Introductory Educational Laboratory Experience for Computer Engineering Undergraduates

by

Michael L. Jacknis

B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1998

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

[M. Eng.]
~~MASTER OF SCIENCE IN~~ ELECTICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 1999

Signature of Author: _____
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by: _____
Gill A. Pratt
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Introductory Educational Laboratory Experience for Computer Engineering Undergraduates

by

Michael L. Jacknis

Submitted to the Department of Electrical Engineering and Computer Science on May 21, 1999 in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science.

Abstract

This thesis presents a series of laboratory experiences directed toward students of introductory computer engineering. The design goals of the series and of individual projects are discussed. One project, a **digital piano**, is implemented and analyzed; the others are works-in-progress.

It is difficult to introduce students to this subject using hands-on design work because modern systems are constructed as monolithic integrated circuits using tools that are too advanced, expensive, and time-consuming for beginners. This thesis offers a way to abstract each of the important concepts so the student can experiment with them in the laboratory. Additionally, the projects are arranged such that they can be completed in a top-down sequence: Students use and experiment with simple processors before they are taught the digital logic composing them.

The projects are intended for student construction on a specialized laboratory kit consisting of an array of Field Programmable Gate Arrays. The staff provides final programming images for the FPGAs, so that the students may think of them as black boxes.

Thesis Supervisor: Gill A. Pratt
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

Many people and events influenced, inspired and made possible this work. I have had the pleasure of being a Teaching Assistant for several related MIT EECS classes. The students and fellow staff members I've worked with were patient, receptive, forgiving, and gave me valuable feedback concerning the art of teaching in general and the specific task of designing meaningful laboratory experiences.

Thank you to Marc D. Tanner and Rachael Lea Leventhal, who gave me the inspiration and confidence to pursue this project in the first place.

Thank you to Andrew "Bunnie" Huang, who designed and constructed the ingenious, reconfigurable, feature-packed hardware laboratory kits in a summer and a semester, using mostly his personal resources, and who went above and beyond the call of duty to provide temporary lab assignments for students during the Fall 1998 semester.

Thank you to Om Prakash Gnawali, Andreas Sundquist and Jason Woolever who are spending January 1999 and beyond implementing the lab assignments described in this thesis as works-in-progress, and who are providing feedback, support and continuity for this project.

Thank you to Gill Pratt, whose Leg Lab and free-form style gives one (in Bunnie's words), "the license to do cool things."

Thank you to my parents, without whom none of this would be possible in the first place.

This thesis is dedicated to the smart people who learn best in unconventional ways.

Table of Contents

List of Figures

Chapter 1:
Introduction

The Problem

The advancement of modern digital computation circuits presents an interesting challenge to educators: Modern circuits are usually constructed as single VLSI elements designed with several layers of software tools. The software assumes a prior understanding of the concepts of digital logic circuit design. However, most people in industry today have obtained this understanding through experiencing simpler logic circuits, from the days when it was common to prototype them by physically connecting hardware elements.

How do you impart a solid understanding of the fundamental principles of logic design when the simple logic circuits upon which they are based are no longer in common use?

How do you give students an introductory construction set for logic design, that offers an experience and understanding equivalent to hacking primitive hardware logic circuits, while involving the issues associated with the convenience and capability of modern VLSI design?

It is difficult to teach modern computer architecture and digital electronics: the state of the art involves sufficient miniaturization and integration that it is no longer possible to give a student a box full of digital parts and have them physically construct a modern digital computer. The concepts and techniques that enable modern computational machinery evolved from earlier, simpler, less integrated technology. In order to understand the latest innovations, and eventually introduce new ones, an engineer must understand the basic concepts underlying digital circuit design and construction. Engineering schools face the challenge of filtering the history of innovation and to provide students with an abbreviated experience that is equivalent to having experienced the evolution of technology first-hand.

This thesis will examine the current Computation Structures curriculum at M.I.T. It will define a set of laboratory experiences that will help students understand the curriculum. Some of these experiences will then be directly implemented and tested. Because of the amount of work that must be done to implement everything, some lab experiences will be implemented by people under the direction of this author and his supervisor.

Curriculum of MIT's Computation Structures Course: From Bottom Up to Top Down

MIT requires all of its Electrical Engineering and Computer Science undergraduates to take course **6.004**[1]**, Computation Structures.** Traditionally, the course covers the

[1] MIT students always refer to the course by number, even in casual conversation. They likely never heard of the name.

elements that make up the modern digital computer and how they are integrated. A conventional semester, prior to Fall 1998, might cover, in this approximate order:

- Boolean logic; DeMorgan's law
- Schematic symbols for logic gates
- Static and dynamic discipline for logic representation as voltage levels
- Flip-flops, registered logic
- Finite State Machines
- Binary number system
- Arithmetic Logic Unit
- Simple microprocessor
- Assembly Language
- Exceptions
- Memory; Caches
- Pipelining
- Theory of Computation and Information

People introduce new technology by using what they have and know to create something new and innovative. They explain their invention to others both in terms of the function of the new device and the already existing components and concepts that make it up. When an idea or product is sufficiently new, the best way to teach people about it is to explain it in terms of its more familiar parts. Digital computers were created from simpler digital circuits; in the early days of digital computing, the concept was easy to teach because people first became familiar with concepts of digital circuits. Digital circuits were created as special cases of analog amplifiers with a special nonlinear transfer curve that proved useful in this application. Then, digital circuits became a primitive concept, and they were used to design the digital computer. The digital computer is now a primitive concept, popularly used, studied and programmed without any understanding of its internal workings.

Soon, however, the new idea gains wide acceptance and use. People without the expert knowledge of the inventor begin to use the innovation without having a full understanding of its internal workings. The new invention soon becomes a primitive, familiar idea, from which new ideas and their explanations may be based. Computers soon became commonplace, and people started to talk about computers and their applications (software, files, etc.) without an understanding of the inner workings of the machines.

At this point, it no longer makes sense to explain the now established innovation in terms of more basic elements, because many people are not familiar with them. Instead, a reversed educational process is required to assist those people who are familiar with the new idea, but not with the components that were originally involved with its formation. These students need to be familiarized with the components of the innovation in terms of the innovation itself.

Gill Pratt refers to this as the Top-Down approach to teaching Computation Structures. It is possible to teach the high-level concepts before teaching the components, as long as the lessons and design problems are carefully thought out.

An important goal of this thesis is to support the top-down approach for teaching Computation Structures. A careful design of laboratory experiences allows for hands-on design and construction using the tools and concepts, while being accessible to the student at each point along the way of the top down curriculum.

By contrast, the traditional method is to review all the ideas leading up to the main goal of the course, by establishing a structured curriculum with a series of prerequisites. A list of concepts to be taught, in order, might appear as:

1. Basic (analog) circuit theory;
2. Digital circuit abstraction;
3. Logic gates;
4. Registered logic;
5. Finite state machines;
6. Datapaths;
7. The basic computer;
8. Bus architecture;

Unfortunately, this approach suffers from many disadvantages. With time, the elementary concepts at the top of the list become less familiar to students on an informal basis. Students find it hard to relate what they are learning to the ultimate goal of understanding computation. Modern progress brings additional concepts, added to the bottom of the list. This increases the total burden on the teaching facility and the students, who begin to feel that there is too much to learn before achieving any sense of satisfaction. It becomes difficult to introduce realistic-seeming lab design and construction problems involving the primitive elements, because in modern devices, these elements are integrated and not easily manipulated individually. Also, the state of the art has advanced sufficiently, that only a few specialists in industry need to continue to have a detailed design problem understanding of the low-level concepts. These specialists design automated software tools that are used by other engineers working at higher levels. Therefore, most engineers in training would make better use of their initial effort to concentrate on the problems involved with the concepts at the end of the list.

Chapter 2:
History of Laboratory Component for Computation Structures

The Beginnings

When 6.004 was initially established in 1980, a laboratory program was introduced whereby students would construct a simple 8-bit computer (called the MAYBE) using Mimimal Scale Integration (MSI) parts. The schematics for the computer were completely provided to each student, along with a schedule for parts of the machine to be constructed and tested. Although there was valuable observation of the machine and experience constructing it, students were not actively engaged in a design effort. Furthermore, many students did not appreciate the details of the design of the machine, and it became a large effort at wiring with little engineering educational value.

As the years advanced, the course sought to teach the inner workings of more complicated 32-bit processors. An easily understood, typical 32-bit processor, called the Beta, was studied in class. Because of the significant investment in the original MAYBE labs, they were retained, and the staff wrote emulation software that could run on the MAYBE to simulate a Beta using several MAYBE instructions per Beta instruction. This further separated the lab component of the course from the rest of the course, because students were no longer wiring up the machine they were studying, and because the students were not directly engaged in the design of the Beta-on-MAYBE emulator.

By 1993, in addition to the academic and educational-value concerns, the lab hardware was becoming damaged and unreliable. Teaching Assistants were spending much time debugging the wiring and physical parts of student's MAYBE computers, further reducing the benefit of the project to students. It was clear that new labs were needed for the course.

The Saga of the "New" Labs

There have been three efforts to develop new labs, of which this thesis forms a part of the third. All efforts tried to incorporate more modern logic design techniques. The first and third efforts involved use of Field Programmable Gate Arrays, while the second involved a specialized hardware description language called Curl.

The first effort required students to construct a Beta processor in a Xilinx FPGA using Xilinx schematic capture tools. While well-intentioned, the software for this effort was difficult to support and took unacceptably long times to compile. Students spent too much time learning the nuances of the design software, and could not glean the high-level ideas the labs were trying to teach. Also, the ultimate result of the student's effort was not physically visible: the FPGA was programmed with the student's design, and the FPGA then behaved just like the simulator did (except faster). It was not possible to open up the FPGA and place scope or logic analyzer probes, nor could students observe or rearrange separate physical entities as part of the construction process.

The second effort eliminated hardware labs and had students construct a Beta using a specially-developed hardware description language. Some students enjoyed the project and felt it helped them understand the structure of the Beta, but many found that the lack of graphical or physical correspondence frustrated them and made it difficult to learn from the experience.

The third (current) revision features a lab kit with an array of sixteen FGPAs available to students in the shape of a 4x4 grid. Students may connect physical wires between the FPGA modules. **Instead of students programming the logic circuits in the FPGA themselves, the staff provides a large variety of pre-compiled bit image files specifying an array of parts of various complexity. Students then use the pre-programmed parts to construct their project by connecting them with real wires.** Students choose what parts they would like to use, and where in the 4x4 grid they should be placed, from a computerized list. Parts include:

- Basic arithmetic and logic elements: 32 bit adders, AND gates, OR gates, MUXes, etc.
- Clocked logic elements such as registers and counters.
- Parts from which a 32-bit Beta processor may be constructed, such as register files, larger and slower memories, student-configurable control ROMs, etc.
- I/O parts, such as D/A converters (currently implemented as a D/A converter chip driving interface, with ribbon cable connecting to the chip located on a protoboard).
- Other specialized parts from which specific lab projects (discussed later) may be constructed.

To reduce the physical wiring effort, **blocks have the ability to transmit and receive 32 bits of data over one physical student wire,** using a 20 MHz serial data protocol. The staff can decide, for any given part, which connections will offer or receive serialized data.

Which New Lab is Best?

At this point, both the second and third revisions offer reasonable advantages to be used for the class. They are two different ways to look at the same thing. Advocates of the hardware description language point out that the simulator conveys concepts just as easily as using physical wires, and offers students the added convenience of being able to see as a text exactly what connections are being made, while working from any computer at any location. The use of hardware description languages is much more common in modern logic design than is the physical connection of discrete logic parts. Some people feel that the Curl HDL is an appropriate simplification that can most effectively introduce students to the concepts of HDLs in general. Proponents of the FPGA module lab kits feel that using physical wires to connect parts gives students who are seeing the material for the first time a more intimate, real understanding of the circuits being constructed. Students take this experience with them when they move on to modern logic design using software tools, and it may be the only experience they will have in which they are able to

physically test and observe the actual hardware in an open, exposed configuration. The kit is the best way we have found to open up a realistic modern digital computation circuit, usually buried in an integrated circuit, for direct testing and study. Also, some people find that constructing something using separate physical entities conveys an enhanced understanding of what is being constructed and how it works.

Computer Engineering Education at Other Universities

Xilinx Corporation currently maintains a University Program whereby FPGA products are donated to educational efforts such as this one. Their website, *http://www.xilinx.com/,* includes a section on the University Program, with links to our website and those of other participating Universities.

The **Georgia Institute of Technology** recently introduced a very interesting laboratory experience, targeted to students who have mastered the basics of digital design. Students work in teams to simultaneously design the hardware and software for a computing system of their choice. After extensive simulation, the hardware design is downloaded into a device known as a Hardware Emulator, essentially a grid of connected FPGAs.

Specialized software optimizes the location of parts from the design into specific FPGAs. Subsequently, FPGA implementation software routes the internal circuits targeted for individual FPGAs. Students observe the behavior of their system using hardware logic analyzers; this is a very satisfying conclusion to the experience.[2]

The hardware emulator enabled students to observe and test their designs on actual hardware without performing physical wiring. This process exposed the difference between simulation and reality, and encouraged students to work more diligently on their projects, knowing they would actually be implemented. Students observed problems on the hardware that had not been caught by the simulation. Also, the way the projects were organized into teams allowed each student to specialize on some aspect of the system, while allowing all students on the team to become generally familiar with the big picture. Communication skills and well-formed abstraction barriers were a must. The logistical use of the hardware emulator was considered; an FPGA was used to route signals to the logic analyzer, allowing students to quickly program their designs into the machine and continue where they left off, and avoiding mechanical wear. [2] Although the projects pursued on the Georgia Institute of Technology system are too advanced for 6.004 students, who are not expected to enter our course with prior digital design experience or knowledge, the success of their program should be used to help guide us.

Chapter 3:
The 6.004 Digital Piano

An important design goal for educational labs is that students experience variations on the material, allowing them to see the same concept applied in a different situation. I wanted there to be a lab for 6.004 that would require students to design a specialized datapath with data flow control, other than the Beta processor architecture that is extensively emphasized as a running example throughout the course. Additionally, I wanted the project to be apparently useful or functional, and enjoyable to use. I also wanted the students to study the basics of digital arithmetic processing. In short, the lab needed to cover:

- Finite State Machines
- Datapaths
- Digital Arithmetic in Hardware

I came up with the project of constructing a digital piano.

The Digital Piano assignment is reproduced in **Appendix A. The reader is urged to review it in conjunction with the following discussion.**

What is a Digital Piano?

The basic idea behind a digital piano is that data from a ROM representing a waveform is output through a digital to analog converter. If the ROM is programmed correctly, its address inputs are increased linearly and allowed to roll over, causing its data output to represent the wave changing in time. The rate at which the address lines are increased determines the frequency of the resultant wave.

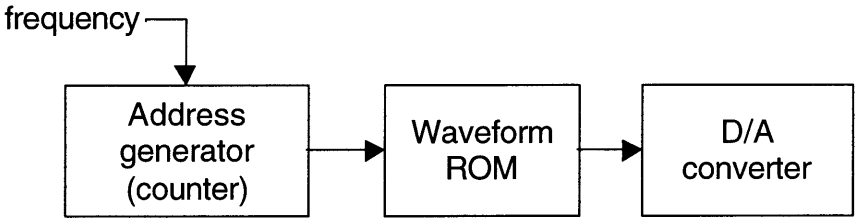


Figure 1: Simple Single-Note Generator

It is possible for more than one musical note to be sounded at once. Since multiple sounds superpose in air (pressure levels add together), an adder can be used to sum multiple simulated sounds:

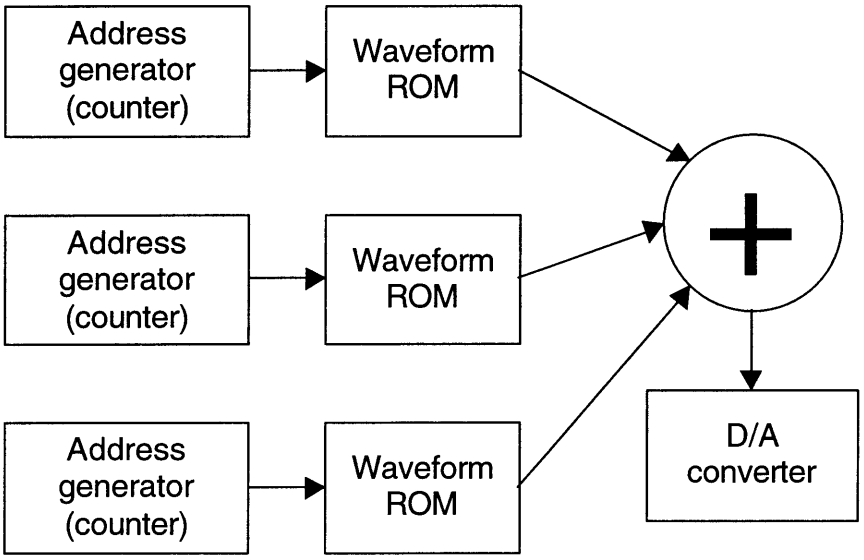


Figure 2: Simple Multiple-Note Generator

To save hardware and allow for a more interesting control system, a single waveform ROM can be used with an accumulator that stores values, and a multiplexer that selects which address generator to use:

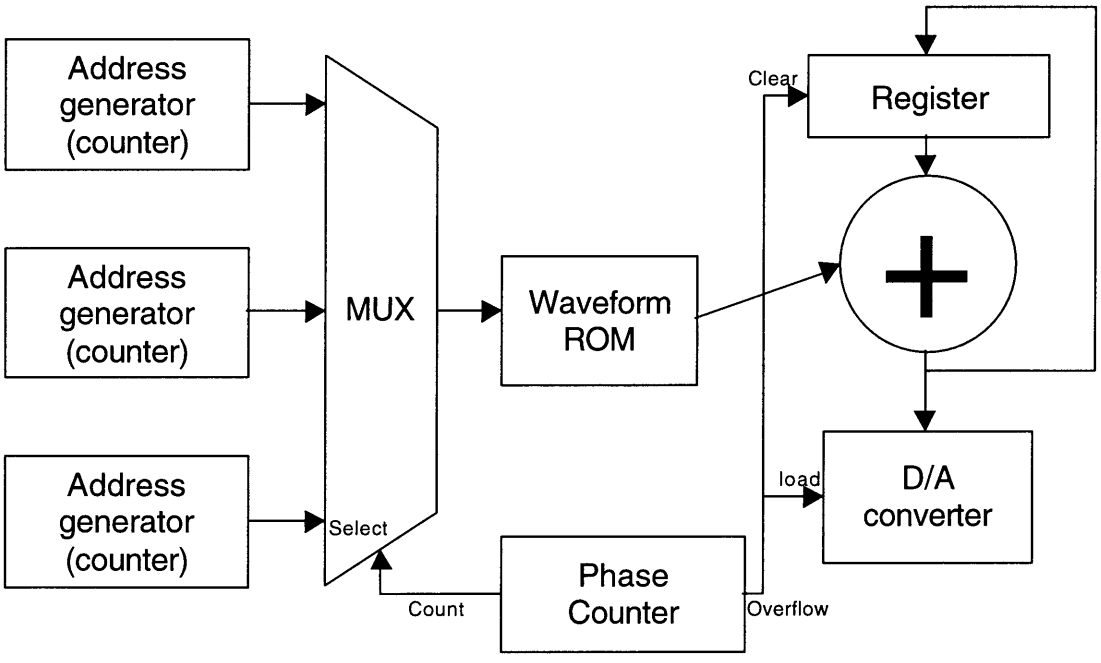


Figure 3: Multiple-Note Generator with Accumulator

In this approach, the Phase Counter in turn activates each of the frequency generators. When the Phase Counter rolls over, after giving each generator a chance to add its corresponding waveform point to the accumulated value, the final accumulation is passed to the D/A converter, and the accumulator is cleared. The maximum value of the Phase Counter and the speed of the clock driving the system should be determined based on the speed of the individual components in the system, the desired frequency range of the final audio output, and the desired maximum number of simultaneously playable tones. (The number of simultaneously playable tones is not the same as the total number of possible tones.)

