Leibniz Universität Hannover
Institut für Mikroelektronische Systeme
Prof. Dr.-Ing. H. Blume

VHDL Implementation, Verification and Logic Synthesis of Memory Bus Arbiters for Multi-Processor System

Master Thesis

von

Pedro Pascual Sánchez López

December 2009

Leibniz Universität Hannover
Institut für Mikroelektronische Systeme
Prof. Dr.-Ing. H. Blume

VHDL Implementation, Verification and Logic Synthesis of Memory Bus Arbiters for Multi-Processor System

Master Thesis

von

Pedro Pascual Sánchez López

Betreuer: Dipl.-Ing. K. Septinus

Erstprüfer: Prof. Dr.-Ing. P. Pirsch
Zweitprüfer: Prof. Dr.-Ing. H. Blume

Ich versichere, dass ich die vorgelegte Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen, Hilfen und Hilfsmittel benutzt habe.

Hannover, den 02.12.2009

Contents

List of Figures

List of Tables

1 Introduction

Multi-core technology is a reality of today. The era of the single processor system has passed; multi-core is real as applications can no longer count on increased processor clock speeds to improve performance. Historically, the majority of applications are based on a single-threaded architecture, and dependent on clock speeds to increase performance. They gain no linear performance improvements when run on multi-core systems if they have not been designed to take advantage of the increased multiple compute engines available on the chips. If the multi-core aspect of systems is not taken into consideration, today's applications may run slower than even with more compute power available in the system.

With the focus on hardware, and increasing processor counts, there is increasing need to understand the new complexities of application design, debug, and optimization in multi-core systems. In order to take advantage of the additional processing power that multi-core systems offer, new development tools are needed that allow the applications to change as well. Exploiting the power of multi-core processors will be critical for customers to improve their business success.

Multicore architectures are everywhere and can be found in all market segments. There is great interest in the scientific community in using Multicore systems efficiently, but without the great effort of reprogramming existing sequential codes.

Task of this thesis is to study how different scenarios behave with different kind of arbiters which are their advantages and disadvantages. The development stats by having a closer look to handshaking, atomic operations and what they are used for. Using the previous knowledge the I/O control strategies are derived and afterwards implemented. The implementation of the arbiters and the environments is then finalized and tested. The work finishes up by taking a look to performance-considerations as well as giving an outlook for future developments.

2 Multicore technology

Multi-core technology is the term that describes today's processors that have two or more working processor chips (more commonly referred to as cores) working simultaneously as one system. Dual cores or chips with two processors that work as one system are the first type of multi-core technology applications.

The multi-core processor technology was conceptualized and has revolved around the idea of being able to make parallel computing possible. Parallel computing could dramatically increase the speed, efficiency and performance of computers by simply putting 2 or more Central Processing Units (or CPU) in only one chip. This would ultimately minimize the power and heat consumption of the system while still being able to greatly boost system performance without sacrificing energy consumption limits. This would give more performance with less or with the same amount of energy.

The multi-core technology would also enable users to do more tasks at the same time. Since more computing workloads could be done at the same time, manufacturers such as Intel and AMD could focus more on increasing computing and processing performance without increasing clock speeds and thus avoid the need for consuming more energy.

Multi-core processors work at their full potential if they are used with multi-threaded programs or software. Multithreaded software could include applications and most importantly operating systems that have the ability to split tasks and commands into a set of separate workloads that could then be processed and run simultaneously on each of the cores present. This means more work is done in less time.

3 Handshaking

The most significant gesture in business and in life is a handshake. In many cultures it is the unspoken message that accompanies our words. But in other aspects as, information technology, telecommunications, and related fields, handshaking is an automated process of negotiation that dynamically sets parameters of a communications channel established between two entities before normal communication over the channel begins. It follows the physical establishment of the channel and precedes normal information transfer.

It is usually a process that takes place when a computer is about to communicate with a foreign device to establish rules for communication.

When a computer communicates with another device like a modem or a printer it needs to handshake with it to establish a connection.

Handshaking makes it possible to connect relatively heterogeneous systems or equipment over a communication channel without the need for human intervention to set parameters. One classic example of handshaking is that of modems, which typically negotiate communication parameters for a brief period when a connection is first established, and thereafter use those parameters to provide optimal information transfer over the channel as a function of its quality and capacity. The "squealing" (which is actually a sound that changes in pitch 100 times every second) noises made by some modems with speaker output immediately after a connection is established are in fact the sounds of modems at both ends engaging in a handshaking procedure; once the procedure is completed, the speaker might be silenced, depending on the settings of operating system or the application controlling the modem.

Because of the different speeds and data requirements of I/O cores, different I/O strategies may be useful, depending on the type of I/O core which is connected to the arbiter. Because the I/O cores are not synchronized with the arbiter, some information must be exchanged between the arbiter and the core to ensure that the data is received reliably. For a complete ``handshake," four events are important:

The device providing the data (the talker) must indicate that valid data is now available.

The device accepting the data (the listener) must indicate that it has accepted the data. This signal informs the talker that it need not maintain this data word on the data bus any longer.

The talker indicates that the data on the bus is no longer valid, and removes the data from the bus. The talker may then set up new data on the data bus.

The listener indicates that it is not now accepting any data on the data bus. the listener may use data previously accepted during this time, while it is waiting for more data to become valid on the bus.

Note that each of the cores and arbiter supplies two signals. The core supplies a signal (say, data valid, or REQUEST) at step (1). It supplies another signal (say, data not valid,

or NOT REQUEST) at step (3). Both these signals can be coded as a single binary value (REQUEST) which takes the value 1 at step (1) and 0 at step (3). The listener supplies a signal (say, data accepted, or STATUS) at step (2). It supplies a signal (say, data not now accepted, or NOT STATUS) at step (4). It, too, can be coded as a single binary variable, STATUS. Because only two binary variables are required, the handshaking information can be communicated over two wires, and the form of handshaking described above is called a two wire Handshake. Other forms of handshaking are used in more complex situations; for example, where there may be more than one controller on the bus, or where the communication is among several devices. Figure 3.1shows a timing diagram for the signals DAV and DAC which identifies the timing of the four events described previously.

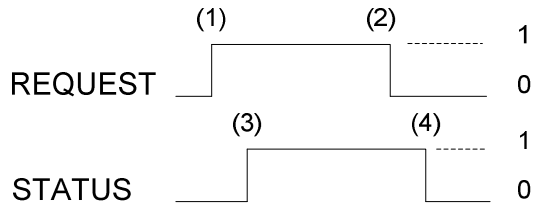

Figure 3.1: Timing diagram for two-wire handshake

Either the arbiter or the I/O core can act as the talker or the listener. In fact, the arbiter may act as a talker at one time and a listener at another.

4 Atomic operations

An atomic operation in computer science refers to a set of operations that can be combined so that they appear to the rest of the system to be a single operation with only two possible outcomes: success or failure.

The conditions when operations can be considered atomic when two conditions are met these conditions are: Until the entire set of operations completes, no other process can know about the changes being made (invisibility) and the second one; if any of the operations fail then the entire set of operations fails, and the state of the system is restored to the state it was in before any of the operations began.

Even without the complications of multiple processing units, this can be non-trivial to implement. As long as there is the possibility of a change in the flow of control, without atomicity there is the possibility that the system can enter an invalid state (invalid as defined by the program, a so-called invariant).

For example, imagine a single process is running on a computer incrementing a value in a given memory location. To increment the value in that memory location follows the next steps: Firstly, process reads the value in the memory location. Secondly, the process adds one to the value. And finally, the process writes the new value back into the memory location.

In the second example, imagine two processes are running incrementing a single, shared memory location. So, the first process reads the value in memory location and in addition the first process adds one to the value. But before it can write the new value back to the memory location it is suspended, and the second process is allowed to run. In the other hand, the second process reads the value in memory location, the same value that the first process read and the second process adds one to the value. Hence, the second process writes the new value into the memory location. The second process is suspended and the first process allowed running again. Now, the first process writes a now-wrong value into the memory location, unaware that the other process has already updated the value in the memory location.

This is a trivial example. In a real system, the operations can be more complex and the errors introduced extremely subtle. For example, reading a 64-bit value from memory may actually be implemented as two sequential reads of two 32-bit memory locations. If a process has only read the first 32-bits, and before it reads the second 32-bits the value in memory gets changed, it will have neither the original value nor the new value but a mixed-up garbage value.

Furthermore, the specific order in which the processes run can change the results, making such an error difficult to detect and debug.

4.1 Locking

While an atomic operation is functionally equivalent to a "critical section" (protected by a lock), it requires great care to not suffer significant overhead compared to direct use of atomic operations, with many computer architectures offering dedicated support. To improve program performance, it is therefore often a good idea to replace simple critical sections with atomic operations for non-blocking synchronization, instead of the other way around, but unfortunately a significant improvement is not guaranteed and lock-free algorithms can easily become too complicated to be worth the effort.

5 I/O control strategies

Several I/O strategies are used between the computer system and I/O devices, depending on the relative speeds of the computer system and the I/O devices. The simplest strategy is to use the processor itself as the I/O controller, and to require that the device follow a strict order of events under direct program control, with the processor waiting for the I/O device at each step.

Another strategy is to allow the processor to be ``interrupted" by the I/O devices, and to have a (possibly different) ``interrupt handling routine" for each device. This allows for more flexible scheduling of I/O events, as well as more efficient use of the processor. (Interrupt handling is an important component of the operating system.)

A third general I/O strategy is to allow the I/O device, or the controller for the device, access to the main memory. The device would write a block of information in main memory, without intervention from the CPU, and then inform the CPU in some way that that block of memory had been overwritten or read. This might be done by leaving a message in memory, or by interrupting the processor. (This is generally the I/O strategy used by the highest speed devices -- hard disks and the video controller.)

Direct memory access

In most mini- and mainframe computer systems, a great deal of input and output occurs between the disk system and the processor. It would be very inefficient to perform these operations directly through the processor; it is much more efficient if such devices, which can transfer data at a very high rate, place the data directly into the memory, or take the data directly from the processor without direct intervention from the processor. I/O performed in this way is usually called direct memory access, or DMA. The controller for a device employing DMA must have the capability of generating address signals for the memory, as well as all of the memory control signals. The processor informs the DMA controller that data is available (or is to be placed into) a block of memory locations starting at a certain address in memory. The controller is also informed of the length of the data block.

There are two possibilities for the timing of the data transfer from the DMA controller to memory:

The controller can cause the processor to halt if it attempts to access data in the same bank of memory into which the controller is writing. This is the fastest option for the I/O device, but may cause the processor to run more slowly because the processor may have to wait until a full block of data is transferred.

The controller can access memory in memory cycles which are not used by the particular bank of memory into which the DMA controller is writing data. This approach, called ``cycle stealing," is perhaps the most commonly used approach. (In a processor with a cache that has a high hit rate this approach may not slow the I/O transfer significantly).

DMA is a sensible approach for devices which have the capability of transferring blocks of data at a very high data rate, in short bursts. It is not worthwhile for slow devices, or for devices which do not provide the processor with large quantities of data. Because the controller for a DMA device is quite sophisticated, the DMA devices themselves are usually quite sophisticated (and expensive) compared to other types of I/O devices.

One problem that systems employing several DMA devices have to address is the contention for the single system bus. There must be some method of selecting which device controls the bus (acts as ``bus master") at any given time. There are many ways of addressing the ``bus arbitration" problem; two techniques which are often implemented in processor systems are the following (these are also often used to determine the priorities of other events which may occur simultaneously, like interrupts). They rely on the use of at least two signals (bus_request and bus_grant), used in a manner similar to the two-wire handshake:

Daisy chain arbitration: Here, the requesting device or devices assert the signal bus_request. The bus arbiter returns the bus_grant signal, which passes through each of the devices which can have access to the bus, as shown in Figure 5.1. Here, the priority of a device depends solely on its position in the daisy chain. If two or more devices request the bus at the same time, the highest priority device is granted the bus first, and then the bus_grant signal is passed further down the chain. Generally a third signal (bus_release) is used to indicate to the bus arbiter that the first device has finished its use of the bus. Holding bus_request asserted indicates that another device wants to use the bus.

