COE3DQ5 – Project Report

Group 63

Akash Sharma & Samarth Mehta

shara98@mcmaster.ca
mehtas30@mcmaster.ca

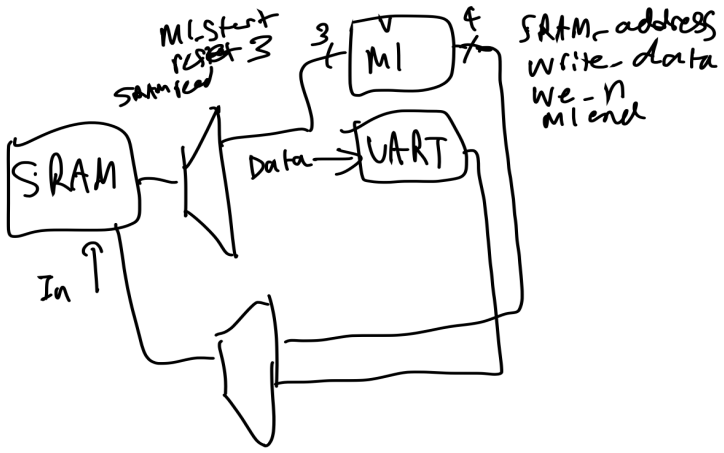
Nov 27th 2022

Introduction

The main objective of the project is to implement the McMaster image compression specification into hardware using custom image decoding circuitry to learn low level hardware design. Compressed 320 x 240 pixel image data is sent to the hardware via universal asynchronous receiver/transmitter (UART) from the personal computer (PC), and ultimately is stored in the static random accessible memory (SRAM). The custom decoding circuitry then reads the SRAM and decodes the data, storing it back into the SRAM once complete. After this process is complete the video graphics array (VGA) reads the SRAM to display the decoded image to the monitor. This custom image decoding circuitry is designed and implemented by the students, starting with a design of the flow represented by a state table. After an initial flow has been represented students can begin programming using Verilog. This Verilog code is tested via ModelSim and two test files that match the pixels generated with the expected pixels of the image.

Design Structure

Our design was partitioned into modules with project.sv as our top level module controlling all the other modules. This top level module connects wires and signals to the hardware modules, ensuring the hardware is correctly designed. Examples of these include the clock signal, resetn, and SRAM_write_data. The other modules initiated by project.sv are PB_controller for the push button unit, VGA_SRAM_interface for the VGA unit, UART_SRAM_interface for the UART unit, and SRAM_controller for the SRAM unit. These modules were given to the students, and initialized all the important hardware required for the project other than the custom image decoding hardware. The image decoder was designed within the milestone module, which is the last module called within project.sv. To get these modules running the top module enables them by sending a high signal through its enable wire, initiating the hardware. This is done in the finite state machine (FSM) of the project.sv file, with each state enabling another piece of hardware. The first few states send enable signals to all the modules provided to us, initializing them with the last state starting up the custom decoding hardware. The decoding hardware chip has 4 inputs, Clock, resetn, SRAM_read_data and m1start. Clock is connected to the clock signal of the chip, resen restarts the chip if the signal is high, SRAM_read_data is the data given by the SRAM, and m1start is the chip enable to control it from the top level. There was an alternate choice, where instead of using a separate module for the custom decoding hardware, the project.sv was used. This can be done by implementing the decoder hardware in states after the first few initialization states were completed. The reason why this method was not adopted was that it keeps it much less organized when it's one big module, in comparison to a couple of smaller and more structured modules with a specific function. This would also be useful if milestone 2 was implemented, as that as well would be a separate module, clearing the chances of any confusion between the two in the future.



