Informatics 1 - CL: Tutorial 3

Propositional Logic: Karnaugh Maps and CNF [1]

Week 5: 15 – 19 October 2018

> Please attempt the entire worksheet in advance of the tutorial, and bring all work with you. Tutorials cannot function properly unless you study the material in advance. Attendance at tutorials is **obligatory**; if you cannot attend your allocated tutorial go to another session. Please let the ITO know if you cannot attend.
>
> You may work with others, indeed you should do so; but you must develop your own understanding; you can't phone a friend during the exam. If you do not master the coursework you are unlikely to pass the exams.

We will use propositional logic in many ways. One important use is to specify and reason about finite state systems. We use a propositional logic with n propositional letters to describe a system with n state bits that can be used to encode up to 2^n states.

Often we will want to specify some subset of the set of states. For example, we may want to specify the set legal states of a traffic light. For a system of cars and traffic lights we may want to specify the set of safe states – in which accidents will not happen.

It is natural to specify safety using a number of clauses, each of which addresses a particular hazard. A safety clause for a crossing with pedestrian and traffic lights might specify that at least one of the lights must be red, $R_P \vee R_T$. Later, we will use the tools developed from ideas suggested by some of the examples in this tutorial to specify and reason about such systems.

In this tutorial we focus on Karnaugh Maps a tool used the design of combinational logic circuits.

[1] *This tutorial exercise sheet was written by Michael Fourman.*
Send comments to `Michael.Fourman@ed.ac.uk`

1 Logic 0

This tutorial introduces you to Boolean logic. You will use Haskell as an interactive calculator, to introduce you to Boolean functions. You should load `CL3.hs` to work on this tutorial.

```
Prelude> :load CL3
[1 of 1] Compiling CL3              ( CL3.hs, interpreted )
Ok, modules loaded: CL3.
*CL3>
```

A Boolean function is a function that takes Boolean arguments and returns a Boolean result. The type `Bool` in Haskell has two values `True` and `False`, and advertises `&&` ($\wedge$), `||` ($\vee$), and `not` ($\neg$), as its basic Boolean functions. *Any* Boolean function (with finitely many arguments) can be implemented in terms of $\wedge, \vee, \neg$.

Every well-formed formula (wff) of propositional logic corresponds to a Boolean function - given Boolean values for the variables in the wff the truth value of the wff is the output of the function. We can use Haskell to implement this Boolean function.

For example the Boolean function `F` corresponding to the wff $(A \vee G) \wedge \neg(R \vee B)$ is modelled by the Haskell function `f` defined as follows:

```
f::Bool->Bool->Bool->Bool -> Bool
f p q r s = (p||q) && not(r||s)
```

QuickCheck is a library for random testing of program properties. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and `quickCheck` then tests that the properties hold in a large number of randomly generated cases. You can use `quickCheck` to try to find whether `f` is a tautology. The property we are interested in is that `f` always returns true.

```
(f a r g b) == True
```

But for any Boolean value `x`, we have the equality `x = (x == True)` (check the truth table).

So `f a r g b == True` is the same as `f a r g b` – and `f` is the property we need to check. `quickCheck f` tests `f` with randomly generated inputs, `a r g b`, to check whether the result `f a r g b` is always `True`.

```
> import Test.QuickCheck
> quickCheck f
*** Failed! Falsifiable (after 1 test and 2 shrinks):
False
False
False
False
```

An expression is a tautology iff it is true for every valuation of the variables. Lines 4–7 give values for `a, g, r, b` that falsify our expression, so it is not a *tautology*. We can also check whether `f` is *satisfiable*, by asking `quickCheck` whether its negation is a tautology. Any valuation that falsifies the negation satifies the original formula.

```
*CL3> quickCheck (f.not)
*** Failed! Falsifiable (after 1 test):
False
False
False
True
```

This valuation makes the original function true, so the original expression is satisfiable. Since it is not a tautology it is *contingent* – some valuations make it true and some make it false.

If `quickCheck` provides a counter-example — a valuation that falsifies a formula — then we can be certain that the formula is not a tautology. However, `quickCheck` merely tries a limited number of randomly selected valuations. If it fails to find a counterexample we can draw no conclusion.

```
f::Bool->Bool->Bool->Bool -> Bool
f p q r s = (p||q) && not(r||s)
```

Try `quickCheck` on this example. QuickCheck may fail, but **you** should easily be able to find a valuation that falsifies this function.

