Altera’s Avalon Communication Fabric

Stephen A. Edwards

Columbia University

Spring 2012
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).
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,

<table>
<thead>
<tr>
<th>Slave</th>
<th>Avalon</th>
</tr>
</thead>
<tbody>
<tr>
<td>← clk</td>
<td>→</td>
</tr>
<tr>
<td>← reset</td>
<td></td>
</tr>
<tr>
<td>← chipselect</td>
<td></td>
</tr>
<tr>
<td>← address[4:0]</td>
<td></td>
</tr>
<tr>
<td>← read</td>
<td></td>
</tr>
<tr>
<td>← write</td>
<td></td>
</tr>
<tr>
<td>← byteenable[1:0]</td>
<td></td>
</tr>
<tr>
<td>← writedata[15:0]</td>
<td></td>
</tr>
<tr>
<td>← readdata[15:0]</td>
<td></td>
</tr>
<tr>
<td>← irq</td>
<td>→</td>
</tr>
</tbody>
</table>
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.
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:

<table>
<thead>
<tr>
<th>Byte</th>
<th>msb</th>
<th></th>
<th></th>
<th>lsb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>31 24</td>
<td>23 16</td>
<td>15 8</td>
<td>7 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th></th>
<th></th>
<th>lsb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halfword</td>
<td>15</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Byte</td>
<td></td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
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.
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.
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.
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);
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
        if write = '1' then -- 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 if;
end过程; end rtl;