Table of Changes to Xilinx MIG controller

Modified MIG Modules

Table 1 below shows the names of the fifteen modules that require changes from the standard Xilinx MIG controller for a XCKU060-2FFVA1156E device. When run properly, the example script will make these changes for you.

Table 1 - List of Modified Xilinx IP Modules.

Category	STT-MRAM Timing Parameter and Performance Changes	*DDR4_0.sv*	*DDR4_0_DDR4.sv*	*DDR4_0_DDR4_mem_infc.sv*	*DDR4_v2_2_cal.sv*	*DDR4_v2_2_mc.sv*	*DDR4_v2_2_mc_arb_c.sv*	*DDR4_v2_2_mc_arb_mux_p.sv*	*DDR4_v2_2_mc_group.sv*	*Ddr4_v2_2_mc_ctl.sv*	*DDR4_v2_2_mc_ref.sv*	*DDR4_v2_2_ui.sv*	*Ddr4_v2_2_ui_cmd.sv*	*DDR4_v2_2_ui_rd_data.sv*	*DDR4_v2_2_ui_wr_data.sv*	*DDR4_v2_2_mc_periodic.sv*
	File Number	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
Timing	Timing settings and counter width changes		*x*			*x*		*x*	*x*							
Power-up	Anti-scribbling changes (NOMEM mode)				*x*											
Power-down	SCRAM input signal to drain writes with CAS page closes	x	x	x		x					x					
Power-down	Created SCRAM output status signals	x	x	x		x			x			x				
Performance	Auto pre-charges on the 8th BL8 of a CAS page		*x*			*x*				*x*		*x*	*x*	*x*	*x*	
Performance	FIFO-DEPTH doubled								*x*							
Performance	Changed to emit requests faster						*x*		*x*							*x*

MIG Module Modifications

Below you will find tables for each one of the fifteen MIG modules that is modified by running the Everspin script **patch_ddr4mig_mram_2.73.tcl**. The table shows changes to the files along with the corresponding standard Xilinx DDR4 MIG outputs.

DDR4_0.sv

Standard Xilinx MIG	Modified MRAM MIG
(null)	*// begin Status and control inputs and outputs* *input power_fail_has_scramed,* *output cntr_power_fail_complete,* *output inflight_writes, // status output* *// end Status and control inputs and outputs*
(null)	*// begin Status and control inputs and outputs* *.power_fail_has_scramed (power_fail_has_scramed),* *.cntr_power_fail_complete (cntr_power_fail_complete),* *.inflight_writes (inflight_writes),* *// end Status and control inputs and outputs*

DDR4_0_DDR4.sv

Standard Xilinx MIG	Modified MRAM MIG
parameter integer COL_WIDTH = 10,	*parameter integer COL_WIDTH = 7,*
parameter tFAW = 28,//In DDR4 clock cycles	*parameter tFAW = 160, //In DDR clock cycles*
parameter tWTR_L = 5, //In DDR4 clock cycles	*parameter tWTR_L = 6, //In DDR clock cycles*
parameter tWTR_S = 2, //In DDR4 clock cycles	*parameter tWTR_S = 6, //In DDR clock cycles*
parameter tRFC = 234, //In DDR4 clock cycles	*parameter tRFC = 830, //In DDR clock cycles*
parameter tREFI = 0, //In DDR4 clock cycles	*parameter tREFI = 17'h1ffff, //In DDR clock cycles*
parameter tRP = 9, //In DDR4 clock cycles	*parameter tRP = 54, //In DDR clock cycles*
parameter tRRD_L = 5, //In DDR4 clock cycles	*parameter tRRD_L = 7, //In DDR clock cycles*
parameter tRRD_S = 4, //In DDR4 clock cycles	*parameter tRRD_S = 7, //In DDR clock cycles*
parameter tRAS = 22, //In DDR4 clock cycles	*parameter tRAS = 69, //In DDR clock cycles*
parameter tRCD = 9, //In DDR4 clock cycles	*parameter tRCD = 90, //In DDR clock cycles*
parameter MEMORY_PART = "MT40A512M16HA-075E",	*parameter MEMORY_PART = "EMD4E001GAS1",*
parameter MEMORY_DENSITY = "8Gb",	*parameter MEMORY_DENSITY = "1Gb",*
parameter MEMORY_SPEED_GRADE = "075E",	*parameter MEMORY_SPEED_GRADE = "125",*
parameter MR0 = 13'b0000100000100,	*parameter MR0 = 13'b0000000000100,*
parameter MR1 = 13'b0001100000001,	*parameter MR1 = 13'b0_0000_0000_0101,*
parameter MR5 = 13'b0010000000000,	*parameter MR5 = 13'b0_0100_1110_0000,*
parameter MR6 = 13'b0000000010100,	*parameter MR6 = 13'b0_0000_0010_0010,*

parameter MR2 = 13'b0000000000000, parameter MR3 = 13'b0000000000000, parameter MR4 = 13'b0000000000000,	parameter MR2 = 13'b0_0000_0000_0000, parameter MR3 = 13'b0_0001_1000_0000, parameter MR4 = 13'b0_0000_0000_0000,

(null)	// begin Status and control inputs and outputs
	input power_fail_has_scramed,
	output cntr_power_fail_complete,
	output inflight_writes,
	// end of the patch

(null)	// begin Status and control inputs and outputs
	.power_fail_has_scramed (power_fail_has_scramed),
	.cntr_power_fail_complete (cntr_power_fail_complete),
	.inflight_writes (inflight_writes),
	// end of the patch

DDR4_0_DDR4_mem_infc.sv

Standard Xilinx MIG	Modified MRAM MIG
,parameter integer DATA_BUF_ADDR_WIDTH = 5	,parameter integer DATA_BUF_ADDR_WIDTH = 6

(null)	// begin Status and control inputs and outputs
	,input power_fail_has_scramed
	,output cntr_power_fail_complete
	,output inflight_writes
	// end of patch

(null)	wire inflight_writes_mc; wire inflight_writes_ui; assign inflight_writes = inflight_writes_mc | inflight_writes_ui;

(null)	// begin Status and control inputs and outputs
	,.power_fail_has_scramed (power_fail_has_scramed)
	,.cntr_power_fail_complete (cntr_power_fail_complete)
	,.inflight_writes (inflight_writes_mc)
	,.inflight_writes_ui (inflight_writes_ui)
	// end of patch

(null)	.inflight_writes (inflight_writes_ui),

DDR4_v2_2_cal.sv

Standard Xilinx MIG	Modified MRAM MIG
(null)	, parameter MRBITS = 14
(null)	typedef struct packed { logic [31:0] during_cal_mr0; logic [31:0] during_cal_mr1; logic [31:0] during_cal_mr2; logic [31:0] during_cal_mr3; logic [31:0] during_cal_mr4; logic [31:0] during_cal_mr5;

```
  logic [31:0]  during_cal_mr6;
  logic [31:0]  after_cal_mr0;
  logic [31:0]  after_cal_mr1;
  logic [31:0]  after_cal_mr2;
  logic [31:0]  after_cal_mr3;
  logic [31:0]  after_cal_mr4;
  logic [31:0]  after_cal_mr5;
  logic [31:0]  after_cal_mr6;
  logic [63:0]  treg_adc;
} ddr_mp_struct_t;

ddr_mp_struct_t ddr_mp;

initial begin
   ddr_mp.during_cal_mr0 = MR0 | 32'h2000;   // a13 = nomem=1
   ddr_mp.during_cal_mr1 = MR1;
   ddr_mp.during_cal_mr2 = MR2;
   ddr_mp.during_cal_mr3 = MR3;
   ddr_mp.during_cal_mr4 = MR4;
   ddr_mp.during_cal_mr5 = MR5;
   ddr_mp.during_cal_mr6 = MR6;
   ddr_mp.after_cal_mr0  = MR0;
   ddr_mp.after_cal_mr1  = MR1;
   ddr_mp.after_cal_mr2  = MR2;
   ddr_mp.after_cal_mr3  = MR3;
   ddr_mp.after_cal_mr4  = MR4;
   ddr_mp.after_cal_mr5  = MR5;
   ddr_mp.after_cal_mr6  = MR6;
   ddr_mp.treg_adc       = 64'h0;
end

wire       ub_owns_cal;
reg ub_cal_now;
assign ub_owns_cal   = (RTL_DDR_INIT == 0) | calDone | ( ub_ready &
ub_cal_now & ~ub_calDone);
reg [3:0] MR6_count;
reg mr2_pass2;         //ddh perform second RTL pass to re-initialize MR2
reg mr2_done;          //ddh complete second RTL pass to re-initialize
MR2
// ----------------------------------------------------------------
// TREG
// ----------------------------------------------------------------
localparam   TREG_AFI_RATE_RATIO     =  4;
localparam   TREG_AFI_ADDR_WIDTH     =  64;
localparam   TREG_AFI_BANKADDR_WIDTH =  12;
localparam   TREG_AFI_CONTROL_WIDTH  =  4;
localparam   TREG_AFI_CS_WIDTH       =  4;
localparam   TREG_AFI_CLK_EN_WIDTH   =  4;
```

```
localparam   TREG_AFI_DM_WIDTH       =  72;
localparam   TREG_AFI_DQ_WIDTH       = 576;
localparam   TREG_AFI_ODT_WIDTH      =  4;
localparam   TREG_AFI_WRITE_DQS_WIDTH =  36;

logic  treg_mux_sel_in = 0;
logic  treg_mux_sel_out;
logic  treg_tmri_stall_req;
logic  treg_tmri_stall_ack = 1;

// Mux output to the PHY logic
logic   [TREG_AFI_ADDR_WIDTH-1:0]      treg_afi_addr;
logic   [TREG_AFI_BANKADDR_WIDTH-1:0]   treg_afi_ba;
logic   [TREG_AFI_CONTROL_WIDTH-1:0]    treg_afi_cas_n;
logic   [TREG_AFI_CLK_EN_WIDTH-1:0]     treg_afi_cke;
logic   [TREG_AFI_CS_WIDTH-1:0]        treg_afi_cs_n;
logic   [TREG_AFI_ODT_WIDTH-1:0]       treg_afi_odt;
logic   [TREG_AFI_CONTROL_WIDTH-1:0]    treg_afi_ras_n;
logic   [TREG_AFI_CONTROL_WIDTH-1:0]    treg_afi_we_n;
logic   [TREG_AFI_DM_WIDTH-1:0]        treg_afi_dm;
logic   [TREG_AFI_CONTROL_WIDTH-1:0]    treg_afi_rst_n;
logic   [TREG_AFI_WRITE_DQS_WIDTH-1:0]  treg_afi_dqs_burst;
logic   [TREG_AFI_DQ_WIDTH-1:0]        treg_afi_wdata;
logic   [TREG_AFI_WRITE_DQS_WIDTH-1:0]  treg_afi_wdata_valid;
logic   [TREG_AFI_RATE_RATIO-1:0]      treg_afi_rdata_en;
logic   [TREG_AFI_RATE_RATIO-1:0]      treg_afi_rdata_en_full;

logic        treg_nowmem_valid;
logic   [7:0]  treg_nowmem_count;
logic   [3:0]  treg_nowmem_cmd  ;
logic   [2:0]  treg_nowmem_ba   ;
logic   [15:0] treg_nowmem_addr ;
logic        treg_nowmem_done ;

logic   [7:0]   treg_init_cal_BG;
logic   [15:0]  treg_init_cal_BA;
logic   [143:0] treg_init_cal_ADR;
logic   [7:0]   treg_init_cal_CS_n;


// Test Mode interface
logic           treg_TM_write_start;
logic           treg_TM_write_complete;

logic   [1:0]      treg_TMR_select = 2'b11;
logic   [6*4-1:0]   treg_TMR_input_values = 0;
logic           treg_cfg_auto_nomem = 0;
logic           treg_local_init_done = 1;
```

	logic [63:0] treg_nowmem_adc; // No W Mem Address, Data, Command
	logic [31:0] treg_nowmem_data; // No W Mem RAM Data out
	
	assign treg_nowmem_adc = ddr_mp.treg_adc[63:0]; // No W Mem Address, Data, Command

reg [5:0] calSt;	reg [6:0] calSt;
reg [5:0] retSt;	reg [6:0] retSt;
assign caldone = (BYPASS_CAL == "TRUE")? ((RTL_DDR_INIT == 1) ? initDone : ub_initDone) : ub_calDone;	assign caldone = (BYPASS_CAL == "TRUE")? ((RTL_DDR_INIT == 1) ? (initDone) : ub_initDone) : ub_calDone & initDone;

assign cal_BG_int = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_BG : init_cal_BG;
assign cal_BA_int = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_BA : init_cal_BA;
assign cal_ADR_int = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_ADR : init_cal_ADR[ABITS*8-1:0];
assign cal_inv_int = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_inv : init_cal_inv;
assign cal_mrs_int = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_mrs : init_cal_mrs;
	assign cal_BG_int = ub_owns_cal ? ub_cal_BG : init_cal_BG;
	assign cal_BA_int = ub_owns_cal ? ub_cal_BA : init_cal_BA;
	assign cal_ADR_int = ub_owns_cal ? ub_cal_ADR : init_cal_ADR[ABITS*8-1:0];
	assign cal_inv_int = ub_owns_cal ? ub_cal_inv : init_cal_inv;
	assign cal_mrs_int = ub_owns_cal ? ub_cal_mrs : init_cal_mrs;

assign rtl_initDone = (RTL_DDR_INIT == 1) ? initDone : 1'b0;	assign rtl_initDone = (RTL_DDR_INIT == 1) ? ub_cal_now : 1'b0;
assign cal_RESET_n = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_RESET_n : init_cal_RESET_n;	assign cal_RESET_n = ub_owns_cal ? ub_cal_RESET_n : init_cal_RESET_n;
assign cal_ACT_n = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_ACT_n : init_cal_ACT_n;	assign cal_ACT_n = ub_owns_cal ? ub_cal_ACT_n : init_cal_ACT_n;
assign cal_C = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_C : '0;	assign cal_C = ub_owns_cal ? ub_cal_C : 1'b0;
assign cal_CKE = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_CKE : init_cal_CKE;	assign cal_CKE = ub_owns_cal ? ub_cal_CKE : init_cal_CKE;
assign cal_CS_n = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_CS_n : init_cal_CS_n;	assign cal_CS_n = ub_owns_cal ? ub_cal_CS_n : init_cal_CS_n;
assign cal_ODT = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_ODT : init_cal_ODT;	assign cal_ODT = ub_owns_cal ? ub_cal_ODT : init_cal_ODT;
assign cal_PAR = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_PAR : init_cal_PAR;	assign cal_PAR = ub_owns_cal ? ub_cal_PAR : init_cal_PAR;
assign cal_WE_n = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_WE : init_cal_WE_n;	assign cal_WE_n = ub_owns_cal ? ub_cal_WE : init_cal_WE_n;
assign cal_CAS_n = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_CAS : init_cal_CAS_n;	assign cal_CAS_n = ub_owns_cal ? ub_cal_CAS : init_cal_CAS_n;
assign cal_RAS_n = ub_ready || (RTL_DDR_INIT == 0) ? ub_cal_RAS : init_cal_RAS_n;	assign cal_RAS_n = ub_owns_cal ? ub_cal_RAS : init_cal_RAS_n;
(null)	,calStTREG = 6'h10
input [5:0] st;	input [6:0] st;
input [12:0] mr;	input [MRBITS-1:0] mr;
for (i = 0; i <= 12; i = i + 1) init_cal_ADR[i*8+:8] <= #TCQ {8{mr[i]}};	for (i = 0; i <= MRBITS-1; i = i + 1) init_cal_ADR[i*8+:8] <= #TCQ {8{mr[i]}};

