3DQ5: Digital Systems Design

Instructor: Professor Mahmoud

Isaac Shoong – Wed 42 – shoongi - 400312914
Andrew Ye – Wed 42 – yeh31 - 400318465

Introduction

In this project we were to design an image decompressor in computer hardware. There were three milestones that needed to be completed which included colour space conversion (CSC), interpolation, inverse discrete cosine transform (IDCT), dequantization, and lossless coding. The objective of this project was to learn how to design hardware systems to perform complex tasks. A main focus of the project was the efficient use of resources, including but not limited to registers, adders, and multipliers.

Designs Structure

The design modules are partitioned into the top-level module which connects everything, every other task is put into its own module. Such as milestones 1 and 2 are put into their own module, SRAM has its own module, the clock has its own module and the dual-port ram has its own module. There are more modules that weren't stated. Many low-level modules are used within each other, being connected from the top module. Such as how the SRAM is used in both the milestone 1 and 2 modules. The modules that have been reused from previous labs are the PB controller module, VGA SRAM interface module, UART SRAM interface module, SRAM controller module, the dual-port RAM module, and the clock module. The custom modules are the top-level project module, the milestone 1 module, and the milestone 2 module. The PB module is used because the buttons and switches are used to reset the program. The VGA SRAM interface module allows for the VGA controller to take data from the SRAM using the SRAM controller module, thus allowing us to display the image on a screen, it is important to enable and disable this module, based on whether it should be in use or not. The UART SRAM module is used to assemble and write data into the SRAM, this module is used at the start of the top level, as a sort of initialization step. The SRAM controller module is probably the most important module in the project; as it is used in essentially every other module. The purpose of the module is to allow us to control what to write and store in the SRAM and what to read from the SRAM, also allowing us to specify which addresses to look at. The dual-port RAM module is only used for milestone 2, and it allows us to store values taken from the SRAM as well as store any calculations that were done during the milestone 2 state and use them for later use. The clock module is also used in every other module, as it tells us when to switch states as when to complete actions. The milestone 1 module completes the tasks of interpolation and CSC. It utilized the SRAM controller module, as it intakes the data stored in the SRAM and writes it back into the SRAM after all calculations are completed. The VGA SRAM interface module uses the data stored from the milestone1 module. The milestone 2 module does the inverse signal transform. It uses the SRAM controller module to both read and writes data, it also uses the dual-port RAM module to read and store calculation data. Lastly, the project module combines every module together as it's top-level and sets the state of the project. There isn't much reasoning for why the lab modules were included, as it was necessary to use. However, the dual-port RAM module was implemented because it's much more efficient and logical to use it to store the calculation data, rather than using upwards of 64 registers. Both module 1 and 2 were created as it was much simpler to separate each milestone into their

own module, allowing for each milestone to be almost independent of the other, which leads to easier testing and debugging.

Implementation Details

Milestone 1

To start with milestone 1, the finite state machine has 3 main states. The lead in state, the common case state, and the lead out state. Each one of these states have smaller states inside them to complete more intricate tasks, rather than the big picture. However, for the big picture of the three main states; the lead in state deals with the interpolation and CSC of the beginning boundary cases, or the left most pixels on the image. As for interpolation, the data leading up to Y5, cannot be included into the common case, as the Y and V prime values repeat themselves, such as having to use Y0 more than once. The common case is just repeating interpolation and CSC with no boundary cases, this repeats until about Y315. The lead out case is essentially the same as the lead in case but flipped. Where it deals with the right most pixels of the image and deals with the test case of no data being available for the right most pixels. In general however, all three states deal with reading data from the SRAM, completing interpolation and CSC calculations, then storing the products back into the SRAM.

In milestone 1 many registers were used. The biggest or most important registers used are the YUV buffer registers, the YUV prime registers, and the RGB registers. The Y registers stored the immediate Y data taken from the SRAM, and is used almost immediately for calculations, thus it is 16-bit wide, as it stores 2 8-bit Y values. The U and V buffer registers are much larger as they must hold many more values. These registers are 56-bit in size, as they store 7 8-but U/V values. The reasoning for the size of the register is because each index stores a very specific value, where register[0] stores the j-5 value, register[1] stores the j-2 value, and so on. Where the last two registers which are 6 and 7; store the next needed j+2 and j+5 values respectively. The biggest part of these two registers, however, is that they're shift registers, as for each new iteration of common case, the oldest two data points which is index 0 and 1 are shifted out and two new values are shifted in.

