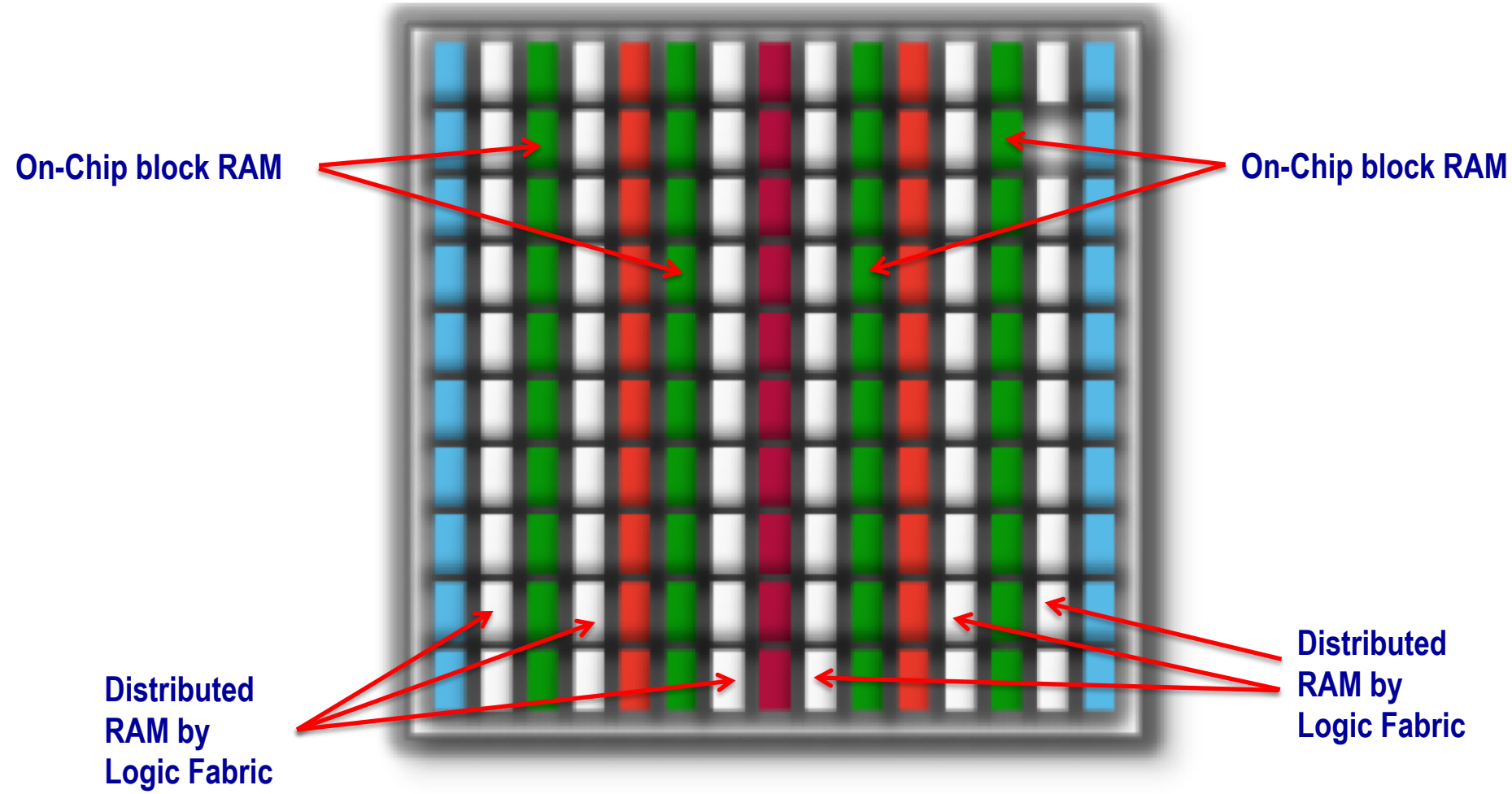
CDA 4253 FGPA System Design

Xilinx FPGA Memories

Dr. Hao Zheng
Comp Sci & Eng
University of South Florida

Xilinx 7-Series FPGA Architecture



Recommended Reading

- 7 Series FPGA Memory Resources: User Guide

  Google search: UG473

- 7 Series FPGA Configurable Logic Block: User Guide

  Google search: UG474

- Xilinx 7 Series FPGA Embedded Memory Advantages: White Paper

  Google search: WP377

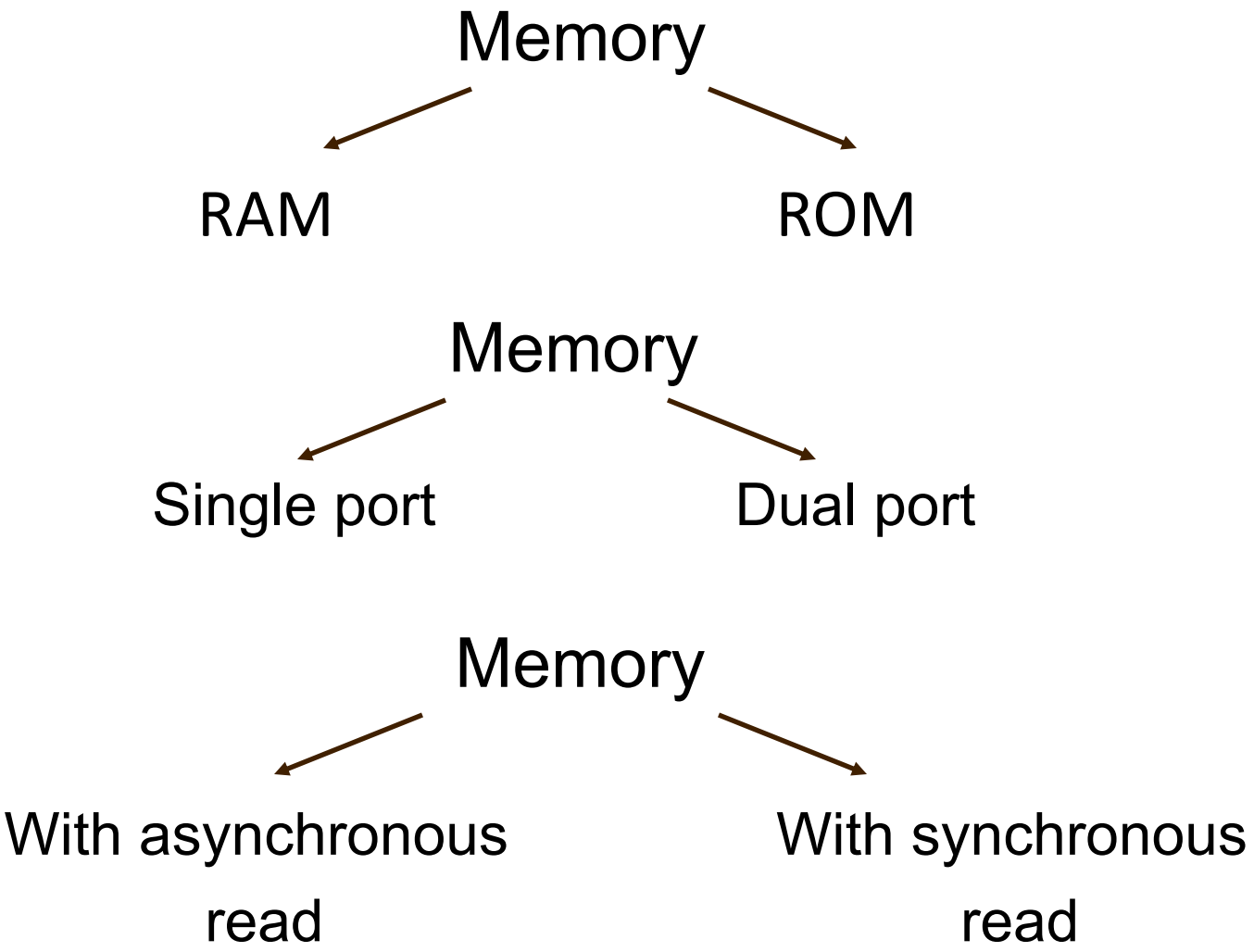
- XST User Guide for Virtex-6, Spartan-6, and 7 Series Device

