Introduction to Digital Design

Introduction to Digital Design

- Why digital processing?
- Basic logical circuits, first conclusions
- Combinational circuits
  - Adder, multiplexer, decoder ...
- Sequential circuits
  - Circuits with memory
  - State machine, synchronous circuits
- General structure of a digital design
  - Top-down
  - Hardware/software

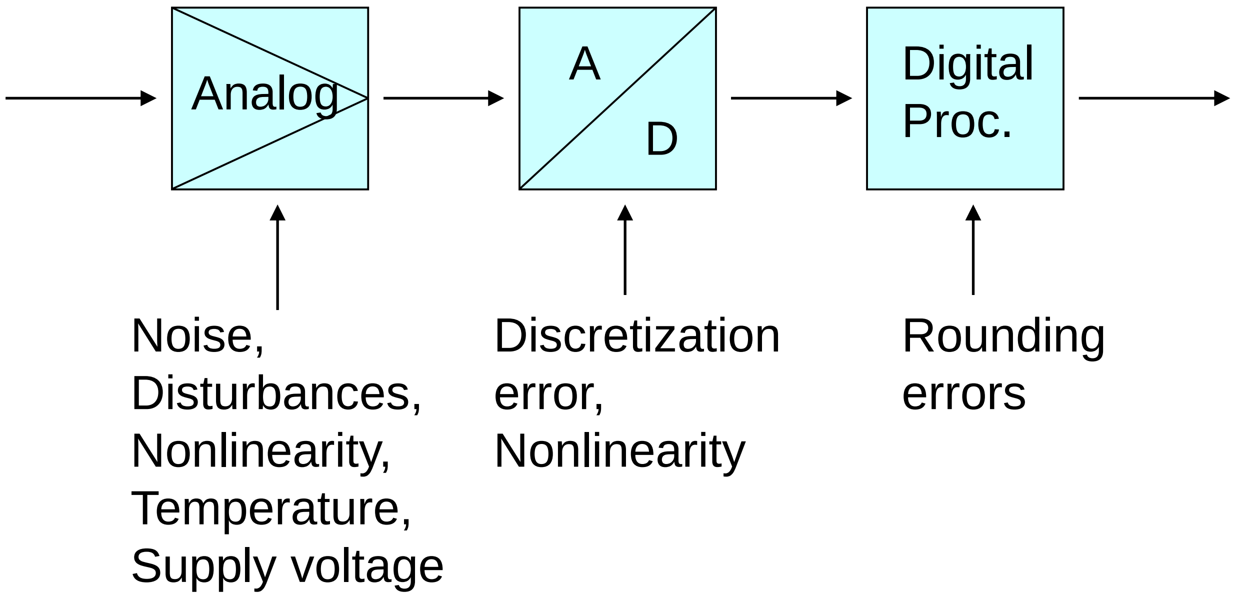
Why digital processing (1)?

- The world is to a good approximation analogue
- The result of some measurement can theoretically take continuous values, but we store it as a discrete value, multiple of some unit
- In most of the measurements additional corrections and processing of the primary information are necessary

$$\frac{d}{dt} \quad \sqrt{a^2+b^2} \quad \int dt \quad \sum \quad \prod$$

Why digital processing (2)?

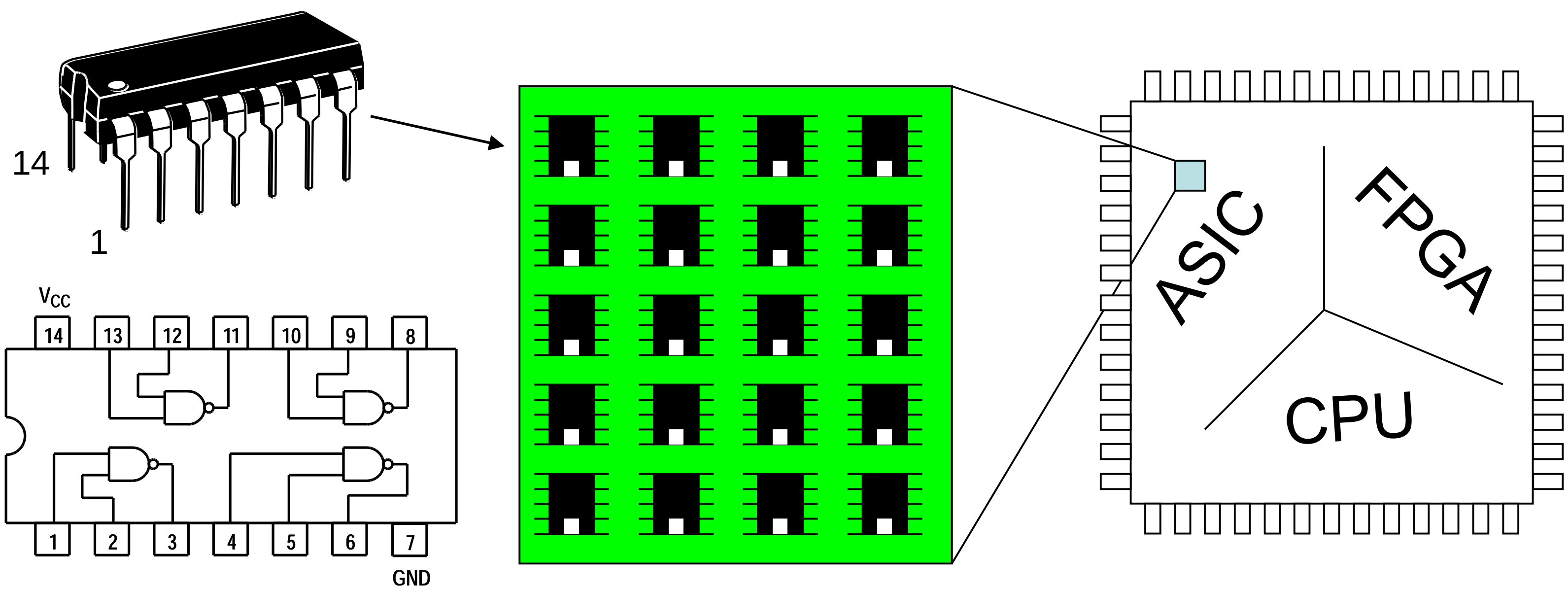
- Block diagram of some measurement device



- How to arrange the full processing in order to get the best results?

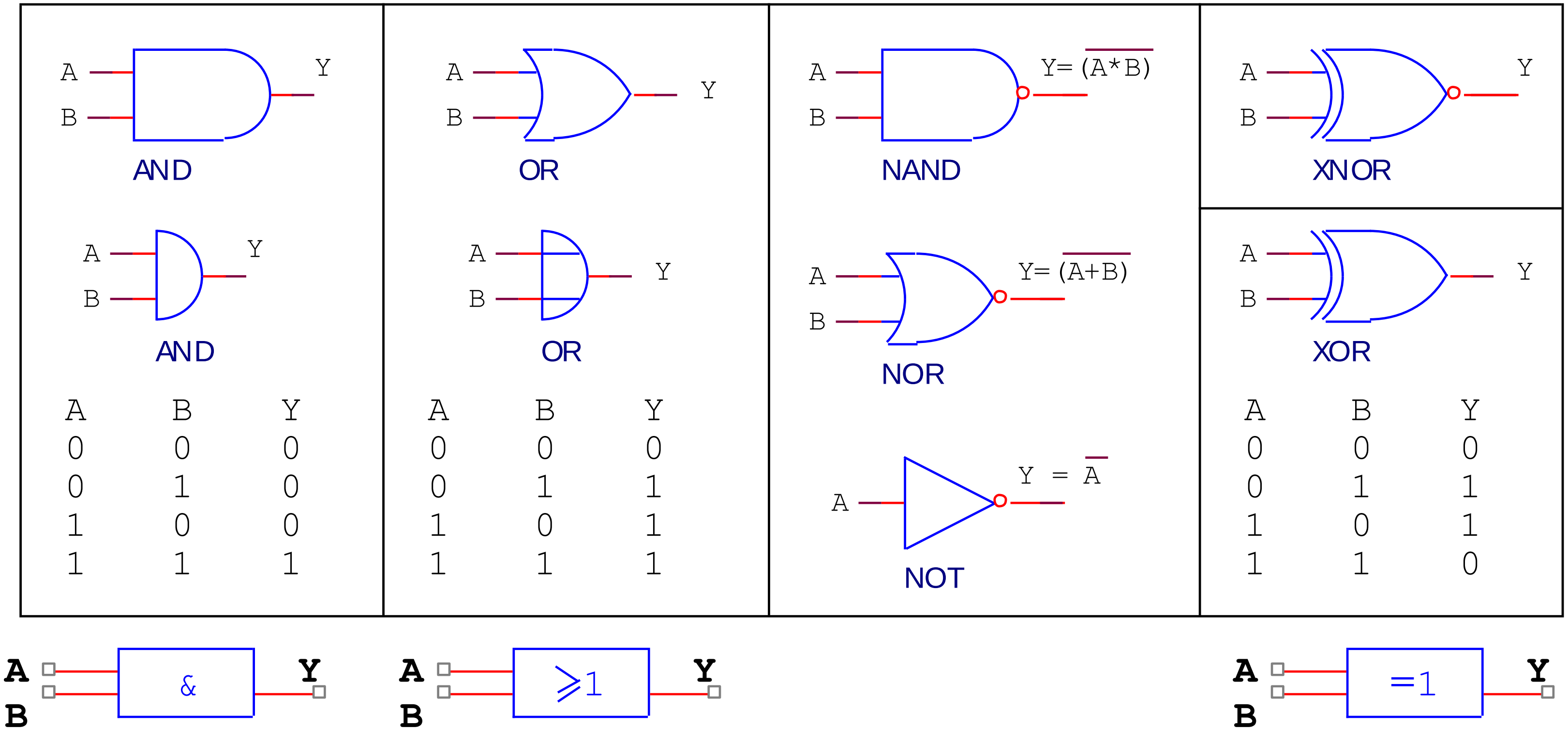
Why digital processing (3)?

- Where to do what? – the tendency is to start the digital processing as early as possible in the complete chain
- How ? This is our main subject now



Notations of the logic elements

- The most used are shown below
- Frequent use of non-standard symbols



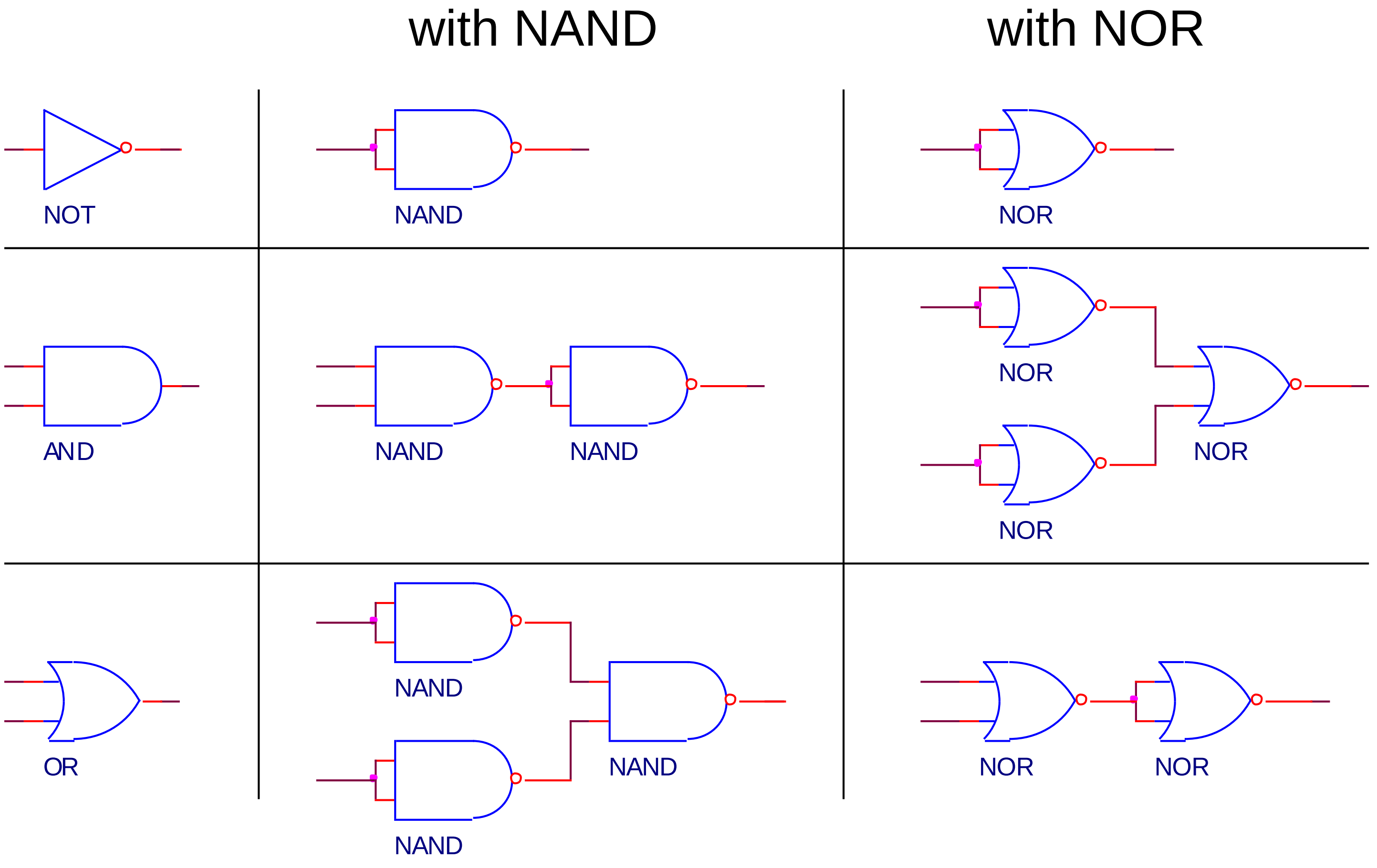
Useful Boolean relations

AND	·	*	∧
OR	+		∨
XOR	:+:		⊕
NOT	!		~

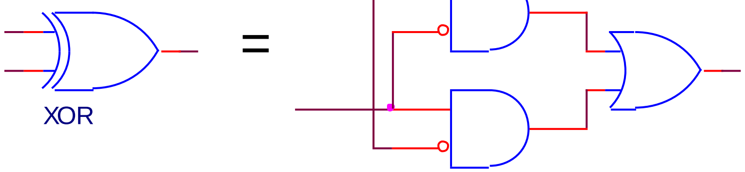
A+!A = 1	A*!A = 0	
A+ A = A	A* A = A	
A+ 0 = A	A* 0 = 0	
A+ 1 = 1	A* 1 = A	
!(!A)= A		
A+B = B+A	A*B = B*A	commutative
A+(B+C) = (A+B)+C	A*(B*C) = (A*B)*C	associative
A*(B+C) = A*B+A*C	A+(B*C) = (A+B)*(A+C)	distributive
A+(A*B) = A	A*(A+B) = A	absorptive
A+(!A*B) = A+B	A*(!A+B)= A*B	
!(A+B) = !A * !B	!(A*B) = !A + !B	de Morgan's

(A*B)+(!A*C) = (A+C)*(!A+B)

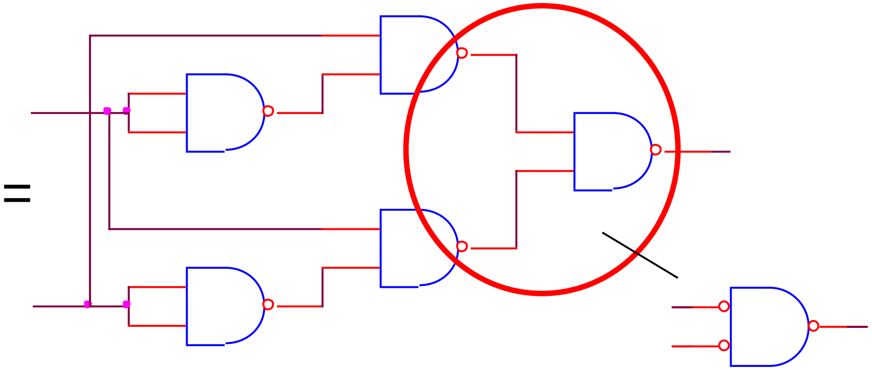
NAND or NOR can do everything…



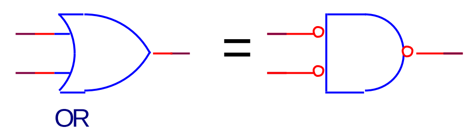
XOR with NAND or NOR



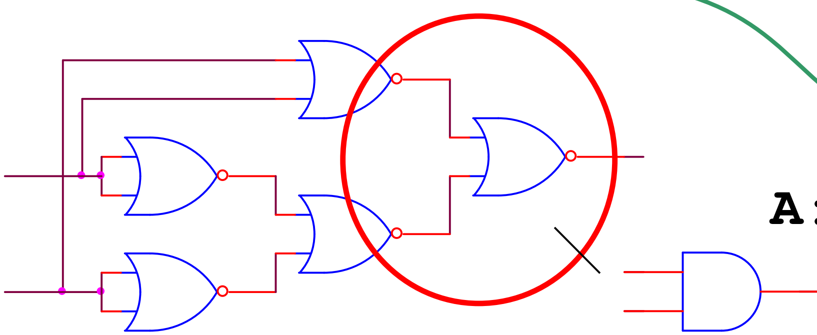
`A :+: B = A*!B + !A*B`



`A + B = !(!A*!B)`



(de Morgan)



