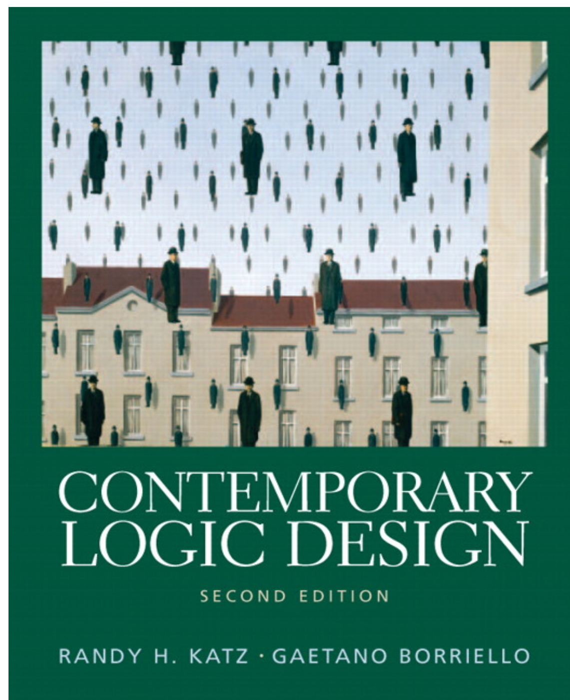
CSE 370 Spring 2006
Introduction to Digital Design

