Digital Logic and Sequential Y86-64 Implementations

CSCI 237: Computer Organization
15th Lecture, Mar 14, 2025

Jeannie Albrecht

Administrative Details

- Lab 3 – great job! Really good progress!
  - Due before spring break (use submit237 3 {files})
- HW 4 due today on Glow
  - We can review on Tuesday if necessary
- Midterm details – material through today
  - Mar 19/20th during lab time in Wege (lots of room to spread out)
  - Closed notes
  - Focuses on Chapters 1-4 (emphasis on Ch 2 and 3) and Labs 1 and 2
  - Review session on Tuesday at 8pm in TPL 205
  - No class on WED! I'll be in my office to answer questions.
  - But we do have class next Friday ☹

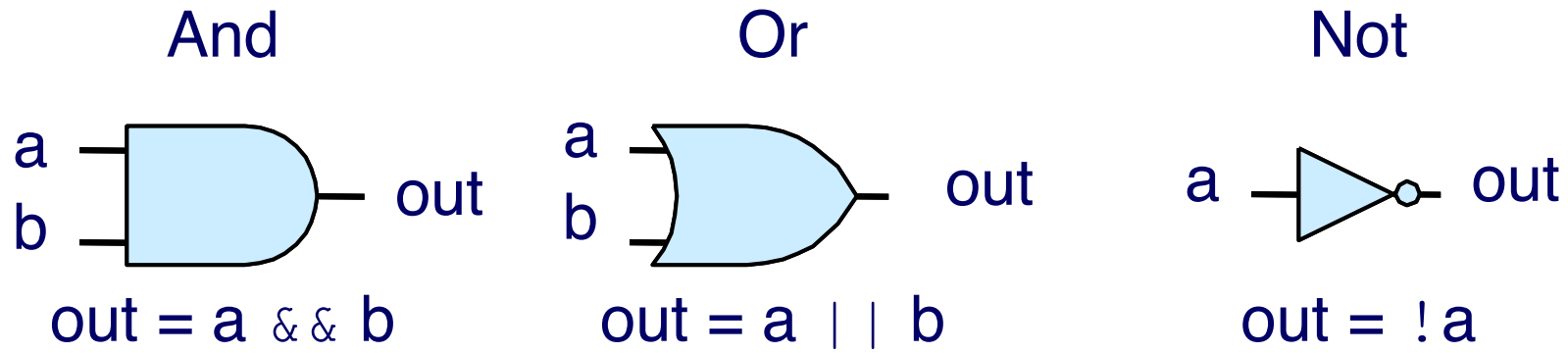
Last time

- Wrapped up Y86-64 overview
- Learned how to write, assemble, and run Y86 code
- Observations from lab
  - Simplified instructions have consequences
  - Some actions take multiple steps in Y86 but only one step in x86
  - RISC vs CISC tradeoffs

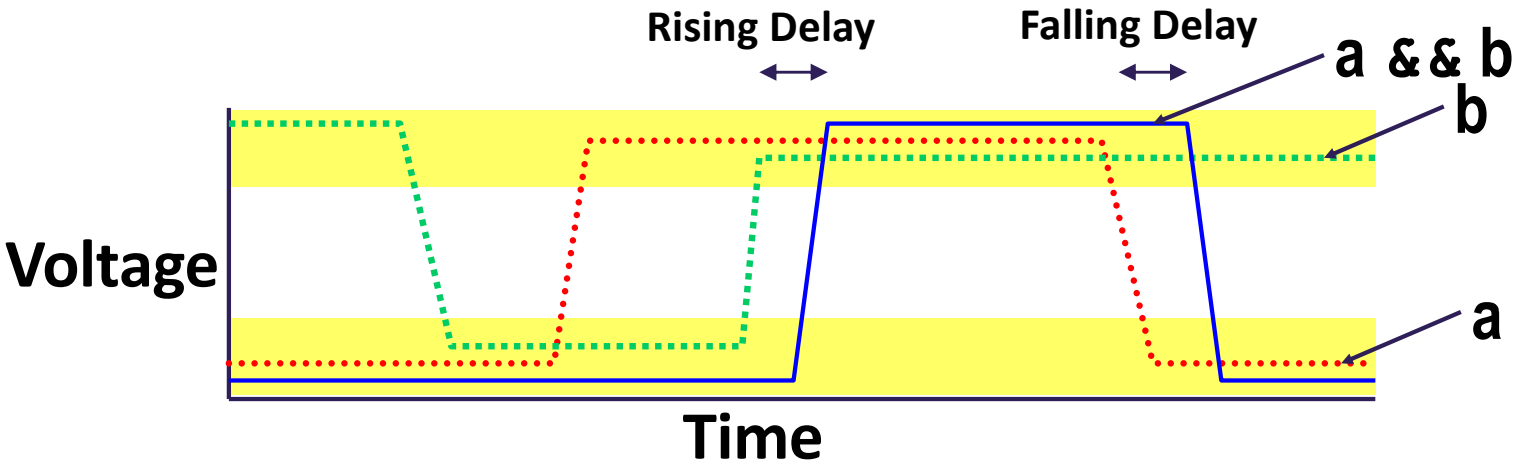
Today

- Discuss digital logic and HCL (Ch 4.2)
- Brief overview of memory and clocking
  - How is information stored
- Intro to sequential Y86-64 implementations (Ch 4.3)
  - Organizing processing into *stages*

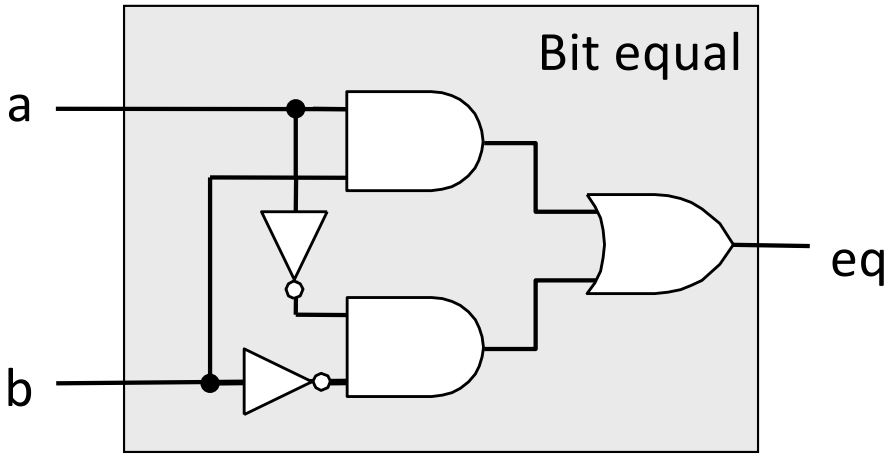
Ch 4.2 – Computing with Logic Gates

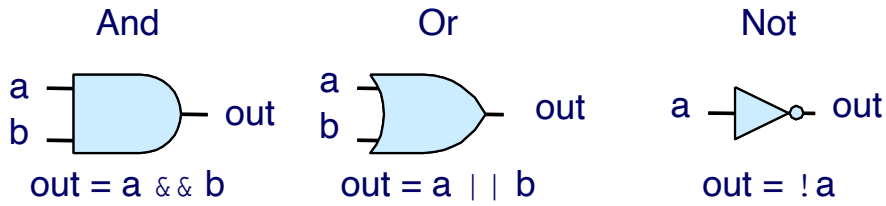


- Logic gates are the basic computing elements for digital circuits
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some small delay



