Proceedings of ISCAS 85

The YORKTOWN Silicon Compiler

R.K. Brayton, N.L. Brenner, C.L. Chen,
G. DeMicheli, C.T. McMullen*, and R.H.J.M. Otten

IBM Watson Research Center, Yorktown Heights, N.Y.
*Harvard University, Cambridge, Mass.

ABSTRACT: *The goals of the Yorktown Silicon Compiler are*

- *completely automatic compilation and*
- *competitive designs.*

We describe the overall system, and sketch some of the internal algorithms. The system consists of

1. *a logic language*
2. *logic synthesis*
3. *layout*
4. *timing analysis and synthesis*
5. *functional and logical simulation*

The system is constructed to be easily extendable. The compilation step consists of the compilation of the logic "subroutines" and a linking operation that interconnects the pieces together. Some initial work on systems timing optimization is discussed. The layout design is based on flexible macros whose preliminary shape is determined by a floorplanning tool based on a slicing structure and data flow stacks. We also describe our experience with the design of a microprocessor using this system and some extensions which we have added because of this design experience.

Introduction: Interest in silicon compilers and the number of silicon compiler projects in universities and industry has been increasing each year. Many of these efforts have different goals and different ideas about what is a silicon compiler. The term is used for simple assemblers or design aids for building a VLSI chip, as well as efforts to work at a very high level language, involving expert system capabilities.

We use the name compiler in a way parallel to its software context. Thus a compiler starts with a language which is expressive in describing hardware. It is reasonably high level in the sense that the programmer need not know assembly code or the details of the final object code, in this case, the gates or the target technology, and the layout design rules. The compiler should compile this language automatically into a valid chip image.

The Yorktown silicon compiler starts with the view of a VLSI chip as consisting of logic blocks which are purely combinational logic, storage blocks such as latches and register stacks, and amplifier blocks such as off-chip drivers and receivers. This is illustrated in Fig. 1. Each of the blocks can be thought of as a subprogram and will be implemented as a rectangular module in the final layout. Timing of the storage devices is controlled by clock signals which are explicitly declared as signals in the compilation process. The programmers job is to partition the task that the chip must perform into the separate building blocks, and then describe each block. The main task here, after the partitioning is done, is to describe the combinational logic blocks. For this, we provide an expressive logic language, YLL, the Yorktown Logic Language.

Ultimately each block will be implemented by the compiler in a rectangular area. This is accomplished by a macro assembler which has the ability of building each logic block or storage block in a rectangular area of prescribed length to width ratio and prescribed positioning of the input and output signals. For logic blocks, the macro assembler can be compared to a PLA assembler which builds a rectangular area implementing a set of logic functions in a particular way. The macro assembler has several advantages over a PLA assembler. First, it can build the macro with a prescribed aspect ratio of the rectangular area. In addition, the macro assembler does not impose constraints on the input and output positions and has better control over the delay through the macro. Thus it can adjust the implementation to the global floorplan requirements. We will see also that the logic is implemented in the macro using a general multistage logic style as compared to the PLA which is confined to a two level sum of products implementation. Thus the macro assembler is equally useful in creating logic blocks for data flow as well as control logic, and can build the macros in a way that fits better into the global floor plan.

The compilation process is illustrated in Figure 2. It consists of

1. Extracting the logic equations from the logic descriptions (logic modules), YLL-LP.
2. Compiling each logic module into a topology of interconnected transistors and other devices, YLE.
3. Linking each compiled "subprogram" to create a global net list, LINKER.
4. Optimizing the delay by assigning device sizes, weights to drive the placement algorithms, and re-synthesis of critical macros, TIMER.
5. Deriving area and shape constraints for each module, LAYOUT DESIGN
6. Generating a global floorplan and assigning shapes and input/output signal positions for each macro, FLOORPLAN DESIGN.
7. Generating each macro with the assigned shape and IO positions, MACRO ASSEMBLY.
8. Completing the global wiring using channel routing procedures, LAYOUT ASSEMBLY.

