

FINE-GRAINED SUPPLY GATING
FOR LOW DYNAMIC AND LEAKAGE POWER
USING HYPERGRAPH PARTITIONING
AND SHANNON DECOMPOSITION

by

LAWRENCE LEINWEBER

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Thesis Adviser: Dr. Swarup Bhunia

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2007

Table of Contents

Abstract.....	4
Introduction.....	5
Shannon Expansion.....	6
Literature Survey	9
Shannonization.....	11
Synthesis Flows	19
Circuit Representation	21
Partitioning Algorithm	24
Cost Functions	29
Annealing Algorithm	32
PMOS Supply Gating	38
Test Output Files.....	41
Test Conditions	42
Test Circuits	43
Test Procedure	44
Selection of Tunable Parameters	47
Test Results.....	49
Future Work.....	56
Conclusion	57
Bibliography	58

List of Tables

Table 1: Descriptions of Benchmark Circuits.....	43
Table 2: Comparison of Partition Sizes	50
Table 3: Optimal Partition Sizes	51
Table 4: Recursive Shannonization	52
Table 5: Relaxed Area Constraint.....	53
Table 6: Comparison of Technology Scales	54
Table 7: Comparison of Cycle Times	55
Table 8: Execution Times	56

List of Figures

Figure 1: General Shannonized Function	7
Figure 2: General Shannonized Function with Shared Logic.....	8
Figure 3: Logic Diagram of Equality Function.....	12
Figure 4: Logic Diagram of Technology Mapped Equality Function	13
Figure 5: Logic Diagram of Cofactored Equality Function.....	15
Figure 6: Logic Diagram of Equality Function with Shared Logic.....	16
Figure 7: Logic Diagram of Shannonized Equality Function.....	18
Figure 8: PMOS Gating Circuit	40

Fine-Grained Supply Gating
For Low Dynamic And Leakage Power
Using Hypergraph Partitioning
And Shannon Decomposition

Abstract

by

LAWRENCE LEINWEBER

Energy requirements and power density issues have a significant impact on microelectronic systems in the form of battery life and bulky cooling fans. These problems continue to grow as leakage power becomes the predominant form of power consumption below 45 nm. A design tool was developed to synthesize combinatorial logic with reduced power consumption using hypergraph partitioning and Shannon decomposition, which reduces leakage power by disabling unused logic in small clusters of gates. The program demonstrates that these techniques can be fully automated. The techniques are described and the results of tests with ISCAS-85 benchmarks are presented. The power savings median value was an improvement of 18% with predictive 32 nm technology.

Introduction

Technological advances in semiconductor fabrication have led to smaller transistors and higher clock speeds; however, power requirements have not scaled down at the same rate. As a result, power density has increased. Energy requirements and power density issues have a significant impact on microelectronic systems as energy enters, remains in, and exits these systems. In many systems, energy enters from a battery whose life limits the effectiveness and even the viability of mobile applications. Energy remains in systems in the form of temperature, which constrains the performance and reliability of electronic devices. And energy exits many systems via a heat sink or fan, which increases the cost and bulk of the system.

Power is lost in the normal operation of digital systems. Because they are designed to process information rather than perform physical work, power is only consumed because the system is realized from sub-optimum devices. But the MOSFET integrated circuits used in the modern systems come close to optimal. As the technology is pushed further, though, minor imperfections have become serious limitations.

The most efficient logic systems are static CMOS, and the limited use of pass transistors. Power is lost in two ways, dynamically, when the circuit is switched, and statically, when the circuit is idle. Dynamic power is consumed in short circuit power, when static CMOS has a path to ground during switching, and capacitive power, lost reversing the charges on capacitive loads in an FET circuit. Static power is the leakage current from the power supply through the transistors to ground. Leakage current includes sub-

threshold leakage, due to the lowering of threshold voltages, and the comparatively minor effects of band-to-band tunneling and gate leakage.

Until recently, dynamic power had been the only significant form of power consumption; however, success in controlling dynamic power has exposed the secondary problem of static power, especially sub-threshold leakage, and some of the success with dynamic power has been at the expense of static power. Dynamic power scales as the square of voltage, so supply voltages have been scaled aggressively. Lower supply voltages require lower switching threshold voltages, which leads to greater sub-threshold leakage. Higher operating temperatures also aggravate sub-threshold leakage. In the 45 nm regime, leakage power has caught up with dynamic power as the greatest form of power consumption.

One of the most fruitful ideas among the many techniques for saving power is reducing the power of unused sections of large systems. Power is saved by raising the threshold voltage or switching off the power supply. Transistors are the obvious choice for switching. On the small scale, transistors within gates have lower leakage when in series. Stacked transistors reverse bias the gate-source voltage. On the large scale, an entire ALU or the like can be put into sleep mode by wide supply gating transistors.

Shannon Expansion

This technique on the scale of a few to one hundred gates is based on the Shannon expansion, in which a Boolean expression is factored into two terms, one in which a

control variable is factored as false and another in which the control variable is factored as true:

$$f(x) = f(0) \cdot \bar{x} + f(1) \cdot x$$

The logic for $f(0)$ is stacked with a supply-gating transistor controlled by \bar{x} and the logic for $f(1)$ is stacked with a transistor controlled by x . The outputs of the two are combined in a multiplexer controlled by \bar{x} and x , as illustrated below:

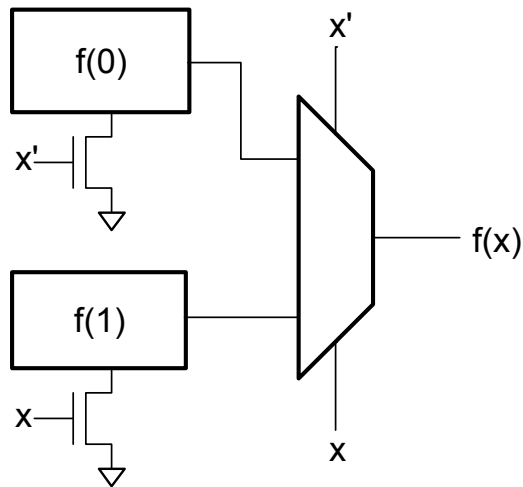


Figure 1: General Shannonized Function

This Shannon decomposition technique requires that the circuit be factored into two subexpressions, or cofactors. This is always possible but if not done carefully can lead to some redundant logic. To prevent this, consider the following equation:

$$g(x, h(x)) = g(0, h(x)) \cdot \bar{x} + g(1, h(x)) \cdot x$$

Here g is factored with respect to x . The functional value $h(x)$ is clearly an input to both cofactors of g . The shared logic that produces any signal that would appear in both cofactors should be implemented once independent of the cofactors and the resultant signal used as an input to each cofactor. It is important to identify such signals when developing the cofactors in order to avoid implementing the same logic twice, which wastes area and consumes some leakage power in the disabled, redundant copy.

The full implementation is illustrated below:

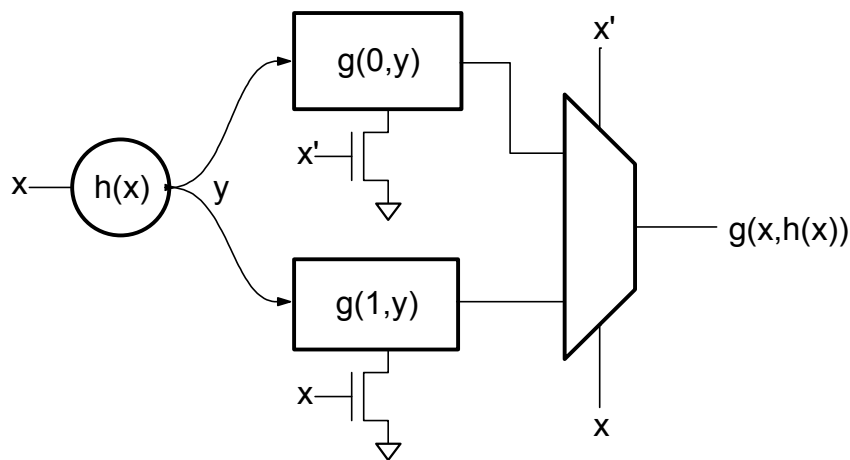


Figure 2: General Shannonized Function with Shared Logic

Factoring out common logic subexpressions at the input of a system to which Shannon decomposition is applied is a refinement that broadens the range of circuits to which the technique can be applied effectively. It is also useful to factor out post-multiplexer logic, but that was not done in the implementation of the program to synthesize Shannonized circuitry.

It is not possible to factor out post-multiplexer logic or even to separate cofactors by simply expressing the output signals in terms of the input signals in sum-of-products form unless the number of inputs is small because each additional input can double the number of minterms required to express the output. The worst case is 2^{n-1} minterms for n inputs, which occurs in an n input exclusive-OR gate.

The stacking transistors are NMOS because of the higher mobility of electrons. The multiplexer is typically a pass transistor design; however, if the last stage of logic before the multiplexer in each cofactor is supply gated with PMOS as well as NMOS, no multiplexer is needed, but each last stage gate must have its own PMOS and NMOS supply gating transistors. It is not possible to share these transistors between gates if more than one output is Shannonized, which is discussed in the PMOS supply gating section below.

Literature Survey

This work is primarily an automatic tool for synthesizing combinatorial logic to save power using the Shannonization technique. The task of synthesizing logic generally is left to the SIS [1] logic synthesis tool. SIS is fairly obsolete but the goal here is to demonstrate how to improve a circuit with Shannonization, not to produce the best synthesized result in absolute terms. A practical use of this work would be to marry the design of the Shannonization program with state-of-the-art synthesis tools.

This work is based primarily on the work of my advisor [2], who described the design of an automatic tool to synthesize logic with Shannonization so the resulting circuit saved power. Most of those ideas are adopted here, including the use of recursion. One idea that was not adopted was collapsing the logic into sum-of-products form. The number of minterms can increase exponentially as the number of inputs increases linearly.

It is not practical to use Shannonization on an arbitrarily large circuit because any particular input soon has little influence on the gates deep in its logic cone. The cofactors are better when they are more independent, so the distance between inputs and multiplexers is better kept to a reasonable size. Operating on an arbitrarily large circuit requires the circuit be divided into partitions each of which is Shannonized independently. Most of the ideas for hypergraph partitioning are taken from a paper by Karypis [3]. A partitioning algorithm specifically oriented toward logic circuits is described in a paper by Saab [4].

Tschanz, et al., describe some of the problems and compare some of the solutions for reducing leakage power [5].

D. Sylvester and H. Kaul provide a survey of the power related problems in nanometer logic [6].

The ISCAS-85 benchmarks are described at the RTL level by M. Hansen, H. Yalcin, and J. P. Hayes [7].

