ABSTRACT

The objectives of this paper are twofold. In the first place we discuss some issues related to the formal description of computer systems and how these issues were handled in a specific project, the selection of a standard computer architecture for the Army/Navy Computer Family Architecture (CFA) project. The second purpose is to present a methodology for automatically gathering architectural data which can be used for evaluation and comparison purposes. We will not discuss the rationale behind the selection of specific test programs and the statistical experiment set up to ascertain the influence of the programmers, the test programs, and the machine architecture on the results. These issues and the actual results of the experiment belong in a companion paper.

Formal descriptions of three candidate architectures (IBM S/370, DEC PDP-II, and Interdata 8/32) were written in ISP, a computer description language. The ISP descriptions of the three architectures were used to simulate the execution of assembly language test programs. The measurements collected during the program simulations were stored into a data base for post-processing. Automating the data collection process not only eliminated tedious and potentially error prone hand calculations, but also provided the means to gather information about dynamic program behavior, information that would be almost impossible to calculate manually.

INTRODUCTION

There have been many attempts to specify computer architectures in some formal notation. The CFA project included, to our knowledge, the first attempt to describe the complete instruction set of several large, commercially available architectures. The candidate architectures were the IBM S/370, DEC PDP-11, and the Interdata 8/32. The experiment described in this paper involved the preparation of formal computer descriptions, the execution of machine language programs under an instrumented simulator, and the collection of data used to evaluate the architectures. Three aspects of the experiment are important to observe: (1) we did not implement specific simulators, tailored for each architecture; the system used in this project is a general purpose computer simulator driven by a formal machine description; (2) we executed a large number of test programs, each ranging from less than a dozen instructions to several hundred instructions; (3) we used real programs that had been executed on actual physical machines and then used to initialize the simulators.

The Naval Research Laboratory selected ISP as the notation to formally describe the candidate machines. This decision was based on the availability of expertise and software support at CMU and in the fact that ISP was better suited than other candidate notations for describing a computer architecture, independently of timing and other

* The work reported in this paper was sponsored in part by the Naval Research Laboratory, the National Science Foundation (under grant G332758X), and the Defense Advanced Research Projects Agency, ARPA (under contract F44620-73-C-0074).
A TYPICAL ISP DESCRIPTION

The ISP notation was developed to formalize the information normally given in basic machine manuals and to supplement or, if possible, eventually replace the "programming reference manuals." Hence its essential requirements were readability, completeness, flexibility, and brevity.

The original notation was introduced for descriptive purposes and, in the context of a book, certain ambiguities were permitted. For more formal use, the notation had to be revised and a language named ISPL was developed between 1973–1975. Further developments on the notation continue at CMU, and a language tentatively named ISPS is being implemented. For the remainder of this paper we shall refer exclusively to ISPL, the dialect used in the description of the CFA architectures.

The example shown in Figure 1 is derived from the IBM S/370 ISP description. We will only present the main declarations and the instruction interpretation cycle.

**The control flow for all instructions in Figure 1 follows a well defined path. The main body of the ISP description is defined by the Run procedure which continuously performs a loop of instruction cycles (IFetch followed by IExec). After an instruction has been executed, a special section of code (INT) is executed. INT checks for the presence of exceptional conditions (errors or external interrupts) and performs the proper context switching to handle these conditions.

The instruction fetch section (IFetch) reads the first half-word of the instructions and from the first two bits (Instr(0) and Instr(1)) it computes the length of the instruction (PSW(32:33)) and updates the program counter (PSW(40:63)). IFetch then proceeds to read one or two more half-words, the rest of the instruction.

The instruction execution section (IExec) uses the first two bits of the instruction (Instr(0:1)) to select an instruction-type specific section. The RR, RX, RSSI, and SS sections handle the corresponding instruction types. RX, RSSI, and SS begin by computing the effective address of the operand(s). After this step is completed the next 6 bits of the instruction (Instr(2:7)) are used to select a 'routine' which describes the behavior of the instruction.

