SELECTED EXAMPLES FROM THE EXAMS

VERSION 2.1 (7 JANUARY 2024)

- Examples were selected from the previous LSP written tests; to illustrate what students can expect on the exams;
- Exams written tests can contain other examples, and of course, they will always have different input values;
- The list does not present theoretical questions, i.e., questions that expect a verbal answer, such as the definitions of the Moore and Mealy automaton;
- If some example does not include its solution, you can check your idea by simulation or look at the lectures;
- The labeling of examples is a teacher's internal reference, and it is not related to the complexity or type of problem.
- The text may, unfortunately, contain unintentional typos.
  - However, these should be pointed out before the exam, when it will be too late.

Consider the VHDL declaration: **signal X, Y, S : signed(9 downto 0);**
If the decimal value of X is 500 and the decimal value of Y = 400, what is the sum S=X+Y ? Write down the resulting value of S as the decimal number S=

The binary adder has the same structure for signed and unsigned numbers, only its result is interpreted differently. If we add 400+500, then in unsigned arithmetic, we get the result 900, but interpreted as a 10-bit signed number. And we can convert 900 to signed in three ways.

A. Binary path
We convert the numbers 400 and 500 to 10-bit binary numbers, then we add them and convert the result back.

```
          400 = 01 1001 0000
          500 = 01 1111 0100
      400+500 = 11 1000 0100
1st complement = 00 0111 1011
    2. not+1 = 00 0111 1011
               +00 0000 0001
          124 = 00 0111 1100
```

the result has a 1 in the highest bit, so it's a negative number whose **absolute** value can be found using twos complement

How do we convert 900 to signed? Either by brute force or by reasoning. If the number had ones in its lower bits, i.e., it was `00 0111 1111`, it would be 2^7-1 = 127, so the binary `00 0111 1100` is 127-3

the result of adding 400+500 in 10-bit signed arithmetic is **-124**

B. The Way of the Ferryman
We convert 900 result to a binary number and find the absolute value of the negative number using the second complement. For example, 900 by dividing by 2 is

900 / 2 - remainder after division	0	LSB
450 / 2 - remainder after division	0	
225 / 2 - remainder after division	1	
112 / 2 - remainder after division	0	
56 / 2 - remainder after division	0	
28 / 2 - remainder after division	0	
14 / 2 - remainder after division	0	
7 / 2 - remainder after division	1	
3 / 2 - remainder after division	1	
1 / 2 - remainder after division	1	MSB

So, 400+500 = **`11 1000 0100 ,`**
which we convert using the same procedure as in A, which gives us **-124**

C. Accountant's rule of three for converting 900
The 10-bit signed arithmetic has a range of displayable numbers from -512 to 511, i.e., from `10 0000 0000` to `01 1111 1111` . We know that 511+1, i.e., `10 0000 0000,` is displayed as -512, and 11 1111 1111 is displayed as -1.

In unsigned arithmetic, 11 1111 1111 would have the value 1023. Now we have everything we need for the rule of three with an inverse proportion:

unsigned 512 .....is ..... -512 signed
unsigned 1023 is -1 signed

The 1023-512 = |-512-(-1)|, the same sizes are the property of numbers and simplifies the solution: How much will unsigned 900 be as signed ? 1023-900 = 123 -1 - 123 = **-124**

More similar examples for you to solve

Let's have 16-bit signed arithmetic, using the negative number representation in twos complement, VHDL type signed(15 downto 0), find

a) the number X, which when added to the number Y=70CA (written hexadecimal) gives 0, i.e. X+Y=0 (write the number X again hexadecimal) X= .....

b) the number X, which when added to the decadic number Y=100 gives 10, i.e., X+Y=10 write the number X again in hexadecimal) X =.....

The answer to the question must contain only hexadecimal digits without any characters!

Let's have 8-bit signed arithmetic, using the negative number representation in the second complement, find the VHDL type signed(7 downto 0)

a) the number X, which when added to the number Y=9A (written in hexadecimal) gives 0, i.e. X+Y=0 (write the number X again in hexadecimal) X=.....

b) a number which when added to the decimal number Y=50 gives -1, i.e. X+Y=-1 (write the number X again in hexadecimal) X =.....

The answer to a question may only contain hexadecimal digits without signs!

Let's have **14-bit** signed arithmetic, using the expression of negative numbers in the second complement, VHDL type signed(13 downto 0) Find the number X that when added to the number Y=1234 (written in decade) gives 0, i.e. X+Y=0 Write the number X in hexadecimal: X=3B2E.....

The answer to the question may only contain hexadecimal digits without signs!

Results

(a) X= 8F36.....

(b) X =FFA6.....

(a) X=66.....

(b) X=0CD.....

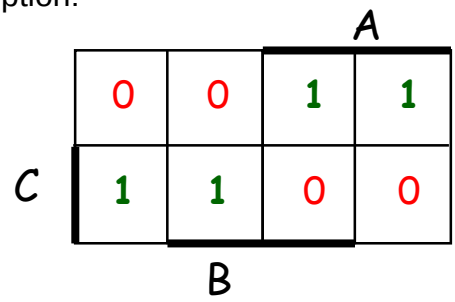
X=3B2E..... - the result must be a 14 bit number!!!.....

Which of these functions are the same?

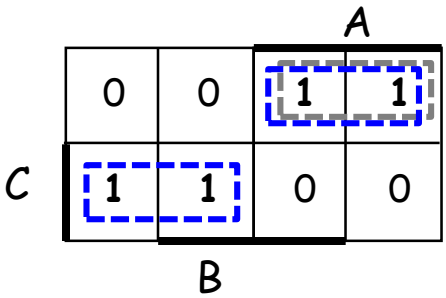
x1<=(A and not C) or (C and not A);
x2<=(A and not B) xor (A and C);
x3<=(A or C) and (not A or not C);
x4<=(A xor C) or **(A and not C)**;

You can solve it by building a logical table from the values, but it is a lengthy operation, and unnecessary here, because there is a much simpler option:

x1 and x3 - can be *directly built as maps, because they are in the forms P-o-S and S-o-P resulting from K-map coverage. So we just perform the inverse operation - we construct maps from the expressions. The function x1 is the S-o-P coverage of ones, and x3 is the P-o-S coverage of zeros.*

