CS/EE 5830/6830 -- CAD Assignment #3

Due Tuesday, February 22th, 11:59pm

Use electronic handin to the CAD3 cs6830 handin directory

This assignment is a mini-project to design two 32-bit tree (prefix tree) adders and measure their performance using Spectre (and mixed mode simulation to provide the inputs). You should have two adders assigned to you semi-randomly. If you don't have your assigned adders you need to see me! Everyone will do two different adders, but not everyone gets the same two. Here's the plan:

1. Look carefully at your adders to figure out how to extend them to 32 bits. The 16-bit versions are given in this document (actually they're 15-bit versions... see the discussion in point 2 below). You need to study the connection of the "black," "gray," and "buffer" cells to see how those schemes can be extended to 32 bits. Make sure to include the carry-input at bit 0. Note that the "buffer" cells are not required for functionality. They might not even be needed for speed. Feel free to play around with placement and use of buffers, and explore the possibility of using inverters as buffering elements and using modified black and gray cells, or leaving the buffers out completely. The black and gray cells, of course, are very important.

2. Remember that all the adders have an implicit PG or PAG generator at the top for all bits, and XORs at the bottom for all bits to generate the sum. You'll need to include those in your adder. See Figure 2.13 in your text for an example, or Figure 10.15 from the alternate text that's included in this handout. Also remember that some of the "wires" in the black/gray diagrams are double-wires that have both P and G, and some are single-wires that have just G.

   There are some things you'll need to watch out for in terms of the diagrams included in this document, and the figures in your text. First note that the "gray cell" in the included figures is the same as the "black dot" cell from your text, and the "black cell" in the figures is the "open dot" cell from your text (Section 2.7 in your text).

   Now look at figure 2.20 in your text. This is essentially an 8-bit Sklansky adder as described in Fig. 10.34 included in this handout. Note that the bit numbering is a little different in the two diagrams. In the book the inputs are labeled 7-0 as you would expect, and the C0 input is the carry in to the low order bit. In Figure 10.34 from the other book that's included in this handout, C0 is still the carry in, but the bits are numbed 8-1 which is a little odd. You can see what's going on in the Fig 10.15 from the alternate book: this is a 4-bit ripple carry adder with the PG generators at the top, the Sum generator at the bottom, and the carry cells (grey cells) in the middle. Note that the Sum is labeled 4-1 which is odd. That means that the examples given in Fig. 10.34 in this handout are really for 15-bit adders with the bits labeled 15-1 and the carry in at C0. Odd. So, keep that in mind as you extend your adders to 32 bits. I want a full 32-bit adder with a carry-in input.

3. For this assignment you should use gates from the UofU_Digital_v1_2 library. You can also build special cells directly from nmos and pmos devices from the UofU_Analog_Parts library if you like. You might want a different set of basic gates, or special complex gates, or even to use different techniques than standard static CMOS. Of course, using gates directly from UofU_Digital_v1_2 is fine too.

4. Figure out how to exercise the worst-case delay in your adders, and how to come up with arguments for the adder inputs that will cause that worst case to happen. Highlight the worst-case path that you are measuring on your schematic, or use the black/gray diagram from this assignment, and turn that in with your solution.

5. Build each of your 32-bit prefix adders as schematics. You can move to a larger sheet size if you need to to make things fit. Remember to add an appropriate frame around your schematic no matter what sheet size you use. (Sheet frames are all in UofU_Sheets) Neatness always counts on schematics!

6. Use NC_Verilog (either behavioral or switch-level) to demonstrate on a few test cases that your adder really works. Justify why your test cases are interesting ones.

7. Use mixed-mode analog/digital simulation to apply the arguments that will demonstrate the worst-case delay in your adder and measure the delay with Spectre from the inputs changing to the sum output being correct for the worst case. Measure only the analog delay in the adder itself – not the extra delay in the inverters required for mixed mode simulation, but do measure the delay including the PG and XOR gates if those contribute to the delay.