  Google search: UG687

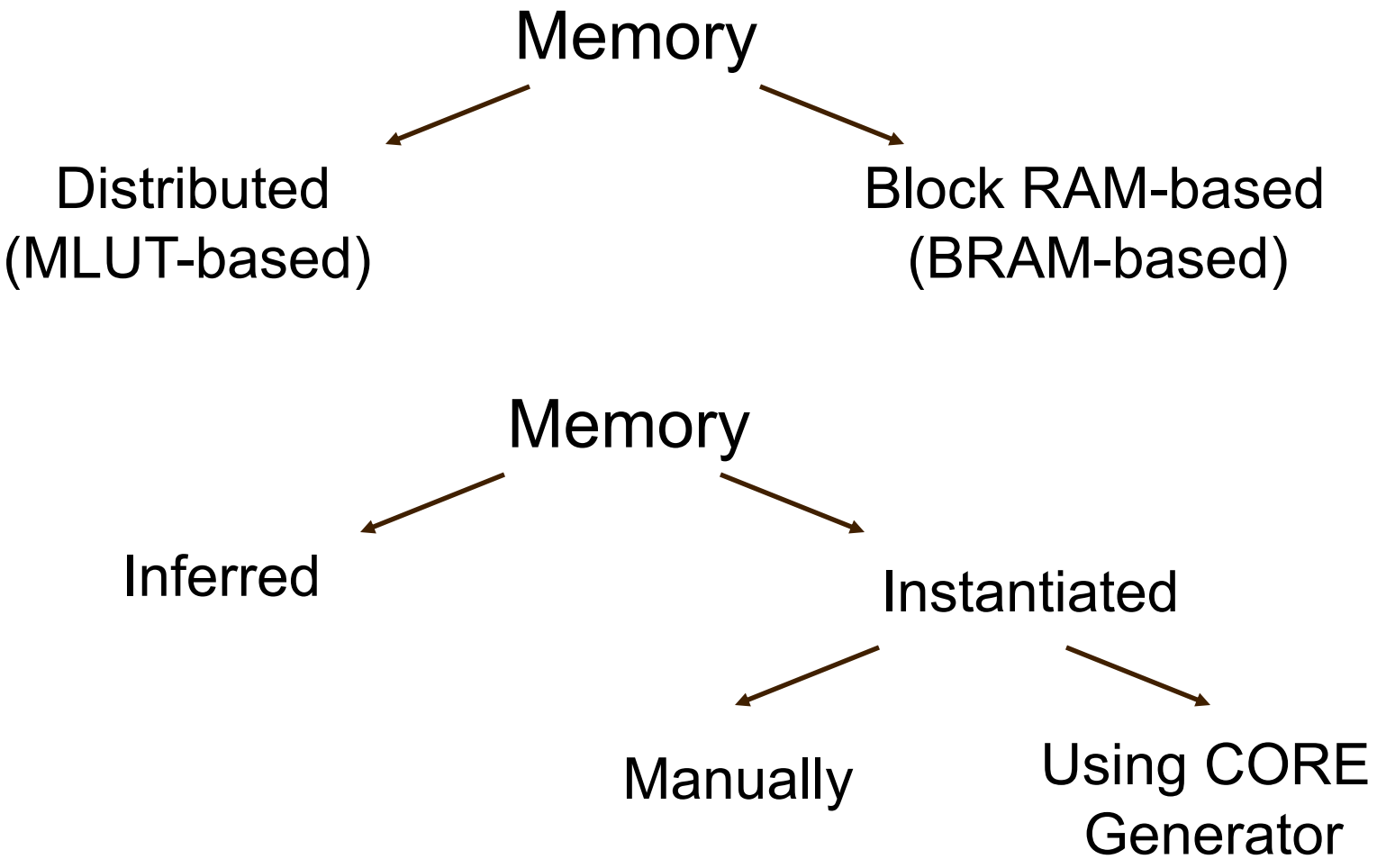
- Chu's book, chapter 7

Memory Types

Generic Memory Types



Memory Types Specific to Xilinx FPGAs



On-Chip Memory

- Distributed RAM
  - Synchronous write
  - Asynchronous read
- Block Ram
  - Synchronous write
  - Synchronous read

FPGA Distributed Memory

7 Series FPGAs Configurable Logic Block User Guide

UG474 (v1.7) November 17, 2014

Table 1: **Zynq-7000 and Zynq-7000S SoCs *(Cont'd)***

	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	**XC7Z007S**	**XC7Z012S**	**XC7Z014S**	**XC7Z010**	**XC7Z015**	**XC7Z020**	**XC7Z030**	**XC7Z035**	**XC7Z045**	**XC7Z100**
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex®-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs
	PCI Express (Root Complex or Endpoint)(3)		Gen2 x4			Gen2 x4		Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security(2)	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication									

Source: *Zynq-7000 SoC Data Sheet: Overview, DS 190*

7 Series FPGA CLB Resources

Table 2-1: **Logic Resources in One CLB**

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM(1)	Shift Registers(1)
2	8	16	2	256 bits	128 bits

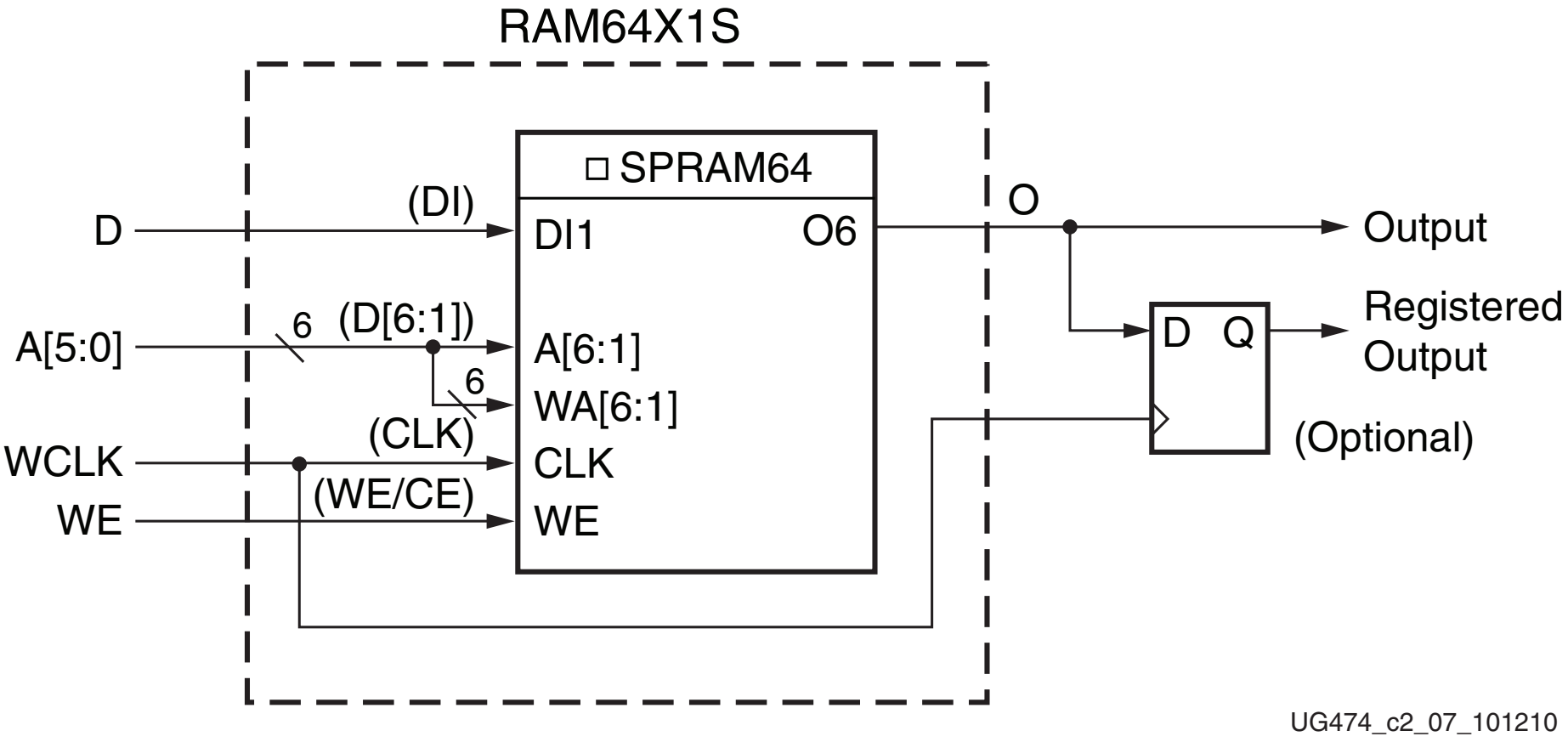
Notes:

