Levels in Processor Design

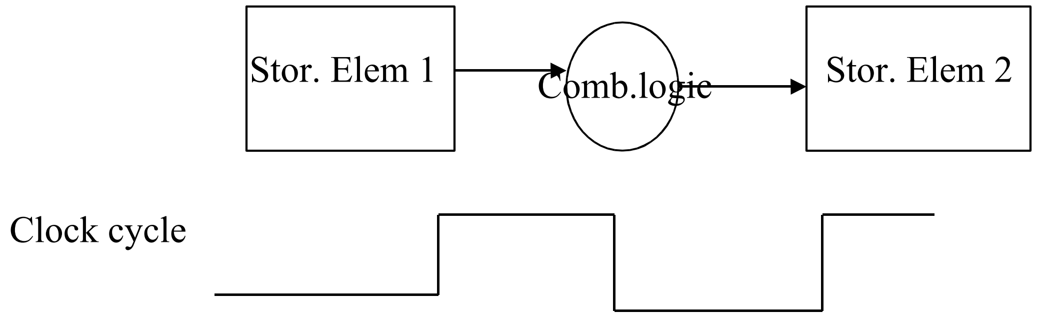
- Circuit design
  - Keywords: transistors, wires etc.Results in gates, flip-flops etc.
- Logical design
  - Putting gates (AND, NAND, …) and flip-flops together to build basic blocks such as registers, ALU's etc (cf. CSE 370)
- ***Register transfer***
  - Describes execution of instructions by showing data flow between the basic blocks
- **Processor description** (the ISA)
- System description
  - Includes memory hierarchy, I/O, multiprocessing etc

Register transfer level

- Two types of components (cf. CSE 370)
  - *Combinational* : the output is a function of the input (e.g., adder)
  - *Sequential*: state is remembered (e.g., register)

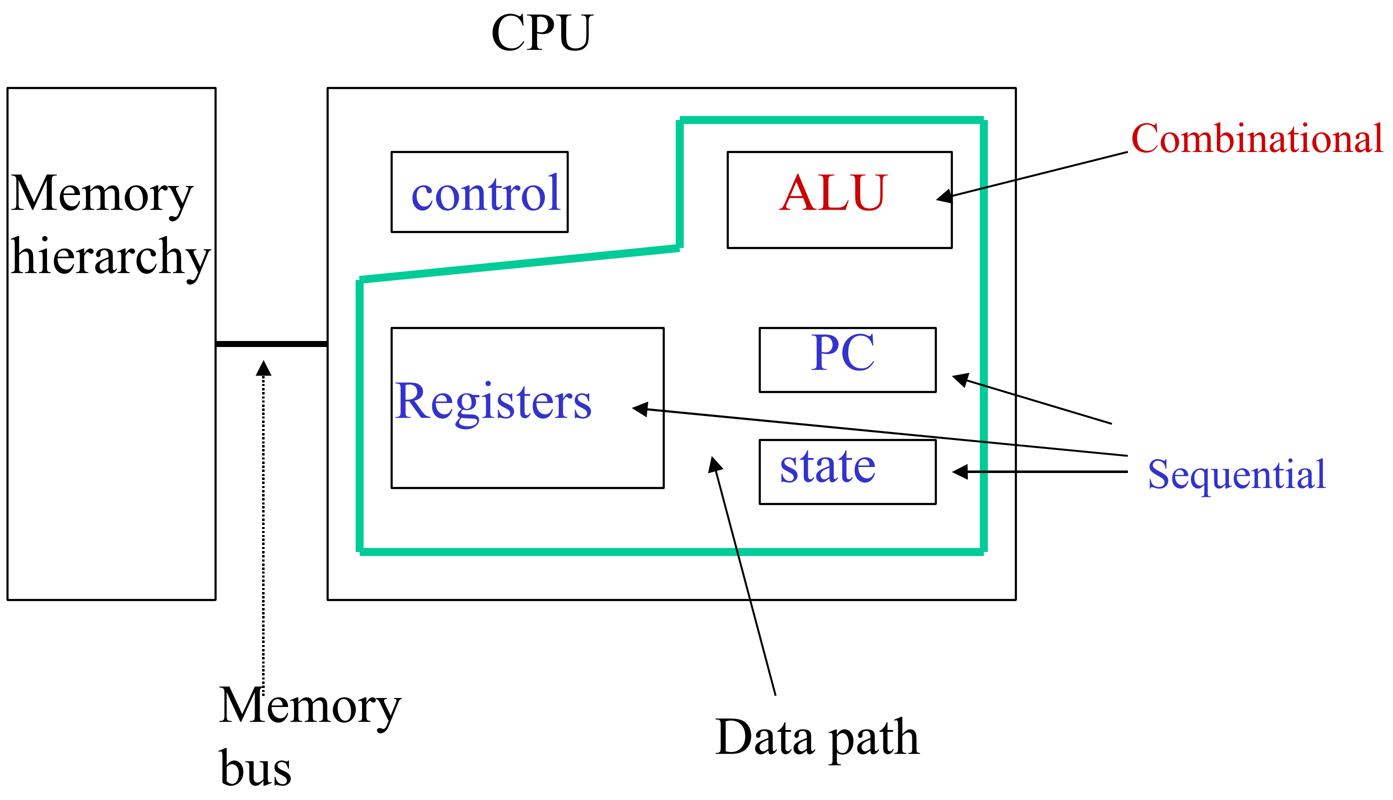
Synchronous design

- Use of a periodic clock
  - *edge-triggered* clocking determines when signals can be read and when the output of circuits is stable
  - Values in storage elements can be updated only at clock edges
  - Clock tells when events can occur, e.g., when signals sent by control unit are obeyed in the ALU



