

Tutorial #1

Verilog Simulation Toolflow

Tutorial Notes Courtesy of Christopher Batten

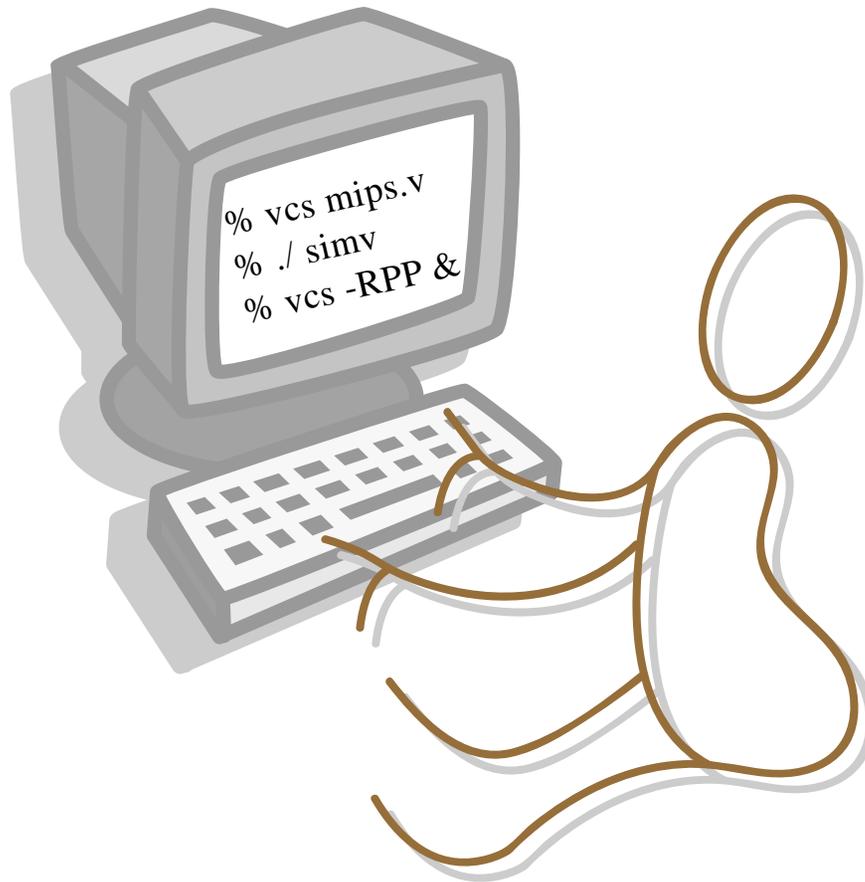
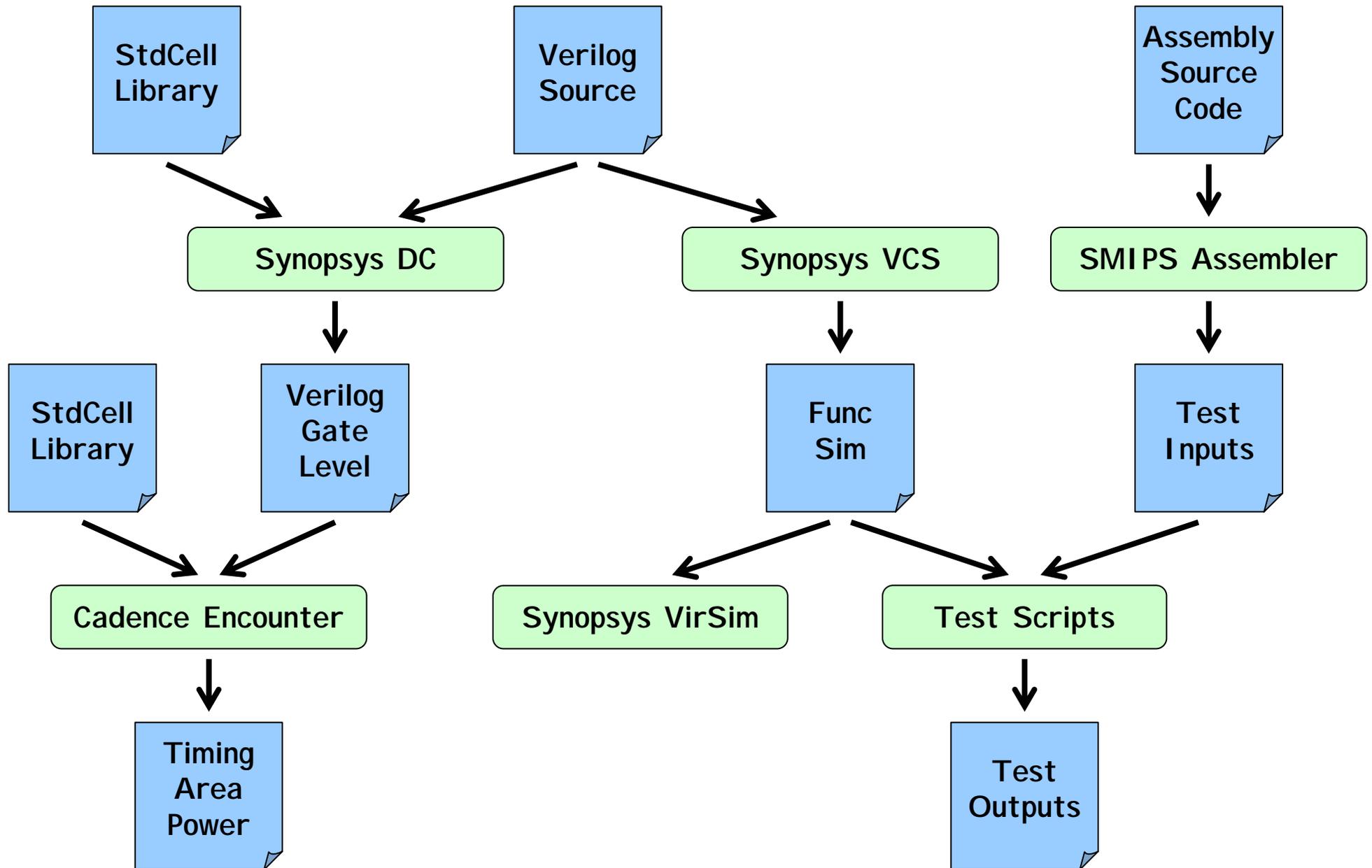


Figure by MIT OCW.

