Université Larbi Ben M'hidi

Oum El Bouaghi

FACULTY OF EXACT SCIENCES AND SCIENCE OF THE NATURE OF LIFE

Department of Mathematics and Computer Science

Report

Logic Gates Implementation

With Logic Programming

Made by:

Bahloul younes

Kabour oussama

Supervised by:

Dr Mellal Nassima

table of content

General Introduction

This paper looks to establish the relation between logic programming and basic logic gates. Ranging from the very basic nand to three-state circuit up to a full ALU

All the coding was done using prolog as the paper explains how to convert from a truth table to horns

Chapter Ⅰ :

Logic Programming

Ⅰ.1. Introduction:

Logic programming is a type of programming paradigm which is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain.

Ⅰ.2. History of Logic Programming:

The use of mathematical logic to represent and execute computer programs is also a feature of the lambda calculus, developed by Alonzo Church in the 1930s. However, the first proposal to use the clausal form of logic for representing computer programs was made by Cordell Green. This used an axiomatization of a subset of LISP, together with a representation of an input-output relation, to compute the relation by simulating the execution of the program in LISP. Foster and Elcock's Absys, on the other hand, employed a combination of equations and lambda calculus in an assertional programming language which places no constraints on the order in which operations are performed.

Logic programming in its present form can be traced back to debates in the late 1960s and early 1970s about declarative versus procedural representations of knowledge in Artificial Intelligence. Advocates of declarative representations were notably working at Stanford, associated with John McCarthy, Bertram Raphael and Cordell Green, and in Edinburgh, with John Alan Robinson (an academic visitor from Syracuse University), Pat Hayes, and Robert Kowalski.

Advocates of procedural representations were mainly centered at MIT, under the leadership of Marvin Minsky and Seymour Papert.[citation needed] Although it was based on the proof methods of logic, Planner, developed at MIT, was the first language to emerge within this proceduralist paradigm. Planner featured pattern-directed invocation of procedural plans from goals (i.e.

goal-reduction or backward chaining) and from assertions (i.e. forward chaining).

The most influential implementation of Planner was the subset of Planner, called Micro-Planner, implemented by Gerry Sussman, Eugene Charniak and Terry Winograd. It was used to implement Winograd's natural-language understanding program SHRDLU, which was a landmark at that time. To cope with the very limited memory systems at the time, Planner used a backtracking control structure so that only one possible computation path had to be stored at a time. Planner gave rise to the programming languages QA-4, Popler, Conniver, QLISP, and the concurrent language Ether.

Hayes and Kowalski in Edinburgh tried to reconcile the logic-based declarative approach to knowledge representation with Planner's procedural approach. Hayes (1973) developed an equational language, Golux, in which different procedures could be obtained by altering the behavior of the theorem prover. Kowalski, on the other hand, developed SLD resolution, a variant of SL-resolution, and showed how it treats implications as goal-reduction procedures. Kowalski collaborated with Colmerauer in Marseille, who developed these ideas in the design and implementation of the programming language Prolog.

The Association for Logic Programming was founded to promote Logic Programming in 1986.

Ⅰ.3. Why Logic Programming:

Logic programming can be viewed as controlled deduction. An important concept in logic programming is the separation of programs into their logic component and their control component. With pure logic programming languages, the logic component alone determines the solutions produced. The control component can be varied to provide alternative ways of executing a logic program. This notion is captured by the slogan

Algorithm = Logic + Control

where "Logic" represents a logic program and "Control" represents different theorem-proving strategies.

Problem solving:

In the simplified, propositional case in which a logic program and a top-level atomic goal contain no variables, backward reasoning determines an and-or tree, which constitutes the search space for solving the goal. The top-level goal is the root of the tree. Given any node in the tree and any clause whose head matches the node, there exists a set of child nodes corresponding to the sub-goals in the body of the clause. These child nodes are grouped together by an "and". The alternative sets of children corresponding to alternative ways of solving the node are grouped together by an "or".

Any search strategy can be used to search this space. Prolog uses a sequential, last-in-first-out, backtracking strategy, in which only one alternative and one sub-goal is considered at a time. Other search strategies, such as parallel search, intelligent backtracking, or best-first search to find an optimal solution, are also possible.

