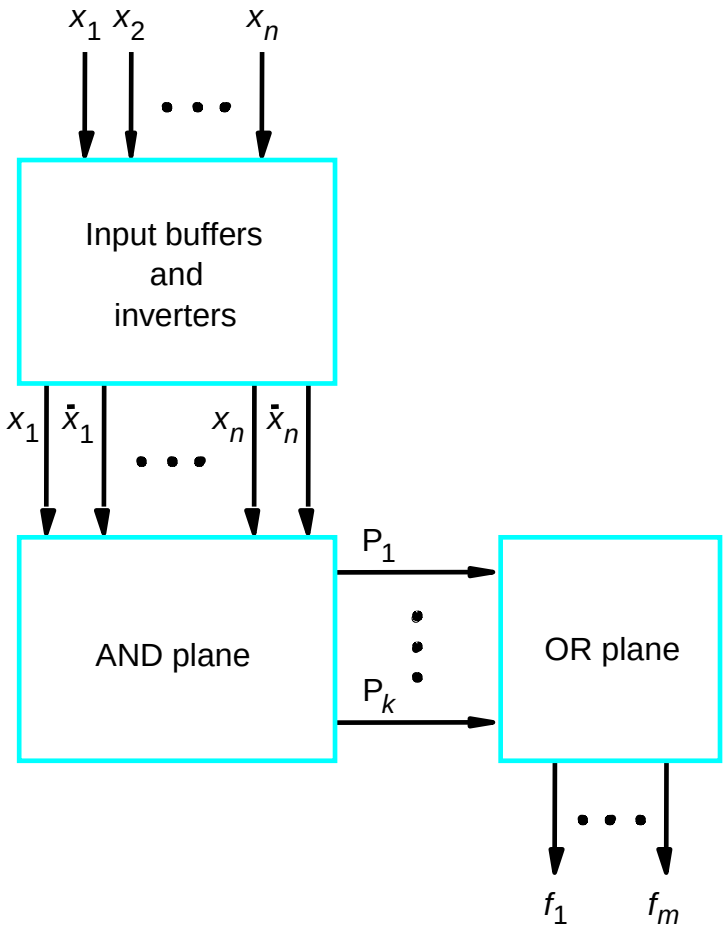
IE1205 Digital Design:

F11: Programmerbar Logik, VHDL för Sekvensnät

Programmable Logic Devices

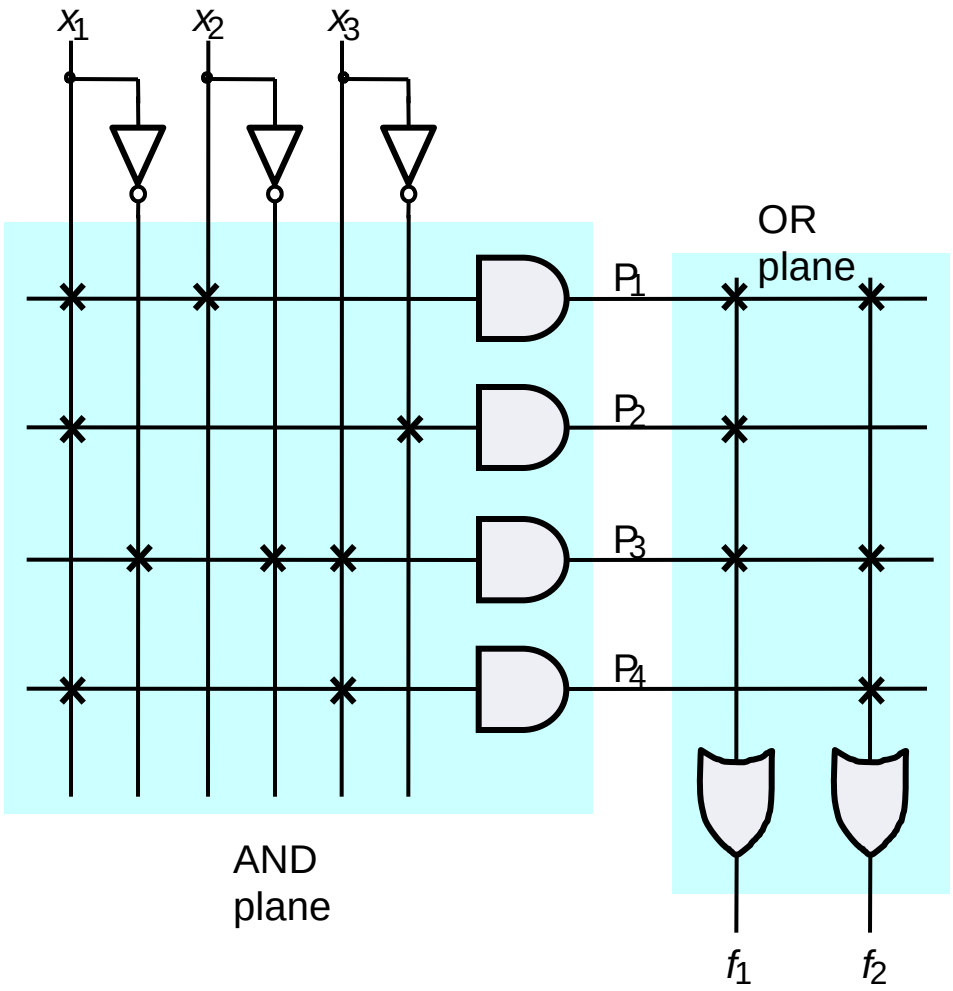
- Under 1970-talet introducerades programmerbara logiska kretsar som betecknas *programmable logic device (PLD)*
- De bygger på en struktur med en AND-OR-matris som gör det enkelt att implementera SOP-uttryck

Struktur av en PLD



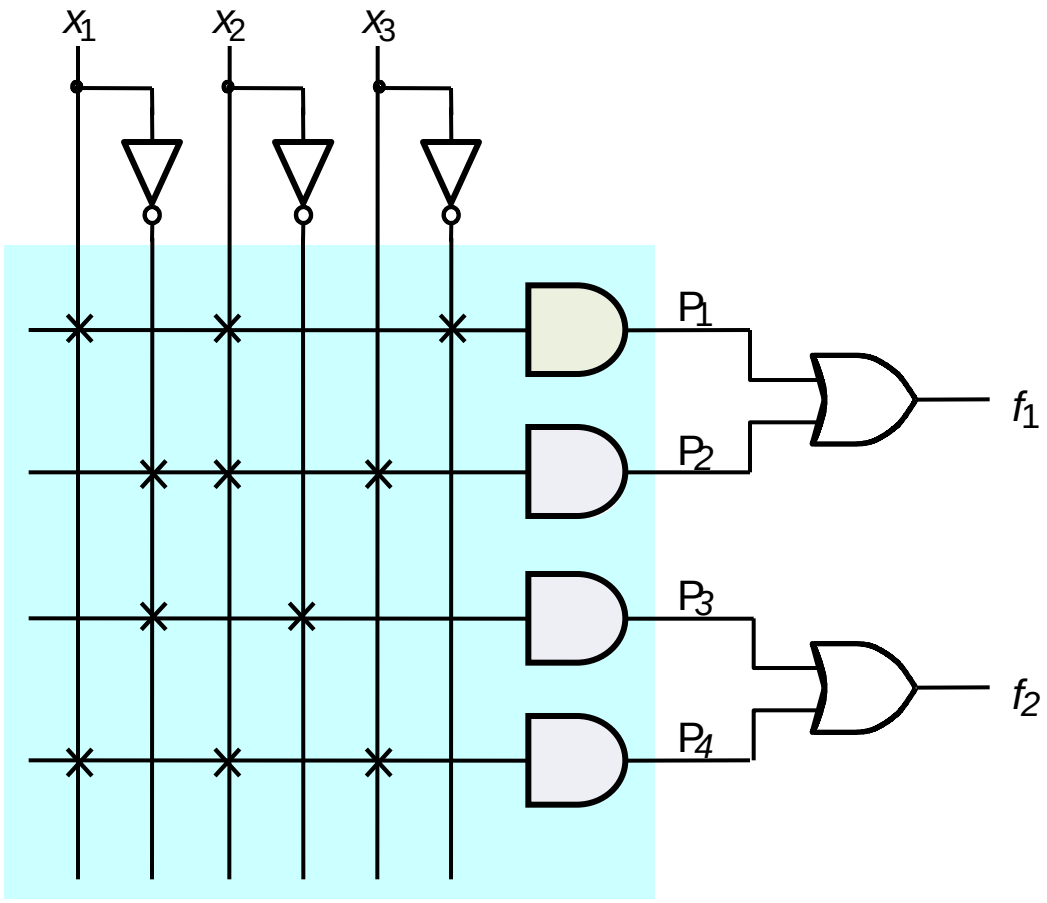
Programmable Logic Array (PLA)

- Både AND- och OR-matriserna är programmeringsbara



Programmable Array Logic (PAL)

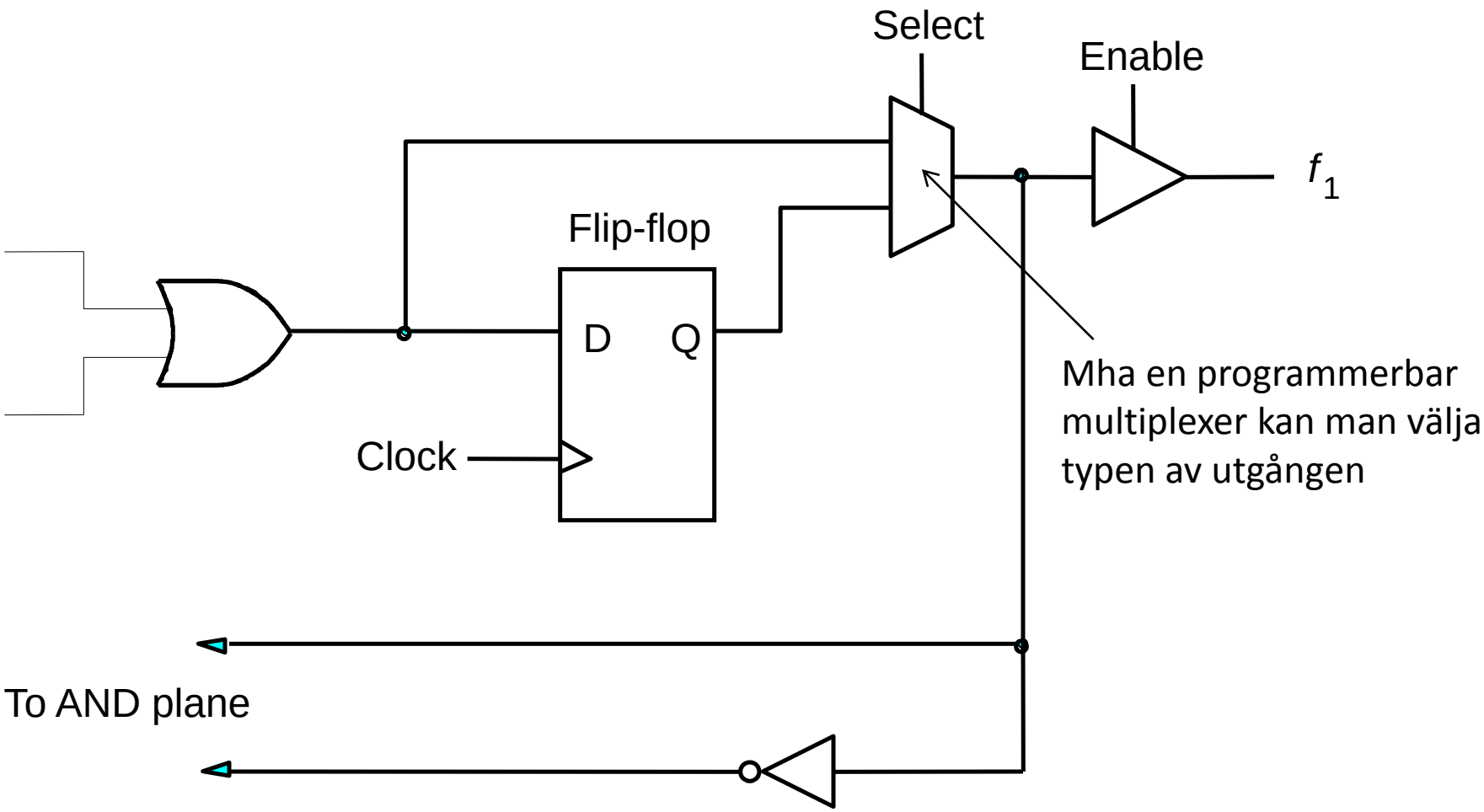
- Bara AND-matrisen är programmeringsbar



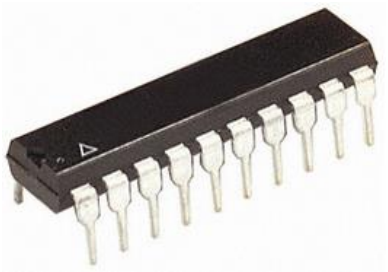
Kombinatoriska och registerutgångar

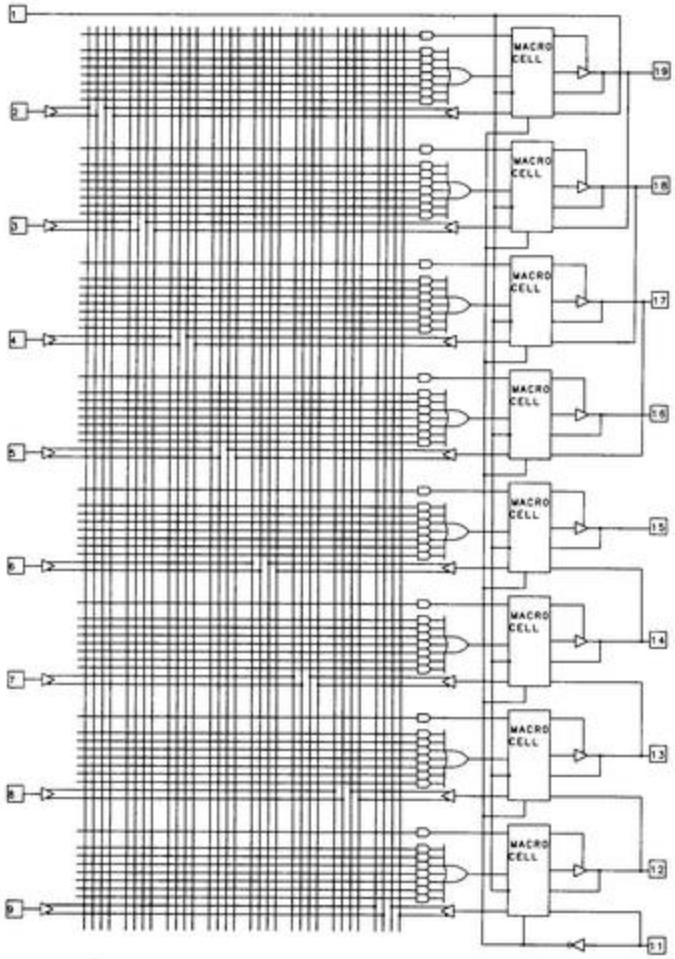
- I de tidigare PLD-kretasarna fanns det
  - kombinatoriska utgångar
  - registerutgångar (utgångar med en vippa)
- För varje krets fanns det ett fast antal kombinatoriska och registerutgångar
- För att öka flexibiliteten introducerade man *makrocellen* där man kunde välja om en utgång skulle vara en kombinatorisk eller en registerutgång

