**T.E.S.S. Data Sheet**

Table of Contents

[Overview 3](#_Toc437946188)

[Features 3](#_Toc437946189)

[Abbreviations 3](#_Toc437946190)

[Block Diagram 4](#_Toc437946191)

[Layout 5](#_Toc437946192)

[Pin Layout 6](#_Toc437946193)

[Pin Table 6](#_Toc437946194)

[Peripheral interface 9](#_Toc437946195)

[Application 10](#_Toc437946196)

[Verilog 10](#_Toc437946197)

[Global Variables 10](#_Toc437946198)

[PC 14](#_Toc437946199)

[ALU 15](#_Toc437946200)

[Sign Extender 16](#_Toc437946201)

[Arbiter 16](#_Toc437946202)

[VGA Controller 16](#_Toc437946203)

[Memory Controller 19](#_Toc437946204)

[Reg Alu 21](#_Toc437946205)

[Controller Logic 23](#_Toc437946206)

[Clock Generators 35](#_Toc437946207)

[Sample Fetcher 36](#_Toc437946208)

[Sample Player 41](#_Toc437946209)

[SPI Interface 42](#_Toc437946210)

[SNES Interface 48](#_Toc437946211)

[Timer 51](#_Toc437946212)

[CPU 52](#_Toc437946213)

[System Top 54](#_Toc437946214)

[41 to 16 Multiplexer 55](#_Toc437946215)

[31 to 16 Multiplexer 55](#_Toc437946216)

[Tristate Buffer 56](#_Toc437946217)

[16-Bit Tristate Buffer 56](#_Toc437946218)

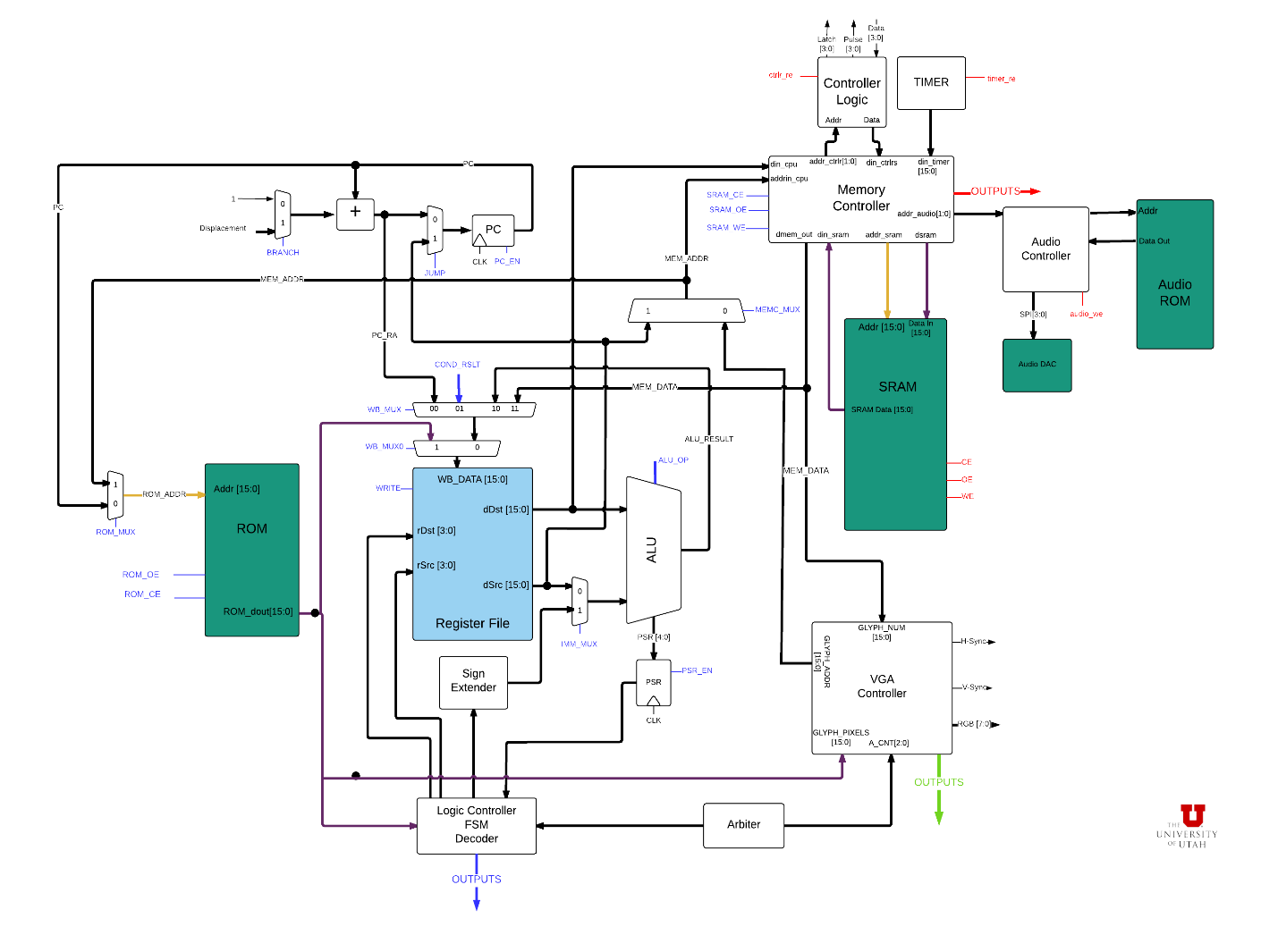
OverviewThe Titan Entertainment Super System (T.E.S.S) is a 16-bit microprocessor modeled after the CR16 architecture. With that being said, both its data path, and instruction set architecture (I.S.A) are very similar, but not identical to that of the CR16. The design also includes a sample-based audio playback system, as well as a 2-bit video graphics array (V.G.A) controller.  
  
An important feature of this integrated circuit (I.C) to note is that the memory needed to function needs to be supplied from off-chip memory I.C.’s . The specifications required for each memory I.C. is discussed in section 7.  
  
The T.E.S.S. was designed with the specific category of simple video games in mind. However the core was designed to be able to run a wide variety of applications. It is important to note that due to the memory I.C.’s being easily interchangeable, it easy to run several different applications, as long as they adhere to the I.S.A. discussed in later sections.

# Features

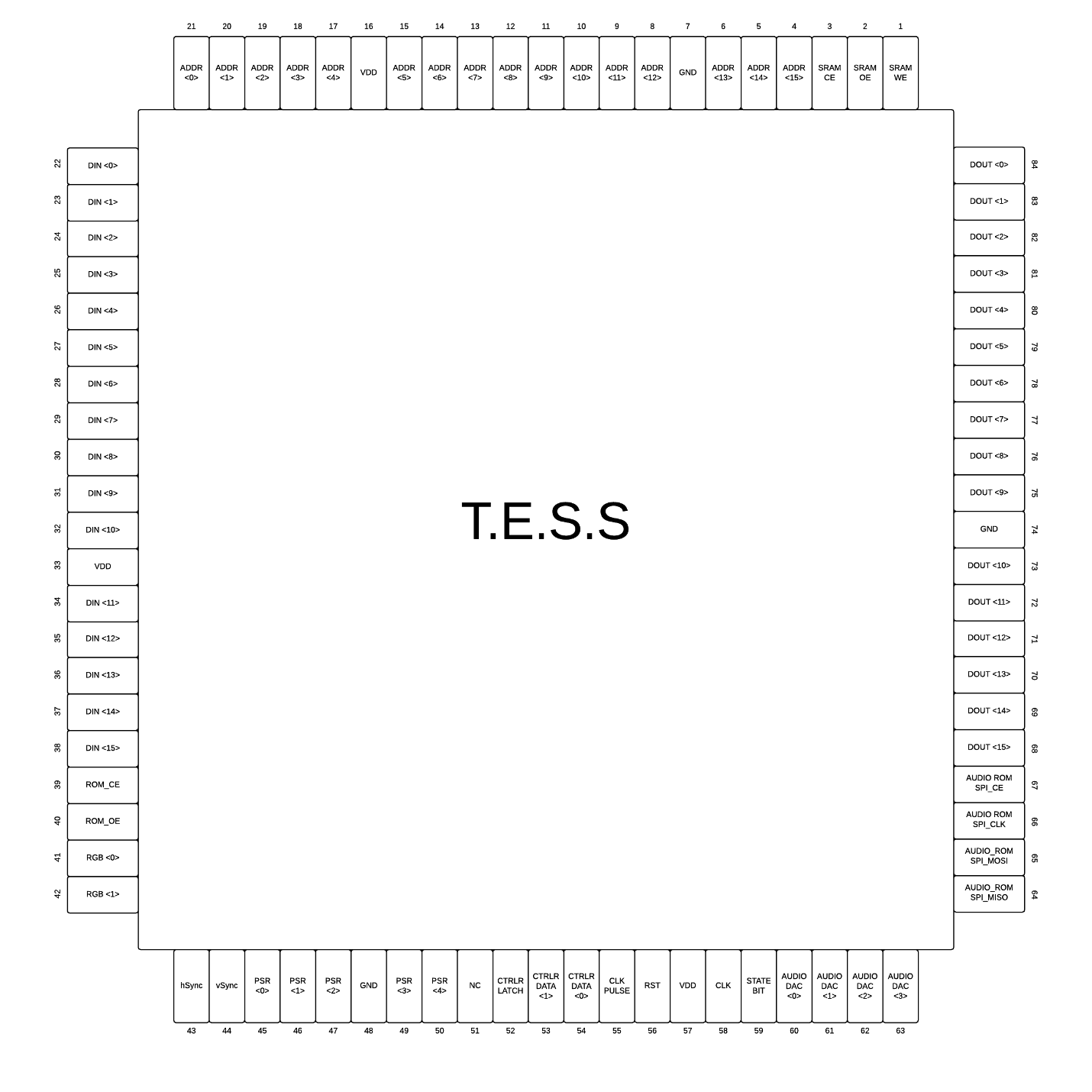
* 16-bit data path
* 2-bit VGA
* Sample-based audio playback
* 12.5 MHz clock speed
* General purpose

# Abbreviations

|  |  |
| --- | --- |
| Abbreviation | Meaning |
| ISA | Instruction Set Architecture |
| TESS | Titan Entertainment Super System |
| VGA | Video Graphics Array |
| IC | Integrated Circuit |
| ALU | Arithmetic Logic Unit |
| SRAM | Static Random Access Memory |
| ROM | Read Only Memory |
| SPI | Serial Periphrial Interface |
| PC | Program Counter |
| DAC | Digital-to-Analog Converter |
| FSM | Finite State Machine |

