

A Live Digital Multitrack Instrument/ MIDI-Controller/Recorder Synth

Olivia Tang

*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
via@mit.edu*

Rey Li

*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
ryli13@mit.edu*

Abstract—We propose a design for a live multitrack recording digital synthesizer. With three inputs and three output options, our design aims to create a playable and expansive instrument for users. Two key features include the ability to record and playback different tracks simultaneously and to play high quality sound files (48 kHz 16-bit) for audio output. In addition to PDM output through the audio jack, our system also outputs MIDI which can be hooked up to a computer. The synthesizer also hooks up through HDMI to a monitor to provide a visual interface for a clearer user experience.

Index Terms—component, formatting, style, styling, insert

I. DESIGN GOALS

We are targeting two use cases; for our minimum viable product we are aiming for a project that is able to work as a live instrument for playing music, but as a stretch goal we hope to create a working synthesizer that can be used for music production with a computer. Its major instrument modules aim to fulfill our first goal, where our synthesizer should be capable of being practically demonstrated through a live performance, where it should be able to do everything we've implemented live with good sound quality. Additionally, the track memory, HDMI display out, and MIDI computer out modules attempt to fulfill our secondary goal of being a tool for music production with the emphasis for flexibility and intuitiveness.

II. SYSTEM OVERVIEW

A. Capabilities

Input to the Digital Multitrack Instrument/MIDI Controller/Recorder Synth consists of an offboard joystick, three offboard buttons, twelve of the sixteen onboard switches, and all four of the onboard buttons. The offboard inputs serve as input for playing the instrument, while the onboard inputs serve as settings to determine the logistics of the playing instrument.

The multitrack synthesizer contains 8 different tracks, each corresponding to a different instrument. These different tracks are fully independent of each other and can be sounded all at the same time (8 note polyphony). In total, the onboard inputs decide what tracks/instruments are audible, what track/instrument the current user is playing, whether the current user input should be recorded to memory, should the tracks be playing or paused, and should the synthesizer go back to the beginning of the recording. Instrument data and

sample song data is stored in an SD card, which is booted by pressing button 1 on the board.

To aid usability, the state of the system can be displayed on an external monitor through the HDMI port. The synth also provides a MIDI out port which allows users to use the device as a direct MIDI controller and to export the recorded song to the computer in a DAW friendly manner.

B. Overall Design

The primary pipeline carries data through a decoder, a track manager, and then both the audio constructor with PDM for aux output and the MIDI out for MIDI over USB output. There are additional top level modules for SDRAM management, SD bootloading, and HDMI out. See Fig. 1.

For discussion, we categorize the core system for this multitrack digital synthesizer into the following subsystems: input translation, memory, and outputs. The input translation decodes external inputs to instrumental values and tracks the system's state. The memory section manages the loading and playing of audio clips and the recording and playback of different tracks. A microSD bootloader is used to load data into SDRAM. Finally, there are three outputs: a PDM output signal through the audio jack, MIDI data transmitted by UART, and a HDMI display of the system's interface.

III. INPUT TRANSLATION

A. ADC Communication

To read the joystick's X and Y values, we use a MCP3008 ADC through SPI communication, and then convert it to digital 7-bit values for our input decoder module. This module largely is based on Lab 2, containing the SPI communication and counter, though modified down to read only two values rather than three. Originally, I had coupled the joystick's xy position and also scaled it down, though I realized a more accurate measurement was not costly and more usable in future modules.

B. Input Decoder

The input signals, 3-bit external buttons, which we'll be referring to as valves from now on, and 7-bit joystick x-y positions, are first debounced. Then, they are converted into instrument parameters for the next modules: a 8-bit pitch and 8-bit volume. The translation methods were largely inspired