Note: the same storage element can be read/written in the same cycle

Processor design: data path and control unit

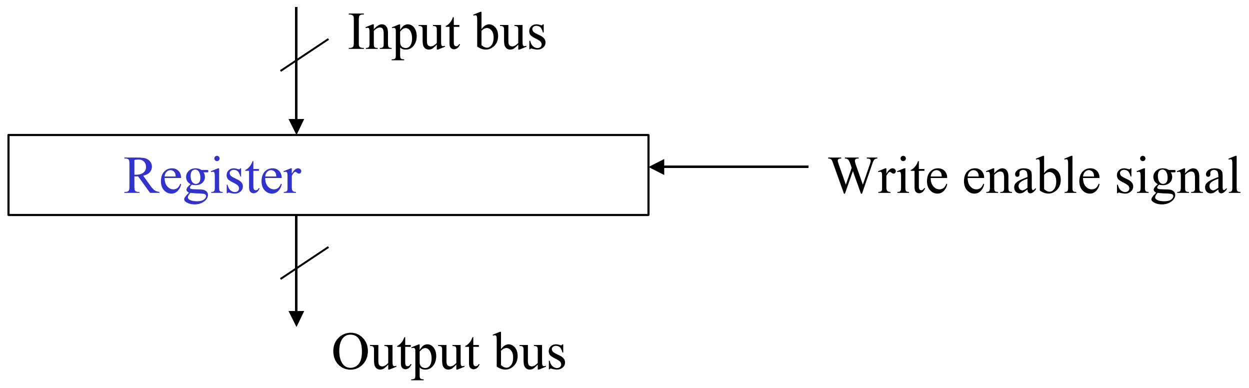


Processor design

- Data path
  - How does data flow between various basic blocks
  - What operations can be performed when data flows
  - What can be done in one clock cycle
- Control unit
  - Sends signals to data path elements
  - Tells what data to move, where to move it, what operations are to be performed
- Memory hierarchy
  - Holds program and data

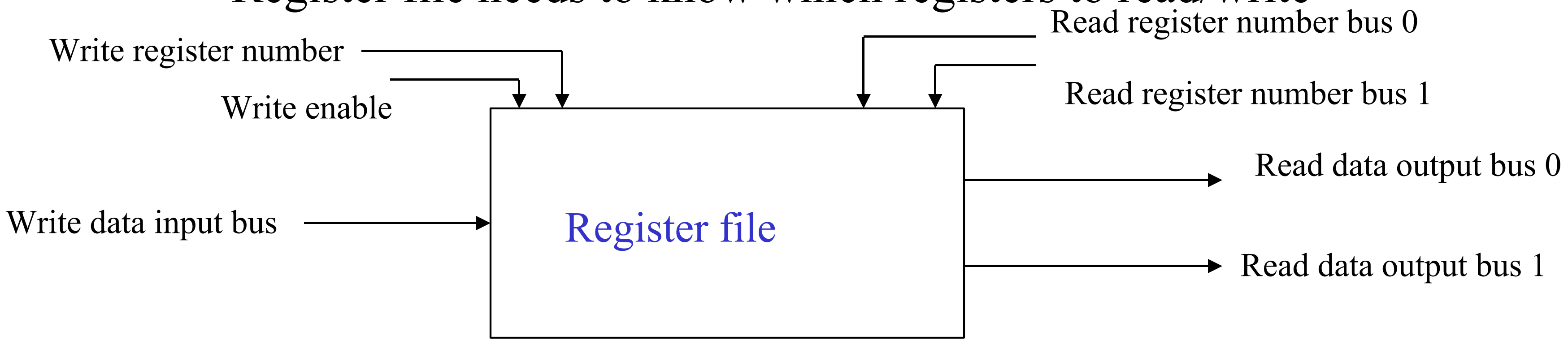
Data path basic building blocks. Storage elements

- Basic building block (at the RT level) is a register
- In our mini-MIPS implementation registers will be 32-bits
- A register can be read or written



