Design Process Model in the Yorktown Silicon Compiler

Raul Camposano

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

The automatic synthesis of an IBM 801 processing unit using the Yorktown Silicon Compiler is presented. The underlying design process model is explained showing the intermediate design stages, while emphasizing high-level issues. First, the principles of operations are translated manually into a high-level behavioral description. The system is decomposed by the designer into concurrent modules (in the 801, four pipeline stages). Structural synthesis automatically generates a circuit structure for each pipeline stage, including the control and the data path. The combinational logic is optimized globally during logic synthesis producing a multi-level implementation. The resulting size (in number of transistors) and the performance of the processor (estimated cycle time and cycles per instruction) are compared to a manual RT-level design.

1. Introduction

This paper presents two main ideas. First, it illustrates the design process model of the Yorktown Silicon Compiler (YSC) explaining the basic decisions taken at each design stage, as exemplified by the design of an IBM 801 processing unit. Second, the results of automatic synthesis of a realistic, fairly large example such as the 801 are interesting enough to merit consideration. These results are compared to a manual design, and conclusions about the virtues and defects of automatic synthesis are inferred.

The YSC [1-6] is described in detail elsewhere, including references to related work. The design process with the YSC is demonstrated starting with the manual steps necessary at a high level to obtain a competitive design. The automatic synthesis steps at different stages of the design are explained, with a clear emphasis on high-level design issues, particularly on structural synthesis (often called behavioral or high-level synthesis). The YSC automates the design process to a large extent. Like most silicon compilers, the YSC uses a fixed design process model where different parts communicate using particular data formats. In the YSC, the formats are not too numerous and most formats have also a textual form which facilitates interfacing with other tools. But the design environment lacks desirable properties, such as uniform interfaces, central data management using state-of-the-art data base technology, flexible design process automation, etc. (such issues are addressed for example in [7]).

Section 2 gives a brief review of the 801 architecture showing the impact of the (given) architectural design on the machine organization, hence on the hardware and its design. The next section is devoted to the system description. The given architecture, described in natural language in the so called "principles of operations", is converted into a formal, high-level behavioral description. This description is then manually decomposed further into concurrent modules. For the 801, decomposition into concurrent modules means the design of the pipeline.

Section 4 reviews the main tasks of structural synthesis in the YSC, explaining how the design is affected by each of the automatic transformations performed. The results of structural synthesis include the number of latches, the number of control states and the generated combinational logic. The combinational logic is optimized using logic synthesis. The results of logic synthesis given as gate count, transistor count and maximum estimated delay are commented in section 5. To complete the picture of the design process model, timing optimization and layout are mentioned briefly. In section 6 the resulting design is compared to a manual design at the device level and in terms of performance.

2. The 801 Architecture

Computer architecture is used to refer to the high-level rules that define the machine conceptually as viewed by the user (instruction set, registers, interrupts, etc.). In spite of the clear separation of the architecture from a particular machine organization (i.e. a particular implementation) on one side, and an operating system and software on the other side, an architecture is usually designed having these two aspects in mind. For hardware synthesis the machine organization is extremely important; failure to synthesize a machine organization in the spirit of what the architects foresaw will usually lead to a poor performance or an excessively large design.

The 801 architecture [8, 9] is a true 32-bit architecture in the sense that most instructions, data, registers and addresses are 32-bit wide. The 801 has 32 general purpose registers allowing an efficient register utilization by a compiler. All operands are aligned according to their size. Only a few instruction formats are used, each instruction being one word (32 bits) long and correspondingly aligned. All instructions can execute in one cycle.

The 801 architecture is often referred to as a RISC (Reduced Instruction Set Computer) architecture. A more proper characterization is a streamlined architecture. This emphasizes the instruction simplicity, the single-cycle execution and the register utilization rather than the number of instructions.

The most important implications of the 801 architecture on the machine organization are:

- To provide enough hardware to allow the single cycle execution of all instructions. This implies that an ALU capable of 32 bit arithmetic and shift/rotate is necessary, and that the general purpose register file has enough ports to allow all required accesses within one cycle.
- A store-in cache for the data bus and a separate instruction cache are necessary to minimize the idle time of the processing unit due to storage access.
- General performance considerations lead to a pipelined implementation. Further analysis quickly leads to a "natural" three-stage pipeline (more on this later). In the YSC, each

CH2540-3/88/0000/0489$01.00 © 1988 IEEE

concurrent module has to be specified as such, thus the pipeline is decomposed into concurrent modules manually.

