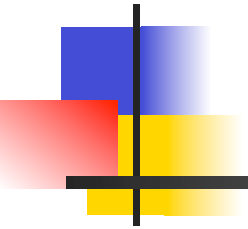
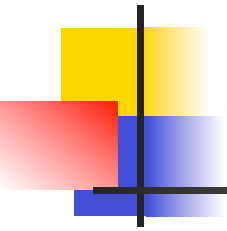


Processor Architecture I

Alexandre David



Overview

- Introduction: from transistors to gates.
  - and from gates to circuits.
- 4.1
- 4.2
- Micro+macro code.

Evolution of Computers

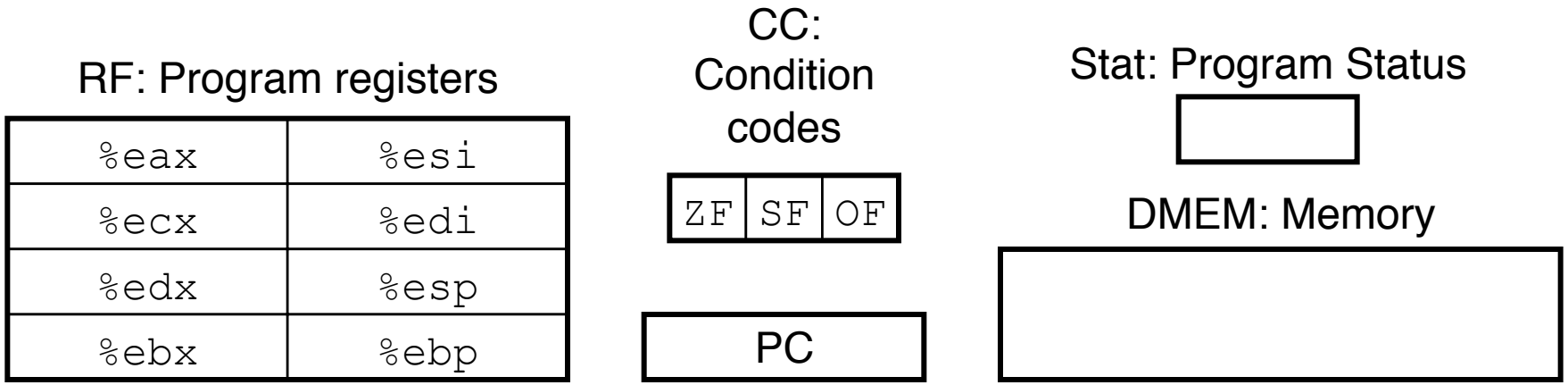
- Early systems
  - CPU (*central* processing unit) controlled the entire system
  - Responsible for I/O, computations, …
- Modern computers
  - Decentralized architecture
  - Processors distributed (I/O)
  - CPU still controls other processors

General Purpose CPU

- Very complex because
  - designed for wide variety of tasks – multiple roles
  - contains special purpose sub-units
  - ex: core i7 has 731M transistors
  - supports protection and privileges (OS/applic.)
  - supports priorities (I/O)
  - data size (32/64-bit registers)
  - high speed – parallelism = replication

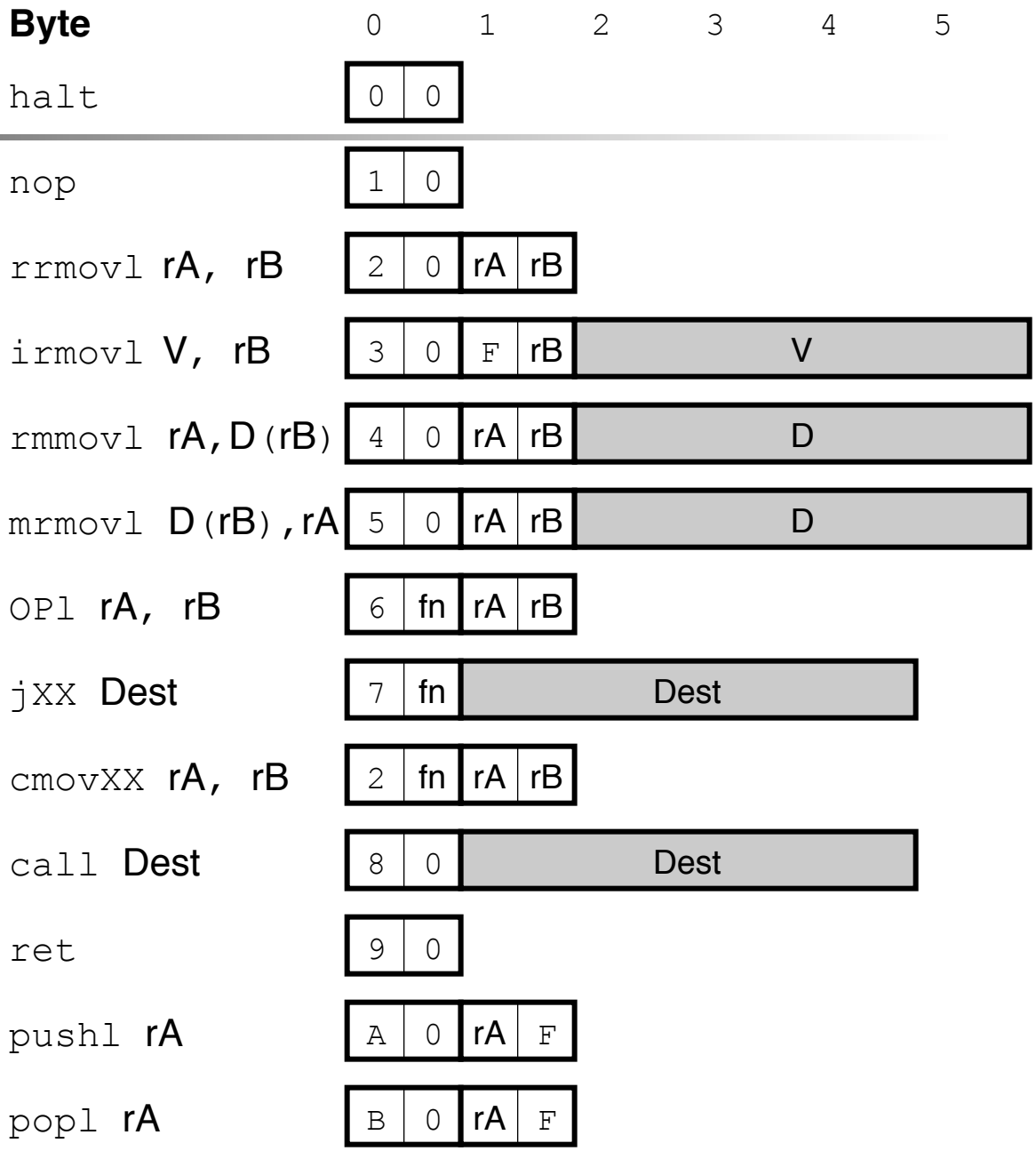
CPU Visible State

- Visible for ISA, used by compiler (& assembler) – there may be other registers etc… that depend on the CPU generation.
- Registers (classify them), condition codes, status & memory.
- Memory=array of bytes (abstraction).



ISA

- Classify:
  - load/store
  - r/i/m operands
  - arithmetics
  - jumps
  - (cmov)
  - call/return
  - stack
- Encoding:
  - opcode+ operands



ISA – Notes

- No memory → memory transfer.
- No imm → memory transfer (x86 can do it).
- Restricted operators (add, sub, and, xor).
  - Only register – register operands.
  - Typical of load/store architectures (also RISC).
- Conditional jumps depend on combinations of flags.
  - Similar for conditional move.
- Call, ret, pop, and push implicitly modify the stack (& stack pointer).

Encoding

PP4.1
PP4.2

- 1 byte encoding = code + function.
  - **Unique combination for every instruction**.
- Register operands have a unique identifier.
  - eax:0, ecx:1,… none:F
  - “none” important for design

Operations

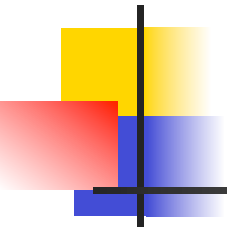
addl	6	0
subl	6	1
andl	6	2
xorl	6	3

Branches

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

Moves

rrmovl	2	0
cmovle	2	1
cmovl	2	2
cmove	2	3
cmovne	2	4
cmovge	2	5
cmovg	2	6



Status Code

- State of the processor
  - AOK normal
  - HLT halted
  - ADR invalid address
  - INS invalid instruction

Y86 – X86

- Y86 simplified model for X86.
- Code is similar, except for
  - move instructions
  - restrictions
- → may need more instructions
  - not important, we abstract from that.
  - Reason on Y86 level, exercise with both (simulator Y86, gcc for X86).

Y86 Assembly

- Instructions as described with registers.
- Assembly directives.
  - Where to put the code (.pos).
  - Align the code (.align).
  - Declare data (.long).
  - X86 has more.
- Label declarations (used for jump offsets).
- Assembled into bytes.
- Y86 interpret the bytes.

Logic Design

- How to implement the hardware to recognize the instruction codes.
- Logic that
  - reads bytes,
  - interprets bytes (switch),
  - performs operations,
  - updates state.
- Transistors → gates → functions & blocks.
- Processor design at block level.