This method requires several separate counters, each set to a separate frequency. To further save hardware, the address generators can be integrated into a single unit that outputs values proportional to the desired frequency:

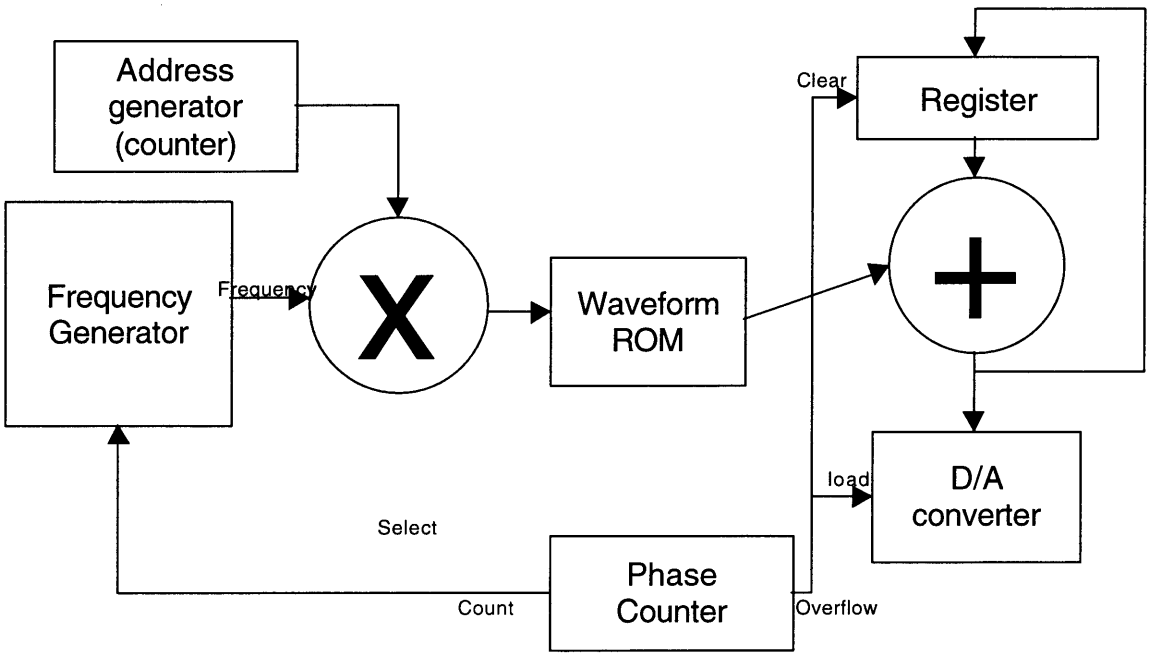


Figure 4: Multiple-Note Generator with Accumulator and Waveform Indexing Multiplier

There are several possible compact designs for the Frequency Generator. If this is to be used as a digital piano, the keys of the piano may be considered inputs to the frequency generator. If the Phase Counter counts up to at least the number of notes on the keyboard, one approach is to use a ROM that stores the note values for each key on the keyboard, and some gating circuitry that zeros out the frequency except when the key is pressed:

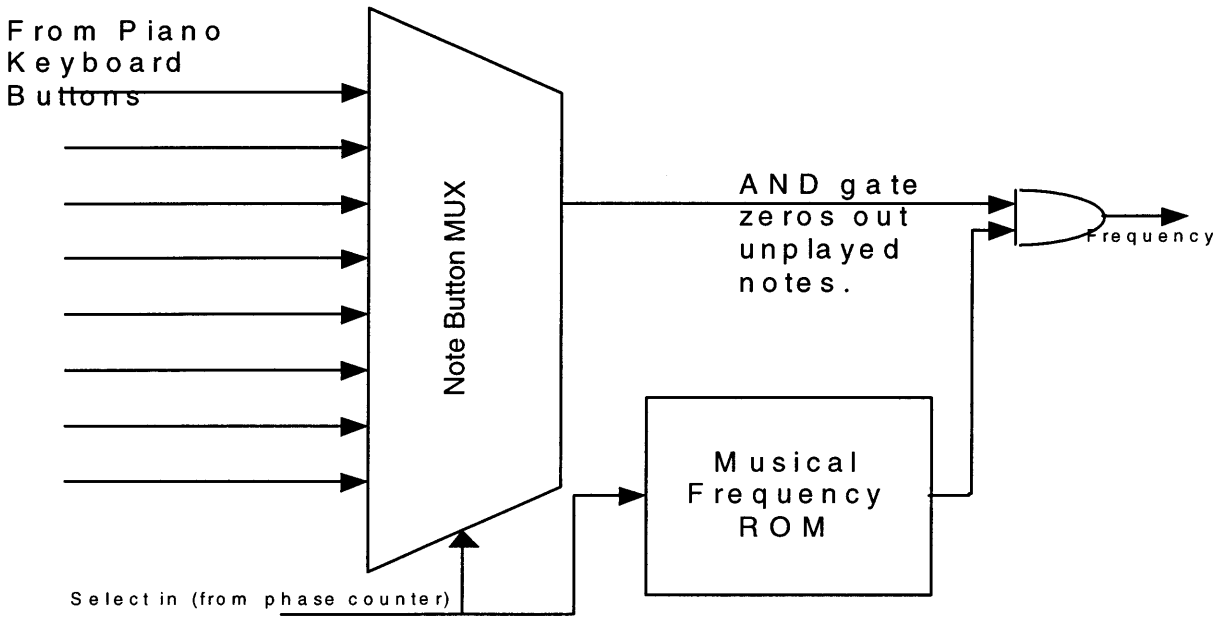


Figure 5: Simple Frequency Generator

This is not the only possible architecture for the frequency generator. For example, one may desire to have a greater number of possible playable notes than the maximum value of the phase counter. In this case, a system is needed to allocate phase counter values to specific notes being played. Or, one may want a simple architecture for playing a prerecorded musical sequence. This may involve a large ROM containing a time-sequenced list of notes or frequencies.

Designing the 6.004 Digital Piano Assignment

These days, the availability of fast, cheap microprocessors means that one is unlikely to actually construct a digital piano as described above. One would more likely get an off-the-shelf microprocessor or microcontroller, with supporting circuitry, and write a program to perform the desired task. However, the purpose of this project is not to build a product that could be successfully marketed. The goal is for students to gain experience, in an alternative context, with some of the hardware elements that make up a digital computer. There are several choices to be made, including which of the above designs should be utilized. Decisions should be made in terms of how they affect what students encounter.

The fourth design above was chosen as the base design for implementing the 6.004 Digital Piano. It allows for multiple notes, it is compact, and it involves several different kinds of components. The fourth design also implies a more complicated control strategy than the others. It also includes the use of a fixed-point fractional multiplier, which

students must construct from primitive elements. The first design is too simple. The second and third designs, while possibly useful in industry, involve too much repetition and not enough variety. Additionally, the fourth design allows for a variation in the system clock or rate of computation, as long as the master address generator counts at the correct rate. If the computations are made twice as quickly, the resulting sound wave is defined with twice as much resolution, but the frequencies don't change. This is good because it allows for a design in which various elements take unequal amounts of time to compute, more closely simulating real-world problems.

The next step was to constrain the design so that it could be completed in a reasonable amout of time, and so that the student effort could be directed toward understanding the concepts being covered in class. At the time this lab was introduced, in the Fall of 1998, there were not enough lab kits for each student, requiring students to submit their designs in advance, work in groups, and sign up for time-limited kit appointments.

It was decided early on that the lab would be intended to be solved in only one way. It is more of a puzzle than a true design problem; students are provided with exact quantities of parts with limited connections available between them. This approach is appropriate because the Digital Piano is intended as an early introduction to the use of digital logic parts. Students will later be given design problems involving less constraints and a free supply of parts. Also, the logistics of scheduling 250 students to use 10 lab kits during time-limited periods necessitated a standard technique for constructing the Digital Piano, so that overworked, underpaid staff members could become competent more quickly, and then assist students more expediently.

Designing a lab assignment like a puzzle requires the designer to first build a completed Digital Piano, and then break it down into sections. The granularity of the sections depends on the goals of the assignment and the restrictions imposed by the hardware lab kits. To do this, I roughly broke the design down into eight subcomponents, using varying levels of granularity, with the goal of each component fitting on one Lab Kit module:

- **Button Press Sensor** outputs a value representing the state of the lab kit's buttons.
- **Control ROM** contains a preprogrammed FSM controlling the rest of the system. The reasons for making it preprogrammed are outlined below.
- **Note Interpreter** contains circuitry from Figure 5 that determines what sets of frequencies should be sounded when.
- **Frequency Indexing Modules** contain the components from which the multiplier pictured in Figure 4 may be constructed. Unfortunately, because of lack of pins, it had to be arbitrarily split into two modules.
- **Waveform ROM** contains a sine function lookup table.
- **Superposition Processor** contains the register and adder in Figure 4 needed to allow multiple notes to be sounded at once.
- **Digital to Analog Output** accepts the final digital waveform and outputs it to amplification circuitry and a loudspeaker.

Details of the connections and behavior of each of these modules are included in the instructions to students, in Appendix A.

Prebuilt Note Interpreter

There are many ways to construct the circuit in Figure 5. If each count of the Phase Counter is regarded as a separate "voice" --- each count represents the opportunity to add one more sound to the accumulation --- there are several strategies for allocating and using each voice. Each voice can be assigned to a specific frequency, as shown in the example in Figure 5, or each voice can be assigned frequencies using an algorithm.

Additionally, the association of musical frequencies to musical notes requires a minimal amount of musical training, which the students are not expected to have.

Because of the ambiguity of this problem, I decided it was inappropriate for students to deal with in this introductory assignment. I provided students with a device that outputs frequency values in response to a phase count input.

Preprogrammed Control ROM

Originally, this was to be an assignment covering the design and application of Finite State Machines. I increased the complexity of the Note Interpreter, to accomodate multiple notes, in the hopes of having students experience the construction of a nontrivial FSM (something that cannot be reduced to a simple counter or linear sequence of states). I also wanted to necessitate the use of an advanced digital math module, the multiplier. Unfortunately, the additional complexity of the datapath, along with the desire to keep a single solution to the assignment, required the FSM to be given to the students. Additionally, at the time the assignment was issued, there did not exist software to allow a student to program data into an EPROM-like device. It was therefore impossible to enable students to build an FSM, because there was no way for them to enter a state transition table into the hardware. As a compromise, an FSM was provided, but students were required to figure out how to connect its inputs and outputs to the status and control ports of the rest of the logic.

Frequency Indexing Module (Multiplier)

The Frequency Indexing Module is a fancy name for a bit serial multiplier. It is so named because it multiplies the frequency being played, and because the result of the multiplication is an index into the sine wave lookup table.

The Frequency Indexing Module was partitioned down into the finest granularity, so that students could have experience with very low level logic components, and experience a significant design challenge.

Additionally, it contains its own concealed state machine, separate from the main state machine, that students build out of elementary logic gates. This allows students to see the same idea from several points of view. It receives a "start" bit, and several clock cycles later, it outputs a "ready" signal. The state of the multiplier is stored within the data being multiplied. There is something for everyone here: the ordinary students can

figure out how to build it, and the advanced ones can try to draw an equivalent state transition table.

There are just enough parts to construct the multiplier in the Frequency Indexing Module. The placement of the AND and OR gates is possibly the trickiest aspect of the assignment, because there is only one way (known so far) that it will work.

The multiplier includes a number of fractional bits. The frequencies contain two fractional bits, and the address counter contain five fractional bits. Therefore, the initial multiplication result has 7 bits. This is explained in the lab handout, because it is the first exposure students have to this concept.

The multiplier takes several clock cycles to complete its computation, whereas the rest of the system could theoretically compute a new value every clock cycle. As a result, there needs to be a system to manage the unequal computation times, and wait for a valid multiplication result. This is done by defining the multiplier to be a black box that accepts a "start" signal and returns a "ready" signal. The multiplication is carried out using two shift registers to store the numbers to be multiplied, and an adder/accumulator combination to keep a running total of partial multiplication results. The multiplication is initiated by loading fresh values into the shift registers. When either of the shift registers becomes zero, after all bits have been shifted out of them, the multiplication is considered to be finished. The key to constructing the multiplier out of the available parts is for the student to realize that:

1. Multiplication is performed by selectively adding shifted copies of one of the multiplicands based on the contents of the other multiplicand.
2. The state of the multiplier can be determined by the contents of the shifters, because, given enough cycles, they will eventually shift in all zeros. This state is used to control the shifters, as well as to create the "ready" signal.
3. Different values may take different amounts of time to multiply.
4. To start the multiplier, you simply load fresh values.

The multiplier as currently designed is not fast enough to accommodate the level of quality and resolution initially expected of the piano. The multiplier is limited by the rate at which serialized data is currently transferred between modules on the lab kit. In order to make the multiplier work, the student must connect the output of the left shifter to the input of the accumulator. This is a critical bottleneck connection, because it is possible for a number to need to pass through there on every multiplier clock cycle. In order to speed up the multiplier, that connection must be done internally. Fortunately, both ends of the required connection reside on frequency indexing module B, so it was possible to make an internal 32-bit parallel connection between the two, instead of routing the number through the wire the student is supposed to connect.

The next version of the multiplier will have to take advantage of the internal 32-bit connection to run with a faster clock than the rest of the system. From the student's perspective, this is a simple as reprogramming the affected module with a new, "fast"

frequency indexing module B. I recommend that students first experience the current implementation's audio output using a speaker and oscilloscope. It is instructive to see the expected frequencies being played, but with very jagged and distorted waveforms, so that the problem is fully understood before trying to solve it. I consider this problem to be an important feature of the project.

Implementing the Digital Piano

Xilinx Foundation Series tools were used to create fuse files that students can download into their nerdkits. The Xilinx tools support an easy to use schematic capture interface that allows digital circuits to be described as if they were built out of discrete logic parts. **Appendix B** contains a complete printout of these source schematics; the reader is encouraged to take a moment to study them.

Several logistical issues had to be addressed when using the Xilinx tools. The tools are intended to be used to construct a hierarchical project to be burned into one FPGA; there is no direct support for breaking a project down into parts burned into several separate FPGAs. Care was needed when storing the parts of the project in libraries, so that a single master copy commonly used parts could be altered when viewing any of the individual FPGA designs, rather than having several copies of commonly used parts that could become differentiated. Additionally, the implementation of the piano raises questions of the students' user interface: Students should be able to probe the parts and experiment with them individually, as well as having visual confirmation that a part was actually programmed. Lastly, a minor flaw in the Fall 1998 edition of the nerdkits caused one of the I/O module pins to be nonfunctional; this required a change in the Digital Piano pinouts to compensate.

Organization of Implementation Files

Xilinx tools refer to each FPGA design as a separate project. Each project may contain several schematic pages arranged in a hierarchical manner. The user can define macros for commonly used parts, and insert them into higher levels of the design. However, when designing a 6.004 lab, one must create *several different projects with a similar structure.* I created a standard appearance for a macro with all the connections for interfacing to the physical features of the module: the I/O pins, the clocks, and the LEDs, shown in Figure 6.

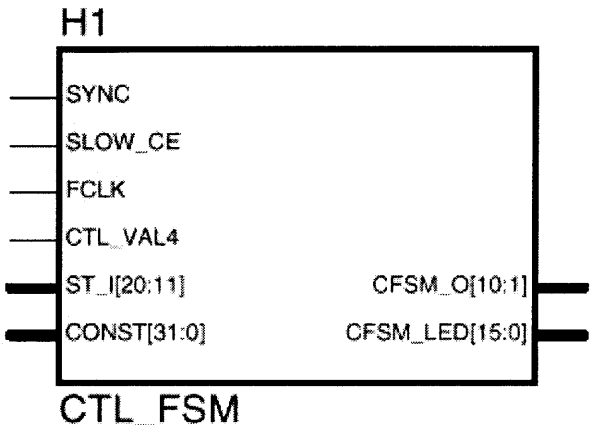


Figure 6: Standard Appearance for Module Contents Macro

I created a new project to store the macros for all modules in the Digital Piano. I then created a separate project for each module that used the macro from the project with the macros, connecting the macro pins to physical FPGA pins. This standardized the locations for all the module designs. In the future, if a complete piano were to be constructed on a single large FPGA, it would be as simple as wiring together the macros representing the modules, in the same manner that a student would wire together the piano using the current nerdkit.

Each module that is a member of the Digital Piano can be identified by means of a flashing leftmost LED. The number of times the LED flashes before pausing and repeating allows one to distinguish modules from each other. This technique helps students to confirm that the FPGA was successfully programmed. Additionally, the other LEDs in each module are used to represent important I/O pin states or internal states that might be interesting or useful to the student. In the next edition of this assignment, students should be encouraged to study the flow of information by single-stepping their clock and examining the LEDs. This approach should be used when designing other nerdkit labs.

If several modules are to be provided to students with very similar behaviors, the circuitry to implement all possiblities may be constructed for a single FPGA design, with only one of the structures activated when the system is running. This is an alternative to using a separate project for each module. A signal sent from the nerdkit to the module determines the module's identity and which structure should be used. This approach saves disk or EPROM space to store compiled fuse files and time to download them to the nerdkit. I did not use this approach because all the part of the digital piano have

vastly different behaviors, and I was not able to fit more than one of my designs onto a single FPGA.

Expected Student Response

The 6.004 Digital Piano should be thought of more like a puzzle than a true design exercise. There is only one known solution (with possibly one minor variation) considering the limitations of the available parts and pin connections. Most students, working in groups of three, were able to work through the problem. It is hoped that more advanced projects will involve greater latitude and require more student creativity.

The solution to the 6.004 Digital Piano is left as an exercise for the reader.

Improvements

Several things can be improved for the next edition of the 6.004 Digital Piano, and for other projects based on the 6.004 Rev. 3 nerdkit:

- **Standardized look-and-feel:** Each of the modules used in the piano has its own individual connection pinout. Additionally, internal state for each module was connected as much as possible to the LEDs on the modules, so that students can easily observe the behavior of the hardware. However, the scheme by which these connections and mappings were made was not systematic. Perhaps preprinted labels or cardboard cutouts, depicting the pinouts of the various available modules, and provided for students to use as they wish, would assist students in identifying which module was programmed as which, and what connections and displays are available.
- **Additional observation of the hardware:** The lab should include a more detailed section on debugging, testing and observation of the hardware. Students should single-step their clock and explain the results. Students should connect an oscilloscope to the analog output as well as serialized digital data.
- **More flexibility in design:** If each student had their own lab kit (a separate problem), it might not be necessary for the staff to constrain the design as much as we did. In addition to the specialized modules intended to build the piano, students could be given a general library of parts, that can also be used for other projects during the semester. The piano would be more of a design exercise than a puzzle.
- **Build-and-test in stages:** If enough lab kits were available, students could be instructed to build a small part of the piano, then observe and debug it, instead of trying to first design on paper and then wire up the entire project at once. Students could wire and test simple circuits during the early design phase, to enable them to better and more quickly understand the behavior of the parts available to them.
- **More Testing Functionality:** A number of modules could be made available to students to assist in debugging, such as a device that outputs notes corresponding to a pre-recorded song, or output the data they are receiving on the LEDs.
- **Clarification of instructions:** *Most digital piano parts have registered outputs.* This information, as well as the LED arrangements and pinouts, should be documented for students.

Chapter 4:
Developing The New 6.004 Lab Curriculum

A sequence of assignments is needed for the Computation Structures class, that supports the new top-down curriculum and that uses the new laboratory kits. A complete lab assignment package includes:

- Instructions to the student, specifying the goals, procedures and requirements for credit;
- A specification of the kinds of pre-programmed specifications that may be loaded onto kit modules;
- The actual downloadable module files that are used to program the lab kits.

Several downloadable module specifications are used in more than one lab assignment; some are unique to a particular lab assignment.

