

Xiang Li

January 21, 2009

This page is intentionally left blank

Contents

5	WISHBONE Specification and Conmax IP Core	5
5.1	Importance of Interconnection Standard	6
5.2	WISHBONE in a Nutshell	8
5.2.1	Overview	9
5.2.2	WISHBONE Interface Signals	11
5.2.3	WISHBONE Bus Transactions	15
5.2.3.1	Single Read/Write Transaction	15
5.2.3.2	Block Read/Write Transaction	18
5.2.3.3	Read-Modify-Write (RMW) Transaction	20
5.2.3.4	Burst Transaction	21
5.3	Conmax IP Core	32
5.3.1	Introduction	32
5.3.2	Conmax Architecture	32
5.3.3	Register File	33
5.3.4	Parameters and Address Allocation	34
5.3.5	Functional Notices	35
5.3.6	Arbitration	36
	References	38

This page is intentionally left blank

Chapter 5

WISHBONE Specification and Conmax IP Core

In this chapter WISHBONE and conmax IP core will be introduced. Because they are very important in the system, I reserve a separate chapter for them.

WISHBONE is the specification of a System-on-Chip (SoC) interconnection architecture. It is a bus standard like ARM's AMBA. It defines the interfaces of IP cores, therefore specifies how the cores should communicate with each other. Further, this influences how the IP core network looks like.

The WISHBONE is adopted to be the interconnection standard used in our thesis project. This is because it is the official standard suggested by the Opencores organization [1], and most open cores from there support (and only support) the WISHBONE standard.

If saying the WISHBONE is a blueprint, conmax is a building. The WISHBONE interCONnect MAtRiX IP core (CONMAX¹) is a real IP core that implements a matrix² interconnection that complies with the WISHBONE standard. In our project, what we did was just connecting all other IP cores to the conmax, which helped us to control the data traffic and handle bus transactions in the system.

Actually everything about the WISHBONE and the conmax are fully included in their specifications [2, 3]. I'd very like to finish this chapter here and ask my readers to study the documents themselves. However I know they will kill me if I really do. So I will still introduce the WISHBONE and

¹The capital characters compose the abbreviation as CONMAX. I have no idea why it abbreviates like this. Maybe sometimes giving a name never needs a reason.

²It is also called crossbar switch structure.

the conmax, but in the way other than just copying texts from the official specifications. I will cover the most contents of the WISHBONE and the conmax, add my own explanations, meanwhile highlights the things that are not so clearly described in the documents.

The intention of this chapter is to help understand the WISHBONE standard and the conmax IP core easier, as well as realize how to use them when needed.

5.1 Importance of Interconnection Standard

Before everything goes into detail, it is necessary to stress the importance of the interconnection standards, which is the WISHBONE in our case.

Nowadays, people have gradually realized that interconnection architecture is the most significant in an electronic system, even more than processors. There are several reasons listed below:

1. First, the bus or the interconnection is becoming the bottleneck of the system performance.

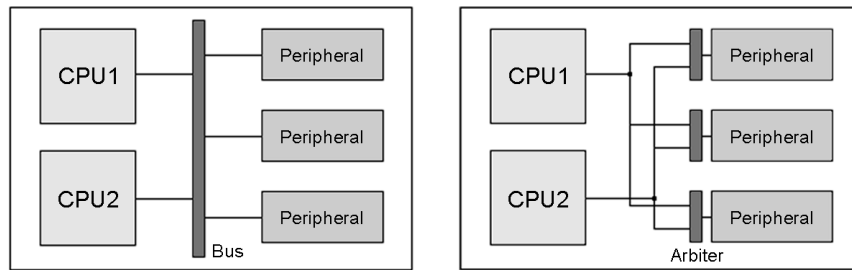
In the past, it was the processor that limited the system performance. One of the methods to improve was by increasing the system frequency. So my first PC was working at 100MHz while most computers now are in GHz level.

But it turned out that increasing the CPU frequency was not always helpful. There are at least two serious drawbacks:

- Higher frequency consumes more power.
- Most peripherals cannot catch up such high speed. As a result, processors spend a lot of time in the idle state waiting for the peripherals to complete an operation while doing nothing.

As a result, multiprocessor systems appeared. By more than one processor, peripherals can be driven in parallel. And in theory the frequency can lower down because the work is now shared by each processor. So it is the trend that the multiprocessor structure will become popular, like Intel's Duo or Quad processor products for sale at the moment. However another problem arose, that the traditional shared bus limited the performance hugely in multiprocessor systems, as Figure 5.1 shows.

In Figure 5.1(a), only one processor is able to access the peripherals at a time. The other have to wait until the first one releases the bus.



(a) Traditional bus limits performance

(b) Matrix structure is better

Figure 5.1: Interconnection is important in multiprocessor systems

This has actually no difference than a single processor system. Figure 5.1(b) shows an improved version. The bus is replaced by a matrix interconnection. Now two processors can access different peripherals in parallel, but still need to be arbitrated when accessing the same peripheral.

In fact, many systems have more than two and even dozens of processors. And the number is still increasing. As a result, the traditional shared bus is evolving more and more like a network. So as we can see, how to communicate efficiently in the multiprocessor system has emerged as a critical problem. Carefully designing architecture of an interconnection to get a maximum throughput for the IP core network become an attracting issue that engineers care about now.

2. Second, IP cores need standards for their interfaces. This is easy to understand. Lots of companies produce IP cores. If there is a standard that everyone follows, all of the cores will get connected easily. This will definitely accelerate a lot on developing the new products. So people need a unified interconnection architecture.
3. Third, a standard for the interconnection has great market potential. Think about it. If a company owns a standard which takes the dominant role in the market, all other vendors have to ask for licenses to make their products with standard interfaces. Furthermore, every time when the standard is updating, the owner will get chances to lead the direction of the development in future.
4. Sometimes, the standard could even be the first criterion of IP core selection. For instance in our thesis project, as we used the WISHBONE, all the open cores chosen for the system have to be WISHBONE compliant. Some of IP cores with different interfaces were given up because they cannot adapt into the system easily, although they might have very good quality.

So now we know how important the interconnection architecture could be. Then I must emphasize a special feature the WISHBONE has: that it is in the public domain.

The WISHBONE is in the public domain means it is not copyrighted, which is another way of saying anyone could do anything with the WISHBONE without any limitation, but of course no warranty at the mean time. This is a great gesture from the developers of the WISHBONE, because they gave up the ownership they could have, so that all other people benefit since they are allowed to develop new products based on the WISHBONE without asking for a license. They could even turn the new products into their own proprietaries because the WISHBONE is not copyrighted. Besides, the WISHBONE will be always for free. Comparing to the ARM's AMBA, it is a copyrighted open standard which do not have to pay right now, but ARM never promise it will be free forever. So it is possible that after next upgrading you may have to pay for the license of each copy of your products which apply the AMBA standard. The public WISHBONE standard also gives a meaningful push to the open core community. Now developers can happily design new open cores by following the WISHBONE standard, with no trouble on interface compatibility and with less worry on legal problems.

Above all, I have introduced how important the WISHBONE is. But it is a shame that in our thesis actually all we did was just including a con-max core into our system, no further investigations. However, it is really a good starting point to research the interconnection architecture with the WISHBONE, like how to adapt the standard into a Network-on-Chip (NoC) system meanwhile keeping a certain throughput, or make benchmarks between the WISHBONE and other interconnection standards, etc.

5.2 WISHBONE in a Nutshell

In this section the WISHBONE standard will be introduced. It can be seen as an explanation or a supplement to the official specification when you feel the descriptions in the standard are not that easy to understand. So you might want to read my thesis after you finish the WISHBONE specification. But before that, here're several suggestions that may help you to read the official specification.

1. Start reading with the tutorial in the appendix. It is very good to give a quick overview of the WISHBONE. Also note that you may not have to spend much time on the examples of the appendix of how to implement a WISHBONE interconnection, because we use the

conmax in our project, which will be introduced in the later section of this chapter.

2. There are lots of Rules, Recommendations, Suggestions, Permissions and Observations in the specification. It might be helpful to skip them all at the first time you read the document. And only read the Rules at the 2nd time.
3. All the timing diagrams from section 3.2 to 3.5 are not good enough to understand. At least they confused me a lot at the beginning. So give up those figures if you feel in trouble because I will redraw them in my thesis.

