Error-oriented architecture testing*

by LARRY KWOK-WOON LAI

Carnegie-Mellon University
Pittsburgh, Pennsylvania

ARCHITECTURE VALIDATION

Motivation

Architecture validation is becoming more and more important as diverging cost/performance criteria and competition cause the number of models within a computer family to proliferate. Some popular architectures are now being manufactured by many different companies and the chances of a company inexperienced with the architecture making mistakes is very high. Not only will errors in an implementation cause software incompatibility, the costs of fixing them are usually prohibitively high once there are a large number of defective machines in the field. Excellent evidence demonstrating the inadequacies of present testing techniques is implementation errors discovered in the field for many major computer families. This study was initiated in the hope that an error-oriented approach to architecture testing may provide a better detection of implementation errors.

In this paper, the term architecture refers to the time-independent functional appearance of a computer system to its users. An implementation of an architecture is an ensemble of hardware/firmware/software that provides all the functions as defined in the architecture.

Architecture validation

Architecture validation is the process of validating that a given machine indeed implements a specified architecture. There are three basic approaches:

1. Verification—prove the correctness of the design of an implementation using formal mathematical techniques.
2. Simulation—based on models of the physical building blocks and a description of the design, simulate the implementation to see if it behaves as expected.
3. Testing—establish a certain level of confidence in an implementation by running test programs on a prototype machine.

The first two approaches are most useful when the implementation is still being designed or when architecture specifications are still being formulated. Once a machine is built, however, the only way to find out whether it actually works is to run programs on it—i.e., through testing.

Before one can set out to validate any implementation, one needs to have a specification of the target architecture. There are two basic ways to specify/describe an architecture: (i) using a formal language, e.g. ISPS, VDL/APL, and (ii) using a natural language, e.g., Principles of Operation, Processor handbook. The latter is often the most important because many architectures do not have a formal specification while a natural language description is almost always available, is more readable, and hence is read by users and implementors alike. For verification and simulation, a complete and consistent formal description is needed. For testing, a natural language description is usually adequate for the test programmer, although any ambiguities in the description must be resolved before tests can be derived for the parts affected.

Before we move on to architecture testing, we will briefly review the work that has been done in verification and simulation.

Verification

Verification seeks to confirm absolutely, on paper, that a given implementation does meet its specifications. Two implementation-specific ways of architecture verification are microprogram verification and hardware verification.

The microprogram verification approach can be summarized as follows: given the formal specifications of the target machine and the description of the underlying microengine, the formal verification system looks at a microprogram written for the microengine and attempts to prove that the microprogram running on the microengine would emulate (i.e., implement) the target machine. So far this approach has only been applied to very simple machines.

* This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) ARPA Order No. 3597, and monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

** For a detailed exposition on architecture specification, see Reference 17.
Hardware verification deals with methods to prove the correctness of hardware designs. In this approach, descriptions of low level components and a description of how they are interconnected are taken as input. The goal is to verify that the given interconnection satisfies some higher level specification.

Both microprogram verification and hardware verification can only verify paper designs. Some test programs are still needed to check out the actual hardware. They must also develop accurate models of low level components which are changing rapidly with technology.

Simulation

Simulation has the advantage that it is easier to do than verification. Simulating computer hardware using software, however, usually results in a speed penalty ranging from a thousand to one up to a million to one. Hence it is usually impractical to test a design using simulation beyond running some very short tests that check some internal workings and critical paths. Besides the severe speed penalty, simulation also faces the problem of developing good models for rapidly advancing components and technology.

Testing

Testing has the strong appeal that it deals directly with the physical implementation, which is what one really wants to validate, rather than some abstract description of the implementation. Testing is also much easier to do than verification or simulation. One can start writing a test program with nothing more than a natural language specification whereas verification and simulation both require modeling, formal description of the architecture and of the particular design, and a software system that can carry out the verification and simulation. These advantages make testing the most practical, though not totally satisfying, way to validate an implementation.