I have worked with Professor Gill Pratt to assemble a team of lab assignment developers, each of which was charged with the development of one lab assignment. Additionally, I developed one assignment myself. These two efforts constitute the deliverable results of this thesis.

The 6.004 curriculum focuses on the architecture of the Beta microprocessor, and uses it as an example to explain general concepts applicable to any current or future architecture. Students are expected to have a thorough understanding of the Beta; thus, a central lab assignment is for the student to construct one out of basic elements. However, not all lab assignments need to use the Beta example; in fact, it is too complicated for such subjects as pipelining. Therefore, while the academic curriculum will primarily use the Beta, the lab curriculum may use other architectures or specialized architectures. This adds variety to the assignments, keeps them simple and focused on specific concepts, demonstrates the use of a concept from different points of view, and makes them more fun.

In the future, additional lab assignment developers may need to be hired, in addition to a lab kit developer. Lab assignment developers work with a lab kit development environment using the Xilinx Foundation Series development tools. Lab kit developers improve the kit itself by constructing special-purpose modules that provide memory, digital to analog conversion capability, etc.

Here is the suggested lab assignment curriculum, as originally discussed with the lab assignment developers, for MIT's 6.004:

1. **Use a Beta in simulation.** Students use Betasim, a Beta simulator developed by Mike Wessler, to learn about the Beta instruction set, assembly programming, and register transfer architecture. Students build a program that produces fractal images.
2. **Fibanacci.** Students wire up a fibanacci number generator using a mux, a couple of registers, an adder, and other basic parts. This is an introduction to the mechanics of using the hardware lab kits.

3. **Use a Beta in hardware.** Students wire up a pre-built Beta to memory units. This is an introduction to the use of the memory unit module, and to running programs on a Beta on the lab kit. This lab was designed by Andrew Huang, using an FPGA-based Beta from previous 6.004 labs. Students use a pre-built line drawing output devices (see the Vector Graphics Lab, below) to observe their fractal code from the simulator executing on the hardware lab kit.
4. **Design the Beta control ROM.** Students wire a Beta lacking a control ROM, to a ROM they program themselves. This lab was designed by Omprakash Gnawali.
5. **Construct a Beta from basic elements.** Students fully construct a Beta out of individual elements. This lab will not be used until each student can be lent his or her own lab kit.
6. **Ant Controller FSM Lab.** Students design the controller for an ant to solve a maze. The ant has antenna for sensors, and it can choose the direction it wants to walk.
7. **The piano lab.** This lab introduces students to special-purpose datapaths and control FSMs, and to analog I/O. Students use special pre-built parts to construct an electric piano.
8. **Vector graphics lab.** Introduction to pipelining and data dependency. Students use several special-purpose modules that can be combined to create a 3-d vector graphics engine. A cube or other object is displayed on an oscilloscope in X-Y mode. The object rotates about its axis in response to the user's command. Students must design a specialized hardware processor using the available parts. Students must program the processor so that it meets or exceeds minimum performance criteria; students discover that they must be careful about the sequence of operations to achieve the required performance.
9. **CMOS Gate; Transmission line lab.** This lab was offered during a previous term of 6.004 (Spring 1997). Students construct and observe an inverter using individual MOSFETs. Students observe a periodic signal traveling along a coaxial cable with taps at regular intervals.
10. **Cache Controller Lab.** Students design a cache controller of their choice using basic parts, such as big slow memories, small fast memories, and an FSM.
11. **Virtual memory lab.** Students design or observe a virtual memory controller. They differentiate between the features of the VM controller and the Cache controller built previously. Finally, students integrate the two controllers and explore how cacheing and virtual memory interact.

These labs conform to the design goals originally set forth. Each one is progressively more challenging and less constrained than the previous.

What is a good lab assignment?

We discussed various design goals in general:

- Proceeds week-by-week thru the course, following the lecture notes and expanding upon the concepts.
- Demonstrates, isolates, differentiates, and creatively applies concepts.
- Incrementally constructs large systems from their easily-understood parts.

    - If lab kits are not available per student, and there is no way to preserve wiring, components simulating the results of a previous assignment may be used to allow continuity from week to week. Alternatively, such blocks may be provided first, for high-level understanding, after which the students construct the lower-level parts.
- Provides more than one way of seeing the same thing.
- Involves more than just wiring up a device presented in the lecture notes – there should be a small challenge or design problem involved:
    - The students may be given parts slightly different from the ones in the lecture notes.
    - Students might be asked to figure out how to connect new parts together to form a device with a desired behavior.
- No grunge – wiring is reduced by integrating parts with which the student need not be concerned.
- The assignment is fun – wherever possible, concepts are applied to new or cool tasks.
- The documentation clearly explains the goals, procedures, results, and requirements for credit.
- The assignment works as expected, such that the student should be expected to know or figure out problems without doubting the integrity of what she has been asked to assume.
- Lab assistants have a clear, easy procedure for check-off. Diagnostics are built into the lab if necessary. If the assignment was sufficiently challenging, the check-off procedure should not have to involve a zillion questions or take a lot of time.
- There is a lab-wide standard for the identification of programmed modules (IE the Morse Flasher on LED 15) so students and TAs can visually confirm the kit configuration.
- The lab sequence progresses to less constrained and more design-oriented labs later in the semester.

Intended 6.004 Lab Assignments for Spring 1999

To design a lab curriculum, we first list the major concepts of the course. We then list the lab assignments available and to be developed (based on cool toys people like to build and play with, applied to illustrate and feature blocks of concepts). Lastly, we identify what concepts each of the assignments is intended to cover.

The Curriculum

- Beta Programming: C to Beta assembly.
- Beta instruction set architecture
- Beta hardware architecture
- Digital Building Blocks
- The Digital Abstraction
- CMOS gates
- Construction of physical memory from gates and capacitors.
- Transmission line effects.

- Asynchronous inputs; metastability.
- Large memories: caching
- Large memories: virtual memory system with disk drive.
- Large memories: the TLB and integrating a set-associative cache with a virtual memory system.
- High-level performance issues.
- Quantum computers; DNA computers; other nonconventional computers.

Lastly, the most important new assignments were prioritized and assigned to developers. There were several concerned raised when developers started working:

- **Build a Beta computer from its parts.** Omprakash Gnawali is developing this assignment. Issues include the sense of satisfaction the student should have from using the parts to build the Beta. But what is the challenge involved with the lab, other than reading the lecture slide and wiring? Also, it is grungy. We may alter the Beta for this lab such that parts are missing, and the student needs to figure out a macro definition to make up for the missing parts (MUXes, constant inputs, memories, etc). We may also investigate the deployment of a wiring block device that each student can take home, so they may construct their Beta over a couple of weeks while preserving the wiring between sessions. This will be a one or two week lab. Some of the concerns about too much wiring and not enough thinking are mitigated by the fact that this lab will be offered early in the semester. Students need practice building things by the book... they will be challenged to design their own thing later on.
- **Vector Graphics Engine.** Andreas Sundquist plans to teach students about ALUs and pipelining by having them build a special-purpose mathematical processor that takes 2-d projections of 3-d coordinates as viewed from changing angles. The results are displayed through a pair of digital to analog converters on a standard Oscilloscope screen in X-Y mode. The challenge will be to design the mathematical processing devices using available parts. Students will learn about floating-point operations. The speed of computation will require simple pipelining of the circuit. Students will have to think about the time it takes for computations from each element to be completed. If there are elements that take a variable amout of time to finish, stalling of the pipeline will become an issue. Expected to be a two-week assignment, one week for design and one for implementation. Parts for the project include basic computation elements that must be combined, as well as support circuitry that draws lines between points by moving the scope beam smoothly over time. Students will need to implement a basic FSM controller for this lab.
- **Cacheing and Virtual Memory.** This is a three-week unit being developed by Jason Woolever with possible future assistance from Alexander Yip. A major goal of this assignment is to clear up the confusion among Cacheing and Virtual memory and show how these two strategies can be integrated into one large system. Jason will provide students with a "fast" memory and an intentionally "slow" memory, along with basic computational blocks (comparitors, counters, etc.) Students will need to design a cache controller of their choice from the computational elements, and connect it to the memories provided. A separate instruction memory, a Beta Lite

processor, and a "performance evaluation" device that snoops the memory bus rounds out the system. The second week, students use a prepared memory module, forget about cacheing, and construct a virtual memory system. They must build the virtual-to-physical conversion system. The third week, the students use a prepared cache device and prepared virtual memory device; these are combined into a final system that illustrates how they work together. Students have to keep track of which bits of the address go where. Students get to evaluate how well their system works, using a pre-made Beta program that does something useful...

The following assignments were considered, but not developed:

- **Serial data communication line between two kits.** Teaches asynchronous inputs, transmission line effects, maximum frequencies, clock skew issues, and I/O data communication protocols. The students might have to discover that they must "center the reception" by making the receiver delay a half clock period before sampling the data, otherwise the data changes on the clock and metastability issues may arise. (There should be an intentional 'bug' in the lab such that the output is random if you sample while the data is changing). A fun toy for possibly sending audio data in digital form, if a single pre-made audio-in and audio-out block is available.
- **The Beta Multiprocessor.** Students are challenged to design a memory system for sharing data amongst Beta Lite processors. Bus snooping, etc. The advanced students can construct a cache on top of the shared memory.

Results of Lab Development Effort: What do we do now?

There were a number of problems during Spring 1999. We did not have the resources to maintain the hardware lab kits, which still had mechanical design problems and were failing. Support for the hardware kits was reluctantly withdrawn mid-semester, and students worked instead on various software simulations.

The lab developers offered the following comments: [3, 4]

- The lab kits have reliability issues that must be addressed before reintroducing them.
- Although it is fun for lab kit developers to use the Xilinx tools to create interesting labs, it can be very constraining for the student to use these labs. Until a very large library of programmable modules is made available generally, there are not too many ways students can use their creativity to construct novel circuits with these kits.
- Once the concepts of wiring and debugging are out of the way, it would be helpful to use software simulation for further concepts. It is difficult to write a lab teaching very advanced concepts, such as pipelining or cacheing, using the Revision 3 lab kit hardware, because the amount of wiring is still too much, and the amount of creativity and flexibility afforded to the student is minimal.
- The lab kit can be thought of as a piece of hardware that simulates another piece of hardware. The use of serialized signals is a *representation* (i.e. simulation) of a 32-bit bus. Therefore, the developers thought that the same could be accomplished, more directly, using good software simulation.

It is not possible to expect students to debug their implementations on the new lab kits, because students have very little **debugging tools** at their disposal. Not only can't they observe the behavior of their project, but they cannot observe or debug the inner workings of the lab kit, which plays a role in the success of what is built on it. Questions arise, such as:

- *Are my signals being transmitted correctly through my wires?*
- *Is this module functioning, and was it actually programmed?*
- *Did the data for this RAM module make it from the computer to the kit?*
- *Is there a clocking or timing issue with regards to data arriving through the student-attached wires? Did any wires fall out?*

If the purpose of having a hardware kit is to promote a sense of practical reality and teach the student debugging and problem solving skills, then the student needs tools such as oscilloscopes, logic analyzers, specialized modules, and software to assist in observing the behavior of the kit. Students need to understand how to use these tools to observe the problem and reach a solution. In many cases, problems with the behavior of the kits were caused by loose internal kit connectors and low-level software problems with the underlying kit infrastructure. These are issues that the student cannot be expected to deal with, because we haven't taught the student how the kit works internally, nor provided any documentation or debugging tools. **We need to obey the abstraction barrier we have established for our students, and own up to our maintenance responsibilities on our side of this barrier.**

Feedback was also sought from previous students who took the class using the original Maybe hardware labs, constructed with Minimum Scale Integration TTL parts. That experience had the important side-effect of teaching debugging skills. Students had to reason through what they built, what they expected, and what they observed, in attempting to locate the cause of the problem. This skill helped one student immensely when moving on to problems in industry[5]. It is very difficult to duplicate this experience using only software simulations. Simulations are imperfect, and they tend to hide many issues and work more optimally than real life does.

There is a discrepancy between the opinions of the current lab developers and the recent graduates who experienced the original Maybe labs. The developers generally think that a wired kit is no longer necessary. The graduates think that the original wired kit was very helpful in certain limited ways.

A hybrid solution, involving some use of a wired nerdkit, some software simulation, and a hardware emulator as in the Georgia Institute of Technology[2], may be needed to teach modern concepts while also giving the student a sense of practical reality. Some labs using Mimimum Scale Integration parts, a breadboard, real wire and real debugging still have their place in an introductory curriculum. At each stage in the curriculum, the most appropriate laboratory medium should be selected to convey the concepts.

Chapter 5:
Physical Lab Environment

Students work more effectively in pleasant surroundings. Unfortunately, it is nontrivial to configure a laboratory with 32 workstations and a lab kit connection for each.

For security reasons, the computers had been locked inside standard 19" equipment racks. Fans used to cool the enclosed equipment generated excessive noise in the lab. The use of GFCI circuit breakers caused excessive false alarms; workstations would frequently turn off in the middle of being used because of a tripped breaker. Workstations and connections were not identified, causing confusion when attempting to use a computer with an attached lab kit.

I first implemented a numbering standard so that each workstation location could be uniquely identified. I then hired students to identify the electrical power feeds from each desk area, and separate power circuits feeding computers and monitors from those feeding tabletop power strips intended for powering the lab kits. A detailed map of the power distribution in the room was placed at each station and in the circuit panels. I then had an electrician rearrange the circuit breakers as needed so that computers would not be "protected" (in actuality, turned off excessively) by GFCI. Circuits protected by GFCI were clearly labeled at their destination outlets.

All the wires associated with a particular workstation and table (power, monitor, keyboard, mouse, serial cable, and parallel cable) were physically bundled, so they would be identified and so that the desks could be more conveniently moved for maintenance access. This allowed students to connect their nerdkits to the workstation they were using, rather than a different one with a similar-looking cable.

Lastly, I obtained fan speed controllers and standard electrical boxes, and, with the help of TAs, assembled these into units that were placed inside each cabinet. The sound level was significantly reduced, while maintaining sufficient air flow.

If these issues are carefully considered when a new laboratory facility is planned, it will not be necessary to redo the installation work later. If a formal procedure is established for initiating and concluding the semester, then security concerns and storage issues can be adequately met.

Chapter 6:
Remaining Work

A significant effort is still needed for the 6.004 labs to be acceptable over the long term:

Lab Assignments:

- Develop several more assignments, including an MSI-based lab.
- Integrate the labs into an environment with standard look-and-feel (i.e. with standard LED lighting configurations)
- Implement a large suite of modules so that students have more flexibility using the lab kits. Allocate organized disk space so that advanced students who design their own modules can upload them into the general distribution, for other students to use in the future.
- Provide the software and documentation needed to enable interested students to design their own modules.
- Future Lab development: make a developer's platform that minimizes the amount of work and learning a lab developer must do to create new modules and have access to all the kit's capability. (Can also be used by advanced students.)
- Decide what assignments are more appropriate for software, and select an appropriate platform for simulation.

Lab Kit Hardware:

- Implement other hardware application modules, such as D/A conversion and memory; either as boards that screw into the kit or boards that connect to the 20-pin connectors.
- Program the EPROM and microprocessor inside the lab kit to store the images for the commonly used library of parts, so that the lab kit becomes self-contained and independent of the computer, for basic operation. Program the kit to persistently remember the location of modules.
- Improve the software interface, *JTerm.*
- Document the use, internal workings, construction and repair procedures for the Rev. 3 nerdkit.
- Identify and standardize multiple platforms that can host labs, including the Rev. 3 nerdkit, a breadboard with MSI parts, and software simulation, such as Mike Wessler's Betasim and Pspice.

It is hoped that future students will benefit from and enjoy new introductory 6.004 labs. We hope to give them an appreciation for the foundations of modern digital computing, allow them to construct devices that represent recent advances, and encourage them to explore future innovations, while maintaining a sense of reality and developing practical design and problem-solving skills.

Appendix A:
6.004 Digital Piano Instructions for Students

On the next pages, the original instructions provided to students in Fall 1998 are reproduced. The paper describes the purpose and usage of the hardware lab kits, specifies the modules available to students for the piano, and guides students in completing the project. Although there were a limited number of lab kits available, it was hoped that one kit per student would be made available in future terms.

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 1998

Lab Assignment #2	**Issued:**	**Wednesday, 11/11/98**
The 6.004 Digital Piano	**Design Due:**	**Wednesday, 11/18/98**
	Implementation Due:	**Wednesday, 11/25/98**

Early points start 3 weekdays prior to due dates. Late points deducted per weekday (*holidays excluded*). Limited lab kit availability makes early completion to your advantage (see below).
Earliest day to get design checked off: Friday, 11/13/98.
Earliest day to use the lab kits: Wednesday, 11/18/98.

Labs this term are new and experimental; we are asking for your patience and understanding. Be sure to ask a staff member if there is any instruction or step that does not seem clear. It could easily have been our omission in the handout. This assignment involves the use of ***new hardware lab kits****, and we hope you enjoy it!*

TABLE OF CONTENTS

OBJECTIVES

In this lab project, you will use the new 6.004 lab kits to **construct a digital piano.** *Don t panic!* Prefabricated subcircuits and detailed instructions will make this task accessible to you. ☺

- You will gain additional experience designing **datapaths**. This project uses logic parts you are familiar with to construct a special-purpose datapath.
- You will use a simple **Finite State Machine** to control the system.
- You will learn about the reuse of circuitry through **time multiplexing**.
- You will gain additional experience with binary representation of numbers, specifically with **sign extension**, the representation of **fractional values,** and arithmetic such as **multiplication.**
- You will design a **serial multiplier**, optimized to reduce the amount of circuitry at the expense of increased computation time.
- You will be introduced to the **New 6.004 Lab Kits** the greatest thing since the Maybe !
- You will learn about **interfacing** digital projects to the outside world, by using circuitry that receives input from buttons and produces audio output with a **Digital to Analog converter**.
- *You will have fun constructing and using a* ***cool toy!***

This project is either very easy, or very hard. It depends on whether or not you plan your time and your work. As explained below, you will be working in teams, and a clear wiring diagram is required before physical implementation can take place. If you work systematically, collaboratively, and confidently, and ask questions of your teammates and the staff, you should have no problem.

PREREQUISITES

In order to understand and complete this assignment, you should understand basic circuits and laboratory procedure. You should understand how to use **wire**, wire **strippers**, a **protoboard**, and an **oscilloscope**. You should understand the **pin numbering standard** for DIP ICs. If you feel unfamiliar with any of these things, a staff member will be happy to bring you up to speed; **get help now!**

You need to understand material presented earlier in the course concerning logic elements, FSMs, ALUs, binary representation, and timing.

STEPS FOR COMPLETION; DEADLINES; LAB PARTNERS

You will construct this project on the **brand-new reconfigurable 6.004 lab kits** designed by Andrew Bunnie Huang. We only have **20** of these lab kits available for use by the entire class, and we are still fixing some of them. Therefore, we have developed the following policies:

- You should **form a group of two or three people**. Although all members of the team will help each other and construct the final project together, one person can specialize in the datapath, one in the FSM, and one in the serial multiplier design. As you design the part of the system you specialize in, you should consult your teammates for help, and to ensure the three designs are mutually compatible. There is an unavoidable dependency in that the datapath needs to be designed or at least specified before the FSM can be completed. (Two-person teams can share the work of one of the three components; no big deal.)

- If you have trouble forming a team, please write your contact information on the laboratory blackboard, or talk to your T.A.

- The **collaboration policy** for this assignment is similar in principle to what is on the Web, applied to teams: Teams should do their own work, consulting other teams as needed. Members of teams should consult other teammates, but each member will present what s/he created.

- The team will need to **present a completed wiring diagram** to a staff member in lab before being allowed to implement its design on a lab kit. Each of the team members can present their part of the design, and the team as a whole will discuss it with the staff member.

- Each team will **work together in one sitting using one lab kit** to implement its design.

- Once the team has checked off its design, it may either proceed to a lab kit immediately (if one is available) or **make an appointment** to use a lab kit later. It is to your advantage to check off your design early to have more of a selection of appointment times.