If `quickCheck` fails to find a counterexample you can ask it to perform more tests, with a call such as `quickCheck (withMaxSuccess 1000 ff)`. How many tests does `quickCheck` need to falsify `ff`? (Your answer will vary if you try several times – `quickCheck` chooses tests at random.)

Here is an even simpler function (simpler for you) that is likely to defeat `quickCheck`.

```
-- type declaration omitted (too long for this margin)
gg a b c d e f g h i j k l m n o p q r s t u v w x y z =
  or [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]
-- you should be able to find a counterexample
-- but quickCheck is unlikely to help you!
```

1. What valuations falsify this function? (There are not many.) How many tests chosen uniformly at random would you need to have a greater than even ($> 50\%$) chance of finding such a valuation?

The functions `taut1` ...`taut4` provide exhaustive checks to determine whether Boolean functions with 1–4 arguments are tautologies. If these return `True` you have a tautology.

```
taut1 f = f True && f False                  -- for functions with one argument
taut2 f = taut1 (f True) && taut1 (f False)  -- for functions with two arguments
taut3 f = taut2 (f True) && taut2 (f False)  -- for functions with three arguments
taut4 f = taut3 (f True) && taut3 (f False)  -- for functions with four arguments
```

2. Use Haskell to check each of the following equations.

$x \vee (y \vee z) = (x \vee y) \vee z$	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$	associative
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	distributive
$x \vee y = y \vee x$	$x \wedge y = y \wedge x$	commutative
$x \vee 0 = x$	$x \wedge 1 = x$	identity
$x \vee 1 = 1$	$x \wedge 0 = 0$	annihilation
$x \vee x = x$	$x \wedge x = x$	idempotent
$x \vee \neg x = 1$	$\neg x \wedge x = 0$	complements

$x \vee (x \wedge y) = x$	$x \wedge (x \vee y) = x$	absorbtion
$\neg(x \vee y) = \neg x \wedge \neg y$	$\neg(x \wedge y) = \neg x \vee \neg y$	de Morgan
$\neg\neg x = x$	$x \to y = \neg x \leftarrow \neg y$	

For example, we can try the distributivity of **or** over **and**: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.

```
*CL3> distoa' x y z = x || (y && z) == (x || y) && (x || z)
*CL3> taut3 distoa'
False
```

Strange! Can you work out what is going wrong? Here is a clue

```
*CL3> :info (==)
  ...
infix 4 ==
*CL3> :info (&&)
(&&) :: Bool -> Bool -> Bool
infixr 3 &&
*CL3> False && True == False
False
*CL3> (False && True) == False
True
```

The problem was that Haskell didn't understand what we meant. Haskell gives `(==)` precedence 4, while `(&&)` has precedence 3. So we have to add some extra brackets to say in Haskell what was written in mathematical notation. Let's try again:

```
*CL3> distoa x y z = (x || (y && z)) == ((x || y) && (x || z))
*CL3> taut3 distoa
True
```

Now we've overcome that pitfall, it should be easy to check the equations.

This distributive law is somewhat like the algebraic law $x(y+z) = (xy+xz)$. Actually, $\wedge$ is in many ways more like $\times$, and $\vee$ is more like $+$, so the algebraic law is even more like the other distributive law, the distributivity of **and** over **or**: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.

