RAMs – FIFOs - Coregen

Calliope-Louisa Sotiropoulou
PhD Candidate/Researcher
Aristotle University of Thessaloniki
RAM

• **Random-access memory (RAM)** is a form of digital data storage. A random-access control circuit allows stored data to be accessed directly in any random order.

• There are two kinds of possible RAM implementations on a Xilinx FPGA:
  • The **Distributed RAM**
  • The **BRAM**
RAM

- **Distributed RAM's:**
  The configuration logic blocks (CLB) in most of the Xilinx FPGA's contain small single port or double port RAM. This RAM is normally distributed throughout the FPGA than as a single block (it is spread out over many LUT's) and so it is called "distributed RAM".

- **A look up table on a Xilinx FPGA can be configured as a 16*1bit RAM , ROM, LUT or 16bit shift register.**
RAM

- **Block RAM's:**
  A block RAM is a dedicated (cannot be used to implement other functions like digital logic) two port memory containing several kilobits of RAM. Depending on how advanced your FPGA is there may be several of them. For example, Spartan 3 has total RAM, ranging from 72 kbits to 1872 kbits in size. While Spartan 6 devices have block RAMs of up to 4824 Kbits in size.
Differences between Distributed and Block RAM's:

Distributed RAM is ideal for small sized memories. But when comes to large memories, this may cause extra wiring delays. But Block RAM's are fixed RAM modules which come in 9 kbits or 18 kbits in size. If you implement a small RAM with a block RAM then its wastage of the rest of the space in RAM. So use block RAM for large sized memories and distributed RAM for small sized memories or FIFO's.

In both, the WRITE operation is synchronous (data is written to ram only happens at rising edge of clock). But for the READ operation, distributed RAM is asynchronous (data is read from memory as soon as the address is given, doesn't wait for the clock edge) and block RAM is synchronous.
The type of inferred RAM depends on its description:
- RAM descriptions with an **asynchronous read** generate a distributed RAM.
- RAM descriptions with a **synchronous read** generate a block RAM or distributed RAM based on:
  - Tool decision
  - User specified constraint

ROMs are read-only versions of RAM.

Not every RAM feature is supported by every Xilinx device:
- Older devices have limited support
- Always read the specific device’s documentation!
There is a multitude of RAM features supported by modern FPGA devices:

- Single-port, simple-dual port, true dual port
- Up to two write ports
- Multiple read ports (depending on write ports and write address)
- Asymmetric ports (block RAM)
- Write enable
- RAM enable (block RAM)
- Data output reset (block RAM)
- Optional output register (block RAM)
- Byte-Wide Write Enable (block RAM)
- Different port clocks
- Initial contents specification
- Parity bits (block RAM — instantiation only)
RAM types

• Single port RAM
  • Single data port, one address port used both for read / write
  • Single clock

• Simple dual port (SDP) RAM
  • Two data ports, two address ports
  • One port is strictly a read port, the other is strictly a write port
  • Two different clocks

• True dual port (TDP) RAM
  • Two data ports, two address ports
  • Each port can be used for reading or writing independent of the other port
  • Two different clocks
  • Half the data width of the SDP configuration
RAM modeling

- **Modeling a RAM in VHDL**

  **Single port 64x16 RAM (64 words of 16 bit)**

  ```vhdl
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  signal RAM: ram_type;
  ```

  **Dual port 64x16 RAM (64 words of 16 bit) with two write ports**

  ```vhdl
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  shared variable RAM: ram_type;
  ```

- **Reading and writing RAM contents**

  **Read / write access**

  ```vhdl
  do <= RAM(conv_integer(addr));
  
  RAM(conv_integer(addr)) <= di;
  ```

- **IEEE.STD_LOGIC_UNSIGNED.ALL for conv_integer function**
Read / write synchronization - 1

