FPGA-Based Implementation of RAM with Asymmetric Port Widths for Run-Time Reconfiguration

R. Senhadji-Navarro, I. García-Vargas, G. Jiménez-Moreno, A. Civit-Balcells
Dpto. Arquitectura y Tecnología de Computadores
Universidad de Sevilla, Sevilla (España)
raouf@atc.us.es

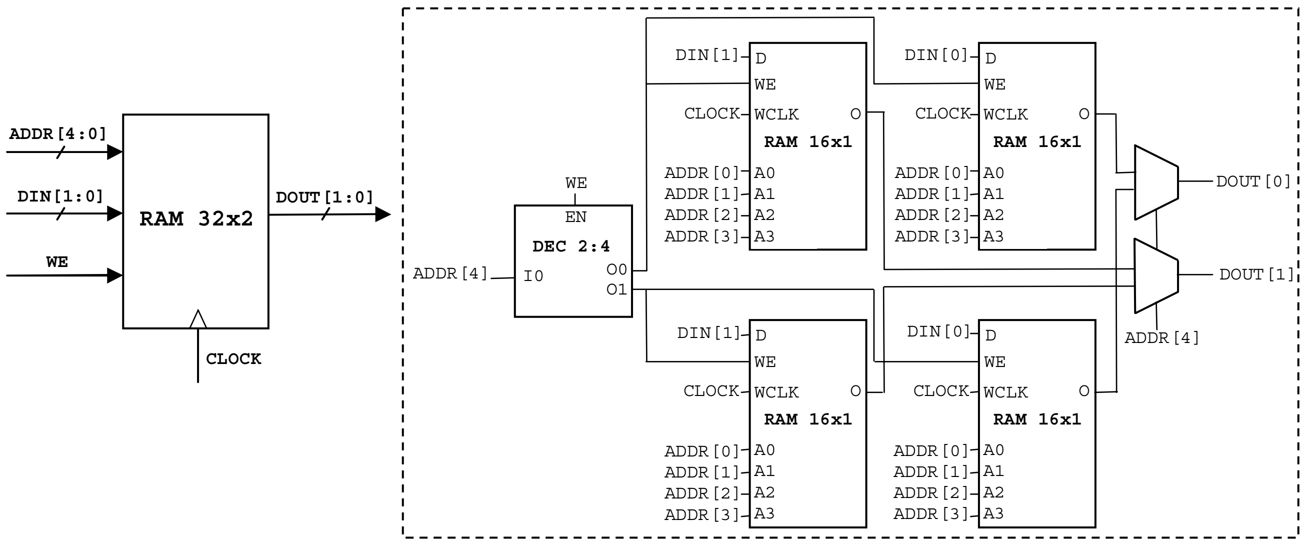
***Abstract*— In this paper, we present a HDL description of a RAM with asymmetric port widths which allows read and write operations with different data size. This RAM is suitable for implementing run-time reconfigurable systems in FPGA. The proposed RAM specification has been tested with different target devices.**

I. Introduction

In spite of the idea of reconfigurable logic is not novel, the growing interest on reconfigurable systems in recent years have been motivated basically by the partial dynamic reconfiguration capabilities of new Field Programmable Gate Array (FPGA) devices [1, 2]. The partial dynamic reconfiguration allows reprogram selected areas of an FPGA after its initial configuration, while the remainder areas continue in operation. This solution presents some disadvantages: the reprogramming cost (e.g. size of reconfiguration data or reconfiguration latency) is high, the reconfiguration depends on the placement and routing of the circuit and the reconfiguration sequences must be generated as a technology-dependent bitstreams. Moreover, not all FPGA devices support dynamic reconfiguration.

An alternative scheme to design reconfigurable systems is to use memory-based implementations. As the functionality of the circuit is defined by the RAM content, it can be changed by reloading the RAM. This alternative saves the technology dependence problem mentioned above due to the independence of the RAM content and its implementation. In the last years, different models of Reconfigurable Finite State Machines (RFSM) using memory-based implementation have been proposed [1, 2, 3]. These RFSM implementations require dual-port capabilities: one read and one write port. The first one is used for normal machine operation and the second one for the reconfiguration process. The width of the read port is determined by the FSM description and depends on the number of outputs and states [4]. However, the write port width depends on the architecture of the reconfigurator system, e.g. a microprocessor, and the strategy of reconfiguration (to modify a state, a transition, an output, etc.). Thus, the data size of read and write operations must be different. The availability of RAM with asymmetric port widths (asymmetric RAM) is useful to implement memory-based reconfigurable systems.