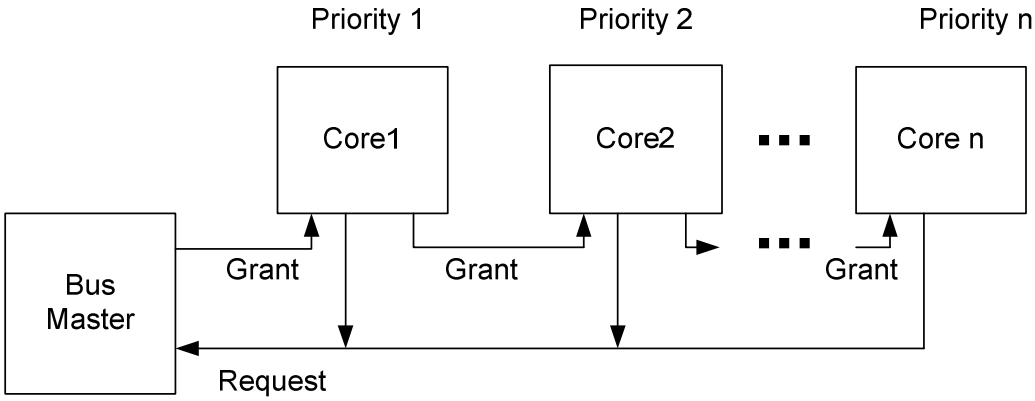

Figure 5.1: Daisy chain bus arbitration

Priority encoded arbitration: Here, each device has a request line connected to a centralized arbiter that determines which device will be granted access to the bus. The order may be fixed by the order of connection (priority encoded), or it may be determined by some algorithm preloaded into the arbiter. Figure 5.2 shows this type of system. Note that each device has a separate line to the bus arbiter. (The bus_grant signals have been omitted for clarity.)

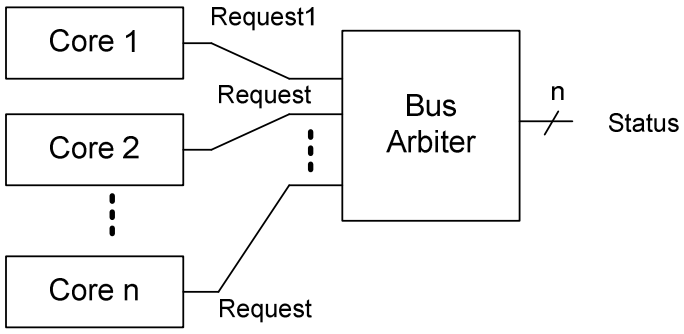

Figure 5.2: Priority encoded bus arbitration

Distributed arbitration by self-selection here, the devices themselves determine which of them has the highest priority. Each device has a bus_request line or lines on which it places a code identifying itself. Each device examines the codes for all the requesting devices, and determines whether or not it is the highest priority requesting device.

These arbitration schemes may also be used in conjunction with each other. For example, a set of similar devices may be daisy chained together, and this set may be an input to a priority encoded scheme.

6 Implementation

The implementation of the simulation environment has been done all with the software Modelsim. ModelSim is a verification and simulation tool for VHDL, Verilog, SystemVerilog, and mixed-language designs.

VHDL is the VHSIC Hardware Description Language. VHSIC is an abbreviation for Very High Speed Integrated Circuit. It can describe the behaviour and structure of electronic systems, but is particularly suited as a language to describe the structure and behaviour of digital electronic hardware designs, such as ASICs and FPGAs as well as conventional digital circuits.

VHDL is a notation, and is precisely and completely defined by the Language Reference Manual (LRM). This sets VHDL apart from other hardware description languages, which are to some extent defined in an ad hoc way by the behaviour of tools that use them. VHDL is an international standard, regulated by the IEEE. The definition of the language is non-proprietary. In addition VHDL can be used to describe hardware at the gate level or in a more abstract way. Successful high level design requires a language, a tool set and a suitable methodology. Moreover, VHDL can be used to describe electronic hardware at many different levels of abstraction. When considering the application of VHDL to FPGA/ASIC design, it is helpful to identify and understand the three levels of abstraction shown opposite - algorithm, register transfer level (RTL), and gate level. Algorithms are unsynthesizable, RTL is the input to synthesis, gate level is the output from synthesis. The difference between these levels of abstraction can be understood in terms of timing.

Algorithm

A pure algorithm consists of a set of instructions that are executed in sequence to perform some task. A pure algorithm has neither a clock nor detailed delays. Some aspects of timing can be inferred from the partial ordering of operations within the algorithm. Some synthesis tools (behavioural synthesis) are available that can take algorithmic VHDL code as input. However, even in the case of such tools, the VHDL input may have to be constrained in some artificial way, perhaps through the presence of an ‘algorithm' clock - operations in the VHDL code can then be synchronized to this clock.

RTL

An RTL description has an explicit clock. All operations are scheduled to occur in specific clock cycles, but there are no detailed delays below the cycle level. Commercially available synthesis tools do allow some freedom in this respect. A single global clock is not required but may be preferred. In addition, retiming is a feature that allows operations to be re-scheduled across clock cycles, though not to the degree permitted in behavioural synthesis tools.

Gates

A gate level description consists of a network of gates and registers instanced from a technology library, which contains technology-specific delay information for each gate.

Basic Simulation Flow
The Figure 6.1 shows the basic steps for simulating a design in ModelSim.

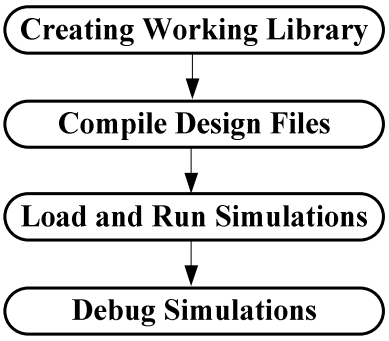

Figure 6.1: Basic Simulation Flow- Overview Lab

In ModelSim, all designs are compiled into a library. Is usual starting a new simulation in ModelSim by creating a working library called "work". "Work" is the library name used by the compiler as the default destination for compiled design units.

Compiling the Design

After creating the working library, is needed to compile the design units into it. The ModelSim library format is compatible across all supported platforms. In addition is possible to simulate the design on any platform without having to recompile it.

Loading the Simulator with the Design and Running the Simulation

With the design compiled, the next step is loading the simulator with your design by invoking the simulator on a top-level module (Verilog) or a configuration or entity/architecture pair (VHDL).As is known in this case will only be used VHDL.

Assuming the design loads successfully, the simulation time is set to zero, and to enter a run command to begin simulation is needed.

Debugging Results

If expected results are not achieved, is possible to use ModelSim’s robust debugging environment in order to track down the cause of the problem.

Project Flow

A project is a collection mechanism for an HDL design under specification or test. The Figure 6.2 shows the basic steps for simulating a design within a ModelSim project.

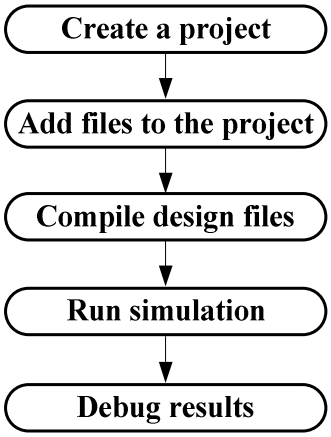

Figure 6.2: Project Flow

As is possible to see, the flow is similar to the basic simulation flow. However, there are two important differences. The first one, is that is not required to create a working library in the project flow; it is done automatically for ModelSim. The second characteristic is that projects are persistent. In other words, they will open every time that ModelSim is invoked (unless closing them is specified).

6.1 Arbiter 1

The first objective of this work has been to design one system which is able to arbiter a group of cores. This group of cores could be variable; therefore the design has to be very compact and easy-scalable code due to the possible changes in the future.

Due that the design is going to be designed in VHDL, it could be necessary to explain the different techniques that can be used as well the advantages and disadvantages of them

It could be possible to create design using mathematics operations and big processes, but as it is known this way implies a lot of recourses as well more physical space and in addition it supposes a bigger energy consume. For this first design the objective is to have a reference for the future arbiters, so for these design will not be take in mind the style to design the circuit.

In order to start the design of this first arbiter is very important to have a very clear idea in the beginning about the design. So, now the only requirement is that the arbiter is able to manage the N cores required where N can change.

In this case, for this arbiter the principle has been, to calculate which core is waiting for more time. And give priority to it. Therefore, the arbiter will count the number of cycles that re signal request is active for each core. A basic idea of the circuit is showed in the Figure 6.3

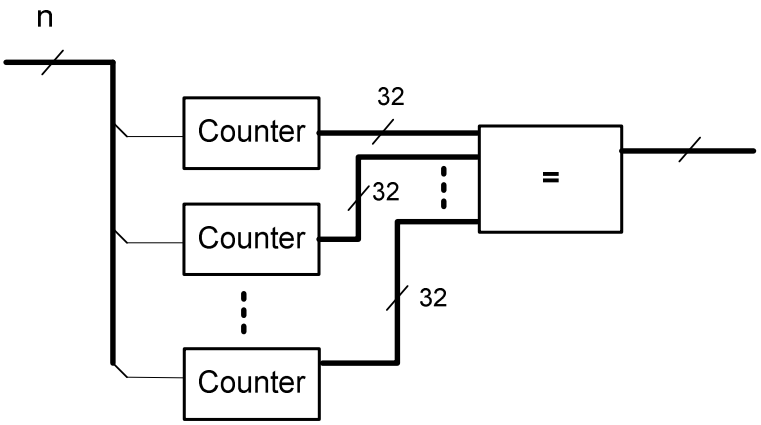

Figure 6.3: Scheme Arbiter 1

Then, the behavior of the design is the next. When the signal request is active, the counter will start to count, in the case that this signal has low value; the counter will reset the intern value. One time here, the requirement is to compare which of the counters has the biggest value, in order to show the desired output. In the case that value of cores is the same the system will always select the core which is in the highest. This is not a problem, because in the next cycles, the maxim value will correspond with the cores that had the same value.

Analyzing the design deeper is easy observable that it will spend a big quantity of area due to for each core that the arbiter has to control , it will needs 32 count and in addition the internal logic which it uses to calculate the output.

As is possible to see in the Figure 6.3, the circuit is divided in two big parts, the counter for each bit of request, and the comparator to select the correct status signal. These parts are going now to be explained deeper.

6.1.1 Counters

Due to the system will measure the number of cycles that request signal request is active asking for permission, could be necessary to use counters for these inputs. These counters that are used are 32-bit-length, and they have the next structure.

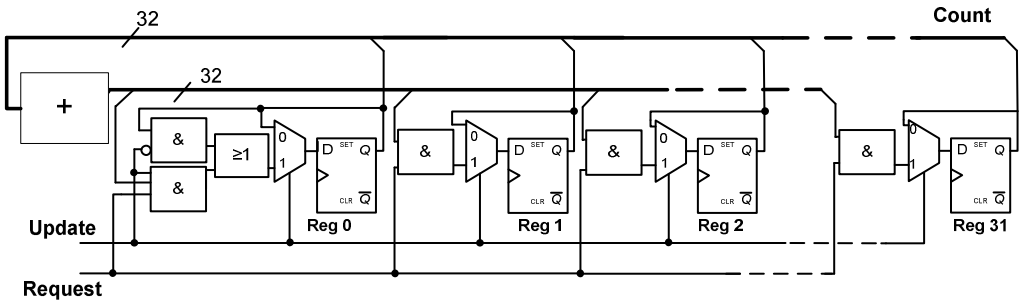

Figure 6.4: Counter Structure

As is possible to observe, the circuit is controlled by the signal update which works like an enable, so when update is not active, the counter will hold the last value of the counter, in addition when the signal request will be not active too, the counter will be reset. Is necessary to remark that counter block will be replicated many times as number of bits has the signal request. (In the Figure 6.4, signals clock and reset have been ignored to show a clearer circuit).

6.1.2 Comparator

To compare the signals is not easy because the comparison will depend of number of cores that managed. So, it means if there are small number of cores the logic that is used to compare the signal will be much less than the logic that is used when there are a big number of cores. This increment of the logic depends basically because the results of all counters have to be compared between all them.

Figure 6.5 shows the logic structure which is used to compare the results of each counter.

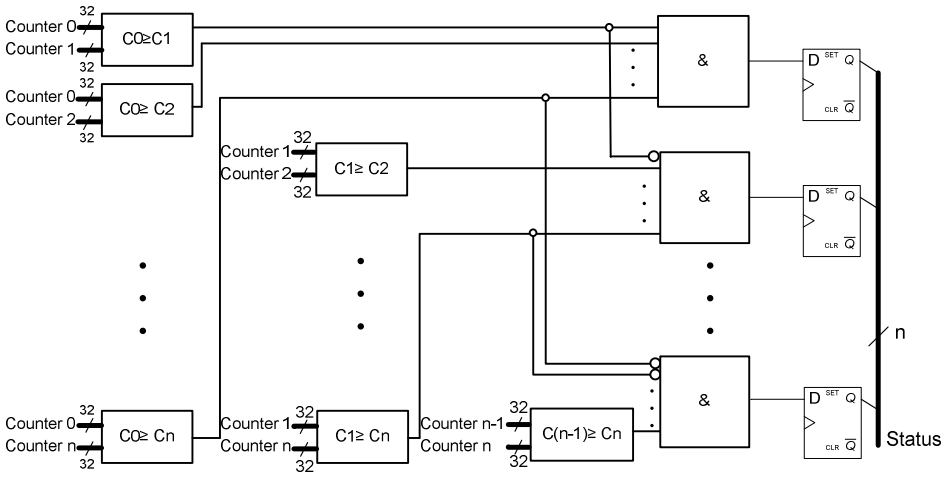

Figure 6.5: Comparison block

In a first look, is possible to appreciate that the circuit of the Figure 6.5will obtain the desired output, but analyzing it is possible to confirm that the circuit could be reduced in terms of components, and it implies a decreasing number of logic comparators. The original number of comparators is for the circuit of Figure 6.5 is given by the formula

$$n^2 - \left(\frac{n^2 + n}{2}\right)$$

Where n is the number of cores to manage. In the other hand, there is other solution with which will be saved an important quantity of area. The alternative circuit is showed in the Figure 6.6.

