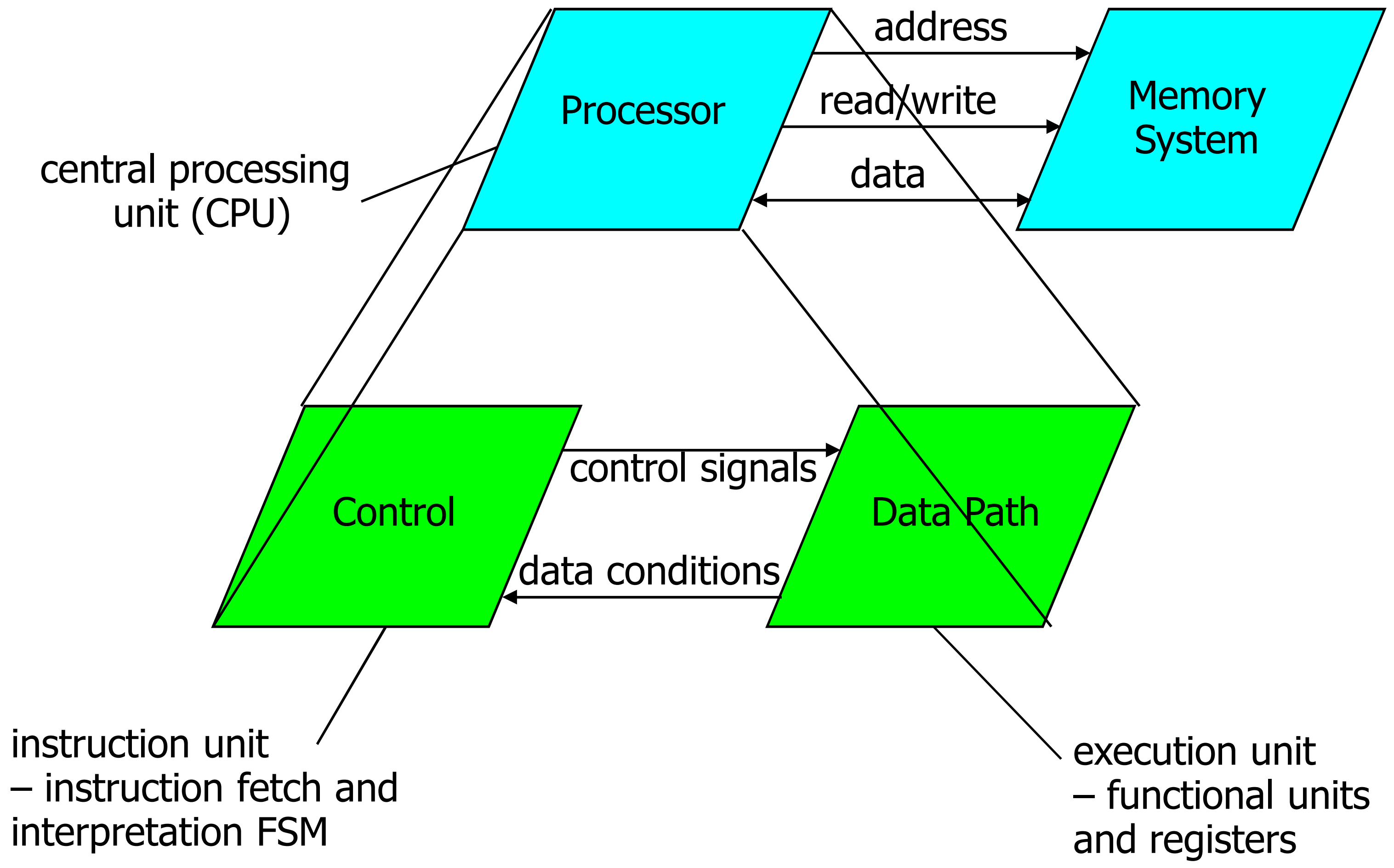
Computer organization

- Computer design – an application of digital logic design procedures
- Computer = processing unit + memory system
- Processing unit = control + datapath
- Control = finite state machine
  - inputs = machine instruction, datapath conditions
  - outputs = register transfer control signals, ALU operation codes
  - instruction interpretation = instruction fetch, decode, execute
- Datapath = functional units + registers
  - functional units = ALU, multipliers, dividers, etc.
  - registers = program counter, shifters, storage registers

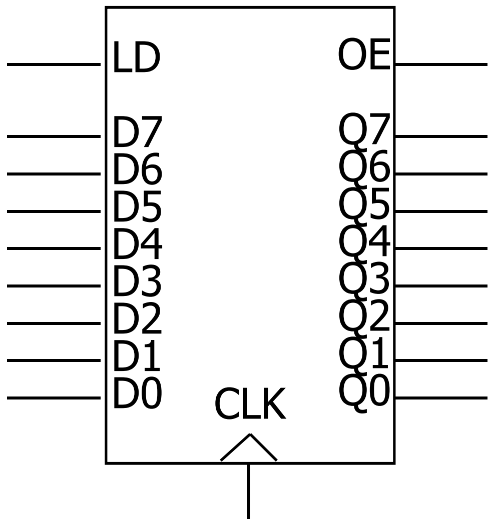
Structure of a computer

- Block diagram view


Processor
address
read/write
data
Memory System
central processing unit (CPU)
Control
control signals
data conditions
Data Path
instruction unit – instruction fetch and interpretation FSM
execution unit – functional units and registers

Registers

- Selectively loaded – EN or LD input
- Output enable – OE input
- Multiple registers – group 4 or 8 in parallel

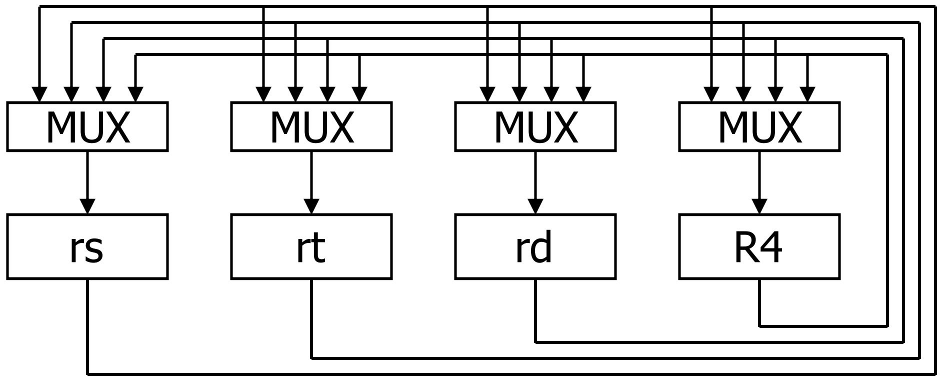


OE asserted causes FF state to be connected to output pins; otherwise they are left unconnected (high impedance)

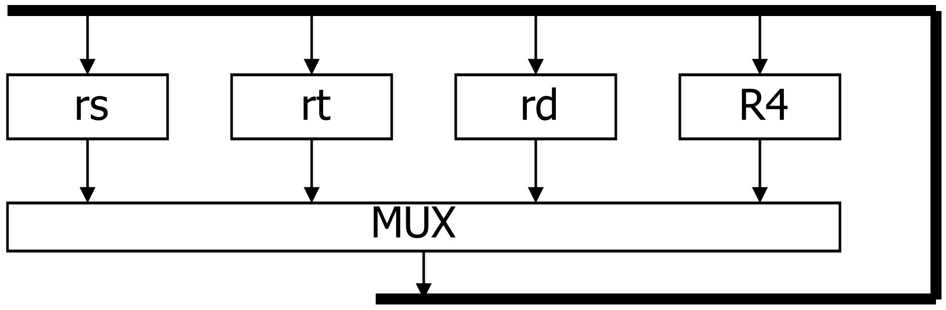
LD asserted during a lo-to-hi clock transition loads new data into FFs

Register transfer

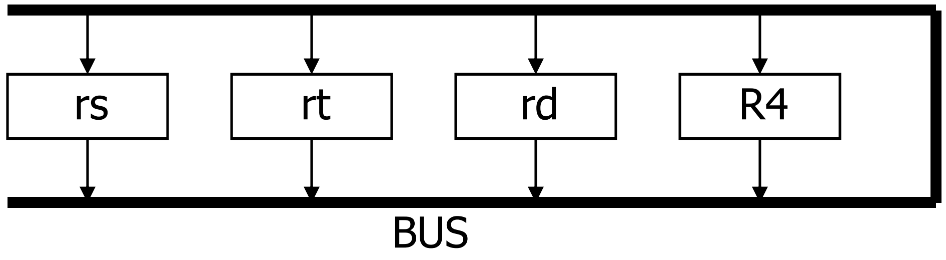
- Point-to-point connection
  - dedicated wires
  - muxes on inputs of each register



