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

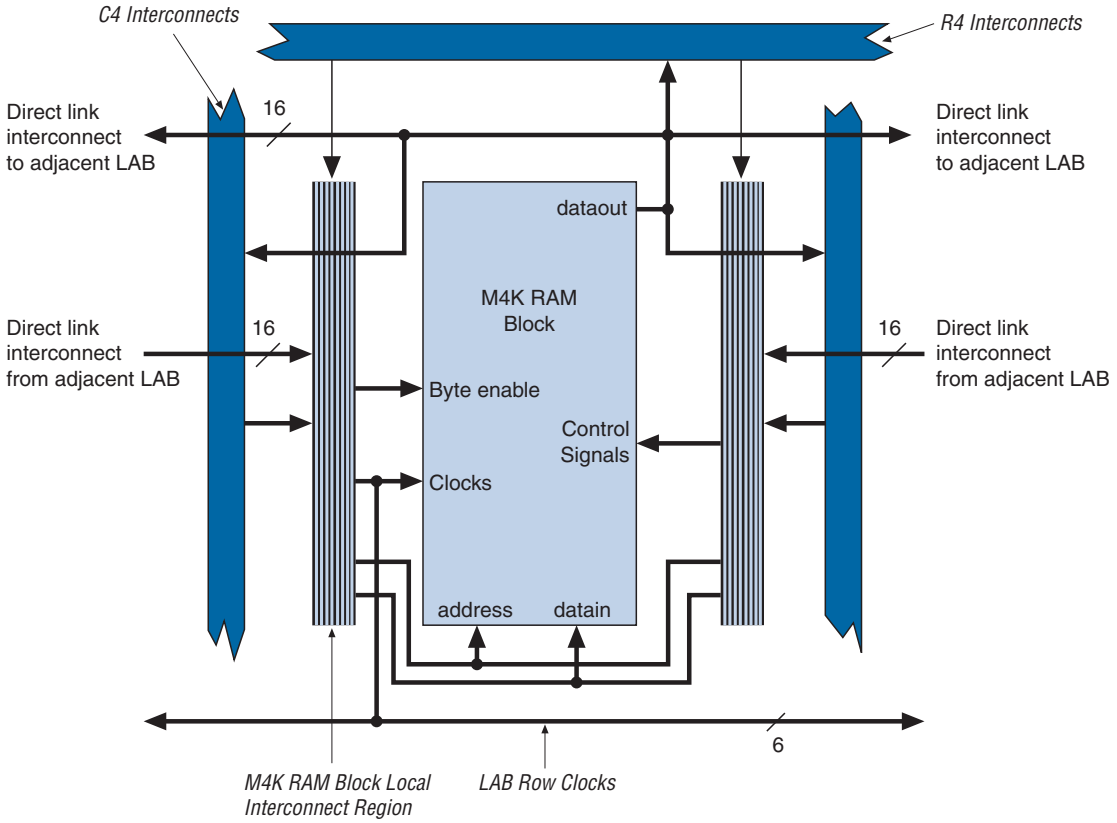
Typical soft IP cores

- 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



Cyclone II M4K RAM

```
4K × 1
2K × 2
1K × 4
512 × 8
512 × 9
256 × 16
256 × 18
128 × 32
128 × 36
```

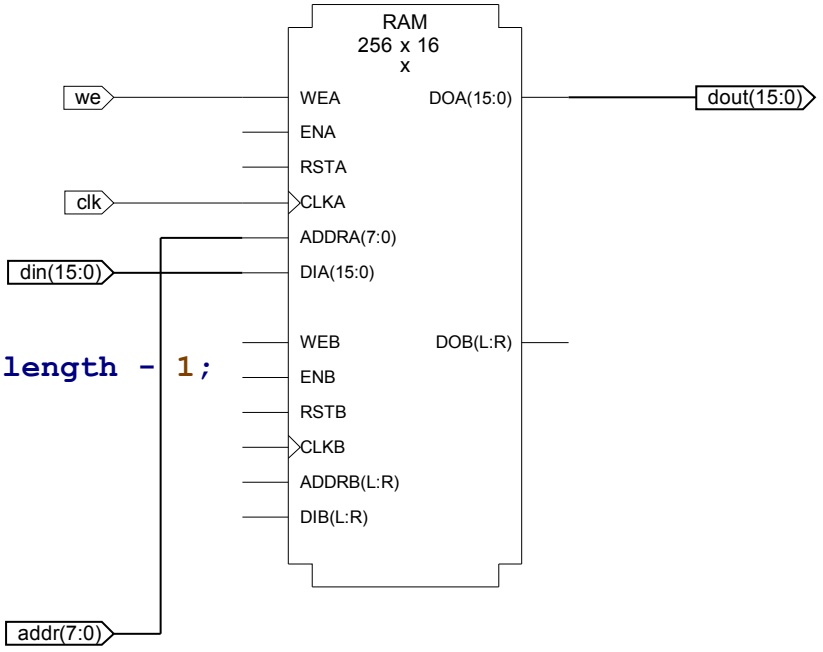
- The width can be different at the input and at the output
- **Dual port** mode
- **FIFO** mode
- up to 250 MHz

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;
```

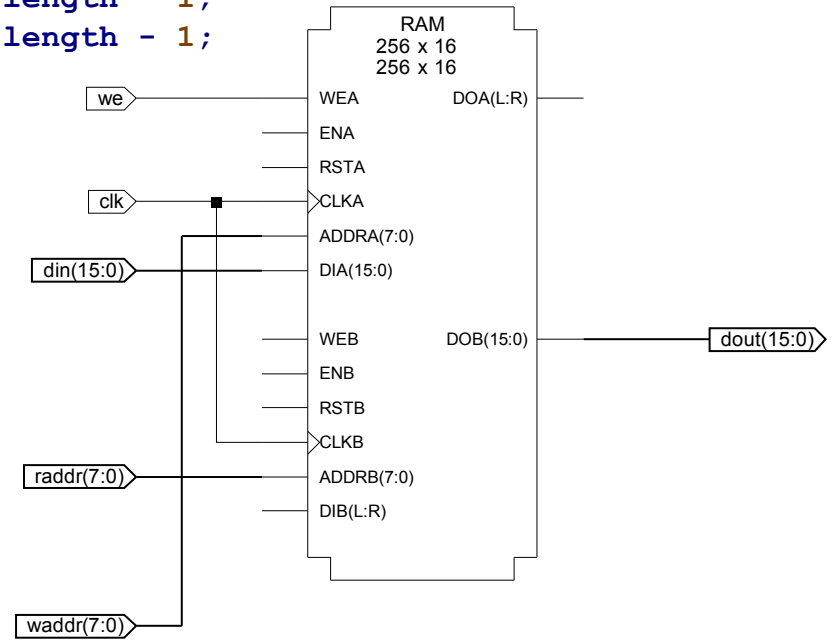


Note that asynchronous reading is not available in the RAM blocks

Dual port RAM in FPGA

```
…
entity dp_sram is
generic (Na : Positive :=  8;
         Nd : positive := 16);
port(
    …
    waddr : in  std_logic_vector(Na-1 downto 0);
