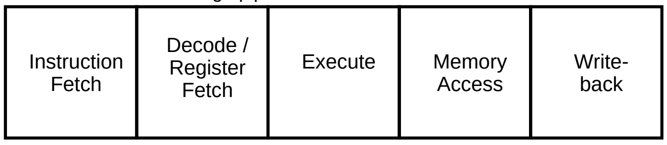
Computer Design — Appendix

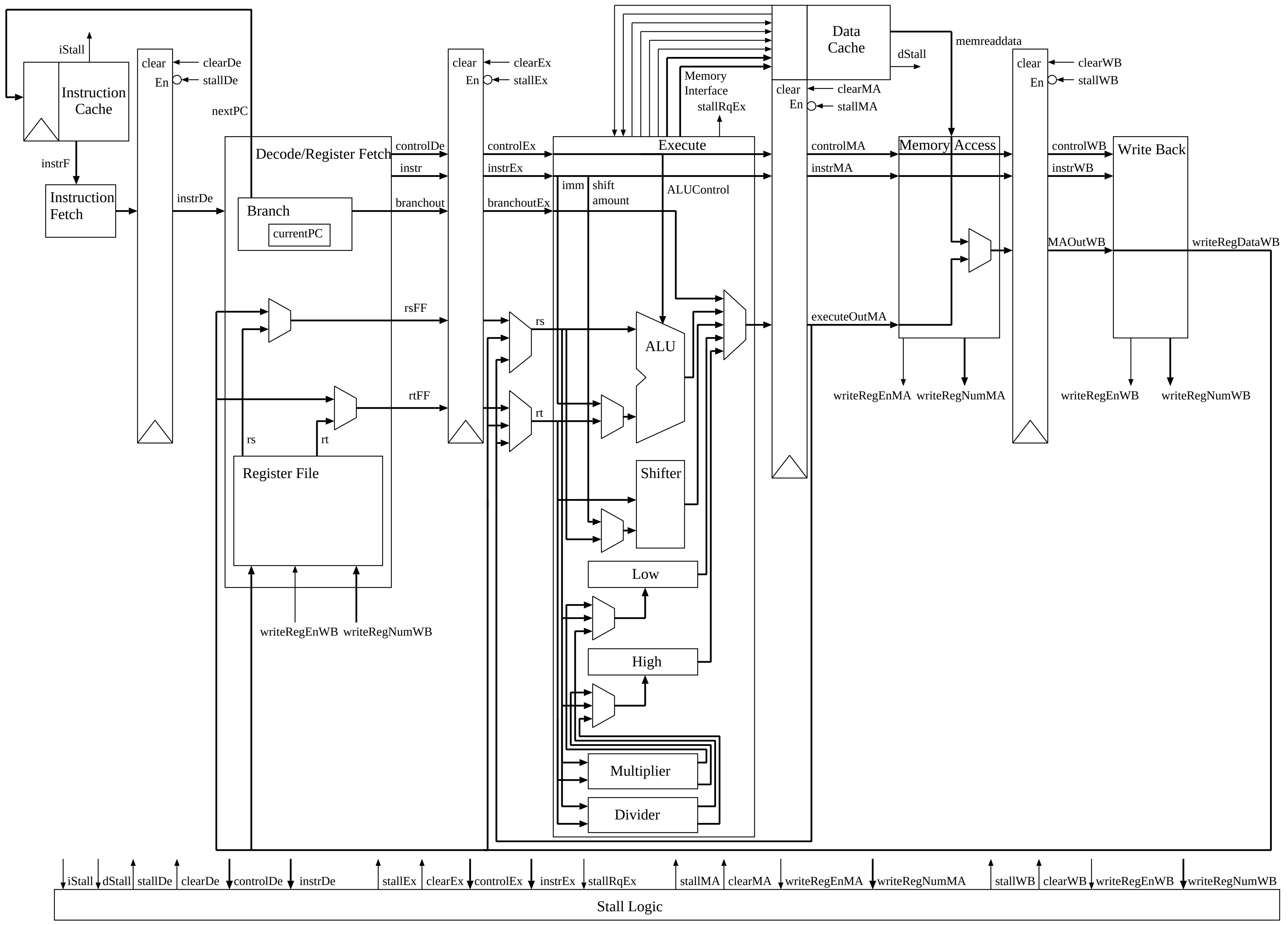
Overview

- 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 though 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:




iStall
Instruction Cache
instrF
Instruction Fetch
clear
clearDe
En
stallDe
nextPC
instrDe
Decode/Register Fetch
Branch
currentPC
controlDe
instr
branchout
rsFF
rtFF
rs
rt
Register File
writeRegEnWB
writeRegNumWB
clear
clearEx
En
stallEx
controlEx
instrEx
branchoutEx
Execute
imm
shift amount
ALUControl
rs
rt
ALU
Shifter
Low
High
Multiplier
Divider
Memory Interface
stallRqEx
Data Cache
dStall
memreaddata
clear
clearMA
En
stallMA
controlMA
instrMA
executeOutMA
Memory Access
writeRegEnMA
writeRegNumMA
clear
clearWB
En
stallWB
controlWB
instrWB
MAOutWB
Write Back
writeRegDataWB
writeRegEnWB
writeRegNumWB
iStall
dStall
stallDe
clearDe
controlDe
instrDe
stallEx
clearEx
controlEx
instrEx
stallRqEx
stallMA
clearMA
writeRegEnMA
writeRegNumMA
stallWB
clearWB
writeRegEnWB
writeRegNumWB
Stall Logic