The 32 nm predictive technology model is described by Cao, et al [8]. The Nano-CMOS program and those PTM models were developed by Cao, et al [9].

Shannonization

The Shannonization process begins with a circuit encoded in BLIF (Berkeley Logic Interchange Format). This format represents a logic circuit as a series of truth tables. For example, a circuit that performs a test for equality of a pair of two bit numbers performs the operation:

$$\text{equal} = a[1:0] == b[1:0]$$

This can be represented in terms of simple gates as follows:

$$\begin{aligned} e0 &= a0 \text{ XNOR } b0 \\ e1 &= a1 \text{ XNOR } b1 \\ \text{equal} &= e0 \text{ AND } e1 \end{aligned}$$

This is coded in BLIF as follows:

```
.model equal
.inputs a0 a1 b0 b1
.outputs equal
.names a0 b0 e0
00 1
11 1
.names a1 b1 e1
00 1
11 1
.names e0 e1 equal
11 1
```

The .names line represents the header of a truth table with the output last. The lines that follow give the body of the table with the output "1" last. Input combinations not

given in the table are understood to produce a "0" output. A "don't care" input value would be indicated with a "-" symbol.

The equality function can be represented in a logic diagram as follows:

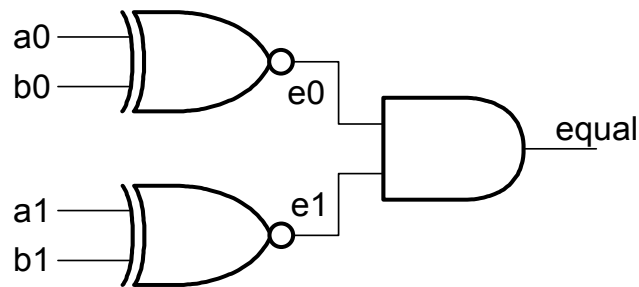


Figure 3: Logic Diagram of Equality Function

After the circuit is read into the internal data structure, it is given an initial technology mapping using Berkeley's SIS, the sequential circuit synthesis program. The initial mapping is important in judging the performance of the Shannonization technique so that the original and modified versions of the circuit have each been technology mapped by SIS. If the initial mapping is not performed then the statistics will compare the quality of the synthesis of the given circuit with the synthesis performed by SIS on the Shannonized circuit. The initial mapping is required if the given circuit is not technology mapped and includes functions that are more complex. However, the initial mapping by SIS is not possible for large circuits of 1000 gates or more; therefore, the initial mapping feature is optional.

In the equality function example, initial mapping is necessary because exclusive-NOR is not an elementary gate in the gate library used. Technology mapping gives the following result:

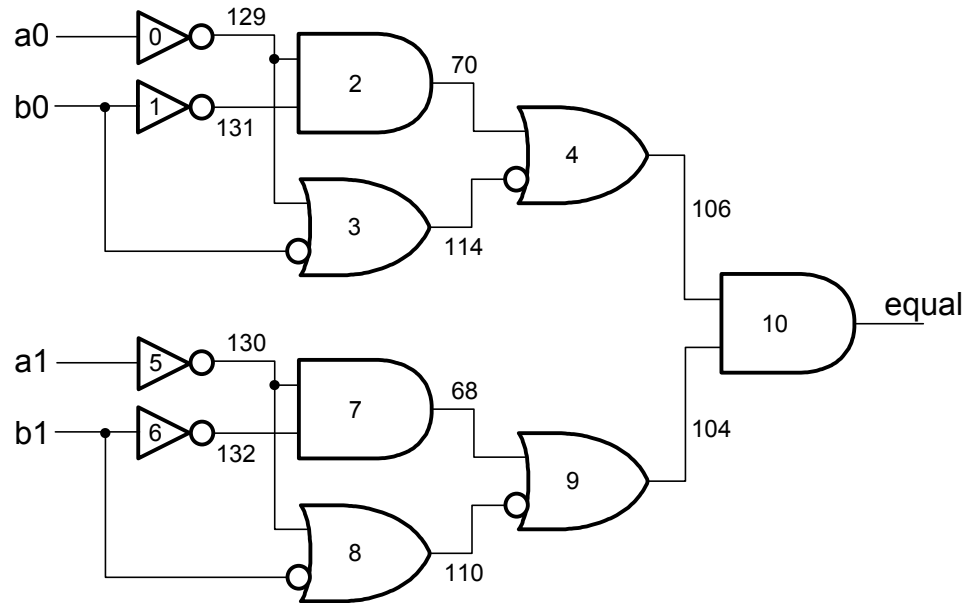


Figure 4: Logic Diagram of Technology Mapped Equality Function

Next, the circuit is partitioned and then Shannonization is performed on each partition independently. In this example, the circuit contains only eleven gates so partitioning is not necessary and Shannonization is performed once on the entire circuit.

Shannonization begins with the selection of a control variable, an input signal used to divide the circuit into two parts, one in which the control variable is false and another in which it is true. The goal of the analysis is to select a variable to minimize power, which is likely to occur for the circuit with the fewest number of gates.

The analysis is performed for each input signal by creating two copying of the circuit and setting the signal to false in one copy and true in the other. This makes some gates trivial and these are removed before the total number of gates is counted. The best candidate signals, which give circuits with the least number of gates, are then analyzed again, but this time each copy is fully technology mapped. The signal that gives the least number of gates with this more thorough test is made the control variable for the circuit partition. The number of candidate signals is three by default, but may be changed.

In this example, every input produces a circuit with the same number of gates. In each case, when one of the inputs is set to a constant, one half of the circuit in the technology mapped logic diagram above reduces to a triviality. That every input is an equally good choice is as expected since all the inputs to this function are symmetric. In this example, a_0 is the control variable, chosen arbitrarily.

Once the control variable is selected, the basic cofactored, or two-part, circuit is constructed. The two cofactor subcircuits are created, one in which the control variable is false and the other in which it is true. The inputs to the original circuit are connected to both cofactors, except for the control variable, which is not an input to the cofactors. The outputs from the two cofactors are combined in a series of multiplexers, one for each output, to produce the outputs of the original circuit. The control variable is the selector signal for all the multiplexers. An inverter provides the inverse of the control variable signal, also connected to all multiplexers. When the cofactor circuits are created, they are

technology mapped but not fully simplified in order to aid in identifying shared logic in the next step.

The circuit is divided into three subcircuits. CF2 is the subcircuit in which the control variable is false; CF1 has the control variable true. MUX is the subcircuit that contains the multiplexers and the inverter for the control variable. The following logic diagram shows the cofactored circuit for the equality function example:

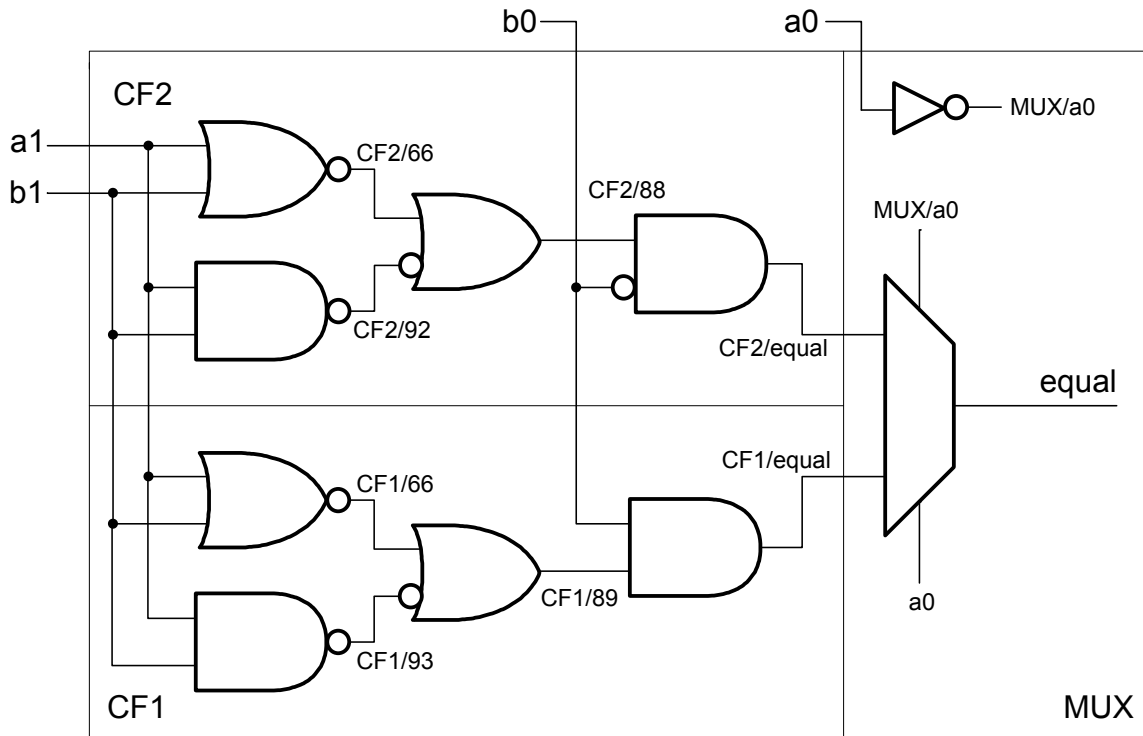


Figure 5: Logic Diagram of Cofactored Equality Function

Next, the cofactors are combined into a single shared subcircuit and resynthesized, to find and eliminate redundant logic. The goal of this step is to identify the logic that appears in both cofactors. This problem occurs because some logic lay outside the logic cone of the

control variable. When the cofactors were created, they each had been simplified somewhat because the control variable was set to a constant. However, some of the remaining logic, especially near other inputs, was unaffected by the control variable. That logic was duplicated between the cofactors. The goal of cofactoring is to save power by disabling unused logic, but logic that appears in both cofactors is never disabled and only wasted area. Therefore, this step of Shannonization removes redundant logic, but leaves the cofactored logic combined in a single subcircuit.