Background

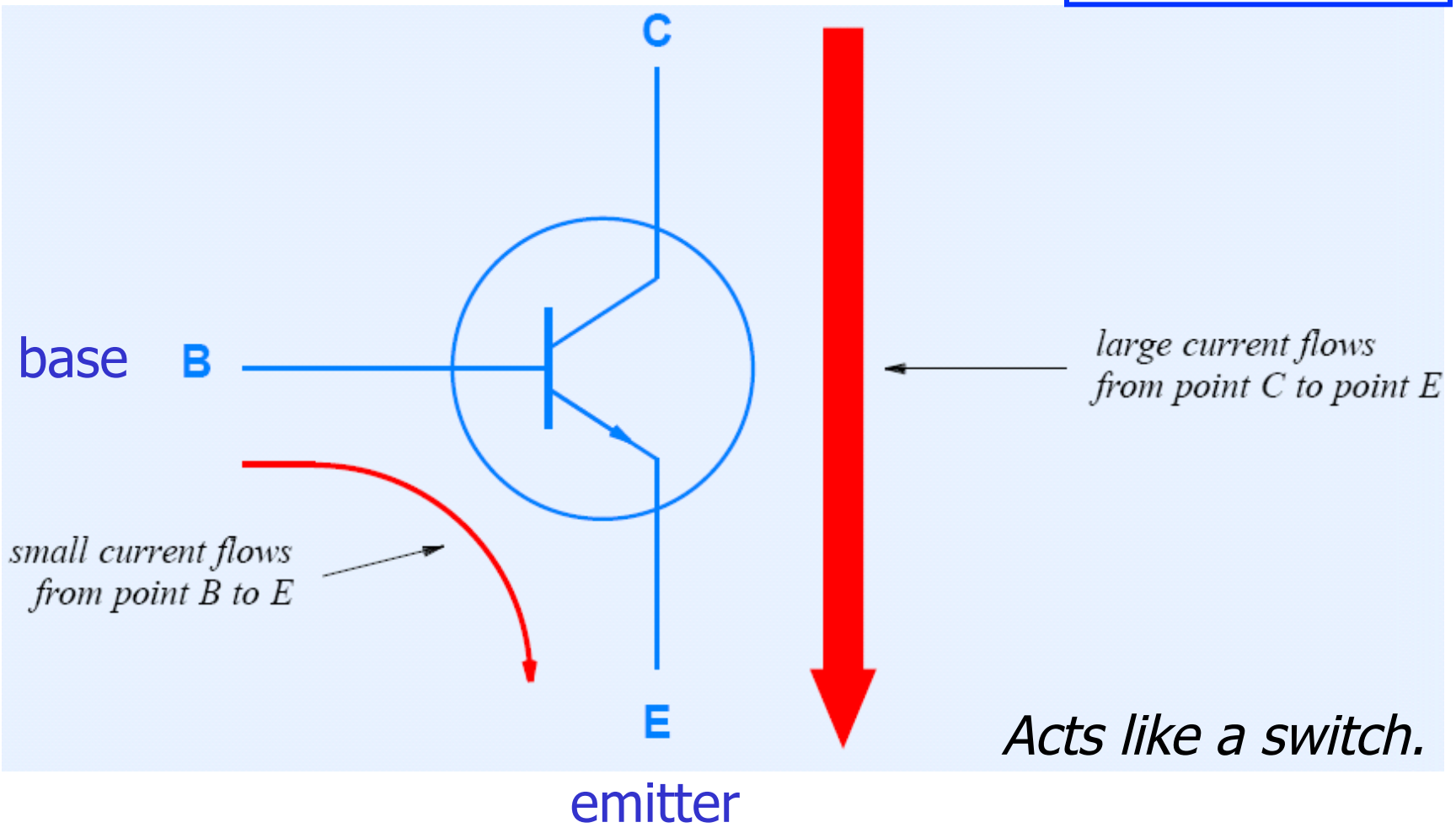
- Voltage: difference of potentials.
  - Vcc – ground (=0).
  - Volts (V)
- Current: flow of electrons.
  - Amperes (A)
- Ohm's law: $U = RI$
- Dissipated power: $P = UI = U^2/R$

Typical Chips

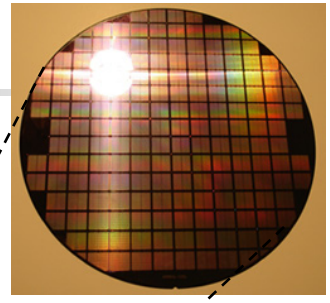
- Operate on low voltage (5V, less for processors) – see power dissipation.
- Always 2 lines
  - ground (0V)
  - power (5V)
- Diagrams usually omit ground and power.

Transistor

Basic building block of digital circuits



How Are They Made?



pure silicon
with perfect
crystalline structure

photo-lithography

cut

slice into wafers

doping materials (N/P)
metal (copper/gold)

connect &
package

sand

Boolean Algebra

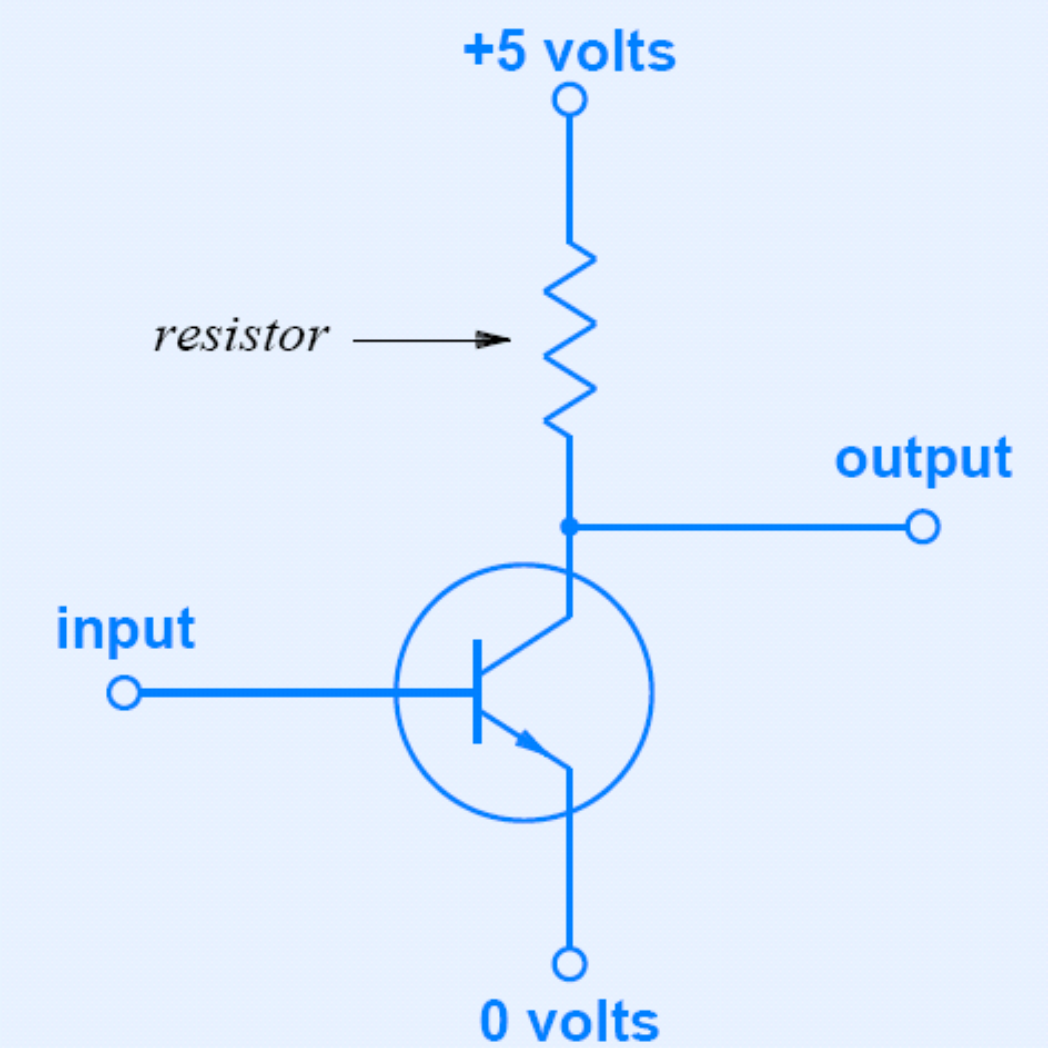
- Mathematical basis for digital circuits.
- From boolean functions to gates.
- Basic functions: and, or, not.
- In practice, cheaper to have nand & nor.

