ECE352F: Computer Organization

Lab #4: Bus Design

Fall 1997

In this lab you will try to implement a simple bus protocol between the Gizmo board PIT ports and the FPGA. You will control signals on the PIT using a program on the Gizmo board and implement an interface to the bus in the FPGA.

This circuit will be used in future designs to allow better testing of the circuits that you will be building.

1 Bus Description

A general picture of the connections is shown in Figure 1. The bus should be capable of sending 8-bit addresses and carrying 16 bits of data. There are only eight wires (**AD**) that will be used to multiplex the address and data in three transactions on the bus. Note that information is carried in both directions on **AD** so you will have to be careful about the timing of the signals. This will be handled by the bus protocol.

The bus protocol will be asynchronous, meaning that there will be handshaking signals instead of a clock. All transactions are initiated from the PIT side. You should read Section 4.5 in the text, particularly the part about asynchronous buses. In your system the skew will be very small, and is likely not even a problem. However, you can easily account for skew on the bus by ordering the the way you change the data and control registers on the PIT side.

1.1 Read Cycle

Figure 2 shows a timing diagram for the read cycle. The handshaking signals are **RDY** on the PIT side and **ACK** on the FPGA side. In the timing diagram, a level in the middle, as shown for the **AD** lines indicates that nothing is driving the lines so the lines are floating.

To start a read, the PIT carries out the following actions:

- Set the read/write line (**R/W**) high to indicate a read cycle
- Drive the address on the **AD** lines
- Set **RDY** to indicate that the **AD** and **R/W** lines are stable.

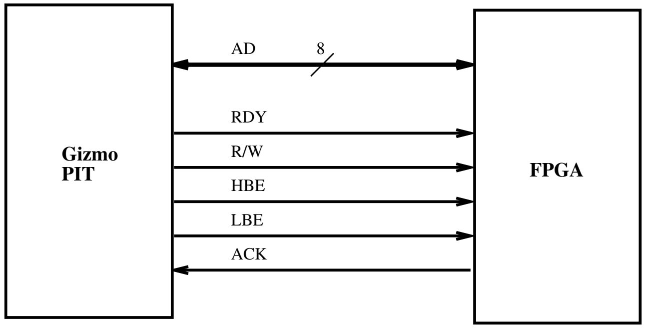

Figure 1: Overview of Connections

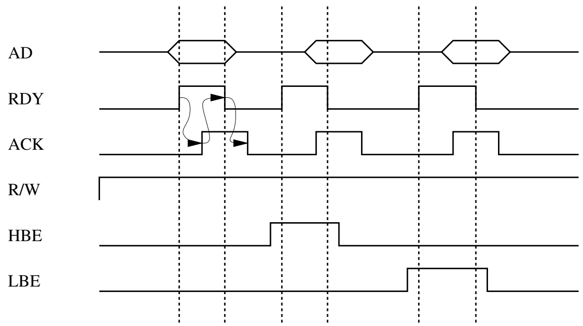

Figure 2: Read Cycle

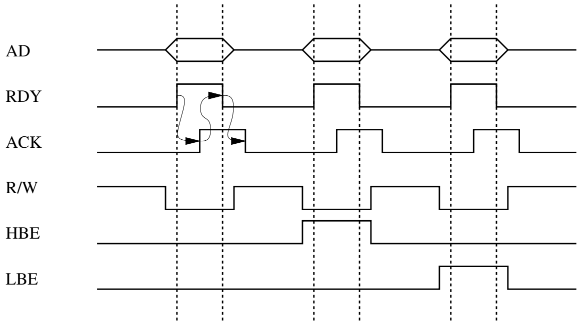

Figure 3: Write Cycle

The FPGA will handshake with the PIT using the **ACK** line. When **ACK** goes high, it means that the FPGA has latched the address into its address register. You can also implement a timeout feature to detect illegal addresses. If **ACK** does not get set high after a certain amount of time, then you can assume that there is nothing at that address, and abort the transaction.

The next transaction on the bus is to read the high byte of the data.

- The PIT sets the high-byte enable line (**HBE**), and then sets **RDY**.
- The PIT waits for **ACK** to indicate that data is now on the bus.
- When the data has been read from the bus, **RDY** is dropped.
- **ACK** is then lowered.

A similar transaction is carried out for the lower data byte, except that **LBE** is used. If only 16-bit transfers were being used, then you do not need both **HBE** and **LBE**. With both lines, you can build your bus protocol so that it can transfer only one byte instead of two bytes in each cycle. Note that if neither byte enable signal is on when **RDY** is high, then an address is being transferred. You can use this to indicate that a new bus transaction is being started.

1.2 Write Cycle

The write cycle is shown in Figure 3. During a write cycle, the **ACK** signal is used during all three bus transactions to indicate that the FPGA has captured the information on the **AD** lines. Here, the **R/W** line is set low at the same time as the byte strobe lines are set to indicate that a write is occurring.

