ON THE VERIFICATION OF COMPUTER ARCHITECTURES USING AN ARCHITECTURE DESCRIPTION LANGUAGE

Subrata Dasgupta

Department of Computer Science
University of Southwestern Louisiana
Lafayette, Louisiana

Abstract

In a previous paper [8], we had presented the notion of a family of languages for the multilevel design and description of computer architectures. Details of a particular language family, currently under development, was also described. One of the constituent members of this family is S^*_A, intended for the specifications of the outer (or exo-) and inner (or endo-) architectures of general purpose von Neumann style computers. In this paper we describe the formalization and application of S^*_A to the formal proofs of correctness of architecture designs.

1. Towards Formal Architecture Design

The design of computer architectures has traditionally suffered from two major drawbacks: firstly, the idea of storing the design in some formal representational form has remained largely outside the mainstream of architectural thought and practice. Secondly, in the absence of a rigorous, theoretical framework, architectural designs have conventionally been evaluated - both with respect to its correctness and its performance - only after they have been implemented as physical systems.

These are major shortcomings of any field which lays some claim to being called a discipline. Their consequence is a compendium of undesirable characteristics that are encountered at one time or another by all involved with computer architecture - whether as designer, teacher, or theorist. In particular, we observe that:

(a) The documentation of the design is usually a combination of (informal) block diagrams and prose descriptions. The design is, as a result, ill-defined both syntactically (i.e., in respect of its form) and semantically (in respect of its function and meaning).

(b) It is extremely difficult, even in principle, to verify the correctness of the design without constructing and testing the physical system.

(c) It is equally difficult to manipulate or alter the design and study the effects of alternative design choices without actually implementing the design in the form of a physical system.

Design theorists, notably Jones [15] have pointed out that one of the characteristics of the craft stage of design is the lack of a symbolic medium in which to capture the shape of the product, and the consequent inability for one to experiment with the design (in contrast to experimenting with the product itself).

The evolutionary design stage that has historically followed the craft stage is termed by Jones, design-by-drawing, chiefly characterized by the replacement of the product itself by its symbolic representation as the medium of experiment and change. In this sense, computer architecture appears to be basically at the more primitive craft stage of design evolution.

Clearly, an essential requirement for a transition to the design-by-drawing stage to take place is the availability of one or more formal pictures of the design. In the realm of computer hardware, the need for such formal descriptions has long been recognized [3, 9, 11]. In the specific domain of computer architecture, the pioneering work was the ISP notation of Bell and Newell [6] which later matured into the ISPS language [5]. Other, more recent efforts include SLIDE [19] and ADL [16].

Recently, we have proposed yet another architecture description language called S^*_A [7,8]. For our present purposes the following fundamental and distinguishing aspects of this language must be noted.

(1) S^*_A is a general purpose, high level, procedural language for the formal specification of computer architectures and was designed as a member of a family of languages called the S* family that could be applied to the multilevel design of computer architectures. The notion of a language family is predicated on the observation that architecture encompasses several levels of abstraction of the physical machine [8]. The design process for such a system would then involve a succession of stages, at each of which the system would be represented at a particular abstraction level, in the medium of the language most suited to that level.

(2) A major goal in designing S^*_A was to be able to use it in the formal verification of architecture designs. Thus, special attention was paid towards defining primitive and structured data types and control structures for which axioms and inference rules could be constructed. In fact, the semantics of each major entity in S^*_A is essentially twofold: an "informal" definition which establishes the pragmatic interpretation of the entity (viz., what kind of hardware interpretation should be attached to that entity) and a "formal" definition that can be used for purposes of design verification.

The use of the S* language family in the systematic design of a multilevel architecture, called QM-C, was described in [8]. In this paper, we focus specifically on the problem of verifying computer architecture designs described in S^*_A.

2. 'Defining' Computer Architecture

At this point, we need to clarify what we understand by the term "computer architecture". Various prescriptions have been made as to what this term means or should mean [2,12,17]. The following characterization offers no startling new insight on this matter; we merely attempt to unify these different definitions in the context of the social practice of architectural design and research. In fact simply based on our observations of the latter we may identify the following general characteristics:

(a) An architecture is an abstraction of the hardware in that it is concerned with the structure and behavior of hardware represented as an abstract information processing system (rather than as an ensemble of physico-electronic devices).