The 801 is a good application for silicon compilation and synthesis since it is a real example large enough to test the tools, yet simple enough to be well understood by the toolmakers. Only the design of the processing unit is considered here. The cache and the cache controller are not part of this exercise.

3. System Description

The specification of a design in the YSC is given in a high-level, behavioral domain, imperative (procedural, sequential, e.g. Pascal-like) language. In the YSC we have chosen the V language to be compatible with other activities within IBM [10]. The first task the designer faces is to translate the 801 architecture given in natural language in the so called "principles of operations" manual, into a formal specification in V.

The initial 801 description is a single sequential program as shown in fig. 1. The modeling technique used to translate the architecture into a sequential program is straight-forward. The program consists essentially of three procedures in the following sequence: instruction fetch, instruction execute and check for interrupts. Instruction decoding (in execute) is modeled as one large CASE statement. The general purpose registers are represented by an array of 32 elements. Architected registers are represented by variables in the program. Although by far the largest part in the description is the procedure P8EXE, the most difficult aspect to model in this description is the interrupts. The instructions themselves can be coded in a straight-forward manner from their natural language description.

Such a sequential description proves to be extremely useful in documentation and in understanding the architecture in all details. For example, using standard software tools such as an editor, it is easy to locate all instructions setting a particular condition code. The sequential description is approximately 1500 lines long, containing 1380 statements.

In principle this initial non-pipelined behavioral description can be used for structural synthesis, but the results would be poor. The sequential nature of the description would produce a sequential implementation with low performance. This version is synthesized only partially as an exercise (see next section).

Ideally, any behavioral description results in an optimal design after using a high-level silicon compiler, but in practice this is not so. The issue of synthesizing automatically an appropriate high-level design from an arbitrary sequential specification is one of the most difficult ones in high-level silicon compilation. It seems difficult to envision, at present, a silicon compiler which is able to find the most suitable design organization for all kinds of designs. Some approaches which are used include interaction with the designer, so called "tuning-knobs" which set design parameters, objective functions, automatic design-space exploration and the use of fixed machine organizations. In the YSC, different specifications yield different designs in a predictable way. The specification includes high-level design decisions such as the pipeline, separate data and instruction buses, the necessary general purpose register file ports, etc.

A design synthesized by the YSC maintains several important characteristics of the initial specification:

- The structure given in V by the partition into sequential procedures and concurrent tasks.
- The loops and the partial order resulting from data dependencies.
- The specified ports. Procedure parameters are considered ports.

```
MODULE P801
 ... declarations ...
 BODY P801;

  MODULE P8RI                     /* Read instruction    */
   ... declarations ...
   BODY P8RI;
   ...
  END P8RI;

  MODULE P8EXE                    /* Execute instruction */
   ... declarations ...
   BODY P8EXE;
    IF ¬((IR::0,6)=63)            /* String at bit 0,    */
     THEN                         /*          length 6 bit */
      WHEN  (IR::0,6)             /* Simple opcodes      */
       CASE 17;                   /* L  RT,D(RA)         */
       ...
       CASE 16;
       ...
      ENDCASE;
     ELSE
      IF ¬((IRT::21,4)=0)
       THEN
        PGM_FAULT:=1;             /* Undefined opcodes   */
       ELSE
        WHEN (IRT::25,7)          /* Extended opcodes    */
         CASE 17;                 /* LX RT,RA,RB         */
         ...
         CASE 16;
         ...
        ENDCASE;
      ENDIF;
    ENDIF;
  END P8EXE;

  MODULE P8CHI;                   /* Check interrupts    */
   ... declarations ...
   BODY P8CHI;
    ...
    IF MACHINE_CHECK              /* Highest priority    */
     THEN
     ...
     ELSE
      IF DATA_STOR                /* Second priority     */
       THEN
        ...

      ENDIF;
    ENDIF;
  END P8CHI;

  ... initializations ...         /* Main program        */
  DO INFINITE LOOP
   ...
   P8RI;
   IF  ¬RI_INTERRUPTS THEN P8EXE;ENDIF;
   P8CHI;
  ENDDO;

 END P801;
```

Figure 1. Straight-forward 801 specification

- Only the specified architectured registers (not to be confused with all registers).
- The specified width of the variables.

