MIPS, 1st implementation

Instruction-Level Parallelism

--- A single job (program/thread)

--- Execute multiple instructions simultaneously

--- As many as we can

--- w/ minimal added hardware

Complex System Design

--- Modularity

--- simple functions w/ guaranteed behavior

--- simple interfaces w/ good abstractions

--- Heirarchical composition

--- low-level complexity is hidden

===> 1 billion components that work!

--- Reusable building blocks

--- Generally applicable

--- Customizable

--- Isolation allows evolution

Implement MIPS / LC3*

- 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) ← controller: Decode, FSM
- Find a decomposition that is simple, and efficient
  - Some are obvious, others can be more subtle

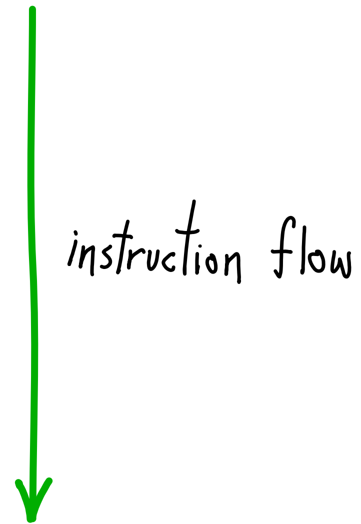
{ Remove annoying instructions

- We will start simple
  - Add stuff to improve performance

multiple instructions executing simultaneously

Phases

- 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 **or memory**
- Finally we need to:
  5. Calculate the next instruction address

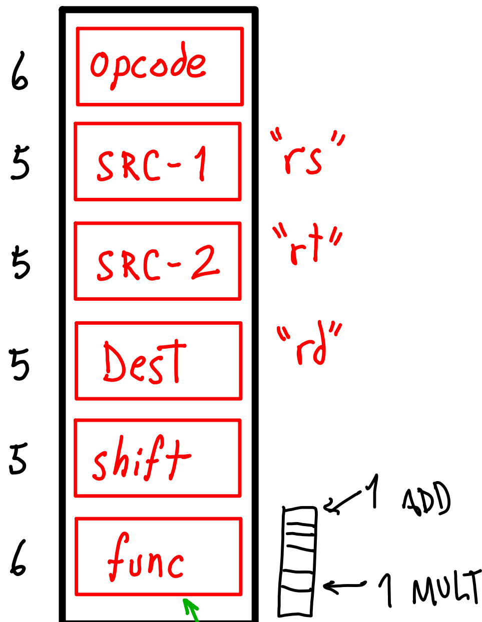


MIPS instructions

not needed?
Use BR via Reg?

R-format

ADD Dest, SRC-1, SRC-2

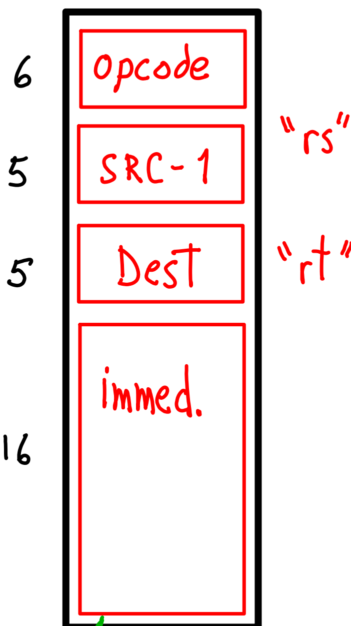


I-format

ADD Dest, SRC-1, immed.

BR SRC-1, SRC-2, offset

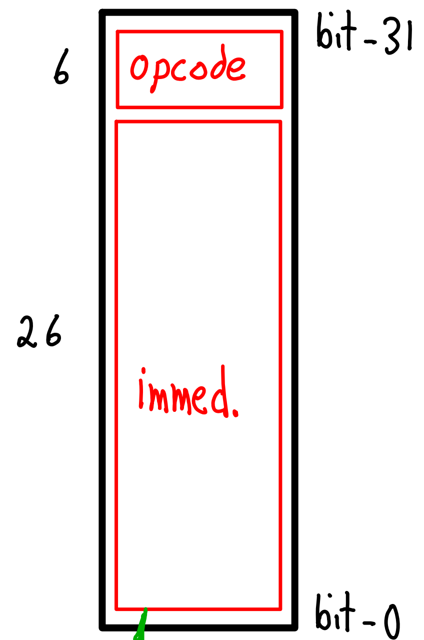
LW Dest, offset(SRC-1)



if address offset: extended w/ low-order 00 → 18 bits

J-format

JMP immed.



