Processor Arch: ISA & Logic

李舒辰

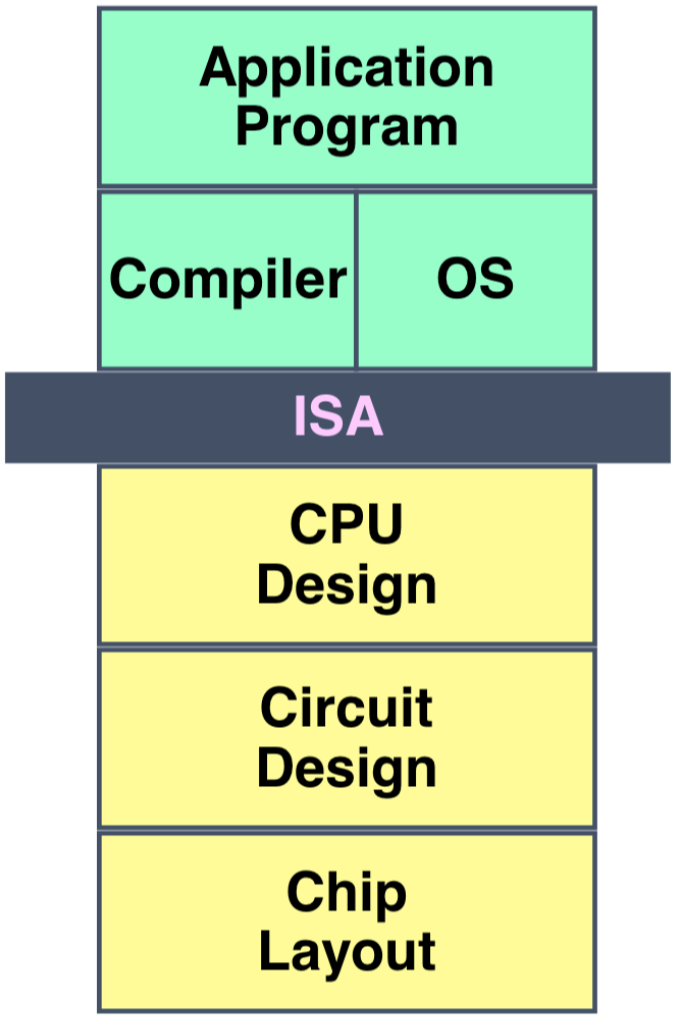
2019 年 10 月 17 日

Outline

1. Part 1: ISA
   - State
   - Instruction Set
   - Exception
   - Program Structure
2. Part 2: Logic Design
   - Communication
   - Computation
   - Storage
3. Additional: Register Implementation

Part 1: ISA

Layer of Abstraction

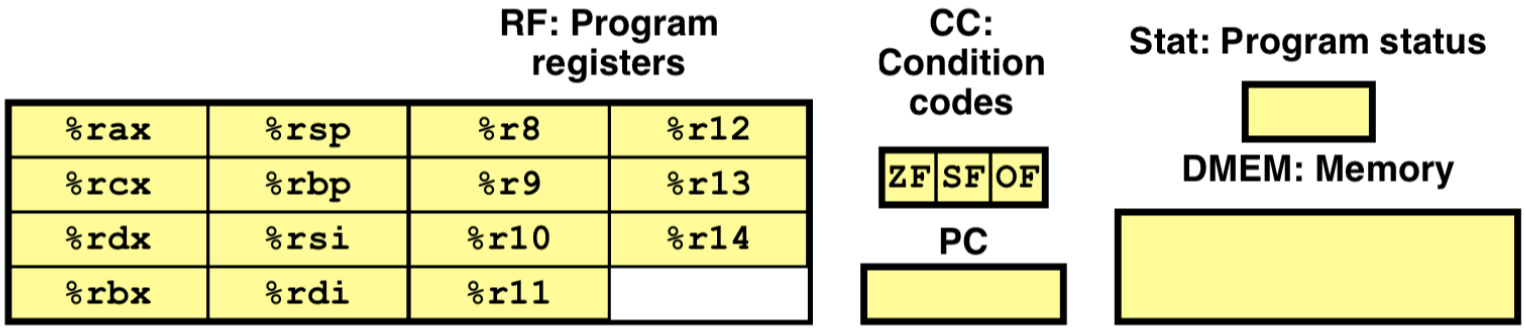


- Above: How to program
- Below: How to implement

Definition

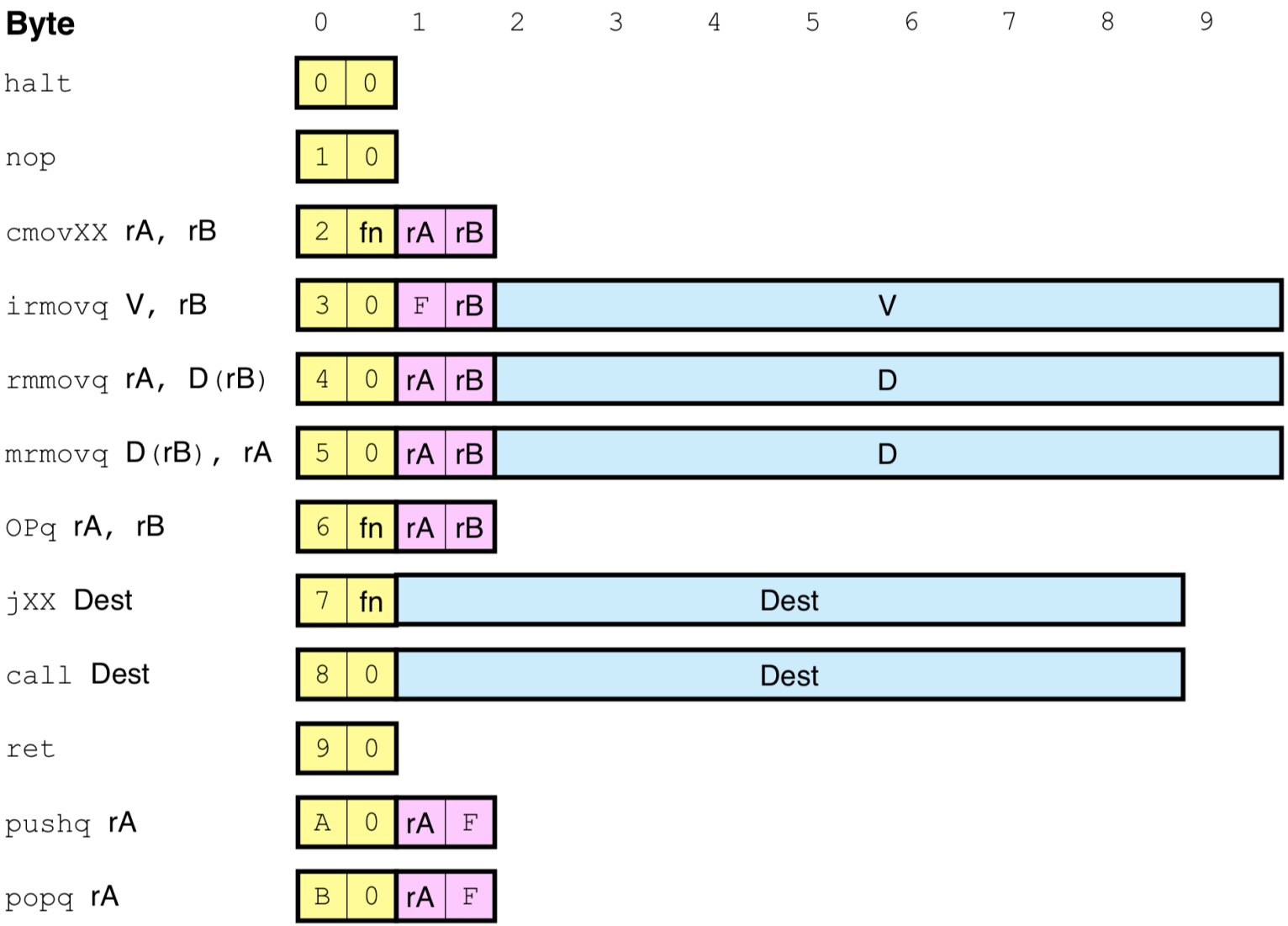
$$\text{ISA} = \begin{cases} \text{states} \\ \text{set of instructions} \ + \ \text{their encodings} \\ \text{programming conventions} \\ \text{exceptional events handling} \end{cases}$$

