CSCE 3953 System Synthesis and Modeling

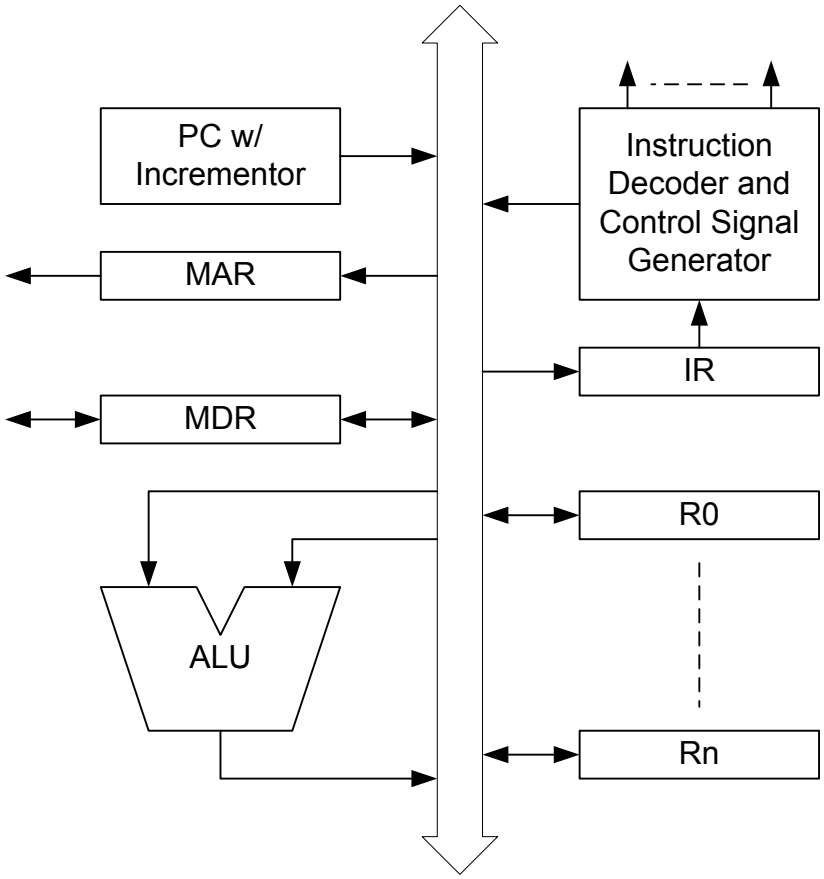
Final Project : Simple RISC Microprocessor

Instructor: Dr. Jia Di

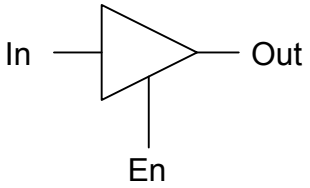
Project Overview

- Design, synthesize, and simulate a simple microprocessor in Verilog
- Combinational and sequential logic circuits
- Hardware and software
- Hardwired control
- Bus architecture

Overall Structure



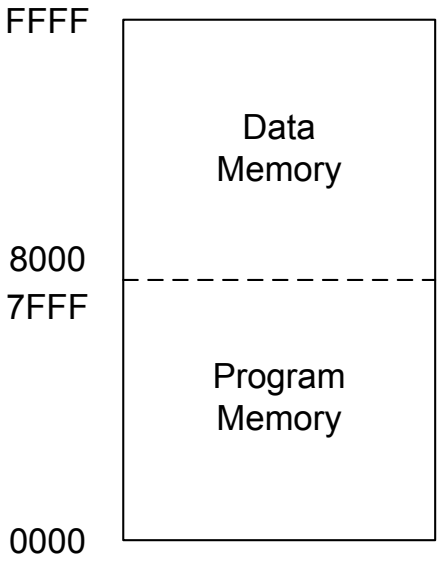
Tri-state buffers are needed for bus drivers



System Specification

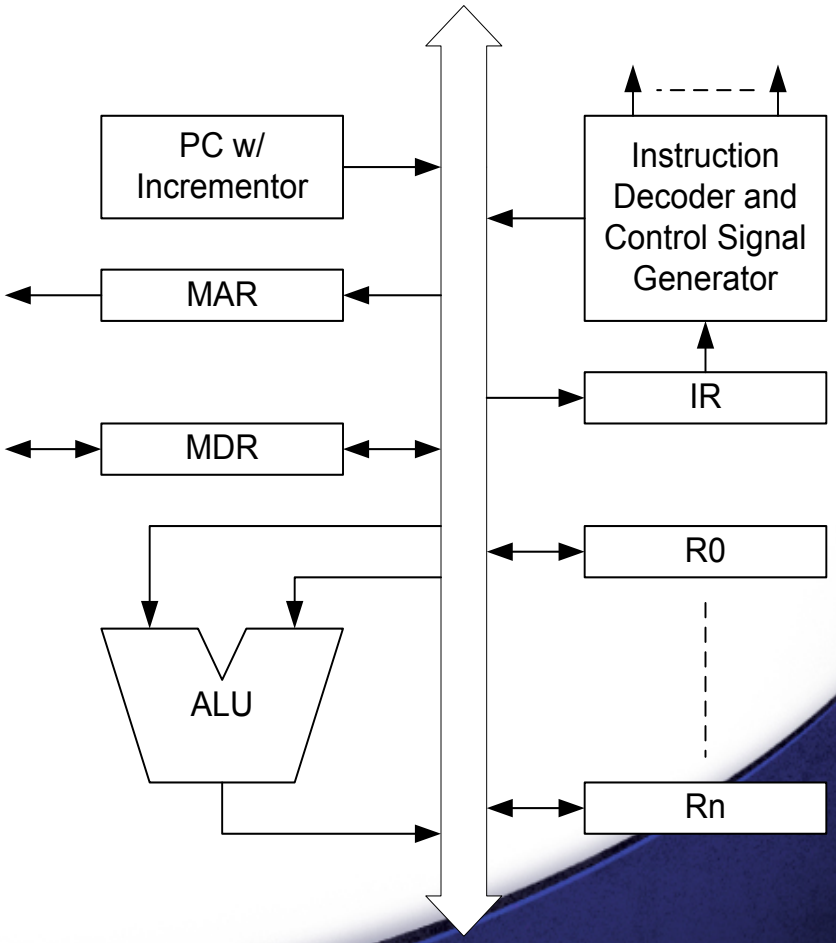
- Single thread
- 16-bit Data bus
- 16-bit Address bus
- Program memory and data memory are mapped into single address space
- Primary inputs: data (16-bit), clock, reset, MFC
- Primary outputs: data (16-bit), address (16-bit), R/$\overline{W}$, EN
- Virtual memory module

