A Greedy Router with Technology Targetable Output

R. Balakrishnan and R. F. Hobson*
Schools of Computing and Engineering Science;
Simon Fraser University; Burnaby, B. C., V5A 1S6; rick@cs.sfu.ca

Abstract

Our objective was to integrate an effective channel routing algorithm with the Chip Design Language (CDL) algorithmic layout tool. CDL uses technology targetable layout techniques, so that the output of the routing algorithm can easily be ported to different technologies. We introduce the technology independent features of CDL and describe how a greedy router can be interfaced to it. Specific features of interest include mapping from the grid based router to the gridless CDL environment, and the automatic insertion of CDL feed-through cells in multi-channel applications.

1 Introduction

Automating the routing of integrated circuit netlist connections has been of interest for some time [6, 7, 8]. There are channel routers based on maze routing algorithms, some based on Left Edge Algorithms, algorithms based on heuristics, channel routers that employ simulated annealing, some based on graphs, and many others [8]. As more metal routing layers become available, Over-The-Cell (OTC) techniques make channel-less routing more feasible [7]. Performance-driven algorithms are also of interest for some applications [6].

Some routing algorithms do not include an interface to a real cell library, stopping at a data structure or symbolic layout level. This reduces dependency on technology, but does not remove the need for a real back-end interface to a specific cell library. Maintaining cell libraries and migrating them from one technology to another can be a time consuming activity. Our Chip Design Language (CDL) tool has been evolving to simplify this task. Since CDL includes a high degree of technology targetability, we have found that it makes a good intermediate level output notation for a router.

This paper introduces some of the technology targetability features of CDL and then describes our initial choice of router algorithms, and how the algorithm was interfaced to CDL.

*This work has been supported by the B.C. ASI, HP-IDACOM Ltd., Micronet, NSERC, and PMC-Sierra Inc.

2 CDL Technology Targetability

CDL has evolved over a period of 13 years from being quite technology dependent to being almost totally technology independent [2]. The basic primitives in CDL have their origins in the days of CIF code [3]. To side-step the burden of constant technology upgrades, technology targetable cell library and chip layout techniques have been woven into CDL. All CDL support is based on the C programming language and the UNIX operating system environment.

To start with, all technology physical parameters are given symbolic names that can be "#INCLUDE'd" into a C program. For example, first layer metal width and spacing are Mwd, and Msp. Sometimes half values are also used, e.g. Mwd2, Msp2. These are rounded up to the nearest minimum-resolution grid value for the technology (represented by U, in CDL). Heavy use is made of C macros to define contacts. For example, Lmp(x,y), places a M2-M1-Poly contact at the coordinates (x,y); Mnd(x,y) places a M1-Ndev contact.

The CDL primitives required for auto-routing standard cells are shown as part of Table 2. All of the constants and macros are brought in with the #INCLUDE <cdl.h>. Standard cells are introduced by *splace*, and *srplace*. *splace* places the named cell at the given coordinates, whereas *srplace* does a relative placement with respect to the cell number given as its first argument (default in East direction). Connection points (pins, or nodes) within a standard cell are located by the *locn* primitive. Referencing nodes and pins by name helps to limit technology dependency.

Routing with Metal2 (Lwire) and Metal1 (Mwire) wires makes use of M2-M1 contacts, Lm(x,y); contact center-center spacing, Lmcc; and occasionally wire center-center spacing, Lcc, Mcc. For the router, horizontal routing is done in Metal1 and vertical routing is done in Metal2 (this is not a CDL limitation). Horizontal wire spacing could either be the worst case contact-contact, Lmcc, or the

0-7695-0104-4/99 $10.00 © 1999 IEEE

contact-to-wire spacing, LmMcc. This provides a chance to optimize spacing when M2-M1 contact congestion is not a problem.

Other CDL standard constants required are the height of the first possible M2/M1 contact above/below a gate channel, HLmtop, HLmbot. These are used to start wiring channels.

The *Hcon* function is used to directly connect the output of one cell to the input of an adjacent cell. This often saves a wiring track.

CDL's UNIX environment contains environment variables to select cell libraries and technology files. Simple changes to the environment followed by a recompile are all that must be done to switch to a new technology. Each technology has its own layout data base directories.

