VECTOR DISPLAY PROCESSOR

CONTENTS

GLOSSARY

Signal **asserted** (**negated**), Signal set to 1 (0).

Cycle: one cycle of the (common) hardware clock.

Synchronous: signals change shortly after the clock active edge, all internal state is implemented using flip-flops clocked by a single active clock edge.

Interface: set of signals connecting one hardware block to another

Interface protocol: the rules which indicate the timing of data transfer across an interface.

Handshake: interface protocol in which two signals together determine data transfer, which each signal controlled by one of the two interfacing blocks. This allows either block to delay transfers if it is not ready.

Host processor: the CPU which would issue commands to the VDP block. The host processor does not exist in hardware, it is simulated in the testbench by some VHDL that issues a sequence of commands as rapidly as the interface protocol allows to your hardware.

External (video) RAM. A hardware memory that stores pixel state and in an application would allow access to this by separate display hardware. The video RAM is simulated by the testbench.

Cache (or cacheing). Block of hardware which holds a small number of contiguous memory locations to allow for faster access. In this project pix_word_cache is an example.

INTRODUCTION

The purpose of this project is for you to learn how to implement larger hardware designs. In this case VHDL coding is only part of the problem: design specification, working in a team, performance evaluation, are all also important.

DESIGN SPECIFICATION & EVALUATION

The project lecture (see web pages) provides a complete top-level specification of your hardware, which interfaces to a host processor & a video RAM and draws lines based on host processor commands. You are provided with a sophisticated testbench which will automatically check correctness of your hardware for the set of host processor commands in a file. You can therefore design your own tests by writing your own host processor commands.

DESIGN IMPLEMENTATION

The project lecture outlines the expected design implementation strategy using two sub-blocks: Draw Block (DB) & (RCB) RAM Control Block. These are instantiated within a structural entity VDP to make the complete Vector Display Processor hardware. Each member of the pair will be responsible for one of these blocks, although both must pay attention to the interface between the blocks. Cooperation is expected so helping your partner, typically by solving design problems and debugging, is part of the project work.

PROJECT STRATEGY

These notes make it clear how to write a simple deliverable which satisfies the various assessment criteria. All groups, even if very ambitious, are advised to ensure that they can fulfill all the requirements of the first two deliverables for a simple design, before they attempt a cleverer or more ambitious design.

Appropriate use of constants and types can make it easy to retrofit more complex design into an initial working simple design.

Students should note that most of the marks available from assessment do not depend on how clever or ambitious is your hardware design. Time however is very limited, so be sure you can satisfy all these deliverables.

The final deliverable has some marks which depend on how fast is your design, and how modular and cleanly written. It also has marks for your understanding of the design issues. Clever designs can in principle gain credit for all these issues.

USE OF PRE-PROJECT EXAMPLE CODE

I assume that each pair has working example code written for all the pre-project exercises, and that this code will be incorporated in the project. Students may use any exercise code (including my model answers) in their work, although it is expected they will normally use their own or their pair's code. Exercise code may be incorporated as tested, using entity instances, or processes may be extracted from examples and used directly in an architecture. There are pros and cons to both approaches. Structural code can be difficult to read and debug, but is more modular.

You may make arbitrary changes to the exercise code but remember that changes may break tested code. I would suggest that you aim to get a fully working deliverable with

minimal changes to example code, and only then make further changes necessary to optimise performance etc.

PARAMETRISATION

You will gain credit for writing code which can be configured using parameters (constants or generics). You are encouraged to use parameters consistently throughout your design even though my testbench will only test fixed size hardware.

DRAW BLOCK

A large part of the DB code has already been designed and tested in the pre-project coding examples (draw_any_octant). The notes below indicate detailed design issues that remain:

You need to combine each (X,Y) pixel coordinates from the previous & current host commands to set start and end (X,Y) values in the draw_any_octant block for each draw command, or for clearscreen. The previous host command values can be stored in a register for this purpose.

You need to determine the correct octant & therefore choice of xbias, negx, negy, swapxy for a given start & end point. Note that xbias has an almost invisible but real affect on the resultant pixels. If it is incorrect your output will look correct but fail my testbench on some lines.

The (X,Y) draw_any_octant outputs must be sent to the RCB as pixel draw commands. When the RCB is not ready to receive these drawing must be delayed: the disable signal to d_a_o will help with this.

The signal RCB_FINISHED from the RCB is asserted when the RCB has no outstanding pixels to write to memory. This is passed back to the host processor as DB_FINISHED when, also, the DB has finished all its pixel output and has no more host processor commands. DB_FINISHED at the end of a test will terminate the testbench.