- Common input from multiplexer
  - load enables for each register
  - control signals for multiplexer

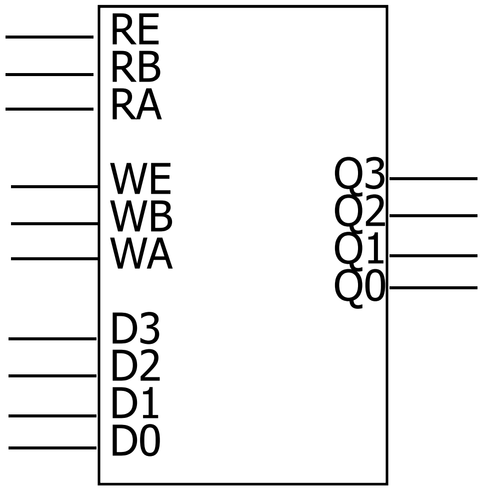


- Common bus with output enables
  - output enables and load enables for each register



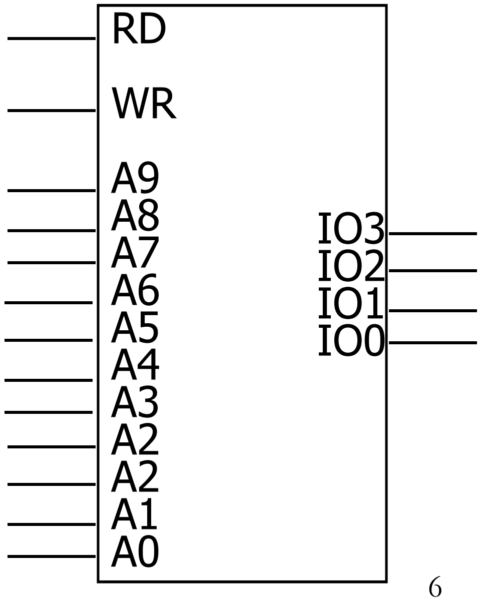
Register files

- Collections of registers in one package
  - two-dimensional array of FFs
  - address used as index to a particular word
  - can have separate read and write addresses so can do both at same time
- 4 by 4 register file
  - 16 D-FFs
  - organized as four words of four bits each
  - write-enable (load)
  - read-enable (output enable)



Memories

- Larger collections of storage elements
  - implemented not as FFs but as much more efficient latches
  - high-density memories use 1 to 5 switches (transitors) per memory bit
- Static RAM – 1024 words each 4 bits wide
  - once written, memory holds forever (not true for denser dynamic RAM)
  - address lines to select word (10 lines for 1024 words)
  - read enable
    - same as output enable
    - often called chip select
    - permits connection of many chips into larger array
  - write enable (same as load enable)
  - bi-directional data lines
    - output when reading, input when writing



Instruction sequencing

- Example – an instruction to
  add the contents of two registers (Rx and Ry)
  and place result in a third register (Rz)
- Step 1: get the ADD instruction from memory into an instruction register
- Step 2: decode instruction
  - instruction in IR has the code of an ADD instruction
  - register indices used to generate output enables for registers Rx and Ry
  - register index used to generate load signal for register Rz
- Step 3: execute instruction
  - enable Rx and Ry output and direct to ALU
  - setup ALU to perform ADD operation
  - direct result to Rz so that it can be loaded into register

Instruction types

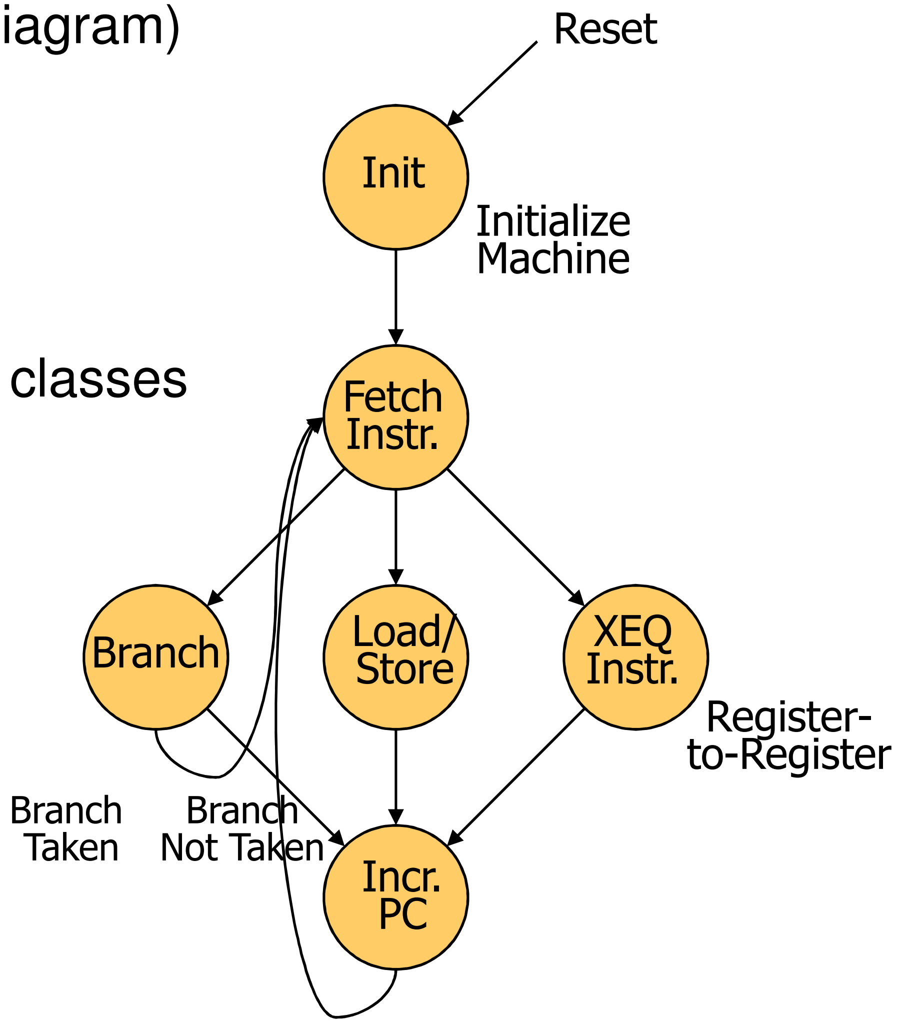
- Data manipulation
  - add, subtract
  - increment, decrement
  - multiply
  - shift, rotate
  - immediate operands
- Data staging
  - load/store data to/from memory
  - register-to-register move
- Control
  - conditional/unconditional branches in program flow
  - subroutine call and return

Elements of the control unit (aka instruction unit)

- Standard FSM elements
  - state register
  - next-state logic
  - output logic (datapath/control signalling)
  - Moore or synchronous Mealy machine to avoid loops unbroken by FF
- Plus additional "control" registers
  - instruction register (IR)
  - program counter (PC)
- Inputs/outputs
  - outputs control elements of data path
  - inputs from data path used to alter flow of program (test if zero)