Implementation Details
Milestone 1

FSM:

Milestone 1 is essentially an FSM that contains states of reading from the 3 YUV locations, calculating U,V prime and then the corresponding RGB values and finally sending them and repeating this process. It can further be viewed more globally as three states with lead in, common case and lead out states in order. The process of going from lead in to common case to lead out repeats until all the rows have been filled with the objective of completing the conversions for 240 rows. The lead in consists of 32 states, the Common case is 18 states and the lead out is 31 states. The main pattern for all 3 state group is reading the needed YUV data, where we collect 1 VU and 2 Y pairs for 2 UV prime calculations each and 4 RGB calculations. This allows us to write to the RGB address 6 times in one cycle and send the read address for the next state. The lead in calculates 6 rgb pairs and reads 4 YUV pairs each. The lead out 3 Y pairs and sends 12 RGB pairs. It is important to note that 6 RGB reads is equivalent to 2 pixels because the RGB is stored in pairs. This is different to our original state table which had 40 states because of the way we used the multipliers which is discussed later. This ultimately did not work because we did 3 rgb calculations however we had too many YUV reads which would actually need an increasing amount of buffer registers. This would not be efficient in terms of hardware resources or software management so the state table was redesigned.

Critical path:
The slack of our critical path is 2.195, this is quite fast and most likely due to the heavy reliance on combinational logic within our design including the operator values, immediately calculating the next value at the start of the clock cycle.

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	2.195	milestone:milestone1|op2[3]	milestone:...ferodd[29]	clk_50	clk_50	20.000	-0.071	17.732
2	2.196	milestone:milestone1|op2[3]	milestone:...ferodd[31]	clk_50	clk_50	20.000	-0.071	17.731

Register:

The total number of registers is 1249. However this includes the setup files. In general, we had 43 different types of registers. This included the SRAM read,write,read/read enable and address registers. These stored the values that were read and sent to the SRAM with a control on what action to do. A tricky part in implementation was getting used to the registers updating a state later due to the nature of flip flops (which always_ff represent). We had RGB buffers to facilitate the need to pass the post added RGB values to the multipliers and then store the calculated values. In retrospect these buffers could be removed to improve efficiency and reduce hardware by directly sending the calculated values to the RGB addresses. We have buffers for U prime even and odd which are used to store the calculated values to be used in the RGB calculation, there are also 2 Y prime buffers which store the odd and even Y value of the pair because it is read twice as much. The UV prime calculations use 5 registers each to hold the values to be multiplied with the coefficients (which are parameters). There are 3 RGB multiplication registers to hold the values of the multiplication to be added to each other and sent to the SRAM. There is also an extra G register as it takes more steps to calculate and we calculate the odd RGB values at the same time. Additionally there are YUV counters, a pixel counter and a line counter. The YUV counters help with which address to access and are NOT reset after leadout. The pixel counter finds how many pixels have been calculated to enter lead out. This increases after every full RGB write. This is reset after lead out. The line counter counts how many lines, when this reaches 239 it stops the milestone one module.

Arithmetic:

For Y and U prime calculations 7 adders are needed as we calculate 3 values at a time which are then added together. The RGB requires 3 adders for each RGB. The division by 256 in the UV prime is done by right shifting by 8 as a division of 2^3. The larger division in RGB CSC was done by clipping. If the value is negative, it is a 0 and over decimal 255, it is 255. Otherwise the values of [23:16] of the input is taken. This is done with combinational logic. For multiplication, 3 multipliers are used as a restriction. We used all 3 for each calculation respectively to be efficient. In the common case the efficiency is (22 x 2)/(27x2)= 81.48% utilization. Our first approach had around the same but was much slower in RGB calculations.
In always_ff

```
op1<=a00;
op2<=ybuff[1];
op3<=a02;//red
op4<=vbufferodd;
op5<=a21;//blue
op6<=ubufferodd;
```

In combinational logic

```
assign multlong1=$signed(op1)*$signed(op2);
assign mult1=multlong1[31:0];
```

