SimpCon – a Simple SoC Interconnect Version 1.1

Martin Schoeberl
martin@jopdesign.com

June 1, 2009

This document describes 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.

Versions

V 1.0 November 13, 2007, Initial version

V 1.1 June 1, 2009, Updated version

- Typo correction
- Additional signals for cache control
- A simple I/O example

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 Features

SimpCon provides following main features:

- Master/slave point-to-point connection
- 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. 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,[1] 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, `rd_data` stays valid until the next read request response. It is therefore possible for a master to issue a pre-fetch command early. When the pre-fetched data arrives too early it is still valid when the master actually wants to read it. Keeping the read data stable in the slave is *mandatory*.

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.

[1]It has to be noted that the read phase can be longer for devices with a high latency. For simple on-chip I/O devices the read phase can be omitted completely (0 cycles). In that case `rdy_cnt` will be zero in the cycle following the address phase.

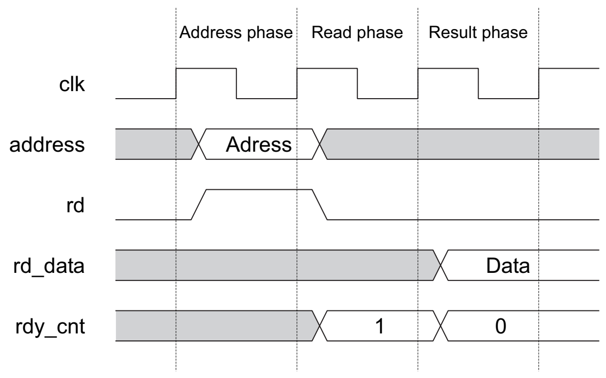

Figure 1: Basic read transaction

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 whether 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 slave's 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` depends on the slave's 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 for a single clock cycle to start a read transaction. `address` has to be valid in the same cycle.

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
`sel_byte`	2	Master	No	Reserved for future use
`uncached`	1	Master	No	Non cached access
`cache_flash`	1	Master	No	Flush/invalidate a cache

Table 1: SimpCon port signals

2.1.4 wr

The `wr` signal is asserted for 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.

2.1.6 uncached

The `uncached` signal is asserted for a single clock cycle during a read or write transaction to signal that a cache, connected in the SimpCon pipeline, shall not cache the read or write access.

2.1.7 cache_flash

The `cache_flash` signal is asserted for a single clock cycle invalidates a cache connected in the SimpCon pipeline.

2.2 Slave Signal Details

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

2.2.1 rd_data

The `rd_data` signals carry the result of a read transaction. The data is valid when `rdy_cnt` reaches 0 and *stays valid* until 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 until 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 transaction will finish in at least 1 or 2 cycles. The maximum value is 3 and means 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.

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 many slaves, e.g. an 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 ready counter (`rdy_cnt`). `rdy_cnt` signals the number of cycles until the read data will be available or the write transaction will be finished. A value of 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.

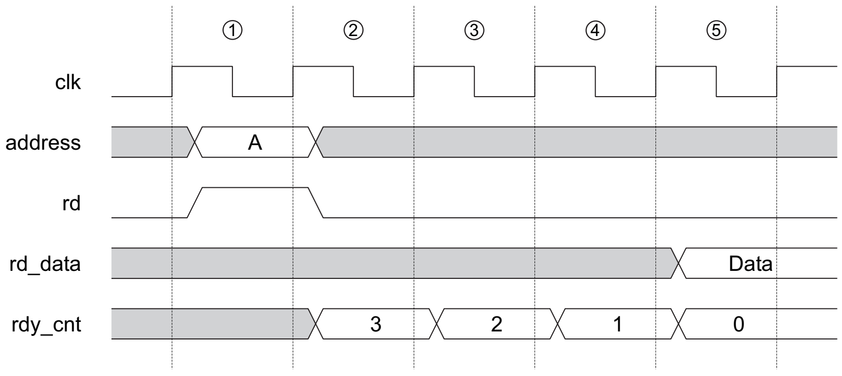

Figure 2: Read transaction with wait states

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 its pipeline. If necessary, an extension for a longer pipeline is straightforward with a larger `rdy_cnt`.[2]

Idle slaves will keep the former value of 0 for `rdy_cnt`. Slaves that do not know in advance how many wait states are needed for the transaction can produce sequences that omit any of the numbers 3, 2, and 1. A simple slave can hold `rdy_cnt` on 3 until the data is available and set it then directly to 0. The master has to handle those situations. Practically, this reduces the possibilities of pipelining and therefore the performance of the interconnect. The master will read the data later, which is not an issue as the data stays valid.