5.2.1 Overview

The WISHBONE is not a complicated standard, but it contains almost all main features that the other bus standards have. And if consider it was published at the year 2002, it was really a brilliant work at that time, although unfortunately it did not keep upgrading¹.

One of the major works that the WISHBONE did was to define interface signals. If an IP core supports all basic signals with correct functions that the WISHBONE specified, it will be compatible with other IP cores that have WISHBONE interfaces. So to this extent, if you can understand those signals and how they work, you will master the WISHBONE standard.

With the signals, the WISHBONE designed a set of protocols that standardizing how the interfaces communicate, i.e. how the data is packaged and sent/received by the signals. These are called “bus transactions”. There are 4 types of transactions described in the specification: single, block, RMW, and burst.

In fact, it is the IP cores’ responsibility to implement such signals and protocols. To be WISHBONE compatible, the cores have to include specialized logic to provide interface signals, as well as to be able to send and recognize bus transactions correctly. Due to the implementation of the WISHBONE interface logic is tightly coupled with IP functions that may vary from one core to another, there isn’t a universal solution of how to design a WISHBONE interface. So I will not introduce the details of the interface implementation in this chapter. However, all IP cores used in the thesis project are with WISHBONE interfaces. Their source codes can be taken as examples to study the interface logic.

¹Today (2008/10/30) the latest WISHBONE standard is still the Revision B.3 released at September 7, 2002.

When more than one WISHBONE compliant IP core is used to form a larger system, a WISHBONE network is constructed. In the Appendix A.2 of the specification (page 96–99) [2], 4 types of the interconnections are introduced. They are the most common ways to compose a WISHBONE network. But of course there are more solutions than the four. Users can design interconnections with new structures, as long as the solutions guarantee all bus transactions are transferred correctly and efficiently in the interconnection. The way to organize a network is also a good issue of the WISHBONE to research.

There are many IP cores already built to help constructing the WISHBONE networks. In our thesis project, we didn't spend much time on designing an interconnection. Instead we used the conmax IP core which implements a crossbar switch structure. With the conmax, we can easily create a network just by connecting all other cores to it. This saved us lots of time and made the system more reliable.

To sum up, I think the WISHBONE standard can be divided into 3 aspects: the signals, the transactions and the interconnections. Both the signals and the transactions are relatively stricter defined by the standard that all the compliant cores have to follow. While the interconnection is more flexible to implement that depends on the situations of different projects. I made a figure to give an overview of the WISHBONE.

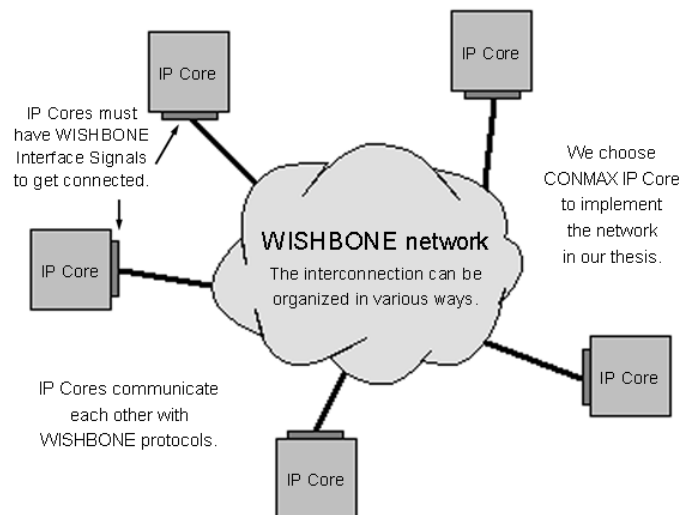


Figure 5.2: Overview of the WISHBONE

5.2.2 WISHBONE Interface Signals

The WISHBONE defines 2 types of interfaces: master and slave. An interface has to be either a master or a slave. Masters always start actions, like request for reads or writes. Slaves always respond to the requests. A connection can be made only between a master and a slave, but not between the same types.

This implies that the signals that located at masters and slaves are in pairs, or would rather say they are complementary. For instance, if a master has a signal named `ADR_O` which outputs an address, there should be an `ADR_I`¹ in the slave which receives the output. So to simplify ONLY the signals from the master side are introduced below.

The WISHBONE standard describes a lot signals, and sorts them in an ascending order. This is not good to understand for the beginners to the WISHBONE. So in the thesis I'd rather divided the signals into 6 groups based on how frequently they are used. Further each group can be marked as basic or extended, which means when designing an IP core, the designer do not have to implement all the signals specified in the standard, but only the signals in the basic groups. Thus, some WISHBONE compliant IP cores may only have minimum basic signals, while some others may have more extended signals to support advanced WISHBONE functions, like block read/write, burst etc.

Here is the 6 groups of the WISHBONE signals. Only the group 1 and 2 are the basic signals that every WISHBONE compliant IP core has to implement.

Group 1

`CLK_I`: The system clock

`RST_I`: The system reset

Clock and reset are the most basic signals that all WISHBONE interfaces have. They are the only two signals that always as input type whatever at the master side or the slave side.

All of the `CLK_I` and all of the `RST_I` are connected together in a WISHBONE network. So all IP cores in the network share the same clock source, and all of them reset at the same time.

Sometimes an IP core may have more than one reset input, in such cases

¹The “I” in the `CLK_I` stands for an input port. It is an output if the port is named as “`XXX_O`”.

normally all reset inputs should be connected together so that only one reset signal drives the whole system.

The WISHBONE is a synchronous interconnection standard, which means all the IP cores in the system examine inputs as well as change outputs at each clock rising edge. This is clearly stated in the specification when describing the WISHBONE features and objectives. In page 9, one of the features is described as “Synchronous design assures portability, simplicity and ease of use.” And in page 12, the last several objectives are about synchronization, like “to create a synchronous protocol to insure ease of use, good reliability and easy testing. Furthermore, all transactions can be coordinated by a single clock.” etc. [2]

Group 2

- STB_O:** When the STB_O=‘1’, it means either a read or a write operation is ongoing. Meanwhile all data signals like the ADR, DAT and WE etc., are valid.
- CYC_O:** The CYC_O keeps high during the period of the whole bus transaction. More than indicating bus transactions, it is also used to request grants from bus arbiters when multiple masters are accessing one slave at the same time.
- ACK_I:** Acknowledgements from the slaves. When a read or write operation is finished, the slave informs the master by giving a one clock cycle’s ACK back. When a master finds the ACK=‘1’ on a clock rising edge, it knows the current read/write operation is done and it’s OK to start the next one.
- ADR_O:** The address to access.
- WE_O:** Indicates either read or write. It is a write operation if WE=‘1’, else read.
- DAT_O:** Data outputs from a master when writing to a slave.
- DAT_I:** Data inputs to a master when reading from a slave.
- SEL_O:** Indicates which fragments of data are valid. For instance, for a 32-bit bus, the SEL_O is 4-bit width. If SEL_O[3:0]=“1000” during a read, it means the master only wants the highest byte of the data DAT_I[31:24]. All other bits are not valid and won’t be processed.

The 8 signals in the group 2 are enough to perform basic WISHBONE functions. Plus the CLK_I and RST_I, all the 10 signals are necessary for every WISHBONE compliant IP core.

Group 3

ERR_I: Indicates errors

RTY_I: Retry signal, ask for a repeat of the last read/write operation

Some IP cores have these signals in their interfaces. They are similar to the ACK_I but have different meanings. Once a read/write operation is successfully finished, a one cycle ACK returns, but if it isn't, the slave may give a one cycle ERR to tell the master that an error occurred in the last read/write operation, or send a RTY to ask for an retry.

To enable this function, both the master and the slave have to support it, i.e. the master should have the ERR_I and the RTY_I and the slave has the ERR_O and the RTY_O. This implies certain functional logic has to be designed in the IP cores. But due to these features are not compulsory according to the WISHBONE standard, so far I haven't seen a pair of cores that both support the ERR and the RTY signals.

If a slave doesn't have an ERR_O or a RTY_O, the ERR_I and the RTY_I of the master can be wired to ground. If a master does not support these signals, the ERR_O, RTY_O and ACK_O of the slave may be connected together with an OR gate and then send to the ACK_I of the master. But in such case the ERR and the ACK signals are treated as an ACK and ignored.

