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

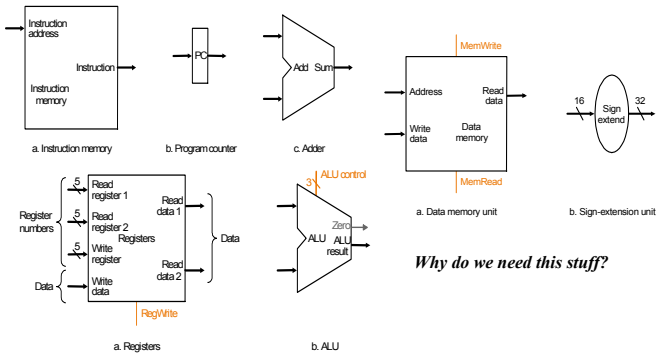
  Why? memory-reference? arithmetic? control flow?

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

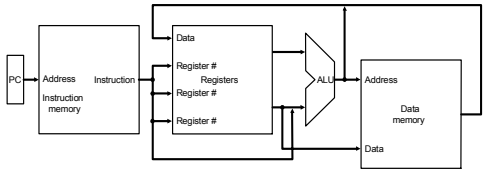
- Include the functional units we need for each instruction



Why do we need this stuff?

More Implementation Details

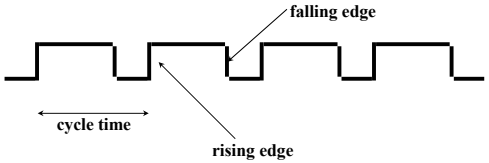
- Abstract / Simplified View:



- 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

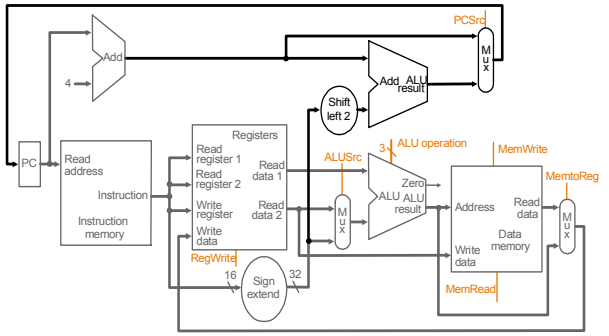
Managing State Elements

- 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



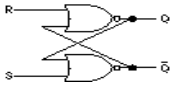
Latches and Flip-flops

- 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

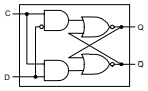
An unclocked state element

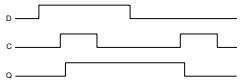
- The set-reset latch
  - output depends on present inputs
  - If present inputs are 00, then it depends on the past inputs
  - What happens if R=1, S=1?



D-latch

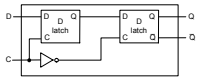
- **Two inputs:**
  - **the data value to be stored (D)**
  - **the clock signal (C) indicating when to read & store D**
- **Two outputs:**
  - **the value of the internal state (Q) and its complement**

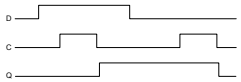




D flip-flop

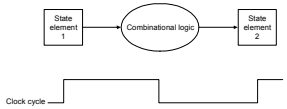
- **Output changes only on the clock edge**





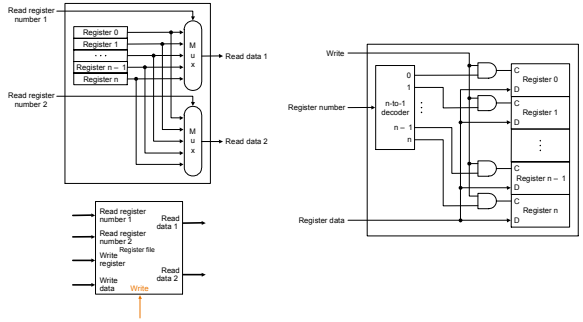
Our Implementation

- **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**



Data Path Composition

Data paths for inst classes

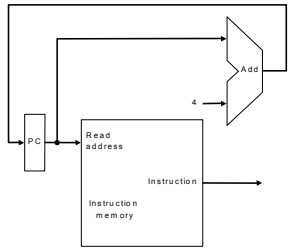
1. Arithmetic-logic
2. Memory references
3. Branch and Jump

Data path stages

- Instruction fetch
- Read operands
- ALU operation
- Memory access
- Register write
- PC Update