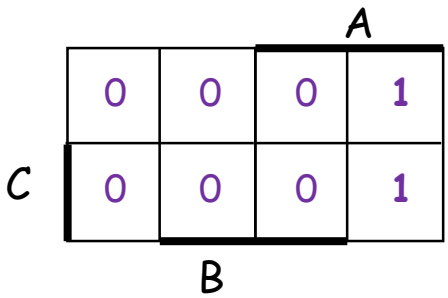


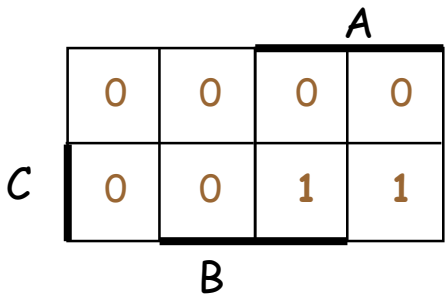
X4 *is S-of-P coverage of ones, except for the* ***xor*** *operation,* ***but*** *we can write that graphically too,*
it will only have a logical 1 in fields where its operands A and C are different because xor is a non-equivalence. We see that the implicant **(A and not C)** *is redundant in the expression.*



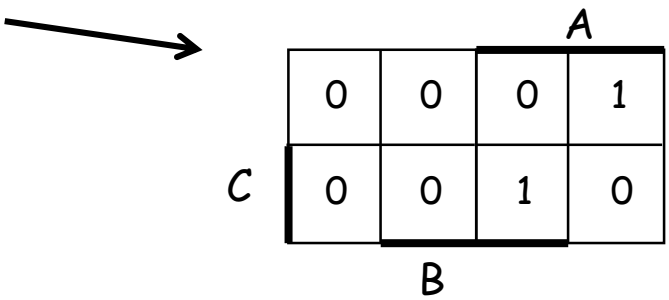
X2 also contains an XOR operation, but with more complex members. Here we will help by constructing a K-map of both members of the XOR operation.

x2<=(A and not B) xor (A and C);





Then we just perform the logical operation xor of the both K-maps, in other words, we write 1 in only those fields where the logical values in both maps differ



C4. Check all the logical functions that have another function identical to them:

f1<=(((not B and not A) or (A and D)) and C) or (B and A); f1 ☐
f2<=((((not A or D) and C) or A) and B); f2 ☐
f3<=((C and (not A or D)) or B) and A and not B; f3 ☐
f4<=((not A or D) and C and not B) or (B and A); f4 ☐
f5<=(A and not B) or (A and D) or (B and A); f5 ☐
f6<=(B or C) and (not A or B or D) and (A or not B); f6 ☐

D4. *Check all the logical functions that have another function identical to them:*

f1<=(A xor C) or (A and not C); f1 ☐
f2<=(B or C) and (not A or B or C); f2 ☐
f3<=((C and not B) or (B and A)); f3 ☐
f4<=(A or C) and (not A or not C); f4 ☐
f5<=(A and not B) xor (A and C); f5 ☐
f6<=(A and not C) or (C and not A); f6 ☐

E4. *Check all the logical functions that have another function identical to them:*

f1<=((C and not B) or (C and B and A)); f1 ☐
f2<=(A xor C) or (A and not C); f2 ☐
f3<=(A or B) and (not A or B or C); f3 ☐
f4<=(not A or not C) and (C or A); f4 ☐
f5<=(not C and A) or (not A and C); f5 ☐
f6<=(A and not B) xor (A and not B and C); f6 ☐

B1. Parse a function written in VHDL syntax so that it contains only the "or", "and" and "not" operations, with the "not" operation not applied anywhere to the expression in parentheses. A, B, C, F1 and F2 are of type boolean.

F1<= A xor (B or C);

F1<=.....

F2<= **not** (A and (B **xor** C));

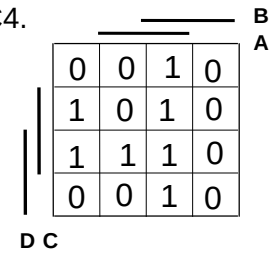
F2<=.....

Solution by simply writing down and applying De Morgan's rule

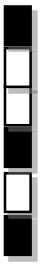
The results can be found on the next page

Results

C4.

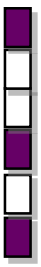


f1<=(((not B and not A) or (A and D)) and C) or (B and A);
f2<=((((not A or D) and C) or A) and B);
f3<=((C and (not A or D)) or B) and A and not B;
f4<=((not A or D) and C and not B) or (B and A);
f5<=(A and not B) or (A and D) or (B and A);
f6<=(B or C) and (not A or B or D) and (A or not B);



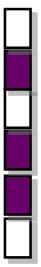
D4. *:*

f1<=(A xor C) or (A and not C);
f2<=(B or C) and (not A or B or C);
f3<=((C and not B) or (B and A));
f4<=(A or C) and (not A or not C);
f5<=(A and not B) xor (A and C);
f6<=(A and not C) or (C and not A);



E4.

f1<=((C and not B) or (C and B and A));
f2<=(A xor C) or (A and not C);
f3<=(A or B) and (not A or B or C);
f4<=(not A or not C) and (C or A);
f5<=(not C and A) or (not A and C);
f6<=(A and not B) xor (A and not B and C);



B1.

F1<= A xor (B or C);

F1<=(A and not B and not C) or not A and (B or C);……

F2<= **not** (A and (B **xor** C));

F2<= not A or ((not B or C) and (B or not C));……

*Function Q = ((*A xor B) and (D xor C)) or ((not A and B) xor (not D and C))

decompose into form>

Q=C.D.f0(A,B) + $\overline{C}$.$\overline{D}$.f1(A,B) + C.$\overline{D}$.f2(A,B) + $\overline{C}$.D.f3(A,B) *using Shannon expansion.*

Write the resulting functions f0, f1, f2 and f3 as Karnaugh maps:

.

