cellTK: Automated Layout for Asynchronous Circuits with Nonstandard Cells

Robert Karmazin, Carlos Tadeo Ortega Otero, Rajit Manohar
Computer Systems Laboratory, Cornell University
Ithaca, New York 14853, U.S.A.
{rob, cto3, rajit}@csl.cornell.edu

***Abstract*—Asynchronous circuits are an attractive option to overcome many challenges currently faced by chip designers, such as increased process variation. However, the lack of CAD tools to generate asynchronous circuits limits the adoption of this promising technology. In this absence of CAD tools, the most time consuming part of chip design is the back-end (physical design) effort. We propose a complete design infrastructure to physically implement an asynchronous digital netlist with orders of magnitude time savings over expert human effort. The core of this flow is the ability to generate customized logic that is compatible with available ASIC flows. We evaluate our flow against several asynchronous circuit benchmarks for which full custom physical implementations exist. Compared to hand-optimized custom designs, our flow produces layout that has, on average, a 51% area overhead, with a 12% increase in energy and a 9% increase in delay.**

***Index Terms*—Design Automation, Integrated Circuit Layout.**

I. Introduction

As process technology scales, transistor count and performance increases along with their variability and their susceptibility to environmental changes, presenting synchronous design methodologies with unique challenges in managing clock distribution and signal skew. As a result, engineers need to include larger margins into designs to ensure correct operation under increasingly uncertain fabrication or environmental conditions.

The asynchronous paradigm offers an attractive solution to this problem. Asynchronous circuits refer to the class of self-timed circuits where synchronization is achieved locally, making asynchronous circuits, especially the Quasi Delay-Insensitive (QDI) family, more robust to process and environmental variation [20]. Additionally, asynchronous circuits are modular by construction, allowing full systems to be composed of modules designed in isolation, greatly reducing design and verification times [19].

Unfortunately, the advantages of asynchronous circuits at tackling these challenges are tempered by the lack of mature and accessible CAD tools. This is especially true for physical design, where the majority of design time is spent. Existing work related to back-end flows for custom logic is described in Section II. In general, these works are classified into three broad categories:

Technology Mapping: This technique maps customized logic onto an existing library, allowing the exploitation of previously designed and characterized libraries, as well as mature ASIC flows. However, the inefficiency of the mapping often negates the advantages of having customized logic [2].

Custom Standard Cells: This method requires the creation of a library of hand-crafted cells onto which custom logic is mapped. However, implementing every possible logic function is intractable, and each library is targeted to a specific process technology, making this method neither generic nor portable [1,25,27].

Full custom layout: This approach involves manually drawing every transistor and wire in the design, typically yielding a physical implementation with the smallest area and best performance. However, it is very expensive in terms of design time and human effort, which typically restricts this approach to only the most critical components of the system.

None of the above solutions are ideal, making the implementation of asynchronous circuits a much more difficult task than it would be for comparable synchronous circuits. As a solution to this problem, we present *cellTK*, a "nonstandard" cell generator and physical design infrastructure. We coin the term nonstandard cells to refer to on-demand custom cells that fit into an automated standard cell flow. *cellTK* automates this custom cell generation, which mitigates the drawbacks of the *Custom Standard Cells* methodology while still offering the flexibility of *Full Custom* and the utility of *Technology Mapping*.

cellTK receives a transistor-level description of the circuit as an input and produces its physical implementation by generating customized logic "packaged" in the form of standard cells, which are compatible with widely-used synchronous tool flows. *cellTK* has the advantage of being able to generate custom cells on demand, allowing the design to map directly to the produced cells. This automated approach removes any loss from an inefficient mapping and frees the designer from being tied to a fixed library. In addition, *cellTK* integration with available CAD tools allows for fast physical implementation of large designs, removing the manual labor cost associated with custom logic. We show that *cellTK* is able to produce layout for various parts of a microprocessor with moderate area and performance overheads, and in some cases, improvements over a full custom implementation.

The rest of this paper discusses prior work (Section II), details the complete *cellTK* flow (Sections III - VI), evaluate the flow against a full custom implementation of several benchmarks (Section VII), and discusses the advantages and

other uses for this flow (Section VIII).

II. Related Work

cellTK aims at generating cells that will be used in an asynchronous layout flow. As such, this section discusses prior works in the fields of cell generation as well as more traditional back-end synthesis flows for asynchronous circuits.

A. Cell Generation

To reduce computational complexity, much of the initial work in this field restricted transistor placement to two rows: P-type transistors are placed in one row and N-type transistors are placed in another. This layout style is known as 1-D, since placement of transistors is allowed only in one dimension. The work by Uehara and VanCleemput [29], and later by Maziasz and Hayes [22], formulate the transistor placement problem as a graph optimization problem: minimizing the number of trail covers of the diffusion graph minimizes the number of diffusion breaks in the cell, and thus, overall cell width. However, these works are restricted to static CMOS gates, where the pull-up and pull-down networks are duals and there are an equal number of N- and P-type transistors. Later works, such as CELLERITY [14], XPRESS [13], and LiB [15], have generalized placement algorithms to handle arbitrary transistor netlists.

The placement solutions in [15,22] use deterministic algorithms to find optimal solutions, but this is only tractable on relatively small netlists. Larger cells require heuristic-based algorithms to handle the additional complexity. XPRESS uses exact algorithms and heuristics to generate the minimum graph covering using transistor trails, minimizing diffusion gaps and overall width. CELLERITY, TEMPO [24], and ASTRAN [31] use a simulated annealing framework for placement. These works use cost functions that incorporate cell width, transistor ordering and orientation, gate alignment, and routing quality. Alternatively, Gopalakrishnan [12] uses a greedy heuristic to do an intra-row refinement as the last phase of a multi-stage placement strategy. Using heuristic algorithms allows cell generators to explore beyond the traditional row-placement layout style, as in [24], and to some extent [12]. Heuristics, however, usually require a compaction stage to achieve designs comparable to hand-optimized layout.

