A Microprocessor Design Using the Yorktown Silicon Compiler

R.K. Brayton, C.L. Chen, G. De Micheli, R.H.J.M. Otten
IBM Watson Research Ctr, Yorktown Heights

J. Katzenelson, Technion-Israel Institute of Technology
C.T. McMullen, Massachussetts Institute of Technology
R.L. Rudell, University of California at Berkeley

Abstract: The process of transforming an architectural specification of a streamlined processor (the IBM 801) into a design specification suitable for input to a silicon compiler is described. Starting from that specification, the authors develop a description of combinational logic, latches and registers, hierarchically organized, which implement a pipelined version of the processing unit. This description is then automatically compiled down to the gate level in single-ended cascode technology.

1. INTRODUCTION

The work described here is part of the ongoing development of a prototype automated design tool, the Yorktown Silicon Compiler (YSC)[1]. As an exercise for the compiler and as an opportunity to learn more about system-level issues, we have developed a design for the IBM 801 processing unit [12] and described it in the languages used as input to the YSC. We chose the 801 architecture for two reasons. First, since the 801 was developed at IBM Watson Research Center, we had access to a detailed description of its operation as well as to the developers themselves. Second, the 801 is sufficiently complex to provide a good test for the compiler, and at the same time sufficiently simple to be understood by novice system designers. This paper describes the process of spanning the gap between the architectural specifications and the input to the silicon compiler.

1.1. The architecture

The architecture of a system defines its attributes as seen by the programmer, that is, the conceptual structure and sequential behavior of the machine, as distinct from the organization, the logical design, the layout, and the performance of any particular implementation. Typical parts of an architecture specification are the instruction set, the addressing capabilities, the registers visible to the assembly language programmer, and the interrupt handling. In spite of the clear separation provided by this definition, it certainly would be injudicious to isolate architecture design from its target operating system and compiler on one side, and its hardware implementation on the other. It was therefore not surprising that again and again during this exercise we were confronted with the fact that the 801 architects had the application and the implementation constantly in mind. Their decisions were mainly guided by the following three basic design principles.

1. Its compiler should be able to produce object code with an efficiency comparable to the best hand code so that assembly language programming is never needed for performance.
2. Each instruction should be executable in one short machine cycle, and yet the path lengths of the computations should not be commensurately larger than those required by machines with more complex instructions.
3. The idle time of the processing unit due to storage access should be kept small.

What follows is meant only as a brief introduction to the architecture of the 801 processor. For more details the interested reader is referred to [12].

The 801 is a true 32-bit minicomputer architecture. The arithmetic and logical operations deal with 32 bit words. Shifts and rotates can have lengths of up to 32 bits. Addresses are 32 bits long. The 801 architects foresaw a compiler capable of a very effective utilization of a large number of registers, and therefore decided to prescribe a register file of 32 fullword general purpose registers (GPR).

Instruction length and format greatly influence the complexity of the hardware for decoding, and the efficiency of a pipeline if implemented [10]. Therefore, accepting some *alignement constraints* can be very beneficial with respect to hardware simplification and pipeline implementation. This was recognized in the architecture design of the 801. Each instruction is exactly one word long, and is aligned on word boundaries. Only few instruction formats are used. All operands are, depending on their size, aligned on halfword or fullword boundaries. These constraints, the emphasis on register utilization, and the single-cycle instruction principle, move the 801 into the class of *streamlined* architectures [7]. Although many of its properties are shared with the so-called reduced instruction set computers (risc), the size of the instruction set alone would make such a qualification a misnomer in the case of the 801.

Typical instructions are provided for *storage access, address computation, branching, and comparing*. Consistent with the third basic design principle, the processor is allowed to continue with instruction execution after the initiation of a storage access. The processor will then only stall if the memory system has not yet provided the requested data when the content of the affected register is needed. To make better use of this facility, the optimizing compiler can reschedule the storage access instructions to reduce the number of stalls. Also, a branch with execute instruction (similar to the delayed branch in reduced instruction sets) allows the compiler to move inevitable instructions into the normally idle period following a taken branch instruction.

The 801 approach to protection is mainly based on its compiler assumptions. Some run time extent checking is necessary in this approach and therefore *condition test* instructions were made available for insertion into object code. *Arithmetic* is 32-bit two's complement, with the usual add and subtract instructions, beside some special instructions for computing the minimum and the maximum of two values. There are also several instructions not normally found in a reduced instruction set, such as multiply step and divide step, which allow complex operations to be easily decomposed into sequences of simple instructions. Also, a rich set of *rotate and shift* operations, controlled by another register or an immediate field are provided to reduce the path length of instruction streams.

The third basic principle immediately led to a *store-in-cache* strategy. But instead of a single conventional cache that delivers a word every cycle, an instruction cache and a data cache were both included separately, thus effectively doubling the cache bandwidth and allowing asynchronous fetching of instructions and data. Explicit instructions for *cache management* are introduced to make a reduction in unnecessary loads and stores of cache lines possible at the software level.

Finally, the number of *interrupts* defined for the processing unit is reduced by imposing software protocols for interrupt handling and prescribing an external interrupt controller.