The following logic diagram shows the equality function example with redundant logic eliminated:

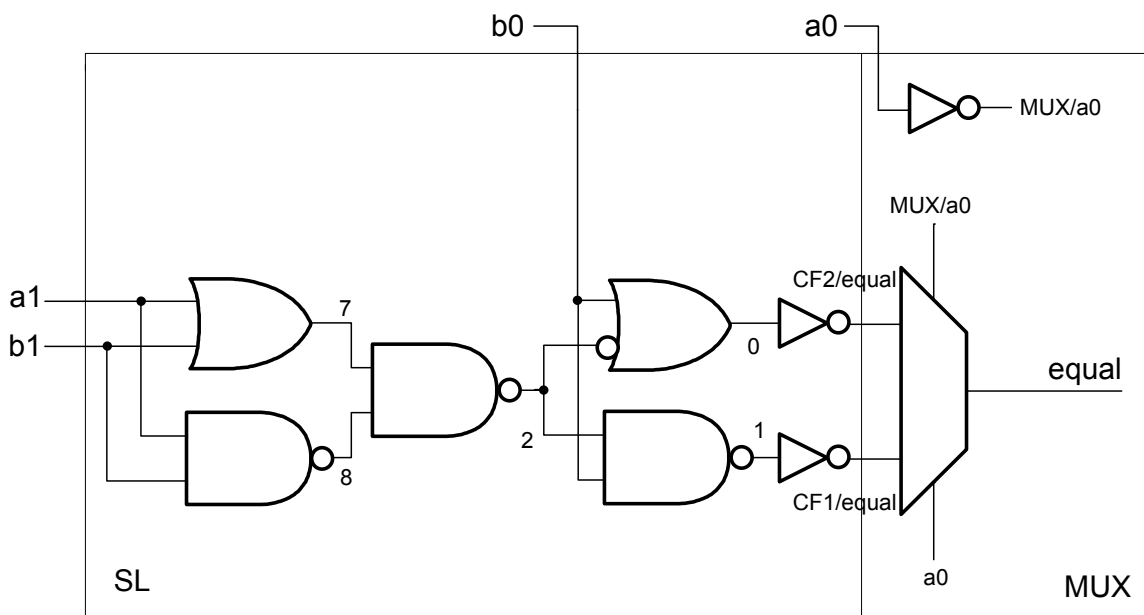


Figure 6: Logic Diagram of Equality Function with Shared Logic

At this point, the circuit is organized so that logic used by both cofactors can be separated, as pre-multiplexer shared logic, and logic used by each cofactor can be supply

gated. This Shannonization technique does not attempt to identify post-multiplexer shared logic.

In the next step, the logic is separated into two cofactors and the shared logic. The logic cone of each gate includes the CF1 and/or CF2 inputs of some multiplexers. If the cone includes both cofactors, then the gate is shared logic. The analysis is carried out by tracing the output signals back from the multiplexers, flagging each new gate as required in CF1 or CF2. Gates flagged as both are placed in the SL, shared logic subcircuit. Gates flagged as required by one or the other are placed in the CF1 subcircuit or the CF2 subcircuit, to be supply gated.

Finally, each of the three subcircuits is given a final technology mapping. Then NMOS supply gating transistors are added. The gating transistor for CF1 is switched by the control variable. The gating transistor for CF2 is switched by the inverse of the control variable.

The following logic diagram shows the final version of the Shannonized equality function example. The shared logic is slightly different than in the diagram above due to technology mapping, but the CF1 and CF2 logic is the same as above. In the diagram below, supply gating is represented as a transistor connected to a cofactor block, indicating that each gate in the cofactor is connected to the virtual ground provided by the drain of the gating transistor.

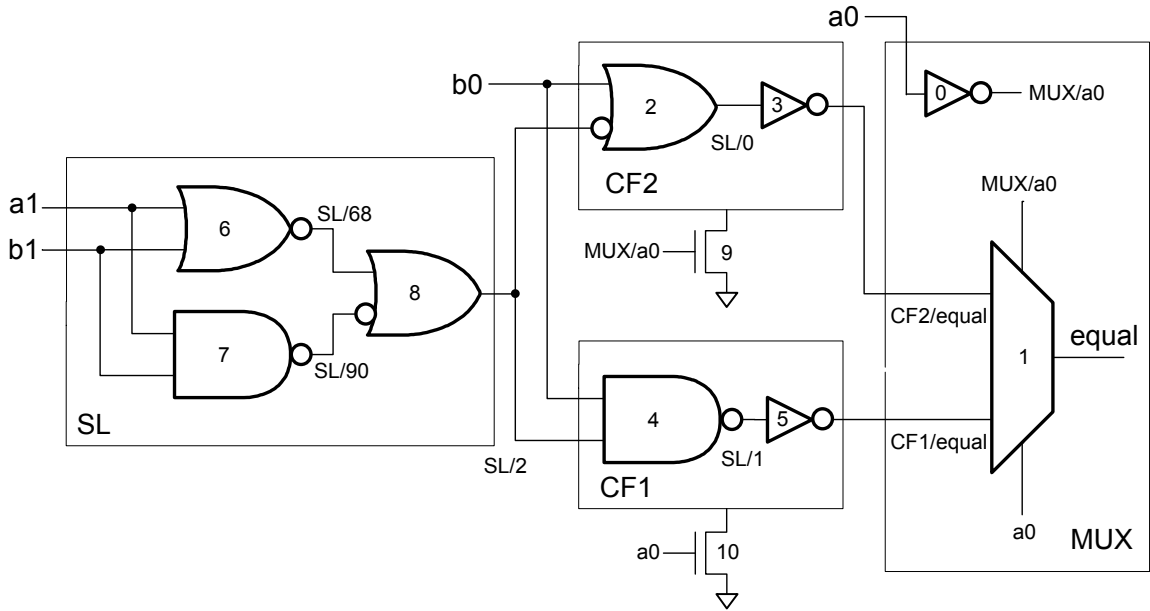


Figure 7: Logic Diagram of Shannonized Equality Function

Shannonization is not carried out if the resulting circuit requires excessive area or estimated power. Power is estimated as area but halved for gates in the cofactors which have a 50% probability of being enabled, depending on the input vector. If the resulting subcircuit requires a 50% increase in area or power over the original, Shannonization is aborted and the original subcircuit is kept, unmodified. The limits in area and power increase can be changed, independently, by the user of the program. The Shannonization technique is applied recursively to a depth of two, but this is an option that can be changed by the user. A depth of one defeats the recursion feature.

The output of the program is a SPICE file with the original circuit, after the initial technology mapping, and the modified, Shannonized circuit, each arranged as a

subcircuit, which can be simulated and compared. Each gate is listed as a subcircuit reference to a gate library.

For the equality function example, the following SPICE subcircuits are generated by the program when given the BLIF input above. These subcircuits are represented graphically in logic diagrams above.

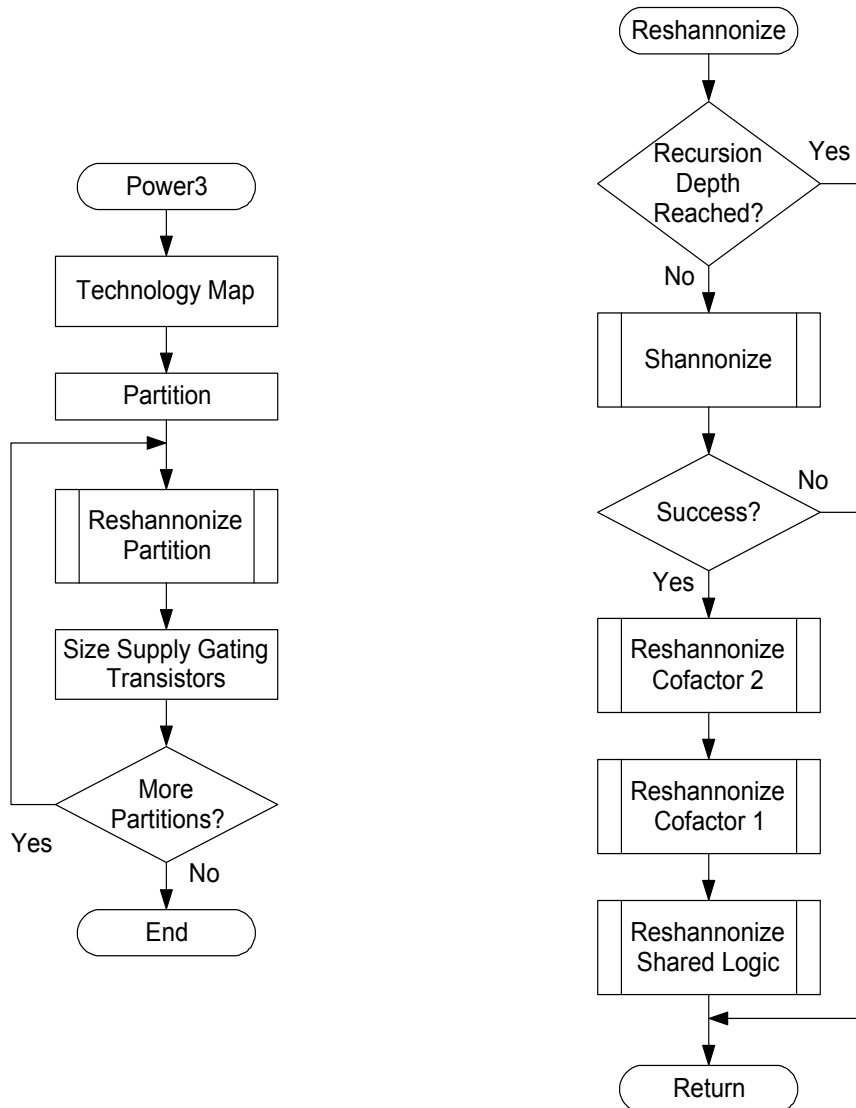
```
.SUBCKT equalorg _a0 _a1 _b0 _b1 _equal _VDD _VSS
XU0 _[129] _a0 _VDD _VSS _VDD _VSS inv1
XU1 _[131] _b0 _VDD _VSS _VDD _VSS inv1
XU2 _[70] _[129] _[131] _VDD _VSS _VDD _VSS and2
XU3 _[114] _[129] _b0 _VDD _VSS _VDD _VSS bor2
XU4 _[106] _[70] _[114] _VDD _VSS _VDD _VSS bor2
XU5 _[130] _a1 _VDD _VSS _VDD _VSS inv1
XU6 _[132] _b1 _VDD _VSS _VDD _VSS inv1
XU7 _[68] _[130] _[132] _VDD _VSS _VDD _VSS and2
XU8 _[110] _[130] _b1 _VDD _VSS _VDD _VSS bor2
XU9 _[104] _[68] _[110] _VDD _VSS _VDD _VSS bor2
XU10 _equal _[106] _[104] _VDD _VSS _VDD _VSS and2
.ENDS
```

```
.SUBCKT equalmod _a0 _a1 _b0 _b1 _equal _VDD _VSS
XU0 _0/MUX/a0 _a0 _VDD _VSS _VDD _VSS inv1
XU1 _equal _0/MUX/a0 _a0 _0/CF2/equal _0/CF1/equal _VDD _VSS _VDD
_VSS mux21
XU2 _0/SL/[0] _b0 _0/SL/[2] _VDD _0/CF2/VSS _VDD _VSS bor2
XU3 _0/CF2/equal _0/SL/[0] _VDD _0/CF2/VSS _VDD _VSS inv1
XU4 _0/SL/[1] _b0 _0/SL/[2] _VDD _0/CF1/VSS _VDD _VSS nand2
XU5 _0/CF1/equal _0/SL/[1] _VDD _0/CF1/VSS _VDD _VSS inv1
XU6 _0/SL/[68] _a1 _b1 _VDD _VSS _VDD _VSS nor2
XU7 _0/SL/[90] _a1 _b1 _VDD _VSS _VDD _VSS nand2
XU8 _0/SL/[2] _0/SL/[68] _0/SL/[90] _VDD _VSS _VDD _VSS bor2
XU9 _0/CF2/VSS _0/MUX/a0 _VDD _VSS _VDD _VSS nmosgate N=2
XU10 _0/CF1/VSS _a0 _VDD _VSS _VDD _VSS nmosgate N=2
.ENDS
```

