ILOC MODULAR AND VECTOR ABSTRACTION

William Bricken
May 2003

CONTENTS

ICONIC LOGIC THEORY	
ABSTRACTION	FIGURES I - IV
ABSTRACTION DURING DESIGN	FIGURES V - IX
Top-down Abstraction	
Bottom-Up Abstraction	
ABSTRACTION OF FUNCTIONAL MODULES	FIGURES X - XVI
Low-level Modularization	
MODULARIZATION TO ASSIST PLACEMENT AND ROUTING	FIGURES XVII - XXIV
Sequential Modularization	
Parallel Modularization	
Output Subcircuit Modularization	
PARAMETRIC GENERATION OF CIRCUITS	FIGURES XXV - XXXI
VECTORIZATION	FIGURES XXXII - ILIV
Vector Modularization	
Matrix Abstraction	

SUMMARY

Iconic logic removes redundancy, thus exposing the structural patterns in any circuit. A structural pattern is an abstraction that condenses many circuit components into a single component. There are several types of structural patterns, all of which can be abstracted. Modular, generator and vector abstraction are described herein. Hierarchical organization techniques provide methods of simplifying the design and comprehension of complex circuits. Abstraction provides generic design components and helps to manage the complexity of a design.

EXECUTIVE OVERVIEW

Look at the ILOC body in Figure XXXIII. It has eight lines with similar parens patterns. Whenever there is a similar parens pattern, it is possible to construct an ILOC abstraction. This memo describes many types of abstractions, the identification and effective manipulation of most of them are unique to ILOC tools. These abstractions help to solve many difficult design problems, such as modular composition and reuse of design elements, identification and control of design hierarchies, technology mapping to library cells, and placement and routing of logic into limited physical resources. The ILOC tools can be applied to accelerate synthesis and transformation of circuit structures, and in general to provide powerful capabilities for the management of large, complex and difficult to understand circuit structures.

PRESENTATION TECHNIQUE

This memo includes many examples using ILOC internal format and parens notation. Although study of these forms is necessary to understand their internal mechanisms, a brief examination of the forms will reveal the parens patterns that guide the ILOC internal algorithms. A cursory scan of the examples is sufficient to convey the fundamental idea the circuits represented by Iconic Logic containers consist of easily identifiable repetitive patterns. that can be mapped into reusable design elements and repetitive silicon architectures with ease.

ICONIC LOGIC THEORY

ILOC data structures can expose structural patterns in circuits because of three unique features of Iconic Logic:

-- gates and wiring are a single concept
-- structural patterns are constructed from a single component type
-- iconic forms concurrently represent many different data structures

The container is the only structural entity in ILOC. As a boundary between spaces, a container both separates and connects. The connectivity aspect -- both spaces share a common boundary -- is interpreted as wiring. The separation aspect -- neither space is confused with the other -- is interpreted as logic.

Pattern-matching is simply aligning the depth and breadth of the homogeneous containment relations. This permits both pattern-matching of identical forms and pattern-matching of forms that are structurally the same but contain different sub-structures, i.e. abstract features.

ABSTRACTION

Abstraction refers to a body of techniques for constructing larger basic structures than are included in a gate level description. Abstraction comes in many varieties, including

-- *cell abstraction*: a pattern is identified and constructed as an individual gate or library cell

-- *module abstraction*: a pattern is identified and constructed in a separate functional grouping while cells provide input and output binding values for the module

-- *input abstraction*: symmetries across inputs are generalized

-- *generator abstraction*: functions are generalized to apply to any bit-width

-- *vector abstraction*: collections of input signals being processed by identical functions are bundled into collections of signals with bit-widths greater than one

-- *partition abstraction*: logic and routing clusters are structured by limited parametric templates that are used for mapping into constrained physical resources.

Cell structure abstraction uses specialized cell-forms with a pre-defined structure. Cell structure abstraction primarily addresses common lower-level patterns such as XOR and MUX. In general, cell-form pattern abstraction is usually used for network abbreviation to control complexity, for hierarchical construction and decomposition of circuit functionality, for identification of time/space trade-offs in circuit structures, and for technology mapping into cell libraries.

In contrast, *module abstraction* constructs a library element as a reusable component, abstracting structure into a separate module body and using i/o ports to connect inputs and outputs of the module to the body of a larger circuit.

Figure I shows the ILOC internal form of a *library element*, or an *abstract module*. The main body of any circuit is in the exact same form, making any circuit a potential module in an even larger context. Figure II shows the format for linking a module into the body of a larger circuit. Figure III shows a simple example of the ILOC modular cell capabilities, in which a 2NOR cell has been abstracted. Modules can be nested within one another. Figure IV shows a three-level example of nested modules.

ABSTRACTION DURING DESIGN

Abstraction is mandatory as designs grow large, since human designers find it exceedingly difficult to extract information from logic networks containing thousands (or more) gates. To accommodate this restriction, design itself has embodied modular principles. Thus modular abstraction already exists as a top-down design practice.

Top-down Abstraction

Nested modules provide an unlimited capacity for hierarchical abstraction and organization of a circuit design. By defining module templates in a top-down manner, a designer can rough-out the architecture of a design without having to specify the details. This is common practice during the specification phases of design. By assembling existing functional modules for specific behavioral components such as adders, comparators, buffers, counters, and the like, a designer can fill in certain aspects of an evolving design while leaving other aspects for later refinement. Thus nested modules provide valuable design tools that assist and facilitate designers during the conceptualization and specification phases of design, they provide organizational tools for the management of complexity of designs, and they enable to modular composition and reuse of design elements.

Figure V shows three stages in the top-down specification of a microprocessor. This figure serves as an example of the design process, and is not intended to illustrate a new design. At first, a designer may draw a very abstract functional block diagram, indicating the highest level units and the channels of communication flow. These functional blocks may then be articulated with some important concepts, such as bit-widths and types of data flow. As the design elaborates, more and more information may be filled in, some of it just placeholders, some it well understood, and perhaps some of it experimental, awaiting more information.

Similarly, Figure VI shows that the ILOC format can begin in a quite rough form, with details lacking and only functional modules present. Given sufficient guidance, ILOC can generate the abstract templates automatically and interactively, serving as a design drawing board, very much like software coding environments provide both templates and structural management for programming concepts. Figure VII shows more detail of the ILOC format being filled in, corresponding to the middle diagram in Figure V. Here emphasis is on the control cycles and steps of the FSM component, and the communication between the components. Instructions, or functionality, is expressed in module cells, while data paths are expressed in the input and output binding of the modules. Figure VIII shows yet another level of top-down refinement, specifying more details about the data flow and instructions between the modules.

This example is not one of hardware design since it illustrates ILOC capabilities for design prior to hardware specification. After the functionality has been defined at the conceptual and functional levels, a hardware engineer would then ask how to achieve this functionality in silicon. Figure IX shows the ILOC modular format for the cell-level hardware description of a functional microprocessor, for comparison. These WOTAN CPU modules have been identified dynamically by ILOC. (The WOTAN design, by Nicholas Wirth, is intentionally highly modular as well.) The WOTAN

microprocessor description is annotated by the number of each module type it incorporates when expanded.

Thus ILOC can provide methods and support for the initial stages of the design cycle, as well as logic synthesis after the gate-level design is complete.

Bottom-Up Abstraction

The methods of abstraction applied to parens forms apply to ILOC body formats in general. The ILOC format, however, allows more diverse types of symmetries to be abstracted than do parens forms. In particular, structural symmetries within parens forms must also be logical symmetries, since parens patterns are logic patterns. In the ILOC circuit body, two cell-forms can be symmetrical but not connected logically. Patterns across cells can identify identical functions that can be put into ILOC module format, they can identify vectors of signals that travel together between logical structures, they can identify opportunities for structure sharing within the circuit, and they can identify glue logic subcircuits with no inherent structure.

What current tools do not provide is bottom-up abstraction, the ability to identify modular patterns in a circuit that have not necessarily been put there during design. The primary advantage of bottom-up abstraction is that designers can identify available trade-offs between circuit area and circuit delay. Another advantage of bottom-up abstraction is that modular components can be partitioned parametrically. Circuits can be sliced up in different ways, some of which may be more desirable for a particular design context or a particular resource constraint. Such re-partitioning is often necessary during design placement into a silicon substrate. Not only does the abstraction aspect of ILOC identify available modular components, it can move across different modular groupings, customizing the level and degree of abstraction.

Although many tools can take a hierarchical design top-down and expand it into more fine-grain elements, the top-down design must first be defined and provided by a designer. ILOC can take netlist descriptions and construct bottom-up the abstract modules that form the circuit. This enhances ease of maintenance and of redesign. As well, poorly optimized circuits can be decomposed, reoptimized and reorganized automatically by the tools and methods incorporated in ILOC. Further, as illustrated by the following figures, ILOC can provide a diversity of decomposition and abstraction perspectives that permit a designer to explore specific aspects of the circuit structure such as logically necessary paths, wiring and routing demands, critical paths, and other design criteria. It should be noted that the small circuits used in these examples do not imply a limitation to the scalability of the ILOC techniques or their capacity to be fully automated.

ABSTRACTION OF FUNCTIONAL MODULES

It can be useful to identify logic subgraphs for abstraction that have the same functional behavior but have different inputs. Thus in the case of MUX abstraction, rather than keeping each MUX within a separate cell, the abstract MUX pattern could be created as a module, and each MUX instance within the circuit could refer to the module. By including all parens patterns and ignoring the various inputs within those patterns, functional abstraction can be used for structure sharing. For example, should a large ALU form occur four times within a given circuit, and should area be over constrained, then the design can be configured so that the same say two ALU forms could be used for different sets of inputs. By converting parallel processes to sequential processes, time is swapped for area.

Figure X shows a functional abstraction of four cells, all having the same parens structure. Using simply pattern-matching, the four common parens structures are identified. The first, or seed, pattern could be designated by the user interactively, or it could be selected automatically.

One important aspect of modularization is that all optimization tools within ILOC can be applied within a module body, thus changing all instances of the module at one time.

Low-level Modularization

Figure XI shows a 4-bit magnitude comparator after a number of rearrangement transformations. The designer may wish to examine the smallest gate structures that are currently present in the circuit body. By selecting low-level abstraction, ILOC will identify parens patterns at the lowest level of structuring that are not simple logic gates. These patterns are visually apparent. The XOR module consists of eight structurally similar patterns, with pairs of similar patterns sharing identical atoms. The 3NOR module consists of four structurally similar patterns. Since the AND patterns in cells 9 through 16 are simple gates, as are the OR patterns in cells 21 and 22, these are expanded to form the OR-AND module.

For clarity, in this example, each of the constructed modules is shown separately in Figures XII, XIII, and XIV. Then they are composed in Figure XV. Automated modularization would construct the three modules concurrently. However, a designer could elect to constrain the method to construct only one module at a time. A designer may also elect to specify specific cells to be modularized, further constraining the scope of application of the method. Figure XVI contains the circuit schematic corresponding to Figure XV, showing abstracted function blocks in place of module component cells. By abstracting gates into functional blocks, the structure of the circuit becomes clearer, and circuit complexity becomes easier to handle both visually and in the context of design.

The necessity for design hierarchy is a recognized problem for designers. EDA tools tend to provide graphic support, but not automated identification and construction of modules. ILOC addresses this problem directly. In addition, ILOC provides multiple levels of design entry, permitting a designer to access the design and the automated design tools conveniently at any level of detail.

MODULARIZATION TO ASSIST PLACEMENT AND ROUTING

The modularization process can be applied to a circuit structure with the design goal of easing placement and routing into physical resources without degrading the gains from logic synthesis. ILOC thus contributes to the solution of an outstanding problem in placement of logic and management of physical routing resources.

By controlling expansion and formation of ILOC cells, both the logic resources and the routing resources required by the cell-form can be controlled at any grain-size. For the smallest grain-size, every cell-form consists of a single container and a contained group of atoms. For the largest grain-size, there is only one cell for each output and that cell contains the entire transitive fan-in of the output. All of the techniques of ILOC rearrangement can be applied to placement and routing. In the case of a fixed logic resource, that resource is a pattern template to be matched. In the case of reconfigurable but fixed resources, the pattern template may match some criterion other than logic placement, for example, grouping into collections of four atoms for 4LUT mapping. In all cases, a minimized parens form that is as deeply nested as possible will require the minimum routing resources, while a flattened two-level parens form will require the maximum routing resources.

Herein, a more abstract notion of routing is described. ILOC identifies *necessary* routing required to achieve a specific functional performance within specific structural limits, particularly those limits defined by a constant length critical path, independent of hardware resources. ILOC abstraction and modularization techniques characterize the functionality of the circuit logic in such a way that physical placement of any form of the circuit must address at least what is dictated by the parens form. For example, should a parens form representing a specific logical functionality require a minimum of 12 containers, then for simple logic gates, a minimum of 12 wires will be required regardless of how the functionality is expressed structurally. This observation gains strength when combined with abstraction. Given that a logic element can accommodate a particular parens configuration, then that configuration can be treated like an atom, a black box for which the internals don't matter to the placement or routing of the circuit. In such a case, the number of containers in the context of the abstracted form still defines the minimum routing required for that unit of

functionality. ILOC incorporates a capability to configure any type of grouping of forms hierarchically using abstraction.

Figure XVII shows the 4-bit magnitude comparator partitioned to emphasize routing between functional modules. Figure XVIII shows the schematic for the circuit abstracted to emphasize routing.

Sequential Modularization

Figure XIX shows a modular decomposition of the 4-bit magnitude comparator to emphasize sequential routing requirements, particularly the critical path through the circuit. The modularization criterion is to construct modules with *shallow* bodies. Cells are expanded under the constraint of no more than two levels of nesting. Figure XX shows the accompanying schematic. Two sequential forms dominate the circuit structure. The chain propagating through the XOR-3NOR blocks defines the longest sequential path delay, ending with the EqualTo output. The tangle of wires connecting the critical path fan-out to the OR-AND blocks provide information to the GreaterThan and LessThan outputs. Upon closer inspection, each XOR-3NOR block provides one signal to each of the OR-AND blocks separately, and one signal that is shared by both blocks. One enable enters each of the OR-AND blocks; interestingly, the EqualTo enable progresses through the critical path chain, identifying a design flaw in this structure.

Parallel Modularization

Just as sequential structure can be identified by forming shallow modules, parallel structure can be identified by generating the *deepest* possible recurrent structures. Figure XXI shows the ILOC format of the 4-bit magnitude comparator based on deep modules. This structure is formed by expanding cells based on whether or not the cell-form contributes to deeper parens patterns. The net result is to place the maximum amount of sequential logic into module blocks, leaving only routing that propagates signals in parallel. Figure XXII shows the accompanying schematic. The ability to partition sequential and parallel components of a logic structure is unique to ILOC.

