

CSE 140L

Lab Assignment 4

Logic Minimization and Static Hazards

Due November 23, 2014, 4:30 pm

Introduction:

This lab consists of two parts. The first part asks you to write your own logic minimizer in Java using the algorithms discussed in class. The second part of the lab asks you to explore the realm of static hazards in digital circuits, asking you to observe and fix static hazards in LogicWorks implementations of boolean functions given a set of transition-don't-cares.

Part 1: Logic Minimization

As you know, logic minimization typically involves 3 steps: (1) Prime Implicant Generation, (2) Prime Implicant Pruning, and (3) Prime Implicant Minimization. In this part of the lab, you will be implementing a variety of algorithms in order to accomplish the aforementioned 3 steps.

1.1 Prime Implicant Generation

In order to implement this lab, you will need to remember some additional vocabulary terms. An **implicant** is a product of one or more variables (e.g. a , $a'bc$), and a **Two-Level-Sum-of-Products** expression is a boolean expression that consists of only a sum of Implicants (e.g. $ab + a'c$). Implicants can be decomposed into smaller implicants (ones with more terms), and the smallest such decomposition for a given number of variables is called a **minterm**. A **prime implicant** is the largest possible implicant (i.e. the terms with the fewest literals possible) given the minterms that are provided within the function. For example, if you have a 4-variable function with the minterms $abc'd$ and $abcd$, their prime implicant would be abd , which covers both of those minterms. Furthermore, while minterms and implicants for a boolean expression indicate what combination of input values should result in an output value of 1, **don't care's** can be used to indicate that we do not care what the output value is when given a particular combination of inputs; for example, if we don't care what the output is when the variables a , b , and c are all high, then we can output either 1 or 0.

Prime Implicant generation can be accomplished by examining all minterms and don't care pairs, combining them if they can be combined, and repeating this process for the newly generated implicants. The number of comparisons, however, can be reduced if one utilizes the **Tabulation Method**. This algorithm relies on the fact that at any stage of the algorithm, only implicants that are Hamming distance 1 apart can be combined. Consequently, if one partitions the implicants into groups based on the number of 1's in the implicant representation, one can greatly reduce the number of comparisons.

The tabulation method starts out by partitioning your minterms into $G \times 0$ -subcubes, where x

corresponds to the number of 1s that the minterm value contains¹. The values from the G_x 0-subcubes are then merged with values from the G_{x+1} 0-subcubes to form G_x 1-subcubes, such that 2 values of Hamming distance 1 can combine to form an implicant, and the values merged are similarly marked as “covered.” This is done for all possible G_x and G_{x+1} 0-subcubes, with the “covered” implicants continuing to participate in the rest of the comparisons. The process continues with the 1-subcubes onwards until no more merging is possible. The remaining uncovered implicants that arise from the algorithm constitute the set of prime implicants.

At this point you may be wondering how *don't cares* fit into the process of forming subcubes. Simply put, *don't cares* can be treated as if they were just additional minterms. Since we do not care what value a *don't care* takes on, allowing it to get merged with minterms to help form bigger subcubes can only benefit us at this point by reducing the number of logic gates needed when implementing a boolean expression. Thus, when performing the tabulation method, if there is a minterm that has a Hamming distance of 1 from a *don't care* we are able to merge them together to form a larger subcube just like we did for minterms that are a Hamming distance of 1 apart.

The details of the algorithm are described more explicitly in your book, and an example is shown for a boolean function containing minterms {0, 1, 2, 3, 4, 5, 6, 8, 10, 11, 12, 15, 19, 22, 23, 24, 26, 27, 30, 31} and *don't cares* {9, 25} in the table on the following page.

In the example shown in Table 1, after 4 steps, the algorithm cannot do any more merging, so the eleven uncovered implicants constitute your prime implicant set (with the individual bits corresponding to abcde), which can be written as: {-0110, 00-0-, 00--0, 0--00, --011, -1-11, 1--11, 1-11-, 11-1-, 0-0--, -10--}. In terms of product terms, these are {b'cde', a'b'd', a'b'e', a'd'e', c'de, bde, ade, acd, abd, a'c', bc'}.

¹ Note: “*i*-subcubes” are termed as “column $i+1$ ” in the narrative of your textbook on pages 165-168.

