IN3170/4170, Spring 2026

Philipp Häfliger
hafliger@ifi.uio.no

Digital Circuits Basics

Content

Boolean Logic

Combinational Logic Circuits

Sequential Logic Circuits

Von Neumann Machine

Content

Boolean Logic

Combinational Logic Circuits

Sequential Logic Circuits

Von Neumann Machine

Basic Boolean Operands Symbols

	unambiguous	ASCII compatible
AND	$a \wedge b$	$a * b$
OR	$a \vee b$	$a + b$
NOT	$\neg a$ or $\overline{a}$	$\sim a$

Basic Boolean Operands Truth Tables

a	$\bar{a}$
0	1
1	0

a	b	a∧b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a∨b
0	0	0
0	1	1
1	0	1
1	1	1

Basic Boolean Expression Rules

$a \wedge b \vee c = (a \wedge b) \vee c$	$a \vee b \wedge c = a \vee (b \wedge c)$	(priority)
$a \wedge b = b \wedge a$	$a \vee b = b \vee a$	(commutativity)
$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$	(associativity)
$\bar{\bar{a}} = a$		(involution)
$a \wedge \bar{a} = 0$	$a \vee \bar{a} = 1$	(completness)
$a \wedge a = a$	$a \vee a = a$	(idempotency)
$a \wedge 1 = a$	$a \vee 0 = a$	(boundedness)
$a \wedge 0 = 0$	$a \vee 1 = 1$	(boundedness)
$a \wedge (a \vee b) = a$	$a \vee (a \wedge b) = a$	(absorbtion)
$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	(distributivity)
$\overline{a \vee b} = \bar{a} \wedge \bar{b}$	$\overline{a \wedge b} = \bar{a} \vee \bar{b}$	(deMorgan)

Content

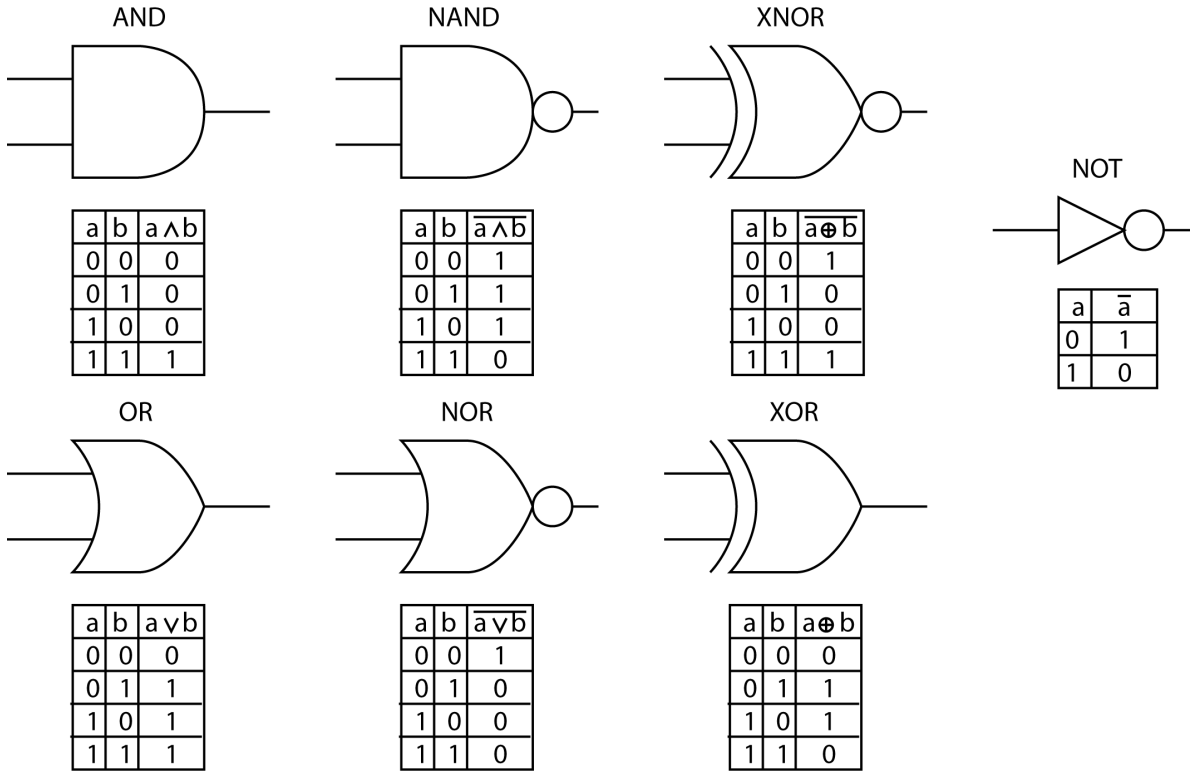
Boolean Logic

Combinational Logic Circuits

Sequential Logic Circuits

Von Neumann Machine

Logic Gates Symbols



a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$\overline{a \wedge b}$
0	0	1
0	1	1
1	0	1
1	1	0

a	b	$\overline{a \oplus b}$
0	0	1
0	1	0
1	0	0
1	1	1

a	$\bar{a}$
0	1
1	0

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

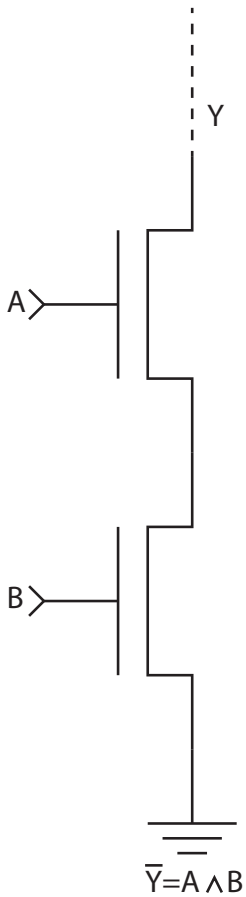
a	b	$\overline{a \vee b}$
0	0	1
0	1	0
1	0	0
1	1	0

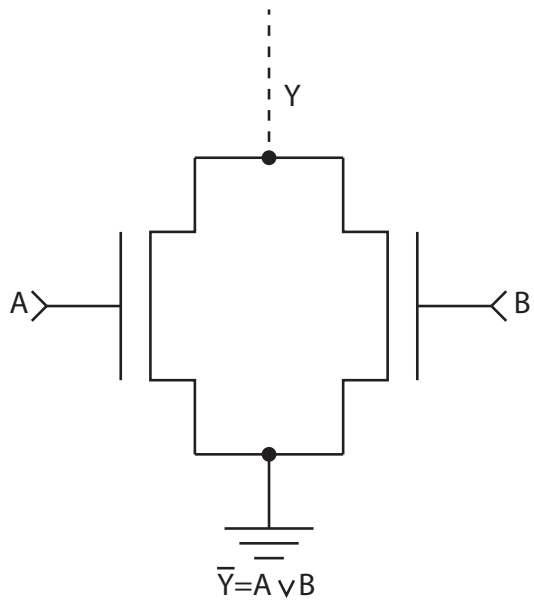
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

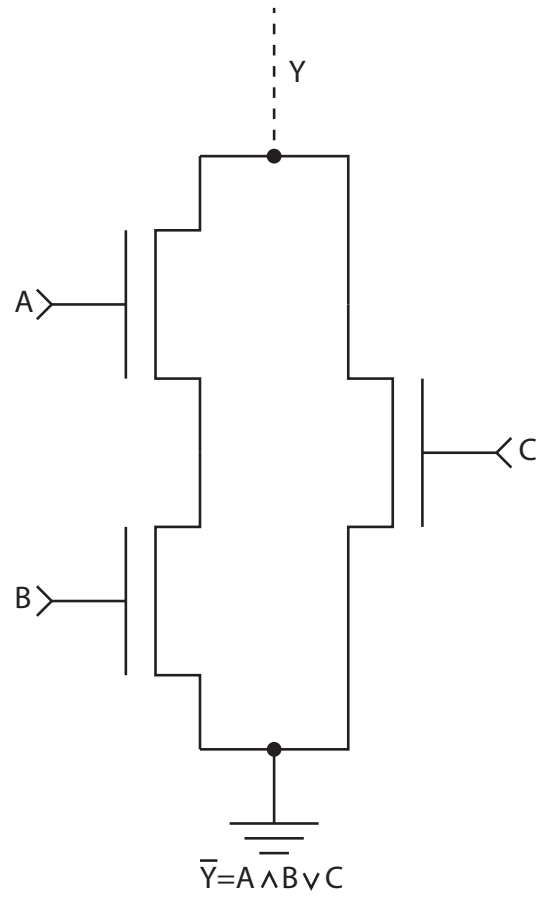
Logic '0' an '1' (also 'true'/'false' or 'H'/'L') are typically represented by voltage (less popular: current) *ranges*, e.g. between supplies Vdd and Gnd above 75% Vdd is interpreted as '1' and below 25% of Vdd is interpreted as '0' and between 25% and 75% as 'uncertain'. Beware settling times and connections between different types of logic families!

PU/PD Networks

Examples of PD networks. Can you write down corresponding Boolean Logic Expressions for some more examples? (Ask your lecturer!)