Y86-64 ISA: State



- Program registers: 15 registers, 64 bit
- Condition codes
- Program counter
- Program status: for exception handling
- Memory: little-endian

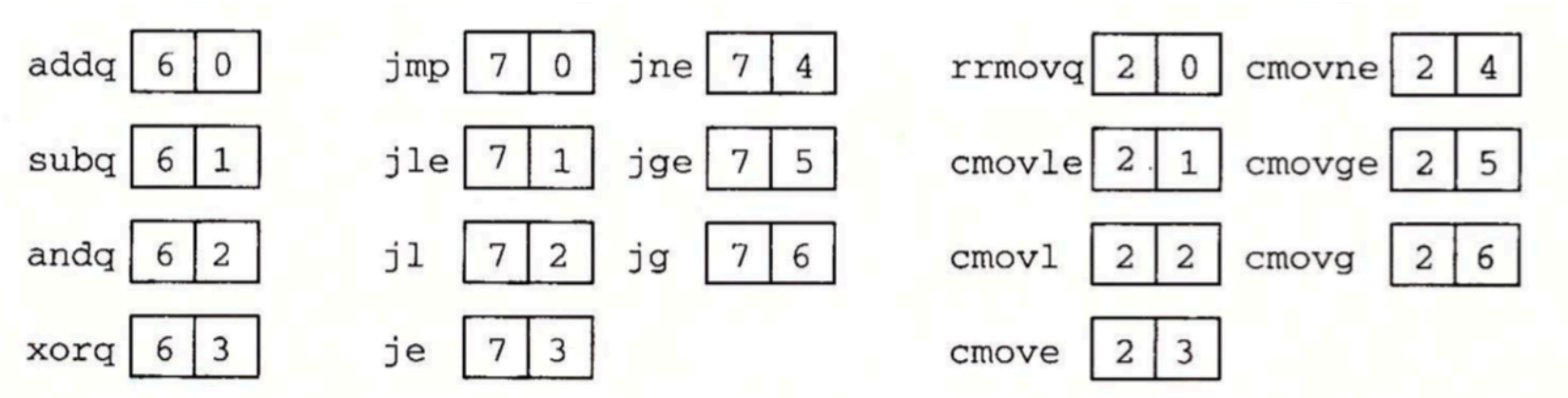
Y86-64 ISA: Instruction Set



Y86-64 ISA: Instruction Set

- The length of instruction can be determined by the first byte
  - Unique interpretation (given the starting position)
- Simpler than x86-64

Y86-64 ISA: Instruction Set



- High-order: code part, ranging from 0x0 to 0xB
- Low-order: function part
  - extensible

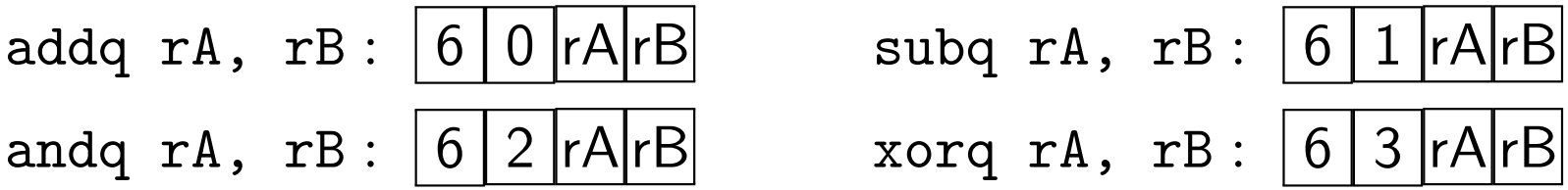
Register ID

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

- ID `0xF` indicates "no register"
  - Useful in hardware implementation

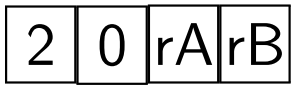
Arithmetic and Logical Operations



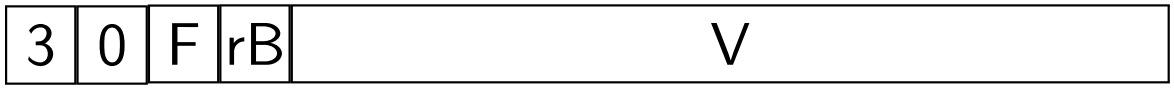
- Only allowing to be applied to register data
- Set condition codes ZF, SF, OF

Move Operations

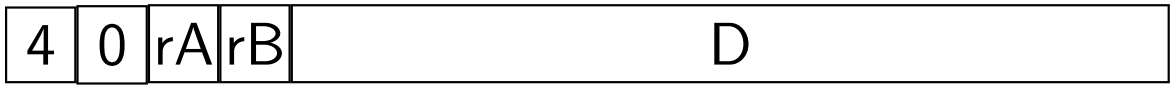
`rrmovq rA, rB`:



`irmovq V, rB`:



`rmmovq rA, D(rB)`:



`mrmovq D(rB), rA`:

- Simpler format of memory reference: displacement + base
- Transfer of immediate data to memory is not allowed
- `D(rB)`
- Little-endian encoding

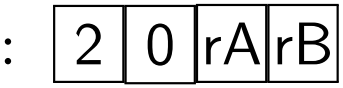
Jump Instructions

`jmp Dest :`	7	0	Dest
`jle Dest :`	7	1	Dest
`jl Dest :`	7	2	Dest
`je Dest :`	7	3	Dest
`jne Dest :`	7	4	Dest
`jge Dest :`	7	5	Dest
`jg Dest :`	7	6	Dest

- Same as x86-64 jump instructions,
- except encoding the full destination address