- **Read first**
  - Old contents are read before new contents are loaded

```vhdl
if (clk'event and clk = '1') then
  if en = '1' then
    if we = '1' then
      RAM(conv_integer(addr)) <= di;
    end if;
    do <= RAM(conv_integer(addr));
  end if;
end if;
end if;
```

- **No-change**
  - Data output does not change while new contents are loaded

```vhdl
if (clk'event and clk = '1') then
  if en = '1' then
    if we = '1' then
      RAM(conv_integer(addr)) <= di;
    else
      do <= RAM(conv_integer(addr));
    end if;
  end if;
end if;
end if;
```
Read / write synchronization - 2

• Write first

• New contents are immediately made available for reading

```vhdl
if (clk'event and clk = '1') then
  if en = '1' then
    if we = '1' then
      RAM(conv_integer(addr)) <= di;
      do <= di
    else
      do <= RAM(conv_integer(addr));
    end if;
  end if;
end if;
end if;
```

or using a shared variable:

```vhdl
if (clk'event and clk = '1') then
  if en = '1' then
    if we = '1' then
      RAM(conv_integer(addr)) := di;
    end if;
    do <= RAM(conv_integer(addr));
  end if;
end if;
end if;
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

describe RAM example - 1

entity RAM64x16 is
  port(
    CLK, WE : in std_logic;
    ADDR : in std_logic_vector(5 downto 0);
    DI   : in std_logic_vector(15 downto 0);
    DO   : out std_logic_vector(15 downto 0)
  );
end RAM64x16;

architecture behv of RAM64x16 is
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  signal RAM: ram_type;
begin

  process(CLK)
  begin
    if (CLK'event and CLK = '1') then
      if WE = '1' then
        RAM(conv_integerADDR)) <= DI;
      end if;
    end if;
  end process;

  DO <= RAM(conv_integerADDR));
end behv;

64x16 Single port RAM with synchronous read

<table>
<thead>
<tr>
<th>IO Pins</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>Positive-Edge Clock</td>
</tr>
<tr>
<td>WE</td>
<td>Write enable</td>
</tr>
<tr>
<td>ADDR</td>
<td>Address port</td>
</tr>
<tr>
<td>DI</td>
<td>Data input</td>
</tr>
<tr>
<td>DO</td>
<td>Data output</td>
</tr>
</tbody>
</table>

Asynchronous read forces implementation in distributed RAM
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RAM64x16 is
port(
    CLK, WE, EN : in std_logic;
    ADDR : in std_logic_vector(5 downto 0);
    DI : in std_logic_vector(15 downto 0);
    DO : out std_logic_vector(15 downto 0)
);
end RAM64x16;

architecture behv of RAM64x16 is
    type ram_type is array (63 downto 0) of
    std_logic_vector (15 downto 0);
    signal RAM: ram_type;
begin
    process(CLK)
    begin
        if (CLK'event and CLK = '1') then
            if EN = '1' then
                if WE = '1' then
                    RAM(conv_integer(ADDR)) <= DI;
                end if;
                DO <= RAM(conv_integer(ADDR));
            end if;
        end if;
    end process;
end behv;

By default this memory will be implemented in a BRAM.

64x16 Single port RAM, read-first mode

<table>
<thead>
<tr>
<th>IO Pins</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>Positive-Edge Clock</td>
</tr>
<tr>
<td>WE</td>
<td>Write enable</td>
</tr>
<tr>
<td>EN</td>
<td>RAM enable</td>
</tr>
<tr>
<td>ADDR</td>
<td>Address port</td>
</tr>
<tr>
<td>DI</td>
<td>Data input</td>
</tr>
<tr>
<td>DO</td>
<td>Data output</td>
</tr>
</tbody>
</table>
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RAM128x8 is
port(
    CLK, WE, RE  : in std_logic;
    WADDR, RADDR : in std_logic_vector(6 downto 0);
    DI : in std_logic_vector(7 downto 0);
    DO : out std_logic_vector(7 downto 0));
end RAM128x8;

architecture behv of RAM128x8 is
    type ram_type is array (127 downto 0) of std_logic_vector (7 downto 0);
    signal RAM : ram_type;
begin

    process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if (WE = '1') then