Lecture 26: Computer Organization



Last Lecture

- Computer Organization

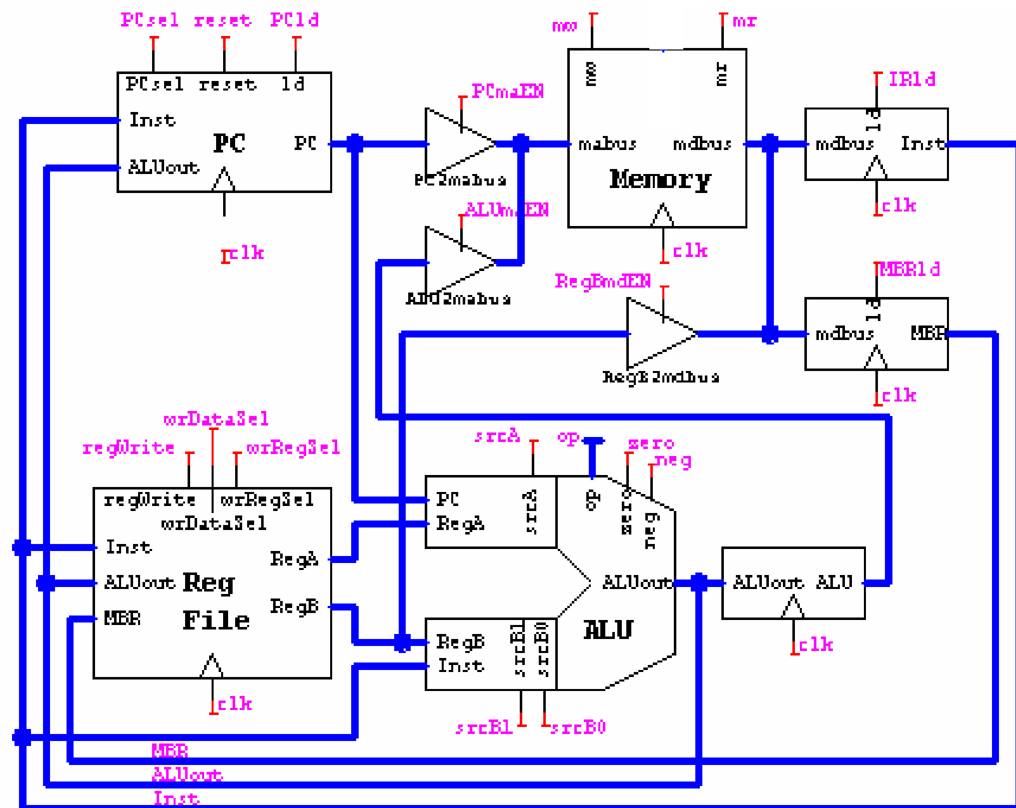
Today

- More Computer Organization

Administrivia

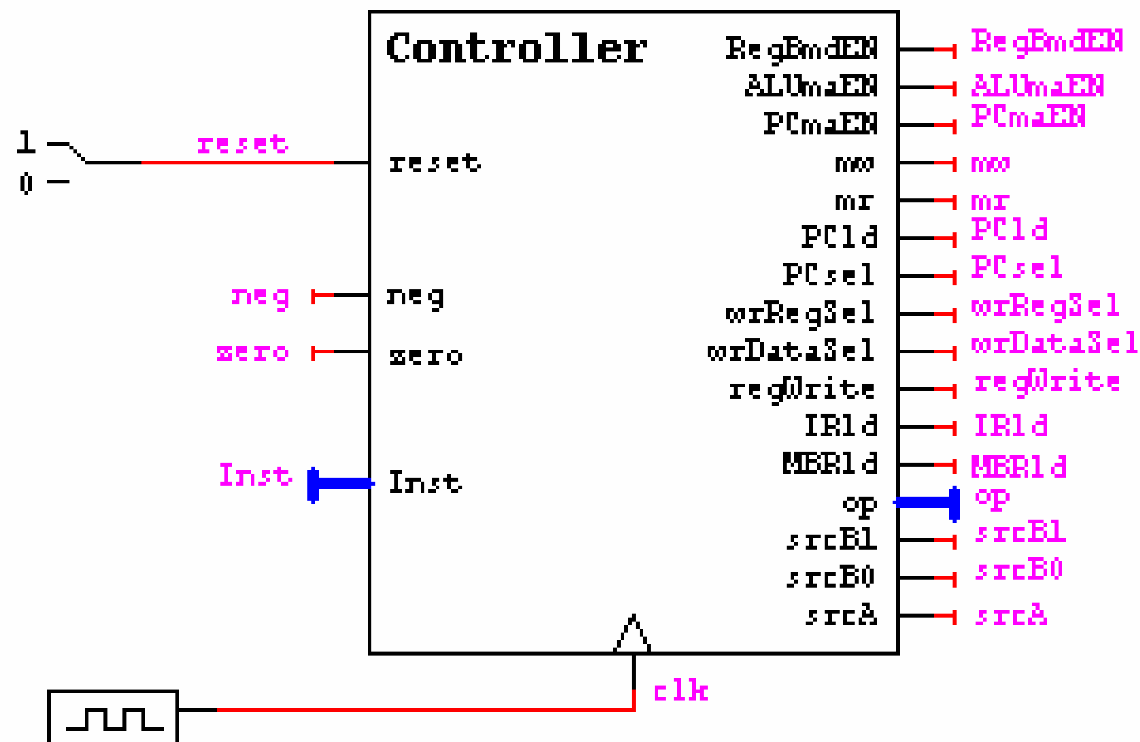
A simplified processor data-path and memory

- Princeton architecture
- Register file
- Instruction register
- PC incremented through ALU
- Modeled after MIPS rt000 (used in 378 textbook by Patterson & Hennessy)
  - really a 32-bit machine
  - we'll do a 16-bit version



Processor control

- Synchronous Mealy or Moore machine
- Multiple cycles per instruction



Processor instructions

- Three principal types (16 bits in each instruction)

type	op	rs	rt	rd	funct
R(egister)	3	3	3	3	4
I(mmediate)	3	3	3	7	
J(ump)	3	13			

- Some of the instructions

R	add	0	rs	rt	rd	0	rd = rs + rt
	sub	0	rs	rt	rd	1	rd = rs - rt
	and	0	rs	rt	rd	2	rd = rs & rt
	or	0	rs	rt	rd	3	rd = rs | rt
	slt	0	rs	rt	rd	4	rd = (rs < rt)
I	lw	1	rs	rt	offset		rt = mem[rs + offset]
	sw	2	rs	rt	offset		mem[rs + offset] = rt
	beq	3	rs	rt	offset		pc = pc + offset, if (rs == rt)
	addi	4	rs	rt	offset		rt = rs + offset
J	j	5	target address				pc = target address
	halt	7	-				stop execution until reset

Tracing an instruction's execution

- Instruction: r3 = r1 + r2

R	0	rs=r1	rt=r2	rd=r3	funct=0

- 1. instruction fetch
  - move instruction address from PC to memory address bus
  - assert memory read
  - move data from memory data bus into IR
  - configure ALU to add 1 to PC
  - configure PC to store new value from ALUout
- 2. instruction decode
  - op-code bits of IR are input to control FSM
  - rest of IR bits encode the operand addresses (rs and rt)
    - these go to register file

Tracing an instruction's execution (cont'd)

