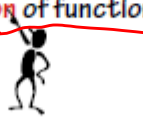
Complex System Design

Why have processing blocks?

The goal of modular design: Abstraction

What does that mean anyway:

- Rules simple enough for a 6-3 to follow…
- Understanding BEHAVIOR without knowing IMPLEMENTATION
- Predictable composition of functions
- Tinker-toy assembly
- Guaranteed behavior, under REAL WORLD circumstances



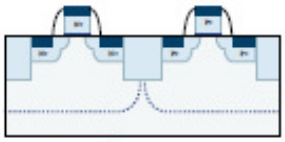
How do you build systems with >1G components?

Personal Computer: Hardware & Software

Circuit Board: ~1-8 / system @ 1-2G devices

Integrated Circuit: ~8-16 / PCB .25M-1G devices

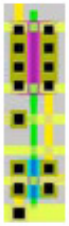
Module: ~8-64 / IC .1M-1M devices



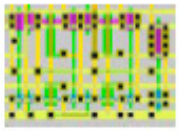
MOSFET

Scheme for representing information

Figure by MIT OpenCourseWare.



Gate: ~2-16 / Cell 8 devices

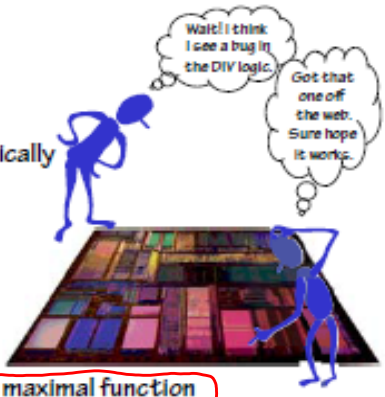


Cell: ~1K-10K / Module 16-64 devices

6.004 – Spring 2009 2/5/09 L01 - Basics of Information 4

What do we see?

- Structure
  - hierarchical design:
    - limited complexity at each level
    - reusable building blocks
- Interfaces
  - Key elements of system engineering; typically outlive the technologies they interface
  - Isolate technologies, allow evolution
  - Major abstraction mechanism
- What makes a good system design?
  - "Bang for the buck": minimal mechanism, maximal function
  - reliable in a wide range of environments
  - accommodates future technical improvements



general purpose

Implement MIPS

- Generally decompose digital systems into two kinds of operation
  - Things that deal with the real data (Datapath)
  - Things that control the stuff operating on the real data (Control) FSM, μSequencer
- Find a decomposition that is simple, and efficient
  - Some are obvious, others can be more subtle
- We will start simple
  - Add stuff to improve performance

- First we need to:
  1. Fetch the instruction
- Then we need to:
  2. Decode instruction / fetch register
- Then we need to:
  3. Do the operation
- Then we need to:
  4. Write the result into register-file
- Finally we need to:
  5. Calculate the next instruction address

where? memory [Read/write]

flow

- To simplify our study of processor design, we will focus on a subset of the MIPS instructions
  - Memory: `lw` and `sw`
  - Arithmetic: `addu`, `subu`, `and`, `ori`, and `slt`
  - Branch: `beq` and `j`

$f(x) = \{0, 1\}$

$f_0(x) = \{0, 1\}$

$f_1(x) = \{0, 1\}$

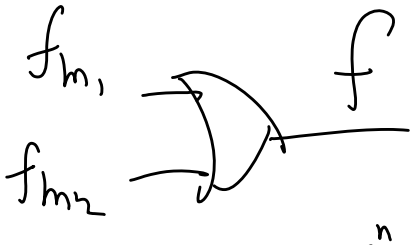
$(f_0(x), f_1(x))$

A	B	f(A,B)
0	0	0
0	1	1
1	0	1
1	1	0

f_{m_1}	
00	0
01	1
10	0
11	0

+

f_{m_2}	
00	0
01	0
10	1
11	0

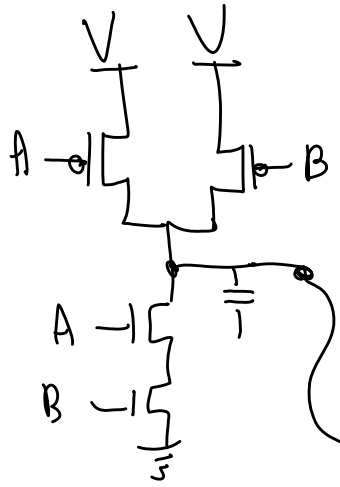


m_0
m_1
m_3
m_3

$$f = \sum_{i \in S} m_i = \sum_{0}^{2^n} a_i m_i$$

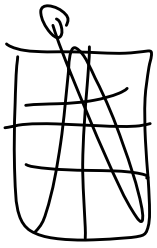
f_{m_2}

0	0	0
0	1	0
1	0	1
1	1	0

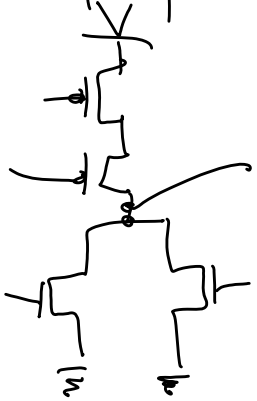


AND	
00	0
01	0
10	0
11	1

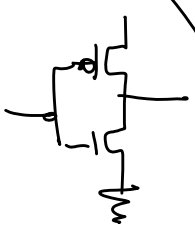
Not	
0	1
1	0

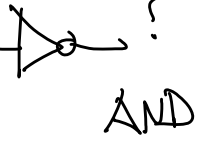


OR	
00	0
01	1
10	1
11	1



A	B	NAND
0	0	V
0	1	V
1	0	V
V	V	0





NOR	
00	V
01	0
10	0
11	0

A	B	m_0
0	0	1
0	1	0
1	0	0
1	1	0

True iff $(\bar{A}, B)$

m_1

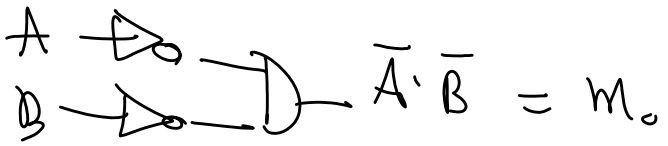
m_2

$\bar{A} \cdot B$

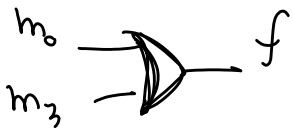
m_3

$A \cdot \bar{B}$

$A \cdot B$



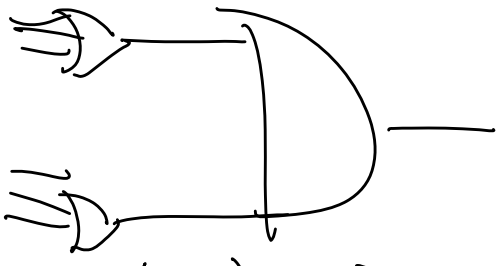
$f = m_0 \oplus m_3$



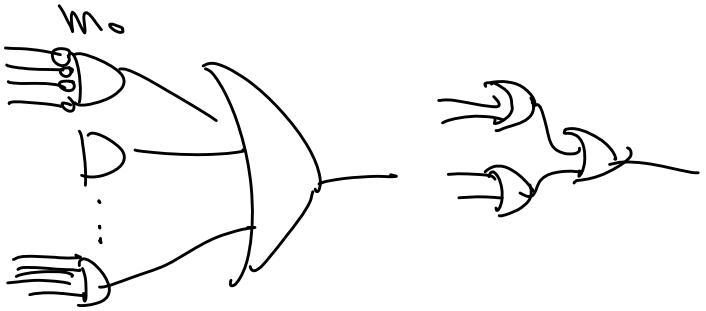
A_{k-1}	A_k ...	A_0	f
0	0	1 1 1	1

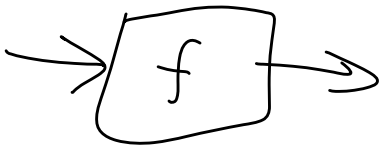
$f = \sum_{i} m_i$

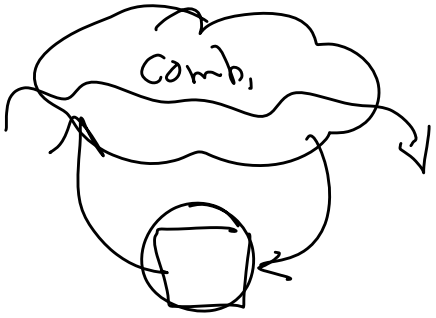
$m_7 = \bar{A}_4 \cdot \bar{A}_3 \cdot A_2 \cdot A_1 \cdot A_0$