The compiler process and the algorithms used in it are designed to be completely automatic. In the layout design phase of the compiler this requires that wirability is guaranteed; a working, fully wired, valid chip image is the end product.

A second objective of the compiler is to be able to create chip designs which are competitive with designs aided by manual intervention. The way this is achieved is by using algorithms better suited for the computer than for humans. In the compiler process these algorithms include

1. logic minimization
2. logic synthesis using complex gates
3. global floorplanning using flexible shapes
4. an integrated system where language, synthesis and layout communicate freely, and
5. timing synthesis and optimization at the system level

The Language: The hardware description language used by the Yorktown silicon compiler was designed to be easy to write and debug and with the primary goal of being a synthesis language rather than a simulation language. Since it was experimental, it needed to be easily extendable. We chose to develop a language for describing logic based on APL. The language, YLL, is simply an extension to APL obtained by embedding it in APL. It is described more fully in <1>. The objective of YLL is to describe logic functions. A YLL subroutine is a description of a set of logic functions with logic variables as inputs. In processing a YLL description, the YLL subroutine which looks very much like a regular APL subroutine is translated into a real APL subroutine. This APL subroutine when executed produces the logic equations stored in an internal data structure. Because the YLL processing consists merely of a translation driven by a translation table, one can extend the language by simply adding to this table. To make an extension to YLL, the user must create an APL subroutine which does the appropriate logic manipulation and creates the correct logic function stored in the YLL logic function data structure. Since the final running of the translated YLL source code occurs in APL, we obtain recursion for free. The full power of APL becomes part of the language by indicating, with quotes, sections of code which should not be translated.

YLL was designed on the principle that any operation allowed in APL which takes a set of 0's and 1's and produces another set of 0's and 1's is allowed. The YLL description looks exactly like the corresponding APL program. The only difference is that the APL program produces a numerical answer whereas the YLL program produces the logic functions which when applied to numbers would give the correct answer. Thus the YLL processor has symbolic output instead of numerical output; the symbolic domain of manipulation is the set of logic functions. Of course, not every APL operator or statement produces a 0,1 output. We have defined some of the regular APL operators to have meaning in YLL in the logic domain. Some constructs missing in APL like if..then..else were added also.

Variables in YLL are either primary logic variables or intermediate variables representing the output value of a logic function. Thus the assignment statements

$$C \leftarrow A \vee B$$

$$E \leftarrow C \wedge D$$

define an intermediate logic variable C which always has the value given by the logic function $A \vee B$. This variable is used in another logic function whose value is named E. Thus for each variable named explicitly in the language, there is either exactly one logic function associated with it or it is an input variable. We make the important requirement for the language processor, that its output reflect precisely a 1-1 correspondence between each variable named in the YLL source code and each logic equation produced by the YLL processor. This gives the "programmer" exact control of the logic equations produced, so he may choose to program at a high or low level. The initial logic equations produced by the language processor before logic synthesis are guaranteed to correspond precisely 1-1 with the input. A designer can force the logic synthesis system to leave the equations unaltered. This may happen if the designer prefers to program at the gate level. In any case, the designer will know the exact starting point for synthesis.

The names used by the designer in the source code are preserved during the compilation process as long as there remains a signal corresponding to the ori-

CH2114-7/85/0000-0391$01.00 © 1985 IEEE

ginal input. These names are used during logic simulation and timing analysis to aid the program debugging process. Of course, during logic synthesis, an intermediate signal may disappear; only the outputs of the logic macros will be guaranteed to be preserved.

Each logic macro is basically a subroutine. We have, for the present, chosen to pass the arguments to these subroutines through the use of global variables rather than passing them by position. During the linking process, two global nets are formed for each global variable mentioned in the set of subroutines to be linked. The positive signal net inherits the variable name; the negative signal name is obtained by underlining the variable name. Thus the variable *A* gives rise to nets (signals) *A* and $\underline{A}$.

