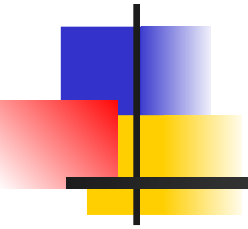
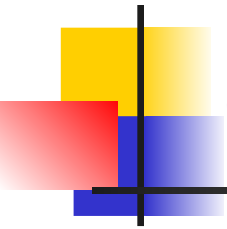


Processor Architecture I

Mads Chr. Olesen

Credit to Alexandre David

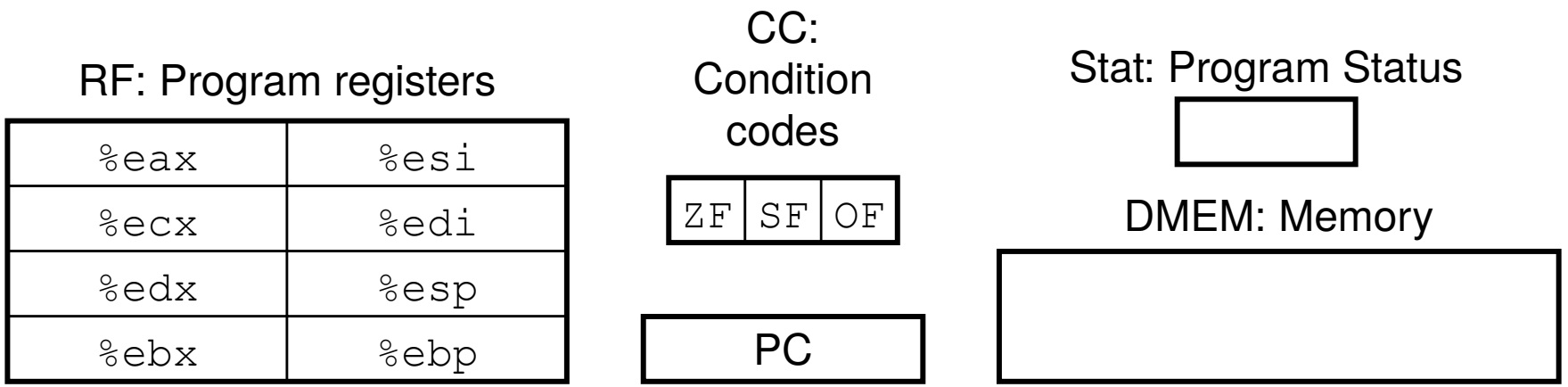


General Purpose CPU

- 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

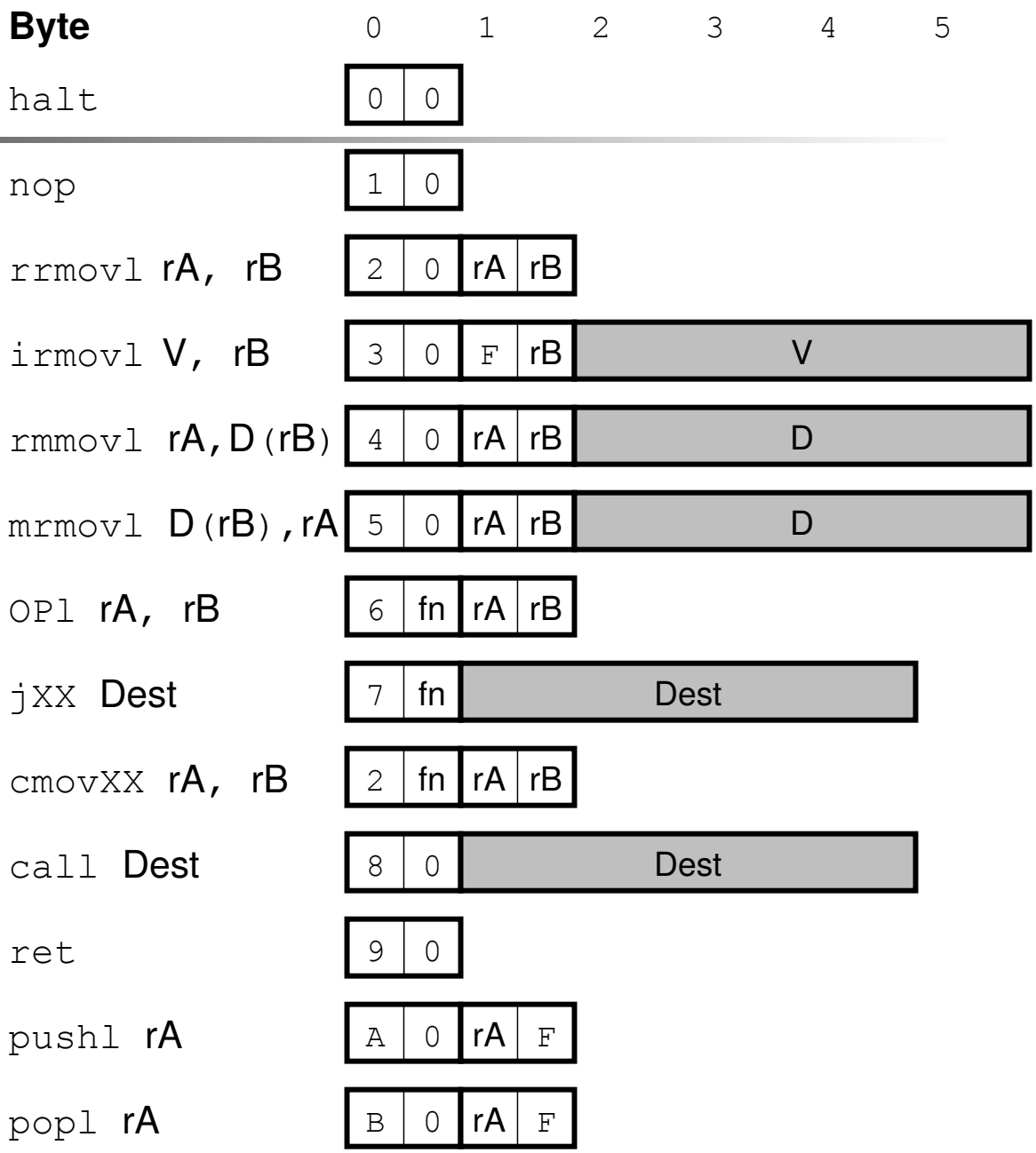
CPU Visible State

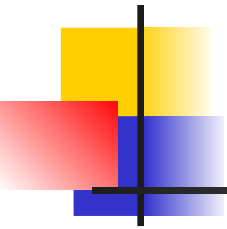
- Visible for ISA, used by compiler (& assembler) – there may be other registers etc… that depend on the CPU generation.
- Registers, condition codes, status & memory.
- Memory=array of bytes (abstraction).



Y86 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.
- 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

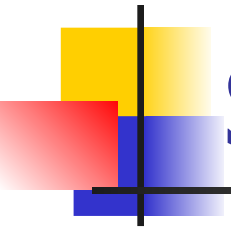
addl	6	0
subl	6	1
andl	6	2
xorl	6	3

Branches

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

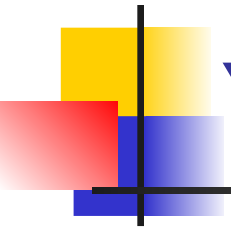
Moves

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
- $\rightarrow$ may need more instructions
  - not important, we abstract from that.
  - Reason on Y86 level, exercise with both (simulator Y86, gcc for X86).

Y86 Assembly

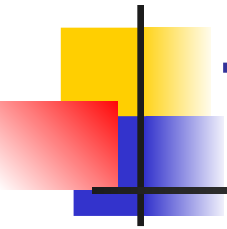
- Instructions as described with registers.
- Assembly directives.
  - Where to put the code (.pos).
  - Align the code (.align).
  - Declare data (.long).
  - X86 has more.
- Label declarations (used for jump offsets).
- Assembled into bytes.
- Y86 interpret the bytes.

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 = R*I
- Dissipated power: P = U*I = U*U/R

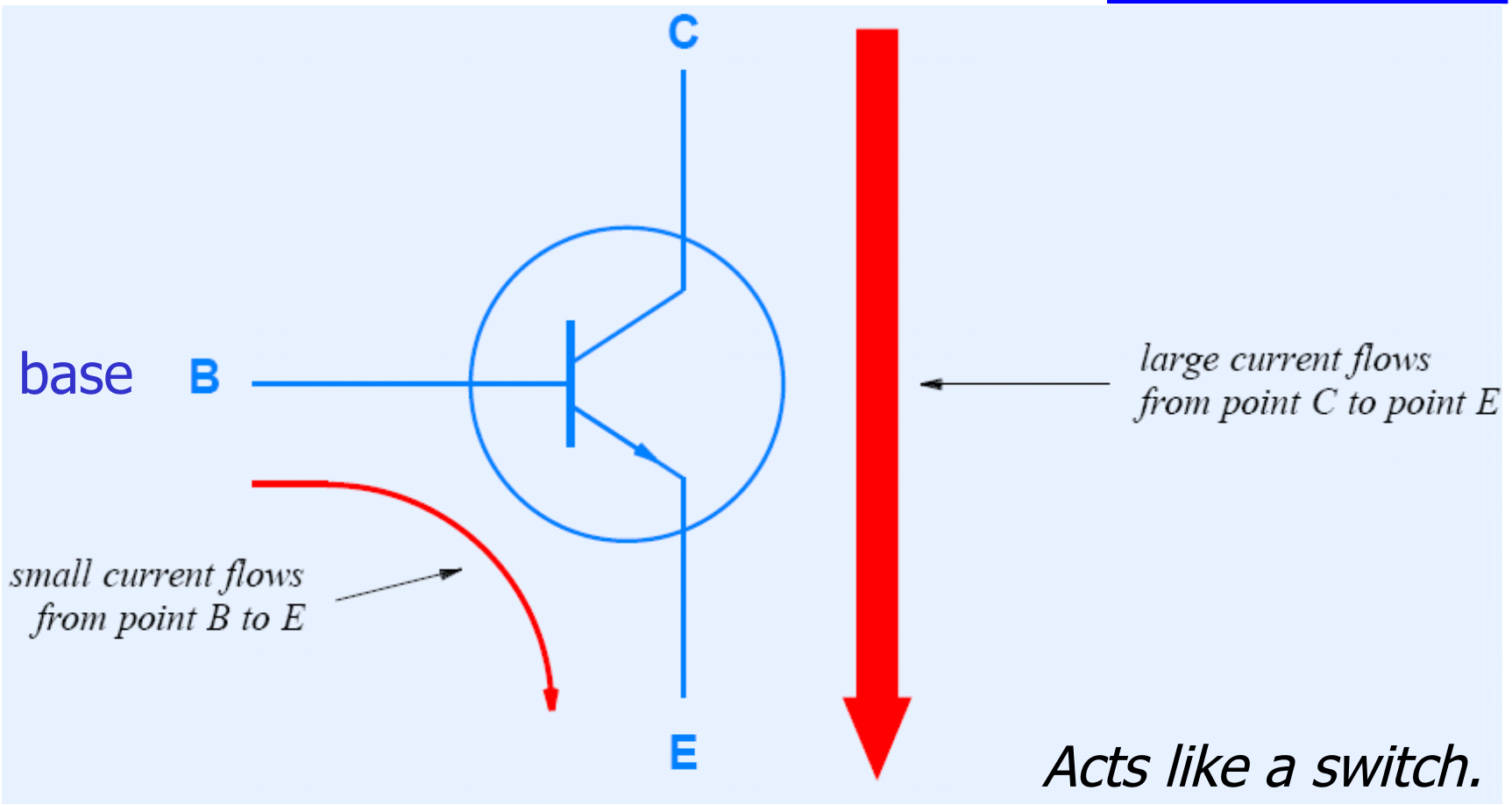


Typical Chips

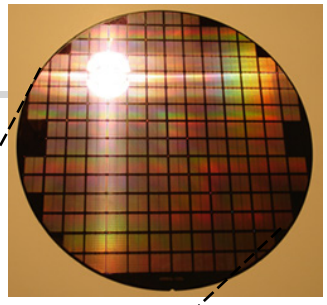
- Operate on low voltage (5V, less for processors) – see power dissipation.
- Always 2 lines
  - ground (0V)
  - power (5V)
- Diagrams usually omit ground and power.

