Overview
- Introduce the MIPS soft processor used in the ECAD+Arch labs
  - programmer’s model
  - architecture
  - Verilog implementation
- Note: this used to be lectured but it was really too much code to go through in that format. Instead a simplified SystemVerilog partial MIPS processor is presented. The design is presented here as an appendix to the slide set for documentation purposes.
- Acknowledgement: many thanks to Gregory Chadwick and Ben Roberts for refining the MIPS processor design, and to Robin Message and David Simner for their initial MIPS design.

MIPS — Overview
- see the Programmer’s Reference for an introduction
- overview of the 5 stage pipeline:

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Decode / Register Fetch</th>
<th>Execute</th>
<th>Memory Access</th>
<th>Write-back</th>
</tr>
</thead>
</table>


# Appendix — Tiger MIPS Processor Implementation

## MIPS — Defines file

```c
#define REGNUM WIDTH 4:0
// CONTROL is a set of signals, CONTROL WIDTH wide
// each define starting with CONTROL declares what
// each bit of the control signal represents
#define CONTROLWIDTH 50:0
#define CONTROL_BRANCH.TYPE 7:5
#define CONTROL_WIDE_REGNUM 4:0

#define BR_NONE 3'b000
#define BR_LTZ 3'b001
#define BR_GTZ 3'b010
#define BR_EQ 3'b011
#define BR_NE 3'b100
#define BR_LEZ 3'b101
#define BR_GTZ 3'b110

#define CONTROL_ALUCONTROL_UNSIGNED 8
#define CONTROL_ALUCONTROL/chat 9
#define CONTROL_ALUCONTROL_VARIABLE 10
#define CONTROL_ALUCONTROL_SHIFT 12:11

// ALU control signals
#define ALU_UNSIGNED 5'b0000_1
#define ALU_NONE 5'b0000_0 // unsigned version needed
#define ALU_MFHI 5'b0001_0
#define ALU_MFLO 5'b0001_0
#define ALU_MTLO 5'b0010_0 // unsigned version needed
#define ALU_MTLO 5'b0010_0
#define ALU_INVALID 5'b0011_0 // unsigned version needed
#define ALU_SUB 5'b0100_0 // unsigned version needed
#define ALU_SUB 5'b0100_0
#define ALU_ADD 5'b0101_0 // unsigned version needed
#define ALU_ADD 5'b0101_0
#define ALU_AND 5'b0110_0 // unsigned version needed
#define ALU_AND 5'b0110_0
#define ALU_NAND 5'b0111_0 // unsigned version needed
#define ALU_NAND 5'b0111_0
#define ALU_XOR 5'b1000_0
#define ALU_XOR 5'b1000_0
#define ALU_BILI 5'b1001_0 // unsigned version needed
#define ALU_BILI 5'b1001_0
#define ALU_LUI 5'b1010_0 // 1010_1 is unused
#define ALU_LUI 5'b1010_0
#define ALU_MFHI 5'b1011_0 // unsigned version needed
#define ALU_MFHI 5'b1011_0
#define ALU_MFLO 5'b1100_0 // unsigned version needed
#define ALU_MFLO 5'b1100_0
#define ALU_MTLO 5'b1101_0 // unsigned version needed
#define ALU_MTLO 5'b1101_0
#define ALU_MFHI 5'b1110_0 // unsigned version needed
#define ALU_MFHI 5'b1110_0
#define ALU_MFLO 5'b1111_0 // unsigned version needed
#define ALU_MFLO 5'b1111_0

#define CONTROL_REGRETURN 13
#define CONTROL_BRANCH 14
#define CONTROL_MEMR 15
#define CONTROL_MEMW 16
#define CONTROL_MEMM 17
#define CONTROL_LINK 18
#define CONTROL_ZEROFILL 19
#define CONTROL_REGJUMP 20
#define CONTROL_JUMP 21
#define CONTROL_MEMREAD 22
#define CONTROL_MEMWRITE 23
#define CONTROL_BRANCH 24
#define CONTROL_REGMEM 25
#define CONTROL_REGWRITE 26
#define CONTROL_COPYREAD 27
#define CONTROL_COPYWRITE 28
#define CONTROL_DCACHEFILLUSH 29
#define CONTROL_ICACHEFILLUSH 30
```
```
MIPS — Top Level Module Part 1

#include "tiger_defines.v"

module tiger(
    input clk, // clock signal
    input reset, // reset signal
    input ISStall, // instruction stall signal
    input DSStall, // data stall signal
    output ICcacheFlush, // instruction cache flush
    output DCacheFlush, // data cache flush
    input canICacheFlush, // can instruction cache flush
    input canDCacheFlush, // can data cache flush
    input irq, // interrupt request signal
    input [5:0] irqNumber, // interrupt request number
    output [31:0] pc, // program counter
    input [31:0] instrF, // fetched instruction
    output memwrite, memread, memf, memr, memzerofill, // memory access mode outputs
    output [31:0] memaddress, memwritedata, // memory address and data outputs
    input memCanRead, // can memory read
    input memCanWrite // can memory write
    )

  wire stallRqEx;
  wire exception;
  wire stallDe;
  wire clearDe;
  wire stallEx;
  wire clearEx;
  wire stallMA;
  wire clearMA;
  wire stallWB;
  wire clearWB;

  wire [31:0] instrDe;
  wire [CONTROL_WIDTH] controlDe;

  wire [31:0] instrEx;
  wire [CONTROL_WIDTH] controlEx;
  wire [31:0] branchOutEx;