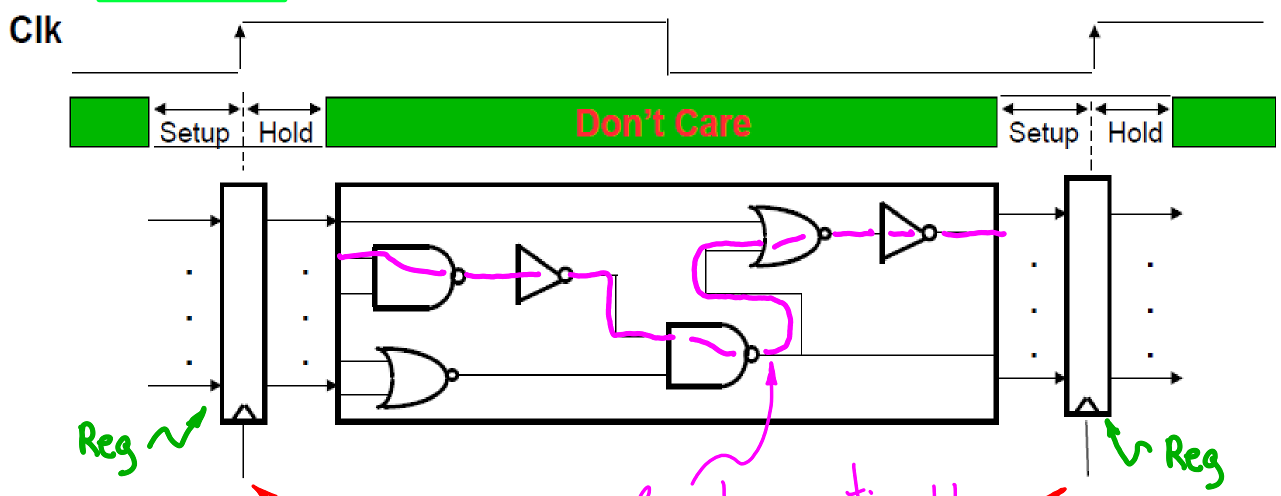
extended w/ low-order 00 → 28 bits

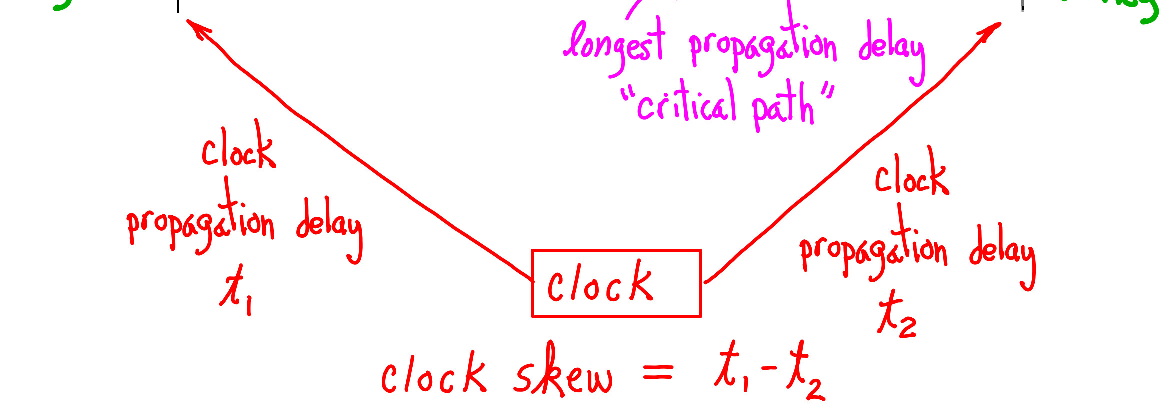
Execution Phase	Uses this HW
--- Fetch Instruction, PC++	--- IMem, PC
--- Read Registers, Decode	--- RegFile, Control decoder
--- Execute ALU operation	--- ALU + PC
Execute BR/JMP operation	--- ALU + PC
--- R/W Data Memory	--- DMEM
--- Write register	--- RegFile

if each phase is independent, could they operate simultaneously?

a structural conflict? 2 instructions using same HW?