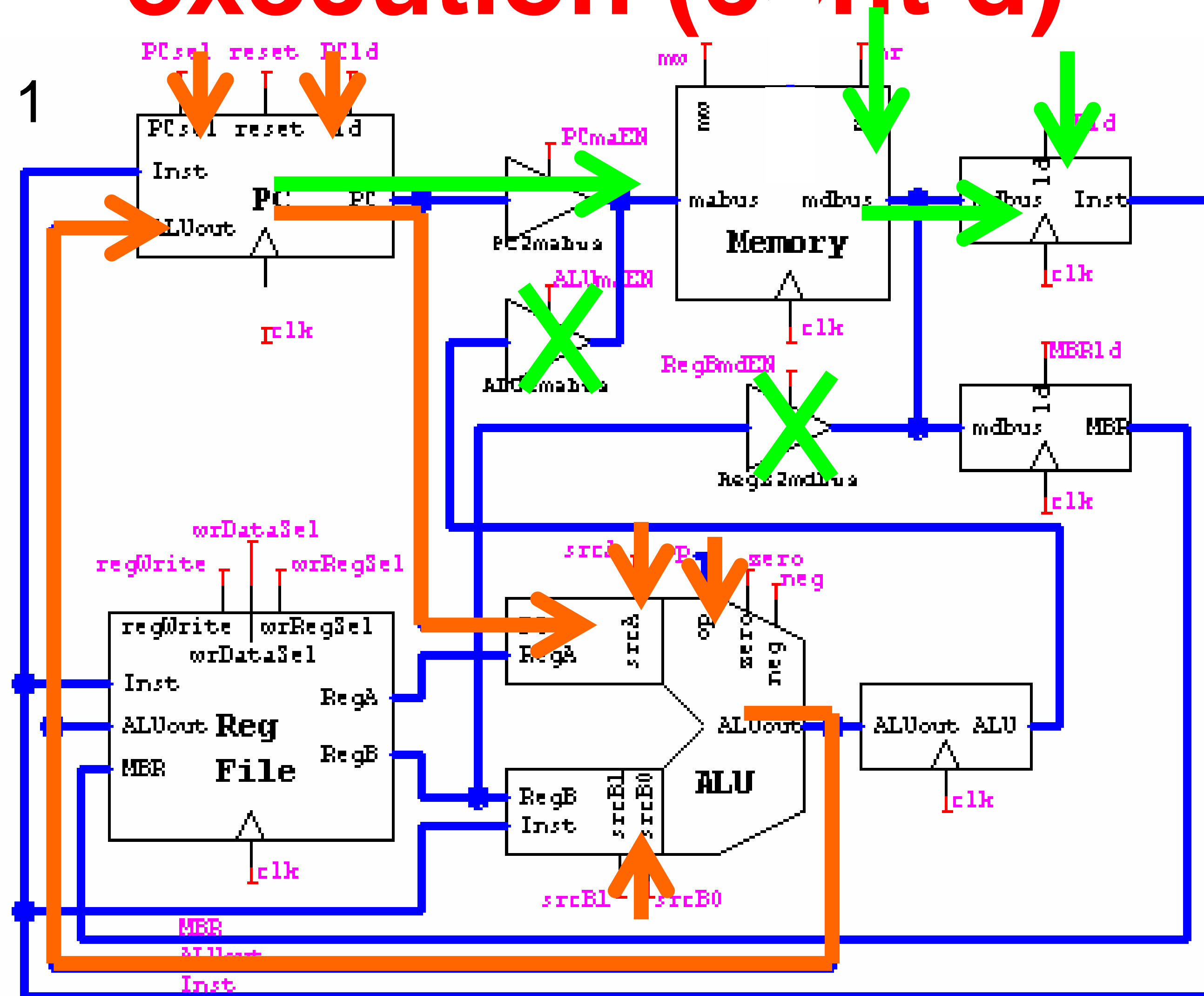
- Instruction: r3 = r1 + r2

R	0	rs=r1	rt=r2	rd=r3	funct=0

- 3. instruction execute
  - set up ALU inputs
  - configure ALU to perform ADD operation
  - configure register file to store ALU result (rd)

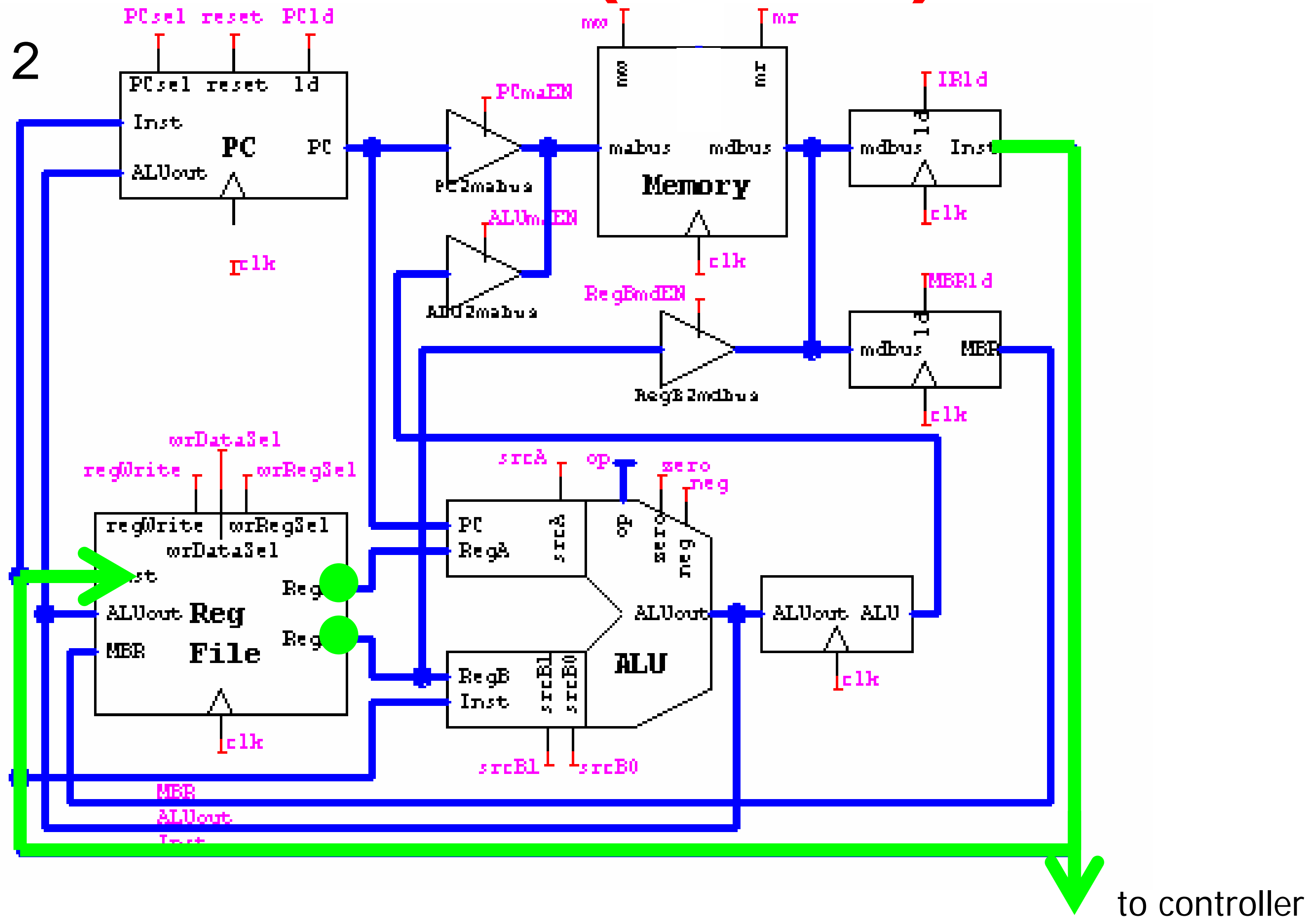
Tracing an instruction's execution (cont'd)