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

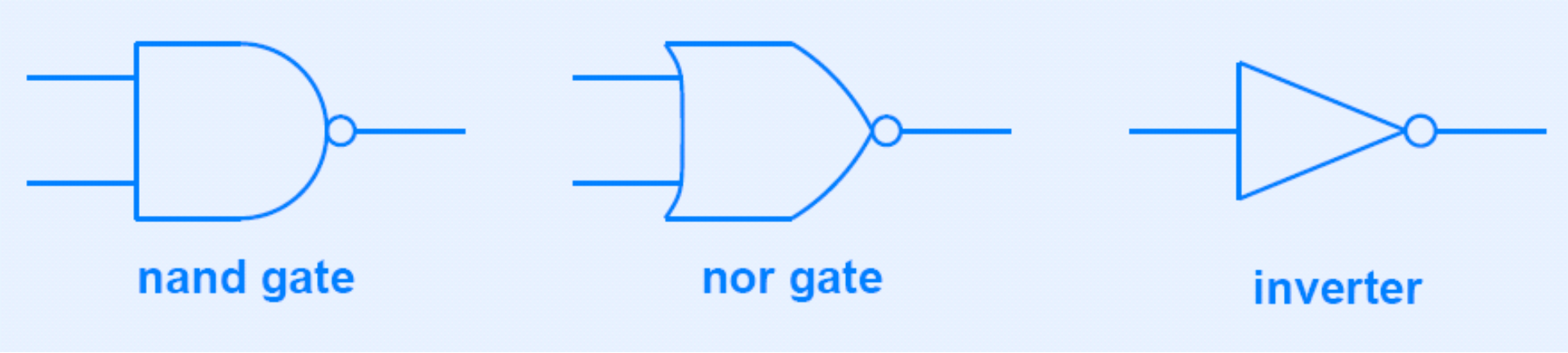
A	not A
0	1
1	0

Example: Not



Gates

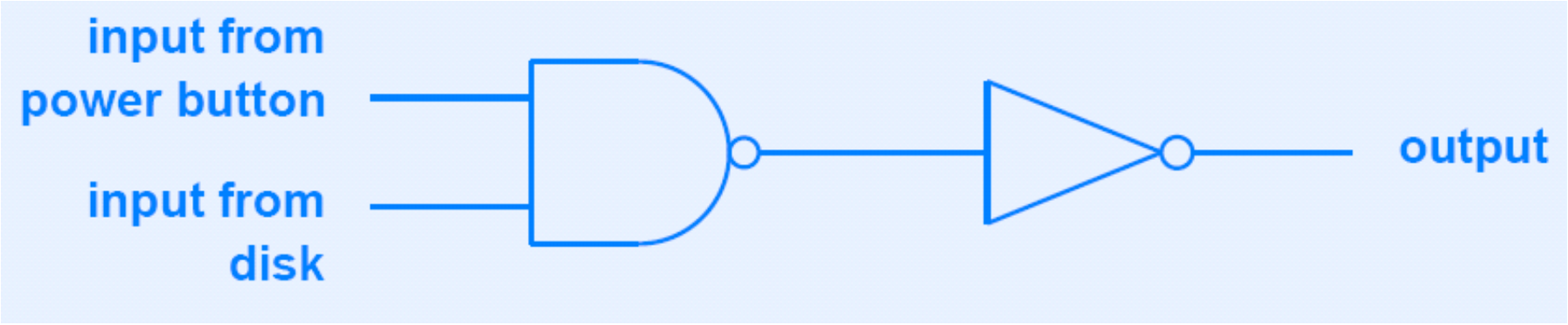
- Primitive boolean functions.
- Level of abstraction on integrated circuits.



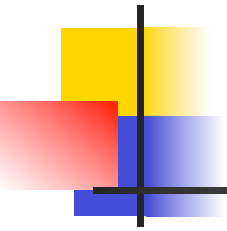
Symbols used in circuits.

Logic Gate Technology

- Transistor-transistor technology (TTL)
  - connect directly gates together to form boolean functions



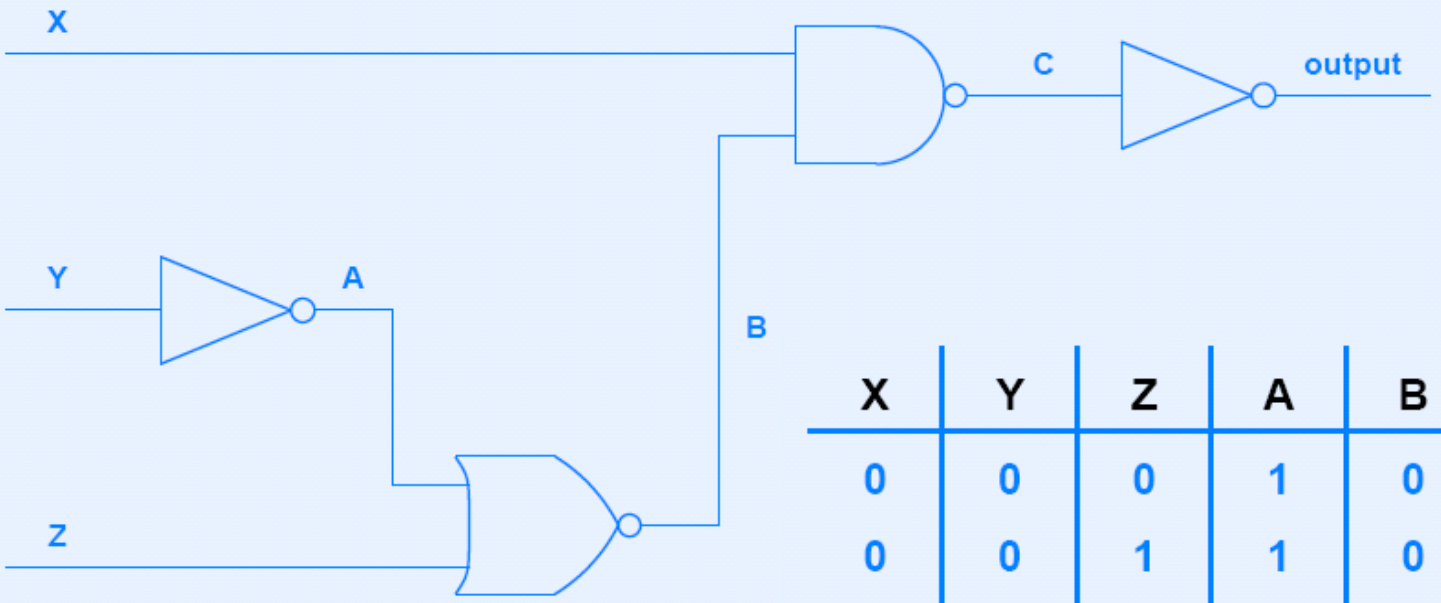
and function



Design of Functions

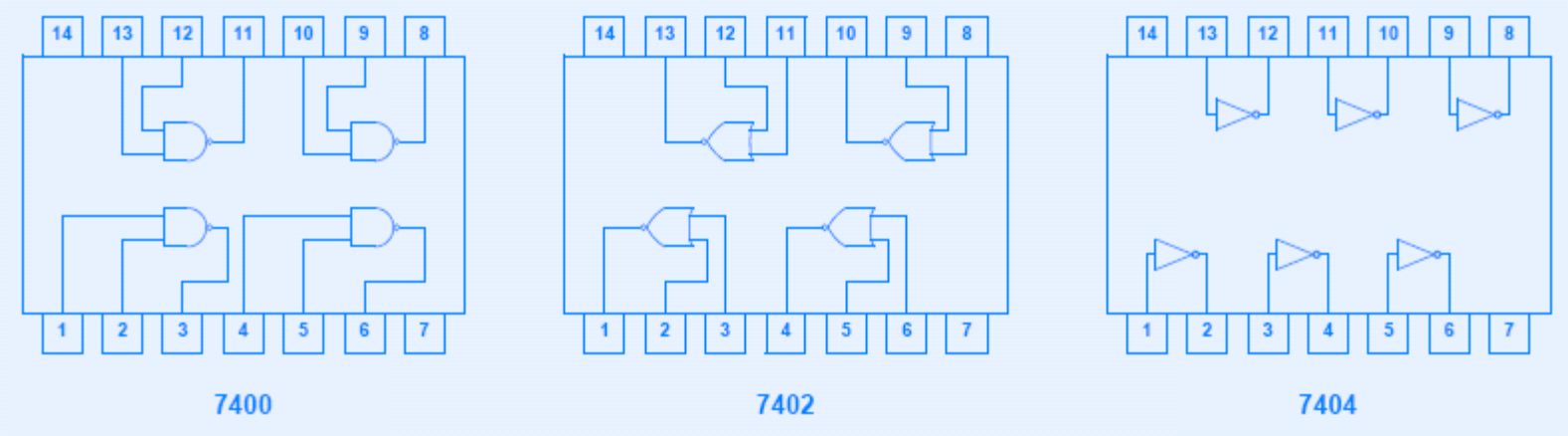
- Find a boolean expression that does what you need
  - and feed it to a tool that optimizes it to minimize the number of gates.
- Come up with the truth table of your function
  - which is converted to a boolean function.

Truth Table



X	Y	Z	A	B	C	output
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	1	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	1	0	1	0
1	1	0	0	1	0	1
1	1	1	0	0	1	0

Combinatorial Circuits

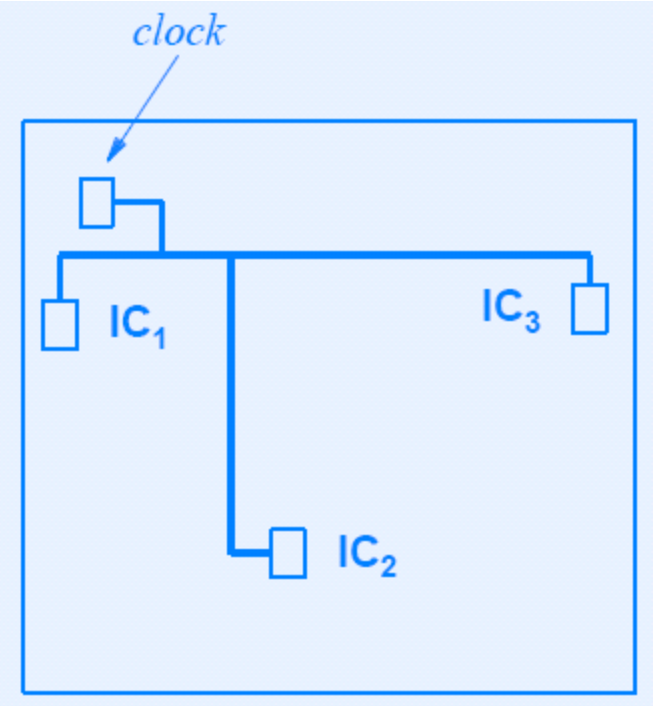


- Outputs = function(inputs)
  - change outputs only when inputs changes
  - need states to perform sequences of operations without sustained inputs
    - maintain states
    - use a clock

Practical Concerns

- Power
  - consumption: how to feed
  - dissipation $P=CFV^2$ (C: capacitance, F: frequency) how not to burn
- Timing – gates need time to settle.
- Clock synchronization.
  - Update upon rise or fall of clock signal.