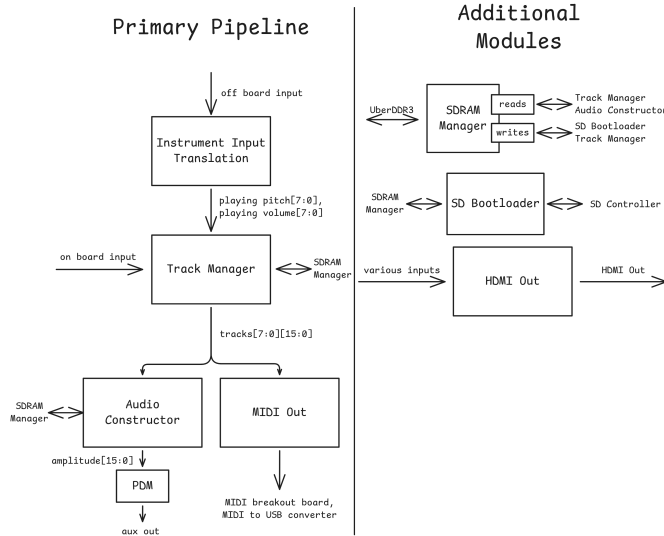


Fig. 1. Overview of the full system

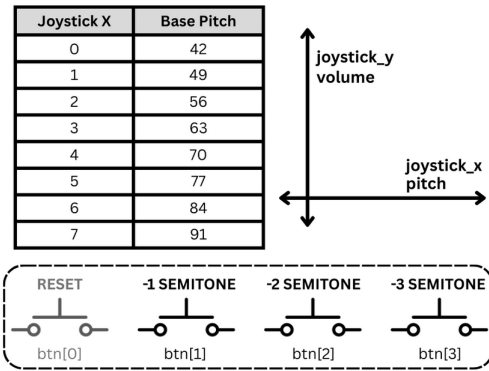


Fig. 2. Inputs to pitch mapping

from playing a trumpet. Similarly to selecting a note on the trumpet, the joystick's x-position gives a base pitch, and three buttons subtract one, two, or three semitones each depending on which ones are pressed. The volume is dependent on the joystick's y-position (see Fig. 2).

IV. MEMORY

A. SDRAM Management

The memory system for our project is quite complex. This is because of the following requirements our design must meet:

- Being able to play through 8 clips at 48 kHz 16-bit quality simultaneously, possibly up to 8 times the original speed or even faster.
- Being able to load 7.3 MB of instrument audio clip and sample song.

The first requirement practically necessitated the use of the SDRAM due to its large storage and throughput. The second requirement, then, is satisfied by introducing a microSD card bootloader, which loads data into the SDRAM on startup.

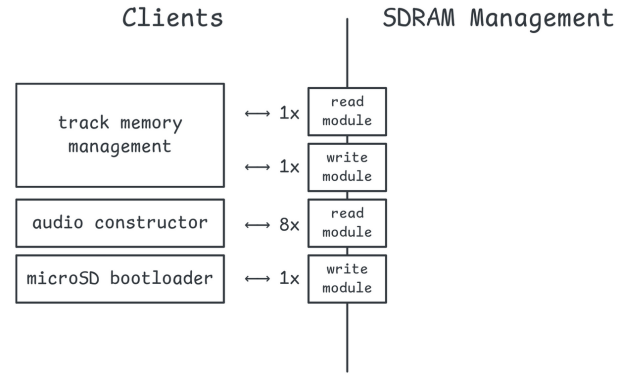


Fig. 3. Client view of interacting with SDRAM



Fig. 4. The interfaces that clients of the abstraction use

Bootloading 7.3 MB takes 15 seconds from FPGA load. It only takes 11 seconds if not booting immediately, because the DDR3 and microSD controller spend 4 seconds booting up before being able to read and write data. The failure mode for forgetting to boot or playing while booting is safe—the audio output may end up sound garbage, but no state becomes fundamentally broken. This load time was deemed a workable tradeoff in the face of the massive throughput benefits and simplicity that comes with using SDRAM over reading directly with microSD. Reading straight from microSD was considered, but its throughput was calculated as barely sufficient for typical use and insufficient in worst cases, it could lead to potential latency hiccups on clip changes, and arguably more complicated to implement.

To achieve these requirements while aiming to minimize complexity of use and complexity of implementation, a level of abstraction is introduced; other parts of the design that wish to use SDRAM functionality do not interface directly with the SDRAM or its controller, but instead interface through *single track SDRAM modules*, of which there is a read variant and a write variant. These modules have basic interfaces with strong guarantees, and can be easily created when implementing new functionality that utilizes the SDRAM. Clients of the SDRAM then can ignore the complexities that come with sharing a single SDRAM controller. Currently there are three clients; the microSD bootloader, the track memory module, and the audio constructor module (see Fig. 3).