- Step 1



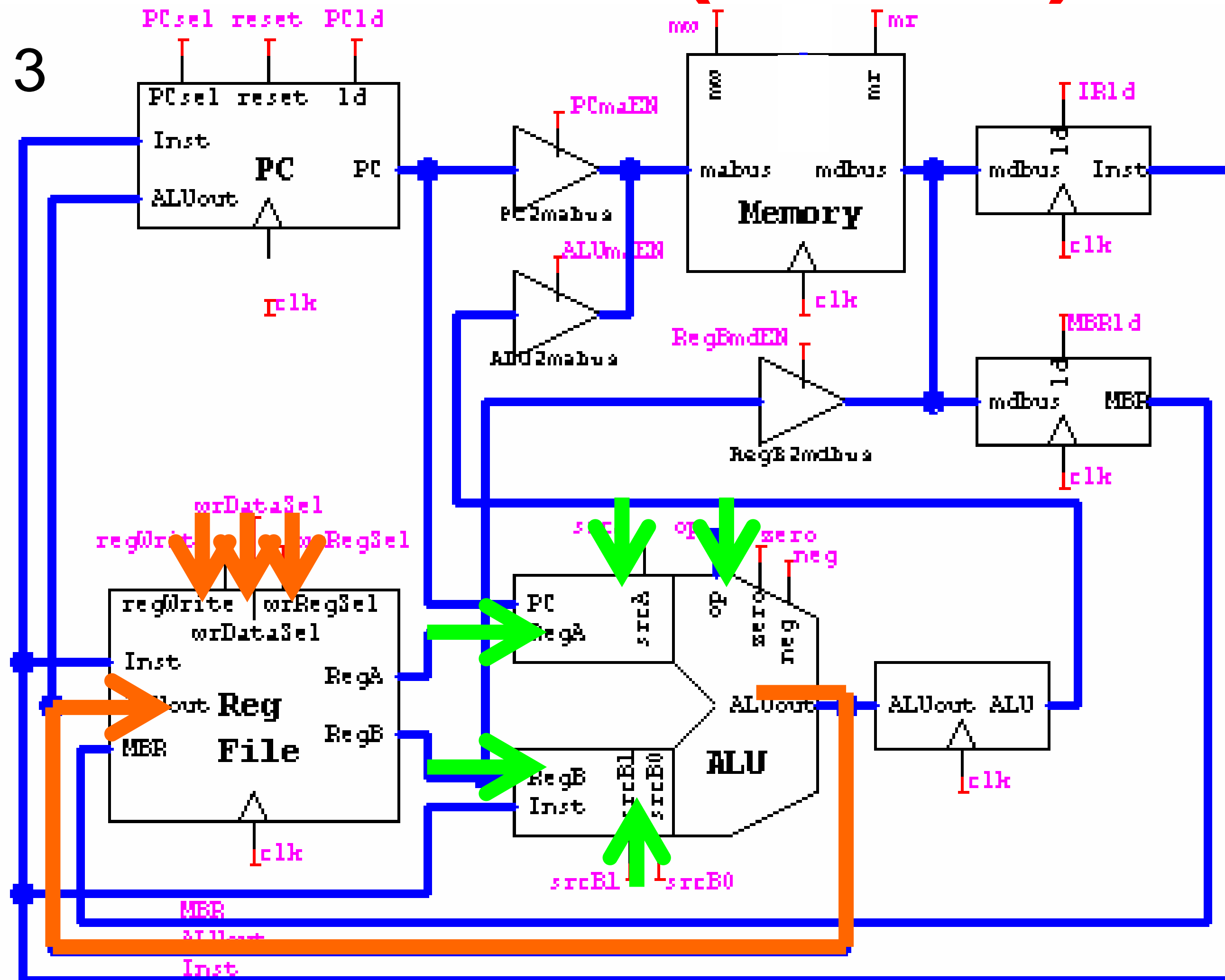
Tracing an instruction's execution (cont'd)

- Step 2



Tracing an instruction's execution (cont'd)