Keeping all the above in mind, the initial specification was rewritten to meet the main implications of the machine architecture on the design. Notice that the changes only involve the overall machine organization. For example, the procedure P8EXE could be reused with almost no changes. The following list includes the main implicit and explicit high-level design decisions reflected in the specification:

- The pipeline, the pipeline control and the communication among the pipeline stages are specified.
- Single cycle execution results from scheduling operations as-fast-as-possible (not ASAP) to obtain the minimum number of control steps per instruction. Often the trade-off between the number of control steps per instruction and the amount of hardware (also the cycle time) is explored during scheduling and hardware allocation (e.g. [11]).
- The number of ports in the general purpose register file (GPR) is specified to allow enough accesses to ensure single

cycle execution. For example, the ideas contained in [12] allow one to automate multi-port memory allocation.

- A 32-bit ALU and a shifter/rotator are obtained by specifying 32 bit operands. Most high-level synthesis systems do this, an exception being [13], specialized in digital signal processing in which case the above choice would often be disastrous.
- Two separate caches for instructions and data are achieved by specifying separate ports for the two caches.

The resulting pipelined specification is sketched in fig. 2. Notice that still some degree of freedom is allowed by the architecture. In fact, the 4 stage pipeline decomposition is a different design than the architects had in mind, as can be seen from [8, 9].

```
MODULE P801P0
/* PREFETCH.
     Read instruction.
     Increment the program counter or load it
     with branch or interrupt address.
     Initiate the fetch of the next instruction.       */
END P801P0;

MODULE P801P1
/* DECODE.
     Decode the instruction.
     Load the appropriate registers
     from the general purpose register file.
     Initiate reading on the data bus.                 */
END P801P1;

MODULE P801P2
/* EXECUTE.
     Execute the instruction and latch the results.    */
END P801P2;

MODULE P801P3
/* STORE.
     Write results into general purpose register file.
     Initiate writing on the data bus.                 */
END P801P3;

MODULE P801P3
/* CONTROL.
     General pipeline synchronization.
     Disable pipeline overlap.
     Flush and reinitiate pipeline.
     Generate interrupt addresses.                     */
END P801P3;
```

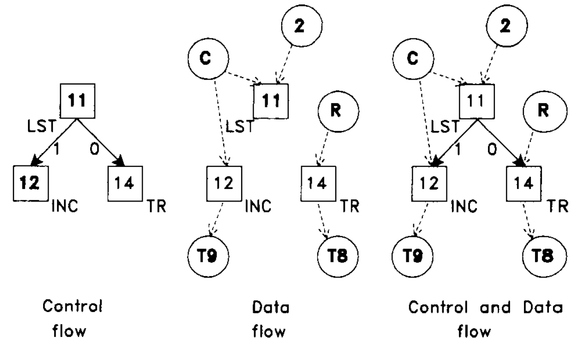
Figure 2. Pipelined 801 specification

The system is now described by 5 concurrent designs; the four pipeline stages and one module for pipeline synchronization and for handling pipeline overlap disable, taken branches, etc. Each module is specified as a sequential program. The most difficult aspect is the pipeline synchronization. Among other things, the control module must be able to generate several control signals within one cycle, before new results are latched at the end of the cycle destroying the processor state. The decomposition into a pipeline results quite naturally. Abandoning the sequential description style resulted in a straight-forward partition into pipeline stages. The complete description in this version is 1460 statements long. The results of its complete synthesis are given in the subsequent chapters.

4. Structural Synthesis

For the 801 example in this and the following chapters, all compilation and synthesis times are given in CPU seconds on an IBM 3090-200 computer. The V compiler is coded in PL/1, structural synthesis and logic synthesis are coded in APL (interpreted).

