Complete Design Report

Michael Li ml1811 , Jake Humphrey jbh111 (Group 36)

1 Functionality

1.1 Draw Block

DB converts both draw line commands and clear screen commands to draw pixel streams for the RCB. DB also has functionality to repackage clear screen commands and forward them to the RCB, but RCB does not currently implement clearscreen, so this is never used. Only clearscreen commands that draw region fill in the positive x and y direction are valid, as per the amended spec.

1.2 Ram Control Block

The Ram Control Block (RCB) takes draw commands from DB and writes the changes into cache. Any clear commands are converted into individual draw commands and passed to RCB. The RCB was fully functional when tested with db_behav, with no known issues. RCB is a composition of two parallel FSMs that concurrently execute allowing for potentially no waiting for ram in RCB FSM by buffering the pixel cache.

1.3 Complete Design

Complete design implements all Host Processor operations: move pen, draw line, and clear screen. Works with all colours:black, white, and invert. There are no known issues in communicating along the shared bus and no known issues remaining after our tests.

2 Testing

2.1 Tom Clarke Star Test

Our combined code mapped to the VDP entity passes the suggested star test. This test was designed to test drawing in each octant and the ability to handle a clear command, as well as inverting an already coloured region. As it draws in a region greater than 3,3 it obviously tests the hardware's ability to draw to pixels not currently in the cache.