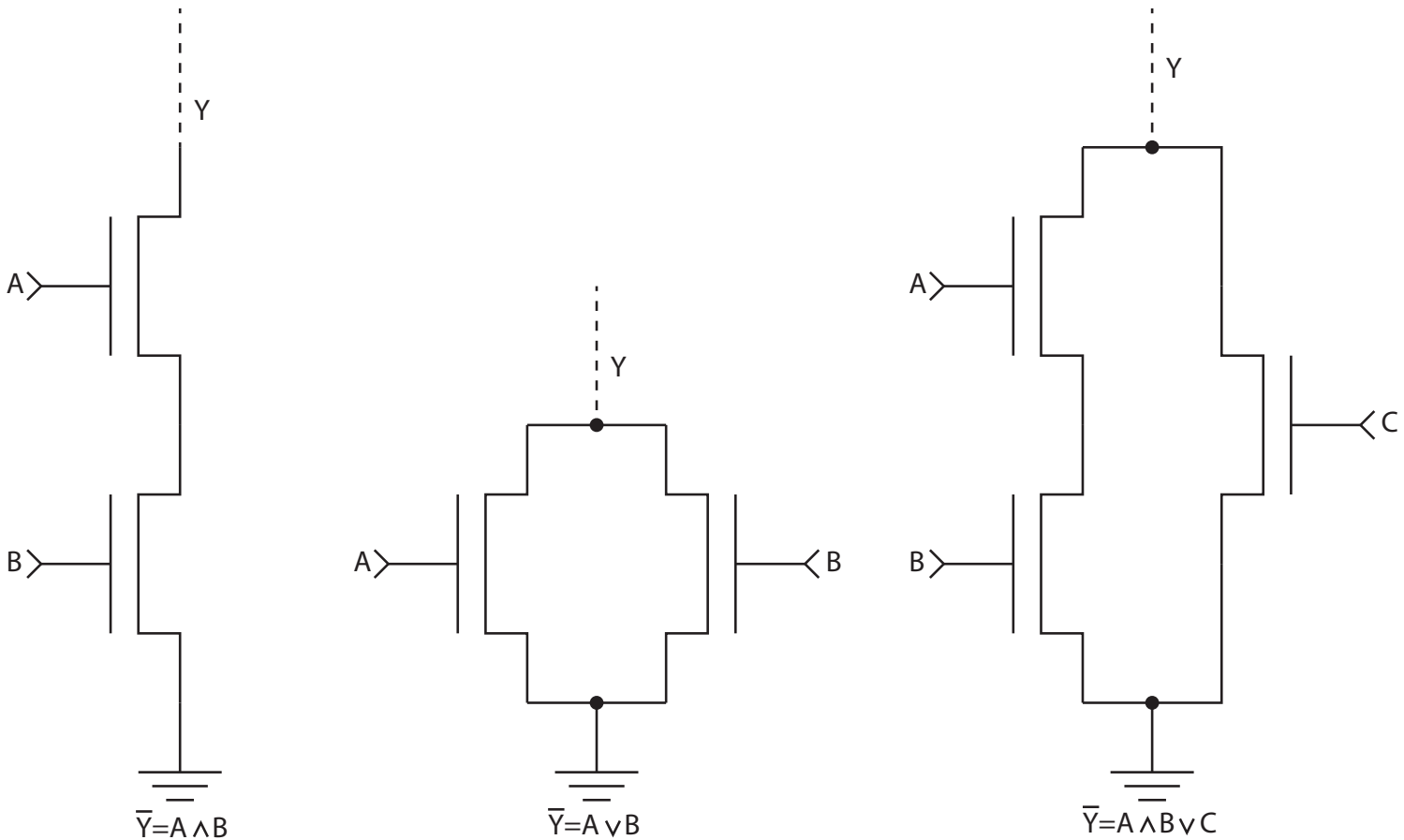






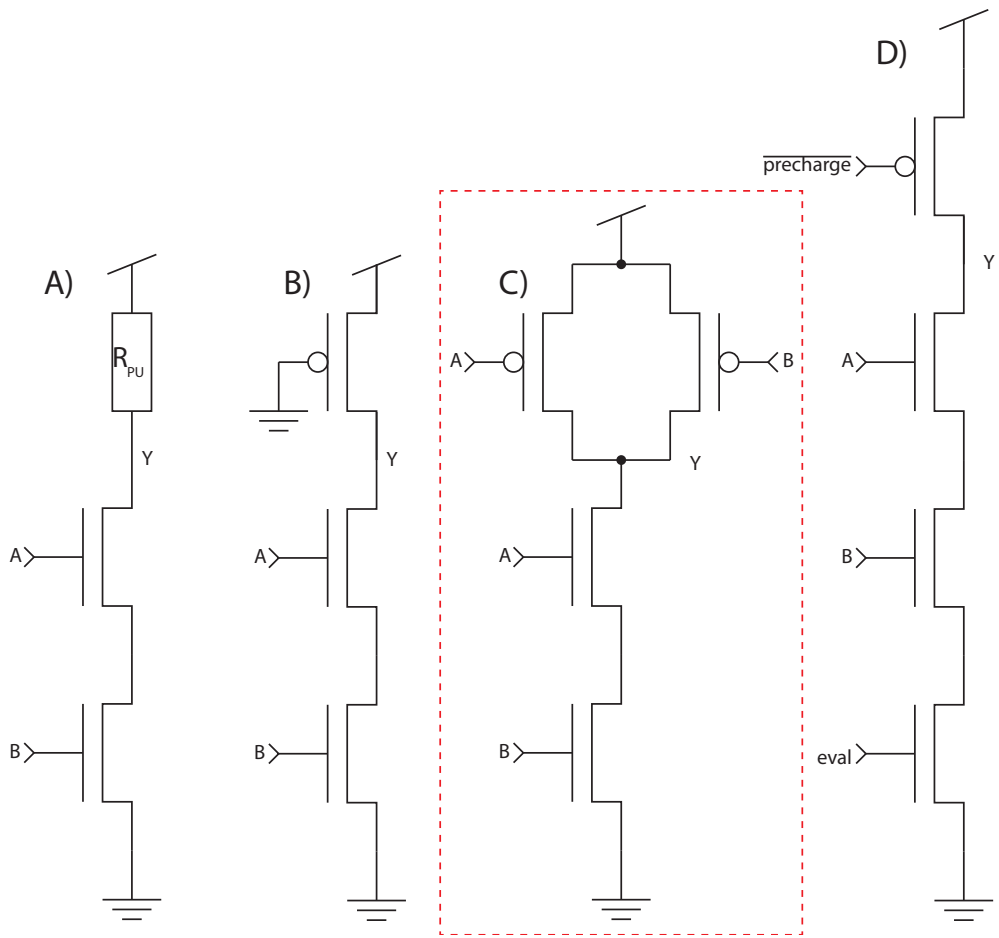
Observations PU/PD Networks

- PU: pFET; PD: nFET
- The above limits the types of Boolean functions one can implement with a single gate.
- Complex Boolean functions: avoid many transistors in series, rather do multiple gates in succession.

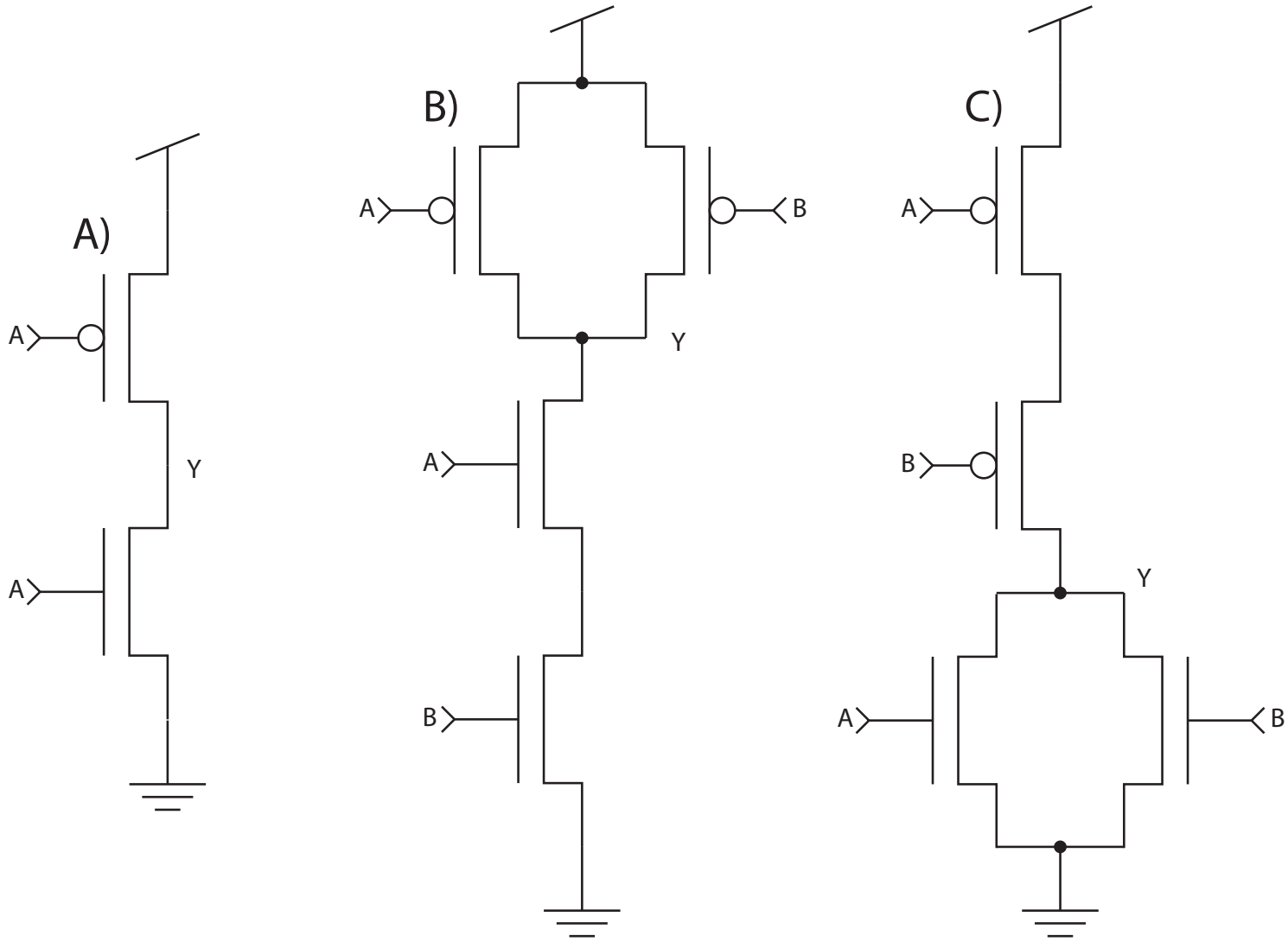


Complementing PU/PD Networks

(The red dashed box is an example with **complementary** PU and PD network, which is the most common way of going about things.)



Complementary PU/PD Basic Set of Gates



A) NOT B) NAND C) NOR

Propagation Delay t_p

Often defined as the average of the PU and PD propagation delay, where the PD propagation delay t_{pHL} is defined as the worst case time it takes to pull down the output from Vdd to $\frac{\text{Vdd}}{2}$ when the PD network is on and the PU network is off, and the PU propagation delay as the worst case time it takes to pull up the output from Gnd to $\frac{\text{Vdd}}{2}$ when the PU network is on and the PD network is off.

$$t_p = \frac{t_{pHL} + t_{pLH}}{2} \propto R_{ON} C_L^* \propto \begin{cases} k = \mu C_{OX} \frac{W}{L} \\ C_L^* = C_O + C_W + C_I \end{cases}$$

REMEMBER: derivation of R_{ON} or I_{ON} and t_p in models chapter!

Inverter R_{ON}

$$\mu_n \approx 2.5 * \mu_p$$

So if an inverter should have **symmetric** t_{pHL} and t_{pLH} as well as a switching threshold near $\frac{\text{Vdd}}{2}$ one often chooses:

$$\frac{W_p}{L_p} \approx 2.5 \frac{W_n}{L_n}$$

and usually one chooses minimum L for both and only different W:

$$\begin{aligned} L_p &= L_n \\ W_p &\approx 2.5 W_n \end{aligned}$$

Symmetry gives the best **margins for correct signal interpretation** and the **optimal worst case delay**, it might however be foregone if minimal layout space is more important.