If any errors are detected during the instruction cycle (address boundary errors, illegal operations, storage protections, etc.) the rest of the instruction is aborted and the proper error code is set in the PSW. A premature termination allows the interrupt handler (INT) to take care of the situation (the usual mechanism is to switch PSWs thus automatically starting the execution of interrupt specific system routines).

We have tried to keep the example as simple as possible by avoiding any details beyond those necessary to follow the example. In particular, the reader might have noticed that we were making explicit references to fields of the Instruction Register (Instr) and the Program Status Word (PSW). It is clear that when we deal with large descriptions such explicit references tend to become cumbersome and error prone. The following section deals with the issues of how to improve the readability and writability of ISP descriptions by using abstractions like pseudo-registers, procedures, temporary registers, etc.

ABstractions and Implementation Dependencies

ISP can be viewed as a programming language for a specific class of algorithms, i.e., Instruction Set Processors or Architectures. Ideally, a language to describe architectures should avoid the specification of any implementation details. Any components introduced beyond these are unnecessary for the programmer of the machine and might even bias the implementor working from the description. While these items must appear in a description of an implementation, a problem arises when describing a family of machines where the abstractions and/ or algorithms may vary across members of the family. The rest of this section illustrates this problem.

* The CFA selection committee adopted the definition of architecture proposed by the designers of the IBM S/360. "The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and control, the logical design, and the physical implementation."** In order to keep the examples within the space limitations of this paper, we have taken some minor liberties with the syntax of ISPL. These alterations should not overly confuse readers familiar with ISPL.

* Even though some portions of the Architectures were left out of the ISP descriptions, notably the Floating-Point Instructions, the ISP descriptions used in this project are non-trivial computer programs. Each description takes between 30 and 40 pages of code. The size of the descriptions (1445 lines for the PDP-11, 2345 lines for the Interdata 8/32, and 2132 lines for the IBM S/370) reflects the size of the instruction set, not necessarily the complexity of the architecture.
An ISP description written using only the architectural components would not only be unreadable but also unwritable. Some form of abstraction is required. The following subsections demonstrate this point by introducing pseudo-registers, procedures, and temporary registers. These abstractions may or may not have a counterpart in some or all physical implementations of the ISP description.

**Pseudo-Registers**

When writing an ISP description for a real machine it immediately becomes apparent that describing everything
in terms of just the components of the architecture would lead to a cumbersome and unreadable description. The concept of a pseudo-register to rename a frequently used field of a register greatly relieves this problem. For example, consider the PDP-11 which has an autoincrement addressing mode. During the address computation an architecture register, pointed to by a subfield of the current instruction, must be incremented. Dealing only with components of the architecture would yield an expression like: \( R[M[Pc](2:0)]-R[M[Pc](2:0)]+2 \) where \( M[Pc] \) represents the current instruction in memory, pointed to by the program counter. Introducing the pseudo-register \( Ir \) (instruction register) for the current instruction would yield: \( R[Ir(2:0)]-R[Ir(2:0)]+2 \). We could further define a pseudo-register, \( Dr \) (for destination register), for the frequently used three bit subfield \( Ir(2:0) \), as in: \( R[Dr]-R[Dr]+2 \).

The pseudo-registers may suggest a register (e.g., \( Ir \)) or a set of wires (e.g., \( Dr \)) in some physical implementation. In reality they may have no physical correspondence at all. In any event, pseudo-registers are a useful and necessary abstraction for readable (and writable) ISP descriptions. However creating pseudo-registers for infrequently used fields or using obscure names may defeat the usefulness of this abstraction leading to reader confusion and excessive page flipping to find definitions.

Procedures

Just as there are frequently used register fields in a machine description, there are frequently used sequences of operations. Forming these operations into procedures greatly enhances readability.

For example, consider operand fetching. Every machine has a more or less complicated effective address calculation that is performed when accessing these operands. A memory reference to a destination operand might appear as: \( M[\text{Dest}] \) where \( \text{Dest} \) is a procedure for calculating the effective address of the destination operand. Without procedures the same reference for the PDP-11 would appear as shown in Figure 2. The situation would further be aggravated if the effective address had to be processed by some form of memory management which provides for address translation and rights checking. These operations would have to be performed in the description on top of the effective address calculation. It should be noted that many minicomputers and all larger computers have some form of memory management.

