**San Jose State University**

**Department of Computer Engineering**

**CMPE 200 Report**

**Assignment-7 Report**

**Title** Pipelined MIPS Processor, I/O and Interfacing

**Semester:** Fall 2023 **Date:** 12/2/2023

**by**

**Name:** Sharath Chandra Mummadi **SID** 017400019

**Name:** Jincy Jose Thottathil **SID:** 016903978

**Name:** Sai Kashyap Kurella **SID:** 016018925

**INTRODUCTION**

The purpose of this lab is to convert the fifteen-instruction 32-bit single-cycle MIPS processor into a five-stage, pipelined design as well as Interface the processor with a factorial accelerator and a general-purpose I/O unit using memory-mapped interface registers.

**DESIGN METHODOLOGY**

**Single Cycle System on Chip**

This assignment is divided into two major parts, a single cycle system on a chip (SoC) design, and a pipelined MIPS processor working with the SoC. The single cycle MIPS processor completed in previous assignment was included and the factorial accelerator was modified such that it allowed each multiplication and countdown to take a single clock cycle. The GPIO was added in order to easily interface with factorial unit and MIPS processor. The address coming from single cycle MIPS is pulled directly off the output from instruction memory and is decoded in system level decoder and the proper addresses are sent to GPIO and Factorial units.

A diagram of a circuit

Description automatically generated

*Figure 1: Single Cycle MIPS microarchitecture*

A diagram of a computer

Description automatically generated

# *Figure 2: SoC Interfacing with Single cycle processor*

|  |  |  |  |
| --- | --- | --- | --- |
| **Mips Control** | | | |
| **Base Address** | **Address Range** | **R/W** | **Description** |
| 0x00000000 | 0x00 - 0xFC | R/W | Data Memory |
| 0x00000800 | 0x00 - 0x0C | R/W | Factorial Accelerator |
| 0x00000900 | 0x00 - 0x0C | R/W | General Purpose I/O |

Table 1. MIPS Control

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Factorial Accelerator Memory Map** | | | | | | | |
|  | | | | | **Decoder Output** | | |
| **Address** | **R/W** | **Register Name** | **Bits** | **Bit Defination** | **WE1** | **WE2** | **RdSel** |
| 00 | R/W | Data Input (n) | 31:4 | Unused |  |  |  |
| 3:0 | n[3:0] | 1 | 0 | 00 |
| 01 | R/W | Control Input (n) | 31:1 | Unused |  |  |  |
| 0 | Go Bit | 0 | 1 | 01 |
| 10 | R | Control Input (Done, Err) | 31:2 | Unused |  |  |  |
| 1 | Err bit | 0 | 0 | 10 |
| 11 | R | Data Output (Result) | 0 | Done bit | 0 | 0 | 10 |
| 31:0 | ans[31:0] | 0 | 0 | 11 |

Table 2. Factorial Accelerator Memory Map

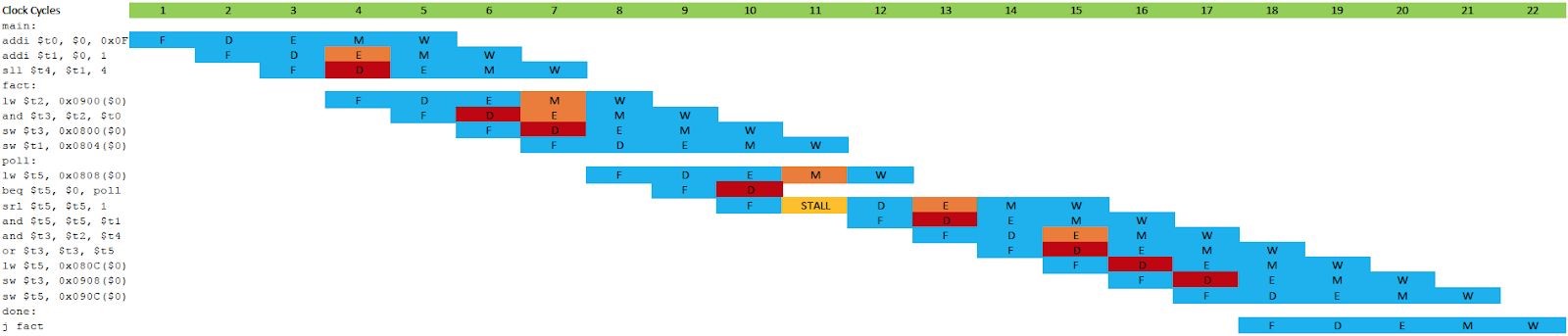
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **GPIO Memory Map** | | | | | | |
|  | | | | **Decoder Output** | | |
| **Address** | **R/W** | **Register Name** | **Bits** | **Bit Defination** | **WE2** | **RdSet** |
| 00 | R | Input1 | 31:0 | General Input | 0 | 00 |
| 01 | R | Input2 | 31:0 | General Input | 0 | 01 |
| 10 | R/W | Output1 | 31:0 | General output | 0 | 10 |
| 11 | R/W | Output2 | 31:0 | General output | 1 | 11 |

Table 3. GPIO Memory Map

**Pipelined MIPS Processor**

MIPS processor was modified, and 4 registers were added to the processor that resulted in 5 stages: Fetch, Decode, Execute, Memory and Writeback. With this addition of registers came a set of data and signal hazards that needed to be addressed.

The following diagram shows where the data hazards occur. It is evident that at least one stall cycle is required depending on the instruction. Nops were used to resolve this problem.