8. Count the number of transistors in each of your adders. You can do this by counting the transistors in each black and grey cell (or whatever cells you end up using), and then adding up the cells, or you can use LVS to count the cells for you. Details on that are included at the end of this assignment.

9. Use handin to turn in your CAD3 project directory that contains both adders. Remember to include your Verilog test benches (which will be in the run directories, not your project directory, so you'll have to handin those separately), justification for your choice of Verilog functionality test vectors, justification for your worst-case vectors, Spectre timing for the worst case, and count of the number of transistors. These should all be in a README file that you handin. If you are doing anying especially tricky to help speed things up, let us know what you're up to!

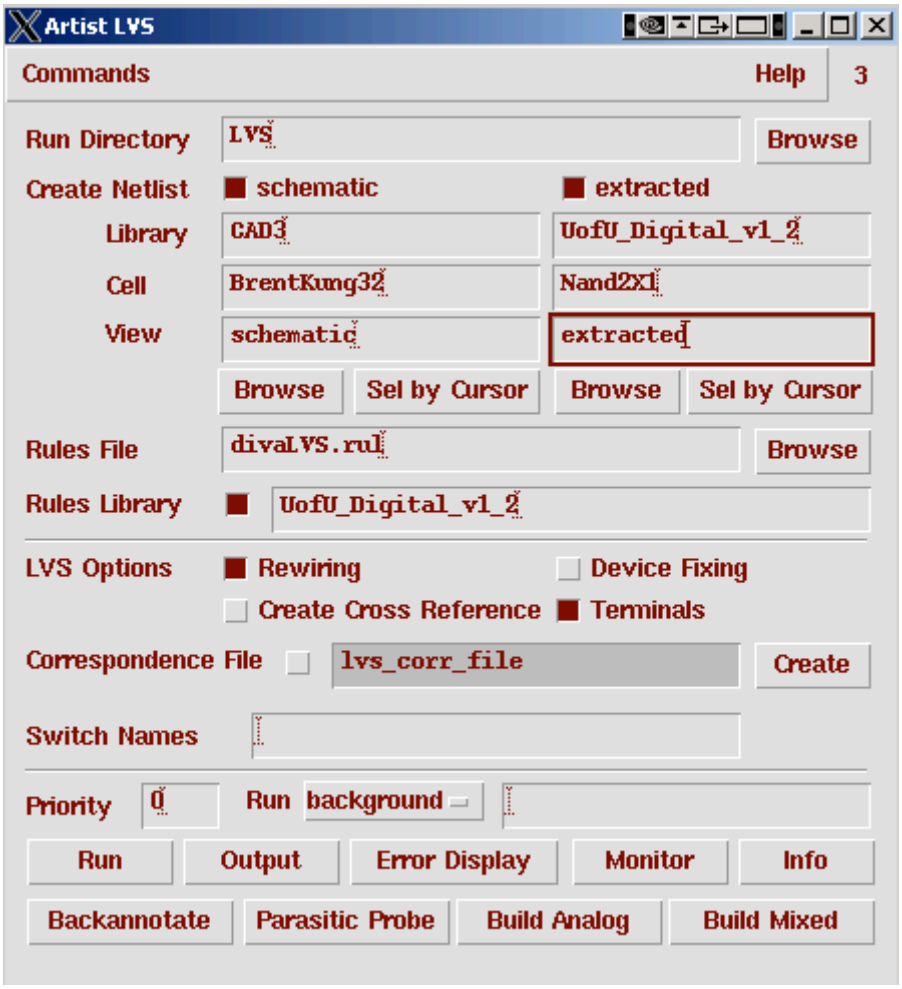
Using LVS to count transistors

LVS is Layout vs. Schematic checking and is really meant to check whether a schematic defines the same transistor network as is extracted from a piece of layout. In our case we'll use a dummy layout (so the check will clearly not match). But, in the process of not matching, LVS will report how many transistors of each type there are in your schematic.