`for(i = 13; i < ABITS; i = i + 1) init_cal_ADR[i*8+:8] <= #TCQ 8'b0;`	`for(i = 13; i <= ABITS; i = i + 1) init_cal_ADR[i*8+:8] <= #TCQ 8'b0;`
`init_cal_ADR[((ABITS-1)*8)+:8] <= #TCQ (ABITS == 18)? 8'b0: init_cal_ADR[((ABITS-1)*8)+:8];` `init_cal_ADR[111:104] <= #TCQ 8'b0;`	Removed
(null)	`for(i = MRBITS; i <= ABITS; i = i + 1) init_cal_ADR[i*8+:8] <= #TCQ 8'b0;`
`if (calSt == calStRESET) begin`	`if ((calSt == calStRESET) || (calSt == calStGOGO)) begin`
(null)	`MR6_count <= 0;`

(null)	`mr2_pass2 <= 1'b0;`
	`mr2_done <= 1'b0;`
	`ub_cal_now <= 1'b0;`
	`treg_TM_write_start <= 0;`

`setMR(MR3);`	`setMR(mr2_pass2 ? ddr_mp.after_cal_mr3 : ddr_mp.during_cal_mr3);`
`setMR(MR6);`	`setMR(mr2_pass2 ? ddr_mp.after_cal_mr6 : ddr_mp.during_cal_mr6);`

`twiddle(tMRD, calStMR5);`	`if(MR6_count > 4) begin`
	`twiddle(tMRD, calStMR5);`
	`MR6_count <= 0;`
	`end else begin`
	`twiddle(tMRD, calStMR6);`
	`MR6_count <= MR6_count +1;`
	`End`

`if (LRDIMM_QUAD_RANK) begin`
`if (cs_mask == 8'b0001) setMR(MR5_0);`
`else if (cs_mask == 8'b0010) setMR(MR5_1);`
`else if (cs_mask == 8'b0100) setMR(MR5_2);`
`else setMR(MR5_3);`
`end`
`else begin`
`setMR(MR5);`
`end`
	`/* if (LRDIMM_QUAD_RANK) begin`
	`if (cs_mask == 8'b0001) setMR(MR5_0);`
	`else if (cs_mask == 8'b0010) setMR(MR5_1);`
	`else if (cs_mask == 8'b0100) setMR(MR5_2);`
	`else setMR(MR5_3);`
	`end`
	`else begin`
	`setMR(MR5);`
	`end`
	`*/`
	`setMR(mr2_pass2 ? ddr_mp.after_cal_mr5 : ddr_mp.during_cal_mr5);`

`setDDROP(MRS);`	`setDDROP(MRS);`
`setMR(MR4);`	`setMR(mr2_pass2 ? ddr_mp.after_cal_mr4 : ddr_mp.during_cal_mr4);`
`setMR(MR2);`	`setMR(mr2_pass2 ? ddr_mp.after_cal_mr2 : ddr_mp.during_cal_mr2);`

`if ((LRDIMM_EN == 0) && (SLOT0_CONFIG == 8'b1111 || SLOT1_CONFIG == 8'b1111)) begin // If Single slot Quad Rank`
`if (cs_mask == 8'b0001) setMR(MR1_0);`
`else if (cs_mask == 8'b0010) setMR(MR1_1);`
`else if (cs_mask == 8'b0100) setMR(MR1_2);`
`else setMR(MR1_3);`
`end`
`else begin`
	`/* if ((LRDIMM_EN == 0) && (SLOT0_CONFIG == 8'b1111 || SLOT1_CONFIG == 8'b1111)) begin`
	`if (cs_mask == 8'b0001) setMR(MR1_0);`
	`else if (cs_mask == 8'b0010) setMR(MR1_1);`
	`else if (cs_mask == 8'b0100) setMR(MR1_2);`
	`else setMR(MR1_3);`
	`end`
	`else begin`
	`setMR(MR1);`

setMR(MR1); end	end */ setMR(mr2_pass2 ? ddr_mp.after_cal_mr1 : ddr_mp.during_cal_mr1);
setDDROP(MRS);	setDDROP(MRS);
setMR(MR0);	setMR(mr2_pass2 ? ddr_mp.after_cal_mr0 : ddr_mp.during_cal_mr0);

(null)	if (mr2_pass2 == 1) begin
	twiddle(tMOD, calStZQCL);
	end
	else begin

(null)	twiddle(tMOD, calStMR3); end

if (mrs_done)
 	
twiddle(tZQINIT, calStGOGO);
	if (mrs_done) begin
	if (mr2_pass2 == 1'b0) begin
	mr2_pass2 <= 1'b1;
	twiddle(tZQINIT, calStGOGO);
	end else begin
	mr2_done <= 1'b1;
	twiddle(tZQINIT, calStTREG);
	end
	end

calStGOGO: initDone <= 1'b1; // Now we are ready for
operations
	calStGOGO: begin
	if (mr2_pass2 == 1'b1 && mr2_done == 1'b0) begin
	ub_cal_now <= 1'b1;
	if (BYPASS_CAL == "TRUE" || ub_calDone == 1'b1)
	twiddle(tZQINIT, MEM=="DDR3" ? calStMR2 : calStMR1);
	end
	else initDone <= 1'b1; // RTL initialization complete
	end
	
	calStTREG: twiddle(tZQINIT, calStGOGO);

DDR4_v2_2_mc.sv

Standard Xilinx MIG	Modified MRAM MIG

(null)	// begin Status and control inputs and outputs
	,input power_fail_has_scramed
	,output reg cntr_power_fail_complete
	,output reg inflight_writes
	,input inflight_writes_ui
	// end Status and control inputs and outputs

(null)	,input [DBAW-1:0] fi_xor_wrdata_bufaddr
wire [3:0] tRASF;	wire [6:0] tRASF;
(null)	wire apgr; wire [3:0] pages_open;
(null)	reg scram_writing_done_pulse;

	reg scram_writing_done_pulsed; reg scram_refReq_started; reg per_state_idle; reg [3:0] clks_counter; reg [DBAW-1:0] dBufAdr_value_seen;

(null)	// begin Status and control outputs
	
	always @(posedge clk) begin
	if(rst_r1) begin
	inflight_writes <= 'b0;
	scram_writing_done_pulse <= 'b0;
	scram_writing_done_pulsed <= 'b0;
	scram_refReq_started <= 'b0;
	cntr_power_fail_complete <= 'b0;
	end
	else begin
	inflight_writes <= (~&txn_fifo_empty
	|| ~&cas_fifo_empty
	|| |pages_open);
	if (power_fail_has_scramed
	&& !(~&txn_fifo_empty
	|| ~&cas_fifo_empty
	|| inflight_writes_ui)
	&& !scram_writing_done_pulse
	&& !scram_writing_done_pulsed
	&& !scram_refReq_started
	&& per_state_idle
	&& !cntr_power_fail_complete) begin
	scram_writing_done_pulse <= 1'b1;
	end
	else if (scram_writing_done_pulse) begin
	scram_writing_done_pulse <= 'b0;
	scram_writing_done_pulsed <= 1'b1;
	end
	else if (scram_writing_done_pulsed
	&& !cntr_power_fail_complete
	&& refReq) begin
	scram_writing_done_pulsed <= 'b0;
	scram_refReq_started <= 'b1;
	end
	else if (scram_refReq_started
	&& !cntr_power_fail_complete
	&& !refReq) begin
	scram_refReq_started <= 'b0;
	cntr_power_fail_complete <= 1'b1;
	end
	else if (cntr_power_fail_complete
	&& !power_fail_has_scramed) begin

	cntr_power_fail_complete <= 'b0;
	end
	end
	end
	
	// end Status and control outputs
	
	assign apgr = ap || (col == 7'h78);

,.ap (ap)	,.ap (apgr)
(null)	,.pages_open (pages_open[bg])
(null)	,.scram_writing_done_pulse (scram_writing_done_pulse)
wire [31:0] periodic_config = (PER_RD_INTVL == 0) ? 32'b0 : { 30'b0, calDone, calDone };	wire [31:0] periodic_config = ((PER_RD_INTVL == 0) || cntr_power_fail_complete || power_fail_has_scramed) ? 32'b0 : { 30'b0, calDone, calDone };
(null)	,.per_state_idle (per_state_idle)

(null)	reg [2:0] cmd_saved;
	
	always @ (posedge clk) cmd_saved <= cmd;
	wire a_mc_003 = useAdr
	&& accept
	&& ((cmd_saved == 3'h0) || (cmd_saved == 3'h3))
	&& |clks_counter
	&& (dBufAdr_value_seen == dBufAdr);
	
	always @ (posedge clk) begin
	if (rst_r1) begin
	clks_counter <= 'b0;
	dBufAdr_value_seen <= 'b0;
	end
	else if(useAdr
	&& accept
	&& (cmd_saved == 3'h3)) begin
	assert property (~a_mc_003);
	clks_counter <= 4'b1;
	dBufAdr_value_seen <= dBufAdr;
	end
	else if (clks_counter > 4'h4) begin
	clks_counter <= 'b0;
	end
	else if (|clks_counter) begin
	clks_counter <= clks_counter + 1'b1;
	end
	end

DDR4_v2_2_mc_arb_c.sv

Standard Xilinx MIG	Modified MRAM MIG

always @(*) begin
  w10 = findWin(last10, reqs[1:0]);
  w32 = findWin(last32, reqs[3:2]);
  winner = findWin(last, {|reqs[3:2], |reqs[1:0]});
  case (winner)
  2'b01: win3210_reorder = {2'b00, w10};
  2'b10: win3210_reorder = {w32, 2'b00};
  default: win3210_reorder = 4'b0000;
  endcase
end
 	
// Select arbitration winner based on ordering mode
	always @(*) begin
	  w10 = findWin(last10, reqs[1:0]);
	  w32 = findWin(last32, reqs[3:2]);
	  if (|(winPort & reqs)) begin
	  win3210_reorder = winPort;
	  end else begin
	  winner = findWin(last, {|reqs[3:2], |reqs[1:0]});
	  case (winner)
	  2'b01: win3210_reorder = {2'b00, w10};
	  2'b10: win3210_reorder = {w32, 2'b00};
	  default: win3210_reorder = 4'b0000;
	  endcase
	  end
	end

DDR4_v2_2_mc_arb_mux_p.sv

Standard Xilinx MIG	Modified MRAM MIG
,output [3:0] tRASF	,output [6:0] tRASF
wire [3:0] tRAS_TEMP = ((tRAS + 3) / 4) - 2;	wire [6:0] tRAS_TEMP = ((tRAS + 3) / 4) - 2;

DDR4_v2_2_mc_group.sv

Standard Xilinx MIG	Modified MRAM MIG
,input [3:0] tRASF	,input [6:0] tRASF
(null)	,output reg pages_open
localparam TXN_FIFO_DEPTH = 4;	localparam TXN_FIFO_DEPTH = 16;
localparam TXN_FIFO_PWIDTH = 2;	localparam TXN_FIFO_PWIDTH = 4;
localparam CAS_FIFO_DEPTH = 4;	localparam CAS_FIFO_DEPTH = 16;
localparam CAS_FIFO_PWIDTH = 2;	localparam CAS_FIFO_PWIDTH = 4;
reg [3:0] trcd_cntr [3:0][S_HEIGHT_ALIASED-1:0][3:0];	reg [6:0] trcd_cntr [3:0][S_HEIGHT_ALIASED-1:0][3:0];
reg [3:0] trcd_cntr_nxt [3:0][S_HEIGHT_ALIASED-1:0][3:0];	reg [6:0] trcd_cntr_nxt [3:0][S_HEIGHT_ALIASED-1:0][3:0];
reg [4:0] trp_cntr;	reg [6:0] trp_cntr;
reg [3:0] tras_cntr_rb [3:0][S_HEIGHT_ALIASED-1:0][3:0];	reg [6:0] tras_cntr_rb [3:0][S_HEIGHT_ALIASED-1:0][3:0];
reg [3:0] tras_cntr_rb_nxt [3:0][S_HEIGHT_ALIASED-1:0][3:0];	reg [6:0] tras_cntr_rb_nxt [3:0][S_HEIGHT_ALIASED-1:0][3:0];
(null)	reg pages_open_nxt;
wire buf_fifo_push = inc_txn_fifo_wptr;	Removed