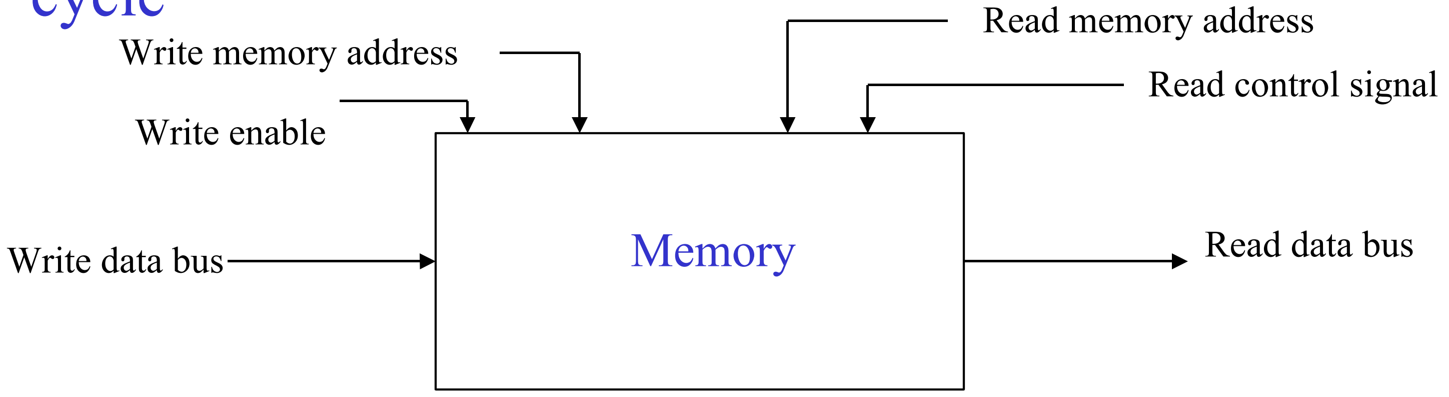
Register file

- Array of registers (32 for the integer registers in MIPS)
- ISA tells us that we should be able to:
  - read 2 registers, write one register in a given instruction (at this point we want one instruction per cycle)
  - Register file needs to know which registers to read/write



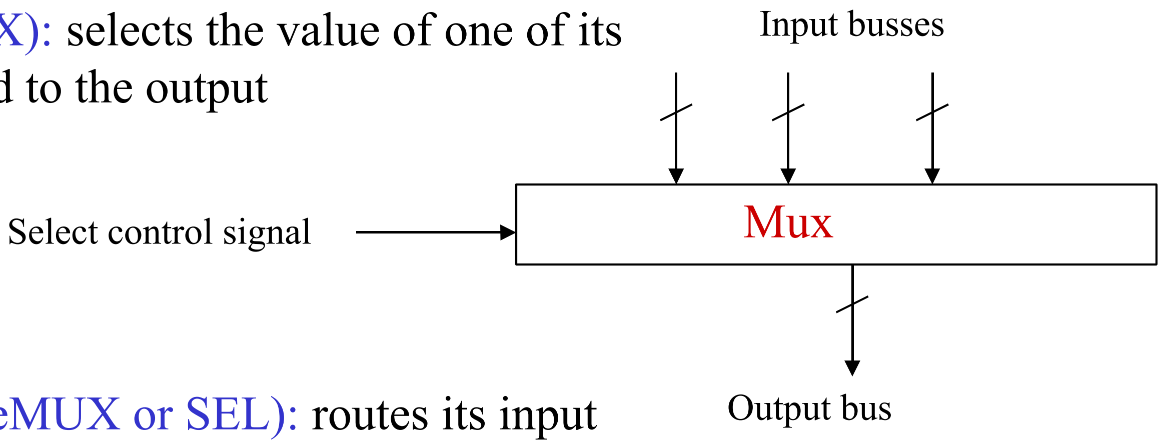
Memory

- Conceptually, like register file but much larger
- Can only read one location or write to one location per cycle

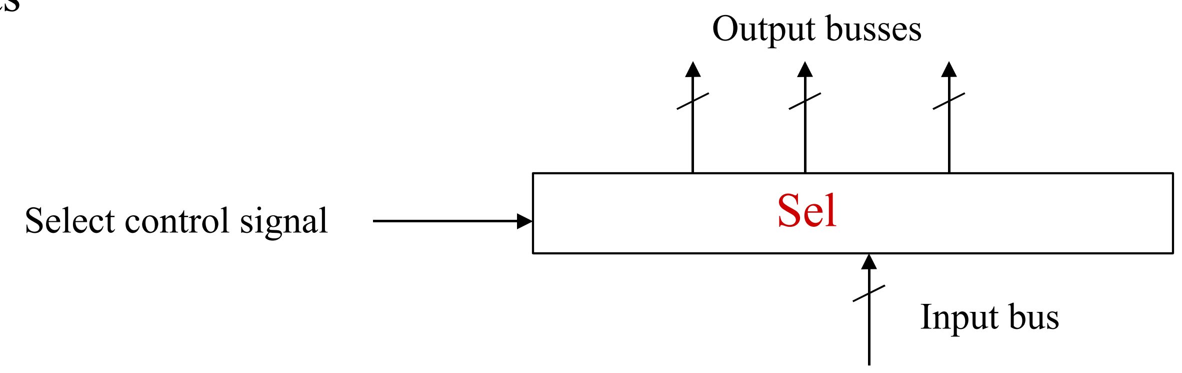


Combinational elements

Multiplexor (MUX): selects the value of one of its inputs to be routed to the output