Clock Skew



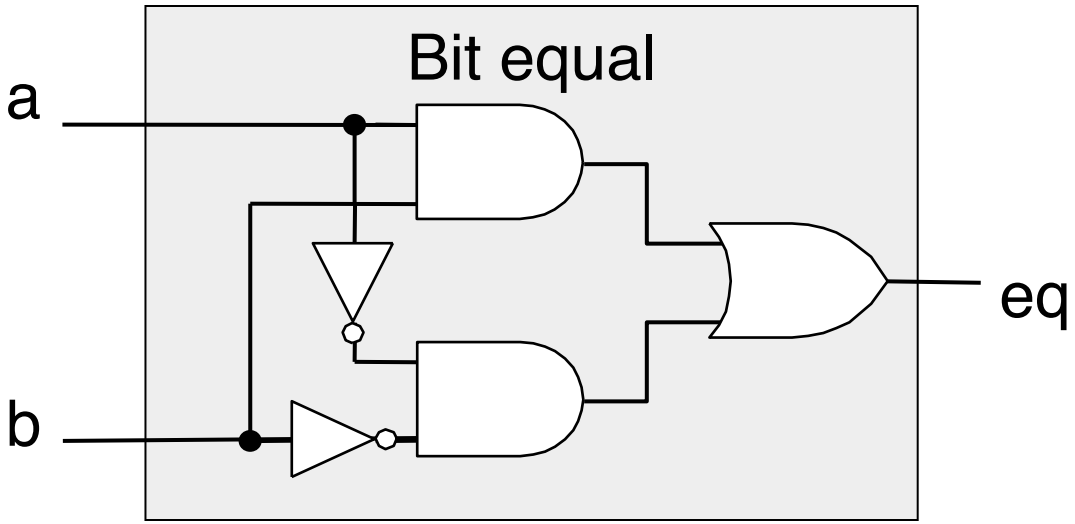
Signals need time to propagate.
Local clocks are used on larger systems → need to synchronize them.
The speed of light is too slow.

Logic Design & HCL

- Design logic with gates – but not one by one and not manually!
  - Use an adapted language for that. Here HCL (hardware control language) for educational purposes.
  - C-*like* language to express boolean formulas.
  - Combinatorial circuits built out of these formulas.
  - Acyclic network of gates: signal propagates from inputs to outputs → boolean functions.

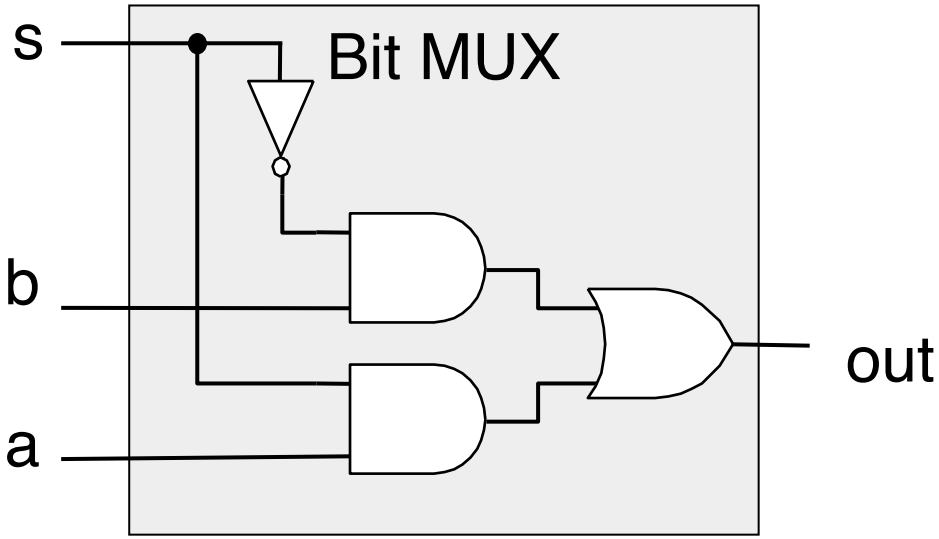
Example

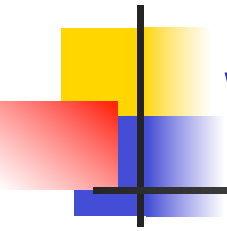
- bool eq = (a && b) || (!a && !b);



Multiplexor

- Function: choose an input signal depending on a selection signal.
  bool out = (s && a) || (!s && b);
  - Select results, functions, etc…





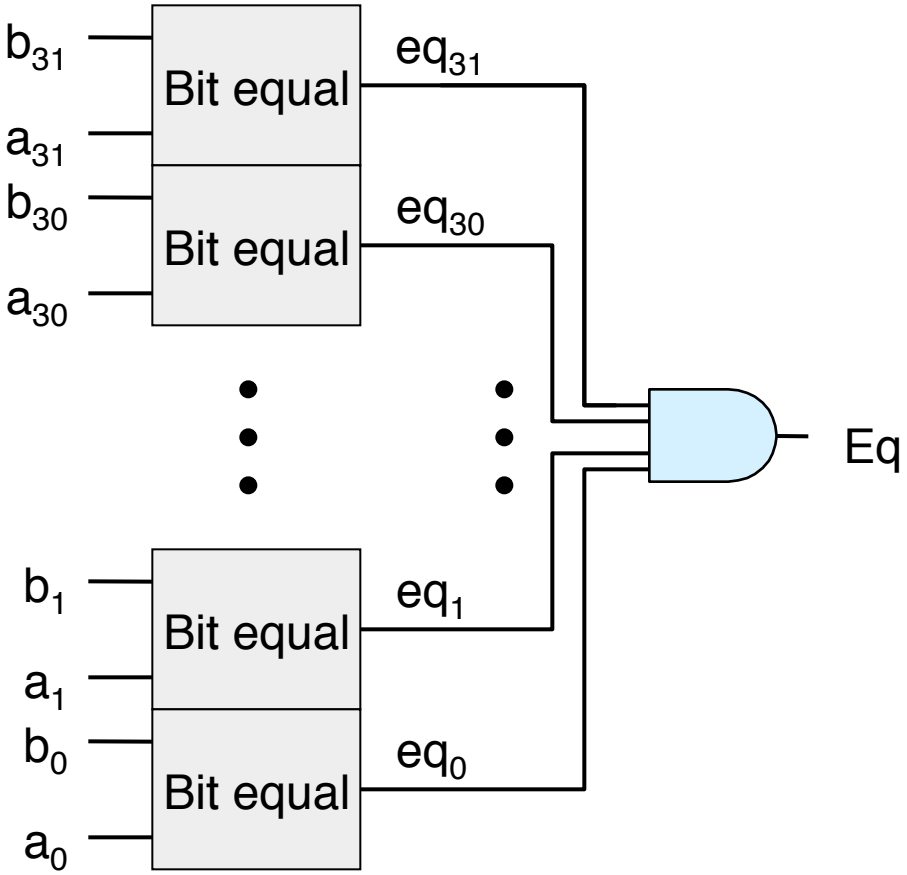
Word-Level Combinatorial Circuits

- Operations defined at word level (~integers).
  - Treat groups of bits together.
  - Define functions at word-level.

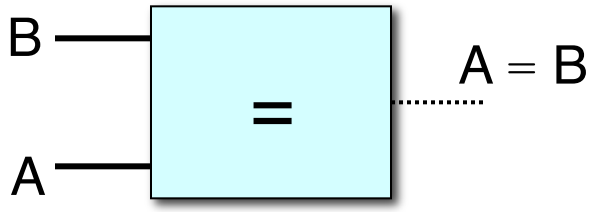
Example: Equality Test

- bool Eq = (A == B);