Macroceller i en PLD



PAL



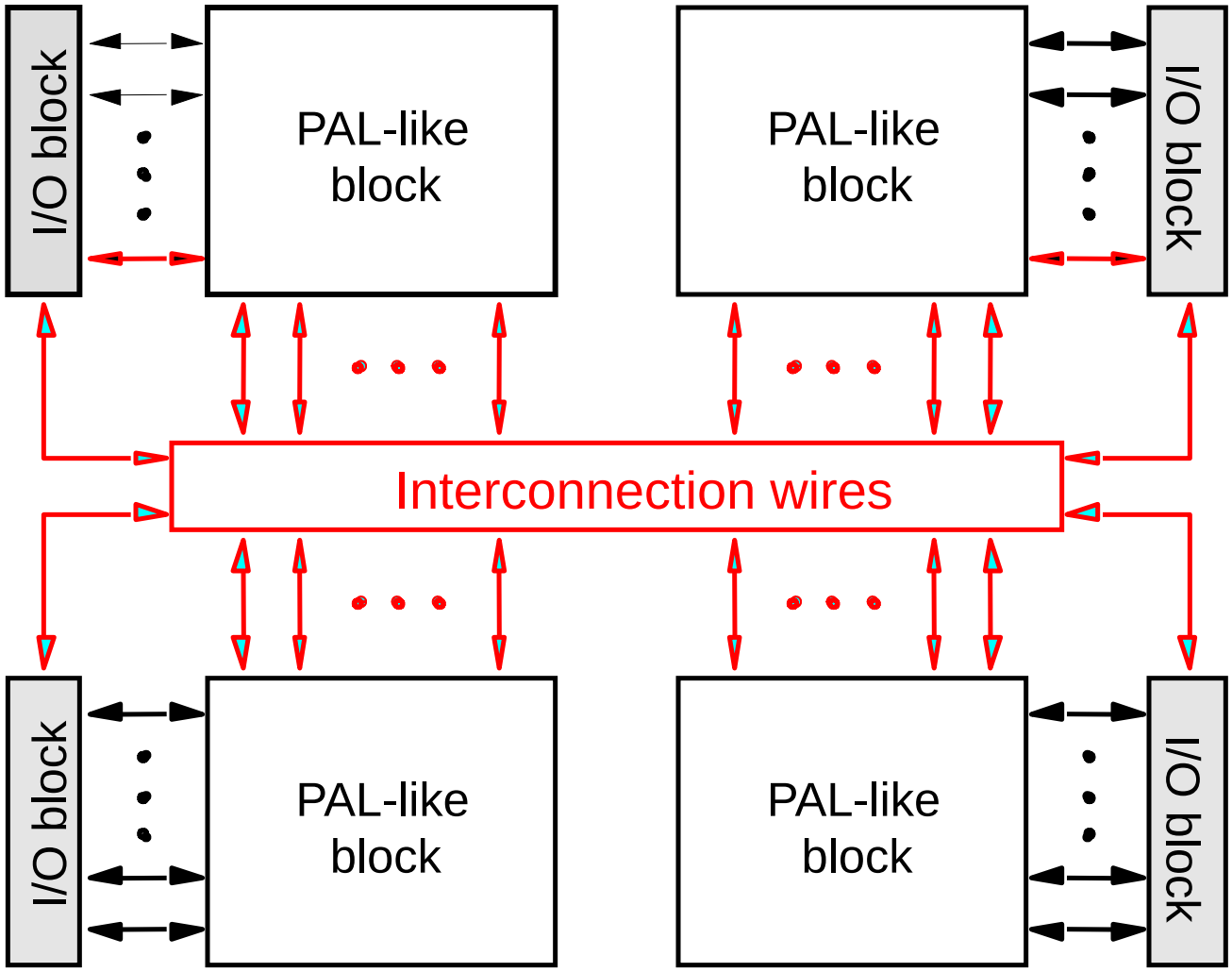

MACRO CELL
MACRO CELL
MACRO CELL
MACRO CELL
MACRO CELL
MACRO CELL
MACRO CELL
MACRO CELL

Programmering av PLD:er

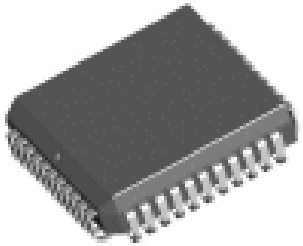
Komplexa PLD:er (CPLD)

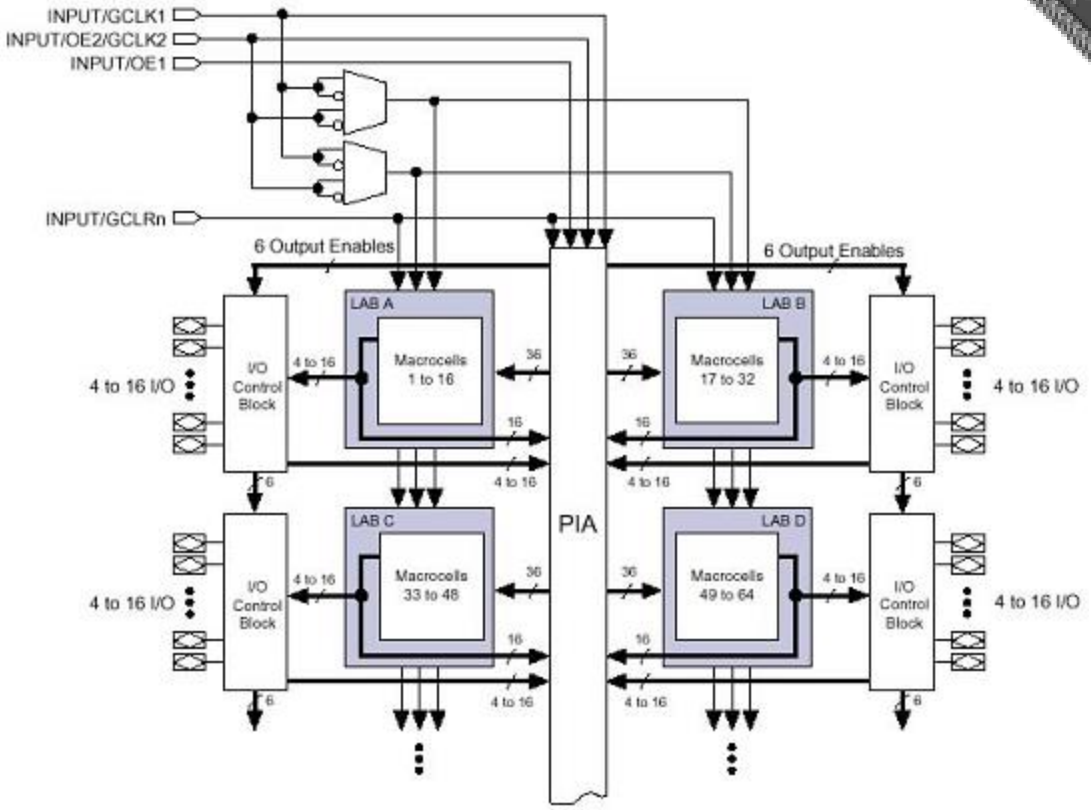
- PLD:erna var ganska små (PALCE 22V10 hade 10 vippor)
- För att skapa större programmerbara kretsar utvecklade man en struktur bestående av flera PLD-liknande block

Struktur CPLD



CPLD (MAX)

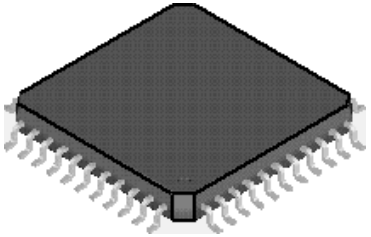



INPUT/GCLK1
INPUT/OE2/GCLK2
INPUT/OE1
INPUT/GCLRn
6 Output Enables
6 Output Enables
LAB A
LAB B
LAB C
LAB D
Macrocells 1 to 16
Macrocells 17 to 32
Macrocells 33 to 48
Macrocells 49 to 64
I/O Control Block
PIA
4 to 16 I/O
4 to 16
36
16
6

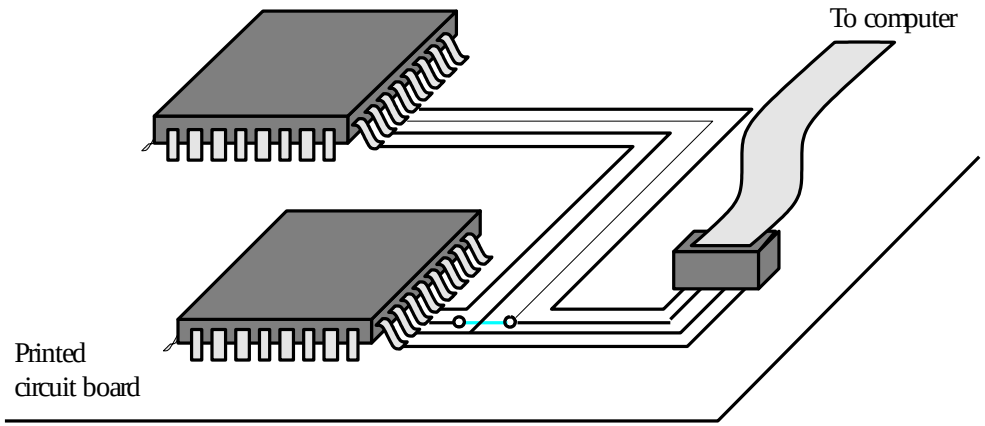
Programmering av CPLD:er via JTAG-interface

- Moderna CPLD:er (och FPGA:er) kan programmeras genom att ladda ned kretsbekrivningen (programmerings-informationen) via en kabel
- Nedladdningen använder oftast en standardiserad port: *JTAG-porten*

Programmering via JTAG-porten



(a) CPLD in a Quad Flat Pack (QFP) package

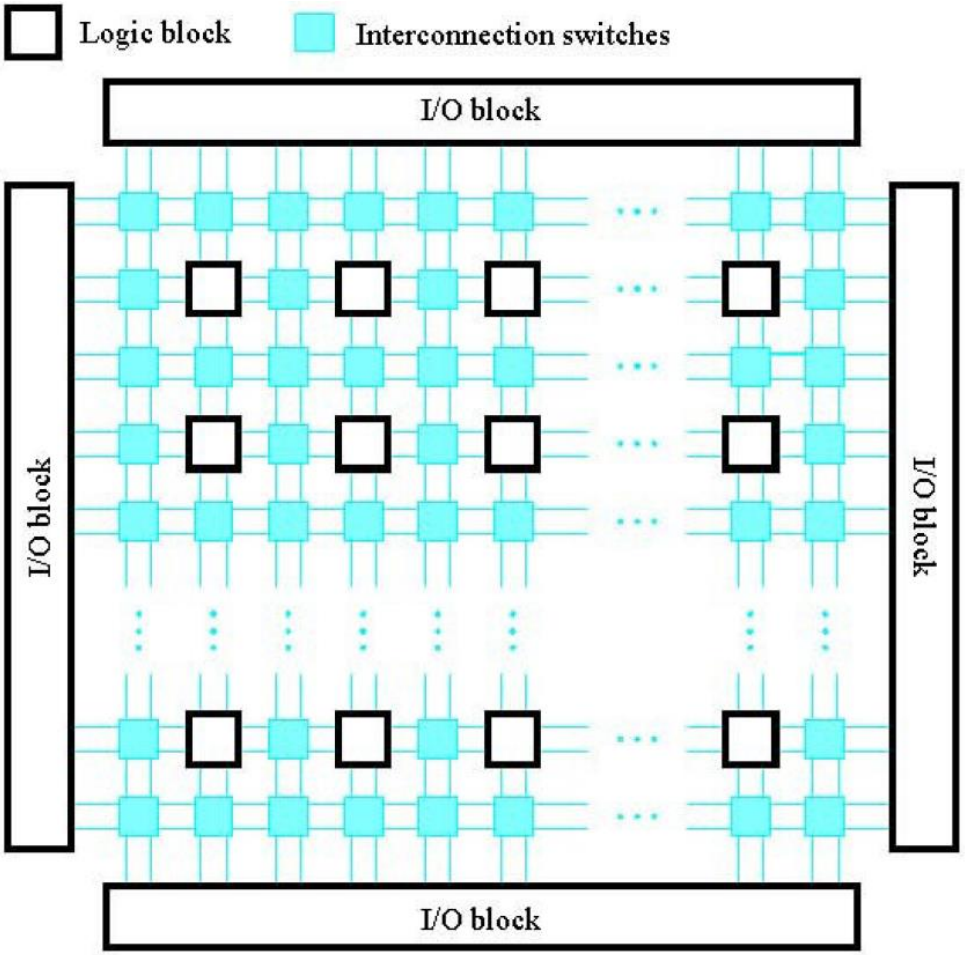


(b) JTAG programming

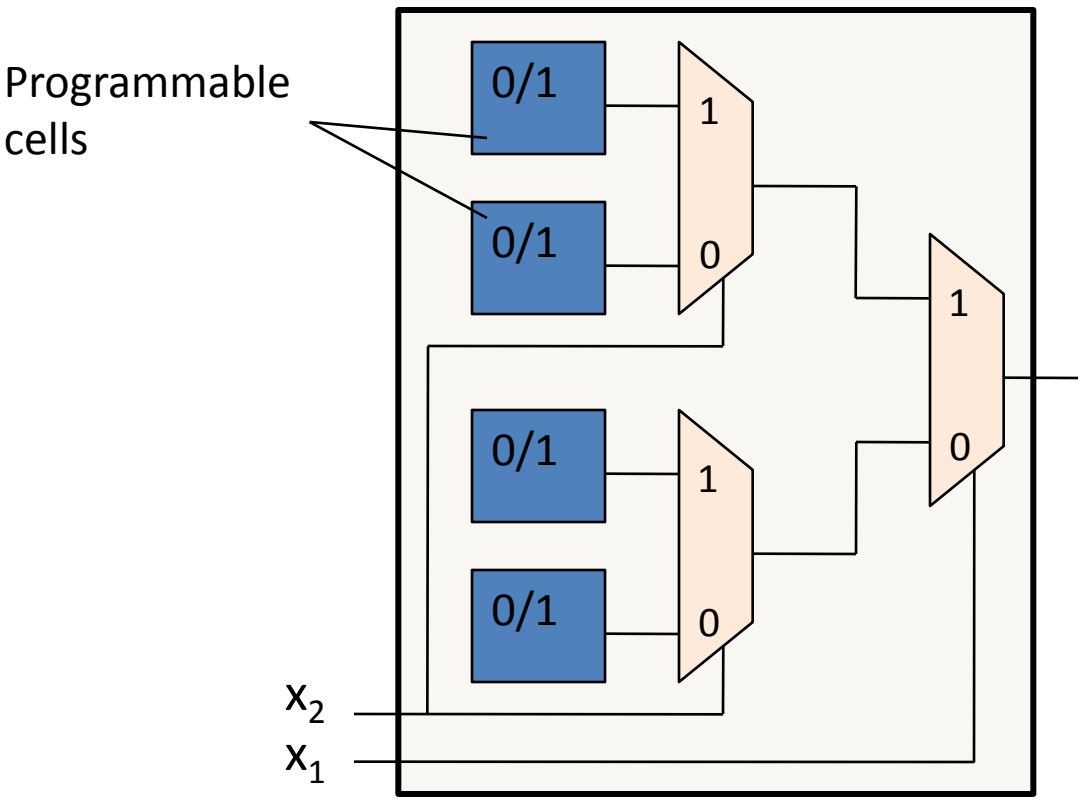
FPGA-kretsar

- CPLD:er baseras på AND-OR-matrisen och det blir svårt att göra riktigt stora kretsar
- FPGA (Field Programmable Gate Array) kretsarna använder en annan koncept som baseras på *logiska block*

Struktur av en FPGA-krets



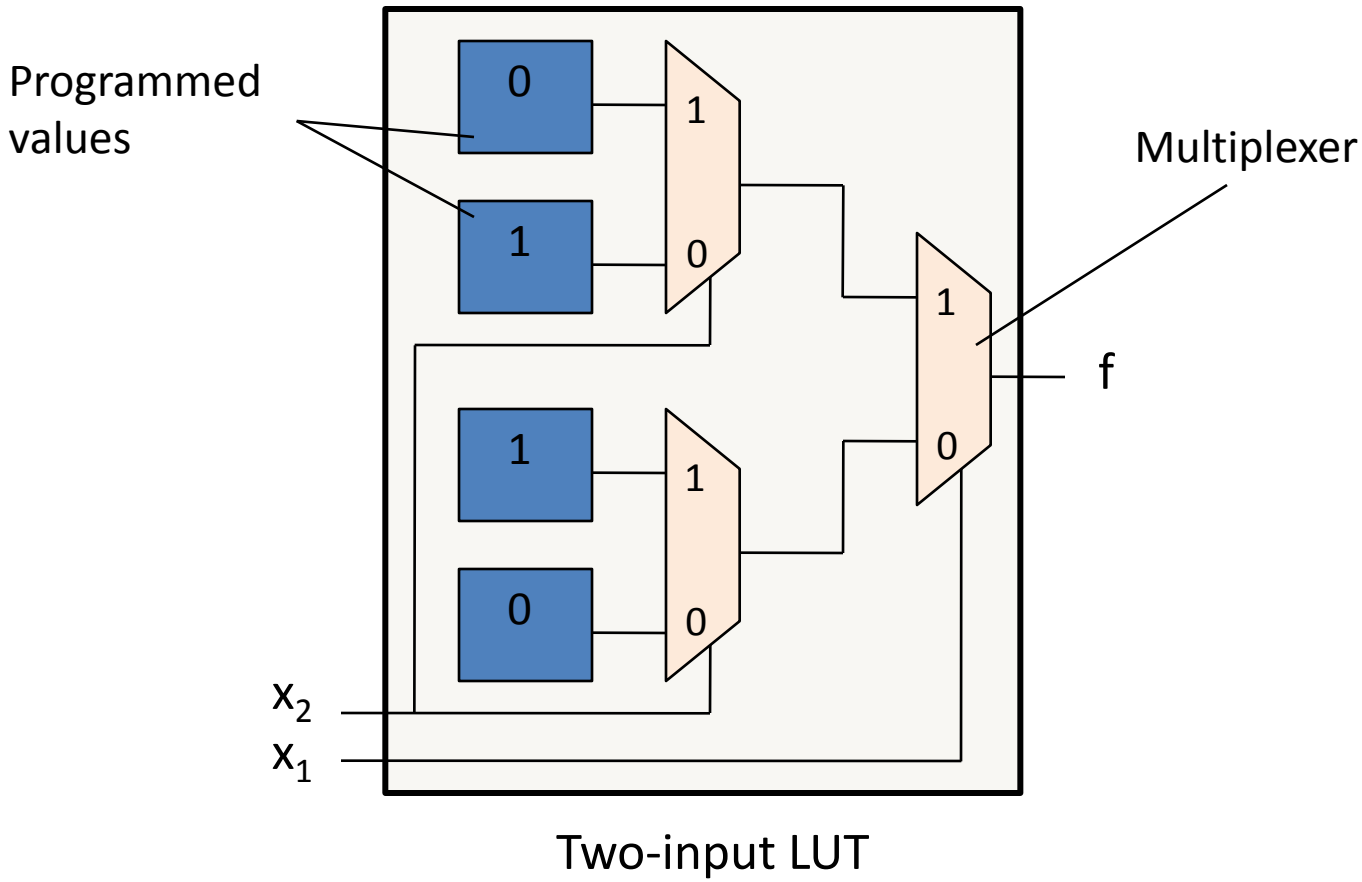
Look-up-tables (LUT)



