Contents

Contents
Organisation 1
Introduction 1
Phase 1: Developing tests 5
Possible graphics algorithms 15
Crib: reading the C++ models 34
Implementation tips & techniques 37
Troubleshooting the tools 43
Phase 2: Developing a module 45
Phase 3: Integration 51
Phase 4: Compilation 61
Appendix A: Floyd-Steinberg dithering hints 68
Appendix B: Trigonometry 70
References 74

Organisation

The laboratory comprises four phases of differing lengths, scheduled as follows:

<table>
<thead>
<tr>
<th>Week</th>
<th>Phase/task</th>
<th>Detail</th>
<th>Deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No lab.</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Test harness</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Test/debug</td>
<td>Demo.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Own unit development</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write/debug</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Test/debug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Test vs. HL model</td>
<td>Code/demo.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Integration</td>
<td>Understand system/integrate</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Add model/test with vscreen</td>
<td>Demo.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Synthesis</td>
<td>FPGA demonstration</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Refinement</td>
<td>Analysis/critique</td>
<td></td>
</tr>
</tbody>
</table>
Normally there will be 1 hour of lectures followed by a 1 hour lab. each week. However week 1 will contain two lectures and week 5 will have 2 hours of lab. to compensate.
Introduction

The object of this practical is to give a flavour of the stages involved – and the sort of tools that are used – in making a working system on chip. Some of the processes should already be familiar, some will probably be new to you.

Implementation will be on FPGA but the tool flow is equally applicable to an ASIC design. However the latter stages of an ASIC design, concerned with minutiae such as electrical power supplies, will only feature in the accompanying lectures.

The project practical work is divided into four phases.

**Test harness**

Understand the details of the interface specification. Given some line-drawing modules, create a test to determine which are functional and demonstrate them running or identify any symptoms of faults. The test should be portable enough to adapt for the next phase.

**Own unit**

Using the models provided (or your own) plus the knowledge/tests from the previous phase, produce a synthesizable Verilog module. Prove this conforms to the same interface as the line module. Finally verify the output data against test data generated from the high-level model. How many clocks/pixel are required?

**Integration**

Integrate the new module into an existing SoC framework. Run full system tests by inputting commands via the processor interface. Demonstrate operation using a virtual screen output through a PLI.

**Compilation**

Add a pad ring designed for the demonstration FPGA and compile the SoC. Download and demonstrate a working system. Analyse the compilation reports for module size and speed. Try to refine the design if time permits.

**Objectives**

The objectives of this laboratory are:

- to understand pre-existing interface specifications and subsequently exploit them
- to develop more Verilog skills, particularly in producing Finite State Machines
- to improve testing processes
- to integrate a novel block into an SoC and watch the system work
Phase 1: Developing tests

Interfaces
The first step in producing tests for a block must be to understand its interface. In this system all drawing machines are accommodated as modules with similar interfaces. The differences between them are their functions and the number and function of their input parameters.

To simplify the testing process there is no reset input. The behavioural models use a Verilog ‘initial’ statement to reset (enough of) the internal state. This also serves when compiled and downloaded into an FPGA. \textit{N.B. This would not be adequate for an ASIC implementation.}

The drawing blocks are fully synchronous. Boolean inputs are active high unless otherwise stated. In addition to the clock there are two (independent) interfaces (\textit{hint: remembering this will make your task easier}).

Command interface
Data parameters must be set up and \textit{req} asserted. After a subsequent clock edge a one-clock-long pulse will be produced on \textit{ack}. \textit{req} should be removed when this occurs or the unit will trigger again when the issued command is complete. \textit{busy} will be asserted at the same time as the \textit{ack} pulse but will be held active for the duration of the command execution. The input parameters are latched at the same time that \textit{ack} is asserted.

**Parameters:** A line is (or \textit{should be}) drawn from (R0, R1) to (R2, R3) in colour R6.
Drawing interface

The output buses are set up and `de_req` is asserted. It is expected that `de_ack` is high for a single clock period when the outputs are accepted and latched. Because the memory interface latches both address and data at the start of a memory cycle write operations can be treated as ‘fire and forget’ and the outputs may change any time after `de_ack` has gone high. (In practice this is likely to be at the next active clock edge.)

In this system a memory cycle takes two clock periods (shaded in figure). Thus output will not be accepted on every clock cycle. This means that, in use, `de_ack` will always pulse rather than staying high. You could choose to exploit this knowledge in your design.

The 18-bit address addresses 256 Kwords of 32 bits (i.e. 1 Mbyte). Writing to individual bytes is controlled by the active low byte strobes. Any combination of byte strobes may be used in any cycle. The data bus is a 32-bit bus; write data will be taken from the appropriate byte ‘lanes’ within the word. Bytes which are not enabled are ignored.

Note: in the line drawing examples supplied only one byte (pixel) is output on any cycle. (In principle it is sometimes possible to output more, if the line is near-horizontal.)

The display uses one byte/pixel with address 00000 in the top-left corner of the screen. Within a word the pixels are mapped in ‘little-endian’ order, so the lowest byte address is on the left. Addresses are mapped contiguously, such that the second scan line begins at word address 000A016 ≡ 16010.
Practical

A number of variants of a line drawing unit have been prepared which, supposedly, comply with the specification described. All these have the same pins and, in case it’s wanted, a symbol with the same pin positions. Some of them have functional errors however. You are assigned a personal library with subset of these units to test and diagnose what, if anything, is wrong with them.

Setup using “mk_cadence 322”; you need to enter your ‘card’ number as drawn in the first week’s lectures. This gives you a COMP32211 library complete with test unit_* when you “start_cadence 322”. Everyone sees the same names but the contents are different. These are compiled Verilog modules so you can’t see the source code.

To facilitate testing of several similar modules a schematic called drawing_line_wrapper is provided. This is simply a wrapper around the unit under test. The unit tested can be changed by changing the cell_name property of the contained symbol to the unit you want. Don’t forget to regenerate the netlist each time you change this so the simulator picks up the intended unit.

Prepare a single test harness into which each assigned unit can be inserted and the tests run. Use this to deduce what, if anything, is wrong with your units.

Remember that you will shortly be building and testing your own unit which must comply to the same interfaces although its function will be different. A well designed test should be able to accommodate this, making subsequent development much easier.

NOTE: when drawing graphics there may be more than one correct answer. If a point should be exactly halfway between two available positions it is an arbitrary choice which is plotted. It is (usually) also irrelevant in which order the pixels are plotted. To ensure this does not cause problems there is a (C++) ALM of a line drawing algorithm (in /opt/info/courses/COMP32211) which produces the same points in the same order as the correct Verilog units.

To extract/run the model you could use:

gtar -zxvf /opt/info/courses/COMP32211/ALM.tgz

cd ALM

make

e xport PATH=$PATH:/opt/cadtools5/vscreen/bin

./controller < draw.txt

and modify the input file (draw.txt) to taste. This tool will dump a list of non-black pixels from the frame store into a file for comparison. If you want to see these in the order they are drawn you will have to edit drawingEngine.cpp and rebuild the model.

Deliverables

A demonstration of your test at work.

A listing of your test file (and schematic if appropriate).

A brief description of the faults found (if any) in each unit.
Marking scheme

- 10 marks for test file/strategy
  - half for thoroughness, half for clarity
- 2 marks for each assigned unit correctly diagnosed
- 20% of coursework

Test Strategy

Different designers prefer different methods for setting up a test environment. Cadence, by default, uses the following flow.

The stimulus file is used to drive the simulation. This may contain all the test logic, data etc.

An alternative is to use the stimulus to provide minimal input – such as a reset and a start indication – and include the test details in a special test module (or modules) in a test design.

In a collaborative project it is sensible for everyone to adopt a ‘house style’ so that it’s easy to read others’ work. In this lab, you have the freedom to choose what suits you best.

Things to test

This list is intended to suggest most of the major things to consider. However no effort has been made to make it exhaustive.

Input protocol

- Does the unit wait until an input request arrives?
- Does the unit acknowledge the request appropriately?
- Is there a single acknowledge for every accepted command?
- Does the unit indicate that it is busy for the duration of operation?
Output protocol

- Does the unit produce output requests?
- Does the output data change in response to acknowledgements?
- Does the output data change only in response to acknowledgements?
- Does the output rate adapt as the acknowledgement frequency varies?

Unit function

- Is the expected number of data outputs produced?
- Are the data (addresses, strobes) the expected ones?
  - Does the ordering of these matter?
- What input conditions will produce ‘extreme’ outputs?
  - e.g. horizontal, vertical, or single pixel line

First get the sequencing right. If the unit is communicating using the correct protocol its function can be tested. Test parameters, chosen to exercise different internal functions, can be sent and the outputs observed. The results need to be verified against a test model. There are different ways to do this: for example the outputs could be saved in a file and compared off-line. Alternatively a test set could be loaded into the simulator and compared during simulation. The test data may be generated from a high-level model in an appropriate form.

Sometimes tests can also be performed by scrutinising the output. With a drawing system it may be very obvious when the figure drawn is wrong and this can give some quick, early feedback. It is rather more difficult to verify that something is absolutely right however; can you spot that a point is one pixel position out by looking at the screen? Furthermore, in ‘real’ designs the sheer size of a test suite means that some automation is necessary.

Regression Testing

Take a ‘perfect’ design which has been tested and is working and add a small enhancement; does it still work?

Unless some features were deliberately removed it should still pass all the original tests – plus some new ones. Tests should therefore be developed such that they are easy to repeat and, in the case of failure, easy to identify the problem. This should be taken into account in their design.

This applies to the development process too. When a change is made, for example during debugging, it is useful to boost confidence by rerunning the test suite to ensure some previously working feature has not been disturbed.
Some useful Verilog features

This section is intended to introduce (or revise) some aspects of Verilog which may prove useful in writing testbenches. It is not intended to be comprehensive! However it should illustrate features which may help to produce neat-yet-powerful stimulus files.

Like any programming language there is more than one way to achieve the desired results. Experiment and find methods which suit your own style.

Iteration and control

Behavioural Verilog has a number of control structures similar to (for example) C.

```
while(!ready) <statement>
for (x = 0; x < 10; x = x + 1) <statement>
repeat (10) <statement> // Repeat statement ten times
forever <statement> // As you might expect
```

Note that, with ‘forever’ there must be a delay (by one mechanism or another) inside the statement or time will never advance. For example:

```
initial // Clock generator
begin
clk = 0;
forever #5 clk = !clk; // Toggle every half period
end
```

```
if (<condition>) <statement #1> else <statement #2>
```

```
case (<expression>)
  0: <statement>
  1,2: <statement>
  default: <statement>
endcase
```

can also be used as behavioural control structures.

Randomisation

$random may be useful in setting up tests where a variety of different values or timings may be exercised. It returns a repeatable pseudorandom sequence of integers.

Memory

```
reg [15:0] xyzzy [0:255];
```

Declares a 256 (16-bit) word memory. It is useful (but not compulsory) to declare the number of elements in the array in this fashion so that word #0 appears first rather than last. Elements of the memory can be used much as may be expected.

```
xyzzy[36] <= 16’h1234; // Assign a value to location #36
data <= xyzzy[address]; // Read the value at ‘address’
```
As will be discussed later (and, probably, in a lecture) there are some constraints to using memories (sensibly) in a compiled design. However for simulation purposes they can be used without these.

A memory’s contents can be initialised in various ways:

```plaintext
integer i;
initial
   for (i = 0; i < 256; i = i + 1) xyzzy[i] = 0;
```

will set all the contents to zero in a behavioural model. (Note that variables such as ‘integer’s may also be used here!) This is not synthesizable though.

```plaintext
$readmemh("init_file", xyzzy);
```

Will read in a hexadecimal file. The format reads at successive addresses; new addresses can be specified by preceding a number with ‘@’.

There is also a ‘$readmemb’ function for reading binary format input.

If it is not initialised then a memory will start with all its locations undefined.

---

[In a Xilinx FPGA compilation $readmemh can also be used to initialise a memory; this is useful if a ROM is required as it can be declared ‘normally’ in the source file. This is not necessarily the case for ASICs where a separate generator utility would probably be used. A RAM on an ASIC would be expected to power up in an undefined state.]

---

**Output**

The function: `$display("Test value %x", value);` acts very much like ‘printf()’ in C (for example). This allows the output of values as the simulation is run.

The may appear in various places, depending on the simulator and the way it is configured. In the lab. the output appears, along with other information, in a file:

```plaintext
~/Cadence/COMP32211/<test_directory_run1>/simout.tmp
```

Variables can be printed in decimal, hexadecimal, etc.

Hint: remember that a text file can be filtered later for points of interest, for example using ‘grep’ to find particular annotations. It may be useful to keep this in mind when defining the strings to print. For example, “ERROR” can highlight problems …

`$strobe()` is very similar to `$display()`; the subtle difference is that any variable values may be printed a bit later, still at the same time but after other, pending simultaneous changes have taken effect.

`$fopen()`, `$fclose()`, `$fdisplay()`, `$fstrobe()`, `$fwrite()` allow you to create your own output files. (“fwrite” is the same as “fdisplay” except for line-feeds.)
There are also some system tasks which may be useful in debugging. In particular $time can help to identify when a problem has been detected.

Sometimes it is convenient to watch particular signals rather than all signals through time. This can be done with the $monitor task.

```verbatim
initial
    $monitor("this = %x that = %x at time %t", this, that, $time);
```
this is similar to:

```verbatim
always @ (this, that)
    $display("this = %x that = %x at time %t", this, that, $time);
```
$monitor is automatically sensitive to any variable changes (except $time and its relatives) which may or may not be useful. It is also possible to switch monitoring on and off with the {$monitoron, $monitoroff} tasks. This means that long traces can be decluttered of everything but the area of interest.

**Events**

Events are things that happen during execution; they can be used to control the progression of a test sequence. An event can be something like:

```verbatim
@ (posedge clk)
```
Which will wait until the specified event occurs. These can be used in combination with other operators. For example:

```verbatim
repeat ($random & 3) @ (posedge clk);
```
will wait for between zero and three (inclusive) rising edges of the clock.

**WARNING** "$random % 4" may not have the same effect because $random returns a signed integer and thus the modulus may also be negative.

**BEWARE** of signed numbers. Most Verilog types default to being unsigned – but not all!

Events can also be declared and generated in a behavioural model. For example, here is a mechanism for generating reports when errors are detected:

```verbatim
event error; // Declare event
always @ (error) $display("Error at time %t", $time);
initial
begin
    ... // Simulation proceeds
    if (<error condition>) -> error; // Generate event
    ...
end
```
Wait

Another way of detecting signal states is to use: `wait (request == 1);`
which should need no extra explanation.