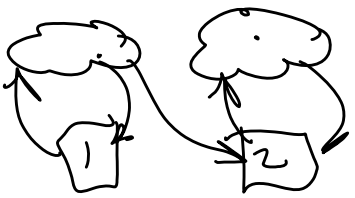


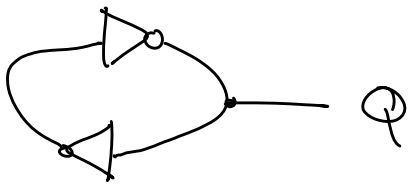
$(A+B)(\bar{A}+B)\ldots$

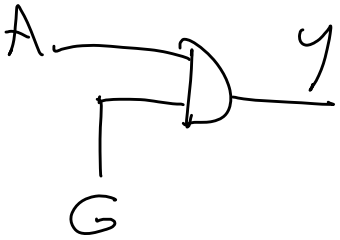


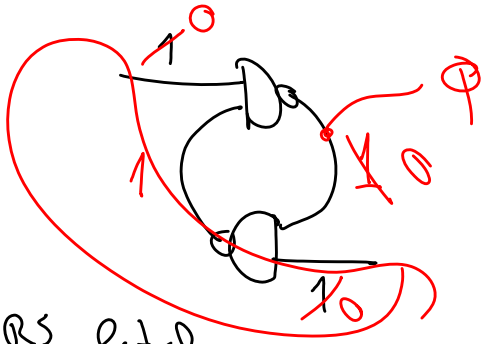


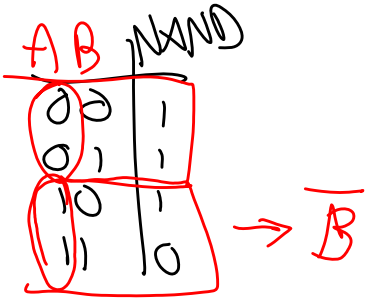


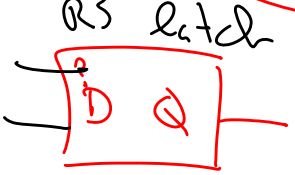


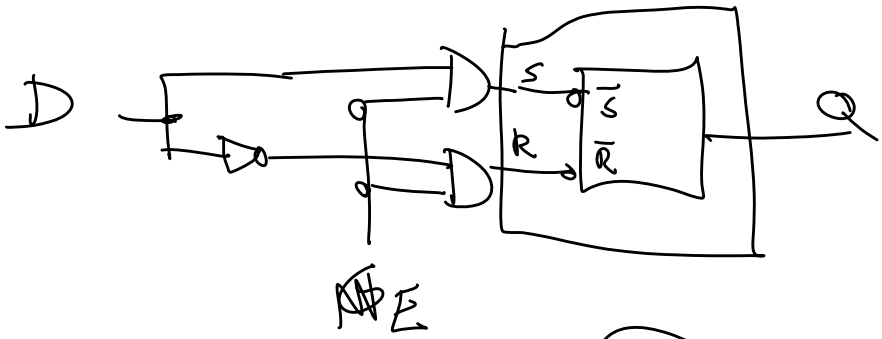


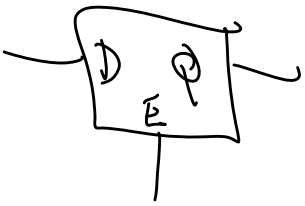


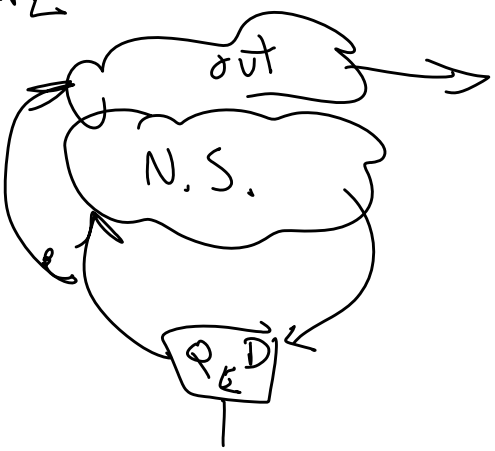




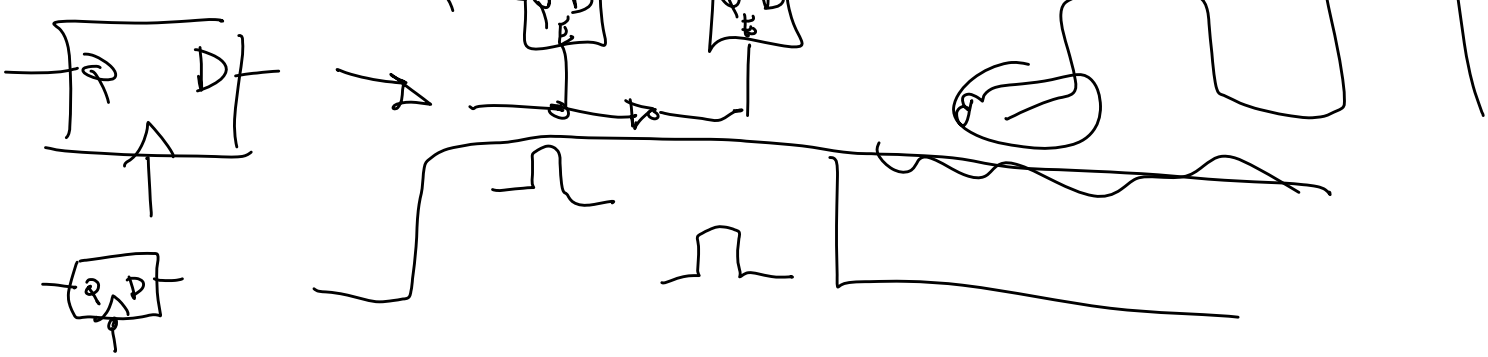


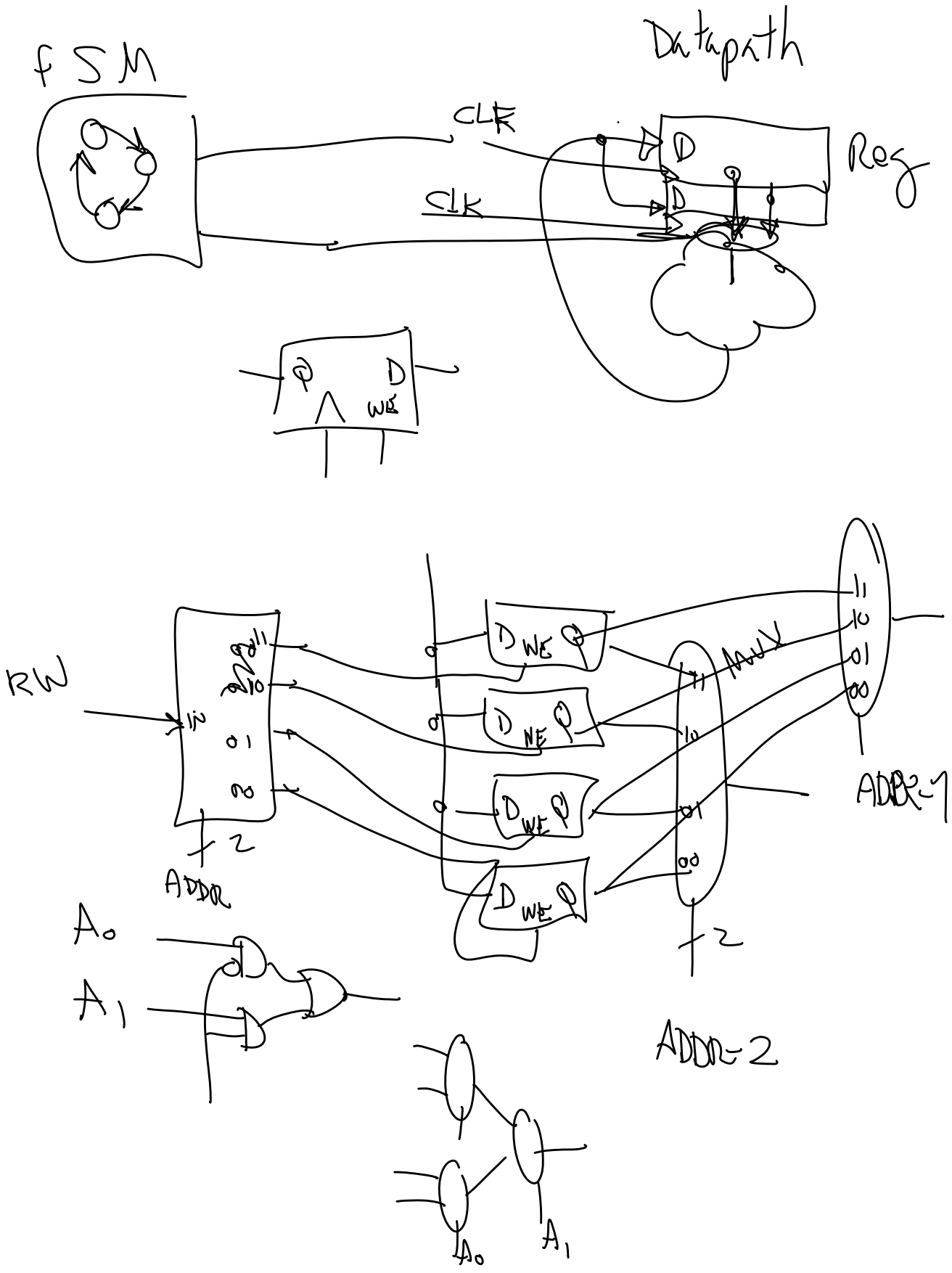






pos. triggered
edge ff




FSM
Datapath
CLK
CLK
D
D
Reg
Q
D
WE
RW
ADDR
2
ADDR-1
ADDR-2
A0
A1
A0
A1

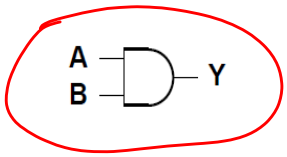
- Steps that occur on each instruction:
  - Fetch instruction from memory - address is specified by PC
  - Read one or two registers
  - Do add/sub/etc. using ALU (see Appendix C.5)
  - Fetch a value from memory
  - Store results to register-file/memory

never for same instruction?