Two-input LUT

A LUT with *n* inputs can realize all combinational functions with *n* inputs
The usual size in an FPGA is *n*=4

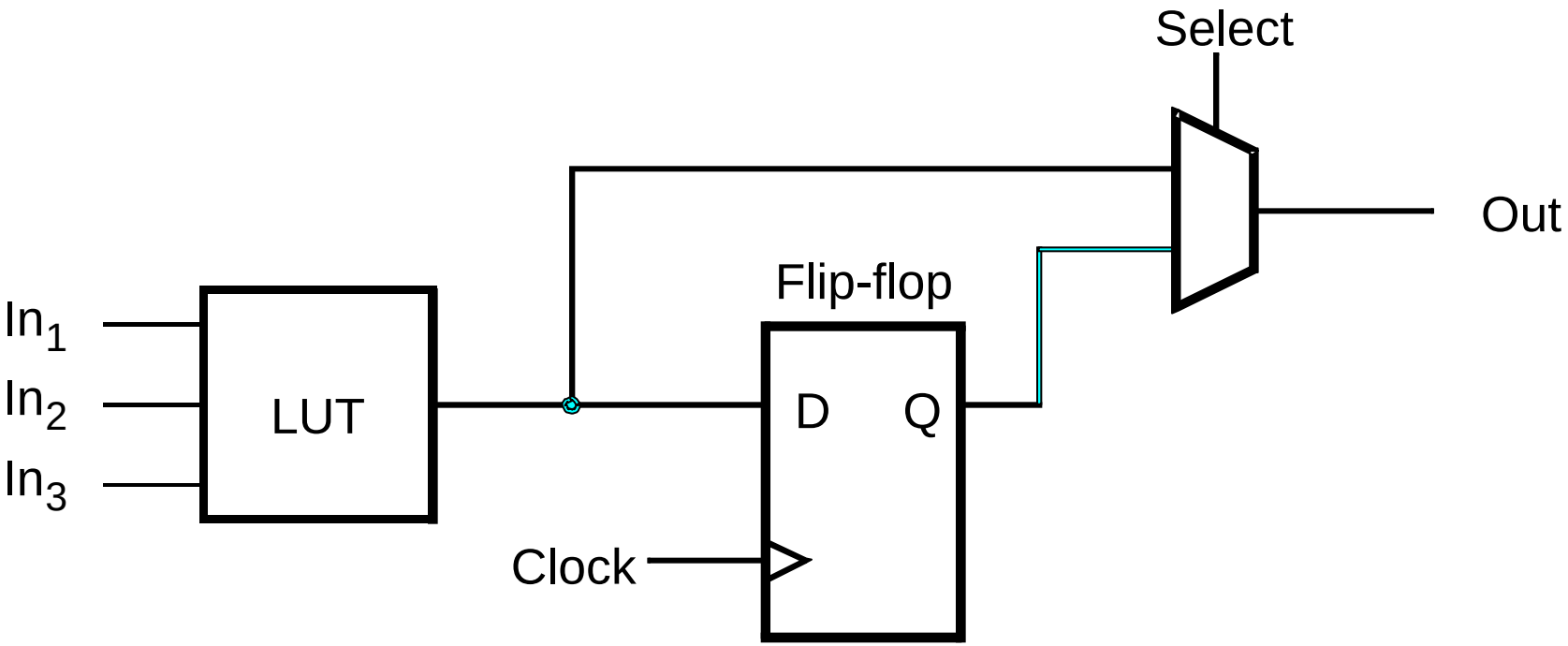
Example: XOR-Gate



Two-input LUT

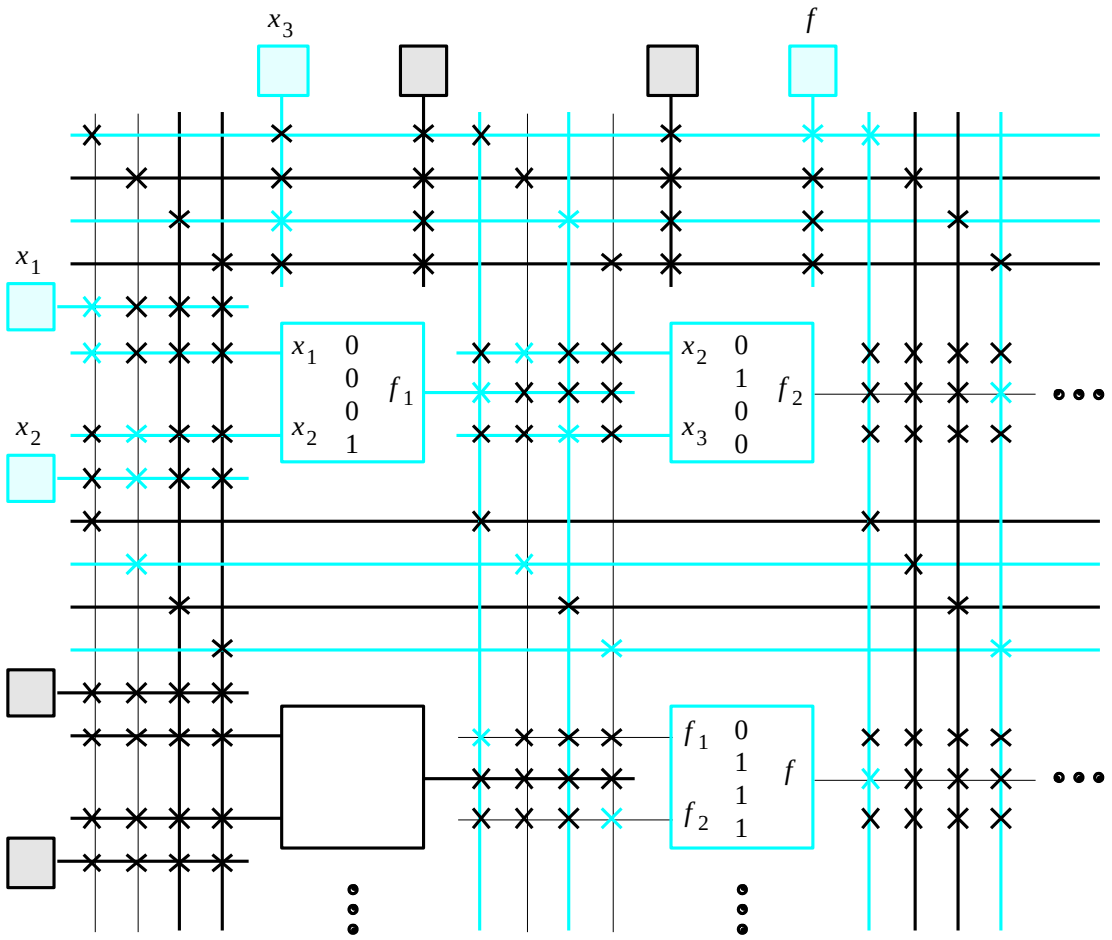
Logisk Block i en FPGA

- En logisk block i en FPGA består ofta av en LUT, en vippa och en multiplexer



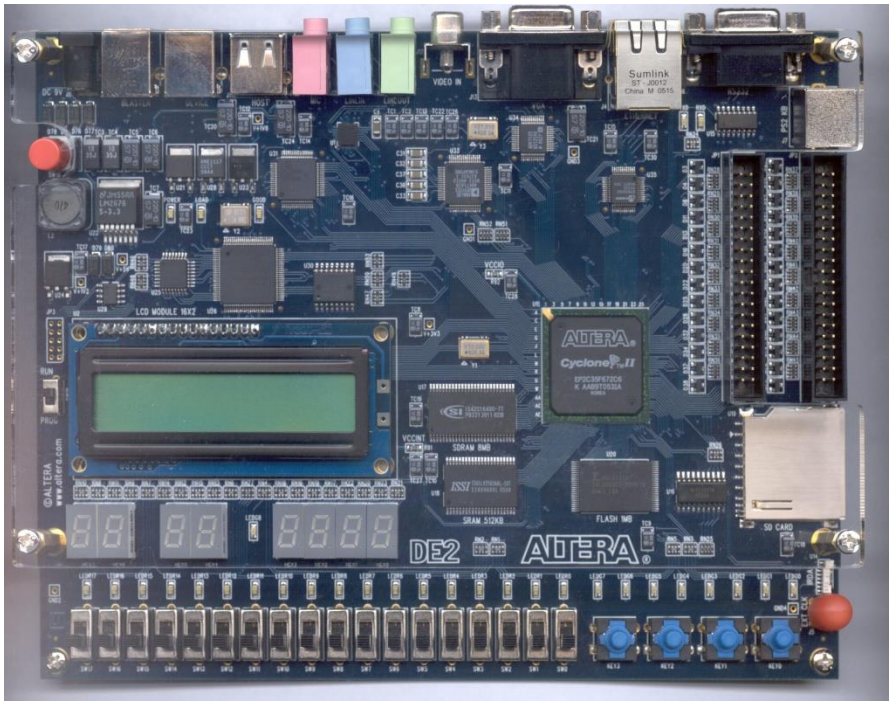
Programmering av LUT:ar och Förbindelsematris i en FPGA

- Blå kryss: Förbindelsen är programmerad
- Svart kryss: Förbindelsen är inte programmerad

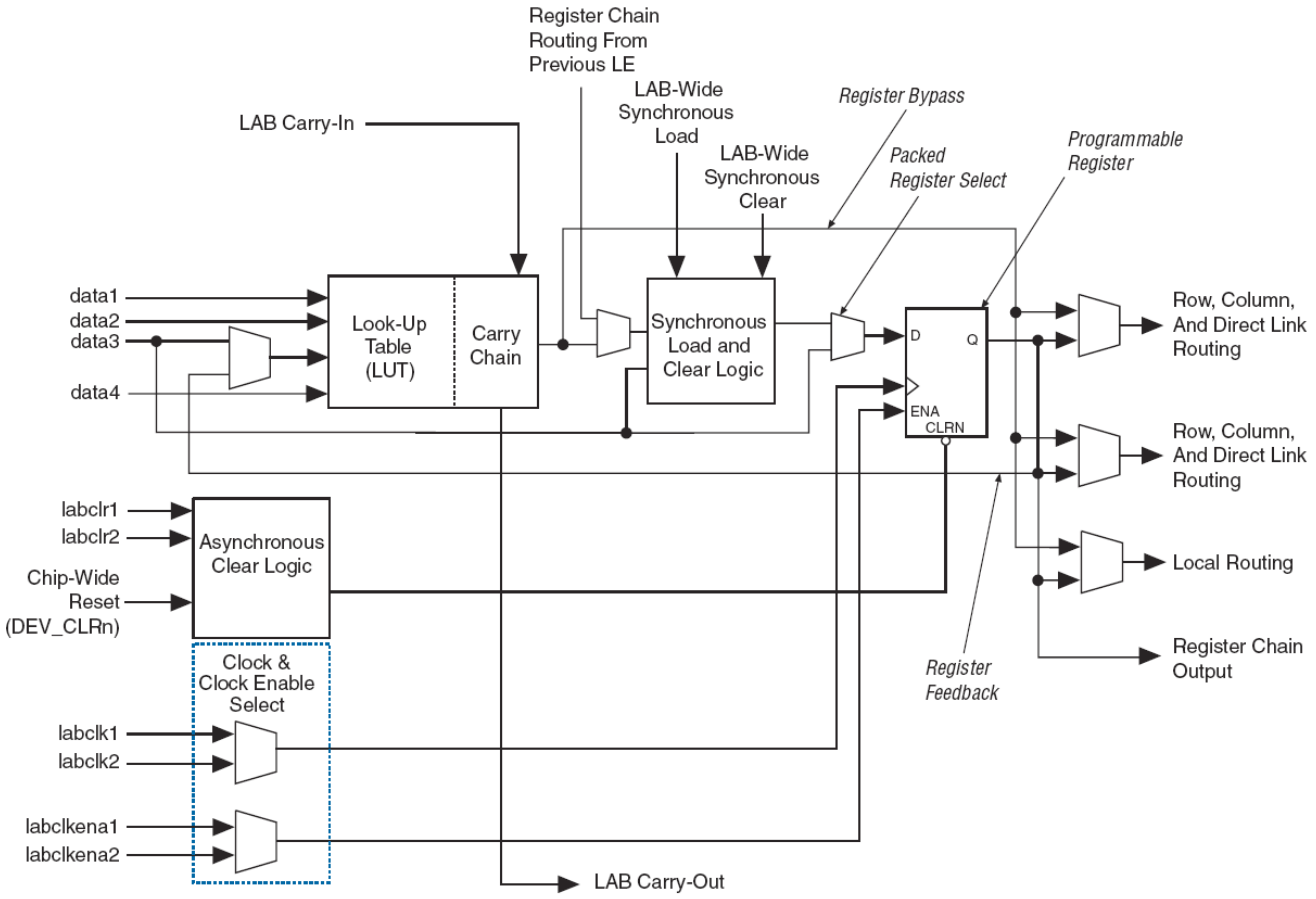


DE2 University Board

- DE2 Board
  - Cyclone II EP2C35 FPGA
  - 4 Mbytes of flash memory
  - 512 Kbytes of static RAM
  - 8 Mbytes of SDRAM
  - Several I/O-Devices
  - 50 MHz oscillator



Cyclone II Logic Element



Cyclone II Family

Table 1–1. Cyclone II FPGA Family Features							
Feature	**EP2C5**	**EP2C8** *(2)*	**EP2C15** *(1)*	**EP2C20** *(2)*	**EP2C35**	**EP2C50**	**EP2C70**
LEs	4,608	8,256	14,448	18,752	33,216	50,528	68,416
M4K RAM blocks (4 Kbits plus 512 parity bits	26	36	52	52	105	129	250
Total RAM bits	119,808	165,888	239,616	239,616	483,840	594,432	1,152,000
Embedded multipliers *(3)*	13	18	26	26	35	86	150
PLLs	2	2	4	4	4	4	4
Maximum user I/O pins	158	182	315	315	475	450	622

DE1 DE2

(3) Total Number of 18x18 Multipliers

Stratix III Family

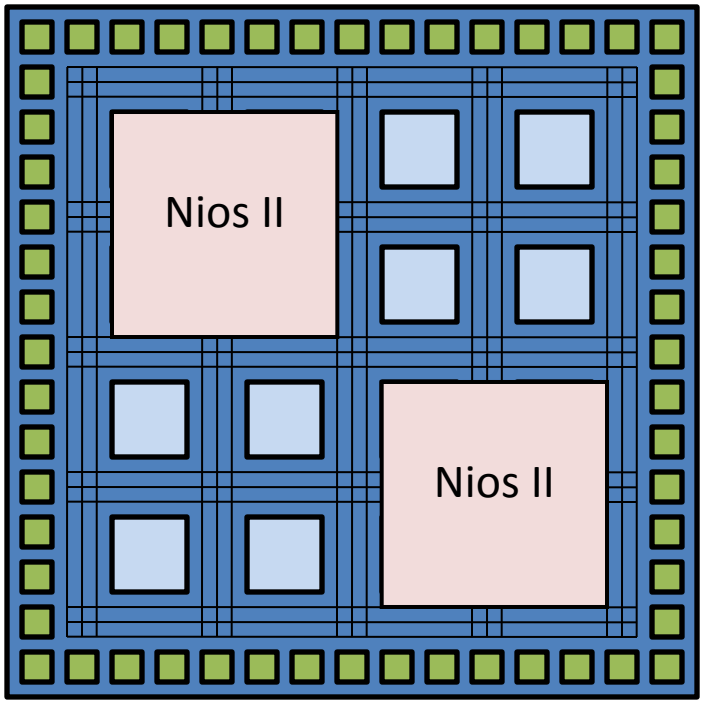
Table 1–1. Stratix III FPGA Family Features

	Device/ Feature	ALMs	LEs	M9K Blocks	M144K Blocks	MLAB Blocks	Total Embedded RAM Kbits	MLAB RAM Kbits *(2)*	Total RAM Kbits *(3)*	18×18-bit Multipliers (FIR Mode)	PLLs
Stratix III Logic Family	EP3SL50	19K	47.5K	108	6	950	1,836	297	2,133	216	4
	EP3SL70	27K	67.5K	150	6	1,350	2,214	422	2,636	288	4
	EP3SL110	43K	107.5K	275	12	2,150	4,203	672	4,875	288	8
	EP3SL150	57K	142.5K	355	16	2,850	5,499	891	6,390	384	8
	EP3SL200	80K	200K	468	36	4,000	9,396	1,250	10,646	576	12
	EP3SE260	102K	255K	864	48	5,100	14,688	1,594	16,282	768	12
	EP3SL340	135K	337.5K	1,040	48	6,750	16,272	2,109	18,381	576	12
Stratix III Enhanced Family	EP3SE50	19K	47.5K	400	12	950	5,328	297	5,625	384	4
	EP3SE80	32K	80K	495	12	1,600	6,183	500	6,683	672	8
	EP3SE110	43K	107.5K	639	16	2,150	8,055	672	8,727	896	8
	EP3SE260 *(1)*	102K	255K	864	48	5,100	14,688	1,594	16,282	768	12

DE3 Board

Flera processorer kan implementeras på en FPGA

- Nios II är en så kallad ‘soft-processor’ (32-bit) som kan implementeras på Altera FPGA-kretsen
- Dagens FPGA-kretsar är så stora att flera processorer får plats på en enda FPGA-krets



Mycket kraftfulla multiprocessor system kan skapas på en FPGA!