Bit Equality



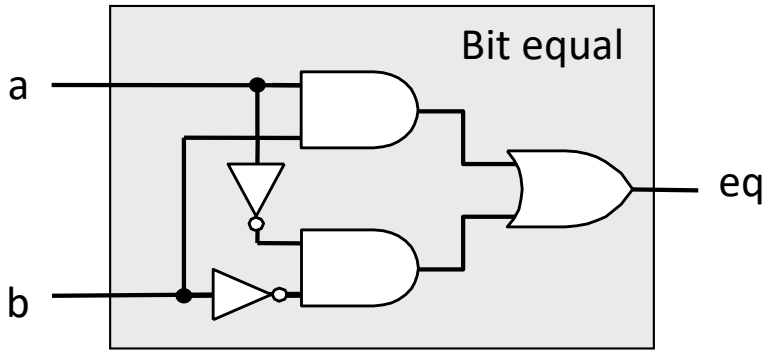


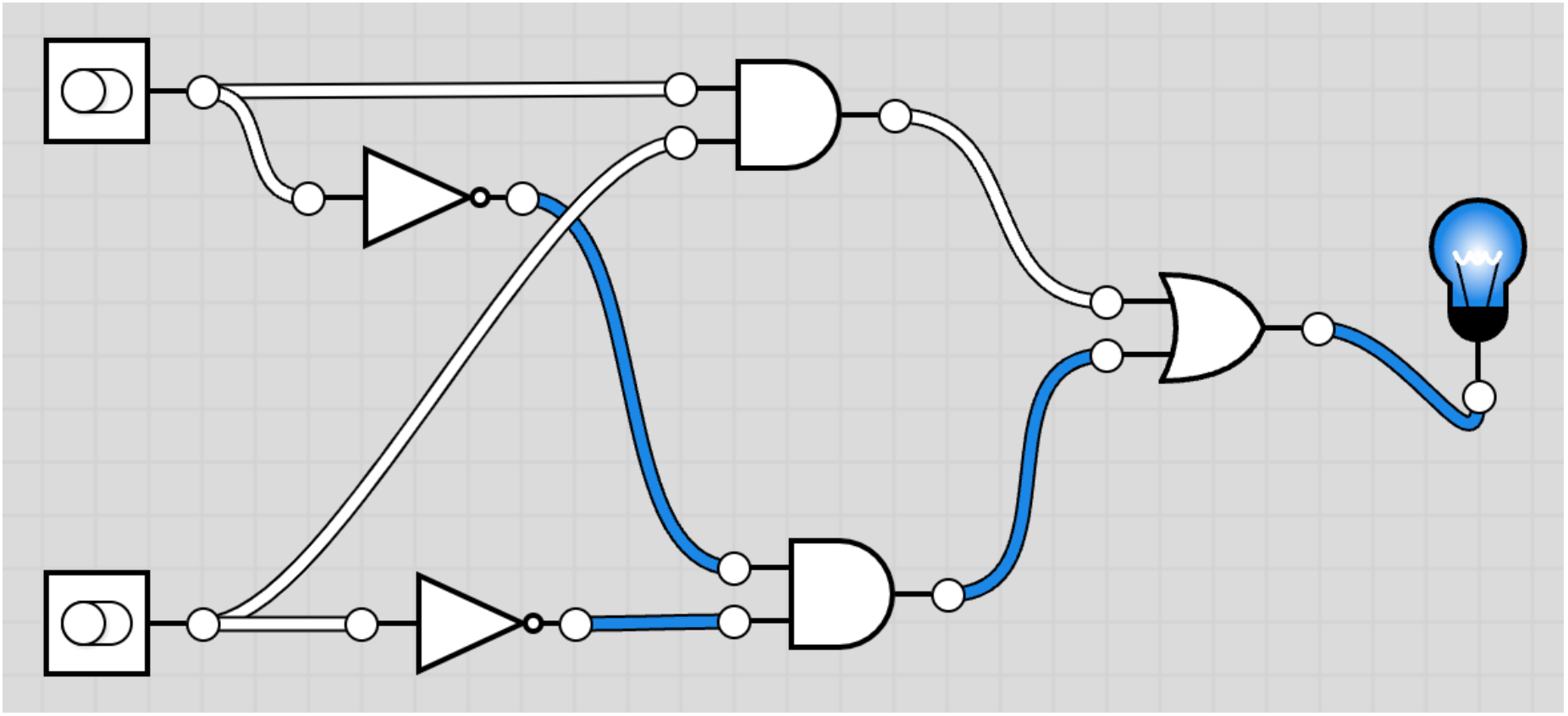
HCL Expression

```
bool eq = (a&&b)||(!a&&!b)
```

- Generate 1 (true) if a and b are equal
- Hardware Control Language (HCL) for Y86 processors
  - Very simple hardware description language (HDL)
    - Boolean operations have syntax similar to C logical operations
  - We'll use it to describe control logic for processors
    - As close as we get to designing hardware
  - Verilog and VHDL are examples of "real" HDLs