MIPS — Defines file 4

```
`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 CONTROL_WIDTH         30:0
`define CONTROL_ALUCONTROL    12:8
`define CONTROL_BRANCHTYPE    7:5
`define CONTROL_WRITEREGNUM   4:0

`define BR_NONE          3'b000
`define BR_LTZ           3'b001
`define BR_GEZ           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_RIGHT     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_MTHI         5'b0001_1
`define ALU_MFLO         5'b0010_0
`define ALU_MTLO         5'b0010_1
`define ALU_MULT         5'b0011_0    // unsigned version needed
`define ALU_DIV          5'b0100_0    // unsigned version needed
`define ALU_ADD          5'b0101_0    // unsigned version needed
`define ALU_SUB          5'b0110_0    // unsigned version needed
`define ALU_AND          5'b0111_0
`define ALU_OR           5'b0111_1
`define ALU_XOR          5'b1000_0
`define ALU_NOR          5'b1000_1
`define ALU_SLT          5'b1001_0    // unsigned version needed
`define ALU_LUI          5'b1010_0
// 1010_1 is unused
`define ALU_MUL          5'b1011_0    // unsigned version needed
`define ALU_SL           5'b1100_0    // unsigned version needed
`define ALU_SR           5'b1101_0    // unsigned version needed
`define ALU_SLV          5'b1110_0    // unsigned version needed
`define ALU_SRV          5'b1111_0    // unsigned version needed


`define CONTROL_IRQRETURN    13
`define CONTROL_MEMR         14
`define CONTROL_MEML         15
`define CONTROL_MEM8         16
`define CONTROL_MEM16        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_USEIMM       25
`define CONTROL_REGWRITE     26
`define CONTROL_COPREAD      27
`define CONTROL_COPWRITE     28
`define CONTROL_DCACHEFLUSH  29
`define CONTROL_ICACHEFLUSH  30
```

MIPS — Top Level Module Part 1 5

```
//////////////////////////////////////////////////////////
// MIPS SOFT PROCESSOR
// TOP LEVEL
//
// 5-stage pipeline:
// Fetch -> Decode -> Execute -> MemoryAccess -> RegisterWriteBack
//////////////////////////////////////////////////////////

`include "tiger_defines.v"

module tiger_tiger(
    input clk,              // clock signal
    input reset,            // reset signal
    input iStall,           // instruction stall signal
    input dStall,           // data stall signal

    output iCacheFlush,
    output dCacheFlush,

    input canICacheFlush,
    input canDCacheFlush,

    input irq,              // interrupt request signal
    input [5:0] irqNumber,

    output [31:0] pc,       // program counter
    input [31:0] instrF,    // fetched instruction

    output memwrite, memread, mem16, mem8, memzerofill,  // memory access mode outputs
    output [31:0] memaddress, memwritedata,              // memory address and data outputs
    input [31:0] memreaddata,                            // memory data input
    input memCanRead,
    input memCanWrite
);

    wire stallRqEx;
    wire exception;

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

    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] executeoutMA;
    wire [31:0] branchoutMA;
    wire [1:0] bottomaddressMA;

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

    wire [31:0] instrWB;
    wire [`CONTROL_WIDTH] controlWB;
    wire [31:0] MAOutWB;
    wire [31:0] branchoutWB;
    wire [31:0] writeRegDataWB;
    wire [`REGNUM_WIDTH] writeRegNumWB;
    wire writeRegEnWB;
    wire writeRegEnCopWB;
```

MIPS — Top Level Module Part 2

```
tiger_stalllogic sl(
        .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 fe(
        .clk(clk),
        .reset(reset),
        .stall(stallDe),
        .clear(clearDe),

        .instr(instrF),

        .instrDE(instrDe)
    );

    // decode stage
    tiger_decode de(
        .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),
        .rsEx(rsEx),
        .rtEx(rtEx),
        .CPOutEx(CPOutEx),

        .branchoutEx(branchoutEx),

        .nextpc(pc)
    );
```

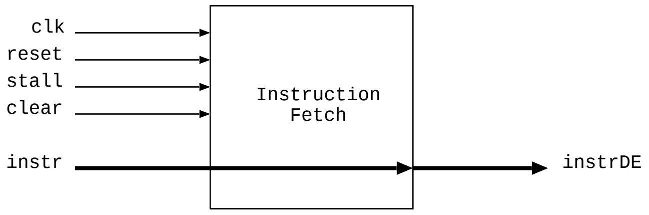
MIPS — Top Level Module Part 3

```
// feed forward path for first operand
    wire [31:0] rsExFF;
    tiger_ff ff_for_rs(
        .regnum(instrEx[25:21]),
        .writereg1(writeRegNumMA),
        .writereg2(writeRegNumWB),
        .writeregen1(writeRegEnMA),
        .writeregen2(writeRegEnWB),
        .regdata(rsEx),
        .writeregdata1(writeRegDataMA),
        .writeregdata2(writeRegDataWB),
        .out(rsExFF)
    );

    // feed forward path for second operand
    wire [31:0] rtExFF;
    tiger_ff ff_for_rt(
        .regnum(instrEx[20:16]),
        .writereg1(writeRegNumMA),
        .writereg2(writeRegNumWB),
        .writeregen1(writeRegEnMA),
        .writeregen2(writeRegEnWB),
        .regdata(rtEx),
        .writeregdata1(writeRegDataMA),
        .writeregdata2(writeRegDataWB),
        .out(rtExFF)
    );

    // excecute stage
    tiger_execute ex(
        .clk(clk),
        .reset(reset),
        .stall(stallMA),
        .clear(clearMA),

        .instr(instrEx),
        .control(controlEx),
        .rs(rsExFF),
        .rt(rtExFF),
        .branchout(branchoutEx),
        .CPOut(CPOutEx),

        .instrMA(instrMA),
        .controlMA(controlMA),
        .executeoutMA(executeoutMA),
        .branchoutMA(branchoutMA),
        //.bottomaddressMA(bottomaddressMA),

        .stallRq(stallRqEx),

        .memread(memread),
        .mem16(mem16),
        .mem8(mem8),
        .memwrite(memwrite),
        .memaddress(memaddress),
        .memwritedata(memwritedata),
        .memCanRead(memCanRead),
        .memCanWrite(memCanWrite),
        .iCacheFlush(iCacheFlush),
        .dCacheFlush(dCacheFlush),
        .canICacheFlush(canICacheFlush),
        .canDCacheFlush(canDCacheFlush)
    );
```