f0=

B \ A	0	1
0		
1		

f1=

B \ A	0	1
0		
1		

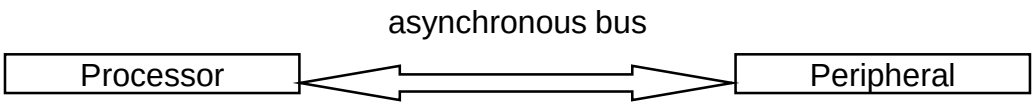
f2=

B \ A	0	1
0		
1		

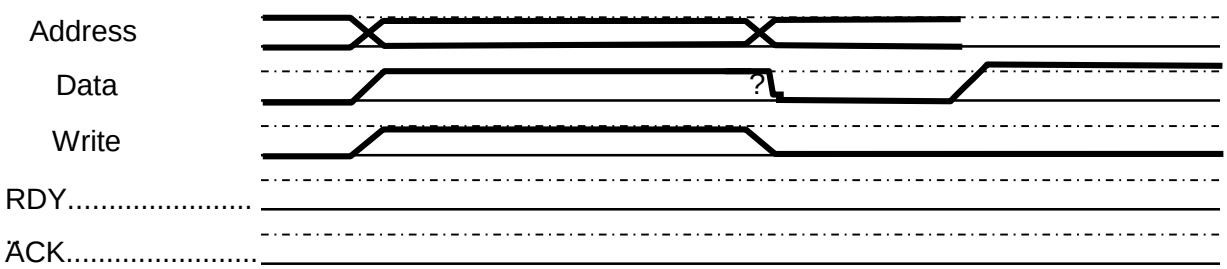
f3=

B \ A	0	1
0		
1		

Note: The line symbolizes the negation of signal only in logical expressions.
In Karn. maps, it means that the corresponding variable is in logical1.



I6. Fill in the missing signals and waveforms on the asynchronous bus, where the signal Write='1' specifies writing data to the peripheral, Write='0' means reading data from the peripheral; Data is a bidirectional signal and the signal "Address" in the figure below selects a peripheral register each time.



.

Results on the next page

Results

*Function Q = ((*A xor B) and (D xor C)) or ((not A and B) xor (not D and C))

decompose into form>

Q=C.D.f0(A,B) + $\overline{C}$.$\overline{D}$.f1(A,B) + C.$\overline{D}$.f2(A,B) + $\overline{C}$.D.f3(A,B) *using Shannon expansion.*

Write the resulting functions f0, f1, f2 and f3 as Karnaugh maps:

.

f0=

B \ A	0	1
0	0	0
1	1	0

f1=

B \ A	0	1
0	1	1
1	1	1

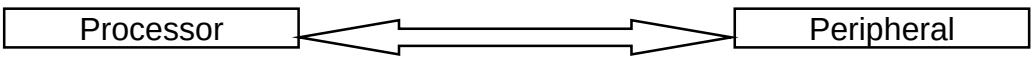
f2=

B \ A	0	1
0	0	1
1	1	0

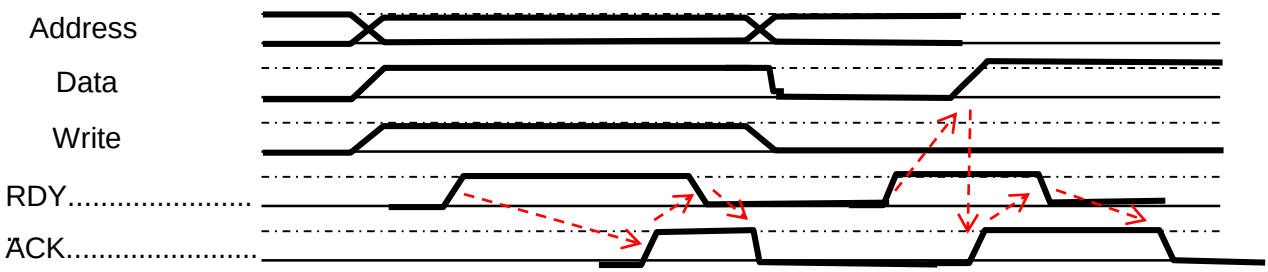
f3=

B \ A	0	1
0	0	0
1	1	0

asynchronous bus



I6. Fill in the missing signals and waveforms on the asynchronous bus, where the signal Write='1' specifies writing data to the peripheral, Write='0' means reading data from the peripheral; Data is a bidirectional signal and the "Address" signal in the figure below selects a peripheral register each time.

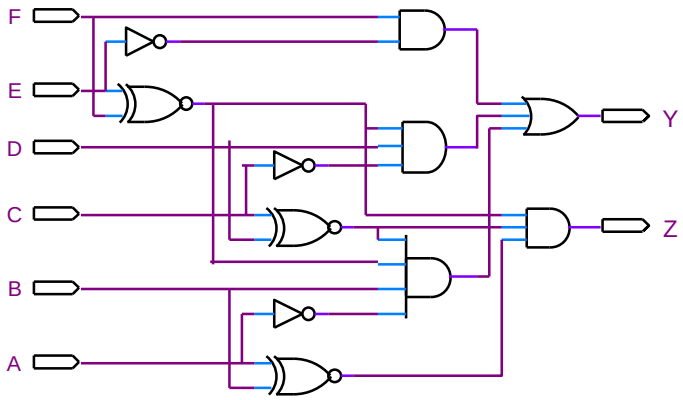


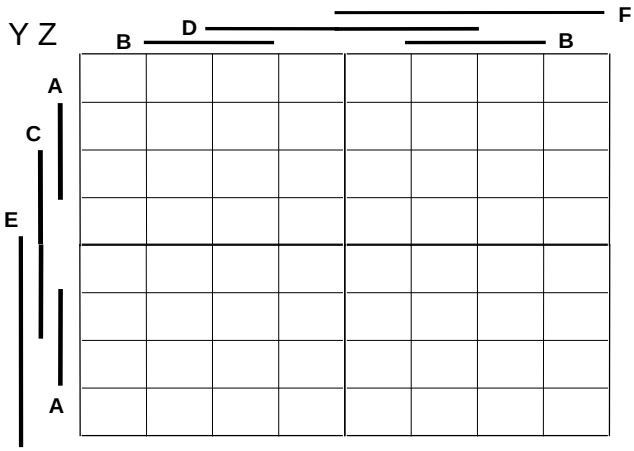
.

RDY device1-provider reports ready to write or ready to read data

ACK device2-consumer confirms reading data or sending data to the bus
Red arrows indicate the sequence between signals

E1. *In the diagram you can see the following wiring. Write its truth table.*





Solution on the next page.

Look for similarities in the schematic and decompose the functions to connect smaller circuits.

E1.  is the equivalence operation ≡, i.e. equality,

so Z value is FDB number = ECA number, i.e., at diagonal is 1, otherwise 0.