*Figure 3: Pipeline diagram of a standard pipelined MIPS processor*

A blueprint of a machine

Description automatically generated

*Figure 4: Pipelined MIPS microarchitecture*

A diagram of a computer hardware

Description automatically generated

# *Figure 5: SoC Interfacing with Pipelined MIPS processor*

A computer screen shot of a computer

Description automatically generated

# *Figure 6: Pipelined MIPS processor schematic with control unit, Hazard unit and data path*

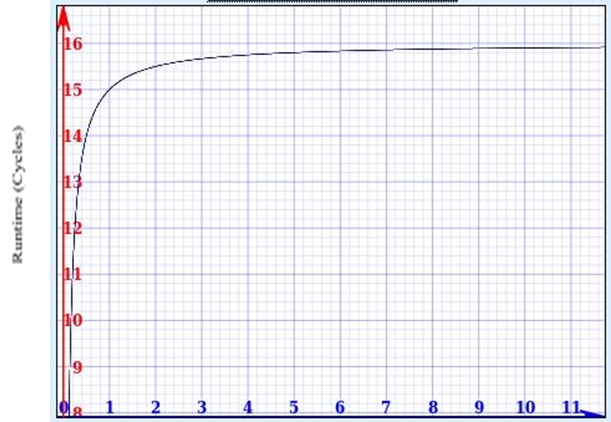
**Performance Analysis of hardware accelerated n!**

The factorial program was compared to analyze the performance gains for the SoC and pipelined MIPS SOC. The following equations show the results along with graphs shown in figure. It was found that the hardware accelerated version was close to 14 times more efficient that the software version of the program.

Single Cycle MIPS:

Hardware accelerated:

Performance gain single cycle:

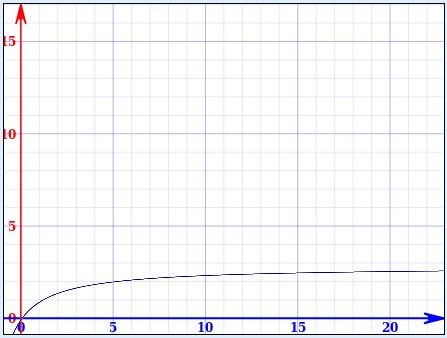


Factorial Input

*Figure 7: Single Cycle MIPS Processor Performance*

Pipelined MIPS:

Performance gained after pipelining:

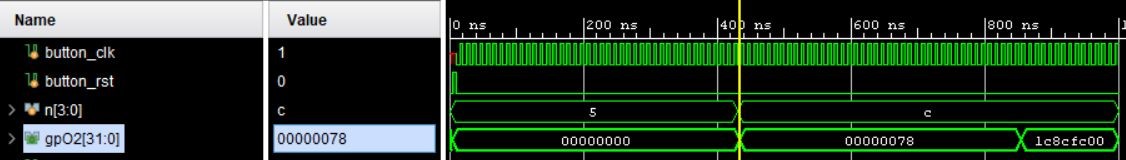


Factorial Input

*Figure 7: Pipelined MIPS Processor Performance*

**TEST RESULTS**

The program resides in memory memfile. When the clock is triggered, the instructions are executed accordingly. Below figures show the results of the testbench waveforms for Single Cycle processor SoC and Pipelined Processor SoC for 5! Respectively. All the simulation results were as expected for both Single Cycle on Chip and the Pipelined Processor.



*Figure 8: Simulations waveforms of Single Cycle MIPS processor running 5! program*

A screenshot of a computer

Description automatically generated

*Figure 8: Simulations waveforms of Pipelined MIPS processor running 5! program*

**CONCLUSION**

The SoC interface design was implemented with full integration of the pipelined MIPS processor, factorial unit and the GPIO module. This assignment gave a deeper insight into the process of implementing SoC design and validating it with a unit level testbench.

Below task were successfully accomplished:

* Drafted of pipelined MIPS microarchitecture schematic.
* Drafted of SoC interface schematic.
* Tables for MIPS control unit and memory maps for SoC interface.
* Performance analysis of hardware accelerated n!
* Unit-level (with interface wrappers) simulation
* Completed interface design for the SoC with the pipelined MIPS processor, the factorial unit and the simple GPIO.

**COLLABORATION:**

Collaborated with my teammate to draft MIPS microarchitecture schematic, Draft of SoC interface schematic, analysis of hardware accelerated n! and interface design for the SoC with the pipelined MIPS processor.

APPENDIX

Adder:

module adder (

input wire [31:0] a,

input wire [31:0] b,

output wire [31:0] y

);

assign y = a + b;

endmodule

ALU:

module alu (

input wire [2:0] op,

input wire [4:0] shamt,

input wire [31:0] a,

input wire [31:0] b,

output wire zero,

output reg [31:0] y

);

assign zero = (y == 0);

always @ (op, a, b) begin

case (op)

3'b000: y = a & b;

3'b001: y = a | b;

3'b010: y = a + b;

3'b110: y = a - b;

3'b111: y = (a < b) ? 1 : 0;

3'b100: y = b << shamt;

3'b101: y = b >> shamt;

endcase

end

endmodule

auxdec:

module auxdec (

input wire [1:0] alu\_op,

input wire [5:0] funct,

output wire we\_reg\_lh,

output wire sel\_reg\_lh,

output wire sel\_lh,

output wire [2:0] alu\_ctrl

);

reg [5:0] ctrl;

assign {alu\_ctrl,we\_reg\_lh,sel\_reg\_lh,sel\_lh} = ctrl;

always @ (alu\_op, funct) begin

case (alu\_op)

2'b00: ctrl = 6'b010000; // ADD

2'b01: ctrl = 6'b110000; // SUB

default: case (funct)