Virtual Memory Map



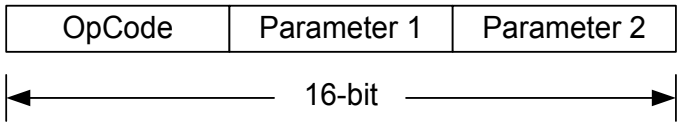
Components and Control Signals

- IR: 16-bit, “In”, “*Out*”
- GPR: 16-bit, “In”, “Out”
- PC: 16-bit, “Out”, “Inc”
- MAR: 16-bit, “In”, “Out”
- MDR: 16-bit, “In-uP”, “In-Mem”, “Out-uP”, “Out-Mem”
- ALU: “In1”, “In2”, “Out”, Add, Sub, Not, And, Or, Xor, Xnor



Instruction Set

- Mov Ri, Rj
- Add Ri, Rj
- Addi Ri, #
- Sub Ri, Rj
- Subi Ri, #
- Not Ri
- And Ri, Rj
- Or Ri, Rj
- Xor Ri, Rj
- Xnor Ri, Rj
- Movi Ri, #
- Load (Ri), Rj
- Store Ri, (Rj)

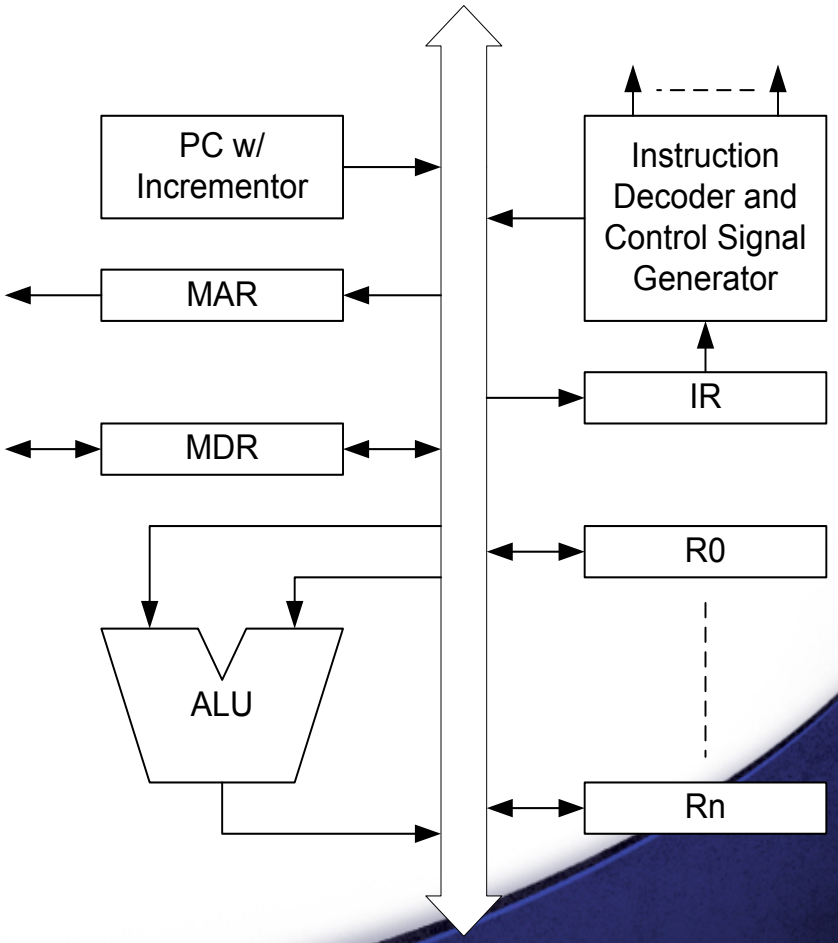


- OpCode: 4 bits
- Parameter 1: 6 bits
- Parameter 2: 6 bits

Hardwired Control Example

- Add R1, R2

1. PC Out, MAR In
2. MAR Out, R/$\overline{W}$ = 1, EN
3. Wait for MFC
4. MDR In-Mem
5. MDR Out-uP, IR In
6. R1 Out, ALU Input-Reg1 In, PC Inc
7. R2 Out, ALU Input-Reg2 In
8. Add, ALU Output-Reg In
9. ALU Output-Reg Out, R1 In

