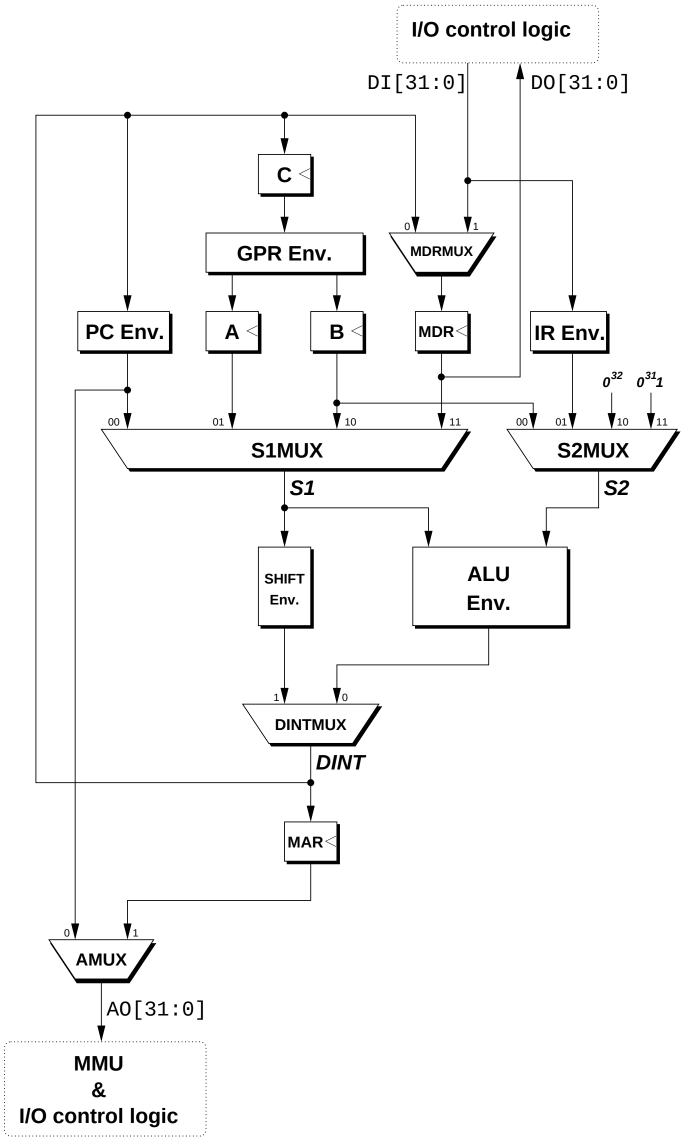
Computer Sturcture Lab Notes

Implemeting a DLX processor on an FPGA

by

Guy Even



Dept. of Electrical Engineering - Systems, Tel-Aviv University.

Fall 2001

Send comments to: *guy@eng.tau.ac.il*

Printed on October 11, 2001

Preface

Students in Computer Engineering programs attend courses on circuit design, computer organization, and computer architecture. I find it rather disturbing that even after attending all these courses, the task of designing a simple microprocessor from scratch is not considered a simple task. That means that all this studying fails to bring the students to what was already very well understood over a quarter of a century ago!

The course "Introduction to digital computers" was designed with the following goals in mind:

- Teach hardware as a formal discipline. This means that the functionality of hardware modules is specified accurately, their design is precisely described, and the correctness is rigorously proved.

  I made a special attempt to use material taught in previous courses whenever possible. In particular, recursion and induction are used whenever possible. In addition, notions and notations from graphs and graph algorithms were used to make discussions both concise and precise.

  An example of a topic that I like to cover with rigor and depth, that are not typical in courses of this level, is computer arithmetic. In particular, I cover the topics of fast adder designs (i.e. parallel prefix computation) and two's complement representation.

- Teach a complete microprocessor design. This includes the instruction set (the programmer's abstraction), the implementation (datapath and control), and even precise support of interrupts and exceptions (which helps understand the functionality of an operating system).

  The DLX microprocessor of Henessey and Paterson is selected as the microprocessor that is studied. This not only follows the current trend of microprocessor design teaching around the world, but it has the advantages of not having to discuss various "hairy" issues that always appear in commercial microprocessors. The course follows the books of Silvia Müller and Wolfgang Paul.

  The course does not address issues such as virtual memory, caching, pipelining, out-of-order execution, and branch prediction. With the exception of virtual memory, I regard all these issues as optimization techniques to be studied in the successive Computer Architecture course.

The goal of the "Computer structure lab" is to demonstrate the soundness of the theoretical DLX design taught in the course "Introduction to digital computers". The motivation follows the motivation of a physics lab; a physical law is taught in class and its correctness is demonstrated by an experiment in the lab. In my eyes, the main argument in favor of the "Computer structure lab" is that implementing a real microprocessor can help students understand the validity of the digital abstraction upon which digital design is based. More importantly, an unsuccessful implementation can provide ideas as to situations in which the digital abstraction fails!

The platform used in our lab is an educational computer called RESA. The RESA was developed in Prof. Paul's Institute of Computer Architecture and Parallel Computing in the Computer Science Department in the University of the Saarland in Germany. The initiative of designing and building the RESA was of Michael Braun (at the time a PhD student). Jörg Fischer designed one of the boards as a bachelor's project. The RESA is based on a Xilinx FPGA chip, and therefore, the lab uses Xilinx Foundation software.

The starting point of these lecture notes was the lab material from the hardware lab in in the Computer Science Department in the University of the Saarland in Germany. Together with Jörg Fischer, this material was adapted to the needs of our "Computer structure lab". In particular, we introduced built-in self monitoring to avoid having to use logic analyzers. More modifications and extensions were done jointly with Marko Markov and Danny Seidner. I would like to thank Jörg Fischer, Marko Markov, and Danny Seidner for their help and devotion in developing this lab.

The lab notes are meant to break the task of designing the DLX into small steps so that almost everything done during the course of the lab contributes directly to the DLX implementation. Needless to say, the lab and the lab notes are an evolving project; without feedback from students, we will not be able to further revise the lab material. Errata and other modifications will be posted in the lab's web side: `http://hyde.eng.tau.ac.il/Resa01/resa_home.html`

Finally, I would like to thank Wolfgang Paul for providing us with the RESA computers and for the many hours spent in discussions on how such a lab should be run.

Guy Even
Tel-Aviv, Oct. 2001

Contents

Chapter 1

Orientation

In this chapter we briefly discuss the environment and tools that are used in the Computer Structure Lab.
The lab consists of: (a) software - mainly the Xilinx Foundation Software, and (b) hardware - a special computer called RESA built for the purpose of a hardware lab.

1.1 The RESA computer

The RESA computer consists of three boards: a CPU board, a Memory board, and an I/O controller. These three boards communicate via a bus. A 4th board, called the I/O Interface, is in your PC. The PC can therefore communicate with the Memory board and the CPU board via the I/O Interface board (which happens to be connected to the RESA's I/O Controller)
The CPU board is a special type of board. It does not contain a microprocessor in the usual sense. Instead, it has an FPGA (Field Programmable Gate Array) chip. Loosely speaking, an FPGA is a device that can be programmed (or configured) while it is on the board (i.e. no need to take it out of the board and plug it into a special programming device). The FPGA's used in the RESA can be programmed to function like any circuit that has at most $25,000$ gates (this is not an accurate figure). The programming of the FPGA in the RESA is done by executing a special program on the PC which instructs the FPGA to configure itself according to a "file" that is stored either in the PC or in the RESA's Memory board. Such a configuration file defines the functionality of the FPGA, which in our lab will be a simplified DLX CPU. The DLX CPU accesses the memory for fetching of instructions, and load/store instructions. We need, therefore, a means to store programs in the RESA's Memory board (which will serve as the main memory of the CPU). Copying of RESA programs from the PC to the RESA's Memory board is done using a special program on the PC.
In order to do all that, the students need to know the following:

- How to design a DLX CPU in detail (all the gates).
- How to use the Xilinx software to design the DLX.
- How to test the correctness of designs (i.e. simulation).
- How to create a configuration file for the FPGA.

- How to configure the FPGA.
- How to run the RESA and check if it is running properly using a monitoring program.

1.2 Work flow

The Students' activity in the lab should follow these steps:

1. The first stage in the lab is called design entry. This is the stage in which you edit (or draw) the design using the software. There are two main methods in design entry: (a) schematic design: in this method a logical design is drawn using blocks (i.e. gates, flip-flops) and nets (i.e. wires). The software has a graphical interface which enables you to choose the type of block, name the ports of a block, and connect between the blocks. (b) HDL (hardware description language): in this method a design is described using a programming language. VHDL and Verilog are considered the most common languages these days. Another language is called ABEL, which is much easier to use. In our lab we will be using VHDL.

   It never hurts to think before designing, but in the case of this lab it is highly recommended! You should have a clear picture of what you plan to design and how it will work. Design entry is time consuming. If you cherish your time, make sure you know what you are designing before you start design entry.

   A general advice is to use a top-down approach: First, draw the "big blocks" and the interconnections between them. Only then proceed to design the details of the blocks. Do not crowd your design with too many blocks, lest your designs will become unintelligible spaghettis. Add remarks to your designs so that you (and the grader) can figure out what you had in mind. In general, clear drawings require shorter explanations and help one find mistakes much faster.

2. The second stage is called simulation. We are actually referring here to logical simulation (or functional simulation) that ignores timing issues (which you will learn later are very important). The simulator works under the "zero propagation" delay model. Namely,all the propagation delays are zero. This can sometimes be a bit confusing (e.g. what signal causes a change in values?) Simulation is software that enables you to choose input values and view the expected output values. This is the first and most important method of testing your design. Typical errors that you are supposed to detect at this stage are: mistakes in connections of wires, wrong polarities (i.e. connection of signal x instead of $not(x)$), incompatible names of signals, etc.

3. The third stage is implementation. During implementation a configuration file is created from our design. This stage is fully automatic in the easy cases (i.e. small designs), and all you do is ask the software to create the configuration file (called a HEX file). In larger designs, the automatic process fails (for example, the delay of the critical path is larger than the clock period). Some guidelines need to be given to the software to assist it in fitting the design into the resources of the FPGA. The configuration file (which resides on the PC) can be copied to the RESA using a special software on the PC, called the RESACTRL - the RESA Control Software.

After implementation, a simulation that does not ignore timing issues is possible. This simulation is called timing simulation, timing verification, or delay analysis. The main goal here is to see if the delay of the critical path is not larger than the clock period. We will elaborate more on this type of simulation later in the lab.

4. The fourth stage is running your design on the RESA. This stage is performed with the aid of the RESACTRL program. There are many options in this software, the simplest of which are writing to the RESA memory and reading values from the RESA memory. A DLX program can be executed by the following stages: copy program to RESA's memory, start execution of DLX, and read RESA's memory to see the outcome of the executed program. More options which allow monitoring of signals and registers in your design will be explained later.

Chapter 2

The RESA's Backplane Bus

In this assignment you will learn about the RESA's backplane bus. We start with a general description of buses, and continue with a description of the bus protocol used in the RESA's backplane bus.

2.1 What is a bus?

In this section we describe basic notions and terminology in busses.
Suppose we have multiple devices that want to communicate with each other (e.g. a CPU board and a memory board). The simplest way to enable communication is to connect all the devices 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 call each 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". The analogy of a transaction is a single telephone conversation: one party calls another party; some communication takes place, and the conversation is over. 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 (i.e. bursts). The party which initiates the transaction is called the *master* and the party which is asked to respond is called the *slave*. This means that devices are divided into two types: masters - devices that initiate transactions, and slaves - devices that respond to transactions. For example, a CPU board is a master and a memory board is a slave in bus transactions that implement load/store instructions.
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. It helps to view the address field as if it is divided into two parts: (a) the slave's name so that the slave can detect that it is being accessed in the current transaction; and (b) the name of the item within the slave that is being accessed. This means that each slave has an *address space* associated with it; different slaves have different address spaces.
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.
The *bus controller* coordinates the usage of the bus. The bus controller can be regarded as the "traffic light" of the bus.