6'b10\_0100: ctrl = 6'b000000; // AND 0

6'b10\_0101: ctrl = 6'b001000; // OR 1

6'b10\_0000: ctrl = 6'b010000; // ADD 2

6'b10\_0010: ctrl = 6'b110000; // SUB 6

6'b10\_1010: ctrl = 6'b111000; // SLT 7

6'b00\_1000: ctrl = 6'b011000; // JR 3

6'b00\_0000: ctrl = 6'b100000; // SLL 4

6'b00\_0010: ctrl = 6'b101000; // SRL 5

6'b01\_1001: ctrl = 6'b000100; // MULTU

6'b01\_0000: ctrl = 6'b000011; // MFHI

6'b01\_0010: ctrl = 6'b000010; // MFLO

default: ctrl = 6'bxxxxxx;

endcase

endcase

end

endmodule

dmem:

module dmem (

input wire clk,

input wire we,

input wire [5:0] a,

input wire [31:0] d,

output wire [31:0] q

);

reg [31:0] ram [0:63];

integer n;

initial begin

for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;

end

always @ (posedge clk) begin

if (we) ram[a] <= d;

end

assign q = ram[a];

endmodule

signext:

module signext (

input wire [15:0] a,

output wire [31:0] y

);

assign y = {{16{a[15]}}, a};

endmodule

regfile:

module regfile (

input wire clk,

input wire we,

input wire we\_pc,

input wire [4:0] ra1,

input wire [4:0] ra2,

input wire [4:0] ra3,

input wire [4:0] wa,

input wire [31:0] pc,

input wire [31:0] wd,

output wire [31:0] rd1,

output wire [31:0] rd2,

output wire [31:0] rd3

);

reg [31:0] rf [0:31];

integer n;

initial begin

for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;

rf[29] = 32'h100; // Initialze $sp

end

always @ (posedge clk) begin

if(we\_pc) rf[31] <= pc;

else if (we) rf[wa] <= wd;

end

assign rd1 = (ra1 == 0) ? 0 : rf[ra1];

assign rd2 = (ra2 == 0) ? 0 : rf[ra2];

assign rd3 = (ra3 == 0) ? 0 : rf[ra3];

endmodule

mux4:

module mux4 #(parameter WIDTH = 8) (

input wire [1:0] sel,

input wire [WIDTH-1:0] a,

input wire [WIDTH-1:0] b,

input wire [WIDTH-1:0] c,

input wire [WIDTH-1:0] d,

output wire [WIDTH-1:0] y

);

wire [WIDTH-1:0] cd;

wire [WIDTH-1:0] ab;

assign y = sel[1] ? (sel[0] ? d:c):( sel[0] ? b:a);

endmodule

mux2:

module mux2 #(parameter WIDTH = 8) (

input wire sel,

input wire [WIDTH-1:0] a,

input wire [WIDTH-1:0] b,

output wire [WIDTH-1:0] y

);

assign y = (sel) ? b : a;

endmodule

mult\_unit:

module mult\_unit(input wire clk,

input wire we,

input wire sel,

input wire [31:0] a,

input wire [31:0] b,

output wire [31:0]rd1);

reg [31:0] reglh[0:1];

reg [63:0] ab;

integer n;

initial begin

for (n = 0; n < 2; n = n + 1) reglh[n] = 32'h0;

end

always @ (posedge clk) begin

if (we)

begin

ab = a\*b;

reglh[0] <= ab[31:0];

reglh[1] <= ab[63:32];

end

end

assign rd1 = reglh[sel];

endmodule

MEM\_WB:

module MEM\_WB\_Reg (

input wire clk,

input wire [4:0] ra12m,

input wire [5:0] ctrl\_sigm,

input wire [31:0] alu\_outm,

input wire [31:0] rd\_dm,

input wire [31:0] mult\_dm,

output wire [4:0] ra12b,

output wire [31:0] alu\_outb,

output wire [31:0] rd\_d\_mb,

output wire [5:0] ctrl\_sigb,

output wire [31:0] mult\_dm\_b

);

reg [31:0] rf [0:2];

reg [4:0] bm;

reg [5:0] rc;

integer n;

initial begin

for (n = 0; n < 3; n = n + 1) rf[n] = 32'h0;

bm = 5'h0;

rc = 6'h0;

end

always @ (posedge clk) begin

rf[0] = alu\_outm;

rf[1] = rd\_dm;

rf[2] = mult\_dm;

bm = ra12m;

rc = ctrl\_sigm;

end

assign alu\_outb = rf[0];

assign rd\_d\_mb = rf[1];

assign mult\_dm\_b = rf[2];

assign ctrl\_sigb = rc;

assign ra12b = bm;

endmodule

maindec:

module maindec (

input wire [5:0] opcode,

output wire branch,

output wire jump,

output wire reg\_dst,

output wire we\_reg,

output wire alu\_src,

output wire we\_dm,

output wire dm2reg,

output wire [1:0] alu\_op

);

reg [8:0] ctrl;

assign {branch, jump, reg\_dst, we\_reg, alu\_src, we\_dm, dm2reg, alu\_op} = ctrl;

always @ (opcode) begin

case (opcode)

6'b00\_0000: ctrl = 9'b0\_0\_1\_1\_0\_0\_0\_10; // R-type

6'b00\_1000: ctrl = 9'b0\_0\_0\_1\_1\_0\_0\_00; // ADDI

6'b00\_0100: ctrl = 9'b1\_0\_0\_0\_0\_0\_0\_01; // BEQ

6'b00\_0010: ctrl = 9'b0\_1\_0\_0\_0\_0\_0\_00; // J

6'b00\_0011: ctrl = 9'b0\_1\_0\_1\_0\_0\_0\_00; //JAL

6'b10\_1011: ctrl = 9'b0\_0\_0\_0\_1\_1\_0\_00; // SW

6'b10\_0011: ctrl = 9'b0\_0\_0\_1\_1\_0\_1\_00; // LW

default: ctrl = 9'bx\_x\_x\_x\_x\_x\_x\_xx;

endcase

end

endmodule

imem:

module imem (

input wire [5:0] a,

output wire [31:0] y

);

reg [31:0] rom [0:63];

initial begin

$readmemh ("memfile.dat", rom);

end

assign y = rom[a];

endmodule

IF\_ID\_Reg:

module IF\_ID\_Reg (

input wire clk,

input wire [31:0] instrf,

input wire [31:0] pc\_4f,

input wire stallD,

input wire clr\_PCSRC,

output wire [31:0] instrD,

output wire [31:0] pc\_4D

);

reg [31:0] rf [0:1];

integer n;

initial begin

for (n = 0; n < 2; n = n + 1) rf[n] = 32'h0;

end

always @ (posedge clk) begin

if(!stallD)

begin

rf[0] = instrf;

rf[1] = pc\_4f;

end

if(clr\_PCSRC == 1 && stallD == 0)