Block Diagram

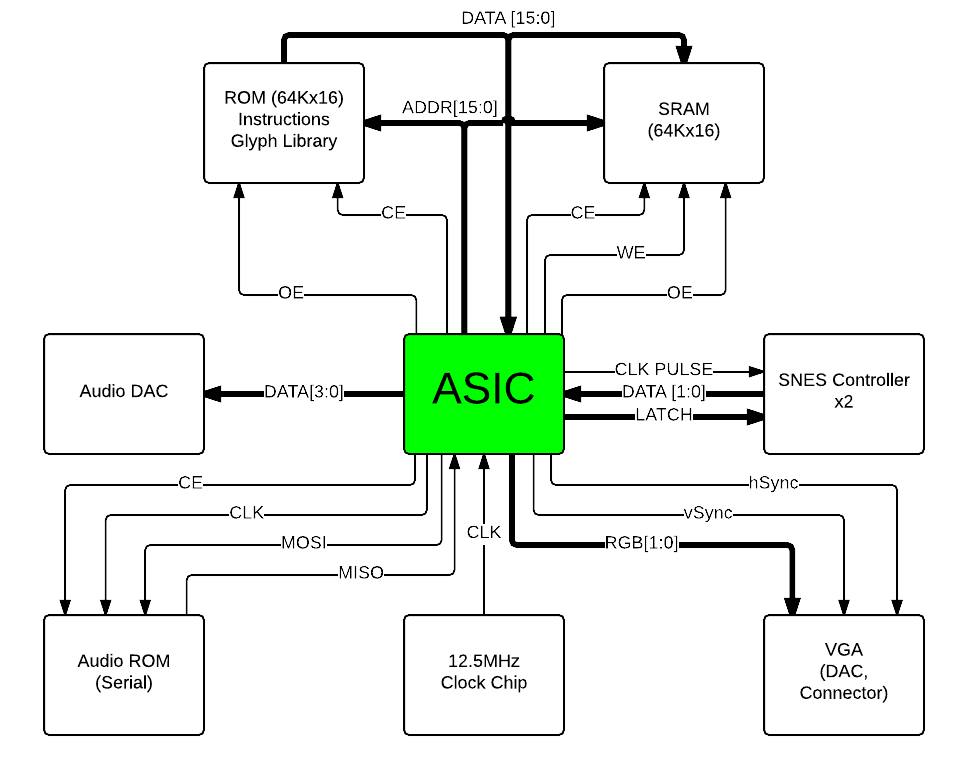
Layout

Pin Layout

# Pin Table

|  |  |  |
| --- | --- | --- |
| Name | Number | Description |
| SRAM WE | 1 | SRAM Write Enable |
| SRAM OE | 2 | SRAM Output Enable |
| SRAM CE | 3 | SRAM Chip Enable |
| ADDR <15> | 4 | Address Bit-15 connected to both SRAM and ROM |
| ADDR <14> | 5 | Address Bit-14 connected to both SRAM and ROM |
| ADDR <13> | 6 | Address Bit-13 connected to both SRAM and ROM |
| GND | 7 | Ground |
| ADDR <12> | 8 | Address Bit-12 connected to both SRAM and ROM |
| ADDR <11> | 9 | Address Bit-11 connected to both SRAM and ROM |
| ADDR <10> | 10 | Address Bit-10 connected to both SRAM and ROM |
| ADDR <9> | 11 | Address Bit-9 connected to both SRAM and ROM |
| ADDR <8> | 12 | Address Bit-8 connected to both SRAM and ROM |
| ADDR <7> | 13 | Address Bit-7 connected to both SRAM and ROM |
| ADDR <6> | 14 | Address Bit-6 connected to both SRAM and ROM |
| ADDR <5> | 15 | Address Bit-5 connected to both SRAM and ROM |
| VDD | 16 | Digital Power Supply |
| ADDR <4> | 17 | Address Bit-4 connected to both SRAM and ROM |
| ADDR <3> | 18 | Address Bit-3 connected to both SRAM and ROM |
| ADDR <2> | 19 | Address Bit-2 connected to both SRAM and ROM |
| ADDR <1> | 20 | Address Bit-1 connected to both SRAM and ROM |
| ADDR <0> | 21 | Address Bit-0 connected to both SRAM and ROM |
| DIN <0> | 22 | Data in Bit-0 connect to both SRAM and ROM |
| DIN <1> | 23 | Data in Bit-1 connect to both SRAM and ROM |
| DIN <2> | 24 | Data in Bit-2 connect to both SRAM and ROM |
| DIN <3> | 25 | Data in Bit-3 connect to both SRAM and ROM |
| DIN <4> | 26 | Data in Bit-4 connect to both SRAM and ROM |
| DIN <5> | 27 | Data in Bit-5 connect to both SRAM and ROM |
| DIN <6> | 28 | Data in Bit-6 connect to both SRAM and ROM |
| DIN <7> | 29 | Data in Bit-7 connect to both SRAM and ROM |
| DIN <8> | 30 | Data in Bit-8 connect to both SRAM and ROM |
| DIN <9> | 31 | Data in Bit-9 connect to both SRAM and ROM |
| DIN <10> | 32 | Data in Bit-10 connect to both SRAM and ROM |
| VDD | 33 | Digital Power Supply |
| DIN <11> | 34 | Data in Bit-11 connect to both SRAM and ROM |
| DIN <12> | 35 | Data in Bit-12 connect to both SRAM and ROM |
| DIN <13> | 36 | Data in Bit-13 connect to both SRAM and ROM |
| DIN <14> | 37 | Data in Bit-14 connect to both SRAM and ROM |
| DIN <15> | 38 | Data in Bit-15 connect to both SRAM and ROM |
| ROM\_CE | 39 | ROM Chip Enable |
| ROM\_OE | 40 | ROM Output Enable |
| RGB <0> | 41 | Least Significant Bit for VGA Color |
| RGB <1> | 42 | Most Significant Bit for VGA Color |
| hSync | 43 | Horizontal Sync required for VGA |
| vSync | 44 | Vertical Sync required for VGA |
| PSR <0> | 45 | Program Status Register Bit-0 used for Debugging |
| PSR <1> | 46 | Program Status Register Bit-1 used for Debugging |
| PSR <2> | 47 | Program Status Register Bit-2 used for Debugging |
| GND | 48 | Ground |
| PSR <3> | 49 | Program Status Register Bit-3 used for Debugging |
| PSR <4> | 50 | Program Status Register Bit-4 used for Debugging |
| CTRLR LATCH <1> | 51 | Controller 2 Latch signal |
| CTRLR LATCH <0> | 52 | Controller 1 Latch signal |
| CTRLR Data in <1> | 53 | Controller 2 Data Signal |
| CTRLR Data in <0> | 54 | Controller 1 Data Signal |
| CLK PULSE | 55 | Clock Output to Both Controllers |
| NC | 56 | Not Connected |
| VDD | 57 | Digital Power Supply |
| CLK | 58 | Clock Input |
| STATE BIT | 59 | Program State Bit used for Debugging |
| AUDIO DAC <0> | 60 | Audio DAC Output Bit-0 |
| AUDIO DAC <1> | 61 | Audio DAC Output Bit-1 |
| AUDIO DAC <2> | 62 | Audio DAC Output Bit-2 |
| AUDIO DAC <3> | 63 | Audio DAC Output Bit-3 |
| AUDIO\_ROM\_SPI\_MISO | 64 | Audio ROM SPI Master in Slave out |
| AUDIO\_ROM\_SPI\_MOSI | 65 | Audio ROM SPI Master out Slave in |
| AUDIO\_ROM\_SPI\_CLK | 66 | Audio rom SPI Clock |
| AUDIO\_ROM\_SPI\_CE | 67 | Audio Rom SPI Chip Enable |
| DOUT <15> | 68 | Data out Bit-15 Connected to SRAM Only |
| DOUT <14> | 69 | Data out Bit-14 Connected to SRAM Only |
| DOUT <13> | 70 | Data out Bit-13 Connected to SRAM Only |
| DOUT <12> | 71 | Data out Bit-12 Connected to SRAM Only |
| DOUT <11> | 72 | Data out Bit-11 Connected to SRAM Only |
| DOUT <10> | 73 | Data out Bit-10 Connected to SRAM Only |
| GND | 74 | Ground |
| DOUT <9> | 75 | Data out Bit-9 Connected to SRAM Only |
| DOUT <8> | 76 | Data out Bit-8 Connected to SRAM Only |
| DOUT <7> | 77 | Data out Bit-7 Connected to SRAM Only |
| DOUT <6> | 78 | Data out Bit-6 Connected to SRAM Only |
| DOUT <5> | 79 | Data out Bit-5 Connected to SRAM Only |
| DOUT <4> | 80 | Data out Bit-4 Connected to SRAM Only |
| DOUT <3> | 81 | Data out Bit-3 Connected to SRAM Only |
| DOUT <2> | 82 | Data out Bit-2 Connected to SRAM Only |
| DOUT <1> | 83 | Data out Bit-1 Connected to SRAM Only |
| DOUT <0> | 84 | Data out Bit-0 Connected to SRAM Only |

Application

The followingdisplays how the TESS would be deployed in a system. It is important to note that anything in the figure not highlighted green is located outside of the TESS package.

In order to ensure the desired application will run, as previously stated it must be stored on a ROM. Also a SRAM is required for the processor to store/access data as it executes the program. In order to ensure that it run’s appropriately the ROM and SRAM must have the following characteristics:

* Access Time greater than or equal to 80 nS
* Tristate output (meaning you must be able to disable its output bus because both the ROM and SRAM share the same address and data busses.)

# Verilog

This section contains the Verilog that was used to synthesize TESS.

## Global Variables

//////////////////////////

// Processor Sizing

`define DATAWIDTH 16 // register size

`define OPWIDTH 4

`define OPEXWIDTH 4

`define PRSWIDTH 5

`define ALUOPWIDTH 5

`define REGWIDTH 4

`define IMMWIDTH 8

// Memory data

`define ROM\_ADDR\_BITS 16

`define SRAM\_ADDR\_BITS 16

//////////////////////////

// Conditional Types

`define EQ 4'b0000 // equal

`define CS 4'b0010 // carry set

`define HI 4'b0100 // higher than (unsigned)

`define GT 4'b0110 // greater than (signed)

`define FS 4'b1000 // flag set - overflow

`define HS 4'b1011 // higher than or same as (unsigned)

`define GE 4'b1101 // greater than or equal (signed)

`define NE 4'b0001 // not equal

`define CC 4'b0011 // carry clear

`define LS 4'b0101 // lower than or same as (unsigned)

`define LE 4'b0111 // less than or equal (signed)

`define FC 4'b1001 // flag clear - no overflow

`define LO 4'b1010 // lower than (unsigned)

`define LT 4'b1100 // less than (signed)

`define UC 4'b1110 // branch unconditionally

`define CondType\_Ig 4'b1111 // Ignore

//////////////////////////

// Conditional Op codes

`define CondOp\_JAL 2'b01

`define CondOp\_BJ 2'b10

`define CondOp\_Ig 2'b00

//////////////////////////

// PSR indexes

`define psrN 4

`define psrZ 3

`define psrF 2

`define psrL 1

`define psrC 0

//////////////////////////

// Op codes (bits 15-12)

`define ADDI 4'b0101

`define ADDUI 4'b0110

//`define MULI 4'b1110

`define SUBI 4'b1001

`define CMPI 4'b1011

`define ANDI 4'b0001

`define ORI 4'b0010

`define XORI 4'b0011

`define MOVI 4'b1101

`define LUI 4'b1111

`define BCOND 4'b1100

`define RTYPE 4'b0000 // Register type

`define STYPE 4'b1000 // Shift type

`define OTYPE 4'b0100 // Other type (load/store/jal/branch...)