  wire [31:0] rsEx;
  wire [31:0] rtEx;
  wire [31:0] CPOutEx;
  wire [31:0] instrMA;
  wire [CONTROL_WIDTH] controlMA;
  wire [31:0] branchOutMA;
  wire [3:0] bottomAddressMA;

  wire writeRegEnMA;
  wire writeRegEnCopMA;
  wire [REGNUM_WIDTH] writeRegNumMA;
  wire [31:0] writeRegDataMA;

  wire [31:0] instrWB;
  wire [CONTROL_WIDTH] controlWB;
  wire [31:0] MACwib;
  wire [31:0] branchOutWB;
  wire [31:0] writeRegDataWB;
  wire [REGNUM_WIDTH] writeRegNumWB;
  wire writeRegEnWB;
  wire writeRegEnCopWB;
tiger_stalllogic al {
  .controlDe (controlDe),
  .controlEx (controlEx),
  .instrDe (instrDe),
  .instrEx (instrEx),
  .writeRegNumMA (writeRegNumMA),
  .writeRegEnMA (writeRegEnMA),
  .writeRegEnCopMA (writeRegEnCopMA),
  .writeRegNumWB (writeRegNumWB),
  .writeRegEnWB (writeRegEnWB),
  .writeRegEnCopWB (writeRegEnCopWB),
  .stallRqEx (stallRqEx),
  .exception (exception),
  .iStall (iStall),
  .dStall (dStall),
  .clearDe (clearDe),
  .stallDe (stallDe),
  .clearEx (clearEx),
  .stallEx (stallEx),
  .clearMA (clearMA),
  .stallMA (stallMA),
  .clearWB (clearWB),
  .stallWB (stallWB),
};

// fetch stage
  tiger_fetch f(
    .clk (clk),
    .reset (reset),
    .stall (stallDe),
    .clear (clearDe),
    .instr (instrF),
    .instrDE (instrDe)
  );

// decode stage
  tiger_decode d(
    .clk (clk),
    .reset (reset),
    .stall (stallEx),
    .clear (clearEx),
    .irq (irq),
    .irqNumber (irqNumber),
    .instr (instrDe),
    .controlDe (controlDe),
    .writeRegEnWB (writeRegEnWB),
    .writeRegEnCopWB (writeRegEnCopWB),
    .writeRegNumWB (writeRegNumWB),
    .writeRegDataWB (writeRegDataWB),
    .exception (exception),
    .instrEx (instrEx),
    .controlEx (controlEx),
    .rtEx (rtEx),
    .CPOutEx (CPOutEx),
    .branchoutEx (branchoutEx),
    .nextpc (pc)
  );
// feed forward path for first operand 
wire [31:0]rsExFF;
tiger_ff_for_rs,
    .regnum(instrEx[25:21]),
    .writer1(writeRegNumMA),
    .writer2(writeRegNumWB),
    .writerent(writeRegEnMA),
    .writerent2(writeRegEnWB),
    .regdata(rsEx),
    .writerregdata1(writeRegDataMA),
    .writerregdata2(writeRegDataWB),
    .out(rsExFF)
];

// feed forward path for second operand 
wire [31:0]rtExFF;
tiger_ff_for_rt,
    .regnum(instrEx[20:16]),
    .writer1(writeRegNumMA),
    .writer2(writeRegNumWB),
    .writerent(writeRegEnMA),
    .writerent2(writeRegEnWB),
    .regdata(rtEx),
    .writerregdata1(writeRegDataMA),
    .writerregdata2(writeRegDataWB),
    .out(rtExFF)
];

// execute stage 
tiger_execute ex(
    .clk(clk),
    .reset(reset),
    .stall(stallMA),
    .clear(clearMA),
    .instr(instrEx),
    .control(controlEx),
    .rs(rsExFF),
    .rt(rtExFF),
    .branchout(branchoutEx),
    .CP0out(CP0outEx),
    .instrMA(instrMA),
    .controlMA(controlMA),
    .executeMA(executeoutMA),
    .branchoutMA(branchoutMA),
    .// .bottomaddressMA(bottomaddressMA),
    .stallRq(stallRqEx),
    .memread(memread),
    .mam16(mam16),
    .mam8(mam8),
    .memwrite(memwrite),
    .memaddress(memaddress),
    .memwritedata(memwritedata),
    .memCanRead(memCanRead),
    .memCanWrite(memCanWrite),
    .iCacheFlush(iCacheFlush),
    .dCacheFlush(dCacheFlush),
    .canICacheFlush(canICacheFlush),
    .canDCacheFlush(canDCacheFlush)
);
MIPS — Top Level Module Part 4

// memory access stage
memoryaccess ma(
    .clk(clk),
    .reset(reset),
    .clear(clearWB),
    .stall(stallWB),
    .memoryaccess ma(  
        .clk(clk),
        .reset(reset),
        .clear(clearWB),
        .stall(stallWB),
    );

// Pipeline registers in
    .instr(instrMA), // instruction
    .control(controlMA), // control signals
    .executeout(executeoutMA), // Output of the execute stage
    .branchout(branchoutMA), // PC value for use with b w/ link
    .bottoms 2 bits of address, used if we're reading from memory
    // using a left or right hand read instruction
    .bottomaddress(bottomaddressMA),

// Pipeline registers out
    .controlWB(controlWB),  
    .branchoutWB(branchoutWB),
    .instrWB(instrWB),  
    .controlWB(controlWB),
    .branchoutWB(branchoutWB),
    .instrWB(instrWB),  
    .controlWB(controlWB),
    .branchoutWB(branchoutWB),
    .instrWB(instrWB),

// Read data from memory
    .memreaddata(memreaddata),

// Signals used for controlling feed-forward and pipeline stall
    .writeRegEn(writeRegEnMA), // True if we will write to a register (in the WB stage)
