UNIVERSITY OF BRITISH COLUMBIA
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EECE 259: Introduction to Microcomputers
Assignment 3: Music Jukebox
Handed out Feb. 26, 2011

Overview

In this homework, you will learn how to:

1. Write a larger program with subroutines,
2. Operate with a multiple larger amount of data (audio),
3. How to access halfword and byte data sizes,
4. How simple audio data is encoded in bits, and
5. How to play and record audio data.

You are encouraged to keep things simple in PA3, and focus extra time you have on your PA4 project.

PART 0: Audio Processing and Byte / Halfword Data Sizes

This section describes how to access audio data and how to output sounds with your DE1 board.

A. Audio Data (example: wavdata2.s)

Sound data is stored as a series of **16b signed numbers**. The first 16b is for the LEFT speaker, the next 16b is for the RIGHT, the next 16b is the next sample the LEFT, and so on. The sample rate is 44.1kHz, the same as a CD. This means you need 44,100 16b audio samples per second for the LEFT speaker.

The 16b value represents an 'amplitude' or intensity level of the sound at that instant. As you go from sample to sample, the sound changes over time so it may increase or decrease. Of course, it will oscillate… oscillating between +32767 and -32768 would be extremely loud.

You can mix 2 songs by adding their sample values (i.e., LEFT1+LEFT2 and RIGHT1+RIGHT2). When you do this, you have to be careful of the possibility of overflow beyond +32767 or -32768. To avoid this being a problem, you may need to rescale the audio data. One way is to divide all of the values by 2, eg MIXLEFT = (LEFT1+LEFT2)/2, but this may lower the volume too much. Instead, you might try something more gradual, like "multiply by 3 then divide by 4". To be really careful, you might mix in two passes: the first pass, you simply check the value of (LEFT1+LEFT2) to find the maximum value, while the second pass you actually mix the samples and scale by a suitable amount to avoid overflow.

B. Audio Playback and Recording (examples: playback.s, recordecho.s, recorddelay.s)

You play sounds by writing a 16b sound value to the **digital-to-analog converter**, also known as a DAC, which converts it to an analog voltage. By sending samples at the right rate (44.1kHz), the voltage goes up and down at the right speed to match the original sound. There are 2 separate DACs, one for LEFT and one for RIGHT audio. You write a value to each DAC using "sthio" to the correct offsets, SNDL and SNDR; the full effective addresses are SNDL+IOBASE and SNDR+IOBASE. You must write a value to both SNDL and SNDR for a sound to be produced. Since SNDL and SNDR should be read/written as halfwords

using ldhio/sthio, the values must be between -32768 and +32767. The following code sends two sound values (left and right) as halfwords to the DAC:

```
                movia r23, IOBASE
                movi       r4, 0xff03       /* sign extended */
                movi       r5, 0xef80       /* sign extended */
                ...
play:           sthio r4, SNDL(r23)         /* LEFT sound */
                sthio r5, SNDR(r23)         /* RIGHT sound */
```

The Nios II operates at 50MHz. How many of these 50MHz clock cycles will occur between sending two samples to the DAC at 44.1kHz? (hint: around 1,000).

Playback at the right speed is simplified on the DE1/DE2 using special hardware. When you write to the SNDL or SNDR addresses, the value doesn't go directly to the DAC. Instead, it is automatically stored in a queue/FIFO buffer that holds up to 128 samples. Each write stores one more sample in the FIFO buffer. The hardware automatically removes one sample every 44.1kHz cycle. If you try to write too many samples, the FIFO will become full and the values written after that point will be ignored. For this reason, you must first check if the audio FIFO has space before writing to SNDL or SNDR. The following code waits until the FIFO is <u>not full</u>, then sends one new sound sample. After the "srli" instruction, register r2 tells you how much space is left in the FIFO (up to 128 samples):

```
playsound:      ldwio r2, SNDRDY(r23)
                srli       r2, r2, 24           /* r2: amount of space */
                beq        r2, r0, playsound    /* wait for space */
play:           sthio r4, SNDL(r23)
                sthio r5, SNDR(r23)
                ret
```

Recording is done in the opposite direction using "ldhio" to read from SNDL or SNDR. One new 16b value will be ready for each left and right channel every 44.1kHz cycle and placed into a FIFO automatically by the hardware. Your program can read this FIFO as follows:

```
                movia r23, IOBASE
                ...
record:         ldhio r2, SNDL(r23)
                ldhio r3, SNDR(r23)
```