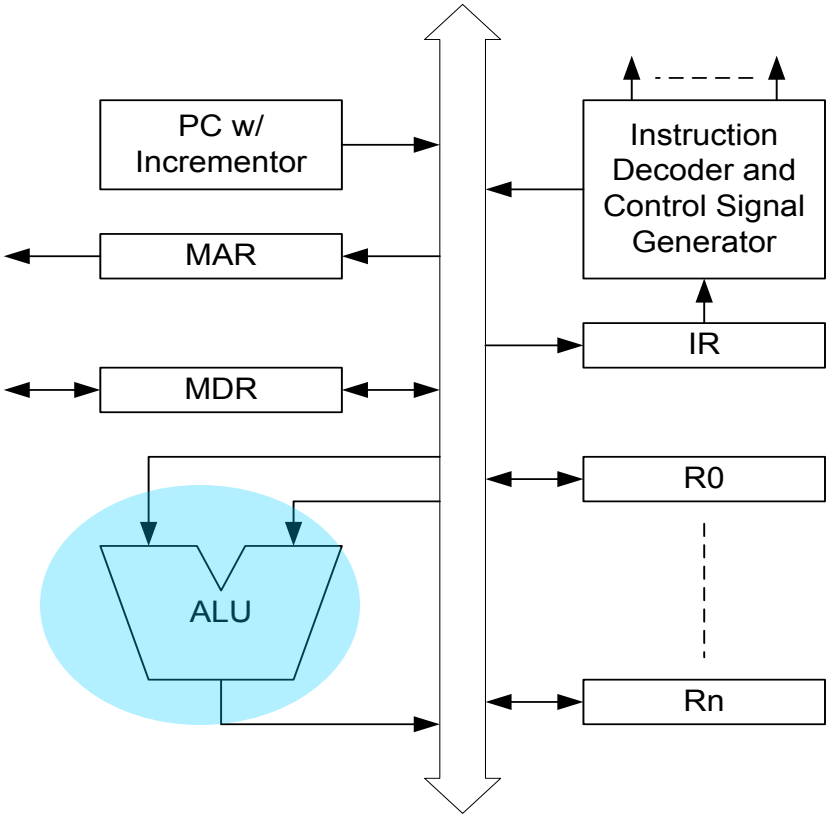


Flexibility

- Additional instructions/addressing modes
- Other hardware blocks
- # of registers
- # of clock cycles for each operation

Divide-and-Conquer #1

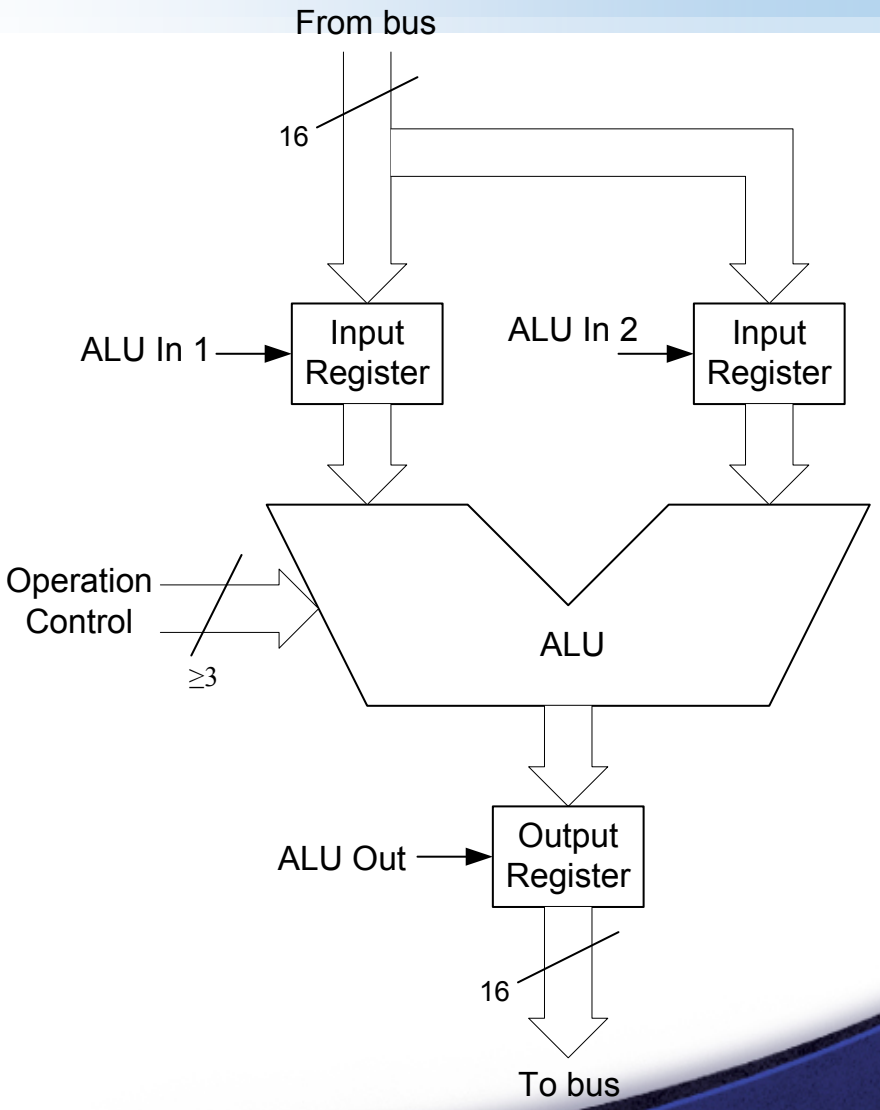
ALU


PC w/
Incrementor
MAR
MDR
ALU
Instruction
Decoder and
Control Signal
Generator
IR
R0
Rn

ALU

Add
Sub
Not
And
Or
Xor
Xnor

Unsigned binary
No overflow detection



Final Project Step #1: ALU

- Design an ALU following the specification
- Simulate in ModelSim to ensure the functionality using a testbench that covers everything
- Synthesize the designed ALU in Design Vision
- Simulate the synthesized netlist
- Write a simple project report (code, testbench, conclusion)

