HPES
High Performance Embedded Systems

EEE4120F

University of Cape Town • iYunivesithi yaseKapa • Universiteit van Kaapstad

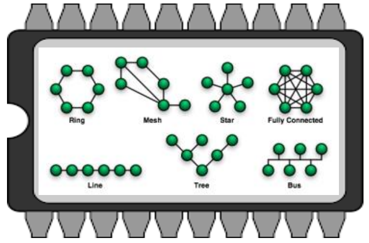
SPES BONA

High Performance Embedded Systems

Lecture 22

Memories Controllers (part 2), On-chip Interfacing Standards, Wishbone

Lecturer:
Simon Winberg



Lecture Overview

- Memory Control Units (part 2 of 2)
  - Dual-port memory control unit
  - Setting up memory in code & simulation
  - FIFO and LIFO memory modules
- On-chip Interfacing Standards*
  - Wishbone
  - The Altera/Intel Avalon Bus

* The topics on these two commonly used interfacing standards are optional (supplementary reading) this year

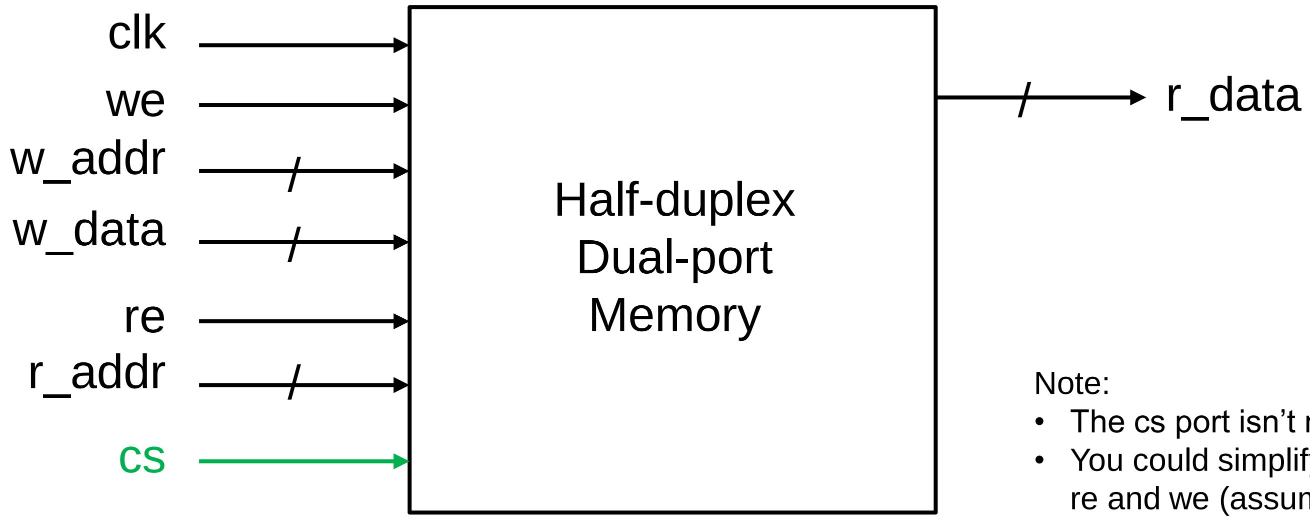
Memory Controllers
(part 2 of 2)

EEE4120F

Dual-port Memory Control Unit

half-duplexed channels, one read/one write

This is a usual (implicit handshaking) interface for a memory control unit



Explanation of ports:

clk : clock input

w_addr, w_data : write address and data

r_addr, r_data : read data address and data

we, re : write enable, read enable

cs : chip select (i.e. chip ignores inputs if cs=0)

Note:

- The cs port isn’t necessarily needed.
- You could simplify this to dropping the re and we (assume always active) and use instead a cs line to decide whether or not to activate the module.
- This module allows up to one read and one write simultaneously
- It is undefined what the r_data will be if you try and write and read at the same time to the same address (the w_data will get written to the location but the returned read value might be the old or the new value)

https://www.edaplayground.com/x/4eSi

Dual-port Memory Control Unit

```
// a simple dual-port RAM in Verilog
// Test in EDAPlayground at: https://www.edaplayground.com/x/4eSi
module hdp_ram (
  clk    ,   // clock input
  we     ,   // write enable
  w_addr ,   // write address
  w_data ,   // write data
  re     ,   // read enable
  r_addr ,   // read address
  cs     ,   // chip select (i.e. chip ignores inputs if cs=0)
  r_data     // output for read operation
);
  // Setup some parameters
  parameter DATA_WIDTH = 8;  // word size of the memory
  parameter ADDR_WIDTH = 8;  // number of memory words, e.g. 2^8-1
  parameter RAM_DEPTH  = 1 << ADDR_WIDTH;
  // Define inputs
  input clk, we, re, cs;
  input  [ADDR_WIDTH-1:0] r_addr, w_addr;
  input  [DATA_WIDTH-1:0] w_data;
  output reg [DATA_WIDTH-1:0] r_data;
  // Private registers
  reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1]; // Set up the memory array

  // Write to or read from memory
  always@ (posedge clk)
  begin
  if (cs)
   begin
     if (we) mem[w_addr] <= w_data;
     if (re) r_data <= mem[r_addr];
   end
  end
endmodule
```

As you can see, the implementation is easier than the single port MCU as it does not need to use a inout tristate port for the data.