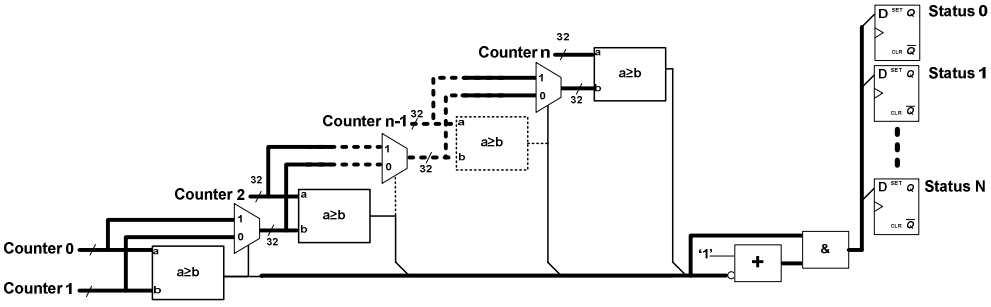

Figure 6.6: Alternative Comparator

The circuit that is showed in the Figure 6.6 needs only so many multiplexors as number of cores are required to manage (n). I addition, in the circuit is used an adder and a two-inputs- and- logic gate. The size of adder also has the size on the number of inputs .In addition could be necessary to explain that the logic that is used (adder and and-logic-gate) calculates the two's complement of the signal in order to choose always the highest priority of the output.

6.1.3 Critical path

One time the circuit has been explained, is necessary to show where will be the critical path of the circuit. The critical path of a digital circuit) is the longest path between sequential storage elements like e.g. flip-flops or latches. The timing of this critical path has to be checked during production test. The problem with this test is the low speed of the test equipment. It is difficult to check fast digital circuits with slower test equipment. Usually the signal transitions along the critical path do consume almost one complete clock cycle. If the chip tester is not able to supply a clock cycle, which is shorter than the delay of the critical path, the timing of this path cannot be checked.

Observing the Figure 6.4, is possible to see that the output of the counter has a flip-flop which will save the value of the count. And the second flip-flop can be found in the Figure 6.6, these flip flops are used to save the signal status. So the analysis of the critical path has to be done from the Figure 6.7.

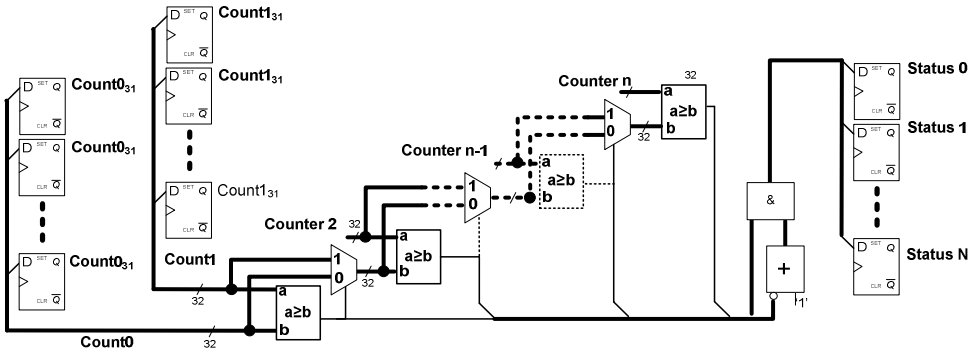

Figure 6.7: Circuit to obtain the critical path.

Looking in the midst of the circuit is easy to appreciate that the critical path will be this one which passes between the comparator and the multiplexors, so the part of the circuit which will limit the frequency behavior of all design. In order to understand better this path, in the Figure 6.8 is showed with a red line where the critical path is.

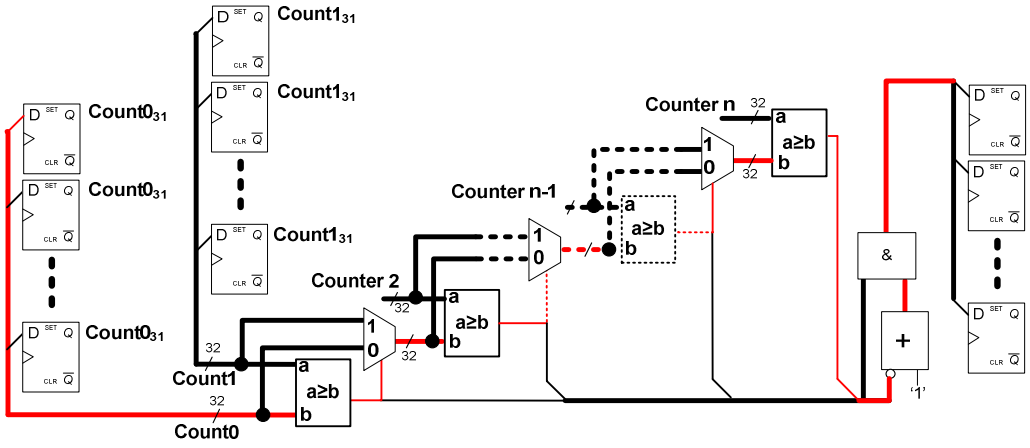

Figure 6.8 :Critical path of Arbiter 1

Therefore the critical path is this one which goes from the output of the counter of the less significant request bit until the register in the output of the circuit.

6.2 Arbiter 2

For this second design another philosophy will take in mind. Now, is very important than system has to be easy-scalable, so the technique of the design will change for this one.

The main idea for this design is to change the counter system that has been used in the first design by logic operations, which physically don't waist to much space in the wafer. Moreover, these kinds of designs are able to work in highest frequencies than design with big mathematic operations charge. Nonetheless, in the beginning of the design process could be more abstract for the designer getting start than in the first option.

Introducing in the ambit of networks is possible to find the round robin system. Round robin is an arrangement of choosing all elements in a group equally in some rational order, usually from the top to the bottom of a list and then starting again at the top of the list and so on. A simple way to think of round robin is that it is about "taking turns." In computer operation, one method of having different program process take turns using the resources of the computer is to limit each process to a certain short time period, then suspending that process to give another process a turn (or "time-slice"). This is often described as round-robin process scheduling.

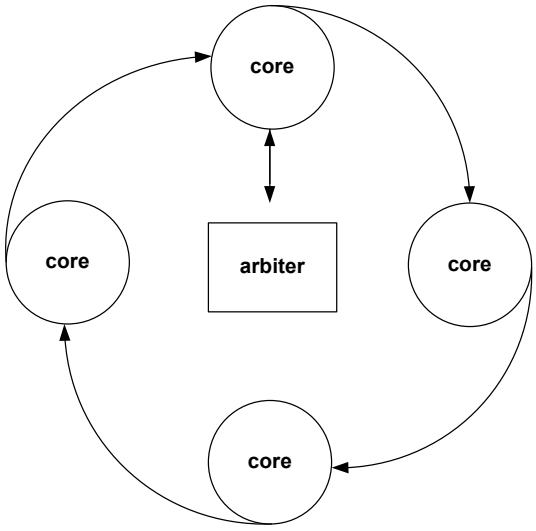

Figure 6.9: Round Robin Scheme

Comparing this kind of structure with the Figure 5.2is possible trying to adapt this system to the desired design.

In order to achieve this design, the main problem that can appear is the delay between the selection of one core and the next one. However, the system must change the selection of the cores in every single clock cycle, with this system will be obtained a big

thrift of time in handshaking. Due to this requisite the calculus of the output's design should be free of process or buckles due that this kind of sentences in the VHDL code are physically built with Flip-Flops and it implies delays.

In the Figure 6.10 is showed the ideal handshaking behavior for the design.

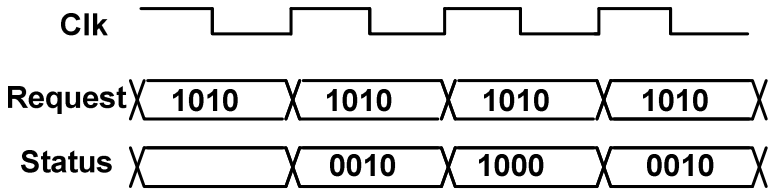

Figure 6.10: Desired behaviour of Arbiter 2

Analyzing the problem, and looking the Figure 6.10, is possible to see that the first output will always be the highest priority of the input Request (is considerate that the most significant bit is the bit which is in the right position and therefore the less significant bit is the bit which is in more in the left). Hence, this state will be named "initial status".

Secondly, is needed to obtain the "New Status". To calculate the new status is necessary to have in mind the last status and the signal request as well due to the status is always in function of Request. So, with these signal is possible to create a mask to help in the calculus. The only requirements of this mask are: Firstly, it doesn't have to have a high value in a more significant position than the "last status" signal and secondly it must show where the next high position in the vector is.

So, for example if request is 1011 the highest priority vector and the "first status" output will be 0001. One cycle after, the result of the output status will be "last status" and with request the system has to be able to create the mask, with the form XX10. As is indicated before, this mask has the two requirements.

The next step is to calculate the desired output from the mask , but this is not complicated due to will be used the same method from the first steep, due to the desired output will be now the highest priority bit of the mask. This operation will be repeated until the mask is a vector with all positions in low value.

Therefore, is possible to say that the cycle of the calculus is completed, if is analyzed the design will have three different states.

-Calculating highest priority of request or mask

-Calculating the mask

-Selecting the desired output.

In the Figure 6.11 is showed a block diagram with the processes that the design needs.

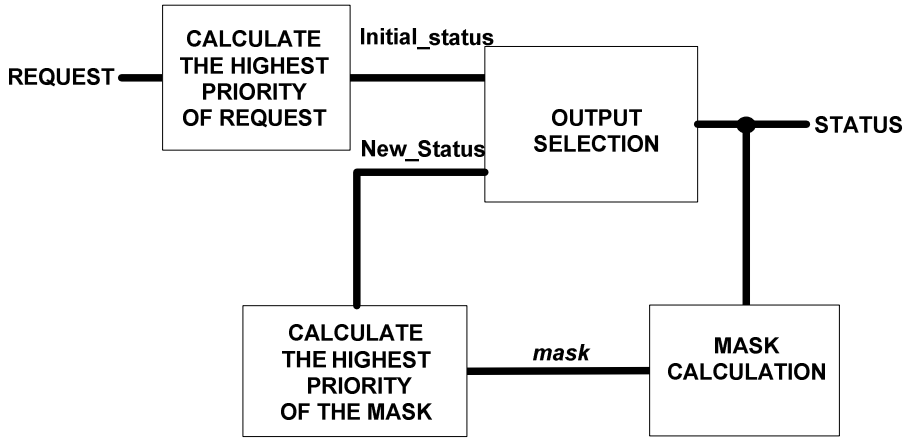

Figure 6.11: Block Diagram for Arbiter 2

6.2.1 Creating the design

In this system are included the inputs for clock, general reset, an enable system and the most important, the input from the cores, this input is request. As output the system only will have the signal status. The wide of the inputs are one bit for all of them except for request which is variable. Therefore the output status will be variable too and it will be in function of request.

Now the techniques to calculate the highest priority and the mask are going to be explained.

6.2.2 Calculating highest priority

To calculate the highest priority of a vector can be used a comparison with an and logic gate between a signal and the two's complement of the same signal. For example to calculate the highest priority of 0110 the system will calculate the two's complement and it obtain 1010 and the next step will be to operate the both signal with an and logic gate. So, the result will be 0010.

6.2.3 Calculating the mask

The function of the mask is to create a vector who helps to find the next active value in request signal, so, the form of this mask vector will be the same of request, but with the exception that it will be created from the last status, it implies that if the mask is created from the last status signal, the mask will have zeros in the previous positions to the last active value in last status output.

For example if the request vector is 1101 and the last output was 0001, the mask will be 1100.

Once here, the system to calculate the highest priority will be used due to the mask has the perfect form to use this algorithm. So, looking the last example, and applying the highest priority algorithm to the mask vector, the new status signal from the mask (1100) will be 0100.

Following this process, is easily observable that the design will work with feedback of status. So now, the last steep to finish the design will be to understand the process of selection between the initial status or the new one. Follow, is going to be explained how this part of the design works.