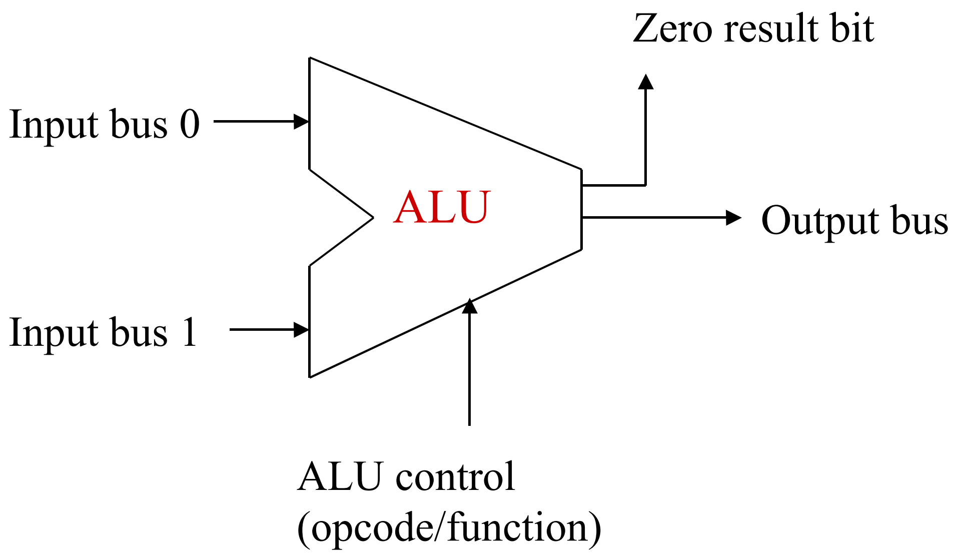


Demultiplexor (deMUX or SEL): routes its input to one of its outputs

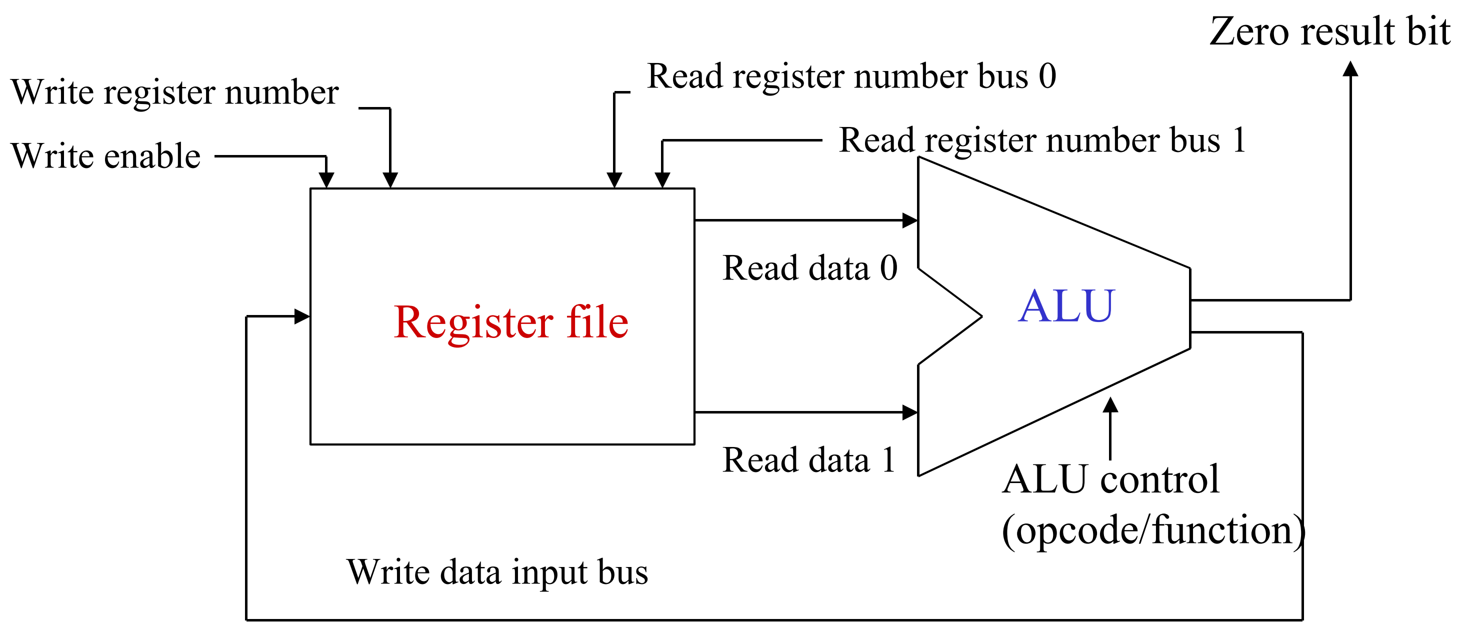


Arithmetic and Logic Unit (ALU - combinational)

