
bus<7:0>
opcode<7:0>
s<2:0>
latchcarry
latchneg
carry
neg
zero
reset
rwbar
bus_out
ysel
xsel<3:0>
zsel<4:0>
rollflag
reg_wr<3:0>
reg_rd<3:0>
pcma_select
pcma_rd<3:0>
pcma_in1<2:0>
pcma_in2<6:4>
ldtmp
ldacc
cin
bus_in
alu_bus_en
I56
I57
I59
I63
I101
I77

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED



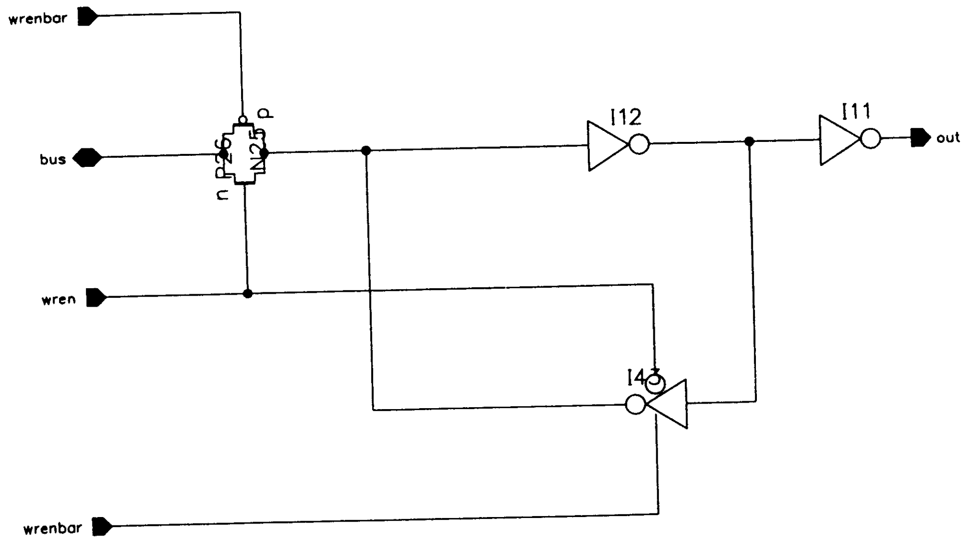
APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

oplatch

SIZE	FSCM NO.	DWG NO.	REV
A			

SCALE | Feb 5 21:53:56 1994 | SHEET OF

226



FSCM NO.	DWG NO.	SH	REV

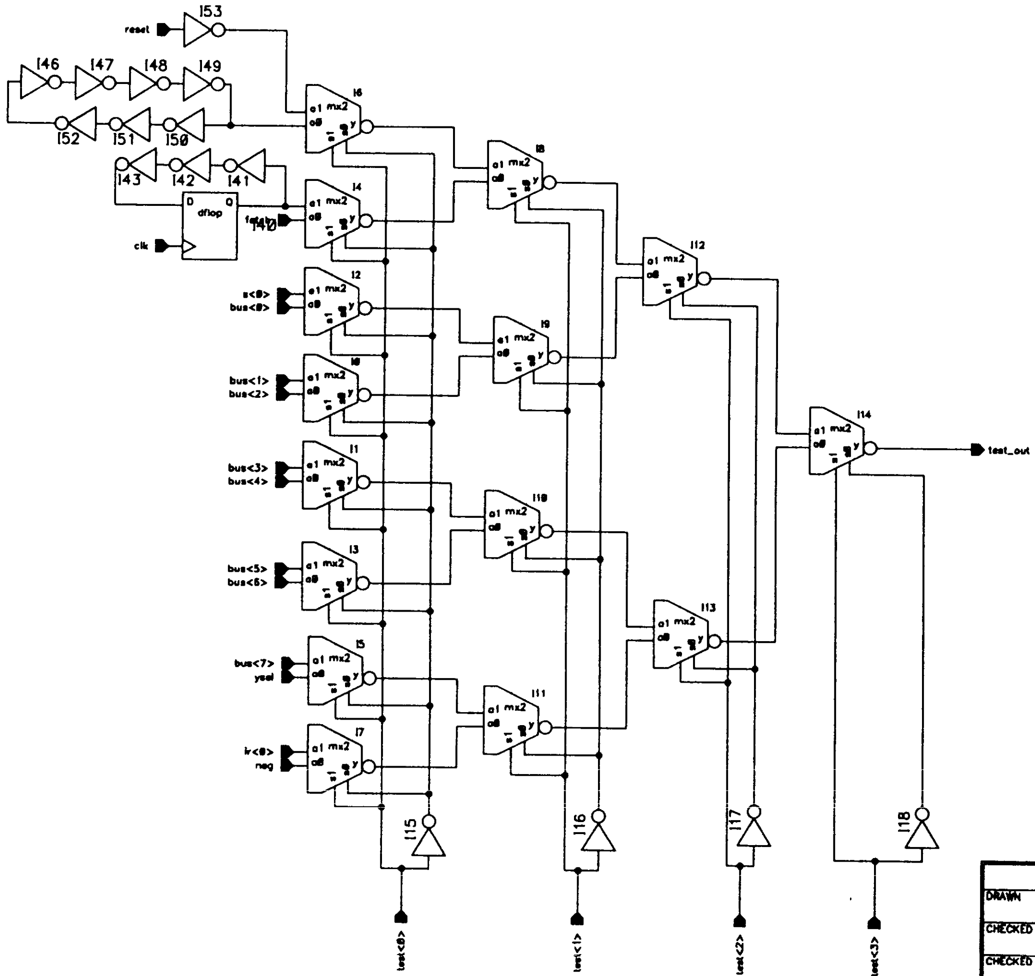
REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

oplatchbit

SIZE A	FSCM NO.	DWG NO.	REV
SCALE	Jan 24 15:33:46 1994	SHEET OF	



REVISIONS

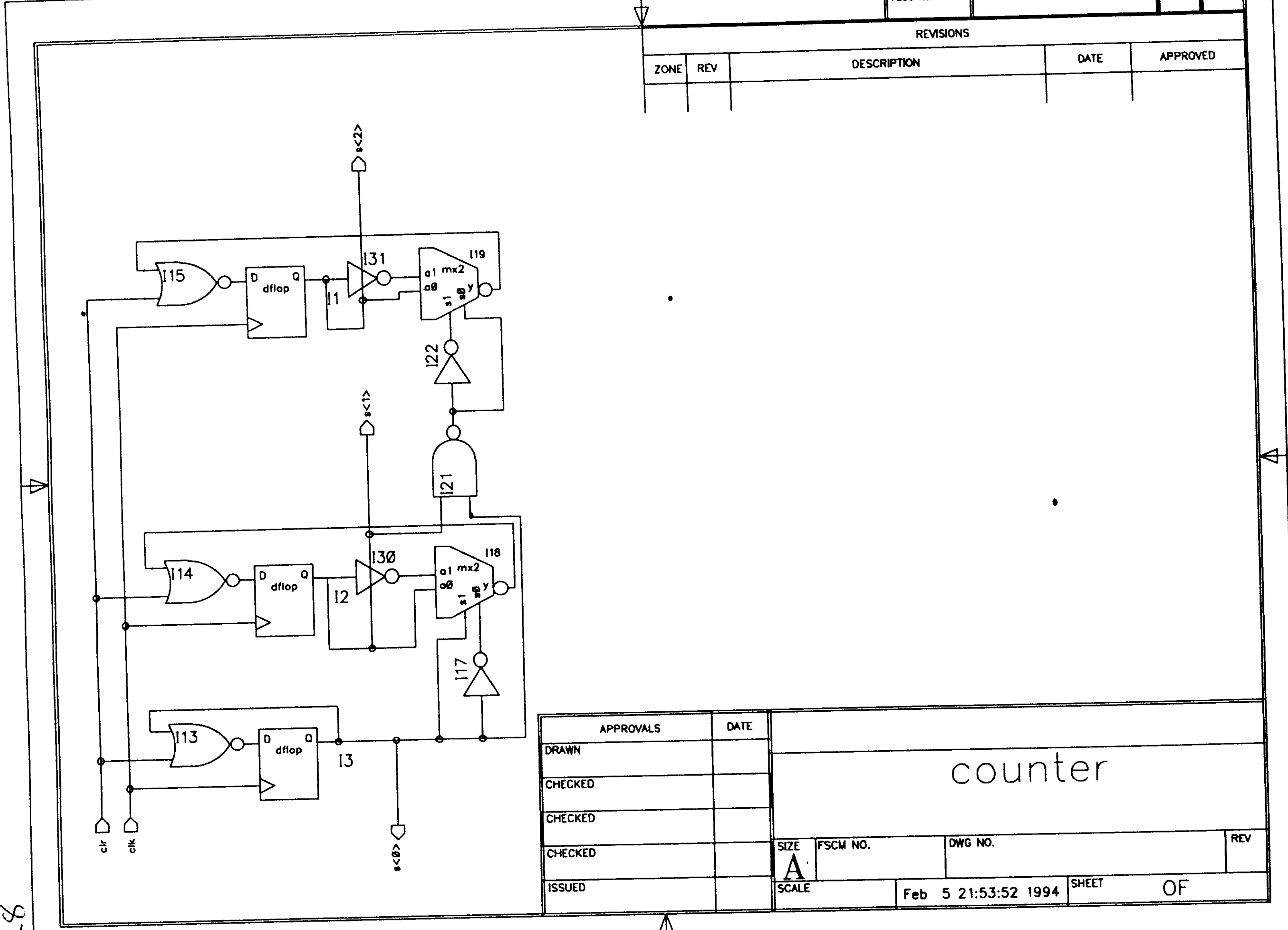
ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

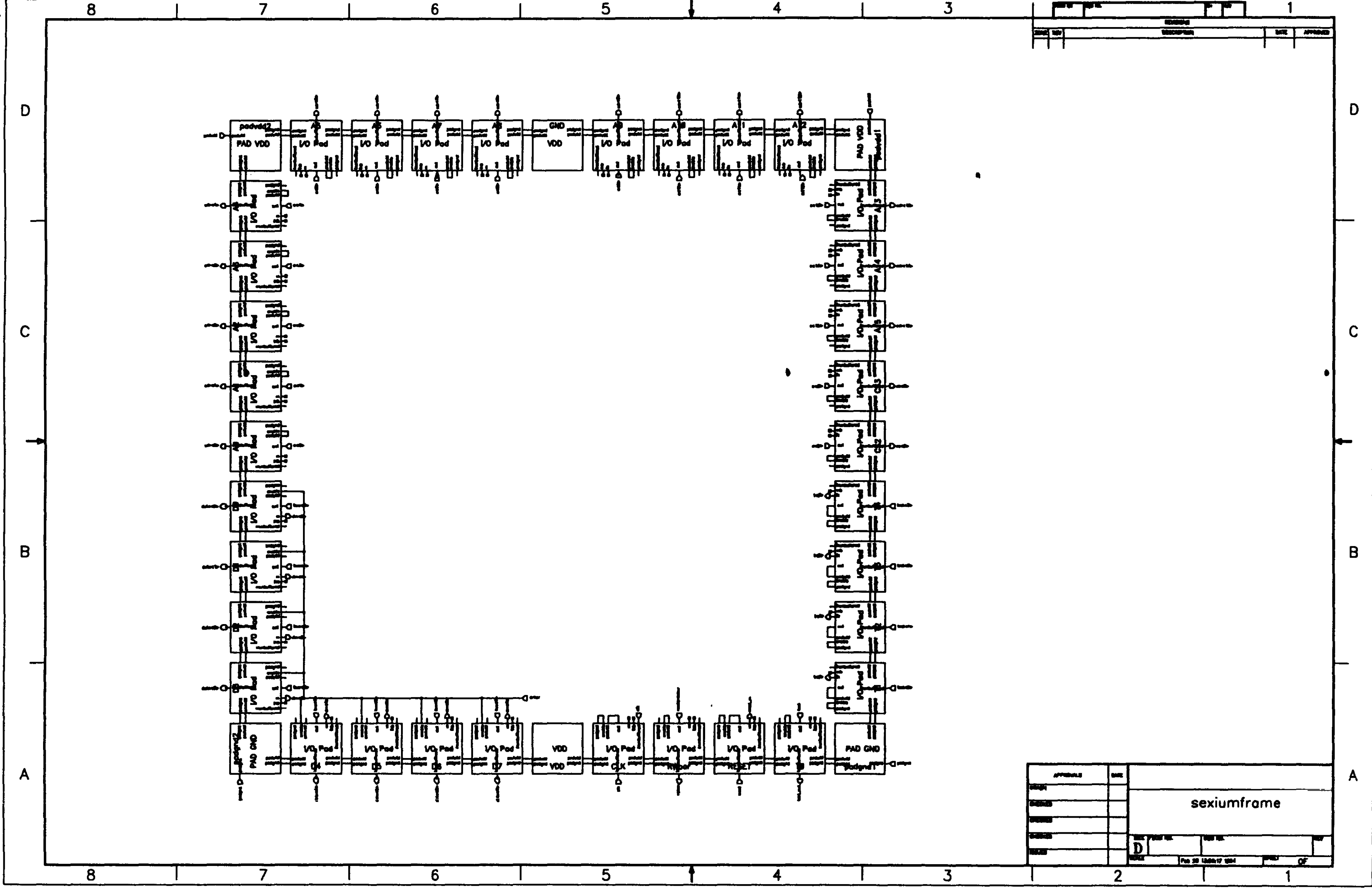
test

SIZE B FSCM NO. DWG NO. REV

SCALE Feb 21 00:04:57 1994 SHEET OF



FSCM NO.
DWG NO.
SH
REV
REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
s<2>
I15
D
dflop
Q
I31
I1
a1
mx2
a0
s1
s0
y
I19
I22
s<1>
I21
I14
D
dflop
Q
I30
I2
a1
mx2
a0
s1
s0
y
I18
I17
I13
D
dflop
Q
I3
clr
clk
s<0>
APPROVALS
DATE
DRAWN
CHECKED
CHECKED
CHECKED
ISSUED
counter
SIZE
A
FSCM NO.
DWG NO.
REV
SCALE
Feb 5 21:53:52 1994
SHEET
OF



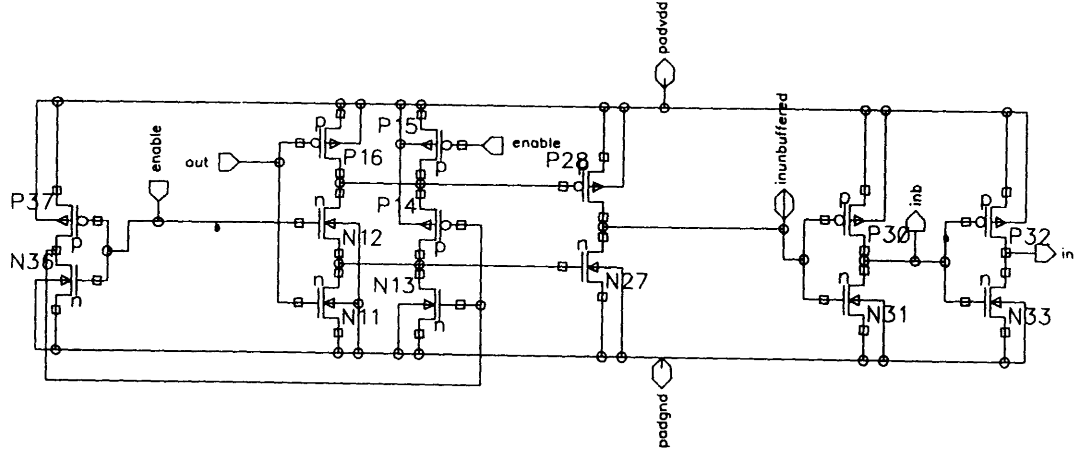
8
7
6
5
4
3
2
1
D
C
B
A
PAD VDD
PAD GND
I/O Pad
VDD
GND
CLK
RESET
sexiumframe
D

230

FSCM NO.	DWG NO.	SH	REV

REVISIONS

ZONE	REV	DESCRIPTION	DATE	APPROVED

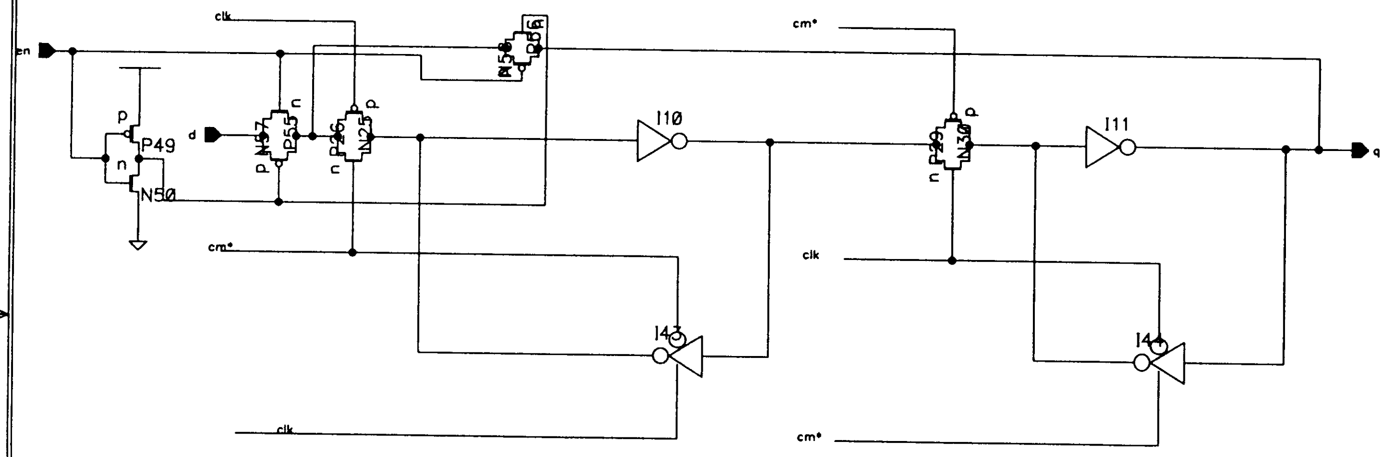


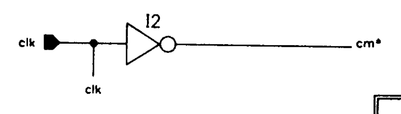
APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

v4io

SIZE A	FSCM NO.	DWG NO.	REV
SCALE	Feb 12 18:48:40 1994	SHEET OF	

231





REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE
DRAWN	
CHECKED	
CHECKED	
CHECKED	
ISSUED	