Conditional Move Instructions

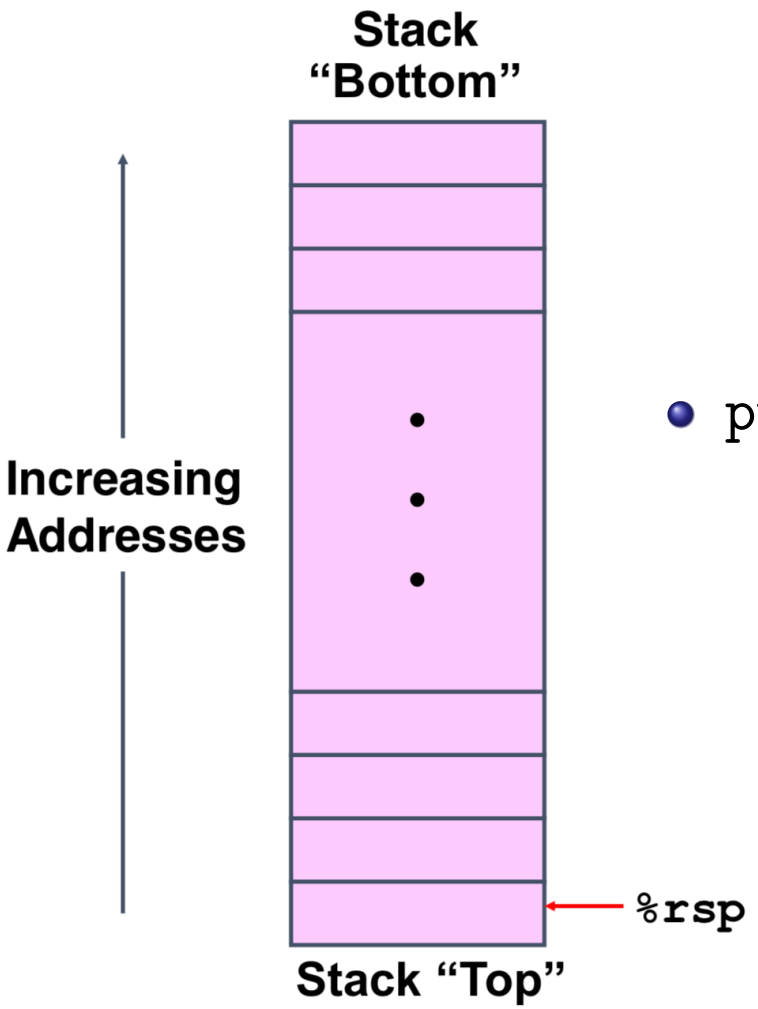
`rrmovq rA, rB` :



`cmovle rA, rB` :	2 1 rA rB	`cmovl rA, rB` :	2 2 rA rB
`cmove rA, rB` :	2 3 rA rB	`cmovne rA, rB` :	2 4 rA rB
`cmovge rA, rB` :	2 5 rA rB	`cmovg rA, rB` :	2 6 rA rB

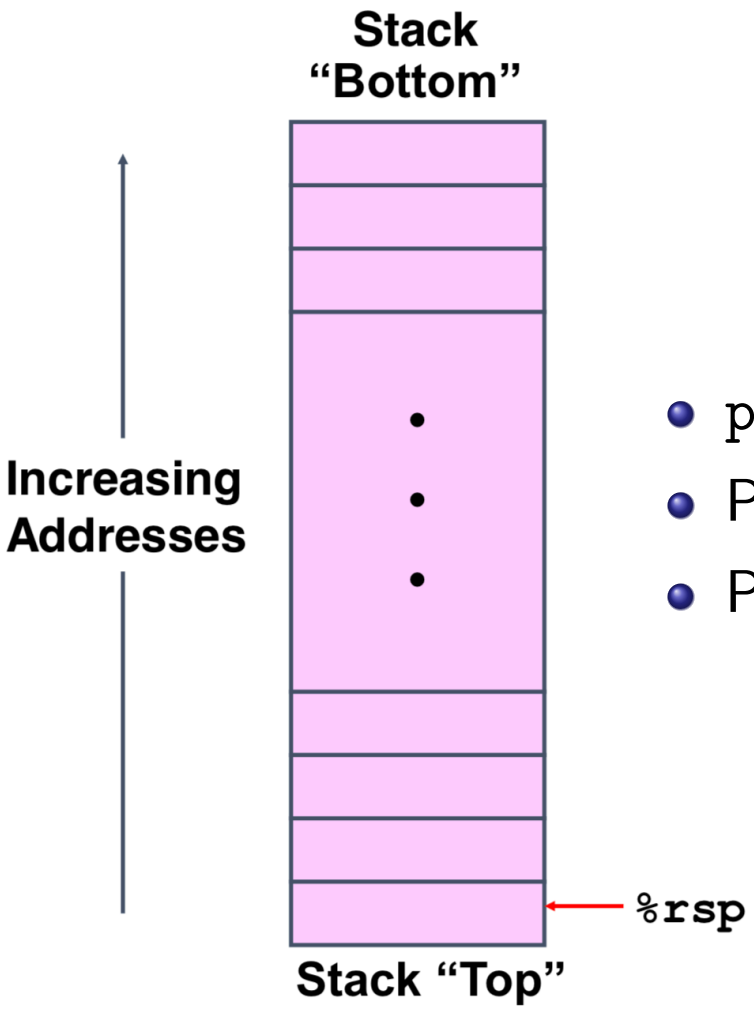
- Only allowed to be applied to register data

Program Stack



- `push %rsp / pop %rsp ?`

Program Stack

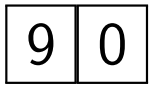


- `push %rsp / pop %rsp` ?
- Push: (1) decrease `%rsp`; (2) store
- Pop: (1) read; (2) increase `%rsp`

Subroutine Call and Return

`call Dest :`

`ret :`



- Like x86-64

Miscellaneous Instructions

`nop` :	1 | 0
`halt` :	0 | 0

- `nop`: a placeholder,
  - when writing assembly code or debugging
  - force memory alignment
  - ...

Status Conditions

Mnemonic	Code	Description
AOK	1	Normal operation
HLT	2	`halt` encounterd
ADR	3	Bad address
INS	4	Invalid instruction

- If HLT, ADR, or INS encountered, stop program execution

Sample Program Structure

```
init:                       # Initialization
    . . .
    call Main
    halt

    .align 8                # Program data
array:
    . . .

Main:                       # Main function
    . . .
    call len    . . .

len:                        # Length function
    . . .

    .pos 0x100              # Placement of stack
Stack:
```

- Start at address 0
- Set up stack: position

Sample Program Structure

```
init:
      # Set up stack pointer
      irmovq Stack, %rsp
      # Execute main program
      call Main
      # Terminate
      halt

# Array of 4 elements + terminating 0
      .align 8
Array:
      .quad 0x000d000d000d000d
      .quad 0x00c000c000c000c0
      .quad 0x0b000b000b000b00
      .quad 0xa000a000a000a000
      .quad 0
```

