EECS151 : Introduction to Digital Design and ICs

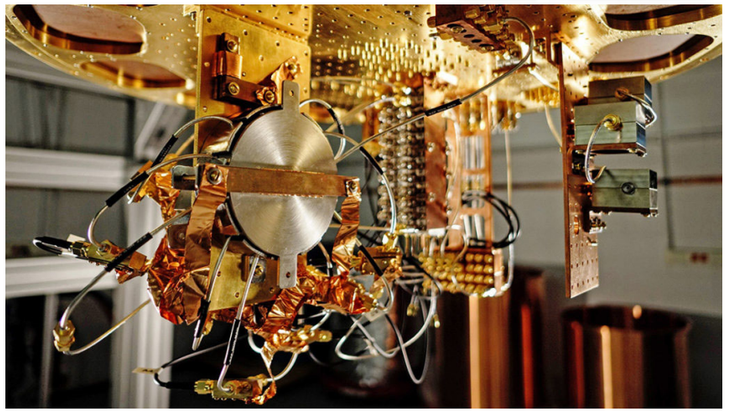
Lecture 7 – Finite State Machines

Bora Nikolić and Sophia Shao

Google claims to have reached 'quantum supremacy'

A paper by Google's researchers seen by the FT, that was briefly posted earlier this week on a Nasa website before being removed, claimed that their processor was able to perform a calculation in three minutes and 20 seconds that would take today's most advanced classical computer, known as Summit, approximately 10,000 years.

Financial Times, Sept 20, 2019.



ERIC LUKERO/GOOGLE

Plain text paper: https://pastebin.com/RfUMXJZE

Review

- Sequential logic:
  - Memory: The outputs depend on both current and previous values of the inputs.
- Finite State Machine:
  - Registers to store current states
  - Combinational logic:
    - Compute the next state
    - Compute the outputs
- Moore vs Mealy FSM:
  - Moore: Outputs depend only on current state
  - Mealy: Outputs depend on current state and inputs

Sequential Logic

Introduction

Finite State Machines

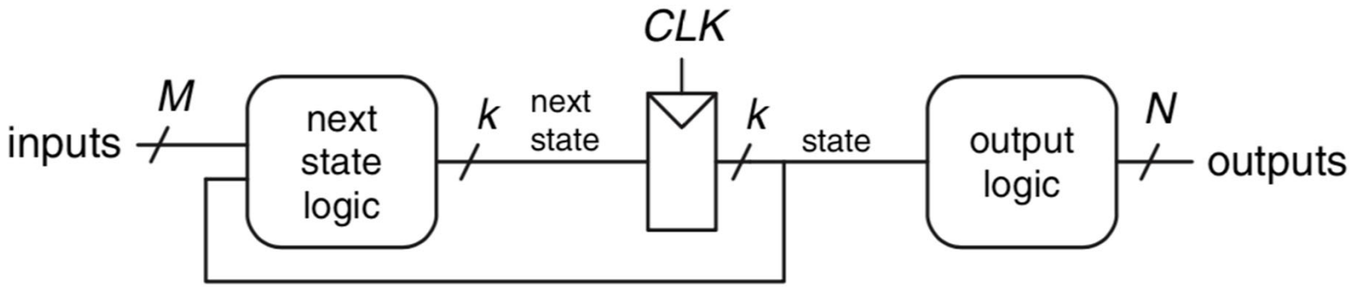
Moore's vs Mealy's FSM

FSM in Verilog

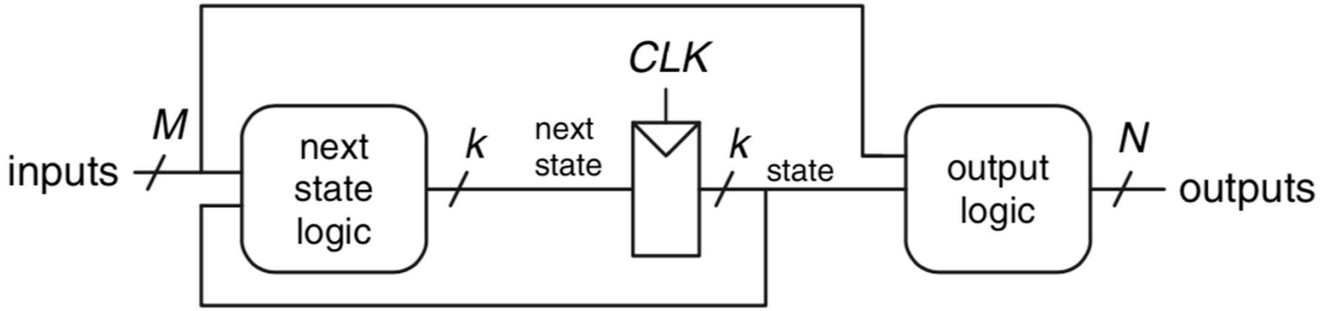
Moore's vs Mealy's FSMs

- Next state is always determined by current state and inputs
- Differ in output logic:
  - Moore FSM: outputs depend only on current state
  - Mealy FSM: outputs depend on current state and inputs

Moore FSM

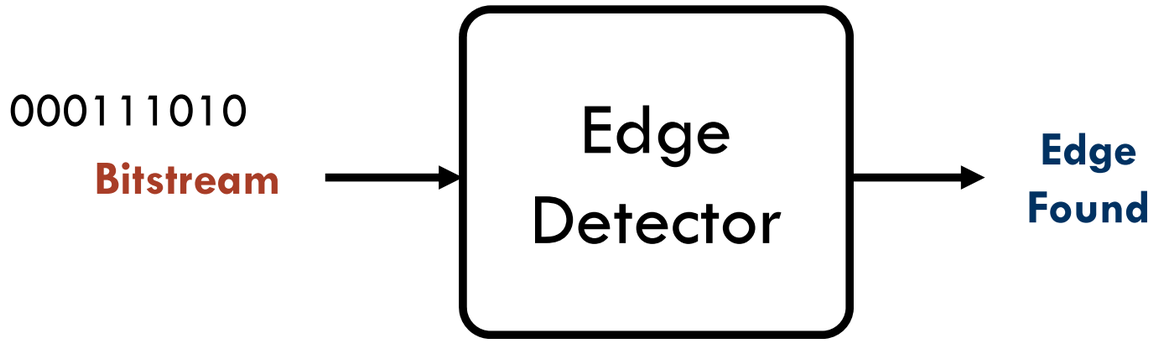


Mealy FSM

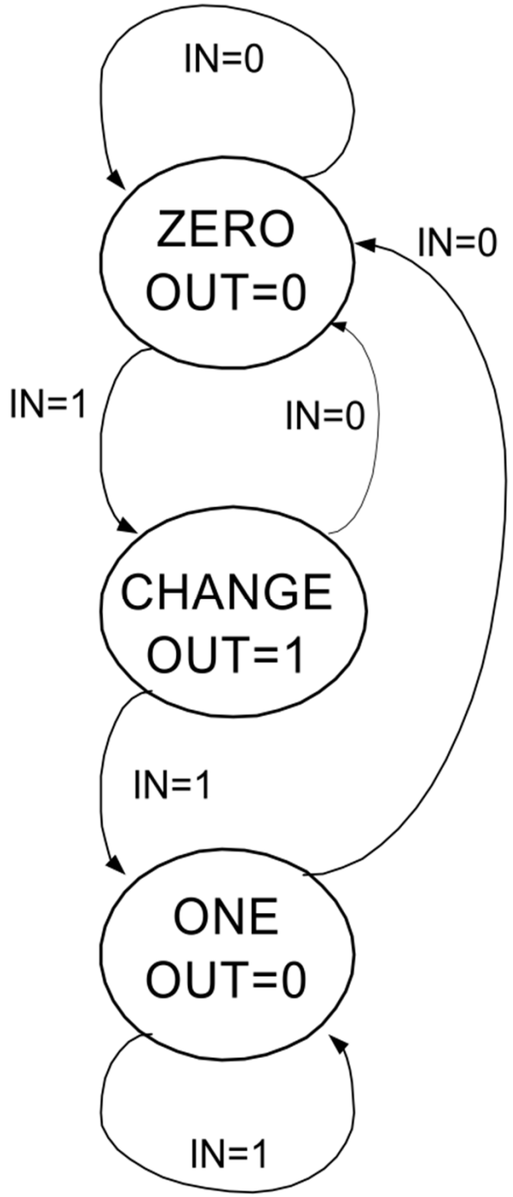


Example: Edge Detector

- Input:
  - A bit stream that is received one bit at a time.
- Output:
  - 0/1
- Circuit:
  - Asserts its output to be true when the input bit stream changes from 0 to 1.

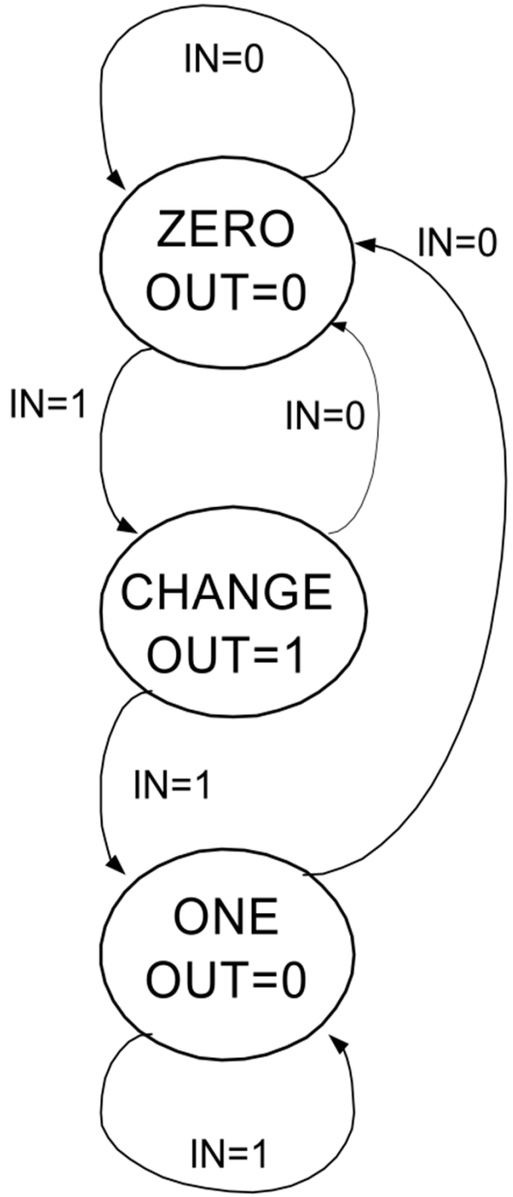


State Transition Diagram Solution A



Input	Current State	Next State	Output
0	Zero (00)	Zero	0

State Transition Diagram Solution A



Input	Current State	Next State	Output
0	Zero (00)	Zero	0
1	Zero (00)	Change	0
0	Change (01)	Zero	1
1	Change (01)	One	1
0	One (11)	Zero	0
1	One (11)	One	0