MIPS — Top Level Module Part 4 8

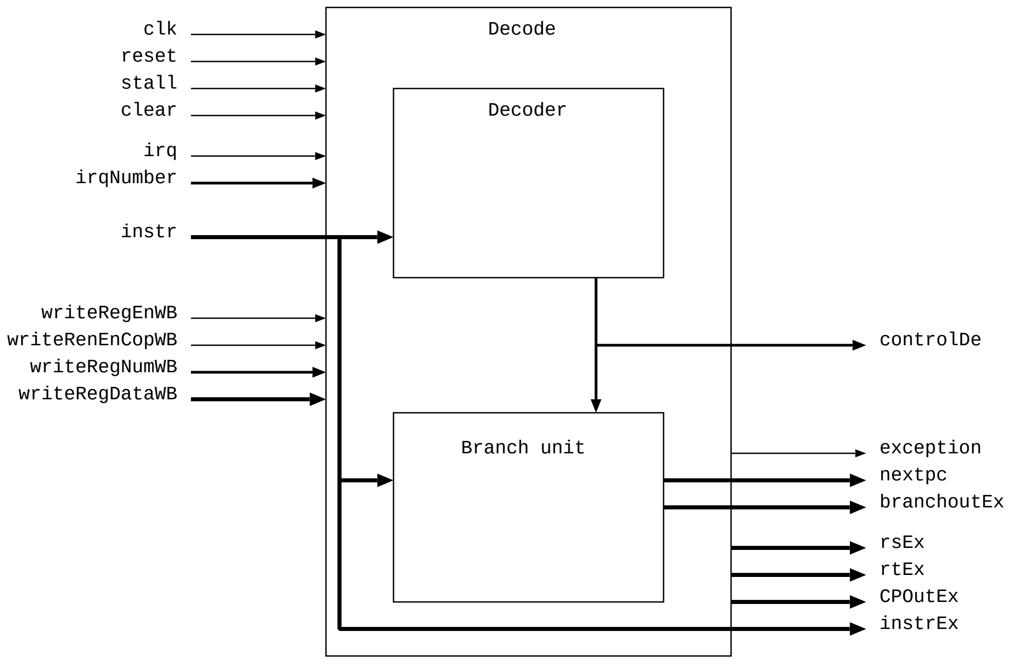
```
    // memory access stage
    tiger_memoryaccess ma(
        .clk(clk),
        .reset(reset),
        .clear(clearWB),
        .stall(stallWB),

        //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),
        .MAOutWB(MAOutWB),
        .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)
        .writeRegEnCop(writeRegEnCopMA),
        .writeRegNum(writeRegNumMA), //Which register we will write to
        .writeRegData(writeRegDataMA) //What we would write to the register
    );

    // writeback stage
    tiger_writeback wb(
        .clk(clk),

        //Pipeline registers in
        .instr(instrWB), //instruction
        .control(controlWB), //control signal
        .branchout(branchoutWB), //PC value for use with b w/ link
        .MAOut(MAOutWB), //Output of the memory access stage

        //Register write control
        .writeRegEn(writeRegEnWB), //True if we want to write to a register
        .writeRegEnCop(writeRegEnCopWB),
        .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
```

MIPS — Instruction Decode



```
`include "tiger_defines.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,

    input [31:0] instr,                     // current instruction

    input writeRegEnWB,                     // True if write back wants to write to a register
    input writeRegEnCopWB,                  // Ture if write back wants to write a coprocessor register
    input [`REGNUM_WIDTH] writeRegNumWB,    // Register write back wants to write to
    input [31:0] writeRegDataWB,            // Data write back wants to write

    output exception,

    output [`CONTROL_WIDTH] controlDe,

    output reg [31:0] instrEx,              // output containing the current instruction
    output reg [`CONTROL_WIDTH] controlEx,  // output for the control signals
    output reg [31:0] rsEx,                 // first of 2 operand outputs
    output reg [31:0] rtEx,                 // second of 2 operand outputs
    output reg [31:0] CPOutEx,              // 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)
    );
```

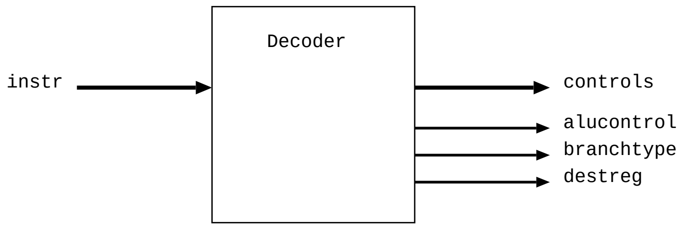
MIPS — Instruction Decode Implementation Part 1

```
// set up 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] rf[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)
begin
    // 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'b0 && writeRegEnWB)
        rf[writeRegNumWB] <= writeRegDataWB;
    if(writeRegEnCopWB) begin
        case(writeRegNumWB)
            //5'b0_0000: cause <= writeRegDataWB;
            5'b0_0001: status <= writeRegDataWB;
            //5'b0_0010: epc <= writeRegDataWB;
        endcase
    end
```

MIPS — Instruction Decode Implementation Part 2

```
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'h0078_0000;
        end else if( stall ) begin
            //Stall instrEx
            //Stall controlEx
            //Stall rsEx
            //Stall rtEx
            //Stall CPOutEx
        end else if (clear || (exception && !stall)) begin
            instrEx     <= 0;
            controlEx   <= 0;
            rsEx        <= 0;
            rtEx        <= 0;
            CPOutEx     <= 0;
        end else begin
            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'bx_xxxx;
        end
    end

    tiger_branch b(
        .clk(clk),
        .reset(reset),
        .stall(stall || clear),

        .exception(exception),

        .instr(instr),
        .control(control),
        .rs(rsFF),
        .rt(rtFF),

        .branchout(branchoutEx),
        .epc(epcDe),
        .branchDelay(branchDelay),

        .nextpc(nextpc)
    );

endmodule
```