`A:+:B = (A + B)*(!A + !B)=`
`(A + B)*!(A * B)`

(de Morgan)

Simplifying Boolean expressions

`F=A*!C + A*!B + !A*B*!C + !A*!B = !C*(A+!A*B) + !B = !C*(A+B)+!B =`

`= !C*A+!C*B+!B = A*!C + !B + !C = !B + !C`

`A+(A*B) = A`
`A+(!A*B)= A+B`

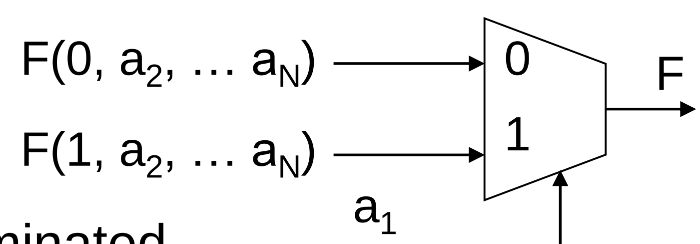
`F=A*!C + A*!B + !A*B*!C + !A*!B`

- For `A=0` ➔ `F=B*!C +!B = !B+!C`
- For `A=1` ➔ `F=!C+!B`

`F=A*!C + A*!B + !A*B*!C + !A*!B`

- For `B=0` ➔ `F=A*!C + A + !A = 1`
- For `B=1` ➔ `F=A*!C + !A*!C = !C`

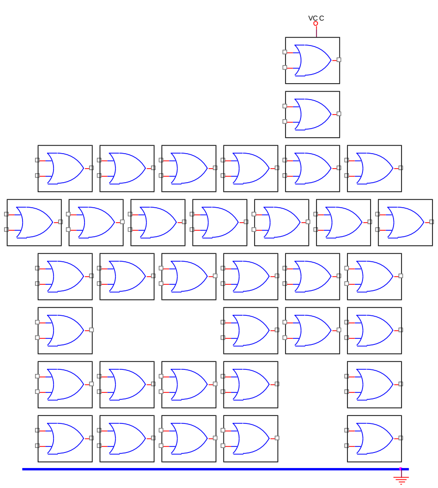
$$F(a_1, a_2, \dots a_N) =$$
$$= a_1*F(1, a_2, \dots a_N) + !a_1*F(0, a_2, \dots a_N)$$



With 4:1 mux, two variables can be eliminated

What kind of logical elements do we need?

- Exactly like a house, that can be built using many identical small bricks, one logical circuit can be built using many identical NOR (OR-NOT) or NAND (AND-NOT) elements
- For practical reasons it is much better to have a rich set of different logical elements, this will save area and power and will speed up the circuit

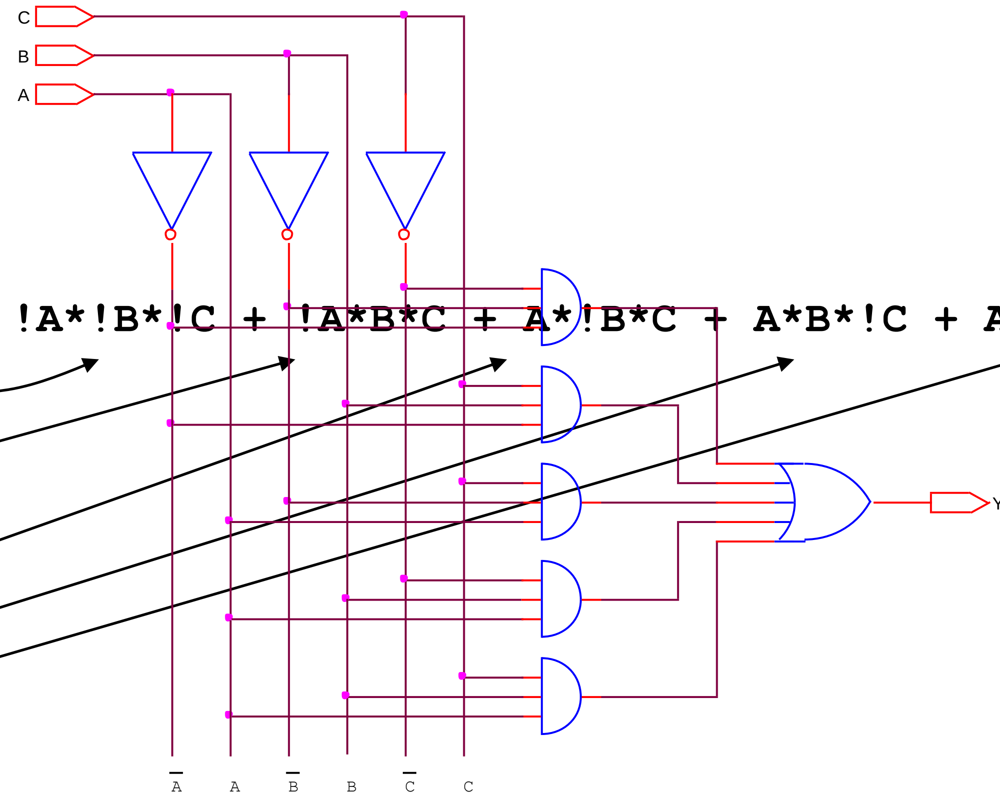


Sum of products representation

- Truth table

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Y = !A*!B*!C + !A*B*C + A*!B*C + A*B*!C + A*B*C



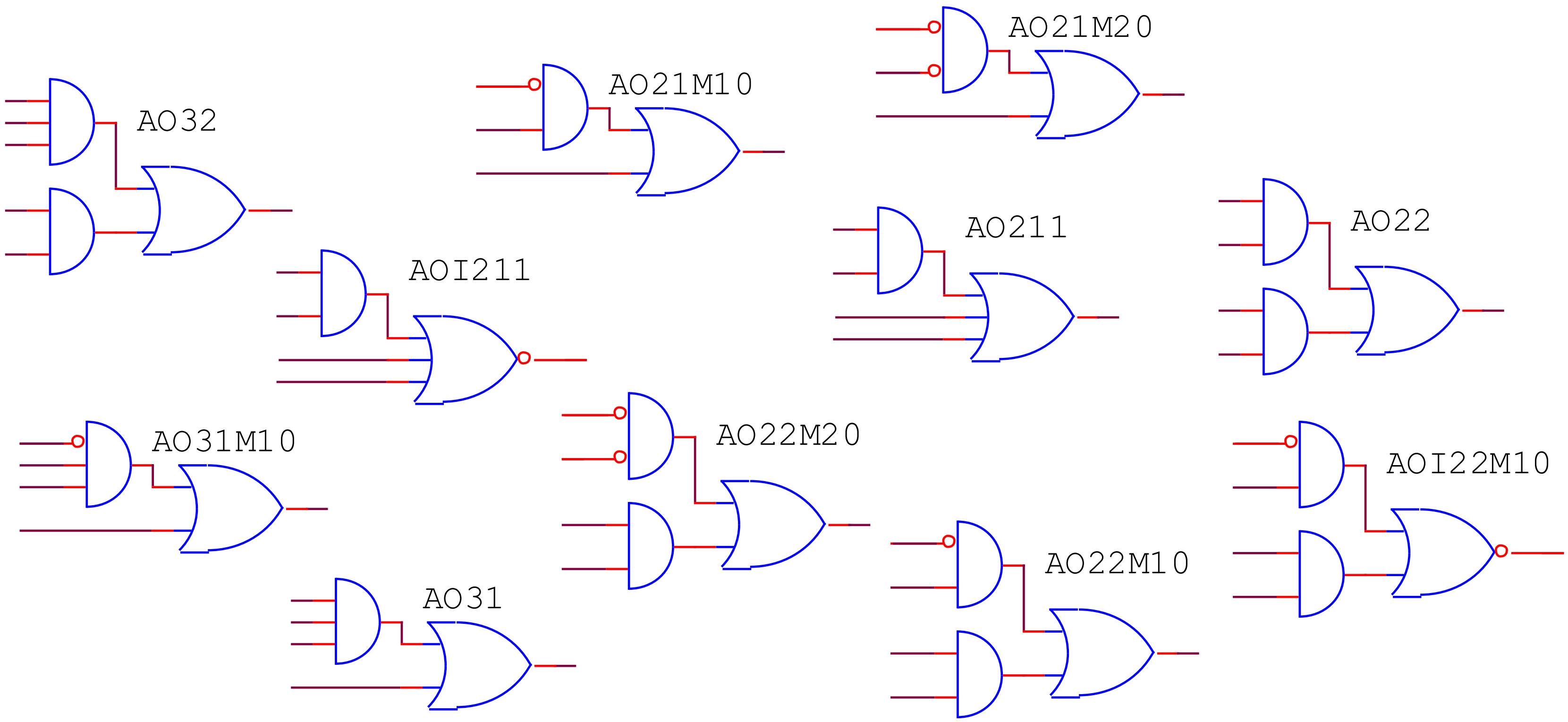
- If the function is more frequently 1, it is better to calculate the inverted function in order to have less terms: **Y = ! (!A*!B*C + !A*B*!C + A*!B*!C)**

Conclusions(1) – PAL/CPLD/HDL

- The sum of products representation was a good move! It seems to be a universal method (with some exceptions) to build any logical function – PAL and CPLD
- Drawing of the circuit is tedious and not very reliable!
- Writing of equations seems to be easier and more reliable → languages to describe hardware (HDL - hardware description language)

Conclusions(2) – ASIC

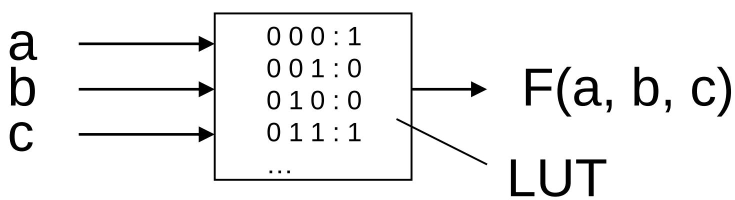
Another possibility is to have many different logic functions. Here are shown only a small subset of the variations with AND-OR-NOT primitive functions available in a typical ASIC library



All about 130 units + with different fanout capability

Conclusions(3) – LUT/FPGA

- Another possible architecture for logical functions is to implement the truth table directly as a ROM
- When increasing the number of the inputs **N**, the size of the memory grows very quickly as **2^N**!
- If we have reprogrammable small memory blocks (LUT - Look Up Table), we could easily realize any function – the only limit is the number of the input signals



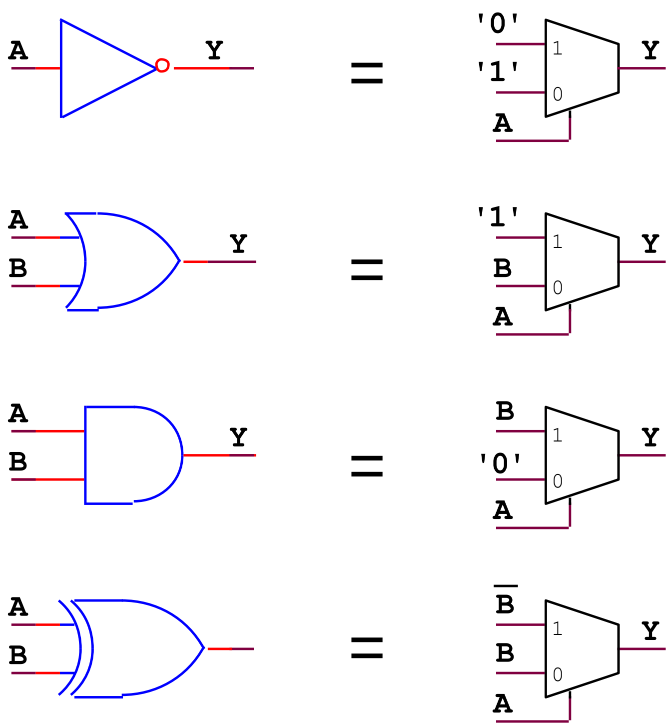
The FPGAs contain a lot of LUT with 4 to 6 inputs + something more

- For larger number of inputs we need to do something

Conclusions(4) – FPGA

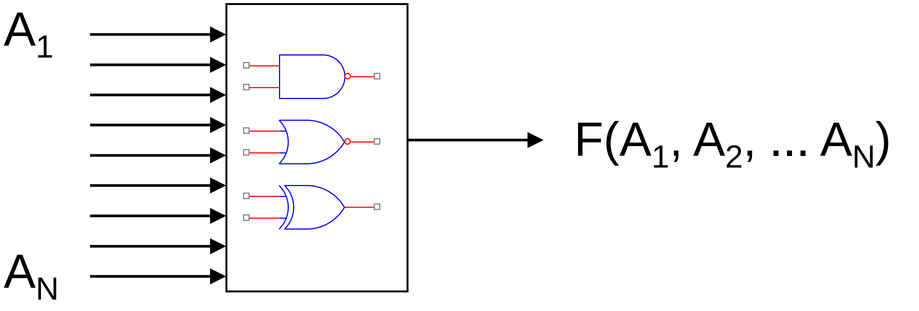
- Another possible architecture is to use multiplexers
- Examples of simple 2-input logical functions built with 2:1 multiplexer

This approach is used in some old FPGA architectures



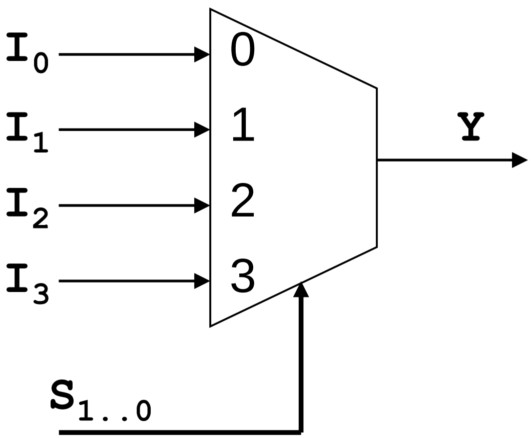
Combinational circuits

- ... are the circuits, where the outputs depend only on the present values of the inputs
- Practically there is always some **delay** in the reaction of the circuit, depending on the temperature, supply voltage, the particular input and the state of the other inputs
  - it is good to know the min and max values (worst/best case)