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` gets 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 until 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 the write data during cycle 1 and performs the write operation 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 clock 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

[2]The maximum value of the ready counter is relevant for the early restart of a waiting master. A longer latency from the slave e.g., for DDR SDRAM, will map to the maximum value of the counter for the first cycles.

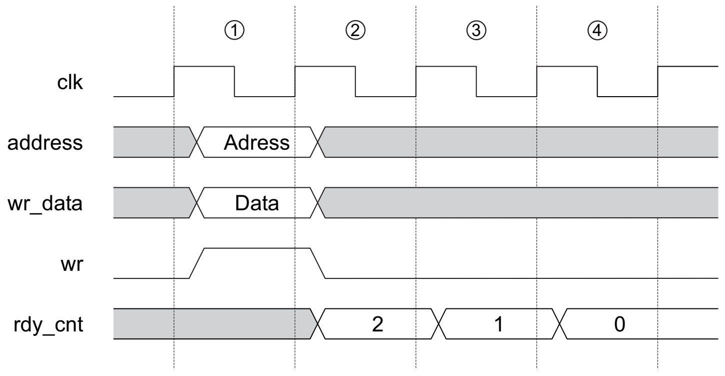

Figure 3: Write transaction with wait states

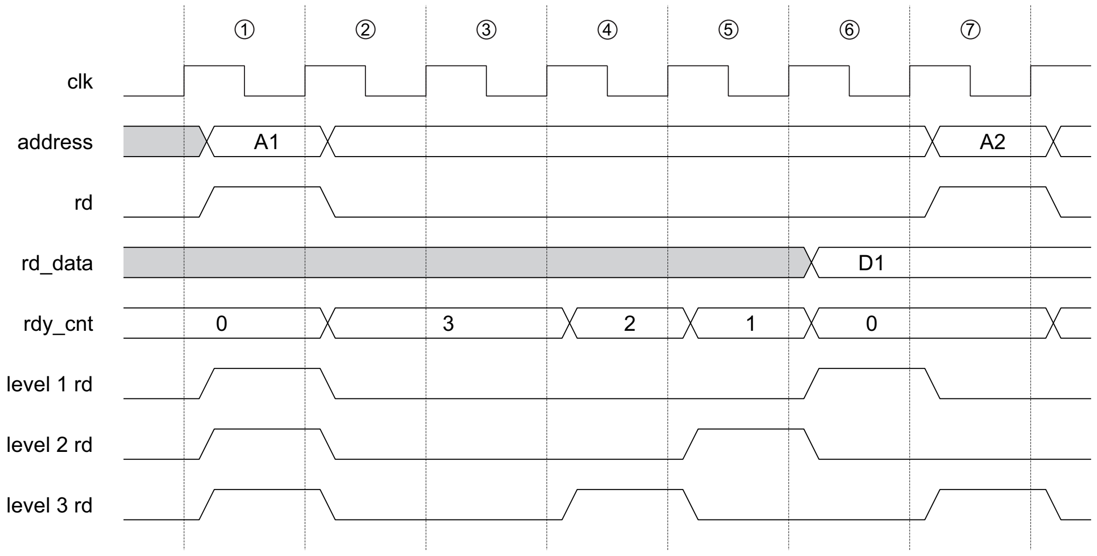

Figure 4: Different pipeline levels for a read transaction

(only the `rd` signal is shown, address lines are omitted from the figure) can 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). At pipeline level 2, a new transaction (either read or write) can start when `rdy_cnt` is 1, for pipeline level 3 the next transaction can start at a `rdy_cnt` of 2.

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

To implement level 3 pipelining in the slave, at least one 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 that this transition is combinatorial. We see in Figure 4 that `rdy_cnt` value of 3 means three or more cycles until the data is available and can therefore not be used to trigger a new transaction. Extension to an even deeper pipeline needs a wider `rdy_cnt`.

5 Interconnect

Although the definition of SimpCon is from a single master/slave point-to-point viewpoint, all variations of multiple slave and multiple master devices are possible.