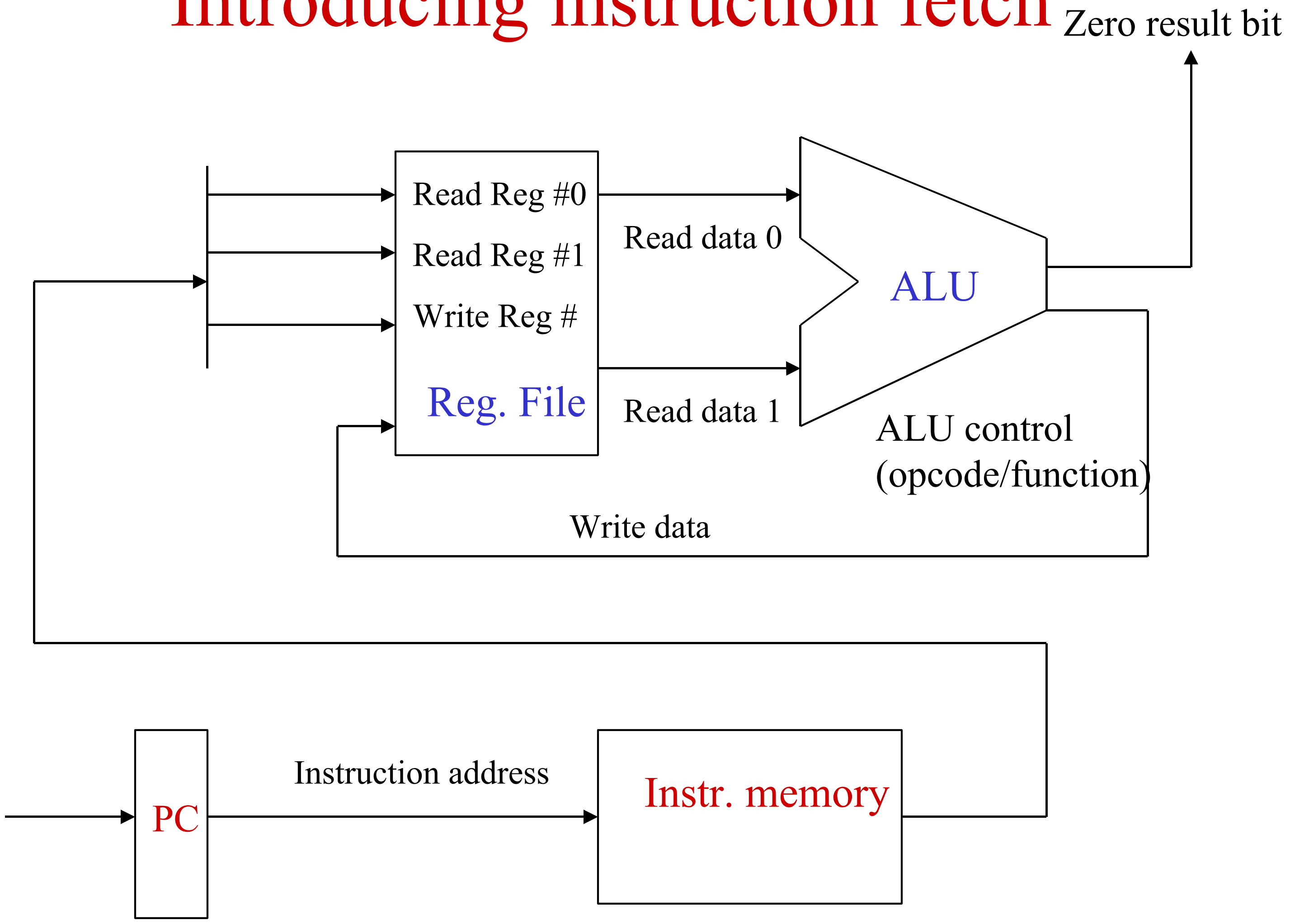
- Computes (arithmetic or logical operation) output from its two inputs



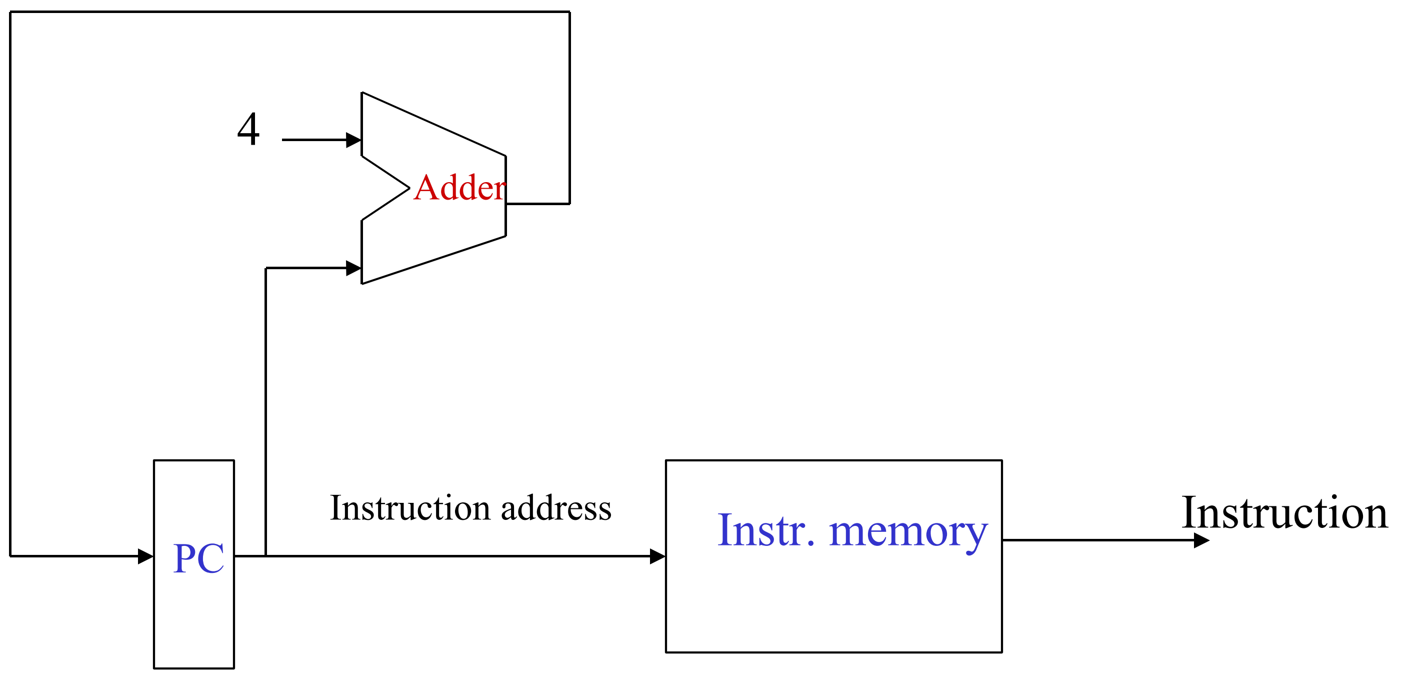
Putting basic blocks together (skeleton of data path for arith/logical operations)