The YUV prime registers are quite straight forward, as they store the interpolation calculation values. The Y values didn't really need a prime register; thus the Y buffer register is used in its place. However, the U and V values need a prime register. The U and V even prime values were stored almost immediately as they have no calculations. Where as the odd prime values had to be calculated first then stored, where they were stored as 32-bit values, as after interpolation the prime values are converted from 8-bit to 32-bit. The values stores in these registers are then used in CSC to generate the RGB values.

For the RGB registers, they store the CSC calculations until it's time to write back to the SRAM. The reason we needed RGB registers, is because we can't write the RBG value right after the calculation, as the calculations all are completed at different times, and we need to write to the SRAM in a very specific way, which is R0G0, B0R1, G1B1.

The RGB registers in theory should store a 32-bit value from the colour space conversion, however as the SRAM only takes 8-bit data, the values are clipped from 32-bit to 8-bit and then stored into the registers. The values stored in these registers are then sent to the SRAM in the order stated before.

Milestone 1 makes use of two main types of counters. Which are the X position counter, the address counters. The X position counter just keeps track of where on screen the pixel we are calculating is at, and it determines when to end the common case. The lead in case and lead out case have a known length, as it can be coded by hand and very precisely. However the common case we must know when to leave it, which is theoretically at Y313. Thus using the X position counter we can tell exactly when the pixel reaches Y313 and tell the program to switch states from common case to lead out. This counter is 9-bits in size.

The address counters all have one purpose; which is to tell the program which SRAM address it should be reading from or writing to. It includes the YUV addresses all separately, and the writing address for the RGB values. These counters increment by 1 each time something is read from or written to from that specific counter. These counters are all 18-bit in size. We implemented three multipliers that take two 32-bit factors each using constant assignment, rather than do multiplication in the FSM to ensure only three multipliers were being used. The 64-bit product register was then truncated to 32-bits to match the reset of the registers being used for arithmetic. These multipliers were kept above 77.5% utilization.

In the grand scheme of this module, it begins with storing the YUV data taken from the SRAM into the YUV buffer registers. The SRAM read address is equal to the corresponding YUV address and is incremented by one each time something is read. Then taking the values from these registers and using adders and three multipliers for the interpolation calculations and shifting the sum by 8-bits at the end, storing it into the YUV prime registers. For lead in, the interpolation is completed first, then the CSC is completed, by using the three multipliers again and some adders to sum up each value, taking the values from the YUV prime registers. Once the RGB is calculated from CSC, the value is stored into the corresponding RGB register. The values from the RGB registers are then sent and written into the SRAM, incrementing the SRAM write counter each time something is written. This process is almost identical in the common and lead out case. However, instead the prime calculations and CSC for even values are completed first, afterwards the CSC for odd values are completed.

The debugging for milestone 1 was done using the testbenches given in Modelsim. Our first and most important issue was that our calculations were wrong. Thus in order to fix this we did very precise hand calculations to see exactly what we should store in each register, state by state. The main issue we fixed was a sign issue, where certain values were being set as signed where they shouldn't. The second issue was that we were reading from the wrong SRAM values. As we were being told that the value being written were incorrect, however we already proved by hand that the calculations were incorrect. This was fixed by noticing we didn't download the correct simulation files from Avenue.

The last issue was that we noticed the common case was ending too early, as we saw there was a consistent mismatch between the common case and lead out case. Thus a counter was added to see where common case was ending, and it was found that it ended one cycle too late.