(null)	wire txn_fifo_full_nxt;
	wire buf_fifo_push = inc_txn_fifo_wptr_accept_useadr | select_periodic_read;
	wire [DBAW-1:0] buf_fifo_input = buf_fifo_push ? dBufAdr : buf_fifo[txn_fifo_wptr];

buf_fifo_nxt[txn_fifo_wptr] = buf_fifo_push ? dBufAdr : buf_fifo[txn_fifo_wptr];	buf_fifo_nxt[txn_fifo_wptr] = buf_fifo_input;
wire txn_fifo_full_nxt = (txn_fifo_wptr - txn_fifo_rptr) >= TXN_FIFO_FULL_THRESHOLD[TXN_FIFO_PWIDTH-1:0];	assign txn_fifo_full_nxt = (txn_fifo_wptr - txn_fifo_rptr) >= TXN_FIFO_FULL_THRESHOLD[TXN_FIFO_PWIDTH-1:0];
wire rd_req_nxt = (rdReqR | set_rd_req) & ~cas_won;	wire rd_req_nxt = (rdReqR & ~cas_won) | set_rd_req;
wire wr_req_nxt = (wrReqR | set_wr_req) & ~cas_won;	wire wr_req_nxt = (wrReqR & ~cas_won) | set_wr_req;
(null)	wire [1:0] cmd_group_cas_nxt; wire [1:0] cmd_rank_cas_nxt; wire [1:0] cmd_bank_cas_nxt; wire [ABITS-1:0] cmd_row_cas_nxt; wire cas_pend_output_valid; wire cas_page_hit_for_req_nxt; assign cas_page_hit_for_req_nxt = cas_pend_output_valid && !cmdAP && (cmd_row_cas == cmd_row_cas_nxt) && (cmd_rank_cas == cmd_rank_cas_nxt) && (cmd_group_cas == cmd_group_cas_nxt) && (cmd_bank_cas == cmd_bank_cas_nxt);
rdReq = cas_won ? 1'b0 : rdReqR;	rdReq = cas_won ? (rdReqR && rd_req_nxt & cas_page_hit_for_req_nxt) : rdReqR;
wrReq = cas_won ? 1'b0 : wrReqR;	wrReq = cas_won ? (wrReqR && wr_req_nxt & cas_page_hit_for_req_nxt) : wrReqR;
wire [TXN_FIFO_WIDTH-1:0] cas_pend_fifo_output = (cas_fifo_empty & (CAS_FIFO_BYPASS == "ON")) ? cas_pend_fifo_input : cas_pend_fifo[cas_fifo_rptr];	wire [TXN_FIFO_WIDTH-1:0] cas_pend_fifo_output; assign cas_pend_fifo_output = (cas_fifo_empty && (CAS_FIFO_BYPASS == "ON")) ? cas_pend_fifo_input : ((!cas_fifo_empty && cas_pend_fifo_pop && cas_fifo_valid[cas_fifo_rptr_nxt]) ? cas_pend_fifo[cas_fifo_rptr_nxt] : cas_pend_fifo[cas_fifo_rptr]); assign cas_pend_output_valid = (cas_fifo_empty && (CAS_FIFO_BYPASS == "ON")) ? cas_pend_fifo_push : ((!cas_fifo_empty && cas_pend_fifo_pop && cas_fifo_valid[cas_fifo_rptr_nxt]) ? cas_fifo_valid[cas_fifo_rptr_nxt] : cas_fifo_valid[cas_fifo_rptr] && !cas_won);

wire [DBAW-1:0] cas_dbuf_output = cas_dbuf_fifo[cas_fifo_rptr];	wire [DBAW-1:0] cas_dbuf_output; assign cas_dbuf_output = (!cas_fifo_empty && cas_pend_fifo_pop && cas_fifo_valid[cas_fifo_rptr_nxt]) ? cas_dbuf_fifo[cas_fifo_rptr_nxt] : cas_dbuf_fifo[cas_fifo_rptr];
wire [1:0] cmd_group_cas_nxt = cas_pend_output_group;	assign cmd_group_cas_nxt = cas_pend_output_group;
wire [1:0] cmd_rank_cas_nxt = cas_pend_output_rank;	assign cmd_rank_cas_nxt = cas_pend_output_rank;
wire [1:0] cmd_bank_cas_nxt = cas_pend_output_bank;	assign cmd_bank_cas_nxt = cas_pend_output_bank;
wire [ABITS-1:0] cmd_row_cas_nxt = cas_pend_output_row;	assign cmd_row_cas_nxt = cas_pend_output_row;

gr_cas_state_nxt = CAS_IDLE;	if (~cas_fifo_empty
	&& cas_fifo_valid[cas_fifo_rptr_nxt]
	&&
	trcd_cntr_is_zero[cmd_rank_cas_nxt][cmd_l_rank_cas_nxt_3ds][cmd_bank_cas_nxt]
	&& ((cmd_cmd_nxt == NATRD)
	|| (cmd_cmd_nxt == NATWR))) begin
	set_rd_req = (cmd_cmd_nxt == NATRD);
	set_wr_req = (cmd_cmd_nxt == NATWR);
	gr_cas_state_nxt = CAS_WAIT;
	end
	else begin
	gr_cas_state_nxt = CAS_IDLE;
	set_rd_req = 1'b0;
	set_wr_req = 1'b0;
	end

wire [4:0] trp_cntr_nxt = (grSt == grPREWAIT) ? tRPF : // spyglass disable W164c	wire [6:0] trp_cntr_nxt = (grSt == grPREWAIT) ? tRPF : // spyglass disable W164c
wire [4:0] tras_cntr_extend = { 1'b0, tras_cntr_rb[cmd_rank_cas][cmd_l_rank_cas_3ds][cmd_bank_cas] };	wire [7:0] tras_cntr_extend = { 1'b0, tras_cntr_rb[cmd_rank_cas][cmd_l_rank_cas_3ds][cmd_bank_cas] };
(null)	pages_open_nxt = 1'b0;
(null)	pages_open_nxt = pages_open_nxt || pageInfo[rank_index][lr_index][bank_index][0];
(null)	pages_open <= #TCQ '0;
(null)	pages_open <= #TCQ pages_open_nxt;
wire [3:0] e_trcd_cntr_0_0 = trcd_cntr[0][0][0];	wire [7:0] e_trcd_cntr_0_0 = trcd_cntr[0][0][0];
wire [3:0] e_trcd_cntr_0_1 = trcd_cntr[0][0][1];	wire [7:0] e_trcd_cntr_0_1 = trcd_cntr[0][0][1];
wire [3:0] e_trcd_cntr_0_2 = trcd_cntr[0][0][2];	wire [7:0] e_trcd_cntr_0_2 = trcd_cntr[0][0][2];
wire [3:0] e_trcd_cntr_0_3 = trcd_cntr[0][0][3];	wire [7:0] e_trcd_cntr_0_3 = trcd_cntr[0][0][3];

DDR4_v2_2_mc_ctl.sv

Standard Xilinx MIG	Modified MRAM MIG
wire group_unchanged = ((MEM == "DDR4") & (BGBITS == 1)) ? (winGroupC == prevGroup) : 1'b0;	wire group_unchanged = 1'b0;

DDR4_v2_2_mc_ref.sv

Standard Xilinx MIG	Modified MRAM MIG
(null)	,input scram_writing_done_pulse
(null)	localparam REFI_DISABLE = (tREFI == 17'h1ffff);
for (i = 0; i < RANKS; i = i + 1) inc_pend_ref_due[i] = ~USER_MODE & (refi[i] == '0);	for (i = 0; i < RANKS; i = i + 1) inc_pend_ref_due[i] = (~USER_MODE & ~REFI_DISABLE & (refi[i] == '0)) || scram_writing_done_pulse;
for (i = 0; i < RANKS; i = i + 1) refi_nxt[i] = { 15 { ~USER_MODE } } & ((refi[i] == '0) ? tREFIF : (refi[i] - { 14'b0, calDone }));	for (i = 0; i < RANKS; i = i + 1) refi_nxt[i] = { 15 { ~USER_MODE } } & ((REFI_DISABLE || (refi[i] == '0)) ? tREFIF : (refi[i] - { 14'b0, calDone }));
wire zq_intvl_load = ~(| zq_intvl_count);	wire zq_intvl_load = 0; // Never do ZQCL

DDR4_v2_2_ui.sv

Standard Xilinx MIG	Modified MRAM MIG
zq_req, app_zq_ack, ui_busy,	zq_req, app_zq_ack, ui_busy, inflight_writes,
(null)	output reg inflight_writes;
wire [3:0] ram_init_addr; // From ui_rd_data0 of ui_rd_data.sv	wire [DATA_BUF_ADDR_WIDTH-1:0] ram_init_addr;
wire [3:0] ram_init_done_r; // From ui_rd_data0 of ui_rd_data.sv	wire [DATA_BUF_ADDR_WIDTH-1:0] ram_init_done_r;
(null)	wire wrdata_fifo_empty;

if(DATA_BUF_ADDR_WIDTH > 4) begin
assign wr_data_buf_addr[DATA_BUF_ADDR_WIDTH-1:4] = 0;
	if(DATA_BUF_ADDR_WIDTH > 6) begin
	assign wr_data_buf_addr[DATA_BUF_ADDR_WIDTH-1:6] = 0;
	assign data_buf_addr[DATA_BUF_ADDR_WIDTH-1:6] = 0;

(null)	always @(posedge clk) begin
	if (rst_r1) begin
	inflight_writes <= 'b0;
	end
	else begin
	inflight_writes <= !(wrdata_fifo_empty);
	end
	end

.data_buf_addr (data_buf_addr),	.data_buf_addr (data_buf_addr[DATA_BUF_ADDR_WIDTH-1:0]),
.wr_data_buf_addr (wr_data_buf_addr),	.wr_data_buf_addr (wr_data_buf_addr[DATA_BUF_ADDR_WIDTH-1:0]),
.rd_data_buf_addr_r (rd_data_buf_addr_r),	.rd_data_buf_addr_r (rd_data_buf_addr_r[DATA_BUF_ADDR_WIDTH-1:0]),

(null)	*.FIFO_ADDR_WIDTH (DATA_BUF_ADDR_WIDTH),*
.wr_req_16 (wr_req_16),	*.wr_req_full (wr_req_16),*
(null)	*.wrdata_fifo_empty (wrdata_fifo_empty),*
.wr_data_buf_addr (wr_data_buf_addr[3:0]),	*.wr_data_buf_addr (wr_data_buf_addr[DATA_BUF_ADDR_WIDTH-1:0]),*
.wr_data_addr (wr_data_addr[3:0]),	*.wr_data_addr (wr_data_addr[DATA_BUF_ADDR_WIDTH-1:0]),*
.rd_data_buf_addr_r (rd_data_buf_addr_r),	*.rd_data_buf_addr_r (rd_data_buf_addr_r[DATA_BUF_ADDR_WIDTH-1:0]),*
.rd_data_addr (rd_data_addr),	*.rd_data_addr (rd_data_addr[DATA_BUF_ADDR_WIDTH-1:0]),*

DDR4_v2_2_ui_cmd.sv

Standard Xilinx MIG	**Modified MRAM MIG**
else if (MEM_ADDR_ORDER == "ROW_BANK_COLUMN" && MEM == "DDR4") begin *assign col = app_rdy_r ? app_addr_r1[0+:COL_WIDTH] : app_addr_r2[0+:COL_WIDTH];* *assign bank = app_rdy_r ? app_addr_r1[COL_WIDTH+:BANK_WIDTH] : app_addr_r2[COL_WIDTH+:BANK_WIDTH];* *assign group = app_rdy_r ? app_addr_r1[COL_WIDTH+BANK_WIDTH+:BANK_GROUP_WIDTH] : app_addr_r2[COL_WIDTH+BANK_WIDTH+:BANK_GROUP_WIDTH];* *assign row = app_rdy_r ? app_addr_r1[COL_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:ROW_WIDTH] : app_addr_r2[COL_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:ROW_WIDTH];* *if (S_HEIGHT == 1)* *assign lr = 'b0;* *else* *assign lr = app_rdy_r ? app_addr_r1[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:LR_WIDTH] : app_addr_r2[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:LR_WIDTH];* *if (RANKS == 1)* *assign rank = 'b0;*	*else if (MEM_ADDR_ORDER == "ROW_BANK_COLUMN" && MEM == "DDR4") begin* *assign col = app_rdy_r ? app_addr_r1[0+:COL_WIDTH] : app_addr_r2[0+:COL_WIDTH];* *assign group = app_rdy_r ? app_addr_r1[COL_WIDTH+:BANK_GROUP_WIDTH] : app_addr_r2[COL_WIDTH+:BANK_GROUP_WIDTH];* *assign bank = app_rdy_r ? app_addr_r1[COL_WIDTH+BANK_GROUP_WIDTH+:BANK_WIDTH] : app_addr_r2[COL_WIDTH+BANK_GROUP_WIDTH+:BANK_WIDTH];* *assign row = app_rdy_r ? app_addr_r1[COL_WIDTH+BANK_GROUP_WIDTH+BANK_WIDTH+:ROW_WIDTH] : app_addr_r2[COL_WIDTH+BANK_GROUP_WIDTH+BANK_WIDTH+:ROW_WIDTH];* *if (S_HEIGHT == 1)* *assign lr = 'b0;* *else* *assign lr = app_rdy_r ? app_addr_r1[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:LR_WIDTH] : app_addr_r2[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:LR_WIDTH];* *if (RANKS == 1)* *assign rank = 'b0;* *else if (S_HEIGHT == 1)* *assign rank = app_rdy_r*

```
      else if (S_HEIGHT == 1)
        assign rank = app_rdy_r
                    ?
app_addr_r1[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:RANK_WIDTH]
                    :
app_addr_r2[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:RANK_WIDTH];
      else
        assign rank = app_rdy_r
                    ?
app_addr_r1[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+LR_WIDTH+:RANK_WIDTH]
                    :
app_addr_r2[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+LR_WIDTH+:RANK_WIDTH];
    end
    // Addressing with ROW - COLUMN - BANK for DDR3
```

```
                    ?
app_addr_r1[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:RANK_WIDTH]
                    :
app_addr_r2[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+:RANK_WIDTH];
      else
        assign rank = app_rdy_r
                    ?
app_addr_r1[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+LR_WIDTH+:RANK_WIDTH]
                    :
app_addr_r2[COL_WIDTH+ROW_WIDTH+BANK_WIDTH+BANK_GROUP_WIDTH+LR_WIDTH+:RANK_WIDTH];
    end  // end of patch

    else if (MEM_ADDR_ORDER ==  "ROW_BANK_COLUMN_BANK" && MEM == "DDR3") begin
      // Copy and modify Addressing with ROW - BANK - COLUMN for DDR3
      // original ROW_BANK_COLUMN code:
      // assign col = app_rdy_r ? app_addr_r1[0+:COL_WIDTH] : app_addr_r2[0+:COL_WIDTH];
      // assign row = app_rdy_r ? app_addr_r1[COL_WIDTH+BANK_WIDTH+:ROW_WIDTH]
      //                        : app_addr_r2[COL_WIDTH+BANK_WIDTH+:ROW_WIDTH];
      // assign bank = app_rdy_r ? app_addr_r1[COL_WIDTH+:BANK_WIDTH]
      //                         : app_addr_r2[COL_WIDTH+:BANK_WIDTH];
      //
      // COL_WIDTH  = 6
      // BANK_WIDTH = 3
      // ROW_WIDTH  = 16
      // RANKS      = 1
      //
      // 2222222221111111111
      // 98765432109876543210987654321O
      //     rrrrrrrrrrrrrrrbbcccbccc
      //                01   2
      //
      // For best performance,
      // bank[0] must be the slowest to change
      // bank[2:1] must be the fastest to change
      // The controller uses bank[2:1] to select the group it uses.
      //
      // synthesis translate_off
      // $error("Need to improve this to use parameters for widths");
```