In the more general case, where sub-goals share variables, other strategies can be used, such as choosing the subgoal that is most highly instantiated or that is sufficiently instantiated so that only one procedure applies. Such strategies are used, for example, in concurrent logic programming.

Ⅰ.4. Modern day uses:

Logic programming is widely used in parsing, both in natural languages and programming languages. Using Definite Clause Grammars in SWI-Prolog is a good tutorial to learn DCG in SWI-Prolog.

Since the creator of logic programming is also an linguist, it once was widely used in natural language processing(NLP), mostly in parsing natural language processing and generating natural language. Prolog and Natural language analysis is a book about this topic. But nowadays, NLP is mostly occupied by statistic approach.

Since it is really easy to write an parser in Prolog, it is also quite often used to implement new programming. The first interpreter of Erlang Programming Language is written in Prolog. You can read Joe Armstrong origin paper Use Prolog to implement an new programming languages . Or you can check Markus Triska Lisprolog - Interpreter for a simple Lisp, written in Prolog to see how to write an lisp interpreter in 100-200 lines of code.

Logic programming is also used widely when there are a lot of relations, like in semantic web's RDF manipulation

Ⅰ.5. Examples:

solving the Soduku using the clpfd library

```
:- use_module(library(clpfd)).

sudoku(Rows) :-
        length(Rows, 9), maplist(length_(9), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
maplist(all_distinct, Columns),
        Rows = [A,B,C,D,E,F,G,H,I],
        blocks(A, B, C), blocks(D, E, F), blocks(G,
H, I).

length_(L, Ls) :- length(Ls, L).

blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
        all_distinct([A,B,C,D,E,F,G,H,I]),
        blocks(Bs1, Bs2, Bs3).

problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```

Chapter Ⅱ :

Logic Gates

Ⅱ.1. Introduction:

In electronics, a logic gate is an idealized or physical device implementing a Boolean function; that is, it performs a logical operation on one or more binary inputs and produces a single binary output. Depending on the context, the term may refer to an ideal logic gate, one that has for instance zero rise time and unlimited fan-out, or it may refer to a non-ideal physical device (see Ideal and real op-amps for comparison).

Ⅱ.2.Electronic gates:

To build a functionally complete logic system, relays, valves (vacuum tubes), or transistors can be used. The simplest family of logic gates using bipolar transistors is called resistor–transistor logic (RTL). Unlike simple diode logic gates (which do not have a gain element), RTL gates can be cascaded indefinitely to produce more complex logic functions. RTL gates were used in early integrated circuits. For higher speed and better density, the resistors used in RTL were replaced by diodes resulting in diode–transistor logic (DTL). Transistor–transistor logic (TTL) then supplanted DTL. As integrated circuits became more complex, bipolar transistors were replaced with smaller field-effect transistors (MOSFETs); see PMOS and NMOS. To reduce power consumption still further, most contemporary chip implementations of digital systems now use CMOS logic. CMOS uses complementary (both n-channel and p-channel) MOSFET devices to achieve a high speed with low power dissipation.

For small-scale logic, designers now use prefabricated logic gates from families of devices such as the TTL 7400 series by Texas Instruments, the CMOS 4000 series by RCA, and their more recent descendants. Increasingly, these fixed-function logic gates are being replaced by programmable logic devices, which allow designers to pack a large number of mixed logic gates into a single integrated circuit. The field-programmable nature of programmable logic devices such as FPGAs has reduced the 'hard' property of hardware; it is now possible to change the logic design of a hardware system by reprogramming some of its components, thus allowing the features or function of a hardware implementation of a logic system to be changed.

Ⅱ.3.History and development:

The binary number system was refined by Gottfried Wilhelm Leibniz (published in 1705), influenced by the ancient I Ching's binary system.