6.2.4 Selecting the desired output

In this point, as has been explained previously, the only steep that is needed is to choose between initial or new status. Hence, is necessary to have in mind the signals mask and new status. Due to when mask is a vector with all position in low value, it means that all the positions of the vector have been checked. So, in this case, the system will show initial status as output. In addition, in the case that new status is other vector with all position in low value, it is because the mask had all its positions in low value, and the system will start to calculate the output for the first position of the vector request.

6.2.5 Behaviour examples

Next, the tables show the behavior of the design where can be observed how the signals change.

In this first example the signal request does not change, so, the status signal will give permission every clock cycle to every slave starting for the MSB.

	1st cycle	**2nd cycle**	**3rd cycle**	**4th cycle**
REQUEST	1101	1101	1101	1101
Initial Status	0001	0001	0001	0001
Mask	0000	1100	1000	0000
New Status	0000	0100	1000	0000
STATUS		0001	0100	1000

Table 6-1: Arbiter Behaviour 1

In this moment the user can question what happen if the signal Request changes in the middle of the calculus process. So, in this second example is showed this case.

	1st cycle	**2nd cycle**	**3rd cycle**	**4th cycle**
REQUEST	1101	1101	0010	0010
Initial Status	0001	0001	0010	0010
Mask	0000	1100	1000	0000
New Status	0000	0100	1000	0000
STATUS		0001	0100	0010

Table 6-2: Arbiter Behaviour 2

In the simulation can be observed that there are not problems if the request signal changes. In this case, the design starts with again with the MSB of the new Request signal.

	1st cycle	**2nd cycle**	**3rd cycle**	**4th cycle**
REQUEST	0101	0101	1001	1001
Initial Status	0001	0001	0001	0001
Mask	0000	1100	1000	0000
New Status	0000	0100	1000	0000
STATUS		0001	0100	1000

Table 6-3: Arbiter Behaviour 3

In this third example the Status changes in the middle of the calculus, and the design works without problems, the status signal respects the round robin behavior.

The Figure 6.12 shows the circuit designed. As is possible to appreciate, the design only will have a flip flop per core managed so, it will decrease as minimum a clock cycle comparing with the first design.

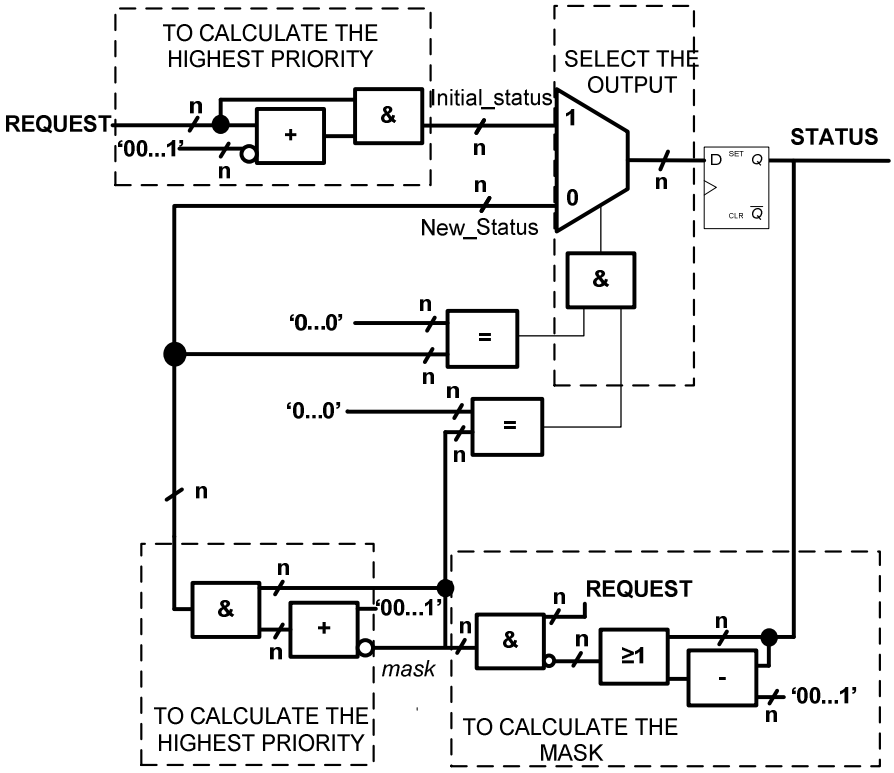

Figure 6.12: Round Robin Arbiter

6.2.6 Critical path

This path will be the route physically between one input and a flip flop which can accumulate the delay of the gates. This involves that it will be the most problematic, when the frequency is higher. Looking in the Figure 6.13 is possible to see the critical path (marked in red).

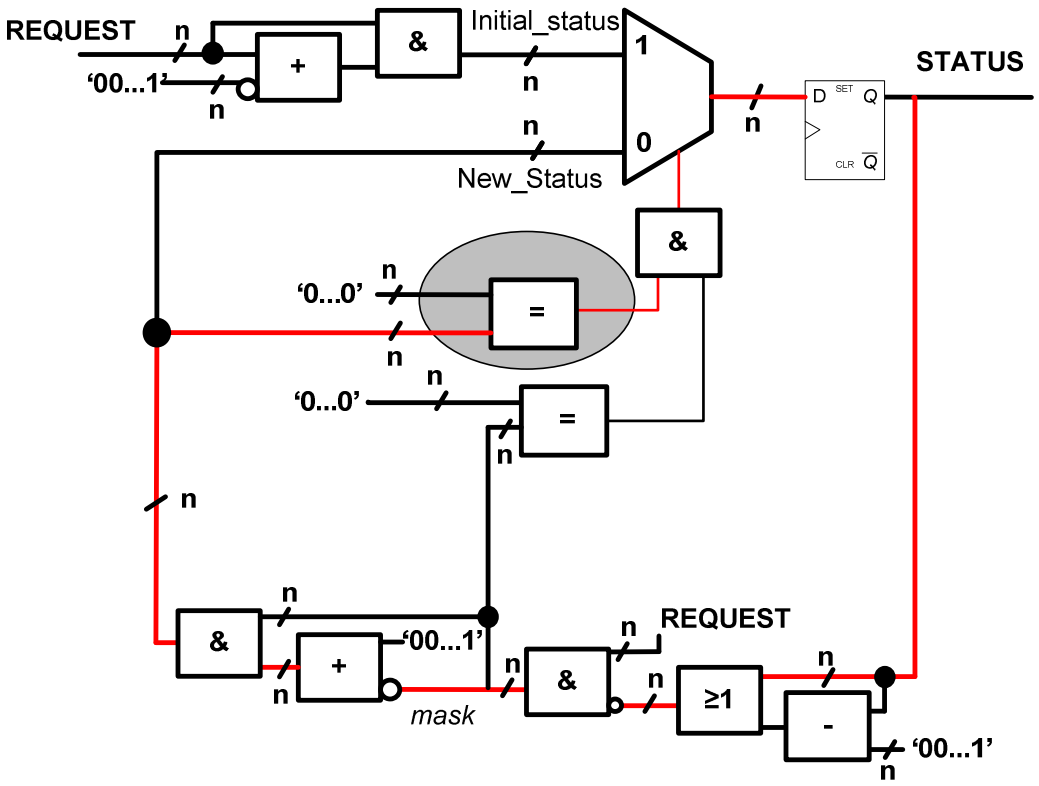

Figure 6.13: Critical Path of Arbiter 2

As is possible to appreciate, there is a grey circle in the Figure 6.13. This circle is rounding the comparator, and the critical path will depend about it, due to when the number of cores increments, this comparator will increment too.

Therefore, due the comparator will increment in order the number of inputs increment; the critical path will be this one which goes from the output of any flip flop until the input of the less significant status output , in other words status (0).

To confirm that this critical path is correct is needed to use Synopsys Design Visio which will show the paths with less slack, or in other words, the critical paths of the circuit. Once here, and one time checked for the different number of cores the results of Synopsys Design Visio always corresponds with the theoretically values that have been calculated previously.

6.3 Arbiter 3

Using as a reference the second arbiter is going to apply the concept of priority list. In this case, will be created a priority vector that indicates which core has the priority to uses the bus. For this design has been used the same circuit that in the second arbiter with the condition that all the logic is duplicated.

As is explained above, the logic is duplicated; one part of this logic will work as a normal round robin arbiter but the difference is that the second part of the logic has as an input the logic and operation between the signal priority and the request input signal. Is important to say that the signal priority is an internal signal of the circuit and it can be modified by the user giving or removing priority to the cores when the user desires. The result of this and logic operation between the signal priority and the signal request is the signal priority2, and it will give information about the cores which have priority and the cores want to use write in the system .When the signal priority2 has all its positions in low value, it means that the output of the design is the round robin arbiter. In the case that priority2 has different value than a null vector, it means that some core with priority wants to use the system, and they will have the baton. The Figure 6.14, shows the design of the circuit

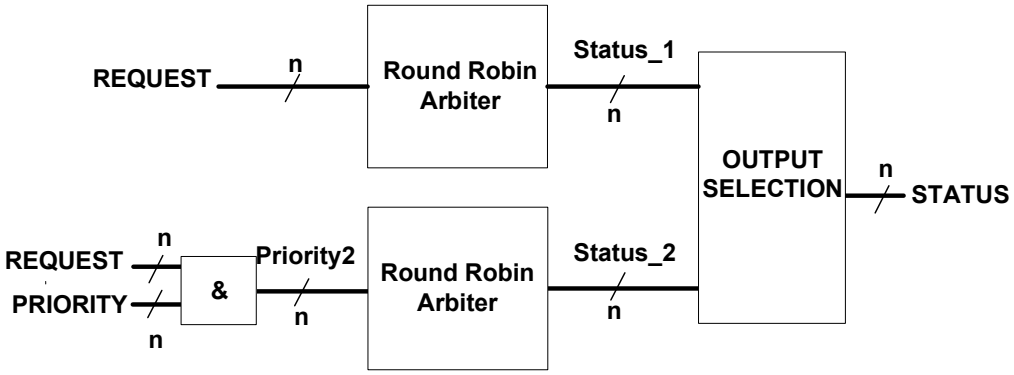

Figure 6.14: Arbiter 3 Block Diagram

In the Figure 6.14 there is two blocks which are called Round Robin Arbiter, those blocs corresponds with the second arbiter designed with the exception that the flip-flop to register the output have been removed. This Round Robin blocs used in the Figure 6.14 are deeper explained in the Figure 6.15 .

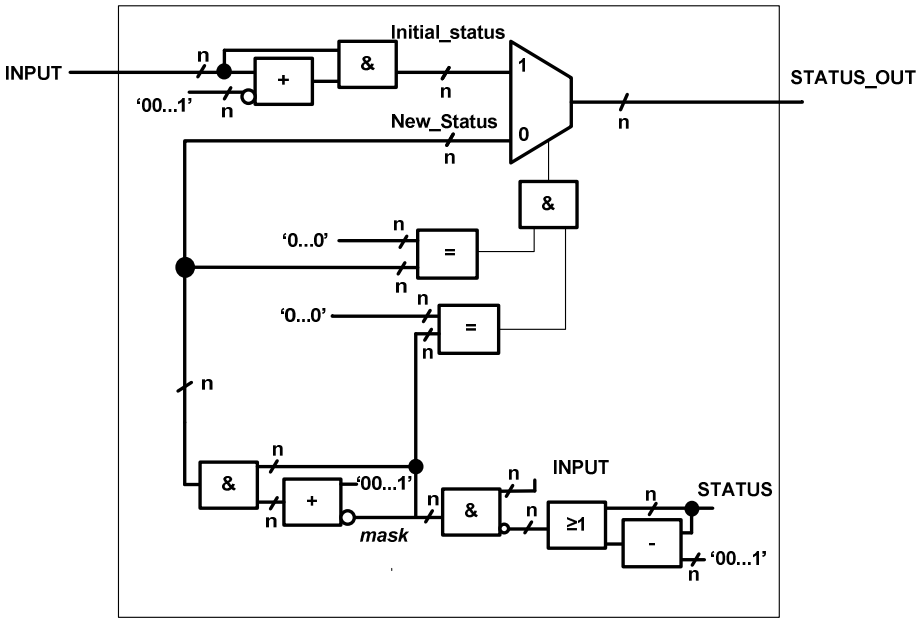

Figure 6.15: Round Robin Block

Hence, is necessary to explain the behavior of the block output selection used in the Figure 6.14. As is explained above, the signal priority2 will indicate if the desired output is status_1 or status_2, the design of this part of the circuit will needed a comparator to compare the signal priority2 with a vector with all positions in low value in addition a multiplexor to select the outputs Status_1 or Status_2 will be required, and finally the circuit will need flip-flops to register the output. To understand it better, the Figure 6.16 shows the circuit.