Group 4

TGD_I: Tag of input data

TGD_O: Tag of output data

TGA_O: Tag of address

TGC_O: Tag of bus cycle

These 4 signals are called tag signals because they are attached with other signals to provide extra information, just like tags. For example when a master is sending data in serial to a slave, if some of the data is more special than the others, it could be marked by the TGD_O which is sending at the same time, so that the slave can recognize the special data when receiving the TGD_O signal. Or for another example, when the CYC='1', the value of TGC could be used to determine which kind of transaction is transferring, for instance the 00, 01, 10 and 11 could be used as tags to stand for single, block, RMW and burst transactions respectively.

An interesting thing is that the WISHBONE doesn't specify the 4 signals in detail, like the width of the signals, or the meanings of the data patterns. This leaves a great freedom to users on how to utilize the signals. In principle,

it is totally possible to define a custom protocol for a system, as long as both the master and the slave understand the tags in the same way.

The tag signals need specialized logic design in both the master and the slave IP cores too. Again, these signals are not mandatory. Most existed WISHBONE IP cores do not have them.

Group 5

LOCK_0: Indicates the current bus transaction is uninterruptable.

LOCK is another signal almost never in use.

The usage of the signal is not clearly described and only mentioned in the chapter 3.3 of the WISHBONE specification (page 51) [2]. According to the waveform in the figure, when the block read/write transactions happen, the LOCK could be used together with the CYC to hold a transaction uninterrupted.

The information from the specification seems not enough, so for curiosity I searched the WISHBONE forum for the LOCK and found a message [4] from Richard Herveille, the author of the WISHBONE specification, which explains a little more about the LOCK. The message is copied below:

What is the purpose of the LOCK_0 signal? From the description it says that it indicates that the bus cycle is uninterruptible. However, the statement:

Once the transfer has started, the INTERCON does not grant the bus to any other MASTER, until the current MASTER negates [LOCK_0] or [CYC_0].

If deasserting CYC_0 causes the lock to end, then this signal doesn't really do anything more than asserting CYC_0 by itself, does it?

[rih] Not really. CYC is a bus-request signal. If it's asserted it validates all other signals. So if CYC is negated, LOCK is invalid. If a higher priority bus master asserts CYC then the bus arbiter might grant that master the bus. Asserting LOCK prevents this. So far nobody uses LOCK.

Group 6

CTI_0: Cycle type identification

BTE_0: Burst type extension

These 2 signals are even less used, but clearly defined in the chapter 4 of the specification. They are designed for the WISHBONE registered feedback

bus cycles, i.e. the burst transactions. Basically they are similar to the tag signals which also provide extra information. By the CTI and the BTE, the slave knows the status of the burst so that can be prepared to handle it. The 2 signals will be discussed again later in the burst transaction section.

5.2.3 WISHBONE Bus Transactions

After the introduction to the signals, in the next step let's focus on how the masters and the slaves communicate through the WISHBONE interconnection.

In the WISHBONE specification, each process of data transferring is called a bus cycle. There are 4 types of bus cycles defined in the specification¹, which are single, block, RMW and registered feedback bus cycles.

In my thesis however, the name of the “bus cycle” is replaced by the “bus transaction”, because when saying “cycles” you probably need to distinguish between clock cycles and bus cycles. The 4 bus cycles here are named as single, block, RMW and burst² bus transaction, respectively.

Besides, each action of read or write is called an “operation”. A single transaction contains only one read or one write operation, while the block, RMW and burst transactions can have multiple operations within one transaction.

According the specification, all 4 types of transactions are optional. But to be WISHBONE compliant, an IP core has to support at least 1 type of transactions to communicate with the others. In fact almost all IP cores choose to implement the single transaction because which is the simplest, whereas the other 3 are merely used. For example in our thesis project, all the IP cores only work with single transactions.

5.2.3.1 Single Read/Write Transaction

Single read/write transaction is the most frequently used. Each transaction contains only 1 read or write operation initiated by the master. The slave responds to the request of the master by returning wanted data in case of reading, or accepting exported data in case of writing.

Figure 5.3 gives an example of the block diagram of the connections of the WISHBONE signals between a master and a slave. The diagram helps to

¹The first 3 types are in the chapter 3, and the last one in the chapter 4 of the WISHBONE specification.

²The name of burst is more popular than the registered feedback.

demonstrate how the bus transactions are transmitted from the master to the slave through the connections. Readers can just imagine that all the waveforms you are going to see below are happening over the wires in the figure.

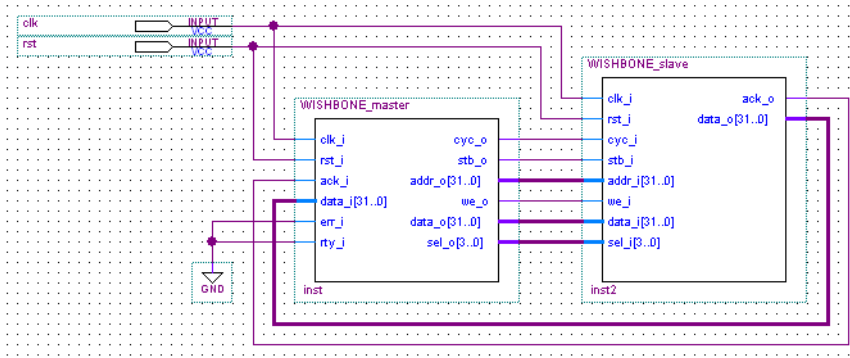


Figure 5.3: An example of WISHBONE signal connections

Figure 5.4 depicts normally the single read/write transactions could be. Four transactions are contained in the figure.

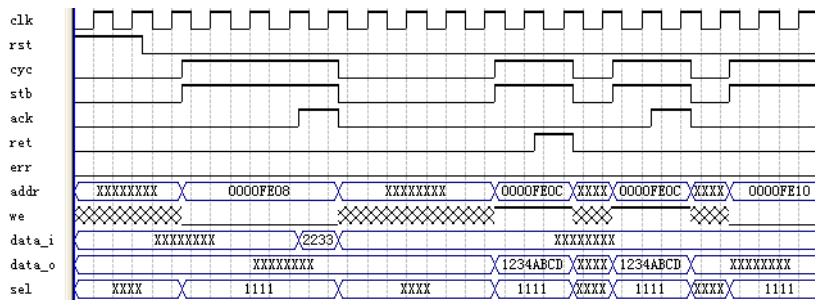


Figure 5.4: An example of WISHBONE single transactions

In the 3rd clock rising edge, the STB became '1' to inform the starting of a signal read transaction (WE='0'). On the next cycle, since the slave did not respond, the master held all the signals unchanged. After a while, the slave answered by asserting ACK to '1'. On the next rising edge, the master detected this ACK and latched the returned data wanted to read. This is the first transaction.

After another 3 cycles, the master was ready again. This time it wrote data to the slave. The slave gave response as soon as possible at the following clock cycle. However, somehow the slave didn't finish this operation correctly, so it returned a RET to request for another try. The master then repeated the

operation again, and got the result (ACK) successfully this time. These are the 2nd and 3rd transactions.

After that, the master started another transaction. As limited by the figure, I cannot draw all of the waveform. The master has to keep all the signals until it gets the response from the slave. It could be even forever if the slave doesn't respond at all.

So far, I hope you've had general idea about how the WISHBONE works with the signal read/write transactions. Except for that, there are several important notes listed below:

1. One of the most important things needs to notice is that the slave always gives exactly 1-clock-cycle ACKs. This is sort of one-way handshake protocol, i.e. the master holds the sending signals and observes the replies from the slave all the time, but the slave only sends back 1-clock-cycle ACK signals and don't care if the master receives the ACKs or not.

Please remember that the master will be confused when they see an ACK signal with multiple cycles. This will be interpreted as several operations are done.

Because of the one-way protocol, if your WISHBONE interconnection is so complicated that somehow a master could miss an ACK, the current bus transaction will be unable to finish and keeps forever. This is quite exceptional, but if such cases do appear, a watchdog inside the master may be considered to be designed, which forces to restart or skip the transaction after timeout.

2. The CYC signal should not be ignored, although as everyone can see the CYC and the STB have exactly the same waveforms in single transactions.

According to the specification, the slaves are only allowed to behave when $CYC=1$. This means the slaves only respond to the transactions when the logical AND of the STB and CYC is true.

For example, in complicated WISHBONE networks, the CYC is usually used to request for grants from bus arbiters. I've met the situation that the arbiter will broadcast bus transactions to all slaves except for delivering only the CYC signal to the right slave. So in such cases, the slaves may respond incorrectly if they don't check the input value of the CYC.