	`if (BANK_WIDTH != 3) begin`
	`$error("Need to determine which bank bits to use for other cases");`
	`end`
	`if (RANKS != 1) begin`
	`$error("Need to determine which rank bits to use for other cases");`
	`end`
	`// synthesis translate_on`
	`assign col = app_rdy_r ? {app_addr_r1[6:4], app_addr_r1[2:0]} : {app_addr_r2[6:4], app_addr_r2[2:0]};`
	`assign row = app_rdy_r ? app_addr_r1[24:9] : app_addr_r2[24:9];`
	`assign bank = app_rdy_r ? {app_addr_r1[3], app_addr_r1[7], app_addr_r1[8]} : {app_addr_r2[3], app_addr_r2[7], app_addr_r2[8]} ;`
	`assign rank = 'b0;`
	`assign group = 'b0;`
	`assign lr = 'b0;`
	`end`

DDR4_v2_2_ui_rd_data.sv

Standard Xilinx MIG	**Modified MRAM MIG**
`output wire [3:0] ram_init_done_r,`	`output wire [DATA_BUF_ADDR_WIDTH-1:0] ram_init_done_r,`
`output wire [3:0] ram_init_addr,`	`output wire [DATA_BUF_ADDR_WIDTH-1:0] ram_init_addr,`
(null)	`localparam DATA_BUF_DEPTH = 32'b1 << DATA_BUF_ADDR_WIDTH;`
`reg [5:0] rd_buf_indx_r [0:19];`	`reg [DATA_BUF_ADDR_WIDTH:0] rd_buf_indx_r [0:DATA_BUF_DEPTH-1];`

`assign ram_init_done_r[0] = ram_init_done_r_lcl[0];`
`assign ram_init_done_r[1] = ram_init_done_r_lcl[1];`
`assign ram_init_done_r[2] = ram_init_done_r_lcl[2];`
`assign ram_init_done_r[3] = ram_init_done_r_lcl[3];`
	Removed

(null)	`Assign ram_init_done_r[DATA_BUF_ADDR_WIDTH-1:0] = ram_init_done_r_lcl[DATA_BUF_ADDR_WIDTH-1:0];`
`assign ram_init_addr = rd_buf_indx_r[2][3:0];`	`assign ram_init_addr = rd_buf_indx_r[2][DATA_BUF_ADDR_WIDTH-1:0];`

`generate`
`genvar j;`
`wire upd_rd_buf_indx = (ram_init_done_r_lcl[4] ? ((ORDERING == "NORM") && (bypass_cpy || rd_data_rdy_cpy)) : 1'b1);`
 	
`always @(posedge clk)`
`if (rst)`
`ram_init_done_r_lcl <= #TCQ 8'h00;`
`else if (rd_buf_indx_r[0][4:0] == 5'h1f)`
`ram_init_done_r_lcl <= #TCQ 8'hFF;`
	`wire end_loop_flag;`
	`assign end_loop_flag = rd_buf_indx_r[0][DATA_BUF_ADDR_WIDTH-1:0] == {DATA_BUF_ADDR_WIDTH{1'b1}};`
	