- Step 3



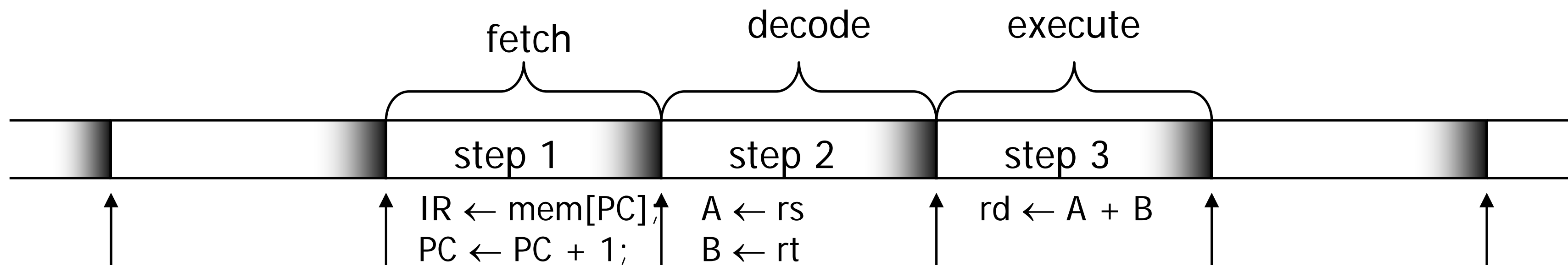
Register-transfer-level description

- Control
  - transfer data between registers by asserting appropriate control signals
- Register transfer notation - work from register to register
  - instruction fetch:
    mabus ← PC; – move PC to memory address bus (PCmaEN, ALUmaEN)
    memory read; – assert memory read signal (mr, RegBmdEN)
    IR ← memory; – load IR from memory data bus (IRld)
    op ← add – send PC into A input, 1 into B input, add (srcA, srcB0, scrB1, op)
    PC ← ALUout – load result of incrementing in ALU into PC (PCld, PCsel)
  - instruction decode:
    IR to controller
    values of A and B read from register file (rs, rt)
  - instruction execution:
    op ← add – send regA into A input, regB into B input, add (srcA, srcB0, scrB1, op)
    rd ← ALUout – store result of add into destination register (regWrite, wrDataSel, wrRegSel)

Register-transfer-level description (cont'd)

- How many states are needed to accomplish these transfers?
  - data dependencies (where do values that are needed come from?)
  - resource conflicts (ALU, busses, etc.)
- In our case, it takes three cycles
  - one for each step
  - all operation within a cycle occur between rising edges of the clock
- How do we set all of the control signals to be output by the state machine?
  - depends on the type of machine (Mealy, Moore, synchronous Mealy)

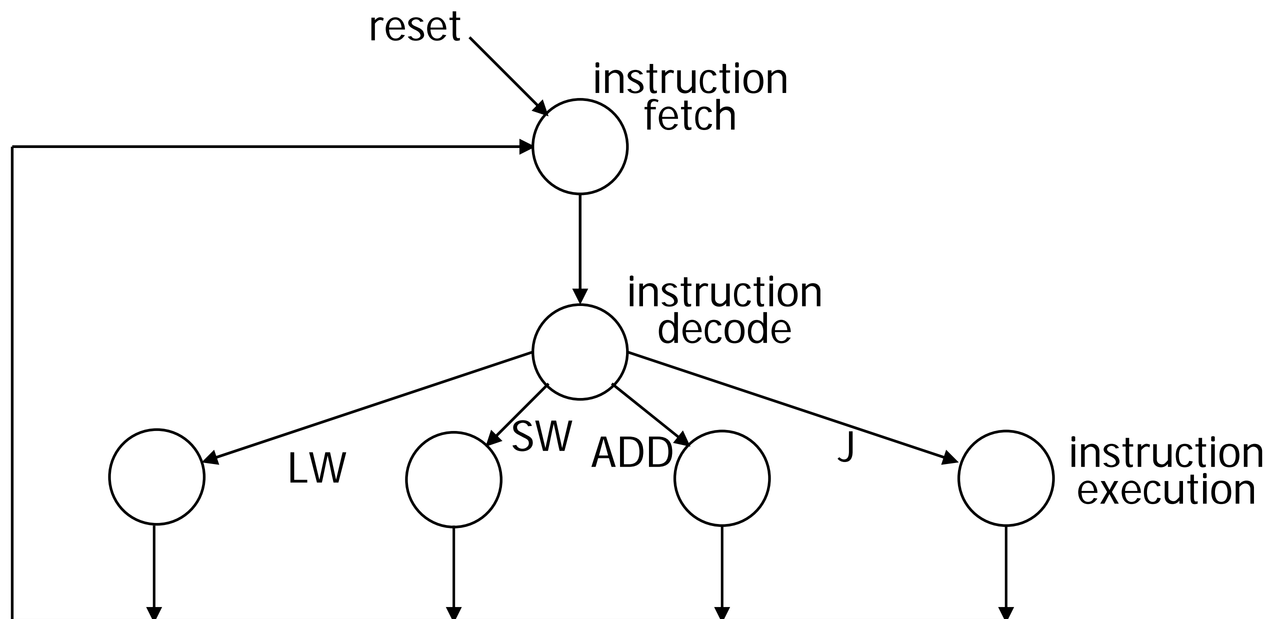
Review of FSM timing



to configure the data-path to do this here, when do we set the control signals?

