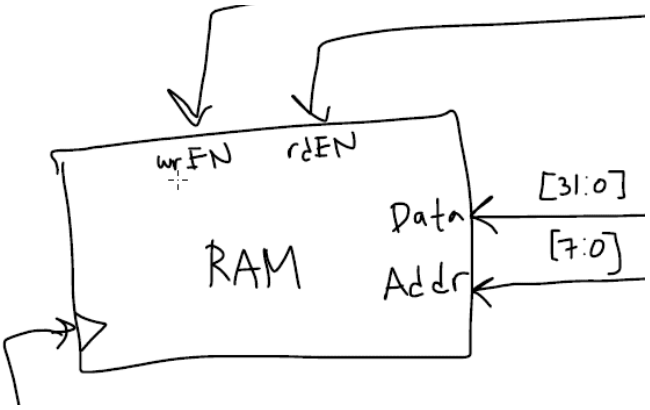
Kevin Hoser and Alex Schendel
EE 435
Project 4
9 April 2020

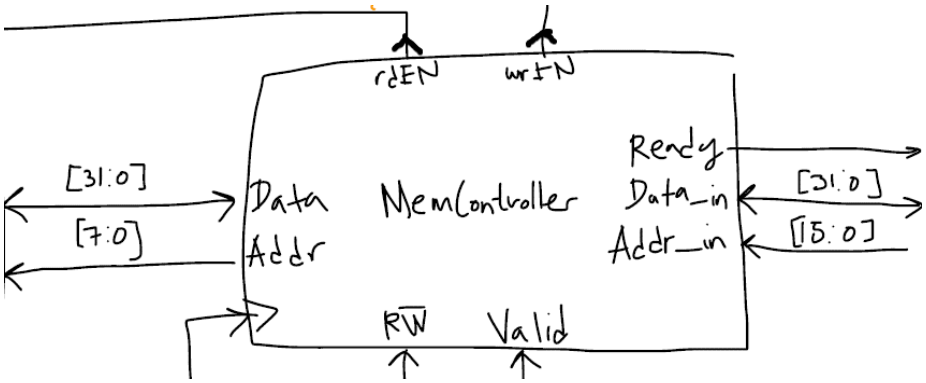
RAM

```
`include "HEADER.vh"

module RAM(Data, clk, rdEn, wrEn, reset, Addr);

parameter MEMDEPTH = 256;
parameter DWIDTH = 32;
parameter AWIDTH = 8;

inout [DWIDTH-1:0] Data;
input clk;
input rdEn, wrEn; // active high enable, one-hot
input [AWIDTH-1:0] Addr;
input reset;

tri [DWIDTH-1:0] Data;
reg [DWIDTH-1:0] dataOut;
reg [DWIDTH-1:0] storage [MEMDEPTH-1:0];

integer i;

assign Data = (~wrEn) ? storage[Addr] : {DWIDTH{1'bz}};

//Resets the entire RAM by iterating through the registers
always @(negedge reset) begin
    for (i = 0; i < MEMDEPTH; i = i + 1) begin
        storage[i] = 32'b0;
    end
end

//Initialize a DFF to set the data properly. See header.vh
`DFFE(storage[Addr], Data, wrEn, reset, clk)

endmodule
```



Memory Controller

```
module MemControl(Data_in, Data, rdEn, wrEn, Addr, Ready, clk, Addr_in, RW, Valid);

parameter MEMDEPTH = 256;
parameter DWIDTH = 32;
parameter AWIDTH = 8;

inout Data_in;
inout Data;

output reg rdEn, wrEn;
output reg [AWIDTH-
1:0] Addr;
output reg Ready;

input clk;
input [15:0] Addr_in;
input RW;
input Valid;
```

