UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

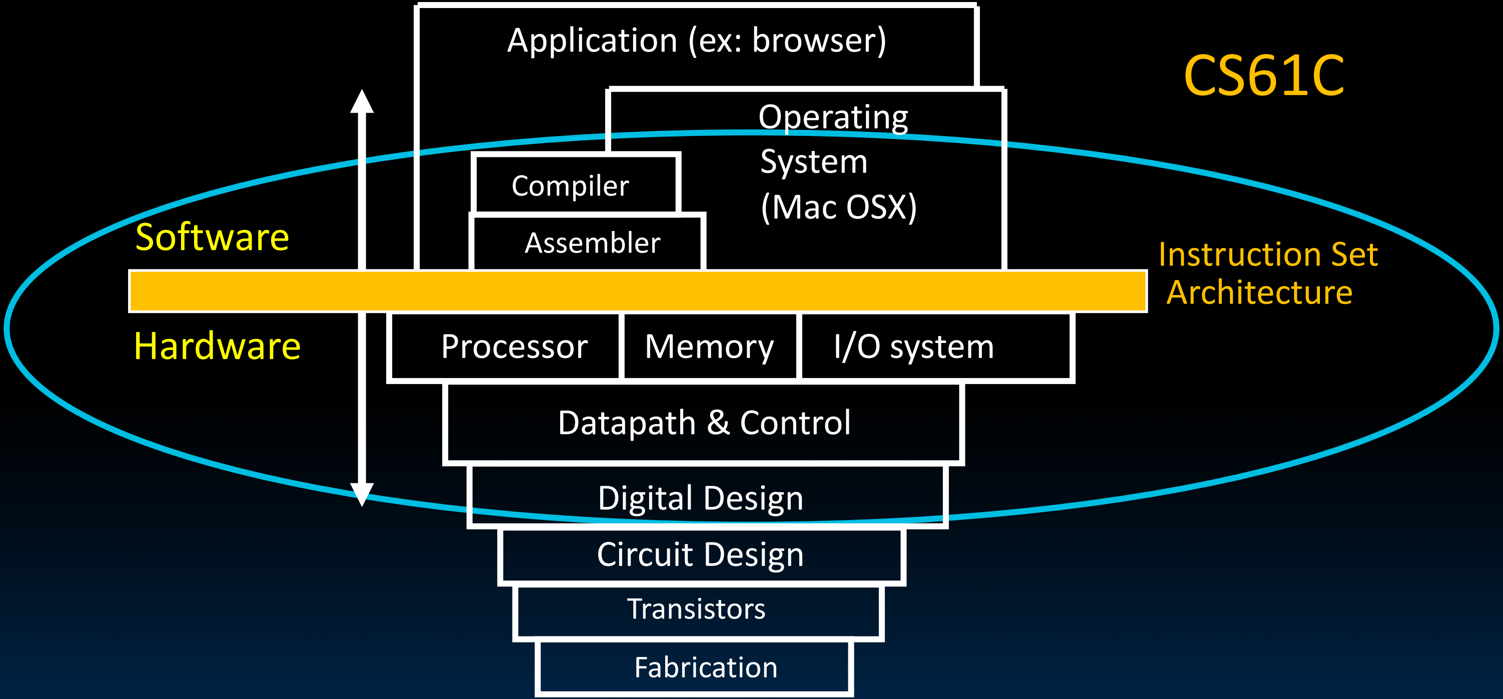
UC Berkeley
Professor
Bora Nikolić

RISC-V Processor Design

Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org

Machine Structures



New-School Machine Structures

Software

Parallel Requests
Assigned to computer
e.g., Search "Cats"

Parallel Threads
Assigned to core e.g., Lookup, Ads

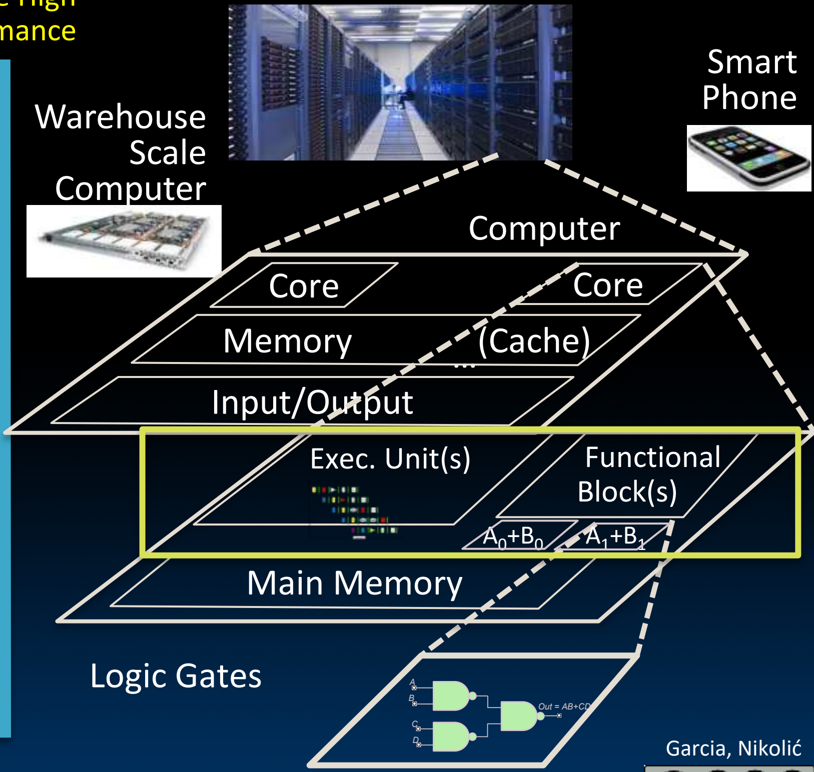
Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

Hardware descriptions
All gates work in parallel at same time

Harness Parallelism & Achieve High Performance

Hardware



Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

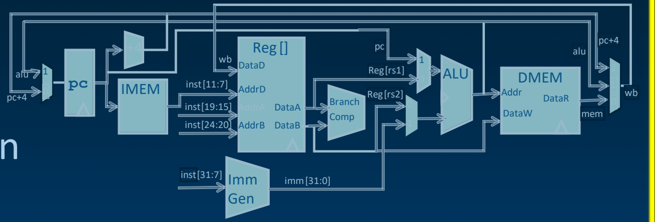
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

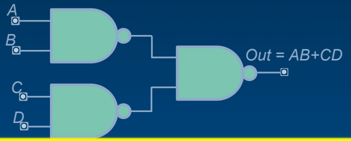
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)