This differs from: `@ (posedge request)` which waits for a switching event and so will pause if `request` is already high.

Tasks & Functions

Tasks are subroutines. They are probably most useful for test purposes, such as in the example below which models a microprocessor bus write.

```verilog
task bus_write(input [5:0] addr, input [15:0] data);
begin
  uP_ncs     = 1’b0;
  uP_address = addr;
  uP_wdata   = data;
  #10 // Set up time
  uP_nwr     = 1’b0; // Activate strobe
  #15 // Pulse width
  uP_nwr     = 1’b1; // Deactivate strobe
  #10 // Hold time
  uP_ncs     = 1’b1; // Deselect
end
endtask
```

Note that there is alternative syntax for (e.g.) the way I/O is specified.

In Verilog ‘functions’ are similar but more primitive; they return a single value and cannot contain delays. Thus they are most appropriate for abstracting particular arithmetic or logic functions.

Both may contain local variables.
Possible graphics algorithms

All the graphics accelerator modules described here must use the same hardware interfaces as in the previous exercise. The only interface difference will be their interpretation of the input parameters which can be set to appropriate values for the figures being drawn.

The units described here are not an exhaustive set of possibilities: you are welcome to define your own! However an attempt has been made to ‘balance’ the difficulty of the units here and they are described in some level(s) of detail and supported by software algorithmic models which you can download, execute, modify etc. The algorithmic model should illustrate how pixels are derived and may give values to test against. It does not necessarily give implementation details, particularly in possible parallelism and mapping operations into cycles.

The models are archived in: /opt/info/courses/COMP32211/C_models/

Each unit requires some interfacing with the SoC – the details of the interfaces should now be clear – and some internal development as a Finite State Machine (FSM).

Unless you’re feeling very energetic, develop only one design. Each section contains some suggestions as to how it could be improved/optimised. Only attempt this if time permits.

Whichever design you are developing it is strongly recommended that you draw the state diagram first, ensuring that you know the circumstances under which each state transition takes place and what the outputs should be in each state. This can then form a basis for your test strategy.

Each design description concludes with some suggested extensions to the basic principle. Some of these are easy, many will be quite time-consuming. These are present for your interest and to act as suggestions if you want to develop things a bit (or a lot) further; they are not part of the scheduled exercise.

Circle

Here we are thinking of an outline circle. This is reasonably similar to plotting a straight line, so you should first refer to Bresenham’s line algorithm and be convinced you understand that, in principle.

We will ignore any clipping against the screen boundaries – at least for the moment – and assume that the circle is centred on the origin. This makes the gradient of the circle’s perimeter easier to calculate.

Unlike a straight line, which has (by definition!) a constant gradient, the gradient of a circle’s perimeter changes continuously. Rather than worry about that it is usual to pick a point and assume a constant gradient for each step (pixel) plotted. This makes a slight difference to the result – especially for very small circles where the gradient changes quickly – but is not really noticeable in most cases. There is a choice as to whether you calculate the gradient before or after you take each step; again it doesn’t make too much difference so you can choose either (but be consistent). The model provided calculates the gradient first, which is the second of the later examples and is more amenable to a parallel implementation.
A circle is a symmetric figure, thus we only have to calculate the gradient over one octant. By reflecting in the x and y axes and by swapping the x and y coordinates all the octants become available. Thus, plotting from (say) coordinates (0, r) to (r/\sqrt{2}, r/\sqrt{2}) (i.e. the 45° point) is adequate. We can do this all with integers: no square roots etc. need be calculated.

What is the gradient of the perimeter of the circle?

The equation which defines the circle: \(x^2 + y^2 = r^2\)

So, differentiating: \(2x + 2y \frac{dy}{dx} = 0\)

Rearranging: \(\frac{dy}{dx} = -\frac{x}{y}\)

As the values are reflected in various ways we can swap signs at will, so let’s start with a positive y and decrement it instead. The algorithm is shown on the right.

We still have division here (yuk!) so multiply through by 2y:

If AA and BB (above) are swapped (or in parallel as is likely in a hardware implementation) the differential approximation is slightly different so the circle is a slightly different shape! Neither is ‘perfect’; the differences are small, especially at larger radii.
Here are a worked examples of the two alternatives:

<table>
<thead>
<tr>
<th>x</th>
<th>error (step 1)</th>
<th>y</th>
<th>error (step 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>20 - 2*1 = 18</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>18 - 2*2 = 14</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>14 - 2*3 = 8</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>8 - 2*4 = 0</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0 - 2*5 = -10</td>
<td>19</td>
<td>-10 + 2*19 = 28</td>
</tr>
<tr>
<td>6</td>
<td>28 - 2*6 = 16</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>16 - 2*7 = 2</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2 - 2*8 = -14</td>
<td>18</td>
<td>-14 + 2*18 = 22</td>
</tr>
<tr>
<td>9</td>
<td>22 - 2*9 = 4</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>4 - 2*10 = -16</td>
<td>17</td>
<td>-16 + 2*17 = 18</td>
</tr>
<tr>
<td>11</td>
<td>18 - 2*11 = -4</td>
<td>16</td>
<td>-4 + 2*16 = 28</td>
</tr>
<tr>
<td>12</td>
<td>28 - 2*12 = 4</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>4 - 2*13 = -22</td>
<td>15</td>
<td>-22 + 2*15 = 8</td>
</tr>
<tr>
<td>14</td>
<td>8 - 2*14 = -10</td>
<td>14</td>
<td>-10 + 2*14 = 18</td>
</tr>
<tr>
<td>15</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

These are plotted below:

![Graph of worked example](image1)

**Extensions**

- Changing the step sizes using a predetermined ratio can draw ellipses … although there are more complications …

- Drawing a filled circle is possible by using the calculated start and end points of a row and filling the pixels between. Ideally this should only write to each row once (rather than overwriting repeatedly near the top and bottom) and should improve performance by writing to each word once, rather than up to four times for the four pixels.


**Rectangle/Blitter**

By ‘rectangle’ here we refer to a solid rectangular area with boundaries aligned along the screen’s axes; simply drawing the boundaries of the area would be rather simple as both horizontal and vertical lines are easier than the general line algorithm.

The simplest way to do this is to use two nested loops, filling in pixels in (say) a horizontal line and then repeating this until the desired area is completed.

However, filling in an area one pixel at a time may be quite slow, especially as the display is accessible in four-pixel words. Filling several pixels at once may give a significant (up to four times) speed-up and, in this case, is worth attempting.

Note that the left and right edges of the rectangle may or may not align exactly with the word boundaries, thus there may be some write operations which only access a subset of the available pixels.

Also note there are some ‘interesting’ cases of very thin rectangles to cope with.

Simply *drawing* a rectangle is a bit boring – and arguably a bit easier than some of the other exercises – so to make it more interesting let’s apply a function where the rectangle is able to take account of the existing frame store. Rather than simply *writing* to the frame store, the pixels can be a function of both the write and the existing contents. This requires that the FSM first reads the frame store, modifies the read word and writes it back to the same address. This is, of course, slower because the number of memory operations is doubled but gives more flexible (and more interesting) drawing functions. Guidance on reading the frame store RAM is given in a section on page 39; note that it requires a little attention to get the timing correct.

There are some common graphics drawing operations, notably simply overwriting and XOR-ing. The latter is convenient because it is non-destructive: XORing the same thing again
reverses the action (it may be used for drawing cursors etc. removing the need to copy out the pixels below the cursor). As it is not known in advance which functions the user requires, a common technique is to supply two masks which allow bits to be individually set, cleared, toggled or left alone. These are applied by first ANDing with one mask and then XORing with the other, as shown below.

<table>
<thead>
<tr>
<th>Input</th>
<th>AND mask</th>
<th>XOR mask</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>!X</td>
</tr>
</tbody>
</table>

For example, to write to all bits to ‘0’ (fill with black) use AND: 00; XOR: 00. To invert just the red colour component (leaving others unaltered) use AND: FF; XOR: E0.

For an outline of the colour map, see the section on page 41.

**Blitter**

This should be regarded as a possible extension to the rectangle machine.

A ‘blitter’ is typically thought of as a block image transfer machine. This is a memory-memory DMA engine, typically capable of some simple processing ‘on the fly’. The basic operation is copying a 2D block of memory (i.e. a rectangle in our terms) from one place to another. It may also apply some function to the destination instead of simply writing the output, much as described above.

A blitter can therefore be seen as an extension to the rectangle function previously described, where the source ‘colour’ is fetched from the frame store rather than being a fixed (per operation) colour value.

*Note that this adds a considerable complexity to the problem, especially if the source and destination rectangles are aligned differently with respect to word boundaries. You need to be enthusiastic to attempt this.*

An intermediate alternative (still quite tricky) is to use a source texture (i.e. a previously stored repeating pattern) instead of the source colour.

**Extensions**

- Develop some blitter functions
- Change colour shade by interpolating across rectangle
- Shade using a texture map

---

1. This is not the original etymology, but it’s a handy reminder
Characters

Drawing a character consists of copying a set of pixels from a template to the display. As a simple example here we first consider characters which are word-aligned with the display, i.e. they always occupy the same set of pixels in each display word. This restricts the horizontal positioning of the characters, in this case to every fourth column of pixels; we could opt to remove this restriction but it makes the algorithm a bit more complex and is not a requirement, here.

What is a requirement is that the particular character can be specified. This is usually in a bit map, where a ‘1’ will specify a pixel to be filled in the foreground colour whereas a ‘0’ will indicate either a background colour or the pixel to be left in the pre-existing background (your choice, or you could give the option of either). A 8×12 character map is available for you as a hexadecimal bitmap as /opt/info/courses/COMP32211/support/characters.hex although you are welcome to discover or define your own.

The characters will be downloaded into a memory on the FPGA. This can be a ROM or could be rewriteable. You have probably seen something similar using the LCD panels in an earlier module.

Reminder: to define a memory in Verilog you need a statement something like this:

```verilog
reg [7:0] my_memory [0:1535];
```

(The size in the above example is scaled to accommodate 128 characters of 12 bytes each.)

The basic drawing algorithm involves nested loops scanning in the predefined area. (You may specify this position on a per character basis or define a cursor which moves as you print within your design.) The simplest algorithm will plot one pixel at a time but it should be reasonably straightforward to be able to speed up the operation by writing whole (four pixel) words in a single operation, especially if the characters remain word-aligned.

Extensions

- Add a ‘cursor’ to ease the software burden in printing strings
- Printing onto non-word aligned positions
- User definable characters (character definitions in RAM)
- Print in different orientations e.g. reflections, 90° rotations
- Proportionally spaced characters
- Scaling up (by integer sizes); print blocks of 4, 9… adjacent pixels
  (The scale could be passed as a parameter.)
- Anti-aliasing
Mandelbrot

Rendering (part of) the Mandelbrot set is a rather specific challenge but requires another FSM which can produce attractive results.

Firstly, what is the Mandelbrot set? This was once a very popular computing challenge in that it is easy to define but very compute intensive. Firstly take an area in the Argand plane (i.e. a representation of a complex number) at some scale: each pixel will have coordinates with a real (x) and an imaginary (y) component. Call this number c such that:  

\[ c = x + i \cdot y \]

Now, initialise an accumulator (Z) to zero and iteratively apply the function:  

\[ Z \leftarrow Z^2 + c \]

and see what happens. If Z remains finite the point c is a member of the Mandelbrot set; if Z tends to infinity it is not.

The characteristic attractive patterns are generated by looking at the number of iterations it takes for Z to become ‘big’ (e.g. \( |Z| > 2 \)) and colouring the pixel appropriately. Pixels from points inside the set will never reach this magnitude; thus there must be some limit (e.g. 256) placed on the maximum number of iterations and values which reach this limit are traditionally coloured black.

Complex multiplication

Some reminders about complex numbers:

- \( i = \sqrt{-1} \)

- A general complex number has the form:  
  \[ c = a + i \cdot b \]

- Normal algebra applies:
  
  - Addition:  
    \[ (p + i \cdot q) + (r + i \cdot s) = (p + r) + i \cdot (q + s) \]
  
  - Multiplication:  
    \[ (p + i \cdot q) \times (r + i \cdot s) = p \cdot r + i \cdot (p \cdot s + q \cdot r) + i^2 \cdot q \cdot s = (p \cdot r - q \cdot s) + i \cdot (p \cdot s + q \cdot r) \]
  
  - Modulus:  
    \[ |(p + i \cdot q)| = \sqrt{p^2 + q^2} \]

Here we are dealing with fairly small numbers. Traditionally this is done using floating point variables; however floating point numbers are really intended to extend the dynamic range of the values being used. Here we can use fixed-point numbers to avoid such complications.

Fixed point arithmetic

In programming languages it is normal to use floating point numbers to represent the complex coefficients; this is largely because the only alternative choice is typically integer, which is clearly inadequate. However fixed point arithmetic is simpler and perfectly adequate for this task.

Fixed point arithmetic can be illustrated with a couple of simple, decimal examples:

- addition works in exactly the same way as integer addition, provided the operands are correctly aligned.

\[
\begin{align*}
1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \\
+ & 5 & 6 & 7 & 8 & + & 5 & 6 & 7 & 8 \\
\hline
6 & 9 & 1 & 2 & 6 & 9 & 1 & 2
\end{align*}
\]
- multiplication is similar to integer multiplication except care must be taken to ensure the ‘point’ is in the correct place. Note that, in general, the result of a multiplication needs more bits than the operands (i.e. as many bits as both the operands).

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\times & 5 & 6 & 7 & 8 \\
\hline
0 & 7 & 0 & 0 & 6 & 6 & 5 & 2
\end{array}
\quad \begin{array}{cccc}
1 & 2 & 3 & 4 \\
\times & 5 & 6 & 7 & 8 \\
\hline
0 & 7 & 0 & 0 & 6 & 6 & 5 & 2
\end{array}
\quad (2\text{d.p.})
\]

**FSM**

The state machine needs to traverse the display identifying the complex value represented (programmably) by the coordinates at each pixel position: this can be done by incrementing values in loops. Each value is then fed to another state machine which iterates the operation \( z = z^2 + c \) until either \( |z| > 2 \) or the iteration limit is reached. The returned number of iterations can then be used to determine the pixel colour, either algorithmically or via some pre-chosen look-up table.