Output Subcircuit Modularization

ILOC modularization can generate modules to abstract and separate structure that is unique to each output from structure that is common to each output in a multiple output circuit. Again, such a decomposition highlights required routing resources, in this case those needed by each output independently. Figure XXIII shows the 4-bit magnitude comparator abstracting each output subcircuit This particular abstraction identifies a structural similarity

between the GreaterThan and the LessThan circuitry, also showing that the EqualTo output is independent of the comparative outputs. Figure XXIV shows the schematic that highlights independent output structures. Close inspection reveals that signals coming from the eight two-input NOR gates all enter the nine-input OR gate. This suggests that the EqualTo magnitude comparison can be implemented separately from the GreaterThan and LessThan magnitude comparisons.

PARAMETRIC GENERATION OF CIRCUITS

The specification of design constraints such as critical path length, fan-out, and wiring complexity can be used to guide the automated generation of candidate circuits for further evaluation and analysis. A common parameter that guides the size and complexity of circuitry is bit-width. ILOC incorporates abstract templates that can be used to construct specific functions for particular bit-widths. This capability allows the design of generic components that can be generated to fit any width of parallel binary inputs. An N-bit adder, for example, can be instantiated in one part of a circuit that uses only 4-bit data as a 4-bit adder, while being instantiated as a 16-bit adder in other parts that use 16-bit data streams.

Bit-width parameters can be used to redesign functions for wider input data structures, for specialization of function size in order to conserve resources, for ease of behavioral specification, for customization of functions to particular input values, and for a diversity of other design and functional proposes.

An example of bit-width configuration combined with other specified design parameters is a decimal-to-binary converter. In order to accommodate decimals with 10 input varieties, a binary system needs four bits that can be configured into 16 (2^4) varieties. However, the six of these binary configurations that would represent the non-existent decimal numbers 10 through 15 are never used. Thus the input to a binary-to-decimal converter is biased. The lowest bits occur probabilistically 50% as 0 and 50% as 1. The highest bit, which covers the decimal numbers 8 through 15 when set to 1, is most often a 0. Specifically four times out of five a decimal number will be 0 through 7, and one time out of five it will be 8 or 9. Thus the circuit can be optimized to take this biasing into account. Biased input is quite common in many applications, however current design tools enforce a 50/50 expectation for input values. Current synthesis techniques address this biasing as don't care values, attempting to optimize logic by analyzing the possible paths through a circuit. ILOC addresses don't care minimization directly in the parens format, not requiring specialized analytic techniques.

The magnitude comparator serves as an example of parametric specification of functionality in ILOC format. Figure XXV presents the ILOC format for a magnitude comparator of any bit-width, without enables. The functional

circuit generator described is intended for specialized bit-widths, such as a 5-bit comparator that would be wasteful if composed of two 4-bit circuits. Also, it is possible to construct functions with slower sequential architectures, and then manipulate their structure using ILOC optimization techniques to construct versions of the cascaded functions that are faster.

The magnitude comparator generator expressed in ILOC format in Figure XXV consists of two components, a base case and a generator case. The base case initializes the generator, providing a 1-bit comparator. As such, the base case 1-bit comparator is inefficient and would be significantly reduced by structure sharing if only one bit was to be compared. Essentially a 1-bit comparator is an XOR gate, but the bare XOR form provides no suggestion as to how to generalize it to be a 2-bit comparator. The generator components in the figure are equivalent, one is expressed as an iterated method and one as a recursive method. The recursive component is a preferred method since it is both more succinct and more efficient. The recursive component expands directly by substitution, while the iterated component requires an independent engine to conduct the iterative expansion. Figure XXVI shows an abstract circuit schematic for the base and recursive generators described in Figure XXV. The schematic can be expanded by direct substitution of generated circuits that have one fewer bits.

In recursive generation of the comparator circuit, the 1-bit template circuit serves as the starting format. The 2-bit comparator is constructed by embedding the 1-bit comparator in the abstract block provided by the generation template for the circuit. Then to construct a 3-bit comparator, the 2-bit comparator is again embedded into the abstract block provided by the generator template. An N-bit comparator is generated by N embeddings.

Figure XXVII provides examples of the ILOC format generated comparator circuits for two, three and four bits to illustrate the expansion process. Figure XXVIII provides the same generated comparator circuits for one, two and three bits in schematic form, showing the substitution process. It should be noted that the generation and substitution process construct circuit designs that are highly inefficient. Bit-width generation is naturally followed by ILOC optimization, which then redesigns the circuit to be both efficient and minimal.

Similar to Figure XXVI, Figure XXIX shows the ILOC generator format for the magnitude comparator circuit, but with the enables included. The figure is an example of a general recursive generator with non-recursive components.

For generator functions, it is easiest to use the non-optimized functions during the generation phase, and then to optimize the final generated function, since the greatest diversity of design choices is maintained by the highly regular structure of the non-optimized generated format. The regularity maintained in non-optimized base case forms permits easier ganging and cascading of components to form larger functional units during technology

mapping. Thus, the reduction benefits of optimizing a circuit generator can be postponed by design until the context of the generated function is more defined.

Figure XXX shows a non-optimized 8-bit adder generated by ILOC, while Figure XXXI shows the optimized version of the same adder. The repeated structural pattern output by the generator is apparent in Figure XXX. The repeated pattern of structural organization of an optimized adder is apparent in Figure XXXI. In the case of adders, optimization is not significantly different than non-optimized generation, rather optimization is primarily an organizing principle.

VECTORIZATION

The ILOC generator format permits replication of functionality across varying bit-widths. The ILOC vectorization tools permit wires of varying bit-widths to be abstracted into a N-bit wire. Combining modularization, vectorization, and function generation creates a set of design tools with which a designer can compose, analyze, and explore a complete circuit abstractly without regard to specific bit-width.

One important aspect of vectorization is that ILOC transformations can be applied to the abstract vector form, thus changing all instances of the vector at one time. All ILOC tools are equally applicable to vectors, permitting efficient transformation of groups of signals over a given functionality.

Some abstractions are particular to a specific functionality or circuit, but are replicated due to bus structures and bit-widths. In vectorization, these are identified by matching containment structures across ILOC cells, similar to the functional abstraction method. *Vectorization* is the combination of functional abstraction and bit-width abstraction. Figure XXXII shows a simple example.

In Figure XXXII, an ILOC body fragment has four cells that are functionally equivalent. In functional abstraction, these four cells could be converted into four instances of a single module that abstracts the common functionality. For comparison, the functional abstraction is also shown in Figure XXXII. To abstract the communality of the input-output relations in the example, two vectors are constructed, the input vector and the output vector. These vectors are ordered lists, each input vector item corresponding to an output vector item with the same vector index. The vectors are then substituted for the set of cells they abstract. In the example, four cells that share the same functional relation are replaced by one vectorized cell that represents all four cells in one vector abstraction.

Function modularization and vectorization are similar methods that achieve similar results. Fundamentally they differ in the way abstracted forms are stored and accessed. In modularization, the function template serves as an abstracted component that can be referenced with particular bindings any number of times. Thus functional abstraction is useful when a fixed library structure is mapped during technology mapping, and when a stable modular component has been identified. To transform functional abstractions, it is necessary to expand them back into the ILOC body, at which time the modular form may be lost due to transformation. Some instances of a modular component may be expanded without abandoning the component throughout the ILOC body. Functional modularization reifies a functional form while keeping i/o bindings flexible

In contrast, vectorization applies to a particular set of matching patterns and usually is not extended when other pattern instances are identified. Thus vectorization applies to a fixed number of instances. The vectorized functional form remains within the ILOC body as a single vector cell, so that transformations on the vector function do not require a binding list to be disassembled. In vector abstraction, an input/output template serves as the abstracted component that is referenced by a given functional structure a specific number of times. Thus vector abstraction is useful when the same functionality is applied to a collection of inputs and results in an aligned collection of outputs. It is useful when a stable i/o bundle can be identified. Instances of a vector output can be referenced independently, in effect decomposing the vector after a function is applied. Vector abstraction reifies an i/o structure while keeping functional form flexible.

Figure XXXIII shows another ILOC vector format example in which eight three-input MUX functions are combined to construct a vector MUX with sixteen inputs and eight outputs, all using the same select signal. The vectorized body consists of three vector binding cells and one functional cell. Figure XXXIV shows the accompanying schematic view of a vectorized ILOC format. The case of N identical functions with different inputs can be vectorized to a single vector function with a vector input of width N. Note that the individual components of the input vector do not become functionally combined. The vector components are not inputs to the same single function, rather they are separate inputs to separate instances of the same function module. Thus there are two types of bit-width abstraction, functional bit-abstraction and vector bit-abstraction.

Vectorization is of particular value whenever the same vector occurs in different functional relations throughout the circuit. Figures XXXVa and XXXVb show an example ILOC format of an 8-bit sequential multiplier. The 82 ILOC cells represent 16 registers and 169 conventional 2-input gates. Figure XXXVI shows the ILOC vector format of the same multiplier. The vectors in the ILOC body are annotated by their size. The vectorization clearly shows that 16 bits are progressing as vectors through only four functional

transformations, labeled as vector cells V3, V4c, =V5, and =V6b in the ILOC vector body.

Vector cell-labels follow the same conventions as do regular cell-labels, so that the exclamation point, for example, identifies a vector of registers, and the equal sign identifies a vector of IFF/XNOR functions. In the example, the end conditions have not be generalized, so that several vectors overlap in the middle, but differ in the first and last vector indices. Thus vectorization permits exacting control over bussing and routing resources.

Figure XXXVII shows the ILOC vector body with the vector relationships annotated. In this circuit, two vectors, V2 and V4, are initialized by binding one vector component to a ground value. Two vectors, V4 and V6, are shifted so that the output of a particular component is used as the input to an adjacent component. These structures characterize a one-bit sequential processor. The Figure also shows a container-based transformation of vectors, expanding vectors V3 and =V5 into the cell-bodies of other vector cells. The expansion, deletion-reduction, and rearrangement processes are identical to those used for transformations with primary cell inputs.

Figures XXXVIII through XLIV show a large vector formatted circuit, I7 from the MCNC benchmarks, that has 199 inputs and 67 outputs. I7 is composed of 331 conventional gates, most of them having 4 and 5 inputs, an equivalent of just over 900 conventional two-input gates.

The ILOC body condenses to 31 vectors of size 2, 4, and 28, as shown in Figure XXXVIII. The ILOC vector body in Figure XXXIX shows that the circuit consists of six separable subcircuits. The first subcircuit is a one bit combination of various enable signals, and fourth subcircuit is a two-bit combination of enables. The other four subcircuits are quite similar, differing primarily in bit-widths of 4 and 28 bits. Figure XL shows the partitioning of the ILOC body with regard to bit-width processing. In two cases, the sequence of labels is divided into two sections. This implies nothing more than non-contiguous cells for the purpose of vector abstraction. Figure XLI shows the vector body, expanded so that each vector output is defined solely in terms of a single parens vector form.

Vector Modularization

One important aspect of management of circuit size and complexity is to be able to express the circuit functionality succinctly. I7 requires over 100 *pages* to describe in EDIF 2 0 0 format. The ILOC vector body achieves a factor of 100 reduction over conventional formats such as EDIF, requiring a single page to describe the functionality of the I7 circuit. Additionally, substantive replication of structure is still observable in Figure XLI.

Figure XLII shows the application of functional modularization to the ILOC vector body for I7. A single vector module reduces the functional body to six lines, one line for each independent output. In the vector modularization, vectors are bound to the module input labels, and to the four module binding cells. These vectors are all of the same size for a particular module cell, but do not have to be of a particular size across module cells. In the example, both 4-bit and 28-bit vectors bind to the same module structure. Transformation can be applied to the module body, even though different bindings may be to different vector lengths. Figure XLII also shows the application of cell abstraction to the vector cells with the vector module. In particular, one XNOR structure and two MUX structures are identified and abstracted into patterns. Since these are vector patterns, they require care in interpretation. For example =2 refers to four different sets of XOR gates, one for each index in the vector Va. The size of the vector Va is indeterminate when localized in the module, so that its size is defined by the binding cells V91- through V94-, In the case of binding cell V91-, Va is bound to Vi11. In turn, the Figure identifies Vi11 as a vector that has 28 indices. Thus, =2 in the vector-module expansion of V91- represents 28 separate XNOR gates, each receiving the same input ac, and one other input from Vi11. Succinctly, =2 tests 28 inputs signals for equality to ac. The rest of the module is also easy to read. For binding cell V91-, cell >3 is 28 MUX gates, all using the input ac as a selector between 28 pairs of input signals. Cell >1 uses the input ad to steer to output either the vector of equality comparisons, or the vector of selected inputs from cell >3. Summarizing, the module selects one of two inputs, tests a different input against the same selector, and then selects one of those results using a second selector. This is replicated for 28 input sets of three.

Matrix Abstraction

Since vector cells can be converted into modules, they can also be converted into higher rank vectors, in effect constructing a matrix abstraction of the circuit. Figure XLIII shows the matrix abstraction of I7. Each new vector consists of other vectors. Here the vector components of a matrix cell are not the same size, they are merely an ordered list of ordered lists. Consistency of size is maintained by each cell in the ILOC matrix body. In the matrix abstraction of I7 shown in Figure XLIII, the ILOC body reduces to seven cells, each consisting of a one line parens cell-form. Six cells provide output in the form of enable signals combined with the central functionality of I7, while one single matrix parens form consisting of 11 containers and 5 different atoms describes the entire central functionality of the circuit. This matrix cell can be transformed using any applicable parens optimization, rearrangement or restructuring rules.

Similar to Figure XLII, the ILOC *matrix* body of Figure XLIII is further abstracted using cell abstraction on matrix cells. The two MUX structures

and the one XNOR structure are again identified, this time referring to a 4x28 matrix of single MUX and XNOR gates.

These abstraction capabilities do more that make large circuits tractable for human designers, they also specify quite exactly how placement and routing can be optimized, since the abstractions show exactly how much fan-out and distance there is between signal bundles, as well identifying the places that the circuit can be most economically partitioned for resource mapping.

To illustrate the placement and routing information, Figure XLIV shows an abstract matrix schematic of the I7 circuit. The gates in this schematic cannot be interpreted as conventional gates since some of their inputs are vectors and vectors of vectors. When a wide bit-width wire enters a gate in this schematic, it is to be interpreted as the gate being replicated a number of times to match the bit-width of the input. Within the schematic, the M4 matrix is decomposed into 4 vectors, indicating that the bit-line bundles split at that point.