//////////////////////////

// Ext Op codes (bits 7-4)

// top op code is 4'b0000 == RTYPE

`define ADD 4'b0101

`define ADDU 4'b0110

//`define MUL 4'b1110

`define SUB 4'b1001

`define CMP 4'b1011

`define AND 4'b0001

`define OR 4'b0010

`define XOR 4'b0011

`define MOV 4'b1101

// top op code is 4'b1000 == ShiftTYPE

//`define SLL 4'b0001

//`define SLLI 4'b1001

//`define SRL 4'b0010

//`define SRLI 4'b1010

//`define SRA 4'b0011

//`define SRAI 4'b1011

`define LSH 4'b0100 // Logical shift (2's compliment)

`define LLSHI 4'b0000 // Logical left shift

`define LRSHI 4'b0001 // Logical right shift

`define ASHU 4'b0110 // Arithmetic shift (2's compliment)

`define ALSHUI 4'b0010 // Arithmetic left shift

`define ARSHUI 4'b0011 // Arithmetic right shift

// top op code is 4'b0100 == OtherType

`define LOAD 4'b0000

`define STOR 4'b0100

`define SCOND 4'b1101

`define JCOND 4'b1100

`define JAL 4'b1000

`define RLOAD 4'b1111

//////////////////////////

// ALU Op codes

`define ALUOp\_ADD 5'b00101

`define ALUOp\_ADDU 5'b00110

`define ALUOp\_SUB 5'b01001

//`define ALUOp\_MUL 5'b01110

`define ALUOp\_AND 5'b00001

`define ALUOp\_OR 5'b00010

`define ALUOp\_XOR 5'b00011

`define ALUOp\_MOV 5'b01101 // move

`define ALUOp\_LUI 5'b01111 // load upper immediate

`define ALUOp\_SLL 5'b10100 // shift left logical

`define ALUOp\_SRL 5'b10001 // shift right logical

`define ALUOp\_SLA 5'b10010 // shift left arithmetic

`define ALUOp\_SRA 5'b10011 // shift right arithmetic

//////////////////////////

// Datapath configurations {BRANCH, JUMP, RA\_MUX, ROM\_MUX, MEMC\_MUX, SRAM\_MUX, ALU\_BUF, VGA\_BUF, IMM\_MUX}

`define DP\_ITYPE 10'b0\_0\_0\_0\_

`define DP\_BCOND 10'b0\_0\_0\_1\_1\_11

`define DP\_RTYPE 10'b1\_0\_0\_0\_0\_11

`define DP\_CMP 10'b0\_0\_0\_0\_0\_11

`define DP\_CMPI 10'b0\_0\_0\_0\_1\_11

`define DP\_LOAD 10'b1\_0\_1\_0\_0\_10

`define DP\_STOR 10'b0\_1\_0\_0\_0\_10

`define DP\_SCOND 10'b1\_0\_0\_0\_0\_01

`define DP\_JCOND 10'b0\_0\_0\_0\_0\_01

`define DP\_JAL 10'b1\_0\_0\_0\_0\_00

// VGA 256x256 60Hz w/12.5MHz clk

// h\* is in clocks

`define hdisp 9'd320

`define hfp 4'd8

`define hpw 6'd48

`define hbp 5'd24

`define hMAX 9'd400

// v\* is in lines (relative to h)

`define vdisp 9'd480

`define vfp 4'd10

`define vpw 3'd2

`define vbp 5'd29

`define vMAX 10'd521

//////////////////////////

// FSM state codes

`define FETCH 0

`define DECODE 1

`define EXECUTE 2

`define BRANCH 3

`define JUMP 4

`define JMPAL 5

`define LOAD0 6

`define LOAD1 7

`define STORE 8

`define SCOND0 9

//////////////////////////

// MMIO Peripherial Addresses

`define CTRLR0 16'hFB40

`define CTRLR1 16'hFB41

`define CTRLR2 16'hFB42

`define CTRLR3 16'hFB43

`define AUDIOREG0 16'hFB44

`define AUDIOREG1 16'hFB45

`define AUDIOREG2 16'hFB46

`define AUDIOREG3 16'hFB47

`define TIMER0 16'hFB48

//////////////////////////

// MMIO Peripherial State Encoding

`define STATE\_CTRLR0 4'h0

`define STATE\_CTRLR1 4'h1

`define STATE\_CTRLR2 4'h2

`define STATE\_CTRLR3 4'h3

`define STATE\_AUDIOREG0 4'h4

`define STATE\_AUDIOREG1 4'h5

`define STATE\_AUDIOREG2 4'h6

`define STATE\_AUDIOREG3 4'h7

`define STATE\_TIMER0 4'h8

//////////////////////////

// Frame Buffer Base Address

`define FRAMEBUF 16'hFB50

## PC

`include "defines.v"

module pc

(input clk, rst, branch, jump, pcEn,

input [`IMMWIDTH-1:0] disp,

input [`DATAWIDTH-1:0] dSrc,

output [`DATAWIDTH-1:0] pc\_ra,

output reg [`DATAWIDTH-1:0] pc);

wire signed [`DATAWIDTH-1:0] bmux\_out, pc\_next;

always@(posedge clk, negedge rst) begin

if(!rst)

pc <= 0;

else if(pcEn)

pc <= pc\_next;

else

pc <= pc;

end

// Sign extend the displacement if branching => 8-bit branch [-127,127]

assign bmux\_out = (branch) ? {{8{disp[`IMMWIDTH-1]}},disp} : 16'd1;

// Add the dispalcement

assign pc\_ra = pc + bmux\_out;

// Consider jumps

assign pc\_next = (jump) ? dSrc : pc\_ra;

endmodule

## ALU

module alu

(input [`DATAWIDTH-1:0] dSrc, dDst,

input [`ALUOPWIDTH-1:0]opCode,

output [`PRSWIDTH-1:0] psrOut,

output [`DATAWIDTH-1:0] result

);

reg [`DATAWIDTH:0] temp;

wire signed [`DATAWIDTH-1:0] srcSigned = dSrc; // also acts as immediate

wire signed [`DATAWIDTH-1:0] dstSigned = dDst;

assign psrOut[`psrZ] = dSrc == dDst;

assign psrOut[`psrL] = dDst < dSrc; // compare unsigned

assign psrOut[`psrN] = dstSigned < srcSigned; // compare signed

assign psrOut[`psrF] = (srcSigned < 0 && dstSigned < 0 && result > 0) || (srcSigned > 0 && dstSigned > 0 && result < 0);

assign psrOut[`psrC] = temp[`DATAWIDTH];

assign result = temp[`DATAWIDTH-1:0];

always @(\*)

case(opCode)

`ALUOp\_ADD: temp = dstSigned + srcSigned;

`ALUOp\_ADDU: temp = dDst + dSrc;

`ALUOp\_SUB: temp = dstSigned - srcSigned;

//`ALUOp\_MUL: temp = dDst[7:0] \* dSrc[7:0];

`ALUOp\_AND: temp = dSrc & dDst;

`ALUOp\_OR: temp = dSrc | dDst;

`ALUOp\_XOR: temp = dSrc ^ dDst;

`ALUOp\_SLL: temp = dDst << dSrc[`REGWIDTH-1:0]; // can only shift by +/- 2\*\*REGWIDTH-1

`ALUOp\_SRL: temp = dDst >> dSrc[`REGWIDTH-1:0];

`ALUOp\_SLA: temp = dstSigned <<< dSrc[`REGWIDTH-1:0];

`ALUOp\_SRA: temp = dstSigned >>> dSrc[`REGWIDTH-1:0];

`ALUOp\_LUI: temp = {dSrc[`IMMWIDTH-1:0], dDst[`IMMWIDTH-1:0]};

`ALUOp\_MOV: temp = dSrc;

default: temp = 0;

endcase

endmodule

## Sign Extender

`include "defines.v"

module signext

(input [`IMMWIDTH-1:0] imm,

output [`DATAWIDTH-1:0] immExt);

assign immExt = {{8{imm[`IMMWIDTH-1]}}, imm};

endmodule

## Arbiter

`include "defines.v"

module arbiter

(input clk, rst,

output reg [2:0] count);

always@(posedge clk) begin

if(!rst)

count <= 0;

else // Count will roll over every 8 cycles.

count <= count + 1;

end

endmodule

## VGA Controller

`include "defines.v"

module vga\_ctrl

(input clk, rst,

input [2:0] acnt, // Arbiter clock count

input [`DATAWIDTH-1:0] glyph\_num, // glyph number from frame buffer in SRAM

input [`DATAWIDTH-1:0] glyph\_pixels, // glyph pixel information from ROM

output h\_sync, v\_sync, // VGA sync pulse signals

output [1:0] rgb, // 8-bit RGB value

output reg [`DATAWIDTH-1:0] glyph\_addr); // frame buffer/glyph library address

wire [8:0] h\_pixel;

wire [7:0] v\_pixel;

wire [`DATAWIDTH-1:0] sram\_addr, rom\_addr;

// timegen generates the VGA timing pulses, and determines the coordinates

// of the pixel that is currently being sent to the display.

vga\_timegen TimeGen(.clk(clk), .rst(rst), .h\_sync(h\_sync), .v\_sync(v\_sync), .bright(bright),

.h\_pixel(h\_pixel), .v\_pixel(v\_pixel));

// addrgen determines which glyph address in the 40x30 display grid needs

// to be fetched from block RAM, based on the horizontal and vertical position

vga\_FBaddr FrameBufAddr(.col(h\_pixel[8:4]), .row(v\_pixel[7:3]), .sram\_addr(sram\_addr));

// pixgen contains the glyph ROM, and outputs the byte for the current

// row of the current glyph, based on the glyph number and row number.

vga\_GLaddr GlyphLibAddr(.col(h\_pixel[3]), .row(v\_pixel[2:0]), .glyph(glyph\_num), .rom\_addr(rom\_addr));

// bitgen generates the RGB output based on the pixel\_data output from pixgen.