State Transition Diagram Solution A

CS / IN	00	01	11	10
0	0	0	0	-
1	0	1	1	-

NS_1 = IN *AND* CS0

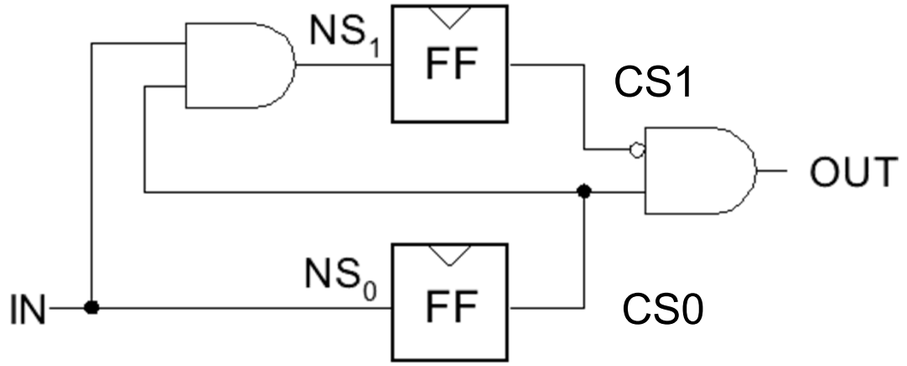
CS / IN	00	01	11	10
0	0	0	0	-
1	1	1	1	-

NS_0 = IN

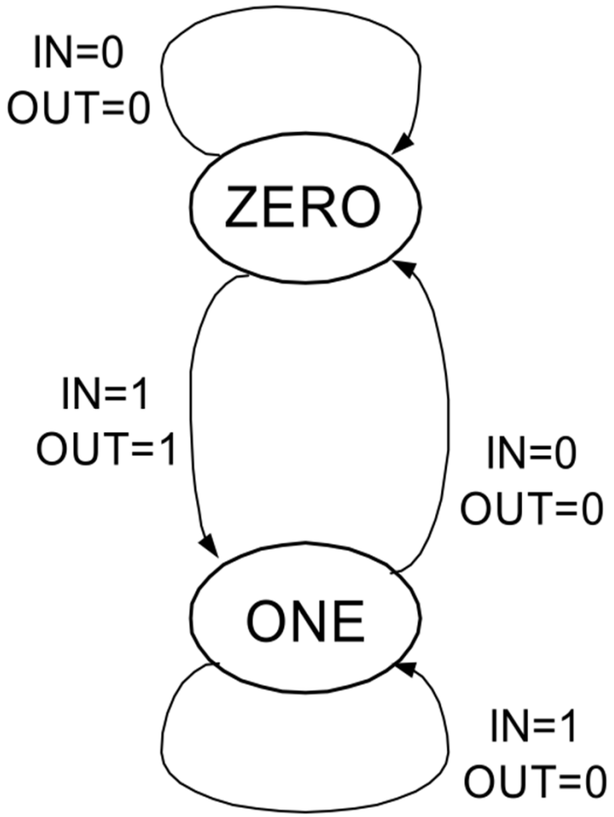
CS / IN	00	01	11	10
0	0	1	0	-
1	0	1	0	-

OUT= *NOT* (CS1) *AND* CS0

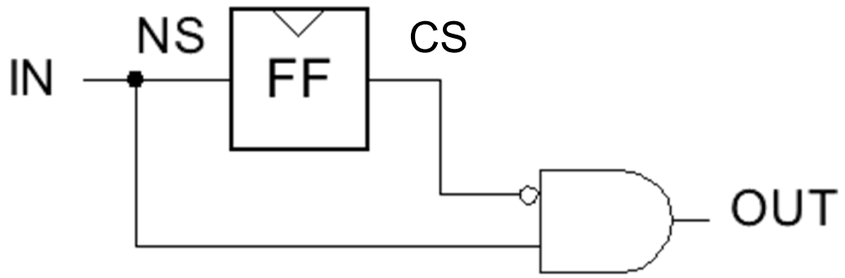
Input	Current State	Next State	Output
0	Zero (00)	Zero	0
1	Zero (00)	Change	0
0	Change (01)	Zero	1
1	Change (01)	One	1
0	One (11)	Zero	0
1	One (11)	One	0



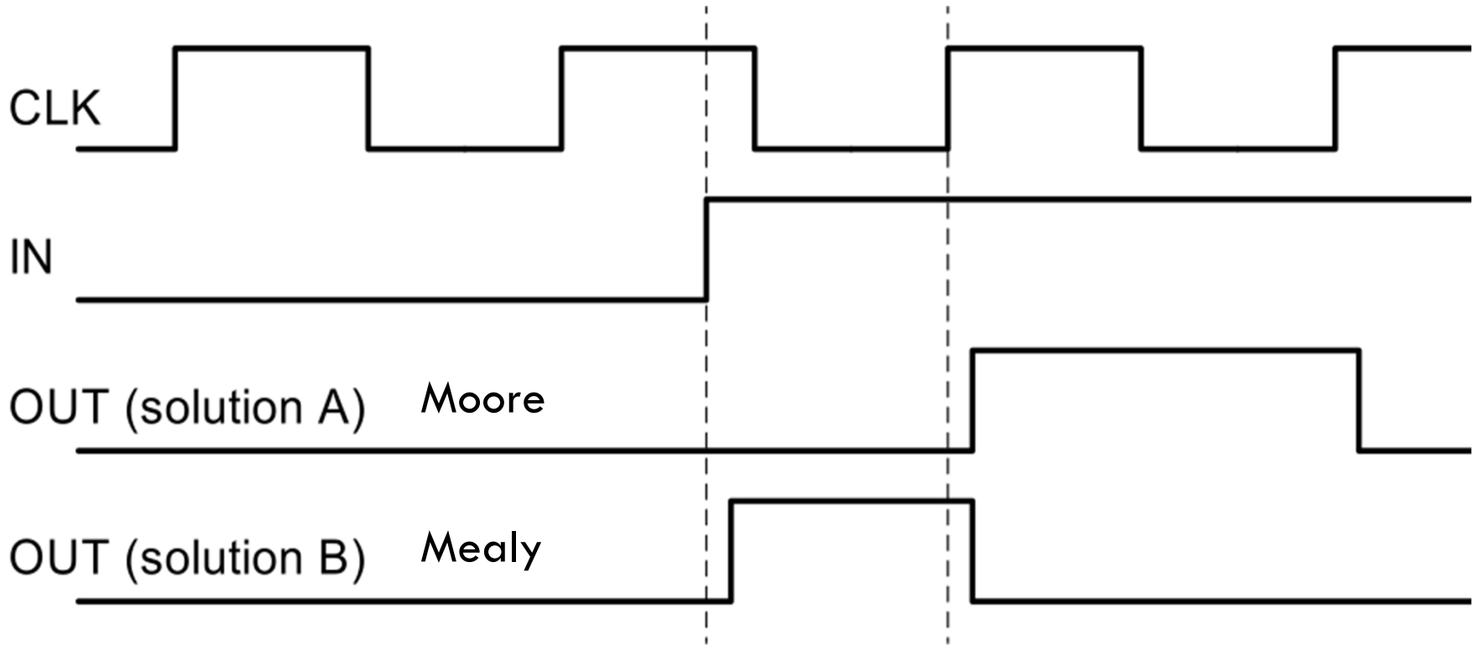
State Transition Diagram Solution B



Input	Current State	Next State	Output
0	Zero (0)	Zero	0
1	Zero (0)	One	1
0	One (1)	Zero	0
1	One (1)	One	0



Edge Detection Timing Diagrams



- Solution A (Moore) : both edges of output follow the clock
- Solution B (Mealy) : output rises with input rising edge and is asynchronous wrt the clock, output fails synchronous with next clock edge

FSM Comparison

Solution A

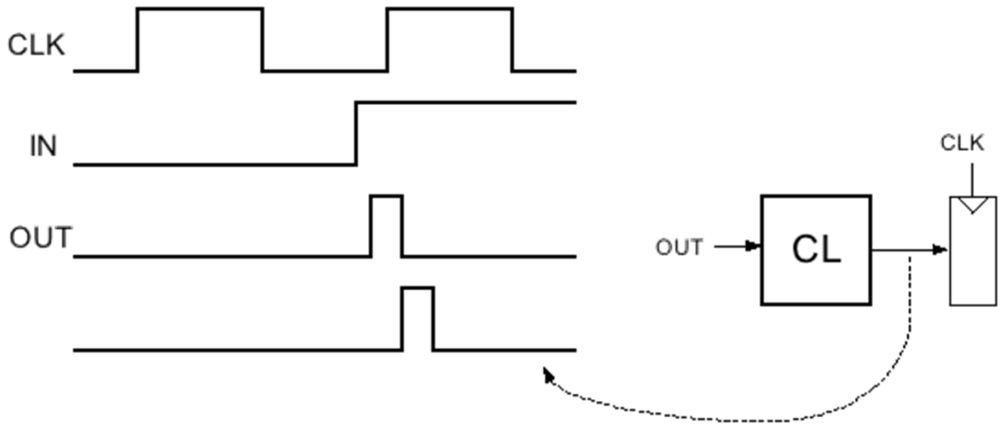
Moore Machine

- output function only of current state
- maybe more states (why?)
- synchronous outputs
  - Input glitches not send at output
  - one cycle "delay"
  - full cycle of stable output

Solution B

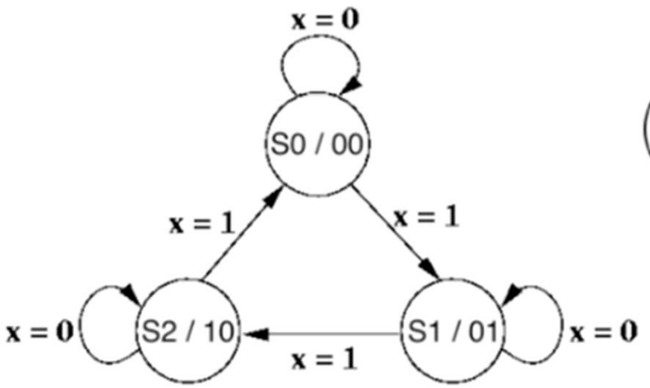
Mealy Machine

- output function of both current = & input
- maybe fewer states
- asynchronous outputs
  - if input glitches, so does output
  - output immediately available
  - output may not be stable long enough to be useful (below):

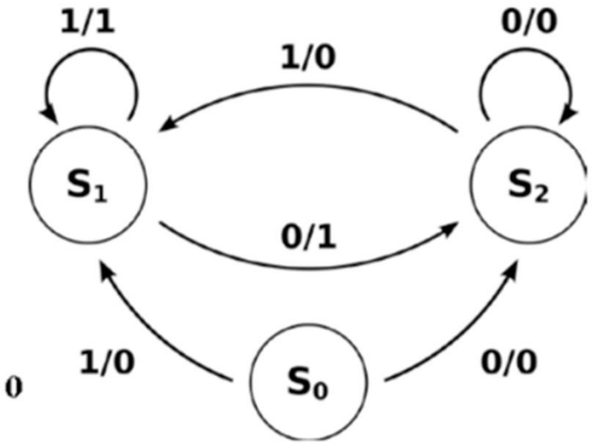


If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate setup time requirement)

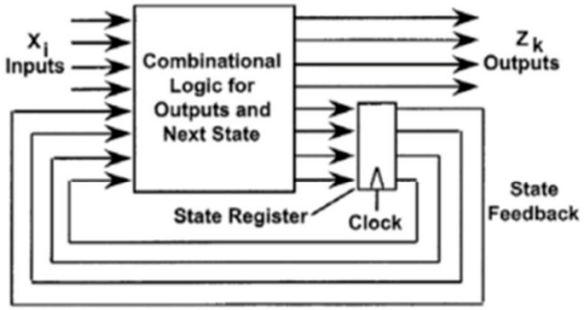
Quiz: Which of the diagrams are **Moore** machines?



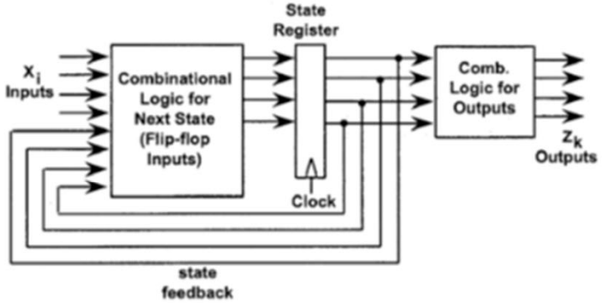
A.



B.



C.



D.