The schematic is a hybrid of conventional logic functionality and abstract functionality, while never loosing a precise mapping to the physical circuit it represents. ILOC vector and matrix tools provide top-down block-like abstraction without enforcing blocks that obscure the actual logical transformations of the particular circuit. These capabilities also apply automated partitioning, placing and routing of logical functionality, addressing what is considered to be one of the most difficult issues in circuit actualization.

An important feature of matrix abstraction is that is illustrates general ILOC capabilities for hierarchical bottom-up abstraction. Should repetitive structure show up in the matrix format, it too could be vectorized to form a rank three matrix. Similarly, during top-down design, each abstraction level can be articulated as an abstract vector slice, or as a module component. Thus ILOC includes powerful methods for the abstraction and abstract transformation of circuit components, tools that can be applied to accelerate synthesis and transformation of circuit structures, enhance design capabilities, provide hierarchical top-down and bottom-up design capabilities, and in general provide management of large, complex, and difficult to understand circuit structures.

Figure I: The Abstraction Module

```
((circuit-name circuit-information)

 ((main main-information)
  (main-inputs)
  (main-outputs)
  (main-parens-cells
   (module-cell module-name  input-binding  output-binding)
   ...))

  ((module-name module-information)
   (module-inputs)
   (module-outputs)
   (module-body))
  ...
  additional-modules  )
```

Figure II: The Module Linking Format

Module-cell in main ILOC body

```
(ILOC-cell-label
    module-name
    ((module-input-label ILOC-cell-label) ...)
    ((module-output-label ILOC-cell-label) ...))
```

Module local input and output labels

```
((module-label)
  ((module-input-label module-input-value) ...)
  ((module-output-label local-module-cell-label) ...)
  module-body)
```

Figure III: An Example of an ILOC Module Serving as a Library Element, the 2/3 Majority Circuit

ILOC format with no library modules

```
((2/3-majority)
 ((main)
  ((a unk)(b unk)(c unk))
  ((oa 0))
  ((0  (1 2 3)   )
   (1  (a b)   )
   (2  (a c)   )
   (3  (b c)   )  ))
```

ILOC format with an abstracted library module

```
((2/3-majority)
 ((main)
  ((a unk)(b unk)(c unk))
  ((oa 0))
  ((0  (1-0 2-0 3-0)   )
   (1-  2nor  ((i1 a)(i2 b))  ((o0 1-0))   )
   (2-  2nor  ((i1 a)(i2 c))  ((o0 2-0))   )
   (3-  2nor  ((i1 b)(i2 c))  ((o0 3-0))   )
  ((2nor)
   ((i1 unk)(i2 unk))
   ((o0 0))
   ((0 (i1 i2)   )  ))  )
```

ILOC format with library module expanded

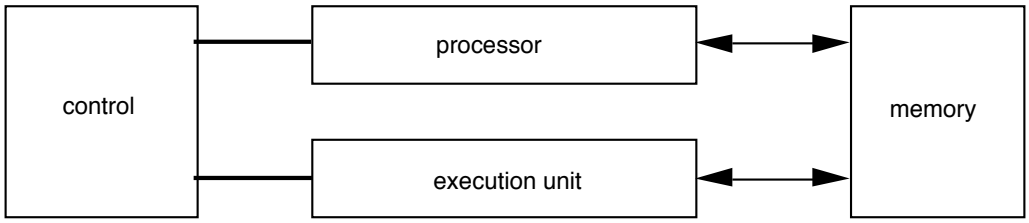
```
((2/3-majority)
 ((main)
  ((a unk)(b unk)(c unk))
  ((oa 0))
  ((0    (1-0 2-0 3-0)   )
   (1-0  (a b)   )
   (2-0  (a c)   )
   (3-0  (b c)   )  ))
```

Figure IV: An Example of Nested Modules

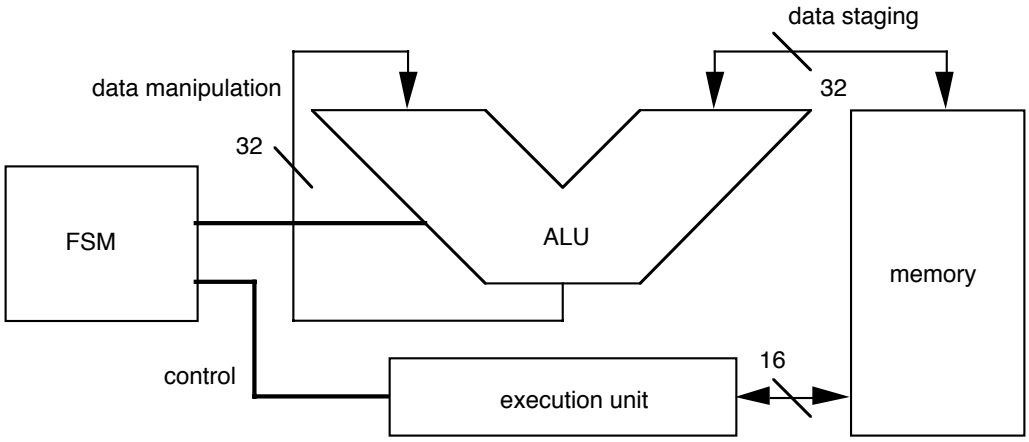
```
((nested-module-example)

 ((reg)
  ((@ unk)(b unk)(c unk)(d unk)(e unk))
  ((oa 1-0)(ob 2-0))
  ((1-    m2  ((a1 c)(b1 3))  ((oa1 1-0))   )
   (2-    m2  ((a1 d)(b1 3))  ((oa1 2-0))   )
   (3     ((@)(b (e)))   )  ))

 ((m2)
  ((a1 unk)(b1 unk))
  ((oa1 0))
  ((0     (1-0 (a1 (b1)))   )
   (1-    ff-en  ((a2 nil)(b2 b1)(@ @))  ((oa2 1-0))   )  ))

 ((ff-en)
  ((a2 unk) (b2 unk) (@ unk))
  ((oa2 0))
  ((0     (1 (2 (@)))   )
   (1     (((a2)(b2)))   )
   (2     (b2 (0))   )  ))  )
```

Figure V: An Example of a Top-down Design

Rough block diagram



More details I



More details II

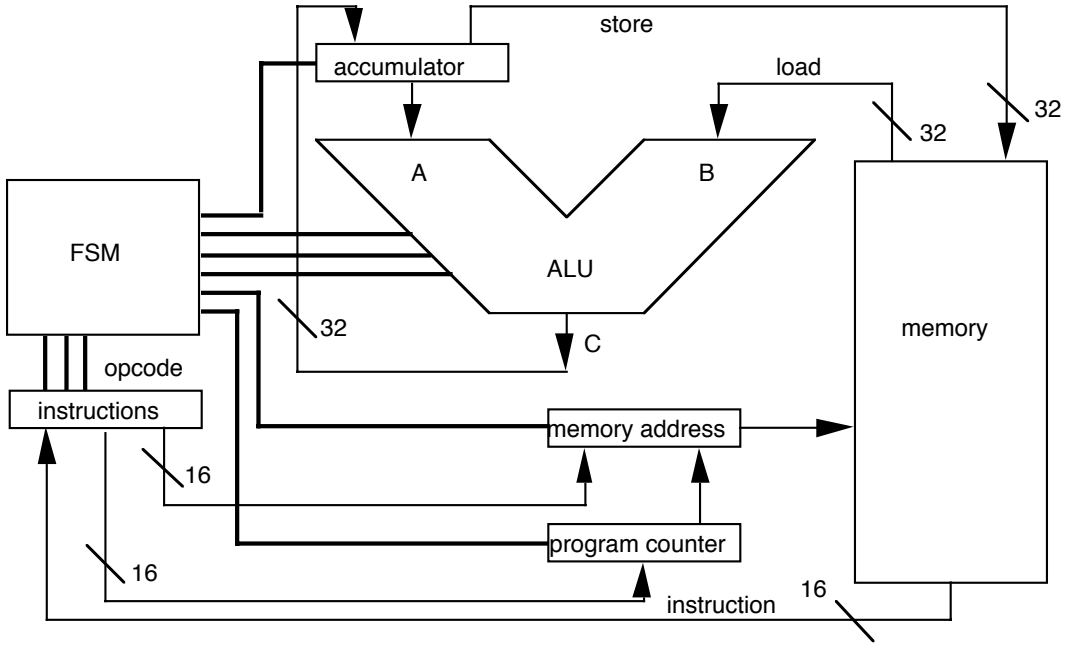


Figure VI: An Example of a Module Template for the Top-down Design in Figure V, Rough Block Diagram

```
((design)
 ((main from-the-perspective-of-the-memory))
 ((proc-in unk)(exe-in1..32 unk)))
 ((proc-out 2)(exe-out 3))
 ((1-  control
        (())
        ((proc-out <?>)(exe-out <?>))   )
  (2-  processor
        ((control-in <?>)(memory-in <?>))
        ((memory-out <?>))   )
  (3-  execution-unit
        ((control-in <?>)(memory-in <?>))
        ((memory-out <?>))   )

((FSM-control)
 (())
 ((proc-out 0)(exe-out 1))
 ((0   <?>)
  (1   <?>  )  )  )

((ALU-processing)
 ((control-in unk)(memory-in <?>))
 ((memory-out 0))
 ((0 <?>)   )

((execution-unit)
 ((control-in unk)(memory-in <?>))
 ((memory-out 0))
 ((0 <?>)   )
```

Figure VII: An Example of a Module Template for the Top-down Design in Figure V, More Details I

```
((design)
 ((main from-the-perspective-of-the-memory))
 ((READ-address1..16 unk)(STORE-in1..32 unk)))
 ((INSTRUCTION-out1..16 READ)(LOAD-out1..32 READ))
 ((1FSM-  FSM-control
         ((RESETin unk)(new-instruction INSTRUCTION-out1..16))
         ((FETCH)(EXECUTE))   )
  (2ALU-  ALU-processing
         ((Ain  2ALU-cout1.32)(Bin  LOAD-out1)(OPin  INSTRUCTION-out1..16))
         ((Cout  2ALU-Ain1.32)(Aout STORE-in1..32))   )
  (3EXU-  execution-unit
         ((rin1..16  INSTRUCTION-out1..16))
         ((rout1..16  READ-address1..16)   )

((FSM-control)
 ((RESETin unk)(new-instruction unk))
 ((FETCH 2)(EXECUTE 7))
 ((0   RESETin)
  (1   RESET:  when 0 do 2 )
  (2   FETCH:  read-memory)
  (3   when READ do 4)
  (4   EXECUTE:  <op> -> ALU   )  )  )

((ALU-processing)
 ((Ain unk)(Bin unk)(OPin unk))
 ((Cout 0)(Aout ?))
 ((0 <A OPin B>)   )

((execution-unit)
 ((rin1..16 unk))
 ((rout1..16 0)(mout1..16 ?))
 ((0 rin1..16   )  )  )
```

Figure VIII: An Example of a Module Template for the Top-down Design in Figure V, More Details II

```
((design)
 ((main from-the-perspective-of-the-memory))
 ((READ-address1..16 unk)(STORE-in1..32 unk)))
 ((INSTRUCTION-out1..16 READ)(LOAD-out1..32 READ))
 ((1FSM-  FSM-control
         ((RESETin unk)(new-instruction INSTRUCTION-out1..16))
         ((FETCH)(EXECUTE))   )
  (2ALU-  ALU-processing
         ((Ain  6AC-rout1..32)(Bin  LOAD-out1)(OPin  INSTRUCTION-out1..16))
         ((Cout  6AC-rin1..32))   )
  (3IR-  16-bit-register
         ((rin1..16  INSTRUCTION-out1..16))
         ((rout1..16  4PCrin1..16)(rout1..16  5MR-rin1..16))   )
  (4PC-  16-bit-register
         ((rin1..16  3IR-rout1..16))
         ((rout1..16  5MR-rin1..16))   )
  (5MR-  16-bit-register
         ((rin1..16  3IR-rout1..16)(rin1..16 4PC-rout1..16)
         ((rout1..16  READ-address1..16))   )
  (6AC-  32-bit-register
         ((rin1..32  2ALU-Cout))
         ((rout1.32  2ALU-Ain)(rout1.32  STORE-in1..32))   )  )  )

((FSM-control)
 ((RESETin unk)(new-instruction unk))
 ((FETCH 2)(FETCH 3)(FETCH 5)
  (EXECUTE 7)(EXECUTE 8)(EXECUTE 10)(EXECUTE 11)
  (EXECUTE 12)(EXECUTE 13)(EXECUTE 14))
 ((0   RESETin)
  (1   RESET:  when 0 do 2 then 3 then 4)
  (2   FETCH:  PC -> MR)
  (3   FETCH:  read-memory)
  (4   when READ do 5)
  (5   FETCH:  memory -> IR)
  (6   DECODE:  if 5 do 7 then 8)
  (7   EXECUTE:  IR -> MR)
  (8   EXECUTE:  read-memory)
  (9   when READ do 10 then 11 then 12 then 13 then 14 then 2)
  (10  EXECUTE:  LOAD -> Bin)
  (11  EXECUTE:  AC -> Ain)
  (12  EXECUTE:  <op> -> ALU)
  (13  EXECUTE:  Cout -> AC)
  (14  EXECUTE:  PC + 1)   )  )
```

```
((ALU-processing)
 ((Ain unk)(Bin unk)(OPin unk))
 ((Cout 0))
 ((0 <A OPin B>)   )

((16-bit-register)
 ((rin1..16 unk))
 ((rout1..16 0))
 ((0  rin1..16))   )

((32-bit-register)
 ((rin1..32 unk))
 ((rout1..32 0))
 ((0  rin1..32))   )
```

Figure IX: Hierarchical Modularization of a CPU, WOTAN

```
                                                          instances        total

      ((main)                                                                --   1
       <input/output>
       (<registers>
        (23-..46-    aluslice  <bindings>)                 -- 24
        (47-..62-    pcslice   <bindings>)                 -- 16
        (63-..64-    decoder   <bindings>)                 --  2
        (65-         encoder <bindings>)                   --  1
        <main-functionality>))

       ((pcslice)                                                        --  16
        <input/output>
        (<register>
         (21-..22-   m2        <bindings>)                 --  2
         <pcslice-functionality>))

       ((aluslice)                                                       --  24
        <input/output>
        (<register>
         (3-..13-    m2        <bindings>)                 -- 11
         (14-..20-   regcell   <bindings>)                 --  7
         <aluslice-functionality>)))

       ((regcell)                                                        -- 168
        <input/output>
        (<register>
         (1-..2-     m2        <bindings>))                --  2

       ((m2)                                                             -- 662
        <input/output>
        (<m2-functionality>))

       ((encoder)                                                        --   1
        <input/output>
        (<encoder-functionality>))

       ((decoder)                                                        --   2
        <input/output>
        (<decoder-functionality>))
```