- 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



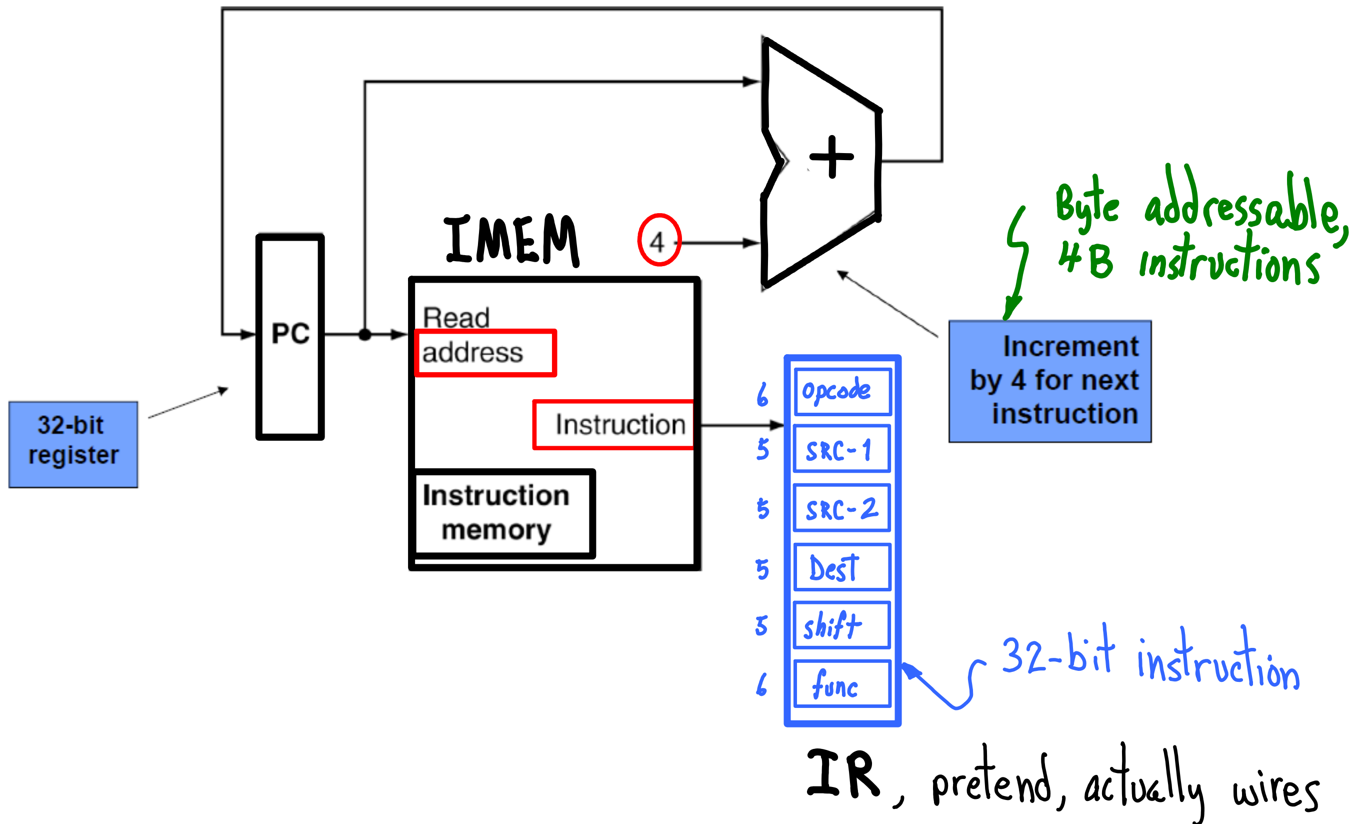


clock skew $= t_1 - t_2$

skews the start of critical path delay for some path.

Total delay = max critical path + max skew + hold + setup → T_{clock}

Instruction Fetch , PC++



think : 16-bit version w/ 2B words (instructions), 2B-addressable
PC ← PC + 1, ISA = { ALU_op, LDR/STR, BR (includes TRAP, JSRR) }
→ simplify

Design ISA for ILP, fast ops

- Instructions are fixed length
  - Don't need to decode first instruction to find next one
  - Always add 4 bytes to instruction pointer (PC) — 1