A chip is described by listing the set of subroutines (logic macros) which are to be on it, and the latches, registers and amplifiers used. For these last three types, we pass the input signals by position. Thus a latch is described by listing

```
LATCH A_IN[0-31] A[0-31] CLOCK0
```

which denotes a group of 32 latches controlled by a clock signal *CLOCK*0. This also defines new variables *A*[0],...,*A*[31], $\underline{A}$[0],... $\underline{A}$[31] which are the outputs of this latch group. The signal *A*[0] will take on the value of *A_IN*[0] when enabled by *CLOCK*0.

The job of the VLSI chip programmer using this system is still a challenging one. He must partition the function into different facilities, determine the pipeline structure, latch boundaries, clocking scheme, interrupt mechanisms, etc. This must be described and debugged. The task of a software programmer is similar. We chose to start this compiler development at this intermediate level to focus initially on the compilation from this down to the chip image done in an automatic and competitive way. Future developments with the compiler will provide a higher level input and further high level design aids.

In summary, YLL has the following features:

1. Constructed for synthesis, not simulation
2. User control of results - one logic function for each variable
3. Extendable
4. Easily written and debugged
5. Recursive

Logic Synthesis: Logic synthesis is performed on the individual logic macros. There will be one logic macro for each YLL subroutine. The objective of logic synthesis is to take the outputs of each macro and create a topological implementation in some specified target technology. By a topological implementation, we mean a directed graph where each node is a gate with inputs and an output. An edge of the graph corresponds to the output of one gate being used as an input to the following gate. The type of each gate is either given implicitly by reference to a finite library of gates, or is given explicitly by another graph detailing the interconnection of transistors and other devices which are used to construct that gate.

Many logic synthesis systems use a data structure which is gate oriented. Each node must be one of a usually small set given in the library of the technology. As synthesis proceeds the directed graph of gates is manipulated, by a set of local transformations. After each transformation, the new graph must be of the same type, one where each node is an acceptable gate. This requirement limits the manipulations which can be performed, in that it must always result in a legitimate graph as defined by that target technology. The set of transformations is restricted to those which map from one legitimate graph to another legitimate graph.

The synthesis part, YLE, of Yorktown silicon compiler has adopted another philosophy. This corresponds closely to what is done with the logic language. The data structure is also a directed graph but each node is only restricted to be a logic function. Each logic function is represented in disjunctive form. Each edge of the graph is the same as above, denoting that the output of one node is used as an input of the following node. Here output means that it defines a new variable and input means that a node (logic function) uses this variable. Synthesis proceeds by transforming this graph into another graph with the same characteristics. Since they are not confined to any particular small set of gates, the set of transformations on this structure is independent of technology. This structure includes all technologies, so it is more general and allows more powerful transformations.

Synthesis in YLE has two stages, one independent of the target technology, and the other dependent on the technology and ultimately producing a legitimate graph. The first stage contains general operations on the graph which can be used for synthesizing into any technology. To produce a synthesis procedure for a new target technology, it is only necessary to change a few procedures of the second stage. This has been done for cascode emitter coupled logic (CECL), standard CMOS, differential cascode current switch (DCCS), and domino CMOS dynamic logic (SCVS) technologies. We have emphasized the latter technology in our compiler experiments to demonstrate the advantages of using a complex cell as the basic gate and to provide one source of leverage over manual designs using standard gates.

The technology independent synthesis process is described in detail in another paper <3>. It is simple in one sense, since it uses logic minimization techniques similar to those developed for PLA synthesis <20>. The synthesis process is based on the sequential application of the procedures,

1. collapse
2. simplify and
3. extract.

The collapsing part makes the logic more compact so that "local" transformations are more global. After collapsing, each node of the graph will usually become a larger logic function, with many of the nodes being eliminated, resulting in a smaller graph. This prepares the graph for the simplification process which works on each node logic function using standard logic minimization algorithms <19,20>. Logic minimization serves as the "local" transformation process, but the locality has been expanded by the collapsing process. The domain of the logic minimization is also extended by using the inputs of a node to be minimized to define a don't care set. After simplification, the extraction process is applied. Its purpose is to find common subexpressions which can be extracted from two or more nodes <2,3>. All such common subexpressions which can be found by our factorization procedure and which have sufficient "value" are extracted. Typically, the entire procedure collapse, simplify, extract will be repeated. At the end, the target technology is brought in to decompose the nodes further into legitimate gates. Local transformations can be applied at this point to continue the synthesis process, but in our experience, most of the gain obtainable by synthesis has already been achieved.

