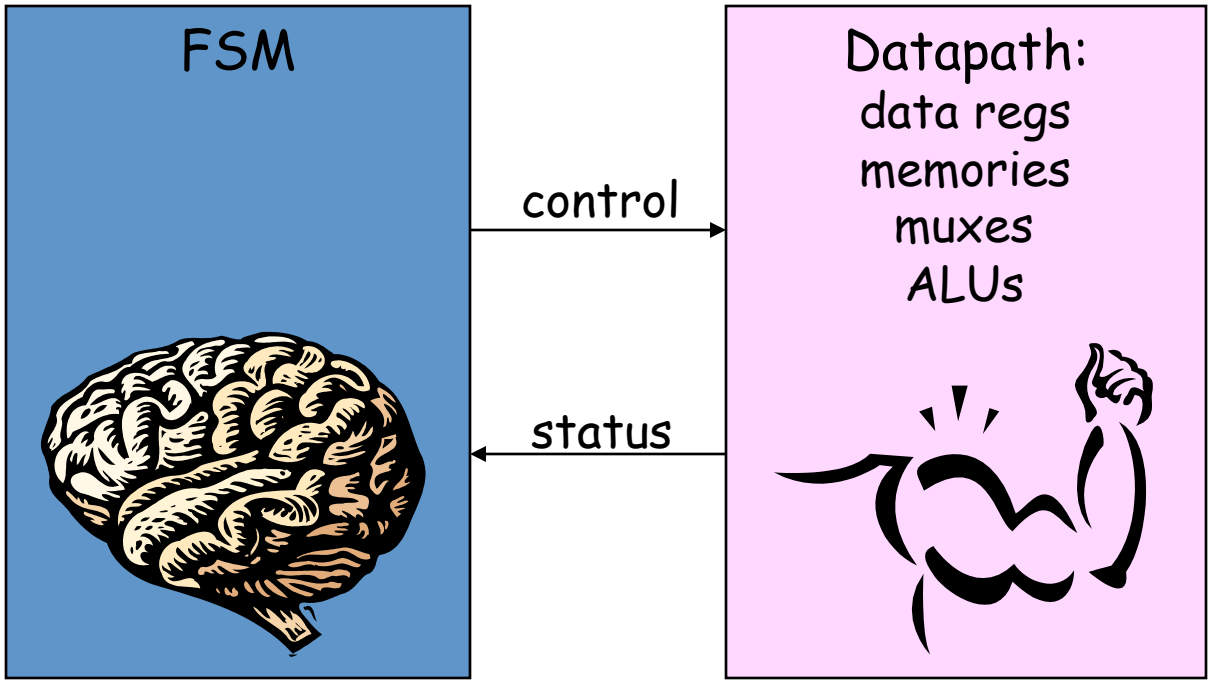
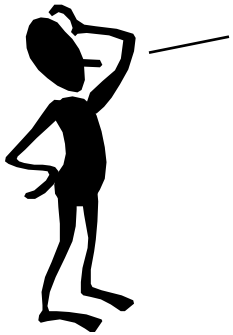
Programmable Control Logic