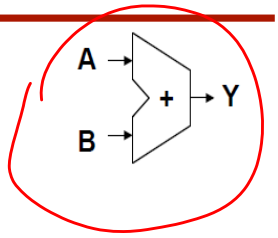
- Needed state for single cycle machine:
  - Instruction pointer (PC), 32 registers, memory

Building Blocks (combinatorial)

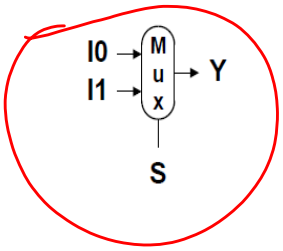
- AND-gate
  - Y = A & B



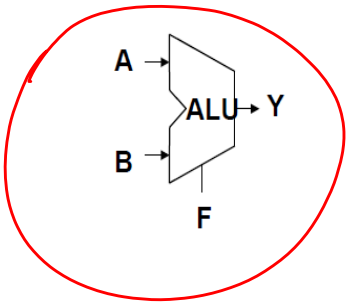
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

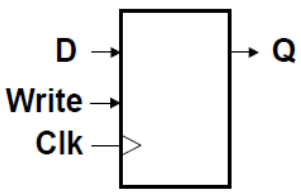


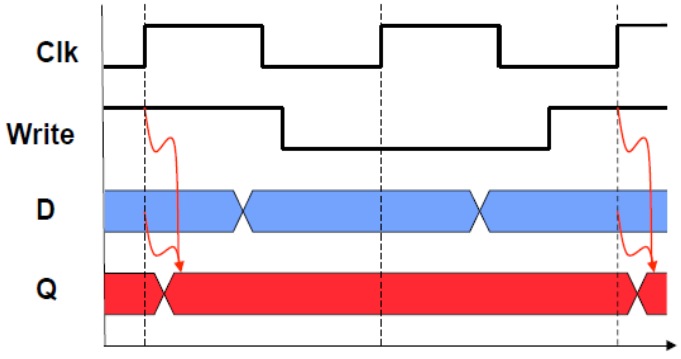
- Arithmetic/Logic Unit
  - Y = F(A, B)



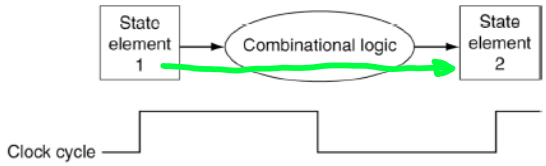
(sequential)

- Flip-flops with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

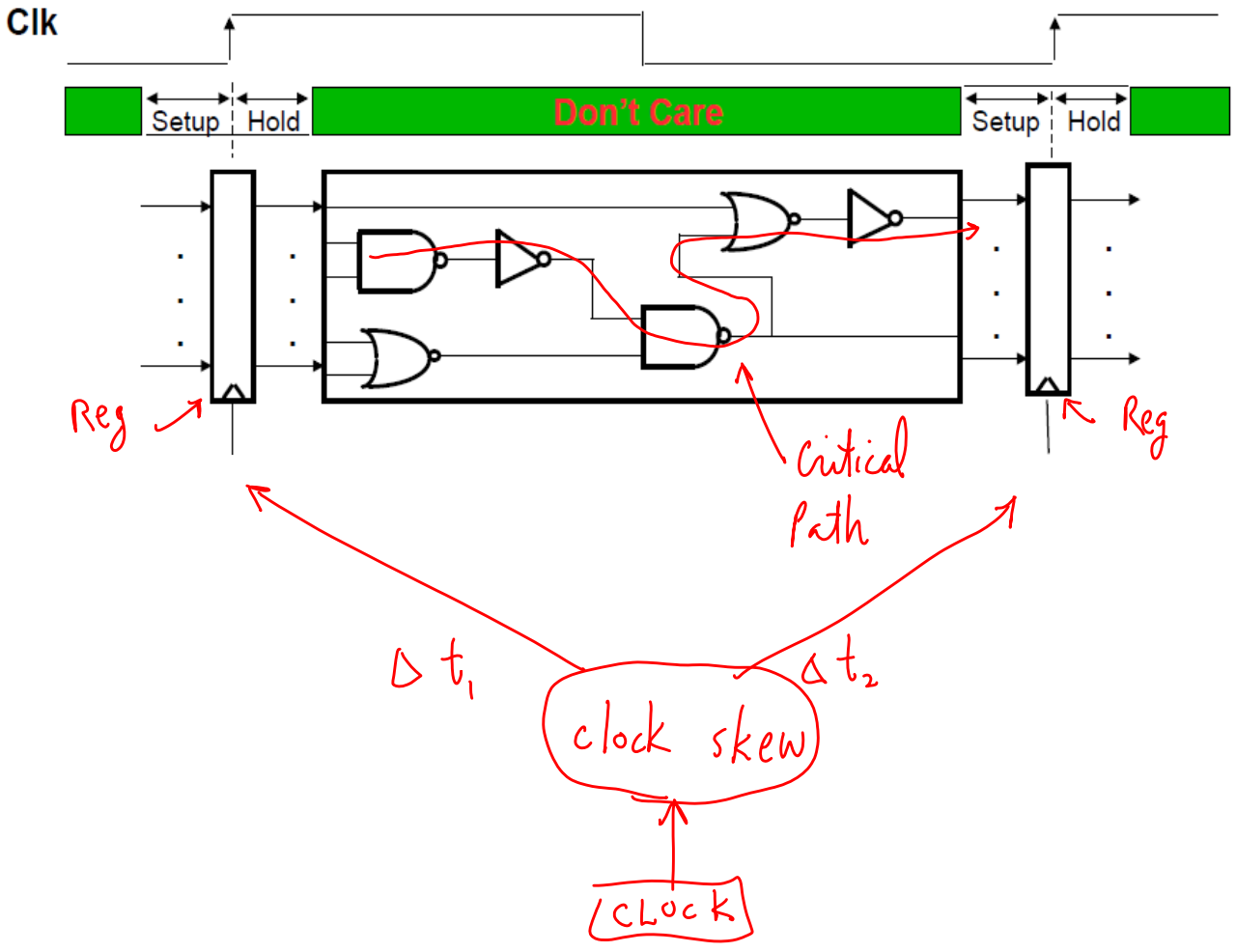




- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period



- Flops work great as long as input is stable when clock rises
  - Called setup and hold windows
  - Clock skew can cause some nasty problems
    - Hold time violations (we won't worry about this in this class)
- Cycle Time = Longest Prop Delay + Setup + Clock Skew



- Interface to Memory can be:
  - Combinational (asynchronous)
  - Clocked (synchronous)
- Combinational memory: ASYNCH
  - Read data is valid some delay after address lines settle
    - There is no clock.
  - Writes are tricky: must supply a write pulse in the middle of your address and data valid times

when to send?

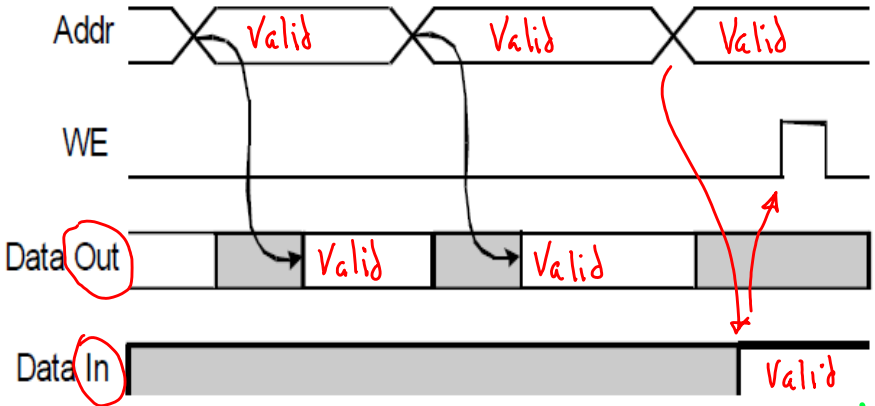
OR

- Clocked memory (most common): SYNCH
  - Memory looks like a standard synchronous device.
  - Address and control signals are sampled on rising edge of clock, and data is valid some number of cycles later

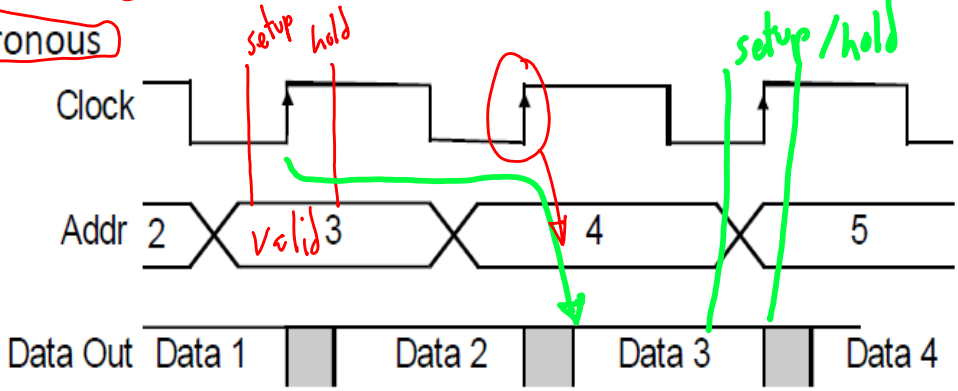
Use WE

latency of mem response

- Combinational/Asynchronous:



- Synchronous



Pipelined access

A

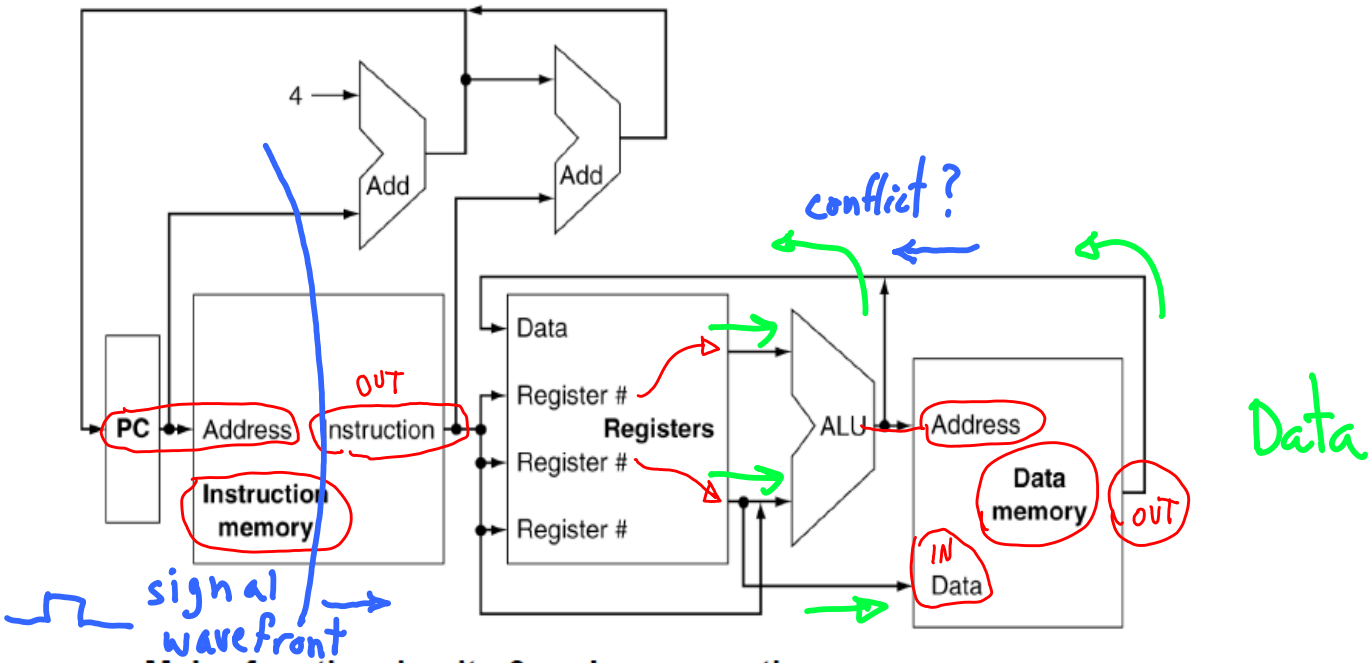
Set addr — A2

read data — A1

Memories In This Design

- They will be combinational ASYNCH w/ fixed delay
  - Otherwise we can't complete an instruction in one cycle!
- Interface is simple:
  - Inputs:
    - Address
    - DataIn
    - WriteEn (WriteEn must be a pulse)
  - Outputs:
    - DataOut
- Register file:
  - It has three address, two for reads, and one for write
  - It is called a 3-port, since it can perform 3 accesses per cycle

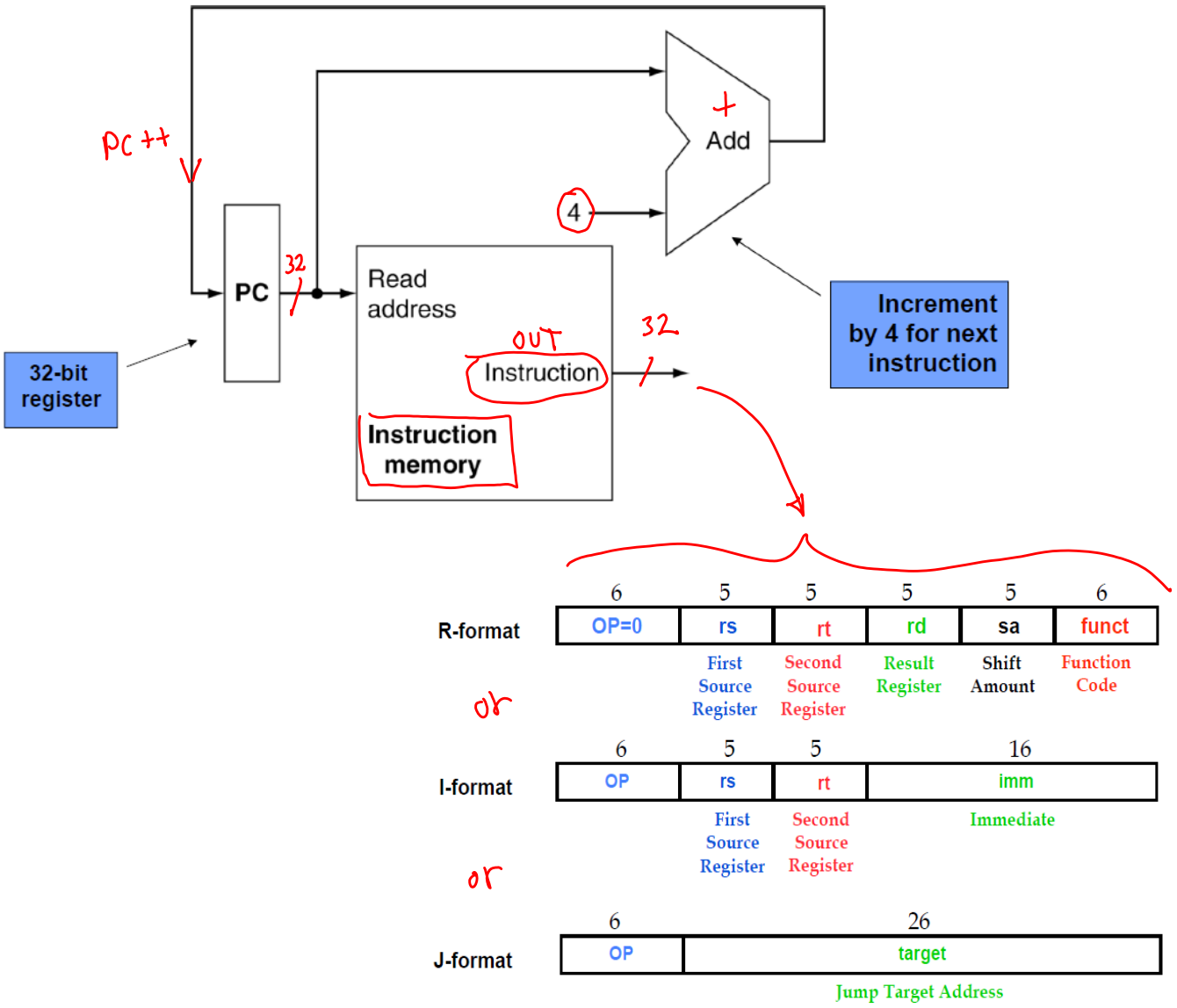
Starting Datapath for Our Processor



- Major functional units & major connections
- Missing: multiplexers (cannot just join wires) & control

INSTR. FLOW

Instruction Fetch



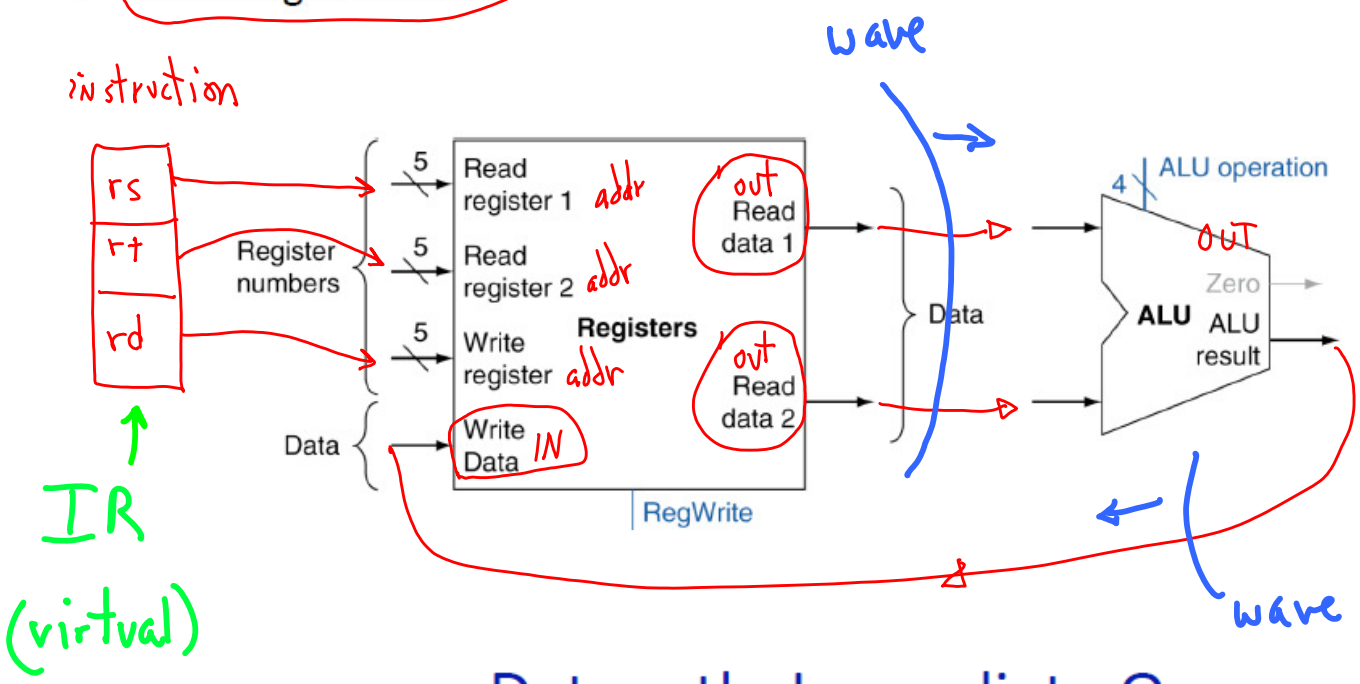
- Instructions are fixed length
  - Don't need to decode first instruction to find next one
  - Always add 4 bytes to instruction pointer (PC)

