Experience with CAD Tools for the AT&T CRISP Microprocessor

David R. Ditzel

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Alan D. Berenbaum

AT&T Information Systems
Holmdel, New Jersey 07733

ABSTRACT

The design methodology used in the design of the CRISP Microprocessor is described. CRISP is a high performance 32-bit microprocessor using 172,163 transistors in a 1.75μ CMOS technology. A top down approach was taken in the design of the machine, using tools ranging from instruction set interpreters to detailed device modelling. In addition to a discussion of the tools, this paper describes the experiences, war stories, and lessons learned in trying to apply these tools to a large microprocessor design.

Introduction

The design and verification of complex microprocessors poses an ever increasing challenge to both designers and their computer aided design (CAD) tools. Without new approaches, the amount of time required to complete a successful design could make it obsolete before its completion. The design of the CRISP Microprocessor represented a challenge in three areas. First, the instruction set of the machine was new, and the hardware architecture used several techniques that had not been tried before. Extensive use of pipelining significantly increased the difficulty of simply getting the logical design of the machine correct. Second, the implementation team consisted of a combination of individuals from research and from development organizations, and from different geographic locations. We were determined to keep the number of designers as small as possible, and to have each designer understand enough of the entire machine to be able to make informed tradeoff decisions. The design and implementation of the chip was performed by a technical staff of about six people over a three year period. Third, we desired to push the state of the design tools, by combining several new tools with well established tools. This combination did improve productivity, but had other costs.

This paper reports on the tools and experiences with the approach taken to design the CRISP Microprocessor. The major design tools used are described along with pleasant and unpleasant surprises in their use. Problems with traditional approaches due to the increased size of the design are discussed. What seem to be minor issues often have a large effect the ability of CAD tools to work together properly. Comments are offered on several of these issues in the hopes that others may avoid their pitfalls.

As the level of detail of simulation increases, substantially more cpu time is required and the total number of instructions one can realistically simulate grows smaller. When a bug occurs, it is often much harder to understand in the more detailed model of the machine than in a less detailed model. For this reason, we started by simulating as much as possible at the highest level. Many bugs will not be present at the most abstract level of simulation, so simulation must eventually be done at a very detailed level.

The CRISP Microprocessor

The CRISP Microprocessor is a high-performance general-purpose single-chip 32-bit microprocessor. Performance at 16 MHz is capable of exceeding 16 mips (millions of instructions per second). The CRISP instruction set is carefully streamlined to allow an efficient pipelined implementation.[1] CRISP has a simple memory-to-memory style instruction set with about 25 basic instruction types. There are four basic addressing modes what can operate on bytes, halfwords and 32-bit words. This simple instruction set allows simple and fast compilers to be written for the C programming language.

CRISP is implemented as two logically separate machines on the same chip. The Prefetch and Decode Unit (PDU) fetches instructions into a 512 byte cache, decodes them, and places 192-bit wide decoded instructions into a decoded instruction cache. The Execution Unit (EU) reads these decoded instructions and is capable of issuing a new decoded instruction to its 3-stage pipeline every clock cycle. Using a technique called Branch Folding[2] a decoded instruction can contain up to two CRISP instructions, allowing the apparent mip rate to break the 1 instruction/cycle barrier. Despite the appearance as a memory-to-memory architecture, CRISP keeps more data references on-chip than general register machines. Registers are automatically allocated by the hardware using a Stack Cache.[3]

CRISP is implemented in a 1.75 micron single-metal dual-poly CMOS technology.[4] The die size is 10.35mm x 12.23mm and contains 172,163 actual transistors, of which approximately 100,000 transistors are memory. The chip has 95 active signal pads, 20 power, 19 ground and 6 test pads. Control of the PDU and EU are each accomplished with a combination of a single PLA and hardwired control. A chip photograph is shown in Figure 1.

The Interpreter

Our first goal was to implement the exact specification and semantics of the instruction set. An instruction interpreter was written to execute instructions with the same semantic effect as the final hardware was to have. Using a C compiler with the interpreter allowed us to execute real programs. Benefits from having an interpreter early in the design are (1) being forced to detail the exact specification of the instruction set and (2) debugging the compiler early. Debugging the compiler early is essential so that detailed logic simulation does not appear to have hardware bugs that turn out to be compiler errors. The interpreter is also the fastest of the instruction simulation models and hence usually the easiest to debug the compiler with. Our interpreter included its own debugging system which allowed one to set breakpoints, dump memory, run for a specified number of cycles, dis-assemble instructions and other features.

Architectural Simulator

The first simulation of the hardware architecture of the processor was done with a C program called the architectural simulator. The architectural simulator modeled the hardware at the register transfer level. This simulator could run large programs, such as the C compiler, and provide an estimate of performance in terms of the number of clock cycles required. The pipeline structure and cache memory sizes could be easily changed at this stage, this allowed us to use the simulator to gather statistics on the performance sensitivity of various features. New features were only added if they could provide a sufficient performance boost to justify increased implementation complexity.

While the architectural simulator modeled all the major machine registers and datapath it was not partitioned to reflect the physical implementation. The control for the datapath elements was not explicitly modeled. Numerous details, such as the exact handling of faults and exceptions were not dealt with in the architectural simulator. The speed of the architectural simulator is about an order of magnitude slower than the instruction set interpreter.

Functional Simulator

As the final architecture of the machine solidified the architectural simulator was rewritten into what became the Functional Simulator (Fsim). The Fsim is partitioned into modules which correspond to the physical partitions of the final silicon, and all datapath and control signals are explicitly modeled. The functional simulator represents the complete design, and so is used to verify the sanity of the design, and also to provide test vectors for all later stages of debugging. CRISP uses a four phase clock and the Fsim models the hardware accurately down to the individual clock phase boundary. Each of the major pieces of layout has a corresponding C header file that details every wire entering or leaving the block and a C code file to model logic within that block. Other files contain display routines, an elaborate debugger and other support for the simulator.

We found the C programming language[5] both sufficient for describing the detailed hardware of the machine and efficient in execution speed. Figure 2a shows some edited excerpts from a header file and figure 2b shows excerpts from the corresponding C file. Each signal entering or leaving the block is described as an *input*, *output* or bidirectional *ioput*. Major internal signals are also identified. For the purposes of the functional simulator, the C preprocessor transforms these new keywords into the standard integer datatype, making this just another ordinary C program. Each signal declaration may optionally be followed by field specifying the number of bits (assumed to start at bit 0), or an explicit naming of the starting and ending bits of a bus. Signals without this field are assumed to be single bit values. The bit range specifiers are ignored by the functional simulator, but are used in backporting, which is described below.

Figure 2b shows more detail of how the functional simulator simulates the hardware. In this case the logic block is called *pdureg*, this subroutine is called once for each of the four clock phases in simulating a single clock cycle. Figure 2b shows a master/slave pipeline register clocked from the master *pdpc* into the slave *Pdpc* during clock phase one if the enable signal *ckpdpc* is active. Adders, subtracters and other bit manipulation are easily handled with standard operators in the C language. A multiplexer can be handled with a switch statement, as well as being able to check for illegal values of the mux control. In short, most hardware constructs could be easily modeled simply by writing in a stylized form. Translation of functional simulator code into schematics was usually quite straightforward.

C was chosen as the simulator language to ensure maximum simulation speed — the more code we could run through simulation, the sooner and easier it would be to uncover bugs. On an IBM 3081, the Fsim runs approximately 500 cycles per second, approximately 10 times slower than the architectural simulator and about 100 times slower than the interpreter. We routinely ran simulations of up to 1,000,000 cycles at a time. In opting for speed, we lost some advantages of specially-designed simulation languages. In particular, modelling parallel behavior in the sequential C model led to calling order bugs that had to be resolved. An understanding of the hardware was required by the Fsim writers to only write code which resembled reasonable circuits.

One of the practical advantages of writing the functional simulator in C was that it was comfortable since all the tools and techniques for standard programming were available. The writers of the functional simulator went beyond mere simulation, and developed an extensive set of commands and debugging facilities to facilitate design and make debugging easier. One of the nicest features was an interactive display of the major machine registers. Each of the pipeline stages was shown on a CRT screen, allowing one to watch multiple instructions flowing through the machine. The practical effect was that newcomers understood the machine more easily than by looking at the simulator code, and those intimately familiar with the machine were able to understand the cause of bugs much sooner.

```
#define input    extern int  /* redefine as standard C datatype */
#define output   extern int  /* redefine as standard C datatype */
#define ioput    extern int  /* redefine as standard C datatype */
#define internal extern int  /* redefine as standard C datatype */

    /* Input output list for data path of the PDU */
input  Pdpra;  /* 0,15 */    /* instruction stream parcel from the queue */
input  Pdprb;  /* 16   */    /* instruction stream parcel from the queue */
input  eunpc;  /* 1,31 */    /* Next pc from IR stage of execute unit */
input  bradj;  /*  2   */    /* branch addition value (0,1,3) */
input  addpdu;           /* put the PDU address on the address bus */
input  ckpdpc;           /* used to enable clock of pdr.pc */

output pipc;   /* 1,31 */    /* Tag for Instruction cache */
output pbpbi;  /* 3,31 */    /* pbi master for prefetch buffer address */
output irpieq;               /* high when eunpc equal to pipc */

ioput  adbus;  /* 2,31 */    /* bidirectional main address bus */

internal offset;         /* output of offset mux and pc adder */
internal tpc;            /* target pc -- pc + offset */
internal spc;            /* sequential pc */
```

Figure 2a. Edited portion of Functional Simulator header file pdureg.h

```
/* PREFETCH DECODE UNIT datapath - Registers and adders and muxes */
#include "fsim.h"
#include "pdureg.h"

pdureg()
{
    /* In phase one, slaves are latched   */
    if  ( CLK1 )
    {   Pbpbi = pbpbi;       /* Latch every phase 1. */
        if  ( ckpdpc )       /* If clock enable is on */
            Pdpc = pdpc;     /* clock master into slave latch. */
    }

    if  ( CLK2 )
    {   if  ( addpdu )   /* double word align the pdu address */
            adbus = (Pbpbi & (~0x7)) | Pdusec << 3;
        irpieq = ( eunpc == pipc ); /* 31 bit compare */
    }

    if  ( CLK4 )
    {   offset += bradj << 1;    /* add in instruction length */
        tpc = offset + (pdpc & ~0x1); /* calculate target of branch */

        switch ( pbpcmx )    /* model pbpc 3 input mux */
        {  case R_PBPCMX:    pbpbi = rpc;
                   break;
           case T_PBPCMX:    pbpbi = tpc;
                   break;
           case P8_PBPCMX:   pbpbi = (Pbpbi + 8 ) & -8;
                   break;
           default:  error(WARNING,"Bad PBR mux control = %d", pbpcmx );
        }
    }
}
```

Figure 2b. Section of C code of Functional Simulator.

The functional simulator models the value of every signal at the end of a phase, but not necessarily the exact combination of logic gates necessary to produce that signal. All signals in the simulator are active high. Undefined signals, precharged nodes in dynamic logic and tri-stated outputs in I/O circuitry are not modelled. These features were left out to reduce the time it took to write the simulator and run the simulator, as well as to improve the clarity of the code. As a result some bugs (especially problems coming out of reset) were only discovered later with transistor level simulation.

In addition to internal consistency checks, like the validation of multiplexer controls to be legal values, the functional simulator was verified by dynamic comparison with the interpreter. No attempt was made to compare on a cycle-by-cycle basis (since the interpreter does not model cycles) or on the basis of any control signals. Data writes, either to internal registers or off-chip memory, were matched to those of the interpreter. Most errors were quickly revealed this way. Since the interpreter ran about 100 times faster than the functional simulator, it was run as a background process with the Fsim, so no test vector files needed to be stored, and virtually every run of the Fsim could be verified this way. Errors that delayed execution but eventually produced the correct result were not automatically detected. A few errors of this type were found, though they are not classical bugs in that the machine was executing correctly, just not optimally. On the whole, this was an extremely effective and efficient technique for catching functional bugs.

Schematic Logic Drawings: Draw

Schematics were used as the exact circuit and conceptual description of the machine. It is certainly possible to build complex chips without schematic diagrams. But as long as some of the layout is done by hand, schematics provide the clearest description for traditionally trained layout engineers. Engineers were comfortable with schematic representations. A few did not express the same appreciation for circuits described using textual representations, such as the Fsim. Some part of this is probably that schematics were able to convey 2-dimensional information, whereas text did not. A larger part was just familiarity, and using a system that is comfortable is an important consideration. We discovered and corrected many bugs at the schematic level before layout had begun.

The CRISP microprocessor was represented using 616 pages of schematics using the UNIX Circuit Design System.[6] Schematics were organized as a series of hierarchical macros. The top level macro went all the way to the describing bonding pads. Unlike many systems, the hierarchy did not stop at the gate level, each gate or macro had to ultimately be defined from p-fet and n-fet transistors. About one third of the schematics define basic cells such as nand gates, inverters, multiplexers and latches. In this fashion, the schematics represented the exact transistor connectivity of the entire layout. Schematics could freely mix transistors in with other macros. This allowed designers to make good use of dynamic logic and to invent their own cells, rather than simply trying to design from a catalog of predefined gates. Every transistor on in the layout, including a few in a test pattern, were precisely described by the schematics.

The UNIX Circuit Design System is composed of a number of small programs that work together, rather than a monolithic schematic system. The circuit editor, named **draw**, is simply a graphics editor that facilitates the drawing of shapes, pins, wires and text. Draw produces one binary graphics file for each 8½ x 11 inch schematic page. Each graphics file is first converted in to an ASCII netlist file. These individual netlist files are then run through a macro expander that produces the final netlist.

A macro is a schematic entity such as a nand gate or a multi-bit adder, that has inputs and outputs, and is composed of transistors or other macros. By convention the name of the file is the same as the type of the macro. A single macro may consist of a single or multiple schematic pages. For multiple page schematics the filename need be the same only until the first non-alphanumeric character. Since one of the main points of schematics is to make the intent of the circuit intelligible the macro expander also helps

eliminate needless repetition.

Figure 3 shows an example circuit that makes use of the macro expansion facilities. This figure has 7 inputs and 1 output, all but *ck4* and *precharge* are busses. Signal *bradj<0:1>* represents the two wires *bradj0* and *bradj1*, and *tpcmx[abd]* is expanded into the three wires *tpcmxa, tpcmxb*, and *tpcmxd*. A 10-bit register can be similarly drawn using the macro expander. Formal and actual parameters are connected through pin names, one interesting feature is that pin names and wire names need not be the same. This is convenient when dealing with busses at high levels of the hierarchy. Consider how confusing multiplexer *abdmux* would be if all 43 inputs and output pins had to be explicitly shown in the macro call. Each macro call has both a name (or names) and a type. Shown is a macro named *tpinc* of type *add2to10*. The call to the *tpinc* macro is drawn in two pieces, so that the precharge input will not visually impede the understanding of the function of the adder. The macro piece containing the *precharge* input would normally be placed on a separate page. The internal circuitry for the *tpinc* macro is shown in figure 4.

One might assume that the design would progress in a completely top down fashion, where pages of schematics would be the specification handed to an engineer to layout. This was the case only for the control logic in our chip. For the remainder of the logic the functional intent was understood by the layout engineer, who then designed the circuit to execute that function according to tradeoffs for layout area and timing. For example, a multiplexer function could be implemented with transmission gates, clocked inverters or and-or-invert gates. Schematics for the bottom levels of the hierarchy were then drawn or re-drawn according to what the layout was. The schematics were also driven by the layout in the sense that the physical topology was also reflected. For example, for a 3 input nand gate, we accurately reflected the order of the 3 inputs. This was necessary for later verification of the layout.

While the schematics were drawn to be as understandable as possible, there came a tradeoff in that they also had to be drawn to reflect the physical layout structure for use in verification. For example, a decoder might be too large to fit in a layout of a group of registers, but might fit conveniently into a hole in an adjacent adder; this had to be reflected in the schematics.

put example.g schematic here

put add2to10.g schematic here

Switch Level Simulation: Soisim

Transistor level simulation was handled with a switch level simulator called Soisim. The input to Soisim is a transistor level netlist containing transistor sizes and net capacitance. This ASCII input file is compiled before simulation begins. The compilation phase analyzes the circuit and produces additional information to facilitate simulation. For example, many gates are recognized and evaluated by function, rather than exhaustive transistor evaluation. Special cases usually hard for switch level simulators, such as the four transistor exclusive-or gate and memory sense amplifiers, are also recognized and handled properly. Another powerful speedup recognized by the compiler is to recognize memory cells and model rather than simulate them, as memory can be a large fraction of the total circuit. Capacitance information is used to handle dynamic logic circuits as well as detect potential problems with charge sharing. Transistor size information is used to evaluate ratioed logic.

Soisim has a number of interactive commands. Small circuits can easily be tested by setting input values, settling the circuit, and printing out nodes of interest. A benefit of the compilation process is that one may inquire as to which "gates" drive a particular node, where the notion of a gate has been automatically reverse engineered from the transistor level input netlist. The Soisim command interpreter follows the conventions of the UNIX Shell in two ways. One may traverse the hierarchy of a circuit using **cd** and **pwd** commands. Names use the character '/' as the hierarchy separator. Second, names may be specified by shell-like regular expressions using *, ?, and character classes. This is particularly convenient for dealing with groups of signals such as a 32-bit bus. Commands to the interpreter can be read from a UNIX file, and output from any command may be sent to a file rather than the terminal. The apparent command set of the Soisim interpreter can be extended with user supplied C programs compiled with Soisim, or by writing output to a file, then using combinations UNIX commands such as *awk*, *sort* and *grep*.

Soisim also has a non-interactive mode which can be driven from a C program. This is useful when the number of inputs and cycles of execution is too large to be conveniently typed in by hand. The values of C program variables can be bound to wires in a circuit to be simulated. Assigning a value of 0,1 or X to a variable will cause that value to be placed on the signal net before the next circuit evaluation begins. Variables must be specified as input, output, or a mixed input/output as specified by a control variable.

Backporting: Switch level simulation without vector files

The conversion from the functional simulator to schematics was a manual process. An engineer stared at the C code in the simulator and drew a series of gates to represent that code. Because of human intervention we needed a mechanism to guarantee that the gates in fact produced the same behavior as the C language statements. The mechanism we used to prove that the schematic logic was correct is called backporting. One by one, C-coded modules in the functional simulator are replaced by switch level simulation of the schematics for that module. Eventually the entire chip is modeled at the switch level by Soisim, and the Fsim is a skeleton which provides an operating environment, verification and debugging facilities. The result is an efficient mixed mode simulation capability.

Each module of the functional simulator, when called by the main driving procedure, takes all its inputs (which are C external variables) and, at the end of the phase, calculates appropriate values into its outputs (also C externals). When backporting a module, the C code is replaced by a special C code that binds a module's inputs to Soisim variables, issues a call to Soisim to settle the circuit, then maps the appropriate Soisim variables back to C variables for the module's outputs. The binding, settling and mapping function is handled automatically from the module's header file and the netlist file generated from the schematics.

The tool which generates the linking module automatically translates between the Fsim naming convention and the schematics naming convention. (The distinction exists primarily so the simulator can represent multi-bit signals as C integers, instead of a collection of 1-bit variables.) In addition, the tool provided another level of verification, since it complained if the inputs and outputs specified in the C header file did not agree precisely in number with the inputs and outputs specified in the schematics. Backporting more than one module at a time is a simple extension of the procedure to backport individual modules. For backporting the full chip, the I/O list was the specification of the chip's I/O.

All the facilities of the functional simulator are available to the backported simulator, so that the interactive debugging interface looks as it did without Soisim, all verification mechanisms function, and all Fsim variables are available for inspection by the debugger, including signals being simulated by Soisim. In addition, all nodes internal to the Soisim model could be examined or traced. The debugging paradigm is the same as the stand-alone Fsim: run a test while comparing its results with the background interpreter, and when a conflict occurs, inspect signals until the problem is revealed. Usually, once a bug was known, its cause could be uncovered in a few minutes. As a result, we did not use any test vector files until just before the chip mask date, when vector files were necessary for detailed I/O protocol verification as well as for the silicon wafer testing machines.

The speed of a backported functional simulator depends on the size and the nature of the circuit being simulated by Soisim. A small block of a few hundred transistors runs in nearly the same time as the bare Fsim. A full CRISP model, all 170,000 transistors, runs *very* slowly, on the order of two cycles per second of IBM 3081 CPU time. This is about two orders of magnitude slower than the bare Fsim. The process, even with the hierarchical names in the Soisim netlist flattened to save space, takes about 7 Megabytes of virtual memory space. Therefore, we ran a full-chip model infrequently, and only after we were reasonably certain that all of its sub-modules backported successfully. A full regression of the full-chip model through the 200-odd tests takes about three 3081-CPU-days to run half a million cycles.

Layout Tools: Mulga

The layout was accomplished by three different methods. The memories and final chip assembly was done with a VLSI graphics editor called GRED. GRED allowed the editing of arbitrary polygons and mask levels in a fixed absolute geometry system. The two control PLA's were generated automatically from high level C-like equations,[7] and major routing channels were generated automatically using the LTX2 routing system. The remainder, and by far the majority of the drawn layout was done using the Mulga system.

Mulga[8, 9, 10, 11] provides a layout system using symbolic virtual grid layout with compaction. Layout is entered using a high resolution color display and digitizing tablet for cursor control. Transistors, wires and contacts are symbolic entities that can be placed at the crossing point of conceptual grid lines. These grid lines appear in the symbolic layout as if a piece of graph paper was superimposed on the screen. Symbolic layout still requires that the designer decide on the relative placement of transistors and wires, but actual physical design rules for spacing can be ignored.

After a symbolic design is entered, is then goes through a compaction process yielding final physical mask data. Virtual grid compaction in Mulga is conceptually accomplished with two passes, first an X-compaction step, then a Y-compaction step. In the X-compaction pass, all elements on that X coordinate are examined to see how far they might be moved to the left without violating the physical design rules to elements on the virtual grid line to its left. A single fixed mask X coordinate is then assigned to all elements on that virtual grid line which is determined by the most constraining path. This step is then repeated for the next virtual grid line to the right, and so on. The Y-compaction step is repeated similarly, with the constraints also including diagonal clearances. After compaction, all X and Y virtual grid coordinates have

a corresponding physical coordinate. A program converts the symbolic data and physical coordinate information into the final physical mask data. Compaction is more a process of pushing objects closer together than one of re-arranging layout. The compaction system also automatically adds additional mask layers not covered by the symbolic representation, such as tubs and N+.

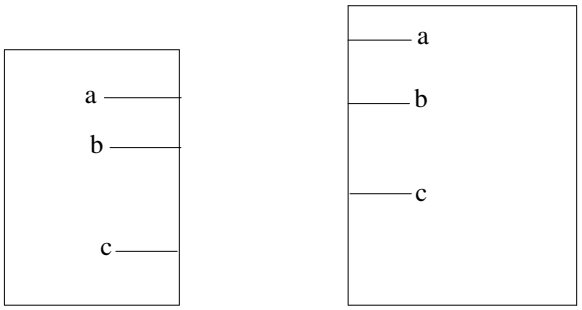


Figure 5a. Representation of 2 cells after compaction

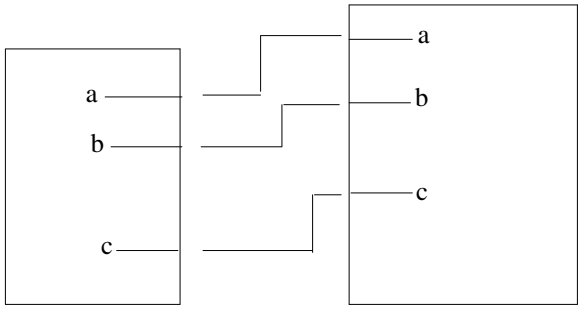


Figure 5b. Routing channel used to connect 2 cells with disjoint IO

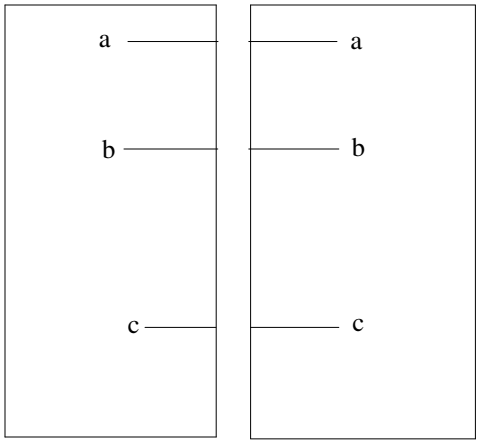


Figure 5c. 2 Cells expanded after pitchmatching now abut.

Using symbolic layout proved beneficial in several ways. Layout of cells seemed to progress about 3 times faster for our designers with symbolic layout rather than with more traditional absolute geometry "paint" style layout editors. Symbolic layout is easier to deal with in several ways. Objects can be placed

with a single atomic operation, rather than having to paint individual mask layers. For example, placing a single symbolic transistor is easier than drawing separate layers for each of diffusion, polysilicon and tubs. Time was also saved by not having to worry about particular design rules, for example the absolute length and width of transistors or the absolute spacing of wires; these would later be determined by the compactor.

Use of the compaction system also eliminated bookkeeping required for pitchmatching. Pitchmatching is necessary when many heterogeneous cells are packed in a matrix with precisely lined up interconnecting busses, as is common with datapath design. In figure 5a two cells need to be connected after layout and compaction, but the busses running though them are not aligned appropriately so that they may be butted together. If these cells are large, re-layout could be very time consuming. In a fixed geometry system one solution would be to interpose a routing channel between the blocks as in figure 5b. Such a routing channel could use large amounts of area, very unacceptable for microprocessor designs where area is always at a premium. Our execution unit datapath had 250 individual cells, each of which had to abut with cells on each of its four sides. The automatic pitchmatching used with the Mulga system aligned these cells by first compacting all cells to minimum size, then expanding the virtual grid lines of cells so that they abut exactly. Once a cell is compacted, adjacent virtual grids can be expanded without causing any design rule violations by holding the relative spacing of other grid lines fixed. Cells would now abut without the need of a routing channel between them, as shown in figure 5c.

Compaction allowed the layout to proceed long before the final design rules stabilized. When a design rule changed the effort was not in fixing the layout, but rather in changing some tables in the compactor. The only real concern was that all cells be compacted with the final version of the compactor. Our compactor was changed several times in the course of the layout, due both to design rule changes and to a few bugs in the compactor uncovered by use of a design rule checker. (A working compactor by definition does not produce any layout errors.) When we wished to evaluate a new circuit design for the ALU before 1.75µ processing was available, we simply took the symbolic layout and recompacted it using a compactor for 2.5µ design rules. Different compactors have also been used to create new layout based on preliminary design rules for future processes.

Circuit Extraction: Goalie

Two problems need to be solved after layout has been generated. First, is the layout correct? Second, what are the timing characteristics of the circuit? The first step in determining these answers is to automatically extract the circuit connectivity and capacitance from the final mask artwork. We used a new circuit extractor called Goalie.[12] Goalie's two most important features for a large chip design are that it is very space efficient, and also very fast compared to other circuit extraction systems. Many extraction systems would simply be unable to extract a chip this large. The working space required by Goalie is proportional to the square root of the size of the artwork being analyzed. Even though Goalie was space efficient, we still had problems due to the large amount of data involved. The final mask data for CRISP was 34 megabytes. The operating system for our largest IBM computer had process size limits of 10 megabytes per process, this limited us to extracting circuits of about 20,000 transistors. Larger circuits had to be extracted on machine with greater process size, but which were an order of magnitude slower. For a 35,000 transistor circuit, circuit extraction took about 12 hours on a VAX-11/750. Hence the CPU efficiency of Goalie was important to us. The final 172,163 transistor mask was extracted in about 6 hours using an Alliant multiprocessor, about 26 hours on a VAX-11/750. These extractions included parasitic capacitance and also generated intermediate files for simulation and netlist verification tools.

Netlist Comparison: Gemini

If schematics are used to specify layout, one needs a technique for certifying that the actual layout is as was intended. One might consider using simulation as a tool for verification, but this quickly becomes prohibitive as the size and complexity of circuits grow. Even if computer time were not a factor, catching a minor layout error would require perfect fault coverage, a very difficult prospect. Simulation can be considered a form of dynamic verification, we instead opted for static verification via netlist comparison. The schematics provide a transistor level netlist for the entire chip. The layout is extracted using Goalie to provide a second netlist. The two netlists are then compared for equality by a computer program called Gemini.[13] Gemini was written at Carnegie-Mellon University by Carl Ebeling and Offer Zajicek and was the only tool obtained from outside Bell Labs.

Gemini has a number of essential features for large circuits. First, its running time and space requirements are nearly linear with the size of the circuit. Second, it is convenient in that it does not depend on the actual signal or node names being equivalent; in fact, no names need to be supplied to get the program started. Third, when the circuits do not match, it does a reasonable job of producing diagnostics to help pinpoint the error.

A number of practical considerations have to be taken into account before relying on this method. One that almost caught us by surprise was the amount of virtual memory space that would be required for the finished chip simply due to the large number of transistors. Our final Gemini run took 110 Megabytes of virtual memory! This is still an average of only slightly more than 300 bytes per transistor, quite respectable. Most of the UNIX systems available to us provided for only 2 to 32 megabytes of virtual memory space per process. We were fortunately able to borrow a recently purchased Alliant minisupercomputer which allowed for up to 256 megabytes per process. Our project could have been in serious trouble without the use of this machine.

The sizes of files also became almost unmanageable. The Gemini input format was a flat ASCII database, with one line per transistor. Each line contained the device type (p or n), the names of the nets on the gate, source and drain, the length and width of the transistor, and a unique name for the transistor. For the extracted circuit the net names and transistor name contained no hierarchy, and were typically 6 characters in length (the default name for nets with no identifying text was "N" followed by a net number). The size of the Gemini file for the extracted chip was about 9 megabytes. The Gemini file generated from the schematics was over 25 megabytes, as the hierarchical names were each often long, for example **crisp/pfdata/pbufdata/array_d/nyble00/z/bit_L**. Other than disk space, one problem with large files was that they occasionally had to be moved between machines and few networks could transfer files that large in reasonable amounts of time. Magnetic tape ("tapenet") was used all too often.

It is imperative to start verifying small pieces of layout first, then gradually work to larger pieces. The intent is to verify a circuit where there are never more than a few errors, otherwise the diagnostics are not likely to be very useful. This incremental debugging approach required us to carefully draw the schematics so that there would be exact correspondence with layout blocks at various stages of the hierarchy. This required many tradeoffs between making the schematics easy to understand versus making them model some physical layout.

Although Gemini could often describe an error in terms of connectivity, finding the error in the physical layout could sometimes be difficult. Consider the case where a long wire is broken in two pieces separated only by a small slit. Looking for the error with the naked eye may be impossible at any resolution large enough to see the entire circuit. Several of our plotting programs had the ability to highlight a particular electrical net, thereby making the physical location of many problems plainly visible. The most difficult problem to identify turned out to be when power was accidentally shorted to ground creating a single

power net. We found no easy way to quickly detect the physical location of this particular error.

Gemini was extremely successful at finding bugs in what we would otherwise have thought was "perfect" layout. One of our strongest recommendations is that netlist comparison be a fundamental part of chip design methodology.

Timing Analysis: ADVICE and Leadout

Detailed timing analysis was first done with a detailed circuit analysis program called ADVICE,[14] a Bell Labs proprietary version similar to the more commonly available SPICE.[15] ADVICE models the device physics of transistors to produce highly accurate analog simulation based upon the actual process parameters. Input decks for ADVICE were initially hand extracted from plots, measuring transistor sizes and wire lengths via ruler and entering best approximations of values. Goalie was later modified to automatically extract an input deck for ADVICE, but it was still necessary to type in input stimulation waveform information. While this was merely tedious for small circuits, the job quickly became difficult for large circuits. Simulations of this kind had absolute limits of a few thousand devices due to both program memory requirements and excessive amounts of CPU time. ADVICE use concentrated around the design of the memories, drive and noise analysis in the IO frame, and on a few expected critical areas such as adder and ALU design.

One option we did not use for large scale chip timing was a less accurate multiple delay simulator such as MOTIS.[16] This could have been used to simulate the execution of the machine for many clock cycles at a given clock frequency, i.e. a dynamic timing analysis. In addition to the still massive CPU requirements this would have cost, we saw several more basic problems with this approach. First, errors caused by signal paths running too slow would likely be very difficult to find. One might have to examine hundreds of nodes to determine which signal was at fault; this seemed unreasonable for large complex circuits. Second, this type of vector driven simulation would have no guarantees of exercising and finding all worst case signal paths. Third, the information conveyed by a timing failure would show only the last gate in the chain. The designer needs to be told the entire path of a signal in order to fix it. For these reasons, we abandoned this technique of dynamic timing analysis in favor of static critical path analysis.

For static critical path analysis we used a new program called Leadout.[17] Leadout's main job is to tell the user what paths do not arrive at their destination soon enough for proper operation at a given clock frequency. Leadout analyzes a circuit over a single clock cycle in a way that is independent of the logical values of most nodes. It does take into account and exploit information about clocks, invariant signals and other controlling signals whose timing behavior is specified. Leadout assumes that a circuit will run correctly if clocked slowly enough. This correct behavior is used to produce a set of internal equations that describe the time of occurrence of signal changes for any frequency, and a set of constraints on these times that must be satisfied in order for the circuit to operate correctly. The result is a fast timing static timing analyzer that could handle our entire chip in reasonable time (minutes) on a VAX-11/750. Accuracy of Leadout is within 10% of the time predicted by ADVICE.

Because Leadout was fast and easy to use it was convenient in sizing transistors. Gates driving signals internal to small blocks of logic were first laid out using minimum size devices. This provided a benefit by reducing the gate loading for most signals. Leadout was then used to selectively identify slow paths, these were then resized with larger transistors. Leadout would provide immediate feedback on timing improvements from transistor sizing. Designer time was probably saved as no time was wasted optimizing non-critical paths. Before Leadout was available we had done transistor sizing based on fanout, assuming that a gate with a large load needed more drive. If a gate had more than an ample timing budget (slack time), then this unnecessarily increased both the area and capacitance on its inputs.

Leadout was not a substitute for ADVICE, but a tool that worked well with it. In fact, one of Leadout's commands is to produce a complete input deck for an ADVICE run for a critical path, including input stimulus and commands for producing plots. Leadout could not be used for circuits of a more analog nature, such as memory sense amplifiers.

Naming Conventions

What's in a name? Headaches. One of the biggest factors in making all the tools work together smoothly was in defining names. Our set of tools came from many sources, each with its own history for why what it had done was the "right" choice. The tools themselves were not even as much of a problem as the languages they had to deal with. The oldest of these was the XYMASK physical mask layout language, which had descended from the days of making printed circuit boards. XYMASK names could be a maximum of 8 uppercase alphanumeric characters, with no special characters such as underscore allowed. The C compilers on the large IBM machines initially also had restrictions of 7 characters on names, but would allow both upper and lower case, and underscore. Seven characters in a flat name space (i.e. no hierarchy) is just not enough. Fortunately the schematic system allowed hierarchical names of essentially unlimited length. In the end we decide not to try to force all CAD tools to use exactly the same names. We cared mainly about the schematics and layout, Gemini provided the key by producing a dictionary of equivalences as it matched the two nets. In practice, using a dictionary turned out to be only a minor nuisance, perhaps no worse than would have been the bookkeeping to try to keep 68,000 unique 8 character net names.

The first fight among any group of engineers is how to represent active low signals in their schematics. Here is how we solved the problem. Names could optionally be followed with an underscore and *tag* field. If the tag was the letter *L*, the signal would be considered active low; *xyz_L* is an example of an active low signal. There were often many variants of the same signal, a common variant was to distinguish the master and slave nodes of a latch. Slave nodes would contain the character *S* in the tag. Buffered versions of the same signal could contain a number in the tag, hence *xyz31_2SL* would be the second slave latched version of bit 31 of bus *xyz* active low. Tags could in practice contain any legal characters, and their use was merely a convention extendible whenever the need would arise.

Most of the tools had their own ASCII intermediate language. Interfacing different CAD systems could quite often be done with a relatively trivial translation program written in awk or C.

Our use of Gemini to match the schematics against the layout required that the nets match exactly. In particular, the order of inputs to gates had to match, so we needed a naming convention for gates. Our convention was that lower valued pins would be nearest the Vss and Vdd supply rails, higher valued pins would be nearest the output. For this definition, the value of 1 was less than 2, and the value of A was less than B. This convention seemed to work well even for gates such as the 3-2-1 input and-or-invert gate of figure 6.

Control the Source Code

In almost every case our project obtained private copies of the source programs for our tools. In retrospect, this was not only a wise choice, but essential in making the tools effective. Our tools came from several different sites, each with its own view of a particular problem. As a result, few of the tools would interface with each other in their original form. For example, our schematic entry system was originally designed to handle wire-wrapped circuit boards of a small number of TTL chips, not a large scale VLSI design. The wire-list format for TTL chips was not used directly by any of our tools, but a simple program converted the file into the format required by the Soisim simulator.

Figure 6. Example of naming convention for an And-Or-Invert gate.

Few of the tools were equipped to handle the massive size of the CRISP chip. In many cases a program would halt with a diagnostic saying that the size of our circuit exceeded the allowed size of some internal data structure. By having the source locally available we were able to quickly identify the problem, change our copy of the source, recompile, and be in operation again in only a few minutes. These errors were too frequent to have the problem fixed by some separate support organization, and of course occurred on nights and weekends when no official support would have been available. The operating system also required changes to support our tools, primarily in adjusting process size limits and adding device drivers.

Having the source to tools helped create dozens of spinoff tools. Because the amount of data was large, many repetitive tasks could be more efficiently handled by a program. The tradeoff was always to determine whether writing a one time tool was more expedient than just doing the brute force repetitive task by hand. By stealing lexical analyzers, parsers and code to build data structures from an existing tool, complex new tools could be put together in only a few hours. In the end we had created many new programs that significantly enhanced the productivity of our CAD environment.

UNIX

The CAD environment consists of more than just layout and simulation tools. For us, the best foundation for our tools was the UNIX operating system. UNIX was useful in several ways. First, it provided a number of support programs, we most frequently used *grep*, *awk* and *sort*. Second, its ubiquity was important in that we had to deal with a large number of different computers. The last thing an engineer wanted to have to deal with was a new operating system. In our particular case, VMS was favored by some of the CAD tool writers as its FORTRAN compiler ran somewhat faster than that offered by UNIX for the same computer hardware. In no case was this change to a non-UNIX operating system appreciated by our designers. Early in the project we had available an order of magnitude increase in performance available on a non-UNIX machine and turned it down.

Some Statistics

Numerology is often interesting. Table 1 contains some of the more relevant statistics gathered during the design of CRISP. The CRISP microprocessor was designed in 36 months by a team that averaged about 6 full time engineers. Additional support staff also made contributions. The sizes of programs are listed to give some feel as to the amount of effort required to install and maintain them. Some of these,

such as Mulga, are actually a suite of several programs. In this case the total lines of code for all programs in the suite is given whether we used them all or not.

1	1.75μ CMOS Microprocessor
172,163	transistors
68,341	nets
616	schematic pages
2,892	files in the mulga subdirectories
3,507	files in the draw subdirectories
26	VAX-11/750 hours to extract with goalie
370	megabyte directory needed to extract chip
5	VAX-11/750 hours to rasterize a 200μ/inch color versatec plot
240	disk megabytes reserved for rasterizing plots
110	megabyte process size for final gemini run
25,500,634	bytes schematic generated gemini file
9,032,022	bytes goalie extracted gemini file
34,205,222	bytes final mask data
3,295	interpreter lines of C
4,308	architectural simulator lines of C
4,654	gemini lines of C
16,670	fsim lines of C
20,023	draw suite lines of C
21,699	Leadout lines of C
36,738	goalie/soisim suite lines of C
119,401	mulga suite lines of C
3	IBM-3081 CPU days for full backport regression
0	card images

Table 1. Some statistics gathered during the design.

Weak Spots

On the whole we were very pleased with our methodology and the tools. One area for improvement would be in the layout system. Layout was done in two vastly different styles, symbolic layout with compaction and a physical layout editor; a single integrated layout system would have been vastly superior. Mulga was excellent for designing small cells, but not very good for large blocks and final chip assembly. Symbolic layout was very time consuming as the size of blocks got large, even for a tiny change as layout verification was necessary every time. Recompacting, then circuit extracting in order to verify with Gemini could take several hours for a large circuit. Many hours could have been saved had the layout editor kept electrical net information and been able to generate a Gemini netlist directly.

A small change in a symbolic cell might change the overall size of a layout block after compaction, as well as the position of its inputs and outputs. This greatly interfered with inserting the compacted layout into the physical layout system during chip assembly. A single integrated system allowing both compaction and physical placement might alleviate this problem.

Results

There now exist working CRISP chips that run at speeds at or above the design frequency. As of this writing we have found only two bugs in the initial silicon. One bug was a charge sharing problem, another was due to Miller effect problems involving a dynamic node. The charge sharing problem was at the interface of two blocks, and would not have been a problem if the timing interface had been constrained in a slightly different way. This bug could have been caught by our Soisim backporting, but due to either haste or lack of communication this simulation was done without ever having included extracted capacitances. The Miller effect problem was a clear design bug, obvious when one looked at the schematics. We had no

tools to effectively look for this type of analog design problem. The Miller effect bug could only have been seen through detailed ADVICE simulation, which was of little use unless one knew exactly where to look. If we had known where to look, we wouldn't have needed simulation to see the bug. A few of our first wafers had been held back from final metalization, we were able to make a minor change to the metal mask to correct these two bugs and obtain working chips from our first silicon wafers.

Conclusion

While many of the CAD tools have not been described, the overall methodology and experiences give a good view of how the CRISP microprocessor was constructed, and the obstructions along the way. We urge designers of large chips to follow a few key recommendations. First, use tools that are capable of handling circuits as large as the entire chip. Piecepart verification would have left us with many bugs. Second, verification efforts should start as early as possible, and be capable of incremental verification. Multiple levels of simulators worked well for us compared to the more traditional approach of not simulating until extracted layout was available. Third, static timing and layout verification is vastly superior to dynamic verification. Apart from the tools themselves, the hardware resources were never sufficient or perfectly reliable. Much time was spent dealing with computer issues of networking, broken hardware, porting software and a constant stream of bugs that stole time that might have been used for useful design.

Acknowledgements

Many people contributed to the tools that made CRISP a working chip, and to the incremental improvement of our CAD environment. Rae McLellan was a fundamental contributor to CRISP and pushed the tools to their limits. Kevin O'Connor was quick to pick up new tools, find their bugs, and request fixes and improvements. T. Szymanski wrote a vast quantity of our tools and was always willing to solve special requests. N. Weste and B. Ackland and the rest of the Mulga crew provided much assistance in getting Mulga running. A. Fraser is thanked for initiating Draw and J. Condon for supporting and improving it over the years. R. Cmelik rewrote the Draw macro expander and worked on many of the early C Machine tools. In addition to his layout responsibilities R. Freeman provided valuable software support for many of the layout tools during the project. S. Eggers is thanked for the meticulous reading and insightful comments on the paper as well as moral support. M. Kahrs and R. Freeman are thanked for the comments they made on this paper. Without the numerous meetings and position papers we could have finished the design much sooner. A.G. Fraser, V.A. Vyssotsky and A.A. Penzias are thanked for their constant moral support, encouragement, and making available the amazing resources of AT&T Bell Laboratories which made our job easier.

References

1. A. D. Berenbaum, D. R. Ditzel, H. R. McLellan, *Introduction to the CRISP Instruction Set Architecture,* Proceedings of the SPRING COMPCON87 (February, 1987).
2. D. R. Ditzel, H. R. McLellan, *Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero,* Submitted to the 14th Annual Symposium on Computer Architecture (June, 1987).
3. D. R. Ditzel, H. R. McLellan, *Register Allocation for Free: The C Machine Stack Cache,* Symposium on Architectural Support for Programming Languages and Operating Systems (March 1982).
4. A. D. Berenbaum, B. W. Colbry, D. R. Ditzel, R. D. Freeman, H. R. McLellan, K. J. O'Connor, M. Shoji, ''A Pipelined 32b Microprocessor with 13Kb of Cache Memory,'' *Proceedings of the 1987 International Solid State Circuits Conference* (February, 1987).
5. B. W. Kernighan, D. M. Ritchie, *The C Programming Language,* Prentice-Hall (1978).
6. A. G. Fraser, ''Circuit Design Aids,'' *The Bell System Technical Journal* **57**(6, Part 2), pp. 2233-2249 (July-August 1978).
7. P. Agrawal, M. J. Meyer, ''Automation in the Design of Finite-State Machines,'' *VLSI Design*, pp. 74-84 (September, 1984).
8. N. H. E. Weste, ''MULGA - An Interactive Symbolic Layout System for the Design of Integrated Circuits,'' *The Bell System Technical Journal* **60**(6, part 1), pp. 823-857 (July-August 1981).
9. N. H. E. Weste , ''Virtual Grid Symbolic Layout,'' *Proceedings of the 18th Design Automation Conference*, pp. 225-233. (June 1981).
10. N. Weste, B. Ackland, ''A Pragmatic Approach to Topological Symbolic IC Design,'' *Proceedings of the 1st International Conference on VLSI* , Edinburgh UK , pp. 117-129 (August 1981).
11. B. Ackland, N. Weste, ''An Automatic Assembly Tool for Virtual Grid Symbolic Layout,'' *Proceedings of the 2nd International Conference on VLSI*, Trondhiem Norway, pp. 457-466. (August 1983).
12. T. G. Szymanski, C. J Van Wyk, ''Goalie: A Space Efficient System for VLSI Artwork Analysis,'' *IEEE Design & TEST* **2**(3), pp. 64-72 (June, 1985).
13. C. Ebeling, O. Zajicek, *Validating VLSI Circuit Layout by Wirelist Comparison,* Proceedings of the Design Automation Conference (1983).
14. L. W. Nagel, ''ADVICE for Circuit Simulation,'' *Proceedings of the 1980 Symposium on Circuits and Systems* (April 28, 1980).
15. L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits,* University of California, Berkeley (May 1975).
16. B. R. Chawla, H. K. Gummel, P. Kozak, ''MOTIS - An MOS Timing Simulator,'' *IEEE Transactions on Circuits and Systems* **22**(12) (December, 1975).
17. T. G. Szymanski, ''LEADOUT: A Static Timing Analyzer for MOS Circuits,'' *Proceedings of the ICCAD Conference* (October 1986).