MIPS — Instruction Decoder

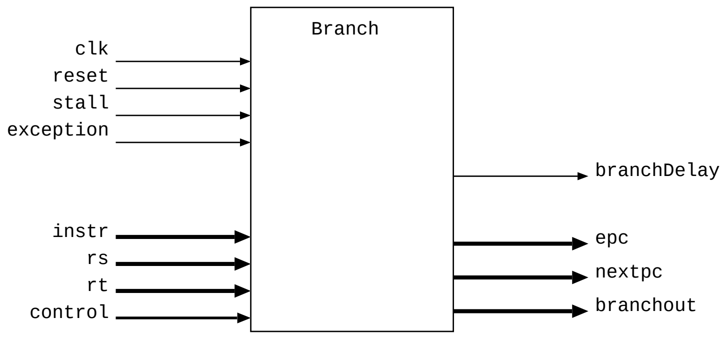


```
`include "tiger_defines.v"

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
                   : (controls[12] || controls[14]) ? instr[20:16]
                   :                                    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'b0001) begin
            controls    <= 16'b0000_1000_0000_0000;
            alucontrol  <= `ALU_NONE;
            branchtype  <= {op[1] | op[0], ~(op[1] ^ op[0]), ~op[0]};
        // BLTZ, BGEZ
        end else if (op == 6'b00_0001) begin
            controls    <= 16'b0000_1000_0000_0000;
            alucontrol  <= `ALU_NONE;
            branchtype  <= {1'b0, instr[16], ~instr[16]};
        end else begin
            branchtype  <= `BR_NONE;
            // R-type instruction
            if (op == 6'b00_0000) begin
                controls    <= (funct == 6'b00_1000) ? 16'b0000_0000_1000_0000
                             : (funct == 6'b00_1001) ? 16'b0010_0000_1010_0000
                             :                         16'b0010_0000_0000_0000;
                // ADD, ADD UNSIGNED, SUB, SUB UNSIGNED
                if (funct[5:2] == 4'b1000) begin
                    alucontrol  <= {2'b01, funct[1], ~funct[1], funct[0]};
                end else if (funct[5:2] == 4'b0110) begin
                    alucontrol  <= {1'b0, funct[1], {2{~funct[1]}}, funct[0]};
                end else if (funct[5:2] == 4'b1001) begin
                    alucontrol  <= {funct[1], {3{~funct[1]}}, funct[0]};
                end else if (funct[5:2] == 4'b0100) begin
                    alucontrol  <= {2'b00, funct[1], ~funct[1], funct[0]};
                end else if (funct[5:3] == 3'b000 && (funct[1] || ~funct[0])) begin
                    alucontrol  <= {2'b11, funct[2:1], ~funct[0]};
                end else if (funct[5:1] == 5'b10101) begin
                    alucontrol  <= {4'b1001, funct[0]};
                end else begin
                    alucontrol  <= `ALU_NONE;
                end
```

MIPS — Instruction Decoder Implementation

```
            // ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI
            end else if (op[5:3] == 3'b001) begin
                alucontrol  <= {op[1], ~op[1], op[2] & (op[0] | ~op[1]), ~op[2] | ~op[1], op[0] & (~op[2] | ~op[1])};
                controls    <= {2'b0, {2{1'b1}}, {5{1'b0}}, op[2] & (~op[1] | ~op[0]), {6{1'b0}}};
            // LB, LH, LWL, LW, LBU, LHU, LWR
            end else if (op[5:3] == 3'b100 && (~op[2] | ~op[1] | ~op[0])) begin
                alucontrol  <= ‘ALU_ADD;
                controls    <= {2'b0, 7'b110_0100, op[2] & ~op[1], 1'b0, ~op[1] & op[0], ~op[1] & ~op[0], ~op[2] & op[1] & ~op[0], op[2] & op[1], 1'b0};
            // SB, SH, SW
            end else if (op[5:2] == 4'b1010 && (~op[1] | op[0])) begin
                alucontrol  <= ‘ALU_ADD;
                controls    <= {2'b0, 9'b0_1010_0000, ~op[1] & op[0], ~op[1] & ~op[0], 3'b000};
            // J, JAL
            end else if (op[5:1] == 5'b0_0001) begin
                alucontrol  <= ‘ALU_NONE;
                controls    <= {2'b0, op[0], 7'b000_0100, op[0], {5{1'b0}}};
            // ERET, JERET
            //end else if (op[5:1] == 5'b0_1000) begin
            //  alucontrol  <= ‘ALU_NONE;
            //  controls    <= op[0]                                          ? 16'b0000_0001_0000_0001
            //                : {instr[25:21], instr[5:0]} == 11'b100_0001_1000  ? 16'b0000_0000_0000_0001
            //                :                                                   16'b0000_0000_0000_0000;
            // MUL
            end else if (op == 6'b01_1100) begin
                alucontrol  <= (funct == 6'b00_0010) ? ‘ALU_MUL                  : ‘ALU_NONE;
                controls    <= (funct == 6'b00_0010) ? 16'b0010_0000_0000_0000 : 16'b0000_0000_0000_0000;
            // MFC0, MTC0
            end else if (op == 6'b01_0000) begin
                alucontrol <= ‘ALU_NONE;
                controls   <= {instr[23], {2{~instr[23]}}, 13'b0_0000_0000};
            end else begin
                alucontrol  <= ‘ALU_NONE;
                controls    <= 16'b0000_0000_0000_0000;
            end
        end
    end
endmodule
```