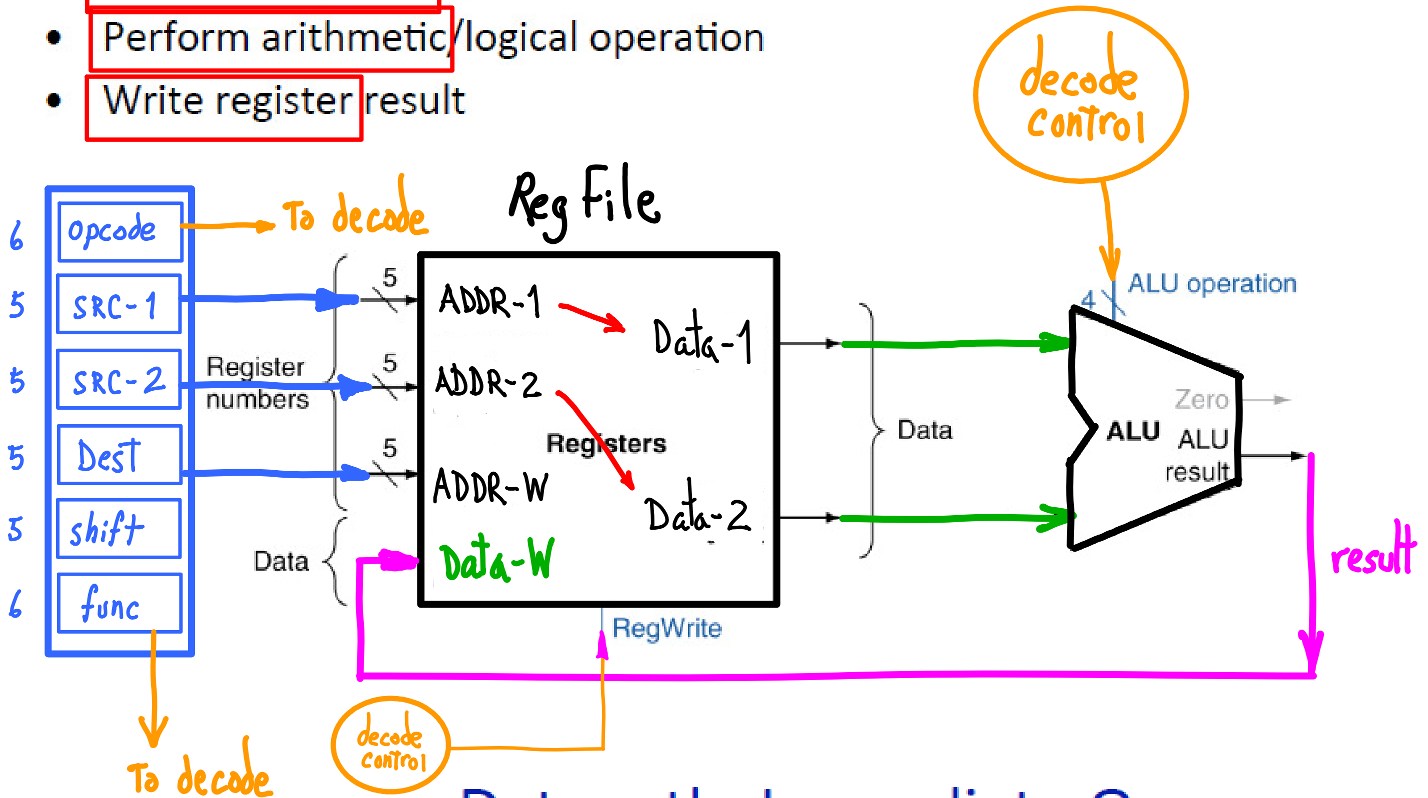
e.g., unlike x86

- 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

Datapath: R-Format Instructions

ADD

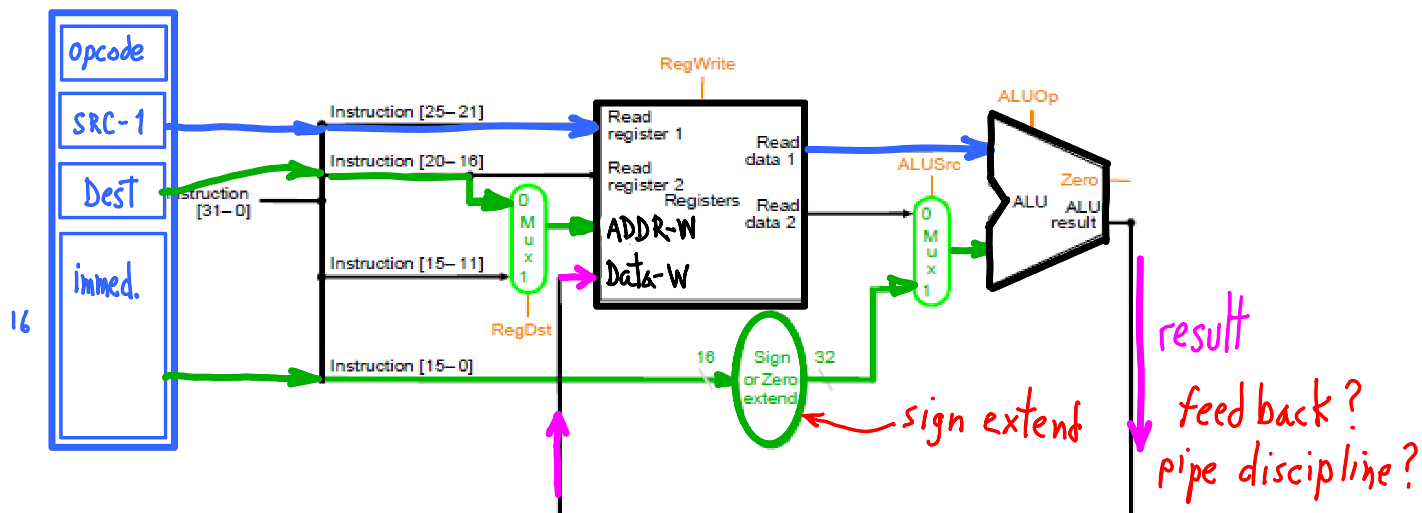
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Datapath: Immediate Ops

I-format ADD

