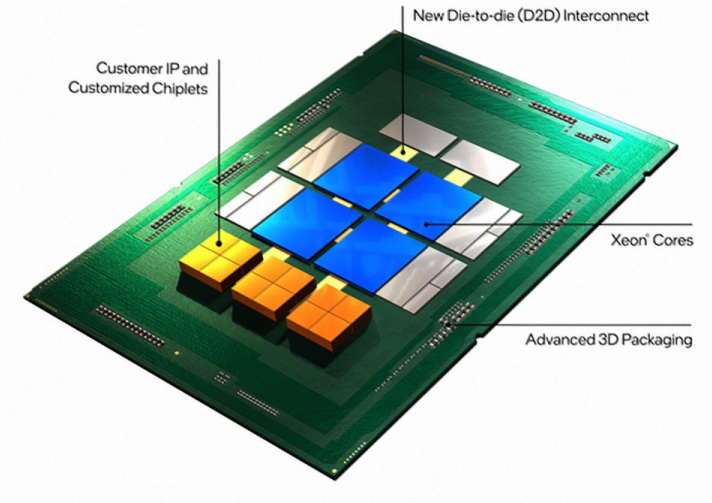
EECS151/251A
Introduction to Digital Design and ICs

Lecture 7: FSM & RISC-V Intro Sophia Shao

Intel's plan to license x86 cores for chips with Arm, RISC-V and more inside

Intel will license its most important asset, the x86 architecture, to those who want to make custom silicon. Depending on the application, customers will be able to mix up x86, Arm and RISC-V CPU cores as well as hardware acceleration units in a custom-designed chip that Intel fabricates.

"We have what we call a multi ISA strategy. That's the first time in Intel's history we'll license x86 soft cores and hard cores to customers who would like to develop chips," Bob Brennan, vice president of customer solutions engineering at Intel's Foundry Services, told *The Register*.



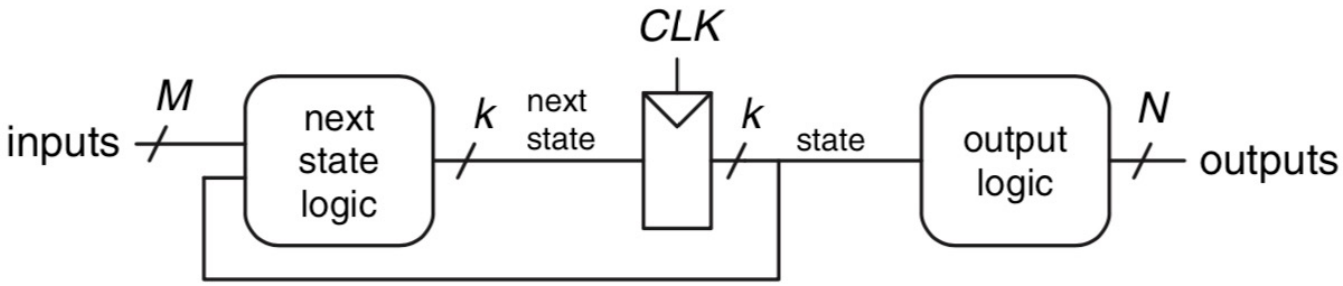
https://www.theregister.com/2022/02/14/intel_x86_licensing/

- **Finite State Machine**
  - Introduction
  - **Moore vs Mealy FSM**
  - **FSM in Verilog**
- **RISC-V**
  - **Introduction**
  - **Datapath Elements**