3 A Router with CDL Output

After examining several routing algorithms, the greedy algorithm by Rivest and Fiduccia was chosen for initial CDL interface experiments [5, 1] (call it the RF algorithm). Although the RF algorithm is based on heuristics, it performs well, and the possibility for local optimization based on specialized maze routing techniques could be added later, as in [4].

The RF algorithm scans a wiring channel from left to right, completing the wiring at each column (of a virtual grid) in a *greedy* manner, before proceeding to the next column [5]. The wiring choices at each column are made by evaluating a series of 6 steps which define operations such as: connecting a gate input to a channel track (top or bottom); adding new wiring tracks; jogging a net on one track to another track; and merging 2 tracks. The inputs are: (1) the list of terminals (connection points or netlist names) at the top and bottom; (2) the left and right set – a list of floating wires that are to be brought to the edge of the wiring channel.

Fig. 1 shows the overall flow of information through our router. To eventually support automated synthesis, we start with a Verilog format input file. This file is parsed by a Pre-router, which we will eventually modify to include automatic placement. For now the placement is done according to the order in the Verilog file. This can be hand edited if necessary to obtain better placements. An example of a 2-input D-flip-flop (D2FF) is shown in Table 1. The pre-router needs to consult CDL libraries to verify port maps and prepare sorted terminal lists for the Auto-Router. This information is output in an Intermediate Netlist File (INF). The Auto-Router parses the INF, calls the Greedy Router, and outputs a CDL source file, which must be compiled and run to generate a physical layout. Part of the D2FF CDL code file is shown in Table 2, and a CMOS physical layout is shown in Fig. 3.

Although the final output is technology targetable, the router heuristics make use of some information encoded in CDL constants, such as Lmcc, LmLcc, and Lcc. Thus changing technology requires running the router for each technology of interest. As mentioned above, this is a simple UNIX environment change followed by a recompile and rerun.

One of the first problems to be encountered is that the Auto-Router works best on a virtual grid whereas CDL cell libraries are gridless. Fortunately, CDL library cells leave enough room to route any input or output connection either up or down through the cell. This meant that the connection pitch inside of a cell was at least as great at the Auto-Router's virtual grid pitch, so the problem was reduced to a small misalignment between the cell coordinates and the virtual grid. This problem was solved as follows.

Fig. 2 shows how a pair of wires have to jog to connect from a cell coordinate system to the virtual grid. The maximum amount of jog is half the grid spacing. With manhattan wiring, a spacing violation can occur, as shown on the left of Fig. 2. Using 45 degree joging plus a small offset in the wire height, as shown on the right of Fig. 2, solves this problem. There is enough room in our standard cells to jog over a dozen wires before running out of room. Fortunately, there is a natural grid re-alignment at regular intervals due to the internal structure and spacing of standard cells. Thus the joging height adjustment can be reset at regular intervals so that it is extremely unlikely that we would ever run out of space.

The Auto-Router can handle more than 2 rows of standard cells. ROW-breaks are either implied by defaults, or by editing the Intermediate Netlist file and placing explicit ROW-break directives. Since feed-throughs (metal openings which leave room for OTC routing) do not occur naturally in our standard cells, we have added special *feed-through* cells to the cell library. When the Auto-Router realizes that a vertical connection is necessary from Row1 to Row3, say, it first looks to see if there is a common terminal in Row2. If not it inserts a *feed-through* cell in Row2 between 2 other standard cells. The *feed-through* terminal is labeled with the common netname so it will be routed like a regular net. With extra layers of metal, one could also use Metal3 to perform OTC routing.

4 Conclusion

We have shown the main technology targetable features of a Greedy Router to CDL interface. CDL has proven capabilities for *megacell* generation for such important blocks as Static Random Access Memories, First-In-First-Out Memories, Data-Path generation, etc. Adding standard cell routing is a good step towards a fully automated technology targetable CDL layout environment.