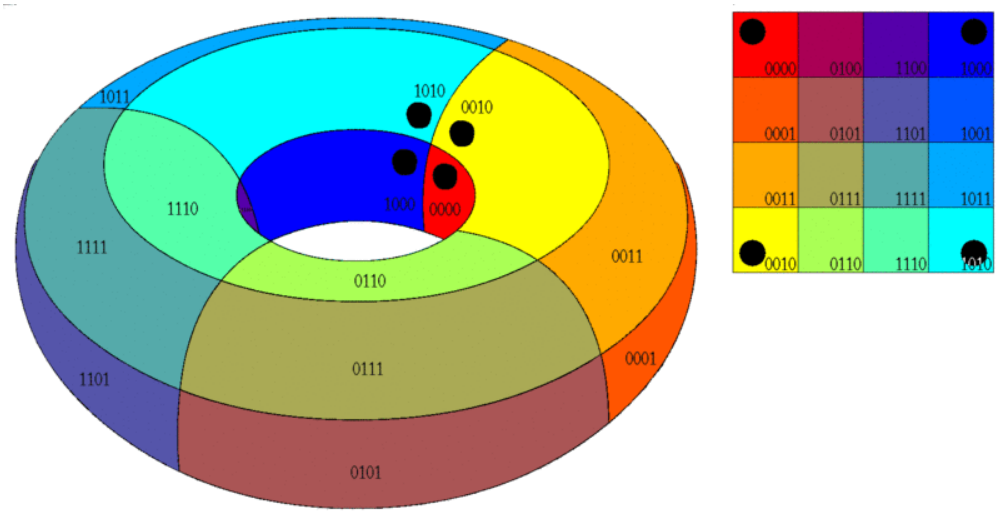
3. In algebra, you are probably familiar with the equation $(a+b)(c+d) = ac+ad+bc+bd$. This follows from the distributive law, so we should expect two corresponding laws for Boolean algebra (since in Boolean algebra we have two distributive laws). Write down these two laws, and use `taut4` to check them.

Using our tautology-checking functions we can tell whether a Boolean expression is a tautology. If the expression is not a tautology the checker returns `False`, and you can use `quickCheck` to try to find a counter-example, and for functions with few variables this works pretty well. Howeveer, it may be more illuminating to look at the Karnaugh map.

Karnaugh Maps

Karnaugh maps provide a graphic representation for a Boolean function of four variables. Two expressions have the same Karnaugh map iff they represent the same Boolean function. The Karnaugh map is a table with 16 cells, corresponding to the sixteen states represented by four boolean values assigned to the propositional letters A, G, R, B.

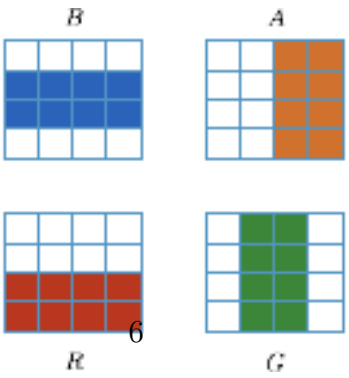
If A, G, R, B have binary values a, g, r, b with 1 representing $\top$ and 0 representing $\bot$, we will refer to the state using the decimal value of the binary string $agrb$. Thus 0 represents the state 0000 in which all four atoms are false, while 15 represents the state 1111 in which they are all true. The map is arranged so that the codes for any two adjacent cells differ by exactly one bit. We view the table as wrapped around both horizontally and vertically, so that each cell on the top row is adjacent to the corresponding cell on the bottom row; and each cell in the left-most column is adjacent to the corresponding cell in the right-most column.



K-map encoding

4. Label each of the sixteen squares in the Karnaugh map with the corresponding decimal number.

		AG			
		00	01	11	10
RB	00				
	01				
	11				
	10				



The coloured diagrams show the states in which each of the atomic propositions A, G, R, B is true.

```
km :: (Bool -> Bool -> Bool -> Bool -> Bool) -> IO ()
```

```
*CL3> km f
   | 00 01 11 10
-- - -- -- -- --
00 |  0  1  1  1
01 |  0  0  0  0
11 |  0  0  0  0
10 |  0  0  0  0
```

The Haskell module `CL3` provides a function `km` that prints a Karnaugh map for a Boolean function with four arguments. Each state is labelled either 1 (true) or 0 (false).

