Altera's Avalon Communication Fabric

Stephen A. Edwards

Columbia University

Spring 2013

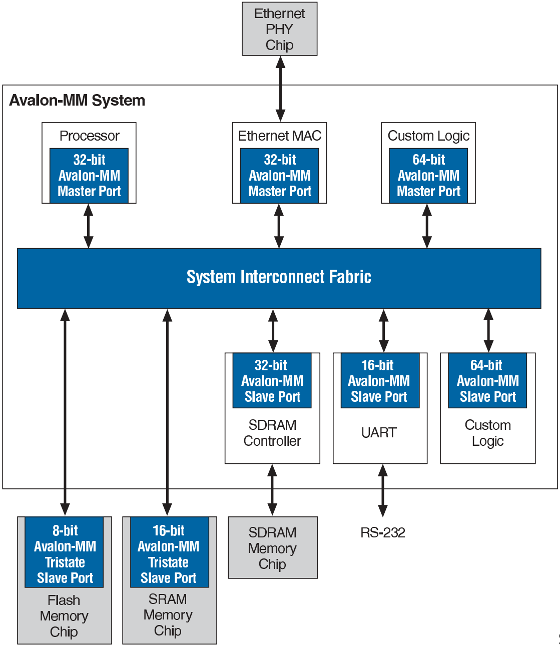
Altera’s Avalon Bus

Something like “PCI on a chip”

Described in Altera’s *Avalon Memory-Mapped Interface Specification* document.

Protocol defined between peripherals and the “bus” (actually a fairly complicated circuit).

Intended System Architecture



Source: Altera

Masters and Slaves

Most bus protocols draw a distinction between

Masters: Can initiate a transaction, specify an address, etc. E.g., the Nios II processor

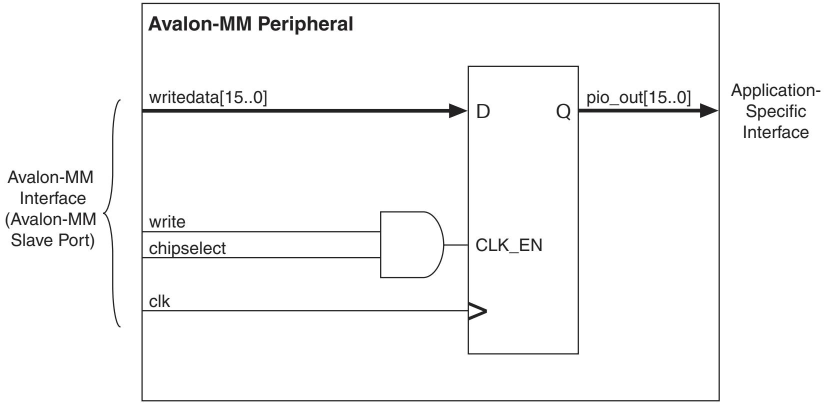
Slaves: Respond to requests from masters, can generate return data. E.g., a video controller

Most peripherals are slaves.

Masters speak a more complex protocol

Bus arbiter decides which master gains control

The Simplest Slave Peripheral



Basically, “latch when I’m selected and written to.”

Naming Conventions

Used by the SOPC Builder's New Component Wizard to match up VHDL entity ports with Avalon bus signals.

type_interface_signal

type is is typically `avs` for Avalon-MM Slave

interface is the user-selected name of the interface, e.g., `s1`.

signal is `chipselect`, `address`, etc.

Thus, `avs_s1_chipselect` is the chip select signal for a slave port called "`s1`."

Slave Signals

For a 16-bit connection that spans 32 halfwords,

Slave			Avalon
	←	clk	
	←	reset	
	←	chipselect	
	⇐	address[4:0]	
	←	read	
	←	write	
	⇐	byteenable[1:0]	
	⇐	writedata[15:0]	
	readdata[15:0]	→	
	irq	→	

Avalon Slave Signals

clk	Master clock
reset	Reset signal to peripheral
chipselect	Asserted when bus accesses peripheral
address[..]	Word address (data-width specific)
read	Asserted during peripheral→bus transfer
write	Asserted during bus→peripheral transfer
writedata[..]	Data from bus to peripheral
byteenable[..]	Indicates active bytes in a transfer
readdata[..]	Data from peripheral to bus
irq	peripheral→processor interrupt request

All are optional, as are many others for, e.g., flow-control and burst transfers.