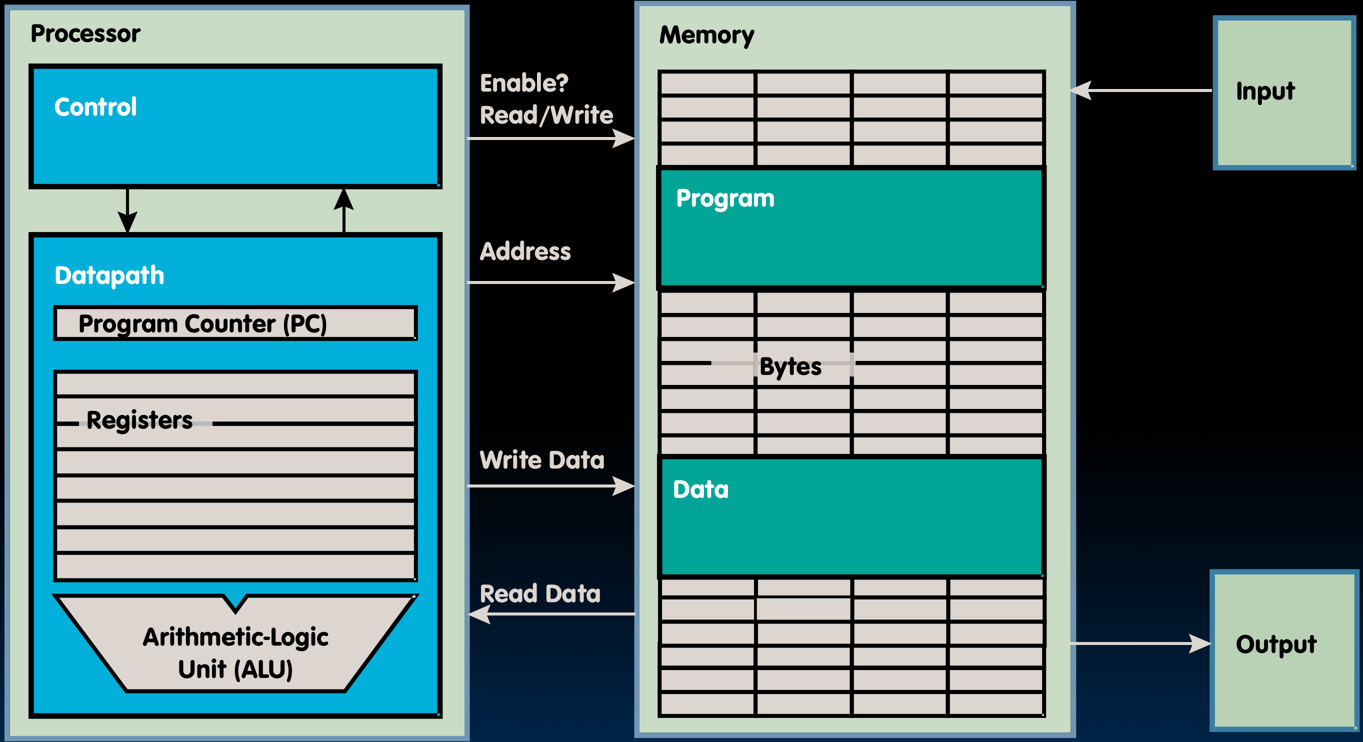


Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Our Single-Core Processor So Far...

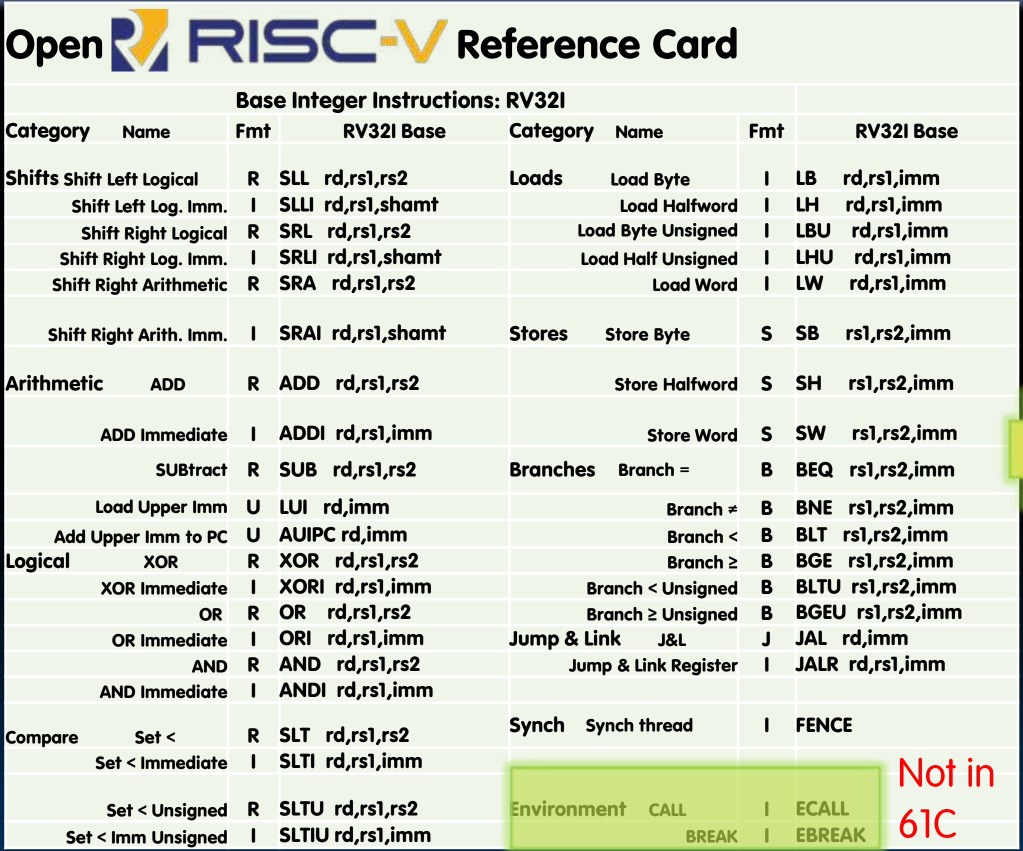


The CPU

- Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making)

- Datapath: portion of the processor that contains hardware necessary to perform operations required by the processor (the brawn)
- Control: portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)