- If you **don t show** for your appointment (or arrive **late**), another group will be allowed to use the kit, and you will need to choose from remaining available appointments, or try your luck coming to lab without one. We ll allow until five after the hour in case you are coming from another class, but that s about it. If one of your team members shows, he/she may begin work, but your project needs to be finished ten minutes before the end of your team s appointment. (explained below)

- We may store the lab kits in a secure location, in which case you will be told how to obtain one when your design is checked off.

- Although you may change your appointment, you will need to select from currently remaining ones, and irreversibly give up your old one.

- With all team members present and working together, and with a completed wiring diagram available, wiring up the kit should be a breeze. ☺

- **The amount of time your team may work on the kit is regulated.** Selecting an earlier day to implement your design allocates you additional time to use the kit (at your one session), making things more convenient. (see table below)

- **If your project is not finished ten minutes before your kit time expires, and people are waiting to use the kits**, a staff member will ask the team for a **demonstration of partial functionality.** You will then be **required to disassemble your unfinished project**. (The ten minutes is the time for the staff member to check you off, and you to clean up.)

- For simplicity, the same grade (hopefully a good one) will usually be given to each member of the team. However, the staff reserves the right to assign individual grades in cases of partial functionality or if a team member requests this (with an explanation). We want to give you good grades for good work, but we will consider difficult situations on a case-by-case basis.

- Depending on how things go, we may **announce lab kit availability** over the 6.004 Zephyr instance, or through some other real-time electronic means. You can also call the lab at 3-7976 and hope that the person who answers is friendly enough to help you. ☺

- Despite these restrictions, we think you will finish your project quickly and **have fun** with the results. We welcome comments about the kit allocation policy to 6004-head-la@mit.edu.

- This policy is **subject to change** via emails and web-based announcements.

The following table indicates for how many hours you may use the kit, based on the date and time period you do so (whether or not you have an appointment). The last row of the table suggests the approximate maximum number of appointments that could be booked that day. This does not consider that some of the shorter appointments might not allow sufficient time for students to complete their project, especially if they don t have thorough plans. You are encouraged to get your design completed early so you may earn a longer appointment. Overnight appointments must be checked off the next morning, when a staff member returns. Individual cases may vary depending on the density of surrounding appointments and availability of staff.

Time	Wed 11/18 -Th 11/19	Fri 11/20- Sun 11/22	M 11/23	T 11/24	W 11/25	R 11/26 & on
10AM-9PM	All Day	5	4	3	2	Staff not available during holiday. If you work on the lab, you should ensure that you can get checked off.
9PM-10AM	Overnite	Overnite	Overnite	Overnite	Holiday	
Max. appts?	40	60	60-80	80-100	80-100	

There are approximately 300 five-hour, all-day and overnight slots, and 250 students in the class (only about 85-125 teams). You *can* get a long appointment.

INTRODUCTION TO THE NEW LAB KITS

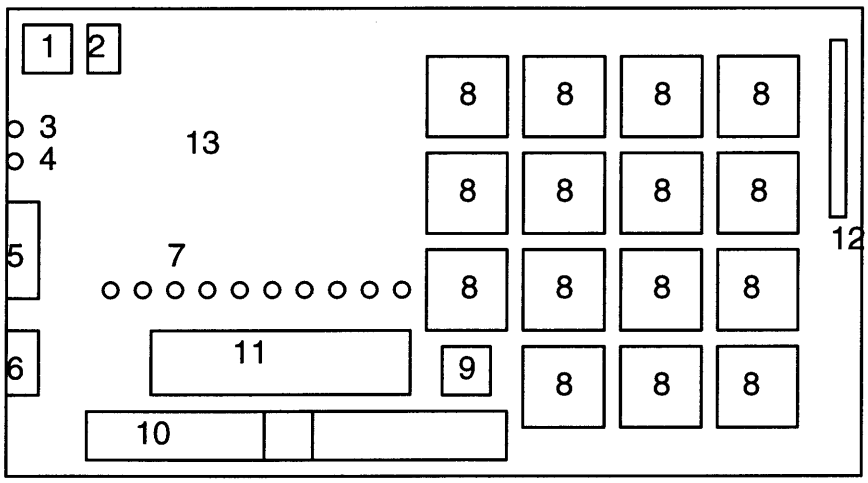
You should understand the hardware we you will be using to implement your design. However, if you have trouble understanding this section, reread it after you study the rest of the lab and examine a lab kit.

The new 6.004 lab kit, AKA **Electric Lego,** is a general platform for allowing students to build digital systems out of components specified by the teaching staff. The kit features reconfigurable blocks composed of Field Programmable Gate Arrays. Each block can assume arbitrary behavior, and students can wire signals to connect the blocks together to form a larger system. The 6.004 staff can reprogram the FPGAs as required for the desired lab assignment. For example, the staff might choose to provide an ALU, some registers, and a control ROM as basic building blocks for one assignment. Students would wire these parts together to form a Beta computer. The staff could later reprogram the kits for the next assignment to provide students with more basic building blocks, so students can wire up their own ALU out of its typical parts. The staff could lastly provide an entire Beta computer pre-built on each of the modules, and students could wire the Betas together to form a Symmetric Multiprocessor.

The kit interfaces to the Serial and Parallel ports of a P.C. Software allows the student to specify the physical locations on the kit for the building blocks available for the current lab project. Additional software allows the student to specify the contents of ROMs. Future software will help the student analyze how she configured and wired her kit, and assist with troubleshooting.

PARTS OF THE KIT

The important parts of the kit are illustrated in the following picture: *Note: Despite our best intentions, we goofed and the kit cover will not close.* ***Do not force the cover closed! Just leave the kit where it is, (or follow our directions for obtaining and returning it) and leave the AC cord and serial line connected (unless specifically directed otherwise).***



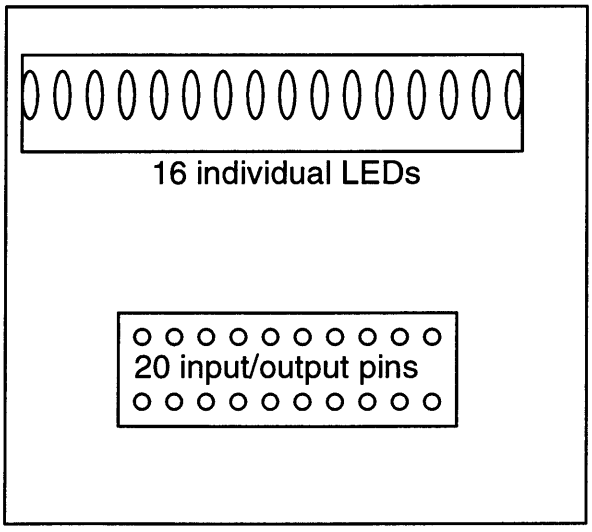
1. AC Power Cord
2. AC Power Switch

3. Reset Button (disregard)
4. NMI Button (disregard for now)
5. Parallel port connector to computer (for high speed data transfer; not used for now)
6. Serial port connector to computer (for data transfer and kit configuration)
7. Buttons for student use; can control projects.
8. There are 15 general configurable LEGO blocks.
9. There is one block specially designed for use as a Control ROM. (not used in this lab)
10. The Control ROM block has many connectors for the address and data lines.
11. There is a small protoboard for constructing circuits.
12. Inter-kit connector (disregard for now)
13. Future location of fancy display screen.

THE RECONFIGURABLE BLOCKS

Students can wire connections between the reconfigurable blocks. Each block has 20 electrical connections. Connections may only be made from one point to another; signals may not be connected to multiple destinations, because of electrical constraints (transmission line effects). To allow for multiple destinations, several output pins may reproduce the same signal, at our option.

The following is an enlarged illustration of a block:



Each lab assignment will list a choice of personalities that each module may assume. Each possibility has a unique pinout associated with it. Your job is to assign each module a particular behavior, construct a wiring diagram based on the pinouts of the modules, and wire the kit to demonstrate the final result. You may also have to specify the contents of ROMs.

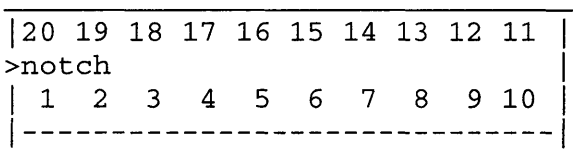
An underlying serial protocol makes some physical wires convey multiple bits of information. Each bit is transferred over the wire one-at-at-time; the kit takes care of this for you so you can pretend the wire is really a bundle of wires. The pinout for each possible configuration specifies which pins are inputs, which are outputs, and which pins are *serialized* (contain several bits of data).

Some basic signals, such as a clock, are already distributed to all modules for use in registered logic (through the kit s substrate). If a register or counter is provided, the part is assumed to be clocked by the system clock unless otherwise noted.

CONVENTIONS FOR SPECIFYING BUILDING BLOCKS

In this class, the staff will provide you many building blocks that you may choose to load into the modules on your lab kit. We need a straightforward way to define what each one does and how to use it. We will do this using a **description** of the component, a **diagram** (if necessary), and a **pin list** showing how you can wire it up. If the circuit uses the LEDs on the module on which it is running, that will be explained, too.

The **pin list** will refer to each pin location by **number**. If you look closely at the electrical connector on the 6.004 Electric Lego module, there are two rows of pins. *It looks just like a Dual Inline part, so we will use the same pin numbering convention.* In case you have forgotten from 6.002, the numbering looks like this:



Additionally, we will have the loose convention that the *top* pins, 11-20, are *inputs,* and the bottom pins, 1-10, are *outputs.* This makes visual sense. There are times when we need to violate this rule for one reason or another. So, the pin list will include the following information for each used pin:

- Pin number.(s)
- Pin description. What other kinds of pins might it be expected to connect to?
- Is it an input or an output?
- Is it a single bit or is it serialized, and to how many bits?

You may leave unused pins disconnected.

MULTIPLE COMPONENTS ON ONE MODULE

Sometimes we will want to provide many small parts for you to build from: a collection of registers, MUXes, gates, etc. If we do this, we may choose to provide several independent subparts on one module. Don t mix these up when referring to the pin list.

GETTING STARTED: THE 6.004 DIGITAL PIANO

For this project, you will be constructing a **specialized datapath** that functions as a **musical instrument.** You will be constructing this device out of **preconstructed modules**. Some building blocks have many things pre-made for you; other building blocks contain more elementary components.

You complete this project in three steps:

- Design the system out of the given parts. This includes the **datapath**, the **multiplier circuit**, and the **Control FSM** (which we ve provided with unlabeled control signals). This project divides nicely into a team of two or three people, each specializing in a part of the system.

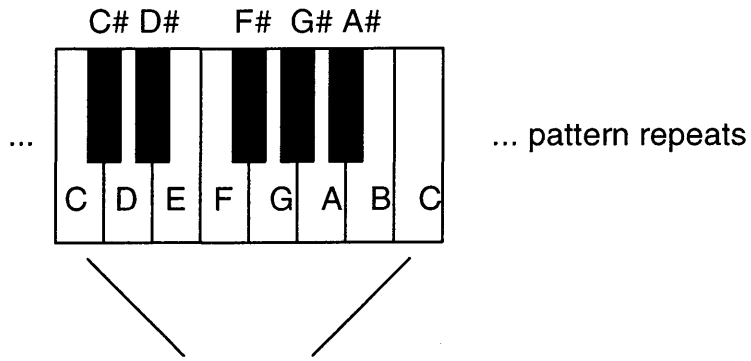
- Present your design to a staff member for a **design review.**

- **Implement** your design using Andrew Bunnie Huang s **Electric Lego Lab Kits,** Mike Wessler s **Betasim Simulator,** or both. This document includes instructions for implementation on the lab kits. If we choose to accommodate implementation on Betasim, you will be updated. Your final implementation should require about **25 wires** in total.

Author s note: This project was designed with the consideration that students would need to share lab kits, and that each student team would have a limited amount of time to build, demonstrate and use their creation. Therefore, the project is more cookbook and less design and debug than what one might desire. It is hoped that future revisions of this assignment will accommodate greater variations in the final project, and allow students more freedom to design their own system and figure more things out, while remaining within students reach.

PLAYING AND HEARING MUSICAL NOTES

Briefly, sound is composed of vibrations that occur at various frequencies. Music is composed of sounds called *notes.* If you look at a piano, each of the keys on the piano corresponds to a specific note, heard when you strike that key. Here is a picture of part of a piano, with the names of the notes and the corresponding frequencies filled in:



The C on the right has twice the frequency of the C on the le

Note Name	N	Frequency = 440 * $2^{n/12}$ Hertz
C	-9	261.63
C#	-8	277.18
D	-7	293.66
D#	-6	311.13
E	-5	329.63
F	-4	349.23
F#	-3	369.99
G	-2	392.00
G#	-1	415.30
A	0	440.00
A#	1	466.16
B	2	493.88
C	3	523.25

Each note is assigned a frequency using a formula based on how many notes away from A it is. In the table, *n* represents this distance. Only the frequencies in the table, and other frequencies computed similarly, may be sounded by a musical instrument (simplistically speaking).

If more than one key on the above pictured piano is pressed together, the notes superpose (the corresponding parts of the waves add together). If we wish to create a digital piano, all we have to do is make a device that can generate sounds of these specific frequencies, and add these sounds together in case of several notes being played at once. The piano should also know how to output sound waves that are realistic (one could start with a sine wave)

DIGITAL PIANO BLOCK DIAGRAM

Based on the above explanation, here are the design goals for the 6.004 Digital Piano:

- It should have a set of buttons, each of which correspond to one of the frequencies listed in the above table (since there are 8 buttons on your lab kit, we ll use only the white piano keys: C,D,E,F,G,A,B,C).

- When a user presses one or more of the buttons, a sine wave is produced and a sound heard containing the appropriate frequencies.

- The user should be able to have the piano play automatically by referring to a stored song represented as lists of notes.

In order to achieve these goals, the 6.004 Piano will need to have the following basic parts: (details are provided later)

- A **note interpreter** receives button presses (or simulated button presses when the piano is automatically playing) and produces a list of frequencies to be played. We have constructed this part for you, using the buttons that are already on your lab kit, a ROM that knows the frequencies of all the notes, and some other things. We have included another ROM with some favorite songs, which you can hear using a special button on your lab kit.

- A **waveform ROM** contains a lookup table of the sine function. ***The piano simply needs to cycle through reading this ROM at different rates of speed depending on which note is to be played.*** There are 256 locations in the ROM, each of which represents the sine function to eight bits of precision. The ROM contains one full period sine wave.

- A **frequency indexing counter** cycles through the waveform ROM addresses at different rates of speed, depending on the frequencies provided by the note interpreter. **You will play an active role in designing this device from simpler elements.** You will need to construct a **multiplying circuit** for this part.

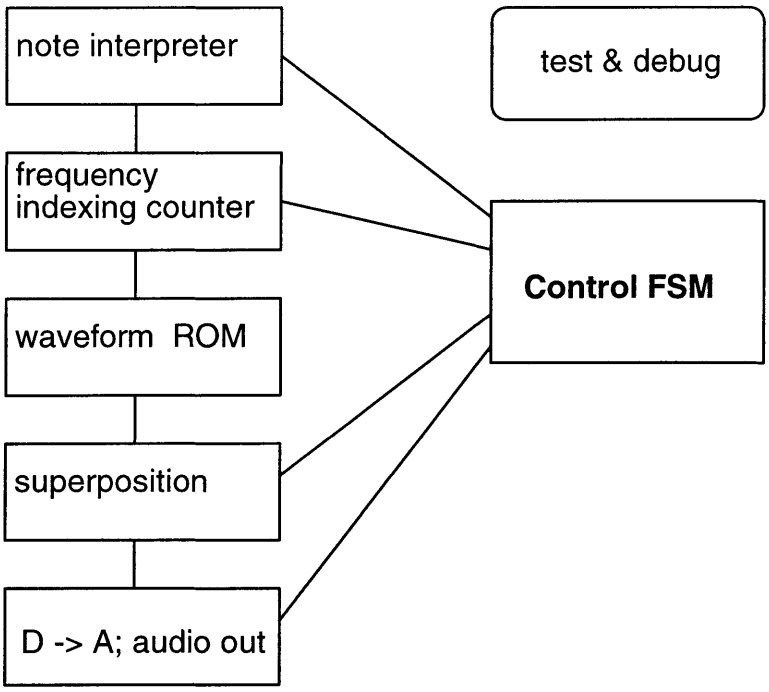
- A **superposition processor** takes the waveforms of all the notes currently being played, adds them up, and scales them. We ve built this for you.

- A **Digital to Analog converter** takes the final waveform from the superposition processor, scales it, and converts it into an analog voltage. From there, a power amplifier drives a speaker. We have provided this part for you with some circuitry on the protoboard.

- Most importantly, a **Control Finite State Machine** orchestrates the entire process, searching for notes that need to be played and shuttling the data through the system. We ve provided the FSM for you, but you will need to figure out which signals go where.

- A **Test and Debug Module** provides test signals useful when constructing the system.

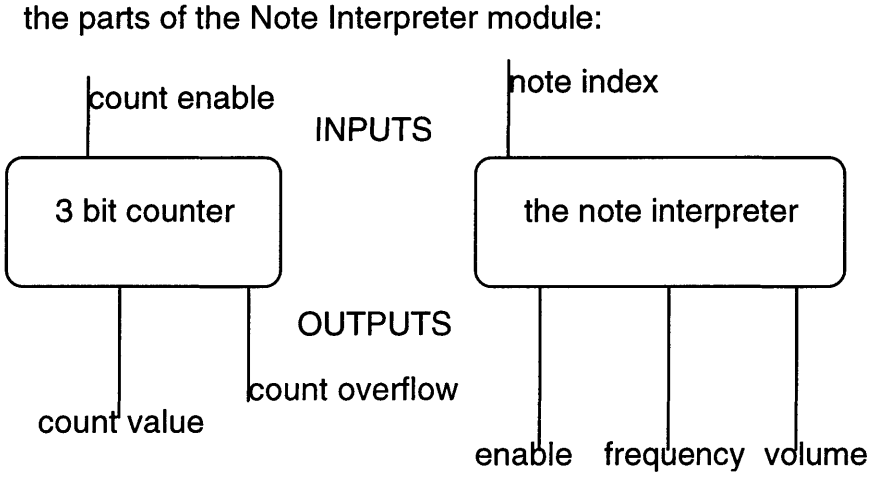
Here is a basic block diagram of the system: (part of your job is to augment this diagram with an exact specification of what signals pass through each interconnection):



AVAILABLE BUILDING BLOCKS FOR THE DIGITAL PIANO

Your job is to figure out how to use all these parts to construct the piano. Some parts are built up to a high level; others require you to perform additional design work. The **pinout list** provided for each part indicates what connections the part needs; the **pin numbers** are used when you wire up your design on the actual lab kits. You might find it helpful to check off each signal on the pin lists as you figure out all places where it should connect.

It might help to draw a picture of each part as you study it. For example, here is a picture of the Note Interpreter:



Often in the following descriptions, a part will refer to how a single note is processed, but our piano can play more than one note at a time. The Control FSM will cycle through all the notes that want to be played simultaneously, shuttling this data through the system in an orderly manner.

Note Interpreter

This part allows the piano to know what notes are to be played. Each of the notes that currently simultaneously want to be sounded is assigned a three-bit identification index number. The Note Interpreter receives one index number (between 0-7) as input, and outputs a corresponding frequency for that note in Hertz. It also outputs an enable bit to indicate whether the note wants to be played, as it is rare that all 8 notes want to be played at a time. The Note Interpreter might output a high enable bit only for a few of the possible index numbers. Lastly, the Note Interpreter outputs a volume control signal that we explain how to use.

The recommended way to poll the Note Interpreter is to have a counter cycle through all the index numbers, while using the enable bit to decide whether to play the output frequency. Therefore, as a separate part, we have provided on the same module a 3 bit counter with count enable and overflow (zero) detection.