- Microsequencers
- Example: building the micro8
- Hardware/Software tradeoff

Lab #5 due Thursday, project abstract next Monday

Digital Systems = FSMs + Datapath

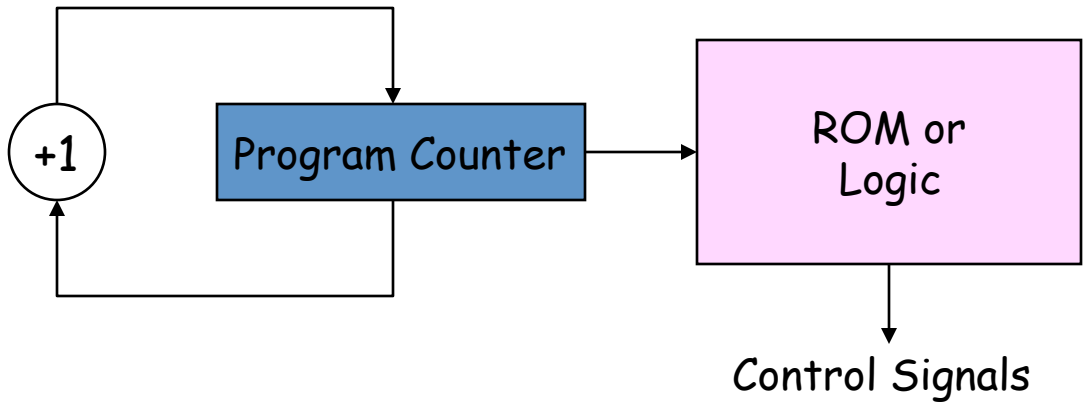




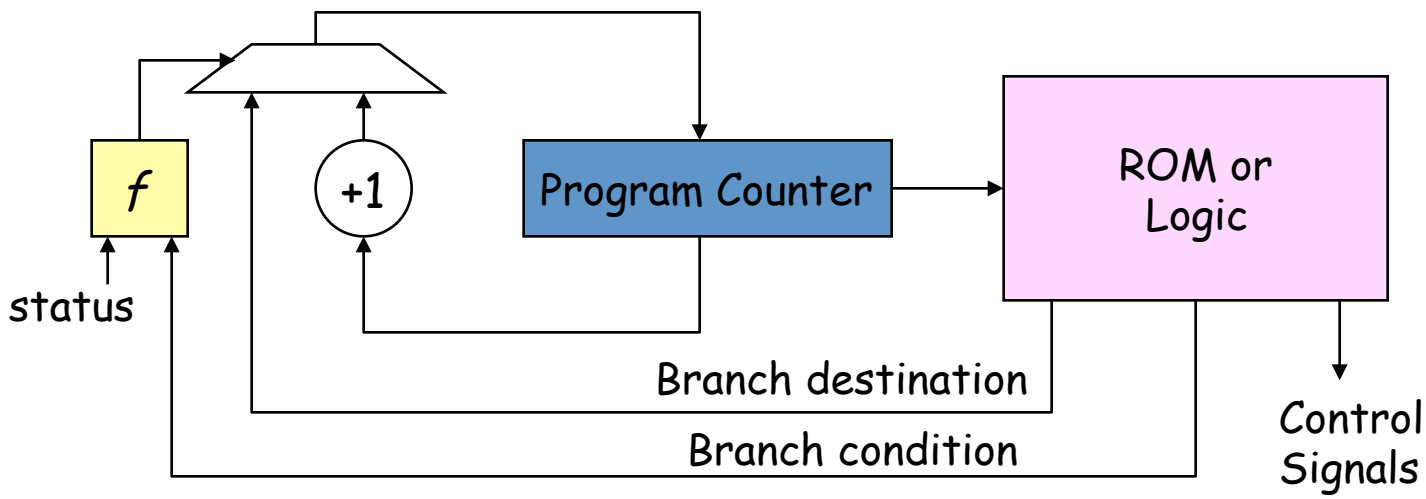
But what if my FSM has hundreds or thousands of states? That's a BIG case statement!

