2D Project Report

Part 1-2. Translation of Netlist to CNF

Team members: MA Ke, WEI Fanding, CHEN Jian, CHEN Ziyi

Introduction

In this part, we are required to translate the netlist of our optimized adder circuit into Conjunction Normal Form (CNF) which is the input format of the SAT solver. This task can be mainly divided into two separate parts: the first is to translate the circuit netlist into Boolean expressions, and the second is to translate the Boolean expressions into CNF format. Given the limited time and the large workload, we decided to do the first task manually and write a simple program to solve the second task.

Selection of Test Bits

Because it is very trivial to test every output bit of the adder, we just select two representative bits for verification. Obviously, the selection of test bits is related to the architecture of our optimized adder, so we will briefly introduce our optimized adder first.

Adder Optimization

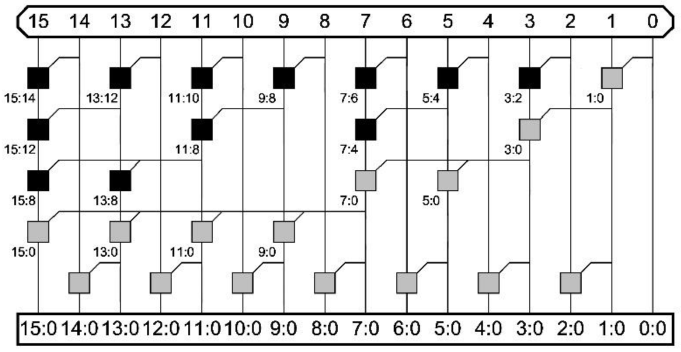
According to the 2D handout, there are several ways to improve the performance of the adder, like minimizing load-dependent delay and using inverting logic, etc. All of these approaches will not influence the main logic of the adder except using different adder architectures. And our goal is to verify whether the optimized logic do the exactly same thing as the original one. Therefore we will focus on the adder architecture in this section.

It is widely recognized that the parallel prefix adders are the fastest combinational adders compared with other adders such as carry-select adders and carry-lookahead adders. This conclusion is also proved by our simulations. However, there are many variations of parallel prefix adders, and each of them has some slight differences from others in aspects of logic depth, fan-out and wire connections. We implement some typical and classical parallel prefix adders in Jsim in order to test their propagation delay and calculate their area. The results are shown below.

Adder Architecture	*Circuit Area (microns^2)*	*Min Clock Cycle (ns)*	*Product*
Kogge-Stone	9990.0	2.0	19980.0
Han-Carlson	7883.0	2.1	16554.3
Brent-Kung	6894.0	2.4	16545.6
Ladner-Fischer	7152.0	2.1	15019.2
Sklansky	7883.0	2.2	17342.6

Note that the circuit areas exclude the areas of input and output registers. Thus we find that Ladner-Fischer adder is the one that meets the requirements best, and we choose this architecture as our optimized adder.

Ladner-Fischer adder is also based on the basic ideas of carry-lookahead adders. First to calculate the propagate signals and generate signals of all input bits by a specific circuit block called Module A. Then to propagate the P/G signals of every single bit to calculate the P/G signals from Bit 0 to every specified bit by another specific circuit block called Module B. Finally to calculate the carry signals and output sum bits according to the P/G signals. Of course there are many layers of Module B to generate the correct P/G signals, and the differences between parallel prefix adders lie in the layer arrangements and wire connections.



The configuration of a 16-bit Ladner-Fischer adder is shown in the above diagram. The P/G signals from Bit 0 to the odd bits are generated by a tree. This tree is not exactly a binary tree, because it adds some nodes to generate the P/G signals of the intermediate bits. Then the P/G signals from Bit 0 to the even bits are generated by the final stage according to the P/G signals of the odd bits.

Test Bits Chosen

We can easily draw a conclusion that the odd bits and the even bits are not treated equally in this architecture. So the test bits must include one odd bit and one even bit.

What’s more, the more significant bits definitely travel longer distances to reach the output than the less significant bits. Therefore we’d better include one relatively significant bit and one relatively insignificant bit.

Considering both the two aspects, we finally choose Bit 7 and Bit 24 as our test bits.

Translation from Boolean Expressions to CNF Format

We write a simple program to solve the task of translation from Boolean expressions to CNF format file. Because there are some requirements for the input of this program, that is, the Boolean expressions should meet some formats, we introduce this part before translation from netlist to Boolean expressions.

