Computer Design — Lecture 10 1

Overview of this lecture

- Review a SystemVerilog implementation of a subset of a MIPS processor
- Compare this SystemVerilog design with the Java design from the previous lecture

☞ Many thanks to Emma Burrows who prepared much of this design whilst working as a summer intern in the Computer Architecture group

MIPS — Simulation Test Bench 2

```
module testBench();

   logic clk, rst;

   initial begin
      clk = 0;
      rst = 1;
   #5 clk = 1;
   #5 clk = 0;
      rst = 0;
      forever #5 clk <= ~clk;
   end

   proc2_top p(.clk, .rst);

endmodule
```

MIPS — Instruction ROM 3

```
module instruction_ROM( input [29:0] instAddr,
        output logic [31:0] instValOut);

  // instruction ROM into the memory based on the diassembly below
  always_comb case(instAddr[29:0]<<2)
    32'h00000000: instValOut<=32'h8c040080;  //  lw      a0,128(zero)
    32'h00000004: instValOut<=32'h34090001;  //  li      t1,0x1
    32'h00000008: instValOut<=32'h340a0001;  //  li      t2,0x1
    32'h0000000c: instValOut<=32'h2084ffff;  //  addi    a0,a0,-1
    32'h00000010: instValOut<=32'h18800005;  //  blez    a0,28 <finished>
    32'h00000014: instValOut<=32'h00000000;  //  nop
    32'h00000018: instValOut<=32'h012a4820;  //  add     t1,t1,t2
    32'h0000001c: instValOut<=32'h012a5022;  //  sub     t2,t1,t2
    32'h00000020: instValOut<=32'h08000003;  //  j       c <fibloop>
    32'h00000024: instValOut<=32'h00000000;  //  nop
    32'h00000028: instValOut<=32'hac090080;  //  sw      t1,128(zero)
    32'h0000002c: instValOut<=32'h1000ffff;  //  b       2c <terminate>
    32'h00000030: instValOut<=32'h00000000;  //  nop
    default: instValOut<=32'h00000000;
  endcase

  // note: the file holding the ROM state could be read in for
  // simulation purposes using the $readmemh function rather than
  // use the approach above
endmodule
```

MIPS — Processor Shared Definitions 4

```
// enumeration type for the ALU functions
typedef enum { ALU_f_NOP, ALU_f_ERROR, ALU_f_ADD,
               ALU_f_SUB, ALU_f_OR, ALU_f_SLL } ALU_f ;

// enumeration type for the memory functions
typedef enum { MEM_f_NONE, MEM_f_LW, MEM_f_SW } MEM_f;

// enumeration type for the operand B source
typedef enum { OPB_s_REG_RT, OPB_s_ZEROIMM, OPB_s_SIGNIMM } OPB_s;

// enumeration type for type of jump/branch
typedef enum { BR_NONE, BR_J, BR_JR, BR_JAL, BR_B, BR_JALR } BRANCH_TYPE;

// enumeration type for branch condition
typedef enum { NO, UNCONDITIONAL, EQ, LEZ } BRANCH_COND;

// define boolean type as enum
typedef enum {FALSE=0, TRUE=1} BOOLEAN;

// holds control data for each pipeline stage
typedef struct packed{
        ALU_f           alu_f;  //ALU function
        BOOLEAN         wben;   // write back enable
        MEM_f           mem_f;  // memory function
        OPB_s           opb_s;  // operand B source
        BRANCH_TYPE     branch_type;  //branch type
        BRANCH_COND branch_cond;  //branch condition
} DECODEROM;

// machine word type
typedef bit [31:0] wordT;

// register number type
typedef bit [4:0] regNumT;

// register number and value type
typedef struct packed {
        regNumT rn;
        wordT val;
} regT;

// context present – indicates whether a pipeline stage should perform
// operation with its context data
typedef enum { presentYes = 1, presentNo = 0 } presenceT;

// decoded instruction into constituent parts
typedef struct packed{
        DECODEROM       func;  // computed decode information
        regNumT         rs, rt, rd; // source and destination registers
        wordT           zeroimm; // extracted zero extended immediate
        wordT           signimm; // extracted sign extended immediate
        bit [27:0]      addr;
        bit [4:0]       shamt;
        bit [5:0]       funct;
} DECODED;
```