    .writeRegNum(writeRegNumMA), // Which register we will write to
    .writeRegData(writeRegDataMA)  // What we would write to the register
);

// writeback stage
writeback wb(
    .clk(clk),
    .instr(instrWB), /* instruction */
    .control(controlWB), /* control signal */
    .branchout(branchoutWB), /* PC value for use with b w/ link */
    .branchoutWB(branchoutWB), /* Output of the memory access stage */

// Register write control
    .writeRegEn(writeRegEnWB), /* True if we want to write to a register */
    .writeRegNum(writeRegNumWB), /* Which register we want to write to */
    .writeRegData(writeRegDataWB) /* What data we want to write to it */
);

endmodule
MIPS — Instruction Fetch

`include "tiger_defines.v"

module tiger_fetch(
    input clk,       // clock signal
    input reset,    // reset signal
    input stall,    // whether this unit is stalled on the pipeline
    input clear,    // clear signal
    input [31:0] instr,   // instruction loaded from memory
    output reg [31:0] instrDE  // output instruction
);

always @(posedge clk)
begin
    if (reset || clear) begin
        instrDE <= 0;
    end else if (stall) begin
        // Stall instrDE
    end else begin
        instrDE <= instr;
    end
end
endmodule
Appendix — Tiger MIPS Processor Implementation

MIPS — Instruction Decode

```
#include "tigerDefines.v"

module tiger_decode (input clk, // clock signal
                    input reset, // reset signal
                    input stall, // stall signal
                    input clear, // clear signal
                    input irq, // interrupt signal
                    input [5:0] irqNumber, // current instruction
                    input [31:0] instr, // current instruction
                    input writeRegEnWB, // True if write back wants to write to a register
                    input writeRenEnCopWB, // True if write back wants to write a coprocessor register
                    input [31:0]writeRegNumWB, // Register write back wants to write to
                    input [31:0]writeRegDataWB, // Data write back wants to write
                    output exception, // output containing the current instruction
                    output [CONTROL WIDTH] controlDe, // output for the control signals
                    output [CONTROL WIDTH] controlEx, // first of 2 operand outputs
                    output [CONTROL WIDTH] rEx, // second of 2 operand outputs
                    output [CONTROL WIDTH] CP0OutEx, // Coprocessor register contents
                    output [31:0] branchoutEx, // branch address
                    output [31:0] nextpc, // next instruction address
                    wire branchDelay;
                    wire iCacheFlush;
                    wire dCacheFlush;
                    // decoder module
                    wire [15:0] controls;
                    wire [4:0] alucontrol;
                    wire [2:0] branchtype;
                    wire [4:0] destreg;
                    tiger_decoder d
                        .instr(instr),
                        .controls(controls),
                        .alucontrol(alucontrol),
                        .branchtype(branchtype),
                        .destreg(destreg));
```

// setup the control signals
wire [CONTROL_WIDTH] control = {iCacheFlush, dCacheFlush, controls, alucontrol, branchtype, destreg};
assign controlDE = control;

// register file - 31 registers (numbered 1 to 31) each 32 bits long
reg [31:0] r[31:1];
reg [31:0] cause;
reg [31:0] status;
reg [31:0] epc;
wire [31:0] epcDe;

wire break = instr[31:26] == 6'b00_0000 && instr[5:0] == 6'b00_1101;
wire syscall = instr[31:26] == 6'b00_0000 && instr[5:0] == 6'b00_1100;
assign exception = (!status[0] && irq) || break || syscall;
assign iCacheFlush = (writeRegEnCopWB && writeRegNumWB == 5'd3 && writeRegDataWB[0] == 1'b1);
assign dCacheFlush = (writeRegEnCopWB && writeRegNumWB == 5'd3 && writeRegDataWB[1] == 1'b1);

// If we're reading from $zero, register value is 0.
// otherwise if it's a register WB is currently wanting to write
// to pass the value straight through; otherwise use the value in the register file
wire [31:0] rsFF = instr[25:21] == 5'b0 ? 32'b0 :
                      instr[25:21] == writeRegNumWB && writeRegEnWB ? writeRegDataWB :
                      rf[instr[25:21]];

// If we're reading from $zero, register value is 0.