R_{ON} and C_L^*

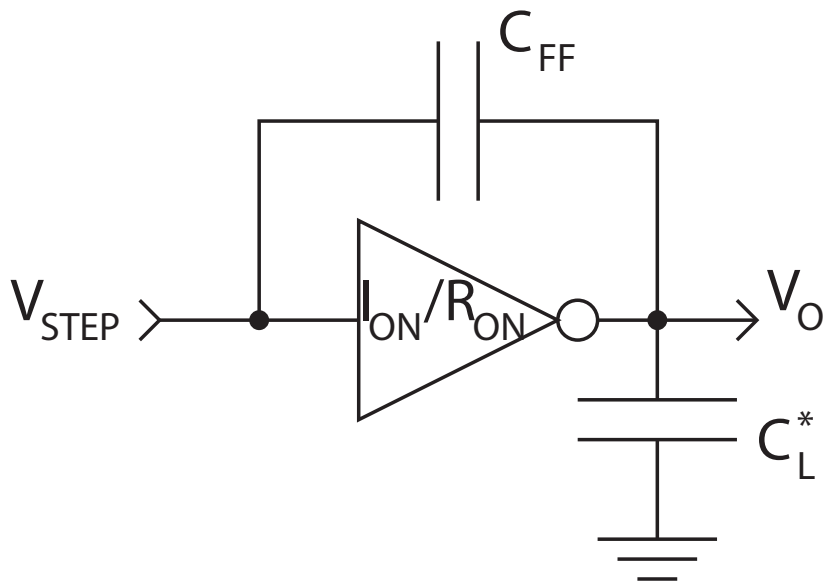
One can also choose to generally increase W to reduce R_{ON}. This is, however, a two edged sword: This also increases C_O (contributing to C_L^*) at the output, as well as C_I for any signal feeding into this gate. It might still be worth it up to a point, since it reduces the relative contribution of C_W.

$$C_L^* = C_O + C_W + C_I$$

Inverter C_L^*

For t_p the Miller effect influences the *output's* capacitance...

$$C_O = 3C_{GDp} + 3C_{GDn} + C_{DBp} + C_{DBn}$$

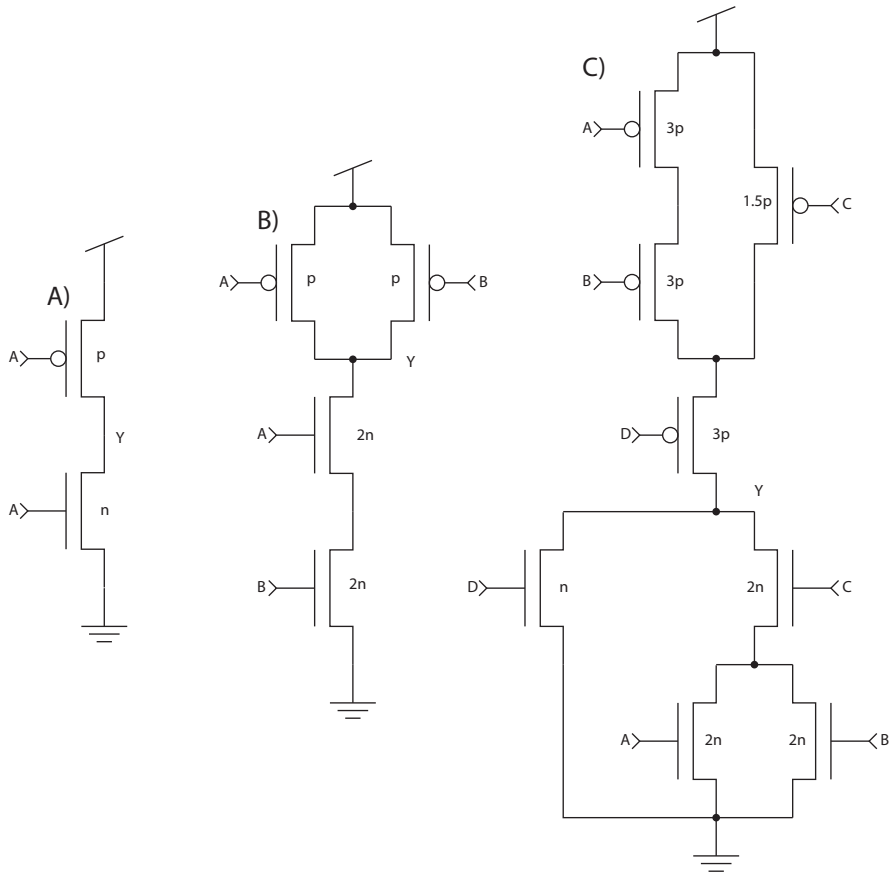


... but *not the input* capacitance (no switching before reaching the threshold). The next gate contributes its C_I to C_L^*. For the inverter:

$$C_I = C_{GDp} + C_{GDn} + C_{GSp} + C_{GSn}$$

Balancing R_{ON} for more Complex Gates

if $n = \frac{W_n}{L_n}$ and $p = \frac{W_p}{L_p}$ for an inverter with *balanced* R_{ON} one can balance more complex gates by ensuring at most a total R_{ON} for each possible PU and PD down path:



NOTE: t_p will be *worse* than for an inverter due to a bigger C_L and also 'internal' capacitances in other nodes!

Observations on Gates

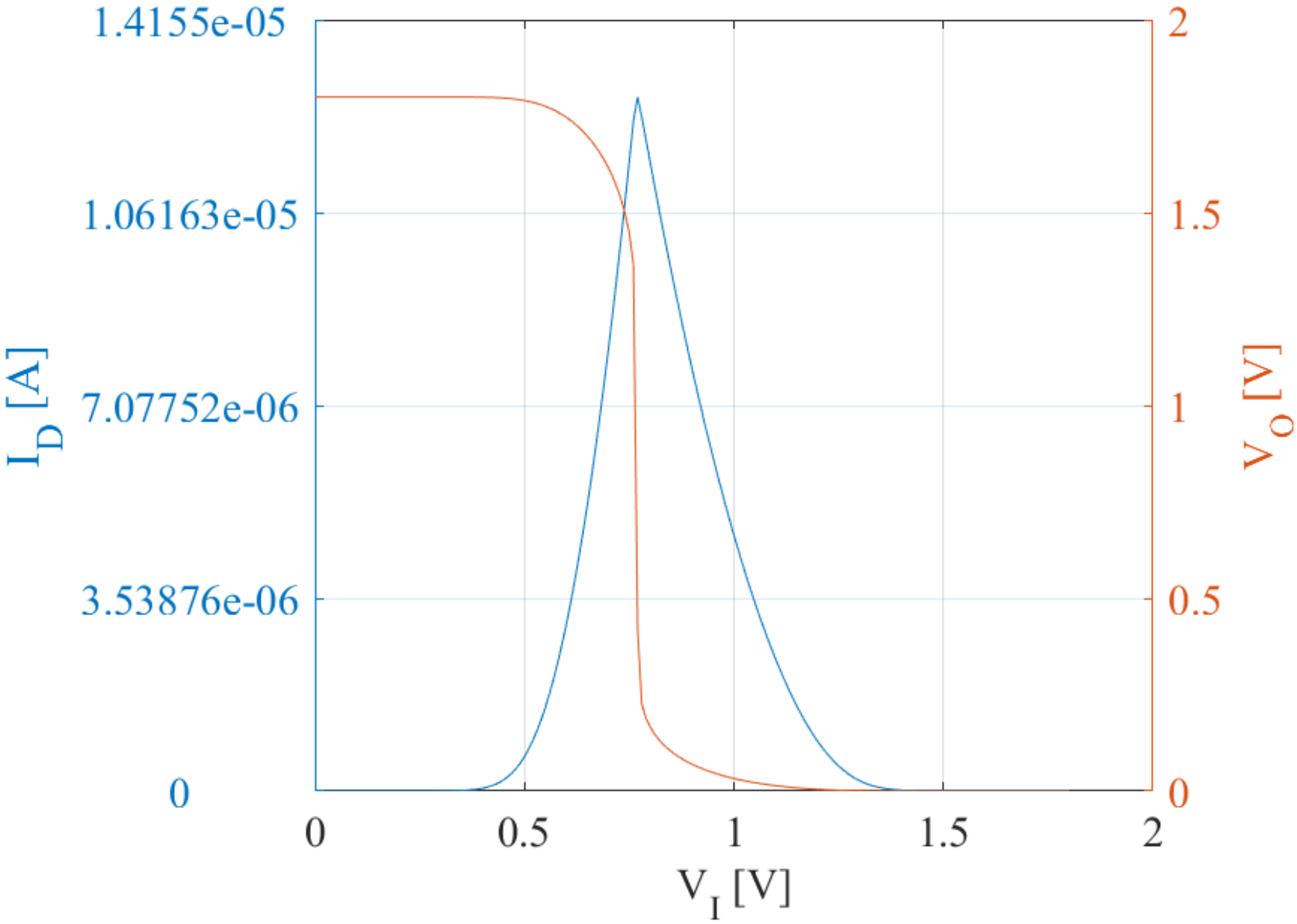
- NAND better than NOR for layout space.
- Equivalent multi stage implementations with smaller gates are often faster than Single stage with complex gates.

Dynamic and static power consumption of logic gates

Static Power is mainly leakage in PN junctions, and through transistors that are supposed to be 'off'.

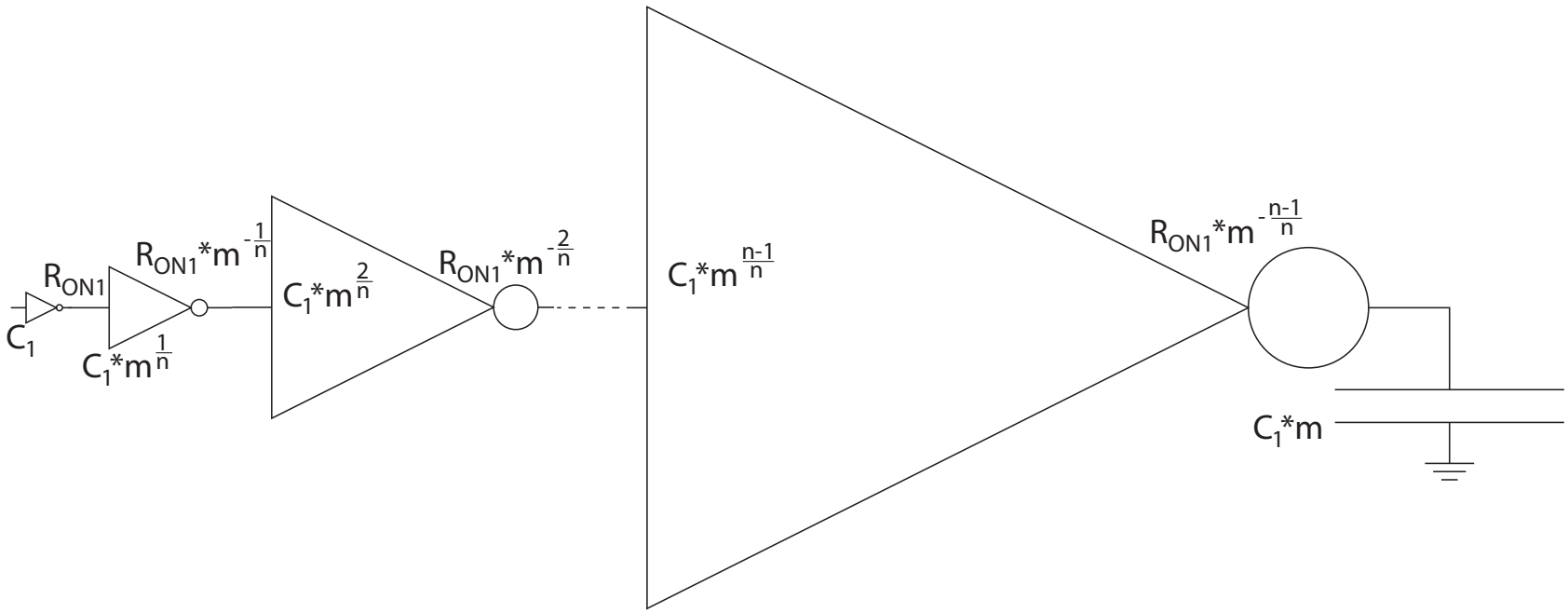
Dynamic Power is energy consumed when switching, i.e. power for charging signal nodes proportional to $C_L^* * \text{Vdd}^2$, but also the so called cross current when the input to a gate is in transition and just for a moment there is a partially open PU *and* PD path, i.e. a direct conductance between Vdd and Gnd.