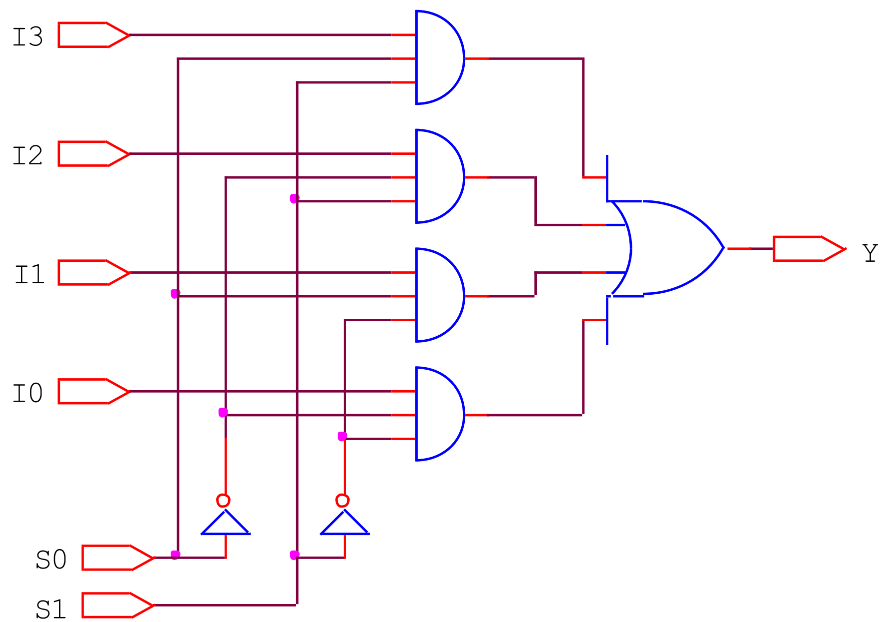


Special combinational circuits - multiplexer

- Used to control data streams – several data sources to a single receiver



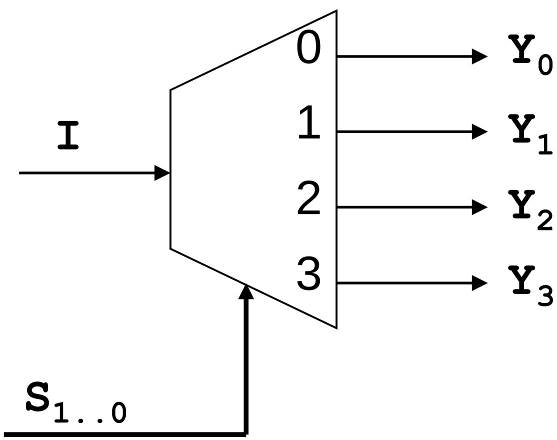
S	Y
0	I0
1	I1
2	I2
3	I3

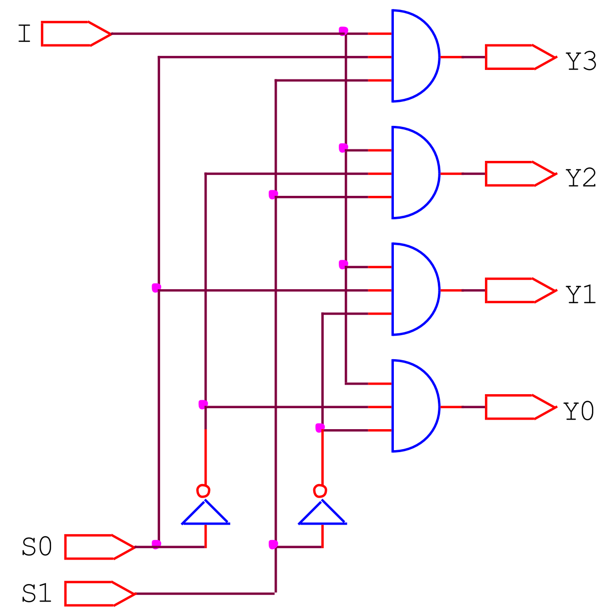



```
Y = !S[1]*!S[0]*I[0]+
    !S[1]* S[0]*I[1]+
     S[1]*!S[0]*I[2]+
     S[1]* S[0]*I[3]
```

Special combinational circuits – demultiplexer/decoder

- To some extend an opposite to the multiplexer





S	Y0	Y1	Y2	Y3
0	I	0	0	0
1	0	I	0	0
2	0	0	I	0
3	0	0	0	I

```
Y[0] = !S[1]*!S[0]*I
Y[1] = !S[1]* S[0]*I
Y[2] =  S[1]*!S[0]*I
Y[3] =  S[1]* S[0]*I
```

Special combinational circuits - adder

- Add/subtract – for more than some bits here it is not practical to use the sum-of-products approach (Why?)
- Binary system
  - Integer numbers ≥ 0 (unsigned)
  - Integer numbers – positive and negative (signed) - later
  - Adding of binary integer numbers, carry

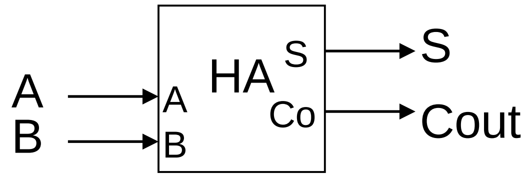
```
+  1011
   0110  ?
  10001
```

To calculate the most significant bit of the result we have to go through all the other bits, the carry jumps from bit to bit and this takes time!

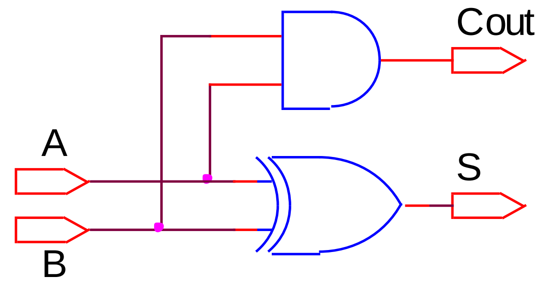
- Building blocks
  - Half- and Full- adder

Half- and Full- adder

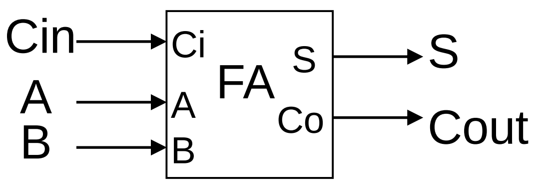
Half-adder



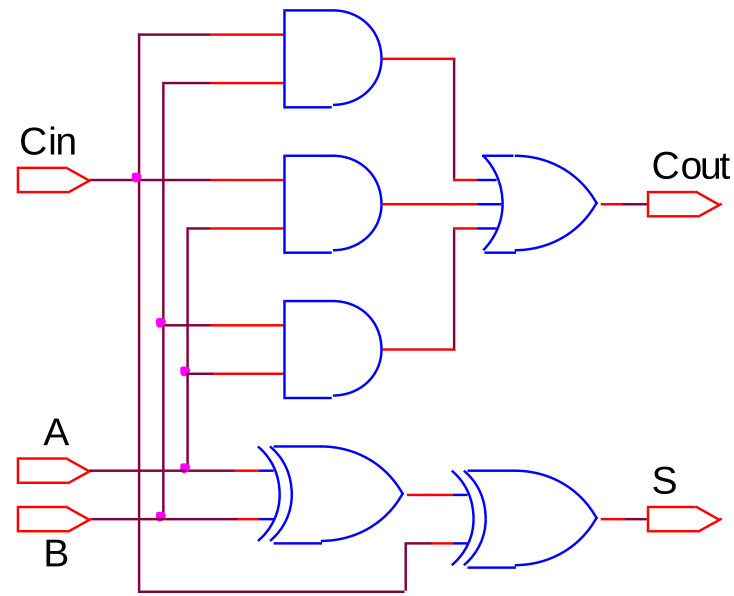
$A + B = 2 \cdot Cout + S$



Full-adder

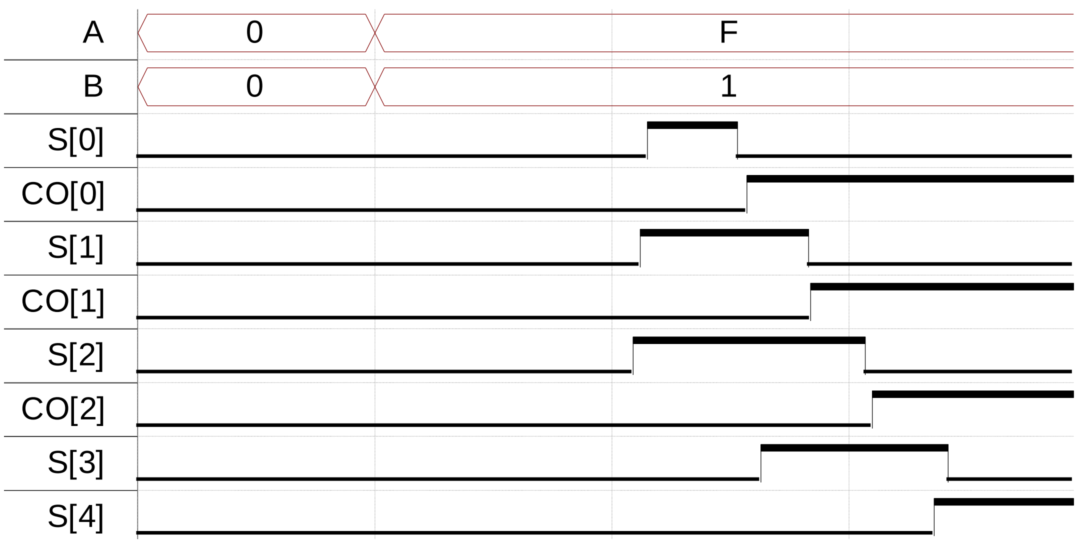


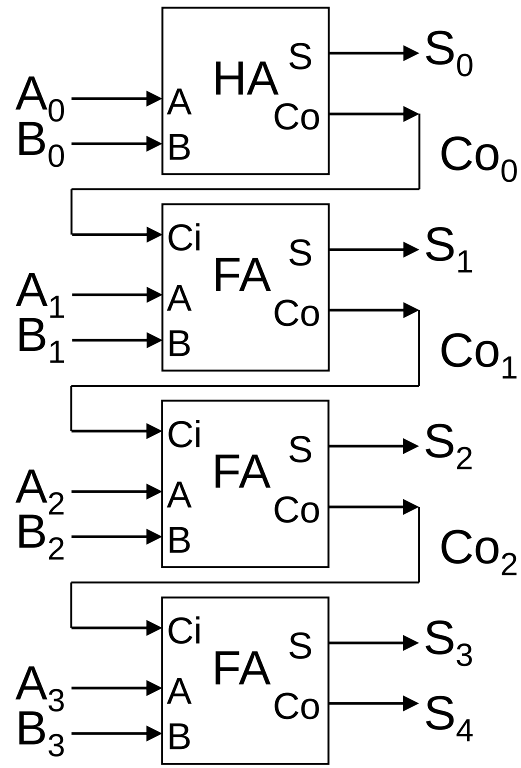
$A + B + Cin = 2 \cdot Cout + S$



4 bit ripple carry adder

```
  1111   A3..0
+ 0001   B3..0
 10000   S4..0
```





Ripple carry adder

Signed integers

- One's complement – invert all bits of **`A`** to get the negative of **`A`**
  - The **`0`** has two representations **`+0`** and **`-0`**.
  - Not practical for mathematical operations
- Two's complement – invert all bits and add 1
  - The sum of **`A`** and **`(not A +1)`** is 2^N but expressed with **`N`** bits is **`00..00`** => $-A=2^N-A$
  - All numbers from **`00..00`** to **`01..11`** are positive (**`0`** to $2^{N-1}-1$)
  - All numbers from **`11..11 (-1)`** to **`10..00`** (-2^{N-1}) are negative, the MSB is **`1`** when the number is negative
  - The full range is asymmetric, from -2^{N-1} to $+2^{N-1}-1$ (for 8 bits, from **`-128`** to **`+127`**). Note that the VHDL **`Integer`** is symmetric: from $-(2^{31}-1)$ to $+(2^{31}-1)$
  - Before doing mathematical operations with two signed numbers with different length, the shorter must be sign-extended to the length of the other

Two's complement – a closer look

- Let **A** be a positive integer:

$$A = \sum_{k=0}^{N-1} a_k 2^k,\ a_{N-1} = 0,\ a_k = 0 \text{ or } 1 \text{ for } k < N-1$$

- Then the negative of **A** is

$$-A \text{ represented as } 2^N - \sum_{k=0}^{N-1} a_k 2^k = \sum_{k=0}^{N-1} b_k 2^k,\ b_{N-1} = 1$$

- Subtracting $\mathbf{2^N}$ from both sides yields ($b_{N-1} = 1$):

$$-A = -\sum_{k=0}^{N-1} a_k 2^k = \sum_{k=0}^{N-1} b_k 2^k - 2^N = \sum_{k=0}^{N-2} b_k 2^k + b_{N-1} 2^{N-1} - 2^N = \sum_{k=0}^{N-2} b_k 2^k - b_{N-1} 2^{N-1}$$

- In two's complement the MSB has weight $\mathbf{-2^{N-1}}$ instead of $\mathbf{+2^{N-1}}$ – note that this is valid for both negative as for positive numbers!

Carry, borrow

- For unsigned integers, carry out = 1 means, that
  - when adding **`A+B`** the result is above 2^N-1

```
         1011  11                _0011  3
       + 0110   6                 0110  6
carry → 10001   1      borrow → 11101 13
```

  - when subtracting **`A-B`**, **`B`** is larger than **`A`**, in this case we speak of *borrow*

Carry, borrow, overflow

- For signed integers, carry output = 1 is not necessary bad

```
        1011  -5                       0011  3
      +                              –
        0110   6        correct        0110  6
carry→ 10001   1               borrow→11101 -3
```

- but

```
        1011  -5                       0011  3
      +                              –
        1010  -6        wrong          1010 -6
carry→ 10101   5               borrow→11001  7
```

Overflow = carry out XOR carry between the last two bits

Overflow

- For signed integers, overflow can be detected by the wrong sign of the result:

	Correct	Wrong
pos + pos →	**pos**	**neg**
neg + neg →	**neg**	**pos**
pos – neg →	**pos**	**neg**
neg – pos →	**neg**	**pos**

This is overflow!

pos + neg
pos – pos
neg – neg

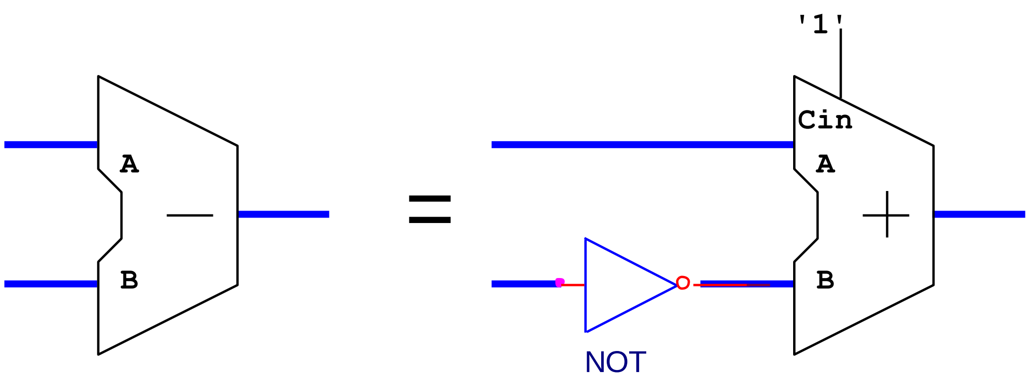
Always correct

The MSB is 1 for the negative numbers and 0 for the positive (incl. 0), so one can detect the overflow only using the MSBs of the two operands and of the result

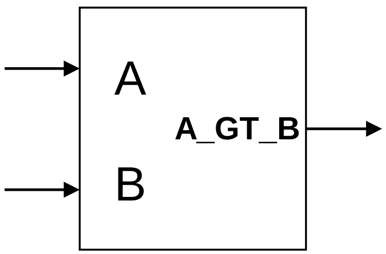
Subtracting using an adder

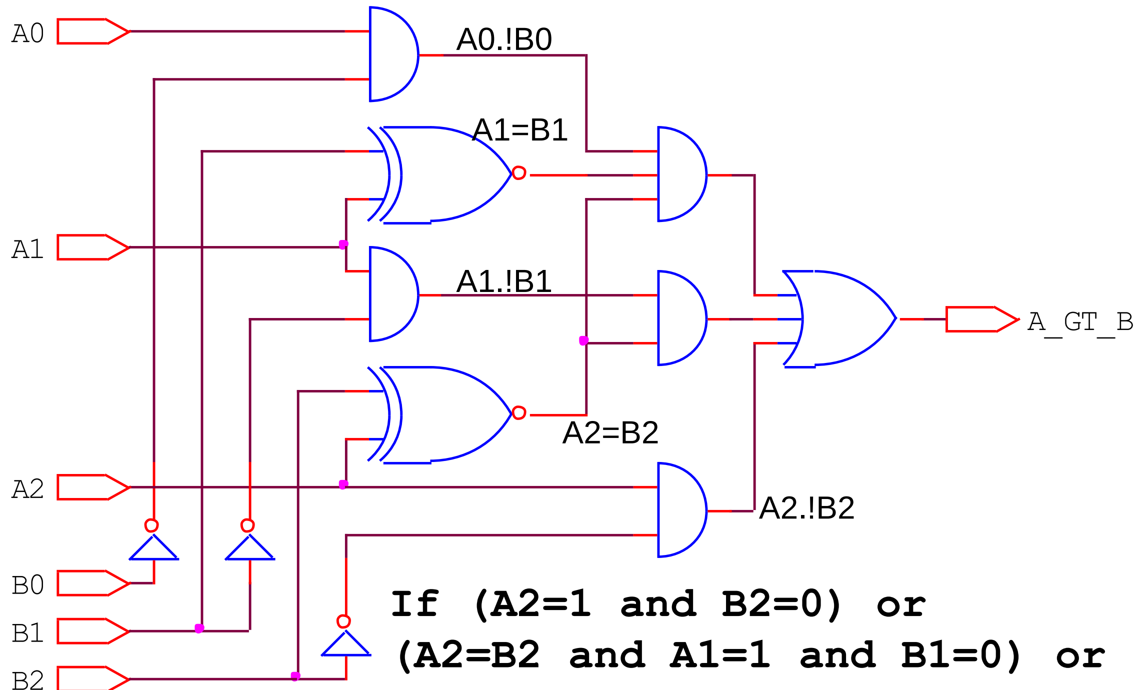
- Add/subtract with two’s complement numbers can be done exactly like with unsigned integers
- For N-bit signed:

$$A-B=A+(2^N-B)=A+two's\_complement(B)$$



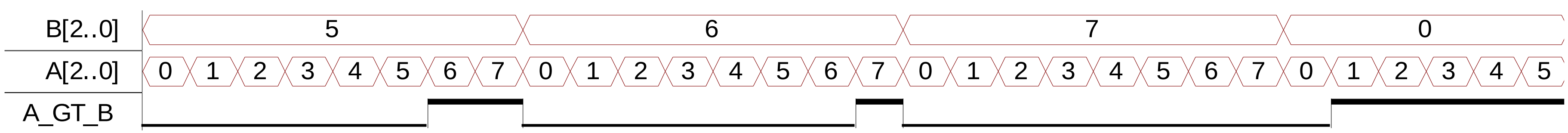
Comparator





Other possibilities:
>=, <, <=,
signed/unsigned

```
If (A2=1 and B2=0) or
(A2=B2 and A1=1 and B1=0) or
(A2=B2 and A1=B1 and A0=1 and B0=0)
...
```