Note, there is no need to calculate a square root to determine the modulus – think about it!

**Multiplication in Verilog**

Multiplying in the Verilog HDL is easy: it has a multiplication operator (’*’) which can be used directly. Synthesizing the multiplication into hardware can be much more of a problem as there are so many alternative implementations (e.g. with different degrees of parallelism and different hardware budgets) that many synthesizers need at least ‘hints’ and, sometimes, explicit RTL design.

The Xilinx FPGA contains multiplier blocks which the Xilinx synthesizer will exploit when possible; in this device (XC3S200) there are twelve 18×18 multiplier blocks. These allow quite rapid combinatorial multiplication so, in a fairly slow circuit, a ‘reasonable’ size multiplication can be performed simply by using the standard operator. In cases such as this design using a straightforward ‘*’ should be okay as long as the limit on the number of multipliers is not exceeded.

(It is possible to direct the synthesizer to pipeline multiplications into several clock cycles to increase performance but that should not be necessary in this case.)

For multiplying by constants it is still sometimes sensible to write the RTL explicitly. Multiplication by powers of two should (obviously) be expressed as left shifts. For a multiplication by (say) 640 – something which may be desirable in calculating screen position – shift-and-add can still be very handy:

\[
y\text{-address} = (y\text{-coord} \ll 9) + (y\text{-coord} \ll 7);
\]
Now many bits? A difficult question to answer as it depends on how precise you want to be and how far you want to zoom in vs. the use of FPGA resources and the multiplication speed.

An experiment using 40-bit numbers has shown quite acceptable results with a simple ‘*’ operation requiring three multiplier FPGA blocks and iterating at >50 MHz\(^1\). The chosen format was as shown below.

\[
\begin{array}{c|c}
2 & \text{38-bit fraction} \\
\end{array}
\]

With 2 bits left of the binary point, numbers up to (but not including) 4 can be represented; as ‘infinity’ for Mandelbrot is typically >2 this is okay. Note that having only one bit here would limit numbers to just below 2.

Going to a slightly longer representation is probably better and unlikely to impact resources too much. (Remember that the product is twice the length if you keep all the bits, though.)

**Iteration**

The resultant machine will need several clock cycles to determine the colour of a given pixel and so may not be super-quick. If you want a further challenge you could consider the opportunities for parallelism as all pixels are effectively independent and several can be evaluated simultaneously. You are limited largely by the number of multipliers available and, of course, your development time.

The tables on page 24 illustrate the behaviour of the value Z (Real and Imaginary parts) as it iterates for various values of c. Some values are within the Mandelbrot set, although the behaviour of Z oscillates in some cases and stabilises in others. Other values are outside the set and the values can be seen tending to infinite magnitude after a fairly small number of iterations.

**Parameterisation**

Although the Mandelbrot set is a fixed function it is possible to explore it in detail only by zooming into chosen regions. It is suggested that you facilitate this by making the starting point coordinates and the pixel-to-pixel step size (in the Argand space) programmable. You could also control the maximum number of iterations (a small number allows faster exploration (but less pleasing results) if a large amount of the picture is in the Mandelbrot set. It is also possible to alter things like the colour map.

Because it can take ‘many’ iteration cycles per pixel and it can cover many pixels you might like to confine the generator to a subset of the screen, at least for initial simulation.

**Extensions**

- Experiment with: the colour palette, the iteration limit, the number of bits in the fixed-point representation
- For the mathematically inclined, try some other functions: e.g. Julia Sets

---

1. The whole machine needs to run at least 25 MHz, set by the pixel output rate to the screen.
Mandelbrot iteration examples:

<table>
<thead>
<tr>
<th>iteration</th>
<th>$c = -1$</th>
<th>$c = +1$</th>
<th>$c = i$</th>
<th>$c = 0.1 + 0.1i$</th>
<th>$c = 0.2 + 0.2i$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Re(Z)$</td>
<td>$\Im(Z)$</td>
<td>$\Re(Z)$</td>
<td>$\Im(Z)$</td>
<td>$\Re(Z)$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-1.0</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>26</td>
<td>0</td>
<td>-1.0</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>0</td>
<td>677</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>458330</td>
<td>0</td>
<td>-1.0</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>-1.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>9</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-1.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>11</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>-1.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Mandelbrot

<table>
<thead>
<tr>
<th>iteration</th>
<th>$c = 0.3 + 0.3i$</th>
<th>$c = 0.4 + 0.4i$</th>
<th>$c = 0.5 + 0.5i$</th>
<th>$c = 0.6 + 0.6i$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Re(Z)$</td>
<td>$\Im(Z)$</td>
<td>$\Re(Z)$</td>
<td>$\Im(Z)$</td>
</tr>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0.300000</td>
<td>0.300000</td>
<td>0.400000</td>
<td>0.400000</td>
</tr>
<tr>
<td>2</td>
<td>0.300000</td>
<td>0.480000</td>
<td>0.400000</td>
<td>0.720000</td>
</tr>
<tr>
<td>3</td>
<td>0.159600</td>
<td>0.588000</td>
<td>0.041600</td>
<td>0.976000</td>
</tr>
<tr>
<td>4</td>
<td>-0.020272</td>
<td>0.487690</td>
<td>-0.550846</td>
<td>0.481203</td>
</tr>
<tr>
<td>5</td>
<td>0.062570</td>
<td>0.280227</td>
<td>0.471874</td>
<td>-0.130137</td>
</tr>
<tr>
<td>6</td>
<td>0.225388</td>
<td>0.335068</td>
<td>0.605730</td>
<td>0.277183</td>
</tr>
<tr>
<td>7</td>
<td>0.238529</td>
<td>0.451040</td>
<td>0.690078</td>
<td>0.735796</td>
</tr>
<tr>
<td>8</td>
<td>0.153459</td>
<td>0.515173</td>
<td>0.334812</td>
<td>1.415514</td>
</tr>
<tr>
<td>9</td>
<td>0.058147</td>
<td>0.458116</td>
<td>-1.491580</td>
<td>1.347862</td>
</tr>
<tr>
<td>10</td>
<td>0.093511</td>
<td>0.353276</td>
<td>0.808079</td>
<td>-3.620885</td>
</tr>
<tr>
<td>11</td>
<td>0.183940</td>
<td>0.366070</td>
<td>-12.057820</td>
<td>-5.451920</td>
</tr>
<tr>
<td>12</td>
<td>0.199827</td>
<td>0.434670</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mandelbrot
**Cellular automaton**

A cellular automaton is a computing system based on the evolution of a regular grid of cells. In each generation (i.e. cycle) the next state of a cell is determined by the state of a set of its neighbours. In principle the cell array can be infinite but as each cell’s ‘neighbourhood’ is finite the state can be computed rapidly if sufficient parallelism is available.

As an illustrative example, the most famous example is probably **Conway’s Life**. This uses a square grid of cells where each cell has a binary state: ‘alive’ or ‘dead’. The cell’s subsequent state is determined by its eight neighbours according to the following rules:

- If exactly two neighbours are ‘alive’ the current state is retained
- If exactly three neighbours are ‘alive’ the next state will be ‘alive’
- Else the next state will be ‘dead’

The example below shows three successive generations of cells obeying these rules.

Life is a 2D automaton and is possibly a rather heavy challenge for this lab. The suggestion is therefore to attempt a **1D automaton** (say, along a single row of pixels) and evolve over ‘time’ in successive screen rows.

![Cellular Automaton Diagram](image-url)

Such automata have been widely studied; some rules produce more interesting patterns than others. It should be possible to build a FSM which allows different rules to be programmed (and, possibly, different initial states to be set up). As an illustrative example, here is the so called ‘Rule 30’ set, which has binary states and sets the next state of a cell according to its own current state and the states of its two immediate neighbours.

The figure on page 26 shows the evolution of a single ‘1’ cell applying Rule 30. Successive generations are shown on successive lines and some cells are annotated with a binary code showing their neighbourhood in the previous line. The figure is drawn with the LSB on the left; this is simply a reflection of the view in, for example, Wikipedia and works with addressing as on the display. The actual answer does not matter!

This example may be a little different from some of the other drawing machines described in this chapter in that it requires considerable state-holding to maintain the cellular array. Assuming the figure is drawn vertically at full-screen resolution there are 640 pixels corresponding to 640 cells/state bits.
There are two ‘obvious’ solutions to this problem:

- keep the cell states on the FPGA
- read the cell states back from the frame store.

**Keeping the cell states on the FPGA** is feasible in this instance. Storing each cell as a D-type flip-flop renders the bits fast to access but is quite costly in resource terms. A more resource efficient solution is to use an on-chip memory, subject to operational restrictions; there needs to be RAM available (which shouldn’t be a problem here) and the FPGA ‘block’ RAMs are synchronous which means ensuring that a clock cycle occurs between presenting the address and reading the data (see page 37). Because of the ‘shape’ of the FPGA RAM blocks it makes sense to store the bit values packed into words; note that sometimes bits from two adjacent words will be required for one state evaluation. This can be done by:

- (sometimes) inserting an extra cycle
- *interleaving* two RAMs so that simultaneous reads – possibly from different addresses – can be concatenated. Some examples are highlighted in the adjacent figure
- retaining some previous state bits and reading *one* new state bit/cycle

With more, or more complex states the FPGA resources may be insufficient. For example, consider the amount of storage needed for a 2D automaton such as ‘Life’. In such a case the practical answer may be to **read back states from the frame store**. This gives much more storage but access is much slower; each memory operation takes two cycles *when it is granted*. It is therefore a poor solution to read all the ‘neighbourhood’ for each output and something ‘cleverer’ is needed.
Clearly each cell appears in several ‘neighbourhoods’ but only needs reading once. For a 1D automaton three cell states need to be established to output the first new state, but after that (assuming cells are processed in succession!) only one new cell is needed for each output.

A further optimisation (in our system at least) is possible by noting that:

- up to four pixels may be fetched at once if they’re in the same word
- the state of a cell outside the array (i.e. off-screen) can be arbitrarily set to (e.g.) ‘0’.

**Reading the frame store** requires some attention to timing: see the section on page 39. If you’re going to read back from the frame store you’ll need to know its initial state. The simulator clears the frame store to zero (black) ‘artificially’; this can be done by invoking the ‘clear_screen’ block provided on the final hardware. However you will also need to ‘seed’ some pattern for the machine to begin on or the result will be very boring!

It is possible to do **direct access from the processor to the frame store**. Three registers are provided: two to supply a pixel address and the third to write/read a data value. Modifying either of the address registers (two are needed to provide the full address – and this is an address, not a pair of coordinates) instigates a read operation which causes the data register to contain the addressed pixel value. Subsequently writing to the data register will instigate a write and change the pixel in the frame store. These registers are summarised on page 57.

Note that direct accesses are not interlocked with the processor cycles (which, in the hardware implementation cannot have ‘bus wait states’) thus there is a delay between changing an address and the frame store read (or write) being completed. This delay is not deterministic because it depends firstly on synchronising the processor’s request with the drawing machine clock and then on what else is competing for access to the frame store. It is bounded in that the worst case is about 7 clock cycles. When running software under Komodo this should never be a problem; when running in simulation you are advised to insert delays between successive accesses to these registers.

This is the sort of design which may benefit from use of the Memory Viewer (see page 38) during development.

**Extensions**

- Scaling: drawing the cells as clusters of (say) 4 pixels to reduce the effective resolution of the display.
- Experiment with larger neighbourhoods
- Non-binary cell states
- Two dimensional automata (e.g. Life) [This is quite challenging!]
Dithering

The display in the lab. has quite a limited range of colours (i.e. 256). This is a trade-off with both the amount of memory and the memory bandwidth available. For example, there is enough RAM to dedicate 16 bits/pixel but to read these to the display would require twice the bandwidth and thus reduce the time available for drawing to just the video blanking times. (Drawing would be much slower.)

Dithering is a mechanism for rendering colours/shades on a limited resolution; it is commonly used in (e.g.) printing to give greyscale images purely from black pixels on white paper. It relies on the average intensity/colour over an area being the desired one even if any particular pixel cannot be.

Dithering can be done in a variety of ways. The method described here is Floyd-Steinberg dithering which is a straightforward algorithm which can process an image with a single raster scan.

The algorithm takes the desired colour of a pixel and finds the nearest approximation which can be rendered; that is the colour used. The difference of that colour and the desired colour is then distributed to the pixel’s neighbours; in this case the neighbours which have not been rendered yet. This difference is then accumulated and affects subsequent ‘desired’ colours.

In the example here we are trying to render the colour 55 into a (monochrome) frame store which can only hold 4 bits. The values/differences are all shown in hex with the final values in bold type. The figure superimposes the frame store values with the accumulating differences and shows the state after rendering the first pixel, the first line and on completion.

Note that the differences need more bits for their representation than the frame store (the bits which are being removed plus 4 bits to accommodate the ‘/16’ fraction but only one line plus one (the ‘next’) pixel need be stored. The difference values are also modified (read/altered/rewritten) at a high rate – four values per pixel rendered. This suggests that it is inappropriate to keep these data in the frame store and a local cache on the FPGA would make sense. With up to 640 pixels and eight bit values this is quite feasible in block RAM.

Note that the FPGA block RAMs are dual-ported so can be read and written to in a single cycle. However they are synchronous so each read-modify-write would need to be pipelined.
As we are not starting with an image with a higher colour resolution, here we will look at rendering a known colour in a lower colour resolution. As a starting point a single, ‘flat’ area of the chosen colour can be painted.

The Floyd-Steinberg dithering implementation need not be more complicated than the other algorithms outlined in this chapter. However a ‘clean’ implementation, especially bearing in mind targeting an FPGA is probably more different from a software model than most of the others. To this end, some additional hints are provided staring on page 68. Please have a serious think about a solution to the problem first – before you choose to read this section – but do consult it if the problem seems overwhelmingly complicated.

Extensions

- Rather than shading a flat area of colour, produce a colour gradient across the rectangle. The desired colour can be determined by using a linear interpolation; the same process as Bresenham’s line algorithm but using colour intensity instead of the y coordinate.

- Try another dithering algorithm.

Edge detection

There are various methods of finding edges (i.e. changes of intensity/colour) in images which can be found in various references. The one described here is a Sobel filter, which is one of the simpler methods. There are plenty of detailed descriptions available so only an outline of the theory is included here.