Need to Implement All RV32I Instructions



Open RISC-V Reference Card

Base Integer Instructions: RV32I

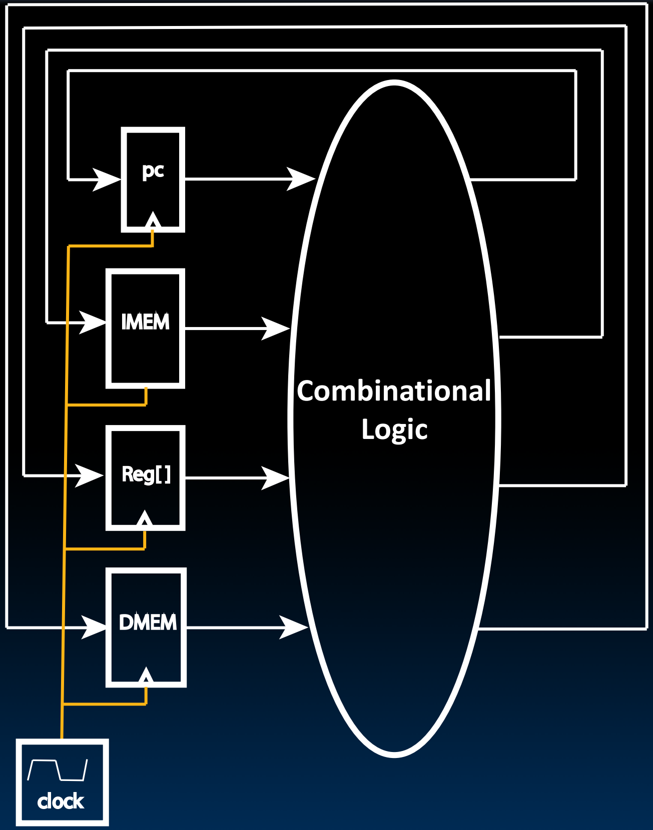
Category	Name	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I	LB rd,rs1,imm
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I	LH rd,rs1,imm
	Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I	LBU rd,rs1,imm
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt		Load Half Unsigned	I	LHU rd,rs1,imm
	Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I	LW rd,rs1,imm
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S	SB rs1,rs2,imm
Arithmetic	ADD	R	ADD rd,rs1,rs2		Store Halfword	S	SH rs1,rs2,imm
	ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S	SW rs1,rs2,imm
	SUBtract	R	SUB rd,rs1,rs2	Branches	Branch =	B	BEQ rs1,rs2,imm
	Load Upper Imm	U	LUI rd,imm		Branch ≠	B	BNE rs1,rs2,imm
	Add Upper Imm to PC	U	AUIPC rd,imm		Branch <	B	BLT rs1,rs2,imm
Logical	XOR	R	XOR rd,rs1,rs2		Branch ≥	B	BGE rs1,rs2,imm
	XOR Immediate	I	XORI rd,rs1,imm		Branch < Unsigned	B	BLTU rs1,rs2,imm
	OR	R	OR rd,rs1,rs2		Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
	OR Immediate	I	ORI rd,rs1,imm	Jump & Link	J&L	J	JAL rd,imm
	AND	R	AND rd,rs1,rs2		Jump & Link Register	I	JALR rd,rs1,imm
	AND Immediate	I	ANDI rd,rs1,imm				
Compare	Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I	FENCE
	Set < Immediate	I	SLTI rd,rs1,imm				
	Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL	I	ECALL
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm		BREAK	I	EBREAK

Not in 61C

Garcia, Nikolić

Building a RISC-V Processor

One-Instruction-Per-Cycle RISC-V Machine



- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

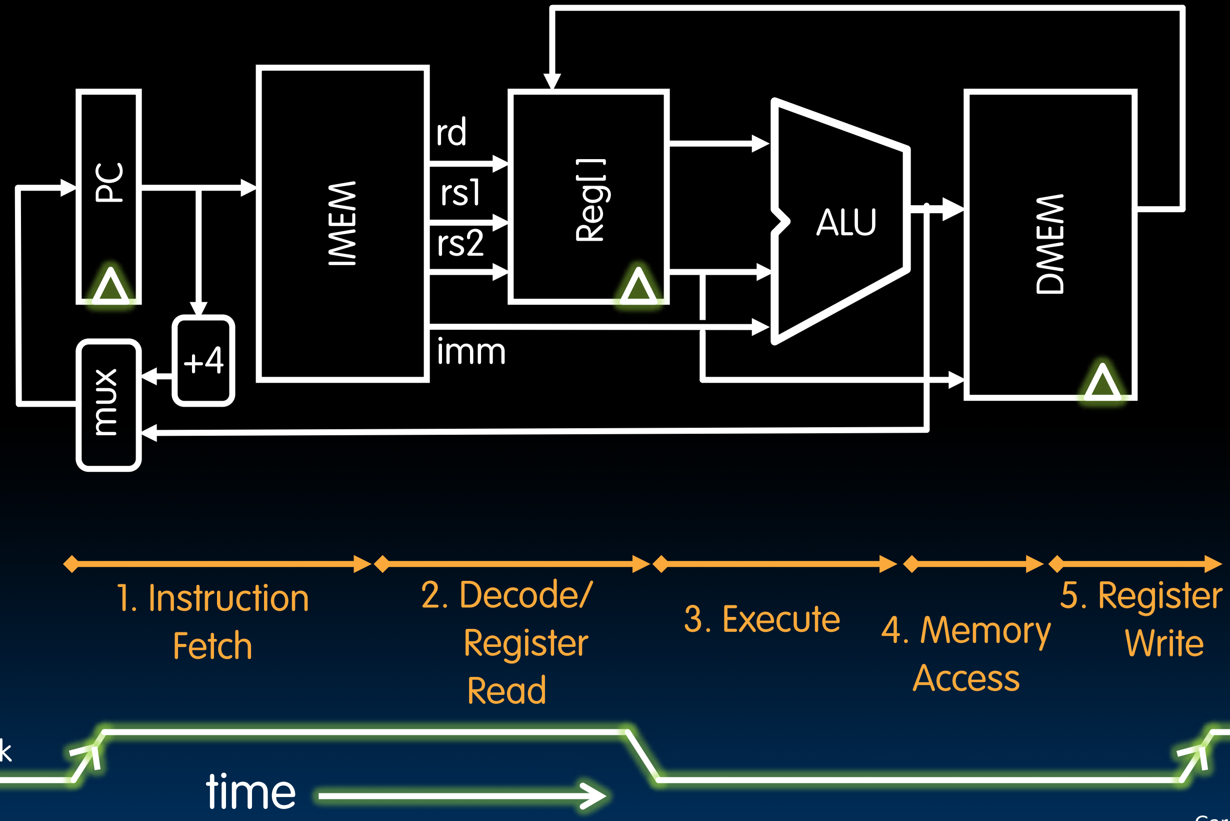
Stages of the Datapath : Overview

- Problem: a single, "monolithic" block that "executes an instruction" (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- Solution: break up the process of "executing an instruction" into stages, and then connect the stages to create the whole datapath
  - smaller stages are easier to design
  - easy to optimize (change) one stage without touching the others (modularity)

Five Stages of the Datapath

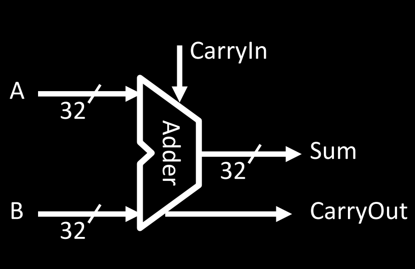
- Stage 1: *Instruction Fetch (IF)*
- Stage 2: *Instruction Decode (ID)*
- Stage 3: *Execute (EX) - ALU* (Arithmetic-Logic Unit)
- Stage 4: *Memory Access (MEM)*
- Stage 5: *Write Back to Register (WB)*