	0-subcubes	1-subcubes-	2-subcubes	3-subcubes
G0	00000	~~0000-~~ ~~000-0~~ ~~00-00~~ ~~0-000~~	~~000-~~ ~~00-0-~~ ~~0-00-~~ ~~00-0~~ ~~0-0-0~~ ~~0-00~~	0-0--
G1	00001 00010 00100 01000	~~000-1~~ ~~00-01~~ ~~0-001~~ ~~0001-~~ ~~00-10~~ ~~0-010~~ ~~0010-~~ ~~001-0~~ ~~0-100~~ ~~0100-~~ ~~010-0~~ ~~01-00~~ ~~-1000~~	~~0-0-1~~ ~~0-01-~~ ~~-100-~~ ~~-10-0~~ ~~010-~~	-10--
G2	00011 00101 00110 01001 01010 01100 11000	~~0-011~~ ~~-0011~~ ~~-0110~~ ~~010-1~~ ~~-1001~~ ~~0101-~~ ~~-1010~~ ~~1100-~~ ~~110-0~~	~~-011~~ ~~-10-1~~ ~~-101-~~ ~~110-~~	None
G3	01011 10011 10110 11001 11010	~~01-11~~ ~~-1011~~ ~~10-11~~ ~~1-011~~ ~~1011-~~ ~~1-110~~ ~~110-1~~ ~~11-10~~ ~~1101-~~	~~-1-11~~ ~~1--11~~ ~~1-11-~~ ~~11-1-~~	None
G4	01111 10111 11110 11011	~~-1111~~ ~~1-111~~ ~~1111-~~ ~~11-11~~	None	None
G5	11111	None	None	None

Table 1: Tabulation Method Example, where covered subcubes are crossed out

1.2 Implicant Representation

Now that we know who our implicants are, we must address the issue of how implicants will be represented in your computer program. If you recall the implicant representation used in Lab 3, feel free to skip this section; however, if you would like to review this issue, keep

reading. Let's revisit this issue of implicant representation in a computer program with an example using the implicant ac' . There are two visible variables within this implicant, a and c , where a is in its uncomplemented form and c is in its complemented form. What about b ? A cursory glance at this implicant would yield the conclusion that b is not present within this implicant. However, as you recall from CSE 140, ac' is equivalent to $abc' + ab'c'$, which means that the variable b actually **is** present within the implicant, as it is present in **both** its complemented and uncomplemented form.

Given the aforementioned 3 states of a variable within an implicant, we introduce the *positional-cube notation*, which we will see is a mechanism for representing the state of variables within a given implicant in our program. These three cases require two bits to represent the state of each variable within an implicant. We ask that you use the following representation for your implicants when implementing the Tabulation Method:

uncomplemented = 01
complemented = 10
both = 11

While the encoding might seem arbitrary, a moment of reflection will reveal to you that the two bits of the encoding follow a certain pattern. We can see that the most significant bit (MSB) of this two-bit representation will be a '1' if and only if the complemented form of the variable is within the implicant and is a '0' if it is not within the implicant. Likewise, a '1' in the least significant bit (LSB) denotes that the uncomplemented form of the variable is within the implicant with conversely a '0' at the LSB denoting its absence.

One question you might now ask is what '00' should correspond to. Given the observation in the previous paragraph, '00' would correspond to a variable that is present in neither its complemented nor uncomplemented form within the given implicant. But what does this mean? Let's go back to our example with $ac' = abc' + ab'c'$. Let's say we want to modify this implicant so that neither b nor b' are present. This would require removing both abc' and $ab'c'$ from the implicant, both of which are needed to make the implicant ac' . Thus, the resulting implicant with both b and b' removed is one in which **no variables** are present within the implicant, meaning that the implicant does not exist. So, we can thus consider '00' for any variable to be a pattern that renders the implicant **invalid**. Additionally, without loss of generality, if one variable has an encoding of '00' in an implicant, then we can replace the implicant such that all variables have an encoding of '00'.

Let's apply this notation to some examples. Let's take the 5-variable implicant $ab'e$. The variable a is not complemented, so it would be represented as '01', while the variable b is complemented, giving it a representation of '10'. The variables c and d are not shown, which means they should be encoded as '11', and e is not complemented, meaning it should be again represented as '01'. So, the implicant $ab'e$ would be represented as 01 10 11 11 01.

