SimpCon – a Simple SoC Interconnect Draft

Martin Schoeberl
martin@jopdesign.com

August 12, 2006

This document proposes a simple interconnection standard for system-on-chip (SoC) components. It is intended to provide pipelined access to devices such on-chip peripherals and on-chip memory controller with minimum hardware resources.

1 Introduction

The intention of the following SoC interconnect standard is to be simple and efficient with respect to implementation resources and transaction latency.

SimpCon is a fully synchronous standard for on-chip interconnections. It is a point-to-point connection between a master and a slave. The master starts either a read or write transaction. Master commands are single cycle to free the master to continue on internal operations during an outstanding transaction. The slave has to register the address when needed for more than one cycle. The slave also registers the data on a read and provides it to the master for more than a single cycle. This property allows the master to delay the actual read if it is busy with internal operations.

The slave signals the end of the transaction through a novel *ready counter* to provide an early notification. This early notification simplifies the integration of peripherals into pipelined masters.

Slaves can also provide several levels of pipelining. This feature is announced by two static output ports (one for read and one write pipeline levels).

Off-chip connections (e.g. main memory) are device specific and need a slave to perform the translation. Peripheral interrupts are not covered by this specification.

1.1 Feature

- Master/slave point-to-point connection

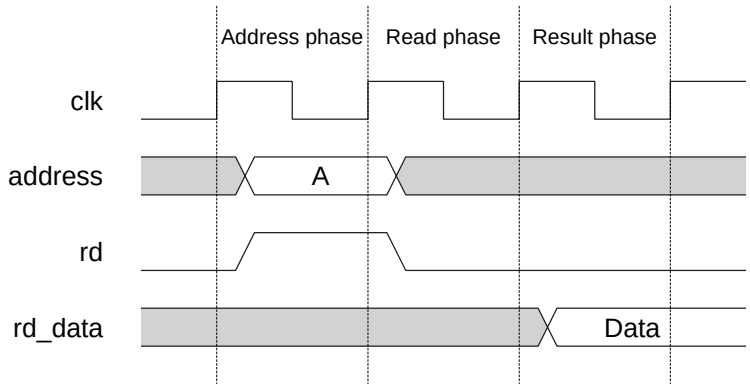

Figure 1: Basic read transaction

- Synchronous operation
- Read and write transactions
- Early pipeline release for the master
- Pipelined transactions
- Open-source specification
- Low implementation overheads

1.2 Basic Read Transaction

Figure 1 shows a basic read transaction for a slave with one cycle latency. The acknowledge signals are omitted from the figure. In the first cycle, the address phase, the `rd` signals the slave to start the read transaction. The address is registered by the slave. During the following cycle, the read phase, the slave performs the read and registers the data. Due to the register in the slave the data is available in the third cycle, the result phase. To simplify the master, the read data stays valid till the next read request response.

1.3 Basic Write Transaction

A write transaction consists of a single cycle address/command phase started by assertion of `wr` where the address and the write data are valid. `address` and `wr_data` are usually registered by the slave. The end of the write cycle is signalled to the master by the slave with `rdy_cnt`. See section 3 and an example in Figure 3.

Signal	Width	Direction	Required	Description
address	1–32	Master	No	Address lines from the master to the slave port
wr_data	32	Master	No	Data lines from the master to the slave port
rd	1	Master	No	Start of a read transaction
wr	1	Master	No	Start of a write transaction
rd_data	32	Slave	No	Data lines from the slave to the master port
rdy_cnt	2	Slave	Yes	Transaction end signalling
rd_pipeline_level	2	Slave	No	Maximum pipeline level for read transactions
wr_pipeline_level	2	Slave	No	Maximum pipeline level for write transactions

Table 1: SimpCon port signals

2 SimpCon Signals

This sections defines the signals used by the SimpCon connection. Some of the signals are optional and may not be present on a peripheral device.

All signals are a single direction point-to-point connection between a master and a slave. The signal details are described by the device that drives the signal. Table 1 lists the signals that define the SimpCon interface. The column Direction indicates wether the signal is driven by the master or the slave.

2.1 Master Signal Details

This section describes the signals that are driven by the master to initiate a transaction.

2.1.1 address

Master addresses represent word addresses as offsets in the slaves address range. address is valid a single cycle either with rd for a read transaction or with wr and wr_data for a write transaction.

The number of bits for address depend on the slaves address range. For a single port slave address can be omitted.

2.1.2 wr_data

The wr_data signals carry the data for a write transaction. It is valid for a single cycle together with address and wr. The signal is typically 32 bits wide. Slaves can ignore upper bits when the slave port is less than 32 bits.

2.1.3 rd

The `rd` signal is asserted a single clock cycle to start a read transaction. `address` has to be valid in the same cycle.

2.1.4 wr

The `wr` signal is asserted a single clock cycle to start a write transaction. `address` and `wr_data` have to be valid in the same cycle.

2.1.5 sel_byte

The `sel_byte` signal is reserved for future versions of the SimpCon specification to add individual byte enables.