// bitgen will swap the 'on' and 'off' colors if the invert signal is asserted.

vga\_bitgen Bitgen(.bright(bright), .pixel\_data(glyph\_pixels), .h\_pixel(h\_pixel[2:0]), .rgb(rgb));

// Assign memory address depending on current clock cycle

always@(posedge clk) begin

case(acnt)

3'd0: glyph\_addr = sram\_addr;

3'd1: glyph\_addr = rom\_addr;

default: glyph\_addr = glyph\_addr;

endcase

end

endmodule

/// Counting clock ticks (h\_count) and horizontal lines (v\_count)

module vga\_timegen

(input clk, rst,

output h\_sync, v\_sync, bright,

output [8:0] h\_pixel, // pixel counts

output [7:0] v\_pixel);

reg [8:0] h\_ticks;

reg [9:0] v\_ticks;

// Count 400 horizontal ticks and 521 vertical ticks.

always @(posedge clk) begin

if(!rst) begin

h\_ticks <= 0;

v\_ticks <= 0;

end

else if (h\_ticks < `hMAX - 1) begin

h\_ticks <= h\_ticks + 1'b1;

v\_ticks <= v\_ticks;

end

else begin

h\_ticks <= 0;

if (v\_ticks < `vMAX - 1)

v\_ticks <= v\_ticks + 1'b1;

else

v\_ticks <= 0;

end

end

wire [8:0] temp\_v\_pixel;

// Derive the pixel counts and sync signals from the horizontal and vertical positions.

assign h\_pixel = (h\_ticks >= (`hpw + `hbp) && h\_ticks < (`hMAX - `hfp)) ? (h\_ticks - (`hpw + `hbp)) : 1'b0;

assign temp\_v\_pixel = (v\_ticks >= (`vpw + `vbp) && v\_ticks < (`vMAX - `vfp)) ? (v\_ticks - (`vpw + `vbp)) : 1'b0;

assign v\_pixel = temp\_v\_pixel >> 1;

assign h\_sync = (h\_ticks < `hpw) ? 1'b0 : 1'b1;

assign v\_sync = (v\_ticks < `vpw) ? 1'b0 : 1'b1;

assign bright = (h\_ticks >= (`hpw + `hbp) && h\_ticks < (`hMAX - `hfp) && v\_ticks >= (`vpw + `vbp) && v\_ticks < (`vMAX - `vfp)) ? 1'b1 : 1'b0;

endmodule

/// Calculates frame buffer address

module vga\_FBaddr

(input [4:0] col, // <-- based on h\_pixel

input [4:0] row, // <-- based on v\_pixel

output [`ROM\_ADDR\_BITS-1:0] sram\_addr); // address to SRAM to access frame buffer

// The screen is 40 glyphs wide, so we need to multiply

// row by 40 to get the right index. Pointer math, yo.

assign sram\_addr = (row \* 5'd20) + col + `FRAMEBUF;

endmodule

/// Gets pixel data from ROM

module vga\_GLaddr

(input col, // <-- based on h\_pixel

input [2:0] row, // <-- based on v\_pixel

input [`DATAWIDTH-1:0] glyph, // one of 256 glyphs

output [`DATAWIDTH-1:0] rom\_addr); // pixel data for entire horizontal row (8 pixels)

wire [(`DATAWIDTH/2)-1:0] glyph\_num = col ? glyph[`DATAWIDTH-1:(`DATAWIDTH/2)] : glyph[(`DATAWIDTH/2)-1:0];

assign rom\_addr = {5'b0, glyph\_num, row};

endmodule

/// Generates RGB value based on current pixel

module vga\_bitgen

(input bright,

input [`DATAWIDTH-1:0] pixel\_data,

input [2:0] h\_pixel,

output reg [1:0] rgb);

//reg[7:0] temp\_rgb;

always@(\*) begin

case(h\_pixel)

3'd0: rgb = (bright) ? pixel\_data[1:0] : 2'b0;

3'd1: rgb = (bright) ? pixel\_data[3:2] : 2'b0;

3'd2: rgb = (bright) ? pixel\_data[5:4] : 2'b0;

3'd3: rgb = (bright) ? pixel\_data[7:6] : 2'b0;

3'd4: rgb = (bright) ? pixel\_data[9:8] : 2'b0;

3'd5: rgb = (bright) ? pixel\_data[11:10] : 2'b0;

3'd6: rgb = (bright) ? pixel\_data[13:12] : 2'b0;

3'd7: rgb = (bright) ? pixel\_data[15:14] : 2'b0;

default: rgb = 2'b0;

endcase

end

endmodule

## Memory Controller

module mem\_ctrl

(input SRAM\_CE, input SRAM\_OE, input SRAM\_WE,

input ROM\_CE, input ROM\_OE,

input [`DATAWIDTH-1:0] din\_cpu,

input [11:0] din\_ctrlrs,

input [`DATAWIDTH-1:0] din\_timer,

input [`DATAWIDTH-1:0] addrin\_cpu,

input [`DATAWIDTH-1:0] EXT\_MEM\_DATA,

input [`DATAWIDTH-1:0] rom\_addr,

output reg CE, output reg OE, output reg WE, output reg ctrlr\_re,

output reg audio\_we,

output reg timer\_re,

output reg [1:0] addr\_ctrlr,

output reg [1:0] addr\_audio,

output reg [`DATAWIDTH-1:0] DOUT\_SRAM,

output reg [`DATAWIDTH-1:0] EXT\_MEM\_ADDR,

output reg [`DATAWIDTH-1:0] dmem);

// OUTPUT LOGIC

always@(\*) begin

CE = 1;

OE = 1;

WE = 1;

ctrlr\_re = 0;

audio\_we = 0;

timer\_re = 0;

addr\_ctrlr = 0;

addr\_audio = 0;

DOUT\_SRAM = 0;

EXT\_MEM\_ADDR = 0;

dmem = 16'd0;

// ROM access is in progress

if(~ROM\_OE & ~ROM\_CE) begin

EXT\_MEM\_ADDR = rom\_addr;

dmem = EXT\_MEM\_DATA;

end

// SRAM access is in progress

else begin

// Case on the address coming in to see if it relates to peripherials

case(addrin\_cpu[15:4])

12'hFDF: // Peripherial space

begin

case(addrin\_cpu[3:0])

`STATE\_CTRLR0:

begin

addr\_ctrlr = 2'b00;

ctrlr\_re = ~SRAM\_OE & SRAM\_WE;

dmem[11:0] = din\_ctrlrs;

end

`STATE\_CTRLR1:

begin

addr\_ctrlr = 2'b01;

ctrlr\_re = ~SRAM\_OE & SRAM\_WE;

dmem[11:0] = din\_ctrlrs;

end

`STATE\_CTRLR2:

begin

addr\_ctrlr = 2'b10;

ctrlr\_re = ~SRAM\_OE & SRAM\_WE;

dmem[11:0] = din\_ctrlrs;

end

`STATE\_CTRLR3:

begin

addr\_ctrlr = 2'b11;

ctrlr\_re = ~SRAM\_OE & SRAM\_WE;

dmem[11:0] = din\_ctrlrs;

end

`STATE\_AUDIOREG0:

begin

addr\_audio = 2'b00;

audio\_we = ~SRAM\_OE;

end

`STATE\_AUDIOREG1:

begin

addr\_audio = 2'b01;

audio\_we = ~SRAM\_OE;

end

`STATE\_AUDIOREG2:

begin

addr\_audio = 2'b10;

audio\_we = ~SRAM\_OE;

end

`STATE\_AUDIOREG3:

begin

addr\_audio = 2'b11;

audio\_we = ~SRAM\_OE;

end

`STATE\_TIMER0:

begin

timer\_re = ~SRAM\_OE & SRAM\_WE;

dmem = din\_timer;

end

default: ; // Address doesn't map to anything, so don't do anything

endcase // case(din\_cpu[3:0])

end

default: // Address reffers to SRAM data space

begin

CE = SRAM\_CE;

OE = SRAM\_OE;

WE = SRAM\_WE;

DOUT\_SRAM = din\_cpu;

EXT\_MEM\_ADDR = addrin\_cpu;

dmem = EXT\_MEM\_DATA;

end

endcase // case(din\_cpu[15:4])

end

end

endmodule

## Reg Alu

// Top-level connection of regfile and alu

`include "defines.v"

module reg\_alu

(input IMM\_MUX,

input COND\_RSLT,

input WB\_MUX0,

input [1:0] WB\_MUX,

input [`ALUOPWIDTH-1:0] aluOp,

input [`DATAWIDTH-1:0] drom,

input [`DATAWIDTH-1:0] pc\_ra,

input [`IMMWIDTH-1:0] imm\_in,

input [`DATAWIDTH-1:0] mem\_data,

input [`DATAWIDTH-1:0] dSrc, dDst,

output [`DATAWIDTH-1:0] wb\_data,

output [`PRSWIDTH-1:0] psrOut);

wire [`DATAWIDTH-1:0] alu\_Result;

wire [`DATAWIDTH-1:0] imm\_ex;

wire [`DATAWIDTH-1:0] alu\_rSrc;

wire [`DATAWIDTH-1:0] wb\_mux0;

alu \_alu(.dSrc(alu\_rSrc), .dDst(dDst), .opCode(aluOp), .psrOut(psrOut), .result(alu\_Result));

// mux for immediate value

assign alu\_rSrc = IMM\_MUX ? imm\_ex : dSrc;

mux41x16 regin\_mux(.cntrl(WB\_MUX), .arg0(pc\_ra), .arg1({15'b0, COND\_RSLT}), .arg2(alu\_Result), .arg3(mem\_data), .dout(wb\_mux0));

assign wb\_data = (WB\_MUX0) ? drom : wb\_mux0;

signext SignExtend (.imm(imm\_in), .immExt(imm\_ex));

endmodule

/// Sign Extender

module signext

(input [`IMMWIDTH-1:0] imm,

output [`DATAWIDTH-1:0] immExt);

assign immExt = {{8{imm[`IMMWIDTH-1]}}, imm};

endmodule

/// Arithmetic Logic Unit

module alu

(input [`DATAWIDTH-1:0] dSrc, dDst,

input [`ALUOPWIDTH-1:0]opCode,

output [`PRSWIDTH-1:0] psrOut,

output [`DATAWIDTH-1:0] result

);

reg [`DATAWIDTH:0] temp;

wire signed [`DATAWIDTH-1:0] srcSigned = dSrc; // also acts as immediate

wire signed [`DATAWIDTH-1:0] dstSigned = dDst;

assign psrOut[`psrZ] = dSrc == dDst;

assign psrOut[`psrL] = dDst < dSrc; // compare unsigned

assign psrOut[`psrN] = dstSigned < srcSigned; // compare signed

assign psrOut[`psrF] = (srcSigned < 0 && dstSigned < 0 && result > 0) || (srcSigned > 0 && dstSigned > 0 && result < 0);

assign psrOut[`psrC] = temp[`DATAWIDTH];

assign result = temp[`DATAWIDTH-1:0];

always @(\*)

case(opCode)

`ALUOp\_ADD: temp = dstSigned + srcSigned;

`ALUOp\_ADDU: temp = dDst + dSrc;

`ALUOp\_SUB: temp = dstSigned - srcSigned;

//`ALUOp\_MUL: temp = dDst[7:0] \* dSrc[7:0];

`ALUOp\_AND: temp = dSrc & dDst;

`ALUOp\_OR: temp = dSrc | dDst;

`ALUOp\_XOR: temp = dSrc ^ dDst;

`ALUOp\_SLL: temp = dDst << dSrc[`REGWIDTH-1:0]; // can only shift by +/- 2\*\*REGWIDTH-1

`ALUOp\_SRL: temp = dDst >> dSrc[`REGWIDTH-1:0];

`ALUOp\_SLA: temp = dstSigned <<< dSrc[`REGWIDTH-1:0];