The first synthesis step is compilation. Compilation transforms the V specification into an internal format consisting of a data flow graph and a control graph. The main tasks of compilation include the decomposition of expressions into single operations and the mapping of complex data types into simple ones. Compilation also computes module interfaces considering all explicit parameters and all variables used in different modules according to their scope. Compilation is well known and fast; it took around 280 CPU seconds for the complete 801 example. The result of compilation is a so called YIF file (Yorktown Internal Format, fig.3). YIF is described as two directed graphs: the control graph and the data flow graph. The control graph represents operations such as additions, ands, etc. as nodes. The arcs indicate the predecessor-successor relationship. Fork nodes allow the selection of one among many successors (IF, CASE). Cycles in the graph model LOOPs in the specification. The data flow graph nodes are the same operations of the control graph plus the variables in the program. Arcs go only from operations to variables (indicating that the variable is an output of the operation) or from variables to operations (indicating that the variable is an input of the operation). YIF graphs are hierarchical in the sense that an operation node can represent another YIF graph. This operation is called "module call". YIF is used as the internal model during most of structural synthesis.




```
INDEX 11 TAG CBR OPERATION LST LINE 678;
INPUTS C(1..2)[0..0] 2;
OUTPUTS *;
PREDECESSORS  10;
SUCCESSORS 11 CONDITIONS 1;
SUCCESSORS 14 CONDITIONS 0;

INDEX 12 TAG SO  OPERATION INC LINE 702;
INPUTS C(1..2)[0..0];
OUTPUTS T9(1..2)[0..0];
PREDECESSORS  11;
SUCCESSORS 13 CONDITIONS *;

INDEX 14 TAG SO  OPERATION TR  LINE 898;
INPUTS R(2..8)[0..0];
OUTPUTS T8(1..7)[0..0];
PREDECESSORS  11;
SUCCESSORS 15 CONDITIONS *;
```

Figure 3. YIF example

Fig. 3 is a fragment of YIF code. It contains 3 operations named 11, 12 and 14. The first operation shown is a conditional branch (CBR). Operation 11 follows if the variable C is less than (LST) 2, otherwise operation 14 follows. Parenthesis after a variable select bits across 2 dimensions. "LINE" indicates a token number in the source code where this node comes from. The other two nodes are tagged as "Simple Operations" which have exactly one predecessor and one successor. Their functions are INCrement and TRansfer.

The complete 801 design at this stage is 12545 lines long. It contains 1851 operation nodes and 8192 variable bits.

The next task during structural synthesis consists of merging nodes representing variables. Initially, each bit is represented by a node. Bits which are always used together are grouped into a single node. In the 801, for example, 32 bit arithmetic variables are mostly used either as one variable or the sign bit and the magnitude are used separately. Hence, two nodes in the YIF graph are enough to represent such a variable. The 8192 variable bits in the YIF

data flow graph are reduced to around 1000 nodes. This size reduction is very important for subsequent synthesis steps. The implemented algorithm merges variables with the same name inspecting all uses of them; this guarantees a resulting graph with the smallest number of variable nodes.

Next, variables requiring addressed memories are identified and tagged in the YIF graph. For the pipelined 801, memories including the general purpose register file were modeled as separate modules, so this step yields no addressable memories. The non pipelined version contains the GPR file modelled as a vector; the GPR vector is tagged as a 32-element memory and the necessary ports are provided.

The following task consists of identifying the cycles in the control graph and eliminating them by removing edges. At this point the concept of control steps or control states is introduced. For each cycle only one new control step is introduced; the removed edge that breaks the cycle is replaced by a control step that "starts" the cycle again, so that the cycle can be repeated at each clock. All cycles are thus replaced by the implicit cycle of the synchronous clock. The YIF control graph is now acyclic. In general, finding the minimal set of edges to break all cycles in a graph is NP-hard. In the YSC the operations involved in loops are tagged by the V compiler; this is easily done by identifying syntactically all possible loops. Thus, eliminating cycles involves only a few local graph transformations and is very fast.

The next step consists of replacing module calls by an appropriate waiting operation which involves just one extra control step. Again, this is just a local transformation in YIF. For a simple case, like calling a combinational module, the waiting state is not introduced and the module call is treated as any other combinational operation.

The next two steps form the core of structural synthesis. First, all variables holding values which must cross a control step boundary are marked as registers. Second, data flow constraints like writing a register twice during one control step or using two different values passed through a port in one control step are detected. This is a NP-hard problem involving the computation of all possible paths in the graph. New control steps are introduced carefully to solve the data flow constraints. The problem here is to find the "cutting" point between two control steps, so that the minimum number of control steps is necessary. This is solved by recursive backtracking, limiting the recursion depth. In practice, a recursion depth of 3 has given optimal solutions in all examples tried up through now. Introducing new registers may create new data flow constraints, and introducing new control steps may create the need for new registers. Consequently, these two steps are repeated iteratively, until no more changes are necessary.