This encoding also provides a number of benefits. The first benefit is that the Hamming distance metric can be effected through a simple **bitwise-xor** and checking whether the location for a particular variable possesses a single pair of 1's. How do we check this? Well, for a variable to differ between two implicants, it must be present in its complemented form in one of the implicants and in its uncomplemented form in the other implicant. Because of our encoding, a variable in its complemented form is encoded as '10' while a variable in its uncomplemented form is encoded as '01'. An xor of these two values will produce '11'. What about the remainder of the variables? Well, since they should be identical between the two

implicants, their xor should produce '00'. Thus, we can compute whether two implicants differ by a single variable by taking a **bitwise-xor** of the two implicants and verifying that the two bits corresponding to a variable are both 1's, as shown in the example below with *templmp1*, 01 01 01 (abc), and *templmp2*, 10 01 01 (a'bc):

$$01\ 01\ 01 \wedge 10\ 01\ 01 = 11\ 00\ 00$$

As you can see, the two bits corresponding to the variable *a* are both 1, meaning that the two implicants only differ by the term *a*. As you can also see, the two bits corresponding to the remaining variables are both 0, meaning that they are identical between the two implicants.

The second benefit is that, if the implicants are hamming distance 1 away from each other, they can be combined through a **bitwise-or**. For example, once we know the two implicants differ by a single variable by performing a bitwise-xor as we did above with *templmp1* and *templmp2*, the two implicants can be combined through a bitwise-or. So, if we take *templmp1*, encoded as 01 01 01, and union it with *templmp2*, 10 01 01, we obtain:

$$01\ 01\ 01 \vee 10\ 01\ 01 = 11\ 01\ 01$$

This result corresponds to the implicant *bc*, which covers both *templmp1* and *templmp2*.

1.3 Prime Implicant Pruning

Upon the generation of the full set of Prime Implicants, steps must now be taken to reduce the size of the candidate set for the final cover while of course still covering the entire function. As you have learned in class, this process is quite expensive, and typically pruning of the Prime Implicant Table should be employed to improve performance. The **Quine-McCluskey** algorithm provides three general rules that enable a great deal of pruning to occur:

1. If a prime implicant *p* is the only implicant that contains minterm *m*, then it must be included in the final cover, so any additional minterms covered by *p* do not need to be considered and can thus be removed.
2. A prime implicant *p1* that covers a superset of minterms covered by *p2* is said to dominate *p2*. In such a scenario, it can be inferred that a final minimal cover exists without including *p2*, and it can thus be removed from consideration.
3. A minterm *m1* that has a superset of implicants covering it when compared to another minterm *m2* can be removed from consideration, as using any implicant that covers *m2* guarantees the coverage of *m1*.

One can iteratively apply these three steps until either all of the implicants disappear or none of the three steps can be further applied. In the latter scenario of encountering a cyclic core, alternative methods for prime implicant minimization must be employed, which will be discussed in detail in 1.5.

Now, you may be wondering how we handle *don't cares*. Fortunately, as *don't cares* can take on a value of either 0 or 1 we do not need to worry about them being covered by a prime implicant. Because ***don't cares* do not need to be covered** by a prime implicant they can safely be ignored when selecting prime implicants using the Quine-McCluskey algorithm. You will notice that as a result of *don't cares* not needing to be covered, the *don't cares* from our

example (9 and 25) do not show up in the columns of the table below.

The subsequent pages detail an example of the application of these three rules for the prime implicants generated in Part 1.1.

To start, we obtain the following table:

	0	1	2	3	4	5	6	8	10	11	12	15	19	22	23	24	26	27	30	31
$b'cde'$							x							x						
$a'b'd'$	x	x			x	x														
$a'b'e'$	x		x		x		x													
$a'd'e'$	x				x			x			x									
$c'de$				x						x			x					x		
bde										x		x						x		x
ade													x		x			x		x
acd														x	x				x	x
abd																	x	x	x	x
$a'c'$	x	x	x	x				x	x	x										
bc'								x	x	x						x	x	x		