3. All the 3 signals of the ACK, RET, and ERR can be used to reply to the master, like the 2nd transaction is finished with a RET. But both

the master and the slave IP cores have to support the signals and the function. Usually they have only one ACK signal, because the RET and ERR are not mandatory by the specification.

4. In the first transaction, the master received the response after 4 clock cycles, but the delay cycles may not necessarily always 4. The slaves may need several cycles to process the data. No ACK will return until they are finished. Therefore, a bus transaction could keep for a long time because too much time is spent on waiting for the slow slave.

Similarly, when the masters receive ACKs, they may need some time to process data too. In such cases, there will be breaks between 2 transactions, just like the delay between the 1st and 2nd transaction in the figure.

In the best situation without any delay, the ACK will be set to '1' by the slave on the next rising edge that the STB asserts. And the master will start another transaction as soon as it gets the ACK. The 2nd to 4th transactions in the figure describe this situation.

5.2.3.2 Block Read/Write Transaction

The WISHBONE specification defines block read/write transactions to transfer more than one data in a bus transaction. It is almost the same as the signal transactions, only multiple single read/write operations are now encapsulated in one transaction. To indicate the current bus transaction is a block transaction, the CYC has to keep high during the whole transaction period.

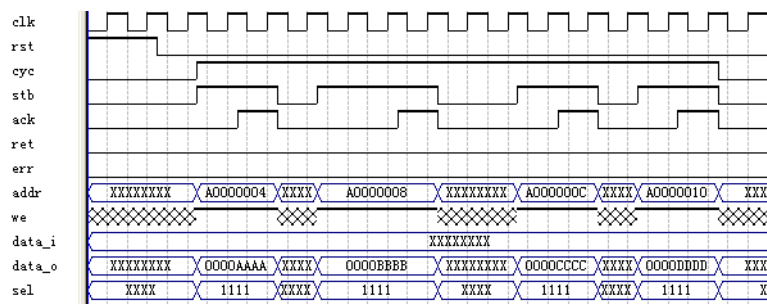


Figure 5.5: An example of a WISHBONE block write transaction

As Figure 5.5 shows, it is a block read transaction which includes 4 read operations. As we can see, now the CYC is keeping high for the duration of the transaction time, while the 4 read operations have no difference with single read/write operations.

According to the specification, block transactions must contain either all read or all write operations, but cannot have both types in one transaction. However, in my perspective it should not be a constraint. In principle both read and write operations are operations and the block transactions are actually a batch of single operations, so to this extent the slaves can always behave correctly if they are able to deal with the single read or write operations, without thinking about if the current block transaction is a block read or a block write transaction at all.

By the way, please don't mix up the "block read/write" in the WISHBONE and the "blocking read/write". They are quite confusing sometimes. The block read/write means to read/write a batch of data in a time, while the blocking read/write means the system will be stuck until the last read/write operation has been finished. For a simple example of the blocking read, when you are programming in C with the function `scanf()` to read from a keyboard, the program won't continue until you press some button.

Some people may wonder why we need the block transactions, if they look pretty much like the single transactions and essentially they do not increase the bus throughput. So I designed the following example to give my answer.

In Figure 5.6, there are 2 masters and 1 slave. Both the masters want to access the slave but only one of them is allowed to do so at a time. So there is an arbiter who makes the decisions. The arbiter gives grants to the masters according their `CYC` signals, like Figure 5.7 shows.

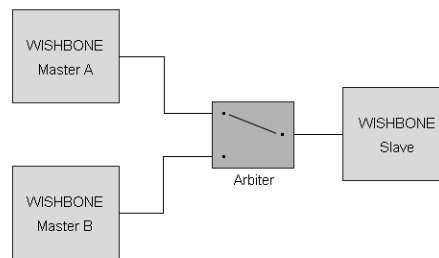


Figure 5.6: Block transactions are helpful in multi-master systems

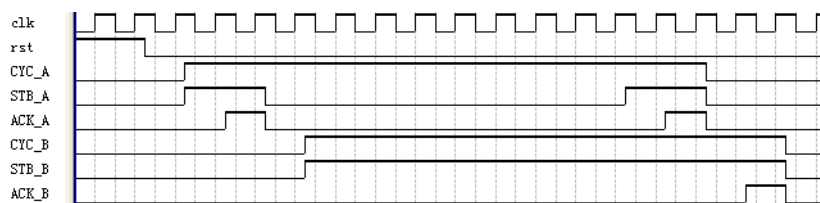


Figure 5.7: Master B is blocked by master A

In Figure 5.7, only the CYC, STB and ACK signals are given, because they are enough to describe the WISHBONE protocol. First, the master A started a block transaction and asserted its CYC at the 3rd clock rising edge. At that time, because no one took the bus, the arbiter gave the grant to the master A and connected it to the slave. In the following 2 clock cycles, one operation was done. However, the other operation from the master A was somehow delayed, so its CYC had to keep holding as this is a block transaction. Because the arbiter gave grants by judging to the CYC, the master A therefore possessed the network all the time during its CYC was '1'. As a result, the master B had to wait until the whole A's block transaction was finished, although it was ready since the 6th clock rising edge.

To summarize the example, the WISHBONE masters use the CYC to request grants from the arbiter when accessing slaves in multi-master systems. If the system has no preemption, i.e. once the grant is given to a master it won't be withdrawn if another higher priority master becomes ready, the CYC actually can be used to hold the line, until all the data from a master is transferred.

Usually the signals of the group 1 and 2 are enough to perform the block transactions through the WISHBONE network, but in the specification it also mentions other signals like the tag signal TGC or the LOCK, which should be involved in block transactions. The TGC could be used to identify which type of the transactions is ongoing. So the slave knows if the current transaction is a single or a block transaction. However I don't feel this is necessary. Because if a slave can process read/write operations correctly, it does not have to recognize what the current transaction is, since the block transactions can be seen as a series of single transactions from the slave's perspective. In the systems without preemption, the LOCK is useless too.

5.2.3.3 Read-Modify-Write (RMW) Transaction

The WISHBONE specification also defines a kind of transactions named Read-Modify-Write (RMW). It is said the RMW transactions are used for "indivisible semaphore operations" [2].

Far from the complicated name, the RMW transactions are fairly simple. In fact, a RMW can be seen as a block transaction with 2 different operations. The first one is a read operation, while the other is a write operation. So by the RMW transactions, we can easily read data, modify it, and then write back to the same address. The RMW waveform is showed in Figure 5.8.

In my opinion the RMW and the block transactions can be merged together. The block transactions are essentially a batch of bus operations which have

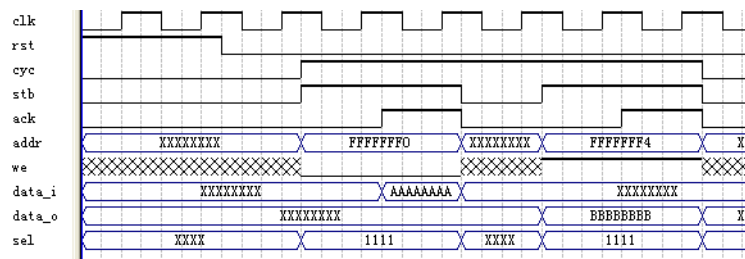


Figure 5.8: An example of a WISHBONE RMW transaction

to be either all reads or all writes. The RMW transactions contain 2 bus operations that a read followed by a write. If we redefine the rules of the WISHBONE specification to allow the block transactions to include any type and any number of operations, the RMW will become a subcategory of the block transactions.

Again, supporting to the RMW transactions is not a mandatory function for the WISHBONE interfaces. And there is not an IP core I've seen that works with the RMW transactions.

5.2.3.4 Burst Transaction

Throughput is always an important criterion to evaluate the performance of an interconnection architecture. Higher throughput stands for transferring larger amount of data in a certain time period. Or in case the bus width is given, it means to finish as many read/write operations as possible.

The WISHBONE interconnection struggles to achieve a good throughput too. This is why it spent the whole chapter 4 to describe registered feedback bus cycles, i.e. “burst transactions”.

The burst transactions are one of the four types of the WISHBONE transactions, which are different from the block transactions. In principle, the block transactions do not increase the throughput of a system. Sometimes they may even ruin the performance if a master holds a line too long but does not transfer data. But the burst transactions do improve the throughput, by a set of carefully defined schemes.