(b) Architectural attributes include both the external (i.e., functional) appearance of the computer as well as its internal form.

The computer architect is, thus, a designer of information processing systems of a particular kind: those that are directly realized by a combination of hardware and microcode. The second point above suggests, in addition, the notion of architectural levels. We may give more complete shape to this notion as follows:

Exo-architecture: A computer's exo-architecture refers to the logical structure and functional capabilities of the hardware system as visible to the machine programmer or compiler writer.

Endo-architecture: A computer's endo-architecture consists of a specification of the functional capabilities of its physical components, the logical structure of their interconnections, the nature of the information flow between components, and the means whereby this flow of information is controlled.

To understand further, the relationship between exo- and endo- architectures we note that a major function of exo-architecture is to hide certain kinds of information concerning the computer's design. These include, for example, whether or not the instruction cycle is pipelined, the presence or absence of a cache memory, whether memory interleaving is used, whether instruction interpretation is done in firmware or by hardware, and so on. These are all, typical endo-architectural features. Thus, exo-architecture is an abstraction of endo-architecture. Conversely, the latter may be viewed as what is revealed of the machine's internal logical structure and behavior when we refine the exo-architectural description.

For our present purposes we will be concerned with the verification of both exo- and endo-architectures.

3. On The Criterion of Correctness

The basic notion we adopt for verifying architecture designs is the Floyd-Hoare inductive-assertion method [13]. Let P be an assertion specifying the states (or relations between states) of some set of architectural data objects,and S be a description (in S^*_A) of an architectural system such that P is assumed to hold when the system is activated. Let Q be another assertion about the states of the data objects; following programming terminology, P,Q will be called the precondition and postcondition respectively of S. Then S will be deemed (partially) correct with respect to P and Q if it can be shown that given P, the activation of S leads to the postcondition Q when S terminates. Notationally this is expressed by the formula

$$\{P\}S\{Q\} \qquad (1)$$

To prove total correctness requires us also to show that S terminates.

It is important to note, however, that cases may arise where S never terminates; for example, an instruction fetch/decode/execute cycle. The S*A loop statement forever do..od allows such activities to be depicted. In such cases, what can be shown is that every time the body of the loop is entered, the precondition P holds, and upon executing the body, Q is satisfied.

The proof of formulas of the above type requires the use of formal axioms and rules of inference (or proof rules). Thus, the heart of this approach to the domain of architecture verification is the construction of proof rules and axioms for S^*_A (section 4).

The closest work in this context is that of Patterson on microprogram verification [20]. We share the common objective of showing that program correctness techniques can be successfully adapted to the domain of computer architecture. However the work reported here differs from Patterson's in the following respects:

(1) Patterson was concerned with the verification of microprograms written in the high level language STRUM. STRUM was primarily oriented towards a particular microprogrammable computer, viz., the Burroughs D machine; consequently, the only data objects defined in STRUM were those available on this machine. In contrast S^*_A contains a general collection of data types and data structuring facilities by which data objects for an arbitrary architecture can be represented. The definition of S^*_A thus, includes an axiomatic characterization of these data types (Section 4) and these play a critical role in architecture verification.

(2) An important application of S^*_A is the description and verification of asynchronous concurrent architecures. This is an issue that would not usually arise in microcode verification, hence was ignored in the STRUM effort.

4. Axiomatization of S^*_A

S^*_A is similar in many respects to high level programming languages. The basic difference lies in the data types and data structuring capabilities, the constructs for modularizing S^*_A descriptions, and in the pragmatic interpretation of constructs in the language.

Most of the "features" in S^*_A have been previously described [7,8] hence we will not repeat the discussion here. However, as noted above, the key to applying the Floyd-Hoare technique is the axiomatization of the description language.

An axiomatic definition of S^*_A has been completed [10]. In this section we take a very small subset of this definition to show the nature of these axioms and proof rules. Rather than discuss constructs that S^*_A shares with other languages we shall focus on those that are rather specific to S^*_A.

4.1 Data Types

A data types in S^*_A conforms to the concept of type as suggested by Hoare [14]. That is: (a) a data type determines a particular class of values which may be assumed by an instance (a variable, a constant or expression) of that type; and (b) associated with each type is a set of primitive operations which can be applied to these values.

S^*_A contains the primitive data type bit, and the structured type seq, array, tuple, assoc array, and stack. The following paragraphs shows typical properties of some of these types.

