Computer Structure Lab - Fall 1998
Dr. Guy Even

1 Introduction to the RESA Computer (version 1)

The RESA Computer was designed for teaching how to design a CPU. Students can use the RESA as a concrete platform for designing a CPU and testing their design by running programs on it. The CPU of the RESA is based on a chip called an FPGA (Field Programmable Gate Array) that can be configured or "programmed" with regular signals without removing it from a board or disconnecting it.

We start by describing the architecture and the buses.

1.1 Architecture

Figure 1 depicts the parts of the RESA Computer. The RESA Computer consists of:

CPU board. The CPU board contains an FPGA which is to be programmed to function as the CPU of the RESA.

Memory board. The memory board contains DRAM (dynamic RAM) chips which serve as the main memory of the computer. A non-volatile memory which is based on EPROM chips (erasable programmable ROM) is used for storing a fixed program (the test program) which can be executed by the CPU after reset or power-up.

Backplane bus. All the cards are connected to the backplane bus. Every card has logic for controlling the communication via the backplane bus. The backplane bus can support upto 4 masters and 4 slaves, which means that upto 4 memory boards and 4 CPU boards can communicate via the the backplane bus.

Backplane bus controller. The backplane bus controller coordinates the communication over the bus. Due to engineering considerations, the connection between the RESA and the PC computers is implemented by a connection from an ISA special purpose card placed in one of the PC slots to the bus controller card.

1.2 Buses

In this section we discuss the backplane bus. Since buses are new to you, we need to start with a general discussion.

1.2.1 What is a bus?

Suppose we have multiple devices that want to communicate with each other. The simplest way to enable communication is to connect them to the same wires so that each device can read and write signals to the wires. A wire which connects devices is called a *bus*. Often, several wires are connected in parallel to the same devices to enable the transmission of more than a single bit simultaneously.

Connecting the devices together by wires does not solve the problem of communication. There are a few issues which need to be addressed, such as: How do the devices refer to

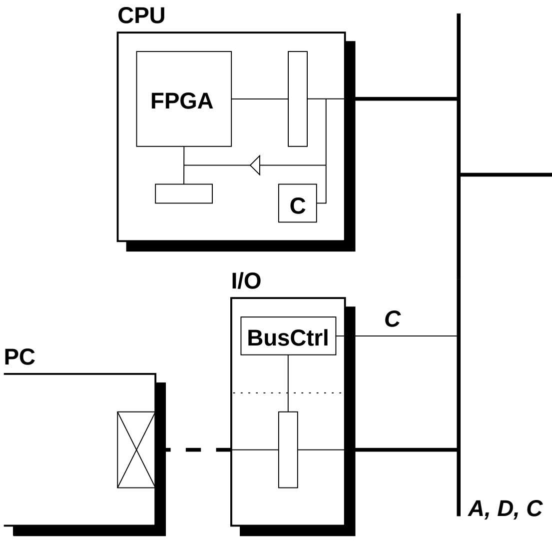

Figure 1: The RESA Cards and buses

other (i.e. naming)? How do they actually communicate (i.e. language)? How do the devices share the common resource (i.e. controlling contention)? How is the common resource shared in a fair way (i.e. fairness)? How do we make sure that a device does not hold the common resource indefinitely (i.e. deadlock prevention)?

We consider a setting in which communication takes place in chunks called *transactions*, which means that transactions are the basic "unit of communication". A transaction is initiated by one party (i.e. a device) which either wants to send data to another party or wishes to retrieve data from another party. Note that the data transmitted during a transaction can be a bit, byte, or even several bytes. The party which initiates the transaction is called the *master* and the party which is asked to respond is called the *slave*. We differentiate between two types of transactions: (a) a *write transaction* is a transaction in which the master wants to send a value to the slave (for example, in an execution of a store instruction, the CPU writes a value to the main memory); and (b) a *read transaction* is a transaction in which the master wants to receive a value from the slave (for example, in an execution of a load instruction, the CPU reads a value from the main memory).