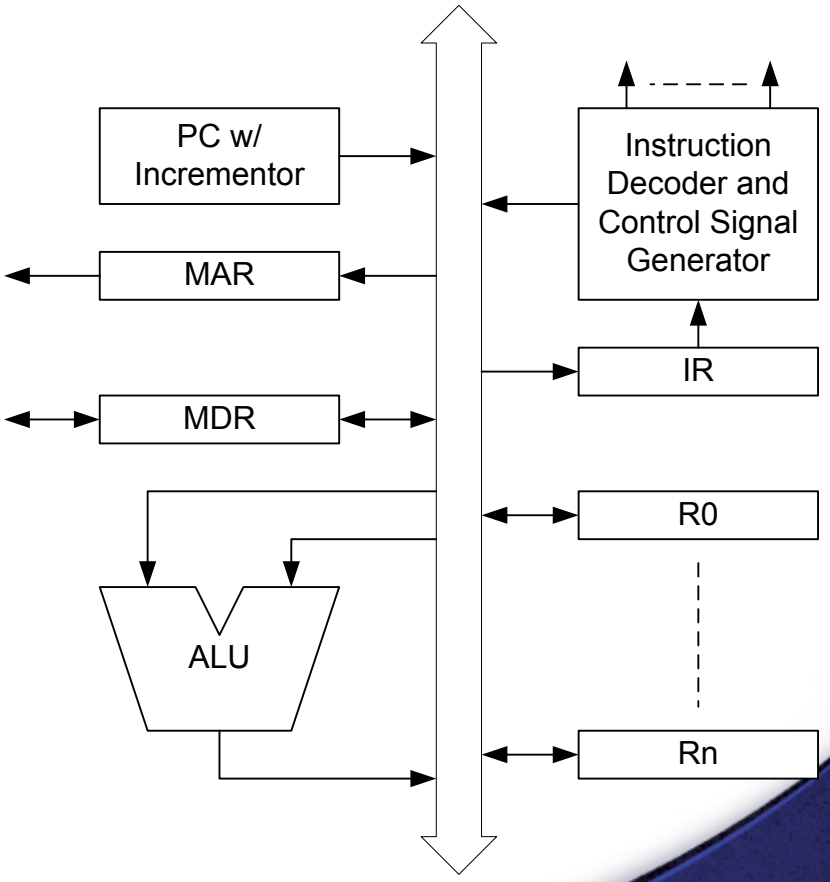
- Due date: Oct. 31

Divide-and-Conquer #2

Registers and Program Counter

Registers

- All registers are D-flip flops
- Edge-sensitive
- All registers driving the bus need to have tri-state buffers at their outputs
- MAR is uni-directional
- MDR is bi-directional
  - Use one register to handle each direction



Program Counter

- 16-bit counter
- Need tri-state buffer at the output
- Inputs include reset and increment control signal

Final Project Step #2: Registers and PC

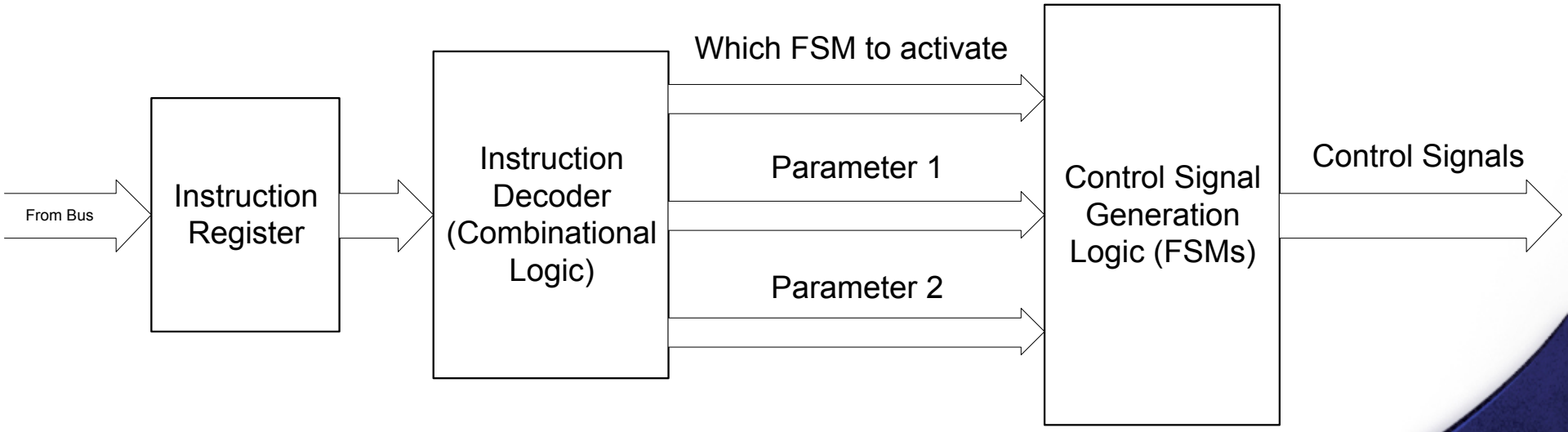
- Design registers and PC following the specification
- Simulate in ModelSim to ensure the functionality using a testbench that covers everything
- Synthesize the design in Design Vision
- Simulate the synthesized netlist
- Write a simple project report (code, testbench, conclusion)

- Due date: next Monday (Nov. 7)

Divide-and-Conquer #3

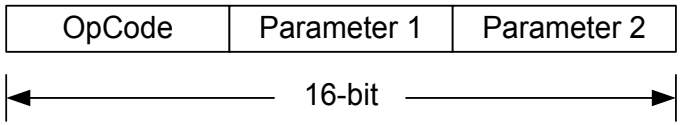
Instruction Decoder

Architecture



Instruction Set

- Mov Ri, Rj
- Add Ri, Rj
- Addi Ri, #
- Sub Ri, Rj
- Subi Ri, #
- Not Ri
- And Ri, Rj
- Or Ri, Rj
- Xor Ri, Rj
- Xnor Ri, Rj
- Movi Ri, #
- Load (Ri), Rj
- Store Ri, (Rj)



- OpCode: 4 bits
- Parameter 1: 6 bits
- Parameter 2: 6 bits

opcode	Parameter 1	Parameter 2

MOV 0100

R1 000001

R2 000011

MOV R1, R2 0100 000001 000011

opcode	Parameter 1	#

MOVI 0101

R2 000011

MOVI R2, #3A 0101 000011 111010

Final Project Step #3: IR and ID

- Design the instruction registers and decoder following the specification
- Simulate in ModelSim to ensure the functionality using a testbench
- Synthesize the design in Design Vision
- Simulate the synthesized netlist
- Write a simple project report (instruction/register-to-machine code mapping, design code, testbench, conclusion)

- Due date: this Friday (Nov. 11)

Divide-and-Conquer #4

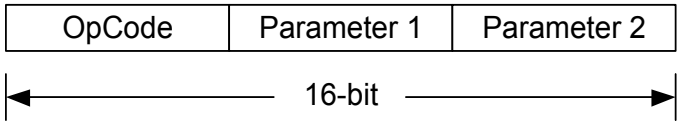
FSMs

FSMs

- For each instruction, a sequence of control signals needs to be generated – finite state machine
- At each active clock edge, some control signals need to switch to either logic 1 or logic 0
- Theoretically, each instruction has its own FSM
- However, some are very similar so sharing is possible