1. SLICEM only, SLICEL does not have distributed RAM or shift registers.

7 Series FPGA Distributed RAM Config.

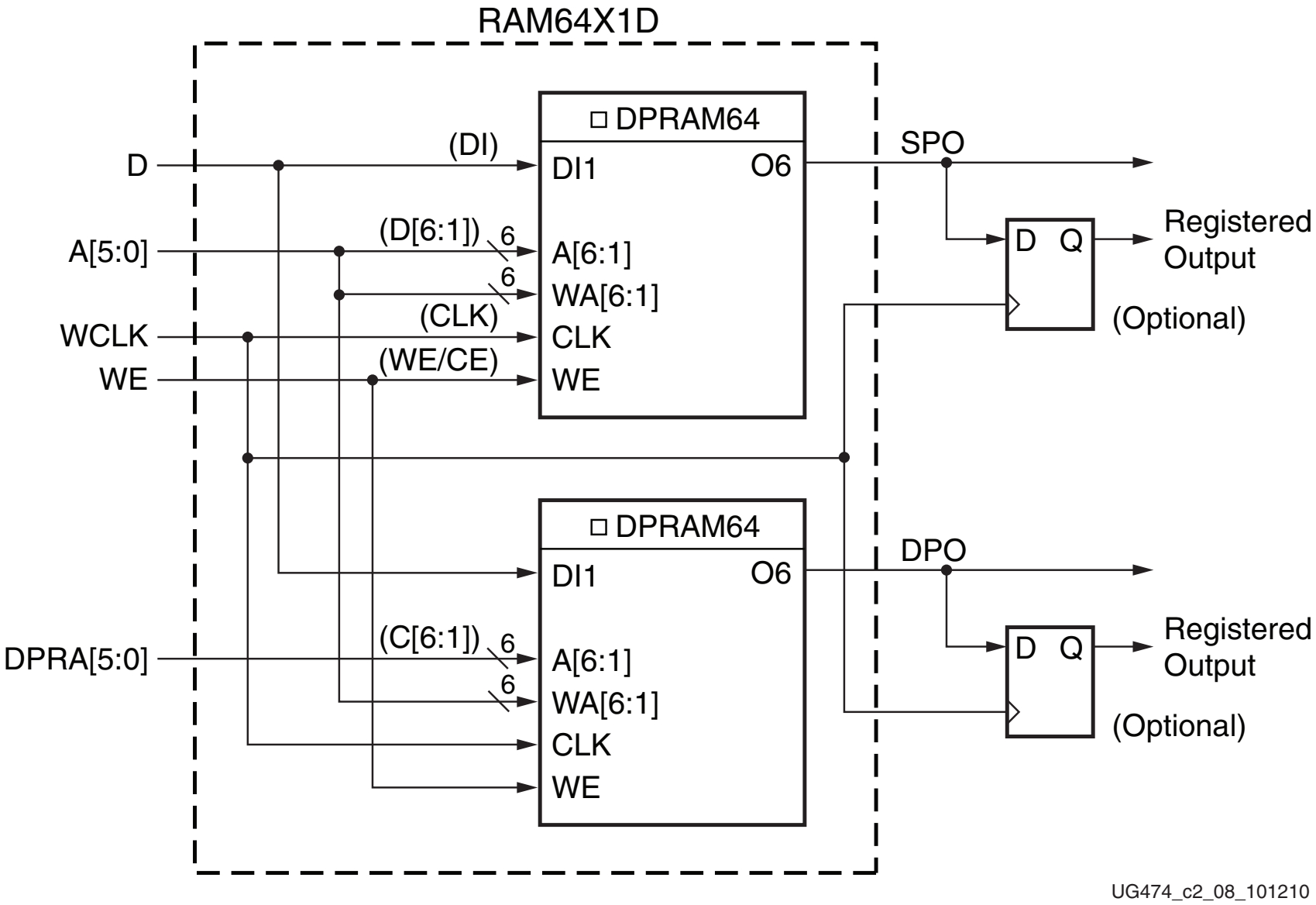
RAM	Description	Primitive	Number of LUTs
32 x 1S	Single port	RAM32X1S	1
32 x 1D	Dual port	RAM32X1D	2
32 x 2Q	Quad port	RAM32M	4
32 x 6SDP	Simple dual port	RAM32M	4
64 x 1S	Single port	RAM64X1S	1
64 x 1D	Dual port	RAM64X1D	2
64 x 1Q	Quad port	RAM64M	4
64 x 3SDP	Simple dual port	RAM64M	4
128 x 1S	Single port	RAM128X1S	2
128 x 1D	Dual port	RAM128X1D	4
256 x 1S	Single port	RAM256X1S	4

Single-Port 64x1-bit Distributed RAM



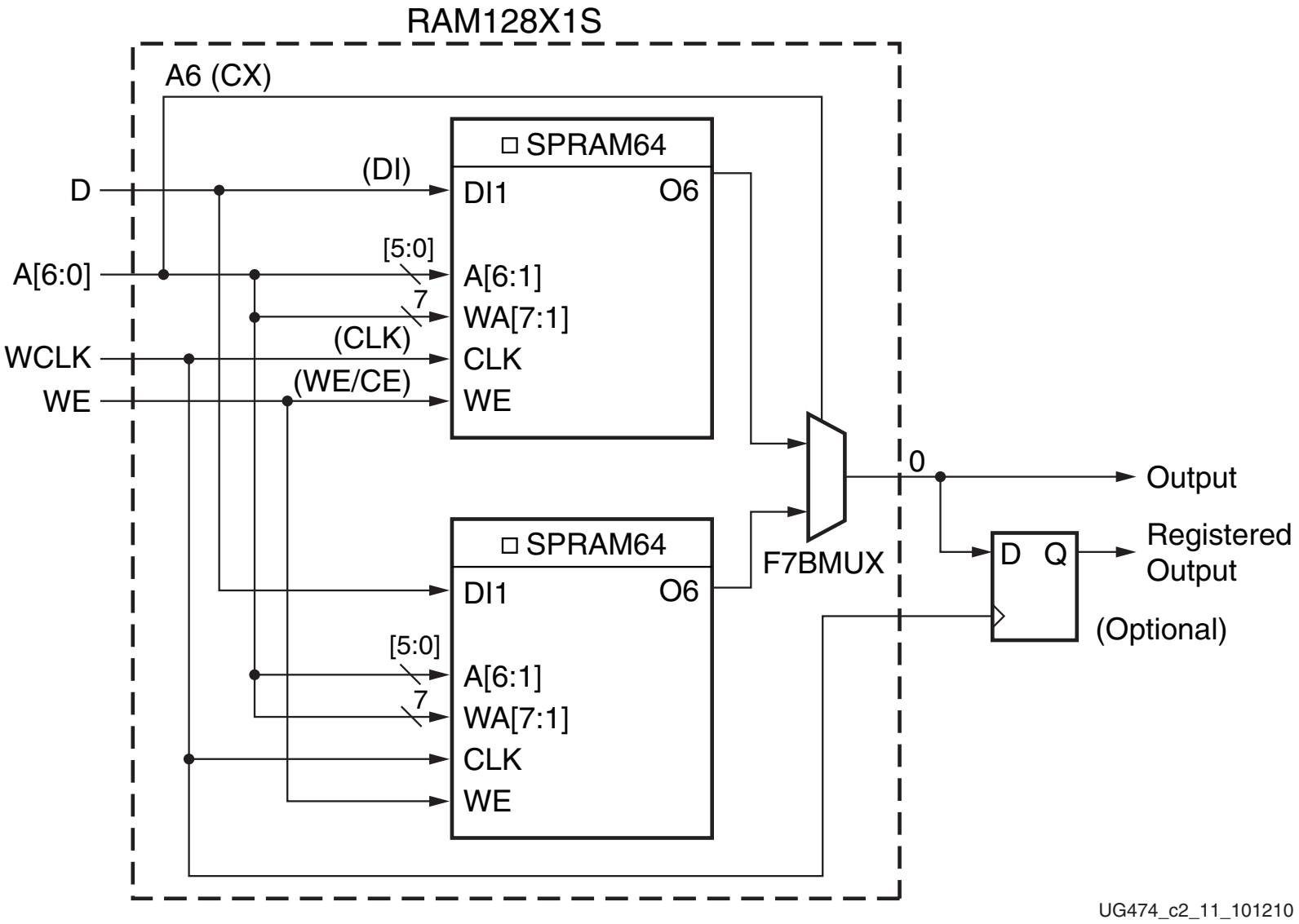
Four of these signal port 64x1 RAMs can be implemented in a single SLICEM to form a 64x4b RAM.

Dual-Port 64x1b Distributed RAM



UG474_c2_08_101210

Single-Port 128x1b Distributed RAM



7 Series FPGA ROM Configurations on LUTs

ROM	Number of LUTs
64 x 1	1
128 x 1	2
256 x 1	4

Configuration Primitives:

- ROM64X1
- ROM128X1
- ROM256X1

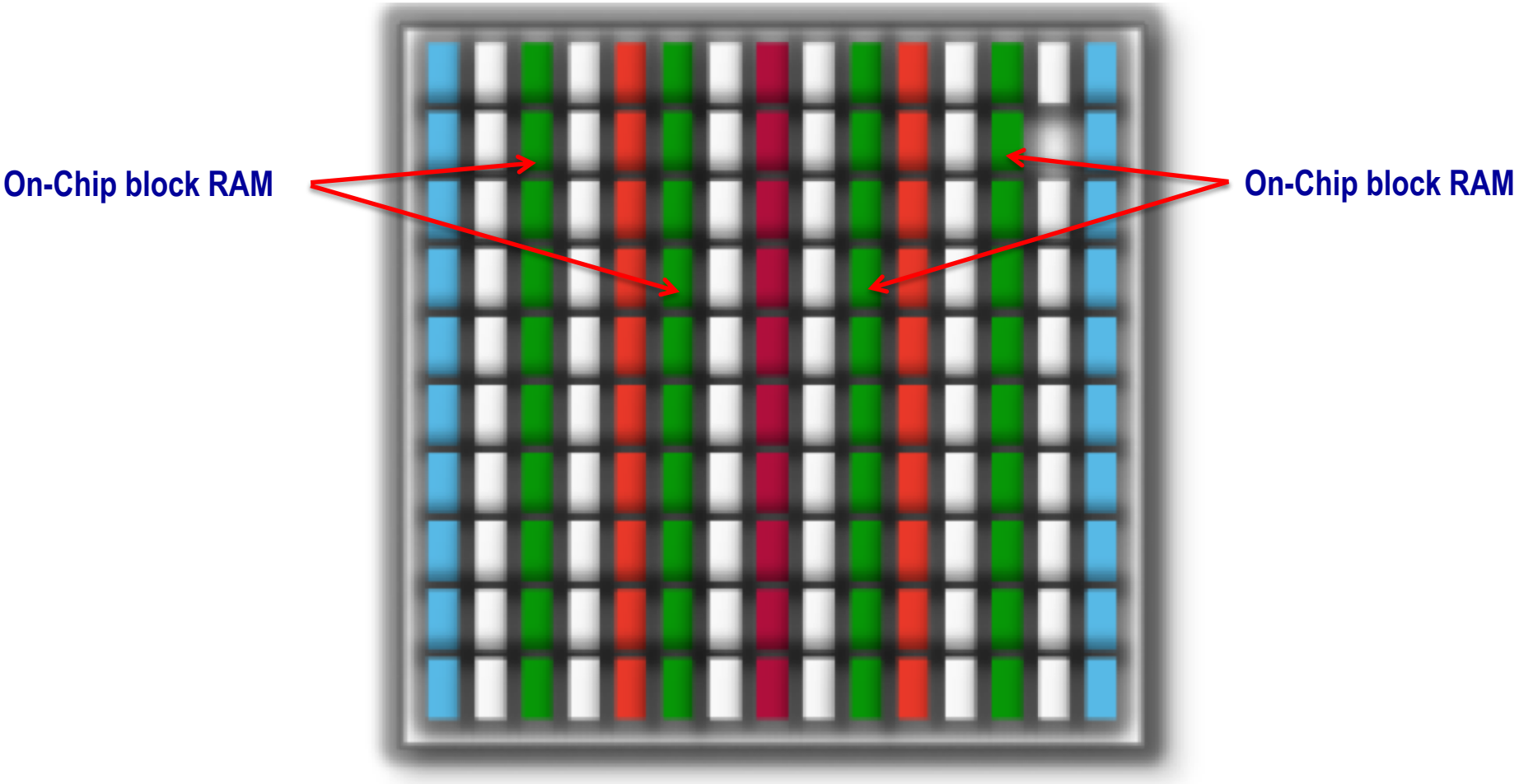
LUTs are often used to implemented small memories with less than 256 bits.

FPGA Block RAM

7 Series FPGAs Memory Resources User Guide

UG 473 2014

Location of Block RAMs



Use block RAM for storage with 64+ depth or 16+ width.

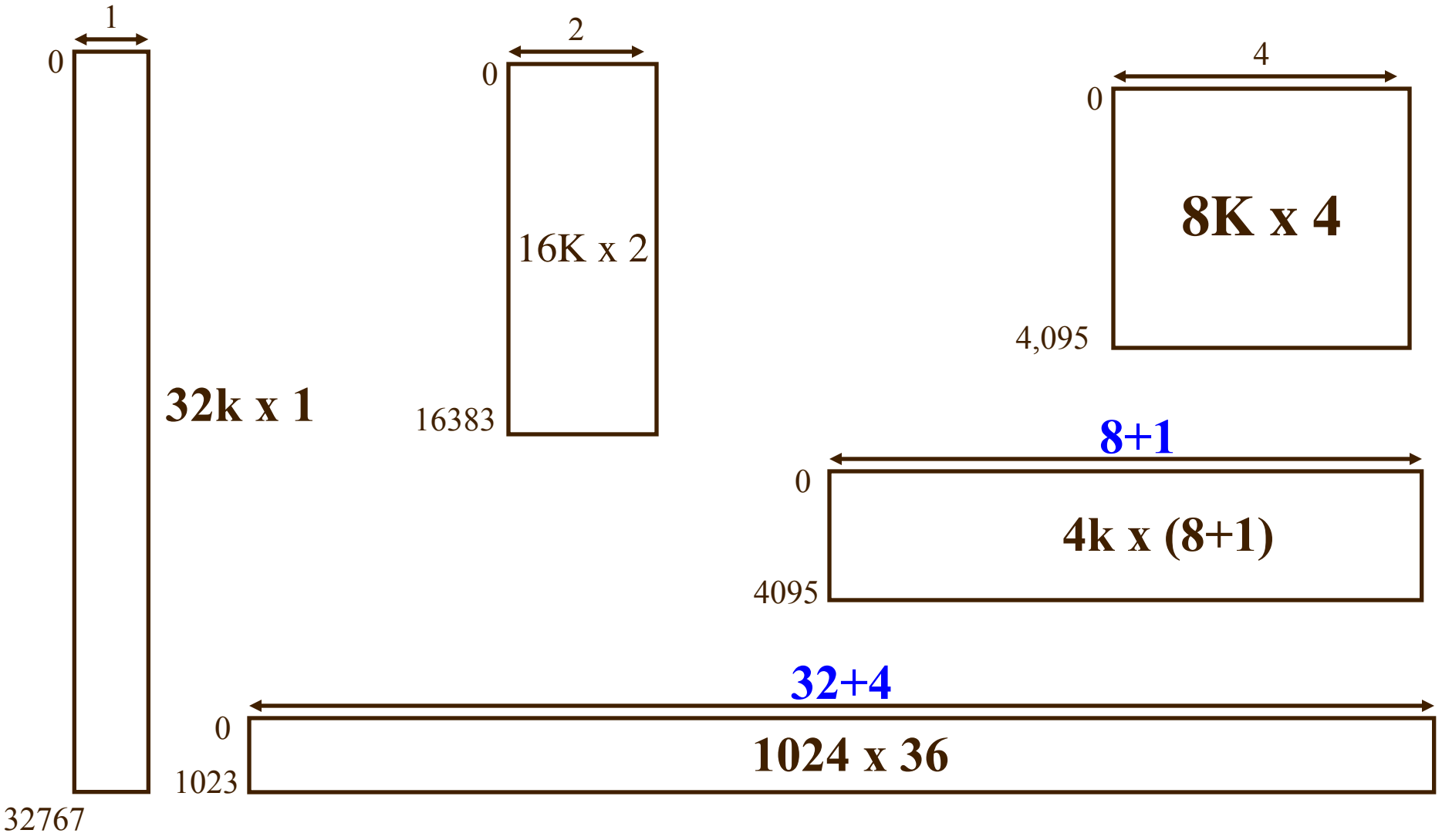
7 Series FPGA Block RAM Resources

Table 1: **Zynq-7000 and Zynq-7000S SoCs *(Cont'd)***

	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	**XC7Z007S**	**XC7Z012S**	**XC7Z014S**	**XC7Z010**	**XC7Z015**	**XC7Z020**	**XC7Z030**	**XC7Z035**	**XC7Z045**	**XC7Z100**
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex®-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs
	PCI Express (Root Complex or Endpoint)[3]		Gen2 x4			Gen2 x4		Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security[2]	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication									

Each 36Kb block RAM can be configured as two independent 18Kb RAM blocks.

Block RAM Configurations (Aspect Ratios)



