Lab #5: Bus Design

ECE352F: Computer Organization

Fall 1999

In this lab you will try to implement a bus to communicate data between the Gizmo board PIT port and the FPGA. The PIT side will be the *master*, meaning that all bus transactions will be initiated from the PIT. The protocol will be implemented in software on the gizmo board, and in hardware in the FPGA using VHDL. This will give you an opportunity to contrast how finite state machines can be implemented using software written in a programming language like C and hardware described with VHDL.

A working circuit will be useful for future designs as an interface to other circuits that you will build. You will want to make it work!

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

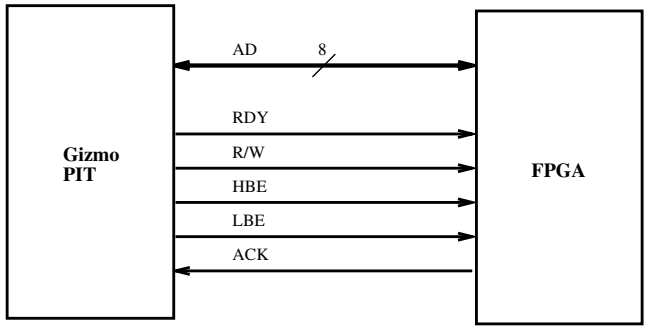

Figure 1: Overview of Connections

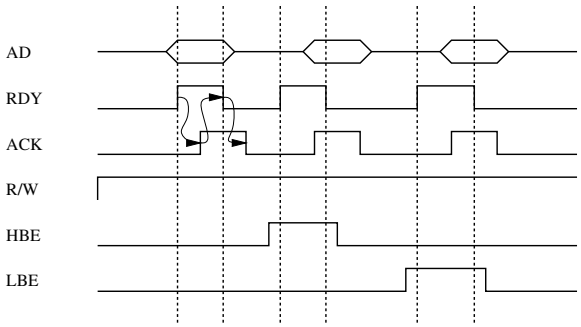

Figure 2: Read Cycle

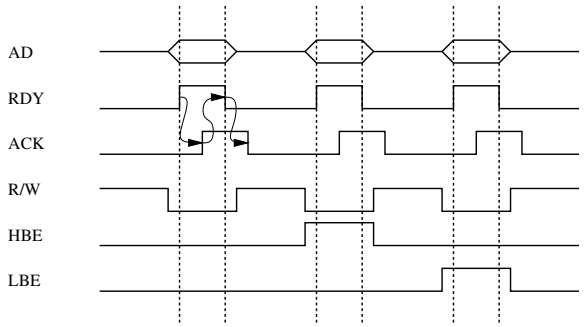

Figure 3: Write Cycle

- Set **RDY** to indicate that the **AD** and **R/W** lines are stable.

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. With these signals, it is also possible to do single byte transfers. 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.

Preparation: Draw the timing diagram showing a read of only one byte followed by the read of the high followed by the low byte of a 16-bit word.

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.

Signal	PIT	FPGA	7-Seg
AD0	Port A 0	XCBUS07	
AD1	Port A 1	XCBUS08	
AD2	Port A 2	XCBUS09	
AD3	Port A 3	XCBUS06	
AD4	Port A 4	XCBUS77	
AD5	Port A 5	XCBUS70	
AD6	Port A 6	XCBUS66	
AD7	Port A 7	XCBUS67	
$\overline{\text{Reset}}$	Port B 0	XCBUS68	
	Port B 1	XCBUS19	Top
	Port B 2	XCBUS23	UR
R/W	Port B 3	XCBUS18	UL
HBE	Port B 4	XCBUS20	Mid
LBE	Port B 5	XCBUS26	LR
ACK	Port B 6	XCBUS24	LL
Ready	Port B 7	XCBUS25	Bot

Table 1: Pin Connections

Preparation: Draw the timing diagram showing a write of only one byte followed by the write of the high followed by the low byte of a 16-bit word.

2 FPGA Implementation

The FPGA circuit should be implemented with VHDL. It is easiest to implement the logic as a finite state machine, because a controller is really an FSM. Do not try to use the asynchronous logic style used in the text. It is better to use a synchronous FSM design. 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.

