

UNIVERSITY OF CALIFORNIA AT BERKELEY
COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Checkpoint 2

Local Video System

1.0 Motivation

This checkpoint serves three purposes:

1. To acquaint you with the arbitration of SDRAM read/write requests
2. To give you experience piecing together sub-modules to accomplish a larger goal.
3. To get local video working, which is a major component of your final project.

This will be the first project checkpoint that requires a significant amount of design to be done by you – this checkpoint is, for the most part, very open-ended. Just make it work!

-Good Luck!!

“BECAUSE YOU WILL BE KEEPING AND RELYING ON THIS CODE FOR MONTHS, IT WILL ACTUALLY SAVE YOU MANY STRESSFUL HOURS TO ENSURE IT WORKS WELL NOW, RATHER THAN WHEN YOU ARE ABOUT TO FINISH THE PROJECT”

2.0 Introduction

In this module you will be building the “glue” that will hold together the entire video system. You have already built the SDRAM controller, now you just need to create a reliable interface such that several data-hungry components can talk to it without conflicts.

Your goals for this checkpoint will be:

1. To design and implement an arbitration scheme for the SDRAM control.
 - a. You will need to be able to handle simultaneous read requests and write requests.
2. To implement local video
 - a. To capture data from the provided video decoder and feed it reliably into the video encoder to display local video
 - b. You will need to modify your address counter to correctly access the data needed by the video encoder and to write the data correctly for your video decoder.
3. To establish a reliable interface between components with different data rates

- a. Using FIFOs as buffers, create an interface to transfer data between the SDRAM arbiter and various connected components.
- b. You must ensure that the FIFO does not overflow or underflow

2.1 Video Format

The video format for the video encoder is the ITU 656 and the format of the video decoder is the ITU 601. As far as this checkpoint is concerned, the only difference between the two is the number of active video lines (more on this later). Both formats are based on the luminance and chrominance used in television instead of the standard RGB format found in computer monitors.

The video data is in 32-bit words, each word representing a pair of pixels side-by-side on the screen. There are two 8-bit luminance (or “brightness”) values per pixel pair: one for the left pixel and one for the right pixel. There is also an 8-bit blue chrominance value and an 8-bit red chrominance value containing the color information of the pixel pair, making a total of 32 bits per pixel pair. This results in the use of **the same chrominance for each pixel pair but different luminance values for each individual pixel**. The reason for this is that your eyes are more sensitive to brightness (luma) rather than color (chroma).

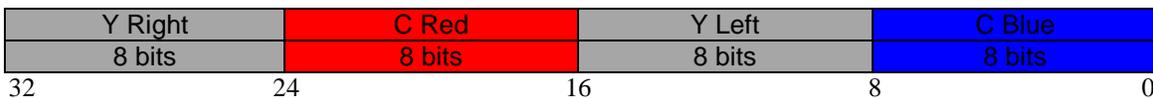


Figure 1: Pixel Pair Representation (Y is luminance, left and right is the pixel in the pair)

These pixel pairs are arranged in a row/column format like a square matrix that make up the screen, with each row corresponding to a row of active video.

The provided video encoder takes in this exact format via its 32-bit *DIn* input. If you wish to read more about the video format, read the specification about the video encoder from Spring 2007.

2.2 Local Video in SDRAM

Of course the point of this checkpoint is to store video in SDRAM. This section details the format of the video data in memory.

Since the video data is organized into 32bit words, this motivated the design of a 32-bit wide SDRAM interface in Checkpoint 1. The organization of pixels into words and words into bursts is shown in figure 2 below.



Figure 2: Video Memory Organization

This simply shows the organization of the pixels within a burst. In addition, we must decide on a format for how to organize the bursts. The scheme we will use is the following:

```
RowAddress = {pad with 0's, PixelRow}
ColumnAddress = {BurstColumn, 3'h0}
BankAddress = 2'b00
```

This addressing scheme should be pretty straightforward; **the row address is the row of active video you are accessing and the column address is what pixel pair to access within that row**. If this sounds confusing, just remember that all you are doing is copying the contents of a square matrix ([pixel row][pixelpair column]) into a bigger square matrix ([row address][column address]).