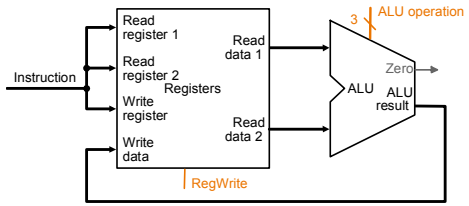
Instruction Fetch

- PC determines the next instruction to fetch (and to execute)
- Branch and jump changes PC



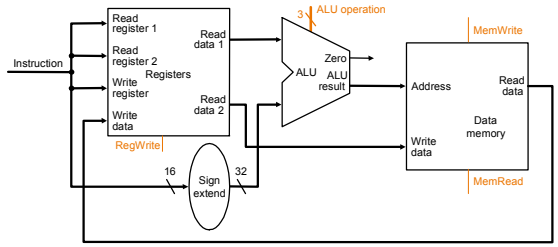
Datapath for R-type Instructions

1. Instruction fetch
2. Read registers: for the two source operands
3. ALU: perform the arithmetic-logic operation
4. Write register: to the destination register



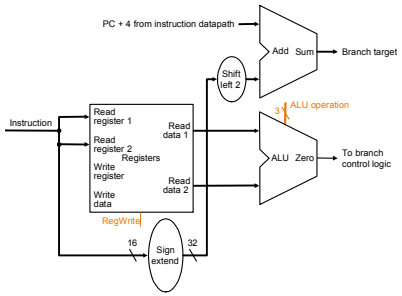
Datapath for Memory Reference Instructions

1. Instruction fetch
2. Read registers: base address and data (for load)
3. ALU: calculating effective address
4. Memory access: read/write data
5. Write register (for store)



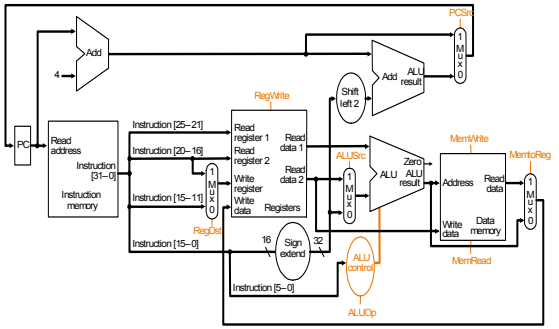
Datapath for Branch Instructions

1. Instruction fetch
2. Read registers: two operands in beq
3. ALU: compare two operands
4. Update PC



A Complete Datapath for Core Instructions

- Supports Lw, Sw, Add, Sub, And, Or, Slt, and Beq
- All control lines identified

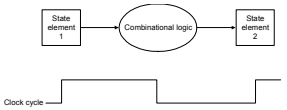


What Else is Needed in Datapath

- Support for j and jr
  - For both of them PC value need to come from somewhere else
  - For J, PC is created by 4 bits (31:28) from old PC, 26 bits from IR (27:2) and 2 bits are zero (1:0)
  - For JR, PC value comes from a register
- Support for JAL
  - Address is same as for J inst
  - OLD PC needs to be saved in register 31
- And what about immediate operand instructions
  - Second operand from instruction, but without shifting
- Support for other instructions like lw and immediate inst write

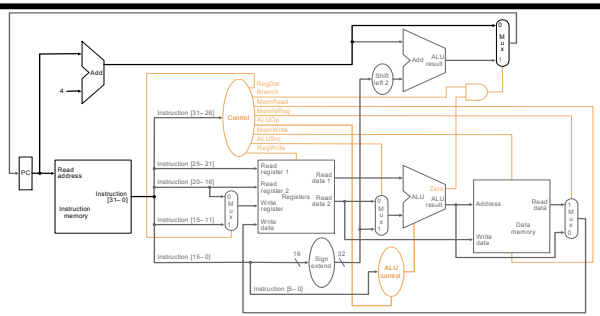
Single-cycle Implementation

- Execute every instruction in one cycle
  - Simple implementation with simple control
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - We use write signals along with clock to determine when to write
  - No single element can be used twice
- Cycle time determined by length of the longest path



We will study more clever implementation

Adding Control to Datapath



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Main Control Descriptions

Signal name	When deasserted	When asserted
RegDst	**Dest reg number ← $rt**	**Dest reg number ← $rd**
RegWirte		**Write register**
ALUSrc	**2nd ALU input ← $rt**	**2nd ALU input ← I-field**
PCSrc (Branch)	**PC ← PC+4**	**PC ← address ALU output**
MemRead		**Enable data memory read**
MemWrite		**Enable data memory write**
MemtoReg .	**Reg data ← main ALU output**	**Reg data← data memory**

Q: What determines the values of those signals?