Start by opening up a layout view of a cell from the UofU_Digital_v1_2 library. Any cell will do. For example, pick the NAND2X1 and open the layout view. You'll get a view of the composite layout of the NAND gate. If you haven't seen layouts before, don't worry: you don't need to know anything about layouts for the following procedure.

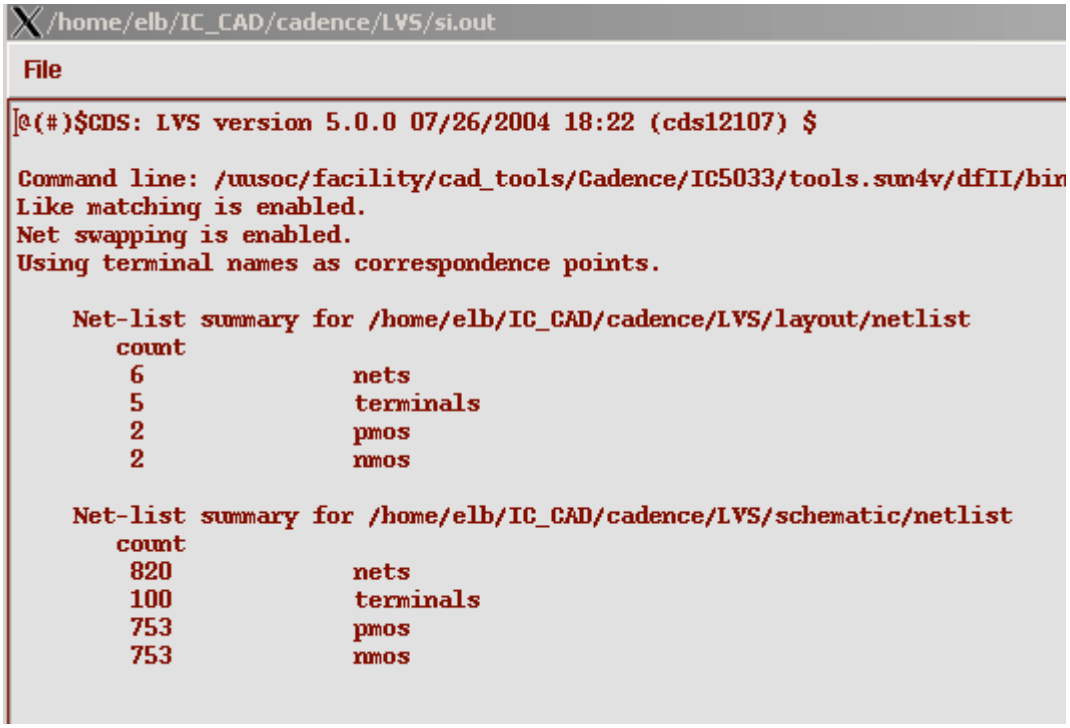
Now select **Verify->LVS** in the menu. You will get a dialog box for the LVS process. LVS is a procedure for checking that a schematic has the same transistor netlist as a piece of layout. So, in order to do this it needs to extract the complete (flattened) transistor netlist of both the schematic and the layout and compare them to see if they're the same. In this case they won't even be close! But because the LVS process needs to count the number of transistors, it's useful for this sneaky purpose.

You now fill in the dialog box with the name of a schematic (your top-level adder schematic in this case) and an extracted layout view (the NAND2X1 extracted view, or any extracted layout view you have sitting around). Here's an example:



Then press the Run button. You'll get a note in the CIW that the LVS process has started. When it's finished (in general this is a very hard graph-matching problem so for large circuits it can take quite a while – in our case it will be quite fast because these are pretty small circuits) you'll get a box that says that the LVS process was successful. Don't be confused. It's just saying that the process finished, not that it actually matched your circuits!

Now click on Output to get the results of your LVS. The top part of that window will look something like this:



so for this example, the NAND gate (the layout view) had 4 transistors (which is what we expected: 2 nmos and 2 pmos), and the 32-bit adder (the schematic view) (a Brent-Kung in this example) had 1,506 transistors (753 pmos and 753 nmos). And best of all, you didn't have to count them by hand!