Since the ITU 601 standard has only 487 lines of active video and the ITU 656 standard has only 487 has 507 lines of active video, you must **pad the top of the row address with 4*1'b0s**, to make it total a 13bit for a row address.

The **bank address is fixed at 2'b00** for local video.

The **column address is based on the video pixel pair column**. Notice that in this case the `BurstColumn` indicates which BURST you wish to access. The `3'h0` on the right ensures that you will always **read from a column address that is a multiple of 8**, since you read out 8 pixel pairs per read/write request.

By **dividing the SDRAM address into row and column along the same boundaries as the video row and column**, we have **simplified debugging**, so that you can easily see which pixels are being read and written at any time.

2.3 Asynchronicity

Unlike everything you have done in the past labs and checkpoints, this checkpoint will require 2 different clock signals. The camera operates on its own independent clock. If you are curious about the differences between the 2 clock signals you can analyze them using the oscilloscope. What's important for this checkpoint, and what will be extremely important in later checkpoints, is that you understand the limitations of working with 2 clocks. You cannot have signals generated by state machines that run off of different clocks communicate directly with each other. For this checkpoint, you will cross this boundary using an asynchronous FIFO. This module is a normal FIFO, except that its reads and writes operate on different clocks. You will use this asynchronous FIFO to communicate to the video decoder.

2.4 ITU 654 vs ITU 601

The video encoder uses the ITU 656 Video Standard. However, the camera operates under the ITU 601 Video Standard. The difference between the 2 standards is that the **ITU 656 standard has 487 lines** of active video, while **the ITU 601 standard has 507 lines**. The way to get around this is to not display the extra lines of valid active video written by the video decoder by simply inputting them to the video encoder during the lines of vertical blanking when data input does not matter. **Both standards have 720 active pixels (360 pixel pairs) per line.**

3.0 Prelab

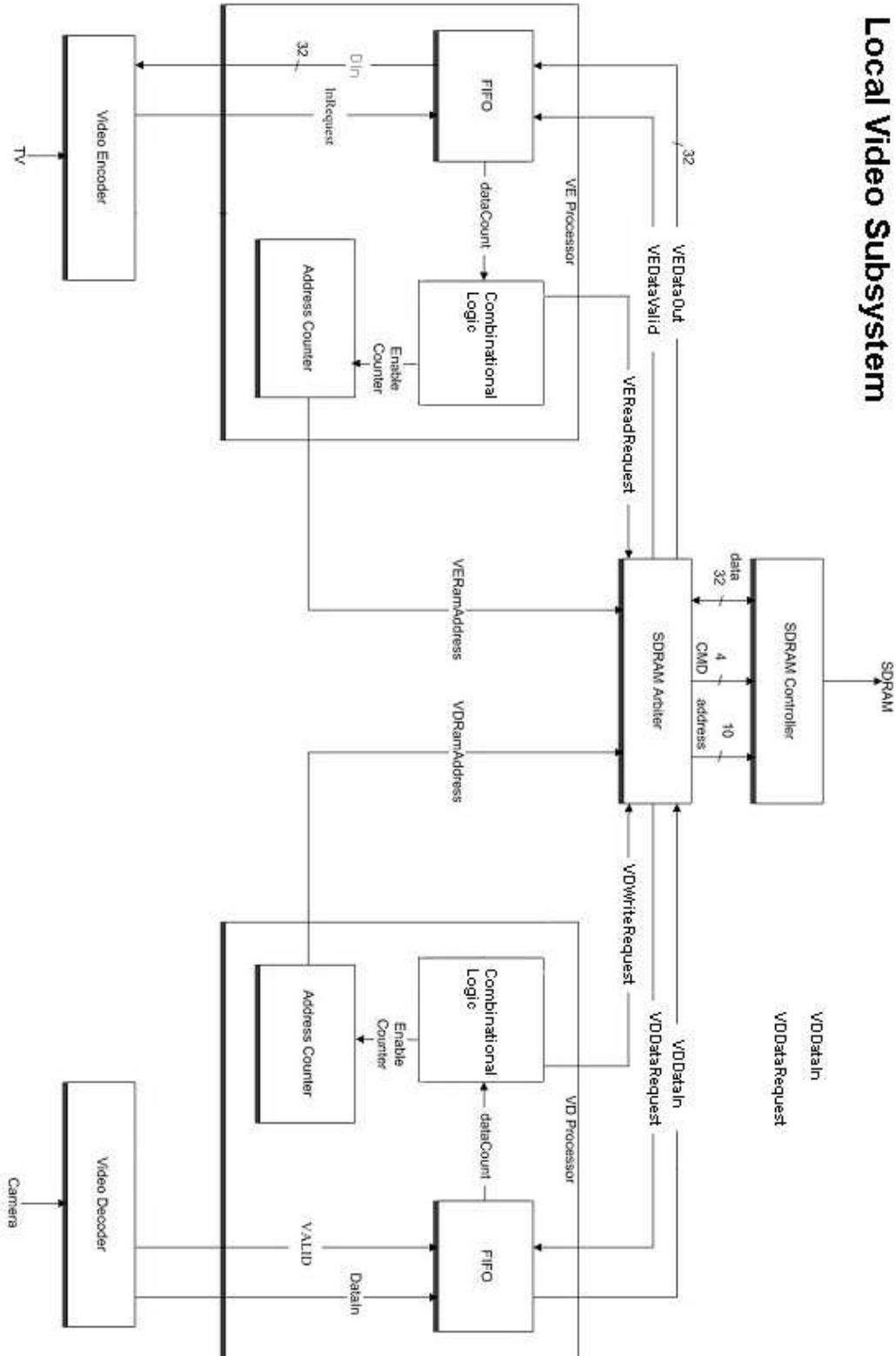
Please make sure to complete the prelab before you attend your lab section. **You will not be able to finish this checkpoint in 3hrs!**

