The Sloop ISA and the SMOK Toolkit

B. DUGAN AND J. ZAHORJAN

University of Washington, Seattle

Sloop-SMOK is a toolkit designed to improve the student design experience in a machine organization course taken by undergraduates in their first year as computer science majors. Students in this course have had some programming experience, and may have taken a one-quarter digital design course. Before Sloop-SMOK, assignments in this course were typically assembly language program implementations of functions related to architecture. The major goals in building Sloop-SMOK were to improve the relevance of homework assignments to machine organization and to emphasize some fundamental concepts of modern processors not easily addressed previously.

Sloop-SMOK has two components. The Sloop is a machine architecture designed to allow implementation of a modern version of the 6502, the processor used in one of the early Atari game stations. The Sloop defines a RISC ISA and a set of on-the-fly translations from the original 6502 CISC ISA into Sloop instructions. The Sloop Machine Organization Kit (SMOK) is a general-purpose software machine organization simulator. The components of a SMOK model are at the level of detail found in typical machine organization texts: ALUs, register files, logic gates, and the like. SMOK provides a graphical interface to construct and debug models.

As homework assignments, students use SMOK to build Sloop machines that successfully run most original 6502 games. Extensions to SMOK provide specific help with these Sloop models. Of particular importance is support for debugging: SMOK, when used to build a Sloop machine, runs a software 6502 simulator and compares the behavior of the student's Sloop machine against the simulator on a per-memory-operation basis. This feature simplifies debugging of Sloop machine organization by raising an error at the earliest cycle at which the student's machine is known to deviate from correct behavior.

Sloop-SMOK, including downloadable software, documentation, and course assignments, is available at

http://www.cs.washington.edu/software/SMOK

Categories and Subject Descriptors: C.0 **[Computer Systems Organization - General]**: Modeling of Computer Architecture; I.6.5 [**Simulation and Modeling**]: Model Development; K.3.1 [**Computers and Education**]: Computer Uses in Education
General Terms: Design
Additional Key Words and Phrases: Computer architecture, education, simulator

1. INTRODUCTION

Sloop-SMOK was designed during autumn 2000, and has been used in the undergraduate machine organization course at the University of Washington since winter 2001. Prior to the development of Sloop-SMOK, the bulk of homework assignments in that course were MIPS assembly language programming and questions from the text *Computer Organization & Design* by Patterson and Hennessy [1997]. With Sloop-SMOK, we

This work was supported by the State of Washington.
Authors' address: Department of Computer Science & Engineering, The University of Washington, Seattle, WA 98195-2350; email: <*zahorjan@cs.washington.edu*> < *dugan@cs.washington.edu*>.

hoped to address a number of problems. First, the existing homework did not align well with the emphasis of the course material, which was focused on machine organization. SMOK provides a tool that allows machine organization assignments. Second, the text, and hence the course, largely ignores the control component of a machine design. By assigning a complete machine implementation, we compel students to address this. Finally, the course itself concentrated on pipelining and caching, ignoring the important concept exploited in the increasingly common separation between a machine instruction set architecture (ISA) and the implementation of that ISA through translation into micro-operations. The Sloop includes this as a fundamental part of its design.

When used together, Sloop-SMOK allows students to build implementations of the 6502 processor. There are two components to this support. The Sloop defines a reduced instruction set computer (RISC) ISA and a set of on-the-fly translations from 6502 instructions into Sloop instructions. SMOK provides a general-purpose machine organization simulator as well as specialized Sloop/6502 support. By using these together, students have successfully built Sloop machines that execute most original Atari game read-only memory (ROM) modules.

We chose the 6502 as the target ISA for a number of reasons. First, we wanted a processor with a sufficiently small instruction set that an implementation for it would be feasible as part of a one-quarter course. Although students do not implement the 6502 ISA directly, we felt that limiting the complexity of the target would help keep the project manageable. Second, there are a number of public domain ROMs available for the 6502, so the result of a successful implementation of a Sloop machine is a system that can load and play a variety of games. Our experience indicated that students are often more motivated by homework resulting in games than in more abstract outcomes. Third, the combination of the 6502, the detailed description of the MIPS processor from the text, and the Sloop machine span a large chronology of processor designs (from a very complex instruction set computer (CISC) architecture to one exploiting instruction set translation). We felt that this context would help the students appreciate the lessons learned through decades of computer architecture research. Finally, on-the-fly translation of CISC instructions to RISC-like instructions is a technique commonly employed by modern architects attempting to improve the performance of legacy architectures, most famously the Intel x86. Using a similar strategy in a somewhat more circumscribed environment gives the students an appreciation of the techniques employed and challenges faced by architects of state-of-the-art processor cores produced by companies such as Intel, AMD, and TransMeta.

Sloop-SMOK is in its fourth quarter of use. The first offering was taught by one of its authors (Dugan); the others were taught by other faculty in our department. Our original intention was to have students build a sequence of machines, following the Patterson and Hennessy text (a single-cycle implementation, then a multicycle implementation, and finally a pipelined implementation), and to observe their relative cycle-time benefits and the design complexity costs. Due to time limitations (our courses are only 10 weeks long, and the Sloop-SMOK homework is only part of the course), we have not yet achieved this. Instead, we have assigned an initial SMOK assignment (building a simple stack machine) and then a single-cycle Sloop machine. Work by the authors shows that a pipelined implementation would be a reasonable assignment given more time.

Sloop-SMOK download, documentation, and assignments are available from the following web address: *http://www.cs.washington.edu/software/SMOK*

2. SMOK

Despite its name, the Sloop machine organization kit (SMOK) is a general-purpose simulator. Special-purpose support of Sloop machine models is provided, as explained in the next section, but SMOK can be and has been used to build other classes of machines.

This section provides a brief introduction to SMOK, concentrating on the issues of model construction, model evaluation, model debugging, and extensibility. A very brief overview of the internal software structure is also provided for those interested in extending the SMOK source code.

2.1 Model Construction

Although designed to allow a number of modes of interaction, as explained in Section 2.5, SMOK models are overwhelmingly constructed through a Windows-based graphical user interface (GUI). Figure 1 shows a screenshot of the main model window for a very simple circuit (a counter).

This model has three standard machine components (a constant register, a latched register, and an adder), a synthetic model component (the halt component), and a status component (the Titlebox component). Components are inserted into the model through a right-click menu. *http://www.cs.washington.edu/software/SMOK/ComponentRef/* lists the set of currently available components.

Component connections are also established through the GUI. To create a connection, the user selects a component input port by clicking on it. In Figure 1, input ports are shown as small, grey boxes inside components. If the next selection operation is of an output port of some other component, a connection is made. (All components in this model have only a single output port, and so for simplicity these are not drawn by SMOK. Instead, the user simply selects the component to establish it as the connection source. The register component has two inputs, a data input and a write-enable bit.) SMOK provides connection undo/redo operations of unbounded depth. These have proved to be vital to usability, as it is not uncommon to inadvertently establish connections when the user's intention was merely to select components.

SMOK implements only a very rudimentary connection layout scheme. The scheme is insensitive to the positions of other components, and so the default layout may be unacceptable, for instance, passing through other components. While the connection layout has no effect on the function of the model, it is important visually, as reducing clutter is an aid to modelers. SMOK allows both component and connection positions to be adjusted manually. A Manhattan connection geometry is enforced, with each connection consisting of five segments. All but the two end segments may be moved by the user any distance perpendicular to the direction of the segment. Additionally, components may be scaled (both up and down), the locations of input and output connection points may be adjusted to help users see details of their models, and various other options are provided (e.g., labeling connection endpoints rather than drawing the complete connection).