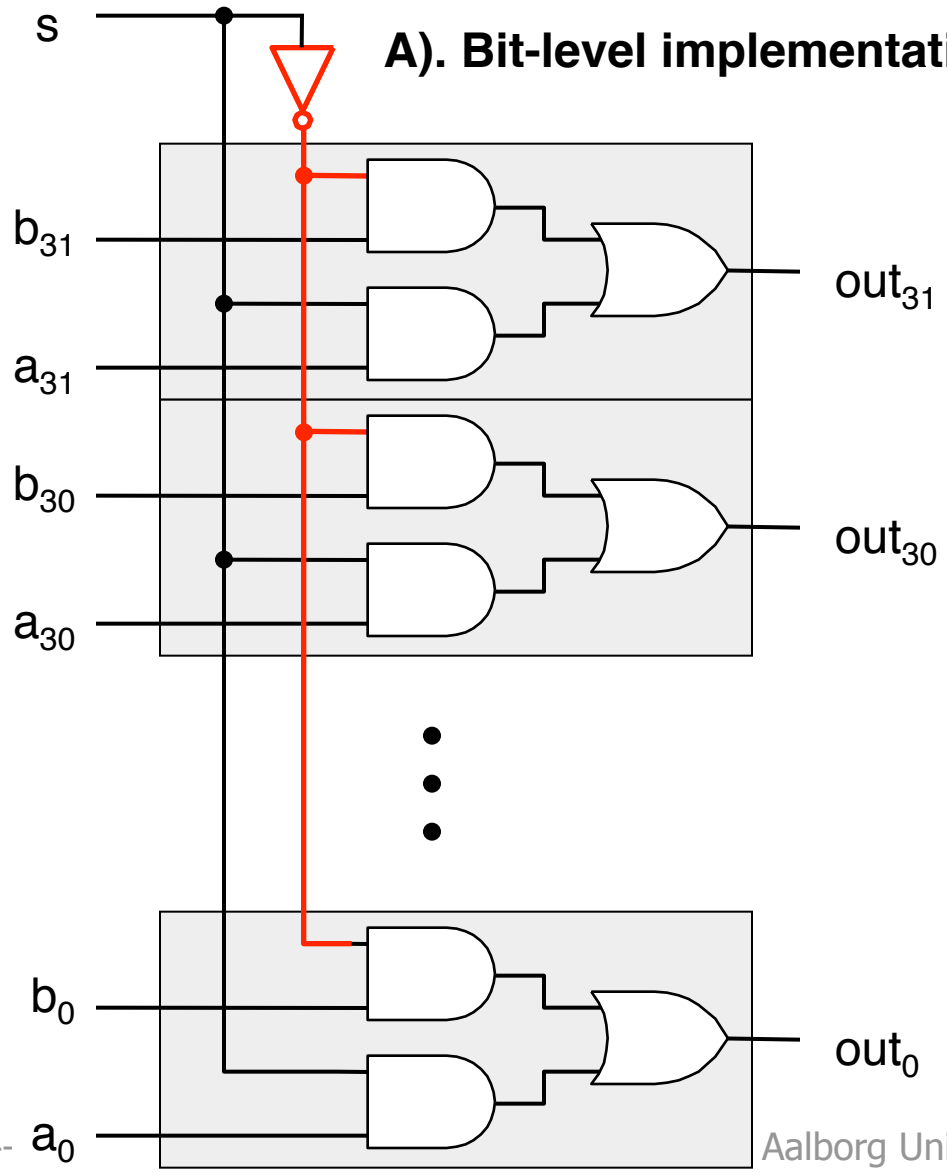
A). Bit-level implementation



B). Word-level abstraction

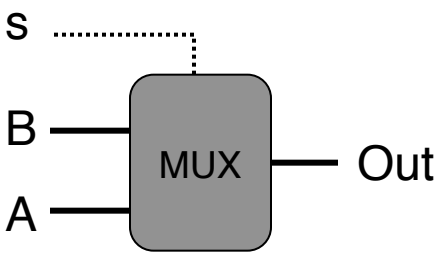


Multiplexor At Word-Level



A). Bit-level implementation

B). Word-level abstraction

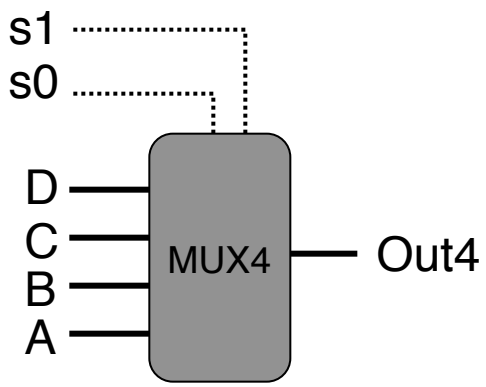



```
int Out = [
   s : A;
   1 : B;
];
```

Use: Select

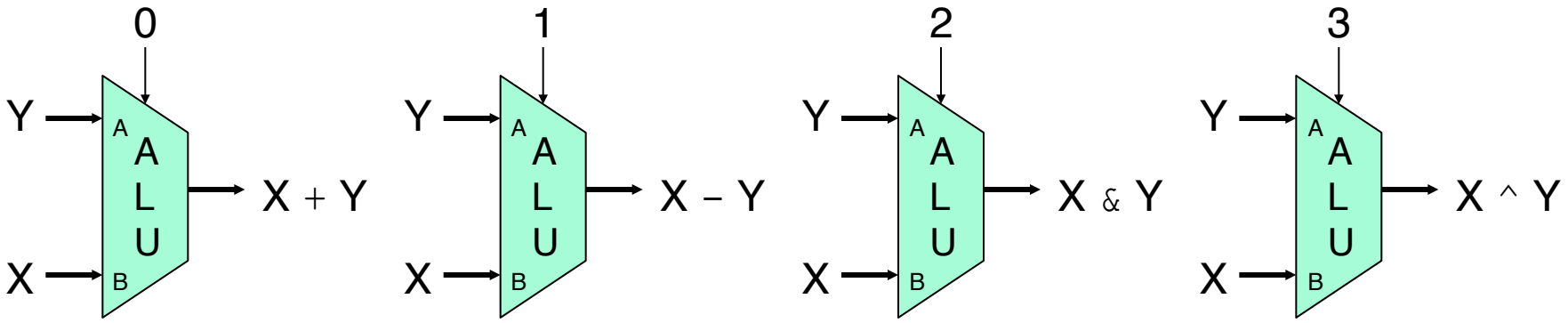
```
int Out4 = [
    !s1 && !s0 : A; # 00
    !s1        : B; # 01
    !s0        : C; # 10
    1          : D; # 11
]
```

- Simplified select.



ALU

- 2 operand inputs + 1 control input.
  - Operands X and Y.
  - Control selects operation.
  - Same principle as select, we abstract from the exact design.

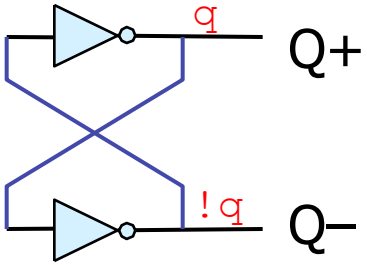


Memory & Clocking

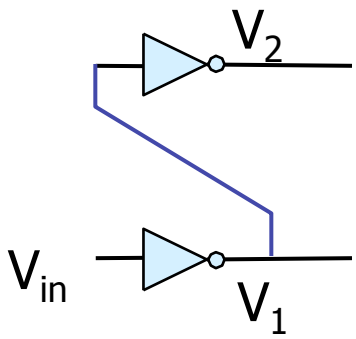
- Memory stores states.
  - Functions only propagates signals.
  - Memory implemented as flip-flop-like circuits. Have feedback loops to “keep” bits.
  - Registers (hardware or program).
- Clocks synchronize when to update.
  - Between updates, signals propagate.
  - Clock signal rises → registers are updated.

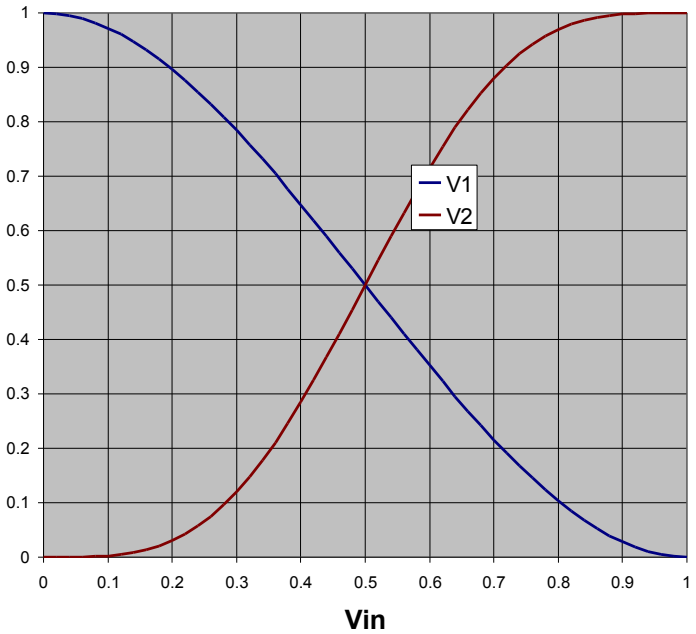
Storing 1 Bit

Bistable Element



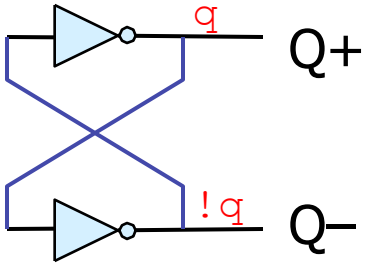
q = 0 or 1



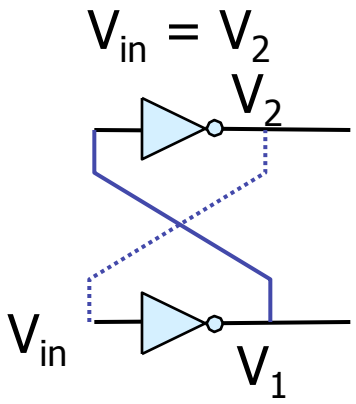


Storing 1 Bit (cont.)