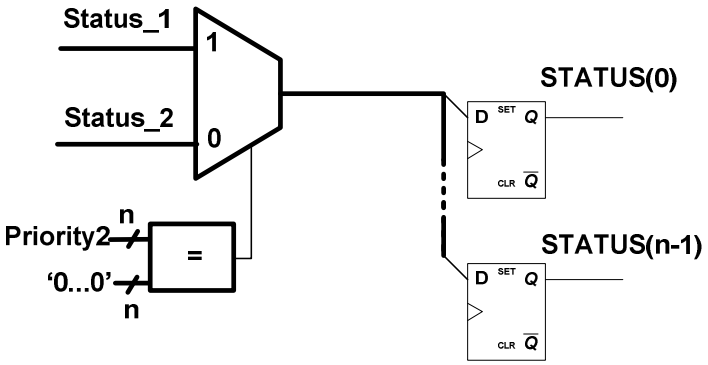

Figure 6.16: Output Selector Block

Therefore the final circuit for the arbiter 3 is showed in the Figure 6.17

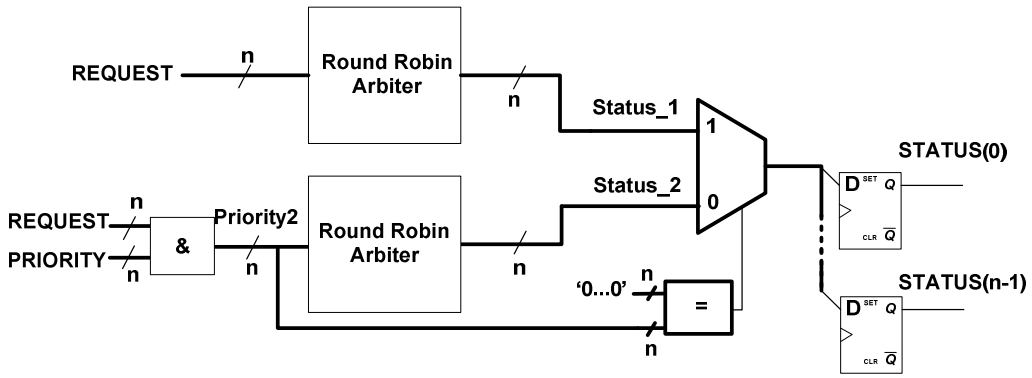

Figure 6.17: Schematic Arbiter 3

6.3.1 Critical path

One more time the critical path of the circuit is going to be analyzed. Hence, and observing that the circuit is very similar than the arbiter 2 implies that the critical path is almost the same. The only difference is the second multiplexor in the output. The complete circuit is showed in the Figure 6.18 with red line.

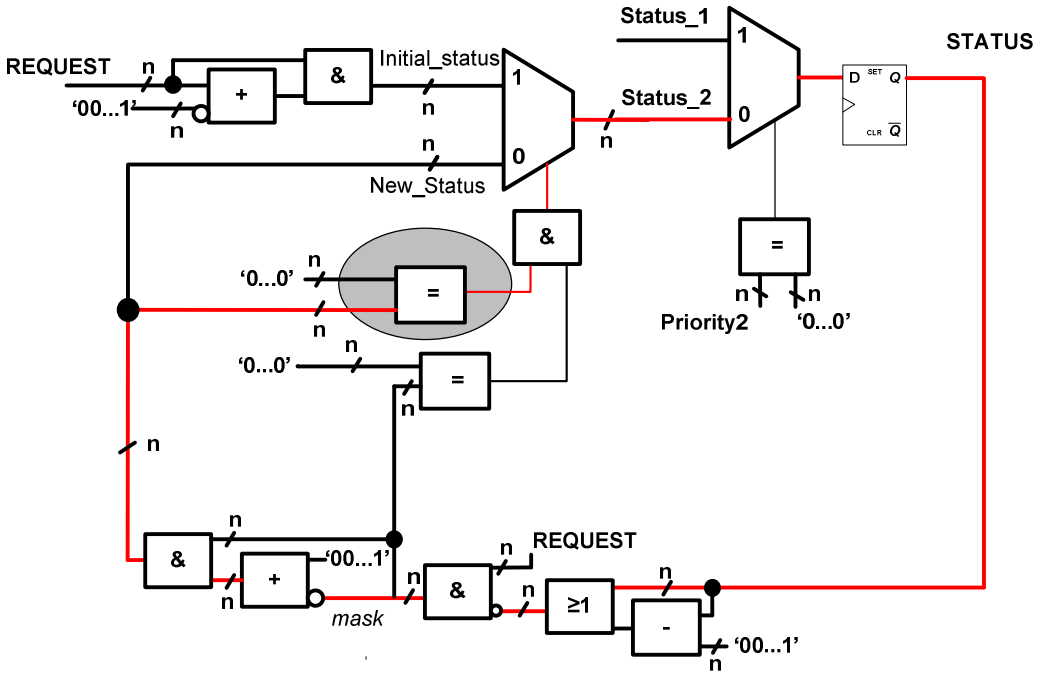

Figure 6.18: Critical Path Arbiter 3

As is possible to appreciate in the Figure 6.18, one more time the critical path depends of the number of inputs, it implies that the comparator with the grey circle is bigger. So finally, the critical path will be again between output status and the input of the register status (0).

7 Environment Simulation

One time here, is necessary to explain which are the scenarios where the test have been done. As is usual, a environment has been created in order to simulate the different arbiters, but although there are a lot of different scenarios, the interface of scenarios is showed in the Figure 7.1.

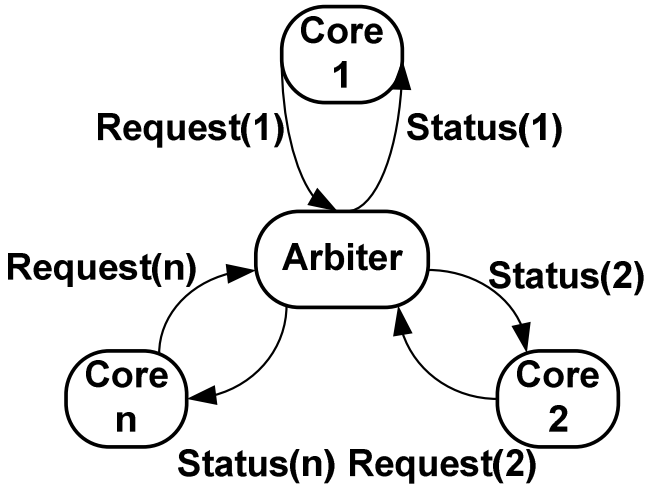

Figure 7.1: Interface of the scenario

As is possible to see in Figure 7.1, the scenarios that have been created are formed basically by two kinds of devices. The first ones are the cores and the second one is the arbiter. The main idea is to create different kinds of environment, and analyze how the arbiter react against changes of the type of cores and changes in the position of them respect to the position of the arbiter's input. With this technique will be obtained different scenarios, where the arbiters will be tested.

7.1 Cores

The cores that have been created are formed basically by a Status input where will be connected to the output status of the arbiter, and a random output request. Is necessary to say that the core has two outputs (Req and request) and one of them (request) is simply a logic or operation between req(1) and req(0). In order to clarify this design, the interface of the cores is showed in the Figure 7.2.

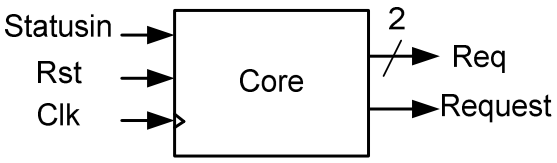

Figure 7.2: Interface of the Core

For the firsts designs will be only used the output request, due to req(1) and req(0)will be used as outputs to select atomic operations. Different combinations of req mean different kind of atomic operations.

The main difference between cores is the frequency which the core asks for an write or read permission to the arbiter, or explaining it with other , it will be the frequency which the output request will be active again, one time the cores receives the permission to operate. This frequency, has been called average requesting time.

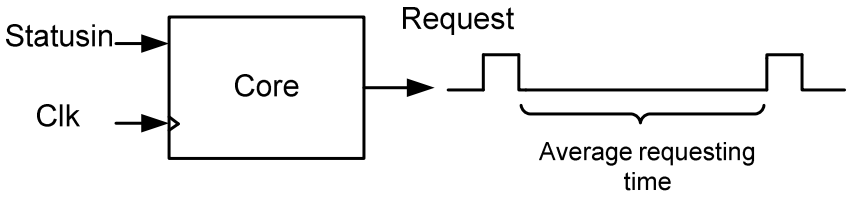

Figure 7.3: Average requesting time

The variable average waiting time is a very important characteristic due that with this parameter will be changed the activity of the scenarios, so changing this parameter new scenarios will be obtained. Moreover, the activity of the scenario will also depend of the number of cores per scenario as is explained before.

The general structure of the core is showed in the Figure 7.4.

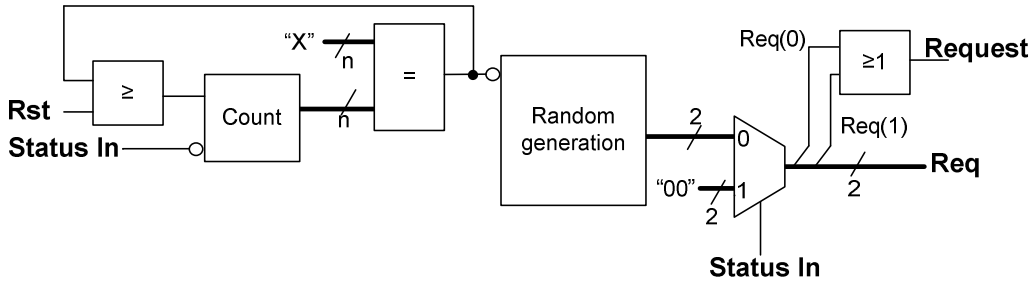

Figure 7.4: General Structure of the Cores

7.2 Scenarios

Hence here, is time to explain the different scenarios that have been created. As is explained previously, scenarios will depend basically about the cores, and the difference between them is the frequency which the cores try to ask to the arbiter for permission to operate. In addition the number of cores will change, so with it will also change the stress of the integration between the arbiters and cores. The concept of stress in this experiment will be related with the number of cores and the average waiting time of each one, for example, if there is a scenario with four cores which write every ten clock cycles, is possible to say that the design will be less stressed that the same structure with twenty cores writing every ten clock cycles.

For these simulations, ten scenarios have been designed and in addition all them have been tried for 4, 8, 16 and 32 cores. With these changes in the cores, the only objective is to change the stress of the scenario and observe the reaction of the arbiter, in several kinds of environments.

To understand the scenarios would be necessary to understand the how the cores are named. But it is very simple. The number of core is always named as the format "Core Type X" where X is a number. This number will show how many cycles the core waits when it is ready to ask for a permission of the arbiter. So, with lower type of core implies more activity in the scenario.

7.2.1 Scenario 1

This scenario will be the most stressed due the cores will always try to ask for a arbiter permission. It will be the most critical scenario.

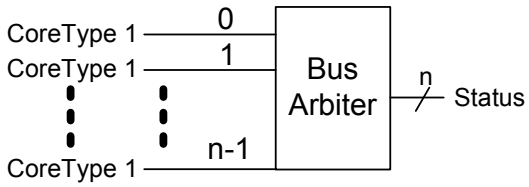

Figure 7.5: Scenario 1

7.2.2 Scenario2

In this scenario the cores with more activity are in the first positions, and they decrement their activity in the last positions.

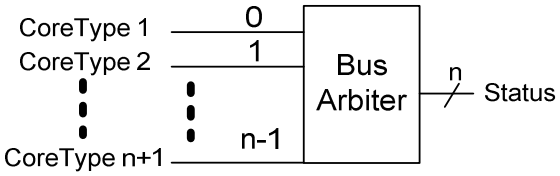

Figure 7.6: Scenario 2

7.2.3 Scenario 3

In this case is the opposite of the scenario 2, the most active cores are in the last positions and less active in the firsts places.

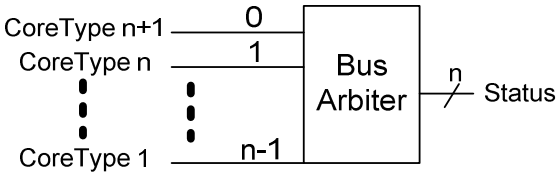

Figure 7.7: Scenario 3

7.2.4 Scenario 4

This scenario is formed by two very similar types of cores, it will be a middle stressed scenario.

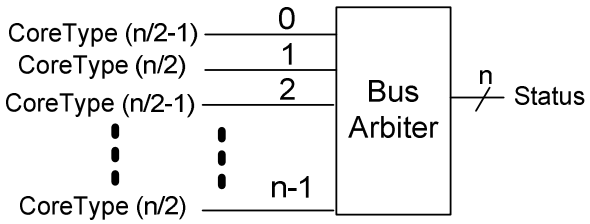

Figure 7.8: Scenario 4

7.2.5 Scenario5

Scenario 5 is formed as the same way that scenario 4 but whit the small difference that cores will have more average requesting time, so it will one of the less stressed scenarios.