Basic Phases of Instruction Execution

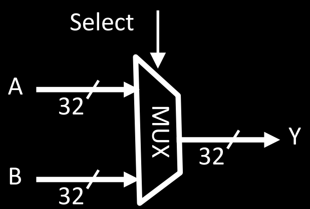


Datapath Components: Combinational

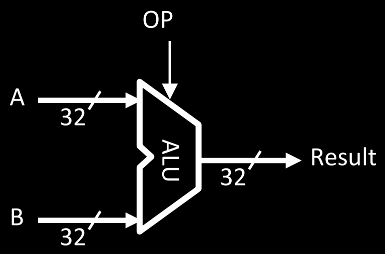
- Combinational elements



Adder



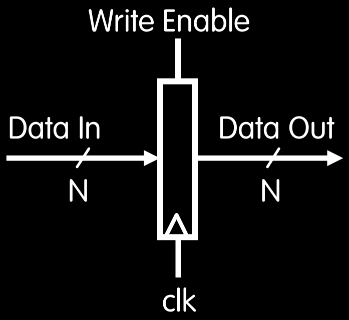
Multiplexer



ALU

- Storage elements + clocking methodology
- Building blocks

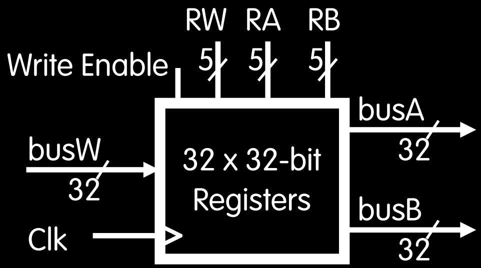
Datapath Elements: State and Sequencing (1/3)



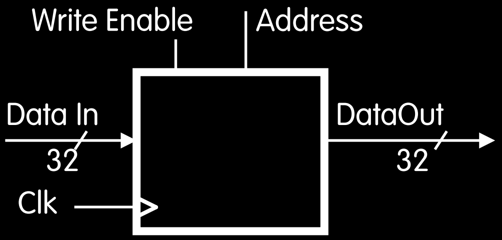
- Register
- Write Enable:
  - Low (or deasserted) (0):
    Data Out will not change
  - Asserted (1): Data Out will become Data In on positive edge of clock

Datapath Elements: State and Sequencing (2/3)

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (Clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after "access time."



Datapath Elements: State and Sequencing (3/3)



- "Magic" Memory
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - For Read: Address selects the word to put on Data Out
  - For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid $\Rightarrow$ Data Out valid after "access time"

State Required by RV32I ISA (1/2)

Each instruction during execution reads and updates the state of : (1) Registers, (2) Program counter, (3) Memory

- Registers (**`x0..x31`**)
  - Register file (*regfile*) **`Reg`** holds 32 registers x 32 bits/register: **`Reg[0]..Reg[31]`**
  - First register read specified by **`rs1`** field in instruction
  - Second register read specified by **`rs2`** field in instruction
  - Write register (destination) specified by *rd* field in instruction
  - **`x0`** is always 0 (writes to **`Reg[0]`** are ignored)
- Program Counter (**`PC`**)
  - Holds address of current instruction

State Required by RV32I ISA (2/2)

- Memory (**`MEM`**)
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We'll use separate memories for instructions (**`IMEM`**) and data (**`DMEM`**)
    - *These are placeholders for instruction and data caches*
  - Instructions are read (*fetched*) from instruction memory (assume **`IMEM`** read-only)
  - Load/store instructions access data memory

R-Type Add Datapath

Review: R-Type Instructions

31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	10 9 8 7	6 5 4 3 2 1 0
R-format : ALU					
[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
7	5	5	3	5	7
func7	rs2	rs1	func3	rd	opcode
0000000	rs2	rs1	000 : ADD	rd	0110011:OP-R
0100000	rs2	rs1	000 : SUB	rd	0110011:OP-R
0000000	rs2	rs1	001 : SLL	rd	0110011:OP-R
0000000	rs2	rs1	010 : SLT	rd	0110011:OP-R
0000000	rs2	rs1	011 : SLTU	rd	0110011:OP-R
0000000	rs2	rs1	100 : XOR	rd	0110011:OP-R
0000000	rs2	rs1	101 : SRL	rd	0110011:OP-R
0100000	rs2	rs1	101 : SRA	rd	0110011:OP-R
0000000	rs2	rs1	110 : OR	rd	0110011:OP-R
0000000	rs2	rs1	111 : AND	rd	0110011:OP-R

- E.g. Addition/subtraction `add rd, rs1, rs2`

  `R[rd] = R[rs1] + R[rs2]`

  `sub rd, rs1, rs2`

  `R[rd] = R[rs1] - R[rs2]`

Garcia, Nikolić

Berkeley UNIVERSITY OF CALIFORNIA

Implementing the `add` instruction

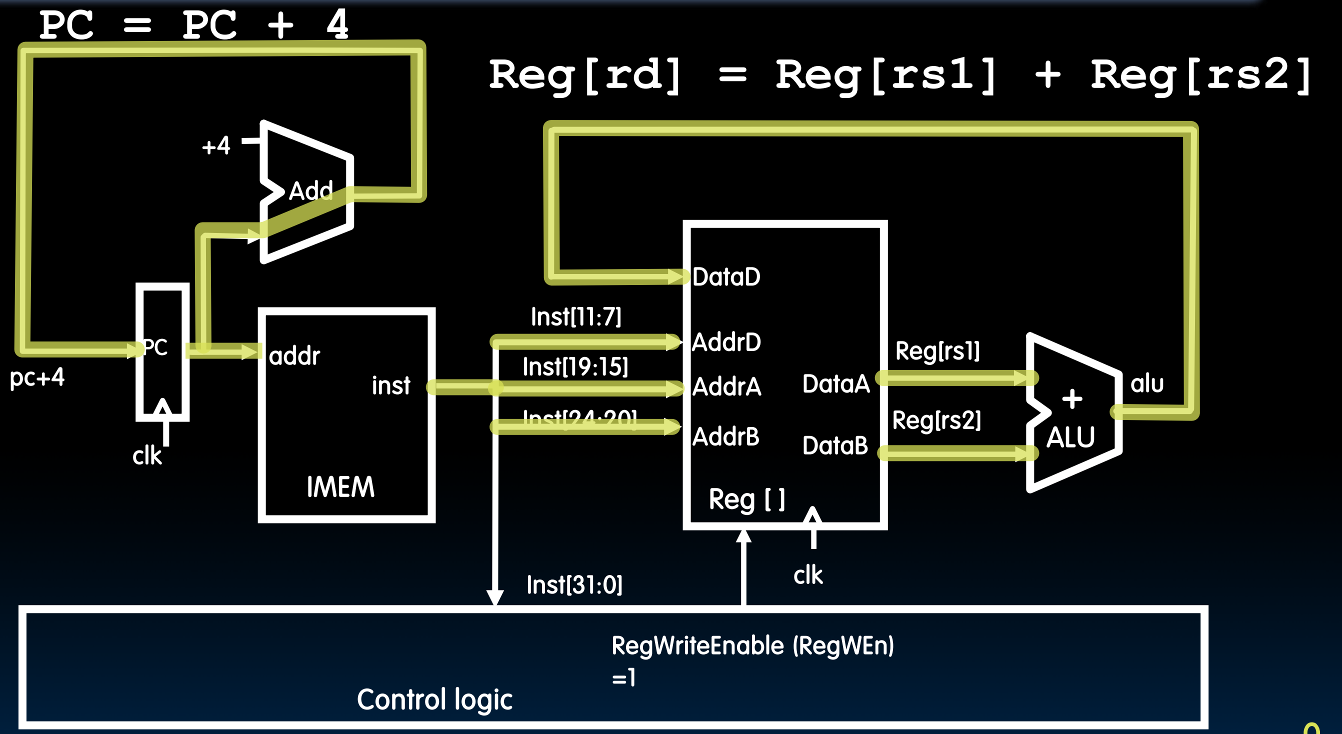
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0110011
7	5	5	3	5	7
add	rs2	rs1	add	rd	Reg-Reg OP

```
add rd, rs1, rs2
```