for (n = 0; n < 2; n = n + 1) rf[n] = 32'h0;

end

assign pc\_4D = rf[1];

assign instrD = rf[0];

endmodule

ID\_EXE\_Reg:

module ID\_EXE\_Reg (

input wire clk,

input wire [12:0] ctrl\_sigd,

input wire [31:0] pc\_4d,

input wire [4:0] ras,

input wire [4:0] ra1d,

input wire [4:0] ra2d,

input wire [4:0] shamntd,

input wire [31:0] sext\_immd,

input wire [25:0] jump\_offset,

input wire [31:0] rd1d,

input wire [31:0] rd2d,

//

input wire flushE,

output wire [12:0] ctrl\_sige,

output wire [31:0] pc\_4e,

output wire [4:0] ra1e,

output wire [4:0] ra2e,

output wire [4:0] ras\_e,

output wire [4:0] shamnte,

output wire [31:0] rd1e,

output wire [31:0] rd2e,

output wire [31:0] sext\_imme,

output wire [25:0] jump\_offset\_e

);

reg [31:0] rf [0:3];

reg [4:0] rf4 [0:3];

reg [12:0] rfc;

reg [25:0] rj;

integer n;

task reset\_exec;

begin

for (n = 0; n < 4; n = n + 1) rf[n] = 32'h0;

for (n = 0; n < 4; n = n + 1) rf4[n] = 5'h0;

rfc = 13'h0;

rj = 26'h0;

end

endtask

initial begin

reset\_exec;

end

always @ (posedge clk) begin

rf[0] = pc\_4d; rf[1] = rd1d; rf[2] = rd2d; rf[3] = sext\_immd;

rf4[0] = ra1d; rf4[1] = ra2d; rf4[2] = shamntd; rf4[3] = ras;

rfc = ctrl\_sigd;

rj = jump\_offset;

if(flushE == 1)

begin

reset\_exec;

end

end

assign pc\_4e = rf[0];

assign rd1e = rf[1];

assign rd2e = rf[2];

assign sext\_imme = rf[3];

assign jump\_offset\_e = rj;

assign ra1e = rf4[0];

assign ra2e = rf4[1];

assign shamnte = rf4[2];

assign ras\_e = rf4[3];

assign ctrl\_sige = rfc;

endmodule

EXE\_MEM\_Reg:

module EXE\_MEM\_Reg (

input wire clk,

input wire zero,

input wire [6:0] ctrl\_sige,

//input wire [31:0] pc\_4e,

input wire [31:0] alu\_oute,

input wire [31:0] rd2e,

input wire [4:0] ra12e,

output wire zerom,

output wire [6:0] ctrl\_sigm,

//output wire [31:0] pc\_4m,

output wire [31:0] alu\_outm,

output wire [31:0] rd2m,

output wire [4:0] ra12m

);

reg [31:0] rf [0:2];

reg [6:0] rc;

reg [4:0] rr;

reg rz;

integer n;

initial begin

for (n = 0; n < 3; n = n + 1) rf[n] = 32'h0;

rc = 7'h0;

rr = 5'h0;

rz = 0;

end

always @ (posedge clk) begin

//rf[0] = pc\_4e;

rf[1] = alu\_oute;

rf[2] = rd2e;

rz = zero;

rc = ctrl\_sige;

rr = ra12e;

end

//assign pc\_4e = rf[0];

assign alu\_outm = rf[1];

assign rd2m = rf[2];

assign zerom = rz;

assign ctrl\_sigm = rc;

assign ra12m = rr;

endmodule

GPIO:

module GPIO(

input wire clk,

input wire we,

input wire [31:0] wd,

output wire [31:0] rd

);

reg [31:0] out;

reg [31:0] in;

initial

out = 32'h5;

always @ (posedge(clk))

if (we)

in = wd;

assign rd = out;

endmodule

dreg:

module dreg # (parameter WIDTH = 32) (

input wire clk,

input wire rst,

input wire stallF,

input wire [WIDTH-1:0] d,

output wire [WIDTH-1:0] q

);

reg [WIDTH-1:0] ireg;

always @ (posedge clk, posedge rst) begin

if (rst) ireg = 0;

else if(stallF == 0) ireg = d;

end

assign q = ireg;

endmodule

hazardunit:

module hazardunit (

input wire we\_reg\_b,

input wire [4:0] ra12b,

input wire we\_reg\_m,

input wire [4:0] ra12m,

input wire dm2reg\_e,

input wire [4:0] rs1e,

input wire [4:0] ra1e,

input wire [4:0] rs1d,

input wire [4:0] rt1d,

input wire jmp,

input wire we\_reg\_e,

input wire dm2reg\_m,

input wire [4:0] ra12e,

input wire branchD,

output wire stallF,

output wire stallD,

output wire flushE,

output wire forwardAD,

output wire forwardBD,

output wire [1:0] forwardAE,

output wire [1:0] forwardBE

);

reg lwstall;

reg branchStall;

reg [1:0] fa;

reg [1:0] fb;

reg fBd;

reg fAd;

initial begin

lwstall = 0;

fa = 2'b0;

fb = 2'b0;

fAd = 0;

fBd = 0;

branchStall = 0;

end

always @(\*) begin

lwstall = ((rs1d == ra1e || rt1d == ra1e) && dm2reg\_e) ? 1:0;

fAd = ((rs1d != 0) && rs1d == ra12m && we\_reg\_m);

fBd = ((rt1d != 0) && rt1d == ra12m && we\_reg\_m);

// branchstall =

//BranchD AND RegWriteE AND (WriteRegE ==rsD OR WriteRegE ==rtD)