Bistable Element

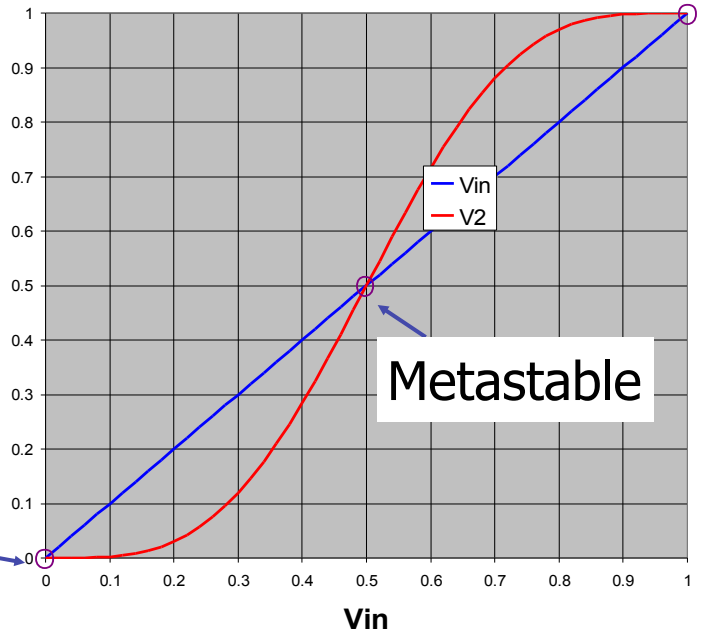


q = 0 or 1

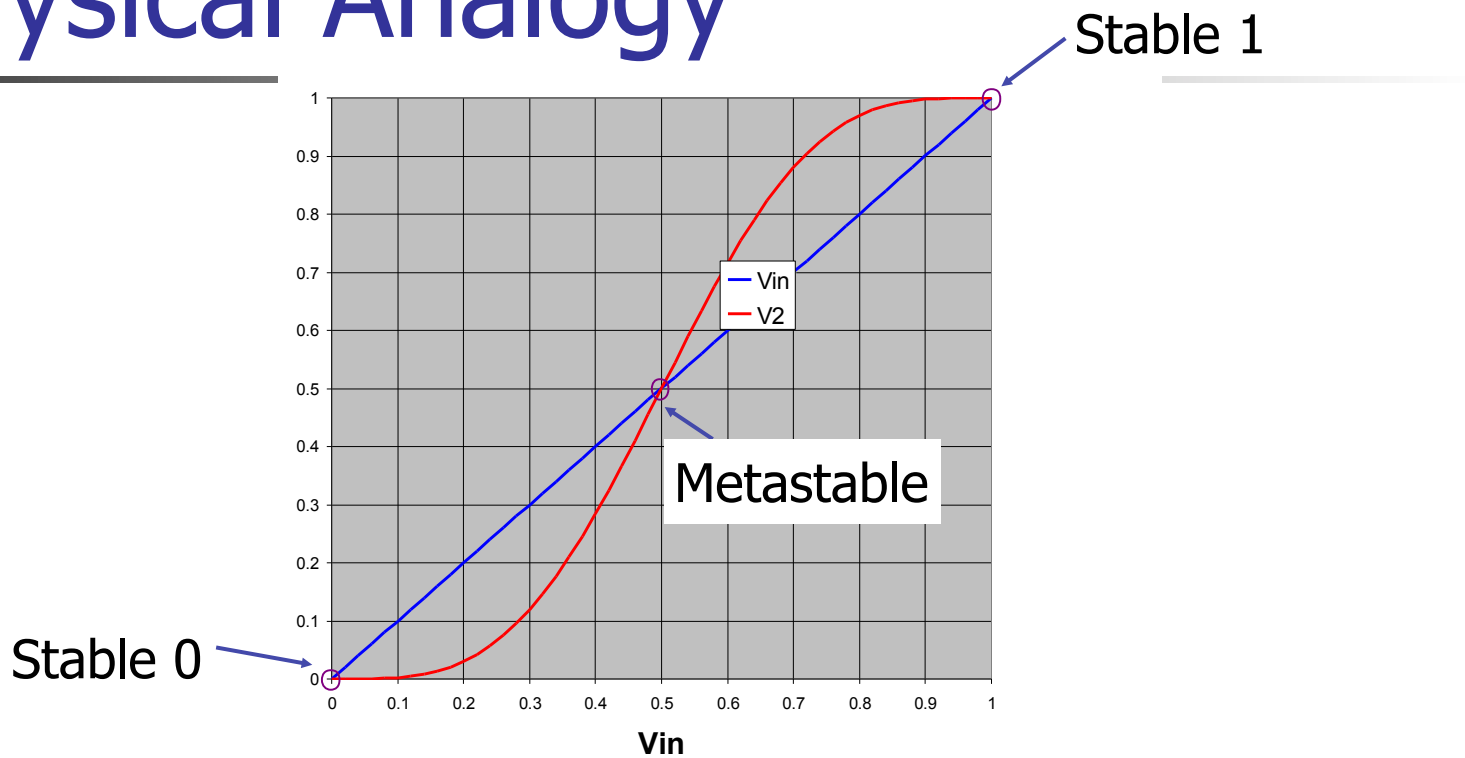


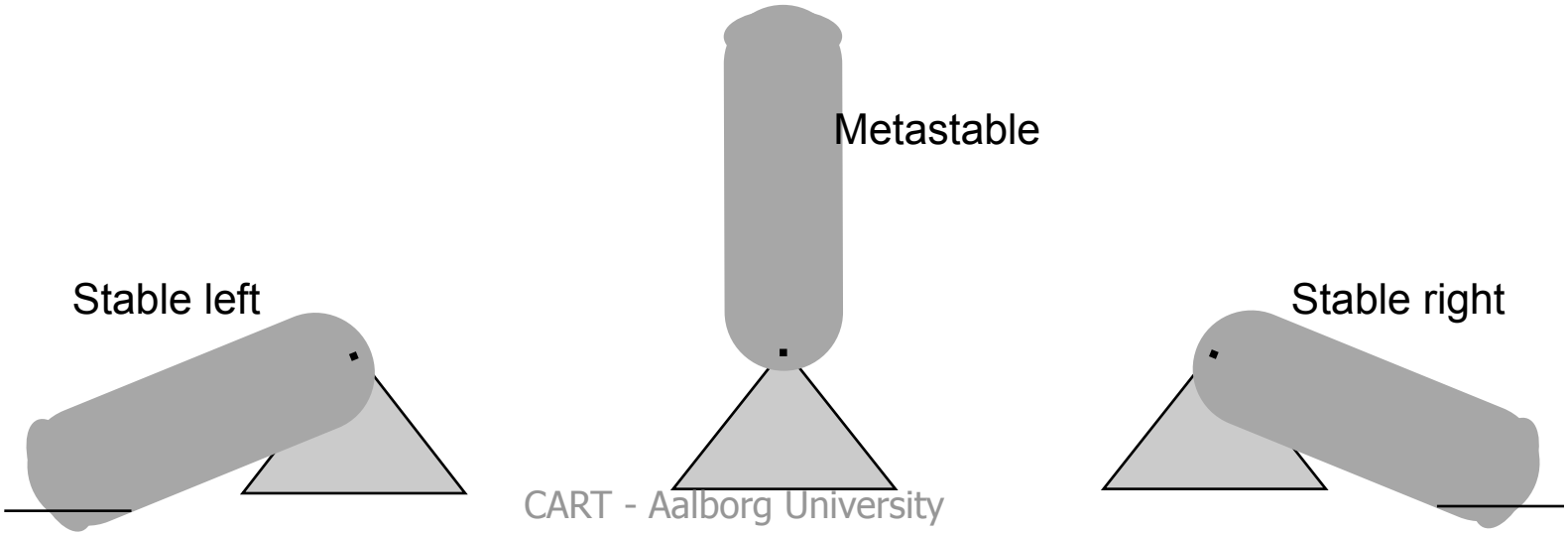
Stable 1

Stable 0



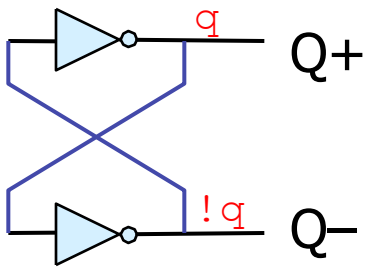
Physical Analogy





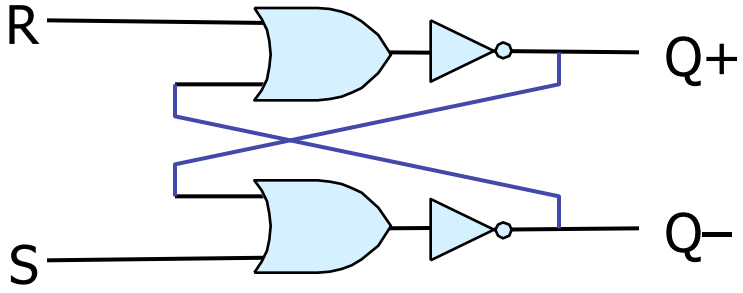
Storing and Accessing 1 Bit

Bistable Element

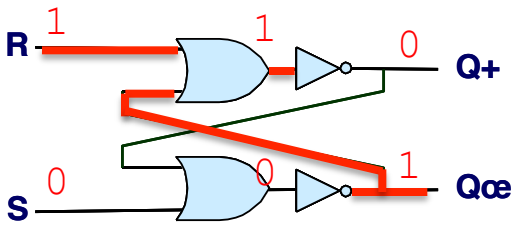


q = 0 or 1

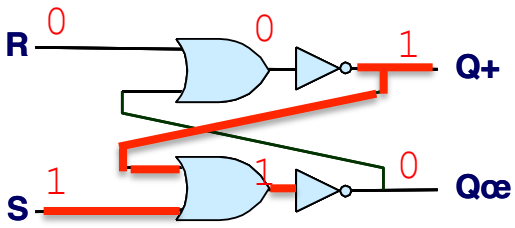
R-S Latch



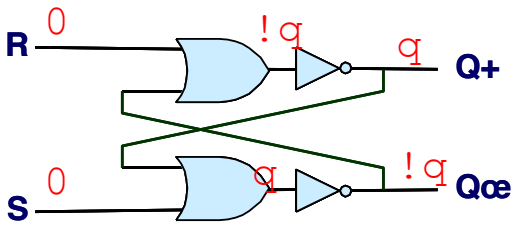
Resetting



Setting

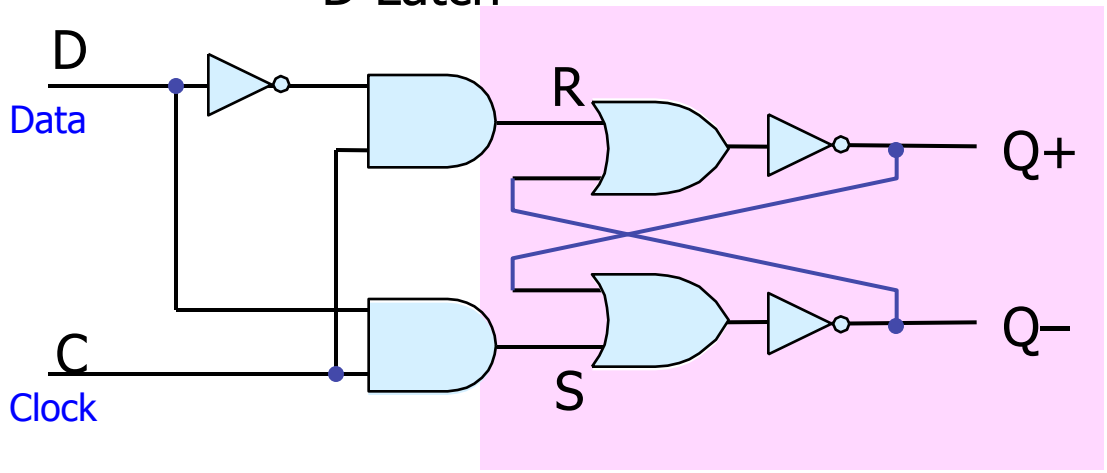


Storing

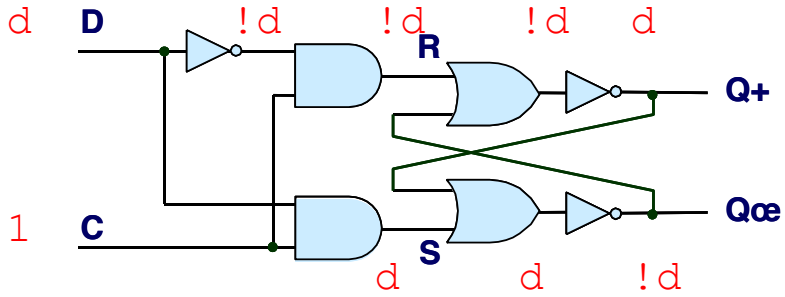


1-Bit Latch

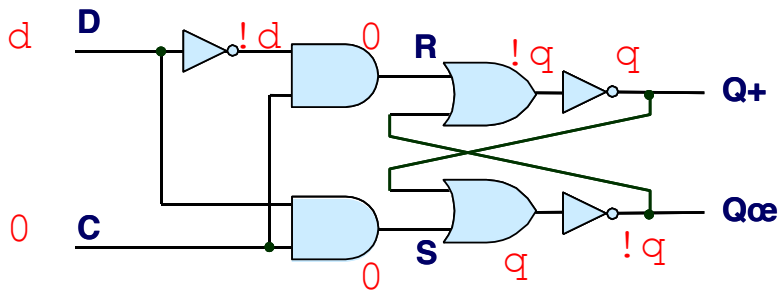
D Latch



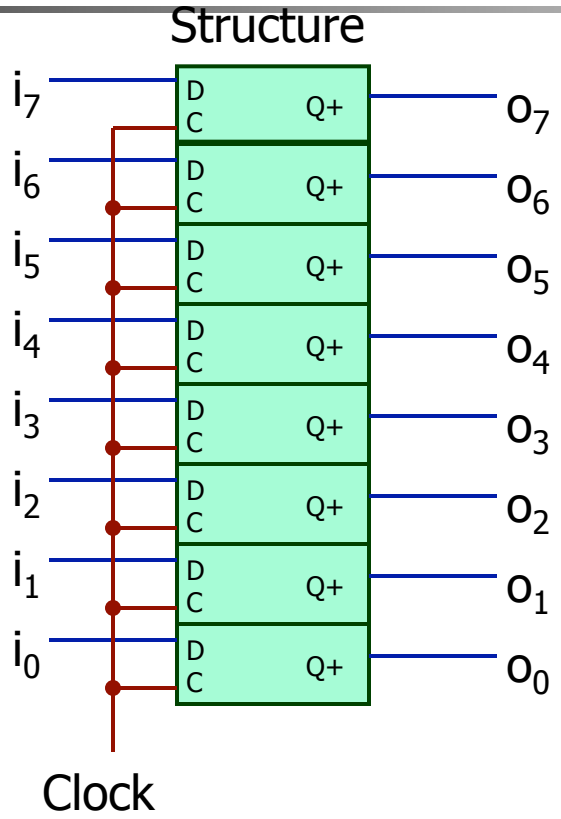
Latching

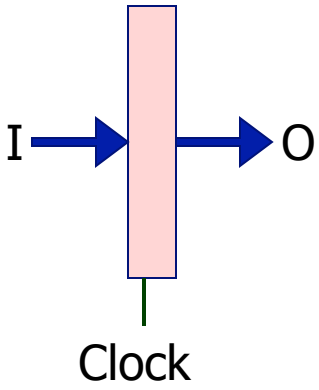


Storing



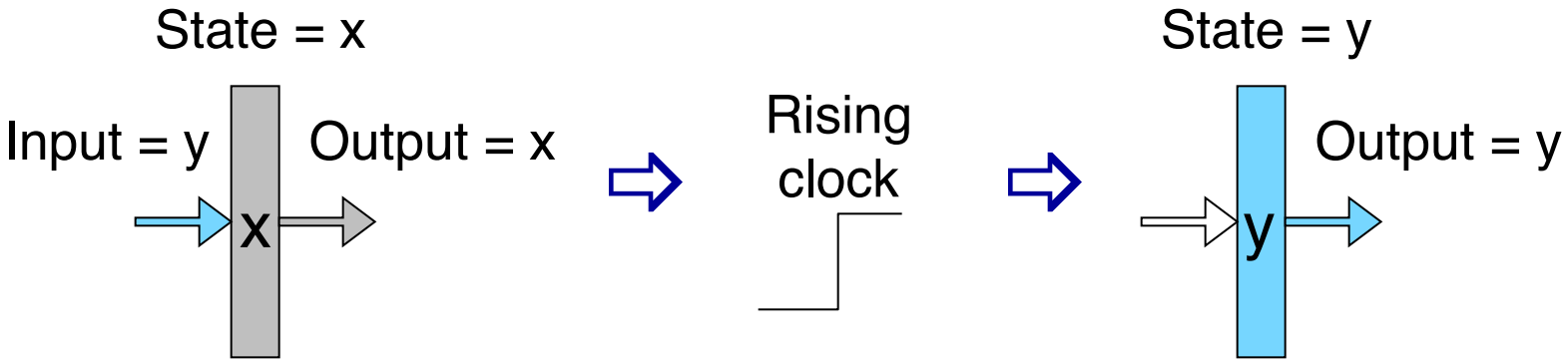
Registers



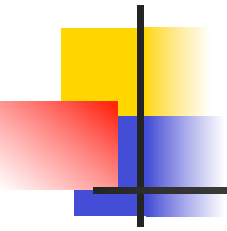