A RAM memory is implemented in FPGA by using smaller memories connected via multiplexors and decoders. Fig. 1 shows an implementation on a Xilinx FPGA of a RAM with 32 words of two bits each using look-up tables (LUT). This manufacturer allows RAM implementation using either RAM blocks or LUTs configured as basic RAM



(a) (b)

Figure 1. LUT-based implementation of a RAM 32x2: (a) block diagram and (b) schematic

This work has been partially supported by the Spanish Government under project TEC2006-11730-C03-02

```
entity asymram is
  generic(
    raw : natural:=13;  -- read address width
    rdw : natural:=4;   -- read data width
    waw : natural:=12;  -- write address width
    wdw : natural:=8;   -- write data width
    baw : natural:=12;  -- basic_ram address width
    bdw : natural:=4;   -- basic_ram data width
    init : bit_vector:="0" ); -- initial ram content
  port(
    clk : in std_logic;
    we : in std_logic;
    raddr : in std_logic_vector(raw-1 downto 0);
    waddr : in std_logic_vector(waw-1 downto 0);
    din : in std_logic_vector(wdw-1  downto 0);
    dout : out std_logic_vector(rdw-1 downto 0));
end asymram;

architecture architectural of asymram is
  component basic_ram ...
  component mux is ...
  component dec is ...
  ...
  type bout_t is array (0 to 2**(raw-baw)*(rdw/bdw)-1)
    of std_logic_vector(bdw-1 downto 0);
  signal bout: bout_t;
  type muxin_t is array (0 to rdw-1)
    of std_logic_vector(2**(raw-baw)-1 downto 0);
  signal muxin : muxin_t;
  signal decout : std_logic_vector(2**(waw-baw)-1 downto 0);
  signal we_i : std_logic_vector(2**(waw-baw)-1 downto 0);
  type init_t is array (0 to (2**(raw-baw))*rdw-1)
    of bit_vector(bdw*2**baw-1 downto 0);
  function initialize_ram return init_t is
    ...
  end function initialize_ram;
  constant init_i : init_t := initialize_ram;
begin
  BRAM0: for i in 0 to 2**(raw-baw)*(rdw/bdw)-1 generate
    we_i(i*bdw/wdw) <= we and decout(i*bdw/wdw);
    BRAM1: basic_ram
      generic map( baw=>baw, bdw=>bdw, init=>init_i(i))
      port map( clk => clk,
        we => we_i(i*bdw/wdw),
        ra => raddr(raw-1 downto raw-baw),
        wa => waddr(waw-1 downto waw-baw),
        di => din( ((i mod (wdw/bdw)) + 1) * bdw - 1
                downto (i mod (wdw/bdw)) * bdw),
        do => bout(i));
  end generate BRAM0;
  MUX0: if raw > baw generate
    MUX1: for j in 0 to rdw-1 generate
      MUX2: mux generic map( raw-baw )
        port map( inputs => muxin(j),
          control => raddr(raw-baw-1 downto 0),
          output => dout(j) );
    end generate MUX1;
    WIRE0: for i in 0 to 2**(raw-baw)*(rdw/bdw)-1 generate
      WIRE1: for j in 0 to bdw-1 generate
        muxin( j+bdw*(i mod (rdw/bdw)) )( i*bdw/rdw )
          <= bout(i)(j);
      end generate WIRE1;
    end generate WIRE0;
  end generate MUX0;
  NULLMUX0: if raw<=baw generate
    NULLMUX1: for i in 0 to rdw/bdw generate
      dout((i+1)*bdw-1 downto i*bdw) <= bout(i);
    end generate NULLMUX1;
  end generate NULLMUX0;
  DEC0: if waw>baw generate
    DEC1: dec generic map( waw-baw )
      port map( inputs => waddr(waw-baw-1 downto 0),
        outputs => decout );
  end generate DEC0;
  NULLDEC0: if waw<=baw generate
    decout <= (others=>'1');
  end generate NULLDEC0;
end architectural;
```

Figure 2. ASYMRAM VHDL description

memories (distributed RAM).