The drawback of testing is that it cannot give complete assurance—in practice it often gives less than satisfactory assurance. The former is a direct result of the affirmative nature of testing and of the complexity of a computer—because exhaustive testing, which is the only way to give complete assurance, is impractical for even a simple computer. The latter, however, is usually caused by the lack of good test methodologies and test programmers. It can be drastically improved if the object to be tested can be analyzed in detail before test programs using the analysis as guidelines are written.

The viability of testing as a validation tool lies in the empirical fact that implementations usually have regular structures and therefore the errors made in designing them are not totally random. To illustrate this, let us consider testing the ADD instruction of a computer. In almost all cases, only a tiny fraction of the \(2^n \times 2^n\) (where \(n\) is the word length) possibilities would be tested and then one would declare with confidence that the adder works, and indeed it usually does! The reason for this is that the tests usually cover most of the probable errors: experience shows that the chances of having errors that could not be discovered by the tests is pretty small. If errors are truly random, however, and one is asked to test a black box whose internal structure one does not know about, then one cannot hope to achieve any high level of confidence by testing only a tiny fraction of the input possibilities. Needless to say, testing all the possibilities is out of the question—even \(2^{256} \times 2^{256} = 2^{512} > 10^{19}\). The question then is: amongst a sea of possible errors, how do we pick out the most probable ones and test for them? The error-analysis techniques developed later in this paper attempt to answer this question.

ARCHITECTURE TESTING

Architecture testing defined

Architecture testing is functional testing aimed at validating implementations of an architecture. An architecture testing program is designed to be a tool for certifying different machines claimed to implement a specific architecture. "Functional testing" means that the tests primarily aim at finding design and logical errors rather than problems in realization (e.g. repeatability and bit dependencies) or hardware failures. The level of confidence an architecture testing program can provide depends on the probability of having errors undetected by its tests.

Considerations in writing an architecture testing program

The most crucial constraint for an architecture testing program is time. Any testing that can be done within a reasonable amount of time is but a tiny fraction of all possible tests. The critical issue in writing an architecture testing program is therefore how to select the most profitable tests and test data for a given time constraint.

Unlike diagnostics which often must locate faults rapidly in the field, architecture testing programs can have run time in the order of days. But that is still only a fraction of the test cycles one would like to have. Fortunately the tests in an architecture testing program do not depend on the results of each other and therefore different parts of the program can be run simultaneously on many prototypes in parallel to obtain more test cycles. Program size should not be a constraint since only one test needs to be in core at any one time (can simply use overlays). Because the total program is likely to be huge, however, a good testing methodology should allow automatic generation of test data and test programs to avoid the tedious task of test programming.

Ideally we would like an architecture testing program to be as implementation-independent as possible. However, as we have pointed out before, the only way to get high confidence in testing "black boxes" is exhaustive testing. Therefore any practical architecture testing program must necessarily make certain assumptions about the implementations it is going to be run on in order to cut down the test space. In other words, an architecture testing program must
be written with certain classes of implementations*** in mind. Within the target classes, the program should be non-implementation-specific in that it should be designed to effectively test all implementations within the classes. This is a case of tradeoff between generality and efficiency—one wants to have a program that can effectively validate as many kinds of implementations as possible while there are practical limitations as to how much resources the program can consume.

Recent developments and related work

Two current developments have contributed to the recent interest in architecture testing. One is standardization efforts like the MCF project which need independent validations of prototypes submitted by various contract bidders. The second development is the spread of microprocessors and LSI components. Many microprocessor and LSI parts are now manufactured by a half-dozen companies representing almost as many different implementations. The need for assurance of compatibility, together with pin limitation, have generated considerable interest in functional testing. 19

Some research in the field of program testing is of considerable interest to architecture testing. The work that is closest to architecture testing is that of compiler validation. 33,14 An area of special interest is test data selection techniques—one can represent a function in an architecture by a canonical procedure written in a hardware description language like ISPS and then use the selection techniques to choose test data. Architecture testing and program testing bear many similarities, and research done in one area is likely to benefit the other.