2.2 The backplane bus protocol

In this section we describe the bus protocol used by the RESA's backplane bus.

Properties

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. There is an exception to this rule, as you will notice later.

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.

Signals

A bus protocol is an algorithm, and the data structures in a bus protocol are 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 ($\mathtt{br}_i$): Master device i (there are up to 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[1] the $\mathtt{br}_i$ signal to signal to the bus controller that it would like to start a transaction. Only master device i may write to the $\mathtt{br}_i$ signal. This signal can be viewed as a private link between master device i and the bus controller.

Bus Grant ($\mathtt{bg}_i$): A second private link between master device i and the bus controller is the $\mathtt{bg}_i$ signal. The $\mathtt{bg}_i$ 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 slave device that participates in this transaction that, in the next clock cycle, information will be sent on the bus. This information is sent by the master and the slave should "listen" to it. Note that when the Address Strobe signal is asserted, the address has not been asserted yet, and therefore, it is not determined yet which device is the slave device in the current transaction.

Address ($\mathtt{A}[31:0]$): 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 the slave's device name as well as the item's name within the slave. 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 ($\mathtt{D}[31:0]$): 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! In a write transaction, the master asserts the Data signals, and in a read transaction the slave asserts the Data signals.

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

[1] The term "to assert" means that it make a signal active. As you will see, some signals are active when they are low and some are active when they are high.

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?).

Protocol

It is common to describe bus protocols using *timing diagrams*. A timing diagram depicts the behavior of the signals during a transaction. Figures 2.1 and 2.2 depict 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 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 (i.e. undefined values).
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.

2.3 Wired-Or

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.

1. The signal value is undefined 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 devices with a write permission are connected to the shared wire via drivers, and each device generates a

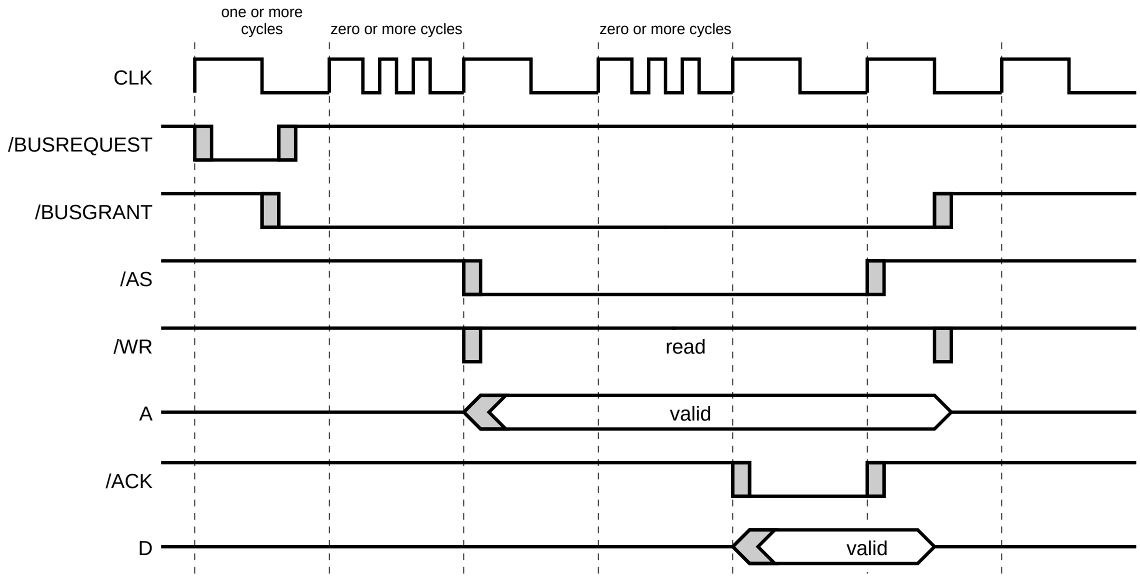


Figure 2.1: A read transaction in the backplane bus protocol (shaded areas denote periods of uncertainty)

local driver enable signal. The bus protocol guarantees that, 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 need not be even logical. 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 undefined.

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 *wired-OR*. Figure 2.3 depicts a wired NOR mechanism (we will return to the issue of NOR vs. OR shortly). The devices are connected to the shared wire via drivers, and the wire is connected via a resistor to VCC (the power source). The resistance of the resistor is much higher than that of an enabled driver, so when `/OE`$=1$, the voltage of the shared wire is low. 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). (Ground wins.) (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.

   This implementation of wired-NOR has the property 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 any electrical problems, but the RESA's bus protocol does not allow such situations.

   The bus signals that require a wired-OR mechanism are the address strobe (`AS`), the write (`wr`), 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

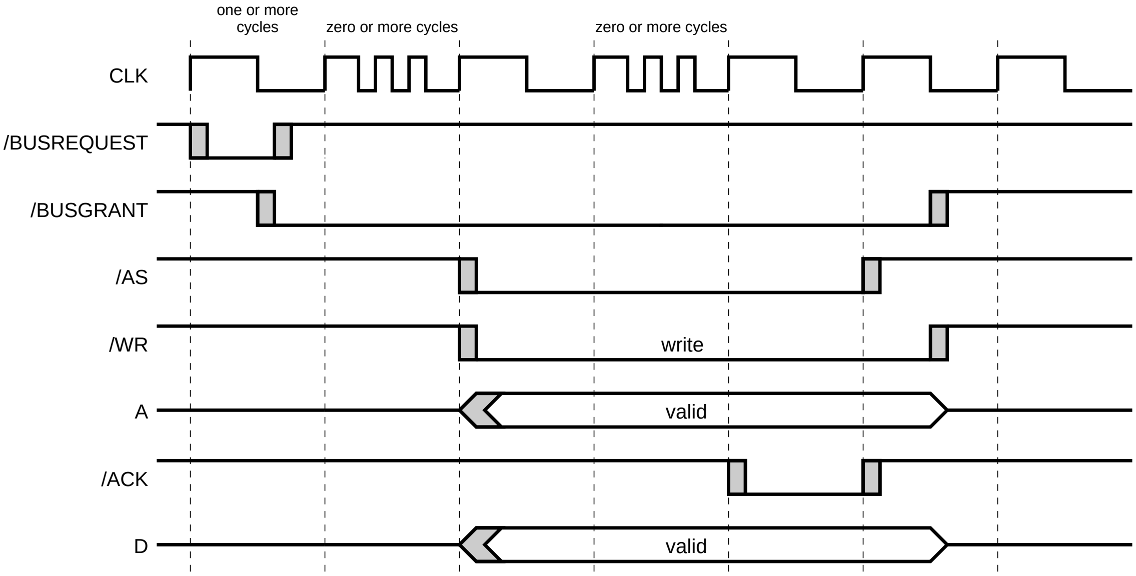


Figure 2.2: A write transaction in the backplane bus protocol (shaded areas denote periods of uncertainty)

a single device, and therefore, do not require a wired OR mechanism. The RESA designers chose to use wired-OR to deal with a situation where two master devices are connected by mistake to the same `br` signal. A conflict in a wired-OR mechanism does not cause electrical problems, whereas two drivers that "fight" each other can cause damages.)

Note that the mechanism proposed in Figure 2.3 computes the NOR of the driver enable signals $\texttt{/OE}_i$. This is equivalent to computing $AND(\texttt{OE}_0, \ldots, \texttt{OE}_{k-1})$. This is why *active low* signals are used rather than *active high*. Namely, instead of using the signals `AS`,`wr`,`br`,`bg`,`ack`, we use the negated signals `AS_N`,`/wr`,`/br`,`/bg`,`/ack`.

Supplementary Bibliography

W. Stallings, *Computer Organization and Architecture*, Prentice Hall International Inc., Chapter 3.4.

J. M. Feldman and C. T. Retter, *Computer Architecture*, McGraw Hill International, Chapter 8.3.

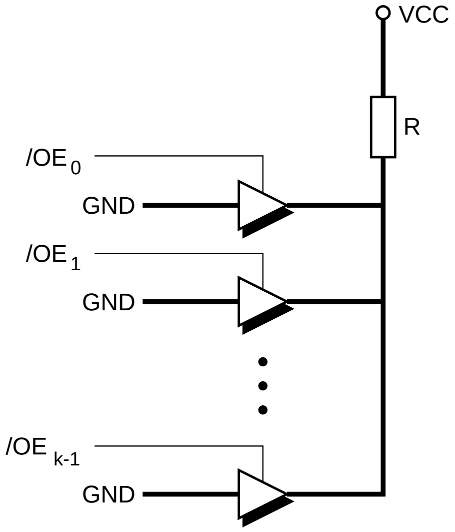


Figure 2.3: A wired NOR (i.e. $NOR(\texttt{/OE}_0, \ldots, \texttt{/OE}_{k-1})$)

Chapter 3

A Simple Slave Device

This chapter describes what you need to know so that you can design a simple slave device on the RESA computer. We overview the structure of the RESA, the I/O CONTROL LOGIC, and then describe the slave device.

3.1 RESA Architecture

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.
Figure 3.1 depicts the parts of the RESA Computer. The RESA Computer consists of:

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

Memory board. The memory board contains two types of memories: a read-only memory and a read-write memory. The type of memory that is accessed depends on the address. A read-only (non-volatile) memory which is based on EPROM chips (erasable programmable ROM) is used for storing a fixed program (the test program). This fixed program can be executed by the CPU after reset. A read-write memory based on DRAM (dynamic RAM) chips serves as the main memory of the CPU.

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 computer is implemented by a connection from an ISA special purpose card placed in one of the PC slots to the bus controller card.

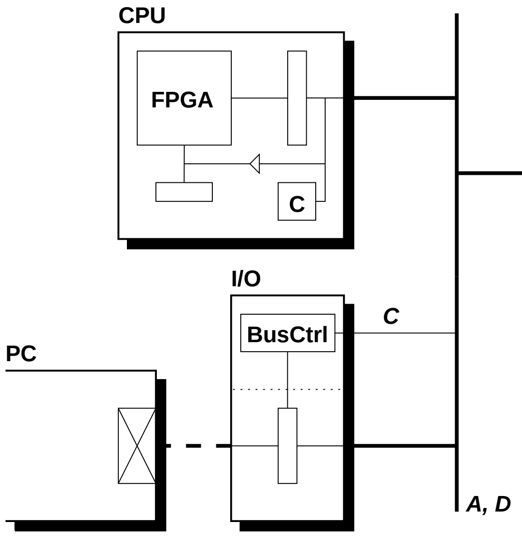


Figure 3.1: The RESA cards and buses

3.2 The RESACTRL Program

The RESACTRL program is a suite of programs which is used to diagnose and setup the RESA. We describe the parts of the RESACTRL program which will be used by you.

1. Configure RESA CPU board. This is how the FPGA is programmed. There are two options. In the first option, the design is read from the EPROM which is placed on the processor board. In the second option, the design is read from a file located in the PC.

2. Access RESA Memory board. Data can be read and written to the memory board of the RESA. This is useful mainly to read blocks of memory after an application (e.g. a CPU) on the RESA completes its execution.

3. Run and Debug. The part has a few roles.

   (a) Upload programs. We often refer to the bus master as an application, for example, a CPU. Uploading a program means storing a program in the application's language in the memory board.

   (b) Set single-step mode or continuous mode. The purpose of this feature is to enable you to run your applications in single-step mode. This means that the application executes only one "instruction" and waits for the monitor program to allow it to execute the next "instruction". Between instructions, you can monitor various values in the memory and application. Usage of this feature requires that the application wait for a "start" signal before executing a new "instruction".