flopen

SIZE A | FSCM NO. | DWG NO. | REV

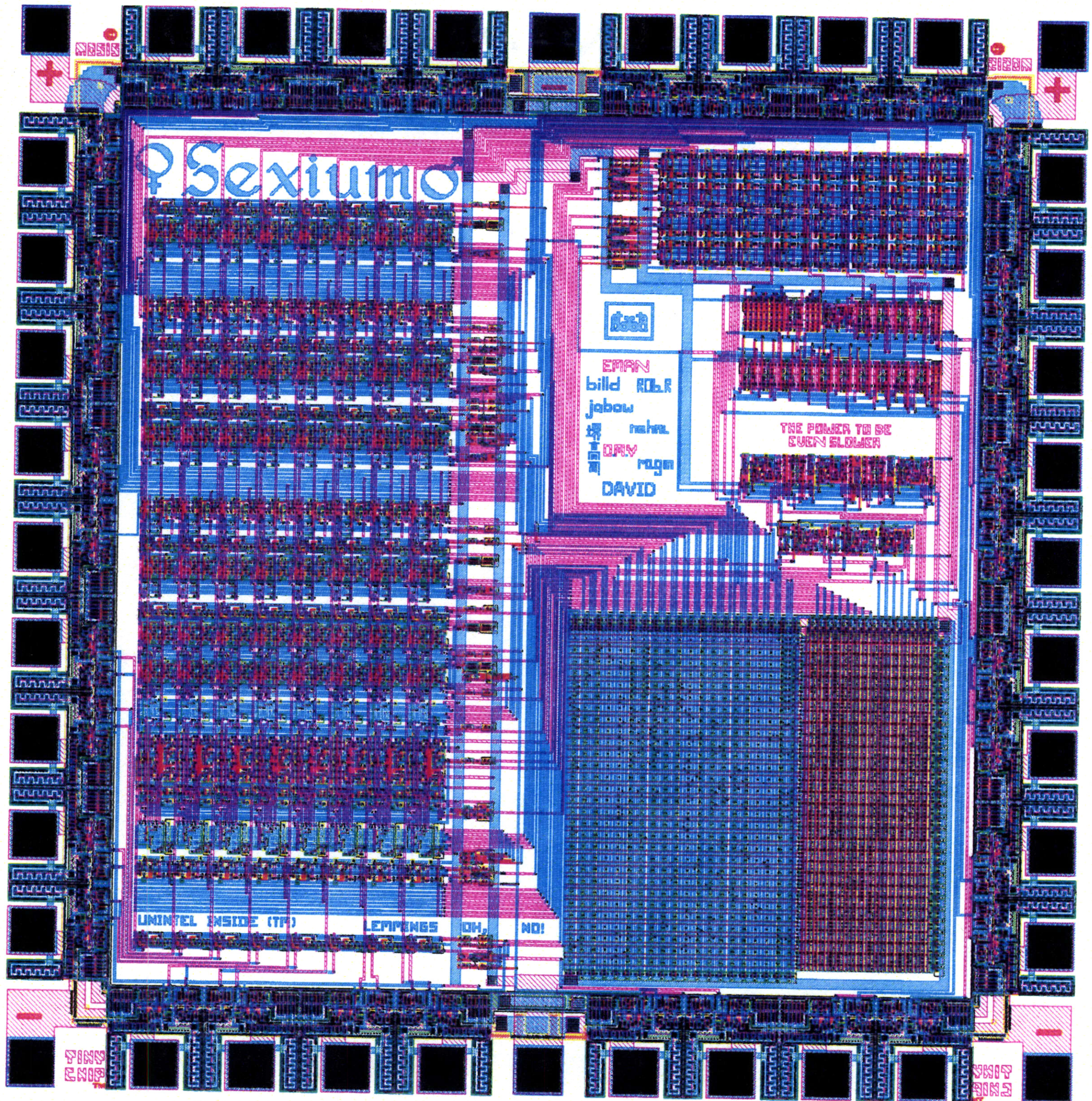
SCALE | Mar 7 17:59:26 1994 | SHEET OF

Sexium Layout

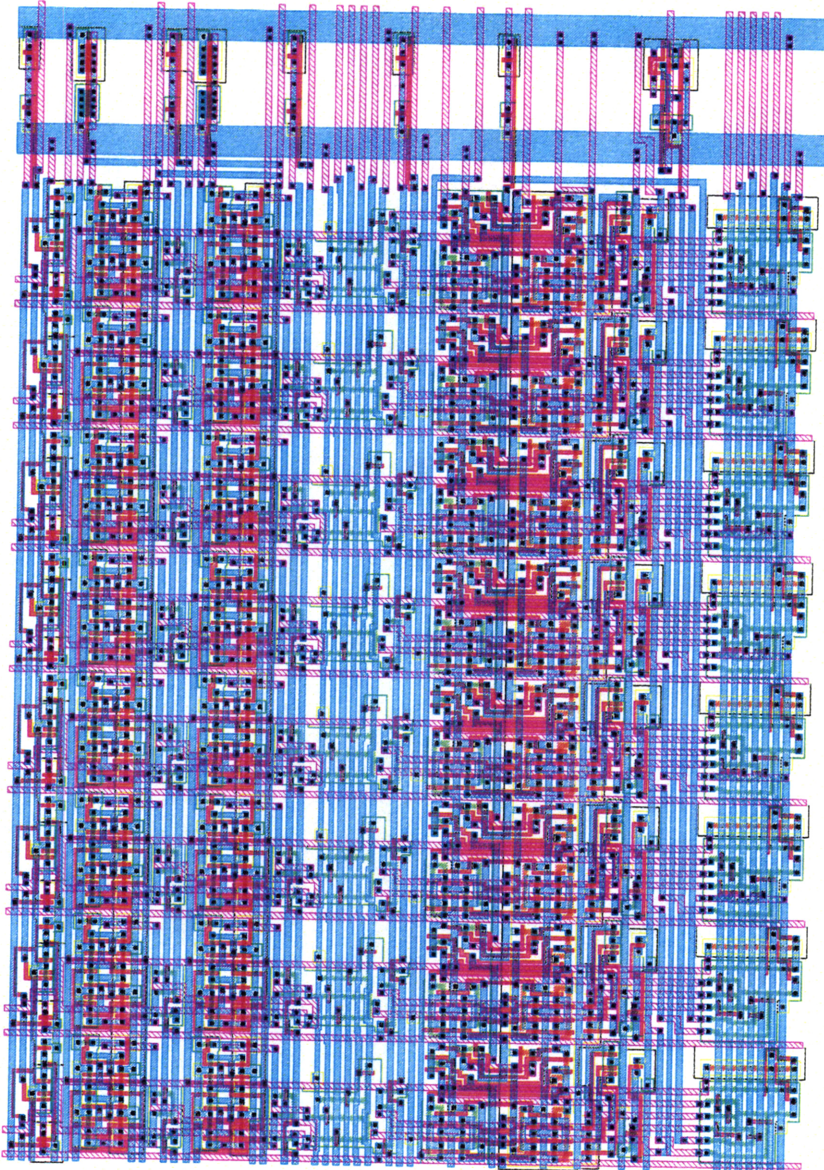
This section contains color layout plots of selected cells from the Sexium, including the major modules and other particularly interesting cells.

sexium 234
alubox 235
flop2bit 236
adderbit 237
regbox 238
pcmabox 239
control_pla 240
v4io 241