We can immediately see that $a'b'd'$, $a'd'e'$, bde , and bc' are essential, since $a'b'd'$ is the only implicant that covers minterm 5, $a'd'e'$ is the only implicant that covers minterm 12, bde is the only implicant that covers minterm 15, and bc' is the only implicant that covers minterm 24. We can thus cross out their rows and corresponding columns:

	0	1	2	3	4	5	6	8	10	11	12	15	19	22	23	24	26	27	30	31
$b'cde'$							x							x						
~~$a'b'd'$~~	~~x~~	~~x~~			~~x~~	~~x~~														
$a'b'e'$	x		x		x		x													
~~$a'd'e'$~~	~~x~~				~~x~~			~~x~~			~~x~~									
$c'de$				x						x			x					x		
~~bde~~										~~x~~		~~x~~						~~x~~		~~x~~
ade													x		x			x		x
acd														x	x				x	x
abd																	x	x	x	x
$a'c'$	x	x	x	x				x	x	x										
~~bc'~~								~~x~~	~~x~~	~~x~~						~~x~~	~~x~~	~~x~~		

After doing this, we know that **a'b'd'**, **a'd'e'**, **bde**, and **bc'** have to be in our minimal cover, so we will keep this in mind. For readability, we can now redraw the table without these implicants and minterms, which is shown as follows:

	2	3	6	19	22	23	30
b'cde'			x		x		
a'b'e'	x		x				
c'de		x		x			
ade				x		x	
acd					x	x	x
abd							x
a'c'	x	x					

While there are no more essentials now, we do see that the row for **acd** is a superset of the row for **abd**, so we can cross out **abd**:

	2	3	6	19	22	23	30
b'cde'			x		x		
a'b'e'	x		x				
c'de		x		x			
ade				x		x	
acd					x	x	x
~~abd~~	<del></del>	<del></del>	<del></del>	<del></del>	<del></del>	<del></del>	~~x~~
a'c'	x	x					

Now, we can cross out the columns for minterms **22** and **23**, since they are supersets of the column representing minterm **30**:

	2	3	6	19	~~22~~	~~23~~	30
b'cde'			x		~~x~~	<del></del>	
a'b'e'	x		x		<del></del>	<del></del>	
c'de		x		x	<del></del>	<del></del>	
ade				x	<del></del>	~~x~~	
acd					~~x~~	~~x~~	x
~~abd~~	<del></del>	<del></del>	<del></del>	<del></del>	<del></del>	<del></del>	~~x~~
a'c'	x	x			<del></del>	<del></del>	

One can also see that **acd** is essential, so we can cross out its row as well as its minterm column, labeled as **30**:

	2	3	6	19	22	23	30
b'cde'			x		x		
a'b'e'	x		x				
c'de		x		x			
ade				x		x	
~~acd~~					x	x	x
~~abd~~							x
a'c'	x	x					

We now know that **a'b'd'**, **a'd'e'**, **bde**, **bc'**, and our newly added **acd** must be part of our final cover. We now have the following table:

	2	3	6	19
b'cde'			x	
a'b'e'	x		x	
c'de		x		x
ade				x
a'c'	x	x		

We can now, again, see that the row for **c'de** dominates the row for **ade**, so we can cross out the row for **ade**. We also see that the row for **a'b'e'** dominates the row for **b'cde'**, so we can cross out the latter row, which gives us the following:

	2	3	6	19
~~b'cde'~~			x	
a'b'e'	x		x	
c'de		x		x
~~ade~~				x
a'c'	x	x		

We can also see that minterm column **3** dominates minterm column **19**, and we can see that column **2** dominates column **6**, so we obtain:

	2	3	6	19
~~b'cde'~~			x	
a'b'e'	x		x	
c'de		x		x
~~ade~~				x
a'c'	x	x		

We now have more two essentials, **a'b'e'** and **c'de**. By using them, we have cleared the entire table, so our final minimal cover will be:

$$a'b'd' + a'd'e' + bde + bc' + adc + c'de + a'b'e'$$

As you have probably realized, the implementation of this algorithm requires the ability to generate sets corresponding to both the rows and the columns. Subset/superset relationships must then be ascertained in order to determine whether columns and/or rows can be removed. To do this, we recommend that you utilize **bit-vectors** to represent your sets.

