Final Exam

The **due date** for this exam is **Mon 13 May 2013 12:59 AM EDT -0400**.

This is the Final Exam for VLSI CAD: Logic to Layout. The Exam has 15 problems. The Exam is structure to look like a Problem Set, except that it has more questions, but they are mostly smaller than the questions on the Problem Sets. The Exam is "untimed", which means you do NOT have to finish it in a short amount of time (e.g. 24 hours). You have roughly 5.5 days to do the Exam, including most of one weekend. The Exam is due **Sun 12 May 2013 11:59 PM CDT -0500.**

We do advise that you save your work and do not try to submit the final immediately upon finishing it, if you happen to finish it really quickly. There may be questions that appear in the class Forums and we will try to answer these very rapidly. These discussions may change your understanding of a specific answer to a specific problem.

☐ **In accordance with the Coursera Honor Code, I (Matthew Kramer) certify that the answers here are my own work.**

Question 1

Computational Boolean Algebra

Suppose we have 2 function $f(x,y)$ and $g(y)$. Note $f(\ )$ is a function of 2 variables, while $g(\ )$ is a function of only one variable. Is the following identity true or false?

$$(\forall x)[f(x,y) \oplus g(y)] = (\forall x)[f(x,y)] \oplus g(y)$$

In other words: can we pull the EXOR with the $g(y)$ function *outside* the quantification with respect to variable x here? Use Boolean algebra and what you know about cofactors to either show that this identity is is true, or this is false.

- False
- True

Question 2

BDDs: Compose Operator

Suppose we have a software package that has data structures representing variables and Boolean functions as BDDs, and that the following operations are available as subroutines in this software. (We will write these in a simplified sort of pseudo-code notation):

- *boolean* **iszero**(*bdd* $func$): Returns boolean 1 just if func is the always-zero function. The input is a BDD, output is boolean 1 or 0 (i.e., the output is a yes/no answer, it is not another BDD data structure).

- *bdd* **cofactor**(*bdd* $func$, *variable* x , *boolean* val): Computes cofactor of $func$ with respect to *var* x, setting $x = val$. $func$ is a BDD, x is a variable, and val is boolean 0 or 1. So, to compute the negative cofactor with respect to variable x for function F, $F_{\bar{x}}$, we would do this as: **cofactor**$(F, x, 0)$.

- *bdd* **AND**(*bdd* f, *bdd* g): This computes the basic logical (gate type) operation AND. Operation AND creates a new BDD, representing the logical AND of BDDs f and g.

- *bdd* **OR**(*bdd* f, *bdd* g): Logical OR of two BDDs.

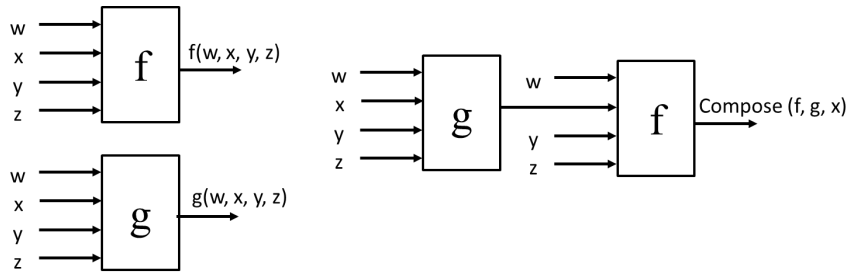
- *bdd* **EXOR**(*bdd* f, *bdd* g): Logical EXOR of two BDDs.

- *bdd* **EXNOR**(*bdd* f, *bdd* g): Logical EXNOR of two BDDs.

- *bdd* **NOT**(*bdd* f): Logical complement of its input BDD.

For this problem, we want you to build a new operator, called **compose**, which does this:

- *bdd* **compose**(*bdd* f, *bdd* g, *variable* x): Creates a new function with variable x in f set to the output of g. The picture below clarifies what we are computing. f and g are BDDs, x is a variable, and **compose** returns a BDD.



Answer this: Which of the following small pieces of pseudo code correctly computes this composition operation?

Hints:

- Things that look *difficult* are often easier when you *cofactor* them and look at combining the cofactors in clever ways.
- For this problem, it helps to think about this composition operator as being implemented by a standard 2:1 *multiplexor*, with two data inputs (data0 and data1) and one select input (Sel). The output is the composed function we seek. What are the inputs to data0, data1, and Sel? The right answer can be computed using only the BDD functions we have listed above.