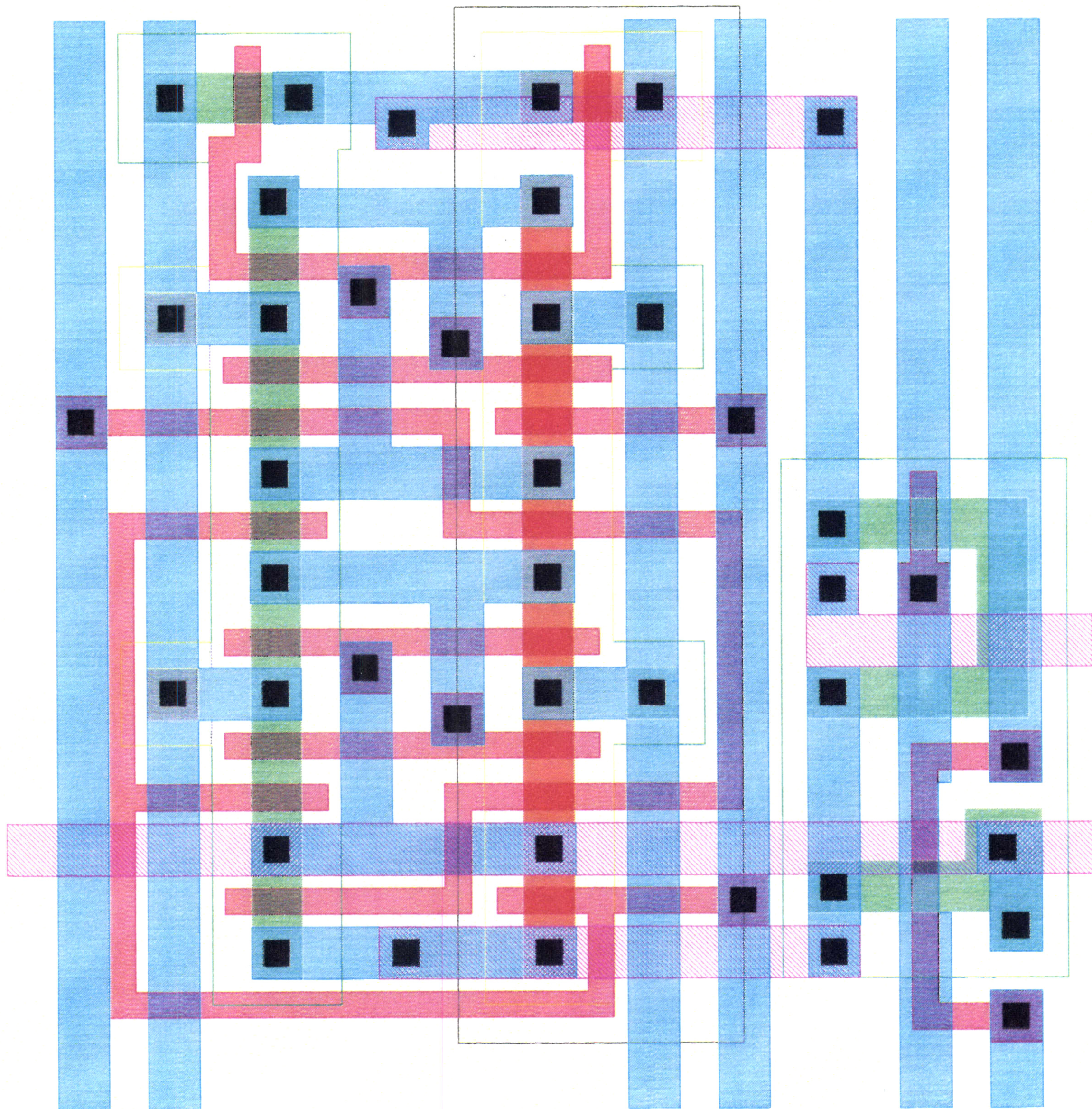
sexium



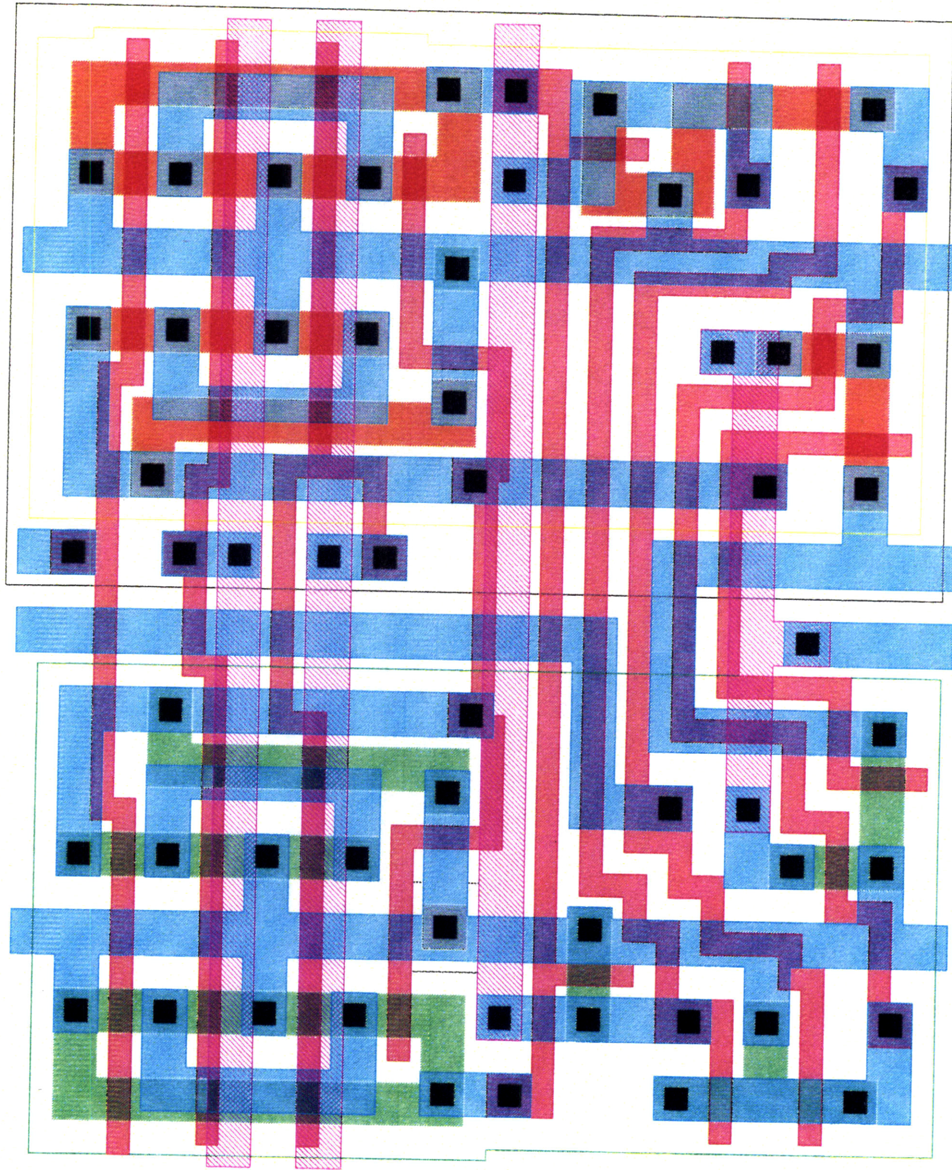
alubox



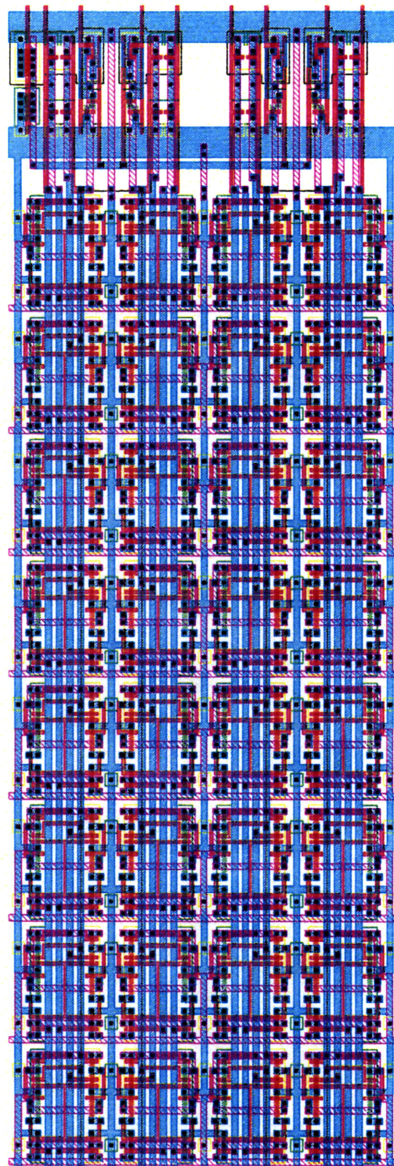
flop2bit



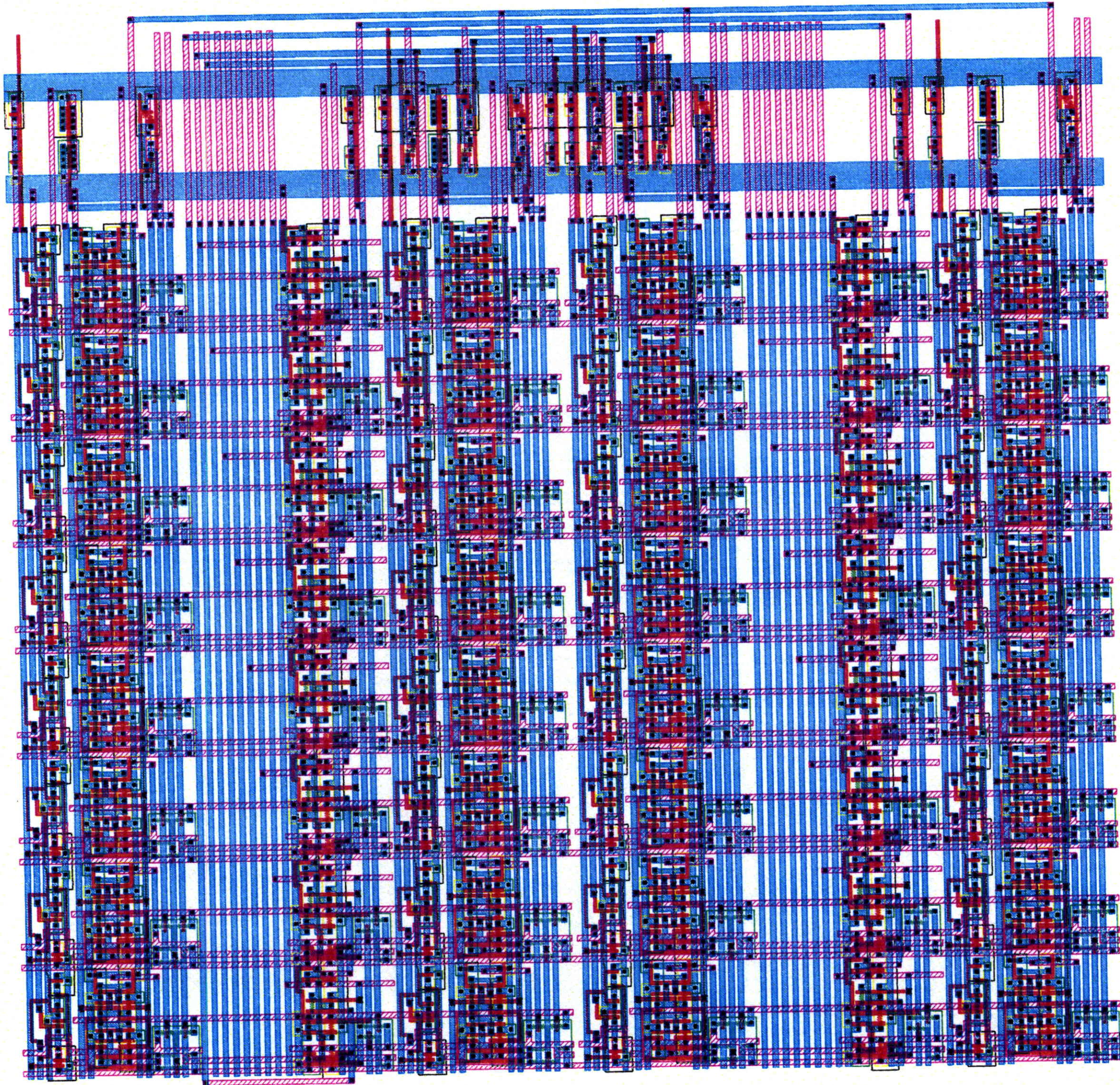
adderbit



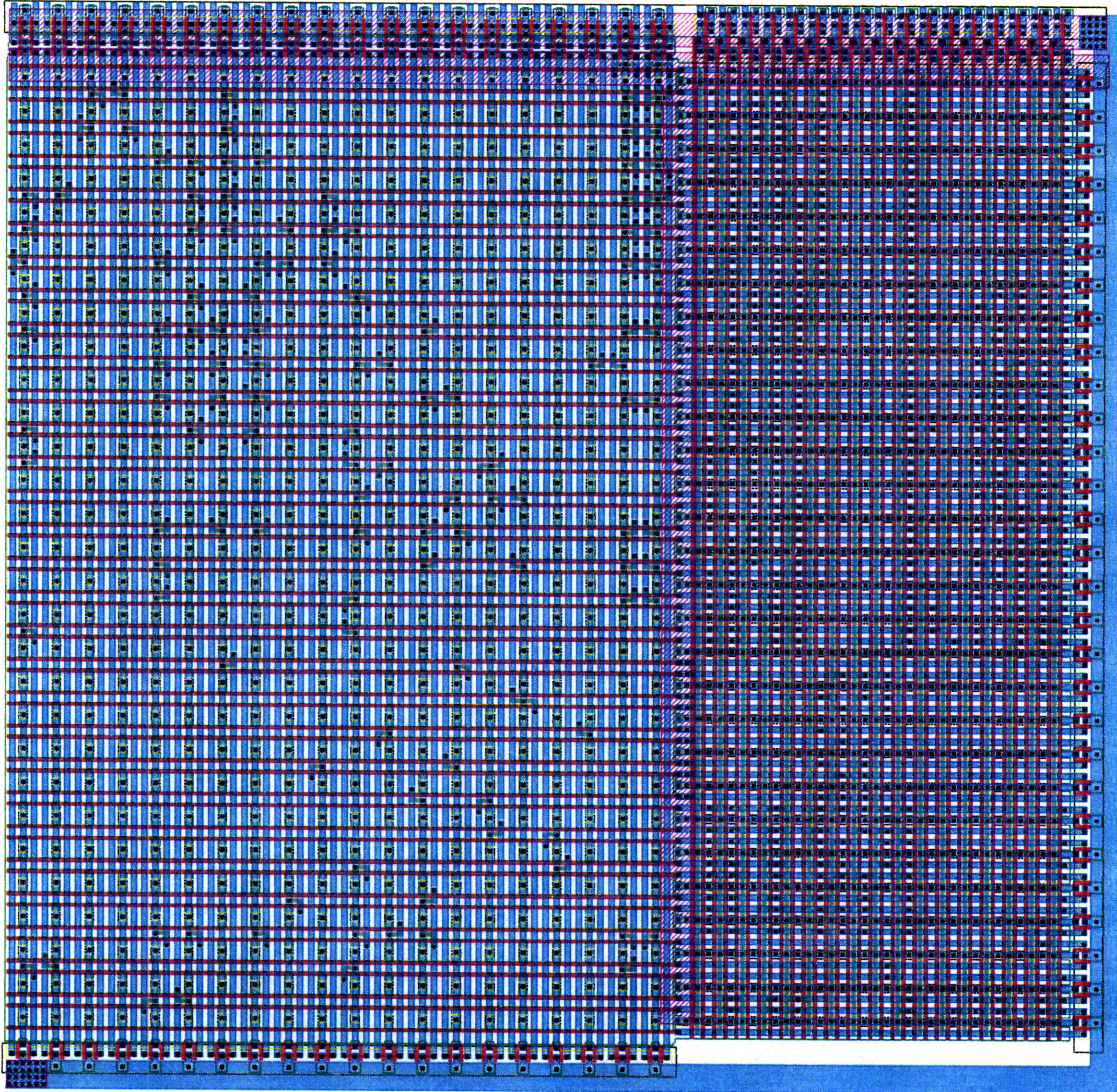
regbox



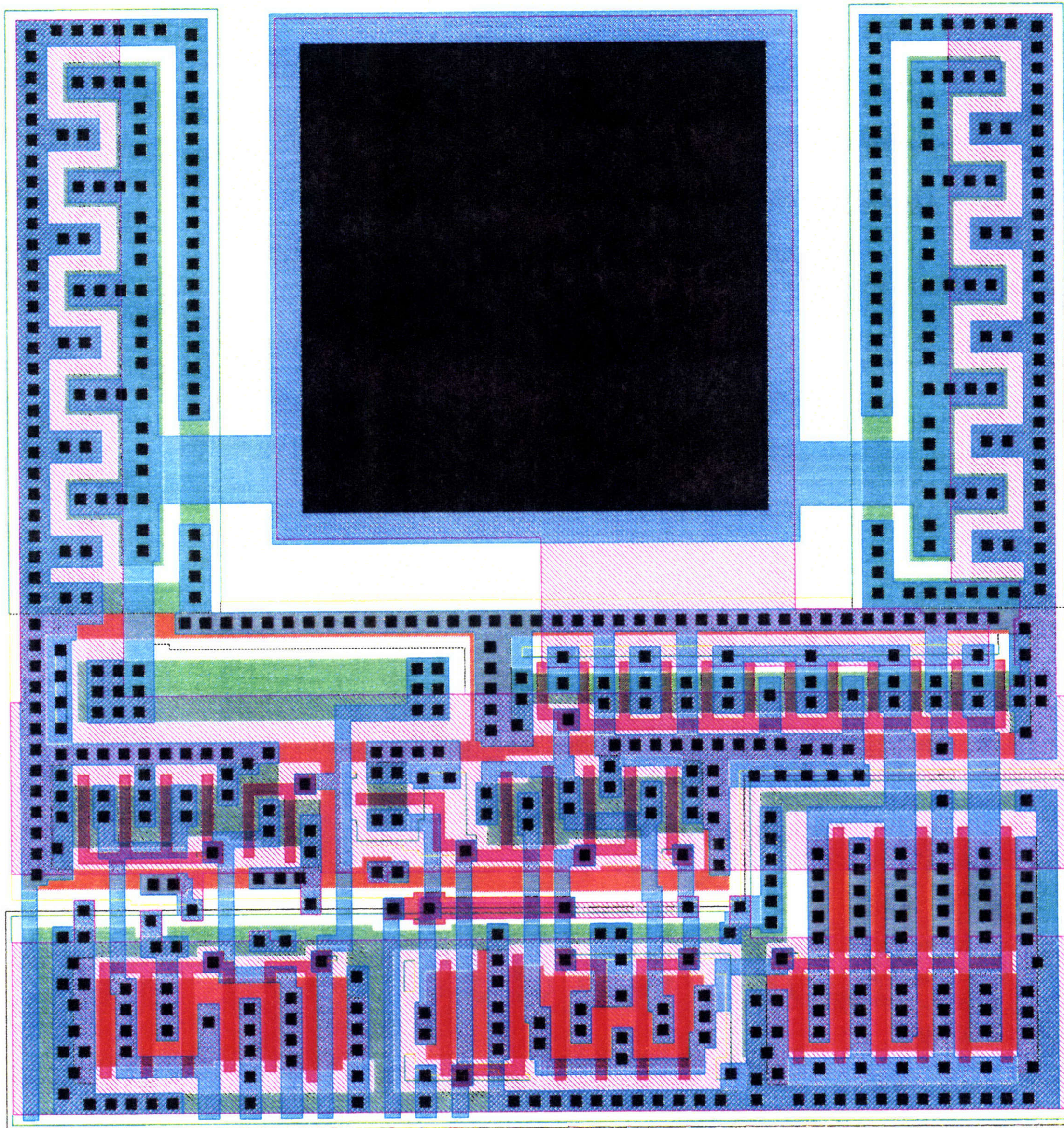
pcmabox



control_pla



v4io



Verilog Model

```
// Sexium.v

// This file contains a Verilog model of the
// Unintel Sexium 8 bit microprocessor.

// Version History
//
// 2/20/94:  Model updated to reflect new control and bus widths
// 1/16/94:  Model passes regression test
// 1/7/94:   Original version developed by David Harris

// Notes:
//  Reset is asynchronus.  This can be fixed if necessary.

/////////////////////////////////////////////
// computer
/////////////////////////////////////////////
module computer();

  // Internal control
  reg           clk;
  reg           reset;

  // Memory interface
  wire [15:0]   adr;
  wire [7:0]    data;
  wire [3:2]    cs;
  wire          rwbar;
  wire [3:0]    test;
  wire          test_out;

  // Instances of Sexium & Memory

  sexium cpu(clk, reset, adr, data, cs, rwbar, test, test_out);
  memory mem(adr, data, rwbar);

  // Simulation routines
  // Generate clock
  initial
    begin
      clk = 1;
      reset = 1;
      reset <= #25 0;
      forever
      begin
        clk = 1;
        #10;
        clk = 0;
        #10;
      end
    end

  // Show waveforms
  initial
    begin
```

```
    $gr_waves("clk",clk,
          "reset",reset,
          "adr",adr,
          "data",data,
          "cs %b",cs,
          "rwbar",rwbar,
          "x",cpu.alu.x,
          "y",cpu.alu.y,
          "z",cpu.alu.z,
          "xsel",cpu.alu.xsel,
          "ysel",cpu.alu.ysel,
          "zsel",cpu.alu.zsel,
          "pcma_in %b",cpu.pcma.pcma_in,
          "mald",cpu.pcma.mald,
          "pcld",cpu.pcma.pcld,
          "pcma_select",cpu.pcma.pcma_select,
          "lda",cpu.cont.pla.lda,
          "bus",cpu.bus);
     $gr_regs(
          "IR  %h   S    %h    F   %h   OP  %s",
          cpu.cont.ir,cpu.cont.s,cpu.cont.f,
          opname(reset,cpu.cont.f,cpu.cont.pla.add,
               cpu.cont.pla.andw,cpu.cont.pla.notw,
               cpu.cont.pla.shr,cpu.cont.pla.ror,
               cpu.cont.pla.put,cpu.cont.pla.get,
               cpu.cont.pla.tst,cpu.cont.pla.lda,
               cpu.cont.pla.ldi,cpu.cont.pla.ldm,
               cpu.cont.pla.sta,cpu.cont.pla.sti,
               cpu.cont.ir[4:0] == 5'b10000,
               cpu.cont.ir[4:0] == 5'b10010,
               cpu.cont.ir[4:0] == 5'b10011,
               cpu.cont.ir[4:0] == 5'b10001,
               cpu.cont.ir[4:0] == 5'b10100,
               cpu.cont.ir[4:0] == 5'b11110),
          "A   %h   TMP  %h", cpu.alu.a,cpu.alu.tmp,
          "R0  %h   R1   %h   R2   %h  R3   %h",
          cpu.regs.r0.state,cpu.regs.r1.state,
          cpu.regs.r2.state,cpu.regs.r3.state,
          "PCH %h   PCL  %h   MAH  %h  MAL  %h",
          cpu.pcma.pchq,cpu.pcma.pclq,
          cpu.pcma.mahq,cpu.pcma.malq);
  end

function [63:0] opname;
  input reset,f,add,andw,notw,shr,ror,put,get,tst,
    lda,ldi,ldm,sta,sti,jmp,bra,skz,cal,ret,brk;
    begin
    if (reset) opname = "RESET";
    else if (f) opname = "FETCH";
    else if (add) opname = "ADD";
    else if (andw) opname = "AND";
    else if (notw) opname = "NOT";
    else if (shr) opname = "SHR";
    else if (ror) opname = "ROR";
    else if (put) opname = "PUT";
    else if (get) opname = "GET";
    else if (tst) opname = "TST";
    else if (lda) opname = "LDA";
```

```
      else if (ldi) opname = "LDI";
      else if (ldm) opname = "LDM";
      else if (sta) opname = "STA";
      else if (sti) opname = "STI";
      else if (jmp) opname = "JMP";
      else if (bra) opname = "BRA";
      else if (skz) opname = "SKZ";
      else if (cal) opname = "CAL";
      else if (ret) opname = "RET";
      else if (brk) opname = "BRK";
      else opname = "UNKNOWN";
      end
  endfunction
endmodule

////////////////////////////////////////////////
// memory
////////////////////////////////////////////////
module memory(adr, data, rwbar);
  input [15:0]  adr;
  inout [7:0]   data;
  input         rwbar;

  // The memory space
  reg [7:0]     mem[65535:0];

  initial
    begin
      $display("memory:  loading initial contents of memory from prog.obj");
      $readmemh("prog.obj",mem);
    end

  // Memory read
  wire [7:0] #3 data = rwbar ?mem[adr] : 8'bz;

  // Memory write
  always
    begin
      if (rwbar == 0) mem[adr] <= #1 data;
      #2;  // Delay, a kludge for simulation
    end
endmodule

////////////////////////////////////////////////
// sexium
////////////////////////////////////////////////
module sexium(clk,reset,adr,data,cs,rwbar,test,test_out);

  // External pins
  input          clk;
  input          reset;
  input  [3:0]   test;
  output [15:0]  adr;
  inout  [7:0]   data;
  output [3:2]   cs;
  output         rwbar;
  output         test_out;
```

```
  // Internal busses
  wire [7:0]    bus;

  // Control signals
  wire          alu_bus_en;
  wire [3:0]    xsel;
  wire          ysel;
  wire [4:0]    zsel;
  wire          rollflag;
  wire          ldacc;
  wire          ldtmp;
  wire          cin;
  wire          cout;
  wire          zero;
  wire          negative;
  wire [3:0]    reg_wr;
  wire [3:0]    reg_rd;
  wire [7:0]    pcma_in;
  wire [3:0]    pcma_rd;
  wire          pcma_select;
  wire          bus_in,bus_out;

  // Top level modules
  tribuf inbuf(bus_in,data,bus);
  tribuf outbuf(bus_out,bus,data);
  alubox alu(clk,bus,alu_bus_en,xsel,ysel,zsel,rollflag,ldacc,ldtmp,
          cin,cout,zero,negative);
  regbox regs(clk,bus,reg_wr,reg_rd);
  pcmabox pcma(clk,bus,pcma_in,pcma_rd,pcma_select,adr,cs);
  control cont(clk,bus,reset,                       // inputs
            xsel,ysel,zsel,alu_bus_en,         // alu control
            rollflag,ldacc,ldtmp,cin,cout,zero,negative,
            reg_wr,reg_rd,                     // reg control
            pcma_rd,pcma_in,pcma_select,       // pcma control
            test_out,test,                     // test muxes
            bus_in,bus_out,                    // bus control
            rwbar);                            // read/write enable
endmodule

/////////////////////////////////////////////
// alubox
/////////////////////////////////////////////
module alubox(clk,bus,alu_bus_en,xsel,ysel,zsel,rollflag,ldacc,ldtmp,
            cin,cout,zero,negative);
  input         clk;
  inout [7:0]   bus;
  input         alu_bus_en;
  input [3:0]   xsel;
  input         ysel;
  input [4:0]   zsel;
  input         rollflag;  // Control input for shifter
  input         ldacc;     // Load enable for accumulator register
  input         ldtmp;     // Load enable for tmp register
  input         cin;
  output        cout;      // When the sum produces a carry
  output        zero;      // When the sum equals zero
  output        negative;  // When the x input is a negative 2's comp integer
```

```
  wire [7:0]    x;          // Input x of ALU
  wire [7:0]    y;          // Input y of ALU
  wire [7:0]    z;          // Output of ALU
  wire [7:0]    sumout;
  wire [7:0]    andout;
  wire [7:0]    negout;
  wire [7:0]    shiftout;
  wire [7:0]    a;          // Output of a register
  wire [7:0]    tmp;        // Output of tmp register
  wire [7:0]    ad;         // Input to a register
  wire [7:0]    tmpd;       // Input to tmp register

  flop areg(clk,ad,a);
  flop tmpreg(clk,tmpd,tmp);
  mux2 amux(a,z,ldacc,ad);
  mux2 tmpmux(tmp,z,ldtmp,tmpd);
  mux4 xmux(a,tmp,8'b0,8'b11111111,xsel,x);
  mux2 ymux(bus,8'b0,ysel,y);
  mux5 zmux(sumout,andout,negout,shiftout,{6'b000000,zero,cout},zsel,z);
  tribuf alubuf(alu_bus_en,x,bus);
  adder op1(x,y,cin,sumout,cout,zero);
  ander op2(x,y,andout);
  neger op3(x,negout);
  shifter op4(x,rollflag,shiftout);

  // Tap off negative bit
  assign #1 negative = x[7];
endmodule

//////////////////////////////////////////////
// regbox
//////////////////////////////////////////////
module regbox(clk,bus,reg_wr,reg_rd);
  input         clk;
  inout [7:0]   bus;
  input [3:0]   reg_wr;
  input [3:0]   reg_rd;

  trilatch r0(clk,reg_rd[0],reg_wr[0],bus);
  trilatch r1(clk,reg_rd[1],reg_wr[1],bus);
  trilatch r2(clk,reg_rd[2],reg_wr[2],bus);
  trilatch r3(clk,reg_rd[3],reg_wr[3],bus);
endmodule

//////////////////////////////////////////////
// pcmabox
//////////////////////////////////////////////
module pcmabox(clk,bus,pcma_in,pcma_rd,pcma_select,adr,cs);
  input         clk;
  inout [7:0]   bus;
  input [7:0]  pcma_in;
  input [3:0]   pcma_rd;
  input         pcma_select;
  output [15:0] adr;
  output [3:2]  cs;

  wire [7:0]    pcld;
  wire [7:0]    pchd;
```

```
  wire [7:0]     mald;
  wire [7:0]     mahd;
  wire [7:0]     pclq;
  wire [7:0]     pchq;
  wire [7:0]     malq;
  wire [7:0]     mahq;
  wire [7:0]     sumh;
  wire [7:0]     suml;
  wire           cout;     // 8th bit carry in 16 bit incrementer
  wire           overflow; // 16th bit carry in incrementer; unused

  flop pcl(clk,pcld,pclq);
  flop pch(clk,pchd,pchq);
  flop mal(clk,mald,malq);
  flop mah(clk,mahd,mahq);
  tribuf pclb(pcma_rd[0],pclq,bus);
  tribuf pchb(pcma_rd[1],pchq,bus);
  tribuf malb(pcma_rd[2],malq,bus);
  tribuf mahb(pcma_rd[3],mahq,bus);
  mux2 muxl(pclq,malq,pcma_select,adr[7:0]);
  mux2 muxh(pchq,mahq,pcma_select,adr[15:8]);

  mux3 pclmux(pclq,bus,suml,{pcma_in[1],pcma_in[0],
                      ~pcma_in[0] && ~pcma_in[1]},pcld);
  mux3 pchmux(pchq,bus,sumh,{pcma_in[3],pcma_in[2],
                      ~pcma_in[2] && ~pcma_in[3]},pchd);
  mux3 malmux(malq,bus,suml,{pcma_in[5],pcma_in[4],
                      ~pcma_in[4] && ~pcma_in[5]},mald);
  mux3 mahmux(mahq,bus,sumh,{pcma_in[7],pcma_in[6],
                      ~pcma_in[6] && ~pcma_in[7]},mahd);
  halfadder incl(adr[7:0],suml,1'b1,cout);
  halfadder inch(adr[15:8],sumh,cout,overflow);
  decoder2to4 csdecode(adr[15:14],cs);
endmodule

//////////////////////////////////////////////
// control
//////////////////////////////////////////////
module control(clk,bus,reset,                        // inputs
             xsel,ysel,zsel,alu_bus_en,          // alu control
             rollflag,ldacc,ldtmp,cin,cout,zero,negative,
             reg_wr,reg_rd,                      // reg control
             pcma_rd,pcma_in,pcma_select,        // pcma control
             test_out,test[3:0],                 // test signals
             bus_in,bus_out,                     // bus control
             rwbar);                             // read/write enable

  input          clk;
  input [7:0]    bus;
  input          reset;   // reset is presently asynchronus
  output [3:0]   xsel;
  output         ysel;
  output [4:0]   zsel;
  output         alu_bus_en;
  output         rollflag;
  output         ldacc;
  output         ldtmp;
  output         cin;
```

```
  input         cout;
  input         zero;
  input         negative;
  input  [3:0]  test;
  output [3:0]  reg_wr;
  output [3:0]  reg_rd;
  output [3:0]  pcma_rd;
  output [7:0]  pcma_in;
  output        pcma_select;
  output        bus_in;
  output        bus_out;
  output        rwbar;
  output        test_out;

  // internal state
  reg [7:0]     ir;
  reg [2:0]     s;
  reg           f;
  reg           carry;
  reg           neg;

  // internal signals
  wire          clrs;
  wire          newf;
  wire          latchcarry;
  wire          latchneg;
  wire          regop;

  wire          ldop;
  wire          store;

  control_pla pla(alu_bus_en, bus_in, cin, ldacc, ldtmp, pcma_in, pcma_rd,
        pcma_select, reg_rd, reg_wr, rollflag, xsel[3:1], ysel, zsel[4:1],
        carry, neg, reset, zero, ldop, newf, clrs, latchneg, latchcarry,
        s, ir, f, store, regop, regop);
  mux16 tester(test,test_out,{reset,reset,clk,f,s[0],bus[7:0],ysel,ir[0],neg});
  // not quite accurate--doesn't model ring oscillator or t flip-flop

  // update state
  always @(posedge clk)
    begin
      if (ldop) ir <= #1 bus;
      if (clrs) s <= #1 0;
      else s <= #1 s+1;
      f <= #1 newf;
      if (latchcarry) carry <= cout;
      if (latchneg) neg <= negative;
    end

  // random logic
  assign #1 xsel[0] = ~xsel[1] && ~xsel[2] && ~xsel[3];
  assign #1 zsel[0] = ~zsel[1] && ~zsel[2] && ~zsel[3] && ~zsel[4];
  assign #1 rwbar = clk || ~store;
  assign #1 bus_out = ~rwbar;

endmodule

////////////////////////////////////////////////
```

```
// control_pla
/////////////////////////////////////////////
module control_pla (alu_bus_en, bus_in, cin, ldacc, ldtmp, pcma_in, pcma_rd,
           pcma_select, reg_rd, reg_wr, rollflag, xsel, ysel, zsel, carry,
           neg, reset, zero, ldop, newf, clrs, latchneg, latchcarry,
           s, ir, f, store, regop, regopin);
    output regop;
    input regopin;

    output alu_bus_en;
    output bus_in;
    output cin;
    output ldacc;
    output ldtmp;
    output [7:0] pcma_in;
    output [3:0] pcma_rd;
    output pcma_select;
    output [3:0] reg_rd;
    output [3:0] reg_wr;
    output rollflag;
    output [3:1] xsel;
    output ysel;
    output [4:1] zsel;
    input carry;
    input neg;
    input reset;
    input zero;
    output ldop;
    output newf;
    output clrs;
    output latchneg;
    output latchcarry;
    input [2:0] s;
    input [7:0] ir;
    input f;
    output store;

  // internal signals
  wire          add, andw, notw, shr, ror, put, get, tst;
  wire          lda, ldi,  ldm,  sta, sti, jmp0, jmp1, jmp2;
  wire          bra0, bra1, bra2, bra3, bra4, skz0, skz1;
  wire          cal0, cal1, cal2, cal3, cal4, cal5, ret0, ret1;
  wire [7:0]    r;

  // product terms

  wire bra32wn,skz0wz,skz1wz,skz1wnz,regopinp;
  wire r0rd,r1rd,r2rd,r3rd,r4rd,r5rd,r6rd,r7rd;
  wire bra3wnn,r0wr,r1wr,r2wr,r3wr,r4wr,r5wr,r6wr,r7wr;
  wire resetp,fp,frst,prod55,prod56;

// Product terms
  assign #1 add = ~ir[4] && ir[3] && ~ir[2] && ~ir[1] && ~ir[0] && ~reset &&
      ~f;
  assign #1 andw = ~ir[4] && ir[3] && ~ir[2] && ~ir[1] && ir[0] && ~reset &&
      ~f;
  assign #1 notw = ~ir[4] && ir[3] && ir[2] && ~ir[1] && ~ir[0] && ~reset &&
      ~f;
```

```
assign #1 shr = ~ir[4] && ir[3] && ir[2] && ~ir[1] && ir[0] && ~reset && ~f;
assign #1 ror = ~ir[4] && ir[3] && ir[2] && ir[1] && ~ir[0] && ~reset && ~f;
assign #1 put = ~ir[4] && ir[3] && ~ir[2] && ir[1] && ir[0] && ~reset && ~f;
assign #1 get = ~ir[4] && ir[3] && ir[2] && ir[1] && ir[0] && ~reset && ~f;
assign #1 tst = ~ir[4] && ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~reset &&
    ~f;
assign #1 lda = ~ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ~ir[0] && ~reset &&
    ~f;
assign #1 ldi = ~ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ir[0] && ~reset &&
    ~f;
assign #1 ldm = ~ir[4] && ~ir[3] && ir[2] && ~ir[1] && ~ir[0] && ~reset &&
    ~f;
assign #1 sta = ~ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~reset &&
    ~f;
assign #1 sti = ~ir[4] && ~ir[3] && ~ir[2] && ir[1] && ir[0] && ~reset &&
    ~f;
assign #1 jmp0 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ~ir[0] && ~s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f;
assign #1 jmp1 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ~ir[0] && ~s[2] &&
    ~s[1] && s[0] && ~reset && ~f;
assign #1 jmp2 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ~ir[0] && ~s[2] &&
    s[1] && ~s[0] && ~reset && ~f;
assign #1 bra0 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f;
assign #1 bra1 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~s[2] &&
    ~s[1] && s[0] && ~reset && ~f;
assign #1 bra2 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~s[2] &&
    s[1] && ~s[0] && ~reset && ~f;
assign #1 bra3 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~s[2] &&
    s[1] && s[0] && ~reset && ~f;
assign #1 bra4 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f;
assign #1 bra3wnn = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~s[2] &&
    s[1] && s[0] && ~reset && ~f && ~neg;
assign #1 cal0 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ir[0] && ~s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f;
assign #1 cal1 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ir[0] && ~s[2] &&
    ~s[1] && s[0] && ~reset && ~f;
assign #1 cal2 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ir[0] && ~s[2] &&
    s[1] && ~s[0] && ~reset && ~f;
assign #1 cal3 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ir[0] && ~s[2] &&
    s[1] && s[0] && ~reset && ~f;
assign #1 cal4 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ir[0] && s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f;
assign #1 cal5 = ir[4] && ~ir[3] && ~ir[2] && ~ir[1] && ir[0] && s[2] &&
    ~s[1] && s[0] && ~reset && ~f;
assign #1 ret0 = ir[4] && ~ir[3] && ir[2] && ~ir[1] && ~ir[0] && ~s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f;
assign #1 ret1 = ir[4] && ~ir[3] && ir[2] && ~ir[1] && ~ir[0] && ~s[2] &&
    ~s[1] && s[0] && ~reset && ~f;
assign #1 skz0 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ir[0] && ~s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f;
assign #1 skz1 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ir[0] && ~s[2] &&
    ~s[1] && s[0] && ~reset && ~f;
assign #1 skz0wz = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ir[0] && ~s[2] &&
    ~s[1] && ~s[0] && ~reset && ~f && zero;
assign #1 skz1wz = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ir[0] && ~s[2] &&
    ~s[1] && s[0] && ~reset && ~f && zero;
```

```
  assign #1 skz1wnz = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ir[0] && ~s[2] &&
      ~s[1] && s[0] && ~reset && ~f && ~zero;
  assign #1 regopinp = regopin;
  assign #1 r0rd = ~ir[7] && ~ir[6] && ~ir[5] && regopin;
  assign #1 r1rd = ~ir[7] && ~ir[6] && ir[5] && regopin;
  assign #1 r2rd = ~ir[7] && ir[6] && ~ir[5] && regopin;
  assign #1 r3rd = ~ir[7] && ir[6] && ir[5] && regopin;
  assign #1 r4rd = ir[7] && ~ir[6] && ~ir[5] && regopin;
  assign #1 r5rd = ir[7] && ~ir[6] && ir[5] && regopin;
  assign #1 r6rd = ir[7] && ir[6] && ~ir[5] && regopin;
  assign #1 r7rd = ir[7] && ir[6] && ir[5] && regopin;
  assign #1 r0wr = ~ir[7] && ~ir[6] && ~ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 r1wr = ~ir[7] && ~ir[6] && ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 r2wr = ~ir[7] && ir[6] && ~ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 r3wr = ~ir[7] && ir[6] && ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 r4wr = ir[7] && ~ir[6] && ~ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 r5wr = ir[7] && ~ir[6] && ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 r6wr = ir[7] && ir[6] && ~ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 r7wr = ir[7] && ir[6] && ir[5] && ~ir[4] && ir[3] && ~ir[2] &&
      ir[1] && ir[0] && ~reset && ~f;
  assign #1 resetp = reset;
  assign #1 fp = f;
  assign #1 frst = f && ~reset;
  assign #1 prod55 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~s[2] &&
      s[1] && s[0] && ~reset && ~f && neg;
  assign #1 prod56 = ir[4] && ~ir[3] && ~ir[2] && ir[1] && ~ir[0] && ~s[2] &&
      s[1] && s[0] && ~reset && ~f && carry;

// Outputs
  assign #1 regop = add || andw || get || tst;
  assign #1 newf = add || andw || notw || shr || ror || put || get || tst ||
      lda || ldi || ldm || sta || sti || jmp2 || bra4 || cal5 || ret1 ||
      skz1 || resetp;
  assign #1 clrs = add || andw || notw || shr || ror || put || get || tst ||
      lda || ldi || ldm || sta || sti || jmp2 || bra4 || cal5 || ret1 ||
      skz1 || resetp || fp;
  assign #1 latchcarry = add || bra1;
  assign #1 latchneg = bra1;
  assign #1 reg_rd[3] = r3rd;
  assign #1 reg_wr[3] = r3wr;
  assign #1 reg_wr[2] = r2wr;
  assign #1 reg_rd[2] = r2rd;
  assign #1 reg_rd[1] = cal3 || ret1 || r1rd;
  assign #1 reg_wr[1] = cal1 || cal4 || r1wr;
  assign #1 reg_wr[0] = cal2 || r0wr;
  assign #1 reg_rd[0] = ret0 || r0rd;
  assign #1 pcma_rd[0] = bra1 || cal2 || r4rd;
  assign #1 pcma_rd[1] = bra3 || cal4 || r5rd;
  assign #1 pcma_rd[2] = r6rd;
  assign #1 pcma_rd[3] = r7rd;
  assign #1 pcma_in[0] = jmp1 || bra2 || cal3 || ret0 || r4wr || resetp;
```

```
  assign #1 pcma_in[1] = ldm || jmp0 || cal0 || cal1 || skz0wz || skz1wz ||
      frst;
  assign #1 pcma_in[2] = jmp2 || bra4 || cal5 || ret1 || r5wr || resetp;
  assign #1 pcma_in[3] = pcma_in[1];
  assign #1 pcma_in[4] = r6wr;
  assign #1 pcma_in[5] = ldi || sti;
  assign #1 pcma_in[6] = r7wr;
  assign #1 pcma_in[7] = pcma_in[5];
  assign #1 pcma_select = lda || ldi || sta || sti;
  assign #1 alu_bus_en = put || sta || sti || jmp2 || bra2 || bra4 || cal5 ||
      resetp;
  assign #1 ldtmp = jmp0 || bra0 || bra1 || bra3 || cal0;
  assign #1 ldacc = notw || shr || ror || get || tst || lda || ldi || ldm ||
      regopinp;
  assign #1 xsel[1] = jmp2 || bra1 || bra2 || bra4 || cal5;
  assign #1 xsel[2] = get || lda || ldi || ldm || jmp0 || jmp1 || bra0 ||
      bra3wnn || cal0 || resetp;
  assign #1 xsel[3] = prod55;
  assign #1 ysel = put || sta || sti || jmp2 || bra2 || bra4 || cal5 ||
      skz0 || skz1 || resetp;
  assign #1 cin = prod56;
  assign #1 rollflag = ror;
  assign #1 zsel[1] = andw;
  assign #1 zsel[2] = notw;
  assign #1 zsel[3] = shr || ror;
  assign #1 zsel[4] = tst;
  assign #1 ldop = fp;
  assign #1 bus_in = lda || ldi || ldm || jmp0 || jmp1 || bra0 || cal0 ||
      cal1 || skz1wnz || frst;
  assign #1 store = sta || sti;

endmodule

/////////////////////////////////////////////
// mux2
/////////////////////////////////////////////
module mux2(in1,in2,sel,out);
  input [7:0]   in1;
  input [7:0]   in2;
  input         sel;
  output [7:0]  out;

  assign #1 out = (sel == 0) ? in1 : in2;
endmodule

/////////////////////////////////////////////
// mux3
/////////////////////////////////////////////
module mux3(in1,in2,in3,sel,out);
  input [7:0]  in1;
  input [7:0]  in2;
  input [7:0]  in3;
  input [2:0]  sel;
  output [7:0] out;

  assign #1 out = (sel[0] == 1) ? in1 :
      (sel[1] == 1) ? in2 :
        (sel[2] == 1) ? in3 :
```

```
            8'bz;
endmodule

//////////////////////////////////////////////
// mux4
//////////////////////////////////////////////
module mux4(in1,in2,in3,in4,sel,out);
  input [7:0]  in1;
  input [7:0]  in2;
  input [7:0]  in3;
  input [7:0]  in4;
  input [3:0]  sel;
  output [7:0] out;

  assign #1 out = (sel[0] == 1) ? in1 :
      (sel[1] == 1) ? in2 :
        (sel[2] == 1) ? in3 :
            (sel[3] == 1) ? in4 :
              8'bz;
endmodule

//////////////////////////////////////////////
// mux5
//////////////////////////////////////////////
module mux5(in1,in2,in3,in4,in5,sel,out);
  input [7:0]  in1;
  input [7:0]  in2;
  input [7:0]  in3;
  input [7:0]  in4;
  input [7:0]  in5;
  input [4:0]  sel;
  output [7:0] out;

  assign #1 out = (sel[0] == 1) ? in1 :
      (sel[1] == 1) ? in2 :
        (sel[2] == 1) ? in3 :
            (sel[3] == 1) ? in4 :
              (sel[4] == 1) ? in5 :
                  8'bz;
endmodule

//////////////////////////////////////////////
// adder
//////////////////////////////////////////////
module adder(in1,in2,cin,out,cout,zero);
  input [7:0]  in1;
  input [7:0]  in2;
  input        cin;
  output [7:0] out;
  output       cout;
  output       zero;

  assign #1 out = in1+in2+cin;
  assign #1 cout = (in1+in2+cin > 255);
  assign #1 zero = ((in1+in2+cin)%256 == 0);
endmodule

//////////////////////////////////////////////
```

```
// ander
/////////////////////////////////////////////
module ander(in1,in2,out);
  input [7:0]  in1;
  input [7:0]  in2;
  output [7:0] out;

  assign #1 out = in1 & in2;
endmodule

/////////////////////////////////////////////
// neger
/////////////////////////////////////////////
module neger(in,out);
  input [7:0]  in;
  output [7:0] out;

  assign #1 out = ~in;
endmodule

/////////////////////////////////////////////
// shifter
/////////////////////////////////////////////
module shifter(in,rollflag,out);
  input [7:0]  in;
  input        rollflag;
  output [7:0] out;

  assign #1 out[6:0] = in[7:1];
  assign #1 out[7] = (rollflag == 1) ? in[0] : 0;
endmodule

/////////////////////////////////////////////
// trilatch
/////////////////////////////////////////////
module trilatch(clk,rden,wren,bus);
  input        clk;
  input        rden;
  input        wren;
  inout [7:0]  bus;

  reg [7:0]    state;

  always
    begin
      if (wren == 1 && ~clk) state = bus;
      #2;
    end

  assign #1 bus = (rden == 1) ? state : 8'bz;
endmodule

/////////////////////////////////////////////
// flop
/////////////////////////////////////////////
module flop(clk,d,q);
  input        clk;
  input [7:0]  d;
```

```
  output [7:0] q;

  reg [7:0] state;

  always @(posedge clk)
      state <= #1 d;

  assign #1 q = state;
endmodule

/////////////////////////////////////////////
// halfadder
/////////////////////////////////////////////
module halfadder(in,out,cin,cout);
  input [7:0]  in;
  output [7:0] out;
  input        cin;
  output       cout;

  assign #1 out = in+cin;
  assign #1 cout = (in == 8'b11111111) && cin;
endmodule

/////////////////////////////////////////////
// decoder2to4
/////////////////////////////////////////////
module decoder2to4(sel,out);
  input [1:0]  sel;
  output [3:2] out;

  assign #1 out[2] = sel[0] || ~sel[1];
  assign #1 out[3] = ~sel[0] || ~sel[1];
endmodule

/////////////////////////////////////////////
// tribuff
/////////////////////////////////////////////
module tribuf( en, a, y);

  input        en;
  input [7:0]  a;
  output [7:0] y;

  wire [7:0] #1 y = en ? a : 8'bz;
endmodule

/////////////////////////////////////////////
// mux16
/////////////////////////////////////////////
module mux16(sel,out,in[15:0]);
  input [3:0]   sel;
  output        out;
  input [15:0]  in;

  assign #1 out = in[sel];
endmodule
```