Leibniz established that, by using the binary system, the principles of arithmetic and logic could be combined.

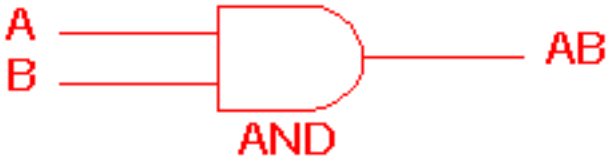
In an 1886 letter, Charles Sanders Peirce described how logical operations could be carried out by electrical switching circuits. Eventually, vacuum tubes replaced relays for logic operations. Lee De Forest's modification, in 1907, of the Fleming valve can be used as a logic gate. Ludwig Wittgenstein introduced a version of the 16-row truth table as proposition 5.101 of Tractatus Logico-Philosophicus (1921). Walther Bothe, inventor of the coincidence circuit, got part of the 1954 Nobel Prize in physics, for the first modern electronic AND gate in 1924. Konrad Zuse designed and built electromechanical logic gates for his computer Z1 (from 1935–38).

From 1934 to 1936, NEC engineer Akira Nakashima introduced switching circuit theory in a series of papers showing that two-valued Boolean algebra, which he discovered independently, can describe the operation of switching circuits. His work was later cited by Claude E. Shannon, who elaborated on the use of Boolean algebra in the analysis and design of switching circuits in 1937. Using this property of electrical switches to implement logic is the fundamental concept that underlies all electronic digital computers. Switching circuit theory became the foundation of digital circuit design, as it became widely known in the electrical engineering community during and after World War II, with theoretical rigor superseding the ad hoc methods that had prevailed previously.

Ⅱ.4. Symbols:

There are two sets of symbols for elementary logic gates in common use, both defined in ANSI/IEEE Std 91-1984 and its supplement ANSI/IEEE Std 91a-1991.

AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

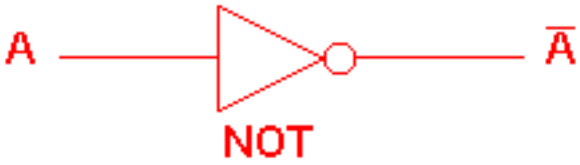
The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

OR gate

2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

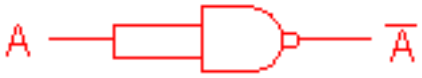
The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

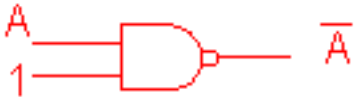
NOT gate



NOT gate	
A	$\overline{A}$
0	1
1	0

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.





NAND gate

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

NOR gate

2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

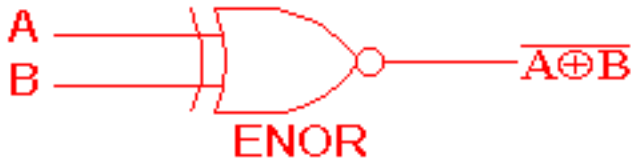
This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if **any** of the inputs are high.
The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

EXOR gate

2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign is used to show the EOR operation.

EXNOR gate



2 Input EXNOR gate		
A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

The '**Exclusive-NOR**' gate circuit does the opposite to the EOR gate. It will give a low output if **either, but not both**, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

Ⅱ.6. De Morgan equivalent:

Not (A *and* B) is the same as *Not* A *or Not* B.

Not (A *or* B) is the same as *Not* A *and Not* B.

By use of De Morgan's laws, an *AND* function is identical to an *OR* function with negated inputs and outputs. Likewise, an *OR* function is identical to an *AND*

function with negated inputs and outputs. A NAND gate is equivalent to an OR gate with negated inputs, and a NOR gate is equivalent to an AND gate with negated inputs.

Chapter Ⅲ: Logic Gates implementation in prolog

Ⅲ.1. Prolog:

Prolog is a logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is intended primarily as a declarative programming language: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.

The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Philippe Roussel.

Prolog was one of the first logic programming languages, and remains the most popular among such languages today, with several free and commercial implementations available. The language has been used for theorem proving, expert systems, term rewriting, type systems, and automated planning,as well as its original intended field of use, natural language processing. Modern Prolog environments support the creation of graphical user interfaces, as well as administrative and networked applications.

Ⅲ.2. Basic Gates:

```
NOT Gate
/*************
   NOT
 ____ _______
|    |       |
|  A-| NOT   |-OUT
|____|_______|
*************/
%   A   OUT
not(0,   1 ).
not(1,   0 ).