//OR

//BranchD AND MemtoRegM AND (WriteRegM ==rsD OR WriteRegM ==rtD)

branchStall = (branchD && we\_reg\_e && (ra12e == rs1d || ra12e == rt1d)) ||

(branchD && dm2reg\_m && (ra12m == rs1d || ra12m == rt1d));

if ((rs1e != 0) && (rs1e == ra12m) && we\_reg\_m)

fa = 2'b10;

else if ((rs1e != 0) && (rs1e == ra12b) && we\_reg\_b)

fa = 2'b01;

else

fa = 2'b00;

if ((ra1e != 0) && (ra1e == ra12m) && we\_reg\_m)

fb = 2'b10;

else if ((ra1e != 0) && (ra1e == ra12b) && we\_reg\_b)

fb = 2'b01;

else

fb = 2'b00;

end

assign stallD = lwstall || branchStall;

assign stallF = lwstall || branchStall;

assign flushE = lwstall || branchStall;

assign forwardAE = fa;

assign forwardBE = fb;

assign forwardAD = fAd;

assign forwardBD = fBd;

endmodule

Datapath:

module datapath (

input wire clk,

input wire rst,

input wire branch,

input wire jump,

input wire reg\_dst,

input wire we\_reg,

input wire alu\_src,

input wire dm2reg,

input wire we\_reg\_lh,

input wire sel\_reg\_lh,

input wire sel\_lh,

input wire we\_dm,

input wire [2:0] alu\_ctrl,

input wire [4:0] ra3,

input wire [31:0] rd\_dm,

//

input wire stallF,

input wire stallD,

input wire flushE,

input wire [1:0] forwardAE,

input wire [1:0] forwardBE,

input wire forwardAD,

input wire forwardBD,

//

output wire dm2reg\_m,

output wire jmp,

output wire [31:0] instr\_out,

output wire [31:0] pc\_current,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd3,

//Hazard

output wire branchD,

output wire we\_reg\_e,

output wire [4:0] ra12e,

output wire we\_reg\_b,

output wire [4:0] ra12b,

output wire we\_reg\_m,

output wire [4:0] ra12m,

output wire dm2reg\_e,

output wire [4:0] rs1e,

output wire [4:0] ra1e,

output wire [4:0] rs1d,

output wire [4:0] rt1d

);

wire [4:0] rf\_wa;

wire [4:0] shamnt;

wire [31:0] pc\_plus4;

wire [31:0] pc\_pre;

wire [31:0] pc\_next;

wire [31:0] sext\_imm;

wire [31:0] ba;

wire [31:0] bta;

wire [31:0] jta;

wire [31:0] alu\_pa;

wire [31:0] alu\_pb;

wire [31:0] wd\_rf;

wire [31:0] pre\_pc\_next;

wire [31:0] multu\_lh;

wire [31:0] p\_wd\_rf;

wire [31:0] instr;

wire [31:0] pc\_4D;

wire zero;

wire pc\_src;

wire we\_pc;

wire we\_register;

wire jr\_jmp;

wire [12:0] ctrl\_sigd;

wire [12:0] ctrl\_sige;

wire [31:0] pc\_4e;

//wire [4:0] ra1e;

wire [4:0] ra2e;

wire [4:0] shamnte;

wire [31:0] rd1e;

wire [31:0] rd2e;

wire [31:0] sext\_imme;

wire [25:0] jump\_offset\_e;

wire clr\_PCSRC;

wire write\_enable\_data\_memory;

wire write\_enable\_dev;

wire we\_dev\_fact;

wire we\_dev\_gpio;

wire [31:0] fact\_out;

wire [31:0] gpio\_out;

wire [31:0] mux\_out\_rd\_dm;

wire zerom;

wire [6:0] ctrl\_sigm;

//wire [31:0] pc\_4m;

wire [31:0] alu\_outm;

wire [31:0] rd2m;

//wire [4:0] ra12m;

wire [5:0] ctrl\_sigb;

//wire [4:0] ra12b;

wire [31:0] alu\_outb;

wire [31:0] rd\_d\_mb;

///////

wire [31:0] mux\_out\_fa;

wire [31:0] mux\_out\_fb;

wire [31:0] multu\_lh\_mb;

//

wire branchEqualD;

wire [31:0] rd1\_alupmx;

wire [31:0] rd2\_alupmx;

assign branchEqualD = (rd1\_alupmx == rd2\_alupmx) ? 1:0;

assign jmp = jump;

//

//

assign write\_enable\_data\_memory = ctrl\_sigm[0] && ~alu\_outm[9];

assign write\_enable\_dev = ctrl\_sigm[0] && alu\_outm[9];

assign we\_dev\_fact = write\_enable\_dev && ~alu\_outm[8];

assign we\_dev\_gpio = write\_enable\_dev && alu\_outm[8];

//

assign clr\_PCSRC = pc\_src || jump || jr\_jmp;

//

assign we\_reg\_b = ctrl\_sigb[3];

assign we\_reg\_m = ctrl\_sigm[4];

assign we\_reg\_e = ctrl\_sige[9];

assign dm2reg\_e = ctrl\_sige[7];

assign dm2reg\_m = ctrl\_sigm[3]; //verify

assign ra12e = rf\_wa;

assign rs1d = instr\_out[25:21];

assign rt1d = instr\_out[20:16];

//

//assign pc\_src = ctrl\_sigm[5] & zerom;

assign pc\_src = branch & branchEqualD;

assign we\_pc = ctrl\_sige[11] & ctrl\_sige[9];

assign we\_register = ctrl\_sigb[3] && ~ctrl\_sigb[1];

assign ba = {sext\_imm[29:0], 2'b00};

assign jta = {pc\_4D[31:28], instr\_out[25:0], 2'b00};

//assign jr\_jmp = (ctrl\_sige[2:0] == 3'b011) ? 1 : 0;

assign jr\_jmp = (alu\_ctrl == 3'b011) ? 1 : 0;

assign shamnt = instr\_out[10:6];

assign branchD = branch;

assign ctrl\_sigd = {branch, jump, reg\_dst, we\_reg, alu\_src,

dm2reg, we\_reg\_lh,

sel\_reg\_lh, sel\_lh, we\_dm, alu\_ctrl};

//Intermediate Register

IF\_ID\_Reg IF\_ID\_Reg(

.clk (clk),

.instrf (instr),

.pc\_4f (pc\_plus4),

.clr\_PCSRC (clr\_PCSRC),

.stallD (stallD),

.instrD (instr\_out),

.pc\_4D (pc\_4D)