**Temporaries**

Occasionally readability is improved by introducing a temporary register in cases where the operands before and after the operation are required or a complex result is used repeatedly. Figure 3 shows a portion of the memory management procedures for the PDP-11.

The Read procedure shows the translation of a virtual address into a physical address. A temporary Memory Address Register (Mar) initially contains the virtual address (the result of the effective address calculation) which is then translated into a physical address in the line that reads:

\[ \text{Mar} \leftarrow \text{(PAR}[11:0]+\text{Mar}[12:6]) \] @ Mar[5:0] \]

The PAR (Page Address Register) and PDR (Page Data Register) arrays contain the necessary address translation information. A bounds check is performed before the actual memory fetch from physical memory. Without the temporary variable Mar the Read procedure would be substantially complicated by having to replace every appearance of the temporary by the complex expression given above. Of course, the temporary variable may or may not have a counterpart in some implementation.

---

**Figure 2**—Inline effective address calculation

\begin{verbatim}
M[ decode Dd =>
  (decode Idm =>
    *37400@Dr;
    R[Dr]+R[Dr]+2 next R[Dr]-2;
    R[Dr]+R[Dr]-2 next R[Dr];
    M[Pc+2] + R[Dr];
  )
  (decode Idm =>
    M[*37400@Dr];
    R[Dr]+R[Dr]+2 next M[R[Dr]-2];
    R[Dr]+R[Dr]-2 next M[R[Dr]]; M[M[Pc+2] + R[Dr]]
  )
]
\end{verbatim}

---

From the collection of the Computer History Museum (www.computerhistory.org)
Implementation dependencies

There are multiple examples of details that must be specified in an implementation description but do not belong in an architecture description. Typically, these are features that exhibit model dependencies. For instance, in the specification of the interrupt handling facility of a computer system, it could be the case that because of cost/performance requirements, different models must respond to simultaneous interrupts in different orders. An ISP description must by its very nature describe a specific order of interrupt trapping, thus losing a degree of freedom that one might wish to provide the machine implementors.

Figure 4 shows how the specific order in which simultaneous interrupts are fielded is build into an ISP description. Individual bits of INTVEC indicate the presence of a pending interrupt of a given priority. When only one interrupt is pending the proper context switching will take place. When more than one is pending there will be multiple context swaps and lower priority interrupts will be delayed to be processed later (the "new PSW" associated with a low priority interrupt will be stored into the "old PSW" position associated with a higher level interrupt).

It is not clear whether having to be specific about ordering of interrupts or similar events is a bad practice. Although one can claim that machine designers will be constrained in their choice of designs, the fact still remains that somebody must write the interrupt handling software, and for these programmers the order of interrupt fielding is important. This type of dilemma occurs quite often when dealing with ISP descriptions. The solution might be simply to write model-dependent ISP procedures whenever this conflict arises and then indicate in the ISP description which version of a given procedure must be implemented for a given model.

Another problem with implementation dependencies is that the definition of the input/output behavior of an instruction might actually imply a particular implementation. For example, consider the PDP-11 Subtract instruction. The carry condition code (C) is set according to the borrow during the subtraction. The PDP-11 Processor Handbook...
In the adder case, the carry is the complement of the borrow which is exactly the definition given by the PDP-11 Processor Handbook. The ISP description of the setting of C becomes:

\[
C\leftarrow (\text{dest} - \text{source})(16); \quad \text{! Subtraction}
\]

\[
C\leftarrow \text{NOT} (\text{dest} + \text{NOT(source)} + 1)(16); \quad \text{! Addition}
\]

As in the previous example (the order of interrupt handling), a complete algorithm had to be given. In this case, the subtractor/borrow algorithm is preferred since it presupposes only the properties of the two’s complement number system. However, if an alternate implementation (such as forming the two’s complement and adding) is utilized, then the implementor should be aware of possible changes in other algorithms in the ISP description.