Usually, CAD tools inference RAM memories from HDL description, but the read and write port widths must be equal (symmetric RAM). This kind of memory is widely used in system-on-chip; however, it is not suitable for memory-based reconfigurable architectures. On the other hand, CAD tools allow architectural descriptions or provide GUI facilities to synthesize asymmetric RAMs. The Xilinx CoreGen tool [5] can be used to implement asymmetric RAMs using memory blocks. However, distributed RAM with asymmetric port widths can not be generated.

The use of a specific HDL architectural description for each asymmetric RAM design is tedious and makes difficult its reusability and scalability. On the other hand, the GUI alternative presents some disadvantages: it is not adequate for script-based design flows, and it requires a separate tool to generate and regenerate the RAM. In addition, the technology-dependent implementation generated by tools like CoreGen may complicate the portability to different devices.

To save these problems the authors present a generic VHDL description with the aim of easy the implementation of asymmetric RAMs.

II. Implementation of RAM with asymmetric width ports

The proposed VHDL architectural description (called ASYMRAM) and the schematic are shown in Fig. 2 and Fig. 3, respectively. The ASYMRAM component can be used as symmetric and asymmetric RAM. The high-level entity consists of one decoder and a set of basic RAM components and multiplexors. The multiplexor and the decoder description use a generic VHDL behaviour description. The basic RAM component depends on the specific device. So, it must be described by the designer for each specific manufacturer using the appropriate primitive or VHDL behaviour description. This component models a dual-port RAM with a configurable depth and width. In order to support ASYMRAM initialization via signal, the basic RAM component must be designed with this feature.

The RAM specification achieves a high degree of portability and flexibility by hiding the specific device architecture. Fig. 4 shows the basic RAM entity (Fig. 4-a) and different architectures for Xilinx FPGA devices: using 4-input LUTs (Fig. 4-b), 6-input LUTs available in new Virtex-5 devices (Fig. 4-c), and SelectRAM blocks (Fig. 4-d). Each case uses the appropriate available resource.

III. Experimental results

With the aim of showing the feasibility and the usefulness of ASYMRAM, it has been implemented using Xilinx ISE 7.1. We have tested ASYMRAM configured as both symmetric and asymmetric RAM with different target devices.