                RAM(conv_integer(WADDR)) <= DI;
            end if;
            if (RE = '1') then
                DO <= RAM(conv_integer(RADDR));
            end if;
        end if;
    end process;

end behv;

128x8 Simple Dual port RAM, read-first mode

<table>
<thead>
<tr>
<th>IO Pins</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>Positive-Edge Clock</td>
</tr>
<tr>
<td>WE</td>
<td>Write enable</td>
</tr>
<tr>
<td>RE</td>
<td>Read enable</td>
</tr>
<tr>
<td>ADDR</td>
<td>Address port</td>
</tr>
<tr>
<td>DI</td>
<td>Data input</td>
</tr>
<tr>
<td>DO</td>
<td>Data output</td>
</tr>
</tbody>
</table>

Common clock and separate enable signals for the read and write ports
A **FIFO (First In First Out)** is a storage element where the first data that are **pushed** (stored) in the FIFO are the first data to be **popped** (read) from the FIFO.
FIFO

FIFOs are used for:

- buffering
- flow control
- clock domain crossing:
  When different parts of the design use different clock frequencies a dual clock FIFO is used to synchronize the data flow between the two different parts
FIFO

- A FIFO is usually implemented as a memory with a control logic that controls the read/write pointers according to the FIFO operation.
- Useful FIFO flags:
  - Full
  - Empty
  - Valid Data
  - Almost Full
  - Almost Empty
FIFO

- You can write code to implement a FIFO on VHDL

but....

Perhaps it’s better that someone else does this for you!!!
Coregen

- A core, also referred to as an IP core, is a pre-made component that can be used directly in your HDL design.

- Usually the available cores are optimized for time and/or space performance.

- Cores can be configured to suit your design’s requirements.

- LogiCOREs are those cores provided free by Xilinx.

- Available LogiCORE components range from simple gate components to memory components, filters, networking components, image processing components and many others.
Coregen

- Coregen can be invoked as a standalone application from the Xilinx tools
  - You must create a Coregen Project, define the target device, save it on a separate folder

- Or it can be added as a new source from the ISE Project Navigator
  - It is integrated in the current project and saved with the rest of the source files
Coregen

- Coregen generates:
  - **The netlist (.ngc file)**
    Binary Xilinx implementation netlist file containing the information required to implement the module in a Xilinx (R) FPGA.
  - **VHDL Wrapper File (.vhd file)**
    VHDL wrapper file provided to support functional simulation. This file contains simulation model customization data that is passed to a parameterized simulation model for the core.
Coregen

- **VHO Template File (.vho file)**
  VHO template file containing code that can be used as a model for instantiating a CORE Generator module in a VHDL design.

- **XCO CORE Generator Input File (.xco file)**
  CORE Generator input file containing the parameters used to regenerate a core.
Coregen

• Implementation:
  • Netlist
  • Instantiate the component in the hierarchy

• Simulation
  • Simulation model
  • Call the XilinxCoreLib library:
    
    ```
    library XILINXCORELIB;
    use XILINXCORELIB.all;
    ```
Component Instantiation – Port Map

- To instantiate a component you must first load it in the declarative part of the architecture
- Instantiate and connect it via **port map**

- e.g. an entity of an Inverter

```vhdl
entity INV is
  port (A: in STD_LOGIC;
       F: out STD_LOGIC);
end INV;
```
architecture STRUCTURE of MUX2 is
  component INV
    port (A: in STD_LOGIC;
          F: out STD_LOGIC);
  end component;

  signal SELB: STD_LOGIC;

begin

  G1: INV port map (SEL, SELB);
  -- G1: INV port map (A => SEL,
  --       F => SELB);

end;