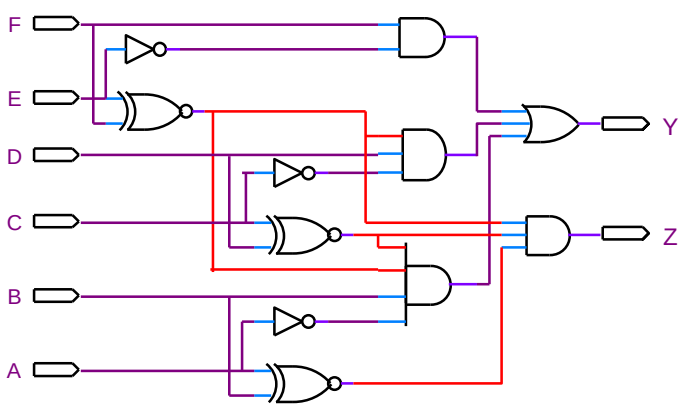
If we realize that (F and not E) is a comparison of (F>E), or the F bit is greater than the E bit, then we can easily construct the equation

Y = (F>E) or ((F ≡ E) and (D>C)) or ((F ≡ E) and (D ≡ C) and (B>A))

, which is a 3-bit "greater than" comparator, Y: FDB number > ECA number.

We fill the K-map of the comparator according to the bit hierarchy.

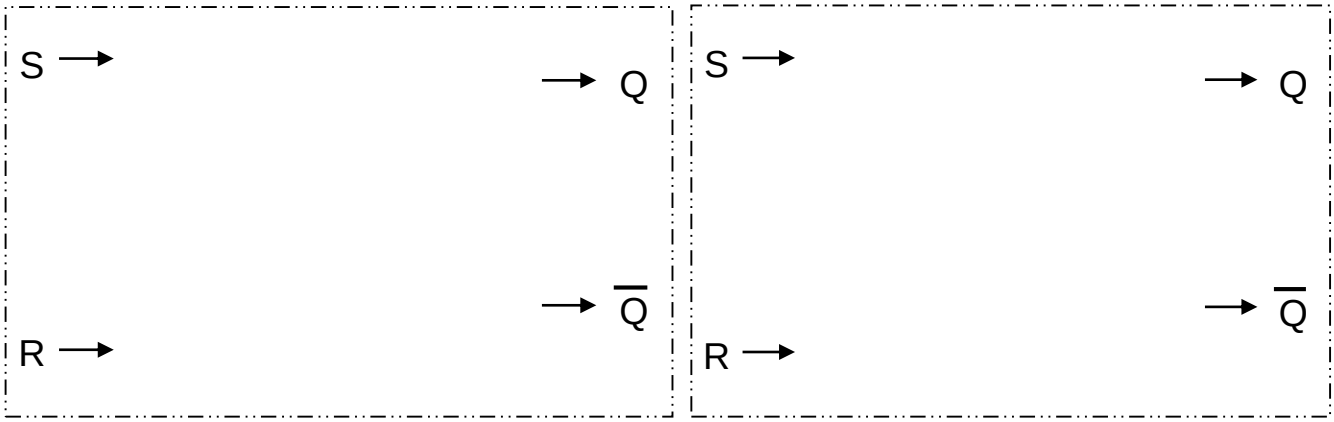
Only when the higher bits are identical, the least significant bits influence the result.
Shown in yellow F ≡ E, in blue D ≡ C, in green B ≡ A



Y Z	B		D				B	
A	01	10	10	10	10	10	10	10
C	00	01	10	10	10	10	10	10
	00	00	01	00	10	10	10	10
E	00	00	10	01	10	10	10	10
	00	00	00	00	01	10	00	00
	00	00	00	00	00	01	00	00
	00	00	00	00	10	10	01	00
A	00	00	00	00	10	10	10	01

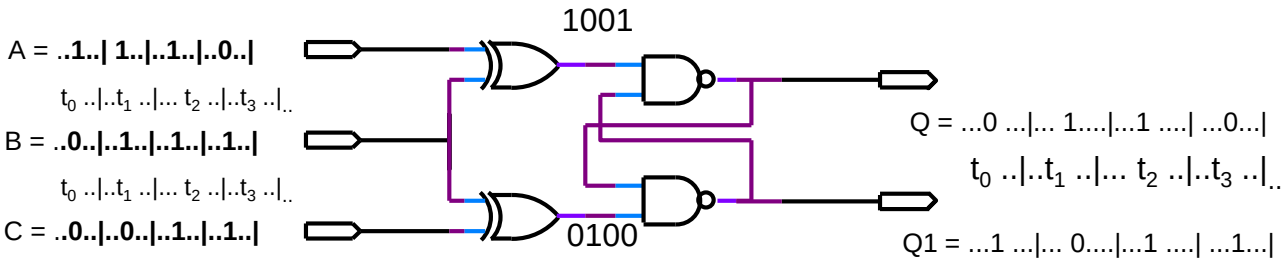
F

E6. Draw an R-S type flip-flop circuit composed of only NOR gates and the same using only NAND gates



Solutions can be found in the lectures and the textbook Logic Circuits.

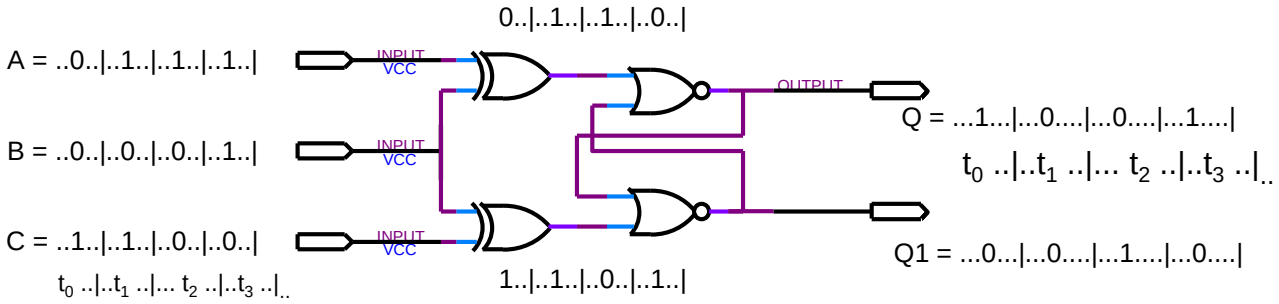
The exam can added the query for the truth table of both latch types



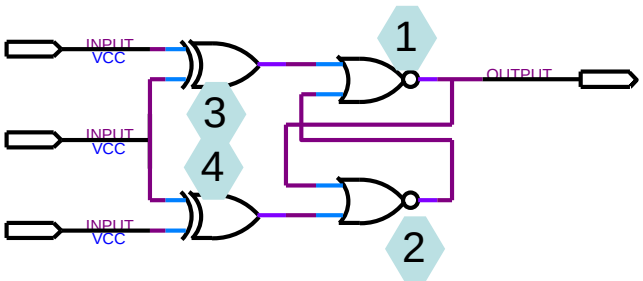
Similar problems involving flip-flop circuits are best solved by making a sequence of signals at the input of the flip-flop circuit and then remembering how the flip-flop circuit works. Non-RS drawn from NAND is controlled by logic zeros. Inputs S=1 R=0 set Q to 0, Inputs S=0 R=1 set Q to 1, values of inputs S=0 R=0 bring the outputs to a state where both Q=1 and Q1=1 and the next state is not known, but fortunately S=1 R=0 follows again so Q will be 0.