);

ID\_EXE\_Reg ID\_EXE\_Reg(

.clk (clk),

.ctrl\_sigd (ctrl\_sigd),

.pc\_4d (pc\_4D),

.ras (instr\_out[25:21]),

.ra1d (instr\_out[20:16]),

.ra2d (instr\_out[15:11]),

.rd1d (alu\_pa),

.rd2d (wd\_dm),

.sext\_immd (sext\_imm),

.shamntd (shamnt),

.jump\_offset (instr\_out[25:0]),

.flushE (flushE),

.ctrl\_sige (ctrl\_sige),

.pc\_4e (pc\_4e),

.ra1e (ra1e),

.ra2e (ra2e),

.shamnte (shamnte),

.rd1e (rd1e),

.rd2e (rd2e),

.sext\_imme (sext\_imme),

.jump\_offset\_e (jump\_offset\_e),

.ras\_e (rs1e)

);

EXE\_MEM\_Reg EXE\_MEM\_Reg(

.clk (clk),

.zero (zero),

.ctrl\_sige ({ctrl\_sige[4],ctrl\_sige[12],ctrl\_sige[9],ctrl\_sige[7],

we\_pc,ctrl\_sige[5],ctrl\_sige[3]}),

//.pc\_4e (bta),

.alu\_oute (alu\_out),

.rd2e (mux\_out\_fb),

.ra12e (rf\_wa),

.zerom (zerom),

.ctrl\_sigm (ctrl\_sigm),

//.pc\_4m (pc\_4m),

.alu\_outm (alu\_outm),

.rd2m (rd2m),

.ra12m (ra12m)

);

MEM\_WB\_Reg MEM\_WB\_Reg(

.clk (clk),

.ra12m (ra12m),

.ctrl\_sigm

({ctrl\_sigm[1],ctrl\_sigm[6],ctrl\_sigm[4],ctrl\_sigm[3],ctrl\_sigm[2],ctrl\_sigm[5]}),

.alu\_outm (alu\_outm),

.rd\_dm (mux\_out\_rd\_dm),

.mult\_dm (multu\_lh),

.ra12b (ra12b),

.alu\_outb (alu\_outb),

.rd\_d\_mb (rd\_d\_mb),

.ctrl\_sigb (ctrl\_sigb),

.mult\_dm\_b (multu\_lh\_mb)

);

// --- PC Logic --- //

dreg pc\_reg (

.clk (clk),

.rst (rst),

.stallF (stallF),

.d (pc\_next),

.q (pc\_current)

);

adder pc\_plus\_4 (

.a (pc\_current),

.b (32'd4),

.y (pc\_plus4)

);

adder pc\_plus\_br (

.a (pc\_4D),

.b (ba),

.y (bta)

);

mux2 #(32) pc\_src\_mux (

.sel (pc\_src),

.a (pc\_plus4),

.b (bta),

.y (pc\_pre)

);

mux2 #(32) pc\_jmp\_mux (

//.sel (ctrl\_sige[11]),

.sel (jump),

.a (pc\_pre),

.b (jta),

.y (pre\_pc\_next)

);

mux2 #(32) pc\_jr\_mux(

.sel (jr\_jmp),

.a (pre\_pc\_next),

.b (alu\_pa),

.y (pc\_next)

);

//MEM

imem imem (

.a (pc\_current[7:2]),

.y (instr)

);

dmem dmem (

.clk (clk),

.we (write\_enable\_data\_memory),

.a (alu\_outm[7:2]),

.d (rd2m),

.q (rd\_dm)

);

//-------------------

//GPIO and FACTORIAL

FactTop ftp(

.clk (clk),

.reset (0),

.we (we\_dev\_fact),

.wa (alu\_outm[1:0]),

.wd (rd2m[3:0]),

.rd (fact\_out)

);

GPIO gpio(

.clk (clk),

.we (we\_dev\_gpio),

.wd (rd2m),

.rd (gpio\_out)

);

mux4 #(32) mux\_fact\_gpio (

.sel (alu\_outm[9:8]),

.a (rd\_dm),

.b (rd\_dm),

.c (fact\_out),

.d (gpio\_out),

.y (mux\_out\_rd\_dm)

);

//------------------------

// --- RF Logic --- //

mux2 #(5) rf\_wa\_mux (

.sel (ctrl\_sige[10]),

.a (ra1e),

.b (ra2e),

.y (rf\_wa)

);

regfile rf (

.clk (clk),

.we (we\_register),

.we\_pc (we\_pc),

.ra1 (instr\_out[25:21]),

.ra2 (instr\_out[20:16]),

.ra3 (ra3),

.wa (ra12b),

.pc (pc\_4e),

.wd (wd\_rf),

.rd1 (alu\_pa),

.rd2 (wd\_dm),

.rd3 (rd3)

);

//Branch Hazard

//After Decode Stage

mux2 #(32) rd1\_palu(

.sel (forwardAD),

.a (alu\_pa),

.b (alu\_outm),

.y (rd1\_alupmx)

);

mux2 #(32) rd2\_palu(

.sel (forwardBD),

.a (wd\_dm),

.b (alu\_outm),

.y (rd2\_alupmx)

);

signext se (

.a (instr\_out[15:0]),

.y (sext\_imm)

);

// --- ALU Logic --- //

mux2 #(32) alu\_pb\_mux (

.sel (ctrl\_sige[8]),

.a (mux\_out\_fb),

.b (sext\_imme),

.y (alu\_pb)

);

//

mux4 #(32) mux\_fa (

.sel (forwardAE),

.a (rd1e),

.b (wd\_rf),

.c (alu\_outm),

.d (32'b0),

.y (mux\_out\_fa)

);

mux4 #(32) mux\_fb (

.sel (forwardBE),

.a (rd2e),

.b (wd\_rf),

.c (alu\_outm),

.d (32'b0),

.y (mux\_out\_fb)

);