`ALUOp\_SRA: temp = dstSigned >>> dSrc[`REGWIDTH-1:0];

`ALUOp\_LUI: temp = {dSrc[`IMMWIDTH-1:0], dDst[`IMMWIDTH-1:0]};

`ALUOp\_MOV: temp = dSrc;

default: temp = 0;

endcase

endmodule

/// 4 Input to 1 Output Mux with 16-bit data width

module mux41x16

(input [1:0] cntrl,

input [`DATAWIDTH-1:0] arg0,

input [`DATAWIDTH-1:0] arg1,

input [`DATAWIDTH-1:0] arg2,

input [`DATAWIDTH-1:0] arg3,

output reg [`DATAWIDTH-1:0] dout

);

always@(\*) begin

case(cntrl)

2'b00: dout = arg0;

2'b01: dout = arg1;

2'b10: dout = arg2;

2'b11: dout = arg3;

default: dout = 16'd0;

endcase

end

endmodule

## Controller Logic

`include "defines.v"

module controller

(input clk, rst,

input [`PRSWIDTH-1:0] psr\_in,

input [`DATAWIDTH-1:0] instruction,

input [2:0] acnt,

output reg ps,

output reg BRANCH,

output reg JUMP,

output reg ROM\_MUX,

output reg MEMC\_MUX,

output reg IMM\_MUX,

output reg PC\_EN,

output reg WRITE,

output reg SRAM\_CE,

output reg SRAM\_OE,

output reg SRAM\_WE,

output reg ROM\_CE,

output reg ROM\_OE,

output reg COND\_RSLT,

output reg WB\_MUX0,

output reg [1:0] WB\_MUX,

output reg [`REGWIDTH-1:0] rDst,

output reg [`REGWIDTH-1:0] rSrc,

output reg [`IMMWIDTH-1:0] imm\_val,

output reg [`ALUOPWIDTH-1:0] ALU\_OP);

// FSM states

//reg ps;

reg ns;

// Internal registers/control signals

reg PSR\_EN, INST\_EN;

reg [`PRSWIDTH-1:0] psr;

reg [`DATAWIDTH-1:0] inst;

// Latch the instruction when enabled

always@(posedge clk) begin

if(INST\_EN)

inst <= instruction;

else

inst <= inst;

end

// Latch PSR only when enabled

always@(posedge clk) begin

if(PSR\_EN)

psr <= psr\_in;

else

psr <= psr;

end

// Present State

always@(posedge clk, negedge rst) begin

if(!rst)

ps <= `FETCH;

else

// The arbiter is an 8-bit counter to synchronize VGA and

// CPU memory accesses. The CPU is scheduled to operate only

// on cycles 2 - 8, and VGA is scheduled on cycles 0 & 1.

case(acnt)

3'd0: ps <= ps;

3'd1: ps <= ps;

default: ps <= ns;

endcase

end

// Next State

always@(\*) begin

case(ps)

`FETCH: ns = `DECODE;

`DECODE: // Decode instruction

begin

//inst[15:12]

case(inst[`DATAWIDTH-1:`DATAWIDTH-`OPWIDTH])

// I-TYPES

`ADDI: ns = `FETCH;

`ADDUI: ns = `FETCH;

`SUBI: ns = `FETCH;

`CMPI: ns = `FETCH;

`ANDI: ns = `FETCH;

`ORI: ns = `FETCH;

`XORI: ns = `FETCH;

`MOVI: ns = `FETCH;

`LUI: ns = `FETCH;

// R-TYPE

`RTYPE:

begin

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH])

`ADD: ns = `FETCH;

`ADDU: ns = `FETCH;

`SUB: ns = `FETCH;

`CMP: ns = `FETCH;

`AND: ns = `FETCH;

`OR: ns = `FETCH;

`XOR: ns = `FETCH;

`MOV: ns = `FETCH;

default: ns = `FETCH;

endcase

end

// O-TYPE

`OTYPE:

begin

//inst[7:4]

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH])

`LOAD: ns = `FETCH; //`LOAD0;

`STOR: ns = `FETCH;

`SCOND: ns = `FETCH;

`JCOND: ns = `FETCH;

`JAL: ns = `FETCH;

`RLOAD: ns = `FETCH;

default: ns = `FETCH;

endcase

end

// BRANCH-TYPE

`BCOND: ns = `FETCH;

// SHIFT-TYPE

`STYPE:

begin

//inst[7:4]

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH])

`LSH: ns = `FETCH;

`LLSHI: ns = `FETCH;

`LRSHI: ns = `FETCH;

`ASHU: ns = `FETCH;

`ALSHUI: ns = `FETCH;

`ARSHUI: ns = `FETCH;

default: ns = `FETCH;

endcase

end

// Unknown op-codes

default: ns = `FETCH;

endcase

end

default: ns = `FETCH;

endcase

end

// Output Logic

always@(\*) begin

BRANCH = 0;

JUMP = 0;

ROM\_MUX = 0;

MEMC\_MUX = 0;

WB\_MUX = 2'b10;

WB\_MUX0 = 0;

//ALU\_BUF = 0;

IMM\_MUX = 0;

PSR\_EN = 0;

PC\_EN = 0;

INST\_EN = 0;

WRITE = 0;

SRAM\_CE = 1; // Active low signal

SRAM\_OE = 1; // Active low signal

SRAM\_WE = 1; // Active low signal

ROM\_CE = 1; // Active low signal

ROM\_OE = 1; // Active low signal

COND\_RSLT = 0;

rDst = `REGWIDTH'b0;

rSrc = `REGWIDTH'b0;

imm\_val = `IMMWIDTH'b0;

ALU\_OP = `ALUOPWIDTH'b0;

case(ps)

`FETCH: // Enable reading from ROM

begin

ROM\_CE = 0;

ROM\_OE = 0;

INST\_EN = 1;

end

`DECODE:

begin

case(inst[`DATAWIDTH-1:`DATAWIDTH-`OPWIDTH])

// I-TYPES

`ADDI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_ADD;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`ADDUI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_ADDU;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`SUBI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_SUB;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`CMPI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_SUB;

IMM\_MUX = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`ANDI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_AND;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`ORI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_OR;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`XORI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_XOR;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`MOVI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_MOV;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`LUI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

ALU\_OP = `ALUOp\_LUI;

IMM\_MUX = 1;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`RTYPE:

begin

//inst[7:4]

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH])

`ADD:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_ADD;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`ADDU:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_ADDU;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`SUB:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_SUB;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`CMP:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_SUB;

PSR\_EN = 1;

PC\_EN = 1;

end

`AND:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_AND;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`OR:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_OR;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`XOR:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_XOR;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

`MOV:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_MOV;

WRITE = 1;

PSR\_EN = 1;

PC\_EN = 1;

end

default:

begin

;

end

endcase

end

// O-TYPE

`OTYPE:

begin

//inst[7:4]

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH])

`LOAD: // From SRAM

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

MEMC\_MUX = 1;

WRITE = 1;

WB\_MUX = 2'b11;

PC\_EN = 1;

SRAM\_CE = 0;

SRAM\_OE = 0;

end

`STOR: // To SRAM

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

MEMC\_MUX = 1;

PC\_EN = 1;

SRAM\_CE = 0;

SRAM\_OE = 0;

SRAM\_WE = 0;

end

`SCOND:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

WB\_MUX = 2'b01;

WRITE = 1;

PC\_EN = 1;

// Check PSR

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0])

`EQ: COND\_RSLT = (psr[`psrZ]) ? 1'b1 : 1'b0;

`CS: COND\_RSLT = (psr[`psrC]) ? 1'b1 : 1'b0;

`HI: COND\_RSLT = (psr[`psrL]) ? 1'b1 : 1'b0;

`GT: COND\_RSLT = (psr[`psrN]) ? 1'b1 : 1'b0;

`FS: COND\_RSLT = (psr[`psrF]) ? 1'b1 : 1'b0;

`HS: COND\_RSLT = (psr[`psrZ] | psr[`psrL]) ? 1'b1 : 1'b0;

`GE: COND\_RSLT = (psr[`psrZ] | psr[`psrN]) ? 1'b1 : 1'b0;

`NE: COND\_RSLT = (!psr[`psrZ]) ? 1'b1 : 1'b0;

`CC: COND\_RSLT = (!psr[`psrC]) ? 1'b1 : 1'b0;

`LS: COND\_RSLT = (!psr[`psrL]) ? 1'b1 : 1'b0;

`LE: COND\_RSLT = (!psr[`psrN]) ? 1'b1 : 1'b0;

`FC: COND\_RSLT = (!psr[`psrF]) ? 1'b1 : 1'b0;

`LO: COND\_RSLT = (!psr[`psrZ] & !psr[`psrL]) ? 1'b1 : 1'b0;

`LT: COND\_RSLT = (!psr[`psrZ] & !psr[`psrN]) ? 1'b1 : 1'b0;

`UC: COND\_RSLT = 1'b0;

default: COND\_RSLT = 1'b0;

endcase

end

`JCOND:

begin

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

PC\_EN = 1;

// Check PSR

case(inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH])

`EQ: JUMP = (psr[`psrZ]) ? 1'b1 : 1'b0;

`CS: JUMP = (psr[`psrC]) ? 1'b1 : 1'b0;

`HI: JUMP = (psr[`psrL]) ? 1'b1 : 1'b0;

`GT: JUMP = (psr[`psrN]) ? 1'b1 : 1'b0;

`FS: JUMP = (psr[`psrF]) ? 1'b1 : 1'b0;

`HS: JUMP = (psr[`psrZ] | psr[`psrL]) ? 1'b1 : 1'b0;

`GE: JUMP = (psr[`psrZ] | psr[`psrN]) ? 1'b1 : 1'b0;

`NE: JUMP = (!psr[`psrZ]) ? 1'b1 : 1'b0;

`CC: JUMP = (!psr[`psrC]) ? 1'b1 : 1'b0;

`LS: JUMP = (!psr[`psrL]) ? 1'b1 : 1'b0;

`LE: JUMP = (!psr[`psrN]) ? 1'b1 : 1'b0;

`FC: JUMP = (!psr[`psrF]) ? 1'b1 : 1'b0;

`LO: JUMP = (!psr[`psrZ] & !psr[`psrL]) ? 1'b1 : 1'b0;

`LT: JUMP = (!psr[`psrZ] & !psr[`psrN]) ? 1'b1 : 1'b0;

`UC: JUMP = 1'b1;

default: JUMP = 1'b0;

endcase

end

`JAL:

begin

JUMP = 1;

rDst = `REGWIDTH'd15;

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

WRITE = 1;

WB\_MUX = 2'b0;

PC\_EN = 1;

end

`RLOAD:

begin

ROM\_CE = 0;

ROM\_OE = 0;

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

MEMC\_MUX = 1;

WRITE = 1;

WB\_MUX0 = 1;

PC\_EN = 1;

end

default: ;

endcase

end

// BRANCH-TYPE

`BCOND:

begin

imm\_val = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:0];

IMM\_MUX = 1;

PC\_EN = 1;

// Check PSR

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0])