Pinout for the Note Interpreter:

Pin Number	Signal Direction	Signal Serialization	Description
11	Input	Serialized, 3 bits.	Index number
1,2	Output	Single Bit	Note enabled? 1=enabled
3,4,5	Output	Serialized, 13 bits	Frequency in Hertz. Low two bits are fractional.
15	Input	Single Bit	Count Enable for independent counter: 1=count on next clock.
6,7	Output	Serialized, 3 bits	Count value for counter
8,9	Output	Single Bit	Count overflow = 1 when value = 0
10	Output	Unspecified	Volume Control; connect to Superposition Processor below.

How to operate the Note Interpreter: All 8 lab kit buttons are a piano with the buttons playing, in order, C,D,E,F,G,A,B,C. However, if you hold down the two rightmost buttons (B and C) for a couple of seconds, the piano stops playing. After you release the two buttons, the eight buttons now correspond to prerecorded test patterns and songs you can play. The rightmost button brings you back to piano mode.

You will later receive more information on the specific test patterns available. Although some of them will play a song, not all of them will play real notes; for example, one test button will play 440 Hz and 441 Hz together, so that you can hear a 1 Hertz beating effect.

The note interpreter will display the notes being played using the LEDs.

Waveform ROM

This ROM stores a digitized image of a sine wave. There are 256 = 2^8 locations in this ROM, and each value is an 8 bit signed number. The 256 locations store one complete period of a sine wave. This part is pretty much constructed for you, so here is the pinout. **This ROM is provided on a normal Electric LEGO module. The special control ROM module is not used for this lab:**

Pin Number	Signal Direction	Signal Serialization	Description
11	Input	8 bits	Address
1,2,3	Output	8 bits	Data
12	Input	3 bits	Instrument Select (ignore; reserved for use in future 6.004 terms)

The ROM will have a single LED lit to identify it, and other LEDs will flash during access.

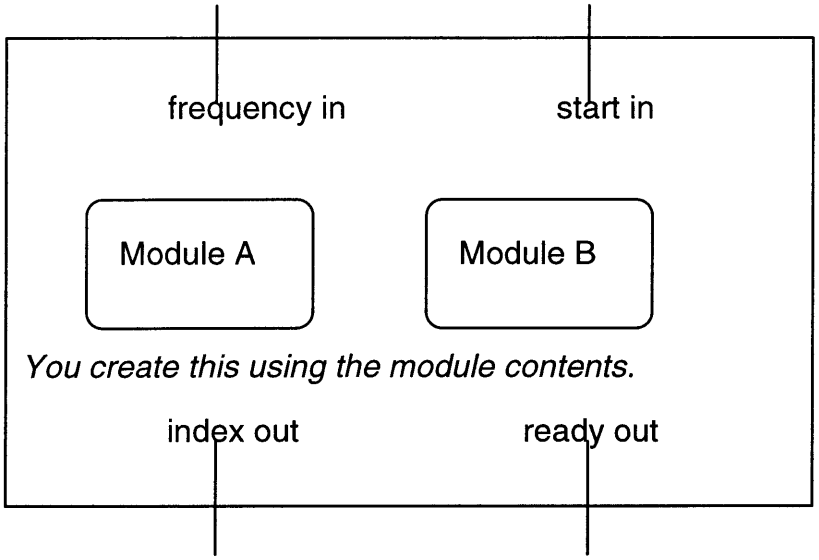
Frequency Indexing Counter

The purpose of this part is to create a counting number that increases at a rate of speed proportional to the frequency of the note being played. In order to do this, the device will need to **multiply the value of the frequency by the output of a free-running counter.** A *free-running counter* is a counter that keeps counting at a regular rate; it is always clocked and count enabled, and it is never loaded or reset. When it reaches its maximum value, it simply rolls over to zero.

The final unit you construct should have a **frequency** input, an **index** output, a **start** control input, and a **ready** status output. You construct them from simpler elements we give you, contained in two modules, A and B, as pictured below.

Frequency in	Frequency of note to be played
Index out	Index into the Waveform ROM
Start in	Goes high when data is ready to be multiplied
Ready out	Goes high when multiplier has finished and produced result.

Constructing the Frequency Indexing Counter from given elements:



Here are useful parameters. If you don t get all this information at first, don t worry. Do your best to understand how these were derived, although we ve already done the work to make them self-consistent and give you pre-made parts based on them:

- Approximate minimum output frequency: 128 Hertz
- Approximate maximum output frequency: 2047 Hertz
- Desired sampling rate of audio output: $16384=2^{14}$ samples/second (well above Nyquist rate; determined based on maximum output frequency)
- Number of samples in ROM: 256 samples/period (This was determined based on how many samples we d need for the minimum output frequency, considering the desired sampling rate. It s also a nice round number, digitally speaking. ☺)
- Format of note frequency: bbbbbbbbbbb.bb (13 bits, with 2 fractional bits; number of bits based on maximum output frequency. For our purposes, this specification well exceeds the ability of the ear to distinguish pitches, or the machine s output stage to reproduce them. ☺)
- Rate of free-running counter: $16384=2^{14}$ counts/second (equal to output sample rate)
- Number of bits in free-running counter: 16. (based on total number of bits needed for the ROM index, including fractional index bits that are thrown away; see multiplication example below.) This means that the counter rolls over once every four seconds.

- Fractional bits in free-running counter: 6 (calculated by comparing the rate at which the counter counts with the number of samples per period in the ROM. [Samples/second]/[samples/period] = unitless ratio relating counting rate of counter to counting rate required for ROM index.)

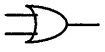
Therefore, **here is a picture of the required multiplication**. Each ☺ represents a bit; each ☹ represents an unused result bit: *All numbers are unsigned.*

☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺.☺ ☺ ☺ ☺ ☺ ☺ *(free-running counter)*
x ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺.☺ ☺ *(frequency)*
... ☹ ☹ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺.☹ ☹ ☹ ☹ ☹ ☹ ☹ ☹ *(resulting index into ROM)*
| *utilize 8 low nonfractional bits* |

In other words, when you multiply a number with 6 fractional bits by one with 2 fractional bits, you get a number with 8 fractional bits. You then ignore the resulting fractional bits, because addresses to ROMs must be integers, and you take the 8 low nonfractional bits, because the ROM only has 256 samples in it.

How do you multiply? One way is to do it the same way you multiply by hand. You start by examining the low bit of one of the numbers (say the bottom number). If the bit is a 1, you add to the subtotal the other (top) number. Then you access the next higher bit of the bottom number, and shift the top number left. You keep shifting the numbers and adding a subtotal if the low bit of the bottom number is 1, until you run out of 1 bits for either number. In this example, only the low bits of the result of the multiplication are desired, so you can stop once the numbers to be multiplied get shifted out of this range. You might want to *draw a picture of this process*, or talk about it with a friend, and we leave you to do so.

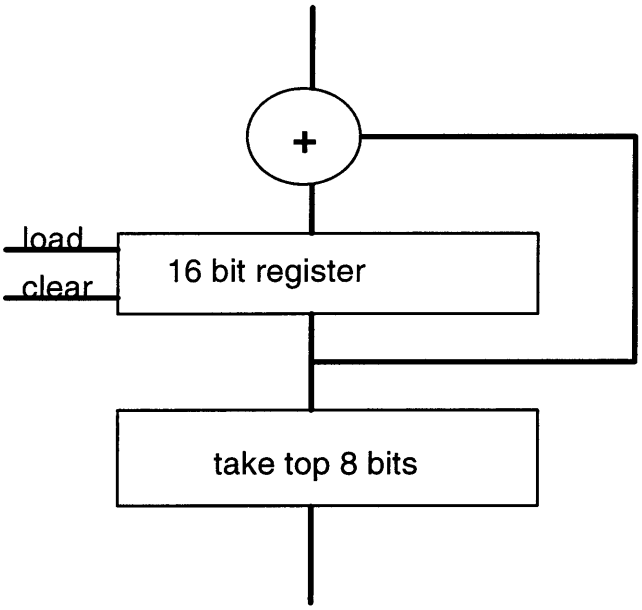
You will need to construct a multiplying machine to do this. The multiplying machine should have two data inputs. It should also have a one-bit input called **start** that goes HI for one clock cycle to indicate that the data is ready to be multiplied. It should have a one-bit output called **ready** to indicate that the multiplication has completed and the result is correct. **Ready** should be LO during the multiplication, and the multiplier should ignore further pulses of **Start** until the multiplication has completed. We give you the following parts:

- Free-running counter, 16 bits of output, rolls over every 4 seconds.
- One two-input single-bit OR gate: 
- One two-input single-bit AND gate. 
- 16 bit left shift register with load enable, value output, and zero flag output (returns 1 when all bits of contents of register are zero) Shifts in 0 bits on the right side. Shifts when not load enabled --- load enable = 1 to load, 0 to shift.

- 13 bit right shift register with load enable, *low bit* output (of the currently stored value), and zero flag output (returns 1 when all bits of contents of register are zero) Shifts in 0 bits on the left side. Shifts when not load enabled --- load enable = 1 to load, 0 to shift.

- 16 bit adder/accumulator unit, with load enable and clear enable. Receives 16 bits of input, but outputs high 8 bits of the accumulator:

Adder/accumulator for the multiplication process



Because there are many parts to give you, we are spitting this part up over two physical modules. The gates, the counter, and the right shifter will be on **Counter Module A.** The left shifter and the accumulator will be on **Counter Module B.** Here are the pinouts:

Counter Module A: LEDs will flash based on the outputs; a more detailed spec. is forthcoming.

Pin Number	Signal Direction	Signal Serialization	Description
1,2	Out	16 bits	Counter Output
11	In	Single bit	AND input 1
12	In	Single bit	AND input 2
3,4,5	Out	Single bit	AND output
13	In	Single bit	OR input 1

14	In	Single bit	OR input 2
6,7,8	Out	Single bit	OR output
15	In	13 bits	Right shift Data in
16	In	Single bit	Right shift Load enable
9	Out	Single bit	Right shift Low Bit out
10	Out	Single bit	Right shift Zero flag (1=all zeros)

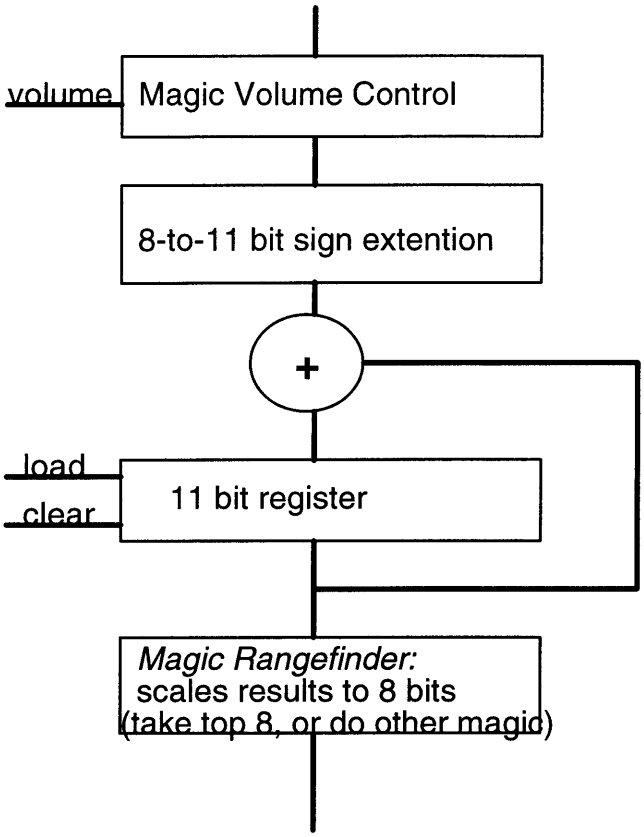
Counter Module B: LEDs will flash based on the outputs; a more detailed spec. is forthcoming.

Pin Number	Signal Direction	Signal Serialization	Description
11	In	16 bits	Left shift Data in
12	In	Single bit	Left shift Load enable
1,2	Out	16 bits	Left shift Data out
3,4	Out	Single bit	Left shift Zero flag (1=all zeros)
14	In	16 bits	Accumulator Data in
15	In	Single bit	Accumulator Load enable
16	In	Single bit	Accumulator Clear enable
5,6,7	Out	8 bits	Accumulator Data out

Remember: you are not allowed to plug more than one wire into any hole on your kit. Each wire connects one input and one output. We have provided everything you need. Think about the behavior of the signals, and discuss this problem with your teammates. It might be a little tricky, but you ll get it. ☺

Superposition Processor

This part sums up the waveforms of all the notes that are currently being played, scales each note based on the volume specified by the Note Interpreter, and scales the sum so as not to exceed the range of the D/A converter.. Originally, we were going to make you build this part yourself, but we decided to be nice and built it for you. So be sure you understand it so you can explain it to us:

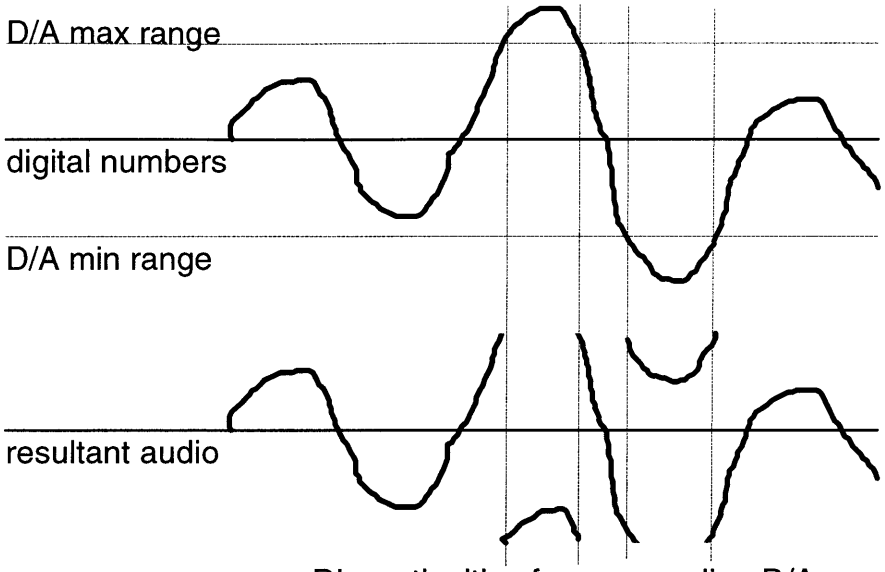


Pinout for the Superposition Processor:

Pin Number	Signal Direction	Signal Serialization	Description
11	In	8 bits	Data In
12	In	Single Bit	Load Enable
13	In	Single Bit	Clear Enable
14	In	Not Specified	Volume Control (from Note Interpreter)
1,2,3	Out	8 bits	Data Out

The LEDs on the Superposition Processor will flash to indicate activity (specification forthcoming)

The Superposition Processor sums up multiple waves, and depending on how many waves are being summed, the peak (maximum point of the wave) could be at different levels. If the peak of the wave exceeds the maximum range of the D/A converter chip, there would be a discontinuity in the final signal it produces:



Therefore, the superposition module has an automatic rangefinding circuit that scales the output values according to the peak amplitude. A simple way to do this is to take the top 8 out of 11 bits, although there are fancier ways.

Digital to Analog converter; output stage

These parts are pre-built. The D/A module contains a register with load enable, and a *nonserialized* output. Notice a colored ribbon cable connecting one of the modules to the protostrip. This transfers the value in the register to the D/A converter chip. You can leave all this wiring alone, and just use two of the pins on the module: the D/A data, and the D/A load enable:

Pin Number	Signal Direction	Signal Serialization	Description
20	Input	8 Bit serialized	D/A data
19	Input	Single bit	Load Enable (on next clock)
3-18	Unspecified	Single Bits	To D/A chip via ribbon cable

LEDs on the output stage will flash to indicate the magnitude of the signal being processed.

Control FSM

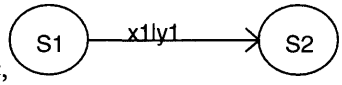
Ah the moment we ve been waiting for. **The control FSM loops through all the notes that want to be played. For each note, it decides whether the note should be played; if so, it plays the note. When finished, it outputs the resulting audio level.** Don t be fooled this is not a trivial FSM! However, it is not very big.

We ve used a Mealy implementation, because this allows you to choose different possible outputs for each state, thus reducing the total number of states.

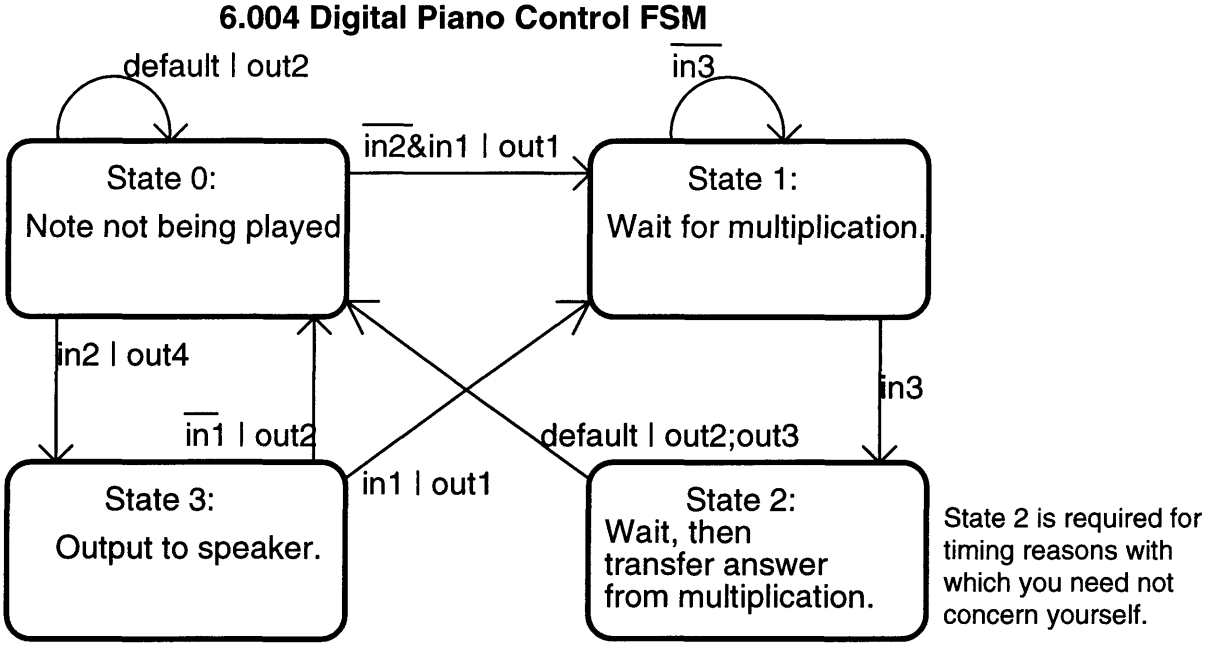
We are providing you with a pre-built FSM, pictured below. You just need to figure out which signals go where. Match up the generic In{1,2,3} and Out{1,2,3,4} with real signals from the datapath you built. You might start by making a list of all the signals that should be connected to the FSM.

Remember the notation for a transition arc:

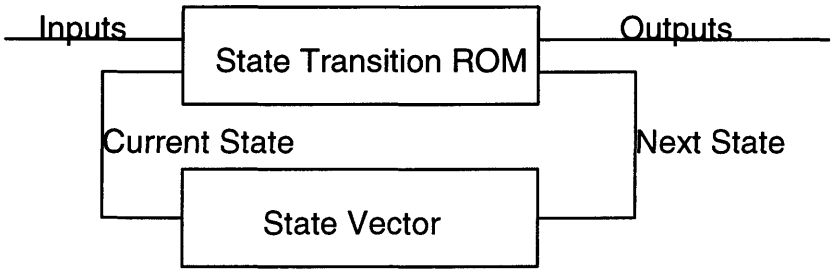
conditions to take this arc | outputs activated in the state ***behind*** *the arc,* ***before*** *it is taken.*

For example, in this arc, 

the FSM will output y1 while in state S1, and then transition to S2, all if x1 is active. (This excerpt does not specify what S2 will output.)