1.2 Silicon compilation

From the specification of the architecture to the complete mask set for a particular implementation of the 801-processor on a single chip is a complicated and certainly not unique process. Many decisions about the machine organization and the implementation technology still have to be taken. At the moment it is not possible to automate the whole process, but it is feasible to have a completely automatic tool performing the last part of that process. Such a tool is often called a *silicon compiler*, because it accepts a description of the design at some intermediate stage and without interaction produces the data needed to implement the system on a silicon chip. The Yorktown Silicon Compiler project is intended to provide such a tool. It accepts input in flexible and versatile languages and then automatically performs the remainder of the design process. In addition to the gain in speed and accuracy of chip design, we hope to produce competitive results in terms of the area, performance and power requirements of the final product.

The YSC requires a description of the chip in terms of modules connected by nets. The modules themselves may be described in the same way. The nets and the undecomposed modules have attributes which drive their synthesis. The thus established hierarchy is used primarily as a guide to the organization of the chip during the layout part of the synthesis process.

A simple language has been developed to specify and generate information concerning nets, modules and their attributes. Only three kinds of undecomposed modules were used : combinational logic modules, storage devices (latches, registers, etc.) and amplifier circuits (drivers, receivers, repeaters, etc.). The specification of the storage devices and the amplifier circuits takes place entirely within this hierarchical description language. Only the combina-

tional logic modules need a specification complex enough to require additional specification. Their specifications are generated from descriptions of each logic module in a logic language called YLL [4]. Thus the input to our compiler consists of (i) a collection of YLL programs for combinational logic together with (ii) a hierarchical description of nets and modules, which indicates how the modules are linked, and which specifies the components of the chip that are not combinational logic modules.

The compiler performs transformations on a description of the chip design starting with the input language specification and ending with the layout of the masks. The transformations are performed by optimizing algorithms which successively refine the design description. This is illustrated in figure 1. These algorithms have been described in a series of papers listed as references in [1]. The subject of this paper, how to get from an architectural specification to the input to the silicon compiler in its present status, is indicated by the dotted box in figure 1.

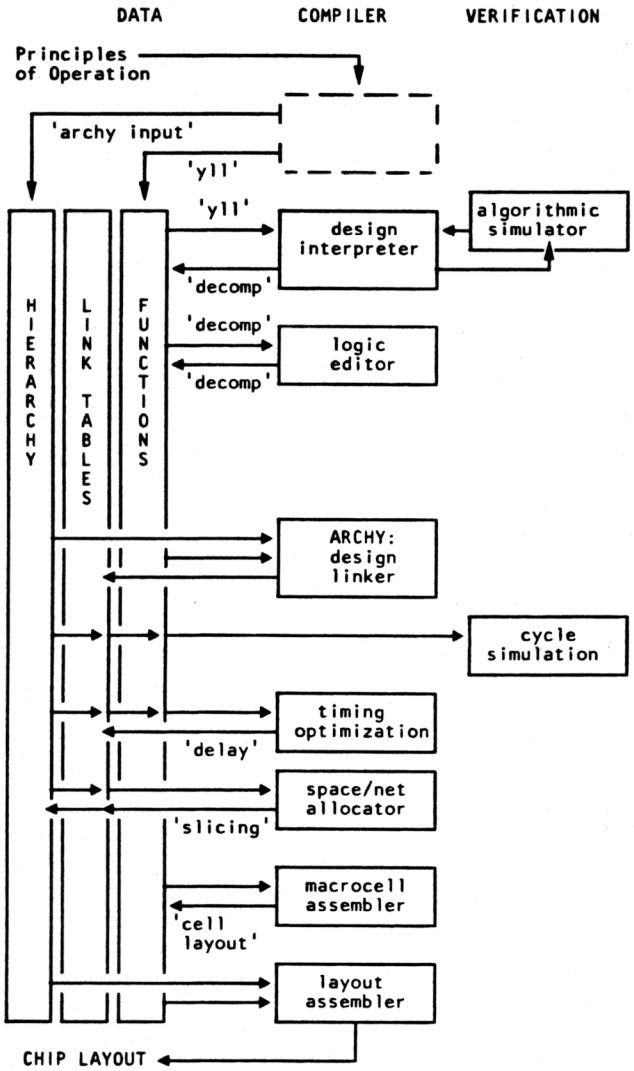


Figure 1. Schematic representation of the components of the YSC

2. SYSTEM DECOMPOSITION

A document describing in precise and verbose detail the 801 architecture, was our starting point. The first step in preparing the input for the YSC was to understand the machine being synthesized. This was achieved by converting this description into an equally precise, but far more concise computer file specifying the effect of each instruction on the architected registers, the details of when interrupts may occur and how they are processed. It became the major reference for the remainder of the project, since it was very useful for settling disputes (for example, technical details of various interrupt conditions), and to find common features (from within an editor, for example, we could easily locate all of the instructions affecting a particular special-purpose register). After that exercise we were ready to take on the organization of the control and data flow in our implementation.

2.1 High level description

In order to condense the bulk of information (as well as to provide us with an understanding of the operation of the machine), we translated the 801 principles of operation into a high level programming language (almost any programming language or hardware description language would have been suitable, but we choose the C-language, because a C compiler was available and we were most familiar with this language.) The main routine is nothing else than a infinite loop which for each cycle simulates the fetching and execution of the instruction, and tests the various interrupt condition (figure 2).

```
#include 'global.declarations'
main()
{
    for(;;) {
        fetch_instruction();
        execute_instruction();
        handle_interrupts();
    }
}
```

```
execute_instruction()
{
    switch(IR.opcode) {
            .
            .
    case LC:
        GPR[IR.rt] = read(mem,eff_addr(),char_type);
        break;
            .
            .
    }
}
```

Figure 2. General structure of the main program and the execute_instruction() procedure

Each machine register was modeled with a variable, and variables were also used to represent the external control wires (e.g., external interrupts). The access to memory was modeled with the functions read() and write(). The functions execute_instruction() and handle_interrupts() were built as case statements with the selection based on the opcode of the fetched instruction (figure 2) and the appropriate interrupt flags respectively.