2.2 Slave Signal Details

This section describes the signals that are driven by the slave as a response to transaction initiated by the master.

2.2.1 rd_data

The `wr_data` signals carry the result for a read transaction. The data is valid when `rdy_cnt` reaches 0 and stays valid till a new read result is available. The signal is typically 32 bits wide. Slaves that provide less than 32 bits should pad the upper bits with 0.

2.2.2 rdy_cnt

The `rdy_cnt` signal provides the number of cycles till the pending transaction will finish. A 0 means that either read data is available or a write transaction has been finished. Values of 1 and 2 mean the the transaction will finish in at least 1 or 2 cycles. The maximum value is 3 and means the the transaction will finish in 3 or *more* cycles. Note that not all values have to be used in a transaction. Each monotonic sequence of `rdy_cnt` values is legal.

2.2.3 rd_pipeline_level

The static `rd_pipeline_level` provides the master with the read pipeline level of the slave. The signal has to be constant to enable the synthesizer to optimize the pipeline level dependent state machine in the master.

2.2.4 wr_pipeline_level

The static `wr_pipeline_level` provides the master with the write pipeline level of the slave. The signal has to be constant to enable the synthesizer to optimize the pipeline level dependent state machine in the master.

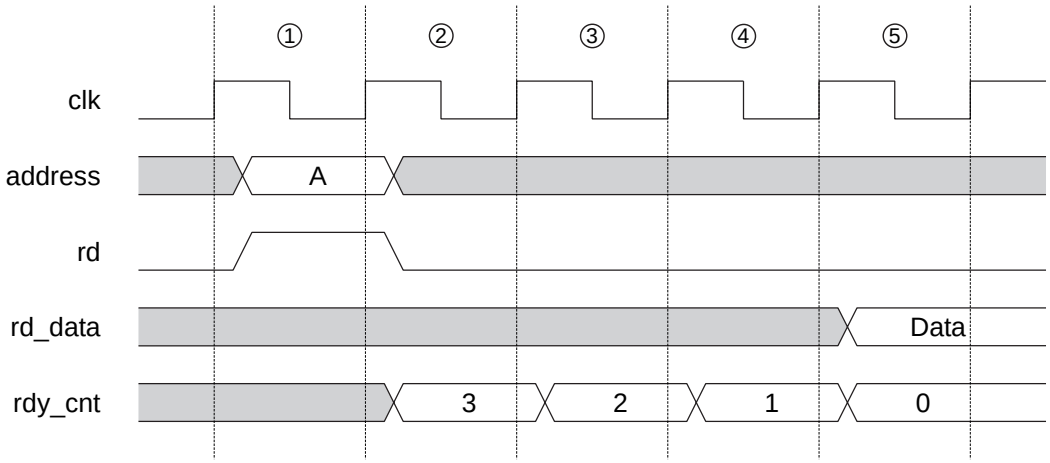

Figure 2: Read transaction with wait states

3 Slave Acknowledge

Flow control between the slave and the master is usually done by a single signal in the form of *wait* or *acknowledge*. The `ack` signal, e.g. in the Wishbone specification, is set when the data is available or the write operation has finished. However, for a pipelined master it can be of interest to know it *earlier* when a transaction will finish.

For a lot of slaves, e.g. a SRAM interface with fixed wait states, this information is available inside the slave. In the SimpCon interface this information is communicated to the master through the two bit signal `rdy_cnt`. `rdy_cnt` signals the number of cycles till the read data will be available or the write transaction will be finished. Value 0 is equivalent to an *ack* signal and 1, 2, and 3 are equivalent to a wait request with the distinction that the master knows how long the wait request will last.

To avoid too many signals at the interconnect `rdy_cnt` has a width of two bits. Therefore, the maximum value of 3 has the special meaning that the transaction will finish in 3 or *more* cycles. As a result the master can only use the values 0, 1, and 2 to release actions in it's pipeline.

Idle slaves will keep the former value of 0 for `rdy_cnt`. Slaves, that don't know in advance how many wait states are need for the transaction can produce sequences that omit any of the numbers 3, 2, and 1. The master has to handle this situations.

Figure 2 shows an example of a slave that needs three cycles for the read to be processed. In cycle 1 the read command and the address are set by the master. The slave registers the address and sets `rdy_cnt` to 3 in cycle 2. The read takes three cycles (2–4) during which `rdy_cnt` is decremented. In cycle 4 the data is available inside the slave and gets registered. It is available in cycle 5 for the master and `rdy_cnt` is finally 0. Both, the `rd_data` and `rdy_cnt` will keep their value till a new transaction is requested.