Individual components have various settable parameters. All components include name and width parameters. The name is the text tag used in display. The width is the number of bits of the output. The components in Figure 1 have all been set to a of width 16 bits.

SMOK models may include container components. These provide for hierarchical modeling. A container may include any other SMOK (or Sloop) component, including

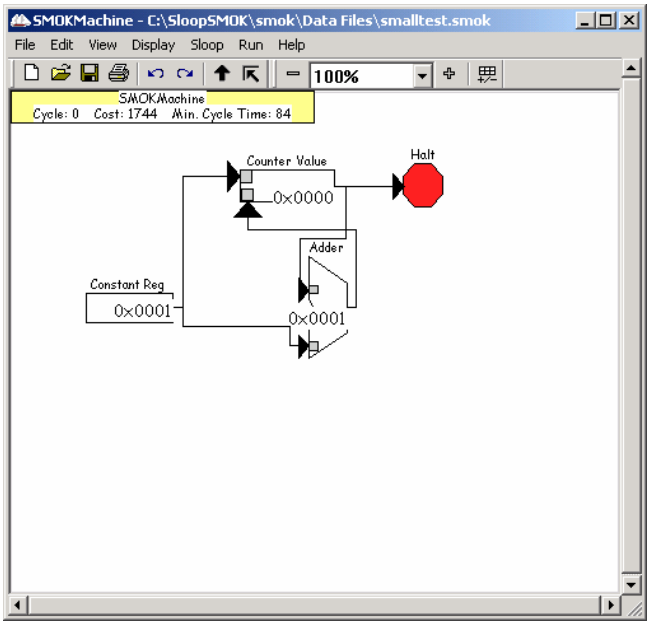

Fig. 1. A simple SMOK model.

other containers. Containers may be individually stored as files. In contrast to storing the entire model as a file, when read back, a container component file recreates the container (and all its subcomponents) in the existing model; reading a model file causes the current model to be replaced.

Containers are useful in a number of ways. For one, they allow the model builder to organize the model. For instance, control components (which are simply containers with the added feature that they and their connections are drawn in red) allow the machine control to be laid out in an easily identified and isolated area. Containers also facilitate team construction and component testing of models. Each member of the team can design their piece inside a container and store it to a file. The overall model is created by inserting the various container components into a single model and connecting them. While testing each piece, the container inputs can be connected to debug input components and the outputs to debug output components. Each debug input component solicits an input value from the user at each clock up. Each debug output component writes the value of its input to a dedicated output window each clock down.

2.2 Model Evaluation

The only notion of time in the simulation of a SMOK model is the clock cycle, which is divided into clock up and clock down phases. On each clock up, each component reads its inputs and asserts the output(s) resulting from those inputs. On clock down, those components that latch inputs do so.

SMOK ensures that all upstream components (i.e., those involved in providing input) of each component have had sufficient time to settle to stable outputs before updating the

destination component. This is done through an on-demand evaluation of all component outputs.

As well as this dynamic simulation of the model, SMOK provides static evaluations to estimate space costs and minimum clock time. These are shown in the (yellow) TitleBox component of Figure 1 (which also includes the model title). The model space cost is simply the sum of the space costs of the individual components. The model minimum cycle time is derived using a dynamic programming evaluation that determines the latest time at which each input to each component will have stabilized, and adding the user-supplied component latency to it to determine the earliest time at which that component's output will be stable. Because this is done statically, it can be an overestimate. For instance, a circuit might be designed in which the selected input to a multiplexor is always on a short delay path, but the one unselected is on a longer path. In this case SMOK will use the delay of the irrelevant input in estimating minimum cycle time. Despite this limitation, we chose to use a static delay analysis rather than a dynamic one in an attempt to maximize simulation speed; a dynamic evaluation would add a large percentage overhead to the per-cycle simulation cost of each component.

Component space and time cost parameters are optionally provided by the user on a per-component-class basis. These are placed in a *smok.ini* file that is read on application startup. The cost specifications can be either constants or functions. In the latter case, the function can include symbolic variables representing an individual component's width or size, where "size" has a natural meaning. (For example, for a register file, it is the number of registers.) Such functions are templates applied individually to each instantiated component of the component class

2.3 Model Debugging

We were quite concerned with how students would debug models, imagining situations in which errors were evident only after thousands of simulated cycles, making the errors difficult to track down. The primary, general-purpose facility provided by SMOK is output tracing/logging. (More specialized support is provided by Sloop-SMOK, as explained in the next section.) Additionally, the halt component allows “breakpoints” to be set.

Output tracing occurs in two ways. By default, each component's inputs and outputs are written to a trace window each cycle. This allows the modeler to scroll back through the logged activity to determine when and why the model deviated from expected behavior. Because the trace window adds a lot of overhead to simulation (logging a component's activity is much more expensive than simply simulating it), the standard SMOK distribution comes with two executables, one including this standard trace window and one in which logging has been eliminated through conditional compilation. The version without tracing is roughly two orders of magnitude faster than the version that includes it.

Additionally, a synthetic model component, debug output, is provided by SMOK. A debug output component mimics a hardware probe. Each debug output has a single input and a single output. A connection from some component A to another component B is monitored by splicing a debug output into that connection: A's output becomes the input to the debug output component, which becomes the input to B. Functionally, the debug output is a no-op, simply passing its input through to its output. However, each debug output creates a dedicated text window into which the value flowing across the

connection is written each cycle. This allows a controlled amount of trace information to be obtained even when running the faster version of the simulator.

We also wanted to allow for testing of subsets of full models. This is facilitated through the debug input component class. A debug input has no input connection in the model. Rather, it is connected to the keyboard. Each clock up, the debug input requests a value from the user and then passes that value on to its output. This allows the user to inject an arbitrary stream of values into a circuit for testing.

Finally, controlling termination of the simulation is an important part of debugging. SMOK allows the usual Step N Cycles and Go Indefinitely invocations of the simulator. It also provides a synthetic halt component that allows conditional termination. A halt component has a single input, which is compared against a user settable value each cycle. When the two values have the parameterizable relationship associated with that halt component, simulation is paused. Halt components can be used to represent the end of the desired simulation; for instance, a model containing a simulated machine's PC simulation might halt when the PC reaches some predetermined address. Halt components can also be used as breakpoints: simulation can proceed at full speed until some model circuit condition is reached when execution pauses. At that point, the modeler can step the simulation, or even edit the model before advancing.

2.4 Extensibility

Functional extensibility in SMOK is provided through the use of container components. New, higher functionality component types can be constructed using the basic set of built-in components and then saved to files. These saved containers then have the properties of built-in component classes: one or more instances can be inserted into a model, be connected to other model components, and be individually parameterized using the same UI controls as in the built-in component types. While not satisfactory for every situation (this would be a difficult way to provide a new component class that included floating-point support, for instance), it has been sufficient for the kinds of uses we have encountered in using SMOK for course work.

It is also possible to extend SMOK through modification of the source code to create new component classes. Making this as simple as possible was one of the design goals of the SMOK software implementation. At a minimum, this requires providing a single procedure defining the new component class's operation during a clock up event. SMOK is written in C++, and all component class implementations are derived from the `SMOKComponent` base class. `SMOKComponent` provides a `Fetch(n)` method, which returns the value of the `n`th component input (after first settling all upstream components, if necessary). It also provides a `SetAndMaskOutput (value, output number)` to assert computed results on a specific component output port. The existing infrastructure then handles the synchronization needed to assure that the inputs are valid when used, that the component is updated each cycle, and that its results are propagated in the modeled circuit. For new component classes that take some action on clock down, an additional procedure must be provided, exploiting a similar set of facilities provided by the `SMOKComponent` base class.