The synthesis process can be "programmed" by giving a sequence of transformations to be applied to the graph, and a set of global parameters controlling the amount of collapsing and extraction is allowed. Thus synthesis can be tailored to various kinds of logic.

Input to the synthesis program is the form as written out by YLL, i.e. a set of logic functions stored in a particular data structure. The output has the same form and hence can be read by YLE or YLL-LP for other purposes.

The Complex Cell: The initial experiments with the compiler involve the use of a complex cell to implement the logic function for each node of the synthesized graph. Standard gates could be used, in which case, the output of the synthesis process would be a graph whose nodes are logic functions each implemented by a gate in the library of gates. In this situation, the gates are predesigned, and the synthesis process is constrained to use only this set of predesigned gates.

The use of the complex cell does not require any predesigned gates. The rules for a legitimate gate are described parametrically. For example, any logic function is allowed as long as it requires no more that 5 transistors in series. As the synthesis process proceeds during the technology dependent stage, each node logic function is tested for legitimacy, and in the end, each is guaranteed to be allowed according to the parameters given. This has the advantage that the synthesis process is able to choose the most appropriate gate for the particular situation, usually leading to a more compact logic implementation. At the end of the synthesis process, a subset has been selected from the family of allowed gates, and it is now the task of synthesis to construct a topological implementation of each such selected gate. Thus synthesis is not restricted to some arbitrary small set of gates. Since each of the complex gates can implement a rather large logic function in a single gate, both wiring area, and the delay through the logic are decreased.

Our complex cell is based on the domino CMOS cell described in <4,5>. It is constructed by first factoring <2> the logic function associated with a node, and then building a series-parallel graph. For example, the expression AC+AD+BC+BD can be factored as (A+B)(C+D) which leads to a series-parallel graph of transistors with A and B in parallel, and C and D in parallel, and these two parallel constructions in series. From this initial series-parallel graph, we construct a cell implementing the function by finding an Euler path through the graph. The transistors are then laid out in a straight line in the Euler path sequence. First-level metal is used to tie the nodes of the graph together. The entire procedure from the factored logic function to the layout is shown in Figure 3. The cell is constructed with small area since the transistor connections between source and drain are made by abutment in the diffusion. Much of the wiring is done in diffusion. Because the transistors are abutted we ensure minimum spacing between transistors. Edge effect capacitance is minimized since most diffusion edges are eliminated. Thus the complex cell provides two important features. First, the logic functions of the cells are chosen to conform to the requirements of the logic, and second the cells are constructed by a computer algorithm using minimum geometric constraints, resulting in small area and good speed.

The Macro Assembler: The macro assembler is based on a standard cell style <6>. The cells that have been constructed above, are arranged in rows for the purpose of bussing power, ground, and the clock signals through the macro. Placement of the cells into their position in the columns is done using annealing

<7,8,16,17>. We use a scoring function which takes into account the total wiring as well as critical timing requirements. Wiring is completed by using a greedy channel router <9> in the channels between the rows. Each wiring channel is allowed to be whatever width is required to completely wire the channel. Experience with this channel router indicates that most channels can be wired close to density.

The sides on which the inputs and outputs arrive at the macro are assumed to be given at the time the macro is built. This information is obtained by the global floorplanning of the chip. For inputs and outputs arriving at the top or bottom of the macro, second level metal positions are assigned previous to the construction of the macro. For inputs and outputs arriving at the sides, wiring will be achieved using the wiring channels of the macro. First level metal and poly are used in the macro wiring channels, while second level metal is used only for inputs, outputs, and interwiring between the channels.