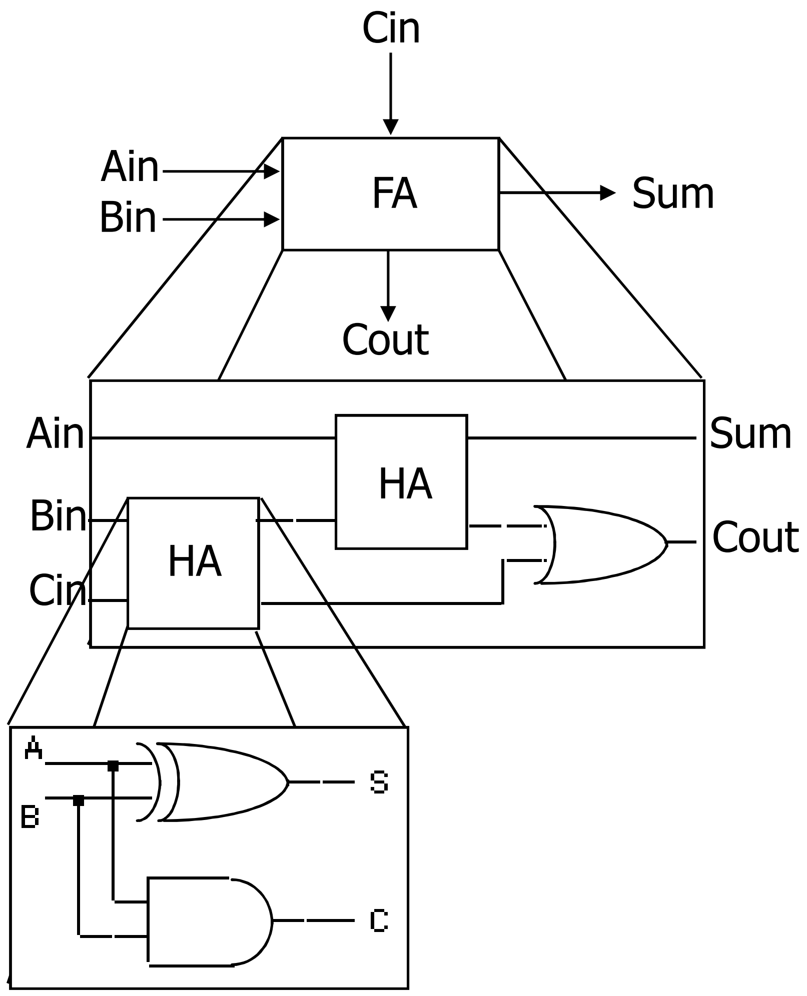
Instruction execution

- Control state diagram (for each diagram)
  - reset
  - fetch instruction
  - decode
  - execute
- Instructions partitioned into three classes
  - branch
  - load/store
  - register-to-register
- Different sequence through diagram for each instruction type



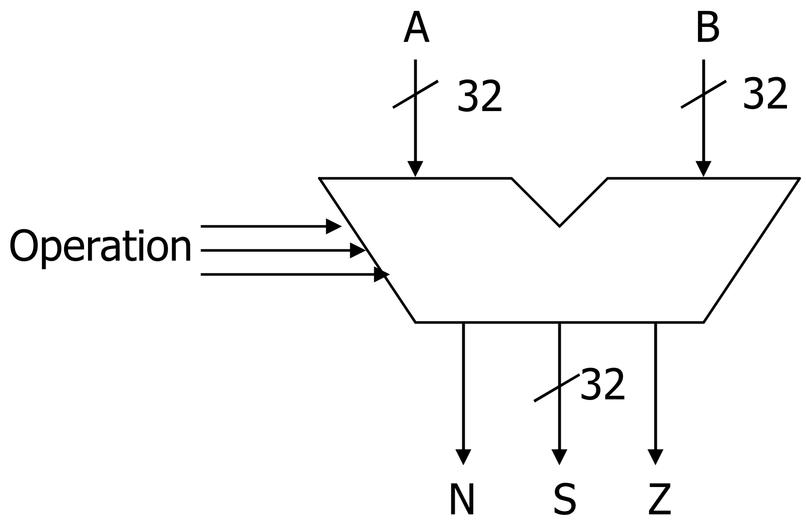
Data path (hierarchy)

- Arithmetic circuits constructed in hierarchical and modular fashion
  - each bit in datapath is functionally identical
  - 4-bit, 8-bit, 16-bit, 32-bit datapaths



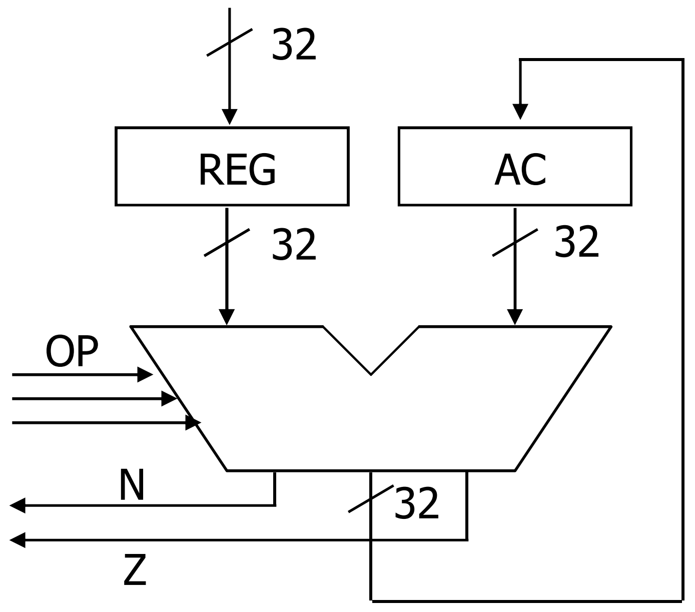
Data path (ALU)

- ALU block diagram
  - input: data and operation to perform
  - output: result of operation and status information



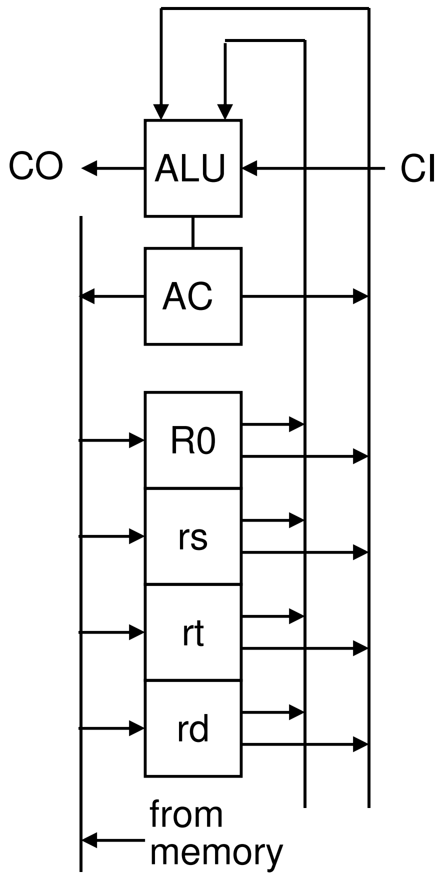
Data path (ALU + registers)

- Accumulator
  - special register
  - one of the inputs to ALU
  - output of ALU stored back in accumulator
- One-address instructions
  - operation and address of one operand
  - other operand and destination is accumulator register
  - AC <– AC op Mem[addr]
  - "single address instructions" (AC implicit operand)
- Multiple registers
  - part of instruction used to choose register operands

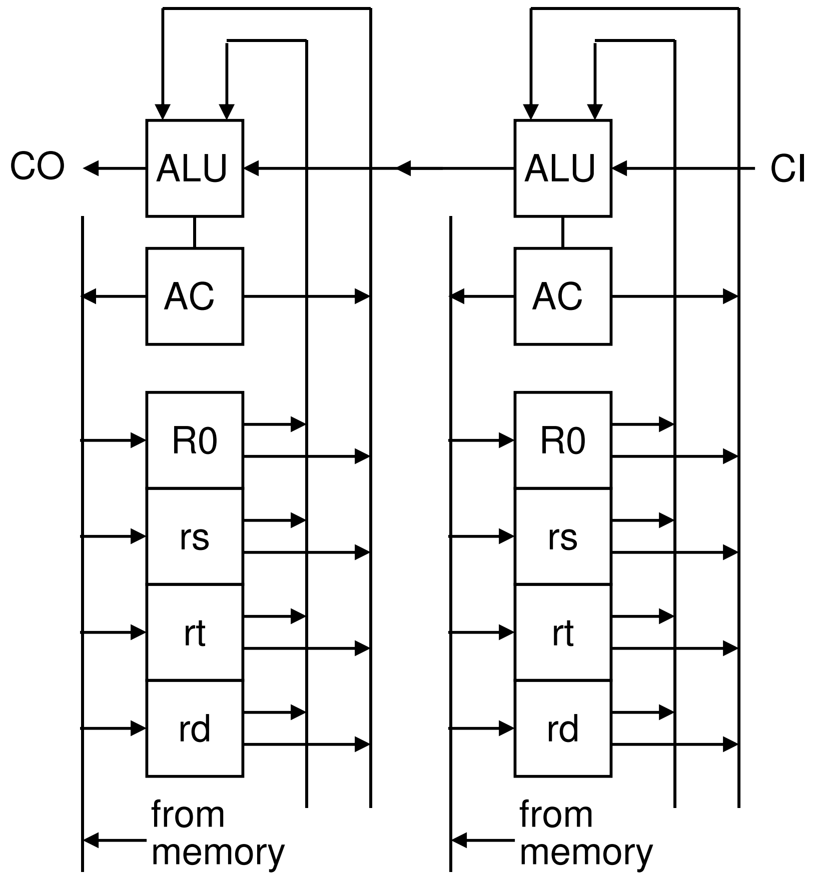


Data path (bit-slice)