Additionally, it is usually desirable, and to some extent necessary, to provide some UI support. Although optional, a specialized visual representation of the new component class may be desired. In this case a `DrawComponent()` procedure must be written. Components are drawn using a system-independent set of drawing routines provided by

SMOK. The `DrawComponent()` method operates in a unit-square coordinate system, isolating it from issues of layout and display scaling. (A default inherited from a base class can be used if no specialized visualization is required.)

Finally, there are a few additional tasks. The component class name must be entered into tables used by the UI code to construct menus. If the new component class has new per-component parameters, methods must be written to provide these parameters and to set them, allowing their integration into the machinery that presents update dialogs to the user. Finally, serialization methods are needed to save these specialized parameters in model files and to read them back. As with the functional and drawing methods above, the interfaces provided by SMOK isolate these methods from nearly all details of SMOK code. Additionally, it is most often possible to derive new component classes from existing SMOK classes, obviating many of these methods. Injection of the specialized Sloop support components, for instance, required only the functional and drawing methods described above.

2.5 Software Internals: System Requirements, Software Structure, and Portability

An unusual amount of effort has been devoted to trying to make the SMOK source code as system-independent as possible. This resulted in part from the Windows-phobia of one of the authors, and in part from the evolution of the implementers' aspirations. Our original design goal did not, for reasons of time, include a GUI front-end. Instead, we intended that students create models by providing a single C++ procedure that would create the model through a set of C++ `new` operations. Besides avoiding 70% of the current SMOK implementation that is devoted to UI, this approach had the advantage of leveraging the compiler's type system in ensuring that models were well formed (a kind of error checking that must be done by the SMOK implementation at this point). When it became clear that a GUI would at least be a tremendous advantage, if not a necessity, we strove to preserve the original capabilities. Additionally, while implementing the GUI for Windows, we wanted to be sure that a port to other system (e.g., X windows) would be as easy as possible.

The result is that the SMOK implementation consists of two largely isolated pieces: the functional code needed for model simulation and the GUI code needed to manipulate the visual representation of the model. Additionally, the GUI code is organized as a system-independent base class that defines a drawing and user input/output Application Programmer Interface (API). Only the lowest-level methods of this API need be implemented for a particular system (through subclassing the generic UI class). These methods include drawing routines such as `DrawLine()`, but also include all the low-level details of user interaction. For example, while the generic UI defines events like "present component parameters to user for editing," the decision that double-clicking on a component triggers this event, and that the parameters should be displayed in a modal dialog box, is made by the Windows-specific subclass.

Needless to say, the decisions to support model creation both through the GUI and through source code extensions, to allow operation of the system in the complete absence of a GUI, to attempt to optimize the simulator for speed (which requires insulating it from responsibility for notifying the UI of state changes), and to make the UI as portable as possible, resulted in the most complicated aspects of software and a substantial increase in the number of methods and lines of code. By far the most complicated aspect had to do with object creation. Objects could be created both in the underlying simulator

(through the programmatic extensions) and in the UI code. Because the two are isolated from each other, simulation component objects are completely distinct from the UI objects presenting them, and so creation of an object on either side of the simulation-UI divide requires that a corresponding component be created on the other side. To minimize the presence of the UI in the simulation code, this symmetric creation occurs in the constructors for the objects, rather than through additional calls from the creating client. This means that neither side can count on a fully created object existing on the other side; the GUI cannot assume that the simulation object is fully realized, or vice-versa. This has lead to numerous "constructor races" in which the GUI accessed simulation objects that were partially formed at best, with unpleasant results. Similar difficulties occur during object destruction.

In addition to the problems presented by allowing object creation in two components that were as separate as possible, we also encountered the typical problems of trying to write portable software in a nonportable language/environment. For instance, a raw, system-dependent event is discovered by the system-specific UI class. It then passes this as a generic event to the generic UI base class, which most often then makes one or more calls back to the system-dependent UI code to interact with the user and deal with the event. Because system-dependent state is often needed to resolve the event (for instance, a mouse click triggering insertion of a new component into the model needs the screen coordinates of the click event when the component is ultimately displayed), this seemingly simple decomposition of interactions into generic semantics and system-dependent specifics cannot be done in an entirely clean manner.

While SMOK is written to be portable, the only existing implementation at the time of this writing runs on Windows. (More particularly, SMOK has been developed and tested on Windows 2000, and is known to be stable there. It is known not to run on Windows 95, although the issue is the relatively trivial (but forbidding) one of DLLs. We have no experience with other varieties of Windows.) Those interested in porting to other systems should get in touch with the authors.

SMOK's resource requirements are relatively modest. A model with 65 components has a memory footprint of 5.4MB. The SMOK distribution, which includes two executables, requires 1.33 MB of disk storage.

2.6 Simulation Performance

There are two performance-critical aspects to SMOK. One is the GUI, and in particular the performance of dragging components, which requires frequent screen updates. The difficulty in this application is that dragging a component also changes the position of its connections, and those connections can extend to any portion of the display. Thus, the region that needs to be repainted is irregular (it follows the connections as well as the bounding box of the component), and when using the simple technique of a rectangular bounding box can span the entire display.

To address the window-update problem, SMOK uses an in-memory buffer for drawing and then blits the buffer to the window. Measurements showed that repainting the background in the buffer was an enormously expensive operation, so this is never done. Instead, SMOK "unpaints" components that have moved by repainting them in the background color. It then repaints all components in their normal colors, a relatively fast operation, and blits the result. Using this technique has provided adequately smooth dragging performance for models of the sizes we have encountered (100+ components).

Table I. Measured Simulation Performance
Each entry is the average cost per SMOK component of executing one SMOK cycle, in microseconds. The timings were taken on a 700MHz Pentium III running Windows 2000 SP2

	No Trace Window	***Trace Window Minimized***	***Trace Window Visible***
Figure 1 model (4 components)	0.162	45.537	334.385
SMOK multiplier (28 components)	0.200	22.109	460.915
Sloop machine (129 components)	0.595	33.519	308.596

The other critical aspect of performance is simulation speed. Because our target is a machine that executes interactive game programs, we made a special attempt to optimize simulation as much as possible. As one example, all inputs to all components are always connected to some other component, even when there are unconnected inputs in the user's model. We use shadow "zero-source" internal components for this, specialized to provide their output quickly. This eliminates the need to do repeated testing for unconnected inputs while simulating.

Table I shows measured simulation performance. Each entry is the number of microseconds per SMOK cycle for a given model, divided by the number of SMOK components in that model. The rows correspond to three different models, two containing only SMOK-native components, and one including Sloop components (and so incurring the overhead of software 6502 simulator used to check correctness of the SMOK implementation of the Sloop architecture). Because performance is very strongly affected by the amount of trace (debugging) output desired, the columns show different scenarios. "No Trace Window" is the version of SMOK compiled to eliminate tracing. The other two columns show the tracing version, in one case with the trace output window visible and in the other with it minimized. The distinctions between these show the Windows overhead of updating visible windows (or, if you prefer, the Windows optimization of updates to minimized windows).

Note that a simulated machine will not operate at the speeds in Table I because the times listed are per-component. For example, the Sloop machine model listed there executes a single *machine* cycle in approximately 70 microseconds. This means that users cannot play legacy ROMs in real time.

3. AN EXAMPLE

3.1 Introduction

A typical SMOK assignment entails building a general-purpose computer. This section presents an example of using SMOK to build a simple stack calculator. A machine at this level of complexity might be used as a warm-up assignment for students.