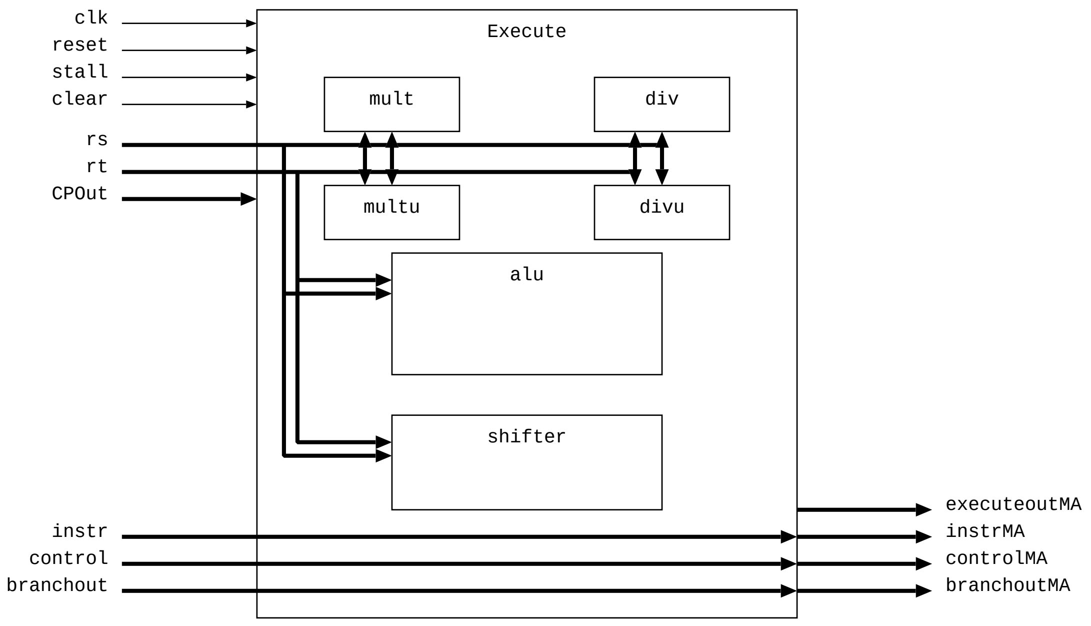
MIPS — Branch Unit



MIPS — Branch Unit Implementation 16

```
`include "tiger_defines.v"

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 [31:0]epc,
    output reg branchDelay,

    output [31:0] nextpc          // address of the next instruction to load
);

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]      // take the branch if:
 &&
 (  (branchtype == `BR_LTZ && rs[31])            // branch if < 0 instruction and the MSB of the first operand is set
 || (branchtype == `BR_GEZ && !rs[31])           // branch if >= 0 instruction and the MSB of the first operand is not set
 || (branchtype == `BR_EQ  && rs == rt)          // branch if equal isntruction and both operands are equal to each other
 || (branchtype == `BR_NE  && rs != rt)          // branch if not equal instruction and both operands are not equal
 || (branchtype == `BR_LEZ && rs[31])            // branch if <= 0 instruction and either the MSB is set
 || (branchtype == `BR_LEZ && rs == 0)           // or the register equals 0
 || (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
 );

// sign extend the immediate constant (used for a relative jump here) and shift 2 places to the left
wire [31:0] signimmsh = {{14{instr[15]}}, instr[15:0], 2'b00};
wire [31:0] pcplus4 = currentpc + 4;

assign nextpc = reset                      ?  BOOT_ADDR                   // reset the start address
            :   stall                      ?  currentpc                   // stalled so do not change the current address
            :   exception                  ?  EXCEPTION_HANDLER_ADDR
            :   takebranch                 ?  currentpc + signimmsh       // relative branch
            :   control[`CONTROL_JUMP]     ?  {currentpc[31:28], instr[25:0], 2'b00}  // immediate jump
            :   control[`CONTROL_REGJUMP]  ?  rs                          // 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 occured
assign epc = branchDelay ? branchout - 8
                 : currentpc - 4;

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
end
endmodule
```

MIPS — Execute Unit



MIPS — Execute Unit Implementation Part 1 18

```
`include "tiger_defines.v"

module tiger_execute(
    input clk,                                  // clock signal
    input reset,                                // reset signal
    input stall,                                // stall signal
    input clear,                                // clear 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 stallRq,                         // do we wish to stall the pipeline

    output memread,mem16,mem8,memwrite, // what type of memory access do we require
    output [31:0] memaddress,memwritedata,      // address in memory to write to + the data to write
    output iCacheFlush, dCacheFlush,
    input memCanRead, memCanWrite,
    input canICacheFlush, canDCacheFlush
);
    // sign-extend the 2 operands and multiply them together to form a 64 bit number
    //wire [63:0] mult = {{32{rs[31] && !control[`CONTROL_ALUCONTROL_UNSIGNED]}}, rs}
    //               * {{32{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),
        .remain(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),
        .remain(divuHI)
    );

    // alu unit
    wire [31:0] aluout;
    tiger_alu alu(
        .srca(rs), // first operand is always rs
        // second operand may be an immediate constant (so we sign-extend to 32 bits) or a second register
        .srcb(control[`CONTROL_USEIMM] ? {{16{instr[15] && !control[`CONTROL_ZEROFILL]}}, instr[15:0]} : rt),
        .alucontrol(control[`CONTROL_ALUCONTROL]),
        .aluout(aluout)
    );
```