https://www.edaplayground.com/x/4eSi

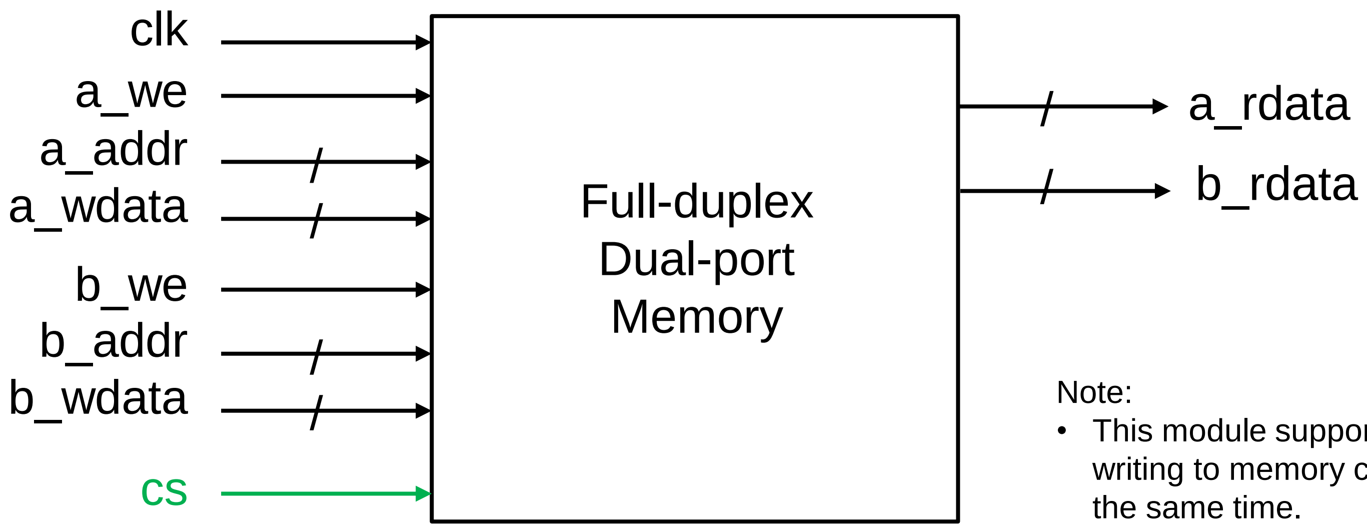
Dual-port Memory Control Unit Testbench

```
// EEE4120F Memory Control Unit Example
// Testbench for the dual port dp_ram RAM control memory unit

module hdp_ram_tb ();
  reg  clk, cs, we, re;
  reg  [7:0] w_data;  // this is the connection to dp_ram write data port
  reg  [7:0] w_addr;  // address to write to
  reg  [7:0] r_addr;  // address to read from
  wire [7:0] r_data;  // link to data returned on a read

  // Instantiate the module to be tested
  hdp_ram hdp_ram_uut(clk,we,w_addr,w_data,re,r_addr,cs,r_data);


  initial begin
    // set up initial conditions
    clk   = 0; cs    = 0; we    = 1;
    re    = 0; r_addr= 1; w_addr= 1;
    w_data= 100;

    $display("clk cs we re raddr waddr rdata wdata");
    $monitor("%b   %b  %b  %b  %03d   %03d   %03d   %03d",
             clk,cs,we,re,r_addr,w_addr,r_data,w_data);

    // set up to write to 100 to [1] and disable read:
    cs <= 1; #5 $display("write 100 to [1]");  // (the #5 is here to force the display output)
    #5 clk = ~clk; #5 clk = ~clk; // do a clock pulse

    re <= 1; w_addr <= 2; w_data <= 101;
    #5 $display("write 101 to [2] and read from 1");
    #5 clk = ~clk; #5 clk = ~clk; // do a clock pulse

  end
endmodule
```

https://www.edaplayground.com/x/4eSi

Full Dual-port Memory Control Unit
full-duplex channels, read&write per channel

This is basically like the dual port memory, but it has a write enable for both data ports, and separate datain and dataout ports so that a dual read or write can be done.



Note:
- This module supports reading or writing to memory channel a or b at the same time.

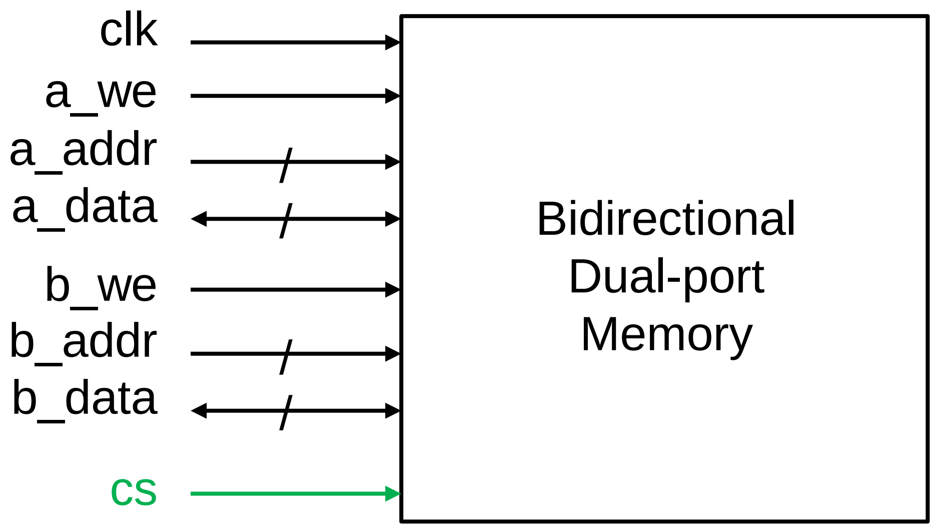
Explanation of ports:
clk : clock input
a_we, b_we: write enable/read enable for a,b
a_addr, b_addr : address for channel a and b
a_wdata, b_wdata : data to write for a,b if we=1
a_rdata, b_rdata : data read for a,b if we=0
cs : chip select (i.e. chip ignores inputs if cs=0)

The code for this is fairly easy to construct using the previous examples. See below link to access the code.

https://www.edaplayground.com/x/64Vt

Full Dual-port Memory Control Unit
with bidirectional channels

This is basically like the dual port memory, but it has a write enable for both data ports, and separate datain and dataout ports so that a dual read or write can be done.



Note:
- This module supports reading or writing to either memory channel a or b at the same time

Explanation of ports:
clk : clock input
a_we, b_we: write enable/read enable for a,b
a_addr, b_addr : address for channel a and b
a_data, b_data : data to write/read for a,b
cs : chip select (i.e. chip ignores inputs if cs=0)

The code for this is fairly easy to construct using the earlier examples of single-port and half-duplex ram. A link to a completed example is below:

https://www.edaplayground.com/x/5d3t

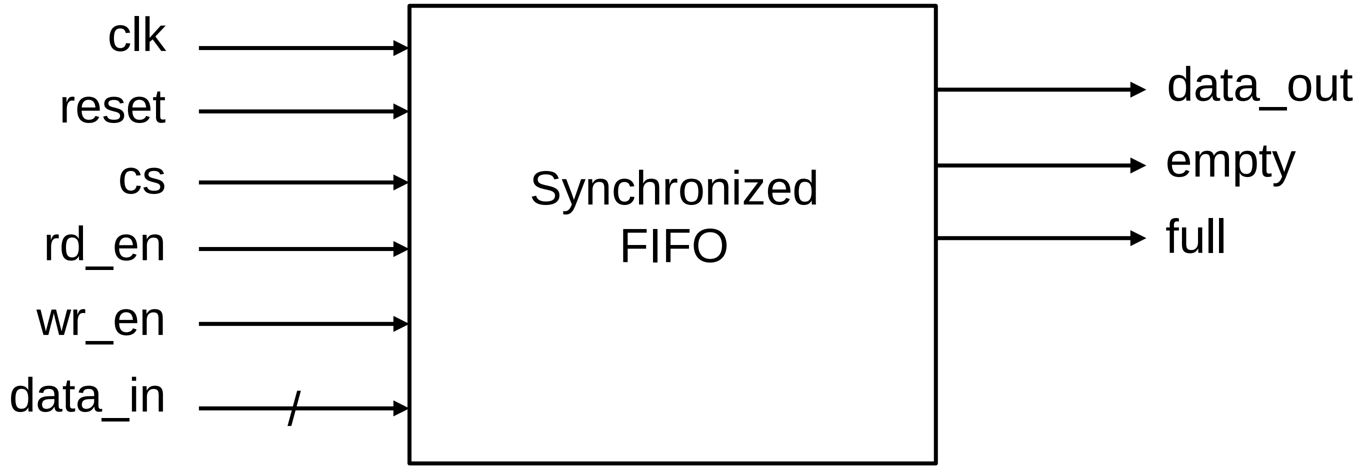
FIFOs and LIFOs
(a specialized type of memory)

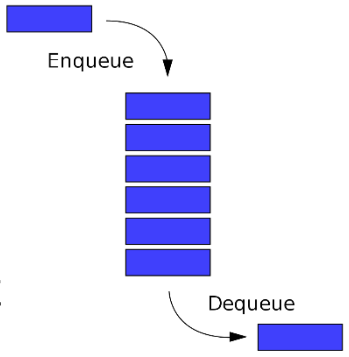
EEE4120F

FIFOs and LIFOs

- First In First Out (FIFO)
  - These are useful for I/O buffering, e.g. for streaming data from serial
- Last In First Out (LIFO) or stack
  - A stack can be used similarly to how it is used in software, or to sequences operations (e.g. reverse polish calculator).

FIFO Verilog Interface





FIFO structure

The module would be designed around a certain size, e.g.:
  // 4-element memory array for fifo queue
  reg [7:0] fifomem [0:3];

The working of the module would be as expected, the empty line will be high when no data is in the queue, and full will be high when the queue is full. A head and tail wrapping index would likely be used. But, when the queue is full and a new item is added, the head and tail would both need to increment to make the last item 'fall' out of the queue.

```
module sfifo (
  clk       , // Clock input
  reset     , // high reset
  cs        , // chip select
  rd_en     , // read enable
  wr_en     , // write enable
  data_in   , // data input
  data_out  , // data output
  empty     , // FIFO empty
  full        // FIFO full
);
```

Examples FIFOs online

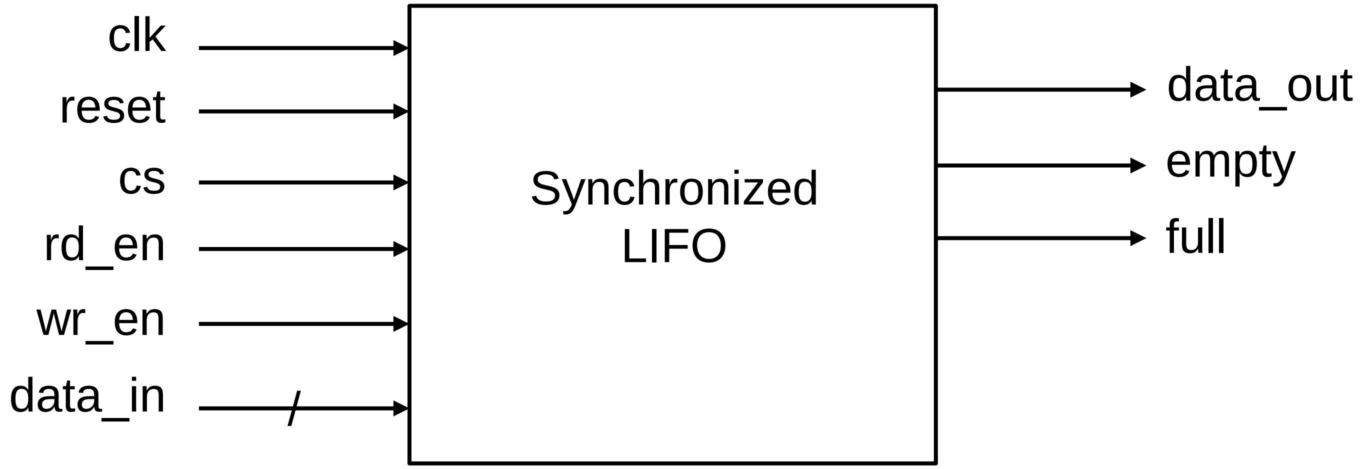
See example code at:

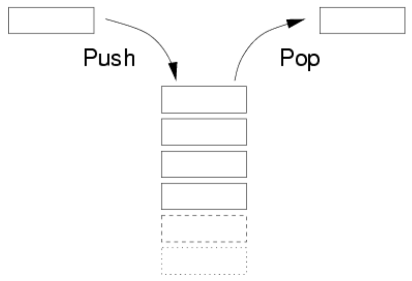
https://opencores.org/projects/generic_fifos

https://www.fpga4student.com/2017/01/verilog-code-for-fifo-memory.html

Notes on comms: if you are just sending data, and not at a very fast rate, then you probably do not need to worry with a FIFO, to save space. But if you are streaming serial data, where it is a problem to have the accastional missing byte, then you will probably want a FIFO – and likely handshaking logic if not flow control to avoid the potential for data being lost.

LIFO Interface





LIFO structure

The module can use the same interface as a FIFO, it’s just that the internal workings will be such that input will be extracted in the reverse order that they are put into the stack. A top index (as apposed to a head and tail) would be used. Like the FIFO, the most recent addition is typically the most desirable to keep when the stack is full, so the top pointer will just continue being incremented (wrapping at the last address) which will keep the most recent data in the stack.

Setting up memory in simulation

- You could hardcode ROM / initialize RAM, e.g.
- Initializing the memory

```
reg [7:0] mem [7:0] = {
    7'b101_1001, // [0]
    7'b001_1100, // [1]
    7'b101_1011, // [2]
    7'b010_1001, // [3]
    7'b110_0110, // [4]
    7'b011_1001, // [5]
    7'b101_1001, // [6]
    7'b100_0001, // [7]
 };
```

- Using a case statement in a RAM MCU

```
always @ (re or address)
begin
  case (address)
    0 : data = 99;
    1 : data = 12
    2 : data = 140;
    3 : data = 110;
    …
  endcase
end
```

Initializing memory in simulation
(using a macro)

```
// You can also intialize your memory as follows…

module mymod_tb ();
 parameter MEM_SIZE = 256
 reg [7:0] mem [0:MEM_SIZE -1]

 initial
   begin
     for (i = 0; i < MEM_SIZE - 1; i = i + 1)
      begin
        mem[i][0] = 0;
     end
 end

endmodule
```

Note that for used here is usable in simulation and when used to implement a module the for is unravelled to generate logic, it does not necessarily handle loop dependencies well.