Overview Nios II family (Extract)

Table 5–1. Nios II Processor Cores (Part 1 of 2)

Feature		Core		
		Nios II/e	Nios II/s	Nios II/f
Objective		Minimal core size	Small core size	Fast execution speed
Performance	DMIPS/MHz *(1)*	0.15	0.74	1.16
	Max. DMIPS *(2)*	31	127	218
	Max. f_{MAX} *(2)*	200 MHz	165 MHz	185 MHz
Area		< 700 LEs; < 350 ALMs	< 1400 LEs; < 700 ALMs	< 1800 LEs; < 900 ALMs
Pipeline		1 Stage	5 Stages	6 Stages
External Address Space		2 Gbytes	2 GBytes	2 GBytes
Instruction Bus	Cache	–	512 bytes to 64 kbytes	512 bytes to 64 kbytes
	Pipelined Memory Access	–	Yes	Yes
	Branch Prediction	–	Static	Dynamic
	Tightly Coupled Memory	–	Optional	Optional

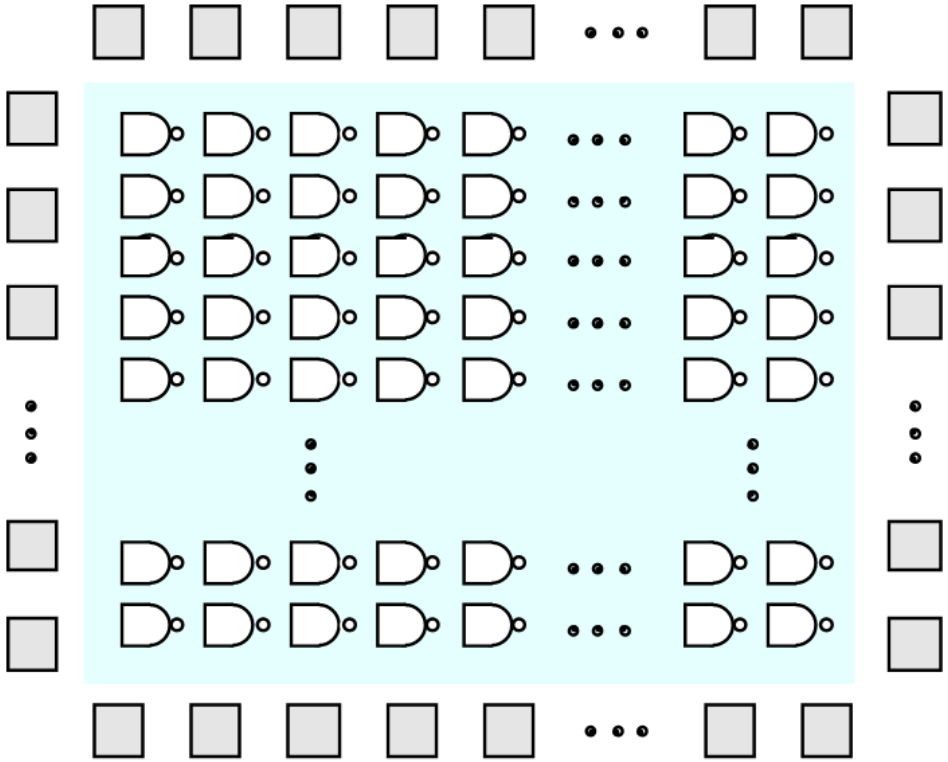
(2) Numbers are for the fastest device in the Stratix II family

ASIC:ar

- En ASIC (Application Specific Integrated Circuit) är en krets som görs i en halvledarfabrik
- I en *full custom* integrerad krets skräddarsy man i princip hela kretsen
- I en ASIC har vissa arbetssteg redan gjorts för att minska design-tiden och kostnaden

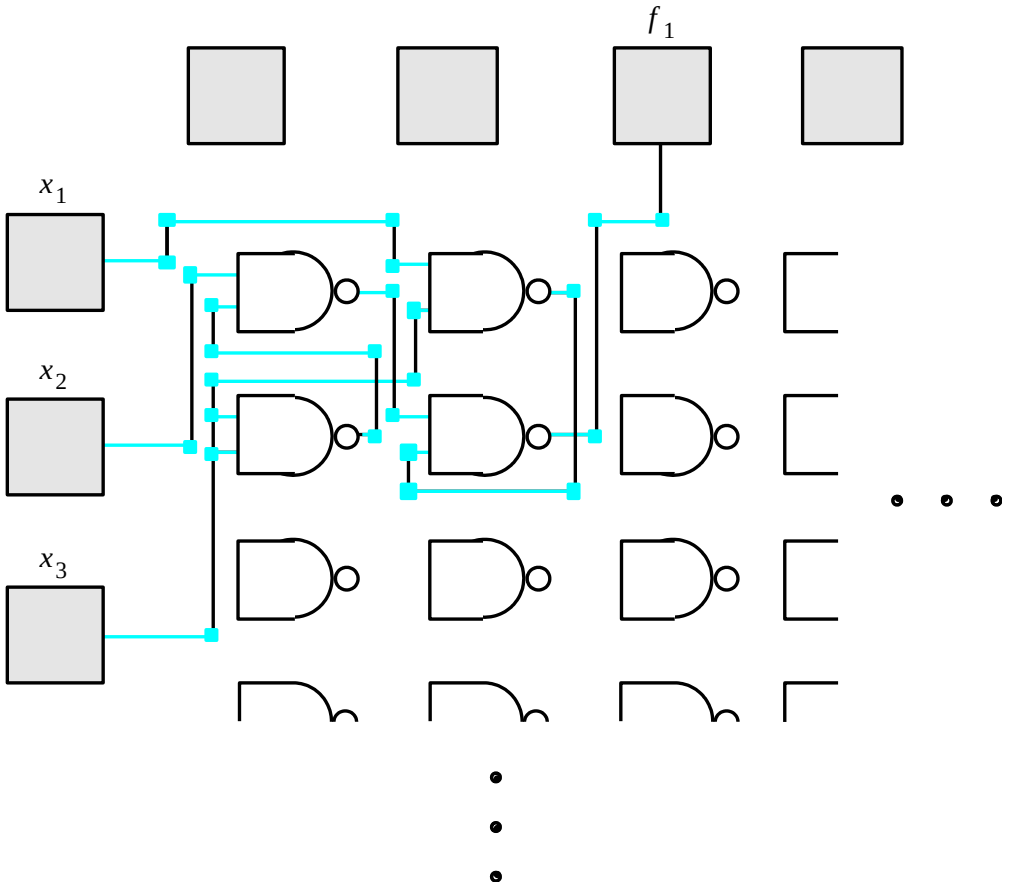
Gate Array

- I en Gate Array finns redan grindarna (eller transistorerna) på kiseln



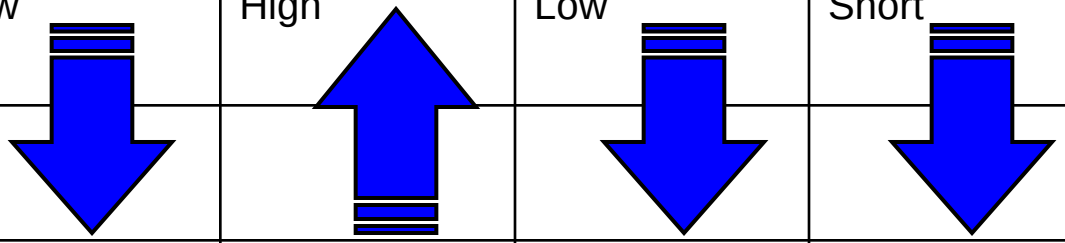
Gate Array

- Man skapar bara förbindelserna mellan ingångarna, grindarna och utgångarna

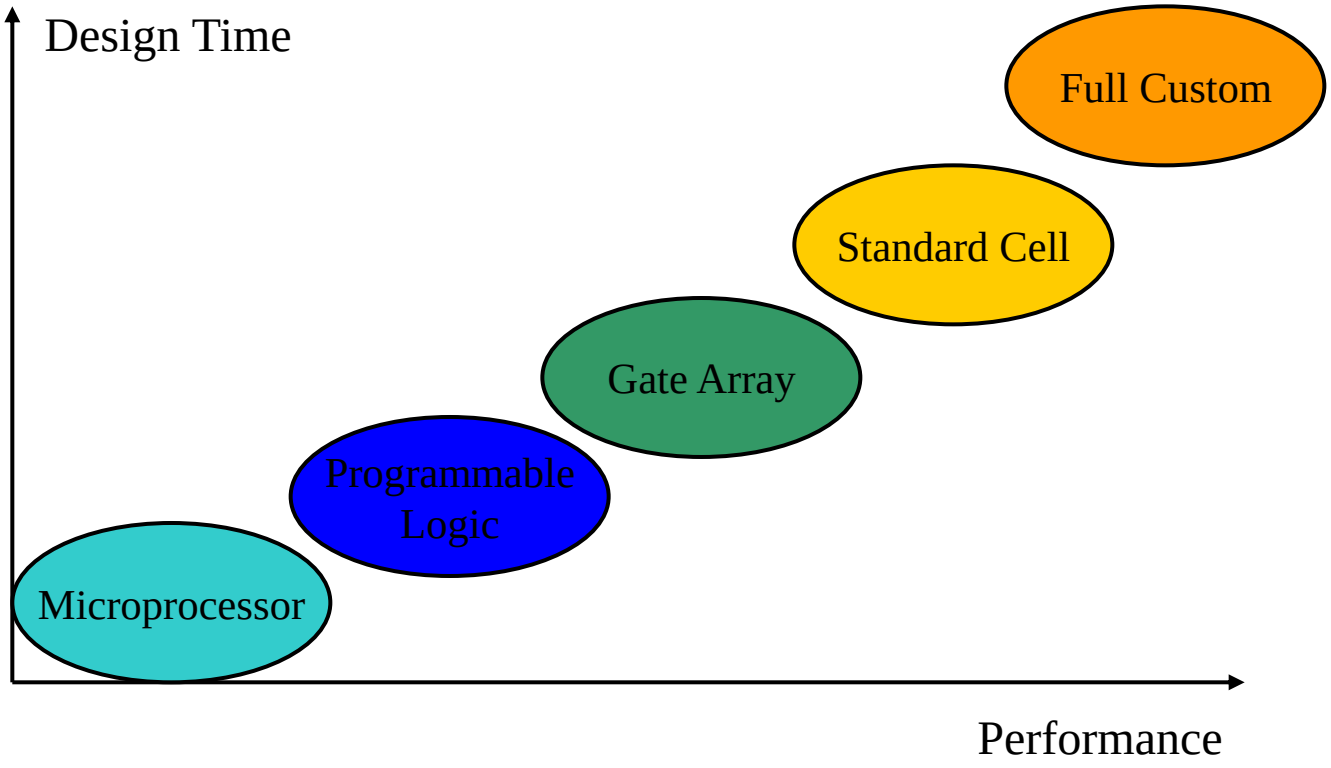


Comparison
FPGA, Gate Array, Standard Cell

	Initial Cost	Cost per part	Performance	Fabrication Time
FPGA	Low	High	Low	Short
Gate Array (ASIC)				
Standard Cell (ASIC)	High	Low	High	Long



Design Trade-Offs

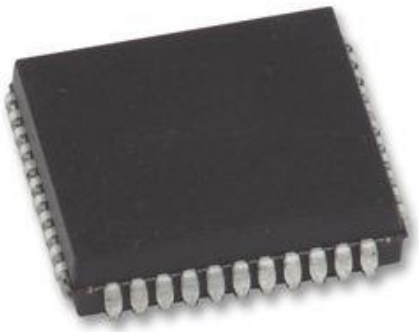


Implementera i programmerbar logik

Vi behöver:

Mjukvara för att ”kompilera” VHDL kod och generara koder för macroceller

Programmerare

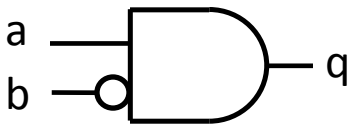


Programmerbar logik

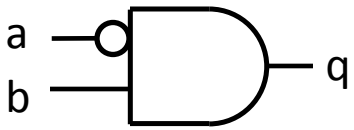
Snabbfråga:Gissa grinden

Vilken logisk grind motsvarar följande VHDL kod?

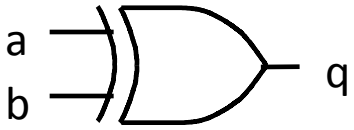
```
q <= a and (not b);
```



Alt: A



Alt: B

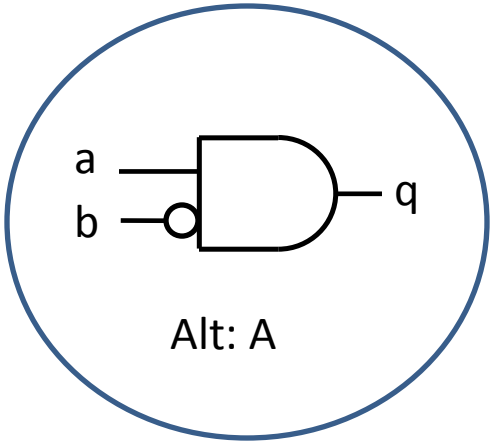


Alt: C

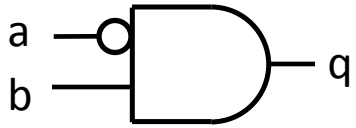
Snabbfråga:Gissa grinden

Vilken logisk grind motsvarar följande VHDL kod?

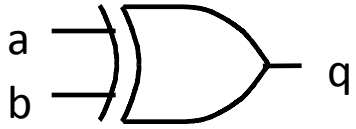
```
q <= a and (not b);
```



Alt: A



Alt: B

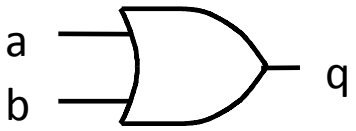


Alt: C

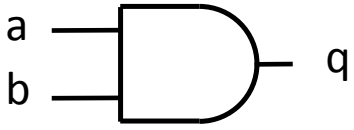
Snabbfråga:Gissa grinden

Vilken logisk grind motsvarar följande VHDL kod?

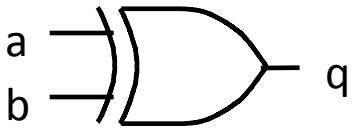
```
if (a /= b) then
  q <= '1';
else
  q <= '0';
end if;
```



Alt: A



Alt: B

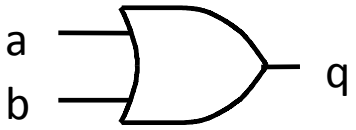


Alt: C

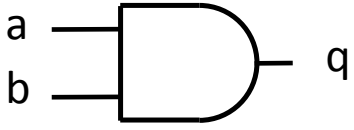
Snabbfråga:Gissa grinden

Vilken logisk grind motsvarar följande VHDL kod?