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

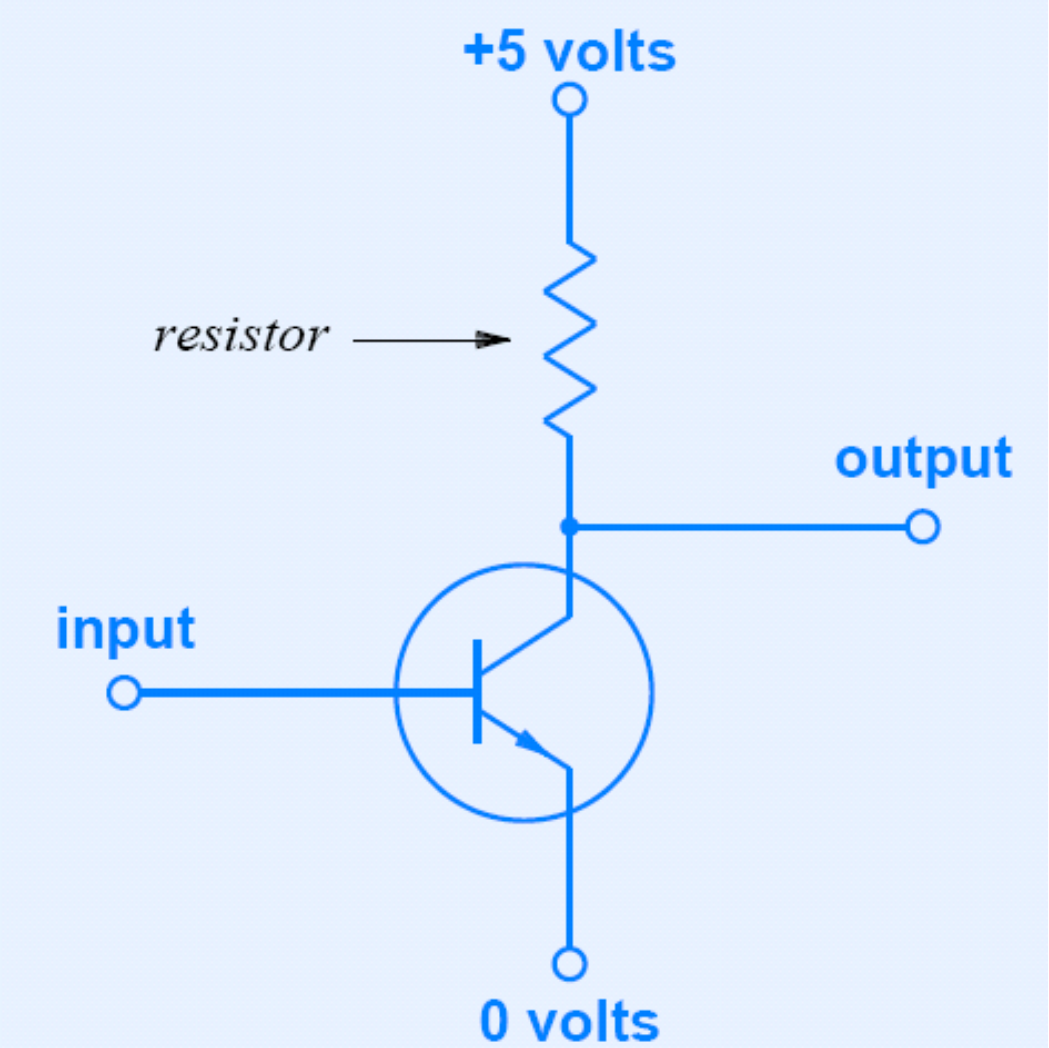
- Mathematical basis for digital circuits.
- From boolean functions to gates.
- Basic functions: and, or, not.
- In practice, cheaper to have nand & nor.

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

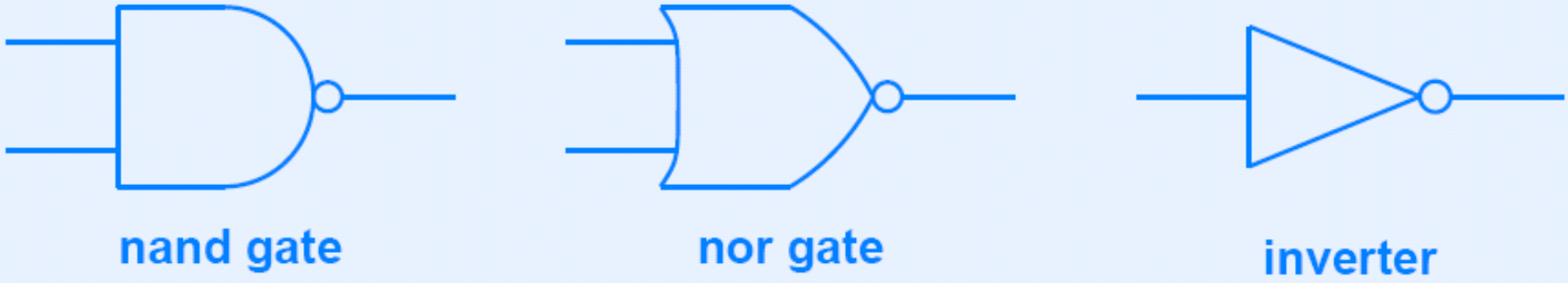
A	not A
0	1
1	0

Example: Not



Gates

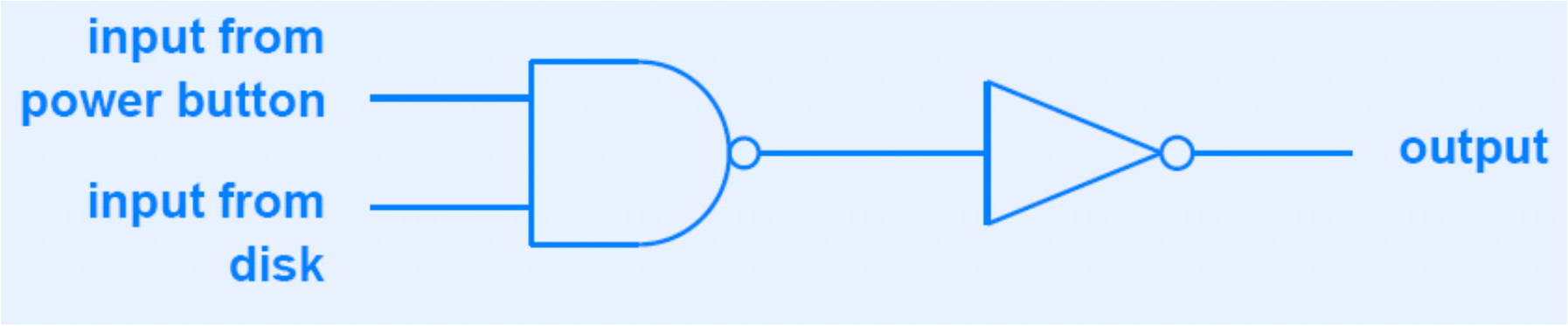
- Primitive boolean functions.
- Level of abstraction on integrated circuits.



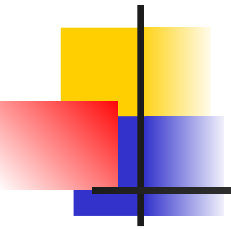
Symbols used in circuits.

Logic Gate Technology

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



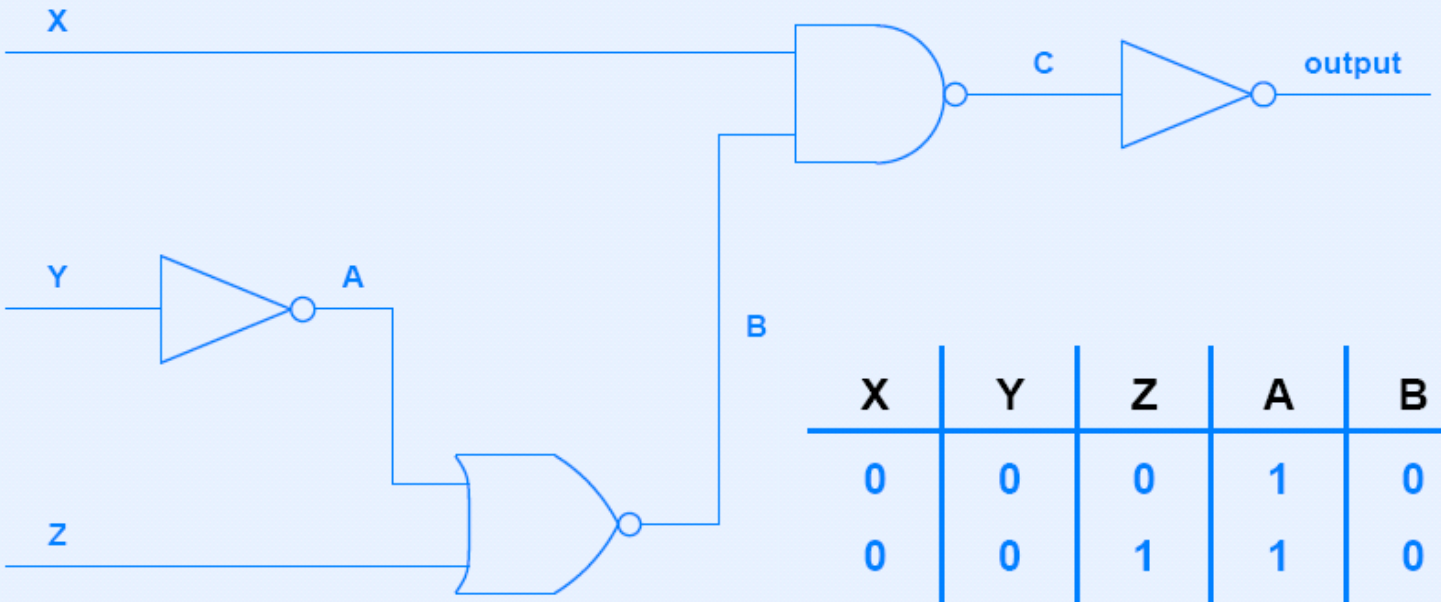
and function



Design of Functions

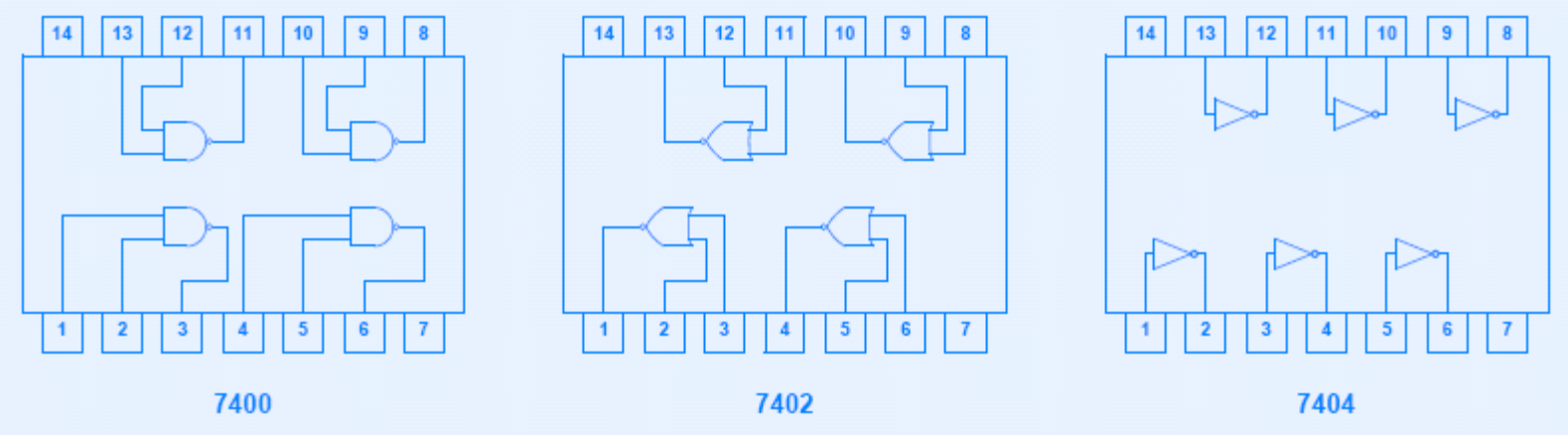
- Find a boolean expression that does what you need
  - and feed it to a tool that optimizes it to minimize the number of gates.
- Come up with the truth table of your function
  - which is converted to a boolean function.