Most of the works mentioned above use grid-based routers. For example, in [31], all transistor placement is aligned to the routing grid to ensure all transistor connections are made. A negotiation-based router typically used for FPGA routing is then used to make the connections. In [14], an area router is used, which creates a non-uniform grid after taking technology parameters and obstacles into account. Routing is accomplished by finding the minimum spanning tree on this grid, followed by a layer-assignment stage. In [15], the cell is divided into regions, where the empty regions are routed by the channel router SILK [18], and the regions over the N- and P-diffusions are routed using a maximal clique formulation of the connection graph representing the routing problem.

B. Design Automation for Asynchronous Circuits

To manage the complexity of VLSI chips, designers use hardware description languages (HDLs) which place an emphasis on high-level architecture, abstracting away many of the implementation details. However, these details must be managed effectively as the design flows to its final physical implementation.

Syntax-directed translation is the transformation of some program written in a well-defined language into an equivalent implementation using rules corresponding to the syntactic constructs of that language. In the context of asynchronous VLSI synthesis, these constructs correspond to simple CMOS circuits that implement some basic functions, such as a boolean assignment or a communication action. Burns and Martin [4] describe a system where CSP-like programs [19] are decomposed recursively until they can be translated to their corresponding circuits. Tangram [30], and later Balsa [10], use a syntax-directed approach to translate CSP to an abstract "handshaking circuit", which is then mapped to a circuit/layout library. However, the resulting circuits tend to be slower and area-inefficient, and libraries implementing these functions are relatively complex compared to a typical standard cell library.

Another approach is to leverage existing HDLs and synthesis engines to generate asynchronous circuits. For example, Pipefitter [3] takes an initial specification in Verilog and uses a commercial synthesis engine along with the asynchronous control synthesizer Petrify [8] to generate a synthesizable verilog netlist, which can then be mapped to a standard cell library. A delay line matched to the latency of the logic in each pipeline stage is added to ensure no race conditions occur. Law et al. [17] present a similar solution, except with more localized, and thus simpler, control circuitry. Weaver [25] compiles a synchronous single-rail RTL description down to a finely-pipelined QDI asynchronous implementation by replacing synchronous logic gates and registers with their QDI equivalents. A small yet sufficient hand-optimized library is implemented on to which the design is mapped. Proteus [1] represents a hybrid approach: The input, specified in CSP, is translated syntactically into an RTL representation, which is then mapped to a set of single-track logic gates using a commercial synthesis engine. These gates are then clustered and mapped onto a comprehensive library of cells following the PCHB template.

III. Nonstandard Cell Flow

We present *cellTK* as both a tool and an infrastructure to automate the physical implementation of asynchronous circuits. This section gives an overview of the entire flow, and Sections IV, V, and VI detail the creation, layout, and place-and-route stages, respectively.

The *cellTK* flow is shown in Figure 1. It begins with a hierarchical Production Rule Set (*PRS*) as an input. The PRS is automatically translated by *Netgen* (Section IV-A) into a hierarchical transistor-level netlist, which is then clustered into cells based on the nets that the transistor networks are driving (Section IV-B). To minimize the amount of work done by the

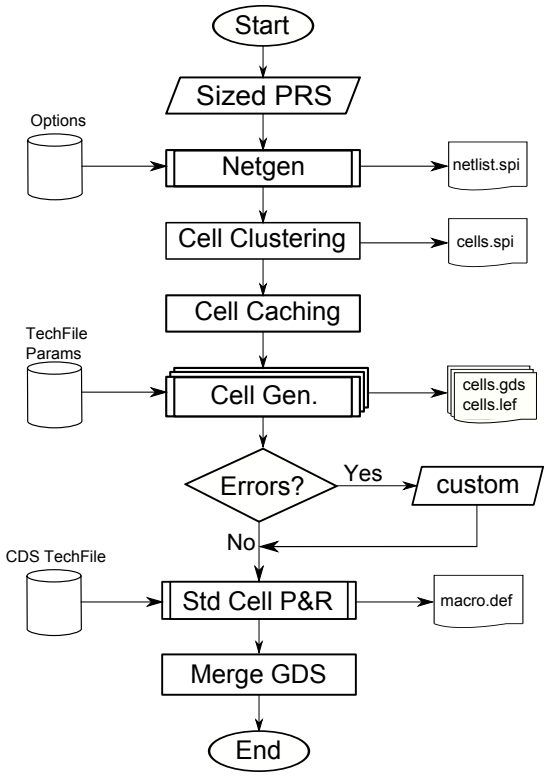


Fig. 1. The complete *cellTK* flow.

cell generator, a cell cache is maintained such that only unique or not-yet-generated cells continue through the rest of the flow.

Once the cells are created, a physical implementation for each unique cell is generated, using a two-step transistor placement and local routing phase (Section V). *cellTK* reads a technology file containing process-specific rules for producing legal (DRC clean) cells. Since each cell can be generated independently, this step is easily parallelized. *cellTK* employs a server/client model: the client sends requests out to multiple layout servers, each running multiple cell generating threads. Once completed, the client collects all the cells.

After all the cells are created, they are exported to a standard cell placement and global routing tool (Section VI). The final layout is generated by merging individual cells' layout with the geometry drawn during the cell placement and routing phase, creating a GDS file ready for fabrication.

Unlike the solutions presented in Section II-B, the *cellTK* flow does not require a library onto which circuits are mapped. Rather, the library cells are generated on demand, yielding a one-to-one mapping between circuit description and physical design. Therefore, there is no need to have a preexisting asynchronous cell library, nor a need to instrument synchronous netlists with delay lines or flop controllers, giving the user control over the generated layout not possible with previous solutions.