Figure X: Functional Modules in an ILOC body

ILOC body

```
((oa 4))
((1   (a (b c))   )
 (2   (d (e f))   )
 (3   (g (h i))   )
 (4   (1 (2 3))   )  )
```

Module abstraction

```
((oa 4-0))
((1-  common  ((i1 a)(i2 b)(i3 c)) ((o1 1-0)))
 (2-  common  ((i1 d)(i2 e)(i3 f)) ((o1 2-0)))
 (3-  common  ((i1 g)(i2 h)(i3 i)) ((o1 3-0)))
 (4-  common  ((i1 1-0)(i2 2-0)(i3 3-0)) ((o1 4-0)))

((common)
 ((i1 unk)(i2 unk)(i3 unk))
 ((o1 2))
 ((1   (i2 i3)   )
  (2   (i1 1)   )  ))
```

Figure XI: The Annotated ILOC Format for a 4-bit Magnitude Comparator Circuit, Prior to Abstraction for Low-level Module Components

```
4-bit magnitude comparator circuit

((oa 22) (ob 18) (oc 21))
  ((1     (a (b))   )                          -- XOR-module
   (2     (b (a))   )                          -- XOR-module
   (3     (c (d))   )                          -- XOR-module
   (4     (d (c))   )                          -- XOR-module
   (5     (e (f))   )                          -- XOR-module
   (6     (f (e))   )                          -- XOR-module
   (7     (g (h))   )                          -- XOR-module
   (8     (h (g))   )                          -- XOR-module
   (9     ((j)(8))   )                         -- OR-AND-module
   (10    ((j)(7))   )                         -- OR-AND-module
   (11    ((3)(20))   )                        -- OR-AND-module
   (12    ((4)(20))   )                        -- OR-AND-module
   (13    ((6)(17))   )                        -- OR-AND-module
   (14    ((5)(17))   )                        -- OR-AND-module
   (15    ((1)(19))   )                        -- OR-AND-module
   (16    ((2)(19))   )                        -- OR-AND-module
   (17    (7 8 (j))   )                        -- 3NOR-module
   (18    (1 2 (19))   )                       -- 3NOR-module
   (19    (3 4 (20))   )                       -- 3NOR-module
   (20    (5 6 (17))   )                       -- 3NOR-module
   (21    ((k 9 12 13 16))   )                 -- OR-AND-module
   (22    ((i 10 11 14 15))   )                -- OR-AND-module
```

Figure XII: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracting the XORG Low-level Module Components

```
4-bit magnitude comparator circuit

XOR-module:   two inputs,  two outputs

  ((oa 22)(ob 18)(oc 21))

  ((1-    xorg  ((i0 a)(i1 b))  ((o0 1-0)(o1 1-1))   )
   (2-    xorg  ((i0 c)(i1 d))  ((o0 2-0)(o1 2-1))   )
   (3-    xorg  ((i0 e)(i1 f))  ((o0 3-0)(o1 3-1))   )
   (4-    xorg  ((i0 g)(i1 h))  ((o0 4-0)(o1 4-1))   )
   (9     ((j)(4-1))   )
   (10    ((j)(4-0))   )
   (11    ((2-0)(20))   )
   (12    ((2-1)(20))   )
   (13    ((3-1)(17))   )
   (14    ((3-0)(17))   )
   (15    ((1-0)(19))   )
   (16    ((1-1)(19))   )
   (17    (4-0 4-1 (j))   )
   (18    (1-0 1-1 (19))   )
   (19    (2-0 2-1 (20))   )
   (20    (3-0 3-1 (17))   )
   (21    ((k 9 12 13 16))   )
   (22    ((i 10 11 14 15))   ) ))

 ((xorg)
  ((i0 unk)(i1 unk))
  ((o0 0)(o1 1))
  ((0  (i0 (i1)) )
   (1  (i1 (i0)) )  )) )
```

Figure XIII: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracting the 3NOR Low-level Module Components

```
4-bit magnitude comparator circuit

3NOR-module:    three inputs,  three outputs

  ((oa 22)(ob 8-2)(oc 21))

  ((1     (a (b))   )
   (2     (b (a))   )
   (3     (c (d))   )
   (4     (d (c))   )
   (5     (e (f))   )
   (6     (f (e))   )
   (7     (g (h))   )
   (8     (h (g))   )
   (9     ((j)(8))   )
   (10    ((j)(7))   )
   (11    ((3)(6-2))   )
   (12    ((4)(6-2))   )
   (13    ((6)(5-2))   )
   (14    ((5)(5-2))   )
   (15    ((1)(7-2))   )
   (16    ((2)(7-2))   )
   (5-    3nor  ((i0 7)(i1 8)(i2   j))  ((o0 5-0)(o1 5-1)(o2 5-2))   )
   (6-    3nor  ((i0 5)(i1 6)(i2 5-2))  ((o0 6-0)(o1 6-1)(o2 6-2))   )
   (7-    3nor  ((i0 3)(i1 4)(i2 6-2))  ((o0 7-0)(o1 7-1)(o2 7-2))   )
   (8-    3nor  ((i0 1)(i1 2)(i2 7-2))  ((o0 8-0)(o1 8-1)(o2 8-2))   )
   (21    ((k 9 12 13 16))   )
   (22    ((i 10 11 14 15))   ) ))

 ((3nor)
  ((i0 unk)(i1 unk)(i2 unk))
  ((o0 i0)(o1 i1)(o2 2))
  ((2     (i0 i1 (i2))   )  )) )
```

Figure XIV: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracting the OR-AND Low-level Module Components

```
4-bit magnitude comparator circuit

OR-AND-module:    nine inputs,  one output

  ((oa 9-0)(ob 18)(oc 10-0))

  ((1     (a (b))   )
   (2     (b (a))   )
   (3     (c (d))   )
   (4     (d (c))   )
   (5     (e (f))   )
   (6     (f (e))   )
   (7     (g (h))   )
   (8     (h (g))   )
   (17    (7 8 (j))   )
   (18    (1 2 (19))   )
   (19    (3 4 (20))   )
   (20    (5 6 (17))   )
   (9-    or-and
    ((i0 i)(i1 j)(i2 7)(i3 3)(i4 20)(i5 5)(i6 17)(i7 1)(i8 19)) ((o0 9-0)) )
   (10-   or-and
    ((i0 k)(i1 j)(i2 8)(i3 4)(i4 20)(i5 6)(i6 17)(i7 2)(i8 19)) ((o0 10-0)) )
   ))

 ((or-and)
  ((i0 unk)(i1 unk)(i2 unk)(i3 unk)(i4 unk)(i5 unk)(i6 unk)(i7 unk)(i8 unk))
  ((o0 0))
  ((0  (( i0 ((i1)(i2)) ((i3)(i4)) ((i5)(i6)) ((i7)(i8)) ))    ) ))  )
```

Figure XV: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracted Using All Available Low-level Module Components

```
4-bit magnitude comparator circuit

XORG, 3OR and OR-AND combined

  ((eq 8-2)(gt 9-0)(lt 10-0))

  ((1-   xorg  ((i0 a)(i1 b))  ((o0 1-0)(o1 1-1))   )
   (2-   xorg  ((i0 c)(i1 d))  ((o0 2-0)(o1 2-1))   )
   (3-   xorg  ((i0 e)(i1 f))  ((o0 3-0)(o1 3-1))   )
   (4-   xorg  ((i0 g)(i1 h))  ((o0 4-0)(o1 4-1))   )

   (5-   3nor  ((i0 4-0)(i1 4-1)(i2   j))  ((o0 5-0)(o1 5-1)(o2 5-2))   )
   (6-   3nor  ((i0 3-0)(i1 3-1)(i2 5-2))  ((o0 6-0)(o1 6-1)(o2 6-2))   )
   (7-   3nor  ((i0 2-0)(i1 2-1)(i2 6-2))  ((o0 7-0)(o1 7-1)(o2 7-2))   )
   (8-   3nor  ((i0 1-0)(i1 1-1)(i2 7-2))  ((o0 8-0)(o1 8-1)(o2 8-2))   )

   (9-   or-and
    ((i0 i)(i1 j)(i2 4-0)(i3 2-0)(i4 6-2)(i5 3-0)(i6 5-2)(i7 1-0)(i8 7-2))
    ((o0 9-0))   )
   (10-  or-and
    ((i0 k)(i1 j)(i2 4-1)(i3 2-1)(i4 6-2)(i5 3-1)(i6 5-2)(i7 1-1)(i8 7-2))
    ((o0 10-0))   )  ))

 ((xorg)
  ((i0 unk)(i1 unk))
  ((o0 0) (o1 1))
  ((0     (i0 (i1))   )
   (1     (i1 (i0))   )  ))

 ((3nor)
  ((i0 unk)(i1 unk)(i2 unk))
  ((o0 i0)(o1 i1)(o2 2))
  ((2     (i0 i1 (i2))   )  ))

 ((or-and)
  ((i0 unk)(i1 unk)(i2 unk)(i3 unk)(i4 unk)(i5 unk)(i6 unk)(i7 unk)(i8 unk))
  ((o0 0))
  ((0     ((i0 ((i1)(i2)) ((i3)(i4)) ((i5)(i6)) ((i7)(i8))))   )  ))
```

Figure XVI: The Circuit Schematic for the 4-bit Magnitude Comparator Generated by the ILOC Module Format, Abstracted Using Low-level Module Components

4-bit magnitude comparator circuit

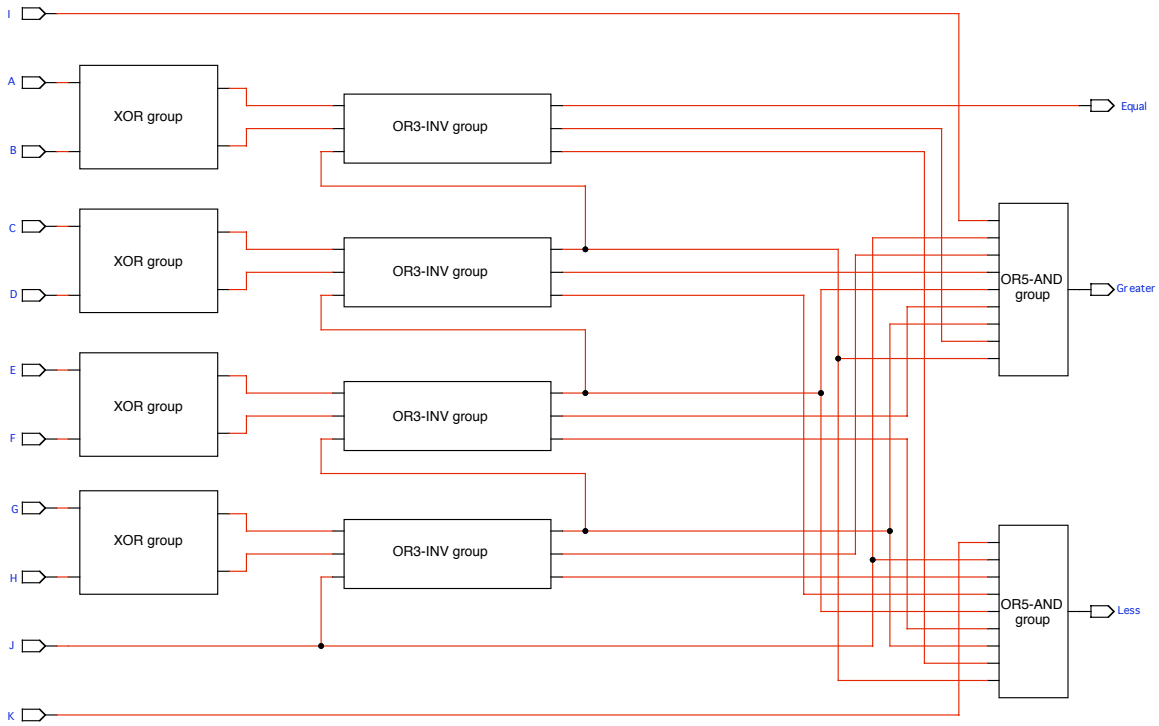


Figure XVII: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracted to Emphasize Routing Between Components

```
4-bit magnitude comparator circuit

AND-INV combined with XOR-3OR

  ((eq 10)(gt 34)(lt 35))

  ((21-    xor-3or
    ((i0 a) (i1 b)(i2 22-2))  ((o0 21-0)(o1 21-1)(o2 21-2))   )
   (22-    xor-3or
    ((i0 c) (i1 d)(i2 23-2))  ((o0 22-0)(o1 22-1)(o2 22-2))   )
   (23-    xor-3or
    ((i0 e) (i1 f)(i2 24-2))  ((o0 23-0)(o1 23-1)(o2 23-2))   )
   (24-    xor-3or
    ((i0 g) (i1 h)(i2    3))  ((o0 24-0)(o1 24-1)(o2 24-2))   )
   (25-    and-inv
    ((i0 21-1)(i1 21-0)(i2 22-2))  ((o0 25-0)(o1 25-1))   )
   (26-    and-inv
    ((i0 22-0)(i1 22-1)(i2 23-2))  ((o0 26-0)(o1 26-1))   )
   (27-    and-inv
    ((i0 23-0)(i1 23-1)(i2 24-2))  ((o0 27-0)(o1 27-1))   )
   (28-    and-inv
    ((i0 24-0)(i1 24-1)(i2    3))  ((o0 28-0)(o1 28-1))   )
   (3     (j)   )
   (10    (21-2)   )
   (34    ((i 28-0 25-1 27-0 26-0))   )
   (35    ((k 28-1 25-0 27-1 26-1))   )  ))

 ((xor-3or)
  ((i0 unk)(i1 unk)(i2 unk))
  ((o0 0)(o1 1)(o2 2))
  ((0     (i0 (i1))   )
   (1     (i1 (i0))   )
   (2     ((i2 0 1))   )  ))

 ((and-inv)
  ((i0 unk)(i1 unk)(i2 unk))
  ((o0 0)(o1 1))
  ((0     (i2 (i0))   )
   (1     (i2 (i1))   )  ))
```

Figure XVIII: The Circuit Schematic for the 4-bit Magnitude Comparator, Abstracted to Emphasize Routing Between Components