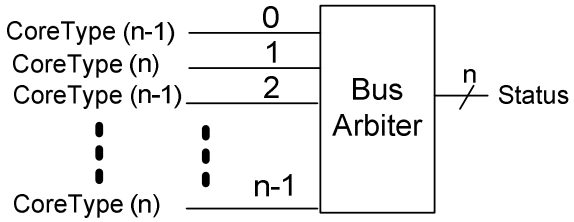

Figure 7.9: Scenario 5

7.2.6 Scenario 6

This core is formed by active cores; the main characteristic of this scenario is that they are duplicated. Is easier to understand this scenario observing the Figure 7.10 .

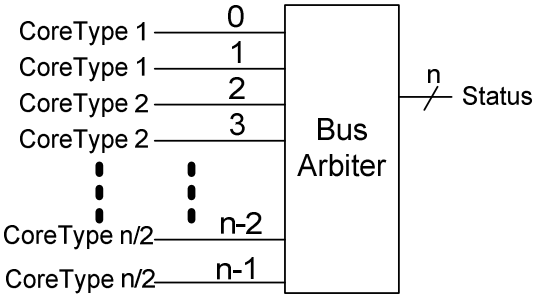

Figure 7.10: Scenario 6

7.2.7 Scenario 7

Is the same than scenario 6 but with the inverted order.

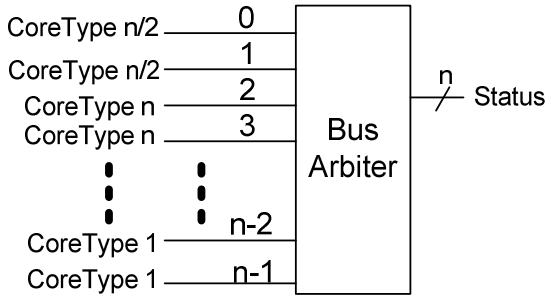

Figure 7.11: Scenario 7

7.2.8 Scenario 8

This scenario and has the same cores than the scenario 2 but with the difference that now, the cores are not in order, now they are mixed.

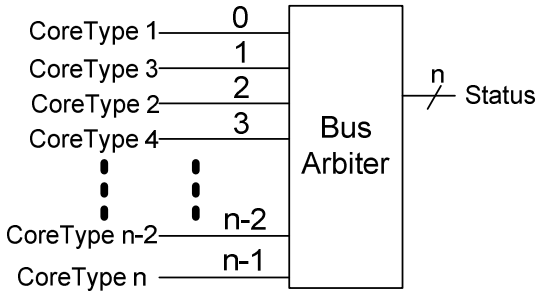

Figure 7.12: Scenario 8

7.2.9 Scenario 9

This scenario and has the same cores than the scenario 3 but with the difference that now, the cores are not in order, now they are mixed.

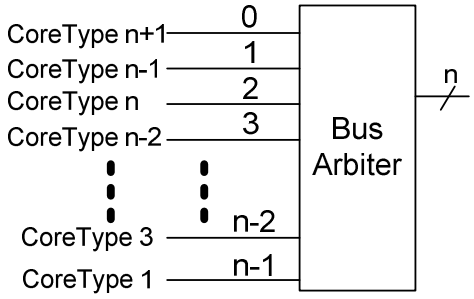

Figure 7.13: Scenario 9

7.2.10 Scenario 10

Scenario 10 has been created in order to observe how the arbiter works with not too much stress. This has been formed by cores which difference between them is they increment 10 cycles the average requesting time every position.

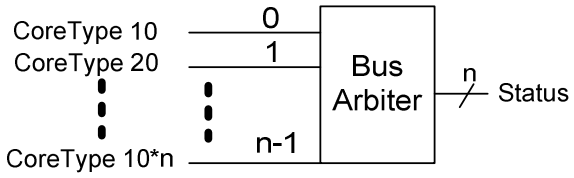

Figure 7.14: Scenario 10

8 Synopsys Design Vision

Design Vision is a graphical user interface (GUI) to the Synopsys synthesis environment and an analysis tool for viewing and analyzing your design at the generic technology (GTECH) level and the gate level.

A synthesis tool takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as output. The resulting gate-level netlist is a completely structural description with only standard cells at the leaves of the design. Internally, a synthesis tool performs many steps including high-level RTL optimizations, RTL to unoptimized boolean logic, technology independent optimizations, and finally technology mapping to the available standard cells. A synthesis tool is only as good as the standard cells which it has at its disposal. Good RTL designers will familiarize themselves with the target standard cell library so that they can develop a solid intuition on how their RTL will be synthesized into gates.

Synopsys provides a library called Design Ware which includes highly optimized RTL for arithmetic building blocks. For example, the Design Ware libraries contain adders, multipliers, comparators, and shifters. DC can automatically determine when to use Design Ware components and it can then efficiently synthesize these components into gate-level implementations.

RTL synthesis is an automated design task in which high-level design descriptions written in Hardware Description Languages (such as VHDL, Verilog, or SystemVerilog) are transformed into gate-level netlists. Gate-level netlist is basically a circuit implementation of the design made of library components (both combinational and sequential cells) available in the technology library and their interconnections. The netlist is generated by the synthesis tool according to the constraints set by the designer. Figure 8.1 below shows an overview of the synthesis.

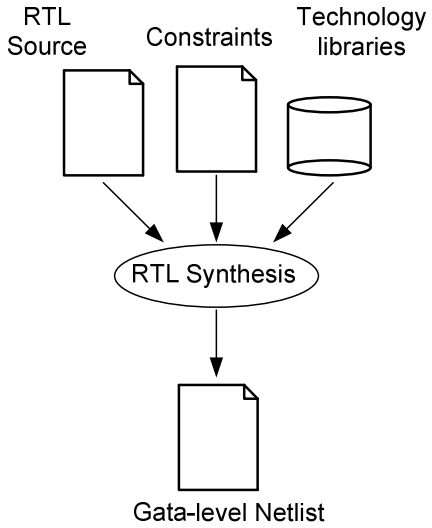

Figure 8.1: An overview of the synthesis.

Synthesis is a complex task consisting of many phases and requires various inputs in order to produce a functionally correct netlist. The following lines presents the basic synthesis flow with Synopsys Design Compiler they are showed in the Figure 8.2.

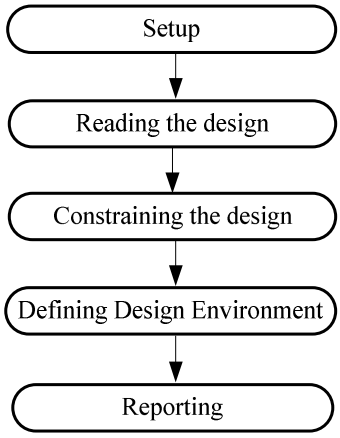

Figure 8.2: Synthesis Flow

8.1 Setup

The `.synopsys_dc.setup` file is the setup file for Synopsys' Design Compiler. Setup file is used for initializing design parameters and variables, declare design libraries, and so on. Shortly, the setup file defines the behavior of the tool and is required for setting the tool up correctly. The commands in this file are executed when Design Compiler is invoked.

8.2 Reading the design

The first task in synthesis is to read the design into Design Compiler memory. Reading in an HDL design description consists of two tasks: analyzing and elaborating the description. The analysis command performs the following tasks. The first one is to read the HDL source and checks it for syntactical errors. And the second is to create HDL library objects in an HDL-independent intermediate format and saves these intermediate files in a specified location.

Is important to say that if the analysis reports errors, they must be fixed, and the design reanalyzed before continuing.

8.3 Constraints the design

The next task is to set the design constraints. Constraints are the instructions that the designer gives to Design Compiler. They define what the synthesis tool can or cannot do with the design or how the tool behaves. Usually this information can be derived from the various design specifications (e.g. from timing specification).

There are basically two types of design constraints:

8.3.1 Design Rule Constraints

Design rules constraints are implicit constraints which means that they are defined by the ASIC vendor in technology library. By specifying the technology library that Design Compiler should use, also are specified all design rules in that library. These rules are not discarded or overrided.

8.3.2 Optimization Constraints

Optimization constraints are explicit constraints (set by the designer). They describe the design goals (area, timing, and so on) the designer has set for the design and work as instructions for the Design Compiler how to perform synthesis.

The optimization constraints comprise timing and maximum area constraints. The most common timing constraints include:

8.3.3 System clock definition and clock delays

Clock constraints are the most important constraints in ASIC design. The clock signal is the synchronization signal that controls the operation of the system. The clock signal also defines the timing requirements for all paths in the design. Most of the other timing constraints are related to the clock signal.

8.3.4 Input and output delays

Input and output delays constrain external path delays at the boundaries of a design. Input delay is used to model the path delay from external inputs to the first registers in the design. Output delay constrain the path from the last register to the outputs of the design.

8.3.5 Minimum and maximum path delays

Minimum and maximum path delays allow constraining paths individually and setting specific timing constraints on those paths.

Note that Design Compiler tries to meet both design rule and optimization constraints but design rule constraints always have precedence over the optimization constraints. This means that Design Compiler can violate optimization constraints if necessary to avoid violating design rule constraints.

8.4 Defining Design Environment

Also is needed to describe the environment in which the design is supposed to operate. The design environment description includes:

8.4.1 Defining Operating Conditions

The operating conditions consider the variations in process, voltage, and temperature (PVT) ranges a design is expected to encounter. These variations are taken into consideration with operating condition specifications in the technology library. The cell and wire delays are scaled according to these conditions.

8.4.2 Modeling Wire Loads

Wire load models are used to estimate the effect of interconnect nets on capacitance, resistance, and area before real data is obtained from the actual layout. These models are statistical models and they estimate the wire length as a function of net's fan-out.

8.5 Reports

Once the synthesis has been completed, is necessary to analyze the results. Design Compiler provides together with its graphical user interface (Design Vision) various means to debug the synthesized design. These include both textual reports that can be generated for different design objects and graphical views that help inspecting and visualizing the design.

There are basically two types of analysis methods and tools:

8.5.1 Generating reports for design object properties

Reporting commands generate textual reports for various design objects: timing and area, cells, clocks, ports, buses, pins, nets, hierarchy, resources, constraints in the design, and so on.

8.5.2 Visualizing design objects (Design Vision)

Some design objects and their properties can be analyzed graphically. You may examine for example the design schematic and explore the design structure,

visualize critical and other timing paths in the design, generate histograms for various metrics and so on.

These methods and tools are used to verify that the design meets the goals set by the designer and described with design constraints. If the design does not meet a design goal then the analysis methods can help determining the cause of the problem.

9 Analysis of the area

One time explained how works Synopsys Design Visio is time to show the areas results by the arbiters.

To do the analysis of the required area for the arbiters has been necessary to analyze the designs with different number of cores to manage. The number of cores to manage has been 4, 8 16 and 32. With the comparison of the results achieved by Synopsys for different number of cores to manage will be possible to appreciate the behavior of the arbiter for n cores. In the next table are showed the results

9.1 Arbiter1

Cores to manage	Required Area(μm^2)
4	5586,20
8	11583,12
16	23791,42
32	50152,60

Table 9-1: Area Results of Arbiter 1

As is possible to appreciate in the Figure 9.1 the growth is almost lineal. In addition is possible to say that the area average increment for each core is 1474,66 μm^2

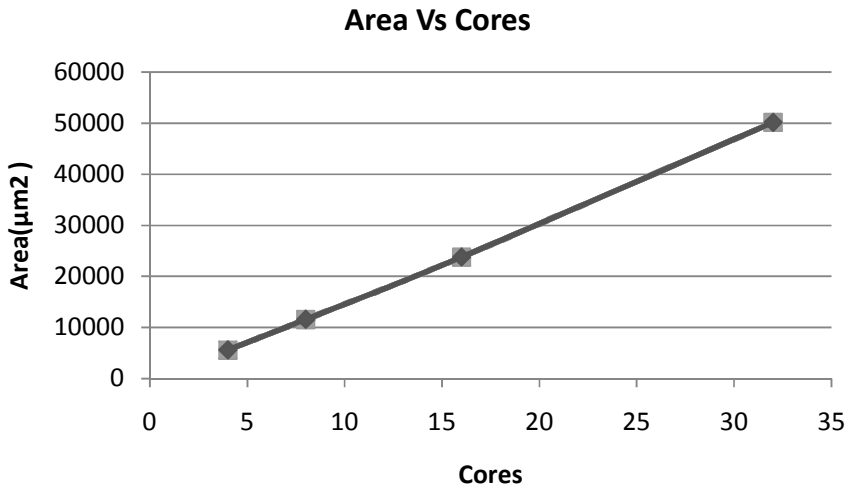

Figure 9.1 :Area results for Arbiter 1

9.2 Arbiter 2

Cores to manage	Required Area(μm^2)
4	218,73
8	529,01
16	1118,37
32	2845,68
64	5802,23
128	11688,72