   (c) Run the PC monitor program. The PC monitor program enables one to initiate read and write transactions. This is a very powerful tool, as you will see later in the lab when we use a concept called "build-in monitoring". In this assignment you will be using the PC monitor program to initiate read transactions from the slave device that you are designing. This way you can see if your slave is functioning properly.

The PC monitor program reads a configuration file (not to be confused with the HEX file used for configuring the FPGA). The configuration file of the PC monitoring program contains a list of physical addresses and labels attached to these addresses. You can use the PC monitor program to initiate read transactions that read the values of the signals and registers corresponding to these labels. A sample configuration file is attached to this handout (you may ignore in this assignment all the references to the Logic Analyzer).

3.3 The I/O control logic

In the previous assignment you designed a bus interface between the RESA bus and a bus master device. There is a risk in connecting devices designed by students to the bus, the reason being that design errors might cause drivers to be in conflict (e.g. two drivers connected to the same bus, one trying to pull the signal up, and the other trying to pull it down). The solution is to provide you with a bus interface module, called the I/O control logic. You will not be able to configure the FPGA unless your design uses this interface.

The I/O control logic serves as the bus interface for one bus master device and one bus slave device. Figure 3.2 depicts the I/O control logic. The RESA bus is drawn within the I/O control logic since both the application and the Monitor Slave access the RESA bus via the I/O control logic. The design environment does not enable you to access the RESA bus directly or even the FPGA's I/O pins. Hence, the "outer world" is hidden from you and the I/O control logic encapsulates the "outer world". The I/O control logic is divided into two parts: one which serves the bus master device and the other serves the bus slave device.

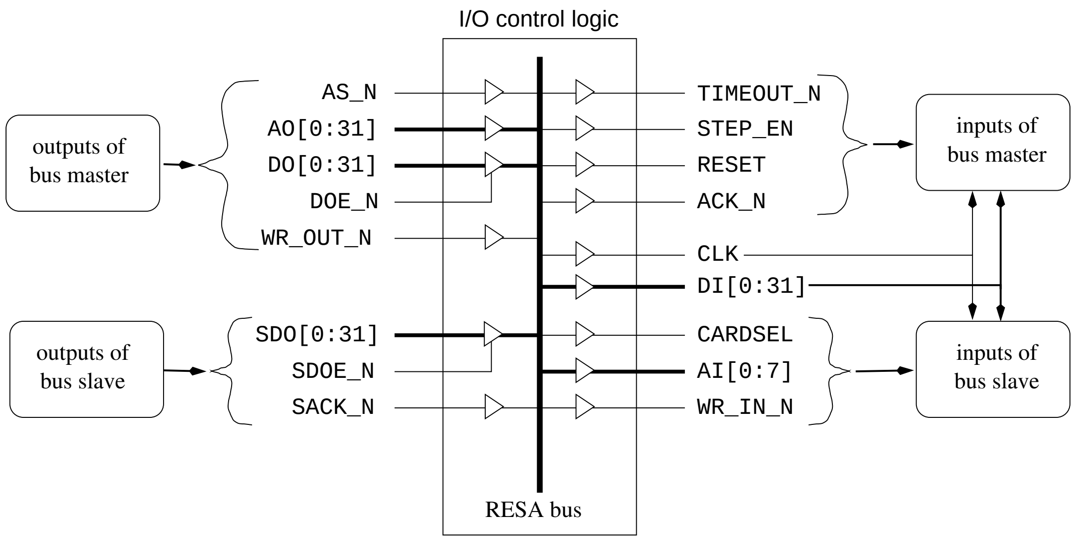


Figure 3.2: Connections between the I/O control logic and the master and slave devices on the FPGA

We start by describing the outputs and inputs of the I/O control logic that serve the bus master device. The following signals are fed to the bus master device:

1. `clk`: clock.
2. `STEP_EN`: This signal notifies the bus master (e.g. a CPU) that it should run in single-step mode. If the bus master is a counter, then the `STEP_EN` signal is a simple clock enable signal. If the bus master is a DLX processor, then the `step_en` signal tells the DLX control whether a new instruction should be executed or not.
3. `RESET`: A reset signal.
4. `ack`: The acknowledge signal is sent by the slave in a bus transaction. This signal is active low.[1]
5. `TIMEOUT`: The timeout signal is generated by the bus controller to indicate that a bus transaction failed. This signal is active low. Whenever a bus transaction starts, the bus controller starts counting. After a certain amount of time (128 cycles), if the bus transaction is not over, then the bus controller terminates it by generating an `ack` and by generating a `TIMEOUT`. The `ack` signal causes the transaction to end, and the `TIMEOUT` signal tells the master that the transaction failed. A situation, where such an intervention by the bus controller is necessary, is when the master tries to access an address which does not correspond to any device on the bus. When such an access occurs, no device will respond, and unless the bus controller kills the transaction, the transaction will never end (why is that undesirable?).
6. `DI`$[0:31]$: This is the data-in bus of the master and slave devices (both the bus master and the bus slave share these signals).

The following signals are fed to the I/O control logic by the bus master device:

1. `AS_N`: This is the address-strobe signal. This signal is active low.
2. `AO`$[0:31]$: This is the address-out bus (no sharing with the slave device).
3. `DO`$[0:31]$: This is the data-out bus (no sharing with the slave device).
4. `DOE_N`: This signal is the data output enable signal. It is active low. When this signal is low, the bus master application wishes to assert the `DO`$[0:31]$ signals to the RESA data-bus.
5. `WR_OUT`: This is the `wr` signal generated by the bus master. This signal is active low.

We now describe the outputs and inputs of the I/O control logic that serve the bus slave device. The following signals are fed to the bus slave device:

1. `clk`: clock.

[1] The design entry tool is not capable of handling signal names starting with the symbols "/" and "!", which makes `/ack` an illegal name. Therefore we denote the polarity of active low signals by adding the suffix "`_N`" to the signals names.

2. DI$[0 : 31]$: This is the data-in bus (shared with the bus master device)

3. CARDSEL: This signal indicates that the slave of the current transaction is on the CPU board. Not all 32 bits of the address bus are fed to the slave device. The CARDSEL signal is active when the value of the address bus corresponds to the addresses of devices in the FPGA in the CPU board. The CARDSEL signal is computed by the I/O control logic from the 23 upper bits of the address in the RESA bus.

4. AI$[0 : 7]$: This is the address-in bus of the slave device. We will later introduce a special type of slave device called the Monitor Slave. We use the convention that the address space of the Monitor Slave is restricted to addresses with AI$[7] = 1$. Therefore, there can be 2^7 different addresses for devices in the FPGA.

5. WR_IN: This is the wr signal that is input to the slave device. This signal is active low.

The input signals from the slave device are:

1. SDO$[0 : 31]$: Data-out bus.

2. SDOE: The data output enable signal. It means that when the SDOE_N signal is low, the slave device wishes to write the SDO$[0 : 31]$ signals to the RESA data-bus.

3. SACK: This is the ack signal generated by the slave device. This signal is active low.

3.4 Simple Master and Slave Devices

In this assignment you will design a trivial master device and a simple slave device. The slave device is connected to nets of the master device by "private wires" (i.e. the private wires are not part of the bus). The values monitored by the slave device are reported in read transactions that the slave participates in. We will be using an extension of this mechanism to design applications (i.e. a DLX) with "built-in monitoring".
The master device we use for this assignment is a 32-bit binary counter. This master device does not initiate any bus transactions, and is therefore, a degenerated master device.
In addition, your design should include a 32 bit register, called the ID-register. The ID register is a 32-bit register that stores the "code" of your lab group. By reading this value, it can be verified that the design is indeed yours. You should use this to make sure that you are executing your own design.
The slave device in this assignment reads either the value output by the counter or the value stored in the ID-register. The address space of the slave device consists of two addresses; one for the counter's value and one for the ID-register. When the PC monitor program wishes to read the counter's value, it initiates a read bus transaction with the address of the counter's output. The slave device receives this request, and routes the counter's output to the SDO-bus. The slave device then acknowledges that the requested data has been sent, and the read transaction is completed. A similar procedure occurs when the PC monitor program wishes to read the value of the ID register.

3.5 A Sample Configuration File

```
// Example of a configuration file for the PC monitor program
// Description file *EXAMPLE* for IDLX

// CPU Board addresses
c PC        0x80
c IR        0x81
c DO        0x82
c AO        0x83
c AR        0x84
c ANALYZER  0x85
c Command   0x86
c Status    0x87

// Memory addresses
m CNT     0x80011a
m Var_X   0x800056
m Var_Y   0x800059

// Waveform bits
- specifying the control signals

w AS     0x07
w ACK    0x00

w ADD    0x03
w AluF0  0x04
w AluF1  0x05

w dcode  0x0b
w delse  0x0c
w DIsel  0x0d
w IRce   0x0f

- specifying the internal DATABUS

w DINT10 0x10
w DINT11 0x11
w DINT12 0x12
w DINT13 0x13
w DINT14 0x14
w DINT15 0x15
w DINT16 0x16
```

```
w DINT17 0x17
w DINT18 0x18
```

Remarks:

1. Order of declarations only effects the order in which the values are displayed.
2. (a) Addresses in lines beginning with a "c" are given labels that correspond to the CPU board.
   (b) Addresses in lines beginning with an "m" are given labels that correspond to the memory board.
   (c) Addresses in lines beginning with a "w" refer to the local RAM of the Logic Analyzer. This is a 32×32-bit RAM, and the address refers to a column in this RAM (i.e. the ith bit in each word).
   (d) Lines beginning with a letter not in $\{c, m, w, C, M, W\}$ are treated as comments (note that the first letter is not case sensitive).
3. Reserved labels are (not case sensitive): Analyzer, Command, Status. The meaning of these reserved labels is:
   (a) Analyzer specifies the address of the Logic-Analyzer (which is an FPGA-Slave). The Logic Analyzer contains a 32×32-bit RAM. We assume that the PC monitor program does not attempt to write to this RAM (only read transactions will be made with the Logic Analyzer by the PC monitor program).
   (b) Command specifies the address of a register in the Logic Analyzer. This register has two fields: the lower 5 bits serve as an address of a row in the Logic Analyzer's RAM. The upper bits of the Command Register may be used to select which data is routed to the data-in input of the Logic Analyzer's RAM.
   (c) Status specifies the address of a register in the Logic Analyzer that holds the number of rows of the Logic Analyzer's RAM that have been filled with fresh data. Loosely speaking, in single-step mode, the application executes "instructions" one by one. The execution of an instruction may take several clock cycles during which data is routed to the Logic Analyzer's RAM, filling it word by word as the clock cycles proceed. The Status Register will hold the number of cycles that have elapsed, so that the PC monitor program can know which rows of the Logic Analyzer's RAM hold relevant data.
4. Illegal lines will simply be ignored without reporting any errors!
5. Labels have a maximum length: c-type labels and m-type labels: 12 characters; w-type labels: 5 characters.
6. Valid address ranges are:
   - c-type: 0x80 - 0xff

- m-type: 0x800000 - 0x8fffff
- w-type: 0x00 - 0x1f

All addresses MUST be specified in hexadecimal!

Chapter 4

Built-In Self Monitoring

4.1 What is built-in monitoring?

Suppose you have designed your favorite design (i.e. a DLX processor), and simulation indicates that the design is correct. You upload the design to the FPGA and you let the design start running (i.e. the DLX runs a program that is stored in the memory). However, you do not see the results that you expected to see (assume that after a sufficient amount of time, you use the PC monitor program to read the RESA memory contents). What do you do then?

What we propose is not to be surprised by such an event, but to be prepared for it. Wouldn't it be useful to be able to run the DLX processor step by step and monitor (i.e. view) the values of registers and control signals? This sounds like standard debugging, and indeed, this is what we are after.