`EQ: BRANCH = (psr[`psrZ]) ? 1'b1 : 1'b0;

`CS: BRANCH = (psr[`psrC]) ? 1'b1 : 1'b0;

`HI: BRANCH = (psr[`psrL]) ? 1'b1 : 1'b0;

`GT: BRANCH = (psr[`psrN]) ? 1'b1 : 1'b0;

`FS: BRANCH = (psr[`psrF]) ? 1'b1 : 1'b0;

`HS: BRANCH = (psr[`psrZ] | psr[`psrL]) ? 1'b1 : 1'b0;

`GE: BRANCH = (psr[`psrZ] | psr[`psrN]) ? 1'b1 : 1'b0;

`NE: BRANCH = (!psr[`psrZ]) ? 1'b1 : 1'b0;

`CC: BRANCH = (!psr[`psrC]) ? 1'b1 : 1'b0;

`LS: BRANCH = (!psr[`psrL]) ? 1'b1 : 1'b0;

`LE: BRANCH = (!psr[`psrN]) ? 1'b1 : 1'b0;

`FC: BRANCH = (!psr[`psrF]) ? 1'b1 : 1'b0;

`LO: BRANCH = (!psr[`psrZ] & !psr[`psrL]) ? 1'b1 : 1'b0;

`LT: BRANCH = (!psr[`psrZ] & !psr[`psrN]) ? 1'b1 : 1'b0;

`UC: BRANCH = 1'b0;

default: BRANCH = 1'b0;

endcase

end

// SHIFT-TYPE

`STYPE:

begin

//inst[7:4]

case(inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH])

`LSH:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_SLL;

WRITE = 1;

PC\_EN = 1;

end

`LLSHI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = {4'b0,inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0]};

ALU\_OP = `ALUOp\_SLL;

WRITE = 1;

PC\_EN = 1;

end

`LRSHI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = {4'b0,inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0]};

ALU\_OP = `ALUOp\_SRL;

WRITE = 1;

PC\_EN = 1;

end

`ASHU:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

rSrc = inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0];

ALU\_OP = `ALUOp\_SLA;

WRITE = 1;

PC\_EN = 1;

end

`ALSHUI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = {4'b0,inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0]};

ALU\_OP = `ALUOp\_SLA;

WRITE = 1;

PC\_EN = 1;

end

`ARSHUI:

begin

rDst = inst[`DATAWIDTH-`OPWIDTH-1:`DATAWIDTH-`OPWIDTH-`REGWIDTH];

imm\_val = {4'b0,inst[`DATAWIDTH-`OPWIDTH-`REGWIDTH-`OPEXWIDTH-1:0]};

ALU\_OP = `ALUOp\_SRA;

WRITE = 1;

PC\_EN = 1;

end

default: ;

endcase

end

// Unknown op-codes

default: ;

end

default:

begin

;

end

endcase

end

endmodule

## Clock Generators

module ClockGenerator

( input clk\_in,

input reset,

output clk\_out\_0, // LRCLK

output clk\_out\_1 // SCLK);

localparam [8:0] comp\_val\_0 = 9'd383;

localparam [3:0] comp\_val\_1 = 4'd11;

// I2S Left-Right Clock Divider

ClockDivider #( .COUNT\_WIDTH(9) ) clk\_div\_0

( .clk\_in (clk\_in),

.reset (reset),

.comp\_val (comp\_val\_0),

.clk\_out (clk\_out\_0) );

// I2S Serial Clock Divider

ClockDivider #( .COUNT\_WIDTH(4) ) clk\_div\_1

(.clk\_in (clk\_in),

.reset (reset),

.comp\_val (comp\_val\_1),

.clk\_out (clk\_out\_1));

endmodule

module ClockDivider

#(parameter COUNT\_WIDTH = 9)

( input clk\_in,

input reset,

input [COUNT\_WIDTH - 1 : 0] comp\_val,

output reg clk\_out);

reg [COUNT\_WIDTH - 1 : 0] div\_count;

always @(posedge clk\_in) begin

if (reset) begin

div\_count <= 0;

clk\_out <= 0;

end

else if (div\_count < comp\_val) begin

div\_count <= div\_count + 1'b1;

clk\_out <= clk\_out;

end

else begin

div\_count <= 0;

clk\_out <= ~clk\_out;

end

end

endmodule

## Sample Fetcher

module SampleFetcher

(input clk,

input reset,

input trigger\_0,

input trigger\_1,

input trigger\_2,

input trigger\_3,

input lrclk,

input [7:0] spi\_data,

input spi\_ce\_n,

output reg [23:0] spi\_addr,

output reg start\_read,

output reg [10:0] sample\_data

);

reg [10:0] next\_sample;

reg [1:0] channel\_count;

wire [14:0] count\_0;

wire [14:0] count\_1;

wire [14:0] count\_2;

wire [14:0] count\_3;

parameter RESET = 3'd0;

parameter IDLE = 3'd1;

parameter FETCH = 3'd2;

parameter WAIT = 3'd3;

parameter ACCUM = 3'd4;

parameter WAIT2 = 3'd5;

always@(\*) begin

case (channel\_count)

2'd0 : spi\_addr = { 9'd0 , count\_0 };

2'd1 : spi\_addr = { 9'd1 , count\_1 };

2'd2 : spi\_addr = { 9'd2 , count\_2 };

2'd3 : spi\_addr = { 9'd3 , count\_3 };

endcase

end

reg [2:0] state;

always@(posedge clk) begin

start\_read <= 1'b0;

sample\_data <= sample\_data;

next\_sample <= next\_sample;

channel\_count <= channel\_count;

if (reset) begin

state <= RESET;

sample\_data <= 11'b0;

channel\_count <= 2'd0;

end

else begin

case (state)

RESET : begin

state <= IDLE;

end

IDLE : begin

if (lrclk) begin

state <= IDLE;

end

else begin

sample\_data <= next\_sample;

next\_sample <= 11'b0;

channel\_count <= 2'd0;

state <= FETCH;

end

end

FETCH : begin

start\_read <= 1'b1;

if (spi\_ce\_n) begin

state <= FETCH;

end

else begin

state <= WAIT;

end

end

WAIT : begin

if (spi\_ce\_n) begin

state <= ACCUM;

end

else begin

state <= WAIT;

end

end

ACCUM : begin

next\_sample <= $signed(next\_sample) + $signed(spi\_data);

if (channel\_count == 2'd3) begin

state <= WAIT2;

end

else begin

channel\_count <= channel\_count + 1'b1;

state <= FETCH;

end

end

WAIT2 : begin

if (~lrclk) begin

state <= WAIT2;

end

else begin

state <= IDLE;

end

end

default : begin

state <= RESET;

end

endcase

end

end

AddressCounter addr\_counter\_0

( .clk(clk),

.reset(reset),

.lrclk(lrclk),

.count\_reset(trigger\_0),

.count(count\_0));

AddressCounter addr\_counter\_1

( .clk(clk),

.reset(reset),

.lrclk(lrclk),

.count\_reset(trigger\_1),

.count(count\_1));

AddressCounter addr\_counter\_2

( .clk(clk),

.reset(reset),

.lrclk(lrclk),

.count\_reset(trigger\_2),

.count(count\_2));

AddressCounter addr\_counter\_3

( .clk(clk),

.reset(reset),

.lrclk(lrclk),

.count\_reset(trigger\_3),

.count(count\_3));

endmodule

module AddressCounter

( input clk,

input reset,

input lrclk,

input count\_reset,

output reg [14:0] count);

localparam [14:0] SAMPLE\_COUNT = 15'd32767;

// generate a pulse for every low-to-high transition of the LR clock

reg a, b;

wire lr\_trig = a & ~b;

always@(posedge clk) begin

a <= lrclk;

b <= a;

end

parameter RESET = 2'd0;

parameter IDLE = 2'd1;

parameter COUNT = 2'd2;

parameter DONE = 2'd3;

reg [1:0] state;

always@(posedge clk) begin

if (reset) begin

state <= RESET;

count <= 15'd0;

end

else if (count\_reset) begin

state <= COUNT;

count <= 15'd0;

end

else begin

case (state)

RESET : begin

state <= IDLE;

count <= 15'd0;

end

IDLE : begin

state <= IDLE;

count <= 15'd0;

end

COUNT : begin

if (count == SAMPLE\_COUNT) begin

state <= IDLE;

count <= 15'd0;

end

else begin

state <= COUNT;

if (lr\_trig)

count <= count + 1'b1;

else

count <= count;

end

end

default : begin

state <= IDLE;

count <= 15'd0;

end

endcase

end

end

endmodule

## Sample Player

module SamplePlayer(

input clk,

input reset,

input read\_enable,

input [1:0] read\_addr,

output i2s\_mclk,

output i2s\_lrclk,

output i2s\_sclk,

output i2s\_sdata,

input spi\_miso,

output spi\_mosi,

output spi\_clk,

output spi\_ce\_n);

wire spi\_start\_read;

wire spi\_data\_ready;

wire [7:0] spi\_data;

wire [23:0] spi\_addr;

wire [10:0] sample\_data;

// internal trigger signals

wire trig\_0 = read\_enable & (read\_addr == 2'd0);

wire trig\_1 = read\_enable & (read\_addr == 2'd1);;

wire trig\_2 = read\_enable & (read\_addr == 2'd2);;

wire trig\_3 = read\_enable & (read\_addr == 2'd3);;

assign i2s\_mclk = clk;

// system reset is active low

wire inv\_reset = ~reset;

ClockGenerator clkgen\_0

( .clk\_in(clk),

.reset(reset),

.clk\_out\_0(i2s\_lrclk),

.clk\_out\_1(i2s\_sclk));

SpiInterface spi\_0

( .clk\_in(clk),

.reset(reset),

.start\_read(spi\_start\_read),

.read\_addr(spi\_addr),

.miso(spi\_miso),

.mosi(spi\_mosi),

.ce\_n(spi\_ce\_n),

.sclk(spi\_clk),

.data\_ready(spi\_data\_ready),

.spi\_data(spi\_data));

SampleFetcher sample\_fetcher\_0

( .clk(clk),

.reset(reset),

.trigger\_0(trig\_0),

.trigger\_1(trig\_1),

.trigger\_2(trig\_2),

.trigger\_3(trig\_3),

.lrclk(i2s\_lrclk),

.spi\_data(spi\_data),

.spi\_ce\_n(spi\_ce\_n),

.spi\_addr(spi\_addr),

.start\_read(spi\_start\_read),

.sample\_data(sample\_data));

I2sInterface i2s\_0

( .clk(clk),

.reset(reset),

.lrclk(i2s\_lrclk),

.sclk(i2s\_sclk),

.sample\_data(sample\_data),

.data\_out(i2s\_sdata));