However, you should only read data from the FIFO if it is non-empty. Reading audio data too frequently while recording (when the FIFO is already empty) produces invalid sound data. You should always read from both left and right sides (even if you just want the left side data). The following record code waits until the FIFO is <u>not empty</u>, then returns one new sound sample in r2 and r3. After the "andi" instruction, register r2 tells you how many samples are already stored in the FIFO waiting to be removed (up to 255 samples):

```
recsound:       ldwio r2, SNDRDY(r23)
                andi       r2, r2, 0xff         /* r2: # samples ready */
                beq        r2, r0, recsound     /* wait for data */
record:         ldhio r2, SNDL(r23)
                ldhio r3, SNDR(r23)
                ret
```

You can only record from the line-in jack.

C. Words, Halfwords, and Bytes

Prior to this assignment, we've used word-sized data exclusively. However, Nios II also supports byte (8b) and halfword (16b) sized data with the following instructions:

```
ldh     r2, 0(r3)   |   ldhio   r2, 0(r3)
ldb     r2, 0(r3)   |   ldbio   r2, 0(r3)
sth     r2, 0(r3)   |   sthio   r2, 0(r3)
stb     r2, 0(r3)   |   stbio   r2, 0(r3)
```

The "io" variants are needed to bypass the cache when you use the special I/O addresses.

When you read a halfword or byte with ldh/ldb, **the upper bits are sign-extended to fill the 32-bit register**. If you don't want sign-extending, **use the unsigned variants, ldhu or ldbu**. When you write a halfword or byte with sth/stb, only the lowest 2 or 1 bytes of the register will be saved to memory (the upper bits in the register will be ignored). When writing to memory, only these 2 or 1 bytes will be modified (that is, the other 2 or 3 adjacent bytes that make up the remainder of the full word will keep their original value).

Working with non-word size data brings the opportunity for ***data alignment bugs***. When you access words, addresses always must always be a multiple of 4. Similarly, when accessing halfwords, the address must be a multiple of 2. If your address is not a proper multiple, you will observe some weird or unpredictable behaviour (you may get incorrect data, or the Nios II may generate a "software exception", meaning it stops running your program and tries to run a special error-handling software routine that you probably haven't written!). Fortunately, you can use any address (even odd ones) to access bytes. Yet, data alignment bugs will creep into your program more frequently now simply because you will have some registers holding addresses of bytes and halfwords – if you mistakenly use these registers with a "ldw" instruction, the alignment bug will raise its ugly head!

You can store byte, halfword, or word sized data in your program. For example, the following three lines store the same value, 0x87654321, as a word, sequence of 2 halfwords, or sequence of 4 bytes:

```
.word     0x87654321
.hword    0x4321, 0x8765
.byte     0x21, 0x43, 0x65, 0x87
```

This order of bytes in a word also shows that Nios II is ***little Endian***. That is, the least significant byte (or least significant halfword) is stored at the smallest address location.

You can also store character strings as byte-sized data encoded in ASCII format. Below, the string consumes 13 bytes of space (the 13th byte is 0x00 character needed to signify the end of the string):

```
.asciz    "Hello, world"
```

Now, you probably notice that it is possible to create data with an ODD number of bytes. If this is followed by a word of data, there is the *potential* for an alignment bug! For example:

```
BYTES:
.byte     0x12, 0x34, 0x56
WORDS:
.word     0xDEADBEEF, 0xCAFEBABE
```

Suppose BYTES starts at a multiple-of-4 address such as 0x100. Then, addresses 0x100, 0x101, and 0x102 will hold the 3 bytes and WORDS will start at 0x103. **Yikes, WORDS is not aligned to be a multiple of 4!** Don't panic, because the assembler/compiler will automatically detect this and will force WORDS to start at the next word boundary, 0x104. If the assembler isn't quite doing what you want it to, you can control its behavior using the ".align" directive. You can learn about this by typing 'gnu gas manual' in Google.

D. Example Code and Converting Audio Data

The course web site ZIP package "hw3 examples" contains several Nios II programs that play and record audio and print to the TERMINAL window.

The ZIP package also contains 2 Windows programs:

- "Audacity" converts several formats (like MP3) into a WAV file.
- "WavConvert.exe" converts a WAV file into assembly language data.

When you run "wavconvert", it will ask you for an input filename (eg: music.wav). It automatically writes the output to the file wavdata.s, which you can rename to include in your program.

PART 1: Study Questions – warning, solving these will be very time-consuming!!!