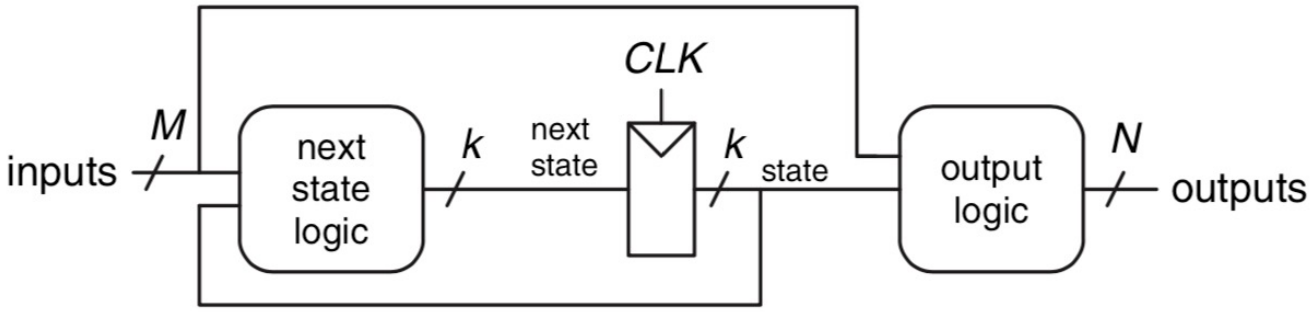
Moore vs Mealy FSMs

- 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

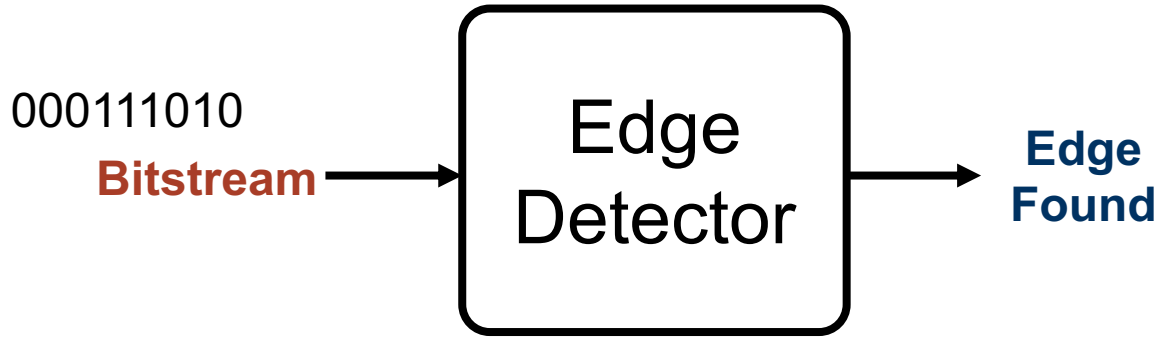


Mealy FSM

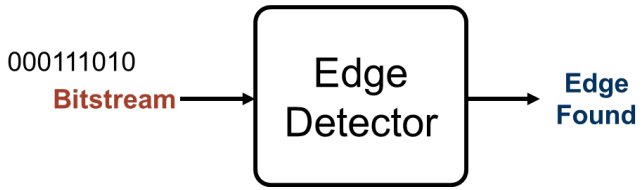


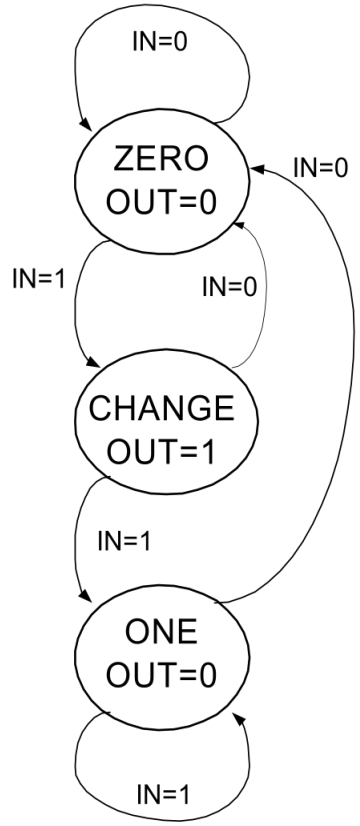
Example: Edge Detector

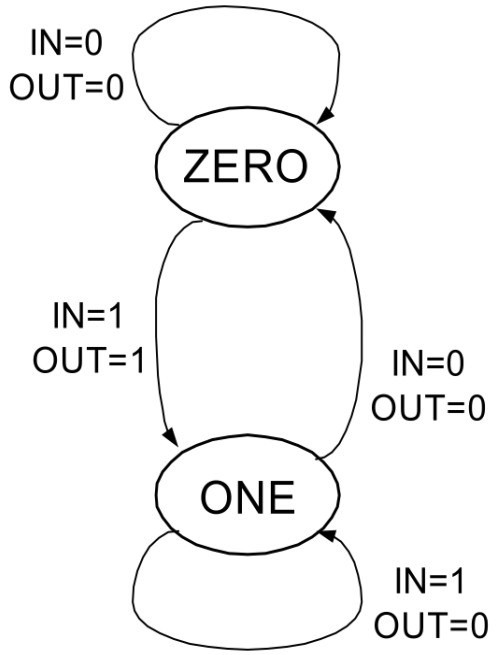
- Input:
  - A bit stream that is received one bit at a time.
- Output:
  - 0/1
- Circuit:
  - Asserts its output to be true when the input bit stream changes from 0 to 1.