Note that such a program still does not address the problem of decomposing the machine or designing a pipeline. However, while writing the program, we naturally extracted frequently occurring suboperations of each instruction and made these common features into subroutines. This is the normal decomposition one uses when faced with a large programming task. The identification of these functions provided us with an initial breakdown of the processor into smaller modules. We found, for example, that the instruction set strongly suggested two modules at the chip level: one with the arithmetic-logic operations as a core, and one with rotating and merging as its most important operations. Once they were identified, most instructions could be translated into *subroutine calls* to procedures specifying the functional behavior of these modules.

2.2 The pipeline

Pipeline design is useful to increase performance through reduced cycle time, and since some of the 801 instructions seem naturally tailored to a pipelined implementation, we chose to implement what seemed to be a natural 4-stage pipelined structure. That this choice was not canonical, can be seen by comparing it to the quite different design discussed in [12].

The pipeline can be made such that the four stages operate synchronously, and each completes its operation in one machine cycle. At the boundary of each stage are latches holding its next input and storing its last result. The stages are as follows:

F(fetch)	Load into IR the instruction pointed to by the program counter.
D(decode)	Load the appropriate registers (RA,RB,RS) with the contents of the general purpose registers pointed to by the instruction, and create a decoded version of the instruction's operation code.
E(execute)	Generate control signals from the operation code, make the arguments available to the execution unit, and latch the results in RESULT. Also update the program counter.
S(store)	Place the result in a register, and initiate requests to main memory.

Under simplest conditions, one instruction resides in each stage of the pipe and flows to the next during one cycle. Added complexity results when we consider the behavior required while branching, handling an interrupt, or waiting for data. To the user, the processor's properties must appear to be the same regardless of whether or not a pipeline is used. Thus the rather complex pipeline control we will describe in section 2.4. can in principle be deduced from an accurate model of the processing unit and the pipeline assumptions.

2.3 Module definition

In the course of the discussion so far we noted several implications and suggestions for modules. Some of these were explicitly mentioned in the architecture specification. Their exact instantiation, however, is still to be determined, mostly on the basis of implementation cost, clocking schemes, and testability considerations. We decided to implement the processing unit as a synchronous *finite-state machine* consisting of combinational logic cells and synchronously clocked registers. All registers, except those in the GPR, consist of master-slave latches, with or without an enable line and controlled by a single two-phase non-overlapping clock, called M/S clock. A machine cycle corresponds with the period between two master clock pulses. For reasons of testability by level sensitive scan design (LSSD) techniques, all registers are provided with a scan-in and scan-out line, and are connected to form a scan chain. The GPR is implemented as a static random access memory, capable of reading out any three and writing any two registers within a single cycle.

A partition of the processing unit into a data-flow and a control module often seems obvious. If that partition is accepted and given to the silicon compiler the two modules will be realized in two nonoverlapping rectangles, and treated almost independently from each other. Closer examination shows that such a decomposition is not necessarily the optimum. The data-flow facilities are controlled by the instructions at the corresponding stage of the pipeline. Since every stage of the pipe operates in a single cycle, the control of each facility is a combinational function. This organization suggests the idea of *local control* of the data-flow: the control of each facility is a separate unit, placed next to the facility itself (or might even be merged into it). Local control is convenient because it reduces the wiring. Moreover, it can be efficiently designed by exploiting the unused or inappropriate op-codes as *don't care* conditions for the local control blocks.

In the previous section we already noted the suggested division of the execution unit into two paths: one containing an arithmetic-logic unit, and one with rotator and merge logic. In the high level description this observation could be translated into *subroutine calls* to procedures specifying the functional behavior of these two modules. In the realization each call becomes part of the control logic, operating multiplexers at the input to the modules and specifying the operation the selected module is to perform.

The decisions with respect to module definition led to the diagram of figure 3.