The concept of a bit-vector is very similar (actually, identical) to the implicant representation described earlier. Bit-vectors are usually applied to sets containing finite numbers of elements and utilize simple integer or long types to represent them. Each possible element in the set is assigned an index within the integer, and its presence within the set is indicated by '1' in that element's index within the integer. Likewise, its absence is indicated by a '0' at that index.

For example, an implicant **a'c'** for a 5-variable function uses minterms {0, 1, 2, 3, 8, 9, 10, 11}. Because the number of total minterms is fixed, one can use a 32-bit integer to represent the set. If we index the set from right-to-left, the bit-vector for the implicant **a'c'** would look like: 00000000000000000000111100001111.

As you can see, this is essentially a simpler form of the implicant encoding that was described earlier.

1.4 Prime Implicant Minimization

While the Quine-McCluskey method is capable of pruning prime implicant sets to more workable sizes, it is not always guaranteed that it will deliver a minimal solution solely by itself, as it may end up reducing the prime implicant table to a cyclic core. The canonical example of a cyclic core is the function containing the following prime implicants: {a'b', b'c, ac, ab, bc', a'c'}. We will label these implicants as P1 to P6, respectively. This produces the following Quine-McCluskey table:

	0	1	2	5	6	7
P1 = a'b'	x	x				
P2 = b'c		x		x		
P3 = ac				x		x
P4 = ab					x	x
P5 = bc'			x		x	
P6 = a'c'	x		x			

As you can see, this table has no essential implicants, row dominances, or column dominances. We thus need to do something else in order to fully minimize our prime implicant list. Luckily, this can be done using Petrick's method, in which the implicant cover is formulated in a product-of-sums form and converted into a sum-of-products form. For example, in the table above, we see that minterm 1 can be covered by P1 or by P6, which gives us the following sum term:

$$P1 + P6$$

Likewise, minterm 2 can be covered by either P1 or P2, allowing us to add (P1 + P2) to the formulation, giving us:

$$(P1 + P6)(P1 + P2)$$

We can continue thus with the rest of the minterms to obtain:

$$(P1 + P6)(P1 + P2)(P5 + P6)(P2 + P3)(P4 + P5)(P3 + P4)$$

The form above is then converted into a sum-of-products formulation by repeated applications of distributivity and absorption. For example, the distribution of the first two terms gives:

$$(P1 + P1P2 + P1P6 + P2P6)(P5 + P6)(P2 + P3)(P4 + P5)(P3 + P4)$$

One can immediately simplify this new sum-of-products term by using absorption to get:

$$(P1 + P2P6)(P5 + P6)(P2 + P3)(P4 + P5)(P3 + P4)$$

This process is repeated until one large sum-of-product expression is obtained:

$$(P1P5 + P1P6 + P2P5P6 + P2P6)(P2 + P3)(P4 + P5)(P3 + P4) =$$

$$(P1P5 + P1P6 + P2P6)(P2 + P3)(P4 + P5)(P3 + P4) =$$

$$(P1P2P5 + P1P3P5 + P1P2P6 + P1P3P6 + P2P6 + P2P3P6)(P4 + P5)(P3 + P4) =$$

$$\begin{aligned}
& (P1P2P5 + P1P3P5 + P1P3P6 + P2P6)(P4 + P5)(P3 + P4) = \\
& (P1P2P4P5 + P1P2P5 + P1P3P4P5 + P1P3P5 + P1P3P4P6 + P1P3P5P6 + P2P4P6 + \\
& \quad P2P5P6)(P3 + P4) = \\
& (P1P2P5 + P1P3P5 + P1P3P4P6 + P2P4P6 + P2P5P6)(P3 + P4) = \\
& P1P2P3P5 + P1P2P4P5 + P1P3P5 + P1P3P4P5 + P1P3P4P6 + P1P3P4P6 + P2P3P4P5 + \\
& \quad P2P4P6 + P2P3P5P6 + P2P4P5P6 = \\
& P1P2P4P5 + \mathbf{P1P3P5} + P1P3P4P6 + P2P3P4P5 + \mathbf{P2P4P6} + P2P3P5P6
\end{aligned}$$

One can then scan the result for the product that contains the fewest terms. In this case, there are two possible solutions, which are bolded, and either of them can be a minimal cover. Thus, the two answers are: $a'b' + ac + bc'$ or $b'c + ab + a'c$.

