RC22342 (W0202-050) February 20, 2002
Electrical Engineering

IBM Research Report

Programming-Based Mixed Custom/Semi-Custom Design Methodology for Low-Power, High-Performance VLSI

George D. Gristede, Wei Hwang, Suhwan Kim, Stephen Kosonocky
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Programming-Based Mixed Custom/Semi-Custom Design Methodology for Low-Power, High-Performance VLSI

George D. Gristede, Wei Hwang, Suhwan Kim, and Stephen Kosonocky
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract

In this paper, a novel design methodology for low-power, high-performance VLSI is proposed allowing mixed, custom/semi-custom designs in much less time than previously possible. Schematics/layouts are generated by programs calling library functions layered above design system programming languages, fully exploiting hierarchy and repetition, maximizing technology independence and re-mapping efficiency, and optimizing power/delay trade-offs. The methodology has been demonstrated in 0.18μm and 0.13μm CMOS technologies and is being used for the datapath and register arrays of an ultra-low-power, high-performance embedded processor and system-on-chip (SoC) designs.

1 Introduction

Today's VLSI design teams face a number of difficult challenges. Prominent among them are requirements on the power and/or performance optimization of circuits, shortening design cycles that result from increasing time-to-market and the growing complexity of designing [1]. Even though logic synthesis and physical layout optimization tools are used, during the process of designing low-power, high-performance VLSI circuits, designers often find that their designs do not meet the timing and/or power constraints after layout. This problem is compounded as feature sizes shrink to quarter-micron and below i.e, the so-called deep sub-micron (DSM) process technologies [2], [3].

In the current state of the art designers still use manual, custom, graphical editing to generate schematics and layout for power and/or performance sensitive macros such as arithmetic and logic units (ALUs), register files, SRAMs, and embedded DRAMs. To cope with DSM design challenges, a different design methodology is required where designers are able to rapidly explore design alternatives in both the schematic and the layout that will impact performance and power dissipation as quickly as possible. Traditional, manual editing of schematics and layouts provides severe limitations on the designer's ability to quickly explore such design alternatives.

The methodology described in this paper employs an extensive layering of programming functions on top of the programming functions typically provided by VLSI design systems (eg. the SKILL programming language for the Cadence design system) to allow users to efficiently automate many tasks (creation of schematics, layouts etc.) in the design of low-power, high-performance VLSI circuits. The functions in these layers are user-written, typically by one or two people on a given design team. The designers can then generate schematics, layouts etc. by writing small programs that call these functions. The functions can be grouped into project dependent/independent and technology dependent/independent groups for maximum re-usability. Using this approach, design remapping to different technologies becomes much easier as technology changes typically show up as changes to values in a look-up table used by the designer-written programs that created the design.

In many VLSI structures (adders, multipliers, shifters, register files, SRAMs, embedded DRAMs etc.) there is a significant amount of exact and inexact repetition which can be fully and efficiently exploited via programming loops and nested function calls. Since each design is described by one or more software programs in text form, it has the highest level of design integrity against file system corruption and can be transferred more easily from one design team to another. Moreover, The use of parameterized schematics/layouts [4] fits nicely into this methodology as the programming functions can automatically

ensure that the sizes of transistors in the layout match the sizes of their schematic counterparts. This is vital for low-power design as an automatic circuit tuner is most often used to minimize the power while maintaining acceptable performance.

2 Design Flow

The design flow used to implement semi-custom designs built from parameterized and full-custom cells is summarized in Fig. 1. This flow is mainly implemented with user-written SKILL functions (all boxes with rounded edges).

2.1 Schematic Design

The design flow begins with the definition of the specifications for the circuit. The architecture is then designed to meet these specifications. Circuit schematics are then generated using custom and/or parameterized cells. Where appropriate, the leaf cells are created manually while the upper level blocks are created using SKILL code leveraging function libraries to exploit repetition and hierarchy. The resulting schematics are then simulated with IBM's PowerSpice circuit simulator and functionally verified with IBM's Verity formal verification tool. The circuit is then tuned using IBM's SST (pattern dependent) and EinsTuner (pattern-independent) circuit tuners to achieve an optimal power/delay trade-off [5], [6], [7].