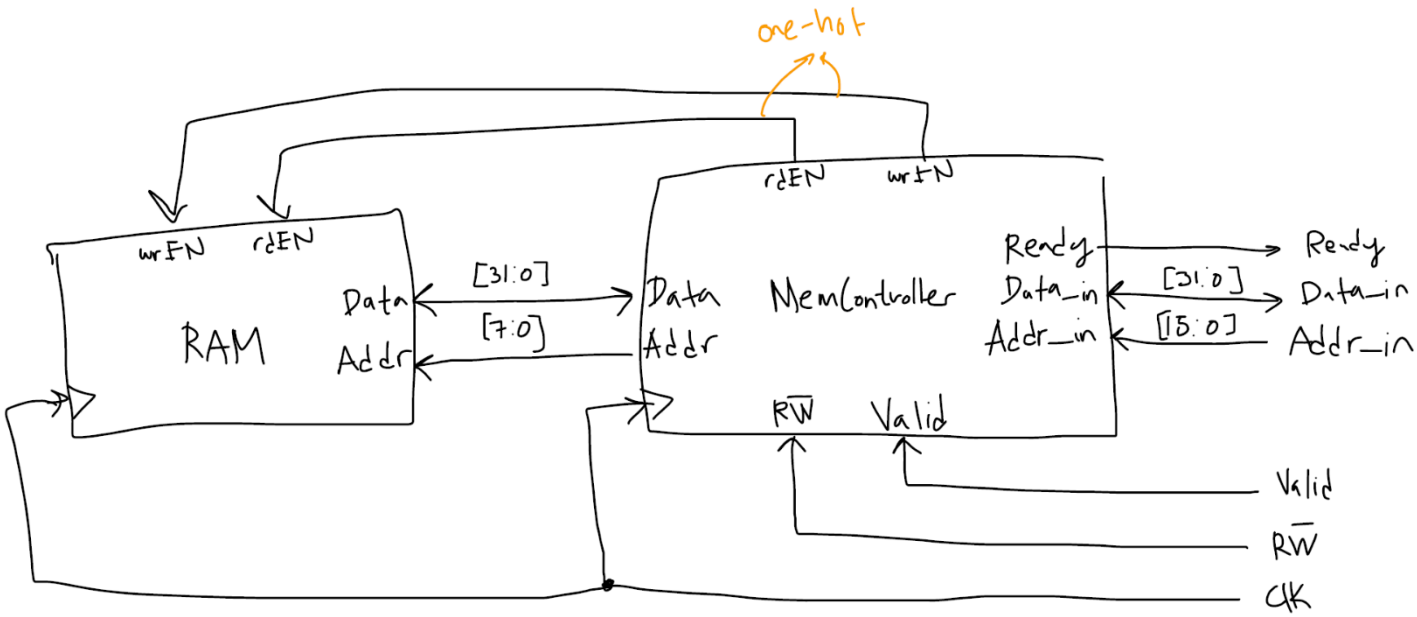



```
tri [DWIDTH-1:0] Data_in, Data;

assign Data = (wrEn) ? Data_in : {DWIDTH{1'bz}};
assign Data_in = (RW) ? Data : {DWIDTH{1'bz}};

always @(posedge clk) begin

    // only run when the inputs valid
    if(Valid) begin
        Ready = 0;

        // wait one cycle to read the address
        wait(~clk);
        wait(clk);

        Addr = Addr_in[AWIDTH-1:0];

        if (~RW)
            wrEn = 1; // write
        else
            rdEn = 1; // read

        // wait one cycle to write/read from ram
        wait(~clk);
        wait(clk);

        // add two cycles of arbitary delay
        wait(~clk);
        wait(clk);

        wait(~clk);
        wait(clk);

        // tell CPU we're done and disable RAM
        Ready = 1;
        rdEn = 0;
        wrEn = 0;
    end
end

endmodule
```

Header File

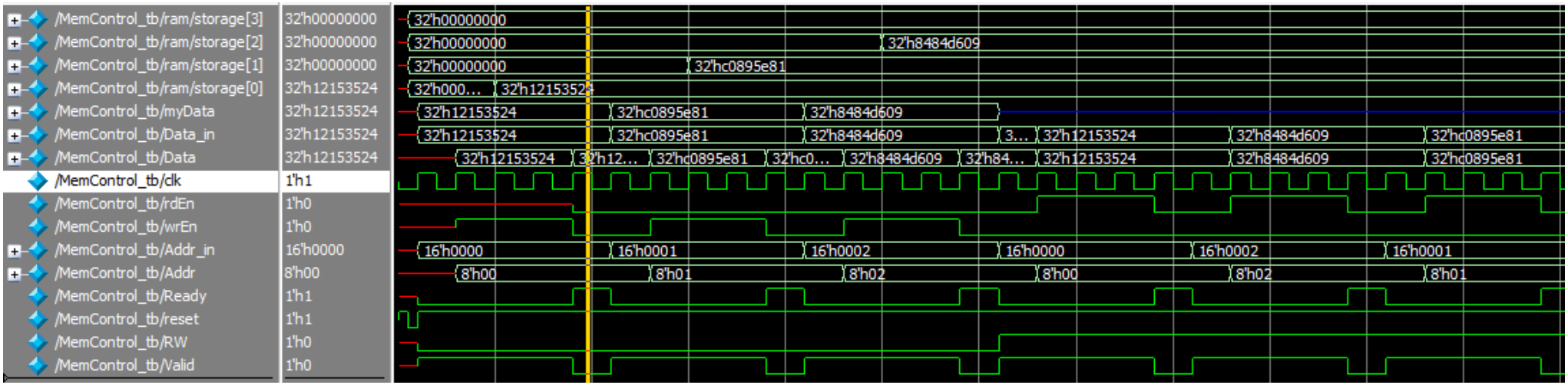
```
`ifndef HEADER
`define HEADER

//Macro to properly set the data of a reg in the RAM
`define DFFE(q, d, en, reset, clk) \
    always @(posedge clk) begin \
        if (reset) \
            q <= (en) ? d : q; \
    end

`endif
```