Inverter Cross-Current



Speed when Driving Large (External/PAD) Loads

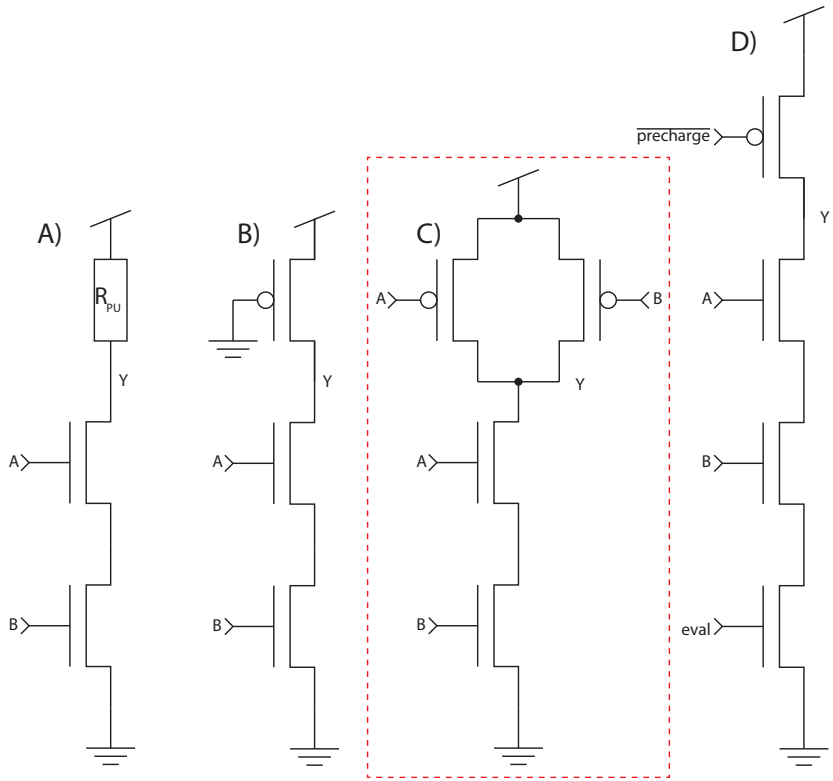
A single puny CMOS inverter with $C_I = C_1$ driving a load $m * C_1$ takes m times longer than just driving C_1. This can be improved by using n inverters with increasing $\frac{W_{k+1}}{W_k} = m^{\frac{1}{n}}$. It can be shown that the ideal factor $m^{\frac{1}{n}} = e$ and the ideal $n = \ln m$.



Since m is often not known at design time one often is pragmatic about it using $n = 2$ or $n = 4$ and the ratio $\frac{W_{k+1}}{W_k}$ quite arbitrarily a good deal bigger than e. Thus, there will be improvement, but not the optimal one.

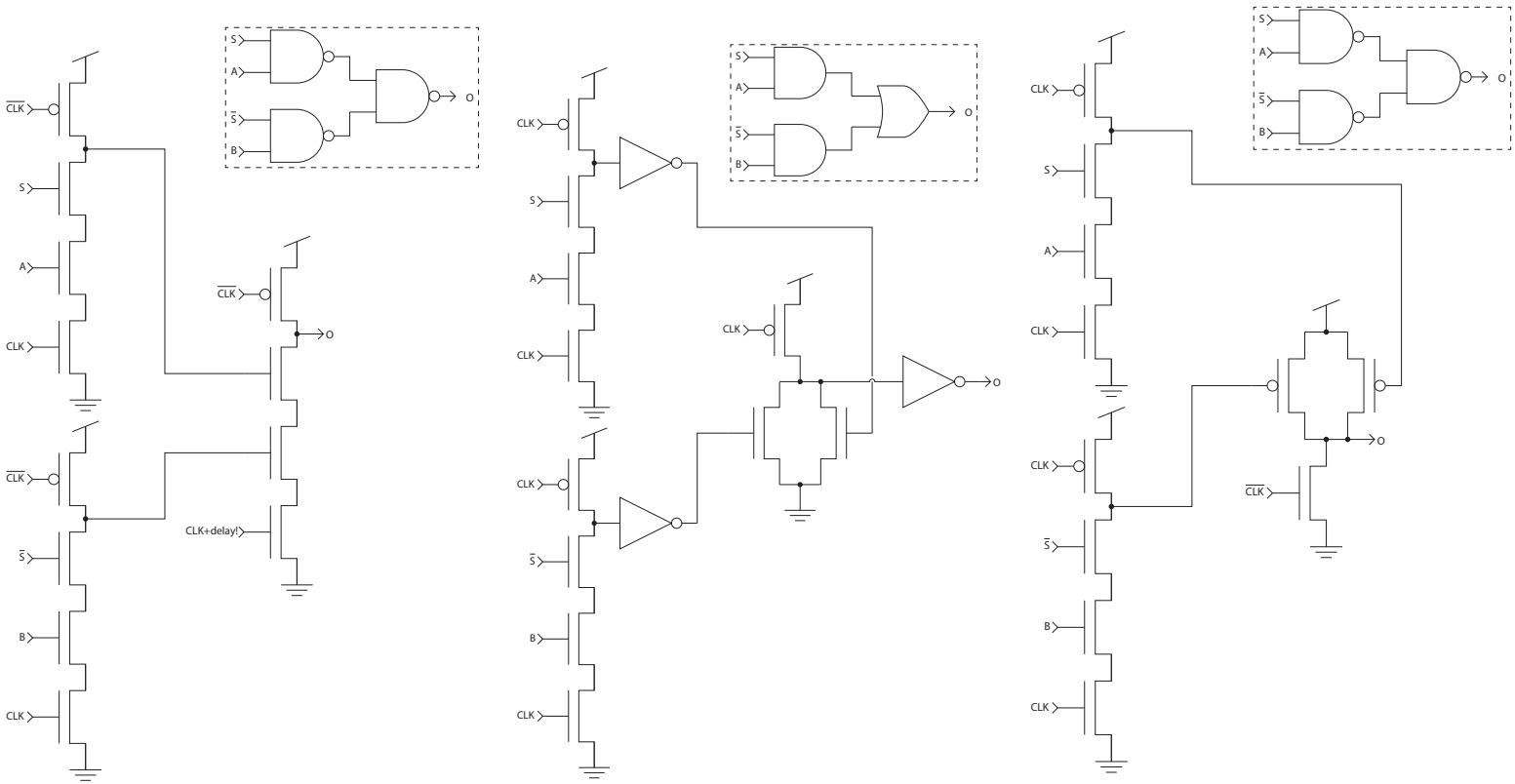
Dynamic Logic

Variant D) in the figure implements only the PD path, and uses a 'precharge' switch to pull up in the precharge phase. In the ensuing evaluation phase the signal stays in its precharge state or switches.

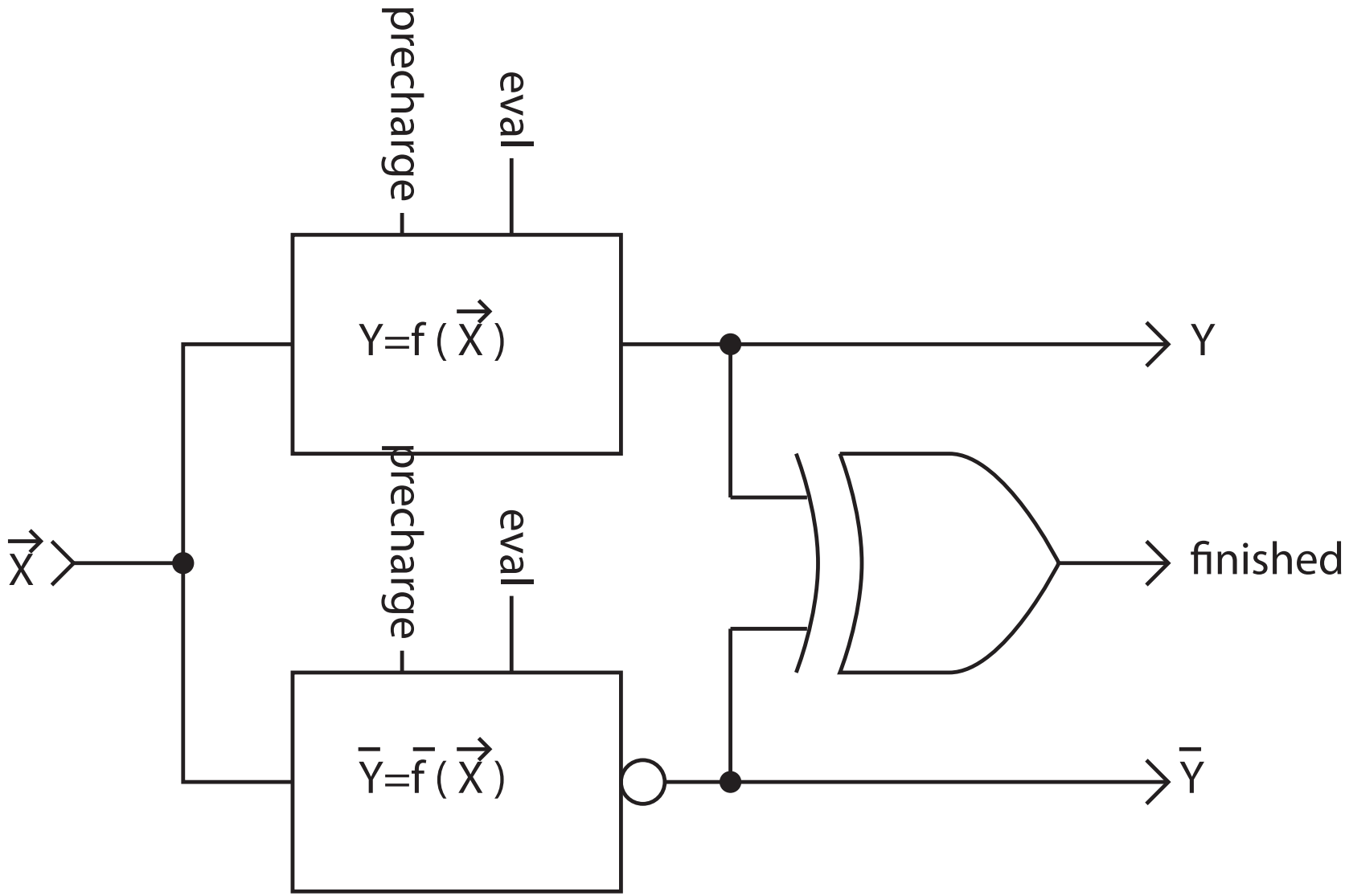


Advantages? Less layout space and less C_L^* ...
Problems? ...