2 FPGA Implementation

The FPGA circuit should be implemented with VHDL. It is easiest to implement the logic as a finite state machine, because that is what this controller is. Do not try to use the asynchronous logic style used in the text. You can use a synchronous circuit. These days with the possibility of really high-speed clocks, you can generally get fast enough response time this way. In the case where you really have to respond quickly to some input, then you would think about doing that part asynchronously. There are examples in the *VHDL Reference Guide* on how to do FSMs.

You should also have a separate input for a **Reset** signal to make sure that your circuit starts in a reasonable state. The worst case is that the FPGA starts up driving the **AD** lines and you do not realize that from the PIT side. You could end up having the two devices driving the wires at the same time.

You can use many of the pins on the FPGA. Any of the pins connected to the 8031 are available. You can disable the 8031 by setting XCBUS36 high (RST). This can be done in your VHDL by assigning a '1' to that pin. We will be using the RAM later, so you might as well stay away from it for now.

You should have three 16-bit registers that are byte-addressable using an addressing model like the 68000. This means that all words are at even addresses, and an odd address refers to the lower byte of a word.

Note that for a port that is bidirectional, you should declare it as *inout* in your entity declaration. *Whenever you are finished outputting data on the* **AD** *lines, you should make sure that you drive a high impedance (AD ⟸ "ZZZZZZZZ") on those lines to stop the output operation, and stop driving the lines so that they can be used by another device.*

2.1 Debugging and Testing Your VHDL

It is good to use a VHDL analyzer and simulator to check your VHDL program for both syntax and functionality as part of your preparation.

I am making the arrangements to get the PC-based simulator.

There is another option for you if you want to work on the *ugsparc* machines. When something appears in the directory */CMC* on the *ugsparcs*, then you are ready to go. We are currently getting that directory enabled.

A short tutorial will be available on top of the file cabinet outside my office (LP484A) by noon Friday. The tutorial should be sufficient for you to do what you need to do. I encourage you to develop simulation scripts so that you do not have to keep typing in the same commands. To get more information on the simulator, you can run the online documentation system called *iview*. To get access:

```
ugsparc1% source /CMC/tools/synopsys.1997.01/source_vrg.csh
```

This will add the correct directories to your search path to access all of the Synopsys commands. You might want to add this to your *.cshrc* file.

Start up *iview* and go to **File** ⟹ **Open Collection**. Select **Synopsys Design Source** and click **OK**. You want to start with the **Design Source and VSS Tutorial**.

3 The C Program

On the PIT side, you should write a C program for the Gizmo. Write two procedures, one for read and one for write. By modifying the I/O parts of your program, you should be able to debug the logic of the program on the *ugsparcs*.

The two procedures are described below:

```
int FPGAread(unsigned char Address, char *HighByte, char *Lowbyte, int Byte)

  Address:  8-bit address
  HighByte: pointer for returning the high byte of data
  LowByte:  pointer for returning the low byte of data
  Byte:     set to 1 for a byte read, 0 for a word read

  Return value: 0 for success, 1 for error


int FPGAwrite(unsigned char Address, char HighByte, char Lowbyte, int Byte)

  Address:  8-bit address
  HighByte: high byte of data to be written
  LowByte:  low byte of data to be written
  Byte:     set to 1 for a byte write, 0 for a word write

  Return value: 0 for success, 1 for error
```

NOTE: Some people had difficulty in the first lab with using the PIT. I believe it is because the registers were not configured correctly in the PIT. Look at the example on page 83 in the Gizmo manual. You should use one PIT port for the **AD** bus and another port to implement the control lines. Remember that you will have to keep configuring the data direction register for the **AD** port depending on what you are doing.

4 Testing and Debugging

Some care must be taken when debugging because you can easily have both the PIT and the FPGA driving the **AD** wires at the same time. In particular, check that both your FPGA logic and your PIT port that connects to the **AD** lines are set up to be reading or not driving the **AD** lines after they finish writing. This should turn off the drivers. Go back and read your code now! Having two devices drive the wires could damage the chips.

To minimize the possibility of problems, you should try to test the two parts individually. On the Gizmo side, you can have your program stop for input at each stage. This will allow you to probe lines and set values on input lines and test your program that way. You can do the same thing with the FPGA. When you are convinced both sides are working, then you can connect them together.

You should be able to demonstrate a program that acts like a simple monitor allowing you to examine and deposit data into the registers in the FPGA.

5 Preparation

An ideal preparation will have a C program written, compiled and ready to be loaded. The logic should have been debugged by running a version of the program on the *ugsparcs*.

For the FPGA, you should have fully simulated VHDL code. For now, you will have to use the *ugsparc* simulator.