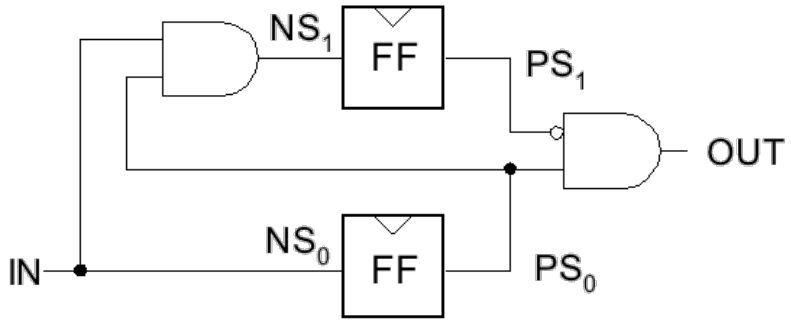


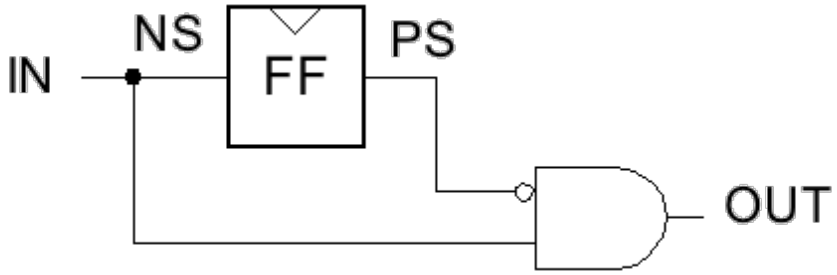
Moore vs Mealy



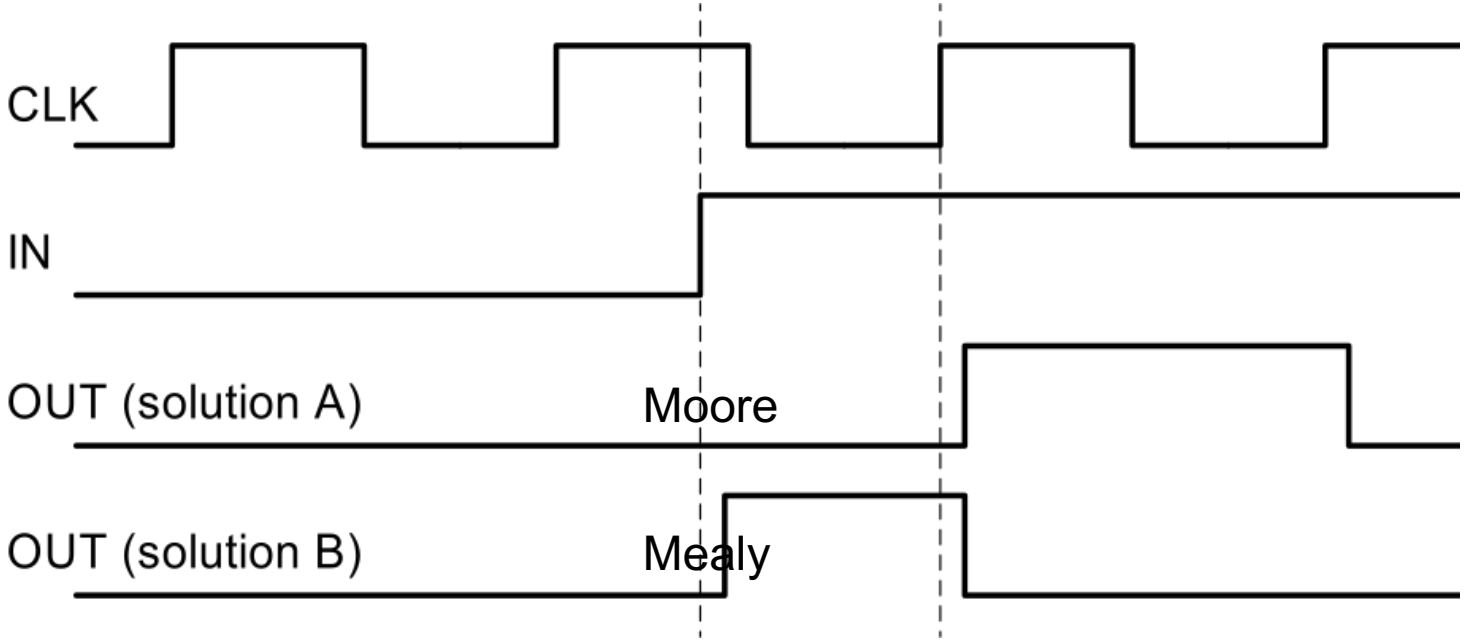


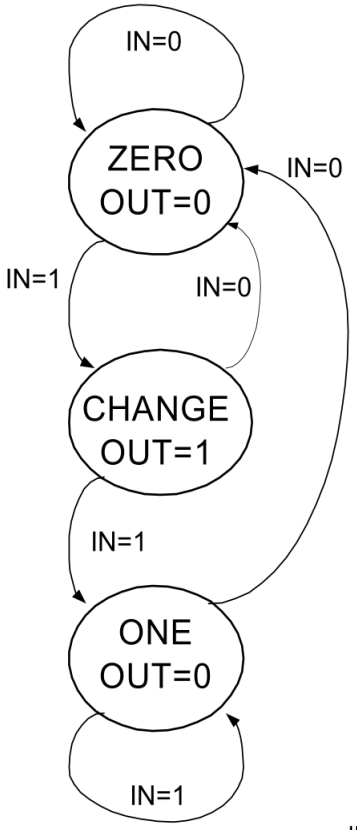


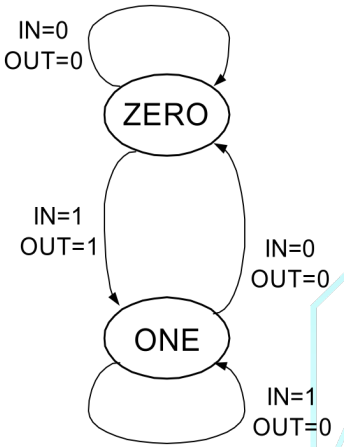




Edge Detection Timing Diagrams







- 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 falls synchronous with next clock edge

FSM Comparison

Solution A

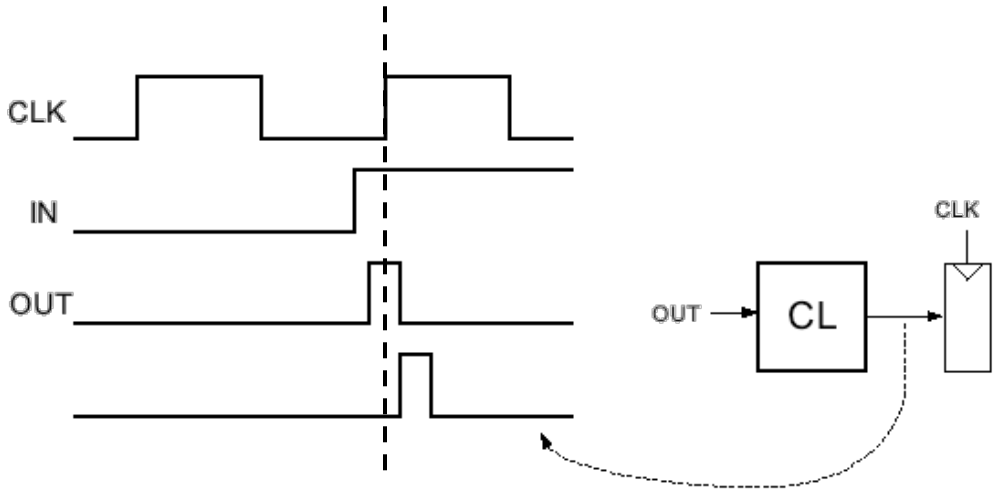
Moore Machine

- 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

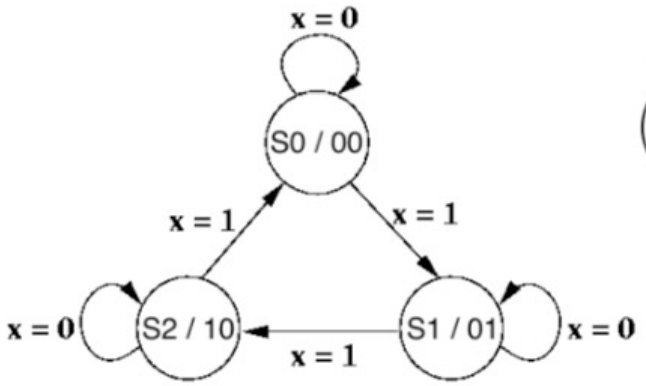
Mealy Machine

- 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):