Microsequencers

Step 1: use a counter for the state

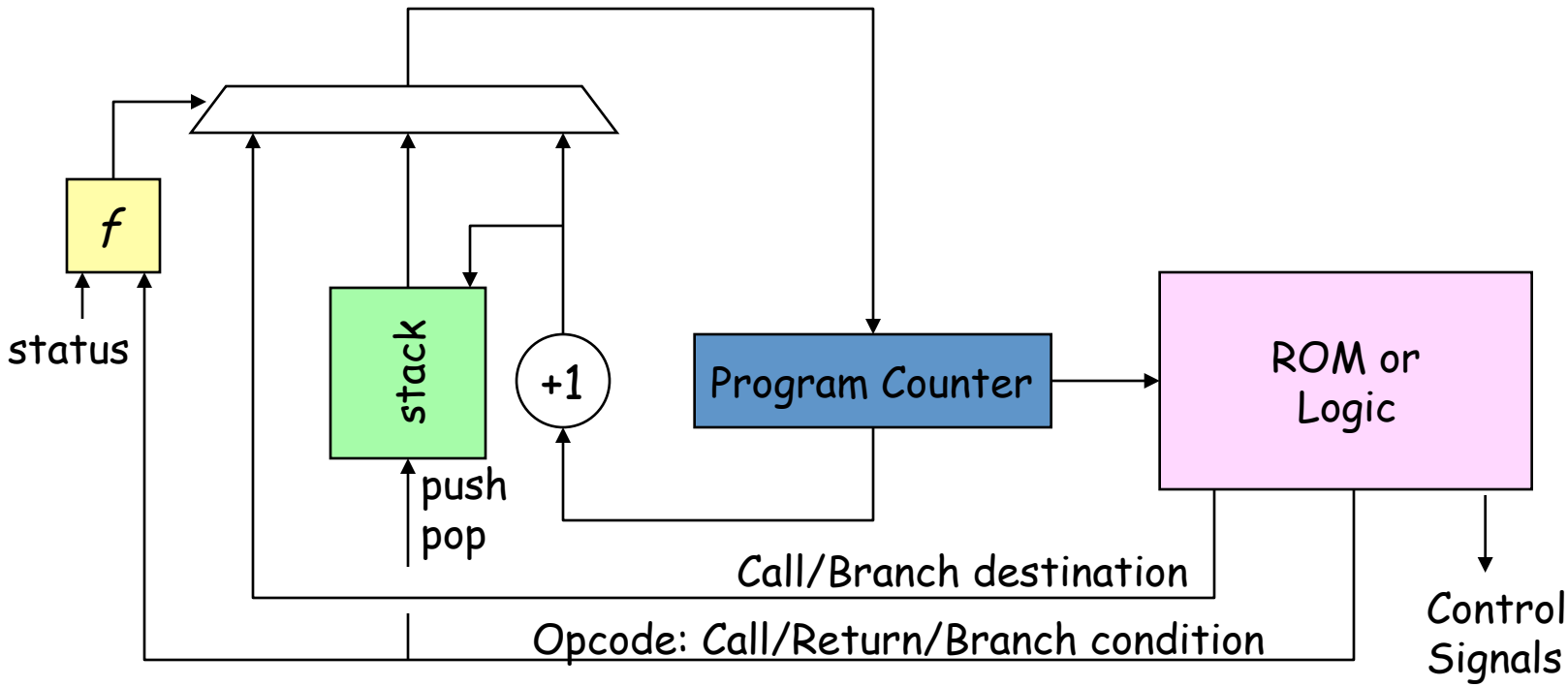


Step 2: add a conditional branch mechanism



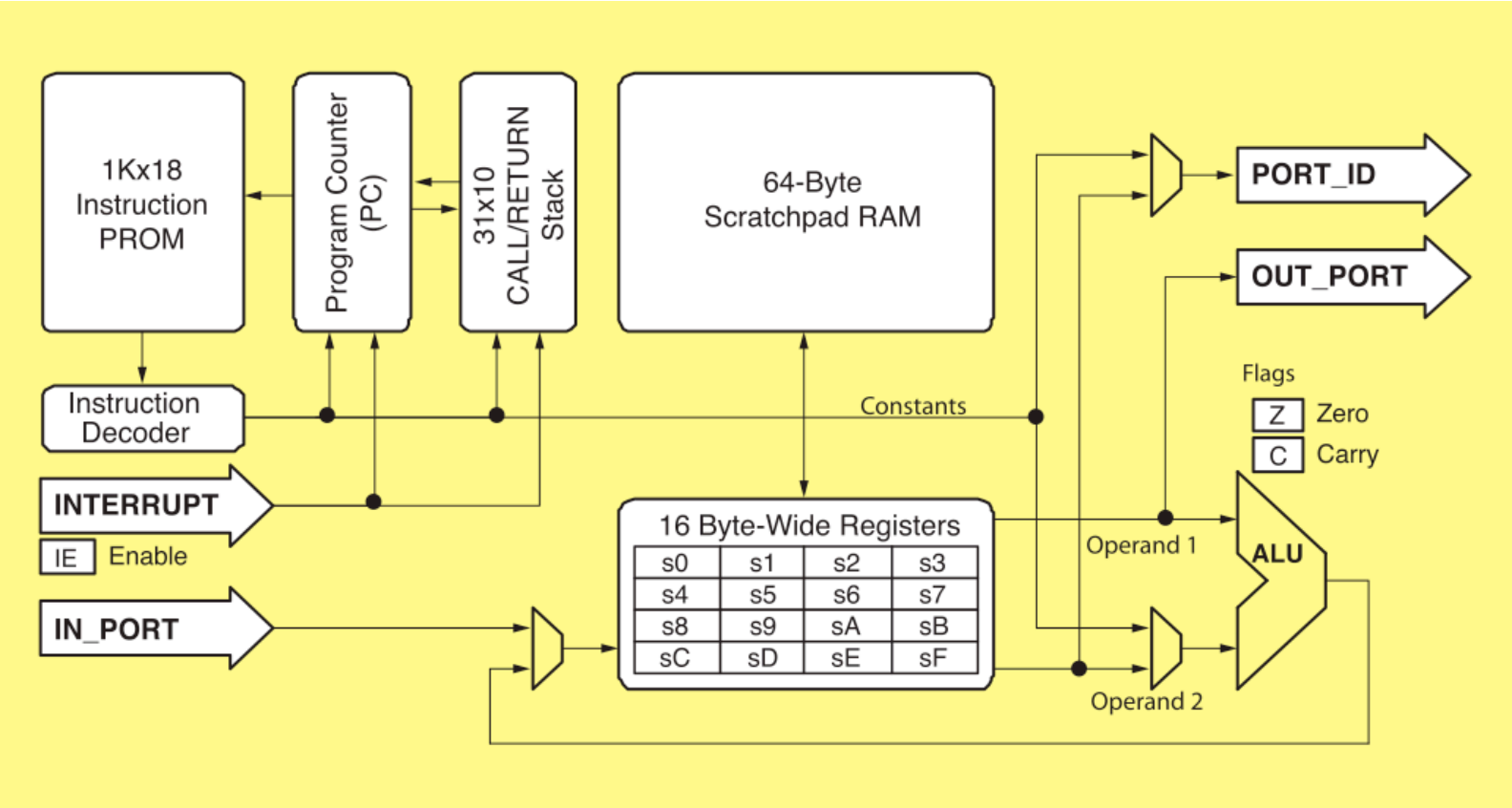
Microsequencers (cont'd.)

Step 3: add a (small) call/return stack to support "subroutines"



Subroutine call: select destination as new PC, push PC+1 onto stack
Subroutine return: select top of stack as new PC, pop stack

Xilinx PicoBlaze™



- 8-bit data path
- internal memory
  - 1K 18-bit insts
  - 31-locn stack
  - 16 8-bit registers
  - 64-locn local mem
- external 8-bit ports
  - 256 in, 256 out
- small: only 96 slices + inst mem
- fast: 2 cycles/inst [IF→EXE]
  200MHz (100MIPS) on labkit

PicoBlaze Instructions

Program Control

JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa

CALL aaa
CALL Z,aaa
CALL NZ,aaa
CALL C,aaa
CALL NC,aaa

RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC

Input/Output

INPUT sX,pp
INPUT sX,(sY)
OUTPUT sX,pp
OUTPUT sX,(sY)

Logical

LOAD sX,kk
AND sX,kk
OR sX,kk
XOR sX,kk
TEST sX,kk
LOAD sX,sY
AND sX,sY
OR sX,sY
XOR sX,sY
TEST sX,sY