☐ This correctly implements the **Compose( )** operator

bdd **Compose**(*bdd* f, *bdd* g, *var* x){
bdd $f_x \leftarrow cofactor(f, x, 1)$;
bdd $f_{\bar{x}} \leftarrow cofactor(f, x, 0)$;
return $EXOR(AND(g, f_x), AND(NOT(g), f_{\bar{x}}))$;
}

☐ *bdd* **Compose**(*bdd* f, *bdd* g, *var* x){
bdd $f_x \leftarrow cofactor(f, x, 1)$;
bdd $f_{\bar{x}} \leftarrow cofactor(f, x, 0)$;
return $OR(AND(g, f_x), AND(NOT(g), f_{\bar{x}}))$;
}

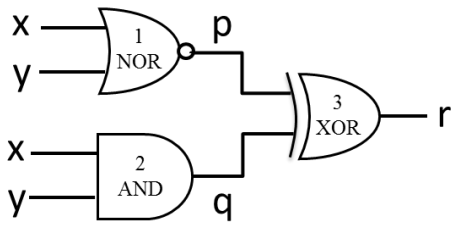
☐ This correctly implements the **Compose( )** operator

bdd **Compose**(*bdd* f, *bdd* g, *var* x){
bdd $f_x \leftarrow cofactor(f, x, 1)$;
bdd $f_{\bar{x}} \leftarrow cofactor(f, x, 0)$;
return $AND(OR(g, f_x), OR(NOT(g), f_{\bar{x}}))$;
}

Question 3

Boolean SAT: Basic Gate-Level Consistency Function

Consider the small logic network shown below. Do this: create the gate consistency function as a standard SAT CNF set of clauses.



Which of the following clauses will appear in the CNF consistency function for this logic network:

- $(\bar{r})$
- $(\bar{p} + q + \bar{r})$
- $(\bar{p} + q + r)$
- $(\bar{x} + \bar{y} + q)$
- $(x + q)$
- $(y + \bar{q})$

Question 4

Boolean SAT: Getting More Than One Solution

A standard SAT solver (like MiniSat) produces just one satisfying solution (or tell us that there is no such solution). But there are many examples where we might like to get second, or third, or Nth solution. It turns out there is a nice trick for doing this.

Suppose we have a function $F(a, b, c, d, e)$. And that we call our SAT engine, and it tells us that this is a satisfying solution:

- $a = 0, b = 1, c = 1, d = 0, e = 1$

How do we get a second solution? They answer is that **we add something new** to the CNF

description. The key idea is to add something extra --a new clause or clauses -- that will prevent us from finding this exact solution again -- $a = 0, b = 1, c = 1, d = 0, e = 1$ -- in the next call to the SAT engine. The additional clause(s) block the solver from discovering this previous solution, by rendering this solution as unsatisfiable.

Answer this: which of these are correct statements about how to modify the original CNF description to allow us to obtain (if it exists) a second satisfying solution:

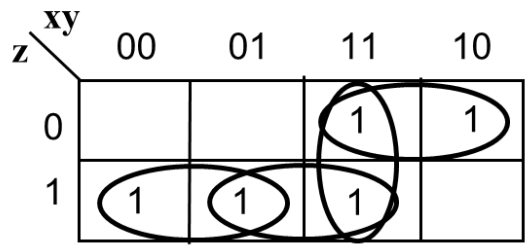
- Add 2 clause: $(a + d)(b' + c' + e')$.
- Add 5 clauses: $(a')(b)(c)(d')(e)$.
- Add 1 clause: $(a + b' + c' + d + e')$.
- Add 2 clause: $(a' + d')(b + c + e)$.

Question 5

ESPRESSO: Extracting the "Essential" Primes

ESPRESSO uses many clever URP and PCN algorithms to do important operations. We only discussed a few of these in lecture. One important step in the ESPRESSO tool is to find what are called **Essential Prime Implicants**. A prime is said to be "essential" if there is a row of the truth table with a 1(or, equivalently, a 1 in the Kmap) that can only be covered by exactly this single prime. In other words: any final minimal cover must include this prime, because it is essential for covering this 1.

Consider this simple function $F(x, y, z)$ shown below:



$$\text{Cover} = \{\text{xy} + \text{xz'} + \text{yz} + \text{x'z}\}$$

You can see that there are two essential primes, based on this definition.

Here is a simplified version of how ESPRESSO finds these essential primes:

```
let H = { original cubes in the cover of F, but omit the cube C we want to test for
 "essential" }
let Q = { } // empty set to start, will hold cubes we construct in the loop
foreach( cube p in the set H){
        let q = INTERSECT( cube p, cube C) // just AND their PCN cube forms
        if ( q is not the EMPTY cube )
        then add cube q to the set Q
}
if ( Q cover cube C ) // can do this with tautology like on HW, just do it by eye,
here
then return ( FALSE - C is not essential )
else return ( TRUE - C is essential )
```

Do the following: run this algorithm on the cube $C = x'z$ from this cover. Which of the following are true statements about this result:

- The resulting Q cover = $\{x'yz\}$ does not cover cube $C = x'z$, so $C = x'z$ is not essential.
- The resulting Q cover $= \{x'yz\}$ does not cover cube $C = x'z$, so $C = x'z$ is essential.
- The resulting Q cover = $\{x'yz,\ x'y'z\}$ covers cube $C = x'z$, so $C = x'z$ is not essential.