The filter works by convolution of a filter function and the pixel array. The pixel array is rectilinear so there are two convolution matrices, filtering for vertical and horizontal edges, respectively; other than a 90° rotation these are the same. The Sobel filter only considers the target pixels nearest (eight) neighbours, so the matrix is 3×3. It is easily illustrated by the example below, which is a monochrome image so values represent only pixel brightness.

<table>
<thead>
<tr>
<th></th>
<th>Filter</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 7 7 6 6 5 5 5</td>
<td>-1 0 +1</td>
<td>0 -1 -4 -7 -4 0</td>
</tr>
<tr>
<td>7 7 7 7 6 5 5 5</td>
<td>-2 0 +2</td>
<td>0 0 -4 -7 -4 -1</td>
</tr>
<tr>
<td>7 7 7 7 6 6 5 5</td>
<td>-1 0 +1</td>
<td>0 0 -4 -5 -4 -3</td>
</tr>
<tr>
<td>7 7 7 7 6 6 6 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This shows a vertical edge filter being applied to a region which includes a change in intensity. Two pixels are highlighted as examples in the input and output. Values have been elided where the filter overlaps the defined edge, hence the output region has fewer defined values.
Convolution of the left-hand highlighted pixel will perform the following operation:

\[-1 \times 7 + (0 \times 7) + (+1 \times 7) + (-2 \times 7) + (0 \times 7) + (+2 \times 7) + (-1 \times 7) + (0 \times 7) + (+1 \times 7)\]

\[= (-7) + (0) + (+7) + (-14) + (0) + (+14) + (-7) + (0) + (+7) = 0\]

Convolution of the right-hand highlighted pixel will perform the following operation:

\[-1 \times 7 + (0 \times 6) + (+1 \times 5) + (-2 \times 7) + (0 \times 6) + (+2 \times 5) + (-1 \times 7) + (0 \times 6) + (+1 \times 6)\]

\[= (-7) + (0) + (+5) + (-14) + (0) + (+10) + (-7) + (0) + (+6) = -7\]

The filter picks out changes in intensity from left to right. In these cases:

- In a ‘flat’ area the result is zero
- In an area of changing intensity the sign indicates the direction of the change (in this example negative as the intensity is reducing, left to right) and the magnitude indicates by how much the intensity has changed over a 3-pixel span.

Often only the magnitude is important and the absolute value can be taken. When dealing with images such as photographs – where some variation due to ‘noise’ might be expected even in a ‘flat’ area – it would be usual to apply a threshold to remove all the ‘insignificant’ small values.

The example above only detects vertical edges (i.e. intensity changes left to right); a quick though-experiment should be enough to verify this. A similar, rotated filter will detect horizontal edges. (In this example the edge is close to vertical so shows less strongly here.)

An edge in a ‘random’ direction can be surmised be combining the two results. The ‘correct’ way to combine the filter outputs is as \( g = \sqrt{g_x^2 + g_y^2} \). When ‘thresholding’ the output the square root is (of course) a lot of unnecessary work as it would be simpler to square the threshold for the comparison.

Pragmatically, even the squaring can often be neglected and it is enough to simply sum the absolute values of the two components. The result is not quite as ‘good’ but a lot of computation is saved.
Application to coloured images is probably easiest by simply separating the colour components (e.g. \{\textbf{Red}, \textbf{Green}, \textbf{Blue}\}), treating these independently and summing the resulting absolute differences before applying any thresholding. This will reveal borders between different colours even with similar luminosity.

**Implementation & Extensions**

A naïve (but perfectly valid) implementation might perform eight pixel reads for each pixel filtered. (The coefficients for the central pixel in both filter matrices is always zero, so that read can be omitted – and 8 is a nicer number than 9. Including an output-write operation that is nine frame store cycles per pixel.

There are a couple of optimisations which can make the process significantly faster:

- On this machine, a word containing four adjacent pixels can be read in a single memory cycle. Note that this word may or may not contain all the desired pixels from that particular row.
- Assuming some ‘sensible’ progress across the image there will be considerable overlap of the pixels needed for successive operations; caching the pixels as they are read can significantly reduce the frame store bandwidth.
- Parallelising the filters can allow several operations to take place in parallel.

Of course, each of these also adds some significant complexity to the design!
Assuming four parallel filters, prefetching and caching parts of the frame store on chip and neglecting edge effects and start-up costs it is possible to process and output four pixels with a single read and a single write, i.e. 0.5 frame store cycles per pixel or an 18\times speed-up!

For other extensions, other (possibly larger?) filter kernels could be tried; the kernel could be made programmable, as could the threshold(s). At the cost of some extra work the thresholds settings could even be automated.

**Pragmatic considerations**

You will need some images to process. One way is first to draw some test images and then process those. It may be more interesting to process photographic images however.

The lab. equipment was not designed with this objective in mind, so loading the display is a bit tedious. In particular there is no direct route from the host PC to the frame store. The suggested route is to load the image into the processor’s RAM\(^1\) and use the ARM to copy it to the frame store. This can be done by first updating the pixel address registers and then the data register in the VDU controller’s interface (see page 57).

There is not enough RAM to contain a complete frame. It is therefore suggested that a \(\frac{1}{4}\) size image is used (i.e. 320\times240 pixels); this also allows input and output to be displayed together, side by side. The colour resolution is also a limitation but pictures should be recognisable.

To convert images into an appropriate format the ‘vscreen’ utility, used during the simulation process, can be employed. Start this with:

```
/opt/cadtools5/vscreen/bin/vscreen -w320 -h240 -c332 -s2
```

Height and width should be clear here; the ‘c’ parameter specifies how many bits to use for RGB respectively; the ‘s’ is a scaling factor onto the PC display which will make the image easier to scrutinise (feel free to change this).

vscreen will allow you to load your own (e.g. JPEG) images, converted into the appropriate resolution. The ‘To source’ button will then output a file suitable for loading to the frame store. The output format is suitable for pasting into an assembly language source file *unless the name ends in ‘v’*, in which case it is formatted for a Verilog `$readmemh()` . You may like to experiment with image manipulation first to enhance contrast or colour saturation; there are many utilities to do this and you no doubt have your own favourite already.

**Trigonometrical functions**

It is possible to derive functions such as sine and cosine with a smallish number of iterations of a fairly simple machine. The CORDIC algorithm has not been fully developed for this lab, but is described in an appendix (page 70) if anyone is interested. This could be used, for example, to plot appropriate curves on the display.

---

1. This is a rather slow process on the serial link – sorry.
2. Okay, it’s a quick bodge to get it going.
**DIY**

You are welcome to develop your own idea instead of one listed here. It is recommended that you talk this through first with a staff member to make sure the idea can be achieved within the time of the course with a sensible amount of work.

---

**What if I want more than eight, 16-bit parameters?**

There are various solutions to this. Possibly the simplest is to sacrifice one of the existing parameters as a ‘command’ register and use this to determine different ‘actions’ for the interface. As well as ‘draw’, another action (or actions!) could be ‘copy these parameters’. This allows *lots* of values to be transferred.

This would be one way (for example) of writing to a character generator RAM or loading a colour translation table.
Crib: reading the C++ models

This is a brief section intended to act as a reminder or ‘crib sheet’ of the operation of C++ as it is used in modelling algorithms in the COMP32211 examples. It is not a tutorial in C++; there are plenty of those out there already (e.g. http://www.cprogramming.com/tutorial/).

The C++ models provided for this module are there to act as a starting point for your own modelling. They can be found in /opt/info/courses/COMP32211/C_models/COMP32211_CPP.tar. This contains a number of C++ source files (which have the filename suffix “.cpp”). Some of these will be referred to here.

There is a ‘Makefile’ included in this archive; this is simple and should be easy to alter – if you feel the need. Each graphics function has its own source file.

Objects

The main model is in “main.cpp”. This commences with declarations, several of which are declaring objects. For an example, let’s follow “VirtualScreen” which provides a display model within a workstation window. The source here is in “virtualScreen.cpp”. This defines a number of ‘methods’. Two of immediate interest - as examples of their type - are:

- The constructor defined by “VirtualScreen::VirtualScreen()”.
- The destructor defined by “VirtualScreen::~VirtualScreen()”.

The constructor executes first, once, when the object is instantiated. Conversely, the destructor executes first, last, when the object is destroyed. In this example the constructor is used to fork a parallel process which executes an independent binary “vscreen” with its own list of parameters. The destructor is there to signal to this daughter process that your model is closing down, so that vscreen can also shut down tidily.

In between construction and destruction the user can invoke methods such as “shm_addr” - which simply returns a value - as required. Return to “main.cpp” to see this done on the newly created “vs” instance to pass this value to the FrameStore instance “fs”.

Constructors/destructors are not compulsory; for example FrameStore does not have them.

I/O

If you’ve been looking at the source files, something you’ve probably noticed by now are statements like:

cerr << “frameStore: WARNING - address “
   << hex << address
   << “ out of bounds in read.”
   << endl;

This is rather similar - if terser and, arguably, clearer - to a C statement:

printf(“frameStore: WARNING - address %x out of bounds in read.
”, address);

University of Manchester School of Computer Science
in that it outputs a message. The particular differences in this case are:

- Syntax.
- Formatting: in this case the keyword `hex` overrides the default (decimal) interpretation of the following integer.
- The output in this case is directed to `stderr`.

**Standard streams**

In brief, any Unix process has three ‘standard streams’

- `stdin` defaults to terminal (keyboard) input
- `stdout` defaults to terminal output
- `stderr` also defaults to terminal output

These streams are like serial files and other files can be opened, of course. Also the streams can be *redirected* to other places, such as files:

```
hello_world > output_file
```

or, more ‘sophistically’, other devices

```
hello_world > /dev/tty20
```

or even discarded

```
hello_world > /dev/null
```

On a command line the ‘>’ symbol redirects `stdout` and the ‘<’ symbol `stdin`.

`cerr` sends the output elements to `stderr`.

- `cin` collects a ‘line’ from `stdin`.
- `cout` is ‘normal’ process output to `stdout`.
- `cerr` is usually used for reporting anomalies to `stderr`, as here.

Note that, by default `stdout` and `stderr` are mixed together; however they can easily be separated, for example routed to different ‘log’ files.

`endl` is simply a line feed.

**main.cpp**

This is an interpreter which reads commands (and parameters) from stdin using the `readCmd` method from the `ci` instance of the `CommandInt` class then sending them to a chosen object (together with a reference (effectively a pointer) to the instance `fs`). It is then the specific instance which needs to interpret these parameters.

It is unlikely that you need to add any cases here, but it should be trivial to do if you need to.
Drawing function

The example drawing modules are separated into files and classes for neatness. It is recommended that this organisation is kept.

Within the appropriate class the coding should be straightforward. The only extra things you need – or, indeed, will have hardware access to – are provided by \texttt{fs} methods:

- \texttt{fs.write} writes a pixel to an address in the virtual frame store.
- \texttt{fs.read} returns a pixel (byte) from an address in the virtual frame store.
Implementation tips & techniques

The device used in the labs is a Xilinx Spartan-3 ‘XC3S200’. Full data can be obtained at:

By modern standards this is a small device. It has

- 480 CLBs (Configurable Logic Blocks)
- 12 18Kbit block RAMs
- 12 18×18 bit multipliers

Which contains designs up to around 200 000 logic gates (manufacturer’s figures).

Using (block) memories on the Xilinx FPGA

A memory will typically be declared in Verilog something like:

```
reg [7:0] my_memory [0:1023];
```

This example instantiates an 8-bit wide, 1024 entry memory.

Large memories are inefficient on the FPGAs unless they exploit the existing ‘block’ RAMs. These are synchronous memories thus the data output of the memory is latched. If the memory is expressed differently the synthesizer will attempt a much less efficient implementation which can rapidly use up the FPGA’s resources.

The synthesiser (‘xst’) appears to identify block RAMs by their ‘template’ in the source file. It appears they need to be separate blocks with external multiplexing for values such as addresses if these come from different sources. Below are some ‘templates’ which appear to work: there are other examples to be found in the ‘XST user guide’.

**ROM**

```
initial $readmemh("hex_file", my_memory);
always @(posedge clk) if (en) rd_data <= my_memory[addr];
```

**Single-port RAM**

```
always @(posedge clk)
begin
  if (we) my_memory[addr] <= wr_data;
  rd_data <= my_memory[addr];
end
```

**Dual-port RAM**

```
always @(posedge clk)
begin
  if (we) my_memory[wr_addr] <= wr_data;
  rd_data <= my_memory[rd_addr];
end
```

[Note: as FPGA technology progresses, block RAMs are becoming usable in more modes.]
Read Only Memory (ROM)

Making a ROM – e.g. for a character look-up – is easy provided the above rules are obeyed. The ROM is simply implemented in a RAM with the ability to write disabled. It is necessary to supply the contents of the ROM to the compiler/synthesizer; this is most easily done using a separate file which is read by a Verilog system task:

```
$readmemh("filename", my_memory);
```

It’s probably safest to give this the full path/filename.

This task reads successive data values and places them in the memory, by default starting at the first address. Verilog allows addresses to be set ‘on the fly’ in the file and permits commenting. However the Xilinx synthesizer is rather fussy: it will:

- object to comments
- ignore the file unless its length exactly matches the memory size

(It is, of course, possible to pre-initialise RAMs this way too although it is less useful in the general case.)

Memory viewer

When debugging a machine like this it is often useful to be able to view the contents of memories. This is difficult on a wave trace (they occupy too much space and are hard to read) so there is a tool which makes this rather easier. Select a memory and Windows ⇒ New ⇒ Memory Viewer (there is also an icon on the toolbar for this) should pop up a clearer view of the memory’s contents.

Memory interleaving

Interleaving is an old technique used in various ways to enhance memory performance. First, let’s take an example from the dithering exercise where we may want to both read and write locations in a memory simultaneously. This is not needed in the FPGA block RAM which is dual-ported. Here we pretend the memory has a single port.

One obvious method is to interleave in time: read-write-read-write… This may be slower than we might like though. Another way – if it can be guaranteed that accesses are systematic – is to break the memory into two (or more) smaller blocks, say one for even and one for odd addresses. Now the operations can be paired in parallel without conflicting: {read-even, write-odd} - {read-odd, write-even} etc.