Here is the **hardware implementation** of the FSM:



Pinout for the Control FSM; **this part is provided on a normal Electric LEGO module. The special control ROM module, in the lower left corner of the kit, is not used for this lab:**

Pin Number	Signal Direction	Signal Serialization	Description
11	In	Single Bit	Input 1
12	In	Single Bit	Input 2
13	In	Single Bit	Input 3
1,2	Out	Single Bit	Output 1
3,4	Out	Single Bit	Output 2
5,6	Out	Single Bit	Output 3
7,8	Out	Single Bit	Output 4
9	Out	Serialized, 4 bits	Current State (testing purposes only)
10	Out	Serialized, 4 bits	Next State (testing purposes only)

The LEDs of the FSM will indicate the current state. Other LEDs will flash in case of an unexpected input.

Testing Module

As you construct the system, you may wish to check how each part of your system is doing. There is a component on your lab kit that provides signals you can use to test your multiplier circuit. Let us know (6004-labs@ai.mit.edu) if you have any ideas for additional signals or devices that we should provide to allow you to test other modules.

The testing module is in the upper-left corner of the kit, because it uses a special module with an alphanumeric display instead of just a bar graph LED.

If you connect the multiplication signals to the multiplier, and view the multiplication result on the alphanumeric LEDs, you see a test pattern scroll by in a marquee style.

Pinout for the Testing Module:

Pin Number	Signal Direction	Signal Serialization	Description
11	In	Serialized, any number of bits.	Displays input value on special alphanumeric LEDs.
12	In	Single bit	Ready signal in from multiplier

1,2	Out	Single bit	Output start signal to multiplier
3,4	Out	16 bits	16 bit number to multiply
5,6	Out	13 bits	13 bit number to multiply

PUTTING IT ALL TOGETHER: WRITING A DESIGN PROPOSAL

Because of limited lab kit availability, and because we want to encourage you to plan your work carefully, we require a simple **written design proposal** that will be **reviewed in person** with a staff member. The proposal should include:

- Complete wiring diagram, with labeled wires. This can be done in schematic form or in a way that corresponds to how the modules are physically laid out on the kit. Most importantly, it must be **clear** and **readable.** Include **pin numbers** on both ends of every wire.
- Short paragraph explaining the high-level workings of the datapath.
- Wiring connections for FSM and brief explanation of what it is doing.
- Wiring diagram for the multiplier can be included in the master-wiring diagram, but its functionality should be specifically explained in a short paragraph.
- Verbal presentation of all these materials to a staff member, by the individual team members who created each part.

This proposal may be done by hand or machine. You may use any size paper or format you wish. Your team will keep these materials. Do not waste time to make it unnecessarily fancy or elegant.

Staff members will only check you off if, to the best of their ability, they believe that your design will work. Otherwise, you will be sent back to do revisions.

IMPLEMENTATION ON THE 6.004 LAB KIT

You must have an approved design proposal with circuit diagram and wiring information before you will be allowed to implement your design on a lab kit.

Please come on time for your appointment to use the lab kit. Many students need to use them. With all your teammembers at the appointment, implementation can proceed in parallel. The datapath and multiplier can be wired up simultaneously; then the FSM can be connected to the rest of the system. If the system does not work, a staff member can help you test it piece-by-piece. Good luck!

The power to the kit should be kept off while adding or changing wires. Before turning the power on, please check that each wire goes from an input (upper row of pins) to an output (lower row of pins). Specifically, outputs should not be connected to other outputs.

Once you think you have it working, try some of the preprogrammed songs (directions forthcoming). You should also try the 440-441 beat test, in which two sounds of 440 and 441 Hertz are made together. You should hear 1 Hertz beating (apparent swells in volume every second) that you learn more about in 6.003.

QUESTIONS

As you work on this project, you may wish to ponder the following issues. These are open-ended questions, and some of them are very hard. You are **not** required to provide complete answers.

- In general, an FSM will not work if an output is dependent on an input that will immediately change if the output changes. In other words, your Mealy FSM must not have a combinational cycle from an output, through the datapath, back to the input.

- Do a timing analysis for the system, considering the propagation delays for the serializers (information forthcoming). What is the limiting factor in your piano performing? The number of notes you play at a time? The frequencies of those notes? Both?

- If the machine starts running out of time to make its computations, how will it fail? Does the design minimize the visible effects of this failure?

- Draw a state transition diagram for your serial multiplier. Yes, you can do it! ☺ You can think of its different modes of operation as states.

- View the serialized signals on an oscilloscope. There might be one somewhere in the lab on a cart. ☺ View the FSM state.

- Consider alternatives to the design we presented. Give us feedback on what components we should provide on the kit to give students more options.

CLEANING UP YOUR LAB KIT

After you demonstrate your project to the staff (and your friends), please remove all circuitry you constructed. You should leave the wiring that supports the D/A functionality and the lab kit buttons (the ribbon cables and the wiring on the protostrip).

FUTURE UPDATES

The following information will further assist you and is forthcoming:

- Possible use of Mike Wessler s Betasim software for this lab. (for now, just concern yourself with getting your design and lab kit wiring plans ready)
- How to play preprogrammed songs.
- Exactly how the LEDs appear under different circumstances.
- Updates on the availability of the lab kits.
- Additional precautions for using the lab kit.
- Timing information.
- A schematic diagram for the analog audio output and amplification portion of the circuit, for your reference.
- Additional instructions on using the JTerm software to assign components to physical locations on the kit.
- Instructions on single-stepping your machine and FSM.
- Additional things to try/test/observe when you get your kit working.
- How to fill out the feedback form for this lab, after you are finished.

Good Luck! ☺

WHAT WE HOPE YOU LEARNED

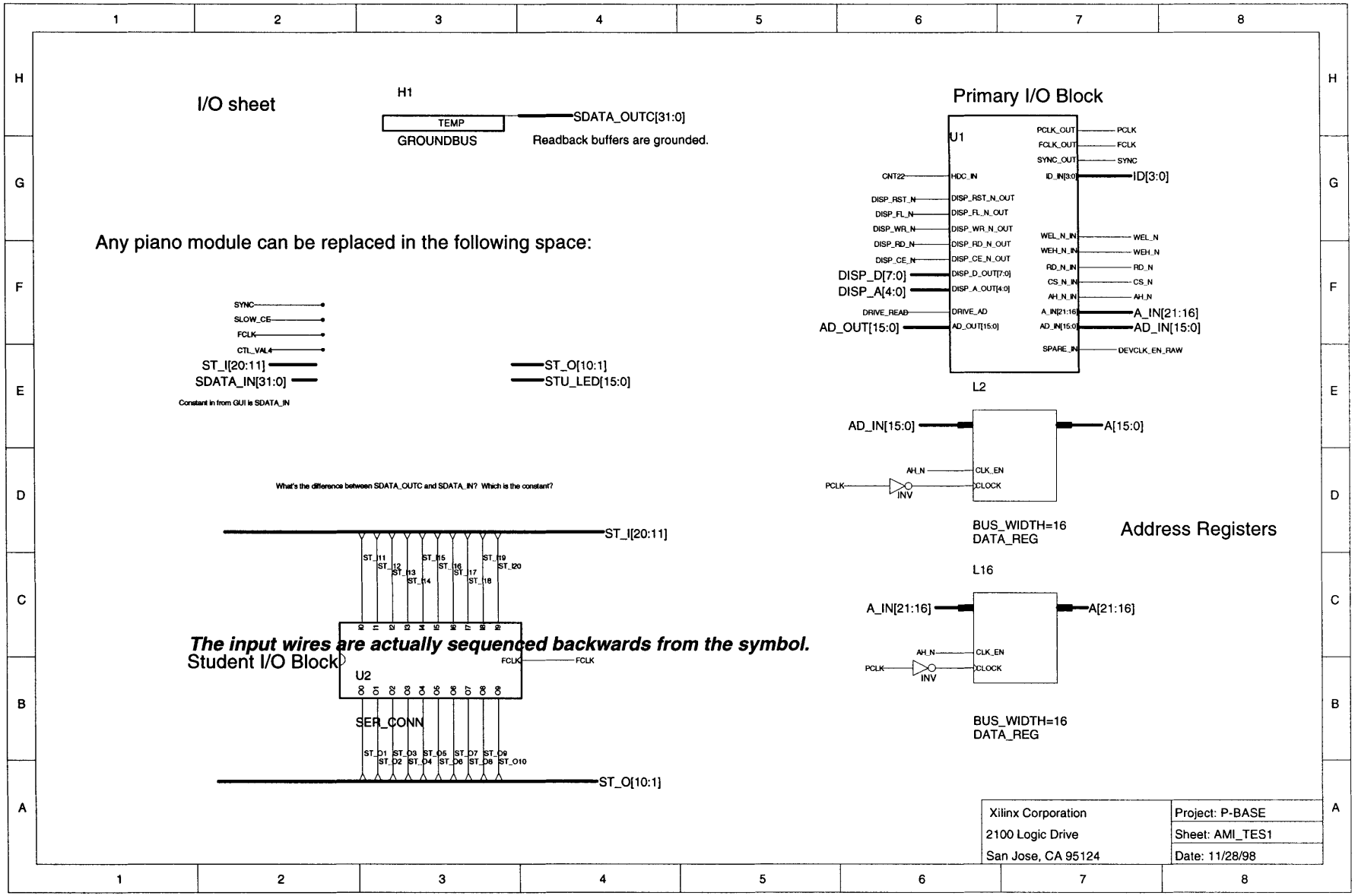
This project gave you experience with some of the components that make up a digital computation system. We hope your experience with this example enlightens you to various datapath design techniques.

We can design a special-purpose digital computation system with datapaths optimized for the one thing it can do. Researchers are currently investigating the idea of *reconfigurable computers.* Just as your lab kit can be reprogrammed by the staff to give you a different parts box for a new assignment, and you can rewire the available parts to create your project, reconfigurable computers use FPGAs with supporting compilers and programming circuitry. A specialized circuit is temporarily constructed to compute the algorithm you specify when you execute your program.

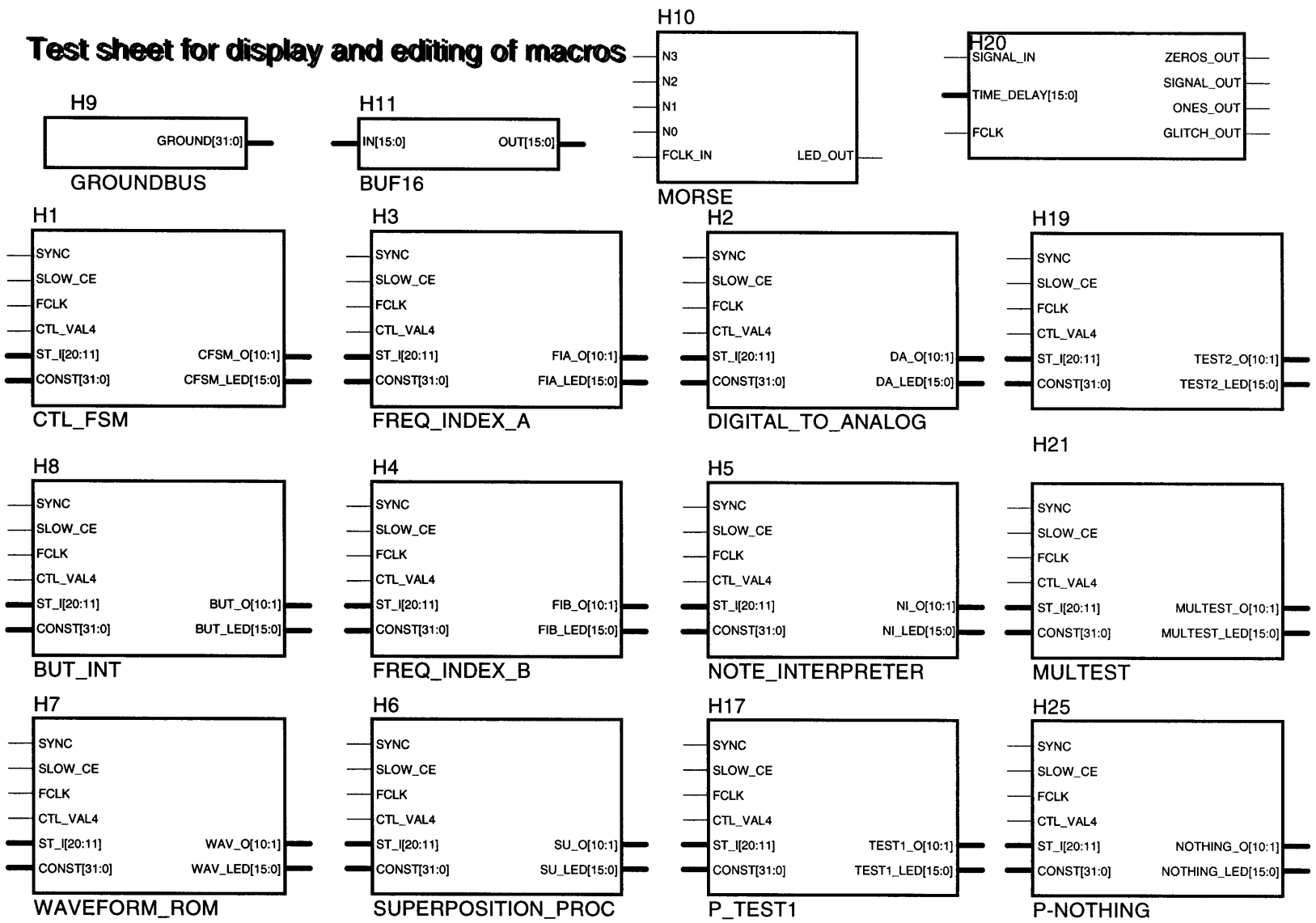
Appendix B:
Digital Piano FPGA Module Schematics

The pre-designed FPGA modules are pictured here. Note that these schematics are a tool by which the user of the Xilinx software specifies the design. They have nothing to do with the physical internal layout of the circuits implemented in FPGA hardware — that is specified by the compilation software.

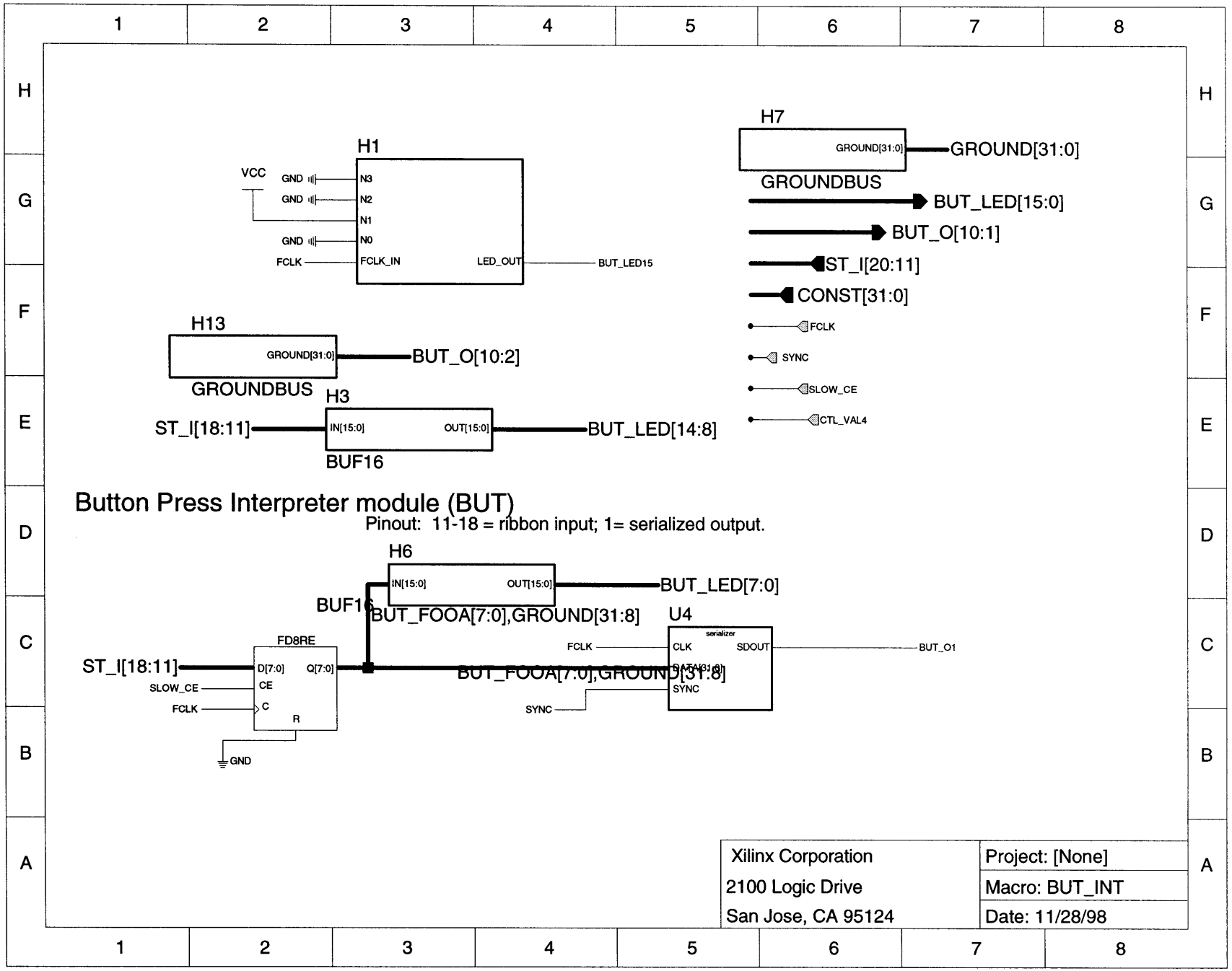
Since there were several FPGAs with some similar properties, a template was first created with a space to insert the specific logic specifications of each of the desired modules. The template is first illustrated. These specifications were then created in one large project, illustrated next. The remaining illustrations represent the designs of specific modules.