- Register specifiers are always in the same place
  - Destination moves around some, but
  - Source registers are always in the same place
    - Or you don't need that register
  - Can fetch the registers BEFORE you decode instruction
    - Feed bits directly from the instruction memory

rT, rS

Datapath: R-Format Instructions

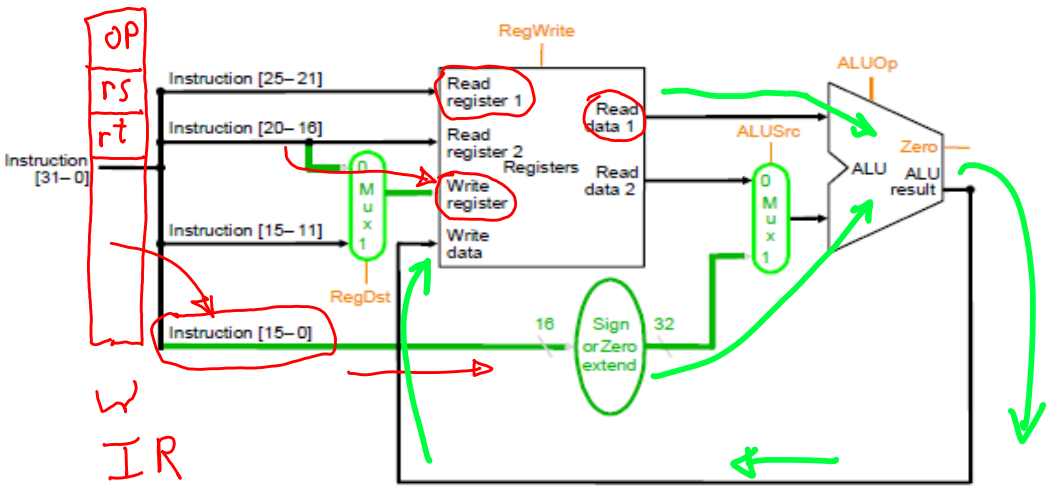
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Datapath: Immediate Ops

I-format

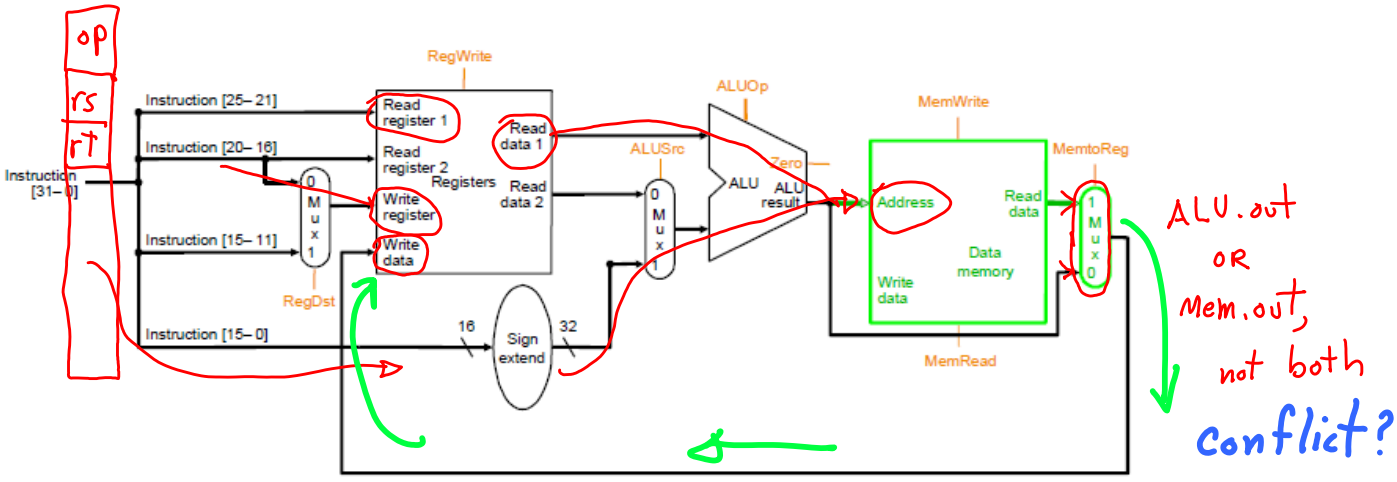
- Extend datapath to support immediate operations
- Write register is rt or rd based on instruction
- Read data 2 is ignored for immediates
- Immediates can be sign or zero extended
- ALUsrc and ALU operation set based on instruction



Datapath: Load

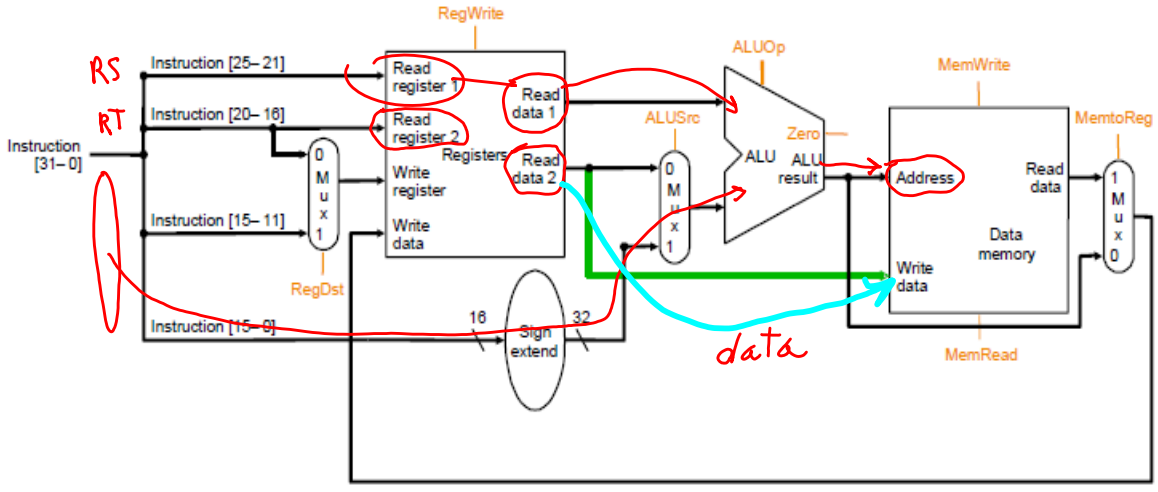
I-format lw

- Extend datapath to support other immediate operations
- Extender handles either sign or zero extension
- MUX selects between ALU result and Memory output

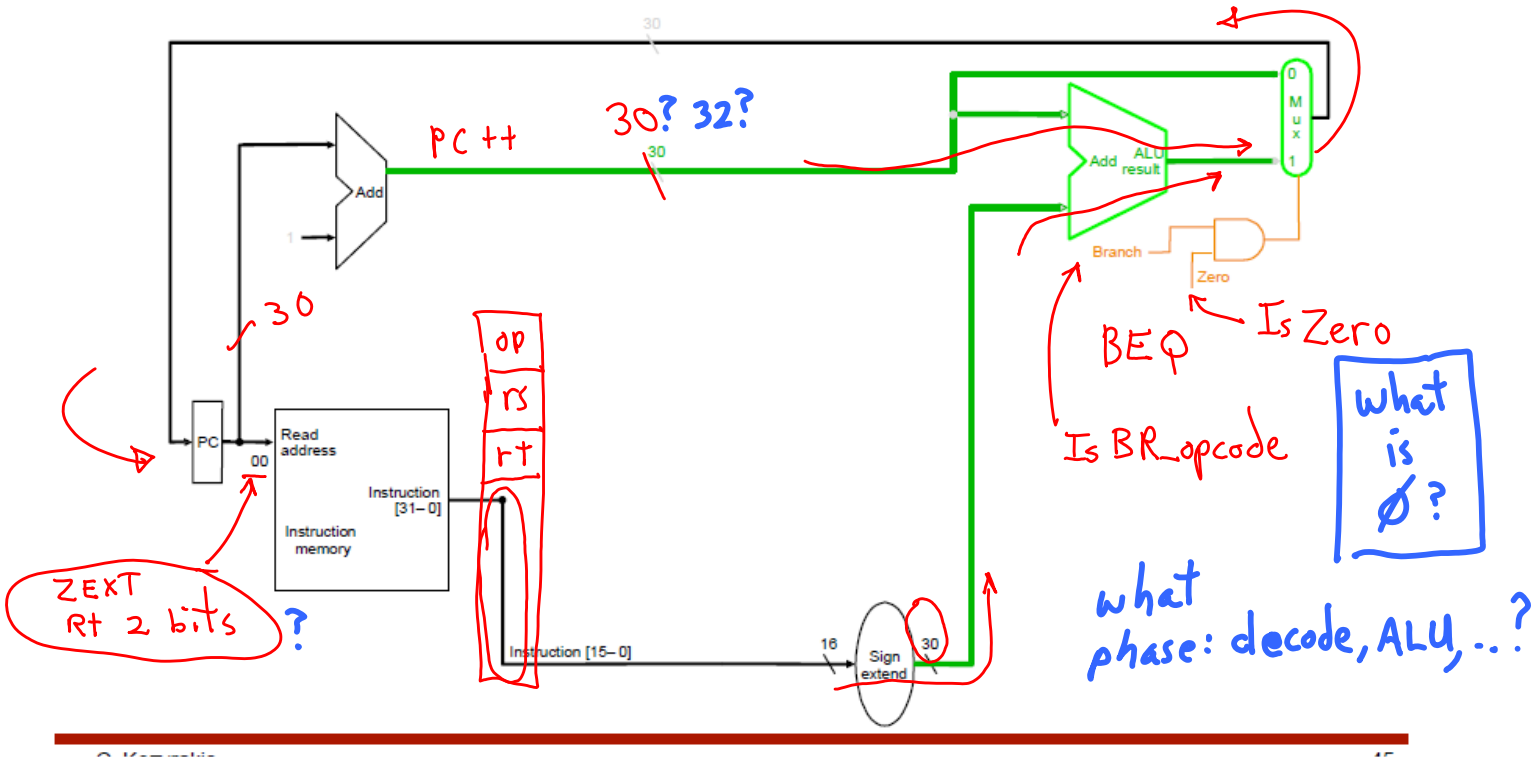


- Read Register 2 is passed on to Memory
- Memory address calculated just as in lw case