- Stores word of data
  - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

Clock Synchronization

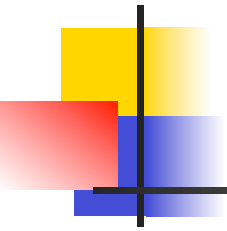


- Register operations are synchronized.
  - Stores data bits.
  - For most of time acts as barrier between input and output.
  - As clock rises, loads input.



Register File

- Set of program registers.
  - Local and fast access storage.
  - Small.
  - Fixed size (machine word).



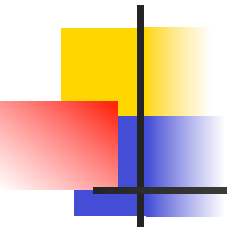
Memory

- Abstract & simplified model.
- Simple array of byte, no hierarchy.
  - We'll see later the hierarchy & virtual memory system.

Micro/Macro-code

Complement

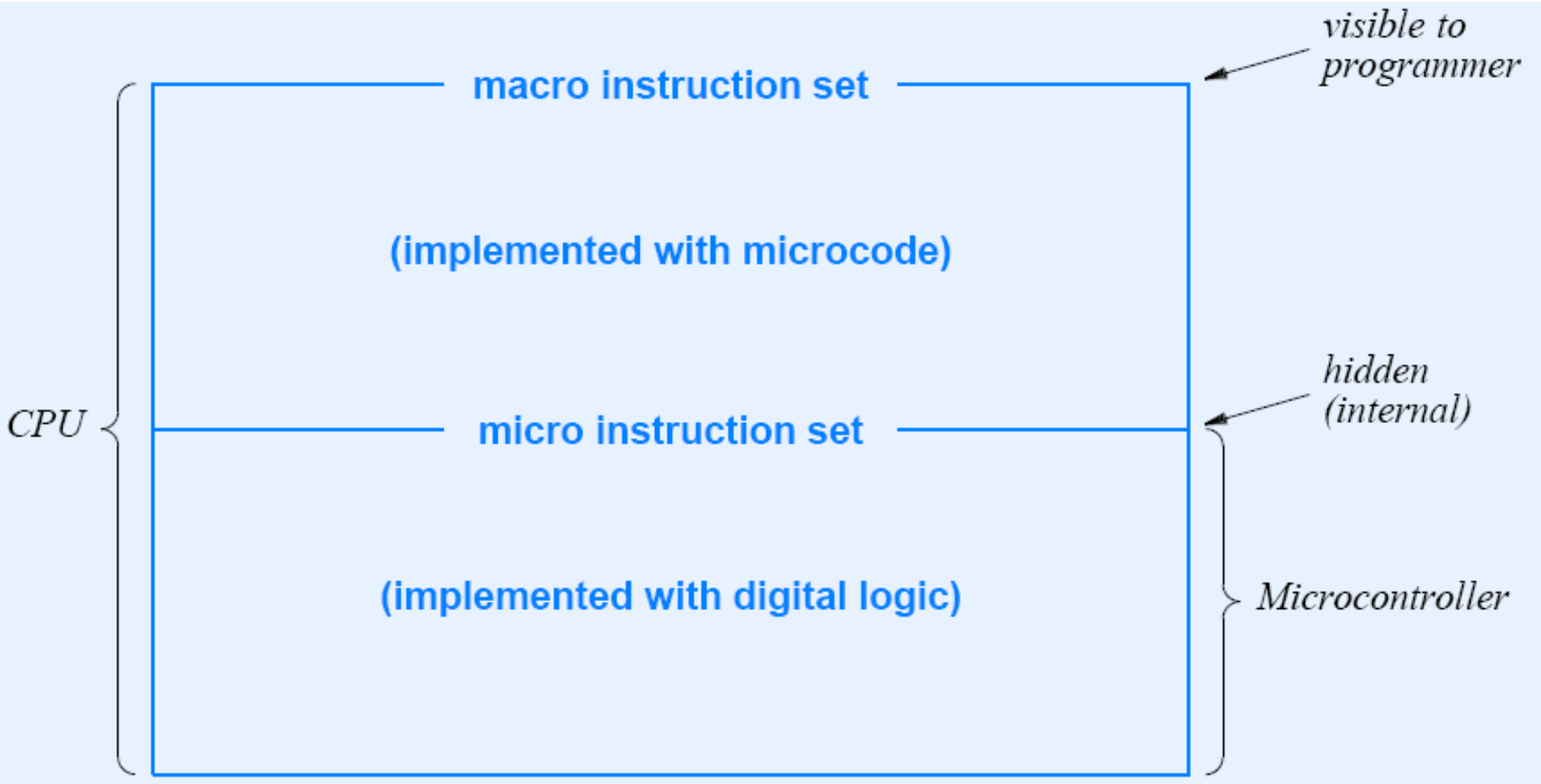
- We'll focus on gate/logic design.
  - = simplified model.
- Reality is more complex.
  - Design is at a higher level.
  - We'll see hints of correspondence to micro-code.