Initializing memory in simulation (reading from file)

```
module rom (
   addr  , // Address input
   data  , // Data output
   re    , // Read Enable
   cs      // Chip Select
);
input  [7:0] addr;
output [7:0] data;
input re, cs;
reg [7:0] mem [0:255] ;

assign data = (cs && re)? mem[addr] : 8'b0;

initial begin
  $readmemb("mem.csv", mem); // mem.csv is memory file
end

endmodule
```

Usually the simulator expects mem.csv to have numbers in text, one column per row, e.g.:

```
mem.csv:
00000000
00000001
00000010
00000011
….
```

(stores 0,1,2,3… as binary)

Optional supplementary reading

Slides 17 – 34 not examined this year!

Wishbone bus

A brief view of the wishbone bus architecture

*Image source: 'book' PNG Designed By Grafix Point from https://pngtree.com

Optional supplementary reading

Interfacing Standards

- The Avalon bus by Altera – Open Standard
- Advanced Microcontroller Bus Architecture (AMBA) by ARM – Open Standard
- On-chip Peripheral Bus (OPB) by Xilinx
- Wishbone bus (originally developed by Silicore Corporation) – Open Standard

Optional supplementary reading

Wishbone bus

A brief view of the wishbone bus architecture

OpenCores
www.opencores.org

Optional supplementary reading

Wishbone

OpenCores
www.opencores.org

- Initially developed by Silicore Corporation.
- OpenCores has chosen to recommend Wishbone compatability for all open IP cores particularly ones added to their repository
- The Wishbone bus structure and protocol is not copyrighted and can be freely copied and distributed
- Wishbone is made to let designers combine several designs written in Verilog, VHDL or some other HDL for reuse and electronic design automation

Optional supplementary reading

Wishbone

- There are two main interfaces for Wishbone:
  - Master and slave interfaces
- Master interface
  - These are IP cores that are capable of <u>initiating</u> bus use cycles.
- Slave interface
  - These capable of accepting and responding to bus use cycles.
- Various interconnection topologies are supported by this standard, including:
  - Point-to-point connection
  - Daisy chain / Dataflow interconnection
  - Shared bus
  - Hierarchical topology / crossbar switches

Optional supplementary reading

Wishbone

- Bus size: 8, 16, 32, 64-bit
- Signals are synchronous to a single clock but some slave responses must be generated combinatorial for maximum performance.
- Wishbone permits addition of a "tag bus" to describe the data. But reset, simple addressed reads and writes, transfer of data blocks, and indivisible bus cycles all work without tags.
- Technically (by decree of OpenCores) “A device does not conform to the Wishbone specification unless it includes a data sheet to describe what it does, bus width, utilization, etc”. This promotes reuse of design.

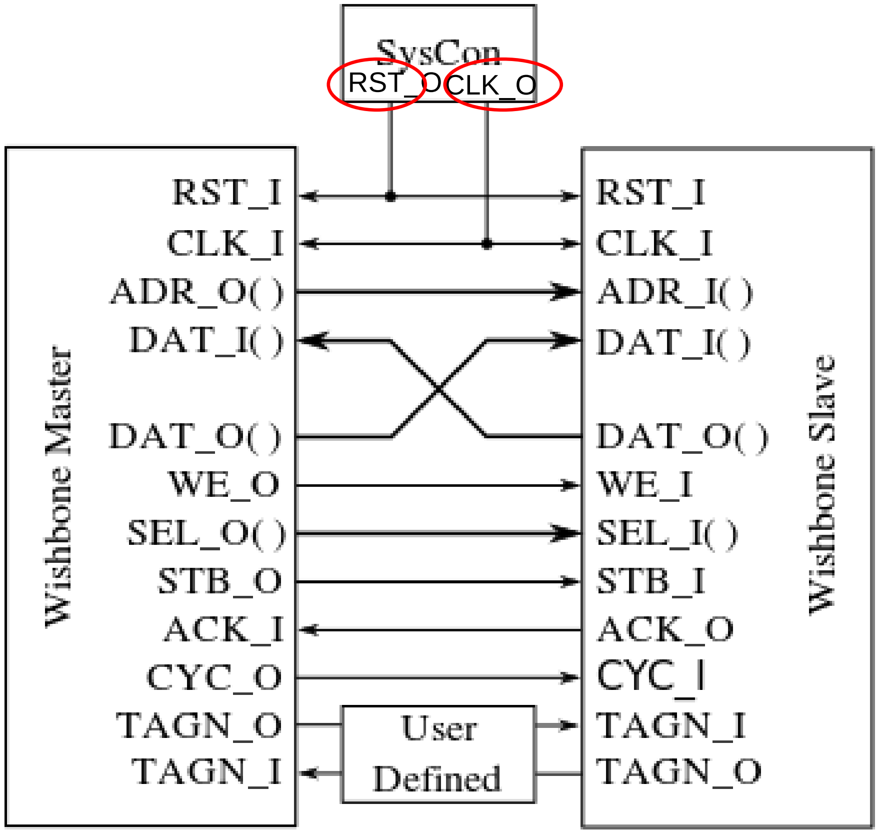
Information Source: https://en.wikipedia.org/wiki/Wishbone_(computer_bus

Wishbone – the interface

Optional supplementary reading

Global signals / SysCon control module

CLK_O : system clock output generated by SysCon module. It coordinates all activities for the internal logic. Connects to CLK_I on MASTER and SLAVE
RST_O: The reset output generated by the SysCon module. Forces all Wishbone interfaces to restart. All internal self-starting state machines are forced into an initial state.



Standard connection for timing diagrams.