Figure 3 shows an example of a slave that needs three cycles for the write to be processed. The address, the data to be written and the write command are valid during cycle 1. The slave registers the address and write data during cycle 1 and performs the write operation

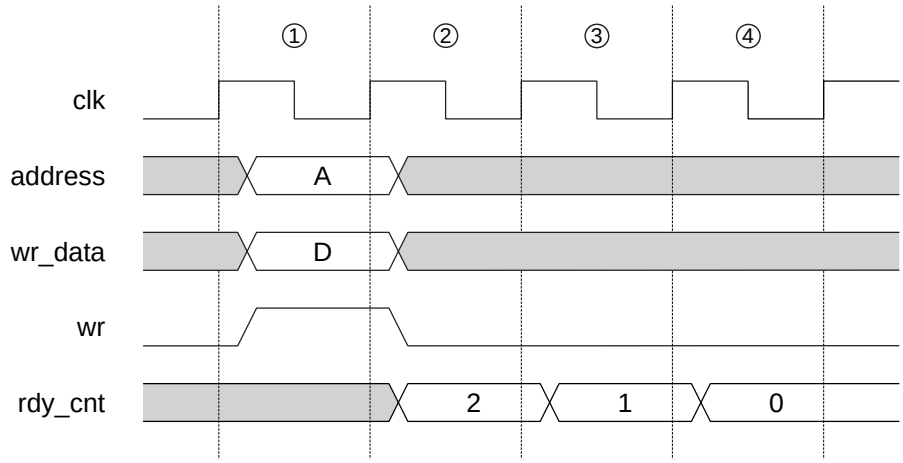

Figure 3: Write transaction with wait states

during cycles 2–4. The `rdy_cnt` is decremented and a non-pipelined slave can accept a new command after cycle 4.

4 Pipelining

Figure 4 shows a read transaction for a slave with four cycles latency. Without any pipelining the next read transaction will start in cycle 7 after the data from the former read transaction is read by the master. The three bottom lines show when new read transactions will be started for different pipeline levels. With pipeline level 1 a new transaction can start in the same cycle when the former read data is available (in this example in cycle 6). Higher levels mean that the next read will start earlier as shown for level 2 and 3.

Implementation of level 1 in the slave is trivial (just two more transitions in the state machine). It is recommended to provide level 1 at least for read transactions. Level 2 is a little bit more complex but usually no additional address or data registers are needed.

To implement level 3 pipelining in the slave at least an additional address register is needed. However, to use level 3 the master has to issue the request in the same cycle as `rdy_cnt` goes to 2. That means this transition is combinatorial. We see in Figure 4 that `rdy_cnt` value of 3 means three or more cycles till the data is available and can therefore not be used to trigger a new transaction.

5 Multiple Master

SimpCon defines no signals for the communication between a master and an arbiter. However, it is possible to build a multi master system with SimpCon. The SimpCon interface can be used as interconnect between the masters and the arbiter and the arbiter and the slaves. In this case the arbiter acts as slave for the master and as master for the peripheral devices.

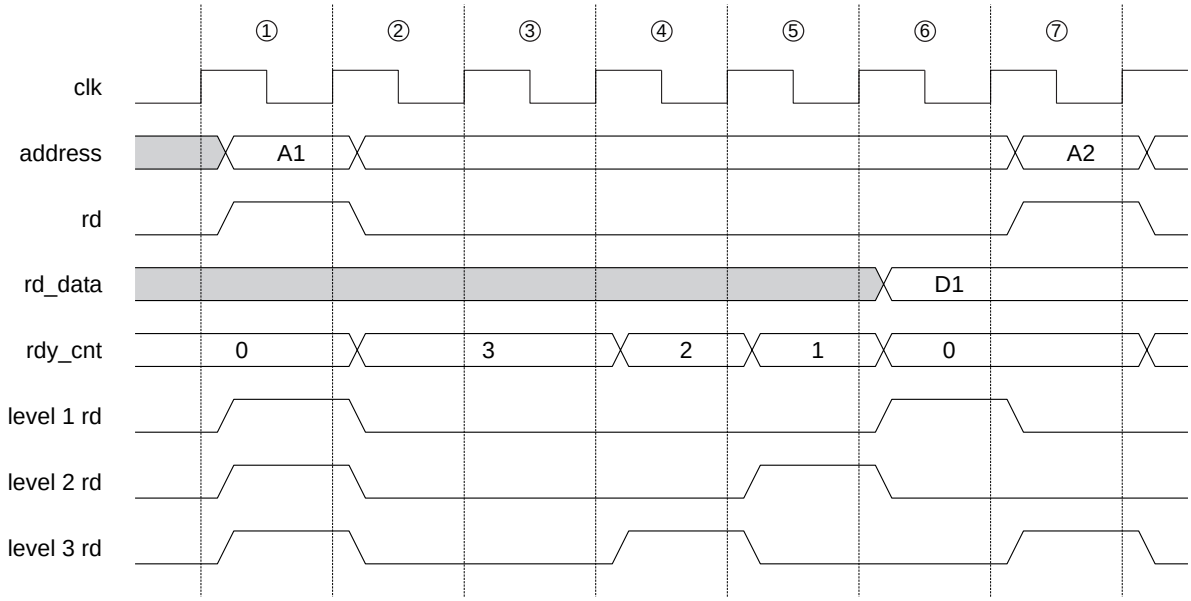

Figure 4: Different pipeline levels for a read transaction

The missing arbitration protocol in SimpCon results in the need to queue $n-1$ requests in an arbiter for n masters. However, for this additional HW we get zero overheads for the bus request. The master, which gets the bus will will start the slave transaction in the same cycle.