We don't have any forbidden inputs in RS here !!!
It only results from the additional constraint Q=not Q1, which is not unspecified.

Inputs A, B, C had the values shown in the figure at times t_0 , t , t_{12} , t_3 . Write the values of output Q. Assume that the intervals between input changes are so long that gate delays can be neglected.



RS latch from NOR gates behaves differently than from NAND-



Decompose the function Q=f(A,B,C,Q) from the previous question into the form Q=$\overline{Q}$.f0(A,B,C) + Q.f1(A,B,C) using Shannon decomposition. Write the resulting functions f0 and f1 as Karnaugh maps.

Solution: a diagram is basically a graphical notation of a logical expression. Ignore the output of Q1.Use the procedure shown in lecture 2. If we enumerate the gates and sequentially rewrite their operations into an expression:

Q := 1 := $\overline{3}$ + 2 := (A xor B) + $\overline{(Q + 4)}$:= $\overline{(A \text{ xor } B) + \overline{(Q + (B \text{ xor } C))}}$

For simplicity, we will break down the upper negation according to de Morgan's rule. We take advantage of the fact that the operation xor is actually a non-equivalence, and so (A xor B) is (A ≠ B), and (A xor B) is (A ≡ B)

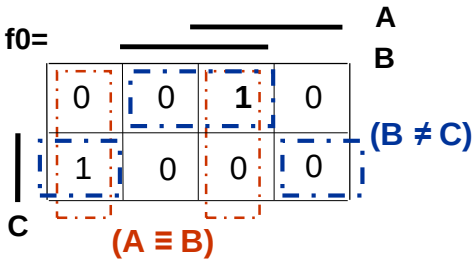
Q := $\overline{(A \text{ xor } B)}$. (Q + (B xor C)) := **(A ≡ B) . (Q + (B ≠ C))**

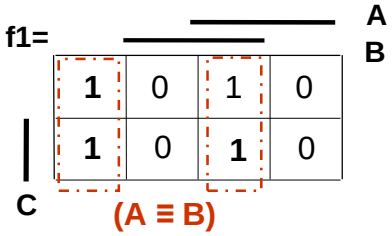
From the last form, we can easily calculate the cofactors f0 and f1 of the Shannon decomposition by finding the values of the function f(A,B,C,Q) at the points Q='0' and Q='1', by substituting them after Q.

f0 := f(A,B,C,'0') := (A ≡ B) . ('0' + (B ≠ C)) := (A ≡ B) . ('0' + (B ≠ C)) := **(A ≡ B) . (B ≠ C)**

f1 := f(A,B,C,'1') := (A ≡ B) . ('1' + (B ≠ C)) := (A ≡ B) . '1' := **(A ≡ B)**

*We write the two found functions f0:=***(A ≡ B) . (B ≠ C)** and *f1:=***(A ≡ B) as** *Karnaugh maps*





Similar example

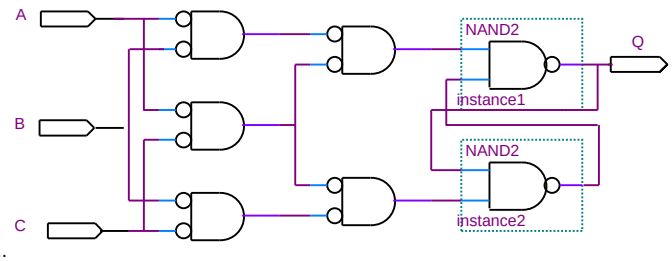
Inputs A, B, C had the values shown in the figure at times t_0 , t , t_{12} , t_3 . Write the values of output Q. Assume that the intervals between input changes are so long that gate delays can be neglected.

A = ..0..|..0..|..1..|..1..|

B = ..0..|..0..|..0..|..1..|

C = ..1..|..0..|..0..|..0..|

t_0 ..|..t_1 ..|... t_2 ..|..t_3 ..|..



Q = ...|.....|.....|.....|

t_0 ..|..t_1 ..|..t_2 ..|..t_3 ..|..

Q1 = ...|.....|.....|.....|

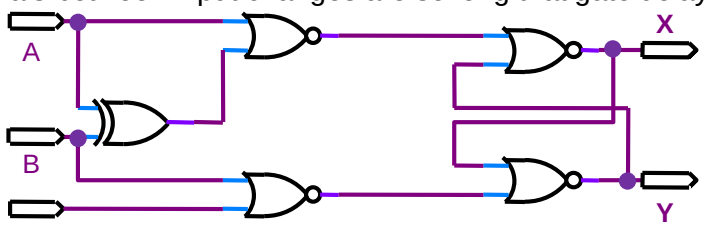
Inputs A, B, C had the values shown in the figure at times t_0 , t , t_{12} , t_3 . Write the values of outputs X and Y. Assume that the intervals between input changes are so long that gate delays can be neglected.

A = ..0..|..0..|..1..|..1..|

B = ..0..|..0..|..0..|..1..|

C = ..1..|..0..|..0..|..0..|

t_0 ..|..t_1 ..|... t_2 ..|..t_3 ..|..



