Datapath & Control Design

- We will design a simplified MIPS processor
- The instructions supported are
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers

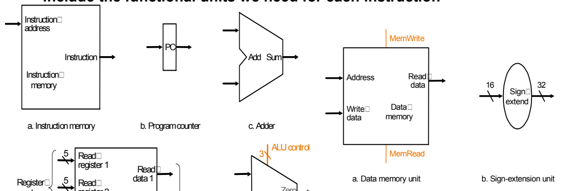
  Why? memory-reference? arithmetic? control flow?

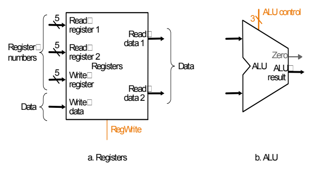
What blocks we need

- We need an ALU
  - We have already designed that
- We need memory to store inst and data
  - Instruction memory takes address and supplies inst
  - Data memory takes address and supply data for lw
  - Data memory takes address and data and write into memory
- We need to manage a PC and its update mechanism
- We need a register file to include 32 registers
  - We read two operands and write a result back in register file
- Some times part of the operand comes from instruction
- We may add support of immediate class of instructions
- We may add support for J, JR, JAL

Simple Implementation

- Include the functional units we need for each instruction

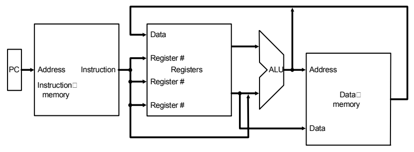




Why do we need this stuff?

More Implementation Details

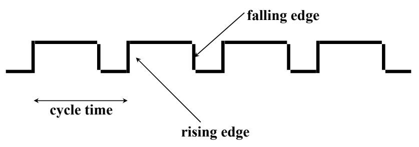
- Abstract / Simplified View:



- Two types of functional units:
  - elements that operate on data values (combinational)
    - Example: ALU
  - elements that contain state (sequential)
    - Examples: Program and Data memory, Register File

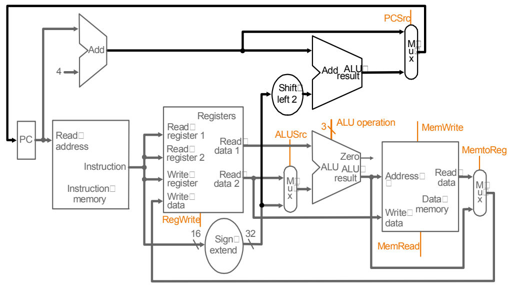
Managing State Elements

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



Building the Datapath

- Use multiplexors to stitch them together



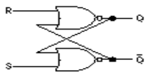
Latches and Flip flops

- Output is equal to the stored value inside the element
  (don't need to ask for permission to look at the value)
  - "logically true" could mean electrically low
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge
  (edge-triggered methodology)

A clocking methodology defines when signals can be read and written
— wouldn't want to read a signal at the same time it was being written

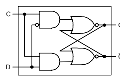
An unclocked state element

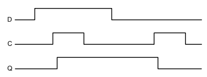
- The set-reset latch
  - output depends on present inputs
  - If present inputs are 00, then it depends on the past inputs
  - What happens if R=1, S=1?



D latch

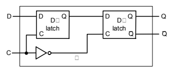
- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement

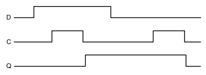




D flip flop

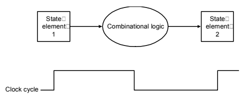
- Output changes only on the clock edge





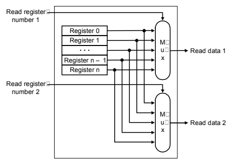
Our Implementation

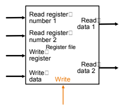
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

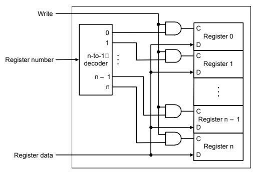


Register File

- Built using D flip-flops


Read register number 1
Register 0
Register 1
Register n – 1
Register n
Mux
Read data 1
Read register number 2
Mux
Read data 2


Read register number 1
Read data 1
Read register number 2
Register file
Write register
Write data
Read data 2
Write


Write
Register number
n-to-1 decoder
0
1
n – 1
n
Register data
C
D
Register 0
Register 1
Register n – 1
Register n