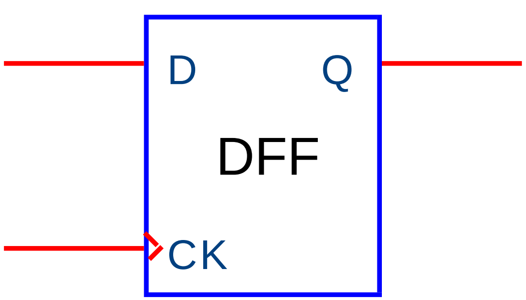
Combinational ↔ sequential circuits

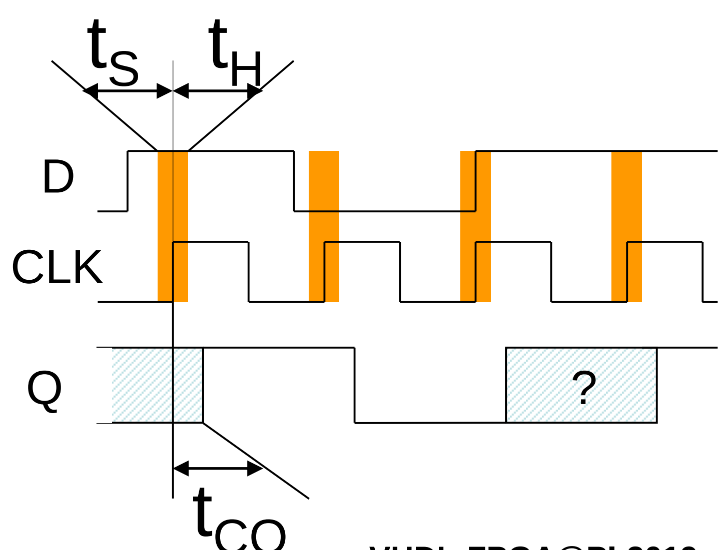
- Theoretically we can make a combinational circuit which gets all input data and solves the complete problem after some delay
- This approach is hardly usable, even when the problem has an analytical solution
- The data come in most of the cases sequentially in time, the algorithms have branches

A typical digital design consists of several blocks of combinational circuits and circuits with memory, the processing is done in small portions in equal steps in time

Circuits with memory (DFF)

- Behaviour: the D input is sampled at the rising (or falling) edge of the clock (CLK, CK) and appears at Q
- Timing:
  - D must be stable t_S (setup) before and t_H (hold) after the active edge of the clock signal CK
  - The output Q settles within some time t_{CO}, if the conditions are violated (t_S , t_H) the state of the flip-flop is unknown, oscillations are possible





Circuits with memory (Latch)

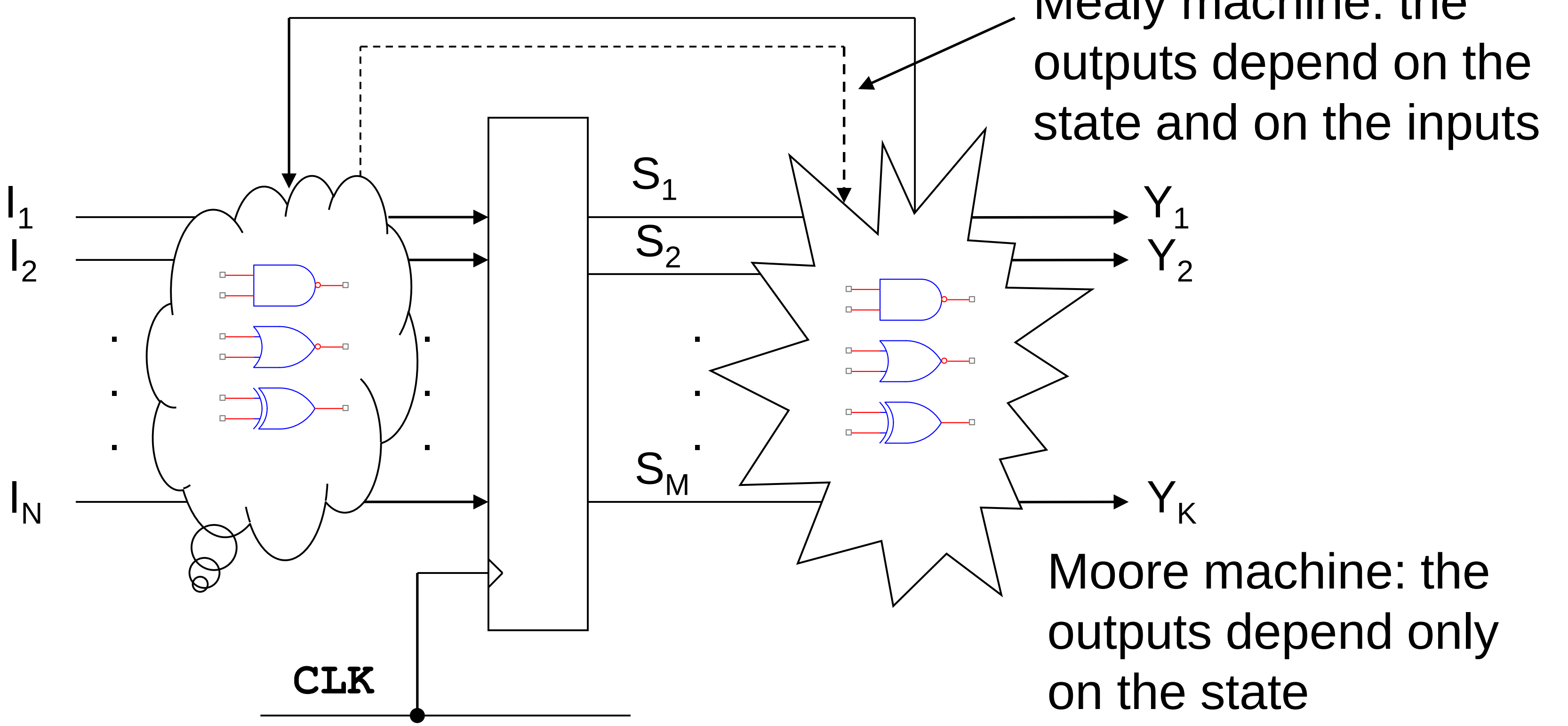
- Behaviour: the D input is sampled continuously while C is high (or low) and appears at Q. The D input doesn’t influence the output when C is low!
- Be aware of the difference to the DFF, do not use latches, except when you really know what you are doing!
- Latches can appear in the design after errors in the HDL description of the circuit, the report files usually contain warnings in such cases (but nobody reads it!)
- Latches mixed with DFF make the timing analysis of the design very difficult
- The asynchronous memories are based internally on latches, but the modern memories have synchronous interface

State of a sequential circuit

According to H. Hellermann (Digital Computer System Principles):

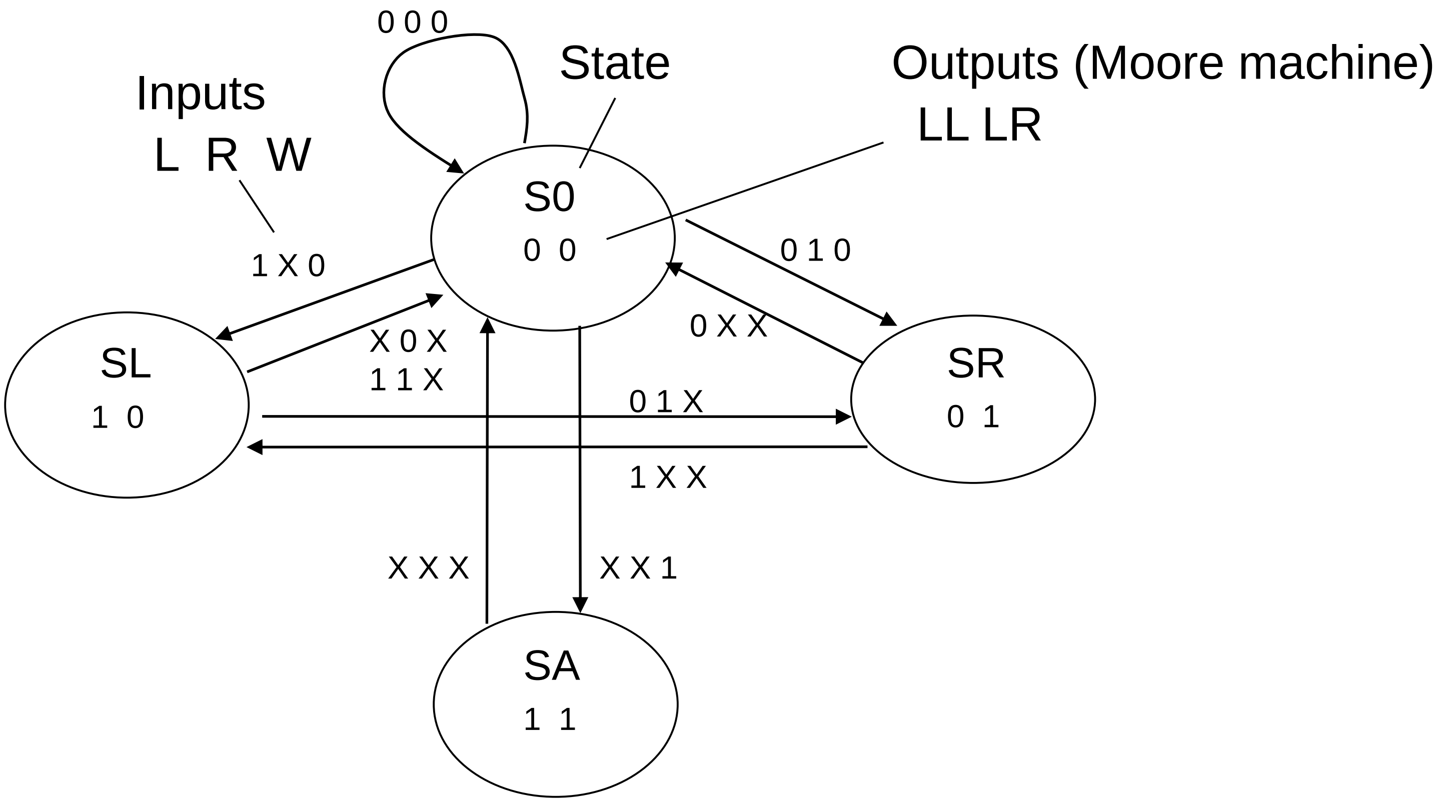
“The *state* of a sequential circuit is a collection of *state variables* whose values at any one time contain all the information about the past necessary to account for the circuit’s future behaviour”

State machines



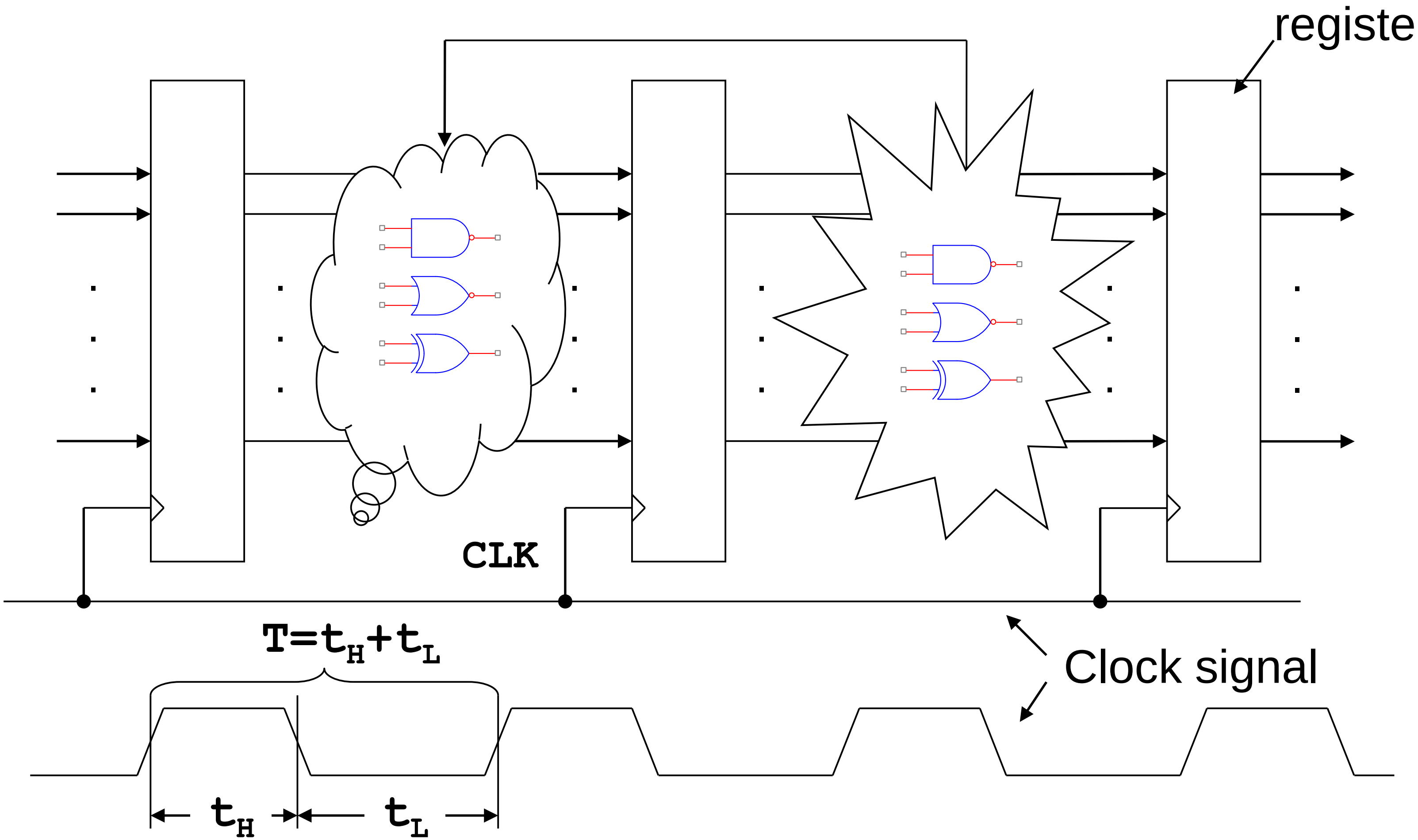
The next state $S[1..M]_{i+1}$ is a function of the present state $S[1..M]_i$ and of the inputs I[1..N]. The outputs Y[1..K] are function of the present state $S[1..M]_i$, but could depend on the inputs I[1..N]

State machine example



The description of a state machine is often done by state diagrams. Here are shown all the states, the transitions with their conditions and the outputs. For convenience the condition to stay in the same state can be omitted. The conditions to exit any state should be never in conflict!