Image source: https://en.wikipedia.org/wiki/Wishbone_(computer_bus)

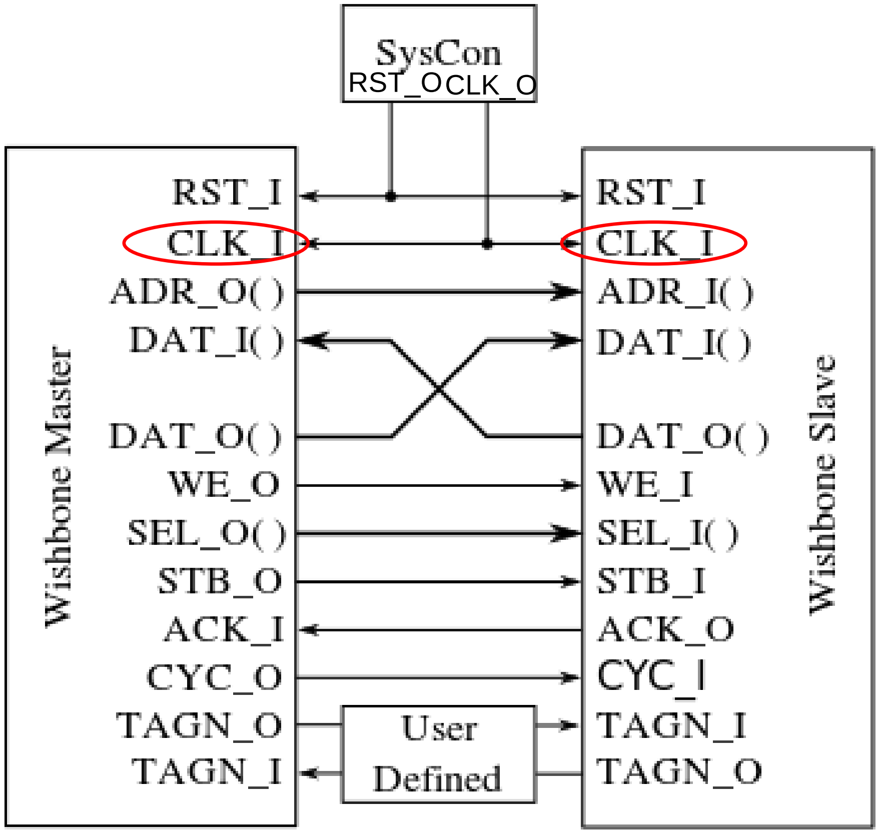
Wishbone – the interface

Optional supplementary reading

Signals Common to MASTER and SLAVE Interfaces

CLK_I: The clock input, coordinates all activities for the internal logic within the Wishbone interconnect.

- All Wishbone input signals are stable before the rising edge of CLK_I.
- Wishbone output signals are registered at the rising edge of CLK_I



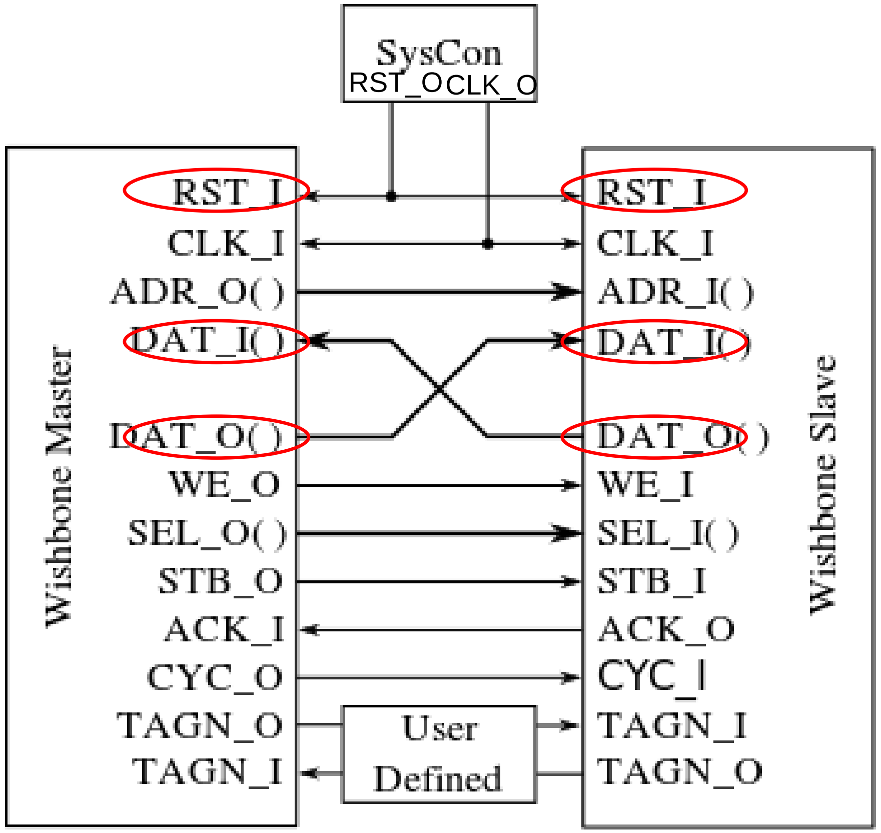
Standard connection for timing diagrams.

Wishbone – the interface

Optional supplementary reading

Signals Common to MASTER and SLAVE Interfaces

RST_I: Force Wishbone interface to restart.
DAT_I: Data input port to slave/master. The bus size is determined by the port size (8, 16, 32 or 64-bit).
DAT_O: Data output port used. Bus size as above, max 64bits.



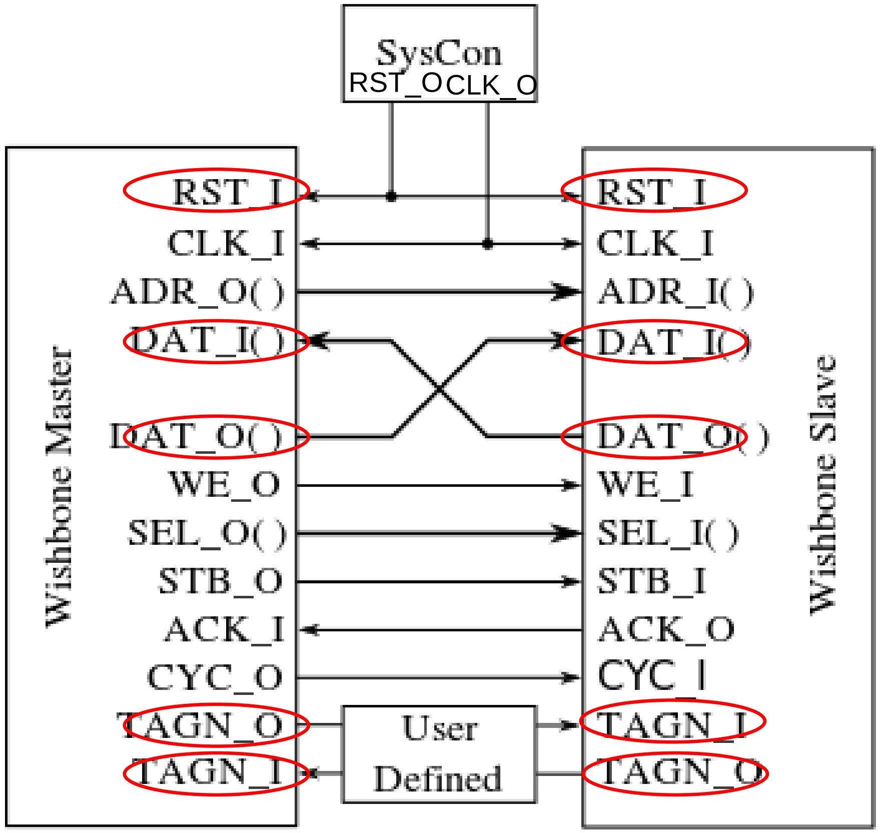
Standard connection for timing diagrams.