Usage of the Program

The program simply takes a text file as input and gives a CNF file as output. As the time is limited, we do not design some user interface for this program. It just looks for “CNFfiles” folder under the project directory, and reads “input.txt” as input. After some processing, it generates a CNF file named “output.cnf” in the same folder.

The input files must meet some requirements. The input files can only include Boolean equations and empty lines, and any other strings are not allowed. One Boolean equation should take up a new line. One Boolean equation must have only one variable on the left of the equal mark, and the Boolean expression is on the right of the equal mark. The Boolean expressions consists of variables, operators and parentheses. Spaces in Boolean expressions will be ignored.

The variable names can be various, and they can include letters, digits and signs except the pre-defined operators (and parentheses). This program is case-sensitive, so two variables with names of the same letters and sequence, but in different cases, are treated as two independent variables.

The operators can be * (AND), + (OR), ^ (XOR) and ! (NOT). The former three operators take two operands and the latter one operator takes only one operand.

The parentheses are important in the format. The order of operations must always be made explicit with parentheses. That is, the sub-expression between two parentheses can only include two sub-expressions (or variables) and one operator. The ! operator can occur before a variable or a sub-expression (before the left parenthesis).

Based on the above principles, here are some examples to demonstrate what is legal and illegal for the program.

Legal Format:	C = A ^ B	D = (A * B) * C
	D = !(A + !B) * C	Xmb4_0.Gbar = !((P4_5 * G0_3) + G4_5)
Illegal Format:	C = A & B [*illegal operator*]	D = A * B * C [*implicit order*]
	!D = !(A + !B) * C [*operator occurring on the left*]	
	Xmb4_0.Gbar = !((P4+5 * G0+3) + G4+5) [*illegal variable name*]	

Since we have little time to perfect the program, please note that any illegal operations may cause exception.

Main Elements of the Program

The program is written in Java language, and it consists of 5 classes, that is, *Variable* class, *Expression* class, *CNFWriter* class, *Converter* class and *Main* class.

Variable class holds the variable data. It has only one attribute named *name*, and one method named *toString* which simply returns its name.

Expression class holds the expression (including sub-expression) data. It may contain only one *Variable*, or one or two *Expression*. It may contains operator AND or OR, and may not contain operator (if it only contains one variable or expression). Note that XOR operator is excluded because it is represented by AND, OR and NOT in the data. It may be the negation. Also it can have a name. The method *parse* gets a Boolean equation string stated above, and generates the

Expression data structure. The method *toString* returns the whole expression excluding the result variable, and uses many parentheses to indicate the variables. For example, the *toString* result for “C=A+B” is “(A)+(B)”. The method *toCNFString* returns the CNF expression derived from the Boolean equation. For example, the *toCNFString* result for “C=A+B” is “(¬ C∨A∨B)∧(¬ A∨C)∧(¬ B∨C)”. It may also generates some intermediate variables. For example, the result for “D=(A+B)*C” is “(¬ D∨[D_0])∧(¬ D∨C)∧(¬ [D_0]∨¬ C∨D)∧(¬ [D_0]∨A∨B)∧(¬ A∨[D_0])∧(¬ B∨[D_0])”, and “[D_0]” is the intermediate variable which is equal to “A+B”.

CNFWriter class simply take one CNF expression string as input, and returns the formatted CNF strings. For example, use “(¬ C∨A∨B)∧(¬ A∨C)∧(¬ B∨C)” as input and it will return the following lines.

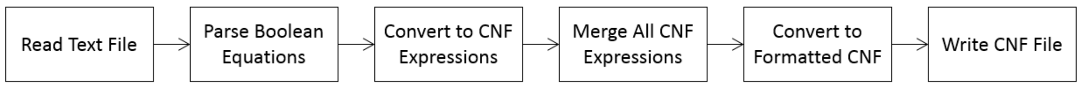
```
c VARIABLE 1: C
c VARIABLE 2: A
c VARIABLE 3: B
p cnf 3 3
-1 2 3 0
-2 1 0
-3 1 0
```

Converter class has only one behavior named *convertBoolExpsToCNF*. It takes a text file as input and writes the formatted CNF strings into a CNF file. What it does is to check the input file line by line, translate all Boolean equations into CNF expression strings by *Expression.toCNFString*, then connect all CNF expression strings with connector “∧”, and finally translate the whole CNF expression string into formatted CNF strings using *CNFWriter* and write it into a CNF file.