TODO: add a timing diagram to explain this concept.

6 Examples

This section provides some examples for the application of the SimpCon definition.

6.1 IO Port

TODO: Show how simple an IO port can be with SimpCon. We need no addresses and can tie `bsy_cnt` to 0. We only need the `rd` or `wr` signal to enable the port.

6.2 SRAM interface

The following example is taken from an implementation of SimpCon for a Java processor. The processor is clocked with 100MHz and the main memory consists of 15ns static RAMs. Therefore the minimum access time for the RAM is two cycles. The slack time of 5ns forces us to use output registers for the RAM address and write data and input registers for the read data in the IO cells of the FPGA. These registers fit nice with the intention of SimpCon to use registers inside the slave.

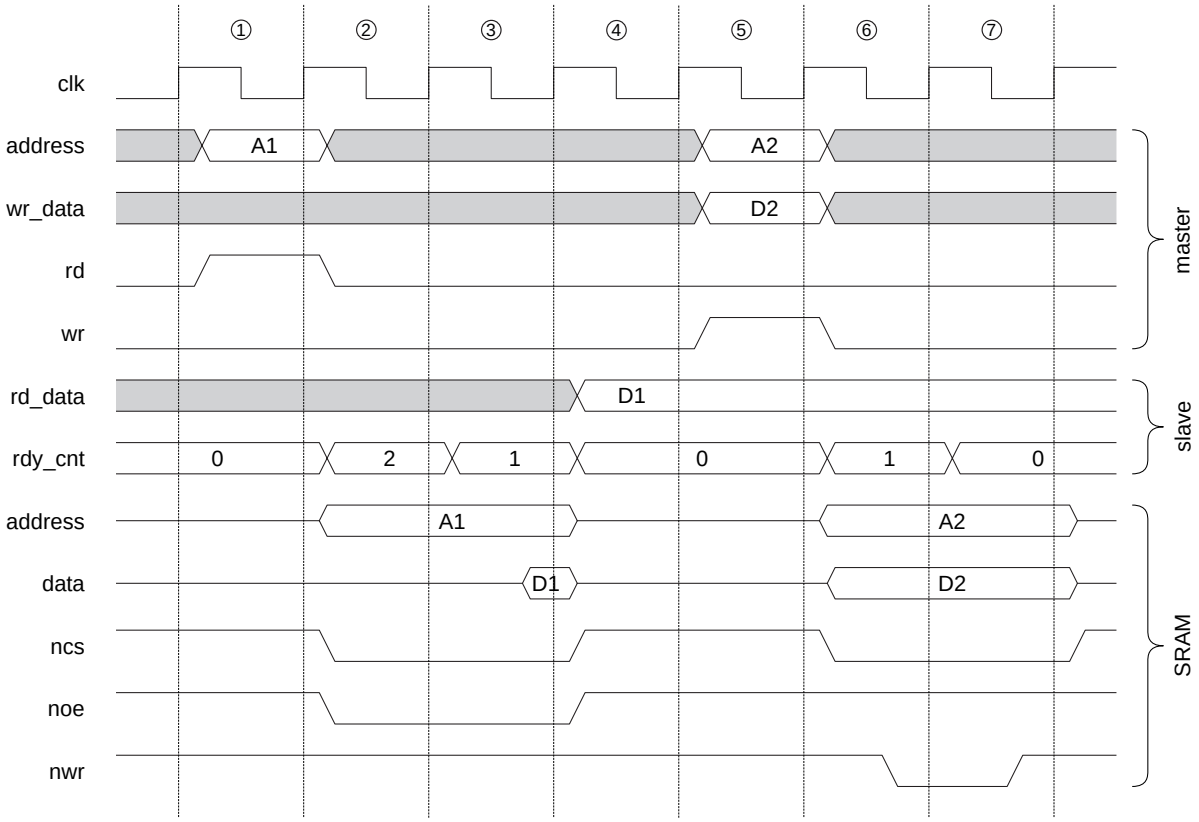

Figure 5: Static RAM interface without pipelining

Figure 5 shows the interface for a non-pipelined read access followed by a write access. Four signals are driven by the master and two signal by the slave. The lower half of the figure shows the signals at the FPGA pins where the RAM is connected.

In cycle 1 the read transaction is started by the master and the slave registers the address. The slave also sets the registered control signals `ncs` and `noe` during cycle1. Due to the IO cell registers, the address and control signals are valid at the FPGA pins very early in cycle 2. At the end of cycle 3 (15ns after `address`, `ncs` and `noe` are stable) the data from the RAM is available and can be sampled with the rising edge for cycle 4.