	`generate`
	`genvar j;`
	`wire upd_rd_buf_indx = (ram_init_done_r_lcl[DATA_BUF_ADDR_WIDTH] ? ((ORDERING == "NORM") && (bypass_cpy || rd_data_rdy_cpy)) : 1'b1);`

```
    for (j=0; j<20; j=j+1) begin : rd_buf_index_cpy
      always @(posedge clk) begin
        if (rst) rd_buf_indx_r[j] <= #TCQ 'b0;
        else if (upd_rd_buf_indx) rd_buf_indx_r[j] <= #TCQ
          rd_buf_indx_r[j] + 6'h1 + (DATA_BUF_ADDR_WIDTH
== 5 ? 0 :
                          (single_data && ~rd_buf_indx_r[j][0]));
      end
    end
  endgenerate

// Compute dimensions of read data buffer.  Depending on
width of
// DQ bus and DRAM CK
// to fabric ratio, number of RAM32Ms is variable.  RAM32Ms
are used in
// single write, single read, 6 bit wide mode.
  localparam RD_BUF_WIDTH = APP_DATA_WIDTH + (ECC ==
"OFF" ? 0 : 2*nCK_PER_CLK);
  localparam FULL_RAM_CNT = (RD_BUF_WIDTH/6);
  localparam REMAINDER = RD_BUF_WIDTH % 6;
  localparam RAM_CNT = FULL_RAM_CNT + ((REMAINDER ==
0 ) ? 0 : 1);
  localparam RAM_WIDTH = (RAM_CNT*6);

// STRICT MODE
  generate
    if (ORDERING == "STRICT") begin : strict_mode
      assign single_data = 1'b0;
      assign rd_buf_full = 1'b0;
      reg [DATA_BUF_ADDR_WIDTH-1:0]
rd_data_buf_addr_r_lcl;
      reg [APP_DATA_WIDTH-1:0] rd_data_r;
      wire [DATA_BUF_ADDR_WIDTH-1:0]
rd_data_buf_addr_ns =
                rst
                 ? 0
                 : rd_data_buf_addr_r_lcl + rd_accepted;
      always @(posedge clk) rd_data_buf_addr_r_lcl <=
                        #TCQ rd_data_buf_addr_ns;
      assign rd_data_buf_addr_r = rd_data_buf_addr_ns;
// app_* signals required to be registered.
      if (ECC == "OFF") begin : ecc_off
        assign app_rd_data = rd_data;
        always @(/*AS*/rd_data_en) app_rd_data_valid =
rd_data_en;
      always @(posedge clk)
        if (rst)
          ram_init_done_r_lcl <= #TCQ 8'h00;
        else if (end_loop_flag)
          ram_init_done_r_lcl <= #TCQ 8'hFF;

    if (DATA_BUF_ADDR_WIDTH < 6) begin
      for (j=0; j<20; j=j+1) begin : rd_buf_index_cpy
        always @(posedge clk) begin
          if (rst) rd_buf_indx_r[j] <= #TCQ 'b0;
          else if (upd_rd_buf_indx) rd_buf_indx_r[j] <= #TCQ
            rd_buf_indx_r[j] + 6'h1 + (DATA_BUF_ADDR_WIDTH == 5 ? 0 :
(single_data && ~rd_buf_indx_r[j][0]));
        end
      end
    end
    else if (DATA_BUF_ADDR_WIDTH == 6) begin
      for (j=0; j<DATA_BUF_DEPTH; j=j+1) begin : rd_buf_index_cpy
        always @(posedge clk) begin
          if (rst) rd_buf_indx_r[j] <= #TCQ 'b0;
          else if (upd_rd_buf_indx) rd_buf_indx_r[j] <= #TCQ
            rd_buf_indx_r[j] + 6'h1 + (DATA_BUF_ADDR_WIDTH == 6 ? 0 :
                          (single_data && ~rd_buf_indx_r[j][0]));
        end
      end
    end
  endgenerate


// Compute dimensions of read data buffer.  Depending on width of
// DQ bus and DRAM CK
// to fabric ratio, number of RAM32Ms is variable.  RAM32Ms are used
in
// single write, single read, 6 bit wide mode.
  localparam RD_BUF_WIDTH = APP_DATA_WIDTH + (ECC == "OFF" ? 0 :
2*nCK_PER_CLK);
  localparam FULL_RAM_CNT = (RD_BUF_WIDTH/6);
  localparam REMAINDER = RD_BUF_WIDTH % 6;
  localparam RAM_CNT = FULL_RAM_CNT + ((REMAINDER == 0 ) ? 0 : 1);
  localparam RAM_WIDTH = (RAM_CNT*6);

// STRICT MODE
  generate
    if (ORDERING == "STRICT") begin : strict_mode
      assign single_data = 1'b0;
      assign rd_buf_full = 1'b0;
      reg [DATA_BUF_ADDR_WIDTH-1:0] rd_data_buf_addr_r_lcl;
      reg [APP_DATA_WIDTH-1:0] rd_data_r;
```

```
      always @(/*AS*/rd_data_end) app_rd_data_end =
rd_data_end;
    end
    else begin : ecc_on
      assign app_rd_data = rd_data_r;
      always @(posedge clk) rd_data_r <= #TCQ rd_data;
      always @(posedge clk) app_rd_data_valid <= #TCQ
rd_data_en;
      always @(posedge clk) app_rd_data_end <= #TCQ
rd_data_end;
      always @(posedge clk) app_ecc_multiple_err_r <= #TCQ
ecc_multiple;
    end
  end

// NON-STRICT MODE
// In configurations where read data is returned in a single
fabric cycle
// the offset is always zero and we can use the bit to get a
deeper
// FIFO. The RAMB32 has 5 address bits, so when the
DATA_BUF_ADDR_WIDTH
// is set to use them all, discard the offset. Otherwise, include
the
// offset.
  else begin : not_strict_mode
    genvar k;
    reg [5:0]  rd_buf_indx_sts_r [0:3];
    (* keep = "true" *) reg       rd_buf_we_r1;
    (* keep = "true" *) reg [3:0]  ram_init_done_r_lcl_sts;
    reg [3:0]  upd_rd_buf_indx_sts;
    wire [1:0] rd_status[0:6];
    wire [3:0] address_match_sts_0;
    wire [3:0] address_match_sts_1;
    wire [3:0] bypass_sts;
    wire [3:0] app_rd_data_end_sts;
    wire [3:0] single_data_sts;
    wire [3:0] rd_buf_we_sts;
    wire [4:0] rd_buf_wr_addr_sts [0:3];
    wire [3:0] rd_data_rdy_sts;


    wire [4:0] rd_buf_wr_addr = (DATA_BUF_ADDR_WIDTH
== 5) ? rd_data_addr[4:0] :
```

```
    wire [DATA_BUF_ADDR_WIDTH-1:0] rd_data_buf_addr_ns =
            rst
             ? 0
             : rd_data_buf_addr_r_lcl + rd_accepted;
    always @(posedge clk) rd_data_buf_addr_r_lcl <=
                    #TCQ rd_data_buf_addr_ns;
    assign rd_data_buf_addr_r = rd_data_buf_addr_ns;
// app_* signals required to be registered.
    if (ECC == "OFF") begin : ecc_off
      assign app_rd_data = rd_data;
      always @(/*AS*/rd_data_en) app_rd_data_valid = rd_data_en;
      always @(/*AS*/rd_data_end) app_rd_data_end = rd_data_end;
    end
    else begin : ecc_on
      assign app_rd_data = rd_data_r;
      always @(posedge clk) rd_data_r <= #TCQ rd_data;
      always @(posedge clk) app_rd_data_valid <= #TCQ rd_data_en;
      always @(posedge clk) app_rd_data_end <= #TCQ rd_data_end;
      always @(posedge clk) app_ecc_multiple_err_r <= #TCQ
ecc_multiple;
    end
  end

 // NON-STRICT MODE
// In configurations where read data is returned in a single fabric cycle
// the offset is always zero and we can use the bit to get a deeper
// FIFO. The RAMB32 has 5 address bits, so when the
DATA_BUF_ADDR_WIDTH
// is set to use them all, discard the offset. Otherwise, include the
// offset.
  else begin : not_strict_mode
    genvar k;
    reg [DATA_BUF_ADDR_WIDTH:0]  rd_buf_indx_sts_r [0:3]; // use one
bit wider
    (* keep = "true" *) reg       rd_buf_we_r1;
    (* keep = "true" *) reg [3:0]  ram_init_done_r_lcl_sts;
    reg [3:0]  upd_rd_buf_indx_sts;
    wire [DATA_BUF_ADDR_WIDTH-1:0] rd_status[0:6];
    wire [3:0] address_match_sts_0;
    wire [3:0] address_match_sts_1;
    wire [3:0] bypass_sts;
    wire [3:0] app_rd_data_end_sts;
    wire [3:0] single_data_sts;
    wire [3:0] rd_buf_we_sts;
    wire [DATA_BUF_ADDR_WIDTH-1:0] rd_buf_wr_addr_sts [0:3];
    wire [3:0] rd_data_rdy_sts;
    wire [DATA_BUF_ADDR_WIDTH-1:0] rd_buf_wr_addr
     = (DATA_BUF_ADDR_WIDTH == 6) ? rd_data_addr[5:0]
```

```
                                        {rd_data_addr[3:0],
rd_data_offset};

    for (k = 0; k < 4; k = k +1) begin : status_ram_signals

      assign address_match_sts_0[k] =
match6_1({rd_buf_wr_addr_sts[k][2:0],rd_buf_indx_sts_r[k][
2:0]});
      assign address_match_sts_1[k] =
match4_1({rd_buf_wr_addr_sts[k][4:3],rd_buf_indx_sts_r[k][
4:3]});
      assign bypass_sts[k] = rd_data_en &&
address_match_sts_0[k] && address_match_sts_1[k];
      assign app_rd_data_end_sts[k] = bypass_sts[k] ?
rd_data_end : rd_status[k][1];
      assign single_data_sts[k] = ram_init_done_r_lcl_sts[k]
&& app_rd_data_end_sts[k];
      assign rd_buf_we_sts[k] = ~ram_init_done_r_lcl_sts[k]
|| rd_data_en;
      assign rd_buf_wr_addr_sts[k] =
(DATA_BUF_ADDR_WIDTH == 5) ? rd_data_addr :
                                        {rd_data_addr,
rd_data_offset};
      assign rd_data_rdy_sts[k] = (rd_status[k][0] ==
rd_buf_indx_sts_r[k][5]);

      always @(*) begin
        casez
({ram_init_done_r_lcl_sts[k],address_match_sts_0[k],address
_match_sts_1[k],
              rd_data_en, rd_data_rdy_sts[k]})
          5'b0???? : upd_rd_buf_indx_sts[k] = 1'b1;
          5'b1???1 : upd_rd_buf_indx_sts[k] = 1'b1;
          5'b11110 : upd_rd_buf_indx_sts[k] = 1'b1;
          default : upd_rd_buf_indx_sts[k] = 1'b0;
        endcase
      end

      always @(posedge clk)
        if (rst)
          ram_init_done_r_lcl_sts[k] <= #TCQ 1'b0;
        else if (rd_buf_indx_sts_r[k][4:0] == 5'h1f)
          ram_init_done_r_lcl_sts[k] <= #TCQ 1'b1;

      always @(posedge clk) begin
        if (rst) rd_buf_indx_sts_r[k] <= #TCQ 'b0;
```

```
                    : (DATA_BUF_ADDR_WIDTH == 5) ?
rd_data_addr[4:0]
                                        : {rd_data_addr[3:0],
rd_data_offset};

    for (k = 0; k < 4; k = k +1) begin : status_ram_signals

      assign address_match_sts_0[k] =
match6_1({rd_buf_wr_addr_sts[k][2:0],rd_buf_indx_sts_r[k][2:0]});
      assign address_match_sts_1[k]
        = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr_sts[k][5:3],rd_buf_indx_sts_r[k][5:3]})
                    :
match4_1({rd_buf_wr_addr_sts[k][4:3],rd_buf_indx_sts_r[k][4:3]});

      assign bypass_sts[k] = rd_data_en && address_match_sts_0[k] &&
address_match_sts_1[k];
      assign app_rd_data_end_sts[k] = bypass_sts[k] ? rd_data_end :
rd_status[k][1];
      assign single_data_sts[k] = ram_init_done_r_lcl_sts[k] &&
app_rd_data_end_sts[k];
      assign rd_buf_we_sts[k] = ~ram_init_done_r_lcl_sts[k] ||
rd_data_en;

      assign rd_buf_wr_addr_sts[k] = (DATA_BUF_ADDR_WIDTH == 5 ||
DATA_BUF_ADDR_WIDTH == 6) ? rd_data_addr :
                                        {rd_data_addr, rd_data_offset};

      assign rd_data_rdy_sts[k] = (rd_status[k][0] ==
rd_buf_indx_sts_r[k][DATA_BUF_ADDR_WIDTH]);

      always @(*) begin
        casez
({ram_init_done_r_lcl_sts[k],address_match_sts_0[k],address_match_st
s_1[k],
              rd_data_en, rd_data_rdy_sts[k]})
          5'b0???? : upd_rd_buf_indx_sts[k] = 1'b1;
          5'b1???1 : upd_rd_buf_indx_sts[k] = 1'b1;
          5'b11110 : upd_rd_buf_indx_sts[k] = 1'b1;
          default : upd_rd_buf_indx_sts[k] = 1'b0;
        endcase
      end

      always @(posedge clk)
        if (rst)
          ram_init_done_r_lcl_sts[k] <= #TCQ 1'b0;
        else if (rd_buf_indx_sts_r[k][DATA_BUF_ADDR_WIDTH-1:0] ==
{DATA_BUF_ADDR_WIDTH{1'b1}})
```

```
        else if (upd_rd_buf_indx_sts[k]) rd_buf_indx_sts_r[k] <=
#TCQ
          rd_buf_indx_sts_r[k] + 6'h1 +
(DATA_BUF_ADDR_WIDTH == 5 ? 0 :
                            (single_data_sts[k] &&
~rd_buf_indx_sts_r[k][0]));
      end

    end

// Instantiate status RAM.  One bit for status and one for
"end".
// Turns out read to write back status is a timing path.
Update
// the status in the ram on the state following the read.
Bypass
// the write data into the status read path.
// Not guaranteed to write second status bit. If it is written,
always
// copy in the first status bit.
    begin : status_ram_0
      reg [4:0] status_ram_wr_addr_r;
      reg [1:0] status_ram_wr_data_r;
      reg wr_status_r1;
      wire [1:0] wr_status;
      wire [4:0] status_ram_wr_addr_ns =
ram_init_done_r_lcl_sts[0]
                            ? rd_buf_wr_addr_sts[0]
                            : rd_buf_indx_sts_r[0][4:0];
      wire [1:0] status_ram_wr_data_ns =
ram_init_done_r_lcl_sts[0] ?
                            {rd_data_end, ~(rd_data_offset
                                  ? wr_status_r1
                                  : wr_status[0])}
                            : 2'b0;
      always @(posedge clk)
        status_ram_wr_addr_r <= #TCQ
status_ram_wr_addr_ns;
      always @(posedge clk) wr_status_r1 <= #TCQ
wr_status[0];
      always @(posedge clk)
        status_ram_wr_data_r <= #TCQ
status_ram_wr_data_ns;
      always @(posedge clk) rd_buf_we_r1 <= #TCQ
rd_buf_we_sts[0];

      RAM32M
       #(.INIT_A(64'h0000000000000000),
```

```
        ram_init_done_r_lcl_sts[k] <= #TCQ 1'b1;

      always @(posedge clk) begin
       if (rst) rd_buf_indx_sts_r[k] <= #TCQ 'b0;
       else if (upd_rd_buf_indx_sts[k]) rd_buf_indx_sts_r[k] <= #TCQ
         rd_buf_indx_sts_r[k] + 6'h1 + (DATA_BUF_ADDR_WIDTH == 5 ||
DATA_BUF_ADDR_WIDTH == 6  ? 0 :
                            (single_data_sts[k] &&
~rd_buf_indx_sts_r[k][0]));
      end

    end

// Instantiate status RAM.  One bit for status and one for "end".
// Turns out read to write back status is a timing path.  Update
// the status in the ram on the state following the read.  Bypass
// the write data into the status read path.
// Not guaranteed to write second status bit. If it is written, always
// copy in the first status bit.
    begin : status_ram_0
      reg [DATA_BUF_ADDR_WIDTH-1:0] status_ram_wr_addr_r;
      reg [1:0] status_ram_wr_data_r;
      reg wr_status_r1;
      wire [1:0] wr_status;
      wire [DATA_BUF_ADDR_WIDTH-1:0] status_ram_wr_addr_ns =
ram_init_done_r_lcl_sts[0]
                            ? rd_buf_wr_addr_sts[0]
                            :
rd_buf_indx_sts_r[0][DATA_BUF_ADDR_WIDTH-1:0];
      wire [1:0] status_ram_wr_data_ns = ram_init_done_r_lcl_sts[0] ?
                            {rd_data_end, ~(rd_data_offset
                                  ? wr_status_r1
                                  : wr_status[0])}
                            : 2'b0;
      always @(posedge clk)
       status_ram_wr_addr_r <= #TCQ status_ram_wr_addr_ns;
      always @(posedge clk) wr_status_r1 <= #TCQ wr_status[0];
      always @(posedge clk)
       status_ram_wr_data_r <= #TCQ status_ram_wr_data_ns;
      always @(posedge clk) rd_buf_we_r1 <= #TCQ rd_buf_we_sts[0];

      genvar kk;
      if (DATA_BUF_ADDR_WIDTH > 5) begin
       for (kk=0; kk<2; kk=kk+1) begin : w6rams
        RAM64M
         #(.INIT_A(64'h0000000000000000),
           .INIT_B(64'h0000000000000000),
           .INIT_C(64'h0000000000000000),
```

```
        .INIT_B(64'h0000000000000000),
        .INIT_C(64'h0000000000000000),
        .INIT_D(64'h0000000000000000)
       ) RAM32M0 (
        .DOA(rd_status[0]),
        .DOB(),
        .DOC(wr_status),
        .DOD(),
        .DIA(status_ram_wr_data_r),
        .DIB(2'b0),
        .DIC(status_ram_wr_data_r),
        .DID(status_ram_wr_data_r),
        .ADDRA(rd_buf_indx_sts_r[0][4:0]),
        .ADDRB(5'h0),
        .ADDRC(status_ram_wr_addr_ns),
        .ADDRD(status_ram_wr_addr_r),
        .WE(rd_buf_we_r1),
        .WCLK(clk)
       );

// Copies of the status RAM to meet timing
      genvar l;
      (* keep = "true" *) reg [4:0] status_ram_wr_addr_cpy_r
[0:2];
      (* keep = "true" *) reg [1:0] status_ram_wr_data_cpy_r
[0:2];
      (* keep = "true" *) reg [2:0] wr_status_r;
      wire [1:0] wr_status_cpy [0:2];
      (* keep = "true" *) wire [4:0] status_ram_wr_addr_cpy
[0:2];
      (* keep = "true" *) wire [1:0] status_ram_wr_data_cpy
[0:2];
      (* keep = "true" *) reg [2:0] rd_buf_we_r;

      for (l = 0; l < 3; l = l+1) begin : copies_of_sts_ram

       assign status_ram_wr_addr_cpy[l] =
ram_init_done_r_lcl_sts[l+1] ?
                              rd_buf_wr_addr_sts[l+1] :
                              rd_buf_indx_sts_r[l+1][4:0];

       assign status_ram_wr_data_cpy[l] =
ram_init_done_r_lcl_sts[l+1] ?
                              {rd_data_end, ~(rd_data_offset ?
                                    wr_status_r[l] :
                                    wr_status_cpy[l][0])} :
                              2'b0;
```

```
          .INIT_D(64'h0000000000000000)
         ) RAM64M0 (
          .DOA(rd_status[0][kk]),
          .DOB(),
          .DOC(wr_status[kk]),
          .DOD(),
          .DIA(status_ram_wr_data_r[kk]),
          .DIB(1'b0),
          .DIC(status_ram_wr_data_r[kk]),
          .DID(status_ram_wr_data_r[kk]),
          .ADDRA(rd_buf_indx_sts_r[0][DATA_BUF_ADDR_WIDTH-1:0]),
          .ADDRB(6'h0),
          .ADDRC(status_ram_wr_addr_ns),
          .ADDRD(status_ram_wr_addr_r),
          .WE(rd_buf_we_r1),
          .WCLK(clk)
         );
       end
      end
      else begin
       RAM32M
        #(.INIT_A(64'h0000000000000000),
          .INIT_B(64'h0000000000000000),
          .INIT_C(64'h0000000000000000),
          .INIT_D(64'h0000000000000000)
         ) RAM32M0 (
          .DOA(rd_status[0]),
          .DOB(),
          .DOC(wr_status),
          .DOD(),
          .DIA(status_ram_wr_data_r),
          .DIB(2'b0),
          .DIC(status_ram_wr_data_r),
          .DID(status_ram_wr_data_r),
          .ADDRA(rd_buf_indx_sts_r[0][4:0]),
          .ADDRB(5'h0),
          .ADDRC(status_ram_wr_addr_ns),
          .ADDRD(status_ram_wr_addr_r),
          .WE(rd_buf_we_r1),
          .WCLK(clk)
         );
      end

// Copies of the status RAM to meet timing
      genvar l;

      (* keep = "true" *) reg [DATA_BUF_ADDR_WIDTH-1:0]
status_ram_wr_addr_cpy_r [0:2];
```

```
        always @(posedge clk) wr_status_r[l] <= #TCQ
wr_status_cpy[l][0];
        always @(posedge clk)
         status_ram_wr_addr_cpy_r[l] <= #TCQ
status_ram_wr_addr_cpy[l];
        always @(posedge clk)
         status_ram_wr_data_cpy_r[l] <= #TCQ
status_ram_wr_data_cpy[l];
        always @(posedge clk) rd_buf_we_r[l] <= #TCQ
rd_buf_we_sts[l+1];

        RAM32M
         #(.INIT_A(64'h0000000000000000),
           .INIT_B(64'h0000000000000000),
           .INIT_C(64'h0000000000000000),
           .INIT_D(64'h0000000000000000)
          ) RAM32M1 (
           .DOA(rd_status[l+1]),
           .DOB(rd_status[l+4]),
           .DOC(wr_status_cpy[l]),
           .DOD(),
           .DIA(status_ram_wr_data_cpy_r[l]),
           .DIB(status_ram_wr_data_cpy_r[l]),
           .DIC(status_ram_wr_data_cpy_r[l]),
           .DID(2'b0),
           .ADDRA(rd_buf_indx_sts_r[l+1][4:0]),
           .ADDRB(rd_buf_indx_sts_r[l+1][4:0]),
           .ADDRC(status_ram_wr_addr_cpy[l]),
           .ADDRD(status_ram_wr_addr_cpy_r[l]),
           .WE(rd_buf_we_r[l]),
           .WCLK(clk)
          );

      end
    end // block: status_ram

    wire [RAM_WIDTH-1:0] rd_buf_out_data;
    begin : rd_buf
     wire [RAM_WIDTH-1:0] rd_buf_in_data;
     if (REMAINDER == 0)
      if (ECC == "OFF")
       assign rd_buf_in_data = rd_data;
      else
       assign rd_buf_in_data = {ecc_multiple, rd_data};
     else
      if (ECC == "OFF")
       assign rd_buf_in_data = {{6-REMAINDER{1'b0}},
rd_data};
```

```
      (* keep = "true" *) reg [1:0] status_ram_wr_data_cpy_r [0:2];
//      reg [DATA_BUF_ADDR_WIDTH-1:0] status_ram_wr_data_cpy_r
[0:2];

      (* keep = "true" *) reg [2:0] wr_status_r;
      wire [1:0] wr_status_cpy [0:2];
      (* keep = "true" *) wire [DATA_BUF_ADDR_WIDTH-1:0]
status_ram_wr_addr_cpy [0:2];
      (* keep = "true" *) wire [1:0] status_ram_wr_data_cpy [0:2];
      (* keep = "true" *) reg [2:0] rd_buf_we_r;

      for (l = 0; l < 3; l = l+1) begin : copies_of_sts_ram

       assign status_ram_wr_addr_cpy[l] = ram_init_done_r_lcl_sts[l+1]
?
                                rd_buf_wr_addr_sts[l+1] :

rd_buf_indx_sts_r[l+1][DATA_BUF_ADDR_WIDTH-1:0];

       assign status_ram_wr_data_cpy[l] = ram_init_done_r_lcl_sts[l+1]
?
                                {rd_data_end, ~(rd_data_offset ?
                                        wr_status_r[l] :
                                        wr_status_cpy[l][0])} :
                                2'b0;

       always @(posedge clk) wr_status_r[l] <= #TCQ wr_status_cpy[l][0];
       always @(posedge clk)
        status_ram_wr_addr_cpy_r[l] <= #TCQ
status_ram_wr_addr_cpy[l];
       always @(posedge clk)
        status_ram_wr_data_cpy_r[l] <= #TCQ
status_ram_wr_data_cpy[l];
       always @(posedge clk) rd_buf_we_r[l] <= #TCQ
rd_buf_we_sts[l+1];



       genvar jj;
       if (DATA_BUF_ADDR_WIDTH > 5) begin
        for (jj=0; jj<2; jj=jj+1) begin : w6rams1
         RAM64M
          #(.INIT_A(64'h0000000000000000),
            .INIT_B(64'h0000000000000000),
            .INIT_C(64'h0000000000000000),
            .INIT_D(64'h0000000000000000)
           ) RAM64M0 (
            .DOA(rd_status[l+1][jj]),
```

```
      else
        assign rd_buf_in_data = {{6-REMAINDER{1'b0}},
ecc_multiple, rd_data};

    reg [5:0] rd_buf_indx_cpy_r [0:RAM_CNT-1];
    reg [RAM_CNT-1:0] upd_rd_buf_indx_cpy;
    (* keep = "true" *) reg [RAM_CNT-1:0] init_done_r;
    wire [RAM_CNT-1:0] address_match_buf0;
    wire [RAM_CNT-1:0] address_match_buf1;
    wire [RAM_CNT-1:0] address_match_dout0;
    wire [RAM_CNT-1:0] address_match_dout1;
    wire [RAM_CNT-1:0] bypass_buf;
    wire [RAM_CNT-1:0] app_rd_data_end_buf;
    (* keep = "true" *) reg [RAM_CNT-1:0] single_data_buf;