To enable such monitoring, our designs will include the hardware that will be responsible for reading various values in the DLX processor without changing the DLX behavior.

4.2 General Structure

Figure 4.1 depicts the general structure that we use for the built-in monitoring. The figure depicts only two components of the RESA: (a) The FPGA contains three modules: the application (e.g. the DLX processor), the Monitor Slave, and the I/O control logic. (b) The PC monitor program.

The PC monitor program is a program that runs on the PC and behaves as a bus master that is controlled by you. Monitoring the application is achieved by initiating bus transactions that accesses a special slave device on the FPGA called the Monitor Slave. The Monitor Slave is a circuit that reads internal signals in the application, stores them, and sends them to the PC monitor program whenever requested to do so.

The FPGA is programmed so that it contains three components: the application, the Monitor Slave, and the I/O control logic. The I/O control logic gives the application and the Monitor Slave the abstraction that they are directly connected to the bus. The application is a bus master. The Monitor Slave is a bus slave, and is meant to participate only in bus transactions that are initiated by the PC monitor program.

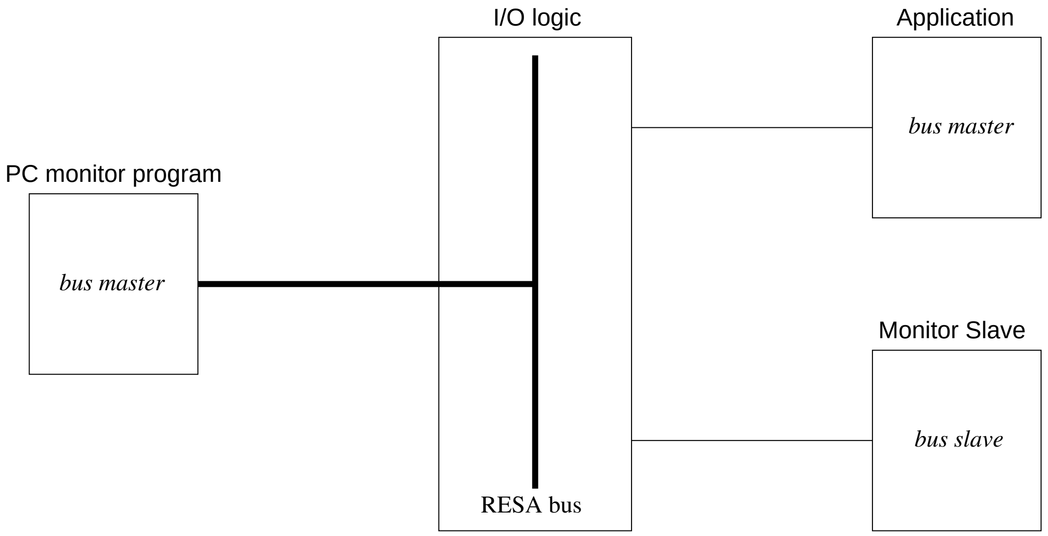


Figure 4.1: General structure of the monitor

In a normal execution (without monitoring), the PC monitor program does not initiate any bus transactions, and therefore, the Monitor Slave is idle.
In Chapter 3 you saw that implementing a Monitor Slave that can report current values of registers in the application is not a complicated task. It can be done as follows:

- Connect special wires from the registers in the application to a multiplexer in the Monitor Slave.
- The multiplexer selects an incoming signal based on the address in the read transaction.
- The selected signal is send as the data in the read transaction.

4.3 Monitoring Tasks

Before describing more complicated monitoring tasks, we need to describe the application. Our goal is to design a processor that executes DLX instructions. The control of the DLX starts each instruction execution in the "fetch" state, passes through a few other states and returns to the "fetch" state. We will have two modes: single-step and continuous. In single-step mode, each execution of an instruction waits until an appropriate signal arrives from the PC monitor program. In continuous mode, the execution of the next instruction is unconditional.
Suppose we wish to monitor signals and register values of the DLX. Consider the following cases:

1. If the DLX is running in continuous mode, we will get a "choppy motion picture". For example, we could try to monitor the value of the PC register (to know which instruction is executed). However, the read transaction might require more cycles than an instruction execution. That would mean that by the time the PC monitor program receives a value, it is no longer relevant. Moreover, consecutive reported values are likely to be non-consecutive

values of the PC register. Such monitoring would provide only very partial information (e.g. is the value of the PC register changing over time?) but not full information (e.g. what is the sequence of executed instructions?).

2. If the DLX is running in single-step mode, we will be sampling signals only when the DLX is in the "fetch" state. This does not suffice. Consider, for example, the following situations:

   - Monitoring the bus activity of the DLX. How can the Monitor Slave report signals related to bus transactions in which the application is a master? There seems to be a conflict since the Monitor Slave uses the bus to report values.
   - Monitoring internal signals of the application over several consecutive cycles. The execution of an instruction in the DLX takes a few cycles (e.g. "fetch", "decode", "execute", "write-back"). Even if the DLX is running in single-step mode, a Monitor Slave that reports only current values is limited to reporting the signals during the "fetch" state.

Our conclusion is that reporting current values is not enough for debugging a design. The purpose of this handout is to design a Monitor Slave that can monitor signals from the application and store the sampled values during a few consecutive clock cycles. These stored values can be later reported to the PC monitor program.

4.4 The Logic Analyzer

The part of the Monitor Slave that stores past signals is called the Logic Analyzer. The Logic Analyzer contains a RAM in which the monitored signals are stored. These sampled values can be reported later to the PC monitor program. This implementation enables us to monitor signals during an instruction execution. (Reminder: An instruction execution is the interval of clock cycles between two consecutive entries to the "fetch" state.) If we want to monitor control signals and register values during the execution of a single instruction, we should be able to do the following:

1. Store the monitored signals cycle by cycle during the execution of an instruction.

2. After the instruction's execution is completed, be prepared to answer bus read transactions in which the PC monitor program asks about the sampled values.

The structure of the Logic Analyzer and its connections with the Monitor Slave and the application is depicted in Figure 4.2. The Logic Analyzer consists mainly of two parts:

1. The Logic Analyzer's RAM. This is a 32×32-bit RAM that stores the sampled values from the application. In each clock cycle, up to 32 signals (i.e. bits) can be stored.

2. The counter. The 5-bit counter generates the address into which sampled values are stored. In the beginning of an execution of an application's instruction, the counter is reset by the application. The counter's output value equals the number of cycles that have elapsed since the beginning of the last instruction. This way, the Logic Analyzer's RAM is filled "row by

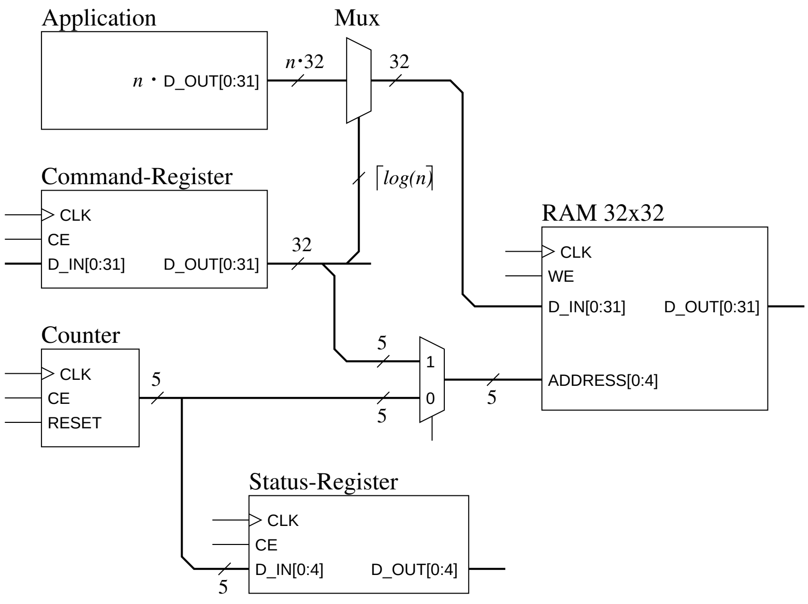


Figure 4.2: The structure of the Logic Analyzer and its connections with the Monitor Slave and the application.

row" with the sampled values until the execution of the current instruction ends. The Monitor Slave can read the value of the counter and send it to the PC monitor program, so that the PC monitor program knows how many queries to make.

The Monitor Slave has two registers that are related to the Logic Analyzer.

1. The Command Register. This 32-bit register is divided into two fields.

   (a) The bottom 5 bits are used to address the Logic Analyzer's RAM when the PC monitor program reads the sampled values (Recall that the address corresponds to the number of cycles that have elapsed since the beginning of the execution of the monitored instruction). Note that the PC monitor slave needs to be able to write values to the Command Register. This means that the Monitor Slave should also support write transactions.

   (b) The top bits are used to select which 32-bit data set is routed from the application to the data-in input of the Logic Analyzer's RAM. Again, this bits are written by the PC monitor slave.

2. The Status Register latches the value of the Logic Analyzer's counter so that the Monitor Slave can report the number of rows that contain relevant data in the Logic Analyzer's RAM. Note that the output of Logic Analyzer's counter is registered. This means that one does not really need a new register to store the counter's output since it suffices to deactivate the counter enable input of the counter.

Chapter 5

A Read Machine and A Write Machine

This chapter deals with designing a bus master that is capable of initiating bus transactions. We consider two types of machines: a read machine and a write machine.

5.1 A Read Machine

The Read Machine is an application that reads the contents of a fixed memory address (0x800000) and stores the value in a register. The Read Machine is connected as a bus master to the I/O Control Logic.
The state diagram of the Read Machine is depicted in Figure 5.1. The functionality of the Read Machine is as follows:

1. The machine exits the "wait" state when the STEP_EN signal is active.

2. The machine initiates a read transaction in the "fetch" state.

3. The machine waits for an `ack` signal during the "wait4ack" state.

4. The machine writes the fetched value in its register when entering the "load" state. Note that the machine keeps the address on the Address bus valid for half a cycle during the "load" state.

5. The `reset` signal causes the machine to transition to the "wait" state.

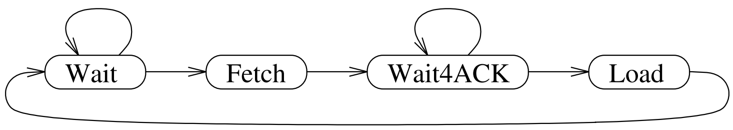


Figure 5.1: State diagram of the Read Machine

5.2 A Write Machine

The Write Machine is an application that writes your favorite value (i.e. 17) to a fixed memory address (0x800000). The Write Machine is connected as a bus master to the I/O Control Logic. The state diagram of the Write Machine is depicted in Figure 5.2. The functionality of the Write Machine is as follows:

1. The machine exits the "wait" state when the STEP_EN signal is active.
2. The machine initiates a write transaction in the "store" state.
3. The machine waits for an `ack` signal during the "wait4ack" state.
4. Note that the machine keeps the address and the data (on the Address and Data busses) valid for half a cycle during the "terminate" state.
5. The `reset` signal causes the machine to transition to the "wait" state.

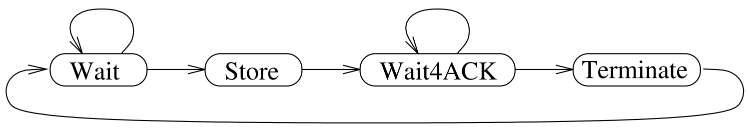


Figure 5.2: State diagram of the Write Machine

Chapter 6

A Load/Store Machine

This chapter focuses on designing memory accesses in the DLX design. To focus on memory accesses, we will consider a primitive application, called the Load/Store Machine. The Load/Store Machine executes DLX programs that consist only of simplified load and store instructions.
The only difference between the memory accesses of the Load/Store Machine and a word-addressable DLX is in the computation of the effective address (which is simpler in the Load/Store Machine).
To simplify the design task, we will introduce a simulation environment that enables one to simulate a Load/Store Machine connected to a RESA bus.