NP Domino Logic



Dual Rail Domino Logic



A useful concept in asynchronous digital circuits

Briefly mentioned: some standard combinational logic circuits

- decoders
- encoders
- multiplexers
- demultiplexers
- simple Arithmetic logic unit (ALU)

Content

Boolean Logic

Combinational Logic Circuits

Sequential Logic Circuits

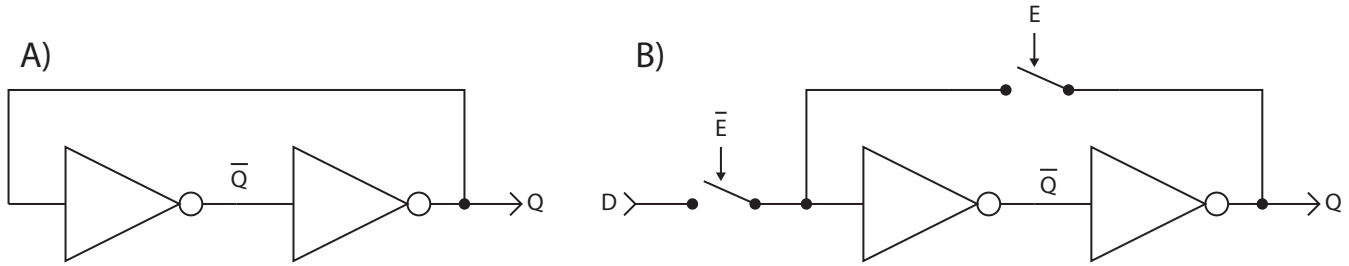
Von Neumann Machine

Sequential vs Combinational

Combinational: No Feedback, no storage/memory, purely 'functional', i.e. implementing boolean functions

Sequential: Containing feedback to implement memory, implementing sequences of operations on memory units, executing a program

D-latch

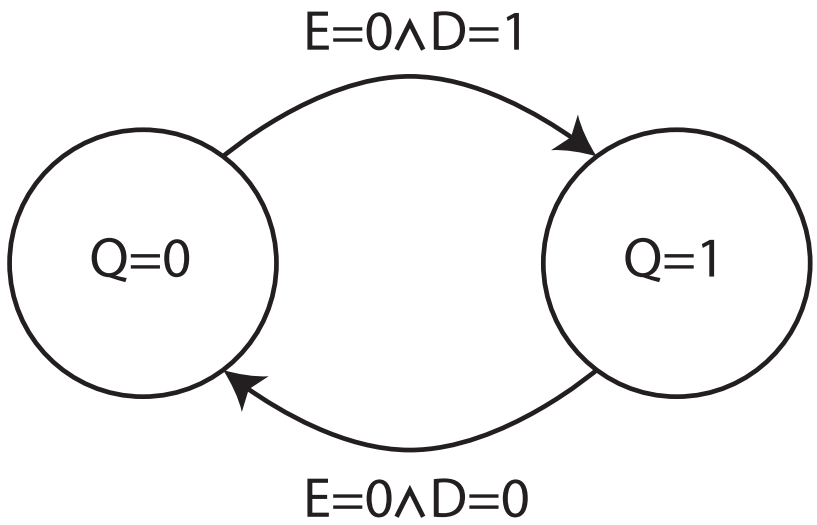


Characteristic table (a truth table 'expanded' in time)

D	E	Q	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

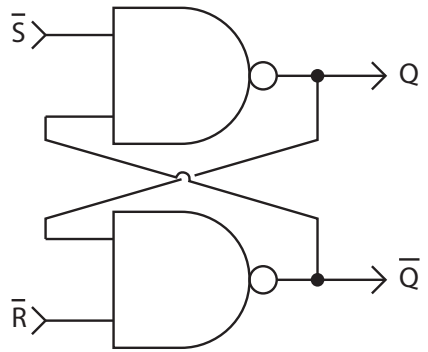
E	Q
0	D
1	Q

State transition diagram

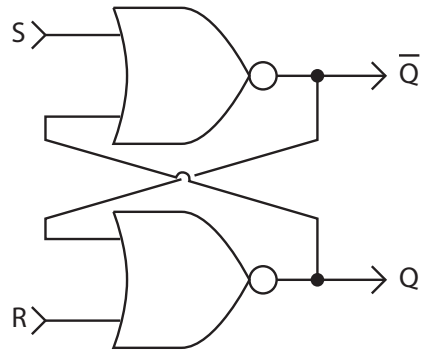


SR-Latch

A)



B)

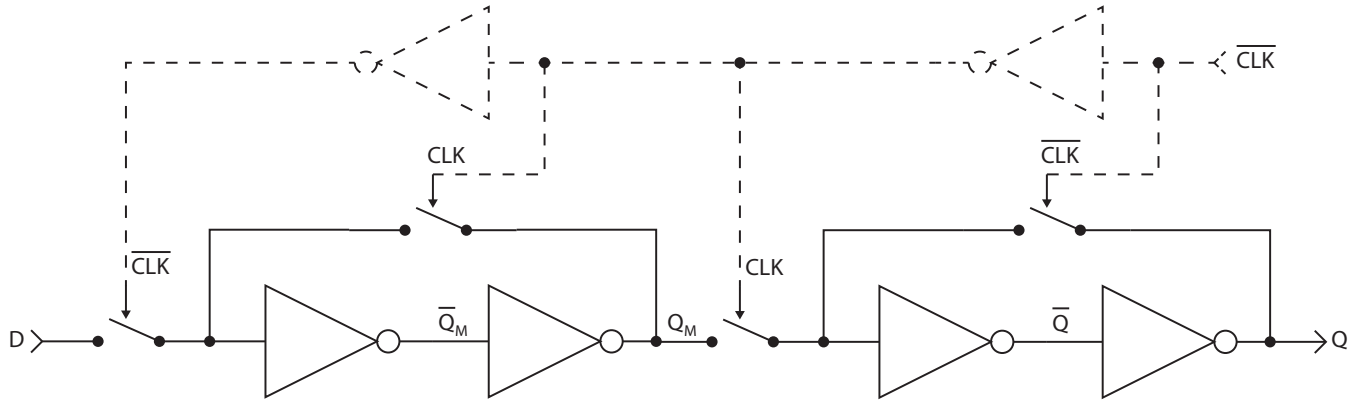


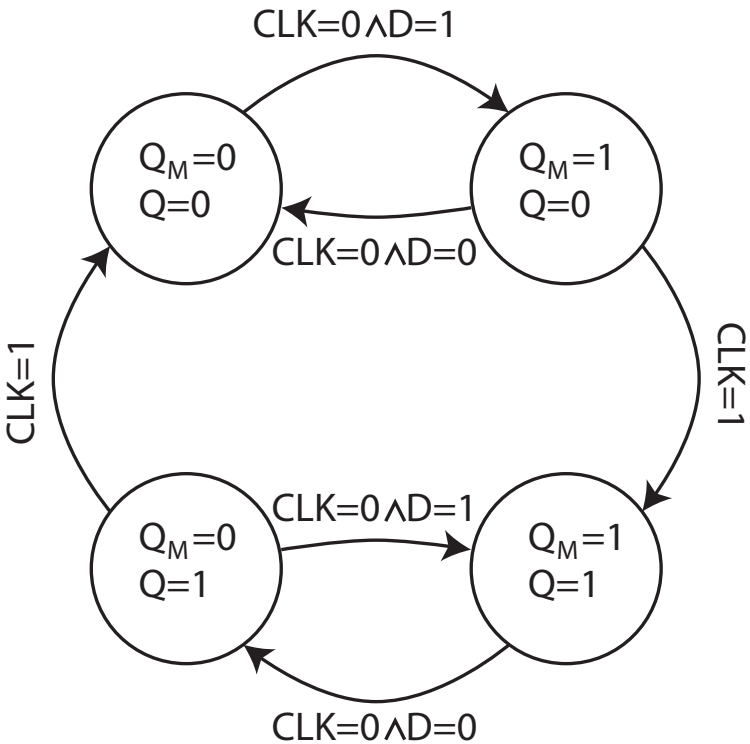
R	S	Q	Q	$\overline{Q}$
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	0
1	1	0	0	0

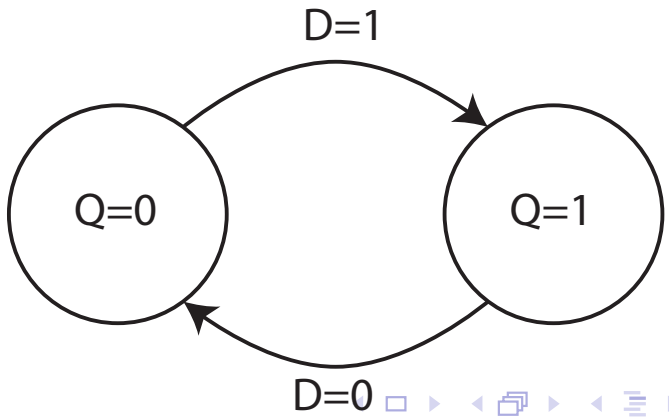
R	S	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	1	0
1	0	0	1
1	1	0	0

From level- to edge triggered: D-flip-flop

(Note: the nomenclature where 'latch' is used for level-triggered and 'flip-flop' for edge-triggerd is not consequently applied in literature!)