- Set up stack: pointer %rsp
- Initialize data

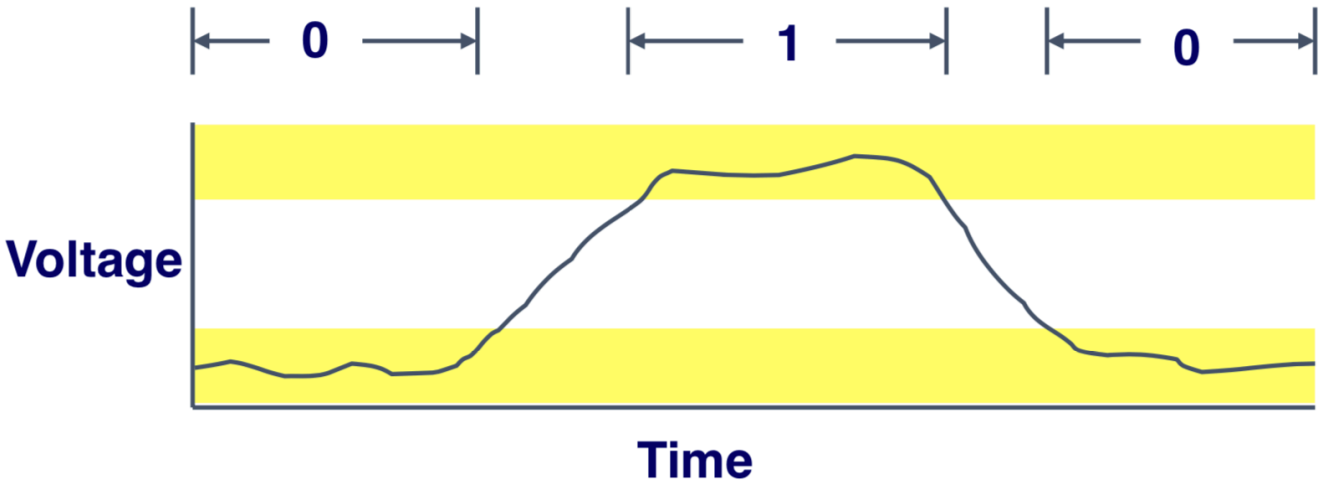
Part 2: Logic Design

Overview

Hardware requirements:

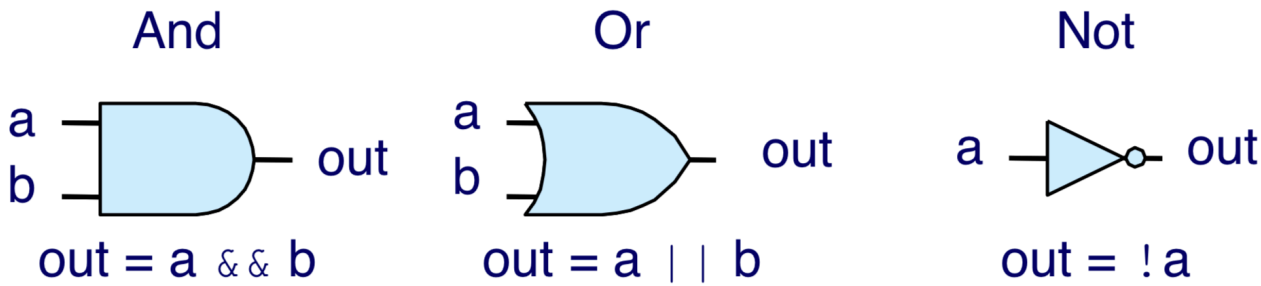
1. Communication
2. Computation
3. Storage

Digital Signals



- Threshold: continuous signal $\rightarrow$ discrete values
- Stable, can make circuits simple, small, and fast

Logic Gates



- Respond continuously to the changes of inputs,
- with some small delay

Combinational Circuits

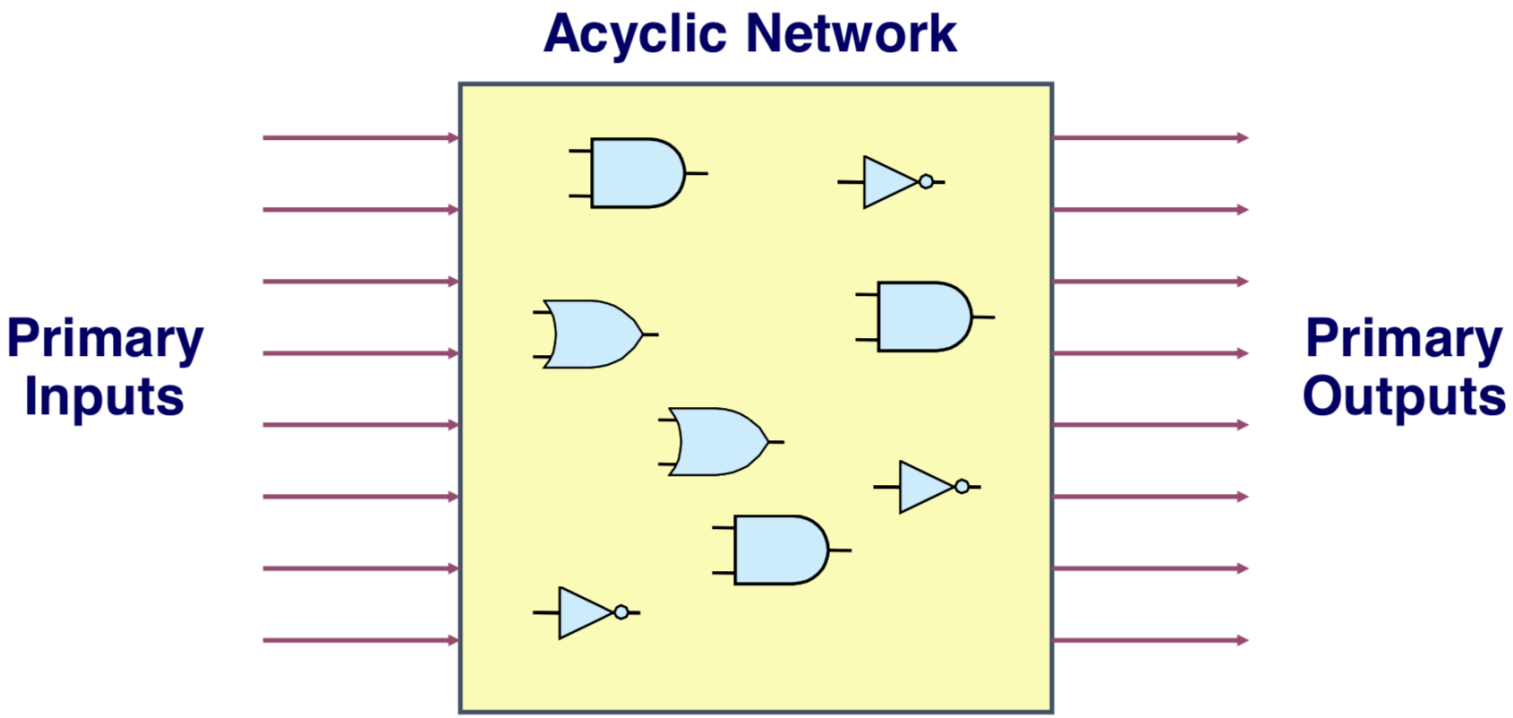
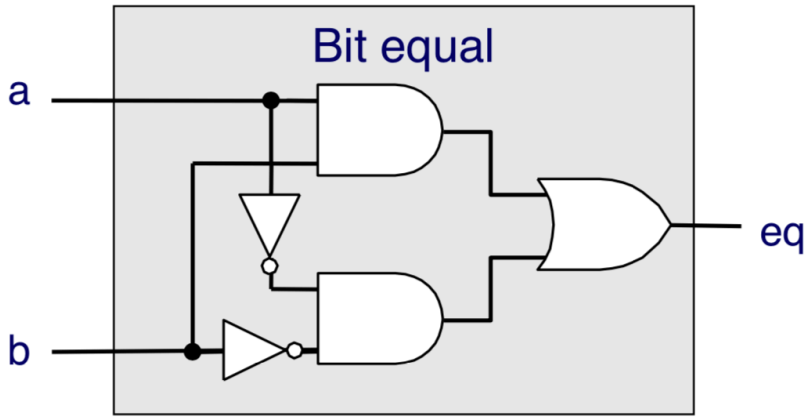