The macro assembler can be given a desired aspect ratio for the the area implementing the macro. The macro assembler will then choose the number of rows in the macro to achieve this aspect ratio as closely as possible. For a given aspect ratio, we estimate the amount of area based on the number of gates contained in the macro, the gate types, and their connectivity. The ability to accurately estimate the size of the macro is important in generating the floor plan. We will see that the macro size estimates and the ability to flexibly build macros are key components in designing a small chip.

It is important to emphasize that for those macros to be built by the macro assembler, the actual construction is done only in the final phase of the compiler. The global floor planning is done first in a top down manner, and only when the floor plan and rough global wiring are completed is the actual macro construction done. Of course, preconstructed macros contained in libraries will at times offer benefits. However, these macros have no flexibility, so floorplanning will be more constrained, possibly offsetting the benefit gained by the hand-honed designs.

The Linking Process: When all of the "subroutines" have been described and topologically synthesized into the prescribed target technology, they are linked using a process very similar to that used in software compilers. The subroutines or macros can be thought of as being partially compiled at this stage. Their inputs and outputs must be linked. Basically the linking process forms a net list describing the interconnectivity of all the signals. The net list data structure has been designed to store additional information for the layout process. This data structure is used to drive the layout design and contains information about powering output devices, and desired short paths for timing. This data structure is created by the linking process and is written into an intermediate 'connectivity file for use by other procedures as illustrated in Figure 2.

The compiler should be a strictly automatic process. However its usefulness will depend on a compatible debugging environment where the "program" can be run and errors corrected. We have provided a number of facilities for this. Once the linking is completed, the chip design is audited, AUDIT, to make sure that each signal is connected properly. Logic simulation, SIMER, is done at the chip level. Since each macro has been compiled down to the gate level, logic simulation is based on an internal logic simulation of the macros and the net list connecting the macros. In our case this simulation is done in APL and thus is aided by the interactive debugging environment of APL. As the chip is debugged, some of the macros will need to be corrected. These are changed at the YLL source code, and the changed macro is recompiled i.e. reprocessed by the YLL processor, YLL-LP, and the synthesizer, YLE. It is necessary to relink only the changed macro in order to proceed with the simulation. During simulation, it is also possible to replace any macro by an APL functional description, i.e. an APL subroutine. We have used this technique during the design of a microprocessor for functionally simulating its 32 word register file and external memory.

This creates a interactive environment for easy debugging of the chip at the chip level. We have been able to simulate a design of a 32 bit microprocessor at this level using about .9 sec. of CPU time for each machine cycle simulated. Debugging is done by inputting a set of assembly code instructions, "running this program", and following the effects of the instructions through the different stages of the pipeline. Many of the initial bugs in the chip description are found at this stage. For faster simulation during the final design stages, it is necessary to link to a higher speed logic simulator or a logic simulation engine.

It is also useful at this stage to analyze the timing of the chip design. For each gate we estimate its internal delay using a delay equation derived by electrical simulation of various gates. By adding an output capacitive loading term to the delay we estimate the arrival time of the output of this gate at its destinations. Estimation of wiring capacitance uses rapid placement algorithms <10,11>. Later in the design phase we replace this with placement information from the layout design phase to better estimate the wiring capacitance. Starting with the outputs of the latches, the delay for each of the signals is calculated taking the maximum of the arrival times for the inputs of a gate and adding the gate delay and net delay. At this stage, the system's critical path is determined, slacks computed and a general system timing optimization procedure is begun. We have experimented with using the global signal delays as inputs to the logic synthesis in order to guide the synthesis into a more parallel decomposition along the critical path. The early arriving signals are extracted and intermediate signals based on them are 'precomputed' so that they are ready when the late signals arrive. In some initial experiments, this was used to decrease the system critical path time by about 20 percent. We have started the design of a general system timing optimization procedure. Our objective will be an algorithm which uses placement, device sizing, and parallel re-synthesis to minimize the system delay.

