

Massachusetts Institute of Technology  
Department of Electrical Engineering and Computer Science  
6.111 - Introductory Digital Systems Laboratory  
Laboratory 3

## Helium Breath: The 6.111 Pitch Shifter

Issued: Monday, March 11, 2002  
MCU Check-off: Monday, March 18, 2002  
Design Check-off: Wednesday, March 20 2002  
Lab Check-off: Friday, April 5, 2002  
Report Due: Monday, April 8, 2002

## INTRODUCTION

This lab gives you the opportunity to design, construct, and test a reasonably complex digital system. A design review with a T.A. is to be held after you have completed the initial design portion. After the design review, you may proceed with the final design, construction, testing, demonstration, and the required lab report.

The purpose of Lab 3 is to familiarize you with microprogrammed controllers, analog to digital converters and digital to analog converters. The lab is divided into two sections, the first being the MCU and timing and the second being the I/O, storage, and processing. The first section must be designed and checked off before the rest of the lab is built. Please also pay attention to the debugging strategy which is described later in this document.

## A NOTE OF CAUTION

This problem, while not impossible, is not trivial. It is substantially more difficult, both in complexity and in scale, than the previous lab exercises. You are strongly urged to start thinking about this design task as soon as possible.

This handout provides substantial guidance for your design. However, you are NOT required to follow these suggestions exactly. You are required to implement a microprogrammed controller with Intel 28F256 (Flash Eeproms) and a simple sequencer. Implementations using an FSM controller are NOT acceptable even though they may reduce the number of ICs required. A major purpose of this lab exercise is for you to gain competence in designs using microprogrammed controllers.

You should use the microprogram assembler software to prepare the files with which to program your Intel 28F256 (Flash Eeproms). You are encouraged to use the sixteen bit microprogrammed sequencer as shown in this handout. You may use multiple PALs and CPLDs in your design.

You are also required to use VHDL to program a CPLD to perform some non-trivial function. We have suggested that you use it to address the SRAM, although you are welcome to use it for something else if such a design change makes sense. We strongly suggest you talk to a TA if you plan on making any significant changes to our specifications or suggested design.

## OVERVIEW

Your task is to build a machine which will accept an analog audio signal and produce a signal which sounds like the input signal except scaled up or down in pitch. Your machine should work

over at least a reasonable range of audio frequencies, say in the range 300 Hz–3000 Hz, which is about the range of frequencies that you can hear over the telephone. You should be able to scale the pitch of the output to anywhere between half the original pitch (one octave lower) and double the original pitch (one octave higher), and perhaps even further.

There are many ways of accomplishing this task. The algorithms which produce the best-quality results are quite complicated and more or less beyond the scope of this course. We believe you will be able to get surprisingly good results using the relatively simple algorithm we describe below.

A simple, overall block diagram is shown in Figure 1. As you can see, the system has an analog audio input and output and several other control signals which you will probably want to connect to switches on your kit. These control signals are explained in Table 1.



Figure 1: General Block Diagram

Table 1: The Pitch Shifter Interface Explained

Signal Name	Function
PitchUp	When asserted, increments the amount we shift the pitch.
PitchDown	When asserted, decrements the amount we shift the pitch.
Shift?	When asserted, output pitch-shifted audio signal.
PassOrig?	When asserted, output original audio signal.
	When both asserted, mix both together and output.
Samp. Freq. Select	Chooses one of two sampling rates.
Buff. Size Select	Chooses one of sixteen different sampling buffer sizes.
/RESET	When asserted, clears pitch shift amount to 0 and resets MCU.

We expand Figure 1 to produce the block diagram of Figure 2. We do this simple expansion to isolate the analog to digital (A2D) and digital to analog (D2A) conversion from the complexities of the digital circuitry. Note that you could (and should) verify the correct performance of your A2D and D2A by providing a rather simple digital system which simply connects them and provides the appropriate control signals.

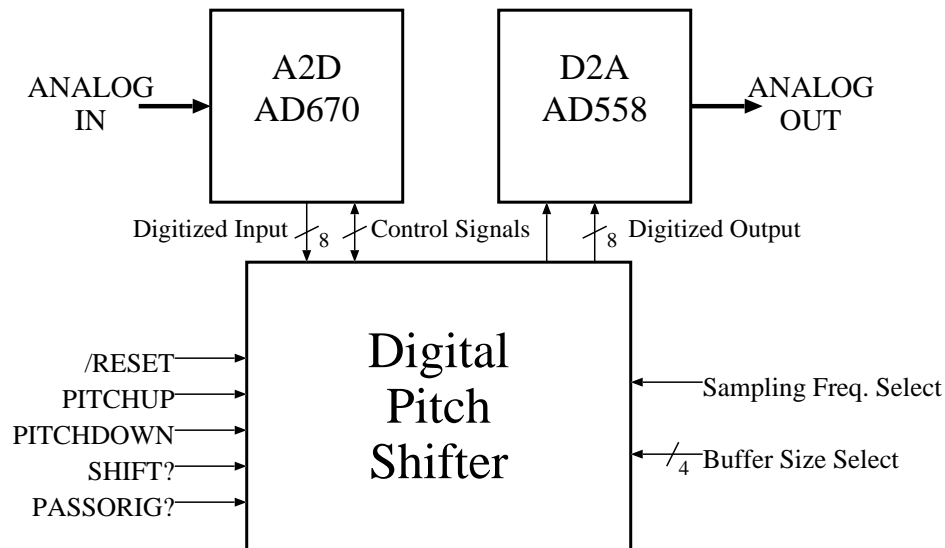


Figure 2: Analog/Digital Separation

## APPROACH

The much simpler approach to pitch shifting which we alluded to above is shown in Figure 3. Basically, we divide the audio signal into relatively short chunks. (The chunk size is selectable using the “Buffer Size Select” switches.) Then, we throw away some portion of each chunk and “stretch” what remains to fill the empty space. Because the waveform is stretched out, the frequency is lower and thus we perceive a lower pitch. The human ear and brain are surprisingly good at filling in the missing information from the audio signal; we expect that in most cases, it will be hard to tell that anything is missing at all.