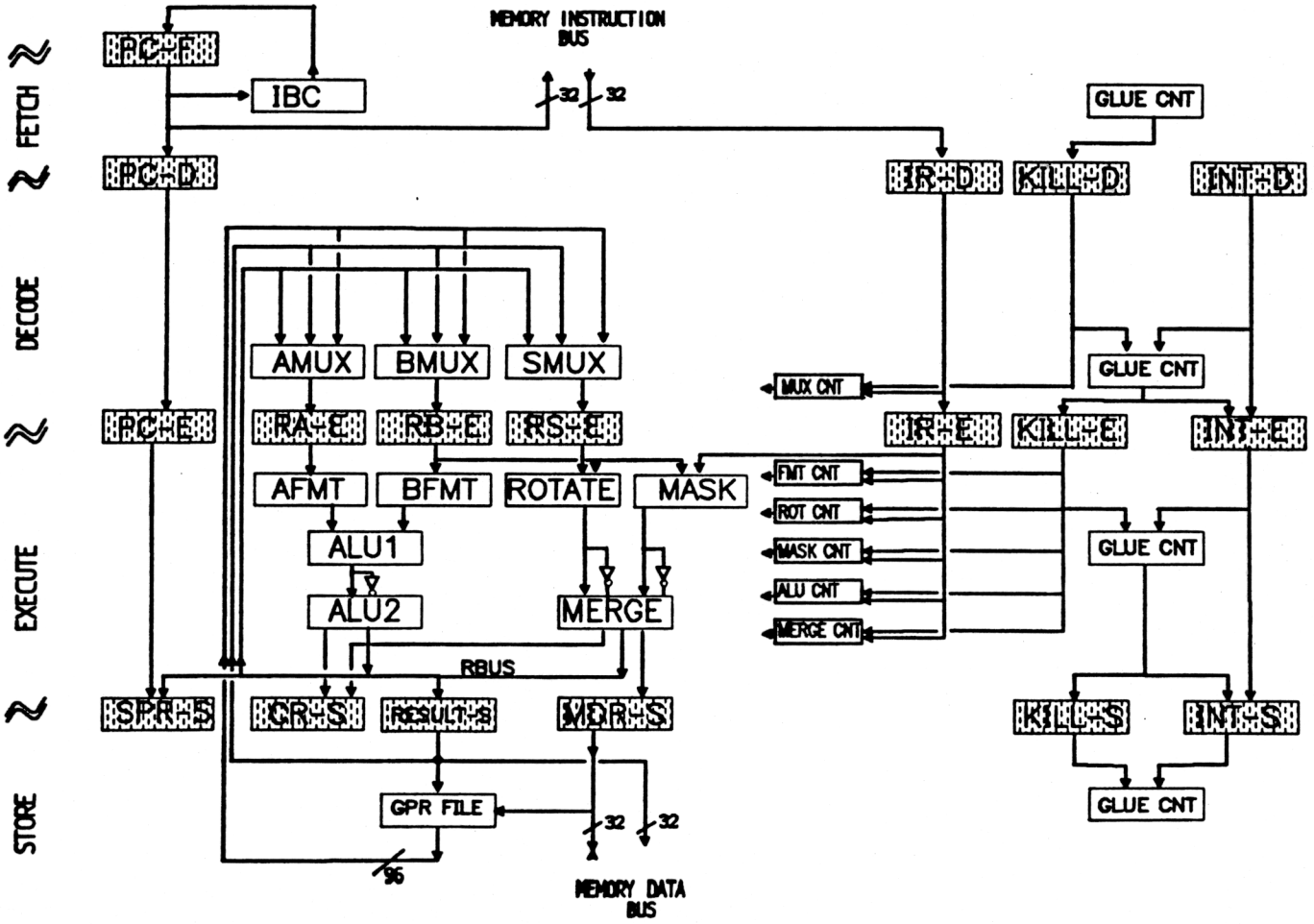


Figure 3. Schematic of our organization of the 801 processor. Shaded boxes represent latches. Small inverters indicate inversion between D1- and D2-domain.

2.4 Pipeline control

The operation of the pipeline under a branch instruction is fairly straightforward. If the branch is taken, the program counter is loaded with the target address at the end of the E-stage. The partially executed instructions in the other stages of the pipeline are now incorrect; in an unpipelined design, they would never have been fetched. To handle this situation, each stage of the pipeline has a 'kill' flag, which is propagated from one stage to the next as the instruction flows down the pipe. A branch taken sets the kill flag for all the other instructions currently in the pipe. When such an instruction reaches the S-stage, its affect on architected storage is inhibited because the kill flag is set. In this way, a branch instruction effectively flushes the contents of the pipe.

Interrupts are very similar to branches. In fact the action of the processing unit under an interrupt is to store the current status in special purpose registers and then to branch to a fixed memory location. One additional consideration is required, because interrupts can be generated at any stage of the pipe. To handle this, associated to each stage are interrupt bits, which flow along with the instruction like the kill flag described above. As interrupts are generated, these bits are set, and when the instruction reaches the S-stage, the interrupt is serviced.

We remark that an instruction can generate an interrupt in, say, the D-stage, and then be killed by a branch instruction ahead of it. In this case, the instruction in the D-stage should never have been executed and the interrupt it generated is ignored.

Another consideration is the 'waiting for data' state of the pipeline. Consider the example of loading a register from external memory. The load instruction generates a memory request, which is sent off chip. Execution continues normally until a reference is made to the register which was supposed to be loaded. If the memory request has not yet been serviced, we must wait for it before proceeding. The waiting function is accomplished by inhibiting the transfer of data from one stage of the pipeline to the next. In this way we effectively halt the pipeline until the external request is serviced.

The flow through the F-D-E-S-stages of the pipeline is altered somewhat by certain bypasses in the pipeline. These speed the flow of the instructions through the pipe. For instance, if an instruction in the D-stage requires the contents of a register which is the target of a load instruction or the target of the result just computed in the E-stage, the data will be routed into the appropriate latch on the boundary between the D- and E-stages. On the next cycle this data is stored in the correct register, but it is also already in the E-stage. This allows, for example, a sequence of additions to be executed without interruption. Another complication is that there are instructions that refer to writing data directly into some of the architected latches. This could be done at the end of the S-stage, but this might cause some delay in the instruction flow. If we know that no interrupt will be processed on the next cycle, then it is allowed to perform this store at the end of the E-stage. This generated the need for a signal *interrupt at next cycle* computed in the E-stage. In general these decisions corrupting the clean design of the pipeline, complicate the control logic and may affect the cycle time of the machine.