    raddr : in  std_logic_vector(Na-1 downto 0);
…
signal raddri : integer range 0 to 2**raddr'length - 1;
signal waddri : integer range 0 to 2**waddr'length - 1;
begin
     waddri <= conv_integer(waddr);
ram: process(clk)
    begin
        if clk'event and clk='1' then
            raddri <= conv_integer(raddr);
            if we = '1' then
                mem_data(waddri) <= din;
            end if;
        end if;
    end process;
    dout <= mem_data(raddri);
end;
```

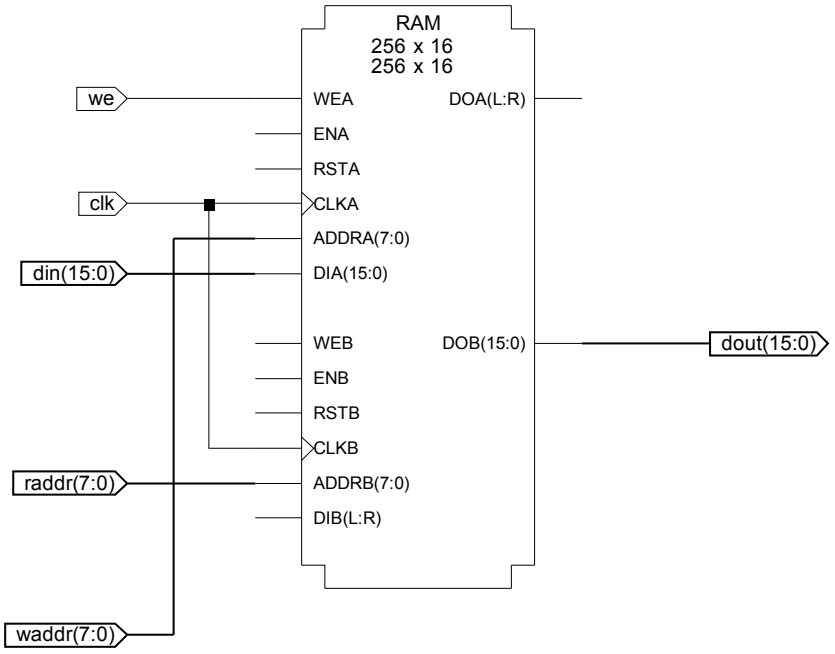
Two independent address ports (write/read) with common clock



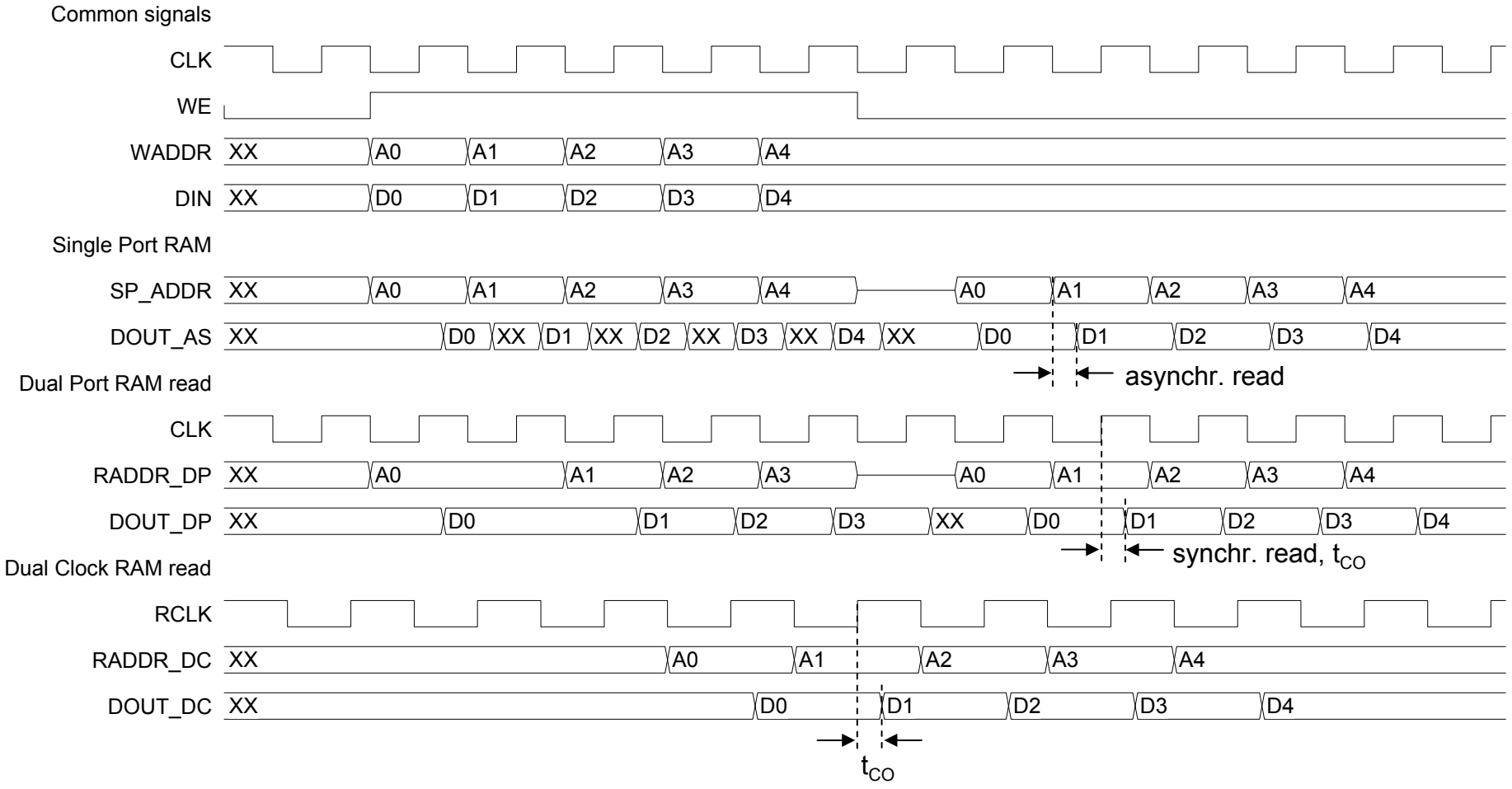
Dual port dual clock RAM in FPGA

```
…
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;
```

Two independent address ports (write/read) with different clocks