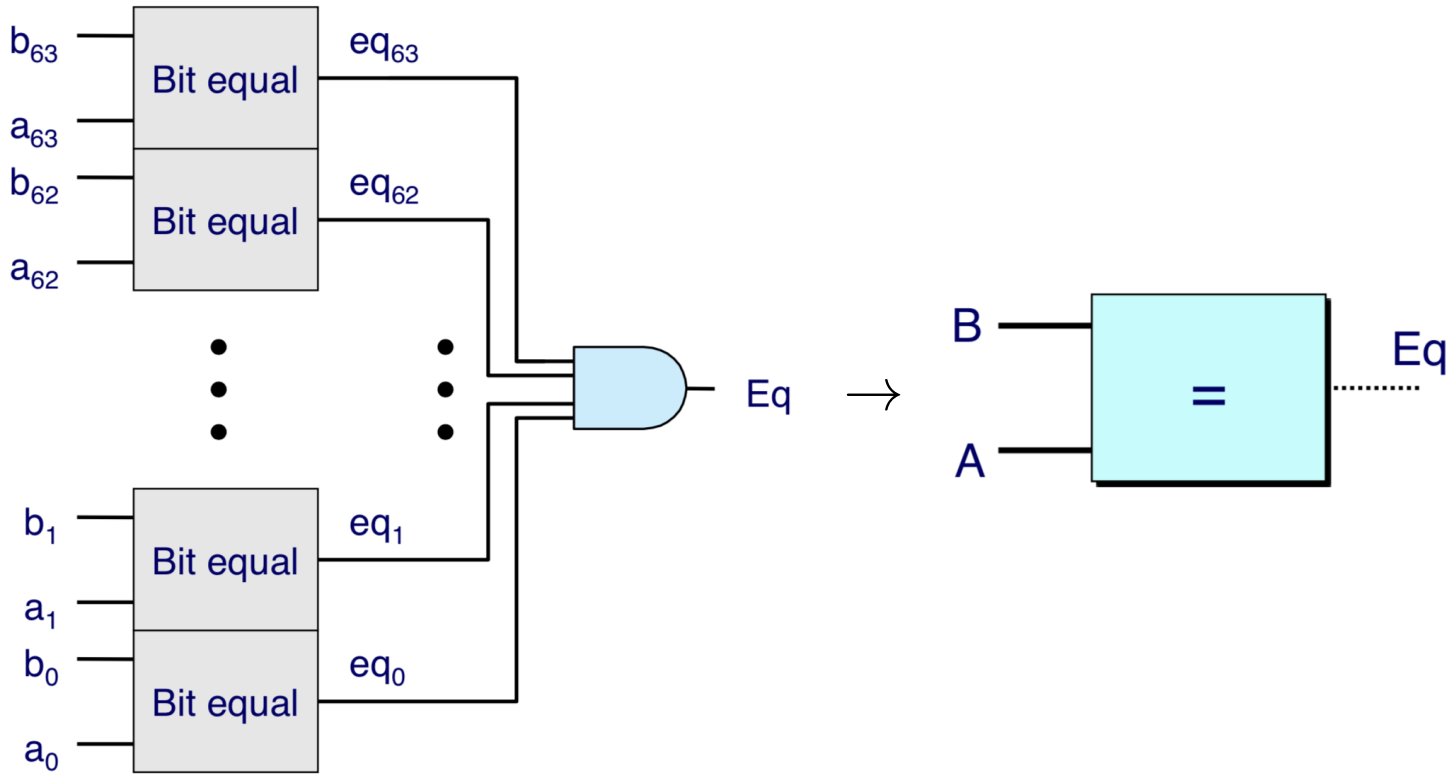
Figure: Acyclic Network of Logic Gates

Bit Equality



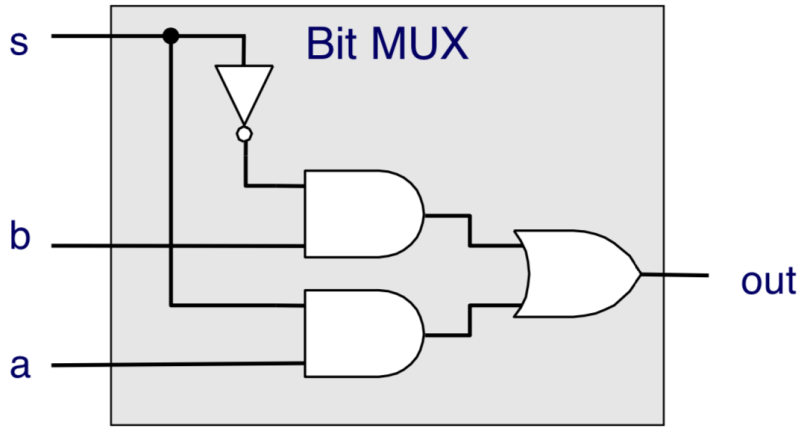
- `bool eq = (a && b) || (!a && !b)`
- Hardware Control Language (HCL)
  - A simple hardware description language with C style expressions
  - static

Word Equality



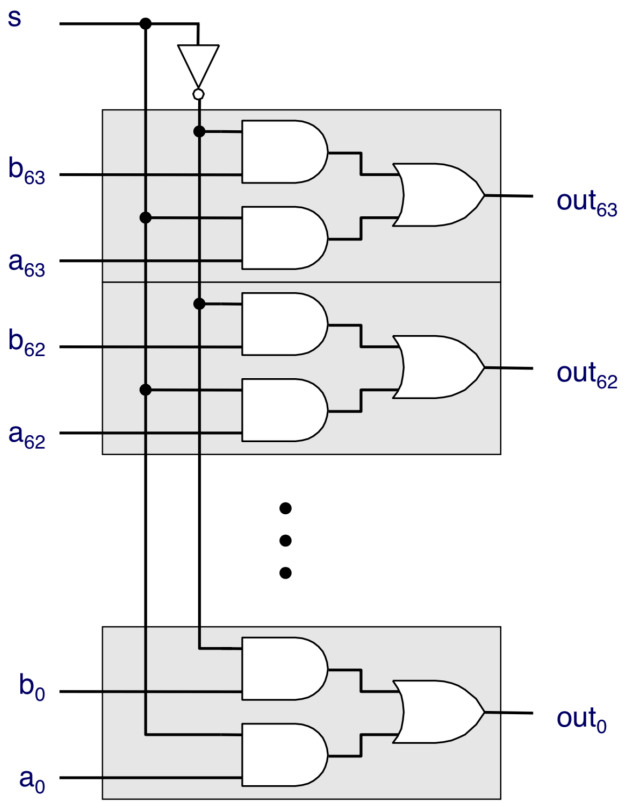
- HCL representation: `bool Eq = (A == B)`