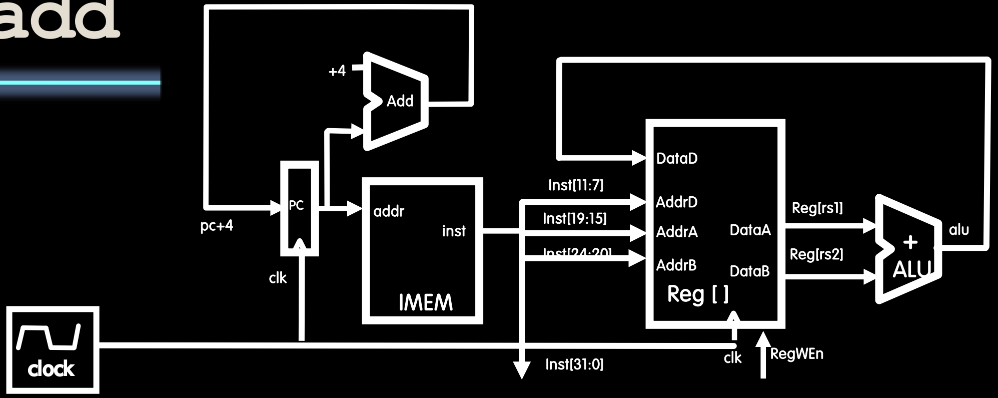
- Instruction makes two changes to machine's state:
  - `Reg[rd] = Reg[rs1] + Reg[rs2]`
  - `PC = PC + 4`

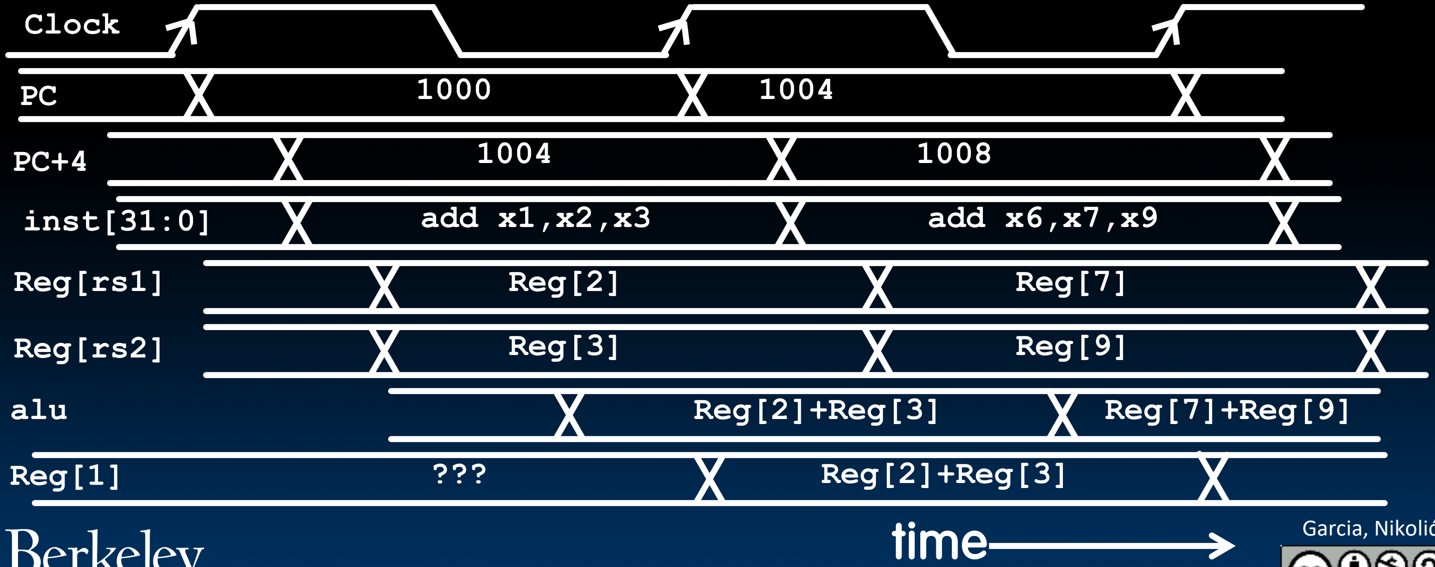
Datapath for add



31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

Timing Diagram for add





Sub Datapath

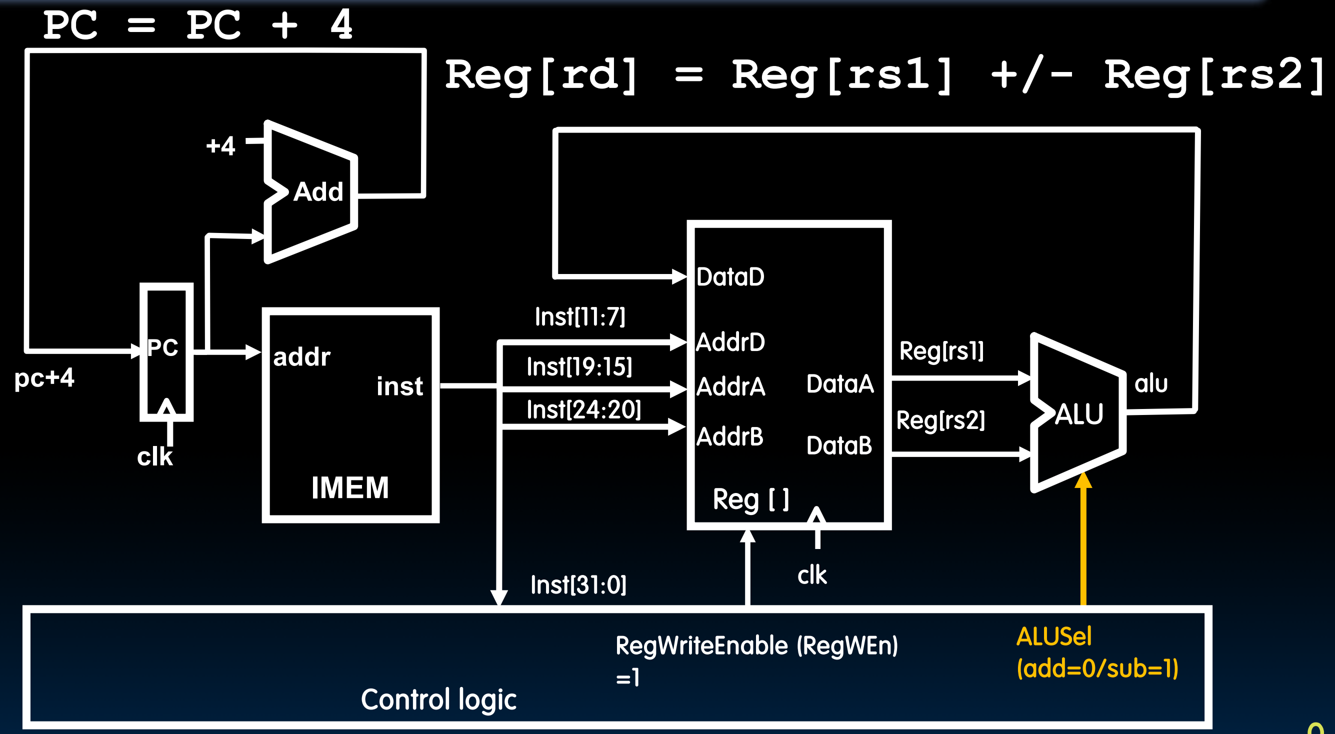
Implementing the `sub` instruction

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

```
sub rd, rs1, rs2
```

- Almost the same as add, except now have to subtract operands instead of adding them
- `inst[30]` selects between add and subtract