Shift and Rotate

SR0 sX
SR1 sX
SRX sX
SRA sX
RR sX
SL0 sX
SL1 sX
SLX sX
SLA sX
RL sX

Arithmetic

ADD sX,kk
ADDCY sX,kk
SUB sX,kk
SUBCY sX,kk
COMPARE sX,kk
ADD sX,sY
ADDCY sX,sY
SUB sX,sY
SUBCY sX,sY
COMPARE sX,sY

Storage

FETCH sX,ss
FETCH sX,(sY)
STORE sX,ss
STORE sX,(sY)

Interrupt

RETURNI ENABLE
RETURNI DISABLE
ENABLE INTERRUPT
DISABLE INTERRUPT

All instructions execute in 2 clock cycles

Conditional and unconditional flow of control

8-bit ALU:
<r1> <op>= <r2>
<r1> <op>= const

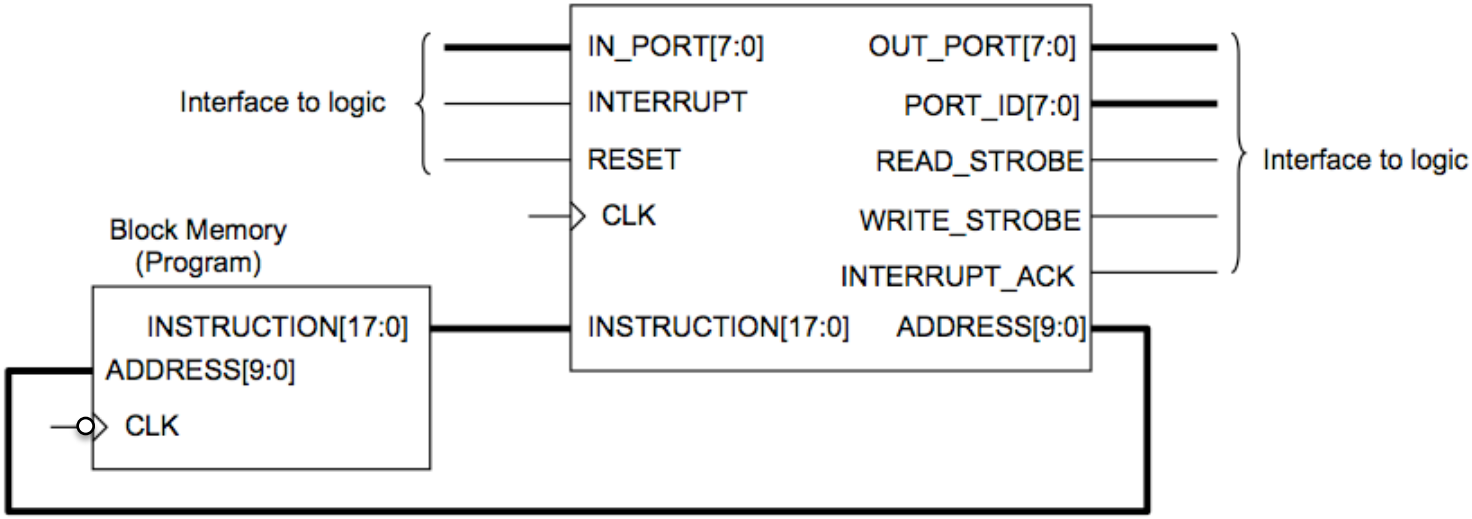
Access to 64-locn local mem

Access to external devices

Interrupt management

Example: building the micro8

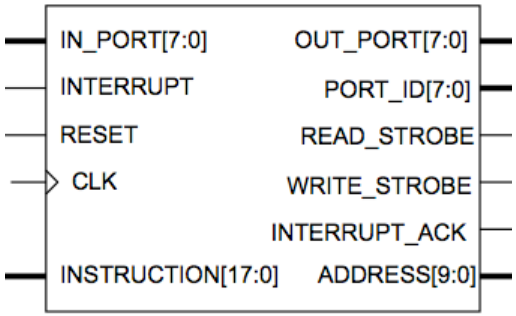
- Let's build a PicoBlaze-compatible clone using standard Verilog (the Xilinx implementation is a netlist of Xilinx CLB macros).
- Approach: incremental featurism
  - Go through instruction set adding datapath elements and control logic necessary for each instruction
  - Goal: see how close we can come to 96 slices and 100MHz…