RAM / Memory Controller Combined Block-Diagram



Memory Controller Testbench

```
`timescale 1ns / 1ns

module MemControl_tb();

parameter MEMDEPTH = 256;
parameter DWIDTH = 32;
parameter AWIDTH = 8;

// component variables
tri [DWIDTH-1:0] Data_in, Data;

wire rdEn, wrEn;
wire [AWIDTH-1:0] Addr;
wire Ready;

reg clk;
reg reset;
reg [15:0] Addr_in;
reg RW;
reg Valid;

MemControl dut(Data_in, Data, rdEn, wrE
n, Addr, Ready, clk, Addr_in, RW, Valid
);
RAM ram(Data, clk, rdEn, wrEn, reset, A
ddr);

// clock
always begin
    clk = 0;
    forever #10 clk = !clk;
end

reg [DWIDTH-1:0] myData;

assign Data_in = (~RW) ? myData : {DWID
TH{1'bz}};

initial
begin
    // reset RAM
    reset = 1;
    #5 reset = 0;
    #5 reset = 1;
    // write to Addr 0
    myData = $random; Addr_in = 16'h000
0; RW = 0; Valid = 1;
    wait(~Ready);
    wait(Ready);
    Valid = 0;

    // write to Addr 1
    #20 myData = $random; Addr_in = 16'
h0001; RW = 0; Valid = 1;
    wait(~Ready);
    wait(Ready);
    Valid = 0;

    // write to Addr 2
    #20 myData = $random; Addr_in = 16'
h0002; RW = 0; Valid = 1;
    wait(~Ready);
    wait(Ready);
    Valid = 0;

    // read from Addr 0
    #20 myData = {DWIDTH{1'bz}}; Addr_i
n = 16'h0000; RW = 1; Valid = 1;
    wait(~Ready);
    wait(Ready);
    Valid = 0;

    // read from Addr 2
    #20 myData = {DWIDTH{1'bz}}; Addr_i
n = 16'h0002; RW = 1; Valid = 1;
    wait(~Ready);
    wait(Ready);
    Valid = 0;

    // read from Addr 1
    #20 myData = {DWIDTH{1'bz}}; Addr_i
n = 16'h0001; RW = 1; Valid = 1;
    wait(~Ready);
    wait(Ready);
    Valid = 0;
end

endmodule
```


/MemControl_tb/ram/storage[3]
/MemControl_tb/ram/storage[2]
/MemControl_tb/ram/storage[1]
/MemControl_tb/ram/storage[0]
/MemControl_tb/myData
/MemControl_tb/Data_in
/MemControl_tb/Data
/MemControl_tb/clk
/MemControl_tb/rdEn
/MemControl_tb/wrEn
/MemControl_tb/Addr_in
/MemControl_tb/Addr
/MemControl_tb/Ready
/MemControl_tb/reset
/MemControl_tb/RW
/MemControl_tb/Valid
32'h00000000
32'h12153524
32'hc0895e81
32'h8484d609
1'h1
1'h0
16'h0000
16'h0001
16'h0002
8'h00
8'h01
8'h02

Multiplier

```
module multiply_32(prod, a, b);

parameter WIDTH = 32;

output [WIDTH-1:0] prod;
input [WIDTH-1:0] a, b;

assign prod = a * b;
```

endmodule

Divider

```
module divide_32(out, a, b);

parameter WIDTH = 32;

output [WIDTH-1:0] out;
input [WIDTH-1:0] a,b;

assign out = a / b;

endmodule
```

Comparator

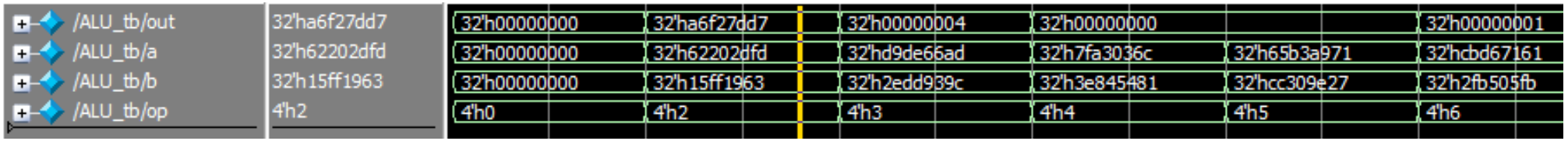
```
module comp_32(lt, eq, gt, a, b);

parameter WIDTH = 32;

output [WIDTH-1:0] lt, eq, gt;
input [WIDTH-1:0] a, b;

assign lt = {{31'b0}, {a < b}};
assign eq = {{31'b0}, {a == b}};
assign gt = {{31'b0}, {a > b}};

endmodule
```

