FPGA Securities Exchange

By Evan Yee and Evan Lankford

Brief Overview of Project

PS2 KEYBOARD

|

FPGA---VGA

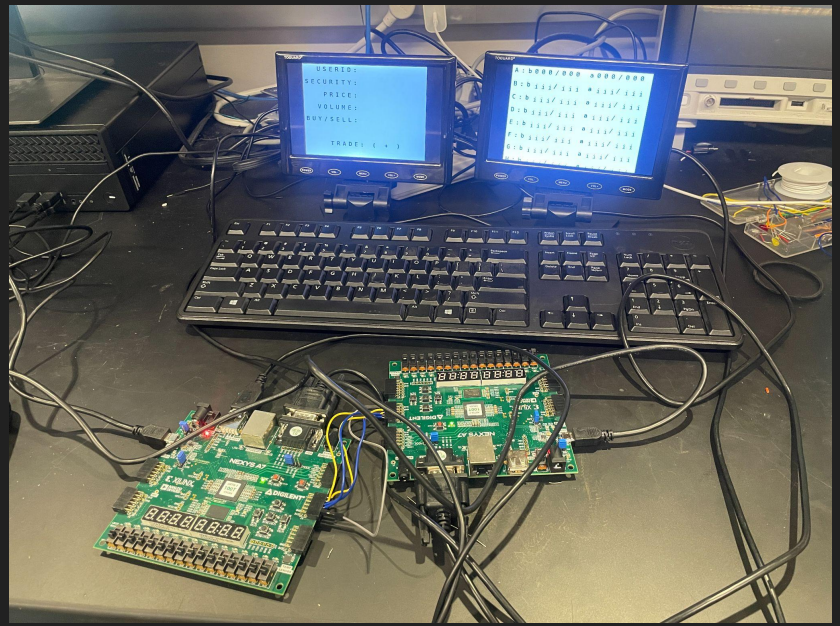
|COM FWD

|COM REV

FPGA---VGA

|

BOOK ALGO

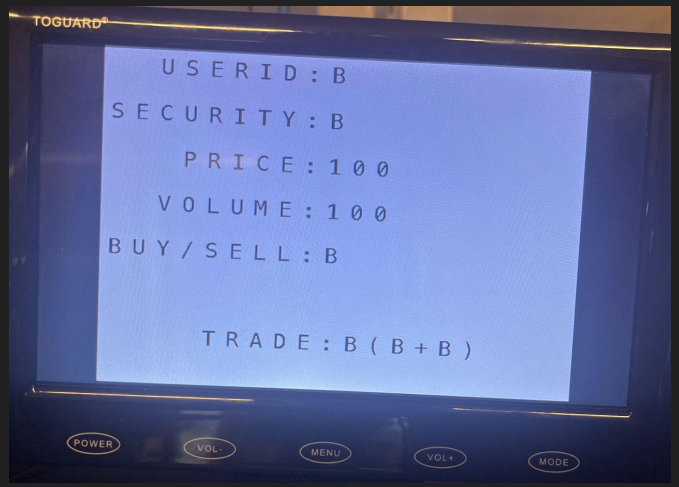


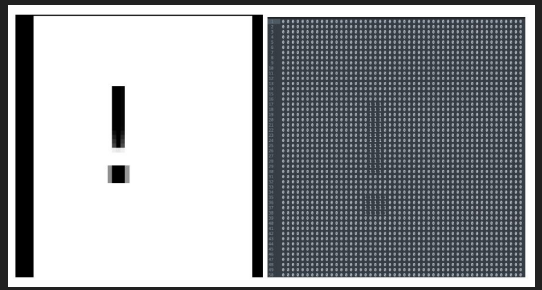
PS2 Interface / Input Parsing

- Used PS2 keyboard interface
  - Latched keyboard inputs and stored enter count
- Converted to ASCII codes with ASCII mem file
- Pixel math to corresponding address in sprite memfile
- Keyboard inputs to order (32 bits)
  - order[31] - buy (1) or sell (0)
  - order[30:28] - security (A-H)
  - order[27:24] - user (A-P)
  - order[23:12] - price
  - order[11:0] - volume