Instruction Set

- Mov Ri, Rj
- Add Ri, Rj
- Addi Ri, #
- Sub Ri, Rj
- Subi Ri, #
- Not Ri
- And Ri, Rj
- Or Ri, Rj
- Xor Ri, Rj
- Xnor Ri, Rj
- Movi Ri, #
- Load (Ri), Rj
- Store Ri, (Rj)

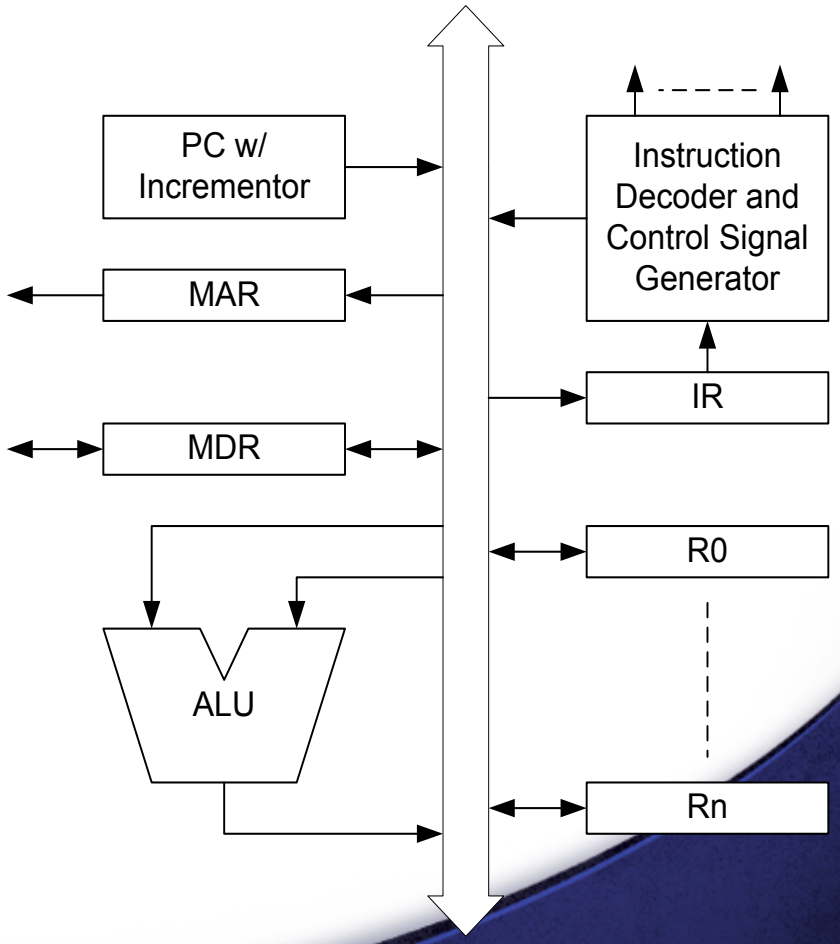


- OpCode: 4 bits
- Parameter 1: 6 bits
- Parameter 2: 6 bits

Detailed Hardwired Control Step Example

- Add R1, R2

1. PC Out, MAR In
2. *MAR Out*, R/$\overline{W}$= 1, EN (PC Out and MAR In are deasserted)
3. Wait for MFC
4. MDR In-Mem (*MAR Out* is deasserted)
5. MDR Out-uP, IR In (MDR In-Mem and EN are deasserted)
6. R1 Out, ALU Input-Reg1 In, PC Inc (MDR Out-uP and IR In are deasserted)
7. R2 Out, ALU Input-Reg2 In (R1 Out, ALU Input-Reg1 In, and PC Inc are deasserted)
8. Add, *ALU Output-Reg In* (R2 Out, ALU Input-Reg2 In, and PC Inc are deasserted)
9. ALU Output-Reg Out, R1 In (*ALU Output-Reg In* is deasserted)
10. PC Out, MAR In (R1 in is deasserted)



FSMs

- **Important: design STG first before designing FSM**
- Design FSMs based on the corresponding STG
- Synchronized by clock
- After pulling a control signal to active, remember to deactivate it later
- Make sure the order of activating/deactivating signals is correct
- Make sure the setup time and hold time are satisfied
- Functionality is more important than optimization

FSMs

- Instruction fetch
- Register-based ALU operations (ADD/SUB/AND/OR/NOT/XOR/XNOR)
- Immediate data-based ALU operations (ADDI/SUBI)
- Memory access (Load/Store)
- MOV
- MOVI