The master reads the data in cycle 4 and starts a write transaction in cycle 5. Address and data are again registered from the slave and are available for the RAM at the beginning of cycle 6. To perform a write in two cycles the nwr signal is registered by a negative triggered flip-flop.

In figure 6 we see a pipelined read from the RAM with pipeline level 2. With this pipeline level and the two cycles read access time of the RAM we get the maximum bandwidth possible.

We can see the start of the second read transaction in cycle 3 during the read of the first data from the RAM. The new address is registered in the same cycle and available for the RAM in the following cycle 4. Although we have a pipeline level of 2 we need no additional address or data register. The read data is available for two cycles (`rdy_cnt` 2 or 1 for the next read)

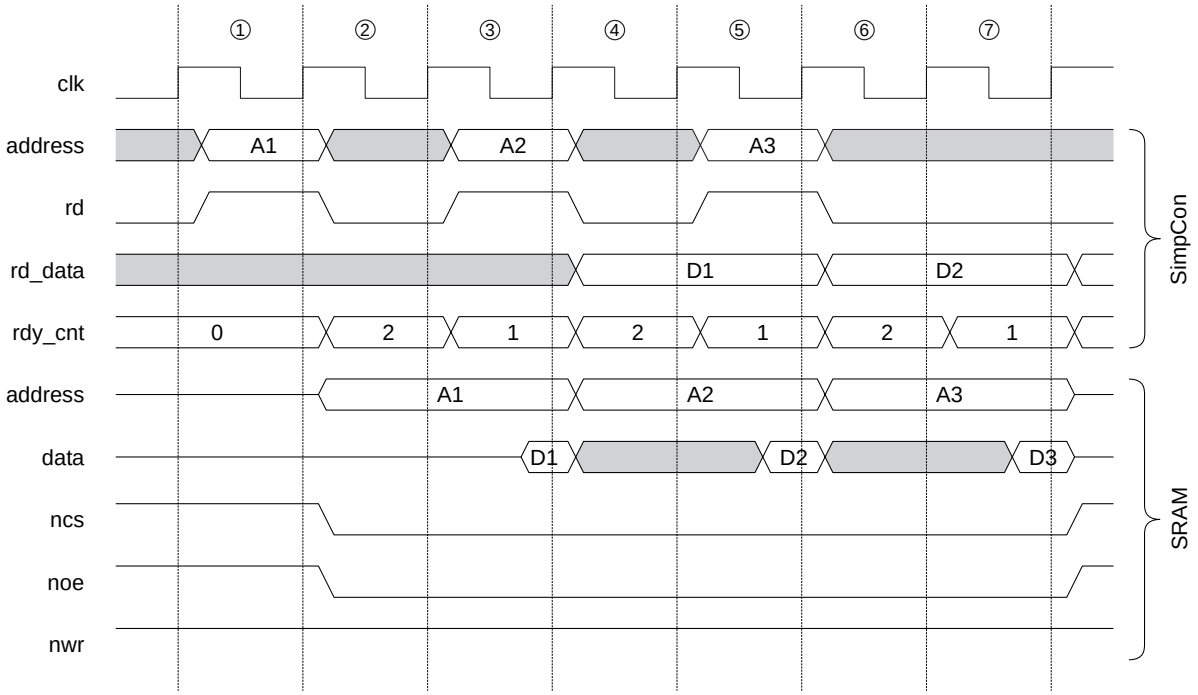

Figure 6: Pipelined read from a static RAM

and the master is free to select one of the two cycles to read the data.

6.3 Master Multiplexing

To add several slaves to a single master the `rd_data` and `bsy_cnt` have to be multiplexed. Due to the fact that all `rd_data` signals are registered by the slaves a single pipeline stage will be enough for a large multiplexer. The selection of the multiplexer is also known at the transaction start but needed at most in the next cycle. Therefore it can be registered to further speed up the multiplexer.

TODO: add a schematic for the master `rd_data` multiplexer.

7 Status

- First timing diagrams drawn
- SimpCon SRAM interface for JOP on Cyclone and Spartan-3 is available
- Project at opencores.org accepted
- Simple UART as SimpCon example
- IO in JOP changed to SimpCon (uart, cnt, usb)

Next steps:

- Continue this document
- Provide Wishbone bridges

to clarify:

- Use transaction or transfer in this document?
- Use address phase or better command cycle?