THE ARCHITECTURE RESEARCH FACILITY

The facility used for the data collection phase of the CFA project is depicted in Figure 6. Reference 4 explains in full detail the features of the ISP compiler and simulator. Some familiarity with their capabilities is needed in order to

---

5 - 3 = 2 (no borrow) 3 - 5 = -2 (borrow)

\[
\begin{array}{c|c}
0101 & 0011 \\
0011 & 0011 \\
--- & ---
\end{array}
\]

Subtracting

\[
\begin{array}{c|c}
0 0010 & 1 1110 \\
\text{borrow} & \text{borrow}
\end{array}
\]

Adding Two's Complement

\[
\begin{array}{c|c}
0101 & 0011 \\
\text{CARRY} & \text{CARRY}
\end{array}
\]

---

---

1 0010 0 1110

carry carry

Figure 5—Implementation dependant condition code setting

describes the setting of the C bit as:

"C condition code is cleared if there was a carry from the most significant bit of the result, set otherwise."

This definition implicitly assumes that subtraction is implemented by forming the two's complement and adding. Figure 5 illustrates the situation. Consider four-bit numbers and the two methods to perform subtractions, by using a subtractor, and by using an adder after forming the two's complement.

---

From the collection of the Computer History Museum (www.computerhistory.org)
understand the data collection phase described later. The following paragraphs attempt to satisfy this need.

The ISP compiler produces code for a hypothetical machine, dubbed the Register Transfer Machine (RTM). The “object code” produced by the compiler can be linked together with a program which is capable of interpreting RTM instructions. This separation between the ISP description, the RTM code, and the RTM interpreter allows the simulation of arbitrary, user defined architectures. The result of linking the RTM code with the RTM interpreter is a running program, a simulator.

The simulator accepts commands from a teletype or user designated command file. The state of the simulator can be dumped to a command file which can be read at a future date when the simulation is continued. Command files can also be used to load programs and data into the simulated target machine memory and registers.

**Debugging of test programs and ISP descriptions**

Most of the test programs were debugged and run on the real machines. Other programs were executed exclusively under the simulator. The latter included those programs using privileged instructions that were not directly available to non-system programmers (e.g., interrupt and I/O handlers). Results from the actual runs, whenever available, were used to check the simulated execution.

Only minor modifications and corrections to the ISP descriptions were performed during the data collection phase. The largest unforeseen problem with the test programs was presented by the memory management feature of the PDP-11 which was based on the PDP-11/40. The test programs which made use of this feature had been tested on a PDP-11/45 which uses different Unibus addresses for the memory management registers. This difference required minor modifications in the test programs. Most other problems were of a simpler nature and required only a few minutes to correct. It should be noted here that the simulator facility was also used to debug some programs for the Interdata 8/32 before they were executed on the real machine. This was dictated by the fact that no 8/32 was available near CMU and a large turn-around time (several days) would have complicated the debugging of the test programs.

**Preparation of simulation tests**

The ISP simulator provides commands for the loading and initialization of the simulated machine memory and internal registers. The single most important feature of the command language which permitted the fast execution and collection of statistics was the ability to read command files containing the test programs to be executed. The command language cannot handle programs in symbolic form (assembly language); it requires the preassembly of the programs into absolute, numeric, code. To get around this problem, a set of utilities was developed at CMU which permitted the transformation of assembly listings prepared by the real machine’s assembler into simulation command files. This operation was performed off-line as shown in Figure 6.

Figures 7 and 8 show the transcript of a typical session using the ISP simulator. The session consists of running one of the test programs (Bit Test, Set, and Reset) on the PDP-11. The input for a simulation session consists of several files prepared off-line. These files include: the test program (derived from the assembly listing), a driver (simulation commands used to initialize the parameters for the test program), and finally, a command file with a list of those ISP procedures which must be “opaque” (these are the procedures during which the activity counters are disabled). A typical command file, derived from an assembler listing is shown in Figure 9. This was the test program used in the sample simulator session shown in Figures 7 and 8.