MIPS — Top-level processor 1 of 2 5

```
`include "proc2_defsinc.v"

//////////////////////////////////////////////////////////////////////
// TOP LEVEL MODULE
// Role:
//      Initialise the processor
//      Instantiate the pipeline stages
//      Pass the appropriate control and data signals through the pipeline
//      stages
//////////////////////////////////////////////////////////////////////

module proc2_top(
        input clk,
        input rst
);

logic[31:0] cycleNum;    // used to count clock cycles
wordT prevPCVal;         // used to stop the processor

wordT instructionVal;    // used to pass the hex value of the instruction to be
                         // decoded from the instruction fetch stage to the decode

// pass the operands through the pipeline stages using the below wires
wordT src_a, src_b, src_aFromBR, src_bFromBR, src_aFromALU, src_bFromALU;

// pass the results from the branch and ALU stages through the processor
wordT resultFromBR, resultFromALU;

// write back connections
BOOLEAN wben; // enable write back
regNumT wb_rd; // register to write back to
wordT wb_result; // value to write back to register

// wires for stall signals
logic SfromDEC, SfromBR, SfromALU, SfromDMEM;

// wires to pass pcVal through the processor
wordT pcFromDMEM, pcFromIF, pcFromDEC, pcFromBR, pcFromALU;

// wires for present bits for context info to pass through the pipeline stages
presenceT presentFromDMEM, presentFromIF, presentFromBR,
        presentFromALU, presentForIF, presentFromDEC;

// control data
DECODED dFromDEC, dFromBR, dFromALU;

//////////////////////////////////////////////////////////////////////
// instantiate pipeline stages

// Instruction Fetch Stage
proc2_fetch IF(
  .clk, .rst,
  .stallIn(SfromDEC), .pcVal(pcFromDMEM), .presentIn(presentFromDMEM),
  .presentOut(presentFromIF), .pcValOut(pcFromIF), .instructionVal
);

// Decode and write back stage
proc2_decode DEC(
  .clk, .rst,
  .stallIn(SfromBR), .pcVal(pcFromIF), .presentIn(presentFromIF),
  .instructionVal,
  .wben, .wb_rd, .wb_result,
  .presentOut(presentFromDEC), .pcValOut(pcFromDEC), .dOut(dFromDEC),
  .src_a, .src_b, .stallOut(SfromDEC)
);

// Branch stage
proc2_branch BR(
  .clk, .rst,
  .stallIn(SfromALU), .pcVal(pcFromDEC), .d(dFromDEC), .src_a, .src_b,
  .presentIn(presentFromDEC),
  .stallOut(SfromBR), .dOut(dFromBR),
  .src_aOut(src_aFromBR), .src_bOut(src_bFromBR),
  .result(resultFromBR), .pcValOut(pcFromBR), .presentOut(presentFromBR)
);

// ALU/execute stage
proc2_ALU ALU(
  .clk, .rst,
  .stallIn(SfromDMEM), .pcVal(pcFromBR), .d(dFromBR), .resultIn(resultFromBR),
  .presentIn(presentFromBR),
  .src_a(src_aFromBR), .src_b(src_bFromBR),
  .stallOut(SfromALU), .dOut(dFromALU),
  .result(resultFromALU), .presentOut(presentFromALU),
  .pcValOut(pcFromALU), .copySrc_a(src_aFromALU), .copySrc_b(src_bFromALU)
);

// Data Memory (load / store) stage
proc2_DMEM DMEM(
  .clk, .rst,
  .pcVal(pcFromALU), .d(dFromALU), .presenceIn(presentFromALU),
  .result(resultFromALU), .copySrcB(src_bFromALU),
  .stallOut(SfromDMEM), .wben, .wb_result, .wb_rd,
  .pcValOut(pcFromDMEM), .presenceOut(presentFromDMEM)
);
```