1) *The Single Track SDRAM Module Interface*: The entire memory system was designed to be as correct and robust as possible. In this vein, simplicity of implementation is deemed a most important quality. This is because debugging the entire memory system in practice is difficult (hard to do outside

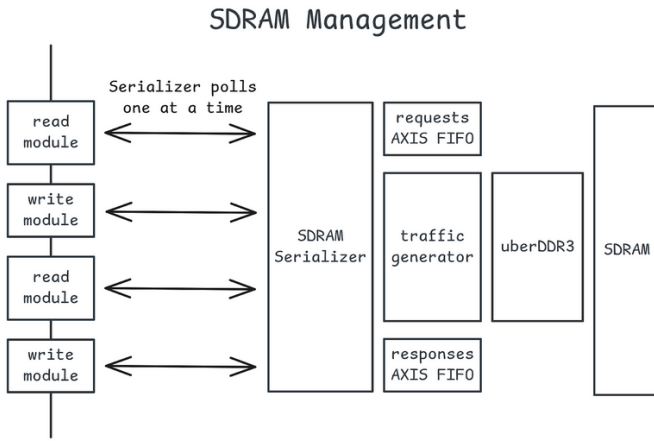


Fig. 5. Implementation diagram of SDRAM management

of hardware), so the shorter and smaller the code, the better. Additionally, if greater functionality is desired, this is instead implemented as a separate module on top of the single track SDRAM module interfaces. Thus, the single track sdrum module interfaces are very simple; addresses are in their base 24 bit form and data is in 128 bit words (see Fig. 4).

For a client to use a read module, when `read` is high and the module is not `full`, a request for the 128 bit word at `req_address` is sent. At some future point, `valid` will go high along with the desired data and the original address.

The write module is very simple; when `write` is high and the module is not `full`, `write_data` will be written to `write_address` with some latency.

As long as the total use of memory does not put pressure on the system, a module will be full for at most 11 cycles (the total number of modules). If the total use of memory is greater than what the memory system can handle, the system is built to gracefully become backed up and communicate being full to clients until unclogged, but this functionality has not been tested.

2) *Implementation:* The overall concept is simple. Every SDRAM read/write module is connected to a single `sdrum_serializer` instance. When a client performs a request to an SDRAM read/write instance, the module holds onto that request and then indicates `full` to be true—modules hold at most one request at a time. While this happens, the `sdrum_serializer` module constantly polls through every module one by one. If it sees that a module holds a request, this request gets placed into an AXIS FIFO for serialization and clock domain crossing, along with an ID to indicate the source module. This ID is ultimately used to return read requests to the correct module. Once serialized, the remaining logic is standard; a pair of AXIS-FIFOs to handle clock domain crosses, and a traffic generator to communicate with the UberDDR3 controller and keep track of pending requests (see Fig. 5).

B. The `mem_requestor` Module

A common need throughout the system is to maintain a FIFO of memory requests, which uniquely have a pending state. This is done by implementing a ring buffer with *three* pointers; one for write/request, one for awaiting response, and one for read/consume. This module is used as the traffic generator with the DDR3 controller and as a buffer for the track manager with a read module. This would be a perfect fit for clip loading inside the `single_track_audio_modules` (part of the audio constructor), but I never got around to switching over.

C. `microSD` Bootloader

The microSD bootloader copies over a large continuous range of memory from the microSD card to the SDRAM through a single track write module. Thanks to Jonathan Matthews for the original SD controller code. Modifications were made to the code to be able to accept a 100 MHz clock to avoid needing another clock domain.

D. Track Manager

The track manager is the primary handler of state in the system, having inputs for what tracks should be active, what track to play on, and whether to record, to pause, or to go back to start. The main output is track data—at this very moment, what is each track doing? This data is MIDI-esque, simply recording how a note is played as pitch and volume, but with no attachment to timbre or instrument. It also outputs how many seconds into the recording the synth is and if it's actively playing (indicated by LED 0).

One design choice is how to encode tracks. We choose to do a simple downsampling at 240 frames per second for each of the 8 tracks storing 2 bytes of information each (exactly 128 bits per frame), for 10 minutes, costing 2.3 MB total (less than the size of audio clips, which is about 4 MB). The microSD bootloader does write track data on boot, which provides the sample song—one could imagine a simple extension to save songs back to the microSD card. The sample song was hard-coded in python for this specific device.

One track read module is used to read ahead and buffer the upcoming frames, and one write module is used to write frames back to the SDRAM.

V. OUTPUT

A. Audio Constructor

From track memory, the audio constructor takes this information and translates it into an amplitude for output. The primary logic is stored inside of `single_track_audio_modules`, which given an `is_playing` and pitch input, determines what amplitude should be output. Volume of each track is then applied afterwards, and the shifted sum of each track is output.