Synchronous circuits

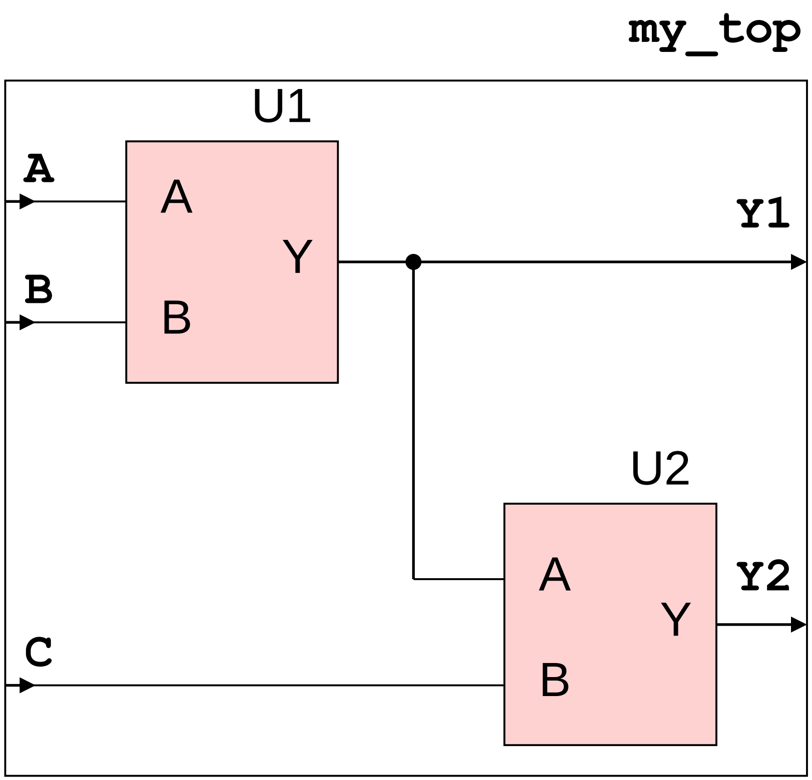


At each rising clock edge the registers store the current values at their inputs. Timing: setup/hold times (t_S , t_H), clock to output delay t_{CO}

Register Transfer Level (RTL)

- A digital synchronous circuit consists of registers and combinational logic between them
- The description of such a circuit actually specifies what happens after each clock cycle – the data transfer between the registers – RTL

Structural approach: top-down



Iterative process !

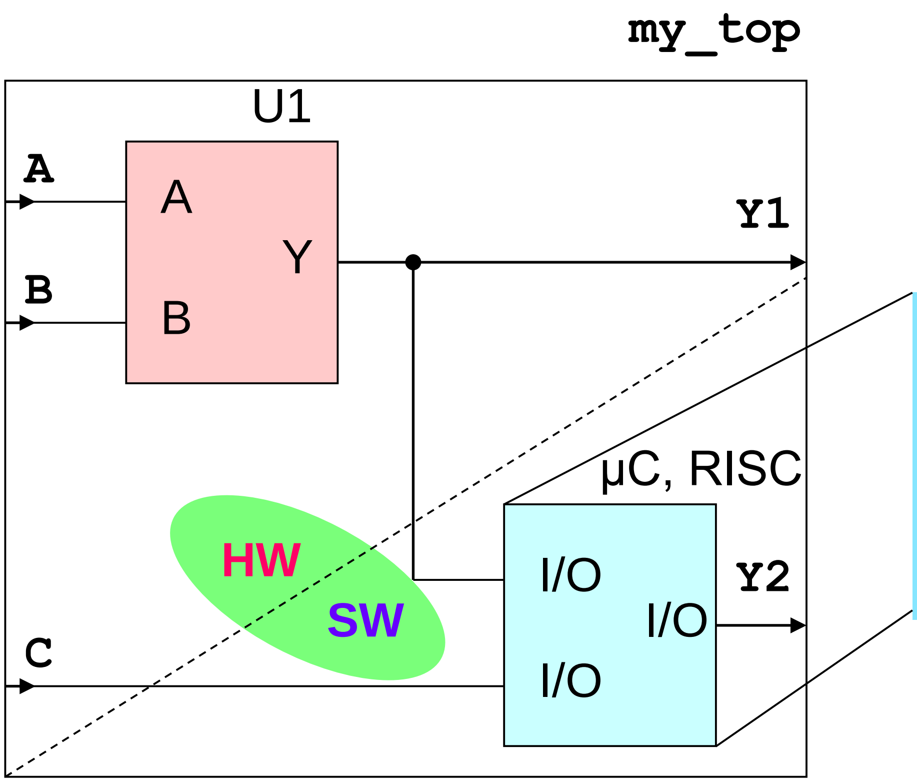
- Don't delay the documentation, it is part of each design phase

- Try to understand the problem, do not stop at the first most obvious solution
- Divide into subdesigns (3..8), with possibly less connections between them, prepare block diagrams before starting with the implementation
- Clearly define the function of each block and the interface between the blocks, independently on the implementation(s) of each block
- Develop the blocks (in team) and then check the functionality
- Combine all blocks into the top module, if any of them is not finished, put temporarily a dummy

Structural approach: top-down

- Think about compatibility, extensibility and scalability of the design
- Try to do the functionality of the module symmetrical and include all simple and reasonable extensions
- Maximize orthogonality, do not implement functions, just because they are "nice", but are combinations of already implemented functions (example: many ways to clear or increment some CPU register). An architecture with high orthogonality tends to provide more function at the same level of complexity and cost
- The hardware should be not damageable by the user, think about auto-consistency of the configuration and about protections
- Do not spread the important constants like dimensions, addresses etc. in the several sources of the design, put them into one central place
- Try to be technologically independent as long as possible
- Make the configuration registers read/write instead of write only
- Think about testing and debugging

Hardware : software?



• Divide in two parts - hardware : software, taking into account the desired speed, size, flexibility, power consumption and other conditions

```
again: inc r5
        load r2, [r5]
       and r2, 0xAB
       bra cc_zero, again
        store [r3], r6
       ...
```

• select the processor core
• for the structure of the hardware part proceed as described before

Just a few questions more:

Questions, questions...

- How to partition the design? Where to put the boundary between software and hardware?
- How to enter the design?
- How to check whether each subblock works as expected, according to the description?
- How to select the possible implementation in a silicon chip?
- How to check whether the chip will work so as we want before ordering it?
- How to check the chip functionality when we get it back?
- How to test the chips in the production (and the boards after assembly)?

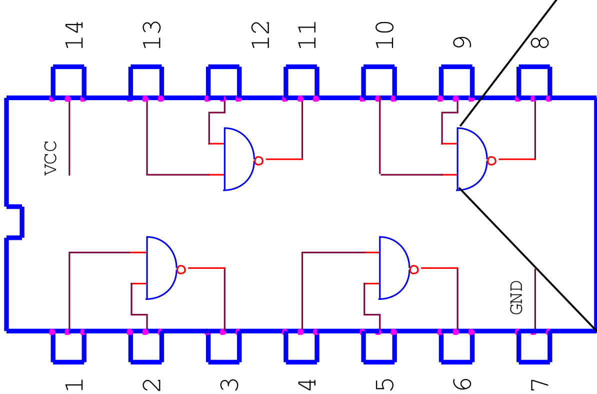
Technologies

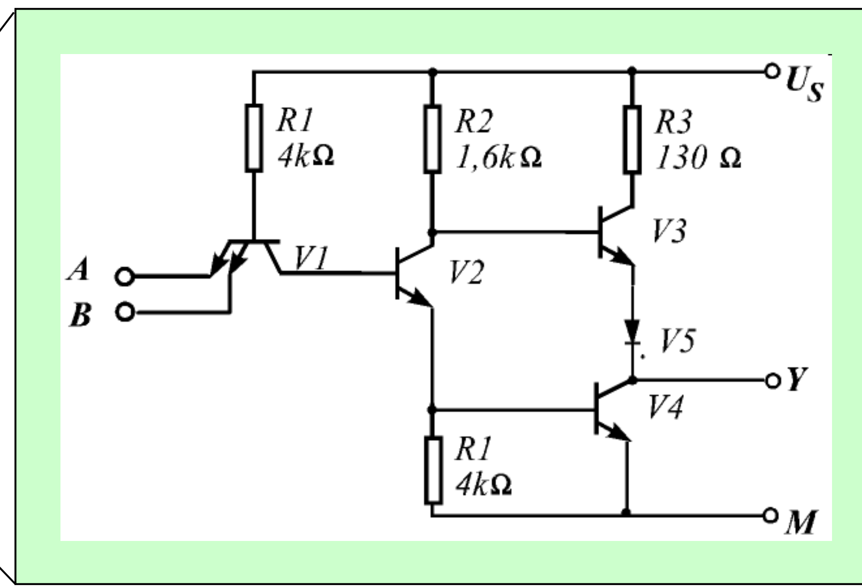
Technologies

- Small Scale Integration (SSI) ICs (74xx, 4000)
- Simple Programmable Logic Devices (SPLD) - PAL (Programmable Array Logic) & GAL (Generic Array Logic), Complex Programmable Logic Devices (CPLD)
  - Architecture, manufacturers, overview of the available products
- Field Programmable Gate Arrays (FPGA)
  - Architecture, manufacturers, overview of the available products
- Design flow FPGA/CPLD
- Application Specific Integrated Circuit (ASIC)
  - Standard cell (structured ASIC)
  - Others (gate array, full-custom)
  - Design flow

TTL (transistor-transistor logic)

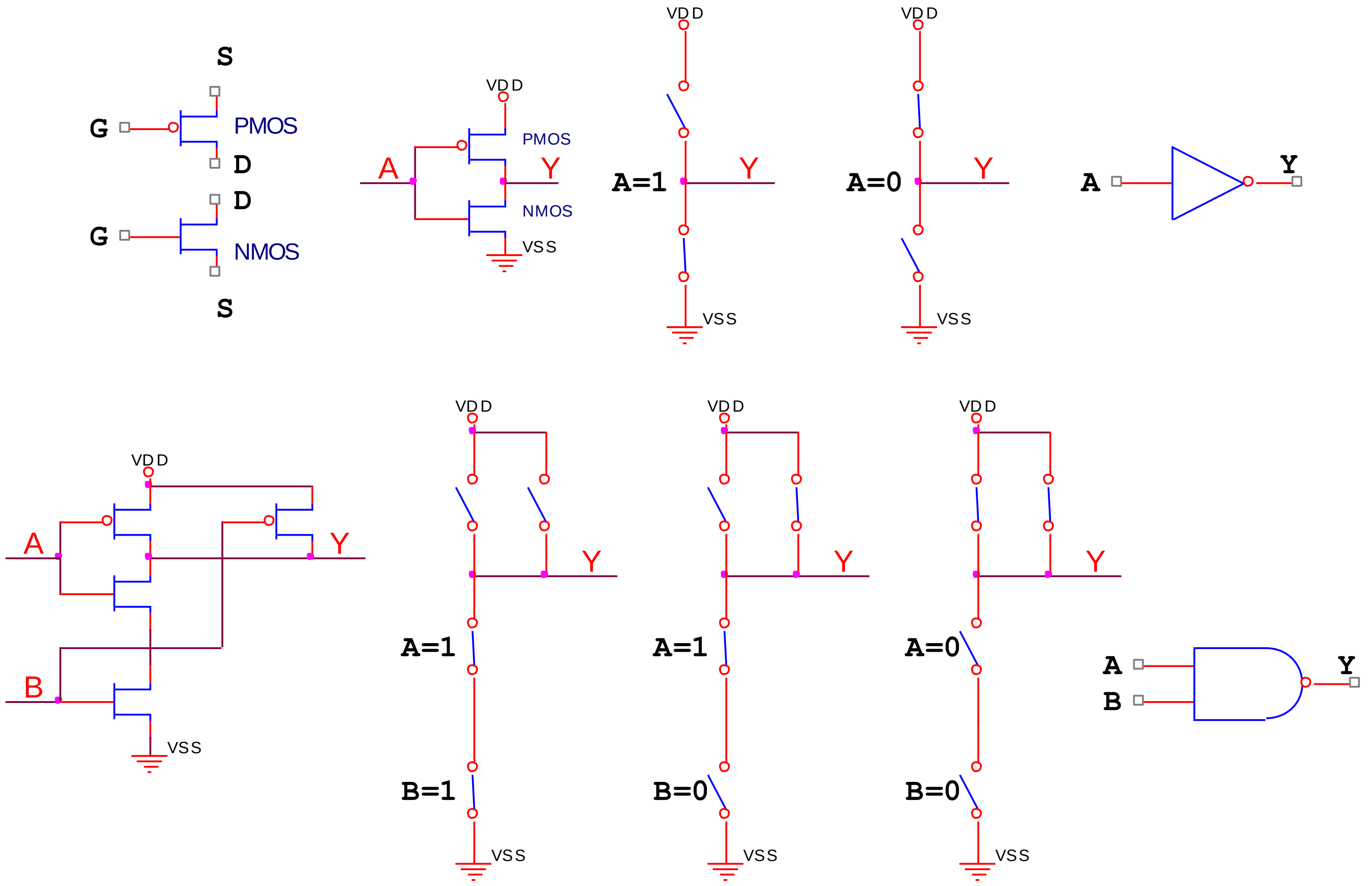
- 7400 - 4 x (4 bipolar transistors + 4 resistors)
- 74xx – many combinations of different logical elements (AND, OR, NOT), flip-flops, counters and many others.
- From the modern point of view – slow, hungry (for electrical power) monster
- Small Scale Integration IC (SSI)