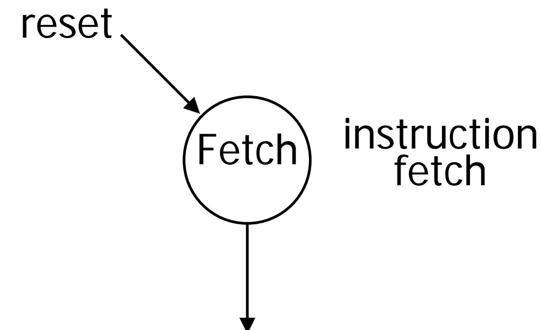
FSM controller for CPU (skeletal Moore FSM)

- First pass at deriving the state diagram (Moore machine)
  - these will be further refined into sub-states



FSM controller for CPU (reset and inst. fetch)

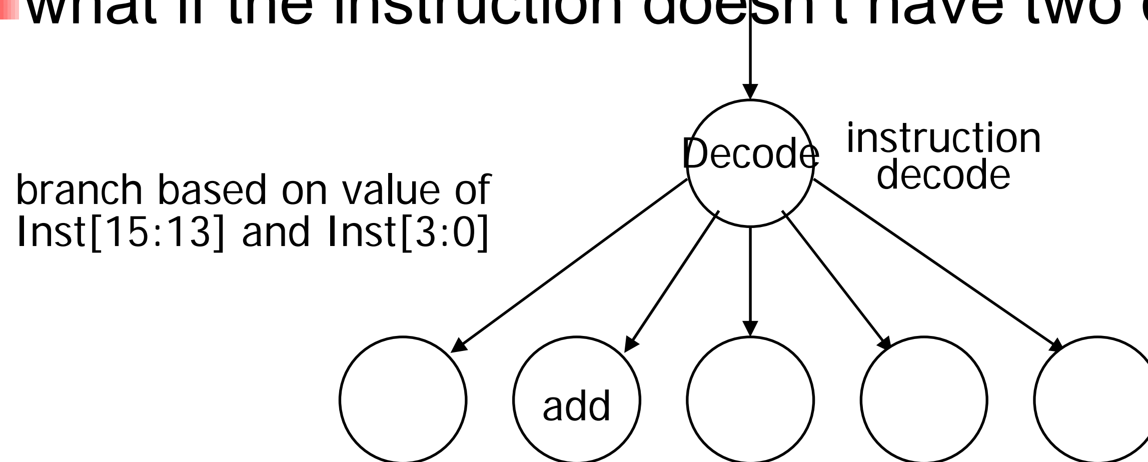
- Assume Moore machine
  - outputs associated with states rather than arcs
- Reset state and instruction fetch sequence
- On reset (go to Fetch state)
  - start fetching instructions
  - PC will set itself to zero



$$\begin{aligned}&\text{mabus} \leftarrow \text{PC;}\\&\text{memory read;}\\&\text{IR} \leftarrow \text{memory data bus;}\\&\text{PC} \leftarrow \text{PC} + 1;\end{aligned}$$

FSM controller for CPU (decode)

- Operation decode state
  - next state branch based on operation code in instruction
  - read two operands out of register file
    - what if the instruction doesn’t have two operands?

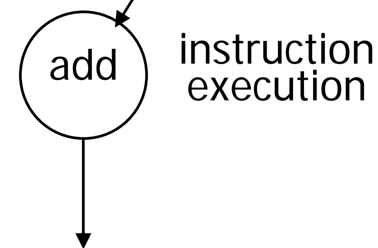


FSM controller for CPU (instruction execution)