8 Notes

8.1 Group comment

```
After implementing the Wishbone interface for main memory access
from JOP I see several issues with the Wishbone specification that
makes it not the best choice for SoC interconnect.

The Wishbone interface specification is still in the tradition of
microcomputer or backplane busses. However, for a SoC interconnect,
which is usually point-to-point, this is not the best approach.

The master is requested to hold the address and data valid through
the whole read or write cycle. This complicates the connection to a
master that has the data valid only for one cycle. In this case the
address and data have to be registered \emph{before} the Wishbone
connect or an expensive (time and resources) MUX has to be used. A
register results in one additional cycle latency. A better approach
would be to register the address and data in the slave. Than there
is also time to perform address decoding in the slave (before the
address register).

There is a similar issue for the output data from the slave: As it
is only valid for a single cycle it has to be registered by the
master when the processor is not reading it immediately. Therefore,
the slave should keep the last valid data at it's output even when
\emph{wb.stb} is not assigned anymore (which is no issue from the
hardware complexity).

The Wishbone connection for JOP resulted in an unregistered Wishbone
memory interface and registers for the address and data in the
Wishbone master. However, for fast address and control output
```

(t_{co}) and short setup time (t_{su}) we want to place the registers in the IO-pads of the FPGA. With the registers are buried in the WB master it takes some effort to set the right constraints for the Synthesizer to implement such IO-registers.

The same issue is true for the control signals. The translation from the \emph{wb.cyc}, \emph{wb.stb} and \emph{wb.we} signals to \emph{ncs}, \emph{noe} and \emph{nwe} for the SRAM are on the critical path.

The \emph{ack} signal is too late for a pipelined master. We would need to know it *earlier* when the next data will be available --- and this is possible, as we know in the slave when the data from the SRAM will arrive. A work around solution is a non-WB-conforming early ack signal.

Due to the fact that the data registers not inside the WB interface we need an extra WB interface for the Flash/NAND interface (on the Cyclone board). We cannot afford the address decoding and a MUX in the data read path without registers. This would result in an extra cycle for the memory read due to the combinational delay.

In the WB specification (AFAIK) there is no way to perform pipelined read or write. However, for blocked memory transfers (e.g. cache load) this is the usual way to get a good performance.

Conclusion -- I would prefer:

* Address and data (in/out) register in the slave
* A way to know earlier when data will be available (or a write has finished)
* Pipelining in the slave

As a result from this experience I'm working on a new SoC interconnect (working name SimpCon) definition that should avoid the mentioned issues and should be still easy to implement the master and slave.

As there are so many projects available that implement the WB interface I will provide bridges between SimpCon and WB. For IO devices the former arguments do not apply to that extent as the pressure for low latency access and pipelining is not high. Therefore, a bridge to WB IO devices can be a practical solution for design reuse.

8.1.1 additional comments

The idea for (some) pipeline support is twofold:

1.) The slave will provide more information than a single \emph{ack}
or wait states. It will (if it is capable to do) signal the number
of clock cycles remaining till the read data is available (or the
write has finished) to the master. This feature allows the pipelined
master to prepare for the upcoming read.

2.) If the slave can provide pipelining the master can use
overlapped wr or rd requests. The slave has a static output port
that tells how many pipeline stages are available. I call this
'pipeline level':

    0 means non overlapping
    1 a new rd/wr request can be issued in the same cycle
      when the former data is read.
    2 one earlier and
    3 is the maximum level where you get full pipelining
      on the basic read cycle with one wait state
      (command - read - read - result).

The draft of the spec at the moment are few sketches on real paper -
takes some time to draw all diagrams for a document.

I have a first implementation of SimpCon on JOP to test the ideas: A
master in JOP and a slave for SRAM access.

8.2 e-mail from Robert Finch

Hi Martin, I read your comments. I've thought some about the
WISHBONE spec myself.

"Martin Schoeberl" <mschoebe@mail.tuwien.ac.at> wrote in message
news:<4384f0b3$0$11610$3b214f66@tunews.univie.ac.at>...
> After implementing the Wishbone interface for main memory access
> from JOP I see several issues with the Wishbone specification that
> makes it not the best choice for SoC interconnect.

> The master is requested to hold the address and data valid through
> the whole read or write cycle. This complicates the connection to a
> master that has the data valid only for one cycle. In this case the
> address and data have to be registered *before* the Wishbone connect
> or an expensive (time and resources) MUX has to be used. A register
> results in one additional cycle latency. A better approach would be
> to register the address and data in the slave. Than there is also
> time to perform address decoding in the slave (before the address
> register).

I've of the opinion that all outputs of masters should be registered. Registering the outputs hides the timing of the master's internal signals from the rest of the system and helps turn it into a 'black box'. However, in my designs I provide both registered and unregistered versions of outputs, as it is quite handy to have unregistered signals sometimes. It would have been nice if the WISHBONE bus spec'd unregistered signals as well as registered ones. I've just been naming the unregistered signals by including '_nxt' in the signal name as in 'adr_nxt_o'. '_nxt' standing for the signal value that will be 'next'.

Why is the MUX needed ?

I've found that a register may indeed result in an additional cycle of latency, depending on the how the system is put together. However, I've also found that it doesn't really make any difference to the performance of the system. Registering the output often allows the cycle time to be decreased, and the 'lost' cycle of latency is made up for by better timing. I've also found that the INTERCON (address decoding, bus muxing logic, and arbitration) typically requires a full cycle by itself and it's best to have the signals feeding into the INTERCON already registered. Unless the system is really small (single master / slave).