A. AC
B. BD
C. AD
D. BC

www.yellkey.com/offline

Sequential Logic

Introduction

Finite State Machines

Moore vs Mealy FSM

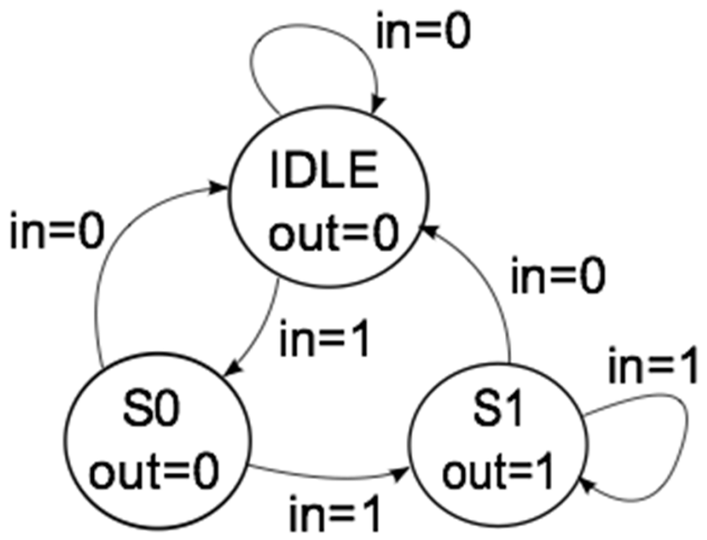
FSM in Verilog

Implement FSM with Verilog

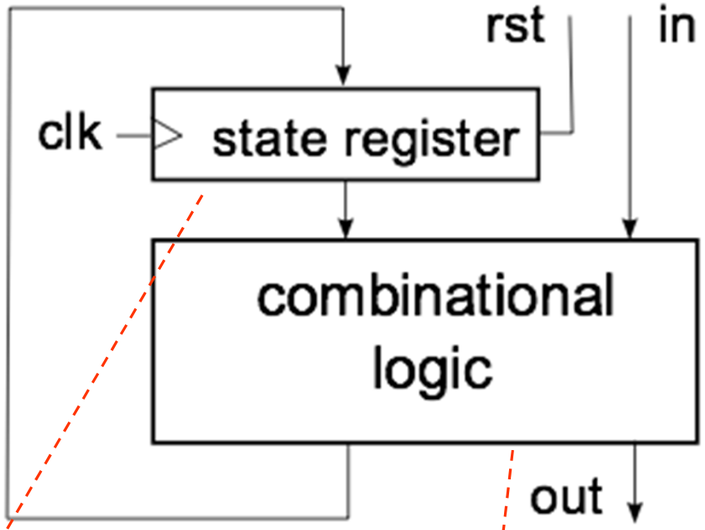
- Specify circuit function
- Draw state transition diagram
- Write down symbolic state transition table
- Assign encodings (bit patterns) to symbolic states
- Code as Verilog behavioral description
  - Use parameters to represent encoded states
  - Use separate always blocks for register assignment and combinational logic block
  - Use case statement for combinational logic.
    - Within each case section (state), assign outputs and next state based on inputs
    - Moore: outputs only dependent on states not on inputs

Finite State Machine in Verilog

State Transition Diagram



Circuit Diagram



Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)

next state = f(in, current state)

Finite State Machine in Verilog

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] current_state, next_state;

// always block for state register
always @(posedge clk)
   if (rst) current_state <= IDLE;
   else current_state <= next_state;
```

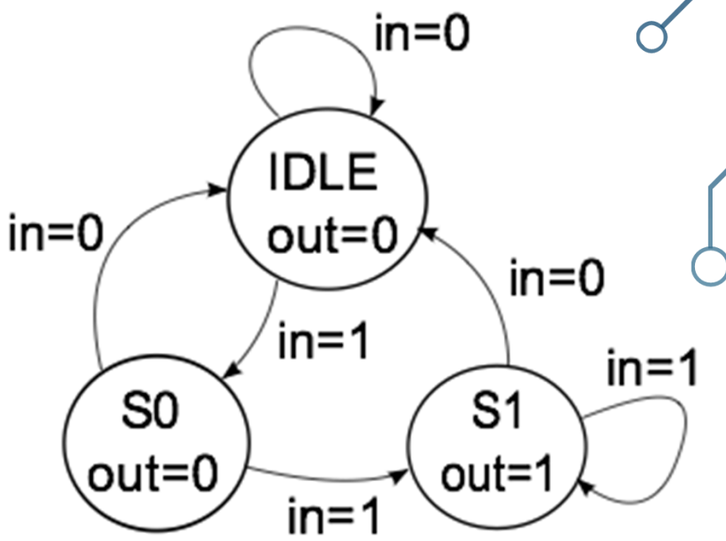
Must use reset to force to initial state.

reset not always shown in STD

Constants local to this module.

Combinational logic signals for transition.

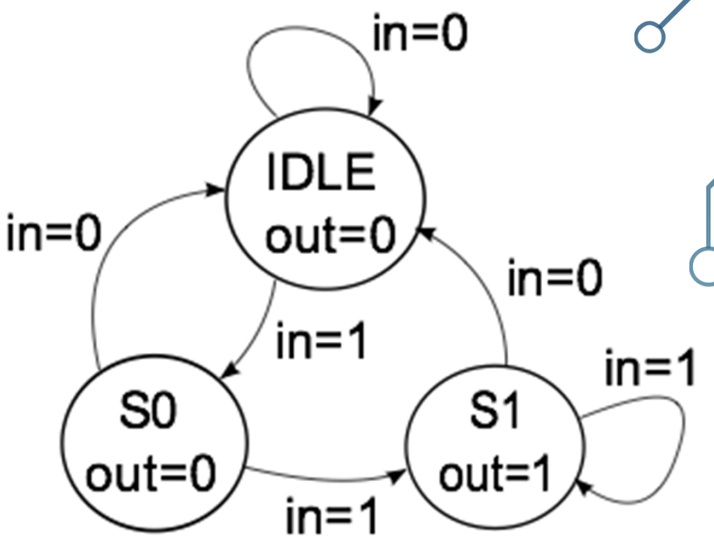
THE register to hold the "state" of the FSM.



A separate always block should be used for combination logic part of FSM. Next state and output generation. (Always blocks in a design work in parallel.)

Finite State Machine in Verilog (cont.)

```
// always block for combinational logic portion
always @(current_state or in)
case (current_state)
// For each state def output and next
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  default: begin
             next_state = IDLE;
             out = 1'b0;
           end
endcase
endmodule
```



Each state becomes a case clause.

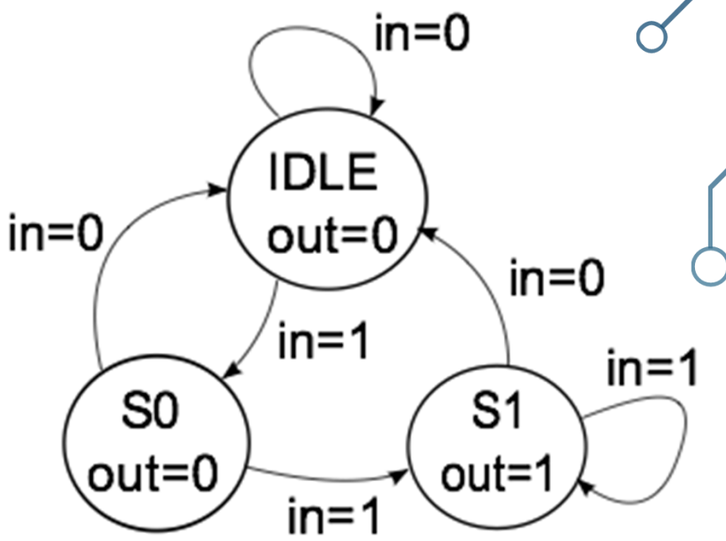
For each state define:

Output value(s)

State transition

Use "default" to cover unassigned state. Usually unconditionally transition to reset state.

Finite State Machine in Verilog (cont.)




```
always @*
  begin
    next_state = IDLE;
    out = 1'b0;
    case (state)
      IDLE    : if (in == 1'b1) next_state = S0;
      S0      : if (in == 1'b1) next_state = S1;
      S1      : begin
                  out = 1'b1;
                  if (in == 1'b1) next_state = S1;
                end
      default: ;
    endcase
  end
endmodule
```

* for sensitivity list

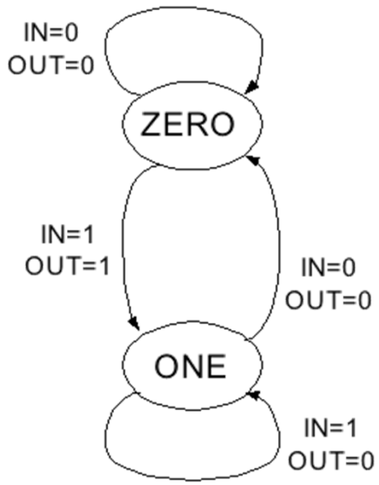
Nominal values: used unless specified below.

Within case only need to specify exceptions to the nominal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

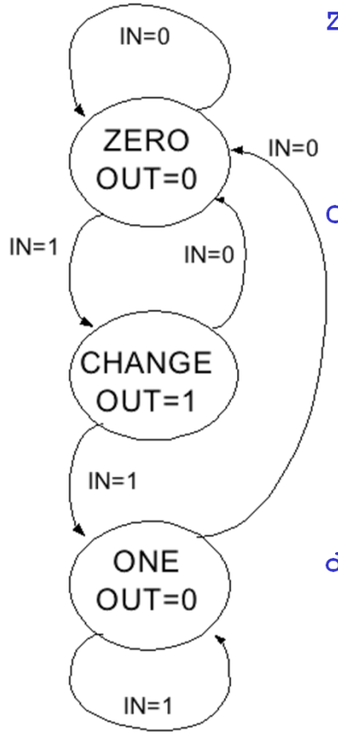
Edge Detector Example

Mealy Machine




```
always @(posedge clk)
     if (rst) ps <= ZERO;
     else ps <= ns;
always @(ps in)
    case (ps)
      ZERO: if (in) begin
              out = 1'b1;
              ns = ONE;
            end
        else begin
          out = 1'b0;
          ns = ZERO;
        end
      ONE: if (in) begin
        out = 1'b0;
        ns = ONE;
       end
       else begin
         out = 1'b0;
         ns = ZERO;
       end
       default: begin
            out = 1'bx;
            ns = default;
           end
```

Moore Machine