The main idea of the scheme is to inform the slaves in advance that they are going to be addressed again and again within a bus transaction, so that they will be prepared to respond continuously. At the same time the masters could initiate operations one after another without waiting for the responses from the slaves, because the slave is assumed to know the data is

sent continuously and be able to handle that.

For the single or block transactions, normally after initiating operations the masters have to stay and hold signals until an ACK feeds back. In the best case that communicating without any delay, the waveform will look like Figure 5.9.

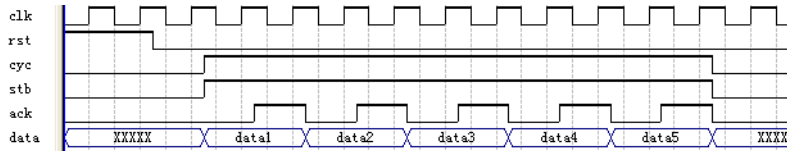


Figure 5.9: Maximum throughput with single transactions

In the figure, firstly the STB is asserted to start an operation. Then the slave replies as soon as possible on the next clock rising edge and give a valid ACK back. After the ACK is received, the master sends the next operation immediately. As we can see in this scenario, each operation takes 2 bus cycles to finish. This means we can get 50% bus utilization in the best case.

To get a throughput yet higher, the burst transactions are used. A demo waveform is showed in Figure 5.10. In the figure, the master starts a request by asserting the STB at the 3rd clock rising edge. Meanwhile it somehow tells the slave that this is a burst transaction. At the 4th clock rising edge, the slave receives the message and gets prepared to handle the burst. By giving back a one-cycle-ACK the slave indicates that it is ready for one more read/write operation. The master sees the ACK at the 5th edge and continues. Then another e operations are done from the 5th and 7th clock cycles.

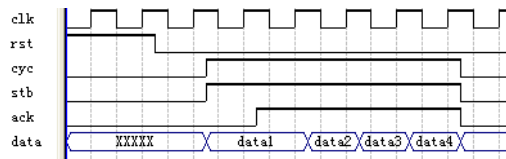


Figure 5.10: Maximum throughput with burst transactions

As we can see, N operations are now completed in $N+1$ clock cycles. So the bus utilization now is $N/(N+1)$. If the N is a very large number, the utilization will theoretically approach 100%. So the burst transactions, if they can perform properly, are the best scheme to achieve the top throughput.

By the way, the burst transaction is also referred as the advanced synchronous cycle termination at the beginning of the chapter 4. The WISHBONE specification compares 3 different ways of terminations, asynchronous, synchronous and advanced synchronous, and points out that the advanced synchronous is the best. But I feel that the conclusion is not well presented, because the table in the specification shows that the asynchronous cycle termination is always the smallest of the three. The writer forgot to stress an important point again here, that the WISHBONE is a synchronous bus standard. Even though asynchronous circuits are considered faster, we have to pick up a regular and stable way to communicate in the WISHBONE. So finally the advanced synchronous is chosen because it is the better one in the synchronous schemes.

More WISHBONE signals are involved in case of the burst transactions, because whose protocols are more complicated than the WISHBONE classical single, block and RMW transactions. These signals are the CTI and the BTE, i.e. the signals of the group 6 described in the previous section.

The CTI is used to identify the burst transactions. At every rising edge, the slaves examine the value of the CTI to see if preparations are needed to execute to handle the burst transactions. If the CTI is “000”, the current transaction is a classical transaction. No need for special preparing. If the CTI is a “001”, the current transaction is a constant burst, or if “010”, it is an incrementing burst. When a burst is about to terminate, the master will give a CTI with “111” to tell the slaves that go back to normal state.

There are 2 kinds of burst. The constant burst always reads or writes the same addresses. This is useful to access FIFOs or certain I/Os which have volatile data. While the incrementing burst contains the operations targeted to adjacent addresses. It is particularly designed for reading or writing a block of data from/to memories.

When the incrementing burst is used, one more BTE is needed to indicate how the address grows. The definition of the BTE is clearly described in the table 4-2 and 4-3 of the WISHBONE specification.

Now it is time to go through the details of how the burst transactions work. To avoid describing by just boring texts, I designed 4 examples with waveforms, which can be seen as supplements to the WISHBONE specification.

The first example is a constant writing burst, which is showed in Figure 5.11. The first line of the waveform marks the number of each clock rising edge. Below that, only the related signals are drawn. I ignored some WISHBONE signals in the picture which are not necessarily needed to explain the protocol. As we can see now the CTI and the BTE are included for burst

transactions. The value “CON” of the CTI shows this is a constant burst, and the “EOB” stands for End-Of-Burst. The BTE is not needed for the constant burst transactions, so its value is not cared about (‘X’) during the whole period. Besides, the signal WE is always ‘1’. This indicates the current burst is a burst write.

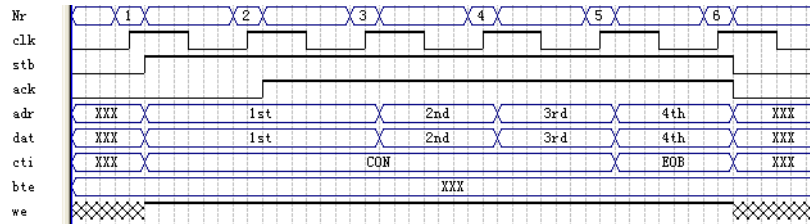


Figure 5.11: An example of a constant writing burst transaction

Edge 1:

Master: The master is ready to initiate a burst transaction. It sets the STB to ‘1’ to start the transaction. Meanwhile it gives 1st valid address, outputs the 1st data to be written, and sets WE to ‘1’. More important, it asserts the CTI as CON to inform the slave this is a constant burst.

Slave: The slave does nothing at the edge 1 because it cannot see anything from its perspective.

Edge 2:

Master: The master checks the value of the ACK to see if the slave replies. Since the ACK is still ‘0’, the master holds all signals unchanged for one more clock cycle.

Slave: The slave now receives the information about the burst from the master. The slave checks and the CTI, and knows it is a constant burst. Because the slave is idle and can handle the 1st data of the burst, it asserts the ACK to inform the master. This ACK means: the slave is capable for accepting the 1st data, please continue. But NOTE that in burst transactions actually the 1st data is not processed here, but at the next clock rising edge.

Edge 3:

Master: The master checks again the value of the ACK. Now as the ACK is ‘1’, the master knows that the slave can take care of the 1st data of the burst and wants more. So it puts the 2nd data onto the bus. Because the constant bursts always access one address, the value of the 1st, 2nd, 3rd and 4th addresses are actually the same. I marked them in sequence is just because I want to match

the addresses with the data.

Slave: The slave firstly latches the 1st data at the edge 3. The 1st data is accepted now. Meanwhile the slave checks the CTI, and knows it is still a CON. As the slave is still capable to receive more data, it keeps the ACK as '1'.

Edge 4:

Master: The master checks the ACK, and finds which keeps as '1'. The master sends another data to the slave.

Slave: The slave latches the 2nd data. The slave checks the CTI, and knows it is still a CON. The slave keeps asserting the ACK because it is capable to handle more data.

Edge 5:

Master: The master checks the ACK, and finds which keeps as '1'. The master sends another data (4th) to the slave. Because the master knows the 4th data is the last one of the burst, it sets the CTI to EOB.

Slave: The slave latches the 3rd data. The slave checks the CTI, and knows it is still a CON. The slave keeps asserting the ACK because it is capable to handle more data.

Edge 6:

Master: The master checks the ACK, and finds which keeps as '1'. The ACK shows that the last data of the burst will be taken care of, so the master de-asserts all signals and terminates the burst.

Slave: The slave latches the 4th data. The slave checks the CTI, and notices that it is an EOB now. So it knows the 4th data will be the last one and does not need to assert the ACK anymore.

After the first example I hope you have understood more about the burst transactions. The next one is another example of an incrementing read burst transaction, showed in Figure 5.12.