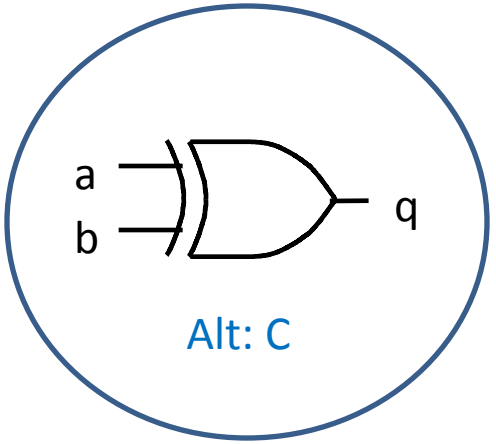
```
if (a /= b) then
  q <= '1';
else
  q <= '0';
end if;
```



Alt: A

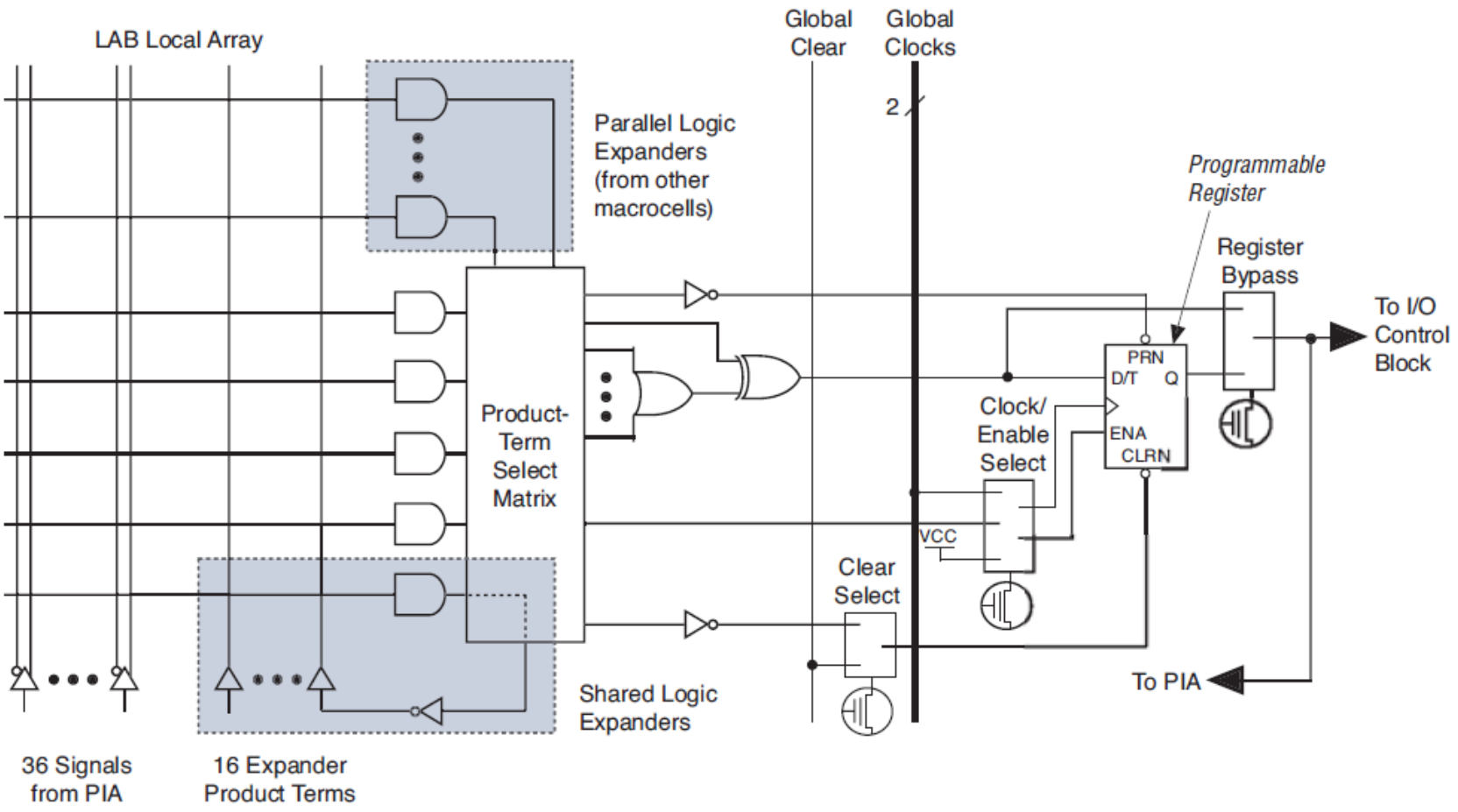


Alt: B

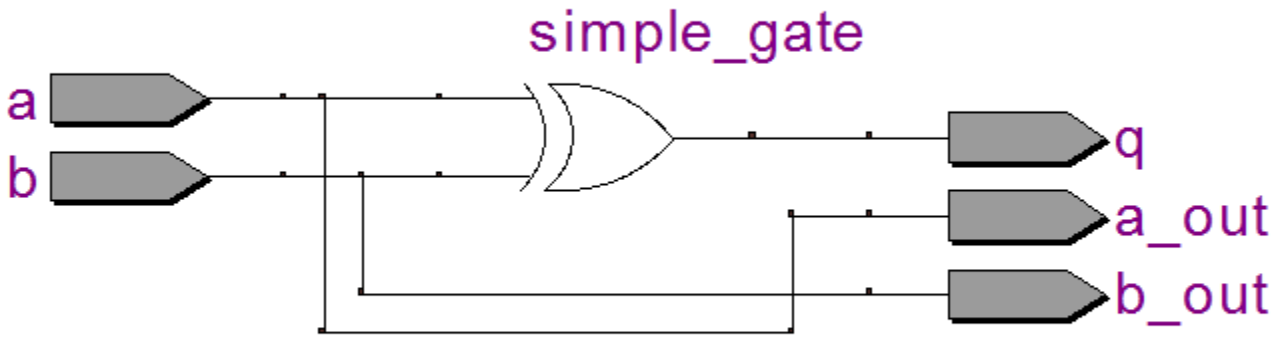


Alt: C

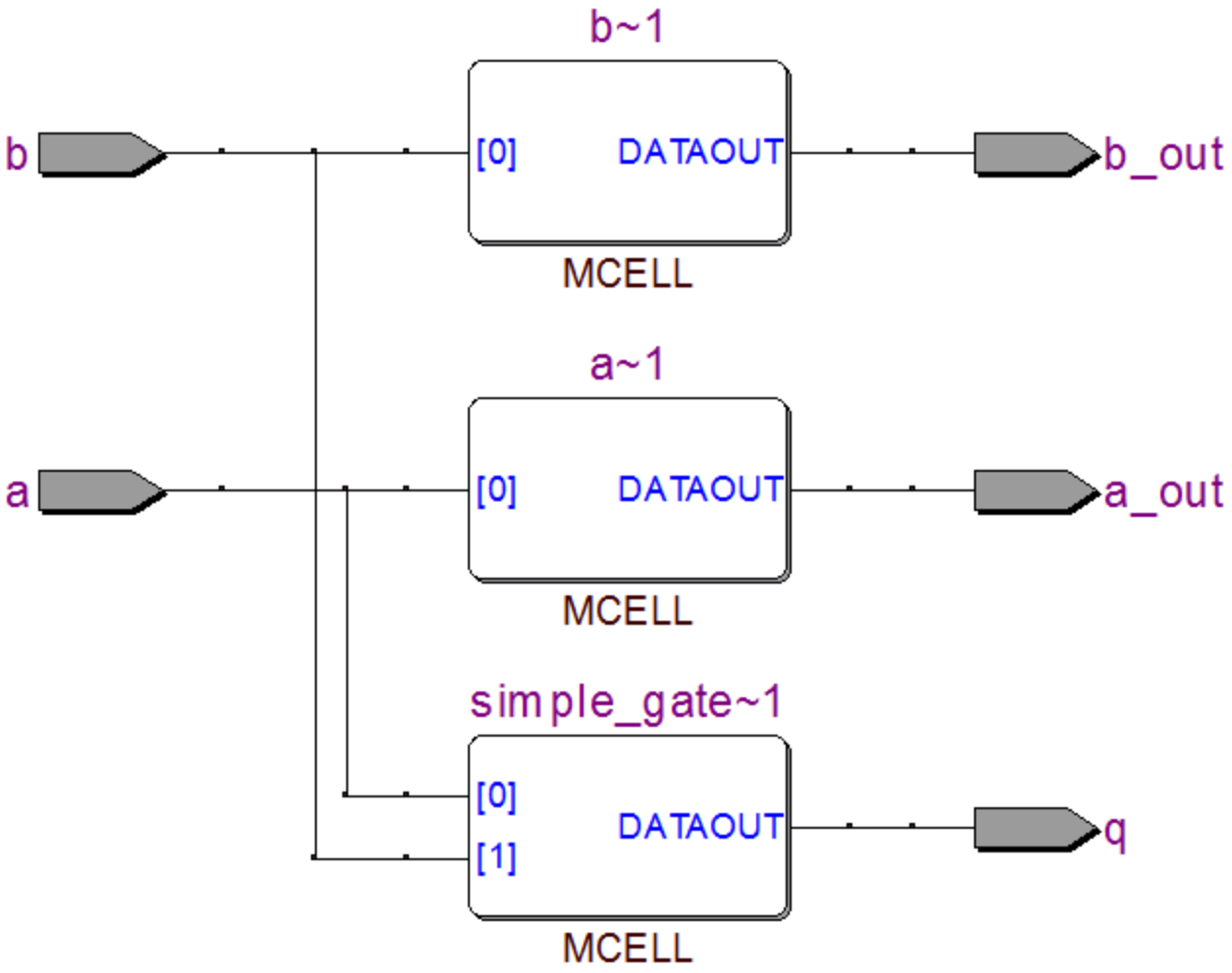
Logik: MAX 3000 macrocell



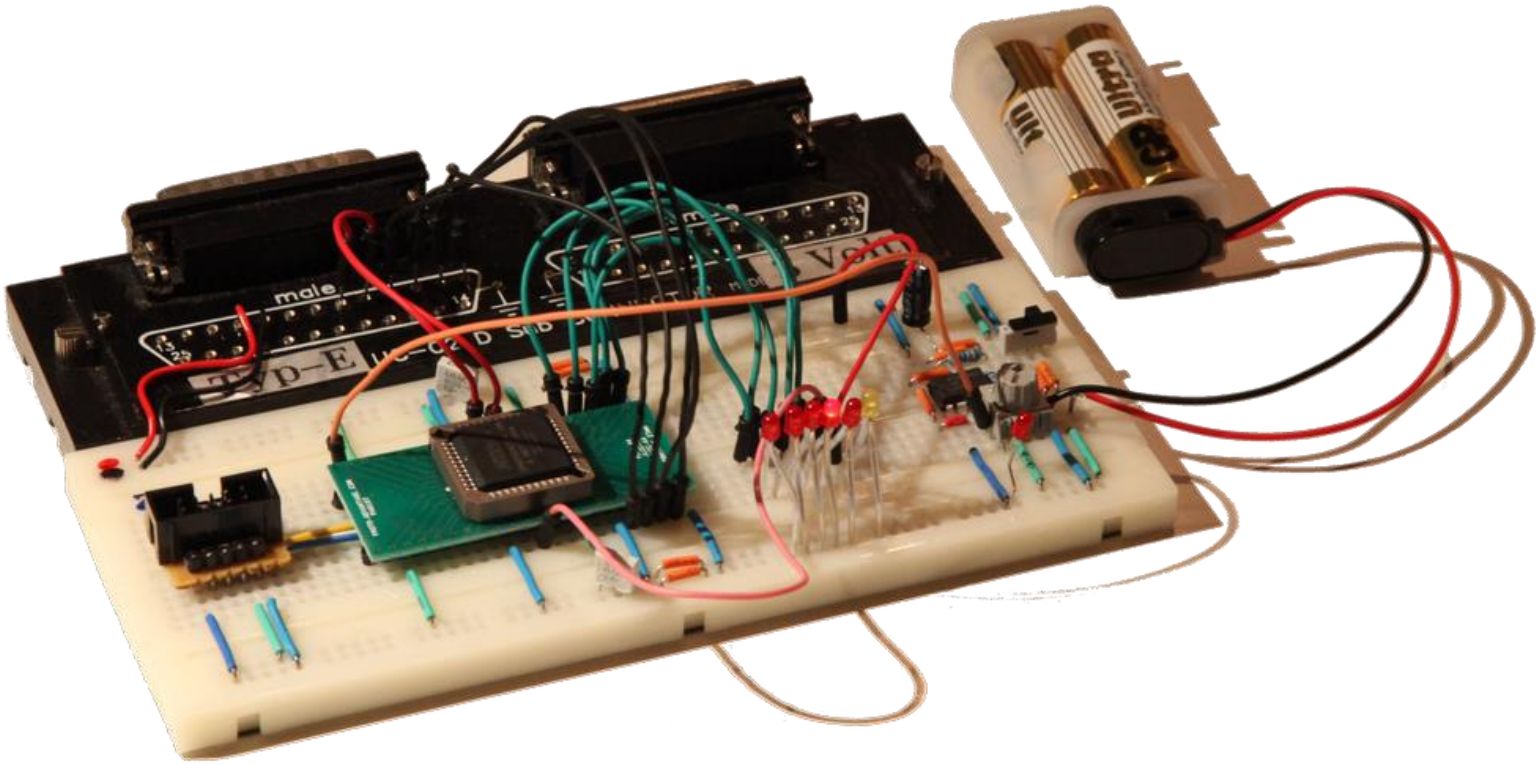
After compilation: RTL view



Mapped to macrocells



Exempel: Ping-Pong



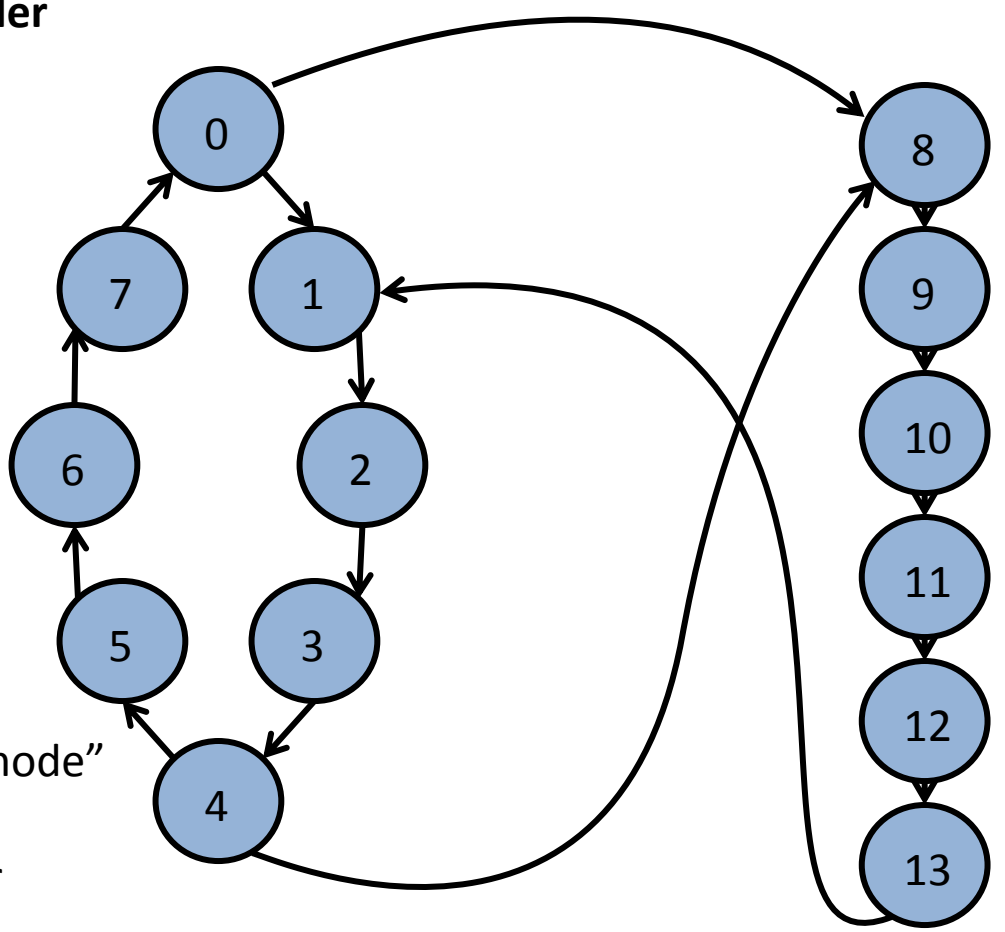
Ping-Pong

Spelas med fem lysdioder och en knapp:

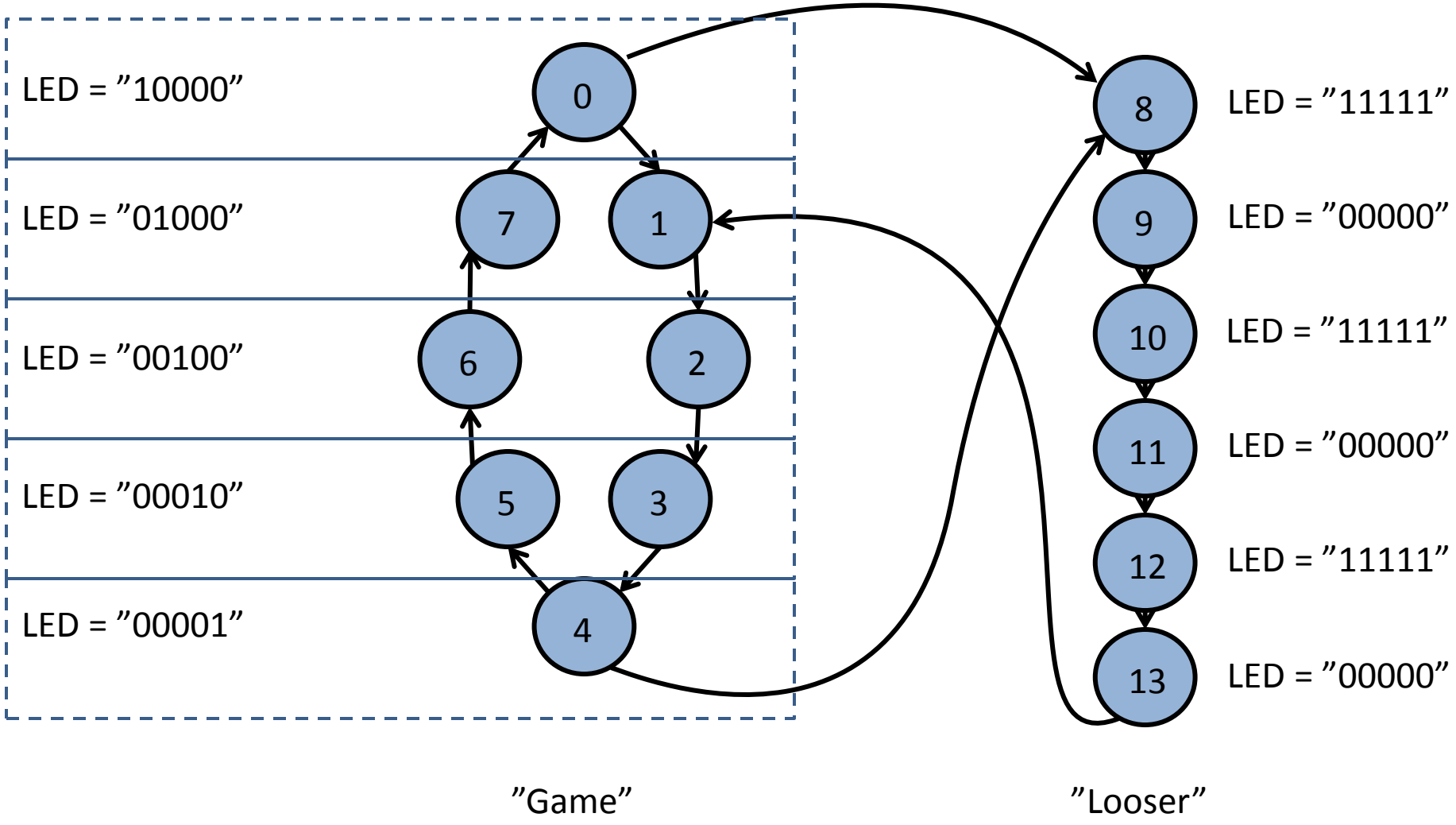
Knappen MÅSTE vara intryckt i state 0 och 4 för att ”bolla tillbaka”

