

CSE 140L

Lab Assignment 4

Logic Minimization and Static Hazards

Due November 19, 2017, 11:59 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 Verilog implementations of boolean functions given a set of transition-don't-cares.

Part 1: Logic Minimization

As you know, logic minimization typically involves 2 steps: (1) Prime Implicant Generation, (2) Prime Implicant Table Pruning. In case the second step does not provide a complete solution, a third step is needed to resolve the cyclic core. 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 refresh on a number of terms you have studied previously. An **implicant** of a function is a product of one or more variables (e.g. a , $a'bd$, etc.) which, when true, implies the function returns true as well. 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* literals). The smallest such decomposition for a given number of variables, in which all variables of the function are represented, is called a **minterm**. A **prime implicant** is an implicant of the function from whose Boolean description no literal can be dropped without violating the function. Another way of describing a prime implicant would be as an implicant whose **complete** set of minterms cannot be described by another single implicant.

While minterms and implicants for a boolean expression indicate which input value combinations should result in an output value of 1, **don't care's** can be used to indicate that the output for that combination of inputs can be either 0 or 1. For example, if we don't care what the output is when the variables a , b , and c are all high, a correct implementation can yield any value as the output given these three input values.

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 **Tabulation Method** helps reduce the number of comparisons. 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_x 0-subcubes, where x corresponds to the number of 1s in the numeric minterm representation¹. The minterms from the G_x 0-subcubes are then merged with minterms from the G_{x+1} 0-subcubes to form G_x 1-subcubes, by combining a pair of minterms that are Hamming distance 1 apart to form an implicant. Once merged, the minterms are 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 implicants of the 1-subcubes onwards until we arrive at a set of k -subcubes where **no merger** is possible, thus being unable to spawn any $(k+1)$ -subcube. 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* produces, allowing it to get merged with minterms to help form bigger subcubes can be beneficial in reducing the number of logic gates needed when implementing a boolean expression. Thus, when performing the Tabulation Method, if there is a minterm (or another don't care) that has a Hamming distance of 1 from a *don't care*, they can be merged 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 in section 6.1, 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, the algorithm fails to perform any more merging after 4 steps. The remaining eleven uncovered implicants, expressed as {-0110, 00-0-, 00--0, 0--00, -011, -1-11, 1--11, 1-11-, 11-1-, 0-0--, -10--}, therefore constitute the prime implicant set (the individual bits denoting the literals a , b , c , d , and e in that order). In product terms, these prime implicants 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; perhaps i -subcubes is a more apt name as an i -subcube is comprised of 2^i minterms (and/or don't cares).

	0-subcubes	1-subcubes-	2-subcubes	3-subcubes
G ₀	00000	0000- 000-0 00-00 0-000	000- 00-0- 0-00- 00--0 0-0-0 0--00	0-0--
G ₁	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--
G ₂	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
G ₃	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
G ₄	01111 10111 11110 11011	-1111 1-111 1111- 11-11	None	None
G ₅	11111	None	None	None

Table 1: Tabulation Method Example, with covered subcubes crossed out

1.2 Prime Implicant Table 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 engender a great deal of pruning:

1. If a prime implicant p is the only implicant that contains minterm m , then it is deemed to be essential and must be included in the final cover. **None** of the minterms covered by p need to be considered any further and can thus be removed from the chart.
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 without $p2$ exists, enabling its removal from consideration.
3. When considering two minterms $m1$ and $m2$, $m1$ can be removed from consideration if the prime implicants that cover $m1$ are a superset of the prime implicants that cover $m2$.

One can iteratively apply these three steps until either all of the minterms have been covered 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 Part 1.3.

Now, you may wonder what role *don't cares* play in the process of prime implicant pruning. Fortunately, as *don't cares* can take on any value (either 0 or 1) we do not need to worry about them being covered by a prime implicant (which would fix them at 1). 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; conveniently, ignoring them is tantamount to simply excluding them from the prime implicant chart. 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~~							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		
a'b'e'	x		x				
c'de		x		x			
ade				x		x	
acd					x	x	x
~~abd~~							x
a'c'	x	x					

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 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 minterm column **2** dominates minterm 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 two more essentials, **$a'b'e'$** and **$c'de$** . By using them, we have cleared the entire table, so our final minimal cover will be:

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

1.3 Cyclic Core Resolution

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 the **Branch and Bound** algorithm. Branch and Bound involves picking a prime implicant and considering two solutions: one with the prime implicant included and one with it excluded². The result solution with the least number of prime implicants is chosen. (If both solutions result in the same number of prime implicants in the final cover, then we can choose either answer.)