In order to raise the pitch of a signal, we take a similar approach: Within each chunk of the signal, “squish” the waveform somewhat and fill the leftover space with part of another copy of the compressed waveform. We expect that that raising the pitch will not be as seamless as lowering the pitch, since the human ear and brain are not as good at ignoring extra information as they are at replacing missing information. The effect of this is that there will be slight echo-like effect added when raising the pitch of an audio signal. We would be interested to know if you discover a way of minimizing this unwanted effect.

The pitch shifting seems to take place in real time, but as you may have realized, there has to be a slight delay. This is because in order to stretch or compress the audio signal, one full chunk of the signal has to have already been stored into memory. The trick is to use two different buffers; as the “Sampling Buffer” is filling up with new samples, we can stretch or squish the contents of the “Shifting Buffer” and output the shifted waveform.

The algorithm we use, then, is the following: Take one sample of the input signal and store it into the Sampling Buffer. Choose one sample from the Shifting Buffer as a function of how much we want to shift the pitch, and output it. Repeat until the Sampling Buffer is full. When the Sampling Buffer is full, swap the roles of the buffers and start again.

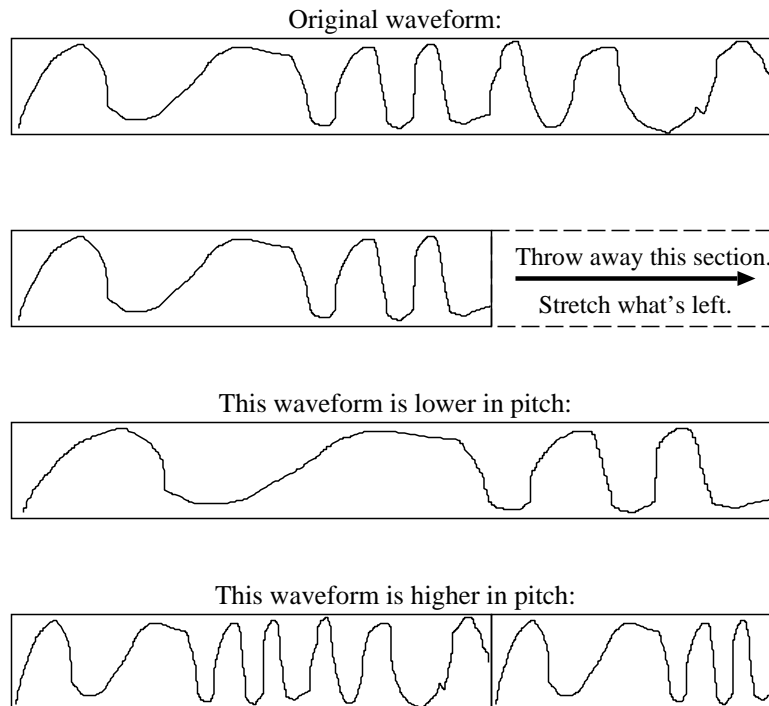


Figure 3: “Stretching” and “Squishing” a Waveform To Change Its Pitch

## SYSTEM ORGANIZATION

A system block diagram is shown in Figure 4.

The TIMING UNIT produces the necessary clock signal to drive the MCU sequencer and the other flip flops on the machine. In addition, this unit must also provide two different frequencies which will be used as a cue for the MCU to start an A2D conversion. Which frequency is used depends on the setting of the sampling frequency selector switch.

The A2D converter is to be implemented by a single chip, the AD670. You should wire it on the left hand proto strip of your kit which has special analog power supplies.

The STORAGE UNIT consists of a static RAM (SRAM) and a somewhat complicated address counter which we expect you will implement in a CPLD. It is used to store the digitized analog input signal as converted by the A2D. You should think of the SRAM as being divided into two equal-size called buffers. At any given point, new data received from the A2D will be placed into the Sample Buffer, as the Shifting Buffer (containing previously-recorded samples) provides the waveform to be stretched or squished. When the Sampling Buffer is full, the roles of the two buffers are swapped.

The SIGNAL ACCUMULATOR allows mixing the pitch-shifted version of your signal with the non-shifted original signal. This makes possible special chord-like effects, especially when several Helium Breaths are connected in series with one another; of course, it should be able to also pass only the shifted signal or only the original signal. It then provides the digital data to the D2A which in turn produces the analog output signal.