```
always @(posedge clk)
     if (rst) ps <= ZERO;
     else ps <= ns;
always @(ps in)
    case (ps)
      ZERO: begin
          out = 1'b0;
          if (in) ns = CHANGE;
                  else ns = ZERO;
        end
      CHANGE: begin
          out = 1'b1;
          if (in) ns = ONE;
          else ns = ZERO;
        end
       ONE: begin
          out = 1'b0;
          if (in) ns = ONE;
          else ns = ZERO;
      default: begin
          out = 1'bx;
          ns = default;
        end
```

Administrivia

- Midterm 1: October 2, in class
  - One double-sided page of hand-written notes
- Midterm 2: November 6

Building a RISC-V Processor

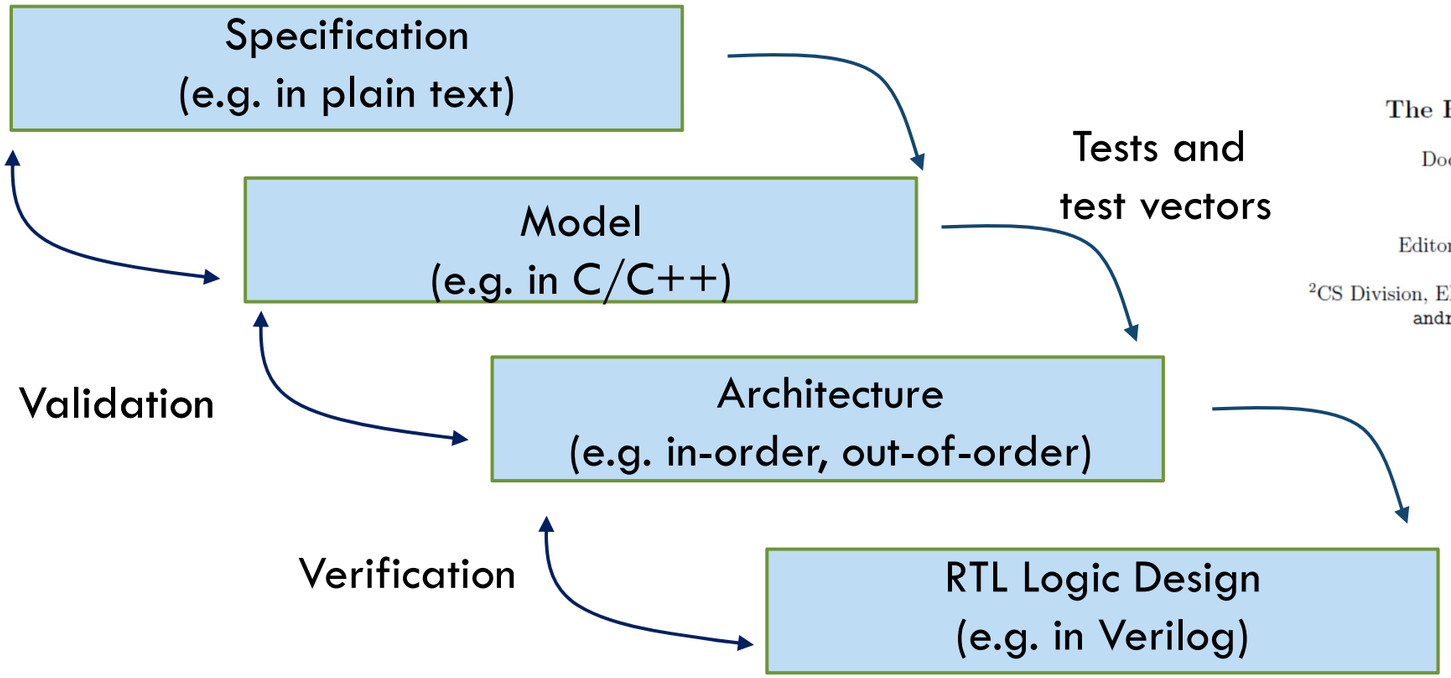
Berkeley RISC-V ISA

www.riscv.org

- An open, license-free ISA
  - Runs GCC, LLVM, Linux distributions, …
  - RV32, RV64, and RV128 variants for 32b, 64b, and 128b address spaces
- Originally developed for teaching classes at Berkeley, now widely adopted
- Base ISA only ~40 integer instructions
- Extensions provide full general-purpose ISA, including IEEE-754/2008 floating-point
- Designed for extension, customization
- Developed at UC Berkeley, now maintained by RISC-V Foundation
- Open and commercial implementations
- RISC-V ISA, datapath, and control covered in CS61C; summarized here

RISC-V Processor Design

- Spec: Unprivileged ISA, RV32I (and a look at RV64I)

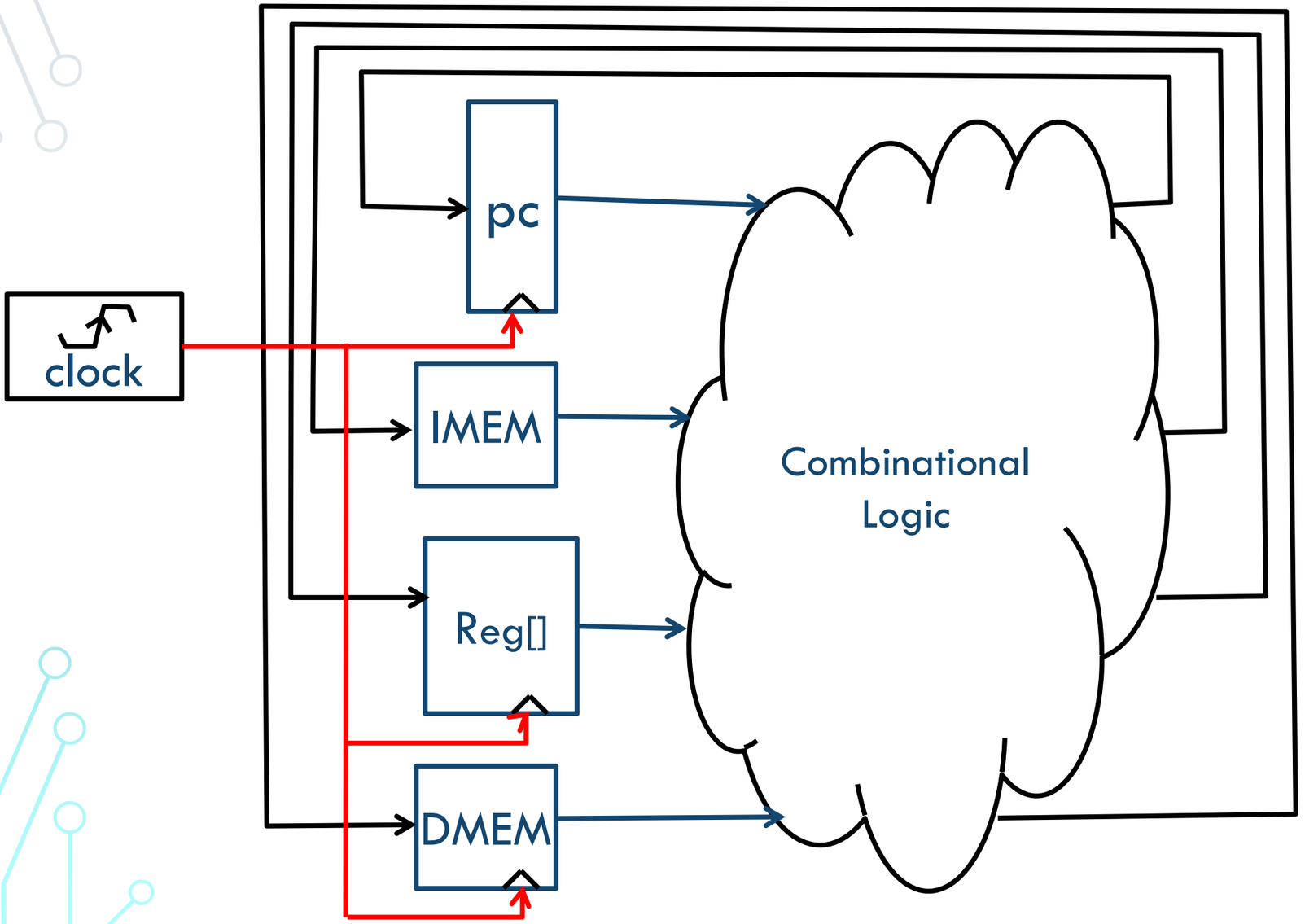


The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20190608-Base-Ratified

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
June 8, 2019

- Tests provided as a part of the project
- Architecture: Single-cycle and pipelined in-order processor
  - Expanded from CS61C

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

State Required by RV32I ISA

Each instruction reads and updates this state during execution:

- 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

- 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
  - Load/store instructions access data memory

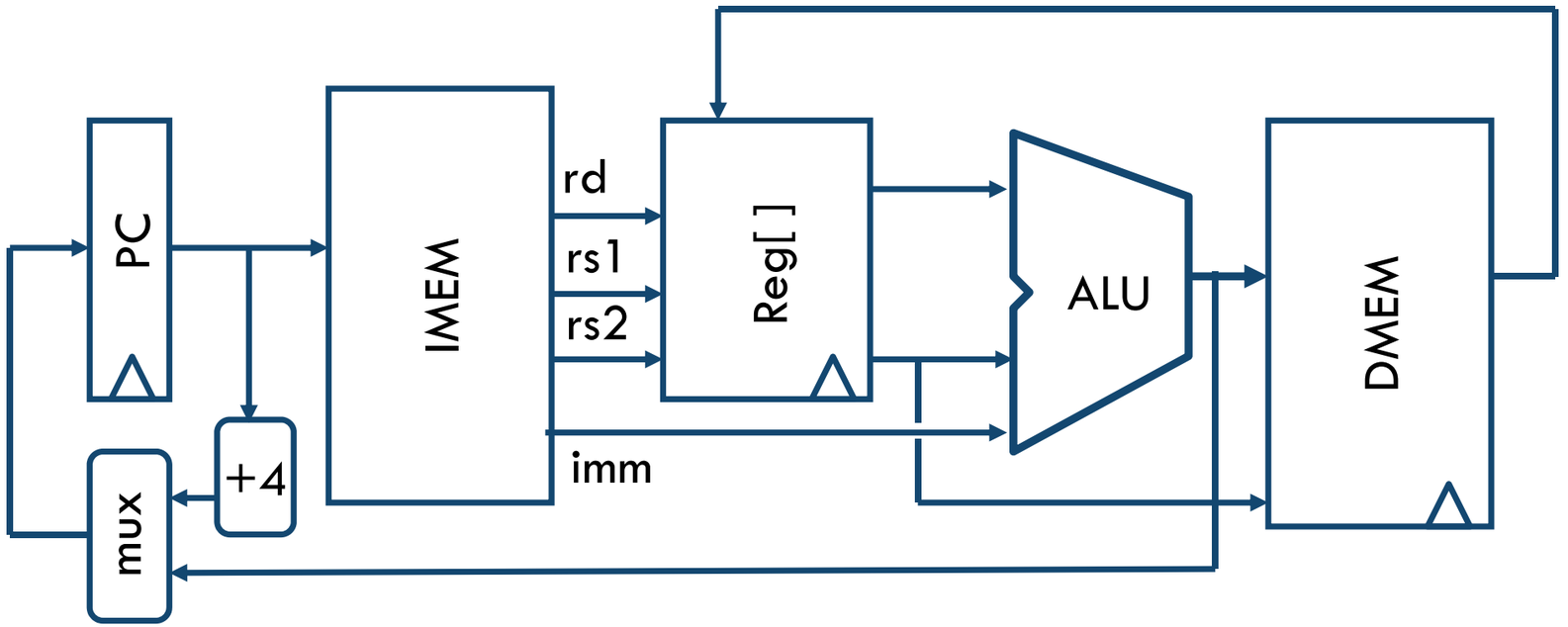
Stages of the Datapath : Overview

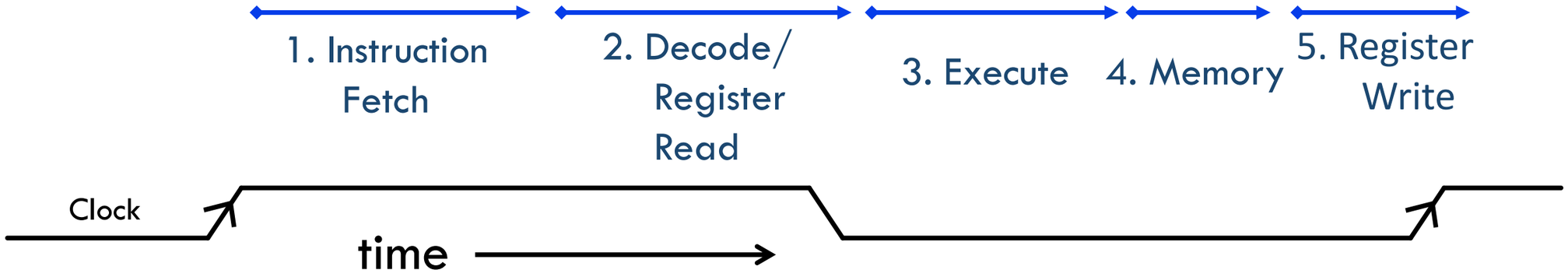
- Problem: A single, "monolithic" CL block that "executes an instruction" (performs all necessary operations beginning with fetching the instruction and completing with the register access) is be too bulky and inefficient
- Solution: Break up the process of "executing an instruction" into stages, and 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)*

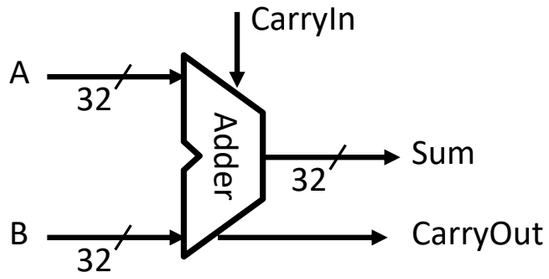
Basic Phases of Instruction Execution



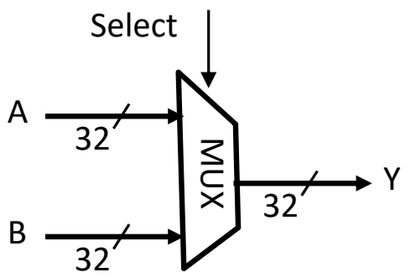


Datapath Components: Combinational

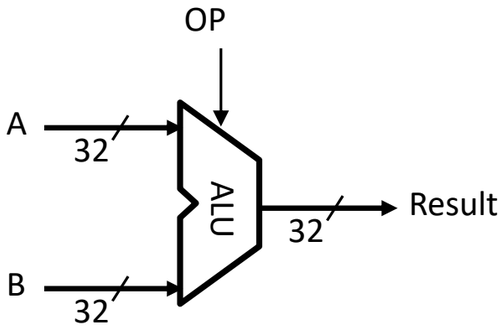
- Combinational Elements



Adder



Multiplexer

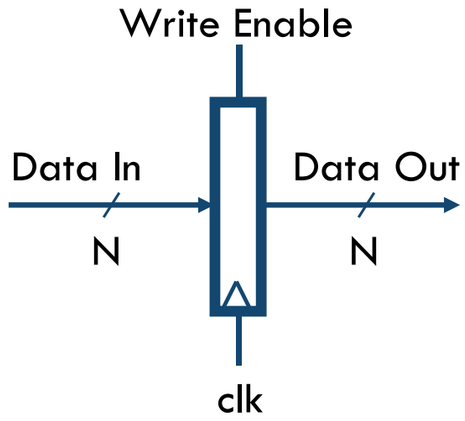


ALU

- Storage Elements + Clocking Methodology
- Building Blocks

Datapath Elements: State and Sequencing (1/4)

- Register

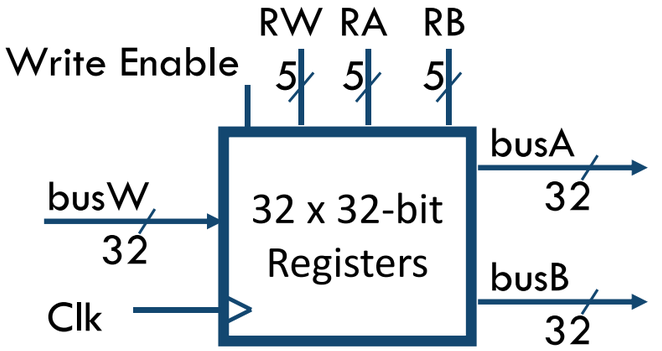