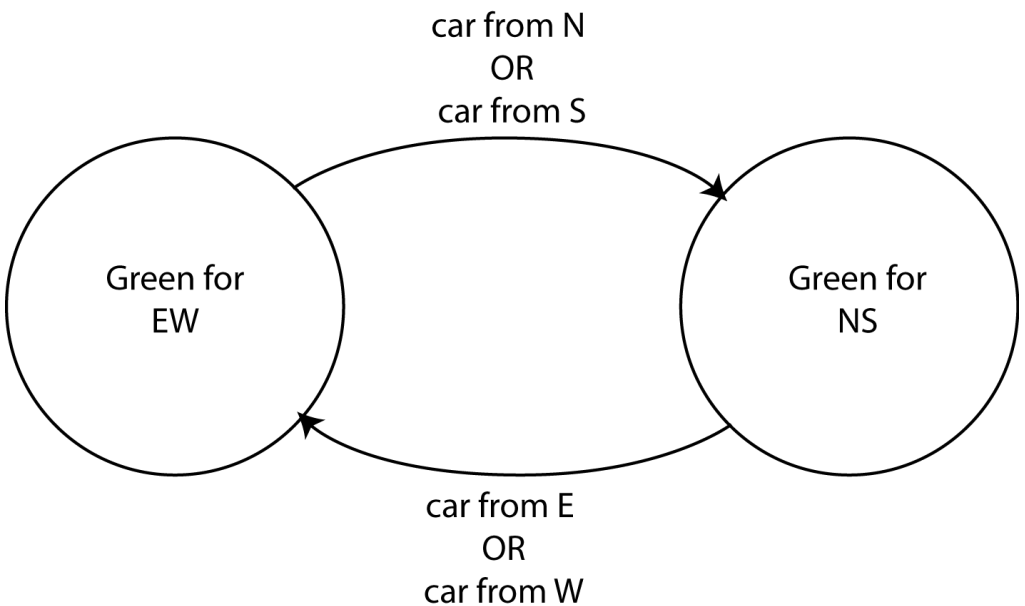






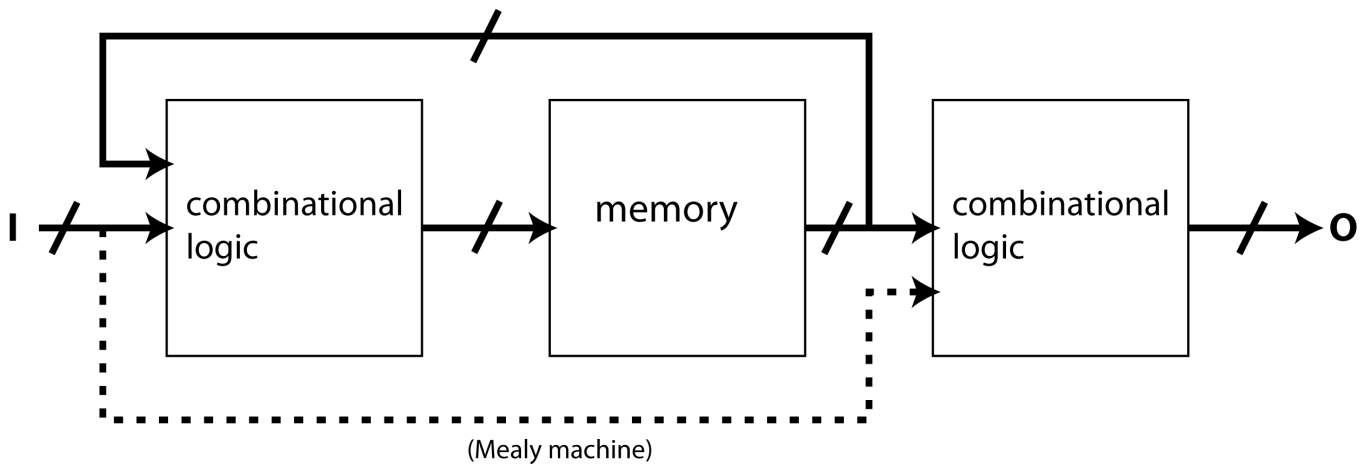
Finite State Machines

More generally, state transition graphs define so called finite state machines (FSM). Synchronous state machines are more popular than asynchronous, and state transitions are implicitly edge triggered by a global clock signal that is not shown.
State transition conditions are boolean functions of both state variables and external (sensor) inputs. States are implemented by a number of memory cells, typically D-flipflops (registers).
Example: traffic light controller



Mealy and Moore FSMs

Thus, FSMs have a typical structure where a combinational logic block receives input from external sources (e.g. sensors or user controlled switches) as well as from a register storing the present state. The output of that block then sets the next state. Control outputs from th FSM may be directly the bits of the state register or another combinational logic block derives output signals from the state register. The Mealy variant may also use inputs directly feeding into the output combinational logic, an approach that has its perils (let's discuss!).

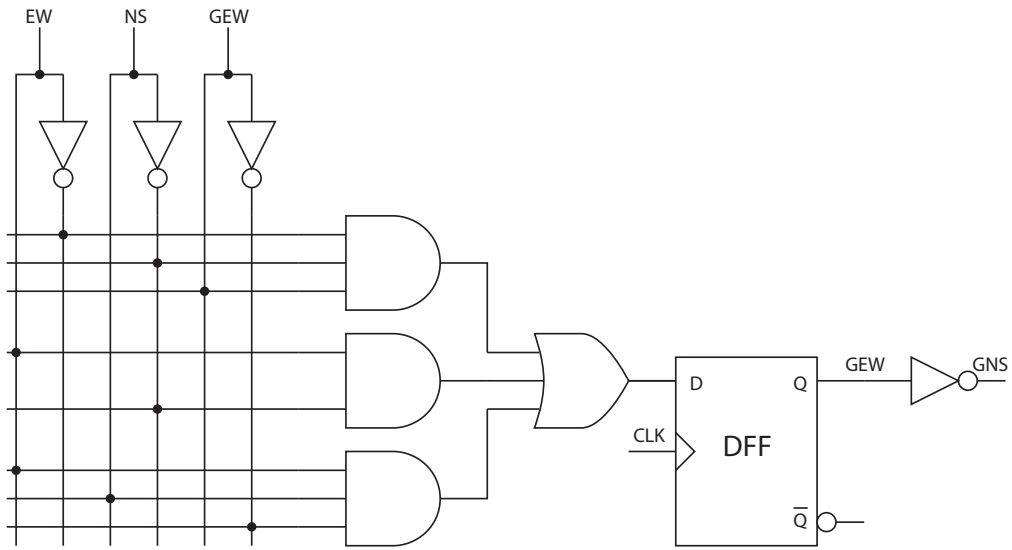


(Mealy machine)

Implementing the traffic light controller

EW	NS	GEW	GEW
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$$
\begin{aligned}
GEW = & \\
& (\overline{EW} \wedge \overline{NS} \wedge GEW) \\
& \vee (EW \wedge \overline{NS} \wedge \overline{GEW}) \\
& \vee (EW \wedge \overline{NS} \wedge GEW) \\
& \vee (EW \wedge NS \wedge \overline{GEW}) \qquad (1)
\end{aligned}
$$



Briefly mentioned: some standard sequential logic circuits

- counters
- appliance controllers (dishwashers, fridges, thermostats, ...)
- control circuit for SAR ADCs
- complex Arithmetic logic unit (ALU)

Content

Boolean Logic

Combinational Logic Circuits

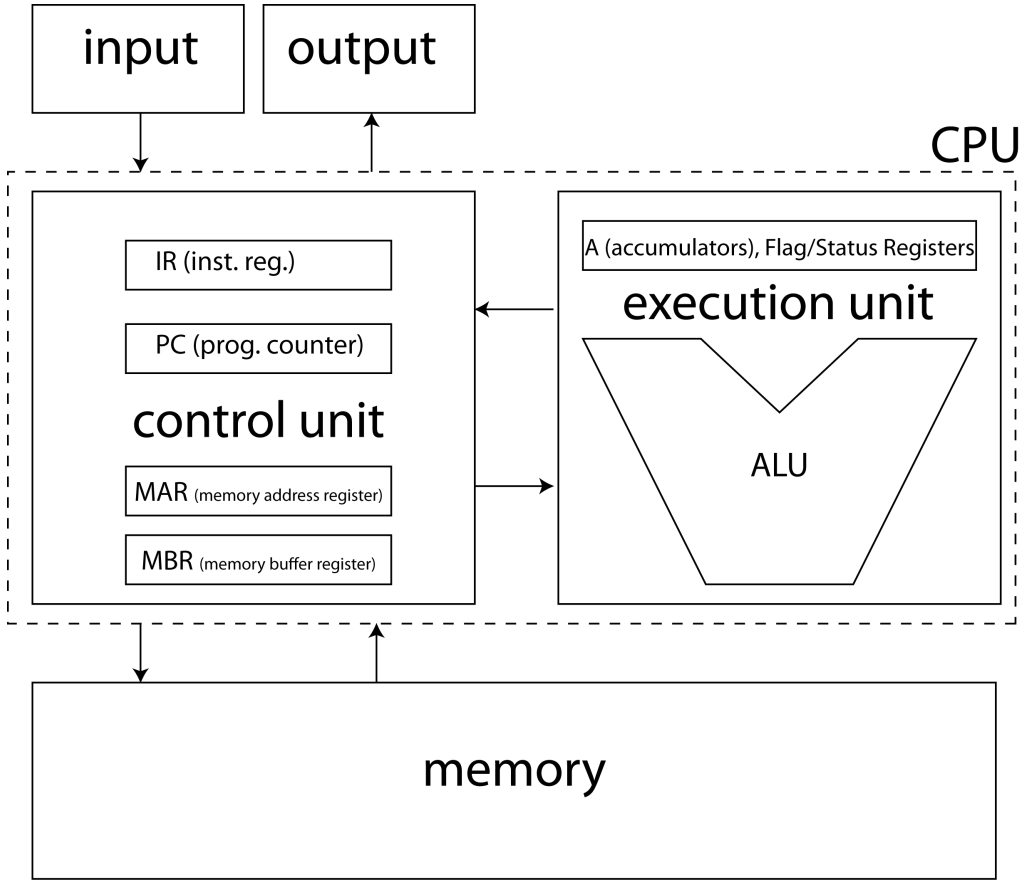
Sequential Logic Circuits

Von Neumann Machine
- Execution Unit
- Memory
- DRAM
- Control Unit

Von Neumann Architecture

In 1945 John von Neumann published his reference model of a computer architecture that is still the basis of most computer architectures today. The main novelty was that a single memory was used for both, program *and* data.

Von Neumann Architecture Block Diagram



Typical Registers(1/3)

PC: (program counter, also called instruction pointer (IP)) the register holding the memory address of the next machine code instruction.

IR: (instruction register) the register holding the machine code of the instruction that is executed.

Typical Registers(2/3)

MAR: (memory address register) half of the registers dedicated as interface of the memory with the CPU, holding the memory address to be read or written to.

MBR: (memory buffer register) the other half of the CPU-memory interface, a buffer holding the data just read from the memory or to be written to the memory. Typically the MBR can be connected as one of the inputs to the ALU.

Typical Registers(2/3)

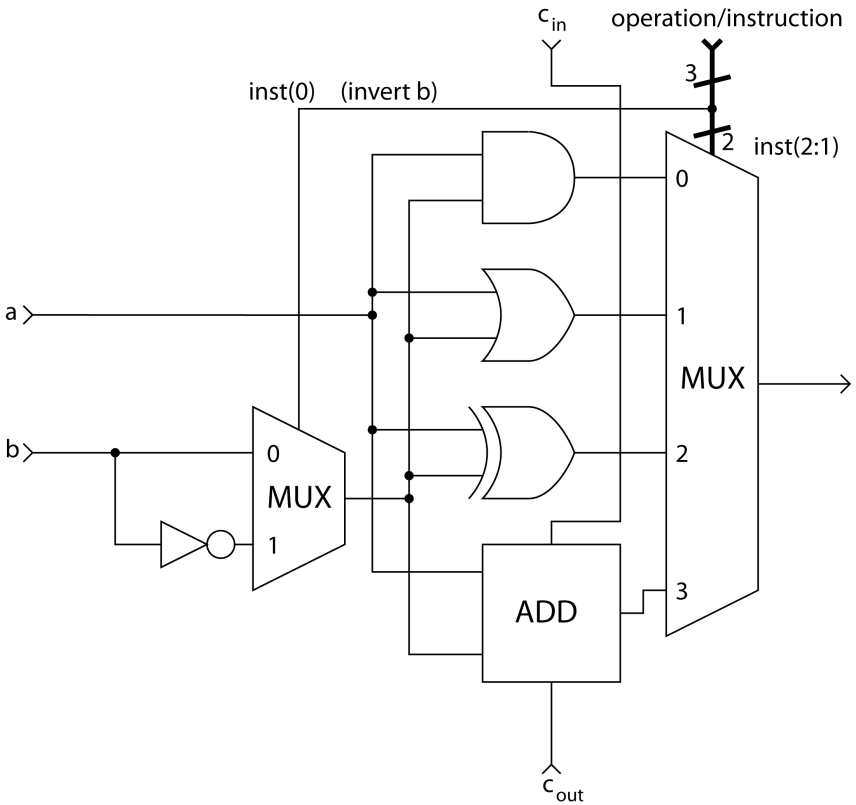
accumulator: a dedicated register that stores one operand and the result of the ALU. Several accumulators (or general purpose registers in the CPU) allow for storing of intermediate results, avoiding (costly) memory accesses.

flag/status register: a register where each single bit stands for a specific property of the result from (or the input to) the ALU, like carry in/out, equal to zero, even/uneven ...