2.2 Circuit Tuning

During the tuning of circuits containing parameterized and full-custom cells, we use the cost function

$$\text{cost} = \alpha \cdot T_{cost} + (1 - \alpha) \cdot P_{cost}$$

where T_{cost} is a delay cost term, P_{cost} is a power cost term and $0 \leq \alpha \leq 1$. The T_{cost} and P_{cost} terms are usually weighted summations of other terms. For example, we might have

$$T_{cost} = \sum_i a_i T_{cost,i}$$

where $T_{cost,i} = T_i^{k_i}$. T_i is the ith delay and k_i is a positive integer. Now suppose that we wanted a particular T_i to be less than or equal to a target value $T_{i,target}$. We could use

$$T_{cost,i} = \max\{0, T_i - T_{i,target}\}^{k_i}$$

Similar results can be obtained for the power cost term P_{cost}. By varying α we can obtain a continuous trade-off between delay and power, and the circuit tuner produces the appropriate changes in the transistor dimensions.

The parameterized cells stated previously fits nicely into our methodology, especially during circuit tuning iterations where changes in transistor sizes in the schematic can be immediately transferred to the layout via the layout generation program.

2.3 Physical Design

All circuit layouts except for the lowest level custom cells are generated using SKILL code calling functions from the methodology function libraries. The creation of circuit layout begins with the initialization of look-up tables containing project and technology data. A typical example of the type of data in the project look-up table for a 40-bit datapath image might be the number of tracks per bit and the track width and pitch. Project independent functions can then act on this data to quickly and efficiently calculate the coordinates of datapath tracks for routing and pin placement.

After the project and technology data structures have been initialized, placement of the cells and top-level pins is easily accomplished with function calls from the function libraries. At this step the transistor size parameters for any parameterized cells are set. The cell placement functions are designed so the designer can easily explore many placement alternatives by merely changing a few lines in the SKILL program while letting the computer do the work of calculating coordinates, spacings (absolute and relative) etc.

Once the cell placement has been accomplished, the process of routing the nets begins. The designer can choose any mix of custom/automatic routing in this methodology. Power and global signal striping are easily handled by function calls in typically just a few lines of SKILL code. Some of these striping functions are project dependent (eg. n-well and substrate bias lines etc.). General full custom signal routing is accomplished for a given net by the following procedure:

1. Obtain the list of pins and instance pins the net is connected to from the schematic (The function

call that handles this automatically flattens vector nets etc.).
2. Calculate coordinates of pins from step 1 and wiring tracks in layout via function calls.
3. Create pins, wires and contacts and assign these shapes to the specified net in the design system database. This step typically calls a large variety of nested functions for coordinate calculation, shape creation etc.

For signals that have similar routes, a function can be written by the user which can use one or more nested loops to apply steps 1-3 to all of the signals in the group.

For nets that the designer desires to be routed automatically, the methodology provides both an automatic, proprietary grid-based routing function and easy access to commercial routing engines (i.e. Virtuoso Custom Router by Cadence etc.).

A major feature of this methodology is that during the layout process, as the designer is exploring alternatives to reduce wire length and power, entire groups of nets can be added/deleted to/from the layout by merely uncommenting/commenting function calls and/or blocks of code. This is a tremendous advantage for low-power designers who need to minimize power as much as possible while maintaining acceptable performance.

2.4 Layout Extraction and Circuit Retuning

Once the layout has been completed, schematic extraction occurs and the circuit power and timing are re-checked using the extracted schematic. If the desired performance is not obtained, the circuit is retuned by the circuit tuner and the program that generated the layout is re-run. If parameterized cells are used in the schematic and layout, this process becomes very efficient and the transfer of the updated transistor sizes to the layout takes from just seconds to a few minutes (depending on the circuit size and topology) for a full custom circuit (adder, multiplier, register file etc.).