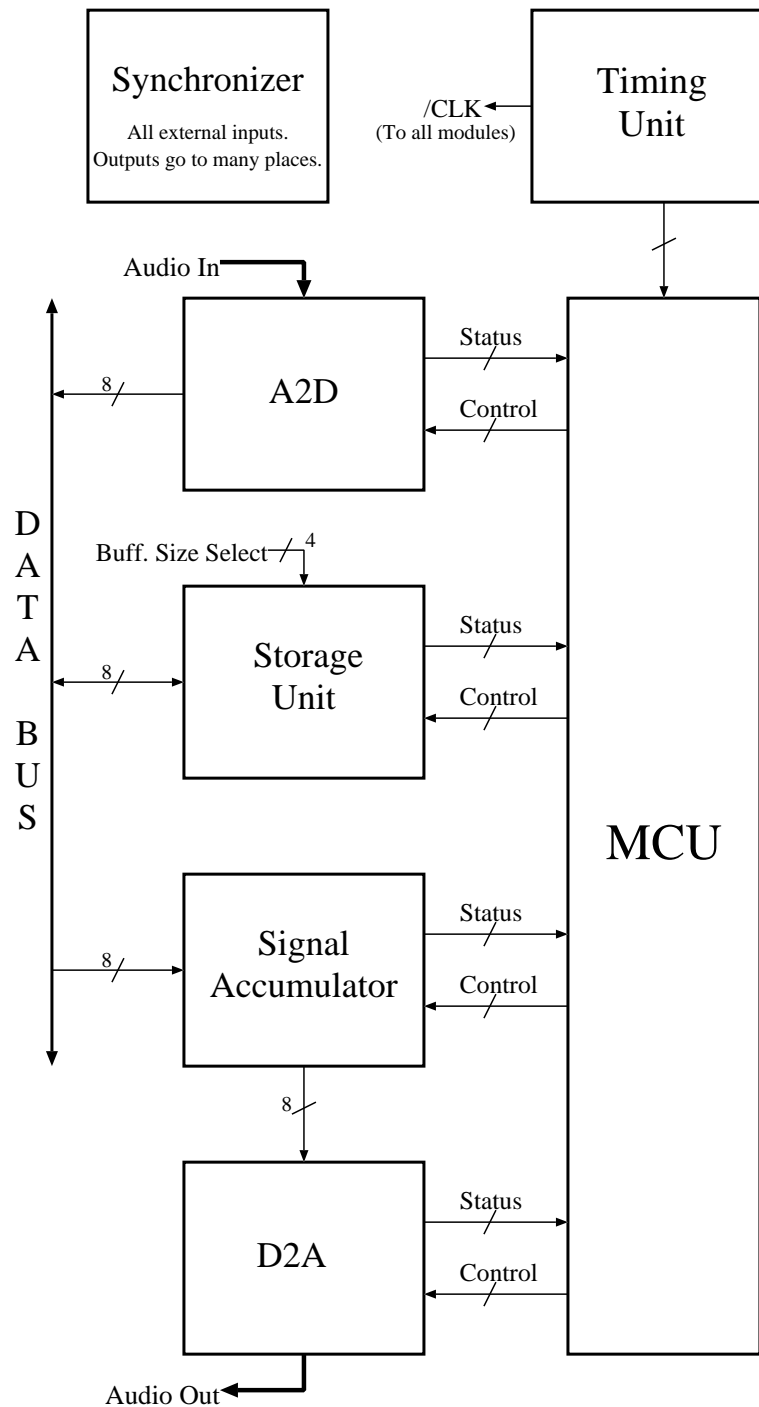


Figure 4: System Block Diagram

The D2A is a registered digital to analog converter which converts the data bytes retrieved from the SIGNAL ACCUMULATOR to an analog voltage. It is to be implemented with a single chip, the AD558. You should wire it on the left hand proto strip of your kit which has special analog power supplies. The analog output signal can be viewed on an oscilloscope or it can be used to drive a speaker.

A Microprogrammed Control Unit (MCU) senses the status signals from the other blocks and provides the control signals to them so as to implement the desired function.

Of course, all signals which are asynchronous to your system clock must be passed through some sort of SYNCHRONIZER before affecting the inputs to multiple flip-flops in your system. The SYNCHRONIZER is also responsible for converting an assertion of PITCHUP or PITCHDOWN to a periodic pulse so that the pitch increases or decreases relatively slowly when the corresponding buttons are pressed.

## SECTION 1: MCU and TIMING UNIT

### MICROPROGRAMMED CONTROL UNIT

This portion of the lab will guide you in the design and testing of the microprogrammed control unit (MCU), which serves as the basis for the rest of the lab. *Once the MCU has been successfully completed, you may continue to build and debug the other sections individually.* However, you should first complete the entire design. Defer the wiring of the second part until your MCU works.

Since the MCU is the mind of the project, it must be able to accomplish several things. First, it must be able to assert signals that control the flow of information. Second, it must be able to read status signals from the other units and make decisions based on those signals. Last, it must be able to determine the address of the next instruction to be executed.

Figure 5 shows a possible implementation for your MCU, using 2 LS163s as a sequencer. It allows for assertion statements, unconditional branches, and conditional branches which can test up to seven different status signals. If you really require testing of additional status signals, then it is fairly straightforward to use more bits of the instruction word as condition selection bits and to provide a wider multiplexer.

All of the code to be executed will be stored in Flash Memory. Each memory address (line of code) contains an encoded instruction for the sequencer and either assertion levels of control signals or a combination of status signal selection and next address value. The instruction set and location of these bit fields are illustrated in Figure 6. The word length is sixteen bits, but you may not have to use all of them. In the simplest possible MCU design, the value of the next address would just be the current address plus one. In this case every instruction would be executed in the order in which it appears in the Flash Memory. These instructions would assert whatever signals were necessary to accomplish the given function. Much power is gained when the facility to jump conditionally to a non-sequential address is added, allowing us to dynamically change the order of instructions.