MIPS — Top-level processor 2 of 2 6

```
// debug output
always @(presentFromIF, presentFromDEC, presentFromBR, presentFromALU, presentFromDMEM)
  $display("presence flags (IF,DEC,BR,ALU,DMEM)=(%1d,%1d,%1d,%1d,%1d)",
           presentFromIF, presentFromDEC, presentFromBR, presentFromALU, presentFromDMEM);

//////////////////////////////////////////////////////////////////////
// start and monitor the processors progress
always @(posedge clk or posedge rst)
  if(rst) begin
     cycleNum <= 0;
     prevPCVal <= 32'hFFFFFFFF;
  end else begin
     if (presentFromDMEM) begin
        prevPCVal <= pcFromIF;
     end

     if (prevPCVal == pcFromIF) begin
        $display("EXIT: branch to self, descheduling thread");
        $finish();
     end

     cycleNum <= cycleNum + 1;
     if (cycleNum >= 50000) begin
        // stop simulation in case of infinite loops
        $display("EXIT: cycle num exceeded");
        $finish();
     end
  end

endmodule
```

MIPS — Fetch Unit 7

```
`include "proc2_defsinc.v"

//////////////////////////////////////////////////////////////////////
// FETCH PIPELINE STAGE
// Role:
//      Retrieve instructions from the instruction_ROM
//      Pass the retrieved instruction to the DECODE
//      Increment the program counter
//////////////////////////////////////////////////////////////////////

module proc2_fetch(
    input clk,                  // clock signal
    input rst,                  // reset signal

    // input side
    input stallIn,              // stall signal
    input wordT pcVal,          // program counter value
    input presenceT presentIn,

    // output side
    output presenceT presentOut,
    output wordT pcValOut,
    output wordT instructionVal // instruction to execute
);

// initialise local wires and registers

// address of instruction to be looked up in the instruction ROM
//  - program counter is byte addressed where as instructions in ROM
//    are word addressed
wire [29:0] instAddr = pcVal >> 2;

// holds the instruction value retrieved from the instruction ROM
wire [31:0] instValOut;

// flag used to initialise the instruction fetch sequence
logic initFetch;

// instantiate copy of the instruction_ROM holding the program to be
// run on the processor
instruction_ROM ir(.instAddr, .instValOut);

// main fetch state machine
always_ff @(posedge clk)
  if (rst) begin
     // Initialise local registers
     pcValOut <= 0;
     presentOut <= presentNo;
     instructionVal <= 0;
     initFetch <= 1;
  end else if (!stallIn && ((presentIn == presentYes) || initFetch)) begin
     initFetch <= 0;
     $display("Sending instruction from PC=%08h",pcValOut);
     // Push presence information through the bus to indicate to DECODE
     // that it should decode the instruction
     presentOut <= (presentIn == presentYes) || initFetch ? presentYes : presentNo;
     instructionVal <= instValOut; // Instruction to be decoded passed to next stage
     pcValOut <= pcVal + 4; // Increment the program counter
  end else begin
     presentOut <= presentIn; // Push presence information through the bus
                              // indicating to DECODE that no instruction is
                              // currently needing decode performed
  end

endmodule
```