```
always @(posedge clk)
    if (wen) dataout <= datain;
endmodule
```

- Write Enable:
  - Negated (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 (2/4)

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
  - x0 is wired to 0



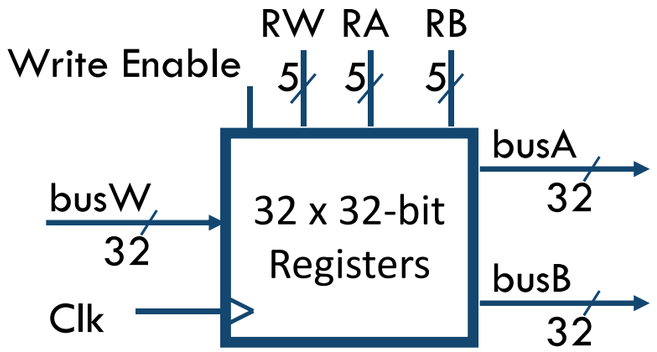
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1

- 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."

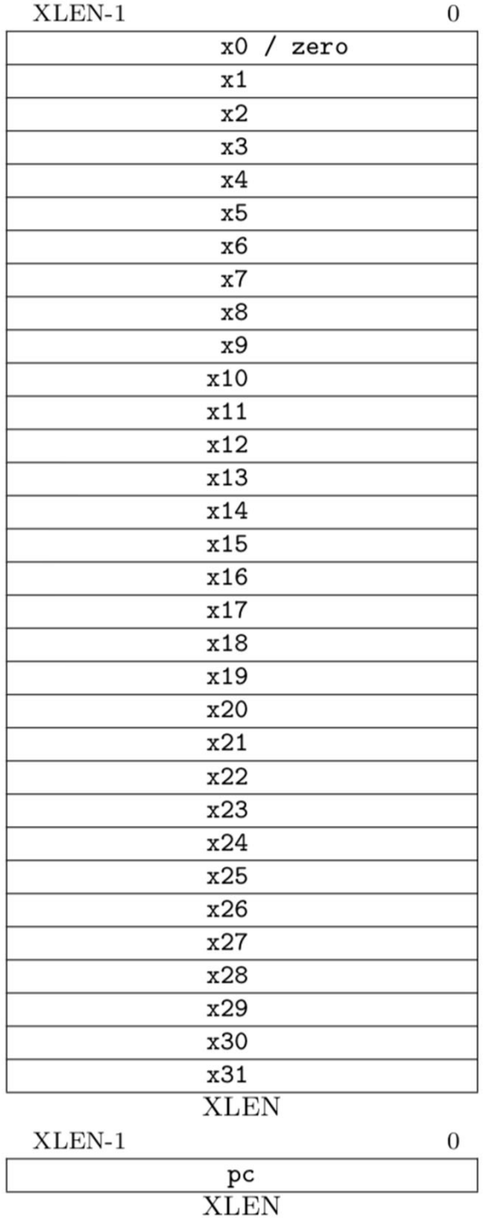
Datapath Elements: State and Sequencing (3/4)

- Reg file in Verilog

```
module rv32i_regs (
    input clk, wen,
    input [5:0] rw,
    input [5:0] ra,
    input [5:0] rb,
    input [31:0] busw,
    output [31:0] busa,
    output [31:0] busb
);
    reg [31:0] regs [0:30];
    always @(posedge clk)
        if (wen) regs[~rw[4:0]] <= busw;
    assign busa = regs[~ra[4:0]];
    assign busb = regs[~rb[4:0]];
endmodule
```

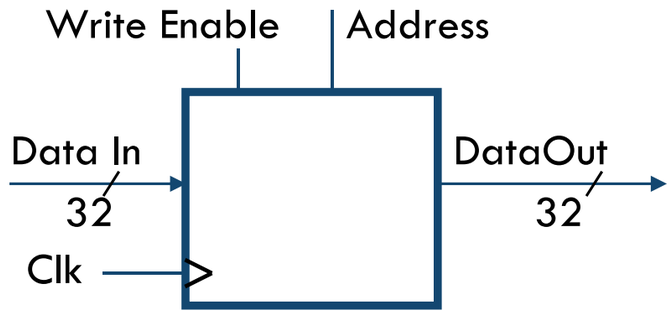


- How does RV64I register file look like?



Datapath Elements: State and Sequencing (4/4)

- "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 ⇒ Data Out valid after "access time"
- Real memory later in the class



Review: Complete RV32I ISA

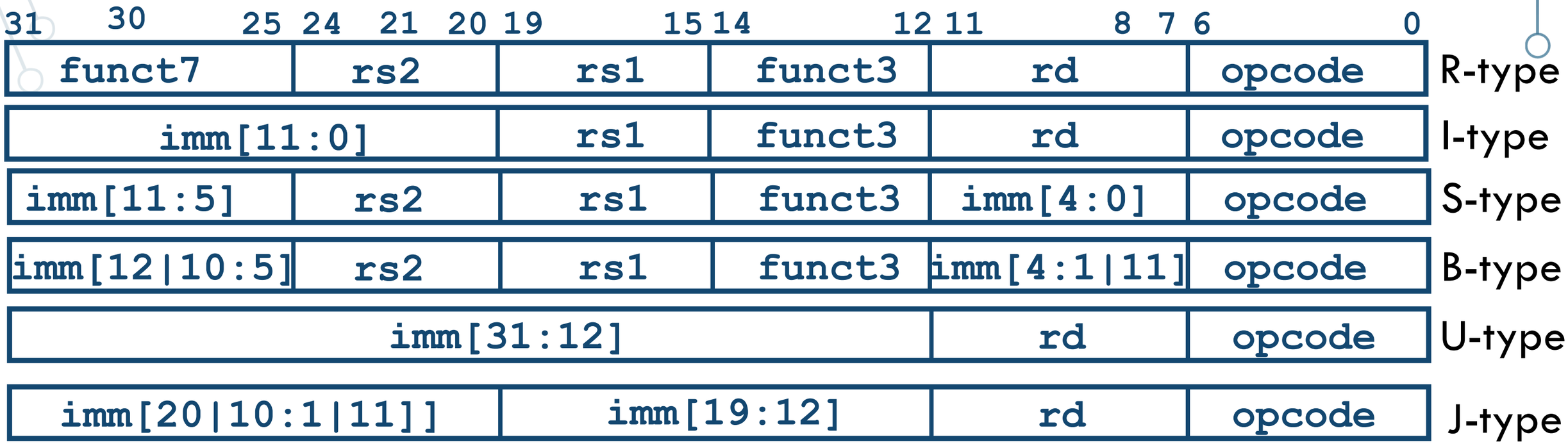
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
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
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREA
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRW
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

CS61C
(added in EECS151)

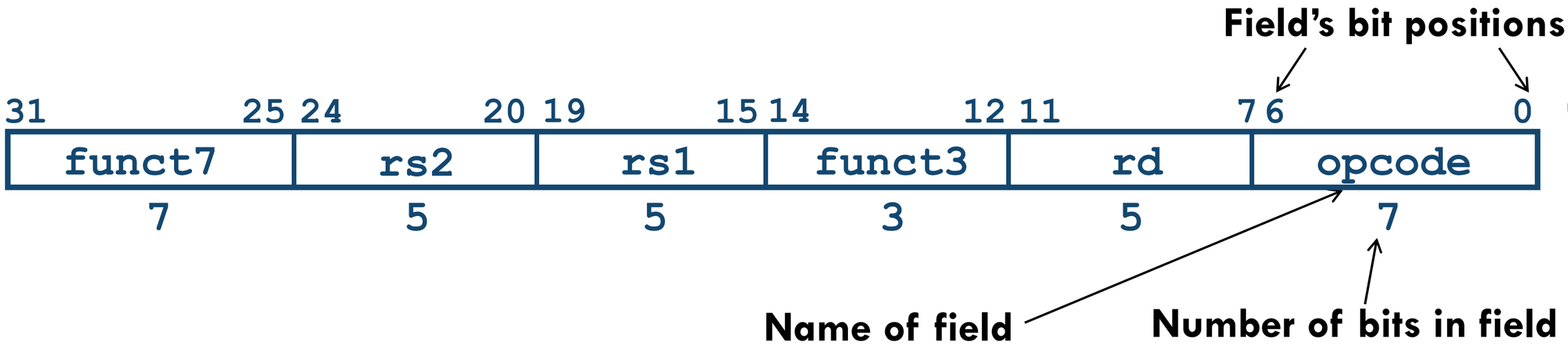
- Need datapath and control to implement these instructions

Summary of RISC-V Instruction Formats



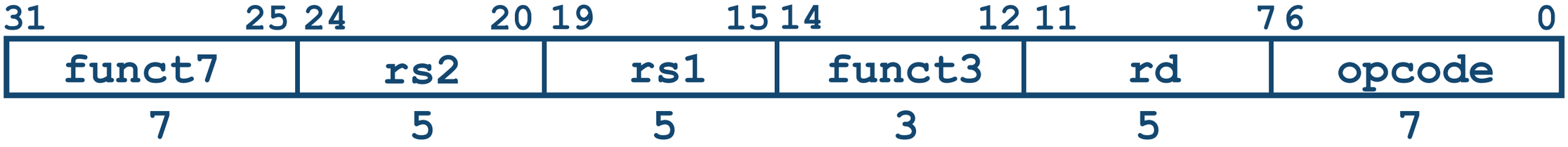
R-Format Instructions: Datapath

R-Format Instruction Layout



- 32-bit instruction word divided into six fields of varying numbers of bits each: $7+5+5+3+5+7 = 32$
- Examples
  - opcode is a 7-bit field that lives in bits 6-0 of the instruction
  - rs2 is a 5-bit field that lives in bits 24-20 of the instruction

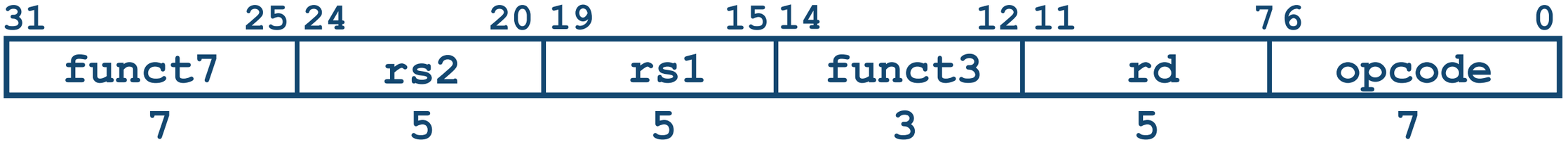
R-Format Instructions opcode/funct fields



- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to 0110011_{two} for all R-Format register-register arithmetic instructions
- **funct7+funct3**: combined with **opcode**, these two fields describe what operation to perform

- **Question: You have been professing simplicity, so why aren't opcode and funct7 and funct3 a single 17-bit field?**
  - **We'll answer this later**

R-Format Instructions register specifiers

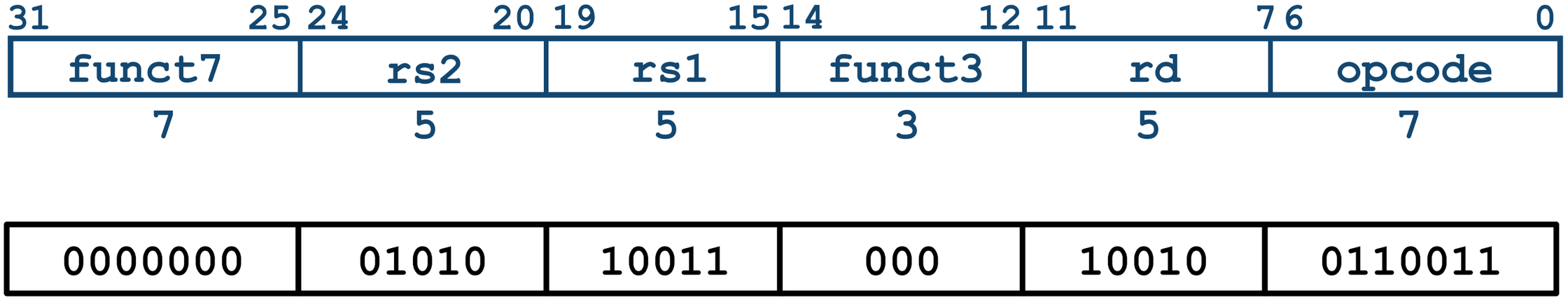


- **`rs1`** (Source Register #1): specifies register containing first operand
- **`rs2`** : specifies second register operand
- **`rd`** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**`x0-x31`**)

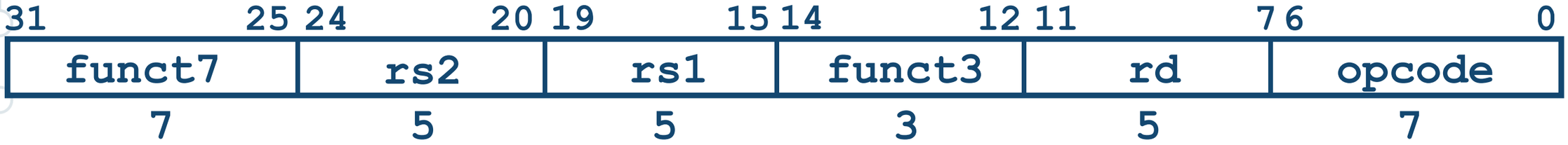
R-Format Example

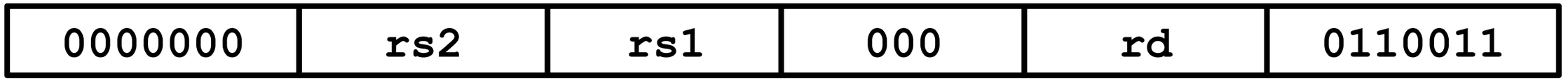
- RISC-V Assembly Instruction:

```
add   x18,x19,x10
```



Implementing the **add** instruction





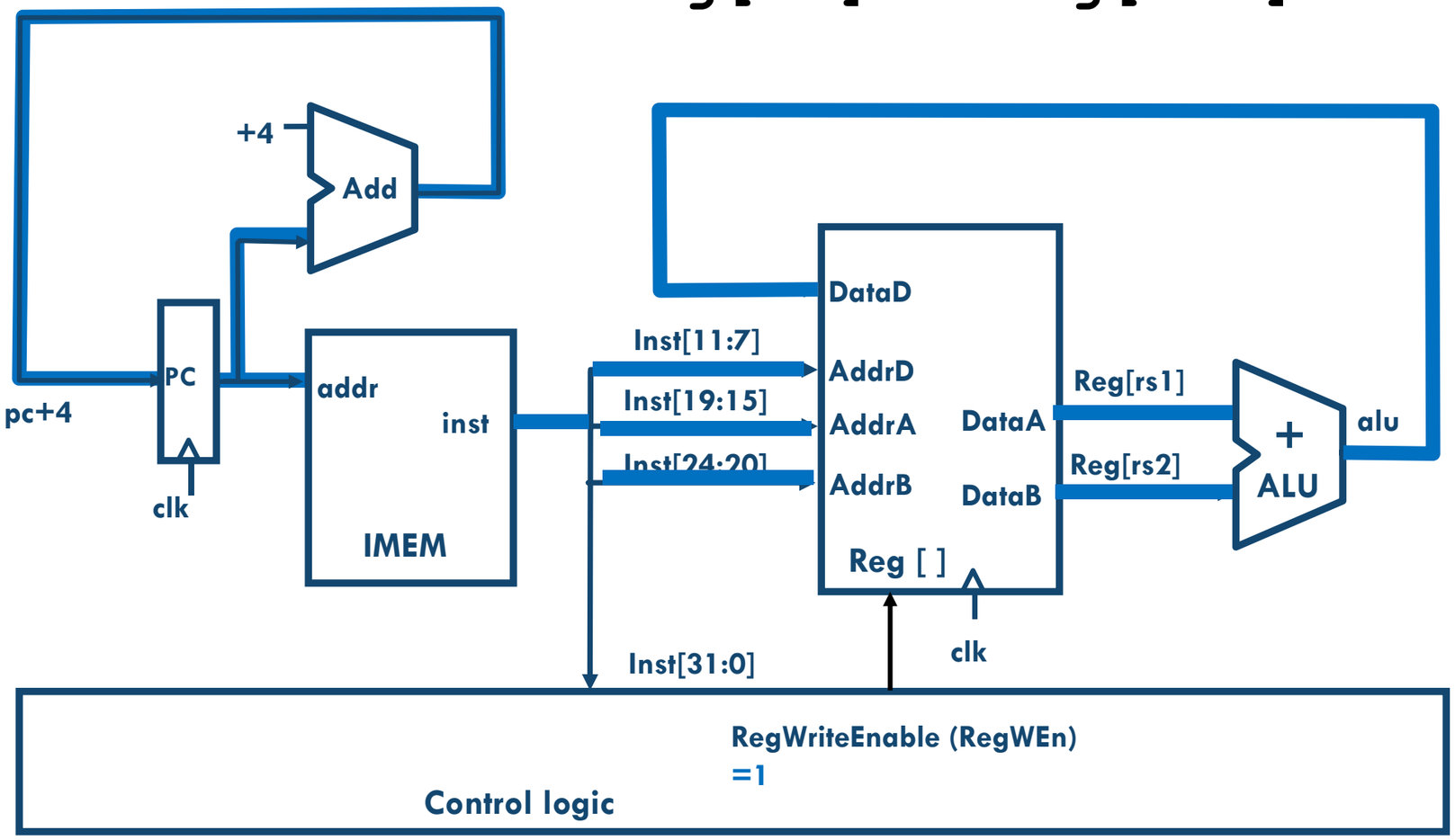
add rs2 rs1 add rd Reg-Reg OP

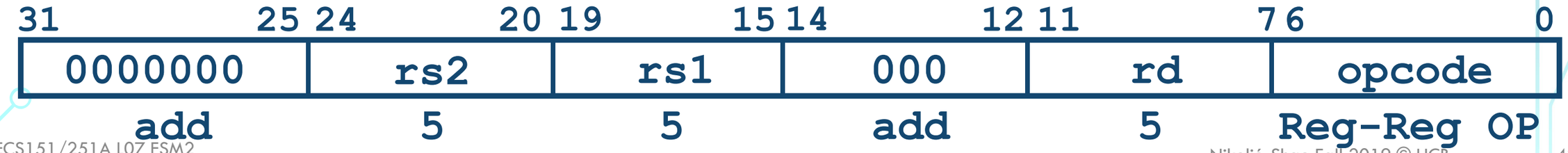
add rd, rs1, rs2

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

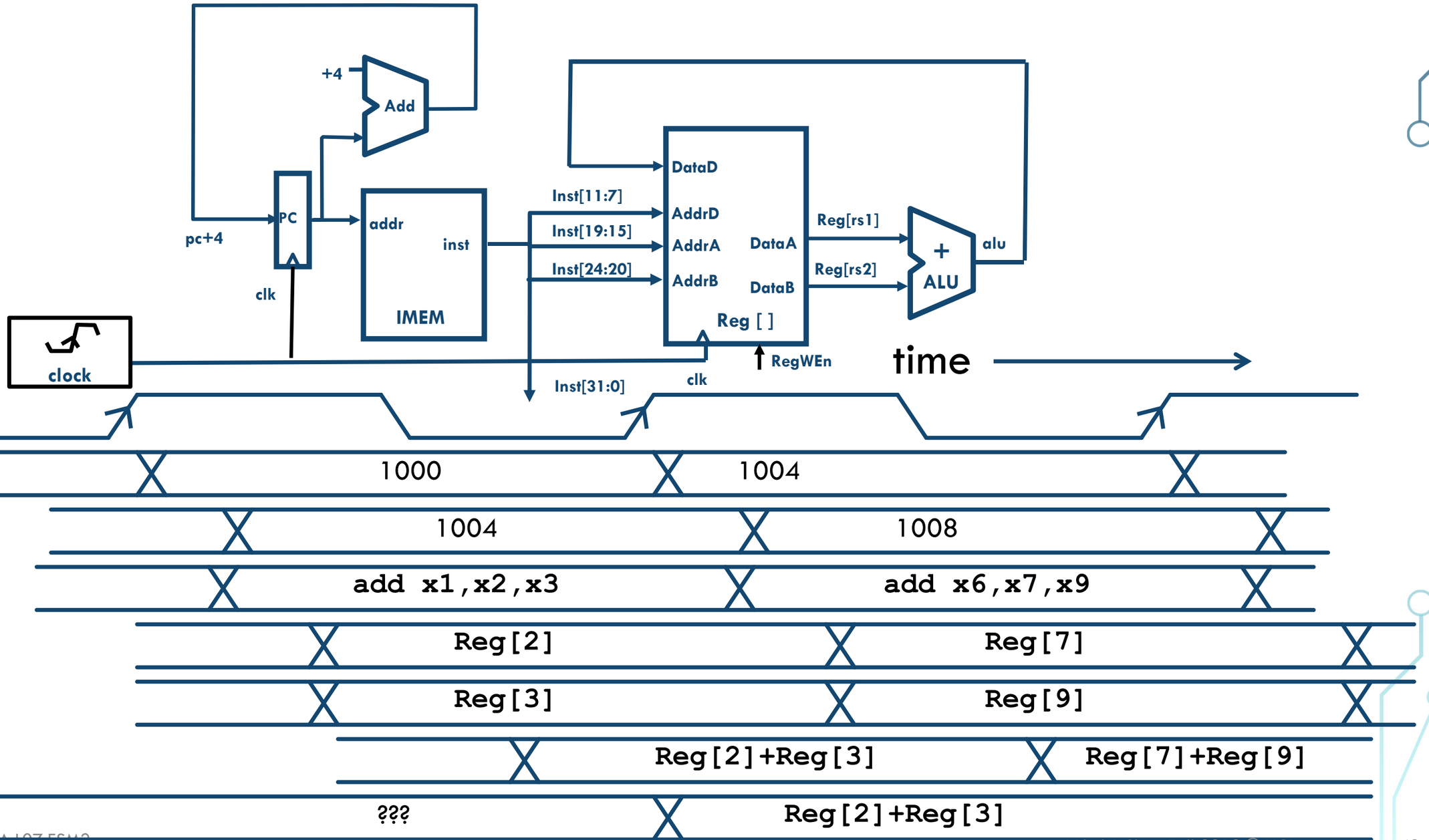
Datapath for add