Global Floor Planning: Global floorplanning proceeds in two stages. The first stage is concerned with building a dataflow stack. In the description listing which "subroutines" are on a chip, we allow an indication of bussing structures. This is information purely for the layout design phase. In the chip description a statement of the type

```
BUS CARRY[0-31]
```

indicates a 32 bit vector of signals which are to be treated by the layout design as a single entity with 32 signals in a particular order. Any subroutine or macro connected to any one of these busses is assigned to the dataflow stack. The dataflow stack is designed by solving a one dimensional placement problem using the algorithm of Asano <12>. The dataflow stack is built so that all busses are routed over the macros using straight lines in second level metal. The one dimensional placement solution allows this stack to be made using minimum width by specifying the sequence of macros in the stack. The bus signals are then assigned to second level metal tracks, with the 0th bits occupying an interval on the left, the 1st bits the next interval, and the 31st bits the last interval. The width of the dataflow stack is determined by either the number of second level tracks required, or by the widest macro if such a macro comes from a library and is not flexible. Once the width and bus positions are known, we use the macro assembler to construct each macro in the dataflow stack with the required width and with the inputs and outputs on preassigned second-level tracks. The advantages of using a dataflow stack are smaller area, because of straight line routing of most of the global wires, and better delay, because of the reduced capacitance of second level metal.

The dataflow stack is considered as another macro which must be fit into the general floorplan. The method used for the general floorplan design is based on a slicing structure <13,14,15> illustrated in Figure 4. A slicing structure is a rectangle which has been sliced into smaller rectangles, each of which are possibly sliced etc. It corresponds to a tree. The root node represents the overall rectangle with the first level nodes the rectangles constructed by slicing this rectangle vertically any number of times. Each of these are then sliced horizontally any number of times etc. Each slicing line will ultimately be enlarged into a wiring channel for effective global wiring. It can be shown that only global floorplans with slicing structures can guarantee that there will be no conflicts in ultimately wiring the chip with channel routing algorithms. Thus this structure is derived from the requirement that the compiler should be completely automatic.

A slicing structure is obtained in two phases. First a point placement procedure places points representing macros in the plane. The placement is guided by the objective that macros with many common nets should be close, and that nets on critical paths should be short. This point placement is done by annealing <16,17> where the point placement is derived from an ordering of the macros in the x direction and an ordering in the y direction. An interval proportional to the area required for the macro is assigned to the x axis and y axis in the sequence given by the orderings. From this the point placement of each macro is derived. This is illustrated in Figure 7. Different point placements are derived by new x and y orderings obtained by pairwise interchange. This pairwise interchange is controlled by the annealing algorithm <16>.

Once an optimal point placement is obtained, the slicing structure is derived by overlaying each point by a square with the required macro area. These squares are shrunk uniformly until a slicing line is obtained. This is illustrated in Figure 8. This process is repeated recursively until a full slicing structure is obtained where each leaf node is a macro.

A rectangular area in the floorplan which has been sliced into smaller rectangles is referred to as a compound module. A rectangle which is not sliced is called a leaf module. In the ultimate floorplan, the leaf modules will be occupied by macros built by the macro assembler, or taken from libraries, or created by the dataflow stack construction. Having obtained a slicing structure for the floorplan, we next assign lengths and widths to the macros to be built. Here we use the flexibility constraints. For each macro we have a curve describing the size requirements for the macro. Several such curves are shown in Figure 6. If all we know is the area required, then the hyperbola for this area is given. Any point (x,y) lying above this curve is allowed. Any shape constraint graph which is nonincreasing is allowed. A constraint that a macro has already been built and has no flexibility can be accepted also. Each compound module has its own shape constraint graph derived by adding the constraints of the sub modules composing it. This procedure is illustrated in Figure 6. By starting with the leaf nodes, the shape constraint graph of the entire chip, the root node of the slicing tree, is derived. The point on the shape constraint graph which minimizes the chip area, or some other objective function, is chosen. This determines the outer dimensions of the entire chip. From this, by recurring down the slicing tree, it is easy to derive the dimensions of each leaf module.