You may wonder how to select the implicant on which to branch. A fair amount of research literature exists to identify the best implicant to select. As these heuristics are somewhat complicated to implement, we recommend a simpler heuristic for your implementation, namely, of selecting the implicant with the most minterms. In case of a tie, i.e. of multiple implicants with the most minterms, please select one arbitrarily among the tied implicants.

This algorithm we have just described will require you to recursively invoke your algorithm for Part 1.2 twice: once with the prime implicant in the final cover and once without. For the recursive case that includes the prime implicant in the final cover, the minterms should also be removed from consideration because the inclusion of the implicant ensures their coverage (just as we did when we included an essential implicant in the final cover as described in the previous subsection).

² Actually, the described method implements a branching algorithm only. In the interest of simplicity, we are forgoing the chance to implement a bounding function.

With P1 in final cover

	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			

Without P1 in final cover

	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			

At this point we simply run the algorithm of Part 1.2, prime implicant table pruning, on the two results separately. You will quickly see that a prime implicant pruning invocation on the two reduced prime implicant charts (each with 5 prime implicants now) will quickly solve both cases. However, it is possible depending on your Boolean function to get unlucky and hit another dead end (meaning that you reach another cyclic core) after invoking prime implicant table pruning. In this case, you must recurse yet again to break the cyclic core. Eventually, all of the minterms will be accounted for and you will have your final set of prime implicants.

At each return of the recursion, the two solutions are compared and the one with more implicants is discarded with the minimal solution returned.

1.4 Representational Framework

In this part, we address the issue of how implicants will be represented in your computer program. Let's explore 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 representation, 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:

complemented = 01
uncomplemented = 10
both = 00

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 '0' if and only if the complemented form of the variable is within the implicant, and is a '1' if it is not within the implicant. Likewise, a '0' in the least

significant bit (LSB) denotes that the uncomplemented form of the variable is within the implicant with conversely a '1' at the LSB denoting its absence.

One question that may be puzzling you is what '11' should correspond to. Given the observation in the previous paragraph, '11' would correspond to a variable that is present in neither its complemented nor uncomplemented form within the given implicant which is not possible. We can thus consider '11' for any variable to be a pattern that renders the implicant **invalid** (i.e. null). Additionally, without loss of generality, if one variable has an encoding of '11' in an implicant, there is no harm in replacing it with one in which all variables have an encoding of '11' since either form would denote a null implicant.

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 '10', while the variable b is complemented, giving it a representation of '01'. The variables c and d are not shown, which means they should be encoded as '00', and e is not complemented, meaning it should be again represented as '10'. So, the implicant $ab'e$ would be represented as 10 01 00 00 10.

As you have probably realized, the implementation of the **Quine-McCluskey** 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 appropriately perform these subset/superset comparisons, we recommend an alternative form of representation for the prime implicants. Since the comparison involves a decision on whether the minterms of one prime implicant are a superset of the minterms of another implicant, it will be more convenient to use yet again a bit vector representation but, this time, denoting the minterms of the implicants rather than the literals. As you know, a bit vector represents each possible element in the set by assigning an index to it and indicating the presence of that element by the appropriate value in the integer (or long) type used to support the representation. In the case of minterms, each possible minterm of the Boolean space is assigned its natural index (i.e. minterm 0 at position 0 and so on) with a value of 1 at that index position indicating the minterm being covered by the implicant and a 0 denoting that minterm isn't being covered.

For example, an implicant $a'c'$ for a 5-variable function covers 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: 000000000000000000000000111100001111.

An analogous representation would need to be effected for the comparison of the minterm **columns**. In this case, the indexed positions in the bit vector will represent the set of prime implicants with a 1 in the position indicating that the prime implicant covers the minterm corresponding to the bit vector.

As you can see, this is a simpler form of the implicant encoding described earlier, one that represents the implicant from a different vantage point. This representation focuses on capturing the information associated with the set of minterms while the previous representation focuses on capturing the information associated with the literals in the implicant representation. You will find the previous encoding of the implicants from the vantage point of its literals appropriate for the Tabulation Method while the newly introduced minterm-based encoding will prove useful for prime implicant table pruning.