The following figures are from an alternate text – Weste and Harris CMOS VLSI Design, 3rd Ed. They are the figures that you should follow when you are designing your adders.

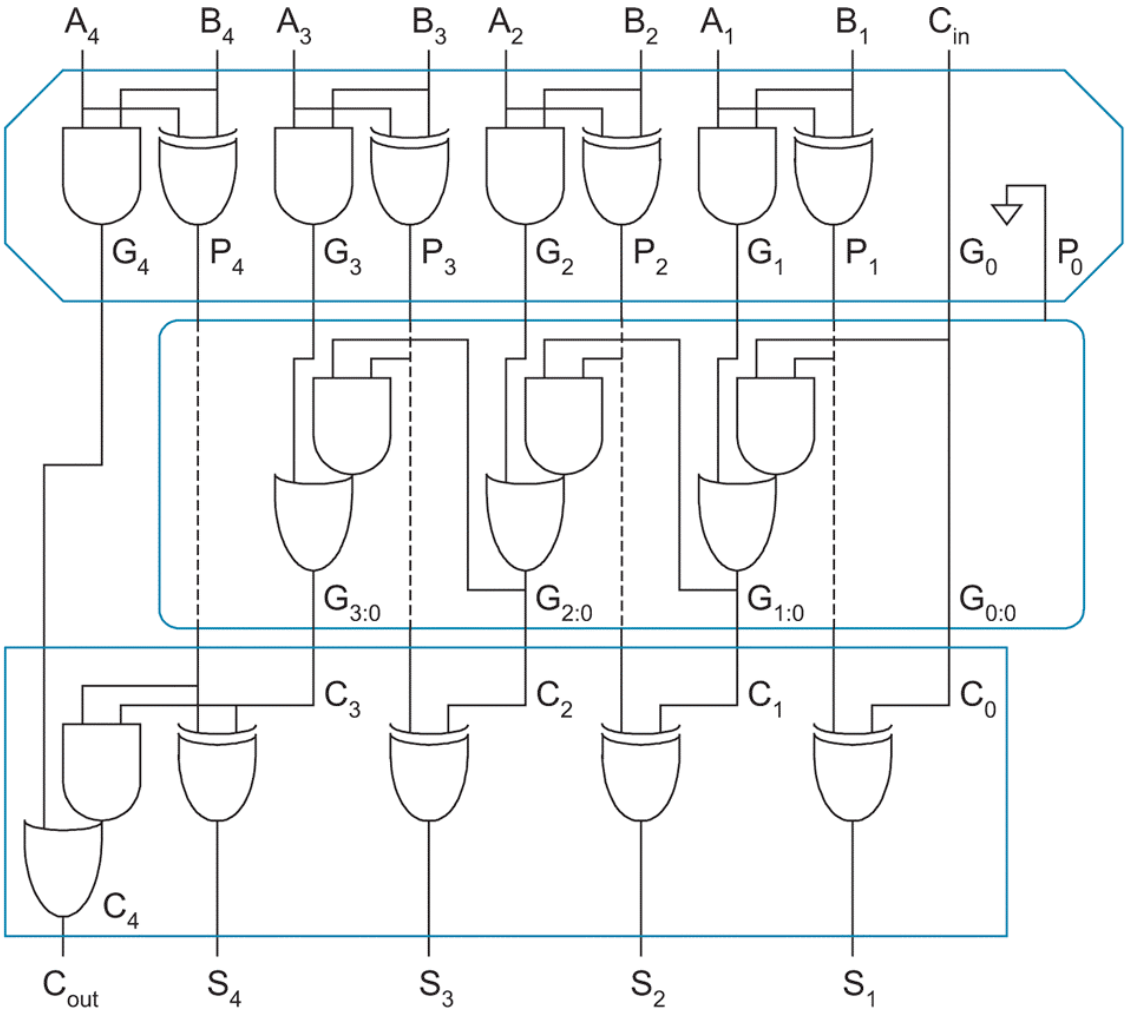


FIG 10.15 4-bit