As you have noticed with Parts 1.2 and 1.4 of this writeup, we have tried to impress you with the power of bit-vectors when implementing the different aspects of logic minimization. We will attempt to do the same here, as one can implement Petrick's method using bit-vectors as well. We can encode a single implicant by using a bit-vector such that every implicant i is represented as a bit-vector containing a 1 at the i th bit of the integer. A product of two implicants can thus be effected through a bitwise-or of two implicants, and a sum of two implicants can be represented as a list of the two implicants we are summing.

As an example, let's encode implicants 1 to 6 as follows:

$$\begin{aligned}
P1 &= 000001 \\
P2 &= 000010 \\
P3 &= 000100 \\
P4 &= 001000 \\
P5 &= 010000 \\
P6 &= 100000
\end{aligned}$$

We can now represent the product-of-sum formulation by denoting each sum term as being a list of the prime implicants. So, for the term $P1 + P6$, we will get the list $\{000001, 100000\}$. For the entire product-of-sum expression, we will thus have the following list of lists:

$$\begin{aligned}
& (P1 + P6)(P1 + P2)(P5 + P6)(P2 + P3)(P4 + P5)(P3 + P4) \Rightarrow \\
& \{\{000001, 100000\}, \{000001, 000010\}, \{010000, 100000\}, \{000010, 000100\}, \\
& \quad \{001000, 010000\}, \{000100, 001000\}\}
\end{aligned}$$

In order to convert the product-of-sums formulation into a sum-of-products formulation, one can start with the first list and then perform a Cartesian product with the second list. The Cartesian product can utilize the bitwise-or operator to obtain the following:

$$\begin{aligned}
& (P1 + P6)(P1 + P2) \Rightarrow \\
& \{000001, 100000\}, \{000001, 000010\} \Rightarrow \\
& \{000001, 000011, 100001, 100010\} \Rightarrow \\
& (P1 + P1P2 + P1P6 + P2P6)
\end{aligned}$$

As you can see, for example, the term 000011 represents a product term containing $P1$ and $P2$. One can then easily apply absorption on this term by performing the same subset algorithm performed in 1.3. For example, exhaustively checking for subset/superset relationships will show that 000001 is a subset of 000011 and 100001, meaning that the latter

two are redundant and can be removed, as shown below:

$$\{000001, 000011, 100001, 100010\} \Rightarrow \{000001, 100010\}$$

Being an astute computer science student, you may have noticed this algorithm runs in exponential time due to the fact that it performs Cartesian products at every step. While the application of absorption helps speed things up, implicant minimization is reducible to the set-cover problem, which is NP-complete, meaning a polynomial time algorithm is unlikely to exist. However, if you manage to come up with one, let your instructors and your TA know, as there is a \$1 million prize for the person who manages to either come up with a polynomial time algorithm for an NP-complete problem or prove that no such algorithm exists.² If you have some free time in between working on this lab and studying for your exams, you can possibly take a stab at that.

1.5 Implementation Guidelines:

We ask that you implement this lab in Java. We provide you with three classes, *BooleanExpression.java*, *Implicant.java*, and *BitVector.java*.

BooleanExpression.java

In *BooleanExpression.java*, you will be tasked with modifying the *doTabulation* method and implementing three methods, *doQuineMcCluskey* and *doPetricksMethod*. The method names correspond to the algorithm that you should be implementing. We have provided a constructor within the class that takes as input a list of minterms and a list of *don't cares*, and converts them to the appropriate implicant objects, storing them in two lists, *implicantList* and *dontcareList*. The method *doTabulationMethod* has already been written for you, however it does not handle *don't cares*; you will need to modify the method to account for the *don't cares* in the *dontcareList*. The *doQuineMcCluskey* method should use the prime implicants in *implicantList* and remove those that are not essential nor comprise a cyclic core. The method *doPetricksMethod* should finally remove all redundant prime implicants from *implicantList*. In order to test your algorithm for correctness, you can use the *evaluate* method that you wrote in the previous lab as part of a *verify* method that will ensure that two boolean expressions have matching outputs when given the same set of inputs (and that ignores the output for inputs that are *don't cares*).