IV. Cell Synthesis

The synthesis stage can be split into two steps: a translation step to create a transistor netlist from the provided PRS, and a clustering step, which creates netlists for the individual cells.

A. Netgen: Netlist Generator

Netgen syntactically translates every Production Rule (*PR*) into a transistor-level SPICE netlist. A single PR takes the following form: $G \mapsto S$, where G is a boolean expression called the guard, and S is a boolean assignment. The PR corresponds to a pull-up or pull-down switching network, depending on whether the boolean assignment S was true or false, respectively. The ordering of the resulting transistor networks is deterministically derived from the PRS. The power rails are assumed to be connected to the source terminal of the transistor generated from the left-most literal in the PR guard, and the output net (referred to as a *node*) is connected to the drain terminal of the transistor derived from the right-most literal of the guard. For example, the PR

```
a & b -> c-
```

is translated by *Netgen* into a network starting from GND (since node `c` is being pulled down), connected to the source of a transistor gated by a followed by a transistor gated by b, whose drain is connected to the output node c. The SPICE netlist corresponding to this PR is given by:

```
M0_ GND a #3 GND nfet W=0.3U L=0.12U
M1_ #3 b c GND nfet W=0.3U L=0.12U
```

A PRS can express not only static CMOS gates, but also arbitrary transistor netlists from various circuit families, which are a common result of Martin's synthesis procedure [19]. To ensure electrically sound circuits are synthesized, *Netgen* can automatically staticize dynamic nodes by adding state-holding feedback transistors. These staticizers take the form of either ratioed keepers or non-ratioed transistor networks (implementing combinational feedback). This feature frees the user from manually implementing staticizers, leading to a cleaner, less cluttered, and easier to maintain PRS.

Netgen supports PRS annotated with attributes, allowing the user even tighter control over the synthesized transistor netlist. Common attributes include transistor widths and lengths, transistor types (high-V_t or low-V_t), and staticizing options (using keepers or feedback transistors) for dynamic nodes.

B. Cell Clustering

Once the netlist is generated, transistors are clustered into *cells*. Transistors are partitioned based on the node which they are driving, similar to [13,15]. *cellTK* recursively walks the pull-up and pull-down transistor networks driving a particular node until the power rails are reached. All the P- and N-type transistors discovered on this walk are then grouped in the same cell. Any output inverters (defined as inverters that are driven by this particular node) are also grouped in the same cell to reduce that node's capacitance. If a dynamic node is staticized with a ratioed keeper, then the keeper transistors are also grouped in the same cell. However, if the node is staticized with a non-ratioed feedback network, those feedback transistors are grouped into their own cell to keep cell complexity manageable.

A cell cache is maintained to minimize the amount of work required of the cell generator. If a cell is found in this cache, it does not need to be regenerated. However, for the cache to be effective, a lookup must take less time than simply regenerating the cell. Thus, performing a graph isomorphism test for every new cell against every cell already existing in the cache must be performed quickly. However, the graph isomorphism problem has been shown to be NP-complete [11], and is not a reasonable solution for large designs with many cells. Instead, we use a heuristic method to determine cell equivalence. Our heuristic compares the number of I/O ports, the number of transistors, and the number of nets in the cells. If these simple checks are true, then the VF2 heuristic [7] is used to determine the existence of structural isomorphisms in the cells' connectivity graphs. If these exist with respect to the I/O ports, we then assume a cell match. Note that this heuristic never yields false positives, and the number of false negatives is, in practice, negligibly small.

It is important to note that the cells generated using the above procedure would not be found in static standard cell libraries [5]. The majority of asynchronous gates specified in a PRS are dynamic, requiring some form of feedback to ensure robust operation. Additionally, there are no restrictions on transistor sizing at the PRS level. Indeed, some ratioed feedback devices might have longer channel lengths (see Figures 4 and 5), a feature that most of the works described in II-A cannot properly handle. Though it is possible to emulate the cells' behavior using static standard cells, this would incur a significant area and latency overhead compared to a direct implementation. Furthermore, this method would have to ensure glitch freedom, an essential property of QDI circuits. It is for these reasons that *cellTK* generates "nonstandard" cells on the fly rather than mapping onto a static cells found in synchronous libraries.

V. Custom Cell Layout

In this section, we describe the method used to generate layout for nonstandard cells. The techniques for transistor placement and local routing described below are designed to be as generic as possible so that the *cellTK* flow can handle all potential cell netlists. In the following discussions of transistor placement and routing, a description of the overall method is given, followed by its implementation details.

A. Transistor Placement

The goal of transistor placement is to place all the transistors in as small an area as possible. *cellTK* places transistors in rows, similar to [14,15,22]. In this 1-D layout style, minimizing cell area is equivalent to minimizing the number of diffusion breaks, or maximizing the sharing of source-drain terminals of logically-connected transistors. *cellTK* uses a graph formulation of this problem similar to [24]: The transistor netlist is converted into a *diffusion graph* where nets are vertices and transistors are edges. Using this formulation, finding the solution minimizing diffusion breaks is equivalent to finding the fewest Euler paths (also referred to as *trails* or *chains* in the literature) that cover the entire graph. *cellTK* then uses these trails as the atomic unit of placement [12].

cellTK takes a novel approach to searching for these optimal trails: the search is guided by a *gate-ordering*, a predetermined order of transistors' gates. *cellTK* performs a recursive depth-first search of the diffusion graph, finding a *transistor ordering* corresponding to the given gate-ordering. A valid match is found when the gate terminals of the resulting transistor ordering match those in the provided gate-ordering. *cellTK* automatically orients these transistors such that diffusion sharing is maximized. This approach has several advantages. It prunes the search space as the algorithm does not have to explore all possible paths through the diffusion graph. It also allows for easy user intervention in the case of complex or critical cells, such as memory cells where symmetry is crucial. For cells with complementary pull-up and pull-down transistor networks, the same gate-ordering can be applied to both networks; non-complementary transistor networks require different gate-orderings. Note that it is possible for a given cell to have multiple transistor orderings for a single gate-ordering. *cellTK* searches all potential transistor orderings and chooses the one minimizing the number of diffusion breaks. The current implementation of *cellTK* searches all possible combinations of gate orders, and for each unique gate order, searches all possible permutations and orientations of transistors, to try and find the transistor ordering minimizing the number of graph covers.

