Processor Implementation

Forecast -- heart of course -- key to project

- Sequential logic design review
- Clocking methodology
- Datapath -- 1 CPI
    - single instruction, 2’s complement, unsigned
- Control
- Multiple cycle implementation
- Microprogramming
- Exceptions

Review Sequential Logic

Logic is combinational if output is solely function of inputs

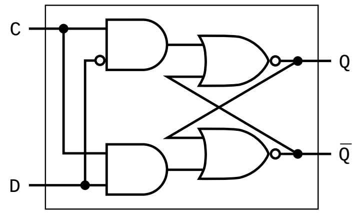
e.g., ALU of previous lecture

Logic is sequential or “has state” if output function of

- past and current inputs
- past inputs “remembered” in “state”
- but no magic!

Review Sequential Logic

E.g., D latch



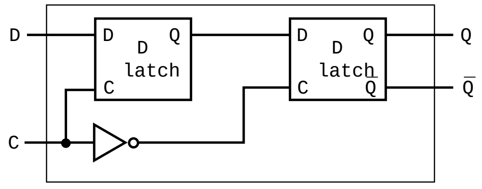
Clock high Q = D, $\overline{Q}$ = $\overline{D}$ after some min to max propagate delay

clock low Q, $\overline{Q}$ remain unchanged

sensitive to clock level

Review Sequential Logic

E.g., D flip-flop



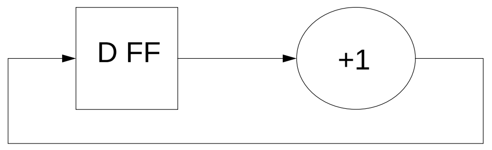
While clock high, D flows into 1st latch, but not 2nd

There Q retains old value

Remebmer D at falling edges and propagates through 2nd latch

Review Sequential Logic

Can build



Why does this fail for latch?

Clocking Methodology

Motivation

- Design datapath/control without thinking about clocks
- Use convention called clocking methodology
    - restricts design freedom
    - hides complexity

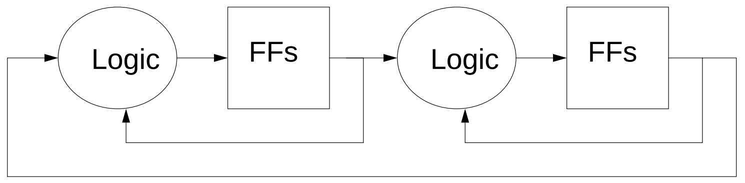
Our Methodology

Only Flip-flops

All on same edge (e.q., falling)

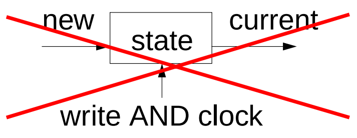
All with same clock (omit from pictures)

All logic finishes in one clock cycle

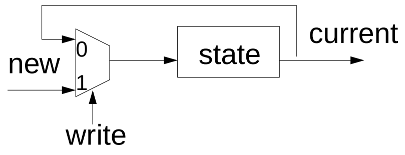


Our Methodology, cont.

Do *NOT* use qualified clocks:I



CorrectI



Datapath - 1 CPI

Assumption: Get whole instruction done in one long cycle

Instructions: **`add, sub, and, or, slt, lw, sw, & beq`**

To do

- For single instruction
- Put it together

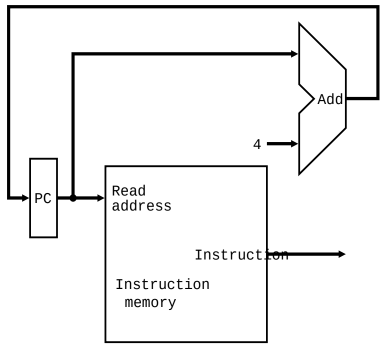
Fetch Instructions

Fetch instruction,
then increment PC

Assumes

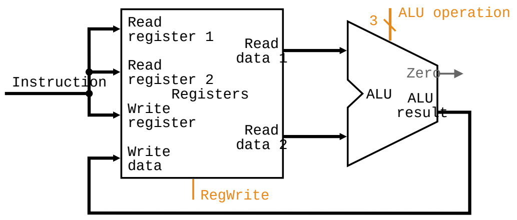
- PC updated every cycle
- no branch or jumps