MIPS — Decode Unit 1 of 2 8

```
`include "proc2_defsinc.v"

//////////////////////////////////////////////////////////////////////////
// DECODE PIPELINE STAGE
// Role:
// * Extract bits corresponding to immediate, rs, rt, rd, shamt, etc. from
//   the instruction
// * Analyse instruction function / opcode and propogate appropriate control
//   information to the next pipeline stage
// * Perform write back
//      NOTE: the write back is not performed in the same clock cycle as the
//            decode
//      - It is peformed when the DMEM stage sets wben high to indicate the
//        write back data is ready
//////////////////////////////////////////////////////////////////////////

module proc2_decode(
        input clk,
        input rst,

        // input side
        input stallIn,
        input wordT pcVal,
        input presenceT presentIn,
        input wordT instructionVal,
        // inputs for write back
        input BOOLEAN wben,
        input regNumT wb_rd,
        input wordT wb_result,

        //output side
        output presenceT presentOut,
        output wordT pcValOut,
        output DECODED dOut,
        output wordT src_a,
        output wordT src_b,
        output stallOut
);

//initialise local wires and registers

// local DECODED information to pass out to dOut when presenceIn is high
DECODED d;

// opcode - the first 6 bits of the instruction
//        - indicates instruction type in case of non-R type instructions
logic[5:0] opcode;

// rFunc - the last 6 bits of the instruction
//       - indicates instruction type in case of R type instructions
logic[5:0] rFunc;

// Rtype - Indicates if instruction is an Rtype instruction
BOOLEAN Rtype;

// decROMaddr - Concatenates and multiplexes above information to create
// an opcode to represent instruction type
logic[6:0] decROMaddr;

// Register file
wordT rf[32];

assign stallOut = stallIn;

// combinational logic to do the main decode
always_comb begin
// Extract bits from instruction
opcode = instructionVal[31:26];
rFunc = instructionVal[5:0];

// Determine if the instruction is an Rtype instruction
Rtype = (opcode==6'h0) ? TRUE : FALSE;

// initial decode - more bit extraction
d.rs = instructionVal[25:21];
d.rt = instructionVal[20:16];
d.rd = Rtype ? instructionVal[15:11] : instructionVal[20:16];
// Zero extended immediate
d.zeroimm = {16'h0000, instructionVal[15:0]};
// Sign extended immediate
d.signimm = {instructionVal[15] ? 16'hffff : 16'h0000, instructionVal[15:0]};
d.addr = instructionVal[25:0] << 2;
d.shamt = instructionVal[5:0];

decROMaddr = {Rtype,Rtype ? rFunc : opcode};
```

