Altera's Avalon Communication Fabric

Prof. Stephen A. Edwards

sedwards@cs.columbia.edu

Columbia University

Spring 2007

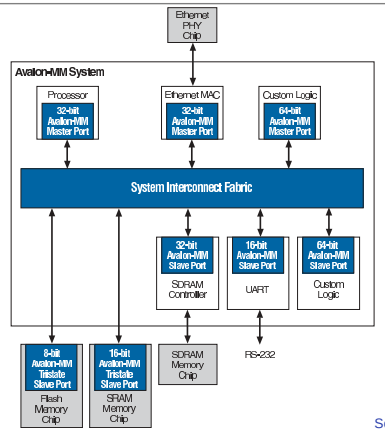
Altera's Avalon Bus

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

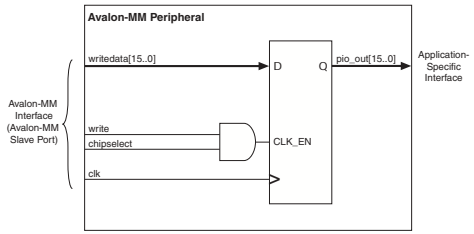
Slaves: Respond to requests from masters, can generate return data. E.g., a video controller

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,

	←	clk	
	⇐	reset	
	←	chipselect	
	←	address[4:0]	
Slave	←	read	**Avalon**
	←	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:

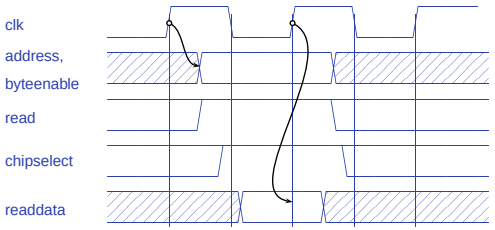
	msb							lsb
Byte	3		2		1		0	
Bit	31	24	23	16	15	8	7	0

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

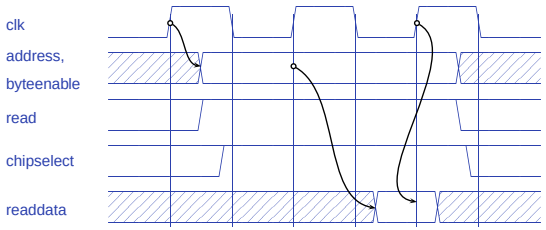


Bus cycle starts on rising clock edge.

Data latched at next rising edge.

Such a peripheral must be purely combinational.

Slave Read Transfer w/ 1 wait state

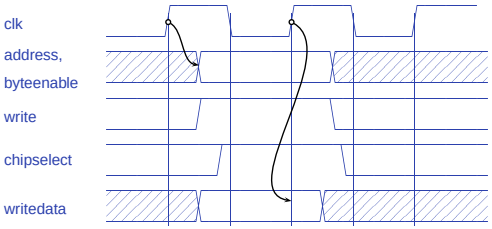


Bus cycle starts on rising clock edge.

Data latched two cycles later.

Approach used for synchronous peripherals.

Basic Async. Slave Write Transfer

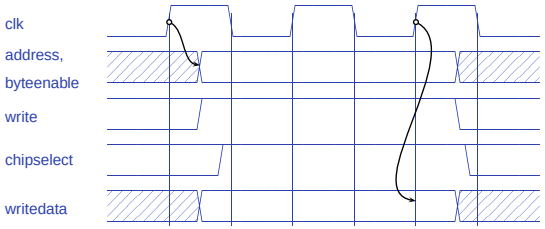


Bus cycle starts on rising clock edge.

Data available by next rising edge.

Peripheral may be synchronous, but must be fast.

Slave Write Transfer 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 Declaration

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity led_flasher 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);

    leds : out std_logic_vector(15 downto 0)
  );

end led_flasher;
```

Architecture (1)

```
architecture rtl of led_flasher is

signal clk, reset_n : std_logic;

type ram_type is array(15 downto 0) of std_logic_vector(15 downto 0);
signal RAM : ram_type;

signal ram_address, display_address : std_logic_vector(3 downto 0);

signal counter_delay : std_logic_vector(15 downto 0);
signal counter : std_logic_vector(31 downto 0);

begin

  clk <= avs_s1_clk;

  reset_n <= avs_s1_reset_n;

  ram_address <= avs_s1_address(3 downto 0); -- "Memory" address
```

Architecture (2)

```
process (clk)
begin
  if clk'event and clk = '1' then

    if reset_n = '0' then
      avs_s1_readdata <= (others => '0');
      display_address <= (others => '0');
      counter <= (others => '0');
      counter_delay <= (others => '1');
    else

      if avs_s1_chipselect = '1' then

        if avs_s1_address(4) = '0' then  -- "Memory" region

          if avs_s1_read = '1' then
            avs_s1_readdata <= RAM(conv_integer(ram_address));
          elsif avs_s1_write = '1' then
            RAM(conv_integer(ram_address)) <= avs_s1_writedata;
          end if;

        else                             -- "Linger" register

          if avs_s1_write = '1' then
            counter_delay <= avs_s1_writedata;
          end if;
```

Architecture (3)

```
        else     -- No chip select: display memory

          leds <= RAM(conv_integer(display_address));

          if counter = x"00000000" then
            counter <= counter_delay & x"0000";     -- Reset counter
            display_address <= display_address + 1; -- Next address
          else
            counter <= counter - 1;
          end if;

        end if;
      end if;
    end if;
  end process;

end rtl;
```