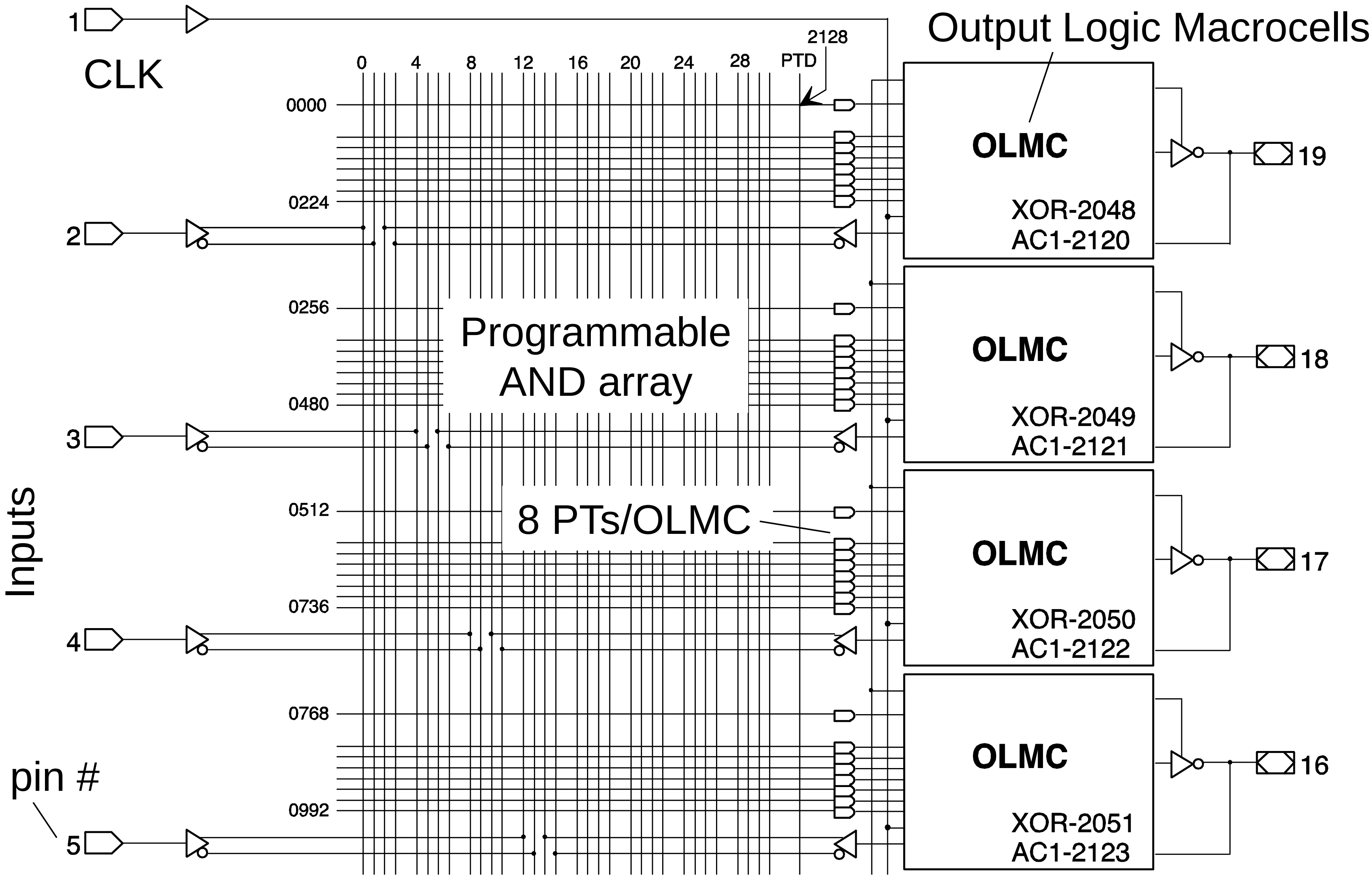


CMOS technology

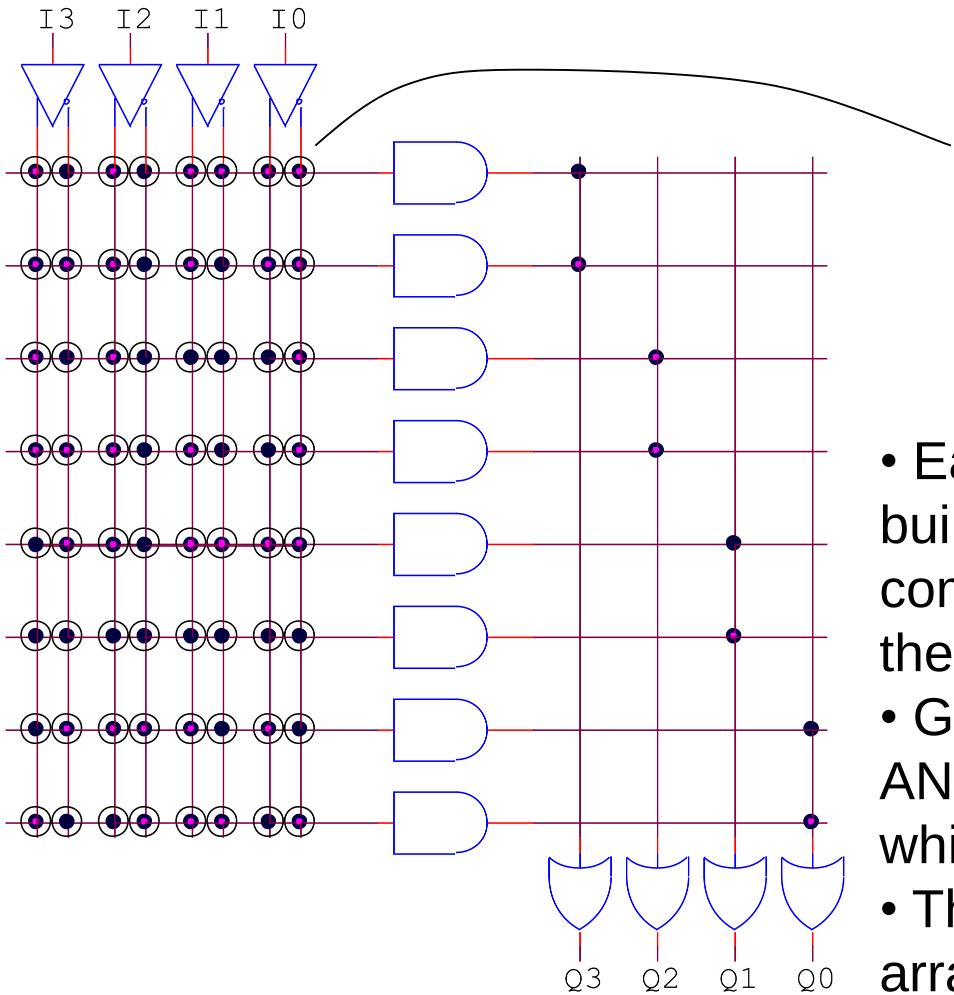
- Built with nMOS and pMOS transistors

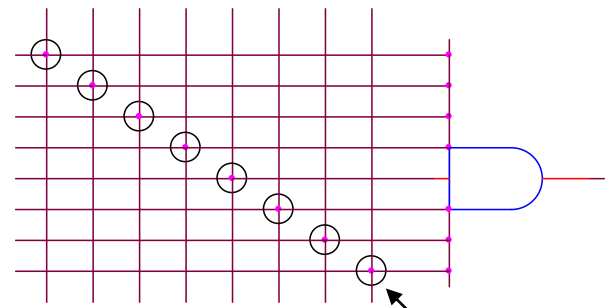


Simple PLD – GAL (generic array logic)



SPLD - the AND array



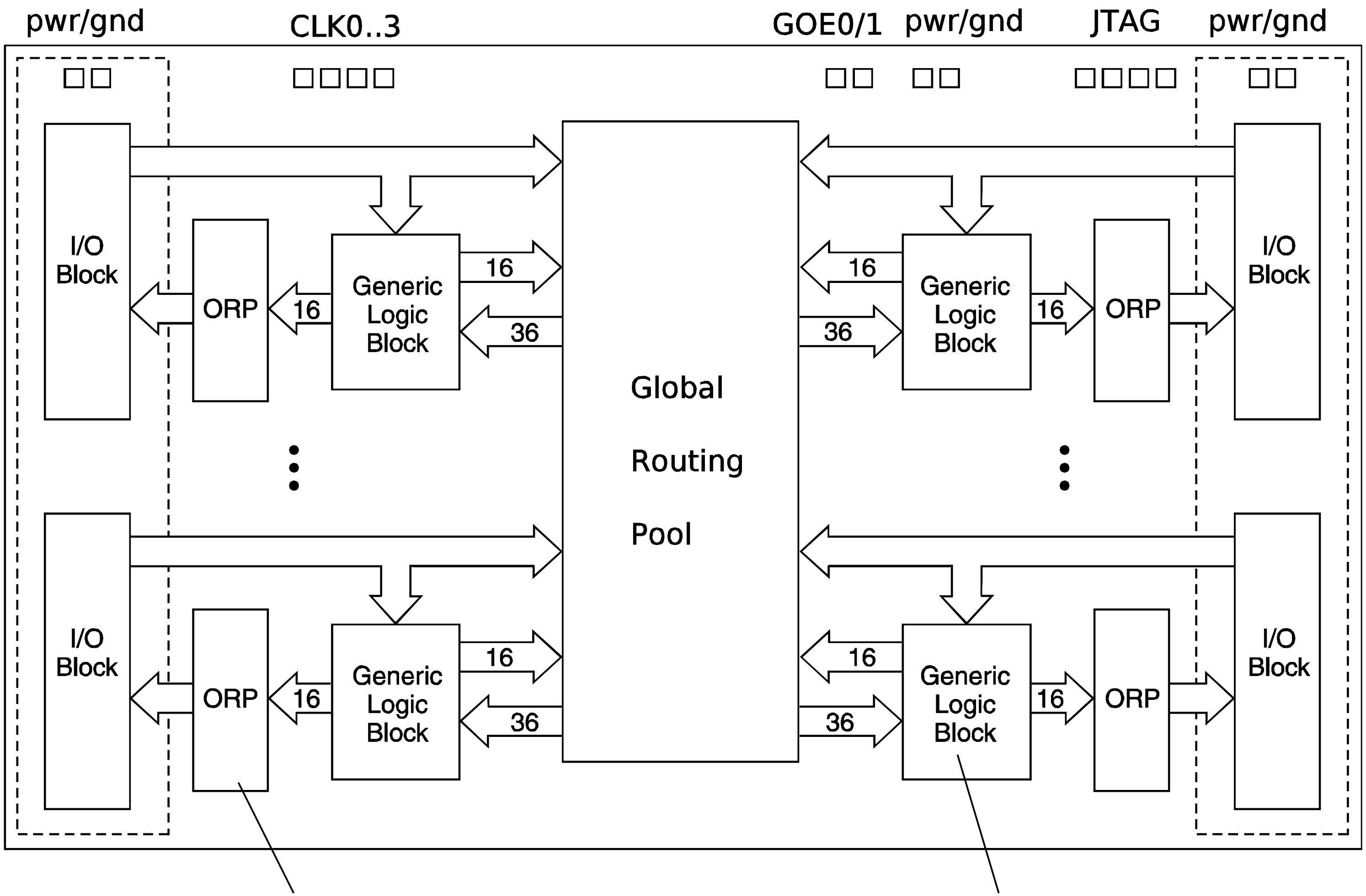


- Each AND has enough inputs to build the product of any combination of the input signals or their negations
- Group of several (typically 8) ANDs are **hardwired** to a OR, which is routed to an output (PAL)
- The PLAs have programmable OR array but were never widely used

PAL/GAL – summary

- The first widely used programmable logic devices
- Used in the past to replace several small scale integration ICs, like 74xx
- Very successfully used for small state machines
- Manufactured first by MMI (Monolitic Memories Inc.), later by AMD, Lattice and others
- The first devices were one time programmable (OTP) and with either combinational or registered macrocells (or a fixed mixture), the later were electrically erasable/programmable (up to 100 times) with freely programmable type of the macrocells
- Software tools – based on <u>H</u>ardware <u>D</u>escription <u>L</u>anguages (HDL) – ABEL, CUPL, PALASM or schematics
- The next generation of PLD – Complex PLD (CPLD) are based on the same architecture

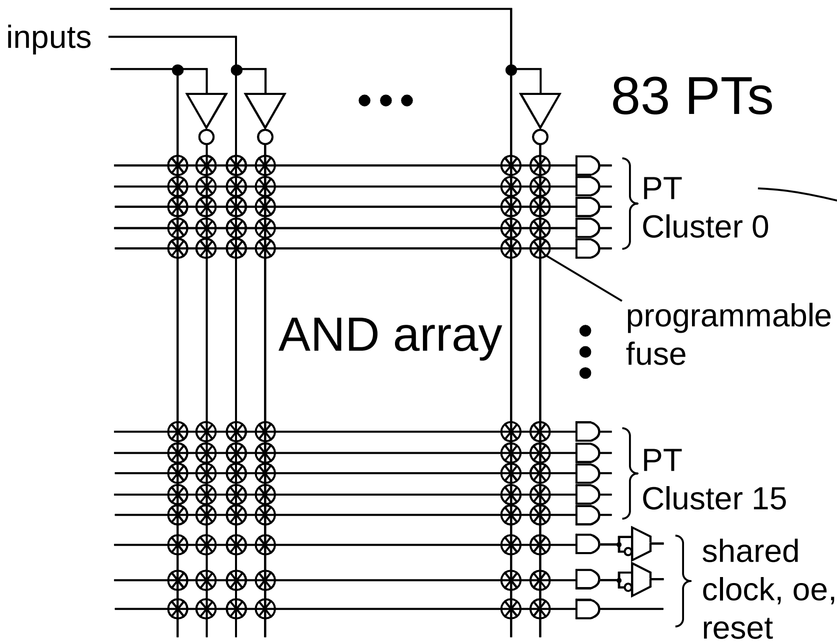
CPLD – ispMACH 4000 (Lattice)

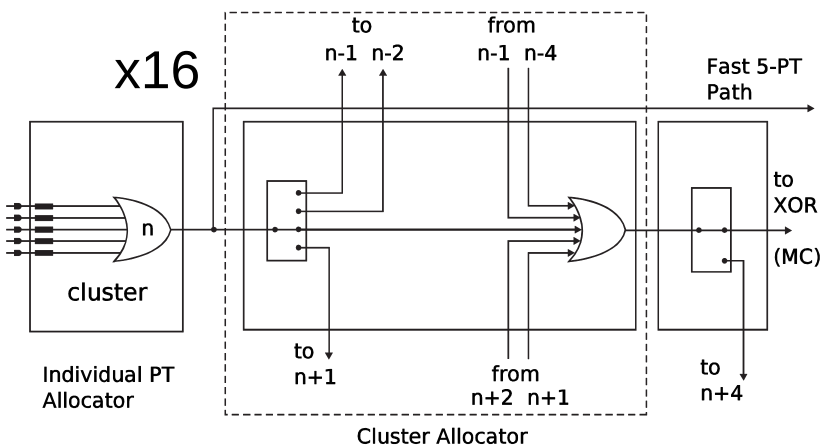


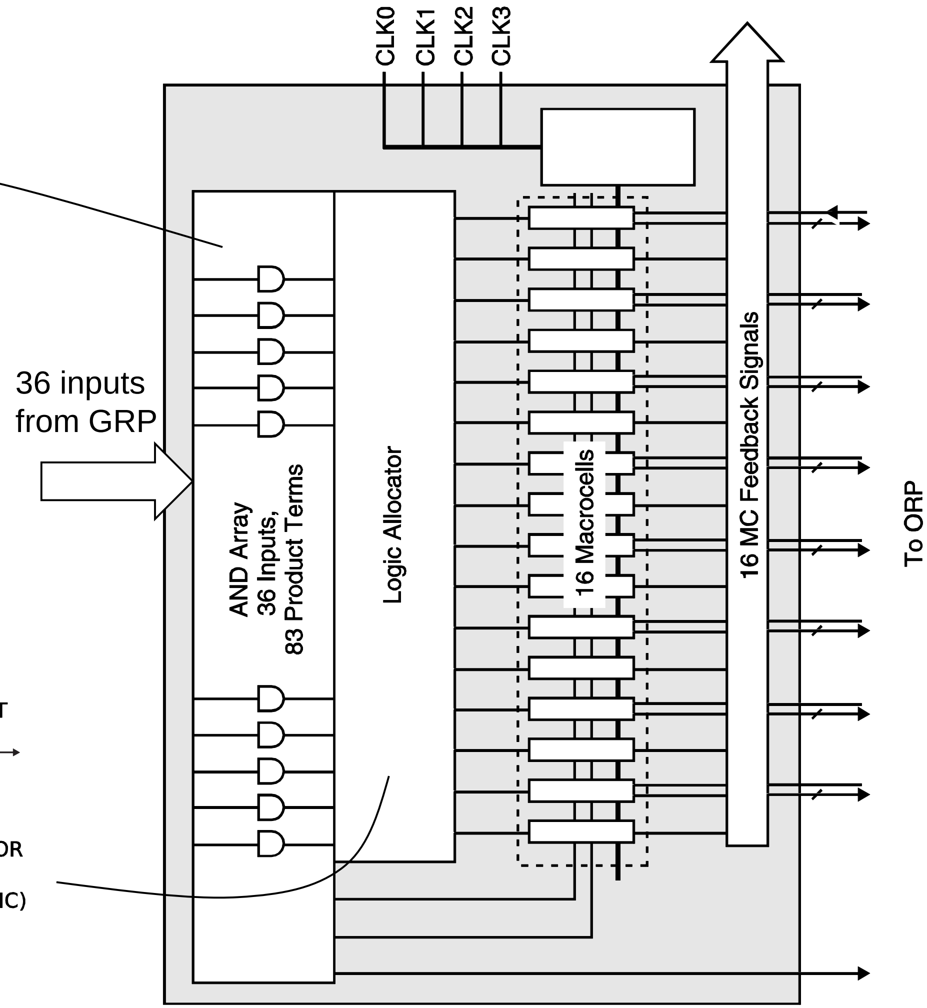
Output Routing Pool

Similar to a GAL

CPLD – ispMACH 4000 - GLB





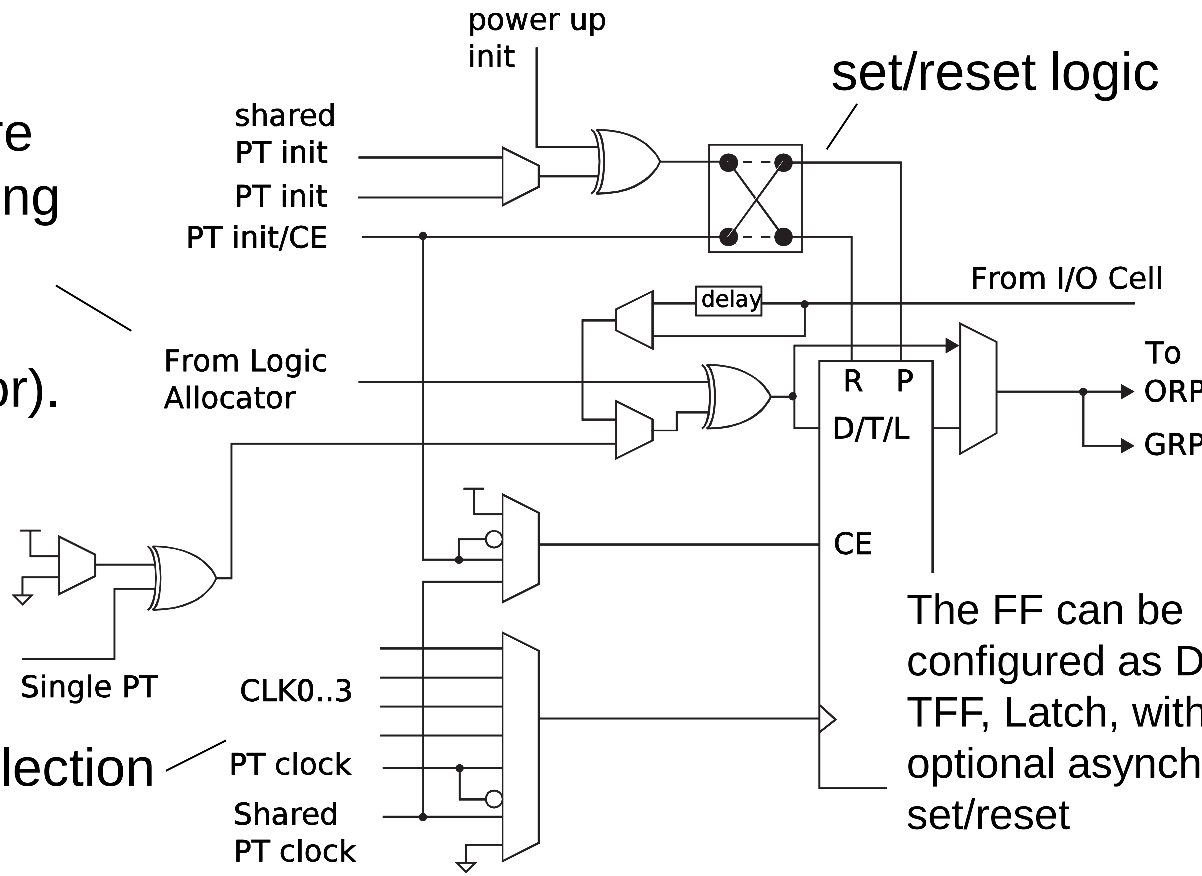


Generic Logic Block

CPLD – ispMACH 4000 Macrocell

Up to 80 PTs are possible, by using the PTs of the neighbour cells (cluster allocator).