A chain assigned physical dimensions is referred to as a *stack*. Unlike previous works, a stack's size is not solely dependent on its transistors; stacks grow or shrink based on the dimensions and orientations of their transistors and the rules of the given process technology. Stacks are stored symbolically as linked lists, with each bucket in the list corresponding to either a transistor's source/drain terminal, or a transistor's gate terminal. These buckets also store physical information, such as transistor widths and lengths, as well as the spacing to adjacent transistors and diffusion contacts. This allows *cellTK* to optimize the stack dimensions by making local adjustments to distances between transistors and contacts while still ensuring all design rule constraints are met. Figure 2 shows an example of the transistor network from the PRS in Section IV, its resulting stack and its representation in memory.

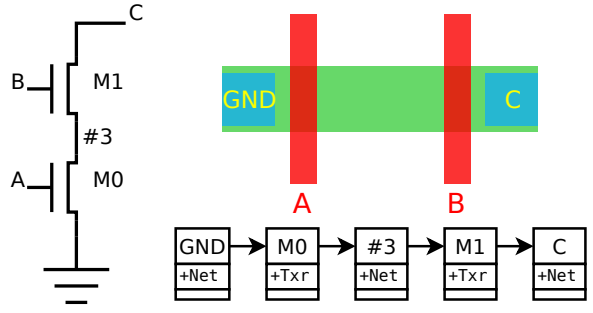


Fig. 2. The transistor netlist corresponding to the *PR* in Section IV-A (left), the corresponding stack (top-right), and the symbolic representation of that stack in memory (bottom-right).

As mentioned above, the atomic unit of placement is the stack, not an individual transistor. The first transistor in the found transistor ordering is the left-most transistor in its appropriate row. Actual placement is done by assigning coordinates to the stacks. Stacks are offset in their respective rows in such a way that gates common to both pull-up and pull-down networks align vertically, simplifying the routing problem.

B. Transistor Router

Once the cell's stacks are placed, its nets can be routed. The overall routing strategy of *cellTK* is to complete intra-cell routes first, using as few routing resources as possible, and then complete routes to the cell's I/O ports with higher layers. This strategy is broken down into four steps:

1) Route all the aligned gates. Given the placement strategy described previously, this can be trivially done with only vertical polysilicon.
2) Route all the intra-cell nets. Priority is given to the cell's output node.
3) Draw power rails horizontally across the top and bottom of the cell, aligned to the routing grid. Route the power and ground nets first within the stacks, and then to the drawn rails.
4) Draw "pins" for the cell's I/O ports on a midlevel routing layer, which are used as points of contact for the commercial router.

In each successive step, the router is allowed to use more routing resources. For example, in step 1, the router is only allowed to use polysilicon, but in step 3, the first and second metal layers are also allowed. The costs of routing in each layer are user-adjustable such that higher level metal layers are used as sparingly as needed. Figure 3 illustrates a cell before and after it has been routed.

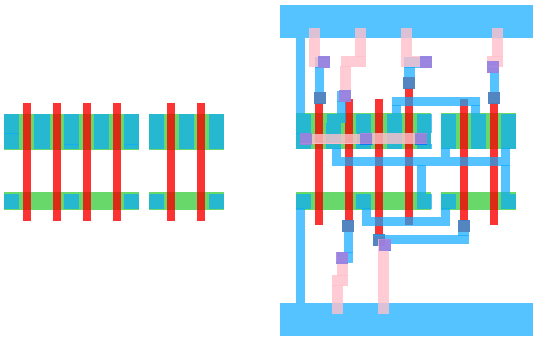


Fig. 3. Sample cell with placed stacks. The left cell has not yet been routed except for the aligned gates in polysilicon. The right cell is completely routed.

The transistor placement strategy described above requires the router to be capable of very detailed routing, since there is no guarantee that net terminals will be on any grid or obey any pitch. To account for this inherent irregularity, *cellTK* incorporates a router based on Contour [9], a gridless, tile-based router built on top of the corner-stitch data structure [23].

The routing algorithm itself is based on A* [6]. Given two unrouted terminals on the same net, potential solutions (referred to as *paths*) are propagated outward into all adjacent tiles free of any material (known as *space* tiles). These new paths are then propagated to the next set of adjacent space tiles, and so on. The search space is pruned by not propagating paths whose resulting cost is higher than an existing path's cost. A solution is found when paths from the two terminals intersect in a single space tile. If no solution is found, then previously completed routes are ripped up and the search is repeated until a solution is found, and the ripped nets are rerouted. This router is fundamentally different from the cell routers discussed in Section II because it does not operate on a grid, allowing it to make fine-grained connections to stack terminals. Previous routers required the stack terminals to be on the routing grid, potentially bloating the total area.

The router integrated into *cellTK* is a modified version of [16], where the basic tile is extended with pointers to tiles in adjacent layers to speed up interlayer path propagation. The tile is also extended with a unique solution tag, allowing the router to rip up portions of a net rather than the entire net. We further extend this router in the following ways:

Minimum Area: Modern processes require materials in a layer to satisfy some minimum area constraint. To enforce this, paths are extended to include a running total of the amount of material in the current layer. Paths are not allowed to propagate to layers above or below until this amount meets the minimum value set in the technology file. Once a path is propagated to a new layer, this running total is reset.