Wishbone – the interface

Optional supplementary reading

Signals Common to MASTER and SLAVE Interfaces

TGD_I: Tag for Data. Contains information that is associated with the data input array (e.g. type of data sent), and is qualified by signal STB_I. e.g. parity protection, error correction and time stamp information can be attached to the data bus. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification. The name and operation of a data tag must be defined in the Wishbone Datasheet for the device.
TGD_O: as per above for output



Standard connection for timing diagrams.

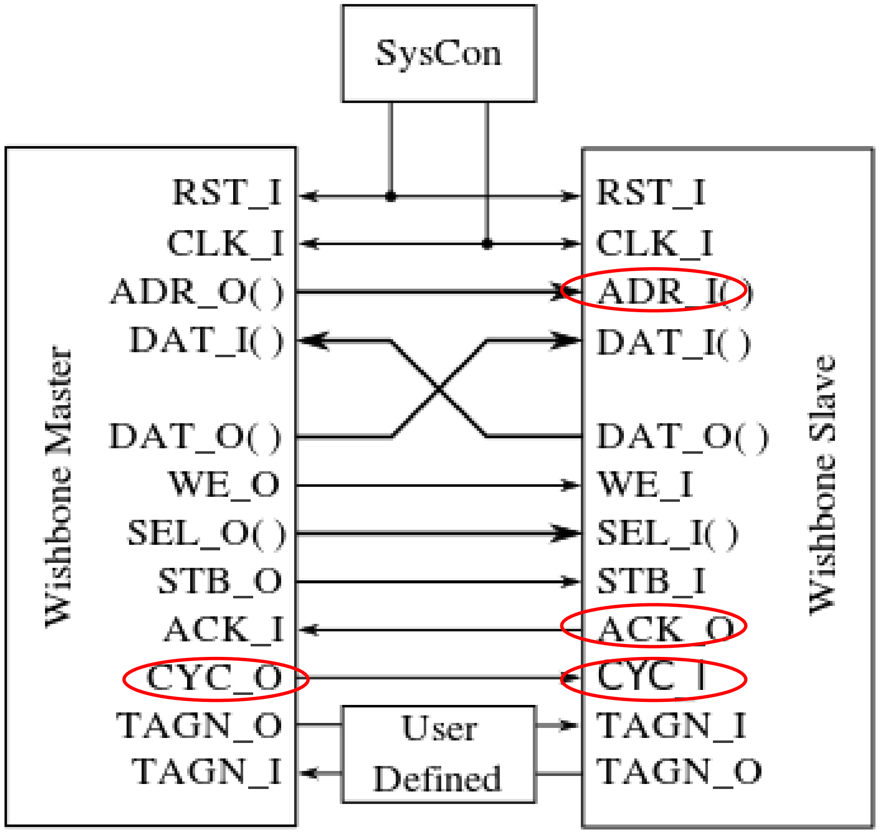
Wishbone – the interface

Optional supplementary reading

MASTER Interfaces

ACK_I: Acknowledge input, indicates the normal termination of a bus cycle (see also ERR_I & RTY_I for exception handling)
ADR_O: The address output bus, used to indicate address if device accessed.
CYC_O: The cycle output marker.

- When asserted, indicates a valid bus cycle is in progress.
- Signal is asserted for duration of all bus cycles. (e.g. during block transfer CYC_O is asserted for first data transfer and remains asserted to last data transfer)
- Useful for interfaces with multi-port interfaces such as dual port memories.



Standard connection for timing diagrams.

Wishbone – the interface

Optional supplementary reading

MASTER Interfaces

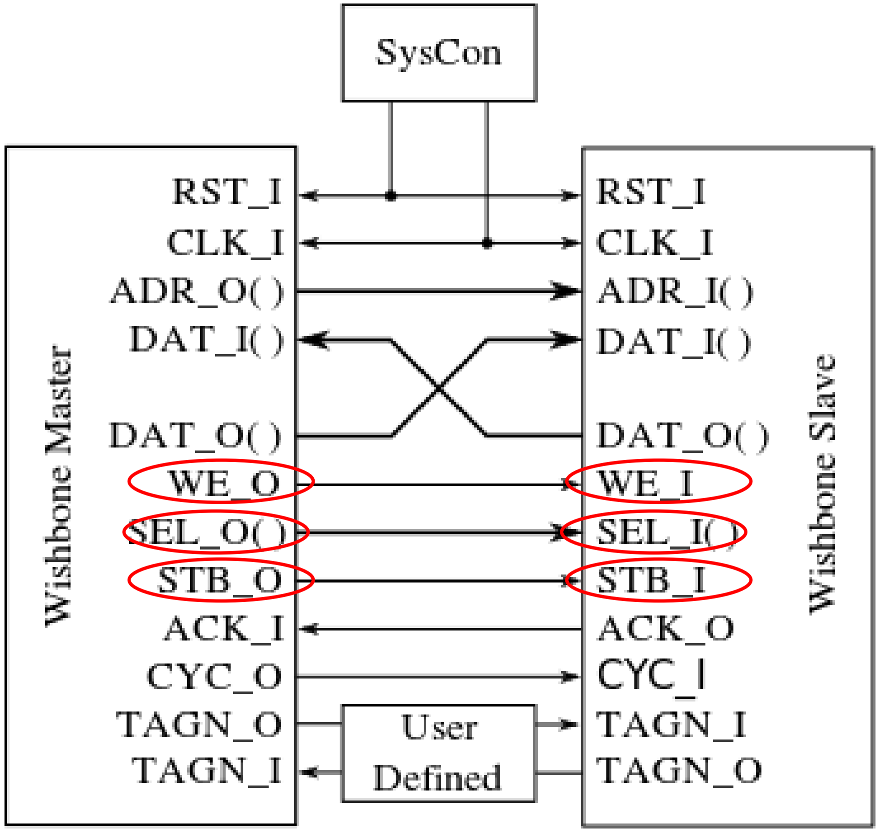
WE_O: Write enable output indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is *negated during READ cycles*, and is *asserted during WRITE cycles*.