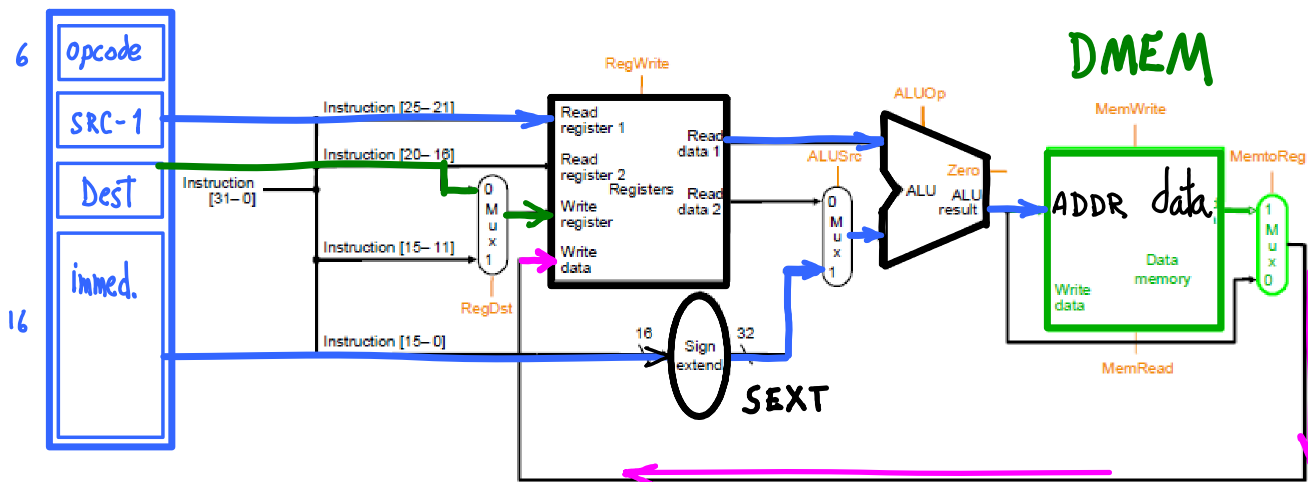
- Extend datapath to support immediate operations
- Write register is rt or rd based on instruction
- Read data 2 is ignored for immediates
- Immediates can be sign or zero extended
- ALUsrc and ALU operation set based on instruction