Microcode PLA Equations

```
# Control.pla
# Created by Ethan Mirsky for 6.008

# Last edited 2/2/94 by David Harris
#  Changed [n] to <n>

# Inputs in order
Inputorder  reset s<2> s<1> s<0> carry zero neg f ir<7> ir<6> ir<5>
      ir<4> ir<3> ir<2> ir<1> ir<0> regopin

# Product Terms:

add   : ir<4>' * ir<3> * ir<2>' * ir<1>' * ir<0>' * reset' * f'
andw  :  ir<4>' * ir<3> * ir<2>' * ir<1>' * ir<0> * reset' * f'
notw  :  ir<4>' * ir<3> * ir<2> * ir<1>' * ir<0>' * reset' * f'
shr  :  ir<4>' * ir<3> * ir<2> * ir<1>' * ir<0> * reset' * f'
ror  :  ir<4>' * ir<3> * ir<2> * ir<1> * ir<0>' * reset' * f'
put  :  ir<4>' * ir<3> * ir<2>' * ir<1> * ir<0> * reset' * f'
get  :  ir<4>' * ir<3> * ir<2> * ir<1> * ir<0> * reset' * f'
tst  :  ir<4>' * ir<3> * ir<2>' * ir<1> * ir<0>' * reset' * f'

lda  :  ir<4>' * ir<3>' * ir<2>' * ir<1>' * ir<0>' * reset' * f'
ldi  :  ir<4>' * ir<3>' * ir<2>' * ir<1>' * ir<0> * reset' * f'
ldm  :  ir<4>' * ir<3>' * ir<2> * ir<1>' * ir<0>' * reset' * f'

sta  :  ir<4>' * ir<3>' * ir<2>' * ir<1> * ir<0>' * reset' * f'
sti  :  ir<4>' * ir<3>' * ir<2>' * ir<1> * ir<0> * reset' * f'

jmp0  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0>' * s<2>' * s<1>' *
      s<0>' * reset' * f'
jmp1  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0>' * s<2>' * s<1>' *
      s<0> * reset' * f'
jmp2  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0>' * s<2>' * s<1> *
      s<0>' * reset' * f'

bra0  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2>' * s<1>' *
      s<0>' * reset' * f'
bra1  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2>' * s<1>' *
      s<0> * reset' * f'
bra2  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2>' * s<1> *
      s<0>' * reset' * f'
bra3  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2>' * s<1> *
      s<0> * reset' * f'
bra4  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2> * s<1>' *
      s<0>' * reset' * f'
#  bra3 with neg':
bra3wnn   :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2>' * s<1> *
      s<0> * reset' * f' * neg'

cal0  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0> * s<2>' * s<1>' *
      s<0>' * reset' * f'
cal1  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0> * s<2>' * s<1>' *
      s<0> * reset' * f'
cal2  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0> * s<2>' * s<1> *
      s<0>' * reset' * f'
```

```
cal3  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0> * s<2>' * s<1> *
      s<0> * reset' * f'
cal4  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0> * s<2> * s<1>' *
      s<0>' * reset' * f'
cal5  :  ir<4> * ir<3>' * ir<2>' * ir<1>' * ir<0> * s<2> * s<1>' *
      s<0> * reset' * f'

ret0  :  ir<4> * ir<3>' * ir<2> * ir<1>' * ir<0>' * s<2>' * s<1>' *
      s<0>' * reset' * f'
ret1  :  ir<4> * ir<3>' * ir<2> * ir<1>' * ir<0>' * s<2>' * s<1>' *
      s<0> * reset' * f'

skz0  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0> * s<2>' * s<1>' *
      s<0>' * reset' * f'
skz1  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0> * s<2>' * s<1>' *
      s<0> * reset' * f'
# These are  skz's and zero:
skz0wz  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0> * s<2>' * s<1>' *
      s<0>' * reset' * f' * zero
skz1wz  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0> * s<2>' * s<1>' *
      s<0> * reset' * f' * zero
# This is skz1 and zero':
skz1wnz  :  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0> * s<2>' * s<1>' *
      s<0> * reset' * f' * zero'

# A product term for the regop term
regopinp  :  regopin

# regopin is the fed-back regop:
r0rd  :  ir<7>' * ir<6>' * ir<5>' * regopin
r1rd  :  ir<7>' * ir<6>' * ir<5> * regopin
r2rd  :  ir<7>' * ir<6> * ir<5>' * regopin
r3rd  :  ir<7>' * ir<6> * ir<5> * regopin
r4rd  :  ir<7> * ir<6>' * ir<5>' * regopin
r5rd  :  ir<7> * ir<6>' * ir<5> * regopin
r6rd  :  ir<7> * ir<6> * ir<5>' * regopin
r7rd  :  ir<7> * ir<6> * ir<5> * regopin

# These are put anded with the particular register select:
r0wr  :  ir<7>' * ir<6>' * ir<5>' * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'
r1wr  :  ir<7>' * ir<6>' * ir<5> * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'
r2wr  :  ir<7>' * ir<6> * ir<5>' * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'
r3wr  :  ir<7>' * ir<6> * ir<5> * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'
r4wr  :  ir<7> * ir<6>' * ir<5>' * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'
r5wr  :  ir<7> * ir<6>' * ir<5> * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'
r6wr  :  ir<7> * ir<6> * ir<5>' * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'
r7wr  :  ir<7> * ir<6> * ir<5> * ir<4>' * ir<3> * ir<2>' * ir<1> *
      ir<0> * reset' * f'

#  These are reset terms, so they will not be repeated.
resetp  :  reset
```

```
#  Product term for f
fp  :  f

frst  :  f * reset'


# Outputs in order:
regop  =  add + andw + get + tst
newf  =  resetp + add + andw + notw + shr + ror + put + get + tst +
      lda + ldi + ldm + sta + sti + jmp2 + bra4 + skz1 + cal5 + ret1
clrs  =  resetp + add + andw + notw + shr + ror + put + get + tst +
      lda + ldi + ldm + sta + sti + jmp2 + bra4 + skz1 + cal5 + ret1 + fp
latchcarry  =  add + bra1
latchneg  =  bra1

reg_rd<3>  =  r3rd
reg_wr<3>  =  r3wr
reg_wr<2>  =  r2wr
reg_rd<2>  =  r2rd
reg_rd<1>  =  r1rd + ret1 + cal3
reg_wr<1>  =  r1wr + cal1 + cal4
reg_wr<0>  =  r0wr + cal2
reg_rd<0>  =  r0rd + ret0

pcma_rd<1>  =  r5rd + bra3 + cal4
pcma_in1<2>  =  resetp + r5wr + jmp2 + bra4 + cal5 + ret1
pcma_rd<3>  =  r7rd
pcma_in2<6>  =  r7wr
pcma_rd<0>  =  r4rd + bra1 + cal2
pcma_in1<0>  =  resetp + r4wr + jmp1 +bra2 + cal3 + ret0
pcma_in1<1>  =  frst + ldm + skz0wz + skz1wz + jmp0 + cal0 + cal1
pcma_select  =  lda + ldi + sta + sti
pcma_rd<2>  =  r6rd
pcma_in2<4>  =  r6wr
pcma_in2<5>  =  ldi + sti

alu_bus_en  =  resetp + put + sta + sti + jmp2 + bra2 + bra4 + cal5
ldtmp  =  jmp0 + bra0 + bra1 + bra3 + cal0
ldacc  =  ldm  + regopinp  + notw + shr + ror + get + lda + ldi + tst
xsel<1>  =  jmp2 + bra1 + bra2 + bra4 + cal5
xsel<2>  =  resetp + ldm + get + lda + ldi + jmp0 + jmp1 + bra0 +
      bra3wnn + cal0
xsel<3>  =  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2>' * s<1> *
      s<0> * reset' * f' * neg
ysel  =  put + sta + sti + jmp2 + skz0 + skz1 + cal5 + resetp + bra2 +
      bra4
cin  =  ir<4> * ir<3>' * ir<2>' * ir<1> * ir<0>' * s<2>' * s<1> * s<0> *
      reset' * f' * carry
rollflag  =  ror
zsel<1>  =  andw
zsel<2>  =  notw
zsel<3>  =  shr + ror
zsel<4>  =  tst

ldop  =  fp
bus_in  =  frst + ldm + lda + ldi + skz1wnz + jmp0 + jmp1 + bra0 +
      cal0 + cal1
```

```
store  =  sta + sti
```