Next we proceed to the rough global wiring. Each global net which must be routed is assigned to a channel of the slicing structure as illustrated in Figure 5. This is done heuristically, minimizing wiring lengths and critical path delays <18>. At the end of this process, each global net is assigned to various segments of the slicing channels. Thus the density of each channel is known, and is used to estimate the channel widths. Each channel is then treated as a rectangle with a shape constraint graph given by requiring the desired width. Thus each area of the floorplan is a rectangle obtained from a module or a channel. The entire structure is still a slicing structure, and each rectangle has an associated shape constraint graph. We now re-optimize the floorplan according to our objective function using all the shape constraints.

At this stage global floor planning and rough global wiring are complete. We know for each global net, which slicing channel it is assigned to, and thus which side of the macro it is arriving for connection. For each macro, we know the size of the macro which best fits into the floorplan. Thus we now call the macro assembler to construct each macro. Finally, the channel routing is done in a bottom up manner using standard channel routing algorithms. Because of the slicing structure, we are guaranteed that this will complete the wiring without any conflict.

Experience with Designing a Microprocessor: We decided to build a reasonably complicated microprocessor using this system in order to test the compiler on a large design. We also wanted to gain experience ourselves in system design to understand better the need for higher level specifications. Some of what has been described already about the compiler has been influenced by this design experience. In this section, we illustrate several ways in which the ability to extend the language YLL aided us in obtaining a better design.

The control and interrupt mechanisms are the most difficult part of microprocessor design. We decided that our control description, in YLL, of the microprocessor should be table driven. After determining the structure and various facilities of the design, we constructed tables, where each row corresponds to one microprocessor instruction, and each column to a facility. The entries in the columns are filled in with a mnemonic indicating what action the facility should take for that instruction. If a facility is not used for a particular instruction, a '?' is entered in the table. The use of the tables led to a 'clean' description of the control which was easy to debug. The tables are created as character matrices in APL and are accessed directly using an APL subroutine which was written for that purpose. Thus to extend YLL so that it could be table driven for this particular application, only one APL subroutine was required.

A second extension was made also. Each instruction opcode for this microprocessor is represented by 8 bits. Since there were only about 130 instructions, nearly one half of the opcode points were not used (representing illegal opcodes). We decided to allow the designer to use these don't-care points to obtain simpler control functions. The don't-care points consist of illegal opcodes as well as the legal opcodes with a '?' in the column for the facility being controlled. There are quite a few of these '?' code points since for many instructions, a particular facility will not be used. For example, for most of the rotate and shift instructions, the ALU is not used and vice versa. Again we extended YLL by allowing the user to specify don't-care logic functions. We created an APL subroutine which simplifies a specified logic function relative to a given don't-care function. The user of YLL simply states which functions he wants simplified relative to which don't-care function. This extension was easy since it just required a small modification of some APL subroutines which were already available as part of the synthesis package, YLE, <3>.

A more subtle influence of this design exercise, was the need to debug our YLL description. Since the chip design was complicated by a four-stage pipeline design and the related interrupt mechanisms, we needed to provide a debugging environment which aided the debugging process. Thus the various auditing and simulation mechanisms were expanded and improved as we got more involved in the chip debugging.

The use of minimization relative to a don't-care function made most of the control functions very simple. This led us to postulate a distributed control structure for the chip. In this structure, the control signals for each facility are derived in a logic macro placed next to or even embedded in the macro for that facility. The eight bits required to represent the instruction opcode are bussed in a way similar to the busses of the dataflow stack. Compared to a central control facility, the distributed control structure saves in global wiring, and because each control function has been made small through the use of a don't-care set, we can afford to decode the controls locally next to or inside the facility it controls.

Conclusions: The Yorktown silicon compiler has derived its design from its two primary goals, automatic compilation and competitive chip designs. The algorithms have been chosen so that they are guaranteed to complete without conflict or the need for intervention, and they offer leverage as computer algorithms over manual procedures. A key part of the system is the ease with which it can be extended, thus offering future extensions as algorithms are improved. This has been justified in our experience with compiling a microprocessor, where some easy extensions have led to an improved chip design.