VGA Display on FPGA 1

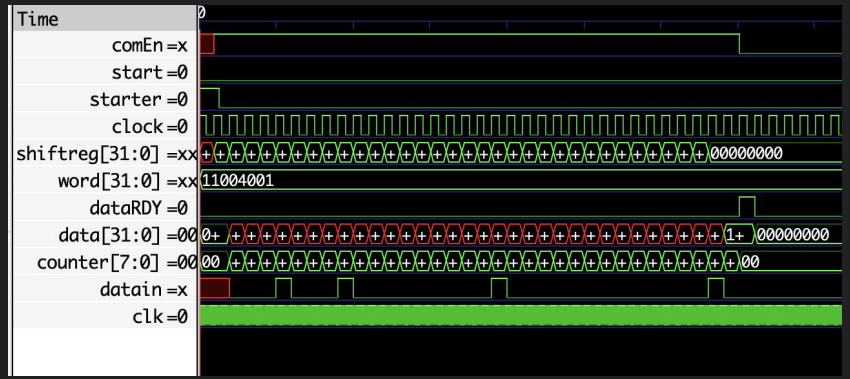
- Memory file divides VGA screen into character blocks
  - Background and static character ASCIIs listed in mem file
  - Variable inputs assigned “1” in mem file
- VGA controller
  - Logic for “cursor” based on current input
  - Displaying previously entered inputs from keyboard
  - Clearing display after executed order
  - 5 input fields, one for most recently executed trade
- Ternary operators





Communication Between FPGAs

- Start signal is initiated on User-end once an order has been completed. Start signal on book-end is initiated when writing to dedicated output execution register
- After 32 slower clock cycles of receiving data (ComEn HIGH) we declare data ready which is latched into our wrappers for one cycle)
- Used JA pins with wires to send ComEn and data between the two FPGAs - downclocked communication and reception due to physical constraints



Sending Data Into Processor

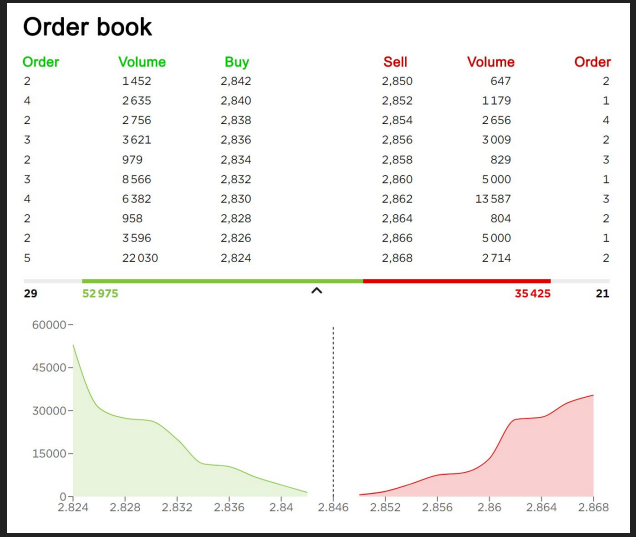
```
wire[31:0] regAReal, regBReal;
   reg[31:0] datainReg = 0;
   assign regAReal = (rs1==20)? datainReg:regA;
   assign regBReal = (rs2==20)? datainReg:regB;
   reg seenRDY = 0;
   always @(posedge clk) begin
       seenRDY = dataRDY;
   end
   always @(posedge ~clk) begin
       if (rwe == 1 & rd == 20) begin
           datainReg <= rData;
       end else if (dataRDY & (seenRDY != dataRDY)) begin
           datainReg <= receiverdata;
       end

   end
```

- Created a “Synthetic Register 20” to provide input data from our wrapper to the processor when encountered a readreg20
- Also allowed processor to clear this synthetic register, so both wrapper and assembly could write to it