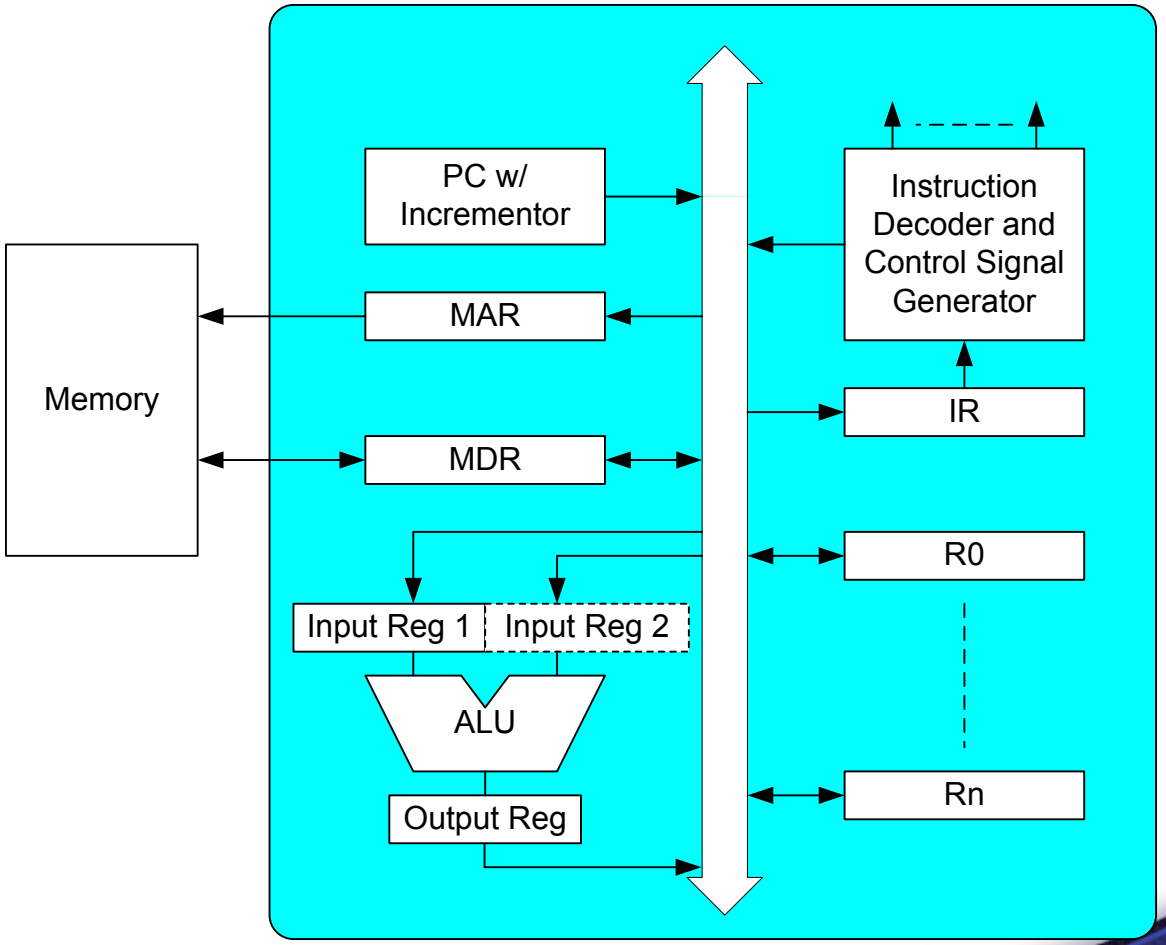
Final Project Step #4: FSM

- Determine the number of FSMs to be designed
- Design STG for each FSM
- Design all FSMs based on STGs
- Simulate in ModelSim to ensure the functionality using testbenches
- Synthesize the design in Design Vision
- Simulate the synthesized netlist
- Write a simple project report (code, testbench, conclusion)

- Due date #1: next Monday (Nov. 21)

Divide-and-Conquer #5

Putting Everything Together


PC w/
Incrementor
Instruction
Decoder and
Control Signal
Generator
Memory
MAR
IR
MDR
R0
Input Reg 1
Input Reg 2
ALU
Rn
Output Reg

Top-Level Design

- Connect everything together using structural model

- Example – XOR gate built from NOT, AND, and OR gates

```
module build_xor (a, b, c);
input a, b;
output c;
wire c, a_not, b_not;
not a_inv (a_not, a);
not b_inv (b_not, b);
and a1 (x, a_not, b);
and a2 (y, b_not, a);
or out (c, x, y);
endmodule
```

Bus

- Bidirectional – either declare as inout or separate input and output in each module
- Declare bus as wire at top-level
- Bus gets its value at top-level, outside of all FSMs, based on the enable signals for all tristate buffers

A Sample Top-Level Code

- module TheGreatProcessor(mar_out, mdr_out,mdr_in, En, RW, clk, reset, MFC);
- input clk, reset, MFC;
- input [15:0] mdr_in;
- output [15:0] mar_out, mdr_out;
- output RW, En;
- wire[15:0] bus;
-
- wire RW, En;
- wire[5:0] sour, dest;
- wire [3:0] command, fsm;
- wire load1, load2, load3, ALU_out, load_mar, en4, load_ir, inc_PC, PC_en, MFC, genload1, genload2, genload3, genload4, tr1, tr2, tr3, tr4, control, tricat;
- instructionFetch Fetching(load1, load2, load3, command, ALU_out, En, loadfrom_mem, loadto_mem, en4, load_mar, load_ir, inc_PC, PC_en, RW, genload1, genload2, genload3, genload4, tr1, tr2, tr3, tr4, reset, clk, MFC, fsm, sour, dest, control, tricat);
- top_level_ALU alu(bus, bus, bus, ALU_out, reset, clk, load1, load2, load3, command);
- top_mar mar(mar_out, bus, clk, reset, load_mar);
- top_MDR mdr(mdr_out, bus, mdr_in, bus, reset, loadto_mem, loadfrom_mem, clk, en4);
- top_pccounter count(bus, reset, inc_PC, PC_en);
- IRregDecode decode(fsm, sour, dest, bus, reset, clk, load_ir);
- N_Registers regis(bus, bus, bus, bus, reset, clk, tr1, tr2, tr3, tr4, genload1, genload2, genload3, genload4, bus);
- topcat cats(bus, reset, control, sour, dest, clk, tricat);
- endmodule