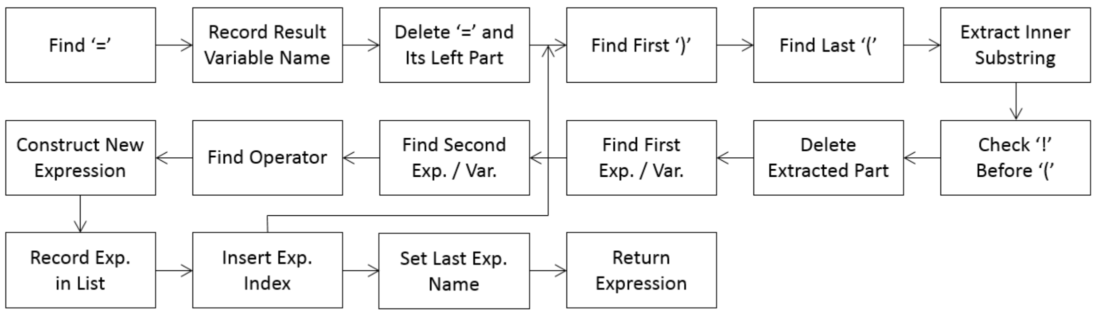
Main class is the entrance of the program, and just specifies "CNFfiles/input.txt" as the input of *Converter*, and "CNFfiles/output.cnf" as the output.

Work Flow of the Program

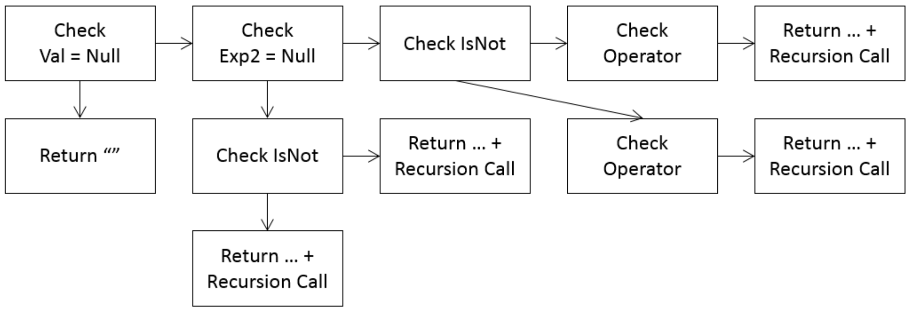
We can easily draw a work flow chart from the above introduction.



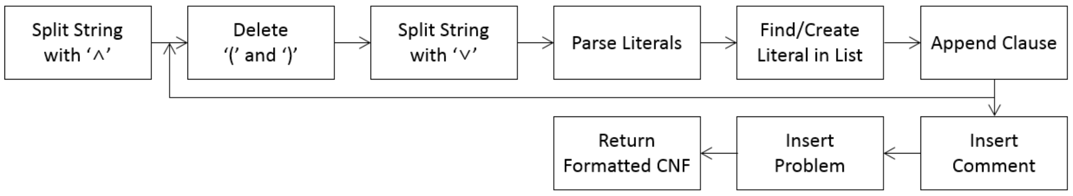
Three key process may be parsing of Boolean equation, translation from Boolean equation to CNF expression and translation from CNF expression to formatted CNF. We just zoom them in and get the following flow charts. Work flow for parsing of Boolean equation is shown below.



Work flow for translation from Boolean equation to CNF expression is shown below. It uses the recursion design approach.


Check Val = Null
Check Exp2 = Null
Check IsNot
Check Operator
Return … + Recursion Call
Return “”
Check IsNot
Return … + Recursion Call
Check Operator
Return … + Recursion Call
Return … + Recursion Call

Work flow for translation from CNF expression to formatted CNF is shown below.



The above flow charts only show the main idea of these processes, and the details are emitted. Please refer to the source code for more detailed information.

Tests for the Program

We have done many tests for the program, and it is proved that this program is able to give correct output when given legal input. Here we use the example from 2D handout to demonstrate the test process.