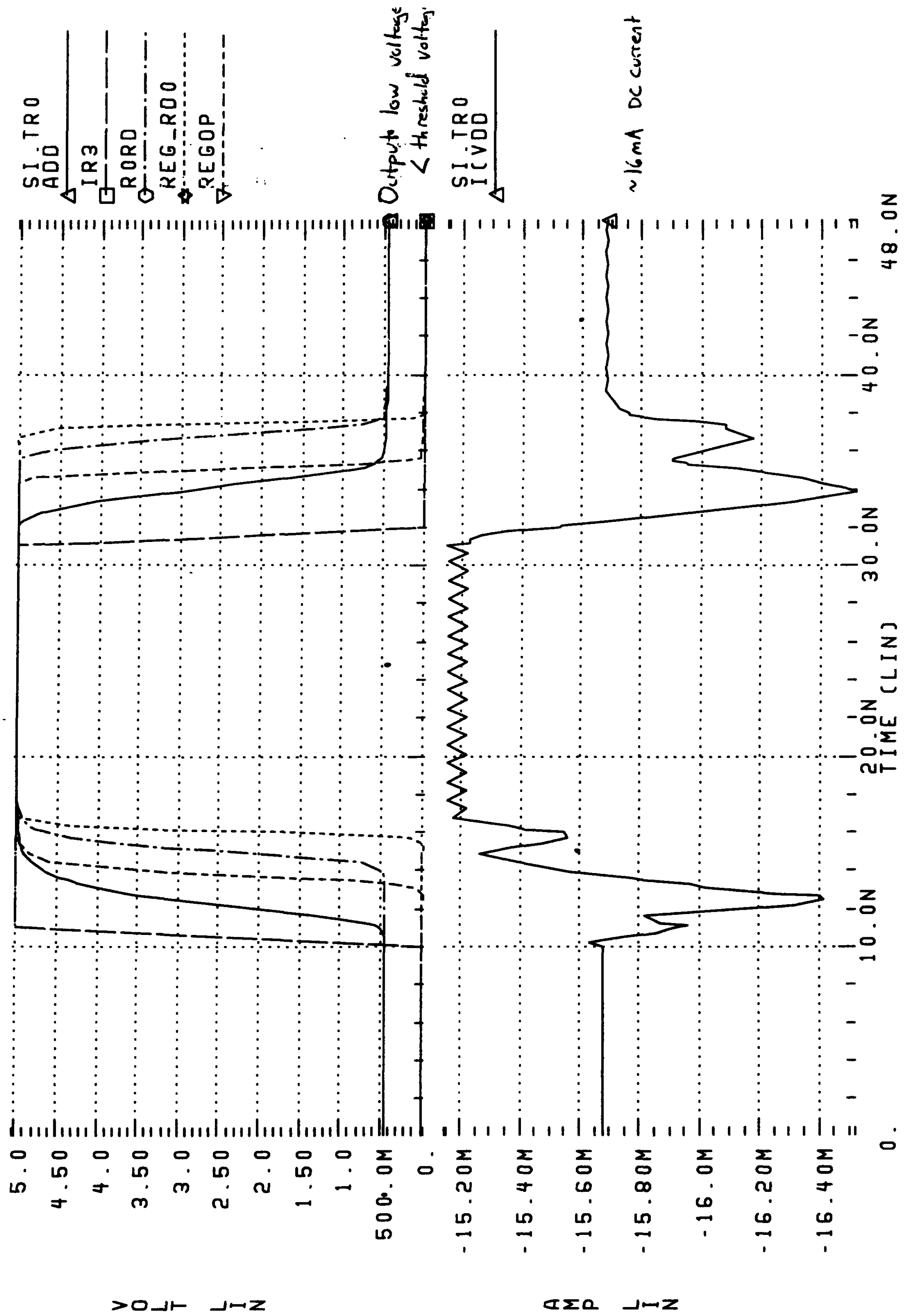
HSPICE Simulations

This section contains several HSPICE runs verifying the PLA and I/O pad cells. All rise and fall times are approximate because they neglect parasitic capacitances and resistances which were not available in the HSPICE model.

PLA Simulation

The critical limitations on the PLA are that it draw low enough current to avoid electromigration in the power rails and that internal voltages drop below the threshold of a subsequent NMOS transistor. Also, no schematic of the PLA was available, so the HSPICE simulation was used to verify proper PLA operation. The following simulation shows that the control PLA functions correctly.

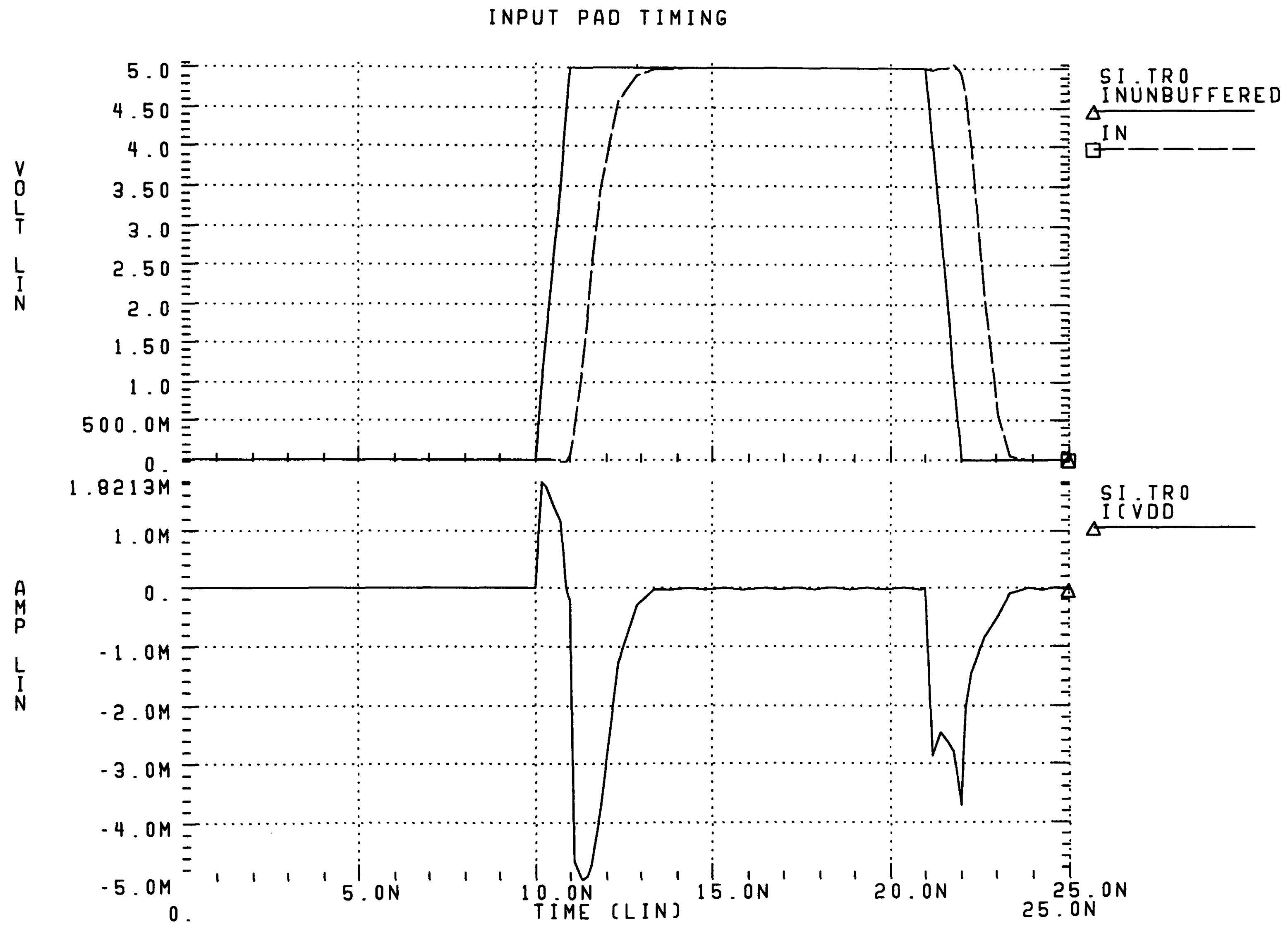
```
* Hold most inputs at 0
* Toggle bit 3 of IR high
* This should simulate ADD R0
*   First add product line goes high
*   Then regop outputs goes high
*   This causes reg0rd product line to go high
*   This causes reg_rd0 to go high
* When IR3 goes back low, process should reverse
*
* add doesn't get labeled in PLA
* thus if the control_pla is renetlisted, add must
* be searched for and replaced

VDD vdd gnd 5

* Inputs

Vi0 ir0 gnd 0
Vi1 ir1 gnd 0
Vi2 ir2 gnd 0
Vi3 ir3 gnd PULSE(0 5 10ns 1ns 1ns 20ns 40ns)
Vi4 ir4 gnd 0
Vi5 ir5 gnd 0
Vi6 ir6 gnd 0
Vi7 ir7 gnd 0
Vs0 s0 gnd 0
Vs1 s1 gnd 0
Vs2 s2 gnd 0
Vf f gnd 0
Vcarry carry gnd 0
Vneg neg gnd 0
Vreset reset gnd 0
Vzero zero gnd 0

Rloop regop regopin 10              * Feedback from regop to regopin
Runused unused gnd 1M               * Don't let unused node float

* Simulation

.options post
.tran .1n 48n
.plot v(ir3)
.plot v(add)
.plot v(regop)
.plot v(r0rd)
.plot v(reg_rd0)
```

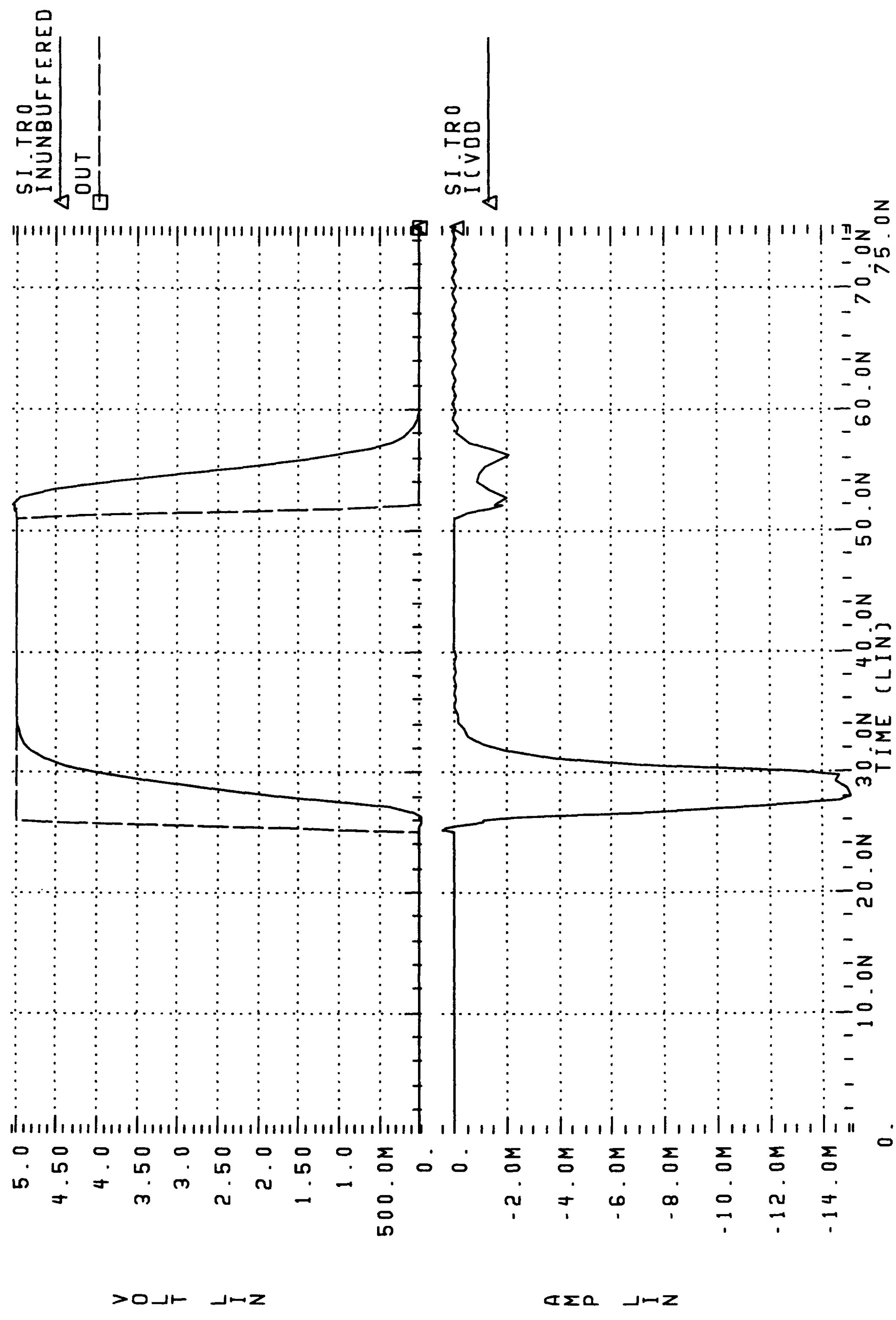

PLA TIMING, VOLTAGE SWINGS, AND CURRENT DRAIN
SI.TR0
ADD
IR3
RORD
REG_RDO
REGOP
Output low voltage
< threshold voltage
SI.TR0
I(VDD)
~16mA DC current
VOLT LIN
AMP LIN
5.0
4.50
4.0
3.50
3.0
2.50
2.0
1.50
1.0
500.0M
0.
-15.20M
-15.40M
-15.60M
-15.80M
-16.0M
-16.20M
-16.40M
0.
10.0N
20.0N
30.0N
40.0N
48.0N
TIME (LIN)

I/O Pad Simulation

The I/O pads were provided by MOSIS in CIF format and had to be read into Cadence and manually patched. Therefore, a simulation of the pads was run to verify proper behavior and to estimate switching current.

```
* These lines are always used
VDD padvdd padgnd 5
VGND padgnd gnd 0
Cload inunbuffered padgnd 10pF
Cchip in padgnd 1pF
.options post

* These lines simulate data coming in an input pad
*Rbogus out padgnd 1G
*Vpad inunbuffered padgnd PULSE(0 5 10ns 1ns 1ns 10ns 25ns)
*Venable enable padgnd 0
*.tran .1ns 25ns
*.plot v(inunbuffered)
*.plot v(in)

* These lines simulate data being driven out an output pad
Vin out padgnd PULSE (0 5 25ns 1ns 1ns 25ns 50ns)
Venable enable padgnd 5
.tran .1ns 75ns
.plot v(out)
.plot v(inunbuffered)
```


INPUT PAD TIMING
SI.TR0
INUNBUFFERED
IN
SI.TR0
I(VDD
VOLT LIN
AMP LIN
5.0
4.50
4.0
3.50
3.0
2.50
2.0
1.50
1.0
500.0M
0.
1.8213M
1.0M
0.
-1.0M
-2.0M
-3.0M
-4.0M
-5.0M
0.
5.0N
10.0N
15.0N
20.0N
25.0N
25.0N
TIME (LIN)


* DEFAULT HSPICE SIMULATION RUN TITLE CARD.
5-FEB94 17:32:58
SI_TRO
INUNBUFFERED
OUT
SI_TRO
I(VDD
VOLT LIN
AMP LIN
5.0
4.50
4.0
3.50
3.0
2.50
2.0
1.50
1.0
500.0M
0.
-2.0M
-4.0M
-6.0M
-8.0M
-10.0M
-12.0M
-14.0M
10.0N
20.0N
30.0N
40.0N
50.0N
60.0N
70.0N
75.0N
TIME (LIN)

Appendix D: Tools

This appendix contains code for the various tools developed on the Sexium project, including msim, the regression suite, and the PLA generator. It also describes the organization of the 6090user directory containing these and other useful files.

Locating tools on-line

The files related to the Sexium project are all located in the /home/cva2/6090user directory. This directory is located on impulse.ai.mit.edu; it was moved there after kiwi's large disk crashed. The contents of the 6090user directory are as follows:

cif:			**CIF files and utilities**
	sexium.cif		the final chip, as sent to MOSIS
	frame.cif		the MOSIS pad frame
	template		a template with parameters for producing CIF output
	ciftocadence		a layer mapping file for reading CIF files
	cadencetocif		a layer mapping file for writing CIF
	howtoreadcif		documentation on reading CIF
control			**Control PLA**
	control.pla		control logic equations fed to PLA generator
doc			**Miscellaneous documentation**
	setup_procedure		steps to create Sexium project and technology files
	scribe		notes on being a scribe
	roster		the class roster
	project		Professor Dally's original project description
	plotting*		notes on using the Building 39 and 7AI plotters
dotfiles			**Cadence dotfiles for Sexium project**
hspice			**HSPICE simulations**
libs			**Sexium project libraries**
	examples		various examples from class
	frame		MOSIS pad frame (10x too small, use unintel pads instead)
	stdcells		student-generated standard cell library
	unintel		sexium microprocessor
	omv		class assignments of Dan Hartman (as an example)
model			**Verilog model of Sexium**
	sexium.v		the model
	prog.obj		inital memory contents to run regression; created by msim
	runv		type runv sexium.v to run simulation
mosis			**MOSIS-related files**
	fabschedule		schedule of 1993-1994 fabrication dates
	pads.doc		MOSIS' padframe documentation
	price		price list
	submission		
		newproject	request to create new project
		submit	request to submit CIF file
		fabricate	request to fabricate chip
techfiles			**Sexium technology files**
	mosis2n.tf		compile this techfile; it sources all others
tools			**Tools created for project**
	msim		
		msim.c	source code for MSIM simulator
		opcodes.h	include file for MSIM simulator
		regress.asm	regression suite for Sexium architecture
		fib.asm	compute fib. numbers

		sum.asm	example of 16 bit addition
	skill		
		makelvsview	create LVS view of cell from extracted view
		find.il	find off-grid rectangles and fix them
		grow.il	expand cells by factor of 2 (change to 10 to fix CIF)
	pla		
		plagen.il	the PLA generator program
		adder.pla	full-adder equations
		coke.pla	coke machine equations
verilog			**Verilog simulations**
	sexium		Run directory for verilog simulation of Sexium
		runver	run this to execute simulation
omv			**Dan Hartman's user account**
ragin			**Ruben Agin's user account**

The session configuration information for the project is located in:

/cds/local/skill/projects/SEXIUM

The mosis2n library with rudimentary HSPICE models is located in:

/cds/local/lib/libs

msim

```
/* sim.c */

/* Written by David Harris

    This is the Sexium simulator, created for
    the 6.008 Intro to VLSI class during IAP 1994.

    Version 1.0:        1/2/94

    Conventions:
        Case insensitive
        labels sensitive to 24 characters
        numbers in hex, start with $
        LOADIMM must use constant, not label

    In addition to the MAYBE NOT instructions, the
    simulator supports the BRK (break) instruction
    for debugging, etc.

*/

/* #includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#include "opcodes.h"

/* Constants */

#define MAXOPS 65536
#define MAXLABELS 256
#define MAXLABELLEN 25

#define UNDEFINED -1

#define TRUE 1
#define FALSE 0

/* Types */

typedef unsigned char BOOL;
typedef unsigned char BYTE;
typedef struct label {
                char    name[MAXLABELLEN];
                int     value;
            } label;

typedef struct patch {
                int     labelNum;
                int     progAddr;
                BYTE    bytes;
```

```
                struct patch *next;
            } patch;

/* Assembler Globals */

char     inputFName[80];
int      numLines;
int      numLabels;
int      numOps;
clock_t startTime;
BYTE     prog[MAXOPS];
label    labelLib[MAXLABELS];
patch    *patchList,*patchCur;

/* Simulator Globals */

BYTE     registers[10];
long     count;
BOOL     breakFlag;
int      breakPoint;
BYTE     oldBreakValue;

/* Prototypes */

void     assembleFile(void);
void     processLine(char*);
BOOL     isLabelDef(char*);
int      addLabel(char*,BOOL);
BYTE     isOp(char*);
void     addOp(BYTE,char*);
void     addSimpleOp(BYTE,char*);
void     addRegOp(BYTE,char*);
void     addByteOp(BYTE,char*,int);
void     insertNum(char*,int);
void     makeLabelPatch(char*,int);
void     patchLabels(void);
void     displayStats(void);

void     simulateCode(void);
void     resetSimulation(void);
void     dumpRegisters(void);
int      disassemble(int);
int      labelFromAddr(int);
void     makeSimpleOp(BYTE);
void     makeRegOp(BYTE);
void     makeByteOp(BYTE,int);
void     makeNumber(BYTE);
void     runSimulation(BOOL);
void     doStep(void);
BYTE     getMem(BYTE,int);
void     putMem(BYTE);
void     increment(BYTE,int);
BYTE     testConditions(BYTE,BYTE);
void     doMemory(void);
void     disassembleMemory(void);
void     readMemory(void);
void     writeMemory(void);
void     doBreakpoint(void);
```

```
/* Main */

void main(int argc, char *argv[])
{
    /* On Mac, prompt for file to compile.  Otherwise use command line */

    #ifdef THINK_C
        printf ("Assembly file: ");
        scanf ("%s",inputFName);
    #else
        if (argc != 2) {
            fprintf (stderr,"Usage: %s assembly_file.\n",argv[0]);
            exit(1);
        }
        else strcpy(inputFName,argv[1]);
    #endif

    assembleFile();
    simulateCode();
}

/* Functions */

void assembleFile()
{
    FILE *fptr;
    char line[255];
    int i;

    /* Open assemlby file */
    strcpy(line,inputFName);
    strcat(line,".asm");
    if ((fptr = fopen(line,"r")) == NULL) {
        fprintf (stderr,"Can't open assembly file %s.\n",line);
        exit(1);
    }

    /* Read and assemble */
    startTime = clock();
    numLines = 0; numLabels = 0; numOps = 0; patchList = NULL;
    patchCur = NULL;
    for (i=0; i<MAXOPS; i++) prog[i] = 0;
    while (fgets(line,255,fptr) != NULL) {
        numLines++;
        processLine(line);
    }
    fclose(fptr);

    patchLabels();
    displayStats();
}

void processLine(char *line)
{
    int i;
    char token[80];
    BYTE code;
```

```
    /* Ignore from # sign (comment) to end of line */
    for (i=0; line[i] != 0;) {
        if (line[i] == '#') line[i] = 0;
        else i++;
    }

    /* Make line uppercase */
    for (i=0; line[i] != 0; i++) {
        line[i] = toupper(line[i]);
    }

    /* Get token and process */
    if (sscanf (line,"%s",token) == EOF) token[0] = 0;
    if (token[0]) {
        if (isLabelDef(token)) addLabel(token,TRUE);
        else if ((code = isOp(token)) != BADOP) addOp(code,line);
        else {
            fprintf (stderr,"Error:  undefined token.\n    '%s' in line %d.\n",
                 token,numLines);
            exit(1);
        }
    }
}

BOOL isLabelDef(char *token)
{
    return (token[strlen(token)-1] == ':');
}

int addLabel(char *token, BOOL isDefinition)
{
    int i;
    int found = -1;
    char tmp[80];

    /* Check that nothing follows the label definition */
    if (sscanf(token,"%*s %s",tmp) != EOF) {
         fprintf (stderr,
              "Error: label definition not on line by itself:  line %d.\n",
              numLines);
        exit(1);
    }

    /* Strip : off of label definition */
    if (isDefinition)
        token[strlen(token)-1] = 0;

    /* See if label is already in library */
    for (i=0; i<numLabels; i++)
        if (strcmp(token,labelLib[i].name) == 0) {
            if (isDefinition && labelLib[i].value != UNDEFINED) {
                fprintf (stderr,
                   "Error:  label %s defined in both lines %d and %d.\n",
                   token,numLines,labelLib[i].value);
                exit(1);
            }
            else found = i;
```

```
        }

    /* If label was already used but not defined, define its value */
    if (found != -1) {
        if (isDefinition) labelLib[found].value = numOps;
        return found;
    }
    /* Otherwise add new label to library */
    else {
        if (numLabels >= MAXLABELS) {
            fprintf (stderr,"Error:  more than %d labels defined (line %d).\n",
                 MAXLABELS,numLines);
            exit(1);
        }
        strcpy(labelLib[numLabels].name,token);
        if (isDefinition) labelLib[numLabels].value = numOps;
        else labelLib[numLabels].value = UNDEFINED;
        return numLabels++;
    }
}

BYTE isOp(char *token)
{
    if (strcmp(token,"LDA") == 0) return _LDA;
    else if (strcmp(token,"LDI") == 0) return _LDI;
    else if (strcmp(token,"LDM") == 0) return _LDM;
    else if (strcmp(token,"STA") == 0) return _STA;
    else if (strcmp(token,"STI") == 0) return _STI;
    else if (strcmp(token,"ADD") == 0) return _ADD;
    else if (strcmp(token,"AND") == 0) return _AND;
    else if (strcmp(token,"NOT") == 0) return _NOT;
    else if (strcmp(token,"TST") == 0) return _TST;
    else if (strcmp(token,"SHR") == 0) return _SHR;
    else if (strcmp(token,"ROR") == 0) return _ROR;
    else if (strcmp(token,"PUT") == 0) return _PUT;
    else if (strcmp(token,"GET") == 0) return _GET;
    else if (strcmp(token,"JMP") == 0) return _JMP;
    else if (strcmp(token,"BRA") == 0) return _BRA;
    else if (strcmp(token,"SKZ") == 0) return _SKZ;
    else if (strcmp(token,"CAL") == 0) return _CAL;
    else if (strcmp(token,"RET") == 0) return _RET;
    else if (strcmp(token,"BRK") == 0) return _BRK;
    else return BADOP;
}

void addOp(BYTE code, char *line)
{
    switch (code) {
        /* opcodes that require no extra info */
        case _LDA:
        case _LDI:
        case _STA:
        case _STI:
        case _NOT:
        case _SHR:
        case _ROR:
        case _SKZ:
        case _RET:
```

```
        case _BRK:  addSimpleOp(code,line); break;

        /* opcodes requiring a register name */
        case _ADD:
        case _AND:
        case _TST:
        case _GET:
        case _PUT:  addRegOp(code,line); break;

        /* opcodes that require one subsequent byte */
        case _LDM:
        case _BRA:  addByteOp(code,line,1); break;

        /* opcodes that require two subsequent bytes */
        case _JMP:
        case _CAL:  addByteOp(code,line,2); break;
        default:    fprintf (stderr,
                        "Error:  Bad Operation Code %d in line %d.\n",
                        code,numLines);
                    exit(1);
    }
}

void addSimpleOp(BYTE code,char *line)
{
    char tmp[80];

    if (numOps >= MAXOPS) {
        fprintf (stderr,"Error:  Program too long: line %d.\n",numLines);
        exit(1);
    }
    if (sscanf(line,"%*s %s",tmp) != EOF) {
        fprintf (stderr,"Error: garbage follows operand:  line %d.\n",
             numLines);
        exit(1);
    }
    prog[numOps++] = code;
}

void addRegOp(BYTE code,char *line)
{
    char reg[80],tmp[80];
    BYTE regCode;

    if (numOps >= MAXOPS) {
        fprintf (stderr,"Error:  Program too long: line %d.\n",numLines);
        exit(1);
    }
    if (sscanf(line,"%*s %s %s",reg,tmp) != 1) {
        fprintf (stderr,"Error:  Register name required: line %d.\n",numLines);
        exit(1);
    }
    if (strcmp(reg,"R0") == 0) regCode = _R0;
    else if (strcmp(reg,"R1") == 0) regCode = _R1;
    else if (strcmp(reg,"R2") == 0) regCode = _R2;
    else if (strcmp(reg,"R3") == 0) regCode = _R3;
    else if (strcmp(reg,"MAL") == 0) regCode = _MAL;
    else if (strcmp(reg,"MAH") == 0) regCode = _MAH;
```

```
    else if (strcmp(reg,"PCL") == 0) regCode = _PCL;
    else if (strcmp(reg,"PCH") == 0) regCode = _PCH;
    else {
        fprintf (stderr,"Error:  Register name required: line %d.\n",numLines);
        exit(1);
    }
    prog[numOps++] = code | regCode<<5;
}

void addByteOp(BYTE code,char *line,int bytes)
{
    char next[80],tmp[80];

    if (numOps >= MAXOPS-bytes) {
        fprintf (stderr,"Error:  Program too long: line %d.\n",numLines);
        exit(1);
    }
    prog[numOps++] = code;
    if (sscanf(line,"%*s %s %s",next,tmp) != 1) {
        fprintf (stderr,"Error:  Byte or label required: line %d.\n",numLines);
        exit(1);
    }
    if (next[0] == '$') insertNum(next,bytes);
    else {
        if (code == _LDM) {
            fprintf (stderr,"Error:  LDM requires byte: line %d.\n",numLines);
            exit(1);
        }
        makeLabelPatch(next,bytes);
        numOps+= bytes;
    }
}

void insertNum(char *next,int bytes)
{
    unsigned long num;

    if (sscanf(next+1,"%lx",&num) != 1) {
        fprintf (stderr,"Error:  hexadecimal number expected: line %d.\n",
             numLines);
        exit(1);
    }
    if (bytes == 1) {
        if (num > 255) {
            fprintf (stderr,"Error:  number out of range ($00-$FF): line %d.\n"
                 ,numLines);
            exit(1);
        }
        prog[numOps++] = num;
    }
    else if (bytes == 2) {
        if (num > 65535) {
            fprintf (stderr,
                 "Error:  number out of range ($00-$FFFF): line %d.\n",
                 numLines);
            exit(1);
        }
        prog[numOps++] = num / 256;
```

```
        prog[numOps++] = num % 256;
    }
}

void makeLabelPatch(char *next, int bytes)
{
    int labelNum;
    patch *cur;

    labelNum = addLabel(next,FALSE);

    /* Maintain linked list of labels that must be patched */

    cur = (patch*)malloc(sizeof(patch));
    cur->labelNum = labelNum;
    cur->progAddr = numOps;
    cur->bytes = bytes;
    cur->next = NULL;

    if (patchList == NULL) {
        patchList = cur;
        patchCur = patchList;
    }
    else {
        patchCur->next = cur;
        patchCur = cur;
    }
}

void patchLabels(void)
{
    int value;
    patch *tmp;

    patchCur = patchList;
    while (patchCur != NULL) {
        value = labelLib[patchCur->labelNum].value;
        if (value == UNDEFINED) {
            fprintf (stderr,"Error:  label %s referenced but never defined.\n",
                            labelLib[patchCur->labelNum].name);
            exit(1);
        }
        if (patchCur->bytes == 1) { /* Compute offset */
            value = value-patchCur->progAddr;
            if (value < -128 || value > 127) {
                fprintf (stderr,
                   "Error:  offset to %s out of range (-128 to 127).\n",
                   labelLib[patchCur->labelNum].name);
                exit(1);
            }
            prog[patchCur->progAddr] = value;
        }
        else { /* two byte patch */
            prog[patchCur->progAddr] = value / 256;
            prog[patchCur->progAddr+1] = value % 256;
        }
        tmp = patchCur;
        patchCur = patchCur->next;
```

```
        free(tmp);
    }
}

void displayStats(void)
{
    int i;
    FILE *fptr;
    char fName[80];

    printf ("Assembly successful!\n");
    printf ("  Source:  %d lines, %d labels\n",numLines,numLabels);
    printf ("  Object:  $%X bytes\n",numOps);
/*    printf ("  Time elapsed:  %2.2f\n", (float)(clock()-startTime)/
          CLOCKS_PER_SEC); */
    /* CLOCKS_PER_SEC not defined on SPARC gcc at AI lab */

    strcpy(fName,inputFName);
    strcat(fName,".obj");
    if ((fptr = fopen(fName,"w")) == NULL) {
        fprintf (stderr,"Error:  Unable to open output file %s.\n",fName);
        exit(1);
    }
    for (i=0; i<numOps; i++) {
        fprintf (fptr,"/* $%.4X */ %.2X\n",i,prog[i]);
    }
    fclose(fptr);

    strcpy(fName,inputFName);
    strcat(fName,".lib");
    if ((fptr = fopen(fName,"w")) == NULL) {
        fprintf (stderr,"Error:  Unable to open output file %s.\n",fName);
        exit(1);
    }
    for (i=0; i<numLabels; i++) {
        fprintf (fptr,"%.4d: %20s $%.4X\n",i,labelLib[i].name,
             labelLib[i].value);
    }
    fclose(fptr);
}

void simulateCode(void)
{
    char cmd;

    breakPoint = UNDEFINED;
    resetSimulation();
    do {
        printf (
        "\n[G]o  [T]race  [S]tep  [M]emory  [B]reakpoint  [R]eset  [Q]uit: ");
        do {
            cmd = getchar();
        } while (cmd == '\r' || cmd == '\n');
        cmd = toupper(cmd);
        switch (cmd) {
            case 'G': runSimulation(FALSE); break;
            case 'T': runSimulation(TRUE); break;
            case 'S': doStep(); dumpRegisters(); break;
```

```
            case 'M': doMemory(); break;
            case 'B': doBreakpoint(); break;
            case 'R': resetSimulation(); break;
        }
    } while (cmd != 'Q');
}

void resetSimulation(void)
{
    int i;

    for (i=0; i<10; i++)
        registers[i] = 0;
    count = 0;
    printf ("Simulator reset.\n");
}

void dumpRegisters(void)
{
    printf ("A  : %.2X  R0 : %.2X  R1 : %.2X  R2 : %.2X  R3 : %.2X\n",
        registers[_ACC],registers[_R0],registers[_R1],registers[_R2],
          registers[_R3]);
    printf ("PCH: %.2X  PCL: %.2X  MAH: %.2X  MAL: %.2X  Count: %ld\n",
        registers[_PCH],registers[_PCL],registers[_MAH],registers[_MAL],count);
    printf ("Current instruction: ");
    disassemble(registers[_PCH]*256+registers[_PCL]);
}

int disassemble(int addr)
{
    BYTE op;
    int labelNum;

    printf ("$%.4X: ",addr);
    if ((labelNum = labelFromAddr(addr)) != UNDEFINED)
        printf ("%10s ",labelLib[labelNum].name);
    else printf ("           ");

    if (addr >= numOps) {
        printf ("*** End of Program\n");
        return 1;
    }

    op = prog[addr];

    switch (op & 0x1F) {
        case _LDA:
        case _LDI:
        case _STA:
        case _STI:
        case _NOT:
        case _SHR:
        case _ROR:
        case _SKZ:
        case _RET:
        case _BRK:  if (op == (op & 0x1F)) makeSimpleOp(op);
                    else makeNumber(op);
                    return 1;
```

```
        case _ADD:
        case _AND:
        case _TST:
        case _GET:
        case _PUT:  makeRegOp(op);
                    return 1;
        case _LDM:
        case _BRA:  if (op == (op & 0x1F)) {
                        makeByteOp(op,addr);
                        return 2;
                    }
                    else {
                        makeNumber(op);
                        return 1;
                    }
        case _JMP:
        case _CAL:  if (op == (op & 0x1F)) {
                        makeByteOp(op,addr);
                        return 3;
                    }
                    else {
                        makeNumber(op);
                        return 1;
                    }
        default: makeNumber(op); return 1;
    }
}

int labelFromAddr(int addr)
{
    int i;

    for (i=0; i<numLabels; i++)
        if (labelLib[i].value == addr) return i;
    return UNDEFINED;
}

void makeSimpleOp(BYTE code)
{
    switch(code) {
        case _LDA: printf ("LDA\n"); break;
        case _LDI: printf ("LDI\n"); break;
        case _STA: printf ("STA\n"); break;
        case _STI: printf ("STI\n"); break;
        case _NOT: printf ("NOT\n"); break;
        case _SHR: printf ("SHR\n"); break;
        case _ROR: printf ("ROR\n"); break;
        case _SKZ: printf ("SKZ\n"); break;
        case _RET: printf ("RET\n"); break;
        case _BRK: printf ("BRK\n"); break;
    }
}

void makeRegOp(BYTE code)
{
    switch(code & 0x1F) {
        case _ADD: printf ("ADD "); break;
        case _AND: printf ("AND "); break;
```

```
        case _TST: printf ("TST "); break;
        case _PUT: printf ("PUT "); break;
        case _GET: printf ("GET "); break;
    }
    switch(code>>5) {
        case _R0: printf ("R0\n"); break;
        case _R1: printf ("R1\n"); break;
        case _R2: printf ("R2\n"); break;
        case _R3: printf ("R3\n"); break;
        case _PCH: printf ("PCH\n"); break;
        case _PCL: printf ("PCL\n"); break;
        case _MAH: printf ("MAH\n"); break;
        case _MAL: printf ("MAL\n"); break;
    }
}

void makeByteOp(BYTE code,int addr)
{
    int labelNum;

    switch (code) {
        case _LDM: printf ("LDM $%.2X\n",prog[addr+1]); break;
        case _BRA: printf ("BRA $%.2X = $%.4X",prog[addr+1],
                     addr+1+prog[addr+1]-256*(prog[addr+1] > 127));
                   if ((labelNum = labelFromAddr
                  (addr+1+prog[addr+1]-256*(prog[addr+1] > 127))) !=
                   UNDEFINED)
                 printf (" = %s\n",labelLib[labelNum].name);
               else printf ("\n");
                   break;
        case _JMP: printf ("JMP $%.4X",prog[addr+1]*256+prog[addr+2]);
                   if ((labelNum = labelFromAddr(prog[addr+1]*256+
                                       prog[addr+2])) != UNDEFINED)
                        printf (" = %s\n",labelLib[labelNum].name);
                        else printf ("\n");
                   break;
        case _CAL: printf ("CAL $%.4X",prog[addr+1]*256+prog[addr+2]);
                   if ((labelNum = labelFromAddr(prog[addr+1]*256+
                                       prog[addr+2])) != UNDEFINED)
                        printf (" = %s\n",labelLib[labelNum].name);
                        else printf ("\n");
                   break;
    }
}

void makeNumber(BYTE code)
{
    printf ("$%.2X\n",code);
}

void runSimulation(BOOL traceFlag)
{
    breakFlag = FALSE;

    do {
        if (traceFlag) dumpRegisters();
        doStep();
    } while (breakFlag == FALSE);
```

```
    dumpRegisters();
}

void doStep(void)
{
    BOOL bumpPC;
    BYTE tmp,op;
    int offset;
    long addr;

    count++;
    bumpPC = TRUE;
    op = getMem(_PCH,0);
    switch (op & 0x1F) {
        case _LDA: registers[_ACC] = getMem(_MAH,0); break;
        case _LDI: registers[_ACC] = getMem(_MAH,0); increment(_MAH,1); break;
        case _LDM: registers[_ACC] = getMem(_PCH,1); increment(_PCH,1); break;
        case _STA: putMem(registers[_ACC]); break;
        case _STI: putMem(registers[_ACC]); increment(_MAH,1); break;
        case _ADD: registers[_ACC] += registers[op >> 5]; break;
        case _AND: registers[_ACC] = registers[_ACC] & registers[op >> 5];
        break;
        case _NOT: registers[_ACC] = ~registers[_ACC]; break;
        case _TST: registers[_ACC] = testConditions(registers[_ACC],
                                        registers[op >> 5]); break;
        case _SHR: registers[_ACC] = registers[_ACC]>>1; break;
        case _ROR: registers[_ACC] = (registers[_ACC]>>1) +
        128 * (registers[_ACC] % 2); break;
        case _PUT: registers[op >> 5] = registers[_ACC]; break;
        case _GET: registers[_ACC] = registers[op >> 5]; break;
        case _CAL: registers[_R0] = registers[_PCH] +
        ((int)registers[_PCL] +2) / 256;
                   registers[_R1] = ((int)registers[_PCL] + 2) % 256;
        case _JMP: tmp = getMem(_PCH,1); registers[_PCL] = getMem(_PCH,2);
                   registers[_PCH] = tmp; bumpPC = FALSE; break;
        case _BRA: addr = registers[_PCL] + 256 * registers[_PCH];
                   offset = getMem(_PCH,1);
                   if (offset > 127) offset -= 256;
                   addr += offset+1; bumpPC = FALSE;
                   registers[_PCL] = addr % 256; registers[_PCH] =
                 addr / 256; break;
        case _SKZ: if (registers[_ACC] == 0) increment(_PCH,2); break;
        case _RET: registers[_PCH] = registers[_R0];
                 registers[_PCL] = registers[_R1]; break;
        case _BRK: breakFlag = TRUE;
                   if (registers[_PCL] + 256 * registers[_PCH] == breakPoint) {
                        bumpPC = FALSE;
                        printf ("*** Hit User Breakpoint\n");
                   }
                   break;
        default:   printf ("*** Bad opcode $.2X encountered.\n",op);
        breakFlag = TRUE; break;
    }
    if (bumpPC) increment(_PCH,1);
}

BYTE getMem(BYTE reg,int offset)
{
```

```
    int addr;

    if (reg == _MAH) {
        addr = registers[_MAL] + 256 * registers[_MAH] + offset;
        if (addr > MAXOPS) {
            printf ("*** Tried to access illegal memory address %.4X\n",addr);
            breakFlag = TRUE;
            return 0;
        }
        return prog[addr];
    }
    else if (reg == _PCH) {
        addr = registers[_PCL] + 256 * registers[_PCH] +offset;
        if (addr >= numOps) {
            printf ("*** Tried to access past end of program address %.4X\n",
                addr);
            breakFlag = TRUE;
            return 0;
        }
        return prog[addr];
    }
    else {
        printf ("*** Illegal call to getMem in simulator.\n");
        breakFlag = TRUE;
    }
}

void putMem(BYTE what)
{
    int addr;

    addr = registers[_MAL] + 256 * registers[_MAH];
    if (addr > MAXOPS) {
        printf ("*** Tried to access illegal memory address %.4X\n",addr);
        breakFlag = TRUE;
    }
    prog[addr] = what;
}

void increment(BYTE reg,int amount)
{
    int addr;

    if (reg == _MAH) addr = registers[_MAL] + 256 * registers[_MAH];
    else if (reg == _PCH) addr = registers[_PCL] + 256*registers[_PCH];
    addr += amount;
    if (reg == _MAH) {
        registers[_MAL] = addr % 256;
        registers[_MAH] = addr / 256;
    }
    else if (reg == _PCH) {
        registers[_PCL] = addr % 256;
        registers[_PCH] = addr / 256;
    }
    else {
        printf ("*** Tried to increment an illegal register\n");
        breakFlag = TRUE;
    }
```

```
}

BYTE testConditions(BYTE acc,BYTE reg)
{
    BYTE tmp;
    int sum;
    int carry, zero;

    sum = acc + reg;
    carry = (int)acc + (int)reg > 255;
    zero = sum%256 == 0;
    /* Don't compute overflow */
    tmp = carry + (zero << 1);
    return tmp;
}

void doMemory(void)
{
    char cmd;

    do {
        printf ("\n[D]isassemble  [R]ead  [W]rite: ");
        do {
            cmd = getchar();
        } while (cmd == '\r' || cmd == '\n');
        cmd = toupper(cmd);
        switch (cmd) {
            case 'D': disassembleMemory(); break;
            case 'R': readMemory(); break;
            case 'W': writeMemory(); break;
        }
    } while (cmd != 'D' && cmd != 'R' && cmd != 'W');
}

void disassembleMemory(void)
{
    int start, bytes, count;

    printf ("Enter starting address and number of bytes in hex: ");
    scanf ("%x %x",&start,&bytes);
    if (start < 0 || start >= numOps)
      printf ("*** Starting address out of range.\n");
    else if (bytes < 0 || bytes > numOps-start)
      printf ("*** Too many bytes specified.\n");
    else {
        count = 0;
        while (count < bytes)
            count += disassemble(start+count);
    }
}

void readMemory(void)
{
    int start, bytes, count;

    printf ("Enter starting address and number of bytes in hex: ");
    scanf ("%x %x", &start, &bytes);
    if (start < 0 || start >= MAXOPS)
```

```
      printf ("*** Starting address out of range.\n");
    else if (bytes < 0 || bytes > MAXOPS-start)
      printf ("*** Too many bytes specified.\n");
    else {
        printf ("$%.4X: ",start);
        count = 0;
        while (count < bytes) {
            if ((start+count) % 8 == 0 && count > 0)
                printf ("\n$%.4X: ",start+count);
            printf ("%.2X ",prog[start+count++]);
        }
        printf ("\n");
    }
}

void writeMemory(void)
{
    int addr, data;

    printf ("Enter address in hex you wish to write: ");
    scanf ("%x",&addr);
    if (addr < 0 || addr >= MAXOPS)
      printf ("*** Address out of range (0-%X).\n",MAXOPS);
    else {
      printf ("Address $%.4X used to have $%.2X.  New value: ",
            addr,prog[addr]);
        scanf ("%x",&data);
        if (data < 0 || data >= MAXOPS)
        printf ("*** Data out of range (0-255).\n");
        else prog[addr] = data;
    }
}

void doBreakpoint(void)
{
    int addr;

    printf ("Enter address in hex to set breakpoint (-1 to remove): ");
    scanf ("%x",&addr);
    if (addr < -1 || addr > numOps)
      printf ("*** Address out of range (-1 to %X).\n",numOps);
    else {
        if (breakPoint != UNDEFINED) {
            prog[breakPoint] = oldBreakValue;
            printf ("Removed old breakpoint from $%4X.\n",breakPoint);
        }
        if (addr != UNDEFINED) {
            breakPoint = addr;
            oldBreakValue = prog[breakPoint];
            prog[breakPoint] = _BRK;
        }
    }
}
```