In order to play different pitches of specific instruments, audio clips are played back at different frequencies; e.g. a C4 piano note will sound like a C5 piano note if played at twice the frequency. The base frequency of every clip is 48

kHz, but a reference BRAM is used to determine the correct frequency for playback given the pitch difference between the recording's base pitch and the desired pitch. A single clip is often not enough to cover an instrument's entire range of pitches, since the further away from the source 48 kHz, the less realistic the sound is. Thus, multiple clips are used—this also is what allows for non-melodic instruments like drums. Ultimately, what clip to use at what pitch and the parameters of the clip (start address, end address, whether or not it should loop, which address to loop back to) is handled as hardcoded parameters to these modules. The complete memory of instrument clips with their corresponding addresses was also created by hand.

Each `single_track_audio_module` has a finite state machine to also handle if the current note is being actively played, if it's just been released (don't want amplitude to immediately cut out but fade cleanly), or if the instrument is idle.

Something silly I want to share is that for handling the non-looping case, it's actually coded the exact same as looping except once a clip reaches the end it gets teleported to an address which never moves and only returns 0.

B. PDM Output

The amplitude from the audio constructor is output as a PDM output signal through the audio jack, which sounds much higher quality than PWM.

In practice, when instruments fade out, weird artifacts can occur. This was determined to be PDM artifacts for amplitudes near 0. To solve this, if the amplitude is less than a certain amount (about 0.1% of full volume) for a full 0.1 seconds, the PDM treats the amplitude as 0. This solves the problem when perceiving the output through speakers, though with headphones artifacts can still be heard.

C. HDMI Display

The FPGA outputs an UI overviewing the system to a PC display through HDMI. This module was conceived to provide a more user-friendly interface, as users would be able to learn its functionality without relying on clunky explanation tables. As shown in the mock-up (see Fig. 6), the display includes a generated output waveform of the audio output, render joystick location, and information on tracks, parameters, and joystick axis.

The dual-port BRAM from Lab 5 was used to buffer the data between the top level that is getting clocked at 100MHz and the HDMI modules that are clocked at 74.25MHz, pixel clock. Originally, the logic was simple enough that just passing the data through was enough so there was no need for the BRAM, but upon attempting to add the waveform display, it errored out with set-up and hold-time issues. This helps conclude that one can sample at 1kHz but cannot clock at 1kHz. Thus, the BRAM is added, though only one address for a `RAM_WIDTH` of 99-bit was used since we need only the inputs every cycle.

There are three submodules underneath the HDMI module. In the top level, the HDMI display module takes in the

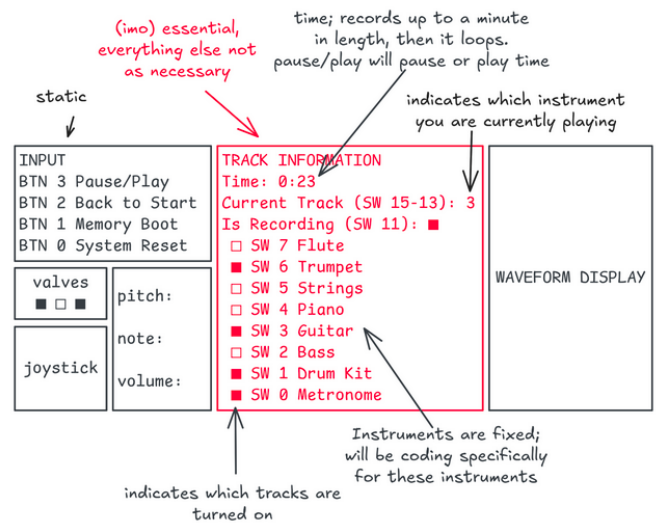


Fig. 6. Mock-up

following from the input translation modules: state, joystick positions, buttons, pitch, and volume. Using the pixel clock line, video signal generation, TMDs encoders and serializers from Lab 4, the visual display is generated through the HDMI UI generator file. The exact sizing of the interface was originally designed and drafted up in Python using the `kivy.graphics` library, before switching over to a Verilog implementation.

At the moment, the interface generation code is completely combinational. Given the horizontal and vertical location of each pixel for a 720x1280 display, it locates the pixel to the subsection to be rendered and sets its associated red, green, and blue values. On the left, the waveform of the audio PDM signal output is displayed in a timescale.