set/reset logic

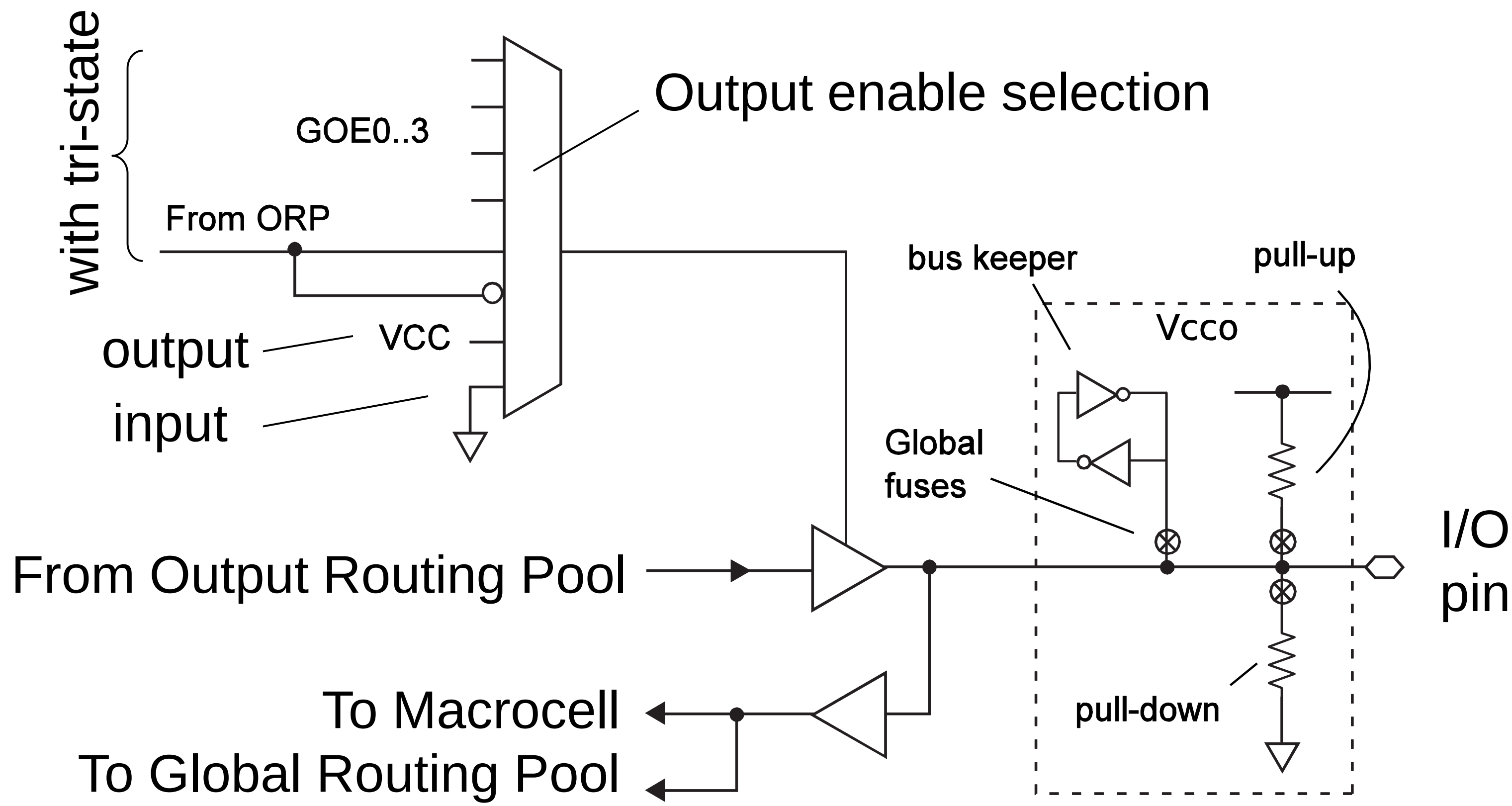


The FF can be configured as DFF, TFF, Latch, with optional asynchronous set/reset

Clock source selection

The output of the cell can be routed to some I/O cell via the Output Routing Pool and/or to other cells via the Global Routing Pool

CPLD – ispMACH 4000 I/O- cell



The output cell can be configured as input, output or bidirectional. Weak pull-up/down resistors and bus keepers are globally available.

CPLDs – Altera, Xilinx

- MAX II (0.18 um) with up to 2k cells and 8k flash bits, with SRAM based configuration + built-in flash memory
- MAX V – like MAX II + PLL
- MAX10 – 55 nm FPGA with built-in flash memory
- MAX 3000A – true CPLD, up to 512 cells, 3.3V

- Cool Runner II, up to 512 cells, 1.8V core, with SRAM based configuration + built in flash
- XC9500XL – true CPLD, up to 288 cells (5V, 3.3V, 2.5V)

CPLDs – Lattice

- ispMACH 4000 Z-ZE (zero power 1.8V core), C, B, V (1.8, 2.5, 3.3V core), up to 512 cells, probably **the fastest** true CPLD now
- MachXO, 1.2, 1.8, 2.5, 3.3V core, up to 2k cells (LUT4), RAM, with SRAM based configuration + built in flash memory
- MachXO2 – same as MachXO + up to 7k cells (LUT4), PLL, hardcores I2C, SPI, user flash memory
- MachXO3 – very similar to MachXO2 + up to 9k cells
- MachXO3D – as MachXO3 + dual boot and security features
- Actually MachXO… are FPGAs with built-in flash

(true) CPLD – summary

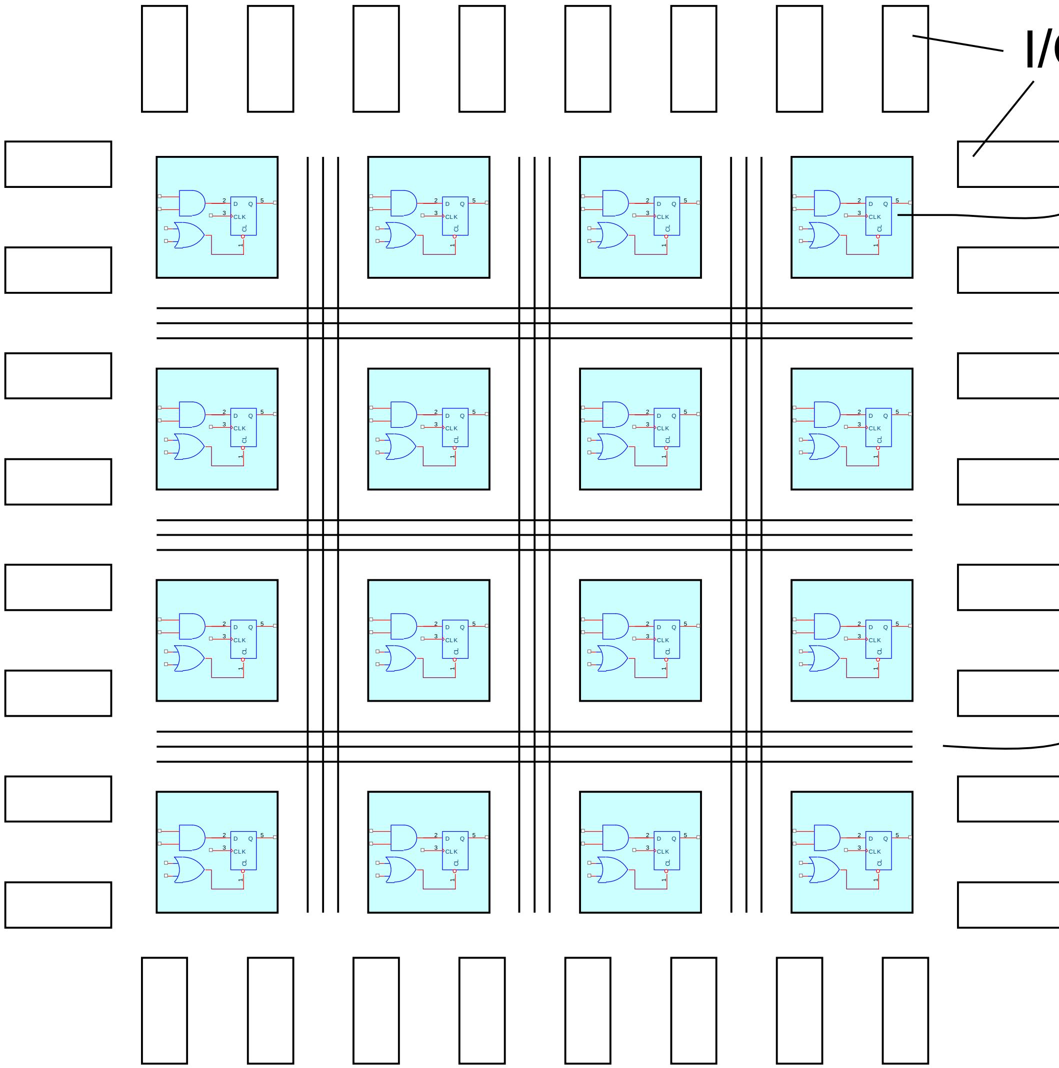
Sum of product terms architecture, similar to PAL/GAL

- Simple model of the internal delays and from pin to pin
- Ready to operate immediately after power up
- In situ programmable using JTAG, FLASH memory cells store the configuration (about 10,000 times)
- Reliable copy protection possible
- Radiation tolerant (the newer CPLDs are similar to FPGA + built-in FLASH and are NOT radiation tolerant!)

- Limited number of logic elements (up to about 1k)
- Higher price/logic element
- No internal RAM

only for the true CPLDs

FPGA – general structure



I/O Blocks and pins

Logic Block (LE, LC, Slice)

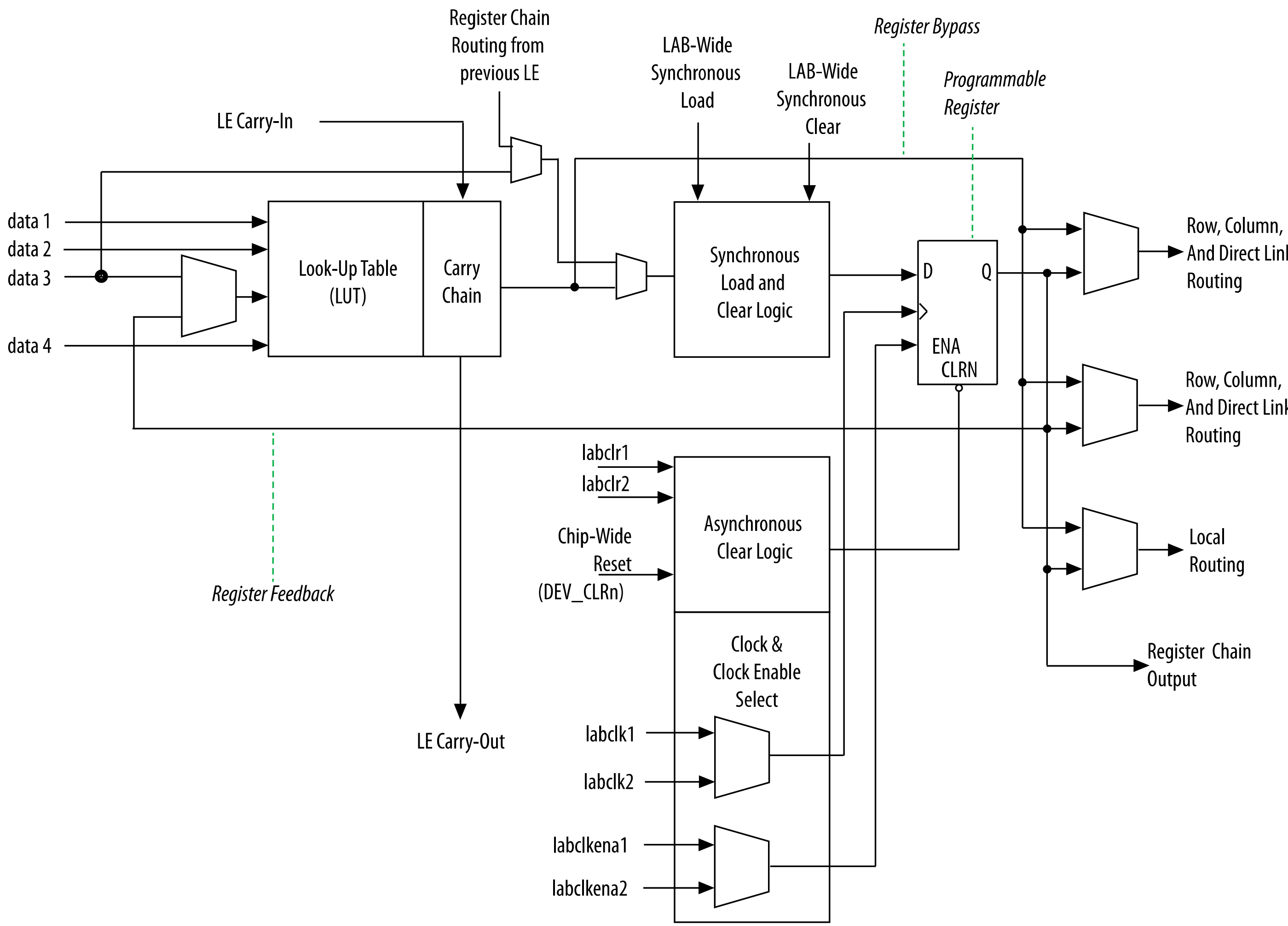
- contains a look up table (LUT) with 4 to 6 inputs and a FF. In some FPGAs several Logic Blocks are grouped into clusters with some local routing.