**Instrumentation**

The ISP simulator permits the instrumentation of an ISP description by associating activity counters with each of the machine registers and memories. These counters allow the collection of statistics indicating the number of times each component of the machine is read from or written into. A separate counter is kept for each label in the ISP description. Labels are included in the ISP descriptions to identify machine instructions, addressing modes, loops (used to describe vector-like instructions like move character on the S/370), as well as other ISP procedures. During the execution of the test programs, a data base was created by collecting dumps of the counters after each test case was completed. The files containing the counters were then processed by other, off-line, programs in order to arrive at the M and R measures described in a later section.

**Artificial labels in the ISP descriptions**

Certain modifications not normally needed were made to the ISP descriptions to aid in the collection of data during the running of the test programs for the CFA project. Several labels and “do-nothing” procedures were added to identify certain phases in the instruction interpretation algorithm and to measure selected events (e.g., different addressing modes). The labels added to count these events are clearly not part of the architecture or even the implementation.

Figure 10 shows an example extracted from the S/370 ISP Description. It shows the use of artificial labels to identify different addressing modes for the RX instruction set. According to the definition of the S/360 and S/370 architectures, the RX instructions can specify both a base and an index register to be added together with the displacement field of the instruction to compute the address of the memory operand. The architecture further specifies that R[0], when specified as either a base or index register does not take place in the effective address calculation, i.e.,
ru pdp11m
ISP SIMULATOR V3 - NAD RAF STAGE 2
Friday 10 Sep 76 17:13:58 PDPI1M.ISP(L40MB25)
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>read fad.lisim ! Read in the benchmark file
»RONIX OCTOL
»DECHO
! The benchmark file disables the listing
! on the user terminal.

>>100 LINES READ
>read fa.drs ! Read in the driver file
>>! HERE COMES THE DRIVER {CALLS}
>>SETVAL MH(30001)+013746 005202 ! MOV @#5202,-(SP) ; F
>>SETVAL MH(30002)+013746 005204 ! MOV @#5204,-(SP) ; N
>>SETVAL MH(30003)+013746 004000 ! MOV @#4000,-(SP) ; R1
>>SETVAL MH(30004)+013746 005200 ! MOV @#5200,-(SP) ; RC
>>SETVAL MH(30010)+013746 005202 ! MOV @#5202,-(SP) ; W
>>SETVAL MH(3012)+060473 001800 ! JSR PC,#1800 ; BTSR
>>SETVAL MH(3014)+080000 ! HLT

>> The above sequence of PDP-11 instructions pushes the parameters
! onto the stack, calls the benchmark as a routine, and halt.

>>SETVAL MH(20000)+123457 071234 167006 145670 ! BIT STRING
>>SETVAL MH(25000)+0 ! RETURN CODE
>>SETVAL MH(2501)+2 ! F
>>SETVAL MH(2502)+25 ! N
>>SETVAL MH(2503)+0 ! WORK AREA
>>SETVAL PC-6000
>>SETVAL SP-200

! The above sequence initializes the data (parameters), the stack
! pointer and the program counter (which now points to the code
! sequence that push the parameters and call the routine.

>>SETVAL R<0 ! This is an ISP internal variable - indicates whether the
! machine is running, halted, or waiting.
>>SETCTR ALL 0,0 ! Reset activity counters
>>READ DPD11.SIM(L40MB25) ! PDP11 Opaqued Procedures
>>DECHO
>>53 LINES READ
>>READ UUD11.SIM(L40MB25) ! UNIMPLEMENTED OPERATION BREAKS
>>DECHO
>>15 LINES READ
>>TRACE IR,PC,R,MHIO ! Trace a few selected registers
! IR is the Instruction Register,
! PC is the Program Counter (R7),
! R[0:7] are the general registers,
! MHIO is the I/O page (R is mapped onto MHIO)

>>BREAK: JSR, RTS
>>26 LINES READ

Figure 7—Initialization of a simulation run

R[0] should be specified whenever one of these two components (base or index) is missing. In the above example
four dummy in-line procedures where introduced to count the number of times each possible combination of base/
index modes occurs. Thus RX0000 is "executed" whenever R[0] is specified as both the base and the index register.
RX00X2 is "executed" whenever R[0] is used as the base register and any of R[1:15] is used as the Index
register. RXB1X2 is "executed" whenever any R[0] is specified as the index register and any of R[1:15] is specified as
the base register. Finally, RXB1X2 is "executed" whenever R[0] is not specified as either the base or index registers.
NOP is a dummy procedure which does not have any side effects.
ARCHITECTURE PARAMETERS

As a means of comparing architectures, three measures were defined for the CFA project:

Measure of Space
S The number of bytes used to represent a test program.