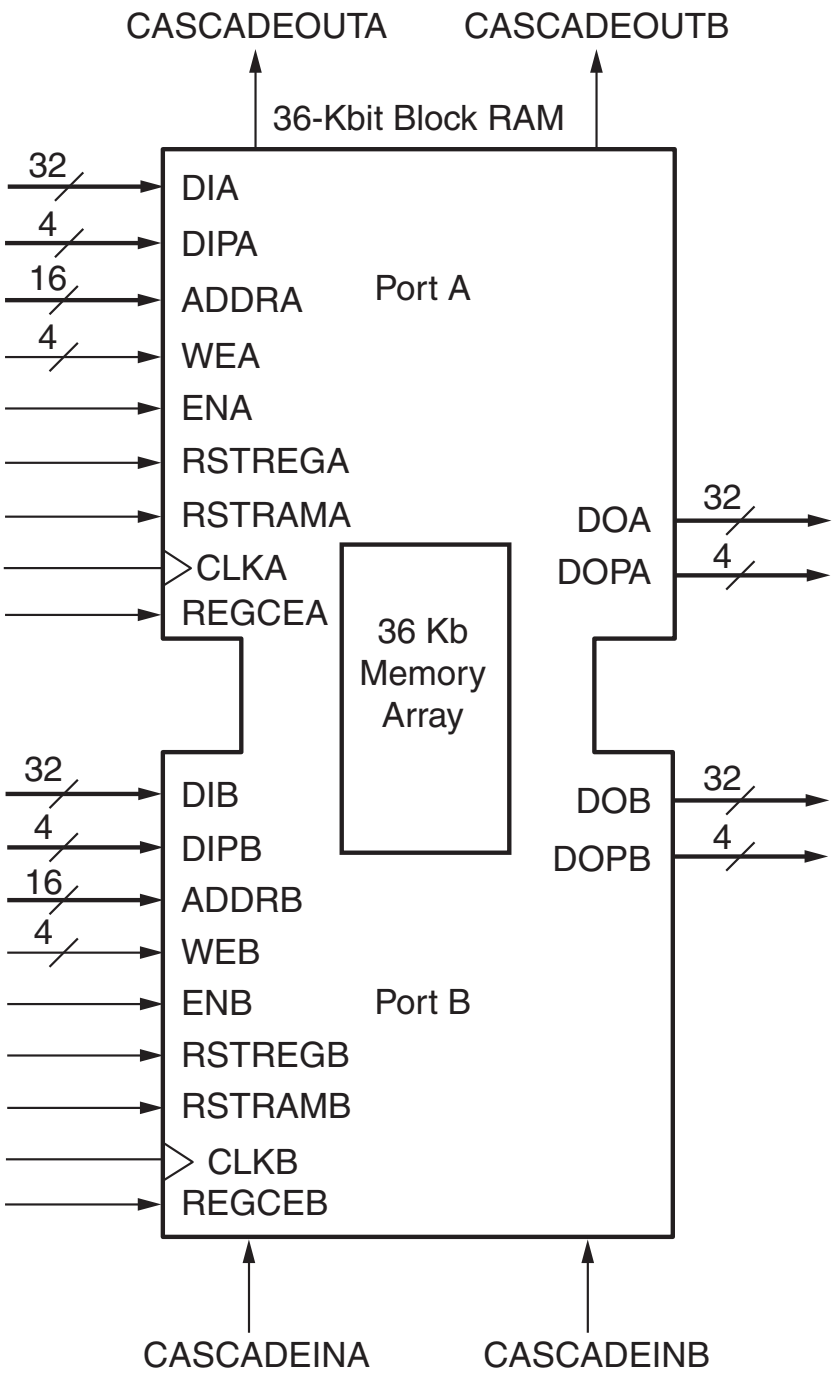
Block RAM Interface True Dual Port

Ports A and B are fully independent.

Each port has its own address, data in/out, clock, and WR enable.

Both read/write are synchronous.

Simultaneously writing to the same address causes data uncertainty.

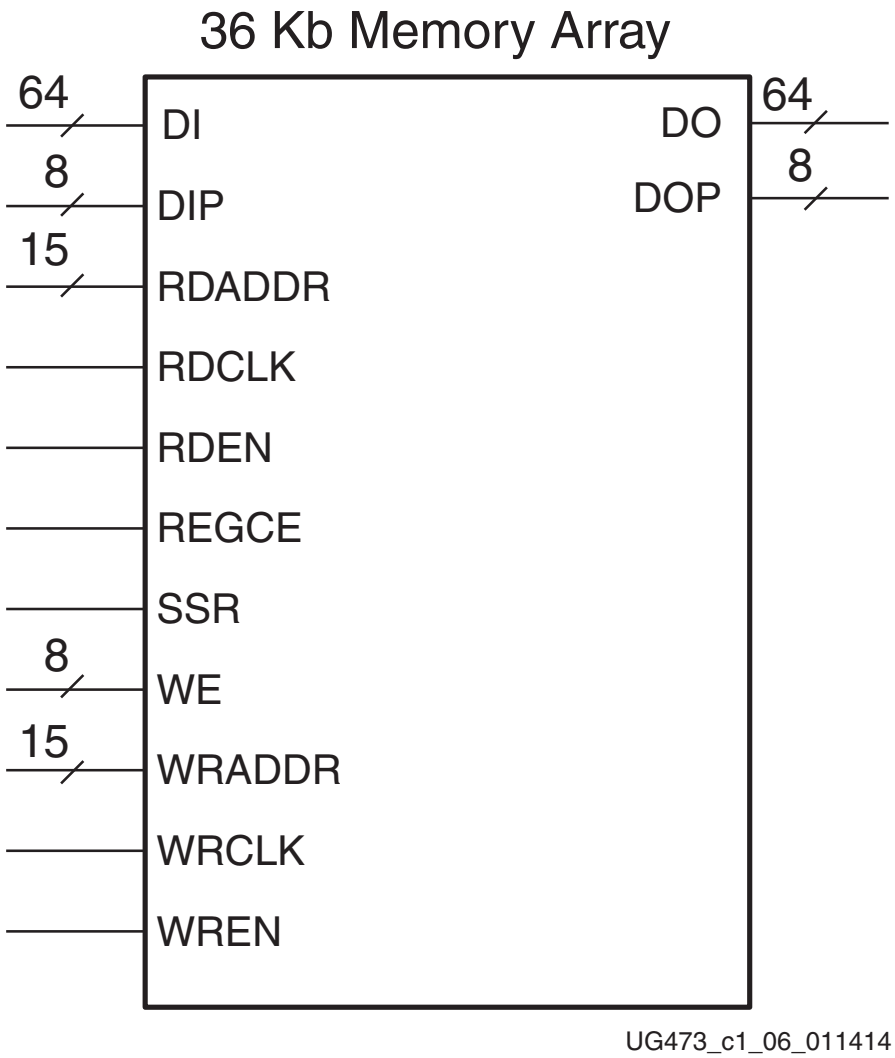


Block RAM Interface
Simple Dual Port

Independent read/write ports.

Max port width is 64+8b.

Reading & writing to the same mem location causes data uncertainty.



Inference vs. Instantiation

There are two methods to handle RAMs: instantiation and inference. Many FPGA families provide technology-specific RAMs that you can instantiate in your HDL source code. The software supports instantiation, but you can also set up your source code so that it infers the RAMs. The following table sums up the pros and cons of the two approaches.

Inference in Synthesis	Instantiation
Advantages	**Advantages**
Portable coding style	Most efficient use of the RAM primitives of a specific technology
Automatic timing-driven synthesis	Supports all kinds of RAMs
No additional tool dependencies	

VHDL Coding for Memory

XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices

Chapter 7, HDL Coding Techniques

Sections:

RAM HDL Coding Techniques

ROM HDL Coding Techniques

Distributed vs Block RAMs

- **Distributed RAM**: must be used for RAM descriptions with *asynchronous* read.
- **Block RAM**: generally used for RAM descriptions with *synchronous* read.
- *Synchronous* write for both types of RAMs.
- Any size and data width are allowed in RAM descriptions.
  - Depending on resource availability
- Up to two write ports are allowed.

Inferring ROM

Distributed ROM with Asynchronous Read

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ROM is
   generic(w : integer := 7; -- number of bits per ROM word
           r : integer := 4   -- 2^r = number of words in ROM
            );
    port (addr  : in std_logic_vector(r-1 downto 0);
          dout  : out std_logic_vector(w-1 downto 0));
end ROM;
```

Distributed ROM with Asynchronous Read

```
architecture behavioral of ROM is
   type rom_type is array (2**r-1 downto 0) of
              std_logic_vector (w-1 downto 0);
   constant ROM_array : rom_type := (
              ”1000000", ”1111001", ”0100100", ”0110000”,
              ”0011001", ”0010010", "0000010", "1111000”,
              “0000000”. “0010000”, “0001000”, “0000011”,
              “1000110”, “0100001”, “0000110”, “0001110”);
begin

   dout <= ROM_array(conv_integer(addr));

end architecture behavioral;
```

How is it implemented?

Dual-Port ROM with Sync. Read in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity roms_dualport is
    port (clk          : in std_logic;
          ena,   enb   : in std_logic;
          addra, addrb : in std_logic_vector(5 downto 0);
          dataa, datab : out std_logic_vector(19 downto 0));
end roms_dualport;

architecture behavioral of roms_dualport is

    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");
```

can be implemented either on LUTs or block RAMs.

Dual-Port ROM with Sync. Read in VHDL

```
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if (ena = '1') then
                dataa <= ROM(conv_integer(addra));
            end if;
        end if;
    end process;

    process (clk)
    begin
        if rising_edge(clk) then
            if (enb = '1') then
                datab <= ROM(conv_integer(addrb));
            end if;
        end if;
    end process;

end behavioral;
```

Port 1

Port 2

Design Example

How to implement $f = \frac{9}{5} \times c + 32$?

Inferring RAM

Single-Port RAM with Async. Read

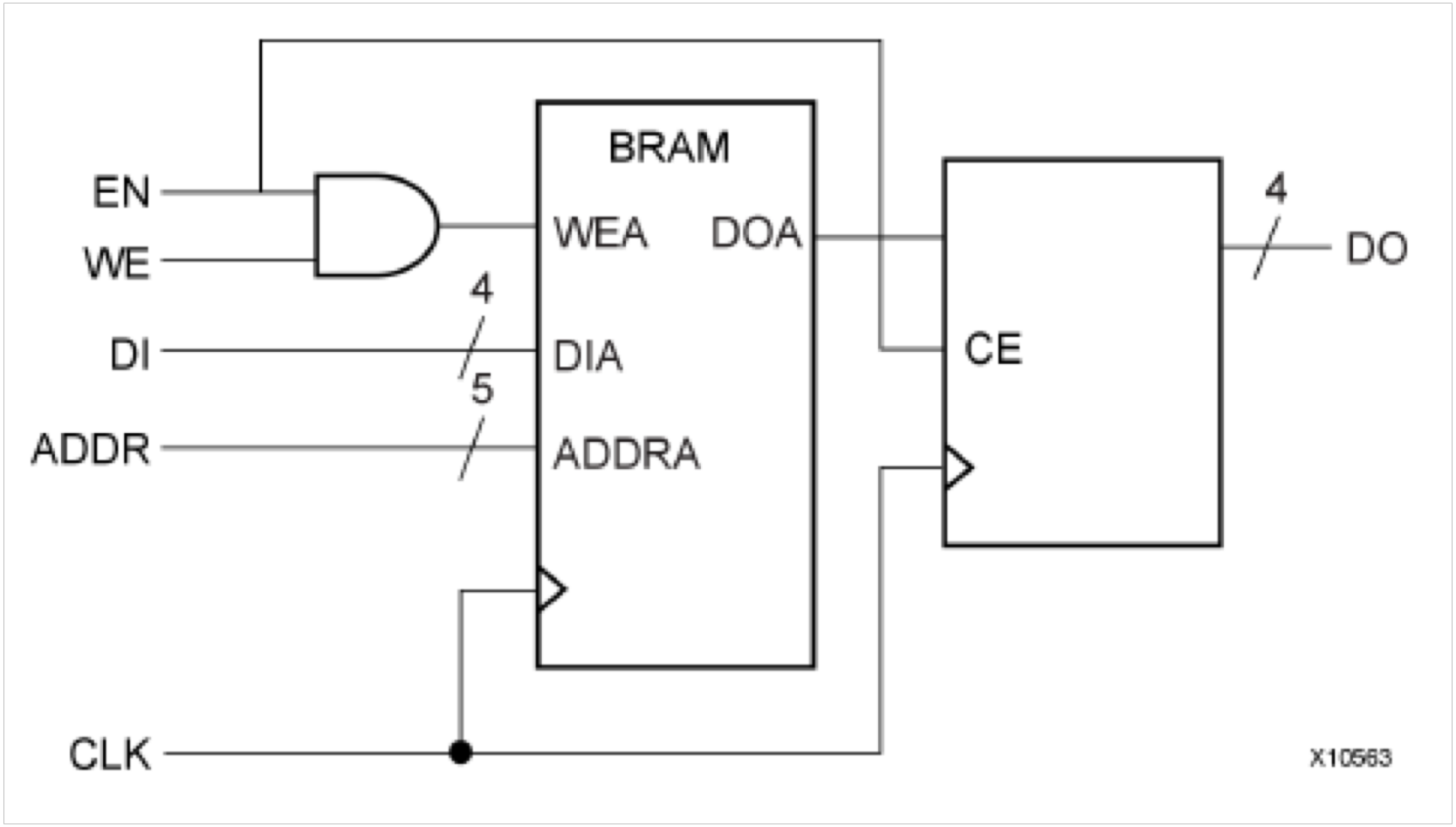
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity raminfr is
   generic(w : integer := 32;  -- number of bits per RAM word
           r : integer := 3    -- 2^r = number of words in RAM
             );
   port(clk  : in  std_logic;
        we   : in  std_logic;
        addr: in  std_logic_vector(r-1 downto 0);
        di   : in  std_logic_vector(w-1 downto 0);
        do   : out std_logic_vector(w-1 downto 0));
end raminfr;
```

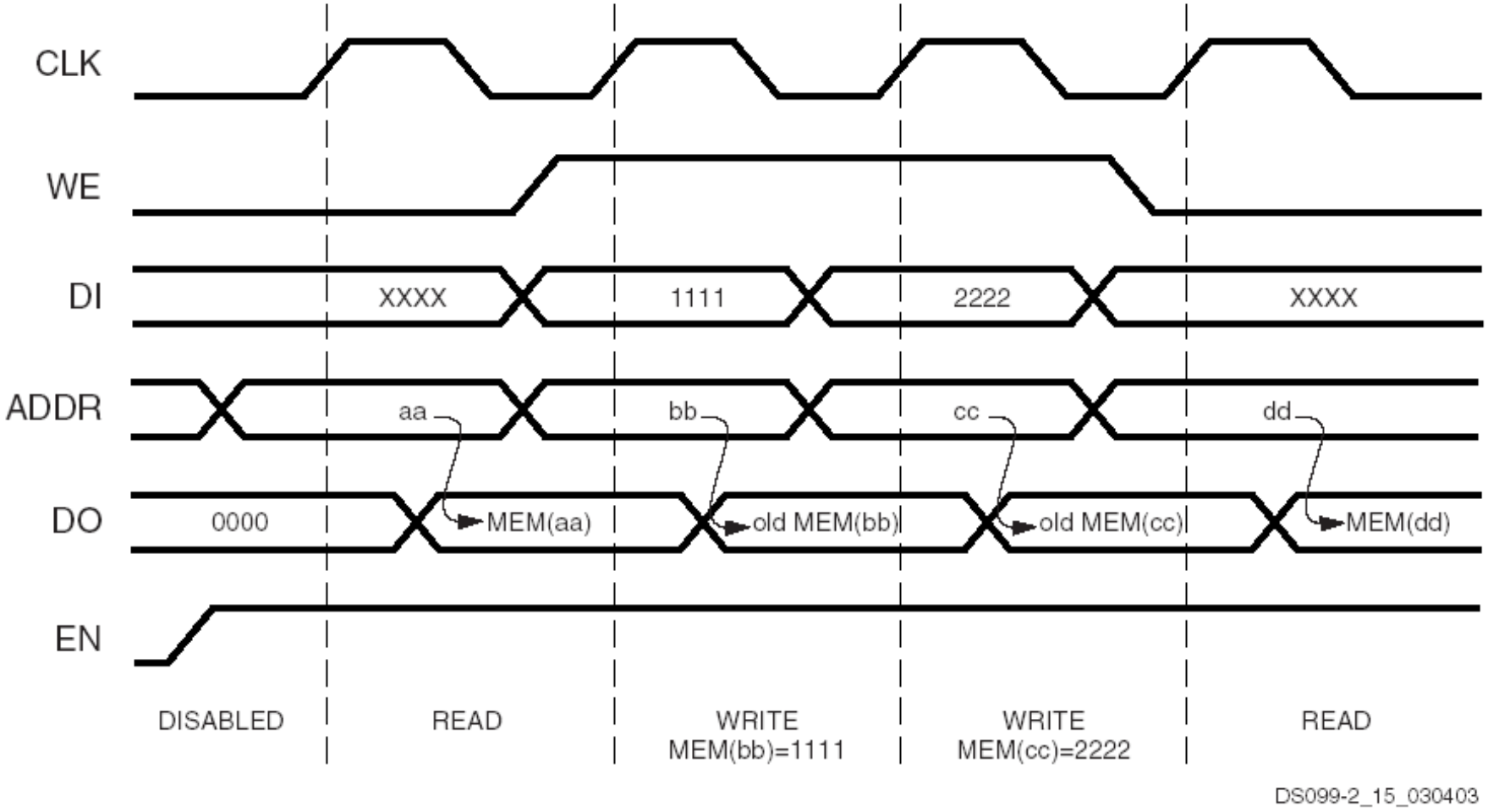
Single-Port RAM with Async. Read – cont'd

```
architecture behavioral of raminfr is
   type ram_type is array (2**r-1 downto 0) of
                          std_logic_vector (w-1 downto 0);
   signal RAM : ram_type;
begin
    process (clk)
    begin
       if rising_edge(clk) then
           if (we = '1') then
               RAM(conv_integer(addr)) <= di;
           end if;
       end if;
    end process;
    do <= RAM(conv_integer(addr));-- async read
end behavioral;
```

Block RAM with Sync. Read (Read-First Mode)



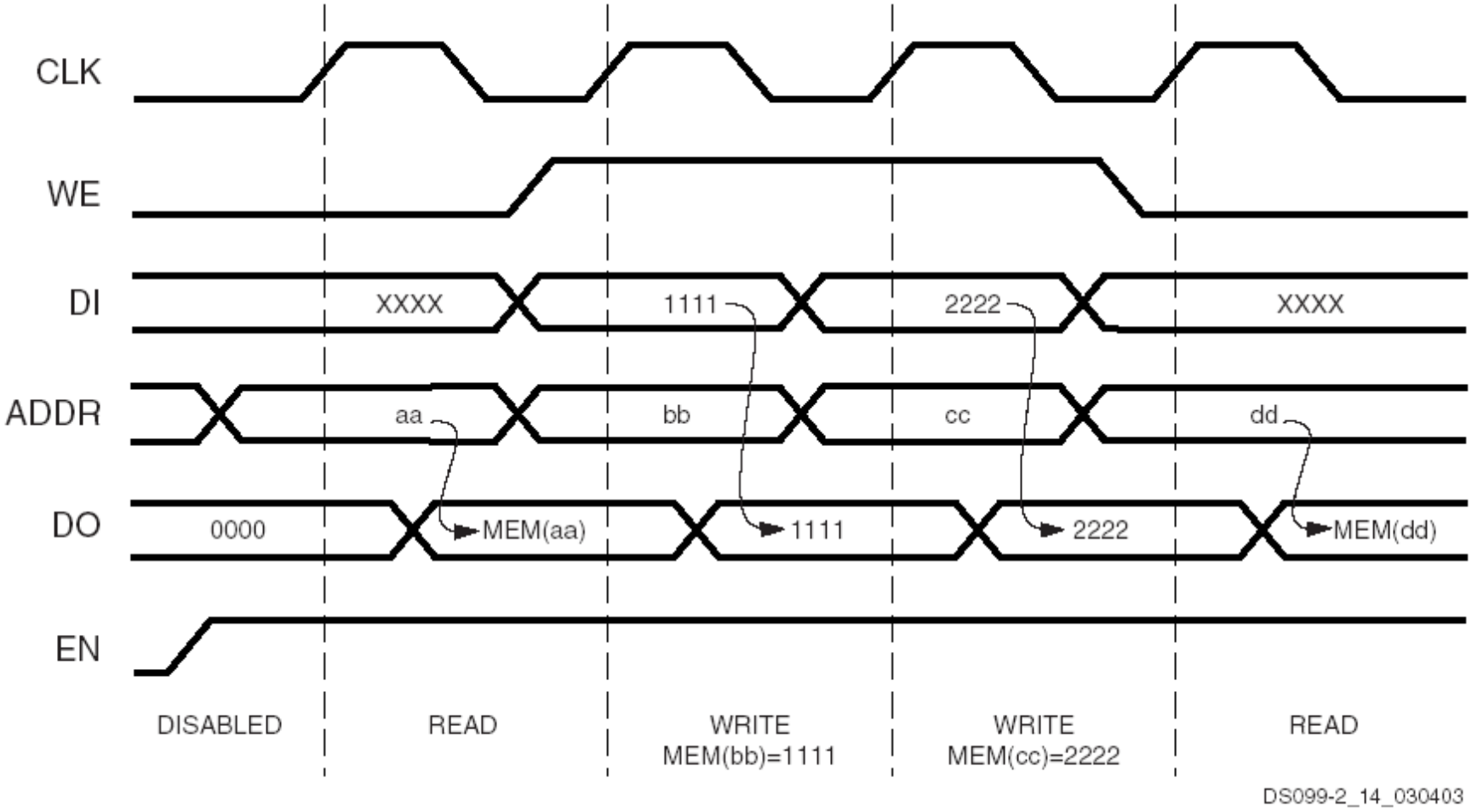
Block RAM with Sync. Read (Read-First Mode)



Block RAM with Sync. Read (Read-First Mode)

```
process (clk)
begin
   if rising_edge(clk) then
     if (en = '1') then
        do <= RAM(conv_integer(addr));
        if (we = '1') then
           RAM(conv_integer(addr)) <= di;
        end if;
     end if;
   end if;
end process;
```

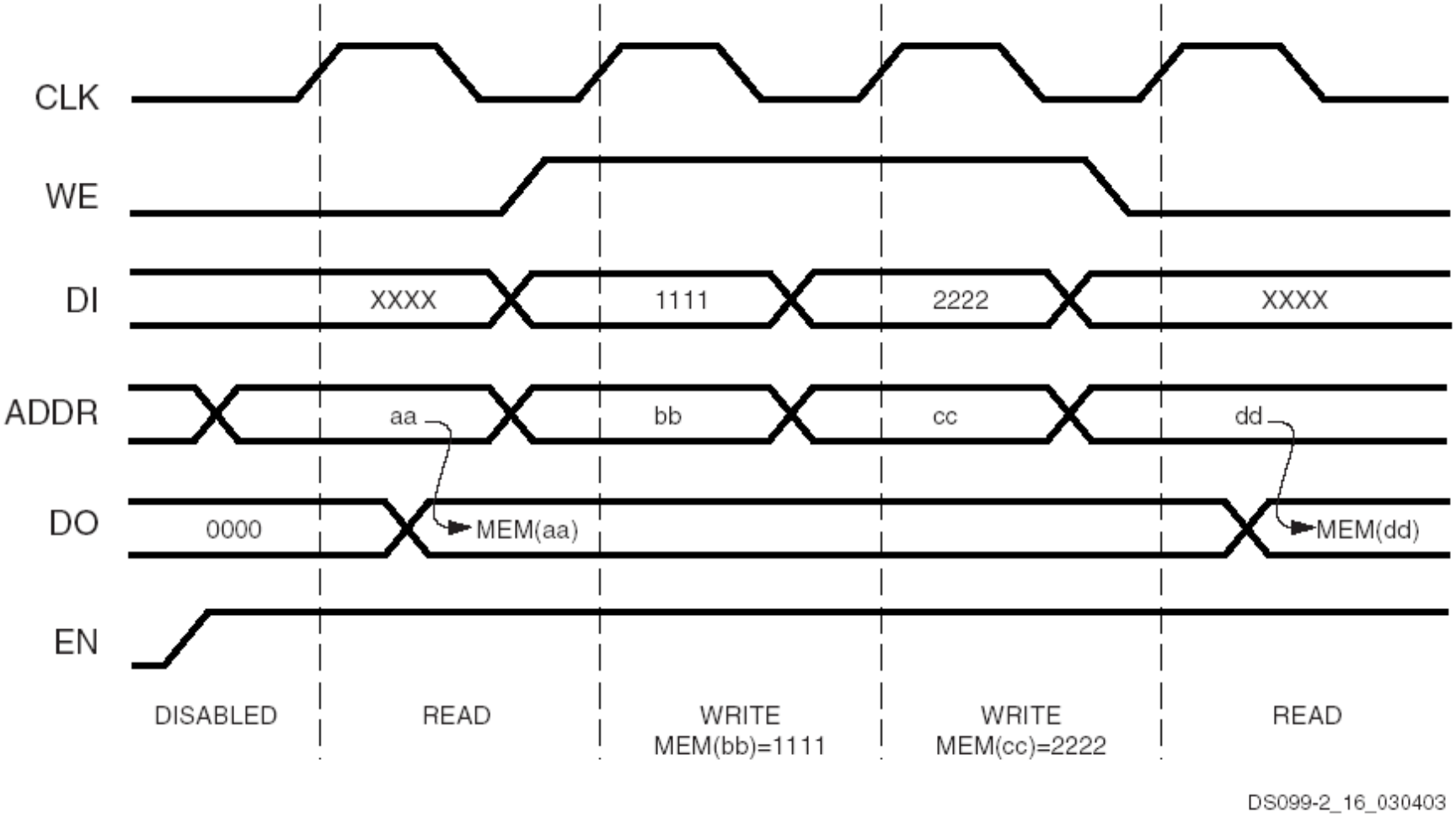
Block RAM with Sync. Read (Write-First Mode)



Block RAM with Sync. Read (Write-First Mode)

```
process (clk)
begin
   if rising_edge(clk) 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 process;
```

Block RAM with Sync. Read (No-Change Mode)



Block RAM with Sync. Read (No-Change Mode)

```
process (clk)
begin
   if rising_edge(clk) 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 process;
```

Block RAM Initialization

Example 1

```
type ram_type is array (0 to 127) of std_logic_vector(15 downto 0);
signal RAM : ram_type := (others => ”0000111100110101”;
```

Example 2

```
type ram_type is array (0 to 127) of std_logic_vector(15 downto 0);
signal RAM : ram_type := (others => (others => ‘1’));
```

Example 3

```
type ram_type is array (0 to 127) of std_logic_vector(15 downto 0);
signal RAM : ram_type := (196 downto 100 => X”B9B5”,
                          others => X”3344”);
```

Block RAM Initialization from a File

```
type RamType is array(0 to 127) of bit_vector(31 downto 0);

impure function InitRamFromFile (RamFileName : in string) return RamType is
    FILE RamFile : text is in RamFileName;
    variable RamFileLine : line;
    variable RAM : RamType;
begin
    for I in RamType'range loop
        readline (RamFile, RamFileLine);
        read (RamFileLine, RAM(I));
    end loop;
    return RAM;
end function;

signal RAM : RamType := InitRamFromFile("rams_20c.data");
```

rams_20c.data:

use binary or hex, not mixing them