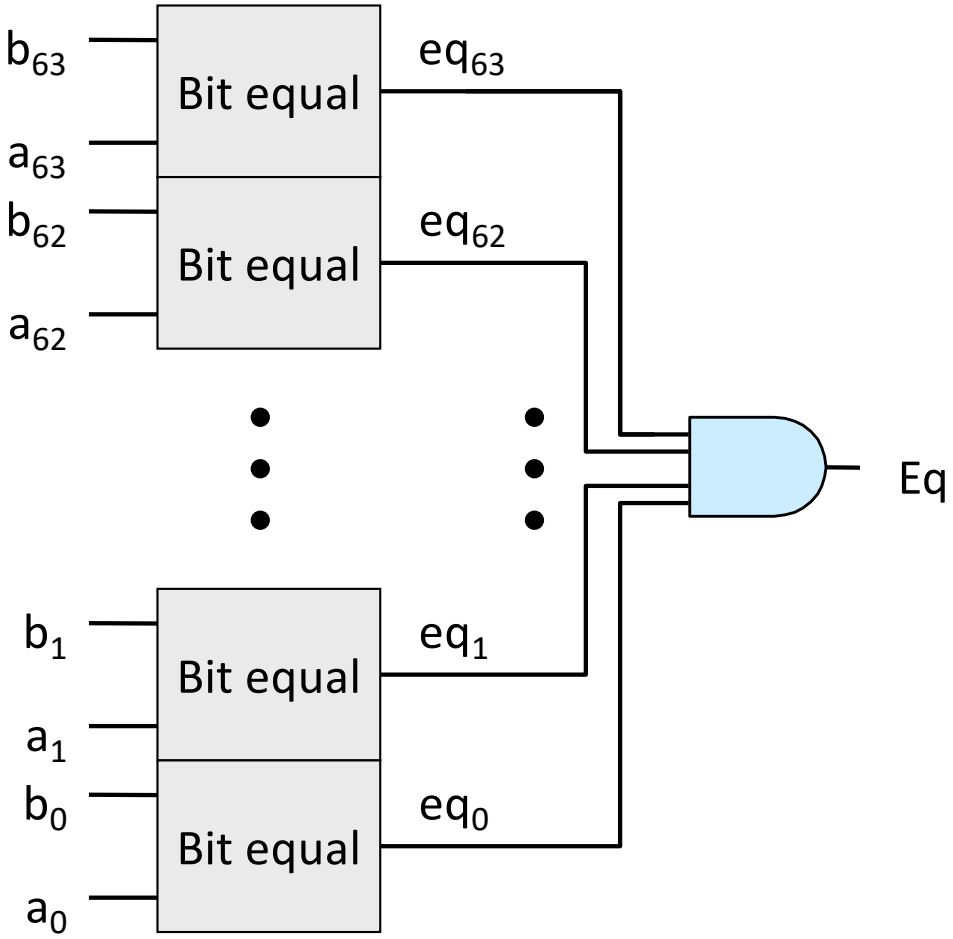
Bit Equality



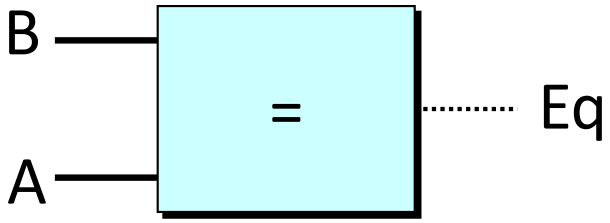


https://logic.ly/demo/

Word Equality



Word-Level Representation

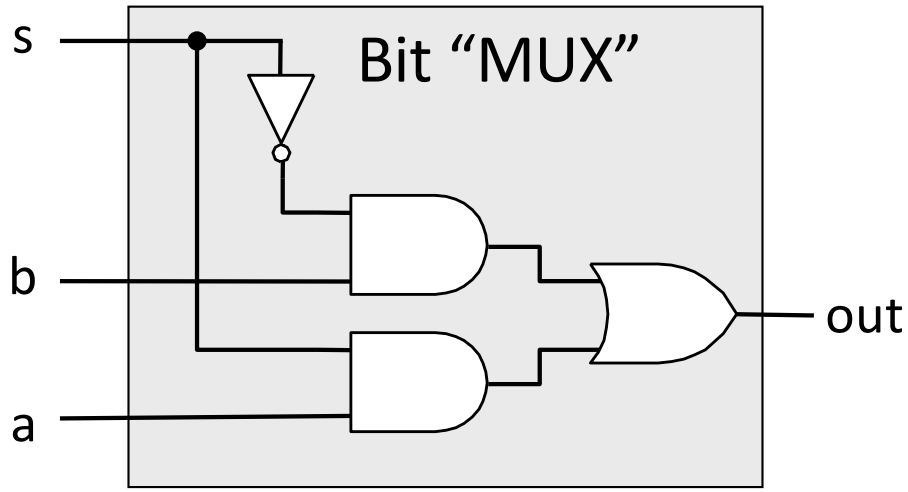


HCL Representation

```
bool Eq = (A == B)
```

- Words are equals if all bits are the same
- 64-bit word size
- HCL representation
  - Equality operation
  - Generates Boolean value