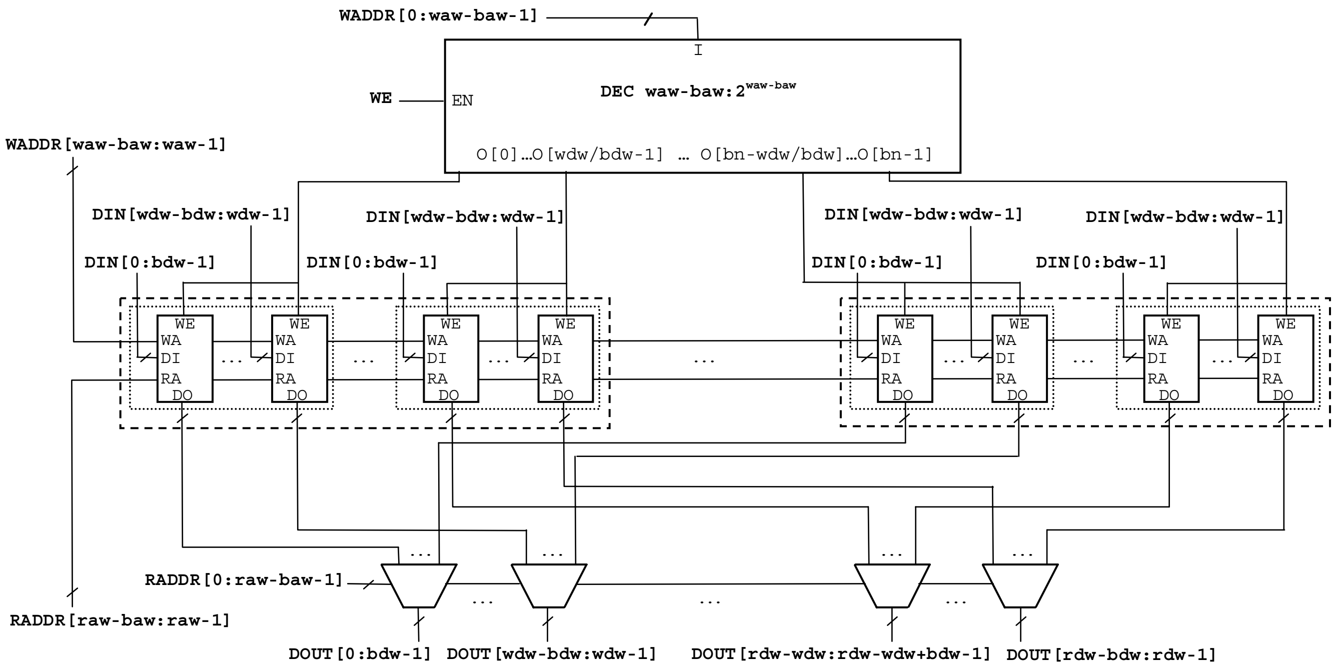


Figure 3. ASYMRAM schematic

As Xilinx do not provide HDL templates to infer asymmetric RAM [6], only symmetric RAMs with different depth have been synthesized in order to compare ASYMRAM with Xilinx HDL templates. For all test cases, the maximum operation frequency is the same in both implementations. However, the number of resources used differs: ASYMRAM spents less number of LUTs than the Xilinx HDL template for RAMs with more than 128 words (Fig. 5). These differences are due to the decoding logic implementation. Fig. 6 shows the VHDL code of ASYMRAM decoder.

IV. CONCLUSIONS

This paper presents a VHDL description of asymmetric RAM. The lack of VHDL templates or library components to generate asymmetric memories makes difficult its use in FPGA-based designs. This is especially interesting to develop memory-based reconfigurable systems. The asymmetric RAM has been designed to allow the portability to different devices and the use of different memory resources. The feasibility and the correctness of the proposed specification have been tested with different target devices.

REFERENCES

[1] M. Boden, M. Koegst, J.L. Tiburcio-Badia, S. Rulke, ‘Cost-Efficient Implementation of Adaptive Finite States Machines’, Proceedings of the EUROMICRO Symposium on Digital System Design (DSD’04), pp. 144-151, Rennes (France), 2004

[2] V. Sklyarov, ‘Reconfigurable models of finite state machines and their implementation in FPGAs’. Journal of Systems Architecture, 47, 2002, pp. 1043-1064.

[3] J. Teich, M Köster, ‘(Self-)reconfigurable Finite State Machines: Theory and Implementation’, Proceedings of the International Conference on Design Automation and Test in Europe (DATE’02), pp 559-567, Paris (France), March, 2002

[4] R. Senhadji-Navarro, I. García-Vargas, G. Jimenez-Moreno and A. Civit-Ballcels ‘ROM-based FSM implementation using input multiplexing’, Electronics Letters, Vol. 40, N. 20, September 2004

[5] Xilinx, Inc. (www.xilinx.com)

[6] Xilinx, Inc.: ‘XST User Guide’

```vhdl
entity basic_ram is
  generic( baw: natural:=4;
      bdw: natural:=1;
      init: bit_vector:="0000000000000000" );
  port( clk: in std_logic;
      we: in std_logic;
      ra: in std_logic_vector( baw-1 downto 0 );
      wa: in std_logic_vector( baw-1 downto 0 );
      di: in std_logic_vector( bdw-1 downto 0 );
      do: out std_logic_vector( bdw-1 downto 0 ));
end basic_ram;
```