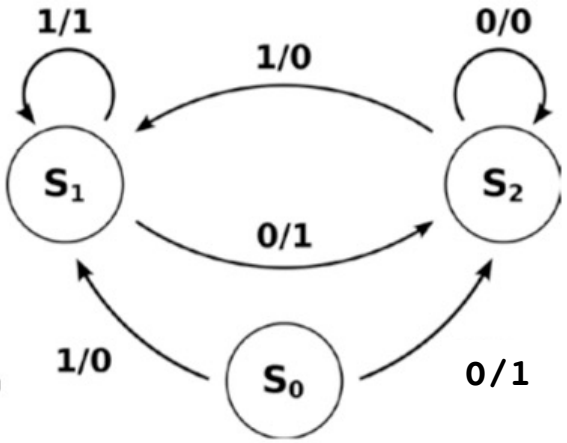


If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate set-up time requirement)

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

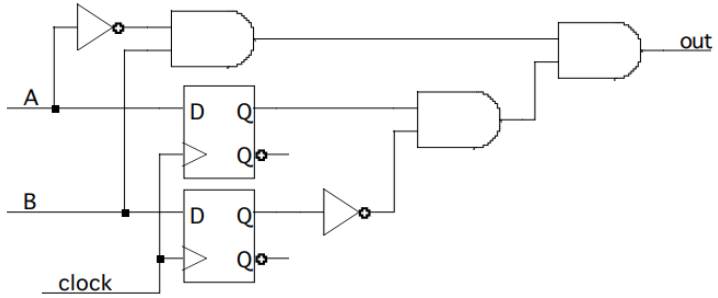


A.

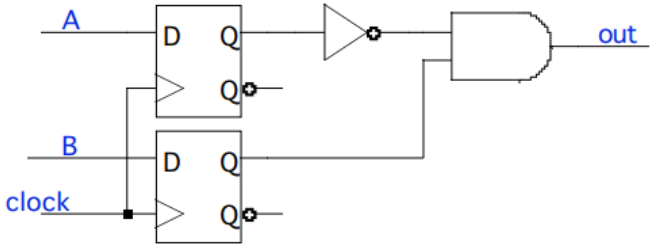


B.

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



C.



D.

- **Finite State Machine**
  - Introduction
  - **Moore vs Mealy FSM**
  - **FSM in Verilog**
- **RISC-V**
  - **Introduction**
  - **Datapath Elements**

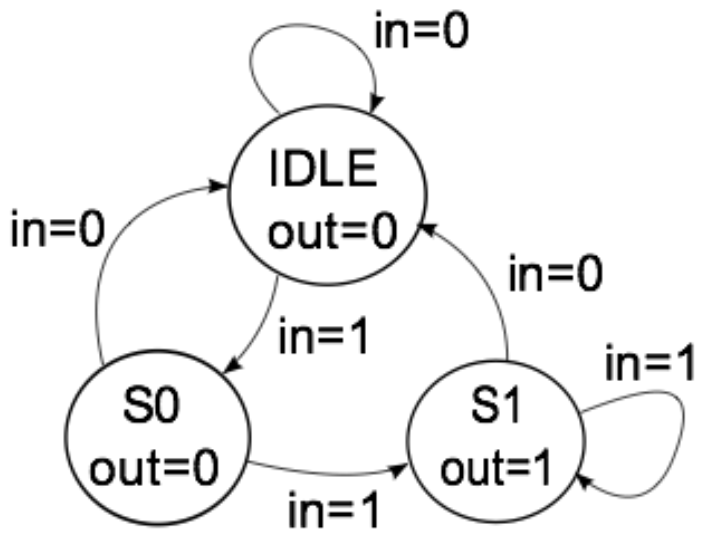
Implement FSM with Verilog

- Specify circuit function
- Draw state transition diagram
- Write down symbolic state transition table
- Assign encodings (bit patterns) to symbolic states

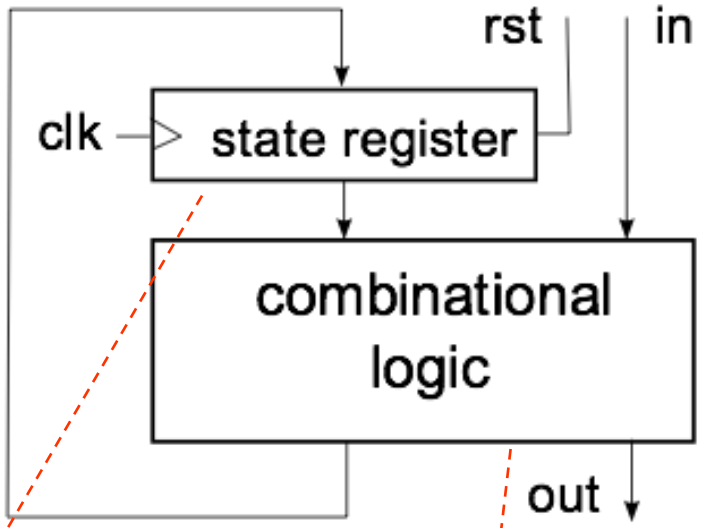
- Code as Verilog behavioral description
  - Use parameters to represent encoded states
  - Use separate always blocks for register assignment and combinational logic block
  - Use case statement for combinational logic.
    - Within each case section (state), assign outputs and next state based on inputs
    - Moore: outputs only dependent on states not on inputs

Finite State Machine in Verilog

State Transition Diagram



Circuit Diagram



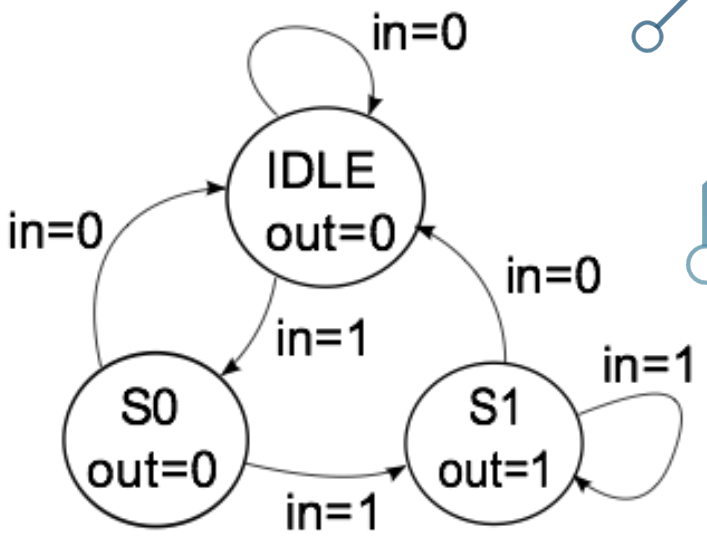
Holds a symbol to keep track of which bubble the FSM is in.