Milestone 2

Milestone 2 is the calculation of the inverse signal transform. Milestone 2 has mega states instead of a lead in, common case, and lead out states. Instead it has a state to Fetch the S' values (FS), a state to calculate the T values (CT), a state to calculate S and fetch S' values (CSFS), a state to write S to SRAM and calculate T (WSCT), a state to calculate S on its own (CS), and finally a state to write S to SRAM (WS). The so called lead in case to this module is the states of FS and CT, where the common case would be mega states CSFS and WSCT. The lead out would be CS and WS. Each mega state would be about 64 clock cycles long, as there are 8x8 matrices that you must fetch data for and compute data from, as well as store the data to SRAM.

We used fewer registers in milestone 2 than milestone 1. This was because instead of storing all the SRAM data values and calculations in registers, we used two dual-port RAMs. However, there are still some registers used. Which are a register to store a single SRAM data value, two registers that store the read and write addresses from the SRAM, two registers that store the S' and C calculation, and a register that stores the S calculation. There are 8 registers that store the values of the C matrix. Lastly there are two registers that store the clipped values of S.

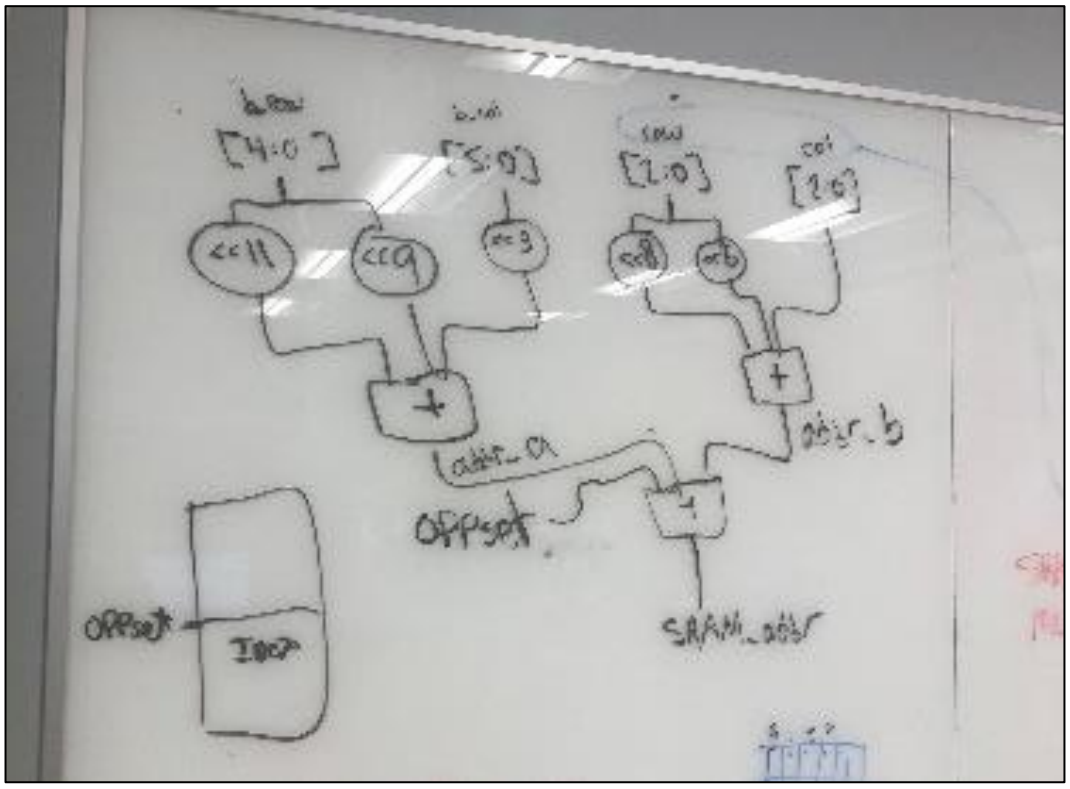

Figure 1

The register to store the SRAM value is used in order to send to the dual-port RAM at the correct time, otherwise that value would get overwritten, as we ream one value at a time for SRAM, however we want to send two values at a time to dual-port RAM. This register is 16-bit in size. The two registers that store the read and write addresses for the SRAM, are derived from counters that track where in the matrix to look. The calculation for these values are calculated from a circuit that was derived with help from a TA. The circuit can be seen above in *figure 1*. These registers are 18-bit in size.