//

alu alu (

.op (ctrl\_sige[2:0]),

.shamt (shamnte),

.a (mux\_out\_fa),

.b (alu\_pb),

.zero (zero),

.y (alu\_out)

);

// --- MULTIPLICATION --- //

mult\_unit mult\_unit(

.clk (clk),

.we (ctrl\_sige[6]),

.sel (ctrl\_sigb[4]),

.a (mux\_out\_fa),

.b (alu\_pb),

.rd1 (multu\_lh)

);

// --- MEM Logic --- //

mux2 #(32) rf\_wd\_mux (

.sel (ctrl\_sigb[2]),

.a (alu\_outb),

.b (rd\_d\_mb),

.y (p\_wd\_rf)

);

mux2 #(32) rf\_wd\_reg\_lh\_mux(

.sel (ctrl\_sigb[5]),

.a (p\_wd\_rf),

.b (multu\_lh),

.y (wd\_rf)

);

endmodule

Control unit:

module controlunit (

input wire [5:0] opcode,

input wire [5:0] funct,

output wire branch,

output wire jump,

output wire reg\_dst,

output wire we\_reg,

output wire alu\_src,

output wire we\_dm,

output wire dm2reg,

output wire we\_reg\_lh,

output wire sel\_reg\_lh,

output wire sel\_lh,

output wire [2:0] alu\_ctrl

);

wire [1:0] alu\_op;

maindec md (

.opcode (opcode),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.we\_dm (we\_dm),

.dm2reg (dm2reg),

.alu\_op (alu\_op)

);

auxdec ad (

.alu\_op (alu\_op),

.funct (funct),

.we\_reg\_lh (we\_reg\_lh),

.sel\_reg\_lh (sel\_reg\_lh),

.sel\_lh (sel\_lh),

.alu\_ctrl (alu\_ctrl)

);

endmodule

module CNT(input wire clk,ld\_cnt,en,

input wire [31:0] in,

output wire [31:0] \_q);

reg [31:0]q;

initial

q = 0;

assign \_q = q;

always @(posedge clk ) begin

if(ld\_cnt)

q = in;

else if(en)

q = q - 1;

end

endmodule

module MUX(input wire sel,

input wire [31:0] a,b,

output wire [31:0] c);

assign c = sel == 0?a:b;

endmodule

module MUL(input wire [31:0] a,b,

output wire[31:0] ab);

assign ab = a \* b;

endmodule

module CMP(input wire [31:0] a,b,

output wire gt);

assign gt = a>b?1:0;

endmodule

module REG(input wire clk,ld,

input wire [31:0] in,

output wire [31:0] \_out);

reg [31:0] out;

initial

out = 0;

assign \_out = out;

always @(posedge clk ) begin

if(ld)

out = in;

end

endmodule

module FDataPath(input wire clk,

input wire [4:0]control\_signals,

input wire[31:0] in,

output wire [1:0] status\_signals,

output wire [31:0] out);

wire [31:0] val\_0,val\_1,val\_12;

assign val\_0 = 0;

assign val\_1 = 1;

assign val\_12 = 12;

wire [31:0] out\_cnt1,out\_reg1,out\_mult,out\_mux1;

CNT cnt1(clk,control\_signals[0],control\_signals[1],in,out\_cnt1);

CMP cmp1(out\_cnt1,val\_1,status\_signals[1]);

CMP cmp2(out\_cnt1,val\_12,status\_signals[0]);

REG reg1(clk,control\_signals[3],out\_mux1,out\_reg1);

MUL mult(out\_reg1,out\_cnt1,out\_mult);

MUX mux1(control\_signals[4],val\_1,out\_mult,out\_mux1);

MUX mux\_out(control\_signals[2],val\_0,out\_reg1,out);

endmodule

FControlUnit:

module FControlUnit(input wire clk,go,reset,

input wire [1:0] status\_signals,

output reg err,done,

output reg [4:0] control\_signals);

reg [2:0] cs,ns;

initial

begin

cs = 0;

err = 0;

done = 0;

end

always @(posedge clk ) begin

cs <= ns;

if(reset == 1)

begin

cs =0;

err =0;

done = 0;

end

end

always @(cs,go,status\_signals) begin

case (cs)

3'b000:

begin

if(go)

ns = 1;

else

ns = 0;

control\_signals = 0;

done = 0;

err = 0;

end

3'b001:

begin

control\_signals = 5'b00001;

ns = 3'b010;

end

3'b010:

begin

if(status\_signals[0])

begin

control\_signals = 5'b00000;

ns = 3'b011;

end

else

begin

control\_signals = 5'b01000;

ns = 3'b100;

end

end

3'b011:

begin

control\_signals = 5'b00000;

ns = 3'b000;

done = 1;

err = 1;

end

3'b100:

begin

if(status\_signals[1])

begin

control\_signals = 5'b10000;

ns = 3'b101;

end

else

begin

control\_signals = 5'b00000;

ns = 3'b111;

end

end

3'b101:

begin

control\_signals = 5'b11000;

ns = 3'b110;

end

3'b110:

begin

control\_signals = 5'b10010;

ns = 3'b100;

end

3'b111:

begin

control\_signals = 5'b00100;

done = 1;

ns = 3'b000;

end

default:

begin

ns = 0;

cs = 0;

control\_signals = 0;

end

endcase

end

endmodule

module FactorialUnit(input wire clk,go,reset,

input wire [3:0] n,

output wire err,done,

output wire [31:0] fans);

wire [4:0] control\_signals;

wire [1:0] status\_signals;

FControlUnit cu1(clk,go,reset,status\_signals,err,done,control\_signals);