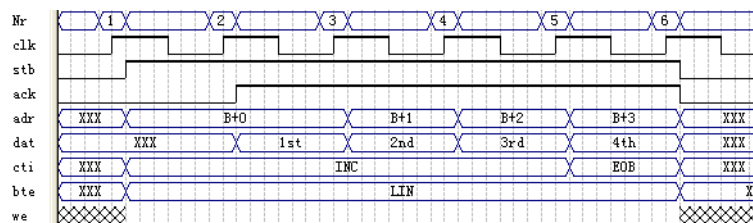


Figure 5.12: An example of an incrementing reading burst transaction

Now the value of the CTI is no longer CON but INC, which shows that the current transaction is an incrementing burst. The BTE is now needed for incrementing bursts to present how the address grows. The LIN means the address increases in a linear manner. So now you see the addresses are B+0, B+1, B+2 ... The “B” represents the “Base” address. Also note that the “B+1” does not mean exactly B plus 1, but rather stands for the address next to the base address. For instance, if the data bus is 32-bit width, the value of the next address B+1 would be probably the base address adds 4.

Edge 1:

Master: The master starts the burst transaction. The master sets CTI to INC and the BTE to LIN. Because this is a read transaction, the value of the data_{in} is unknown by the edge 1.

Slave: The slave does nothing because it cannot see the transaction has initiated.

Edge 2:

Master: The master checks the ACK and finds which is ‘0’. So the master holds all signals.

Slave: The slave sees all the signals and knows this is an incrementing read burst. The slave feels capable to handle the burst. So the slave 1) returns the 1st data according to the address B+0; 2) sets ACK to ‘1’ to inform the master to keep transferring; 3) calculates the next address based on the value of the current address and the BTE.

Edge 3:

Master: The master checks the ACK and finds which is ‘1’, so it knows the slave is capable to handle more data. The master puts the next address B+1 and other signals onto the bus.

Slave: The slave checks the STB, CTI, BTE, and knows the burst is still happening. The slave feels capable to handle more data. So the slave 1) returns the 2nd data according to the address B+1, which is the address calculated by the slave itself at the previous edge 2; 2) continues setting ACK to ‘1’ to inform the master to keep transferring; 3) calculates the next address based on the current address (B+1) and the BTE.

Edge 4 is similar to the Edge 3.

Edge 5:

Master: The master checks the ACK and finds which is ‘1’, so it knows the slave is capable to handle more data. The master puts the

next address B+3 and other signals onto the bus. The master knows this will be the last one of the burst, so it changes the CTI from INC to EOB.

Slave: The slave checks the STB, CTI, BTE, and knows the burst is still happening. The slave feels capable to handle more data. So the slave 1) returns the 4th data according to the previously calculated address B+3; 2) continues setting ACK to '1' to inform the master to keep transferring; 3) calculates the next address which is the address B+5, although this address will not be used.

Edge 6:

Master: The master checks the ACK and finds which is '1', so it knows the slave is still working and the last data is ready to read. The master latches the 4th data. The master de-asserts all signals to terminate the burst.

Slave: The slave checks the STB, CTI, BTE, and knows this is the end of the burst. So there is nothing to do now except for de-asserting the ACK to '0'.

So far we have seen 2 examples. I explained what the master and the slave actually do at every clock rising edge. After the specific descriptions now it is time to summarize some general rules about the WISHBONE burst transactions.

1. According to the specification, all burst transactions have to be either read or write, i.e. a burst transaction must contain either all read operations or all write operations, but cannot have both within one burst.
2. The CTI and the BTE signals are used to assist the slaves to indentify the type and the status of the current burst. All burst end up with an EOB in the CTI.
3. The slaves behave different between when it is a read burst and when it is a write. If it is a write, the slaves don't do anything special than just latch the written data at every rising edge. However if it is read, the slaves need to pre-calculate the next address based on the current address and the value of the BTE. And then use the calculated address to access the next data for the masters.
4. The WISHBONE specification does not mention that the slaves have the ability to predict the next address automatically. But this is true. I found a message from the forum of the opencores.org which said so, And in this way all the waveforms in the specification are well explained.

5. The masters assert one clock cycle of the STB is saying that there is more data to read or write. The slaves assert one clock cycle of the ACK implies the last operation has been taken care of and they are ready to handle the next read or write.
6. Both the masters and the slaves are allowed to break the current burst, i.e. to insert wait states (WSM or WSS) at any time during the burst. This will be described later.

I made a general algorithm for the WISHBONE burst transactions, which lists all the things in detail that the masters and the slaves should do at every clock rising edge to perform bursts.

For masters:

Firstly at a clock rising edge, initiate a burst by asserting the STB, CTI, BTE and other signals.

After that at every clock rising edge, check the value of the ACK.

IF: the ACK='0', set the STB to '1' and hold all other signals unchanged for one more clock cycle.

ELSE: the ACK='1',

IF: the current CTI is not an EOB, set the STB to '1', meanwhile

IF: it is a write burst, output the next data to the slaves.

ELSE: if it is a read burst, latch the current data and output the next address.

ELSE: if the CTI=EOB, set the STB to '0', meanwhile

IF: it is a write burst, de-assert all signals to finish the burst.

ELSE: if it is a read burst, latch the last data and de-assert all signals to finish the burst.

IF: no wait state is needed, then that's it. Go to the next clock rising edge.

ELSE: if the masters currently are unable to handle more data, a wait state is inserted by resetting the STB to '0'. All other signals can output as 'X'. However, the masters should still go through the previous IF-ELSE block, and somehow remember what the output signals should be. In case of read bursts and the ACK is '1', the masters should latch the current data before they turn into the wait states. When the masters come back from the wait states, they should resume all signals remembered before they fell into the wait states. Note that, at the edges when masters return from wait states, the only thing they do is to resume the remembered signals. The first IF-ELSE block is skipped at that clock edge.

For slaves:

At every clock rising edge, check the value of the STB.

IF: the STB='0',

 IF: no burst is started yet, do nothing.

 ELSE: if a burst has already started, set the ACK to '1' and hold all other signal unchanged for one more clock cycle.

ELSE: the STB='1', check the CTI, BTE, WE and other signals.

 IF: the current CTI is not an EOB, set the ACK to '1', meanwhile,

 IF: it is a write burst, accept and write the data to the address currently transferred through the bus. Exception: if this is the first write operation of the burst, don't process the data.

 ELSE: if it is a read burst, do: 1) return the data to the master based on the previously calculated address. Exception: if this is the first read operation of the burst, i.e. there's no pre-calculated address, return the data based on the current address sent by the master. 2) calculate the next address to read according to the value of the current address, the CTI, and the BTE.

 ELSE: if the CTI=EOB, set the ACK to '0', meanwhile,

 IF: it is a write burst, latch the last data and de-assert all signals.

 ELSE: if it is a read burst, just de-assert all signals.

IF: no wait state is needed, then that's it. Go to the next clock rising edge.

ELSE: if the slaves currently are unable to handle more data, a wait state is inserted by resetting the ACK to '0'. All other signals can output as 'X'. However, the slaves should still go through the previous IF-ELSE block, and somehow remember what the next output signals should be. In case of write bursts and the STB is '1', the slaves should latch the current data before they turn into the wait state. When the slaves come back from the wait states, they should resume all signals remembered before they fell into the wait states. Note that, at the edges when slaves return from wait states, the only thing they do is to resume the remembered signals. The first IF-ELSE block is skipped at that clock edge.

The last thing about the burst transaction needed to explain is about the wait state. According to the WISHBONE specification, both the masters and the slaves are allowed to insert wait states at any time during the burst transactions when they cannot accept more data temporarily. The following is an examples about the wait state of the burst transactions. Since the rules summarized above also suit for the wait state cases, the readers are

suggested to examine them in the example.

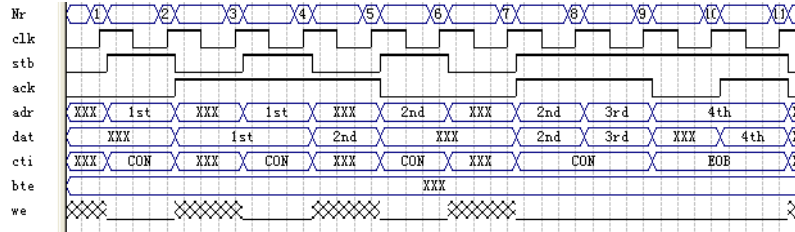


Figure 5.13: An burst transaction with wait states

Figure 5.13 is a constant read burst, which deliberately inserts some wait states both by the master and the slave. When the master or the slave turns into wait states, they output 'X', i.e. unknown signal.

Edge 1: The master initiates the burst. The slave does nothing.