MIPS — Decode Unit 2 of 2 9

```
// add control information to DECODED
case(decROMaddr)
7'h2:  d.func={ALU_f_NOP,FALSE,MEM_f_NONE,OPB_s_REG_RT,BR_J,NO};      // J
7'h3:  d.func={ALU_f_NOP,TRUE,MEM_f_NONE,OPB_s_REG_RT,BR_JAL,NO};     // JAL
7'h4:  d.func={ALU_f_NOP,FALSE,MEM_f_NONE,OPB_s_REG_RT,BR_B,EQ};      // BEQ
7'h6:  d.func={ALU_f_NOP,FALSE,MEM_f_NONE,OPB_s_REG_RT,BR_B,LEZ};     // BLEZ
7'h8:  d.func={ALU_f_ADD,TRUE,MEM_f_NONE,OPB_s_SIGNIMM,BR_NONE,NO};   // ADDI
7'h9:  d.func={ALU_f_ADD,TRUE,MEM_f_NONE,OPB_s_SIGNIMM,BR_NONE,NO};   // LI
7'hD:  d.func={ALU_f_OR,TRUE,MEM_f_NONE,OPB_s_SIGNIMM,BR_NONE,NO};    // ORI
7'h23: d.func={ALU_f_ADD,TRUE,MEM_f_LW,OPB_s_SIGNIMM,BR_NONE,NO};     // LW
7'h2B: d.func={ALU_f_ADD,FALSE,MEM_f_SW,OPB_s_SIGNIMM,BR_NONE,NO};    // SW
7'h40: d.func={ALU_f_SLL,TRUE,MEM_f_NONE,OPB_s_REG_RT,BR_NONE,NO};    // SLL
7'h48: d.func={ALU_f_NOP,TRUE,MEM_f_NONE,OPB_s_REG_RT,BR_JR,NO};      // JR
7'h49: d.func={ALU_f_NOP,TRUE,MEM_f_NONE,OPB_s_REG_RT,BR_JALR,NO};    // JALR
7'h60: d.func={ALU_f_ADD,TRUE,MEM_f_NONE,OPB_s_REG_RT,BR_NONE,NO};    // ADD
7'h62: d.func={ALU_f_SUB,TRUE,MEM_f_NONE,OPB_s_REG_RT,BR_NONE,NO};    // SUB
default: begin
  d.func = { ALU_f_NOP,FALSE,MEM_f_NONE,OPB_s_REG_RT,BR_NONE,NO}; // NOP
  $display("DEC: ERROR: ROM address %b %b %b not defined opcode(%b) Rtype(%s)",
           decROMaddr[6],decROMaddr[5:2],decROMaddr[1:0],opcode,Rtype.name);
  $finish();
end // end: default
endcase
end // end: always_comb

// The main decode state machine
always_ff @(posedge clk or posedge rst)
  if (rst) begin
     presentOut <= presentNo;
  end else if(!stallIn) begin
     // Perform write back to register file
     if ((wben == TRUE) && (wb_rd != 0)) begin
        rf[wb_rd] <= wb_result;
     end
     // Propogate control information out to next pipeline stage
     if (presentIn == presentYes) begin
        src_a <= rf[d.rs]; // fetch src A from register file
        src_b <= rf[d.rt]; // fetch src B from register file
        dOut <= d;
        pcValOut <= pcVal;
     end
     presentOut <= presentIn;
  end
endmodule
```

MIPS — Branch 10

```
`include "proc2_defsinc.v"

//////////////////////////////////////////////////////////////////////
// BRANCH PIPELINE STAGE
// Role:
//       In the case of conditional branches, analyse if the branch should
//       be taken.  Update the pcVal to a value appropriate for the branch
//       (if branch taken)
//////////////////////////////////////////////////////////////////////

module proc2_branch(
        input clk,
        input rst,

        // input side
        input stallIn,
        input wordT pcVal,
        input DECODED d,
        input wordT src_a,
        input wordT src_b,
        input presenceT presentIn,

        // output side
        output stallOut,
        output DECODED dOut,
        output wordT src_aOut,
        output wordT src_bOut,
        output wordT result,
        output wordT pcValOut,
        output presenceT presentOut
);

assign stallOut = stallIn;

// Instantiate local wires and registers
// Indicates if branch condition was true or not
BOOLEAN branch_taken;

// Used to ensure that number is interpreted as signed
// (otherwise si_a < 0 wouldn't work)
int signed si_a;

always_comb begin
   // Perform combinatorially in order that branch_taken is set
   // before the positive clock edge appears

    si_a = src_a;

   // Analyse if the branch should be taken
   case(d.func.branch_cond)
     NO: branch_taken = FALSE;
     EQ: branch_taken = (src_a == src_b) ? TRUE : FALSE;
     LEZ: begin
        branch_taken =    (si_a < 0 || si_a == 0) ? TRUE : FALSE;
     end
     default: branch_taken = TRUE; // unconditional branch
   endcase
end

always_ff @(posedge clk)
  if (rst) begin
     presentOut <= presentNo;
     pcValOut <= 0;
  end else if (!stallIn && (presentIn == presentYes)) begin

     // Propogate control information out
     presentOut <= presentIn;
     dOut <= d;
     src_aOut <= src_a;
     src_bOut <= src_b;

     // Set the pcValOut appropriately depending on branch_type and whether
     // branch_taken is set
     case(d.func.branch_type)
       BR_JAL: begin
          result <= pcVal;
          pcValOut <= d.addr;
       end
       BR_JALR: begin
          result <= pcVal;
          pcValOut <= src_a;
       end
       BR_JR: begin
          pcValOut <= src_a;
       end
       BR_J: begin
          pcValOut <= d.addr;
       end
       BR_B: begin
          if (branch_taken) pcValOut <= pcVal+(d.signimm<<2);
          else pcValOut <= pcVal;
       end
       default: begin
          pcValOut <= pcVal;
       end
     endcase // case (d.func.branch_type)
  end else if (presentIn == presentNo) presentOut <= presentNo;
endmodule
```