The above transformations are defined to minimize the number of control steps for each of the possible execution sequences in the original specification, as proven in [5]. In a processor design, this means that the number of cycles per instructions is minimal.

The final step to obtain a structure is called "variable unfolding". Each variable is duplicated as many times as necessary to achieve single assignment. All references to the variable have to be changed accordingly. Each unfolded variable has only one source and thus can be implemented by a net.

By now, the YIF has been converted into a structure. A possible implementation consists of implementing each operation by combinational logic, providing one register for each variable marked as such and a net for each unfolded variable. To reduce the costs, optimizations reduce the number of registers and combinational logic. By maintaining the given control steps the problems of allocating the minimum number of registers, of required operators and of buses and/or multiplexers can be formulated as a clique covering problem [14]. This is called folding in the YSC. For the 801 example, for instance only one 33 bit adder/ALU is required. The number of registers computed is given in table 1. Communication is implemented using multiplexers.

Another optimization involves introducing additional control steps in order to reduce further hardware requirements, called cutting. For the 801, this is not desirable as explained earlier, so this step is skipped.

Finally, the structure is generated. The structure consists of a netlist connecting latches and blocks of combinational logic. For each block of combinational logic, structural synthesis generates a logic specification. Ideally, all the combinational logic of a module could be generated as one large block. Since this logic is minimized by logic synthesis there is a limit to its size. If this limit is exceeded, the combinational logic is partitioned into several smaller blocks. In the YSC, partitioning is done automatically considering not only the size of the blocks but also their connectivity and their function [3]. The structure is given in HND (Hierarchical Network Definition), the logic function in YLL (Yorktown Logic Language). Structural synthesis does not separate the data part from the control. Control is generated as one finite automaton for each module and merged with the data path.

As an exercise, we synthesized 40% of the instruction set of the non pipelined version, including the complete data path (see [2]). The main results of structural synthesis are 155 control states and 578 latches (excluding the GPR). The main disadvantages of such a design are the poor performance (not pipelined) and the large number of control states. The control states are due to the fact that in most instructions, the processor may stall waiting for data. Most control states just "remember" the particular instruction and the processor state in these cases. Notice that the large number of states is not a problem in principle, i.e. a few latches are enough to hold the state. The state transitions are simple, resulting in a small amount of combinational logic that is generated automatically.

Pipeline stage	P0	P1	P2	P3	CON	REST
V lines	33	425	730	74	283	275
V statements	20	269	599	38	190	354
Compilation time	1.0	16	249	1.9	14	12
YIF lines	119	2195	5333	354	2096	2448
YIF nodes	16	329	806	50	312	338
Signal bits	292	1083	4232	364	1074	1147
Structural synth. time	4.9	1691	9828	15.5	1260	209
Latches (bits)	98	181	180	35	99	0
Control states	4	2	1	4	1	0
Combinational inputs	68	219	255	73	202	118
Combinational outputs	69	131	152	44	167	87
Partitions	1	4	16	1	1	5
Functions	349	522	4920	297	1422	1021
Literals	597	4654	236476	689	4466	6388
Levels	37			30	40	

Table 1. Structural synthesis results for the 801 processor

Structural synthesis results for the pipelined version are given in detail in table 1. What is called "REST" are 5 procedures resulting in combinational logic. Some of them are used more than once.

The number of latches and the number of control states is indicated. The combinational logic generated as a logic specification is given by the number of inputs and outputs, logic functions, literals and levels. For pipeline stages P1 and P2 the combinational logic was partitioned automatically into smaller pieces of logic. In these cases the number of inputs and outputs correspond to the external connections (i.e. inputs and outputs without partitioning). Functions and literals are the corresponding sums over all partitions. The number of levels is not additive over partitions and is therefore not given. For REST, the 5 partitions result directly from the specification as 5 modules.

5. Logic Synthesis and Layout

Each combinational logic block is minimized separately during logic synthesis by the Yorktown Logic Editor (YLE) [2]. The YLE initially minimizes the size of the combinational logic and produces a multi-level implementation, both for SCVS (Single Cascode Voltage Switch) and CMOS. The results can be seen in table 2.