5.0.5 Slave Multiplexing

To add several slaves to a single master, `rd_data` and `rdy_cnt` have to be multiplexed. Due to the fact that all `rd_data` signals are already 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 at least needed one cycle later. Therefore it can be registered to further speed up the multiplexer.

5.0.6 Master Multiplexing

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 an interconnect between the masters and the arbiter and the arbiter and the slaves. In this case the arbiter acts as a slave for the master and as a master for the peripheral devices. An example of an arbiter for SimpCon, where JOP and a VGA controller are two masters for a shared main memory, can be found in [10]. The same arbiter is also used to build a chip-multiprocessor version of JOP [12].

The missing arbitration protocol in SimpCon results in the need to queue $n-1$ requests in an arbiter for n masters. However, this additional hardware results in a zero cycle bus grant. The master, which gets the bus granted, starts the slave transaction in the same cycle as the original read/write request.

To add several slaves to a single master the `rd_data` and `rdy_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.

6 Examples

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

6.1 I/O Port

Figure 5 shows a simple I/O port with a minimal SimpCon interface. As address decoding is omitted for this simple device signal `address` is not needed. Furthermore we can tie `rdy_cnt` to 0. We only need the `rd` or `wr` signal to enable the port. Listing 1 shows the VHDL code for this I/O port.

```
entity  sc_test_slave  is  generic ( addr_bits  :  integer );

port  (
      clk          :  in   std_logic ;
      reset        :  in   std_logic ;

−− SimpConn interface

      wr_data              :  in   std_logic_vector (31  downto 0);
      rd,  wr              :  in   std_logic ;
      rd_data              :  out  std_logic_vector (31  downto 0);
      rdy_cnt              :  out  unsigned(1 downto 0);

−− input/output ports

      in_data              :  in   std_logic_vector (31  downto 0)
      out_data             :  out  std_logic_vector (31  downto 0)

);  end sc_test_slave;

architecture  rtl  of  sc_test_slave  is

begin

      rdy_cnt  <= "00";      −− no wait states

process(clk,  reset)  begin

      if  (reset='1')  then
            rd_data  <= (others => '0');
            out_data  <= (others => '0');
      elsif  rising_edge(clk)  then
            if  rd='1'  then
                  rd_data  <= in_data;
            end  if;
            if  wr='1'  then
                  out_data  <= wr_data;
            end  if;
      end  if;

end process;

end  rtl;
```

Listing 1: VHDL source for the simple input/output port

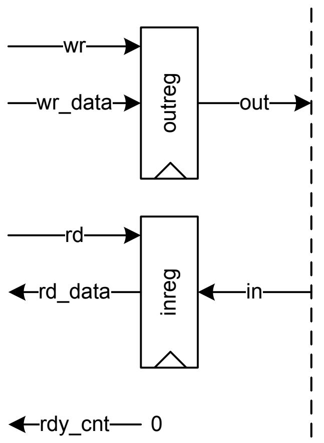

Figure 5: A simple input/output port with a SimpCon interface

6.2 SRAM interface

The following example assumes a master (processor) clocked at 100 MHz and an static RAM (SRAM) with 15 ns access time. Therefore the minimum access time for the SRAM is two cycles. The slack time of 5 ns forces us to use output registers for the SRAM address and write data and input registers for the read data in the I/O cells of the FPGA. These registers fit nicely with the intention of SimpCon to use registers inside the slave.

Figure 6 shows the memory interface for a non-pipelined read access followed by a write access. Four signals are driven by the master and two signals by the slave. The lower half of the figure shows the signals at the FPGA pins where the SRAM 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 cycle 1. Due to the placement of the registers in the I/O cells, the address and control signals are valid at the FPGA pins very early in cycle 2. At the end of cycle 3 (15 ns after `address`, `ncs` and `noe` are stable) the data from the SRAM is available and can be sampled with the rising edge for cycle 4. The setup time for the read register is short, as the register can be placed in the I/O cell. The master reads the data in cycle 4 and starts a write transaction in cycle 5. Address and data are again registered by the slave and are available for the SRAM 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 7 we see a pipelined read from the SRAM with pipeline level 2. With this pipeline level and the two cycles read access time of the SRAM we achieve the maximum possible bandwidth.