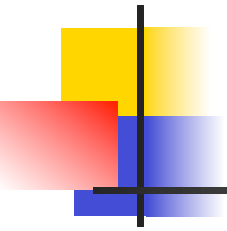


Microcode

- How to implement complex CPU?
  - Program the complex instructions.
  - Visible machine language = macro instruction set.
  - Internal language = micro-code.
  - Microcontroller inside CPUs that decode and execute macro-instructions.
    - RISC
    - Processors are all RISCs in the end.
  - Key: Easier to write programs with micro-code than to build hardware from scratch.

Microcode


visible to programmer
macro instruction set
(implemented with microcode)
hidden (internal)
CPU
micro instruction set
(implemented with digital logic)
Microcontroller



Data and Register Sizes

- Size of visible register may differ from size of internal registers.
  - Ex: Could implement 32-bit instruction set on a 16-bit microcontroller.

Advantages/Drawbacks

- Advantages
  - Can change microcode and keep the same macro-instruction set!
  - Less prone to errors, can be updated more easily.
- Drawback
  - Cost in performance – overhead.

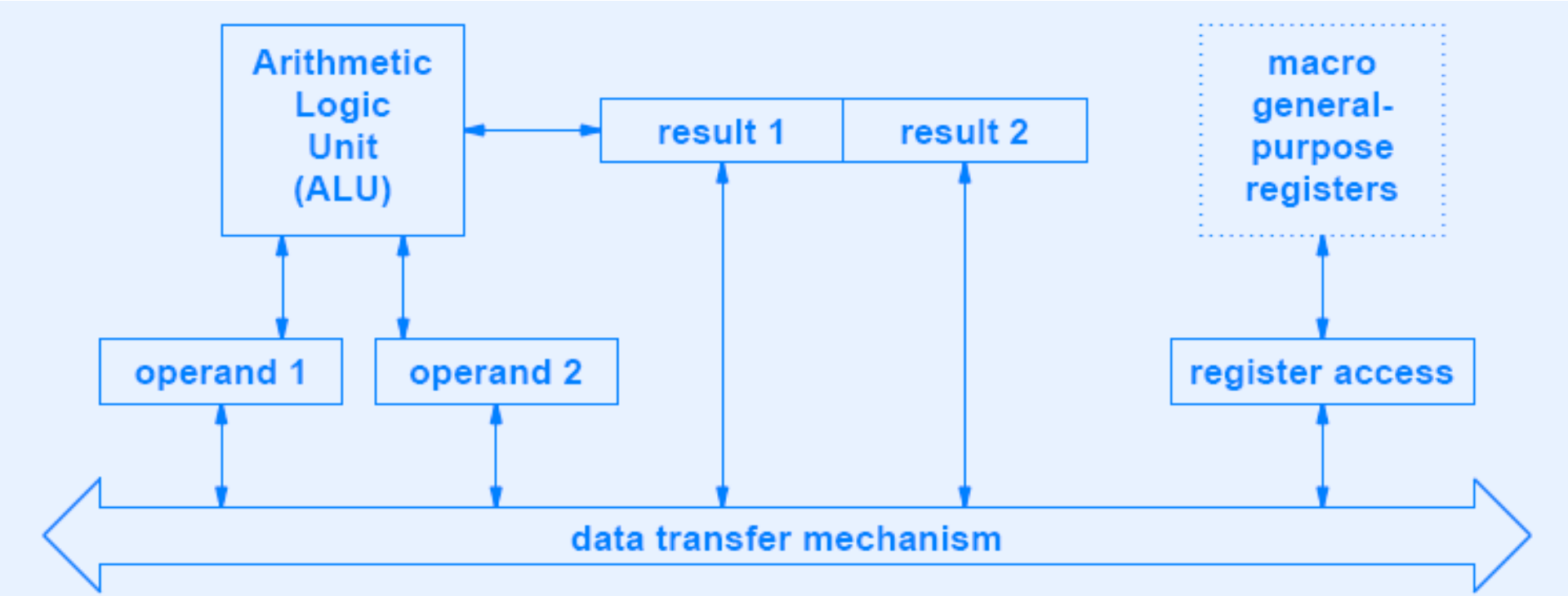
Vertical Microcode

- Simple view of microcontroller ~ standard processor.
- Execution of micro-code like assembly.
- One micro-instruction at a time.
- Access to different units.
- Decode each macro-instruction and execute micro-code.
- Easy to read/write, bad performance.
- Not the case in practice.

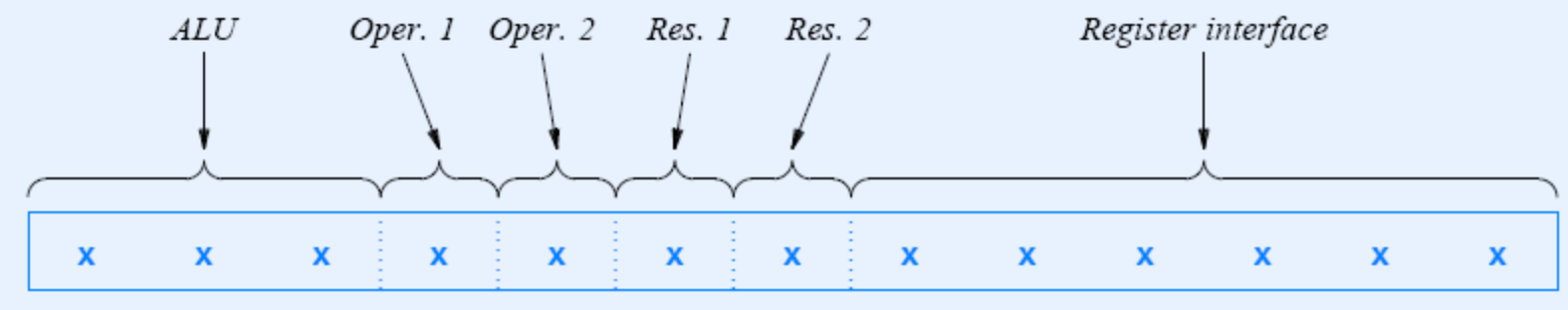
Horizontal Microcode

- Use implicit parallelism.
  - Utilize units in parallel when possible.
- Control data movements and the different hardware units *at the same time*.
- Very difficult to program.
- Long instruction:
  |exec op1 unit1|exec op2 unit2|transfer this register there|…

Example Architecture



Unit	Command	Meaning
ALU	0 0 0	No operation
	0 0 1	Add
	0 1 0	Subtract
	0 1 1	Multiply
	1 0 0	Divide
	1 0 1	Left shift
	1 1 0	Right shift
	1 1 1	Continue previous operation
operand 1 or 2	0	No operation
	1	Load value from data transfer mechanism
result 1 or 2	0	No operation
	1	Send value to data transfer mechanism
register interface	0 0 x x x x	No operation
	0 1 x x x x	Move register xxxx to data transfer mechanism
	1 0 x x x x	Move data transfer mechanism to register xxxx
	1 1 x x x x	No operation



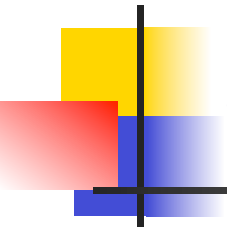
Horizontal Microcode

- Not like conventional programs.
- Each instruction takes one cycle
  - but not all operations take one cycle
  - special care for timing, wait for units that need more cycles

continue

Intelligent Microcontroller

- Schedules instructions & units.
- Handles operations in parallel.
- Performs branch prediction.
  - May try 2 paths and discard the results of the wrong one later.
  - Important: Keep the sequential semantics.
- Out-of-order execution
  - use scoreboard to keep track of results and dependencies



Conclusion

- Does it matter?
  - Yes! Understand your hardware and its technology.
  - Use it in a better way. Reduce branches, or make them easy to guess.
    Ex:
    for(i =0, j = n-1; i < j; ++i,--j) swap(&a[i],&a[j])
    harder than
    for(i = 0; i < n/2; ++i) swap(&a[i],&a[n-1-i])