Pipeline stage	P0	P1	P2	P3	CON	REST
SCVS						
Logic synthesis time	50	915	10147	66	1608	494
Gates	85	320	1607	46	212	146
Transistors	417	2098	10414	191	1107	1049
Levels	11			4	13	
Estimated delay (ns)	23.4			7.5	28.3	
CMOS						
Logic synthesis time	62	695	11218	64	1550	804
Gates	112	379	2526	49	229	203
Transistors	361	1715	10387	179	1017	890
Levels	29			6	16	

Table 2. Logic synthesis results for the 801 processor

Timing optimization then finds the critical path with respect to the delay in the complete design and reduces delay by different measures such as transistor resizing, logic resynthesis, etc. [15]. The design is finally passed to layout synthesis to obtain the design image [2]. The complete design process is depicted in figure 4. After timing optimization, logic synthesis is usually invoked again to resynthesize for smaller delay. This loop may repeat several times. In case the timing obtained is not satisfactory, it may be necessary to go to the original V specification and change it by hand.

6. Result Evaluation

Table 3 gives a comparison between a manual design (manual in the sense that an RT level structural specification and the combinational logic were designed by human designers) and the automatic design. Both used the YLE for logic synthesis.

The number of latches does not include the general purpose register file. The combinational logic transistors include only the pull down trees, excluding the buffers and clock (SCVS) or the pull-up trees (CMOS). The total number of transistors refers to the complete design including the general purpose register file, buffers, clock, receivers and drivers. The complete design was done only in SCVS. As expected, in both the automatic and manual

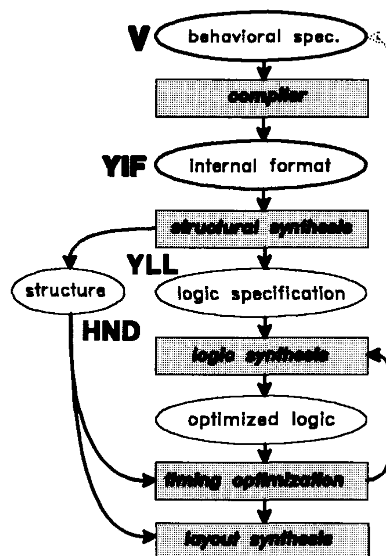


Figure 4. Design process in the YSC

Structural design	SCVS Manual	SCVS Auto	CMOS Manual	CMOS Auto
Total number of latches	534	593	534	593
Comb. logic transistors	11164	16259	9896	15321
Total number of transistors	55066	69524	-	-
Execution cycles/instruction	1	1	1	1
Unoptimized comb. delay	72.7ns	72.9ns	-	-
Structural synthesis time	-	13303s	-	13303s
Logic synthesis time	1680s	13820s	3039s	14255s

Table 3. Comparison of manual and automatic structural design

design every instruction can be executed in 1 cycle. The automatic version was not optimized for timing, but the critical path in combinational logic has an unoptimized delay of 72.9ns, which compare to 72.7ns before timing optimization in the manual design. The delay is estimated for the design after logic synthesis using a 5 parameter equation. The critical path involves computing the condition codes (in P2), generating a trap (in CON) and multiplexing the trap jump address into the program counter (in P0). Timing optimization for the manual version yields a maximum combinational delay of 49.9ns (see [2]). Timing was only computed for the SCVS design.

The above exercise focuses mainly on synthesis. Verification is done to a certain extent by simulation, but this can not be qualified as exhaustive. Design for testability is not considered explicitly, but all latches have scan path capabilities and logic synthesis should not generate redundant logic.

Manually designing the course structure of the machine organization proved to be a good compromise in automatic synthesis. The behavioral domain description of the 801 architecture could be

easily decomposed into a pipeline. The automatic synthesis of concurrency at a high level in general is beyond todays capabilities.

The execution times of several CPU hours result primarily from the size of the problem, but also from two additional reasons. First, structural synthesis and logic synthesis were coded in APL which is interpreted. Second, no special care was taken to produce fast code; the main goal is to demonstrate the feasibility of the approach and to obtain good results, especially concerning performance of the design.

The differences between the manual, RT level design and the design obtained by structural synthesis are mainly explained as follows:

- The design of the pipeline although similar is not exactly the same. For example, the difference in latches arises principally from this fact, i.e. there are only 9 latches in the automatic design that are not data path registers on the boundary between pipeline stages.
- Structural synthesis is not yet minimizing the number of interconnections. This not only leads to a large number of wires but may also hide some logic minimization potential.
- Automatic partitioning of pipeline stages P1 and P2 may lead to less optimization possibility during logic synthesis.
- In SCVS, the automatic design does not take advantage of the possibility of distributing combinational logic among two clock phases, as was done in the manual design.
- For large unoptimal logic designs, as they are generated by structural synthesis, logic synthesis may not discover all the optimization potential.