1.5 Computational Framework

Such bit vector encodings provide a number of computational benefits. Let us start by looking at the benefits from the vantage point of the literal representation used in the Tabulation Method. 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 '01' while a variable in its uncomplemented form is encoded as '10'. An xor of these two values will produce '11'. What about the remaining variables? Well, since they should be identical between the two implicants for a possible merging of the two implicants in question, 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. If the two implicants have a Hamming distance of 1, then the result of the xor would have two 1's at a single positional cube and a 0 everywhere else. Here is an example with *templmp1*, 10 10 10 (abc), and *templmp2*, 01 10 10 (a'bc):

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

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. Do not worry about the remaining literals showing a '00' in this operation since you are not generating new implicants in this part but rather using the xor operation to ascertain the Hamming distance between 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-and** operation. For example, once we know the two implicants differ by a single variable by performing a bitwise-xor as we discussed previously with *templmp1* and *templmp2*, the two implicants can be combined through a **bitwise-and**. So, if we take *templmp1*, encoded as 10 10 10, and perform an intersection operation on it with *templmp2*, 01 10 10, we obtain:

$$10\ 10\ 10 \& 01\ 10\ 10 = 00\ 10\ 10$$

This result corresponds to the implicant *bc*, which covers both *templmp1* and *templmp2*. Consider also what happens if you use **bitwise-and** on two implicants that are not Hamming distance 1 apart. If the two implicants being combined are not Hamming distance 1 apart, the intersection operation will not give you their sum but rather their "super-cube" which is the smallest cube that includes the two implicants. This operation is not particularly helpful for this assignment, but if you wish to delve further into issues of logic synthesis, you may find use for it.

1.6 Subset Algorithm

Next, we will move onto the details of prime implicant pruning which uses the minterm-based encoding instead. As you recall from CSE 20, a set *S2* is a subset of *S1* if $S1 \cup S2 = S1$. A dual subset check exists as well, in which *S2* is a subset of *S1* if $S1 \cap S2 = S2$. Our encoding becomes useful when implementing this function, as a union can be effected through a **bitwise-or**, while an intersection can be effected using a **bitwise-and**.

Let us illustrate this by focusing on the minterm-based implicant encoding. We can write a quick algorithm to determine subset relationships between implicants. Let's look again at two implicants: *ab* (*imp1*) and *abc* (*imp2*). In a 3-variable Boolean space, *ab* is composed of minterms 6 and 7 while *abc* only has minterm 7.

$$\begin{aligned} \text{imp1} &= ab = 11000000 \\ \text{imp2} &= abc = 10000000 \end{aligned}$$

If we compute a bitwise-or of them, we obtain:

$$\text{imp1} \mid \text{imp2} = 11000000 \mid 10000000 = 11000000 = \text{imp1}$$

As you can see, the result of the union is simply the implicant *ab*, encoded as 10 10 00, which means that *abc* (*imp2*) is a subset of *ab* (*imp1*). Equivalently, one can utilize a **bitwise-and** to perform an intersection and obtain the same result, as shown below:

$$\text{imp1} \& \text{imp2} = 11000000 \& 10000000 = 10000000 = \text{imp2}$$

Because the result of the intersection is equivalent to *abc* (*imp2*), we can again conclude that *abc* (*imp2*) is a subset of *ab* (*imp1*). The algorithms are summarized below in pseudocode:

Implicant Subset Algorithm 1 (returns true if imp2 is a subset of imp1):

```
isSubset(imp1, imp2):
    return (imp1 | imp2) == imp1
```

Implicant Subset Algorithm 2 (returns true if imp2 is a subset of imp1):

```
isSubset(imp1, imp2):
    return (imp1 & imp2) == imp2
```

1.7 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 implementing the *doTabulation* method in order to set up your prime implicants as well as the *doQuineMcCluskey* private method which we will slightly augment to make recursive calls to resolve any remaining cyclic cores utilizing the Branch and Bound algorithm. 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 *createFinalCover* method should use the prime implicants in *implicantList* and remove those that are neither essential nor comprise a cyclic core. This is already provided for you, but you may choose to modify the implementation if you wish. In order to test your algorithm for correctness, you can use the *genVerilog* method to create Verilog code to test your values. You may add and modify whatever methods you like, but you should refrain from modifying the public method headers because we will be using these to test your code.

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 **10 10 10**. The bold and italicized portions 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 111, and the corresponding 3 bits of the *myLSB* integer would be 000. 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 any unnamed variables, it should be obvious that the remaining bits should be set to 0. Thus, the implicant *abc* will have a *myMSB* value of 0x0000000000000007 and a *myLSB* value of 0x0000000000000000, with the relevant literals being placed in the least significant bits of the longs.