You should also have a separate input for a $\overline{\textbf{Reset}}$ 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.

As a means of providing some standardization and inter-operability between designs, you should use the connections as shown in Table 1.

You can disable the 8031 by setting XCBUS36 high (RST). This can be done in your VHDL by assigning a '1' to that pin.

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

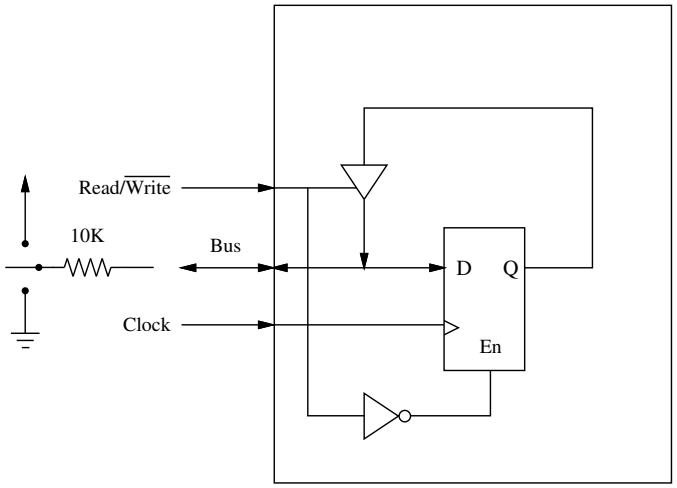

Figure 4: Simple Test of In/out Bus

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 A Simpler Step

In this particular design it is especially important to think about how the signals connect to the various parts of your system. Remember to draw a block diagram.

To get familiar with the bidirectional ports, you should start with a much simpler circuit that you can build and control with just the switches. Consider the circuit in Figure 4. This is just a one-bit memory with a single data port labeled **Bus**. In this picture, a **Read** means that the data in the register bit should be driven on the **Bus**. A $\overline{\textbf{Write}}$ (a low on that line) means that the data on **Bus** should be stored in the register. If you want to connect a switch to the **Bus** input, do it through a resistor. Why is this important?

2.2 Debugging and Testing Your VHDL

There are various levels of simulation when doing designs in VHDL. The first level is strictly *behavioural* simulation. In this case you are simply simulating the VHDL source code and debugging it much like you would debug a C program. Then, depending on the technology that you are synthesising to, there can be various levels of netlists that you can get after synthesis. For example, after synthesis to a gate-level netlist, you may use behavioural

simulation of the gates to see whether it still has the desired function. This netlist could be further annotated with timing information for more accurate timing simulation.

NOTE: Remember that your hardware will be running much faster than any human can react. A common error during simulation is to change input signals on every clock cycle when in the real world many, many cycles would be run before an input changes. In these parts of your circuit, it is good to run at least three or four cycles to make sure that your state machines do wait for input changes properly.

Preparation:

1. Write and simulate a VHDL program corresponding to the circuit in Figure 4.
2. Draw a block diagram showing all of the components in your circuit, especially all of the buses. Show which buses are bidirectional. If you write your VHDL according to this diagram, you will encounter much fewer difficulties. Handling the bidirectional pins can be particularly tricky.
3. Write and simulate the VHDL to implement the protocol required on the FPGA.
4. You should have some timing diagrams to show that your circuit simulates correctly.

If you end up with more than 3–4 pages of VHDL then you are making this much more complicated than necessary.

3 The C Program

For the Gizmo, you should write C routines to handle the control of the PIT. 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: The configuration of the PIT will be similar to what was used in Lab 1. Remember that you will have to keep configuring the data direction register for the **AD** port depending on what you are doing.

Preparation: Write and debug, as much as possible, the *FPGAread* and *FPGAwrite* procedures.

Using those procedures, write a simple monitor that allows you to examine and deposit data into the registers in the FPGA.

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, after each bus transaction, check that your FPGA logic is driving high impedance and your PIT port connecting to the **AD** lines is set up to be reading or not driving the **AD** lines. 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.

5 In the Lab

1. Test out your simple circuit to show that you can operate a bidirectional bus.
2. Build the main circuit. Build your circuit neatly. If you get it to work, then you can use it for future labs.
3. Demonstrate your system.