Measure of Execution Time
M The number of bytes transferred between primary memory and the processor during the execution of the test programs.
R The number of bytes transferred among internal registers of the processor during execution of the test program.

Measures of Execution Time

>start inter
@ INTER + 15 IR = 13746
@ INTER + 20 PC = 6002
@ SINCD + 22 R (7) = 6004
@ DDECRO + 21 R (6) = 176
@ INTER + 15 IR = 13746

... ... ... ! Pushing Parameters

@ INTER + 15 IR = 12746
@ INTER + 20 PC = 6022
@ SINCD + 22 R (7) = 6024
@ DDECRO + 21 R (6) = 166
@ INTER + 15 IR = 4737
@ INTER + 20 PC = 6026
BREAK: AFTER JSR
setctr all 0,0
! The simulation stops on a breakpoint
! The real benchmark starts here; we must
! reset all counters (they were modified
! during the benchmark calling sequence)
! we continue the simulation

>cont
@ DINCD + 22 R (7) = 6030
@ JSR + 14 R (7) = 6030
@ JSR + 15 PC = 1000
@ INTER + 15 IR = 10046
@ INTER + 20 PC = 1002

... ... ... ! Program Execution Trace

@ INTER + 20 PC = 1072
@ SINCD + 22 R (7) = 1034
@ WRITE + 131 NWID (374000) = 0
@ INTER + 15 IR = 287
@ INTER + 20 PC = 1074
BREAK: AFTER RTS
! the simulation stops at the end of the
! benchmark (the return instruction)

>outctr facl. rm3
! we dump all the counters into a file
! we continue the simulation
>cont
@ RTS + 2 PC = 1074
@ RTS + 7 R (7) = 6030
@ INTER + 15 IR = 0
@ INTER + 20 PC = 6032
SIMULATION COMPLETED
! we executed the Halt instruction
RUN TIME (in usec units) = 831678
RTM OPS EXECUTED = 4535
>exit
EXIT
! we finish the session
EXIT

Figure 8—Program execution trace

From the collection of the Computer History Museum (www.computerhistory.org)
ProgrAnM, ProgranMor Idontification (Suprassed)

13 01300 ; Offsets of parameters frOIl stack p
14 01400 ;
15 00000; SAVE=4 ; we need to save 2
16 01600 ;
17 00016 01780 F=12+SAVE ; function code
18 00014 01880 N=18+SAVE ; relative bit nums
19 00012 01980 N=6+SAVE ; address of bit str
20 00010 02880 R=6+SAVE ; address of return
21 00006 02180 WOD=2+SAVE ; address of work ar
22 02280 ;
23 00000' 02380 BTSR;
24 00000' 01046 02480 MOV R9,-(SP)
25 00002' 01046 02580 MOV R1,-(SP)
26 00004' 005076 000010 02680 CLR eRC(SP) ; ze
27 00010' 016600 000014 02780 MOV N(SP),R9 ge

Relocatable Object Code Listing

41 000066' 012601 04180 QUIT: MOV (SP)+,R1 ; ex
42 000070' 012600 04280 MOV (SP)+,R0
43 000072' 000207 04380 RTS PC
44 000074' 150110 04480 SET: BISB R1,eR0 ; FC
45 000076' 000773 04580 BR QUIT
46 000001 04680 .END

Cross-Reference Listing

\begin{verbatim}
! Here begin the simulation commands
! derived from the above listing
! relocation address = word 400 (octal) = byte 1000
\end{verbatim}

Target Machine Program Loading

\begin{verbatim}
SETVAL MN[4400]-010046
SETVAL MN[4401]-010146
SETVAL MN[4402]-005076 000010
SETVAL MN[4404]-016600 000014
\end{verbatim}

Echo

Figure 9-A command file derived from an assembly listing

The S measure is a static parameter which can be computed independently of the ISP description. For the purposes of this paper we will restrict the discussion to the other two measures. The actual computation of the M and R measures was done through a semiautomatic process.