For the above example, structural synthesis yields results which are equal in performance to a manual design at the RT level. The size in number of transistors is 26% larger for the automatic design, a number that results from 45% more combinational logic and 11% more latches, using the same general purpose register file and the same amount of receivers and drivers. The design time was decreased drastically using structural synthesis, a conservative estimation is at least 5 times faster than the RT level manual design.

Present and future work include enhanced methods of partitioning, generation of microcoded control, additional optimizations and techniques for an incremental system description. Also an optional automatic separation of regular data path modules to be taken from a library or to be generated in the layout domain by module generators is being considered. With these enhancements, a reduction of the excess transistor count over the manual design to 50% of the reported values may be possible. Structural synthesis will also be tested for a complex instruction set computer architecture.

Acknowledgements

Many people contributed to the Yorktown Silicon Compiler. The work reported here could not have been possible without the assistance of R. Brayton, G. De Micheli, R. Otten and J. van Eijndhoven.

References

1. R.K. Brayton, N.L. Brenner, C.L. Chen, G. DeMicheli, C.T. McMullen and R.H.J.M. Otten. The YORKTOWN Silicon Compiler. *1985 ISCAS Proceedings*, pages 391-394, Kyoto, June 1985.

2. R.K. Brayton, R. Camposano, G. DeMicheli, R.H.J.M. Otten and J.T.J. van Eijndhoven. The Yorktown Silicon Compiler System. in D. Gajski (Editor), editor, *Silicon Compilation*, Addison-Wesley, 1988. (also IBM Research Report RC 12500, Mathematics, Yorktown Heights, December 1986).

3. R. Camposano, R.K. Brayton. Partitioning Before Logic Synthesis. *Procedings of the ICCAD'87*, Santa Clara, California, November 1987. (also International Workshop on Logic Synthesis, Research Triangle Park, North Carolina, May 1987).

4. R. Camposano. Structural Synthesis in the Yorktown Silicon Compiler. *VLSI'87*, Vancouver, August 1987.

5. R. Camposano. A Method for Structural Synthesis, IBM Research Report, RC 13081 (#56822), Computer Science, Yorktown Heights. August 1987.

6. R. Camposano, J.T.J van Eijndhoven. Combined Synthesis of Control Logic and Data Path. *Procedings of the ICCAD'87*, Santa Clara, California, November 1987. (also International Workshop on Logic Synthesis, Research Triangle Park, North Carolina, May 1987).

7. Z. Mehmood, A. Singhal, N.C. Srinivas, S.L. Taylor, K. Wu. IDEAS - An Integrated Design Automation System. *Proceedings ICCD'87*, pages 407-412, Port Chester, New York, October 1987.

8. G. Radin. The 801 minicomputer. *IBM Journal of Research and Development*, C-33(12):237-246, December 1983.

9. T. Agerwala, J. Cocke. High Performance Instruction Set Processors, IBM Research Report, RC 12434 (#55845), Computer Science, Yorktown Heights. March 1987.

10. V. Berstis, D. Brand, R. Nair. An Experiment in Silicon Compilation. *1985 ISCAS Proceedings*, pages 655-658, Kyoto, June 1985.

11. M.C. McFarland. Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions. *Proceedings of the 23rd Design Automation Conference*, pages 474-480, Las Vegas, June 1986.

12. M. Balakrishnan, A.K. Majumdar, D.K. Banerji and J.G. Linders. Allocation of Multi-Port Memories in Data Path Synthesis. *Proceedings ICCAD-87*, pages 266-269, Santa Clara, California, November 1987.

13. J. Rabaey, H. DeMan, J. Vanhoff, G. Goossens, F. Catthoor. Cathedral II : A synthesis System for Multiprocessor DSP Systems. in D. Gajski (Editor), editor, *Silicon Compilation*, pages 311-360, Addison-Wesley, 1988.

14. C.-J. Tseng, D.P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. *IEEE Transactions on Computer-Aided Design*, CAD-5(3):379-395, July 1986.

15. G. De Micheli. Performance-Oriented Synthesis of Large-Scale Domino CMOS Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):751-765, September 1987.