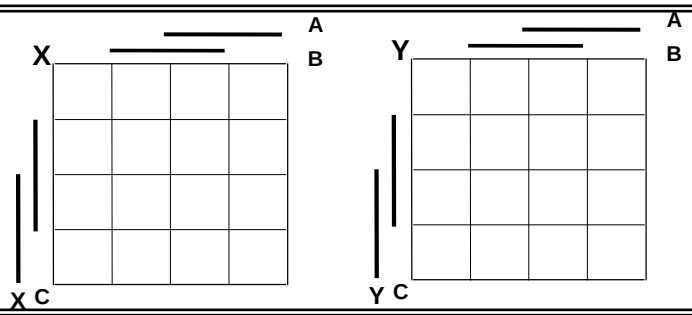
X = ...|.....|.....|.....|

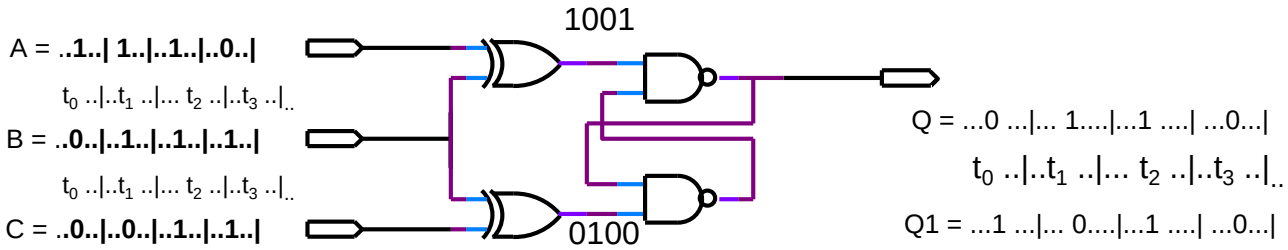
Y = ...|.....|.....|.....|

t_0 ..|...t_1 ..|... t_2 ..|...t_3 ..|..

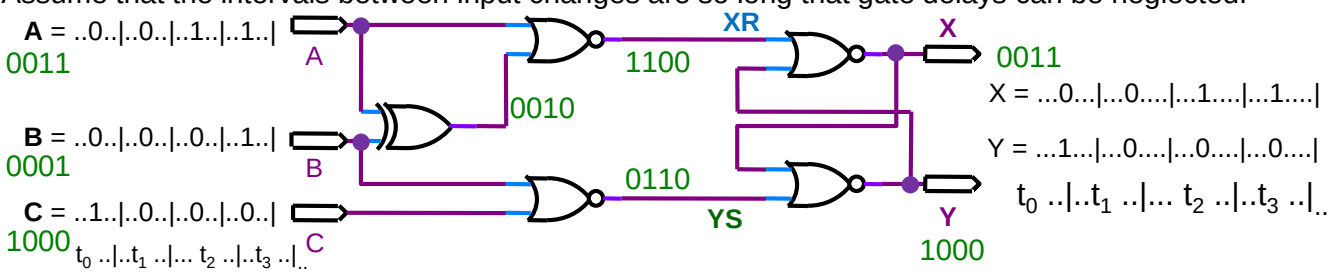
For the X and Y outputs of Example 4 write their Karnaugh maps, i.e. the maps of the functions X = f(A, B, C, X)

Y = f(A, B, C, Y)

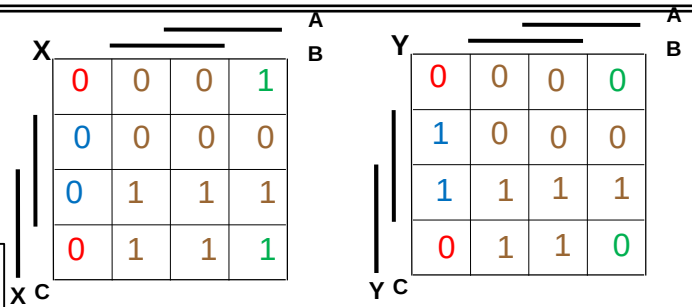




Inputs A, B, C had the values shown in the figure at times t_0 , t , t_{12} , t_3 . Write the values of outputs X and Y. Assume that the intervals between input changes are so long that gate delays can be neglected.



For the X and Y outputs of Example 4 write their Karnaugh maps, i.e. the maps of the functions X = f(A, B, C, X)
Y = f(A, B, C, Y)



To construct the maps, we can either use logical expressions for the X and Y outputs that we find, or we can calculate the maps of the flip-flop circuit inputs, labeled XR (reset) and YS (set) in the schematic. We then fill in the X and Y maps according to the RS flip-flop circuit table from the NOR gates. If both inputs XS and YS are in states 0, i.e. the circuit remembers, then there is a 1 everywhere in the map where X=1, and a 0 where X=0 - i.e. no change in the next state. Same for Y.

YS	XR	X	Y
0	0	$X(t)$	$Y(t)$
0	1	0	1
1	0	1	0
1	1	*0*	*0*

*Fill in the missing parts of the VHDL program to create a D flip-flop circuit with input **d** and output **q**, having asynchronous zeroing at aclrn='0' and synchronous zeroing at sclrn='0'.*

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mydff is port (clock, d, aclrn, sclrn : in std_logic; q : out std_logic) end mydff;
architecture rtl of mydff is begin
.....
.....
.....
.....
.....
.....
end rtl;
```

*Fill in the missing parts of the VHDL program to produce a 100 bit shift register, i.e. its output **q** will have an input **d** delayed by 100 clock cycles of the signal **clk**. The register is synchronously zeroed at signal sclrn='0'. Choose the most economical code. (Hint: The shortest code has no instruction cycles).*

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity pos100 is port (clock, d, sclrn : in std_logic; q: out std_logic) end pos100;
architecture rtl of pos100 is begin
.....
.....
.....
.....
.....
.....
.....
end rtl;
```

The editors of a technical magazine have asked you to decipher the following mysterious VHDL code.

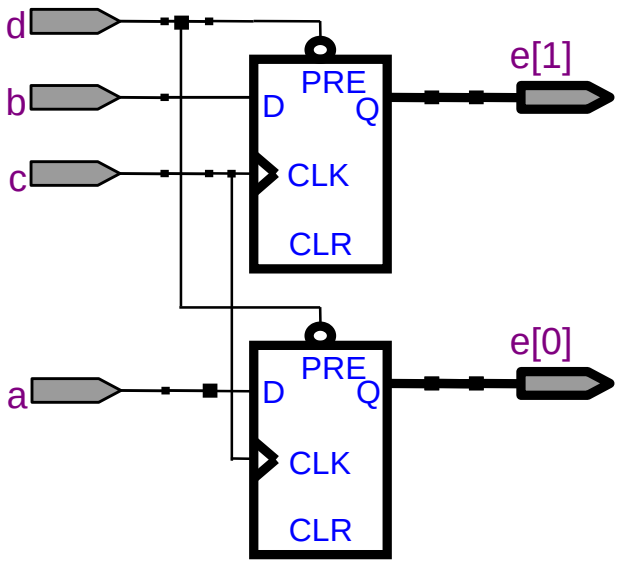
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zahadum is port (a, b, c : in std_logic; d : out std_logic); end;
architecture rtl of zahadum is signal z:std_logic_vector(0 to 2);
begin process(a, b) begin
  if a='0' then z<=(others=>'0'); elsif rising_edge(b) then z<=c & z(0 to 1); end if; d<=z(2);
end process; end rtl;
```

Using gates and flip-flop circuits, draw a diagram of the logic circuit corresponding to the VHDL code shown. Name the circuit correctly:

The editors of a technical magazine have asked you to decipher the following mysterious VHDL code.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zahadum is port (a, b, c, d : in std_logic; e : out std_logic_vector(1 downto 0)); end;
architecture rtl of zahadum is signal z:std_logic_vector(1 downto 0);
begin process(d, c) begin
  if d='0' then z<=(others=>'1'); elsif c'event and c='1' then z<=b & a; end if; e<=z;
end process; end rtl;
```