6.884 Toolflow For Lab 1 and 2



Tour of the MIT server Locker

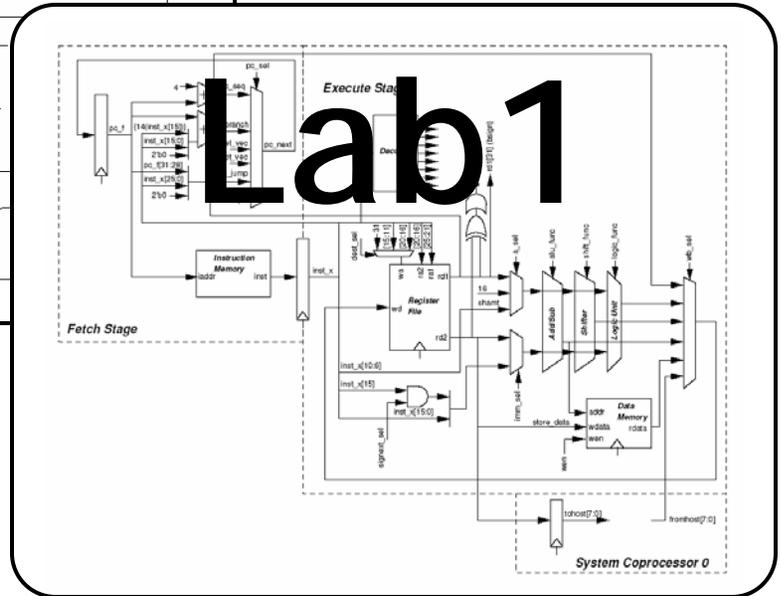
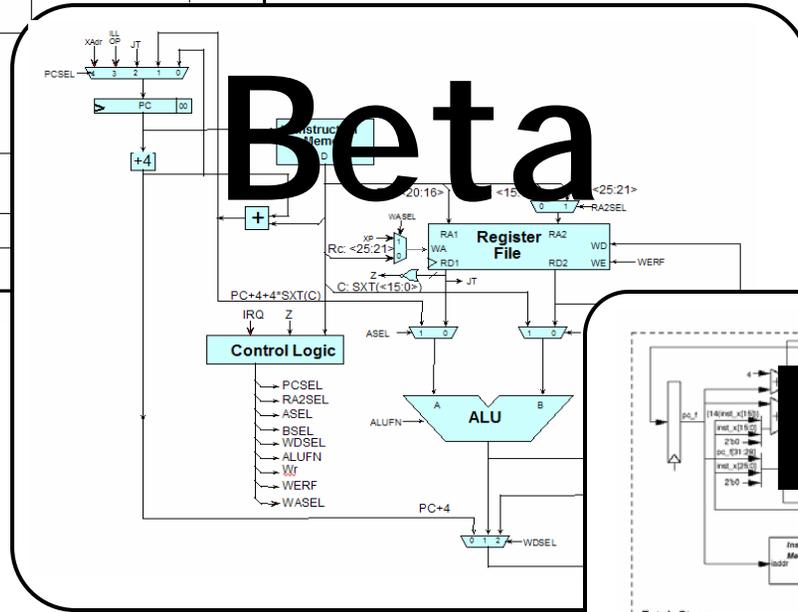
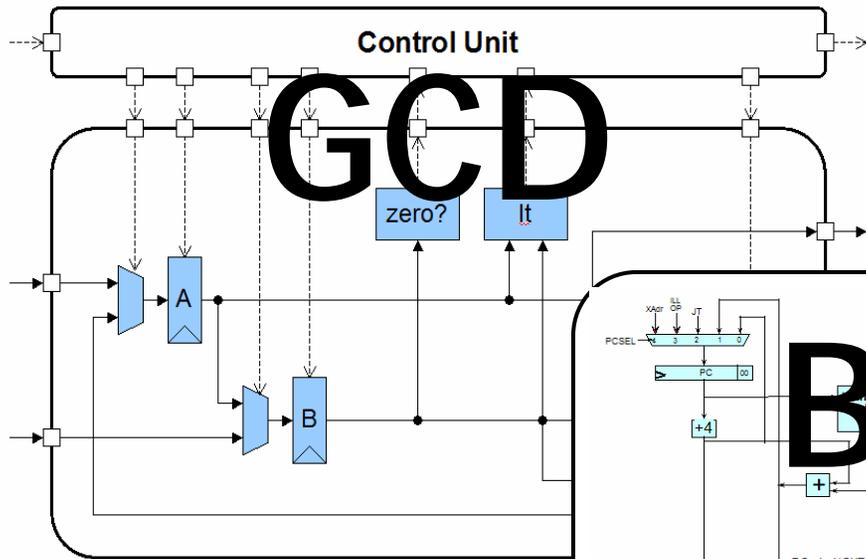
To access the locker use `setup 6.884`

- `/mit/6.884/tools`
- `/mit/6.884/doc`
- `/mit/6.884/examples`

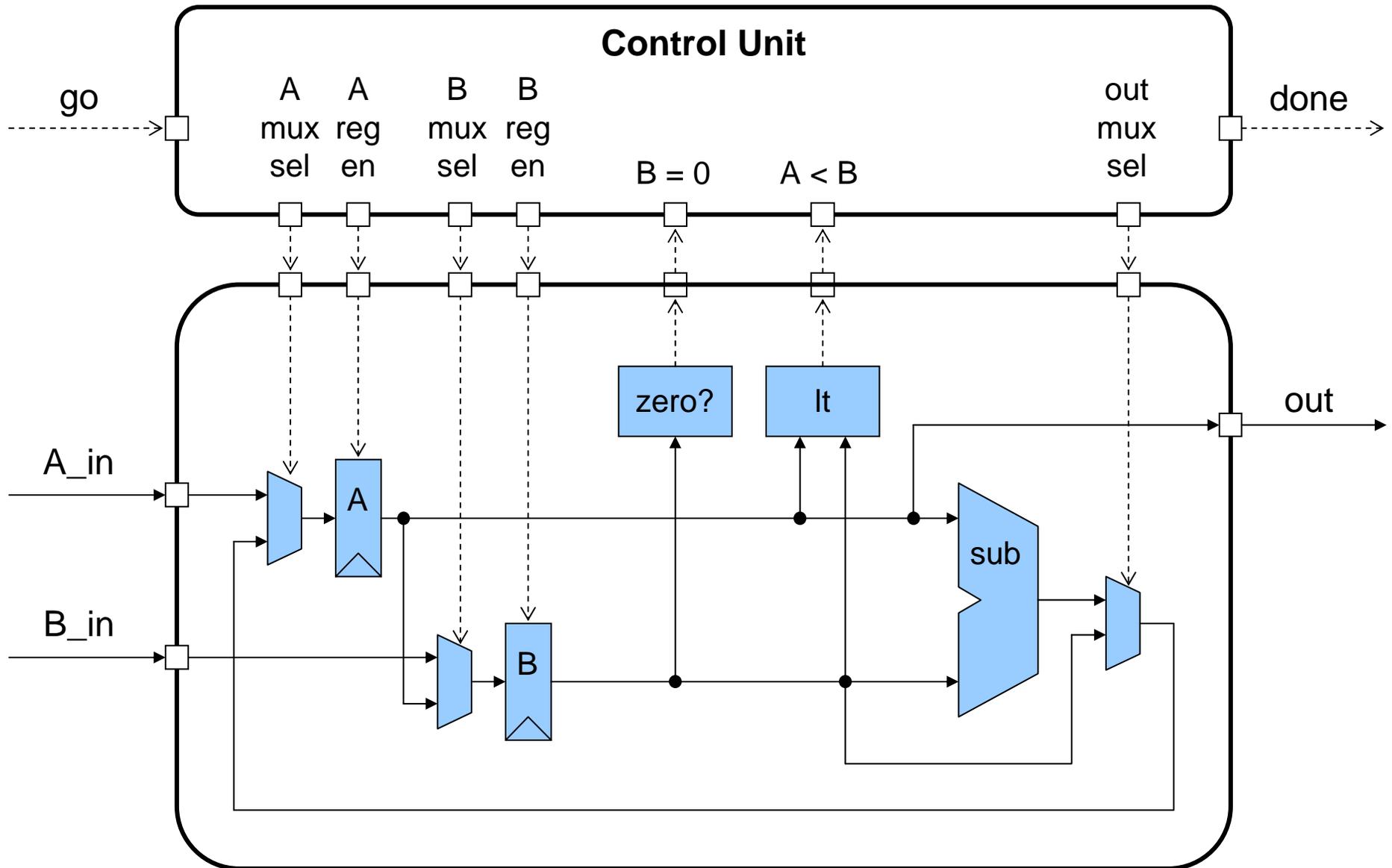
For those with access to the CAG network please make use of the CAG infrastructure