We can see the start of the second read transaction in cycle 3 during the read of the first data from the SRAM. The new address is registered in the same cycle and available for the SRAM 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) and the master has the freedom to select one of the two cycles to read the data.

It has to be noted that pipelining with one read per cycle is possible with SimpCon. We just

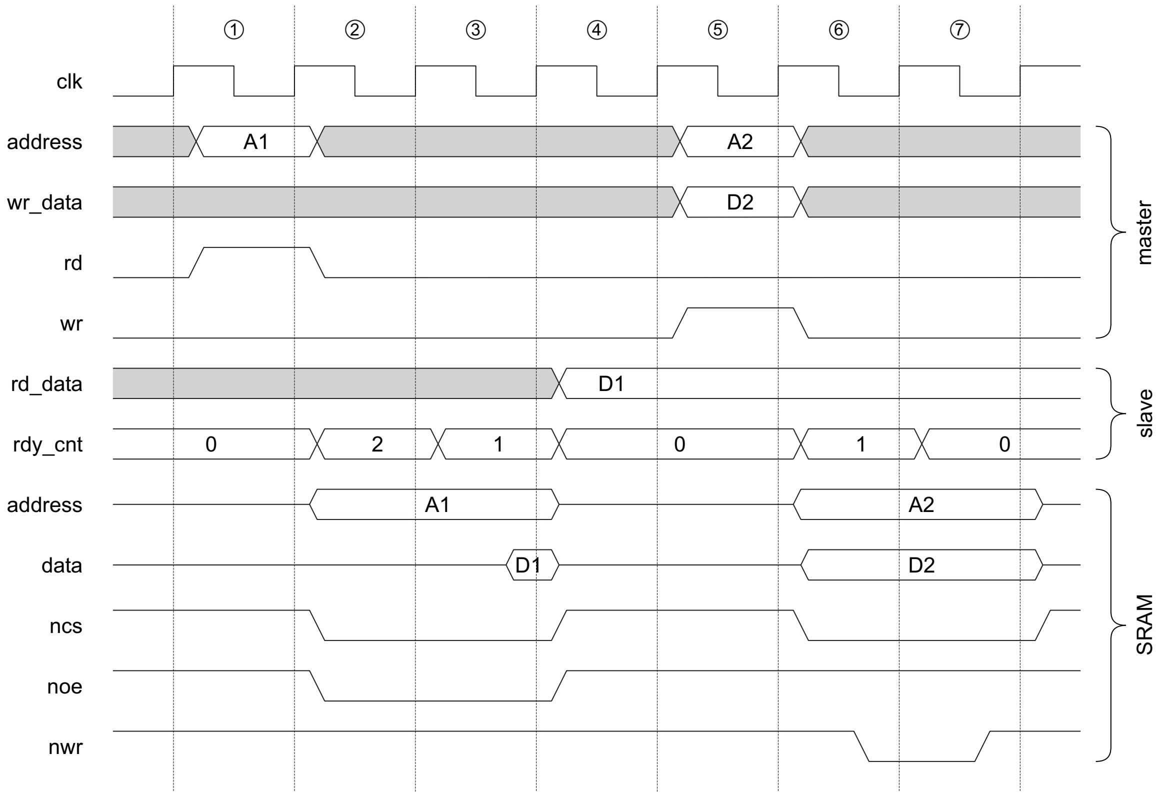

Figure 6: Static RAM interface without pipelining

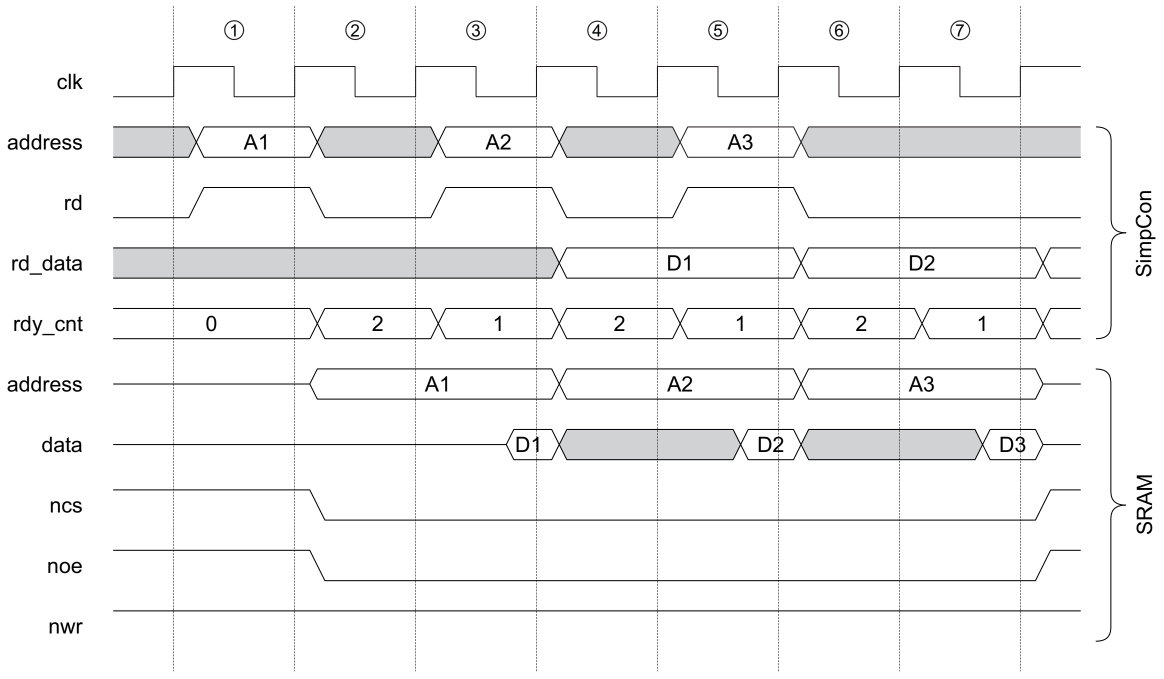

Figure 7: Pipelined read from a static RAM

showed a 2 cycle slave in this example. For a SDRAM memory interface the ready counter will stay either at 2 or 1 during the single cycle reads (depending on the slave pipeline level). It will go down to 0 only for the last data word to read.

7 Available VHDL Files

Besides the SimpCon documentation, some example VHDL files for slave devices and bridges are available from `http://opencores.org/?do=project&who=simpcon`. All components are also part of the standard JOP distribution.

7.1 Components

- sc_pack.vhd defines VHDL records and some constants.
- sc_test_slave.vhd is a very simple SimpCon device. A counter to be read out and a register that can be written and read. There is no connection to the outer world. This example can be used as basis for a new SimpCon device.
- sc_sram16.vhd is a memory controller for 16-bit SRAM.
- sc_sram32.vhd is a memory controller for 32-bit SRAM.
- sc_sram32_flash.vhd is a memory controller for 32-bit SRAM, a NOR Flash, and a NAND Flash as used in the Cycore FPGA board for JOP.