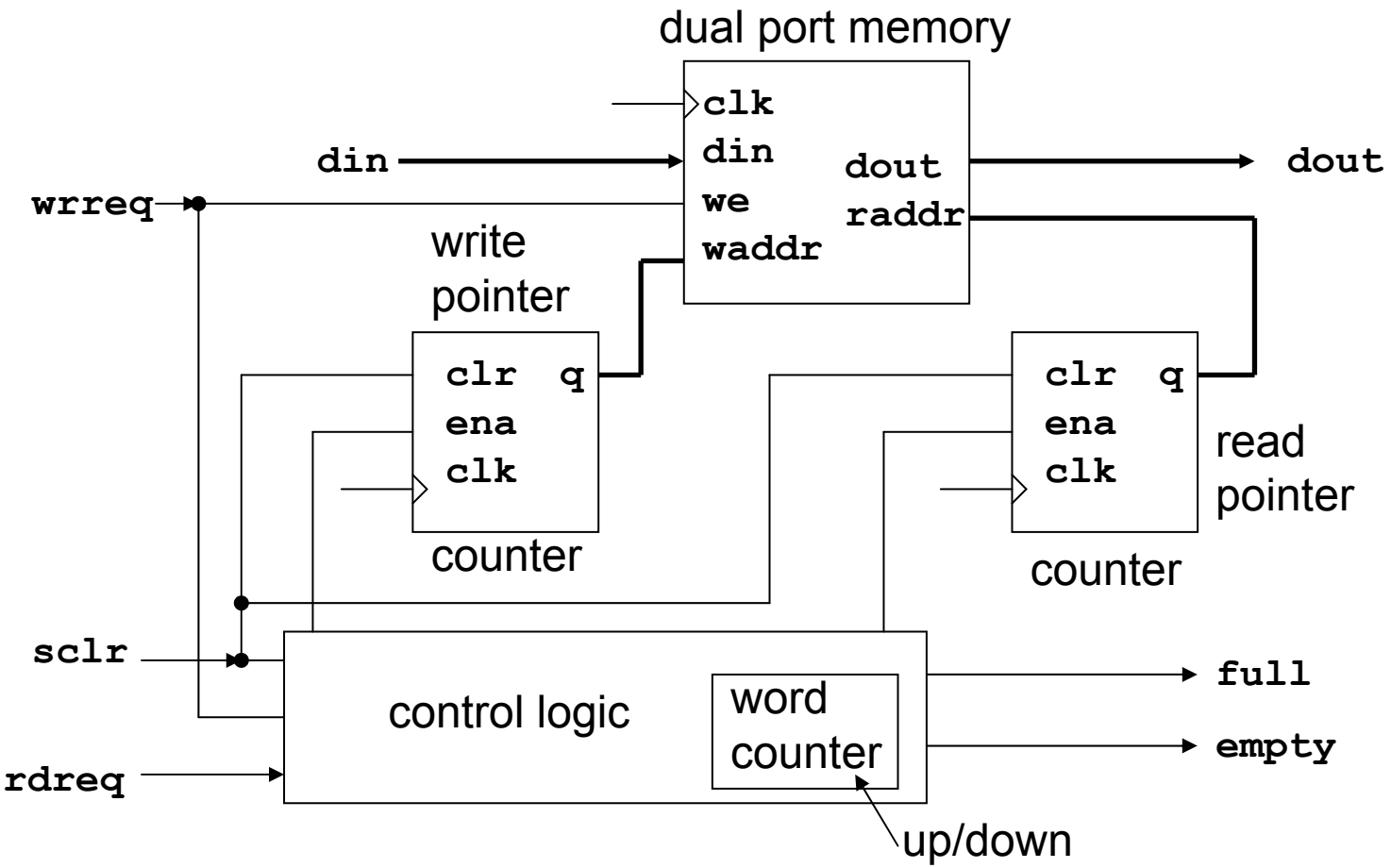
Waveforms of RAM in FPGA



FIFO

- **F**irst **I**n **F**irst **O**ut
- 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

Single clock FIFO(2)

```
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;
…
signal almost_full  : std_logic_vector(Na-1 downto 0);
signal almost_empty : std_logic_vector(Na-1 downto 0);
signal wpointer     : std_logic_vector(Na-1 downto 0);
signal rpointer     : std_logic_vector(Na-1 downto 0);
signal nwords       : std_logic_vector(Na-1 downto 0);
signal fifo_cmd     : std_logic_vector( 1 downto 0);
signal fifo_status  : std_logic_vector( 1 downto 0);
signal fifo_full    : std_logic;
signal fifo_empty   : std_logic;
begin
  almost_full  <= (0 => '0', others => '1');