regress.asm

```
# MAYBE NOT regression test
# Written 1/2/94 by David Harris

LDM0:           # test the LDM instruction
    LDM $01     # load a 1
    SKZ
    BRA LDM1
    BRK
LDM1:
    LDM $00     # load a 0
    SKZ
    BRK
    BRK
LDA0:           # test the LDA instruction
    LDM $81     # Set up $81FE-$8100 with data $42 - $44
    PUT MAH
    LDM $FE
    PUT MAL
    LDM $42
    STA
    LDM $FF
    PUT MAL
    LDM $43
    STA
    LDM $00
    PUT MAL
    LDM $82
    PUT MAH
    LDM $44
    STA
    LDM $C3     # Try storing something in bank 3
    PUT MAH
    STA
    LDM $81     # Read back contents of $81FE
    PUT MAH
    LDM $FE
    PUT MAL
    LDA
    PUT R0      # And check if it is $42
    LDM $BE     # $BE = -$42
    TST R0
    PUT R0
    LDM $02     # Mask off Z bit
    AND R0
    SKZ
    BRA LDI0    # Read successfully
    BRK
LDI0:           # Test LDI instruction
    LDI         # Read $42 from $81FE
    LDI         # Read $43 from $81FF
    PUT R1      # And check that we didn't get 42 again
    LDM $BE
    TST R1
    PUT R1
    LDM $FA     # Mask off all bits except C and V
```

```
    AND R1
    SKZ
    BRK
    BRK
    LDI         # Read $44 from $8200
    PUT R2
    LDM $BC     # And check that we got it
    TST R2
    PUT R3
    LDM $02     # Mask off Z bisk
    AND R3
    SKZ
    BRA STA0    # Read the $44 successfully
    BRK
STA0:           # Check that STA works
    LDM $0  # Put a 0 in $8201
    STA
    LDM $AA # Then try reading it back
    LDA
    SKZ
    BRK
    BRK
STI0:       # Check that STI works
    LDM $01
    STI           # Store 1 in $8101
    STI             # And in $8102
    LDM $02     # Store 2 in $8103
    STI           # Now try reading back
    LDM $01
    PUT MAL
    LDI
    PUT R0      # 1 from $8101 should be in R0
    LDI
    PUT R1      # 1 from $8102 should be in R1
    AND R0
    SKZ
    BRA STI1
    BRK
STI1:
    LDI           # Load 2 from $8103
    AND R1
    SKZ
    BRK
    BRK
ADD0:
    LDM $1  # Load a 1
    PUT R2
    ADD R2  # 1 + 1 = 2
    SKZ
    BRA ADD1
ADD1:
    PUT R2
    ADD R2  # 2 + 2 = 4
    PUT R3
    ADD R3  # 4 + 4 = 8
    PUT R1
    ADD R1  # 8 + 8 = 10
    PUT R0
```

```
    ADD R0   # 10 + 10 = 20
    PUT R0
    ADD R0   # 20 + 20 = 40
    PUT R1
    LDM $C0 # C0 = -40
    TST R1
    PUT R2
    LDM $02
    AND R2
    SKZ
    BRA ADD2
    BRK
ADD2:
    LDM $0
    ADD R1   # 0 + 40 = 40
    PUT R0
    ADD R0   # 40 + 40 = 80
    PUT R0
    ADD R0   # 80 + 80 = 0
    SKZ
    BRK
    BRK
AND0:        # Test AND instruction
    LDM $FF      # Calculate $FF & $FF
    PUT R1
    AND R1
    PUT R2
    LDM $1
    ADD R2
    SKZ
    BRK
    BRK
    LDM $AA        # Calculate $AA & $55
    PUT R3
    LDM $55
    AND R3
    SKZ
    BRK
    BRK
NOT0:        # Test NOT instruction
    LDM $FF # Calculate ~$FF
    NOT
    SKZ
    BRK
    BRK
    LDM $32 # Calculate ~$32
    NOT
    SKZ
    BRA SHR0
    BRK
SHR0:        # Test SHR instruction
    LDM $20      # Calculate 20 >> 1
    SHR
    PUT R2
    ADD R2
    PUT R3
    LDM $E0 # E0 = -20
    TST R3       # Should give test code $03
```

```
    PUT R0
    LDM $FD
    ADD R0
    SKZ
    BRK
    BRK
    LDM $3
    SHR
    SKZ
    BRA SHR1
    BRK
SHR1:
    SHR
    SKZ
    BRK
    BRK
ROR0:         # Test ROR instruction
    LDM $1        # $1 ROR = $80
    ROR
    SKZ             # Make sure we don't get 0 instead
    BRA ROR1
    BRK
ROR1:
    PUT R1
    ADD R1
    SKZ
    BRK
    BRK
PUT0:         # Test PUT instruction
    LDM $42 # Add 42 to two's complement of 42
    PUT R0
    NOT
    PUT R1
    LDM $1
    ADD R1
    ADD R0
    SKZ
    BRK
    BRK
GET0:
    LDM $73       # Test GET instruction
    PUT R2
    LDM $42
    PUT R0
    GET R2        # Reload the $73
    NOT           # Compute -$73
    PUT R3
    LDM $01
    ADD R3
    ADD R2        # Check -$73 + $73 = 0
    SKZ
    BRK
    BRK
JMP0:         # Test JMP instruction
    JMP JMP2      # Forward jump
    BRK
JMP1:
    JMP BRA0
```

```
    BRK
JMP2:
    JMP JMP1      # Backward jump
    BRK
BRA0:        # Test BRA instruction
    BRA BRA2    # Forward branch
    BRK
BRA1:
    BRA SKZ0
    BRK
BRA2:
    BRA BRA1      # Backward branch
    BRK
SKZ0:        # Test SKZ instruction
    LDM $0   # Skip
    SKZ
    BRK
    BRK
    LDM $1
    SKZ
    BRA TST0
    BRK
TST0:
    LDM $0      # Test Z flag
    PUT R3
    TST R3
    PUT R2
    LDM $FE
    ADD R2
    SKZ
    BRK
    BRK
    LDM $80     # Test Z and C flags together
    PUT R3
    TST R3
    PUT R2
    LDM $FD
    ADD R2
    SKZ
    BRK
    BRK
    LDM $90     # Test C flag alone
    TST R3
    PUT R1
    LDM $FF
    ADD R1
    SKZ
    BRK
    BRK
    GET R1      # Test no flags
    TST R1
    SKZ
    BRK
    BRK
CALL0:
    CAL CALL1   # Test CAL instruction
    PUT R1
    LDM $D9     # Check return from right place
```

```
    ADD R1
    SKZ
    BRK
    BRK
    BRA DONE
CALL2:
    GET R3      # Return to CALL0, not CALL1
    PUT R1
    GET R2
    PUT R0
    LDM $27     # Note that this happened
    RET
CALL1:
    GET R0      # Save return address in R2:R3
    PUT R2
    GET R1
    PUT R3
    CAL CALL2   # Call another place
DONE:
    LDM $FF
    PUT R0
    PUT R1
    PUT R2
    PUT R3
    BRK
```