- Bit-slice concept – replicate to build n-bit wide datapaths



1 bit wide



2 bits wide

Instruction path

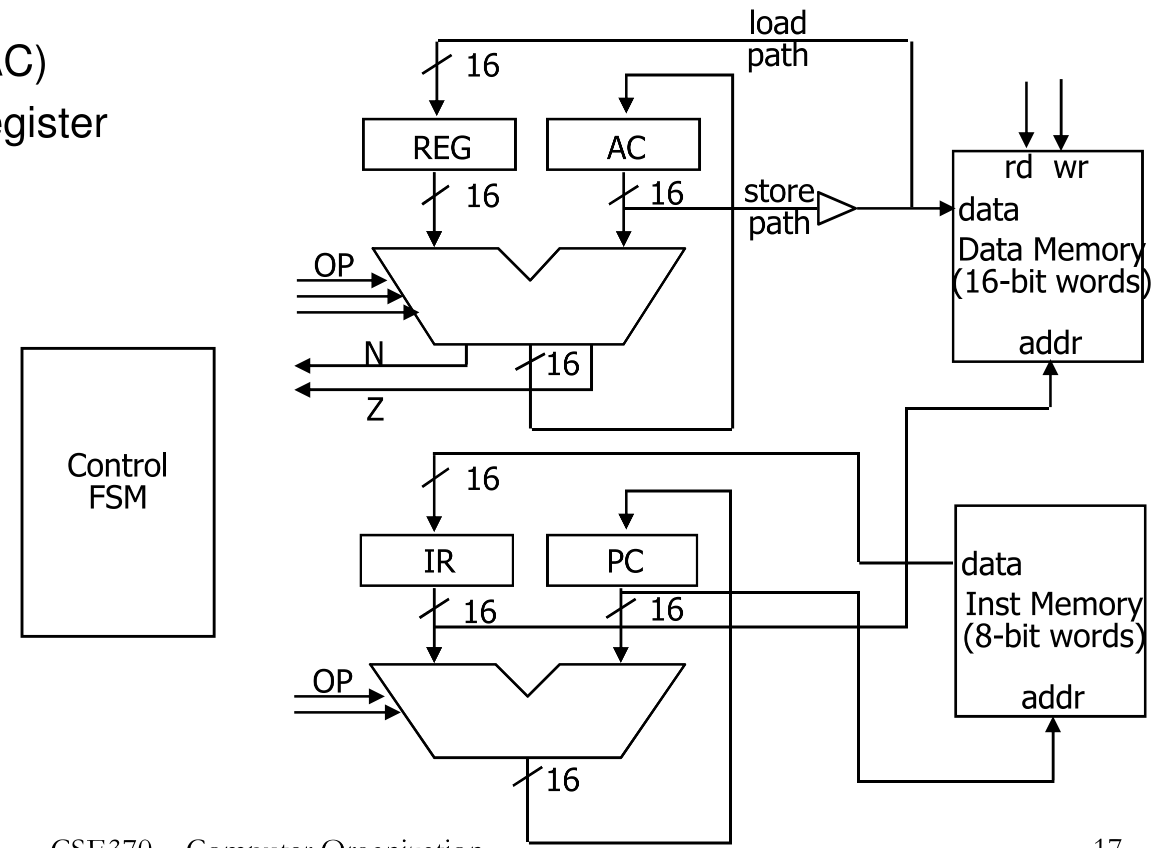
- Program counter (PC)
  - keeps track of program execution
  - address of next instruction to read from memory
  - may have auto-increment feature or use ALU
- Instruction register (IR)
  - current instruction
  - includes ALU operation and address of operand
  - also holds target of jump instruction
  - immediate operands
- Relationship to data path
  - PC may be incremented through ALU
  - contents of IR may also be required as input to ALU – immediate operands

Data path (memory interface)

- Memory
  - separate data and instruction memory (Harvard architecture)
    - two address busses, two data busses
  - single combined memory (Princeton architecture)
    - single address bus, single data bus
- Separate memory
  - ALU output goes to data memory input
  - register input from data memory output
  - data memory address from instruction register
  - instruction register from instruction memory output
  - instruction memory address from program counter
- Single memory
  - address from PC or IR
  - memory output to instruction and data registers
  - memory input from ALU output

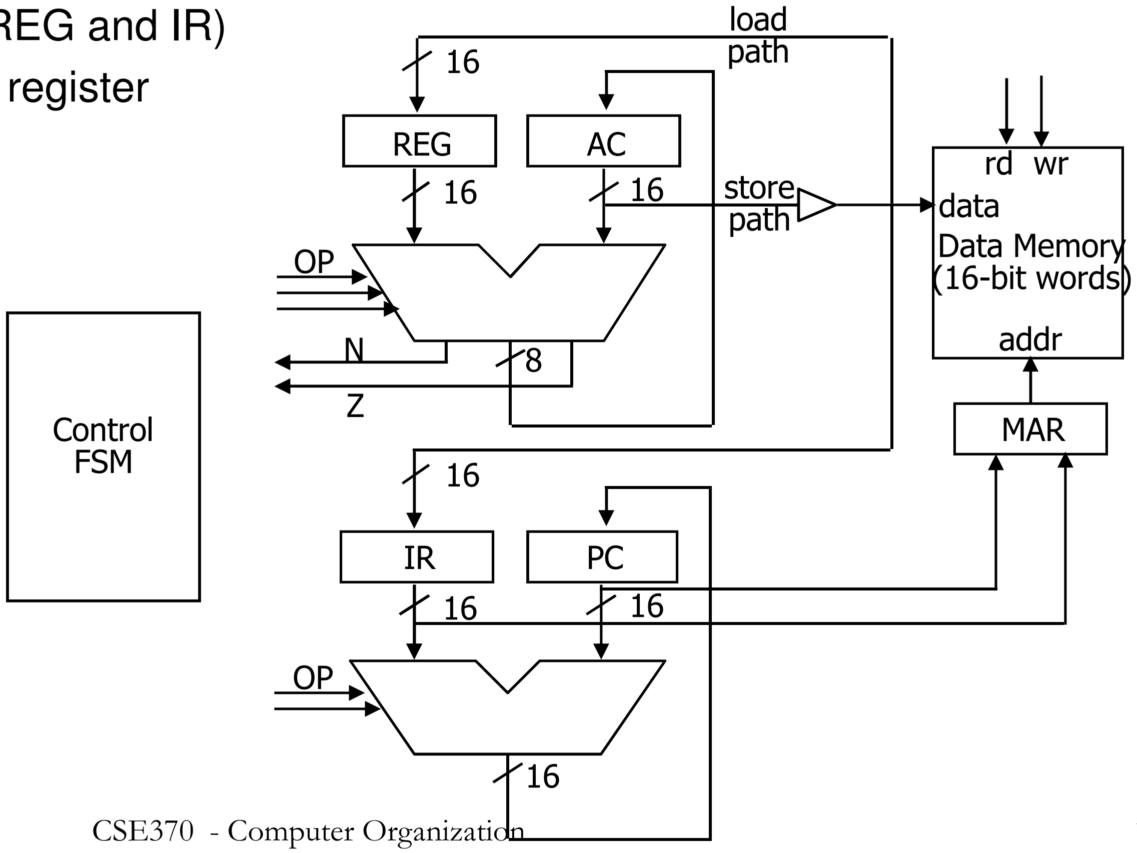
Block diagram of processor (Harvard)

- Register transfer view of Harvard architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - two MARs (PC and IR)
  - two MBRs (REG and AC)
  - load control for each register