SW

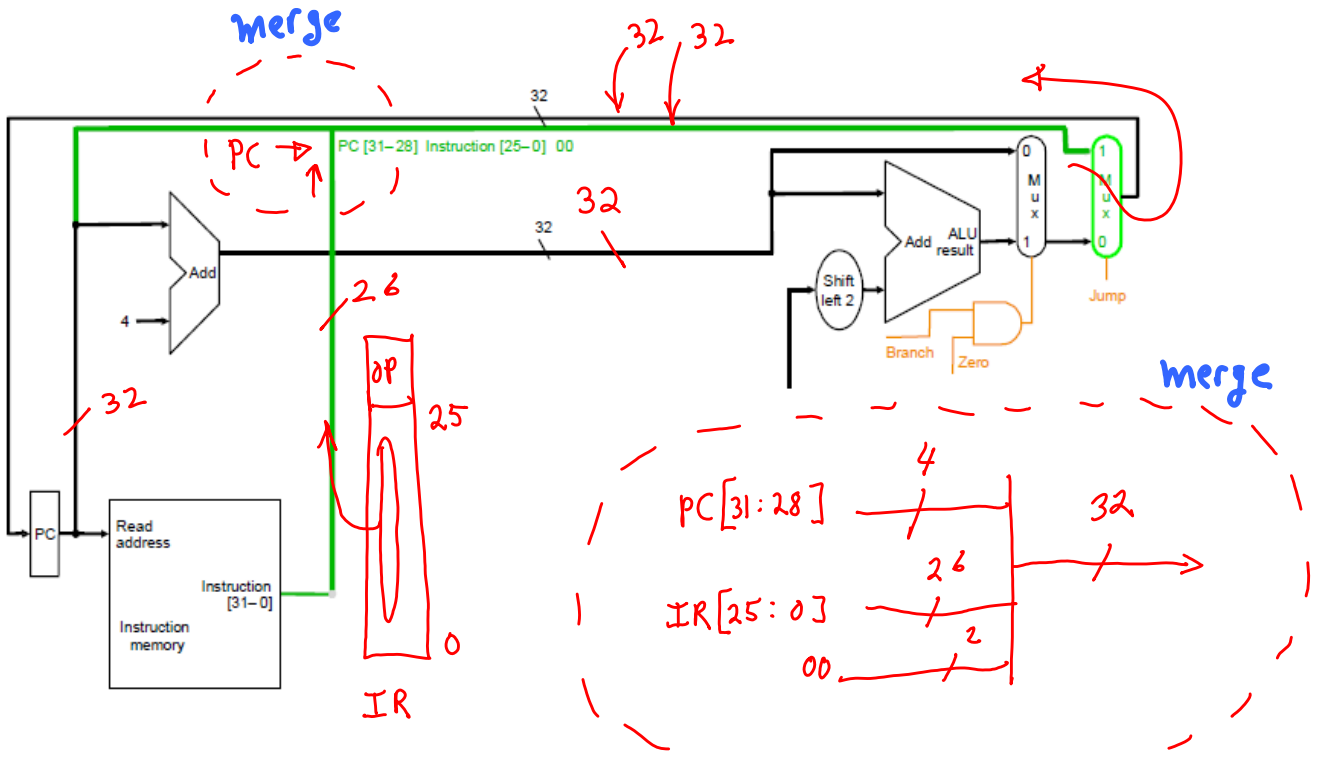


Datapath: IF + Branch

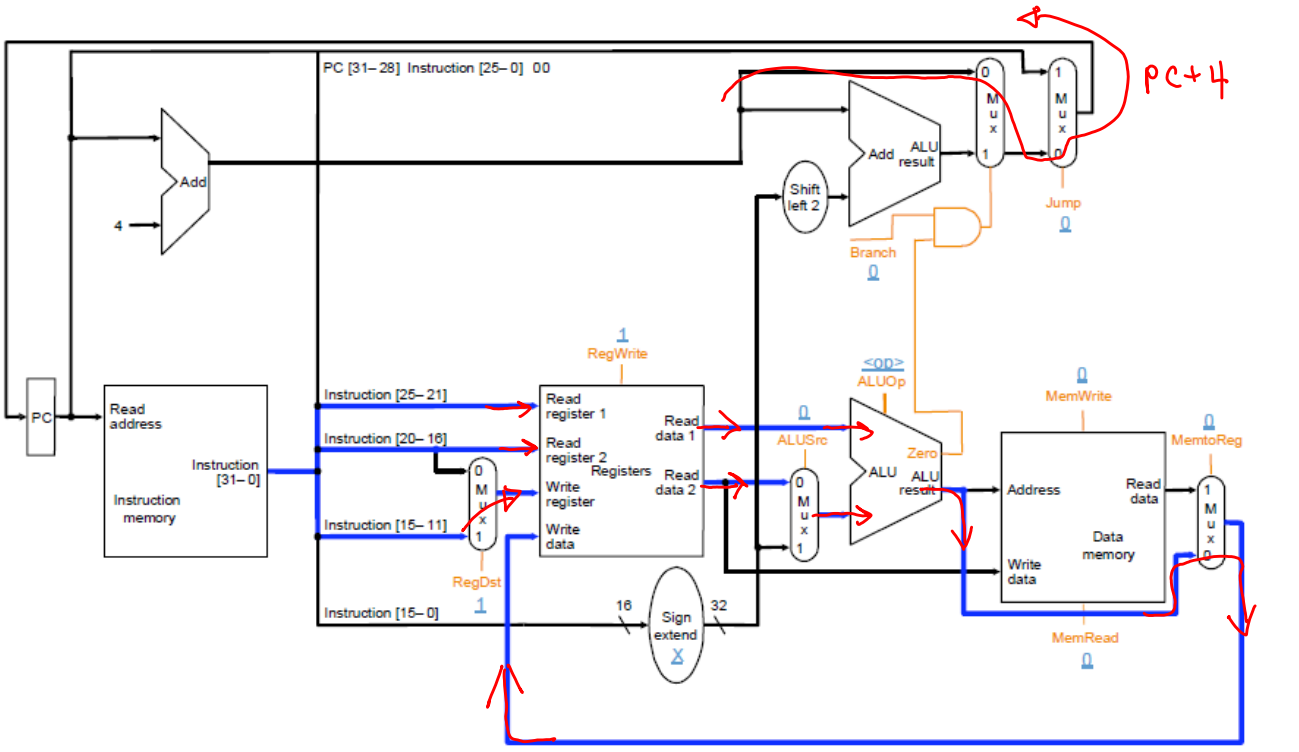


Datapath: IFU + Jump

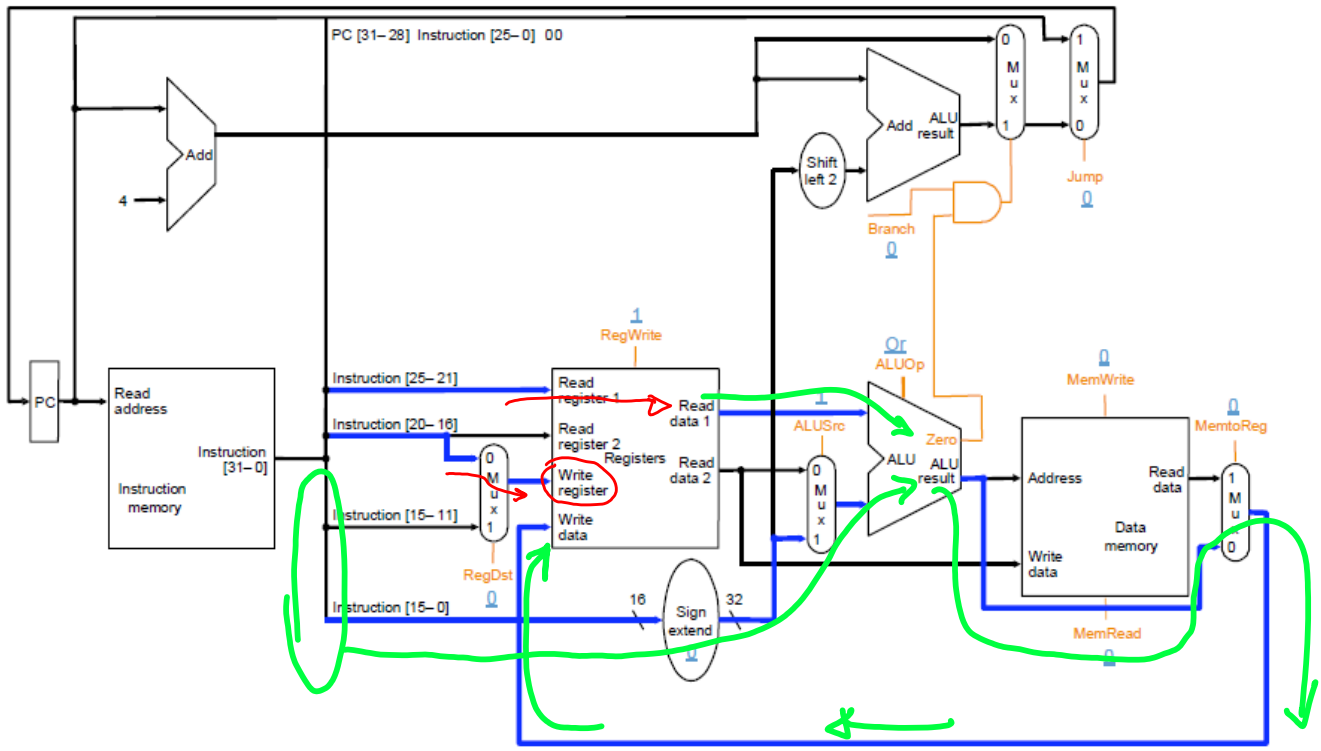