Edge 2:

Master: The master wants to insert a wait state. If there is no wait state, according to the ACK which is '0' at the edge 2, the master should hold all signals unchanged. So now the master must remember the value of all output signals, and repeats them when the master resumes from the wait state. By resetting the STB to '0', the master turns into the wait state.

Slave: The slave, however, sees the STB='1' at the edge 2. Since it can handle this reading request, the slave sets the ACK to '1' and returns the 1st data. Because this is the first read operation in the burst, the slave outputs the data based on the 1st address sent from the master. Besides, the slave needs to predict the next address based on the 1st address and the CTI.

Edge 3:

Master: The master is back from the wait state. Now it should recall all signals logged at the edge 2. Because at the edge 2 the master should keep signals unchanged, the signals at the edge 3 thus are the same as those at the edge 1.

Slave: The slave checks the ACK and finds out which is '0', so it holds all outputs (the 1st data) unchanged for 1 more cycle.

Edge 4:

Master: The master feels not like working again. If there is no wait state, the master should latch the current data and output the next address because the current ACK is '1'. But due to the master

inserts another wait state, it only latches the 1st data, outputs 'X' for other signals. By now the first read operation to the 1st address is finished.

Slave: The slave checks the STB and which is '1'. Since it is capable to keep working, it 1) sets the ACK to '1' for 1 more cycle; 2) output 2nd data based on pre-calculated address at the edge 2; 3) calculate the next address.

Edge 5:

Master: The master comes back from the wait state, and resumes the signals should have sent at the edge 4.

Slave: The slave wants to insert a wait state. As the STB is '0' at the edge 5, the slave should keep all signals unchanged. But since this is a wait state, it remembers the data and output 'X' instead.

Edge 6:

Master: The master plans to insert another wait state. If no wait state, the master should repeat the signals resumed at the edge 5, because right now the ACK='0'. So once more it remembers these signals, which will be resumed later.

Slave: The slave has been in the wait state.

Edge 7: Both the master and the slave resume from the wait state. The master recalls the signals remembered at the edge 6. The slave repeats the signals remembered at the edge 5.

Edge 8:

Master: The master wants to work. So it 1) sets the STB to '1'; 2) latches the current data (2nd); 3) outputs the next address (3rd) and other signals.

Slave: The slaves wants to work too. It 1) asserts the ACK to '1'; 2) returns the 3rd data according to the pre-calculated address at the edge 4; 3) calculate the next address based on the current address, the CTI and the BTE.

Explanations to the **Edge 9 and 10** are skipped.

Edge 11:

Master: Since the ACK is '1' and the CTI is EOB, the master knows it is time to finish the burst. Because this is a read burst, the master has to latch the last 4th data, and then de-asserts all signals.

Slave: The slave also realizes it is the end of the burst by the STB and the CTI. Because it is a read burst, the slave has nothing to do

except for de-asserting signals.

Above all, almost all important things of the WISHBONE specification are covered, from the interface signals to the bus transactions. I hope my work could help people to understand the specification easier, so that the WISHBONE standard will be even more widely accepted and applied into real projects. Best wishes to the WISHBONE in the coming competitions of the interconnection standards.

5.3 Conmax IP Core

5.3.1 Introduction

The WISHBONE interCONnect MAtriX IP Core (CONMAX) is an IP core designed by Rudolf Usselmann in Verilog HDL. It constructs a WISHBONE interconnection with a crossbar switch structure, which can be used as the “bus” of a system. The conmax core supports up to 8 masters and 16 slaves, as well as 4 priority levels. This is enough to compose a quite complicated network. Using the core will save a lot time for designers on thinking about how to compose all the modules to become a system, because the conmax helps to handle the traffic within the system. All users need to do is just to connect all other IP cores to the conmax.

If you take a look at the website of the opencores.org, there are several other IP cores that also implement similar functions. But we finally chose the conmax among the competitors because of the following reasons: 1) it supports more masters and slaves; 2) it provides more priority levels; and 3) it is designed by Rudolf Usselmann from the asics.ws [5]. According our experience, the IP cores from that team always have better quality and more detailed documents.

The conmax IP core is not complicated. Its source codes are not many and very structured, so it is easy to read and understand the design of the core. Plus there is a clear and concise document [3]. Those people who are good at Verilog HDL are suggested to skip the following text and turn to read the source codes instead.

5.3.2 Conmax Architecture

There is a figure in the conmax document well exhibits the structure of the core. So I just copy it here as Figure 5.14. As we can see there are

8 master interfaces in the IP core to provide the possibility of supporting 8 WISHBONE masters, and also 16 slave interfaces for 16 slaves. Besides, there is a Register File included in the slave interface 15 which is used to save the information of the priorities.

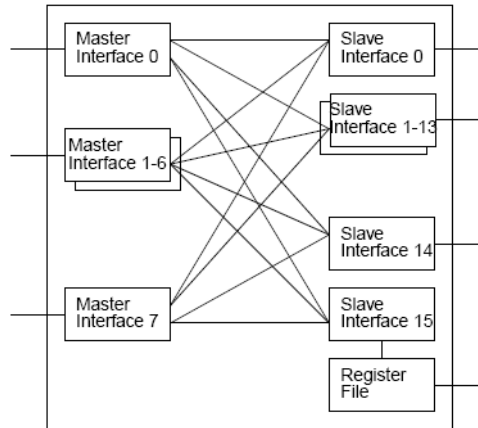


Figure 5.14: Core architecture overview

5.3.3 Register File

The Register File is actually a group of 16 registers. Each register is 32-bit width said in the specification. Actually the source code only implements 16-bit width for the registers. So writing to the higher 16 bits results nothing and reading from those bits always returns zero. There is 1 arbiter in each slave interfaces, which reads the data stored inside the registers to identify the priority of each master.

For instance if the register CFG12 contains the value of “0x000040F0” (only last 16 bits are valid), it means that for the slave interface 12, the priorities from the master 7 to the master 0 are 2, 0, 0, 0, 3, 3, 0, 0, i.e. the master 2 and 3 both have the highest priority “3” to access the slave 12. The next higher master is the master 7, and then all other masters in the third level. Please note that this configuration only applies to the slave interface 12. It is totally possible to configure other 15 registers individually with different priority data, so that each slave has a different configuration of the priorities of the masters.

5.3.4 Parameters and Address Allocation

There are several parameters used to configure the conmax core. They are the **dw**, **aw**, **rf_addr**, and **pri_selN**. All of them can be changed ONLY in the Verilog HDL source file. This means that after the conmax is compiled and downloaded into a FPGA, these parameters cannot be further modified. So the designers have to think about the value of the parameters when designing the system.

The **dw** and the **aw** stand for the width of the data bus and the address bus. They are allowed to be set to different number but usually are both set to 32-bit. The **rf_addr** is a 4-bit width argument. It defines the base address of the Register File. The **pri_selN** is a group of 16 arguments with 2-bit width from **pri_sel0** to **pri_sel15**. Each of them corresponds to one slave interface. The **pri_selN** specifies how many priority levels are supported. This means the 16 slaves can be set to support different priority levels if necessary.

The conmax uses the highest 4-bit of the address to decide which one of the 16 slaves is accessed. For example if the address width is 32-bit, a bus transaction accessing the addresses from 0xB0000000 to 0xBFFFFFFF will be sent to the slave 11, regardless which master the transaction is from.

This also implies that once a slave is connected to the conmax, its address range is determined, e.g. the slave attached on the slave port 8 will have the address range from 0x80000000 to 0x8FFFFFFF.

The only exception is for the slave interface 15, because the Register File is included and its base address is set by the **rf_addr**. If there is a match between the 2nd highest 4-bit of the address and the **rf_addr**, the Register File is selected.

For example if the 4-bit **rf_addr** is configured as “0101”, when writing a number 0x12345678 to the address 0xF500000C, the value of 0x5678 will be written into the register for the slave interface 3 (the last “C” is the address of the register 3), because the highest 0xF selects the slave interface 15 and the 2nd highest 4-bit 0x5 matches the **rf_addr** “0101”.

As we can see from the example, because there is some address space reserved for the Register File, the address space can be used for the external slave is reduced. In last example, the IP core connected to the slave port 15 will have the valid address ranges from 0xF0000000 to 0xF4FFFFFF and from 0xF6000000 to 0xFFFFFFFF. All addresses starting with 0xF5 are reserved for the Register File.

5.3.5 Functional Notices

To describe some notices of the functions of the conmax, an example is designed with the waveform showed in Figure 5.15.