Question 6

Multi-Level Optimization: Kerneling, Extraction

Consider this Boolean logic network over the variables p, q, r, s, t, u:

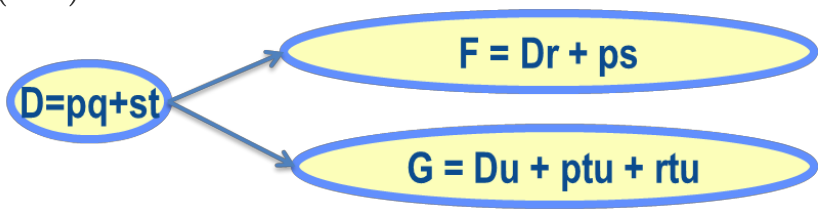
$F = pqr + str + ps$

$G = pqu + stu + ptu + rtu$

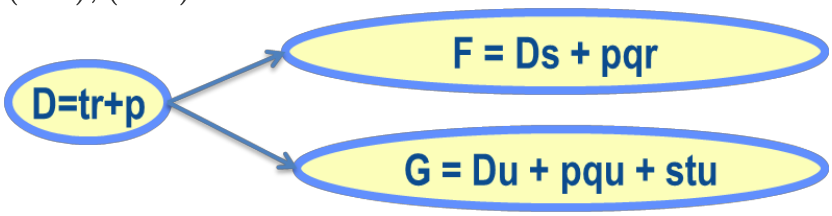
Do this: First, kernel each node in the network. Second, build the cokernel-cube matrix for this network. Third, look for good prime rectangles in this network, for use as common divisors.

Which of these are correct statements about the results of this set of optimization steps?

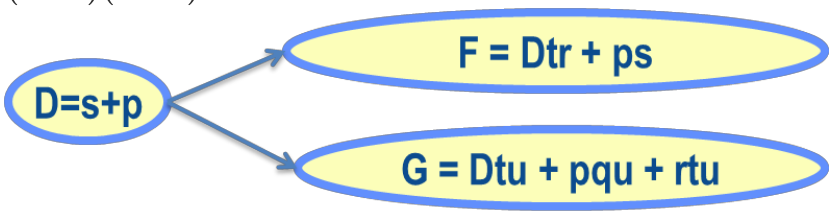
- [ ] Multiple-cube divisor $D = pq + st$ can be extracted as a prime rectangle from this matrix. The prime rectangle has 2 columns and 2 rows: (Function Co-kernel) rows are: $(F\ r)$, $(G\ u)$. This extraction saves 4 literals.



- [ ] F has $pq + st$ with co-kernel r; and G has kernel $q + t$ with co-kernel pu.
- [ ] Multiple-cube divisor $D = tr + p$ can be extracted as a prime rectangle from this matrix. The prime rectangle has 2 columns and 2 rows: (Function Co-kernel) rows are: $(F\ s), (G\ u)$. This extraction saves 4 literals.



- [ ] Multiple-cube divisor $D = s + p$ can be extracted as a prime rectangle from this matrix. The prime rectangle has 2 columns and 2 rows: (Function Co-kernel) rows are: $(F\ tr)(G\ tu)$. The extraction save 4 literals.

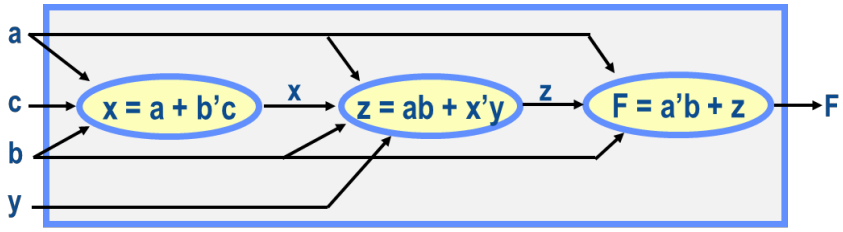


- [ ] Before any common divisor extraction, the network with just the F and G nodes in it has 20 literals.
- [ ] G has kernel $su + pu + ru$ with co-kernel t.

Question 7

Multi-Level Optimization: Don't Cares

Consider the mult-level Boolean network model shown below:



Input variables are a, b, c, y. Output is F. Internal nodes x and z. Do the following:

- Compute the SDC associated with node x.
- Compute the CDCs associated with node z.
- Determine if these CDCs assist with simplifying the Kmap for node z
- Determine the ODCs for node z.
- Determine if these ODCs assist with simplifying the Kmap for node z.