MIPS — ALU 11

```
`include "proc2_defsinc.v"

//////////////////////////////////////////////////////////////////////
// ALU / EXECUTE PIPELINE STAGE
// Role:
//       Select appropriate operands to perform the ALU operations on
//       Perform the ALU operations on the operands
//////////////////////////////////////////////////////////////////////

module proc2_ALU(
    input clk,
    input rst,

    // input side
    input stallIn,
    input wordT pcVal,
    input DECODED d,
    input wordT resultIn,
    input presenceT presentIn,
    input wordT src_a,
    input wordT src_b,

    // output side
    output DECODED dOut,
    output stallOut,
    output wordT result,
    output presenceT presentOut,
    output wordT pcValOut,
    output wordT copySrc_a,
    output wordT copySrc_b
);

// initialise local wires and registers
wordT operand_a;
wordT operand_b;

assign stallOut = stallIn;

always_comb begin
   // operand_a always equals the value fetched for register rs from
   // the register file
   operand_a = src_a;

   // select operand B – can be a signed immediate, a zero extended
   // immediate or a value from the register file (rt)
   case (d.func.opb_s)
     OPB_s_ZEROIMM: operand_b <= d.zeroimm;
     OPB_s_SIGNIMM: operand_b <= d.signimm;
     default: operand_b <= src_b;
   endcase
end

// main ALU state machine
always_ff @(posedge clk)
  if (rst) begin
     // rst outputs
     presentOut <= presentNo;
     pcValOut <= 0;
  end else if (!stallIn && (presentIn == presentYes)) begin
     // execute
     case(d.func.alu_f)
       ALU_f_ADD: result <= operand_a + operand_b;
       ALU_f_SUB: result <= operand_a – operand_b;
       ALU_f_OR : result <= operand_a | operand_b;
       ALU_f_SLL: result <= operand_b << d.shamt;
       ALU_f_NOP: result <= resultIn;
       default  : result <= 32'hxxxxxxxx;
     endcase

     // propogate context information forward in the pipeline
     dOut <= d;
     presentOut <= presentIn;
     pcValOut <= pcVal;
     copySrc_a <= src_a;
     copySrc_b <= src_b;
  end else if (presentIn == presentNo ) begin
     presentOut <= presentNo;
  end

endmodule
```