Knapen får INTE vara intryckt i state 1..3 samt 5..7, då slås ”bollen” av bordet

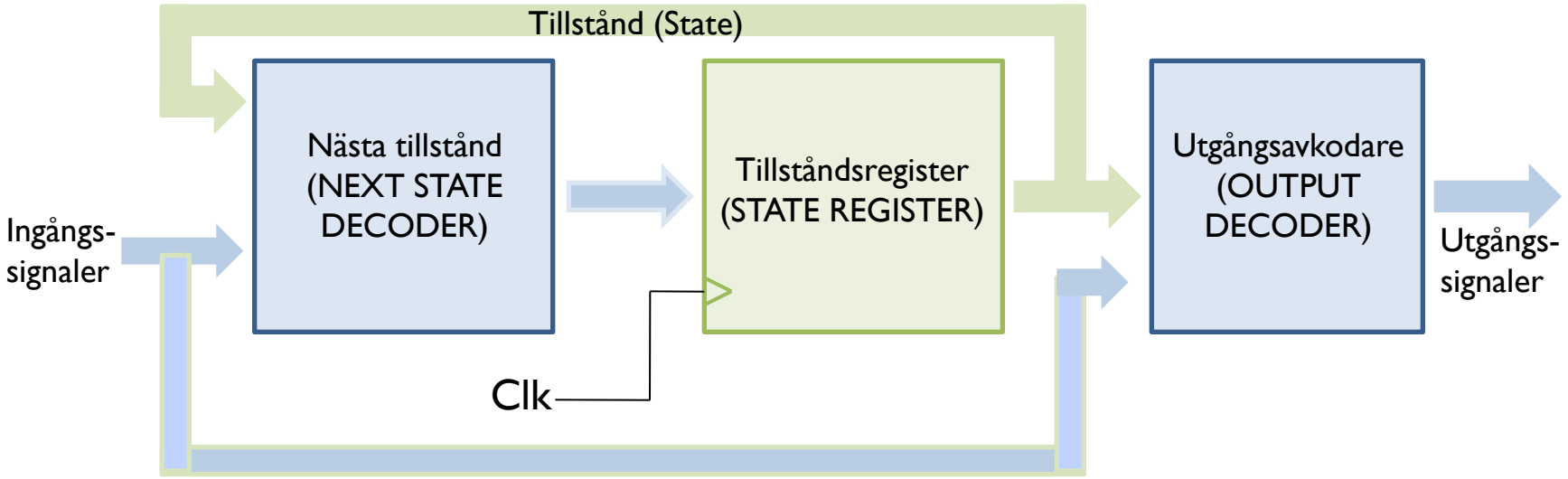
Om spelaren missar hamnar man i ”looser mode” där alla dioder blinkar tre gånger, sedan börjar spelet om på state 1



Ping-Pong: Utgångar

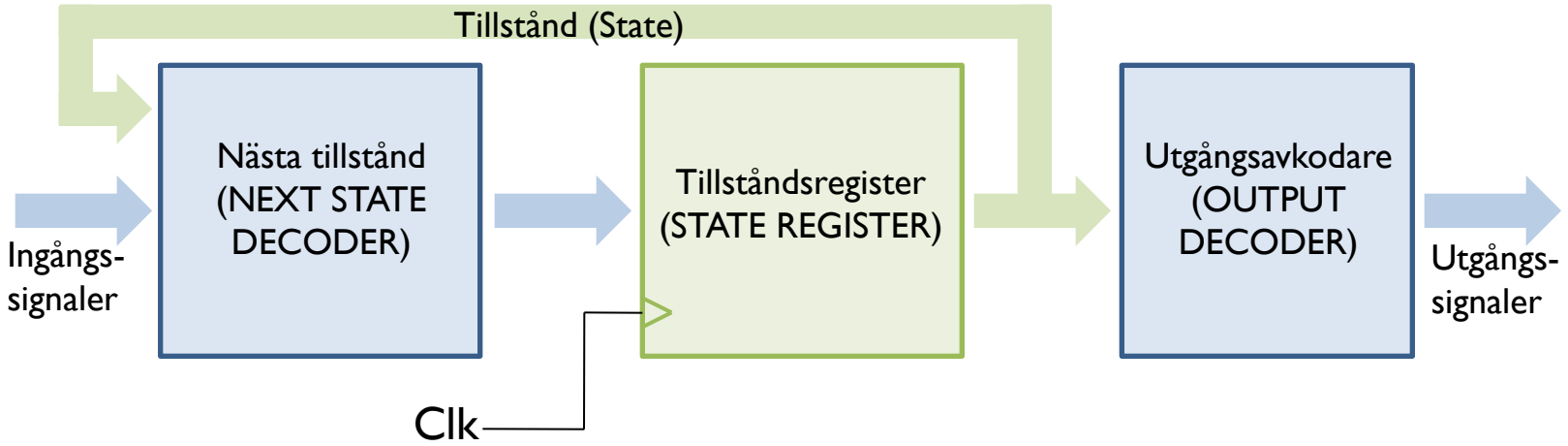


Mealy-Automat



- I en Mealy-Automat beror utgångssignalerna både på nuvarande tillstånd och ingångarna

Moore-Automat



- I en Moore-automat beror utgångssignalerna bara på nuvarande tillstånd

Hur modellerar vi tillståndsmaskiner i VHDL?

- I en Moore-automat har vi tre block
  - Nästa-tillståndsavkodare
  - Utgångsavkodare
  - Tillståndsregister
- Dessa block exekverars parallellt

Processer i VHDL

- En *architecture* i VHDL kan innehåller flera processer
- Processer exekveras parallelt
- En process är skriven som ett sekvensiellt program

Hur modellerar vi tillståndsmaskiner i VHDL?

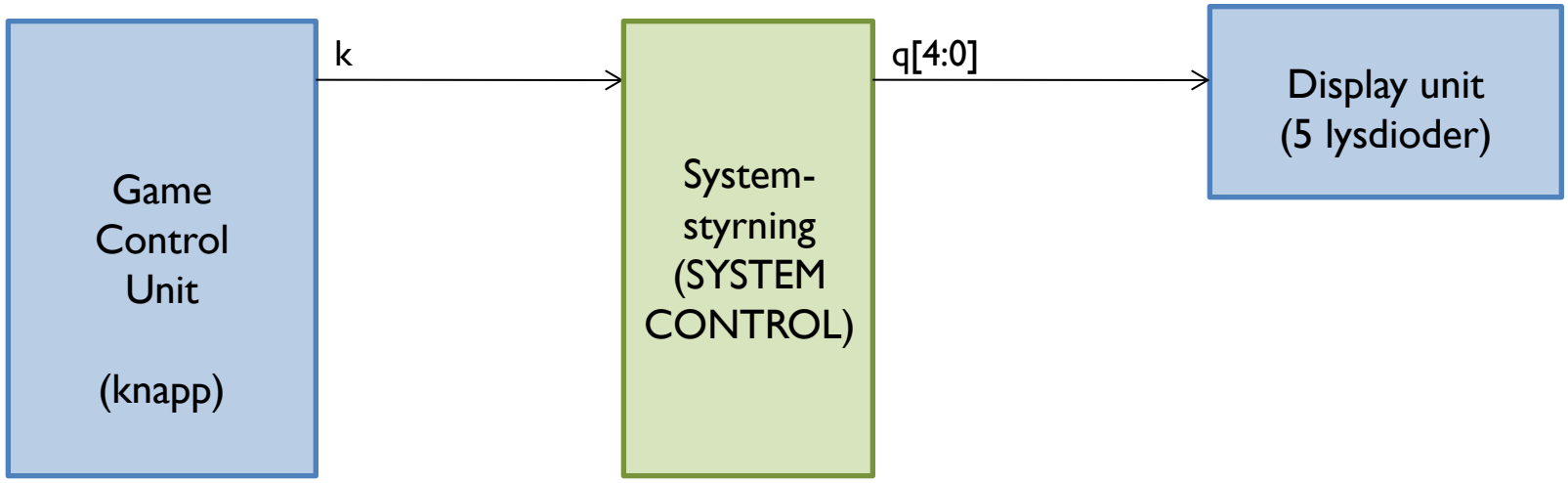
- För en Moore-automat kan vi skapa tre processer för
  - Nästa-tillståndsavkodare
  - Utgångsavkodare
  - Tillståndsregister

Interna Signaler

- Moore-automaten innehåller interna signaler för
  - Nästa tillstånd
  - Nuvarande tillstånd
- Dessa signaler deklareras in *architecture*-beskrivningen

PingPong i VHDL

- Vi använder ett (mycket) enkelt ping-pong spel som konkret exempel
- Vi beskriver systemstyrningen I VHDL



Ping-Pong: entity

```
entity pingpong is
   port( clk: in  std_logic;
         k: in  std_logic;
     reset_n: in  std_logic;
         q: out std_logic_vector(4 downto 0));
end pingpong;
```

- Entityn beskriver systemet som en '*black box*'
- Entityn beskriver gränssnittet mot omvärlden
- Bara in- och utsignaler beskrivs här
- Bortsett från in- och utsignaler som finns i blockdiagrammet behövs det signaler för
  - Clock
  - Reset (aktiv låg)

Ping Pong: Architecture

- Arkitekturen beskriver funktionen av automaten
- Vi definierar
  - interna signaler för nuvarande och nästa tillstånd
  - tre processer för nästa-tillstånds-, utgångsavkodare och tillståndsregister

Ping-Pong: states

- Vi måste skapa en typ för den interna signalen
- Eftersom vi beskriver tillstånden använder vi en integer
- Vi deklarerar en variabel för nuvarande tillstånd (`state`) och en för nästa tillstånd (`next_state`)

```
architecture behavior of pingpong is
subtype state_type is integer range 0 to 13;
signal state, nextstate: state_type;
```

Ping-Pong: states

- Alternativt kan vi beskriver tillstånden med uppräkningstyp, tex med värdena a,b,c,d,e,f,g
- (Exempel states från flaskautomat)

```
ARCHITECTURE Moore_FSM OF Vending_Machine IS
   TYPE   state_type IS (a, b, c, d, e, f, g);
   SIGNAL current_state, next_state      : state_type;
BEGIN  -- Moore_FSM
```

…

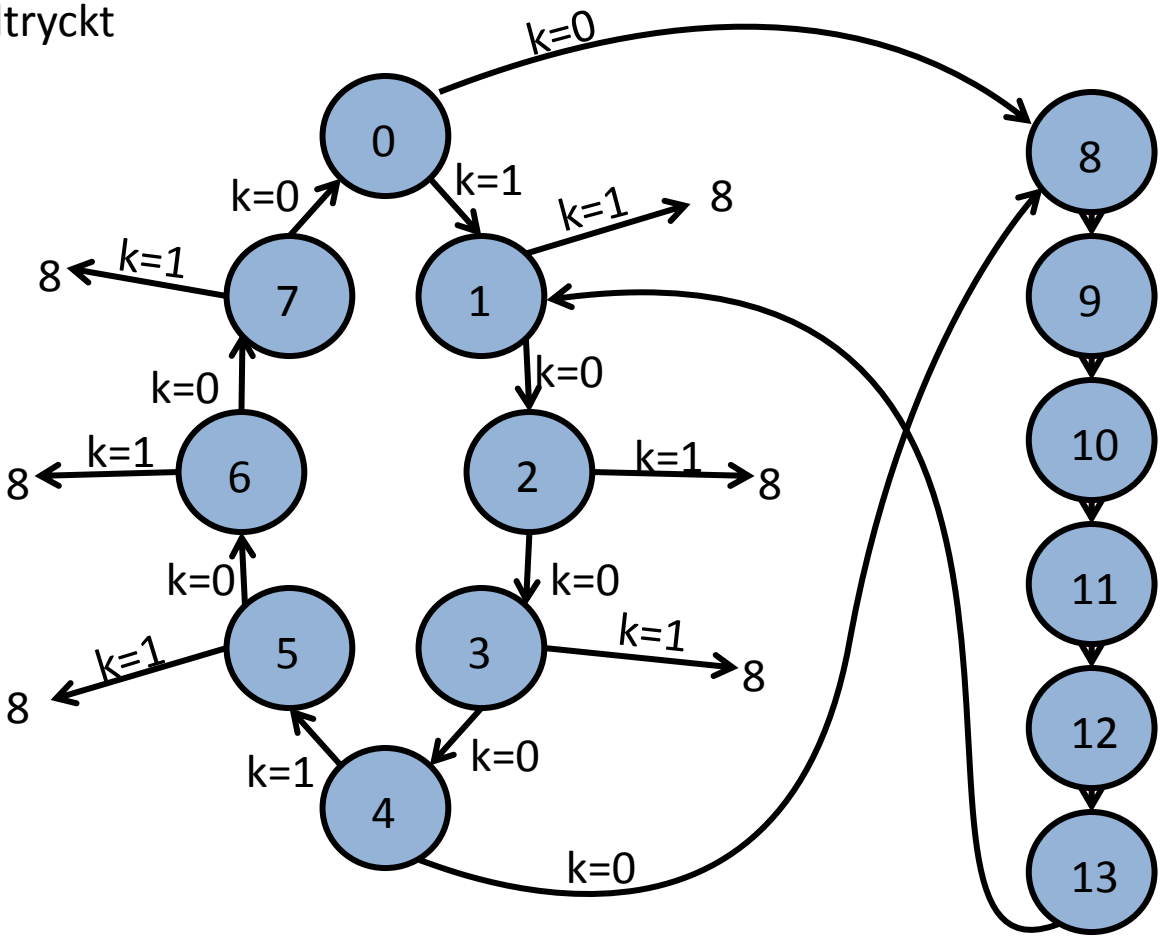
Interna Signaler

- Om vi inte specificerar tillståndskodningen så väljer syntesverktyget kodningen
- Vi kan tvinga den till en viss kodning med attributer (**OBS! Attributer är beroende på syntesverktyget och därmed inte portabel!)**

```
ARCHITECTURE Moore_FSM OF Vending_Machine IS
    TYPE   state_type IS (a, b, c, d, e, f, g);
    -- We can use state encoding according to BV 8.4.6
    -- to enforce a particular encoding (for Quartus)
    ATTRIBUTE enum_encoding : string;
    ATTRIBUTE enum_encoding OF state_type : TYPE IS "000
    001 011 110 111 100 101";
    SIGNAL current_state, next_state      : state_type;
BEGIN  -- Moore_FSM
…
```

Ping-Pong: Next state

k=0 : Knapp ej nedtryckt
k=1 : Knapp nedtryckt



Flaskautomat: Next-State-Decoder

- Next-State-Decoder beskrivs som process
- Sensitivity list innehåller alla insignaler som ‘aktiverar’ processen

Ping-Pong: States

```
begin nextstate_decoder:
process(state, k)
 begin
   case state is
      when 0 =>
        if (k = '1') then
          nextstate <= 1; -- Continue if key pressed
        else
          nextstate <= 8; -- Fail if key not pressed
        end if;
      when 4 =>
        if (k = '1') then
          nextstate <= 5; -- Continue if key pressed
        else
          nextstate <= 8; -- Fail if key not pressed
        end if;
```

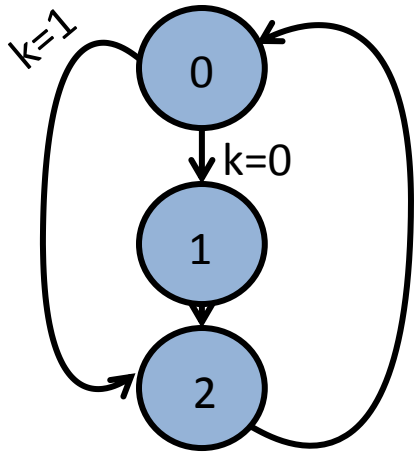
Ping-Pong: States cont

```
        when 7 => nextstate <= 0; -- Loop to beginning
        when 8 to 12 => nextstate <= state + 1; -- Looser mode
        when 13 => nextstate <= 1; -- End of looser mode,
                                   -- loop to play mode
        when others => -- Play mode
          if(k = '1') then
             nextstate <= 8; -- fail IF key pressed
          else
            nextstate <= state + 1; -- otherwise continue
          end if;
    end case;
end process;
```

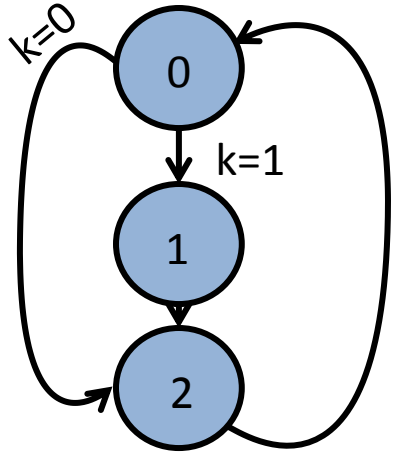
Snabbfråga:Gissa grinden

Vilken statemaskin motsvarar följande VHDL kod?