ERRORS IN IMPLEMENTING AN ARCHITECTURE

Why do people make errors? What errors do people and design systems make? If one knows why people make errors, one can try to prevent them in the first place, thereby getting at the root of the problem. If one knows what kind of errors are likely to occur in a particular environment, one can orient one's testing effort accordingly to maximize return on the effort. It is wasteful to test for errors that are almost certain not to occur while more likely errors are not tested for. The kind of errors that people make varies with their task environments. In implementing a computer architecture, the likelihood of different types of errors varies with technology, design tools used, experience and training of the design group, available history of previous implementation errors (designers are usually more aware of them), project management etc..

We began with the conjecture that although every implementa-
Errors in implementing an architecture

The preliminary study revealed eight likely sources of errors. They are:

1. Incomplete and imprecise specification
2. Interdependent side-effects
3. Asymmetry/nonuniformity
4. Logical complexity
5. Boundary values
6. Counter-intuitive and unusual features
7. Inconsistencies in nomenclature and style
8. Missing functions

Each of these categories is explained in detail in the following subsections. Real-life examples, mostly from the PDP-11, are given whenever appropriate. Since the categories overlap with one another, some examples have been somewhat arbitrarily classified.

Incomplete and Imprecise Specification

Whether the incompleteness in an architecture specification is intentional or accidental, the hardware of any implementation does something for the unspecified operations. Users are often tempted to use those peculiar "features" in their programs. If later models do not have the same "features," there is a software incompatibility problem. An incomplete specification may cause implementors to use an incompatible scheme in implementing the unspecified operations.

A specification which appears precise to its writer may be imprecise or ambiguous for others because of nontrivial implicit assumptions made by the former. Following is an example from the PDP-11 handbook:

- The instruction MUL Rn,SRC will cause Rn to contain the low order part of the result if R is an odd-numbered register and cause Rn to contain the high order part of the result if Rn is an even-numbered register. This asymmetry would not have occurred if the multiply instruction is defined such that the low order part of the result, which is what is needed most of the time, is always stored into Rn, and not Rn+1.

- The automatic sign extension that occurs in moving a byte to a register with the MOVB instruction often catches programmers off guard. One would expect a byte instruction to operate on a byte and yield a one byte result. This nonuniformity is actually caused by the more fundamental nonuniformity* that registers are not byte addressable while all memory locations are.

Logical complexity

Some instructions are error-prone due to their sheer complexity. The human mind does not efficiently handle complexities beyond a certain threshold. Complex interactions that change a lot of processor states (trace traps, interrupts etc.) are conceptually hard to understand as well as difficult to implement, especially if multiple activities can occur at the same time. Extra testing is required to ensure the correctness of complex instructions.

* We are not saying that this nonuniformity was a bad design decision; in fact it was probably a good one. We just want to point out that any nonuniformity is a likely source of errors.

Interdependent side-effects

Instructions which have multiple side-effects are error-prone, especially if the outcome of the instruction depends on the order in which the side-effects are carried out. Sometimes ambiguities can occur at the interfaces of architectural features which are individually well-defined. An instruction consisting of multiple operations is inherently ambiguous if the order of the operations is not clearly specified and the effect of the instruction depends on this order. Most often this arises when there are multiple operations on the same register or memory location within an instruction and the order of operations is not explicitly stated in the specification.

Asymmetry/nonuniformity

Asymmetry/nonuniformity often causes additional complexity in programming and in implementation. Asymmetrical/nonuniform side effects, notably condition code settings, are usually counter-intuitive as well.
Boundary values

Boundary values for an instruction are input values that are at the boundaries of different decision regions in the input domain of the instruction. This concept is analogous to decision branches inside a program which compare input or computed values against some test values in order to determine the execution paths that the program should follow. In fact, the inclusion of this category is inspired by the frequent occurrence of boundary value errors in programming. The most prevalent kinds of errors in this category are missing boundaries and off-by-one errors. Missing boundaries are situations in which one or more of the boundaries between decision regions are missing. Off-by-one errors are errors in which a boundary is off a distance of one (for some appropriate definition of distance) away from where it should be.

Counter-intuitive and unusual features

Features that deviate from or behave just opposite to what one would normally expect or find in other architectures are error-prone. They also considerably slow down programmers who have to deal with them. Similarly, inappropriate or non-mnemonic names for instructions invite errors. Without proper explanation and motivation, even a useful feature may create confusion.