The sequencer determines the next address based on the current instruction and the status bit which it sees during that instruction. The status bit is selected by an LS151 8-to-1 multiplexer using condition select bits from the Flash Memory. Using LS163s as a sequencer, only one condition bit may be viewed at any time. Therefore, the flow of information is as follows. The PROM contains

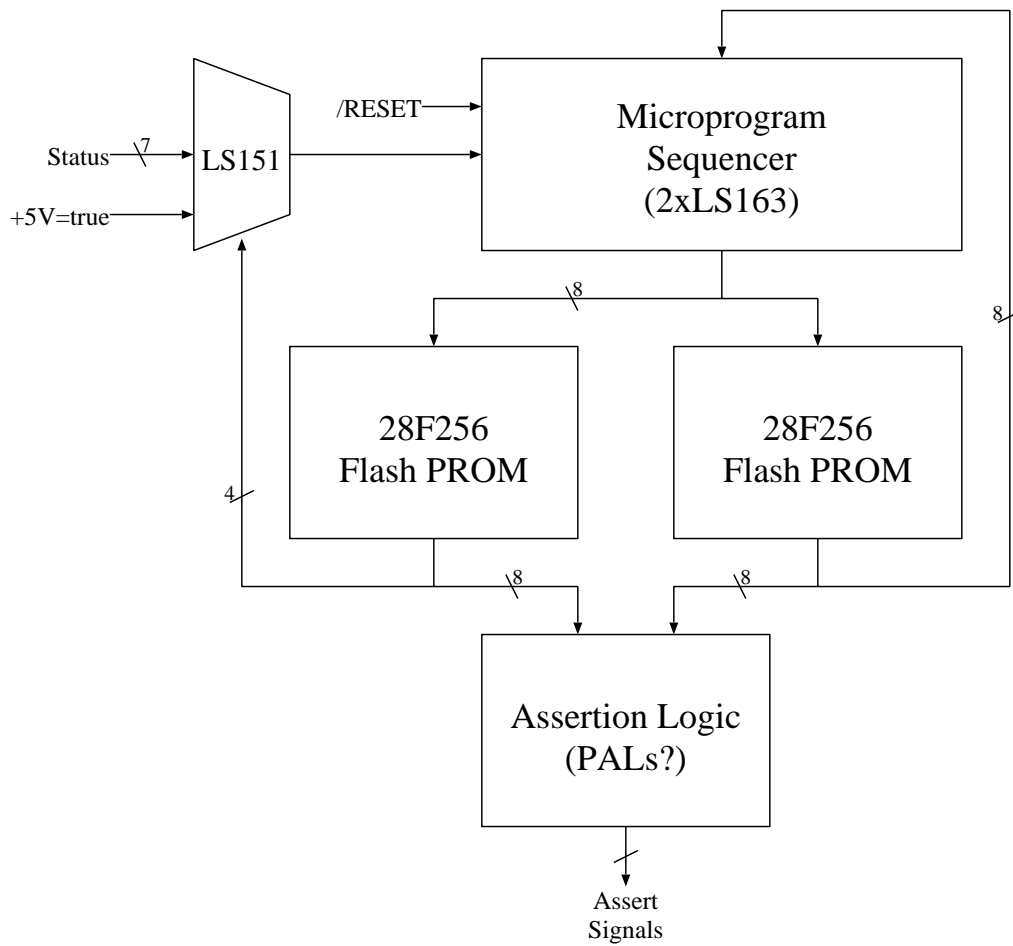


Figure 5: MCU

```

/*****
/* Instruction Word Organization: */
/* conditional branches          0cccxxxx aaaaaaaaa */
/* unconditional branches       0111xxxx aaaaaaaaa */
/* assertion statements         1sssssss ssssssss */
/* where c = status selection   */
/* a = alternative address, i.e. jump address */
/* s = assertion signals        */
*****/

op <15:0>;          /* Indicates the available bits */
address op <7:0>;   /* Indicates bit locations for addresses */
value op <7:0>;     /* Indicates bit locations for data (optional) */

/*
 * There is nothing magic about upper case.
 * You may change things to lower case as you wish.
 * Remember, the assembler maps all characters to lower case anyway!
 */

/*
 * Instruction set for your MCU
 */

CJMP   op<15>=0;    /* Conditional JuMP */
JMP    op<15:12>=%b0111; /* unconditional JuMP */
ASSERT op<15>=1;    /* unconditional ASSERT */

/* These are defined so that you may use them to make your code more
 * readable. Their use is not required. */

IF     nop;
THEN   nop;
TRUE   op<14:12>=%b111; /* This selects the +5 input to the mux */
RESET  op<15:0>=%b0111000000000000;

```

Figure 6: MCU Instruction Set

the encoded instruction and address or signal assertions. The instruction bit is fed from the high-order Flash PROM to the /G enable input of the LS151. The selector inputs of the LS151 are the condition select bits, which come from the same PROM. The inverted output of the LS151 is then fed into the condition code (/LOAD) input of the sequencer. This causes the LS163s to load only on a rising clock edge where the instruction bit is '0' (i.e. a **branch** instruction) *and* the MUX input specified by the select bits is high. Tying one of the MUX inputs to +5 V allows for unconditional branching by testing a signal that is always true.

The Qn outputs of the sequencer are the address of the current instruction; this address is tied to the inputs of the code Flash Memory, so whatever instruction is on the PROM outputs should correspond to the address at the LS163 outputs. On a rising clock edge in the case of a conditional jump that is true, the LS163s load the address at their D through A inputs (coming from the low order PROM outputs). In the case of an assert instruction or a failed conditional branch, the LS163s simply increment their output by 1. The /RESET signal is tied to the /CLR inputs of the LS163s, so pulling /RESET low effectively forces a jump back to the first instruction in the microprogram.