We also feel that the compiler structure and language are sufficiently general to allow a wide range of different types of digital chips to be designed with relative ease. The input to the compiler is at a medium level but it does allow the user to accurately control the results if desired. Future development of various higher level languages for different applications could use YLL as an intermediate object language. Other kinds of compiler structures may still be necessary. For example, systolic arrays need compilers which tile the chip area with images designed for a particular task. There is little control and wiring is by abutment. It remains to be determined how such a requirement can be fit into the Yorktown compiler structure.

Acknowledgements: The compiler started with our efforts in logic minimization and synthesis and is a result of the efforts of many people over the last few years. We would like to thank L. van Ginneken, G. Hachtel, J. Katzenelson, M. Lightner, R. Rudell, A. Sangiovanni-Vincentelli, E. Talsma, and Y. Yamour for many helpful discussions and ideas.

References:

1. Brenner, N., The Yorktown Logic Language: an APL-like Design Language for VLSI Specification, ICCD84, Portchester, NY, 1984.
2. Brayton, R.K. and McMullen, C., The Decomposition and Factorization of Boolean Expressions, ISCAS Proceedings, Rome, April 1982.
3. Brayton, R.K., McMullen, C.T., Synthesis and Optimization of Multistage Logic, ICCD84, Portchester, NY, 1984.
4. Chen, C.L., Otten R.H.J.M., Considerations for Implementing CMOS Processors, ICCD84, Port Chester, NY, 1984.
5. Krambeck, R. H., Lee C. M., and Law, H. S., High-Speed Compact Circuits with CMOS, IEEE J. Solid-State Circuits, Vol. SC-17, No. 3, June 1982.
6. Brayton, R.K., Chen, C. L., McMullen, C.T., Otten, R.H.J.M., and Yamour, Y. Y., Automatic Implementation of Switching Functions as Dynamic Cmos Circuits, CICC 1984.
7. Kirkpatrick, S., Gelatt, Jr. C. D., and Vecchi, M. P., Optimization by Simulated Annealing, Science, May 13 1983.
8. Otten, R.H.J.M., Ginneken, L.P.P.P. van, Annealing: the Algorithm, submitted to IEEE Transactions on CAD, 1985.
9. Rivest, R. and Fiduccia, C.M., A Greedy Channel Router, Proc. 19th Design Automation Conference, June 1982, pp. 428-424.
10. Otten, R.H.J.M., Automatic Floorplan Design, 19th Design Automation Conference, June 1982, pp. 261-267.
11. Hall, K.M., An r-Dimensional Quadratic Placement Program, Management Science 17, 1970 pp.219-229.
12. Asano, T., An Optimum Gate Placement Algorithm for MOS One-Dimensional Arrays, J. Digital Systems, VI, No.1, pp. 1-28.
13. Otten, R.H.J.M., Layout Structures, Proc. 1st IEEE Large Scale System Symposium, 1982.
14. Otten, R.H.J.M., Efficient Floorplan Optimization, ICCD 1983, Port Chester N. Y.
15. Ginneken, L.P.P.P. van, Otten, R.H.J.M., Stepwise Layout Refinement, ICCD84, Portchester, NY, 1984
16. Otten, R.H.J.M., Ginneken, L.P.P.P. van, Floorplan Design by Annealing, Design & Test, 1985.
17. Otten, R.H.J.M., Ginneken, L.P.F.P. van, Floorplan Design Using Simulated Annealing, Proc. ICCAD 1984, Santa Clara, Cal, 1984.
18. Ginneken L.P.P.P. van, Otten, R.H.J.M., Global Wiring for Custom Layout Design, Proc ISCAS 1985, Kyoto, 1985.
19. Brayton, R.K., Cohen, J., Hachtel, G.D., Trager, B. and Yun, D.Y.Y., Fast Recursive Boolean Function Manipulation, ISCAS Proceedings, Rome, April 1982.
20. Brayton, R.K., Hachtel, G.D., McMullen, C. and Sangiovanni-Vincentelli,A. Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, Boston 1984.