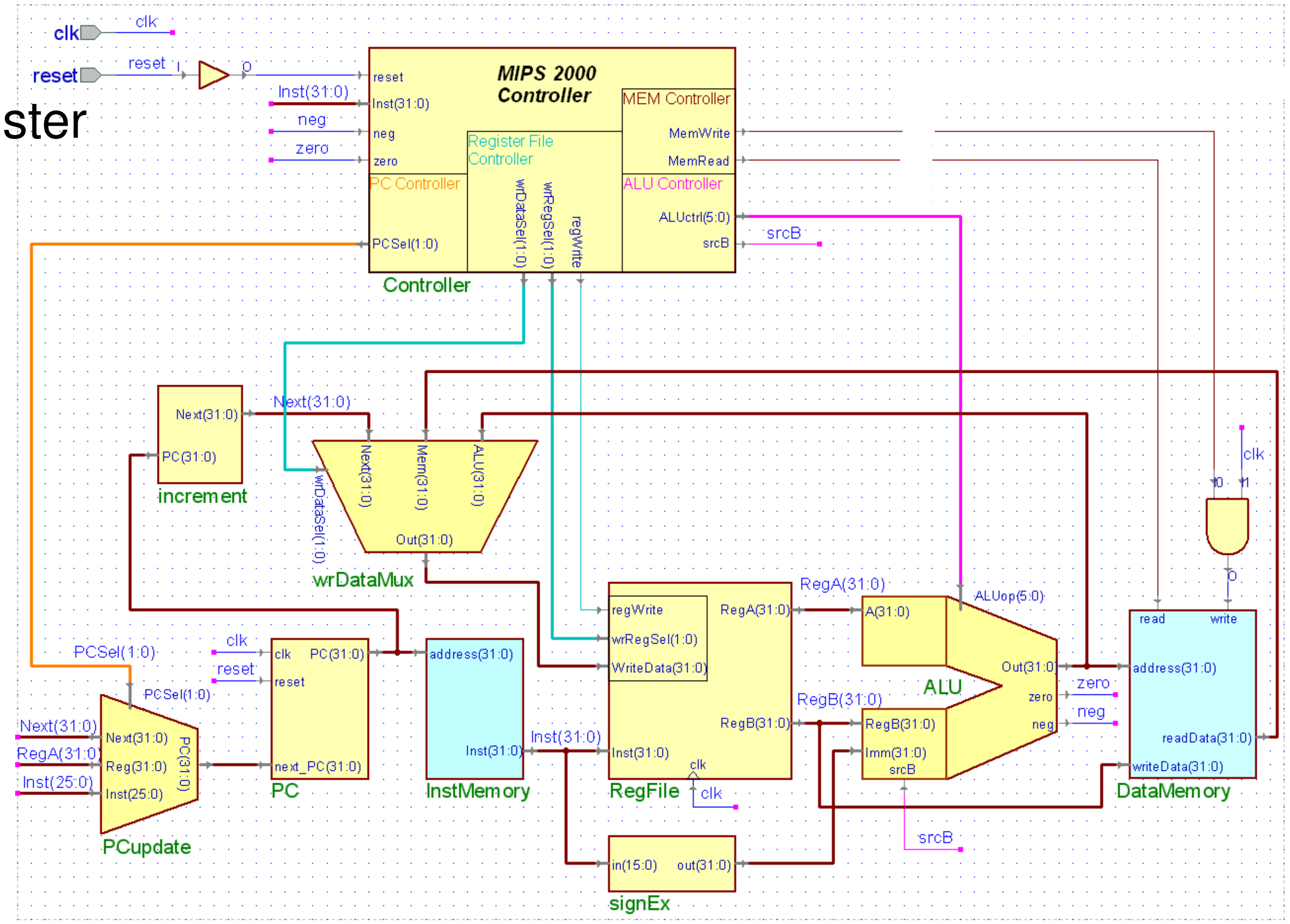
Block diagram of processor (Princeton)

- Register transfer view of Princeton architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - MAR may be a simple multiplexer rather than separate register
  - MBR is split in two (REG and IR)
  - load control for each register

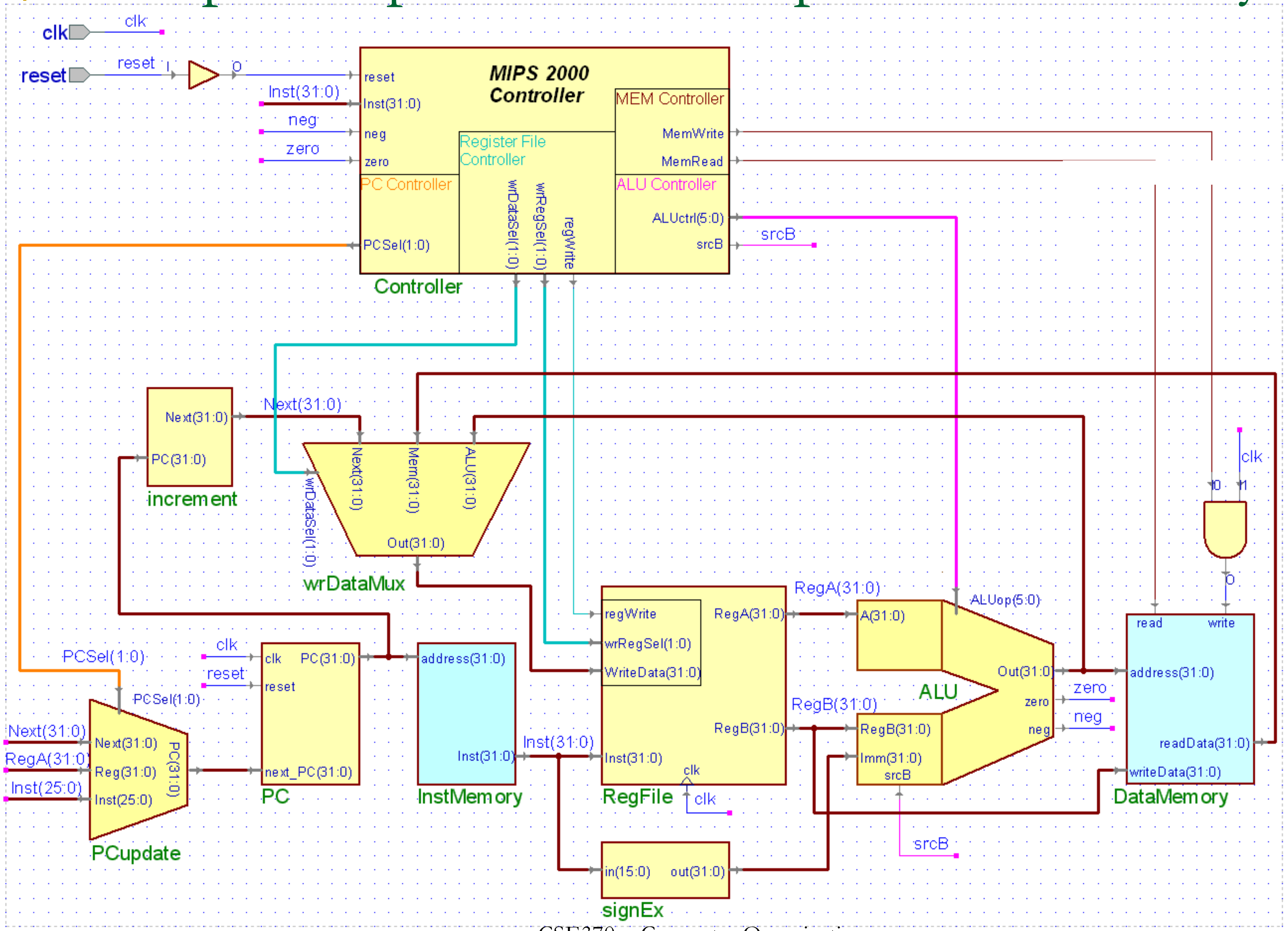


A simplified processor data-path and memory

- Harvard architecture
- Register file
- Instruction register
- Modeled after MIPS rt2000 (used in 378 textbook by Patterson & Hennessy)
- a 32-bit machine

