Memory Controller on FPGA using LALP (High level language for FPGA)

Karan Bhandari, CECS 530

**Abstract**— In order to interact with the on board memory it is crucial to have a memory controller on FPGA which is responsible for address encoding/decoding and data encoding/ decoding. It ought to have the capability to set and unset certain bits which are specific to board manufacturers. Conventionally the memory module on most board is implemented on VHDL which is further synthesized and processed to bit file. But here we use the aid of LALP, a high level language for writing highly parallelized FPGA logic using aggressive pipelining. LALP allows us to code FPGA faster using familiar C like constructs and lowers learning curve. LALP is proved to be better than their commercial counterparts to write high level language. In this paper LALP code for memory controller is provided. A sample implementation is presented, an implementation of memory controller on Digilent NEXYS2 in order for the processor (PICOBLAZE) to talk to the SDRAM manufactured by Micron. This paper further enlightens us with a method to statically schedule SDRAM controllers for specific applications by exploiting constrained local search and also provides a methodology for synthesis.

**Index Terms**—Memory Controller, LALP, SDRAM controller

xxxx-xxxx/0x/$xx.00 © 2010 CECS 530

————————————————

1. Karan Bhandari is with the California State University, Long Beach, CA 90804. E-mail: karan.bhandari@student.csulb.edu

*Manuscript received (October, 2011).*

—————————— ◆ ——————————

# 1 Introduction

T

HIS document explains the implementation of memory controller for FPGA board using an efficient and simple high level programming language termed as LALP. It uses aggressive pipelining in order to combat the drawbacks of sequential coding in conventional languages like C and C++.

Prior to the advent of LALP, commercial softwares that converted higher level programs to verilog/VHDL code were not able to completely exploit the parallel nature of FPGA.

**The implementation of memory controller on FPGA is discussed in this paper using LALP.**

The LALP was created so that FPGA designers could write inherently parallel FPGA by using aggressively pipelined setup. The programmer has more controllers of clock cycles and hardware sections despite having a high level language contruct. This allows programmers to write code as a quicker pace and provide an opportunity to implement popular algorithms like Dijkstra’s algorithm.

In this project the memory we are controlling is Micron residing on NEXYS2 by Digilent. It has 16bit bidirectional data bus. Memory controller acts an interface between SDRAM and its master – the processor.

# 2 Description of Memory module

## 2.1 Overall description

## In a typical FPGA board like Diligent NEXYS2 when an asynchronous SRAM (manufactured by Micron) is operated upon, the Cellular RAM automatically refreshes its internal DRAM arrays. Due to this inherent simplicity refreshing the Refresh Configuration Register (RCR), Device ID Register (DIDR) and Bus Configuration Register (BCR) is not required. MIB (Memory Interface Block) unit and external RAM interface in the fashion depicted in Fig 1.

MIB\_unit

Processor

M

SDRAM

Fig 1. Processor, MIB and multiplexer

The processor is the master that sends instructions and transmits/receives data that the SDRAM transmits to the processor through the memory controller. The reference processor that we take into consideration is PICOBLAZE. PICOBLAZE has capability to send and receive 8bits of data, but the SDRAM in our reference board has a 16bit bidirectional data pin. So the MIB and Multiplexer helps the processor to overcome the insufficient data bandwidth. The pin out of MIB is depicted in figure 2.

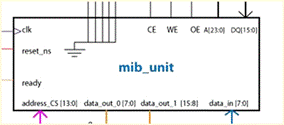


Fig 2. MIB (Memory interface block)

## 2.2 MIB

Internally the core of the MIB is a state machine which is brandished by internal registers that is memory mapped to the processor via the decoder. There are three address registers which are loaded one by one before we commence a read or write operation. Figure 3 depicts how the address and data registers are connected. Figure 4 walks us through the various states of the memory interface block.