  almost_empty <= (0 => '1', others => '0');
  fifo_cmd <= wrreq & rdreq;
  fifo_status <= fifo_full & fifo_empty;
```

```
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;
```

Single clock FIFO(3)

```
  when "11" => -- write & read
   case fifo_status is
   when "00" => -- not full, not empty
      wpointer <= wpointer+1;
      rpointer <= rpointer+1;
   when "10" => -- full, not empty
      nwords <= nwords -1;
      fifo_full <= '0';
   when "01" => -- not full, empty
      nwords <= nwords +1;
      fifo_empty <= '0';
   when others => NULL; -- impossible !!!
   end case;
  when "00" => NULL;
  when others => wpointer   <= (others => '-');
                 rpointer   <= (others => '-');
                 nwords     <= (others => '-');
                 fifo_full  <= '-';
                 fifo_empty <= '-';
  end case;
 end if;
end if;
end process;
```

dual port single clock memory

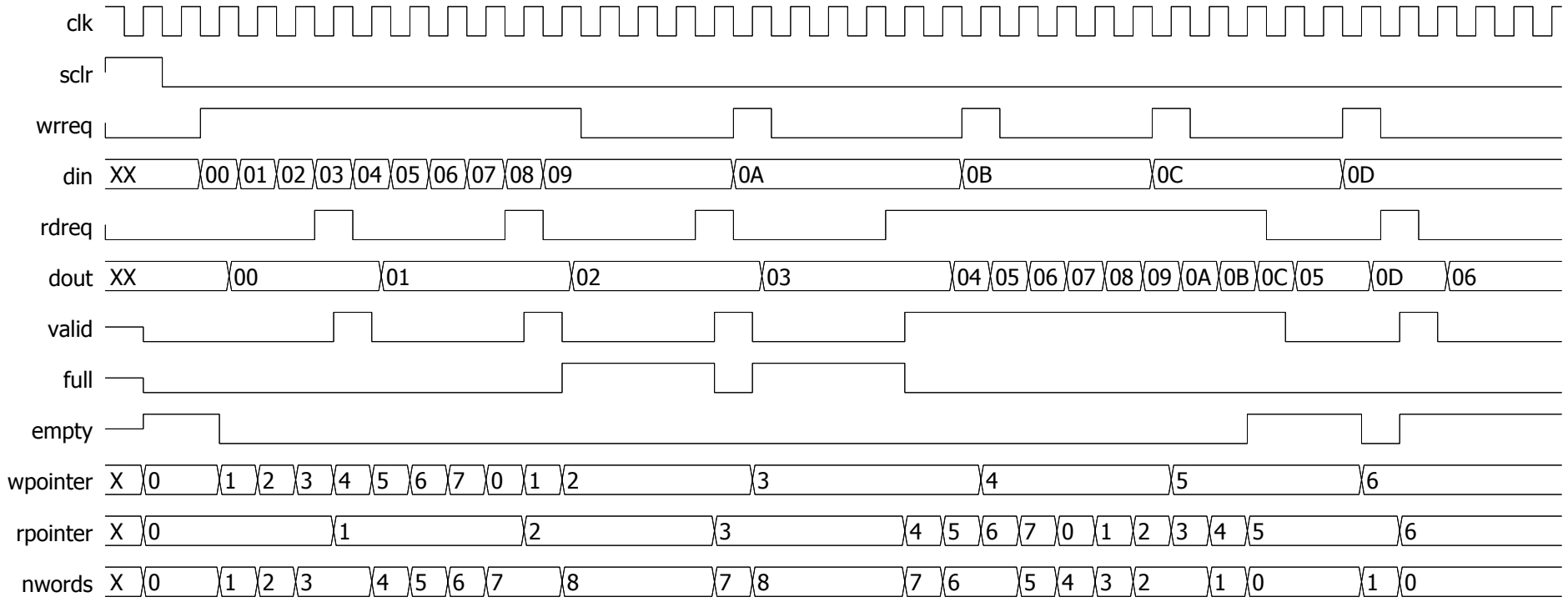
```
dpr: dp_sram
generic map(
     Na => Na,
     Nd => Nd)
PORT map(
     din   => din,
     waddr => wpointer,
     raddr => rpointer,
     we    => wrreq,
     clk   => clk,
     dout  => dout);

empty <= fifo_empty;
full  <= fifo_full;
```

Writing to a **full** FIFO and reading from an **empty** FIFO should **NEVER** happen!

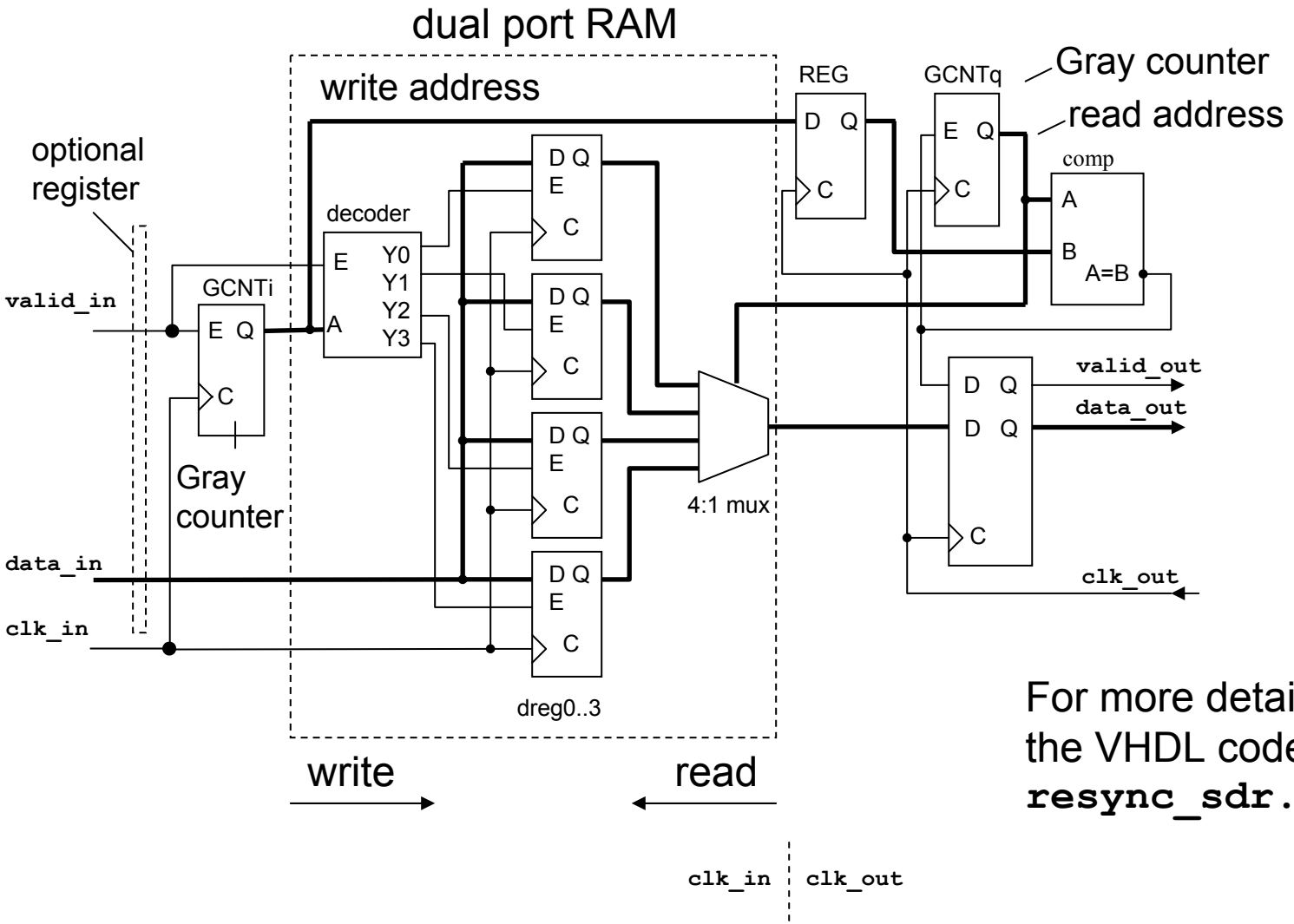
Implementing some special logic to treat these cases in the FIFO is not very reasonable (either some input data will be lost or some wrong data will be read), as it will slow down its normal operation!