4-bit magnitude comparator circuit

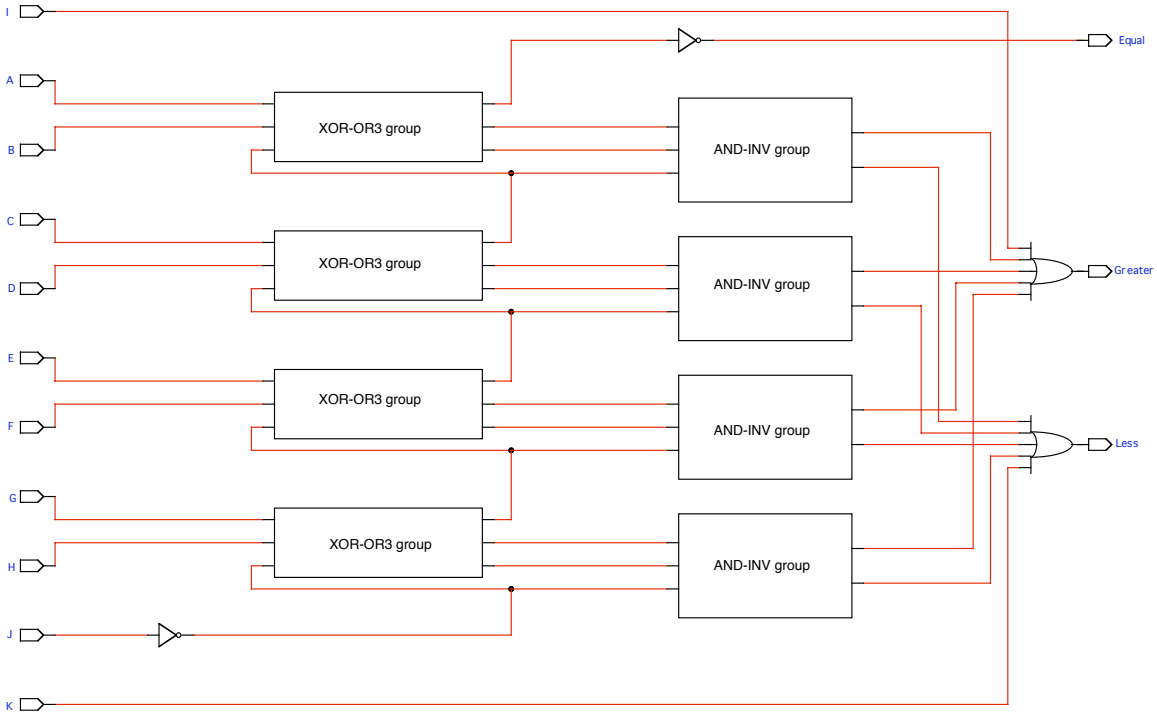


Figure XIX: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracted to Emphasize Sequential Routing Between Components

```
4-bit magnitude comparator circuit

XOR-3NOR combined with OR-AND

  ((eq 14-2)(gt 9-0)(lt 10-0))

  ((11-    xor-3nor
    ((i0 g)(i1 h)(i2    j))  ((o0 11-0)(o1 11-1)(o2 11-2))   )
   (12-    xor-3nor
    ((i0 e)(i1 f)(i2 11-2))  ((o0 12-0)(o1 12-1)(o2 12-2))   )
   (13-    xor-3nor
    ((i0 c)(i1 d)(i2 12-2))  ((o0 13-0)(o1 13-1)(o2 13-2))   )
   (14-    xor-3nor
    ((i0 a)(i1 b)(i2 13-2))  ((o0 14-0)(o1 14-1)(o2 14-2))   )
   (9-     or-and
    ((i0 i)(i1 j)(i2 11-0)(i3 13-0)(i4 12-2)(i5 12-0)
     (i6 11-2)(i7 14-0)(i8 13-2))  ((o0 9-0))   )
   (10-    or-and
    ((i0 k)(i1 j)(i2 11-1)(i3 13-1)(i4 12-2)(i5 12-1)
     (i6 11-2)(i7 14-1)(i8 13-2))  ((o0 10-0))   )  ))

 ((xor-3nor)
  ((i0 unk)(i1 unk)(i2 unk))
  ((o0 0)(o1 1)(o2 2))
  ((0     (i0 (i1))   )
   (1     (i1 (i0))   )
   (2     (0 1 (i2))   )  ))

 ((or-and)
  ((i0 unk)(i1 unk)(i2 unk)(i3 unk)(i4 unk)(i5 unk)(i6 unk)(i7 unk)(i8 unk))
  ((o0 0))
  ((0     ((i0 ((i1)(i2)) ((i3)(i4)) ((i5)(i6)) ((i7)(i8))))   )  ))
```

Figure XX: The Circuit Schematic for the 4-bit Magnitude Comparator Generated by the ILOC Module Format, Abstracted to Emphasize Sequential Processing Between Components

4-bit magnitude comparator circuit

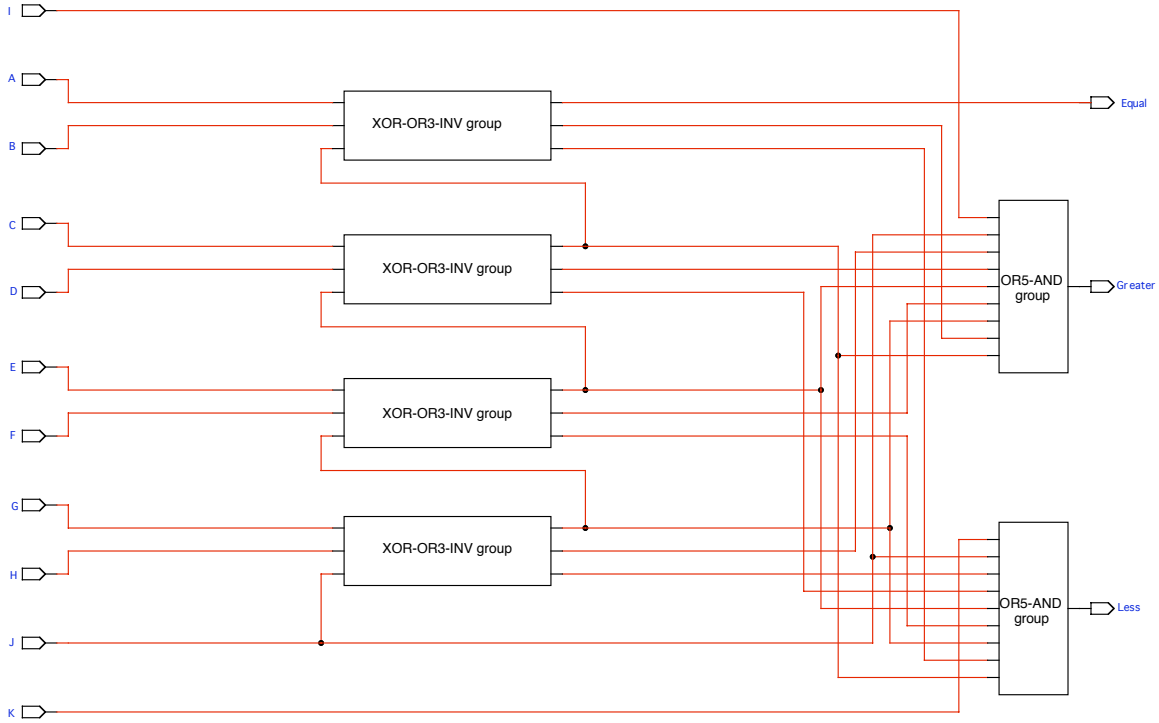


Figure XXI: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracted to Emphasize Parallel Processing Between Components

```
4-bit magnitude comparator circuit

3OR-ORAND combined with XOR-group:

  ((eq 8-2)(gt 15-0)(lt 16-0))

  ((1-     xorg  ((i0 a)(i1 b))  ((o0 1-0)(o1 1-1))   )
   (2-     xorg  ((i0 c)(i1 d))  ((o0 2-0)(o1 2-1))   )
   (3-     xorg  ((i0 e)(i1 f))  ((o0 3-0)(o1 3-1))   )
   (4-     xorg  ((i0 g)(i1 h))  ((o0 4-0)(o1 4-1))   )
   (15-    3or-or5-and
    ((i0 i)(i1 j)(i2 4-0)(i3 2-0)(i4 3-1)(i5 3-0)(i6 4-1)(i7 1-0)(i8 2-1))
    ((o0 15-0))   )
   (16-    3or-or5-and
    ((i0 k)(i1 j)(i2 4-1)(i3 2-1)(i4 3-0)(i5 3-1)(i6 4-0)(i7 1-1)(i8 2-0))
    ((o0 16-0))   )
   (8-2   (1-0 1-1 2-0 2-1 3-0 3-1 4-0 4-1 (j))   )  ))

 ((xorg)
  ((i0 unk)(i1 unk))
  ((o0 0)(o1 1))
  ((0     (i0 (i1))   )
   (1     (i1 (i0))   )  ))

 ((3or-or5-and)
  ((i0 unk)(i1 unk)(i2 unk)(i3 unk)(i4 unk)(i5 unk)(i6 unk)(i7 unk)(i8 unk))
  ((o0 0))
  ((0     ((i0 ((i1) (i2 (i6 (i5 (i4 (i3 (i8 (i7))))))))))   )  ))
```

Figure XXII: The Circuit Schematic for the 4-bit Magnitude Comparator Generated by the ILOC Module Format, Abstracted to Emphasize Parallel Processing Between Components

4-bit magnitude comparator circuit

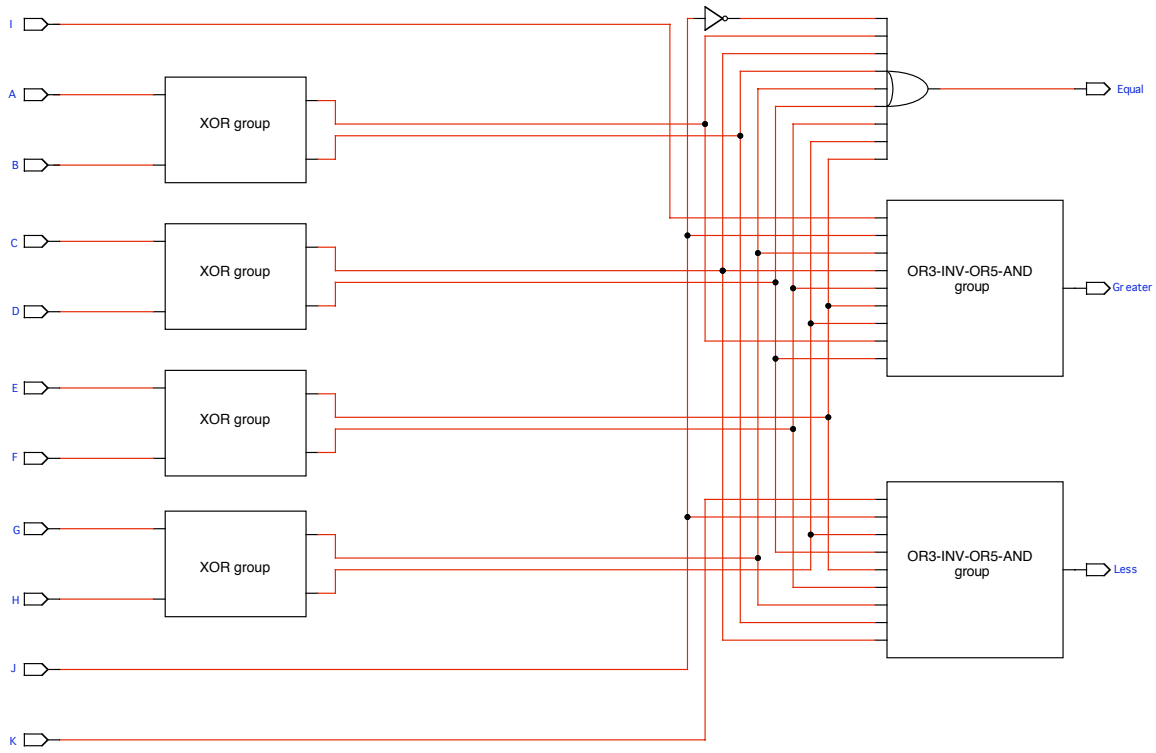


Figure XXIII: The ILOC Module Format for the 4-bit Magnitude Comparator Circuit, Abstracted to Emphasize Components Involved in Output

4-bit magnitude comparator circuit

XORG-3OR-ORAND group:

```
  ((eq 8-2)(gt 17-0)(lt 18-0))

  ((17-    xorg-3or-orand
    ((i0 i)(i1 j)(i2 g)(i3 c)(i4 f)(i5 e)(i6 h)(i7 a)(i8 d)(i9 b))
    ((o0 15-0))   )
   (18-    xorg-3or-orand
    ((i0 k)(i1 j)(i2 h)(i3 d)(i4 e)(i5 f)(i6 g)(i7 b)(i8 c)(i9 a))
    ((o0 16-0))   )
   (8-2   ((a (b))(b (a))(c (d))(d (c))(e (f))(f (e))(g (h))(h (g))(j))  ) ))

 ((xorg-3or-orand)
  ((i0 unk)(i1 unk)(i2 unk)(i3 unk)(i4 unk)
   (i5 unk)(i6 unk)(i7 unk)(i8 unk)(i9 unk))
  ((o0    0))
  ((0     ((i0
             ((i1)
              ((i2 (i6))
               ((i6 (i2))
                ((i5 (i4))
                 ((i4 (i5))
                  ((i3 (i8))
                   (i7 (i9) (i8 (i3)))))))))))   )  ))
```

Figure XXIV: The Circuit Schematic for the 4-bit Magnitude Comparator, Abstracted to Emphasize Components Involved in Output

4-bit magnitude comparator circuit

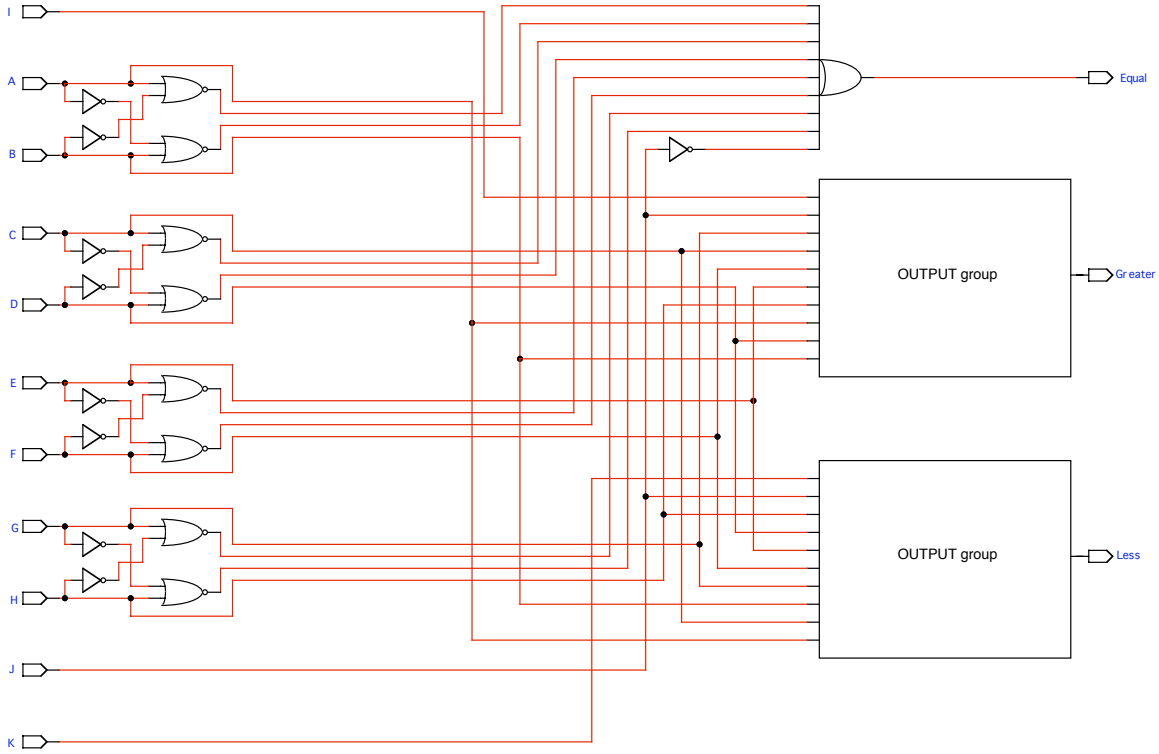


Figure XXV: The ILOC Generator Format for the 4-bit Magnitude Comparator Circuit with Enable Inputs Removed, Abstracted to Provide Parametric Generation of Functionality