Bit-Level Multiplexor

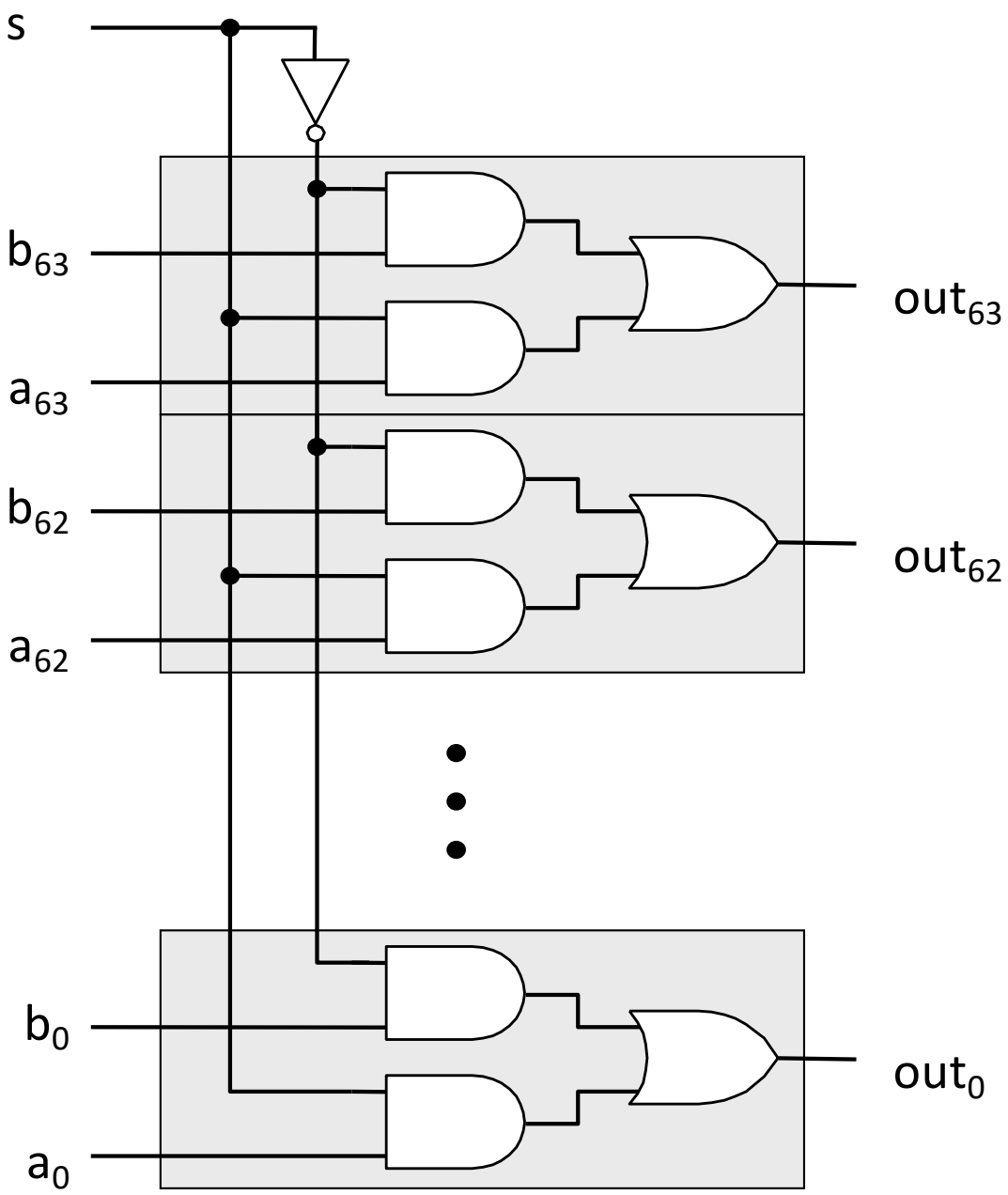


HCL Expression

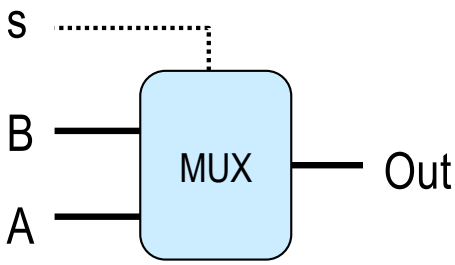
```
bool out = (s&&a)||(!s&&b)
```

- Control input signal s
- Data signals a and b
- Output a when s=1, b when s=0

Word Multiplexor



Word-Level Representation



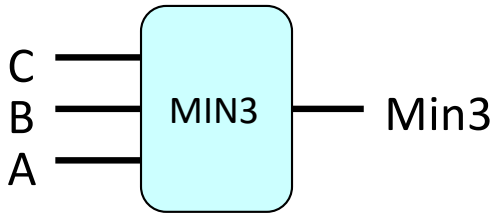
HCL Representation

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

- Select input word A or B depending on control signal s
- HCL representation
  - Case expression
  - Series of test : value pairs
  - Output value for first successful test

HCL Word-Level Examples

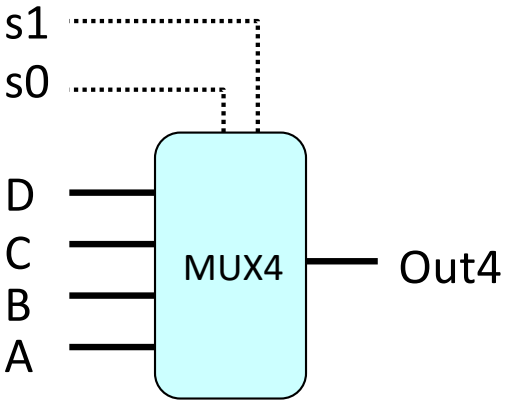
Minimum of 3 Words




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

- Find minimum of three input words
- HCL case expression
- Final case guarantees match

4-Way Multiplexor




```
int Out4 = [
  !s1&&!s0: A; #00
  !s1     : B; #01
  !s0     : C; #10
  1       : D; #11
];
```

- Select one of 4 inputs based on 2 control bits
- HCL case expression
- Simplify tests by assuming sequential matching

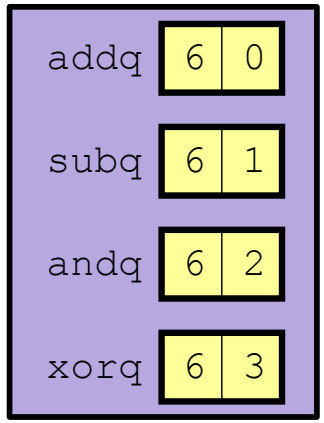
Hardware Control Language (HCL) Summary

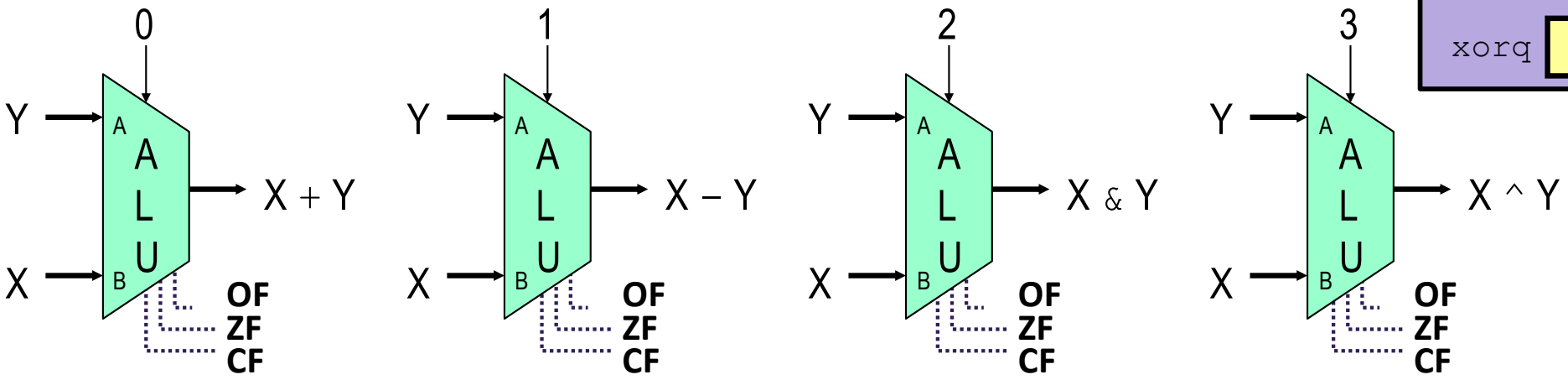
- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify
- Data Types
  - `bool`: Boolean
    - `a`, `b`, `c`, …
  - `int`: words
    - `A`, `B`, `C`, …
    - Does not specify word size—bytes, 64-bit words, …
- Statements
  - `bool a = bool-expr ;`
  - `int A = int-expr ;`

HCL Operations

- Classify by type of value returned
- Boolean Expressions
  - Logic Operations
    - `a && b, a || b, !a`
  - Word Comparisons
    - `A == B, A != B, A < B, A <= B, A >= B, A > B`
  - Set Membership
    - `A in { B, C, D }`
      - Same as `A == B || A == C || A == D`
- Word Expressions
  - Case expressions
    - `[ a : A; b : B; c : C ]`
    - Evaluate test expressions `a`, `b`, `c`, … in sequence
    - Return word expression `A`, `B`, `C`, … for first successful test

Arithmetic Logic Unit (ALU)





- Very important combinational logic circuit
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