- sc_uart.vhd is a simple UART with configurable baud rate and FIFO width.
- sc_usb.vhd is an interface to the parallel port of the FTDI 2232 USB chip. The register definition is identical to the UART and the USB connection can be used as a drop in replacement for a UART.
- sc_isa.vhd interfaces the old ISA bus. It can be used for the popular CS8900 Ethernet chip.
- sc_sigdel.vhd is a configurable sigma-delta ADC for an FPGA that needs just two external components: a capacitor and a resistor.
- sc_fpu.vhd provides an interface to the 32-bit FPU available at `www.opencores.org`.
- sc_arbiter_*.vhd different zero cycle SimpCon arbiters for CMP systems written by Christof Pitter [11].

7.2 Bridges

- sc2wb.vhd is a SimpCon/Wishbone [9] bridge.
- sc2avalon.vhd is a SimpCon/Avalon [2] bridge to integrate a SimpCon based design with Altera's SOPC Builder [3].
- sc2ahbsl.vhd provides an interface to AHB slaves as defined in Gaisler's GRLIB [6]. Many of the available GPL AHB modules from the GRLIB can be used in a SimpCon based design.

8 Why a New Interconnection Standard?

There are many interconnection standards available for SoC designs. The natural question is: Why propose a new one? The answer is given in the following section. In summary, the available standards are still in the tradition of backplane busses and do not fit very well for pipelined on-chip interconnections.

8.1 Common SoC Interconnections

Several point-to-point and bus standards have been proposed. The following section gives a brief overview of common SoC interconnection standards.