Table 9-2: Area Results of Arbiter 2

As is possible to appreciate in the Figure 9.2 the growth is almost lineal. Is possible to appreciate that between 4 and 16 cores the arbiter grows 63,56 μm^2 , in the other hand, for values since 32 cores, the arbiter grows 90,30 μm^2 per core. This probably because when there is small number of cores Synopsys optimize the design with logic instead to use adders, comparator, etc…

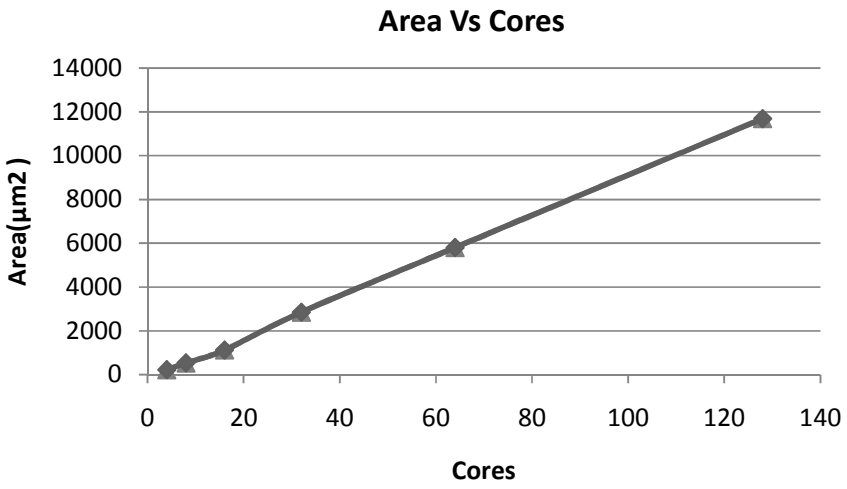

Figure 9.2: Area results for Arbiter 2

As is possible to appreciate the firs arbiter uses much more area than the round robin arbiters. Arbiter 1 spends almost 20 times more area than the second arbiter. The main reason of this big increment is the use of the 32-bits-counter that is used to calculate how long the signal request is active.

9.3 Arbiter 3

Cores to manage	Required Area(μm^2)
4	354,91
8	862,45
16	1823,27
32	4685,14
64	9587,69
128	19311,56

Table 9-3: Area Results of Arbiter 3

As is possible to see in the Figure 9.3 the growth in this case is almost lineal too. Due to in this design is using more logic, the area increments. For example between 4 and 16

cores, the design grows 103, 49 μm^2 per core to manage. Between 32 and 128 the design grows with 149,02 μm^2 per core.

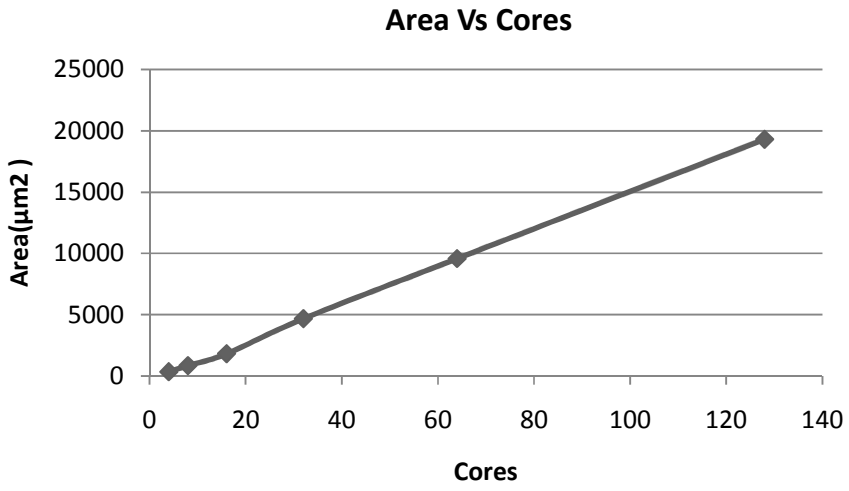

Figure 9.3: Area results for Arbiter 3

Analyzing the obtained data is possible to appreciate that the Arbiter 3 uses approximately the double of area than the second arbiter. This result is logic due to for the design has been used two times the Arbiter 2. Therefore, analytically the conclusion can be that the required area of Arbiter 3 is the double of arbiter2 but this is not completely real because Synopsys optimizes the design and it reduces the relation between areas until 1,65.

10 Analysis of the maximum work frequency

Now is the time to analyze the maximum frequency that the circuit is able to work with, as is possible to intuit the first arbiter will be slower than the second one due to the internal structure of it.

As in the area analysis, in this case has been also the circuits analyzed changing the number of cores to manage. Anyway in the next tables and graphics are showed the result for them.

10.1 Arbiter 1

Cores to manage	Maximum frequency(MHz)
4	116,95
8	52,63
16	25,05
32	14,29

Table 10-1: Frequency results of Arbiter 1

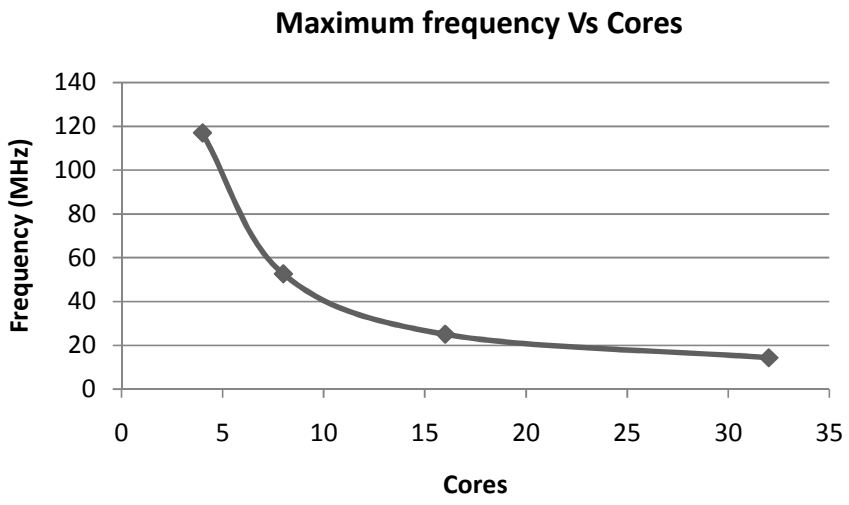

Figure 10.1: Maximum frequency results for Arbiter 1

As is normal, for bigger number of cores to manage, the frequency will decrease.

10.2 Arbiter 2

The obtained results for the Arbiter 2 are showed in the table 8.

Cores to manage	Maximum frequency(MHz)
4	1449,28
8	1098,90
16	724,64
32	392,16
64	230,95
128	128,04

Table 10-2: Frequency results of Arbiter 2

Is important to appreciate that due that the critical path of the first arbiter is formed for more logic than the second the first arbiter will be slower than the second arbiter.

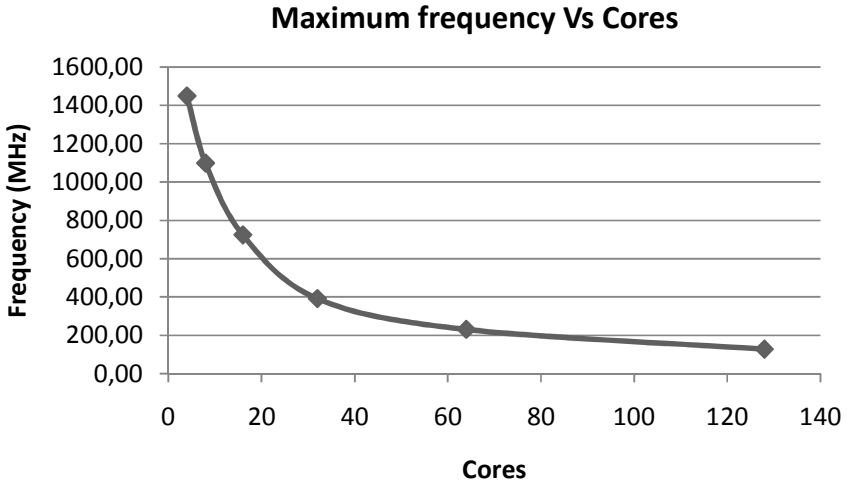

Figure 10.2: Maximum frequency results for Arbiter 2

10.3 Arbiter 3

In this case, after the simulation with Synopsys, the results are showed in the table 9

Cores to manage	Maximum frequency(MHz)
4	1282,05
8	1020,41
16	666,67
32	386,10
64	234,19
128	127,23

Table 10-3: Frequency results of Arbiter 3

Is important to appreciate that the critical path of the Arbiter 3 is very similar than Arbiter 2 and it implies than Figure 10.3 and Figure 10.2 are very similar with the difference that for the Arbiter 3 the frequency will be a little smaller due to the critical path is longer.

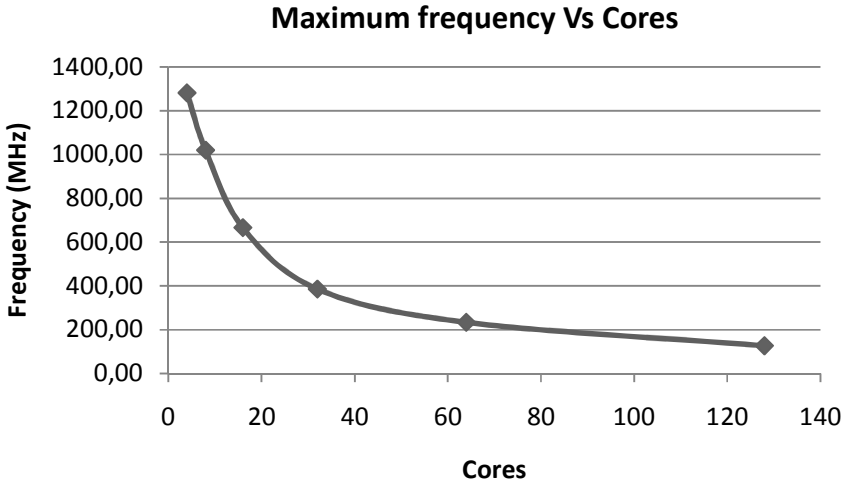

Figure 10.3: Maximum frequency results for Arbiter 3

Analyzing the values of the frequency and observing the Table 10-2 and Table 10-3 and in addition Figure 10.4,Figure 10.5 and Figure 10.6 as well, is possible to see how the arbiter cuts in half its maximum frequency every time that the number of cores is squared. Looking the results of the second design Figure 10.2 , is easily observable that

maximum work frequency is not reduced like the first model. This is due, every time that the circuit add logic to control new cores, the critical path is longer, and it implies that in the case of the first arbiter , the increment of a comparator of 32 bits instead the round robin arbiter which only needs one-bit comparator. In addition, the first design uses 33 flip-flops per core to control (32 for the counter and 1 to register the output status). For the second design , is only needed one flip flop in order to register the output status and this is because the biggest part of this design has been created using adders, comparators , and logic. In the case of the Arbiter 3, the behavior is very similar than Arbiter 2 due the critical path doesn't change too much.

Now is showed the area that the design requires versus the maximum frequency that the circuit is able to work with, for the different values of cores that the arbiter must manage.

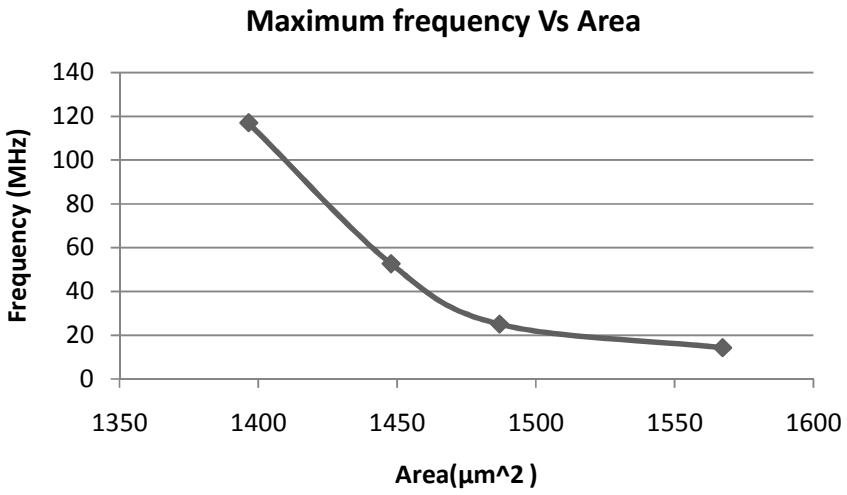

Figure 10.4: Frequency vs. area analysis. Arbiter 1

As is normal, for bigger number of cores, the frequency will decrease and the required area will be bigger.