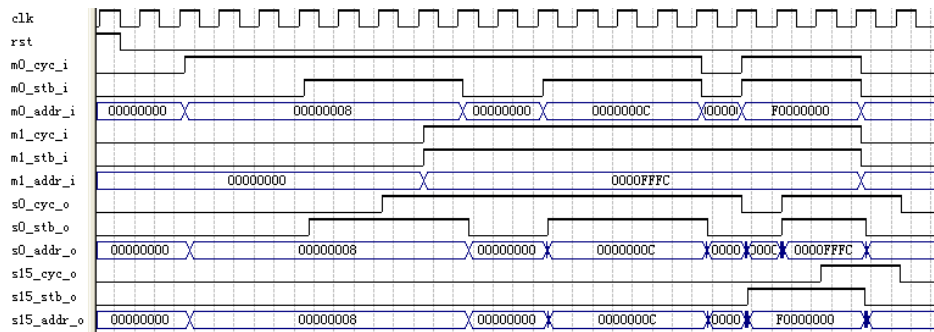


Figure 5.15: An example of using conmax

The figure displays 2 masters and 2 slaves that working with the conmax. The m0_XXX.i and m1_XXX.i are the signals that come from 2 masters and are connected to the master interface 0 and 1 of the conmax. Similarly the s0_XXX.o and s15_XXX.o are the signals that output from conmax to 2 slaves. All other signals are ignored in this example.

The scenario demonstrated in the example is fairly simple. Firstly master 0 accesses slave 0 and gets the grant from the conmax. Later master 1 wants to access the slave 0 too, but is blocked because the grant is still taken by the master 0. After a while the master 0 turns to slave 15, so the grant of the slave 0 is released and the master 1 is allowed to access at that time.

There are several things needed to note in the example:

1. The conmax uses the CYC for its arbiters to determine which masters should be given grants at the moment. But to activate the arbiter both the CYC and the STB have to be '1' at the beginning.

As we can see the CYC of the master 0 becomes '1' quite early, several cycles before the STB. But the s0_cyc_o is output after a long time. This is because the arbiter is not activated until both the CYC and the STB are '1'. However, between the 2 accesses to the address 0x8 and 0xC there is a short gap. But the master 0 keeps holding the grant although the master 1 is ready at that time. This is because the arbiter is already activated, which judges only the CYC to make decisions.

2. The conmax uses finite state machine (FSM) by judging CYC inputs to make complicated arbitration, i.e. round-robin. This results in that

the CYC is usually delayed for several cycles than other signals, which is clearly showed in the waveform. As we know, according to the specification, a WISHBONE interface should respond only when the logic AND of the CYC and the STB is '1'. This means that the delay of the CYC is actually sacrificing the performance of the whole system to achieve a complicated scheme of arbitration.

3. The conmax may broadcast a bus transaction to idle slave ports except for the CYC and the STB signals. As in the example, s15_addr_o is the same as s0_addr_o although no one is accessing 0x8 and 0xC in the s15.

The designers have to be very careful to deal with the broadcasting. In some cases, even the STB may be delivered incorrectly for about 1 to 2 cycles, because the arbiter is judging the CYC and hasn't made the decision yet. The only safe way is always to strictly check the CYC and the STB at every clock rising edge, and don't respond to the bus transaction unless the logic AND of the 2 signals is '1'.

5.3.6 Arbitration

The only thing I am not so satisfied about the conmax specification is the document doesn't tell the details about how the arbitrations are done. For example it says if all priorities are equal, the arbiters work in a round-robin way. But how the round-robin is performed? So after studying the source codes I summarize the arbitration of the conmax as the following rules:

1. The conmax is reset by the RST signal of the WISHBONE. Since all the RST signals in a WISHBONE network are supposed to be connected together, if any RST input becomes '1', the conmax will be reset.
2. After resetting, all the registers of the Register File of the conmax are cleared. In this default case, all priorities of the masters are reset to '0', i.e. all the masters have equal priorities.
3. The conmax works in a round-robin way for the masters with equal priorities. In the arbiter of every slave interface, there is a FSM with 8 states. The states are used to decide which master is allowed to access this slave at the moment. Let's name the 8 states m0, m1, m2 . . . m7. If the current state is m(n), the master N will have the highest priority to access the slave. After master N have accessed successfully, the FSM will jump to m(n) state. All other priorities are arranged in a circle.

If the current state is m6, the priorities are $m6 > m7 > m0 > m1 > m2 > m3 > m4 > m5$.

For example, when power up the FSM resets to state m0. Now the priorities for the 8 masters are $m0 > m1 > m2 > \dots > m7$. At the next moment if 2 masters m0 and m1 struggle for the grant, the m0 will 100% win because the current state is m0, although both masters have equal priorities in the Register File. Note that here the round-robin arbitration has no randomly selection mechanism.

Besides, please also remember, when m(n) is accessing the FSM will turn to m(n) state. Thus the m(n) will have the highest priority. Please note that this implies if the m(n) won't quit and is always involved into the subsequent competitions, no one else can get the grant any more.

4. The masters can be set to different priorities by writing numbers into the Register File, which could be from 0 to 3. The priorities are $3 > 2 > 1 > 0$. The masters with higher priorities always win the arbitration. But the masters with the same priority numbers are still arbitrated in the round-robin way.
5. To support multiple levels of priorities, the value of the pri_selN has to be correctly configured. When the pri_selN is set to "00", i.e. only support 1 level priority, writing numbers 1, 2, 3 into registers takes no effect. All masters are treated with the same level priorities.
6. When the pri_selN is set to "01", the conmax supports 2 priority levels. Now you can write "00" or "01" into registers. The masters with "01" have higher priorities than those have "00". If you somehow write "11" or "10" into the register, the sequence will be "11" = "01" > "10" = "00". Because in case of the pri_selN is "01", the arbiter only judges the last bit in the Register File for the masters.
7. When the pri_selN is "10", the conmax supports 4 levels. In such case all 0 to 3 priority are valid and "11" > "10" > "01" > "00".
8. If you feel interested to know what will happen if configure the pri_selN to "11", I can tell you I did. The result is just like the pri_selN was set to "01".
9. When a master gets a grant, there is no way to interrupt it, unless the master gives it up by de-asserting the CYC. Even if higher priority masters become ready during the time, they still have to wait for the next competition until the current master terminates itself and gives up the grant.

10. Once again I'd like to remind, the 16 slaves can be configured individually in the Register File. This means one master can have different priorities when it accesses different slaves.

So far, all the descriptions about the conmax specification are finished. I hope this is clear enough for you to understand how the interconnection is organized and operated by the conmax, which is one of the most important components in a system.

Reference:

- [1] Webpage, Frequently Asked Questions, *3.2 What is the preferred System-On-A-Chip (SoC) bus for opencores?*, OpenCores Organization, <http://www.opencores.org/faq.cgi/section/3/3.2#3.2>, Last visit: 2009.01.11,
This is the official FAQ which suggests the WISHBONE as the preferred bus standard.
- [2] OpenCores Organization, *Specification for the: WISHBONE System-on-Chip (SoC), Interconnection Architecture for Portable IP Cores*, Revision: B.3, Released: September 7, 2002, Available at: <http://www.opencores.org/projects.cgi/web/wishbone/wbspec.b3.pdf>, Last visit: 2009.01.11.
- [3] Rudolf Usselmann, *WISHBONE Interconnect Matrix IP Core*, Rev. 1.1, October 3, 2002, Available at: http://www.opencores.org/cvsweb.shtml/wb_conmax/doc/conmax.pdf, Last visit: 2009.01.11.
- [4] Webpage, A message posted at Opencore's forum, *Wishbone spec clarification*, <http://www.opencores.org/forums/cores/2007/04/002651>, Last visit: 2009.01.11,
This message contains useful information to explain the LOCK signal of the WISHBONE standard.
- [5] Website, ASICS.ws, <http://asics.ws/>, Last visit: 2009.01.11,
This is the website of a specialized FPGA/ASIC design team. They produced many free IP cores with very good quality and document.
- [6] Rudolf Usselmann, *OpenCores SoC Bus Review*, Rev. 1.0, January 9, 2001, Available at: http://www.opencores.org/projects.cgi/web/wishbone/soc_bus_comparison.pdf, Last visit: 2009.01.11,
This article compares several bus standard. It is a little out of date but worth to take a look.