The Advanced Microcontroller Bus Architecture (AMBA) [4] is the interconnection definition from ARM. The specification defines three different busses: Advanced High-performance Bus (AHB), Advanced System Bus (ASB), and Advanced Peripheral Bus (APB). The AHB is used to connect on-chip memory, cache, and external memory to the processor. Peripheral devices are connected to the APB. A bridge connects the AHB to the lower bandwidth APB. An AHB bus transfer can be one cycle at burst operation. With the APB a bus transfer requires two cycles and no burst mode is available. Peripheral bus cycles with wait states are added in the version 3 of the APB specification. ASB is the predecessor of AHB and is not recommended for new designs (ASB uses both clock phases for the bus signals – very uncommon for today's synchronous designs). The AMBA 3 AXI (Advanced eXtensible Interface) [5] is the latest extension to AMBA. AXI introduces out-of-order transaction completion with the help of a 4 bit transaction ID tag. A ready signal acknowledges the transaction start. The master has to hold the transaction information (e.g. address) until the interconnect signals ready. This enhancement ruins the elegant single cycle address phase from the original AHB specification.

Wishbone [9] is a public domain standard used by several open-source IP cores. 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,[3] 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 *before* the Wishbone connect

[3]Multiplexers are used instead of busses to connect several slaves and masters.

or an expensive (time and resources) multiplexer 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. In that case the address decoding in the slave can be performed in the same cycle as the address is registered. A similar issue, with respect to the master, exists for the output data from the slave: As it is only valid for a single cycle, the data has to be registered by the master when the master is not reading it immediately. Therefore, the slave should keep the last valid data at its output even when the Wishbone strobe signal (*wb.stb*) is not assigned anymore. Holding the data in the slave is usually *for free* from the hardware complexity – it is *just* a specification issue. In the Wishbone specification 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 achieve good performance.

The Avalon [2] interface specification is provided by Altera for a system-on-a-programmable-chip (SOPC) interconnection. Avalon defines a great range of interconnection devices ranging from a simple asynchronous interface intended for direct static RAM connection up to sophisticated pipeline transfers with variable latencies. This great flexibility provides an easy path to connect a peripheral device to Avalon. How is this flexibility possible? The *Avalon Switch Fabric* translates between all those different interconnection types. The switch fabric is generated by Altera's SOPC Builder tool. However, it seems that this switch fabric is Altera proprietary, thus tying this specification to Altera FPGAs.

The On-Chip Peripheral Bus (OPB) [7] is an open standard provided by IBM and used by Xilinx. The OPB specifies a bus for multiple masters and slaves. The implementation of the bus is not directly defined in the specification. A distributed ring, a centralized multiplexer, or a centralized AND/OR network are suggested. Xilinx uses the AND/OR approach and all masters and slaves must drive the data busses to zero when inactive.

Sonics Inc. defined the Open Core Protocol (OCP) [8] as an open, freely available standard. The standard is now handled by the OCP International Partnership[4].

8.2 What's Wrong with the Classic Standards?

All SoC interconnection standards, which are widely in use, are still in the tradition of a backplane bus. They force the master to hold the address and control signals until the slave provides the data or acknowledges the write request. However, this is not necessary in a clocked, synchronous system. Why should we force the master to hold the signals? Let the master move on after submitting the request in a single cycle. Forcing the address and control valid for the complete request disables any form of pipelined requests.

Figure 8 shows a read transaction with wait states as defined in Wishbone [9], Avalon [2], OPB [7], and OCP [8].[5] The master issues the read request and the address in cycle 1. The slave has to reset the `ack` in the same cycle. When the slave data is available, the acknowledge signal is set (`ack` in cycle 3). The master has to read the data and register them within the same clock cycle. The master has to hold the address, write data, and control signal active until the acknowledgement from the slave arrives. For pipelined reads, the `ack` signal can be

[4] `www.ocpip.org`

[5] The signal names are different, but the principle is the same for all mentioned busses.