The communication over the bus is determined by a *protocol*. A protocol is an algorithm, however, it differs from the algorithms you have seen so far in a few ways: (a) The protocol is executed by multiple parties as opposed to the algorithms you have seen which are executed by a single central "entity". (b) A protocol does not perform a "pure computation" like an algorithm that computes, for example, primes numbers. The purpose of a protocol is to coordinate the usage of the common resource, namely, the bus. One can view the bus protocol as the "rules of the road".

The transactions which we consider are *address based*. This means that the bit signals sent along the bus are divided into fields, which include: an address field, a data field, and some additional control signals. The master in a transaction transmits the address of the data item to be transmitted. In a write transaction, the master also writes the data in the data field. In a read transaction, the slave writes the data in the data field.

When a transaction begins, the master device does not know how long it will take the slave device to complete the transaction. There are two reasons for this uncertainty: (a) the delay of the slave device might not be constant; and (b) this helps simplify the design. The master doesn't need to know the delays of all the slave devices, and the master's control is not based on the delays of the slave devices. Instead, the slave device reports the completion of the transaction to the master device. The number of cycles it takes the slave device to respond is referred to as the number of *wait states*. A slave device with k wait states responds $k+1$ clock cycles after the master requests the transfer of data.

1.2.2 The backplane bus protocol

Before we describe the RESA bus signals and the protocol, we mention two properties of the RESA bus protocol:

1. The backplane bus protocol of the RESA computer is a *synchronous* protocol. This means that there is a global clock signal which is present in all the devices that are connected to the bus. Every transition of a signal in the bus is synchronized with an edge of the clock. There are transitions that are synchronized with the falling edges, and there are transitions that are synchronized with the rising edges.
2. In each transaction, one word of data is transmitted. Other protocols exist that can support bursts of data to save the setup time spent in establishing a transaction.

Since the bus protocol is an algorithm, we start by describing the data structure, which in this case are called signals. The following signals are transmitted along the bus:

Clock (`clk`)**:** This is a single bit signal which serves as the common clock. The bus controller is the only device that may write this value. The other devices only read it. The clock signal is usually a periodic signal which has a pulse width equal to half the period.

Bus Request ($\texttt{br}_i$)**:** Master device i (there are upto 4 master devices) is connected to the bus controller with a bit signal called the bus request. Whenever master device i wishes to initiate a transaction, it asserts the $\texttt{br}_i$ signal to signal to the bus controller that it would like to start a transaction. Only master device i may write to the $\texttt{br}_i$ signal. This signal can be viewed as a private link between master device i and the bus controller.

Bus Grant ($\texttt{bg}_i$)**:** A second private link between master device i and the bus controller is the $\texttt{bg}_i$ signal. The $\texttt{bg}_i$ signal is asserted by the bus controller to signal that master device i may start a transaction, that is, the bus is free and master device i may use it. This is like a green light in a traffic light.

Address Strobe (`as`)**:** The beginning of a transaction is signaled by a master device by asserting the `as` signal. This tells the other (slave) devices that in the next clock cycle, information will be sent on the bus by the master that they should be listening to.

Address ($\texttt{A}[31:0]$)**:** The address signals are used to transmit the address of the data item which is transmitted during the transaction. Note that the address of a data item holds information about the device which holds it as well as information about where it is held in the device. For this purpose, some of the address bits are used to address the slave device, the the rest are used to address the data item within the device. The address signals are always asserted by the master.

Data (D[31 : 0])**:** The data signals are used to transmit the data item which is actually being transmitted in a transaction. Recall that this is the purpose of the whole transaction!

Write (wr)**:** The wr signal is asserted by the master to indicate whether the transaction is a read transaction or a write transaction.

Acknowledge (ack)**:** The slave acknowledges the transmission of the data item by asserting the ack signal. In a write transaction, the slave signals that the data has been read and processed by it. In a read transaction, the slave signals that the requested data item will be transmitted in the next clock cycle (Why not wait until the requested data is transmitted? Why does the slave warn the master that the data will be transmitted?).