Table 1. D2FF example Verilog input.

```
module D2FF ( d0, d1, lde0, lde1, outez,q);
input  d0, d1, lde0, lde1, outez; output q;
wire q_int2,n52,n53,n54,n55,n56,
        \*cell*8/U5/Z_0;
AOI22 U20 ( .a1(n52), .a2(n53), .b1(d0),
        .b2(lde0), .outz(\*cell*8/U5/Z_0));
AOI22 U21 (.a1(\*cell*8/U5/Z_0 ),.a2(n54),
        .b1(q_int2), .b2(outez), .outz(q));
OAI22 U22 (.a1(\*cell*8/U5/Z_0 ),.a2(lde1),
        .b1(n55), .b2(n56), .outz( n52));
 INV U23 ( .in(q), .outz(q_int2) );
 INV U24 ( .in(lde1), .outz(n56) );
 INV U25 ( .in(lde0), .outz(n53) );
 INV U26 ( .in(d1), .outz(n55) );
 INV U27 ( .in(outez), .outz(n54) );
endmodule
```

References

[1] R. Balakrishnan. A greedy algorithm based router. Master's thesis, Simon Fraser University, 1998.

[2] R. Hobson and D. Smith. Cdl: A tool for algorithmic vlsi layout and design. Technical report, School of Computing Science, Simon Fraser University, 1996.

[3] R. W. Hon and C. H. Sequin. *A Guide to LSI Implementation*. Xerox, PARC, 1980.

[4] J. Reed, A. Sangiovanni-Vincentelli, and M. Santomauro. A new symbolic channel router: Yacr2. *IEEE Transactions on CAD*, CAD-4(3):208–219, 1985.

[5] R. L. Rivest and C. Fiduccia. A 'greedy' channel router. In *Proc. Design Automation Conference*, pages 418–424. IEEE Press, 1982.

[6] N. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic, 2nd edition, 1995.

[7] N. Sherwani, S. Bhingarde, and A. Panyam. *Routing in the Third Dimension. From VLSI Chips to MCMs*. IEEE Press, 1995.

[8] G. Zobrist, editor. *Routing, Placement and Partitioning*, chapter Routing Techniques, by Zargham, M.R. Alex Publishing, New Jersey, 1994.

Table 2. Partial CDL code listing for D2FF.

```
#include <cdl.h>
main() { start_cell("D2FF"); {
  int x, y, xr, yr;
  int x1, y1, x2, y2, x3, y3, y4, step;
  int aoi22b_1, aoi22b_2, inv_a_3, inv_a_4;
  int oai22b_5, inv_a_6, inv_a_7, inv_a_8;
  step = 6 * U;
/* Cell row : 0 */
  x = 0, y = 0; xr = 0, yr = HLmbot;
aoi22b_1= splace("aoi22b", x, y); /* U20
aoi22b_2=srplace(aoi22b_1,"aoi22b"); /* U21
inv_a_3 = srplace(aoi22b_2,"inv_a"); /* U26
inv_a_4 = srplace(inv_a_3, "inv_a"); /* U27
  Hcon(aoi22b_1,"outz", aoi22b_2, "a1",MP);
/* Cell row : 1 */ xr = 0, yr = y + HLmbot;
  x = 0, y  -= 8 * Lmcc + HLmtop - HLmbot;
oai22b_5 = splace("oai22b", x, y); /* U22
inv_a_6 = srplace(oai22b_5,"inv_a"); /* U23
inv_a_7 = srplace(inv_a_6, "inv_a"); /* U24
inv_a_8 = srplace(inv_a_7, "inv_a"); /* U25
/* Net : n52 */
  locn(aoi22b_1, "a1", &x1, &y1);
  Lmp(x1, y1);
  x2 = xr + 1 * Lmcc;  y3 = yr+Lmcc+0*step;
  y2 = yr - 1 * Lmcc;
  if (x1 == x2)
    { Lwire(x1, y1, x2, y2, END); }
  else
{Lwire(x2,y2,x2,y3,x1,y3+x2-x1,x1,y1,END);}
  Lm(x2, y2);
  x1 = xr + 1 * Lmcc; x2 = xr + 9 * Lmcc;
  y1 = y2 = yr - 1 * Lmcc;
  Mwire(x1-Mwd2, y1, x2, y2, END);
 locn(oai22b_5, "outz", &x1, &y1);
  Lm(x1, y1);
  x2 = xr + 6 * Lmcc;  y3 = yr - 9 * Lmcc;
  y3 = y3  - 1 * step; y2 = yr - 5 * Lmcc;
  if (x1 == x2)
    { Lwire(x1, y1, x2, y2, END); }
  else
{Lwire(x2,y2,x2,y3,x1,y3-abs(x2-x1),
                              x1,y1,END);}
  x1 = xr + 6 * Lmcc; x2 = xr + 9 * Lmcc;
  y1 = y2 = yr - 5 * Lmcc;
  Mwire(x1-Mwd2, y1, x2, y2, END);
  x1 = x2 = xr + 9 * Lmcc;
  y1 = yr - 1 * Lmcc; y2 = yr - 5 * Lmcc;
  Lwire(x1, y1, x1, y2, END);
  Lm(x1, y1); Lm(x1, y2);
/* etc. ... */  } end_cell();  exit(0); }
```