// otherwise if it's a register WB is currently wanting to write
// to pass the value straight through; otherwise use the value in the register file
wire [31:0] rtFF = instr[20:16] == 5'b0 ? 32'b0 :
                      instr[20:16] == writeRegNumWB && writeRegEnWB ? writeRegDataWB :
                      rf[instr[20:16]];

always @(posedge clk)
begins
  // if the register number is not 5'b00000 and register writeback from memory is enabled
  // then store the data from memory in the register file
  if (writeRegNumWB != 5'b00_0000 && writeRegEnWB) begin
    rf[writeRegNumWB] <= writeRegDataWB;
  end
  if (writeRegEnCopWB) begin
    case(writeRegNumWB)
      5'b00_0000: cause <= writeRegDataWB;
      5'b00_0001: status <= writeRegDataWB;
      5'b00_0010: epc <= writeRegDataWB;

endcase
end
if ( exception && ! clear && ! stall ) begin
  if ( irq )
    cause <= (branchDelay, 15'b0, irqNumber, 10'b0);
  else if (break)
    cause <= (branchDelay, 26'b0, 4'd9, 1'b0);
  else if (syscall)
    cause <= (branchDelay, 26'b0, 4'd8, 1'b0);
  else
    cause <= (branchDelay, 26'b0, 4'hf, 1'b0);
  status <= {status[31:1], 1'b1};
  epc <= epcDe;
end
if ( reset ) begin
  instrEx <= 0;
  controlEx <= 0;
  rsEx <= 0;
  rtEx <= 0;
  cause <= 0;
  status <= 0;
  epc <= 0;
  // reset the stack pointer
  rf[29] <= 32'h00780000;
end else if ( stall ) begin
  // Stall instrEx
  // Stall controlEx
  // Stall rsEx
  // Stall rtEx
  // Stall CPOutEx
  instrEx <= instr;
  controlEx <= control;
  rsEx <= rsFF;
  rtEx <= rtFF;
  CPOutEx <= instr[15:11] == 5'b0_0000 ? cause :
             instr[15:11] == 5'b0_0001 ? status :
             instr[15:11] == 5'b0_0010 ? epc :
             5'b_xxxx;
end
endmodule
module tiger_decoder(
    input [31:0] instr, // instruction to decode
    output reg [15:0] controls, // control signals
    output reg [4:0] alucontrol, // alu control code
    output reg [2:0] branchtype, // branch code
    output [4:0] destreg // destination register for result
    );

assign destreg = controls[5] ? 5’d31 :
    instr[15:11];

wire [5:0] op = instr[31:26];
wire [5:0] funct = instr[5:0];

always @(+) begin
    // BEQ, BNE, BLEZ , BGTZ
    if (op[5:2] == 4’d0001) begin
        controls <= 16’b0000_0000_0000_0000;
        alucontrol <= 'ALU_NONE;
        branchtype <= (op[1] | op[0] , ~ (op[1] ^ op[0]) , ~ op[0]);
    end else if (op == 6’d00001) begin
        controls <= 16’b0000_0000_0000_0000;
        alucontrol <= 'ALU_NONE;
        branchtype <= (1’d0, instr[16] , ~ instr[16]);
    end else begin
        branchtype <= 'BR_NONE;
    end
    // R-type instruction
    if (op == 6’d000000) begin
        controls <= (funct == 6’d00_1000) ? 16’b0000_0000_0000_0000 :
            : (funct == 6’d00_1001) ? 16’b0000_0000_0010_0000 :
            : 16’b0000_0000_0000_0000;
        // ADD, ADD UNSIGNED, SUB, SUB UNSIGNED
        if (funct[5:2] == 4’d0000) begin
            alucontrol <= (2’d00, funct[1], ~funct[1], funct[0]);
        end else if (funct[5:2] == 4’d0010) begin
            alucontrol <= (1’d0, funct[1], (2’d00)) , funct[0]);
        end else if (funct[5:2] == 4’d0011) begin
            alucontrol <= (funct[1], (0’d00)) , funct[0]);
        end else if (funct[5:2] == 4’d1000) begin
            alucontrol <= (funct[1], funct[1], funct[0]);
        end else if (funct[5:3] == 3’d0000 && funct[1] || ~ funct[0]) begin
            alucontrol <= (2’d01, funct[2:1], funct[0]);
        end else if (funct[5:1] == 5’d0100) begin
            alucontrol <= (4’d0101, funct[0]);
        end else begin
            alucontrol <= 'ALU_NONE;
        end
end
MIPS — Instruction Decoder Implementation

```vhdl
// ADD, ADDU, SLT, SLTU, AND, OR, XOR, LW
end else if (op[5:3] == 3'b001) begin
  alucontrol <= 'ALU_ADD;
  controls <= (2'b0, 3'b11, instr[12:20], instr[5:0], 5'b1_); end
// SB, SH, SW
end else if (op[5:2] == 4'b1010 && (op[1] | op[0])) begin
  alucontrol <= 'ALU_ADD;
  controls <= (2'b0, 9'b000, instr[12:5], instr[5:0], 5'b0_);
// J, JL
end else if (op[5:1] == 5'b0000) begin
  alucontrol <= 'ALU_NONE;
  controls <= (2'b0, 7'b000_0000, instr[12:5], instr[5:0], 5'b0_);
// MUL, MFC0, MTC0
end else if (op == 6'b010000) begin
  alucontrol <= 'ALU_ADD;
  controls <= (2'b0, 7'b000_0000, instr[12:5], instr[5:0], 5'b0_);
end else begin
  alucontrol <= 'ALU_NONE;
  controls <= 16'b0000_0000_0000_0000;
end
endmodule
```
MIPS — Branch Unit

clk → Branch
reset → Branch
stall → Branch
exception → Branch

instr → Branch
rs → Branch
rt → Branch
control → Branch

Branch → branchDelay
Branch → epc
Branch → nextpc
Branch → branchout
module tiger_branch(
    input clk,          // clock signal
    input reset,        // reset signal
    input stall,        // stall signal
    input exception,    // interrupt request signal
    input [31:0] instr, // current instruction
    input [CONTROL_WIDTH] control, // control signals
    input [31:0] rs,    // first operand
    input [31:0] rt,    // second operand
    output reg [31:0] branchout, // return address if we are doing a branch with link operation
    output reg [31:0] epc,   // address of the next instruction to load
    output reg [31:0] branchDelay
)

parameter EXCEPTION_HANDLER_ADDR = 32'h0000_0A00;
parameter BOOT_ADDR = 32'h0000_0000;

reg [31:0] currentpc;
wire [2:0] branchtype = control['CONTROL_BRANCHTYPE];