- Use `setup 6.884` (will setup tools needed for the course)
- Locker (partially) mirrored at `/projects/assam/6.884`
- CAG updates will probably lag a bit behind the MIT server

Three Examples



Greatest Common Divisor



GCD Control Logic

```

while ( !done )
  if ( A < B )
    swap( A, B )
  else if ( B != 0 )
    A = A - B
  else
    done = true
endwhile

```

	A_en	B_en	A_mux_sel	B_mux_sel	out_mux_sel	done
Load	1	1	0	0	X	0
Swap	1	1	1	1	1	0
Sub	1	0	1	X	0	0
Done	0	0	X	X	X	1

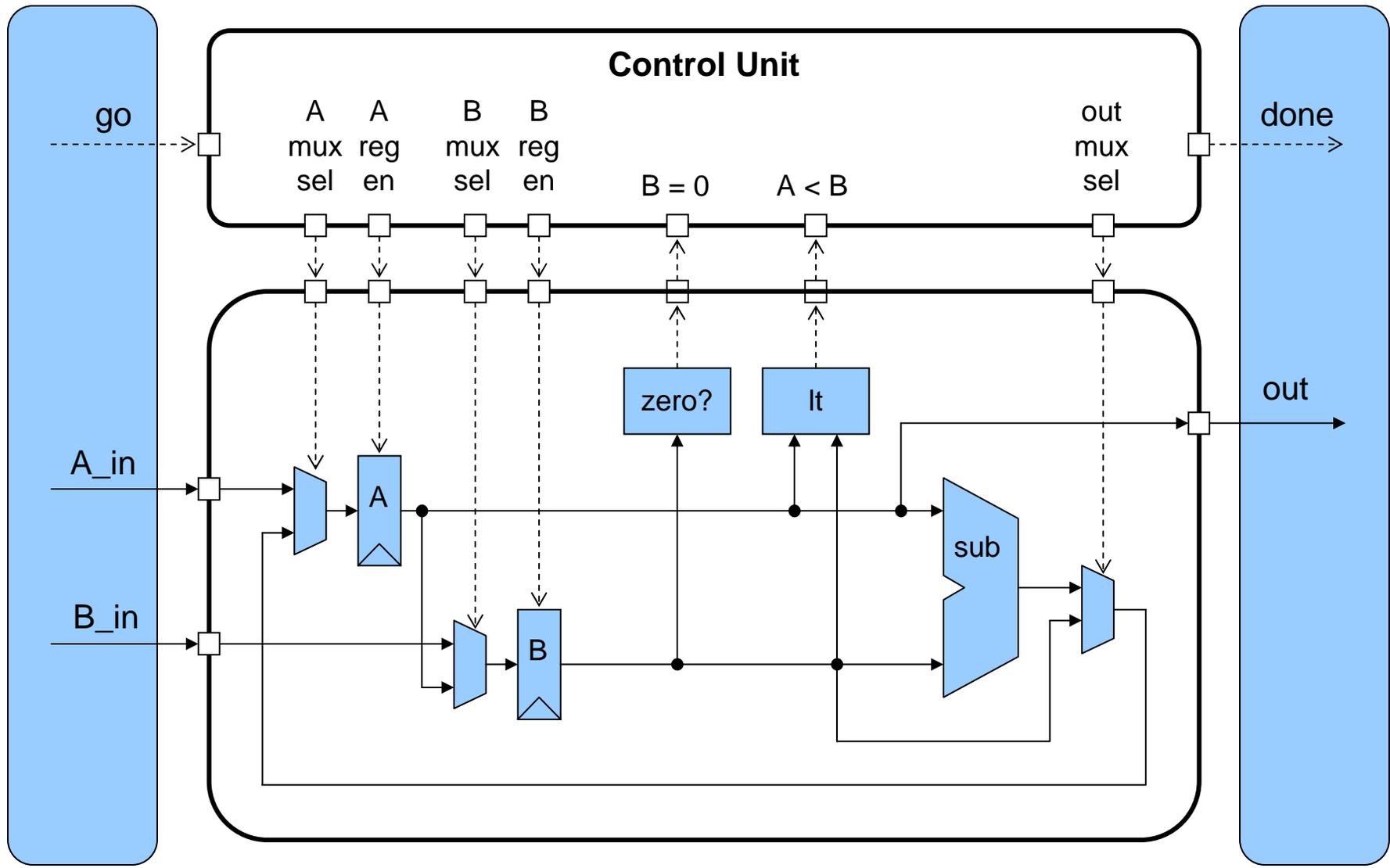
```

reg [5:0] ctrl_sig;
assign { A_en, B_en, A_mux_sel, B_mux_sel, out_mux_sel, done } = ctrl_sig;

always @(*)
begin
  if ( !running )      ctrl_sig = 6'b11_00x_0; // Latch in A and B values
  else if ( A_lt_B )   ctrl_sig = 6'b11_111_0; // A <= B and B <= A
  else if ( !B_zero ) ctrl_sig = 6'b10_1x0_0; // A <= A - B and B <= B
  else                 ctrl_sig = 6'b00_xxx_1; // Done
end

```

GCD Test Harness



GCD Test Harness

```
module gcd_test;
  reg clock = 0; reg reset = 0; reg go = 0;
  reg [15:0] A_in, B_in;

  wire done;
  wire [15:0] Y;

  always #10 clock = ~clock;
  gcd_rtl gcd_unit( .clock(clock), .reset(reset), .go(go),
                  .A_in(A_in), .B_in(B_in), .done(done), .Y(Y) );

  initial begin
    $vcdpluson(0);
    A_in = 27; B_in = 15;           // Set defaults
    $value$plusargs("a-in=%d",A_in); // Read in cmdline args
    $value$plusargs("b-in=%d",B_in);

    #5 reset = 1; #20 reset = 0;   // Strobe reset signal
    #20 go = 1; #20 go = 0;       // Strobe go signal
  end

  always @( done ) begin
    if ( done ) begin
      #15;
      $display(" a-in    = %d", A_in );
      $display(" b-in    = %d", B_in );
      $display(" gcd-out = %d", Y    );
      $finish;
    end
  end

endmodule
```

Try to keep the Verilog
part of the test harness
as simple as possible!

Building a GCD Simulator with VCS

```
% setup 6.884
```

```
Attaching 6.884 ...
```

```
Running commands in /mit/6.884/.attachrc ...
```

```
% vcs -PP +lint=all +v2k gcd_rtl.v
```

```
% ./simv
```

```
VCD+ Writer 4.4R10 Copyright 1993-2004 Synopsys Inc.
```

```
  a-in      =      27
```

```
  b-in      =      15
```

```
  gcd-out   =       3
```

```
$finish at simulation time          245
```

```
% ./simv +a-in=49 +b-in=28
```

```
VCD+ Writer 4.4R10 Copyright 1993-2004 Synopsys Inc.
```

```
  a-in      =      49
```