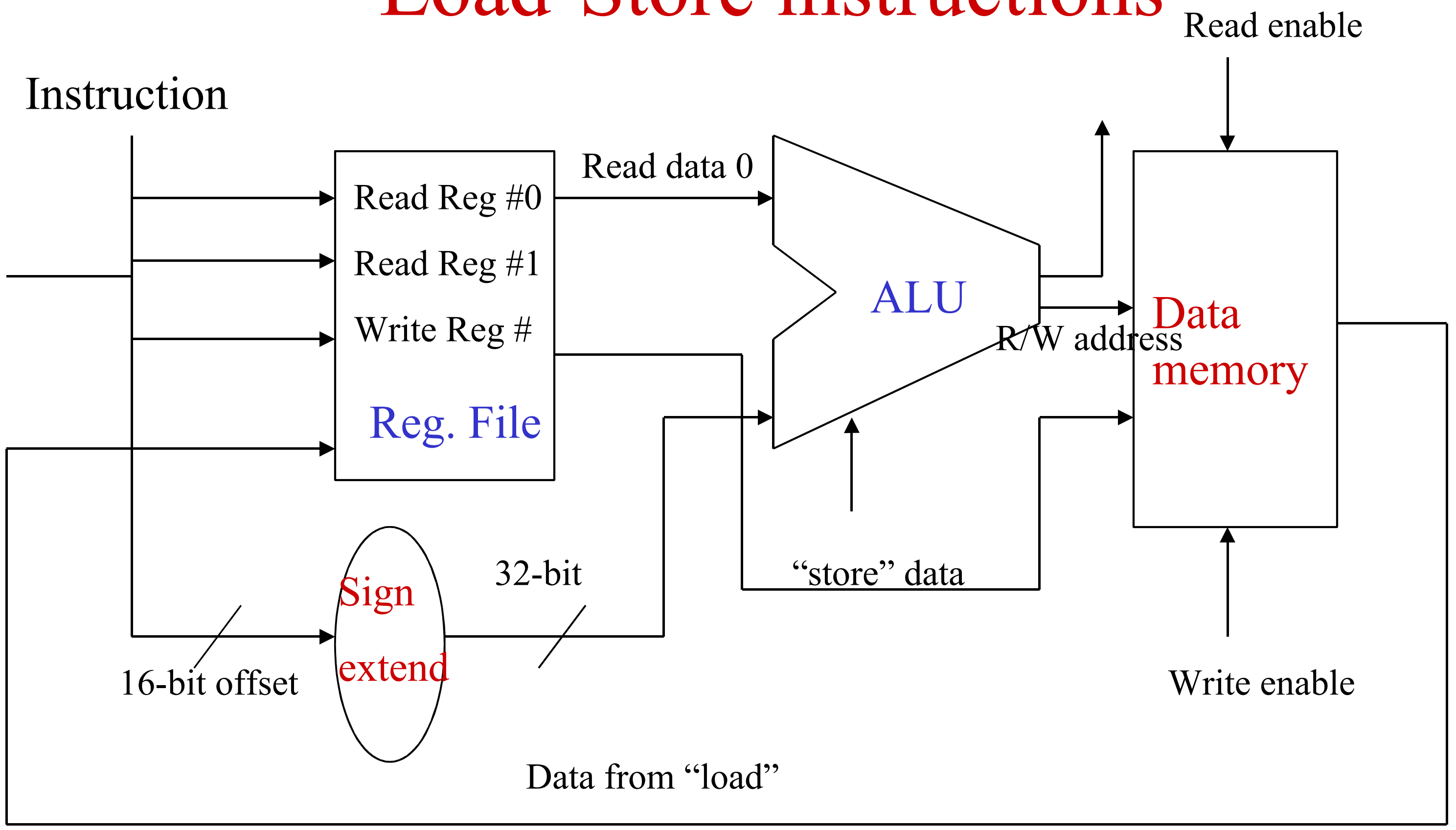
Introducing instruction fetch