Synthesis Flows

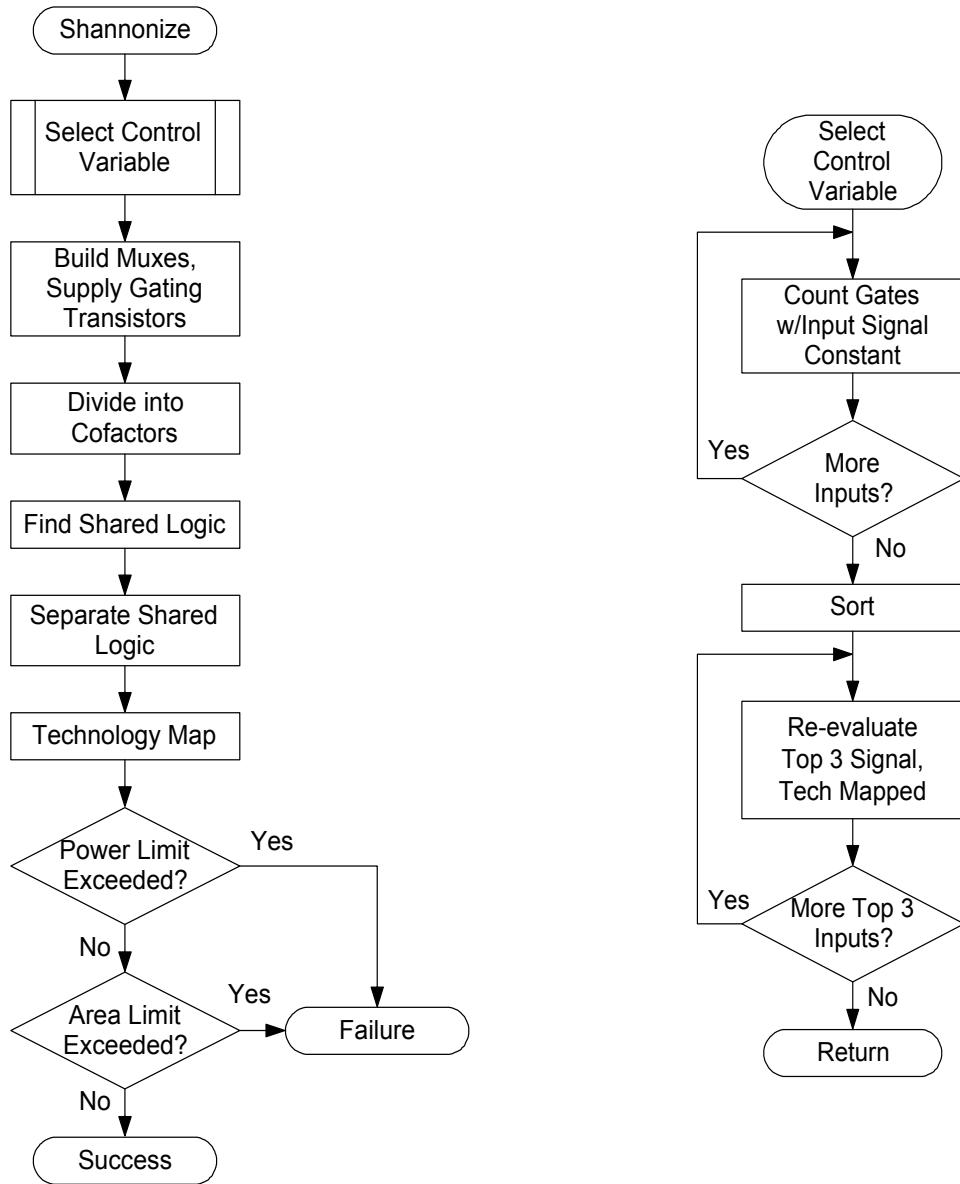
The following flowcharts show the major steps in the synthesis process. The Power3 flowchart is the top-level organization, which consists of technology mapping the given logic description, partitioning, and performing recursive Shannonization on each

partition. The Reshannonize flowchart indicates that the cofactors and shared logic of each Shannonized subcircuit is reprocessed until the recursion depth limit is reached.



In the last two flowcharts below, the details of Shannonization are carried out. The control variable is selected by counting gates for each input signal set to a constant and the trivialized gates removed. The top three candidates are evaluated again after a full technology mapping. Candidates are tested to avoid creating a dependency loop. Then the multiplexers and supply gating transistors are built. The circuit is divided into

cofactors. Then the shared logic is found and separated and the final technology mapping is performed. If the result exceeds power or area limits, the Shannonization is aborted.



Circuit Representation

In the most general form, circuits consist of components connected by wires. This is the form used for analog electrical circuit analysis by the program SPICE from Berkeley. At

the logic synthesis level, a circuit is a system of gates connected by signals. Each signal is the output of one gate. An input signal to the circuit is presumed driven by an external gate. Each other signal is driven by an internal gate. This is the form used for logic analysis by the program SIS, also from Berkeley. Communication with the SIS program is done in BLIF, Berkeley Logic Interchange Format.

For sophisticated techniques, a system of gates and signals is not adequate to represent a circuit. A two-tiered system is needed to support partitioning. Moreover, a full hierarchical system provides the flexibility for the recursive application of the Shannon Expansion technique. Therefore, a circuit includes a tree of subcircuits and a set of signals. Each signal includes a tree of subsignals. The tree of a signal is a subtree of the circuit tree.

The data structure that represents a subcircuit includes links to its siblings, children, and parent as well as the circuit to which it belongs. Similarly, the data structure that represents a subsignal includes links to its siblings, children, and parent as well as the signal to which it belongs. There is also a link, called a join, from each subsignal to a subcircuit that represent the connection of a subsignal to a subcircuit, so that each subcircuit can enumerate its list of connecting subsignals. A connection can be one of several types: input, output, internal or power supply.

Typically, the order of subsignals within a subcircuit is not important. In this representation, subsignals are organized into signal trees. For communication with SIS,

signals are identified by name. With SPICE, component nodes are identified by position, but most gates represent commutative logical operations, in which order has no effect. However, for some gates, especially multiplexers, order matters. For this purpose, the data structure maintains the order of subsignals within a subcircuit.

Despite the general hierarchical organization, the lowest two levels of the circuit hierarchy always represent gates as sums of products and minterms. A sum of products subcircuit is understood to represent a gate by having one output subsignal, a set of input subsignals, and a set of minterm subcircuits. The sum of products is the logical OR of its minterms. Each minterm has a set of input subsignals. The minterm is the logical AND of its subsignals. Each subsignal may be flagged as inverted.

A subcircuit includes a partition number, which was assigned to each gate by the partitioning algorithm. There are also several measurements of a subcircuit's size, used during the recursive Shannonization process. These are the gate count, with which a cofactor's supply gating transistor was sized, and the subcircuit's area and estimated power, which were used to prevent Shannonization that would produce excessively large or inefficient circuits.

Since communication with SIS required named signals, the circuit data structure has a system of naming signals relative to the circuit tree. Each subcircuit and signal includes a name. To allow any signal to be named relative to any subcircuit, a signal needs to have a home subcircuit in which to anchor the signal name, so that a relative name can be

resolved. Tying signals to particular subcircuits prevents copying and editing subcircuits, important operations in the Shannonization process. So the system of naming signals with relative subcircuit “pathnames” was never solved satisfactorily for this program. In most parts of the program, editing subcircuits and naming signals is handled well with this design, but there is also a lot of code to correct the subcircuit to which a signal's name is anchored.

A signal also has fields to indicate which cofactors depend on it for supply-gating, its distance along the signal graph, in which gates are vertices, for logic cone analysis and an extra link to a duplicate to facilitate copying subcircuit trees.

Partitioning Algorithm

Partitioning divides the circuit into subcircuits of a manageable size. The goal is to group related gates so that there are a minimum number of signals straddling partitions.

Shannon expansion or any kind of logic synthesis, carried out on the individual partitions is constrained by the input and output signals it must reproduce.

Partitioning is performed in terms of graph theory rather than logic gates, so a complete circuit is represented by a hypergraph. A gate is a node, and a signal is a hyperedge. In an ordinary graph, an edge connects a pair of nodes. But since a logic signal output can fan out to many inputs, a signal has to be represented by a hyperedge, which can connect many nodes.

In this program, there is a similar distinction between circuits and graphs. The circuit data structure is mapped onto a hypergraph data structure before partitioning is carried out on the hypergraph. When complete, the circuit is divided into subcircuits using the partition numbers from the hypergraph.

While the circuit is represented as a hypergraph, two stages are carried out called coarsening and uncoarsening. During coarsening a series of progressively simpler hypergraphs are produced that mean to represent the circuit at progressively higher levels. This continues until the average node represents the desired number of nodes in the original hypergraph. At the highest level, each node represents one partition. Then during uncoarsening, the lower level nodes are assigned to their partitions and some are reassigned temporarily to other partitions in search of a better configuration.

The hypergraphs form a linked list, each representing the original hypergraph at a different level of abstraction. Each hypergraph contains a list of vertices and edges. Each edge contains a list of vertices that the edge connects. Vertices do not contain lists of edges because the bookkeeping would not be worth the effort. The list of vertices associated with an edge must be sorted, so it is better to keep the connection information in the edges.