Datapath for add/sub



31–25	24–20	19–15	14–12	11–7	6–0
0100000	rs2	rs1	000	rd	0110011
7	5	5	3	5	7

Implementing Other R-Format Instructions

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

Datapath With Immediates

Implementing I-Format - addi instruction

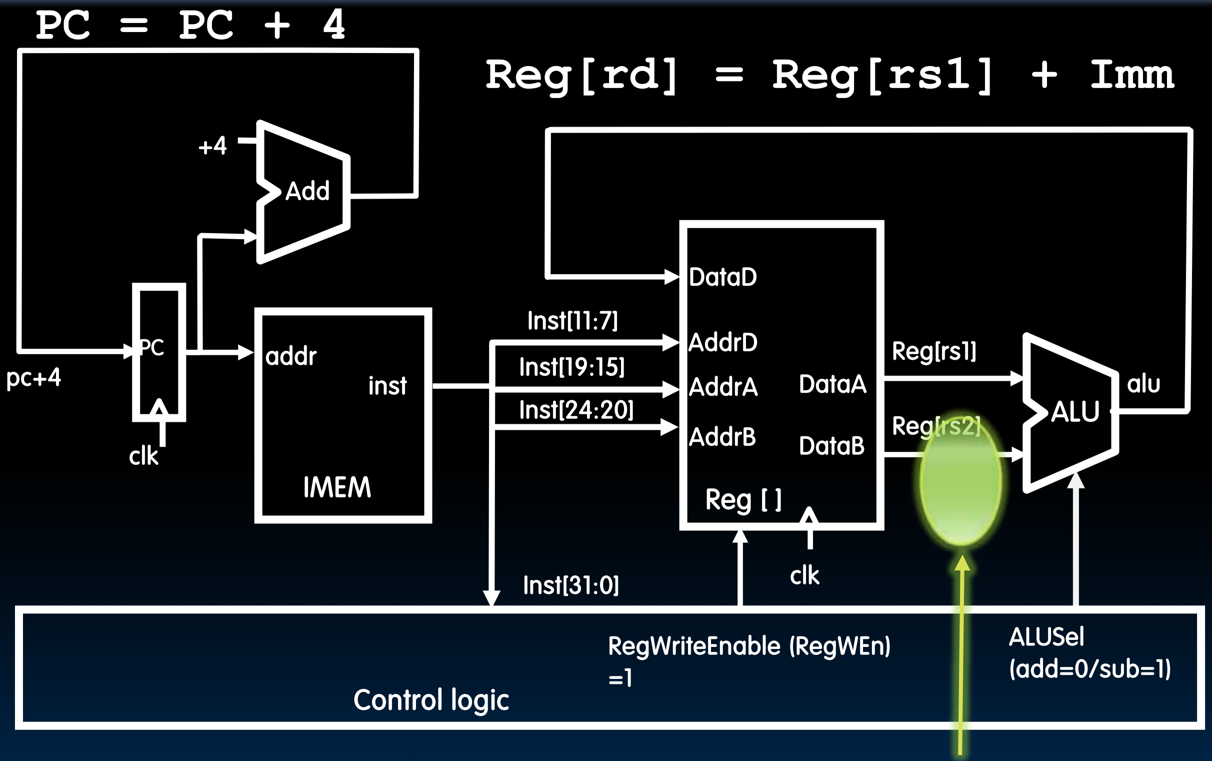
- RISC-V Assembly Instruction:

```
addi   x15,x1,-50
```

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

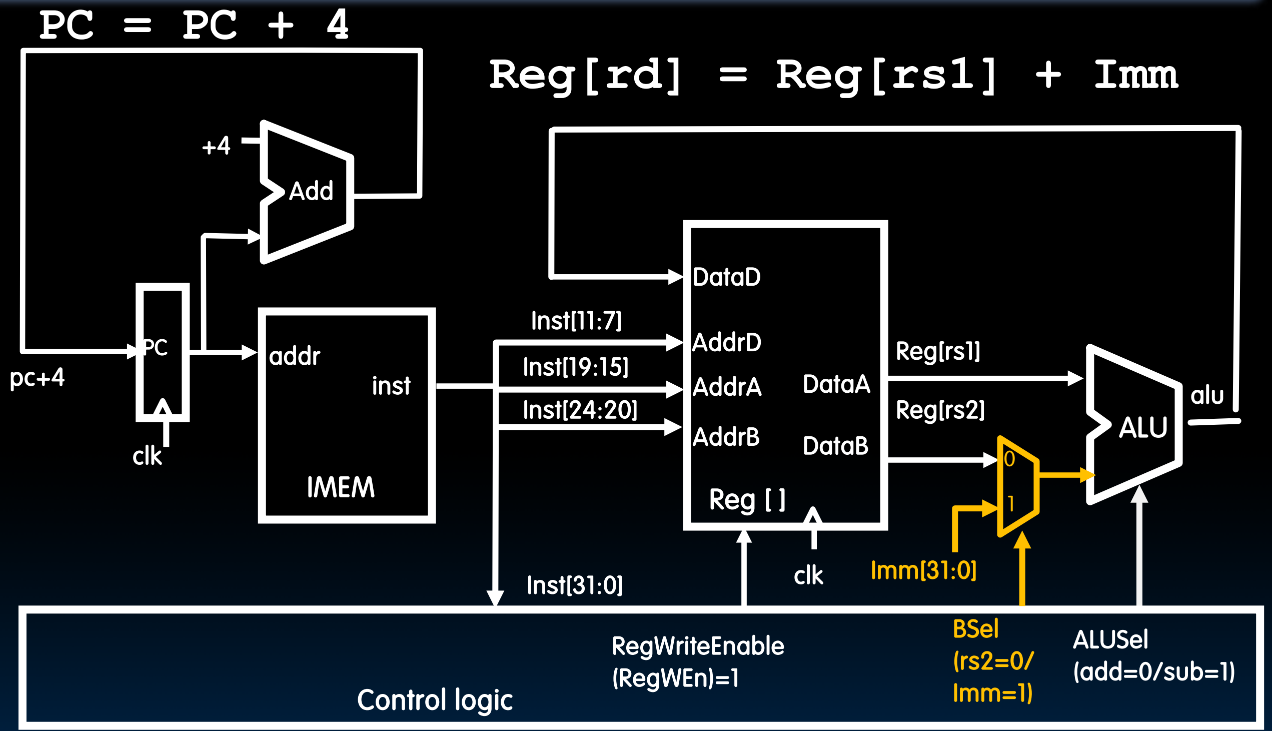
111111001110	00001	000	01111	0010011
imm=-50	rs1=1	add	rd=15	OP-Imm