The Haskell type `Bool` has operations corresponding to conjunction $\wedge$, disjunction $\vee$, and negation $\neg$, built-in. In fact it also has built-in functions corresponding to the truth tables for implication $\rightarrow$, bi-implication $\leftrightarrow$, reverse implication $\leftarrow$, and xor $\oplus$. For example, $\leftrightarrow$ corresponds to the equality `==` on Booleans.

```
(<->) :: Bool -> Bool -> Bool
p <-> q = p == q
```

To show the truth table for this operation on a Karnaugh map, we define a function of the four variables `a, g, r, b` that depends only on `a` and `r`.

```
eq a g r b = a == r
```

```
*CL3> km eq
   | 00 01 11 10
-- - -- -- -- --
00 |  1  1  0  0
01 |  1  1  0  0
11 |  0  0  1  1
10 |  0  0  1  1
```

5. The Haskell `if ...then ...else ...` expression (ITE) uses a Boolean parameter to choose between two values.
   (a) Check that for any Boolean values `a, b`

   ```
   (a || b) == if a then True else b
   (a == b) == if a then b else not b
   ```

   (b) Use ITE, `not, True, False`, in a similar way, to define haskell functions for the remaining binary connectives, $\wedge$, $\downarrow$, $|$. Google for the definitions of Sheffer stroke $|$ and Quine's dagger $\dagger$, or consult Appendix B-I in MML. Use km to check the truth tables of your implementations.
6. Find an existing Haskell function of type `Bool -> Bool -> Bool` with the right truth table for each of the operations, $\rightarrow$, $\leftarrow$, $\oplus$.

		Mathematical	MML	Haskell
truth values	true, false	$\top, \bot$	$1, 0$	`True, False`
negation	not	$\neg$	$\sim$	`not`
conjuction	and	$\wedge$	$\&$	`&&`
disjunction	or	$\vee$	$\vee$	`||`
implication	implies	$\rightarrow$	$\rightarrow$	
equivalence	iff	$\leftrightarrow$		
		$\leftarrow$		
	xor	$\oplus$		
Quine's dagger	nor	$\downarrow$	$\downarrow$	
Sheffer stroke	nand	$\mid$	$\mid$	

Constraints

You will now use Haskell and Karnaugh maps to discover how to express any Boolean function of four variables in a simple form. The questions concern Boolean expressions using (some or all of) the letters A, G, R, B. You should translate these into Haskell functions of four variables, and generate their Karnaugh maps.

7. Consider the Karnaugh maps for the expressions, $A \wedge R$, $G \vee B$. Compare these with the Karnaugh maps for the four propositional letters. Which of the following statements are correct?
   (a) A cell is labelled 1 in the Karnaugh map for a disjunction of two expressions iff it is labelled 1 in either (or both) of the kmaps for the constituent expressions.
   (b) A cell is labelled 0 in the Karnaugh map for a disjunction of two expressions iff it is labelled 0 in either (or both) of the kmaps for the constituent expressions.
   (c) A cell is labelled 0 in the Karnaugh map for a disjunction of two expressiosn iff it is labelled 0 in both of the kmaps for the constituent expressions.
   (d) A cell is labelled 1 in the Karnaugh map for a conjunction of two expressions iff it is labelled 1 in either (or both) of the kmaps for the constituent expressions.
   (e) A cell is labelled 1 in the Karnaugh map for a conjunction of two expressions iff it is labelled 1 in both of the kmaps for the constituent expressions.
   (f) A cell is labelled 0 in the Karnaugh map for a conjunction of two expressions iff it is labelled 0 in either (or both) of the kmaps for the constituent expressions.
   (g) A cell is labelled 0 in the Karnaugh map for a conjunction of two expressions iff it is labelled 0 in both of the kmaps for the constituent expressions.

A **literal** is either a propositional letter or its negation. We call a disjunction of literals a **clause**, or **constraint**. In this definition, we include the empty disjunction, equivalent to $\bot$. A conjunction of clauses is called a *conjunctive normal form* (CNF).

For example, this equation is in CNF:

$$(\neg A \vee \neg G \vee R) \wedge (A \vee G) \wedge \neg R \tag{1}$$

Equation (1) is a conjunction of three clauses. We say that the propositional letter A occurs negatively in $\neg A \vee \neg G \vee R$ and positively in $A \vee G$.

8. Consider the expression $(\neg A \vee R) \wedge (B \vee G)$ and the two subexpressions $(\neg A \vee R)$ and $(B \vee G)$).

   Compare the Karnaugh map for the expression with the Karnaiugh maps for the two subexpressions. You should find that the Karnaugh map for each subexpression has a block of zeros, and that the these two blocks of zeros cover the zeros of the Karnaugh map for the expression.