4-bit magnitude comparator circuit

Base-case, 1-bit:

```
  ((eq 1)(gt 2)(lt 3))

  ((1   (((a0 (b0)) (b0 (a0))))   )                          -- EQ1
   (2   ((a0 (b0)))   )                                      -- GT1
   (3   ((b0 (a0)))   )  ))                                  -- LT1
```

Iterated, n-bit:

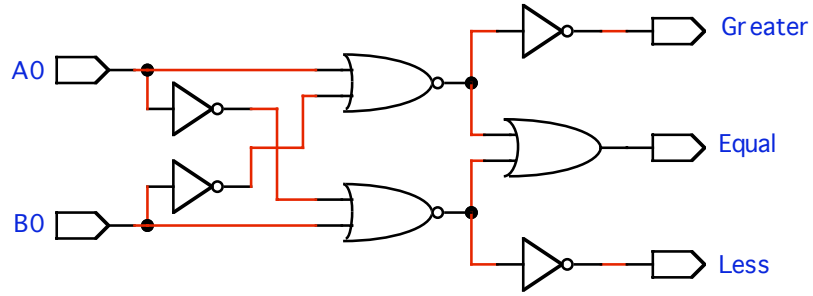
```
  ((eq 1)(gt 2)(lt 3))

  ((1   (((a0 (b0))(b0 (a0))(a1 (b1))(b1 (a1)) ... (an (bn))(bn (an))))   )
   (2   ((an (bn))
          ((bn (an))
           ...
            ((a1 (b1))
             ((b1 (a1))
              ((a0 (b0)))))))   )
   (3   ((bn (an))
          ((an (bn))
           ...
            ((b1 (a1))
             ((a1 (b1))
              ((b0 (a0)))))))   )  ))
```

Recursive generator, n-bit:

```
  ((eq 1)(gt 2)(lt 3))

  ((1   (((an (bn)) (bn (an)) EQn-1))   )
   (2   ((an (bn)) ((bn (an)) GTn-1))   )
   (3   ((bn (an)) ((an (bn)) LTn-1))   )  ))
```

Figure XXVI: The Circuit Schematic for the 4-bit Magnitude Comparator Generated by the ILOC Generator Format with Enable Inputs Removed, Abstracted to Provide Parametric Generation of Functionality

N-bit magnitude comparator circuit



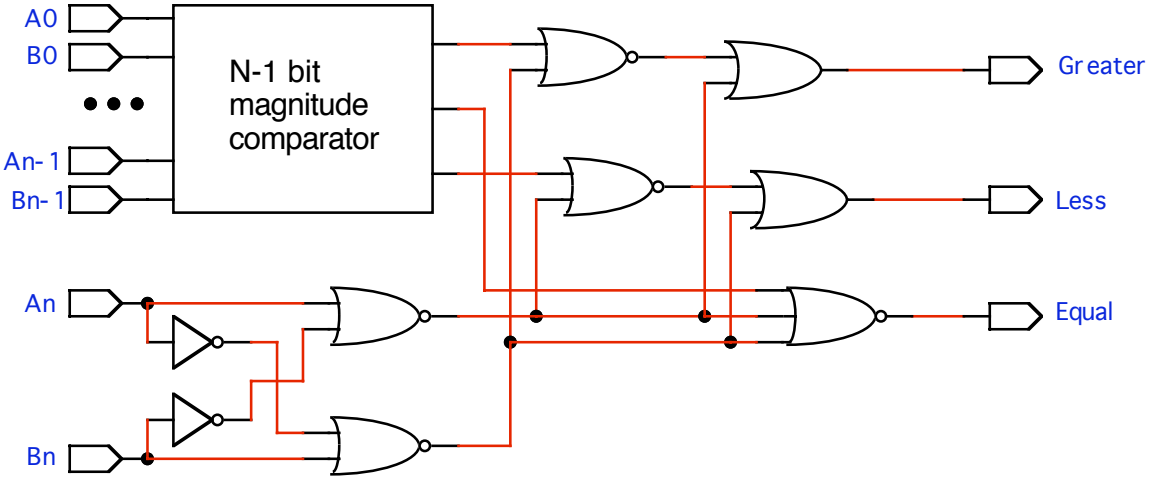


Figure XXVII: 2-bit and 3-bit Magnitude Comparators Generated by Recursive Expansion of the ILOC Generator Format

4-bit magnitude comparator circuit

Base-case, 1-bit

```
  ((eq 1)(gt 2)(lt 3))

  ((1   (((a0 (b0)) (b0 (a0))))   )                          -- EQ1
   (2   ((a0 (b0)))   )                                      -- GT1
   (3   ((b0 (a0)))   )  ))                                  -- LT1
```

Template, n-bit

```
  ((1   (((an (bn)) (bn (an)) EQn-1))   )
   (2   ((an (bn)) ((bn (an)) GTn-1))   )
   (3   ((bn (an)) ((an (bn)) LTn-1))   )  ))
```

Recursively generated, 2-bit, with substitutions in italics

```
  ((1   (((a1 (b1)) (b1 (a1)) (a0 (b0)) (b0 (a0))))   )
   (2   ((a1 (b1)) ((b1 (a1)) a0 (b0)))   )
   (3   ((b1 (a1)) ((a1 (b1)) b0 (a0)))   )  ))
```

Recursively generated, 3-bit, with substitutions in italics

```
  ((1   (((a2 (b2)) (b2 (a2)) (a1 (b1)) (b1 (a1)) (a0 (b0)) (b0 (a0))))   )
   (2   ((a2 (b2)) ((b2 (a2)) ((a1 (b1)) ((b1 (a1)) a0 (b0)))))   )
   (3   ((b2 (a2)) ((a2 (b2)) ((b1 (a1)) ((a1 (b1)) b0 (a0)))))   )  ))
```

Recursively generated, 4-bit, with substitutions in italics

```
  ((1   (((a3 (b3)) (bn (a3))
          (a2 (b2)) (b2 (a2)) (a1 (b1)) (b1 (a1)) (a0 (b0)) (b0 (a0))))   )
   (2   ((a3 (b3)) ((bn (a3))
         ((a2 (b2)) ((b2 (a2)) ((a1 (b1)) ((b1 (a1)) a0 (b0)))))))   )
   (3   ((b3 (a3)) ((an (b3))
         ((b2 (a2)) ((a2 (b2)) ((b1 (a1)) ((a1 (b1)) b0 (a0)))))))   )  ))
```

Figure XXVIII: 2-bit and 3-bit Magnitude Comparator Schematics Generated by Recursive Expansion of the ILOC Generator Schematics

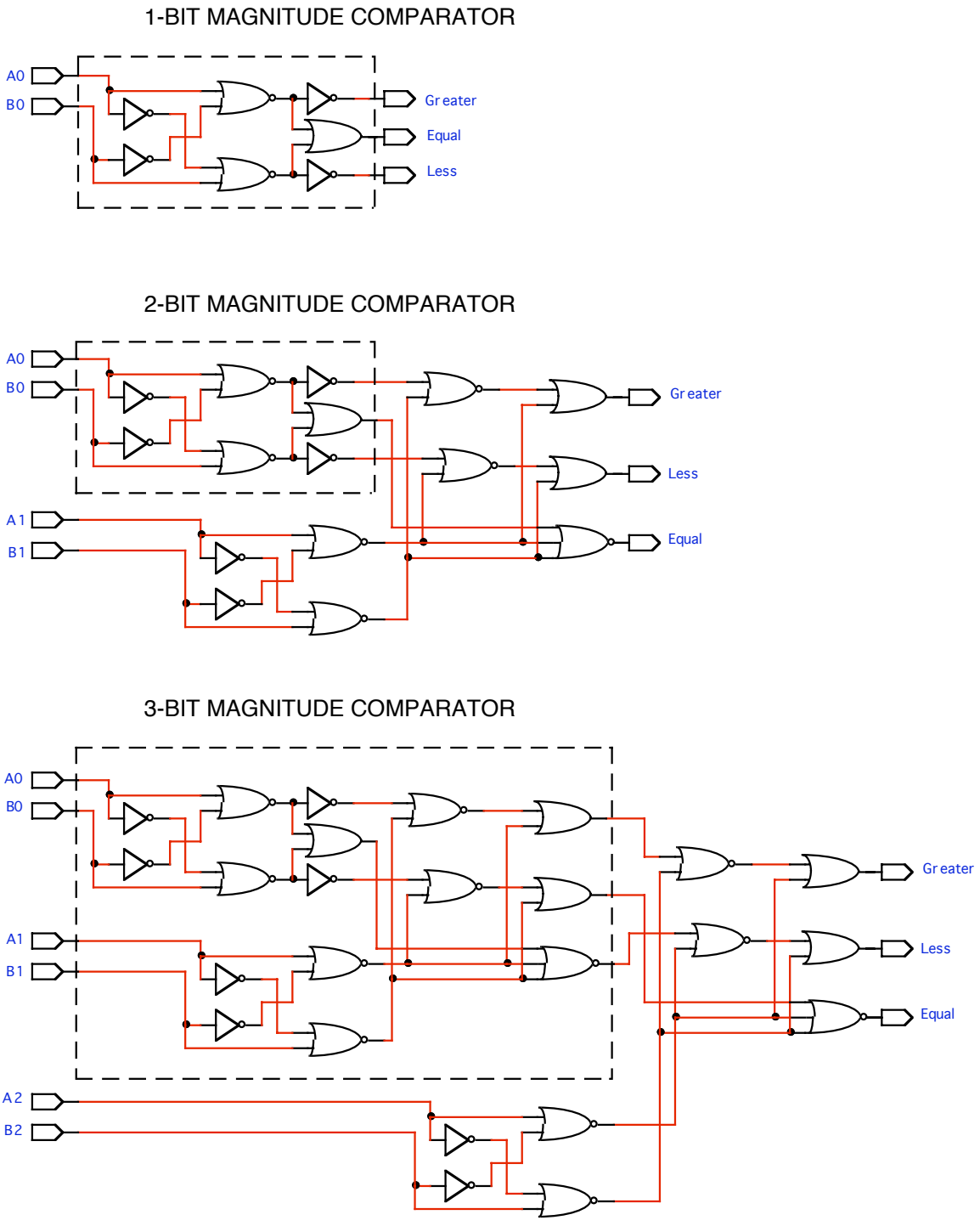


Figure XXIX: The Circuit Schematic for the Entire 4-bit Magnitude Comparator Generated by the ILOC Generator Format, Abstracted to Provide Parametric Generation of Functionality

N-bit magnitude comparator circuit

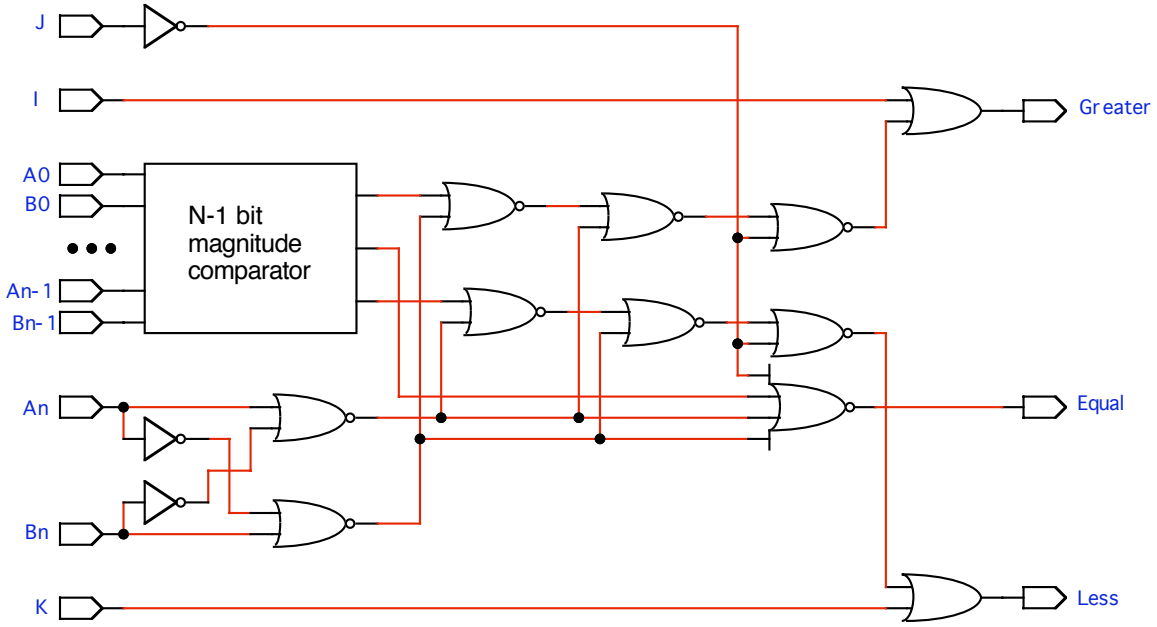


Figure XXX: ILOC Format for a 8-bit Adder Generated Recursively, Before Optimization