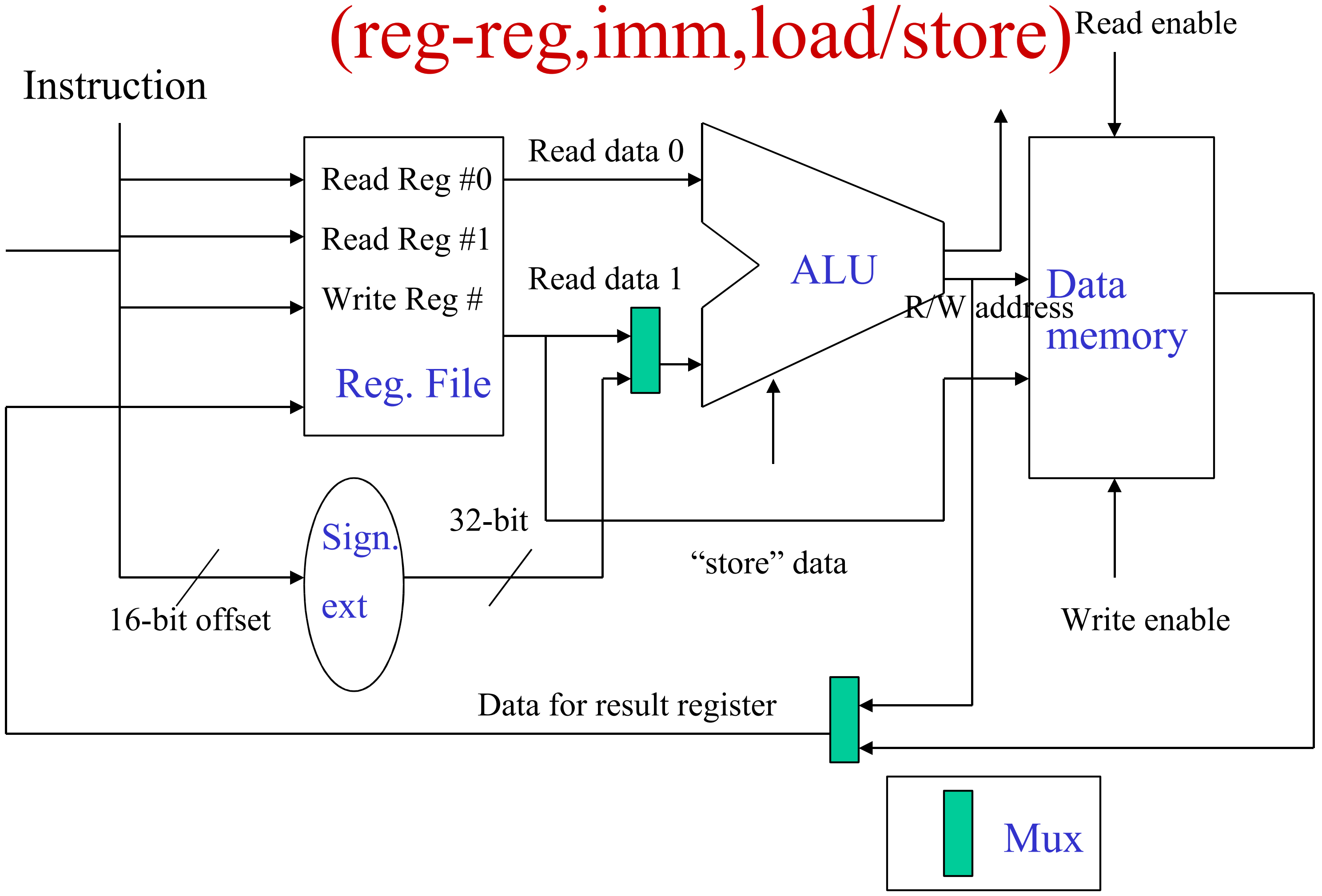
PC has to be incremented (assume no branch)