(a)

```vhdl
architecture architectural of basic_ram is
begin
  RAM16X1D_inst : RAM16X1D
     generic map (
        INIT => init )
     port map (
        DPO => do(0),     -- Port B 1-bit data output
        A0 => wa(0),      -- Port A address[0] input bit
        ...
        A3 => wa(3),      -- Port A address[3] input bit
        D => di(0),       -- Port A 1-bit data input
        DPRA0 => ra(0),   -- Port B address[0] input bit
        ...
        DPRA3 => ra(3),   -- Port B address[3] input bit
        WCLK => clk,      -- Port A write clock input
        WE => we );       -- Port A write enable input
end architectural;
```

(b)

```vhdl
architecture architectural of basic_ram is
begin
  RAM64X1D_inst : RAM64X1D
     generic map (
        INIT => init )
     port map (
        DPO => do(0),     -- Port B 1-bit data output
        A0 => wa(0),      -- Port A address[0] input bit
        ...
        A4 => wa(4),      -- Port A address[4] input bit
        D => di(0),       -- Port A 1-bit data input
        DPRA0 => ra(0),   -- Port B address[0] input bit
        ...
        DPRA4 => ra(4),   -- Port B address[4] input bit
        WCLK => clk,      -- Port A write clock input
        WE => we );       -- Port A write enable input
end architectural;
```

(c)

```vhdl
architecture architectural of basic_ram is
begin
  RAMB16_S4_S4_inst : RAMB16_S4_S4
     generic map (
        INIT_00 => init( 255 downto 0 ),
        INIT_01 => init( 511 downto 256 ),
        ... )
    port map (
        DOA => do,        -- Port A 4-bit Data Output
        ADDRA => ra,      -- Port A 12-bit Address Input
        ADDRB => wa,      -- Port B 12-bit Address Input
        CLKA => clk,      -- Port A Clock
        CLKB => clk,      -- Port B Clock
        DIA => "0000",    -- Port A 4-bit Data Input
        DIB => di,        -- Port B 4-bit Data Input
        ENA => '1',       -- Port A RAM Enable Input
        ENB => '1',       -- Port B RAM Enable Input
        SSRA => '0',      -- Port A Synchronous Set/Reset Input
        SSRB => '0',      -- Port B Synchronous Set/Reset Input
        WEA => '0',       -- Port A Write Enable Input
        WEB => we );      -- Port B Write Enable Input
end architectural;
```

(d)

Figure 4. Basic RAM component description: (a) VHDL entity, (b) 4-LUT, (c) 6-LUT, and (d) SelectRAM 4Kx4

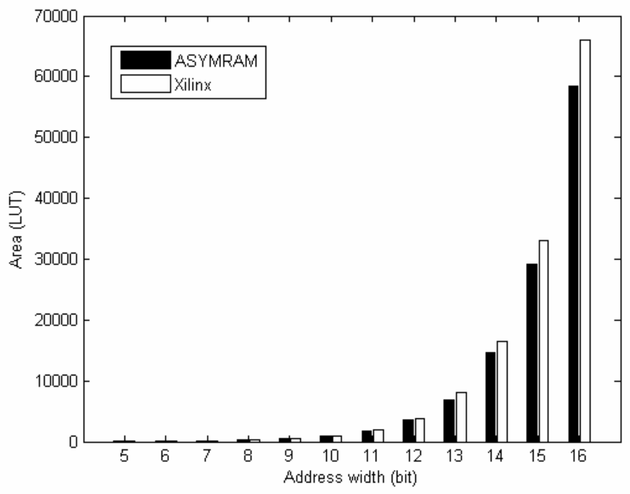


Figure 5. Area occupation of distributed RAM 2^n×8: ASYMRAM vs Xilinx HDL template

```vhdl
entity dec is
  generic( n: natural:=2 );
  port(
    inputs: in std_logic_vector(n-1 downto 0);
    outputs: out std_logic_vector(2**n-1 downto 0));
end dec;

architecture behavioral of dec is

  function decode(inputs : std_logic_vector(n-1 downto 0))
  return std_logic_vector is
    variable outputs : std_logic_vector(2**n-1 downto 0);
  begin
    outputs := (others=>'0');
    outputs( conv_integer(inputs) ) := '1';
    return outputs;
 end decode;

begin
  outputs <= decode(inputs);
end behavioral;
```

Figure 6. ASYMRAM decoder VHDL description