Another approach, possibly useful for the cellular automaton, tackles a problem where we want a string of consecutive bits from the memory which are not necessarily in the same word. If they are all in one word there’s no problem, even if the memory is divided ‘vertically’; however if the string starts near the top of one address and extends into the bottom of the next, interleaved blocks may provide this in a single cycle (see figure on page 26). This may make the RAM more complex but it can speed up action and probably simplify the control FSM.
**Reading the frame store**

Some of the algorithms may require you to read the frame store as well as writing to it. This requires some attention to timing because the interface has really been optimised to make writing easier. Note that, in the same fashion as a write, the request will be latched and an acknowledge generated when the arbiter decides to allow access but that is not the end of a read process because you have to wait for the data.

The acknowledgement being high at a rising clock edge indicates that valid data will be returned on the next rising clock edge: refer carefully to the accompanying figure. The address (etc.) does not need to be held locally as it is latched by the arbiter. However, if you want to use the lower address bits as a byte select on the incoming data word (for example) then the address will need to be retained, either directly or by latching (pipelining) a local copy. Pay attention to the signal timings!

**Data forwarding**

Some applications may require the frame store to be read, then written with a value derived from the memory. This could cause some consternation with timing if the write is to (try to) immediately follow the read. This is because the outgoing (‘write’) data will be latched at the same time as the outgoing address but incoming (‘read’) data may be arriving just before that edge.

Scenario 1: write delayed after read.

Here the data read from the memory must be latched (“temp”), pending the grant of a write cycle. The original data – direct from memory – was withdrawn sometime before it is recaptured for the write operation.
Scenario 2: write immediately follows read.

Here a local data register (“temp”) will change too late for its output to be valid at the start of the write cycle. “data_out” needs to be ready to be captured externally before the clock edge.

Possible solutions:

- Always insert at least one cycle of delay between a read and a consequent write operation.
  - Disadvantage: may cause an unnecessary performance penalty

- Use a transparent latch to hold the data; thus data will be available before the clock edge.
  - Disadvantages: not strictly ‘synchronous’ behaviour – latch may be ‘opened’ by control glitches; may be expensive in FPGAs (which are optimised to use D-type latches).
  - In ASIC design this is likely to be the most efficient speed/area trade-off but may be difficult to generate reliably through many toolchains.

- **Forward** (sometimes called “bypass”) the data depending on the control state. Data is latched with a D-type register in case it has to be held but offered at the output directly in parallel.
  - Disadvantage: the extra edge-triggered registers are (relatively) area-expensive.
  - Arguably the ‘preferred’ solution in most cases; ‘safe’ as HDL specification.
Colour map

The lab. equipment (and accompanying simulation) uses eight bits per pixel for its colours. These are divided as shown on the left of the figure. Human eyes are less sensitive to blue than to red and green so, if a compromise has to be made, it is normal to use a lower resolution here. For direct comparison purposes, if the output values for red and green are regarded as 0-7, the values for blue are as shown on the right of the figure.

<table>
<thead>
<tr>
<th>Colour</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>White</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>FF</td>
</tr>
<tr>
<td>Red</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>E0</td>
</tr>
<tr>
<td>Yellow</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>FC</td>
</tr>
<tr>
<td>Green</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>1C</td>
</tr>
<tr>
<td>Blue</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>03</td>
</tr>
<tr>
<td>Grey</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>92</td>
</tr>
</tbody>
</table>

As a simple reference to some of the available colours:
Verilog signed values

In the early Verilog standards the majority of values were unsigned. It is now possible to declare values as signed but this is still an awkward ‘corner’ of the language. Sign extension rules are still confusing and (arguably) non-intuitive!

If adding/subtracting with operands and a result of the same size then the sign is not relevant; in two’s complement notation the same answer is produced. If an operand is truncated (i.e. the value is still representable, just the sign extension is lost) there is also no problem. Difficulties can arise when a new sign extension is needed.

The most likely forms of sign extension wanted in this lab. are probably:

- the addition of a smaller signed quantity to a larger accumulator
- the multiplication of signed values

```verilog
reg [11:0] aaa;
reg [7:0] bbb;
reg [11:0] ccc;
initial begin
  aaa = 12'h100;
  bbb = 8'h80;
  ccc = aaa + bbb; // Result 12'h180;
end

reg signed [11:0] aaa;
reg signed [7:0] bbb;
reg [11:0] ccc;
initial begin
  aaa = 12'h100;
  bbb = 8'h80;
  ccc = aaa + bbb; // Result 12'h080;
end
```

It is clumsy looking but sometimes seems sensible to sign extend explicitly, e.g.:

```verilog
bbb_ext = { {4{bbb[7]}}, bbb }; // Fill LHS with sign bit.
```

When multiplying, the result will (typically) be the sum of the sizes of the operands. If only the bottom bits are required this doesn’t matter; if a larger result is (or other bits of the result are) wanted then sign-extending the operands is important if they can be negative. It is suggested that these are declared as ‘signed’, which should do the job.

It is recommended you check some sample values in a simple simulation to see if Verilog is behaving as you expect.

---

**Rounding**

Puzzle for you: you want to round a number to the nearest representable value (e.g. a fixed point number to the nearest integer. How can you do this quickly and without tests?

(This can be a very useful technique in several circumstances.)
Troubleshooting the tools

[At present these are *unintegrated* notes they may still prove useful.]

Creating a cell

It is suggested that the Library Manager is a good tool to use. Select the appropriate directory and use **New ⇒ Cell View**. The usual options are tabulated below.

<table>
<thead>
<tr>
<th></th>
<th>Verilog</th>
<th>Schematic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell</td>
<td>&lt;cell name&gt;</td>
<td>&lt;cell name&gt;</td>
</tr>
<tr>
<td>View</td>
<td>functional</td>
<td>schematic</td>
</tr>
<tr>
<td>Type</td>
<td>Verilog</td>
<td>schematic</td>
</tr>
<tr>
<td>Open with</td>
<td>Read HDL</td>
<td>Schematics L</td>
</tr>
<tr>
<td>Effect</td>
<td>Text editor with template</td>
<td>Blank schematic</td>
</tr>
</tbody>
</table>

Cell compilation

When developing a Verilog module it needs to be compiled before it can be simulated. In this toolset the compiler is run when the default editor is closed – syntax is checked and errors reported at this time. Simply saving the file (from any editor) does not perform all the steps.

This may cause complications. In the event of a problem, especially if the interface has been edited, try opening the file with the default editor (e.g. from the library manager), *touching* it (e.g. add or delete a space character so it has been modified) and exit.

Setting up a simulation

The simulator may be started in various ways, including directly as **Tools ⇒ NC-Verilog…**

The Library/Cell/View should be set up to point at the design you are to test (e.g. COMP32211/my_thing/functional for a Verilog design). The ‘Run Directory’ contains the particular environment for this test and should look something like:

“$HOME/Cadence/COMP32211/my_thing_run1”; sometimes this name will be substituted automatically (but sometimes, mysteriously, isn’t). This directory will contain items such as the test stimulus file and the test output report (‘simout.tmp’). It is possible to have several different tests for a particular design, although this is often unnecessary.

Sometimes, when starting a simulation in a new directory, the tools will ask if you want to ‘keep the previous environment’; for a new design the answer should be ‘no’.
Wrong ‘modelName’?

If/when you copy a design – a handy way to start your own module – it will change most of the parameters to the new name. However (for some obscure reason) the ‘modelName’ is preserved.

It is quite difficult to discover a way to change (rather than just view) this. One way is to open the design (the Symbol will do) for editing and Edit ⇒ Properties ⇒ Cellview… (accelerator shift-Q) which brings up a dialogue box where this is editable.

‘Warning’ messages

‘Warnings’ are messages about possible errors which are not, necessarily wrong. Usually they are due to some oversight and are useful at helping fix and ‘tidy up’ a design.

There are some ‘warnings’ when checking the ‘drawing_engine’ schematic. These are due to the ‘cmd’ bus from the processor interface to the drawing engine; a 16-bit bus leaves the interface but most bits are not used. The naming of the adjacent bus segments identifies which bits are connected.

Other ‘warnings’ should be examined to see if there are possible problems emerging.

There will probably be numerous warnings in the synthesis report. This is because the synthesizer is doing things like stripping out logic whose outputs are never examined. Is that okay (indeed useful in saving resources) or is it indicating that you forgot something? Other common warnings include the stripping of delays (added to make simulation traces clearer) and the ‘finding’ of latches. The second of these may be part of the deliberate design – there are a number in the processor interface block used later in this lab. – but check to see you haven’t accidentally specified a latch where you intended a combinatorial circuit.

Space for contributions

If you experience particular issues with the toolflow which you think could benefit from some ‘tips’ here, please let us know. Ideally, you can write the advice!
Phase 2: Developing a module

At this point you should have:

- a high-level (algorithmic) model of a unit you want to build
- an understanding of the interfaces required
- a test environment able to exercise your design

Practical

Using the model from the previous module plus the knowledge/tests from the previous phase, produce a functional, synthesizable Verilog module. Prove this conforms to the same interface as the line module. Finally verify the output data against test data generated from the high-level model.

Demonstrate good test coverage.

Synthesize your unit to obtain some statistics on its size and speed.

How many clocks/pixel are required?

Deliverables

- A demonstration of your block at work.
- A listing of your module code.
- A test coverage report.
- A synthesis report showing estimates of resources required and maximum speed.

To complete this phase you may need to:

- modify the high-level model to produce appropriate test output
- devise some more test cases
- generalise your test environment slightly

Detailed block design

A simple translation of serial code into an RTL description is likely to result in an inefficient implementation. There are a few things to bear in mind when preparing the RTL.

The hardware can be parallel. Parallelism not only speeds up evaluation but may simplify the description by reducing the need for sequencing logic. Parallelism may be realised with separate functional units; operations may also be pipelined. It is sometimes difficult to picture this when looking at a text description, which appears serial. A sketch of an RTL block diagram often helps.
Example: Line drawing

The inner loop of Bresenham’s line algorithm, written in C, may look something like this:

```c
while (count > 0)
{
    x = x + 1;  // aaa
    e = e + dy; // bbb
    if (e > 0)   // ccc
    {
        y = y + 1; // ddd
        e = e - dx; // eee
    }
    plot(x, y); // fff
    count = count - 1; // ggg
}
```

At first glance this may appear to need (perhaps) seven cycles to execute; not so!

For example: aaa and bbb are independent operations and, if we choose to implement two adders, can be done in parallel. ccc is simply looking at the sign bit of e, so it takes no time. ddd and eee are also independent and could be done in parallel with each other. However they are predicated on the result of an earlier operation so they may need to wait for that result.

ggg is independent of all the other operations and can happen at any time, and the loop control is simply the sign bit (or zero status?) of count so it is easy to evaluate.

fff needs to wait for x and y but nothing depends on it so x and y could be captured and dispatched in the next cycle, so the calculation and plotting is pipelined.

The result is that all the calculations can be performed in a single cycle, eliminating the sequencer completely. The disadvantage in the above picture is that the cycle time may be rather slow because there are two arithmetic blocks (plus a multiplexer) in series which gives a long critical path. With a little more ingenuity this can be avoided – see the lecture notes.
There are some limits to the parallelism which is exploitable, of course. In this system we know that it takes two clocks to perform a memory cycle, even if we ignore any additional latency, so some optimizations may not be worthwhile. Also, only one pixel can be written per memory cycle – unless the pixels fall into the same word. Is there something to exploit there?

**Data representation**

In the line drawing example above, coordinates were represented as \((x, y)\). This may or may not be the best way to represent this information.

To plot a point the coordinates need to be transformed into an address. In this case the pixel address is \(640 \times y + x\) and the word address is this shifted right two places. The latter operation is trivial. Multiplications can be hard (time consuming) to do. However this multiplication is not too difficult. It can be performed by a single adder combined with some shifters.

\[
640 = 2^7 \times 5 = 2^7 \times (4 + 1) = 2^9 + 2^7
\]

Address conversion therefore requires two additional adders.

\[\text{pixel\_address} = (y << 9) + (y << 7) + x\]

An alternative way of representing the coordinates would be to calculate the address and then step such that:

\(x = x + 1;\) becomes \(\text{addr} = \text{addr} + 1;\)
\(y = y + 1;\) becomes \(\text{addr} = \text{addr} + N;\)

Where \(N\) is the programmed ‘stride’ on the frame store (in this instance, 640). Of course this can be reprogrammed for different display modes simply by changing this argument.

The example given in lectures uses this approach.

**Multiplication in Verilog**

The Verilog HDL supports the ‘\(*\)’ operator. However this is not generally synthesizable. One reason is that there are numerous ways of implementing multiplication logic.

The **FPGAs** we are using (Xilinx Spartan3) contain some dedicated (hard-wired) multiplier blocks: each is capable of an \(18\times18\)-bit combinatorial multiplication. The development tools

<table>
<thead>
<tr>
<th>Simple constant multipliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most standard graphics resolutions are chosen to make multiplication simple. Examples:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Width</th>
<th>Binary</th>
<th>Width</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>640</td>
<td>(2^9 + 2^7)</td>
<td>800</td>
<td>(2^9 + 2^8 + 2^5)</td>
</tr>
<tr>
<td>1024</td>
<td>(2^{10})</td>
<td>1152</td>
<td>(2^{10} + 2^7)</td>
</tr>
<tr>
<td>1280</td>
<td>(2^{10} + 2^8)</td>
<td>1440</td>
<td>(2^{10} + 2^8 + 2^7 + 2^5)</td>
</tr>
<tr>
<td>1600</td>
<td>(2^{10} + 2^9 + 2^6)</td>
<td>2048</td>
<td>(2^{11})</td>
</tr>
</tbody>
</table>
support these by being able to recognise the multiply operator and will substitute a multiplier
block. The tools are also able to spot multiplications by constants and can optimise them as
described above. Thus, in this case, **you can safely use multiplication in your Verilog source.**

Larger multipliers can be constructed from a sum of partial products. Consider the following
(decimal) multiplication:

\[
1234 \times 5678 = (1200 + 34) \times (5600 + 78) = 1200 \times 5600 + 1200 \times 78 + 34 \times 5600 + 34 \times 78
\]

\[
= (12 \times 56) \times 10000 + (12 \times 78) \times 100 + (34 \times 56) \times 100 + (34 \times 78)
\]

All the (difficult) multiplications are the smaller, bracketed operations.

The Xilinx Synthesis Tool (xst) will also do this. However the logic depth may result in a
slower than desired implementation. ‘Hints’ are allowed, for instance to pipeline this operation
over several clock cycles.