- MUX selects pseudodirect jump target



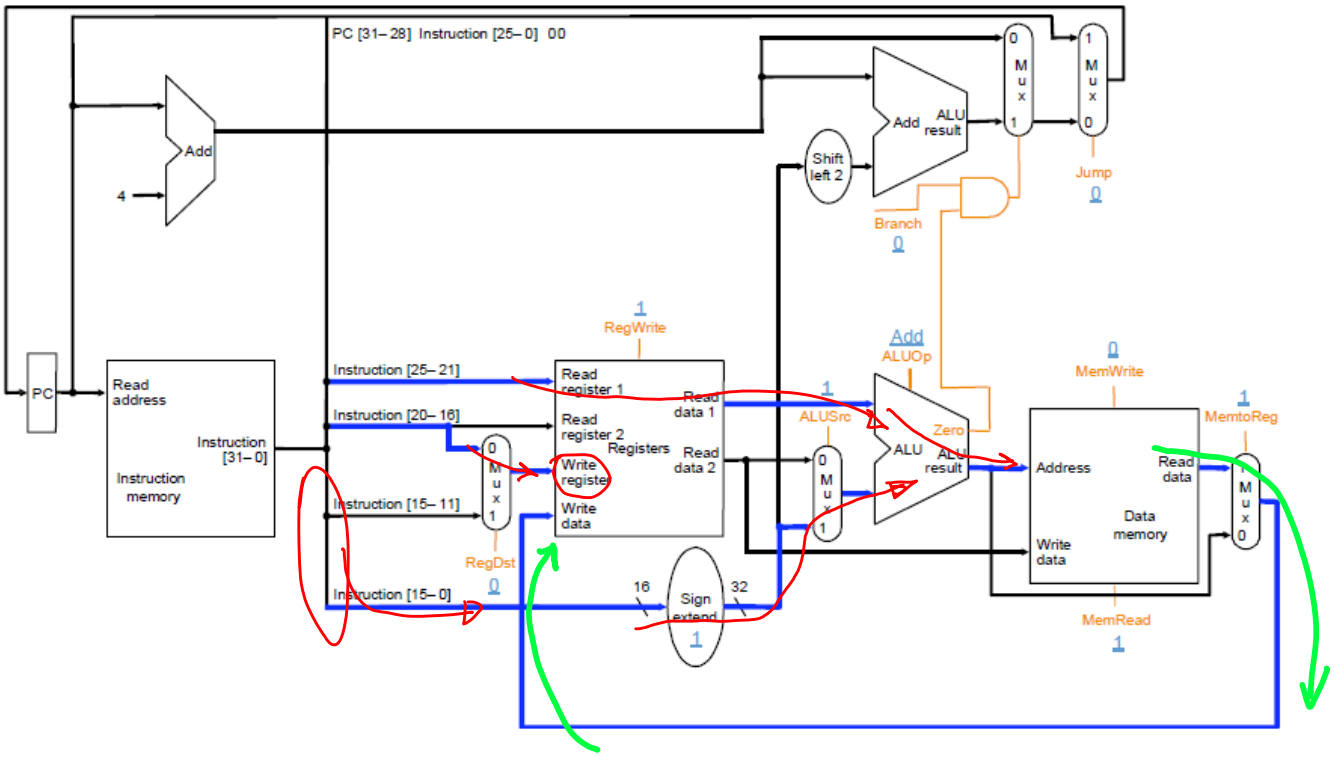
Control for Arithmetic



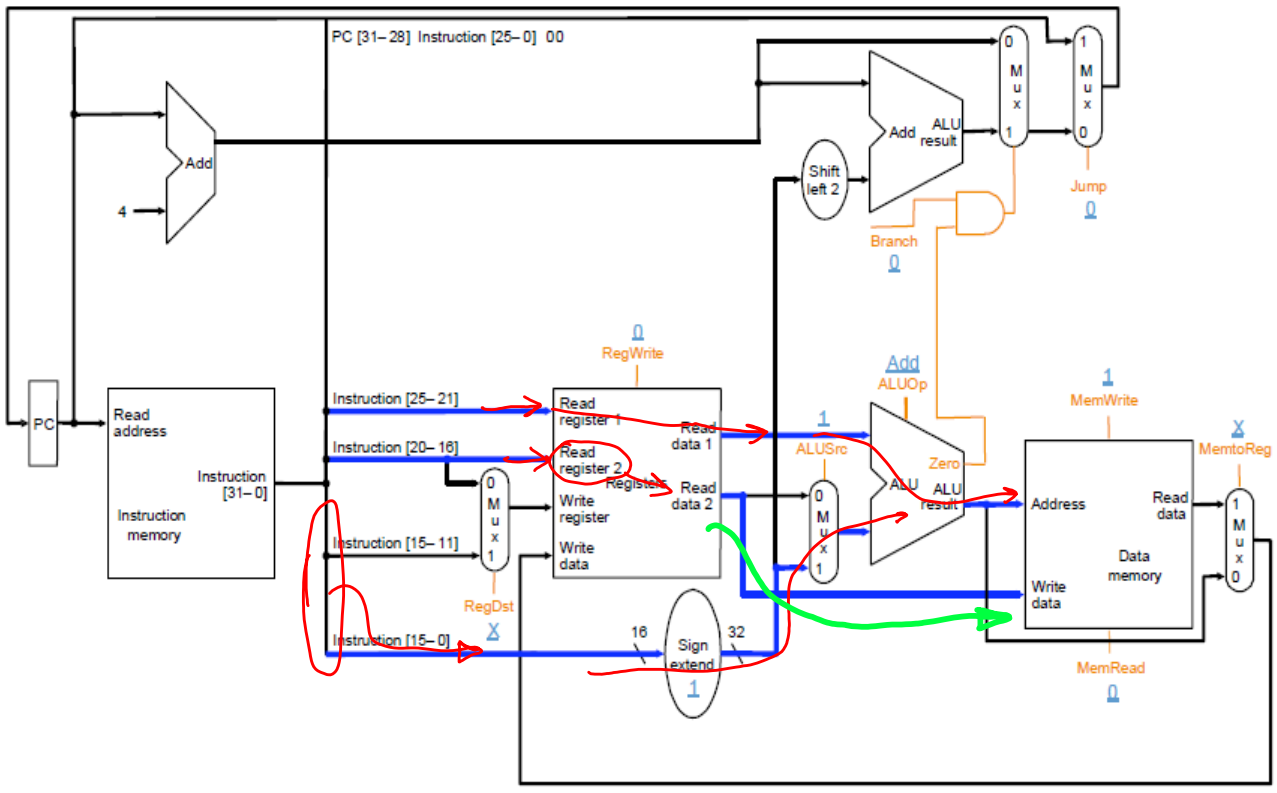
Arithmetic Immediate (`ori`)