3 Re-usability Across Technologies and Projects

The proposed methodology significantly enhances the migration of designs across technologies, projects and architectures. Shown in Figure 2 is a diagram of the methodology illustrating the infrastructure needed for technology re-mapping and project re-usability. Technology ground rules can be directly accessed via function calls (see `techGetParam()` function references in SKILL code in Figure 7) and used in coordinate calculations. If the technology changes, the code can automatically make the appropriate changes in the layout. This greatly facilitates re-mapping as the technology rules are already "programmed in". Project-specific data such as datapath bit/wire pitches, tracks per bit, track width, bit orientation, etc. can all be stored in a look-up table for reference by project level functions and designers in their code. This allows for easy conversion of designs across projects. In addition, it is much easier to derive circuits from other, similar circuits by adding minor modifications/enhancements to the code for the original circuits. If a designer has code for a 16-bit adder and wants to generate a 40-bit adder, for example, the designer can usually easily make such modifications/extensions to the code and get the desired 40-bit adder, as shown in Figure 3.

4 Case Studies

4.1 Datapath Examples

Shown in Figure 4 and 5 are 16-bit and 40-bit datapaths, respectively, consisting of adders, shifters, multipliers etc. These datapaths and their underlying macros were assembled entirely with our programming-based design methodology described in this paper.

4.2 An Register File Example

Shown in Figure 6 is a multi-port register file (2 read/2 write ports) in which the array of the register file was dimensioned at 16-words × 40-bits. This register file was also assembled entirely with our design methodology.

5 Conclusion

A novel programming-based methodology for the rapid, efficient generation of low-power, high-performance VLSI hardware has been presented. The methodology has been demonstrated in 0.18μm and 0.13μm CMOS technologies with great success. In each case, large macros were assembled quickly and DRC/LVS problems were minimal or non-existent. Full custom macros which used to take months to assemble now could be assembled in just a week or two. The methodology is currently being used for the low-power, high-performance datapath arithmetic macros, register files and memories of an ultra-low-power, high-performance embedded processor. The methodology is also being applied to the rapid, efficient development of system-on-a-chip (SoC) designs.

References

[1] J. M. Rabaey and M. Pedram, *Low Power Design Methodologies*. Kluwer Academic Publishers, 1996.

[2] K. Roy and S. C. Prasad, *Low-Power CMOS VLSI Circuit Design*. Wiley-Interscience, 2000.

[3] W. H. Kao, C. Lo, M. Basel, and R. Singh, "Parasitic extraction: Current state and the art and future trends," *Proceedings of the IEEE*, vol. 89, pp. 729–739, May 2001.

[4] G. A. Northrop and P. Lu, "A semin-custom design flow in high-performance microprocessor deisgn," in *Proceedings of the Design Automation Conference*, pp. 426–431, June 2001.

[5] A. R. Conn, I. M. Elfadel, W. W. Molzen, Jr., P. R. O'Brien, P. N. Strenski, C. Visweswariah, and C. B. Whan, "Gradient-based optimization of custom circuits using a static-timing formulation," *Proc. 1999 Design Automation Conference*, pp. 452–459, June 1999.

[6] C. Visweswariah and A. R. Conn, "Formulation of static circuit optimization with reduced size, degeneracy and redundancy by timing graph manipulation," *IEEE International Conference on Computer-Aided Design*, pp. 244–251, November 1999.

[7] G. D. Gristede, W. Hwang, and C. Tretz, "Method and system for tunning of components for integrated circuits," *U.S. Patent No. 6,219,822*, 2001.

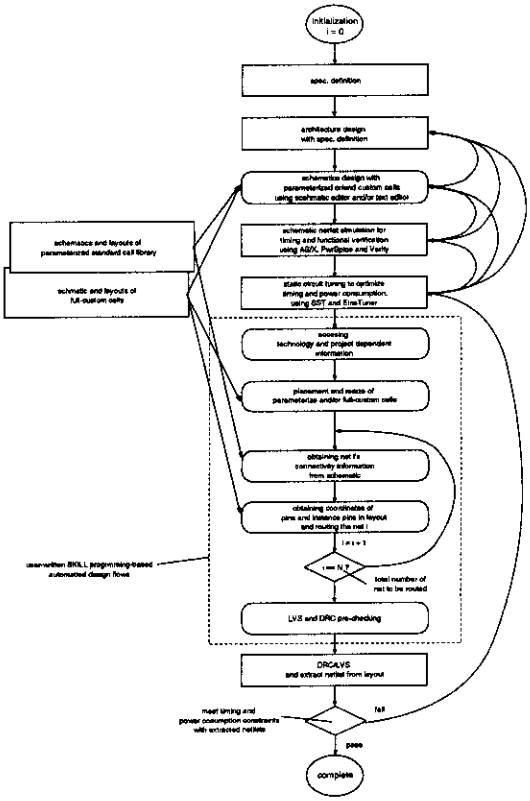

Figure 1. Complete design flow using parameterized and full-custom cells.