wire takebranch = control['CONTROL_BRANCH]
  && (branchtype == 'BR_LTZ && rs[31]) // branch if < 0 instruction and the MSB of the first operand is set
  || (branchtype == 'BR_GEZ && rs == rt) // branch if >= 0 instruction and both operands are equal to each other
  || (branchtype == 'BR_EQ && rs == rt) // branch if not equal instruction and both operands are not equal
  || (branchtype == 'BR_NE && rs != rt) // branch if != instruction and both operands are not equal
  || (branchtype == 'BR_LEZ && rs[31]) // branch if <= 0 instruction and either the MSB is set
  || (branchtype == 'BR_GTZ && !rs[31] && rs != 0) // we have a branch if > 0 instruction, and the MSB is not set, and the register does not equal 0;

wire [31:0] signimmsh = {{14:14 instr[15]:0}, instr[15:0], 2'b00};
wire [31:0] pcplus4 = currentpc + 4;
assign nextpc = reset ? BOOT_ADDR // reset the start address
  : stall ? currentpc + 4 // stalled so do not change the current address
  : takebranch ? currentpc + signimmsh // relative branch
  : control['CONTROL_JUMP] ? (currentpc[31:28], instr[25:0], 2'b0) // immediate jump
  : control['CONTROL_REGJUMP] ? rs + popplus4 // jump to the address contained in operand 1
  : pcplus4; // if a jump is not being performed, pc=pc+4

// If we're in a branch delay slot the exception program counter needs to
// point to the branch rather than the instruction in the delay slot
// otherwise we need to point to the instruction where the exception occurred
assign epc = branchDelay ? branchout : 8;

always @(posedge clk)
begin
  currentpc <= nextpc;
  if ((takebranch || control['CONTROL_JUMP'] || control['CONTROL_REGJUMP']) & stall) branchDelay <= 1;
  else if (!stall) branchDelay <= 0;
  if (reset) begin
    branchDelay <= 0;
  end else if (!stall) begin
    // set the return branch address
    branchout <= pcplus4;
  end
endmodule
MIPS — Execute Unit

 clk
 reset
 stall
 clear
 rs
 rt
 CPOut

 instr
 control
 branchout

 Execute
 mult
 div

 multu
 divu

 alu

 shifter

 executeoutMA
 instrMA
 controlMA
 branchoutMA
`include "tiger_defines.v"

module tiger_execute(
  input clk,          // clock signal
  input reset,        // reset signal
  input stall,        // stall signal
  input [31:0] instr, // current instruction
  input [CONTROL_WIDTH] control, // control signals
  input [31:0] rs,    // first operand
  input [31:0] rt,    // second operand
  input [31:0] branchout, // return branch address
  input [31:0] CPOut, // coprocessor register value
  output reg [31:0] instrMA, // output instruction
  output reg [CONTROL_WIDTH] controlMA, // output control signals
  output reg [31:0] executeoutMA, // output execute unit result
  output reg [31:0] branchoutMA, // output branch return address
  output reg [1:0] bottomaddressMA, // output bottom 2 bits of the address
  output stallReq,    // do we wish to stall the pipeline
  output memread, mem16, mem8, memwrite, // what type of memory access do we require
  output iCacheFlush, dCacheFlush, // address in memory to write to + the data to write
  input memCanRead, memCanWrite, // input canCacheFlush, canDCacheFlush
)

// sign-extend the 2 operands and multiply them together to form a 64 bit number
// wire [63:0] mult = [[32:0]rs[31] & !control[CONTROL_ALUCONTROL_UNSIGNED]] & rs
//     * [[32:0]rt[31] & !control[CONTROL_ALUCONTROL_UNSIGNED]] & rt;
wire [63:0] mults;
wire [63:0] multu;
tiger_mult ms(rs, rt, mults);
tiger_multu mu(rs, rt, multu);

// countdown indicates how many clock cycles until the HI and LO registers are valid
reg [3:0] countdown;
reg source;

// HI and LO registers
reg [31:0] high,
low;

// perform a signed division on the 2 operands
wire [31:0] divLO, divHI;
tiger_div div 
  .clock(clk),
  .denom(rt),
  .numer(rs),
  .quotient(divLO),
  .remainder(divHI);

// perform an unsigned division on the 2 operands
wire [31:0] divuLO, divuHI;
tiger_divu divu 
  .clock(clk),
  .denom(rt),
  .numer(rs),
  .quotient(divuLO),
  .remainder(divuHI);

// alu unit
wire [31:0] aluout;
tiger_alu alu 
  .srca(rs), // first operand is always rs
  .srcb(control[CONTROL_USEIMM] ? (31:0) & !control[CONTROL_ZEROFILL]) & instr[15:0] : rt ),
  .alucontrol(control[CONTROL_ALUCONTROL]),
  .aluout(aluout);

);
MIPS — Execute Unit Implementation Part 2

// shifter unit
wire [31:0] shiftout;
tiger shifter shift(
    .src(t),  // shift register
    .dir(control | CONTROL_ALUCONTROL_RIGHT),
    .aluigned((control | CONTROL_ALUCONTROL_UNSIGNED),
              .shifted(shiftout));

// set the countdown
// MTHI, MTLO and MUL only require a single cycle, but division requires 12 cycles to complete
wire [3:0] newcountdown1 =.reset
  ? 4'd0
  : countdown > 0
      ? countdown - 4'd1
      : 4'd0;
wire [3:0] newcountdown2 =reset
  ? 4'd0
  : countdown > 0
      ? countdown - 4'd1
      : 4'd0;

// if the pipeline is stalled we only decrement the countdown (if allowed)
wire [3:0] newcountdown2 =reset
  ? 4'd0
  : countdown > 0
      ? countdown - 4'd1
      : 4'd0;

// set the LO register
wire [31:0] newlow1 =reset
  ? 32'd0
  : control | CONTROL_ALUCONTROL == 'ALU_MLO
      ? rs /* move to LO */
      : control | CONTROL_ALUCONTROL == 'ALU_MULT
      ? multi[31:0] // low 32 bits of multiplication
      : (source ? divLO : divuLO)  // low
        = countdown1?