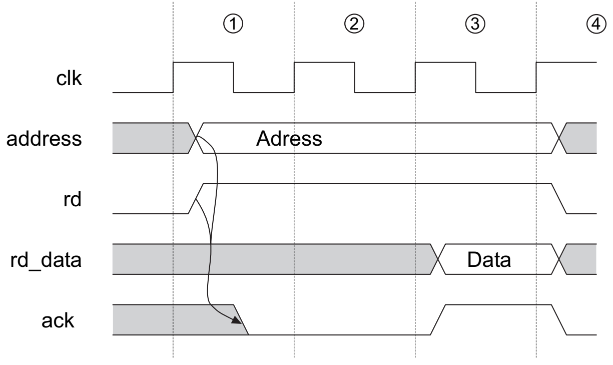

Figure 8: Classic basic read transaction

split into two signals (available in Avalon and OCP): one to accept the request and a second one to signal the available data.

The master is blind about the status of the outstanding transaction until it is finished. It could be possible that the slave informs the master in how many cycles the result will be available. This information can help in building deeply pipelined masters.

Only the AMBA AHB [4] defines a different protocol. A single cycle address phase followed by a variable length data phase. The slave acknowledgement (HREADY) is only necessary in the data phase avoiding the combinatorial path from address/command to the acknowledgement. Overlapping address and data phase is allowed and recommended for high performance. Compared to SimpCon, AMBA AHB allows for single stage pipelining, whereas SimpCon makes multi-stage pipelining possible using the ready counter (`rdy_cnt`). The `rdy_cnt` signal defines the delay between the address and the data on a read, signalled by the slave. Therefore, the pipeline depth of the bus and the slaves is only limited by the bit width of `rdy_cnt`.

Another issue with all interconnection standards is the single cycle availability of the read data at the slaves. Why not keep the read data valid as long as there is no new read data available? This feature would allow the master to be more flexible when to read the data. It would allow issuing a read command and then continuing with other instructions – a feature known as data pre-fetching to hide long latencies.

The last argument sounds contradictory to the first argument: provide the transaction data at the master just for a single cycle, but request the slave to hold the data for several cycles. However, it is motivated by the need to free up the master, keep it *moving*, and move the data hold (register) burden into the slave. As data processing bottlenecks are usually found in the master devices, it sounds natural to move as much work as possible to the slave devices to free up the master.

Avalon, Wishbone, and OPB provide a single cycle latency access to slaves due to the possibility of acknowledging a request in the same cycle. However, this feature is a scaling issue for larger systems. There is a combinatorial path from master address/command to address decoding, slave decision on `ack`, slave `ack` multiplexing back to the master and the master decision to hold address/command or read the data and continue. Also, the slave output data multiplexer is on a combinatorial path from the master address.

Performance	Memory	Interconnect
16,633	32 bit SRAM	SimpCon
14,259	32 bit SRAM	AMBA
14,015	32 bit SRAM	Avalon/PTF
13,920	32 bit SRAM	Avalon/VHDL
15,762	32 bit on-chip	Avalon
14,760	16 bit SRAM	SimpCon
11,322	16 bit SRAM	Avalon
7,288	16 bit SDRAM	Avalon

Table 2: JOP performance with different interconnection types

AMBA, AHB, and SimpCon avoid this scaling issue by requesting the acknowledge in the cycle following the command. In SimpCon and AMBA, the select for the read data multiplexer can be registered as the read address is known at least one cycle before the data is available. The later acknowledgement results in a minor drawback on SimpCon and AMBA (nothing is for free): It is not possible to perform a single cycle read or write without pipelining. A single, non pipelined transaction takes two cycles without a wait state. However, a single cycle read transaction is only possible for very simple slaves. Most non-trivial slaves (e.g. memory interfaces) will not allow a single cycle access anyway.

8.3 Evaluation

We compare the SimpCon interface with the AMBA and the Avalon interface as two examples of common interconnection standards. As an evaluation example, we interface an external asynchronous SRAM with a tight timing. The system is clocked at 100 MHz and the access time for the SRAM is 15 ns. Therefore, there are 5 ns available for on-chip register to SRAM input and SRAM output to on-chip register delays. As an SoC, we use an actual low-cost FPGA (Cyclone EP1C6 [1] and a Cyclone II).

The master is a Java processor (JOP [14, 17]). The processor is configured with a 4 KB instruction cache and a 512 byte on-chip stack cache. We run a complete application benchmark on the different systems. The embedded benchmark (*Kfl* as described in [13]) is an industrial control application already in production.

