IP cores
IP cores

• Soft IP cores
• Hard IP cores
  – ROM, RAM, FIFO
  – RISC CPU
  – DSP - Multiplier
  – SerDes
  – Flash memory (boot, user)
  – PCI, PCIe
  – JTAG
Soft IP cores

- Nobody wants to invent the bicycle again
- Modules, which are widely used, are available for different technologies, not always for free
- One such module can have many parameters and a special software (Wizard/Generator) to set them properly
- Each module should come with a reference design and a testbench in VHDL/Verilog
- The source code is typically not available, or is just a question of price
- In general the usage of IP cores saves time and money, but for small companies and small productions can be too expensive
- OpenCores is an alternative
Typical soft IP cores

- PCI master-target, 32/64 bit, PCIe
- Ethernet, UART
- μP, μC (incl. old ones), RISC CPUs
- Interface to DRAM, SSRAM
- DSP: CORDIC, DDS, FFT, Filters
- VME, USB, CAN, I2C, SPI, SD card
- encryption / decryption
RAM in FPGA
distributed – embedded

• Xilinx and Lattice have the option to use the LUTs (but not all in the chip) as 16x1 or 32x1 RAM (the so called distributed RAM)
  – small RAM blocks can be created by combining many such LUTs and MUXes (implemented in other LUTs)
  – not good for large RAM blocks – bad performance, routing and logic resources
  – can be read asynchronously!

• The alternative is to use embedded RAM blocks
  – typically asynchronous reading is not supported!
  – the blocks are of fixed size
RAM in FPGA – embedded blocks

- The width can be different at the input and at the output
- **Dual port** mode
- **FIFO** mode
- up to 250 MHz
RAM in FPGA – how to use it in VHDL

• In order to use RAM in FPGAs either
  – generate it using a specific tool from the FPGA vendor (Core Generator, MegaWizard)
  – instantiate directly the corresponding primitives (but if you need to change frequently the size of the memory this is not very practical)
  – describe it in VHDL following the expected style, otherwise it will be not implemented as block RAM
    • check the report after the compilation!
    • this is very nice, as the design remains easily portable to other technologies!
Single port RAM in FPGA

... USE IEEE.std_logic_unsigned.ALL;
entity ssram is
generic (Na : Positive := 8; Nd : Positive := 16; async_rd : Boolean := false);
port(clk : in std_logic;
we : in std_logic;
addr : in std_logic_vector(Na-1 downto 0);
din : in std_logic_vector(Nd-1 downto 0);
dout : out std_logic_vector(Nd-1 downto 0)) ;
end ssram;
architecture a of ssram is
type t_mem_data is
array(0 to 2**addr'length - 1)
of std_logic_vector(dout'range);
signal mem_data : t_mem_data;
signal raddri, addri : integer range 0 to 2**addr'length - 1;
begin
addri <= conv_integer(addr);
ram: process(clk)
begin
if clk'event and clk='1' then
raddri <= addri;
if we = '1' then
mem_data(addri) <= din;
end if;
end if;
end process;
ar: if async_rd generate dout <= mem_data( addri); end generate;
sr: if not async_rd generate dout <= mem_data(raddri); end generate;
end;

Note that asynchronous reading is not available in the RAM blocks
Two independent address ports (write/read) with common clock
Dual port dual clock RAM in FPGA

entity dc_dp_sram is
generic (Na : Positive := 8;
         Nd : positive := 16);
port(wclk    : in std_logic;
    rclk    : in std_logic;
begin
    waddri <= conv_integer(waddr);
    ram: process(wclk)
    begin
        if wclk'event and wclk='1' then
            if we = '1' then
                mem_data(waddri) <= din;
            end if;
        end if;
    end process;
    process(rclk)
    begin
        if rclk'event and rclk='1' then
            raddri <= conv_integer(raddr);
        end if;
    end process;
    dout <= mem_data(raddri);
end;
Waveforms of RAM in FPGA