          ? (source ? divLO : divuLO)
          : low;
wire [31:0] newlow2 =reset
  ? 32'd0
  : countdown1
    ? (source ? divLO : divuLO)
    : low;

// set the HI register
wire [31:0] newhigh1 =reset
  ? 32'd0
  : control | CONTROL_ALUCONTROL == 'ALU_MTHI
      ? rs /* move to HI */
      : control | CONTROL_ALUCONTROL == 'ALU_MULT
      ? multi[63:32] // high 32 bits of the multiplication
      : control | CONTROL_ALUCONTROL == 'ALU_UNSIGNED
      ? multi[63:32]
      : (source ? divHI : divuHI)  // high
        = countdown1?
          ? (source ? divHI : divuHI)
          : high;
wire [31:0] newhigh2 =reset
  ? 32'd0
  : countdown1
    ? (source ? divHI : divuHI)
    : high;
always @ (posedge clk)
begin
    countdown <= stall ? newcountdown1 : newcountdown2;
    low <= stall ? newlow1 : newlow2;
    high <= stall ? newhigh1 : newhigh2;
end
if (reset || clear) begin
    instrMA <= 0;
    controlMA <= 0;
    executeoutMA <= 0;
    branchoutMA <= 0;
end else if (stall) begin
    // stall instrWB
    // stall controlWB
    // stall executeoutWB
    // stall branchoutWB
end else begin
    instrMA <= instr;
    controlMA <= control[CONTROL_ALUCONTROL] == 'ALU_MUL ? mul[31:0] // lower 32 bits of the multiplication
                                          : control[CONTROL_ALUCONTROL] == 'ALU_MFHI ? high // MFH => contents of the HI register
                                          : control[CONTROL_ALUCONTROL] == 'ALU_MFLO ? low // MFLO => contents of the LO register
                                          : control[CONTROL_ALUCONTROL] == 'ALU_MFHI ? rt // second operand
                                          : control[CONTROL_ALUCONTROL] == 'ALU_MFLO ? rt // second operand
                                          : control[CONTROL_ALUCONTROL] == 'ALU_MFHI ? branchout // return address for link operation
                                          : control[CONTROL_ALUCONTROL] == 'ALU_MFLO ? branchout // return address for link operation
                                          : control[CONTROL_ALUOUT] || control[CONTROL_ALUOUTSHIFT] ? aluout // otherwise give the output of the ALU
                                          : bottomaddressMA <= aluout[1:0];
    end
end module
module tiger_alu
input signed [31:0] srca, srcb; // 2 operands
input [4:0] alucontrol; // What function to perform
output [31:0] aluout; // Result of the function

wire unsigned [31:0] srcau = srca;
wire unsigned [31:0] srcbu = srcb;

endmodule
MIPS — Shifter

Shifter

src
amt
dir
alusigned

shifted
module tiger_shifter(
    input [31:0] src, // source data
    input [4:0] amt, // number of bits to shift by
    input dir, // direction to shift (0 = left; 1 = right)
    input alusigned, // signed shift? 0 = unsigned; 1 = signed
    output [31:0] shifted // output
); // fill bit for right shifts
wire fillbit = alusigned & src[31];
// do a right shift by shifting 0-5 times
wire [31:0] right16;
wire [31:0] right8;
wire [31:0] right4;
wire [31:0] right2;
wire [31:0] right1;
wire [31:0] right;
assign right8 = amt[3] ? {(8{fillbit}), right16[31:8]} : right16;
assign right4 = amt[2] ? {(4{fillbit}), right8[31:4]} : right8;
assign right2 = amt[1] ? {(2{fillbit}), right4[31:2]} : right4;
assign right1 = amt[0] ? {(1{fillbit}), right2[31:1]} : right2;
assign right = right1;
// select the correct shift output
assign shifted = dir ? right : left;
endmodule
MIPS — Memory Access

```
#include "tiger_defines.v"

module tiger_memoryaccess(
    input clk,          // clock signal
    input reset,        // reset signal
    input clear,        // clear signal
    input stall,        // stall signal
    input [31:0] instr, // current instruction
    input [CONTROL_WIDTH] control, // control signals
    input [31:0] executeout, // output of the execute stage
    input [31:0] branchout);

output reg [CONTROL_WIDTH] controlWB,
output reg [31:0] MAOutWB,
output reg [31:0] branchoutWB,
output reg [31:0] instrWB,
output reg [31:0] memreaddata, // data read from memory
output [31:0] memData;

assign writeRegEn = control[CONTROL_REGWRITE];
assign writeRegEnCop = control[CONTROL_COPWRITE];
assign writeRegNum = control[CONTROL_WTEREGNUM];
assign writeRegData = executeout;
wire [31:0] memData;
assign memData = memreaddata;
// Pipeline, on the positive clock edge move everything to the next pipeline stage
always @posedge clk begin
    if (reset || clear) begin
        controlWB <= 0;
        MAOutWB <= 0;
        branchoutWB <= 0;
        instrWB <= 0;
    end else if (stall) begin
        stall controlWB
        stall MAOutWB
        stall branchoutWB
        stall instrWB
    end else begin
        controlWB <= control;
        MAOutWB <= control[CONTROL_MEMWRITE] ?