Read this handout thoroughly.

- a. Pay particular attention to section **2 and 4** as it describes background information that you will need to know
2. **Examine the Verilog provided for this checkpoint**
 - a. The code provided is heavily commented, make use of it!
3. Before you begin to design anything or to write any code, think about the project at hand.
4. **Start your design ahead of time.**
 - a. Begin with **schematics** and **bubble-and-arc** diagrams
 - b. Come prepared for your design review

4.0 Lab Procedure

4.1 Local Video Overview



4.2 SDRAMArbiter.v

The arbiter's job is to toggle control of the SDRAM Controller. It will often be the case that the Video Encoder needs data at the same time that the Video Decoder needs to write data. The job of your arbiter is to decide who gets control and when. The implementation is completely up to you and the suggested solution is simply a *suggestion*, feel free to modify it in any way.

Suggested SDRAM Arbiter Operation:

1. While nothing interesting is happening the arbiter should sit in some kind of an idle state.
2. If a module needs a read or a write it asserts a Read/WriteRequest signal.
3. If the arbiter decides to give attention and serve the module's Read/Write request, it must change to a "Serving" state and tell the SDRAM controller to do the read or write. The arbiter then needs to handle the timing, command signal assignment, and passing of data to the SDRAM controller.
 - In the case of a WriteRequest, the arbiter must assert the appropriate "DataRequest" signal to read from that module's FIFO
 - In the case of a ReadRequest, the arbiter must assert the appropriate "DataValid" signal to write into that module's FIFO
 - If both modules requires attention, the arbiter must choose a module to serve and a priority of who to serve must be established
4. When the request is done, the arbiter moves to the idle state and waits for a new request

UPDATE: You will need a negative edge triggered Register at the output of the RAM_DQ line from the SDRAM! (similar to what you had in checkpoint 1)