plagen.il

```
/* plagen2.il */

/* Written 1/8/94 by David Harris

  This program automatically generates PLAs from
  sum of products form.

  Test with command
  (procedure (run) (load "/home/cva2/6090user/tools/pla/plagen.il")
           (plagen "/home/cva2/6090user/tools/pla/test.pla" "unintel" "pla"))

  Sorry this code is so messy; it's been a while since I've hacked LISP
  and I know my string parsing is suboptimal.
*/

MAXINPUTS = 100
MAXPRODUCTS = 100
MAXOUTPUTS = 100

(defstruct intype name uninverted inverted)

/* Key data structures

   This program creates three main data structures, each arrays.
   outputs[i]: name of ith output
   products[i]: list of product name, (list of outputs), (list of inputs & inv)
   inputs[i]: name of ith input and need for inverted/uninverted
      structure of type intype
*/

(procedure (initeqns)
         /* Initialize data structures */
         (declare inputs[MAXINPUTS])
         (declare products[MAXPRODUCTS])
         (declare outputs[MAXOUTPUTS])
         numinputs = 0
         numoutputs = 0
         numproducts = 0
         numlines = 0
         )

(procedure (goodoutput name)
         (for i 0 numoutputs-1
            (if (strcmp name outputs[i]) == 0
                (error "Output %s redefined: line %d"
                     name numlines)
                )
            )
         )

(procedure (addoutput out)
         (let ((str (instring out)))
           (fscanf str "%s" name)
           (if (goodoutput name) then
             outputs[numoutputs] = name
```

```
            numoutputs = numoutputs + 1
            )
          )
        )

(procedure (goodproduct name)
        (for i 0 numproducts-1
           (if (strcmp name (car products[i])) == 0
               (error "Product %s redefined: line %d"
                    name numlines)
               )
           )
        )

(procedure (addproductname out terms)
        (let ((str (instring out))
            (literals (parseString (car terms) " *\n")))
          (fscanf str "%s" name)
          (if (goodproduct name) then
            products[numproducts] = (list name nil
                                      (mapcar 'markinversion
                                            literals))
            (foreach inp (caddr products[numproducts])
                 (addinput inp))
            numproducts = numproducts + 1
            )
          )
        )

(procedure (markinversion name)
        (let ((len (strlen name)))
          (if (getchar name len) == (getchar "'" 1) then /* gross hack */
            (list (substring name 1 len-1) 'inverted)
            else
            (list name 'uninverted)
            )
          )
        )

(procedure (addinput inp)
        found = nil
        (for i 0 numinputs-1
           (if (strcmp (car inp) inputs[i]->name) == 0 then
               found = 't
               (if (cadr inp) == 'uninverted
                 inputs[i]->inverted = 't
                 inputs[i]->uninverted = 't
                 )
               )
           )
        (if (found == nil) then
          (if (eq (cadr inp) 'uninverted)
            inputs[numinputs] = (make_intype
                                  ?name (car inp) ?uninverted 't)
            inputs[numinputs] = (make_intype
                                ?name (car inp) ?inverted 't)
            )
         numinputs = numinputs + 1
```

```
        )
      )

(procedure (findproductbyname term)
      found = -1
      (for i 0 numproducts-1
         (if (equal term (car products[i]))
             found = i)
         )
      )

(procedure (addterm term)
      ; strip spaces to get name
      name = (buildString (parseString term " \n") "")
      (findproductbyname name)
      (if (found != -1) then
        products[found] = (list (car products[found])
                            (cons numoutputs-1 (cadr products[found]))
                            (caddr products[found]))
        else (reallyaddterm term))
      )

(procedure (reallyaddterm term)
      (let ((literals (parseString term " *\n")))
        products[numproducts] = (list
                             (sprintf tmpstr "prod%d" numproducts)
                             (list numoutputs-1)
                             (mapcar 'markinversion literals))
        (foreach inp (caddr products[numproducts])
               (addinput inp))
        numproducts = numproducts+1
        )
      )

(procedure (defineinputs inp)
      found = nil
      (for i 0 numinputs-1
         (if (strcmp inp inputs[i]->name) == 0 then
             found = 't
             )
         )
      (if (found == nil) then
        inputs[numinputs] = (make_intype ?name inp)
        numinputs = numinputs + 1
        )
      )

(procedure (addeqn line)
      (let ((out (car (parseString line "=")))
          (productdef (car (parseString line ":")))
          (firstword (car (parseString line " ")))
          (terms (cdr (parseString line "=+:"))))
        (if (firstword == "inputorder") then
          (foreach inp (cdr (parseString line " \n"))
               (defineinputs inp))
         else
          (if (null (index line ":")) then
          (addoutput out)
```

```
            (foreach term terms
                  (addterm term))
            else
            (addproductname productdef terms)
            )
            )
          )
        )

(procedure (rounduptoeven)
        (if (mod numproducts 2) == 1 then
            products[numproducts] = (list "unused" nil nil)
            numproducts = numproducts + 1
            )
        (if (mod numoutputs 2) == 1 then
            outputs[numoutputs] = "null"
            numoutputs = numoutputs+1
            )
        )

(procedure (addbar name)
        (let ((where (nindex name "<")))
          (if where then
            (strcat (substring name 1 where-1) "bar" (index name "<"))
            else (strcat name "bar")))
        )

(procedure (drawinputlines inp)
        (dbCreateRect pla "poly" (list (list i*colwidth1+5 0)
                                (list i*colwidth1+7 height-6)))
        (dbCreateLabel pla "labels" (list i*colwidth1+6 -11)
                  inputs[i]->name "centerCenter" "R90" "stick" 2)
        (dbCreateRect pla "poly" (list (list i*colwidth1+13 0)
                                (list i*colwidth1+15 height-6)))
        (dbCreateLabel pla "labels" (list i*colwidth1+14 -11)
                  (addbar inputs[i]->name) "centerCenter"
                  "R90" "stick" 2)
        (dbCreateRect pla "ndiff" (list (list i*colwidth1+8 2)
                                (list i*colwidth1+12 height-8)))
        (dbCreateRect pla "nselect" (list (list i*colwidth1+5 -1)
                                   (list i*colwidth1+15 height-5)))
)

(procedure (drawandlines prod)
        (dbCreateRect pla "metal1"
                  (list (list 0 prod*rowheight*3+2)
                        (list width1 prod*rowheight*3+6)))
        (if prod > 0
            (dbCreateLabel pla "labels" (list -6 prod*rowheight*3+4)
                      (car products[prod*2])
                      "centerCenter" "R0" "stick" 2)
            )
        (dbCreateRect pla "metal1"
                  (list (list 0 prod*rowheight*3+2+rowheight)
                        (list width1 prod*rowheight*3+6+rowheight)))
        (dbCreateRect pla "nselect"
                  (list (list -2 prod*rowheight*3-1)
                        (list width1 prod*rowheight*3+9)))
```

```
        (dbCreateRect pla "metal1"
                  (list (list 0 prod*rowheight*3+2+rowheight*2)
                        (list width1 prod*rowheight*3+6+rowheight*2)))
        (dbCreateRect pla "nselect"
                  (list (list -2 prod*rowheight*3-1+rowheight*2)
                        (list width1 prod*rowheight*3+9+rowheight*2)))
        (dbCreateLabel pla "labels" (list -6
prod*rowheight*3+4+rowheight*2)
                   (car products[prod*2+1])
                   "centerCenter" "R0" "stick" 2)
        (for i 0 numinputs-1
           (dbCreateRect pla "psub"
                      (list (list i*colwidth1+16
                              prod*rowheight*3+rowheight+2)
                          (list i*colwidth1+20
                              prod*rowheight*3+rowheight+6)))
           (dbCreateRect pla "pselect"
                      (list (list i*colwidth1+15
                              prod*rowheight*3+rowheight+1)
                          (list i*colwidth1+21
                              prod*rowheight*3+rowheight+7)))
           (dbCreateRect pla "cont_aa"
                      (list (list i*colwidth1+17
                              prod*rowheight*3+rowheight+3)
                          (list i*colwidth1+19
                              prod*rowheight*3+rowheight+5)))
           (dbCreateRect pla "cont_aa"
                      (list (list i*colwidth1+9
                              prod*rowheight*3+rowheight+3)
                          (list i*colwidth1+11
                              prod*rowheight*3+rowheight+5)))
           )
        )

(procedure (draworlines prod)
        (dbCreateRect pla "poly"
                  (list (list width1-4 prod*rowheight*3+2)
                        (list width1 prod*rowheight*3+6)))
        (dbCreateRect pla "cont"
                  (list (list width1-3 prod*rowheight*3+3)
                        (list width1-1 prod*rowheight*3+5)))
        (dbCreateRect pla "poly"
                  (list (list width1 prod*rowheight*3+3)
                        (list width2 prod*rowheight*3+5)))
        (dbCreateRect pla "nselect"
                  (list (list width1 prod*rowheight*3+4)
                        (list width2 prod*rowheight*3-4)))
        (dbCreateRect pla "poly"
                  (list (list width1-4 prod*rowheight*3+2+rowheight*2)
                        (list width1 prod*rowheight*3+6+rowheight*2)))
        (dbCreateRect pla "cont"
                  (list (list width1-3 prod*rowheight*3+3+rowheight*2)
                        (list width1-1 prod*rowheight*3+5+rowheight*2)))
        (dbCreateRect pla "poly"
                  (list (list width1 prod*rowheight*3+3+rowheight*2)
                        (list width2 prod*rowheight*3+5+rowheight*2)))
        (dbCreateRect pla "nselect"
                  (list (list width1 prod*rowheight*3+4+rowheight*2)
```

```
                        (list width2 prod*rowheight*3+12+rowheight*2)))
        )

(procedure (drawpullups1 prod)
        vddpos = -23
        (dbCreateRect pla "metal1"
                (list (list vddpos+6 prod*rowheight*3+10)
                      (list vddpos+12 prod*rowheight*3+14)))
        (dbCreateRect pla "pselect"
                (list (list vddpos+6 prod*rowheight*3)
                      (list vddpos+15 (prod+1)*rowheight*3)))
        (dbCreateRect pla "pdiff"
                (list (list vddpos+9 prod*rowheight*3+2)
                      (list vddpos+12 (prod+1)*rowheight*3-2)))
        (dbCreateRect pla "pdiff"
                (list (list vddpos+12 prod*rowheight*3+2)
                      (list vddpos+13 prod*rowheight*3+6)))
        (dbCreateRect pla "pdiff"
                (list (list vddpos+8 prod*rowheight*3+2+rowheight)
                      (list vddpos+9 prod*rowheight*3+6+rowheight)))
        (dbCreateRect pla "pdiff"
                (list (list vddpos+12
                         prod*rowheight*3+2+2*rowheight)
                      (list vddpos+13
                         prod*rowheight*3+6+2*rowheight)))
        (dbCreateRect pla "cont_aa"
                (list (list vddpos+10 prod*rowheight*3+3)
                      (list vddpos+12 prod*rowheight*3+5)))
        (dbCreateRect pla "cont_aa"
                (list (list vddpos+9
                         prod*rowheight*3+3+rowheight)
                      (list vddpos+11
                         prod*rowheight*3+5+rowheight)))
        (dbCreateRect pla "cont_aa"
                (list (list vddpos+10
                         prod*rowheight*3+3+2*rowheight)
                      (list vddpos+12
                         prod*rowheight*3+5+2*rowheight)))
        (dbCreateRect pla "poly"
                (list (list vddpos+15 prod*rowheight*3+7)
                      (list vddpos+19 prod*rowheight*3+17)))
        (dbCreateRect pla "poly"
                (list (list vddpos+7 prod*rowheight*3+7)
                      (list vddpos+16 prod*rowheight*3+9)))
        (dbCreateRect pla "poly"
                (list (list vddpos+7 prod*rowheight*3+15)
                      (list vddpos+16 prod*rowheight*3+17)))
        (dbCreateRect pla "metal1"
                (list (list vddpos+9 prod*rowheight*3+2)
                      (list 0 prod*rowheight*3+6)))
        (dbCreateRect pla "metal1"
                (list (list vddpos+9 prod*rowheight*3+2+2*rowheight)
                      (list 0 prod*rowheight*3+6+2*rowheight)))
        (dbCreateRect pla "metal1"
                (list (list vddpos+15 prod*rowheight*3+2+rowheight)
                      (list 0 prod*rowheight*3+6+rowheight)))
        (dbCreateRect pla "cont"
                (list (list vddpos+16 prod*rowheight*3+3+rowheight)
```

```
                        (list vddpos+18 prod*rowheight*3+5+rowheight)))
        (dbCreateRect pla "nsub"
                (list (list vddpos prod*rowheight*3+2+rowheight)
                        (list vddpos+4 prod*rowheight*3+6+rowheight)))
        (dbCreateRect pla "nselect"
                (list (list vddpos-2 prod*rowheight*3+rowheight)
                        (list vddpos+6 prod*rowheight*3+8+rowheight)))
        (dbCreateRect pla "cont_aa"
                (list (list vddpos+1 prod*rowheight*3+3+rowheight)
                        (list vddpos+3 prod*rowheight*3+5+rowheight)))
        )

(procedure (inputinprod inp prod)
        (exists p (caddr prod) (inp == (car p)))
        )

(procedure (isinverted inp prod)
        (exists p (caddr prod)
             (and (inp == (car p)) ((cadr p) == 'inverted)))
        )

(procedure (drawtransistor1 inp prod)
        (if (inputinprod inputs[inp]->name products[prod]) then
          xpos = colwidth1*inp
          inv = nil
          (if !(isinverted inputs[inp]->name products[prod]) then
            xpos = xpos+8
            inv = 't
            )
          bot = (prod/2)*3*rowheight + (mod prod 2)*2*rowheight + 2
          (dbCreateRect pla "ndiff"
                  (list (list xpos bot) (list xpos+12 bot+4)))
          (if (inv == 't) then
            (dbCreateRect pla "cont_aa"
                        (list (list xpos+9 bot+1) (list xpos+11 bot+3)))
            else
            (dbCreateRect pla "cont_aa"
                      (list (list xpos+1 bot+1) (list xpos+3 bot+3)))
            )
          )
        )
(procedure (drawinverters1 in)
        left = in*colwidth1
        (dbCreateRect pla "poly"
                (list (list left+5 0) (list left+7 -38)))
        (dbCreateRect pla "poly"
                (list (list left+7 -34) (list left+12 -38)))
        (dbCreateRect pla "poly"
                (list (list left+13 0) (list left+15 -19)))
        (dbCreateRect pla "poly"
                (list (list left+10 -19) (list left+13 -15)))
        (dbCreateRect pla "ndiff"
                (list (list left -12) (list left+12 -8)))
        (dbCreateRect pla "nselect"
                (list (list left-2 -15) (list left+14 -5)))
        (dbCreateRect pla "pdiff"
                (list (list left -31) (list left+12 -27)))
        (dbCreateRect pla "pselect"
```

```
                (list (list left-2 -34) (list left+14 -24)))
        (dbCreateRect pla "metal1"
                (list (list left -12) (list left+4 -3)))
        (dbCreateRect pla "metal1"
                (list (list left -40) (list left+4 -22)))
        (dbCreateRect pla "metal1"
                (list (list left+8 -34) (list left+12 -40)))
        (dbCreateRect pla "metal1"
                (list (list left+8 -8) (list left+12 -31)))
        (dbCreateRect pla "metal1"
                (list (list left+12 -15) (list left+14 -19)))
        (dbCreateRect pla "nsub"
                (list (list left -36) (list left+4 -40)))
        (dbCreateRect pla "nselect"
                (list (list left-2 -34) (list left+6 -41)))
        (dbCreateRect pla "via"
                (list (list left+1 -4) (list left+3 -6)))
        (dbCreateRect pla "via"
                (list (list left+1 -23) (list left+3 -25)))
        (dbCreateRect pla "cont_aa"
                (list (list left+1 -9) (list left+3 -11)))
        (dbCreateRect pla "cont_aa"
                (list (list left+9 -9) (list left+11 -11)))
        (dbCreateRect pla "cont_aa"
                (list (list left+1 -28) (list left+3 -30)))
        (dbCreateRect pla "cont_aa"
                (list (list left+9 -28) (list left+11 -30)))
        (dbCreateRect pla "cont_aa"
                (list (list left+1 -37) (list left+3 -39)))
        (dbCreateRect pla "cont"
                (list (list left+11 -18) (list left+13 -16)))
        (dbCreateRect pla "cont"
                (list (list left+9 -37) (list left+11 -35)))
        (dbCreateLabel pla "labels" (list left+10 -39)
                 inputs[i]->name "centerCenter" "R90" "stick" 2)
        (leCreatePin pla (list "metal1" "drawing") "rectangle"
               (list (list left+8 -38) (list left+12 -42))
               inputs[i]->name "input" (list "bottom"))
        )

(procedure (drawoutputlines out)
        xpos = width1+out*pairwidth2+4
        (dbCreateRect pla "metal1"
                (list (list xpos -2) (list xpos+4 height)))
        (dbCreateRect pla "nselect"
                (list (list xpos-3 -2) (list xpos+7 height)))
        xpos = xpos + 8
        (dbCreateRect pla "metal1"
                (list (list xpos -2 ) (list xpos+4 height)))
        xpos = xpos + 8
        (dbCreateRect pla "metal1"
                (list (list xpos -2) (list xpos+4 height)))
        (dbCreateRect pla "nselect"
                (list (list xpos-3 -2) (list xpos+7 height)))
        )

(procedure (drawpullups2 out)
        top = height+19
```

```
        left = out*pairwidth2 + width1 + 2
        (dbCreateRect pla "metal1"
                (list (list left+10 top-12) (list left+14 top-6)))
        (dbCreateRect pla "pselect"
                (list (list left top-6) (list left+24 top-15)))
        (dbCreateRect pla "pdiff"
                (list (list left+2 top-9) (list left+22 top-12)))
        (dbCreateRect pla "pdiff"
                (list (list left+2 top-12) (list left+6 top-13)))
        (dbCreateRect pla "pdiff"
                (list (list left+18 top-12) (list left+22 top-13)))
        (dbCreateRect pla "pdiff"
                (list (list left+10 top-8) (list left+14 top-9)))
        (dbCreateRect pla "cont_aa"
                (list (list left+3 top-10) (list left+5 top-12)))
        (dbCreateRect pla "cont_aa"
                (list (list left+19 top-10) (list left+21 top-12)))
        (dbCreateRect pla "cont_aa"
                (list (list left+11 top-9) (list left+13 top-11)))
        (dbCreateRect pla "poly"
                (list (list left+7 top-15) (list left+17 top-19)))
        (dbCreateRect pla "poly"
                (list (list left+7 top-7) (list left+9 top-15)))
        (dbCreateRect pla "poly"
                (list (list left+15 top-7) (list left+17 top-15)))
        (dbCreateRect pla "metal1"
                (list (list left+2 top-9) (list left+6 height)))
        (dbCreateRect pla "metal1"
                (list (list left+18 top-9) (list left+22 height)))
        (dbCreateRect pla "metal1"
                (list (list left+10 top-15) (list left+14 height)))
        (dbCreateRect pla "cont"
                (list (list left+11 top-16) (list left+13 top-18)))
        (dbCreateRect pla "nsub"
                (list (list left+10 top) (list left+14 top-4)))
        (dbCreateRect pla "nselect"
                (list (list left+8 top+2) (list left+16 top-6)))
        (dbCreateRect pla "cont_aa"
                (list (list left+11 top-1) (list left+13 top-3)))
        )

(procedure (drawsubcontact out prod)
        left = out*pairwidth2+12+width1
        bot = prod*rowheight*3+10
        (dbCreateRect pla "psub"
                (list (list left bot) (list left+4 bot+4)))
        (dbCreateRect pla "pselect"
                (list (list left-1 bot-1) (list left+5 bot+5)))
        (dbCreateRect pla "cont_aa"
                (list (list left+1 bot+1) (list left+3 bot+3)))
        )

(procedure (drawinverters2 out)
        left = width1+out*pairwidth2+4
        /* Make left inverter */
        (dbCreateRect pla "metal1"
                (list (list left -2) (list left+4 -8)))
        (dbCreateRect pla "poly"
```

```
          (list (list left -4) (list left+7 -8)))
(dbCreateRect pla "poly"
          (list (list left+5 -8) (list left+7 -31)))
(dbCreateRect pla "cont"
          (list (list left+1 -5) (list left+3 -7)))
(dbCreateRect pla "metal1"
          (list (list left -11) (list left+4 -40)))
(dbCreateRect pla "cont_aa"
          (list (list left+1 -12) (list left+3 -14)))
(dbCreateRect pla "cont_aa"
          (list (list left+1 -26) (list left+3 -28)))
(dbCreateLabel pla "labels" (list left+2 -38)
           outputs[out*2] "centerCenter" "R90" "stick" 2)
(leCreatePin pla (list "metal1" "drawing") "rectangle"
         (list (list left -38) (list left+4 -42))
         outputs[out*2] "output" (list "bottom"))
/* Make middle stuff */
(dbCreateRect pla "ndiff"
          (list (list left -11) (list left+20 -15)))
(dbCreateRect pla "nselect"
          (list (list left-2 -8) (list left+22 -18)))
(dbCreateRect pla "pdiff"
          (list (list left -25) (list left+20 -29)))
(dbCreateRect pla "pselect"
          (list (list left-2 -22) (list left+22 -32)))
(dbCreateRect pla "metal1"
          (list (list left+8 -2) (list left+12 -15)))
(dbCreateRect pla "metal1"
          (list (list left+8 -19) (list left+12 -40)))
(dbCreateRect pla "cont_aa"
          (list (list left+9 -12) (list left+11 -14)))
(dbCreateRect pla "cont_aa"
          (list (list left+9 -26) (list left+11 -28)))
(dbCreateRect pla "via"
          (list (list left+9 -6) (list left+11 -8)))
(dbCreateRect pla "via"
          (list (list left+9 -20) (list left+11 -22)))
(dbCreateRect pla "nsub"
          (list (list left+8 -34) (list left+12 -38)))
(dbCreateRect pla "nselect"
          (list (list left+6 -32) (list left+14 -39)))
(dbCreateRect pla "cont_aa"
          (list (list left+9 -35) (list left+11 -37)))
/* Make left inverter */
(dbCreateRect pla "metal1"
          (list (list left+20 -2) (list left+16 -8)))
(dbCreateRect pla "poly"
          (list (list left+20 -4) (list left+13 -8)))
(dbCreateRect pla "poly"
          (list (list left+15 -8) (list left+13 -31)))
(dbCreateRect pla "cont"
          (list (list left+19 -5) (list left+17 -7)))
(dbCreateRect pla "metal1"
          (list (list left+20 -11) (list left+16 -40)))
(dbCreateRect pla "cont_aa"
          (list (list left+19 -12) (list left+17 -14)))
(dbCreateRect pla "cont_aa"
          (list (list left+19 -26) (list left+17 -28)))
```

```
        (dbCreateLabel pla "labels" (list left+18 -38)
                  outputs[out*2+1] "centerCenter" "R90" "stick" 2)
        (leCreatePin pla (list "metal1" "drawing") "rectangle"
                (list (list left+16 -38) (list left+20 -42))
                outputs[out*2+1] "output" (list "bottom"))
        )

(procedure (outputinprod out prod)
        (exists x (cadr prod) (x == out))
        )

(procedure (drawtransistor2 out prod)
        (if (outputinprod out products[prod]) then
          left = (out/2)*pairwidth2 + (mod out 2)*8 + 4 + width1
          right = left+12
          bot = (prod/2)*3*rowheight + (mod prod 2)*(rowheight+4) - 2
          top = bot + 16
          (if (mod prod 2) == 1 then
            (dbCreateRect pla "ndiff"
                  (list (list left top) (list right top-4)))
            (if (mod out 2) == 0 then
                (dbCreateRect pla "cont_aa"
                        (list (list right-3 top-1)
                            (list right-1 top-3)))
                (dbCreateRect pla "ndiff"
                        (list (list left bot+4) (list left+4 top-4)))
                (dbCreateRect pla "cont_aa"
                        (list (list left+1 bot+5)
                            (list left+3 bot+7)))
                else
                (dbCreateRect pla "cont_aa"
                        (list (list left+1 top-1)
                            (list left+3 top-3)))
                (dbCreateRect pla "ndiff"
                        (list (list right-4 bot+4)
                            (list right top-4)))
                (dbCreateRect pla "cont_aa"
                        (list (list right-3 bot+5)
                            (list right-1 bot+7)))
                )
            else
            (dbCreateRect pla "ndiff"
                  (list (list left bot) (list right bot+4)))
            (if (mod out 2) == 0 then
                (dbCreateRect pla "cont_aa"
                        (list (list right-3 bot+1)
                            (list right-1 bot+3)))
                (dbCreateRect pla "ndiff"
                        (list (list left bot) (list left+4 top-4)))
                (dbCreateRect pla "cont_aa"
                        (list (list left+1 top-5)
                            (list left+3 top-7)))
                else
                (dbCreateRect pla "cont_aa"
                        (list (list left+1 bot+1)
                            (list left+3 bot+3)))
                (dbCreateRect pla "ndiff"
                        (list (list right-4 bot+4)
```

```
                              (list right top-4)))
                (dbCreateRect pla "cont_aa"
                         (list (list right-3 top-5)
                              (list right-1 top-7)))
                )
            )
          )
        )

(procedure (makenwells)
        (dbCreateRect pla "nwell" (list (list vddpos-3 -3)
                                (list -5 height+3)))
        (dbCreateRect pla "nwell" (list (list width1-1 height+22)
                                (list width2+3 height+1)))
        (dbCreateRect pla "nwell" (list (list vddpos-3 -43)
                                (list width2+3 -20)))
;       leHiLayerGen()
        )

(procedure (linkvddgnd)
        /* Place vdd! */
        (dbCreateRect pla "metal1"
                  (list (list vddpos-14 height+13)
                        (list width2 height+33)))
        (dbCreateRect pla "metal1"
                  (list (list vddpos-14 -37)
                        (list vddpos+6 height+13)))
        (dbCreateRect pla "metal1"
                  (list (list vddpos+6 -37)
                        (list vddpos+16 -13)))
        (dbCreateRect pla "metal2" (list (list vddpos-3 -13)
                                 (list width2 -37)))
        (for i 0 3
           (for j 0 4
                (dbCreateRect pla "via"
                         (list (list vddpos-2+5*i -36+5*j)
                              (list vddpos+5*i -34+5*j)))
                )
           )
        (leCreatePin pla (list "metal2" "drawing") "rectangle"
                 (list (list vddpos-5 -13) (list vddpos-1 -37))
                 "vdd!" "input" (list "left"))
        (leCreatePin pla (list "metal2" "drawing") "rectangle"
                 (list (list width2-2 -13) (list width2+2 -37))
                 "vdd!" "input" (list "right"))
        (dbCreateLabel pla "labels" (list -11 -26)
                   "vdd!" "centerCenter" "R0" "stick" 4)
        /* Place gnd! */
        (dbCreateRect pla "metal2" (list (list vddpos-3 15)
                                 (list width2 -9)))
        (dbCreateRect pla "metal2" (list (list width1-14 15)
                                 (list width1+10 height-16)))
        (for i 0 (numproducts-1)/2
           bot = i*3*rowheight + rowheight + 3
           (dbCreateRect pla "via" (list (list width1-3 bot)
                                  (list width1-1 bot+2)))
           )
        (leCreatePin pla (list "metal2" "drawing") "rectangle"
```

```
                (list (list vddpos-5 15) (list vddpos-1 -9))
                "gnd!" "input" (list "left"))
        (leCreatePin pla (list "metal2" "drawing") "rectangle"
                (list (list width2-2 15) (list width2+2 -9))
                "gnd!" "input" (list "right"))
        (dbCreateLabel pla "labels" (list -11 -4)
                  "gnd!" "centerCenter" "R0" "stick" 4)
        )

(procedure (drawpla pla)
        productlines = numproducts/2 * 3
        outpairs = numoutputs/2
        colwidth1 = 8*2
        pairwidth2 = 24
        rowheight = 8
        height = productlines * rowheight+6
        width1 = numinputs * colwidth1 + 10
        width2 = width1 + outpairs * pairwidth2 +2
        /* Create vertical lines for each input */
        (for i 0 numinputs-1

           (drawinputlines i)
           (drawinverters1 i)
           )
        /* Create horizontal lines for each product term */
        (for i 0 (numproducts-1)/2
           (drawandlines i)
           (draworlines i)
           (drawpullups1 i)
           )
        /* Create vertical lines for each output */
        (for i 0 (numoutputs-1)/2
           (drawoutputlines i)
           (drawpullups2 i)
           (drawinverters2 i)
           (for j 0 (numproducts-1)/2
                (drawsubcontact i j))
           )
        /* Place diffusion and transistors for AND plane */
        (for i 0 numproducts-1
           (for j 0 numinputs-1
                (drawtransistor1 j i)
                )
           )
        /* Place diffusion and transistors for OR plane */
        (for i 0 numproducts-1
           (for j 0 numoutputs-1
                (drawtransistor2 j i)
                )
           )
        /* Place nwells where needed */
        (makenwells)
        /* Link power and gnd */
        (linkvddgnd)
        )

(procedure (dumpstructs)
        (let ((verilog (outfile "verilog.out")))
```

```
            (for i 0 numinputs-1
               (printf "input[%d] = %L\n" i inputs[i]->name)
               )
            (fprintf verilog "\n// Product terms\n")
            (for i 0 numproducts-1
               (printf "products[%d] = %L\n" i products[i])
               (fprintf verilog "    assign #1 %s ="
                      (car products[i]))
               (foreach inp (caddr products[i])
                      (fprintf verilog " %s%s &&"
                             (if (equal (cadr inp) 'inverted)
                               "~"
                               " ")
                             (car inp)))
               (fprintf verilog " 1;\n")
               )
            (fprintf verilog "\n// Outputs\n")
            (for i 0 numoutputs-1
               (printf "output[%d] = %L\n" i outputs[i])
               (fprintf verilog "    assign #1 %s ="
                      outputs[i])
               (for j 0 numproducts-1
                    (if (outputinprod i products[j]) then
                    (fprintf verilog " %s ||"
                           (car (products[j])))
                    )
                    )
               (fprintf verilog " 0;\n")
               )
            (close verilog)
            )
         )

(procedure (plagen eqns lib cell)
         (let ((file (infile eqns))
              (pla (dbOpenCellView lib cell "layout" nil "w")))
            (initeqns)
            (while ((gets line file) != nil)
              numlines = numlines + 1
              (if (and (line != nil)
                   (line != "\n")
                   (strncmp(line "#" 1) != 0))
                (addeqn (lowerCase line))
              )
              )
            (close file)
            (dumpstructs)
            (rounduptoeven)
            (drawpla pla)
            (dbSave pla)
;           (dbClose pla)
            )
         )
```