Datapath: Load

I-format, LW

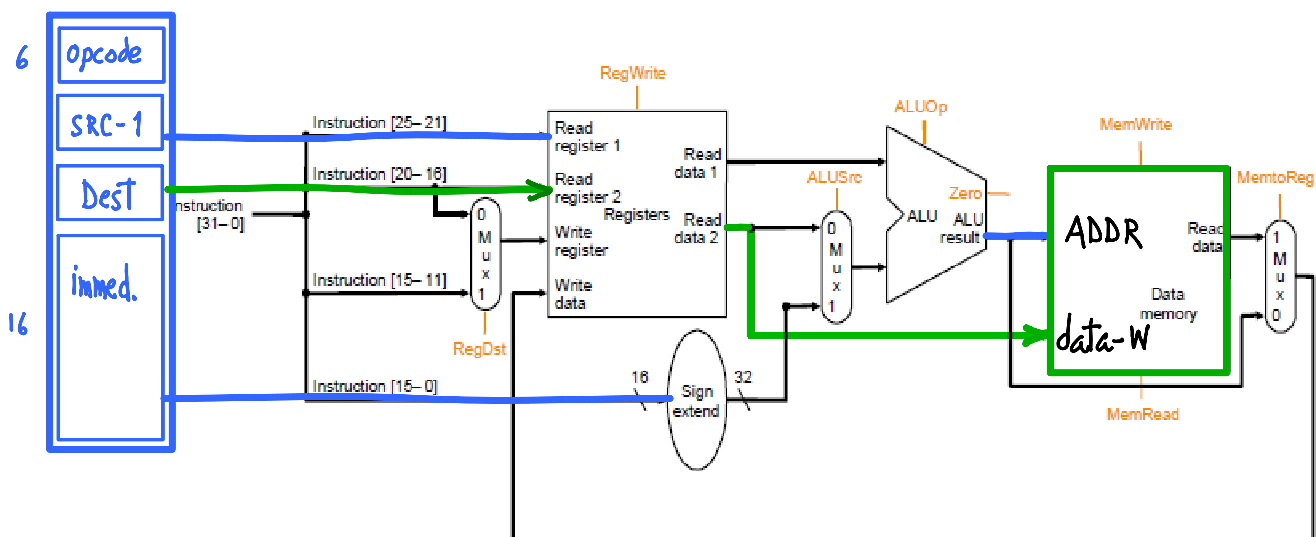
- Extend datapath to support other immediate operations *address calc.*
- Extender handles either sign or zero extension
- MUX selects between ALU result and Memory output



I-format, SW

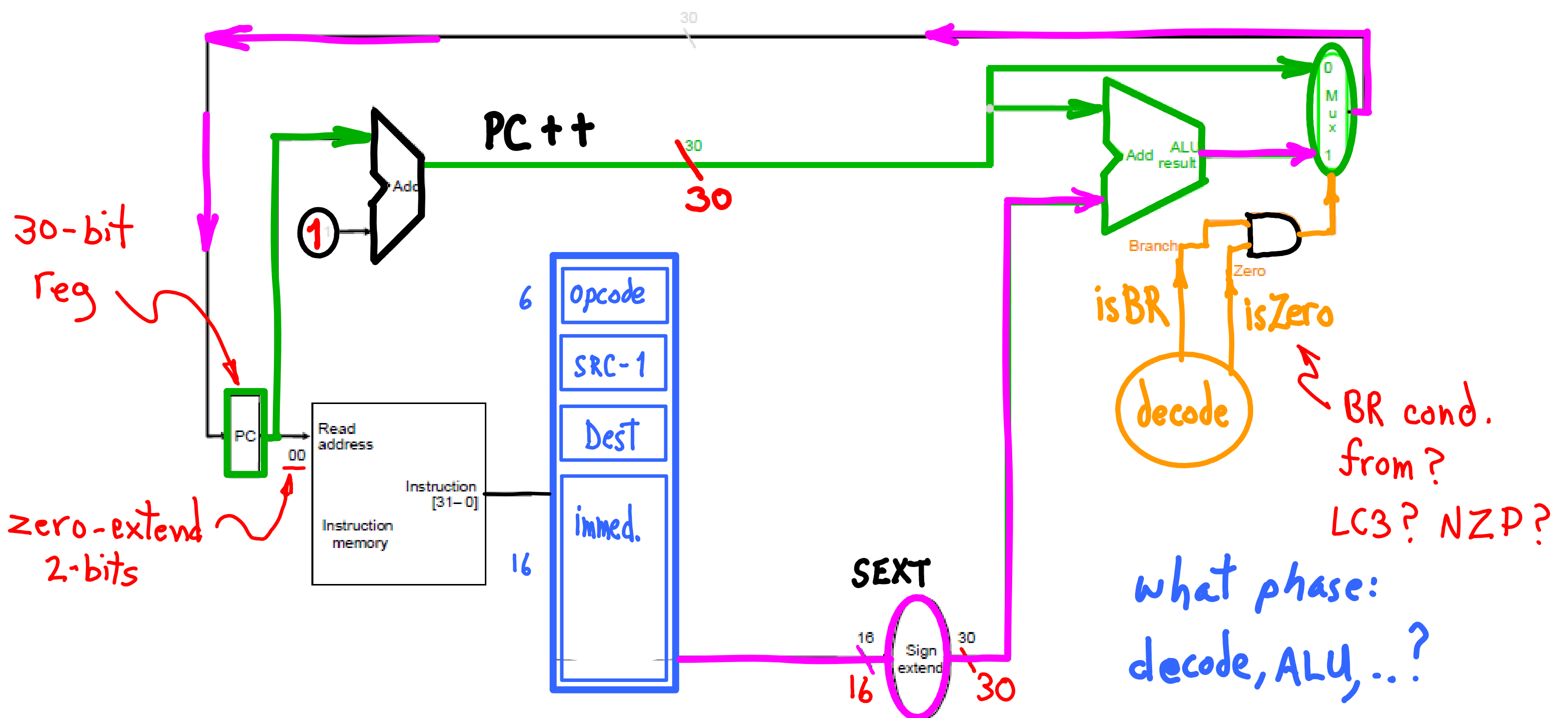
- Read Register 2 is passed on to Memory
- Memory address calculated just as in lw case