8-bit adder, generated

```
  ((s0 4)(s1 8)(s2 12)(s3 16)(s4 20)(s5 24)(s6 28)(s7 32)(s8 33))

  ((1     (nil)   )
   (2     (a0 b0)   )
   (3     ((a0)(b0))   )
   (4     (((1 2 3) ((1)(2 3))))   )
   (5     ((3 (2 (1))))   )
   (6     (a1 b1)   )
   (7     ((a1)(b1))   )
   (8     (((5 6 7) ((5)(6 7))))   )
   (9     ((7 (6 (5))))   )
   (10    (a2 b2)   )
   (11    ((a2)(b2))   )
   (12    (((9 10 11) ((9)(10 11))))   )
   (13    ((11 (10 (9))))   )
   (14    (a3 b3)   )
   (15    ((a3)(b3))   )
   (16    (((13 14 15) ((13)(14 15))))   )
   (17    ((15 (14 (13))))   )
   (18    (a4 b4)   )
   (19    ((a4)(b4))   )
   (20    (((17 18 19) ((17)(18 19))))   )
   (21    ((19 (18 (17))))   )
   (22    (a5 b5)   )
   (23    ((a5)(b5))   )
   (24    (((21 22 23) ((21)(22 23))))   )
   (25    ((23 (22 (21))))   )
   (26    (a6 b6)   )
   (27    ((a6)(b6))   )
   (28    (((25 26 27) ((25)(26 27))))   )
   (29    ((27 (26 (25))))   )
   (30    (a7 b7)   )
   (31    ((a7)(b7))   )
   (32    (((29 30 31) ((29)(30 31))))   )
   (33    ((31 (30 (29))))   )  )
```

Figure XXXI: ILOC Format for a 8-bit Adder Generated Recursively, After Optimization

8-bit adder, generated and optimized

```
  ((s0 16)(s1 =31)(s2 =32)(s3 =33)(s4 =34)(s5 =35)(s6 =36)(s7 =37)(s8 23))

  ((1     (a1 b1)   )
   (2     (a2 b2)   )
   (3     (a3 b3)   )
   (4     (a4 b4)   )
   (5     (a5 b5)   )
   (6     (a6 b6)   )
   (7     (a7 b7)   )
   (8     ((a0)(b0))   )
   (9     ((a1)(b1))   )
   (10    ((a2)(b2))   )
   (11    ((a3)(b3))   )
   (12    ((a4)(b4))   )
   (13    ((a5)(b5))   )
   (14    ((a6)(b6))   )
   (15    ((a7)(b7))   )
   (^24   ((1 9))   )
   (^25   ((2 10))   )
   (^26   ((3 11))   )
   (^27   ((4 12))   )
   (^28   ((5 13))   )
   (^29   ((6 14))   )
   (^30   ((7 15))   )
   (16    (8 (a0 b0))   )
   (17    ((9 (1 (8))))   )
   (18    ((10 (2 (17))))   )
   (19    ((11 (3 (18))))   )
   (20    ((12 (4 (19))))   )
   (21    ((13 (5 (20))))   )
   (22    ((14 (6 (21))))   )
   (23    ((15 (7 (22))))   )
   (=31   (((8 ^24) ((8)(^24))))   )
   (=32   (((17 ^25) ((17)(^25))))   )
   (=33   (((18 ^26) ((18)(^26))))   )
   (=34   (((19 ^27) ((19)(^27))))   )
   (=35   (((20 ^28) ((20)(^28))))   )
   (=36   (((21 ^29) ((21)(^29))))   )
   (=37   (((22 ^30) ((22)(^30))))   )  )
```

Figure XXXII: A Simple Example of Vectorization Compared to Functional Abstraction and Bit-width Abstraction

ILOC body fragment

```
((1   (a 6)   )
 (2   (a 7)   )
 (3   (a 8)   )
 (4   (a 9)   )
 (5   (1 2 3 6 7)   )
 (6   (b c)   )
 (7   (d e)   )  )
```

Function abstraction using modularization

```
((1-   fn  ((i1 6))  ((o1 1-0))   )
 (2-   fn  ((i1 7))  ((o1 2-0))   )
 (3-   fn  ((i1 8))  ((o1 3-0))   )
 (4-   fn  ((i1 9))  ((o1 4-0))   )
 (5    (1-0 2-0 3-0 6 7)   )
 (6    (b c)   )
 (7    (d e)   )  )
((fn)
 ((i1 unk))
 ((o1 1))
((1   (a i1)   )  ))
```

Vector abstractions

	explicit	*range*
V1:	<1 2 3 4>	<1..4>
V2:	<6 7 8 9>	<6..9>

Vector abstracted ILOC body fragment

```
((V1-   V1  <1 2 3 4> )
 (V2-   V2  <6 7 8 9> )
 (V1   (a V2)   )
 (5    (V1-1 V1-2 V1-3 6 7)   )
 (6   (b c)   )
 (7   (d e)   )  )
```

Figure XXXIII: A Vectorized MUX Function

ILOC body

```
((oa >1)(ob >2)(oc >3)(od >4)(oe >5)(of >6)(og >7)(oh >8))

((>1    ((a i)(j (i)))   )
 (>2    ((b i)(k (i)))   )
 (>3    ((c i)(l (i)))   )
 (>4    ((d i)(m (i)))   )
 (>5    ((e i)(n (i)))   )
 (>6    ((f i)(o (i)))   )
 (>7    ((g i)(p (i)))   )
 (>8    ((h i)(q (i)))   )  )
```

Vector abstractions

	explicit	*range*
Vo1 =	<oa ob oc od oe of og oh>	<oa..oh>
V1 =	<>1 >2 >3 >4 >5 >6 >7 >8>	<>1..>8>
V2 =	<a b c d e f g h>	<a..h>
V3 =	<j k l m n o p q>	<j..q>

ILOC vectorized body

```
((Vo1-  Vo1  <oa..oh>   )
 (Vo1 >V1))

((V1-   V1   <>1..>8>   )
 (V2-   V2   <a..h>   )
 (V3-   V3   <j..q>   )
 (>V1   ((V2 i)(V3 (i))   )  )
```

Figure XXXIV: Vector Abstraction in a Schematic Form

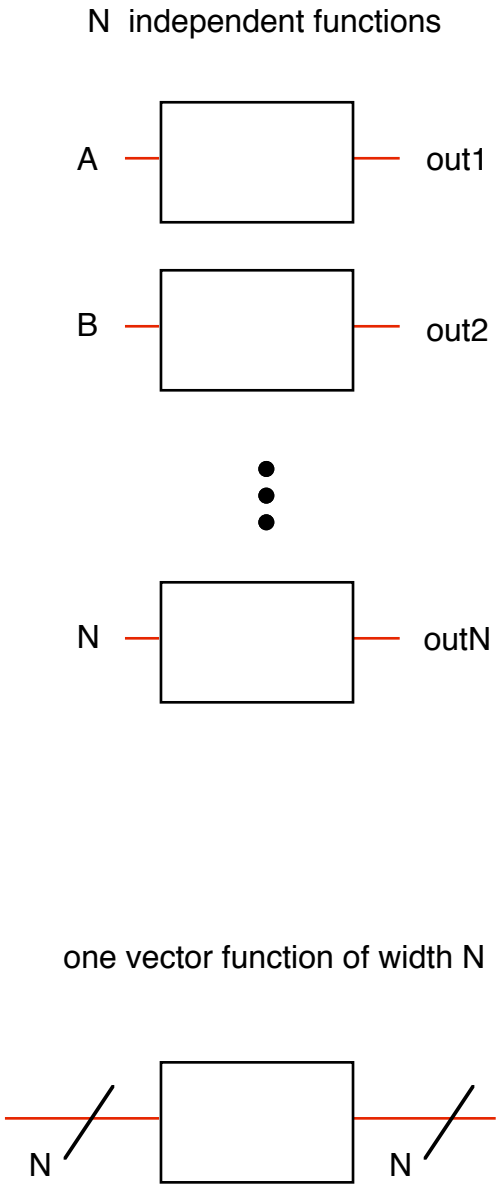


Figure XXXVa: Vectorization of a 8-bit Sequential Multiplier, the Non-vectorized ILOC Format

8-bit sequential multiplier ILOC body part 1

```
  ((oa 17))
  ((!1    (@  35)   )
   (!2    (@ =66)   )
   (!3    (@ =67)   )
   (!4    (@ =68)   )
   (!5    (@ =69)   )
   (!6    (@ =70)   )
   (!7    (@ =71)   )
   (!8    (@ =72)   )
   (!9    (@ =73)   )
   (!10   (@ =74)   )
   (!11   (@ =75)   )
   (!12   (@ =76)   )
   (!13   (@ =77)   )
   (!14   (@ =78)   )
   (!15   (@ =79)   )
   (!16   (@ =80)   )
   (18    ((a)(q))   )
   (19    ((b)(q))   )
   (20    ((c)(q))   )
   (21    ((d)(q))   )
   (22    ((e)(q))   )
   (23    ((f)(q))   )
   (24    ((g)(q))   )
   (25    ((h)(q))   )
   (26    ((i)(q))   )
   (27    ((j)(q))   )
   (28    ((k)(q))   )
   (29    ((l)(q))   )
   (30    ((m)(q))   )
   (31    ((n)(q))   )
   (32    ((o)(q))   )
   (33    ((p)(q))   )
   (35    (((18)( !1)) (36  (18  !1)))   )
   (36    (((19)( !2)) (37  (19  !2)))   )
   (37    (((20)( !3)) (38  (20  !3)))   )
   (38    (((21)( !4)) (39  (21  !4)))   )
   (39    (((22)( !5)) (40  (22  !5)))   )
   (40    (((23)( !6)) (41  (23  !6)))   )
   (41    (((24)( !7)) (42  (24  !7)))   )
   (42    (((25)( !8)) (43  (25  !8)))   )
   (43    (((26)( !9)) (44  (26  !9)))   )
   (44    (((27)(!10)) (45  (27 !10)))   )
   (45    (((28)(!11)) (46  (28 !11)))   )
   (46    (((29)(!12)) (47  (29 !12)))   )
   (47    (((30)(!13)) (48  (30 !13)))   )
   (48    (((31)(!14)) (49  (31 !14)))   )
   (49    (((32)(!15)) (82  (32 !15)))   )
   (82    (((33)(!16)) (( ) (33 !16)))   )
```

Figure XXXVb: Vectorization of a 8-bit Sequential Multiplier, the Non-vectorized ILOC Format Continued

8-bit sequential multiplier ILOC body part 2

<continued>

```
   (=50   (((18 36) ((18)(36))))   )
   (=51   (((19 37) ((19)(37))))   )
   (=52   (((20 38) ((20)(38))))   )
   (=53   (((21 39) ((21)(39))))   )
   (=54   (((22 40) ((22)(40))))   )
   (=55   (((23 41) ((23)(41))))   )
   (=56   (((24 42) ((24)(42))))   )
   (=57   (((25 43) ((25)(43))))   )
   (=58   (((26 44) ((26)(44))))   )
   (=59   (((27 45) ((27)(45))))   )
   (=60   (((28 46) ((28)(46))))   )
   (=61   (((29 47) ((29)(47))))   )
   (=62   (((30 48) ((30)(48))))   )
   (=63   (((31 49) ((31)(49))))   )
   (=64   (((32 82) ((32)(82))))   )
   (=65   ((( 33 !16) (( 33)(!16))))   )
   (=66   (((=50  !1) ((=50)( !1))))   )
   (=67   (((=51  !2) ((=51)( !2))))   )
   (=68   (((=52  !3) ((=52)( !3))))   )
   (=69   (((=53  !4) ((=53)( !4))))   )
   (=70   (((=54  !5) ((=54)( !5))))   )
   (=71   (((=55  !6) ((=55)( !6))))   )
   (=72   (((=56  !7) ((=56)( !7))))   )
   (=73   (((=57  !8) ((=57)( !8))))   )
   (=74   (((=58  !9) ((=58)( !9))))   )
   (=75   (((=59 !10) ((=59)(!10))))   )
   (=76   (((=60 !11) ((=60)(!11))))   )
   (=77   (((=61 !12) ((=61)(!12))))   )
   (=78   (((=62 !13) ((=62)(!13))))   )
   (=79   (((=63 !14) ((=63)(!14))))   )
   (=80   (((=64 !15) ((=64)(!15))))   )
   (=17   (((=65    ) ((=65)(   ))))   )  )  ))
```

Figure XXXVI: Vectorization of a 8-bit Sequential Multiplier, ILOC Vectorized Format

Vectors

```
V1  = <a b c d e f g h i j k l m n o p>

V2a = <!1 !2 !3 !4 !5 !6 !7 !8 !9 !10 !11 !12 !13 !14 !15 !16>

V2b = <!1 !2 !3 !4 !5 !6 !7 !8 !9 !10 !11 !12 !13 !14 !15 LOW>

V3  = <18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33>

V4a = <   36 37 38 39 40 41 42 43 44 45 46 47 48 49 82 HIGH>

V4b = <   36 37 38 39 40 41 42 43 44 45 46 47 48 49 82 !16>

V4c = <35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 82    >

V5  = <=50 =51 =52 =53 =54 =55 =56 =57 =58 =59 =60 =61 =62 =63 =64 =65>

V6a = <35 =66 =67 =68 =69 =70 =71 =72 =73 =74 =75 =76 =77 =78 =79 =80    >

V6b = <   =66 =67 =68 =69 =70 =71 =72 =73 =74 =75 =76 =77 =78 =79 =80 =17>
```

8-bit sequential multiplier ILOC vector body

```
((oa =V6b-16))

((V1-     Vi1   <a..p>   )                                  -- 16
 (V2a-    !V2a  <!1..!16>   )                               -- 16
 (V2b-    !V2b  <!1..!15 LOW>   )                           -- 16
 (V3-     V3    <18..33>   )                                -- 16
 (V4a-    V4a   <36..49 82 HIGH>   )                        -- 16
 (V4b-    V4b   <36..49 82 !16>   )                         -- 16
 (V4c-    V4c   <35 36..49 82>   )                          -- 16
 (V5-     =V5   <=50..=65>   )                              -- 16
 (V6a-    V6a   <35 =66..=80>   )                           -- 16
 (V6b-    =V6b  <=66..=80 =17>   )                          -- 16

 (!V2a  (@ V6a)  )
 (V3    ((q)(Vi1))   )
 (V4c   (((V3)(!V2a)) (V4a (V3 !V2a)))   )
 (=V5   (((V3 V4b) ((V3)(V4b))))    )
 (=V6b  (((V5 !V2b) ((V5)(!V2b))))   )  )
```