This representation simplifies three operations. One operation that is simplified is how one establishes whether an implicant is valid. As you recall, an implicant is null if at least one variable is represented with '11'. If this occurs in our new representation, then the index of *myMSB* and *myLSB* will be 1 at the same place. If we therefore perform a **bitwise-and** of the two longs, we would get a result that has a 1 in some spot, whereas a valid implicant would have a 0 at every spot, generating a numeric value of 0. *Thus, to establish the validity of an implicant, we simply need to perform a bitwise-and of the two longs and compare the result with 0, as show below:*

```
checkValidity(imp)
    invalidBits    = imp.myMSB & imp.myLSB
    return (invalidBits == 0)
```

The second operation that is simplified is the determination of whether or not two implicants differ by a single literal. Let's consider two implicants, *imp1* and *imp2*. If we take the **bitwise-xor** of two identical encoded implicants, the result will contain only 0's because every bit will match between the two. If the implicants differ by a single literal, then the encoding for that particular literal must be '01' in one implicant and '10' in the other. The **bitwise-xor** for that variable would produce '11', and all other literals would be '00'. As you recall, each implicant will have a *myMSB* and a *myLSB* value. In this case, if we take the **bitwise-xor** of the two *myMSB* values and then of the two *myLSB* values, both results will contain exactly a single '1' bit at the same index if the two implicants differ by a single literal. We can check if the two results are identical to determine if the location of the '1' is at the same index. Thus, the following algorithm will successfully check if *imp1* and *imp2* differ by a single variable:

```
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.

A third operation that is simplified is checking which group an implicant belongs to for the Tabulation Method. You may recall that each group G_x contains variables that have x literals in their uncomplemented form. In our representation, the group number can be found by counting the number of literals encoded as '10'. Assuming the implicant is valid (there are no '11' literals), the *myMSB* value will contain a '1' if and only if that literal is present in its uncomplemented form, since all other literals are encoded as '01' or '00'. So, we can check the group number simply by counting the number of '1' bits in the *myMSB* value:

```
groupNumber (imp)
    return bitCount(imp.myMSB)
```

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

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 necessitates the addition of redundant gates. Minimizing this redundancy 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
 1. the transition $1001(abcd_{current})$ to $1011(abcd_{next})$ is a *Transition Don't Care*.
 2. the transition $1011(abcd_{current})$ to $1001(abcd_{next})$ is a *Transition Don't Care*.
- For Circuit 2: What extra logic is needed to avoid any static hazard, if
 1. the transition $0000(abcd_{current})$ to $0010(abcd_{next})$ is a *Transition Don't Care*.
 2. the transition $0010(abcd_{current})$ to $0000(abcd_{next})$ 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	0	0	1	0
01	x	x	1	0
11	x	1	x	x
10	1	1	1	0

Karnaugh Map for Circuit 3:

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

Transition Don't-Cares for Circuit 3:

abcd _{current} → abcd _{next}
0101 → 0100
0101 → 0001
0110 → 0100
1001 → 1000
1010 → 0010
1010 → 1000

2.1 Verilog Implementation

As timing plays a fundamental role in the observation & handling of the static hazards, we have built a library of gates with the timing specified for you. The delays that we have built in together with the names of the corresponding gate are given below. You will find that these timings are identical to what you used in Lab 3, so they should not surprise you.

Name	Delay (ns)	Name	Delay (ns)
2-input AND (andgate2)	24	2-input OR (orgate2)	24
3-input AND (andgate3)	28	3-input OR (orgate3)	28
4-input AND (andgate4)	32	4-input OR (orgate4)	32
5-input AND (andgate5)	36	5-input OR (orgate5)	36
Inverter	10		

Table 1: Gate names and their delays

Please implement the 3 circuits using **structural Verilog**. We have included these gates with the accompanying delays in a .v file included with the resources. You can instantiate these gates the same as you would for a module in Verilog. We are not interested in the synthesis of the designs into hardware, so there is no need to use Quartus. You only need to use ModelSim to simulate the designs and view the waveforms. Please do not change the module interfaces in the Verilog files provided.

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 .v files containing your 3 hazard-free implementations, taking into account the specified transition-don't-cares.

- **Report:**

- For Part 1, your report should discuss how you implemented the Tabulation Method, the basic Quine-McCluskey algorithm, and how you extended Quine-McCluskey using the Branch (and Bound) algorithm to handle cyclic cores:
 - 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, using the *Transition Don't Care* table shown before 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), please 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 *Transition Don't Care* set that we provided you result in such static hazards, as well as which ones do not, and explain the observed behavior.