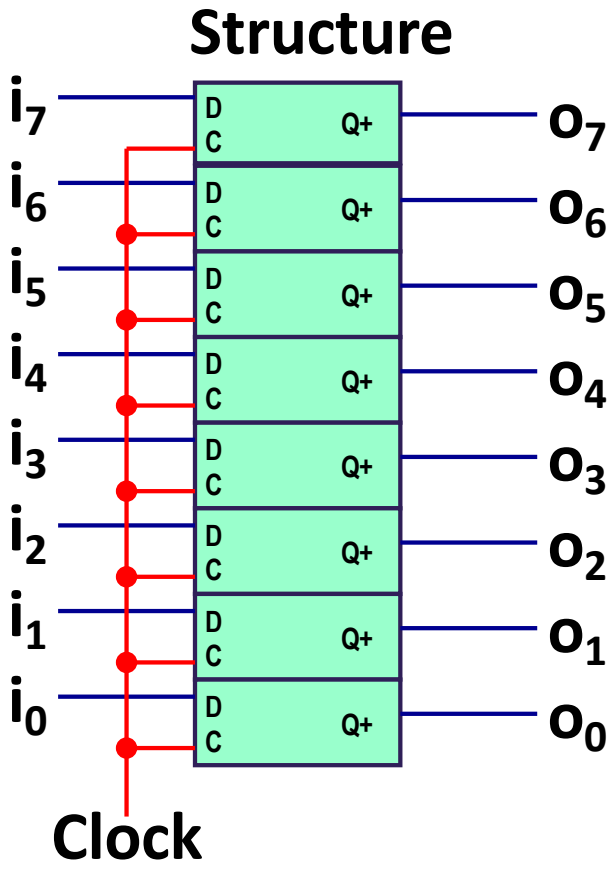
As aside: VHDL example

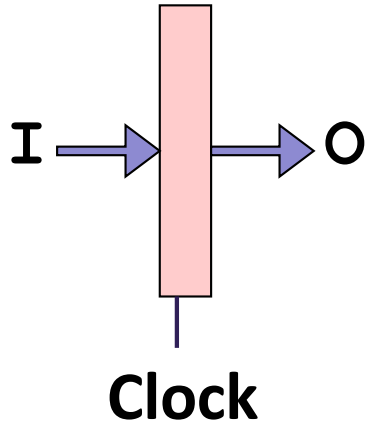
```
1  -- Simple OR gate design
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4
5  entity or_gate is
6  port(
7    a: in std_logic;
8    b: in std_logic;
9    q: out std_logic);
10 end or_gate;
11
12 architecture rtl of or_gate is
13 begin
14   process(a, b) is
15   begin
16     q <= a or b;
17   end process;
18 end rtl;
19
```

Moving on: Storing Bits

- Combination circuits do not store information
  - Only react to signals at inputs and generate outputs
- Creating a *sequential circuit* requires storage
  - Seq circuits have state and perform computations on that state
- Storage devices are controlled by a single *clock*
  - A periodic signal that determines when new values are to be loaded
- Two classes of memory devices:
  - Clocked registers – store individual bits or words
  - Random access memories (RAM) – store multiple words using an address to select where word should be read/written
- Distinction between hardware registers and program registers
  - Hardware registers are directly connected to circuits
  - Program registers are stored in register file, which is a type of RAM

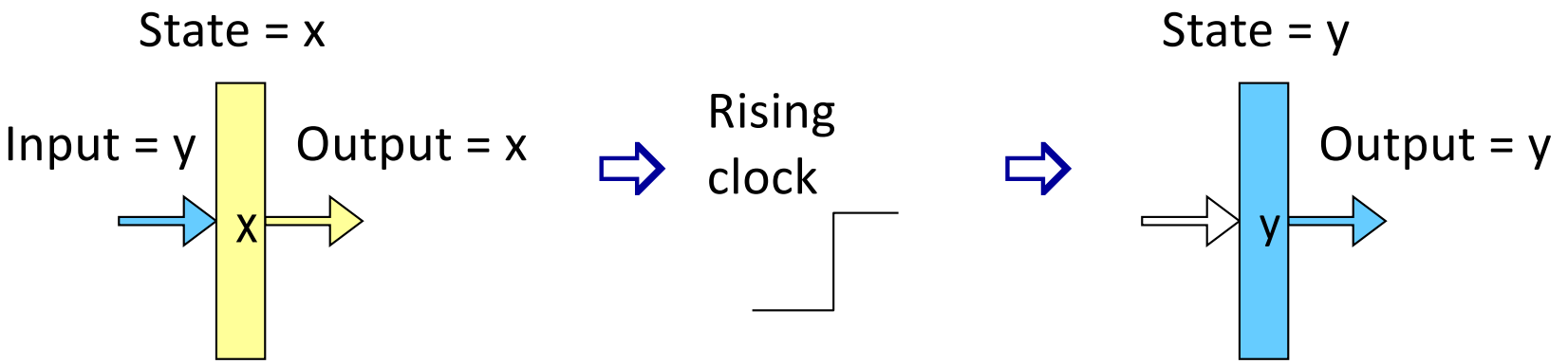
Hardware Registers





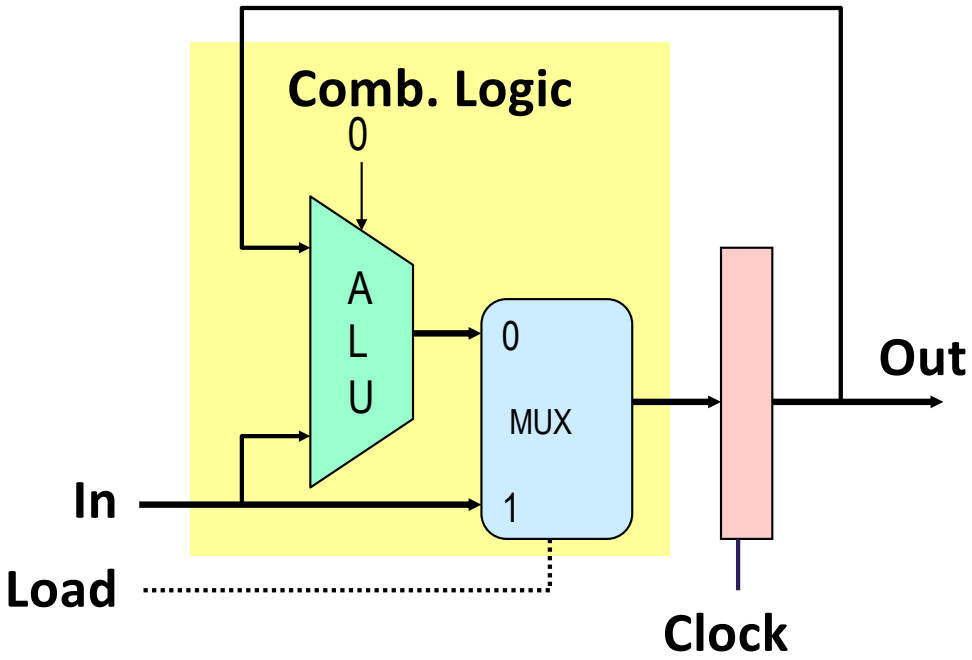
- Stores word of data
- Different from *program registers* seen in assembly code (e.g., %rdi)
- Collection of edge-triggered *latches*
- Loads input on *rising edge* of clock