Preliminaries

- Instruction formats: 18-bits w/ 5-bit opcode
  - `ooooo ccc aaaaaaaaaa` condition, 10-bit address
  - `ooooo 1 xxxx yyyy 0000` two registers (R[x] = R[x] op R[y])
  - `ooooo 0 xxxx kkkkkkkk` register, 8-bit const (R[x] = R[x] op k)
- So
  - Dispatch on opcode of current inst., i.e., `case (inst[17:13])` to determine what control signals should be
  - Use a three port register file
    - 16 locations => 4-bit addresses
    - 2 combinational read ports, addresses of xxxx and yyyy
    - 1 clocked write port written at end of cycles (use xxxx as addr)
- Make each instruction execute in one clock cycle
  - Instruction latched every ↑clk, state changes (eg, register writes, status bits) latched at next ↑clk
  - Start fetch of next instruction from BRAM on ↓clk, so instruction data available at ↑clk

Initial Verilog template




```
module micro8(
  input wire clk,reset,interrupt,
  input wire [17:0] instruction,
  input wire in_port[7:0],
  output wire [7:0] address,
  output wire [7:0] port_id,out_port,
  output wire read_strobe,writestrobe,
  output wire interrupt_ack
);
  reg [17:0] inst;
  always @(posedge clk) inst <= instruction;

  // datapath elements here…

  // control logic
  always @* begin
    // default ctl sig values here…
    …
    if (reset) …
    else (irq) …
    else
      case (inst[17:13])
      …
      endcase
  end
endmodule
```

Sequential Instruction Fetch

- The program counter (PC) register holds the address of the current instruction
- Usually the next instruction comes from location PC+1 in the instruction memory (ie, the next instruction in the sequence)

```
// address of current instruction
reg [9:0] pc;
wire [9:0] pc_inc = pc + 1;

// multiplexer for selecting address
// of next inst
reg [1:0] pcsel;
localparam pc_next = 2'b00;
…
always @*
  case (pcsel)
    pc_next: address = pc_inc;
    default: address = 10'hxxx;
  endcase
always @(posedge clk) pc <= address;
```

```
// control logic
always @* begin
  // default ctl sig values here
  pcsel = pc_next;
  …
end
```

JUMP: transfer of control

- JUMP chooses another instruction as the next to be executed
  - Unconditional jumps always transfer control
  - Conditional jumps transfer only if condition is true, otherwise have no effect
  - Conditions Z, ~Z, C, ~C

```
// address of current instruction
reg [9:0] pc;
wire [9:0] pc_inc = pc + 1;

// multiplexer for selecting address
// of next inst
reg [1:0] pcsel;
localparam pc_next = 2'b00;
localparam pc_aaa = 2'b01;
…
always @*
  case (pcsel)
    pc_next: address = pc_inc;
    pc_aaa:  address = inst[9:0];
    default: address = 10'hxxx;
  endcase
always @(posedge clk) pc <= address;
```

```
reg cond;  // true if we should branch
always @*
  casez (inst[12:10])
    3'b0??: cond = 1b'1;
    3'b100: cond = z;
    3'b101: cond = ~z;
    3'b110: cond = c;
    3'b111: cond = ~c;
    default: cond = 1'bx;
  endcase

// control logic
always @* begin
  // default ctl sig values
  pcsel = pc_next;
  …
  case (inst[17:13])
    …
    5'b11010: // JUMP
              if (cond) pcsel = pc_aaa;
    …
  endcase
end
```