NAND Gate

/*************
   NAND Gate
 ____ _______
|  A-|       |
|    |NAND   |-OUT
|  B-|_______|
*************/
%   A   B   OUT
```

```
nand(0, 0, 1 ).
nand(0, 1, 1 ).
nand(1, 0, 1 ).
nand(1, 1, 0 ).
```

AND Gate

```
/*************
   AND Gate

   ____ _______
  A-|      |
    | AND  |-OUT
  B-|______|
*************/
%   A  B  OUT
and(0, 0, 0 ).
and(0, 1, 0 ).
and(1, 0, 0 ).
and(1, 1, 1 ).
```

OR Gate

```
/*************
   OR Gate

   ____ ______
  A-|    |
    | OR |-OUT
  B-|____|
*************/
%  A  B  OUT
or(0, 0, 0 ).
or(0, 1, 1 ).
or(1, 0, 1 ).
or(1, 1, 1 ).
```

XOR Gate

```
/*************
   XOR Gate

   ____ _______
  A-|      |
    | XOR  |-OUT
  B-|______|
*************/
%   A  B  OUT
xor(0, 0, 0 ).
xor(0, 1, 1 ).
xor(1, 0, 1 ).
xor(1, 1, 0 ).
```

Ⅲ.3. Three-state logic gate: MUX Gate

```
MUX Gate

/********

    MUX Gate

    SEL
     |
  A-|    |
    | MUX |-OUT
  B-|    |
 ********/
%   A B SEL OUT

mux(A,B,0,   A).
mux(A,B,1,   B).

DMUX Gate
/********

 DMUX Gate

    SEL
     |
   |    |-A
 IN-| DMUX|
   |    |-B
 ********/
%    IN SEL A B

dmux(IN,0,  IN,0).
dmux(IN,1,  0,IN).
```

Ⅲ.4. Compound Logic Gates:

```
/*************
   Half Adder Gate
 ____________
  A-|        |-sum
    | hadd   |
  B-|________|-carry
*************/
%   A  B  sum carry

halfadder(A,B,S,C) :-
      xor(A, B,  S ),
      and(A, B,  C ).
/*************
   Full Adder Gate
 ____________
  A-|        |-sum
  c-| fadd   |
  B-|________|-carry
*************/
%   A  B  sum carry

fulladder(A,B,C,S,Carry) :-
      halfadder(A, B,  AB,Cab ),
      halfadder(AB, C,  S, Cs ),
      or(Cab,Cs,Carry).
```

Ⅲ.5. ALU:

```
/**
 * The ALU. Computes one of the following functions:
 * x+y, x-y, y-x, 0, 1, -1, x, y, -x, -y, !x, !y,
 * x+1, y+1, x-1, y-1, x&y, x|y on two 16-bit inputs,
 * according to 6 input bits denoted zx,nx,zy,ny,f,no.
 * The bit-combinations that yield each function are
 * documented in the book. In addition, the ALU
 * computes two 1-bit outputs: if the ALU output
 * is 0, zr is set to 1; otherwise zr is set to 0;
 * If out<0, ng is set to 1; otherwise ng is set to 0.
 */

// Implementation: the ALU manipulates the x and y
// inputs and then operates on the resulting values,
// as follows:
// if (zx==1) set x = 0        // 16-bit constant
```

```
// if (nx==1) set x = ~x        // bitwise "not"
// if (zy==1) set y = 0         // 16-bit constant
// if (ny==1) set y = ~y        // bitwise "not"
// if (f==1)  set out = x + y   // integer 2's complement addition
// if (f==0)  set out = x & y   // bitwise "and"
// if (no==1) set out = ~out    // bitwise "not"
// if (out==0) set zr = 1
// if (out<0) set ng = 1


CHIP ALU {
    IN
        x[16], y[16],  // 16-bit inputs
        zx, // zero the x input?
        nx, // negate the x input?
        zy, // zero the y input?
        ny, // negate the y input?
        f,  // compute  out = x + y (if 1) or out = x & y (if 0)
        no; // negate the out output?

    OUT
        out[16], // 16-bit output
        zr, // 1 if (out==0), 0 otherwise
        ng; // 1 if (out<0),  0 otherwise
}
```

Final conclusion:

Noting the earlier chapters we managed to establish a close relation between logic Programming and logic gates

From the implementation we can conclude that logic programming can convert basic truth tables into horns and using language properties it can work up to more complex relations

The Prolog language features made extremely easy to make this conversion it really seems like end to end conversion

References :

Chapter I:

https://en.wikipedia.org/wiki/Logic_programming

Chapter II:

https://en.wikipedia.org/wiki/Logic_gate

Logic gates HDL code:

http://people.duke.edu/~nts9/logicgates/