Bytes, Bits, and Words

The Nios II and Avalon bus are little-endian:

31 is the most significant bit, 0 is the least

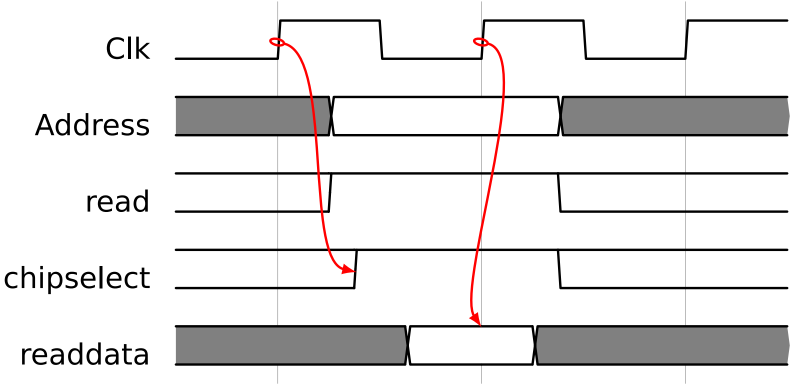
Bytes and halfwords are right-justified:

	msb			lsb
Byte	3	2	1	0
Bit	31 24	23 16	15 8	7 0
Word	31			0
Halfword			15	0
Byte				7 0

In VHDL

```
entity avalon_slave is
  port (
    avs_s1_clk        : in  std_logic;
    avs_s1_reset_n    : in  std_logic;
    avs_s1_read       : in  std_logic;
    avs_s1_write      : in  std_logic;
    avs_s1_chipselect : in  std_logic;
    avs_s1_address    : in  std_logic_vector(4 downto 0);
    avs_s1_readdata   : out std_logic_vector(15 downto 0);
    avs_s1_writedata  : in  std_logic_vector(15 downto 0);
  );
end avalon_slave;
```

Basic Async. Slave Read Transfer

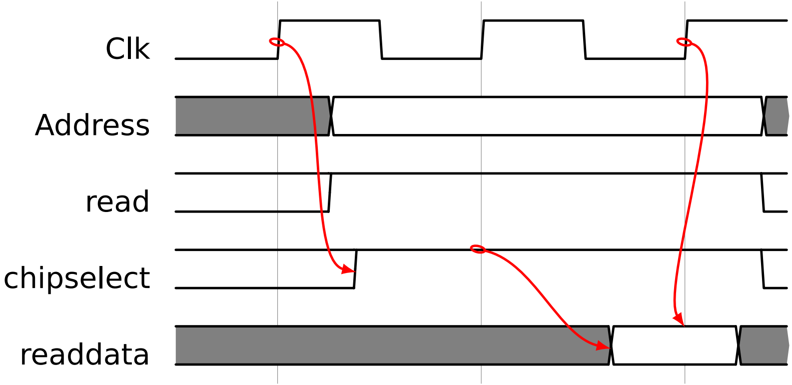


Bus cycle starts on rising clock edge.

Data latched at next rising edge.

Such a peripheral must be purely combinational.