- For add instruction
  - configure ALU and store result in register

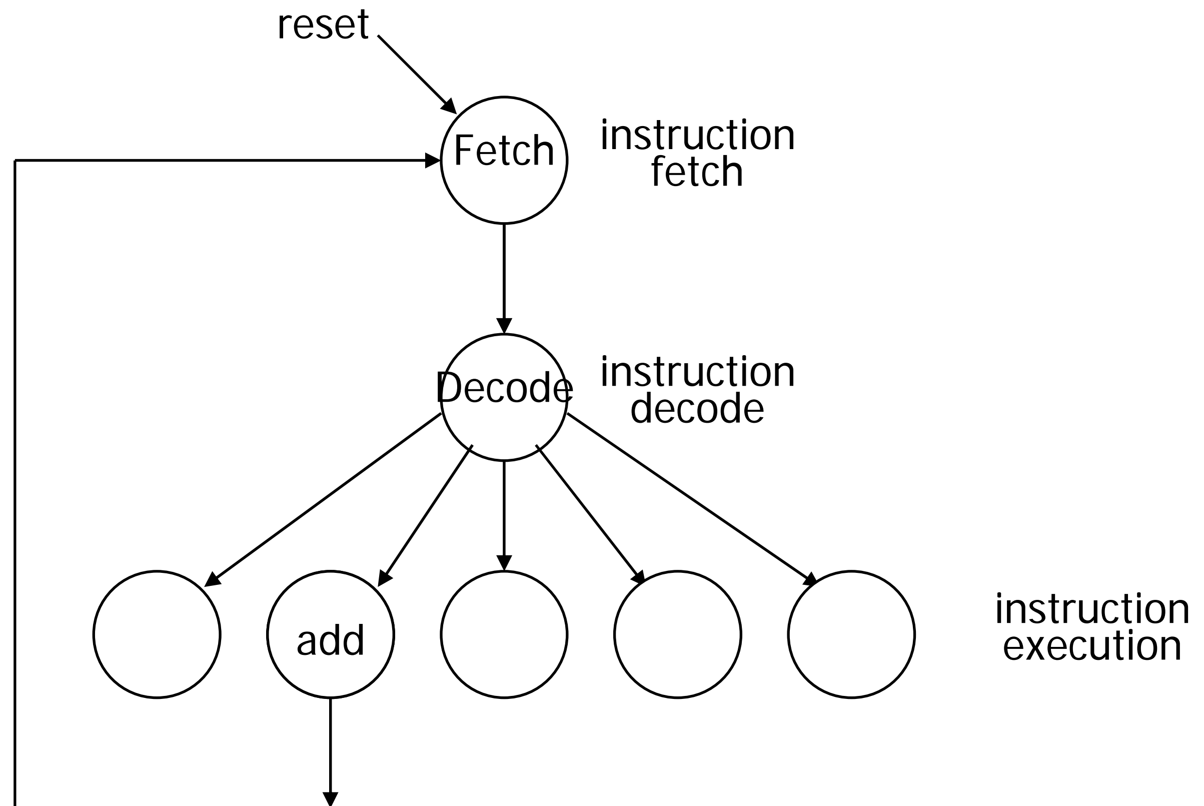
    $rd \leftarrow A + B$

  - other instructions may require multiple cycles



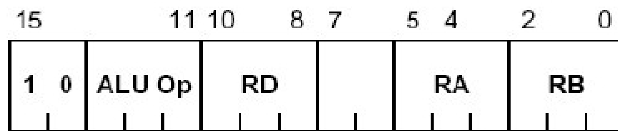
FSM controller for CPU (add instruction)

- Putting it all together and closing the loop
  - the famous instruction fetch decode execute cycle



FSM controller for CPU

- Now we need to repeat this for all the instructions of our processor
  - fetch and decode states stay the same
  - different execution states for each instruction
    - some may require multiple states if available register transfer paths require sequencing of steps

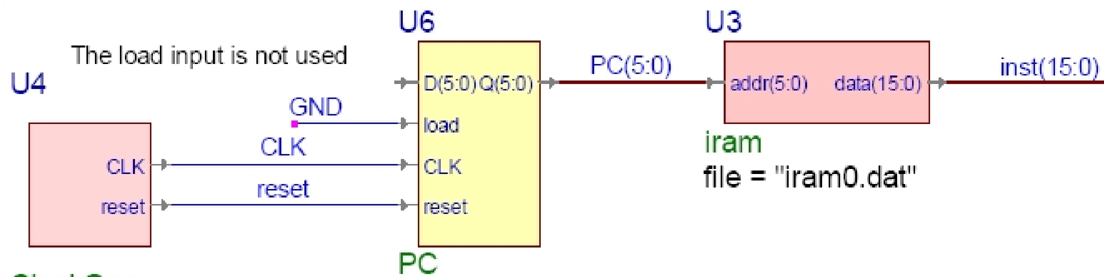


RD = RA op RB

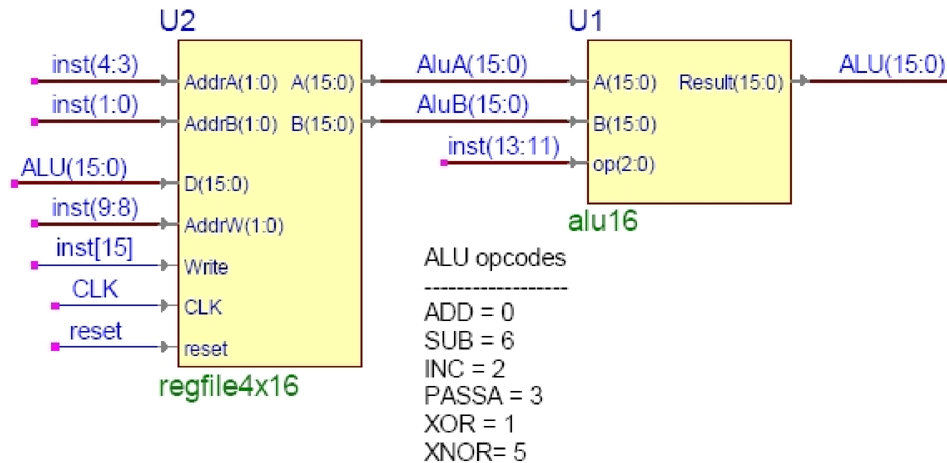
ALU instruction

This design needs to be run with a clock period of 20ns.
To single-step, set the simulation step to 20ns.