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.

Constants local to this module.

out not a register, but assigned in always block

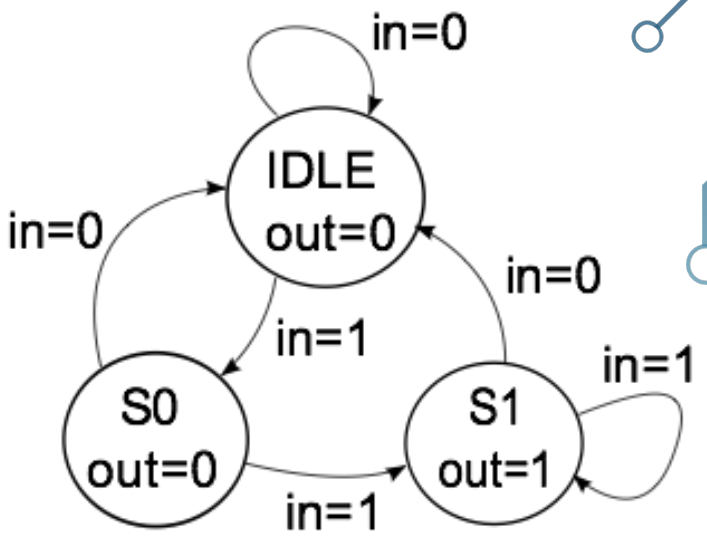
Combinational logic signals for transition.

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 @(*)
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)

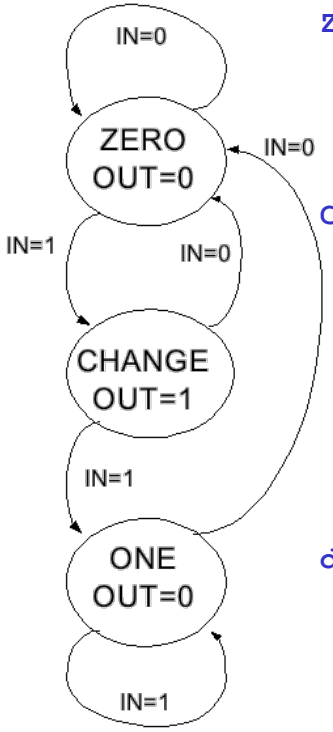
State transition

Moore or Mealy?

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

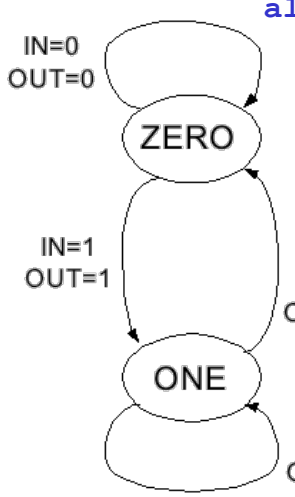
Edge Detector Example

Moore Machine




```
always @(posedge clk)
     if (rst) cs <= ZERO;
     else cs <= ns;
always @(*)
    case (cs)
      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
```

Mealy Machine




```
always @(posedge clk)
     if (rst) cs <= ZERO;
     else cs <= ns;
always @(*)
    case (cs)
      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
```

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

Administrivia

- Mega-thread for each lab/HW on Ed.
  - Share your understanding
- Homework 2 due this week.
  - Homework 3 out.

- **Finite State Machine**
  - Introduction
  - **Moore vs Mealy FSM**
  - **FSM in Verilog**
- **RISC-V**
  - **Introduction**
  - **Datapath Elements**