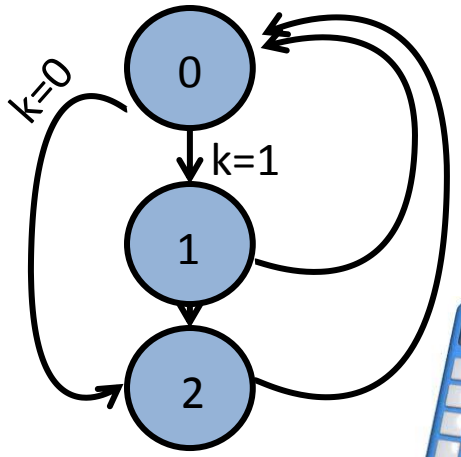
```
case state is
  when 0 =>
    if (k = '1') then
      nextstate <= 1;
    else
      nextstate <= 2;
    end if;
  when 1 => nextstate <= 2;
  when others => nextstate <= 0;
end case;
```



Alt: A



Alt: B

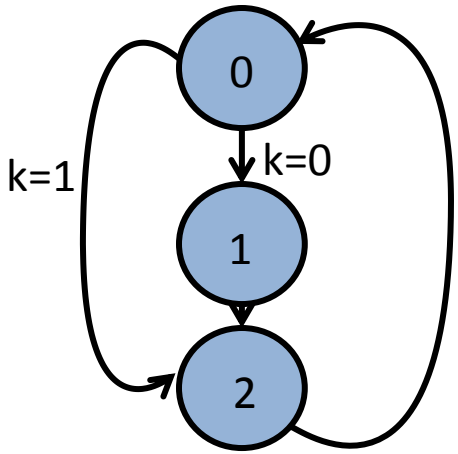


Alt: C

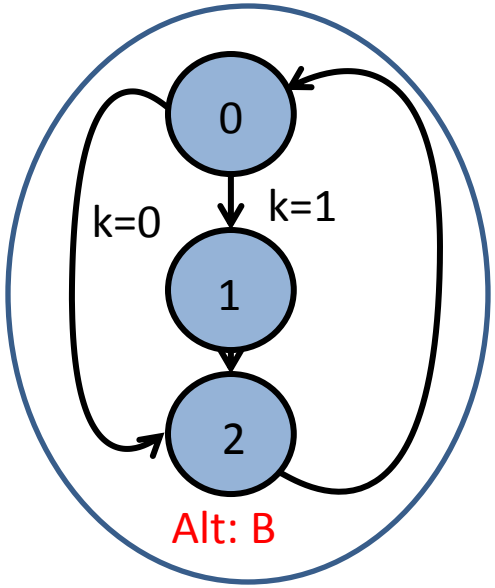
Snabbfråga:Gissa grinden

Vilken statemaskin motsvarar följande VHDL kod?

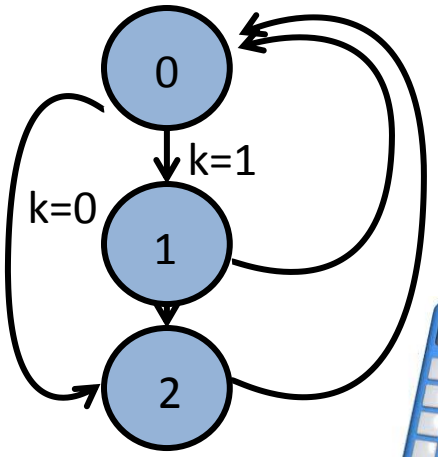
```
case state is
  when 0 =>
    if (k = '1') then
      nextstate <= 1;
    else
      nextstate <= 2;
    end if;
  when 1 => nextstate <= 2;
  when others => nextstate <= 0;
end case;
```



Alt: A

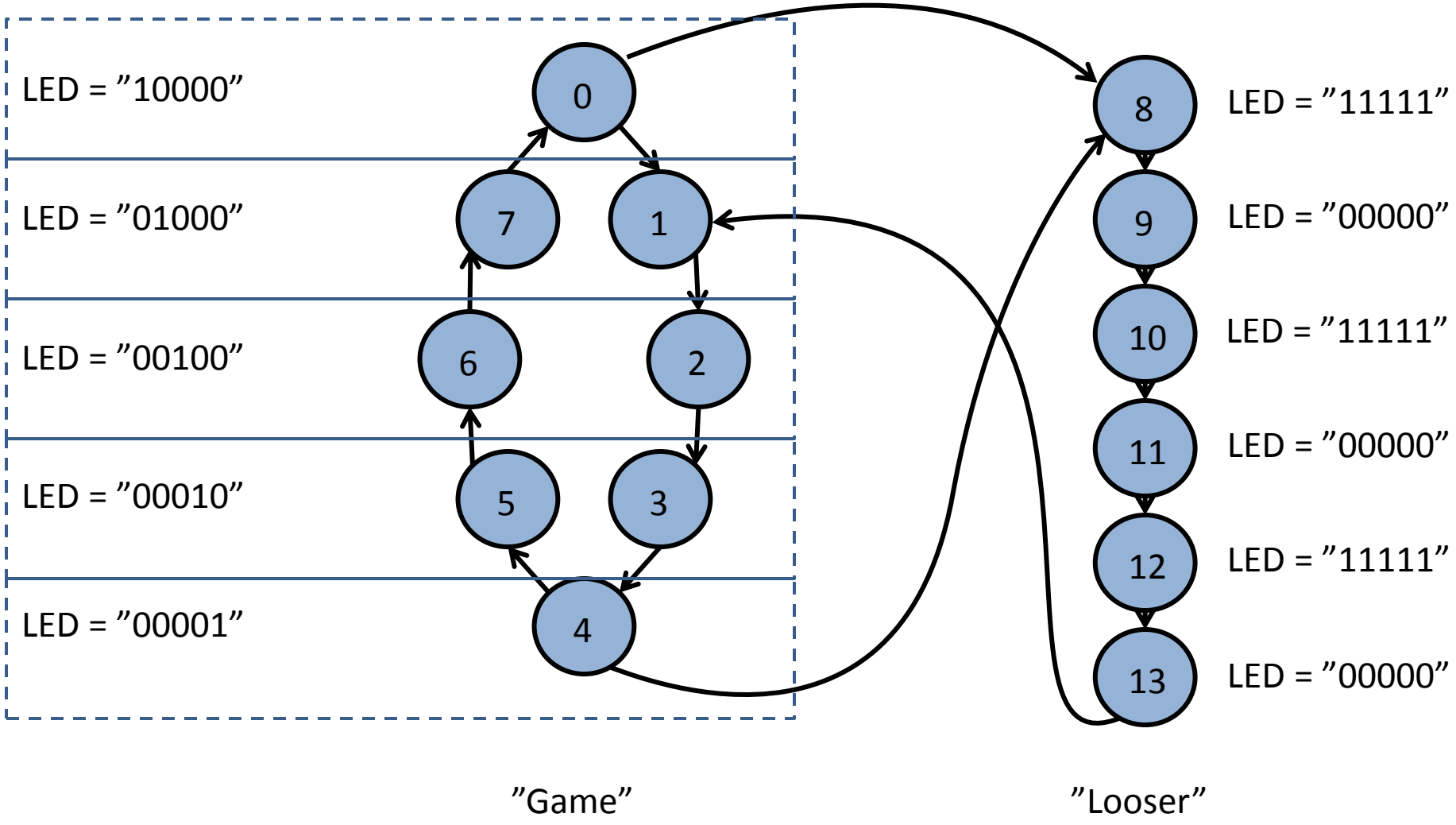


Alt: B



Alt: C

Ping-Pong: Utgångar



Flaskautomat: Utgångsavkodaren

- Utgångsavkodaren beskrivs som en egen process
- Sensitivity-listan innehåller bara nuvarande tillstånd eftersom utgångarna är direkt beroende på tillståndet

Ping-Pong: Outputs

```
output_decoder: -- output decoder part
process(state)
begin
  case state is
     when 0 => q(4 downto 0) <= "10000";
     when 1 => q(4 downto 0) <= "01000";
     when 2 => q(4 downto 0) <= "00100";
     when 3 => q(4 downto 0) <= "00010";
     when 4 => q(4 downto 0) <= "00001";
     when 5 => q(4 downto 0) <= "00010";
     when 6 => q(4 downto 0) <= "00100";
     when 7 => q(4 downto 0) <= "01000";
     when 8 | 10 | 12 => q(4 downto 0) <= "11111";
     when 9 | 11 | 13 => q(4 downto 0) <= "00000";
  end case;
end process;
```

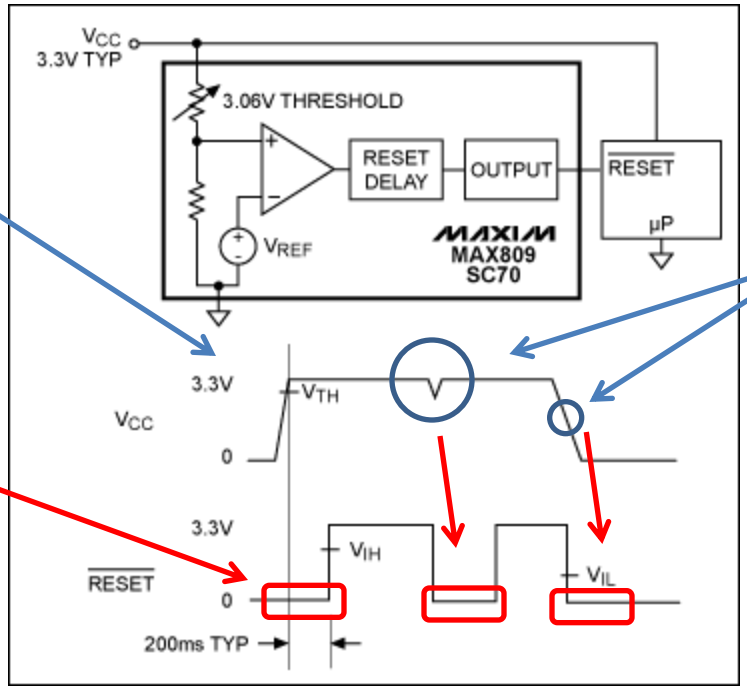
Tillstånds register

- Tillståndsregistret modelleras som en synkron process med asynkron reset (aktiv låg)

```
state_register:
process(clk, reset_n)
begin
  if reset_n = '0' then
     state <= 0;
  elseif rising_edge(clk) then
     state <= nextstate;
  end if;
end process;
```

(RESET-generator chip)

Matnings-spänning
”på”
ger **RESET** i 200 ms



Om matnings-spänningen får problem, eller sjunker under viss nivå, så blir det **RESET**

Bättre än att behöva skaffa extra skydd, är att designa förebyggande och från början ”ta hand om” alla tillstånd …

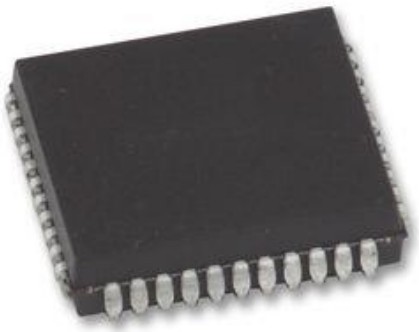
William Sandqvist william@kth.se

Implementera i programmerbar logik

Vi behöver:

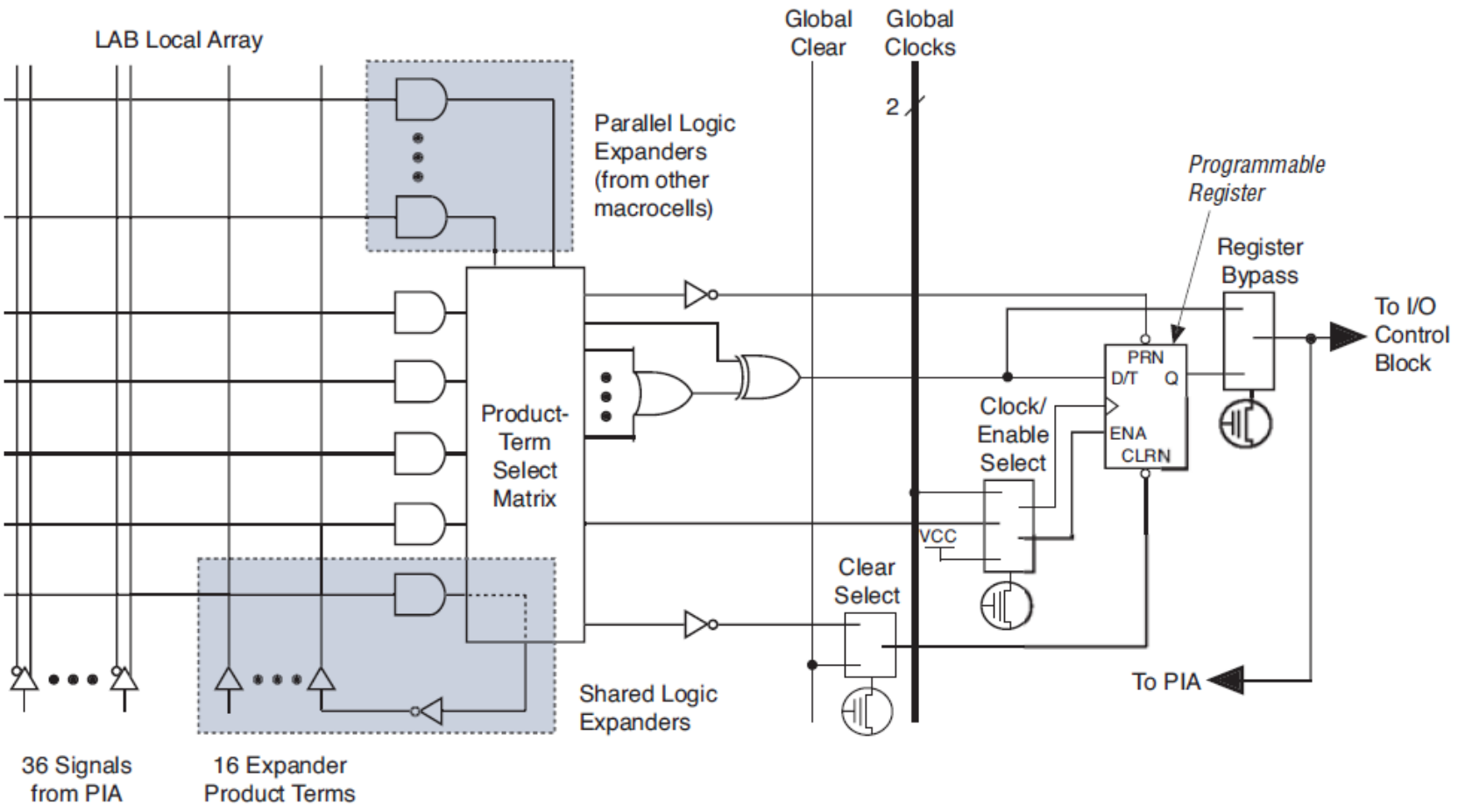
Mjukvara för att ”kompilera” VHDL kod och generara koder för macroceller