The registers that store calculations are used for future use and accumulation in the register, so we can keep adding values to get to a finalized S value. Without them, we would pass over the amount of multipliers allowed for use and a lot more adders would need to be used, as the calculation would have to be done in one clock cycle. These registers 32-bit in size.

The C value registers are used to store the values of C as stated in the name. These register values are chosen by 8 multiplexers, which are dependent on which column we want to be selecting for multiplication. The clipping registers store the value of S after clipping, as before clipping the size of S is 32-bit, however we must store an 8-bit value to the SRAM, thus this register is used to store that new value before we can send it to the SRAM. These registers are 8-bit in size as discussed before.

There are many counters that are used, which track two main things. The location in the SRAM that we must read from and write to, and the location on the image matrix that we are looking at. For the address counters, the work the same as milestone 1. Where each time the address location is used, whether it be reading or writing. It increments by one as the next address location will now be used. There is one for the S' values, and one for the YUV location. The second type of counter is used to tell X and Y coordinates on the image. As in order to use the circuit in *figure 1* we must know the exact location of the pixel we want to look at. We also must know which matrix block we are in along the image, as the image is cut up into 8x8 blocks. Thus we have two sets of X and Y registers. One for the big picture of the image, in which it's which block we are looking at, and one for within the block, which specific pixel we want to be looking at.

In this milestone, we used eight multiplexers, to select the C matrix values for which we want to multiply by. The selection is dependent on the matrix X counter. We used two multipliers that we kept above 92.5% utilization. The rest of the calculations are completed by adders.

Due to us using a dual-port RAM, the hardware can be described a little easier, as the state definitions are also described above. In FS we read the data from the SRAM and stored one of the values at a time into the SRAM read register, then write two values into the dual-port RAM at a time. Next is CT, where we retrieve the data from the dual-port RAM and calculate the multiplication between S' and C. This state uses the 8 multiplexers to get a value of C and stores the product to the dual-port RAM. In mega state CSFS we calculate the final S value by taking the previous CT values from the dual-port RAM, as well it completes the same actions of FS as well. It then stores the

calculations into the dual-port RAM. The WSCT state writes all the calculated S values from the dual-port RAM into the SRAM and clips the values before being sent, as well as repeating steps from CT. States CS and WS are explained in states CSFS and WSCT.

Most of the debug surrounded making sure a single row of the final S matrix was calculated and written correctly to SRAM. Due to the way we developed our state table, most of the issues were due to incorrect conditions for incrementing counters. Counters played an enormous role in milestone 2—they controlled many addresses, state transitions, and multiplexing the C matrix column—so when one of them was incorrect, it would throw off all the calculations. But we used this to our advantage—by observing how the counters incremented or reset, we could often pinpoint exactly what was causing the problem and adjust the FSM accordingly. Once a single row was successful, all we had to do was increment the counters that controlled row/column transitions and matrix transitions, and the rest would complete itself.

Top Level

The top-level module just pieces everything together and creates a hierarchical state machine. Where it begins with UART, then to milestone 1, milestone 2, and finally VGA. There are multiplexers that control which module has access to the SRAM address, SRAM write data, and write enable.

Table

Week 1	Read documentation (Isaac), M1 state table (Isaac)
Week 2	M1 state table (Isaac and Andrew), M1 Implementation (Isaac and Andrew)
Week 3	M1 Implementation (Isaac and Andrew), M1 debug (Isaac and Andrew)
Week 4	M2 state table (Isaac and Andrew), debugged M1 VGA (Isaac)
Week 5	M2 state table (Andrew), M2 Implementation (Isaac and Andrew), M2 debug (Andrew), Final Report (Isaac)

Conclusion

To summarize, we learned how to effectively create state tables for both milestone 1 and 2. Understanding what type of hardware to use and how to deal with utilization constraints. Learned more on how to deal with clock cycles and delays. Learned more how to write to and read from SRAM as well as dual-port. And lastly, we gained a lot more experience in debugging programs, learning many strategies along the way. In general this project was very hard and tedious—we unfortunately did not make it to milestone 3—however we learned a lot about the development process from planning to implementation and debugging.