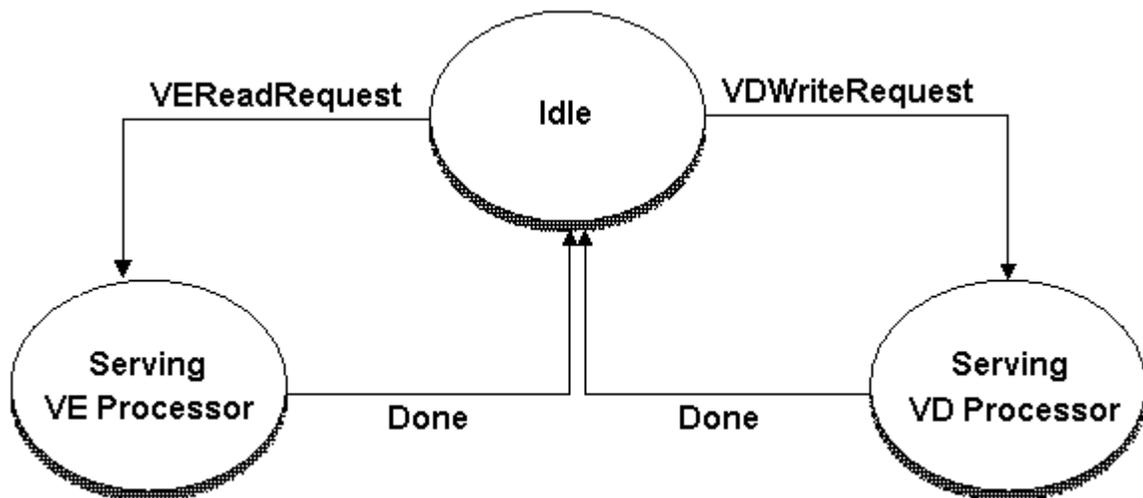


Figure 3: SDRAMArbiter State Machine

A suggested basic port specification is below. However, you should read the included verilog file and are free to deviate from the suggested design.

Signal	Width	Dir	Description
Clock	1	I	Clock
Reset	1	I	Reset
MemReady	1	I	Ready Signal from the SDRAMControl
SDRAM_DQ	32	I/O	SDRAM DQ Line
ColumnAddress	9	O	Just what it sounds like
RowAddress	13	O	Just what it sounds like
BankAddress	2	O	Just what it sounds like
MemDone	1	I	Done signal from the SDRAMControl
RAMReadRequest	1	O	Request a read from the SDRAMControl
RAMWriteRequest	1	O	Request a write from the SDRAMControl
RAMReadValid	1	I	Valid Signal
VEDataOut	32	O	Output to the VE Processor
VEAddress	24	I	Address from the VE Processor
VEReadRequest	1	I	VE Processor requests attention
VEDataValid	1	O	FIFO write Enable for the VE Processor
VDDataIn	32	I	Received data from the VD Processor
VDAddress	24	I	Address from the VD Processor
VDWriteRequest	1	I	VD Processor requests attention
VDDataRequest	1	O	FIFO read Enable for the VD Processor

Table 1: Port Specification for SDRAMArbiter.v

4.3 fifo_sync32d.v

The sync FIFO is a simple, two-interface module, you write into one and read from the other. The way this module assists is to buffer your data and allow the Video Encoder and the SDRAM to run at different rates.

Signal	Width	Dir	Description
Clk	1	I	Clock
Sinit	1	I	Reset, will empty the FIFO
Din	32	I	Data input bus
Wr_en	1	I	Write the value on din into the FIFO on the clock
Rd_en	1	I	Read enable, Dout will be valid after next rising edge
Dout	32	O	Data output bus, valid next cycle after rd_en
Full	1	O	Indicates that the FIFO is currently full
Empty	1	O	Indicates that the FIFO is currently empty
data_count	2	O	Indicates how many words are in the FIFO 2'b00 means 0-¼ full 2'b01 means ¼ - ½ full 2'b10 means ½-¾ full 2'b11 means ¾ -1 full

Table 2: Port Specification for fifo_sync32.v

4.4 fifo_async.v

The operation of the asynchronous FIFO is exactly the same as the one specified above. However, you should notice that the read and write clocks are different. This doesn't change the basic operation of the FIFO. Keep in mind that you need to make sure that your FIFO doesn't overflow or underflow by keeping track of the `data_count`.

NOTE: The `data_count` on the asynchronous is a word count (1 word is 32 bits, 8 words is 1 burst). You should attempt write as soon as you have a full-burst worth of data in the FIFO.

ALSO NOTE: Because you have 2 different clocks, you have 2 different `data_count` signals. Each one is asserted on the positive edge of its respective clock. Make sure you use the correct `data_count` signal.

4.5 I/O Processors

The following 2 modules are up to you to design. The design is completely open-ended – all that we care about it is that they work.

These modules should be very simple. Their only job is to keep track of when a read or a write is needed and toggle the correct signals to make it happen. Don't make them more complicated than they need to be.

