Memoria cache
Ierarhia memoriei: ierarhie expusă, ierarhie ascunsă
Localitatea temporală si spațiala
Clasificare rata eşec (miss rate)
Amplasarea blocurilor în memoria cache
Mapare directă
Mapare total asociativă
Maparea asociativă pe seturi
înlocuirea blocurilor din memoria cache
Strategii scriere: Write trough, Write back

Memoria cache - coerența memoriei cache
protocol MSI
protocol MESI
protocol MOESI
protocol MESIF

Topologii de rețea & algoritmi de rutare
Crosbar switch
Hipercub
Multistage Interconnection Networks – Rețele cu interconexiuni multinivel
Store-and-Forward (SF)
Wormhole (WH)
Virtual-Cut-Through (VCT)

Ex. sub. referat:

- Protocol AHB (Advanced High performance)
- ASB (Advanced System Bus)
- AXI (Advanced Extensible Interface)
- ACE (AXI Coherency Extensions)
- CHI (Coherent Hub Interface)

- Wishbone Bus
- AVALON Bus
- CoreConnect Bus

Arhitectura procesoare:

- Proc. Intel Core i3/i5/i7/i9
- Proc. Intel Xeon/Xeon Phi
- AMD Ryzen
- AMD Ryzen Threadripper
- Proc.AMD EPYC

- Arhitectura ARM big.LITTLE

- microarhitectura NVIDIA Pascal

Proiecte

Implementati un mecanism pentru mentinerea corentei memoriilor cache

1. utilizand protocol MSI
2. utilizand protocol MESI
3. utilizand protocol MOESI
4. bazat pe un singur director
5. bazat pe un director distribuit
6. utilizand protocol MESIF

7. Implementați un modul timer sub forma unui IP care se conecteaza utilizand AXI la un proc. MicroBlaze.
8. Implementati un modul PWM sub forma unui IP care se conecteaza utilizand AXI la un proc. MicroBlaze.
9. Implementați un generator de semnal periodic(4 segmente cu durata si panta configurabilă) sub forma unui IP care se conecteaza utilizand AXI la un proc. MicroBlaze.

Creare AXI IP >> link
Hello World MicroBlaze >>link , link2

Creare fisier *.VHD

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity multiplier is
    Port ( clk : in STD_LOGIC;
           a : in STD_LOGIC_VECTOR (15 downto 0);
           b : in STD_LOGIC_VECTOR (15 downto 0);
           prod : out STD_LOGIC_VECTOR (31 downto 0));
end multiplier;

architecture Behavioral of multiplier is
begin

    process(clk)
    begin
        if rising_edge(clk) then
           prod <= a*b;
        end if;
    end process;

end Behavioral;
```

```
architecture arch_imp of IP_AXI_PROD_HG_v1_0_S00_AXI is

       -- AXI4LITE signals
       signal axi_awaddr     : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
       signal axi_awready    : std_logic;
       signal axi_wready     : std_logic;
```

```
architecture arch_imp of IP_AXI_PROD_HG_v1_0_S00_AXI is
    component multiplier is
        Port ( clk : in STD_LOGIC;
               a : in STD_LOGIC_VECTOR (15 downto 0);
               b : in STD_LOGIC_VECTOR (15 downto 0);
               prod : out STD_LOGIC_VECTOR (1 downto 0));
    end component;

    SIGNAL myMultiplier_OUT:std_logic_vector(31 downto 0);
       -- AXI4LITE signals
       signal axi_awaddr     : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
       signal axi_awready    : std_logic;
```

```
        signal axi_wready     : std_logic;
```

Instantiere multiplier in arhitectura `arch_imp`

```
...
        end process;


        -- Add user logic here

        -- User logic ends

end arch_imp;
```

```
        end process;


        -- Add user logic here
 eti0: multiplier PORT MAP(S_AXI_ACLK,slv_reg0(31 downto 16),slv_reg0(15 downto
0),myMultiplier_OUT );
        -- User logic ends

end arch_imp;
```

```
-- Implement memory mapped register select and read logic generation
	-- Slave register read enable is asserted when valid address is available
	-- and the slave is ready to accept the read address.
	slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;

	process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN,
slv_reg_rden)
	variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
	begin
	    -- Address decoding for reading registers
	    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
	    case loc_addr is
	      when b"00" =>
	        reg_data_out <= slv_reg0;
	      when b"01" =>
	        reg_data_out <= slv_reg1;
	      when b"10" =>
	        reg_data_out <= slv_reg2;
	      when b"11" =>
	        reg_data_out <= slv_reg3;
	      when others =>
	        reg_data_out  <= (others => '0');
	    end case;
	end process;
```

```
-- Implement memory mapped register select and read logic generation
	-- Slave register read enable is asserted when valid address is available
	-- and the slave is ready to accept the read address.
	slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;

	process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN,
slv_reg_rden)
	variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
	begin
	    -- Address decoding for reading registers
	    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
	    case loc_addr is
	      when b"00" =>
	        reg_data_out <= slv_reg0;
	      when b"01" =>
	        reg_data_out <= myMultiplier_OUT;
	      when b"10" =>
	        reg_data_out <= slv_reg2;
	      when b"11" =>
	        reg_data_out <= slv_reg3;
	      when others =>
```

```
            reg_data_out  <= (others => '0');
        end case;
    end process;
```