4.1.1 (a) The data type bit consists of the values {0,1}. Operations defined on type bit consist of the logical operators $\{\wedge, \vee, , \circ, \wedge, \overline{\vee}\}$ and the arithmetic operators {+,-,*,/} which carry with them the usual meaning.
(b) Given x,y of type bit, x/y is undefined for y=0.
(c) Given x of type bit, x represents a binary storage element.

4.1.2 Let T denote a sequence type: seq $[i_n..i_\ell]$ bit.
Then (a) $i_n \geq i_\ell$, where i_n, i_ℓ are non-negative integers

(b) Let $i_n \geq i \geq i_\ell$. Then $T : i_n..i_\ell \rightarrow$ bit. That is, T(i) is an element of type bit

(c) Let $<bb..b>^{i_n}_{i_\ell}$ denote a binary string of length $i_n - i_\ell + 1$. Then $T = \{<00..0>^{i_n}_{i_\ell}, <00..1>^{i_n}_{i_\ell}, \ldots, <11..1>^{i_n}_{i_\ell}\}$

(d) The logical operations $\{\wedge, \vee, , \circ, \wedge, \vee\}$ are defined on T according to the following rule: let Θ denote a logical operation and x,y be of type T such that $x=<x_{i_n}\ x_{i_n-1}..x_{i_\ell}>$ and $y=<y_{i_n}\ y_{i_n-1}..y_{i_\ell}>$. Then

$x \Theta y = x_{i_k} \Theta y_{i_k}$ for all i_k such that $i_n \geq i_k \geq i_\ell$.

(e) The arithmetic operations +,-,*,/ are defined on T as follows: let +´, -´, *´, /´ denote ordinary arithmetic operations (/´denotes integer quotient division). Let x,y be of type T and max be the maximum value $<11..1>^{i_n}_{i_\ell}$ for type T. Then +, -, *, / on x,y are defined by:
(i) $x +' y \leq max \supset x+y = x +' y$
(ii) $x \geq y \supset x-y = x -' y$
(iii) $x *' y \leq max \supset x*y = x *' y$
(iv) $y \neq <00..0>^{i_n}_{i_\ell} \supset x/y = x /' y$

Note that the arithmetics are defined only for unsigned integers. Furthermore, no side effects result from these operations. In actual fact, a given architecture may not only admit arithmetic operations of various types (e.g., 2's complement) but they may also result in side effects (such as an 'overflow' flag being set). In S*A such arithmetic operators would be constructed as procedures from the predefined operators. The semantics of the newly defined operators (including side effects) would then be inferred using the basic S*A axioms and rules of inference.
(f) The arithmetic, logical, and shift operations defined on type T represent primitive (hardware defined) functional logic units. (Note that the axioms governing the shift operations are not shown here).
(g) Let x be of type T. Then x represents any device capable of storing binary strings in the range of values defined by T such that elements of the binary string can be accessed in parallel.

4.1.3 Let T denote either of the stack types: stack [i] of $T_\circ$ or stack [i] of $T_\circ$ with $V_1, \ldots, V_n$, where $V_1 \ldots, V_n$ are explicitly declared stack pointers.
(a) $V_1, V_2, \ldots, V_n$ are of type seq.

(b) Let intval (Vj) denote possible (decimal) integer values of $Vj \epsilon \{V_1, V_2, .., V_n\}$. Then intval $(Vj) \leq i$

(c) $T_\circ$ is of type seq or tuple

(d) The standard procedures push and pop are defined on the second stack type T as follows: let x be of type T and $X_\circ$ be of type $T_\circ$. Let $Vj \in \{V_1, \ldots, V_n\}$ be a stack pointer. Finally, let length (x) denote the number of elements of type $T_\circ$ in X.
Then:
(i) $\{\text{intval}\ (Vj) = Vj^\circ \wedge \text{length}\ (x) = \ell_\circ : \ell_\circ < i \wedge x_\circ = x_\circ^\circ\}$

$$\text{push}\ (x[Vj], x_\circ)$$

$\{x[Vj] = x_\circ = x_\circ^\circ \wedge \text{intval}\ (Vj) = Vj^\circ + 1 \wedge \text{length}\ (x) = \ell_\circ + 1\}$