Using gates and flip-flop circuits, draw a diagram of the logic circuit corresponding to the VHDL code shown. Name the circuit correctly in the figure caption:



Two-bit register with asynchronous presetting

Write the other VHDL codes and test if they work.

Writing the whole VHDL circuit will be just in the premium question.

C program searches for the maximum in the **data** field.

```
int data[] = { 0, 1, -2, 3, 4, -5, -6, -7, 8, 9 };
int max = INT_MIN; // =-2147483648 (v <limits.h>)
for (int i = 0; i < sizeof(data)/sizeof(int); i++) // i<10
{ if (data[i] > max) max = data[i];
}
```

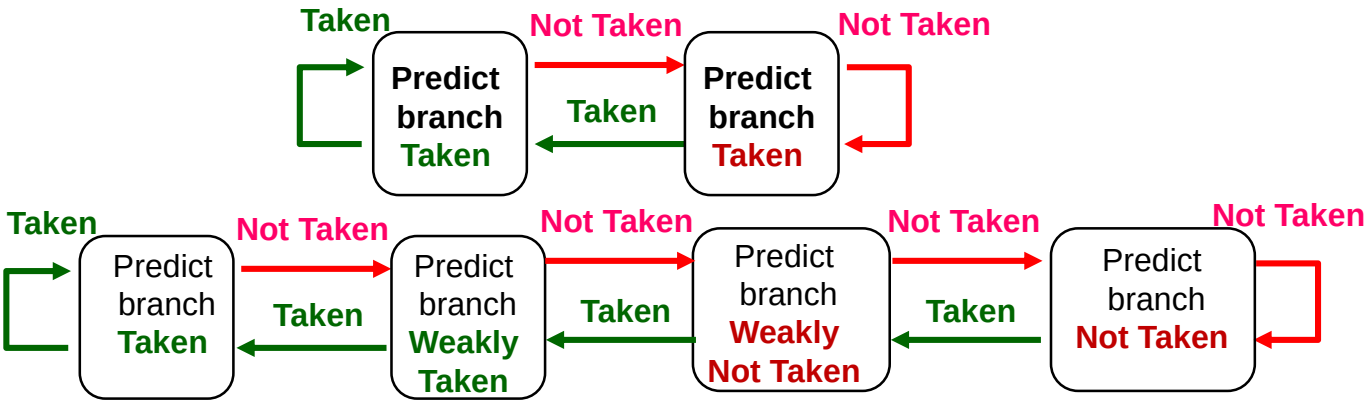
How many wrong jump predictions will it have, assuming that a for-loop with constant bounds is economically translated by a do-while cycle and the processor uses only:

* single-bit predictors that had their default state Not-taken, NT , misses=.....
* two-bit predictors that had their default state WT, Weakly Taken, misses=.....

Solution: Processor assemblers do not know structured programming. Let's rewrite the program in jumps first. And for loop with clear bounds is replaced by a do-while, i.e. a test at the end, which is a common translation for it:

```
int data[] = { 0, 1, -2, 3, 4, -5, -6, -7, 8, 9 };
int max = INT_MIN; // =-2147483648 (v <limits.h>)
int i = 0;
Loop: if (data[i] <= max) goto Next;
      max = data[i];
Next: i++; if(i < 10) goto Loop;
```

Let us recall the predictors. Each branch in the program has, of course, its own independent predictor.



A 2-bit predictor (starting in NT) makes 2 errors in the outer loop of Loop-Next (1 at the beginning + 1 at the end), a 1-bit predictor (starting in WT) only 1 at the end. Additional mispredictions will add **if** to the data, for which we build a table to update the predictor state based on whether or not it jumped.

Branch	Not	Not	Taken from	Not	Not	Taken from	Taken from	Taken from	Not	Not
Data	0	1	-2	3	4	-5	-6	-7	8	9
1 bit	NT	NT	NT>T	T>NT	NT	NT>T	T	T	T>NT	NT
He hit	hit	hit	miss	miss	hit	miss	hit	hit	miss	hit
2 bit	WT>WNT	WNT>NT	NT>WNT	WNT>NT	NT	NT>WNT	WNT>WT	WT>T	T>WT	WT>WNT
He hit	miss	hit	miss	hit	hit	miss	miss	hit	miss	miss

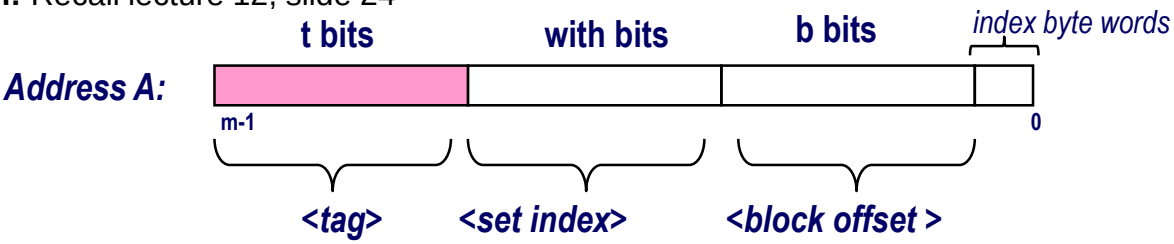
1-bit: 2 (loop) + 4 (if) = 6 **2-bit: 1 (loop) +6 (if) = 7**

The 32-bit processor reads data from address 0x1234. What will be its block-offset and set index, and the cache tag if the cache has a total of only 64 bytes organized:

* directly mapped cache with line length 1 word(block) or 2 words.