I/O sheet
H1
TEMP
GROUNDBUS
SDATA_OUTC[31:0]
Readback buffers are grounded.
Primary I/O Block
U1
PCLK_OUT
FCLK_OUT
SYNC_OUT
ID_IN[3:0]
CNT22
HDC_IN
DISP_RST_N
DISP_FL_N
DISP_WR_N
DISP_RD_N
DISP_CE_N
DISP_D[7:0]
DISP_A[4:0]
DRIVE_READ
AD_OUT[15:0]
DISP_RST_N_OUT
DISP_FL_N_OUT
DISP_WR_N_OUT
DISP_RD_N_OUT
DISP_CE_N_OUT
DISP_D_OUT[7:0]
DISP_A_OUT[4:0]
DRIVE_AD
AD_OUT[15:0]
WEL_N_IN
WEH_N_IN
RD_N_IN
CS_N_IN
AH_N_IN
A_IN[21:16]
AD_IN[15:0]
SPARE_IN
PCLK
FCLK
SYNC
ID[3:0]
WEL_N
WEH_N
RD_N
CS_N
AH_N
A_IN[21:16]
AD_IN[15:0]
DEVCLK_EN_RAW
Any piano module can be replaced in the following space:
SYNC
SLOW_CE
FCLK
CTL_VAL4
ST_I[20:11]
SDATA_IN[31:0]
Constant in from GUI is SDATA_IN
ST_O[10:1]
STU_LED[15:0]
L2
AD_IN[15:0]
A[15:0]
AH_N
PCLK
INV
CLK_EN
CLOCK
BUS_WIDTH=16
DATA_REG
Address Registers
L16
A_IN[21:16]
A[21:16]
BUS_WIDTH=16
DATA_REG
What's the difference between SDATA_OUTC and SDATA_IN? Which is the constant?
ST_I[20:11]
The input wires are actually sequenced backwards from the symbol.
Student I/O Block
U2
FCLK
SER_CONN
ST_O[10:1]
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: P-BASE
Sheet: AMI_TES1
Date: 11/28/98

Test sheet for display and editing of macros

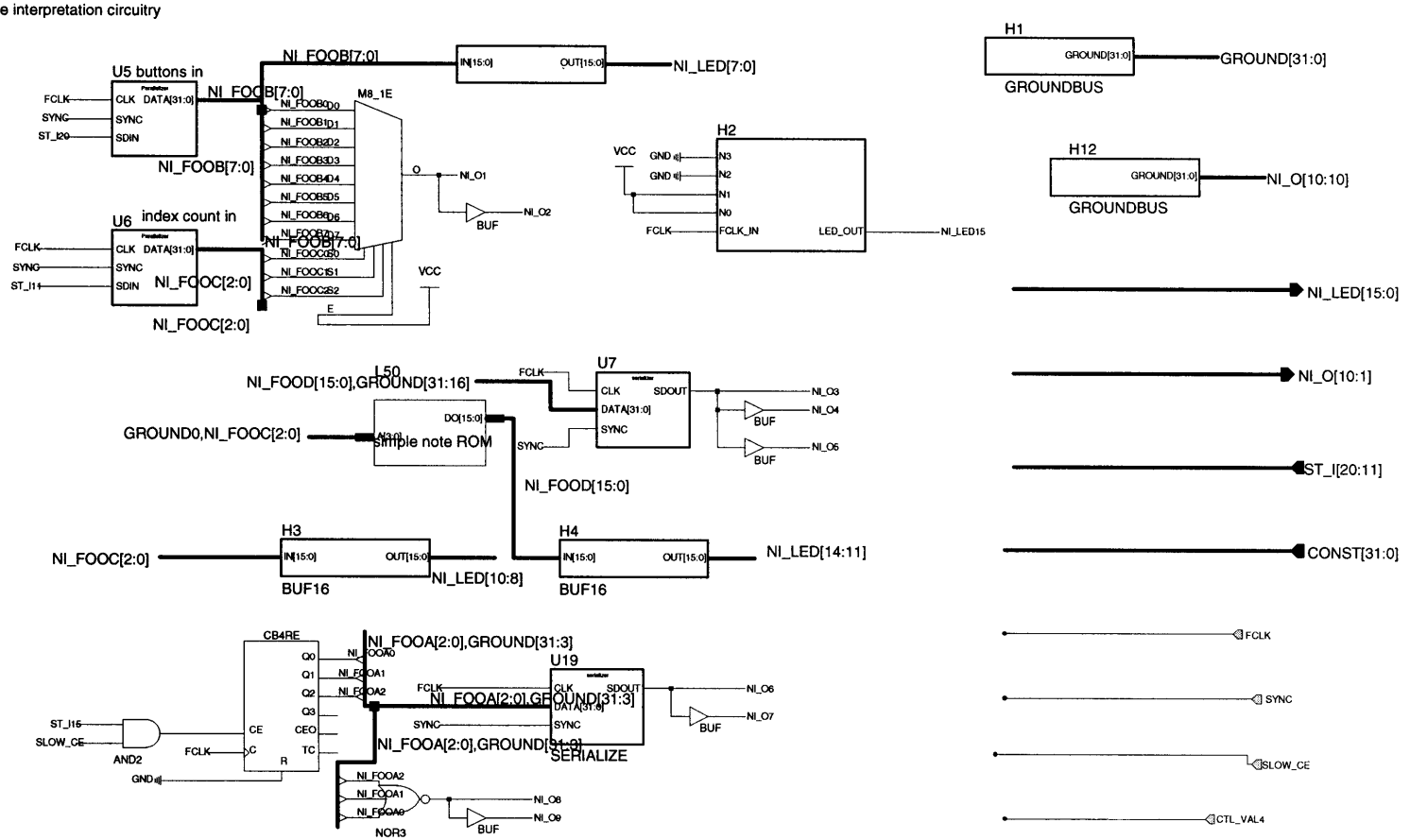


Xilinx Corporation	Project: P-MACROS
2100 Logic Drive	Sheet: P-MACRO4
San Jose, CA 95124	Date: 11/28/98

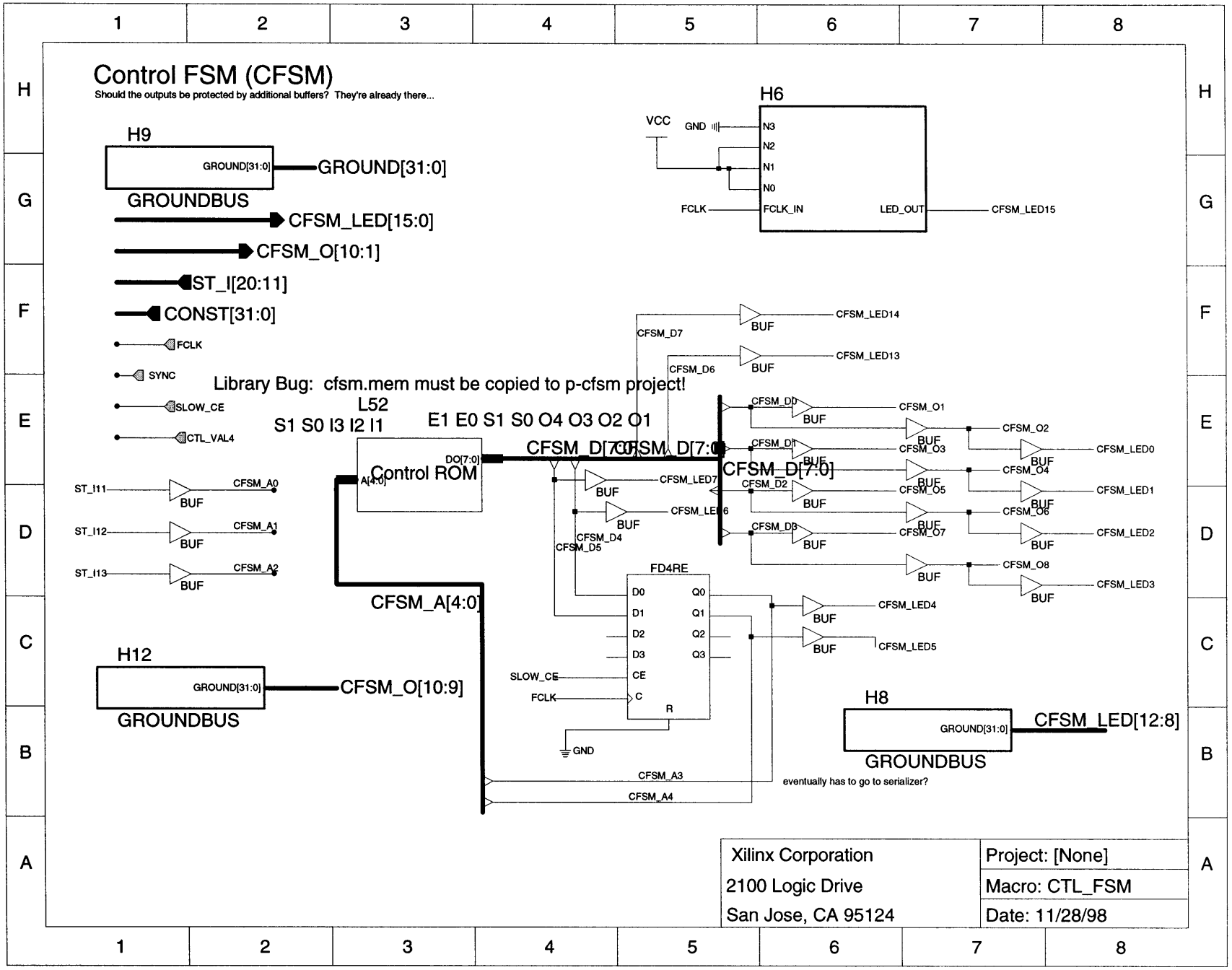

H1
VCC
GND
GND
GND
FCLK
N3
N2
N1
N0
FCLK_IN
LED_OUT
BUT_LED15
H7
GROUND[31:0]
GROUND[31:0]
GROUNDBUS
BUT_LED[15:0]
BUT_O[10:1]
ST_I[20:11]
CONST[31:0]
FCLK
SYNC
SLOW_CE
CTL_VAL4
H13
GROUND[31:0]
BUT_O[10:2]
GROUNDBUS
H3
ST_I[18:11]
IN[15:0]
OUT[15:0]
BUT_LED[14:8]
BUF16
Button Press Interpreter module (BUT)
Pinout: 11-18 = ribbon input; 1= serialized output.
H6
IN[15:0]
OUT[15:0]
BUT_LED[7:0]
BUF16
BUT_FOOA[7:0],GROUND[31:8]
U4
serializer
FCLK
CLK
SDOUT
BUT_O1
DATA[31:0]
SYNC
BUT_FOOA[7:0],GROUND[31:8]
SYNC
FD8RE
ST_I[18:11]
D[7:0]
Q[7:0]
SLOW_CE
CE
FCLK
C
R
GND
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: [None]
Macro: BUT_INT
Date: 11/28/98

Note Interpreter Module (NI)

Note interpretation circuitry

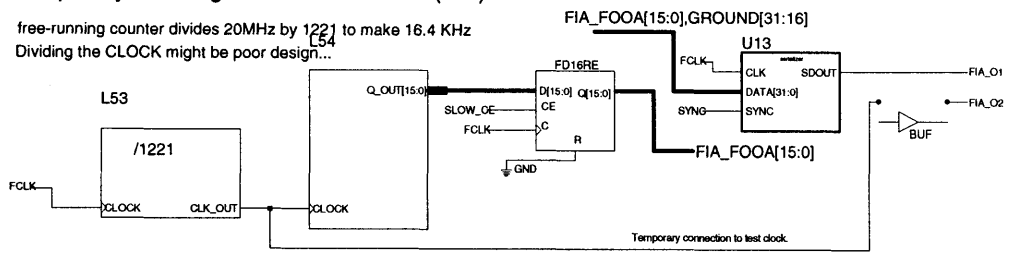


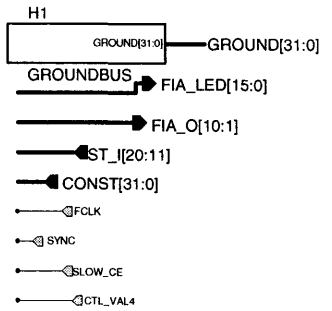
Xilinx Corporation	Project: [None]
2100 Logic Drive	Macro: NOTE_INTERPRETER
San Jose, CA 95124	Date: 11/28/98

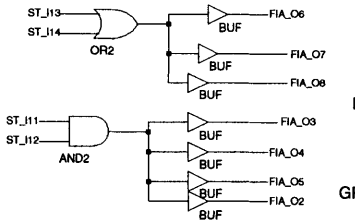

Control FSM (CFSM)
Should the outputs be protected by additional buffers? They're already there...
H9
GROUND[31:0]
GROUNDBUS
CFSM_LED[15:0]
CFSM_O[10:1]
ST_I[20:11]
CONST[31:0]
FCLK
SYNC
SLOW_CE
CTL_VAL4
H6
VCC
GND
N3
N2
N1
N0
FCLK
FCLK_IN
LED_OUT
CFSM_LED15
Library Bug: cfsm.mem must be copied to p-cfsm project!
L52
S1 S0 I3 I2 I1
E1 E0 S1 S0 O4 O3 O2 O1
Control ROM
DO[7:0]
A[4:0]
CFSM_D[7:0]
ST_I11
ST_I12
ST_I13
BUF
CFSM_A0
CFSM_A1
CFSM_A2
CFSM_A[4:0]
FD4RE
D0
D1
D2
D3
CE
C
R
Q0
Q1
Q2
Q3
SLOW_CE
FCLK
GND
CFSM_D7
CFSM_D6
CFSM_D5
CFSM_D4
CFSM_LED14
CFSM_LED13
CFSM_LED7
CFSM_LED6
CFSM_LED4
CFSM_LED5
CFSM_O1
CFSM_O2
CFSM_O3
CFSM_O4
CFSM_O5
CFSM_O6
CFSM_O7
CFSM_O8
CFSM_LED0
CFSM_LED1
CFSM_LED2
CFSM_LED3
H12
GROUND[31:0]
GROUNDBUS
CFSM_O[10:9]
H8
GROUND[31:0]
GROUNDBUS
CFSM_LED[12:8]
CFSM_A3
CFSM_A4
eventually has to go to serializer?
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: [None]
Macro: CTL_FSM
Date: 11/28/98

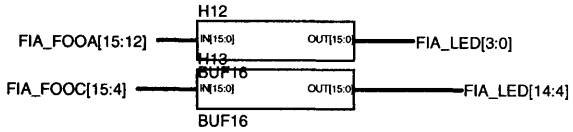
Frequency Indexing Counter Module A (FIA)

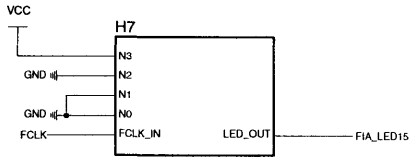
free-running counter divides 20MHz by 1221 to make 16.4 KHz
Dividing the CLOCK might be poor design...