6.1 Specification

The instruction set of the Load/Store Machine consists only of load and store instructions.

6.1.1 Modified Load/Store Instructions

Load/Store	Semantics
`lw    RD R0 imm`	RD := M(imm)
`sw    RD R0 imm`	M(imm) := RD

Note that we allow the source register to be only `R0`. Recall that the value stored in Register `R0` is always zero. This means that the computation of the effective address does not require adding the source register and the immediate constant.

6.1.2 I-Format

Load/store instructions are encoded in the I-Format. An instruction in the I-Type-Format is divided into four fields depicted below.

6	5	5	16
Opcode	*RS1*	*RD*	*immediate*

6.1.3 Encoding

The load instruction is encoded by IR$[31:26] = 100011$. The store instruction is encoded by IR$[31:26] = 101011$.

6.2 Memory Accesses

In the the Read and Write Machines a memory access was implemented using a state machine with 4 states. However, we would like to simplify the control of the DLX and allocate only one state for each of the actions: “fetch”, “load”, and “store”. How do we do bridge the gap between 4 states for a memory access and a single state? The way this is done is by cascading state machines.
Consider the Load/Store Machine. It accesses the main memory during “fetch”, “load”, and “store” states. Communication between the Load/Store Machine and the I/O Control Logic is done via a state machine called the “Memory Access Control”. The Memory Access Control resembles the Read and Write Machines. The connections between these three state machines is depicted in Figure 6.1. The state diagram of the Memory Access Control is depicted in Figure 6.2. The control signals have the following meanings:

1. `mr` means memory read. It is active during the “fetch” and “load” states.
2. `mw` means memory write. It is active during the “store” state.
3. `req` means either `mr` or `mw` is active.
4. The other signals connect to the I/O control logic and are well understood by now.

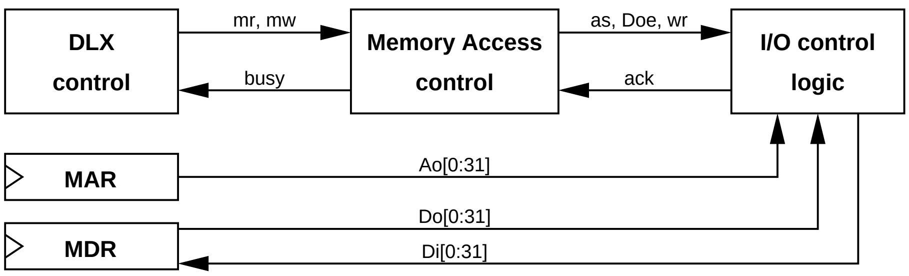


Figure 6.1: Memory accesses in the DLX machine

6.3 Implementation

We outline the datapath and control of the Load/Store Machine.

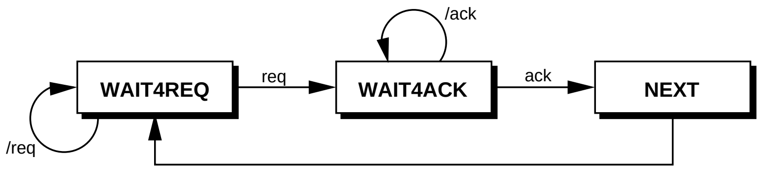


Figure 6.2: Finite state diagram of the memory access control.

6.3.1 Datapath

Figure 6.3 depicts a block diagram of the datapath of the Load/Store Machine. Buses are connected to the I/O control logic, as depicted in Fig. 6.1. Control signals are omitted from this figure, and you are asked to decide which signals are needed and when they are active.

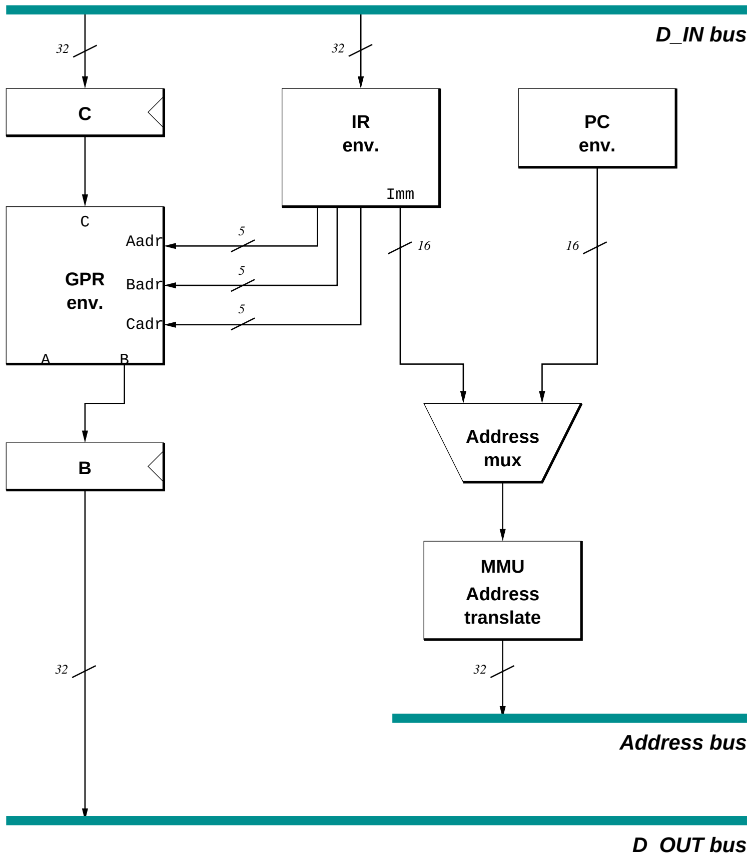


Figure 6.3: Block diagram of Load/Store Machine datapath.

6.3.2 Control

Figure 6.4 depicts a state diagram of the control of the Load/Store Machine. The RTL instruction in each state is specified in Table 6.1. A `reset` signal causes a transition to the INIT state.

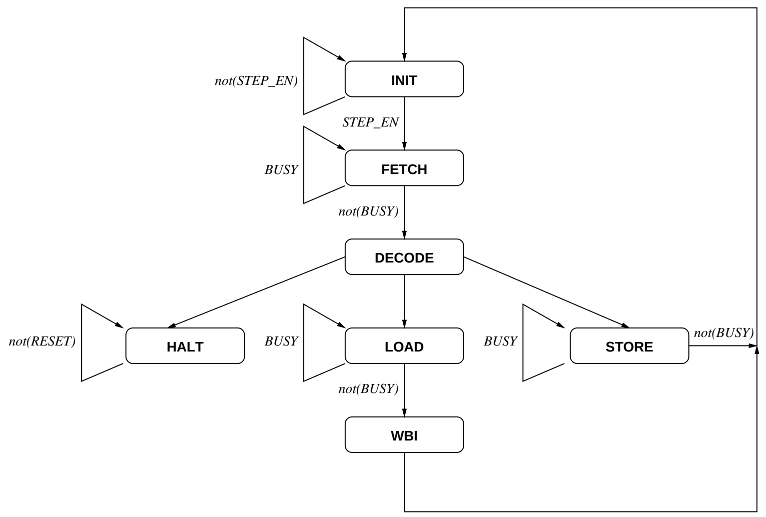


Figure 6.4: State diagram of the control of the Load/Store Machine. An active `reset` signal causes a transition to the INIT state.

State	RTL Instruction
INIT	*wait for step enable*
FETCH	$IR = M(PC)$
DECODE	$B = RD$, $PC = PC + 1$
STORE	$M(Imm) = B$
LOAD	$C = M(Imm)$
WBI	$RD = C$
HALT	*machine is stuck till reset*

Table 6.1: RTL instructions for the states of the control

6.3.3 Translating logical addresses to physical addresses

In the Load/Store Machine, the logical addresses of the machine are translated to the physical address space 0x00800000 - 0x008fffff.

In the DLX microprocessor, we want to have two modes. In the first mode, no translation takes place (i.e. the logical and physical addresses are identical). In the second mode, we wish to restrict memory accesses to addresses in the range 0x800000-0x8FFFFF. The reason we wish to have both modes is that we would like to run two types of programs in the DLX microprocessor: (a) test programs stored in addresses starting with zero (which means that no translation should take place at all), and (b) user programs. Memory access during user programs is done under the assumption that the memory space starts at zero. However, the lower part of the memory, that stores the test programs, is read-only (i.e. implemented using EPROM). Therefore, during execution of user programs, the addresses of memory accesses are translated to the range 0x800000-0x8FFFFF. From a hardware point of view, the addresses in the range 0x800000-0x8FFFFF correspond to DRAM, which enables both reading and writing.

Our goal is not to have to worry about the memory translation when we design the Load/Store machine (and later the DLX microprocessor). For example, we want to set the PC register to zero whenever a reset signal is active. This uniform abstraction is obtained by using a Memory Mapping Unit (MMU) that maps a physical address to a logical one. The inputs to this mapping are the AO (address out) bus and a mem_map signal. The output of the MMU is the MMU_AO bus that connects to the I/O control logic. The mem_map signal tells the MMU how to translate the logical address into a physical one. In the Load/Store Machine, the signal mem_map is always set to 1. **The MMU should set address bits in positions** $[31:24]$ **to zero unconditionally to avoid access to restricted areas.**
We suggest that the mem_map signal be taken from bit 23 of the command register. This bit controls the functionality of the MMU as follows: If the memory space should start at 0x0 (i.e. the test program is running), then MMU_AO=AO. If the memory space should start at 0x00800000 (i.e. a user program is running), then MMU AO = AO OR 0x00800000. You may assume that the user program's logical address space is limited to $[0, 2^{23} - 1]$ to simplify the MMU design.

6.4 The GPR environment

The GPR environment has the following inputs: C, Aadr, Badr, Cadr, gpr_we. The outputs are A, B. Note that the Load/Store Machine could do with one output, however, we specify two outputs so that this module can be used for the DLX design as well. It can support one of two operations in each cycle:

1. Write the value of input C in register $\mathtt{R}[\mathtt{Cadr}]$ if $\mathtt{gpr\_we} = 1$.
2. Read the contents of the registers with indexes Aadr and Badr. The outputs A and B are defined by:

$$\mathtt{A} = \begin{cases} \mathtt{R}[\mathtt{Aadr}] & \text{if } \mathtt{Aadr} \neq 0 \text{ and } \mathtt{gpr\_we} = 0 \\ 0 & \text{if } \mathtt{Aadr} = 0 \text{ and } \mathtt{gpr\_we} = 0 \\ \text{arbitrary} & \text{otherwise.} \end{cases}$$

and

$$\mathtt{B} = \begin{cases} \mathtt{R}[\mathtt{Badr}] & \text{if } \mathtt{Badr} \neq 0 \text{ and } \mathtt{gpr\_we} = 0 \\ 0 & \text{if } \mathtt{Badr} = 0 \text{ and } \mathtt{gpr\_we} = 0 \\ \text{arbitrary} & \text{otherwise.} \end{cases}$$

6.5 A simulation environment

Simulation refers to computing values of signals when the circuit is fed by given input values. The problem with simulating a design like the Load/Store Machine is that it interacts with other devices through the RESA bus. It is not a trivial task to generate manually the signals fed to the Load/Store Machine by the RESA bus. To enable a simulation environment in which you do not need to determine the values of the RESA bus signals, a module called IO_SIMUL was designed. The IO_SIMUL Module encapsulates the I/O Control Logic, the RESA bus, the bus controller, and the main memory. By combining your design with the IO_SIMUL Module, you can simulate your circuit as if it is connected to the RESA bus.