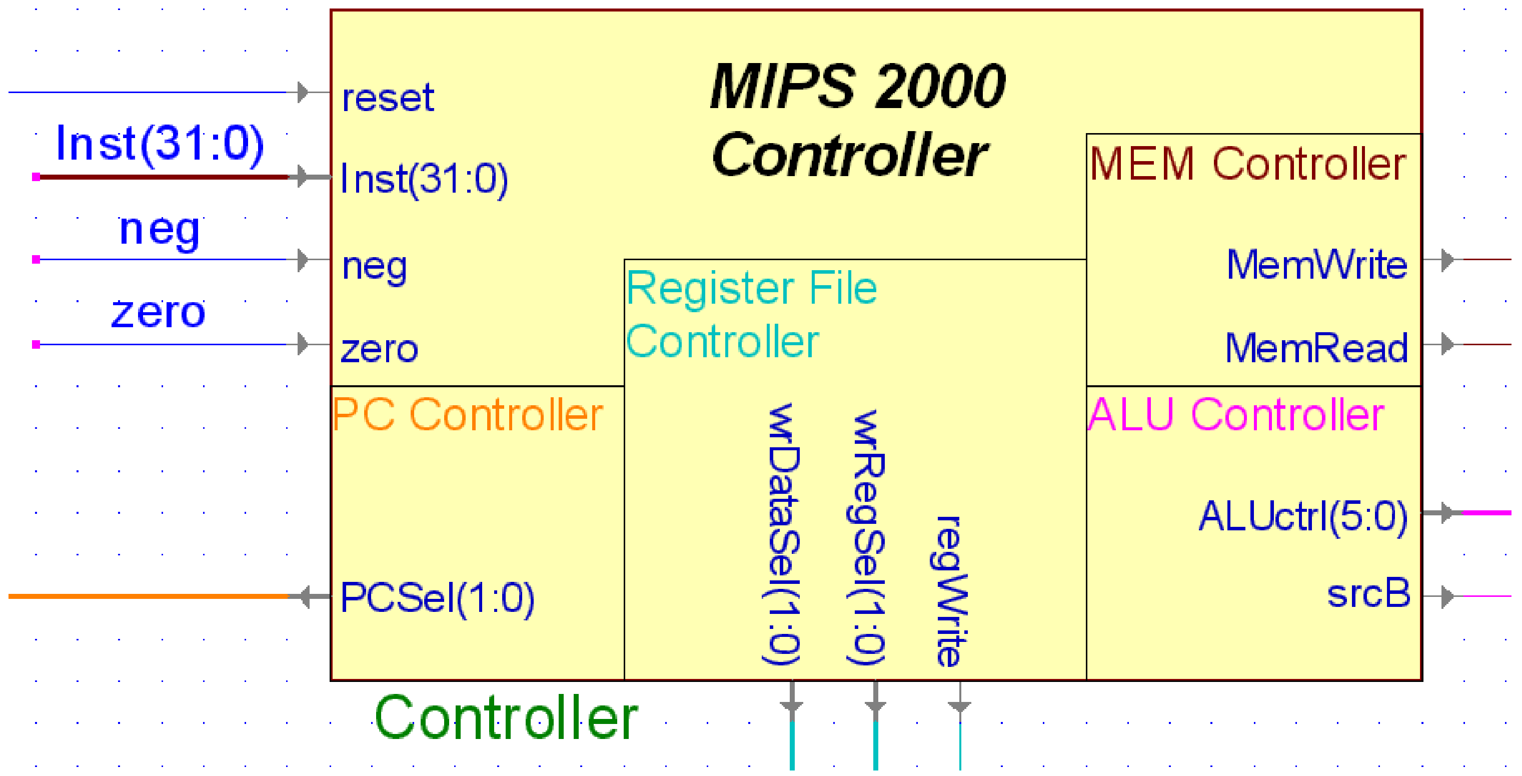


A simplified processor data-path and memory



Processor control

- Combinational Controller– decodes instruction and sets control lines
- Single cycle per instruction
- This is what you will finish in HW 8



Processor instructions

- Three principal types (32 bits in each instruction)

type	op	rs	rt	rd	shift	funct
R(egister)	6	5	5	5	5	6
I(mmediate)	6	5	5	16- immediate		
J(ump)	6	26- immediate				

- Some of the instructions

R	add	6'h00	rs	rt	rd	6'h20	rd = rs + rt
	sub	6'h00	rs	rt	rd	6'h22	rd = rs - rt
	and	6'h00	rs	rt	rd	6'h24	rd = rs & rt
	or	6'h00	rs	rt	rd	6'h25	rd = rs | rt
	slt	6'h00	rs	rt	rd	6'h2a	rd = (rs < rt)
I	lw	6'h23	rs	rt	offset		rt = mem[rs + offset]
	sw	6'h2b	rs	rt	offset		mem[rs + offset] = rt
	beq	6'h04	rs	rt	offset		pc = pc + offset, if (rs == rt)
	addi	6'h08	rs	rt	offset		rt = rs + offset
J	j	6'h02	target address				pc = target address
	halt	6'h3f	-				stop execution until reset

Tracing an instruction's execution

- Instruction: r3 = r1 + r2

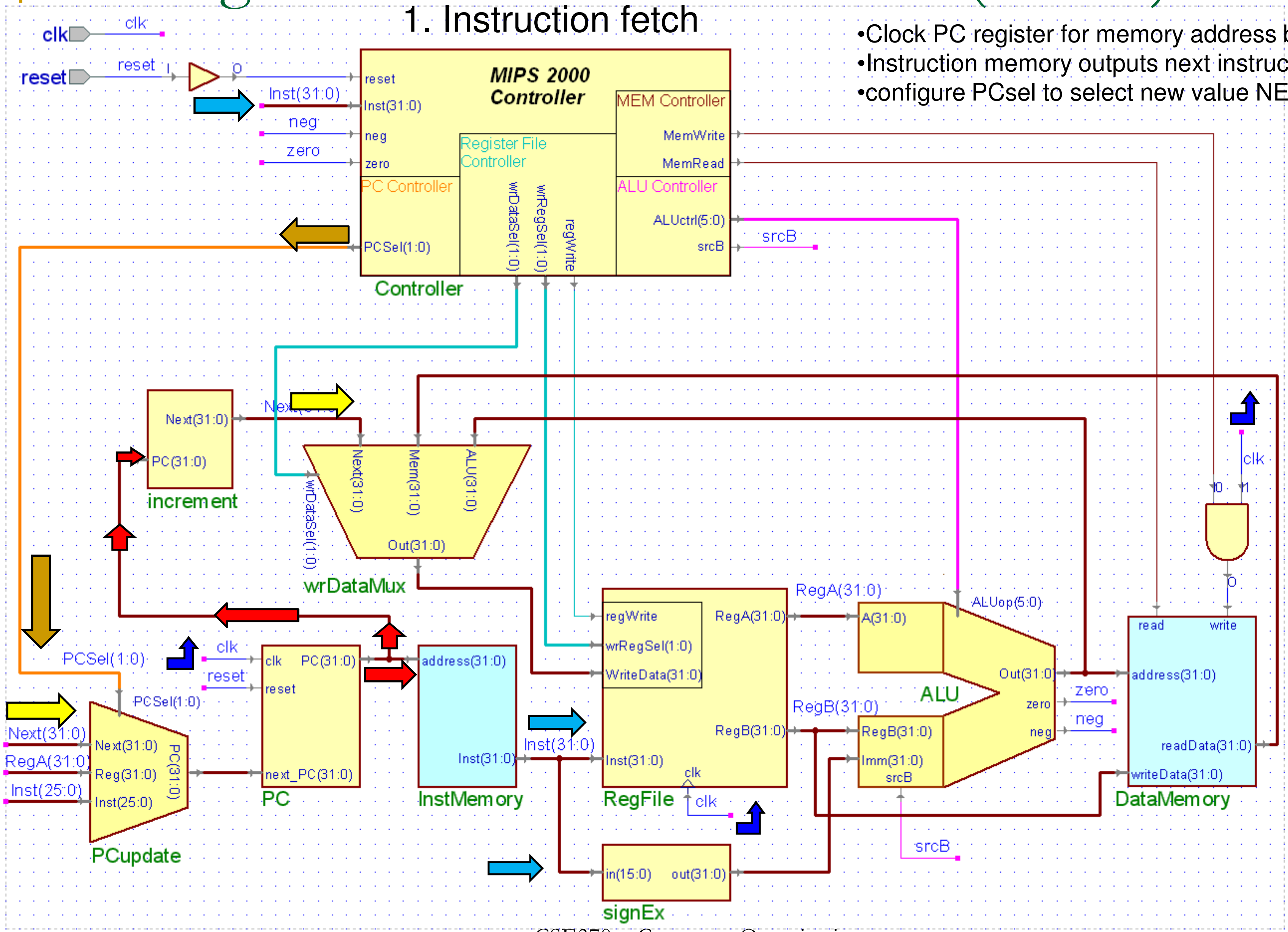
R	0	rs=r1	rt=r2	rd=r3	funct=0

- Phase 1. instruction fetch
  - Clock PC register for memory address bus
  - Instruction memory outputs next instruction
  - configure PCsel to select new value NEXT
- Phase 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
- Phase 3. instruction execute
  - set up ALU inputs
  - configure ALU to perform ADD operation
  - configure register file to store ALU result (rd)