Endmodule

## SPI Interface

module SpiInterface

( input clk\_in, // system clock (12.5 MHz)

input reset, // synchronous reset

input start\_read, // triggers a read from the input address

input [0:23] read\_addr, // read address

input miso, // serial input from SPI slave

output reg mosi, // serial out to SPI slave

output reg ce\_n, // active-low chip enable for SPI slave

output reg sclk, // SPI serial clock

output reg data\_ready, // signals that the data is ready to be read

output reg [7:0] spi\_data // the byte read from the SPI slave

);

parameter RESET = 3'd0;

parameter IDLE = 3'd1;

parameter ENABLE\_CHIP = 3'd2;

parameter ISSUE\_CMD = 3'd3;

parameter ISSUE\_ADDR = 3'd4;

parameter RECEIVE\_DATA = 3'd5;

parameter DISABLE\_CHIP = 3'd6;

parameter MOVE\_DATA = 3'd7;

localparam [0:7] read\_cmd = 8'h03;

reg [4:0] clk\_count;

// create an inverted clock signal to drive SCLK

wire inv\_clk\_in = ~clk\_in;

// state assignment

reg [2:0] state = RESET;

always@(posedge clk\_in) begin

clk\_count <= 5'd0;

if (reset) begin

state <= RESET;

end

else begin

case (state)

RESET : begin

state <= IDLE;

end

IDLE : begin

if (start\_read) begin

state <= ISSUE\_CMD;

clk\_count <= clk\_count;

end

else begin

state <= IDLE;

end

end

ENABLE\_CHIP : begin

state <= ISSUE\_CMD;

end

ISSUE\_CMD : begin

if (clk\_count < 5'd7) begin

state <= ISSUE\_CMD;

clk\_count <= clk\_count + 1'b1;

end

else begin

state <= ISSUE\_ADDR;

end

end

ISSUE\_ADDR : begin

if (clk\_count < 5'd23) begin

state <= ISSUE\_ADDR;

clk\_count <= clk\_count + 1'b1;

end

else begin

state <= RECEIVE\_DATA;

end

end

RECEIVE\_DATA : begin

if (clk\_count < 5'd7) begin

state <= RECEIVE\_DATA;

clk\_count <= clk\_count + 1'b1;

end

else begin

state <= DISABLE\_CHIP;

clk\_count <= 5'd0;

end

end

DISABLE\_CHIP : begin

state <= MOVE\_DATA;

end

MOVE\_DATA : begin

if (clk\_count < 5'd7) begin

state <= MOVE\_DATA;

clk\_count <= clk\_count + 1'b1;

end

else begin

state <= IDLE;

clk\_count <= 5'd0;

end

end

endcase

end

end

always @(negedge inv\_clk\_in) begin

case (state)

ISSUE\_ADDR : spi\_data <= 8'b0;

RECEIVE\_DATA : spi\_data <= { spi\_data[6:0], miso };

DISABLE\_CHIP : spi\_data <= { spi\_data[6:0], miso };

default : spi\_data <= spi\_data;

endcase

end

always@(\*) begin

case (state)

ISSUE\_CMD : mosi = read\_cmd[clk\_count];

ISSUE\_ADDR : mosi = read\_addr[clk\_count];

default : mosi = 1'b0;

endcase

end

always@(\*) begin

case (state)

RESET : ce\_n = 1'b1;

ENABLE\_CHIP : ce\_n = 1'b0;

ISSUE\_CMD : ce\_n = 1'b0;

ISSUE\_ADDR : ce\_n = 1'b0;

RECEIVE\_DATA : ce\_n = 1'b0;

DISABLE\_CHIP : ce\_n = 1'b0;

MOVE\_DATA : ce\_n = 1'b1;

default : ce\_n = 1'b1;

endcase

end

always@(\*) begin

case (state)

ISSUE\_CMD : sclk = inv\_clk\_in;

ISSUE\_ADDR : sclk = inv\_clk\_in;

RECEIVE\_DATA : sclk = inv\_clk\_in;

default : sclk = 1'b0;

endcase

end

always@(posedge clk\_in) begin

case (state)

RESET : data\_ready <= 1'b0;

ISSUE\_CMD : data\_ready <= 1'b0;

DISABLE\_CHIP : data\_ready <= 1'b1;

default : data\_ready <= data\_ready;

endcase

end

endmodule

module I2sInterface

( input clk,

input reset,

input lrclk,

input sclk,

input [10:0] sample\_data,

output reg data\_out);

reg a, b;

wire lr\_trig = a & ~b;

always@(posedge clk) begin

a <= lrclk;

b <= a;

end

reg c, d;

wire s\_trig = c & ~d;

always@(posedge clk) begin

c <= ~sclk;

d <= c;

end

reg [3:0] shift\_count;

parameter RESET = 2'd0;

parameter SYNC = 2'd1;

parameter SHIFT\_RIGHT = 2'd2;

parameter SHIFT\_LEFT = 2'd3;

reg [1:0] state;

always@(posedge clk) begin

if (reset) begin

state <= RESET;

shift\_count <= 4'd0;

end

else begin

case (state)

RESET : begin

state <= SYNC;

shift\_count <= 4'd0;

end

// wait for an LR transition becore starting the shift count

SYNC : begin

if (lr\_trig) begin

state <= SHIFT\_RIGHT;

if (s\_trig)

shift\_count <= 4'd15;

else

shift\_count <= shift\_count;

end

else begin

state <= SYNC;

shift\_count <= 4'd0;

end

end

SHIFT\_RIGHT : begin

if (shift\_count == 4'd0) begin

if (s\_trig) begin

shift\_count <= 4'd15;

state <= SHIFT\_LEFT;

end

else begin

shift\_count <= shift\_count;

state <= SHIFT\_RIGHT;

end

end

else begin

state <= SHIFT\_RIGHT;

if (s\_trig)

shift\_count <= shift\_count - 1'b1;

else

shift\_count <= shift\_count;

end

end

SHIFT\_LEFT : begin

if (shift\_count == 4'd0) begin

if (s\_trig) begin

shift\_count <= 4'd15;

state <= SHIFT\_RIGHT;

end

else begin

shift\_count <= shift\_count;

state <= SHIFT\_LEFT;

end

end

else begin

state <= SHIFT\_LEFT;

if (s\_trig)

shift\_count <= shift\_count - 1'b1;

else

shift\_count <= shift\_count;

end

end

endcase

end

end

// shift the sample data out the the I2S serial output, MSB first

// fill the lowest 5 bits with zeros to boost the sample to full 16-bit scale

always@(posedge clk) begin

if (s\_trig) begin

if (shift\_count > 3'd4) begin

data\_out <= sample\_data[shift\_count - 3'd5];

end

else begin

data\_out <= 1'b0;

end

end

else begin

data\_out <= data\_out;

end

end

endmodule

## SNES Interface

module SnesInterface

(

input sys\_clk, // 12.5MHz Clock

input sys\_reset,

input [1:0] address,

input read\_enable, // Reads the latched data for the controller specified by the read address

output reg [11:0] read\_data, // The button data for the specified controller

input [1:0] snes\_data, // Data returned from the SNES controller shift registers

output reg snes\_latch, // Pulse sent to the SNES controller to latch the state of the buttons

output reg snes\_pulse // Pulses sent to shift out the button values from the SNES controller

);

// count used for dividing the 12.5MHz system clock

reg [1:0] count;

// keep track of how many button values have been shifted in

reg [3:0] button\_count;

// registers to hold the data shifted in from each controller

reg [11:0] buttons\_0;

reg [11:0] buttons\_1;

// put the selected controller's button values on the output

always@(posedge sys\_clk) begin

if (sys\_reset) begin

read\_data <= 12'd0;

end

else if (read\_enable) begin

case (address)

0 : read\_data <= buttons\_0;

1 : read\_data <= buttons\_1;

default : read\_data <= read\_data;

endcase

end

else begin

read\_data <= read\_data;

end

end

// FSM States

parameter RESET = 3'd0;

parameter IDLE = 3'd1;

parameter LATCH = 3'd2;

parameter WAIT1 = 3'd3;

parameter SHIFT\_HI = 3'd4;

parameter SHIFT\_LO = 3'd5;

// FSM

reg [2:0] state;

always@(posedge sys\_clk) begin

snes\_latch <= 1'b0;

snes\_pulse <= 1'b0;

count <= 2'd0;

button\_count <= button\_count;

if (sys\_reset) begin

state <= RESET;

count <= 2'd0;

button\_count <= 1'b0;

end

else begin

case (state)

RESET : begin

state <= IDLE;

button\_count <= 1'b0;

end

IDLE : begin

if (read\_enable & address[1]) begin

state <= LATCH;

snes\_latch <= 1'b1;

end

else begin

state <= IDLE;

end

end

// hold the latch signal high for more than 200ns

LATCH : begin

if (count < 2'd3) begin

state <= LATCH;

snes\_latch <= 1'b1;

count <= count + 1'd1;

end

else begin

state <= WAIT1;

count <= 2'd0;

end

end

// wait for 1/2 period of the snes\_pulse clock

WAIT1 : begin

if (count < 2'd1) begin

state <= WAIT1;

count <= 2'd1;

end

else begin

state <= SHIFT\_HI;

snes\_pulse <= 1'b1;

button\_count <= 4'd1;

count <= 2'd0;

end

end

SHIFT\_HI : begin

if (count < 2'd1) begin

state <= SHIFT\_HI;

snes\_pulse <= 1'b1;

count <= 2'd1;

end

else begin

state <= SHIFT\_LO;

snes\_pulse <= 1'b0;

count <= 2'd0;

end

end

SHIFT\_LO : begin

if (count < 2'd1) begin

state <= SHIFT\_LO;

count <= 2'd1;

end

else begin

count <= 2'd0;

if (button\_count < 4'd11) begin

state <= SHIFT\_HI;

snes\_pulse <= 1'b1;

button\_count <= button\_count + 1'd1;

end

else begin

state <= IDLE;

button\_count <= 4'd0;

end

end

end

default : begin

state <= IDLE;

end

endcase

end

end

wire data\_latch;

assign data\_latch = snes\_latch | snes\_pulse;

// latch the button data on the falling edge of the latch and pulse signals

always @(negedge data\_latch) begin

if (state == LATCH | state == WAIT1 | state == SHIFT\_HI | state == SHIFT\_LO) begin

buttons\_0 <= { buttons\_0[10:0], ~snes\_data[0] };

buttons\_1 <= { buttons\_1[10:0], ~snes\_data[1] };

end

else begin

buttons\_0 <= buttons\_0;

buttons\_1 <= buttons\_1;

end

end

endmodule

## Timer

module Timer\_100us

(

input clk,

input reset,

input read,

output reg [15:0] count\_out

);

// 1250 cycles is 100us @ 12.5MHz

localparam TICKCOUNT = 1249;

// system reset is active low

wire inv\_reset = ~reset;

// increments every 100us, wraparound behavior

reg [15:0] count = 0;

// counts down the number of clock cycles until the next count increment

reg [10:0] ticks = 0;