Could we use LDR/STR? How?



I-format, Datapath: IF + Branch

(We can squeeze more range out of offsets: ignore low 2-bits for instruction addresses.)



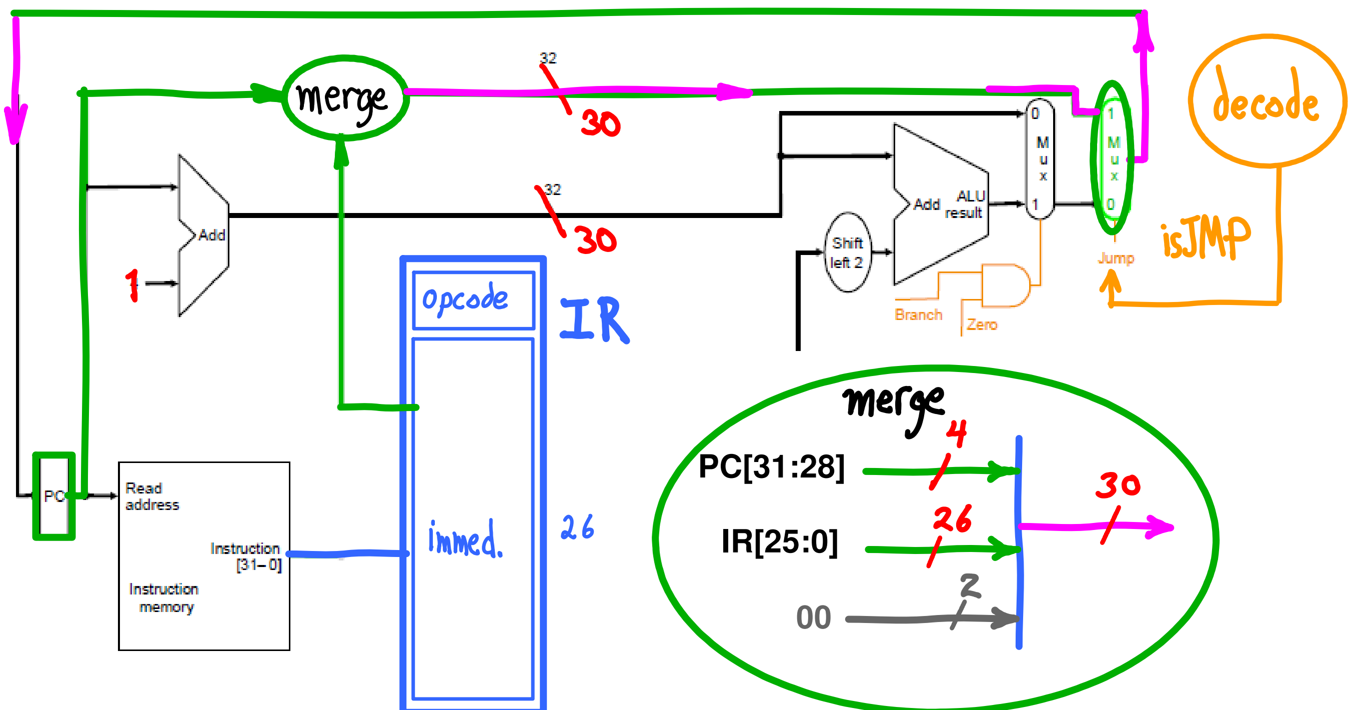
Datapath: IFU + Jump

J-format, JMP

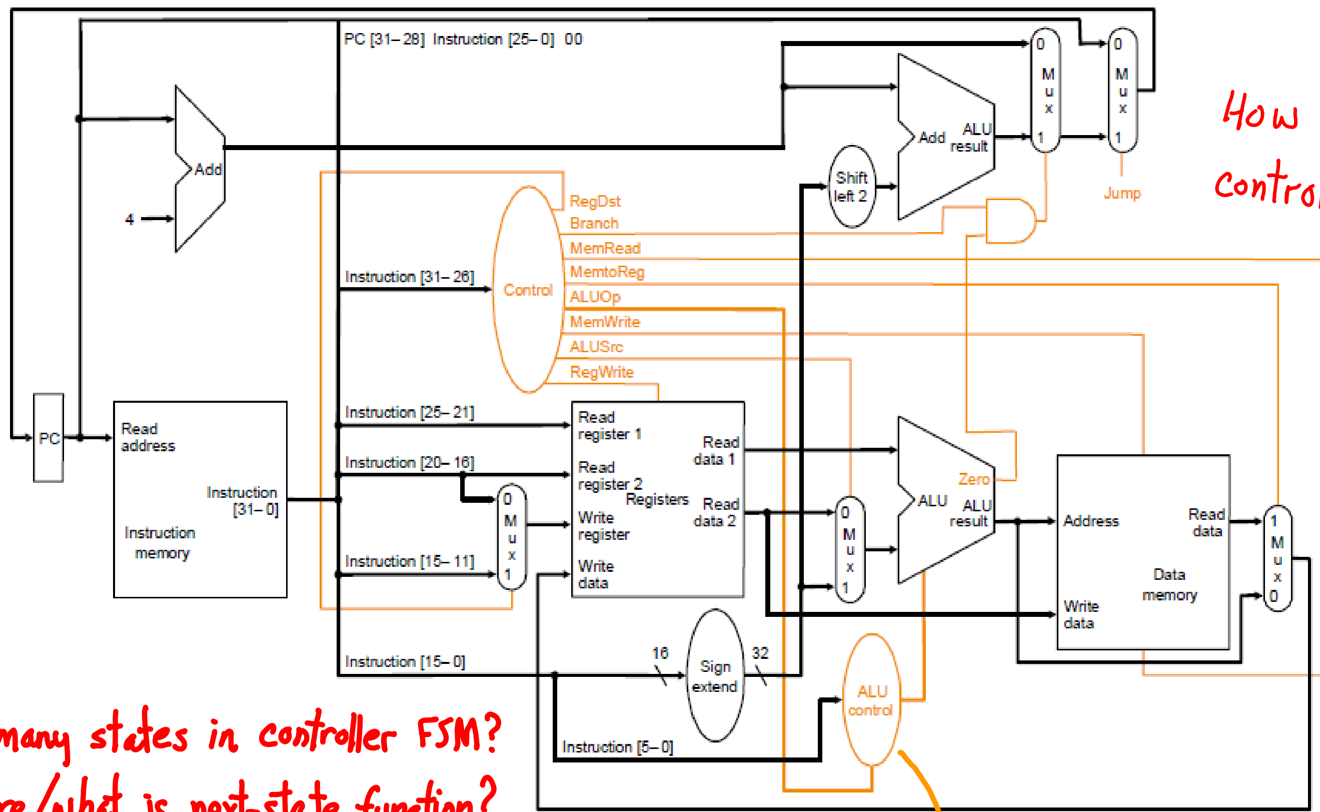
- MUX selects pseudodirect jump target

JMP anywhere within either OS or USER space.

Low 2 bits always 00: ignore them; Upper 4 bits == (x8, super mode) or (x0, user mode)



Putting It All Together:
Your first processor

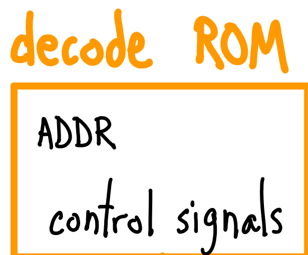


How many controls? simplify

How many states in controller FSM?
Where/what is next-state function?

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

decode ROM



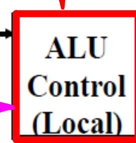
7 + (3)

3

ALUop

op
6

func
6
ALUop
3



ALUctr
3

encode ALU's operation for:
--- r-format arith/logic
--- i-format arith/logic
--- LW, SW address arith

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

simplify ?

opcode = decode ROM Address = row index

ALU decode ROM row index

	add	sub	ori	lw	sw	beq	jump
func	10 0000	10 0010	Not Important				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
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 decode

datapath control signals — decode columns

decode ROM

ALU decode ROM

decode rows

ROM is turned sideways.
This is just a portion of decode ROM and ALU control decode ROM.
6-bit FUNC ===> 64 arith/logic operations per opcode for R-format instructions.
"ADD" is 3-bit code to select ALU's output to be ADD.

simplify?
some opcodes use ALU for Logic/arith, some for address calc. LC3 → get rid of ALU decode? Immed. values?