The specified interface between DB and RCB is mandatory.

The DB code must wait for the next host command. In practice my testbench will feed commands as fast as it can to the VDP, however this cannot be guaranteed since it is not part of the host interface specification.

RAM CONTROL BLOCK

Many of the design issues in the DB apply to the RCB, so read the above section carefully.

You are expected to use a single **pixel_word_cache** unit, which will interface with **ram_fsm** easily and can be made to interface with the DB. You will need a register to store the “current memory word” address of the current 4X4 pixel square, so that you can detect when the next command leaves this and therefore it is necessary to write pix_word_cache out to RAM. The least significant two X & Y address bits determine the position within this 4X4 pixel square, so the remainder of the X & Y bits must be stored as a number which determines the “current memory word” address.

It is possible (conceptually simplest and easiest to impelment) to implement RCB by modifying **ram_fsm** to incorporate the whole of the RCB behaviour in a single FSM. Alternately you can use an additional FSM which runs in parallel with **ram_fsm** and which controls the rest of the RCB. Think of this conceptually as being "series" or "parallel" combination of FSMs. The parallel solution is potentially more optimal, and also more hierarchical, because **ram_fsm** is already tested, but it requires clear thinking about the relative timing of the two state machines. Practically, many people find this parallel solution more difficult to get working than the series one. If you have any trouble understanding this paragraph you should implement the series solution.

You may modify the RAM timing in **ram_fsm** to increase performance. However you will not know what would help until you have both blocks written and synthesised, so you do this at your own peril in advance of fully working hardware.

The signal RCB_FINISHED must be asserted when all outstanding pixels have been written to RAM. In order to ensure termination of the testbench you must implement a timer which, when there have been no more pixel output commands from the DB for a suitable length of time, writes out the current pix_word_cache contents to RAM, after which there will be no outstanding pixels and RCB_FINISHED will be asserted.

TESTBENCH ISSUES

The given testbench for complete VDP testing must not be changed. However it uses a package **config_pack** which contains constant definitions whose values you may change as required. See "Module-level Verification" for information about how simulation clock frequency, which you can adjust, affects correctness of pre & post synthesis code. When your code is tested this may be with a standard config_pak, so although you can change constant values in the package you cannot add or remove constant definitions.

REQUIRED FILE STRUCTURE

You are required to submit a file **project_pack.vhd** which contains a package **project_pack**. This should contain all type & constant definitions you wish to use yourself that apply to both **rcd** and **vdp**. You are required to submit your project code as 3 files, **rcb.vhd**, **db.vhd**, and a top-level **vdp.vhd**.

The top-level file **vdp.vhd** must have the **vdp** entity definition defined in the testbench. The two sub-blocks are expected to contain one entity each (**rcb** or **db**) and be owned one each by the two project team members. Other entities, for example the 4 exercises (or modifications thereof) should be added to your project as design files **ex1.vhd**, **ex2.vhd**, **ex3.vhd**, **ex4.vhd**. Each design file may contain more than on design unit (e.g. an entity and its package). Your design files will be analysed (compiled) in the order:

config_pack, project_pack, ex1, ex2, ex3, ex4, db, rcb, vdp.

Please respect these names (**even though it will mean renaming previous exercise files**). You may put additional design units, for example your own entities, into any of these files if this is necessary.

Make sure you have no backwards dependencies which prevent compiling in the order above. Make sure your files compile under VHDL 1993 compiler.

Note that config_pack is NOT owned by you. The constants therein will have values adjusted during assessment as required. However you may be sure that all of the constants in config_pack will be available to you.

TEST DATA

You are encouraged to use your own sets of commands for testing. The provided commands are not a complete test set.

APPENDIX A: DATA INTERFACES

In the VHDL course examples & project there are a number of interfaces where data is transferred from one block to another. In each case the transfer must be handshaken: the source will not be able to supply new data on every clock cycle, the destination will not be able to receive data every clock cycle. Thus both source and destination must have the ability to wait for the other side to catch up.

The design of data interfaces is an important and bug-prone part of hardware design. In this project work every interface you use has the same pattern, controlled by two so-called **handshake** signals as in Table 1.

All data transfer is synchronised to a common clock so we need consider only what happens on each clock cycle (clock edge).