Figure 2 depicts the *timing diagrams* of read and write transactions. We make the following remarks about the notation used in the timing diagrams:

1. Some of the bus signals are negated for reasons that are explained in Section 1.2.3. The negation of signal x is denoted by /x.

2. The time it takes for a transition of a signal from one logical level to another is denoted by a box, the vertical sides of which determine the earliest and latest possible times at which it is stable. For example, the as signal is asserted and disabled after a rising edge of the clock. This notation is used for transitions of signals that can only hold logical levels (i.e. low or high).

3. The assertion of a signal the value of which is not uniquely determined before it is asserted is denoted a diagonal transition. This notation is used for signals that can hold logical levels as well as non-logical levels (which at the logical level are not defined).

4. Signals that can be either low or high during a transaction are depicted using two parallel horizontal lines, one for the low value and the other for the high value. For example, the address and data signals can be either high or low, and their value before they are asserted may be non-logical. Therefore, the transitions are depicted using diagonal signals, and the asserted cycles are depicted using both high level and low level signals. Note that the address and data buses each have 32 bits. However, in the timing diagram we depict only a single bit, since all the bits have the same behavior.

1.2.3 Electrical Implementation

In a bus there are wires that multiple devices can write to. For example, the data signals can be written by every device (master or slave). We now describe the electrical implementation which enables multiple devices to write to the same wire.

There are two situations depending on the desired signal value when no device writes to a shared wire.

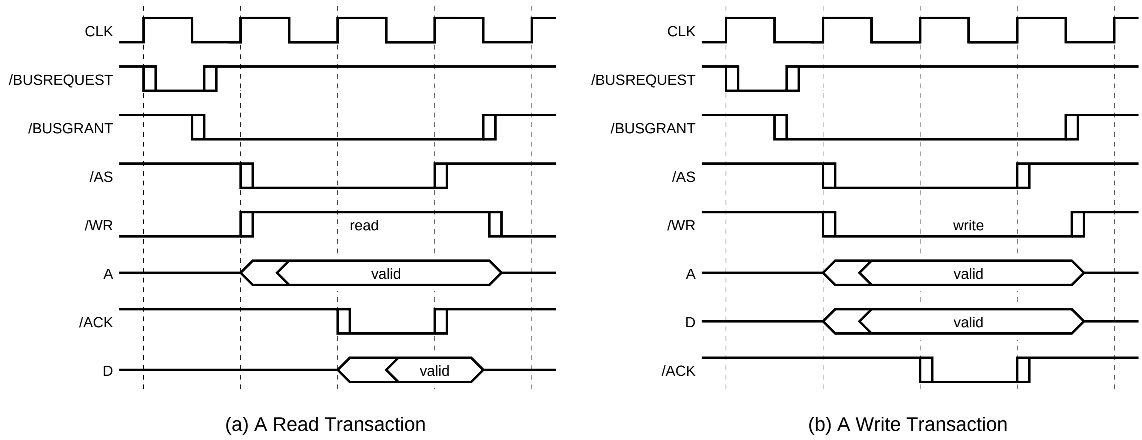

Figure 2: The backplane bus protocol

1. The signal value is meaningless when no device writes to the shared wire, namely, we do not care what its value is (it may be logical or even non-logical). In this case, the writing devices are connected to the shared wire via drivers, and each device generates a local driver enable signal. At every given point of time, at most one driver enable signal is active. If all the driver enable signals are inactive, the signal value on the wire is meaningless. For example, the values of the address and data signals are relevant only if there is a device that is writing them, otherwise, their value is meaningless.

2. If the signal value is important even when no device writes to the shared wire, then a special mechanism is used for enforcing a default signal value. The mechanism used for this purpose is called *open collector* or *wired OR*. Figure 3 depicts a wired OR mechanism. The devices are connected to the shared wire via drivers, and the wire is connected via a resistor to VCC (the power source). We consider 3 cases: (a) no device wants to write a value. In this case, all the driver enable signals are disabled, and the wire is high due to current flowing through the resistor. (b) at least one device wants to write a 0 to the wire. In this case, the driver enable signal is activated, and the driver pulls the wire down since the driver input is GND (ground). (c) one device wants to write a 1 to the wire. This case is reduced to case (a) by disabling all the driver enable signals.