Datapath for `add/sub`



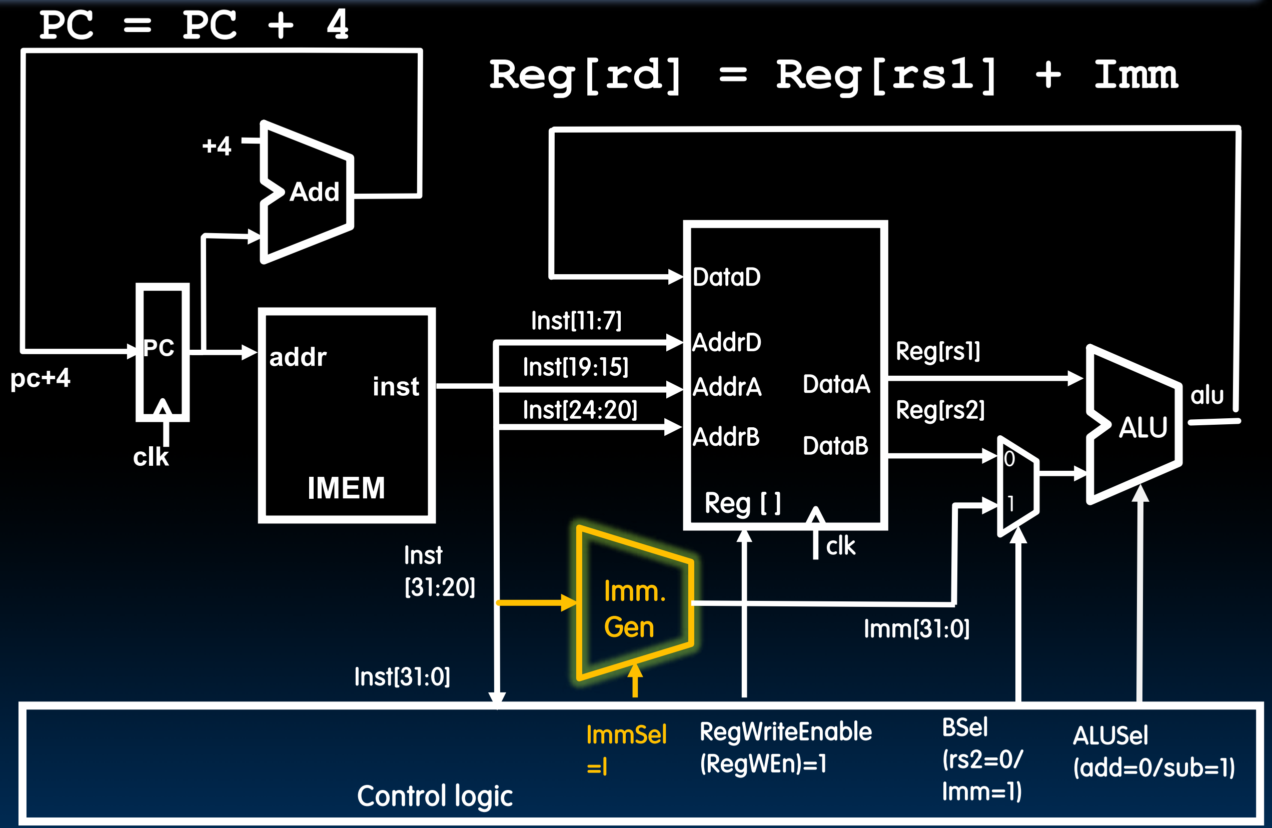
Immediate should be here

Adding `addi` to Datapath



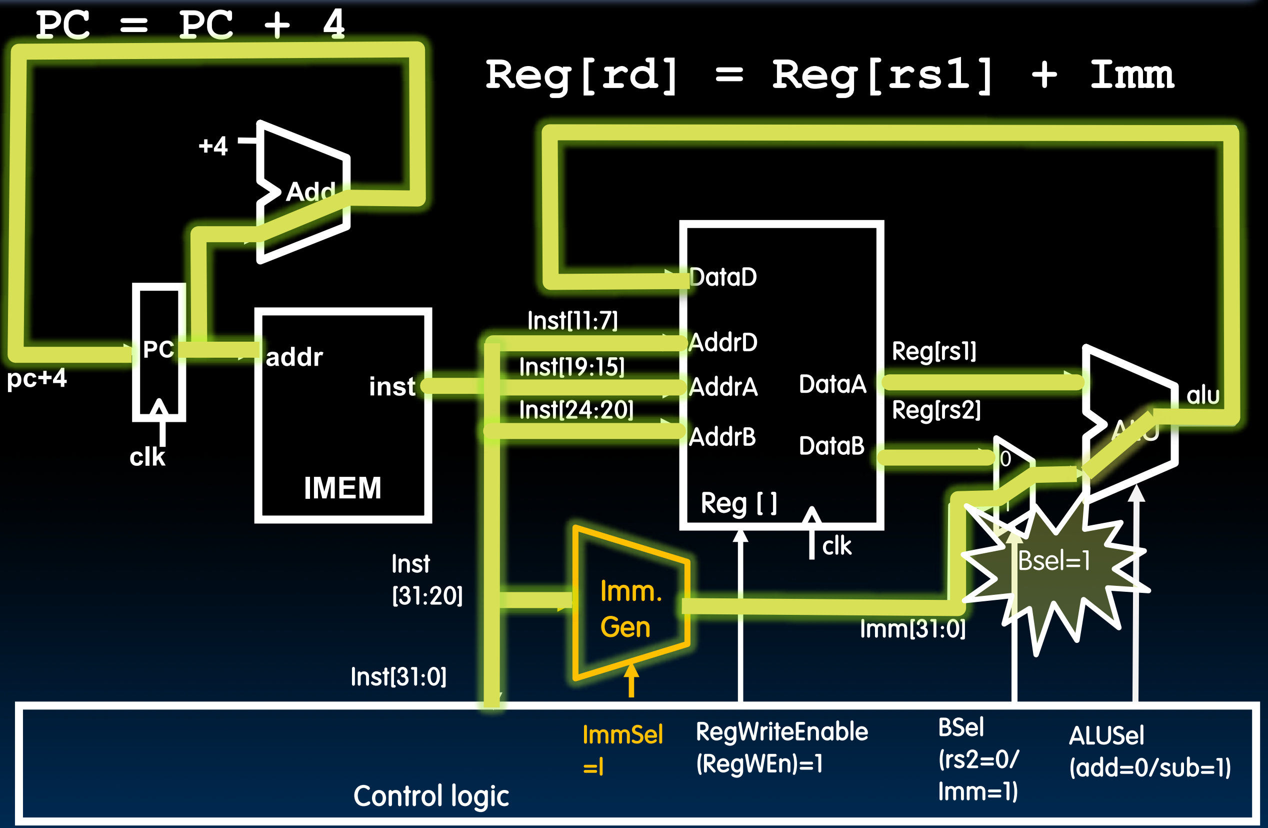
31–20	19–15	14–12	11–7	6–0
imm[11:0]	rs1	000	rd	0010011
12	5	3	5	7