Lowest level edges and vertices contain references to the subsignals and subcircuits each represents, in order to map the hypergraph back to the circuit when partitioning is finished. Edges and vertices also contain their current cost and the number of lowest

level edges and vertices each represents so that the cost functions can be accurately evaluated at the higher levels. Edges also contain the number of partitions spanned in service of the hyperedge cost function. Finally, vertices contain the current and best partition number, which is the indication of how the hypergraph is partitioned.

In each step of coarsening, a new, simpler hypergraph is produced to represent the previous one. This is carried out by selecting a hyperedge and drawing together all the nodes connected to that hyperedge into a group. The group is represented as a single node in the simpler hypergraph. This is done repeatedly until all the nodes have been drawn together. The hyperedges are represented in the simpler graph if they join distinct groups of nodes.

Hyperedges are selected for drawing together by cost. Lowest cost edges have highest priority. In this process, the cost of an edge includes the number of vertices connected (similar to the number of partitions) but is subtracted from the cost of the joined vertices. The purposes of this stage are to envelop expensive edges and join inexpensive vertices leaving comparably expensive groups of vertices joined by cheaper edges. After these costs are determined, the edges are sorted with lowest cost first. Then the vertices attached to each edge are drawn together, but only if they form a group disjoint from any other group. This process is repeated until there are no edges left. The groups need not be disjoint across repetitions. (Finally, if there are any vertices remaining, each is taken to the higher level as in own group.)

Much of the rest of the coarsening process consists of constructing the new hypergraph from the groups assigned to the nodes in the lower level hypergraph. Vertices in the new hypergraph get the sum of the number of lower level vertices represented. Edges that are entirely enveloped by this process are discarded. Edges that become parallel, joining the same set of new vertices, are combined and the number of lower level edges represented is summed. An edge that joins a subset of the vertices that are joined by another edge is similarly combined into the larger edge. These are found by sorting the vertices of each edge by partition number, which facilitates finding and discarding duplicate higher level vertices. After the vertices within an edge are sorted, the edges are sorted by the number of vertices, largest first, so subset edges always appear later on the list.

When the highest level of coarsening is reached, the number of vertices has reduces until each vertex represents the desired number of lowest level vertices. At that point, each vertex represents a partition.

At each step of uncoarsening, the partition numbers from the vertices at the next higher level that represent a group at the lower level are assigned to the lower level vertices. Then the cost of the entire hypergraph partitioning is evaluated and recorded as the best partitioning encountered. Uncoarsening is then the repetition of the refinement process and the best partitioning encountered is also recorded, and becomes the final product of the partitioning algorithm.

Refinement is carried out for the number of iterations given by the tunable parameter, *tries*. In each iteration of refinement, a vertex is selected at random and the relative cost of putting it in each partition is evaluated. Relative costs of putting the vertex in each partition are evaluated simultaneously. Given the absolute cost of the hypergraph and the relative cost of each partition, the absolute cost of moving the vertex to any partition can be determined by subtracting the old partition's relative cost and adding the new partition's relative cost to the old absolute cost. This allows for the use of a relative cost calculation, which is simpler than an absolute calculation.

The relative cost is found by removing the vertex from its original partition and for each partition evaluating the cost of having the vertex relative to not having it. Then for each edge that includes the vertex, subtracting the edge cost from each partition in which the edge has a vertex, except the vertex being tested. Of course, this will underestimate the cost of putting the vertex in each partition, but it will do so for all partitions equally.

Once the relative cost for every partition is determined, the lowest cost partition other than the vertex's current partition is selected. If this other partition has a lower cost than the current partition, it is accepted; otherwise, the annealing strategy is used to determine if it should be accepted in spite of having a higher cost. If it is accepted, the current cost and the partitions' vertex count is updated, and for each edge that includes the vertex, the vertex is positioned in the edge's list of vertices so that the partition numbers are in order.

Cost Functions

The cost functions used in evaluating the quality of a partitioning are two: for hyperedges and vertices in each partition. The original partitioning algorithm had a cost function for hyperedges, which was essentially the $(K-1)$ metric. The cost of a hyperedge was the number of partitions, K , that it straddled, which therefore would require $K-1$ wires to join. The number of vertices was constrained by a hard limit. That design was more appropriate for mapping logic onto a fixed resource like an FPGA. For the current application, the size of the partition is not important, but cannot be ignored. A circuit with everything in one partition has no cost associated with hyperedges straddling partitions, so there has to be a cost associated with too many vertices in a partition in order to prevent a one partition circuit from being the ideal.

The early version of the cost function for vertices in a partition simply increased linearly with the number of vertices. This had a tendency to leave orphan vertices in partitions by themselves, so a $1/f$ term was added to penalize very small partitions and a tunable parameter, $vopt$, was introduced to indicate the ideal number of vertices per partition, so the cost of n vertices, $nvert$, was:

$$cost(nvert) = \frac{vopt}{nvert} + \frac{nvert}{vopt}$$

But that gave partition sizes over a relatively wide range both above and below $vopt$ so to make that goal more desirable, the terms were squared. Finally, when the annealing algorithm was implemented, it became important to have a zero cost that represented an

ideal state. The existing vertex cost function had its minimum at two, so this offset was removed leaving the final cost function for vertices in a partition:

$$cost(nvert) = \left(\frac{vopt}{nvert} \right)^2 + \left(\frac{nvert}{vopt} \right)^2 - 2$$

The hyperedge cost function was based on the original (K-1) metric, although that is only relevant during uncoarsening. During coarsening, the partitions are still in the formative stage.

In order to compare partitionings the cost of vertices and cost of edges need to be combined into an overall cost. The number of partitions can be ignored, however. It is the same for the partitionings being compared. But the cost function for hyperedges is evaluated for each hyperedge whereas the cost function for vertices is evaluated for each partition. For the hyperedge function, the value was scaled down by $vopt$, which is a number of vertices in a partition and multiplied by a new parameter, $kvpe$, which is designed to be the cost of a vertex relative to a hyperedge. So the final hyperedge cost is:

$$cost = kvpe \cdot \frac{(K - 1)}{vopt}$$

This leaves a cost per edge, which summed over all edges is a unitless result. As with the cost of vertices, neither has any units of vertices or hyperedges, so they can be summed together to get the overall cost of a partitioning. Of course there is no requirement that these formulas match in units since they are dependent on tunable parameters and used to

inform an heuristic, but the benefit of getting the units right is that the parameters are more likely to work independent of scale, whether the circuit is small or large.

The hyperedge cost function is still dependent on the assumption that the number of vertices and cut hyperedges are related independent of partition size. What if cut hyperedges go as the square root of vertices? It would still be possible to get scale-independent values of *k_{vpe}* and *v_{opt}* although it would change the meaning of *k_{vpe}*. And there were some tests that suggested the relationship was linear.

An earlier version of the program (version 2) had a metric for activity variance not in the current version (version 3). That metric was designed to relate gates that had a similar activity level so the Shannon expansion could be carried out on the most active gates only, in order to save power without too much cost in circuit area. It was important to use activity variance, not just activity, as a metric in order to cluster gates with a similar activity level. If activity level were the metric, the most active gates would be the most desirable to all partitions and the most active gates would be as likely to be split up among rivals as clustered together. With that metric, there were three things to balance: vertices, hyperedges and activity variance. But the goal of saving area was abandoned when saving power proved difficult enough.

The design of a cost function for activity variance demonstrated the difficulty of working with cost functions across the hierarchy of coarsened hypergraphs. Higher level vertices are simply weighted according to the number of lower level vertices they represent.

Higher level hyperedges have the weight of parallel lower level hyperedges and, more crudely, the lower level hyperedges that straddle partitions. Activity variance was chosen in part because it could be accurately scaled up to high levels by summing statistics. A more useful measure would have been correlated activity relating gates that change together, but no technique could be found to apply except at the level of individual gates.

Annealing Algorithm

The partitioning algorithm was improved with the inclusion of an annealing strategy. Prior to this improvement, a greedy algorithm was used, which was just the trivial strategy of choosing the alternative with the minimum cost. Annealing is more sophisticated in order to avoid getting caught choosing the best among immediate alternatives but not the best of all alternatives, some of which are not immediately apparent.

During the uncoarsening phase, the hypergraph is refined by selecting a vertex at random and determining the cost of moving it to each of the other partitions. The goal is to select the best partition for the vertex, and then to do this repeatedly with a long series of vertices to produce the best overall partitioning. But because only one vertex at a time is a candidate for movement, better alternatives that appear only after moving two or more vertices will not be found by a greedy algorithm. Instead, that was replaced with an annealing strategy, which allows some temporary movement away from the goal in order to increase the likelihood of reaching the goal overall.

The refinement process still consists of a series of trials of selecting a vertex at random and evaluating the cost of moving it to each other partition. The decision is whether to leave the vertex in its original partition or to move it to the best alternative partition, which was trivial with the greedy strategy. Note that the annealing strategy implemented does not consider partitions other than the original and the lowest cost alternative. The strategy could have been designed to choose a random partition as well as a random vertex and evaluate the cost; however, the likelihood of stumbling upon an improvement and the inefficiency of calculating the cost of moving to a single partition made this design alternative impractical.

The annealing strategy requires only the cost of the original and alternative partitionings and the history of these. The strategy's output is a Boolean flag which indicates whether to move the vertex to the new partition or not. The strategy is named for the natural process by which crystals are formed, where the system as a whole dissipates heat and moves to a lower energy state although any particular part of the system may increase in energy level, especially temporarily. The annealing strategy used in the program does not model the movement of energy between parts of the system, but it does model the likelihood that a part will enter a higher energy level and the overall tendency toward lower levels over time.

The value of an annealing strategy is that it can overcome local cost minima in the graph of possible partitionings. At any particular time, the current partitioning may be represented as a node on the graph, and each change of one vertex to one other partition

may be represented as an edge to an adjacent node. If the cost associated with each adjacent node is higher than the current node, then any movement through the graph requires at least a temporary movement to a higher cost partitioning. If the minimum configuration is represented by a node elsewhere on the graph, then it can only be reached via the higher cost partitioning. The greedy strategy is the only one that lacks this capability, but the significance of the annealing strategy is that it tolerates the higher cost configurations by amounts and durations that resemble physical systems, which have been tested in the field. Any non-trivial strategy might be made to work but the design was made to resemble annealing because it is known to be robust and efficient. In particular, the cost functions were designed so that the ideal configuration representing no cost had a value of zero. Of course, the minimal cost configuration is not known in the program except in trivial cases, but this zero cost models the zero energy state or absolute zero in temperature. This allows for proper scaling in comparing non-optimal configurations.