(ii) $\{\text{intval}\ (Vj) = Vj^\circ \wedge \text{length}\ (x) = \ell_\circ : \ell_\circ \geq 1 \wedge x[Vj] = x^\circ\}$

$$\text{pop}\ (x[Vj], x_\circ)$$

$\{x_\circ = x^\circ \wedge \text{intval}\ (Vj) = Vj^\circ - 1 \wedge \text{length}\ (x) = \ell_\circ - 1\}$

While a type declaration "type T=T' "introduces a class of possible objects of type T that satisfies the properties of type T', the declaration of a data object of a given type T denotes the existence, in the architecture being designed or described, of a storage device whose abstract properties are prescribed by the properties of the data type T. Each distinct data object declared is a specification of a distinct storage device.

4.2 Executional Statements

Executional statements in S^*_A are basically simple or structured. Simple statements signify indivisible units of action and their meanings are usually defined formally by axioms. Structured statements are composed of one or more elementary statements and their meanings are formally specified by rules of inference.

Simple statements in S^*_A include the assignment, the procedure call and act (activate) statements, trap, await and sig, the procedure exit and return statements, and the goto. The axiom of assignment is as defined for Pascal [1] while the goto is defined according to the proof rule due to Alagic and Arbib [1]. As specific examples of the definition of simple statements, consider the await and sig constructs.

The basic synchronization facilities in S^*_A are provided by means of the standard procedures await and sig defined on synchronizing variables. A declaration of a synchronizing variable (or "synchronizer") is of the form

sync x : T

where T is of types bit or seq. The statements await x and sig x are defined on a synchronizer according to the following rules:

(i) $\{x = x_o > 0\}$ await x $\{x = x_o - 1 \geq 0\}$

$\{x = x_o = 0\}$ await x $\{false\}$

(ii) $\{x = x_o : 0 \leq x_o < max\}$ sig x $\{x = x_o + 1 > 0\}$

where max is the maximum integer valued state for x. Intuitively, an await operation will never terminate as long as the value of x is 0; when x>0 the await decrements the value of x and terminates. The sig operation is only defined if x < max. In that case it simply increments x by 1.

The sig and await constructs are used in S*A to establish synchronization between concurrently executing mechanisms (e.g., the components of an instruction pipeline). Thus, the above proof rules can be used for proving the correctness of concurrent systems, using the Owicki-Gries approach [18].

In general, there are three classes of structured statements. Of these, compound statements allow the sequential or parallel composition of other, simpler statements while conditionals allow actions to be taken based on specified tests of the machine state; the third group are the repetition statements. We give below, two examples of the proof rules for structured statements.

(a) The parallel statement $S_1 \square S_2$ specifies the simultaneous execution of S_1 and S_2. The next statement in sequence begins execution only when both S_1 and S_2 have terminated. Formally, provided S_1 and S_2 are dynamically disjoint:

$$\frac{\{P_1\}\ S_1\ \{Q_1\}\ ,\ \{P_2\}\ S_2\ \{Q_2\}}{\{P_1 \wedge P_2\}\ S_1 \square S_2\ \{Q_1 \wedge Q_2\}}$$

Two statements S_1, S_2 are said to be dynamically disjoint if during the period that S_1, S_2 are both in execution, their data resource sets are disjoint. Note that the above proof rule is expressed in the usual notation

$$\frac{H_1\ ,\ H_2\ ,\ldots,\ H_n}{H}$$

which states that whenever the assertions H_1, H_2, ...,H_n are true then H is also true.

(b) Structured await statements are of two forms: the first of these is "await x do S od" where x is a synchronizer; the statement S will execute if and only if x≥1; the second form is "await B do S od" where B is a boolean expression. In this case, B is continuously evaluated until it is true, at which point S begins execution. The proof rules for these two statements are:

$$\frac{\{Q\}\ S\ \{R\}\ ,\ \{P\}\ \text{await } x\ \{Q\}}{\{P\}\ \text{await } x \text{ do } S \text{ od } \{R\}}$$

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \text{await } B \text{ do } S \text{ od } \{Q\}}$$

5. On The Notation For Logical Formulas

As previously noted, assertions are stated in the form of formulas in the first order predicate logic. Such assertions will refer to the various data objects which, we have seen, are declared as instances of S^*_A types. In developing assertions we shall find it convenient (in order to make them more understandable) to "tag" a data object identifier with its type. For example, referring to Fig. 1 (see next section), the data object reg. sp will be designated within assertions as "reg. sp: ls_ register".