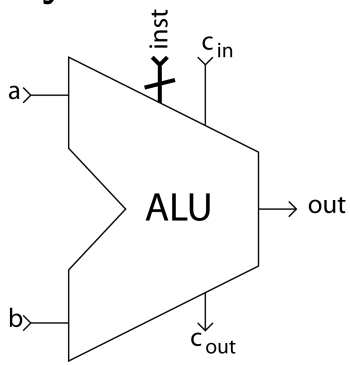
Data and Instruction Bus

Buses are connections between registers, the functional units (such as the ALU), the memory, and I/O units. They are often shared by several of those units and usually only one unit sends data on the bus to one other at a time. Since there is only one bus between the memory and the CPU for both instruction and data transfer in a von Neumann architecture (actually two: one for the address and one for the data), this bus can be a main speed limiting factor (von Neumann bottleneck). Internally in the CPU there is one or several buses for data exchange among the registers.

A 1-bit Arithmetic Logic Unit (ALU) Example

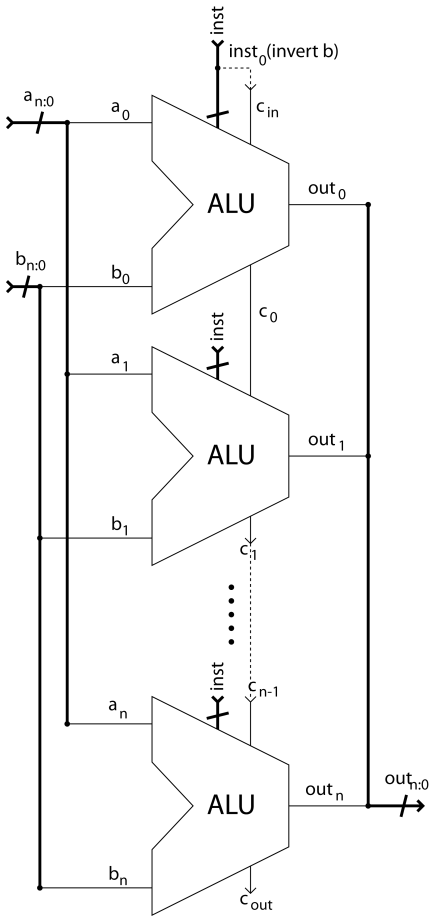


Symbol:

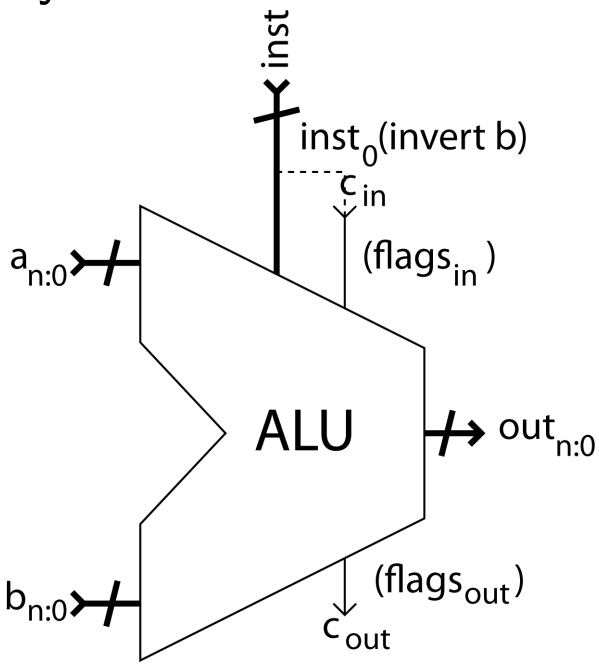


inst	computation
000	$a \wedge b$
001	$a \wedge \overline{b}$
010	$a \vee b$
011	$a \vee \overline{b}$
100	$a \oplus b$
101	$a \oplus \overline{b}$
110	$a + b$
111	$a - b^{(!)}$

n-bit ALU example



Symbol:

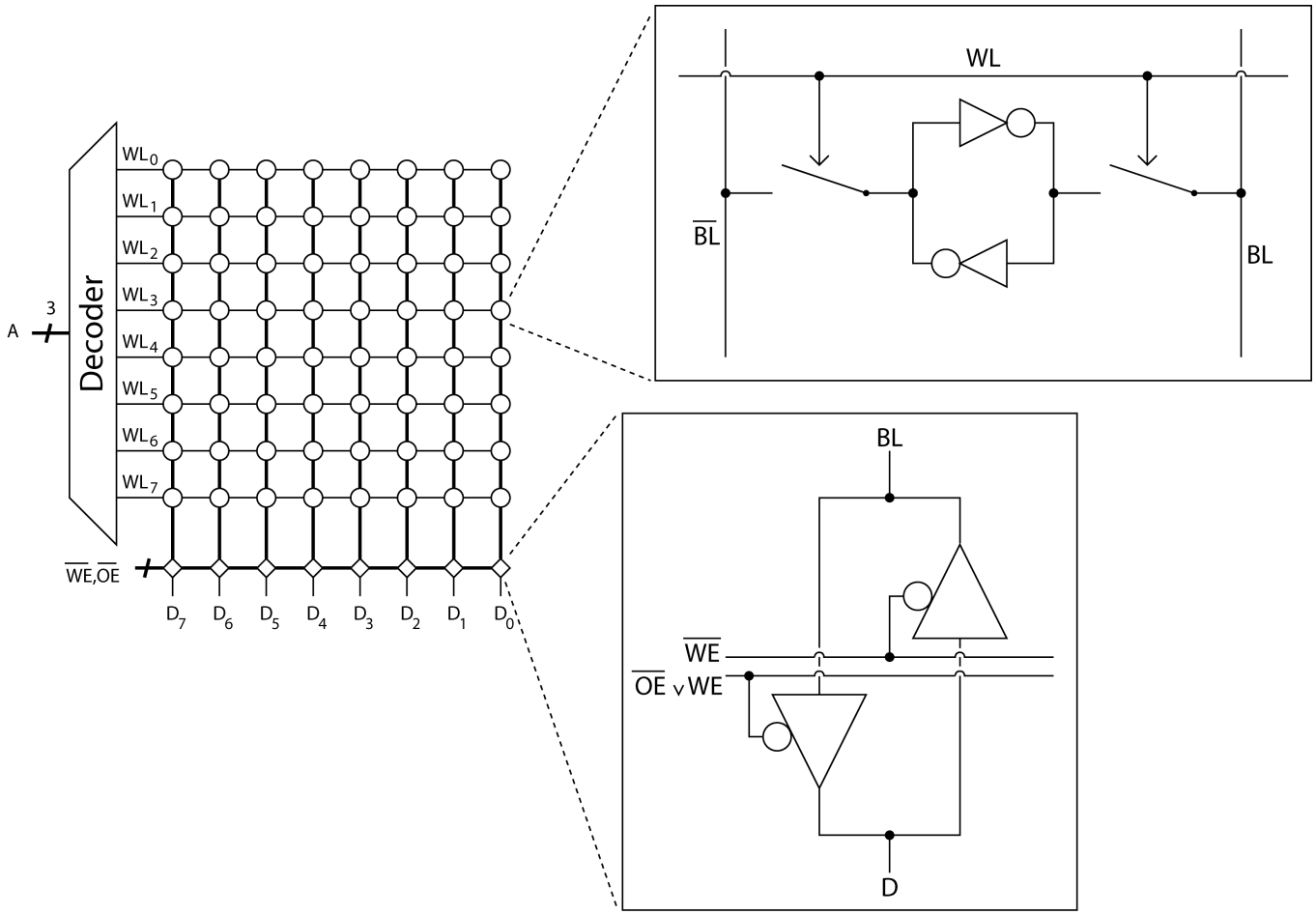


More complicated ALUs will have more flags, e.g. overflow, divide by zero ...

ALUs in CPUs

Modern CPUs contain several ALUs, e.g. one dedicated to memory pointer operations and one for data operations. ALUs can be much more complex and perform many more functions in a single step than the example shown here, but note that even a simple ALU can compute complex operations in several steps, controlled by the software. Thus, there is always a trade-off of where to put the complexity: either in the hardware or in the software. Complex hardware can be expensive in power consumption, chip area and cost. Furthermore, the most complex operation may determine the maximal clock speed. The design of the ALU is a major factor in determining the CPU performance!

Static Random Access Memory Principle



RAM terminology

address space: the number of addressable units (either words or bytes) in a RAM. In the previous example its equivalent to the number of rows.

word length: The number of bits that can be accessed in a single read/write operation, i.e. the number of bits addressed with a single address. In the previous example the number of columns.

memory size: adressable unit multiplied with address space or equivalent the number of words multiplied with word size.

Most Typical RAM Signals (1/3)

$\overline{\text{WE}}$, write enable (often active low): This signal distinguishes a read from a write access. If there is no $\overline{\text{RAS}}/\overline{\text{CAS}}$ input to the RAM, $\overline{\text{WE}}$ going low directly causes D to be stored into the RAM at address A.

Most Typical RAM Signals (2/3)