**Getting Started**

There is a minimalistic (function-free) block imported with your Cadence set-up:
drawing_dummy. It is recommended that you use the Library Manager to Edit ⇒ Copy this to
give you a template to edit. This will give you the interface (barring, perhaps, some wire ⇒ reg
swapping) and will give you a symbol which will fit in place easily in the next phase.
However see note on modelNames on page 44.

**Test coverage**

The test coverage tool included in this design flow is called ICCR (Incisive Comprehensive
CoveRage). It can be found in the Command menu on the NC Verilog window. It requires
output from the simulator so a simulation run must have been completed (and the simulator
quit) before running this.

**Verilog attributes**

Extensions to Verilog to guide and constrain specific tools are called ‘attributes’. There is
a specified syntax for setting attributes which looks like this\(^1\):

\[
(* \text{ full\_case}=1, \text{ parallel\_case}=1 *)
\]

However not all tool makers appear to have followed this syntax and other diverse ways
of setting attributes may be discovered. Here is another example\(^2\):

```verbatim
// synthesis attribute mult_style of modulename is
"pipe_lut";
```

Other manufacturers have other styles …

Here we don’t need (or want) particular guidance for the synthesizer – or other tools in
the flow – but that such things exist and you may well meet them in the future.

---

1. Example from IEEE Verilog 2001 standard
2. Example from Xilinx XST user guide
This opens a window with four tabs (at the bottom); the default display is the overall ‘Summary’. More interesting is the coverage of a particular module.

Use File ⇒ Open Test to get a browser and then descend from the test directory (probably called <design_name>_run1 or something similar) through cov_work ⇒ scope ⇒ test. This pops up a ‘Coverage Totals’ window.

This is a summary and is probably not the most useful view. Possibly most interesting details are in the Code/Data tab which allows the display of coverage of blocks, expressions and toggles.

You can now view the hierarchy in the test run. A menu under the right-hand mouse button allows you to ‘Show Details’ (in yet another window) of a chosen unit in the hierarchy. The details typically display a source listing with error reports below.

Roughly speaking, the coverage is given by:

- blocks: monolithic slabs of code which are always executed together
- expressions: decisions made as the code executes
- toggles: the bits in the variables having made both positive and negative transitions.

Your test coverage should try to exercise all blocks and expressions of your source code. It is unlikely that you will toggle all the data bits but you might manage to achieve this too. In practice it is probable that some eventualities will not be covered; are these things which can, reasonably, be justified?

Remember that automatic test coverage can only test that something happened; it can’t check that the right thing happened.

If you have time (and inclination) you could explore some other functions. For example, it is possible to examine the test coverage of the test code.

**Synthesis**

To synthesize your individual module, start with the NC Verilog window. Use Commands ⇒ XST to invoke the Xilinx Synthesis Tool.

This generates a report which is displayed as synthesis proceeds. It is also retained in ~/Cadence/COMP32211/xilinx_compile/<design_name>_xilinx_compile.log.

The log shows which structures were identified by the synthesizer and, later, gives a “Device Utilization Summary” enumerating the basic FPGA structures used. Below this is a “Timing Report” which gives an estimate of the maximum clock rate (with an unreasonable precision).

In larger syntheses the reports are given on a block-by-block basis as well as an overall summary. This can then be used to profile the design so that it can be reworked if necessary.

As a rough guide the rest of the target SoC requires about 30% of the device’s logic. Note that a 100% utilized device may not route successfully though.
Marking scheme

- 35% of coursework
- 5 marks for a demonstration of the block test traces.
- 20 marks for the module code: structure, clarity, commenting etc.
- 5 marks for the test coverage with justification for ‘untested’ features.
- 5 marks for the synthesis report, abstracting the requested information.
Phase 3: Integration

At this point you should have:

- a tested drawing unit of your own

To this we add:

- a proven SoC design with a compliant interface
- a test harness template including a frame store memory model

The purpose of this phase is to integrate your design into a larger system and to verify that it works correctly in context. If the preceding tests have been passed then the actual integration should be straightforward. The wider environment of the system is, briefly, explored.

The simulation also invokes a virtual screen (C code, as used previously) via a Programming Language Interface (PLI) to aid observability and assist with testing.

The result of this phase is to be the chip ‘core’ – that is all the logic functions that the chip will perform. The only thing omitted at this stage will be the ‘pad ring’. Instead of simulating the chip up to its boundary, first the core will be simulated and then the whole circuit board including the ARM microprocessor and the display.

VDU interface

In this example the frame store contains 300 Kbytes of data which must be transferred to the monitor sixty times each second (i.e. approaching 15 Mb/s). This is done by serialising the data through a DMA process performed by the VDU controller. For many years this interface was intended to drive a CRT (Cathode Ray Tube) display; modern systems use LCDs (Liquid Crystal Displays) but the interface is still compatible.

Each pixel is represented by one byte. Data may be fetched as 32-bit words so four pixels are fetched on each memory cycle; these are then, in turn, converted to analogue values for the dis-
play interface. Each byte is subdivided into three fields representing the intensities of Red, Green and Blue in the pixel. Red and green each have eight discrete levels; blue is represented by two bits which are converted the equivalent intensities \{0, 2, 5, 7\}. This is chosen as a compromise because human eyes are less sensitive to blue light.

The data is raster-scanned from top-left to bottom-right, in horizontal lines. At the end of each scan line the display is first blanked, then a sync. pulse is issued – in this interface using a separate digital signal – and after a subsequent blanking period the data for the next line is output.

The vertical timing goes through a similar, slower, cycle, counting lines rather than pixels.

The sync. pulses’ function is to locate the serial stream correctly on the display. A modern monitor will also time and count these to establish the display resolution. The blanking intervals are there for ‘flyback’: on a CRT it is necessary to steer the electron beam back across the display and it needs to be off (black/blanked) or the picture will be corrupted. On ‘old-fashioned’ analogue television (PAL) horizontal blanking accounted for almost 20% of the broadcast time – an obvious reason why digital video transmissions can contain more data.

**Memory timing**

The frame store memory timing needs consideration. The pixel clock rate – which is used to set the speed of all the units here – is 25 MHz which gives a period of 40 ns. The static RAM (SRAM) chips used for the frame store have a minimum access time of 55 ns. Thus a RAM access must take more than one clock cycle.

The RAM accesses are performed in two clocks, i.e. 80 ns. In this design it has been decided that incoming requests are acknowledged during the first of these two cycles. This allows the drawing engine time in the second cycle to decide if it will have another request ready and thus whether to keep its request asserted or not. The drawing engine can request access on any clock cycle but it will be granted, at most, on every alternate cycle.

In practice the drawing engine will not know the difference between clock edges where arbitration takes place and those in the middle of a memory cycle until it has made a request and ack is high at an active clock edge. At that point it knows that it is in the middle of a memory cycle and it can remove its request (if it has nothing else ready) or retain it (if subsequent output will be available in the following period). Although in principle it could track the phase after the first request it is better not to bother as there is no inherent penalty and any memory timing changes in the future can then be accommodated.
If the VDU is being refreshed – which is necessary on a regular basis – this process has priority. If this is not satisfied at the correct time ‘flecks’ will appear on the display where pixels are missed. (Some early displays did do this!)

For display purposes, each access fetches four pixels so there are ‘gaps’ between updates for the display. The drawing engine may be able to use these. The figure below shows drawing in progress just as a new scan-line of pixels starts to be displayed. Drawing can continue, but at half the previous rate.

An important point is that the drawing engine is not aware of the memory’s prior commitments. It is controlled simply by being informed when a cycle has been granted and the time between these acknowledgements may vary.

Note that it is possible to write up to four pixels at the same time if they are in the same word. In general, pixel addresses will be ‘random’ so this is not possible but it can significantly accelerate some operations. In particular, if filled objects are drawn then it is expected that several adjacent pixels will be shaded. It makes sense to fill these with ‘horizontal lines’ so that full advantage can be taken of the available drawing bandwidth. A simple ‘clear screen’ operation on this display can therefore be speeded up by a factor of four.

How fast can the screen be cleared without/with that optimization?

[Answer at the end of the next section.]

Some numbers

The display here has 640 × 480 pixels which are refreshed to the display at 60 Hz. Pixels are read four-at-a-time in 32-bit words. Thus 4,608,000 read cycles are needed each second. The memory is cycled in 80 ns which provides 12,500,000 cycles/s. The VDU therefore uses about 37% of the available bandwidth leaving about 63% available for other purposes. In this system the only other accesses serviced are the drawing engine and ‘direct’ CPU reads and writes. The latter are instigated individually by software and are therefore infrequent.

Note that increasing the display resolution by increasing the number of pixels or the number of colours would mean increasing the bandwidth devoted to refreshing the display and consequently decreasing the bandwidth available for drawing.
The memory bandwidth could be increased by:

- providing a wider (e.g., 64-bit) data interface
- providing separate memories for the two tasks, drawing a picture then swapping it to be displayed
- cycling the memory faster

Neither of the first two options are available on the board as it stands. The last option may be possible (with care).

Design simplicity suggests that the clock ratios should be simple integers. This has been used to govern the timing of the model.

Assuming a 25 MHz pixel rate the next simplest option would be to move to a 50 MHz clock and export a pixel on every alternate clock. The memory could be cycled in three clock periods (60 ns), although this is quite ‘tight’ as the memory access time is 55 ns and there will be some off- and on-chip delays. The screen would still require a new 32-bit word every 160 ns (now 8 clock periods) in its active phase. Eight is not a multiple of three so the display data would need to be prefetched into a FIFO and held until it is needed.

This approach would give slightly over 50% more drawing cycles at the cost of a significantly more complicated display control system.

The answer to the screen clearing problem. The exact elapsed time depends on the overlap of the operation and the active video. In one second there are \((12,500,000 - 4,608,000 = 7,892,000)\) memory cycles available for drawing. There are \((640 \times 480 = 307,200)\) pixels to write to. If written individually there could all be written in about 39 ms (average) which is a little over 25 times in a second. Remembering the display rate is 60 Hz this would take more than two displayed frames. With the optimization this would take about 10 ms, about 60% of the frame time.

Sanity check: with the optimization the writing four pixels will slot between each display read of four pixels. As writing can continue (at double rate) in blanking the expected answer should be somewhat less than one frame. The sums are therefore not obviously wrong! It’s always a good idea to do the rough checks for this.
**Test environment**

A stimulus file ($HOME/Cadence/COMP32211/drawing_main_stimulus.v) is provided to act as the basis for your customised stimulus file: $HOME/Cadence/COMP32211/drawing_main_run1/stimulus.v, or similar. It exercises the system through 20 ms which is just over one frame scan of the 60 Hz VDU or half a million clock cycles. The VDU controller is initialised to a state a few scan lines before the vertical synchronisation pulse which leaves about 1.2 ms before the active video begins. This initialisation is possible in the FPGA implementation but would require a reset input in an ASIC. In practice the machine as written will always reach a legal state at some stage so an ASIC without a reset would recover to a legal output within a fraction of a second and would not be noticeable as the monitor will take somewhat longer to identify and synchronise with the input stream. However a simulation would stay ‘unknown’ indefinitely, which would not be helpful.

The control interface has been simplified with a task (“bus_write”) which writes data to a chosen register address. Some register names have been defined too. (Although it is not part of either language, the same convention as in C programming is often observed, with ‘defined’ constants being in upper case, for clarity.)

Take the opportunity to view some of the output signals over the frame time. By zooming out it is possible to see the VDU controller activity on signals such as ‘fs_ncs’ (the frame store RAM select). An active frame and the vertical blanking should be apparent. By zooming into the active area individual lines, separated by horizontal blanking intervals, should become visible. Zoom further in and individual cycles become apparent. The pixel fetches can be seen on ‘fs_rdata’ and later on ‘pixel’, although the latter is not very interesting because the frame store has been initialised (i.e. the screen cleared) at the start of the simulation.

It’s worth noting the frame store read time (which is simulated with reasonable accuracy in the test harness) is more than a clock cycle. The simulated frame store contents have been zeroed at startup time to make it easier to see when they are changed. Without this the memory would read ‘xxxxxxxx’.

![Diagram of VDU signals](image)

Zooming out the pattern of the $VDU_{req}$/VDU_ack signals (and, in the absence of other activity, the frame store bus) exhibit activity patterns characteristic of a VDU. At one scale (e.g. zoom out fully – scale ~20 ms) a whole frame and the start of another should be visible. Zoom in maybe 100 times (scale ~200 µs) on the active frame and half a dozen lines should be discernable. Zoom in another 100 times (scale ~2 µs) and individual memory accesses are apparent.

Note that half a million cycles is quite a small trace when integrating a system on chip.
Practical

Demonstrate your drawing engine operating as part of the larger system.

The chip core has been assembled as a schematic: drawing_main. This should (largely) reflect the block diagrams you have seem previously. [Note: the important instances in this schematic have been given more meaningful names than ‘I0’, ‘I1’ … This should be helpful when you come to navigate the hierarchy.]

First, simulate this system ‘as is’. To assist with this a template for a stimulus file is supplied which is designed to give a reasonably useful initial display. The template also contains a Verilog memory model which provides the off-chip frame store RAM, complete with reasonably representative timing.

The template stimulates a line-drawing unit to draw a single line into the frame store, early on in the simulation run. The drawing interface input has been simplified by using tasks. The map of the various I/O registers is given on the following pages although how the arguments are used in your block is up to you.

It is difficult to ‘see’ whether the block is integrated correctly just from wave traces. As this is a graphics exercise it is much easier to examine output if a ‘screen’ is available. To allow this the ‘vscreen’ software used previously has been interfaced to the Verilog memory model using a PLI (Programming Language Interface) which allows it to reflect the contents of the frame store. The test harness starts and maintains this display using user defined tasks.

The source code of the user defined tasks is available as /opt/info/courses/COMP32211/libenv.c if you wish to peruse it.

[Note: this does not prove that a picture will be displayed, only that it is drawn. However we have already commissioned the display controller for you.]

View a selection of the signals in the waveform browser.

‘drawing_engine’ contains two existing drawing machines (clear and line) with two ‘dummy’ spaces for other units. A few wires have been ‘wired off’ because of the missing or slightly variant blocks. Your unit should fit into one of the ‘dummy’ spaces. Fit it, add some test data to the stimulus file and resimulate.

Deliverables

A demonstration of your block at work.

A schematic showing your unit in place.