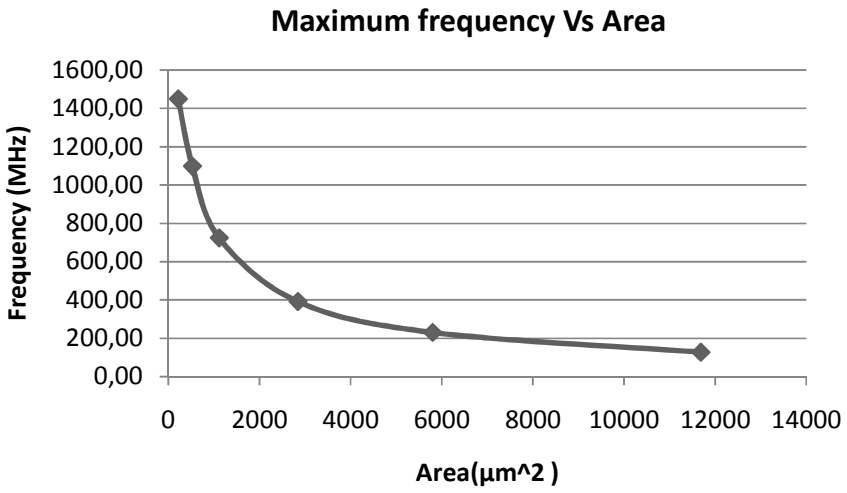

Figure 10.5: Frequency vs. area analysis. Arbiter 2

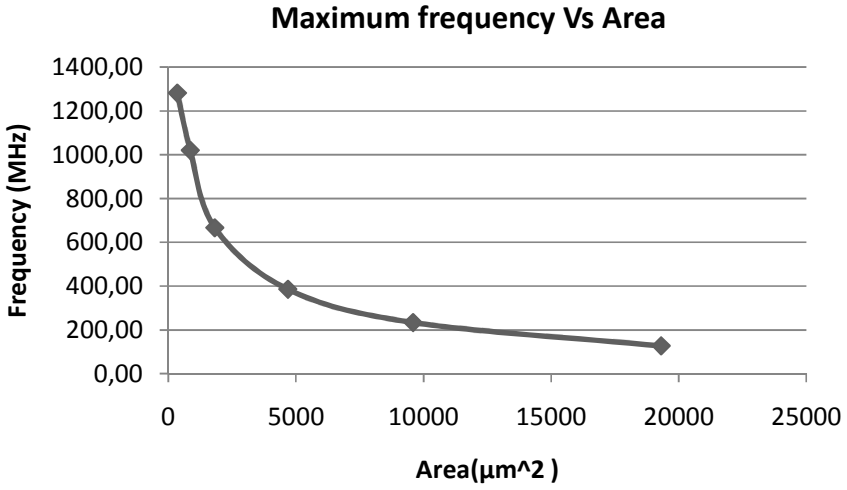

Figure 10.6: Frequency vs. area analysis. Arbiter 3

As is possible to appreciate comparing, Figure 10.4, Figure 10.5and Figure 10.6, Arbiter 2 and Arbiter 3 is much more fast and compact than Arbiter 1. A deeper discussion about the result will be showed in the next chapter.

11 Cycle time analysis

One time the scenarios has been created would be necessary to simulate them and in addition to measure the average time between the core ask for a permission to write in the bus until the arbiter gives the that the arbiter needs in give the baton to it.

Therefore, is necessary to use a virtual environment to verify the correctness or soundness of the design, this virtual environment is the testbench, which will be used to measure the times.

This file is composed by much process as cores the design has. Furthermore, this test bench has been designed to configure by the user the number of times that the cores ask for a permission of the arbiter, so, for the simulations has been used 1024 petitions per core, otherwise if the number of simulations is very small it implies that obtained results could not be realistic. Therefore, in order to calculate the average of the handshaking time, every process accumulates the time between the signal request stats to be active and the status signal becomes active as well. All these times will need to be divided by 1024 that in this case is the time that has been elected for the simulations.

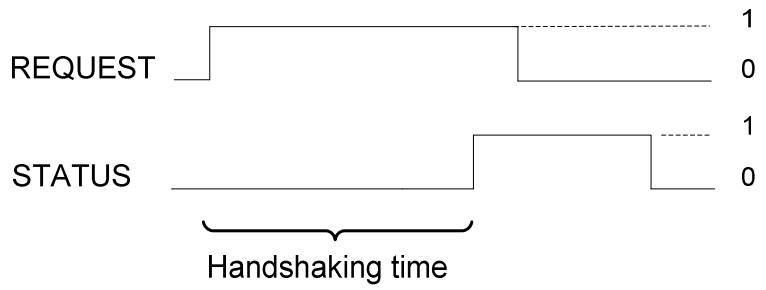

Figure 11.1: Handshaking Time

Therefore, now the results for the arbiters with different number of cores are showed in the tables 9, 10, 11 and 12. The values that are showed in these tables are the average handshaking time for all the cores and in addition all they are nanoseconds.

Is very important to inform that the clock cycle used in the simulation has been 10 ns. And the results of the tables are all in nanoseconds.

Analysis for 4 Cores

	Sce1	Sce2	Sce3	Sce4	Sce5	Sce6	Sce7	Sce8	Sce9	Sce10
Arbiter 1	70	55	55	65	45	65	65	55	55	21,2
Arbiter 2	10,5	13	13,5	14,2	13,5	14,2	17	13,5	13,2	10,7
Arbiter 3	21,0	20,3	18,0	15,8	24,3	24,5	21,8	20,3	18,8	16,0

Table 11-1: Handshaking average time for 4 cores

Now is incremented until 8 cores, and the results are showed in the table

Analysis for 8 Cores

	Sce1	Sce2	Sce3	Sce4	Sce5	Sce6	Sce7	Sce8	Sce9	Sce10
Arbiter 1	190	155	154,3	165	125	160,2	160,2	155	156,3	28,7
Arbiter 2	50	34	20,1	25	13,5	38,8	30,2	21,5	20,7	11,3
Arbiter 3	--	40,1	31,0	41,6	23,4	45,9	30,5	40,8	51,1	16,4

Table 11-2: Handshaking average time for 8 cores

Is possible to appreciate in the tables, that when the number of cores increments, is easier to observe that there are different kind of scenarios.

Analysis for 16 Cores

	Sce1	Sce2	Sce3	Sce4	Sce5	Sce6	Sce7	Sce8	Sce9	Sce10
Arbiter 1	430,0	355,0	337,9	355,0	285,0	395,6	370,0	354,4	337,3	36,4
Arbiter 2	130,0	74,9	52,9	55,0	15,3	95,3	70,1	60,5	53,1	11,5
Arbiter 3	--	162,6	74,6	51,1	27,9	--	--	414,4	71,2	17,0

Table 11-3: Handshaking average time for 16 cores

Analysis for 32 Cores

	Sce1	Sce2	Sce3	Sce4	Sce5	Sce6	Sce7	Sce8	Sce9	Sce10
Arbiter 1	910,0	755,0	726,4	765,0	538,4	834,4	813,8	725,2	726,0	50,8
Arbiter 2	290,0	149,2	143,9	104,2	15,6	214,5	214,5	122,6	150,8	12,1
Arbiter 3	--	--	164,5	285,2	20,7	--	--	--	199,5	17,7

Table 11-4: Handshakin average time for 32 cores

Is in this part of the analysis where is easier to observe the results in the behavior of the arbiters. Observing the tables is possible to see the average handshaking time for the different scenarios. In the tables is possible to find empty blocks, the reason of it, will be explained now.

Starting this analysis for the scenarios with 4 cores is possible to confirm that for the first arbiter the scenarios with more activity are 1, 4, 6 and 7. In the case of the Arbiter 2 and Arbiter 3, due to the number of cores to manage is small, is a little difficult to appreciate the differences between each scenario due to the arbiters tend to behave in the same way. So in this case is not possible to appreciate clearly the behavior of the arbiters.

Working with 8, 16 and 32 cores, is possible to appreciate the big difference between all the scenarios and scenario number 10 (less stressed). In the case of the second arbiter, due to the queue of the system is bigger, is possible to see in the table 10 that start to appear differences between the average handshaking time for the scenarios. This difference is especially easy to appreciate in the scenario 1. In addition, in this point, is necessary to signalize the different behavior of the arbiters with the scenario number 4. For the first arbiters is possible to say that this scenario is one of the most stressed, otherwise for the round robin arbiter the scenario number 4 is considerate non-stressed . In addition, is not difficult to see that the first arbiter is much slower than the second one, and observing the circuits is possible to appreciate that the first arbiter uses two flip-flops per input. It implies that the delay of the output in the first case is almost 2 clock cycles. In the other hand, in the second arbiter is only used the 1 flip-flop per input, so the minimum time that the arbiter will spend is 1 clock cycle.

Is important to say that the analysis of the Arbiter 3 is very different that Arbiter 1 and 2. This difference is because the way to work of Arbiter 3. Arbiter 1 and 2 work with all the cores, on the contrary Arbiter 3 works with priority concept, and it means that the cores that the user signalize will have always preference for the arbiter. So, in the simulations different priority inputs have been used. And the results have been very satisfactory due to the cores which had priority were attended in only 1 clock cycle. One important thing to say, is that when there is a priority in a very active core, the rest of the cores will be ignored, and this is the reason because in the tables are not all the holes filled.

In addition, analyzing the second arbiter the obtained conclusion is that if there are n processes in the ready queue and the time slice is q ,the behavior of the arbiter can be modeled by

$$\frac{1}{n}$$

and each process would wait no longer than nq time units until its next quantum.

A more realistic formula would be:

$$n(q + \theta)$$

Where θ is the context switch overhead. So, for practical purposes, it is desirable that the context switch be negligible compared to the timeslice.

The performance of the Round-Robin algorithm depends heavily on the size of the quantum. If the quantum is very large, the Round-Robin algorithm is similar to the First-Come, First-Served algorithm. If the quantum is very small, the Round-Robin approach is called processor sharing.

12 Summary

As is possible to appreciate in this project three arbiters have been designed in order to manage a memory bus, in addition has been designed the entire environment and verified with the software ModelSim and Synopsys Studio Vision.

With ModelSim have been created three kinds of arbiter and ten different scenarios for different number of cores to manage (4, 8, 16 and 32) as well. In addition, sixty types of cores have been designed too. In order to analyze the cycle time of the arbiters in the different scenarios have been designed test benches too.

With Synopsys Design Vision have been analyzed all the arbiters. Analyzing the obtained result is possible to confirm that Arbiter 2 and Arbiter 3 achieve the requirements due to using the library of 90nm; the designs can work in frequencies near to Giga Hertz with small number of cores to manage. On the contrary, Arbiter 1 is not an implementable design due to it spends a lot of area and the maximum work frequency is really low, for the future works one objective could be to optimize it using pipeline methods.

In the other hand, Arbiter 2 is really fast and compact, and in addition it is able to work in every kind of scenario, this is because this arbiter has been designed having in mind the Round Robin Algorithm. The results of this arbiter in the area of cycle time, have been satisfactory due to the average handshaking time is reduced a lot.

The last design Arbiter 3 has been a variation of Arbiter 2, but with the difference that Arbiter 3 uses the concept of priority in its design. Is possible to confirm that Arbiter 3 can not work in scenarios with a lot of activity, but this design is very optimal in order to reduce the handshaking time in non very active cores.

13 Bibliography

[Brown 2009] Stephen Brown. *Fundaments of digital logic with VHDL design*. McGraw-Hill, 2009

[Ashenden 1990] Peter J. Ashenden. *The VHDL Cookbook*. Dept. Computer Science University of Adelaide South Australia, 1st Edition

[Bergeron 2003] Janick Bergeron .*Writing Testbenches: Functional Verification of HDL Models*. Springer, 2nd Edition.

[Fischer 2007] Philipp Martin Fischer, P. Pirsch (2007).Desig and Implementation of a Scalable Simulation Framework for Multicore System-on-Chip Architectures. Institut für Mikroelektronische Systeme Universität Hannover

[Wikipedia 2009] Wikipedia(2009) Round Robin Scheduling- Wikipedia the free encyclopedia. http://en.wikipedia.org/wiki/Round-robin_scheduling. [Wikipedia 2009]

[Synopsys 2008] RTL-to-Gates Synthesis using Synopsys Design Compiler (2006) http://csg.csail.mit.edu/6.375/6_375_2008_www/handouts/tutorials/tut4-dc.pdf

[SynopsysECE 2008] ECE 551 Design Vision Tutorial (2007) http://eceserv0.ece.wisc.edu/~morrow/ECE551/tutorials/DesignVisionTutorial_f07.pdf

[Modesim2003] ModelSim SETutorial Version 5.7 f (2003) http://pages.cs.wisc.edu/~markhill/cs552/Fall2006/handouts/se_tutor.pdf