Single-Cycle MIPS Processor Summary

---- Advantages

=== Simple control logic

=== All instructions execute in 1 cycle (CPI = 1)

=== minimal hardware

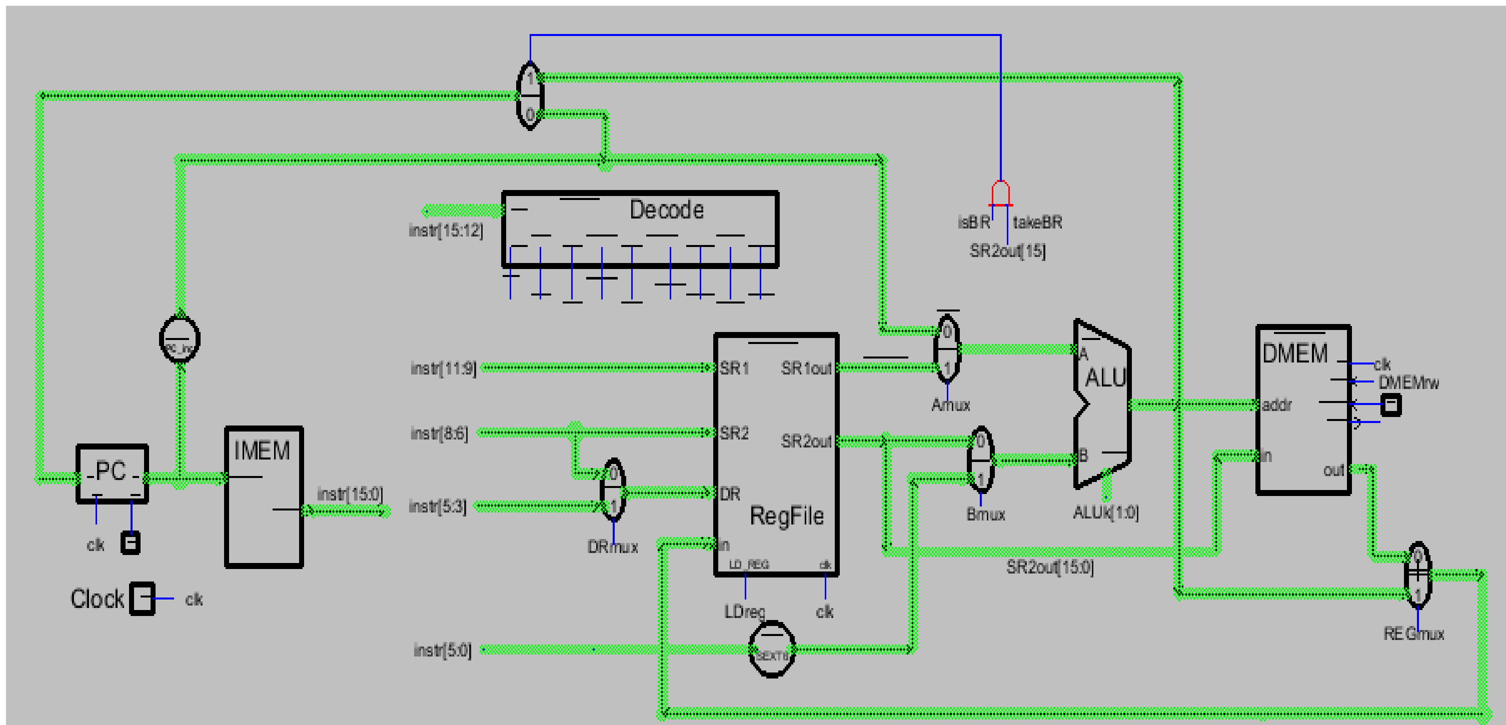
--- Disadvantages

=== Each component is idle most of the time as wave of logic signals traverse entire circuit.

=== Cycle time is sum of component delays ===> very slow CR.

=== Slowest instruction (LW) determines CR for all instructions.

LC4, a 1-cyle, simplified ISA, LC3



ADD AND NOR MOV (pass through), opcodes: 0000, 0001, 0010, 0011

instr[15:12]	instr[11:9]	instr[8:6]	instr[5:3]	instr[2:0]	
OPcode	SR1	SR2	DST	unused	(ALUk == low 2 bits)

ADD R3, R1, R2 ;--- R3 <=== R1 + R2 [0000 001 010 011 *xxx*]
MOV R3, *xx*, R2 ;--- R3 <=== R2 [0011 *xxx* 010 011 *xxx*]

....

LDR STR, opcodes: 1001, 1010

instr[15:12]	instr[11:9]	instr[8:6]	instr[5:0]	
OPcode	baseR	DST/SRC	offset	(ALUk == 2'b00)

LDR R2, R1, x7 ;--- DMEM[R1 + x7] ===> R2 [1001 001 010 000111]

....

LEA LIM, opcodes: 1011, 1100

instr[15:12]	instr[11:9]	instr[8:6]	instr[5:0]	
OPcode	unused	DST	offset	(ALUk == 2'b00, 2'b11)

LEA R2, PC, x7 ;--- R2 <=== PC + x7 [1011 xxx 010 000111]
LIM R2, xx, x7 ;--- R2 <=== x7 [1100 xxx 010 000111]

....

BRR, opcode: 1111

instr[15:12]	instr[11:9]	instr[8:6]	instr[5:0]
OPcode	baseR	CND	offset

BRR R2, R1, x7 ;--- PC <=== R1 + x7 [1111 001 010 000111]
(taken if R2 < 0)