   For each subexpression the block of zeros corresponds to a clause, and the conjunction of these two clauses is the expression.

9. Observe that the 0s in the kmap for $\neg A \vee R$, and in the kmap for $G \vee B$, each form a square block *block* of cells.

   Experiment with the kmaps for various disjunctions of literals. **Give a definition** of **block** (or look one up) so that the 0s in the kmap for any disjunction of literals will form a block.

For the remainder of this tutorial we will focus on blocks of zeros, which correspond to disjunctive clauses.

10. Consider this k-map. Your task is to find an expression for the function `h` that generates this k-map.

    To do this you must find blocks that cover the zeros, find a clause corresponding to each block, and give the conjunction of these clauses as your answer.

```
*CL3> km h
   | 00 01 11 10
-- - -- -- -- --
00 |  0  0  1  1
01 |  0  0  0  1
11 |  1  0  0  1
10 |  1  1  1  1
```

    You can use Haskell to check that your answer really does give this k-map.

Since $\vee$ is associative and commutative, two clauses are equivalent if they mention the same literals. Any in which some letter occurs both positively and negatively is equivalent to $\top$ (for example, $A \vee \neg A \vee X$ and $B \vee \neg B$ are both equivalent to $\top$). We say such clauses are **trivial**. So every clause is either trivial – equivalent to $\top$, or is equivalent the disjunction of a (finite) set of literals that is not trivial.

We say that a constraint eliminates those states that make the corresponding disjunction false. The **constraint is satisified** by those states that make at least one of the literals in the constraint true. *The empty clause can never be satisfied* – there is no literal we can make true – it corresponds to $\bot$.

11. For each of the following examples – each is a conjunction of constraints – **show** that the 0s in the kmap can be described in terms of the blocks corresponding to the constituent disjunctions of literals.]

    (a) $(A \vee R) \wedge (\neg A \vee \neg R)$
    (b) $(A \vee R) \wedge (\neg G \vee \neg B)$
    (c) $(A \vee G) \wedge (R \vee B)$
    (d) $(A \vee \neg G) \wedge (R \vee B)$
    (e) $(A \vee \neg G \vee R \vee B)$
    (f) $(A \vee \neg G) \wedge (G \vee B)$
    (g) $(A \vee R) \wedge (G \vee B)$

12. For each of the following kmaps find a set of blocks of 0s that cover all the 0s on the map. Find the disjunction of literals that corresponds to each of these blocks. Write down a conjunction of disjunctions of literals whose Boolean function corresponds to the original kmap. Use km to test your answers.

    (a)

	00	01	11	10
00	0	0	1	1
01	0	0	0	1
11	1	0	0	1
10	1	0	1	1

	00	01	11	10
00	0	0	1	1
01	0	0	0	1
11	1	1	0	1
10	1	1	1	1

    (b)

	00	01	11	10
00	0	0	1	1
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

	00	01	11	10
00	0	0	1	0
01	0	0	0	1
11	1	0	0	1
10	0	0	1	0

13. How many non-equivalent clauses are there for a system with n propositional letters?
    Hint: In any non-trivial clause, each letter occurs either positively or negatively, or not at all. First count the non-trivial clauses, then add the trivial one.

An expression is in conjunctive normal form (CNF) if it is a *conjunction of constraints*, where each constraint is a disjunction of literals.

We say that a non trivial constraint eliminates those states that make the corresponding disjunction false. The constraint is satisified by those states that make at least one of the literals in the constraint true. The trivial constraint $\top$ is satisfied by every state; the impossible constraint $\bot$ eliminates every state.

A state satisfies some CNF iff it statisfies all of the conjoined constraints.

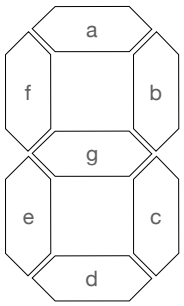
A state is eliminated by some CNF iff it is eliminated by at least one of the conjoined constraints.

Karnaugh maps are routinely used in the design of logic circuits. In the tutorial. You will work as a group to design the logic to drive a seven-segment display – set of LED (or LCD) segments that render numerals 0 through 9 depending on a four-bit input, as shown below.