Another important property of the strategy implemented is that it does not require any adjustable parameters, except the number of iterations to run the refinement process. The strategy scales the relative cost of the alternative partitioning relative to the current one. The result is independent of the absolute scale of the cost functions. A negative result represents a lower energy state, which is always accepted, just like the greedy algorithm. But in annealing, a positive result is given further consideration. These are assumed to form the positive half of a normal distribution. The assumption is that the cost of successive partitionings resembles a signal with proportional noise. The signal reduces

over time and the noise does as well. Scaling the relative alternative cost to the current cost leaves only the noise, which is assumed normally distributed, at least on the positive side. So a data point, x , depends on the cost of the current partitioning, $cost$, and the cost of the alternative, $newcost$:

$$x = \frac{newcost - cost}{cost}$$

The values at this point have a mean of zero, but some measurement of the variability is needed so the strategy can avoid selecting very high cost configurations. So the variance of these values is computed; however, this would give the more weight to the earlier values and maintain too long of a history. Instead, a weighted average of the squared values is used as the variance, so that the weight of older values decays exponentially. The choice of the weighting factor determines how long older values have significance. The reciprocal of this weighting factor is the number of iterations before the significance of a value has diminished to 37% of its original amount. That was selected to be the square root of the number of iterations for the entire refinement process. Therefore, refinement is as many iterations of the 37% diminishment cycle, as there are trials in each diminishment cycle. So the variance, var , depends on the weighting factor, $delta$, the data point, x , and the previous value of the variance, $oldvar$. And $delta$ depends on the number of iterations, $ntries$:

$$delta = \frac{1}{\sqrt{ntries}}$$

$$var = delta(x^2 - oldvar) + oldvar$$

The values are scaled by the square root of the variance, producing a z-score:

$$zscore = \frac{x}{\sqrt{var}}$$

The probability density function for a normal distribution is used to relate the z-score with the probability that the higher cost alternative will be accepted. The essence of the annealing strategy is that there should be an exponential function of the cost that produces two extremes, each with an infinitesimally small probability of occurring: one in which an infinity bad alternative is accepted and one in which any higher cost alternative is rejected. Clearly, the first case corresponds to an exponential approaching zero at infinity and the second, one at zero. There is one degree of freedom left in such an exponential, which appears as the reciprocal of the Boltzmann constant in the physical equation. This is the coefficient of the argument to the exponential and determines the steepness of the exponential function. By specifying the value of the coefficient, the probability of the decision is determined.

The simplest way to express this function is as the desired probability raised to a power, so that when the power is one, the result is the desired probability. This function will give zero when the power is infinity, and one when the power is zero. The alternative partitioning should be selected when a random number is less than this result. The power must be a coefficient multiplied by the z-score so that the power is one when the z-score corresponds to the desired probability. Therefore, the z-score of the alternative partitioning is scaled by the z-score of the desired probability, which is the inverse probability density function for a normal distribution of the desired probability.

The inverse probability density function for a normal distribution, sometimes called the probit function, can be difficult to compute. In fact, it has no closed form. But near the center of the distribution, it is a relatively straight line. For the annealing strategy, the probability of selecting a higher cost alternative partitioning was chosen to be 0.5 initially and reduce linearly with the number of iterations to zero probability at the end. Since the higher cost alternatives form only the positive half of a normal distribution, these probabilities correspond to 0.75 to 0.5 on a complete bell curve and z-scores of 0.6745 to zero. In this region, the normal distribution is almost flat. At zero, the value is $1/\sqrt{2\pi} \approx 0.3989$ and at 0.6745, the value is about 0.3178. The inverse cumulative function, probit, can be approximated by a straight line with slope $\sqrt{2\pi} \approx 2.5$ so that at 0.5, the error is zero and at 0.75, the error is 7%. Using a distribution that is only the positive half of a normal distribution, this constant becomes 1.25.

So the wanted probability, *prob*, depends on the total number of iterations, *ntries*, and the current iteration number, *n*. The value of the probit function, *probit*, depends on *prob*:

$$prob = 0.5 \left(\frac{tries - n}{tries} \right)$$

$$probit = 1.25 \cdot prob$$

The decision to accept the alternative partitioning is made by the following expression. The alternative is accepted when the expression is true. The right-hand side gives the

probability that the partitioning with the z-score should be accepted given the average probability, *prob*, and its probit function value, *probit*. The left-hand side of the expression is a random number, *r*, between zero and one:

$$r < \text{prob}^{\frac{\text{zscore}}{\text{probit}}}$$

In addition to the annealing strategy, an alternative partitioning is also accepted when its cost is less than that of the current configuration. Since the annealing strategy allows inferior partitionings to be pursued, the partitioning at the end of the uncoarsening algorithm may not be the lowest cost configuration during the whole process. So the algorithm keeps track of the lowest cost and the partitioning to go with it, which is what is returned by the algorithm.

PMOS Supply Gating

The Shannon expansion technique saves power by switching off one set of gates or another with an NMOS supply-gating transistor. The outputs of these sets of gates, the cofactors, are combined in a multiplexer. But if one or the other of the cofactor outputs were completely disabled, then the multiplexer itself could be eliminated and the outputs of the cofactors tied together as the final output signal for the Shannonized circuit.

To switch off a cofactor completely requires PMOS supply-gating as well as the usual NMOS supply-gating transistor. NMOS gating prevents a logical "0" signal from being output when switched off while PMOS prevents a "1" from being passed. PMOS gating is required only on the final level gates, not the entire cofactor, because the goal is only to

disable the final output, not to switch off the gates to save power as with the NMOS gating transistor. A gate with PMOS and NMOS supply gating resembles a tri-state buffer where the inputs to the supply gating transistors are the inverted and non-inverted signals that enable the output. If the original last level gates of both cofactors were simple inverters, then with the supply gating and the outputs tied together, these buffers would look a lot like a static CMOS inverting multiplexer. That makes sense, since the goal is to obviate the multiplexer in the design without PMOS gating. So the purpose of adding PMOS gating to the last level gates is to take advantage of the existing NMOS gating and complete the formation of the multiplexer there, so no discrete multiplexer is required.

The technique of using PMOS gating to avoid discrete multiplexers only works if each final level gate has its own independent PMOS and NMOS supply gating transistors. Shannonized circuits with multiple outputs cannot share gating transistors lest these form unexpected electrical paths between the multiplexers. The consequences of sharing gating transistors are evident in a circuit consisting of two static CMOS inverting multiplexers with selector signal transistors tied together. The circuit is equivalent to a Shannonized circuit with final level gates that are inverters and shared PMOS and NMOS gating transistors. The following schematic shows the result with NMOS gating only. Gates that are off are marked with an "X":

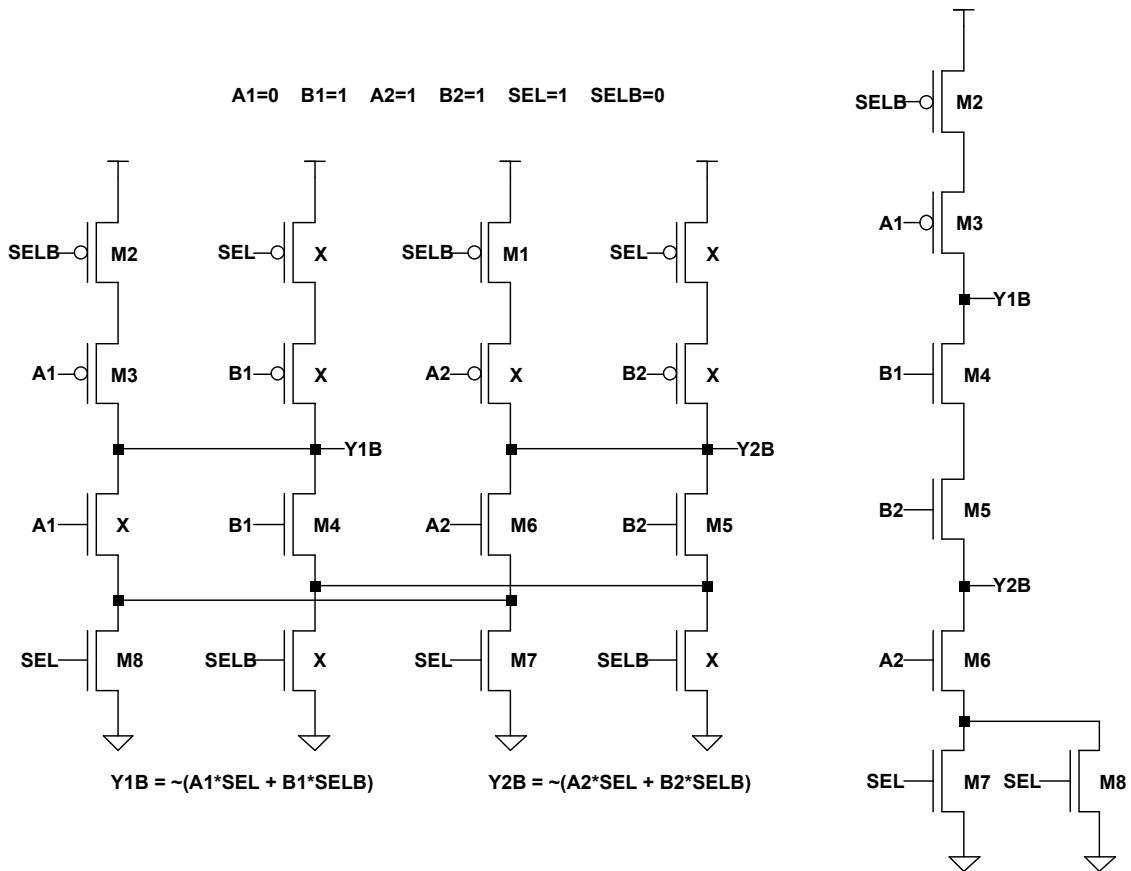


Figure 8: PMOS Gating Circuit

The flaw in the design is that when a supply-gating transistor is turned off it provides a floating node that connects the two multiplexers together (between M4 and M5). For some input vectors, the resulting circuit is a short through six transistors to ground. The output signals are connected through transistors so the signals are weakened and a high output has a lower voltage than it should and a low output has a higher voltage than it should.

The solution to this problem is to use separate supply gating transistors.

Test Output Files

The output of the program is a SPICE netlist that includes a table of circuit area measurements. The file is input to a version of SPICE to measure power. Both the area and power information is provided for the original and modified versions of the circuit so they can be compared and tested.