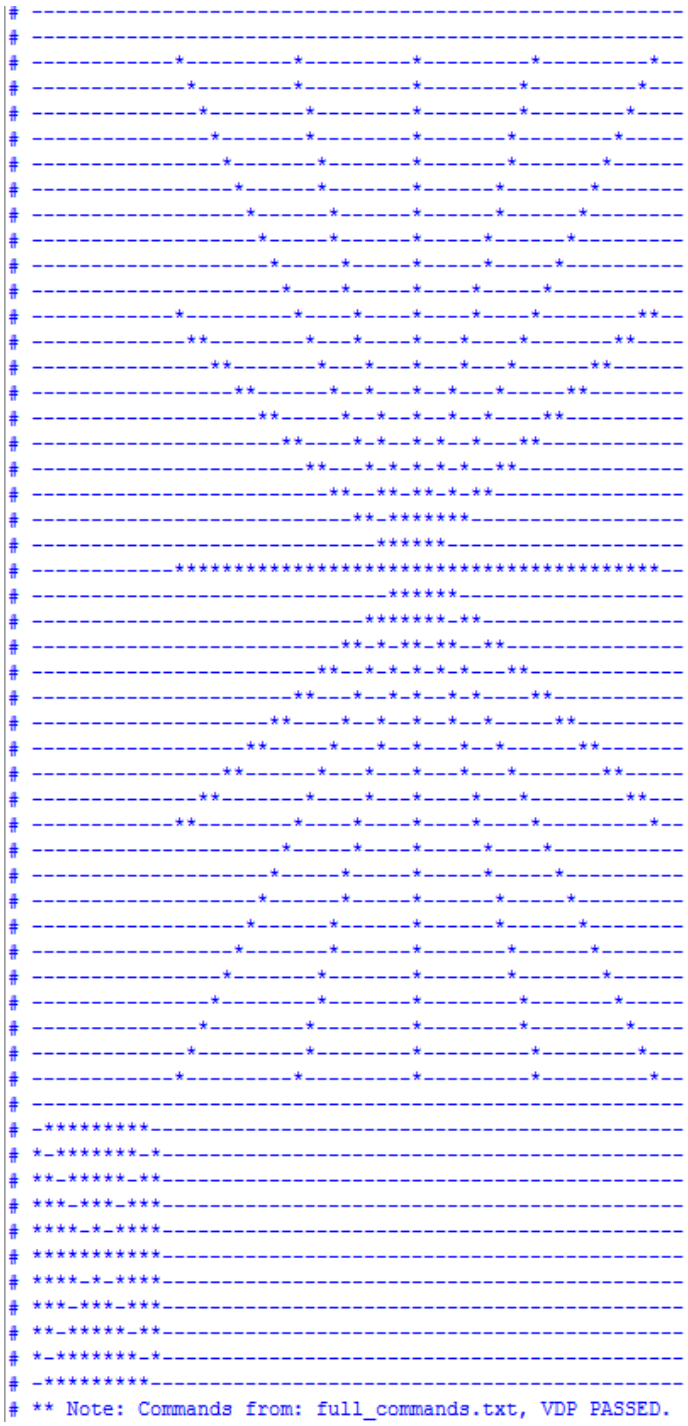

Figure 1: Star Test Result

2.2 Full Clearscreen Test

The next test we chose was one that we knew was guaranteed to write to every pixel by clearing the entire screen. Also in this test we test the case where we clear to the current location, thus a clear of one pixel. This can be seen by the black pixel at (0,0). This test was also passed.

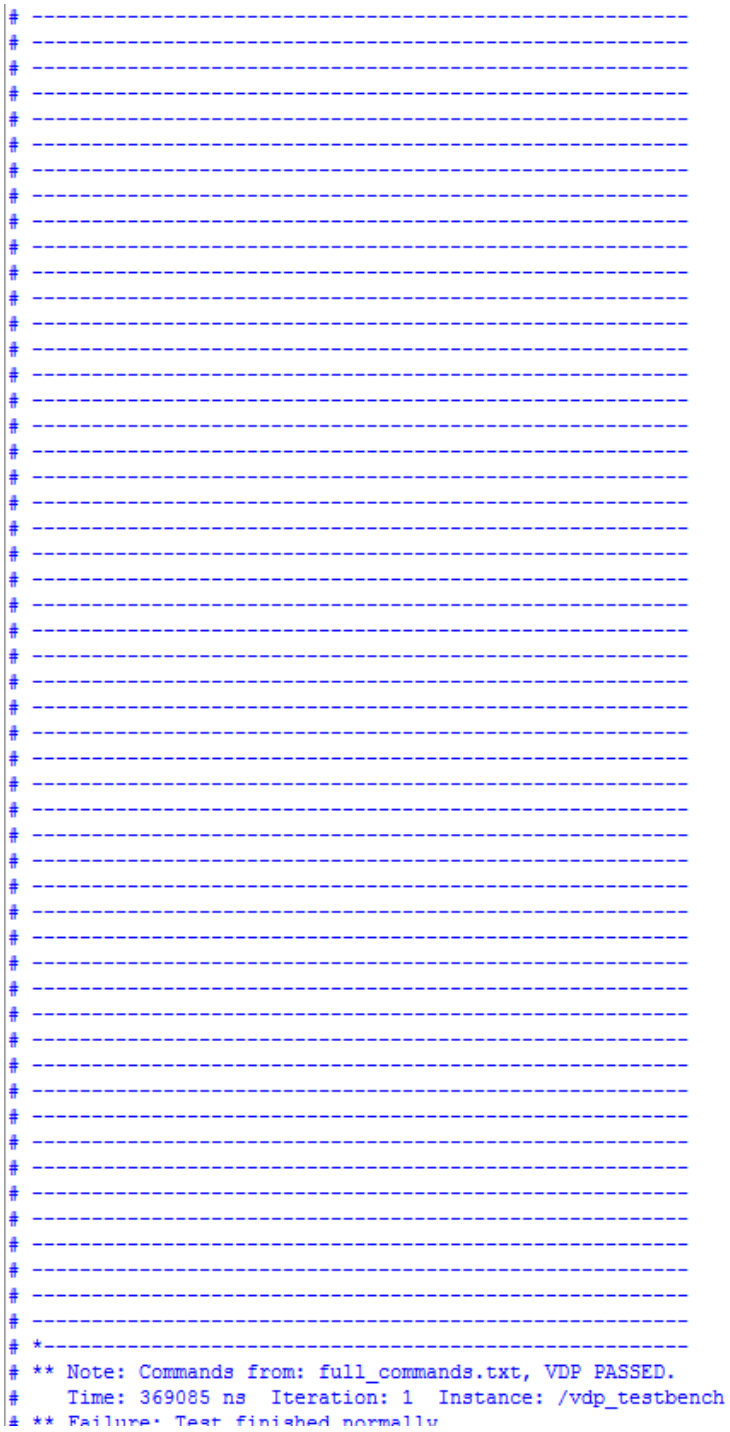

Figure 2: Clear Test Result

2.3 Perimeter Test

This test attempts to draw an upgraded version of the star by starting at the middle pixel and drawing in invert colour to every pixel on the perimeter of the screen. The goal was to see if it can draw all directions of lines as some directions are missing from the star. The pen colour is chosen as invert to test the cache's ability to also invert non block colours as it did in the clearscreen test. Our code passed this test.