SEL_O: The select output array indicates:

- when valid data is expected on the DAT_I bus during READ cycles, and
- when it is placed on the DAT_O signal array during WRITE cycles.

STB_O: The strobe output indicates a valid data transfer cycle. Qualifies various other signals on the interface such as SEL_O.

The SLAVE asserts either ACK_I, ERR_I or RTY_I in response to every assertion of STB_O by master.



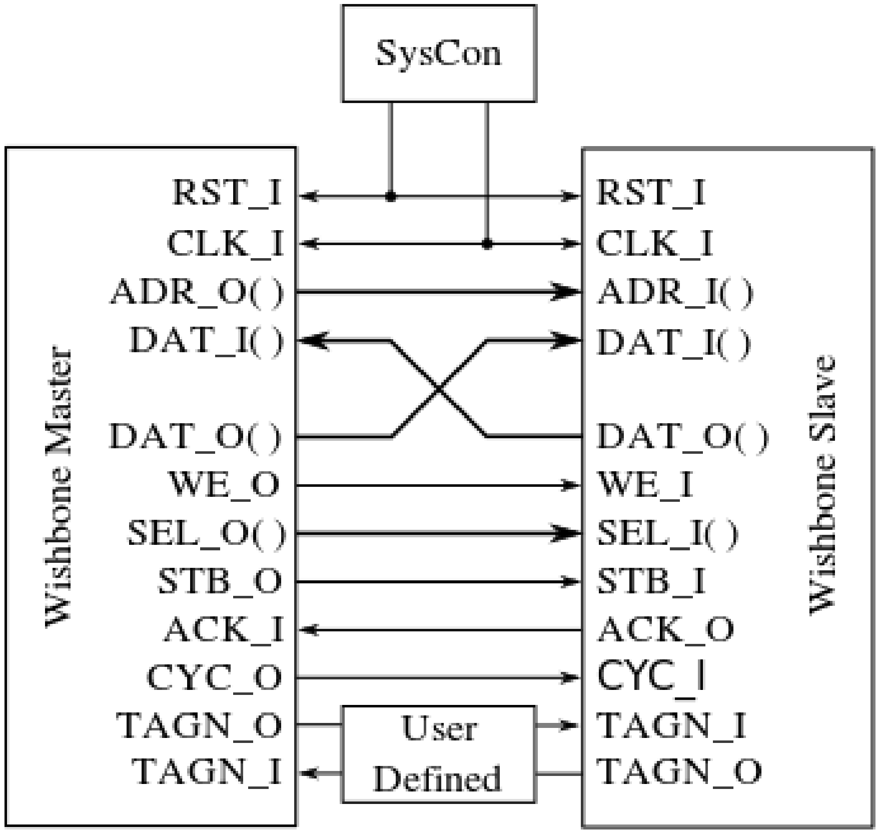
Standard connection for timing diagrams.

Wishbone – the interface

Optional supplementary reading

MASTER Interfaces – special error handing signals

STALL_I: pipeline stall input indicates current slave is not able to accept the transfer in the transaction queue (used in pipelined mode).
ERR_I: indicates an abnormal cycle termination.
LOCK_O: The lock output when asserted indicates the current bus cycle is uninterruptible. Lock is asserted to request complete ownership of the bus.
RTY_I: The retry input indicates the interface is not ready to accept or send data, and that the cycle should be retried. (when and how the cycle is retried is defined by the IP core supplier).



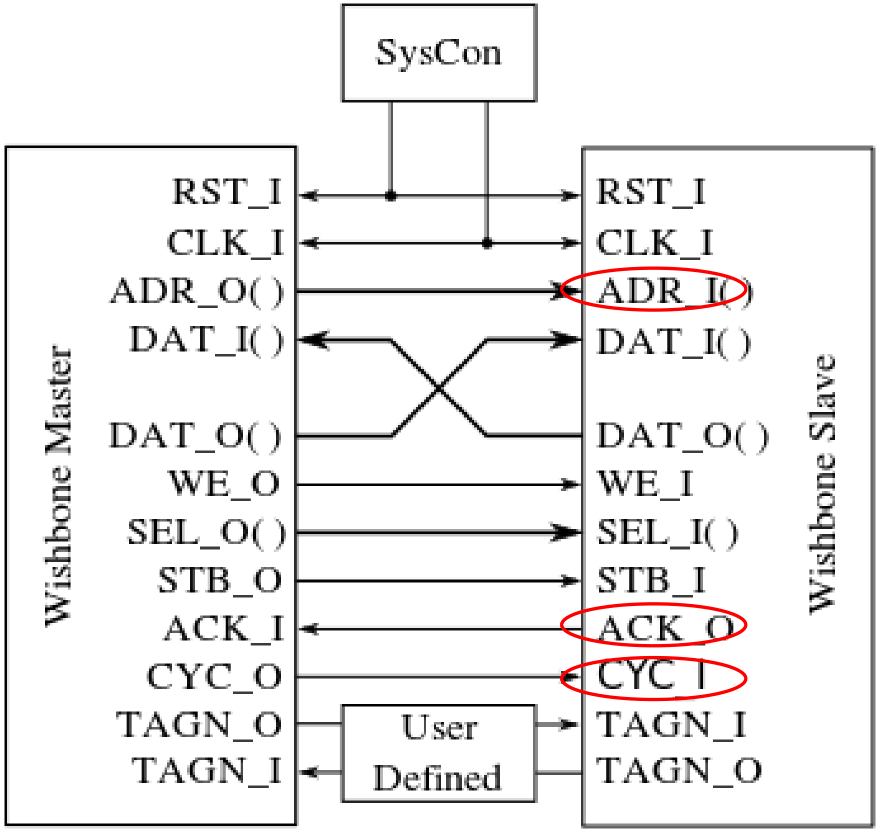
Standard connection for timing diagrams.

Wishbone – the interface

Optional supplementary reading

SLAVE Interfaces

ACK_O: Acknowledge output, when asserted, indicates the termination of a normal bus cycle.
ADR_I: Address input array passes a binary address. Bus size is specific to the address width of the core. *NB: Lower array boundary is determined by the data port size*. E.g. 32-bit data port has ADR_O [31:2].
CYC_I: When asserted, indicates that a valid bus cycle is in progress



Standard connection for timing diagrams.