Figure 6.5 depicts the inputs and outputs of the IO_SIMUL module. The signals on the left hand side are the signals from the Memory Access Control, the MAR, and MDR to the I/O Control Logic. The signals on the right are signals from the I/O Control Logic. The signals on the bottom are: `N0` - the global clock, `STEP` - the step enable signal of the bus master, and `RST` - a global reset signal. The `RST` signal resets the bus master, resets the bus controller, and sets the contents of the main memory to their initial value (the initial values are specified in the "`read16`" module which is part of IO_SIMUL module).
The IO_SIMUL module is designed using ABEL. You can change the values stored in the main memory during reset. A sample file is given in Section 6.7.

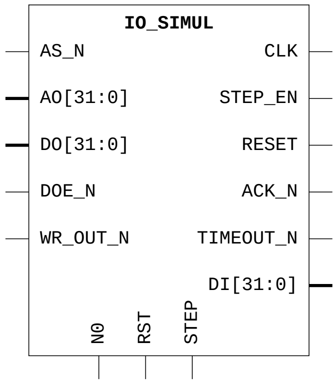


Figure 6.5: The IO_SIMUL module.

6.6 Testing

In this section we describe a testing procedure that will enable you verify the correctness of your design. The procedure is divided into three stages:

1. Test by simulation the transitions of the control of your design.
2. Testing of RTL instructions. During this step, contents of registers are checked as well as control signals. This step can be done both by simulation and by monitoring.
3. Testing executions of whole instructions. During this step the contents of registers and contents of memory (i.e. addresses into which values are stored) are checked. It suffices to perform this step by monitoring.

We elaborate on the first two testing steps.

6.6.1 Testing a finite state machine

The task of testing combinational circuitry by simulation should be well understood by now. We focus on testing of finite state machines (i.e. the control of the Load/Store Machine).

We will be using simulation to perform testing of the control. The goal is to test if all the transitions of the finite state machine are correct. This can be done by "covering" all the transitions of the control by paths. Namely, find a set of paths (each path starting in the initial state), such that every transition belongs to at least one path. The reason that we consider paths starting in the initial state is that an active reset signal should cause a transition to the initial state. For each path, one needs to compute input values that will cause the control to traverse the path. For the purpose of testing, it is often useful to add output signals to the design of the control that indicate the current state.
A test that should be performed even before the paths are tested is to check if indeed the reset signal initializes the control, and if the step enable signal causes a a transition to the "fetch" state. The technique of performing simulation with a given sequence of inputs and the expected output sequence is called *test vectors*. Simulation with test vectors reports mismatches between the expected output values and the one computed by the simulator. VHDL files may contain testing procedures. However, it is not that simple to use VHDL testing features, so you may conduct your testing using the Xilinx Foundation simulator.

6.6.2 Testing RTL instructions

We consider an example of RTL testing. Suppose we wish to test if an instruction fetch functions properly, namely: `IR:=M(PC)`. We suggest the following test procedure for it.

1. Load a short program of load and store instructions starting at memory address $0x00800000$.

2. Use single-step mode and generate a reset. Check if: (a) the state is "Init"; and (b) `PC=0` (recall that the actual fetch in the LS machine is from address `PC`$+0x00800000$).

3. Perform a single step and check the contents of the `IR`. The first part is completed if the `IR` contents are correct

4. Check the signals related to a memory access: `mr,mw, busy, Ao[0:7], Di[26:31], as, doe, wr` and the current state of the controls (LS control and memory access control). If you are using the Logic Analyzer, note that 32 bits can be stored in each clock cycle, and in this example we sample roughly 24 bits, so a few more signals can be sampled if required. The second part of the test is completed if the sampled signals are correct and if the expected transitions occur at the right clock periods.

6.7 The contents of the main memory

An example of the ABEL file that determines the initial contents of the main memory is depicted below.

```
module read16
Title 'read16'
..................
```

```
..................
"  <<add your ASM Program here>>
REG0=^h8c01000d; //adr=0x800000
REG1=^h8c02000e; //adr=0x800001
REG2=^h8c03000f; //adr=0x800002
REG3=^hac01000a; //adr=0x800003
REG4=^hac02000b; //adr=0x800004
REG5=^hac03000c; //adr=0x800005
REG6=^hac03000a; //adr=0x800006
REG7=^hac02000b; //adr=0x800007
REG8=^hac01000c; //adr=0x800008
REG9=^h00000000; //adr=0x800009
REGa=^haaaaaaaa; //adr=0x80000a
REGb=^hbbbbbbbb; //adr=0x80000b
REGc=^hcccccccc; //adr=0x80000c
REGd=^h00000001; //adr=0x80000d
REGe=^h01234567; //adr=0x80000e
REGf=^hfedcba98; //adr=0x80000f
.................
.................

end read16
```

6.8 The DLX Assembly Language

Programs for the DLX need not be written in machine code. To simplify the task of programming the DLX an assembly language was written as well as an assembler program (this is a program that translates assembly programs to machine code programs).
The assembler is called `DLXASS.exe`. We use the following naming conventions: The suffix of an assembly file is `.s` and the suffix of a machine code file is `.cod`. If requested, the assembler program also outputs a “dump” of the machine code file with the suffix `.lst`.
An example of an assembly program is depicted below:

```
* Example file for the load/store machine

          pc= 0x00800000        * Address of program in main memory

          lw R1 R0 data1        * R1=M(data1)=0x00000001
          lw R2 R0 data2        * R2=M(data2)=0x01234567
          lw R3 R0 data3        * R3=M(data3)=0xfedcba98
          lw R4 R0 data4        * R4=M(data4)=0xaffeaffe

          sw R1 R0 adr1         * M(adr1)=R1
          sw R2 R0 adr2         * M(adr2)=R2
```

```
            sw R3 R0 adr3          * M(adr3)=R3
            sw R4 R0 adr4          * M(adr4)=R4

            sw R4 R0 adr1          * M(adr1)=R4
            sw R3 R0 adr2          * M(adr2)=R3
            sw R2 R0 adr3          * M(adr3)=R2
            sw R1 R0 adr4          * M(adr4)=R1

            dc 0x00000000          * This is an illegal instruction!


* Here the data of this program starts

adr1:   ds 1                       * First memory address
adr2:   ds 1                       * Second memory address
adr3:   ds 1                       * Third memory address
adr4:   ds 1                       * Fourth memory address

data1:  dc 0x00000001              * First data value
data2:  dc 0x01234567              * Second data value
data3:  dc 0xfedcba98              * Third data value
data4:  dc 0xaffeaffe              * Fourth data value
```

To generate the "dump" file type: `dlxass -d ls_ex.lst ls_ex.s ls_ex.cod`. Below is a listing of a dump file.

```
0x00000000: 0x8C010011              LW R1 R0 data1
0x00000001: 0x8C020012              LW R2 R0 data2
0x00000002: 0x8C030013              LW R3 R0 data3
0x00000003: 0x8C040014              LW R4 R0 data4
0x00000004: 0xAC01000D              SW R1 R0 adr1
0x00000005: 0xAC02000E              SW R2 R0 adr2
0x00000006: 0xAC03000F              SW R3 R0 adr3
0x00000007: 0xAC040010              SW R4 R0 adr4
0x00000008: 0xAC04000D              SW R4 R0 adr1
0x00000009: 0xAC03000E              SW R3 R0 adr2
0x0000000A: 0xAC02000F              SW R2 R0 adr3
0x0000000B: 0xAC010010              SW R1 R0 adr4
0x0000000C: 0x00000000             DC 0x00000000

0x0000000D:                  adr1: DS 0x00000001
0x0000000E:                  adr2: DS 0x00000001
0x0000000F:                  adr3: DS 0x00000001
0x00000010:                  adr4: DS 0x00000001
0x00000011: 0x00000001      data1: DC 0x00000001
```

```
0x00000012: 0x01234567      data2: DC 0x01234567
0x00000013: 0xFEDCBA98      data3: DC 0xFEDCBA98
0x00000014: 0xAFFEAFFE      data4: DC 0xAFFEAFFE
```

The assembly language has the following rules:

1. Everything that appears in a line after a "*" is a comment.
2. The line `pc= 0x12345` means that next translated line will be mapped to address `0x12345`.
3. A label is defined by placing its name at the beginning of a line followed by a ":". (in the example: `adr1:`). A label is a shortcut for an address. The address that corresponds to a label is the address of the line in which it is defined. Every occurrence of a label (even before the place in which the label is defined) is replaced with the address that corresponds to the label.
4. `dc` x means "place the constant x in the address that corresponds to the current line".
5. `ds` y means "reserve y words in memory". The address of the next line is y plus the address of the current line.
6. Instructions are given the mnemonics used in the description of the instruction sets.

Chapter 7

A simplified DLX

In this handout we describe a simplified DLX-Architecture which you will be implementing on the RESA-CPU. It is assumed that the reader is familiar the following topics: (a) The DLX architecture and implementation that was taught in the course "Introduction to Digital Computers" as described in the book by Müller and Paul. The architecture described here is a simplified version. (b) The RESA bus and memory accesses. In particular, instruction fetch and execution of load/store instructions are done by initiating bus transactions.

7.1 General Architecture

7.1.1 Comparison between the DLX and the simplified DLX

The simplified DLX-Architecture differs from the DLX-Architecture in a few ways:

1. There are only two instruction formats: I-Type and R-Type. The J-Type-Format is canceled and unified into the I-Type by using the 16-bit immediate constant in the I-Type-Format.

2. The memory is word addressable which means that only words (32 bits) can be read from and written to the main memory. This means that successive word addresses in memory have successive addresses (instead of increments by 4 as in the DLX which is byte-addressable). The fact that the simplified DLX is word accessible simplifies load/store instruction execution because it is not necessary to shift the data anymore.

3. The instruction set is reduced.

7.1.2 Architectural Registers and Modules

The architectural registers of the simplified DLX are all 32 bits wide and listed below.

- 32 General Purpose Registers (GPR): `R0` to `R31`. Note that `R0` always holds the value 0;
- Program Counter (PC);
- Instruction Register (IR); and

- Special Registers: `MAR, MDR, A, B` and `C`;

The main modules in the datapath of the simplified DLX are as follows:

- The GPR-File is a dual-port RAM which supports either two reads or one write.
- The ALU supports 2's complement integer addition, subtraction, comparison and bitwise logical operations.
- The Shifter supports logical left and right shifts by one position.

7.1.3 Instruction Formats

There are two instruction formats:

1. An instruction in the I-Type-Format is divided into four fields depicted below.

6	5	5	16
Opcode	*RS1*	*RD*	*immediate*

2. An instruction in the R-Type-Format is divided into five fields depicted below. Note that the 5 bit field used in the DLX-Architecture for the shift amount (SA) is not used in the simplified DLX.

6	5	5	5	5	6
Opcode	*RS1*	*RS2*	*RD*		*Function*

7.1.4 Instruction Set

We list below the instruction set of the simplified DLX. Note that *imm* denotes the value of the immediate field in an I-Type-Instruction. *sext(imm)* denotes the 2's complement sign extension of *imm* to 32 bits.