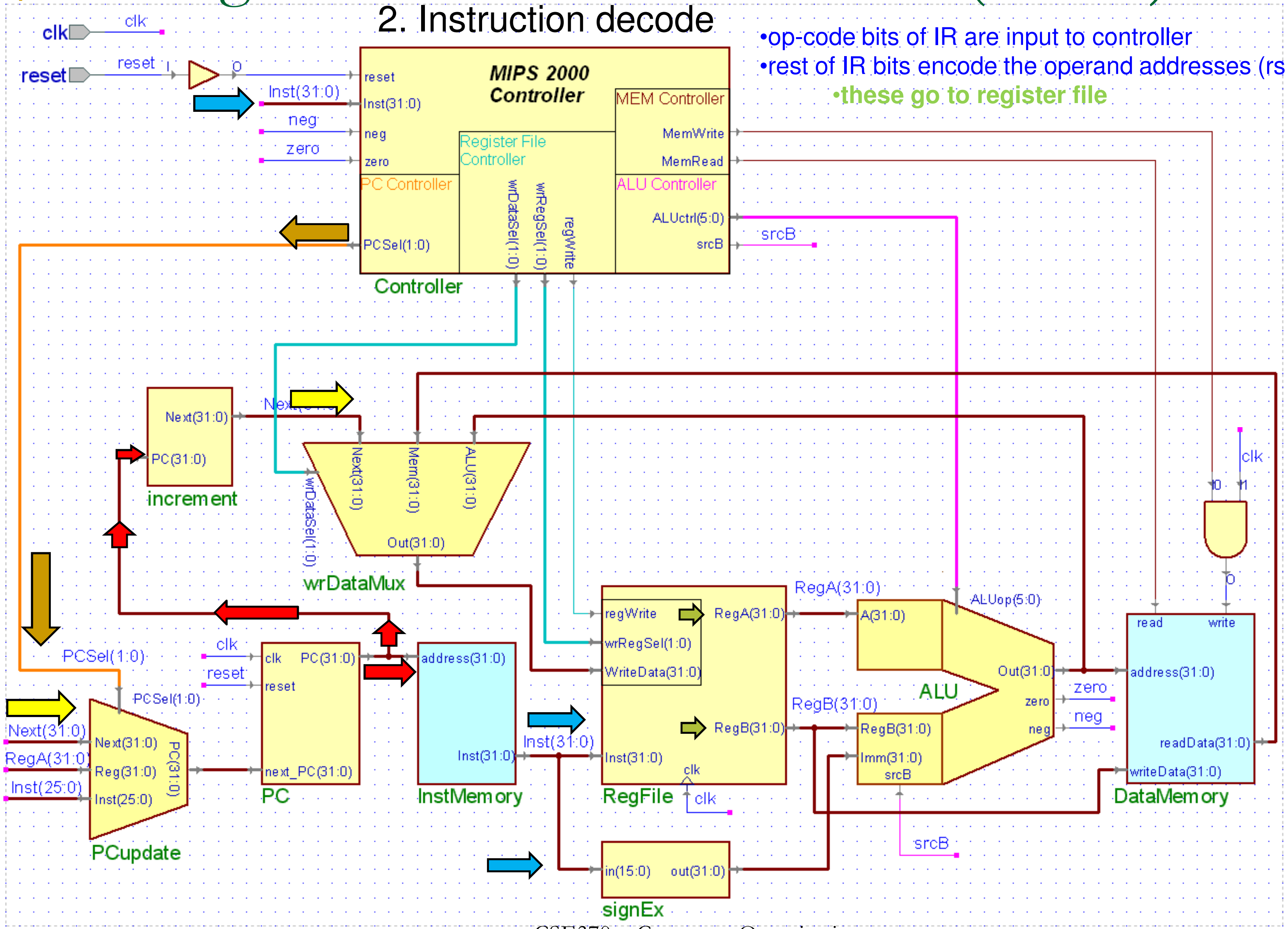
Tracing an instruction's execution (cont'd)

1. Instruction fetch

- Clock PC register for memory address bus
- Instruction memory outputs next instruction
- configure PCsel to select new value NEXT

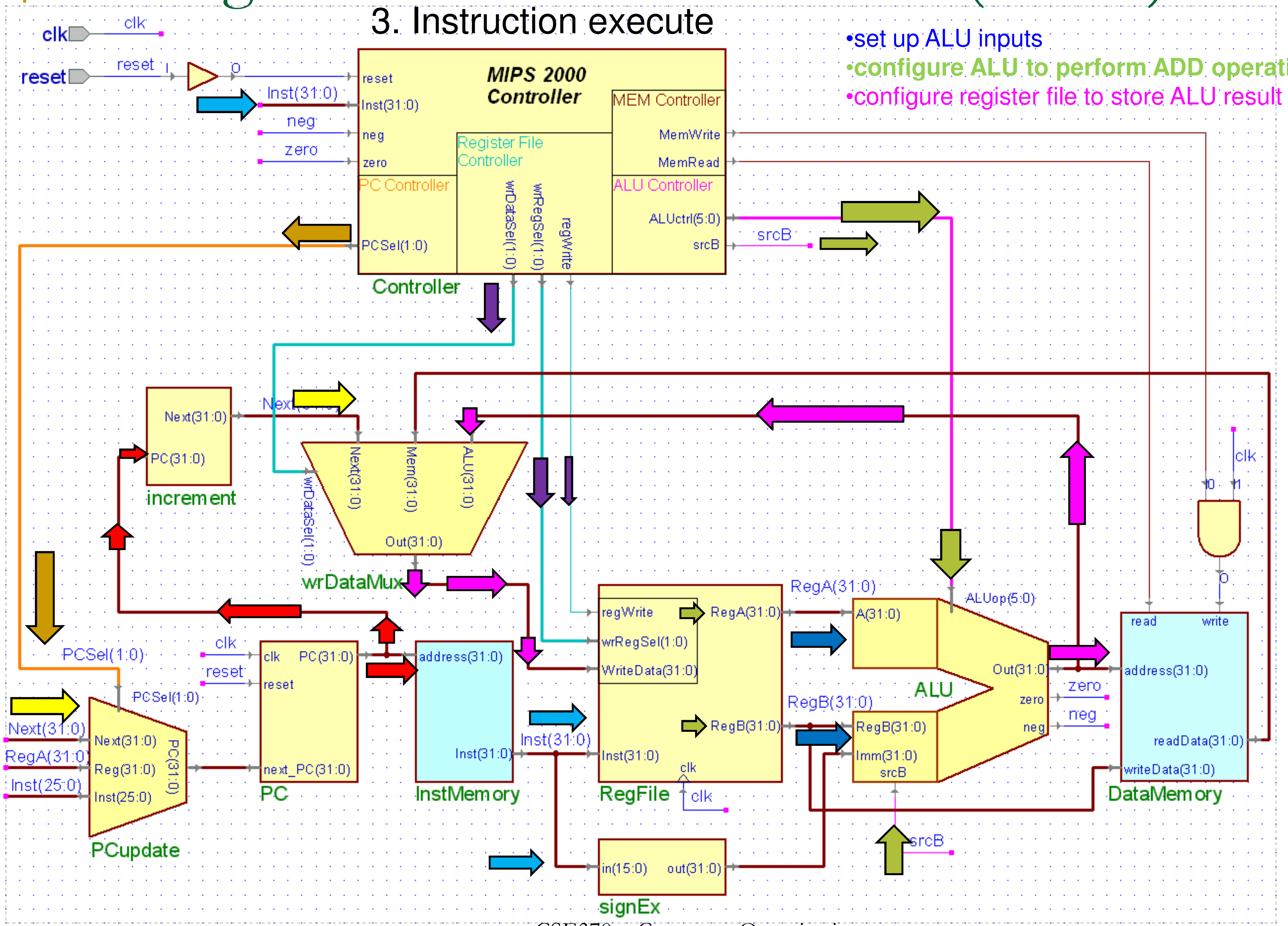


Tracing an instruction's execution (cont'd)



- op-code bits of IR are input to controller
- rest of IR bits encode the operand addresses (rs and rt)
  - these go to register file

Tracing an instruction's execution (cont'd)



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:

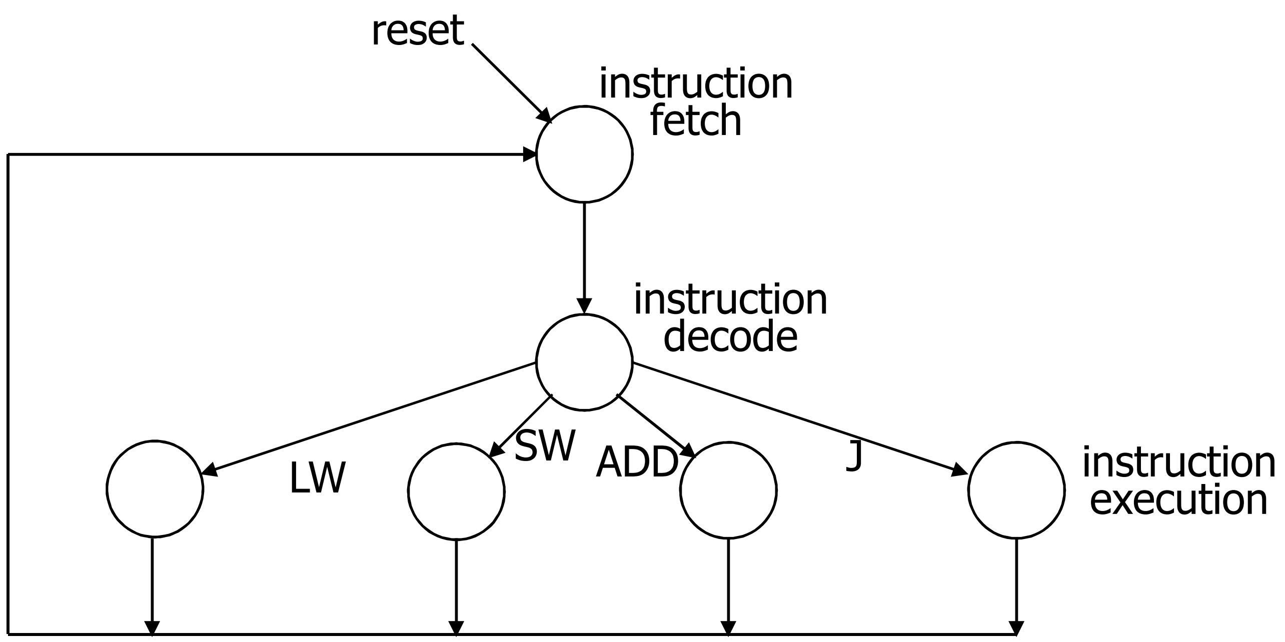
op ← add	– send regA into A input, regB into B input, add (srcA, srcB0, scrB1, op)
rd ← ALUout	– store result of add into destination register (regWrite, wrDataSel, wrRegSel)

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, All operations take 1 (long) cycle
- 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)

FSM controller for CPU (skeletal Moore FSM)

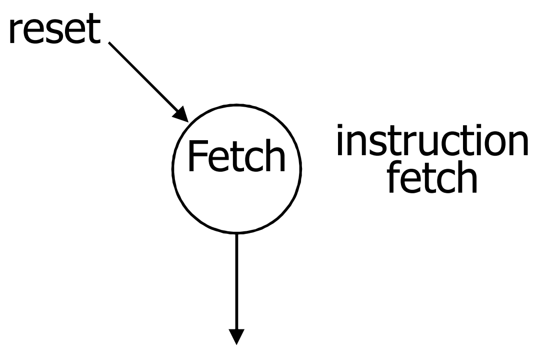
- First pass at deriving the state diagram (Moore machine)
  - these will be further refined into sub-states



FSM controller for CPU (reset and inst. fetch)

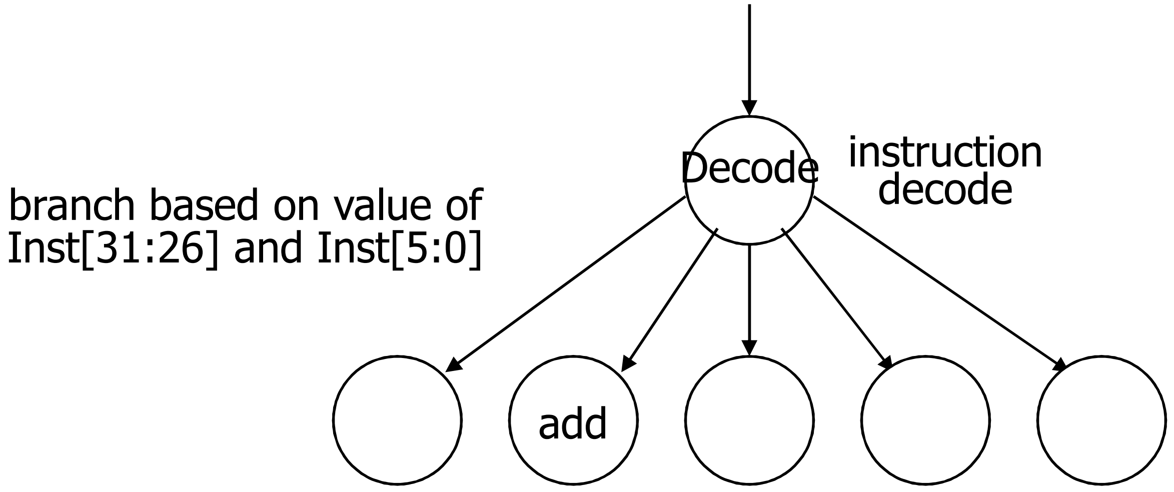
- Assume Moore machine
  - outputs associated with states rather than arcs
- Reset state and instruction fetch sequence
- On reset (go to Fetch state)
  - start fetching instructions
  - PC will set itself to zero

```
mabus ← PC;
memory read;
IR ← memory data bus;
PC ← PC + 1;
```



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?

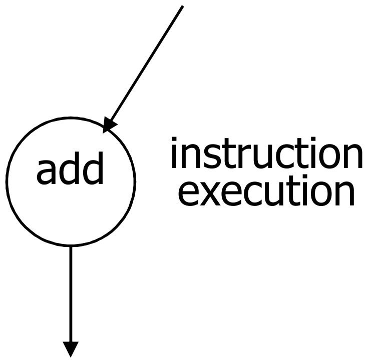


FSM controller for CPU (instruction execution)

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

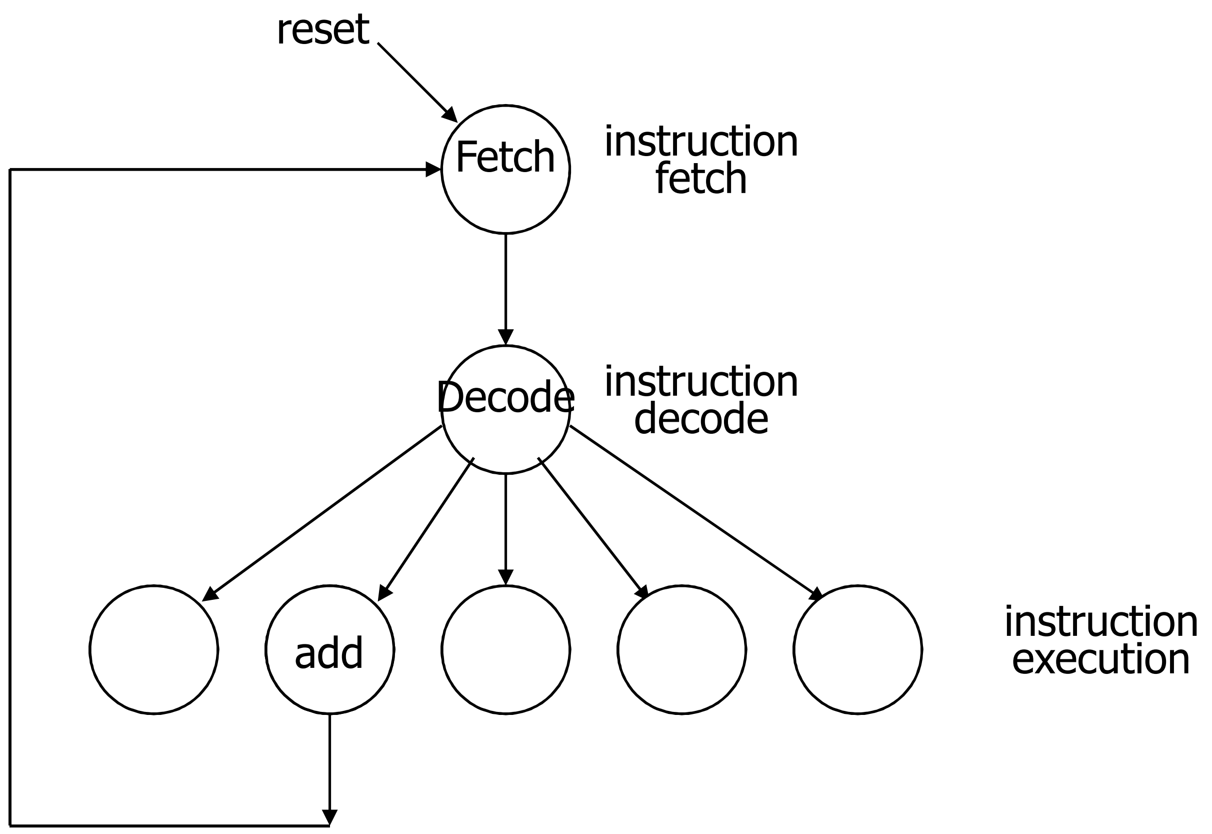
    rd ← A + B

  - 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