Register Operation

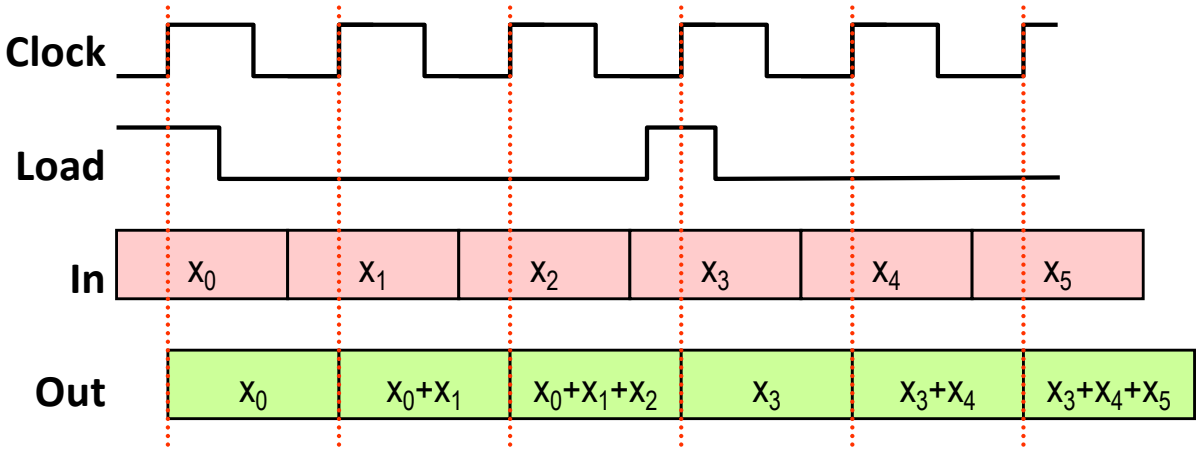


- Register stores data bits
- Acts as barrier between input and output
- As clock rises, loads input, possibly changes output
- Y86-64 processor uses clocked registers to hold PC, CC, and Stat

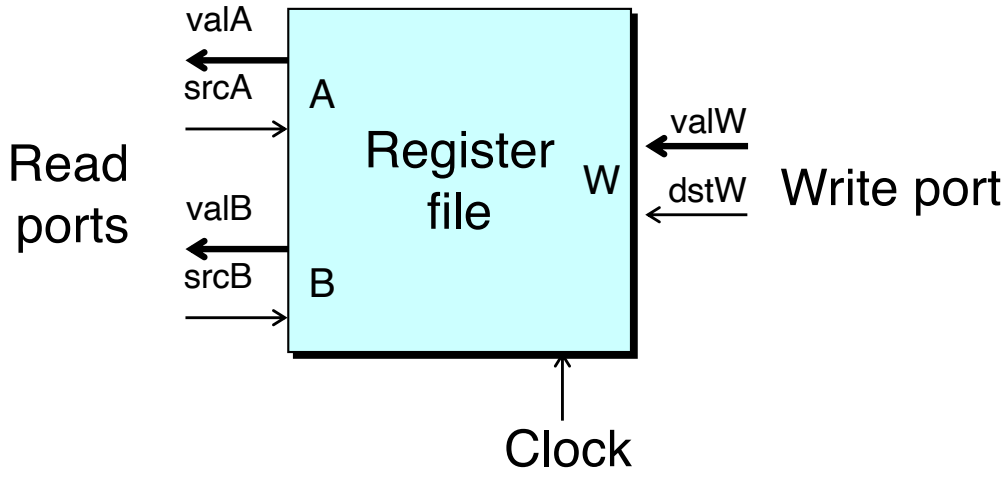
State Machine Example



- Accumulator circuit
- Load or accumulate on each cycle
- (Notice effect of clock on Out)

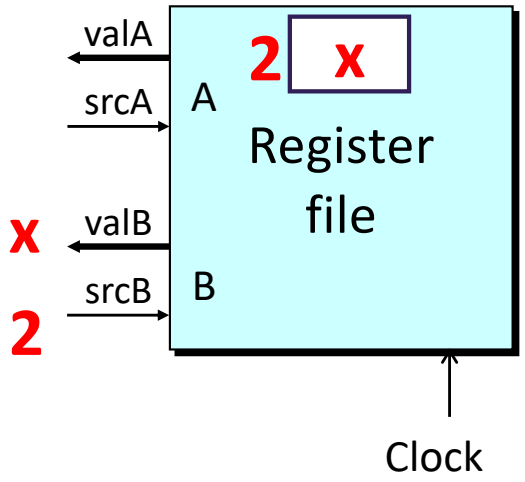


Random-Access Memory

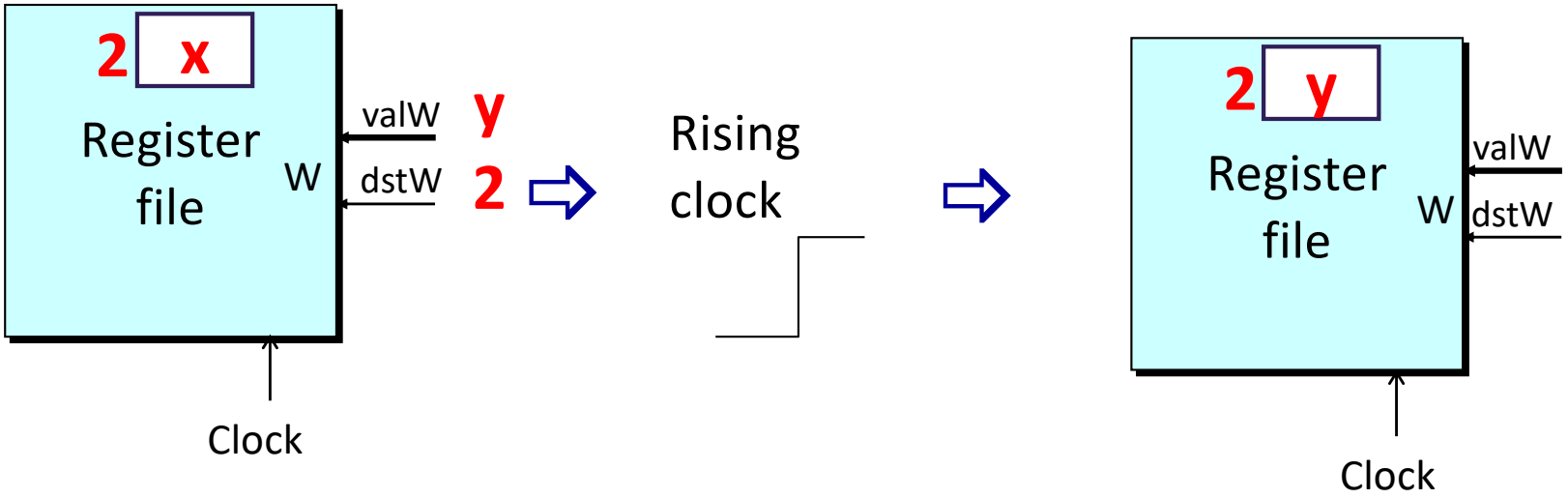


- Stores multiple words of memory (has internal storage)
  - Address input specifies which word to read or write
- Register file
  - Holds values of program registers (`%rax`, `%rsp`, etc.)
  - Register identifier serves as address
    - ID 15 (0xF) implies no read or write performed
- Multiple Ports
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output