Implicant.java

The *Implicant.java* class is used to represent implicants. The major piece of understanding that you should note is the implementation of the encoding. In order to facilitate the computation of some aspects of your algorithm, the representation of implicants is split into an MSB and an LSB that are stored in integers. For example, the implicant *abc* is **01 01 01**. The bold and italicized portion of the encoding are stored in separate longs, which are called *myMSB* and *myLSB* within the *Implicant.java* class. In the aforementioned example, the corresponding 3 bits of the *myMSB* value will be 000, and the corresponding 3 bits of the *myLSB* integer would be 111. One question you may have is what the remaining 61 bits of the long should be, but since we know that an implicant implicitly contains both the complemented and uncomplemented forms of the remaining variables, it is obvious that the remaining bits should be set to 1. Thus, the implicant *abc* will have a *myMSB* value of

² http://www.claymath.org/millennium/P_vs_NP/

0xFFFFFFFFFFFFFFF8 and a *myLSB* value of 0xFFFFFFFFFFFFFFF, with the relevant literals being placed in the least significant bits of the longs.

This representation simplifies two operations. One operation that is simplified is how one establishes whether an implicant is valid. As you recall, an implicant is invalid if at least one variable is represented with '00'. If this occurs in our new representation, then the index of *myMSB* and *myLSB* will be 0 at the same place. If we therefore perform a bitwise-or of the two values, we would get a result that has a 0 in some spot, whereas a valid implicant would have a 1 at every spot, giving a result of -1 (in two's complement). *Thus, to establish the validity of an implicant, we simply need to perform a bitwise-or of the two longs and compare the result with -1.*

The second operation that is simplified is the determination of whether or not two implicants differ by a single variable. Let's consider two implicants, *imp1* and *imp2*. As you recall, each implicant will have a *myMSB* and *myLSB* value. One can perform a bitwise-xor of the two *myMSB* and two *myLSB* values to obtain two new values. If the original implicants differ by one variable, these new values will both be equivalent and will both possess a single 1. Thus, the following algorithm will successfully check if *imp1* and *imp2* differ by a single variable as you probably remember from your Lab 3 implementation:

```
differBySingleVariable(imp1, imp2)
    newMSB = imp1.myMSB ^ imp2.myMSB
    newLSB = imp1.myLSB ^ imp2.myLSB
    bitCountNewMSB = bitCount(newMSB)
    bitCountNewLSB = bitCount(newLSB)
    return (bitCountNewMSB == 1) && (bitCountNewLSB == 1) && (newMSB == newLSB)
```

In Java, one can use *Long.bitCount* in order to count the number of 1's in a long.

There are also two lists called *minterms* and *dontcares*, that will prove quite useful for this lab. We recommend that you keep these lists updated every time two implicants are combined, as this will make the *doQuineMcCluskey* method much easier.

BitVector.java

Finally, the *BitVector.java* class is an implementation of bit-vectors that we provide to you. You should not have to modify it, and its use is optional, as you are absolutely allowed to implement your own bit-vector data structure. However, implementing your sets without using bit-vectors will have severe performance implications that can negatively impact your grade, so we strongly urge you to use them.

Part 2: Static Hazards

This part of the lab concerns the design of static hazard free circuits. You will be returning to your familiar environment of LogicWorks. As is discussed in Section 8.4 of your textbook, static hazards are caused by signal transitions on two complementary signals (such as x and x'). If these two signals propagate through paths of distinct delays, they may exhibit identical values for short periods of time, possibly resulting in a signal glitch at the logic where these two signals reconverge.

Fundamentally, static hazards can be detected using K-Maps. A **static 1-hazard** occurs if there exist two adjacent **1-minterms** that are not covered by a common **product** term in a **sum-of-products** implementation. Similarly, a **static 0-hazard** occurs if there exist two adjacent **0-maxterms** that are not covered by a common **sum** term in a **product-of-sums** implementation. As you probably observed, the elimination of a static hazard is not a free lunch; it is achieved at the cost of redundant gates. Minimizing such a cost thus becomes an important goal in designing hazard-free circuits. For certain circuits whose behavior is easy to profile, it is possible to find the set of input transitions that will never occur during the regular operation of the circuit. This set of input transitions can be considered as *Transition Don't Cares*, and be utilized to minimize the redundant hardware needed in a hazard-free design. This is achieved by removing the redundant gates that only cover hazard-generating transitions in the *Transition Don't Care* set.