Assembly Algorithm

- A Linked-List Structure based sorting algorithm that is broken down into overlap delivery and priority organization
  - Parses user[31], security[30:28], and buy/sell[27:24], then examines top of book to identify potential overlaps until volume is depleted or no overlap exists anymore
  - Then pushes order to sellside or buyside allocation in memory to do pointer math to maintain book order
  - Can give example of expected behavior!




```
#reg29 inputs data from wrapper I/O write
#reg28 stores next open bit address in DMEM for buys
#reg27 stores next open bit address in DMEM for sells
#reg26 SELLAhead
#reg25 BUYAhead
#reg24 output for execution BUY
#reg23 SELL

#r29[31] tells us 1=buy, 0=sell; 1
#r29[30:28] tells us security 3
#r29[27:24] tells us user 4
#r29[23:12] tells us price 12
#r29[11:0] tells us volume 12

#PUT SECURITY POINTERS IN MEMORY

#everytime we hit 0 on vol for our HEADS, move to next as HEAD!!

#we have pointers to our lists in dmem

##
#MEM ORGANIZATION
#0 1 BUY HEAD
#1 2 BUY HEAD
#2 3 BUY HEAD
#3 4 BUY HEAD
#4 5 BUY HEAD
#5 6 BUY HEAD
#6 7 BUY HEAD
#7 8 BUY HEAD

#8 1 SELL HEAD
#9 2 SELL HEAD
#10 3 SELL HEAD
#11 4 SELL HEAD
#12 5 SELL HEAD
#13 6 SELL HEAD
#14 7 SELL HEAD
#15 8 SELL HEAD

#16 1 BUY TAIL
#17 2 BUY TAIL
#18 3 BUY TAIL
#19 4 BUY TAIL
#20 5 BUY TAIL
#21 6 BUY TAIL
#22 7 BUY TAIL
#23 8 BUY TAIL

#24 1 SELL TAIL
#25 2 SELL TAIL
#26 3 SELL TAIL
#27 4 SELL TAIL
#28 5 SELL TAIL
#29 6 SELL TAIL
#30 7 SELL TAIL
#31 8 SELL TAIL

#32 1 Head Buy Val
#33 2 Head Buy Val
#34 3 Head Buy Val
#35 4 Head Buy Val
#36 5 Head Buy Val
#37 6 Head Buy Val
#38 7 Head Buy Val
#39 8 Head Buy Val

#40 1 Head Sell Val
#41 2 Head Sell Val
#42 3 Head Sell Val
#43 4 Head Sell Val
#44 5 Head Sell Val
#45 6 Head Sell Val
#46 7 Head Sell Val
#47 8 Head Sell Val

# BUY A MEM ALLOC
# SELL A MEM ALLOC
# BUY B MEM ALLOC
# SELL B MEM ALLOC
# ...continues
```

Pulling Values from Memory for Output

- Whenever a trade was executed (prices parsed with sll and modified sra) or a new head (low ask or high bid) was identified, we write to specific memory addresses or registers after performing a Binary-to-Decimal conversion with div,mul,sub…

```
always @ (posedge ~clk) begin
        if (mwe==1 & memAddr==32) begin //BUYA
            buyA <= memDataIn;
        end
        else if (mwe==1 & memAddr==33) begin
//BUYB
            buyB <= memDataIn;
        end
        else if (mwe==1 & memAddr==34) begin
//BUYC
            buyC <= memDataIn;
        end
        else if (mwe==1 & memAddr==35) begin
//BUYD
            buyD <= memDataIn;
        end
```

VGA 2

- List of each security with high bid and low ask (price and vol for each)
- Separate mem file with blocks for VGA screen
- Takes inputs from processor regfile
  - Parses inputs to get ascii codes
- Initially wrote verilog BCD
  - Saw lots of degradation, moved over to MIPS
- Struggled with timing

Challenges

- Memory Files
  - Sprites were originally 50x50
  - Changed to 32px by 64px to replace division with shifting
  - Solved initial VGA degradation issue
- VGA Degradation
  - BCD too slow in verilog
  - Communicate/receive in both directions
- Reproducing Testing
  - Often times when we had bugs, they would not be reproducible even given identical input sequences
- Memory allocation pointer sorting
  - Writing our complicated algorithm in assembly often gave us pointer errors or edge cases that would be difficult to debug
- BCD working in MIPS
  - Multdiv slow and BCD from MIPS output pretty buggy

Future Work / Improvements

- Implement multiple user terminals
- Integrate a automatic graphing function to make a price-plot
- Total sale volume displayed on terminal
- Error correction for unreadable order inputs
- Overall nicer UI and display
- Faster Communication for lower latency