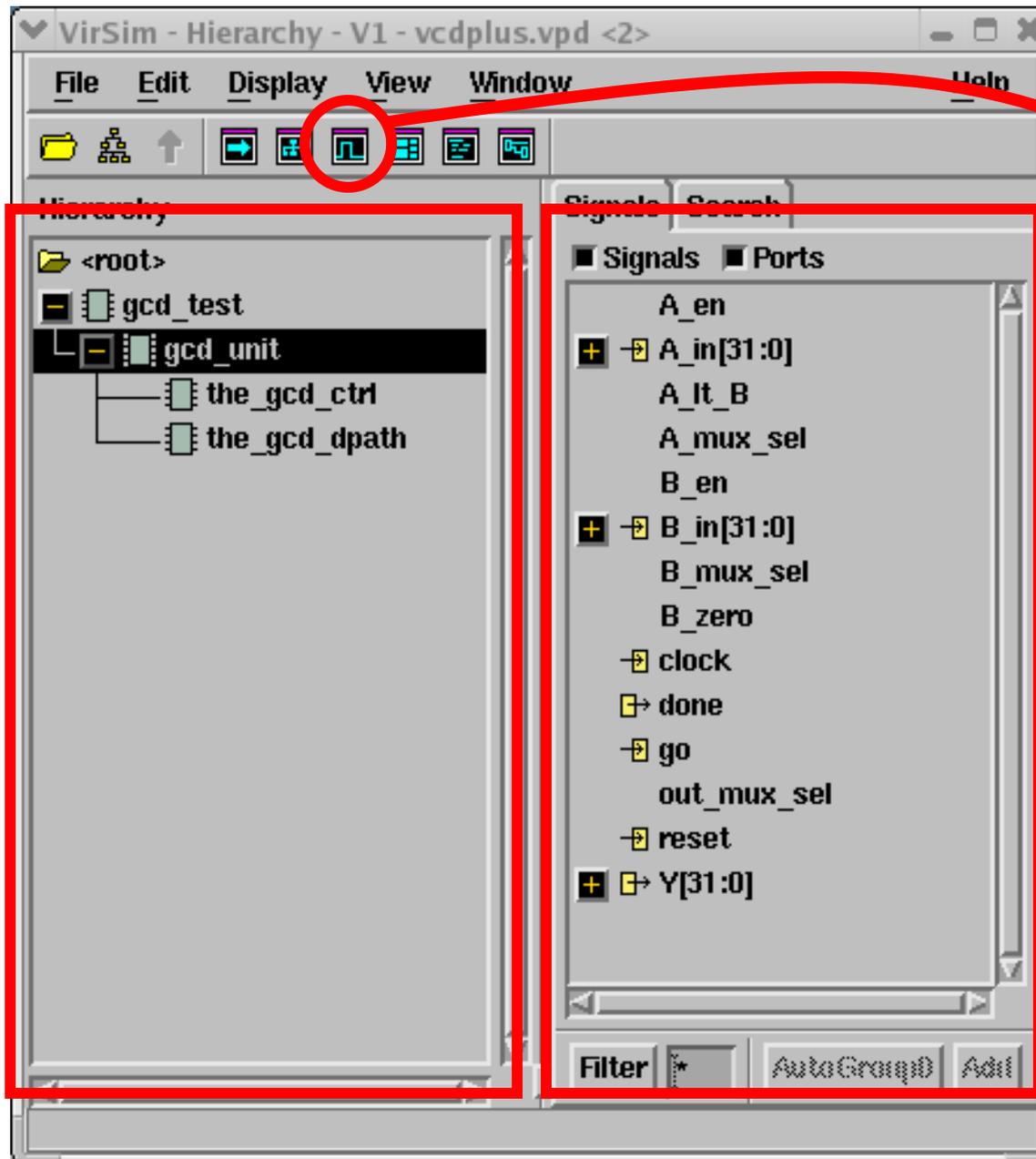
```
  b-in      =      28
```

```
  gcd-out   =       7
```

```
$finish at simulation time          225
```

```
% vcs -RPP &
```

Viewing Waveforms with VirSim



Open up a waveform viewer

Browse the module hierarchy

Select signals for display

Viewing Waveforms with VirSim



Using Test Scripts

```
% vcs -PP +lint=all +v2k gcd_rtl.v
% vmodel-tester.pl -v ./simv gcd-test.dat

* Verilog model = ./simv
* Test data file = gcd-test.dat
* Running tests
+ Testing a-in=27 b-in=15 gcd-out=3
+ Testing a-in=21 b-in=49 gcd-out=7
+ Testing a-in=25 b-in=30 gcd-out=5
+ Testing a-in=19 b-in=27 gcd-out=1
+ Testing a-in=40 b-in=40 gcd-out=40
+ Testing a-in=250 b-in=190 gcd-out=10
+ Testing a-in=5 b-in=250 gcd-out=5
+ Testing a-in=0 b-in=0 gcd-out=0

*** PASSED ***
```

Using the mkasic.pl Script

```
gcd/  
mkasic.pl  
verilog/  
  gcd_behavioral.v  
  gcd_rtl.v  
config/  
  gcd_behavioral.cfg  
  gcd_rtl.cfg  
tests/  
  gcd-test.dat
```

```
gcd/  
  gcd_behavioral/  
    vcs/  
    vcstest/  
  gcd_rtl/  
    vcs/  
    vcstest/
```

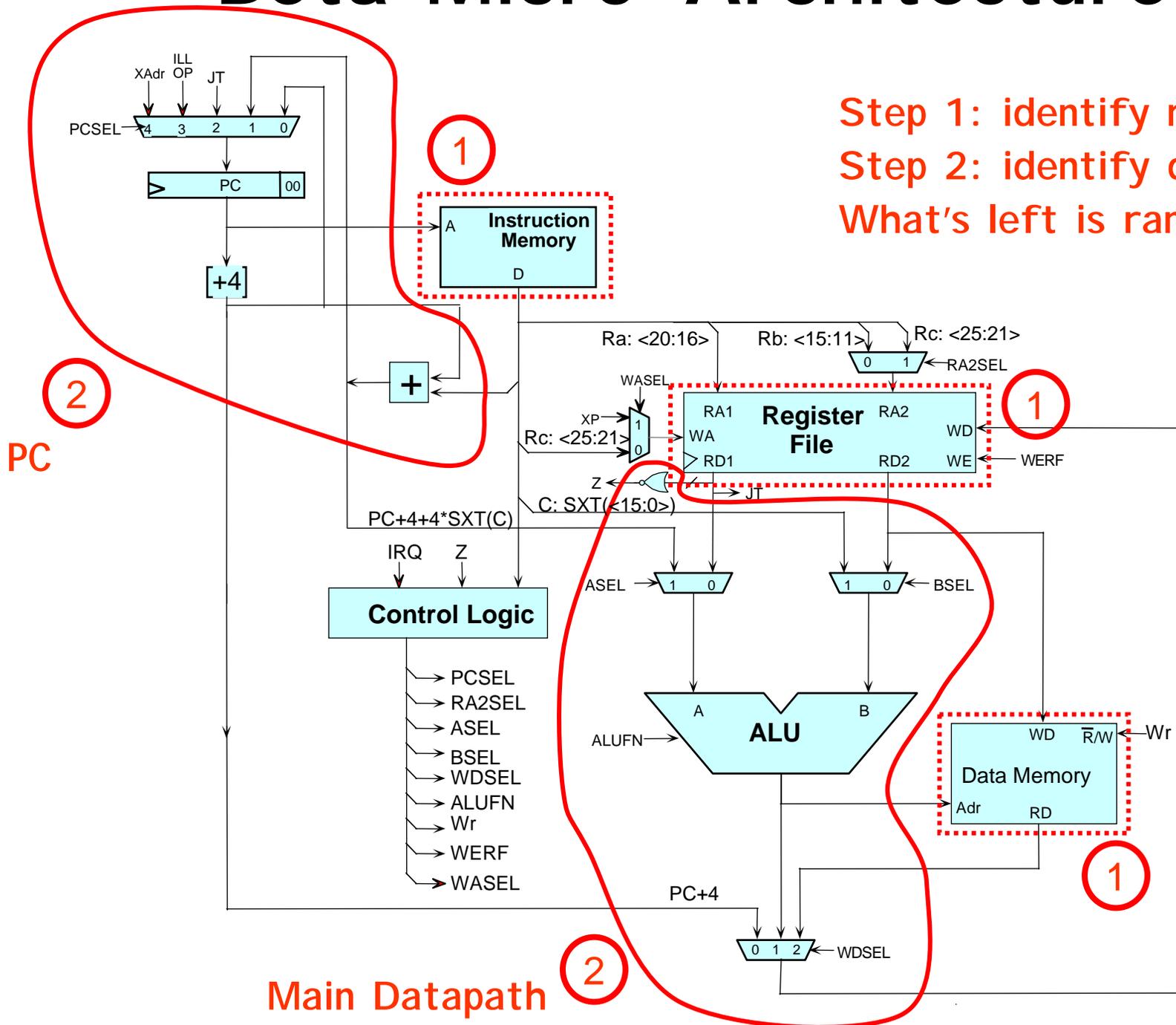
```
% cp -r /mit/6.884/examples/gcd  
% cd gcd  
% ./mkasic.pl -v config/gcd_rtl.cfg vcs  
% ./mkasic.pl -v config/gcd_behavioral.cfg vcs  
% ./mkasic.pl -v config/gcd_rtl.cfg vcstest  
% ./mkasic.pl -v config/gcd_behavioral.cfg vcstest
```

Writing Config Files for mkasic.pl

```
#-----  
# General configuration options  
  
package general;  
$outputDir      = "gcd_rtl";      # Dir for all generated product  
$verilogSrcDir  = "verilog";      # Dir where all verilog source is located  
$toplevelSource = "gcd_rtl.v";    # Verilog src file w/ toplevel module  
$toplevelModule = "gcd_test";     # Name of toplevel verlog module  
  
#-----  
# VCS - Synopsys Verilog compiler  
  
package vcs;  
$simName = "gcd-rtl";             # Name to use for the VCS simulator  
$cmdLineOptions = "";            # Any additional VCS compiler options  
  
#-----  
# Test  
  
package vcstest;  
$inputDir = "tests";             # Dir containing test inputs  
@testList = ( "gcd-test.dat" );  # A list of test inputs  
$cmdLine  = "vmodel-tester.pl -v __SIMNAME__ __TESTINPUT__";  
  
return 1; # Return true for mkasic.pl
```

Beta Micro-Architecture

Step 1: identify memories
 Step 2: identify datapaths
 What's left is random logic ...



Main Datapath

PC Dpath

```
module pc
(
    input          clk,
    input          reset,           // forces PC to 0x80000000
    input  [2:0]   pcsel,           // selects source of next PC
    input  [15:0]  offset,          // inst[15:0]
    input  [31:0]  jump_addr,       // from Reg[RA], used in JMP instruction
    output [31:0]  branch_addr,     // send to datapath for LDR instruction
    output [31:0]  pc,              // used as address for instruction fetch
    output [31:0]  pc_plus_4        // saved in regfile during branches, JMP, traps
);

reg [31:0] pc;
wire [30:0] pcinc;
wire [31:0] npc;

assign pcinc = pc + 4;
assign pc_plus_4 = {pc[31],pcinc};

// Branch address = PC + 4 + 4*sxt(offset)
assign branch_addr = {1'b0, pcinc + {{13{offset[15]}},offset[15:0],2'b00}};

assign npc = reset          ? 32'h80000000      :
            (pcsel == 0) ? {pc[31],pcinc}       : // normal
            (pcsel == 1) ? {pc[31],branch_addr[30:0]} : // branch
            (pcsel == 2) ? {pc[31] & jump_addr[31],jump_addr[30:0]} : // jump
            (pcsel == 3) ? 32'h80000004        :
            (pcsel == 4) ? 32'h80000008        : // illop, trap
            32'hXXXXXXXX;                       // catch errors...

always @( posedge clk )
    pc <= npc;

endmodule
```

Beta Register File

```
module regfile
(
    input  [4:0]  ra1,    // address for read port 1 (Reg[RA])
    output [31:0] rd1,    // read data for port 1
    input  [4:0]  ra2,    // address for read port 2 (Reg[RB], Reg[RC] for ST)
    output [31:0] rd2,    // read data for port 2
    input                clk,
    input                werf, // write enable, active high
    input  [4:0]  wa,    // address for write port (Reg[RC])
    input  [31:0] wd     // write data
);

// The register file itself
reg [31:0] registers[31:0];

// read paths are combinational, register 31 is always zero register
assign rd1 = (ra1 == 5'b11111) ? 32'h00000000 : registers[ra1];
assign rd2 = (ra2 == 5'b11111) ? 32'h00000000 : registers[ra2];

// write port is active only when WERF is asserted
always @(posedge clk)
begin
    if ( werf )
        registers[wa] <= wd;
end

endmodule
```

```

module datapath
(
    input  [15:0] inst,           // constant field from instruction
    input  [31:0] rd1,           // Reg[RA] from register file
    input  [31:0] rd2,           // Reg[RB] from register file (Reg[RC] for ST)
    input  [31:0] pc_plus_4,     // incremented PC
    input  [31:0] branch_addr,   // PC + 4 + 4*sxt(inst[15:0])
    input  [31:0] mem_rd_data,   // memory read data (for LD)
    input          asel,         // select A operand for ALU
    input          bsel,         // select B operand for ALU
    input  [1:0]  wdsel,         // select regfile write data
    input  [5:0]  alufn,         // operation to be performed by alu
    output [31:0] wdata,         // regfile write data (output of WDSEL mux)
    output [31:0] mem_addr,      // alu output, doubles as data memory address
    output [31:0] jump_addr,     // jump address (from Reg[RA])
    output [31:0] mem_wr_data,   // data memory write data (from Reg[RC])
    output          z            // true if Reg[RA] is zero, used during branches
);

    wire [31:0] alu_a;           // A input to ALU
    wire [31:0] alu_b;           // B input to ALU

    // compute A and B inputs into alu, also Z bit for control logic
    dp_misc misc(asel,bsel,inst,rd1,rd2,branch_addr,
                alu_a,alu_b,jump_addr,mem_wr_data,z);

    // where all the heavy-lifting happens
    dp_alu alu(alufn,alu_a,alu_b,mem_addr);

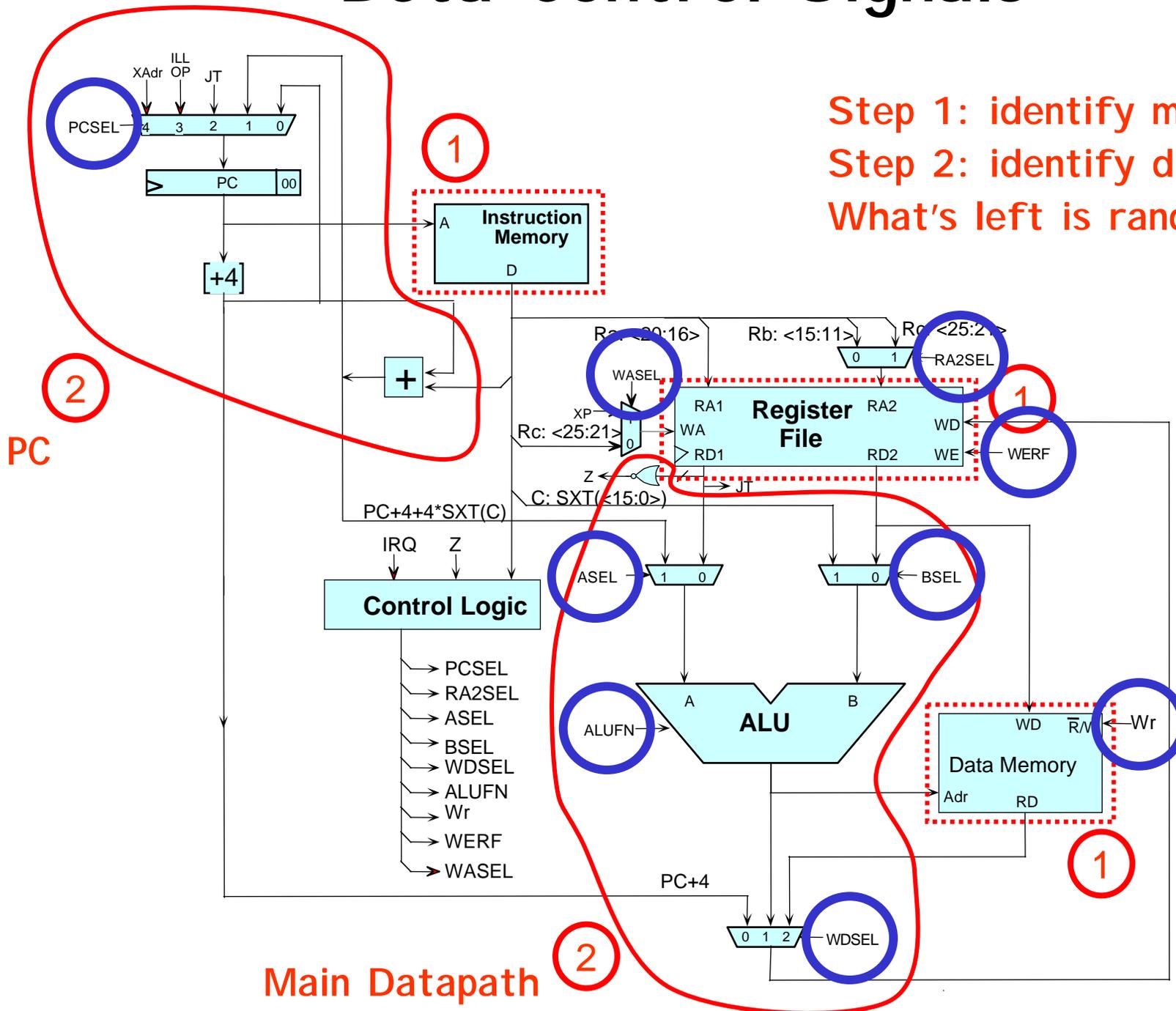
    // Mux to select regfile write data from PC+4, alu output, and memory data
    dp_wdata_mux wdata_mux(wdsel,pc_plus_4,mem_addr,mem_rd_data,wdata);

endmodule