PC Stack for subroutines

- CALL: if (cond) pushes PC+1 onto PC stack, jumps to aaa
- RETURN: if (cond) jumps to addr @ top of stack, pops stack

```
 // 31 location PC stack
(* ram_style = "distributed" *)
reg [9:0] stack[31:0]; // the stack itself
reg [4:0] sp,sp_inc;   // sp points to top of stack
reg [1:0] stsel;
localparam st_nop = 2'b00;
localparam st_push = 2'b01;
localparam st_pop = 2'b10;
always @*              // select increment to sp this cycle
  case (stsel)
    st_nop:  sp_inc = 5'h00;
    st_push: sp_inc = 5'h01;
    st_pop:  sp_inc = 5'h1F;
    default: sp_inc = 5'hxx;
  endcase
wire [4:0] sp_next = sp + sp_inc;

always @(posedge clk) begin
  sp <= reset ? 5'h00 : sp_next;
  if (stsel == st_push) stack[sp_next] <= pc_inc;
end

wire [9:0] tos = stack[sp];   // async read of top of PC stack
```

CALL and RETURN

```
// address of current instruction
reg [9:0] pc;
wire [9:0] pc_inc = pc + 1;

// multiplexer for selecting address
// of next inst
reg [1:0] pcsel;
localparam pc_next = 2'b00;
localparam pc_aaa = 2'b01;
localparam pc_tos = 2'b02;
…
always @*
  case (pcsel)
    pc_next: address = pc_inc;
    pc_aaa:  address = inst[9:0];
    pc_tos:  address = tos;
    default: address = 10'hxxx;
  endcase
always @(posedge clk) pc <= address;
```

```
// control logic
always @* begin
  // default ctl sig values
  pcsel = pc_next;
  stsel = st_nop;
  …
  case (inst[17:13])
    …
    5'b10101: // RETURN
              if (cond) begin
                pcsel = pc_tos;
                stsel = st_pop;
              end
    5'b11000: // CALL
              if (cond) begin
                pcsel = pc_aaa;
                stsel = st_push;
              end
    …
  endcase
end
```

Executing R[x] = R[x] op R[y]/kkk

- Operands come from register file or instruction
  - Two registers from register file
  - One register from register file and 8-bit constant from inst.
- Usually implement different datapath units for each class of operations
  - Adder/subtractor (ADD,SUB,ADDCY,SUBCY,COMPARE)
  - Boolean operations (AND, OR, XOR, TEST)
  - Rotate Operations (shift left/right with choice of shifted-in data)
- Use multiplexer to select which unit supplies data to be written back into register file
- Condition codes (Z,C) are also set by instruction execution and they have their own multiplexers to choose where their value comes from.
- Instruction decode supplies necessary control signals
  - Control signals for each functional unit
  - Sel signals for each mux, write enables for regfile/cc's

Register File (2 reads, 1 write)

```
// dual-port 16-location register file
(* ram_style = "distributed" *)
reg [7:0] regfile[15:0];
wire [7:0] wdata;
reg werf;
always @(posedge clk)
 if (werf) regfile[inst[11:8]] <= wdata;
wire [7:0] sx = regfile[inst[11:8]];
wire [7:0] sy = regfile[inst[7:4]];

// regfile write data mux
reg [7:0] wdatax;
reg [2:0] wsel;
localparam w_bool = 3'b000;
localparam w_rot = 3'b001;
localparam w_sp = 3'b010;
localparam w_in = 3'b011;
localparam w_addsub = 3'b100;
always @*
  case (wsel[1:0])
    2'b00: wdatax = bool;
    2'b01: wdatax = rotate;
    2'b10: wdatax = spdata;
    2'b11: wdatax = in_port;
    default: wdatax = 8'hxx;
  endcase
assign wdata = wsel[2] ? addsub : wdatax;
```

Data to be written into regfile comes from 5 places:

- adder unit
- boolean unit
- rotate unit
- 64-location scratchpad
- input port

Adder Unit

```
reg z,c;  // condition codes

// second ALU operand: register or constant
wire [7:0] b = inst[12] ? sy : inst[7:0];

// adder -- we only want one, so make it explicitly here!

// subtracts of B implemented as adds of ~B+1
wire [7:0] addsub_b = inst[14] ? ~b : b;

// cin should 0 for adds, 1 for subtracts, C condition code for ADDCY,SUBCY
wire addsub_cin = inst[13] ? c : inst[14];

wire [7:0] addsub;
wire addsub_cout;
assign {addsub_cout,addsub} = sx + addsub_b + addsub_cin;
wire zaddsub = ~|addsub;
```

Boolean & Rotate Units

```
// boolean ops
reg [7:0] bool;
always @*
  case (inst[14:13])
    2'b00: bool = b;      // LOAD
    2'b01: bool = sx & b; // AND, TEST
    2'b10: bool = sx | b; // OR
    2'b11: bool = sx ^ b; // XOR
    default: bool = 8'hxx;
  endcase
wire zbool = ~|bool;
```

```
// rotate
reg [7:0] rotate;
always @* begin
  if (inst[3]) begin  // SR
    rotate[6:0] = sx[7:1];
    case (inst[2:1])
      2'b00: rotate[7] = c;       // SRA
      2'b01: rotate[7] = sx[7];   // SRX
      2'b10: rotate[7] = sx[0];   // RR
      2'b11: rotate[7] = inst[0]; // SR0,SR1
      default: rotate[7] = 1'bx;
    endcase
  end
  else begin  // SL
    rotate[7:1] = sx[6:0];
    case (inst[2:1])
      2'b00: rotate[0] = c;       // SLA
      2'b01: rotate[0] = sx[0];   // SLX
      2'b10: rotate[0] = sx[7];   // RL
      2'b11: rotate[0] = inst[0]; // SL0,SL1
      default: rotate[0] = 1'bx;
    endcase
  end
End
wire zrot = ~|rotate;
```

Control Logic

```
// control logic
always @* begin
  // default ctl sig values
  werf = 1'b0;   // no write into regfile
  wsel = 3'bxxx; // so don't care what write data is!
  csel = c_nop;
  zsel = z_nop;  // assume no changes to Z,C
  …
  case (inst[17:13])
    …
   5'b01100: // ADD
             begin
               werf = 1'b1;     // write result into regfile
               wsel = w_addsub; // select adder unit to supply data
               csel = c_addsub; // C condition code comes from adder
               zsel = z_addsub; // Z condition code comes from adder
             end

  endcase
end
```

Strategy: set default values for all control signals and then have each instruction override the defaults as appropriate.

Other details…

- Handling reset, interrupts
  - Reset sets PC to 10'h000
  - Interrupt sets PC to 10'h3FF
- `INPUT xxxx,pppppppp` and `OUTPUT xxxx,pppppppp` instructions
  - 8-bit PORT-ID `pppppppp` selects I/O device
  - Read data is stored into register xxxx
  - Write data comes from register xxxx
- 64-location local memory (aka scratchpad)
  - Address comes from instruction (kkkkkk) or register yyyy
  - `FETCH xxxx,…` loads data into register xxxx
  - `STORE xxxx,…` write data from register xxxx
- Result: ~100MHz, ~150 slices (i.e., fast but small!)

Hardware/Software Tradeoff

- What should be done in hardware? *As little as possible…*
  - High-speed processing (e.g., video)
  - Low-level device interfaces (I/O, memories)
- What should be done in software? *As much as you can…*
  - User interface (e.g., game logic)
  - Configuration of HW (e.g., sprite position)
- Example: 6ES -- 6.111 Entertainment System!
  - Micro8 as main controller for game logic
  - interfaces for switch, buttons, leds, … (appear as ports on micro8)
    - 60Hz interrupt to micro8
  - 4 copies of sprite device, registers determine properties:
    - Location on screen: xlo,xhi,ylo,yhi
    - Dimensions of blob: w,h
    - Color: rrrgggbb (8'h00=transparent)
  - Every 1/60th second code reads switches, moves sprites around screen, detects collisions, etc.