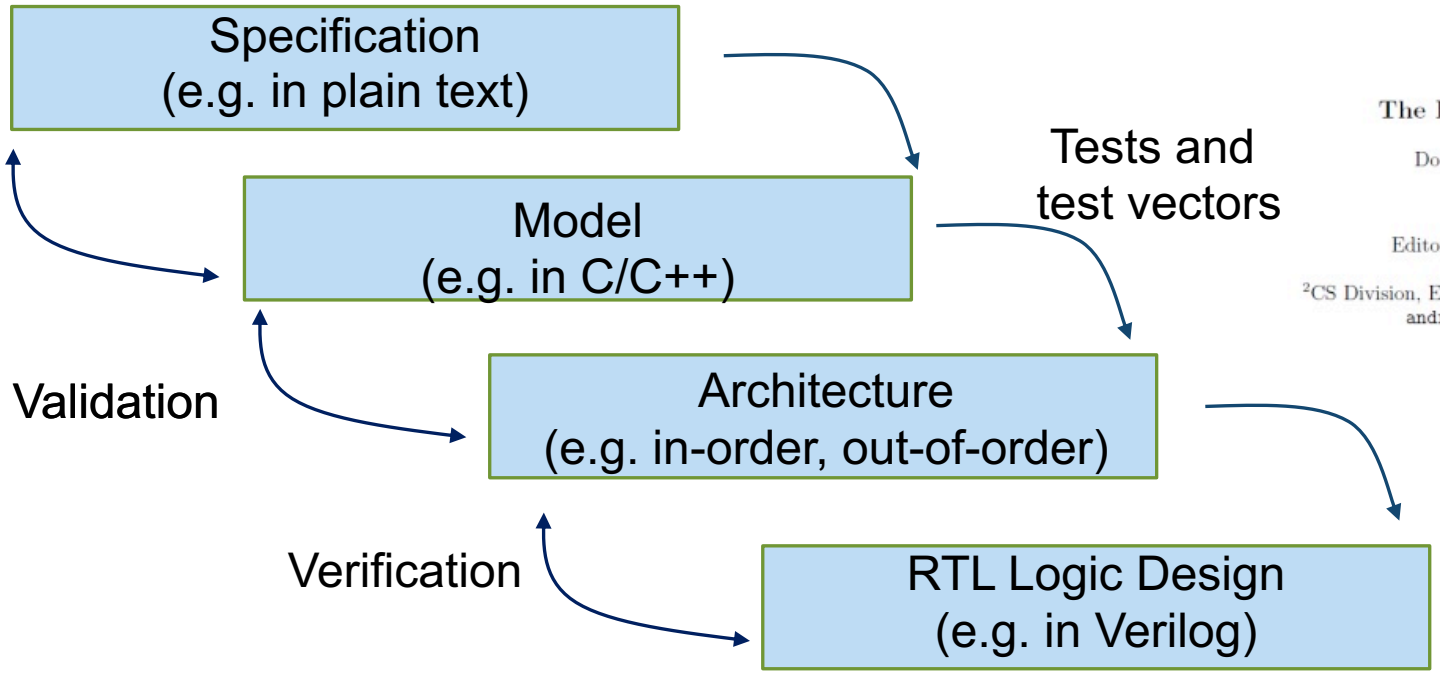
Berkeley RISC-V ISA

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)