* two-way partially associative cache with line length 1 word or 2 words.

Solution: Recall lecture 12, slide 24



First, we express the address in bits 0001 0010 0011 0100. The lower two are not considered in the cache of a 32-bit processor, its word has 4 bytes. And it only reads from addresses divisible by 4, which contain 00. So in total there will be only 16 words (=64/4) in the cache,.

If it has a directly mapped cache it has 1 word in the block and so does not need a block-offset. There are 16 sets, so each word in the cache is a set. We have address decomposition:

0 0 0 1 0 0 1 0 0 0 | 1 1 0 1 | *0 0*

tag | set index

*** set-index** = 1101 = 13_{10} , and **tag** 0001001000 = 64+8=72_{10} , block index -.

If it increases the block length to 2 words, then 1 bit is needed for the block-offset. When the cache length is fixed, the number of sets is reduced to 8 (3 bits). We have a decomposition:

0 0 0 1 0 0 1 0 0 0 | 1 1 0 | 1 | *0 0*

tag | set index | blok offset

*** set-index** = 110 = 6_{10} , and **tag** 0001001000 = 64+8=72_{10} , block index 1.

The 2-way partially associative cache with a 1 word block does not have a block-offset, but the number of sets is reduced to 8 (3 bits), because each set contains two rows, from which it is selected associatively according to the address tags, their contents are currently stored in them:

0 0 0 1 0 0 1 0 0 0 1 | 1 0 1 | *0 0*

tag | set index

*** set-index** = 101 = 5, and **tag** 00010010001 = 128+16+1=145_{10} , block index -.

If the length of the block is increased to 2 words (1 bit of the index) and the set has 2 lines, then 1 set takes **4 words of cache memory.** Since the entire cache has only **16 words (64 bytes)** in total, we end up with **4 sets** (2 bits):

0 0 0 1 0 0 1 0 0 0 1 | 1 0 | 1 | *0 0*

tag | set index | blok offset

*** set-index** = 10=2, and **tag** 10010001=128+16+1=145_{10} , block index = 1.

32-bit processor reads data from addresses 0x4, 0x8, 0xC, 0x48, 0x4, 0x8, 0xC, 0x88, 0x8, 0xC Let its cache be only **64 bytes**. What will be the cache hit and miss if its organization is:

* directly mapped cache with a line length **of 2 words (blocks)**.

* 2-way partially associative cache with **2 word (block)** lines with LRU substitution

Solution: First convert the addresses to tags, set indexes and block-offset (i.e. the index of the word in the block)

		directly mapped			**2-way**		
			3-bits	1 bit		2-bits	1-bit
		tag	**set**	**block**	**tag**	**set**	**block**
0x4	00*...00000100*	0	0	1	0	0	1
0x8	00*...00001000*	0	1	0	0	1	0
0xC	00*...00001100*	0	1	1	0	1	1
0x48	00*...01001000*	1	1	0	2	1	0
0x88	00*...10001000*	2	1	0	4	1	0

For the directly mapped cache we have 8 sets with blocks of 2 words, see the previous example. All words in the block are read at once. *Note: Blocks longer than 1 word are only chosen if we can also fill them quickly from the main memory.*

0x4 - **1*miss**, set 0 from main memory from addresses 0x0 and 0x4 is filled

0x8 - **1*miss**, but at the same time as 0x8 the data from address 0xC is read into set 1,

0xC - so it has a hit

0x48 **- 1*miss**, but the data in set 1 is replaced by the content of addresses 0x48 and 0x4C

0x4 - hit on set 0

0x8 - **1*miss**, in set 1, the address tag is 0x48, so 0x8 and 0xC are read again from the main memory

0xC - hit, it was loaded from main memory while solving miss 0x8

0x88 - **1*miss**, in set 1 the content is replaced from addresses 0x88 0x8C

0x8 - **1*miss**, because tag 2 of address 0x88 was found in set 1, so it is read again from main memory.

0xC - was loaded together with the previous 0X8, so it has a hit

Result 6*miss, 10-6=4*hit

We can even draw the final as values read from [addresses] of the main memory

	tag	**block-0**	**block-1**
set 0	**0**	[0x0]	[0x4]
set 1	**0**	[0x8]	[0xC]
set 2	?	?	?
set 7	?	?	?

32-bit processor reads data from addresses 0x4, 0x8, 0xC, 0x48, 0x4, 0x8, 0xC, 0x88, 0x8, 0xC

For a partially associative cache with 2 lines and a block of length 2 words, we will have only 4 sets, see the previous address decomposition example:

0x4 - **1*miss**, set 0 is read from the contents of addresses 0x0 and 0x4

0x8 - **1*miss**, but at the same time as 0x8 the data from address 0xC was also read into set 1, rows 1 were marked as LRU (least recently used) for sets 0 and 1 at the same time

0xC - so it has a hit

	line 0			line 1		
	tag	block-0	block-1	tag	block-0	block-1
set 0	0	[0x0]	[0x4]	- LRU	?	?
set 1	0	[0x8]	[0xC]	- LRU	?	?
set 2	- LRU	?	?	-	?	?
set 3	- LRU	?	?	-	?	?

0x48 - **1*miss**, but the data from the address is stored here in a free line and the opposite line is marked as LRU.

	line 0			line 1		
	tag	block-0	block-1	tag	block-0	block-1
set 0	0	[0x0]	[0x4]	- LRU	?	?
set 1	0 LRU	[0x8]	[0xC]	2	[0x48]	[0x4C]
set 2	- LRU	?	?	-	?	?
set 3	- LRU	?	?	-	?	?

0x4 - hit on set 0 and

0x8 and 0xC, 2*hit, but whose reading simultaneously marks the opposite line as LRU

	line 0			line 1		
	tag	block-0	block-1	tag	block-0	block-1
set 0	0	[0x0]	[0x4]	- LRU	?	?
set 1	0	[0x8]	[0xC]	2 LRU	[0x48]	[0x4C]
set 2	- LRU	?	?	-	?	?
set 3	- LRU	?	?	-	?	?

0x88 - although it has **1*miss**, in set 1 it will replace its LRU line 1, where 0x48 and 0x4C used to be. Instead, the content from addresses 0x88 and 0x8C with tag 4 will be loaded.

0x8 and 0xC have hit again.

Result 4*miss, 10-4 **=6*hit**