   Note that if there is a device that wants to write a 0 and another device that wants to write a 1, then the 0 wins. Such a conflict does not create an error, although the protocol we consider does not allow such situations.

   The bus signals that require a wired OR mechanism are the address strobe (`as`), the write (`wr`), bus request (`br`), bus grant (`bg`), and the acknowledge (`ack`). The pull-up resistor of the wired OR signals is placed on the bus controller card. (Note that the bus request and bus grant signals are only written by a single device, and therefore, do not require a wired OR mechanism. However, the wired OR mechanism is also helpful in cases of errors in which a device writes to the bus even when it is not supposed to).

   Note that the mechanism proposed in Figure 3 computes the NOR of the driver enable signals /OE$_i$. This implies the usage of signals that are *active low* rather than *active high*. What this means is that instead of using the signals `as`,`wr`,`br`,`bg`,`ack`, we use the negated signals `/as`,`/wr`,`/br`,`/bg`,`/ack`.

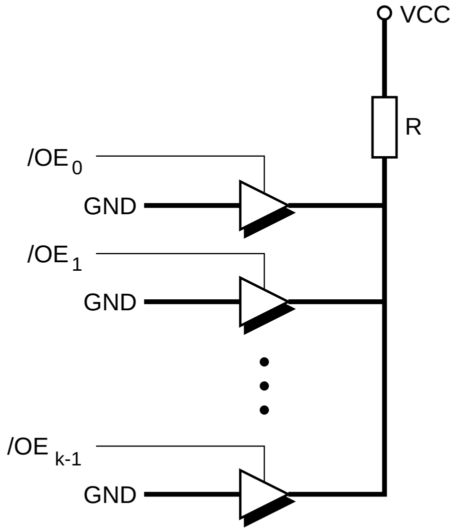

Figure 3: A wired OR

2 Homework

1. Consider a CPU that wants to communicate over the RESA bus as a master device. The CPU is connected to the RESA bus via a simple I/O interface. The simple I/O interface is placed on the CPU board and is connected to the CPU and to the RESA bus. Communication between the CPU and the I/O interface is implemented by 5 registers that are addressed by 3 bits. The functionality of the registers is as follows:

   $R0$: a data-in register through which data is fed to the CPU.

   $R1$: a data-out register through which data is sent from the CPU.

   $R2$: an address register that is written by the CPU.

   $R3$: a status register that is read by the CPU.

   $R4$: a command register that is written by the CPU.

   For example, a read transaction is implemented as follows: When the CPU wishes to read data from a slave, it writes the address (combined addresses of the slave and the data item) to the address register $R2$, and sets the command register $R4$ to a value that means "read". The I/O interface, handles the request, and initiates a read transaction over the RESA bus. When the data is fetched, it is stored by the I/O interface in the data-in register $R0$, and the status register $R3$ is set to "done". The CPU will then change the command register $R4$ to "idle", and the I/O interface then sets the status register $R3$ to "ready".

(a) Describe how a write transaction takes place.

(b) Draw the datapath, registers, and drivers of this I/O interface which is part of the CPU card, so that the CPU can communicate via the RESA bus.

(c) Design the control logic of the I/O interface. Define the control signals and their equations (include all the RESA bus signals). Use active low signals for all the signals that require a wired OR mechanism.

(d) Draw the timing diagram of the control logic including all the bus signals for a read transaction and a write transaction (include all the signals).

(e) Write the equation for the `/ack` signal in the slave device participating in such an I/O interface. Write equations for 1 and 0 wait states.

(f) Design the I/O interface using Schematic Entry and Abel. Simulate your design and compare the timing diagrams you prepared with the simulation results.