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

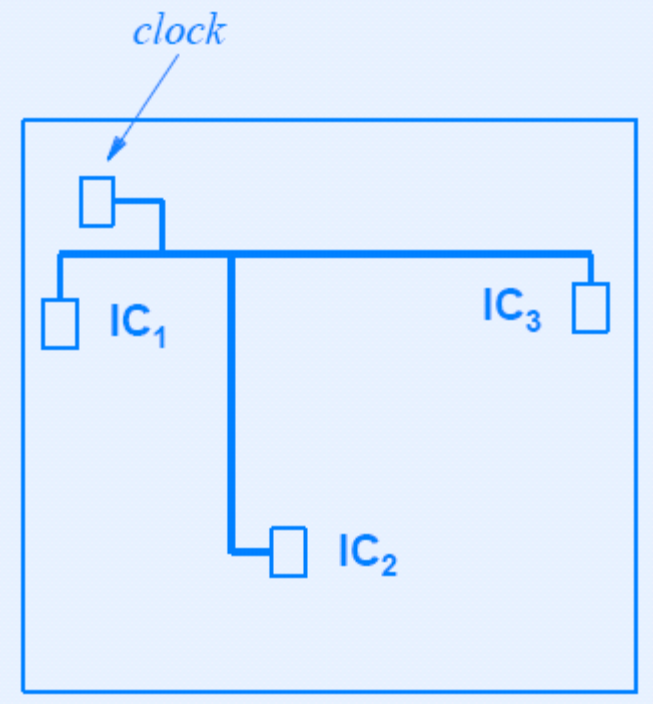


- Outputs = function(inputs)
  - change outputs only when inputs changes
  - need states to perform sequences of operations without sustained inputs
    - maintain states
    - use a clock

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



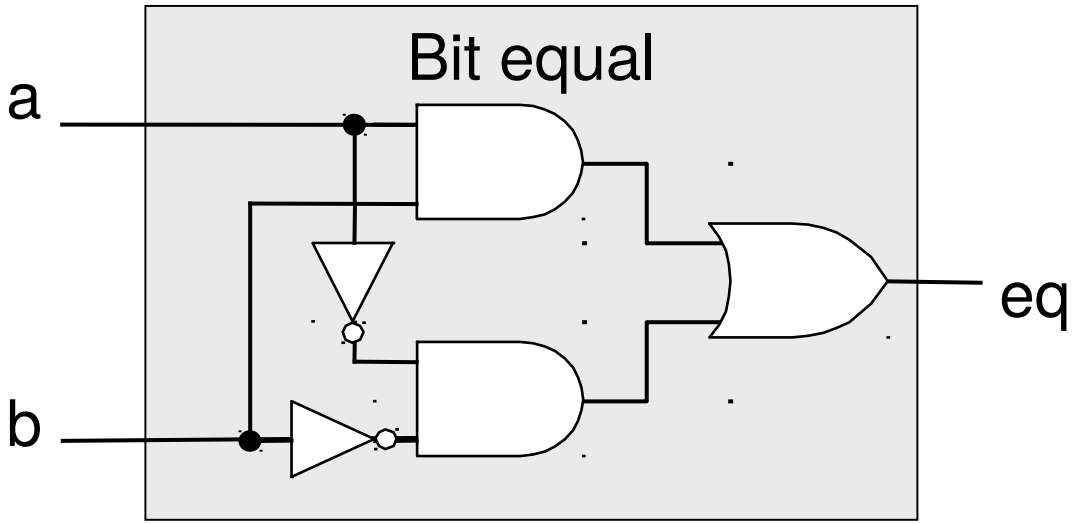
Signals need time to propagate.
Local clocks are used on larger systems → need to synchronize them.
The speed of light is too slow.

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 $\rightarrow$ boolean functions.

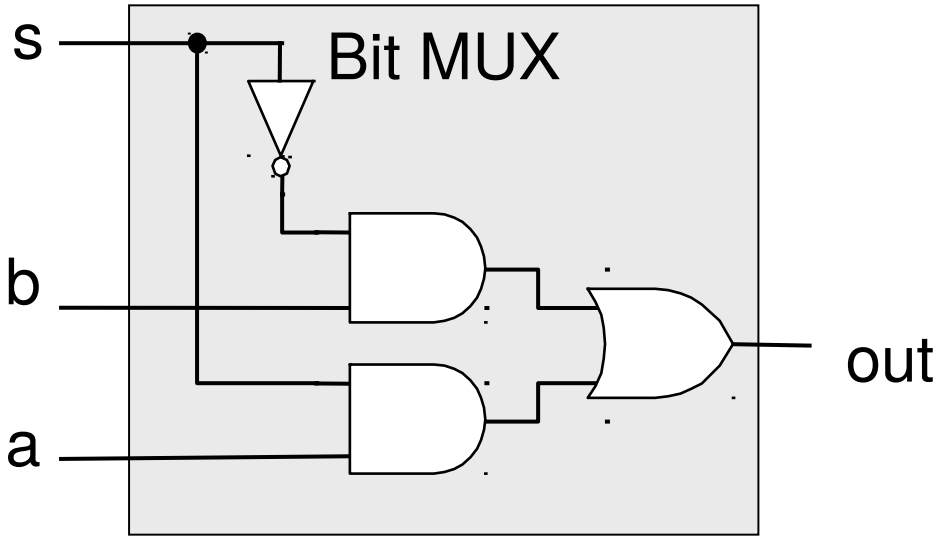
Example

- 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…



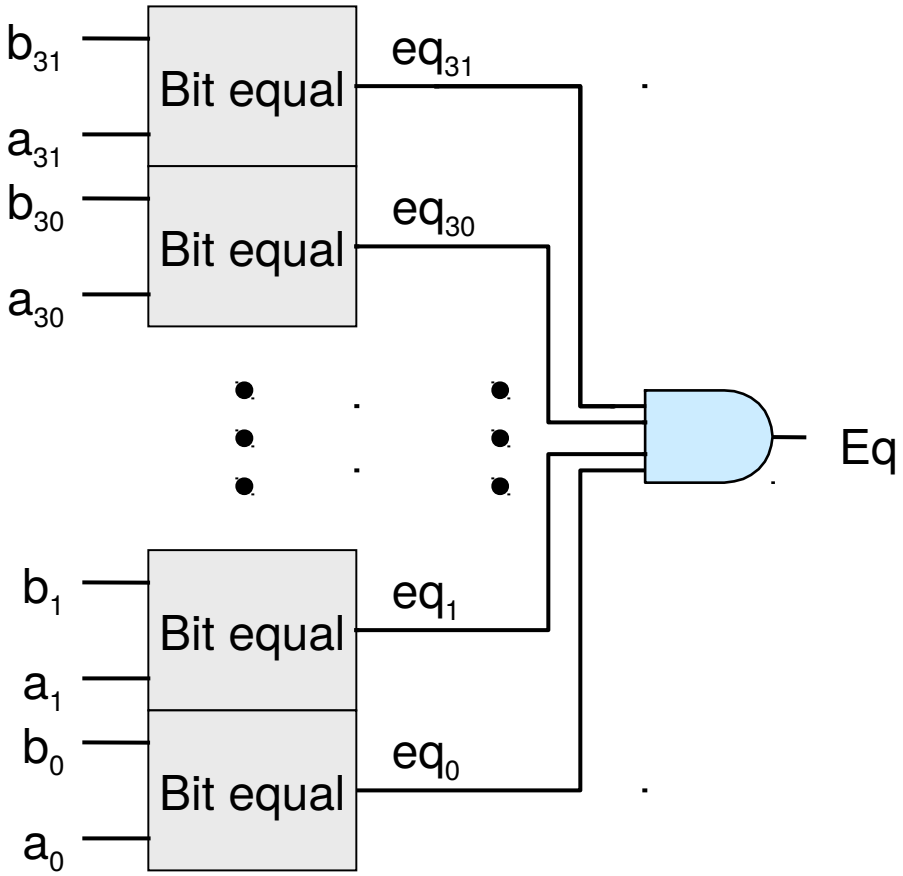
Word-Level Combinatorial Circuits

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

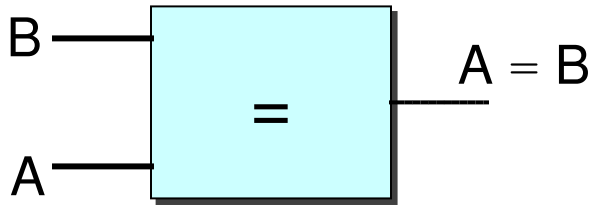
Example: Equality Test

- bool Eq = (A == B);