GND

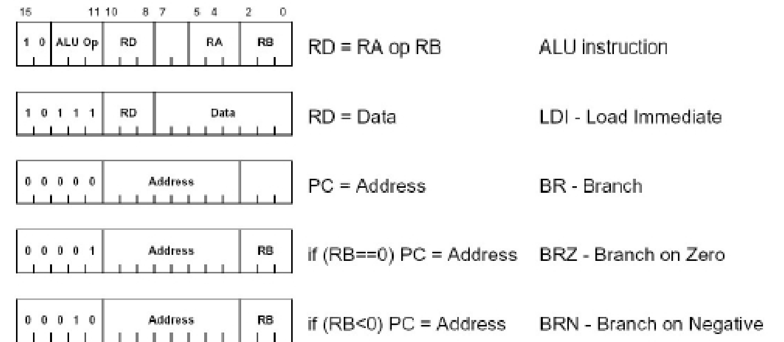


ClockGen
period = 10
number = 1000000
number =
period = 20



Instruction Fields

```
wire [4:0] opcode = inst[15:11];
assign ALUopcode = inst[13:11];
assign RA = inst[4:3];   // only 4 registers
assign RB = inst[1:0];   // => 2-bit register address
assign RD = inst[9:8];
assign data = inst[7:0];
```



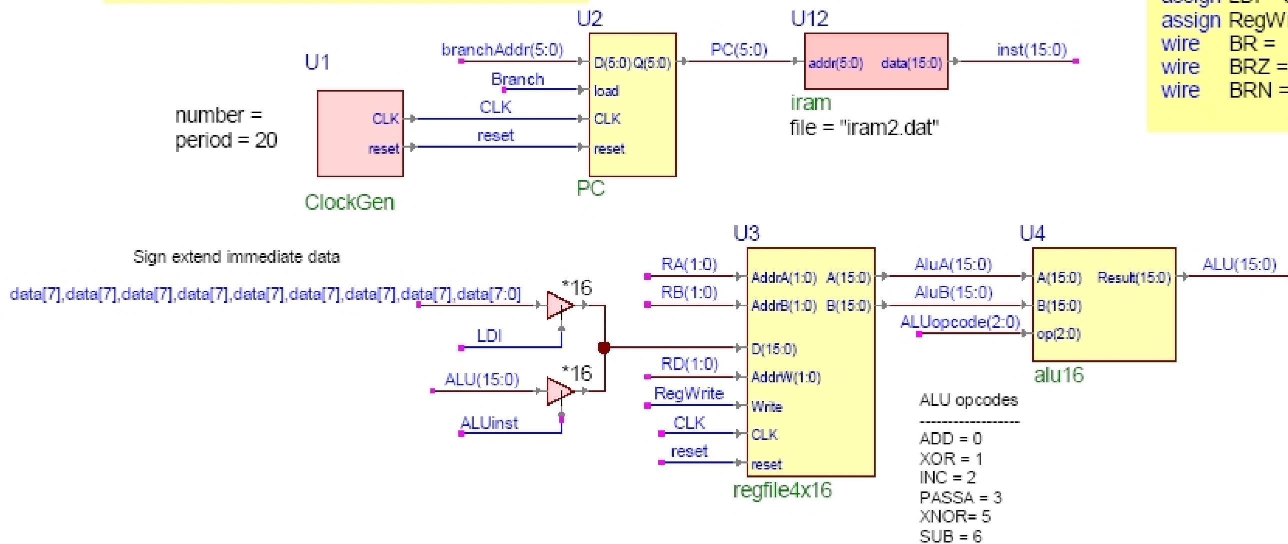
This design needs to be run with a clock period of 20ns.
To single-step, set the simulation step to 20ns.

Branch Logic

```
assign Branch = BR ||
                (BRZ && (AluB==0)) ||
                (BRN && (AluB[15]==1));
```

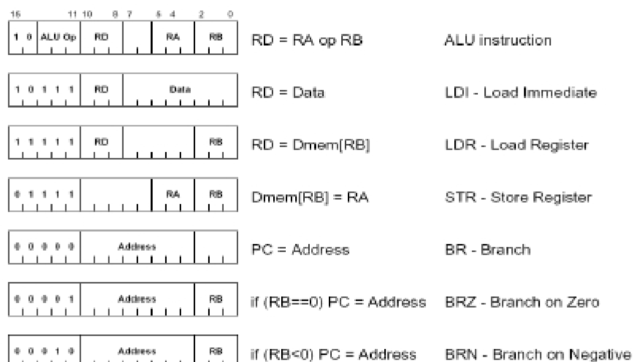
Instruction Decode

```
assign ALUinst = ALUopcode != 7;   // ALU performs first six op co ...
assign LDI = opcode == 'h17;      // Load immediate op code
assign RegWrite = opcode[4];      // Write to register
wire    BR =   opcode == 0;
wire    BRZ = opcode == 1;
wire    BRN = opcode == 2;
```



Instruction Fields

```
wire [4:0] opcode = inst[15:11];
assign ALUopcode = inst[13:11];
assign RA = inst[4:3];   // only 4 registers
assign RB = inst[1:0];   // => 2-bit register address
assign RD = inst[9:8];
assign data = inst[7:0];
```



This design needs to be run with a clock period of 20ns.
To single-step, set the simulation step to 20ns.

Branch Logic

```
assign Branch = BR ||
                (BRZ && (AluB==0)) ||
                (BRN && (AluB[15]==1));
```

Instruction Decode

```
assign ALUinst = ALUopcode != 7;   // ALU performs first six op co ...
assign LDI = opcode == 'h17;       // Load immediate op code
assign RegWrite = opcode[4];       // Write to register
wire   BR =  opcode == 0;
wire   BRZ = opcode == 1;
wire   BRN = opcode == 2;
assign LDR = opcode == 'h1f;
assign STR = opcode == 'hf;
```