Adding `addi` to Datapath

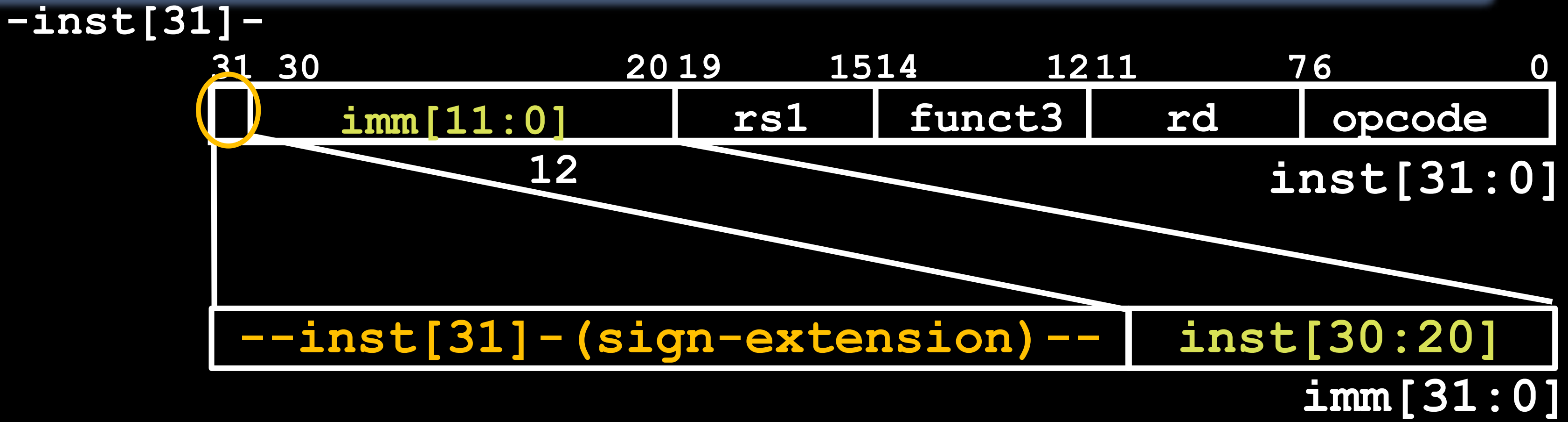


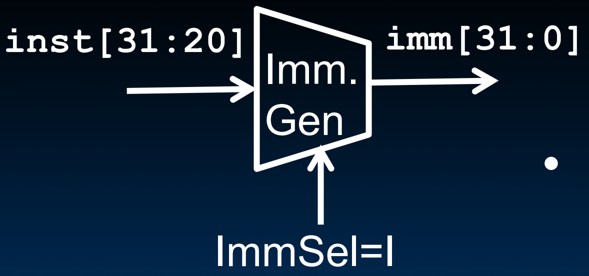
Garcia, Nikolić

Adding addi to Datapath



I-Format Immediates

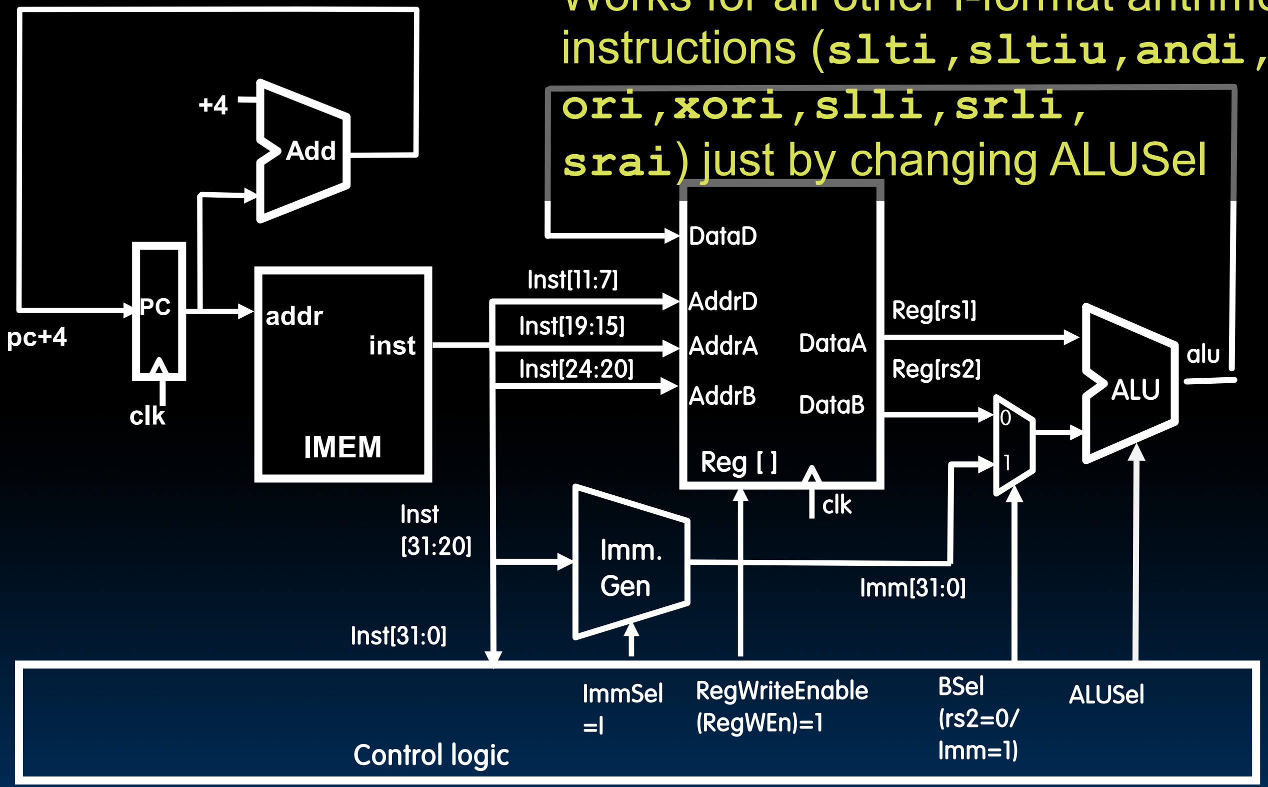




- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

Adding `addi` to Datapath

Works for all other I-format arithmetic instructions (`slti`,`sltiu`,`andi`, `ori`,`xori`,`slli`,`srli`, `srai`) just by changing ALUSel


+4
Add
pc+4
PC
clk
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
Reg[rs1]
Reg[rs2]
0
1
ALU
alu
Inst
[31:20]
Imm.
Gen
Imm[31:0]
Inst[31:0]
ImmSel
=I
RegWriteEnable
(RegWEn)=1
BSel
(rs2=0/
Imm=1)
ALUSel
Control logic