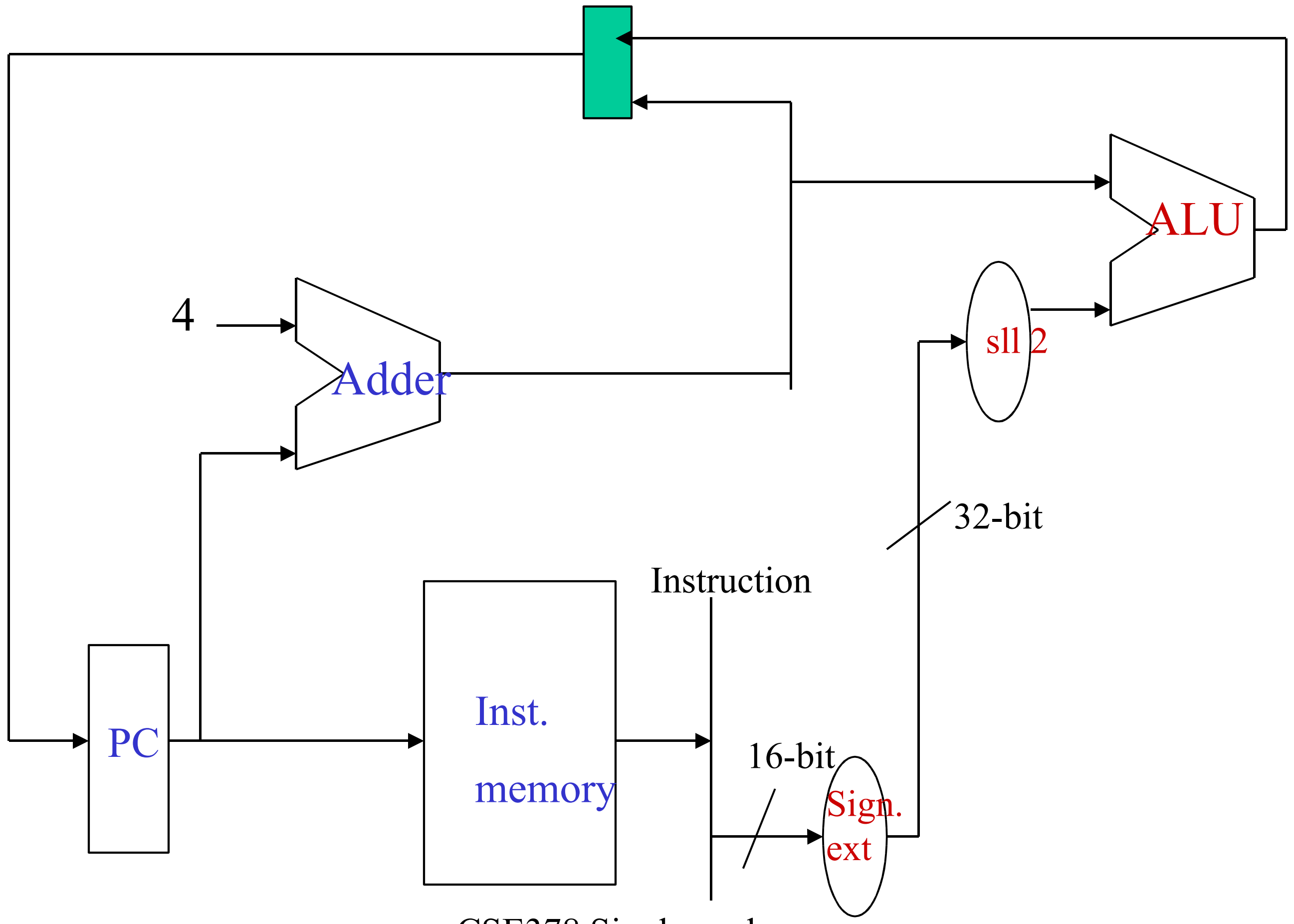
Load-Store instructions



Data path for straight code (reg-reg,imm,load/store)



Branch data path



Control Unit

- Control unit sends control signals to data path and memory depending
  - on the opcode (and function field)
  - results in the ALU (for example for Zero test)
- These signals control
  - muxes; read/write enable for registers and memory etc.
- Some "control" comes directly from instruction
  - register names
- Some actions are performed at every instruction so no need for control (in this single cycle implementation)
  - incrementing PC by 4; reading instr. memory for fetching next inst.

Building the control unit

- Decompose the problem into
  - Data path control (register transfers)
  - ALU control
- Setting of control lines by control unit totally specified in the ISA
  - for ALU by opcode + function bits if R-R format
  - for register names by instruction (including opcode)
  - for reading/writing memory and writing register by opcode
  - muxes by opcode
  - PC by opcode + result of ALU

Example

- Limit ourselves to:
  - R-R instructions: add, sub, and, or, slt –
    - OPcode = 0 but different function bits
  - Load-store: lw, sw
  - Branch: beq
- ALU control
  - Need to *add* for: add, lw, sw
  - Need to *sub* for: sub, beq
  - Need to *and* for :and
  - Need to *or* for :or
  - Need to *set less than* for : slt

ALU Control

- ALU control: combination of opcode and function bits
- Decoding of opcodes yields 3 possibilities hence 2 bits
  - AluOp1 and ALUOp2
- ALU control:
  - Input 2 ALUop bits and 6 function bits
  - Output one of 5 possible ALU functions
  - Of course lots of don’t care for this *very* limited implementation

Implementation of Overall Control Unit

- Input: opcode (and function bits for R-R instructions)
- Output: setting of control lines
- Can be done by logic equations
- If not too many, like in RISC machines
  - Use of PAL's (cf. CSE 370).
  - In RISC machines the control is "hardwired"
- If too large (too many states etc.)
  - Use of microprogramming (a microprogram is a hardwired program that interprets the ISA)
- Or use a combination of both techniques (Pentium)

Where are control signals needed (cf. Figure 5.15)

- Register file
  - RegWrite (Register write signal for R-type, Load)
  - RegDst (Register destination signal: rd for R-type, rt for Load)
- ALU
  - ALUSrc (What kind of second operand: register or immediate)
  - ALUop (What kind of function: ALU control for R-type)
- Data memory
  - MemRead (Load) or MemWrite (Store)
  - MemtoReg (Result register written from ALU or memory)
- Branch control
  - PCSrc (PC modification if branch is taken)

How are the control signals asserted

- Decoding of the opcode by control unit yields
  - Control of the 3 muxes (RegDst, ALUSrc,MemtoReg): 3 control lines
  - Signals for RegWrite, Memread,Memwrite: 3 control lines
  - Signals to activate ALU control (e.g., restrict ourselves to 2)
  - Signal for branch (1 control line)
    - decoding of opcode ANDed with ALU zero result
- Input Opcode: 6 bits
- Output 9 control lines (see Figure 5.17)