14. (a) Complete the table to show which segments should be on (1) or off (0), for each combination of inputs. Column (a) has been filled in to show that the (a) segment should be on except when the input represents 1 or 4.

Digit	Display
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9

digit	AGRB	a	b	c	d	e	f	g
0	0000	1						
1	0001	0						
2	0010	1						
3	0011	1						
4	0100	0						
5	0101	1						
6	0110	1						
7	0111	1						
8	1000	1						
9	1001	1						



		AG			
	a	00	01	11	10
RB	00	1	0	X	1
	01	0	1	X	1
	11	1	1	X	X
	10	1	1	X	X

(b) The Karnaugh map is filled in from the (a) column.
X represents an unspecified output – your logic may produce 0 or 1.
Give a CNF for the logic to drive the (a) segment.

Tutorial Activities

1. (m) Compare your answers to questions 1-3 with a buddy, then check that your group agrees on the answers.

   Ask one of the tutors if you have questions.

2. Logic for a seven-segment decoder.

   (a) First check that you all agree on the truth table, and the logic for the (a) segment you constructed in exercise 14.

   (b) As a group, you will need to construct a Karnaugh map for the six remaining segments. Individually, you should each construct maps for two or three different segments, using the templates provided below. Make sure you allocate the segments so that each is tackled by at least two people independently – this will allow you as a group to cross-check your answers. Each Karnaugh map is labelled in the top-left corner with the letter of a segment. It will help you to cross-check your work if you use the right map for each segment.

   We have not specified which segments shoule be lit when the four-bit binary input represents a number in the range $10 - -15$. For these inputs you should enter an X in the Karnaugh map to represent a *don't care* output. When you come to design the logic, you can treat these don't cares as either 0s or 1s – whichever makes life easier.

   (c) For each of your Karnaugh maps you should identify a CNF for the logic driving that segment of the display. Each clause of the CNF corresponds to a block of squares that includes no 1 entries on the Karnaugh map. Taken together the blocks must cover all of the 0 entries. We don't care whether the X entries fall within or outwith the blocks.

   Use the space below the KM to list the clauses – use a pencil!

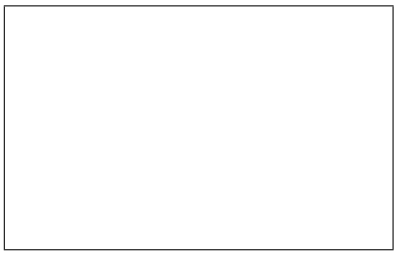
   Cross-check your answer for each of your segments with those of the others in your group who have also worked on that segment.

   (d) Although each segment requires its own logic, some clauses may be shared between different segments. When this happens we can simplify the decoder.

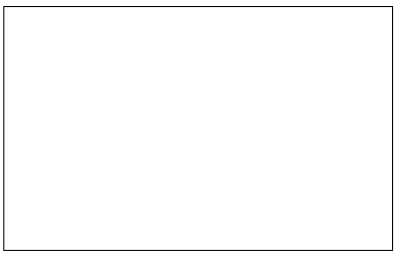
   Look again at your four segments. Can you share clauses between different segments? What is the smallest number of clauses you can find that can be combined to drive all four segments?

   (e) Now work as a group to find the smallest number of clauses you can find to drive all seven segments.

		AG			
	b	00	01	11	10
RB	00				
	01				
	11				
	10				

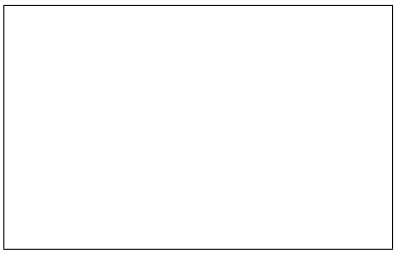


		AG			
	e	00	01	11	10
RB	00				
	01				
	11				
	10				



		AG			
	c	00	01	11	10
RB	00				
	01				
	11				
	10				

		AG			
	f	00	01	11	10
RB	00				
	01				
	11				
	10				



		AG			
	d	00	01	11	10
RB	00				
	01				
	11				
	10				

		AG			
	g	00	01	11	10
RB	00				
	01				
	11				
	10