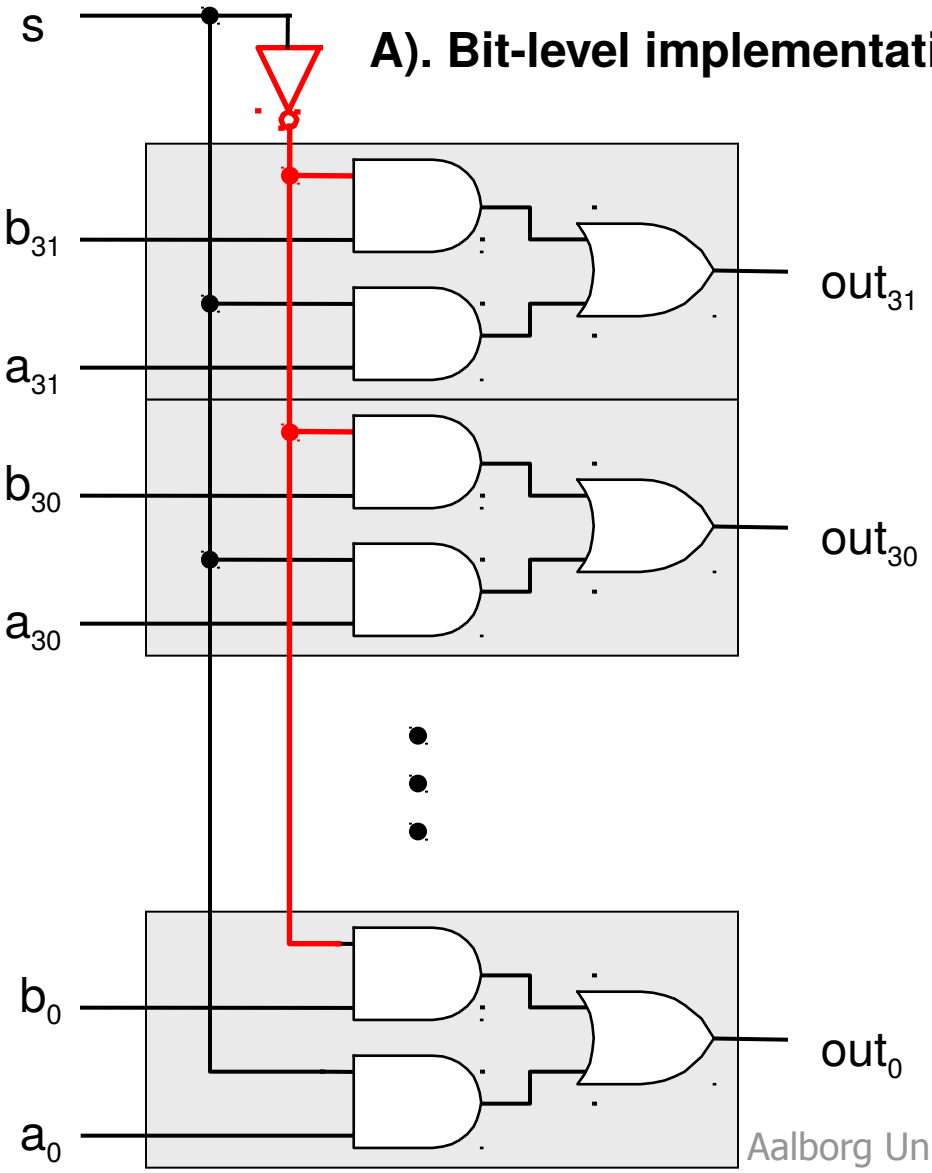
A). Bit-level implementation



B). Word-level abstraction

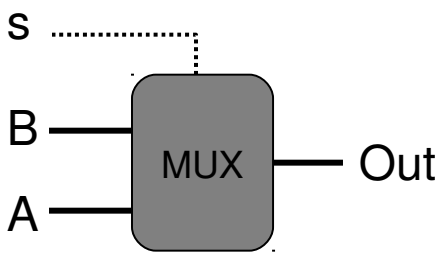


Multiplexor At Word-Level



A). Bit-level implementation

B). Word-level abstraction

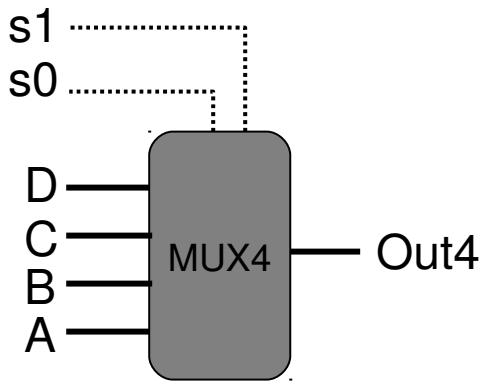



```
int Out = [
   s : A;
   1 : B;
];
```

Use: Select

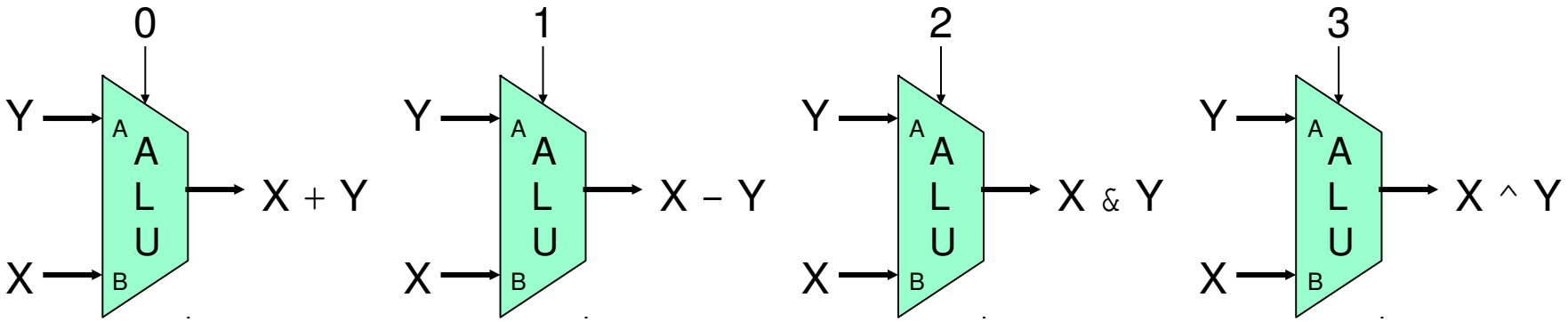
```
int Out4 = [
    !s1 && !s0 : A; # 00
    !s1        : B; # 01
    !s0        : C; # 10
    1          : D; # 11
]
```

- 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.

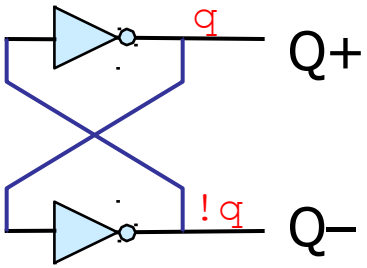


Memory & Clocking

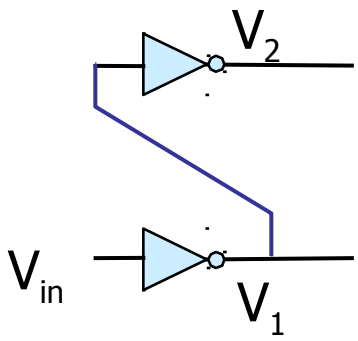
- Memory stores states.
  - Functions only propagates signals.
  - Memory implemented as flip-flop-like circuits. Have feedback loops to “keep” bits.
  - Registers (hardware or program).
- Clocks synchronize when to update.
  - Between updates, signals propagate.
  - Clock signal rises $\rightarrow$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

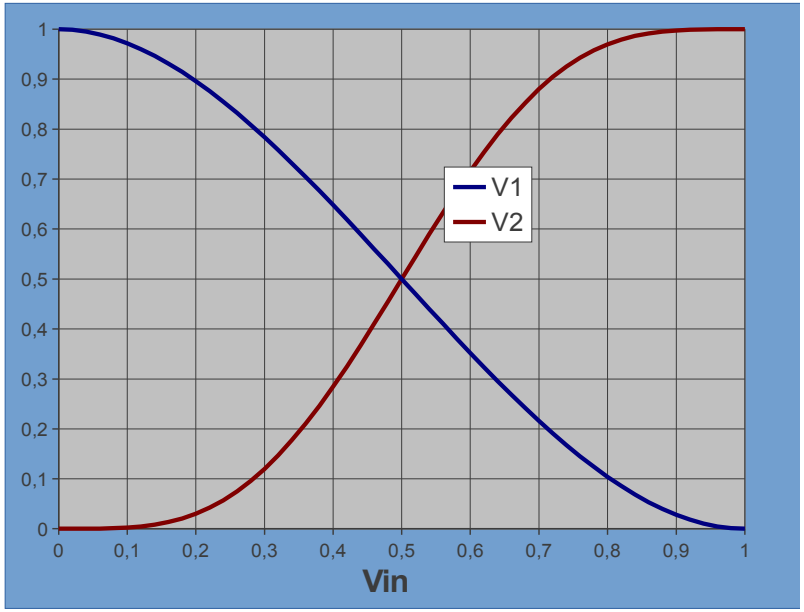
Storing 1 Bit

Bistable Element



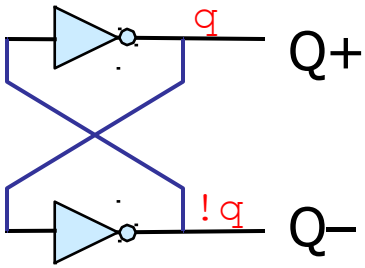
q = 0 or 1



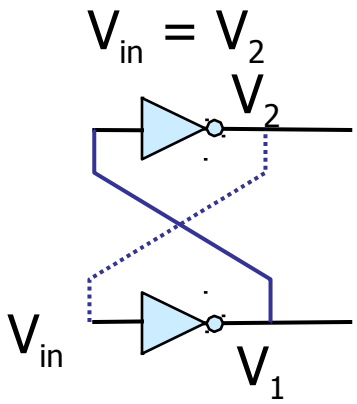


Storing 1 Bit (cont.)