Register File Timing



- Reading
  - Like combinational logic
  - Output data generated based on input addr
    - After some small delay
- Writing
  - Like hardware register
  - Update only as clock rises

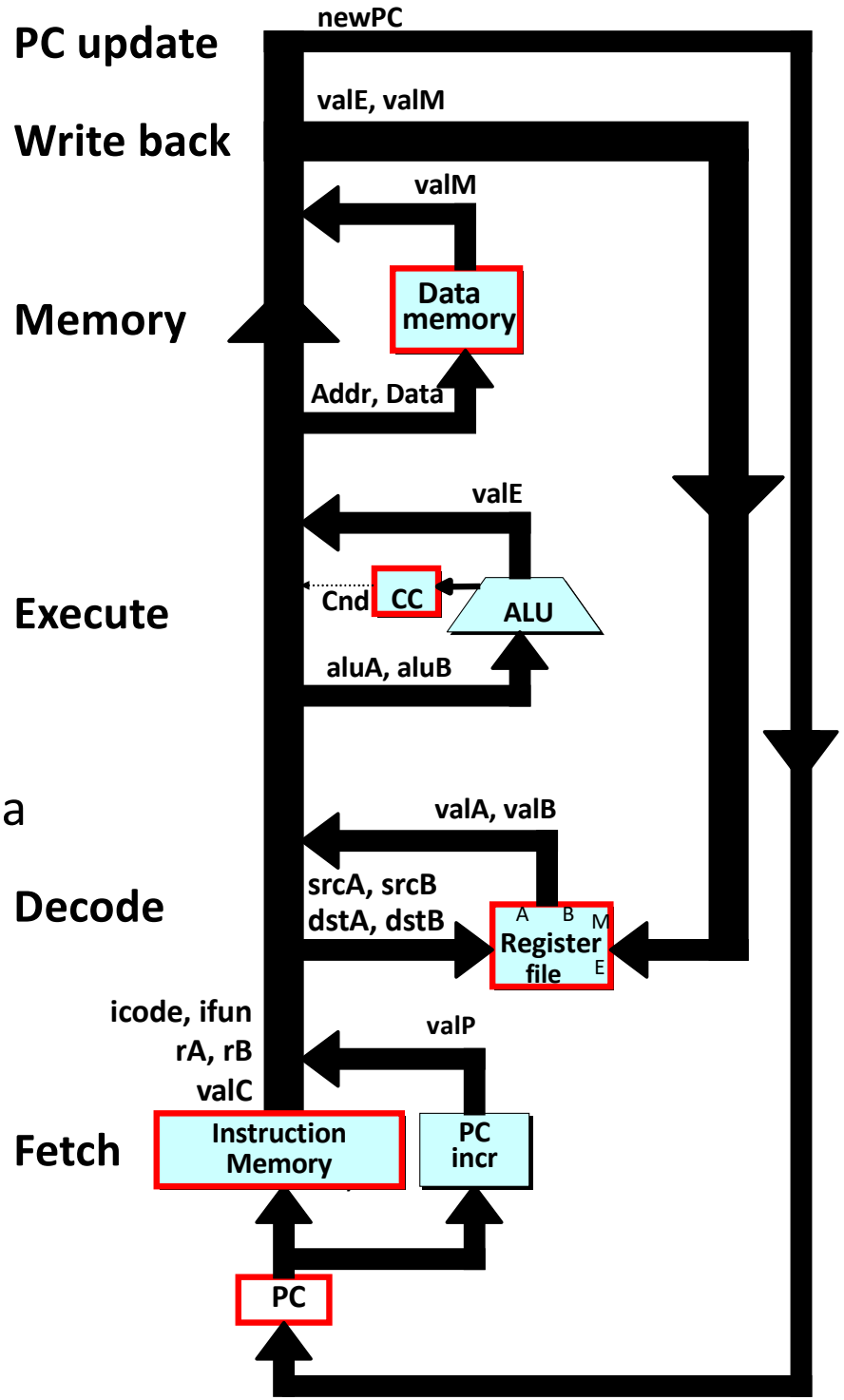


Ch 4.2 Summary

- Computation
  - Performed by combinational logic
  - Computes Boolean functions
  - Continuously reacts to input changes
- Storage
  - Registers (hardware)
    - Hold single words
    - Loaded as clock rises
  - Random-access memories
    - Hold multiple words
    - Possible multiple read or write ports
    - Read word when address input changes
    - Write word as clock rises

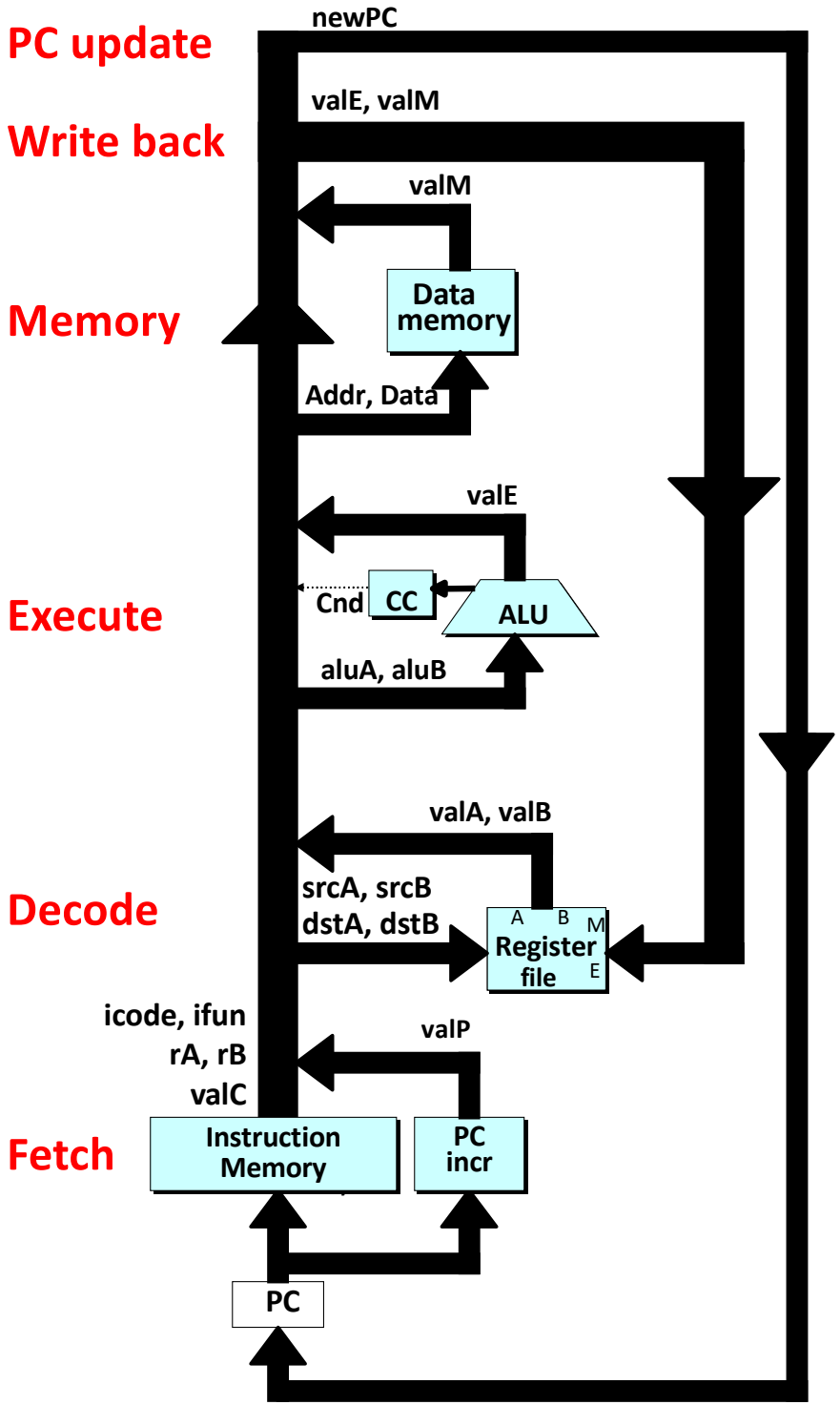
Moving on to Ch 4.3: SEQ Hardware Structure