The register-level documentation of your block interface.
Processor interface

The table below depicts the register map accessible to the processor interface. Note that the interface is 16 bits wide and the offsets specified are thus halfword offsets.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Size</th>
<th>Access</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Argument 0</td>
<td>16</td>
<td>R/W</td>
<td>Input R0 for drawing machines</td>
</tr>
<tr>
<td>1</td>
<td>Argument 1</td>
<td>16</td>
<td>R/W</td>
<td>Input R1 for drawing machines</td>
</tr>
<tr>
<td>2</td>
<td>Argument 2</td>
<td>16</td>
<td>R/W</td>
<td>Input R2 for drawing machines</td>
</tr>
<tr>
<td>3</td>
<td>Argument 3</td>
<td>16</td>
<td>R/W</td>
<td>Input R3 for drawing machines</td>
</tr>
<tr>
<td>4</td>
<td>Argument 4</td>
<td>16</td>
<td>R/W</td>
<td>Input R4 for drawing machines</td>
</tr>
<tr>
<td>5</td>
<td>Argument 5</td>
<td>16</td>
<td>R/W</td>
<td>Input R5 for drawing machines</td>
</tr>
<tr>
<td>6</td>
<td>Argument 6</td>
<td>16</td>
<td>R/W</td>
<td>Input R6 for drawing machines</td>
</tr>
<tr>
<td>7</td>
<td>Argument 7</td>
<td>16</td>
<td>R/W</td>
<td>Input R7 for drawing machines</td>
</tr>
<tr>
<td>8</td>
<td>Command</td>
<td>2</td>
<td>WO</td>
<td>Code to activate particular drawing unit</td>
</tr>
<tr>
<td>9</td>
<td>Output port</td>
<td>6</td>
<td>R/W</td>
<td>LEDs</td>
</tr>
<tr>
<td>A</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Unused</td>
</tr>
<tr>
<td>B</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Unused</td>
</tr>
<tr>
<td>C</td>
<td>IQ addr. (L)</td>
<td>16</td>
<td>R/W</td>
<td>Pixel read/write address (Low)</td>
</tr>
<tr>
<td>D</td>
<td>IQ addr (H)</td>
<td>4</td>
<td>R/W</td>
<td>Pixel read/write address (High)</td>
</tr>
<tr>
<td>E</td>
<td>IQ data</td>
<td>8</td>
<td>R/W</td>
<td>Pixel read/write port</td>
</tr>
<tr>
<td>F</td>
<td>Status</td>
<td>16</td>
<td>RO</td>
<td>See status definition</td>
</tr>
</tbody>
</table>

The status bits are mapped as follows:

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  0 0 Input buttons 0 0 Frame Vblank 0 0 DEbusy IQbusy
```

A drawing unit is invoked by writing its number to the command register. It will then take any parameters it requires from the parameter registers.
Pre-existing drawing interfaces

**Line** drawing (installed as unit #1). This is the unit optimised for size/speed as described in the lecture rather than the one used in the first exercise, so the parameters are different. It is capable of outputting one pixel per clock cycle.

The example draws a white line from (100, 100) to (200, 150).

<table>
<thead>
<tr>
<th>Argument</th>
<th># bits</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>10</td>
<td>Larger of distances between endpoints</td>
<td>100</td>
</tr>
<tr>
<td>r1</td>
<td>10</td>
<td>Smaller of distances between endpoints</td>
<td>50</td>
</tr>
<tr>
<td>r2</td>
<td>11</td>
<td>Address increment/decrement in larger direction</td>
<td>1</td>
</tr>
<tr>
<td>r3</td>
<td>11</td>
<td>Address increment/decrement in both directions</td>
<td>641</td>
</tr>
<tr>
<td>r4</td>
<td>16</td>
<td>Start address (low)</td>
<td>64100</td>
</tr>
<tr>
<td>r5</td>
<td>4</td>
<td>Start address (high)</td>
<td>0</td>
</tr>
<tr>
<td>r6</td>
<td>8</td>
<td>Colour</td>
<td>255</td>
</tr>
<tr>
<td>r7</td>
<td>—</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N.B. Numbers are in decimal!

**Clear** (installed as unit #3). This fills frame store memory with a prespecified byte (colour) covering the whole 640×480 pixels.

<table>
<thead>
<tr>
<th>Argument</th>
<th># bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>8</td>
<td>Colour</td>
</tr>
</tbody>
</table>

Wider environment

The overall system is then interfaced with the virtual screen again using a PLI (Programming Language Interface) in the Cadence simulator. This serves to allow further (visual) checks of function. It also illustrates the facility to use different simulation environments which is typically necessary for large scale chip development.

The figure below shows the processes which support the simulation. In this case the Device Under Test (DUT) is the Verilog design and the ‘vscreen’ is compiled from C.
Synchroniser

As an illustration, the synchroniser circuit used to bridge from the microprocessor input to the drawing/display clock is illustrated here.

Source code

```verilog
always @ (posedge uP_nwr, posedge cmd_ack)
if (cmd_ack) go <= 0;
else
  if (!uP_ncs && (uP_address == 6'h08))
    go <= 1;

always @ (posedge clk, posedge cmd_ack)
if (cmd_ack)
  begin
    go_1 <= 0;
    go_2 <= 0;
  end
else
  begin
    go_1 <= go;
    go_2 <= go_1;
  end
```

Equivalent schematic

[Diagram of the synchroniser circuit]
Timing

In the most cases ‘go’ is set between clock edges and the signal propagates normally.

If an unfortunate coincidence occurs, ‘go’ is set at the ‘wrong’ time and the subsequent flip-flop may become metastable. It has a clock period to resolve this before it is sampled.

‘cmd_ack’ is an asynchronous clear here, but is produced by the synchronous FSM.

The ‘back’ edge of the (active low) write pulse is used so that any accompanying data, which is captured by transparent latches, will be stable before ‘go’ is signalled

There is an assumption that another write cycle will not occur until the current one has been captured and acknowledged. In principle, the ‘go’ signal can be included in any ‘busy’ status, so the unit becomes busy even as the initiating cycle is completing.

Marking scheme

- 20% of coursework
- 10 marks for a demonstration of your block drawing on the virtual screen.
- 5 marks for a schematic showing your unit in place.
- 5 marks for your user interface specification (such as on page 58).
Phase 4: Synthesis

At this point you should have:

• a working SoC core

To this we add:

• a pad ring

If the previous phases have been completed successfully then this phase should be easy!

First the design needs to be provided with a ‘pad ring’. On an ASIC these are special cells which interface with the outside world. They are ‘traditionally’ placed around the perimeter of the die for connection purposes but may be scattered in some technologies. Pads provide electrical drive capacity, electrical protection and usually, nowadays, a voltage level shift. Pads may also provide a ‘tristate’ capability to allow bidirectional interfacing, e.g. to memories. There is also a need for special pads to provide electrical power connection.

The FPGA has pads already in place. It is possible to allow the compilation tool to assign signals to pads but our FPGA is already on a circuit board so the pad placement is already constrained. There are different ways of specifying the constraints; for the purposes of this exercise this has been done with a schematic with the added ‘LOC’ property.

Open the schematic drawing_top. This contains the ‘core’ design and the pads, together with a small amount of logic to control the interfaces. Your design should already be in this hierarchy and can now be compiled.

As a reminder, you can invoke the FPGA compiler from the ‘NC-Verilog Integration’ tool. First ‘Generate Netlist’ then ‘Xilinx Synthesis’ should produce the following outputs:

• ~/Cadence/COMP32211/xilinx_compile/drawing_top_xilinx_compile.log
• ~/Cadence/COMP32211/xilinx_compile/drawing_top.bit

amongst others. The first of these contains the records of the compilation and gives some feedback on how the design might perform. The ‘bitfile’ will only be produced if compilation was successful.

Hopefully no logical errors have escaped the simulation process by this stage. The major risk is that your design has exceeded the capacity of the device. This is easily possible but shouldn’t be necessary for the scale of designs here. There are numerous possible causes for this and you may need to consult a member of staff. Here’s an example:

reg [15:0] my_mem [0:1023];
always @(posedge clk)
if (blank) my_bus <= 0;
else my_bus <= my_mem[addr];

What’s wrong with that? [Answer overleaf]
Logically nothing, but it doesn’t map to the Xilinx FPGA’s ‘block RAM’.

The logic following the memory is before the latch and so is inserted serially. The ‘block RAM’ must have an edge-triggered latch immediately at its output. The memory is therefore synthesized from logic cells which is extremely inefficient in space; meanwhile the ‘block RAM’ is unused.

A small change in the design may allow the ‘block RAM’ to be used and the design to fit. Note that there are some small differences in the functionality.

- The timing of blank_1 is different from the timing of blank. It could be provided by delaying blank through a flip-flop.
- The timing path into the register has been shortened.
- The time taken for the stage after the register has been lengthened.

The latter two effects may be slight here, but greater if the combinatorial logic is more complicated. In fact, in the FPGA case, even if both designs compile the exploitation of the ‘block RAM’ will almost certainly result in a faster design.

```
reg [15:0] new_mem [0:1023];
reg [15:0] new_mem_out;
wire [15:0] new_bus;

always @ (posedge clk)
begin
  new_mem_out <= new_mem[addr];
  blank_1 <= blank;
end

assign new_bus = new_mem_out & {16{~blank_1}}; // See note below
```

Note: this assign could have been written – perhaps more clearly – with an if and blocking assignments. This style has been chosen to exemplify the replication operation. If this were omitted then ~blank_1 would be zero-extended to 16 bits … which should be detected as an error in the functional tests.
The log file

~/Cadence/COMP32211/xilinx_compile/drawing_top_xilinx_compile.log

This file contains the compilation log, including the error reports. In common with many such tools there are different classes of ‘errors’.

- **ERROR**: something is wrong which cannot be resolved.
- **WARNING**: this may be a mistake but it can be ignored.
- **INFO**: your design may benefit by following this advice.

A very good habit is to check all the error reports to see if something is wrong.

Some example ‘WARNING’s from the compiler:

- “The signals <…> are missing in the sensitivity list of always block.”
  - Probably an oversight that wants fixing.
- “Delay is ignored for synthesis.”
  - Ignorable; delays were placed for clarity in simulation.
- “Input <…> is never used.”
  - Should it have been?
- “Signal <…> is assigned but never used.”
  - Should it have been?
- “Signal <…> is used but never assigned. Tied to value 0.”
  - Probably an oversight.
- “FF/Latch <…_0> is unconnected in block <…>”
  - Perhaps a (in this case the least significant) bit in a register was not needed.
- The signal … has no load
  - Likely to be a dangling wire. May have been left for later modification.
- “Found …-bit latch for signal <…>”
  - A transparent latch has been inferred. This is easy to do by mistake; for example incompletely specifying all the ‘case’ s in a statement may do this, which is why including a ‘default’ is a good habit.
- “Due to other FF/Latch trimming, FF/Latch <line_address_1> (without init value) has a constant value of 0 in block <drawing_crtc>.”
Some ‘WARNING’s are almost unavoidable in any sizeable design. However beware of ignoring them because something important could be lurking in there. It is a good idea to address them where possible simply to reduce the volume of the output files that need checking.

Note that the synthesis process will try to strip out any unused logic. Thus if a signal is not connected to any inputs (or a pad) its driving logic can be removed. This may allow further signals to be removed. In the extreme case, if no output signals are connected to pads, the design could have no observable effect and all the logic may be optimised away!

The log file also reports on ‘macros’ the structures identified by the synthesizer such as registers, counters, multiplexers, adders, RAMs, ROMs, FSMs … These are succeeded by a report in terms of the FPGA’s primitives – primarily ‘LUT’s and ‘FD’s (Look Up Tables and D-Flipflops) but maybe block RAMs etc. as well – and a summary of how much of the total resource is required. This information matters most when a design will not fit a device as it can indicate which block(s) may require optimising.

Subsequently a timing estimate for the unit is generated. Whilst not precise, this gives an indication of which blocks may be the speed limiting factor for the device. In this design the target clock speed is a mere 25 MHz so it should be easy to meet; if not you need a serious look back at the RTL structure and the depth of the combinatorial logic blocks.

When all the blocks have been reduced to FPGA primitives they can be placed and routed onto the device itself. After the final checks – and if there are no ERRORs – a bitfile is generated which can be loaded onto the FPGA itself.

**Downloading**

The most appropriate way to download to the FPGA is to use a Komodo feature.

Ensure a lab. board is connected (and switched on) and start the appropriate Komodo version:

```
start_komodo COMP32211
```

Select ‘Features’ and ‘Spartan-3 XC3S200’. Enter the name of the bitfile (there is a browser to help with this: it should be in directory `~/Cadence/COMP32211/xilinx_compile/`) and ‘Download’. When the download process is complete the design can be used.

If there is a monitor connected (and switched on) a picture should appear. After power-up this will be the random(ish) contents of the on-board SRAMs. The next steps are up to you, either using software control or ‘poking’ the values via the front-panel.

**Bus interface**

The 16-bit wide bus interface to the FPGA is at 3000_0000₁₆. Note that previously documented address offsets are for halfwords and will need multiplying by two to match the ARM byte-addressable space.
Practical

Synthesize your drawing_top and download it to the FPGA. Demonstrate your drawing function.

When you first start the unit the frame store RAM will be uninitialised. For most drawing experiments you will therefore want to clear the screen. This can be done with the ‘clear’ unit, setting the colour to black (00).

Write a short demonstration programme for the ARM.

Deliverables

Design demonstration, with drawing on-screen.

Test software listing.

Answer the following questions about the overall design:

- What resources, in terms of FPGA ‘Slices’ and other elements, are used by your block?
- What resources on the FPGA are for the whole SoC?
- How ‘full’ is the device?
- What is the estimated maximum speed of the SoC?

NOTE: we want the abstracted numbers not a copy of the whole synthesis report!

Write a short critique (about one paragraph per question, above) of your block design.

- What, in hindsight, was well designed?
- What would you do differently if you were starting again?
- What were the major surprises and lessons learnt?

Direct frame store access

It is possible to read and write pixels ‘directly’ from the ARM. The unit which does this has been called the ‘inquisitor’ (IQ). To read a pixel first set the pixel address in the address registers: there are two to accommodate the 20-bit pixel address. Writing to these initiates a frame store read and soon afterwards (it should be fast enough to beat the processor) the pixel value will be available via IQ_data. To modify this, write to IQ_data and this will initiate a write cycle.

Because they are necessarily low bandwidth (at software speeds) these accesses have a higher priority than the drawing engine. Losing one opportunity to output when drawing a line is a small handicap whereas waiting for a line (worse, a ‘clear screen’) to finish before allowing processor access would be a serious latency penalty.
Marking scheme

- 25% of coursework.
- 5 marks for the demonstration.
- 5 marks for the test software.
- 5 marks for the SoC synthesis values.
- 10 marks for the critique.
The FPGA in the lab.

We are targeting a Xilinx **XC3S200 FT256**. This is nominally capable of implementing designs of up to 200 000 ‘gates’, although exactly how that is interpreted is hard to define. [You can find the full datasheet at http://www.xilinx.com.]

This device has 480 CLBs (Configurable Logic Blocks) each comprising four ‘slices’ with each ‘slice’ containing two LUTs (Look-Up Tables) and two D-type flip-flops. Each LUT is a 16 bit RAM so is capable of implementing any Boolean function of up to four inputs. Think of it as a truth table.

Twelve blocks of RAM. Each RAM block has 18 Kbits (think 2 Kbytes + byte parity), is synchronous (i.e. inputs and outputs are edge-triggered) and dual-ported so a read and write can be performed in the same cycle.

Twelve multiplier blocks. Each block multiplies two 18-bit words to produce a 36-bit result.

Four DCMs (Digital Clock Managers), capable of multiplying, dividing and phase-shifting clocks, together with their associated clock distribution networks.

173 user I/Os, configurable as input, output or bidirectional signals. For your design the necessary pin constraints (getting each I/O signal to the correct pad on the PCB) are supplied at the top level of the design hierarchy.

**Why care?**

In one sense the design should be independent of the target technology. When compiling and mapping a design the tools will do the best they can. However some knowledge of the underlying technology can make the difference between meeting and failing to meet design objectives such as size or speed.

One such case is the use of RAM on an FPGA. In the device above the built-in RAMs have synchronous interfaces. If the design specifies asynchronous registers or memory (the off-chip frame store we use is an asynchronous RAM, for example) then these RAMs can’t be used. The compiler would need to make memory cells from the logic tables which will be slower and VERY resource hungry; it will soon run out.

A small difference in the way the design is specified can make a big difference to the result.
Appendix A: Floyd-Steinberg dithering hints

Imperative software implicitly works by processing variables (in memory) one at a time. Hardware is not like that: there is a desire to exploit data flows and process as much as possible in parallel. Local variables will be kept in registers with lots of bandwidth available; access to a large memory is (relatively) expensive.

- Before proceeding, make sure you understand the principle of the algorithm.
- For convenience the differences accumulated from the quantisation process will be referred to here as ‘errors’; this doesn’t imply that they are wrong.
- Here we deal with a single ‘shade’ rather than each colour component. It is easy to extend the implementation at the end.

Minimising the memory use

The shade of each pixel is influences its neighbours: in this algorithm just the neighbours which have yet to be plotted. The are the pixel immediately to the right – which will be the next one to be plotted – and three neighbours from the row below. Because the length of a row is ‘large’ these ‘errors’ need to be stored which implies an addressable memory. **This memory need only be the size of a single row because it can be reused as each row is scanned.**

The ‘error’ terms necessarily have more bits than the frame store’s pixel locations (this is what the algorithm is about!) but the memory can be smaller. There are also some accesses to this memory competing with frame store writes. As it will fit, **it therefore makes sense to separate this memory and keep it locally on the FPGA.**

Let’s designate the error terms as in this figure. X represents the current position.

- A is derived from X and an error from the previous row
- B is derived from X
- C is derived from the previous B and X
- D is derived from the previous C and X and nothing more

Thus A (in part) needs an input from the ‘error’ memory. D needs to be saved in the ‘error’ memory. B & C can be kept locally for further processing.

As the processing moves left to right, (the component of) A can be read and D can be written into the same memory, recycling it. **There is one read and one write needed per pixel.** There are various solutions to this but, as the FPGA has dual-port memory we can perform these operations in parallel.

The FPGA block RAM is synchronous, so the read data arrives on a clock edge; it is not possible to read the data and use it in the same cycle. The data needs to be ‘ordered’ in advance.

---

1. If you can think of a better name, suggest it!
Dataflows

This figure shows a possible dataflow model; values are latched where an arrow enters a box and these can all happen simultaneously.

‘X’ calculates both the pixel shade to output – based on the desired shade, modified by the ‘error’ generated from the previous row (A’) and that from the previous pixel (stored as B and multiplied up on return) – and the new ‘error’ term from the current quantisation which is accumulated as the scan proceeds; the final value is written directly to the local memory.

The ÷16 (which is, of course, a right shift) is omitted from the figure. It can be done at any stage in the process – for example as ‘A’ is imported.

Rounding, data precision and signed values

The exact rounding approach used may influence your final result, slightly.

Assume (inaccurately in this example) that a pixel shade is represented by four bits but the user wants to specify an 8-bit value (hence the need for dithering). The initial quantisation error will thus be a 4-bit value … which is then ÷16 to require a 12-bit value.

However (in this case) the difference will only require eight bits because the most significant bits (pixel) will have been plotted. The accumulated ‘error’ is thus a signed value which requires fewer bits than might, at first, be thought.

Note that the second pixel’s difference uses a fraction of the fractional difference from the first (needing more bits to handle exactly); pragmatically this value needs truncating or rounding to avoid the variables growing to an unbounded size. The exact ‘rounding’ approach used will influence the output pattern slightly but this should not make ‘devastating’ differences.

Dealing with signed values in Verilog is dealt with on page 42.

Special cases

- The first row of pixels can have no previously stored influences so these should be suppressed.
- The first column of pixels needs a value prefetching before plotting can begin.
- Writing to the last column of pixels has not updated the last ‘error’; another cycle is required.
  - The last column of one row is followed by the start of the next, which may suggest an optimisation.
Appendix B: Trigonometry

The ‘traditional’ – and probably previously encountered – mechanism for calculating a sine or cosine is a Taylor series. Thus: \[ \sin \Theta = \frac{\Theta}{1!} - \frac{\Theta^3}{3!} + \frac{\Theta^5}{5!} - \frac{\Theta^7}{7!} + \ldots \] (where \( \Theta \) is in radians, of course). This involves a number of multiplications, which can be computationally expensive. (There are, apparently, divisions too – which are even more expensive – but these are by constants so can easily be transformed into multiplications by the (precomputed) reciprocal.)

As an alternative, this description concentrates on the **CORDIC** (COordinate Rotation DIgital Computer) algorithm which can reproduce these functions efficiently without generalised multiplication.

To picture how this works, consider the adjacent figure. A point \((x, y)\) is to be rotated about the origin by an angle \( \gamma \).

The right-angled triangles simply highlight the coordinates \(x\) and \(y\) before the transform and how these move after the rotation.

After a rotation by angle \( \gamma \):
\[ x' = x \cdot \cos(\gamma) - y \cdot \sin(\gamma) \]
\[ y' = x \cdot \sin(\gamma) + y \cdot \cos(\gamma) \]

This can be checked geometrically in the figure.

Expressed as a matrix multiplication:
\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix}
= \begin{bmatrix}
  \cos(\gamma) & -\sin(\gamma) \\
  \sin(\gamma) & \cos(\gamma)
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]

With the identities:
\[ \tan(\gamma) = \frac{\sin(\gamma)}{\cos(\gamma)} \]
\[ \sin^2(\gamma) + \cos^2(\gamma) = 1 \]

So:
\[ \cos^2(\gamma) = \frac{1}{1 + \tan^2(\gamma)} \]
\[ \sin^2(\gamma) = \frac{\tan^2(\gamma)}{1 + \tan^2(\gamma)} \]

The transformation can be rewritten as:
\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix}
= \frac{1}{\sqrt{1 + \tan^2(\gamma)}}
\begin{bmatrix}
  1 & -\tan(\gamma) \\
  \tan(\gamma) & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]
Although that doesn’t initially look easier, *some* values of $\gamma$ inevitably have simple tangents, such as: $1, \frac{1}{2}, \frac{1}{4}, \ldots$ Neglecting (for a moment) the normalising factors, this makes the multiplications (divisions) into shift operations: $x' := x - (y \gg i)$ $y' := (x \gg i) + y$

The relevant angles turn out to be:

<table>
<thead>
<tr>
<th>Tangent</th>
<th>Angle (degrees)</th>
<th>Angle (radians)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45.000°</td>
<td>0.78540</td>
</tr>
<tr>
<td>$\frac{1}{2}$</td>
<td>26.565°</td>
<td>0.46365</td>
</tr>
<tr>
<td>$\frac{1}{4}$</td>
<td>14.036°</td>
<td>0.24498</td>
</tr>
<tr>
<td>$\frac{1}{8}$</td>
<td>7.125°</td>
<td>0.12435</td>
</tr>
<tr>
<td>$\frac{1}{16}$</td>
<td>3.576°</td>
<td>0.06242</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Any angle (to a given precision) can be specified as a *series of successive rotations* in one direction or the other through each of these angles in turn. The direction – always rotating towards the desired value – dictates the sign of the angle which is the sign of the tangent since $\tan(-\gamma) = -\tan(\gamma)$; the magnitude is the same. The angles (inverse tangents) need to be known but these are constants so can be worked out in advance and stored in a look-up table.

One point to note is that, for small angles: $\tan\gamma = \gamma$ when $\gamma$ is expressed in radians.

E.g. $\tan\left(\frac{1}{16}\right) = \tan(0.0625) = 0.0624$ or $\tan\left(\frac{1}{1024}\right) = \tan(0.00097656) = 0.00097656$

Thus the look-up table can typically be disregarded after the first ‘few’ entries since the small angle approximation will be ‘good enough’ and the angle is simply halved on each step.

For example, to rotate anticlockwise by 37° (say) the rotations would be:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Adjustment</th>
<th>Accumulated angle</th>
<th>too …</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000°</td>
<td>0.000° small</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>+45.000°</td>
<td>45.000° big</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-26.565°</td>
<td>18.435° small</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+14.036°</td>
<td>32.471° small</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>+7.125°</td>
<td>39.596° big</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-3.576°</td>
<td>36.020° small</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+1.790°</td>
<td>37.810° big</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>-0.895°</td>
<td>36.915° small</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>+0.448°</td>
<td>37.362° big</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>-0.224°</td>
<td>37.138° big</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-0.056°</td>
<td>36.971° small</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>+0.028°</td>
<td>36.999° …</td>
<td></td>
</tr>
</tbody>
</table>
To calculate (say) the sine of an angle, start with a unit vector \[ \begin{bmatrix} 1 \\ 0 \end{bmatrix} \] and rotate this to the desired angle. The resultant y coordinate will be the sine (the x coordinate will be the cosine) except that the scaling factors have been neglected.

At each iteration the vector should be scaled by:

\[
\frac{1}{\sqrt{1 + \tan^2(\gamma)}}
\]

This depends on \( \gamma \) – and hence the iteration – but not on the direction of the rotation. The values are always:

\[
\frac{1}{\sqrt{2}}, \frac{2}{\sqrt{5}}, \frac{4}{\sqrt{17}}, \frac{8}{\sqrt{65}}, \frac{16}{\sqrt{257}}, \ldots, \frac{2^i}{\sqrt{1 + 2^{2i}}}
\]

- Obtaining these values is complicated – but they can be precalculated and looked up.
- Multiplying is quite complicated – but doesn’t need to be done every iteration because the appropriate set of coefficients can be pre-multiplied.
- Even one multiplication is a bit complicated – but can be applied at the start (to the ‘unit vector’) rather than at the end; it is thus multiplying a known constant (for a given number of iterations) by ‘1’.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Scaling factor</th>
<th>Scaling factor</th>
<th>Cumulative scaling factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>1</td>
<td>( \frac{1}{\sqrt{2}} )</td>
<td>0.70711</td>
<td>0.70711</td>
</tr>
<tr>
<td>2</td>
<td>( \frac{2}{\sqrt{5}} )</td>
<td>0.89443</td>
<td>0.63246</td>
</tr>
<tr>
<td>3</td>
<td>( \frac{4}{\sqrt{17}} )</td>
<td>0.97014</td>
<td>0.61357</td>
</tr>
<tr>
<td>4</td>
<td>( \frac{8}{\sqrt{65}} )</td>
<td>0.99228</td>
<td>0.60883</td>
</tr>
<tr>
<td>5</td>
<td>( \frac{16}{\sqrt{257}} )</td>
<td>0.99805</td>
<td>0.60765</td>
</tr>
<tr>
<td>6</td>
<td>( \frac{32}{\sqrt{1025}} )</td>
<td>0.99951</td>
<td>0.60735</td>
</tr>
</tbody>
</table>

(Note that the scaling factors tend towards 1 rapidly as the angles become small.)

The scaling factor is always <1 so starting with the appropriate size means the vector magnitude will ‘grow’ to 1 after the determined number of iterations. For example, for seven iterations starting with a vector representing (0.60728, 0) will result in a unit length vector at the appropriate time.

Thus, no multiplication is needed for scaling either.

Below is the same example previously used (+37°) given over 12 iterations showing how the x and y coordinates evolve. In this example the values are arbitrarily rounded to 5 decimal places at each step, to imitate the sort of limitation one might expect in a digital computer. Rounding errors might, therefore, be expected to accumulate.
For comparison purposes:  \( \cos(37^\circ) \approx 0.79863551; \quad \sin(37^\circ) \approx 0.601815023 \)

### Implementation

From a hardware perspective it should be apparent that \( x \) and \( y \) are updated ‘simultaneously’. The most expensive unit is liable to be the shifter, not because it is complex but because it needs to shift by different numbers of places. There is a trade-off between speed (number of cycles or critical path) and hardware cost (size) as to which form of shifter to implement. (There are some alternatives in the course notes from COMP22111.)

You may have spotted that most numbers in this section are not integers. This does not mean you need a floating point unit; the values are easily represented as fixed point values by simply (mentally!) shifting an integer by \( 2^N \) places. Thus, for example:

\[
9A_{16} \gg 20_{10} = 631044_{10} / 2^{20} \approx 0.60181
\]

Fixed point representation is discussed more in the section on Mandelbrot generation (page 21); here it is simple because there is no multiplication (only addition/subtraction and relative shifting) so the scaling is of no real consequence.

The 20 places scaling above is an arbitrary choice (as was the 12 iterations earlier).

Note that for purposes of plotting on a 640x480 screen the resolution is ~9 bits so any precision much above this range will not be significant. Choose a precision which is appropriate for your application!
References

http://www.verilog.com/

http://www.asic-world.com/verilog/

IEEE Standard 1364-2001 – Verilog Hardware Description Language