Bistable Element

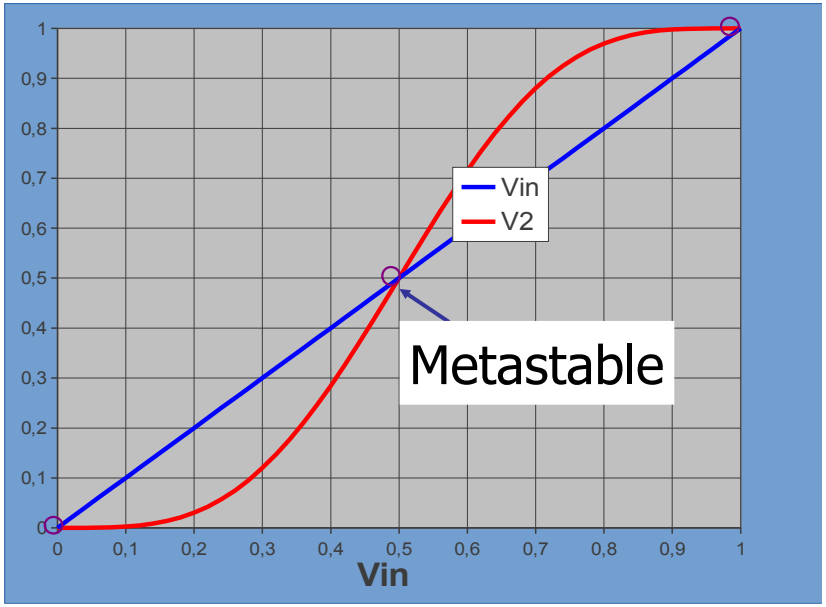


q = 0 or 1

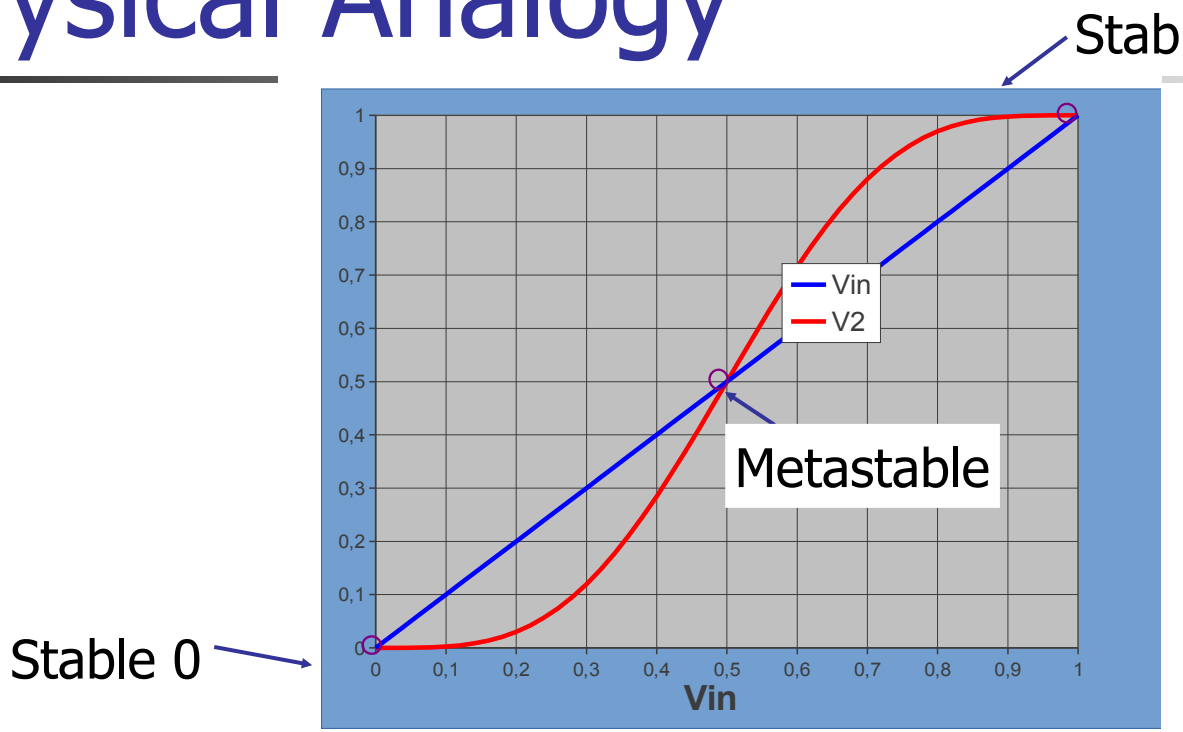


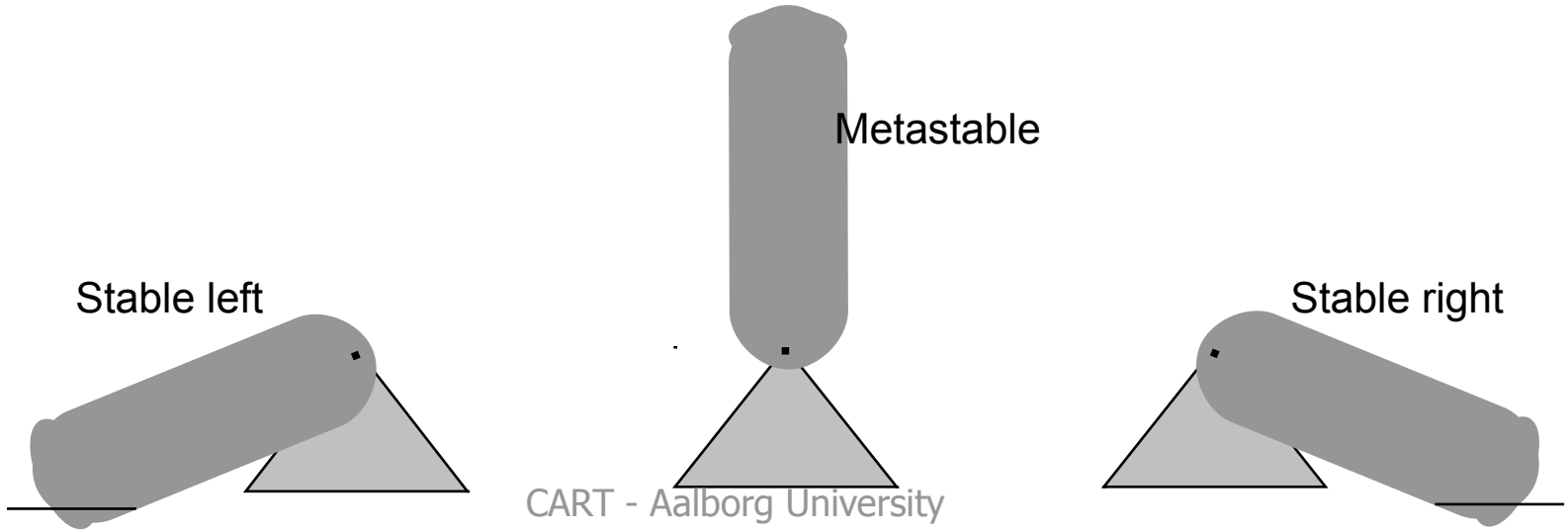
Stable 1

Stable 0



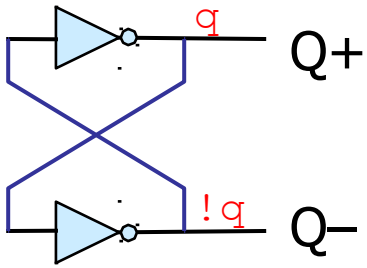
Physical Analogy





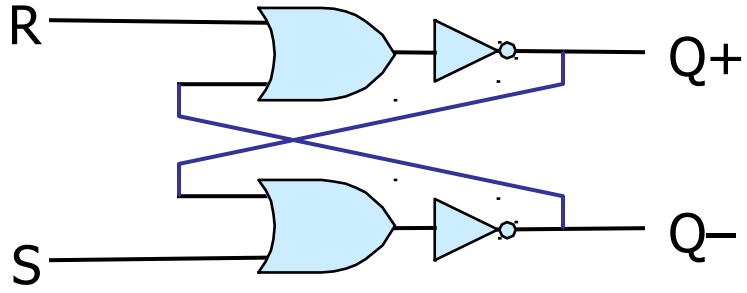
Storing and Accessing 1 Bit

Bistable Element

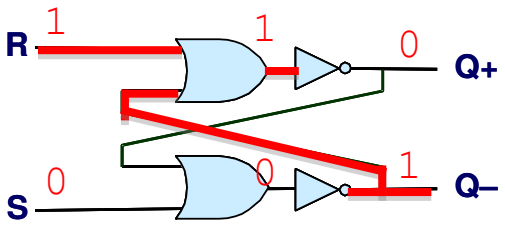


q = 0 or 1

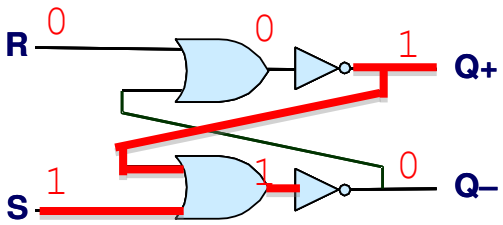
R-S Latch



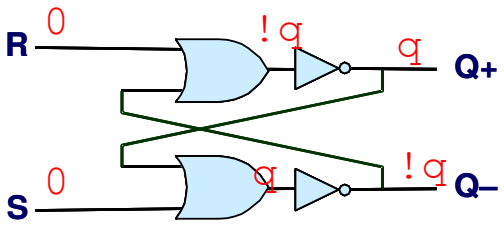
Resetting



Setting

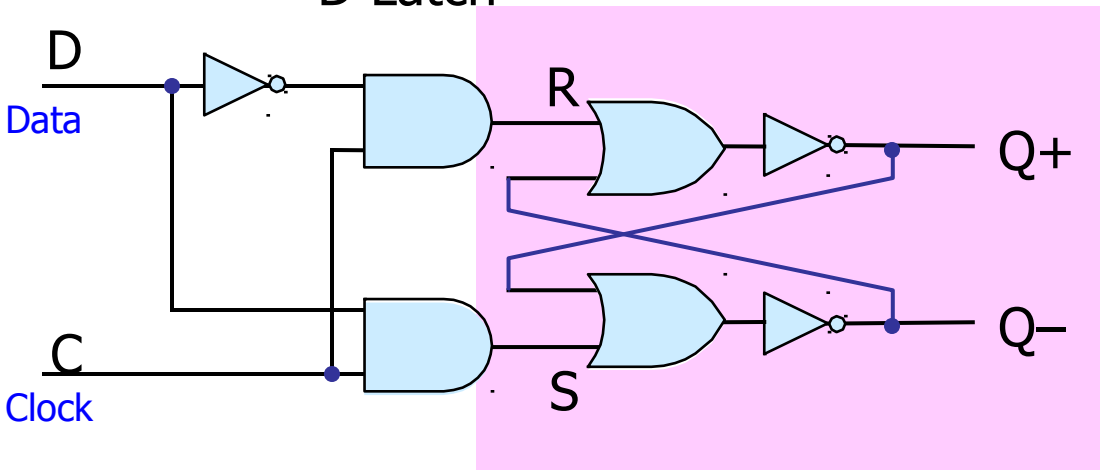


Storing

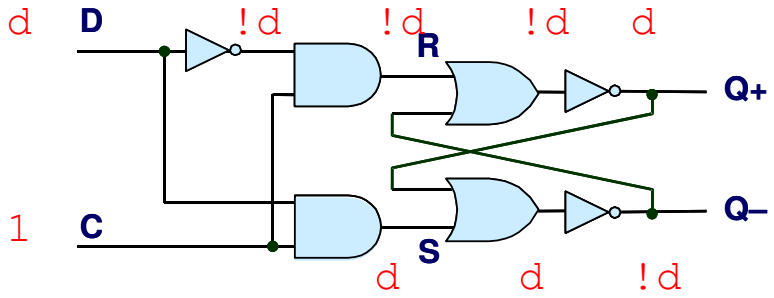


1-Bit Latch

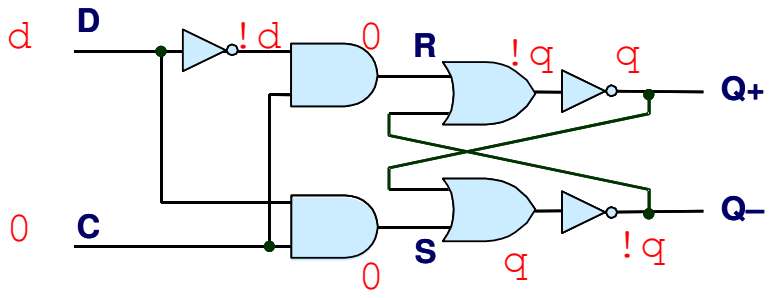
D Latch



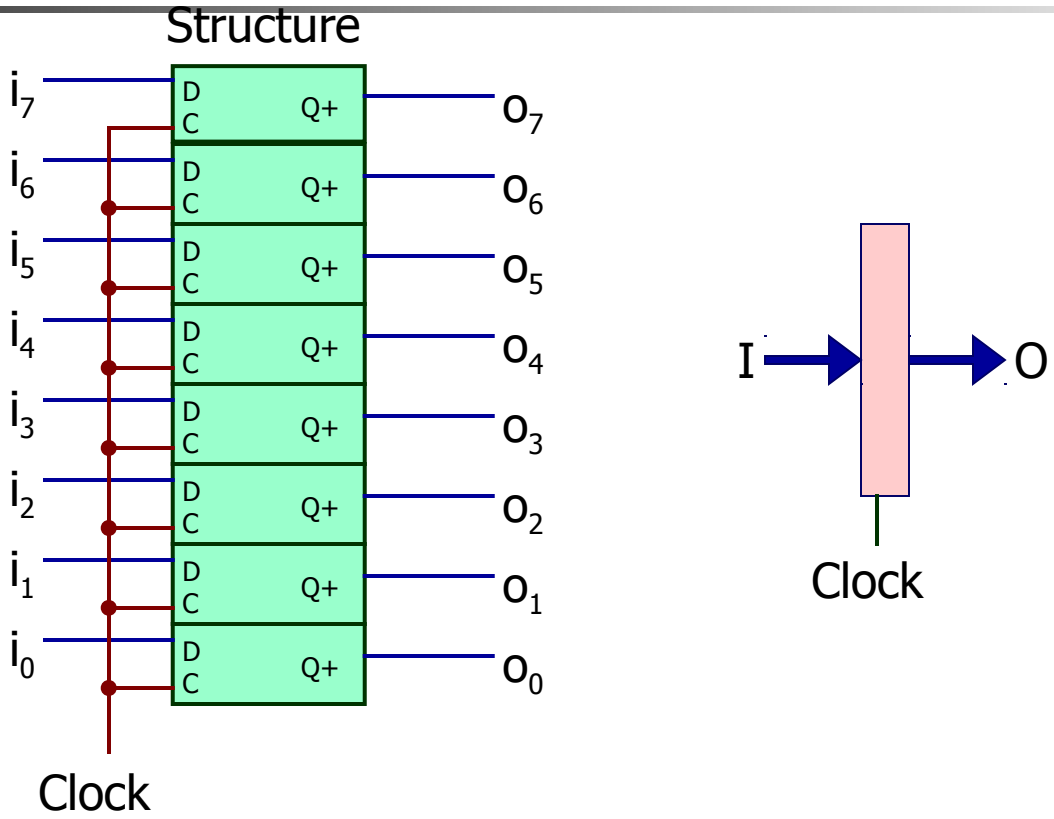
Latching



Storing

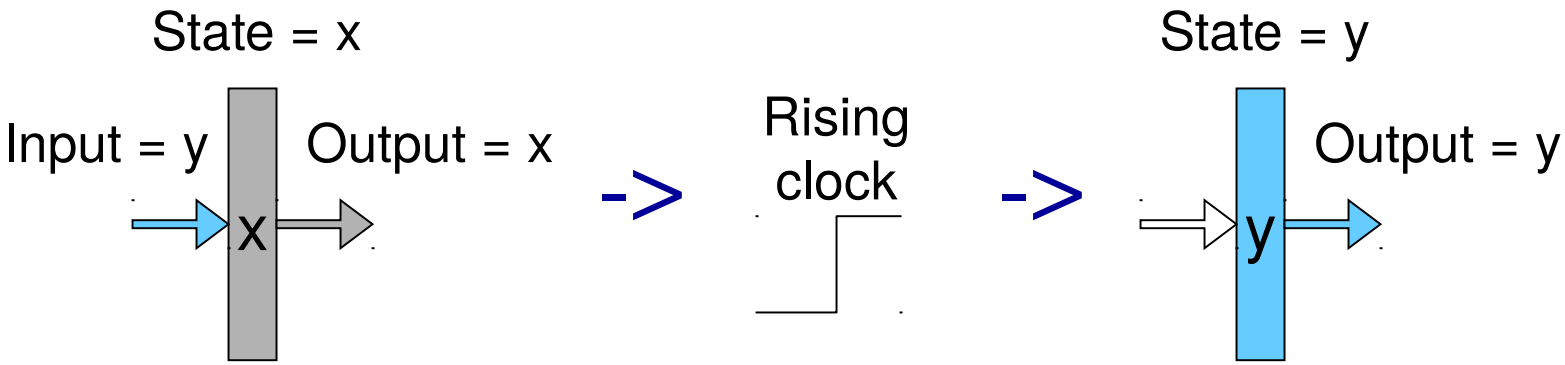


Hardware Registers

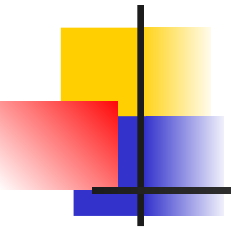


- Stores word of data
  - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

Clock Synchronization

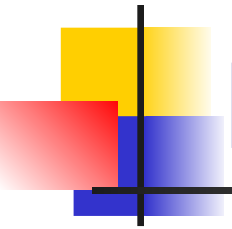


- Register operations are synchronized.