carry-ripple adder using PG logic

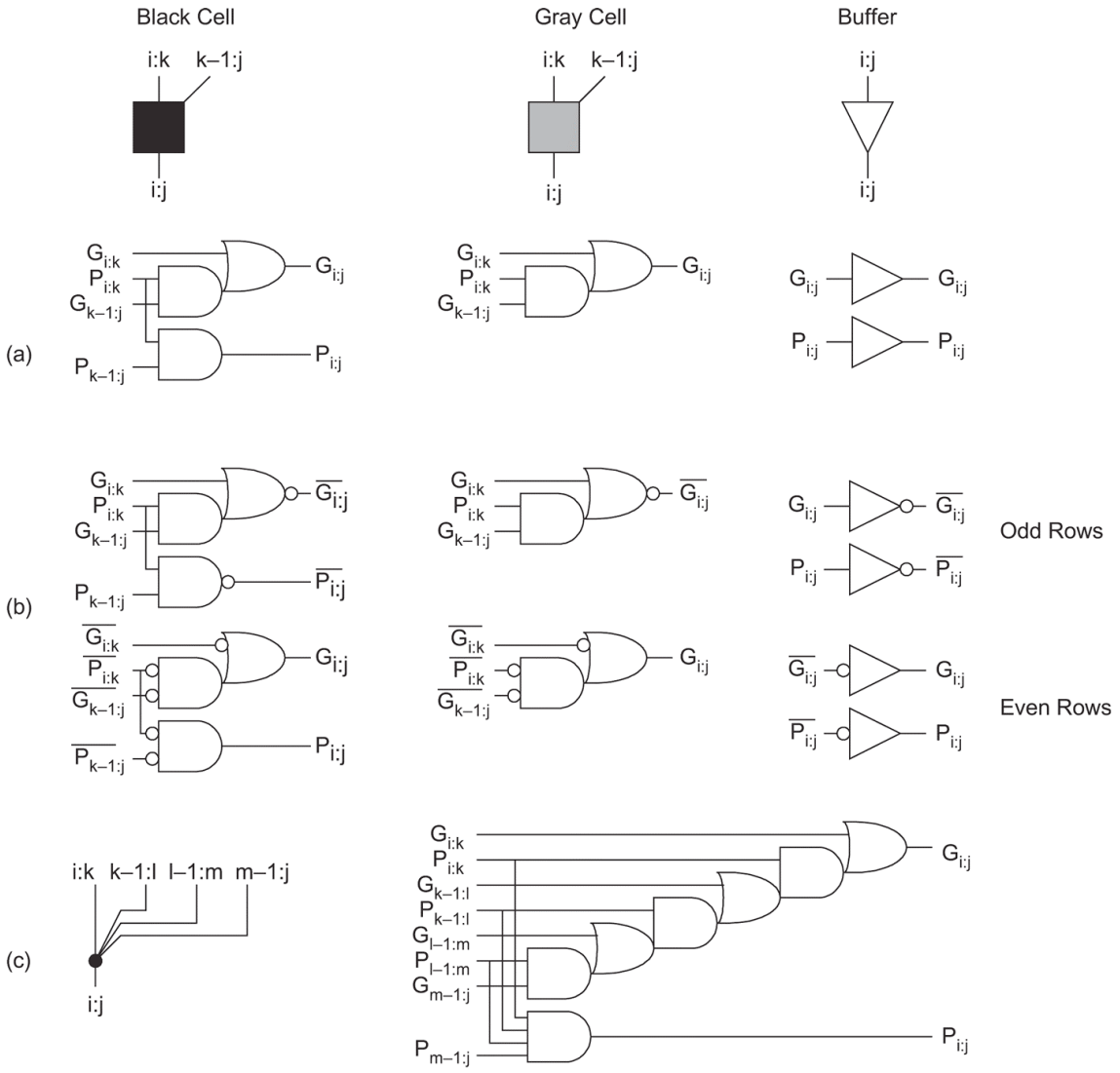


FIG 10.17 Group PG cells

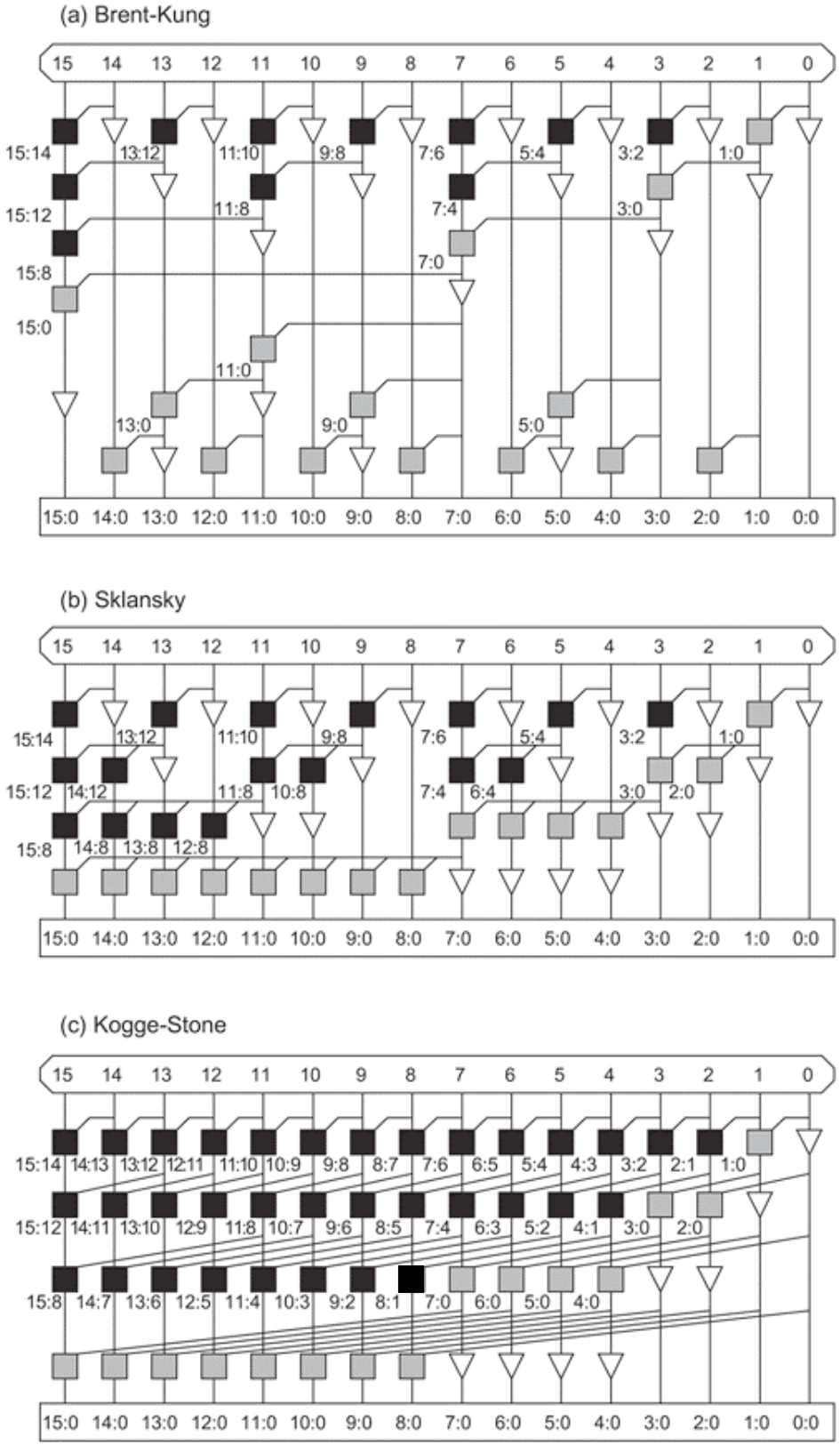