Table II. Example Stack Machine Instruction Set

Instruction	*Example Encoding*	*Semantics*
ADD	0x00000000	Stack[$SP – 1] = Stack[$SP-1] + Stack[$SP-2]; $SP = $SP - 1
SUB	0x10000000	Stack[$SP – 1] = Stack[$SP-1] - Stack[$SP-2]; $SP = $SP -1
MULT	0x20000000	Stack[$SP – 1] = Stack[$SP-1] * Stack[$SP-2]; $SP = $SP - 1
DIV	0x30000000	Stack[$SP – 1] = Stack[$SP-1] / Stack[$SP-2]; $SP = $SP - 1
PUSH	0x80000005	Stack[$SP] = 5; $SP = $SP + 1

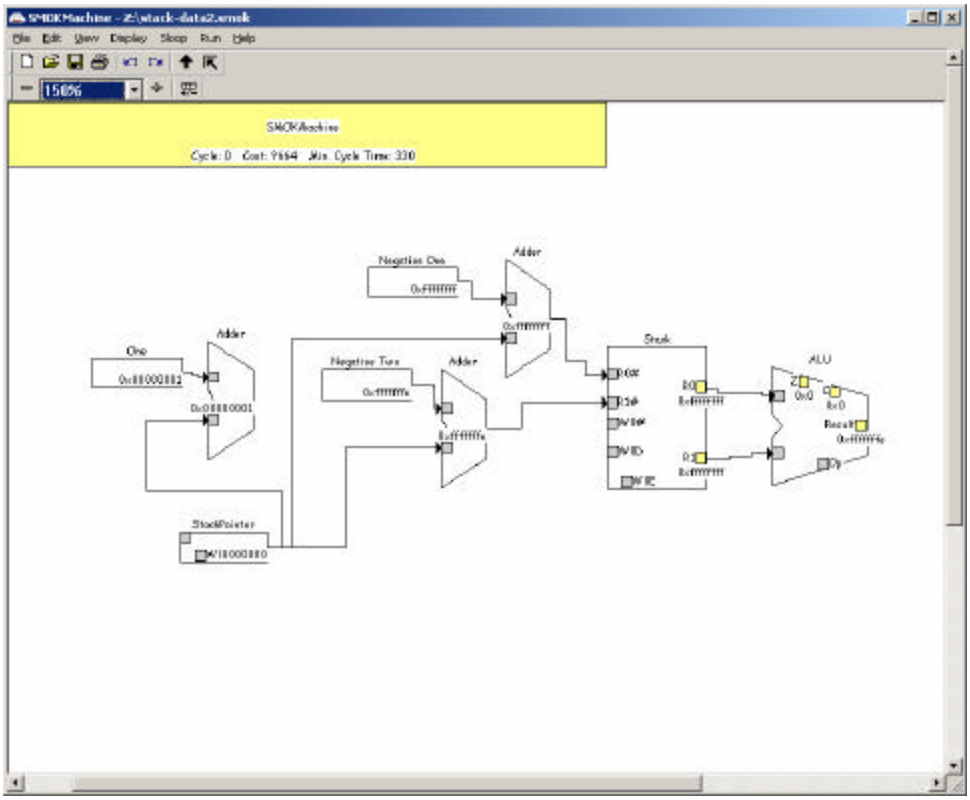

Fig. 2. Datapath components for the stack calculator.

3.2 The Stack Calculator Instruction Set

The stack calculator ISA defines just five instructions: four arithmetic instructions and a PUSH instruction. The encoding is regularized, so that the "opcode" always lives in bits 28 through 31. In the case of arithmetic instructions, the low 28 bits are wasted. In the case of the PUSH instruction, immediate data is encoded in the low 28 bits. We use a

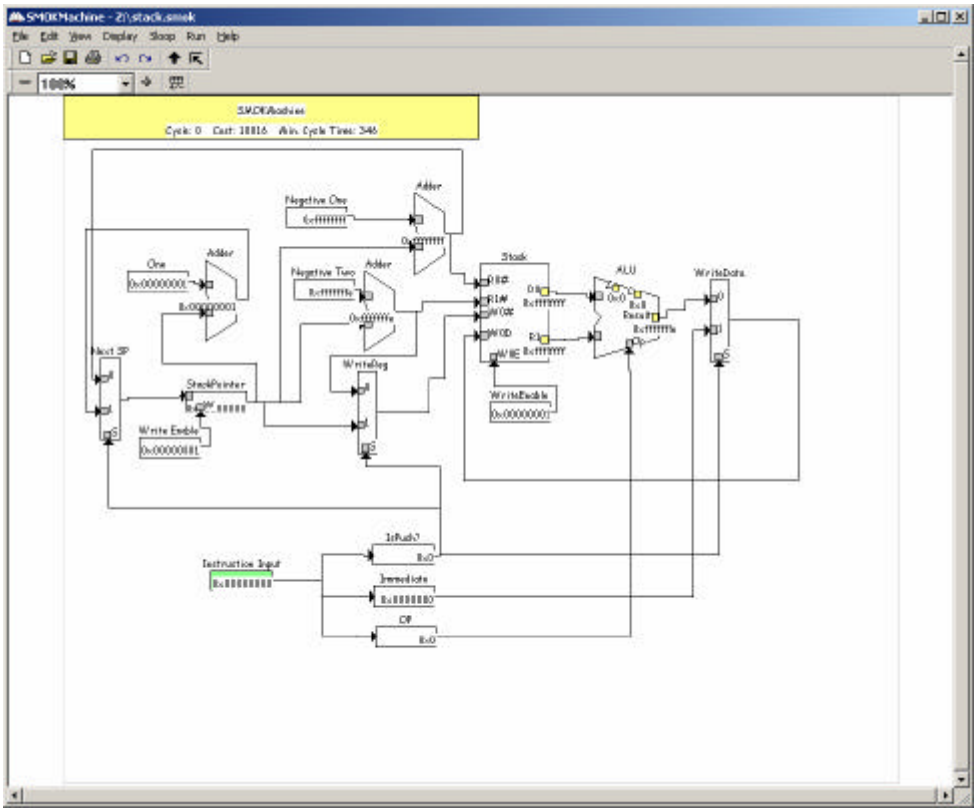

Fig. 3. Stack machine with control.

register transfer language (RTL) notation to define the semantics of our instructions. Table II defines the five instructions.

3.3 The Datapath

After opening the application, the user begins by adding the components that will comprise the datapath of their machine. Figure 2 shows a screen snapshot of the model up to this point. Notice that the model consists of just a handful of components: a register file represents the stack; an ALU performs arithmetic operations; and a simple register contains the stack pointer value. The model also contains three adders. The adder on the left is used to compute the stack pointer value for the next cycle (in case of a push), which is just the current stack pointer value plus one. The other two adders compute the register numbers that may be used to read stack values out of the register file. As the model indicates, these are negative offsets (negative one and negative two) from the stack pointer. The model also contains three "constant" registers, which represent constant values in the machine.

3.4 Adding Control

To add control to their model, the user typically adds an assortment of multiplexors and logic gates. In Figure 3, the user has added control elements to his or her model.

The control consists primarily of three multiplexors. The first (labeled WriteData) selects between the output of the ALU and the immediate data decoded from the instruction. The output of this multiplexor is attached to the write data input of the register file. The second (labeled NextSP) selects the stack pointer for the next cycle. In the case of the PUSH instruction, it will select the incremented stack pointer; in the case of arithmetic instructions, it will select the decremented stack pointer. The output of this multiplexor is connected to the input of the stack pointer register. The third (labeled WriteReg) selects the appropriate write register number (stack location). In the case of the PUSH instruction, it selects the stack pointer; in the case of arithmetic instructions, it selects the stack pointer decremented by two.