3. TECHNOLOGY CONSIDERATIONS

Up to this point our design has been technology independent. Before passing the design to the silicon compiler, decisions concerning the technology and the circuit family have to be taken. Most of the limitations of the chosen technology will be taken care of by the compiler, but some decisions that can improve the final result considerably have to be forced manually.

3.1 Single-ended cascode voltage switch circuits

The main benefit of a silicon compiler are fast error-free design without communication problems between experts of quite different fields. These should not be obtained at the price of considerable increases in area, power and delay, compared to manual designs. Achieving competitiveness is facilitated by the use of a fast, area-efficient, powerful logic family in which a complex function can be performed in a single circuit. Methods for manipulating complex logic functions are better suited for computers than for humans. So, if there is an advantage in using large logic families, it can only be realized through high degrees of automation.

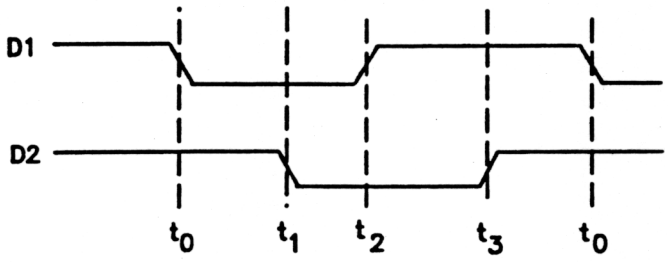
Single-ended cascode voltage switch (scvs), realized in a cmos technology, is such a family. Its circuits operate according to the *domino principle* [9,8,5,6], i.e. a capacitance is charged during a so-called precharge phase, and discharged during the evaluation phase if its switching network has a conducting path to ground under the current values of the logic inputs. The phases are derived from a single clock, thus avoiding delays due to clock skew. The switching network can be realized by the relatively small nmos transistors, and only part of the small (compared to conventional cmos) overhead has to be realized in the much larger pmos transistors. Previous experiments on individual control logic macros and arithmetic-logic units indicated that significant savings in area and delay could be obtained at the macro level by using this technology [2,5], in spite of the requirement that signals cannot be inverted to feed other circuits in the same evaluation phase.

3.2 Clock domains

SCVS seems to be an excellent candidate for our purposes, provided that some rules are taken into account if this technology is going to be used for every combinational logic cell. We therefore extended the basic domino-circuit, and provided the system with two distinct clock signals, D1 and D2 respectively (Figure 4). The circuits between two latch boundaries are partitioned into two domains by applying these two clock signals, a D1-domain and and a D2-domain. The rules for interconnecting these circuits are as follows:

1. Only circuits in the D1-domain can have an inverted output.
2. An inverted output can only be input to a circuit in the D2-domain.
3. Only one inverter is allowed in any chain from input latch to output latch.

The assignment of each of the logic macros to the two domains was done manually. The crucial part of the assignment was that for the two main functional data paths, the ALU path and the rotate-merge path.



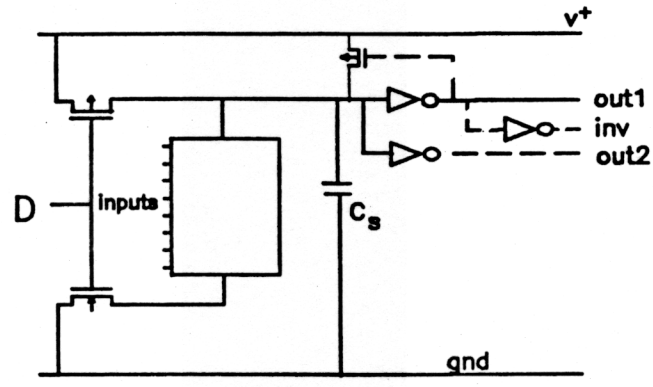


Figure 4. The extended basic domino circuit and the timing diagrams of the two clock signals. The dashed parts in the circuit are included when necessary. Logic controlled by D1 begin executing at t_2, while the logic controlled by D2 begins at t_3. The time t_3 is chosen so that all inverters have their valid state by then. Thus when the D2 goes high at t_3, no invalid discharging of sense nodes in logic macros is possible. The time at which the precharging of D2 circuits is begun, t_1, is chosen different from t_0, to reduce the peak current drawn from the power sources at the beginning of precharge.

The addition computation in the ALU was broken down into ALU1 and ALU2, and these were assigned to the D1-domain and the D2-domain respectively (figure 3). ALU1 consists of the *carry generate* part of the adder, and ALU2 consists of the exclusive-or operations for obtaining the *sum* bits. The *carry generate* functions are positive functions of their inputs, such as

$$C_{OUT} = AB + (A + B)C_{IN}.$$

Thus no inverts are required inside ALU1. The *sum* output of ALU2 corresponds roughly with

$$S = A \oplus B \oplus C$$

and therefore requires both phases of all signals so that the inverts of A,B and C must be made available. This is accomplished by inverting signals produced in ALU1. Without the use of inverters, additional positive logic would be required to compute the inverse of A, B and C (roughly doubling the size of ALU1). This would be a severe penalty in terms of area and power.

The rotate-merge block also requires inversions in its final phase. The function of this block is to rotate one word and merge this result with another word where the merge is controlled by a 32-bit mask. The mask acts like a set of selector controls and as such both phases of the mask must be available. In addition some outputs of the merge result must be available in the negative phase to set a condition code. To circumvent having to propagate these complements back to a latch boundary, the mask generator macro and the rotate macro are in the D1-domain, while the merge macro is in the D2-domain (figure 3).

4. COMPILER INPUT

Once the numerous decisions concerning pipelining, choice of modules, control signals, interrupt mechanism and so on were made, we began the task of describing this proposed implementation of the 801 architecture in the languages required by the Yorktown Silicon Compiler. In the sequel we will assume some familiarity with that system, particularly its logic language.

4.1 The logic language

We begin by discussing the descriptions in the logic language, YLL. Once the control signals which link the D- and E-stages of the pipeline had been established, writing YLL programs for each logic macro in the 801 turned out to be a fairly straightforward and modular programming task. Since the use of YLL for describing ALUs, rotators and other structured macros has been described elsewhere [4], we will limit ourselves to two remarks on our experience in writing control logic.

First, the description of control logic is organized and simplified by the use of tables. These allow us to refer to instruction codes directly by mnemonic. Furthermore, the tables list for each instruction numerous properties of the operation it is intended to perform, allowing us to write logical equations that directly query specific properties of the instruction at hand.

Example. Shown below is a portion of ROTIN, the table of rotate and merge instructions. For each instruction, we have symbolically described the operation it performs. In this case, the entries in the table answer questions such as: where does the rotation count come from? Is the rotation to the right or to the left?

```
⍝-------------------------------------------
⍝    Opcode      Count      Merge Type    R/L
⍝-------------------------------------------
RIMI   29        C_RB       MASKIN        LEFT
RMI    22        C_RB       MASKIN        LEFT
RINM   11        C_IR       SHORT         LEFT
RNM    10        C_IR       SHORT         LEFT
RRIB   63        C_RA       BITIN         RIGH
```

The YLL code below describes a portion of the control logic for the rotate/merge block. INSTR is an 8-bit logic signal that contains the current instruction code in the E-stage. These three lines examine INSTR to determine if the count for the rotation comes from the register IR, RB or RA. The first line of code constructs a control signal C_IR which is turned on exactly when the current instruction corresponds to any line in the ROTIN table bearing the entry 'C_IR', such as RINM or RNM above. The logic for this control signal is simply an expression which tests to see if the 8-bit data INSTR (suitably encoded) matches the opcode of RINM or RNM, as given in the table. In this way, a correspondence between lists of opcodes and conceptual properties of instructions is established.

```
⍝-----------------------------------
⍝   Count-Selector Control Signals.
⍝-----------------------------------
C_IR←∨/(ROTIN IS 'C_IR')⊥INSTR
C_RB←∨/(ROTIN IS 'C_RB')⊥INSTR
C_RA←∨/(ROTIN IS 'C_RA')⊥INSTR
```

We remark that these instruction tables are not part of the architecture, but were *inferred* from it. In fact, the selection of control signals and the construction of versatile macros implementing a given instruction set via these controls is a fundamental step in the design process. The automation of this step represents an important area for future research.

A second novel feature of our YLL description is that we have explicitly described conditions under which we *don't care* about the values of certain signals, and we have used this information to simplify the resulting logic.

Example. The statements shown below follow the control logic above. When the rotate/merge block is in use, the count must come from either IR, RB or RA. So when the count-selector control signals are all zero (false), the current instruction is not using the output of the merger. Under such a condition, we actually don't care what the values of these control signals are (since they don't affect the final result). To take advantage of this additional freedom, we define below an intermediate logical expression DC specifying exactly these don't-care conditions.

```
DC←~∨/(ROTIN IS 'C_IR C_RB C_RA')⊥INSTR
DC SIMPDC C_IR,C_RB,C_RA
```

The first statement creates a logic function which is the negation of the *care* conditions, in this case those instruction opcodes that are associated with *C_IR*, *C_RB*, and *C_RA*. The second statement tells YLL to simplify the logic for the three count-selector signals, using the fact that their values can be arbitrary under the condition DC.

This facility allows us to automatically take advantage of an intelligent choice of operation codes, such as that specified for the 801 architecture. Suppose, for example, the first two instruction bits are always '0' for a rotate/merge instruction. Then there is never any need to interrogate these bits when computing rotate/merge control. We can simply assume they are both zero, since when they are not, we don't care what the control signals are. This effect is achieved by using the don't care set as above. (Of course there is some control signal which records the fact that we are using the output of the merger, and which *does* care about the first two bits of the instruction code.)

4.2 The hierarchy description

We now turn to the description of nets and modules. Once the YLL descriptions of each logic macro are available, they are processed by a logic synthesis program [3] to produce *decomp* files, which describe multistage circuitry implementing the original logic in the target technology (SCVS). The names of the primary outputs of each logic macro are retained from the original YLL description. One can think of the *decomp* files as *compiled subroutines* or *object modules*, which must be linked together to form a program. The data is produced by the *design linker*, called ARCHY.

The ARCHY processor takes as input a simple language describing the contents of a collection of modules. The lowest-level modules are physically realizable structures, such as registers or logic macros. Associated to each such structure is an *ordered* list of nets, indicating the signal wires to which it is attached. This list is simply a collection of alphanumeric names which have global significance. The list is either specified explicitly in the input to ARCHY, or derived implicitly, as for *decomp* files, where the net names have been preserved from the original YLL code. Most modules are connected to a clock signal, which is specified in the input to ARCHY, and controls the activation of the module. Thus the organization of sequential and parallel execution of devices is implied in this language.

The ARCHY processor builds up a netlist, referring to the existing *decomp* files as necessary. It also builds a description of each module. This description includes the *type* of the module, which is DECOMP for combinational logic, LATCH for master-slave latches and registers, and so on. As mentioned in section 1.2, many objects are completely described in the ARCHY input.

Example. To describe a master-slave register with an additional enable signal (which can inhibit the data transfer), we use the ARCHY statement

```
LATCHE RA_IN[0-31] RA[0-31] RA_EN (MSCLK
```

This specifies that the 32 signals RA_IN are to be latched to the signals RA at the clock pulse MSCLK. The signal RA_EN can enable or disable this transfer. No additional information is needed for the layout program to construct this register.

Part of the ARCHY input describing the rotate/merge section of our 801 implementation is shown below.

```
MODULE ROTNMRG
  DECOMP SHIFTC              (DOMINO-I
  DECOMP ROTATE              (DOMINO-I
  DECOMP MASKMX              (DOMINO-I
  DECOMP MERGE               (DOMINO-II
A    Make sure inverts are available
  INVERT MASKMX MERGE        (DOMINO-I
```

In the above we are describing part of the module ROTNMRG. It contains several logic macros; the declaration that these macros all belong to one module, will group them as a unit for the layout process. We have specified that the first three macros run on the D1-clock signal, while the merge takes place at second stage, controlled by the D2-clock signal. Before D2 goes to the high value, signals from the D1-domain can be inverted as needed. The statement INVERT MASKMX MERGE examines the corresponding *decomp* files to see if a signal required by MERGE in one polarity is produced by MASKMX only in the opposite polarity. If so, an inverter will be generated. Thus we have used ARCHY not only to describe the linkage of the logic macros, but to generate linkage as necessary. It also has a facility for auditing the net generation, to check that connections are made, that no module requires a signal which has not been made available, and that the rules for creating inverters are not violated.

5. DESIGN STATISTICS

In this section we summarize very briefly some results obtained by exercising the YSC with our 801 processor design. A more extensive report on results of the YSC for this design will be published in the future.

5.1 Input code

The high level description consisted of 1602 lines of C-code (with 722 semicolons indicating the amount of active code). These were translated into

- 1877 lines of YLL code (33 pages); 1106 were comments.
- 547 lines of tables (9 pages); 126 were comments.
- 199 lines of ARCHY code (4 pages); 68 were comments.

In total there are 1323 lines of active code (24 pages) for input to YSC. Since YSC has no libraries except for a register file and bit images for single latches, drivers and receivers, the data above contains the entire chip description.

5.2 Logic synthesis

Our decomposition led to a total of 118 different undecomposed modules. The combinational logic modules have been synthesized down to the gate level. Also a global netlist was created for the chip. These data have been used for gate level simulation and timing optimization at the chip level.

	total blocks	total gates	min/max gates inblck	min/max xstrs inblck	total xstrs
Combinational logic blks	58	1415	1/110	10/1344	17660
Inverter blks	8	150	1/97	3/291	450
Sub totals	64	1565			18110
Regular latch blks (LSSD)	5	38	1/32	24/768	912
Latch/Enable blks (LSSD)	31	514	1/32	32/1024	16448
Register file	1	1184	1184	19000	19000
Sub totals	37	1736			36360
Driver blks	4	69	1/32	4/128	276
Receiver blks	10	41	1/32	0/0	0
Tri-state driver/receiver	1	32	1	320	320
Sub totals	15	142			596
Totals	118	3443			55066

Some statistics of the synthesis results have been entered in the table. The totals in this table do not include the internal powering of signals, for example by clock repeaters, but do include all the off-chip drivers and receivers. In compiling the statistics, all output buffers are assumed to be of minimum size, and none of the domino circuits was assumed to have any of the options given in figure 4, except possibly for the inverters between the clock domains. So the transistor totals for the domino circuits include only the transistors required for the gate logic and 5 transistors for the buffer and clock. The decomposition of the logic was done with the possibility of distributing the control to or near to the modules being controlled. Thus of the 58 combinational logic modules, 25 generate control signals.

5.3 Layout design

The system requires 142 bonding pads for data communication with its environment. Together with the supply pads and the clock, they require a periphery of 31.7 mm. The bonding pad macros take an area of 23.8 mm^2, leaving about 39 mm^2 as active area. The area taken by cells was computed to be 19.4 mm^2 which leaves fifty percent of the active area for the global wiring. In the global netlist, there are 2500 nets, but of these only 500 are not 32 bit vectors which are part of the dataflow. The floorplan is to contain a *data flow stack* where most of these 32 bit signal nets are routed in straight lines in second level metal over the stack, thus reducing the global wiring space considerably.

The design rule set used by the macro assembler was competitive, but not a production standard. Once the macro assembler has been adapted to a current production standard, and all its components are in place the design will be compiled by the YSC using that design rule set.

6. CONCLUSIONS

With the design methodology supported by the current version of the YSC, the translation from the high level description to the block diagram is left up to the human designer. Notable attempts have been made in the past to automate this step (once given a description of the machine similar to our high level description). For example, the MIMOLA program [14] and the CMU's Datapath-Analyzer [11,12] attack the problems of translating a high-level description of a machine into basic modules. Input to the YSC at a higher level is under consideration.

For the 801 design, the translation from the high level description to the diagram was not difficult. Only a relatively minor part of the design effort would have been saved with an automated system. However, there are several advantages to a more automated system. One is that no new errors are introduced during the translation process. Another is that more experimentation with the processor structure is possible. With a silicon compiler such as the YSC, quite accurate information about critical timing and layout can be produced quickly. In this design exercise it would have been interesting to try different pipeline variations, to determine their effect on the cycle time. Experimentation with different facilities, dataflow paths, tri-state buses etc., would have revealed their effect on the overall layout. A ripe area for immediate development seems to lie in design aids operating in the range between full automation at the high level and the level represented by the input to an optimizing silicon compiler such as the YSC.

The full automation of this translation process into an efficient structure appears very difficult. For example, in this design, the intentions of the machine architect in choosing the instructions to facilitate a pipeline were rather obvious. Once the pattern was discovered, designing the pipeline was relatively easy, but automating this is clearly a much more difficult task.

Currently, the power of the YSC is that once the individual logic macros have been identified and described functionally in the Yorktown Logic Language (YLL), the logic editor and layout tools produce a highly-optimized macrocell which implements the function. Thus, the YSC provides a very powerful translation from an intermediate-level description into a mask-layout.

The introduction of two distinct clock signals essentially allows us to obtain the advantages of domino-type circuits at the system level. This notion can be extended, but more experience is required to see how generally applicable this technique would be. Although system design is made slightly more complex by the invert-free constraint, this exercise has demonstrated that the benefits of single-ended cascode voltage switch circuits can be realized for a

reasonably large chip. Further design aids should be able to simplify the system aspects of the specifications.

Finally, our impression was that the 801 architecture is relatively easy to implement. The implications in the instruction set with respect to short and simple machine cycles and pipelining were soon recognized, and our decisions concerning system decomposition and clocking scheme did not adversely affect the effective cycle time of our implementation of the 801 processor. We believe that our design confirmed that the goal set by the the architects to allow an implementation having a short cycle time was successfully met.

Acknowledgement

Several people at the IBM Watson Research Center have been helpful during our first steps in the area of system level implementations. Among them, John Cocke, Martin Hopkins, Eric Kronstadt, and Yannis Yamour, certainly deserve mentioning. We are also grateful to Carla Otten for helping us produce the figures for this paper.

References

[1] Brayton, R.K., Brenner, N.L., Chen, C.L., DeMicheli, G., McMullen, C.T., and Otten, R.H.J.M., *The Yorktown Silicon Compiler* , ISCAS'85, Kyoto, Japan, June 1985.

[2] Brayton,R.K., Chen, C.L. , McMullen, C.T., Otten, R. and Yamour, Y. *Automated implementation of switching functions as dynamic CMOS circuits*, CICC 84 Rochester, May 1984, pp.346-350.

[3] Brayton, R.K., McMullen, C.T., *Synthesis and Optimization of Multistage Logic* , ICCD84, Port Chester, NY, Oct. 1984, pp.23-28.

[4] Brenner, N., *The Yorktown Logic Language: an APL-like Design Language for VLSI Specification* , ICCD84, Port Chester, NY, Oct. 1984, pp.11-15.

[5] Chen, C.L. and Otten, R. *Considerations for implementing cmos processors*, ICCD84, Port Chester, NY, Oct. 1984, pp.48-53.

[6] Heller, L.G., Griffin, W.R., Davis, J.W., Thoma, N.G., *Cascode Voltage Switch Logic - A Differential Logic Family*, ISSCC 84, San Francisco, Feb. 1984, 16-17.

[7] Hennessy, J.L., *VLSI processor architecture*, IEEE Trans. on Comp., Vol C-33, nr. 12, December 1984, pp.1221-1246.

[8] Krambeck, R.H. Lee, C.M. Law, H.S., *High-speed compact circuits with cmos*, IEEE Journal of Solid-state Circuits, Volume SC-17, nr 3, June 1982, pp. 614-619.

[9] Luisi, J.A. *High-speed, low-cost, clock-controlled cmos logic implementation*, US Patent 3982138, Sept 21, 1976

[10] Matick,R.E., Ling, D.T. *Architecture implications in the design of microprocessors*, IBM Systems Journal, Vol 23, nr 3, 1984, pp.264-280.

[11] Parker, A.C., Thomas, D.E., Sieworek D. , Barbacci, M.R. , Leive, G. and Kim J. *The CMU Design Automation System: An Example of Automated Data Path Design*, 16th Design Automation Conference, June 1979, 73-80.

[12] Radin, G, *The 801 minicomputer*, IBM Journal of Res.and Dev., Vol 27, nr 3, May 1983, pp.237-246.

[13] Thomas, D.E., Hitchcock III, C.T. Kowalski, T.J., Rajan, J.V. and Walker R.A., *Automatic Data Path Synthesis*, IEEE Computer, Dec. 1983, 59-70.

[14] Zimmermann, G., *The MIMOLA Design System: Computer-Aided Digital Processor Design Method*, 16th Design Automation Conference, June 1979, 53-58.