As the HDMI pixels are clocked at 74.25MHz, we need to downsample this as updating the waveform data buffer every frame is too fast. Thus, we ended up only updating the waveform data buffer with a new value at 1kHz. In total, the waveform data buffer stores 100 8-bit amplitudes. It definitely could be rewritten better, I originally planned it as a circular buffer, but I do not think I have the time to do so. At the moment, every value is shifted by 1 bit every time it updates, but incrementing read and write pointers would be a more efficient option. Originally, the waveform buffer was much smaller, but the wave was not as representative as desired.

The track information presents which tracks are currently selected in the track memory, the function information depicts those enabled from the state manager. Finally, the joystick x-y position is depicted with its translated pitch, volume, and note next to it.

The algorithm to render the pixel words was separated into an individual pixel rendering submodule. Using the horizontal and vertical offsets set in the UI generator, this submodule assigns it to the text that is meant to be there, and then uses the font ROM submodule to check for the pixel's rgb values. The



Fig. 7. An actual image of the HDMI output

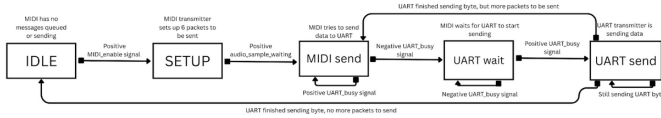


Fig. 8. MIDI FSM

font ROM uses Daniel Hepper’s 8x8font Github repository for its bitmaps, though each pixel letter is scaled up to 16x16bit in pixel rendering.

D. MIDI Output

For this module, we generate a MIDI output from the FPGA through UART based on whether MIDI is enabled in the state manager. Externally, it’s connected to a MIDI breakout board’s TX pin, then the MIDI to USB converter, and then to a laptop. From there, the Digital Audio Workstation application can register the MIDI inputs in. Online MIDI sequencers are not useful as they require to be connected to a MIDI device first, rather be able to accept MIDI packets straight.

MIDI is transmitted over UART, with a BAUD rate of 31250 bits per second. MIDI protocol requires three packages to be sent each time: a status byte and two data bytes of note and velocity. Each byte has 8 bits, with a start and a stop bit. Bytes are divided into two types: status and data. The MSB dictates the type, with 1 being status, and 0 being data. The rest of the seven bits are the actual information. It ranges in value from 0 to 127 in decimal for data bytes and between 128 and 255 for status bytes. MIDI has no start-up protocol, so you can start just by sending off messages.

For this module, I wrote a small state machine: IDLE, SETUP, MIDI SEND, UART WAIT, UART SEND (see Fig. 8).

Since the track memory module outputs all eight channels’ tracks, with only their associated pitch and volume information, another signal is needed for transmission. A module before the midi transmitter is needed to evaluate the tracks output to see if there was a change on any of the channels, as the transmitter should only send MIDI messages when there was a note change on a channel. Thus, the midi controller module was made to compare each channel’s current pitch and



Fig. 9. Scope shot of the MIDI packets sent over UART

volume compared to their previous, and enables transmission if yes. It also takes in a `midi_busy` input from the midi controller so it only queues a new MIDI message if the midi transmitter is not busy.

In the setup stage, after receiving a positive `midi_busy` signal, there are six packets for each track update. The first three are to signal the NOTE-OFF status for the previous pitch and volume on the channel, and then a NOTE-ON message for the current pitch and volume. This way, the previous note is actually turned off, as there are no chords on the same channel. The status byte commands are hard-coded in, while the data bytes depend on the controller for each track’s current and previous pitch and volume values. It then transitions to the MIDI SEND state where MIDI sets the UART data with its packets and positive valid signal.

The reason the UART WAIT state is necessary is because the UART transmitter is much slower than our main clock. Thus, a flag was added to indicate this state so that the messages in the UART data are not changed before they get sent. Otherwise, the `uart_data_in` gets updated by the MIDI SEND before the `uart_busy` flag is raised by the transmitter and the previous data does not get sent at all.

From the DAWS application on the laptop, the MIDI input was received and translated in its function. However, notably, there was a bit of junk data, likely due to transmission errors. The MIDI controller works as intended, as MIDI packets are only sent during track changes, but occasionally the laptop receives contaminated packets. With future work, one could likely use status bytes as message delimiters to filter these contaminated packs out. The other option I foresee is sending NOTE_OFF messages for every pitch except for the current. This seems excessive, especially since there are a lot of pitches to correct for.