The PROM outputs are also fed into assert logic which determines when to assert the control signals. We recommend you implement this assertion logic in one or more PALs. The assert PALs must have the information as to what type of instruction is being executed so that an address on the PROM outputs is not interpreted as signals to be asserted. The PAL keeps each unasserted signal at its proper logic level, which could be either a 1 or 0, whereas a 0 logic level on the PROM always means unasserted and a 1 always means asserted (provided you are executing an assert instruction).

Remember to synchronize status signals to your system clock. Some of the signals asserted by your MCU must be “glitch-free”. Remember that both PROM and PAL outputs will likely have glitches. You should provide glitch-free signals (where required) either by registering your assertions in a PAL flip flop or by gating the assertion signal with the second half of the clock period.

Software to ease the task of programming the two Flash Memory chips is described in a separate document. The microcode assembler processes symbolic microcode and produces an output consisting of a hex integer for each microcode instruction. The dat2ntl program is then used to produce two separate Intel HEX files, byt0xxx.ntl and byt1xxx.ntl, which are required to program the low and high bytes of your sixteen bit microinstruction words. A shell script, `assem16to8`, is provided so that you only need to type one command:

```
assem16to8 <filename>
```

to produce the two Intel HEX files directly from your microcode source file.

It is unlikely that you will get your microcode exactly as you want it to be the first time. We urge you to spend the time and effort required to learn to use the microcode assembler. It is far easier to make minor changes and try again if you merely have to edit your symbolic microcode and run `assem16to8` another time.

Available are partial assembly (mcu.as) and specification (mcu.sp) files from which you may wish to start. You are not required to use these, but they may be helpful. They are located in the course locker:

```
/mit/6.111/handouts/labs/lab3.s02/mcu.as  
/mit/6.111/handouts/labs/lab3.s02/mcu.sp
```

You may copy these into your own locker and edit them to produce your own source files.

## TIMING UNIT

The Timing Unit is responsible for supplying the clock signals for the MCU. The MCU may comfortably run fairly quickly; 921.6 kHz would not be an unreasonable clock frequency. You should convince yourself that whatever frequency you choose does not violate the timing constraints of the devices in your system. All flip-flop inputs should be stable and valid for at least the setup time of the flip-flop before the next rising clock edge. Since the clock signal must be fed to many inputs, it is a good precaution to buffer your clock signal with multiple inverters to avoid fan-out and noise problems. See a TA if you're confused about how to do this.

The Timing Unit must also generate the necessary SAMPLING signal that tells the MCU to initiate an A2D conversion. This signal should actually have one of two different frequencies depending on the setting of the sampling frequency selector switch. The choice of frequencies is up to you, but remember that the Nyquist Sampling Theorem requires you to sample at at least twice the frequency of your audio signal in order to accurately reconstruct it. To this end, we suggest 9600 Hz as a good base sampling rate. 19.2 kHz might be a good sampling rate for higher quality sound. It will become clear to you that the faster you sample, the more memory you will require to perform the pitch shift; therefore, faster is not always better.

## SECTION 2: I/O and STORAGE UNIT

### ANALOG INPUT

The A2D converter is to be implemented by a single chip, the AD670. You should wire it on the left hand proto strip of your kit which has special analog power supplies.

The specification sheet for the AD670 offers several different approaches for handling the timing of the device. We recommend that you follow the timing diagram in Figure 7. Be sure you understand the various propagation delays of the A2D.

The actual audio input signal could come from a number of different sources: a microphone, a cd player, a function generator, even a simple voltage divider. You can borrow function generators and microphones for a short amount of time from the equipment desk. In any case, you will need to think about the input voltage range of the AD670 and how to place your signal within this range.

### STORAGE UNIT

The STORAGE UNIT consists of a static RAM (SRAM) and a complex programmable logic device (CPLD). We expect you to build on the techniques you learned in Lab 2 to interface with the SRAM. The complexity here is not in interfacing with the SRAM, but in the design of the counters

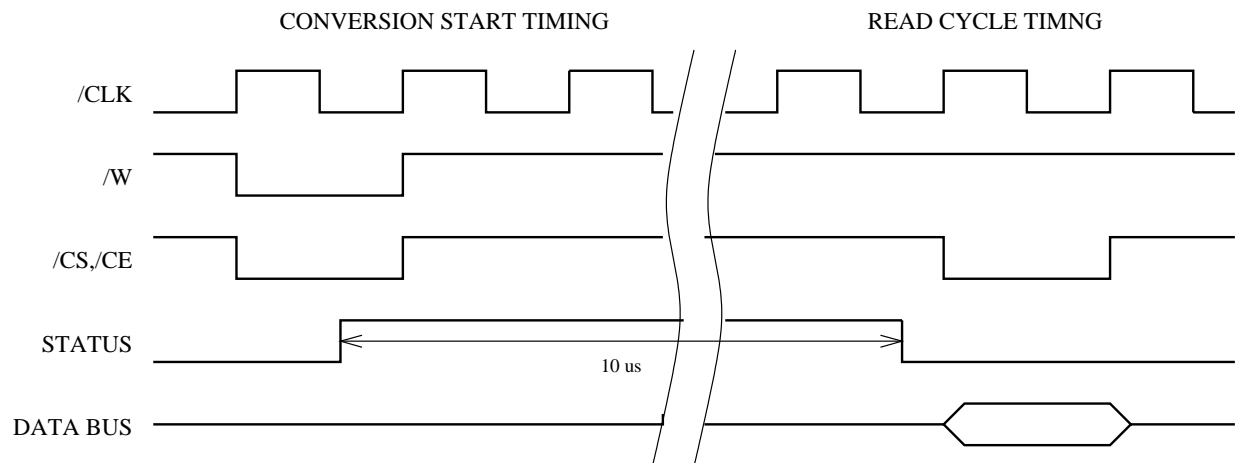


Figure 7: Analog to Digital Conversion Timing

used to index it; clever indexing of the SRAM is what gives us the desired stretch or squish of the sampled audio waveform. We expect that you’ll be able to implement this indexing mechanism in one C374i CPLD on the CPLD board.