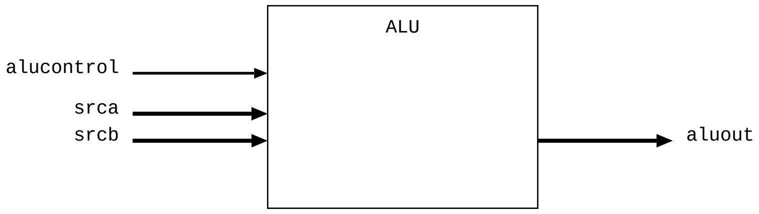
MIPS — Execute Unit Implementation Part 2

```
// shifter unit
wire [31:0] shiftout;
tiger_shifter shift(
    .src(rt),
    .amt(control[‘CONTROL_ALUCONTROL_VARIABLE] ? rs[4:0] : instr[10:6]),
    .dir(control[‘CONTROL_ALUCONTROL_RIGHT]),
    .alusigned(!control[‘CONTROL_ALUCONTROL_UNSIGNED]),
    .shifted(shiftout)
);

// set the countdown
// MTHI, MTLO and MULT only require a single cycle, but  division requires 12 cycles to complete
wire [3:0] newcountdown1=reset ||
                          control[‘CONTROL_ALUCONTROL] == ‘ALU_MTHI ||
                          control[‘CONTROL_ALUCONTROL] == ‘ALU_MTLO ||
                          control[‘CONTROL_ALUCONTROL] == ‘ALU_MULT ||
                          control[‘CONTROL_ALUCONTROL] == (‘ALU_MULT | ‘ALU_UNSIGNED)?   4'd0
                         :control[‘CONTROL_ALUCONTROL] == ‘ALU_DIV ||
                          control[‘CONTROL_ALUCONTROL] == (‘ALU_DIV | ‘ALU_UNSIGNED) ?   4'd11
                         :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_MTLO      ?  rs          // move to LO
                    :control[‘CONTROL_ALUCONTROL] == ‘ALU_MULT      ?  mults[31:0] // low 32 bits of multiplication
                    :control[‘CONTROL_ALUCONTROL] == (‘ALU_MULT | ‘ALU_UNSIGNED) ? multu[31:0]
                    :countdown==1                                   ?  (!source ? divLO : divuLO)
                    :                                                  low;

// the quotient of the division - used only when the pipeline is stalled
wire [31:0] newlow2= reset                                        ?  32'd0
                    :countdown==1                                   ?  (!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      ?  mults[63:32] // high 32 bits of the multiplication
                    :control[‘CONTROL_ALUCONTROL] == (‘ALU_MULT | ‘ALU_UNSIGNED) ? multu[63:32]
                    :countdown==1                                   ?  (!source ? divHI : divuHI)
                    :                                                  high;

// the remainder of the division - used only when the pipeline is stalled
wire [31:0] newhigh2=reset                                        ?  32'd0
                    :countdown==1                                   ?  (!source ? divHI : divuHI)
                    :                                                  high;
```

MIPS — Execute Unit Implementation Part 3 20

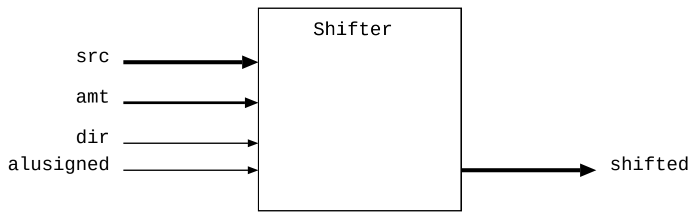
```
always @(posedge clk)
begin
    countdown <= !stall ? newcountdown1 : newcountdown2;
    low <= !stall ? newlow1 : newlow2;
    high <= !stall ? newhigh1 : newhigh2;

    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;
        executeoutMA <=  control['CONTROL_ALUCONTROL] == 'ALU_MUL        ?   mults[31:0] // lower 32 bits of the multiplication
                        :&control['CONTROL_ALUCONTROL_SHIFT]             ?   shiftout    // output of the shift unit
                        :control['CONTROL_ALUCONTROL] == 'ALU_MFHI       ?   high        // MFHI => contents of the HI register
                        :control['CONTROL_ALUCONTROL] == 'ALU_MFLO       ?   low         // MHLO => contents of the LO register
                        :control['CONTROL_MEML]||control['CONTROL_MEMR]  ?   rt          // second operand
                        :control['CONTROL_LINK]                          ?   branchout   // return address for link operation
                        :control['CONTROL_COPREAD]                       ?   CPOut
                        :control['CONTROL_COPWRITE]                      ?   rt
                        :                                                    aluout;     // otherwise give the output of the ALU
        branchoutMA <= branchout;
        //bottomaddressMA <= aluout[1:0];

        if( control['CONTROL_ALUCONTROL] == 'ALU_DIV ) begin
            source <= 0; // signed division flag
        end else if( control['CONTROL_ALUCONTROL] == ('ALU_DIV | 'ALU_UNSIGNED) ) begin
            source <= 1; // unsigned division flag
        end
    end
end

// we stall the pipeline if
assign stallRq =
(
    // the current instruction is MFHI or MFLO and the countdown has yet to reach 0
    // (i.e. the contents of the registers are not valid at this point in time)
    (control['CONTROL_ALUCONTROL] == 'ALU_MFHI || control['CONTROL_ALUCONTROL] == 'ALU_MFLO)
    && countdown > 0
)
|| // OR
(!memCanRead && control['CONTROL_MEMREAD]) //Can't read currently and we want to read
|| // OR
(!memCanWrite && control['CONTROL_MEMWRITE]) //Can't write currently and we want to write
||
(!canICacheFlush && control['CONTROL_ICACHEFLUSH])
||
(!canDCacheFlush && control['CONTROL_DCACHEFLUSH]);


// assign outputs appropriately
assign memread       = clear || !memCanRead ? 1'b0 : control['CONTROL_MEMREAD];
assign memwrite      = clear || !memCanWrite ? 1'b0 : control['CONTROL_MEMWRITE];
assign mem16         = clear ? 1'b0 : control['CONTROL_MEM16];
assign mem8          = clear ? 1'b0 : control['CONTROL_MEM8];
assign memaddress    = clear ? 0 : aluout;
assign memwritedata = clear ? 0 : rt;
assign iCacheFlush  = clear || !canICacheFlush ? 1'b0 : control['CONTROL_ICACHEFLUSH];
assign dCacheFlush  = clear || !canDCacheFlush ? 1'b0 : control['CONTROL_DCACHEFLUSH];
endmodule
```