UPDATE: The video encoder has been changed to accept the same number of lines as the video decoder (it just won't display them). This should make it so that you can use the same AddressCounter (counting from 0-506) in your VEProcessor and VDProcessor. You no longer have to worry about ANY differences with the formats.

4.5.1 VEProcessor.v

The purposes of this module are:

1. To determine the address of the current burst that needs to be read in order to feed the correct data into the video encoder.
2. To determine when a RAM read is needed. HINT: This is done using the "data_count" signals from the FIFO.
3. To handle read/write signals for the FIFO that will buffer the data between the SDRAM and the video encoder.

We were able to build this module using only the Address Counter, FIFO, and very little combinational logic.

4.5.2 VDProcessor.v

The purposes of this module are:

1. To determine the address of the current burst that needs to be written in order to allow the VE Processor to access the correct data.
2. To determine when a RAM write is needed. HINT: This is done using the “data_count” signals from the FIFO.
3. To handle read/write signals for the FIFO that will buffer the data between the SDRAM and the video decoder.

This module is also a very module built similarly to the VEProcessor. Once again, we were able to build this module with the Address Counter, FIFO, and some combinational logic.

4.6 VideoEncoder.v

This module has been provided for you via the black box. It features a 32-bit input DIn for you to put in valid pixel pair data, a InRequest signal to tell you that a new 32-bit input must be placed at DIn *on the next cycle*, and InRequestLine/InRequestPair signals to tell you what pixel pair it needs. You do not need to worry about the other signals, as they are used by it to connect to the on-board video encoder. Basically to sum up its I/O behavior, **when InRequest is asserted, you must give it a new pixel pair (corresponding to the line and row numbers indicated in InRequestLine and InRequestPair) after the next rising edge.**

UPDATE: The video encoder has been changed to accept the same number of lines as the video decoder (it just wont display them). This should make it so that you can use the same AddressCounter (counting from 0-506) in your VEProcessor and VDProcessor. You no longer have to worry about ANY differences with the formats.

Signal	Width	Dir	Description
VE_*****	1-10	I/O	Signals to the on-board video encoder, you don't need to worry about these
CycleCount	7	O	You don't need to worry about this
DisableSCLop	1	O	You don't need to worry about this
SDataEnable	1	O	You don't need to worry about this
Sdata	1	O	You don't need to worry about this
Clock	1	I	The Clock signal
Reset	1	I	The Reset signal
Din	32	I	Requested pixel pair, YC _R YC _B
InRequest	1	O	Request signal, is high for one out four cycles when requesting active video. DIn needs to be valid after the riding edge where this is 1'b1.
InRequestLine	9	O	The line from which you are requesting video data
InRequestPair	9	O	The pair of pixels which you are requesting Notice that this address should go up by one for each 32-bit pixel pair.

4.6 VideoDecoder.v

This module is provided for you in verilog and is commented extensively. You should be able to determine how it works by examining the code and applying your understanding of your video encoder. This module is basically just a backwards version of checkpoint #1. It, however, does not have InRequestPair and InRequestLine signals, you will need to keep track how where/when to write to the RAM yourself.

Below is an explanation of the relevant signals. We have left all of the wiring intact from our implementation of this checkpoint. You should be able to determine everything you need for a combination of an analysis of the code, the wiring, and the table below.

Signal	Width	Dir	Description
EN	1	I	This is the enable signal, we use the memready signal for the SDRAM Control for this purpose.
SEF	1	O	High at the start of an even field
SOF	1	O	High at the start of an odd field
EAV	1	O	High at the end of active video
SAV	1	O	High at the start of active video
VALID	1	O	High When the data on the DATA_OUT line is valid
DATA_OUT	32	O	The pixel pair currently being output.

Hints

- The decoder will output video signals in the opposite direction as the Video Encoder did. i.e. the signals will be $C_B Y C_R Y$.
- **The SDRAM takes a while to initialize (100us) so you should keep the Video Encoder and the Arbiter's reset signal high during this time (should already be done for you in the provided code)**
- The main modules you must write for this checkpoint are the arbiter and the two video processors.
- THINK about the problem before jumping into designing or coding.
- Remember that the data_count on the asynchronous FIFO is in units of words and there are separate signals that are valid for each clock.