  - Stores data bits.
  - For most of time acts as barrier between input and output.
  - As clock rises, loads input.



Register File

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

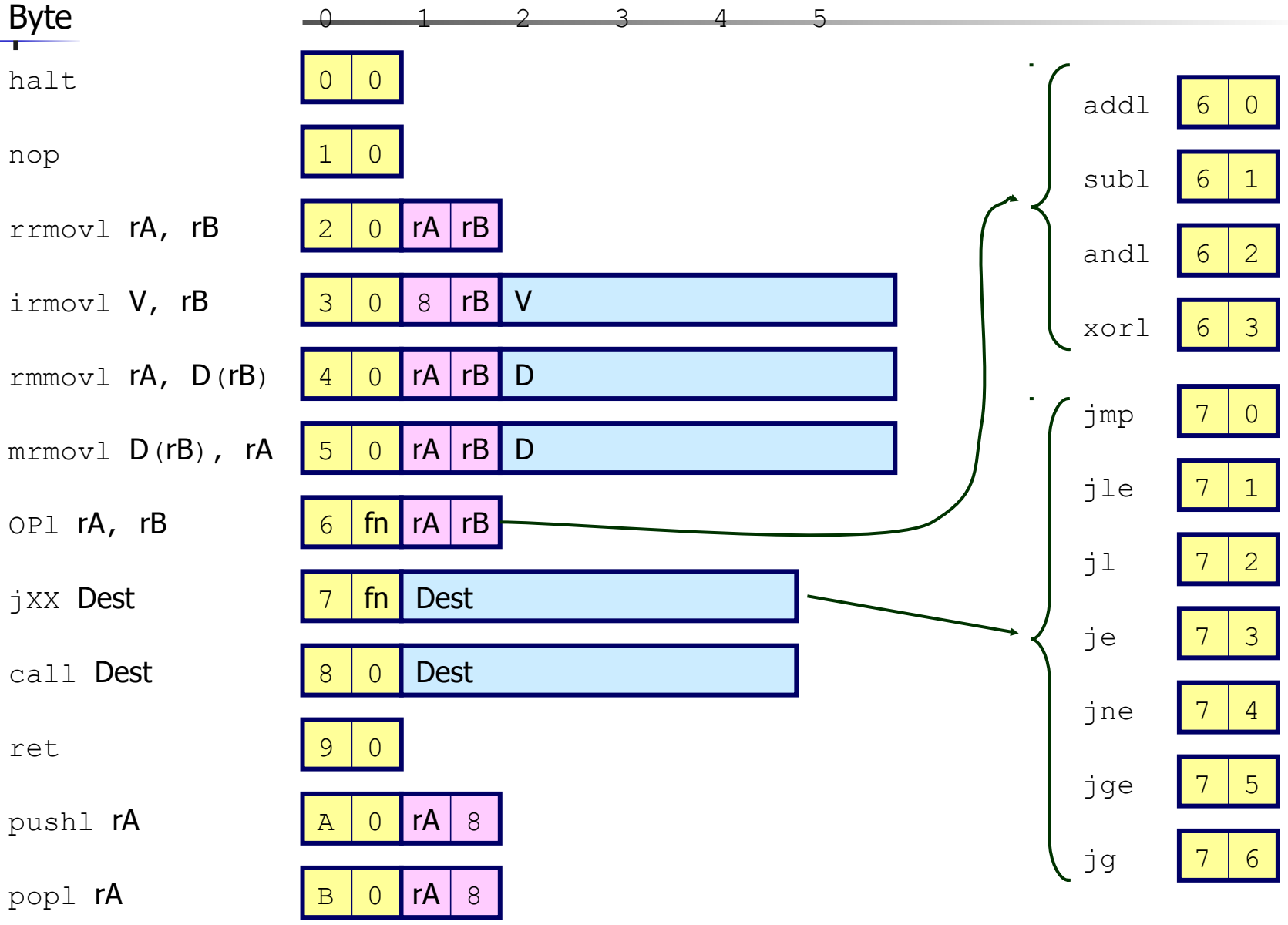


Memory

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

SEQ: Sequential Processor

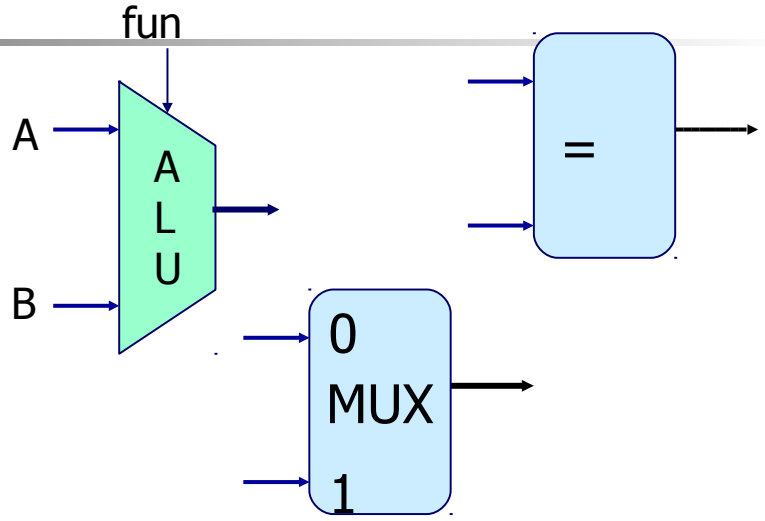
Y86 Instruction Set



Building Blocks

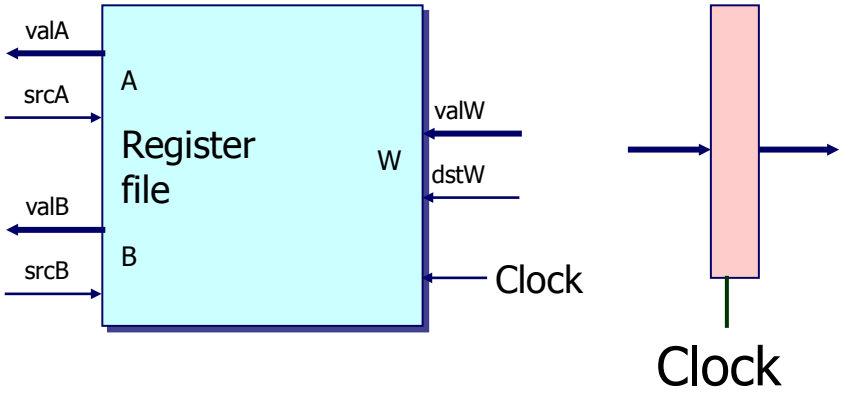
Combinational Logic

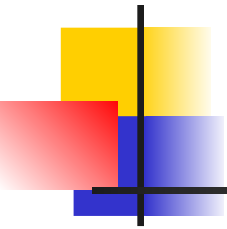
- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control



Storage Elements

- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises





Hardware Control Language

- 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, 32-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

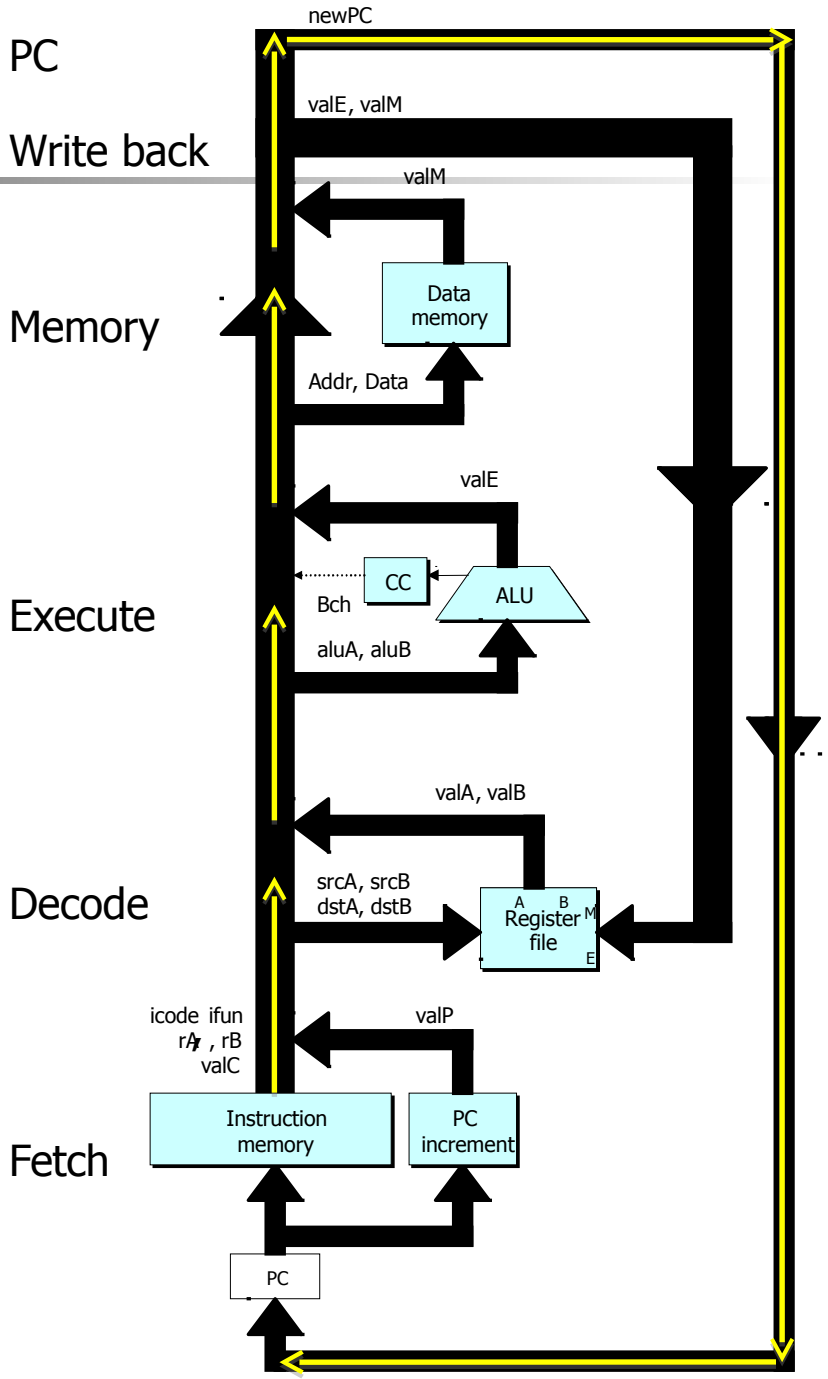
SEQ Hardware Structure

State

- 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

- Read instruction at address specified by PC
- Process through stages
- Update program counter



