Theory:

As mentioned in the abstract we used basic digital components to make the controller. The working is explained below along with the components used in detail.

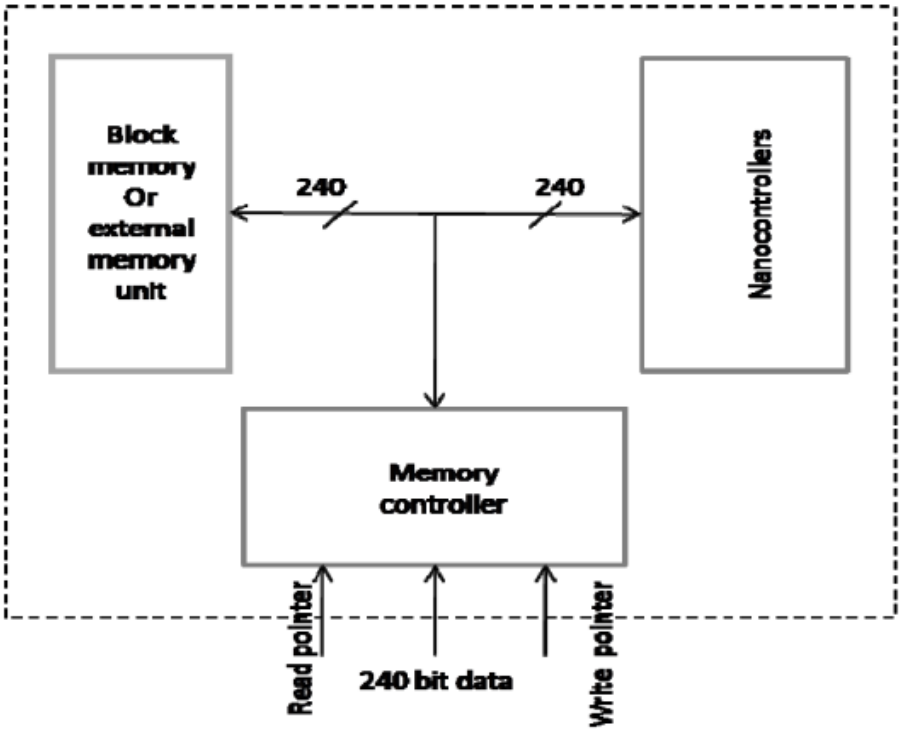
Functional units:

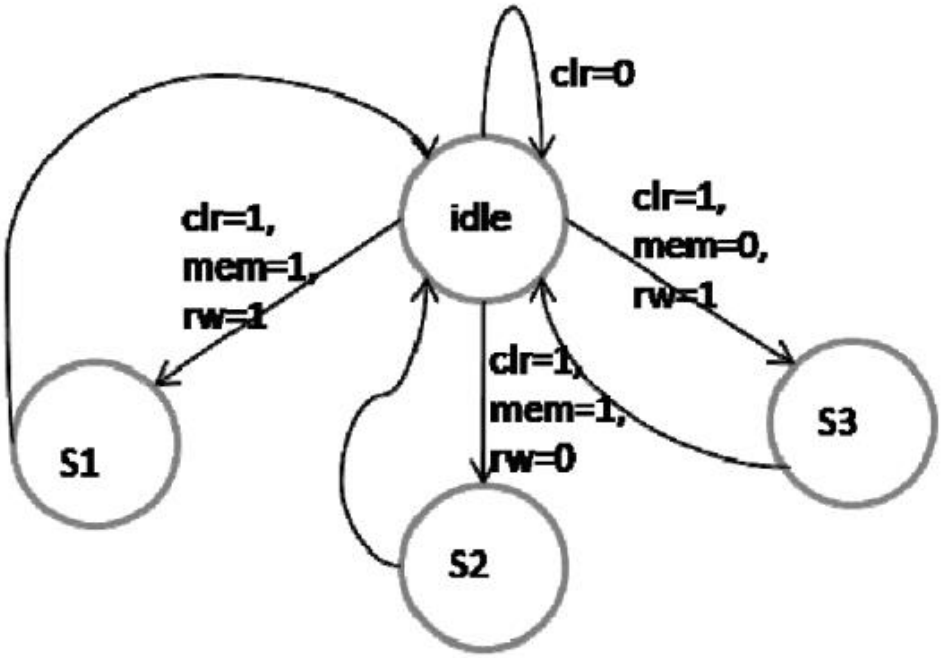
1. memory controller

This is a very a simple sdram controller which works on the De0 Nano. The project also contains a simple push button interface for testing on the dev board.

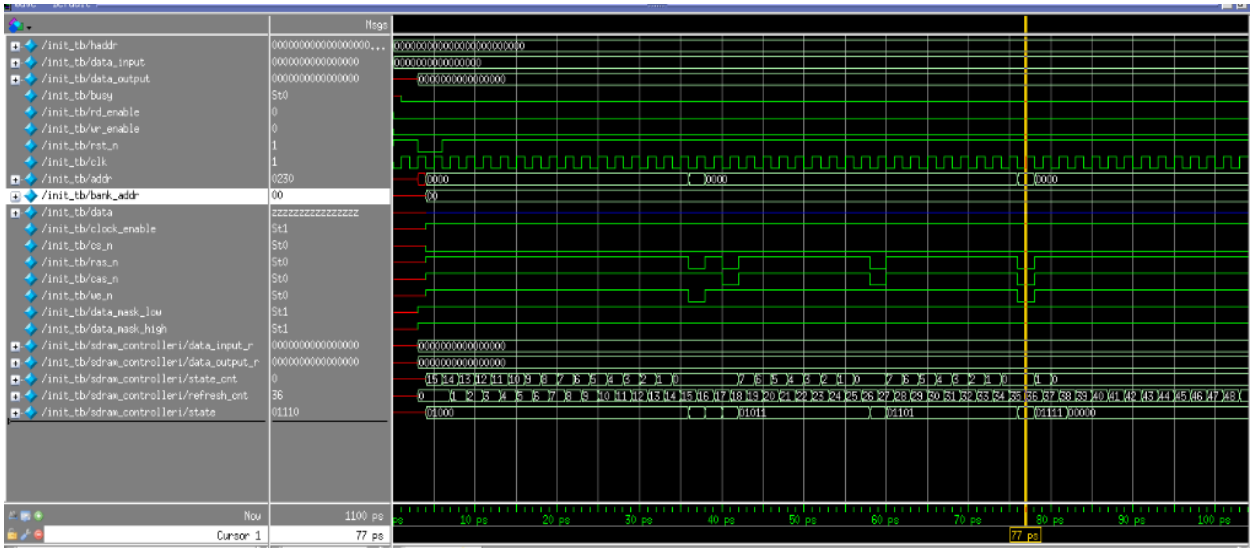
Basic features

- Operates at 100Mhz, CAS 3, 32MB, 16-bit data
- On reset will go into INIT sequnce
- After INIT the controller sits in IDLE waiting for REFRESH, READ or WRITE
- REFRESH operations are spaced evenly 8192 times every 32ms
- READ is always single read with auto precharge
- WRITE is always single write with auto precharge


Block memory Or external memory unit
240
240
Nanocontrollers
Memory controller
Read pointer
240 bit data
Write pointer


clr=0
idle
clr=1,
mem=1,
rw=1
clr=1,
mem=0,
rw=1
clr=1,
mem=1,
rw=0
S1
S2
S3

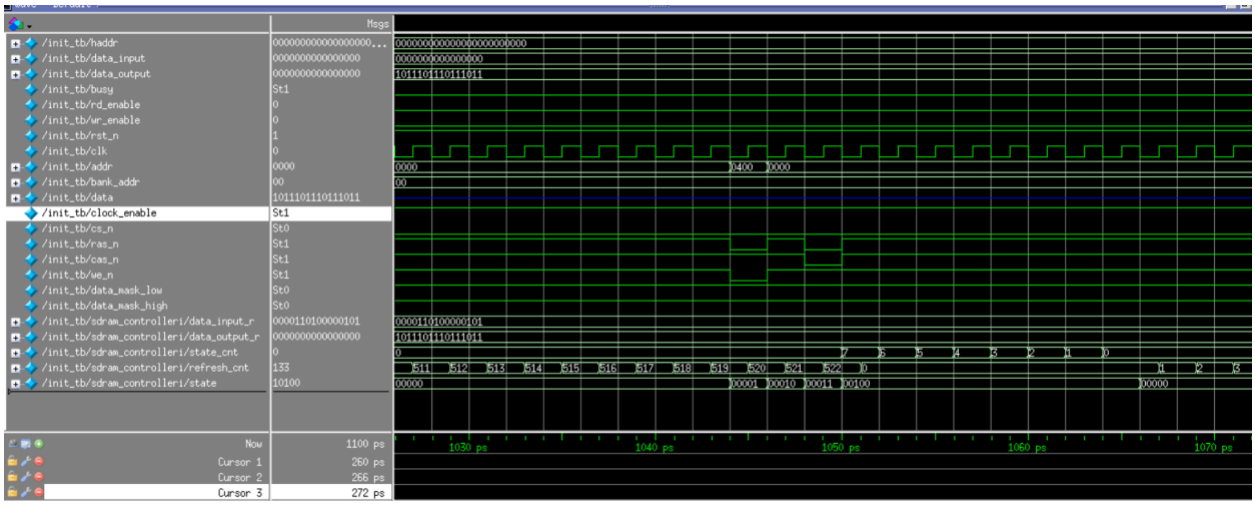
Initialization:



Initialization process showing:

- Precharge all banks
- 2 refresh cycles
- Mode programming

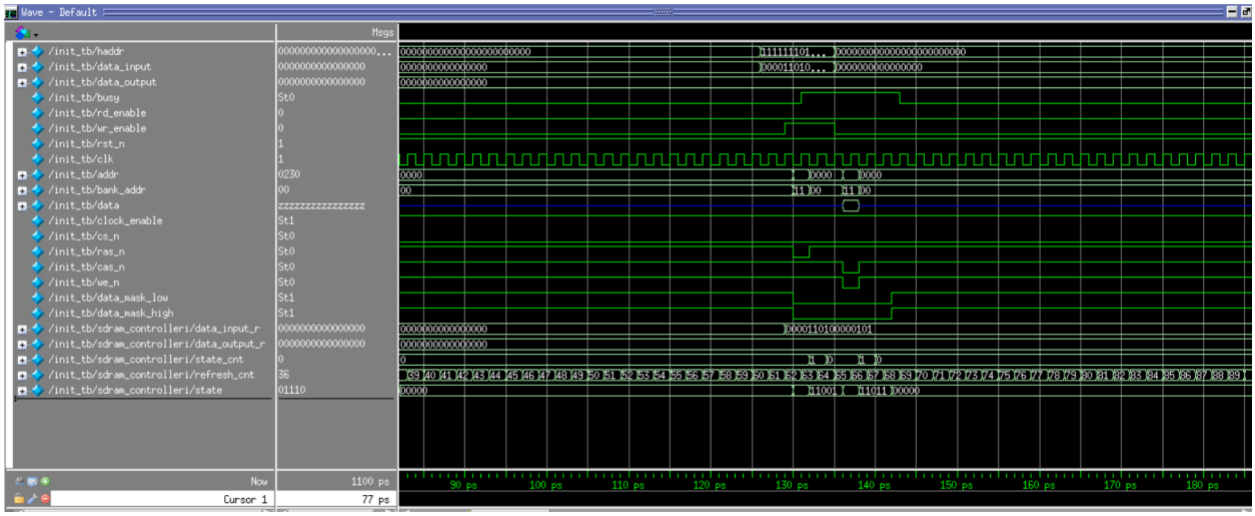
Refresh:



Refresh process showing:

- Precharge all banks
- Single Refresh
- All banks must be precharged when a refresh command is issued
- A DDR2 memory needs to be refreshed onces every 64ns

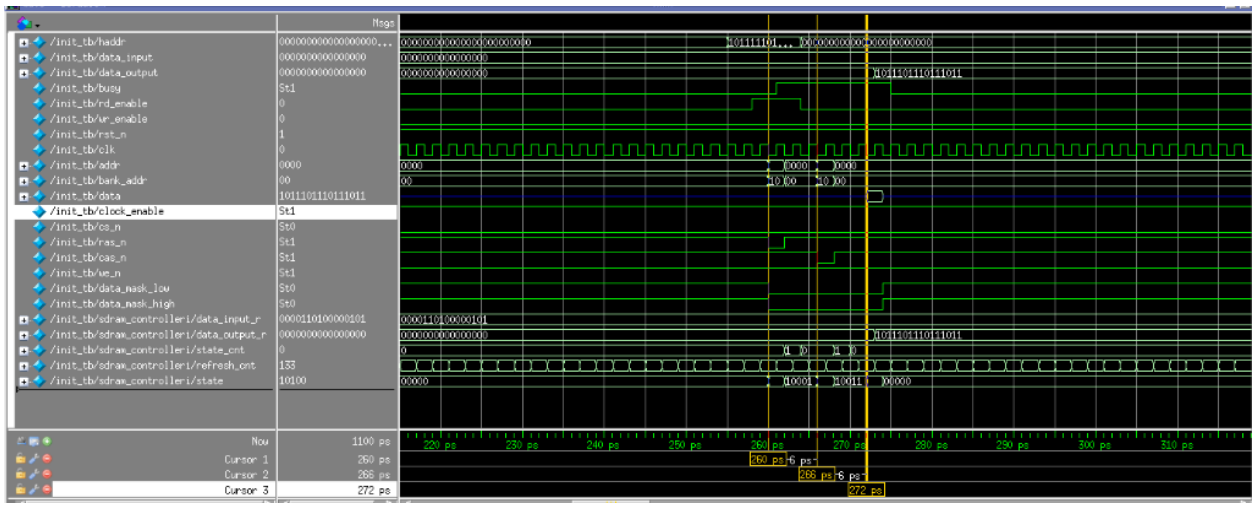
Writes:



Write operation showing:

- Bank Activation & Row Address Strobe
- Column Address Strobe with Auto Precharge set and Data on bus

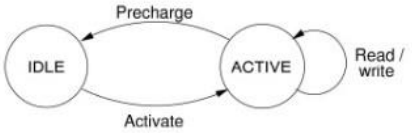
Reads:



Read operation showing:

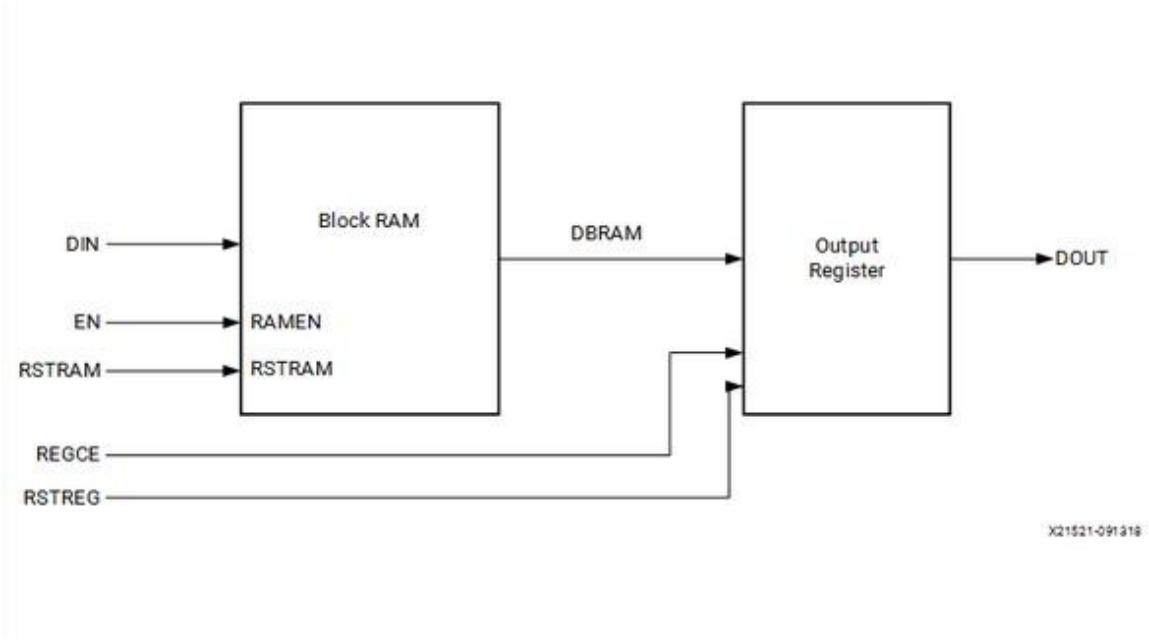
- Bank Activation & Row Address Strobe
- Column Address Strobe with Auto Precharge set
- Data on bus

SDRAM command summary



No operation	NOP	Ignores all inputs
Activate	ACT	Activate a row in a particular bank
Read	RD	Initiate a read burst to an active row
Write	WR	Initiate a write burst to an active row
Precharge	PRE	Close a row in a particular bank
Refresh	REF	Start a refresh operation

Block RAM:



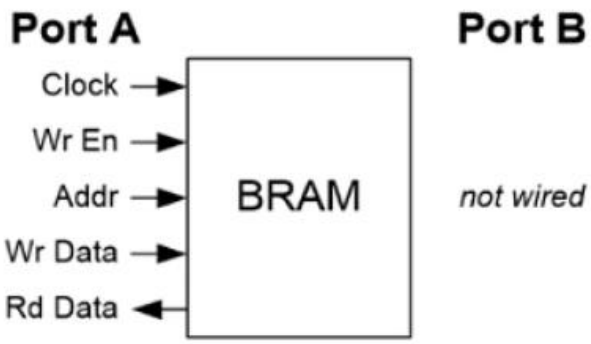
Block RAMs (or BRAM) stands for Block Random Access Memory. Block RAMs are used for storing large amounts of data efficiently inside of

your FPGA like images or video, for high-performance state machines or FIFO buffer, for large shift registers, large look up table or ROMs inside of FPGA. It is a discrete part of FPGA, meaning there are only so many of them available of them on the chip. Usually the bigger and more expensive the FPGA, the more Block RAM it will have on it.

A Block RAM (sometimes called embedded memory, or Embedded Block RAM (EBR)), is a discrete part of an FPGA, meaning there are only so many of them available on the chip. Each FPGA has a different amount, so depending on your application you may need more or less Block RAM. Knowing how much you will need gets easier as you become a better Digital Designer. As I said before, it's used to store "large" amounts of data inside of your FPGA. It's also possible to store data outside of your FPGA, but that would be done with a device like an SRAM, DRAM, EPROM, SD Card, etc.

A BRAM is used for storing large amount of data. A block RAM has width and depth and can be initialized to non-zero value during implimentation.

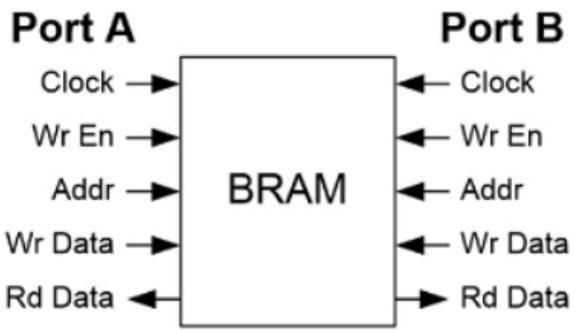
Single port BRAM configuration



The Single Port Block RAM configuration is useful when there is just one interface that needs to retrieve data. This is also the simplest configuration and is useful for some applications. One example would be storing Read-Only Data that is written to a fixed value when the FPGA is programmed. The way they work is all based on a Clock. Data will be read out on the positive edge of the clock cycle at the address specified by Addr as long as Wr En signal is not active. Read values come out on Rd Data, this is the data stored in the BRAM. Note that you can only read

one Rd Data value per clock cycle. So if your Block RAM is 1024 values deep, it will take at least 1024 clock cycles to read the entire thing out.There might be an application where you want to write some data into the Block RAM buffer, then read it out at a later time. This would involve driving Wr En high for one clock cycle and Wr Data would have your write data. For the single port configuration, you can either read or write data on Port A, you can't do both at the same time. If you want to read and write data at the same time, you will need a Dual Port Block RAM!

Dual port BRAM configuration



The Dual Port Block RAM (or DPRAM) configuration behaves exactly the same way as the single port configuration, except you have another port available for reading and writing data. Both Port A and Port B behave exactly the same. Port A can perform a read on Address 0 on the same clock cycle that Port B is writing to address 200. Therefore a DPRAM is able to perform a write on one address while reading from a completely different address. I personally find that I have more use cases for DPRAMs than I do for Single-Port RAMs.

One possible use case would be storing data off of an external device. For example, you want to read data off an SD Card, you could store it in a Dual Port RAM, then read it out later. Or maybe you want to interface to an Analog to Digital Converter (ADC) and will need some place to store the converted ADC values. A DPRAM would be great for this. Additionally, Dual Port RAMs are commonly turned into FIFOs, which

are probably one of the most common use-cases for Block RAM on an FPGA.

Prefetch unit:

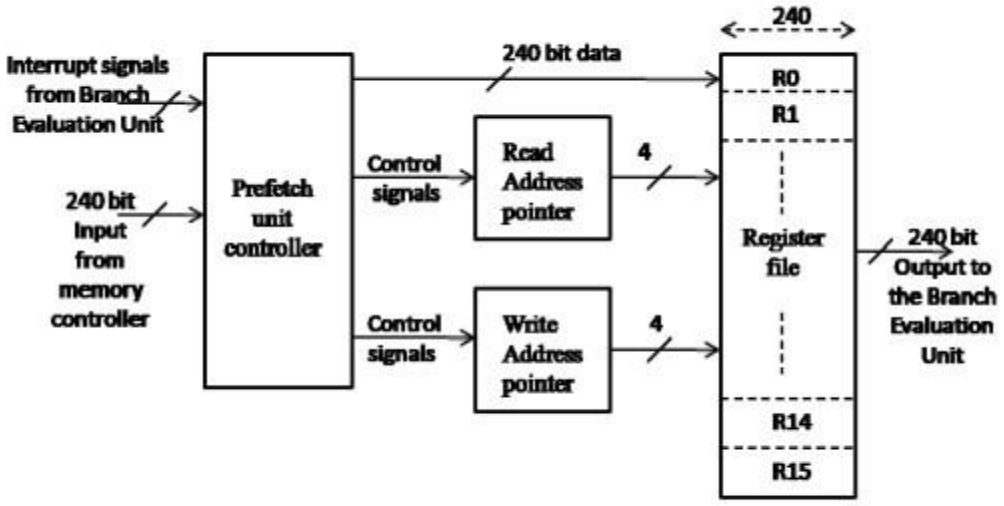
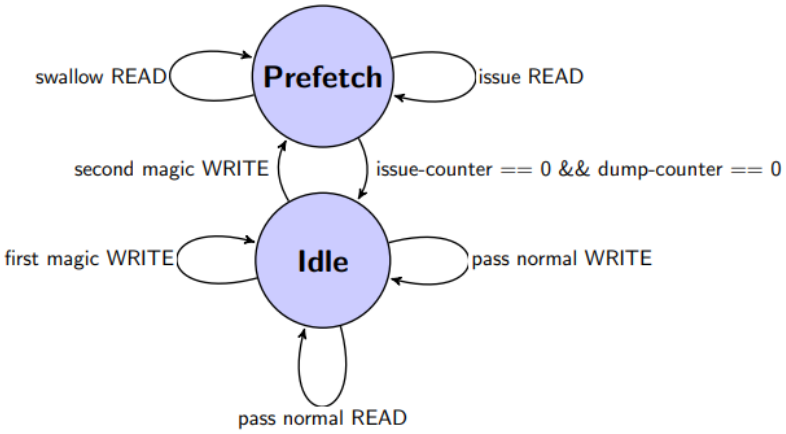


Figure 4.5: Block diagram of prefetch unit

Since the term ”prefetching” comes up repeatedly in this report, its meaning and background shall be briefly discussed, to later understand the difference between common prefetching and the work presented here.In general prefetching describes the process of heuristically loading data in advance, expecting it to be needed in the near future. For this there usually exists a cache, being either a piece of hardware or software, which can store small amounts of data and be accessed very fast (e.g. by a central processing unit (CPU)), compared to the main memory component of the system (e.g. some kind of RAM).In case of a CPU cache, the data in the cache memory is stored with a tag, containing the address of the same data in main memory. When the CPU wants to load data from main memory, it first searches for it inside the cache. If a cache hit happens, meaning the data is found in the cache, the latency between the CPU and main memory was bypassed. Given a cache miss, the data must be loaded from main memory, resulting in the latency becoming visible. CPU caches are used for over three decades, nowadays being split into several levels. Web browser programs similarly use

caches to prefetch the contents of web addresses. To maximize the hit rate of the cache, deciding which data to store in the cache is essential and can be handled in different ways, which are not further discussed. As the prefetching mechanism implemented here does not include a cache, it fundamentally differs from what is explained above. Yet one should notice, that the goal, as well as the effects, are to some degree the same, leading to no misuse of the term ”prefetching”.

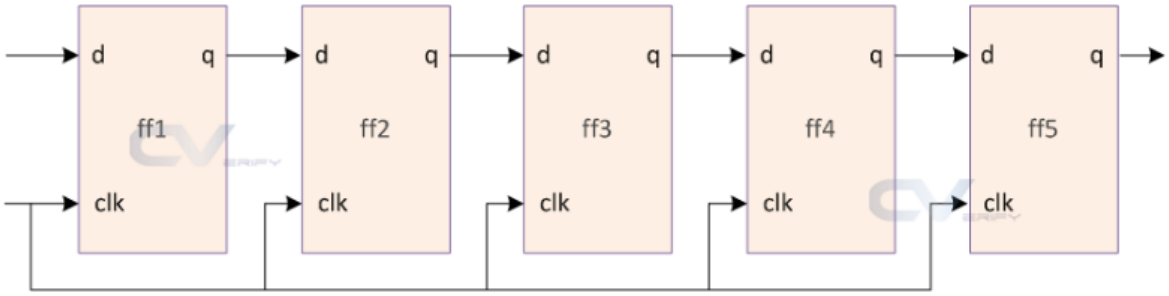


Branch evaluation unit:

This unit has 3 blocks

1. shift register

in digital electronics a shift register is cascade of flipflop where the output pin q of one flip flop coonected to the data input pin(d) of the next because all flops work on the same clock the bit array stored in the shift register will shift by one position.

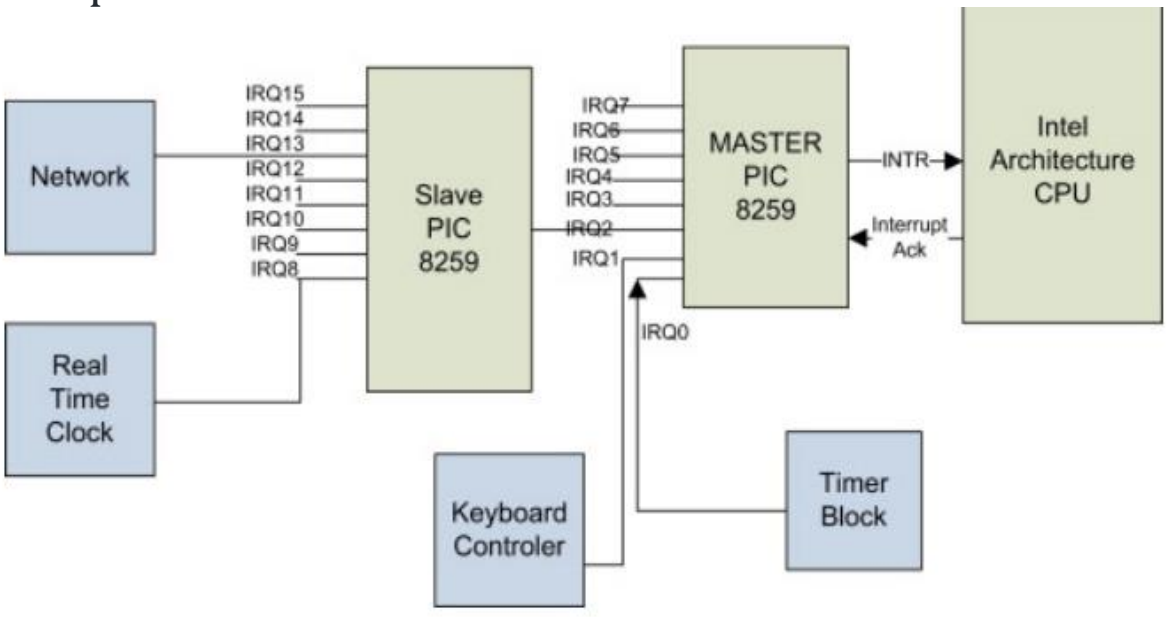


The shift register has five inputs and one n-bit output and the design is parameterized using MSB parameters to signify width of the shift register if n is 4 the it becomes a 4 bit shift register if n is 8 then it becomes 8 bit shift register.

The shift register has few key features:

Can be enabled or disbled by driving en pin of the design
Can shift to the left as well as right when dir is driven
If rstn is pulled low, it will reset the shift register and output will become 0
Input data value of the shift register can be controlled by d pin

2. interrupt controller



The Programmable Interrupt Controller functions as an overall manager in an Interrupt-Driven system. It accept requests from the peripheral equipment, determines which of

the incoming requests is of the highest priority, ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination. Each peripheral device or structure usually has a special program or “routine” that is associated with its specific functional or operational requirements; this is referred to as a “service routine”. The Priority Interrupt Controller, after issuing an interrupt to the CPU, must some how input information into the CPU that can “point” the Program Counter to the service routine associated with the requesting device. This “pointer” is an address in a vectoring table and is referred to as vectoring data.
This Priority Interrupt Controller is a device specifically designed for use in real time, interrupt driven microcomputer systems. It manages eight levels of requests and has built-in features for expandability to other 82C59As that is up to 64 levels. It is programmed by system software as an I/O peripheral. A selection of priority modes is available to the programmer so that the manner in which the requests are processed by the Priority Interrupt Controller (82C59A) can be configured to match system requirements. The priority modes can be changed or reconfigured dynamically at any time during main program operation.

3.SITE buffer

FIFO is an acronym for first in, first out (the first in is the first out), a method for organizing the manipulation of a data structure (often, specifically a data buffer) where the oldest (first) entry, or "head" of the queue, is processed first.Such processing is analogous to servicing people in a queue area on a first-come, first-served (FCFS) basis, i.e. in the same sequence in which they arrive at the queue's tail.

FIFOs are commonly used in electronic circuits for buffering and flow control between hardware and software. In its hardware form, a FIFO primarily consists of a set of read and write pointers, storage and control logic. Storage may be static random access memory (SRAM), flip-flops, latches or any other suitable form of storage. For FIFOs of non-trivial size, a dual-port SRAM is usually used, where one port is

dedicated to writing and the other to reading.A synchronous FIFO is a FIFO where the same clock is used for both reading and writing. An asynchronous FIFO uses different clocks for reading and writing and they can introduce metastability issues. A common implementation of an asynchronous FIFO uses a Gray code (or any unit distance code) for the read and write pointers to ensure reliable flag generation. One further note concerning flag generation is that one must necessarily use pointer arithmetic to generate flags for asynchronous FIFO implementationsA hardware FIFO is used for synchronization purposes. It is often implemented as a circular queue, and thus has two pointers:

*Read pointer / read address register

*Write pointer / write address register

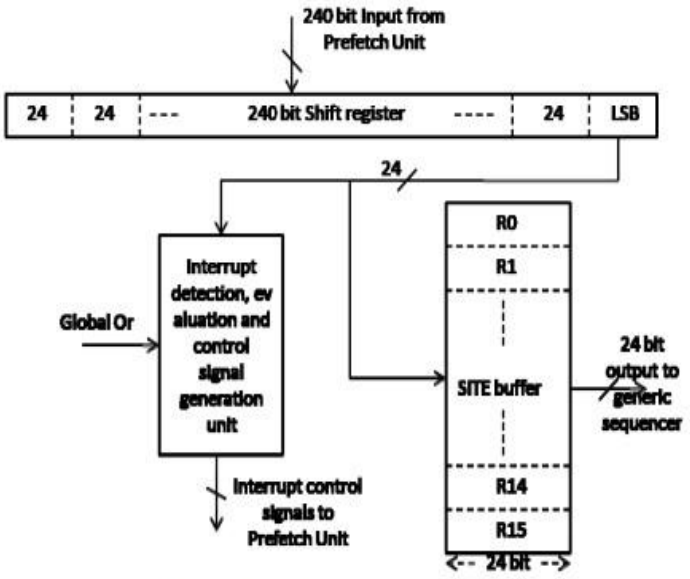


Figure 4.6: Block diagram of branch evaluation Unit

Code:

```
module double_click (  button,  single, double  clk, rst_n);
parameter WAIT_WIDTH = 19;
input  button;
output single, double;
input clk, rst_n;
reg btn_now, btn_last, collect;
reg [2:0] click_cnt;
reg [WAIT_WIDTH-1:0] dbl_click_cnt;
// if we are done counting and we have 1 click its single, else double
assign single = (!dbl_click_cnt & (click_cnt == 3'b001)) ? 1'b1 : 1'b0;
assign double = (!dbl_click_cnt & (click_cnt != 3'b001)) ? 1'b1 : 1'b0;
// detect button down vs button up
wire btn_down = btn_now & ~btn_last;
//wire btn_up =  ~btn_now &  btn_last;
always @ (negedge clk)
 if (~rst_n)
  { btn_last, btn_now } <= 2'b00;
 else
  { btn_last, btn_now } <= { btn_now, button };
// start down counter and count clicks
always @ (posedge clk)
 if (~rst_n)
  begin
  click_cnt <= 3'd0;
  dbl_click_cnt <= {WAIT_WIDTH{1'b1}};
  collect <= 1'b0;
  end
 else
  begin
  if (collect & (dbl_click_cnt != {WAIT_WIDTH{1'b0}}))
```

```
   dbl_click_cnt <= dbl_click_cnt - 1'b1;
  else
   dbl_click_cnt <= dbl_click_cnt;
  if (btn_down)
   begin
   collect <= 1'b1;
   click_cnt <= click_cnt + 1'b1;
   end
  else
   begin
   collect <= collect;
   click_cnt <= click_cnt;
   end
  end
endmodule
```

test bench:

```
module double_click_tb();
vlog_tb_utils vlog_tb_utils0();
 reg button_r;
 reg rst_n, clk;
 wire single, double;
initial
begin
  button_r = 1'b0;
  rst_n = 1'b1;
  clk = 1'b0;
end
always
  #1 clk <= ~clk;
initial
begin
  #3 rst_n = 1'b0;
```

```
 #3 rst_n = 1'b1;

 #10 button_r = 1'b1;
 #100 button_r = 1'b0;
 #3 rst_n = 1'b0;
 #3 rst_n = 1'b1;
 #10 button_r = 1'b1;
 #4 button_r = 1'b0;
 #5 button_r = 1'b1;
 #3 button_r = 1'b0;
 #100 $finish;
end
double_click #(.WAIT_WIDTH(4)) double_clicki (
 .button(button_r), .single(single), .double(double),
 .clk(clk), .rst_n(rst_n)
);

endmodule
```

code:

```
module fifo ( wr_clk,  wr_data,  wr,  full,    rd_data, rd_clk,  rd,  empty_n,
rst_n);
parameter BUS_WIDTH = 16;
input [BUS_WIDTH-1:0]  wr_data;
input              wr_clk;
input              wr;
output              full;     // Low-Means in side can write
output [BUS_WIDTH-1:0] rd_data;
input              rd_clk;
```

```
input              rd;
output              empty_n;   // High-Means out side can read
input              rst_n;
reg [BUS_WIDTH-1:0]    wr_data_r;
reg [BUS_WIDTH-1:0]    rd_data;
reg                wr_r, wr_syn1, wr_syn2, wr_ack1, wr_ack2;
reg                rd_r, rd_syn1, rd_syn2, rd_ack1, rd_ack2;
reg                wr_fifo_cnt;
reg                rd_fifo_cnt;
assign full = wr_fifo_cnt == 1'b1;
assign empty_n = rd_fifo_cnt == 1'b1;
always @ (posedge rd_clk)
 if (~rst_n)
   begin
     rd_fifo_cnt <= 1'b0;
     {rd_ack2, rd_ack1} <= 2'b00;
     {wr_syn2, wr_syn1} <= 2'b00;
   end
 else
   begin
{rd_ack2, rd_ack1} <= {rd_ack1, rd_syn2};
     {wr_syn2, wr_syn1} <= {wr_syn1, wr_r};
     if (rd)
       rd_r <= 1'b1;
     else if (rd_ack2)
       rd_r <= 1'b0;
     if (rd)
       rd_fifo_cnt <= 1'b0;
     if ({wr_syn2, wr_syn1} == 2'b01)
       rd_fifo_cnt <= 1'b1;
     if (wr_syn2)
       rd_data <= wr_data_r;
   end
always @ (posedge wr_clk)
```

```
 if (~rst_n)
  begin
    wr_fifo_cnt <= 1'b0;
    {rd_syn2, rd_syn1} <= 2'b00;
    {wr_ack2, wr_ack1} <= 2'b00;
  end
 else
  begin
   {wr_ack2, wr_ack1} <= {wr_ack1, wr_syn2};
   {rd_syn2, rd_syn1} <= {rd_syn1, rd_r};
   if (wr)
     wr_r <= 1'b1;
   if (wr_ack2)
     wr_r <= 1'b0;
   if (wr)
     wr_fifo_cnt <= 1'b1;
   if ({rd_syn2, rd_syn1} == 2'b01)
     wr_fifo_cnt <= 1'b0;
   // register write data on write
   if (wr)
     wr_data_r <= wr_data;
  end
endmodule
```

testbench:

```
module fifo_tb();
vlog_tb_utils vlog_tb_utils0();
reg rst_n, clka, clkb, rd, wr;
 reg [3:0] datain;
wire [3:0] dataout_slow;
 wire [3:0] dataout_fast;
 wire full_fast, empty_slow, full_slow, empty_fast;
initial
```

```
begin
  rd = 0;
  wr = 0;
  datain = 4'b0000;
  rst_n = 1'b1;
  clka = 1'b0;
  clkb = 1'b0;
end
always
  #1 clka <= ~clka;
always
  #13 clkb <= ~clkb;
initial
begin
  #3 rst_n = 1'b0;
  #3 rst_n = 1'b1;
    #5 datain = 4'b0110;
  wr = 1'b1;
  #2 wr = 1'b0;
  #5 datain = 4'b0000;

  #80 rd = 1'b1;
  #26 rd = 1'b0;
  #100 $finish;
end
fifo #(.BUS_WIDTH(4)) fifo_f2si (
  .wr_data (datain),
  .rd_data (dataout_slow),
  .wr_clk  (clka),
  .rd_clk  (clkb),
  .wr      (wr),
  .rd      (rd),
  .full    (full_fast),
  .empty_n (empty_slow),
```

```
 .rst_n   (rst_n)
);
fifo #(.BUS_WIDTH(4)) fifo_s2fi (
 .wr_data (datain),
 .rd_data (dataout_fast),
 .wr_clk  (clkb),
 .rd_clk  (clka),
 .wr      (wr),
 .rd      (rd),
 .full    (full_slow),
 .empty_n (empty_fast),
 .rst_n   (rst_n)
);
Endmodule
```

code:

```
module sdram_controller (  wr_addr,  wr_data, wr_enable,
d_addr,rd_data,rd_ready,rd_enable,busy, rst_n, clk,
addr, bank_addr, data, clock_enable, cs_n, ras_n, cas_n, we_n,
data_mask_low, data_mask_high
);
/* Internal Parameters */
parameter ROW_WIDTH = 13;
parameter COL_WIDTH = 9;
parameter BANK_WIDTH = 2;
parameter SDRADDR_WIDTH = ROW_WIDTH > COL_WIDTH ?
ROW_WIDTH : COL_WIDTH;
parameter HADDR_WIDTH = BANK_WIDTH + ROW_WIDTH +
COL_WIDTH;
parameter CLK_FREQUENCY = 133;  // Mhz
parameter REFRESH_TIME =  32;   // ms     (how often we need to
refresh)
```

```
parameter REFRESH_COUNT = 8192; // cycles (how many refreshes
required per refresh time)
localparam IDLE      = 5'b00000;
localparam INIT_NOP1 = 5'b01000,
        INIT_PRE1 = 5'b01001,
        INIT_NOP1_1=5'b00101,
        INIT_REF1 = 5'b01010,
        INIT_NOP2 = 5'b01011,
        INIT_REF2 = 5'b01100,
        INIT_NOP3 = 5'b01101,
        INIT_LOAD = 5'b01110,
        INIT_NOP4 = 5'b01111;
localparam REF_PRE  =  5'b00001,
        REF_NOP1 =  5'b00010,
        REF_REF  =  5'b00011,
        REF_NOP2 =  5'b00100;
localparam READ_ACT  = 5'b10000,
        READ_NOP1 = 5'b10001,
        READ_CAS  = 5'b10010,
        READ_NOP2 = 5'b10011,
        READ_READ = 5'b10100;
localparam WRIT_ACT  = 5'b11000,
        WRIT_NOP1 = 5'b11001,
        WRIT_CAS  = 5'b11010,
        WRIT_NOP2 = 5'b11011;
localparam CMD_PALL = 8'b10010001,
        CMD_REF  = 8'b10001000,
        CMD_NOP  = 8'b10111000,
        CMD_MRS  = 8'b1000000x,
        CMD_BACT = 8'b10011xxx,
        CMD_READ = 8'b10101xx1,
        CMD_WRIT = 8'b10100xx1;
input  [HADDR_WIDTH-1:0]   wr_addr;
input  [15:0]           wr_data;
```

```
input                wr_enable;
input  [HADDR_WIDTH-1:0]   rd_addr;
output [15:0]          rd_data;
input                rd_enable;
output                rd_ready;
output                busy;
input                rst_n;
input                clk;
/* SDRAM SIDE */
output [SDRADDR_WIDTH-1:0] addr;
output [BANK_WIDTH-1:0]    bank_addr;
inout  [15:0]          data;
output                clock_enable;
output                cs_n;
output                ras_n;
output                cas_n;
output                we_n;
output                data_mask_low;
output                data_mask_high;
reg  [HADDR_WIDTH-1:0]   haddr_r;
reg  [15:0]          wr_data_r;
reg  [15:0]          rd_data_r;
reg                busy;
reg                data_mask_low_r;
reg                data_mask_high_r;
reg [SDRADDR_WIDTH-1:0]  addr_r;
reg [BANK_WIDTH-1:0]     bank_addr_r;
reg                rd_ready_r;
wire [15:0]          data_output;
wire                data_mask_low, data_mask_high;
assign data_mask_high = data_mask_high_r;
assign data_mask_low  = data_mask_low_r;
assign rd_data       = rd_data_r;
reg [3:0] state_cnt;
```

```
reg [9:0] refresh_cnt;
reg [7:0] command;
reg [4:0] state;
reg [7:0] command_nxt;
reg [3:0] state_cnt_nxt;
reg [4:0] next;
assign {clock_enable, cs_n, ras_n, cas_n, we_n} = command[7:3];
// state[4] will be set if mode is read/write
assign bank_addr     = (state[4]) ? bank_addr_r : command[2:1];
assign addr          = (state[4] | state == INIT_LOAD) ? addr_r : {
{SDRADDR_WIDTH-11{1'b0}}, command[0], 10'd0 };

assign data = (state == WRIT_CAS) ? wr_data_r : 16'bz;
assign rd_ready = rd_ready_r;
// all registered on posedge
always @ (posedge clk)
 if (~rst_n)
   begin
   state <= INIT_NOP1;
   command <= CMD_NOP;
   state_cnt <= 4'hf;
   haddr_r <= {HADDR_WIDTH{1'b0}};
   wr_data_r <= 16'b0;
   rd_data_r <= 16'b0;
   busy <= 1'b0;
   end
 else
   begin
   state <= next;
   command <= command_nxt;
   if (!state_cnt)
     state_cnt <= state_cnt_nxt;
   else
     state_cnt <= state_cnt - 1'b1;
```

```
if (wr_enable)
     wr_data_r <= wr_data;
   if (state == READ_READ)
     begin
     rd_data_r <= data;
     rd_ready_r <= 1'b1;
     end
   else
     rd_ready_r <= 1'b0;
   busy <= state[4];
   if (rd_enable)
     haddr_r <= rd_addr;
   else if (wr_enable)
     haddr_r <= wr_addr;
end
// Handle refresh counter
always @ (posedge clk)
 if (~rst_n)
  refresh_cnt <= 10'b0;
 else
  if (state == REF_NOP2)
    refresh_cnt <= 10'b0;
  else
    refresh_cnt <= refresh_cnt + 1'b1;
always @*
begin
   if (state[4])
     {data_mask_low_r, data_mask_high_r} = 2'b00;
   else
     {data_mask_low_r, data_mask_high_r} = 2'b11;
  bank_addr_r = 2'b00;
  addr_r = {SDRADDR_WIDTH{1'b0}};
  if (state == READ_ACT | state == WRIT_ACT)
    begin
```

```
    bank_addr_r = haddr_r[HADDR_WIDTH-1:HADDR_WIDTH-
(BANK_WIDTH)];
    addr_r = haddr_r[HADDR_WIDTH-
(BANK_WIDTH+1):HADDR_WIDTH-(BANK_WIDTH+ROW_WIDTH)];
    end
  else if (state == READ_CAS | state == WRIT_CAS)
    begin
    bank_addr_r = haddr_r[HADDR_WIDTH-1:HADDR_WIDTH-
(BANK_WIDTH)];
    addr_r = {
            {SDRADDR_WIDTH-(11){1'b0}},
            1'b1,                   /* A10 */
            {10-COL_WIDTH{1'b0}},
            haddr_r[COL_WIDTH-1:0]
           };
    end
  else if (state == INIT_LOAD)
    begin
    addr_r = {{SDRADDR_WIDTH-10{1'b0}}, 10'b1000110000};
    end
end
always @*
begin
  state_cnt_nxt = 4'd0;
  command_nxt = CMD_NOP;
  if (state == IDLE)
      if (refresh_cnt >= CYCLES_BETWEEN_REFRESH)
        begin
        next = REF_PRE;
        command_nxt = CMD_PALL;
        end
      else if (rd_enable)
        begin
        next = READ_ACT;
```

```
      command_nxt = CMD_BACT;
      end
    else if (wr_enable)
      begin
      next = WRIT_ACT;
      command_nxt = CMD_BACT;
      end
    else
      begin
      next = IDLE;
      end
 else
  if (!state_cnt)
    case (state)
      INIT_NOP1:
        begin
        next = INIT_PRE1;
        command_nxt = CMD_PALL;
        end
      INIT_PRE1:
        begin
        next = INIT_NOP1_1;
        end
      INIT_NOP1_1:
        begin
        next = INIT_REF1;
        command_nxt = CMD_REF;
        end
      INIT_REF1:
        begin
        next = INIT_NOP2;
        state_cnt_nxt = 4'd7;
        end
      INIT_NOP2:
```

```
  begin
  next = INIT_REF2;
  command_nxt = CMD_REF;
  end
INIT_REF2:
  begin
  next = INIT_NOP3;
  state_cnt_nxt = 4'd7;
  end
INIT_NOP3:
  begin
  next = INIT_LOAD;
  command_nxt = CMD_MRS;
  end
INIT_LOAD:
  begin
  next = INIT_NOP4;
  state_cnt_nxt = 4'd1;
  end
// REFRESH
REF_PRE:
  begin
  next = REF_NOP1;
  end
REF_NOP1:
  begin
  next = REF_REF;
  command_nxt = CMD_REF;
  end
REF_REF:
  begin
  next = REF_NOP2;
  state_cnt_nxt = 4'd7;
  end
```

```
// WRITE
WRIT_ACT:
 begin
 next = WRIT_NOP1;
 state_cnt_nxt = 4'd1;
 end
WRIT_NOP1:
 begin
 next = WRIT_CAS;
 command_nxt = CMD_WRIT;
 end
WRIT_CAS:
 begin
 next = WRIT_NOP2;
 state_cnt_nxt = 4'd1;
 end
 // READ
READ_ACT:
 begin
 next = READ_NOP1;
 state_cnt_nxt = 4'd1;
 end
READ_NOP1:
 begin
 next = READ_CAS;
 command_nxt = CMD_READ;
 end
READ_CAS:
 begin
 next = READ_NOP2;
 state_cnt_nxt = 4'd1;
 end
READ_NOP2:
 begin
```

```
          next = READ_READ;
          end
         default:
          begin
          next = IDLE;
          end
        endcase
     else
      begin
      // Counter Not Reached - HOLD
      next = state;
      command_nxt = command;
      end
end
endmodule
```

testbench:

```
module sdram_controller_tb();
   vlog_tb_utils vlog_tb_utils0();
   reg [23:0]  haddr;
   reg [15:0]  data_input;
   wire [15:0] data_output;
   wire busy;
   reg rd_enable, wr_enable, rst_n, clk;
   /* SDRAM SIDE */
   wire [12:0] addr;
   wire [1:0] bank_addr;
   wire [15:0] data;
   wire clock_enable, cs_n, ras_n, cas_n, we_n, data_mask_low,
data_mask_high;
   reg [15:0] data_r;
   assign data = data_r;
```

```
    initial
    begin
       haddr = 24'd0;
       data_input = 16'd0;
       rd_enable = 1'b0;
       wr_enable = 1'b0;
       rst_n = 1'b1;
       clk = 1'b0;
       data_r = 16'hzzzz;
    end
    always
       #1 clk <= ~clk;
    initial
    begin
      #3 rst_n = 1'b0;
      #3 rst_n = 1'b1;
      #120 haddr = 24'hfedbed;
      data_input = 16'd3333;
      #3 wr_enable = 1'b1;
      #6 wr_enable = 1'b0;
      haddr = 24'd0;
      data_input = 16'd0;
      #120 haddr = 24'hbedfed;
      #3 rd_enable = 1'b1;
      #6 rd_enable = 1'b0;
      haddr = 24'd0;
      #8 data_r = 16'hbbbb;
      #2 data_r = 16'hzzzz;
      #1000 $finish;
    end
  sdram_controller sdram_controlleri (
    /* HOST INTERFACE */
    .wr_addr(haddr),
    .wr_data(data_input),
```

```
   .rd_data(data_output),
   .busy(busy), .rd_enable(rd_enable), .wr_enable(wr_enable),
.rst_n(rst_n), .clk(clk),
   /* SDRAM SIDE */
   .addr(addr), .bank_addr(bank_addr), .data(data),
.clock_enable(clock_enable), .cs_n(cs_n), .ras_n(ras_n), .cas_n(cas_n),
.we_n(we_n), .data_mask_low(data_mask_low),
.data_mask_high(data_mask_high)
);

endmodule
```

block RAM code:

```
module dual_port_ram
   (   input clk, //clock
       input wr_en,    //write enable for port 0
       input [7:0] data_in,    //Input data to port 0.
       input [3:0] addr_in_0,  //address for port 0
       input [3:0] addr_in_1,  //address for port 1
       input port_en_0,    //enable port 0.
       input port_en_1,    //enable port 1.
       output [7:0] data_out_0,    //output data from port 0.
       output [7:0] data_out_1 //output data from port 1.
   );
//memory declaration.
reg [7:0] ram[0:15];
//writing to the RAM
always@(posedge clk)
begin
```

```
   if(port_en_0 == 1 && wr_en == 1)    //check enable signal and if write
enable is ON
       ram[addr_in_0] <= data_in;
end
//always reading from the ram, irrespective of clock.
assign data_out_0 = port_en_0 ? ram[addr_in_0] : 'dZ;
assign data_out_1 = port_en_1 ? ram[addr_in_1] : 'dZ;

endmodule
```

test bench:

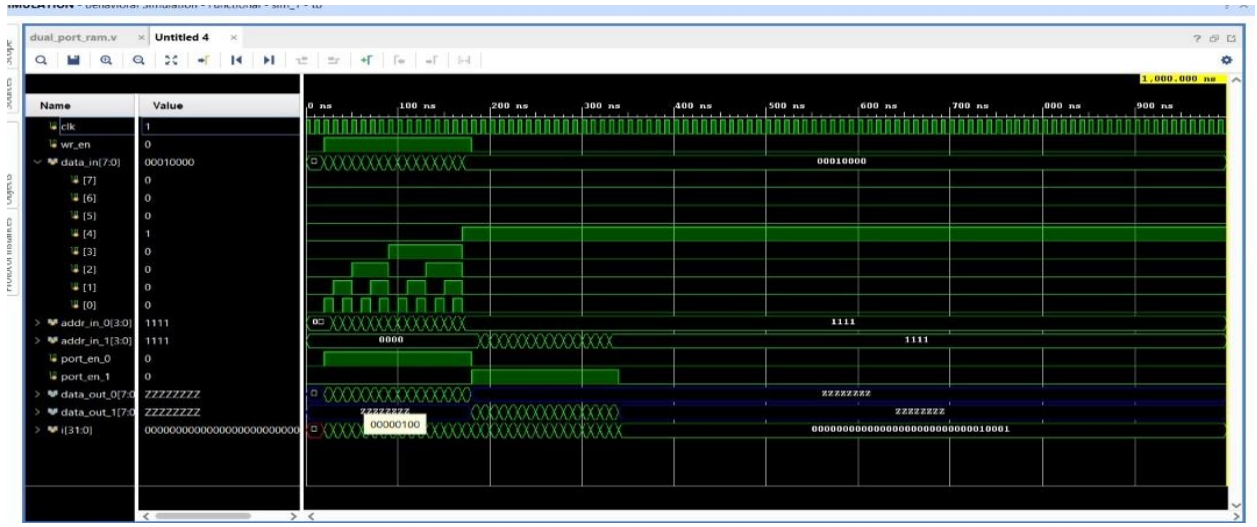
```
module tb;
   // Inputs
   reg clk;
   reg wr_en;
   reg [7:0] data_in;
   reg [3:0] addr_in_0;
   reg [3:0] addr_in_1;
   reg port_en_0;
   reg port_en_1;
   // Outputs
   wire [7:0] data_out_0;
   wire [7:0] data_out_1;
   integer i;
   // Instantiate the Unit Under Test (UUT)
   dual_port_ram uut (
      .clk(clk),
      .wr_en(wr_en),
      .data_in(data_in),
      .addr_in_0(addr_in_0),
      .addr_in_1(addr_in_1),
```

```
    .port_en_0(port_en_0),
    .port_en_1(port_en_1),
    .data_out_0(data_out_0),
    .data_out_1(data_out_1)
);

always
    #5 clk = ~clk;
initial begin
    // Initialize Inputs
    clk = 1;
    addr_in_1 = 0;
    port_en_0 = 0;
    port_en_1 = 0;
    wr_en = 0;
    data_in = 0;
    addr_in_0 = 0;
    #20;
    //Write all the locations of RAM
    port_en_0 = 1;
    wr_en = 1;
  for(i=1; i <= 16; i = i + 1) begin
        data_in = i;
        addr_in_0 = i-1;
        #10;
    end
    wr_en = 0;
    port_en_0 = 0;
    //Read from port 1, all the locations of RAM.
    port_en_1 = 1;
    for(i=1; i <= 16; i = i + 1) begin
        addr_in_1 = i-1;
        #10;
    end
```

```
    port_en_1 = 0;
  end

endmodule
```



Prefetch code:

```
initial o_wb_cyc <= 1'b0;
initial o_wb_stb <= 1'b0;
initial o_valid <= 1'b0;
always @(posedge i_clk)
if (i_reset)
        o_wb_cyc <= 1'b0;
        o_wb_stb <= 1'b0;
        o_valid <= 1'b0;
end else if (!o_wb_cyc)
begin
        if ((i_new_pc)||((o_valid)&&(i_stall_n)))
        begin
                o_wb_cyc <= 1'b1;
                o_wb_stb <= 1'b1;
if (i_new_pc)
      o_wb_addr <= i_pc;
```

```
else if ((o_valid)&&(i_stall_n))
      o_wb_addr <= o_wb_addr+1'b1;
o_valid <= 1'b0;
          end
end else begin // if (o_wb_cyc)
if (!i_wb_stall)
                    o_wb_stb <= 1'b0;
if (i_wb_ack)
          begin
                    o_wb_cyc <= 1'b0;
o_insn <= i_wb_data;
                    o_valid <= 1'b1;
          end
end

actual prefetch code
assign          o_wb_we = 1'b0;
          assign          o_wb_data = 32'h0000;
initialo_wb_cyc = 1'b0;
          initialo_wb_stb = 1'b0;
always @(posedge i_clk)
                    if ((i_reset)||((o_wb_cyc)&&((i_wb_ack)||(i_wb_err))))
                    begin
                              // End any bus cycle on a reset, or a return ACK
                              // or error.
                              o_wb_cyc <= 1'b0;
                              o_wb_stb <= 1'b0;
end else if ((!o_wb_cyc)&&(
                                        // 1.
                                        ((i_stalled_n)&&(!o_illegal))
                                        // 2.
                                        ||(invalid)
                                        // 3.
                                        ||(i_new_pc)))
```

```
                begin
                        // Initiate a bus transaction
                        o_wb_cyc <= 1'b1;
                        o_wb_stb <= 1'b1;
end else if (o_wb_cyc)
                begin
if (!i_wb_stb)
        o_wb_stb <= 1'b0;
if (i_new_pc)
                        begin
                                o_wb_cyc <= 1'b0;
                                o_wb_stb <= 1'b0;
                        end
                end
initialinvalid = 1'b0;
        always @(posedge i_clk)
                if ((i_reset)||(!o_wb_cyc))
                        invalid <= 1'b0;
                else if (i_new_pc)
                        invalid <= 1'b1;
initialo_wb_addr= 0;
        always @(posedge i_clk)
                if (i_new_pc)
                        o_wb_addr  <= i_pc;
                else if ((o_valid)&&(i_stalled_n)&&(!o_illegal))
                        o_wb_addr  <= o_wb_addr + 1'b1;
assign          o_pc = o_wb_addr;
always @(posedge i_clk)
                if ((o_wb_cyc)&&(i_wb_ack))
                        o_insn <= i_wb_data;
initial o_valid   = 1'b0;
        initial o_illegal = 1'b0;
always @(posedge i_clk)
                if ((i_reset)||(i_new_pc)||(i_clear_cache))
```

```
		begin
			// ....
			o_valid   <= 1'b0;
			o_illegal <= 1'b0; end else if
((o_wb_cyc)&&((i_wb_ack)||(i_wb_err)))
		begin
			// ...
			o_valid   <= 1'b1;
			o_illegal <= ( i_wb_err); end else if (i_stalled_n)
		begin
			// ---
			o_valid <= 1'b0; end
```

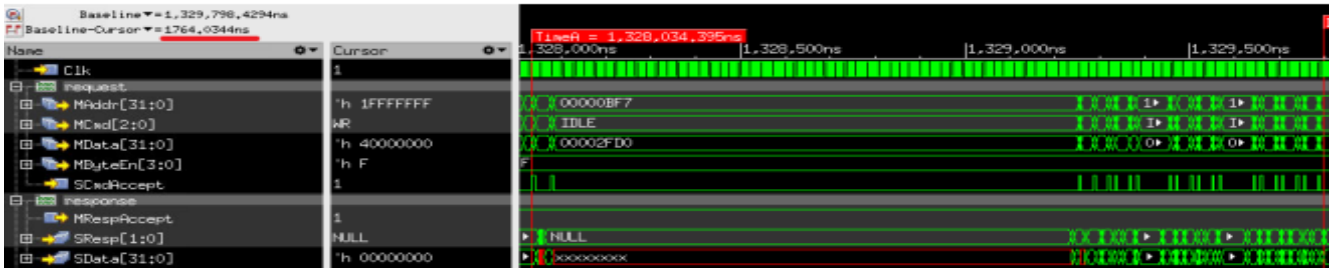
formula verification

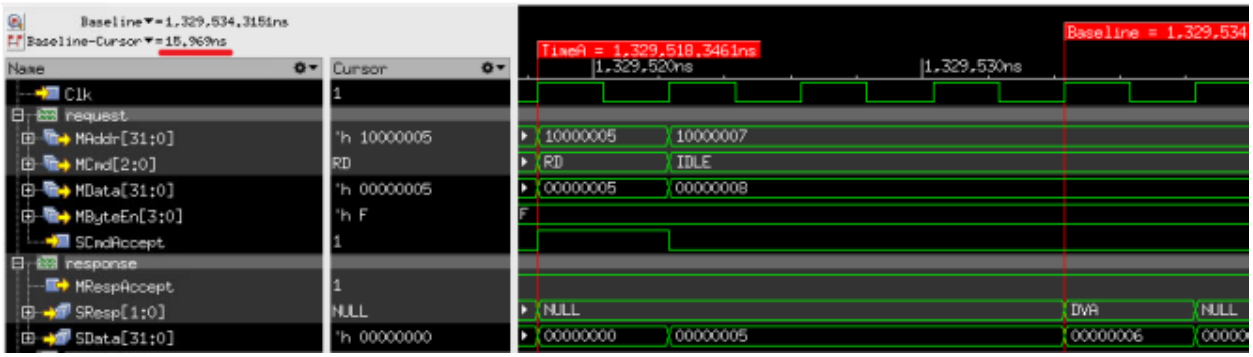
```
initial`ASSUME(i_reset); always @(posedge i_clk)
		if ((f_past_valid)&&($past(i_reset)))
			`ASSUME(i_new_pc); always @(posedge i_clk)
		if ((f_past_valid)&&($past(i_clear_cache)))
			`ASSUME(i_new_pc); always @(posedge i_clk)
		if ((f_past_valid)&&($past(i_reset)))
			`ASSUME(i_stalled_n); always @(posedge i_clk)
		if ((f_past_valid)&&($past(!o_valid))&&($past(i_stalled_n)))
			`ASSUME(i_stalled_n); localparam     F_CPU_DELAY =
4; always @(posedge i_clk)
		// If no instruction is ready, then keep our counter at zero
		if ((i_reset)||(!o_valid)||(i_stalled_n))
			f_cpu_delay <= 0;
		else
			// Otherwise, count the clocks the CPU takes to
respond
			f_cpu_delay <= f_cpu_delay + 1'b1; always @(posedge
i_clk)
		assume(f_cpu_delay < F_CPU_DELAY); fwb_master
#(.AW(AW), .DW(DW),.F_LGDEPTH(F_LGDEPTH),
```

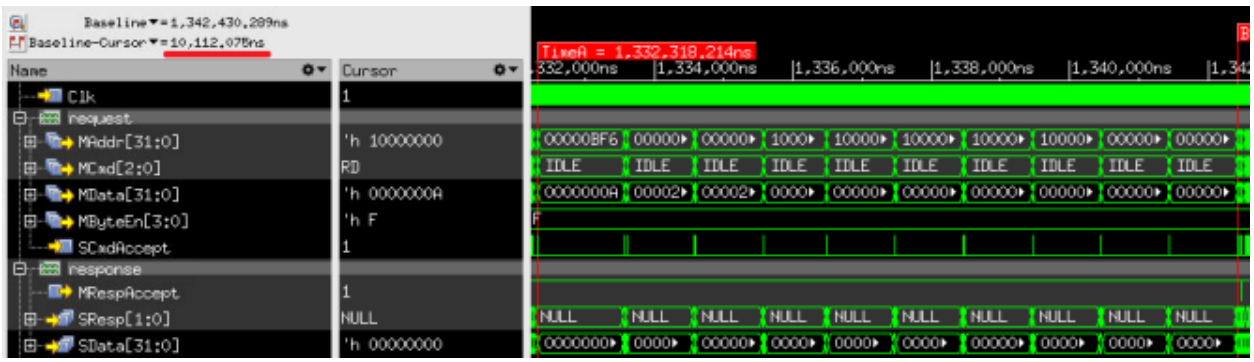
```
                .F_MAX_REQUESTS(1), .F_OPT_SOURCE(1),
                .F_OPT_RMW_BUS_OPTION(0),
                .F_OPT_DISCONTINUOUS(0))
        f_wbm(i_clk, i_reset,
                o_wb_cyc, o_wb_stb, o_wb_we, o_wb_addr, o_wb_data,
4'h0,
                i_wb_ack, i_wb_stall, i_wb_data, i_wb_err,
                f_nreqs, f_nacks, f_outstanding); always @(posedge
i_clk)
        if (o_wb_stb)
                assert(!o_wb_we); always @(posedge i_clk)
        if ((f_past_valid)&&($past(f_past_valid))
                        &&($past(i_clear_cache,2))
                        &&($past(o_wb_cyc,2)))
                // ...
                assert((!$past(o_wb_cyc))||(!o_wb_cyc)); always
@(posedge i_clk)
        if ((f_past_valid)&&($past(o_valid))&&(o_valid))
                assert($stable(o_wb_addr)); always @(posedge i_clk)
        if ((f_past_valid)&&($past(!i_reset))&&($past(invalid)))
                assert(o_wb_cyc); always @(posedge i_clk)
        if ((f_past_valid)&&($past(o_valid))&&($past(i_stalled_n)))
                assert(!o_valid); always @(*)
        if (o_wb_cyc)
                assert(!o_valid); always @(posedge i_clk)
        if ((f_past_valid)&&(!$past(i_reset))
                &&($past(o_wb_cyc))
                &&($past(!i_clear_cache))
                &&($past(i_wb_ack))&&(!$past(i_wb_err)))
        begin
                if (!invalid)
                        assert(o_valid);
        endalways @(posedge i_clk)
        if ((f_past_valid)&&($past(i_clear_cache)))
```

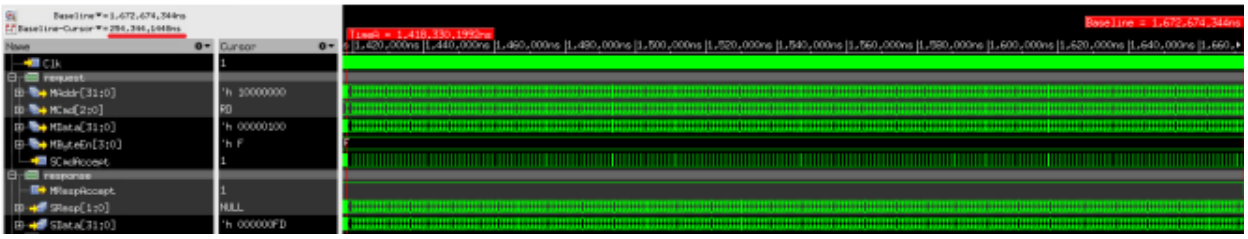
```
                assert(!o_valid); always @(posedge i_clk)
        if ((f_past_valid)&&($past(f_past_valid))
                        &&($past(i_clear_cache,2))
                        &&($past(o_wb_cyc,2)))
                // ...
                assert(!o_valid); always @(posedge i_clk)
    if ((f_past_valid)&&(!$past(i_reset))
                &&(!$past(i_new_pc))&&(!$past(i_clear_cache))
                &&($past(o_valid))&&(!$past(i_stalled_n)))
        assert($stable(o_valid)); always @(posedge i_clk)
    if ((f_past_valid)&&($past(o_valid))&&(o_valid))
    begin
        assert($stable(o_pc));
        assert($stable(o_insn));
        assert($stable(o_illegal));
    endalways @(posedge i_clk)
    if ((f_past_valid)&&(!$past(i_reset))
                &&(!$past(i_new_pc))&&(!$past(i_clear_cache))
                &&($past(!o_wb_cyc)))
        assert($stable(o_illegal)); initial       f_last_pc_valid = 1'b0;
    always @(posedge i_clk)
        if ((i_reset)||(i_clear_cache)||(i_new_pc)||(invalid))
            f_last_pc_valid <= 1'b0;
        else if (o_valid)
            f_last_pc_valid <= (!o_illegal); always @(posedge i_clk)
        if (o_valid)
            f_last_pc  <= o_pc; else if (f_last_pc_valid)
            assert(o_pc == f_last_pc + 1'b1); always @(posedge
i_clk)
        if ((f_past_valid)&&(o_valid)
                    &&(!$past(o_valid))&&(f_last_pc_valid))
            assert(o_pc == (f_last_pc + 1'b1)); initial    f_req_addr
= 0;
    always @(posedge i_clk)
```

```
                if (i_new_pc)
                        f_req_addr <= i_pc;
                else if ((!invalid)&&(o_wb_cyc)&&(i_wb_ack)&&(!i_wb_err))
                        f_req_addr <= f_req_addr + 1'b1; always @(posedge
i_clk)
                if (o_wb_cyc)
                        assert((invalid)||(f_req_addr == o_wb_addr)); else if
((!o_valid)&&(!i_new_pc)&&(!i_reset))
                        assert(f_req_addr == o_wb_addr); always @(posedge
i_clk)
                if ((f_past_valid)&&($past(invalid)))
                        assert(!invalid);
`end
endmodule
```









Branch evaluation unit:

Shift register:

module shift_reg #(parameter MSB=8)(input d,input clk,input en,input dir,input rstn,output reg [MSB-1:0]out);

always@(posedge clk)

if (!rstn)

out <=0;

else begin

if(en)

case (dir)

0:out<={out[MSB-2:0],d};

1:out<={d,out[MSB-1:1]};

```
 endcase
 end
    out <=out;
     end
endmodule
```

Interrupt controller code:

```
module INTR_CNTRL (
      input   wire          clk_in,     // Clock
      input   wire          rst_in,     // Reset
      input   wire    [7:0]   intr_rq,    // Interrupt request
      inout   wire    [7:0]   intr_bus,   // Bidirectional data bus
      input   wire          intr_in,    // Ack from processor
      output  wire          intr_out,   // Interrupt to processor
      output  wire          bus_oe      // Bus output enable
   );
   localparam  [3:0]   S_Reset              = 4'b0000
                S_GetCommands         = 4'b0001,
                S_JumpIntMethod       = 4'b0010,
                S_StartPolling        = 4'b0011,
                S_TxIntInfoPolling    = 4'b0100,
S_AckTxInfoRxPolling   = 4'b0101,
                S_AckISRDonePolling   = 4'b0110,
                S_StartPriority       = 4'b0111,
```

```
            S_TxIntInfoPriority    = 4'b1000,
 S_AckTxInfoRxPriority   = 4'b1001,
            S_AckISRDonePriority   = 4'b1010,
            S_Reserved1          = 4'b1011,
            S_Reserved2          = 4'b1100,
            S_Reserved3          = 4'b1101,
            S_Reserved4          = 4'b1110,
            S_Reserved5          = 4'b1111;
reg    [3:0]   state_reg, state_next;
reg    [1:0]   cmdMode_reg, cmdMode_next;
reg    [1:0]   cmdCycle_reg, cmdCycle_next;
reg    [2:0]   intrIndex_reg, intrIndex_next;
reg    [2:0]   intrPtr_reg, intrPtr_next;
reg    [2:0]   prior_table_next [0:7];
reg    [2:0]   prior_table_reg [0:7];
reg          oe_reg, oe_next;
reg    [7:0]   intrBus_reg, intrBus_next;     // Bus <= register if using bus as output
reg          intrOut_reg, intrOut_next;     // Interrupt output
integer       i;
always @ (posedge clk_in or posedge rst_in) begin
   if (rst_in) begin
      state_reg         <=  S_Reset;
      cmdMode_reg       <=  2'b00;
      cmdCycle_reg      <=  2'b00;
      oe_reg            <=  1'b0;
```

```
      intrBus_reg       <= 8'bzzzzzzzz;
      intrOut_reg       <= 1'b0;
      intrIndex_reg     <= 3'b000;
      intrPtr_reg       <= 3'b000;
      for (i = 0; i < 8; i = i + 1) begin
         prior_table_reg[i] <= 3'b000;
      end
   end
   else begin
      state_reg         <= state_next;
      cmdMode_reg       <= cmdMode_next;
      cmdCycle_reg      <= cmdCycle_next;
      intrBus_reg       <= intrBus_next;
      intrOut_reg       <= intrOut_next;
      oe_reg            <= oe_next;
      intrIndex_reg     <= intrIndex_next;
      intrPtr_reg       <= intrPtr_next;
      for (i = 0; i < 8; i = i + 1) begin
         prior_table_reg[i] <= prior_table_next[i];
      end
   end
end
always @(*) begin
   state_next        =  state_reg;
   cmdMode_next      =  cmdMode_reg;
```

```
cmdCycle_next      =   cmdCycle_reg;
oe_next            =   oe_reg;
intrOut_next       =   intrOut_reg;
intrBus_next       =   intrBus_reg;
intrIndex_next     =   intrIndex_reg;
intrPtr_next       =   intrPtr_reg;
for (i = 0; i < 8; i = i + 1) begin
   prior_table_next[i] =   prior_table_reg[i];
end
case (state_reg)
   // Reset state, every variable is set to zero and the bus is tristated.
   S_Reset: begin // 4'b0000
      cmdMode_next       =   2'b00;
      cmdCycle_next      =   2'b00;
      intrIndex_next     =   3'b000;
      intrPtr_next       =   3'b000;
      for (i = 0; i < 8; i = i + 1) begin
         prior_table_next[i] =   3'b000;
      end
      //intrBus_next       =   8'bzzzzzzzz;
      oe_next            =   1'b0;
      state_next  =   S_GetCommands;
   end
    S_GetCommands: begin // 4'b0001
      oe_next =   1'b0;
```

```
            case (intr_bus[1:0])
              2'b01: begin
                cmdMode_next   =  2'b01;
state_next     =  S_JumpIntMethod;

              2'b10: begin
                case (cmdCycle_reg)
                  2'b00: begin
                    prior_table_next[0] =  intr_bus[7:5];
                    prior_table_next[1] =  intr_bus[4:2];
                    state_next        =  S_GetCommands;
                    cmdCycle_next     =  cmdCycle_reg + 1'b1;
                  end
                  2'b01: begin
                    prior_table_next[2] =  intr_bus[7:5];
                    prior_table_next[3] =  intr_bus[4:2];
                    state_next        =  S_GetCommands;
                    cmdCycle_next     =  cmdCycle_reg + 1'b1;
                  end
                  2'b10: begin
                    prior_table_next[4] =  intr_bus[7:5];
                    prior_table_next[5] =  intr_bus[4:2];
                    state_next        =  S_GetCommands;
                    cmdCycle_next     =  cmdCycle_reg + 1'b1;
                  end
```

```
                2'b11: begin
                  prior_table_next[6] =  intr_bus[7:5];
                  prior_table_next[7] =  intr_bus[4:2];
                  state_next        =  S_JumpIntMethod;
                  cmdCycle_next     =  cmdCycle_reg + 1'b1;
                  cmdMode_next      =  2'b10;
                end
                default: begin
                  state_next    =  S_GetCommands;
cmdCycle_next  =  2'b00;
                  cmdMode_next  =  2'b00;
                end
              endcase
            end
            default: begin
              state_next  =  S_GetCommands;
            end
          endcase
        end
        S_JumpIntMethod: begin // 4'b0010
          intrIndex_next  =  3'b000;
          intrPtr_next    =  3'b000;

          case (cmdMode_reg)
            2'b01: begin
```

```
        state_next  =   S_StartPolling;
      end
      2'b10: begin
        state_next  =   S_StartPriority;
      end
      default: begin
        state_next  =   S_Reset;
      end
    endcase
    //intrBus_next    =   8'bzzzzzzzz;
    oe_next       =   1'b0;
  end
  S_StartPolling: begin // 4'b0011
    if (intr_rq[intrIndex_reg]) begin
      intrOut_next    =   1'b1;
      state_next      =   S_TxIntInfoPolling;
    end
    else begin
      intrOut_next    =   1'b0;
      intrIndex_next  =   intrIndex_reg + 1;
    end
    //intrBus_next    =   8'bzzzzzzzz;
    oe_next       =   1'b0;
  end
  S_TxIntInfoPolling: begin // 4'b0100
```

```
            if (~intr_in) begin
              intrOut_next   =  1'b0;                  i
ntrBus_next   =  {5'b01011, intrIndex_reg};
              oe_next       =  1'b1;
              state_next     =  S_AckTxInfoRxPolling;
            end
            else
              state_next     =  S_TxIntInfoPolling;
          end
        S_AckTxInfoRxPolling: begin // 4'b0101
            if (~intr_in) begin
              //intrBus_next   =  8'bzzzzzzzz;
              oe_next       =  1'b0;
              state_next     =  S_AckISRDonePolling;
            end
          end
             if ((~intr_in) && (intr_bus[7:3] == 5'b10100) && (intr_bus[2:0] == intrIndex_reg))
begin
              state_next  =  S_StartPolling;
            end
            else if ((~intr_in) && (intr_bus[7:3] != 5'b10100) && (intr_bus[2:0] !=
intrIndex_reg)) begin
              state_next  =  S_Reset;
            end
            else begin
```

```
      state_next  =   S_AckISRDonePolling;
    end
  end
  S_StartPriority: begin // 4'b0111
    if (intr_rq[prior_table_reg[0]]) begin
      intrPtr_next    =   prior_table_reg[0];
      intrOut_next    =   1'b1;
      state_next      =   S_TxIntInfoPriority;
    end
    else if (intr_rq[prior_table_reg[1]]) begin
      intrPtr_next    =   prior_table_reg[1];
      intrOut_next    =   1'b1;
      state_next      =   S_TxIntInfoPriority;
    end
    else if (intr_rq[prior_table_reg[2]]) begin
      intrPtr_next    =   prior_table_reg[2];
      intrOut_next    =   1'b1;
      state_next      =   S_TxIntInfoPriority;
    end
     else if (intr_rq[prior_table_reg[3]]) begin
      intrPtr_next    =   prior_table_reg[3];
      intrOut_next    =   1'b1;
      state_next      =   S_TxIntInfoPriority;
    end
     else if (intr_rq[prior_table_reg[4]]) begin
```

```
    intrPtr_next    =   prior_table_reg[4];
    intrOut_next    =   1'b1;
    state_next      =   S_TxIntInfoPriority;
  end
  else if (intr_rq[prior_table_reg[5]]) begin
    intrPtr_next    =   prior_table_reg[5];
    intrOut_next    =   1'b1;
    state_next      =   S_TxIntInfoPriority;
  end
  else if (intr_rq[prior_table_reg[6]]) begin
    intrPtr_next    =   prior_table_reg[6];
    intrOut_next    =   1'b1;
    state_next      =   S_TxIntInfoPriority;
  end
  else if (intr_rq[prior_table_reg[7]]) begin
    intrPtr_next    =   prior_table_reg[7];
    intrOut_next    =   1'b1;
    state_next      =   S_TxIntInfoPriority;
  end

  else begin
    state_next  =   S_StartPriority;
  end
  //intrBus_next    =   8'bzzzzzzzz;
  oe_next         =   1'b0;
```

```
        end
        S_TxIntInfoPriority: begin // 4'b1000
          if (~intr_in) begin
            intrOut_next   =  1'b0;
            intrBus_next   =  {5'b10011, intrPtr_reg};
            oe_next        =  1'b1;
            state_next     =  S_AckTxInfoRxPriority;
          end
        end
            S_AckTxInfoRxPriority: begin // 4'b1001
          if (~intr_in) begin
            //intrBus_next   =  8'bzzzzzzzz;
            oe_next        =  1'b0;
            state_next     =  S_AckISRDonePriority;
          end
        end

        S_AckISRDonePriority: begin // 4'b1010
                    if ((~intr_in) && (intr_bus[7:3] == 5'b01100) && (intr_bus[2:0] ==
intrPtr_reg)) begin
            state_next  =  S_StartPriority;
          end
          else if ((~intr_in) && (intr_bus[7:3] != 5'b01100) && (intr_bus[2:0] != intrPtr_reg))
begin
            state_next  =  S_Reset;
```

```
            end
            else begin
               state_next  =   S_AckISRDonePriority;       .
            end
         end
         default: begin
            state_next     =   S_Reset;
            //intrBus_next    =   8'bzzzzzzzz;
            oe_next        =   1'b0;
         end
      endcase
   end
   assign bus_oe   =   oe_reg;
endmodule
```

testbench:

```
module INTR_CNTRL_TB;
   reg          clk_in;        // Clock
   reg          rst_in;        // Reset
   reg    [7:0]   intr_rq;       // Interrupt request
   reg    [7:0]   intr_bus_in;    // Bidirectional data bus
   reg          intr_in;       // Ack from processor
   wire          intr_out;      // Interrupt to processor
   wire          bus_oe;        // High if controller drives the bus.
   wire    [7:0]   intr_bus_test;
```

```
wire    [7:0]   intr_bus_out;
INTR_CNTRL DUT (
   .clk_in     (clk_in        ),
   .rst_in     (rst_in        ),
   .intr_rq    (intr_rq       ),
   .intr_bus   (intr_bus_test  ),
   .intr_in    (intr_in       ),
   .intr_out   (intr_out      ),
   .bus_oe     (bus_oe        )
   );
assign intr_bus_test    =   (bus_oe == 0) ? intr_bus_in : 8'bz;
assign intr_bus_out     =   (bus_oe == 1) ? intr_bus_test : 8'bz;
reg     [2:0]   currentService;
integer j;
initial begin
         clk_in         =   1'b0;
   rst_in         =   1'b1;
   intr_rq        =   8'b0;
   intr_bus_in    =   8'b0;
   intr_in        =   1'b1;
   #20;
   rst_in         =   1'b0;
   #20;
   intr_bus_in    =   8'b0000_0001;
   #40;
```

```
    intr_rq        =   8'b1010_1010;


    $display
$display
 $display
    for (j = 0; j < 8; j = j + 1) begin
        if (j == 4) begin
            intr_rq = 8'b0101_0101;
        end
        $display
        $display
        $display
        $display ("Interrupt Request = %b", intr_rq);


        $display ("Waiting for interrupt from controller");
        wait(intr_out);
        #60;
          intr_in     =   1'b0;
        #10
        intr_in     =   1'b1;
        $display ("Interrupt Acknowledged by processor.");
        $display("Currently servicing %b", intr_bus_out[2:0]);
        $display ("The controller is driving the bus, %b", intr_bus_out);
        currentService          =   intr_bus_out[2:0];
        intr_rq[currentService] =   1'b0;
```

```
    if (intr_bus_out == {5'b01011, currentService}) begin
        $display ("Proper address on the bus");
    end else begin
        $display ("ERROR: Wrong address on the bus");
        $finish;
    end
    #60
    intr_in    =   1'b0;
    #10
    intr_in    =   1'b1;
    $display ("Address acknowledged by processor");

    #60
    intr_bus_in    =   {5'b10100, currentService};
    intr_in        =   1'b0;
    #10
    intr_in        =   1'b1;
    $display ("ISR Routine complete by processor");
    $display ("The processor is driving the bus, %b", intr_bus_in);
    $display
    #100;
end
#100;
rst_in         =   1'b1;
intr_rq        =   8'b0;
```

```
intr_bus_in    =  8'b0;
intr_in        =  1'b1;
#50;
rst_in         =  1'b0;
#50;
// Set priorities => 5, 3, 7, 0, 4, 2, 6, 1.
intr_bus_in    =  8'b101_011_10;
#10;
intr_bus_in    =  8'b111_000_10;
#10;
intr_bus_in    =  8'b100_010_10;
#10;
intr_bus_in    =  8'b110_001_10;
#40;
intr_rq        =  8'b1111_1111;  // Activate random interrupts.
    $display
for (j = 0; j < 10; j = j + 1) begin
  if (j == 4) begin
    intr_rq [3] = 1'b1;
  end
  if (j == 6) begin
    intr_rq [5] = 1'b1;
  end
  wait(intr_out);
  #60;
```

```
    intr_in    =   1'b0;
     #10
    intr_in    =   1'b1;
          currentService        =   intr_bus_out[2:0];
    intr_rq[currentService] =   1'b0;
    if (intr_bus_out == {5'b10011, currentService}) begin
       $display ("Proper address on the bus");
    end else begin
       $display ("ERROR: Wrong address on the bus");
       $finish;
    end
    #60
    intr_in    =   1'b0;
    #10
    intr_in    =   1'b1;
    #60
    intr_bus_in    =   {5'b01100, currentService};
    intr_in       =   1'b0;
    #10                                  // Active for 1 clock.
    intr_in       =   1'b1;                    // Ack ends.
    #100;
  end
  #100;
  $finish;
end
```

```
   always
      #5 clk_in   =   ~clk_in;
endmodule // INTR_CNTRL_TB
```

side buffer code:

```
module FIFObuffer( Clk,   dataIn,  RD,  WR, EN,
                dataOut, Rst, EMPTY, FULL );
input  Clk,  RD, WR,  EN,  Rst;
output  EMPTY,
      FULL;
input   [31:0]    dataIn;
output reg [31:0] dataOut; // internal registers
reg [2:0]  Count = 0;
reg [31:0] FIFO [0:7];
reg [2:0]  readCounter = 0,
        writeCounter = 0;
assign EMPTY = (Count==0)? 1'b1:1'b0;
assign FULL = (Count==8)? 1'b1:1'b0;
always @ (posedge Clk)
begin
 if (EN==0);
 else begin
  if (Rst) begin
   readCounter = 0;
   writeCounter = 0;
```

```
 end
 else if (RD ==1'b1 && Count!=0) begin
  dataOut  = FIFO[readCounter];
  readCounter = readCounter+1;
 end
 else if (WR==1'b1 && Count<8) begin
  FIFO[writeCounter]  = dataIn;
  writeCounter  = writeCounter+1;
 end
 else;
end
if (writeCounter==8)
 writeCounter=0;
else if (readCounter==8)
 readCounter=0;
else;
if (readCounter > writeCounter) begin
 Count=readCounter-writeCounter;
end
else if (writeCounter > readCounter)
 Count=writeCounter-readCounter;
else;
end
endmodule
```