Wishbone reset

Optional supplementary reading

- Reset can be asserted for any length of time
- All Wishbone interfaces must initialize themselves at the **rising CLK_I** edge following the assertion of RST_I.
- They must stay in the initialized state until the **rising CLK_I edge** that follows the negation of RST_I.
- RST_I must be asserted for at least one complete clock cycle.
- Note: **CLK_I** line schedules the process as to by when signals are asserted and sensed (e.g. the order of asserting STB_O and CYC_I is arbitrary, but must be done before a positive clock edge)
  - There is method in this clock-triggered approach because it simplifies the interfacing logic and the statemachines for the master and slave by having them only trigged by posedge CLK_I and not anything else (FPGAs typically are designed around making the clock accessible and synched, also save space for other modules).

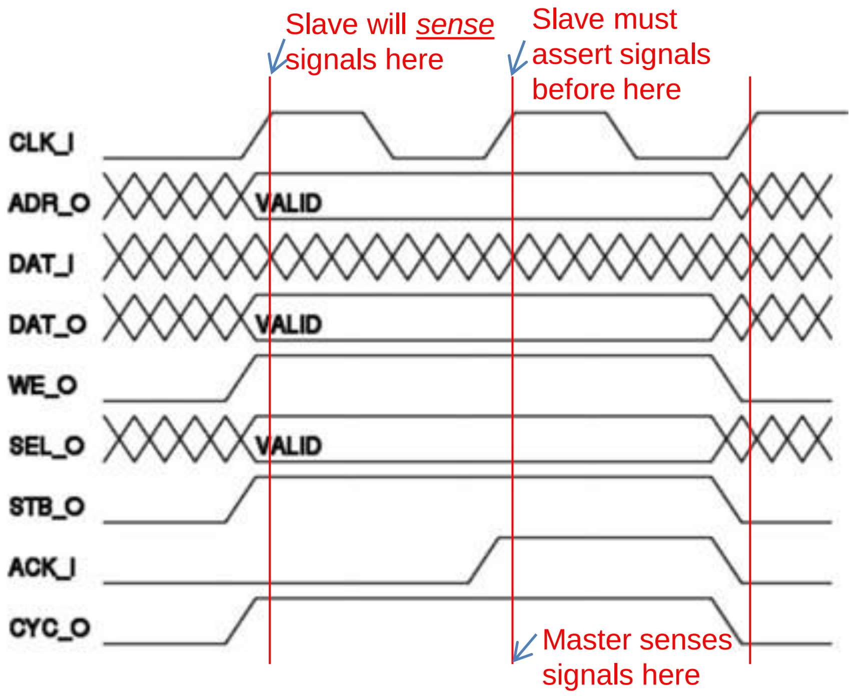
Wishbone Standard Read/Write C

Optional supplementary reading

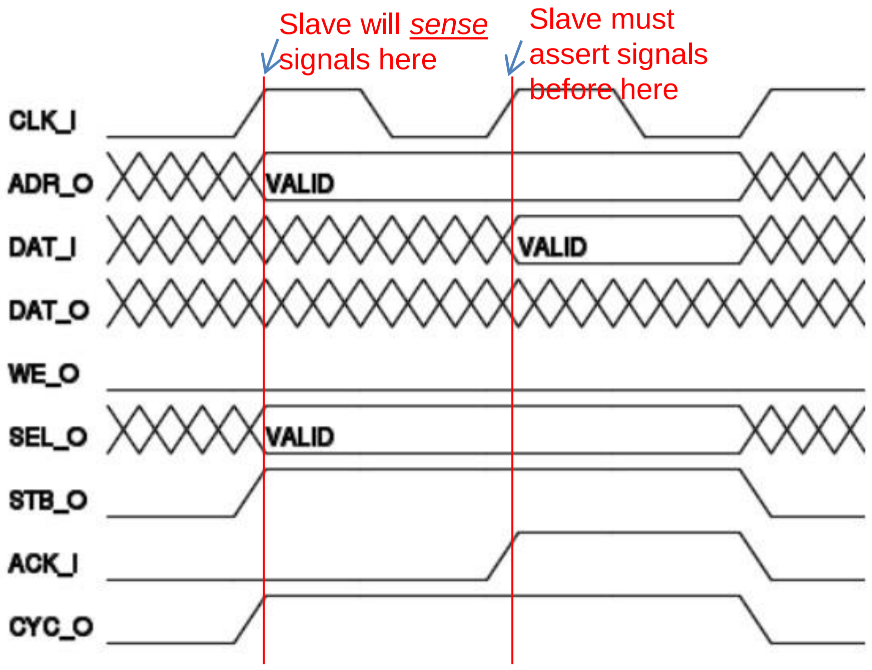
MASTER initiate a transfer cycle by asserting CYC_O
MASTER MUST assert CYC_O for the duration of READ/WRITE cycle

MASTER asserts STB_O when ready to transfer data
    STB_O remains asserted until SLAVE asserts one of the cycle terminating signals ACK_I], ERR_I or RTY_I

At every rising edge of CLK_I the slave/master samples the signals and must respond by asserting relevant signals before the next rising clock edge.


Slave will *sense* signals here
Slave must assert signals before here
CLK_I
ADR_O
DAT_I
DAT_O
WE_O
SEL_O
STB_O
ACK_I
CYC_O
VALID
VALID
VALID
Master senses signals here

Standard SINGLE WRITE cycle


Slave will *sense* signals here
Slave must assert signals before here
CLK_I
ADR_O
DAT_I
DAT_O
WE_O
SEL_O
STB_O
ACK_I
CYC_O
VALID
VALID
VALID

Standard SINGLE READ cycle

Wishbone examples / templates

Optional supplementary reading

- Recommended
  - Simple starting point provided by OpenCores: https://opencores.org/forum,Cores,0,608

Optional supplementary reading

Altera/Intel Avalon Bus

- Altera Avalon interface bus used as Nios embedded processor peripheral bus
- Designed (originally) to accommodate peripheral development for the System-On-a-Programmable-Chip (SOPC) environment.
- Qsys now replaces SOPC tool
- The generated switch fabric logic includes several chip select signals for
  - data-path multiplexing, address decoding, wait-state generation, interrupt-priority assignment, dynamic-bus sizing, multi-muster arbitration logic and advanced switch fabric transfer.
- Mainly intended for implemented on Altera devices using Qsys / SOPC Builder
- Avalon Bus is generated automatically, when a new Nios core with peripherals is created in Sys/SOPC-builder
- **Is an open standard** (can develop own Avalon modules)

End of Lecture

Any Question??

Disclaimers and copyright/licensing details