Figure XXXVII: Vector Transformation of a 8-bit Sequential Multiplier

8-bit sequential multiplier ILOC vector body

```
((oa =V6b-16))

((V1-     Vi1   <a..p>   )
 (V2a-    !V2a  <!1..!16>   )
 (V2b-    !V2b  <!1..!15 LOW>   )                    -- !V2a edge grounding
 (V3-     V3    <18..33>   )
 (V4a-    V4a   <36..49 82 HIGH>   )                 -- V4b edge grounding
 (V4b-    V4b   <36..49 82 !16>   )
 (V4c-    V4c   <35 36..49 82>   )                   -- V4b shift down
 (V5-     =V5   <=50..=65>   )
 (V6a-    V6a   <35 =66..=80>   )
 (V6b-    =V6b  <=66..=80 =17>   )                   -- V6a shift up

 (!V2a  (@ V6a)  )
 (V3    ((q)(Vi1))   )
 (V4c   (((V3)(!V2a)) (V4a (V3 !V2a)))   )
 (=V5   (((V3 V4b) ((V3)(V4b))))    )
 (=V6b  (((V5 !V2b) ((V5)(!V2b))))   )  )
```

Expanding V3 and V5

```
((oa =V6b-16))

((V1-     Vi1   <a..p>   )
 (V2a-    !V2a  <!1..!16>   )
 (V2b-    !V2b  <!1..!15 LOW>   )                    -- !V2a edge grounding
 (V4a-    V4a   <36..49 82 HIGH>   )                 -- V4b edge grounding
 (V4b-    V4b   <36..49 82 !16>   )
 (V4c-    V4c   <35 36..49 82>   )                   -- V4b shift down
 (V6a-    V6a   <35 =66..=80>   )
 (V6b-    =V6b  <=66..=80 =17>   )                   -- V6a shift up

 (!V2a  (@ V6a)  )
 (V4c   (((q)(Vi1)(!V2a)) (V4a (!V2a ((q)(Vi1)))))   )
 (=V6b  (((!V2b (V4b ((q)(Vi1))) ((q)(Vi1)(V4b)))
          ((!V2b) ((V4b ((q)(Vi1))) ((q)(Vi1)(V4b))))))   )  )
```

Figure XXXVIII: MCNC Benchmark Circuit I7, a Large XOR/MUX Circuit in ILOC Vector Format, Vector Components

```
(Vo2-   Vout2   <pn..qo>   )                          -- 28
(Vo3-   Vout3   <oh..pi>   )                          -- 28
(Vo4-   Vout4   <ob oc>    )                          --  2
(Vo5-   Vout5   <od..og>   )                          --  4
(Vo6-   Vout6   <pj..pm>   )                          --  4

(V2-    Vtop2   <1..28>    )                          -- 28
(V32-   >V32    <>214..>241>    )                     -- 28
(V33-   =V33    <=113.. =140>    )                    -- 28
(V34-   >V34    <>173..>200>    )                     -- 28
(V41-   Vi41    <gp..hq>   )                          -- 28
(V42-   Vi42    <fj..gk>   )                          -- 28
(V11-   Vi11    <dz..ec>   )                          -- 28

(V3-    >Vtop3  <>33..>41 >46..>64>    )              -- 28
(V31-   >V31    <>201..>209 >246..>264>    )          -- 28
(V19-   =V19    <=81..=108>    )                      -- 28
(V23-   >V23    <>141..>168>    )                     -- 28
(V10-   Vi10    <al..bm>   )                          -- 28
(V12-   Vi12    <cx..dy>   )                          -- 28
(V13-   Vi13    <br..c2>   )                          -- 28

(V4-    Vtop4   <65 66>    )                          --  2
(V17-   =V17    <=75 =76>    )                        --  2
(V29-   Vi29    <af ag>    )                          --  2

(V5-    Vtop5   <67..70>   )                          --  4
(V30-   >V30    <>242..>245>    )                     --  4
(V16-   ^V16    <^71..^74>    )                       --  4
(V18-   =V18    <=77..=80>    )                       --  4
(V7-    Vi7     <ct..cw>   )                          --  4
(V8-    Vi8     <bn..bq>   )                          --  4
(V9-    Vi9     <ah..ak>   )                          --  4

(V6-    >Vtop6  <>42..>45>    )                       --  4
(V28-   >V28    <>210..>213>    )                     --  4
(V35    =V35    <=109..=112>    )                     --  4
(V25-   >V25    <>169..>172>    )                     --  4
(V39-   Vi39    <gl..go>   )                          --  4
(V40-   Vi40    <ff..fi>   )                          --  4
(V43-   Vi43    <ed..fe>   )                          --  4
```

Figure XXXIX: MCNC Benchmark Circuit I7, a Large XOR/MUX Circuit in ILOC Vector Format, ILOC Vector Body

ILOC vector body for the I7 benchmark circuit

```
  ((oa 31)(Vout2 >Vtop2)(Vout3 >Vtop3)
          (Vout4 Vtop4)(Vout5 Vtop5)(Vout6 >Vtop6))

  ((31     (ad (ab)(ae))   )                              --  1

   (Vtop2  (>V32)   )                                     -- 28
   (>V32   ((ad =V33) ((ad)(>V34)))   )
   (>V34   ((ac Vi41) (Vi42 (ac)))   )
   (=V33   (((ac Vi11) ((ac)(Vi11))))   )

   (>Vtop3 (((aa (ac)) (>V31 (aa))))   )                  -- 28
   (>V31   ((ad =V19) ((ad)(>V23)))   )
   (=V19   (((ac Vi10) ((ac)(Vi10))))   )
   (>V23   ((ac Vi12) (Vi13 (ac)))   )

   (Vtop4  ((ac)(ad (ab)(=V17)))     )                    --  2
   (=V17   (((ac Vi29) ((ac)(Vi29))))   )

   (Vtop5  (((>V30 (ab)) ((ac)((ab) ((Vi8)(ad))))))   )  --  4
   (>V30   ((ad =V18) (^V16 (ad)))   )
   (=V18   (((ac Vi9) ((ac)(Vi9))))   )
   (^V16   ((ac (Vi7)))   )

   (>Vtop6 (((aa (ac)) (>V28 (aa))))   )                  --  4
   (>V28   ((ad =V35) ((ad) (>V25)))   )
   (>V25   ((ac Vi39) (Vi40 (ac)))   )
   (=V35   (((ac Vi43) ((ac)(Vi43))))   )  )
```

Figure XL: MCNC Benchmark Circuit I7, I/O and Internal Vector Partitions

Output

```
<oa>                                    --  1
<ob oc>                                 --  2
<od..og>                                --  4
<oh..pi>                                -- 28
<pj..pm>                                --  4
<pn..qo>                                -- 28
```

Body

```
<1..28>                                 -- 28
<>33..>41>                              --  9 --|
<>42..>45>                              --  4    |
<>46..>64>                              -- 19 --|
<65 66>                                 --  2
<67..70>                                --  4
<^71..^74>                              --  4
<=75 =76>                               --  2
<=77..=80>                              --  4
<=81..=108>                             -- 28
<=109..=112>                            --  4
<=113..=140>                            -- 28
<>141..>168>                            -- 28
<>169..>172>                            --  4
<>173..>200>                            -- 28
<>201..>209>                            --  9--|
<>210..>213>                            --  4   |
<>214..>241>                            -- 28   |
<>242..>245>                            --  4   |
<>246..>264>                            -- 19--|
```

Input

```
<aa><ab><ac><ad><ae>                    --  1
<af ag>                                 --  2
<ah..ak>                                --  4
<al..bm>                                -- 28
<bn..bq>                                --  4
<br..cs>                                -- 28
<ct..cw>                                --  4
<cx..dy>                                -- 28
<dz..ec>                                -- 28
<ed..fe>                                --  4
<ff..fi>                                --  4
<fj..gk>                                -- 28
<gl..go>                                --  4
<gp..hq>                                -- 28
```

Figure XLI: MCNC Benchmark Circuit I7, Vector Expansion

ILOC vector body for the I7 benchmark circuit, expanded with vectors

```
((oa 31)(Vout2 Vtop2)(Vout3 Vtop3)(Vout4 Vtop4)(Vout5 Vtop5)(Vout6 Vtop6))

((31     (ad (ab)(ae))   )                                    --  1

 (Vtop2                                                       -- 28
    (((ad (ac Vi11) ((ac)(Vi11)))
      ((ad) (ac Vi41) ((ac) Vi42))))   )

 (Vtop3                                                       -- 28
    (((aa (ac))
      ((aa) ((ad (ac Vi10) ((ac)(Vi10)))
             ((ad) (ac Vi12) ((ac) Vi13))))))   )

 (Vtop4  ((ac)(ad (ab)(Vi29)))     )                         --  2

 (Vtop5                                                       --  4
    (((ab) (ad (ac Vi9) ((ac)(Vi9)))
           ((ad) (ac Vi7) ((ac) Vi8))))   )

 (Vtop6                                                       --  4
     (((aa) (ad (ac Vi43) ((ac)(Vi43)))
            ((ad) (ac Vi39) ((ac) Vi40))))   )  )
```

Figure XLII: MCNC Benchmark Circuit I7, Vector Modularization

Preparation

```
((oa 31)(Vout2 Vtop2)(Vout3 Vtop3)
        (Vout4 Vtop4)(Vout5 Vtop5)(Vout6 Vtop6))

((31       (ad (ab)(ae))   )                                  --  1
 (Vtop4    ((ac)(ad (ab)(Vi29)))     )                        --  2
 (Vtop2    (V91)   )                                          -- 28
 (Vtop3    (((aa (ac)) ((aa) V92)))   )                       -- 28
 (Vtop5    (((ab)(V93)))   )                                  --  4
 (Vtop6    (((aa)(V94)))   )                                  --  4

 (V91  ((ad (ac Vi11) ((ac)(Vi11))) ((ad) (ac Vi41) ((ac) Vi42)))   )
 (V92  ((ad (ac Vi10) ((ac)(Vi10))) ((ad) (ac Vi12) ((ac) Vi13)))   )
 (V93  ((ad (ac  Vi9) ((ac)( Vi9))) ((ad) (ac  Vi7) ((ac)  Vi8)))   )
 (V94  ((ad (ac Vi43) ((ac)(Vi43))) ((ad) (ac Vi39) ((ac) Vi40)))   )  )
```

Modularization

```
  ((oa 31)(Vout2 Vtop2)(Vout3 Vtop3)
          (Vout4 Vtop4)(Vout5 Vtop5)(Vout6 Vtop6))

  ((31      (ad (ab)(ae))   )                                  --  1
   (Vtop2   (V91-0)   )                                        -- 28
   (Vtop3   (((aa (ac)) ((aa) V92-0)))   )                     -- 28
   (Vtop4   ((ac)(ad (ab)(Vi29)))     )                        --  2
   (Vtop5   (((ab)(V93-0)))   )                                --  4
   (Vtop6   (((aa)(V94-0)))   )                                --  4

   (V91-  vector-module ((Va Vi11)(Vb Vi41)(Vc Vi42)) ((Vo V91-0))   )
   (V92-  vector-module ((Va Vi10)(Vb Vi12)(Vc Vi13)) ((Vo V92-0))   )
   (V93-  vector-module ((Va Vi9) (Vb Vi7) (Vc Vi8))  ((Vo V93-0))   )
   (V94-  vector-module ((Va Vi43)(Vb Vi39)(Vc Vi40)) ((Vo V94-0))   )  )

   ((vector-module)
    ((Va unk)(Vb unk)(Vc unk))
    ((Vo 1))
    ((1   ((ad (ac Va) ((ac)(Va))) ((ad) (ac Vb) ((ac) Vc)))   )  ))
```

Cell abstraction of the vector module body

```
        ((1   ((ad (ac Va) ((ac)(Va))) ((ad) (ac Vb) ((ac) Vc)))   )  )

  ==>   ((>1   ((ad =2) ((ad)(>3)))   )
         (=2   (((ac Va) ((ac)(Va))))   )
         (>3   ((ac Vb) ((ac) Vc))   )  )
```

Figure XLIII: MCNC Benchmark Circuit I7, Matrix Abstraction

Preparation

```
((oa 31)(Vout2 Vtop2)(Vout3 Vtop3)(Vout4 Vtop4)(Vout5 Vtop5)(Vout6 Vtop6))

((31       (ad (ab)(ae))   )                              --  1
 (Vtop4    ((ac)(ad (ab)(Vi29)))     )                    --  2
 (Vtop2    (V91)   )                                      -- 28
 (Vtop3    (((aa (ac)) ((aa) V92)))   )                   -- 28
 (Vtop5    (((ab)(V93)))   )                              --  4
 (Vtop6    (((aa)(V94)))   )                              --  4

 (V91  ((ad (ac Vi11) ((ac)(Vi11))) ((ad) (ac Vi41) ((ac) Vi42)))   )
 (V92  ((ad (ac Vi10) ((ac)(Vi10))) ((ad) (ac Vi12) ((ac) Vi13)))   )
 (V93  ((ad (ac  Vi9) ((ac)( Vi9))) ((ad) (ac  Vi7) ((ac)  Vi8)))   )
 (V94  ((ad (ac Vi43) ((ac)(Vi43))) ((ad) (ac Vi39) ((ac) Vi40)))   )  )
```

Matrix abstraction

```
((M1-    M1  <Vi11 Vi10 Vi9 Vi43>   )                   -- 28
 (M2-    M2  <Vi41 Vi12 Vi7 Vi39>   )                   -- 28
 (M3-    M3  <Vi42 Vi13 Vi8 Vi40>   )                   -- 28
 (M4-    M4  <V91 V92 V93 V94>   )                      -- 28

((31       (ad (ab)(ae))   )                                    --  1
 (Vtop4    ((ac)(ad (ab)(Vi29)))     )                         --  2
 (Vtop2    (M4-1)   )                                          -- 28
 (Vtop3    (((aa (ac)) ((aa) M4-2)))   )                       -- 28
 (Vtop5    (((ab)(M4-3)))   )                                  --  4
 (Vtop6    (((aa)(M4-4)))   )                                  --  4
 (M4       ((ad (ac M1) ((ac)(M1))) ((ad) (ac M2) ((ac) M3)))   )  )  -- 4x28
```

Cell abstraction of the ERC matrix body

```
((31       (ad (ab)(ae))   )                                   --  1
 (Vtop4    ((ac)(ad (ab)(Vi29)))     )                         --  2
 (Vtop2    (M4-1)   )                                          -- 28
 (Vtop5    (((ab)(M4-3)))   )                                  --  4
 (Vtop6    (((aa)(M4-4)))   )                                  --  4
 (>Vtop3   (((aa (ac)) ((aa) M4-2)))   )                       -- 28
 (>M4      ((ad =M5) ((ad)(>M6))   )                           -- 4x28
 (=M5      (((ac M1) ((ac)(M1))))   )                          -- 4x28
 (>M6      ((ac M2) ((ac) M3)))   )  )                         -- 4x28
```

Figure XLIV: MCNC Benchmark Circuit I7, Matrix Abstraction Schematic

Matrix schematic for the I7 benchmark circuit