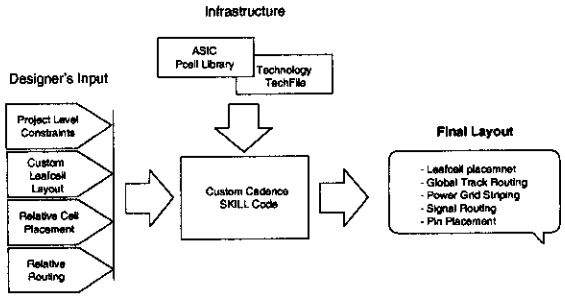

Figure 2. Skill-based design methodology for parameterized and/or full-custom cells.

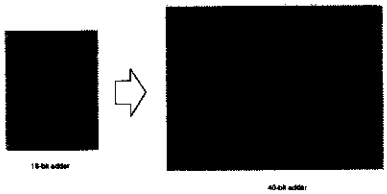

Figure 3. Conversion of 16-bit adder to 40-bit adder.

Figure 4. Physical layout of 16-bit datapath.

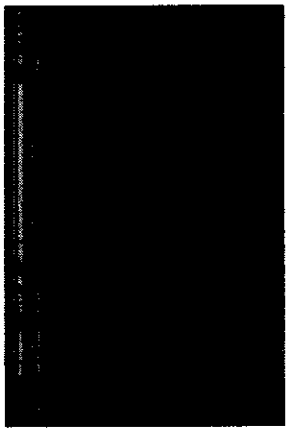

Figure 5. Physical layout of 40-bit datapath.

Figure 6. Physical layout of register file

```
...
foreach(s1 list("ra" "wa")
  for(i 1 2
    for(j 0 3
      foreach(s2 list("T" "C")
        net=dbMakeNet(layout sprintf(nil "%s_%L%s<%L>" s1 i s2 j))
        connec-
tions=GDGSchematicGetSignalConnections(schematic net~>name)
        outputConnections=setof(p connec-
tions nthelem(3 p)=="output")
        inputConnections=setof(p connec-
tions nthelem(3 p)=="input")
        x=m2Tracks[net~>name]
        foreach(connection inputConnections
          pins=GDGLayoutFindPinsInHierarchy(layout list(nthelem(1 connection))
            nthelem(2 connection))
          p1=car(car(pins))
          p2=x:yCoord(p1)
          dbCreatePath(layout list("M3" "drawing") list(p1
            GDGMathVectorAdd(p2 -d1:0)) techGet-
Param(techId "grM3Width")
            "extendExtend")~>net=net
          dbCreateRect(layout list("V2" "draw-
ing") GDGMathBBoxSquare(p2
            techGetParam(techId "grV2Width")))~>net=net
        )
        foreach(connection outputConnections
          pins=GDGLayoutFindPinsInHierarchy(layout list(nthelem(1 connection))
            nthelem(2 connection))
          p1=car(car(pins))
          p2=x:yCoord(p1)
          dbCreatePath(layout list("M1" "drawing") list(p1
            GDGMathVectorAdd(p2 d1:0)) techGet-
Param(techId "grM1Width")
            "extendExtend")~>net=net
          dbCreateRect(layout list("V1" "draw-
ing") GDGMathBBoxSquare(p2
            techGetParam(techId "grV1Width")))~>net=net
        )
        pList=setof(p net~>figs mem-
ber(p~>layerName list("V1" "V2"))!=nil)
        pList=foreach(mapcar p pList GDGMathBBoxCenter(p~>bBox))
        pList=sort(pList lambda((p1 p2) yCoord(p1)<yCoord(p2)))
        p1=car(pList)
        p2=car(last(pList))
        dbCreatePath(layout list("M2" "draw-
ing") list(GDGMathVectorAdd(p1
          0:-d1) GDGMathVectorAdd(p2 0:d1)) techGet-
Param(techId "grM2Width")
          "extendExtend")~>net=net
      )
    )
  )
)
...
```

Figure 7. SKILL language nested loop for register file in Figure 6.