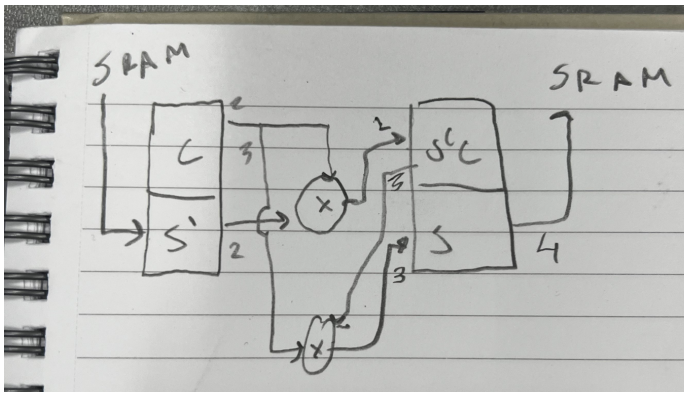
```
assign multlong2=$signed(op3)*$signed(op4);
assign mult2=multlong2[31:0];

assign multlong3=$signed(op5)*$signed(op6);
assign mult3=multlong3[31:0];
```

Debugging and Approaches:

Some debugging which delayed prod of the milestones was a bad state table as discussed before. Additionally, forgetting to shift U and V values to the next register also caused mismatches. Keeping all arithmetic as 32 bits to keep digit values was also needed in debugging. Modelsim and Hexviewer was used to check timing, values, addresses and states to assist in debugging. Most of our errors came in later cycles due to forgetting to move the U and V values to be multiplied in the next prime calculation.

Milestone 2

In terms of milestone 2 we were able to get a strong understanding of it but unfortunately did not have enough time to implement it. We started off with a megastate table built with the 4 main functions of milestone 2. These functions are fetch S' (FS'), compute T (CT), compute S (CS), and write S (WS). The lead in of the megastate table starts with one FS' and then one CT, before the common case is started. In the common case CS and FS' are run simultaneously followed by WS and CT run simultaneously. This can be done because of the way the memory was handled, with the two dual port rams as well as the SRAM always being used within the common case. FS' fetches values from the SRAM and then stores the value into port b of the 1st dual port RAM. CT multiplies the port a and b values from the 1st dual port RAM, and then stores the result in dual port RAM port a. CS takes the T values from dual port RAM 2 port a, multiplies them with the values from port a of the 1st dual port RAM, and stores the result into the 2nd dual port RAM port b. WS takes the values in dual port RAM 2 port b and writes them into the SRAM, completing the S calculation. This is shown further in the diagram below as well in the state table made for milestone 2.



This simultaneous run is possible as the two states that run at the same time use different memory locations at all times, leading to a 100% efficiency in theory for the design. For the megastate table lead out, CS and WS were used for the final S value write. The individual megastates were also somewhat planned out with a finished FS' lead in, with the roll over for

milestone 2 considered, as every 8 x 8 block increments the column block, and every 39 blocks finished increments the row block. This is represented through a row index, column index, block row index, and block column index.

Week	Akash	Sam
1	-Worked on reading over the project specification, and understanding the objective	-Worked on reading over the project specification, and understanding the objective
2	-Worked on state table design	-Worked on state table design
3	-Worked on lead out state table	-Started coding the lead in and common case states
4	-Finished common case -Debugging m1	-Finished lead out code -Debugging m1
5	-Understanding and completing m2 state table Report	-Redoing State table Debugging and finalizing m1 Report

Conclusion

To conclude, this project was full of incredible learning opportunities and pushed the class to the edge, teaching them a lot about digital design and verification. This project bonded us together and left us with a higher appreciation for low level code design. This course will be greatly beneficial to our future career paths as well, with Co-ops employers really taking interest in the course, leading to better opportunities in the short future. This course really brought us to deeply think through digital design and we enjoyed that satisfaction of getting milestone 1 completed. The project brings a perspective to us as higher level programmers that have lots of experience with C, Java, JavaScript and opened our eyes to what Computer Engineering can really offer.

References

https://avenue.cllmcmaster.ca/d2l/le/content/479686/Home?itemIdentifier=D2L.LE.Content.ContentObject.ModuleCO-3925362