We also noted in [8] that through the use of synonyms, an object of a given type with a given identifier may be alternatively viewed as a data object with a different name. In specifying assertions involving a data object of some given type T with alternative names X,Y,..,Z, we shall also, where necessary, use the notation "X|Y|...|Z:T" meaning "X or Y or ... or Z of type T".

Finally, in order to simplify assertions and enhance their readability, auxiliary variables will be introduced where necessary [18]. Auxiliary variables may appear in the assertions but not in the S^*_A text. These will be denoted by subscripted identifiers, e.g., X_o, Y_1, etc.

6. An Example

Consider the specification of the QM-C machine CALL instruction as previously described in [8]. Its overall objective is to save the contents of the QM-C registers and allocate space on a stack prior to transferring control to the called procedure.

The data objects used by CALL are defined as synonyms of previously declared variables, and are shown in Fig. 1 (for conciseness the original variable declarations for the QM-1 data objects "local_store" and "f_store" are not shown here - they are described in [8]). Note that the name "reg" is made synonymous with "local_store" and that two alternative data types (an array and a tuple) are associated with this name. Some of the fields of the tuple themselves have alternative data type attributes - for example "reg. inst_reg" is specified as an instance of two different tuple types.

At the time the CALL instruction is to be executed the stack (held in QM-C's main memory) is as shown in Fig. 2 and the first word of a procedure being called is a "mask" whose format is given in Fig. 3. In addition, the following conditions are assumed to hold:

(i) the framepointer fp points to the start of the activation record for the calling procedure; (ii) the program counter pc points to the CALL instruction in memory; (iii) eb denotes the base address for the entire object program; and (iv) inst_ reg contains the operand of the CALL instruction (actually, it points to a word in memory relative to the base address specified in eb that holds the aforementioned mask).

This is stated formally by means of the following assertion:

PRE_CALL:

```
type ls_register = seq [17..0] bit;
type f_register  = seq [5..0] bit;
......
syn main_output = control_store_output; /* main memory output bus*/
syn reg : array [0..31] of ls_register  /*QM-C register file */
        : tuple
            dummy : array[0..11] of ls_register /*not used by QM-C*/
             temp : array[0..3]  of ls_register /*temporaries*/
              var : array[0..7]  of ls_register /*variable regs */
            index : array[0..3]  of ls_register /*QM-C index regs */
                  : tuple
                         fp : ls_register        /*frame pointer */
                         pc : ls_register        /*program counter */
                         eb : ls_register        /*ext. base reg. */
                         ax : ls_register        /*aux. mem index */
                    endtup
                         sp : ls_register        /*stack pointer */
                       scr1 : ls_register        /*scratch reg */
                       scr2 : ls_register        /*scratch reg */
                   inst_reg : tuple
                                 opcode : seq [6..0] bit
                                     ab : tuple
                                              a : seq [4..0] bit
                                              b : seq [5..0] bit
                                          endtup
                              endtup
                            : tuple
                                   c : f_register
                                   a : f_register
                                   b : f_register
                              endtup
          endtup            = local_store:
syn mm_addr_select  : seq [5..0] bit = f_store.fcia;
syn mm_index_select : seq [1..0] bit = f_store.fmpc[1..0];
syn mm_data_select  : seq [5..0] bit = f_store.fcid.
```

Fig. 1

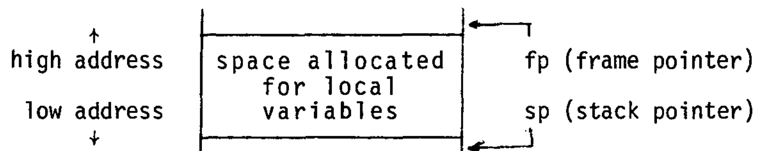

Fig. 2

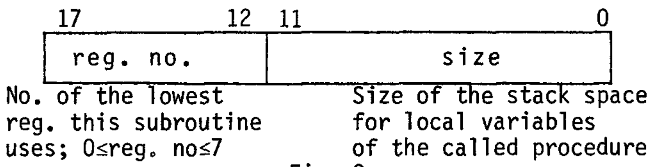
17
12
11
0
reg. no.
size
No. of the lowest reg. this subroutine uses; 0≤reg. no≤7
Size of the stack space for local variables of the called procedure