Listing 1: Python Code to generate perimeter draw test

```
for i in range(64):
	for j in range(64):
		if i in[0,63] or j in [0,63]:
			print "MW {} {}".format(32,32)
			print "DI {} {}".format(i,j)
```

2.4 Exhaustive Draw Test

Listing 2: Generator for exhaustive draw test

```
for x1 in range(64):
	for y1 in range(64):
		for x2 in range(64):
			for y2 in range(64):
				print "MW {} {}".format(x1, y1)
				print "DI {} {}".format(x2, y2)
```

The above python code generates a command file that will draw to every pixel from every pixel thus truly testing the hardware's ability to draw every kind of line possible. The downside is this weighs in at about 33 Million commands and at a rate of 2000 commands per seconds that was going to take around 4 hours to complete (mainly due to the scroll speed of the transcript which I ended up breaking modelsim trying to increase). Incidentally light would have travelled 3 Billion miles in the time taken. To this effect we wrote a random test to avoid this extremely lengthy testing. This only tests every type of drawing, then to be truly extensive we would need to do the same for all pen colours, for all moves, and all clears with the test size getting to a ridiculous size.

2.5 Random Test

Random sampling from a set of exhaustive inputs is a good compromise and approximation of exhaustive testing. It's more robust than ad-hoc testing as it is hard to guarantee or special cases are accounted for and a lot faster than exhaustive tests. Obviously the limitations of random tests are not complete coverage and it is very unlikely to generate tests that test only one kind of command, such as only moves. Also it will not test all the possible ways of ending the command file unless a large number of random tests are performed. As we were reasonably confident at this point of our code's validity, we wrote a python generator to create random tests for us. The commands are weighted such that there are more draws than moves than clears, as we don't want to keep moving about all the time, nor do we want to continually test the ability to draw in squares. There is more benefit in testing random line lengths and directions thus the distribution shown below was chosen.

Listing 3: Generator for configurable random tests

```
import random

numCommands = 20000
pen = (0, 0)
with open("randomcommands.txt", "w+") as fo:
       for i in range(numCommands):
              rand_command = random.choice(['D','D','D','C','M','M'])
              rand_colour = random.choice(['W','B','I'])
              if (rand_command == 'C'):
                     rand_xCoor = random.randint(pen[0],63)
                     rand_yCoor = random.randint(pen[1],63)
              else:
                     rand_xCoor = random.randint(0,63)
                     rand_yCoor = random.randint(0,63)
              pen = (rand_xCoor, rand_yCoor)

              # print rand_command+rand_colour, rand_xCoor, rand_yCoor
              fo.write(rand_command+rand_colour+" "+str(rand_xCoor)+" "+str(rand_yCoor)+'\n')

print "test file created"
```

This generator enforces the fact that we can only clear in the postive x,y direction by only generating commands that satisfy this rule using the condition if C was selected as the command.The output sample size can be changed by alterating numCommands and the distribution by changing the frequency of of occurrence of literals. We first tested with a sample size of 5000 to check for code coverage using Modelsim.

2.5.1 Code Coverage

We tested the complete design with a random set of 5,000 valid input commands. We turned on Coverage Analysis to observe how much of the design was tested by the random commands. We then examined the uncovered code to decide if either *a*) the missed lines were unnecessary; or *b*) tests could be written to test the missed lines.After refactoring our code, the DB had 94.6% coverage, and the RCB had 68.7% coverage.

DB There was a missed FSM state corresponding to sending clear screen commands to the RCB. Since the RCB does not implement clearscreen commands, this was never run. When the DB is tested with the behavioural implementation of the RCB, this code is run.

RCB Whilst it would appear that the test did not fully exercise all the parts of the RCB hardware, further inspection of the results showed that most of the unexecuted percentage was due to transitions to error state that should not be achieved under valid operation. The error state was creating for debugging purposes and in the event that upstream attempts to send an invalid command such as "100".The untested code is explained below.

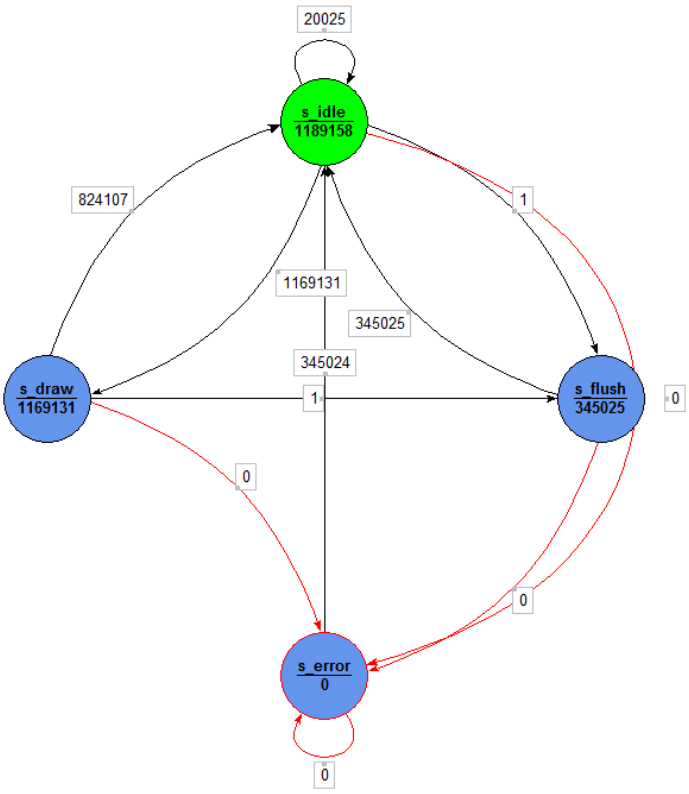

Figure 3: State transitions in coverage for RCB

```
Finite State Machine: state
Instance: sim:/vdp_testbench/DUT/rcb_mike

State Coverage:
   s_idle: 1189158
  s_flush:  345025
   s_draw: 1169131
  s_error:       0

Transition Coverage:
    s_idle -> s_draw: 1169131
   s_idle -> s_flush:       1
    s_idle -> s_idle:   20025
   s_idle -> s_error:       0
   s_flush -> s_idle:  345025
  s_flush -> s_error:       0
    s_draw -> s_idle:  824107
   s_draw -> s_flush:  345024
   s_draw -> s_error:       0
  s_error -> s_error:       0
   s_error -> s_idle:       1

State coverage: 75.00% (3/4)
Transition coverage: 63.64% (7/11)
```

Figure 4: State transitions in coverage for RCB

As one of the 4 states are designed to hopefully not be used and all transitions to error should not be taken, this accounts for a large amount of essentially untested code by this testbench.

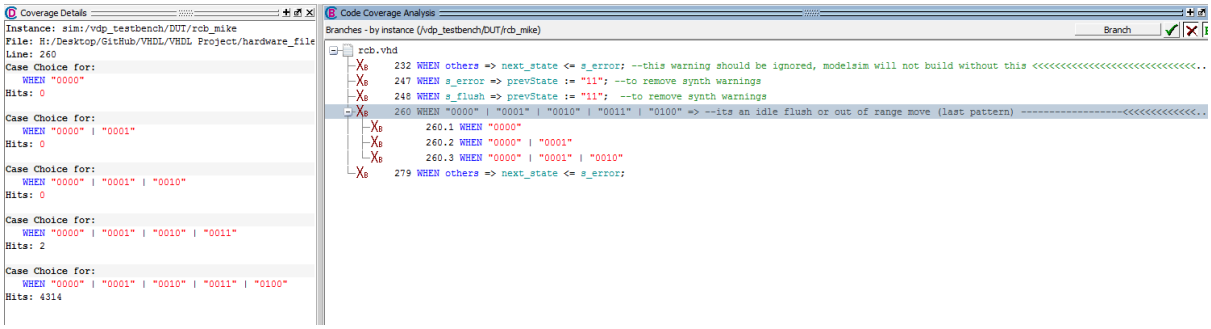

Figure 5: Branch coverage for RCB

The untaken branches are shown to be transitions to error which is good and the expanded line on 260 is the code that executes an idle flush. As the hardware can only potentially be idle at the end due to the testbench providing a constant stream of instructions until the end, only one kind of branch to idle state will occur. (Actually in this case we idle flushed twice at the end due to the interleaving of the ram fsm and rcb fsm). The last full set of cases accounts of Movepen commands which are fairly frequent. The first 3 zero hits can only be achieved with a different kind of testbench that pauses midway.

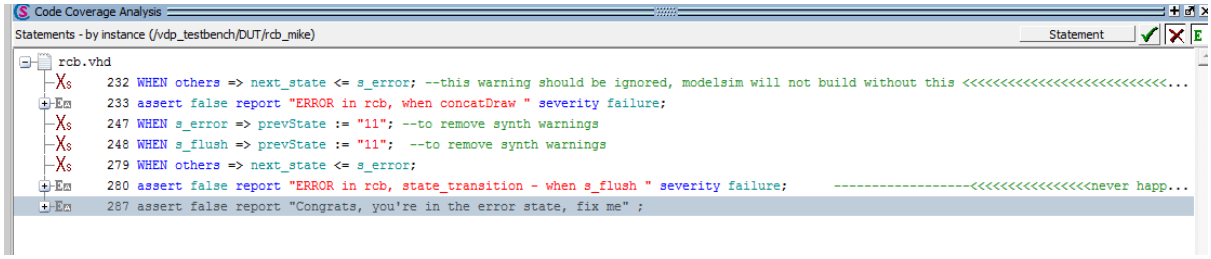

Figure 6: Statement coverage for RCB

Here the unexecuted statements shown are asserts or transitions into the error state that should not happen under good operation.

From the code coverage metrics, it would suggest that in fact 5K random commands was sufficient to cover all of the executable code that this particular testbench could in fact handle.

To see if the coverage would increase and also in an attempt to fish out any remaining edge cases we tested with a few more 5K instances of the test and a 20K and 50K sample size. All of which passed.

Figure 7: Results of 20K random test

2.6 Other Tests

As the random test seemed to cover pretty much all of the code, we assumed no further tests would be needed. However, one random test was able to trip up our design, as the RCB asserted finish too high under some conditions. However, since a testbench can only finish once, we would need one testbench per condition we wanted to test. This, along with our confidence that we had fixed the bug, lead us to decide not to use any further testbenches.

3 Synthesis - DB

3.1 Investigation of Synthesis Options

Synthesis to Altera Cyclone II with Pipelining , Resource Sharing, and FSM Compiler (the defaults) gives Required Frequency 108.9MHz and Estimated Frequency 92.5MHz.

3.1.1 Generic Options

Turning on Fast Synthesis raises frequency to req 113.1MHz and est 96.1MHz.
Using FSM Explorer gives no change in frequency.
Using Retiming lowers frequency to req 106.5MHz est 90.5MHz.
Turning off Pipelining makes no difference.
Turning off FSM Compiler makes no diference.
Turning off Resource Sharing increases the frequency to req 128.4MHZ est 109.2MHz.
The highest frequency we can get is with all options off except Fast Synthesis, which gives req 131.1MHz est 111.4MHz.

3.1.2 Device-Specific Options

With the fastest options above:
Turning off Enhanced Optimisation makes no difference.
Reducing Fanout guide slightly reduces maximum frequency. Increasing makes no difference.
Disable I/O Insertion makes no difference.
Disable Sequential Optimisations makes no difference.
Disabling Read Write Check on RAM makes no difference.
Resolve Mixed Drivers makes no difference.

3.2 Synthesis and Post-Synthesis Testing

Target: MicroSemi 3200DX Speed -2
Putting the design through Synplify spat out a couple of warnings about pruned registers. These were relating to unused signals I had forgotten to delete. After cleaning up the code I got no synthesis warnings.
I then tested the post-synthesis VHDL with the original testbench, using the adapter to interface them. The design passed all the tests, just as it had pre-synthesis.

4 Synthesis - RCB

4.1 Investigation of Synthesis Options

Synthesis to Altera Cyclone II using Synplify Pro with the generic default options: pipelining, resource sharing and fsm compiler yields an estimated frequency of 157.3 MHz. The default specific options for this test are shown below as well as the results of synthesis.

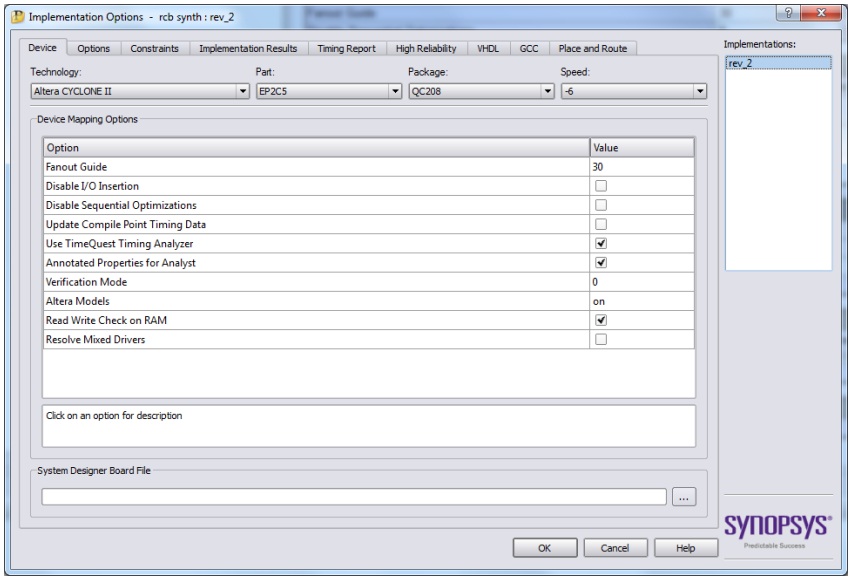

Figure 8: Default options

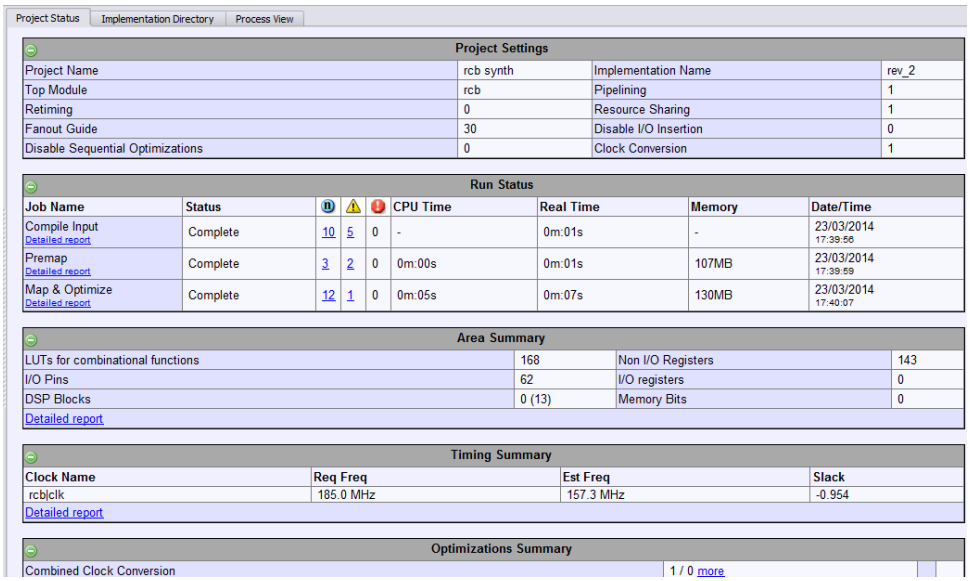

Figure 9: Default Synthesis Results

4.1.1 Device Specific Options

Fanout guide increased appears to make no difference to the clock speed however reducing the fanout slightly increases the clock. A fanout of 10 increases estimated frequency to 160MHz, 8 gives 162MHz and 5 gives 165MHz. Reducing all the way to 1 returns it to around the 157 mark.

I/O insertion disabled suppresses automatic insertions of I/O into the design and raises the clock speed to 159MHz but overall I would say has little effect.