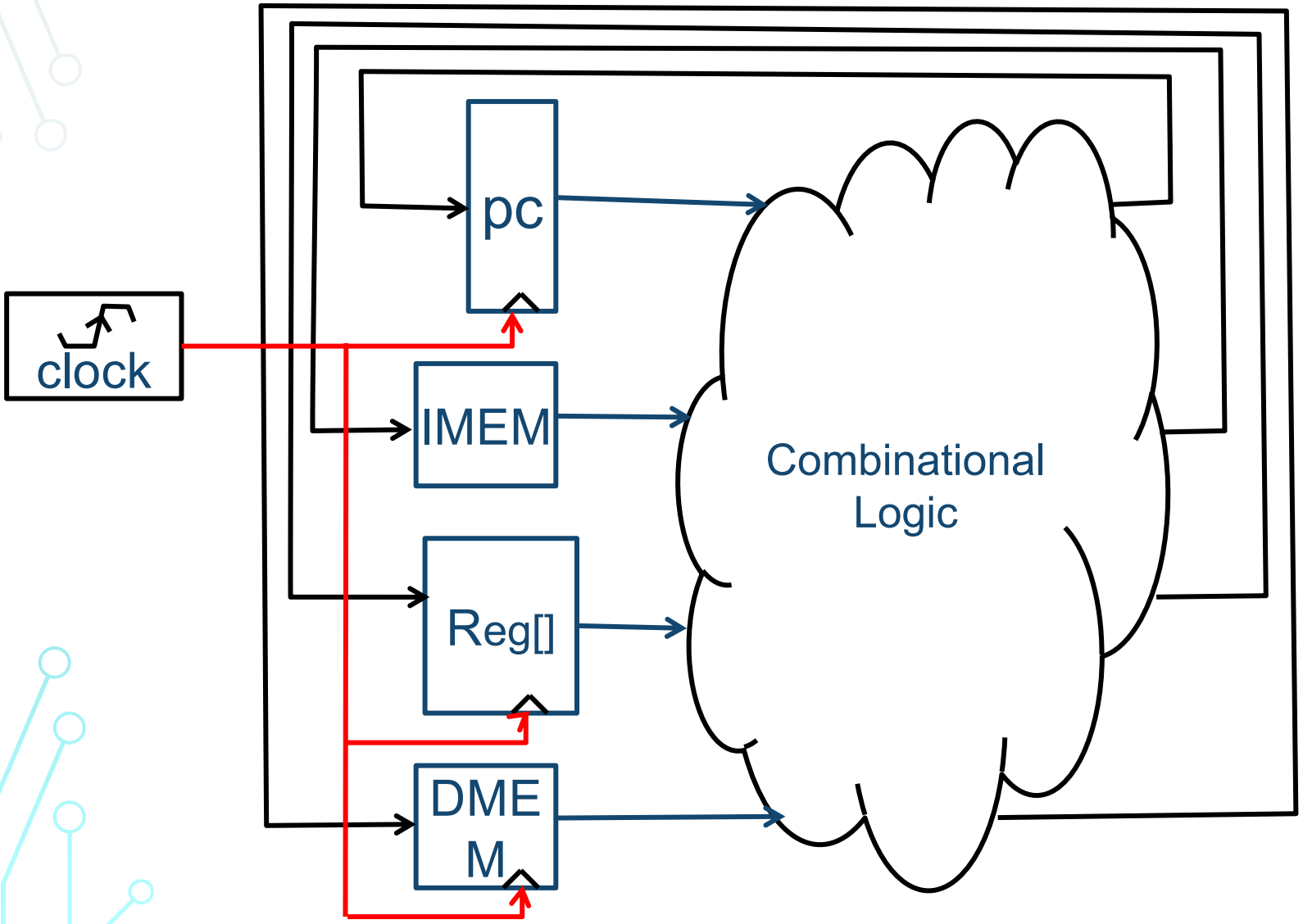


The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20190608-Base-Ratified

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

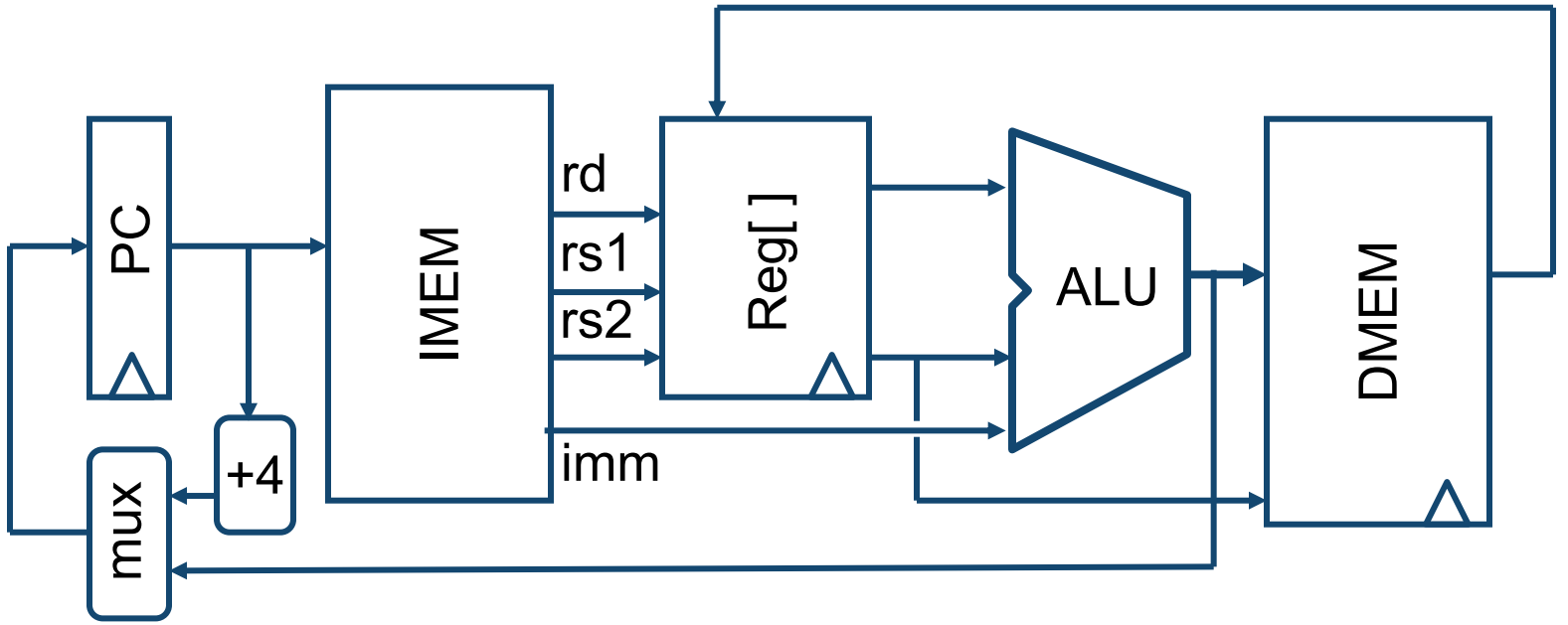
Stages of the Datapath : Overview

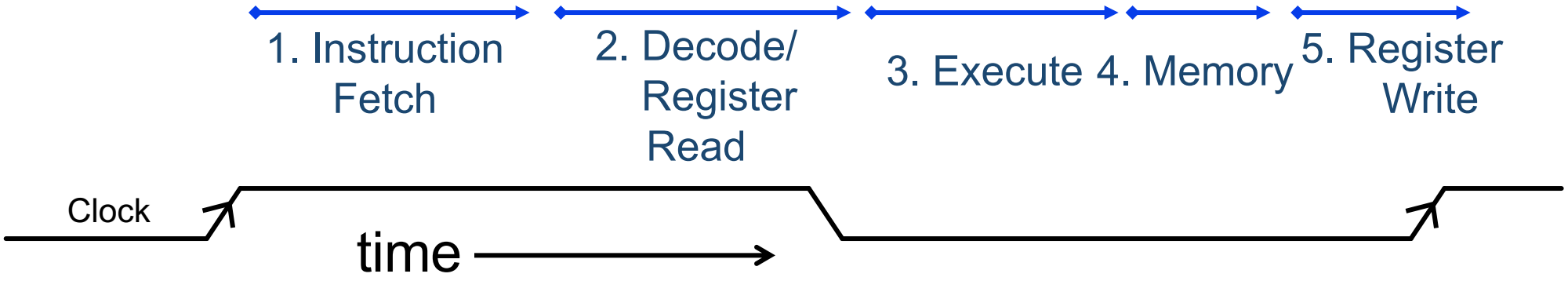
- **Problem:**
  - A single, “monolithic” CL block that “executes an instruction” (performs all necessary operations beginning with fetching the instruction and completing with the register access) is be too bulky and inefficient
- **Solution:**
  - Break up the process of “executing an instruction”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