```

Beta Control Signals

Step 1: identify memories
 Step 2: identify datapaths
 What's left is random logic ...



Main Datapath

Beta Control Signals

	<i>OP</i>	<i>OPC</i>	<i>LD</i>	<i>ST</i>	<i>JMP</i>	<i>BEQ</i>	<i>BNE</i>	<i>LDR</i>	<i>Illop</i>	<i>trap</i>
<i>ALUFN</i>	F(op)	F(op)	"+"	"+"	—	—	—	"A"	—	—
<i>WERF</i>	1	1	1	0	1	1	1	1	1	1
<i>BSEL</i>	0	1	1	1	—	—	—	—	—	—
<i>WDSEL</i>	1	1	2	—	0	0	0	2	0	0
<i>WR</i>	0	0	0	1	0	0	0	0	0	0
<i>RA2SEL</i>	0	—	—	1	—	—	—	—	—	—
<i>PCSEL</i>	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	3	4
<i>ASEL</i>	0	0	0	0	—	—	—	1	—	—
<i>WASEL</i>	0	0	0	—	0	0	0	0	1	1

Beta Control Unit

```
module control( ... );
```

```
    // Control ROM
    reg [15:0] ctl;
    always @(*)
    begin
```

```
        if (interrupt)
            ctl = 16'bx_100_1xx_xxxxxx_00_0;
        else
            case (inst[31:26])
                // Label control signal vector here ...
                default:    ctl = 16'bx_011_1xx_xxxxxx_00_0; // illegal opcode
                6'b011000:  ctl = 16'bx_000_001_00xxx0_10_0; // LD
                6'b011001:  ctl = 16'bx_000_x01_00xxx0_10_1; // ST
                6'b011011:  ctl = 16'bx_010_0xx_xxxxxx_00_0; // JMP
                6'b011101:  ctl = 16'b1_001_0xx_xxxxxx_00_0; // BEQ
                6'b011110:  ctl = 16'b0_001_0xx_xxxxxx_00_0; // BNE
                6'b011111:  ctl = 16'bx_000_010_011010_10_0; // LDR
                ...
                6'b111100:  ctl = 16'bx_000_001_10xx00_01_0; // SHLC
                6'b111101:  ctl = 16'bx_000_001_10xx01_01_0; // SHRC
                6'b111110:  ctl = 16'bx_000_001_10xx11_01_0; // SRAC
            endcase
    end
```

```
    // We now use the local control signal vector to create the
    // appropriate named control signals.
```

```
    assign werf    = ~ctl[0];
    assign mem_we  = !reset & ctl[0];
    assign wdsel   = ctl[2:1];
    assign alufn   = ctl[8:3];
    assign bsel    = ctl[9];
    assign asel    = ctl[10];
    assign wa      = ctl[11] ? 5'b11110 : inst[25:21];
    assign pcsel   = ((ctl[14:12] == 3'b001) & (ctl[15] ^ z)) ? 3'b000 : ctl[14:12];
```

```
    assign ra2     = ctl[0] ? inst[25:21] : inst[15:11];
endmodule
```

Putting It All Together

```
module beta
(
  input      clk,
  input      reset,
  input      irq,
  output [31:0] inst_addr,    // address of instruction to be fetched
  input  [31:0] inst_data,    // instruction returning from memory
  output [31:0] mem_addr,     // address of data word to be accessed
  input  [31:0] mem_rd_data,  // read data returning from memory
  output      mem_we,        // memory write enable, active high
  output [31:0] mem_wr_data  // memory write data
);

// ... declare a bunch of internal signals here ...

// Control logic, reg file address generation
control ctl(reset,irq,inst_addr[31],z,inst_data[31:0],
            alufn,asel,bsel,pcsel,ra2,wa,wdsel,werf,mem_we);

// PC Datapath
pc pc(clk,reset,pcsel,inst_data[15:0],jump_addr,branch_addr,
      inst_addr,pc_plus_4);

// Register file
regfile regfile(inst_data[20:16],rd1,ra2,rd2,clk,werf,wa,wdata);

// Main datapath
datapath dp(inst_data[15:0],rd1,rd2,pc_plus_4,branch_addr,mem_rd_data,
            asel,bsel,wdsel,alufn,
            wdata,mem_addr,jump_addr,mem_wr_data,z);

endmodule
```

Building Beta + Running Programs

```
% setup 6.884
% setup 6.004
% vcs -PP +lint=all +v2k beta.v
% uasm2vmh.pl self_test.uasm self_test.vmh
% ./simv +exe=self_test.vmh
*** PASSED ***

% ./mkasic.pl -v config/beta.cfg vcs
% ./mkasic.pl -v config/beta.cfg mktests
% ./mkasic.pl -v config/beta.cfg vcstest
```