The raw data collected from the simulator was used to count frequencies of instructions and addressing modes. These counters were multiplied by certain hand calculated factors in order to arrive at the M and R measures for each test program. Ideally, the ISP simulator should perform the
entire operation and this would be a better approach, less subject to human errors. We had to use the hand computed factors due to our inability to determine the influence of the ISP writing style on the architecture parameters as defined above. The actual results of the experiments (the M and R measures for each test program) are presented in Reference 9.

The exact methodology for writing ISP descriptions so that the M and R measures can be calculated automatically has yet to be developed. It is clear, however, that a careful control of the counting mechanism is paramount to the collection of meaningful data. During the data collection phase we made use of the following techniques towards this goal.

**Opaqued procedures**

A simulator command allows the selective masking of in-line and off-line procedures. Masking or opaquing a procedure inhibits all activity counts inside the body of the procedure.

Certain operations, such as incrementing the program counter after an instruction, or the setting of the condition codes as a result of an instruction, do not affect the R measure and should not be counted. This is typical of those actions which, in a reasonable implementation, can be done using ad hoc circuitry, separate from the main operational units of the machine. These operations could be implemented by combinational logic (e.g., setting condition codes from ALU lines), special registers (e.g., using a counter instead of a simple register for the program counter), or even complex sequential networks (e.g., the virtual address translation can be performed using its own arithmetic units and data paths).

Operations like those described above can be easily marked by adding artificial labels to the ISP description and then disabling the counters while the selected operation is being performed.

**Pseudo-Register chains**

Every component declared in an ISP description has activity counters associated with it. When a register is defined in terms of another register, such as: \( \text{Pc}(15:0) := \text{R}[7](15:0); \) a redefinition chain is established. Accesses higher up in the chain increment all counters lower in the chain but not vice versa. In the above example an access of the \( \text{Pc} \) causes the register file counter for \( \text{R} \) to be incremented but accessing \( \text{R}[7] \) does not increment the program counter (\( \text{Pc} \)). By establishing appropriate redefinition chains, distinction between access types can be maintained. One variation of this technique is the use of "shadow" registers. For example two instruction registers can be defined: \( \text{Ir}(15:0) := \text{Irl}(15:0); \) where \( \text{Irl} \) is the shadow register. The loading of the \( \text{Ir} \) from memory is to be counted in the R measure, however, the combinational logic decoding of the instruction and effective addressing mode is not to be counted. The former is performed on \( \text{Ir} \), the latter on \( \text{Irl} \) thus distinguishing the two different types of accesses.

**Memory access procedures**

Modern machines provide the user with an address space defined in terms of small units of information, typically 8-bit bytes. For convenience, however, the architectures also define larger access units in multiples of bytes. Thus, the IBM S/370 provides bytes, half-words, full-words, and double-words. Since the physical memory is the same, the ISP description must declare the different address spaces by building a redefinition chain in which the different address spaces are declared as "pseudo-memories" so that the M measure component of each address space is properly accounted for.

Machines like the PDP-11 add some more complexity to the issue of having multiple address spaces. The PDP-11 architecture defines the concept of an I/O page as a
reserved portion of the address space, not necessarily implemented as a physical memory. Addresses in the upper 4K bytes of the PDP-11 are used to address I/O devices, machine registers, etc. Addresses in the I/O page must be handled differently when computing the M measure. If one attempts to include in-line address checks in the ISP description, the description quickly becomes bulky and unreadable. A satisfactory solution is simply to define memory access procedures (Read and Write), which can then be properly instrumented, thus enabling the automatic computation of the M measure.