The SRAM you will use is a 6264, available from the equipment desk. Like the 6116, it is eight bits wide and has /CS, /WE, and /OE, but it has two additional address inputs and is therefore four times larger.

For the purposes of this lab, we suggest that you divide the SRAM into two buffers of 2048 bytes each; by setting one of the SRAM address bits to 0 or 1, you can control which buffer is being accessed. At any given point in time, the Sampling Buffer is being filled up with samples, while certain samples are being selected for output from the Stretching Buffer, which has already been filled.

Figure 8 shows the design we propose you use for your memory unit. As you can see, it consists of two separate address counters whose outputs are multiplexed together. The first one, the so-called “Sampling Counter”, is designed to index the Sampling Buffer. It counts up by one on every rising clock edge that Count is asserted. The Sampling Counter is 11 bits wide, and therefore can address all 2048 samples in one SRAM buffer. The ClearSamp signal is a synchronous clear for the Sampling Counter.

The “Shifting Counter”, on the other hand, is intended to keep track of which sample to output next when stretching or squishing the waveform stored in the Shifting Buffer. When ClearShift is asserted during a rising clock edge, the Shifting Counter clears. Otherwise, it is capable of counting in non-integer increments, greater or less than one. The idea is that by incrementing the Shifting Counter by slightly more than one, some samples will be skipped, thus squishing the waveform and producing a higher-sounding output. Likewise, by counting an increment slightly less than one, some samples will be repeated, stretching the waveform out and giving a lower-sounding output.

You can implement a counter that counts by non-integers by building it wide—we suggest 17 bits for your Shifting Counter. The most significant 11 bits represent the integers 0-2047; they can be used to address the 2048 samples in an SRAM buffer. The least significant 6 bits, then, are the fractional component of the counter. We can increment the counter by exactly 1.0 by

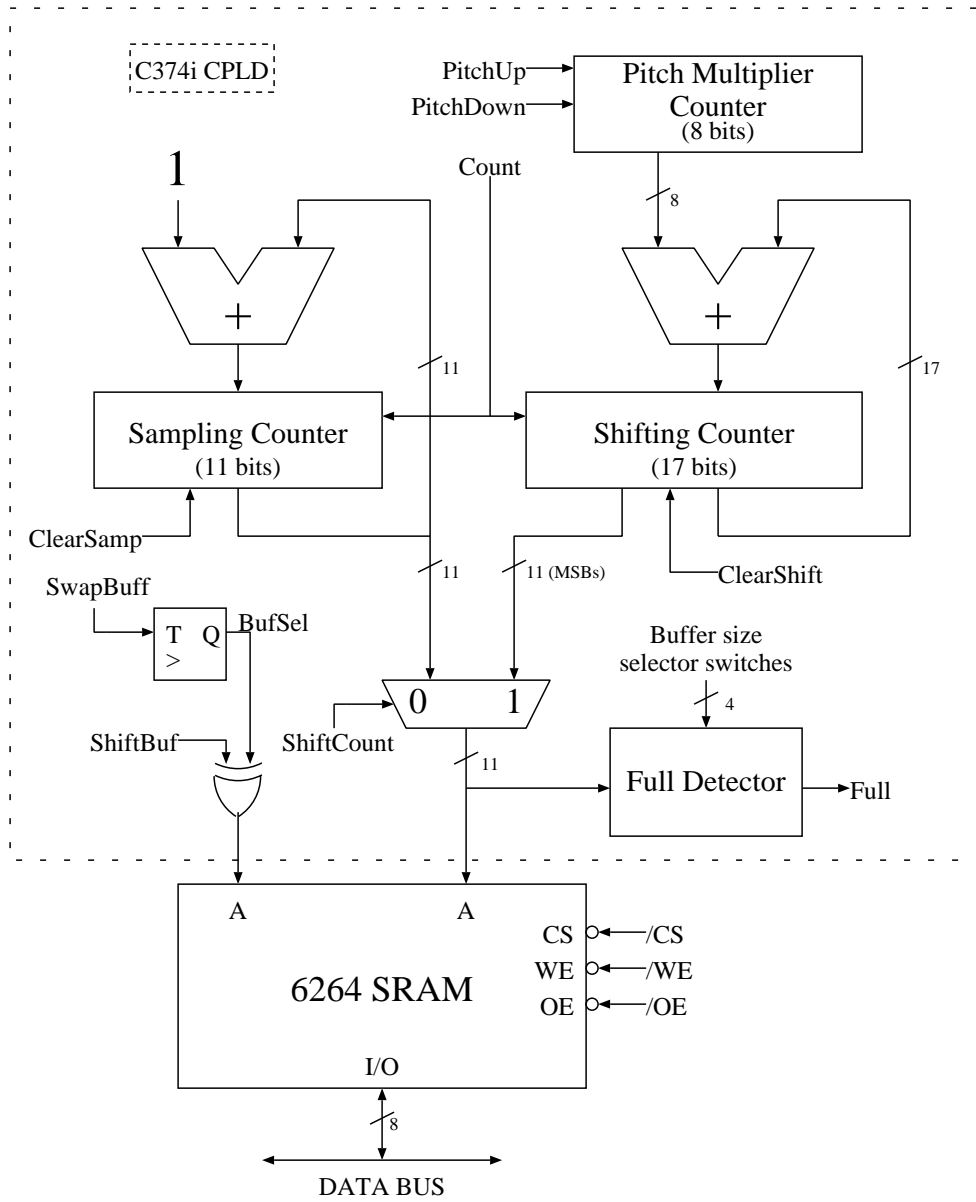


Figure 8: Storage Unit

adding 01000000 to the value it contains. Adding 01000001 increments the counter by slightly more than 1.0, and adding 00111111 increments it by slightly less.