            (control[CONTROL ZEROFILL] && control[CONTROL_MEM]) ?
            (memData[7]) : memData[7 : 0]
            : memData;
        branchoutWB <= branchout;
        instrWB <= instr;
    end
endmodule
```
module tiger_writeback(
  input clk, // clock signal
  input [31:0] instr, // current instruction
  input [CONTROL_WIDTH] control, // control signals
  input [31:0] branchout, // the branch address to save if we are doing a link operation
  input [31:0] MAOut, // result from the memory access stage
  output writeRegEn, // output indicating if we wish to write to a register
  output writeRegEnCop, // output indicating if we wish to write to a coprocessor register
  output [REGNUM_WIDTH] writeRegNum, // what register number to write to
  output [31:0] writeRegData // data to store in the register
);

assign writeRegEn = control[CONTROL_REGWRITE];
assign writeRegEnCop = control[CONTROL_COPWRITE];
assign writeRegNum = control[CONTROL_WRITEREGNUM];
assign writeRegData = MAOut;
endmodule
MIPS — Feed-Forward Paths

module tiger_ff(
    input [4:0] regnum, // register number that we are writing to
    input writerreg1, // register number the execute unit wishes to write to
    input writerreg2, // register number the MIB unit wishes to write to
    input writerregen1, // enable WB for the execute unit
    input writerregen2, // enable WB for the MIB unit
    input [31:0] regdata, // current contents of the register
    input writerregdata1, // data the execute unit wishes to write to the register
    input writerregdata2, // data the MIB unit wishes to write to the register
    output [31:0] out // output)
);

// The following code performs the same operation as the line commented
// below but synthesises to a better circuit

/*
   assign out = (regnum == 5'b0) ? 32'b0:
   (regnum == writerreg1) && writerregen1 ? writerregdata1:
   (regnum == writerreg2) && writerregen2 ? writerregdata2 : regdata;
*/

// check to see if regnum == writerreg2 and writerregen2 is enabled
wire en2;
tiger_ff_compare c2(regnum, writerreg2, writerregen2, en2);
// if it is enabled, then write the data to the register, otherwise write the current contents of the register
// back to it (i.e. do not change the contents of the register
wire [31:0] muxedin2=en2 ? writerregdata2 : regdata;

// check to see if regnum == writerreg1 and writerregen1 is enabled
wire en1;
tiger_ff_compare c1(regnum, writerreg1, writerregen1, en1);
// if it is enabled, then write the data to the register, otherwise use the result of the MIB checker above
wire [31:0] muxedin1=en1 ? writerregdata1 : muxedin2;
// check to see if the register number is zero
// we do not have a 5-input or gate, so use a 4 input and feed
// the result into a 2-input gate
wire notzero1;
or nz1(notzero1, regnum[0], regnum[1], regnum[2], regnum[3]);
wire notzero;
or nz(notzero, notzero1, regnum[4]);
// if the register number is zero, the 'and' ensures we return the constant zero, as per the MIPS specification
// otherwise we return the data determined by the multiplexer
and a[31:0]out, muxedin1, notzero);
endmodule
module tiger_ff_compare(
    input [4:0] regnum, // the register number we are writing to
    input [4:0] writereg, // the register number we wish to write to
    input writeregen, // write-enable signal
    output en // output indicating if the write is enabled, and the register that is being
             // written to is the same as the one we would like to write to
);

// The following code performs the operation below
// but synthesizes to a better circuit on the FPGA

// assign en = (regnum == writereg) && writeregen ? 1'b1 : 1'b0;

// wires indicating equality
wire eq[5:0];
wire anded[2:0];

// check to see if bits [1:0] of the register numbers are equal
xnor eq[0], regnum[0], writereg[0]; // and ab(anded[0], eq[0], eq[1]);

// check to see if bits [3:2] of the register numbers are equal
xnor eq[2], regnum[2], writereg[2]; // and a1(anded[1], eq[2], eq[3]);

// check to see if bit [4] of the register numbers is equal
xnor eq[4], regnum[4], writereg[4]; // and a2(anded[2], eq[4], eq[5]);

// we assign the 6th bit of eq the write-enable signal
assign eq[5] = writeregen;

// the write is allowed only if all bits of the register numbers are equal and
// the write has been enabled
and(en, anded[0], anded[1], anded[2]);
endmodule
MIPS — Stall Logic

#include "tiger_defines.v"

module tiger_stall_logic (input [CONTROL_WIDTH]controlDe, input [CONTROL_WIDTH]controlEx, input [31:0]instrDe, input [31:0]instrEx, input [REGNUM_WIDTH]writeRegNumMA, input writeRegEnMA, input writeRegEnCopMA, input [REGNUM_WIDTH]writeRegNumWB, input writeRegEnWB, input writeRegEnCopWB, input stallReqEx, input exception, input iStall, input dStall, output clearDe, output stallDe, output clearEx, output stallEx, output clearMA, output stallMA, output clearWB, output stallWB);

// needStallX is high for stage X if that stage
// needs a stall for some reason, so the stage before
// it must also stall, and every stage after it
// must be cleared (introduce bubbles)
// So we clear a stage if the stage before it needs
// a stall (needStallX where X is the previous stage is
// high) and if the stage itself is not stalled
// We stall a stage if it needs a stall or if a stage
// following it is stalled.