1. Design a flowchart and write a Nios II program that converts the value of a decimal number represented by a null-terminated ASCII string at label TEXT into an unsigned 32-bit binary number. This ASCII string contains only the ASCII characters "0" through "9". The string may contain leading zeros, and it will only contain valid values between 0 and 4294967295 (2^{32}-1). Store the result in memory at label NUMBER.

   For example, for the text "65535" your program must place 0x0000FFFF at label NUMBER. Similarly, the text "0123" must result in 0x0000007B. Hint: you can convert from an ASCII character to a decimal digit by subtracting the ASCII value of "0".

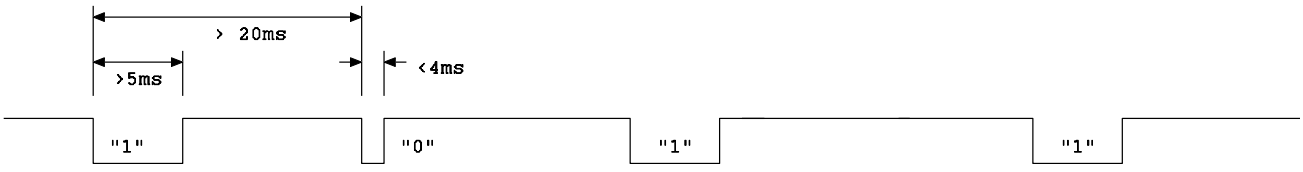
```
TEXT:
.asciz "32767"
ZERO:
.ascii "0"
NUMBER:
.word 0
```

2. Consider two separate null-terminated strings starting at labels NEEDLE and HAYSTACK. Design a flowchart and write a Nios II program that finds the starting memory location of the first occurrence of the NEEDLE in the HAYSTACK. Store this location in memory starting at label MATCH_ADDR. If NEEDLE is not present in HAYSTACK, store the value 0xFFFFFFFF instead. Do not exceed the length of the strings.

   For example, if HAYSTACK starts at 0x00001000 and contains "The Search Is Over" and NEEDLE contains "ear", the program would write 0x00001005 to MATCH_ADDR.

```
HAYSTACK:
.asciz "The Search Is Over"
NEEDLE:
.asciz "ear"
MATCH_ADDR:
.word 0xFFFFFFFF
```

3. *A one-finger keyboard!* You can enter 1 byte of data into a computer serially, one bit at a time, using just a single switch. By holding the switch closed for a long time (say, >5ms), you can enter a binary '1'. However, by holding it closed for a short time (say, <4ms), you can enter a binary '0'. To be sure you won't miss any data, you know the spacing between two bits will be at least 20ms apart. Design a flowchart and write a Nios II program that inputs exactly ONE byte of data from ONE switch and displays it on the 8 green LEDs. Assume the MSB arrives first. You can detect the arrival of each bit using polling. Use the built-in 50MHz counter to get nearly-precise delays.



4. Design a flowchart and write a Nios II program to compute the first N prime numbers. Store these numbers in a list in memory, starting at label PRIMES.

PART 2: Practical Assignment

This practical assignment requires a *significant amount of time* to complete. Start early!

You will write a program to turn the DE1 into a music jukebox according to the requirements below. Where something isn't clearly specified, you can fill in the details any way you wish.