The amount by which we increment the Shifting Counter comes from the eight-bit “Pitch Multiplier Counter”. When PitchUp is asserted on a rising clock edge, the Pitch Multiplier Counter should count up. When PitchDown is asserted on a rising clock edge, it should count down. Thus, PitchUp and PitchDown raise and lower the pitch of our output signal by controlling the amount we stretch or squish the stored waveform.

The ShiftCount signal determines which counter, the Sampling Counter or the Shifting Counter, addresses the SRAM. You will want to use the Sampling Counter when storing samples to memory or outputting a non-shifted version of your signal. You will want to use the Shifting Counter when outputting a shifted version of your signal.

The T flip-flop labeled “BufSel” has one simple purpose: to remember which buffer is currently the Sampling Buffer and which is the Shifting Buffer. Asserting SwapBuf causes the two buffers to switch roles.

Finally, the buffer size selector switches allow the user to optionally use buffers smaller than 2048 bytes. You may discover that different buffer sizes work better for different types of signals and different pitch multipliers. Depending on the setting of the switches, your CPLD should detect when its address outputs are greater than a certain number, and if so, the Full output should go high. This lets the MCU know to clear one or both of the counters and perhaps also swap buffers.

All of the modules outside the SRAM should fit fairly easily into the C374i CPLD. What remains is for you to write the VHDL. We’ve provided you a starting point (the entity declaration); you can find it in:

```
/mit/6.111/handouts/labs/lab3.s02/mar.vhd
```

## SIGNAL ACCUMULATOR

This unit is designed to give you the chance to output *both* the pitch-shifted signal and the original, unshifted signal at the same time. This should allow for some interesting effects, especially if you connect multiple kits together.

Mixing two signals together can be accomplished by an adder-accumulator pair. The accumulator should be a loadable, clearable, eight-bit register. The input to the accumulator is the output of an eight-bit adder; this device adds the previous value at the output of the accumulator to the value currently on the bus.

We suggest using a pair of LS283s as an eight-bit adder. There may be other, more optimal approaches to the design of this module, and you are welcome to take them.

## SYNCHRONIZER

As with all synchronous digital systems, all inputs which affect multiple flips flops must be guaranteed not to change during a rising clock edge. Your system should be sure to synchronize all asynchronous inputs.

In addition to synchronizing the PitchUp and PitchDown inputs, your system should also convert them to a pulse, possibly a periodic pulse. The problem with connecting them directly to the CPLD is that a single press of the PitchUp button will last many clock cycles; your machine's pitch-shift will increase too rapidly to be controlled. You need to make sure the pitch increases only a manageable amount with each press of the button. A nice behavior would be a slow increase in pitch when the PitchUp button is held down.

## ANALOG OUTPUT

The D2A you should use is the AD558 contained in your lab kit. This D2A is quite easy to use and includes an eight bit latch. Be careful about the nomenclature for the MSB, etc. Often A2D and D2A converters do not follow the usual convention where D0 is the LSB and D7 is the MSB. In fact, different analog devices are labeled differently, i.e. inconsistently! Note that the register is a latch and not an edge-triggered register. Be careful and do not allow any glitches on the clock input! Read the data sheet!

When you are ready to listen to your pitch-shifted output signal, check out an amplified speaker from the Instrument Room and connect your D2A to the speaker through a capacitor (0.05  $\mu\text{F}$ ).

## TROUBLESHOOTING HINTS

1. Build and debug in stages. Try to find ways of introducing a manageable amount of complexity to your system, testing that it works, and only then introducing more complexity. Building the whole system at home before testing any of it out is a proven recipe for disaster.
2. For test purposes, a reasonable approach is to initially provide the data inputs to your storage unit by wiring the CPLD memory address counter outputs to the memory tristate I/O pins through a tristate buffer. This way, you can store a ramp into the memory and check out the storage unit and MCU.
3. Wire your A2D outputs directly to the D2A and provide a microprogram to control the A2D and D2A to test this pair of chips.
4. Be careful with your memory timing. Remember lessons learned with Lab 2.
5. Do NOT attempt to debug everything using only your logic probe! The logic probe is provided as a convenience for you when you cannot get to the lab. Your first instinct should be to look at signals with the scope or the logic analyzer.
6. Do not use real-life audio signals like voice or music to debug and test your system. Instead, use some kind of periodic function like a sine or square wave, running at some audible frequency. Compare your input and output on the scope. Can you see the stretch and squish effect? Do the buffer size selector switches make a difference in the appearance of your output?
7. The Sample Accumulator should probably be the last thing you build. First make sure you can output both your original signal and a pitch-shifted signal, with the D2A inputs connected directly to the bus. Experience with LS283s leads us to suggest you take extra care in wiring them—it's easy to make a mistake! Consider writing a special test program which isolates the adder from the rest of the system.
8. You can include several test programs (and even the "real" one) in your Flash Memory by using the `#new_program` command statement and wiring the high order Flash Memory address

bits to allow selection of the program to be run. Read the microcode assembler and dat2nt1 program documentation.