Common signals

CLK
WE
WADDR XX | A0 | A1 | A2 | A3 | A4
DIN XX | D0 | D1 | D2 | D3 | D4

Single Port RAM

SP_ADDR XX | A0 | A1 | A2 | A3 | A4
DOUT_AS XX | D0 | XX | D1 | XX | D2 | XX | D3 | XX | D4 | XX

Dual Port RAM read

CLK
RADDR_DP XX | A0 | A1 | A2 | A3 | A4
DOUT_DP XX | D0 | D1 | D2 | D3 | XX | D0 | D1 | D2 | D3 | D4

Dual Clock RAM read

RCLK
RADDR_DC XX | A0 | A1 | A2 | A3 | A4
DOUT_DC XX | D0 | D1 | D2 | D3 | D4
FIFO

- **First In First Out**
- **Used as**
  - derandomizing buffer between devices with different speed
  - data resynchronization between different clock domains
- **Single clock and dual clock**
- **Consists of a dual port memory, write pointer, read pointer and control logic**
Single clock FIFO(1)

If the reading is synchronous, the output data will have 1 clock delay
process(clk)
begin
  if clk'event and clk='1' then
    if sclr = '1' then
      wpointer <= (others => '0');
      rpointer <= (others => '0');
      nwords <= (others => '0');
      fifo_full <= '0';
      fifo_empty <= '1';
    else
      case fifo_cmd is
        when "10" => -- write
          if fifo_full='0' then
            wpointer <= wpointer+1;
            if nwords = almost_full then
              fifo_full <= '1'; end if;
            nwords <= nwords +1;
            fifo_empty <= '0';
          end if;
        when "01" => -- read
          if fifo_empty='0' then
            rpointer <= rpointer+1;
            if nwords = almost_empty then
              fifo_empty <= '1'; end if;
            nwords <= nwords -1;
            fifo_full <= '0';
          end if;
      end case fifo_cmd;
    end if;
  end if;
end process(clk);

ENTITY sc_fifo is
generic (Na : Integer := 4; Nd : Integer := 8);
PORT ( din : in std_logic_vector(Nd-1 downto 0);
        wrreq : in std_logic;
        rdreq : in std_logic;
        clk : in std_logic;
        sclr : in std_logic;
        dout : out std_logic_vector(Nd-1 downto 0);
        full : out std_logic;
        empty : out std_logic);
end sc_fifo;

Single clock FIFO(3)

Implementing some special logic to treat these cases in the FIFO is not very reasonable (either some input data will be lost or some wrong data will be read), as it will slow down its normal operation!

Writing to a full FIFO and reading from an empty FIFO should NEVER happen!
Single clock FIFO(4)

Simulation of the `sc_fifo`. The `valid` signal in the testbench is just the registered `rdreq`. Writing to a full and reading from an empty are NOT tested!
Data resynchronization with dual clock FIFO

For more details see the VHDL code in resync_sdr.vhd
Data resynchronization – simulation
FPGA – Multiplier blocks

• Typically two possible configurations:
  \[ 2 \times (9 \times 9 \rightarrow 18) \] or \[ 18 \times 18 \rightarrow 36 \]

• Optional registers at the input/output

• All 4 combinations of the integer types (signed/unsigned)

• Basic element to implement digital filters and DSP

**Size and performance without/with using multiplier blocks:**

<table>
<thead>
<tr>
<th>Size</th>
<th>f_{MAX} [MHz]</th>
<th>LE</th>
<th>f_{MAX} [MHz]</th>
<th>MultBlocks (9x9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 x 9</td>
<td>144</td>
<td>122</td>
<td>260</td>
<td>1</td>
</tr>
<tr>
<td>18 x 18</td>
<td>90</td>
<td>426</td>
<td>260</td>
<td>2</td>
</tr>
</tbody>
</table>

© V. Angelov

VHDL-FPGA@PI 2013
FPGA – additional hard coded modules

Multi-Gigabit serializer / deserializer
FPGA – other modules

CPU cores
PowerPC in
Xilinx Virtex 4 / 5

Internal FLASH memory, for FPGA booting and to initialize RAM blocks, as well as for data which should be not lost after power down – Actel Fusion, ProASIC and IGLOO, LatticeXP2, Xilinx Spartan 3AN
JTAG

Control of all I/O cells

For the inputs:
- read the signal coming to the pin
- set the signal to the core

For the outputs:
- Read the signal from the core
- set the signal to the pin

Programming of CPLD/FPGA, test of ASIC

Connectivity test on the PCB