MIPS — ALU



```
`include "tiger_defines.v"

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;

    assign aluout  =  alucontrol == `ALU_ADD || alucontrol == (`ALU_ADD | `ALU_UNSIGNED)  ?  srca + srcb
                   :  alucontrol == `ALU_SUB || alucontrol == (`ALU_SUB | `ALU_UNSIGNED)  ?  srca - srcb
                   :  alucontrol == `ALU_AND                                              ?  srca & srcb
                   :  alucontrol == `ALU_OR                                               ?  srca | srcb
                   :  alucontrol == `ALU_XOR                                              ?  srca ^ srcb
                   :  alucontrol == `ALU_NOR                                              ?  ~(srca | srcb)
                   :  alucontrol == `ALU_SLT                                              ?  srca < srcb
                   :  alucontrol == (`ALU_SLT | `ALU_UNSIGNED)                            ?  srcau < srcbu
                   :  alucontrol == `ALU_LUI                                              ?  {srcb[15:0], 16'b0}
                   :                                                                         32'hxxxx_xxxx;
endmodule
```

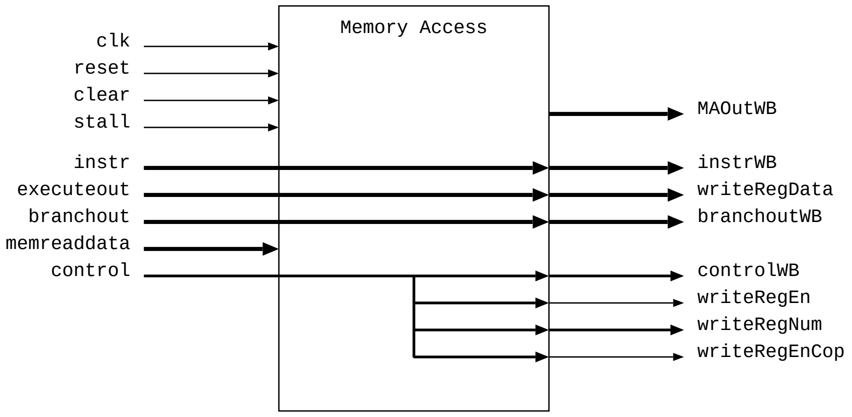
MIPS — Shifter



MIPS — Shifter Implementation

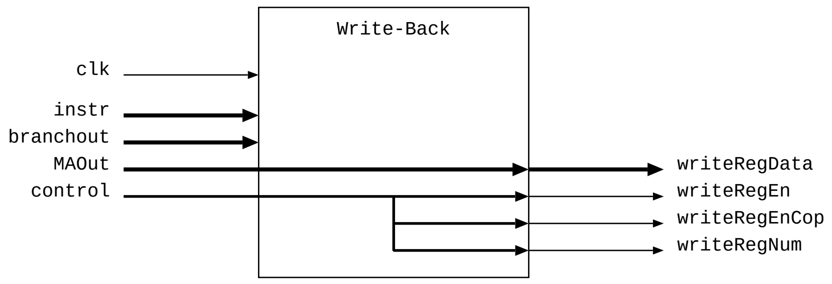
```
module tiger_shifter(
        input [31:0] src,          // source data
        input [4:0] amt,           // number of bits to shift by
        input dir,                 // direction to shift (0 = left; 1 = right)
        input alusigned,           // signed shift? 0 = unsigned; 1 = signed
        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 right16 = amt[4] ? {{16{fillbit}} , src[31:16]}      : src;
  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;

  // do a left shift by shifting 0–5 times
  wire [31:0] left16;
  wire [31:0] left8;
  wire [31:0] left4;
  wire [31:0] left2;
  wire [31:0] left1;
  wire [31:0] left;

  assign left16 = amt[4] ? {src[15:0]     , 16'b0} : src;
  assign left8  = amt[3] ? {left16[23:0]  , 8'b0}  : left16;
  assign left4  = amt[2] ? {left8[27:0]   , 4'b0}  : left8;
  assign left2  = amt[1] ? {left4[29:0]   , 2'b0}  : left4;
  assign left1  = amt[0] ? {left2[30:0]   , 1'b0}  : left2;

  assign left = left1;

  // select the correct shift output
  assign shifted = dir ? right : left;
endmodule
```

MIPS — Memory Access



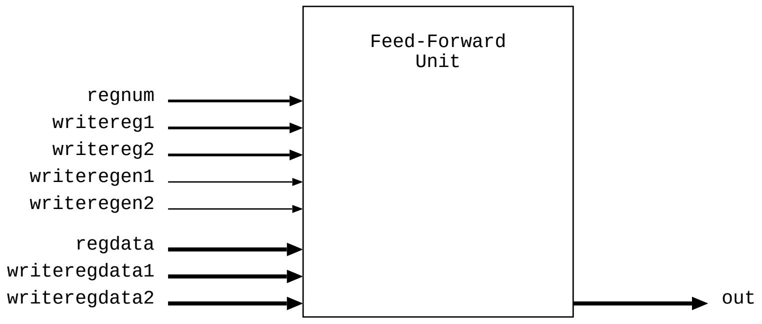
```
‘include ”tiger_defines.v”

module tiger_memoryaccess(
    input clk,                          // clock signal
    input reset,
    input clear,
    input stall,

    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,

    //input [1:0] bottomaddress,            // bottom bits of the address
    input [31:0] memreaddata,           // data read from memory

    //true if we wish to write to the register given in writeRegNum
    //the write will actually be done in the following WB stage
    output writeRegEn,
    //the number of the register we’re going write to (in the WB stage)
    output [‘REGNUM_WIDTH] writeRegNum,

    //true if we wish to write to the coprocessor register given in writeRegNum
    //the write will actually be done in the following WB stage
    output writeRegEnCop,

    //if we are going to write to a register, what data we would write
    //note that this is the output from the execute stage
    //if we are reading from memory, the memory read data will
    //be put into the next pipeline register at the next clock
    output [31:0] writeRegData
);

    assign writeRegEn = control[‘CONTROL_REGWRITE];
    assign writeRegEnCop = control[‘CONTROL_COPWRITE];
    assign writeRegNum = control[‘CONTROL_WRITEREGNUM];
    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_MEMREAD] ?
                    (!control[‘CONTROL_ZEROFILL] && control[‘CONTROL_MEM8] ? {{24{memData[7]}}, memData[7 : 0]}
                     : !control[‘CONTROL_ZEROFILL] && control[‘CONTROL_MEM16] ? {{16{memData[15]}}, memData[15 : 0]}
                     : memData)
                : executeout;
            branchoutWB <= branchout;
            instrWB <= instr;
        end
    end
endmodule
```