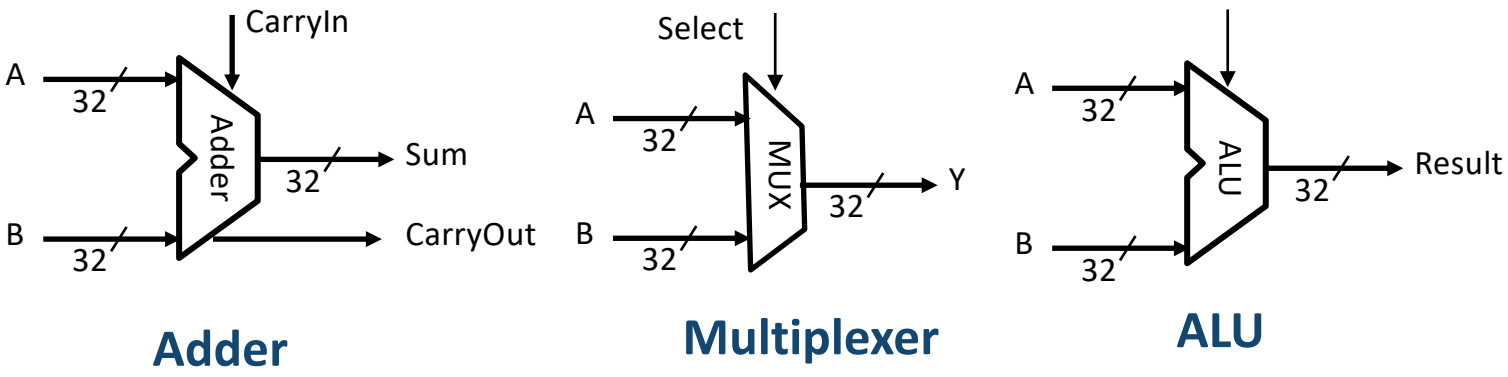
Basic Phases of Instruction Execution





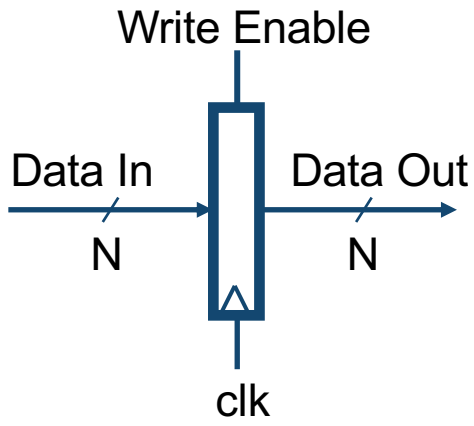
Datapath Components: Combinational

- Combinational Elements



Datapath Elements: State and Sequencing (1/4)

- Register

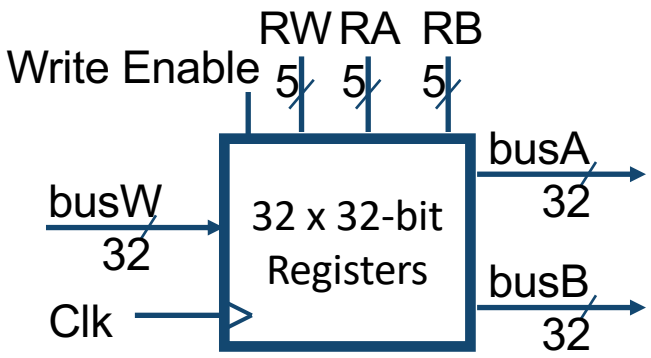



```
always @(posedge clk)
    if (wen) dataout <= datain;
endmodule
```

- 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

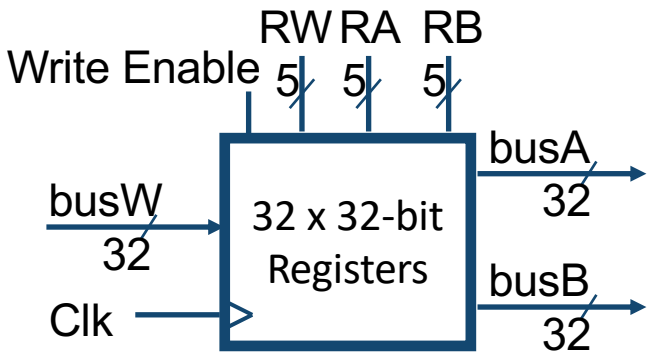


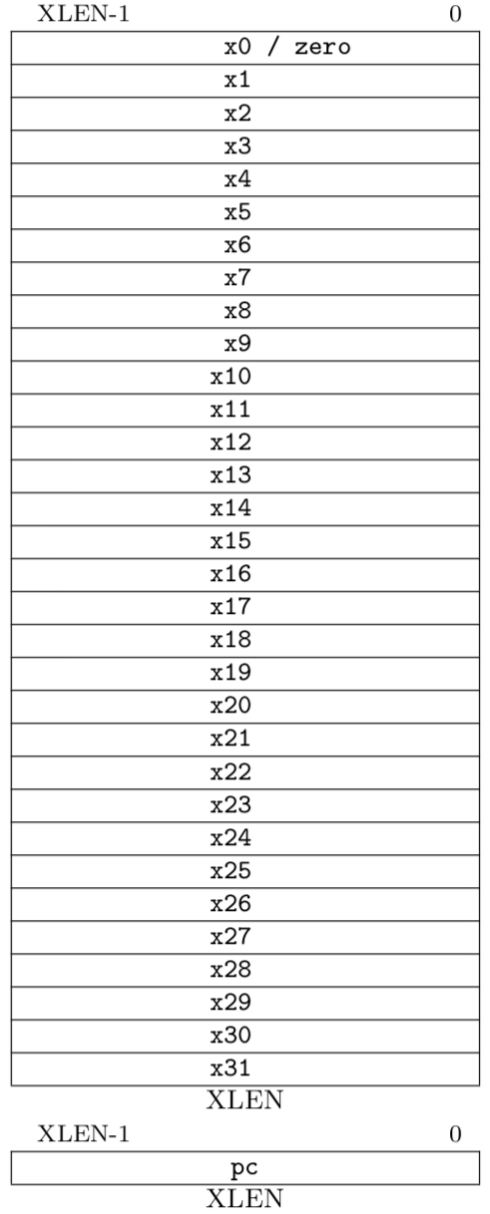
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - 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 [4:0] rw,
    input [4:0] ra,
    input [4:0] rb,
    input [31:0] busw,
    output [31:0] busa,
    output [31:0] busb
);
    reg [31:0] regs [0:31];
    always @(posedge clk)
        if (wen) regs[rw] <= busw;
    assign busa = (ra == 5'd0) ? 32'd0: regs[ra];
    assign busb = (rb == 5'd0) ? 32'd0: regs[rb];
endmodule
```

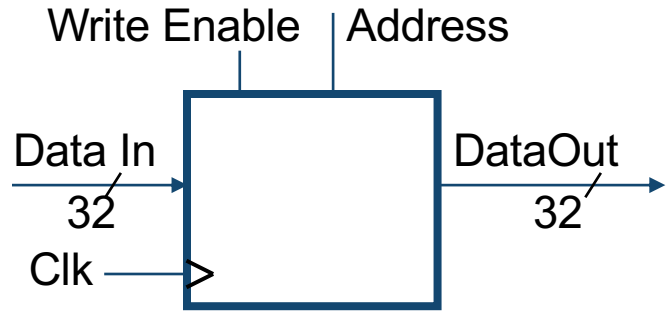




- How does RV64I register file look like?

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

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