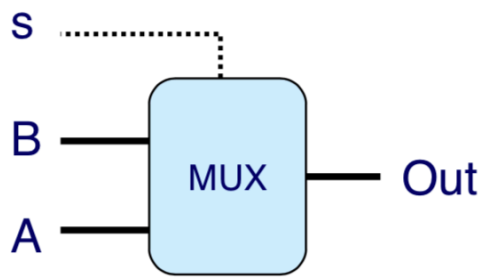
Bit-Level Multiplexor



- `out = s ? a : b`
- HCL representation: `bool out = (s && a) || (!s && b)`

Word Multiplexor






```
int Out = [
  s : A;
  1 : B;
];
```

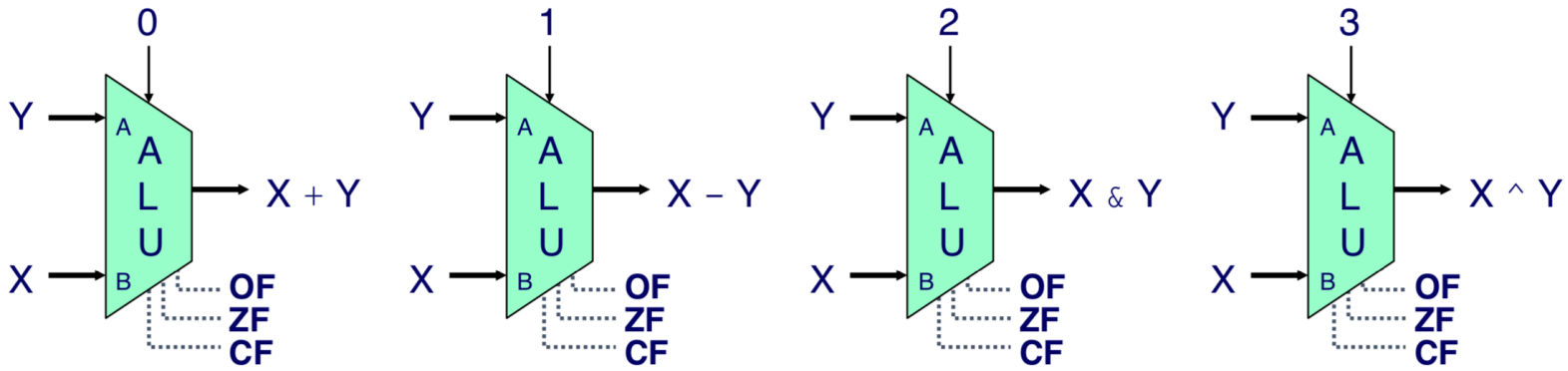
- Case expression: output value for the first successful test
- Only produce `!s` once

More Examples of Case Expressions

```
int Min3 = [
  A < B && A < C : A;
  B < A && B < C : B;
  1              : C;
];
```

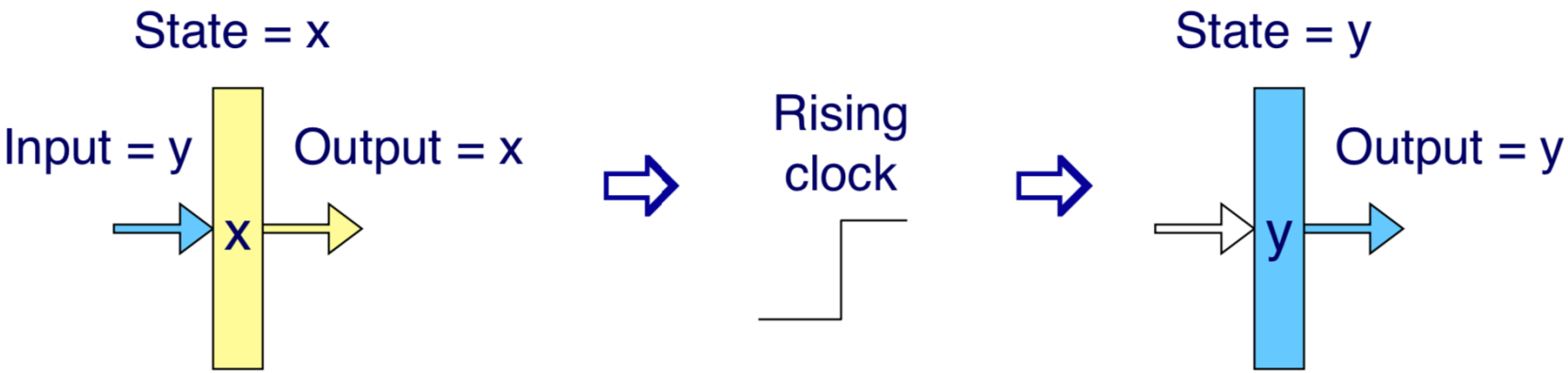
```
int Out4 = [
  !s1 && !s0 : D0;
  !s1        : D1;
  !s0        : D2;
  1          : D3;
];
```

Arithmetic Logic Unit

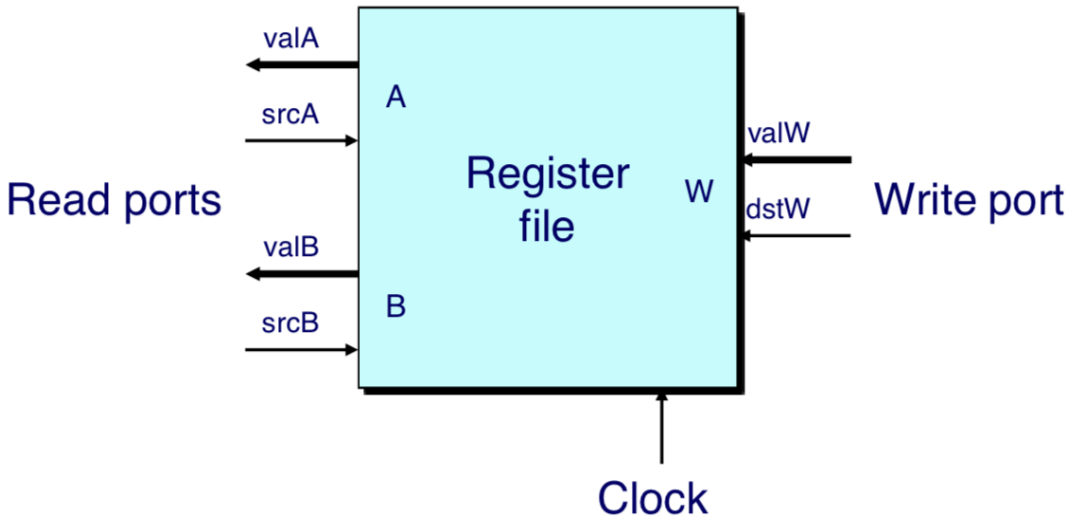


- Corresponds to 4 operations in Y86-64 ISA
- B - A: in anticipation of the ordering in `subq A B`
- Condition Codes