    wire [RAM_CNT-1:0] rd_data_rdy_buf;
    (* keep = "true" *) wire [RAM_CNT-1:0] rd_buf_we;
    reg [RAM_WIDTH-1:0] app_rd_data_ns;  // spyglass
disable W498

    genvar i;
    for (i=0; i<RAM_CNT; i=i+1) begin : rd_buffer_ram

      // Dedicated copy for driving distributed RAM.
      assign address_match_buf0[i] =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_cpy_r[i][2:0]});
      assign address_match_buf1[i] =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_cpy_r[i][4:3]});
      assign address_match_dout0[i] =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_cpy_r[i][2:0]});
      assign address_match_dout1[i] =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_cpy_r[i][4:3]});
      assign bypass_buf[i] = rd_data_en &&
address_match_buf0[i] && address_match_buf1[i];
      assign app_rd_data_end_buf[i] = bypass_buf[i] ?
rd_data_end : rd_status[i%6+1][1];  // spyglass disable
UndrivenNet-ML
      assign rd_data_rdy_buf[i] = (rd_status[i%6+1][0] ==
rd_buf_indx_cpy_r[i][5]);
      assign rd_buf_we[i] = ~init_done_r[i] | | rd_data_en;
       always @(posedge clk)
         if (rst)
           single_data_buf[i] <= #TCQ 1'b0;
         else if (init_done_r[i])
           single_data_buf[i] <= #TCQ app_rd_data_end_buf[i]
&& ~(DATA_BUF_ADDR_WIDTH == 5) &&
                          ~rd_buf_indx_cpy_r[i][0];

      always @(posedge clk)
            .DOB(rd_status[l+4][jj]),
            .DOC(wr_status_cpy[l][jj]),
            .DOD(),
            .DIA(status_ram_wr_data_cpy_r[l][jj]),
            .DIB(status_ram_wr_data_cpy_r[l][jj]),
            .DIC(status_ram_wr_data_cpy_r[l][jj]),
            .DID(1'b0),
            .ADDRA(rd_buf_indx_sts_r[l+1][DATA_BUF_ADDR_WIDTH-
1:0]),
            .ADDRB(rd_buf_indx_sts_r[l+1][DATA_BUF_ADDR_WIDTH-
1:0]),
            .ADDRC(status_ram_wr_addr_cpy[l]),
            .ADDRD(status_ram_wr_addr_cpy_r[l]),
            .WE(rd_buf_we_r[l]),
            .WCLK(clk)
           );
        end
       end
       else begin
        RAM32M
         #(.INIT_A(64'h0000000000000000),
           .INIT_B(64'h0000000000000000),
           .INIT_C(64'h0000000000000000),
           .INIT_D(64'h0000000000000000)
          ) RAM32M1 (
           .DOA(rd_status[l+1]),
           .DOB(rd_status[l+4]),
           .DOC(wr_status_cpy[l]),
           .DOD(),
           .DIA(status_ram_wr_data_cpy_r[l]),
           .DIB(status_ram_wr_data_cpy_r[l]),
           .DIC(status_ram_wr_data_cpy_r[l]),
           .DID(2'b0),
           .ADDRA(rd_buf_indx_sts_r[l+1][4:0]),
           .ADDRB(rd_buf_indx_sts_r[l+1][4:0]),
           .ADDRC(status_ram_wr_addr_cpy[l]),
           .ADDRD(status_ram_wr_addr_cpy_r[l]),
           .WE(rd_buf_we_r[l]),
           .WCLK(clk)
          );
       end

     end
   end // block: status_ram

   wire [RAM_WIDTH-1:0] rd_buf_out_data;
   begin : rd_buf
    wire [RAM_WIDTH-1:0] rd_buf_in_data;
```

```
        if (rst)
          init_done_r[i] <= #TCQ 1'b0;
        else if (rd_buf_indx_cpy_r[i][4:0] == 5'h1f)
          init_done_r[i] <= #TCQ 1'b1;

      always @(*) begin
        casez
({init_done_r[i],address_match_buf0[i],address_match_buf1[
i],
              rd_data_en, rd_data_rdy_buf[i]})
          5'b0???? : upd_rd_buf_indx_cpy[i] = 1'b1;
          5'b1???1 : upd_rd_buf_indx_cpy[i] = 1'b1;
          5'b11110 : upd_rd_buf_indx_cpy[i] = 1'b1;
          default : upd_rd_buf_indx_cpy[i] = 1'b0;
        endcase
      end

      always @(posedge clk) begin
        if (rst)
          rd_buf_indx_cpy_r[i] <= #TCQ 'b0;
        else if (upd_rd_buf_indx_cpy[i])
          rd_buf_indx_cpy_r[i] <= #TCQ rd_buf_indx_cpy_r[i] +
6'h1;
      end

      RAM32M
        #(.INIT_A(64'h0000000000000000),
          .INIT_B(64'h0000000000000000),
          .INIT_C(64'h0000000000000000),
          .INIT_D(64'h0000000000000000)
      ) RAM32M0 (
          .DOA(rd_buf_out_data[((i*6)+4)+:2]),
          .DOB(rd_buf_out_data[((i*6)+2)+:2]),
          .DOC(rd_buf_out_data[((i*6)+0)+:2]),
          .DOD(),
          .DIA(rd_buf_in_data[((i*6)+4)+:2]),
          .DIB(rd_buf_in_data[((i*6)+2)+:2]),
          .DIC(rd_buf_in_data[((i*6)+0)+:2]),
          .DID(2'b0),
          .ADDRA(rd_buf_indx_cpy_r[i][4:0] +
single_data_buf[i]),
          .ADDRB(rd_buf_indx_cpy_r[i][4:0] +
single_data_buf[i]),
          .ADDRC(rd_buf_indx_cpy_r[i][4:0] +
single_data_buf[i]),
          .ADDRD(rd_buf_wr_addr),
          .WE(rd_buf_we[i]),
          .WCLK(clk)
```

```
      if (REMAINDER == 0)
        if (ECC == "OFF")
          assign rd_buf_in_data = rd_data;
        else
          assign rd_buf_in_data = {ecc_multiple, rd_data};
      else
        if (ECC == "OFF")
          assign rd_buf_in_data = {{6-REMAINDER{1'b0}}, rd_data};
        else
          assign rd_buf_in_data = {{6-REMAINDER{1'b0}}, ecc_multiple,
rd_data};

      reg [DATA_BUF_ADDR_WIDTH:0] rd_buf_indx_cpy_r [0:RAM_CNT-
1];
      reg [RAM_CNT-1:0] upd_rd_buf_indx_cpy;
      (* keep = "true" *) reg [RAM_CNT-1:0] init_done_r;
      wire [RAM_CNT-1:0] address_match_buf0;
      wire [RAM_CNT-1:0] address_match_buf1;
      wire [RAM_CNT-1:0] address_match_dout0;
      wire [RAM_CNT-1:0] address_match_dout1;
      wire [RAM_CNT-1:0] bypass_buf;
      wire [RAM_CNT-1:0] app_rd_data_end_buf;
      (* keep = "true" *) reg [RAM_CNT-1:0] single_data_buf;
      wire [RAM_CNT-1:0] rd_data_rdy_buf;
      (* keep = "true" *) wire [RAM_CNT-1:0] rd_buf_we;
      reg [RAM_WIDTH-1:0] app_rd_data_ns;  // spyglass disable W498



      genvar i;
      for (i=0; i<RAM_CNT; i=i+1) begin : rd_buffer_ram

        // Dedicated copy for driving distributed RAM.
        assign address_match_buf0[i] =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_cpy_r[i][2:0]});
        assign address_match_buf1[i]
         = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_cpy_r[i][5:3]})
                                    :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_cpy_r[i][4:3]});
        assign address_match_dout0[i] =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_cpy_r[i][2:0]});
        assign address_match_dout1[i]
         = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_cpy_r[i][5:3]})
                                    :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_cpy_r[i][4:3]});
```

```
        );

      always @(posedge clk)
       if (rd_data_en & address_match_dout0[i] &
address_match_dout1[i])
         app_rd_data_ns[i*6+:6] <= #TCQ
rd_buf_in_data[i*6+:6];
       else
         app_rd_data_ns[i*6+:6] <= #TCQ
rd_buf_out_data[i*6+:6]; // spyglass disable UndrivenNet-ML

    end // block: rd_buffer_ram

    assign app_rd_data =
app_rd_data_ns[APP_DATA_WIDTH-1:0];

  end

  wire address_match_cpy2_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[9][2:0]});
  wire address_match_cpy2_1 =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[9][4:3]});
  assign bypass_cpy = rd_data_en &&
address_match_cpy2_0 && address_match_cpy2_1;
  assign rd_data_rdy_cpy = (rd_status[0][0] ==
rd_buf_indx_r[9][5]);

  wire address_match_cpy_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[1][2:0]});
  wire address_match_cpy_1 =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[1][4:3]});
  wire address_match_cpy6_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[14][2:0]});
  wire address_match_cpy6_1 =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[14][4:3]});
  wire address_match_cpy11_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[5][2:0]});
  wire address_match_cpy11_1 =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[5][4:3]});

  wire bypass = rd_data_en && address_match_cpy_0 &&
address_match_cpy_1;

  wire rd_data_rdy = (rd_status[0][0] ==
rd_buf_indx_r[5][5]);
  wire bypass_cpy2 = rd_data_en &&
address_match_cpy11_0 && address_match_cpy11_1;
```

```
    assign bypass_buf[i] = rd_data_en && address_match_buf0[i] &&
address_match_buf1[i];
    assign app_rd_data_end_buf[i] = bypass_buf[i] ? rd_data_end :
rd_status[i%6+1][1];  // spyglass disable UndrivenNet-ML
    assign rd_data_rdy_buf[i] = (rd_status[i%6+1][0] ==
rd_buf_indx_cpy_r[i][DATA_BUF_ADDR_WIDTH]);
    assign rd_buf_we[i] = ~init_done_r[i] || rd_data_en;
     always @(posedge clk)
      if (rst)
        single_data_buf[i] <= #TCQ 1'b0;
      else if (init_done_r[i])
      single_data_buf[i] <= #TCQ app_rd_data_end_buf[i] &&
~(DATA_BUF_ADDR_WIDTH == 6 || DATA_BUF_ADDR_WIDTH == 5)  &&
                            ~rd_buf_indx_cpy_r[i][0];



    always @(posedge clk)
     if (rst)
       init_done_r[i] <= #TCQ 1'b0;
     else if (rd_buf_indx_cpy_r[i][DATA_BUF_ADDR_WIDTH-1:0] ==
{DATA_BUF_ADDR_WIDTH{1'b1}})
       init_done_r[i] <= #TCQ 1'b1;

    always @(*) begin
     casez
({init_done_r[i],address_match_buf0[i],address_match_buf1[i],
          rd_data_en, rd_data_rdy_buf[i]})
       5'b0???? : upd_rd_buf_indx_cpy[i] = 1'b1;
       5'b1???1 : upd_rd_buf_indx_cpy[i] = 1'b1;
       5'b11110 : upd_rd_buf_indx_cpy[i] = 1'b1;
       default : upd_rd_buf_indx_cpy[i] = 1'b0;
     endcase
    end

    always @(posedge clk) begin
     if (rst)
       rd_buf_indx_cpy_r[i] <= #TCQ 'b0;
     else if (upd_rd_buf_indx_cpy[i])
       rd_buf_indx_cpy_r[i] <= #TCQ rd_buf_indx_cpy_r[i] + 6'h1;
    end


    genvar ii;
    if (DATA_BUF_ADDR_WIDTH > 5) begin
     for (ii=0; ii<2; ii=ii+1) begin : w6rams
       RAM64M
        #(.INIT_A(64'h0000000000000000),
```

```
    always @(posedge clk) begin
     if (rst)
       app_rd_data_valid <= #TCQ 1'b0;
     else if (ram_init_done_r_lcl[5])
       app_rd_data_valid <= #TCQ (bypass_cpy2 ||
rd_data_rdy);
    end

    always @(posedge clk) begin
     if (rst)
       app_rd_data_end <= #TCQ 1'b0;
     else begin
       if (rd_data_en & address_match_cpy6_0 &
address_match_cpy6_1)
         app_rd_data_end <= #TCQ rd_data_end;
       else
         app_rd_data_end <= #TCQ rd_status[0][1];
     end
    end

    wire address_match_cpy13_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[7][2:0]});
    wire address_match_cpy13_1 =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[7][4:3]});
    wire app_rd_data_end_cpy0 = bypass ? rd_data_end :
rd_status[0][1];
    assign single_data = ram_init_done_r_lcl[6] &&
app_rd_data_end_cpy0;

    if (ECC != "OFF") begin : assign_app_ecc_multiple
     wire [2*nCK_PER_CLK-1:0] app_ecc_multiple_err_ns =
                          bypass
                          ? ecc_multiple
                          :
rd_buf_out_data[APP_DATA_WIDTH+:8];
      always @(posedge clk) app_ecc_multiple_err_r <=
                       #TCQ app_ecc_multiple_err_ns;
    end

    //Added to fix timing. The signal app_rd_data_valid has
    //a very high fanout. So making a dedicated copy for
usage
    //with the occ_cnt counter.
    (* keep = "true" *) reg app_rd_data_valid_cpy_r;
    wire address_match_cpy12_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[19][2:0]});
    wire address_match_cpy12_1 =
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[19][4:3]});
```

```
          .INIT_B(64'h0000000000000000),
          .INIT_C(64'h0000000000000000),
          .INIT_D(64'h0000000000000000)
         ) RAM64M0 (
          .DOA(rd_buf_out_data[((i*6)+4)+ii]),
          .DOB(rd_buf_out_data[((i*6)+2)+ii]),
          .DOC(rd_buf_out_data[((i*6)+0)+ii]),
          .DOD(),
          .DIA(rd_buf_in_data[((i*6)+4)+ii]),
          .DIB(rd_buf_in_data[((i*6)+2)+ii]),
          .DIC(rd_buf_in_data[((i*6)+0)+ii]),
          .DID(2'b0),
          .ADDRA(rd_buf_indx_cpy_r[i][5:0] + single_data_buf[i]),
          .ADDRB(rd_buf_indx_cpy_r[i][5:0] + single_data_buf[i]),
          .ADDRC(rd_buf_indx_cpy_r[i][5:0] + single_data_buf[i]),
          .ADDRD(rd_buf_wr_addr),
          .WE(rd_buf_we[i]),
          .WCLK(clk)
         );
       end
     end
     else begin
       RAM32M
        #(.INIT_A(64'h0000000000000000),
          .INIT_B(64'h0000000000000000),
          .INIT_C(64'h0000000000000000),
          .INIT_D(64'h0000000000000000)
        ) RAM32M0 (
          .DOA(rd_buf_out_data[((i*6)+4)+:2]),
          .DOB(rd_buf_out_data[((i*6)+2)+:2]),
          .DOC(rd_buf_out_data[((i*6)+0)+:2]),
          .DOD(),
          .DIA(rd_buf_in_data[((i*6)+4)+:2]),
          .DIB(rd_buf_in_data[((i*6)+2)+:2]),
          .DIC(rd_buf_in_data[((i*6)+0)+:2]),
          .DID(2'b0),
          .ADDRA(rd_buf_indx_cpy_r[i][4:0] + single_data_buf[i]),
          .ADDRB(rd_buf_indx_cpy_r[i][4:0] + single_data_buf[i]),
          .ADDRC(rd_buf_indx_cpy_r[i][4:0] + single_data_buf[i]),
          .ADDRD(rd_buf_wr_addr),
          .WE(rd_buf_we[i]),
          .WCLK(clk)
         );
       end



      always @(posedge clk)
```

```
    wire bypass_cpy1 = rd_data_en &&
address_match_cpy12_0 && address_match_cpy12_1;
    wire rd_data_rdy_cpy2 = (rd_status[0][0] ==
rd_buf_indx_r[19][5]);

    always @(posedge clk) begin
     if (rst)
       app_rd_data_valid_cpy_r <= #TCQ 1'b0;
     else if (ram_init_done_r_lcl[7])
       app_rd_data_valid_cpy_r <= #TCQ (bypass_cpy1 ||
rd_data_rdy_cpy2);
    end

    // Keep track of how many entries in the queue hold data.
    // changed to use registered version of the signals in
    // ordered to fix timing
    wire free_rd_buf = app_rd_data_valid_cpy_r &&
app_rd_data_end;

    reg [DATA_BUF_ADDR_WIDTH:0] occ_cnt_r;
    wire [DATA_BUF_ADDR_WIDTH:0] occ_minus_one =
occ_cnt_r - 1;
    wire [DATA_BUF_ADDR_WIDTH:0] occ_plus_one =
occ_cnt_r + 1;
    begin : occupied_counter
     always @(posedge clk) begin
      if (rst) occ_cnt_r <= #TCQ 'b0;
      else case ({rd_accepted, free_rd_buf})
           2'b01 : occ_cnt_r <= #TCQ occ_minus_one;
           2'b10 : occ_cnt_r <= #TCQ occ_plus_one;
      endcase // case ({wr_data_end, new_rd_data})
     end
     //assign rd_buf_full =
occ_cnt_r[DATA_BUF_ADDR_WIDTH];
     assign rd_buf_full =
(((occ_cnt_r[DATA_BUF_ADDR_WIDTH-1:0] ==
{DATA_BUF_ADDR_WIDTH{1'b1}}) && rd_accepted) ||
                          occ_cnt_r[DATA_BUF_ADDR_WIDTH])
                         ? 1 : 0;


`ifdef MC_SVA
 rd_data_buffer_full: cover property (@(posedge clk) (~rst
&& rd_buf_full));
 rd_data_buffer_inc_dec_15: cover property (@(posedge clk)
     (~rst && rd_accepted && free_rd_buf && (occ_cnt_r ==
'hf)));
 rd_data_underflow: assert property (@(posedge clk)
```

```
       if (rd_data_en & address_match_dout0[i] &
address_match_dout1[i])
         app_rd_data_ns[i*6+:6] <= #TCQ rd_buf_in_data[i*6+:6];
       else
         app_rd_data_ns[i*6+:6] <= #TCQ rd_buf_out_data[i*6+:6]; //
spyglass disable UndrivenNet-ML

     end // block: rd_buffer_ram

     assign app_rd_data = app_rd_data_ns[APP_DATA_WIDTH-1:0];

    end

    wire address_match_cpy2_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[9][2:0]});
    wire address_match_cpy2_1
       = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_r[9][5:3]})
                          :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[9][4:3]});
    assign bypass_cpy = rd_data_en && address_match_cpy2_0 &&
address_match_cpy2_1;
    assign rd_data_rdy_cpy = (rd_status[0][0] ==
rd_buf_indx_r[9][DATA_BUF_ADDR_WIDTH]);

    wire address_match_cpy_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[1][2:0]});
    wire address_match_cpy_1
       = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_r[1][5:3]})
                          :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[1][4:3]});
    wire address_match_cpy6_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[14][2:0]});
    wire address_match_cpy6_1
       = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_r[14][5:3]})
                          :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[14][4:3]});
    wire address_match_cpy11_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[5][2:0]});
    wire address_match_cpy11_1
       = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_r[5][5:3]})
                          :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[5][4:3]});
```

```
      (rst || !((occ_cnt_r == 'b0) && (occ_cnt_r == 'h1f))));
 rd_data_overflow: assert property (@(posedge clk)
      (rst || !((occ_cnt_r == 'h10) && (occ_cnt_r == 'h11))));
`endif
    end // block: occupied_counter


// Generate the data_buf_address written into the memory
controller
// for reads.  Increment with each accepted read, and rollover
at 0xf.
    reg [DATA_BUF_ADDR_WIDTH-1:0]
rd_data_buf_addr_r_lcl;
    assign rd_data_buf_addr_r = rd_data_buf_addr_r_lcl;
    begin : data_buf_addr
      always @(posedge clk) begin
       if (rst) rd_data_buf_addr_r_lcl <= #TCQ 'b0;
       else if (rd_accepted) rd_data_buf_addr_r_lcl <= #TCQ
                        rd_data_buf_addr_r_lcl + 1;
      end
    end // block: data_buf_addr
  end // block: not_strict_mode
 endgenerate

endmodule // ddr4_v2_2_6_ui_rd_data
```

```
    wire bypass = rd_data_en && address_match_cpy_0 &&
address_match_cpy_1;

    wire rd_data_rdy = (rd_status[0][0] ==
rd_buf_indx_r[5][DATA_BUF_ADDR_WIDTH]);
    wire bypass_cpy2 = rd_data_en && address_match_cpy11_0 &&
address_match_cpy11_1;

    always @(posedge clk) begin
     if (rst)
      app_rd_data_valid <= #TCQ 1'b0;
     else if (ram_init_done_r_lcl[DATA_BUF_ADDR_WIDTH])
      app_rd_data_valid <= #TCQ (bypass_cpy2 || rd_data_rdy);
    end

    always @(posedge clk) begin
     if (rst)
      app_rd_data_end <= #TCQ 1'b0;
     else begin
      if (rd_data_en & address_match_cpy6_0 &
address_match_cpy6_1)
       app_rd_data_end <= #TCQ rd_data_end;
      else
       app_rd_data_end <= #TCQ rd_status[0][1];
     end
    end

    wire address_match_cpy13_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[7][2:0]});
    wire address_match_cpy13_1
       = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_r[7][5:3]})
                          :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[7][4:3]});
    wire app_rd_data_end_cpy0 = bypass ? rd_data_end :
rd_status[0][1];
    assign single_data = ram_init_done_r_lcl[6] &&
app_rd_data_end_cpy0;

    if (ECC != "OFF") begin : assign_app_ecc_multiple
     wire [2*nCK_PER_CLK-1:0] app_ecc_multiple_err_ns =
                        bypass
                        ? ecc_multiple
                        : rd_buf_out_data[APP_DATA_WIDTH+:8];
     always @(posedge clk) app_ecc_multiple_err_r <=
                       #TCQ app_ecc_multiple_err_ns;
    end
```

```
    //Added to fix timing. The signal app_rd_data_valid has
    //a very high fanout. So making a dedicated copy for usage
    //with the occ_cnt counter.
    (* keep = "true" *) reg app_rd_data_valid_cpy_r;
    wire address_match_cpy12_0 =
match6_1({rd_buf_wr_addr[2:0],rd_buf_indx_r[19][2:0]});
    wire address_match_cpy12_1
        = (DATA_BUF_ADDR_WIDTH > 5) ?
match6_1({rd_buf_wr_addr[5:3],rd_buf_indx_r[19][5:3]})
                                :
match4_1({rd_buf_wr_addr[4:3],rd_buf_indx_r[19][4:3]});
    wire bypass_cpy1 = rd_data_en && address_match_cpy12_0 &&
address_match_cpy12_1;
    wire rd_data_rdy_cpy2 = (rd_status[0][0] ==
rd_buf_indx_r[19][DATA_BUF_ADDR_WIDTH]);

    always @(posedge clk) begin
     if (rst)
      app_rd_data_valid_cpy_r <= #TCQ 1'b0;
     else if (ram_init_done_r_lcl[7])
      app_rd_data_valid_cpy_r <= #TCQ (bypass_cpy1 ||
rd_data_rdy_cpy2);
    end

    // Keep track of how many entries in the queue hold data.
    // changed to use registered version of the signals in
    // ordered to fix timing
    wire free_rd_buf = app_rd_data_valid_cpy_r && app_rd_data_end;

    reg [DATA_BUF_ADDR_WIDTH:0] occ_cnt_r;
    wire [DATA_BUF_ADDR_WIDTH:0] occ_minus_one = occ_cnt_r - 1;
    wire [DATA_BUF_ADDR_WIDTH:0] occ_plus_one = occ_cnt_r + 1;
    // synthesis translate_off
    int count_rd_accepteds, count_free_rd_bufs;
    // synthesis translate_on
    begin : occupied_counter
     always @(posedge clk) begin
      if (rst) occ_cnt_r <= #TCQ 'b0;
      else case ({rd_accepted, free_rd_buf})
           2'b01 : occ_cnt_r <= #TCQ occ_minus_one;
           2'b10 : occ_cnt_r <= #TCQ occ_plus_one;
      endcase // case ({wr_data_end, new_rd_data})
     end
     //assign rd_buf_full = occ_cnt_r[DATA_BUF_ADDR_WIDTH];
     assign rd_buf_full = (((occ_cnt_r[DATA_BUF_ADDR_WIDTH-1:0] ==
{DATA_BUF_ADDR_WIDTH{1'b1}}) && rd_accepted) ||
                          occ_cnt_r[DATA_BUF_ADDR_WIDTH])
                         ? 1 : 0;
```