In this part of your lab, you are asked to design the **minimal** hazard-free implementation for three circuits. Circuits 1 and 3 must be implemented in the **sum-of-products** form while Circuit 2 must be implemented in the **product-of-sums** form. You will notice of course that Circuits 1 and 2 share the same Karnaugh Map. You have been given the following set of questions which should be of guidance to you in coming up with your minimal logic.

- For Circuit 1: What extra logic is needed to avoid any static hazard, if
 - a. the transition $1000(abcd_{current})$ to $1010(abcd_{next})$ is a *Transition Don't Care*.
 - b. the transition $1010(abcd_{current})$ to $1000(abcd_{next})$ is a *Transition Don't Care*.
- For Circuit 2: What extra logic is needed to avoid any static hazard, if
 - a. the transition $0001(abcd_{current})$ to $0011(abcd_{next})$ is a *Transition Don't Care*.
 - b. the transition $0011(abcd_{current})$ to $0001(abcd_{next})$ is a *Transition Don't Care*.
- For Circuit 3, a set of *Transition Don't Cares* has been specified in the table below its K-Map. While you are choosing the implicants (and there are multiple minimal covers as you can see, if static hazard handling was not an issue) to find the minimal logic, make use of the *Don't care transitions* to optimize the logic for the static hazard free implementation that you are asked to complete. We expect you to provide a hazard-free implementation with the least number of gates.

Karnaugh Map for Circuits 1 and 2:

ab \ cd	00	01	11	10
00	1	0	0	0
01	x	x	0	x
11	x	0	x	x
10	1	0	1	1

Karnaugh Map for Circuit 3:

ab \ cd	00	01	11	10
00	1	0	1	1
01	0	0	0	0
11	1	0	1	1
10	1	0	1	0

Transition Don't-Cares for Circuit 3:

$abcd_{\text{current}} \rightarrow abcd_{\text{next}}$
0000 $\rightarrow$ 0010
0000 $\rightarrow$ 1000
0011 $\rightarrow$ 0010
1100 $\rightarrow$ 1110
1111 $\rightarrow$ 1011
1111 $\rightarrow$ 1110

Deliverables:

- **Files:**

- Part 1: Please submit all code required in order to run your program (excluding any Main.java wrapper classes that you used for testing). Be sure to test your code thoroughly. We will provide you with some test cases with which you can compare your results.
- Part 2: Please submit all relevant .cct files containing your 5 hazard-free implementations, taking into account the specified transition-don't-cares.

- **Report:**

- For Part 1, your report should discuss how you implemented each of the required algorithms, including *Tabulation*, *Quine-McCluskey*, and *Petrick's Method*. Furthermore, for the following function defined by a set of minterms and *don't cares*, report the number of prime implicants in the minimal cover:
 - Minterms: {1, 2, 5, 6, 9, 15, 16, 18, 21, 22, 23, 24, 25, 26, 27, 28, 31, 32, 36, 39, 41, 43, 46, 49, 53, 54, 55, 56, 58, 59, 64, 65, 66, 67, 70, 71, 73, 74, 76, 77, 78, 79, 80, 82, 83, 84, 86, 88, 89, 91, 92, 95, 97, 98, 99, 100, 102, 104, 105, 106, 108, 110, 113, 115, 117, 118, 127}
 - *Don't Cares*: {3, 4, 10, 12, 37, 47, 72, 87, 94, 101, 103, 111, 121, 122, 123, 124}
- For Part 2, your report needs to clearly state how a transition being a *don't care* or not effects the logic needed to avoid a static hazard for Circuits 1 and 2. For Circuit 3, describe which of the hazardous minimal implementations formed the basis of your final hazard-free minimal implementation, and explain how the transition *don't cares* effected your choice. Please present the corresponding boolean function representing the implicants of the hazard-free circuit. Timing diagrams are an easy tool to visually observe if there is any static-hazard in your logic when the inputs transition from one value to another. For Circuit 3, if the *Transition Don't Care* table shown before is used as the input transition table (i.e. use the input change from the value in the left column to the corresponding value in the right), we ask you to capture the timing diagram to show how the output changes when the input transitions. Some of these transitions will exhibit a glitch. Please identify which input transitions in the set that we provided you result in such static hazards, as well as which ones do not, and explain the observed behavior.