VI. SYSTEM WIDE EVALUATION

For our modules, we qualified the latency as clock cycles and throughput as visible throughput. We felt that we could not obviously detect a delay in the system’s audio or visual response when changing an input, which is a quality we definitely did not want in our digital synthesizer. In fact, the

primary latency is likely 5ms due to the debouncing (5ms = 50000 cycles).

Our longest delay is with our initial BOOT sequence for the SD card, as it takes 11 seconds. This delay is justifiable as it only occurs at the beginning and gains us a lot of memory for our tracks. Additionally, the SD card can only boot in 512 groups of addresses. The second longest delay would be our MIDI module, as it needs 13 cycles to fully send its messages and the transmitter is clocked at the lowest baud rate of 31250 bits per second, so each byte takes 256 microseconds to send. Thus, the MIDI transmitter does not send every change in our system, but we are restricted to that timing due to the MIDI protocol's requirements. After this, the HDMI would be the longest pathway, with 13 clock cycles with its pixel clock frequency at 74.25mHz. Again, this is a restriction due to the HDMI clock itself. By restricting ourselves to these timing delays, we are able to generate our HDMI output and MIDI outputs.

For the waveform downsampling, I ended up running into setup and hold time constraints due to trying to clock at a lower frequency. The 100 8-bit waveform buffer shifts 99 8-bit waveforms every time it updates itself, so I wanted to avoid doing this expensive logic every frame and downsample as much as possible. This does result in us having several pins in the HDMI module with tight setup and hold constraints, but the reduced frequency of waveform buffer writes I felt was worth it.

Overall, an area of concern is that our HDMI module causes us to fail timing for setup and hold times, which was something we knew. The rest of the system makes timing well.

We use two block RAMs in our code: a single port ram in the `single_track_audio_module` for track manager and a dual port ram in the `top_level` for HDMI. The first BRAM has a RAM width of 32 bits and 256 entries, which was the default. The second BRAM has a RAM width of 99 bits and 32 entries. For the dual port module, it can definitely be optimized further, as only address 0 is used as we are only passing in the inputs and nothing else. Thus, 31 entries are wasted. We have no DSPs in our system.

Our project can be used as a multi-track digital synthesizer, MIDI controller, and recorder with its multiple functionalities. Someone can use each of these aspects on their own, or use it as a complete system. As shown through our sample song demo, we already allow the ability to load in a complete song if you do not want to compose it on the fly. For our project checklist, we have almost met our ideal goals: input decoder, track memory, audio constructor, HDMI display, and MIDI output modules. We ended up removing the state manager as it was not necessary since we could just pass in the switches instead. We did achieve one stretch goal of moving buttons off the FPGA, though that was more out of necessity as we needed additional input options for our memory modules. We did end up dropping the reverb option from the audio constructor. Our design is fully integrated and functional.

With minimal changes to our design, I think we could add more inputs to our system, as there are a decent amount of

I/O pins unused. Additionally, a Fourier transform could be applied to the waveform image generated on the HDMI output. It should also be simple to add saving to the SD card.

In addition, the MIDI transmitter could be revised to be more resilient to UART transmission error, which unfortunately is not uncommon. For instance, we could use status bytes as delimiters and validation for messages. Additionally, another fix would be adding a timeout interval between data bytes. However, the latter fix is more tricky in deciding the time interval, as what is the difference between a mistake NOTE ON versus a long NOTE ON? More functionality would be useful for our MIDI controller, as we could add additional messages like Channel Pressure and Aftertouch.

VII. IMPLEMENTATION INSIGHTS

A. Olivia

For all of my modules, I ended up rewriting them at least once, with the more complicated modules going through several complete revisions. For the larger changes, it often came after I did more research or took a few days to think about my implementation. It would have definitely been faster and better if I did the thorough research ahead of time, and planned out my implementation more instead of knowing briefly what I wanted.

I made minor mistakes that made debugging difficult as I assumed correctness for certain aspects. Instead, I should have double checked datasheets such as the MIDI breakout board requiring a 5V supply instead of 3V3 or hooking up the MIDI out connector instead of the MIDI in.

Oftentimes, I would decide or realize my initial implementation was incredibly inefficient or excessive after my thought. For instance, for the HDMI module, I completely rewrote my modules for the interface, and then rewrote it again to include the dual-port BRAM after failing timing. When I originally coded the letter rendering, I had thought of bitmaps for each letter, but I had manually placed each letter at a specific xy. This felt tedious and repetitive, so I ended up doing more research and learned about font ROMs, which I ended up implementing instead and that helped structure the code into several submodules instead of one large UI module.

Instead of a large module, having these multiple smaller submodules with a wrapper module is useful for code isolation and specialized testing. I tried my best to write most of my code within my modules, so that we did not need to merge file differences as often, especially if we had very different implementations for shared modules in mind.

In addition, rewriting modules completely was helpful in figuring out bugs, as it would work on the second try. A lot of my errors were caused by incorrect bit widths, as I would start writing and set a random bitwidth for variables since I hadn't decided it yet. However, this would often bite me later as my counters would not be able to reach their period or signed numbers became unsigned due to larger bit widths. Additionally, it also causes any math to result in incorrect values due to overflow or underflow. I realized how important it was to have a standardized bitwidth for everything, as well

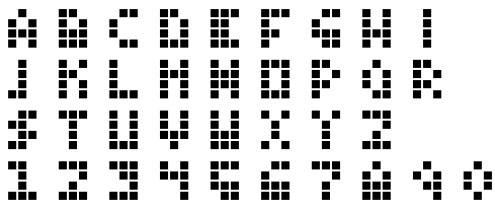


Fig. 10. 3 x 4 bit custom designed font

as having these bitwidths share factors. Additionally, I had to be careful of mixing combinational and sequential logic as next cycle updates often caused me to be one off in my code.

For instance, I originally made my own 3x4bit font (see Fig. 10), but I realized the importance of choosing mutually divisible numbers for xy-positions, spacing, and size. This way, scaling pixels up and down does not result in remainders, which can become a problem with edge cases and out of bounds indexing. I originally prioritized having as small bit widths as needed, but I realized the space needed to store these values was not an issue, instead it was the need to constantly scale them to each other, which left room for mistakes.

I also struggled with integrating modules of different clock cycles, as my default assumption of everything being there next cycle for both is incorrect when one is much faster than the other. This happened primarily with MIDI, where I noticed various bytes were not being sent due to the input data being changed before it is buffered in.

Also, I realize I need to start implementing bottom-up instead of top-down. Oftentimes, I have written the entire module and proceeded to test individual submodules, but I think it should have been the opposite. By starting with writing and testing each submodule, I can ensure correctness and also use that information for the other modules. The top-down approach also caused me to forget minor details such as needing to debounce the inputs, which Rey reminded me of.

I'm glad Rey and I both validated each of our individual modules first, and integrated our respective systems. This allowed our overall system integration to be extremely smooth other than adding additional modules to convert her outputs into the outputs my modules were expecting, and vice versa.

Another problem I individually encountered was I am not good at git, so it was helpful that Rey could work on a separate branch while I freely committed to main. However, I would lose code sometimes and have to reimplement it because I forgot to commit it, which definitely wasted time, or I would commit broken code or files I didn't want to commit yet.

B. Rey

For the memory system, the emphasis on simplicity for correctness was the result of attempting and failing to build a memory system with stronger functionality. The redesign was both much simpler to code, actually possible to debug, and utilized significantly less resources in build—overall, just much better. However, this second iteration memory system originally did not provide the original address of read response

data—this was added because it allows for greater robustness and simplicity by client logic.

A huge class of bugs I kept running into was coding incorrect bit widths (e.g. I forget to set the bit width at all, or I accidentally made an address variable the length of a data variable). In many instances I worked with packed arrays because they were less clunky to code with (e.g. allows me to concatenate and assign variables in a quick manner, can reshape easily) which made these bugs much more possible and devastating; I would suggest to use unpacked arrays whenever possible, at the cost of having assigning to them be super annoying, because failing to statically check for this class of bugs is even worse. Ultimately though, I wish Vivado had a way to enforce strict bit width matching, if not for the whole system, but for just my files, so I can have the ease of use of packed arrays while still having the desired static type checking.

Test benches are really great and fast for catching "blatant" errors (e.g. an `==` should be a `!=`). Thus, I recommend running a test bench with very simple inputs—there is no need to check edge cases or correctness thoroughly to get value out of a testbench, and I wish I wrote more testbenches with this specific intent in mind (I overheard a TA saying something to this effect. I don't remember who they are, but I appreciate them). It also provides a much faster syntax checking and compilation, and iVerilog can be stricter than System Verilog in specific instances (e.g. can catch some of the bit width errors described above).