MIPS — Data Memory Unit 12

```
`include "proc2_defsinc.v"

//////////////////////////////////////////////////////////////////////////
// DATA MEMORY PIPELINE STAGE (LOADS / STORES)
// Role:
//       Instantiate data memory
//       Initialise instruction_ROM data into memory
//       Write / load from memory in the case of loads and stores
//       Generate appropriate control and data signals for write back
//////////////////////////////////////////////////////////////////////////

module proc2_DMEM(
    input clk,
    input rst,

    //input side
    input wordT pcVal,
    input DECODED d,
    input presenceT presenceIn,
    input wordT result,
    input wordT copySrcB,

    // output side
    output stallOut,
    output BOOLEAN wben,
    output wordT wb_result,
    output regNumT wb_rd,
    output wordT pcValOut,
    output presenceT presenceOut
);

parameter int memsize = 4048;
parameter int userInput = 4;

// instantiate local wires and registers
logic [29:0] instAddr;

logic [31:0] instValOut;

// instantiate memory for processor
reg [31:0] dmem[memsize:0];

// instantiate copy of the instruction ROM holding the program to be copied
// into the data memory
instruction_ROM ir(.instAddr, .instValOut);

wire initMem = (instAddr < memsize);
assign stallOut = initMem;

// DMEM state machine
always_ff @(posedge clk or posedge rst)
  if(rst) begin
     // reset
     instAddr <= 0;
     presenceOut <= presentNo;
     pcValOut <= 0;
     wben <= FALSE;
  end else if (initMem) begin
     // Initalise memory with instruction_ROM
     dmem[instAddr] <= instValOut;
     instAddr <= instAddr + 1;
  end else if (presenceIn == presentYes) begin
     case(d.func.mem_f)
       MEM_f_LW: begin
          if(result==128) begin // address to simulate input
             $display("Using %1d as user input value.",userInput);
             wb_result <= userInput;
          end else
            // Perform load
            wb_result <= dmem[result >> 2];
          wb_rd <= d.rd;
          wben <= TRUE;
       end
       MEM_f_SW: begin
          dmem[result >> 2] <= copySrcB; // Write to DMEM
          wben <= FALSE;
          if(result==128) // address to simulate output
            $display("********** Result output: %1d **********",copySrcB);
       end
       default: begin
          // Set write back register to $ra in the case of
          // a jump and link instruction
          if (d.func.branch_type == BR_JAL ||
              d.func.branch_type == BR_JALR) wb_rd <= 31;
          else begin
             // Otherwise set write back register to rd (dest. reg.)
             wb_rd <= d.rd;
          end
          wben <= d.func.wben; // Indicate if write back should be performed
          wb_result <= result;
       end
     endcase

     // Propogate control information to instruction fetch
     pcValOut <= pcVal;
     presenceOut <= presenceIn;
  end else if (presenceIn == presentNo) begin
     presenceOut <= presenceIn;
     wben <= FALSE;
  end

endmodule
```

Example simulation output 13

```
./simv
Chronologic VCS simulator copyright 1991-2005
Contains Synopsys proprietary information.
Compiler version Y-2006.06-SP1-1_Full64; Runtime version Y-2006.06-SP1-1_Full64;
Oct  9 16:52 2008

Sending instruction from PC=00000000
presence flags (IF,DEC,BR,ALU,DMEM)=(1,0,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,1,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,1,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,1,0)
Using 4 as user input value.
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,0,1)
Sending instruction from PC=00000004
presence flags (IF,DEC,BR,ALU,DMEM)=(1,0,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,1,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,1,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,1,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,0,1)
Sending instruction from PC=00000008
presence flags (IF,DEC,BR,ALU,DMEM)=(1,0,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,1,0,0,0)

          ⋮

Sending instruction from PC=00000014
presence flags (IF,DEC,BR,ALU,DMEM)=(1,0,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,1,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,1,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,1,0)
********** Result output: 5 **********
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,0,1)
Sending instruction from PC=0000002c
presence flags (IF,DEC,BR,ALU,DMEM)=(1,0,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,1,0,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,1,0,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,1,0)
presence flags (IF,DEC,BR,ALU,DMEM)=(0,0,0,0,1)
Sending instruction from PC=00000030
presence flags (IF,DEC,BR,ALU,DMEM)=(1,0,0,0,0)
EXIT: branch to self, descheduling thread
$finish at simulation time                41755
           V C S   S i m u l a t i o n   R e p o r t
Time: 41755
CPU Time:      0.030 seconds;       Data structure size:   0.0Mb
Thu Oct  9 16:52:58 2008
```

Comments 14

- only a subset of the processor has been implemented here
- ECAD+Arch Labs use a different (earlier) Verilog2001 MIPS processor which is much more complete, but is more complex
  - this used to be lectured and the old slides are available via the ECAD+Arch web page
- the design in this lecture is highly sequential

☞ next lecture will look at extracting more parallelism