wire needStallIDe;
wire needStallIME;
wire needStallWB;
assign clearDe = exception & iStallDe;
assign stallIDe = needStallIDe | stallEx | iStall;
assign clearEx = (needStallIDe | iStall) & !stallEx;
assign stallEx = needStallEx | stallMA;
assign clearMA = needStallEx & iStallMA;
assign stallMA = needStallMA | stallWB;
assign clearWB = needStallMA & iStallWB;
assign stallWB = needStallWB;
// Is the instruction in the decode stage a eq/ne branch?
wire takeBranchEqNeDe = controlDe[CONTROL_BRANCH] &
{controlDe[CONTROL_BRANCHTYPE] == BR_EQ || controlDe[CONTROL_BRANCHTYPE] == BR_NE};

// Is the instruction in the decode stage any kind of branch or a register jump?
wire takeBranchOrJumpDe = controlDe[CONTROL_BRANCH] || controlDe[CONTROL_JUMP];

// Does the instruction in the execute stage want to write to
// either the rs or the rt registers for the instruction in
// the decode stage. If writing to $zero, we don’t care, so set to false

// Does the instruction in the memory access stage want to write to
// either the rs or the rt registers for the instruction in
// the decode stage. If writing to $zero, we don’t care, so set to false
wire rtInMA = instrDe[20:16] == writeRegNumMA && instrDe[20:16] != 0;

// We need a stall in the decode stage
assign needStallDe =
// if we’re performing an eq/ne branch and the rt register is in the execute stage or
// memory access stage (so the stall will cause a wait until it is written back so
// we can then use them for the branch)
takeBranchEqNeDe && (rtInE || controlEx[CONTROL_REGWRITE]) || (rsInMA && writeRegEnMA);  
// or
// if we’re taking any branch or a register jump and the rs register is in the execute stage
// or memory access stage (so the stall will cause a wait until it is written back so
// we can then use them for the branch)
takeBranchOrJumpDe && (rsInE || controlEx[CONTROL_REGWRITE]) || (rsInMA && writeRegEnMA);  
// or
// if there’s a read instruction in execute and it’s going to write to
// a register we need, so we must wait for the read to complete (load stall)
takeBranchOrJumpDe || (rtInE || controlEx[CONTROL_MEMREAD]) || (rsInMA && writeRegEnMA);  
// or
// if in decode there’s a coprocessor read instruction and we’re writing to the coprocessor
// further up the pipeline (not always a hazard, and could have been solved by forwarding,
// however we read or write from the coprocessor than often we use a small amount
// of stall logic to handle possible hazards, rather than forwarding or more complex stall
// logic as the performance benefits of doing so are negligible).
controlDe[CONTROL_COPREAD] && (controlEx[CONTROL_COPWRITE]
| writeRegEnCopMA
| writeRegEnCopWB);
Appendix — Tiger MIPS Processor Implementation

MIPS — Stall Logic Implementation Part 2

// We need a stall in the execute stage if the execute stage requests one
assign needStallEx = stallReqEx;
// We need a stall in the memory access stage if there is a data stall (i.e.
// we must wait for the data cache to complete its fetch)
assign needStallMA = dStall;

// If the execute stage is stalled and write back needs to write a register
// that execute needs we must stall write back as well otherwise when the
// execute stage ceases to be stalled the feedforward from the write back
// will not give the correct value. So...
// We need a stall in the write back stage if
assign needStallWB =
  // The write back stage will write to a register, that isn't $zero
  writeRegEnWB && writeRegNumWB == 5'd0 && // And
  (needStallEx || needStallMA)
  ||
  (writeRegEnCopWB && writeRegNumWB == 5'd3 &&
   clearDe || needStallDe || iStall);

// Are we going to need register rs, given the control signals
// and it's not a branch
function needsRsAndNotBranch;
begin
  needsRsAndNotBranch = !control[CONTROL_IRQRETURN] &&
    !control[CONTROL_JUMP] &&
    !control[CONTROL_REGJUMP] &&
    !control[CONTROL_BRANCH];
endfunction

// Are we going to need register rt, given the control signals
// and it's not a branch
function needsRtAndNotBranch;
begin
  needsRtAndNotBranch = !control[CONTROL_IRQRETURN] &&
    !control[CONTROL_JUMP] &&
    !control[CONTROL_REGJUMP] &&
    !control[CONTROL_BRANCH] &&
    (!control[CONTROL_USEIMM] ||
     control[CONTROL_MEMWRITE] ||
     control[CONTROL_MEMLE] ||
     control[CONTROL_JALM]);
endfunction
endmodule