Additionally, a few minor components have been added to control the write-enable ports on the register file and the stack pointer register. Since the stack and the stack pointer are written on every instruction, these can be simply wired to constant registers set to the value one. Of course, a more complicated machine would require additional control components to decide when the stack or stack pointer should be modified.

In order to simplify the presentation, this particular machine does not store its program in memory. Rather, its instructions are input by means of a special component known as a DebugInput. On each simulation cycle, the user is simply prompted for a new instruction value.

The output of the DebugInput passes through three bit extractors, which extract the ALU operation, the immediate data, and the push bit from the instruction. The output from these extractors are routed to the ALU, the WriteData multiplexor, and the WriteReg multiplexor, respectively.

3.5 Testing the Model

When the user starts the simulation, the DebugInput component will prompt them for a new instruction on each cycle. Figure 4 shows the input dialog box.

After three cycles, suppose the user has entered three push instructions. At this point, they may wish to examine the contents of the stack, which is illustrated in Figure 5.

After entering a multiply instruction on the fourth cycle, the image of Figure 6 results.

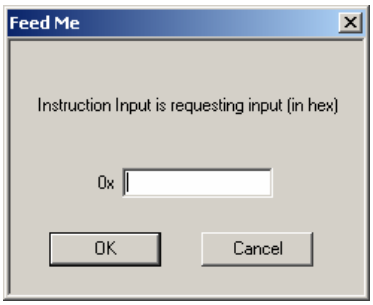

Fig. 4. Debug component input dialog.

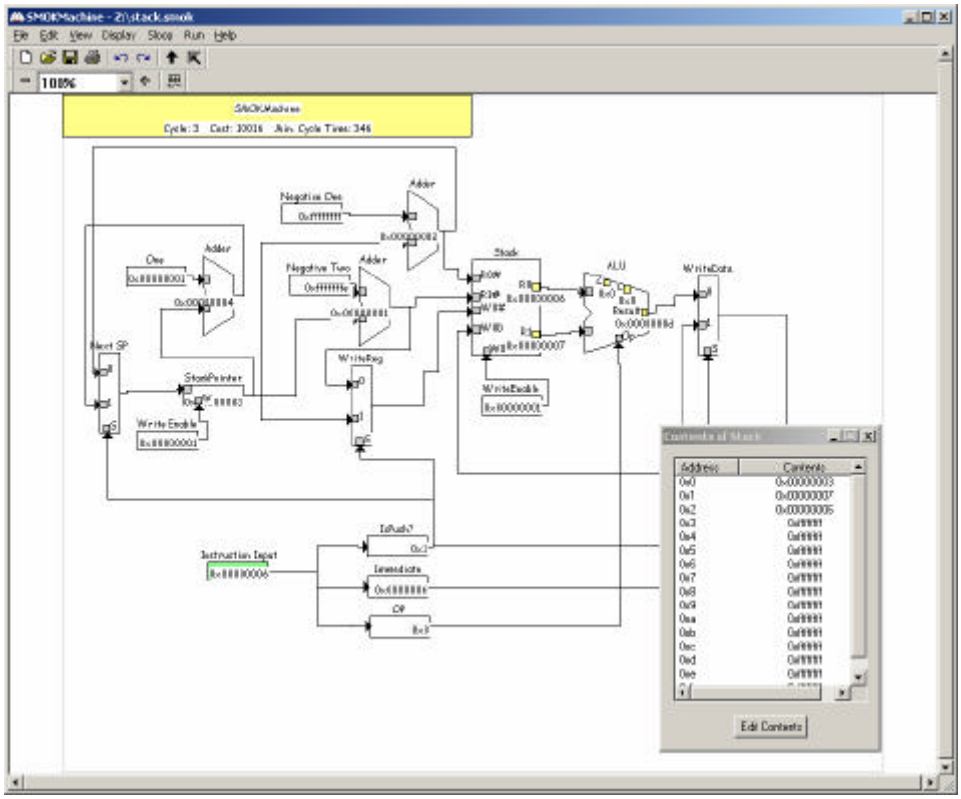

Fig. 5. Model state after the third cycle.

Notice the changes that have occurred to the contents of the register file (stack), as well as the stack pointer. The stack pointer has been decremented by one and the result of the multiply has been pushed onto the stack.

3.6 Extensions

The above example can be extended in a number of interesting ways. Adding a program memory, for instance, would be accomplished by simply adding a memory unit, a memory interface unit, a program-counter register, and an adder. On each cycle, the program counter would be used to fetch the next instruction from memory. The adder would increment the program counter by a fixed amount. Furthermore, the ISA could be extended to include branch or load/store instructions, which would require the students to build more complicated control for updating the program-counter and selecting inputs to the register file. Finally, students might be asked to handle error conditions such as stack over- or under-flow.

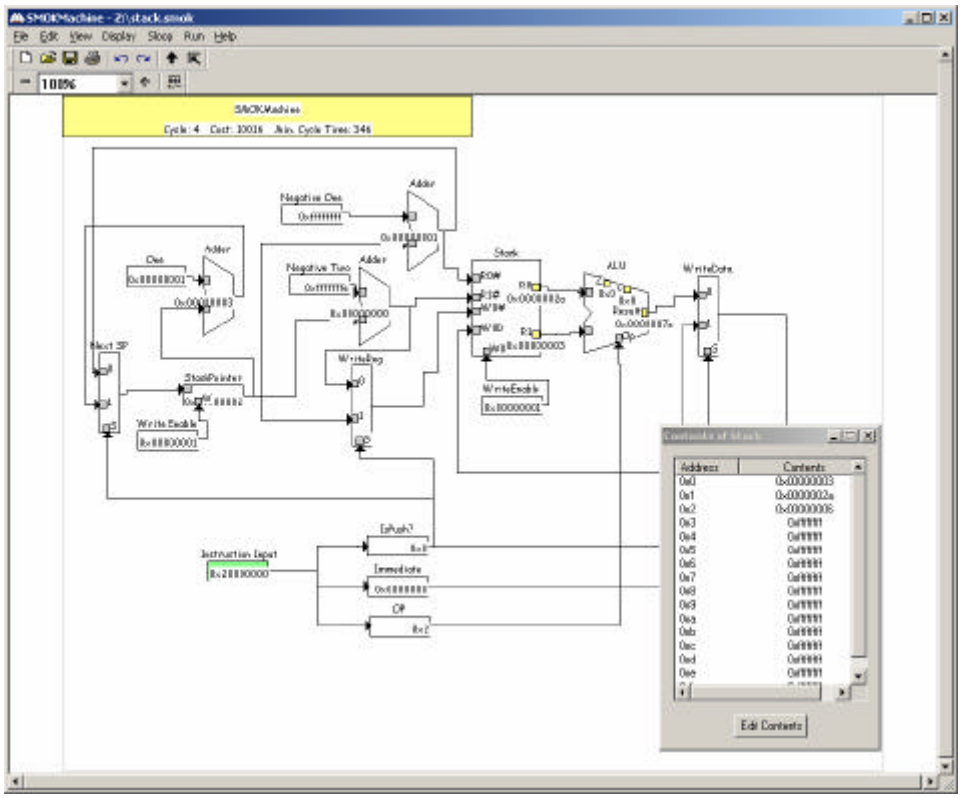

Fig. 6. Model state after the fourth cycle.

4. THE SLOOP ISA

4.1 Introduction

The Sloop ISA specifies a simple RISC-like architecture. While the Sloop machine could be used in isolation, its primary goal is to provide an easy-to-pipeline CPU core for the 6502 architecture. Each 6502 instruction has a simple translation into a small sequence (usually no more than two or three) of Sloop instructions, which can then be executed efficiently in a pipelined processor. Due to the need to support the 6502 ISA, Sloop contains a few non-RISC-like features, such as a status register.