Exclusive Routing Directions: Future process nodes may require that some materials be drawn in a single direction. The router is extended with controls to restrict planar path propagation such that a routing solution has no bends or jogs in a given layer.

Short-Vertex: A short vertex is defined as a vertex connected to two edges whose lengths do not meet some minimum value. These jagged edges may be flagged as DRC errors. A post-processing step is added to the router to fix these errors by adding material to smooth out the jagged corner without creating any new spacing violations.

Out-of-Order Rip-up and Reroute: Unlike the original implementation which only allowed rip-ups in the reverse order of the route's completion, the router can rip up portions of nets out-of-order. The rip-up mechanism has been modified to consider spacial and temporal locality; the net chosen for rip-up is the most recently completed net closest to the net that most recently failed.

There may be instances where *cellTK* is presented with a difficult routing problem, causing it to timeout. In such instances, *cellTK* uses a checkpoint and replay method that allows it to generate partially completed cells should an error occur, reducing the amount of manual effort required by the user to complete the route.

Examples of 2-input operators automatically generated by *cellTK* are shown in Figure 4. Transistor gates common to complementary chains are aligned and routed, as are the other nets, pins, and power rails. Figure 5 shows the more complicated WCHB. As can be seen, both the C-element and the Nand-gate have their output inverters grouped into their

respective cells. The C-element cell also contains its ratioed staticizer.

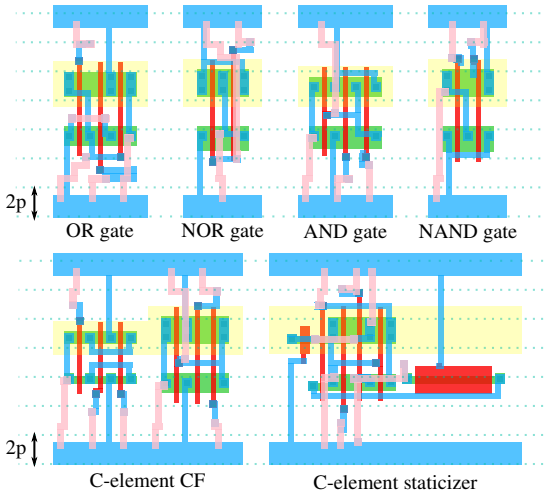
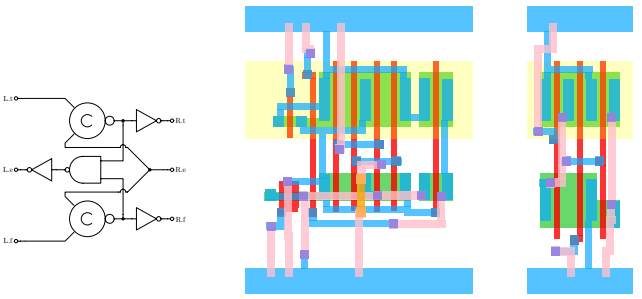


Fig. 4. A cross-section of common 2-input operators found in both synchronous and asynchronous designs. p = wire pitch



(a) Schematic of a WCHB

(b) Layout of a C-element with inverter

(c) Layout of a Nand-gate with inverter

Fig. 5. Schematic and layout of WCHB cells. The C-element has staticizers and reset logic (not shown in the schematic).

VI. Cell Place and Route

Once all the cells are generated, they are placed and routed using a readily-available commercial tool. However, before the cells can be placed, two additional cells need to be generated: a filler cell and a welltap cell. Welltaps are placed in a repeating pattern throughout the layout, and fillers are used to ensure continuity in the well regions. *cellTK* can generate these cells automatically based on the cell height and the well positions computed from all the generated cells.

In addition to having filler and welltap cells, the logic cells must meet the following requirements to ensure a DRC and LVS clean final layout:

Wells: All cells must have uniformly-positioned wells. Otherwise, wells in adjacent cells may create notches, causing DRC errors. Once all the cells are generated, they are iterated over to find the dimensions of the wells satisfying the DRC rules of all cells. Wells are then drawn in each cell.

Cell Size: All cell heights and widths must be a multiple of the global routing pitch so cells can be placed on the grid used by the global router (unlike the gridless contour router). Placing cells on the global router's grid makes it easier to route inter-cell nets, as well as to insert fillers and welltaps.

Cell Pins: All the cells' pins must be drawn on the global router pitch. Then, when the cells are placed on the routing grid, all their pins will also be on the global router's grid, connecting the intra-cell wiring to the inter-cell routes created by the global router.

cellTK ensures that all generated cells meet these minimum requirements. Once all the cells are generated, including the fillers and welltaps, they can be fed into the standard cell placer and router. *cellTK* can export Library Exchange Format (LEF) files, so cells can be imported by any placer that supports this file format. *cellTK* also exports a Verilog netlist that captures inter-cell connectivity. Once the cell placement and routing phase is complete, *cellTK* merges the resulting layout with the layout of the individual cells, producing the final physical implementation. This layout can be compared against the initial pre-clustered transistor netlist using commercially available layout versus schematic (LVS) tools, a unique feature of this flow.