	`// synthesis translate_off`
	`always @(posedge clk) begin`
	`if (rst) begin`
	`count_rd_accepteds <= #TCQ 'b0;`
	`count_free_rd_bufs <= #TCQ 'b0;`
	`end`
	`else begin`
	`if (rd_accepted) count_rd_accepteds <= #TCQ count_rd_accepteds + 1'b1;`
	`if (free_rd_buf) count_free_rd_bufs <= #TCQ count_free_rd_bufs + 1'b1;`
	`end`
	`end`
	`// synthesis translate_on`
	
	`` `ifdef MC_SVA ``
	`rd_data_buffer_full: cover property (@(posedge clk) (~rst && rd_buf_full));`
	`rd_data_buffer_inc_dec_15: cover property (@(posedge clk)`
	`(~rst && rd_accepted && free_rd_buf && (occ_cnt_r == 'hf)));`
	`rd_data_underflow: assert property (@(posedge clk)`
	`(rst || !((occ_cnt_r == 'b0) && (occ_cnt_r == 'h1f))));`
	`rd_data_overflow: assert property (@(posedge clk)`
	`(rst || !((occ_cnt_r == 'h10) && (occ_cnt_r == 'h11))));`
	`` `endif ``
	`end // block: occupied_counter`
	
	`// Generate the data_buf_address written into the memory controller`
	`// for reads. Increment with each accepted read, and rollover at 0xf.`
	`reg [DATA_BUF_ADDR_WIDTH-1:0] rd_data_buf_addr_r_lcl;`
	`assign rd_data_buf_addr_r = rd_data_buf_addr_r_lcl;`
	`begin : data_buf_addr`
	`always @(posedge clk) begin`
	`if (rst) rd_data_buf_addr_r_lcl <= #TCQ 'b0;`
	`else if (rd_accepted) rd_data_buf_addr_r_lcl <= #TCQ`
	`rd_data_buf_addr_r_lcl + 1;`
	`end`
	`end // block: data_buf_addr`
	`end // block: not_strict_mode`
	`endgenerate`
	
	`endmodule`

DDR4_v2_2_ui_wr_data.sv

Standard Xilinx MIG	Modified MRAM MIG

(null)	parameter FIFO_ADDR_WIDTH = 5,
app_wdf_rdy, wr_req_16, wr_data_buf_addr, wr_data, wr_data_mask,	app_wdf_rdy, wr_req_full, wr_data_buf_addr, wr_data, wr_data_mask,
raw_not_ecc,	raw_not_ecc, wrdata_fifo_empty,
(null)	localparam WR_BUF_BITS_PER_RAM = (FIFO_ADDR_WIDTH == 4)? 6 : (FIFO_ADDR_WIDTH == 5)? 3 : 1;
localparam FULL_RAM_CNT = (WR_BUF_WIDTH/6);	localparam FULL_RAM_CNT = (WR_BUF_WIDTH/WR_BUF_BITS_PER_RAM);
localparam REMAINDER = WR_BUF_WIDTH % 6;	localparam REMAINDER = WR_BUF_WIDTH % WR_BUF_BITS_PER_RAM;
localparam RAM_WIDTH = (RAM_CNT*6);	localparam RAM_WIDTH = (RAM_CNT*WR_BUF_BITS_PER_RAM);
input [3:0] wr_data_addr;	input [FIFO_ADDR_WIDTH-1:0] wr_data_addr;
reg [3:0] wr_data_addr_r;	reg [FIFO_ADDR_WIDTH-1:0] wr_data_addr_r;
reg [3:0] rd_data_indx_r;	reg [FIFO_ADDR_WIDTH-1:0] rd_data_indx_r;
rd_data_indx_r <= #TCQ rd_data_indx_r + 5'h1;	rd_data_indx_r <= #TCQ rd_data_indx_r + 1'b1;
reg [3:0] data_buf_addr_cnt_r;	reg [FIFO_ADDR_WIDTH-1:0] data_buf_addr_cnt_r;
reg [3:0] data_buf_addr_cnt_ns;	reg [FIFO_ADDR_WIDTH-1:0] data_buf_addr_cnt_ns;
if (rst) data_buf_addr_cnt_ns = 4'b0;	if (rst) data_buf_addr_cnt_ns = 'b0;
data_buf_addr_cnt_r + 4'h1;	data_buf_addr_cnt_r + 1'b1;
wire [3:0] wr_data_pntr;	wire [FIFO_ADDR_WIDTH-1:0] wr_data_pntr;
wire [4:0] wb_wr_data_addr;	wire [FIFO_ADDR_WIDTH:0] wb_wr_data_addr;
wire [4:0] wb_wr_data_addr_w;	wire [FIFO_ADDR_WIDTH:0] wb_wr_data_addr_w;
reg [3:0] wr_data_indx_r;	reg [FIFO_ADDR_WIDTH-1:0] wr_data_indx_r;
wr_data_indx_r <= #TCQ 4'h1;	wr_data_indx_r <= #TCQ 9'b1;
wr_data_indx_r <= #TCQ wr_data_indx_r + 4'h1;	wr_data_indx_r <= #TCQ wr_data_indx_r + 1'b1;
reg [4:1] wb_wr_data_addr_ns;	reg [FIFO_ADDR_WIDTH:1] wb_wr_data_addr_ns;
reg [4:1] wb_wr_data_addr_r;	reg [FIFO_ADDR_WIDTH:1] wb_wr_data_addr_r;
if (rst) wb_wr_data_addr_ns = 4'h0;	if (rst) wb_wr_data_addr_ns = 'b0;
input [3:0] ram_init_done_r;	input [FIFO_ADDR_WIDTH-1:0] ram_init_done_r;
(null)	output wire wrdata_fifo_empty; localparam ENTRIES_WIDTH = 32'b1 << FIFO_ADDR_WIDTH;
reg [15:0] occ_cnt;	reg [ENTRIES_WIDTH-1:0] occ_cnt; reg wrdata_fifo_empty_r;
occ_cnt <= #TCQ 16'h0000;	occ_cnt <= #TCQ 'b0;
2'b01 : occ_cnt <= #TCQ {1'b0,occ_cnt[15:1]}; 2'b10 : occ_cnt <= #TCQ {occ_cnt[14:0],1'b1};	2'b01 : occ_cnt <= #TCQ {1'b0,occ_cnt[ENTRIES_WIDTH-1:1]}; 2'b10 : occ_cnt <= #TCQ {occ_cnt[ENTRIES_WIDTH-2:0],1'b1};
(occ_cnt[14] && wr_data_end && ~rd_data_upd_indx_cpy_r) || (occ_cnt[15] && ~rd_data_upd_indx_cpy_r));	(occ_cnt[ENTRIES_WIDTH-2] && wr_data_end && ~rd_data_upd_indx_cpy_r) || (occ_cnt[ENTRIES_WIDTH-1] && ~rd_data_upd_indx_cpy_r));
(occ_cnt[14] && wr_data_end && ~rd_data_upd_indx_r) || (occ_cnt[15] && ~rd_data_upd_indx_r));	(occ_cnt[ENTRIES_WIDTH-2] && wr_data_end && ~rd_data_upd_indx_r) || (occ_cnt[ENTRIES_WIDTH-1] && ~rd_data_upd_indx_r));
always @(posedge clk) if (rst)	always @(posedge clk) if (rst) begin