1. The jukebox must play at least 4 different "songs". When you get to the "end" of a song, start it over again to simulate a long-playing song. *Each song must be of different length.* Of the 4 songs, one must be the complete "wavdata2.s" file given on the course web site. The other 3 songs can be "stored" or "recorded" in any number of ways (your choice) such as: i) other WAV music (your choice), or ii) a recording made with the DE1 using the mic/line-in connector, or iii) generating the sound directly, e.g. by computing a square wave, triangular wave, or something more complex (such as a sine wave) in your program.
2. Switches SW7 to SW0 mix together the first 4 songs during playback. For each song, 2 switches work as a pair: setting the left switch to 1 mixes that song's left sound to the LEFT speaker, while setting the right switch to 1 mixes that song's right sound to the RIGHT speaker. Hence, setting SW1 and SW0 only (and all others to 0) will play back song 0 on both speakers, while setting SW3 and SW2 only will play back song 1 on both speakers. Setting all four switches SW3..SW0 will mix both songs together (both left sounds appear on the LEFT speaker, both right sounds appear on the right). Any time SW7 to SW0 changes, print the names of the songs that are currently being mixed together on the terminal in ASCII text. Clearly indicate which songs are on the LEFT and RIGHT speakers.
3. Two KEY buttons are used to *increase* or *decrease* volume playback level. Pressing the third KEY button should restore the volume level to the natural level in the middle setting (where you are not adjusting the sound in any way). Display the current volume level on LEDG. You should have *at least* 8 volume levels. When changing the volume, the sound level must be adjusted immediately (without delay). *Do not modify the sound data in memory!!!* Instead, adjust the sound level on the fly as you send out each individual sound sample.
4. Display the average sound level on LEDR during playback. To compute the sound level, take the average of the *absolute value* of the last 2500 samples. You should add both left+right data to compute the sound level for one sample. The average sound level should be computed after song mixing and any volume adjustments. *Note:* computing the average will probably slow the CPU. Find a way to speed it up so it doesn't affect playback quality: the LEDR must change fairly rapidly in a natural way as the sound increases or decreases in intensity. The LEDR must display intensity as a "bar graph": a loud sound turns on more LEDs.
5. The sound playback should be of high quality without interruptions or gaps.
6. Like homework 2, you must hand in your program electronically. Save your solution as *Z:\eece259\hw3\program.s*. Using **SSH Secure Shell Client** or **PUTTY**, connect and log in to the computer *ssh.ece.ubc.ca* and type:

   handin eece259 hw3

   exit

   Hand in your program *immediately* after marking, or you will be assigned a grade of 0!

PART 3: Project Proposal for Practical Assignment 4

For PA4, you will create your own interesting project. It should involve programming your DE1 board, and it may involve interfacing with other hardware. To get you started, a few suggestions will be given in a list on the web site (it will probably change often).

Before you start working on your project, *it must be approved by a TA or the instructor*. The precise steps are as follows:

1. Think of a few cool project ideas. Talk to friends, TAs, or the instructor. Keep it simple enough you can actually do it on time, but make it interesting!

2. Write up a short 1-page proposal/specification and present it to your TA during PA3.

   Your proposal should be less than ½ page of text. ***Use point form!***

   Fully describe all of the inputs, outputs, and operation of the system.

   You are encouraged to use diagrams or figures to help explain the goals of your project.

   The proposal should *clearly specify* what is being promised (i.e., the deliverable details). Think of it as a *contract* between you (vendor) and the TA (customer). The customer needs to know *exactly* what he is going to get for the money he gives you (grades). Note: the contract must be *complete enough* that if another group in 259 saw it, they could build the *exact same thing* and the customer could not tell the difference without "looking inside" at the source code etc.

3. The TA will give you some verbal feedback. You will probably have to rewrite your proposal, or you may have to choose a different topic. Get the form back to the TA for signing.

4. When the proposal is exactly perfect, the TA or instructor will sign your Proposal Form. **You must keep this signed form (the TA does not need a copy).**

5. A previously *Signed Proposal Form* is a requirement to get marked in PA4. It indicates the TA/instructor agreed the proposal is complete, unambiguous, and solvable in the given time.

6. The *Signed Proposal Form* is a formal contract. Do not lose it. You must present it to your TA again during the marking session for PA4. The TA will mark you based upon *how well you meet your prior commitments given in the contract* as well as *how well you exceeded them*. Bonus grades will be given to projects that significantly exceed their contracts.

A blank proposal/specification and contract form will be placed on the course website.

A sample form appears on the next page.

UNIVERSITY OF BRITISH COLUMBIA
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
EECE 259: Introduction to Microcomputers
Practical Assignment 4: Project

Sample Proposal/Specification and Contract Form

Students: _____ and _____

Project Description

EXAMPLE: DE1 Alarm Clock

- HH:MM on hex display
- SW9=1 activates *test mode* that runs 60x faster, *i.e.* MM:SS on hex display
- SW8=1 buzzer (square or triangular waveform), SW8=0 music
- KEY1 advances minutes
- KEY2 advances hour
- KEY3 displays Alarm time
- Holding KEY3 and pressing KEY2 (hours) or KEY1 (minutes) sets Alarm time
- During Alarm, sound will play for 5 minutes (5 seconds in test mode), then quiet for 5 minutes, then play for 5 minutes, etc
- During Alarm, pressing KEY3 will snooze for 5 minutes (5 seconds in test mode)
- During Alarm, changing any SW from original position ends the Alarm

By signing below, the TA/instructor agrees that this proposal is complete, unambiguous, and of an appropriate level of difficulty to be solvable in time for PA4.

Signed: _____ Date: _____