Altera's Avalon Communication Fabric

Prof. Stephen A. Edwards
sedwards@cs.columbia.edu
Columbia University
Spring 2008

Altera's Avalon Bus

Something like “PCI on a chip”
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

Avalon-MM Interface (Avalon-MM Slave Port)
Application-Specific Interface
\[
\begin{array}{c}
\text{writedata}[15..0] \\
\text{chipselect} \\
\text{CLK_EN} \\
\end{array}
\]

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, the Slave Signals:
\[
\begin{array}{c}
\text{clk} \\
\text{reset} \\
\text{chipselect} \\
\text{address}[4:0] \\
\text{read} \\
\text{write} \\
\text{byteenable}[1:0] \\
\text{readdata}[15:0] \\
\text{writedata}[15:0] \\
\text{irq} \\
\end{array}
\]

Avalon Slave Signals

\[
\begin{array}{c}
\text{clk} \quad \text{Master clock} \\
\text{reset} \quad \text{Reset signal to peripheral} \\
\text{chipselect} \quad \text{Asserted when bus accesses peripheral} \\
\text{address}[..] \quad \text{Word address (data-width specific)} \\
\text{read} \quad \text{Asserted during peripheral—bus transfer} \\
\text{write} \quad \text{Asserted during bus—peripheral transfer} \\
\text{writedata}[..] \quad \text{Data from bus to peripheral} \\
\text{byteenable}[..] \quad \text{Indicates active bytes in a transfer} \\
\text{readdata}[..] \quad \text{Data from peripheral to bus} \\
\text{irq} \quad \text{Peripheral—processor interrupt request} \\
\end{array}
\]

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:
\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{Byte} & 3 & 2 & 1 & 0 \\
\hline
\text{Bit} & 31 & 24 & 16 & 8 & 7 & 0 \\
\text{Word} & 31 & 15 & 0 \\
\text{Halfword} & 15 & 0 \\
\text{Byte} & 7 & 0 \\
\end{array}
\]
In VHDL

entity avalon_slave is
port ( clk, address, byteenable, read, chipselect, readdata );
end avalon_slave;

Bus cycle starts on rising clock edge.
Data latched at next rising edge.
Such a peripheral must be purely combinational.

Basic Async. Slave Write Transfer

clk
address, byteenable
write
chipselect
writedata

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

clk
address, byteenable
write
chipselect
writedata

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.

Architecture (1)

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

entity do2_led_flasher is
port ( clk, reset_n, read, chipselect, address, byteenable, readdata, writedata, leds );
end do2_led_flasher;

architecture rtl of do2_led_flasher is
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
readdata <= (others => '0');
display_address <= (others => '0');
else
if write = '1' then
readdata <= (others => '0');
display_address <= (others => '0');
end if;
end if;
end if;
end if;
end if;
end do2_led_flasher;

`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 process;
end rtl;