after this instruction,
do different things



ALU Instructions

And $1, $2, $3   # $1 destination



Use R-format

opcode rs rt rd shamt funct

6 5 5 5 5 6

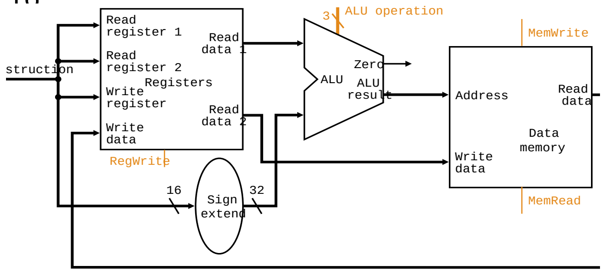
Load/Store Instructions

xw $1, immed($2)   # $1 <-> M[SE(immed)+$2]

Use I-format

opcode rs rt immed

6 5 5 16



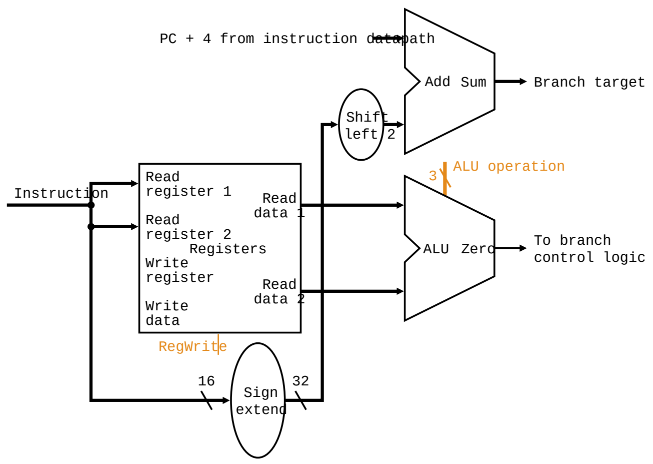
Branch Instructions

beq $1, $2, addr   # if ($1 == $2) PC = PC + addr<<2

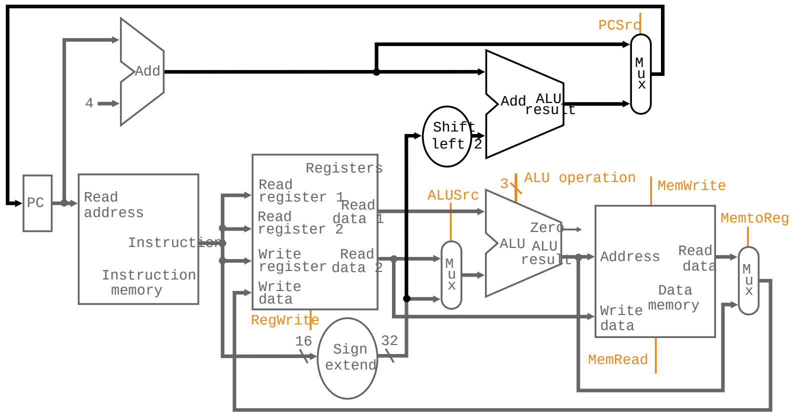
Actually

- newPC = PC + 4
- target = newPC + addr<<2 # in MIPS offset from newPC
- if ($1-$2 == 0)
-     PC = target
- else
-     PC = newPC

Branch Instructions



All Together


PCSrc
Add
4
M u x
Add ALU result
Shift left 2
PC
Read address
Instruction
Instruction memory
Registers
Read register 1
Read data 1
Read register 2
Write register
Read data 2
Write data
RegWrite
ALUSrc
M u x
3
ALU operation
Zero
ALU ALU result
MemWrite
MemtoReg
Address
Read data
Data memory
Write data
M u x
MemRead
16
Sign extend
32