FDataPath dp1(clk,control\_signals,{28'b0,n},status\_signals,fans);

endmodule

FactTop:

module FactTop(

input wire clk,

input wire reset,

input wire we,

input wire [1:0]wa,

input wire [3:0]wd,

output wire [31:0] rd

);

reg err\_o;

reg done\_o;

wire er,dn;

wire [31:0] fans;

reg [31:0] ans;

reg [3:0] n;

reg go\_out,go;

reg [31:0] ird;

initial begin

err\_o = 0;

done\_o = 0;

go = 0;

go\_out = 0;

n = 0;

end

always @(posedge clk) begin

if(er)

err\_o = 1;

if(dn)

done\_o = 1;

if(we)

begin

go = we && wd[0];

if(wa == 2'b00)

begin

n = wd;

end

else if(wa == 2'b01)

begin

go\_out = wd[0];

err\_o = 0;

done\_o = 0;

end

end

if(dn)

ans = fans;

end

always @(\*)

begin

case(wa)

2'b00: ird = n;

2'b01: ird = {30'b0,err\_o,go};

2'b10: ird = {31'b0,done\_o};

2'b11: ird = ans;

default: ird = 32'bXX;

endcase

end

assign rd = ird;

FactorialUnit fu(

.clk (clk),

.go (go),

.reset (reset),

.n (n),

.err (er),

.done (dn),

.fans (fans)

);

endmodule

mips\_top:

module mips\_top (

input wire clk,

input wire rst,

input wire [4:0] ra3,

output wire we\_dm,

output wire [31:0] pc\_current,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd\_dm,

output wire [31:0] rd3

);

wire [31:0] DONT\_USE;

mips mips (

.clk (clk),

.rst (rst),

.ra3 (ra3),

.rd\_dm (rd\_dm),

.we\_dm (we\_dm),

.pc\_current (pc\_current),

.alu\_out (alu\_out),

.wd\_dm (wd\_dm),

.rd3 (rd3)

);

endmodule

MIPS:

module mips (

input wire clk,

input wire rst,

input wire [4:0] ra3,

input wire [31:0] rd\_dm,

output wire we\_dm,

output wire [31:0] pc\_current,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd3

);

wire branch;

wire jump;

wire reg\_dst;

wire we\_reg;

wire alu\_src;

wire dm2reg;

wire we\_reg\_lh;

wire sel\_reg\_lh;

wire sel\_lh;

wire jmp;

//

wire stallF;

wire stallD;

wire flushE;

wire [1:0] forwardAE;

wire [1:0] forwardBE;

// -----

wire we\_reg\_b;

wire [4:0] ra12b;

wire we\_reg\_m;

wire [4:0] ra12m;

wire dm2reg\_e;

wire [4:0] rs1e;

wire [4:0] ra1e;

wire [4:0] rs1d;

wire [4:0] rt1d;

wire we\_reg\_e;

wire dm2reg\_m;

wire [4:0] ra12e;

wire branchD;

wire forwardAD;

wire forwardBD;

//

wire [2:0] alu\_ctrl;

wire [31:0] instr;

datapath dp (

.clk (clk),

.rst (rst),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.dm2reg (dm2reg),

.we\_reg\_lh (we\_reg\_lh),

.sel\_reg\_lh (sel\_reg\_lh),

.sel\_lh (sel\_lh),

.we\_dm (we\_dm),

.alu\_ctrl (alu\_ctrl),

.ra3 (ra3),

.rd\_dm (rd\_dm),

.dm2reg\_m (dm2reg\_m),

.forwardAD (forwardAD),

.forwardBD (forwardBD),

.stallF (stallF),

.stallD (stallD),

.flushE (flushE),

.forwardAE (forwardAE),

.forwardBE (forwardBE),

.instr\_out (instr),

.pc\_current (pc\_current),

.alu\_out (alu\_out),

.wd\_dm (wd\_dm),

.rd3 (rd3),

.jmp (jmp),

.we\_reg\_e (we\_reg\_e),

.ra12e (ra12e),

.branchD (branchD),

.we\_reg\_b (we\_reg\_b),

.ra12b (ra12b),

.we\_reg\_m (we\_reg\_m),

.ra12m (ra12m),

.dm2reg\_e (dm2reg\_e),

.rs1e (rs1e),

.ra1e (ra1e),

.rs1d (rs1d),

.rt1d (rt1d)

);

hazardunit hu(

.we\_reg\_b (we\_reg\_b),

.ra12b (ra12b),

.we\_reg\_m (we\_reg\_m),

.ra12m (ra12m),

.dm2reg\_e (dm2reg\_e),

.rs1e (rs1e),

.ra1e (ra1e),

.rs1d (rs1d),

.rt1d (rt1d),

.we\_reg\_e (we\_reg\_e),

.ra12e (ra12e),

.branchD (branchD),

.jmp (jmp),

.dm2reg\_m (dm2reg\_m),

.stallF (stallF),

.stallD (stallD),

.flushE (flushE),

.forwardAE (forwardAE),

.forwardBE (forwardBE),

.forwardAD (forwardAD),

.forwardBD (forwardBD)

);

controlunit cu (

.opcode (instr[31:26]),

.funct (instr[5:0]),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.we\_dm (we\_dm),

.dm2reg (dm2reg),

.we\_reg\_lh (we\_reg\_lh),

.sel\_reg\_lh (sel\_reg\_lh),

.sel\_lh (sel\_lh),

.alu\_ctrl (alu\_ctrl)

);

endmodule

module tb\_mips\_top;

reg clk;

reg rst;

wire we\_dm;

wire [31:0] pc\_current;

wire [31:0] alu\_out;

wire [31:0] wd\_dm;

wire [31:0] rd\_dm;

wire [31:0] DONT\_USE;

mips\_top DUT (

.clk (clk),

.rst (rst),

.we\_dm (we\_dm),

.ra3 (5'h0),

.pc\_current (pc\_current),

.alu\_out (alu\_out),

.wd\_dm (wd\_dm),

.rd\_dm (rd\_dm),

.rd3 (DONT\_USE)

);

task tick;

begin

clk = 1'b0; #5;

clk = 1'b1; #5;

end

endtask

task reset;

begin

rst = 1'b0; #5;

rst = 1'b1; #5;

rst = 1'b0;

end

endtask

integer i;

initial begin

reset;

while(pc\_current != 32'h1C) tick;

tick;tick;tick;tick;tick;tick;

$finish;

end

endmodule