SEQ Stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

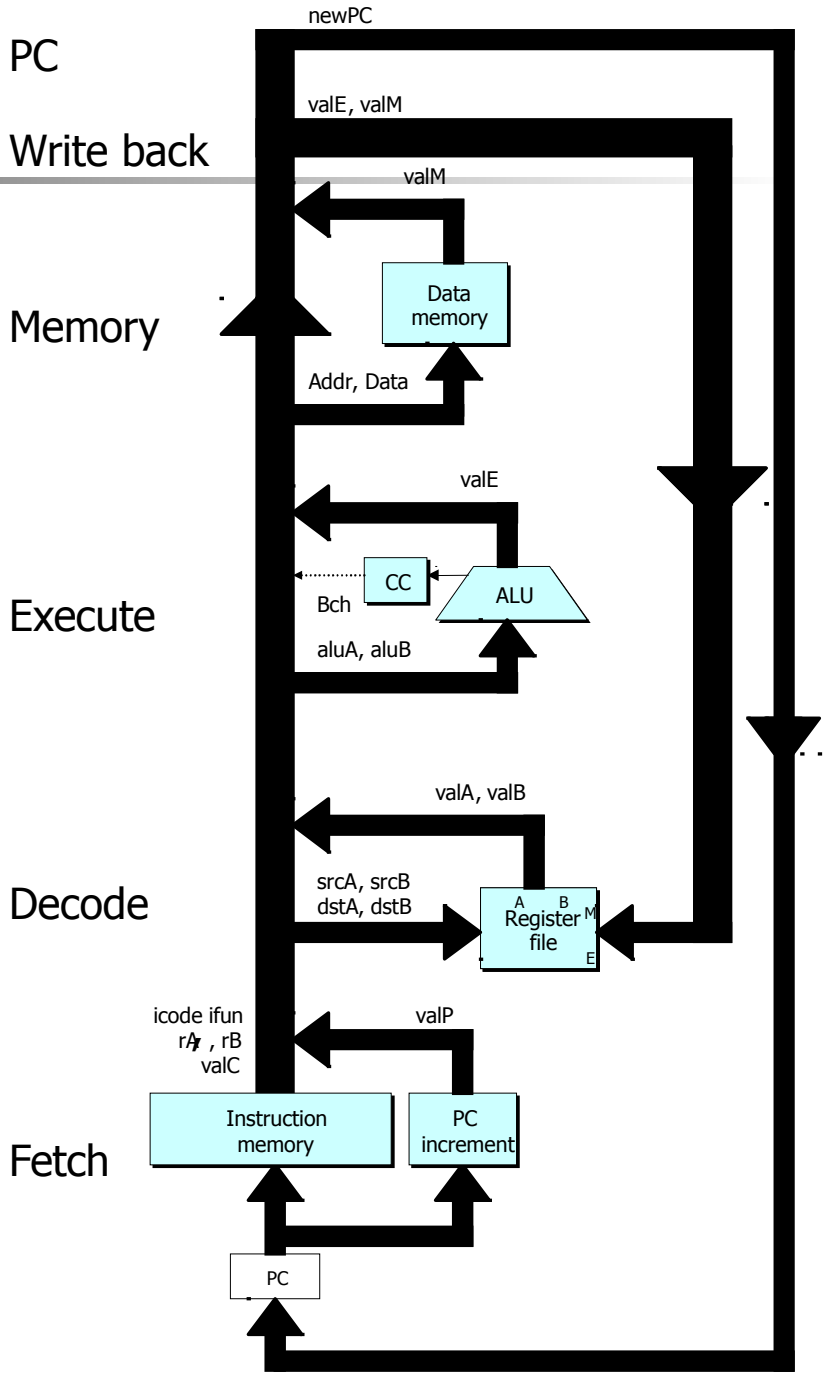
- Read or write data

Write Back

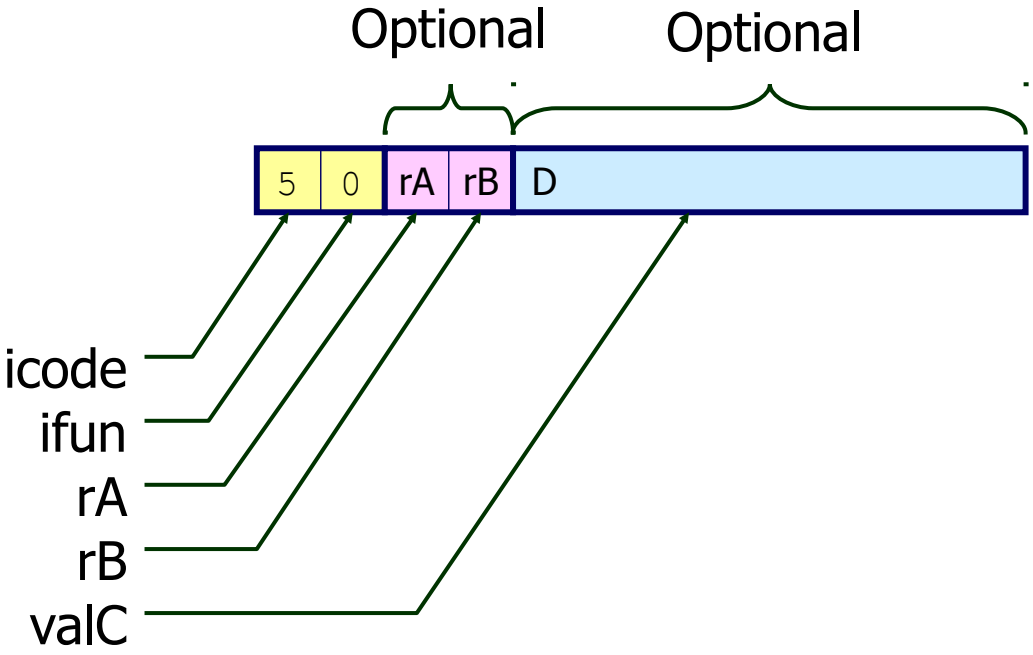
- Write program registers

PC

- Update program counter



Instruction Decoding



Instruction Format

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

Executing Arith./Logical ops.

`OPl` rA, rB

Fetch

- Read 2 bytes

Decode

- Read operand registers

Execute

- Perform operation
- Set condition codes

Memory

- Do nothing

Write back

- Update register

PC Update

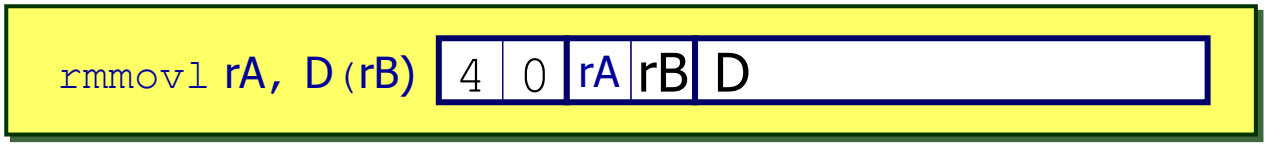
- Increment PC by 2

Stage Computation: Arith/Log. Ops

	OPl rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Writeback	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

Executing `rmmovl`



Fetch

- Read 6 bytes

Decode

- Read operand registers

Execute

- Compute effective address

Memory

- Write to memory

Write back

- Do nothing

PC Update

- Increment PC by 6

Stage Computation: `rmmovl`

	`rmmovl` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[PC+2]	Read displacement D
	valP ← PC+6	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_4[valE] ← valA	Write value to memory
Writeback		
PC update	PC ← valP	Update PC

- Use ALU for address computation

Executing `popl`

`popl` rA

Fetch

- Read 2 bytes

Decode

- Read stack pointer

Execute

- Increment stack pointer by 4

Memory

- Read from old stack pointer

Write back

- Update stack pointer
- Write result to register

PC Update

- Increment PC by 2

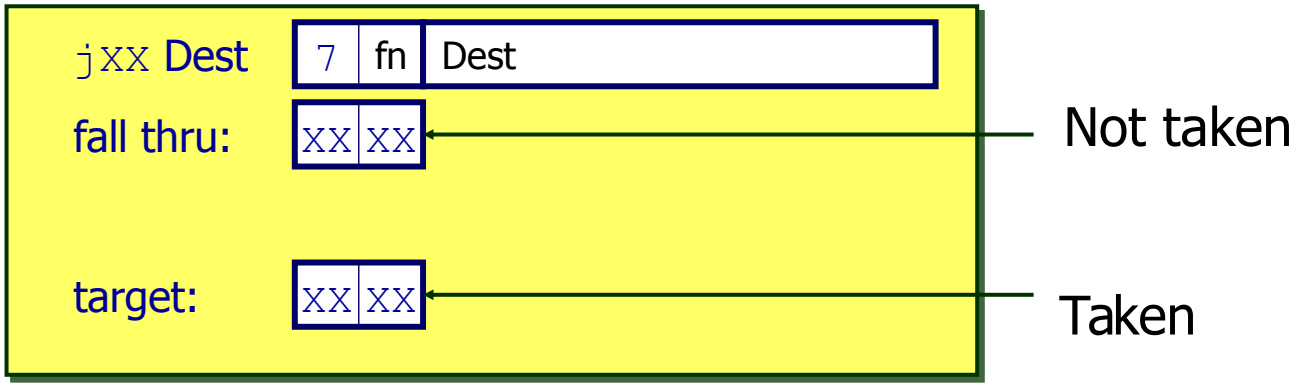
PP 4.12
PP 4.13

Stage Computation: `popl`

	`popl` 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[`%esp`]	Read stack pointer
	valB ← R [`%esp`]	Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Writeback	R[`%esp`] ← 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

Executing Jumps



Fetch

- Read 5 bytes
- Increment PC by 5

Decode

- Do nothing

Execute

- Determine whether to take branch based on jump condition and condition codes

Memory

- Do nothing

Write back

- Do nothing

PC Update

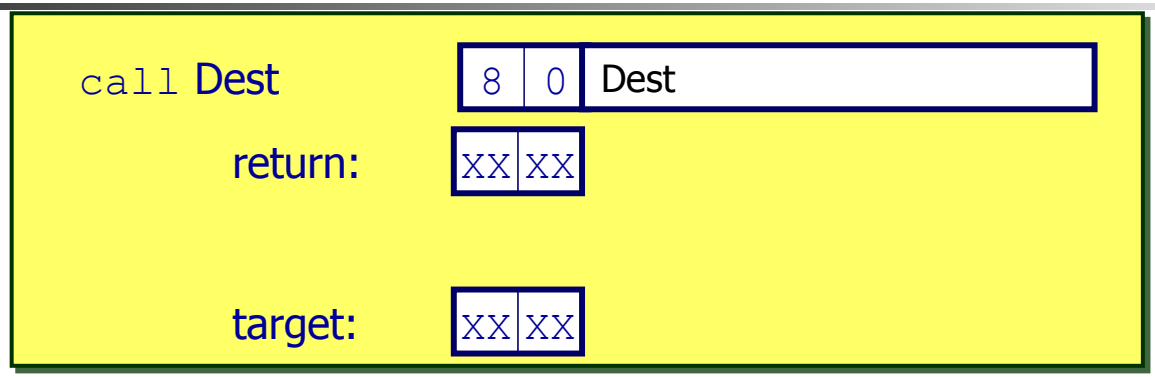
- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Fall through address
Decode		
Execute	Bch ← Cond(CC,ifun)	Take branch?
Memory		
Writeback		
PC update	PC ← Bch ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Executing `call`