- Load/Store-instructions:

Load/Store	Semantics
`lw    RD RS1 imm`	RD := M(sext(imm)+RS1)
`sw    RD RS1 imm`	M(sext(imm)+RS1) := RD

- Immediate–instructions:

Instruction	Semantics
`addi  RD RS1 imm`	RD := RS1 + sext(imm)

- Shift–/Compute–Instructions:

Instruction	Semantics
`slli RD RS1`	RD := RS1 << 1
`srli RD RS1`	RD := RS1 >> 1
`add RD RS1 RS2`	RD := RS1 $+$ RS2
`sub RD RS1 RS2`	RD := RS1 $-$ RS2
`and RD RS1 RS2`	RD := RS1 $\wedge$ RS2
`or RD RS1 RS2`	RD := RS1 $\vee$ RS2
`xor RD RS1 RS2`	RD := RS1 $\oplus$ RS2

- Test–Instructions:

Instruction	Semantics
s*rel*i RD RS1 imm	RD := 1, if condition is satisfied, RD := 0 otherwise
if *rel* $=$`lt`	test if RS1 $<$ sext(imm)
if *rel* $=$`eq`	test if RS1 $=$ sext(imm)
if *rel* $=$`gt`	test if RS1 $>$ sext(imm)
if *rel* $=$`le`	test if RS1 $\leq$ sext(imm)
if *rel* $=$`ge`	test if RS1 $\geq$ sext(imm)
if *rel* $=$`ne`	test if RS1 $\neq$ sext(imm)

- Jump–instructions:

Instruction	Semantics
`beqz RS1 imm`	PC = PC + 1 + sext(imm), if RS1 = 0
	PC = PC + 1, if RS1 $\neq$ 0
`bnez RS1 imm`	PC = PC + 1, if RS1 = 0
	PC = PC + 1 + sext(imm), if RS1 $\neq$ 0
`jr RS1`	PC = RS1
`jalr RS1`	R31 = PC+1; PC = RS1

- Miscellaneous–instructions:

Instruction	Semantics
`special-nop`	causes transition to Init/Fetch states
`halt`	causes transition to HALT state

Encoding of the Instruction Set

Tables 7.1 and 7.2 specify the binary encoding of the instructions.

7.2 Implementation

7.2.1 Datapath

The datapath of the simplified DLX is depicted in Figure 7.1. The proposed datapath is not a typical datapath; it lacks busses and drivers. There are two reasons for this: (a) the interface with the I/O Control Logic does not requires busses; and (b) to protect the hardware we do no allow

IR[31 : 26]	Mnemonic	Semantics
Data Transfer		
100 011	lw	RD = M(sext(imm)+RS1)
101 011	sw	M(sext(imm)+RS1) = RD
Arithmetic, Logical Operation		
001 011	addi	RD = RS1 + sext(imm)
Test Set Operation		
011 rel	s rel i	RD = (RS1 rel sext(imm))
011 001	sgti	RD = (RS1 $>$ sext(imm))
011 010	seqi	RD = (RS1 $=$ sext(imm))
011 011	sgei	RD = (RS1 $\geq$ sext(imm))
011 100	slti	RD = (RS1 $<$ sext(imm))
011 101	snei	RD = (RS1 $\neq$ sext(imm))
011 110	slei	RD = (RS1 $\leq$ sext(imm))
Control Operation		
000 100	beqz	PC = PC + 1 + (RS1 $= 0$? sext(imm) : 0)
000 101	bnez	PC = PC + 1 + (RS1 $\neq 0$? sext(imm) : 0)
010 110	jr	PC = RS1
010 111	jalr	R31 = PC + 1; PC = RS1
Miscellaneous Instructions		
110 000		no operation (not supported by DLX assembler)
111 111		stop program (not supported by DLX assembler)

Table 7.1: I-type Instructions

using drivers. Note that control signals as well as some connections between the environments are not depicted in Fig. 7.1.

ALU environment

Even though the 32-bit adder in the ALU can be designed using VHDL, we suggest that you use three 16-bit adder/subtractors from the Xilinx library (ADSU16) to build a 32-bit conditional sum adder. The Xilinx library component is optimized with respect to the FPGA's technology and the usage of the adder/subtractor eliminates the need to invert the second argument in a subtraction instruction.

It is very likely that the ALU will lie on the critical path. A VHDL design will probably end up much slower and costlier. Moreover, a costly design might create additional delay due to wire and routing delays.

The ALU supports bitwise logical instructions and comparison instructions. A comparison is implemented by a subtraction followed by examining the sign of the result and whether the result equals zero. More implementation details appear in the book of Müller and Paul.

IR$[5:0]$	Mnemonic	Semantics
Shift Operation		
000 000	slli	RD = RS1 $\ll 1$
000 010	srli	RD = RS1 $\gg 1$
Arithmetic, Logical Operation		
100 011	add	RD = RS1 $+$ RS2
100 010	sub	RD = RS1 $-$ RS2
100 110	and	RD = RS1 $\wedge$ RS2
100 101	or	RD = RS1 $\vee$ RS2
100 100	xor	RD = RS1 $\oplus$ RS2

Table 7.2: R-type Instructions (in R-type instructions IR$[31:26] = 0^6$)

Shifter environment

The shifter is a 32-bit left/right logical shifter. This means that a zero is pushed in from the right (left) in case of a left (right) shift. The control inputs of the shifter are: shift and right. The shift input indicates whether a shift should take place (otherwise the output equals the input). The right signal indicates whether the shift is a right shift. (Why do we want the shifter to support also the identity function?)

The GPR environment

The GPR environment is identical to that of the Load/Store Machine.

7.2.2 Control

Figure 7.2 depicts the state diagram of the control of the simplified DLX. Note that the transitions leading to the Init State and Fetch State are depicted as one "edge" although in reality these are two separate transitions. Access to the memory is done via the Memory Access Control module as described for the Load/Store Machine. Note that Figure 7.2 does not depict the `reset` signal generated by the I/O control logic. The `reset` signal causes a transition in the control of the DLX to "init" state.
The control signals that appear in Fig. 7.2 have the following meanings:

1. `step_en` is the `step_en` signal from from the I/O control logic.
2. `busy` is the `busy` signal from the memory access control.
3. `D2..D13` are the monomials corresponding to the decoding of the instructions.
4. `else` corresponds to an illegal instruction, namely, when all the monomials `D2..D13` are not satisfied.
5. `bt` (branch taken) corresponds to the event that the condition of a conditional branch is satisfied.

The following control signals are used to communicate between the datapath and the control:

1. `IRCE, PCCE, ACE, BCE, CCE, MARCE, MDRCE`: clock enable signals of the registers.
2. `S1SEL0, S1SEL1, S2SEL0, S2SEL1, DINTSEL, MDRSEL, ASEL`: select signals of the muxes.
3. `ADD, TEST, SHIFT, RIGHT, ALUF0, ALUF1, ALUF2`: signals that control the functionality of the ALU.
4. `ITYPE`: active when the current instruction is an i-type instruction.
5. `JLINK`: active during a "jalr" instruction.

You may want to add some control signals that are active during the "init" and "halt" states. These control signals can be used by the Monitor Slave (i.e. Logic Analyzer).
Table 7.3 lists the active control signals in each state. (An RTL instruction describes the action that takes place during the state).
Table 7.4 lists the control signals of the simplified DLX.
Table 7.5 lists the monomials computed by the control. These monomials are correct under the assumption that all instructions are legal. We henceforth refer to decoding under the assumption that all the instructions are legal as *imprecise decoding*.
For example, to simplify the decoding of a transition to the ALU state, we suggest to check if $IR[31:28] = 0^4$ and $IR[5] = 0$. That means that if $IR[31:28] = 0^4$, the bits $IR[27:26]$ are ignored. If they happen not to be equal to zero, then the instruction is an illegal instruction. Nevertheless, in this case we choose to treat it as an ALU instruction. Now, one might ask what about the bits of $IR[4:0]$? Suppose we decode an instruction as an ALU instruction, but the bits in $IR[4:0]$ do not encode a legal instruction? We do not wish to add a transition from the ALU state to the HALT state, so some ALU instruction must take place in this case.
The advantage of imprecise decoding is that it is easier, and therefore, faster. The justification for using imprecise decoding is that legal programs are decoded correctly by imprecise decoding and that it is very easy to write a program that checks if a legal program is stored in a given file (or memory segment). Note that if imprecise decoding is used, then a illegal instruction is decoded as a legal one.
It is possible to almost perfectly combine the advantages of precise decoding and imprecise decoding by performing imprecise decoding. Every instruction execution has a "commit" stage (namely, write-back, store, or changing of the PC in a jump). One could condition the commit stage of execution upon the legality of the instruction. The legality of the instruction is computed while the instruction is executed (not in the decode stage), and so time is not wasted on checking the legality of legal instructions. We will not consider this more sophisticated option, and use only imprecise decoding.
The conclusion is that we suggest to do an imprecise decoding of the instructions. That mean that some illegal instructions are considered as legal encodings of other instructions. We leave it to you to choose which instructions are decoded in such cases.
The usage of imprecise decoding means that we cannot reach the Halt state using an illegal instruction. We therefore, add a HALT instruction that causes a transition to the Halt state.

Name	RTL Instruction	Active Control Signals
Fetch	$IR = M(PC)$	MR, IRce
Decode	$A = RS1$, $B = RS2$ $PC = PC + 1$	Ace, Bce, S2sel[1], S2sel[0] PCce, add
Alu	$C = A$ op B	S1sel[0], Cce
TestI	$C = (A$ rel $imm)$	S1sel[0], S2sel[0], Cce, test, Itype
AluI(add)	$C = A + imm$	S1sel[0], S2sel[0], Cce, add, Itype
ShiftI	$C = A$ shift sa $sa = 1, (-1)$	S1sel[0], Cce, Itype DINTsel, shift (,right)
Adr.Comp	$MAR = A + imm$	S1sel[0], S2sel[0], MARce, add
Load	$MDR = M(MAR)$	MDRce, Asel, MR, MDRsel
Store	$M(MAR) = MDR$	Asel, MW
CopyMDR2C	$C = MDR(\gg 0)$	S1sel[0], S1sel[1], S2sel[1], DINTsel, Cce
CopyGPR2MDR	$MDR = B(\ll 0)$	S1sel[1], S2sel[1], DINTsel, MDRce
WBR	$RD = C$ (R-type)	GPR_WE
WBI	$RD = C$ (I-type)	GPR_WE, Itype
Branch	branch taken?	
Btaken	$PC = PC + imm$	S2sel[0], add, PCce
JR	$PC = A$	S1sel[0], S2sel[1], add, PCce
Save PC	$C = PC$	S2sel[1], add, Cce
JALR	$PC = A$ $R31 = PC$	S1sel[0], S2sel[1], add, PCce GPR_WE, jlink

Table 7.3: The active control signals in each state

Signal	Value	Semantics
ALUf[2:0]		Controls the functionality of ALU
Rce		Register clock enable
S1sel[1:0]	00 01 10 11	PC A B MDR
S2sel[1:0]	00 01 10 11	B IR 0 1
DINTsel	0 1	ALU Shifter
MDRsel	0 1	DINT DI
Asel	0 1	PC MAR
shift		explicit Shift-Instruction
right		Shift to the right
add		Forces an addition
test		Forces a test (in the ALU)
MR		Memory Read
MW		Memory Write
GPR_WE		GPR write enable
itype		Itype-Instruction
jlink		jump and link

Table 7.4: List of control signals

Nontrivial DNF	Target State	IR[31 : 26]	IR[5 : 0]
D1	Init/Fetch	110***	******
D2	Alu	0000**	1*****
D4	ShiftI	0000**	0*****
D5	AluI	001***	******
D6	TestI	011***	******
D7	Adr.Comp	10****	******
D8	JR	010**0	******
D9	JALR	010**1	******
D12	Branch	0001**	******
D13	Copy GPR2MDR	**1***	******
/D13	Load	**0***	******
bt	Btaken	AEQZ $\oplus$IR[26]	
/bt	Fetch	/(AEQZ $\oplus$IR[26])	

Table 7.5: Monomials of the control

7.3 The RESA environment

7.3.1 Special DLX assembly instructions

The DLX assembly language includes for convenience two additional instructions which are aliases for DLX instructions.