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!

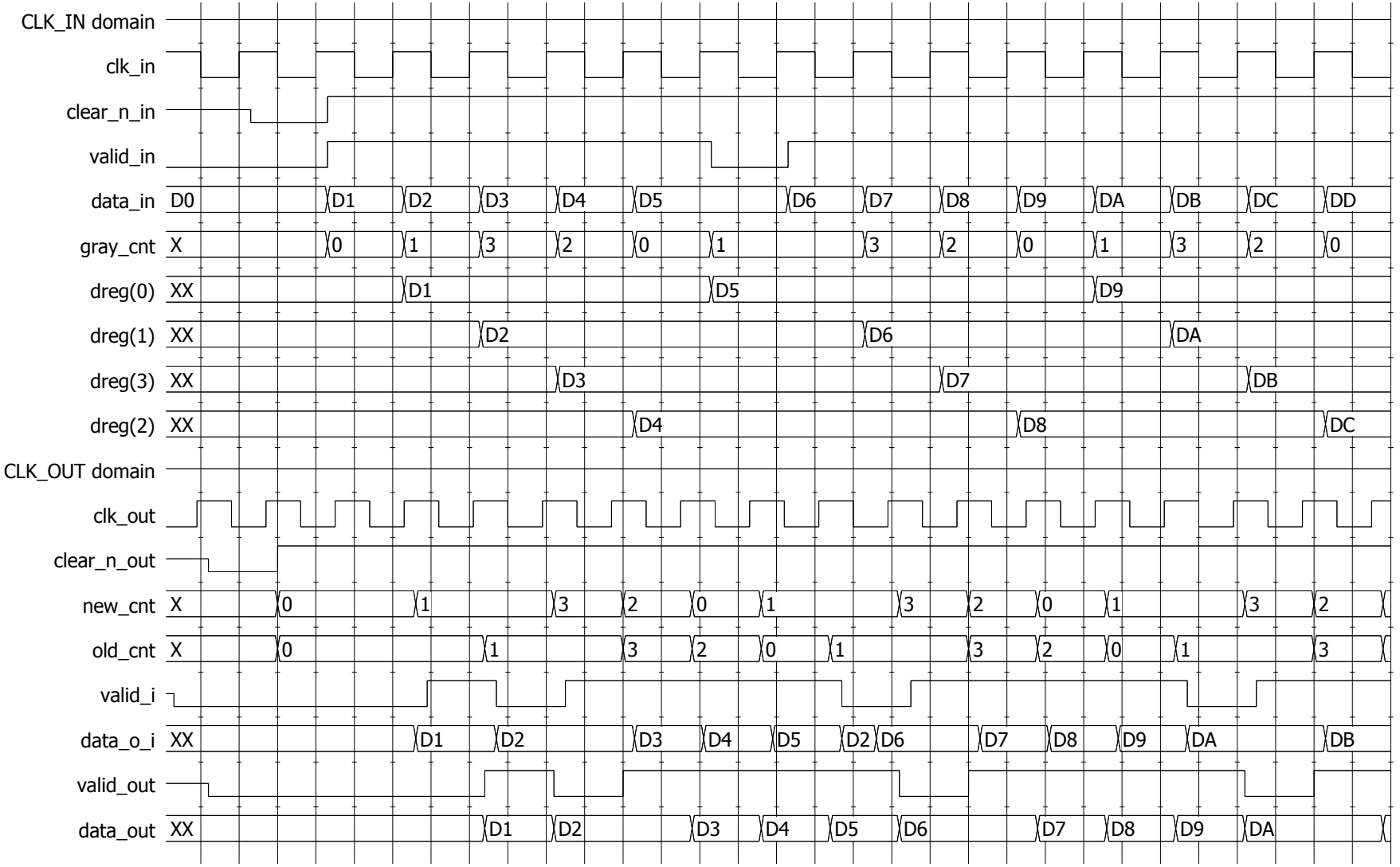


Data resynchronization with dual clock FIFO

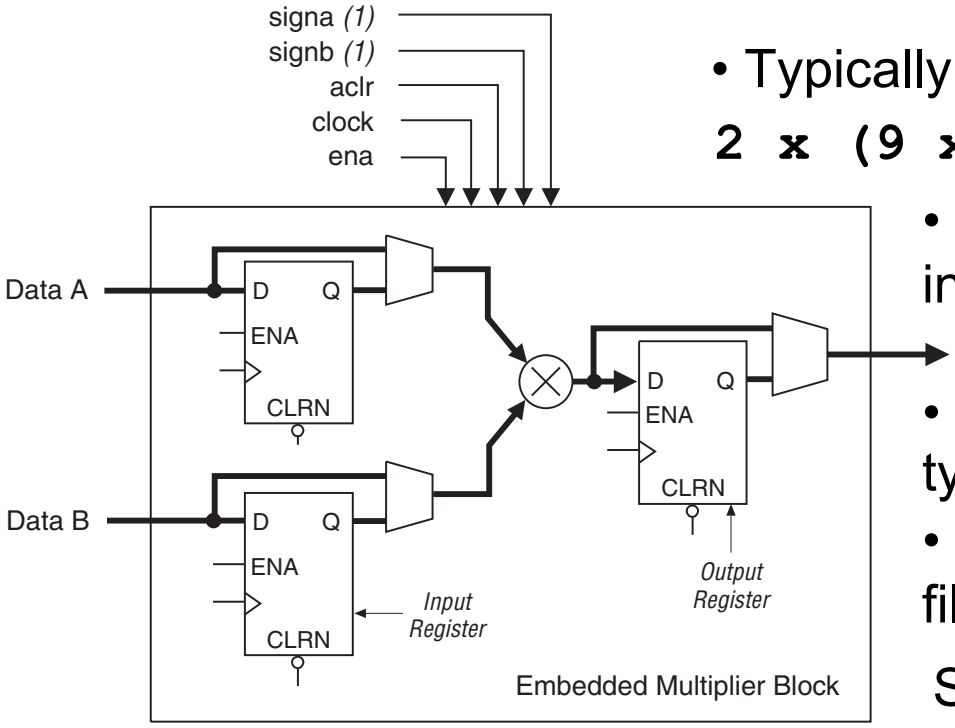


For more details see the VHDL code in **`resync_sdr.vhd`**

Data resynchronization – simulation



FPGA – Multiplier blocks



Cyclone II Multiplier