- State (red boxes)
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow (arrows)
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

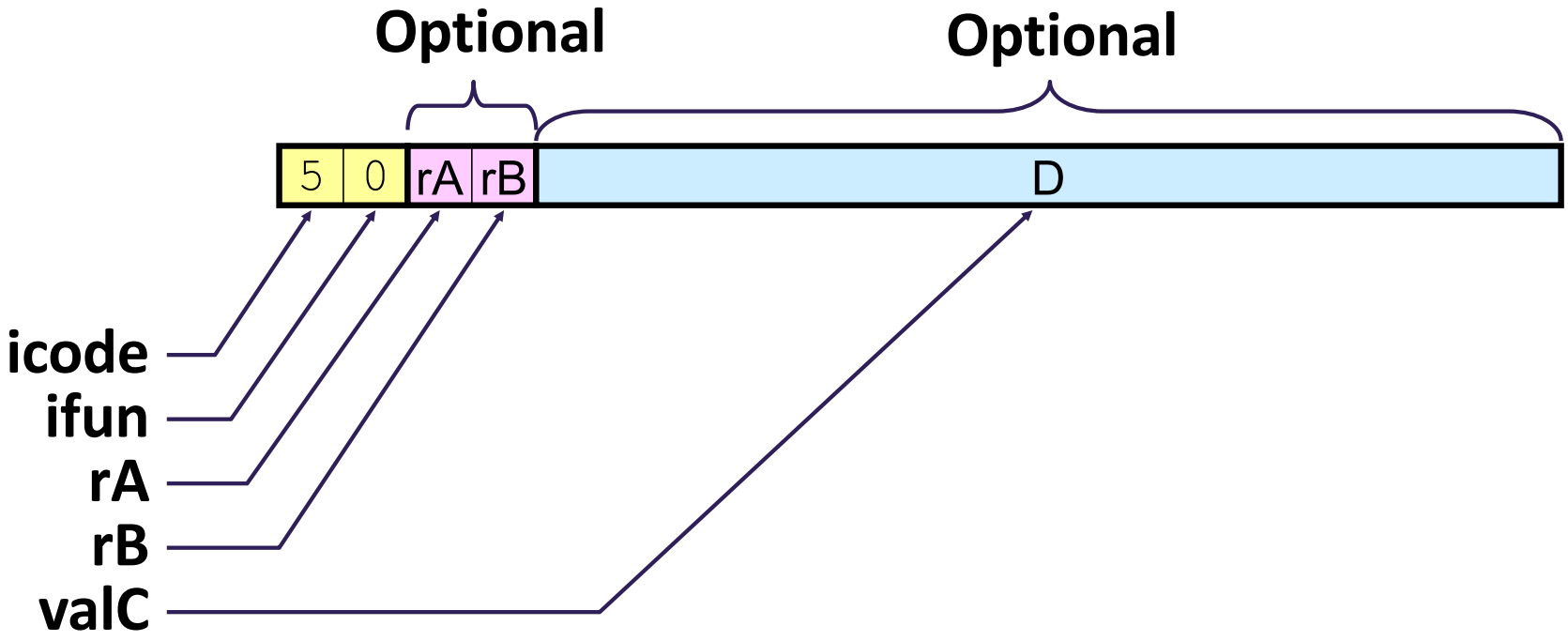


SEQ Stages

- Fetch
  - Read instruction from instr memory
- Decode
  - Read program registers from instr
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers to register file
- PC update
  - Update program counter

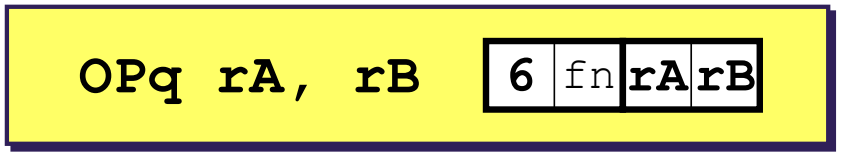


Instruction Decoding



- Instruction Format
  - Instruction byte — icode:ifun
  - Optional register byte — rA:rB
  - Optional constant word — valC

Executing Arithmetic/Logical Operation



- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register rB
- PC Update
  - Increment PC by 2 bytes

Stage Computation: Arith/Log Ops

	OPq rA, rB	(R[rB]=R[rB] OP R[rA])
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valP ← PC+2	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB OP valA	Perform ALU operation
	Set CC	Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
- Note: M_1 indicates we're accessing 1 byte of memory (usually instruction memory), while M_8 indicates we're accessing 8 bytes (usually data memory)

Executing `rmmovq`

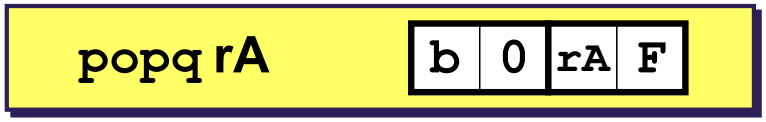
- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory (rB+D)
- Write back
  - Do nothing
- PC Update
  - Increment PC by 10

Stage Computation: `rmmovq`

	rmmovq rA, D(rB)	(Move R[rA] to M_8[R[rB]+D])
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- Use ALU for address computation

Executing popq



- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 8
- Memory
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

Stage Computation: popq

	popq rA	(Move M_8[R[%rsp]] to R[rA])
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valP ← PC+2	Compute next PC
Decode	valA ← R[%rsp]	Read stack pointer
	valB ← R[%rsp]	Read stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read from stack
Write back	R[%rsp] ← valE	Update stack pointer
	R[rA] ← valM	Write back result
PC update	PC ← valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer