LiTerm: An Independent File Editor, Assembler, and Processor

Tsegazeab Beteselassie
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA
tsegaz@mit.edu

Ziyad Hassan
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA
zhassan3@mit.edu

Simon Opsahl
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA
sopsahl@mit.edu

***Abstract*—We present a design for a terminal-based text editor fully supported in hardware on a Xilinx FPGA. We utilize a independent RISC-V processor that runs assembler code located in instruction memory. Separate from the processor exists a terminal-based text editor that accepts PS/2 keyboard input and allows for dynamic program file editing. We propose a system that performs the reduction of assembly code to binaries entirely on the FPGA. In order to validate our results, we use a data memory-mapped buffer to display the process of bubble sort on a set of elements.**

***Index Terms*—digital design, field programmable gate array, assembly, processor, terminal, RISC-V, PS/2**

I. Physical System

Our design consists of three physical components:

- Monitor: We connect to a 720p monitor by HDMI. The monitor visualizes the text editor and MMO plotting.
- Keyboard: We use a PS2 keyboard connected to the FPGA by cable and a MDFLY breakout board [1]. The keyboard is used for generating assembly code.
- FPGA: We use a Real Digital Urbana FPGA development board provided by the course staff of 6.205: Digital Systems Laboratory. Most of the text editor, assembler, processor, and visualization components are hosted here.

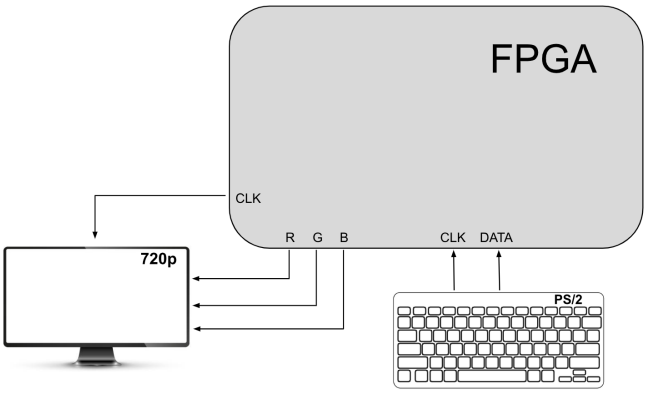

Fig. 1. Overall Physical System

A. Monitor

The 720p monitor communicates with the terminal via the TMDS protocol learned in lab. The relevant signals for the monitor are generated with the signal generation module, which get passed into the terminal controller to coordinate sprite drawing. The monitor displays our terminal at 60FPS.

B. Keyboard

The main pieces of external hardware for LiTerm are the PS2 keyboard and the 720p HDMI monitor. For device-to-host communication, the PS2 keyboard sends data as follows: 1 start bit, 8 data bits (LSB first), 1 parity bit, and 1 stop bit as shown by figure 2. What's also different about this protocol is that the bits get sent on the falling edge of the PS2 clock. Additionally, the PS2 clock is slower than our FPGA's clock; the PS2 clock–depending on brand–runs between 10-16kHZ while the FPGA is 100Mhz. Although we used the HDMI Clock (74.25Mhz), this is still way faster than the keyboard clock. In order to cross this clock domain, we used 2 buffers and sampled the state of the PS2 clock two states back. Thus, we are able to detect a falling edge whenever the first buffer was high and the second was low. This way we were able to use state machine to accumulate the data until we got to the stop bit. At that point we just raised a signal to indicate that data is ready to be sent. We get rid of extraneous keyboard signals(such as scrolling buttons) then that data is sent to the terminal controller.

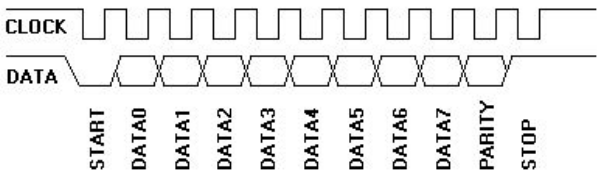

Fig. 2. PS2 Protocol

1) Keyboard FSM: The Keyboard FSM starts with **IDLE** until we the data drops low, which indicates a start bit. This allows it to go the **Processing** stage which we accumulate data until we get to the Done stage which is triggered by a data high after 10 bits.

2) Keyboard Protocol Adjustment: The Keyboard had a protocol of sending in the [Letter][Special Break Character][Letter]. This gave us a bit of trouble since we had

originally only expected one letter at a time. Additionally it could do [Letter][Letter]....[Break character][Letter] depending on how long you held it. Since we didn't have a mapping to the break character it would appear as a space, which was a subtle bug we had. Additionally, the keyboard mechanically was a bit old so sometimes it would hold a button even though we clicked on it once(even with a debouncer module) Due to this mechanical issue we ended up waiting for the [Break character] then took the [Letter] after it allowing us to only type one by one, which allowed us to type faster and not have to deal with physical mistakes as much.

II. Terminal Controller

The terminal controller takes in keypress data and maps spritesheet locations generated by the translator module into ASCII characters. These ASCII characters are what the terminal controller sends to the sprites module to get saved to the memory BRAM, while the spritesheet locations that get sent are used to display the characters onto the 720p HDMI monitor. In addition to this, based on what key is pressed, the terminal controller updates the location of the key cursor. For example, backspace moves the x-value back by one, while enter changes the y-value by one. In addition, the values sent to the character sprites module to be written into the display memory also get sent to the text editor memory. The text editor memory is what the assembler reads from.

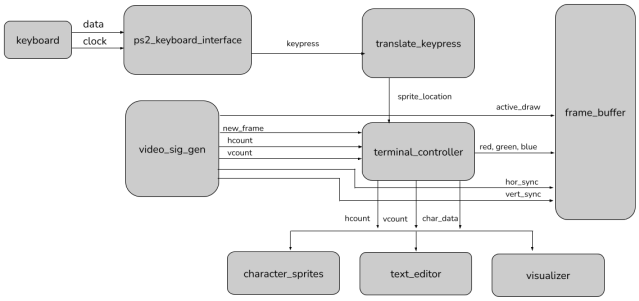

Fig. 3. Terminal and Display Block Diagram

The status of the processor will be displayed on the FPGA using LED lights. Blue is the idle state, where the terminal is still being written to. When the user is ready to compile, the LED will turn either red or green depending on whether their code was successfully compiled. If it was successful, they can also then the MMO can be displayed and the speed of execution will be slowed to show how the states of the register change. For a program such as bubblesort, at each step of execution, the heights of the different registers will change, showing how the registers go from unsorted to sorted.

A. Text Editor

Both the character sprites and the current state of the terminal are stored using BRAM modules. The character sprites are turned into a spritesheet and palletized, with the relevant portion of the spritesheet being cut out and displayed for each part of the terminal. The terminal display itself is split up into a 76x42 grid, with each grid square containing an 8 bit ASCII value corresponding to the character it contains. The grid values are stored in a BRAM and updated with every keypress. To enable scrolling, the terminal grid actually extends past the screen, and the number of times the scroll button gets pressed adds an offset to the BRAM address input, which moves all the values down by one each time.

As the hcount and vcount progress through the screen, the BRAM address to read from gets changed, and the sprite to display gets set based on what grid square it's in. This has a six-stage pipeline due to using multiple BRAMs consecutively. The text editor generally only supports only letters, numbers, slashes and apostrophes.

III. Processor

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Fig. 4. 32 bit instruction template

For our processor we used an 8 stage pipeline Risc-V processor. The ISA that we use for this will be a Risc-v Base Integer 32I instruction set without ebreak and ecall, since we won't be using an OS. This ISA will be shared with the assembler so that when the assembler receives instructions, it'll translate it to 32 bit instructions that the processor understands. This was a 8 stage processor because we had to deal with requesting data from the BRAM which took two cycles so we had to delay fetch 2 extra cycles and consequentially since we had to request data for memory instructions during the mem stages, we had to wait extra cycles for that.

A. Pipeline Stages

Fetch 1 is just request stage where we kept track of which PC the bram is fetching, and whether it was a valid request. **Fetch 2** is a pipelined copy of **Fetch 1** one cycle later. Finally, **Fetch 3** is also the same except the data is now available from the bram two cycle later, so the information(pc, valid, instruction) is passed to decode. **Decode**: the Decode stage is where the instruction is translated it into relevant program information(funct7, funct3, opcode, imm, register reads, etc.) Decode is also where registers are read and if there's any hazards indicated by a flag down the line, the processor stalls here. **Execute**: During this stage the processor determines any jumps and branching, additionally it performs ALU instructions. If there are any jumps or branches execute sends out a signal to annul the instructions after it. **Memory**:This is where we request memory from the bram for any memory instructions, we wait two cycles and we get to writeback, if there are none we don't request anything. **Writeback**: Once we have our

memory data we are able to store it in a register or write back to memory. Here there's a potential hazard, so if there's an instruction that uses a memory location that is the same as the one being edited here(specifically Store instruction), it stalls the instruction up the pipeline.

B. Instruction writing

Since we won't be instantiating our processor with the program information we needed a way to write to the instruction memory. So what we did was give the processor an **Instruction Write Enable, Instruction Write Data, Instruction Write Address** ports, which allowed outside modules to write to our instruction BRAM. We also prevented our processor from running until we got a signal that the outside module(the assembler) was done writing to the processor.

C. MMO Data Writes

Since we had to also visualize our memory we decided to have the visualizer have it's own BRAM instantiated to the same dataMem.mem file as the processor, and we added 3 Output ports to the processor. **Processor Write Enable, Processor Write Data, Processor Write Address**. We connected these ports to our store signals, so whenever we stored something back in our data memory, we also sent it out to be written to the screen.

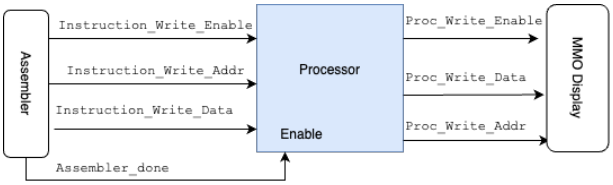

Fig. 5. Processor interaction with other modules

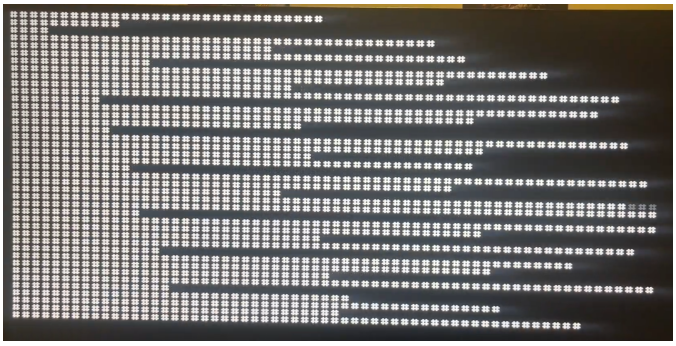

Fig. 6. (Unsorted) Data Memory Mapped Visualization

D. Visualization

In order to visualize the results of the program being run, we take the first 42 values in data memory and create a bar graph out of them. This uses the same display system as the terminal, so each memory location is represented on the y-axis as a 16 by 16 grid location. The heights of that row, going along the x-axis, represents its current value. We use a hashtag as the unit of measurement for the bar graph: for example, three hashtags for memory location one represents the value 3. We wanted the processor to run slow enough so that we saw the visualization. So we did some calculations and we got that in order to see it using our eyes slowly we would need about $1,000,000$ cycles of the HDMI Clock. We successfully down cycled the processor by setting an artificial clock that went high whenever was $< 500,000$ / Half Clock Cycle and low otherwise. We used this artificial clock as the processor's clock. An example visualization of this is shown in Figure 6 and 7.

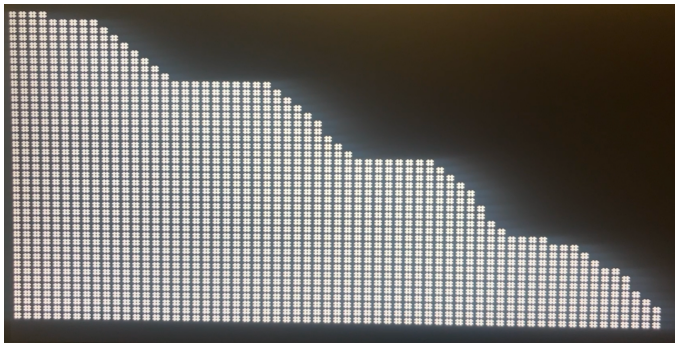

Fig. 7. (Sorted) Data Memory Mapped Visualization

IV. Assembler

The assembler's goal is to integrate the code written into the text editor into the processor. When we trigger the assembler with `sw[1]`, the assembler maps the code to a set of instructions, converting the values into binaries that are written to Instruction Memory (IMem). Our control flow depends on a variable titled `assembler_state`. Each state coresponded to a specific mode of operation:

- `IDLE` : Keyboard input updates the text buffer.
- `PC_MAPPING` : The assembler is iterating through the text buffer and storing the PCs of each instantiated label.
- `INSTRUCTION_MAPPING` : The assembler is iterating through the text buffer and writing translated instructions into IMem.
- `ERROR` : An error was encountered in assembling the code. Changes must be made before code can be reassembled.
- `SUCCESS` : The code is successfully assembled and written into IMem. The processor is now executing.

The control flow is either triggered by `sw[1]`, which signals a transition from `IDLE` to `PC_MAPPING`. Once we fully iterate through the text, we transition to `INSTRUCTION_MAPPING`, and then again to `SUCCESS`. If an error is encountered at any point to the assembly, then we transition to `ERROR`.

When we assemble the code in the text editor, we rely on several design choices:

- All registers must be in the format `r__`.

```
.start.
    / Initialize base address and array size
    addi r01, r00, x1           / x1 = 1 for decrementing
    addi r10, r00, x0          / x10 = base_addr = 0
    addi r11, r00, x02b         / x11 = n = 10 (array size)

    / Initialize outer loop index i
    addi r12, r00, x0          / x12 = i = 0

.outer.
    sub r31, r11, r01        / x31 = n - 1
    bge  r12, r31, .end. / if i >= n - 1, exit
```

Fig. 8. Example of Correct Syntax

- All immediate values must be hexadecimal and preceded by `x`.
- All text after `/` in a line will be ignored.
- Labels must be in the format `.LABEL.` and the length of the label must not exceed than 6 characters.

These design choices ensure that the assembly step proceeds consistently and limit the amount of syntax difference that needs to be supported. If all these design choices are followed, and the instruction syntax obeys what is to be expected, then the assembler should work. If they are not obeyed, then we will transition to `ERROR` and remain there until the syntax error is fixed in the code. Potential improvements could incorporate specified error handling and increase the syntax support for things like decimal immediate values.

A. Components of the Assembler

The assembler accepts the following inputs:

- `incoming_character[7:0]` : The ASCII values of the incoming character stored in the text editor buffer.
- `new_character` : Pulses high when a new character is ready to be processed.
- `new_line` : Pulses high when a new line is to be sent.
- `assembler_state` : Described above.

These signals are then fed into one of four key submodules corresponding to each of the patterns that need be recognized: instructions (`READ_INST`), registers (`READ_REG`), immediate values (`READ_IMM`), and labels (`READ_LABEL`). The label submodule is also active when we record the PCs of the labels in the first iteration. Each submodule is activate based on the state of the decoding process. If for example, we were reading a I-Type instruction, we would have the following pattern:

`READ_INST` → `READ_REG` → `READ_REG` → `READ_IMM`

B. Submodule Architecture

Each of the four submodules is ordered in roughly the same manner. They are only on when their `valid_in` signal is high. Then, they use a rolling buffer to accumulate the characters until termination or a recognizable pattern in the case of labels and instructions. Then, each module returns data consistent with their purpose:

- `READ_INST` : `opcode`, `funct7`, `funct3`
- `READ_REG` : `reg` (the index of the register)
- `READ_IMM` : `imm`
- `READ_LABEL` : `offset`

Each of these items is placed into the instruction struct, and as soon as the last piece of data for a specific instruction is extracted, the instruction is assembled using all of its components and sent as an output to the module along with a single-cycle-high `new_instruction`.

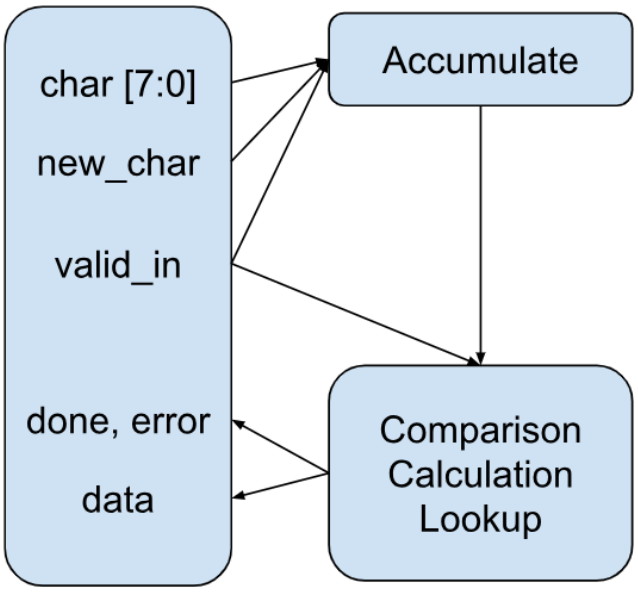

Fig. 9. High-level Submodule Architecture

C. Resource Utilization

The assembler module used a miniscule fraction of the resources available on the FPGA. There was no need for BRAMs, and everything was computed using registers. We made use of 619 LUTs, with the majority in use by the `instruction_interpreter` and `label_controller`, likely for their extensive utilization of registers. I made an effort to minimize the register usage, however, using methods like ASCII compression to store label buffers and high modularity to avoid redundancy.

One domain that could be improved upon is that of clocking. In order to manage state transitions after instruction decoding, I had to downcycle the sending of characters by a factor of 2. As a result, I could speed up the process by a factor of 2 through speculative state transitions. This effect is minimal, however, as the assembler works in a real time system, and 20,000 cycles at a 100 MHz clock is really no different than 10,000 from the application and user side.

V. Reflection

Ultimately, this project was a great way for us to learn a lot more about why people making ISAs like RISC-V make the

design decisions they do. We also learned and incorporated a new set of protocols: namely PS/2 for the keyboard interface. We adapted lab code for the terminal display, and we designed and built a framework for testing the processor and assembler. A few key insights we identified:

- Simulation in cocotb was essential to the success of our project. Being able to gather as much data as possible about why something is going wrong is the first step in solving any major roadblock. I think testing was the only way we were able to finish this project not necessarily that we were good at testing from the get go. We had to learn as we went and eventually got so stuck at times that we got better at it.
- Understanding what your module should do and how it behaves is essential to all testing because at times we were testing behaviors according to how we perceived the system should work, so it worked in simulation because we wrote the test wrong. So any future advice to ourselves or to others is to understand the protocols and the behaviors of the system.
- Be one with the waves. Should you use a simulator such as cocotb, sometimes it has it's own bugs and glitches, which we ran into at times. But being able to look at the behavior through the waves and double check whether it's a simulation error or user error was very important
- As long as the build size did not grow too large, utilizing registers was key in interpretability and portability of the code. Making use of SystemVerilog feature like packages helped greatly in this regard.

As per the last point, we made very little use of the resources available to us on the FPGA. Unless we expand the number of concurrent processes that are available–essentially by building an OS–there is no clear way to make use of more resources. We were not timing-constrained either, as the final WNS was 2.092.

Regardless, in our project we set out to create a system that worked–not as a result of the features available to us on the FPGA–but in spite of them. With our final build, that is what we got. We were able to edit a text file in assembler, assemble any legal set of instructions in our ISA, and visualize the results of a processor executing the assembled code. As a result, we achieved the goal that we set out to achieve for this project.

If we were to expand on our project, we may create support for multiple files and construct the beginnings of a rudimentary file system. We may also increase the syntax support for the assembler so that coding on our setup would feel no different than one's IDE of choice. We might also want to add command line navigation of our program and begin incorporating everything into an OS.

References

[1] Amazon.com: 6P 6-Pin PS/2 PC Connector Signals Breakout Board Screw Terminals PUR Keyboard: Electronics. (n.d.). https://www.amazon.com/dp/B0814K7D7L?ref=ppx_yo2ov_dt_b_fed_asin_title

[2] PS2 Keyboard Protocol https://www.burtonsys.com/ps2_chapweske.htm

[3] PS2 Keyboard Decode Values https://www.eecg.utoronto.ca/~jayar/ece241_08F/AudioVideoCores/ps2/ps2.html#id439258

Contributions

Ziyad Hassan - Responsible for the processor and keyboard
Simon Opsahl - Responsible for the assembler
Tsegazeab Beteselassie - Responsible for the terminal controller and keyboard

Codebase

All of our code is located in Github at https://github.com/sopsahl/LiTerm.