Slave Read Transfer w/ 1 Wait State

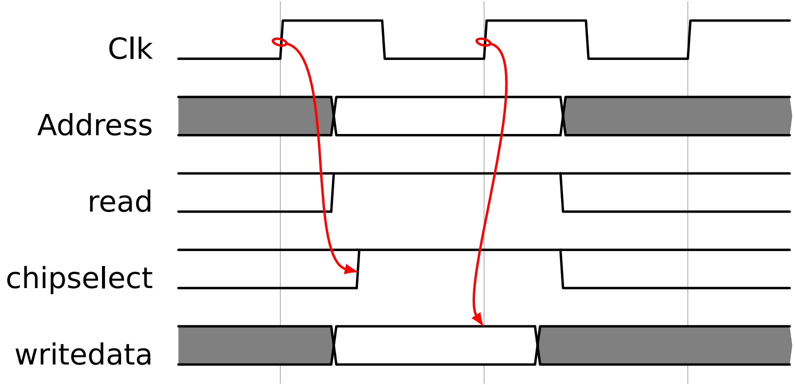


Bus cycle starts on rising clock edge.

Data latched two cycles later.

Approach used for synchronous peripherals.

Basic Async. Slave Write Transfer

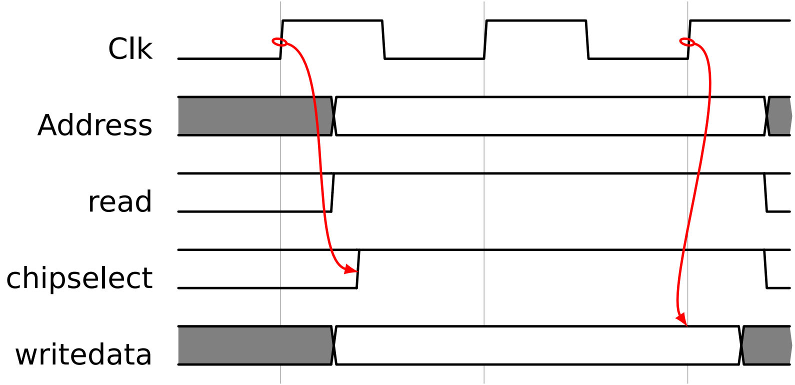


Bus cycle starts on rising clock edge.

Data available by next rising edge.

Peripheral may be synchronous, but must be fast.

Basic Async. Slave Write w/ 1 Wait State



Bus cycle starts on rising clock edge.

Peripheral latches data two cycles later.

For slower peripherals.

The LED Flasher Peripheral

32 16-bit word interface

First 16 halfwords are data to be displayed on the LEDS.

Halfwords 16–31 all write to a “linger” register that controls cycling rate.

Red LEDs cycle through displaying memory contents.

Entity and Architecture Declaration

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity de2_led_flasher is
  port (
    clk        : in  std_logic;
    reset_n    : in  std_logic;
    read       : in  std_logic;
    write      : in  std_logic;
    chipselect : in  std_logic;
    address    : in  unsigned(4 downto 0);
    readdata   : out unsigned(15 downto 0);
    writedata  : in  unsigned(15 downto 0);
    leds       : out unsigned(15 downto 0)
    );
end de2_led_flasher;

architecture rtl of de2_led_flasher is
  type ram_type is array(15 downto 0) of unsigned(15 downto 0);
  signal RAM : ram_type;
  signal ram_address, display_address : unsigned(3 downto 0);
  signal counter_delay : unsigned(15 downto 0);
  signal counter : unsigned(31 downto 0);
begin
  ram_address <= address(3 downto 0);
```

Architecture (2)

```
  process (clk) begin
    if rising_edge(clk) then
      if reset_n = '0' then
        readdata <= (others=>'0'); display_address <= (others=>'0');
        counter <= (others => '0'); counter_delay <= (others=>'1');
      else
        if chipselect = '1' then
          if address(4) = '0' then -- read or write RAM
            if read = '1' then
              readdata <= RAM(to_integer(ram_address));
            elsif write = '1' then
              RAM(to_integer(ram_address)) <= writedata;
            end if;
          else
            if write = '1' then -- Change delay
              counter_delay <= writedata;
          end if; end if;
         else -- No access to us: update display
          leds <= RAM(to_integer(display_address));
          if counter = x"00000000" then
            counter <= counter_delay & x"0000";
            display_address <= display_address + 1;
          else
            counter <= counter - 1;
end if; end if; end if; end if; end process; end rtl;
```