```
00101100010111101111001000010000 1111
10101100011001101010101011010 1110111
...
10101111011100101111100011000 1010000
```

128

number of lines in the file must match the number of rows in memory

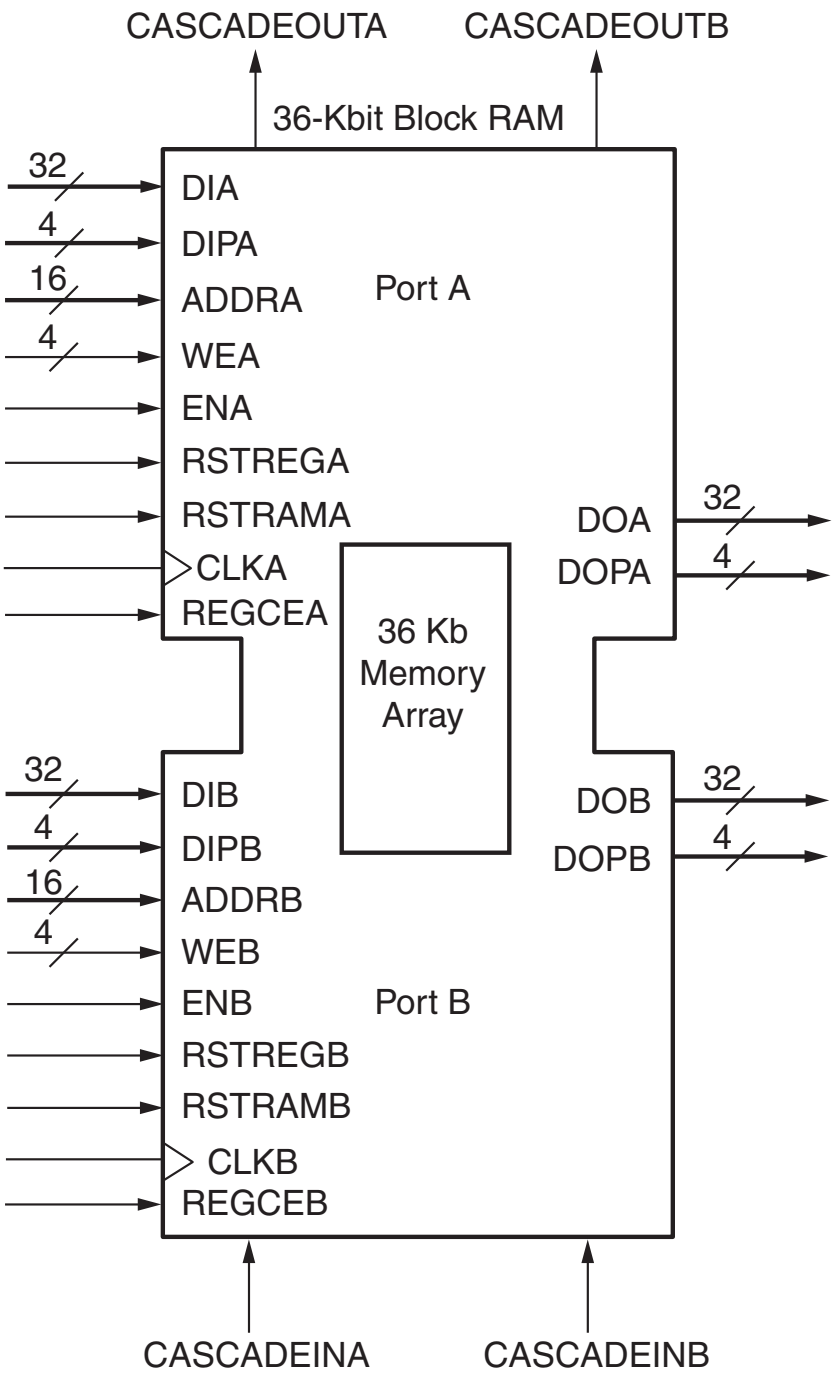
Block RAM Interface True Dual Port

Ports A and B are fully independent.

Each port has its own address, data in/out, clock, and WR enable.

Both read/write are synchronous.

Simultaneously writing to the same address causes data uncertainty.



Dual-Port Block RAM

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rams_16b is
    port(clka  : in std_logic;
         clkb  : in std_logic;
         ena   : in std_logic;
         enb   : in std_logic;
         wea   : in std_logic;
         web   : in std_logic;
         addra : in std_logic_vector(6 downto 0);
         addrb : in std_logic_vector(6 downto 0);
         dia   : in std_logic_vector(15 downto 0);
         dib   : in std_logic_vector(15 downto 0);
         doa   : out std_logic_vector(15 downto 0);
         dob   : out std_logic_vector(15 downto 0));
end rams_16b;
```

Dual-Port Block RAM

```
architecture syn of rams_16b is
    type ram_type is array (127 downto 0) of std_logic_vector(15 downto 0);
    shared variable RAM : ram_type;
begin

    process (CLKA)
    begin
        if CLKA'event and CLKA = '1' then
            if ENA = '1' then
                DOA <= RAM(conv_integer(ADDRA));
                if WEA = '1' then
                    RAM(conv_integer(ADDRA)) := DIA;
                end if;
            end if;
        end if;
    end process;

    process (CLKB)
    begin
        if CLKB'event and CLKB = '1' then
            if ENB = '1' then
                DOB <= RAM(conv_integer(ADDRB));
                if WEB = '1' then
                    RAM(conv_integer(ADDRB)) := DIB;
                end if;
            end if;
        end if;
    end process;

end syn;
```

Port A

Port B

Simple Dual-Port BRAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sync_rw_port_ram is
generic(    ADDR_WIDTH : integer := 10;
            DATA_WIDTH : integer := 12);
port (clk : in std_logic;
       we   : in std_logic;
       addr_w, addr_r : in
            std_logic_vector(ADDR_WIDTH-1 downto 0);
       din  : in std_logic_vector(DATA_WIDTH-1 downto 0);
       dout : out std_logic_vector(DATA_WIDTH-1 downto 0);
end sync_rw_port_ram;
```

Simple Dual-Port BRAM

```
architecture beh_arch of sync_rw_port_ram is
   type ram_type i s array (0 to 2** ADDR_WIDTH -1) of
            std_logic_vector(DATA_WIDTH-1 downto 0);
   signal ram : ram_type;
begin
   process(clk)
   begin
      if (clk'event and clk = '1') then
         if (we = '1') then
            ram(to_integer(unsigned(addr_w))) <= din;
         end if;
         dout <= ram(to_integer(unsigned(addr_r)));
      end if;
   end process;
end beh_arch ;
```

Single-Port RAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sync_rw_port_ram is
generic(    ADDR_WIDTH : integer := 10;
            DATA_WIDTH : integer := 12);
port ( clk : in std_logic;
       we   : in std_logic;
       addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
       din  : in std_logic_vector(DATA_WIDTH-1 downto 0);
       dout : out std_logic_vector(DATA_WIDTH-1 downto 0);
end sync_rw_port_ram;
```

Single-Port BRAM

```
architecture beh_arch of sync_rw_port_ram is
   type ram_type i s array (0 to 2** ADDR_WIDTH -1) of
            std_logic_vector(DATA_WIDTH-1 downto 0);
   signal ram : ram_type;
begin
   process(clk)
   begin
      if (clk'event and clk = '1') then
         if (we = '1') then
            ram(to_integer(unsigned(addr))) <= din;
         end if;
         dout <= ram(to_integer(unsigned(addr)));
      end if;
   end process;
end beh_arch ;
```

Design Example - FIFO

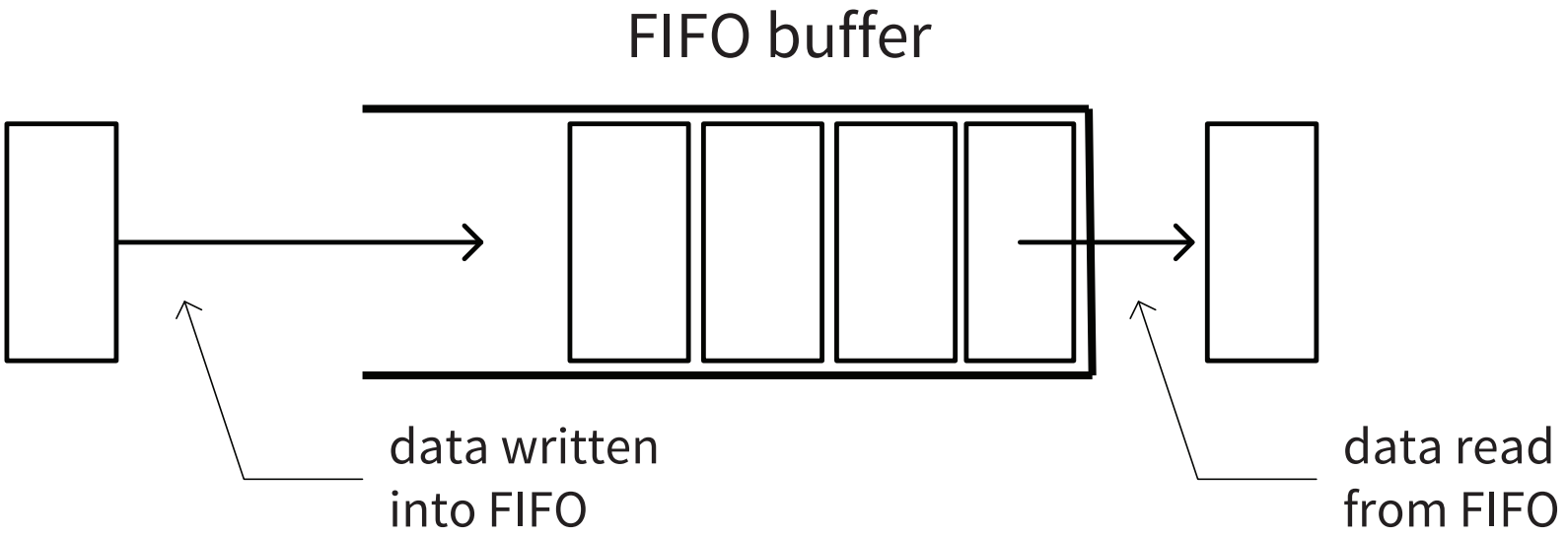


Figure 7.2 Conceptual diagram of a FIFO buffer.

FIFO Design using a circular buffer