9. If you're having trouble getting everything to fit in your CPLD, change Warp's "Generic" options to optimize for area rather than speed. It is most likely that your VHDL is "wrong" rather than overflowing the capacity of the C374i CPLD.

## WHAT WE EXPECT FROM YOU

### MCU CHECKOFF

Before building anything else, you should first make sure your MCU and Timing Unit are working reliably. We therefore require you to demonstrate their functionality using a test program which we have provided. The program tests ASSERT and CJMP instructions, and makes an attempt to verify that all your branch address bits are wired correctly. Note that our test program doesn't guarantee correct functionality, it just tries to catch some common errors.

You should also design a frequency divider that will take as input your sampling frequency and produce a much lower-frequency signal as output. Recall that you should design your sampling signal to have one of two frequencies depending on the setting of a switch. The VHDL you write here should divide the lower of these two frequencies down to somewhere in the range of 10–20 Hz.

The programs are:

<code>/mit/6.111/handouts/labs/lab3.s02/mcutest.as</code>	assembler file, microcode
<code>/mit/6.111/handouts/labs/lab3.s02/mcutest.sp</code>	specification file
<code>/mit/6.111/handouts/labs/lab3.s02/mcutest.vhd</code>	accompanying assert PAL
<code>/mit/6.111/handouts/labs/lab3.s02/mcutest1.vhd</code>	frequency divider

If all is working fine, you should be able to use the switches to control a nice display of blinking LEDs. The MCU test code uses the output of your CPLD to determine how fast the lights blink. Therefore, toggling the sampling frequency selector switch should change the blinking speed. Be sure to pay attention to the comments in the files we have provided. They give not only wiring instructions but also hints for what to do if something doesn't work.

After you've gotten your working MCU checked off with a staff member, feel free to remove the wiring to the switches and LEDs. You won't need it in the rest of the lab.

### DESIGN REVIEW

You are not required to, but are **STRONGLY** encouraged to, meet with a T.A. for a design review before you begin wiring the second half of your system. You do not have to have detailed circuit diagrams or completed microcode before the design review.

You should, however, think about the problem and come for a design review with a block diagram, flowchart for your microcode, and a reasonably clear idea of the approach. Specifically, you should think about how you are going to test your circuits. A complicated design rarely works the first time you turn on the power. Consider how you might test individual modules before

interconnecting them. A little bit of well chosen extra test circuitry can actually reduce the time and effort required to get your design to work.

You should bring along your first attempt at VHDL for your memory counter.

Be aware that seriously thinking about how to optimize your design before you begin building can significantly reduce the amount of wiring you have to do. In any case, make sure your design will fit on your kit by placing the chips you expect to use; try to leave space for last-minute “patches”. Consider using the multiple CPLDs on the CPLD board.

## IMPLEMENTATION

After arriving at the final design, you should prepare detailed logic diagrams for both the data paths and your control circuitry. These should include pin numbers and chip locations. Wire your kit from these detailed logic diagrams (not from block diagrams) and test the system. When operational, it should be demonstrated to any staff member in the laboratory. Bring your circuit diagrams to the lab when coming for a demonstration as the T.A. will initial and date your diagrams. Be sure to include this witnessed copy in your report. After the demo, complete your report and turn it in.

Actually, there is nothing wrong with writing your report before the demo. Often it is a good idea to write the report while the debugging and testing are proceeding. This can not only provide a change of pace, but the thought going into the organization of the report can actually be beneficial in the debugging process. If you write your report before finishing the debugging, you will likely have to change the report only in a minor way, if at all, when your system is fully operational.

## LABORATORY REPORT

After the completion of Lab 3, a report is required. It should emphasize both the theory of the design and the problems of practical implementation.

Your report should include the following items:

1. Introduction - a brief description of the problem and a block diagram
2. Information and description of your MCU
  - (a) Define your instruction format.
  - (b) Include a flowchart for your microcode.
  - (c) Describe the operation of your code in English.
  - (d) Include your microcode listing file (with comments).
3. Detailed circuit diagrams
  - (a) Include a circuit diagram which shows all gates, flip flops, PALs, and registers, etc.
  - (b) Try to make a reasonable compromise between legibility and detail. Include IC position designations and pin numbers. Where reasonable, draw the wire connecting two (or more) pins. Do not, however, run wires all over your diagram when this makes it hard to understand. Instead, label the pins with (unique) signal names.

- (c) You do not have to draw equivalent circuits to describe the contents of CPLDs and/or PALs. You must include your PALASM or VHDL file for any PLDs used. These should be accompanied by a paragraph describing the function of the PLD circuitry. This, of course, could be in the form of comments in the PALASM or VHDL file.
4. Detailed circuit description. Accurately and clearly describe the functions of the circuits.
  5. Timing diagrams for major signals, especially those that, in your opinion, are tricky, non-obvious, or interesting. Refer to these timing diagrams in your detailed circuit description.

The keys to a successful report are organization and clarity. A short paragraph with a diagram or table usually communicates your intent to the reader far better than a long-winded written explanation.

### **PROBLEMS?**

If you encounter problems in your design, please come in and ask. We regard this lab exercise as a most important experience for you in preparation for the final project. We want you to succeed and will do our best to help. Please do not leave everything to the last minute as this will make it difficult to have time to help you.