This section introduces the Sloop architecture (and so the machine built by students in our courses). The full specification for the Sloop architecture can be found at *http://www.cs.washington.edu/software/SMOK.* The on-the-fly translation of 6502 instructions into Sloop instructions is presented in a subsequent section.

4.2 Registers

The Sloop ISA defines eight, 32-bit-wide programmer-visible registers, which are all "general purpose" in that there are no restrictions on the kind of data they may contain. Table III summarizes the registers.

Table III. Sloop Registers

Number	*Name*	*Notes*
0	R0	Zero register (NoOp on write)
1	RT	Temporary register
2	RS	Stack pointer
3	RA	Accumulator register
4	RX	X index register
5	RY	Y index register
6	RP	Status register
7	RPC	Program counter

Table IV. Sloop Instructions

Mnemonic	*Opcode*	*Function*	*Encoding*	*Notes*
NOP	0	N/A	All zero	Do nothing
ADD	1	2	R-type	RD <- RS + RT
ADDI	2	N/A	I-type	RT <- RS + IMM
SUB	1	3	R-type	RD <- RS – RT
SUBI	3	N/A	I-type	RT <- RS – IMM
AND	1	4	R-type	RD <- RS & RT
ANDI	4	N/A	I-type	RT <- RS & IMM
OR	1	5	R-type	RD <- RS | RT
ORI	5	N/A	I-type	RT <- RS | IMM
XOR	1	6	R-type	RD <- RS ^ RT
XORI	6	N/A	I-type	RT <- RS ^ IMM
SLL	1	7	R-type	RD <- RS << 1
SRL	1	8	R-type	RD <- RS >> 1 (unsigned)
ROL	1	9	R-type	RD <- rotate_left(RS, 1)
ROR	1	10	R-type	RD <- rotate_right(RS, 1)
CMP	1	11	R-type	RD <- compare(RT, RS)
BEQZ	7	N/A	B-type	Branch iff SP & mask == 0
BNEZ	8	N/A	B-type	Branch iff SP & mask != 0
J	9	N/A	J-type	Jump unconditionally to IMM
JR	10	N/A	I-type	Jump to the address in RT
LD	11	N/A	I-type	RT <- MEM[RS + IMM] (byte transfer)
ST	12	N/A	I-type	MEM[RS + IMM] <- RT (byte transfer)
LDW	13	N/A	I-type	RT <- MEM[RS + IMM] (2-byte transfer)
STW	14	N/A	I-type	MEM[RS + IMM] <- RT (2-byte transfer)

Most of the registers above map to the 6502 registers, with the exception of R0 and RT. A sequence of Sloop instructions that translates a 6502 instruction typically uses the registers in a way that the machine state can be easily mapped back into the 6502 world.

4.3 Instruction Encoding

Every Sloop instruction is 32 bits in length and is encoded in one of four formats. Readers familiar with standard RISC instruction formats may wish to skip the format descriptions that follow.

4,3.1 *R-Type Format.* The R-type format is used to encode three-register operations. An example of such an instruction is one that adds the contents of two registers and

places the result into a third. This format encodes three register numbers, as well as the opcode. Additionally, a function field is used to define the ALU operation that should be used.

4.3.2 *I-Type Format.* The I-type is used to encode instructions that operate on a small, immediate operand. An example of such an instruction is one that adds a 16-bit immediate operand to the contents of a register and places the result into a second register. This format encodes two register numbers, an opcode, and a 16-bit immediate value.

4.3.3 *B-Type Format.* The B-type format is used to encode branch instructions. To support 6502-style branching (which is based on the state of the status register) we encode a mask in our branch instructions. This mask is logically ANDed to the status register, and then branches are taken or not taken based on the "zero-ness" of the result.

4.3.4 *J-Type Format.* J-type is used to encode jump instructions. The 6502 supports 16-bit addressing, and the J-type format simply encodes the target address in the low 16 bits of the instruction.

4.4 Instructions

The Sloop ISA defines a very small set of instructions. These are shown in Table IV.

5. 6502 TO SLOOP INSTRUCTION TRANSLATION

5.1 Brief Overview of the 6502

The 6502 represents a milestone in processor technology. It was one of the first general-purpose, single-chip CPUs introduced at a price low enough to find widespread adoption in a variety of home game machines and computers, including systems developed by Commodore, Apple, and Atari. Architecturally, the 6502 is an 8-bit accumulator-based system, with 10 or more addressing modes and over 70 operations. The 6502 had only a few non-general-purpose registers, including a program counter, a stack pointer, two index registers, an accumulator, and a status register (described in some detail, above). The addressing modes and operations were not orthogonal, that is, not every addressing mode is usable by every operation. In all, there are over 200 legal operation-addressing mode combinations, compactly encoded in 8-bit opcodes. Immediate addressing modes could operate on 16-bit entities, so that the total instruction length varied from one to three bytes. The 6502 has 16 bits of memory addressing. The actual processor used in the Atari 2600 Video Computer System (VCS) is the 6507, which is in most ways identical to the 6502, aside from a 12-bit address bus. This small address space ultimately places a severe limitation on the complexity of games that could be published for the Atari 2600. In time, however, engineers devised an ingenious workaround, called bankswitching. The impact of bankswitched ROMs is discussed below.

5.2 Translation Strategy

Typically, each 6502 instruction will translate into a small sequence (generally one to three) of Sloop instructions. Once this translation has taken place, pipelining techniques may be used to efficiently execute the resulting instruction stream.

Due to the variable length of 6502 instructions, the fetch unit must fetch at least 3 bytes (the size of the longest 6502 instruction) from the program memory. We make few assumptions about how the fetch unit operates. Either the path to memory can be widened to 24 bits, or the fetch unit may operate asynchronously with the Sloop core. When enough data from the 6502 stream has been fetched (i.e., the above quantities are all available) the translation unit can proceed. The translation unit will inform the fetch unit of how many bytes have been consumed, so that the fetch unit will know from where to get the next 6502 instruction.

In most cases, much of the data required for an operation is already available in the result of the 3-byte 6502 fetch. If the operation is on immediate data, it has already been fetched, so the translation is trivial. If the operation requires that its operand be fetched, the situation becomes more complicated. In some addressing modes, the fetch unit has already fetched the required base (or absolute) address. In these cases, all that remains is to load the operand and then operate on it. The most complicated addressing modes are those that require several fetches to get the operand. These include the absolute indirect and indirect indexed modes present in the 6502 ISA.

By way of example, we show translations for the 6502 ADC (add to accumulator) instruction in each of its eight addressing modes.

In Table V, the mnemonics IB and IW simply refer to the next byte (immediate byte) and the next two bytes (immediate word) in the 6502 instruction stream. The 6502 ADC instruction in the absolute addressing mode, for example, contains the absolute address in the two bytes that follow the instruction. The translator, having fetched these bytes, can construct the appropriate two Sloop instructions. The first instruction loads the temporary register from the address specified by the 16-bit offset from the zero register. The second instruction sums the temporary register with the accumulator register, placing the result in the temporary register. Other translations are more complicated, of course, but the strategy is the same: generate a sequence of Sloop instructions that fetch the operand into a temporary register, and then perform the fundamental operation on that data.