// latch the count data to hold it stable during reads

always @(posedge clk) begin

if (read)

count\_out <= count;

else

count\_out <= count\_out;

end

// counter block

always @(posedge clk) begin

if (reset) begin

ticks <= TICKCOUNT;

count <= 0;

end

else if (ticks == 11'd0) begin

ticks <= TICKCOUNT;

count <= count + 1'b1;

end

else begin

ticks <= ticks - 1'b1;

count <= count;

end

end

endmodule

## CPU

‘include “defines.v”

module cpu

(input clk,// System clock

input rst,// Active low reset

input [`DATAWIDTH-1:0] EXT\_MEM\_DATA,// data from external memory

input [`DATAWIDTH-1:0] glyph\_addr,// Address for accessing glyph in ROM

input [`DATAWIDTH-1:0] dDst,// Destination register data

input [`DATAWIDTH-1:0] dSrc,// Source register data

input [11:0] din\_ctrlrs,// Data in from SNES Controllers

input [`DATAWIDTH-1:0] din\_timer,// Data in from Timmer

output ps,// Present Sate bit

output CE,// Chip enable for SRAM chip

output OE,// Output enable for SRAM chip

output WE,// Write enable for SRAM chip

output ROM\_CE,// Chip enable for ROM chip

output ROM\_OE,// Output enable for ROM chip

output write,// Control signal for write back to the reg file

output ctrlr\_re,// Control signal for reading from SNES controller

output audio\_we,// Write signal for audio peripherial

output timer\_re,// Reading from timer peripherial

output [1:0] addr\_ctrlr,// Address for SNES controller

output [1:0] addr\_audio,// Address for Audio register

output [2:0] acnt,// Arbiter count

output [`PRSWIDTH-1:0] psr,// Program Status Register

output [`REGWIDTH-1:0] rDst, rSrc,// Registers in regfile

output [`DATAWIDTH-1:0] wb\_data,// Data to be written back to register file

output [`DATAWIDTH-1:0] DOUT\_SRAM,// Output data to the SRAM

output [`DATAWIDTH-1:0] EXT\_MEM\_ADDR,// Address for external memory

output [`DATAWIDTH-1:0] dmem);// Output data from memory controller

wire [1:0] wb\_mux;// Mux that controls what is written back to regfile

wire [`IMMWIDTH-1:0] imm;// Immediate value

wire [`ALUOPWIDTH-1:0] alu\_op;// ALU operation code

wire [`DATAWIDTH-1:0] pc\_ra;// Return address

wire [`DATAWIDTH-1:0] pc;// Program counter

wire [`DATAWIDTH-1:0] mem\_addr;// Output of mux determining address from CPU or VGA

wire [`DATAWIDTH-1:0] rom\_addr;// Output of mux determining address from PC or VGA

// Module instances

controller LogicCtrl (.clk(clk), .rst(rst), .psr\_in(psr), .instruction(dmem), .acnt(acnt), .ps(ps), .BRANCH(branch), .JUMP(jump),

.ROM\_MUX(rom\_mux), .MEMC\_MUX(memc\_mux), .IMM\_MUX(imm\_mux), .PC\_EN(pc\_en), .WRITE(write),

.SRAM\_CE(SRAM\_CE), .SRAM\_OE(SRAM\_OE), .SRAM\_WE(SRAM\_WE), .ROM\_CE(ROM\_CE), .ROM\_OE(ROM\_OE),

.COND\_RSLT(cond\_rslt), .WB\_MUX0(WB\_MUX0), .WB\_MUX(wb\_mux), .rDst(rDst), .rSrc(rSrc),

.imm\_val(imm), .ALU\_OP(alu\_op));

pc PrgmCount (.clk(clk), .rst(rst), .branch(branch), .jump(jump), .pcEn(pc\_en), .disp(imm), .dSrc(dSrc),

.pc\_ra(pc\_ra), .pc(pc));

reg\_alu Execute (.IMM\_MUX(imm\_mux), .COND\_RSLT(cond\_rslt), .WB\_MUX0(WB\_MUX0), .WB\_MUX(wb\_mux),

.aluOp(alu\_op), .drom(dmem), .pc\_ra(pc\_ra), .imm\_in(imm), .mem\_data(dmem),

.dSrc(dSrc), .dDst(dDst), .wb\_data(wb\_data), .psrOut(psr));

mem\_ctrl MEM\_CTRL(.SRAM\_CE(SRAM\_CE), .SRAM\_OE(SRAM\_OE), .SRAM\_WE(SRAM\_WE), .ROM\_CE(ROM\_CE), .ROM\_OE(ROM\_OE), .din\_cpu(dDst),

.din\_ctrlrs(din\_ctrlrs), .din\_timer(din\_timer), .addrin\_cpu(mem\_addr), .EXT\_MEM\_DATA(EXT\_MEM\_DATA), .rom\_addr(rom\_addr), .CE(CE),

.OE(OE), .WE(WE), .ctrlr\_re(ctrlr\_re), .audio\_we(audio\_we), .timer\_re(timer\_re),

.addr\_ctrlr(addr\_ctrlr), .addr\_audio(addr\_audio), .DOUT\_SRAM(DOUT\_SRAM), .EXT\_MEM\_ADDR(EXT\_MEM\_ADDR), .dmem(dmem));

arbiter Arbiter(.clk(clk), .rst(rst), .count(acnt));

assign rom\_addr = (rom\_mux) ? mem\_addr : pc;

// Memory Controller mux

assign mem\_addr = (memc\_mux) ? dSrc : glyph\_addr;

endmodule

/// Arbiter to schedule memory access between CPU and VGA

module arbiter

(input clk, rst,

output reg [2:0] count);

always@(posedge clk) begin

if(!rst)

count <= 0;

else // Count will roll over every 8 cycles.

count <= count + 1;

end

endmodule

## System Top

`include "defines.v"

module sys\_top

(input clk, rst,

output [`DATAWIDTH-1:0] dmem\_out,

output [`DATAWIDTH-1:0] rom\_dout

);

// Internal buses

wire [`DATAWIDTH-1:0] sram\_dout;// Data from SRAM feeds into both CPU and VGA controller

wire [`DATAWIDTH-1:0] rom\_dout;// Data from ROM feeds into both CPU and VGA controller

wire [`DATAWIDTH-1:0] glyph\_addr;// Address from VGA to ROM

wire [`DATAWIDTH-1:0] alu\_result;// Result of ALU to feed into Audio Controller

wire [`DATAWIDTH-1:0] din\_cpu;// Data that feeds into Memory Controller

wire [`DATAWIDTH-1:0] rom\_addr;// Output address to ROM chip

wire [`DATAWIDTH-1:0] mem\_addr;// Address that feeds into Memory Controller

wire [`DATAWIDTH-1:0] dmem\_out;// Data out from the memory controller back into cpu/vga

wire [`DATAWIDTH-1:0] dsram;// Data from memory controller as data\_in for SRAM

wire [`DATAWIDTH-1:0] addr\_sram;// Address from memcory controller to addr for SRAM

wire [1:0] addr\_ctrlr;// Address from mem controller to controller logic

wire [1:0] addr\_audio;// Address from mem controller to audio logic

wire [`REGWIDTH-1:0] rDst, rSrc;// Destination and Source register encodings

wire [`DATAWIDTH-1:0] dDst, dSrc;// Destination and Source register data

wire [`DATAWIDTH-1:0] wb\_data;// Data to be written back to register file

cpu CPU(.clk(clk), .rst(rst), .sram\_dout(dmem\_out), .rom\_dout(rom\_dout), .glyph\_addr(16'd0), .dDst(dDst), .dSrc(dSrc),.SRAM\_CE(SRAM\_CE), .SRAM\_OE(SRAM\_OE), .SRAM\_WE(SRAM\_WE), .ROM\_CE(ROM\_CE), .ROM\_OE(ROM\_OE),.write(write), .acnt(acnt), .rDst(rDst), .rSrc(rSrc), .wb\_data(wb\_data), .memc\_din0(din\_cpu), .rom\_addr(rom\_addr), .mem\_addr(mem\_addr));

sram SRAM(.clk(clk), .CE(CE), .OE(OE), .WE(WE), .LB(1'b0), .UB(1'b0), .addr(addr\_sram), .din(dsram), .dout(sram\_dout));

rom ROM(.clk(clk), .CE(ROM\_CE), .OE(ROM\_OE), .addr(rom\_addr), .dout(rom\_dout));

regfile RegFile(.clk(clk), .write(write), .rSrc(rSrc), .rDst(rDst), .write\_data(wb\_data), .dSrc(dSrc), .dDst(dDst));

endmodule

## 41 to 16 Multiplexer

`include "defines.v"

module mux41x16

(input [1:0] cntrl,

input [`DATAWIDTH-1:0] arg0,

input [`DATAWIDTH-1:0] arg1,

input [`DATAWIDTH-1:0] arg2,

input [`DATAWIDTH-1:0] arg3,

output reg [`DATAWIDTH-1:0] dout

);

always@(\*) begin

case(cntrl)

2'b00: dout = arg0;

2'b01: dout = arg1;

2'b10: dout = arg2;

2'b11: dout = arg3;

default: dout = 16'd0;

endcase

end

endmodule

## 31 to 16 Multiplexer

`include "defines.v"

module mux31x16

(input [1:0] cntrl,

input [`DATAWIDTH-1:0] arg0,

input [`DATAWIDTH-1:0] arg1,

input [`DATAWIDTH-1:0] arg2,

input [`DATAWIDTH-1:0] arg3,

output reg [`DATAWIDTH-1:0] dout

);

always@(\*) begin

case(cntrl)

2'b00: dout = arg0;

2'b01: dout = arg1;

2'b10: dout = arg2;

2'b11: dout = arg3;

default: dout = 16'd0;

endcase

end

endmodule

## Tristate Buffer

module tribuf

(input din,

input en,

output dout);

assign dout = en ? din : 1'bz;

endmodule

## 16-Bit Tristate Buffer

`include "defines.v"

module tribuf16

(input en,

input [`DATAWIDTH-1:0] din,

output [`DATAWIDTH-1:0] dout);

tribuf \_tb0 (.din(din[0]), .en(en), .dout(dout[0]));

tribuf \_tb1 (.din(din[1]), .en(en), .dout(dout[1]));

tribuf \_tb2 (.din(din[2]), .en(en), .dout(dout[2]));

tribuf \_tb3 (.din(din[3]), .en(en), .dout(dout[3]));

tribuf \_tb4 (.din(din[4]), .en(en), .dout(dout[4]));

tribuf \_tb5 (.din(din[5]), .en(en), .dout(dout[5]));

tribuf \_tb6 (.din(din[6]), .en(en), .dout(dout[6]));

tribuf \_tb7 (.din(din[7]), .en(en), .dout(dout[7]));

tribuf \_tb8 (.din(din[8]), .en(en), .dout(dout[8]));

tribuf \_tb9 (.din(din[9]), .en(en), .dout(dout[9]));

tribuf \_tb10(.din(din[10]), .en(en), .dout(dout[10]));

tribuf \_tb11(.din(din[11]), .en(en), .dout(dout[11]));

tribuf \_tb12(.din(din[12]), .en(en), .dout(dout[12]));

tribuf \_tb13(.din(din[13]), .en(en), .dout(dout[13]));

tribuf \_tb14(.din(din[14]), .en(en), .dout(dout[14]));

tribuf \_tb15(.din(din[15]), .en(en), .dout(dout[15]));

endmodule