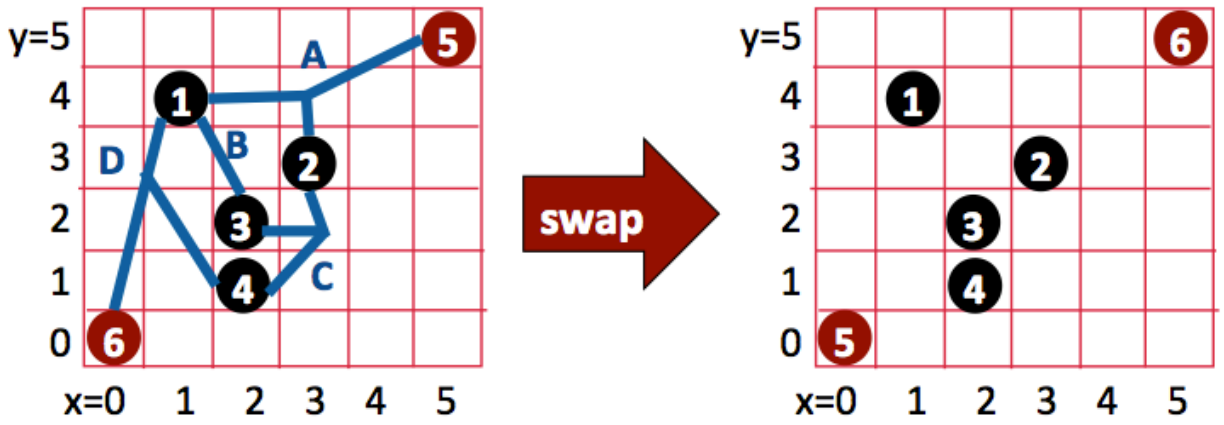
Which of the following are correct statements of the result of the Don't Care analysis for our network?

- [ ] SDC_x can be represented as $ax + b'cx$
- [ ] Looking at the 4-variable Kmap for node z, adding the CDC and ODC Don't Care patterns change 8 cells in the Kmap to Don't Care, but we cannot use these new DC patterns to simply the z function.
- [ ] CDC_z can be represented as $ax' + a'bx$
- [ ] ODC_z can be represented as $a'b$
- [ ] CDC_z can be represented as $ax' + a'bx + a'b'cx' + a'b'c'x$
- [ ] SDC_x can be represented as $ax' + a'bx + a'b'cx' + a'b'c'x$

Question 8

Anneal Placement

Consider this small placement of 6 gates in a small 6x6 grid.



Each gate is drawn as a circle with number (1–6). Assume the gate is located at the center of the grid cell, and its (X, Y) coordinates are taken from the column (X) and row (Y) coordinates in the figure. There are 4 nets, labeled A, B, C and D, connected as follows:

- Net A: gates 1, 2, 5
- Net B: gates 1, 3
- Net C: gates 2, 3, 4
- Net D: gates 1, 4, 6

A simple illustration of each net is also shown on the placement grid.

We now swap gates 5 and 6, as shown in the diagram. Assume this happens inside a simulated annealing placer, and that the current temperature is **T=10**. We use HPWL as the cost function.

Do this:

- Compute Δ**L = (new HPWL after swap) – (old HPWL before swap)**. To be clear: if the wirelength increases, this is a positive number; if it decreases, it is a negative number.
- Compute the probability that this swap of gates 5 and 6 will be accepted at this temperature. Round your probability to **3** significant digits, e.g., 0.871 or 0.094.

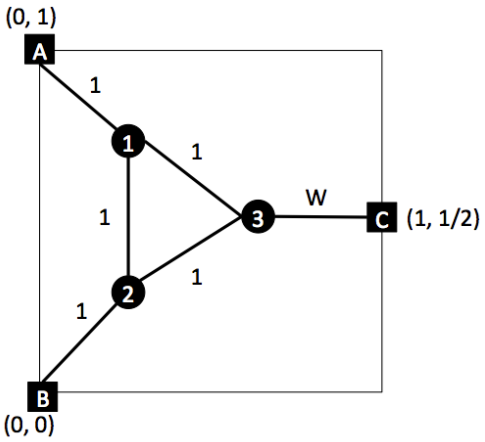
Which of these are correct statements?

- [ ] If we choose a random value R uniform in [0, 1] and we get R=0.819, we accept the swap.
- [ ] $\Delta L = 3$
- [ ] The acceptance probability for the swap is 0.819.
- [ ] The acceptance probability for the swap is 0.741.

Question 9

Quadratic Placement

Consider the simple netlist shown below. 3 gates to be placed, 3 fixed pads, 6 2-point connections with the weights shown. One of these weights, ***W***, is unknown.