Temporary Registers

The automatic computation of the R measure is more difficult. In an ISP description there are three types of registers to consider: architectural, standard implementation, and temporaries. Architectural registers and certain standard implementation registers (instruction register, memory address register, and memory buffer register) can be handled using the same techniques used to automate the M measure (declaration chains and encapsulating procedures). Handling temporary registers presents a more difficult problem. The number, type, and manipulation of temporary registers are a matter of writing style.

Architecture parameters which are based solely on architecture registers while ignoring temporary registers introduced for clarity might overlook hidden computations performed on these registers. Unlike the memory, architectural registers, and standard implementation registers, a tightly defined writing style cannot be developed for temporary registers. One solution would be to use well-known expression optimization techniques on the ISP description to uniformly minimize the temporary register activity. Hopefully the optimization would lead to similar results for equivalent algorithms.

Architectural parameters should be independent of the experience, style, and objectives of the ISP writer. This will then guarantee that the ISP descriptions which make use of abstractions (pseudo-registers, procedures, and temporary registers, etc.) to enhance clarity and readability will not be penalized. By the same token, no advantage should be derived from the use of "clever" programming tricks which might attempt to bias the measurements.

ADVANTAGES OF AN ARCHITECTURAL RESEARCH FACILITY

Although for the purposes of this paper we have presented the uses of the ISPL compiler and simulator in the context of a specific project, we should point out the wider range of applications in which a system like ARF can be of great value.

A simulator as a training tool

In this paper we described how machine language test programs can be executed under the simulator. The implied assumption during the data collection phase was that we were dealing with correct, finished programs. With no extra effort the ISP simulator can be a powerful training device for novice programmers. Speed of simulation is not an issue in this application. Programmers learning a new machine language tend to spend long hours single-stepping via the machine console. An interactive simulator can easily satisfy the needs of these users, while providing much better diagnostic and debugging facilities than a computer console (did you ever see a "help" button on a machine?). ISP descriptions exist for the following machines: DEC PDP-8, PDP-10, PDP-11, IBM S/370, Interdata 8/32, and Intel 8080.

Architecture evaluation

The S, M, and R measures are by no means the only set of architecture parameters one might wish to evaluate. Nothing in the ISP simulator depends upon this particular set of parameters. The instrumentation in the simulator allows counting every event we care to define by simply labelling the event. There is no need to create new procedures which might impact the organization or readability of the description; even a single register transfer operation can be labelled and counted. More details on architecture evaluation can be found in Reference 5.

Experimentation

Once the initial effort of writing an ISP description is accomplished, only moderate effort is required to perturb it to reflect proposed or actual changes in the architecture. Thus the effect of a modification in an architecture can be measured and studied before any funds are committed to the development of a new machine. By a careful design of the ISP description it is possible to pattern a description along the lines of the organization of the physical machine. Thus one would be able to measure and evaluate different models of the architecture. For instance, functional units and data paths can be represented by separate procedures in the ISP description. An ISP description could then be parameterized to invoke these procedures in different order, concurrently or sequentially, with or without intermediate steps, etc. as the different models differ in their implementation. An example might be determining the effect of a cache memory on the apparent instruction execution speed in high performance implementations.
Machine relative software

As the number of different architectures coming into existence increases every year, it is becoming more and more expensive to develop the necessary software support base that allows the effective use of these machines. The availability of user micro-programmable machines enlarges the space of possible architectures to the point that automatic software generation systems will become a necessity. Tools that operate relative to a computer description could represent a significant breakthrough in the manner that computer systems (hardware/software) are designed and evaluated. The Advanced Research Projects Agency (ARPA) of the Department of Defense is currently sponsoring this area of research at CMU and elsewhere.

In the future one can foresee hardware and software automation systems that take as input computer descriptions, and language and problem specifications; and from these, generate operating systems, compilers, and other support and application software automatically. Other areas of current research include automatic diagnostic generation, microcode generation, machine verification, etc.

Formal computer descriptions will play an increasing and important role in the evaluation, procurement, verification, and programming of computers. The ARF facility is a step in this direction.

REFERENCES