FIG 10.34 Tree adder PG networks

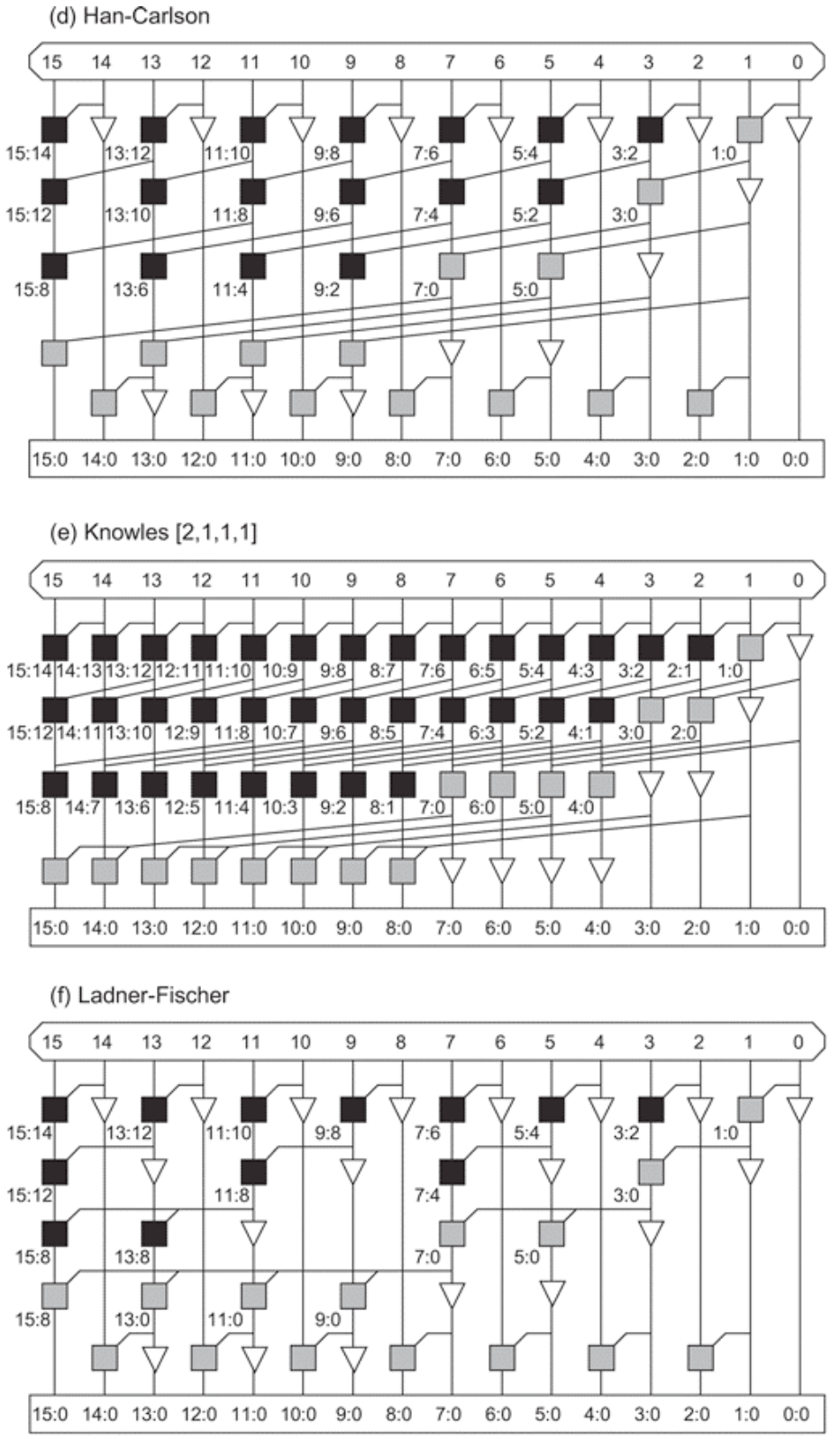


Figure 10.34 continued.