Registers



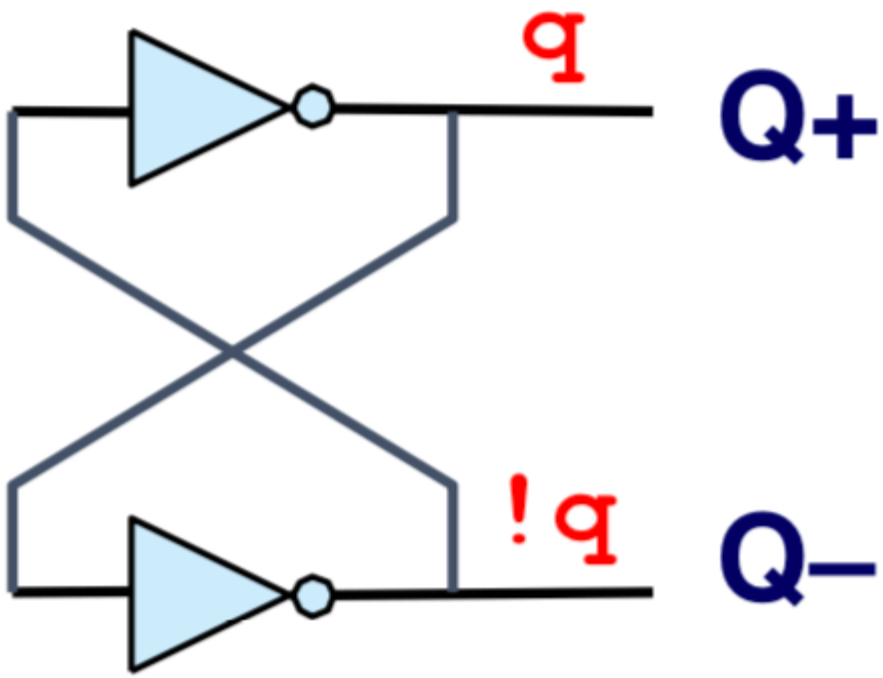
Register File



- Multiple ports
- Reading as if it were a block of combinational logic
  - Output generated by the input address, after some delay
- Writing like register, updating only when clock rises