By 'address decoding in slaves' I'm assuming you mean partial address decoding for only register selection. Full address decoding shouldn't be done in slaves as it wastes a lot of resources. The address decoding (device/slave selection) should be done by the INTERCON, and is a function of the system.

Almost always masters are designed to hold address and data valid until the external system acknowledges the request.

>

> There is a similar issue for the output data from the slave: As it
> is only valid for a single cycle it has to be registered by the
> master when the processor is not reading it immediately. Therefore,
> the slave should keep the last valid data at it's output even when
> wb.stb is not assigned anymore (which is no issue from the hardware
> complexity).

I'm not sure I understand the 'single cycle' timing. Slave devices I've worked on present valid data as long as the signals coming from the INTERCON indicate that it should do so. Otherwise the output data from the slave is allowed to flip around according to whatever register is addressed as it doesn't affect the system since it's not muxed to the master's inputs unless it's the addressed device.

Generally, during a read request the master will always be ready to read data immediately. If it wasn't ready to read the data it shouldn't have requested it, as this wastes bus bandwidth.

>
> The Wishbone connection for JOP resulted in an unregistered Wishbone
> memory interface and registers for the address and data in the
> Wishbone master. However, for fast address and control output (tco)
> and short setup time (tsu) we want the registers in the IO-pads of
> the FPGA. With the registers buried in the WB master it takes some
> effort to set the right constraints for the Synthesizer to implement
> such IO-registers.
>
> The same issue is true for the control signals. The translation from
> the wb.cyc, wb.stb and wb.we signals to ncs, noe and nwe for the
> SRAM are on the critical path.

I've come to the conclusion that it's unrealistic to expect that external memory can be accessed at a high rate using only a single clock cycle. There is naturally a multi-cycle latency when dealing with an external device operating a high clock rate. The registered outputs of a WISHBONE master typically wouldn't need to be registered at the IO-pads.

> The ack signal is too late for a pipelined master. We would need to
> know it *earlier* when the next data will be available --- and this
> is possible, as we know in the slave when the data from the SRAM
> will arrive. A work around solution is a non-WB-conforming early ack
> signal.

I ran into this too. I built a system similar to this and it worked okay. But, I decided not to build newer systems this way. A problem is that the latency of external device may vary. This makes it difficult to pipeline the master. SRAM may have a latency of three cycles, BRAM two cycles, and IO-devices a single cycle. My (current) master already has an internal three stage pipeline, adding three more pipeline stages for memory would turn it into a six stage monster.

>
> Due to the fact that the data registers not inside the WB interface
> we need an extra WB interface for the Flash/NAND interface (on the
> Cyclone board). We cannot afford the address decoding and a MUX in
> the data read path without registers. This would result in an extra
> cycle for the memory read due to the combinational delay.
>
Yes. Can the delay be hidden using mult-masters (later) ?

> In the WB specification (AFAIK) there is no way to perform pipelined
> read or write.

This is something I've thought was missing from the spec as well. However, doing pipelined access across a system bus could be quite a feat.

However, for blocked memory transfers (e.g. cache
> load) this is the usual way to get a good performance.
>
> Conclusion -- I would prefer:
>
>   * Address and data (in/out) register in the slave
>   * A way to know earlier when data will be available (or
>   a write has finished)
>   * Pipelining in the slave
>
> As a result from this experience I'm working on a new SoC
> interconnect (working name SimpCon) definition that should avoid the
> mentioned issues and should be still easy to implement the master
> and slave.
>
> As there are so many projects available that implement the WB
> interface I will provide bridges between SimpCon and WB. For IO
> devices the former arguments do not apply to that extent as the

> pressure for low latency access and pipelining is not high.
> Therefore, a bridge to WB IO devices can be a practical solution for
> design reuse.
>
> A question to the group: What SoC interconnect are you using?
> A standard one for the peripheral devices and a 'home-brewed' for
> more demanding connections (e.g. external RAM access)?
>
> Martin
>

I'm using an 'enhanced' WISHBONE bus (I added one or two signals, and renamed a couple).

I found that for my systems it wasn't necessary to pipeline the memory system to get good performance. The reason being that there are multiple bus masters, and all the memory bandwidth is consumed anyway. (CPU, VIDEO, AUDIO, SPRITE, DISK, CPU2). I ended up building a shared memory controller with an arbitrater that allows each device access only every third cycle. This effectively hides a three cycle latency though the memory. The external memory can service a request every single clock cycle (at 40MHz!). (Just not from the same master) Every cycle one of the masters is selected to be allowed a memory access. Three cycles later, read data is available for that master. From the master's perspective it looks like a normal WISHBONE bus.

Even though the system isn't pipelined, it's using the maximum amount of performance it can get out of the memory. As a result, it's turned out that the WISHBONE bus serves as a suitable bus system to use.

I'm not sure what's included in JOP system (I'm a news-subscriber), but it may be easier to get better performance by using multiple CPU's. For example, one cpu could be handling network communcations while a second is running Java code (JVM). If there is any kind of VIDEO or audio (eg MP3) that could be handled by another master as well.