Programmerare

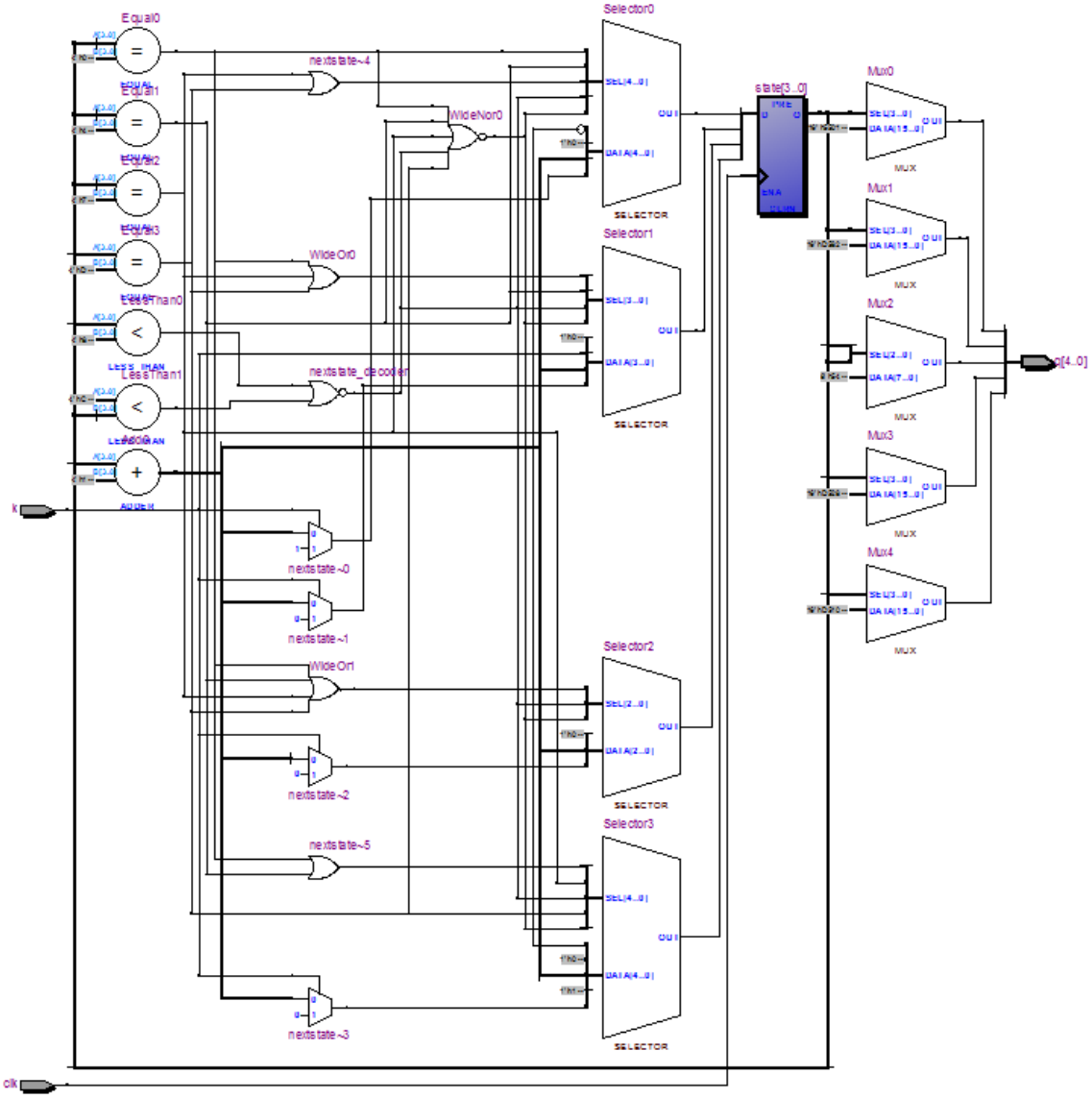


Programmerbar logik

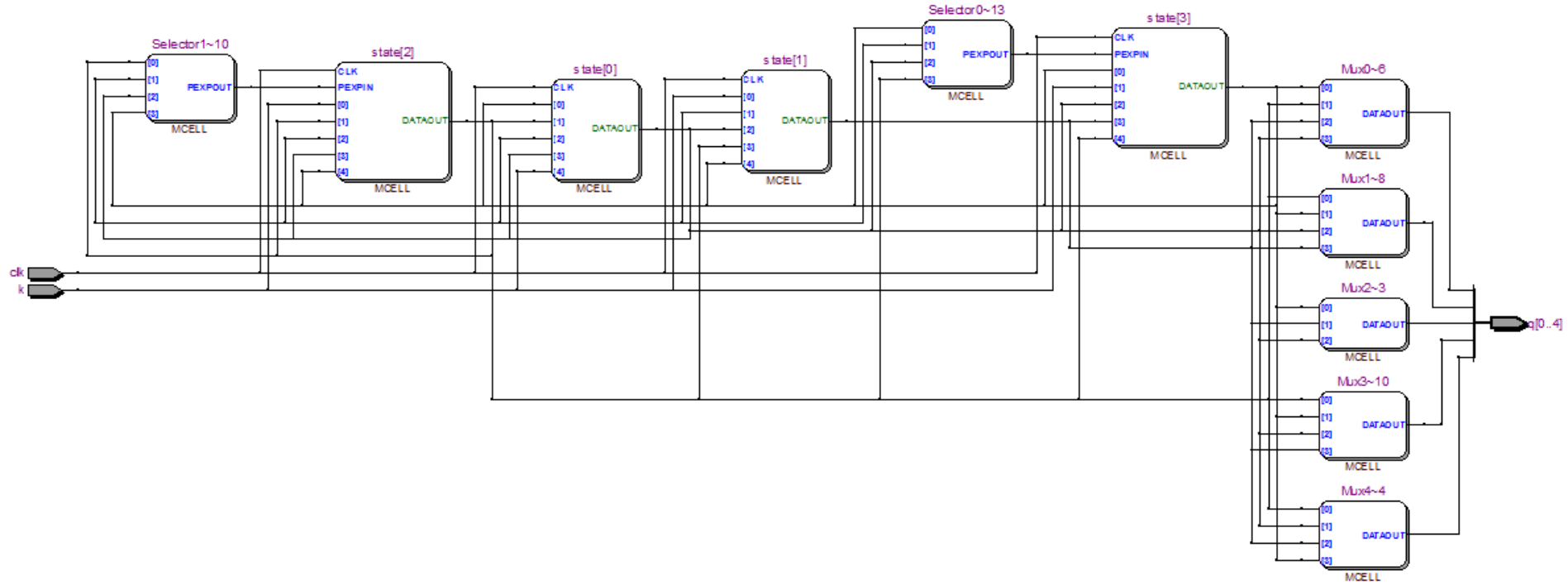
Logik: MAX 3000 macrocell



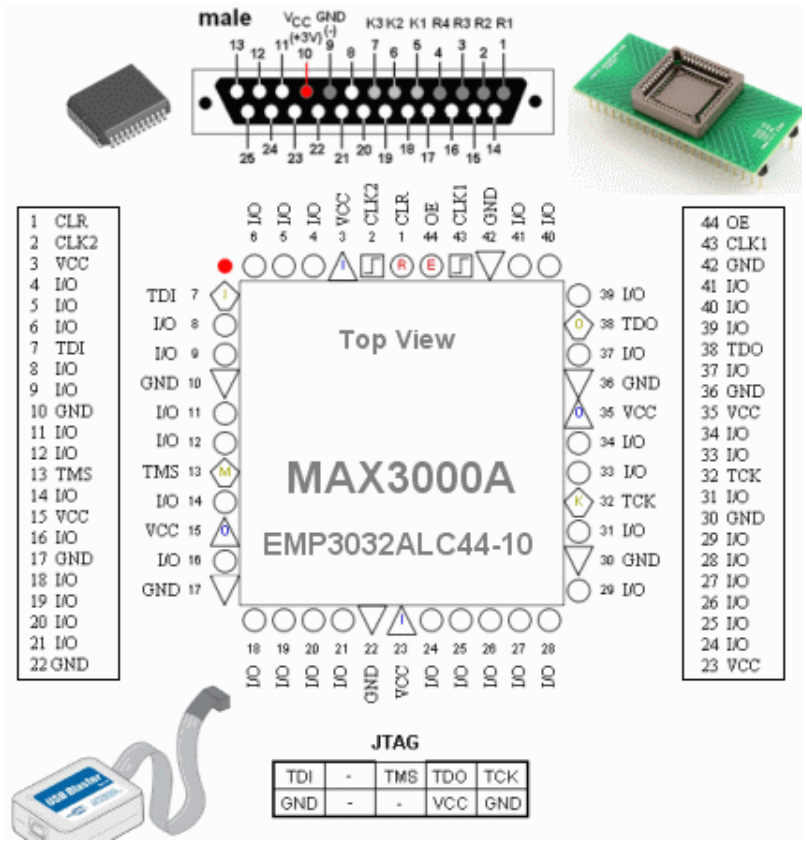
After compilation: RTL view

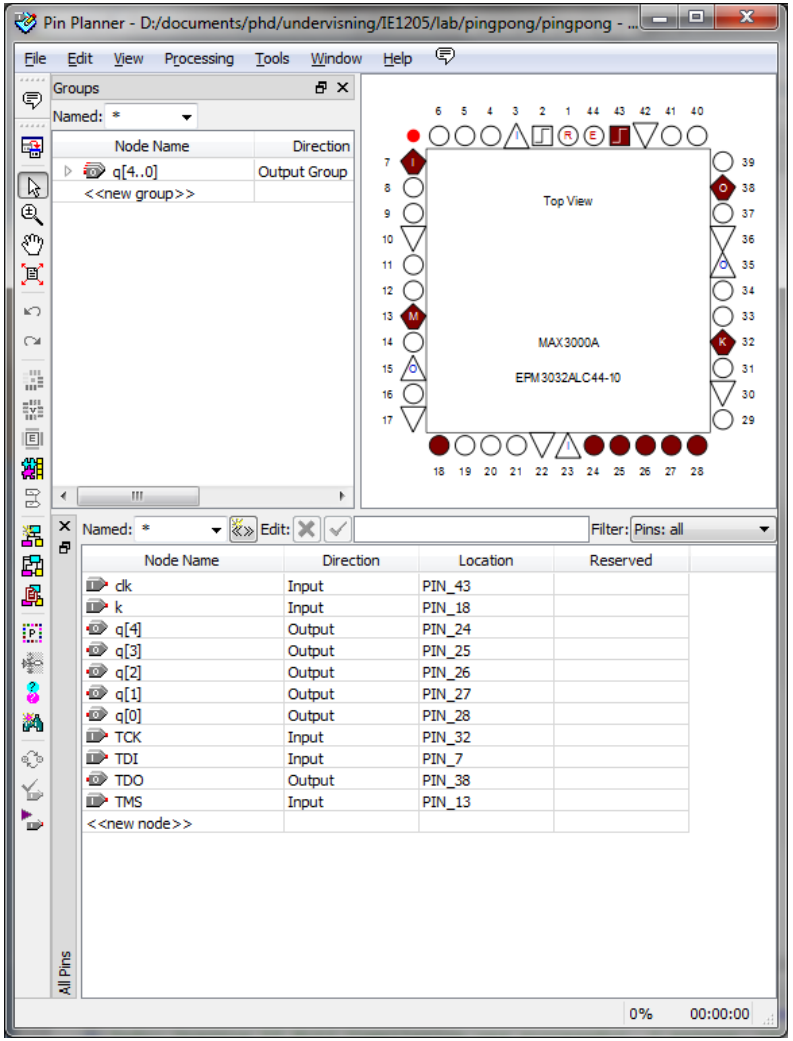


Mapped to macrocells

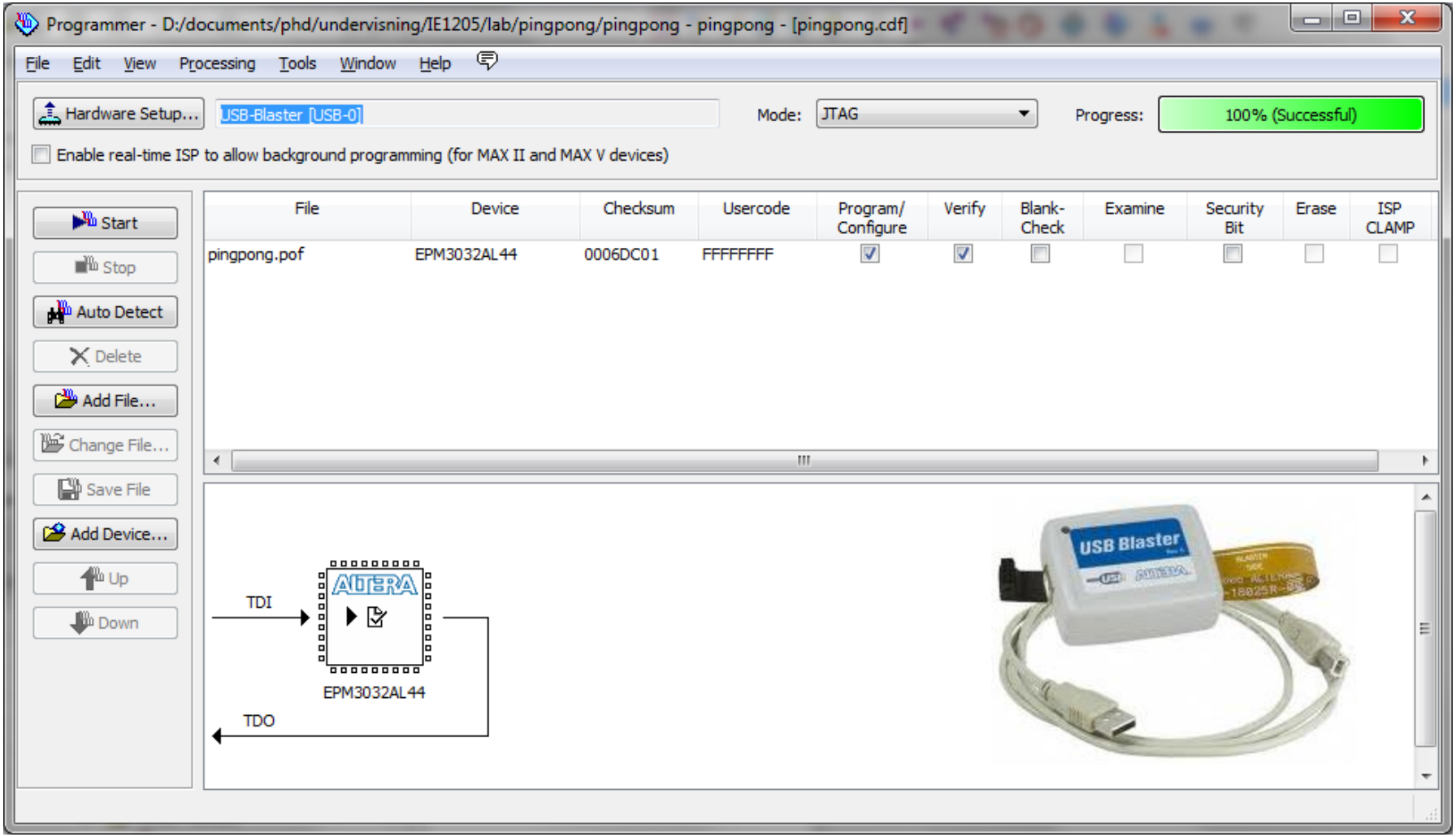


Pin mapping


male
Vcc GND (+3V) (-)
K3 K2 K1 R4 R3 R2 R1
13 12 11 10 9 8 7 6 5 4 3 2 1
25 24 23 22 21 20 19 18 17 16 15 14
1 CLR
2 CLK2
3 VCC
4 I/O
5 I/O
6 I/O
7 TDI
8 I/O
9 I/O
10 GND
11 I/O
12 I/O
13 TMS
14 I/O
15 VCC
16 I/O
17 GND
18 I/O
19 I/O
20 I/O
21 I/O
22 GND
I/O I/O I/O VCC CLK2 CLR OE CLK1 GND I/O I/O
8 5 4 3 2 1 44 43 42 41 40
TDI 7
I/O 8
I/O 9
GND 10
I/O 11
I/O 12
TMS 13
I/O 14
VCC 15
I/O 16
GND 17
Top View
MAX3000A
EMP3032ALC44-10
39 I/O
38 TDO
37 I/O
36 GND
35 VCC
34 I/O
33 I/O
32 TCK
31 I/O
30 GND
29 I/O
18 19 20 21 22 23 24 25 26 27 28
I/O I/O I/O I/O GND VCC I/O I/O I/O I/O I/O
44 OE
43 CLK1
42 GND
41 I/O
40 I/O
39 I/O
38 TDO
37 I/O
36 GND
35 VCC
34 I/O
33 I/O
32 TCK
31 I/O
30 GND
29 I/O
28 I/O
27 I/O
26 I/O
25 I/O
24 I/O
23 VCC
JTAG
TDI - TMS TDO TCK
GND - - VCC GND


Pin Planner - D:/documents/phd/undervisning/IE1205/lab/pingpong/pingpong - ...
File Edit View Processing Tools Window Help
Groups
Named: *
Node Name Direction
q[4..0] Output Group
<<new group>>
Top View
MAX3000A
EPM3032ALC44-10
Named: * Edit: Filter: Pins: all
Node Name Direction Location Reserved
clk Input PIN_43
k Input PIN_18
q[4] Output PIN_24
q[3] Output PIN_25
q[2] Output PIN_26
q[1] Output PIN_27
q[0] Output PIN_28
TCK Input PIN_32
TDI Input PIN_7
TDO Output PIN_38
TMS Input PIN_13
<<new node>>
All Pins
0% 00:00:00

Programming



Mealy-automat i VHDL

- En Mealy-automat kan modelleras på samma sätt som Moore-automaten
- Skillnaden är att utgångsavkodaren också är beroende på insignalerna
- Processen som modellerar utgångssignalerna behöver också ha insignalerna i sensitivity list!

Mer om VHDL

- Kodexemplet för flaskautomaten finns på kurshemsidan
- Titta på studiematerialet om “VHDL-syntes” på kurshemsidan
- Både Brown/Vranesic- och Hemert-boken innehåller kodexempel

Sammanfattning

- PLD och FPGA
- Statemaskin i VHDL