VII. Evaluation

cellTK is capable of generating customized cells that would not be found in typical synchronous standard cell libraries, and would thus require manual effort to implement. Furthermore, licensing restrictions preclude comparative analysis of commercial standard cell libraries. It is for these reasons that *cellTK* is evaluated against several transistor-level netlists for which a best effort full custom physical layout is available. The majority of these modules were originally part of an asynchronous processor designed in a 90nm process technology. The evaluated modules are from the main datapath (the logic and shift units) and the instruction front-end (the fetch and decode units). These modules were chosen because they encompass various layout styles, from a structured datapath to a "sea of gates." The logic, shift units were compiled using precharge half buffer reshuffling, similar to the ones found in MiniMIPS. The fetch and decode modules were decomposed using similar techniques as the ones used in the CAM microprocessor [21]. All modules in Table I interface with other components using an four-phase handshake protocol and all values are transmitted using a delay insensitive encoding.

The metrics used for evaluating the quality of the resulting layout are area, throughput (cycle time), and energy (per cycle). Area is calculated by looking at the bounding box of the final (placed and routed) design, accounting for the penalty of having some unused space resulting from sub-optimal placement. Energy and cycle times are computed using a transistor-level simulation with resistor and capacitor parasitics extracted from the final layout. Such detailed simulations accurately reflect the impact of cell quality and routing quality on the energy and latency of the generated module.

The results of this evaluation are presented in Table I. Total transistor count, average number of transistors per unique cell, and total number of cells are presented. Area is expressed as a percentage overhead over the full custom implementation of

TABLE I
EVALUATION OF *cellTK* NONSTANDARD CELL LAYOUT

	Txr Count	Avg Txr/Cell	Total Cells	Cell Height	Wire Cap. μ	Wire Cap. M	Area Overhead	Energy Overhead	Delay Overhead	Manual cells
Logic	736	10.2	11	17	2.86	1.69	76%	2%	2%	1,1
				14	2.41	1.63	60%	1%	1%	3,3
Barrel Shift	8900	7.42	51	18	1.08	0.98	28%	−2%	5%	0,0
				17	1.10	0.94	14%	3%	10%	1,1
				15	1.12	0.95	6%	9%	15%	7,5
Decode	2698	10.2	56	20	1.87	1.58	92%	26%	25%	1,1
				17	1.87	1.61	61%	33%	50%	3,0
				15	2.27	1.68	45%	34%	37%	18,8
Fetch	3517	7.67	99	20	0.88	0.71	166%	−3%	−17%	1,1
				17	0.83	0.69	120%	−3%	−17%	6,5
				15	0.86	0.73	93%	−3%	−16%	16,12
Average (shortest height)	—	—	—	—	1.66	—	51%	11.75%	9.25%	15%

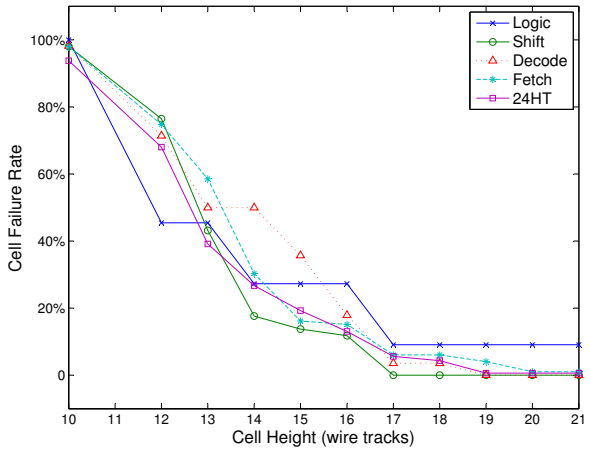


Fig. 6. Rate of failure as cell heights shrink.

Fig. 7. To-scale image of the layout of the decode module. With nonstandard cells (left), best-effort full custom (right). A cell height of 20 tracks was used.

the module. The metrics μ and M represent the mean and median of the wire capacitance ratios of all nets in the design, respectively. μ is computed by Eq 1, where C is the extracted wiring capacitance.

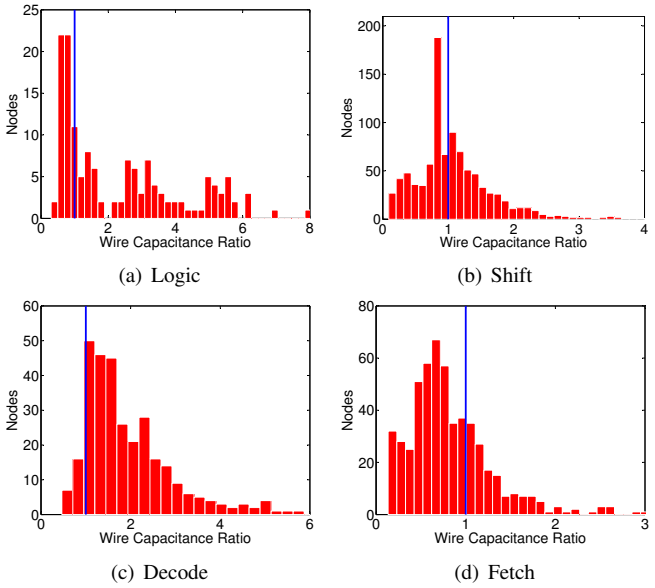
$$\mu = \frac{1}{N} \times \sum_{i \in nodes}^{N} \frac{C_{i,celltk}}{C_{i,full\ custom}} \quad (1)$$

Energy and delay are also expressed as a percentage overhead over the full custom layout. The last metric, the number of cells requiring manual effort, is represented as a tuple (x, y), where x is the number of failing cells fixed by transistor folding, and y is the number of failing cells for some other reason (usually routing failures).