Fetch

- Read 5 bytes
- Increment PC by 5

Decode

- Read stack pointer

Execute

- Decrement stack pointer by 4

Memory

- Write incremented PC to new value of stack pointer

Write back

- Update stack pointer

PC Update

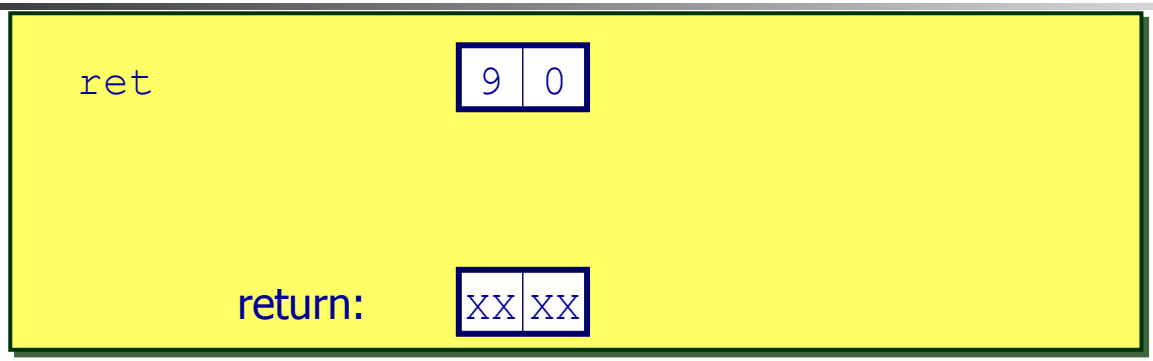
- Set PC to Dest

Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Compute return point
Decode	valB ← R[`%esp`]	Read stack pointer
Execute	valE ← valB + –4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Writeback	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing `ret`



Fetch

- Read 1 byte

Decode

- Read stack pointer

Execute

- Increment stack pointer by 4

Memory

- Read return address from old stack pointer

Write back

- Update stack pointer

PC Update

- Set PC to return address

Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%esp`] valB ← R[`%esp`]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Writeback	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

Computation Steps

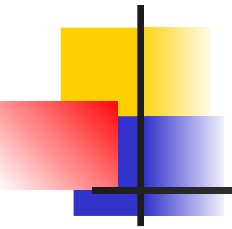
		OPl rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set condition code register
Memory	valM		[Memory read/write]
Writeback	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

		`call` Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_4[PC+1]	Read constant word
	valP	valP ← PC+5	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[`%esp`]	Read operand B
Execute	valE	valE ← valB + –4	Perform ALU operation
	Cond code		[Set condition code reg.]
Memory	valM	M_4[valE] ← valP	[Memory read/write]
Writeback	dstE	R[`%esp`] ← valE	[Write back ALU result]
	dstM		Write back memory result
PC update	PC	PC ← valC	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step



Computed Values

Fetch

icode	Instruction code
ifun	Instruction function
rA	Instr. Register A
rB	Instr. Register B
valC	Instruction constant
valP	Incremented PC

Decode

srcA	Register ID A
srcB	Register ID B
dstE	Destination Register E
dstM	Destination Register M
valA	Register value A
valB	Register value B

Execute

- valE ALU result
- Bch Branch flag

Memory

- valM Value from memory

SEQ Stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

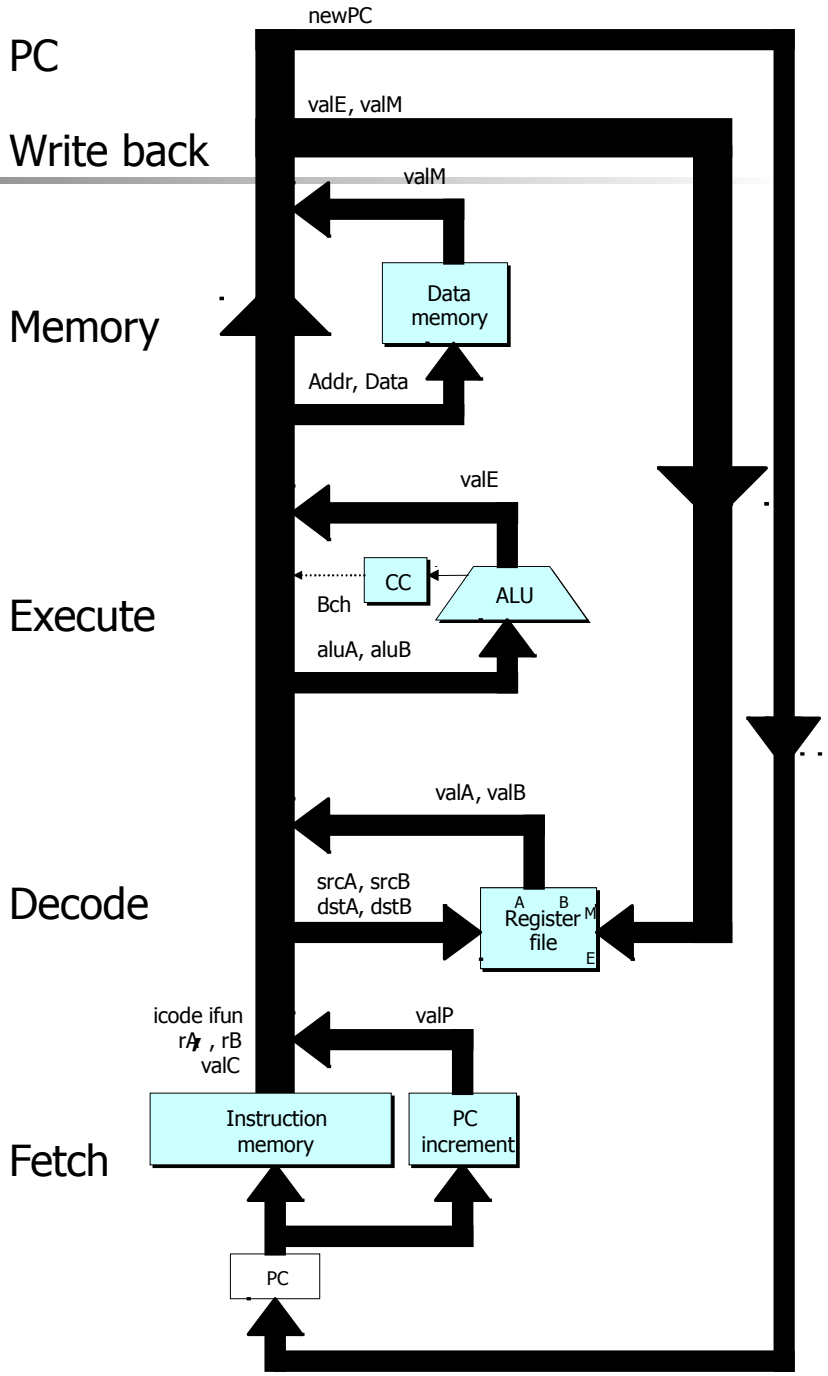
- Read or write data

Write Back

- Write program registers

PC

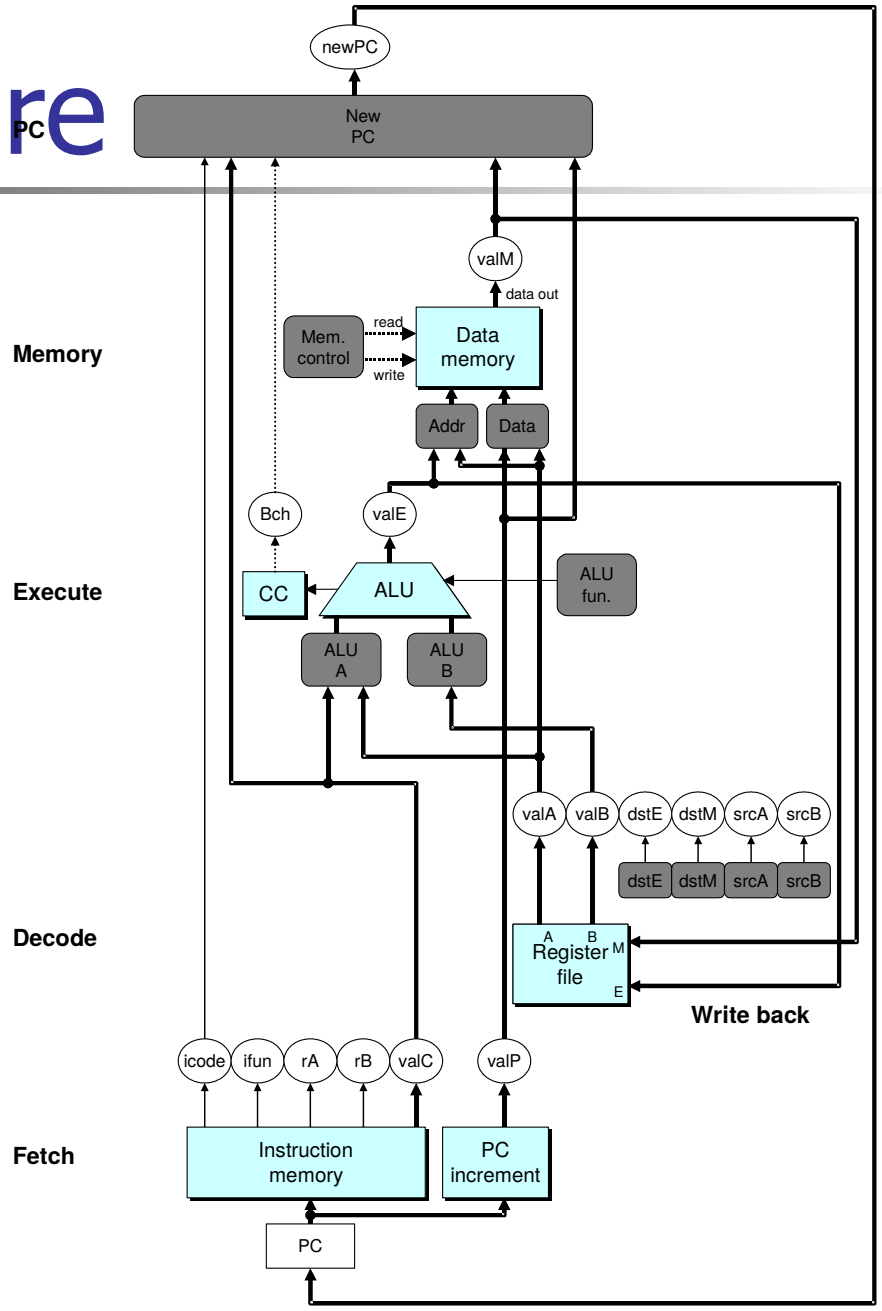
- Update program counter

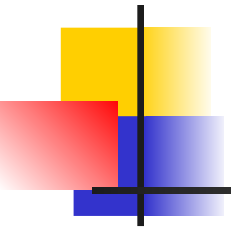


SEQ Hardware

Key

- Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 32-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values





På mandag: Workshop

- Assembly programmering
- Disassembling
- Buffer overflows