app_wdf_rdy_r <= #TCQ 1'b0;
else
app_wdf_rdy_r <= #TCQ wdf_rdy_ns_cpy;
 	
assign app_wdf_rdy = app_wdf_rdy_r;
	app_wdf_rdy_r <= #TCQ 1'b0;
	wrdata_fifo_empty_r <= #TCQ 1'b1;
	end
	else begin
	app_wdf_rdy_r <= #TCQ wdf_rdy_ns_cpy;
	wrdata_fifo_empty_r <= #TCQ !(rd_data_upd_indx_r || |occ_cnt);
	end
	
	assign app_wdf_rdy = app_wdf_rdy_r;
	assign wrdata_fifo_empty = wrdata_fifo_empty_r;

output wire wr_req_16;	output wire wr_req_full;
reg [4:0] wr_req_cnt_ns; reg [4:0] wr_req_cnt_r;	reg [FIFO_ADDR_WIDTH:0] wr_req_cnt_ns; reg [FIFO_ADDR_WIDTH:0] wr_req_cnt_r;
if (rst) wr_req_cnt_ns = 5'b0;	if (rst) wr_req_cnt_ns = 'b0;
2'b01 : wr_req_cnt_ns = wr_req_cnt_r - 5'b1; 2'b10 : wr_req_cnt_ns = wr_req_cnt_r + 5'b1;	2'b01 : wr_req_cnt_ns = wr_req_cnt_r - 1'b1; 2'b10 : wr_req_cnt_ns = wr_req_cnt_r + 1'b1;
assign wr_req_16 = (wr_req_cnt_ns == 5'h10);	assign wr_req_full = (wr_req_cnt_ns == ENTRIES_WIDTH);
wr_req_mc_full: cover property (@(posedge clk) (~rst && wr_req_16));	wr_req_mc_full: cover property (@(posedge clk) (~rst && wr_req_full));
(~rst && wr_accepted && rd_data_upd_indx_r && (wr_req_cnt_r == 5'hf)));	(~rst && wr_accepted && rd_data_upd_indx_r && (wr_req_cnt_r == ENTRIES_WIDTH-1)));
(rst || !((wr_req_cnt_r == 5'b0) && (wr_req_cnt_ns == 5'h1f))));	(rst || !((wr_req_cnt_r == 'b0) && (wr_req_cnt_ns == ENTRIES_WIDTH*2-1))));
(rst || !((wr_req_cnt_r == 5'h10) && (wr_req_cnt_ns == 5'h11))));	(rst || !((wr_req_cnt_r == ENTRIES_WIDTH) && (wr_req_cnt_ns == ENTRIES_WIDTH+1))));
input [3:0] ram_init_addr; output wire [3:0] wr_data_buf_addr;	input [FIFO_ADDR_WIDTH-1:0] ram_init_addr; output wire [FIFO_ADDR_WIDTH-1:0] wr_data_buf_addr;
localparam PNTR_RAM_CNT = 2;	localparam PNTR_RAM_CNT = (FIFO_ADDR_WIDTH == 4)? 2 : (FIFO_ADDR_WIDTH == 5)? 3 : 6;
wire [3:0] pointer_wr_data = ram_init_done_r[2]	wire [PNTR_RAM_CNT*2-1:0] pointer_wr_data = ram_init_done_r[2]
wire [3:0] pointer_wr_addr = ram_init_done_r[2]	wire [FIFO_ADDR_WIDTH-1:0] pointer_wr_addr = ram_init_done_r[2]

genvar i;
 	
for (i=0; i<PNTR_RAM_CNT; i=i+1) begin : rams
RAM32M
#(.INIT_A(64'h0000000000000000),
.INIT_B(64'h0000000000000000),
.INIT_C(64'h0000000000000000),
.INIT_D(64'h0000000000000000)
) RAM32M0 (
.DOA(),
.DOB(wr_data_buf_addr[i*2+:2]),
.DOC(wr_data_pntr[i*2+:2]),
.DOD(),
.DIA(2'b0),
.DIB(pointer_wr_data[i*2+:2]),
	genvar i;
	
	for (i=0; i<PNTR_RAM_CNT; i=i+1) begin : rams
	if (FIFO_ADDR_WIDTH > 5) begin
	RAM64M
	#(.INIT_A(64'h0000000000000000),
	.INIT_B(64'h0000000000000000),
	.INIT_C(64'h0000000000000000),
	.INIT_D(64'h0000000000000000)
	) RAM64M0 (
	.DOA(),
	.DOB(wr_data_buf_addr_tmp[i]),
	.DOC(wr_data_pntr_tmp[i]),
	.DOD(),
	.DIA(1'b0),

```
        .DIC(pointer_wr_data[i*2+:2]),
        .DID(2'b0),
        .ADDRA(5'b0),
        .ADDRB({1'b0, data_buf_addr_cnt_r}),
        .ADDRC({1'b0, wr_data_indx_r}),
        .ADDRD({1'b0, pointer_wr_addr}),
        .WE(pointer_we),
        .WCLK(clk)
       );
   end // block : rams
endgenerate
```

```
        .DIB(pointer_wr_data[i]),
        .DIC(pointer_wr_data[i]),
        .DID(1'b0),
        .ADDRA(6'b0),
        .ADDRB(data_buf_addr_cnt_r),
        .ADDRC(wr_data_indx_r),
        .ADDRD(pointer_wr_addr),
        .WE(pointer_we),
        .WCLK(clk)
       );
end
else if (FIFO_ADDR_WIDTH > 4) begin
  RAM32M
    #(.INIT_A(64'h0000000000000000),
      .INIT_B(64'h0000000000000000),
      .INIT_C(64'h0000000000000000),
      .INIT_D(64'h0000000000000000)
    ) RAM32M0 (
      .DOA(),
      .DOB(wr_data_buf_addr_tmp[i*2+:2]),
      .DOC(wr_data_pntr_tmp[i*2+:2]),
      .DOD(),
      .DIA(2'b0),
      .DIB(pointer_wr_data[i*2+:2]),
      .DIC(pointer_wr_data[i*2+:2]),
      .DID(2'b0),
      .ADDRA(5'b0),
      .ADDRB(data_buf_addr_cnt_r),
      .ADDRC(wr_data_indx_r),
      .ADDRD(pointer_wr_addr),
      .WE(pointer_we),
      .WCLK(clk)
     );
end
else begin
  RAM32M
    #(.INIT_A(64'h0000000000000000),
      .INIT_B(64'h0000000000000000),
      .INIT_C(64'h0000000000000000),
      .INIT_D(64'h0000000000000000)
    ) RAM32M0 (
      .DOA(),
      .DOB(wr_data_buf_addr_tmp[i*2+:2]),
      .DOC(wr_data_pntr_tmp[i*2+:2]),
      .DOD(),
      .DIA(2'b0),
      .DIB(pointer_wr_data[i*2+:2]),
      .DIC(pointer_wr_data[i*2+:2]),
```

	.DID(2'b0),
	.ADDRA(5'b0),
	.ADDRB({1'b0, data_buf_addr_cnt_r}),
	.ADDRC({1'b0, wr_data_indx_r}),
	.ADDRD({1'b0, pointer_wr_addr}),
	.WE(pointer_we),
	.WCLK(clk)
	);
	end
	end
	assign wr_data_buf_addr = wr_data_buf_addr_tmp;
	assign wr_data_pntr = wr_data_pntr_tmp;
	endgenerate

wire [4:0] rd_addr_w;	wire [FIFO_ADDR_WIDTH:0] rd_addr_w;

genvar ii;
for (ii=0; ii<RAM_CNT; ii=ii+1) begin : wr_buffer_ram
RAM32M
#(.INIT_A(64'h0000000000000000),
.INIT_B(64'h0000000000000000),
.INIT_C(64'h0000000000000000),
.INIT_D(64'h0000000000000000)
) RAM32M0 (
.DOA(wr_buf_out_data_w[((ii*6)+4)+:2]),
.DOB(wr_buf_out_data_w[((ii*6)+2)+:2]),
.DOC(wr_buf_out_data_w[((ii*6)+0)+:2]),
.DOD(),
.DIA(wr_buf_in_data[((ii*6)+4)+:2]),
.DIB(wr_buf_in_data[((ii*6)+2)+:2]),
.DIC(wr_buf_in_data[((ii*6)+0)+:2]),
.DID(2'b0),
.ADDRA(rd_addr_w),
.ADDRB(rd_addr_w),
.ADDRC(rd_addr_w),
.ADDRD(wb_wr_data_addr_w),
.WE(wdf_rdy_ns),
.WCLK(clk)
);
end // block: wr_buffer_ram
endgenerate
 	
output [APP_DATA_WIDTH-1:0] wr_data;
	genvar ii;
	for (ii=0; ii<RAM_CNT; ii=ii+1) begin : wr_buffer_ram
	if (FIFO_ADDR_WIDTH > 5) begin
	RAM128X1D
	#(.INIT(128'h00000000000000000000000000000000)
	) RAM128X1D0 (
	.DPO(wr_buf_out_data_w[ii]), // Read port 1-bit output
	.SPO(), // Read/write port 1-bit output
	.A(wb_wr_data_addr_w), // Read/write port 7-bit address input
	.D(wr_buf_in_data[ii]), // RAM data input
	.DPRA(rd_addr_w), // Read port 7-bit address input
	.WE(wdf_rdy_ns),
	.WCLK(clk)
	);
	end
	else if (FIFO_ADDR_WIDTH > 4) begin
	RAM64M
	#(.INIT_A(64'h0000000000000000),
	.INIT_B(64'h0000000000000000),
	.INIT_C(64'h0000000000000000),
	.INIT_D(64'h0000000000000000)
	) RAM64M0 (
	.DOA(wr_buf_out_data_w[(ii*3)+2]),
	.DOB(wr_buf_out_data_w[(ii*3)+1]),
	.DOC(wr_buf_out_data_w[(ii*3)+0]),
	.DOD(),
	.DIA(wr_buf_in_data[(ii*3)+2]),
	.DIB(wr_buf_in_data[(ii*3)+1]),
	.DIC(wr_buf_in_data[(ii*3)+0]),
	.DID(1'b0),
	.ADDRA(rd_addr_w),
	.ADDRB(rd_addr_w),
	.ADDRC(rd_addr_w),

	.ADDRD(wb_wr_data_addr_w), .WE(wdf_rdy_ns), .WCLK(clk)); end else begin RAM32M #(.INIT_A(64'h0000000000000000), .INIT_B(64'h0000000000000000), .INIT_C(64'h0000000000000000), .INIT_D(64'h0000000000000000)) RAM32M0 (.DOA(wr_buf_out_data_w[((ii*6)+4)+:2]), .DOB(wr_buf_out_data_w[((ii*6)+2)+:2]), .DOC(wr_buf_out_data_w[((ii*6)+0)+:2]), .DOD(), .DIA(wr_buf_in_data[((ii*6)+4)+:2]), .DIB(wr_buf_in_data[((ii*6)+2)+:2]), .DIC(wr_buf_in_data[((ii*6)+0)+:2]), .DID(2'b0), .ADDRA(rd_addr_w), .ADDRB(rd_addr_w), .ADDRC(rd_addr_w), .ADDRD(wb_wr_data_addr_w), .WE(wdf_rdy_ns), .WCLK(clk)); end end // block: wr_buffer_ram endgenerate output [APP_DATA_WIDTH-1:0] wr_data;

DDR4_v2_2_mc_periodic.sv

Standard Xilinx MIG	**Modified MRAM MIG**
(null)	output per_state_idle,
(null)	assign per_state_idle = periodic_state == IDLE;

(null)	if (!periodic_fsm_start) begin
	// Abort waiting if the enable goes away.
	// This cuts the majority of the waiting time.
	// In subsequent states don't try to abort because
	// we might leave ourselves in a bad state.
	periodic_state_nxt = IDLE;

Everspin Technologies, Inc.

Contact Information:

How to Reach Us:

www.everspin.com

E-Mail:

support@everspin.com

orders@everspin.com

sales@everspin.com

USA/Canada/South and Central America

Everspin Technologies

5670 W. Chandler Blvd, Suite 100

Chandler, AZ 85224

+1-877-347-MRAM (6726)

+1-480-347-1111

Europe, Middle East and Africa

support.europe@everspin.com

Japan

support.japan@everspin.com

Asia Pacific