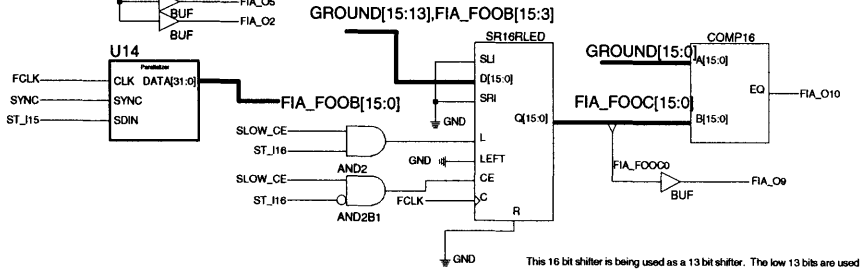


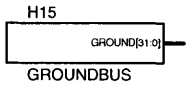






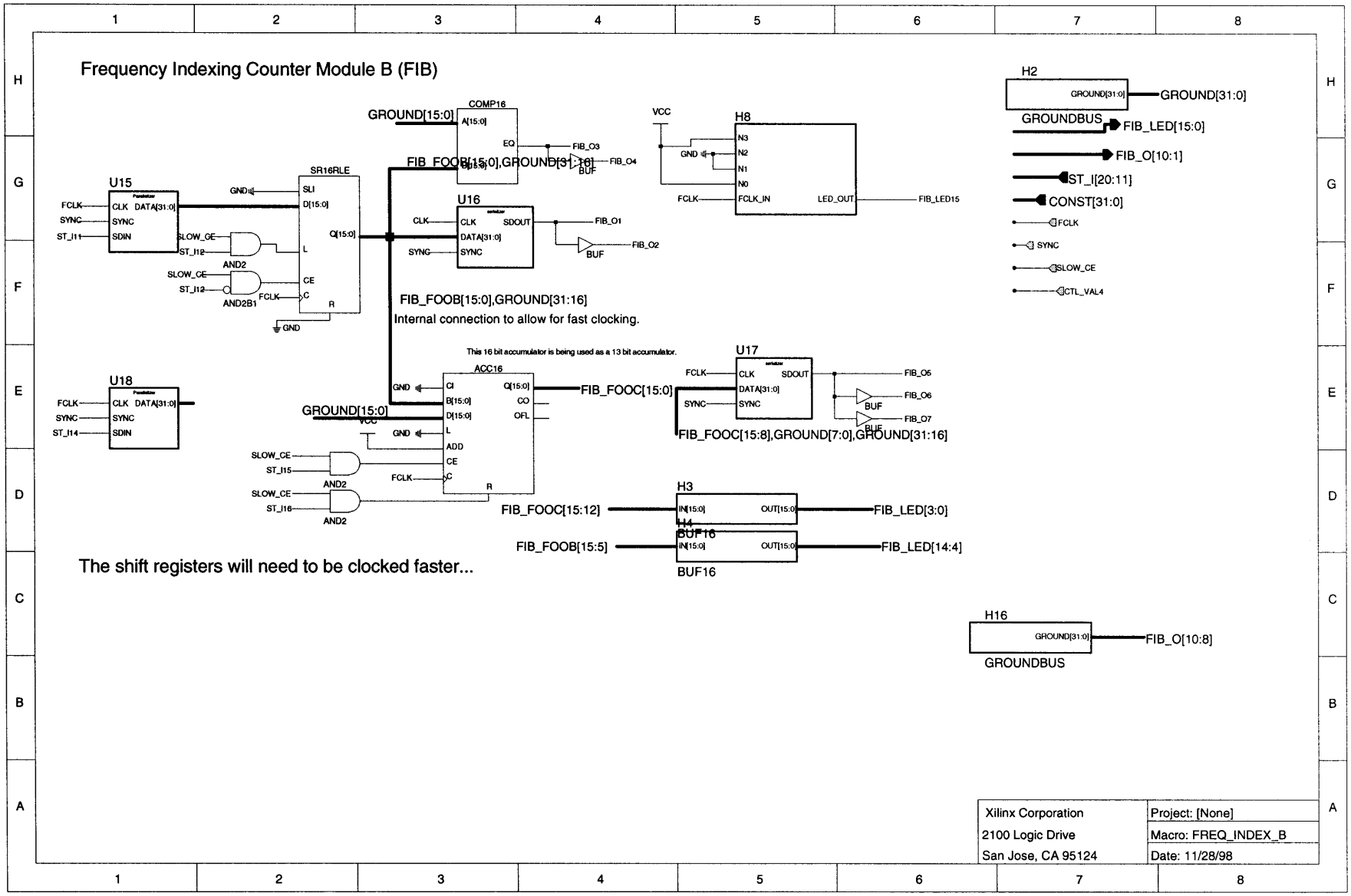


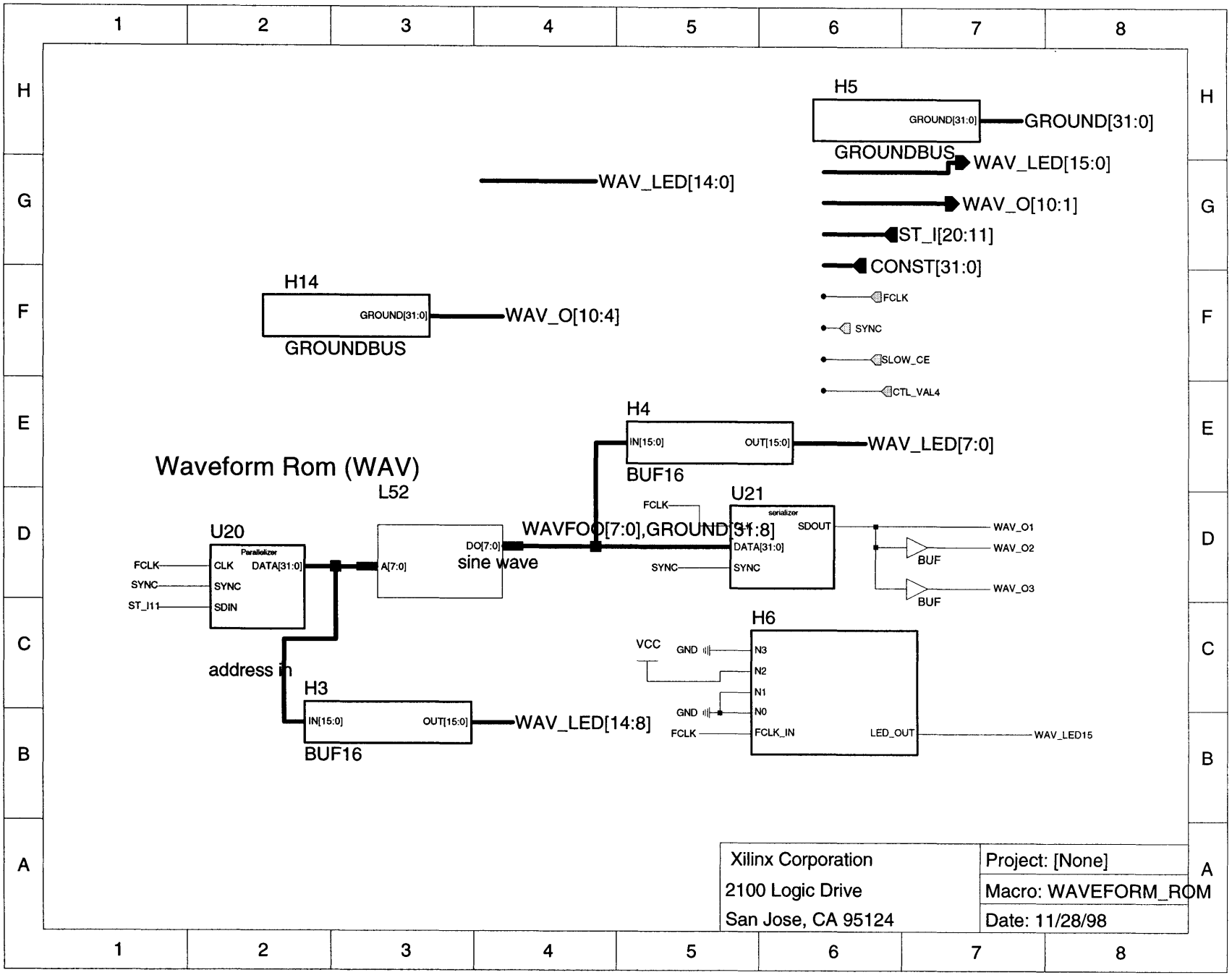


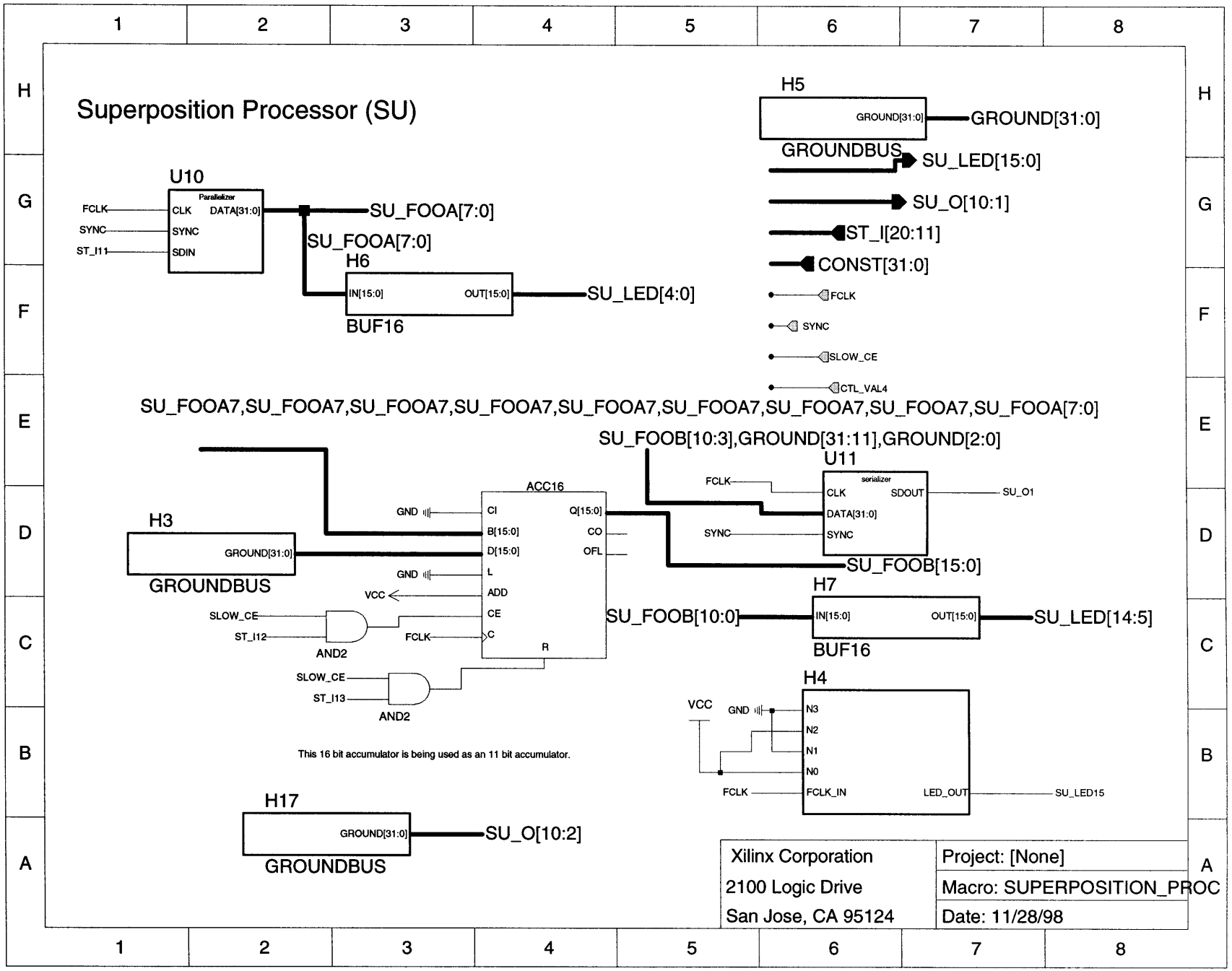


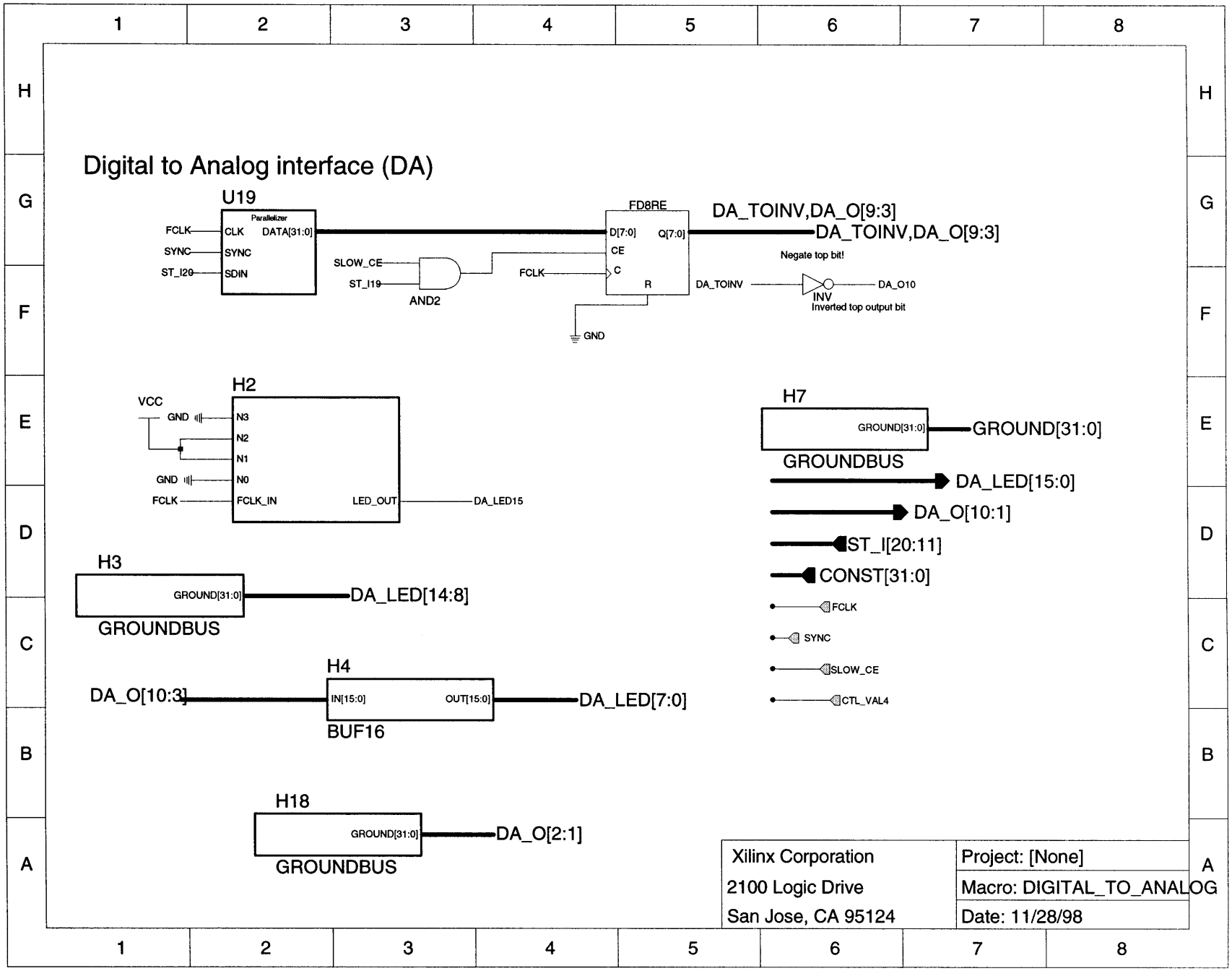
The shift registers will need to be clocked faster...

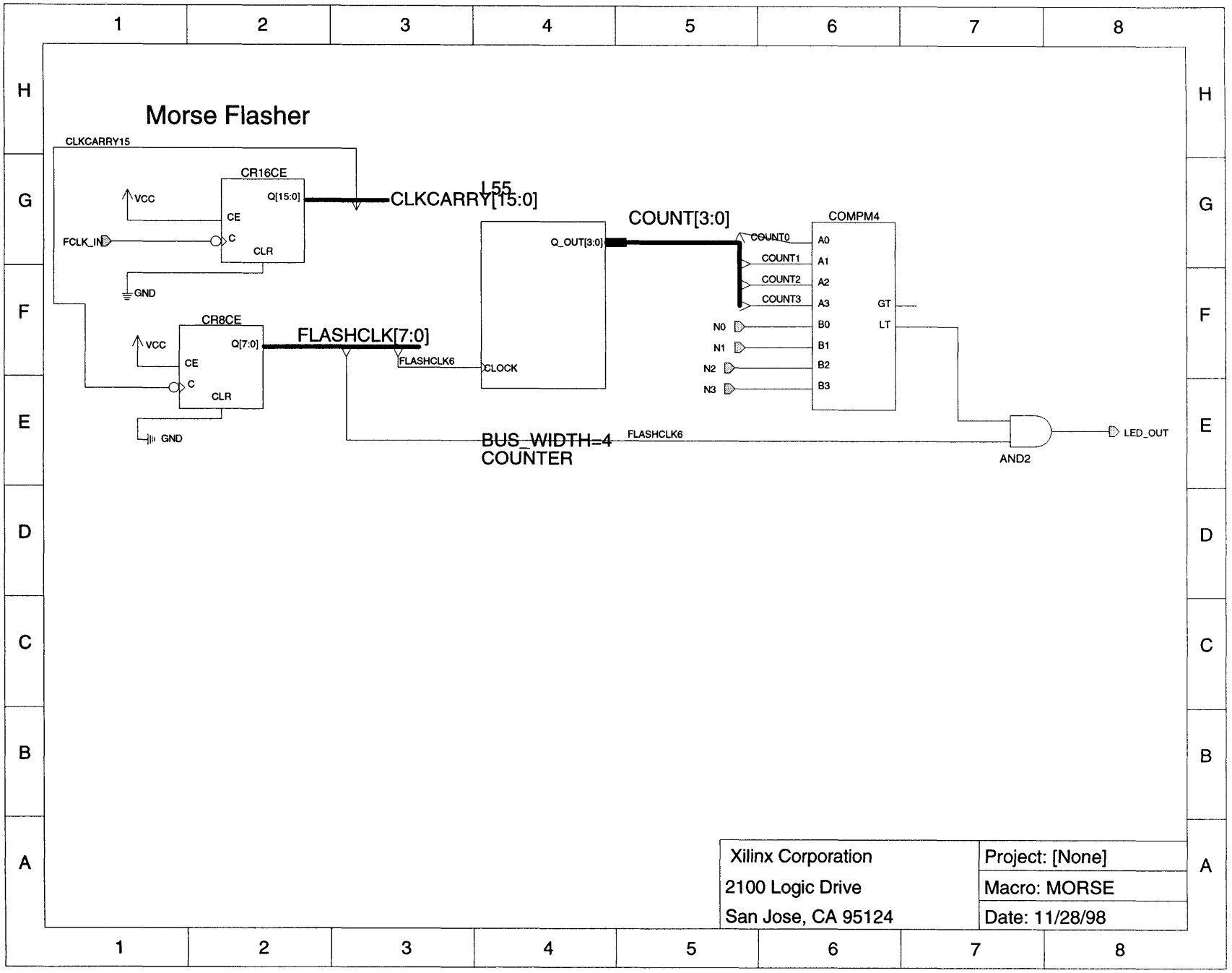
Xilinx Corporation	Project: [None]
2100 Logic Drive	Macro: FREQ_INDEX_A
San Jose, CA 95124	Date: 11/28/98


Frequency Indexing Counter Module B (FIB)
GROUND[15:0]
COMP16
A[15:0]
EQ
FIB_O3
FIB_FOOB[15:0],GROUND[31:16]
FIB_O4
BUF
VCC
H8
N3
N2
N1
N0
GND
FCLK
FCLK_IN
LED_OUT
FIB_LED15
U15
CLK DATA[31:0]
SYNC
SDIN
FCLK
SYNC
ST_I11
SLOW_CE
ST_I12
AND2
AND2B1
SR16RLE
SLI
D[15:0]
Q[15:0]
L
CE
C
R
GND
U16
CLK
SDOUT
DATA[31:0]
SYNC
FIB_O1
FIB_O2
BUF
FIB_FOOB[15:0],GROUND[31:16]
Internal connection to allow for fast clocking.
This 16 bit accumulator is being used as a 13 bit accumulator.
ACC16
CI
B[15:0]
D[15:0]
ADD
Q[15:0]
CO
OFL
FIB_FOOC[15:0]
U17
FIB_O5
FIB_O6
FIB_O7
FIB_FOOC[15:8],GROUND[7:0],GROUND[31:16]
U18
ST_I14
ST_I15
ST_I16
H3
IN[15:0]
OUT[15:0]
FIB_FOOC[15:12]
FIB_LED[3:0]
H4
BUF16
FIB_FOOB[15:5]
FIB_LED[14:4]
The shift registers will need to be clocked faster...
H2
GROUND[31:0]
GROUNDBUS
FIB_LED[15:0]
FIB_O[10:1]
ST_I[20:11]
CONST[31:0]
CTL_VAL4
H16
FIB_O[10:8]
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: [None]
Macro: FREQ_INDEX_B
Date: 11/28/98


H5
GROUND[31:0]
GROUND[31:0]
GROUNDBUS
WAV_LED[15:0]
WAV_LED[14:0]
WAV_O[10:1]
ST_I[20:11]
CONST[31:0]
FCLK
SYNC
SLOW_CE
CTL_VAL4
H14
GROUND[31:0]
WAV_O[10:4]
GROUNDBUS
H4
IN[15:0]
OUT[15:0]
WAV_LED[7:0]
BUF16
Waveform Rom (WAV)
L52
U21
serializer
U20
Parallelizer
FCLK
CLK
DATA[31:0]
SYNC
SYNC
ST_I11
SDIN
DO[7:0]
A[7:0]
sine wave
WAVFOO[7:0],GROUND[31:8]
DATA[31:0]
SDOUT
SYNC
SYNC
WAV_O1
WAV_O2
WAV_O3
BUF
BUF
H6
VCC
GND
N3
N2
N1
N0
GND
FCLK
FCLK_IN
LED_OUT
WAV_LED15
address in
H3
IN[15:0]
OUT[15:0]
WAV_LED[14:8]
BUF16
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: [None]
Macro: WAVEFORM_ROM
Date: 11/28/98


Superposition Processor (SU)
H5
GROUND[31:0]
GROUND[31:0]
GROUNDBUS
SU_LED[15:0]
SU_O[10:1]
ST_I[20:11]
CONST[31:0]
FCLK
SYNC
SLOW_CE
CTL_VAL4
U10
Parallelizer
FCLK
CLK
DATA[31:0]
SYNC
SYNC
ST_I11
SDIN
SU_FOOA[7:0]
SU_FOOA[7:0]
H6
IN[15:0]
OUT[15:0]
SU_LED[4:0]
BUF16
SU_FOOA7,SU_FOOA7,SU_FOOA7,SU_FOOA7,SU_FOOA7,SU_FOOA7,SU_FOOA7,SU_FOOA7,SU_FOOA[7:0]
SU_FOOB[10:3],GROUND[31:11],GROUND[2:0]
U11
serializer
FCLK
CLK
SDOUT
SU_O1
DATA[31:0]
SYNC
SYNC
SU_FOOB[15:0]
ACC16
H3
GND
CI
Q[15:0]
GROUND[31:0]
B[15:0]
CO
D[15:0]
OFL
GROUNDBUS
GND
L
VCC
ADD
SLOW_CE
CE
ST_I12
FCLK
C
AND2
R
SLOW_CE
ST_I13
AND2
SU_FOOB[10:0]
H7
IN[15:0]
OUT[15:0]
SU_LED[14:5]
BUF16
H4
VCC
GND
N3
N2
N1
N0
FCLK
FCLK_IN
LED_OUT
SU_LED15
This 16 bit accumulator is being used as an 11 bit accumulator.
H17
GROUND[31:0]
SU_O[10:2]
GROUNDBUS
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: [None]
Macro: SUPERPOSITION_PROC
Date: 11/28/98


Digital to Analog interface (DA)
U19
Parallelizer
FCLK
CLK
DATA[31:0]
SYNC
SYNC
ST_I20
SDIN
FD8RE
D[7:0]
Q[7:0]
CE
C
R
DA_TOINV,DA_O[9:3]
DA_TOINV,DA_O[9:3]
SLOW_CE
ST_I19
AND2
FCLK
Negate top bit!
DA_TOINV
INV
DA_O10
Inverted top output bit
GND
H2
VCC
GND
N3
N2
N1
GND
N0
FCLK
FCLK_IN
LED_OUT
DA_LED15
H7
GROUND[31:0]
GROUND[31:0]
GROUNDBUS
DA_LED[15:0]
DA_O[10:1]
ST_I[20:11]
CONST[31:0]
FCLK
SYNC
SLOW_CE
CTL_VAL4
H3
GROUND[31:0]
DA_LED[14:8]
GROUNDBUS
H4
DA_O[10:3]
IN[15:0]
OUT[15:0]
DA_LED[7:0]
BUF16
H18
GROUND[31:0]
DA_O[2:1]
GROUNDBUS
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: [None]
Macro: DIGITAL_TO_ANALOG
Date: 11/28/98


Morse Flasher
CLKCARRY15
CR16CE
VCC
CE
C
CLR
FCLK_IN
GND
Q[15:0]
CLKCARRY[15:0]
L55
Q_OUT[3:0]
COUNT[3:0]
COMPM4
COUNT0
COUNT1
COUNT2
COUNT3
A0
A1
A2
A3
GT
LT
N0
N1
N2
N3
B0
B1
B2
B3
CR8CE
VCC
CE
C
CLR
GND
Q[7:0]
FLASHCLK[7:0]
FLASHCLK6
CLOCK
BUS_WIDTH=4
COUNTER
FLASHCLK6
AND2
LED_OUT
Xilinx Corporation
2100 Logic Drive
San Jose, CA 95124
Project: [None]
Macro: MORSE
Date: 11/28/98

Bibliography

[1] Ward, Stephen A. and Robert H. Halstead, Jr. *Computation Structures*. Cambridge, MA: MIT Press, 1990.

[2] Hamblen, James O., et. al. "An Undergraduate Computer Engineering Rapid Systems Prototyping Design Laboratory." *IEEE Transactions on Education,* Volume 42, Number 1, 8-14, February 1999.

[3] Gnawali, Om Prakash. Personal interview, May, 1999.

[4] Sundquist, Andreas. Personal interview, May, 1999.

[5] Nadkarni, Vivek. Personal interview, May, 1999.