Each version of the circuit is given in a subcircuit definition that consists of a list of subcircuit calls to library gates. The circuits and library gates are defined with explicit power supply nodes to facilitate the supply gating and to ensure independent measurement of power for original and modified circuits. In the main body of the netlist, each version of the circuit is called. The circuits share the same set of input signals, but each has its own separate output signals and power supplies. Each output signal is loaded with a fan-out-of-four inverters library subcircuit. The only other things in the SPICE netlist are include file references, one for each input signal and one overall header file. This organization allows the circuit descriptions to be stored separately from the input vectors and to include only as many input vectors as are needed. The separate input vector files and header file free the circuit description file of any cycle time information or any peculiar syntax that might cause portability problems. The output of the program works with both Linear Technology's LTSPICE and Synopsys's HSPICE but slightly different header files were needed because of syntactical differences.

The main header file contains the actual power supply voltage sources, directives to record the power required from each, and the directive for temperature and to run the transient analysis. The main header also has include file references to the transistor models file and the gate library file. The input signal files were generated with a small utility program that created SPICE PWL (arbitrary piece-wise linear) voltage sources for random input vectors with certain specifications.

Test Conditions

The following conditions were used as the baseline conditions for power tests:

- Predictive 32 nm transistor technology
- 0.9 V power supply
- Fan-out-of-four inverter loads on each output
- 5 ns cycle times (200 MHz clock frequency)
- 1000 cycles per test
- 20% input activity level
- 20 ps rise and fall times
- 110° C
- Two levels of Shannonization recursion
- Limits of 50% increase in area and estimated power (powered area) per partition

In addition, for certain tests, the 32 nm model was compared with 45 and 65 nm technologies using a 1.0 V supply and the 5 ns clock was compared with 2 ns, 10 ns and 1 μ s clocks. The largest benchmark, c6288, was run with only 100 cycles.

The program output was designed to work with any convenient version of SPICE, but the tests for this experiment were carried out using LTSPICE from Linear Technologies. For several 10% length runs with five benchmark circuits, the results from LTSPICE were compared with those of HSPICE, from Synopsys, and the two were found to agree within 1%.

Test Circuits

Tests were carried out on ten ISCAS-85 benchmarks and two other benchmark ALUs:

Name	Gates	Description	<i>Vopt</i>
c432	184	27 channel interrupt controller	11
alu2	277	2-bit ALU (not ISCAS-85)	91
c880	451	8-bit ALU	15
c1908	466	Error Correcting Circuit	10
c499	534	Single-Error Correcting circuit with XORs	6
c1355	594	Single-Error Correcting circuit with NANDs	7
c2670	776	ALU with comparator, equality checker, parity trees	30
alu4	816	4-bit ALU (not ISCAS-85)	100
c3540	1134	8-bit ALU with BCD arithmetic, logical shift	100
c5315	1743	9-bit ALU with addition, logic, parity	16
c7552	2126	32-bit adder, magnitude comparator, parity	22
c6288	2848	Multiplier, 240 full- and half-adders	5

Table 1: Descriptions of Benchmark Circuits

The *vopt* column included in the table above gives the optimal number of vertices (gates) per partition, which were found experimentally to produce the greatest power savings in the Shannonized version of the circuit under baseline conditions.

Larger circuits, alu4, c3540, c5315 c7552 and c6288, were not given the normal initial technology mapping by the program because of limitations of the SIS logic synthesis program, so these test results compare original circuits that were technology mapped by other means compared with modified, partitioned circuits technology mapped by SIS.

A number of benchmark circuits include a few inherent partitions. Each has several disjoint sets of signals and gates. There is nothing to be gained by merging these partitions for Shannonization, and the program cannot be arranged to do so. The minimum number of partitions for c2670 is 81; c5315 is 6; c7552 is 5 and c880 is 3.

Test Procedure

Tests were conducted to measure the power, area and delay of logic circuits synthesized by the program in order to determine its performance. Power was measured using LTSPICE. Area results were reported directly by the program. Delay was measured using the Design Compiler from Synopsys. In its default operating mode, the program outputs the circuit in SPICE format, but with an alternate output mode, it produced Verilog format, for compatibility with the Design Compiler. This allowed for delay results to be produced independent of power results. There is also a separate program, a

UNIX awk script, to derive Verilog from the SPICE output, so as to avoid resynthesizing the circuit from scratch and ensuring the circuits were equivalent. The Design compiler was also used to test that the modified circuits contained no dependency loops.

In every case the original and modified circuits were tested side-by-side and the percent change was reported according to the following formula:

$$\% \Delta x = \frac{x_{\text{mod}} - x_{\text{org}}}{x_{\text{org}}} \cdot 100\%$$

Since the desired outcome is less power, smaller area and shorter delays, improvements are reported as negative numbers.

To ensure the modified circuits produced logic results equivalent to those of the original circuit, all the outputs of the five smallest benchmarks were checked for logical correctness for hundreds of input vectors, using SPICE.

For power tests, the program generated SPICE output with reference to an input signal include file for each input signal. Of course, both the original and modified circuits used the same inputs. The input signal files were SPICE PWL (piecewise linear source) voltage sources generated randomly by a separate utility program. The voltage, cycle time and rise and fall times were parameterized in these signal files. Each output signal was tied to a fan-out-of-four inverters library subcircuit. Of course, original and modified circuit outputs did not share outputs. The gates were referenced in the generated SPICE output as subcircuits, defined in a separate gate library file.

The program's SPICE output referred to a general header include file, which encapsulated parameters and measurements commands whose syntax tend to vary among SPICE versions. The header file for baseline conditions follows:

```
* 32 nm, 5 ns header

.param T=5N
.param L=32N
.include "32nm_bulk2005.sp"

.include "gatelib.sp"

.param TR=20P
.param V=0.9

VDDorg VDDorg 0 dc 'V'
VSSorg VSSorg 0 dc 0

VDDmod VDDmod 0 dc 'V'
VSSmod VSSmod 0 dc 0

*LTSPICE
.tran '1001*T' uic
.measure tran avgpowerorg avg abs(V(VDDorg)*I(VDDorg))
      trig time val='T'
.measure tran avgpowermod avg abs(V(VDDmod)*I(VDDmod))
      trig time val='T'
.options temp=110
```

For delay tests, Verilog language output of original and modified circuits was analyzed by the Design Compiler and the results extracted via the following script:

```
for a in $*; do
for suffix in .sp .org.v .mod.v; do
a=`basename $a $suffix`
done
orgcmd="read -f verilog $a.org.v; report_timing"
modcmd="read -f verilog $a.mod.v; report_timing"
cmd=$orgcmd
for i in org mod; do
orgval=$val
val=`echo $cmd | dc_shell | \
      nawk '/data arrival time/{print $4}'`
```

```
cmd=$modcmd
done
modval=$val
echo $a $orgval $modval | tr ' ' '\t'
done
```

This method of measuring delay does not account for the delay due to Shannonization, in which the switching of cofactors due to the control variable adds a significant extra delay of about 25%.

The gate library included inverter, NAND, NOR, AND and OR two-input gates. In order to simplify the analysis using SIS, the library also included gates with each input inverted. AAND and AOR had the A input inverted. BAND and BOR had B inverted.

Selection of Tunable Parameters

Over the course of the development of the program, tests were conducted to determine the performance of the program, in terms of its ability to operate correctly, synthesize Shannonized logic that was equivalent to the original circuit and to minimize power consumption. Because of the development of these algorithms many new parameters were introduced, though of course every attempt was made to avoid them by deriving them from formulas based on general principles and existing parameters.

The remaining, tunable parameters may be changed by user of the program on the command line or from environment variables. The default values for these were the best values found experimentally during the development of the program. The performance of the program is not particularly sensitive to most of these parameters either in terms of

a change in the parameter value or the type of benchmark circuit under test, with the exception of the parameters related to partitioning, *vopt* and *seed*. The *vopt* parameter is the optimal number of vertices (or gates) per partition and seemed to vary significantly with the type of circuit. The *seed* parameter is the random number seed for partitioning and changes the particular partitioning. The program tries to group gates neatly with the fewest signals between groups, but there is a significant amount of variability in how this translates into a Shannonized circuit that requires minimum power.

Since there was no clear best single number for each of these two parameters, they were adjusted in preliminary 10% length runs of the program before the final tests were performed. Firstly, a series of *vopt* values were tried on the benchmark circuits under baseline conditions. The *vopt* values tried were all seven values from four to ten, inclusive, and 24 values in the decade up to 100, in the exponential series used for resistor values. For each benchmark, the power and delay results were plotted log-linearly and a second order curve fit was used to find the *vopt* that gave the minimum power requirement, while allowing for the variability between runs due to the random number seed. From these plots, optimal *vopt* values were selected by the advisor. For each benchmark, the optimal *vopt* was used for most of the final tests. There was also a series of final tests in which a series of *vopt* values were tried.

After the full complement of *vopt* values for each benchmark was determined, there was still a need to consider the variability in performance due to “random” effects. The partitioning algorithm uses random numbers because of a Monte Carlo aspect in selecting

vertices of the hypergraph to consider moving to other partitions. The random number *seed* variable has no meaning except that different values give different partitionings, which are of equivalent effectiveness, according to the partitioning algorithm. But the partitioning cost function is not the same as the power saving measurement of the final circuit, which appears as a random variation when the program is run with different *seed* values.

Since the goal of the program is to produce any synthesis of the logic that saves power, it is legitimate to run the program repeatedly with several random numbers and select the version that is best. That procedure is limited by the time required to carry it out. So to take reasonable advantage of the variability in *seed*, five values were chosen at random for five runs for each *vopt* value, for each benchmark. Because both *seed* and *vopt* are important in partitioning, any selection of *seed* is dependent on the value of *vopt*; however, Shannonization and its parameters are theoretically independent of partitioning and its parameters, so there is no need to evaluate a selected *seed* with more than one set of Shannonization parameters.

For the final test results, an optimal *vopt* was found for each benchmark and a *seed* was selected for minimum power for each *vopt* for each benchmark.

Test Results

The following tables summarize the results of tests of the performance of the program in using the Shannonization technique to synthesize logic with lower power without undue

increase in area or delay. The tables are organized with a row for each benchmark circuit and columns arranged primarily for power, area and delay and secondarily for a parameter that is the subject of comparison. Except for the last table, all table entries are percent change from original to modified circuit relative to the original, so decreases in power, area and delay are expressed in the table as negative numbers.