Another Sample of Top-Level Code

```
module uP(data_in , data_out , clk , reset , MFC , address , R_W , EN);
    input clk , reset , MFC;
    input[15:0] data_in;
    output R_W , EN;
    output[15:0] data_out , address;

    wire[15:0] bus, data_out;
    wire R_W, EN;

    wire inst_fetch_start , inst_fetch_done , mov_start , mov_done , movi_start , movi_done , ALU_start , ALU_done , ALUI_start , ALUI_done , LOAD_start , LOAD_done , STORE_start , STORE_done;
    wire reg1_in , reg1_out , reg2_in , reg2_out , reg3_in , reg3_out , reg4_in , reg4_out;
    wire MOV_RegB_In , MOVI_RegB_In , ALU_RegB_In , ALUI_RegB_In , LOAD_RegB_In , MOV_RegA_Out , ALU_RegA_Out , ALU_RegB_Out , ALUI_RegB_Out , LOAD_RegA_Out , STORE_RegA_Out , STORE_RegB_Out;
    wire MAR_IN , MAR_Out , INST_MAR_In , LOAD_MAR_In , STORE_MAR_In , INST_MAR_Out , LOAD_MAR_Out , STORE_MAR_Out;
    wire MDR_In_uP,MDR_Out_MEM, MDR_In_MEM, MDR_Out_uP, STORE_MDR_In_uP , STORE_MDR_Out_MEM, LOAD_MDR_In_MEM , LOAD_MDR_Out_uP, INST_MDR_In_MEM , INST_MDR_Out_uP;
    wire PC_OUT, PC_INC, IR_IN, ID_uP_OUT,MOV_ID_DATA_Out,ALU_ID_DATA_OUT;
    wire ALU_IN1,ALU_IN2,ALU_OUT,ALUI_ALU_In1 , ALUI_ALU_In2 , ALUI_ALU_OUT , ALU_ALU_In1 , ALU_ALU_In2 , ALU_ALU_OUT;
    wire inst_EN, load_EN, store_EN, inst_R_W, load_R_W, store_R_W;
    wire[2:0] ALU_op, FSM;
    wire[5:0] SRC , DST;
    wire[15:0] IR_out_data;

    //Components
    PC PC1(clk , PC_OUT , PC_INC , reset , bus);//DONE
    MAR MAR1(clk , MAR_IN , MAR_Out , reset , bus , address);//DONE
    MDR MDR1(clk , MDR_In_uP , MDR_Out_uP , MDR_In_MEM , MDR_Out_MEM , reset , bus , bus , data_in , data_out);//DONE
    IR IR1(clk , IR_IN , reset , bus , IR_out_data);//DONE
    ALU ALU1(clk , ALU_IN1 , ALU_IN2 , ALU_OUT , bus , ALU_op , reset , bus);//DONE
    Register REG1(clk , reg1_in , reg1_out , reset , bus , bus);//DONE
    Register REG2(clk , reg2_in , reg2_out , reset , bus , bus);//DONE
    Register REG3(clk , reg3_in , reg3_out , reset , bus , bus);//DONE
    Register REG4(clk , reg4_in , reg4_out , reset , bus , bus);//DONE
    ID ID1(clk , ID_uP_OUT , IR_out_data , reset , FSM , SRC , DST , ALU_op , bus);//DONE

    //FSMs
    Inst_Fetch Inst_Fetch1(clk , MFC , reset , inst_fetch_start , PC_OUT , INST_MAR_In , INST_MAR_Out , inst_R_W , inst_EN , INST_MDR_In_MEM , INST_MDR_Out_uP , IR_IN, PC_INC , inst_fetch_done);
    MOV_FSM MOV1(clk , reset , mov_start , MOV_RegA_Out , MOV_RegB_In , mov_done);//DONE
    MOVI_FSM MOVI1(clk , reset , movi_start , MOV_ID_DATA_Out , MOVI_RegB_In , movi_done);//DONE
    ALU_FSM ALU_FSM1(clk , reset , ALU_start , ALU_RegA_Out , ALU_RegB_In , ALU_RegB_Out , ALU_ALU_In1 , ALU_ALU_In2 , ALU_ALU_OUT , ALU_done);//DONE
    ALUI_FSM ALUI_FSM1(clk , reset , ALUI_start , ALU_ID_DATA_OUT , ALUI_RegB_In , ALUI_RegB_Out , ALUI_ALU_In1 , ALUI_ALU_In2 , ALUI_ALU_OUT , ALUI_done);//DONE
    Load_FSM Load1(clk , MFC , reset , LOAD_start , LOAD_MAR_In , LOAD_MAR_Out , load_R_W , load_EN , LOAD_MDR_In_MEM , LOAD_MDR_Out_uP , LOAD_RegA_Out , LOAD_RegB_In , LOAD_done);//DONE
    STORE_FSM Store1(clk , MFC , reset , STORE_start , STORE_MAR_In , STORE_MAR_Out , store_R_W , store_EN , STORE_MDR_In_uP , STORE_MDR_Out_MEM , STORE_RegA_Out , STORE_RegB_Out , STORE_done);//DONE

    assign inst_fetch_start = (mov_done || movi_done || ALU_done || ALUI_done || LOAD_done || STORE_done)? 1:0;
```

Another Sample Top-Level Code (Cont')

```
assign reg1_in = ((MOV_RegB_In ||MOVI_RegB_In || ALU_RegB_In || ALUI_RegB_In || LOAD_RegB_In) && DST == 0)?1:0;
assign reg2_in = ((MOV_RegB_In ||MOVI_RegB_In || ALU_RegB_In || ALUI_RegB_In || LOAD_RegB_In) && DST == 1)?1:0;
assign reg3_in = ((MOV_RegB_In ||MOVI_RegB_In || ALU_RegB_In || ALUI_RegB_In || LOAD_RegB_In) && DST == 2)?1:0;
assign reg4_in = ((MOV_RegB_In ||MOVI_RegB_In || ALU_RegB_In || ALUI_RegB_In || LOAD_RegB_In) && DST == 3)?1:0;
assign reg4_out = (((ALU_RegB_Out || ALUI_RegB_Out || STORE_RegB_Out) && DST == 3) || ((MOV_RegA_Out || ALU_RegA_Out || LOAD_RegA_Out || STORE_RegA_Out) && SRC ==3 ))?1:0;
assign reg3_out = (((ALU_RegB_Out || ALUI_RegB_Out || STORE_RegB_Out) && DST == 2) || ((MOV_RegA_Out || ALU_RegA_Out || LOAD_RegA_Out || STORE_RegA_Out) && SRC ==2 ))?1:0;
assign reg2_out = (((ALU_RegB_Out || ALUI_RegB_Out || STORE_RegB_Out) && DST == 1) || ((MOV_RegA_Out || ALU_RegA_Out || LOAD_RegA_Out || STORE_RegA_Out) && SRC ==1 ))?1:0;
assign reg1_out = (((ALU_RegB_Out || ALUI_RegB_Out || STORE_RegB_Out) && DST == 0) || ((MOV_RegA_Out || ALU_RegA_Out || LOAD_RegA_Out || STORE_RegA_Out) && SRC ==0 ))?1:0;

assign MAR_IN = (INST_MAR_In || LOAD_MAR_In || STORE_MAR_In)? 1:0;
assign MAR_Out = (INST_MAR_Out || LOAD_MAR_Out || STORE_MAR_Out)? 1:0;
assign MDR_In_uP = (STORE_MDR_In_uP)? 1:0;
assign MDR_In_MEM = (LOAD_MDR_In_MEM || INST_MDR_In_MEM)? 1:0;
assign MDR_Out_uP = (LOAD_MDR_Out_uP || INST_MDR_Out_uP)? 1:0;
assign MDR_Out_MEM = (STORE_MDR_Out_MEM)? 1:0;

assign ALU_IN1 = (ALUI_ALU_In1 || ALU_ALU_In1)? 1:0;
assign ALU_IN2 = (ALUI_ALU_In2 || ALU_ALU_In2)? 1:0;
assign ALU_OUT = (ALUI_ALU_OUT || ALU_ALU_OUT)? 1:0;

assign ID_uP_OUT = (MOV_ID_DATA_Out || ALU_ID_DATA_OUT)? 1:0;

assign mov_start = (FSM == 2 && inst_fetch_done)? 1:0;
assign movi_start = (FSM == 3 && inst_fetch_done)? 1:0;
assign ALU_start = (FSM == 0 && inst_fetch_done)? 1:0;
assign ALUI_start = (FSM == 1 && inst_fetch_done)? 1:0;
assign LOAD_start = (FSM == 4 && inst_fetch_done)? 1:0;
assign STORE_start = (FSM == 5 && inst_fetch_done)? 1:0;

assign EN = (inst_EN || load_EN || store_EN)? 1:0;
assign R_W = (inst_R_W && inst_EN) || (load_R_W && load_EN) || (store_R_W && store_EN);

endmodule
```