Fig. 3

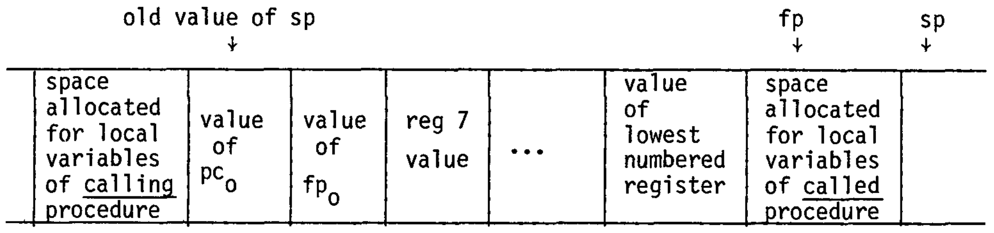
old value of sp
fp
sp
space allocated for local variables of calling procedure
value of pc0
value of fp0
reg 7 value
...
value of lowest numbered register
space allocated for local variables of called procedure

Fig. 4

```
proc CALL;
   do reg.index.pc := reg.index.pc+1          /*pc points to next instruction */
   □ reg.scrl := reg.inst_reg                  /* save offset */
   □ mm_index_select := 26
   od;
   call MAIN_MEM.READ_I_L ;                    /* read mask word onto bus */
   reg. inst_reg := main output ;              /* prepare to decode */
   do mm_addr_select := 28                     /* prepare to save registers */
   □ mm_data_select : 26
   od;
   repeat
      mm_data_select := mm_data_select-1;      /* save registers on stack */
      call MAIN_MEM.PUSH_G                     /* iteratively until reg. */
   until mm_data_select = reg.inst_reg.c+16    /* specified in inst_reg.c has */
                                               /* been saved */
   reg.index.fp := reg.sp                      /* set new frame pointer */
   reg.sp := reg.sp-reg.inst_reg.ab            /* allocate for local vars */
   reg.index.pc := reg.index.eb+reg.scrl+1     /* first instr. of proc */
   mm_index_select := 25;
   call MAIN_MEM.READ_I
endproc
```

Fig. 5

reg.index.fp : ls_register = fp_0 ∧ reg.sp : ls_register = sp_0
∧ reg.index.pc : ls_register = pc_0 ∧ reg.index.eb : ls_register = b_0
∧ reg.inst_reg : ls_register = opd_addr_0
∧ main_mem [b_0+opd_addr_0][17..11] = r_lowest_0 : integer ∧ $0 \le r\_lowest_0 \le 7$
∧ main_mem [b_0+opd_addr_0][10..0] = $local\_space_0$: integer
∧ (∀j : 7,6,..., r_lowest_0)(reg_var[j] = reg_var_0 [j])

The intended postcondition of the CALL instruction prior to activating the next instruction fetch phase will be such that:
(i) The state of the stack and associated pointers should be as shown in Fig. 4. Note that the saved value of pc must be such that it points to the instruction following CALL (in the calling procedure).
(ii) The program counter is pointing to the first instruction of the called procedure.
(iii) The first instruction of the called procedure is on the main memory output bus.

The following assertion formalizes these conditions:

POST_CALL:

main_mem [sp_0] =pc_0+1 ∧ main_mem [sp_0-1] = fp_0
∧ (∀j : 7,6,...,r_lowest_0)(main_mem [sp_0-(9-j)] = reg. var_0[j]
∧ reg. index.fp : ls_register = sp_0-(9-r_lowest_0) -1
∧ reg. sp : ls_register = sp_0-(9-r_lowest_0) -1 -$local\_space_0$
∧ reg. index.pc : ls_register = b_0+opd_addr_0+1
∧ main_output : bus = main_mem [reg.index.pc]

The actual S_A^* code describing the CALL instruction is shown in Fig. 5. Note that this includes invocation of three other procedures inside another mechanism, MAIN_MEM. Thus, to fill in the details of the proof for the CALL procedure, proof outlines for these three procedures must be constructed. Clearly, inside the CALL procedure, the assertions holding at the time any of these MAIN_MEM procedures is called must imply the precondition for the corresponding MAIN_MEM procedure. The postcondition for the latter will then become part of the postcondition of the MAIN_MEM procedure call statement.