MIPS — Write Back



```
`include "tiger_defines.v"

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

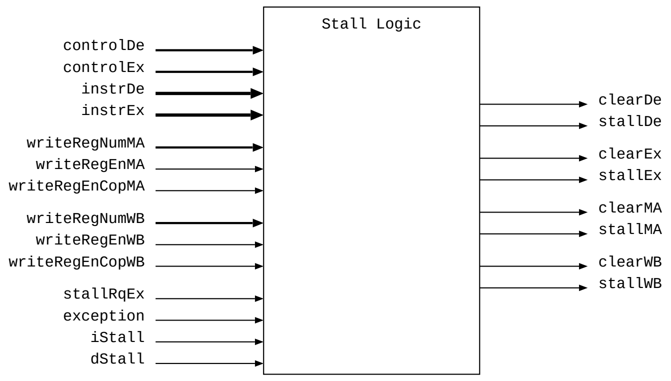


```
module tiger_ff(
    input [4:0] regnum,          // register number that we are writing to
                writereg1,       // register number the execute unit wishes to write to
                writereg2,       // register number the MWB unit wishes to write to
    input       writeregen1,     // enable WB for the execute unit
                writeregen2,     // enable WB for the MWB unit
    input [31:0]regdata,         // current contents of the register
                writeregdata1,   // data the execute unit wishes to write to the register
                writeregdata2,   // data the MWB unit wishes to write to the register
    output [31:0]out
);
    ///////////////////////////////////////////////////////////////////////
    // 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 == writereg1) && writeregen1 ? writeregdata1 :
            (regnum == writereg2) && writeregen2 ? writeregdata2 : regdata;
*/

    // check to see if regnum == writereg2 and writeregen2 is enabled
    wire en2;
    tiger_ff_compare c2(regnum,writereg2,writeregen2,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 ? writeregdata2 : regdata;

    // check to see if regnum == writereg1 and writeregen1 is enabled
    wire en1;
    tiger_ff_compare c1(regnum,writereg1,writeregen1,en1);
    // if it is enabled, then write the data to the register, otherwise use the result of the MWB checker above
    wire [31:0] muxedin1=en1 ? writeregdata1 : 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 multiplexor
    and a[31:0](out,muxedin1,notzero);
endmodule
```

MIPS — Feed-Forward Paths Implementation

```
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 synthesises to a better circuit on the FPGA
    //////////////////////////////////////////////////////////////

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

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

    // check to see if bits [1:0] of the register numbers are equal
    xnor e0(eq[0],regnum[0],writereg[0]);   and a0(anded[0],eq[0],eq[1]);
    xnor e1(eq[1],regnum[1],writereg[1]);

    // check to see if bits [3:2] of the register numbers are equal
    xnor e2(eq[2],regnum[2],writereg[2]);   and a1(anded[1],eq[2],eq[3]);
    xnor e3(eq[3],regnum[3],writereg[3]);

    // check to see if bit [4] of the register numbers is equal
    xnor e4(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_stalllogic(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 stallRqEx,
                        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 needStallDe;
    wire needStallEx;
    wire needStallMA;
    wire needStallWB;

    assign clearDe = exception && !stallDe;
    assign stallDe = needStallDe || stallEx || iStall;

    assign clearEx = (needStallDe || iStall) && !stallEx;
    assign stallEx = needStallEx || stallMA;

    assign clearMA = needStallEx && !stallMA;
    assign stallMA = needStallMA || stallWB;

    assign clearWB = needStallMA && !stallWB;
    assign stallWB = needStallWB;
```

MIPS — Stall Logic Implementation Part 1

```
//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_REGJUMP];

//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
wire rsInE = instrDe[25:21] == controlEx[‘CONTROL_WRITEREGNUM] && instrDe[25:21] != 0;
wire rtInE = instrDe[20:16] == controlEx[‘CONTROL_WRITEREGNUM] && instrDe[20:16] != 0;

//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 rsInMA = instrDe[25:21] == writeRegNumMA && instrDe[25:21] != 0;
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]) || (rtInMA && 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)
    controlEx[‘CONTROL_MEMREAD] && ((rsInE && needsRsAndNotBranch(controlDe))
                                    || (rtInE && needsRtAndNotBranch(controlDe)))
    || //Or
    //If in decode there's a coprocessor read instruction and we're writing to the coprocessor
    //futher up the pipeline (not always a hazard, and could have been solved by forwarding,
    //however as we don't read or write from the coprocessor that 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 negligable).
    controlDe[‘CONTROL_COPREAD] && (controlEx[‘CONTROL_COPWRITE]
                                    || writeRegEnCopMA
                                    || writeRegEnCopWB);
```

MIPS — Stall Logic Implementation Part 2 30

```
    //We need a stall in the execute stage if the execute stage requests one
    assign needStallEx = stallRqEx;
    //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
            (
                (
                    //The instruction in the execute stage needs it for rs
                    instrEx[25:21] == writeRegNumWB
                    && needsRsAndNotBranch(controlEx)
                )
                || //Or
                (
                    //The instruction in the execute stage need it for rt
                    instrEx[20:16] == writeRegNumWB
                    && needsRtAndNotBranch(controlEx)
                )
            )
            && //And
            //Either the execute or memory access stage needs a stall
            //(if MA needs a stall then execute will also be stalled,
            //we can't just use stallEx directly as this creates a loop
            //that will cause a stall that never ends)
            (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;
        input [`CONTROL_WIDTH] control;
        begin
            needsRsAndNotBranch = !control[`CONTROL_IRQRETURN]
            && !control[`CONTROL_JUMP]
            && !control[`CONTROL_REGJUMP]
            && !control[`CONTROL_BRANCH];
        end
    endfunction

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