Memory

- Memory block is a FSM located in either your testbench or a separate module
- Inputs: R/$\overline{W}$, EN, Addr, Data In (optional – clock)
- Outputs: Data Out, MFC (memory function complete)
- Two types of operations: read and write
- MFC signal needs to be generated for both operation types
- The delay between the EN signal becomes active and the raise of MFC signal is flexible
- Don’t forget to drop MFC signal
- The contents of memory can be a behavioral code with pre-loaded machine codes corresponding to addresses

A Sample Memory Module

```
module mainmemory(Dataout, MFC, EN, addr, datain, RW);
  input EN, RW;
  input[15:0] addr, datain;
  output [15:0] Dataout;
  output MFC;

  reg [15:0] Dataout, memorycell;
  reg MFC;


  always@(posedge EN)
  begin
    if(RW==1) begin
```

```
case(addr)
  16'b0000000000000000: Dataout = 16'b1011001001000001;
  16'b0000000000000001: Dataout = 16'b0110000001111110;
  16'b0000000000000010: Dataout = 16'b0001000001000011;
  16'b0000000000000011: Dataout = 16'b0101000001000111;
  16'b0000000000000100: Dataout = 16'b1001000001000011;
  16'b0000000000000101: Dataout = 16'b0110000001111111;
  16'b0000000000000110: Dataout = 16'b1101000011000001;
  16'b0000000000000111: Dataout = 16'b1100000001000010;
  default: Dataout = memorycell;
    endcase
      end
    else memorycell = datain;
      #5 MFC = 1;
    end

    always@(negedge EN)
      MFC = 0;

    endmodule
```

A Sample Memory-in-Testbench Code

```
//uP TestBench
module uP_Test;
reg clk, reset, MFC;
reg[15:0] data_in;
wire R_W, EN;
wire[15:0] data_out,address;
reg[15:0] mem[65535:0];  //65535 = FFFF(hex)
integer x;
uP test(data_in ,  data_out ,  clk ,  reset ,  MFC ,  address ,  R_W ,
EN);
initial
begin

  for(x =0; x<65536; x=x+1)begin
  mem[x] = 16'b0;
  end

  mem[0] = 16'b0001000000111110;  //MOVI  R0, #3E
  mem[1] = 16'b1101000000000001;  //MOV  R0, R1
  mem[2] = 16'b1001000111000001;  //ADDI  R1, #7
  mem[3] = 16'b1010000100000000;  //SUBI  R0, #4
  mem[4] = 16'b0111000000000001;  //XOR  R0, R1
  mem[5] = 16'b0100000000000000;  //INV  R0
  mem[6] = 16'b1100000001000000;  //STORE  R1, (R0)
  mem[7] = 16'b1011000000000010;  //LOAD  (R0), R2

 clk = 0;
  reset = 1;
  MFC = 0;
  data_in = 16'bz;
  @(negedge clk) reset = 0;
end

  always begin
  #5000 clk = ~clk;
  end

  always @(address)
  begin
    if(EN)
    begin
      if(R_W)
      begin
        data_in = mem[address];
        #5000 MFC = 1;
        #10000 MFC = 0;
      end
      else
      begin
        mem[address] = data_out;
        #5000 MFC = 1;
        #10000 MFC = 0;
      end
    end
    else
    data_in = 16'bz;
  end
endmodule
```

Sample Test Program

- MOVI R2, #3D
- SUBI R2, #9
- MOV R3, R2
- MOVI R1, #15
- XOR R1, R2
- INV R1
- STORE R3, (R1)
- LOAD (R1), R0

How to Debug

- Simulate each individual block to make sure all of them work as planned
- **After system-level simulation, monitor all inputs and outputs of every block**
- Analyze these signals to see what the problem is, e.g.,
  - Reset doesn't work as expected
  - multiple FSMs are active at the same time
  - control signal sequence is not correct
  - data/control timing is wrong
  - PC doesn't increment
  - something is holding the bus
  - a register is always latching new data
  - next instruction fetch never happens
  - memory doesn't give the correct instruction back
- Fix the problem

Final Project Report

- Project objective
- Design methodology of each block
- Instruction set with machine code assignments
- Other assumptions made
- Synthesis process
- Simulation method – translate the program into machine code
- Result (simulation waveform screenshot) – need transactions on bus
- Analysis and conclusion – explain the waveforms
- Source code (pre- and post-synthesis file, testbench)

- Due date: December 8 at midnight (before the 9th), email your report to jdi@uark.edu in either Word or PDF format