Ultimately, if you write with as minimal state as possible and there aren't any blatant problems, verifying correctness becomes significantly easier. One reason for this is that with less state, there are less possible "incorrect" states and edge cases. As an example of trying to maintain minimal state, in the track memory module there is a read to a specific address and a write that is supposed to be for the same exact address. To ensure this is true and there is never a desync, the write address is assigned to be exactly the read response address (this is what pre-empted the addition of a read response address to the memory system). In a similar-ish way, the lookahead read address in the track manager module used for reading and writing is actually completely decoupled from the time counting signal output to the visual display. Although the code is written so that they should stay in sync, even if somehow they don't, since the two states don't interact at all the system will still proceed in a practically correct manner, and there is code to realign them on restart.

In general, coding for correctness and minimal debugging is very important for hardware. If state does have to get complicated because the underlying logic is complicated (like the single track audio modules), besides abstracting into smaller, reused, systems (which is awesome for ensuring correctness), thinking about if/how the system can course correct into a valid state if the state is somehow invalid is useful to think about. Coding so that even if there are bugs from specific edge case inputs, the system is able to move into a correct state and keep going after being incorrect for some time, makes

debugging much less necessary. Any time in which this is not possible should be mentally noted (and probably also written as a comment). One simple example of this kind of thinking is to ask "what happens if on start up the register is initialized to some random value with no reset" (which I believe is what actually happens on start up, though I'm not sure). This is e.g. why in counter code, one should check if counter $\neq$ period-1 instead of counter == period-1. (In some places I used reg x = 0 if I'm particularly scared of a bad startup state. It's also useful to have a mental model of how wires and regs work as types, even though logic should be used as default all the time (logic gets transformed into wire or reg depending on context)).

Sometimes you do have to just do a complete rewrite. It'll go faster the second time.

Copying and pasting data or transforming data that is in a not very program friendly format to a program friendly format takes a while (creating the instrument clip memory file and the sample song memory file by (mostly) hand took on the order of 3-6 hours each). It might even be worth programming and debugging a converter over doing it by hand. Or just be aware of this time sink and start on everything earlier.

I wish I dropped one of my other classes sooner after realizing I was falling behind for the final project for this class.

Defining a single, canonical addressing scheme to use through the entire system is better than using a 24 bit wide, 128 bit word system sometimes and a 32 bit wide, 8 bit word system other times (e.g. have the SDRAM read and write modules use 32 bit addresses). Accidentally forgetting to convert is another class of bugs that does not need to exist with proper planning.

VIII. CONTRIBUTIONS AND THANKS

Rey researched, implemented, and wrote up the SDRAM memory management, microSD bootloader (including compiling the clip data and creating the song data), track manager, audio constructor, and PDM. Olivia did the same for the output MIDI, output HDMI, and input modules. Both of us evaluated our own modules while helping debug when needed and discussed implementation designs repeatedly to ensure a smooth system integration. We both spent an equal time writing each assignment and making necessary diagrams.

For this project, we would like to thank Joe, the 6.205 TAs and LAs for their help in our project. EDS and Joe helped provide a lot of hardware needed for our project.

Rey would also like to thank Fromixty, the author of Pokemon RSE HD Soundfont Release which was the source of most of the instrument clips. Thanks to errorjones for the metronome clicks. The developers of Polyphone and HxD. Koji Kondo for composing the Gerudo Valley theme and Seventh Sam for his transcription and analysis I used for my arrangement.

IX. GITHUB REPOSITORY

<https://github.mit.edu/6205F25/fa25-6205-team27>

REFERENCES

- [1] dhepper. font8x8: 8x8 monochrome bitmap fonts for rendering. GitHub repository, <https://github.com/dhepper/font8x8>
- [2] HINTON INSTRUMENTS: MIDI Protocol Guide. (n.d.). Copyright © 2001-2018 Hinton Instruments. <https://hinton-instruments.co.uk/reference/midi/protocol/index.htm>
- [3] Instructables. (2017, October 28). Send and receive MIDI with Arduino. Instructables. <https://www.instructables.com/Send-and-Receive-MIDI-with-Arduino/>
- [4] Jonathan Matthews. SD controller code.
- [5] Joy-IT. (2017). KY-023 Joystick module (XY-Axis). <https://naylampmechatronics.com/img/cms/Datasheets/000036%20-%20datasheet%20KY-023-Joy-IT.pdf>
- [6] Wikipedia contributors. (2025, April 10). Circular buffer. Wikipedia. https://en.wikipedia.org/wiki/Circular_buffer