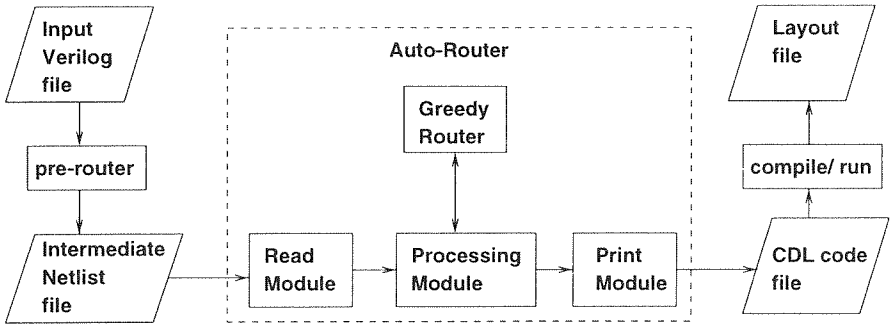

Figure 1. The router information flow.

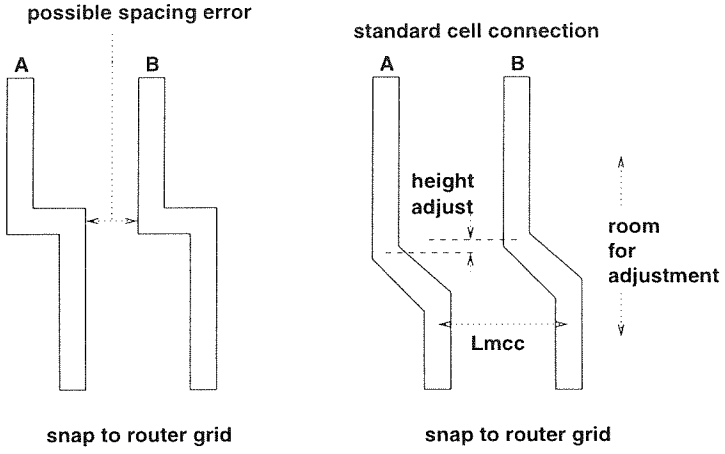

Figure 2. Jogging wires to connect from a cell to the virtual grid.

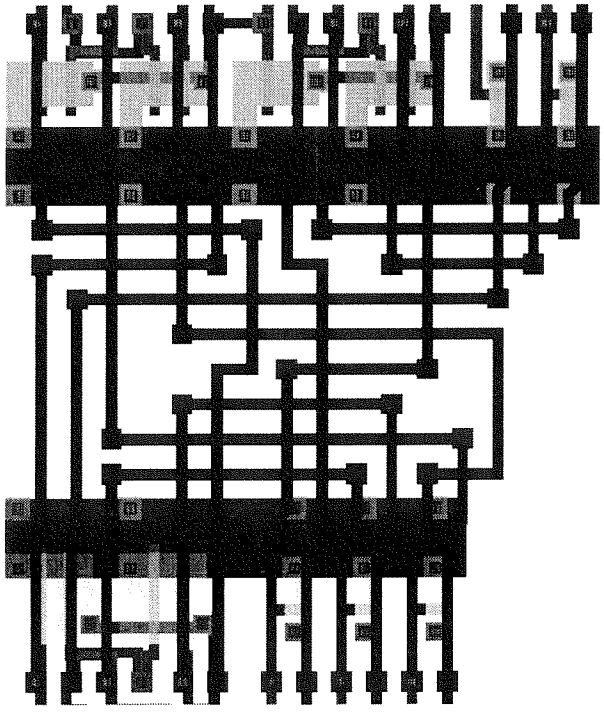

Figure 3. D2FF Auto-Routed Layout.