```
PC = PC + 4    Reg[rd] = Reg[rs1] + Reg[rs2]
```

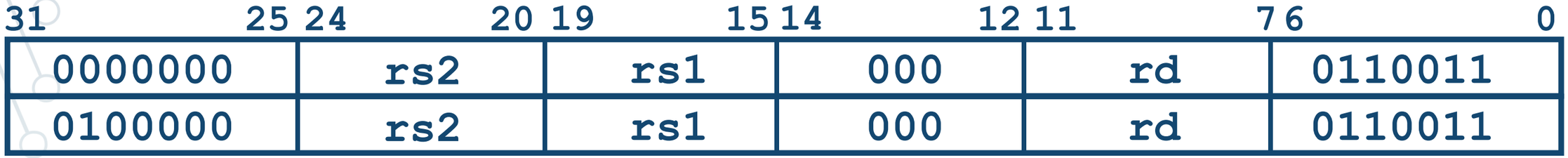




Timing Diagram for **add**



Implementing the `sub` instruction

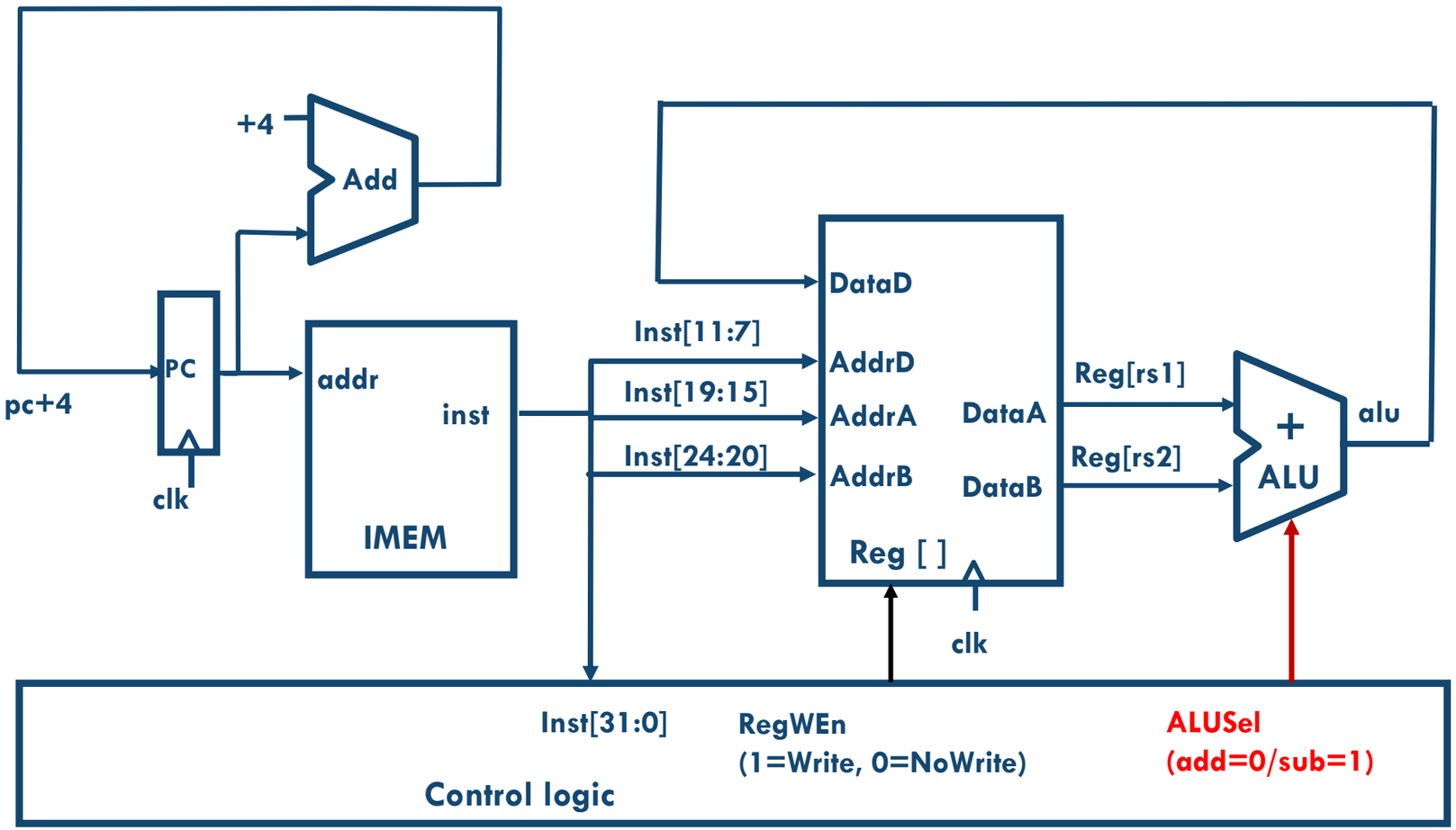


31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

`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`



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

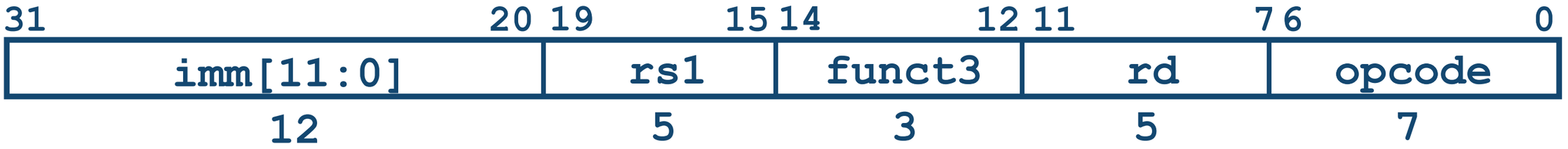
Summary

- State machines:
  - Specify circuit function
  - Draw state transition diagram
  - Write down symbolic state-transition table
  - Assign encodings (bit patterns) to symbolic states
  - Code as Verilog behavioral description
- RISC-V processor
  - A large state machine
  - Datapath + control
  - Reviewed R-format instructions

Implementing I-Format - `addi` instruction

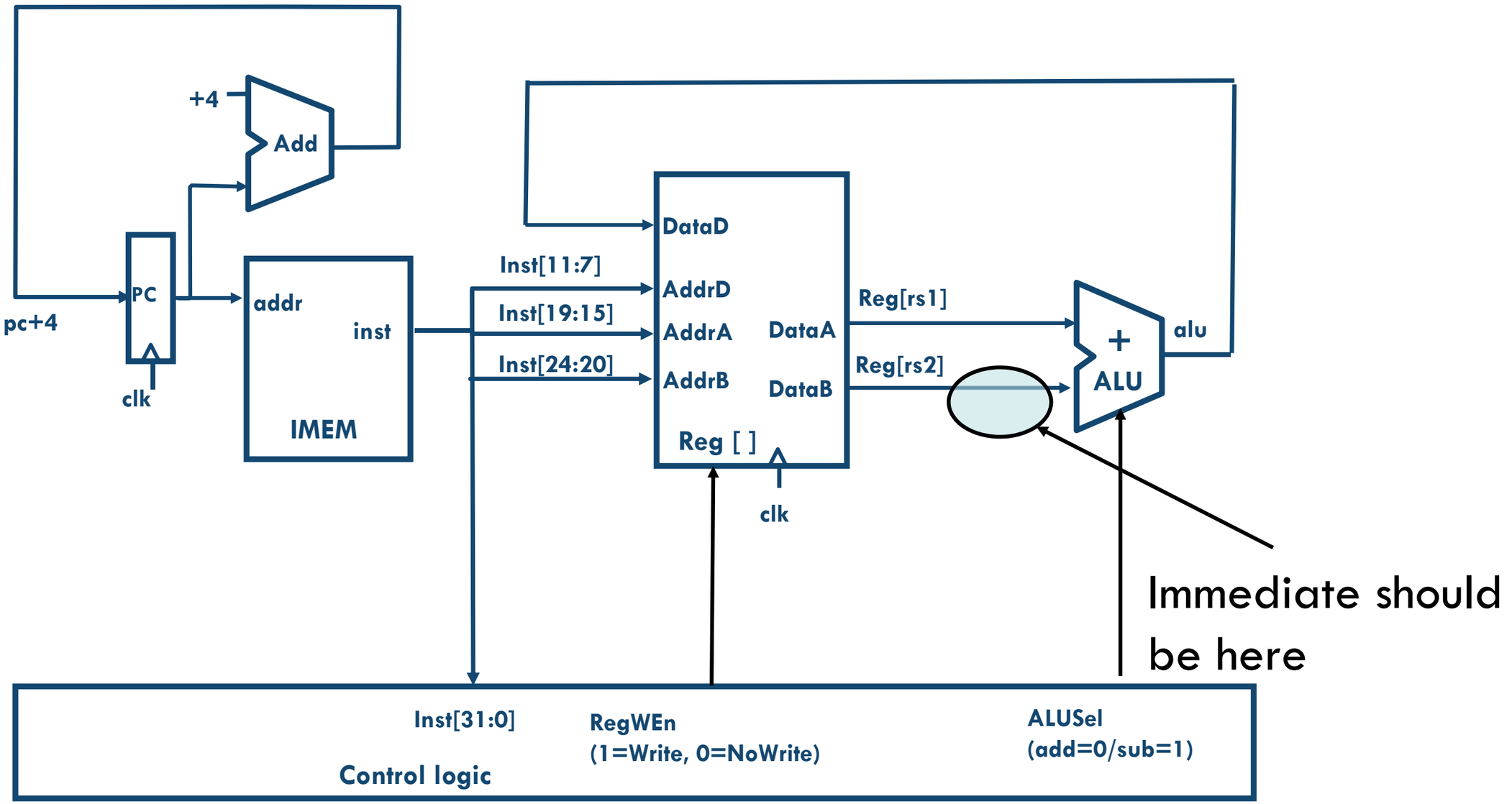
- **RISC-V Assembly Instruction:**

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

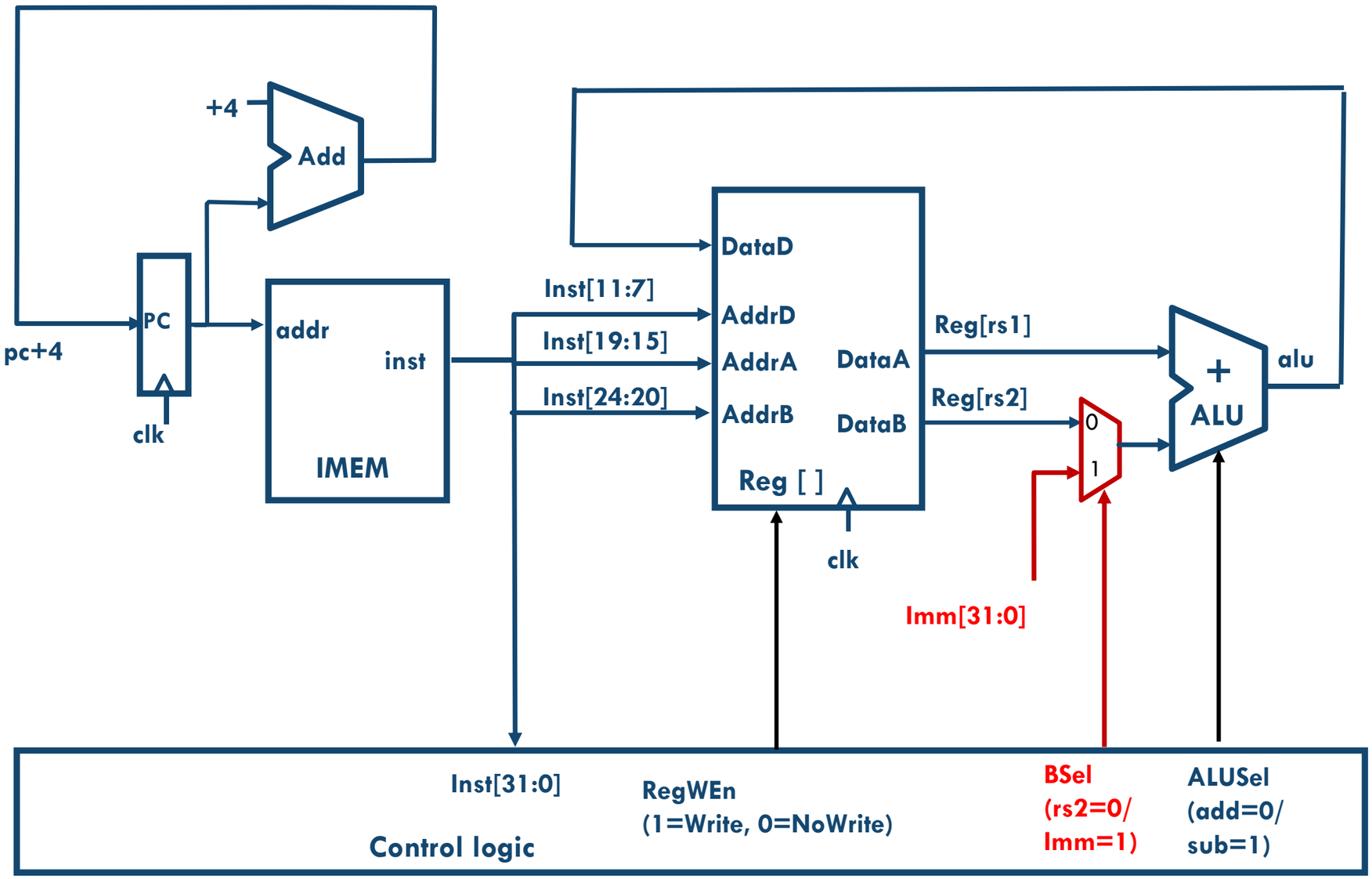


111111001110	00001	000	01111	0010011