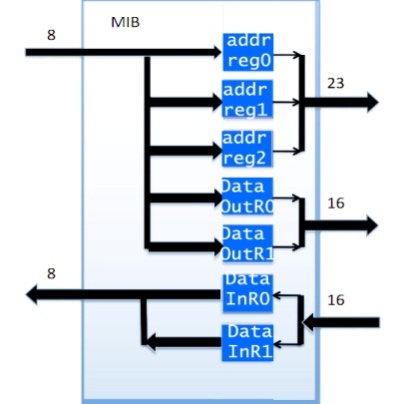


Fig 3. Address and data registers in MIB

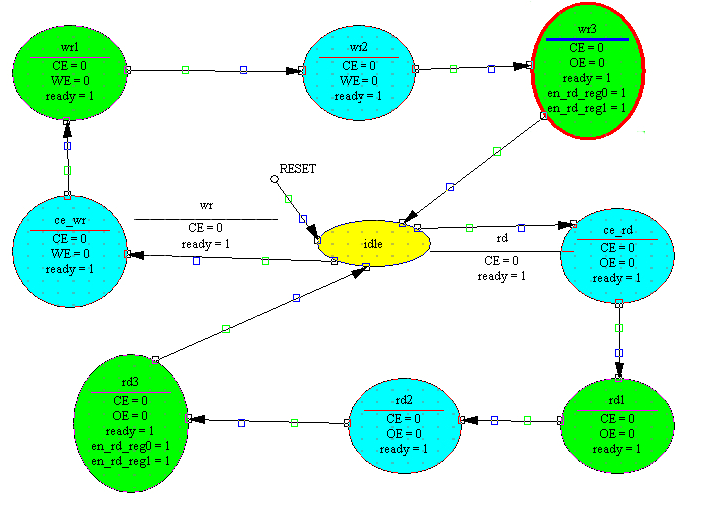


Fig 4. The state diagram of MIB. Upper is write, lower is read

## 2.3 Read and Write Strategies

READ operations are commenced by bringing the pins CE#, OE#, and LB#/UB# LOW while keeping WE# HIGH. Valid data will be driven out of I/Os after the specified access time has elapsed.

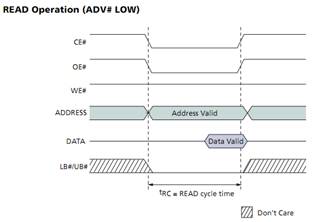


Fig 4. Read Operation Timing Requirement

Write operation occurs when CE#, WE#, and LB#/UB# are set to LOW. During asynchronous Write operations, the OE# level is a "Don't Care," and WE# will override OE#. The data to be written is latched on the rising edge of CE#, WE#, or LB#/UB# (whichever occurs first).

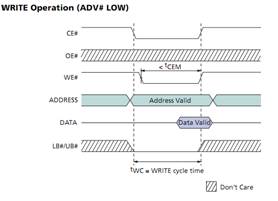


Fig 5. Write Operation Timing Requirement

Note: During asynchronous operation, the CLK input must be held static LOW. The decoder selects Address CS line in sequence 1, 2 and 3 which achieve population of the three address registers one by one through the data\_in line (which is connected to Data Out Port of the processor). Through the same data\_in line we populate DATA OUT registers and we write to SDRAM.

TABLE 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PORT ID** |  |  |  |  |  |
| **(A[0] - A[3])** | **RD** | **WR** | **IN** | **OUT** | **Purpose** |
| 0001 |  |  |  |  | Write to Address Reg 0 [7:0] of 23 bits |
| 0010 |  |  |  |  | Write to Address Reg 1 [15:8] of 23 bits |
| 0011 |  |  |  |  | Write to Address Reg 2 [23:16] of 23 bits |
| 0100 |  |  |  |  | Write to Data Out Reg 0 [3:0] of 8 bits |
| 0101 |  |  |  |  | Write to Data Out Reg 1 [7:4] of 8 bits |
| 0110 |  |  |  |  | Read from Data In Reg 0 [3:0] of 8 bits |
| 0111 |  |  |  |  | Read from Data In Reg 1 [7:4] of 8 bits |

# 3. Implementation

## 3.1 Verilog implementation

module FSM\_MIB(

input clk,

input reset\_ns,

input wr,

input rd,

output reg ready,

output reg CE,

output reg OE,

output reg WE,

output reg en\_rd\_reg0,

output reg en\_rd\_reg1

);

localparam [3:0]

idle = 4'b0000,

ce\_wr = 4'b0001,

ce\_rd = 4'b0010,

wr1 = 4'b0011,

wr2 = 4'b0100,

wr3 = 4'b0101,

rd1 = 4'b0110,

rd2 = 4'b0111,

rd3 = 4'b1000;

// signal declaration

reg [3:0] state\_next, state\_reg;

always @(posedge clk, negedge reset\_ns)

if (!reset\_ns)

state\_reg <= idle;

else

state\_reg <= state\_next;

// next\_state logic and output logic

always @\*

begin

state\_next = state\_reg; //default state

CE = 1'b1;

OE = 1'b1;

WE = 1'b1;

ready = 1'b0;

en\_rd\_reg0 = 1'b0;

en\_rd\_reg1 = 1'b0;

case(state\_reg)

idle:begin

if(wr)

begin

CE = 1'b0;

ready = 1'b1;

state\_next = ce\_wr;

end

else if (rd)

begin

CE = 1'b0;

ready = 1'b1;

state\_next = ce\_rd;

end

end

ce\_wr:begin

CE = 1'b0;

WE = 1'b0;

ready = 1'b1;

state\_next = wr1;

end

ce\_rd:begin

CE = 1'b0;

OE = 1'b0;

ready = 1'b1;

state\_next = rd1;

end

wr1:begin

.

.

.

//other states

## 3.2 Equivalent LALP implementation

const DATA\_WIDTH = 16;

const HALF\_DATA\_WIDTH = 8;

typedef fixed(DATA\_WIDTH, 1) int;

typedef fixed(1, 0) bit;

MIB\_unit(in clk,in rst,in wr,in rd,out bit ready,out bit CE,out bit OE,out bit WE,out bit enable\_read,out bit enable\_write)

{

int state\_nxt;

int state\_reg;

counter(i=0;state\_nxt!=rst;i++@100);

state\_reg=idle;

state\_nxt=state\_reg;

CE=0; when (state\_reg==idle & wr==1) & i.step@2;

ready=1; when (state\_reg==idle & wr==1) & i.step@3;

state\_nxt=enable\_write; when (state\_reg==idle & wr==1) & i.step@4;

CE=0 when (state\_reg==idle & rd==1) & i.step@2;

CE = 0 when (state\_reg==enable\_write & wr==1) & i.step@7;

WE = 0 when (state\_reg==enable\_write & wr==1) & i.step@8;

ready = 1 when (state\_reg==enable\_write & wr==1) & i.step@9;

state\_next = wr1 when (state\_reg==3 & wr==1) & i.step@10;

//similar for read

//process for state when write is between 3-5 and read is between 6-8

}

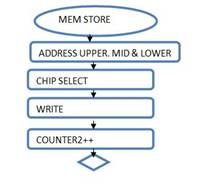


Figure 6. Memory Write Flow Chart

LALP allows us to implement loops by using counter statement. It behaves as for loop with sync qualifiers. At the step we mention the number of clock cycles so that each loop executes distinctly. LALP gives the programmer the reigns of dependency and hazard control since this inconvenience is a tradeoff it makes to achieve efficiency in parallelism. But the programmer can siphon out the flaws of synchronization with the aid of visualization tools like GraphvIz.

The operations are not implemented along the lines of state diagrams. LALP also allows hardware sharing and shift registers and multiplexs.

The memory interface block helps us to extenuate the hassles associated with uneven data bus width. The bits instead of being lost are stored in registers instead. The processor which acts as the master sets the bits to enable storing or reading from the registers.

# 4 Execution

## 4.1 Command line

Create a new environment varaiable LALP\_HOME which points to the directory containing LALP files.

Add LALP to PATH of the system.

Type

“lalp –vh <CodeFile.alp>” to generate vhdl code and use –do option to generate both hardware and software files.

## 4.2 Graphical user

Open Eclipse development IDE, click on File> Import> Import new project. Navigate to the LALP directory and import the eclipse project. Now right click the project, click on project and select Java build path. Add three libraries- JUNG, colt and common collections. Now go to run configuration and set the main project as ALPGUI. Click on File>open, select an existing ALP file and click on compile

# 6 Conclusion

Higher level, simple, yet parallized version of memory controller using LALP is implementable. Memory controller is imperative to coordinate between the processor and SD memory block on FPGA boards.

**Acknowledgment**

I would like to thank Group 5 of CECS 530 for their untiring support and help. We would like to thank the Professor to consider us capable enough to undertake the topic. Special thanks to the creator Mr. Recardo for exhending his helping hald.

**References**

1. R. Menotti, E. Marques, and J. M. P. Cardoso, “Aggressive

Loop Pipelining for Reconfigurable Architectures,” in Field

Programmable Logic and Applications, 2007. FPL 2007.

1. Menotti, R.; Cardoso, J.M.P.; Fernandes, M.M.; Marques, E.; , "LALP: A Novel Language to Program Custom FPGA-Based Architectures," Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on , vol., no., pp.3-10, 28-31 Oct. 2009

doi: 10.1109/SBAC-PAD.2009.23

1. Bayliss, S.; Constantinides, G.A.; , "Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search," Field-Programmable Technology, 2009. FPT 2009. International Conference on , vol., no., pp.304-307, 9-11 Dec. 2009

doi: 10.1109/FPT.2009.5377664

1. Digilent Nexys2 Board Reference Manual

**Karan Bhandari** –Currently enrolled as a student pursuing masters in computer engineering from California State University, Long Beach. To glean more about him please visit http://tinyurl.com/karanbhandari