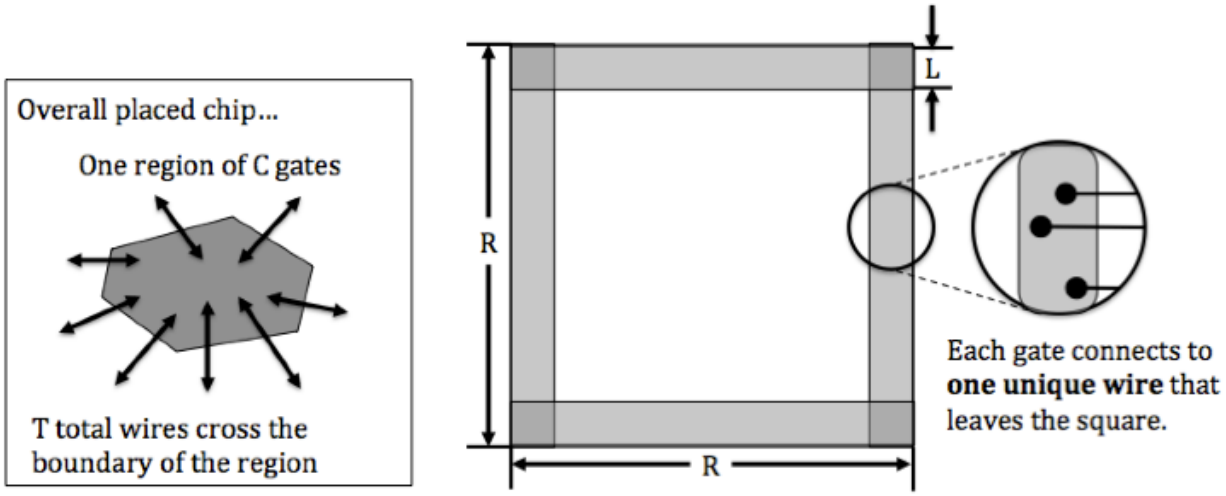
Do this: what is the value of wire weight ***W*** needed so that, when you solve for the x coordinate location of all gates using basic quadratic placement, the result will have $x_3 = 0.75$? ***W*** is a positive number. Type the **value** of ***W*** in the box. Round the value to 2 significant digits, i.e., if ***W*** = 1.235, then you type '1.24'.

Note: you do not need to solve for any y coordinates. You only need to solve for ***W*** so that $x_3 = 0.75$.

Question 10

Placement: The "Structure" of Well-Placed Logic

In the 1960s, Ed Rent of IBM discovered a remarkable relationship: if you look at a "well placed" region of logic with **C** gates in it, then the total number of wires **T** (**T** for "terminals") needed to get signals into and out of this region can be estimated as: $T = AC^p$, **p** and **A** are empirical constants, **p** is typically between 1/2 and 1. This is called ***Rent's Rule***, and **p** is called the **Rent Exponent**. Since this isn't physics, **p** and **A** are not fixed constants for all logic (like Boltzman's constant). These numbers vary with different kinds of logic.



Rent's rule says if we take a region of "well placed" logic with **C** gates in it, we can estimate the number of wires **T** that will cross that region to reach the gates inside, as $T = AC^p$.

Think of Rent's Rule as being a "surface to volume" ratio for well-placed logic. The "volume" is **C**, the number of gates in a region; the "surface" is measured in units of "wires", as in "how many wires need to enter or escape from this region?" Rent's Rule gives a mathematical relationship between this surface and its interior volume; this was entirely unexpected, and it's why Ed Rent is famous.

We can use Rent's Rule to roughly estimate a sort of "average wirelength" **L** inside a region of well-placed logic, from only the "**A**" and "**p**" values. **L** is the length of a "typical wire" in this placement. There is a very simple, very approximate derivation for this length **L**. The derivation goes like this:

- Assume gates are very small, and units are normalized; a region with area = **X** has **X** gates in it.
- Take a square with dimensions $R \times R$, it has area R^2 and so it also has $C = R^2$ gates in it. Assume $R >> L$, the average wirelength we are trying to compute.
- Assume that only the gates that are in a thin band of area within a distance **L** from the edges of this square actually connect to a wire that will leave the square. For simplicity, assume each of the gates in this thin band of area near the edges connects to **one unique wire** that leaves the square. (See the right figure above)
- With just a few assumptions, and a very little bit of algebra, derive a formula for **L**, the average wirelength. We want **L = Function(A, C, p)**, you need to tell us what this formula looks like.

(**Hints**: Rent's Rule tells you how many wires leave this $R \times R$ square. But your assumption that all gates in this "thin band" also leave this region gives you a different way to relate the number of gates in an area to the quantity **L**. This is the right place to start. You use the **R**'s,

but then you have to get rid of them and replace them with something that only depends on **C**. Make sensible geometric approximations.)

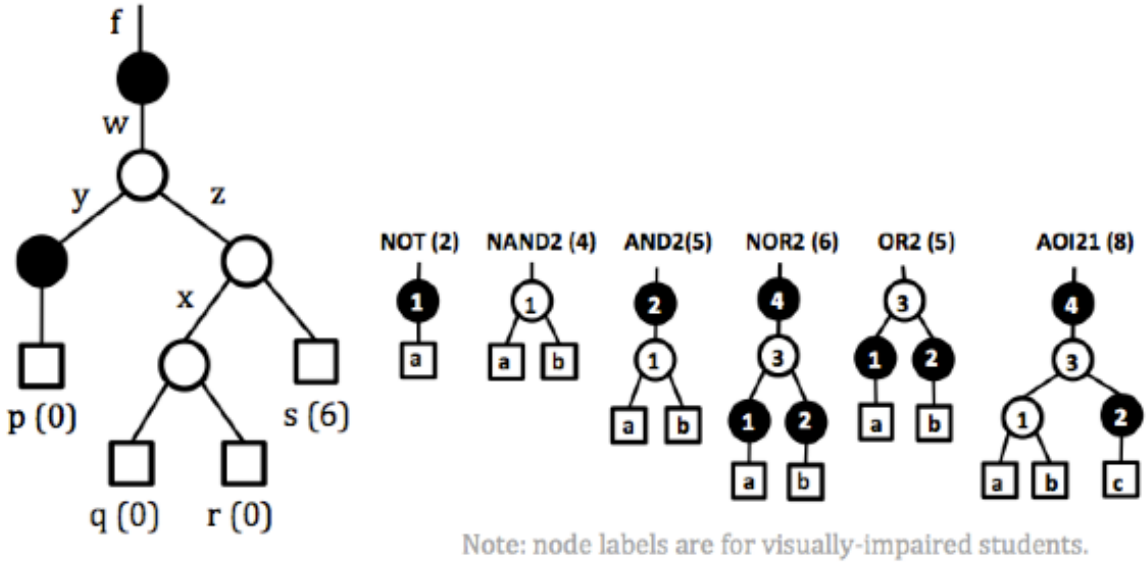
Do this: which of these are the correct statement about the formula for the average wirelength?

- [ ] The number of gates in the four thin bands can be estimated as $4RL$, since $R >> L$. This is also the number of wires that leaves the square.
- [ ] The average wirelength **L** is proportional to $C^{p-1/2}$.
- [ ] The average wirelength **L** is proportional to C^{2p}.
- [ ] The number of gates in the four thin bands can be estimated as $4R^2/L$. This is also the number of wires that leaves the square.
- [ ] By Rent's rule, there are AR^p wires entering/exiting the $R \times R$ square.
- [ ] By Rent's rule, there are AR^{2p} wires entering/exiting the $R \times R$ square.

Question 11

Technology Mapping for Delay Minimization

We can change the way cost is minimized in standard tech-mapping to minimize delay. Consider the network to be mapped (at left), and technology library (at right) shown below. We give the delay "through" each library gate (assume it's the same delay value from each gate input to the gate output). We also give the arrival time of each primary input to the network we want to map. For example, input **p** arrives at time **0**, but input **s** arrives at time **6**. The delay through the NAND2 is **4**; the delay through AOI21 is **8**.



The matching part of the tech-mapping algorithm works as usual: we look at what can structurally match at each node, and record this for each node. But we need to modify the "mincost" tree-covering algorithm to do this delay optimization.

The key idea is that the "cost" of choosing to use a particular library gate at a particular node in the network is a bit different than the minimum-cost mapping we showed in lecture. But it turns out the modification is rather straightforward. We still need a recursive cost formula for a pattern **P** matched at a node **n**, but now instead of the cost used in the lecture (which we can think of the silicon area of the gate), we use the **Arrival Time** as the **cost**. This is, surprisingly, just like an "AT" from our lecture on static timing. The arrival time is determined by the latest (maximum) arrival time of any of the gate (pattern P's) inputs, and the delay through the gate itself. So, we get this for the basic recursive formula for cost:

```
cost(n, P) = Arrival(n, P) = Delay(P) + max{minArrival(i)}, for all input i of P
```

where the notation `max{minArrival(i)}, for all input i of P` means, for example, if P has 2 inputs a and b: `max{minArrival(a), minArrival(b)}`.

Instead of writing the recursion in terms of the mincost( ) value for each leaf node of the pattern P, we write this in terms of the minimum delay, which is "minArrival()". And just as with the classical tech-mapping algorithm, we want the minimum-cost cover of the subject tree. So now, when we minimize "cost" we are minimizing "Arrival Time" for the signal at the root of the tree, i.e., we are minimizing the delay through the overall mapped network.

So, putting these ideas together: the minArrival (i.e., smallest delay) at node **n** is:

```
minArrival(n) = min{Arrival(n, P)}, for all matched patterns P at n
```

where the notation `min{Arrival(n, P)}, for all matched patterns P at n` means, for example, if there are two matches P1 and P2 at n: `min{Arrival(n, P1), Arrival(n, P2)}`. Clearly, minArrival of a PI is just its arrival time.

For example, the "cost" that we need to calculate to put a NAND2 at node **z** is

```
Arrival(z, NAND2) = 4 + max{minArrrival(x), 6}
```

The "**4**" is the delay through the library NAND2 gate itself. The "**6**" is the arrival time for signal **s**. You have to recursively compute the minArrival (smallest arrival time) for **x** based on the best mapping. The maximum operation is required because for ATs, we care about the slowest delay, i.e., the maximum delay value. Just as with the standard mapping algorithm, you want to choose the best value at the root of the tree, which in this case is the minimum total delay; then you can work backwards to find the best match at each internal node.

Do this:

- Determine, for each node in the network node (**x**, **y**, **z**, **w**, **f**), which library gates can match.
- Compute, using a table like in the lectures, the recursive calculation of the smallest delay mapping for this network. This is just like regular tech-mapping, but you use "Arrival Time" as the cost. and you calculate the minArrival( ) value for each node.
- Determine the final mapped network, and calculate the final minimum delay.

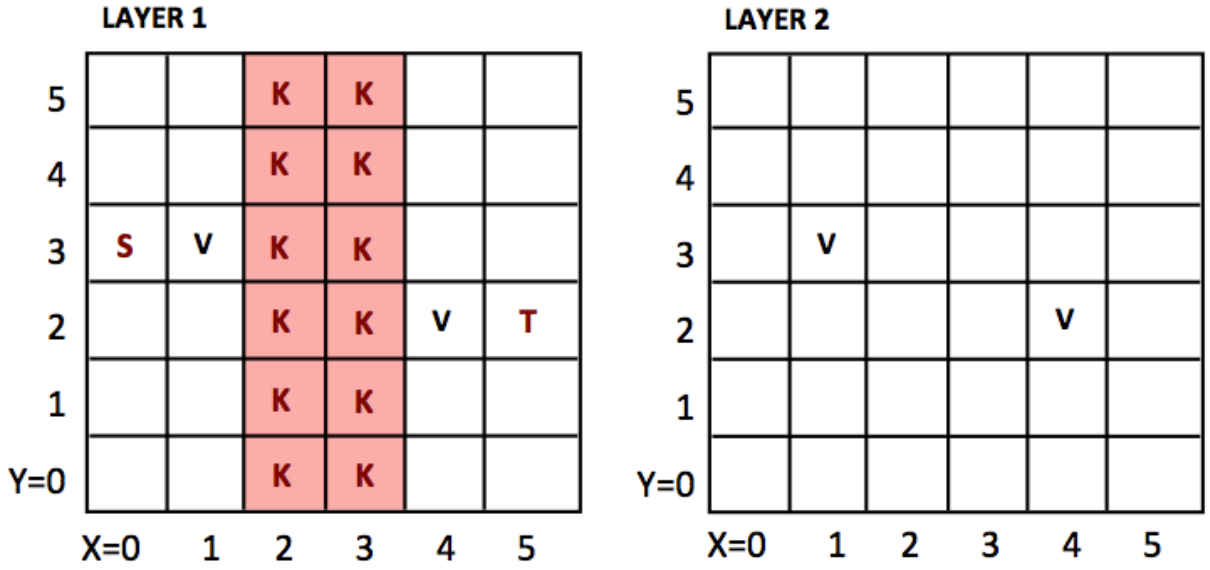
Answer the following: which of these are correct statements about the mapping:

- [ ] minArrival(y)=4
- [ ] minArrival(z)=Arrival(z, NAND2)
- [ ] The best cover of the tree has minArrival 14, and it has 2 total library gates.
- [ ] The best cover of the tree has minArrival 16, and it has 5 total library gates.
- [ ] The AND2 pattern from the library matches the subject tree at node f.
- [ ] The OR2 pattern from the library matches the subject tree at node w.

Question 12

Maze router

Consider the simple pair of 6x6 routing grids shown below. We are routing in 2 layers: Layer 1 (at left) and Layer 2 (at right). We label one **source S in Layer 1** and one **target T in Layer 1**, and several non-unit cells (in pink, with **positive integer** cost **K**). All white cells have unit cost. Vias can be inserted to move from one layer to the other; however, we **only** allow vias where there is "V" marked in the grid. Each Via has cost=**10**. Assume your router uses the cheapest-pathcost-first maze routing expansion algorithm.



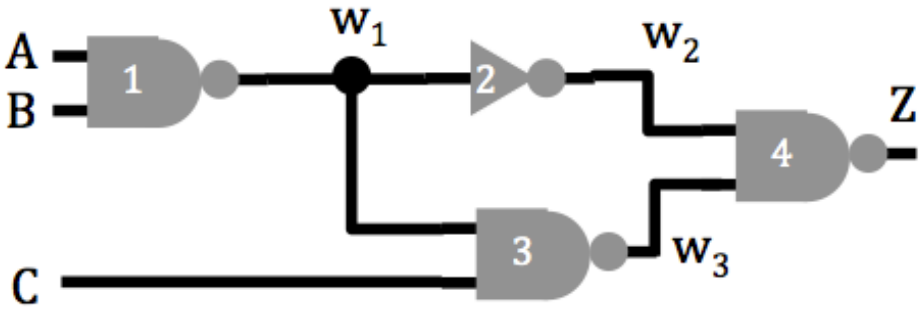
Answer this: what is the **minimum** value of **K** that will **force** the router to choose the final path that (1) uses vias to changes layers, **and** (2) uses none of the high-cost (**K**) cells, in the final path? **K** must be large enough that it is **not** possible to find a path through the high cost cells that has the **same** cost as the path that uses the vias and routes on both layers; in other words, solve for **K** large enough so that there are no "ties" in this cost. Type the **value** of K in the box. For example, if K=1, then you type '1' in the box.

Question 13

Static timing

Consider the small logic network shown below. There are 3 primary inputs (PIs) labeled: **A**, **B**, **C**. There is 1 primary output (PO): **Z**. There are 4 logic gates. There are 3 internal wires that

represent gate outputs/inputs, labeled: **w1**, **w2** and **w3**. The gate delay (cell arc) for each inverter is 2. The gate delay for each NAND2 is 3. Ignore delay for the wires. This logic operates on a clock with a **Cycle Time** = 7.



Do the following:

- Build the **Delay Graph** for this logic network, including one source (**SRC**) and one sink (**SNK**) node, and appropriate edges representing all the delays.
- Walking this graph from SRC to SNK, calculate the **Arrival Time (AT)** for each node in this graph. Just use your eyes to determine the necessary longest paths.
- Walking this graph from SNK to SRC, calculate the **Required Arrival Time (RAT)** for each node in this graph. Again, just use your eyes to determine the necessary longest paths.
- Using these computed ATs and RATs, compute the **Slack** value for each node in this graph.

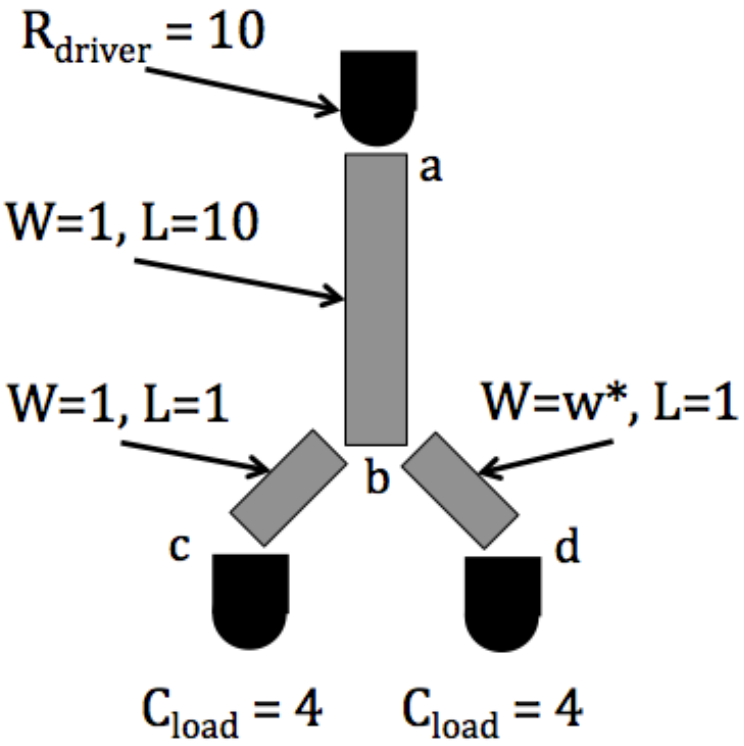
Which of the following are correct statements of the results of this Static Timing Analysis:

- [ ] After the static timing analysis, node SRC is labeled with AT=0, RAT=0, Slack=0.
- [ ] After the static timing analysis, node w3 is labeled with AT=6, RAT=4, Slack=-2.
- [ ] The path SRC, B, w1, w3, Z, SNK is a worst-case longest-delay path, with slack = -2 on each node.
- [ ] The delay graph has 7 nodes.
- [ ] The path SRC, A, w1, w3, Z, SNK is a worst-case longest-delay path, with slack = 0 on each node.
- [ ] The delay graph has 9 nodes including the SRC and the SNK nodes.

Question 14

Elmore delay

Consider the simple 3-terminal interconnect shown below. It is composed of 3 wire segments.



Use the following formulas to compute the **R** and **C** of a wire segment:

- $R = 5L/W$
- $C = 2WL$

Do this: Compute the width **w*** of the bottom right segment such that it forces the left node delay to be exactly **10** time units **slower** than in delay to right load. For example, if the left delay is, say 100, the right delay should be 90. Round your solution to the **nearest integer**. Type the number in the box. For example, if **w***=15.1, then you type '15'.

Question 15

VLSI CAD: LOGIC TO LAYOUT – THE NEXT GENERATION

What topics would you like to see more coverage on – either topics we included in the class already, or new topics you wish we had covered – the next time we offer this MOOC? If you think the class would be better with more background on some topic, or there are topics in the VLSI CAD area you would like us to consider for the next offering of our MOOC, we would like

to know. Similarly, if you think some topics could be shortened or covered in less detail, we would appreciate that input as well. Tell us briefly what you think in the text box below.

Yes – you get points if you type something **longer** than 10 words. (Because we're nice…)

☐ **In accordance with the Coursera Honor Code, I (Matthew Kramer) certify that the answers here are my own work.**

Submit Answers Save Answers

You cannot submit your work until you agree to the Honor Code. Thanks!