Table V. Example 6502 to Sloop Instruction Translations

6502 Instruction	*6502 Addressing Mode*	*Sloop Sequence*
ADC (Add to accumulator)	Immediate	`ADDI $RA, $RA, IBS`
	Zero pointer	`LD  $RT, IB($R0)` `ADD  $RA, $RA, $RT`
	Absolute	`LD  0, $RT, IW($R0)` `ADD  $RA, $RA, $RT`
	Absolute indexed (X index register)	`LD  $RT, IW($RX)` `ADD  $RA, $RA, $RT`
	Absolute indexed (Y index register)	`LD  $RT, IW($RY)` `ADD  $RA, $RA, $RT`
	Zero page indexed	`LD  $RT, IB($RY)` `ADD  $RA, $RA, $RT`

6502 Instruction	*6502 Addressing Mode*	*Sloop Sequence*

	Indexed indirect	`LD $RT, IB($RX)`
		`LD $RT, 0($RT)`
		`ADD $RA, $RA, $RT`

	Indirect indexed	`LD $RT, IB($R0)`
		`ADD $RT, $RT, $RY`
		`LD $RT, 0($RT)`
		`ADD $RA, $RA, $RT`

Addressing modes and instructions are not orthogonal in the 6502. In other words, not every addressing mode can be used with every instruction. In all, there are over 200 instruction/address mode combinations in the 6502 ISA. For the sake of brevity, we do not show translations for all of the combinations. However, a complete specification of the translation scheme can be found at *http://www.cs.washington.edu/software/SMOK.*

6. IMPLEMENTING THE SLOOP MACHINE IN SMOK

6.1 New Components

To implement a 6502-compatible Sloop machine in SMOK, we needed to add five new components to the SMOK toolkit, listed below:

1. SloopMemory: allows the user to choose and load a ROM image from file.
2. SloopALU: very similar to a standard ALU, but with a few extra inputs and outputs. The biggest difference is that the SloopALU handles the setting of status bits internally, so that the students do not need to worry about that task.
3. SloopRegisterFile: again, like other register files, except that it can read three values per cycle (two generic registers and the status register), and write two (status register (always) and a selectable register number).
4. SloopMemoryInterface: again, similar to other memory interfaces, but can write either byte or 2-byte entities. Also, the memory interface handles the setting of status bits internally.
5. SloopInstructionFetch: is the instruction fetch/translation unit. This unit handles the nasty task of fetching and translating 6502 instructions and translating them into sequences of Sloop instructions, as well as maintaining a buffer of to-be-executed Sloop instructions. It also computes the PC on any cycle, and makes it available to the rest of the datapath. (It has to do this, since only it knows how many bytes were actually fetched to find the boundary of the next instruction.) In this model, the PC is actually maintained inside the SloopInstructionFetch unit. Machine implementers can instruct the IF to fetch from a new location by asserting a SetPC line and providing the new PC to fetch from.

It should be emphasized that it is possible to implement a 6502 compatible Sloop machine using the generic palette of SMOK components. However, the existence of the status register in the 6502 considerably complicates this implementation. There exists a subset of 6502 instructions that may tangentially affect the status register. For example, assume the accumulator contains the value 1 and is then decremented. On that cycle, the accumulator register needs to be written and the Z bit in the status register needs to be set as well. Taken alone, handling this case is not difficult. However, the fact that the various 6502 instructions do not set the status bits in a consistent manner means that we

needed a compact and simple way to remember where and when to set the bits. For this reason, we developed the notion of a *status mode*. A status mode defines an "algorithm" for deciding which status bits to set under what conditions. Sloop instructions contain a field that represents the status mode, which is determined by the translation unit. The status mode is passed along with the Sloop instruction corresponding to the 6502 instruction that would have affected the status bits. Even with the status mode, however, the students would be required to erect a considerable amount of circuitry just to manage

the modification of the status bits. To make their lives easier, we internalized the modification of the status bits in our new components. The new components typically take the old status bits and status mode as inputs, and provide a possibly modified set of status bits as outputs. The student simply needs to pass the status bits through the datapath and to be sure to update them inside of the register file.

6.2 Basic Datapath (Single Cycle)

Figure 7 shows a (simplified) sketch of the Sloop single cycle datapath. Students were provided with this sketch and asked to implement control units for the ALU as well as the rest of the machine. Note the StatusRegister lines that run between the register file, the ALU, and the memory unit. As discussed above, by handling the setting of the status bits within the Sloop components, we greatly simplified the task of managing these bits for the students. Students merely had to connect status bit lines correctly between the components.

6.3 Implementing the Components

Implementing the new SMOK components was relatively easy thanks to the STELLA Atari 2600 simulator [Bradford 2002]. STELLA implements a complete software simulation of the Atari 2600 game machine. It handles the recognition and loading of the many formats of ROMs (cartridge games), as well as the simulation of the TIA (television interface adaptor), event handling, and of course the 6507 processor itself.

We were able to integrate the bulk of the STELLA code behind the SloopMemory component; when a user loads a new ROM, the appropriate STELLA simulation objects are created, including a graphical display window that simulates the television screen used for game play. Then, on each SMOK simulation cycle, the InstructionFetch unit fetches, decodes, and translates the next 6502 instruction via the STELLA memory model. Modifications to the SloopRegisterFile are mirrored in the STELLA register set; the loads/stores via the SloopMemoryInterface are similarly passed on to the STELLA memory model.

6.4 Error Checking

In order to assist the student's debugging process, the system actually runs a "shadow" simulation, which is a simple software simulation of the 6502. At the end of each cycle we compare the 6502 register set to the student's register file and flag discrepancies. The shadow simulation also allows us to verify memory reads and writes, and to report these errors at the earliest possible moment.

6.5 A Pipelined Implementation

Implementing a pipelined Sloop machine that is compatible with the 6502 is still an open problem. While building a Sloop machine with a standard five-stage (instruction fetch, decode, execute, memory, and writeback) pipeline is not difficult, the memory

organization unique to the Atari use of the 6507 does cause some complications. The 6507 used in the Atari system can only generate 12-bit memory addresses, limiting its physical address space to 4KB. To take advantage of larger ROMs, bankswitched memory was used. Reads and writes to special memory addresses caused the ROM to switch memory banks. If a program wished to execute instructions in the high 4KB of an 8KB ROM, for example, it might touch a memory address that would cause subsequent reads to issue to the high memory bank.

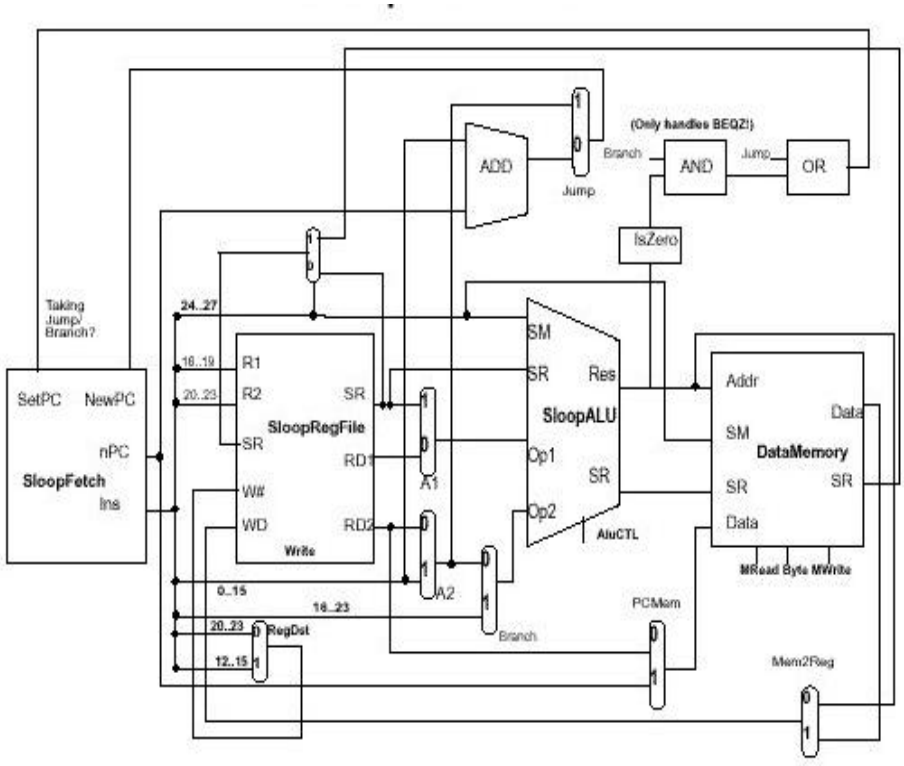

Fig. 7. Sloop single cycle datapath.