Signal	Driven by	Meaning to source	Meaning to destination
Start	source	Drive '1' when new data is available, keep '1' until data is successfully transferred	When '1' new data is available on bus.
Delay	destination	When '1' new data on bus cannot be received this cycle, so data transfer operation length must be extended.	Drive '1' when new data cannot be received immediately to extend length of data transfer cycle.
Bus	source	data to be transferred (any number of signals)	

Table 1 - interface signals

Start	Delay	State

0	0	idle, waiting for source
0	1	not allowed
1	0	New data is transferred from source to destination, one new item every cycle with these handshake signal values.
1	1	The new data cannot be transferred yet and remains on bus until destination is ready .

Table 2 - interface states

APPENDIX B: EXTERNAL RAM CONTROL

In the project work it is expected that you will (at least initially) use the example 3 FSM code to control RAM access. The only RAM operation needed by RCB is a combination read followed by write operation. This is used to read existing pixel data (black or white) from one word of RAM. The stored pixel operation is applied to this data and the result written back to RAM, immediately in the later part of the operation.

Although the example code provides an FSM it does not fully implement timing. Din & Address must be delayed by 1/2 a clock cycle to make sure there are no setup or hold timing violations (see timing diagram in example 3 handout). This is important. In VHDL when signals are notionally simultaneous it is difficult to be sure which will happen first. Post-synthesis can differ from pre-synthesis. And of course the real hardware could be different again. So the solution is to explicitly allow a time difference whenever the order in which signals change affects operation.

The only target-independent and safe way to make a time difference is using the system clock, and delaying by 1/2 a cycle is easy (using a negative edge triggered register).

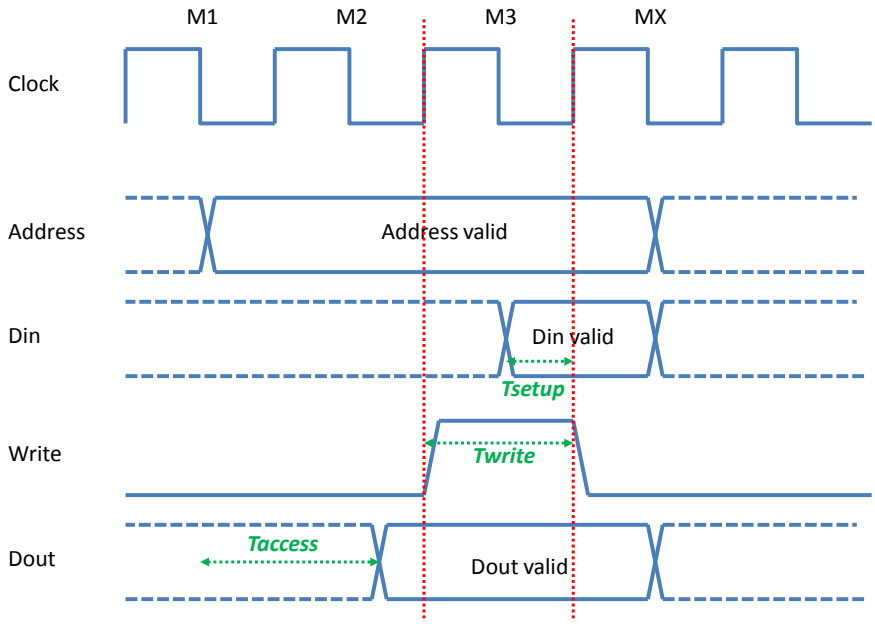

Figure 1

A memory operation always takes 3 cycles, and both reads and then writes one word, determined by the address lines. The required timing is shown above, where M1,M2,M3 are the three cycles of a memory operation, and MX marks one or more idle cycles, see Figure 2. A new memory operation is initiated by START high in states M3 or MX. The data out from the memory can be input on either of the two clock edges shown with dotted lines. Note that the half cycle delay of Din, Address means that signals always stay stable for at least 1/2 a cycle longer than the time at

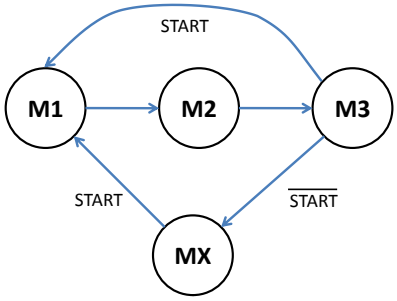

Figure 2

which they are read. This can be implemented by running these signals through an extra negative edge triggered register. Note that this must be a separate process.

This timing means that although the memory itself has a static (asynchronous) interface there will be no problems with gate delays causing timing errors. It also means that post-synthesis simulation for verification will work, since with a low clock rate, the operation will be insensitive to any timing delays in the post-synthesis VHDL output.

Note that data from the memory location selected by address is valid on Dout before it is required on Din. This allows "read-modify-write" (RMW) operation which will be useful.