For lack of space we cannot elaborate on the proof details here. Most of the actual proof is quite straightforward, however. The only non-trivial part is to show the correctness of the iterative statement and, as is usual in such cases, an appropriate loop invariant [1] must be constructed and used to prove the desired postcondition for the loop. The correctness of the invariant can be shown using induction. For further details, the interested reader may refer to [10].

7. Conclusions

The formal description and verification of architecture designs is very similar to the process of program design and verification. The main distinctions appear to lie in the specific nature of data types, in the pragmatics of the constructs used, and in the kind of information processing systems that the architect is required to design. Keeping these caveats in mind, the idea of formally describing architectures, and verifying their correctness at the design stage itself seems perfectly feasible. An important question that remains to be answered is: given an architectural design in a language such as S_A^*, how can we transform such a design to lower levels of description and demonstrably preserve its correctness. We are currently studying this problem within the framework of the S* family of languages.

8. Acknowledgements

We thank Alan Wagner, Lou Hafer, and the referees for their comments on an earlier version of this paper.

REFERENCES

[1] Alagic, S., and Arbib, M.A., The Design of Well

Structured and Correct Programs, Springer Verlag, N.Y., 1978.

[2] Baer, J-L., Computer Systems Architecture, Computer Science Press, Potomac, MD, 1980.

[3] Barbacci, M.R., "A Comparison of register transfer languages for describing computers and digital systems," IEEE Trans. Comput., C-24,2, 1975.

[4] Barbacci, M.R., and Parker, A., "Using emulation to verify formal architecture descriptions," Computer, May 1978, pp. 51-56.

[5] Barbacci, M.R., Barnes, G.E., Cattell, R.G., and Siewiorek, D.P., "The ISPS Computer description language," Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA., 1978.

[6] Bell, C.G., and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, N.Y., 1971.

[7] Dasgupta, S., "S_A^* : A language for describing computer architectures," in Computer Hardware Description Languages and Their Applications, M.A. Breuer & R. Hartenstein (Ed.), North-Holland, Amsterdam 1981, pp. 65-78.

[8] Dasgupta, S., and Olafsson, M., "Towards a family of languages for the design and implementation of machine architectures," Proc. 9th Annual Symposium on Computer Architecture, IEEE Computer Society Press, 1982.

[9] Dasgupta, S., "Computer Design and Description Languages" in Advances in Computers, M. Yovits (Editor), Vol. 21, Academic Press, 1982.

[10] Dasgupta, S., The Design and Description of Computer Architectures, John Wiley & Sons (Wiley-Interscience), Forthcoming, 1983.

[11] Dietmeyer, D.L., and Duley, J.R., "Register transfer languages and their translation" in Digital Systems Design Automation, M.A. Breuer (Ed.), Computer Science Press, 1975.

[12] Fuller, S.H., Stone, H.S., Burr, W.E., "Initial selection and screening of the CFA candidate computer architectures," Proc. Natl. Comput. Conf., 1977, Vol. 46

[13] Hoare, C.A.R., "An axiomatic basis for computer programming," Comm. ACM, 12, 10, 1974, pp. 549-557.

[14] Hoare, C.A.R., "Notes on Data Structuring," in O-J Dahl, E.W. Dijkstra, & C.A.R. Hoare, Structured Programming, Academic Press, N.Y., 1972.

[15] Jones, J.C., Design Methods: Seeds of Human Future, John Wiley & Sons, London, 1970.

[16] Leung, C.K.C., "ADL: An Architecture Description Language for Packet Communication Systems," Proc. 4th Int. Symp. on Computer Hardware Description Languages, Palo Alto, CA, 1979.

[17] Myers, G.J., Advances in Computer Architecture, John Wiley and Sons (Wiley-Interscience), N.Y., 1981, (2nd Edition).

[18] Owicki, S., and Gries, D.G., "An axiomatic proof technique for parallel programs," Acta Informatica, 6, 1976, pp. 319-340.

[19] Parker, A.C., and Wallace, J.J., "An I/O Hardware Description Language," IEEE Trans. Comput., C-30, 6, June 1981, pp. 423-428.

[20] Patterson, D.A., "STRUM: structured programming system for correct firmware," IEEE Trans. Comput., C-25, 10, Oct. 1976, pp. 974-985.