ALU

```
module ALU(out, a, b, op);

output [31:0] out;
input [31:0] a, b;
input [3:0] op;

wire [31:0] add, sub, my_and, my_or, my_xor, my_xnor,
            my_shl, my_shr, my_sra, my_mult, my_div,
            my_lt, my_gt, my_eq;

full_adder_32 adder(, add, a, b, 1'b0);
full_subtractor_32 subber(, sub, a, b, 1'b0);
and32 ander(my_and, a, b);
or32 orer(my_or, a, b);
xor32 xorer(my_xor, a, b);
xnor32 xnorer(my_xnor, a, b);
shiftLeft32 shiftLefter(my_shl, a, b);
shiftRight32 shiftRighter(my_shr, a, b);
shiftRight32Arithmetic shiftRightAer(my_sra, a, b);
multiply_32 multiplierer(my_mult, a, b);
divide_32 dividerer(my_div, a, b);
comp_32 comparatorer(my_lt, my_eq, my_gt, a, b);

mux16_32 muxer(out, add, sub, my_mult, my_div, my_lt, my_eq, my_gt, ,
    my_and, my_or, my_xor, my_xnor, my_shl, my_shr, my_sra, , op);

endmodule
```

ALU Testbench

```
`timescale 1ns / 1ns

module ALU_tb();

wire [31:0] out;
reg [31:0] a, b;
reg [3:0] op;

ALU dut(out, a, b, op);

initial // Test stimulus
  begin
    #10  a = 32'hBC157222; b =  32'h222751CB; op = 4'h0;
    #10  a = 32'h00000000; b =  32'h583bd1cc; op = 4'h1;
    #10  a = 32'h10101010; b =  32'h583bd1cc; op = 4'h8;
    #10  a = 32'h0189ABCD; b =  32'h1DC74A54; op = 4'h9;
    #10  a = 32'he9eec208; b =  32'h583bd1cc; op = 4'hA;
    #10  a = 32'h1fbc8148; b =  32'h20ce01ee; op = 4'hB;
    #10  a = 32'h2416e099; b =  32'h3ef9d5ec; op = 4'h2;
    #10  a = 32'h687f3b5a; b =  32'h71252c6f; op = 4'h3;
    #10  a = 32'h555984ab; b =  32'h6c886316; op = 4'h4;
    #10  a = 32'hFFFEFFFF; b =  32'h00000004; op = 4'hC;
    #10  a = 32'hFFFF7FFF; b =  32'h00000009; op = 4'hD;
    #10  a = 32'h80000000; b =  32'h0000001E; op = 4'hE;
    #10  a = 32'h23337af9; b =  32'h89bfe491; op = 4'hF;
  // empty
    #10  a = 32'h00000000; b =  32'h00000000; op = 4'h0;

    #10  a = 32'h62202dfd; b =  32'h15ff1963; op = 4'h2;
    #10  a = 32'hd9de66ad; b =  32'h2edd939c; op = 4'h3;
    #10  a = 32'h7fa3036c; b =  32'h3e845481; op = 4'h4;
    #10  a = 32'h65b3a971; b =  32'hcc309e27; op = 4'h5;
    #10  a = 32'hcbd67161; b =  32'h2fb505fb; op = 4'h6;

    #10 $stop;
  end

endmodule
```



CPU

```
`include "header.vh"

module CPU;

parameter IWIDTH = 32;
parameter DWIDTH = 32;
parameter AWIDTH = 10;

reg clk;

// configure IR, MDR, MAR
reg [IWIDTH-1:0] IR;
reg [DWIDTH-1:0] MDR;
reg [AWIDTH-1:0] MAR;

wire [IWIDTH-1:0] IR_nxt;
wire [DWIDTH-1:0] MDR_nxt;
wire [AWIDTH-1:0] MAR_nxt;

`DFFE(IR, IR_nxt, 1'b1, 1'b1, clk);
`DFFE(MDR, MDR_nxt, 1'b1, 1'b1, clk);
`DFFE(MAR, MAR_nxt, 1'b1, 1'b1, clk);

// configure memory controller and RAM
tri [DWIDTH-1:0] Data_in, Data;

wire rdEn, wrEn;
wire [7:0] Addr;
wire Ready;

reg reset;
reg [15:0] Addr_in;
reg RW;
reg Valid;

assign Data_in = (~RW) ? MDR : {DWIDTH{1'bz}};

MemControl m_control(Data_in, Data, rdEn, wrEn, Addr, Ready, clk, MAR[7:0], RW, Valid)
;
RAM ram(Data, clk, rdEn, wrEn, reset, Addr);

// clock
always begin
    clk = 0;
    forever #10 clk = !clk;
end

endmodule
```