imm=-50	rs1=1	add	rd=15	OP-Imm

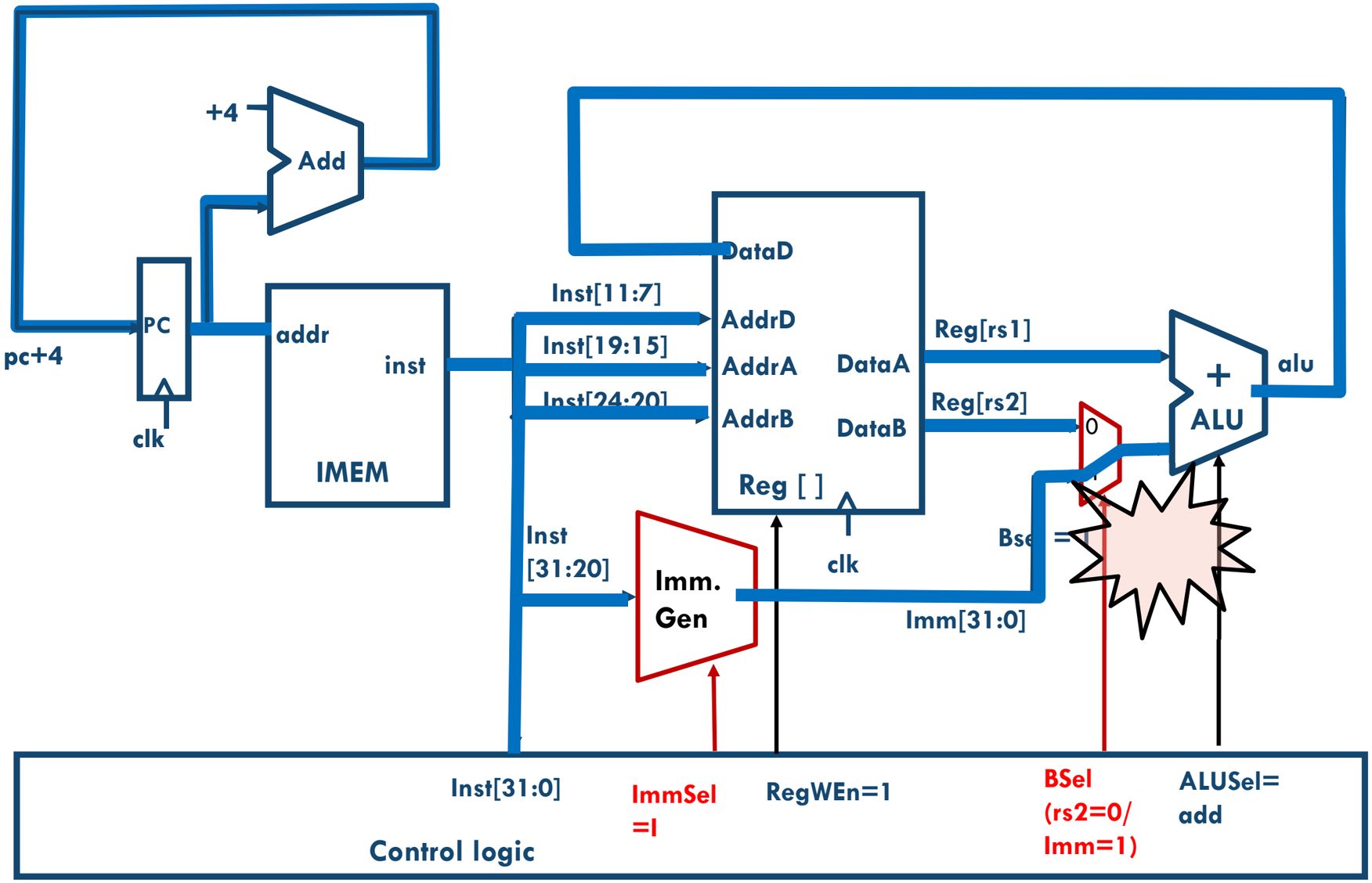
Datapath for `add/sub`



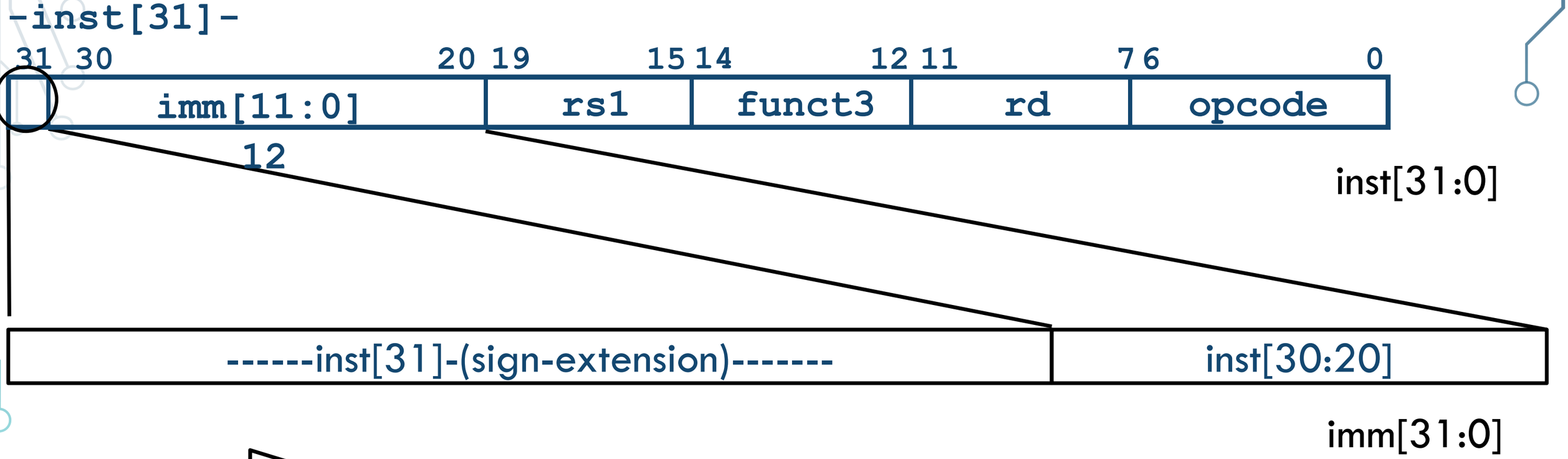
Adding `addi` to Datapath

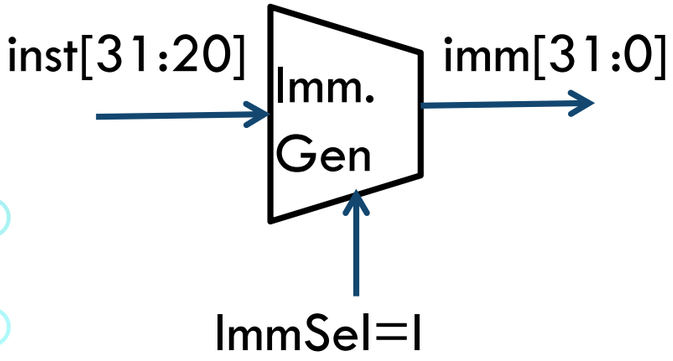


Adding `addi` to Datapath



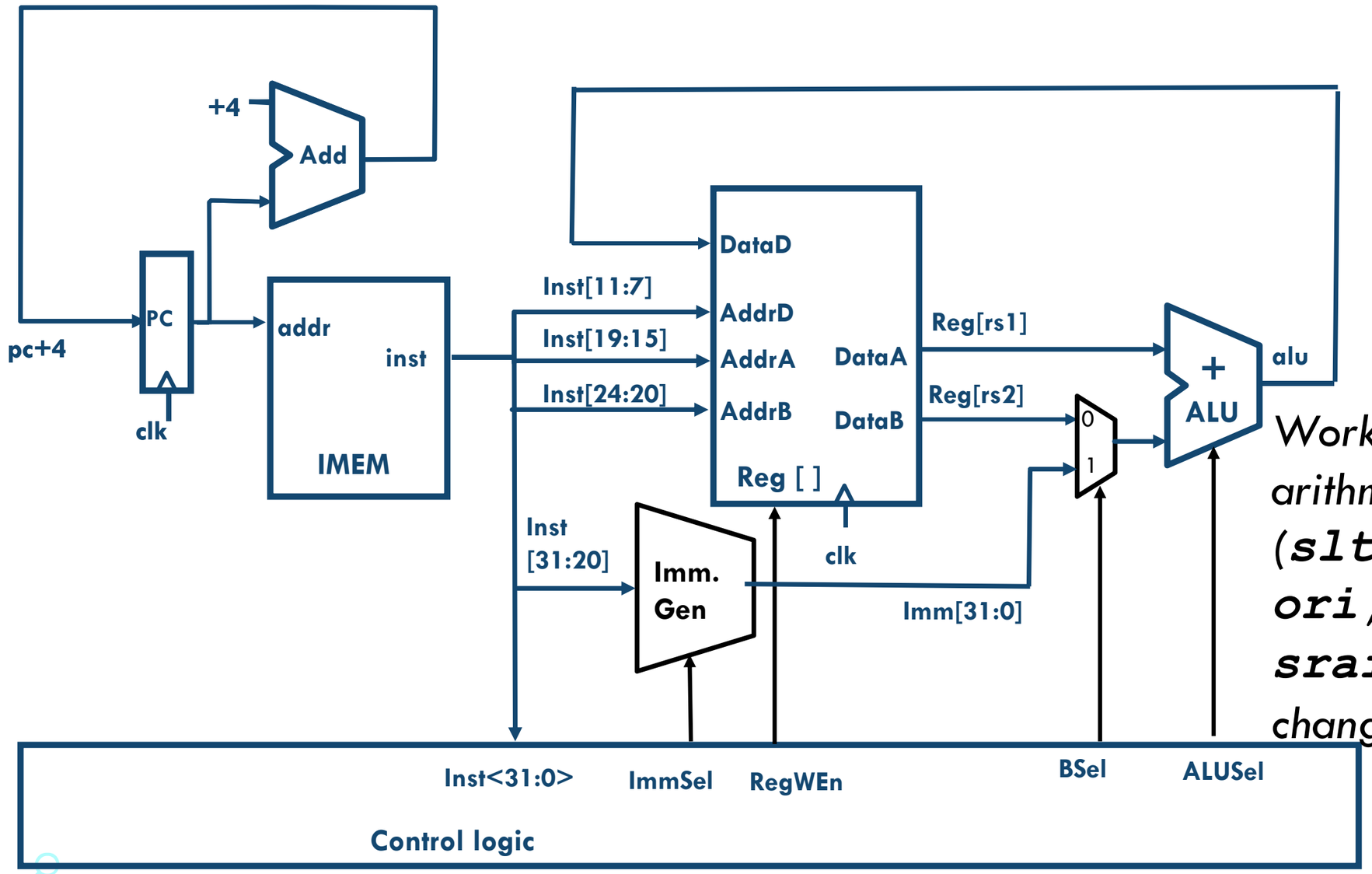
I-Format immediates





- **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])**

R+I Datapath



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

Peer Instruction (five growth)

1) We should use the main ALU to compute PC=PC+4 in order to save some gates
2) The ALU is a synchronous state element
3) Program counter is a register

123
FFF
FFT
FTF
FTT
TFF
TFT
TTF
TTT

Summary

- Sequential logic:
  - Memory: the outputs depend on both current and previous values of the inputs.
- Finite State Machine:
  - Registers to store current states
  - Combinational logic:
    - Compute the next state
    - Compute the outputs
- Moore vs Mealy FSM:
  - Moore: Outputs depend only on current state
  - Mealy: Outputs depend on current state and inputs