- Typically two possible configurations:

`2 x (9 x 9 → 18)` or **`18 x 18 → 36`**

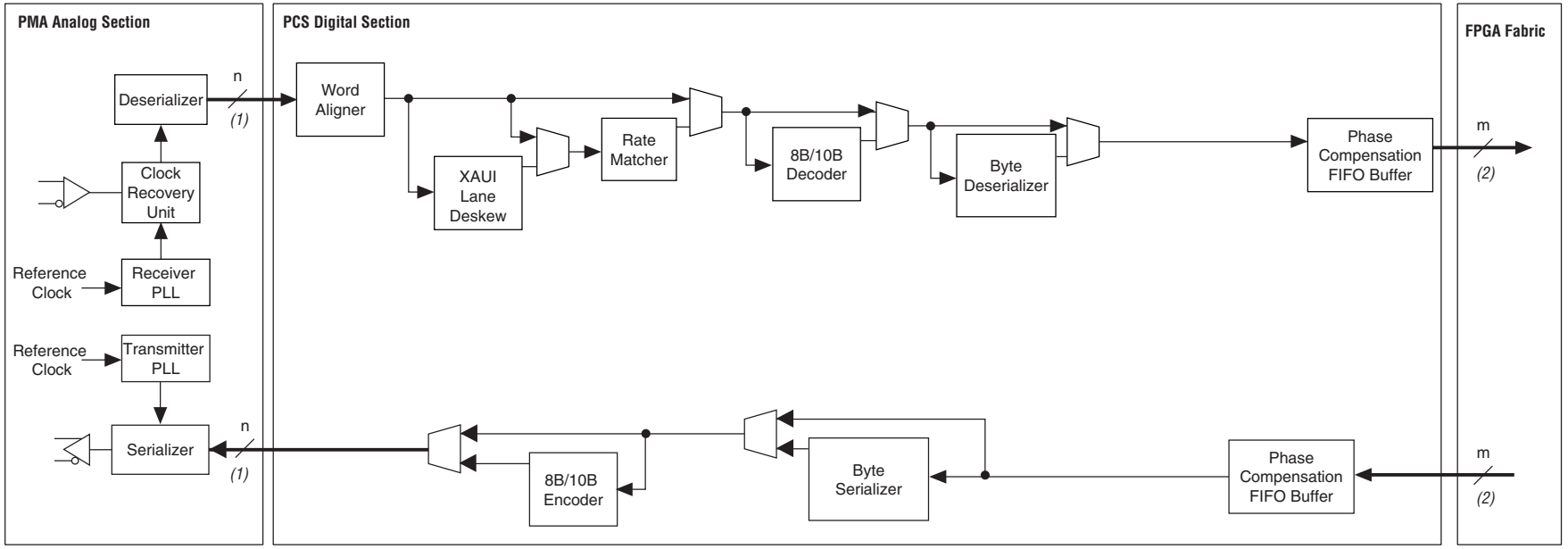
- Optional registers at the input/output
- All 4 combinations of the integer types (**`signed/unsigned`**)
- Basic element to implement digital filters and **DSP**

Size and performance without/with using multiplier blocks:

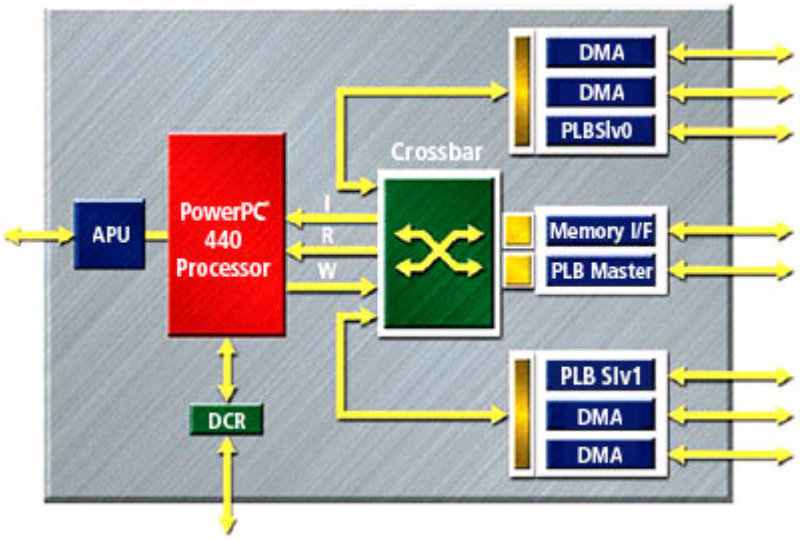
Size	f_{MAX}[MHz]	LE	f_{MAX}[MHz]	MultBlocks(9x9)
9 x 9	144	122	260	1
18 x 18	90	426	260	2

FPGA – additional hard coded modules

Multi-Gigabit serializer / deserializer



FPGA – other modules

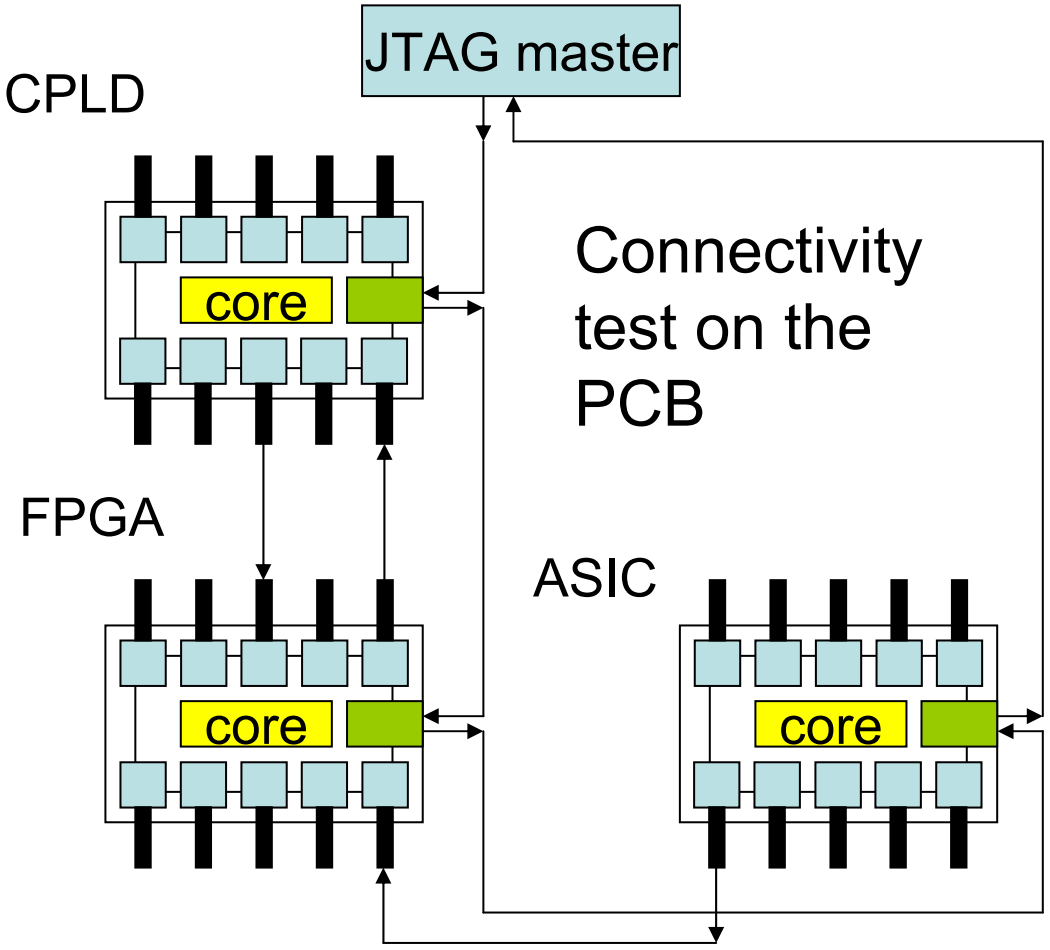


CPU cores

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