Table 2 shows the performance numbers of this JOP/SRAM interface on the embedded benchmark. It measures iterations per second and therefore higher numbers are better. One iteration is the execution of the main control loop of the *Kfl* application. For a 32 bit SRAM interface, we compare SimpCon against AMBA and Avalon. SimpCon outperforms AMBA by 17% and Avalon by 19%[6] on a 32 bit SRAM.

The AMBA experiment uses the SRAM controller provided as part of GRLIB [6] by Gaisler Research. We avoided writing our own AMBA slave to verify that the AMBA implementa-

[6]The performance is the measurement of the execution time of the whole application, not only the difference between the bus transactions.

tion on JOP is correct. To provide a fair comparison between the single master solutions with SimpCon and Avalon, the AMBA bus was configured without an arbiter. JOP is connected directly to the AMBA memory slave. The difference between the SimpCon and the AMBA performance can be explained by two facts: (1) as with the Avalon interconnect, the master has the information when the slave request is ready at the same cycle when the data is available (compared to the `rdy_cnt` feature); (2) the SRAM controller is conservative as it asserts `HREADY` one cycle later than the data is available in the read register (`HRDATA`). The second issue can be overcome by a better implementation of the SRAM AMBA slave.

The Avalon experiment considers two versions: an SOPC Builder generated interface (PTF) to the memory and a memory interface written in VHDL. The SOPC Builder interface performs slightly better than the VHDL version that generates the Avalon `waitrequest` signal. It is assumed that the SOPC Builder version uses fixed wait states within the switch fabric.

We also implemented an Avalon interface to a single-cycle on-chip memory. SimpCon is even faster with the 32 bit off-chip SRAM than with the on-chip memory connected via Avalon. Furthermore, we also performed experiments with a 16 bit memory interface to the same SRAM. With this smaller data width the pressure on the interconnection and memory interface is higher. As a result the difference between SimpCon and Avalon gets larger (30%) on the 16 bit SRAM interface. To complete the picture we also measured the performance with an SDRAM memory connected to the Avalon bus. We see that the large latency of an SDRAM is a big performance issue for the Java processor.

9 Summary

This document describes a simple (with respect to the definition and implementation) and efficient SoC interconnect [15]. The novel signal `rdy_cnt` allows an early signalling to the master when read data will be valid. This feature allows the master to restart a stalled pipeline earlier to react for arriving data. Furthermore, this feature also enables pipelined bus transactions with a minimal effort on the master and the slave side.

We have compared SimpCon quantitative with AMBA and Avalon, two common interconnection definitions. The application benchmark shows a performance advantage of SimpCon by 17% over AMBA and 19% over Avalon interfaces to an SRAM.

SimpCon is used as the main interconnect for the Java processor JOP in a single master, multiple salves configuration. SimpCon is also used to implement a shared memory chip-multiprocessor version of JOP. Furthermore, in a research project on time-triggered network-on-chip [16] SimpCon is used as the *socket* to this NoC.

The author thanks Kevin Jennings and Tommy Thorn for the interesting discussions about SimpCon, Avalon, and on-chip interconnection in general at the Usenet newsgroup `comp.arch.fpga`.

References

[1] Altera. Cyclone FPGA Family Data Sheet, ver. 1.2, April 2003.

[2] Altera. Avalon interface specification, April 2005.

[3] Altera. Quartus ii version 7.1 handbook, May 2007.

[4] ARM. AMBA specification (rev 2.0), May 1999.

[5] ARM. AMBA AXI protocol v1.0 specification, March 2004.

[6] Jiri Gaisler, Edvin Catovic, Marko Isomäki, Kristoffer Carlsson, and Sandi Habinc. GRLIB IP core user's manual, version 1.0.14. Available at http://www.gaisler.com/, February 2007.

[7] IBM. On-chip peripheral bus architecture specifications v2.1, April 2001.

[8] OCP-IP Association. Open core protocol specification 2.1. http://www.ocpip.org/, 2005.

[9] Wade D. Peterson. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores, revision: B.3. Available at http://www.opencores.org, September 2002.

[10] Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, pages 317 – 322, Amsterdam, Netherlands, August 2007.

[11] Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 144–151, Vienna, Austria, September 2007. ACM Press.

[12] Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.

[13] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[14] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[15] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[16] Martin Schoeberl. A time-triggered network-on-chip. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, pages 377 – 382, Amsterdam, Netherlands, August 2007.

[17] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.