1. MOVE. A move instruction has two arguments (e.g. `move RD RS1`), and is an alias for `addi RD RS1 0`.

2. NOP. A no-operation instruction has no arguments and is an alias for `addi R0 R0`. Note that the Special-NOP instruction is not supported by the DLX assembler. The execution of NOP and Special-NOP are different: one goes through the ALU state and the other transitions immediately to the Fetch or Init States. A Special-NOP instruction can be used by the "dc" directive.

7.3.2 Memory map

Your DLX design should work in the RESA environment. We have already described the RESA bus, the usage of the I/O Control Logic, and the RESACTRL Program. We now elaborate on the memory map of the RESA.
Recall that the bus protocol is address based and that every "item" has a 32 bit address (for example, the status register in your Logic Analyzer has a 32 bit address). You control (and are aware of) only the 8 least significant bits of this address. This means that there is a "map" that specifies which addresses are allocated to which boards. Within the memory board (which hosts the main memory of your DLX design) not all addresses may be used by you. We outline the memory map of the RESA below.

1. 0x00000000 - 0x0000ffff : 64KW of ROM. This ROM stores a test program for the DLX design. We elaborate more on this program in Section 7.3.3.

2. 0x00010000 - 0x007fffff : This address space is not allocated to any device.

3. 0x00800000 - 0x008fffff: 1MW of RAM. This space is used for the user programs.

4. Some of the other address space is allocated to various devices (CPU board, monitor slave, etc.). *Do not attempt to access addresses in this space since it might damage the RESA*.

7.3.3 The Test Program

The test program consists of nine test procedures. The source of the test program is posted in the lab's homepage. An outline is depicted below.

```
*-----------------------------------------------------------------*
* Outline of DLX test program
*-----------------------------------------------------------------*
        pc=0x00000000
test0:  ...

RESA:   dc 0x52455341           * "RESA"
USER:   dc 0x00800001           * start address of user's program
RAM:    dc 0x00800000

test1:  ...
test8:  dc 0xc0000008           * Special-NOP

        lw   R30 R0  RAM        * R30 := address of RAM
        lw   R1  R0  RESA
        lw   R2  R30 0x0        * read from M[0x800000]
        xor  R2  R2  R1         * if (M[0x800000]=="RESA")
        beqz R2  start          *   then jump to user's program
        bnez R2  ramtst         * else perform memory test
        ...

start:  lw   R30 R0 RAM         * load start address of RAM
        lw   R1  R0 USER        * load start address of user's program
        jalr R1                 * jump to user's program

end:    beqz R0  end
```

The first 8 tests are not "destructive" (i.e. they do not write to the user's memory space). After 8 tests are executed by test program, the 9th test reads and writes to the whole user space to test the memory (which is something one would not like to happen if a user's program is stored in that area). Therefore, before the 9th test starts, it is possible to jump to the user's program instead of running the 9th test. This is done as follows: The value stored in the address 0x00800000 is compared with a special value (given the name "RESA" in the listing above). If the values are equal, then the 9th test is skipped and a jump to address 0x00800001 is performed. Otherwise, the 9th test takes place.

7.3.4 Executing a user's program after the test program

If you would like to start running your program only after the first 8 tests are executed, you can use the following mechanism.

```
*-------------------------------
* having your program run after the first 8 tests
*-------------------------------
        pc= 0x800000            * beginning of user’s memory space

        dc 0x52455341           * "RESA"
        beqz R0 start           * first instruction

a:      dc 10                   * 0x800002
b:      dc 20                   * 0x800003
erg:    ds 1

start: lw   R1  R30  a
        lw   R2  R30  b
        add  R1  R1   R2
        sw   R1  R30  erg       * erg = a + b

end:   beqz R0  end
```

Note that (a) In address 0x00800000 the constant corresponding to the ”RESA” special value is stored. (b) The jump to address 0x00800001 is implemented in the test program by a jalr instruction (it could be also implemented by a jr instruction). The register R1 is set to hold the value 0x00800001 (which can not be stored by a 16-bit immediate constant) using a variable that is initialized to this value in the test program. (c) The register R30 is set to hold the value 0x00800000 also by using a variable that is initialized to this value in the test program. The R30 register can then be used by the user’s program as the offset to compute addresses so that the user’s program variables are accessible (e.g. load and store instructions).
Note also that a failure in one of the 8 tests (due to an error in the DLX’s functionality) might cause an endless loop, in which case the user’s program is not executed at all.

7.3.5 Executing a user’s program without the test program

We avoid running the test program by mapping the logical address 0x00000000 to the physical address 0x00800000, as described in the section on translating addresses.

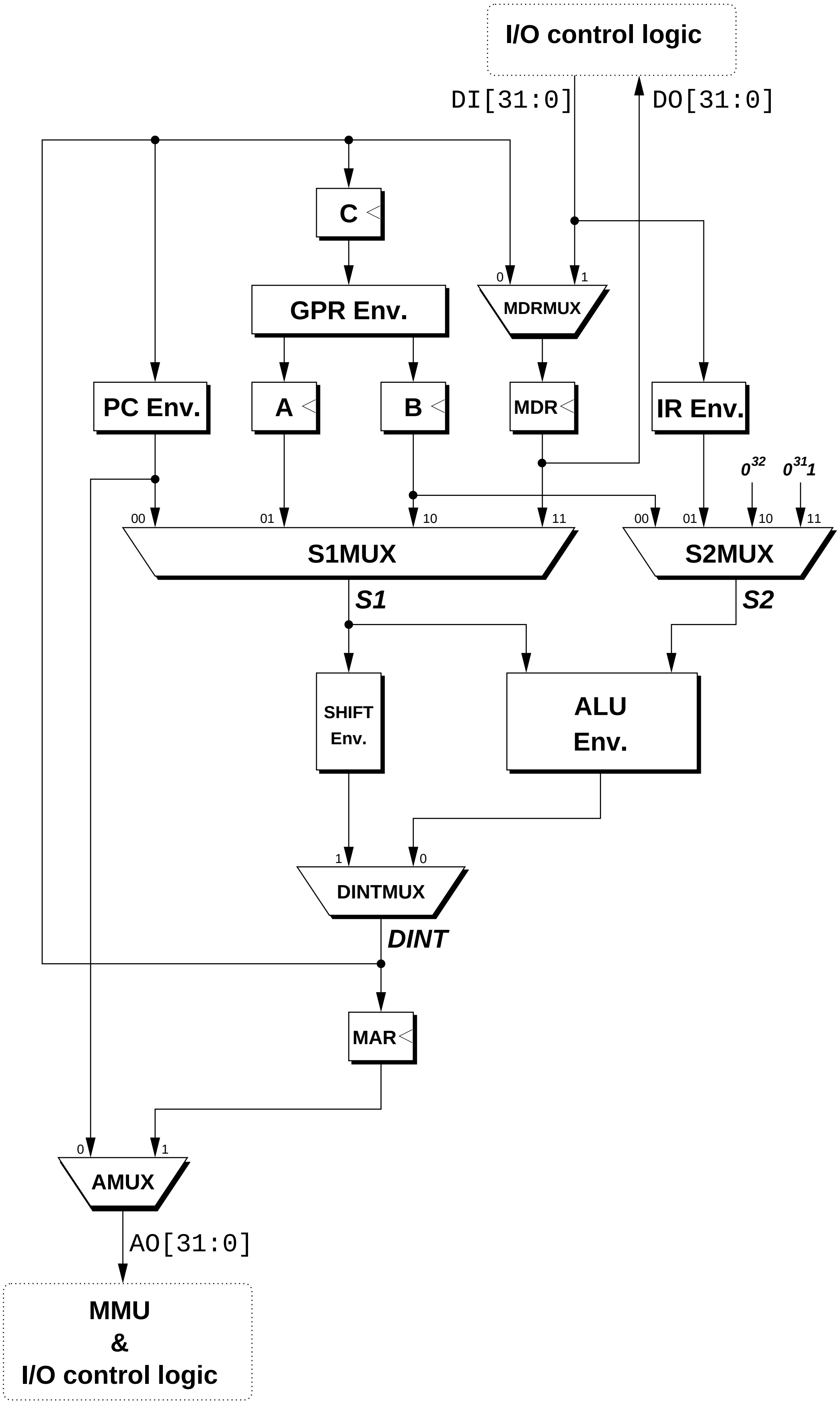


Figure 7.1: Datapath of the simplified DLX machine

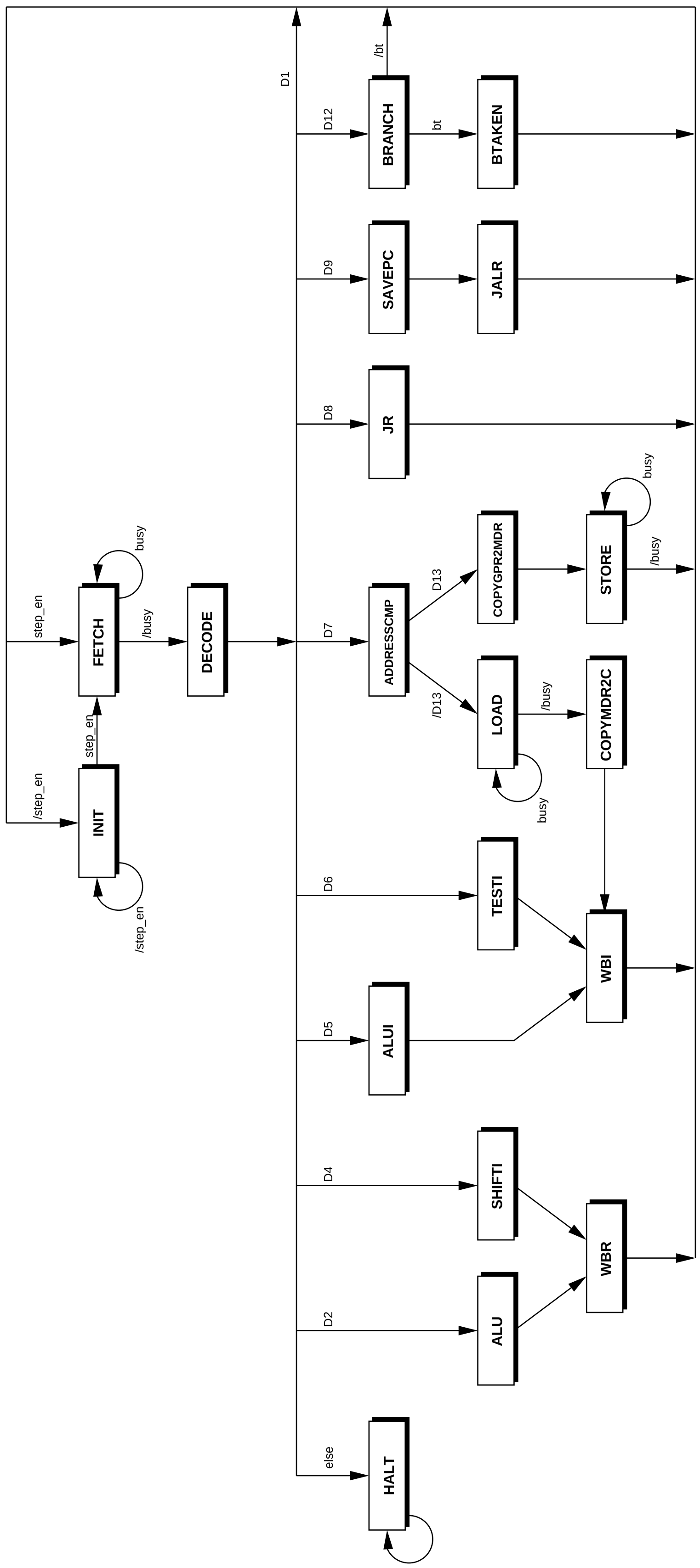


Figure 7.2: Finite state diagram of the control of the simplified DLX machine