<i>vopt</i>	Power				Area				Delay			
	10	25	50	100	10	25	50	100	10	25	50	100
c432	-16%	-11%	-5%	0%	2%	8%	28%	0%	-12%	-2%	-4%	0%
alu2	-9%	-13%	-5%	-14%	11%	13%	19%	21%	-9%	1%	9%	-20%
c880	-20%	-20%	-18%	-18%	-2%	-1%	7%	2%	-19%	-14%	-9%	-9%
c1908	-26%	-10%	-10%	-8%	-16%	1%	2%	10%	-3%	-4%	7%	-8%
c499	-20%	-12%	-7%	-4%	-21%	-5%	-3%	5%	3%	8%	4%	3%
c1355	-21%	-9%	-1%	-2%	-21%	-3%	6%	20%	4%	10%	15%	14%
c2670	-11%	-15%	-7%	-11%	-8%	-7%	-2%	-4%	-7%	-4%	-7%	-11%
alu4	-2%	-2%	3%	4%	6%	7%	12%	18%	20%	37%	25%	9%
c3540	-1%	-2%	-1%	-3%	1%	2%	7%	14%	14%	12%	14%	-1%
c5315	-6%	-2%	-3%	-3%	-4%	0%	3%	5%	-5%	2%	3%	-11%
c7552	-8%	-4%	-4%	-7%	-3%	6%	13%	5%	18%	31%	31%	-7%
c6288	-28%	-7%	0%	-2%	6%	15%	15%	12%	46%	96%	27%	13%

Table 2: Comparison of Partition Sizes

The table above shows the effects of varying *vopt*, the optimal number of vertices (gates) per partition, for the partitioning algorithm. The baseline conditions were used for these tests but with no Shannonization recursion. Most circuits benefit from a small *vopt*, 25 or less. ALUs tend to use the least power with large partitions.

	Power	Area	Delay
c432	-20%	5%	2%
alu2	-14%	22%	-21%
c880	-22%	-2%	-10%
c1908	-26%	-16%	-3%
c499	-20%	-24%	10%
c1355	-21%	-23%	8%
c2670	-15%	-5%	-12%
alu4	4%	18%	9%
c3540	-3%	14%	-1%
c5315	-5%	-5%	-5%
c7552	-5%	5%	10%
c6288	-42%	-16%	-11%

Table 3: Optimal Partition Sizes

The table above gives results for the optimal number of vertices (gates) per partition, selected for each benchmark circuit. Again, the baseline conditions were used except that no Shannonization recursion was used. Power savings are significant, varying from 20% to 30% for many circuits. The cost in area was usually negative—the modified circuit actually used less area—but some circuits required up to 10% more area. The delay time varied from a 21% decrease to a 10% increase.

	Power	Area	Delay
c432	-19%	7%	-3%
alu2	-12%	31%	-22%
c880	-24%	-2%	-14%
c1908	-28%	-24%	-9%
c499	-27%	-35%	-4%
c1355	-27%	-36%	-1%
c2670	-17%	-12%	-19%
alu4	5%	27%	15%
c3540	-4%	20%	11%
c5315	-5%	-6%	-14%
c7552	-8%	3%	3%
c6288	-40%	-15%	-9%

Table 4: Recursive Shannonization

The table above gives results for two level recursive Shannonization. The baseline conditions were used for these tests. The power results were a bit better for the error correcting circuits and just a few percent better for most other circuits. Except for alu4, which actually required more power, the other circuits improved from 4% to 40%. Overall, the power savings median was an 18% improvement.

	Power	Area	Delay
c432	-17%	10%	0%
alu2	-12%	31%	-22%
c880	-23%	-1%	-14%
c1908	-28%	-23%	-9%
c499	-27%	-35%	-4%
c1355	-27%	-36%	-1%
c2670	-12%	-10%	-18%
alu4	5%	27%	15%
c3540	-4%	20%	11%
c5315	-8%	-5%	-11%
c7552	-6%	8%	34%
c6288	-40%	-14%	-10%

Table 5: Relaxed Area Constraint

The table above gives results with the area of the modified circuit limited to a 70% increase for each partition instead of 50%, and using baseline conditions otherwise. This constraint relaxation had little effect and in fact most circuits synthesized no differently because of it. Shannonization on a partition is aborted if the area or estimated power constraint is exceeded. Estimated power is calculated as area for shared logic and multiplexers but half of area for gates in the cofactors, which are estimated to be disabled half of the time. Since this table allows area increases of as much as 70% but leaves power increases limited to 50%, in almost all cases the Shannonization decision is determined by the power constraint so the circuit is synthesized the same as with a 50% area constraint. Only in rare instances is the circuit synthesized differently with the relaxed constraint.

<i>len (nm)</i>	Power			Area			Delay		
	32	45	65	32	45	65	32	45	65
c432	-19%	-11%	-4%	7%	7%	7%	-3%	-3%	-3%
alu2	-12%	-2%	5%	31%	31%	31%	-22%	-22%	-22%
c880	-24%	-19%	-15%	-2%	-2%	-2%	-14%	-14%	-14%
c1908	-28%	-21%	-18%	-24%	-24%	-24%	-9%	-9%	-9%
c499	-27%	-20%	-17%	-35%	-35%	-35%	-4%	-4%	-4%
c1355	-27%	-22%	-19%	-36%	-36%	-36%	-1%	-1%	-1%
c2670	-17%	-14%	-9%	-12%	-12%	-12%	-19%	-19%	-19%
alu4	5%	9%	13%	27%	27%	27%	15%	15%	15%
c3540	-4%	3%	7%	20%	20%	20%	11%	11%	11%
c5315	-5%	-3%	-1%	-6%	-6%	-6%	-14%	-14%	-14%
c7552	-8%	-3%	-1%	3%	3%	3%	3%	3%	3%
c6288	-40%	-43%	-43%	-15%	-15%	-15%	-9%	-9%	-9%

Table 6: Comparison of Technology Scales

The table above shows the effects of technology scales. The tests were conducted using baseline conditions and the 32 nm predictive technology model from 2005 and 45 and 65 nm models generated by the Nano-CMOS program using default parameters except $V_{th} = 0.16V$ and $V_{dd} = 1.0V$. These results show that the Shannonization technique becomes more important as technology scales down. This occurs because leakage power becomes the greatest form of power consumption below 45 nm, greater than dynamic power. Area does not change significantly due to Shannonization with technology scale and in this experiment, did not change at all. Delay did not change because this experiment did not account for technology scale.

<i>cycle (ns)</i>	Power				Area	Delay
	2	5	10	1000		
c432	-9%	-19%	-24%	-31%	7%	-3%
alu2	-3%	-12%	-17%	-23%	31%	-22%
c880	-19%	-24%	-26%	-30%	-2%	-14%
c1908	-22%	-28%	-31%	-37%	-24%	-9%
c499	-20%	-27%	-30%	-36%	-35%	-4%
c1355	-22%	-27%	-30%	-34%	-36%	-1%
c2670	-9%	-17%	-20%	-23%	-12%	-19%
alu4	11%	5%	3%	0%	27%	15%
c3540	2%	-4%	-8%	-14%	20%	11%
c5315	-5%	-5%	-11%	-15%	-6%	-14%
c7552	-4%	-8%	-10%	-14%	3%	3%
c6288	-44%	-40%	-37%	-26%	-15%	-9%

Table 7: Comparison of Cycle Times

The table above compares power consumption by cycle time. The tests were conducted with baseline conditions except for cycle time. The baseline cycle time is 5 ns, which is included in this table. For most circuits, power savings due to Shannonization increase with cycle time because the technique saves leakage power and not dynamic power. For the minimum cycle time, Shannonization is not effective because it introduces additional switching elements. The table above includes results for 1000 ns = 1 μ s cycle time, which approximates static power consumption.

Area and delay are not functions of cycle time.

	Execution Time (s)
c432	220.77
alu2	49.67
c880	38.78
c1908	50.11
c499	44.95
c1355	46.88
c2670	129.52
alu4	557.75
c3540	231.74
c5315	335.37
c7552	434.62
c6288	613.46

Table 8: Execution Times

The table above gives execution time of the Shannonization program, including time to run the SIS logic synthesis program. These runs were carried out using a GNU C++ compiled program run on a Red Hat Linux system with a 1.4 GHz AMD Athlon processor. Execution time varied from 39 seconds to 10 minutes. Total memory required for both programs simultaneously was about 10 MByte for the largest benchmark.

Future Work

Future work includes Shannon-aware partitioning, in which aspects of Shannon decomposition influence partitioning. At minimum, this might include combining partitions that use the same control variable. More generally, this might be a separate

stage of moving gates near partition boundaries, selecting control variables and evaluating the control variable selection criterion (fewest number of gates in the result).

A simple model of power consumption in switching circuits with supply gating transistors would make it possible to compare alternative syntheses and search for the best while the program is running, instead of waiting for a separate and slow SPICE simulation.

This program means to automate the Shannonizing logic synthesis process, but there needs to be a thorough analysis of small circuits to evaluate the program's ability to select the best partitioning and Shannonization.

This program operates on logic at the gate level. Better results might be possible by applying Shannonization of a digital design at the RTL level as well.

Conclusion

With good partitioning, selection of a control variable and extraction of shared logic, it is possible to significantly reduce power consumption in combinatorial logic, with a 18% median improvement with ISCAS-85 benchmarks at the 32 nm level. Techniques such as these will only become more important as leakage power becomes an ever greater share of power consumption, as transistor technology scales down. This program demonstrates that Shannon expansion power analysis can be an integral and valuable part of logic synthesis software.

Bibliography

- [1] E. M. Sentovich, et al., SIS: A System for Sequential Circuit Synthesis, Memorandum No. UCB/ERL M92/41, University of California, Berkeley, 1992.
- [2] S. Bhunia, N. Banerjee, Q. Chen, H. Mahmoodi, K. Roy. A novel synthesis approach for active leakage power reduction using dynamic supply gating. Design Automation Conference, 2005. Proceedings. 42nd, pp. 479-484, June 2005.
- [3] G. Karypis, Multilevel Hypergraph Partitioning, University of Minnesota, Minneapolis.
- [4] Y. Saab, A New Effective And Efficient Multi-Level Partitioning Algorithm, University of Missouri-Columbia.
- [5] J. W. Tschanz, S. G. Narendra, Y. Ye, B. A. Bloechel, S. Borkar, V. De. Dynamic Sleep Transistor and Body Bias for Active Leakage Power Control of Microprocessors. IEEE Journal of Solid-State Circuits, Vol. 38, No. 11, November 2003.
- [6] D. Sylvester and H. Kaul. Power-Driven Challenges in Nanometer Design. IEEE Design and Test of Computers, Nov.-Dec. 2001.
- [7] M. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," IEEE Design and Test, vol. 16, no. 3, pp. 72-80, July-Sept. 1999.
- [8] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, C. Hu, "New paradigm of predictive MOSFET and interconnect modeling for early circuit design," pp. 201-204, CICC, 2000.
- [9] Y. Cao et al. <http://www.eas.asu.edu/~ptm/>, University of Arizona, Tempe.