For each of the evaluated modules, the layout was generated with multiple cell heights. Figure 6 shows that taller cells have fewer failures than shorter cells. This is because as cell heights shrink, there is less room for intra-cell routing. However, at a cell height of 15 routing tracks, there is a sharp inflection point in the failure rate. At this height, cell failures are caused by the transistors, not just by routing errors. For this reason, we did not evaluate *cellTK* for cell heights less than 15 routing tracks, except for the logic core. Generally, the magnitudes and trends of the overheads incurred by using *cellTK* over full custom layout are dependent on the type of module. For example, the area penalty for the logic block is larger compared to the shifter, a result of the shifter's more complicated connections: *cellTK* is better able to discover difficult routing solutions, resulting in many more wires routed over cells, whereas the custom layout requires reserved space for the wiring. This also explains the small μ for the barrel shifter. Alternatively, control-heavy modules like the decode, which fall under the "sea-of-gates" category, have larger area, energy, and delay overheads when laid out using *cellTK*. The cluttered nature of the logical connections presents a difficult problem for the standard cell placer. Careful planning by designers yields a near optimal placement and routing of the cells, which is difficult to achieve with automated tools. An example of this disparity can be seen in Figure 7. As expected, the area overhead of the generated layout for all the benchmarks decreases with smaller cell heights. An unexpected result is that energy and delay get worse as cell heights shrink. This is because for smaller cell heights, increased cell density presents a more difficult problem to the global router, generating worse routes. We verify this by examining mean net capacitances, which increase with decreased cell heights.

Another interesting result is that the fetch module generated by *cellTK* improves the energy and delay over the full custom implementation, despite the large area overhead, a consequence of the module architecture as well as a human's inclination for modular design. The fetch module contains an

adder and a register, which in the custom implementation, are placed side-by-side, whereas *cellTK* can interleave cells from these sub-modules, reducing average wire length, and consequently, energy and delay. This intuition is confirmed by Figure 8, a histogram of wire capacitance ratios for all nets. For the fetch module, the majority of nets have capacitance ratios less than 1, indicating that *cellTK* produces a layout where most nets are less capacitive than those in the full custom implementation. In contrast, the majority of nets in the decode module have capacitance ratios greater than 1, incurring significant energy and delay overheads.



(a) Logic (b) Shift

(c) Decode (d) Fetch

Fig. 8. Histogram of wire capacitance ratios used to compute μ.

Table II shows the improvement in design time using *cellTK* compared to custom implementation by experienced designers. In particular, the barrel shift and decode units were laid out by professional layout engineers with 6+ years of experience. Due to the inaccuracies of quantifying human labor, Table II contains only approximations for the custom design time. Nevertheless, the orders of magnitude savings in design time is still apparent.

TABLE II
DESIGN TIME OF DIFFERENT MODULES

	Full custom Design Time (months)	*celltk* Design Time	
		Computer (minutes)	Human (minutes)
Logic	▬	2	40
Barrel Shift	▬▬▬▬▬	10	60
Decode	▬▬▬▬	4	480
Fetch	▬▬	13	1200

As proof of the viability, versatility, and usefulness of *cellTK*, two chips have been designed, implemented, and sent for fabrication using this flow. The first is a 6-channel baseband GPS processor [26], and the second is an asynchronous FPGA based on [28]. A summary of the results are presented in Table III. This demonstrates that *cellTK* can handle complex designs with relatively little manual effort as well as easily port to multiple technology nodes from different foundries. The GPS is particularly noteworthy because it was designed using both QDI and bundled-data families of asynchronous circuits, both of which are compatible with *cellTK*. Additionally, *cellTK* is capable of generating cells for both 65nm and 45nm technology nodes.

VIII. EXTENSIBILITY OF *cellTK*

The *cellTK* flow described in the previous sections is designed to generate nonstandard cells to automate layout for circuit families for which standard cell libraries do not exist. It is evaluated on asynchronous circuits, but it can be applied to a much broader range of circuit families, such as domino logic in the synchronous domain. This section describes how the existing *cellTK* flow can be extended for various applications outside the realm of nonstandard cell generation.

A. Software Modularity

All the methods employed by *cellTK* to generate cells are built on top of low-level functions that manage transistors and stacks, as well as control the contour router. These functions are grouped together in a library referred to as *LayoutTK*, which exposes an API used by *cellTK*. Note that the interface to *LayoutTK* is generic; it has no features that make it inherently geared towards the work presented in this paper.

This modular software organization is very powerful in that it allows *LayoutTK* to be used for a number of different physical design applications, with cell generation being just one of them. Indeed, datapath generators or memory compilers can all be built on top of *LayoutTK*, given the expressiveness and versatility of the API.

B. Rapid Feedback

One of the advantages of having this level of automation is the ability to easily explore various design tradeoffs that would otherwise be impossible due to the high cost of implementing physical layout. *cellTK* allows the user to collect statistics about the circuit quickly, which can aid in making informed decisions at all design levels, from the architecture down to the layout. *cellTK* provides the following tools and statistics:

1) Height Finder: Cell height is one of the most important factors affecting the quality of the final layout because, in the current implementation, all cells must have the same height. *cellTK* offers a search feature to find the smallest height for which cells can be successfully generated. A failure tolerance value is provided, which sets the number of cells that a user

TABLE III
FULL CHIP DESIGNS FINISHED WITH LAYOUTTK

	GPS	FPGA
Technology Node.	90nm	130nm
Total Txr. Count	3.20M	1.33M
Unique Cells (manual)	1404 (55)	180 (24)
Cell Height (tracks)	22	15
Chip Dimensions	4mm x 5mm	3mm x 3mm

is willing to manually implement. For example, if it is critical to have a small row height, the user may choose to minimize cell height at the expense of accepting a higher failure rate.

2) Diffussion Density and Area Estimation: Before actually placing and routing a macroblock, *cellTK* is able to provide estimates for the diffusion density as well as overall area by sampling the cells that were just produced. These statistics are usually available within a couple of minutes of initializing *cellTK*, even for million-transistor designs.

3) Comformity Checker: Should *cellTK* fail on certain cells, statistics are reported back to the user regarding potential causes for this failure. Specifically, *cellTK* reports which transistors have the largest dimensions, and the cells of which they are a part, informing the user about where folding should take place as a potential fix for the failing cell.