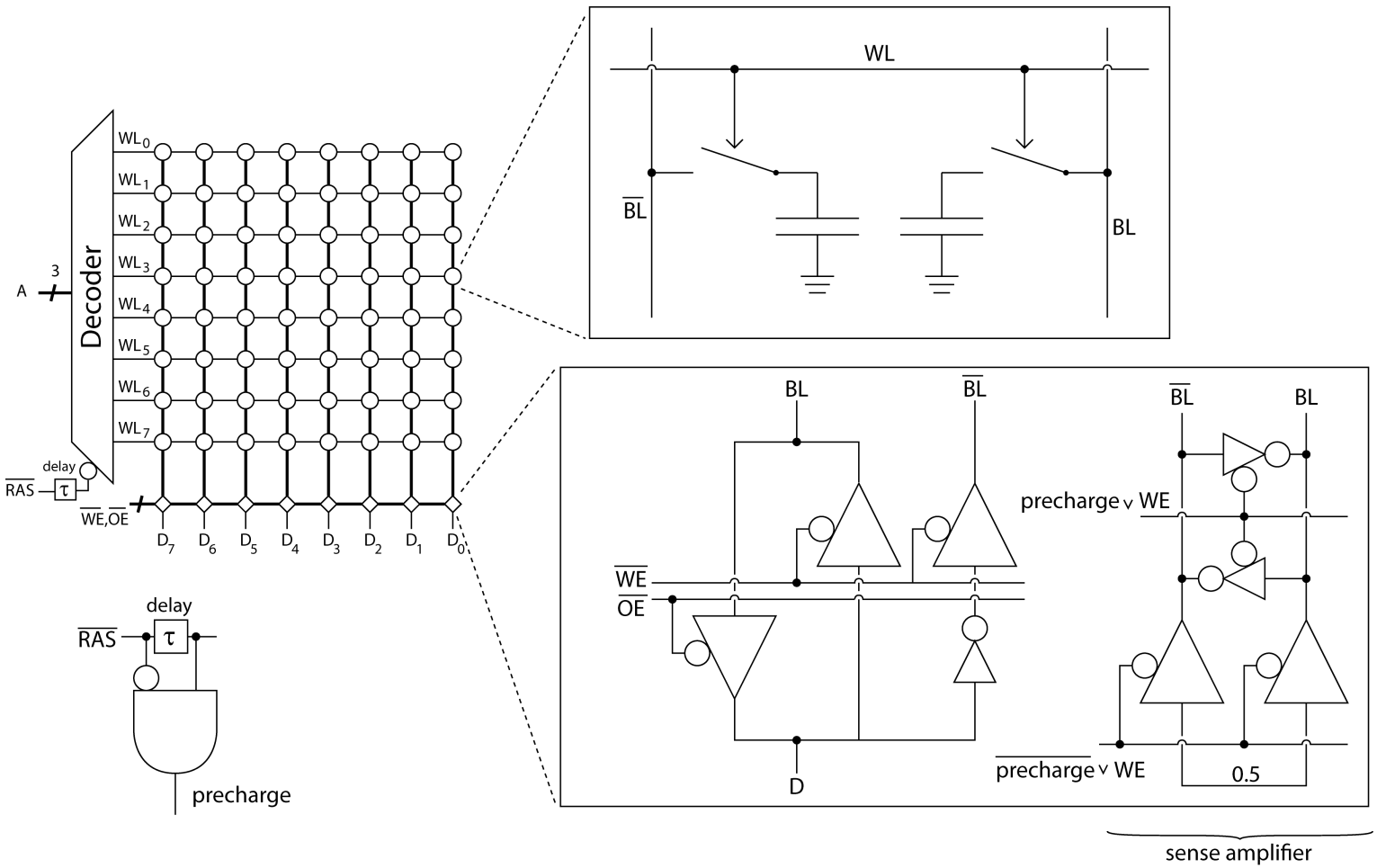
$\overline{\text{RAS}}/\overline{\text{CAS}}$, row/column access strobe: appears in DRAM that actually has a 3-D structure: one decoder for the row address, one for the column address and the word (conceptually) extends into the third dimension. The address bus is reused for both row and column address. First the row address is asserted on the address bus A and $\overline{\text{RAS}}$ is pulsed low, then the column address is asserted on the address bus and $\overline{\text{CAS}}$ is pulsed low. $\overline{\text{CAS}}$ is the final signal that triggers the either read or write. The other signals are asserted *before*. Several column accesses can be made for a single row access for faster access times.

Most Typical RAM Signals (3/3)

$\overline{\text{OE}}$, output enable: A signal that regulates the access if there are several devices using the bus. Only one of them should be allowed to drive the bus at anyone time.

$\overline{\text{CS}}$, chip select: A control line that allows to use several RAMs instead of just one on the same address bus and sharing all other control signals. If $\overline{\text{CS}}$ is not asserted all other signals are ignored and the output is not enabled. Extra address bits are used to address one specific RAM and a decoder issues the appropriate $\overline{\text{CS}}$ to just one RAM at a time. This extends the address space.

Dynamic Random Access Memory Principle



DRAM refresh

Capacitive storage is not self maintaining like flip flops. Memory content is lost over time. Thus, the sense amplifier has to be connected to every memory cell within a given time period, while the memory is idle. In modern DRAM internal state machines take care of this refresh cycle.

Static vs. Dynamic RAM

	SRAM	DRAM
access speed	+	-
memory density	-	+
no refresh needed	+	-
simple internal control	+	-
price per bit	-	+

Register Transfer Language (RTL)

expression	meaning
X	register X or unit X
[X]	the content of X
$\leftarrow$	replace/insert or execute code
M()	memory M
[M([X])]	memory content at address [X]

RTL Examples

$[IR] \leftarrow [MBR]$	transfer the content of the MBR to the IR

A Simple Example CPU Executing Machine Code

At start-up of the CPU the program counter is initialized to a specific memory address. Here to address 0. The memory content is as follows:

mem adr	content
0	move 4
1	add 5
2	store 6
3	stop
4	1
5	2
6	0
⋮	⋮

Fetch

At the beginning of a instruction cycle a new instruction is fetched from the memory. A finite state machine in the control unit generates the right sequence of control signals. Actually the CU *is* nothing but a finite state machine controlled by the instructions.

$[\text{MAR}] \leftarrow [\text{PC}]$
$[\text{PC}] \leftarrow [\text{PC}] + 1$
$[\text{MBR}] \leftarrow [\text{M}([\text{MAR}])]$
$[\text{IR}] \leftarrow [\text{MBR}]$

Decode

As a last stage of the fetch phase the operation code ('move') of the instruction is decoded by the control unit (CU).

$\text{CU} \leftarrow [\text{IR}(\textit{opcode})]$

and triggers a cycle of the finite state machine with the appropriate signals to execute a sequence of operations specific to the instruction. The order, type and number of the individual operations may vary among different instructions and the set of instructions is specific to a particular CPU.

Machine Code

The other part of the 'machine code' instruction in our (16-bit) processor is the 'operand' 4. What we have written as 'move 4' is actually a bit pattern:

$$\underbrace{10110010}_{\text{opcode: move}} \qquad \underbrace{00000100}_{\text{operand: 4}}$$

As mentioned before, the set of instructions and the machine codes are specific to a family of CPUs defined by their 'instruction set'. Machine code is *not portable* between different CPU families.

Execute

The data from memory location 4 (1) is moved to the accumulator A (often the accumulator is the implicit target of instructions without being explicitly defined):
$[\text{MAR}] \leftarrow [\text{IR}(\textit{operand})]$
$[\text{MBR}] \leftarrow [\text{M}([\text{MAR}])]$
$[\text{A}] \leftarrow [\text{MBR}]$

2nd Instruction (1/2)

A fetch and decode exactly like before:
[MAR] $\leftarrow$ [PC] (now [PC]=1)
[PC] $\leftarrow$ [PC] $+ 1$
[MBR] $\leftarrow$ M([MAR])
[IR] $\leftarrow$ [MBR]
The instruction in the IR is now 'add 5'.
CU $\leftarrow$ [IR(*opcode*)]

2nd Instruction (2/2)

Again the accumulator is the implicit target of the instruction:
[MAR] $\leftarrow$ [IR(*operand*)]
[MBR] $\leftarrow$ [M([MAR])]
The ALU receives the appropriate instruction from the state machine triggered by the opcode, or sometimes parts of the opcode *are* the instruction for the ALU.
ALU $\leftarrow$ [A];ALU $\leftarrow$ [MBR]
[A] $\leftarrow$ ALU
The number from memory location 5 (2) has been added and the result (3) is stored in the accumulator.

3rd Instruction

Fetch and decode like before (not shown).

$\vdots$

and then executing a write operation to the memory:

$[\text{MBR}] \leftarrow [\text{A}]$

$[\text{MAR}] \leftarrow [\text{IR}(\textit{operand})]$

$[\text{M}([\text{MAR}])] \leftarrow [\text{MBR}]$

4^{th} Instruction

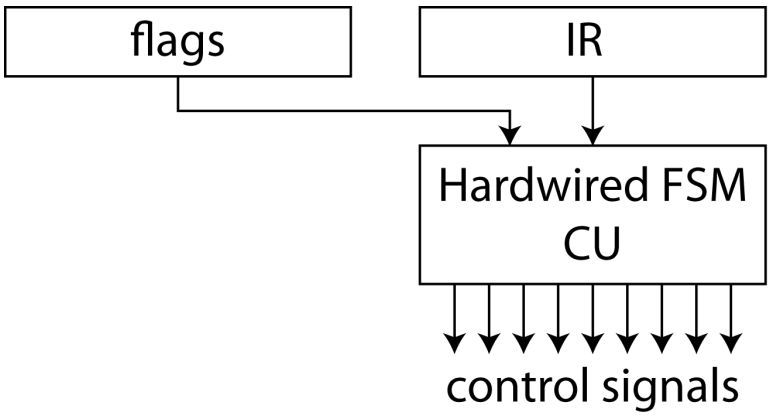
The forth instruction is a stop instruction which halts the execution of the program. The memory content is now changed to:

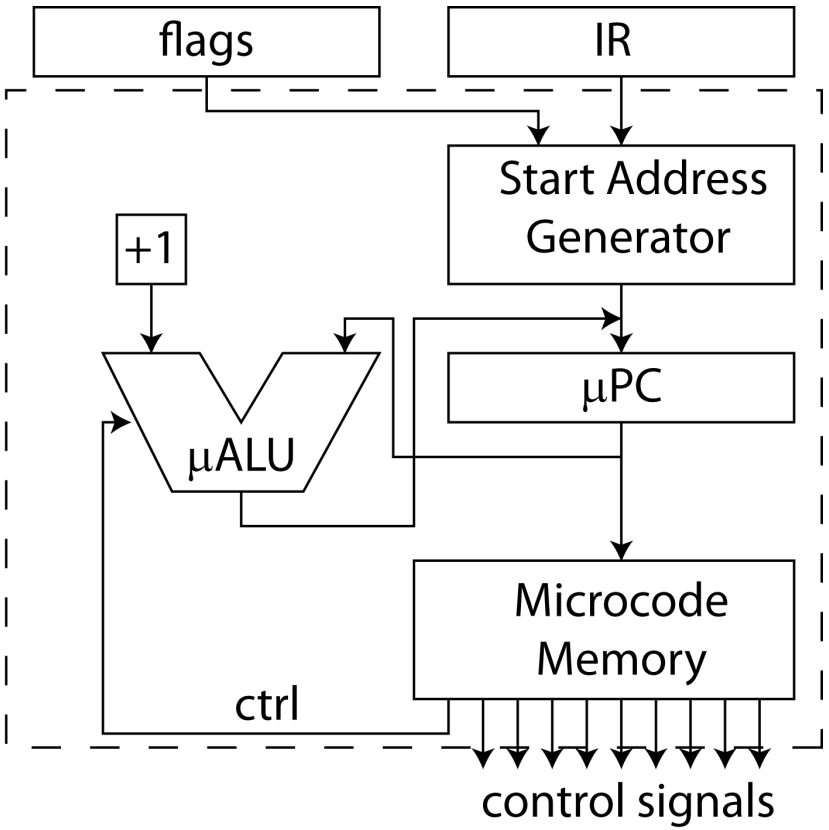
mem adr	content
0	move 4
1	add 5
2	store 6
3	stop
4	1
5	2
6	3
⋮	⋮

Microarchitecture

- So far we have considered a so called *hardwired* CU architecture, where a hardwired FSM issues the right sequence of control signals in response to a machine code in the IR.
- A more flexible alternative is to use *microcode* and a simple 'processor' within the processor that simply issues a sequence of control signals stored as *microinstructions* in the *microprogram memory*, typically a fast read only memory (ROM) but sometimes also a flash memory (i.e. electrically erasable programmable read only memory (EEPROM)).

Hardwired and Microprogrammed CU





Pros and Cons

	Microarchitecture	Hardwired
Occurrence	CISC	RISC
Flexibility	+	-
Design Cycle	+	-
Speed	-	+
Compactness	-	+
Power	-	+