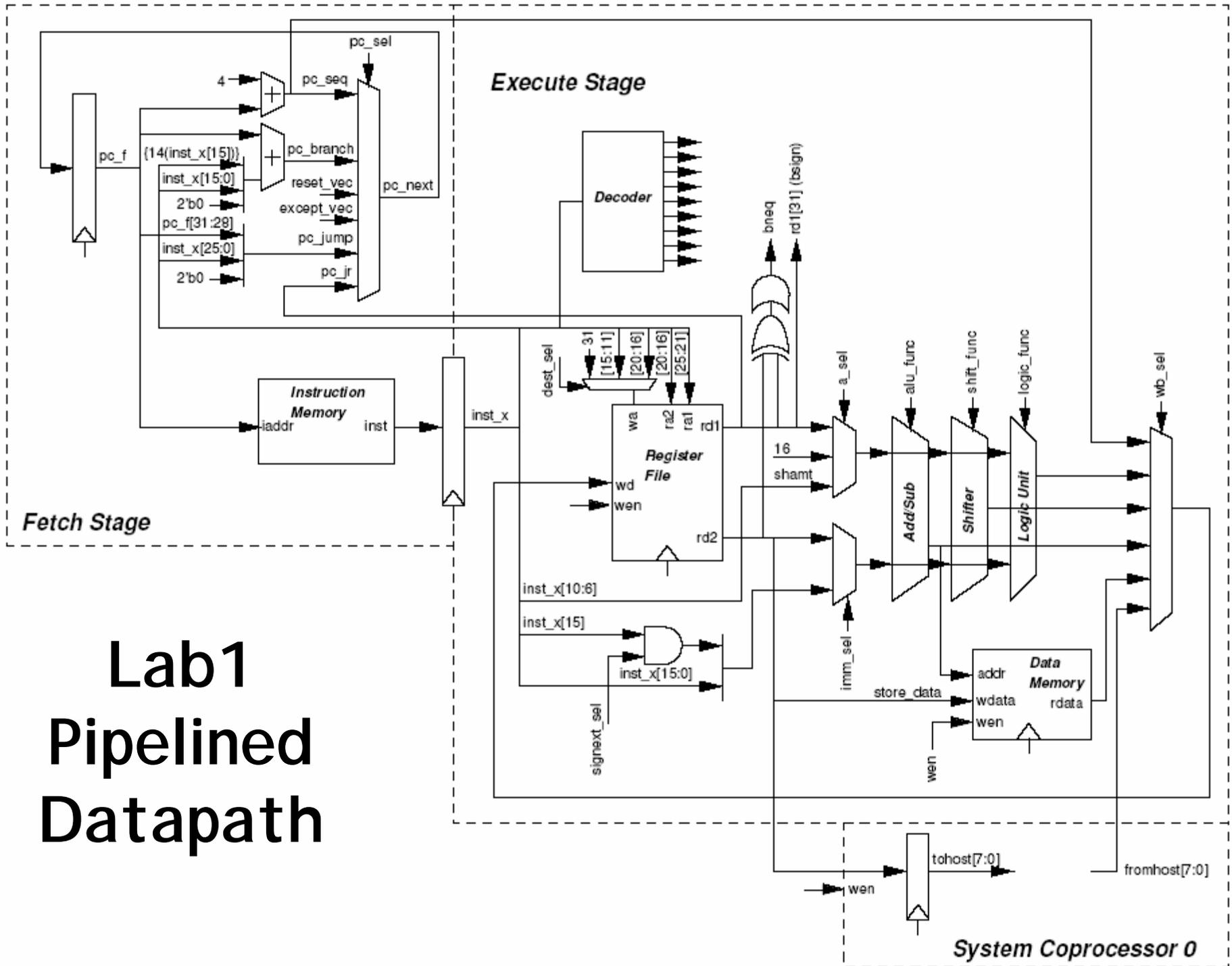
Use different config files for different design points - for example, try out different adder implementations or synthesis constraints and modular testing

Lab1

Implement a 2 stage pipelined processor which supports SMIPS ISA subset

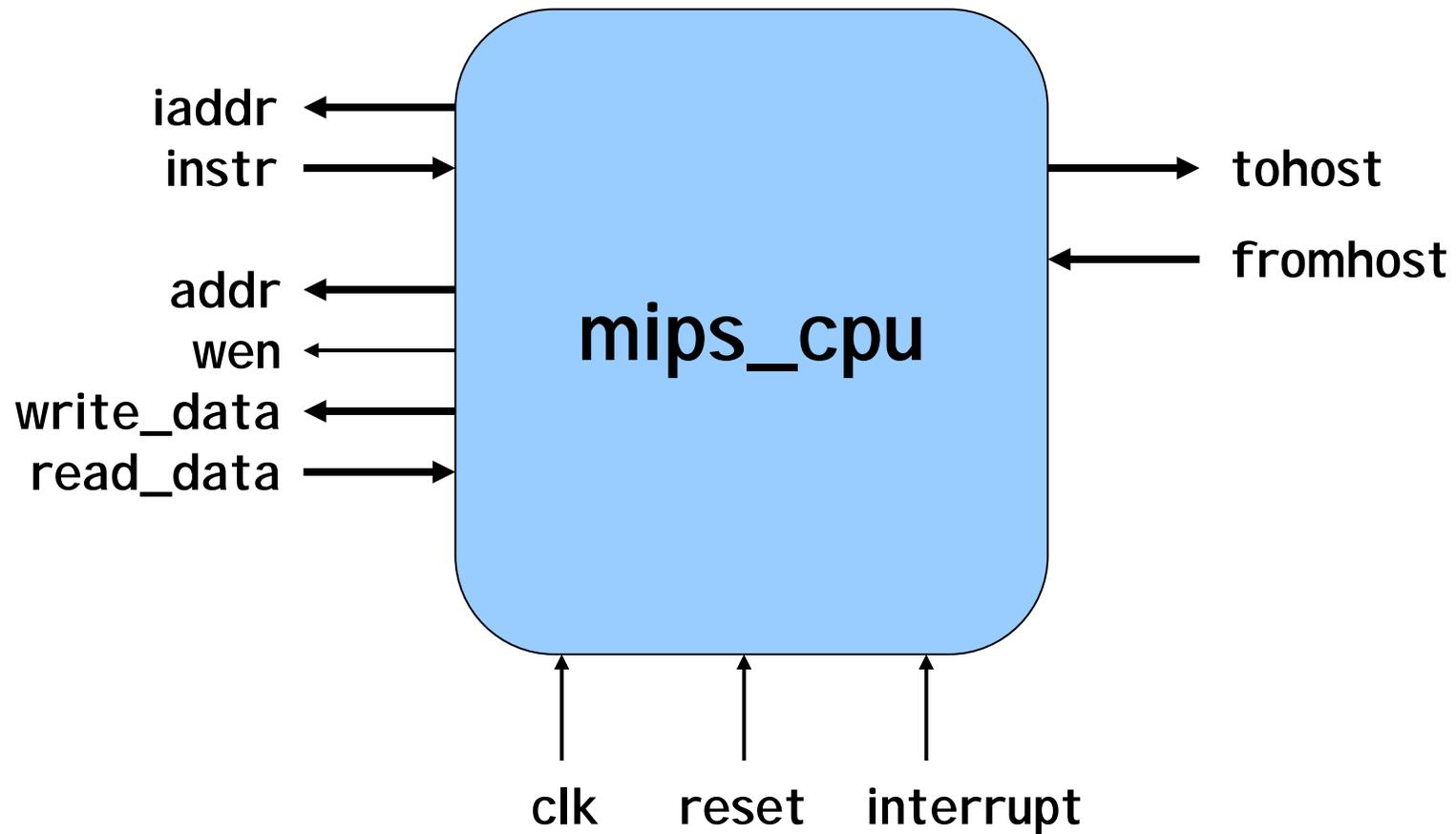
- The SMIPS Processor Spec is in the labs section
- Feel free to grab some code from the Beta implementation, but then again the Beta code is not always the best approach