C. Fast Prototyping

cellTK is written to generate cells in the traditional 1-D layout style. Given its modular nature, it is simple to modify *cellTK* to generate cells in a different style. For example, a user might want to explore the tradeoffs of placing transistors of the same type in multiple rows, requiring an increase in cell height, but decreasing the cell width, a worthwhile tradeoff if the design already contains many wide transistors that would otherwise require folding. Another example is having the power rails routed on different materials. Traditionally, cells' power and ground rails are routed on the first metal layer above and below the active regions, respectively. These power rails can alternatively be routed on higher metals over the active region, effectively compressing cell height, allowing rows of cells to be grouped tighter, and increasing overall diffusion density.

IX. Conclusion

This paper presents *cellTK*, an automated nonstandard cell generator and complete design infrastructure to physically implement an asynchronous digital netlist. *cellTK* can create layout with an average 51% area overhead and 12% and 9% overhead in energy and delay, respectively, compared to hand-optimized custom implementations, at a fraction of the time. The core of this flow is a generic and versatile layout library which can physically implement an arbitrary transistor netlist, independent of logic family and design paradigm. With this freedom from predetermined libraries and the great reduction in layout design time, *cellTK* enables the widespread adoption of asynchronous circuits as a solution to future technology challenges.

Acknowledgment

This work has been supported in part by DARPA award HR0011-09-C-0002, AFRL award FA8750-09-2-0010, IARPA award N66001-12-C-2009, NSF award CCF-1065307.

References

[1] P.A. Beerel, G.D. Dimou, and A.M. Lines. Proteus: An asic flow for ghz asynchronous designs. *IEEE Des. Test*, 28(5), 2011.

[2] P.A. Beerel, R.O. Ozdag, and M. Ferretti. *A designer's guide to asynchronous VLSI*. Cambridge University Press, 2010.

[3] I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral verilog hdl. In *ASYNC*. IEEE, 2000.

[4] S.M. Burns and A.J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. Technical report, DTIC Document, 1988.

[5] D. G. Chinnery and K. Keutzer. Closing the gap between asic and custom: An asic perspective. In *DAC*. IEEE Computer Society, 2000.

[6] G.W. Clow. A global routing algorithm for general cells. In *DAC*. IEEE, 1984.

[7] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10), 2004.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *Transactions on Information and Systems*, 80(3), 1997.

[9] J. Dion and L.M. Monier. *Contour: A tile-based gridless router*. Digital, Western Research Laboratory, 1995.

[10] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1), 2002.

[11] M.R. Gary and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, 1979.

[12] P. Gopalakrishnan and R.A. Rutenbar. Direct transistor-level layout for digital blocks. In *ICCAD*. IEEE Press, 2001.

[13] A. Gupta, J.P. Hayes, et al. Xpress: a cell layout generator with integrated transistor folding. In *DATE*. IEEE, 1996.

[14] M. Guruswamy, R.L. Maziasz, D. Dulitz, S. Raman, V. Chiluvuri, A. Fernandez, and L.G. Jones. Cellerity: a fully automatic layout synthesis system for standard cell libraries. In *DAC*. ACM, 1997.

[15] Y.C. Hsich, C.Y. Hwang, Y.L. Lin, and Y.C. Hsu. LiB: A CMOS cell compiler. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 10(8), 1991.

[16] C.C. LaFrieda. Custom-quality wire routing using modern design rules. Master's thesis, Cornell University, 2005.

[17] C.F. Law, B.H. Gwee, and J.S. Chang. Modeling and synthesis of asynchronous pipelines. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 19(4), 2011.

[18] Y.L. Lin, Y.C. Hsu, and F.S. Tsai. SILK: a simulated evolution router. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 8(10), 1989.

[19] A.J. Martin. Compiling communicating processes into delay-insensitive vlsi circuits. *Distributed computing*, 1(4), 1986.

[20] A.J. Martin. Asynchronous logic for high variability nano-CMOS. In *ICECS*. IEEE, 2009.

[21] A.J. Martin, M. Nystrom, and C.G. Wong. Three generations of asynchronous microprocessors. *Design Test of Computers, IEEE*, 2003.

[22] R.L. Maziasz and J.P. Hayes. Layout optimization of static cmos functional cells. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 9(7), 1990.

[23] J.K. Ousterhout. Corner stitching: a data-structuring technique for VLSI layout tools. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 3(1), 1984.

[24] M.A. Riepe and K.A. Sakallah. Transistor placement for noncomplementary digital vlsi cell synthesis. *TODAES*, 8(1), 2003.

[25] A. Smirnov, A. Taubin, Ming Su, and M. Karpovsky. An automated fine-grain pipelining using domino style asynchronous library. In *ACSD*, 2005.

[26] B.Z. Tang, S. Longfield, S.A. Bhave, and R. Manohar. A low power asynchronous gps baseband processor. In *ASYNC*. IEEE, 2012.

[27] A. Taubin, J. Cortadella, and L. Lavagno. *Design automation of real-life asynchronous devices and systems*. Now Publishers Inc, 2007.

[28] J. Teifel and R. Manohar. An asynchronous dataflow FPGA architecture. *IEEE Trans. Comput.*, 53(11), 2004.

[29] T. Uehara and W.M. VanCleemput. Optimal layout of cmos functional arrays. *IEEE Trans. Comput.*, 100(5), 1981.

[30] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The vlsi-programming language tangram and its translation into handshake circuits. In *EDAC*. IEEE, 1991.

[31] A. Ziesemer and C. Lazzar. Transistor level automatic layout generator for non-complementary cmos cells. In *VLSI-SoC*. IEEE, 2007.