To use this FSM in a memory unit we need to include data & address input & output, with internal registers to store data and address input so that once a memory RMW operation is initiated data can be changed. The block will use the store output data from **pixel_word_cache** to determine operations on each of the 16 RAM data bits within one word as shown below.

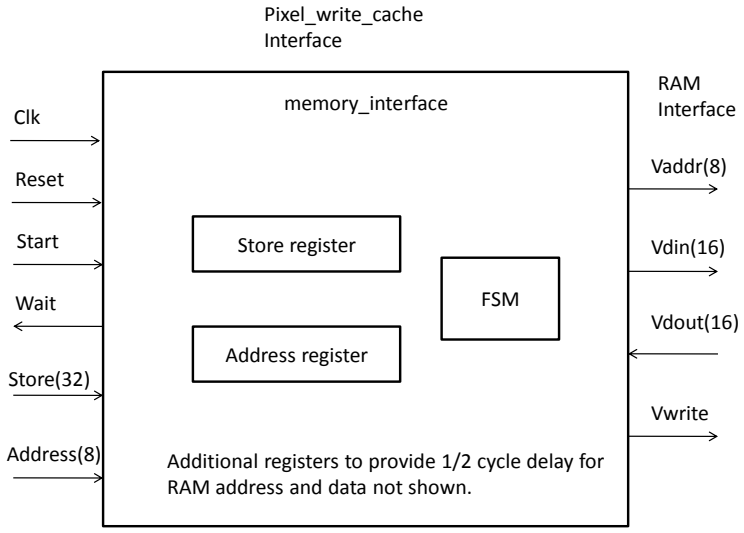

Figure 3

Pixop(i)	Din(i)
same	**Dout(i)**
white	**0**
black	**1**
invert	**not Dout(i)**

The START input is used to initiate a RMW operation and store the pixop data and memory word address. The handshake signal WAIT is defined to be high only if the storing data (store, address) for the next memory RMW is delayed because a previous RMW has not finished. WAIT will stay high until the clock cycle in which store, address are correctly stored as shown in Figure 4. Note that START must stay high until the clock cycle immediately before the corresponding RMW operation starts. Figure 4 shows START may go high to request RMW operation 3 immediately after RMW operation 2 ends. In this case it will be continuously high. When there is no new RMW cycle needed START will remain low and the memory becomes idle.

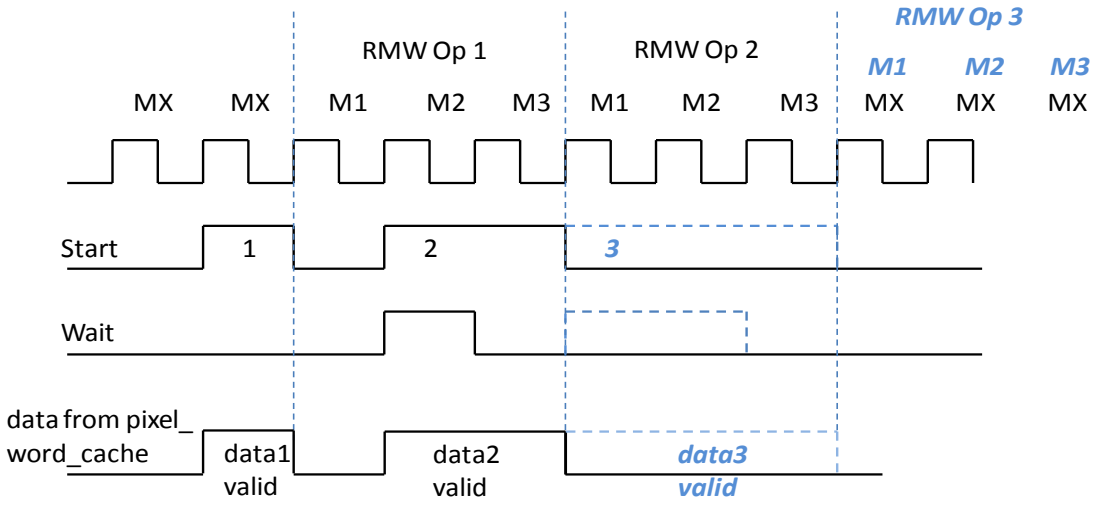

Figure 4

The **data valid** signal shown here is identical to START and indicates when address and pixop data presented to the memory interface should be stored in a register. Thus the hardware driving the memory interface need not preserve this information after data valid goes low, and this signal could be used to enable a register in the memory interface unit to store the data.