Sequential Optimisations disabled also raises the speed to 159MHz, again a very slight change.

Read Write Check on RAM disabled does not affect the clock speed.

Resolve Mixed Drivers does not alter the clock speed.

Update compile timing data does not alter the clock speed.

4.1.2 Generic Options

Using the default device specific options from above:

Fast Synthesis option is not available on Synplify Pro and apparently only Premiere.

With all the options unchecked the clock speed was raised to 211MHz.

Pipelining on its own makes no difference.

Auto Compile Point makes no difference.

FSM Compiler on its own reduces the speed to 158MHz. The FSM compiler attempts to optimise the logic in state machines instead of treating them as regular logic by re-encoding the state representations. This reduced the LUTs for comb functions to 168 from 184

FSM Explorer used with FSM Compiler returns the speed back to 211MHz and increases the number of LUTs back to 184. The explorer tests different encoding styles but I guess deemed the original one best as it yields results identical to not having the compiler checked at all.

Resource sharing enabled attempts to minimise area, but made no difference to the clock speed.

Pipelining makes no difference to the speed of the design, however pipelining with retiming reduced the maximum clock speed to 186.7MHz and increased the number of LUTs to 224. **Retiming** applies register balancing by moving them across gates or LUTS to improve timings as a whole but does not touch those on a path from a primary input to output.

Checking all the options reduced the clock speed to 143MHz and 204 LUTs were used.

4.2 Synthesis Warnings

```
@W:CD604 : rcb.vhd(232) | OTHERS clause is not synthesized
@W:CL169 : rcb.vhd(90) | Pruning register prev_dbb_bus.rcb_cmd(2 downto 0)
@W:CL169 : rcb.vhd(90) | Pruning register prev_dbb_bus.startcmd
@W:CL271 : rcb.vhd(90) | Pruning bits 1 to 0 of prev_dbb_bus.X(5 downto 0) -- not in use ...
@W:CL271 : rcb.vhd(90) | Pruning bits 1 to 0 of prev_dbb_bus.Y(5 downto 0) -- not in use ...
```

OTHERS clause is not synthesized
This others clause was left in, as otherwise modelsim insists that not all cases are covered, due to multiple or's in the when condition instead of one case for each 3bit number. For the sake of readability I elected to have multiple conditions inside the when case.
Pruning registers in prev_dbb_bus
This warning is fine as I am registering the entire of `prev_dbb_bus` when all I needed was part of the x and y coordinates. Whilst arguably sloppy design, I chose to register the entire bus as it I felt it clearer in vhdl to access the previous x and y as part of the bus instead of as 2 orphaned variables.
Pruning bits 1 to 0 of prev_dbb_bus x and y
This warning is fine as we only need bits 5 downto 2 for computing the related vram address from xy coords and I thought it would be cleaner in the code to register all of it than slice out arbitrary part of the bus. If space was a bigger constraint then the obvious choice would be only register the required bits.

4.3 Post Synthesis Testing

Target: MicroSemi3200DX Speed -2

Initial post synthesis on the RCB part showed some differences between pre and post behaviour. This manifested in the inability to draw the very very first pixel which was likely due to something not being explicitly initialised. In order to discover which was the cause, I attempted to look at the waveform from the post synthesis code, however a lot of the signal names had been randomised. Some still retained somewhat sensible names, the ports for instance and I checked to see if there were any U's or X's propagating through the waveform. This did not appear in pre-synthesis due to how modelsim's default initialisation mechanics behave. Post synthesis vhdl was first tested on the star and various other simple lines to determine if it was indeed the case it was the very very first pixel drawn. When I had resolved this issue I moved onto testing it with the 5K random samples. The post synthesis code for RCB was also tested with 2 random test sizes of 20K commands and a 50K test. All of which it passed.

5 Synthesis - Combined

With both halves working individually we tested them with the generated vhdl on the star test as well as a couple of 5K random tests. This was done by having synplify generate code for both halves individually and then using the adaptors as the entities inside VDP. We chose to only test the smaller samples as the clock speed was set much lower. There was no need to check for code coverage as machine generated vhdl coverage would not have been very representative as our code isn't being tested directly.