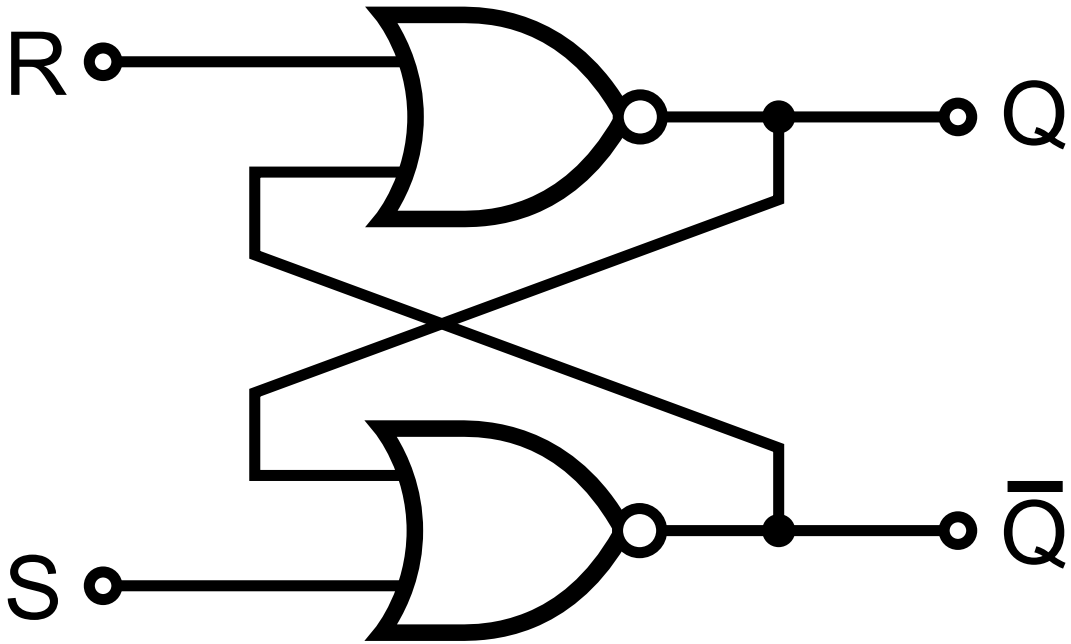
Additional: Register Implementation

Bistable Element

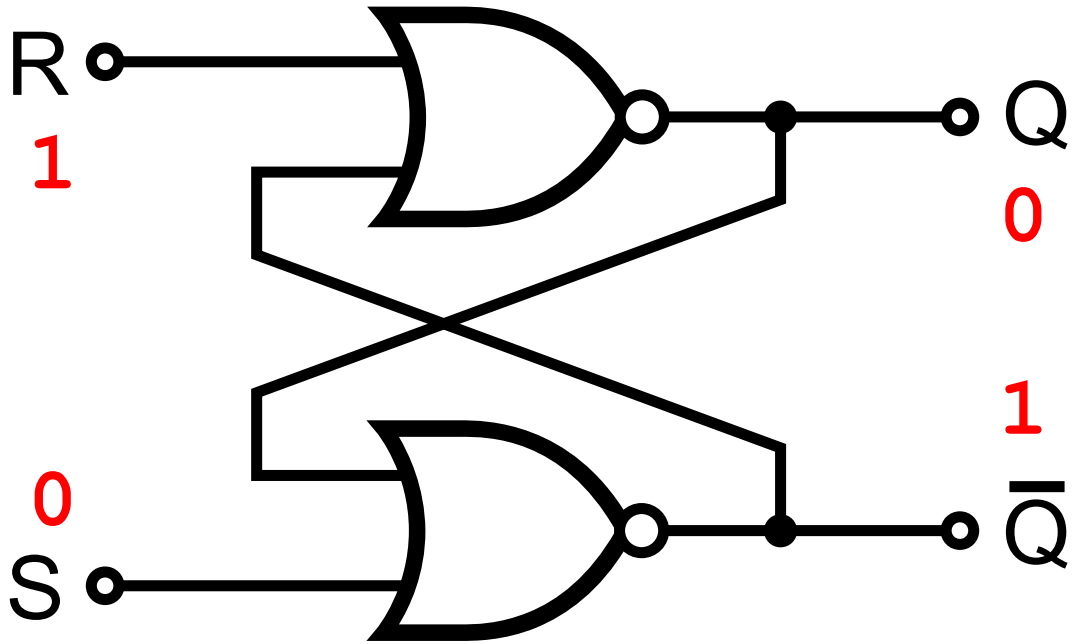


q = 0 or 1

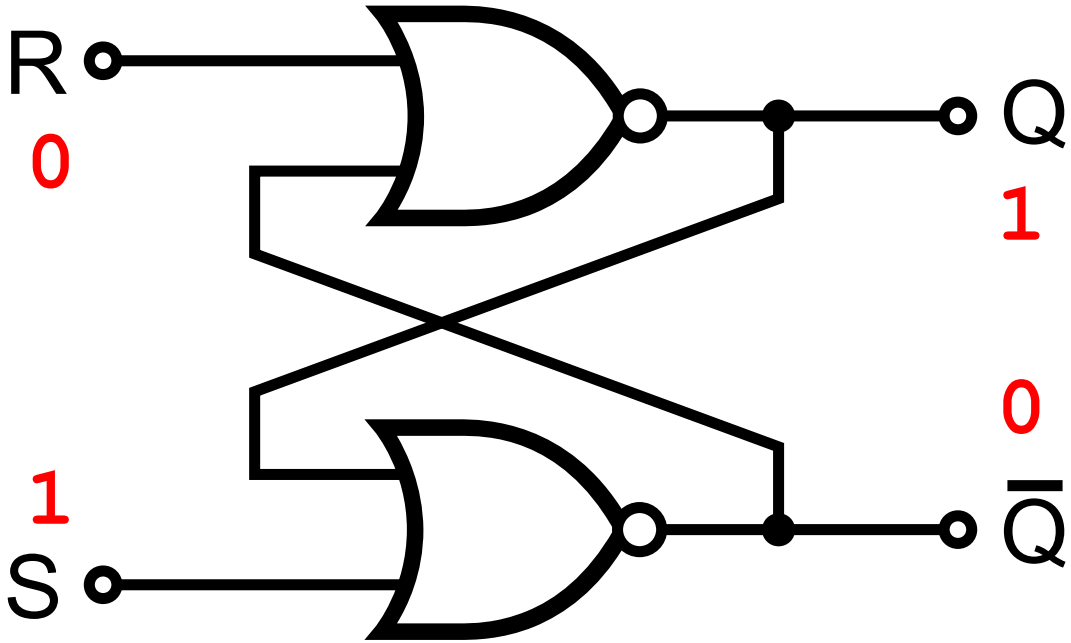
R-S Latch



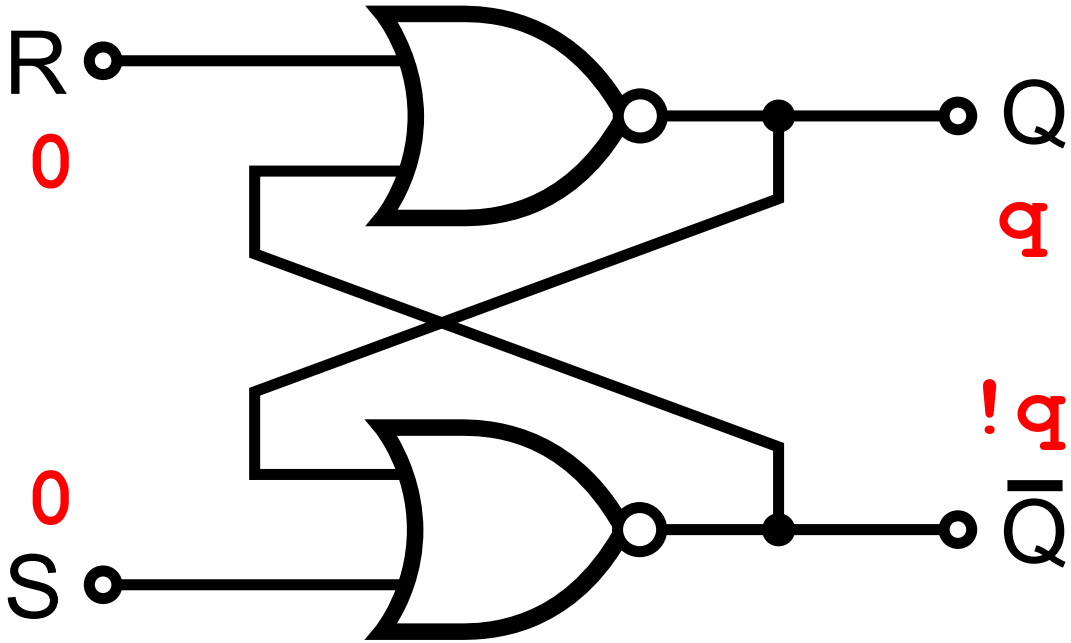
R-S Latch: Resetting



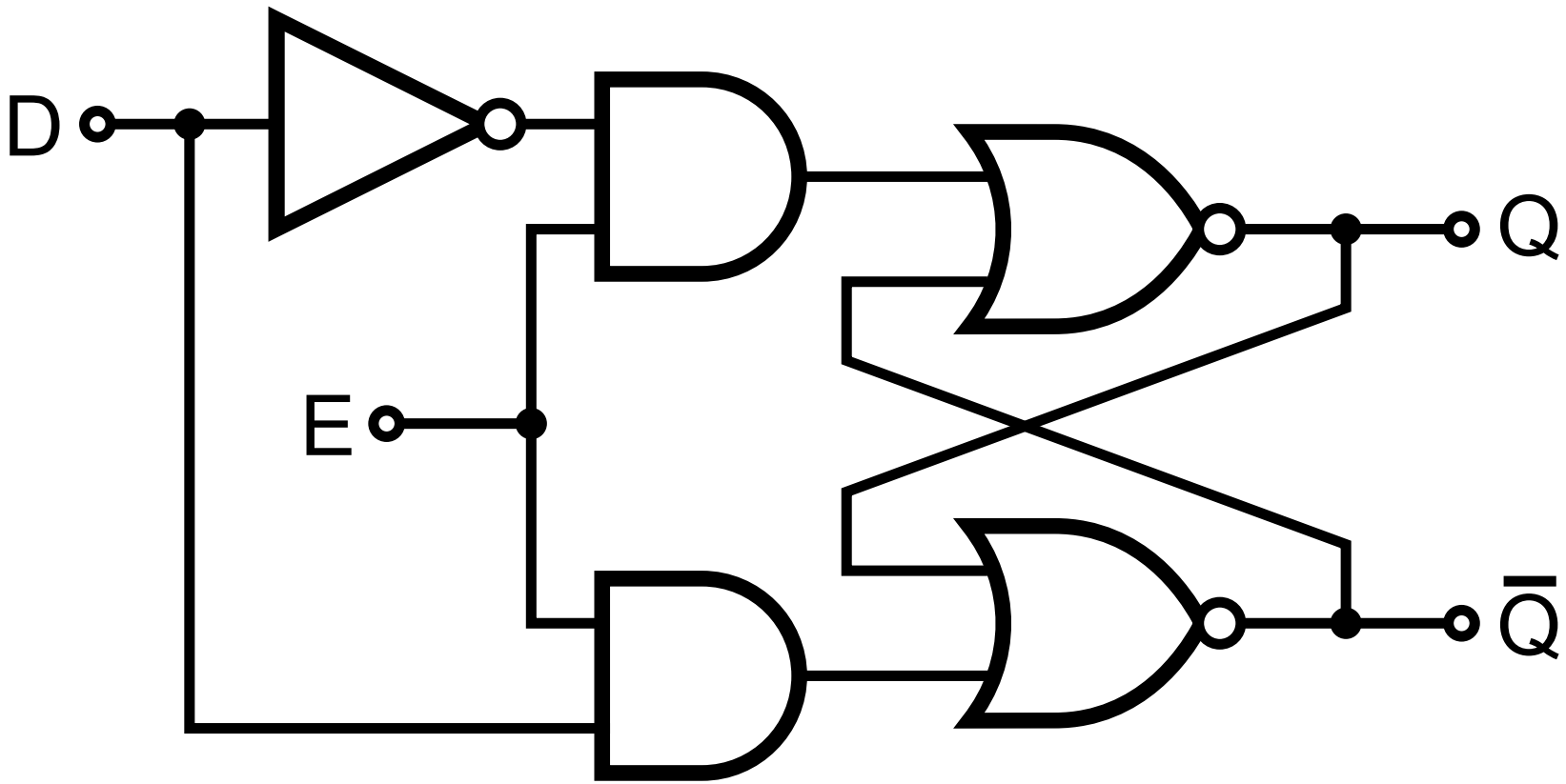
R-S Latch: Setting



R-S Latch: Holding



D Latch



Edge-Triggered Latch