The example is an OR gate, and its Boolean equation is “o = a + b”.

a) Write “o = a + b” into “input.txt”;

b) Run the program; and

c) Check the result in “output.cnf”.

The content of the resulting CNF file is shown below (left). The example CNF is also shown below (right). They indicate the exactly same CNF expression, except that the orders of the variables are different.

```
c VARIABLE 1: o
c VARIABLE 2: a
c VARIABLE 3: b
p cnf 3 3
-1 2 3 0
-2 1 0
-3 1 0
```

```
c test OR gate
p cnf 3 3
-3 1 2 0
-1 3 0
-2 3 0
```

Translation from Circuit Netlist to Boolean Expressions

We finish the translation from circuit netlist to Boolean expression manually. Because the configuration of the adder/subtractor unit is almost the same for different adder architectures, we only focus on the adders themselves. We compare Bit 7 and Bit 24 of the output of a Ladner-Fischer adder and a ripple-carry adder.

Converting Gates to Boolean Equation

The adders implemented in Jsim are all composed of various gates. So we should translate the gates into Boolean equations. Note that the translated Boolean equations must meet the requirements of the program.

The used gates and their corresponding Boolean equations are listed below. Because the inverting logic has better performance, we do not use any AND/OR gate. Note that gates with more than 2 inputs should be carefully translated into Boolean equations in order to meet the requirements.

Gate Statement	*Boolean Equation*
.connect a z	z = a
Xid a z inverter	z = !a
Xid a b z nand2	z = !(a*b)
Xid a b c z nand3	z = !(a*(b*c))
Xid a b z xor2	z = a^b
Xid a1 a2 b z aoi21	z = !((a1*a2)+b)

We may have trouble in translation when meeting the user-defined sub-circuits. We will deal with the sub-circuits later, so we should only write some mnemonic equations for now. One possible equation format is shown below.

```
Xid in0 in1 … out0 out1 … subckt_name    =>    (out0,out1,…) = subckt_name(in0,in1,…)[Xid]
```

Now we can start to translate our netlist. Because we do not need to translate every output bit, it is recommended that we start from the specified output bit, and then repeat looking for its relevant signals until finding the input bits.

Take the 32-bit ripple-carry adder as an example. The netlist is shown below. The *A[31:0]*, *B[31:0]* and *C0* are inputs, and *s[31:0]* and *c32* are outputs.

```
.connect c0 C0
XFA A[31:0] B[31:0] c[31:0] s[31:0] c[32:1] FA
```

First, we should deal with the iterators.

```
.connect c0 C0
XFA#0 A31 B31 c31 s31 c32 FA
XFA#1 A30 B30 c30 s30 c31 FA
…
XFA#31 A0 B0 c0 s0 c1 FA
```

Then, we should find the specified output bit, e.g. *s7*.

```
XFA#24 A7 B7 c7 s7 c8 FA
```

We can translate this statement first.

```
(s7,c8) = FA(A7,B7,c7)[XFA#24]
```

And we can find the relevant signal *c7*. We only focus on the undetermined input signals. Because *A7* and *B7* are input bits, we do not any extra expression to represent them.

```
XFA#25 A6 B6 c6 s6 c7 FA
```

We can finish our translation in this way. The translated Boolean equations are listed below.

```
(s7,c8) = FA(A7,B7,c7)[XFA#24]
(s6,c7) = FA(A6,B6,c6)[XFA#25]
(s5,c6) = FA(A5,B5,c5)[XFA#26]
(s4,c5) = FA(A4,B4,c4)[XFA#27]
(s3,c4) = FA(A3,B3,c3)[XFA#28]
(s2,c3) = FA(A2,B2,c2)[XFA#29]
(s1,c2) = FA(A1,B1,c1)[XFA#30]
(s0,c1) = FA(A0,B0,c0)[XFA#31]
c0 =C0
```

Dealing with Sub-Circuits

In order to complete the Boolean equations of the adders, we have to deal with the sub-circuits. The only sub-circuit used in the ripple-carry adder is *FA* (Full Adder). And the sub-circuits used in the Ladner-Fischer adder are *adder_cl_A* (Carry-Lookahead Adder Module A) and *adder_cl_B* (Carry-Lookahead Adder Module B).