Routing channels

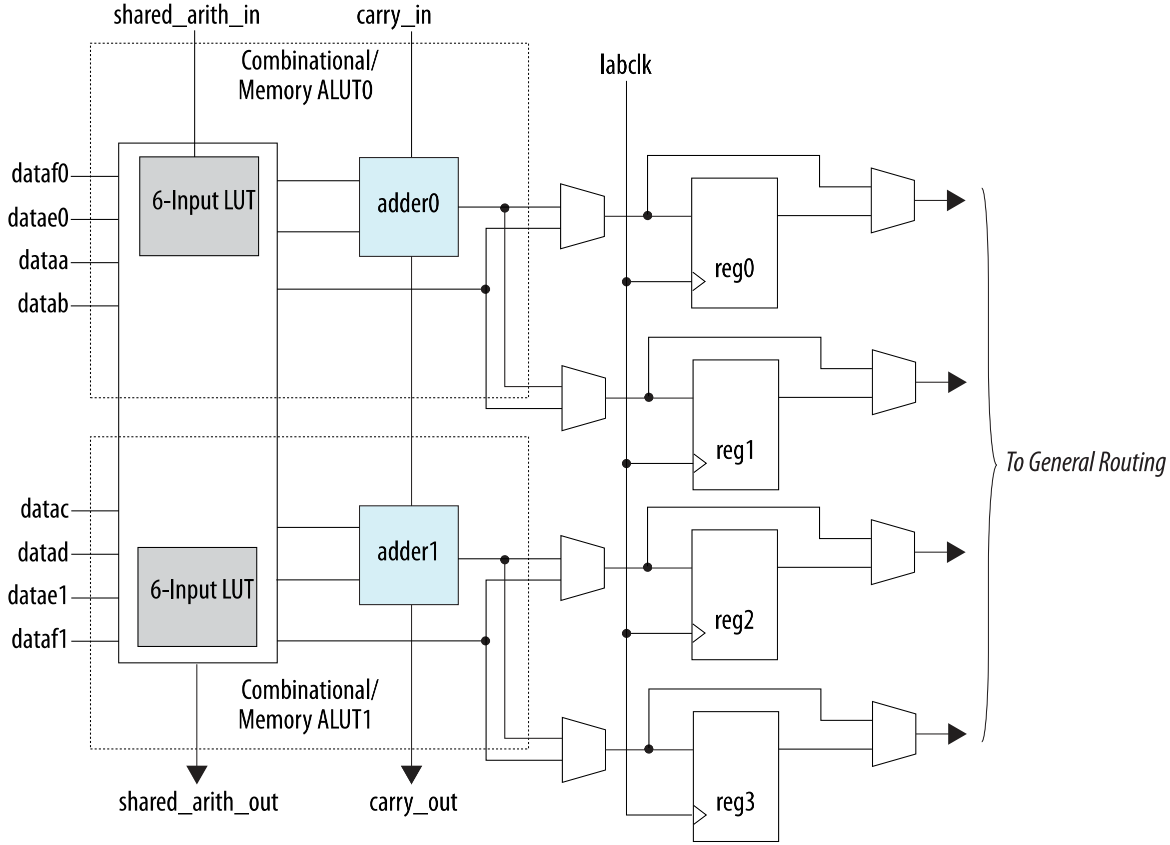
- general purpose
- for global signals like clocks, reset, output enable, with high fanout and low skew

FPGA – MAX10 with LUT4

This is the classical FPGA building block, still used in the low cost FPGAs

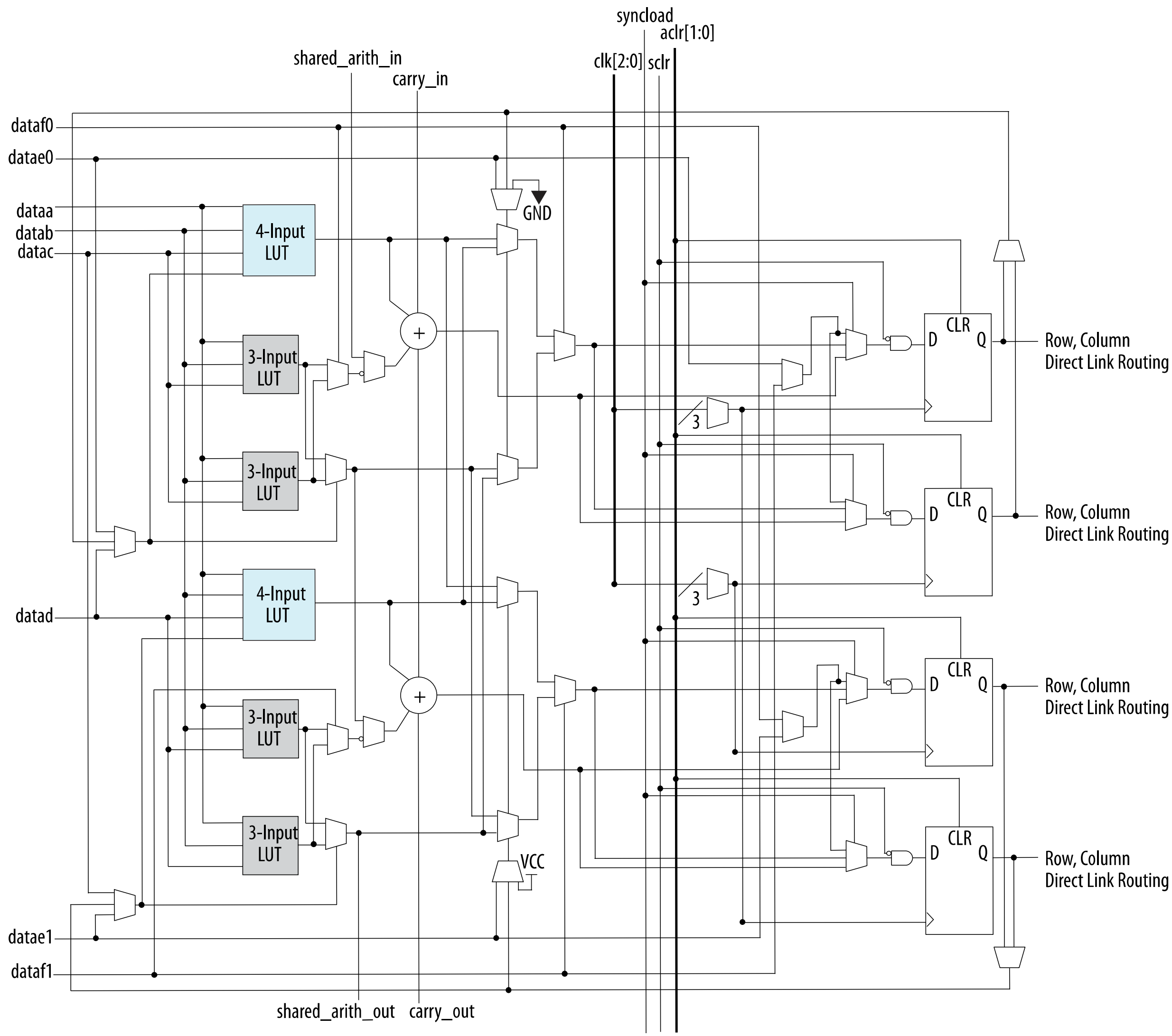


FPGA – Cyclone10GX with LUT6



In the modern sub-micron processes the routing delay becomes a substantial part of the whole delay. On the other side the logic needs less area. Therefore the leading manufacturers go to the larger LUT6.

FPGA – Cyclone10GX with LUT6



… same cell with more details

Low cost FPGAs overview

Name	LUT4 (k)	RAM kBits	18x18	PLLs	Tech
Cyclone V E	25-301	1760-12200	50-684	4-8	28nm (lp)
Cyclone 10 LP	6-120	270-3888	15-288	2-4	60nm
MAX10	2-50	108-1640	16-144	2-4	55nm

built-in flash

Name	LUT4 (k)	RAM kBits	18x18	PLLs	Tech
Spartan 6	4-147	216-4800	8-180	2-6	45nm
Spartan 7	6-102	180-4320	10-160	2-8	28nm
Artix-7	13-215	208-974	40-740	3-10	28nm

equivalent LUT4, but LUT6

Low cost FPGAs overview

LATTICE SEMICONDUCTOR

Name	LUT4 (k)	RAM kBits	18x18	PLLs	Tech
LatticeXP	3-20	54-396	---	2-4	130nm
LatticeXP2	5-40	166-885	12-32	2-4	90nm

XP and XP2 have built-in configuration flash

with SerDes

Name	LUT4 (k)	RAM kBits	18x18	SerDes	Speed Gbps	Tech
Cyclone V GX/T	77-301	4460-12200	300-684	2-12	6.144	28nm
Cyclone 10 GX	85-220	5820-11740	168-384 floating point mode	6-10	12.5	20nm
Spartan 6	24-147	936-4824	38-180	2-8	3.125	45nm
Artix-7	16-215	208-2888	60-740	2-16	6.6	28nm

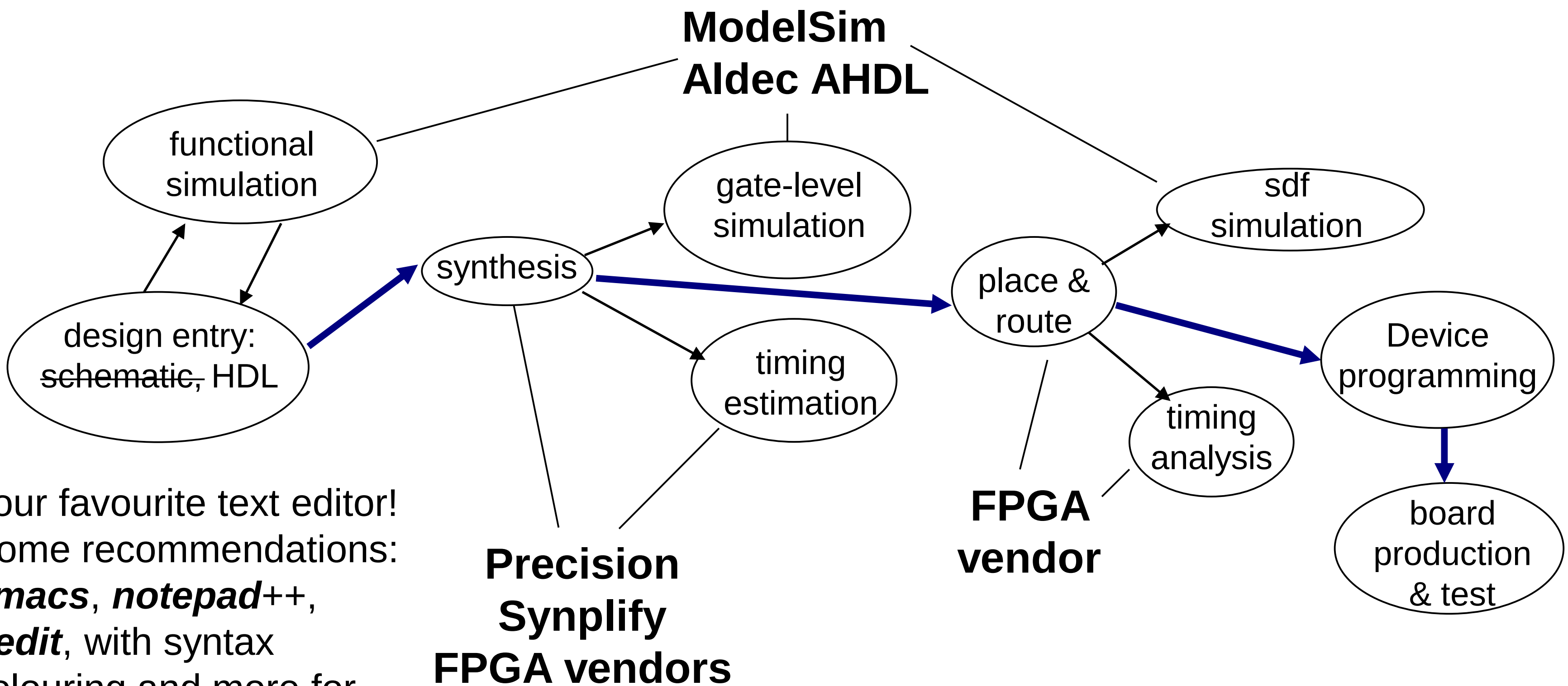
FPGA summary

- The price/logic goes down
- The speed goes up
- Special blocks like RAM, DSP (multiplier), CPU …
- Flexible I/O cells, including fast serial links and differential signals
- Infinitely times programmable (with some exceptions)

- External memory or interface for initialization after power up needed – copy protection impossible (with some exceptions)
- More sensitive to radiation, compared to CPLD (with some exceptions)

Manufacturers: **Actel-Microsemi-Microchip, Altera-Intel, Lattice, Xilinx**

Design flow CPLD/FPGA



Your favourite text editor! Some recommendations: ***emacs***, ***notepad***++, ***nedit***, with syntax colouring and more for **`VHDL`** and **`Verilog`**

Each step can take seconds, minutes, hours ... (place & route)

FPGA development tools

- Each manufacturer has own tools, absolutely necessary for placing and routing, optionally for synthesis, simulation etc. The free versions have some limitations
- Leading suppliers of synthesis tools - Mentor Graphics (Precision), Synopsys (Synplicity - Synplify)
- Leading suppliers of simulation tools - Mentor Graphics (ModelSim), Aldec (Active HDL)
- The FPGA manufacturers offer free but limited versions of the synthesis and simulation tools mentioned above

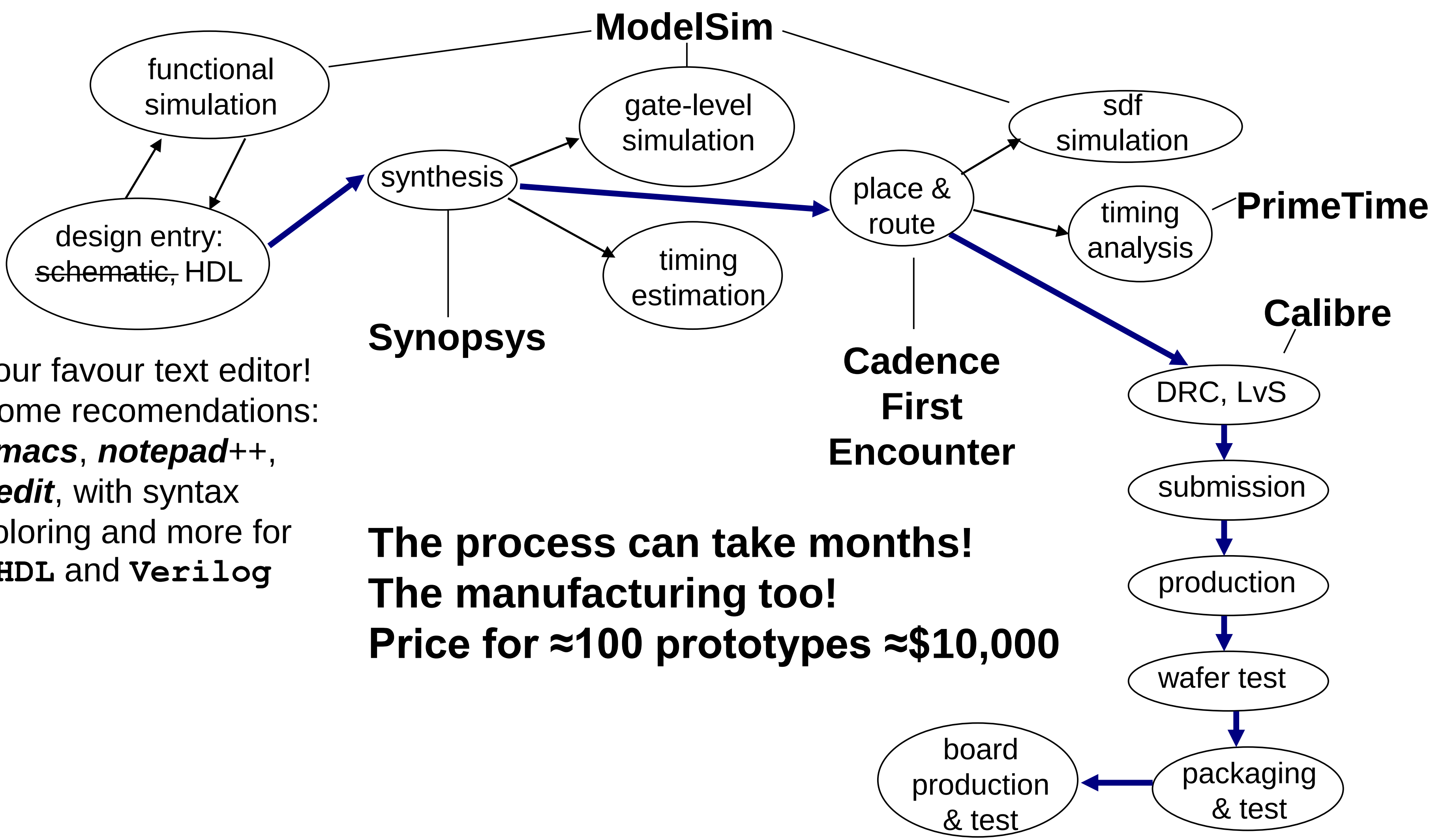
ASICs - Standard Cells, Gate Arrays, Full Custom

- Standard Cells
  - rich library with primitive functions and flip-flops
  - I/O cells for different standards and voltages
  - core generators for memory, CPU, interfacing, PLL
  - the user must pay all production masks
  - multiproject wafer option for prototyping
- Gate Array
  - array of ready simple gates
  - the user prepares only some routing masks
  - compared to Standard Cells: cheaper, slower, no mixed mode
- Full custom – for very high volumes
  - the most optimal, even longer development time and higher costs

ASIC $\leftrightarrow$ FPGA

- ASICs compared to CPLD and FPGAs:
  - lower price in high volume production runs
  - possibility for mixed mode designs (with analog part)
  - higher design density, higher operation speed, lower power
  - much longer development time, several months per submission
  - higher development costs and much more expensive software
  - more tolerant to radiation
- FPGA to ASIC
  - eASIC – now part of Intel?
  - faster development, lower risk and cost, compared to pure ASIC

Design flow ASIC



Your favour text editor!
Some recomendations:
emacs, ***notepad***++,
nedit, with syntax
coloring and more for
`VHDL` and **`Verilog`**

The process can take months!
The manufacturing too!
Price for ≈100 prototypes ≈$10,000