31	26	25	21	20	16	15	11	10	6	5	0	
opcode		rs	rt	rd	shamt		funct					R-type
opcode		rs	rt	immediate								I-type
opcode		target										J-type
Load and Store Instructions												
100011	base	dest	signed offset									LW rt, offset(rs)
101011	base	dest	signed offset									SW rt, offset(rs)
I-Type Computational Instructions												
001001	src	dest	signed immediate									ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate									SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate									SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate									ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate									ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate									XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate									LUI rt, zero-ext-imm.
R-Type Computational Instructions												
000000	00000	src	dest	shamt	000000						SLL rd, rt, shamt	
000000	00000	src	dest	shamt	000010						SRL rd, rt, shamt	
000000	00000	src	dest	shamt	000011						SRA rd, rt, shamt	
000000	rshamt	src	dest	00000	000100						SLLV rd, rt, rs	
000000	rshamt	src	dest	00000	000110						SRLV rd, rt, rs	
000000	rshamt	src	dest	00000	000111						SRAV rd, rt, rs	
000000	src1	src2	dest	00000	100001						ADDU rd, rs, rt	
000000	src1	src2	dest	00000	100011						SUBU rd, rs, rt	
000000	src1	src2	dest	00000	100100						AND rd, rs, rt	
000000	src1	src2	dest	00000	100101						OR rd, rs, rt	
000000	src1	src2	dest	00000	100110						XOR rd, rs, rt	
000000	src1	src2	dest	00000	100111						NOR rd, rs, rt	
000000	src1	src2	dest	00000	101010						SLT rd, rs, rt	
000000	src1	src2	dest	00000	101011						SLTU rd, rs, rt	
Jump and Branch Instructions												
000010	target											J target
000011	target											JAL target
000000	src	00000	00000	00000	001000						JR rs	
000000	src	00000	dest	00000	001001						JALR rd, rs	
000100	src1	src2	signed offset									BEQ rs, rt, offset
000101	src1	src2	signed offset									BNE rs, rt, offset
000110	src	00000	signed offset									BLEZ rs, offset
000111	src	00000	signed offset									BGTZ rs, offset
000001	src	00000	signed offset									BLTZ rs, offset
000001	src	00001	signed offset									BGEZ rs, offset
System Coprocessor (COP0) Instructions												
010000	00000	dest	cop0src	00000	000000						MFC0 rt, rd	
010000	00100	src	cop0dest	00000	000000						MTC0 rt, rd	



Lab1 Pipelined Datapath

Lab1 Test Harness



Lab1 Test Harness

```
`include "mips_cpu.v"

module mips_tester;
    reg clk      = 0;           // Clock to core.
    reg reset    = 0;           // Reset signal.
    reg int_ext  = 0;           // Interrupt signal.

    reg [7:0] fromhost = 8'b0; // Control reg to CPU.
    wire [7:0] tohost;         // Control reg from CPU.

    wire [31:0] addr;          // Data address.
    wire        wen;           // Write enable.
    wire [31:0] read_data;     // Read data back to CPU.
    wire [31:0] write_data;    // Write data from CPU.
    wire [31:0] iaddr;         // Fetch address.
    wire [31:0] inst;          // Instruction.

    always #5 clk = ~clk;      // Clock generator (10ns clock period)
    reg [ 1023:0 ] exe_filename;

    mips_cpu mips(clk, reset, int_ext, fromhost, tohost,
                  addr, wen, write_data, read_data, iaddr, inst);

    memory mem(clk, addr, wen, write_data, read_data, iaddr, inst);

    ...
endmodule
```

Lab1 Test Harness

```
...

initial
begin
    $vcdpluson(0);

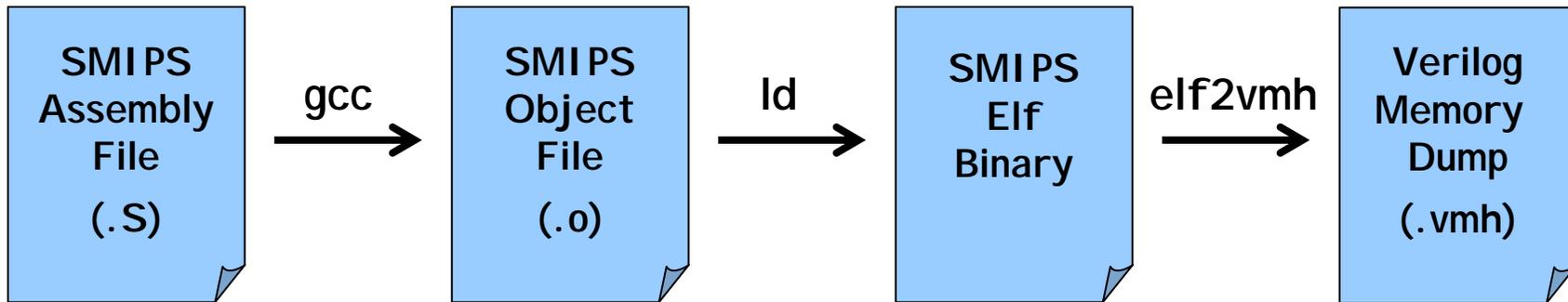
    if( $value$plusargs( "exe=%s", exe_filename ) )
        $readmemh( exe_filename, mem.m );
    else begin
        $display( "ERROR: No executable specified! (use +exe=<filename>)" );
        $finish;
    end

    #0 reset = 1; #16 reset = 0; // Strobe reset
    #10000; // Safety net to catch infinite loops
    $display("*** FAILED ***");
    $finish;
end

// Stop running when tohost changes.
always @(tohost) begin
    case (tohost)
        8'd0: ;
        8'd1: begin $display("*** PASSED ***"); #20 $finish; end
        default: begin $display("*** FAILED ***"); #20 $stop; end
    endcase
end

endmodule
```

Building Tests with smips-testbuild



```
% smips-testbuild -vmh -smips self_test.S -o self_test.vmh
```

```
% mips2stage +exe=self_test.vmh
```

```
*** PASSED ***
```

```
% ./mkasic.pl config/mips2stage.cfg mktests
```

```
% ./mkasic.pl config/mips2stage.cfg vcstest
```

```
* Running tests on mips2stage/vcs/mips2stage simulator
```

```
+ [ PASSED ] simple_test.vmh
```

```
+ [ PASSED ] self_test.vmh
```

Final Notes

Tools must be run either on the MIT server/Linux or CAG

- Lab is open 24 hours a day
- smips-testbuild should be working this afternoon
- CAG toolflow not quite setup yet

Lab is due at the start of class on Friday, Feb 18

- Write lots of tests (self_test is not enough!)
- We will email more information on exact collection procedure

Online resources

- Links to various Verilog whitepaper and references
- First chapter of Sutherland's logical effort book

Office hours

- Tuesday + Thursday, 5:30pm – 7:00pm