We can regard sub-circuits as templates. We should also translate the sub-circuit netlist into Boolean equations first. Take the *FA* sub-circuit as an example. The netlist is shown below. The *a*, *b* and *ci* are inputs, and *s* and *co* are outputs.

```
.subckt FA a b ci s co
Xxor1 a b abx xor2
Xxor2 abx ci s xor2
Xnand1 a b abna nand2
Xnand2 a ci acna nand2
Xnand3 b ci bcna nand2
Xnand4 abna acna bcna co nand3
.ends
```

The translated Boolean equations are shown below.

```
abx = a^b
s = abx^ci
abna = !(a*b)
```

*acna = !(a*ci)*
*bcna = !(b*ci)*
co = !(abna(acna*bcna))*

Then we can use this *FA* sub-circuit template in the adder circuit. To use the template, we should first substitute the input and output variables with the real signal names, and then add "Xid." before intermediate variables. Take the below mnemonic equation as an example.

(s7,c8) = FA(A7,B7,c7)[XFA#24]

This mnemonic equation should be replaced by the following Boolean equations.

XFA#24.abx = A7^B7
s7 = XFA#24.abx^c7
*XFA#24.abna = !(A7*B7)*
*XFA#24.acna = !(A7*c7)*
*XFA#24.bcna = !(B7*c7)*
c8 = !(XFA#24.abna(XFA#24.acna*XFA#24.bcna))*

Note that all mnemonic equations should be replace by the real Boolean equations using the sub-circuit templates. Then we can get the whole Boolean equation list for one output bit of an adder.

Matters about Translation

We usually translate two adders separately and then combine them together to test whether the corresponding output bits do the same work. In the final Boolean equation file, the two adders share the same input bits *A[31:0]*, *B[31:0]* and *C0*, but produce different output bits. Thus we should pay attention to the variable names in the two adder circuits. That is, they only have common input bits, but names of their intermediate and output signal are all different. In order to distinguish between the two adder circuits, we use uppercase variable names in the Ladner-Fischer adder circuit and lowercase variable names in the ripple-carry adder circuits.

Then we can combine the two Boolean equation list together. Before translating the combined file into formatted CNF file using the program, we should add one more equation in the file.

Result = Sx^sx

Sx is the x_{th} output bit of the Ladner-Fischer adder, and *sx* is the x_{th} output bit of the ripple-carry adder. If they always produce the same value under the same inputs, *Result* should always be false.

Now we can write the whole Boolean equation list into "input.txt", and produce "output.cnf" using our program.

Combinational Equivalence Checking

Our final goal of this project is to do the Combinational Equivalence Checking (CEC) for the two adders.

Before the Test

We have already got the formatted CNF file “output.cnf”. Actually we can use it as the input for the SAT solver now, but the SAT solver may return a large number of solutions indicating that *Result* variable is always false. So we can simply add one clause in the CNF file to solve this problem. Assume that *Result* variable corresponds to “1” in the CNF file.

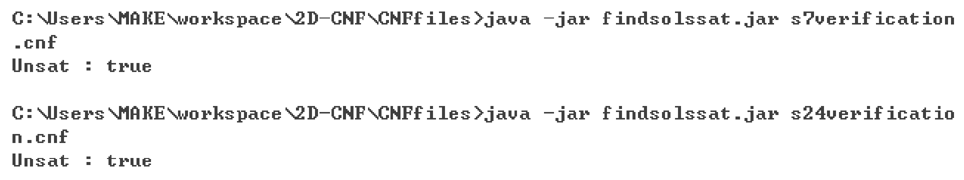
1 0

This clause indicates that *Result* variable should be true. Of course it is impossible, so the SAT solver will return something indicating this CNF problem is unsatisfiable.

Don’t forget to add 1 in the variable counter in the header of the CNF file. It is the last number in the line starting with “p”.

Test Results

We have got two test files from the above steps, “s7verification.cnf” and “s24verification.cnf”. We first use “findsolssat.jar” provided by the instructors to solve these files.



```
C:\Users\MAKE\workspace\2D-CNF\CNFfiles>java -jar findsolssat.jar s7verification
.cnf
Unsat : true

C:\Users\MAKE\workspace\2D-CNF\CNFfiles>java -jar findsolssat.jar s24verificatio
n.cnf
Unsat : true
```

It shows that these two problems are unsatisfiable, which is just the expected result.

Then we use our SAT solver to do the same things. Of course we get the same results, but the time used is a little longer.