Good Luck with you're bus design.

Robert

8.3 comp.arch.fpga

```
>> The last days I played around with the Quartus SOPC builder [1].
>> Although I'm more a batch/make guy, I'm impressed by the easy to use
>> tool. In order to scratch a little bit on the dominance of the NIOS II
>> in the SOPC world I wrapped JOP [2] into an Avalon component ;-)
>
> Kudos, that is excellent. Any lessons/gotchas about turning JOP into an
> SOPC components should someone else fancy a similar undertaking?

The Avalon bus is very flexible. Therefore, writing a slave or
master (SOPC component) is not that hard. The magic is in the Avalon
switch fabric generated by the builder. However, an example would
have helped (Altera listening?). I didn't find anything on Altera's
website or with Google. Now a very simple slave can be found at [1].

One thing to take care: When you (like me) like to avoid VHDL files
in the Quartus directory you can easily end up with three copies of
your design files. Can get confusing which one to edit. When you
edit your VHDL file in the component directory (the source for the
SOPC builder) don't forget to rebuild your system. The build process
copies it to your Quartus project directory.

When you want to start over with a clean project the only files
needed for the project are: .qpf, .qsf, .ptf

The master is also ease: just address, read and write data,
read/write and you have to react to waitrequest. See as example the
SimpCon/Avalon bridge at [2]. The Avalon interconnect fabric handles
all bus multiplexing, bus resizing, and control signal translation.

>> However, of course there is some drawback. The performance of the
>> Avalon system is lower than a 'native' connection (or in my case
>> via SimpCon [5]) of the main memory to the CPU. I can provide some
>> numbers if there is interest...
>
> Care to elaborate? I'd expect going over Avalon could add latency, but
> if you can exploit multiple outstanding transactions (aka "posted
> reads") and/or bust transfers, the bandwidth should be the same as
> "native".

Yes, the latency is the issue for JOP. JOP does not trigger several
read or write transactions. However, it can trigger one transaction
and than continue to execute microcode. When the (read) result is
```

needed, the JOP pipeline is stopped till the result is available.
What helps is to know in advance (one or two cycles) when the result
will be available. That's the trick with the SimpCon interface.
There is not a single ack or waitrequest signal, but a counter that
will say how many cycles it will take to provide the result. In this
case I can restart the pipeline earlier.

Another point is, in my opinion, the wrong role who has to hold data
for more than one cycle. This is true for several busses (e.g. also
Wishbone). For these busses the master has to hold address and write
data till the slave is ready. This is a result from the backplane
bus thinking. In an SoC the slave can easily register those signals
when needed longer and the master can continue. On the other hand,
as JOP continues to execute and it is not so clear when the result
is read, the slave should hold the data when available. That is easy
to implement, but Wishbone and Avalon specify just a single cycle
data valid.

>> BTW: The Cyclone II FPGA cannot be clocked really faster than the
>> Cyclone (just a few %). I hoped to get some speed-up for free due
>> to a new generation FPGA :-(
>
> I was surprised too when I saw that. I gather the only way the Cyclone
> II can gain you speed over Cyclone I is when you can use the embedded
> multipliers. Makes me wonder about the upcoming Cyclone III.

Are there any other data available on that. I did not find many
comments in this group on experiences with Cyclone I and II. Looks
like the CII was more optimized for cost than speed. Yes, waiting
for III ;-)

Martin

[1]
http://www.opencores.org/cvsweb.cgi/~checkout~/jop/sopc/components/avalon_test_slave/h

[2]
http://www.opencores.org/cvsweb.cgi/~checkout~/jop/vhdl/scio/sc2avalon.vhd

Hi Antti,

> most of the SOPC magin happens in the perl package "Europe" ASFAIK.
> dont expect a lot of information about the internals of the package.

That's fine for me. When the connection magic happens and I don't have to care it's fine. OK, one exception: Perhaps I would like to know more details on the latency. The switch fabric is 'plain' VHdL or Verilog. However, generated code is very hard to read.

> as very simple example for avalon master-slave type of peripherals there
> is on free avalon IP core for SD-card support the core can be found
> at some russian forum and later it was also added to the user ip
> section of the microtronix forums.

Any link handy for this example?

> the avalon master is really as simple as the slave.

Almost, you have to hold address, data and read/write active as long as waitrequest is pending. I don't like this, see above.

In my case e.g. the address from JOP (= top of stack) is valid only for a single cycle. To avoid one more cycle latency I present in the first cycle the TOS and register it. For additional wait cycles a MUX switches from TOS to the address register. I know this is a slight violation of the Avalon specification. There can be some glitches on the MUX switch. For synchronous on-chip peripherals this is absolute not issue. However, this signals are also used for off-chip asynchronous peripherals (SRAM). However, I assume that this possible switching glitches are not really seen on the output pins (or at the SRAM input).

Martin