APPENDIX C ADVANCED DESIGN TECHNIQUES

This project is open-ended and many different ways to speed up operation are possible, also the RCB can be made configurable and capable of working with different speeds of RAM. Each of these requires additional design and testing, some are not compatible with each other, and you should note the warning given in the project strategy subsection before reading any of the material below. Just because an optimization speeds up one block of course it does not mean that it speeds up the entire design. Careful analysis of hardware bottlenecks will gain you credit in your report, and can be used to improve performance with relatively small changes to each block.

MEMORY TIMING

Figure 1 shows three timing parameters which determine correct operation of the external RAM, explained in the table below. You should normally assume that higher clock frequency will be matched by faster RAM, so all these parameters are scaled by clock_period. Config_pack gives some nominal values however it is expected that the RCB will allow larger values of Tracc and Twp.

Parameter	Value	Description
Tracc (Taccess)	1*clock-period	Delay time between address stable and Dout valid.
Twp (Twrite)	0.5*clock_period	Minimum length of write high pulse. for correct operation
Tds (Tsetup)	0.2*clock_period	Minimum Din valid time before end of write pulse

		(write 1->0 transition) for correct operation.

The VDP testbench enforces these timings and will display error messages if they are not met.

TIMING OPTIMIZATION

The proposed design in its simplest form can vary greatly in performance according to how many pixels per clock cycle it can write, with the limit being 1. Implementing buffering between the host processor and VDP, or the VDP an RCB, or simply optimising state machines, can speed up this basic operation.

You should pay attention to timing optimization: it is the easiest way to get better performance.

CLEARSCREEN OPTIMISATION

If clearscreen commands are passed in some compact form to the RCB they can be implemented more efficiently by writing whole memory words without gaps, at a speed faster than one pixel per cycle. The standard interface specifies optional commands which can be used for this, but be warned, the extra hardware design effort is considerable.

If clearscreen commands are implemented in the DB and passed to the RCB as pixel change operations in the same way as normal line draws there is still room for optimization by ordering pixels so that pixel are output in 4X4 block order so maximizing the benefit of pix_word_cache. This is not easy to do, but possible.

PARALLEL INTERFACE

The standard interface between DB and RCB can be turned into an array allowing more than one pixel per cycle to be output. Good hardware (on either side) can adapt to use different numbers of interfaces optimally. This is in fact easier than it looks, but should not be attempted till you have a basic design fully working.

CLOCK DOMAIN STRATEGY

The expected implementation will use a single clock for all hardware (that is a single signal, passing through ports, never passing through assignment statements). Certainly you should ensure you have fully working single clock deliverables before attempting anything else. However different speeds for the two blocks can with care be retrofitted. If you plan to do this the notes below will help.

It is possible to optimise performance by operating the two sub-blocks at different speeds.

This MUST NOT be implemented using multiple clocks (gating a master clock to make it conditional), since post-synthesis verification becomes very difficult without specialised interface circuitry between the different clock domains.

This CAN be implemented with a single clock by synchronously disabling all register changes in n-1 cycles out of n and thus effectively running part of the hardware at 1/n the speed of the master clock. The headline maximum speed results from Synplify will not take this into account, but if it is done throughout a block, and all the block's outputs are registered, it will be correct to allow nT delay, where T is the clock period, for all critical paths within the block. Synplify 10, the latest version, may have clock constraints that can be used guide smart synthesis in this case.

FASTER RCB

Multiple pix_word_cache units can be used to speed up operation. Even two such units can allow external ram cycles to be overlapped with writing pixels to a pix_word_cache. Many such units can possibly give greater speedup, but note that cacheing the entire external RAM is not allowed because it changes the nature of the hardware.

FASTER DB

There are a number of ways to speed up drawing lines. No guarantees any of these will work, but worth considering:

- Use parallel line draw units and draw lines in parallel where coordinates and/or colour mean that it does not matter which happens first. If lines do not intersect (or, easier to calculate, line bounding rectangles do not intersect) lines can be drawn in parallel. If lines have same colour, or are both invert, the same is true even if they do intersect.
- Split long lines into a number of shorter lines. A point in the middle of a line can be calculated in advance by using multiplication and division to determine the correct pixel position. The division remainder can be used to determine the error value for this middle point. Using this point as starting point for another line, but with non-zero error, the second half of the line can be drawn independently of the first half. Note that the first half of the line must have drawing interrupted when it reached the pre-calculated midpoint.
- Draw two or more pixels in parallel by calculating all possible error values in parallel. The same technique, modified, can be used to speed up error calculation if this is in a critical path by claulcting possible errors in the previous cycle an selecting them with a multiplexor.