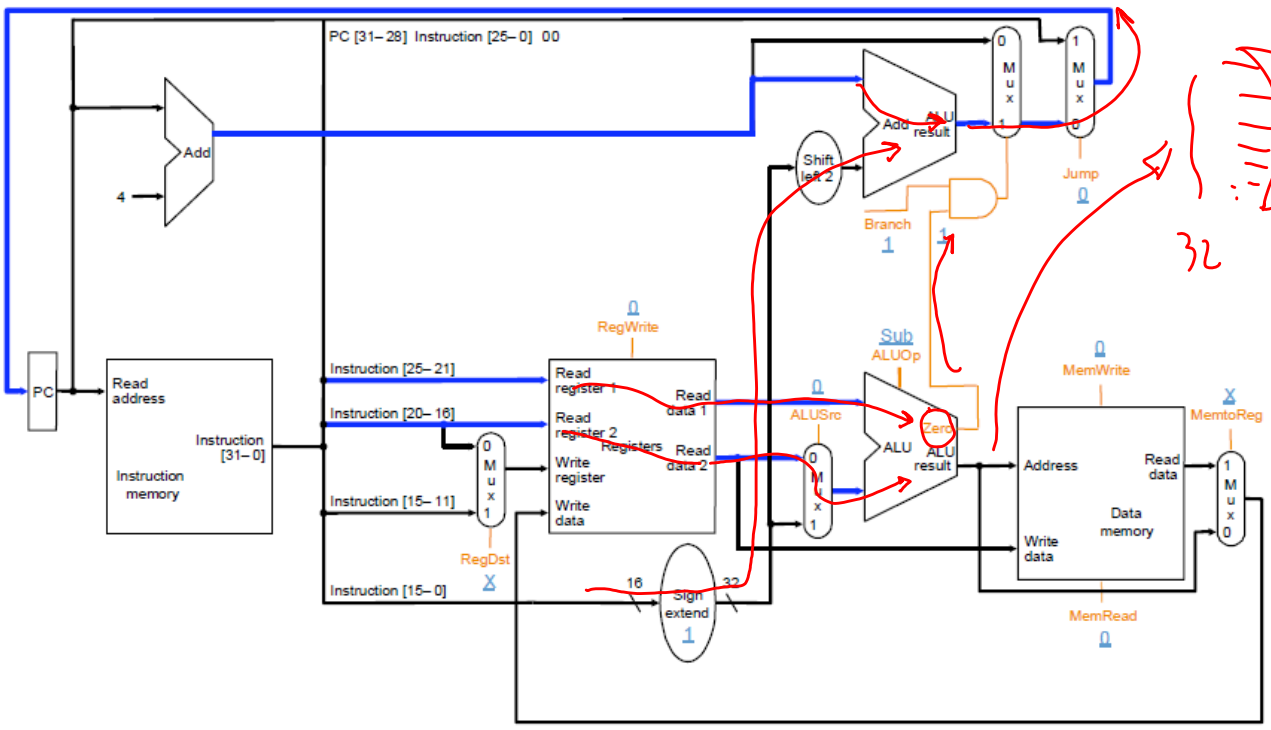
Control for Load



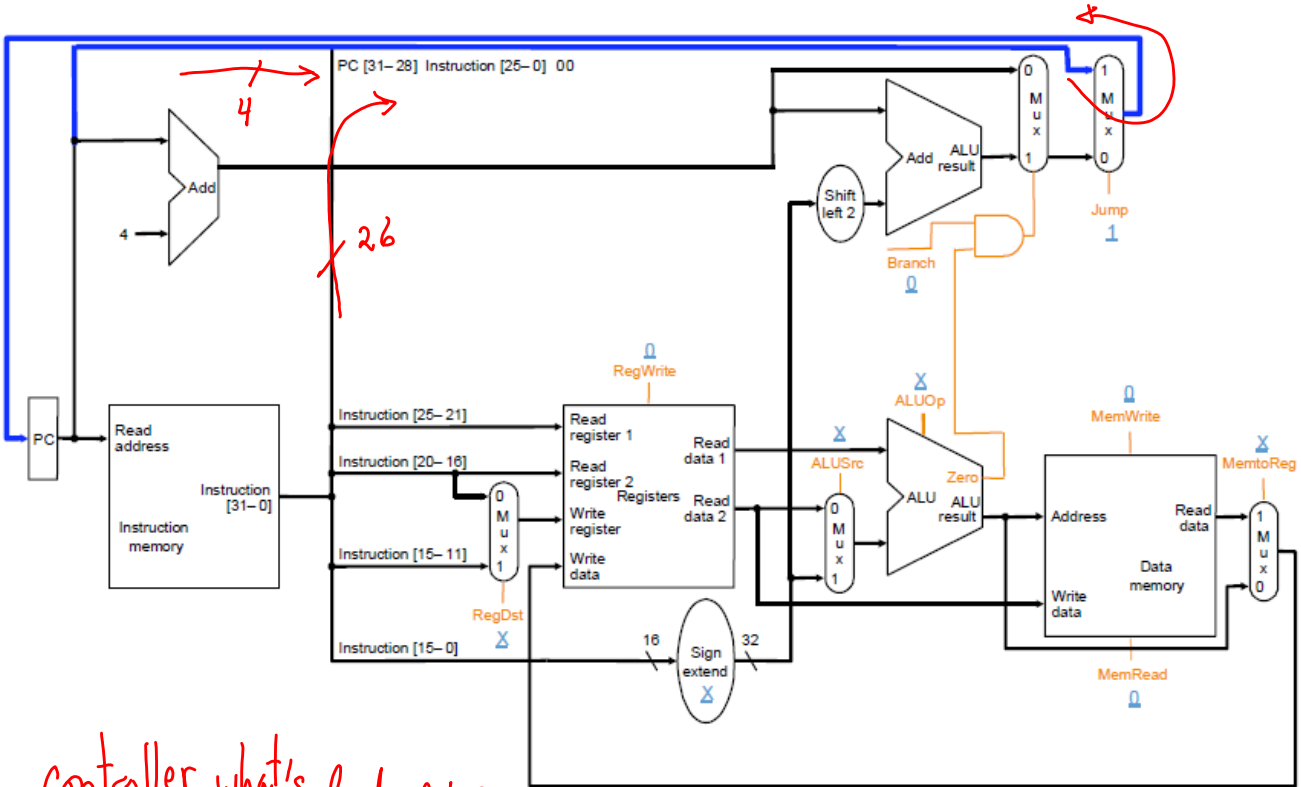
Control for Store



Control for Branch (beq)

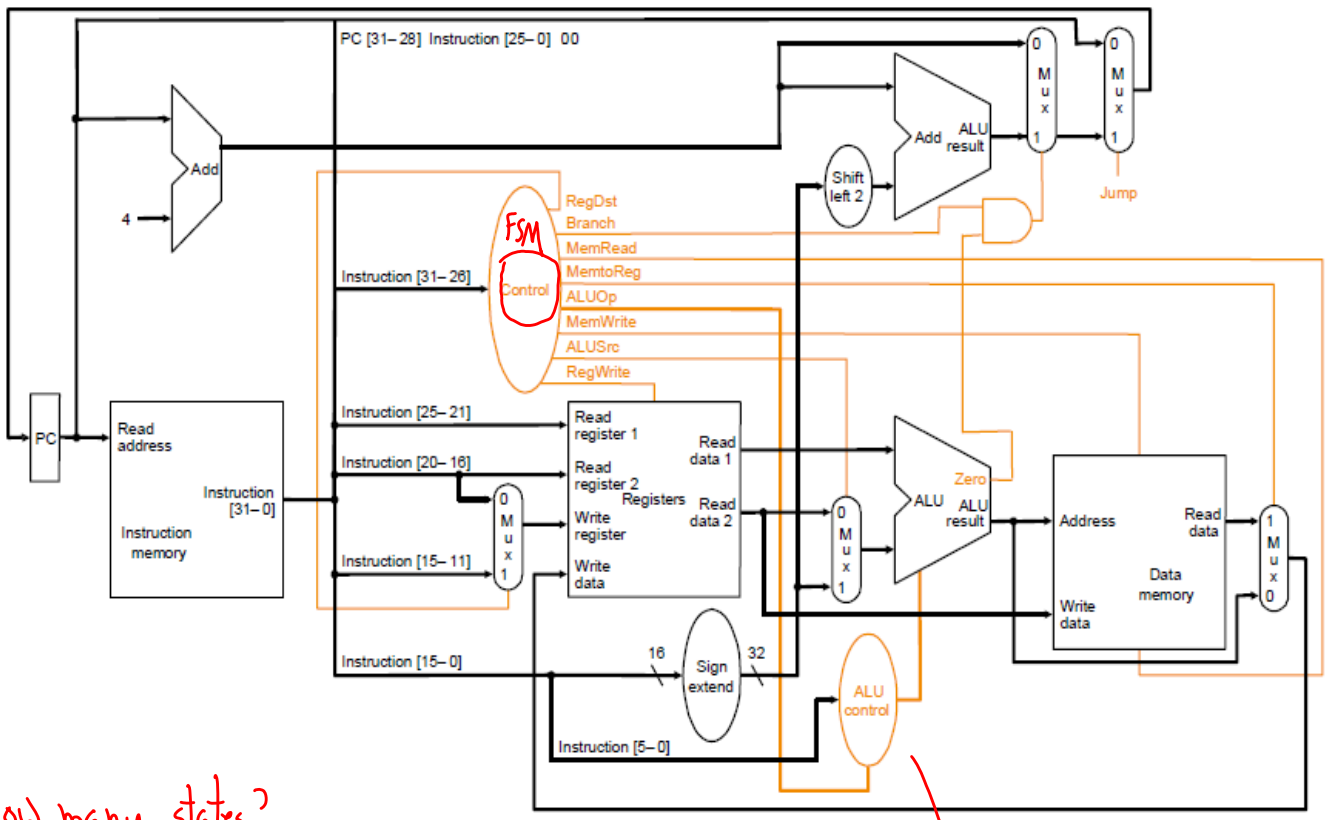


Control for Jump (j)



Q. controller, what's look like?

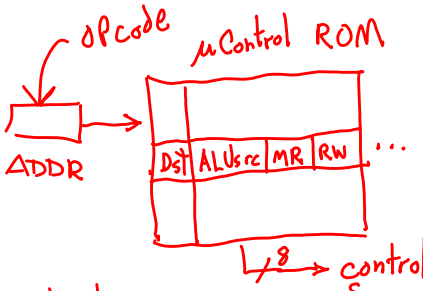
Putting It All Together: Your first processor



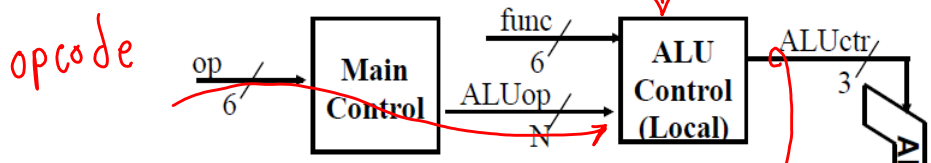
Q. How many states?

Q. How is next-state implemented for controller?

- Since only the ALU needs the func field
  - Pass it to the ALU unit, and have a local decoder there



external-to-ROM portion of controller



control signals

ROM

cols

← ROM rows →

func	10 0000	10 0010	Not Important				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Sub	Or	Add	Add	Sub	xxx

ALU complications:
1. define op, r-format
2. define op, i-format
3. define op, lw, sw

efficiency
1. fewer bits to ALU mux
2. fast response

Single Cycle Processor

- Advantages
  - Single cycle per instruction makes logic and clock simple
- Disadvantages
  - Inefficient utilization of memory and functional units since different instructions take different lengths of time
    - ALU only computes values a small amount of the time
  - Cycle time is the worst case path → long cycle times = lw
    - Load instruction
  - Best possible CPI is 1