Because the Fetch/Translation unit aggressively fetches bytes from ROM memory, it may inadvertently fetch addresses that can initiate a bankswitch, especially in the face of branch instructions. The solution, of course, is to pad all branches with a number of NOPs. Otherwise, the problem can be defined away by specifying that only 4KB or smaller ROMs are compatible. The latter solution is not optimal, as many of the more exciting games use larger ROMs.

The status register also complicates the task of appropriately forwarding data throughout the datapath. Not only do normal registers possibly need to be forwarded, but the status register may also need to be forwarded.

7. RELATED WORK

There are a large number of simulators available for teaching machine organization and architecture. Wolffe et al. [2002] present a taxonomy of these simulators, which identifies seven categories, as shown in Table VI.

Viewed on its own, the SMOK toolkit is most closely related to the digital logic and advanced microarchitecture classes of simulators. Like the former, it does not impose any specific machine architecture or organization, but rather can be used to implement

Table VI. Simulator Categories (from Wolffe [2002])

Simulator Category	*Description*	*Representative Example*
Historical machine	Typically an instruction-level simulation of a machine no longer available.	*SIMH* [Computer System Simulation Project]
Digital logic	Logic-level simulations, often of both functional and timing characteristics	*DesignWorks* [Capilano Computing]
Simple Hypothetical machine	Simulation of simplified machines for pedagogy (often at ISA level)	*Compiler/Architecture Simulation for Learning and Experimentation* [CASLE]
Intermediate instruction s*/et	ISA level simulations of intermediate complexity	*SPIM* [Larus]
Advanced microarchitecture	Most often datapath level simulation of a complicated machine.	*DLXView* [DLXView]
Multiprocessor	Simulators that allow and focus on implementation of multi- and parallel processors	*SimOS* [SimOS]
Memory subsystem	Cache and other memory hierarchy simulators	*Dinero* [Edler et al.]

almost whatever is desired. Like the latter, it includes higher-level components, such as ALUs, as basic building blocks. Of course there are some tradeoffs required to achieve this balance. SMOK's timing model is very simple, ignoring layout issues, being limited to a static analysis of parameterized component delays instead. Since no architecture is imposed, there is no compiler or assembler support to produce executables.

When used with the Sloop ISA, the package acquires additional resemblance to historical machine simulators, in that the machine produced runs 6502 executables; some rigidity of organization is introduced as well. However, the focus for students is to build a machine that executes the Sloop architecture, whose pedagogic focus is on-the-fly instruction translation. While some of the less clean features of the 6502 demand specialized support at the component level, the basic activity the students engage in is still to design and implement a suitable machine organization.

Finally, SMOK provides facilities to decompose a large project into smaller pieces, facilitating group work in much the same way that a software project can be decomposed into modules.

8. CONCLUSIONS

Sloop-SMOK implements an active, animated simulation of a nontrivial processor. SMOK provides the right level of abstraction for building and animating models out of

familiar components. SMOK animates the behavior and function of datapath diagrams, which ordinarily require many pages of static diagrams to trace even trivial instruction sequences.

From an extensibility perspective, the SMOK system proved itself to be well designed. While one author (Zahorjan) was responsible for the development of the SMOK infrastructure, the other author (Dugan) worked on the design and implementation of the Sloop ISA as well as the 6502/Sloop translation unit. At the end, Dugan was able

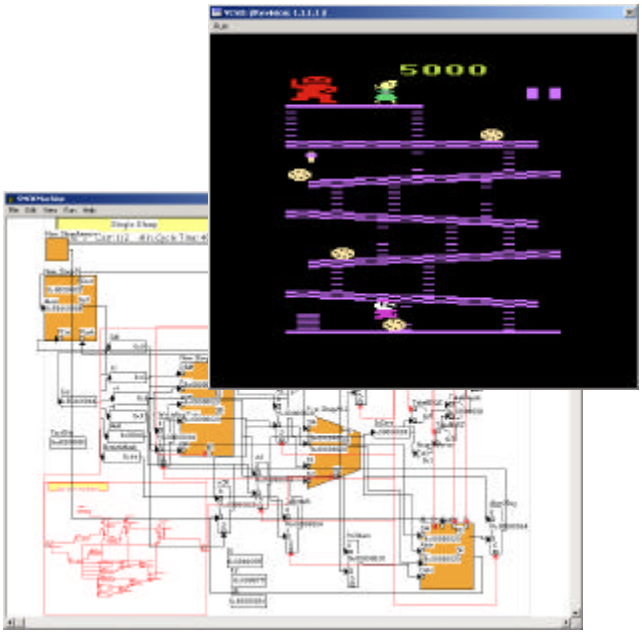

Fig. 8. A Sloop machine implemented as a SMOK model.

to create the new SMOK components required to implement Sloop in a short period of time. Development of the new components and the creation of a functioning Sloop model inside SMOK took this author about a day.

From an educational perspective, the development of SMOK/Sloop was a great success. Student feedback was positive, and the reliability and usability of the system compared favorably with similar systems such as DesignWorks [Capilano Computing]. Students also found that they could apply lessons learned in other courses (such as Karnaugh maps from digital design courses) to minimize the logic required for implementing control units. In building a RISC-like machine that could execute a dynamically translated CISC instruction stream they learned a number of valuable lessons. First, they received detailed exposure to a CISC instruction set. Second, they implemented a simplified, yet non-trivial, RISC machine, which gave meaning to the endless datapath diagrams so common in the textbook approach. Finally, in linking their RISC machine core to the CISC instruction translator, they gained deeper insight into performance-enhancing strategies used by today's cutting-edge processors.

REFERENCES

BRADFORD, M. 2002. Stella: A multiplatform Atari 2600 VCS simulator. *http://www.redlinelabs.com/stella*.
CAPILANO COMPUTING. 2002. *http://www.capilano.com*.
CASLE. 2002. http://shay.ecn.purdue.edu/~casle
DLXVIEW. 2002. http://yara.ecn.purdue.edu/~teamaaa/dlxview.
COMPUTER HISTORY SIMULATION PROJECT. 2002. http://simh.trailing-edge.com.
EDLER, J. AND HILL, M. D. 2002. Dinero IV: Trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/~markhill/DineroIV.
LARUS, J. 2002. SPIM: A MIPS R2000/R3000 simulator. http://www.cs.wisc.edu/~larus/spim.html.
PATTERSON, D. A. AND HENNESSY, J. L. 1997. *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann.
SIMOS. 2002. http://simos.stanford.edu/introduction.html.
WOLFFE, G. S., YURCIK, W., OSBORNE, H., AND HOLLIDAY, M. A. 2002. Teaching computer organization/architecture with limited resources using simulators. In *Proceedings of the ACM SIGCSE 2002 Conference* (Cincinnati, OH). ACM Press, New York, NY.

Received November 2001; accepted January 2002.