- Some condition code settings in the PDP-11 are counter-intuitive. For example, the increment and decrement instructions do not affect the carry.25

Inconsistencies in nomenclature and style

Inconsistencies and exceptions are often introduced due to carelessness or ignorance. It is often penny-wise and pound-foolish to foul up an otherwise uniform and clean style just to squeeze an extra bit of performance out of an architecture.

- In PDP-11 instruction nomenclature, instructions that end with a B are supposed to be byte instructions. The SWAB instruction, despite having a B as the last character of its name, is actually a word instruction—it takes a word operand and generates a word result. It would probably be better to call it SWAP.
- The multiply and divide instructions store the high-order word of their two-word results in Rn and the low order word in Rn+1, which is just opposite to the practice of storing a high byte at the higher address within a word. The same comment applies to the scheme of storing the high part of a floating number in the lower word.

Missing features

This is not really a source of error, but it is put here as a reminder that a good test program should at least test for the existence (not necessarily the complete correctness) of every feature. It is not uncommon that relatively simple features are left out of an implementation due to the oversight or lack of experience of its designers. To illustrate what I mean by features in an architecture, the major features of two typical instructions are presented below.

ADD:
- Add source operand to destination operand and store result in destination.
- If there is a carry out, set the carry bit, clear it otherwise.
- If the result is zero, set the zero indicator, clear it otherwise.
- If the result is negative, set the negative indicator, clear it otherwise.
- If the addition results in an overflow/underflow, set the corresponding indicator.

HALT:
- If in user mode, causes an “illegal user instruction” trap.
- If in supervisor mode, stop all operations:
  1. All flags are left untouched,
  2. The program counter points to the next instruction following the HALT.

A major feature can often be broken down further into several minor features, depending on the complexity of the instruction. To guard against leaving some major features untouched, a comprehensive checklist for features of each instruction should be used. Each entry on the checklist roughly corresponds to a leaf on the decision tree of the instruction.

More likely than not, most features would have been exercised by tests written for other categories, thus it will only be necessary to write special tests for the items that are left untouched by other tests. In order to reduce the test space to a practical size, often only the existence, and not the correct functioning, of features can be established by such tests.

A CASE STUDY OF THE PDP-11 ARCHITECTURE

The architecture testing philosophy advocated in this paper is rather straightforward: given the amount of resources one is willing to spend in testing, try to minimize the probability that an error is undetected. This implies that one should test for the most likely errors first. In fact, these are often the only errors that one can afford to test for. The crucial question here is: what are the most likely errors?
This section presents a "likely error analysis" of the basic PDP-II architecture** as specified in PDP-II processor handbooks. The likely errors are identified and classified using the proposed criteria. It must be pointed out here that what "likely errors" are depends on when in the development of an implementation the tests are made. We have been assuming all along that we will test prototypes that have most instructions "working" and that we are looking for the obscure errors. The PDP-II is selected for case study because (i) it is a major computer family having numerous implementations; (ii) the history of its implementation errors is readily accessible to allow evaluation of the proposed testing strategy; and (iii) the basic architecture is simple enough for a thorough study of this sort.

The analysis is intended to be illustrative rather than complete—someone who is willing to spend the energy needed to analyze an architecture for likely errors can probably turn up more potential bugs. A list of recommended tests is given at the end of each section. As a whole the recommended tests should be viewed as "hole-plugging" tests to be added on top of any testing scheme that covers basic and obvious functions.

Incomplete or imprecise specification

The handling of hardware error conditions is only briefly mentioned in the processor handbook. There is no well-defined priority scheme for handling multiple, simultaneous processor trap conditions. The handbook also does not specify clearly what happens if a trap occurs in the middle of an instruction.

Other problems in the specification:

- The specification of the overflow V bit setting is wrong for the subtract carry (SBC) instruction. It is stated as follows in the manual:
  V: set if (dst) was 100000; cleared otherwise
  It should instead be
  V: set if (dst) was 100000 and C was I; cleared otherwise
- The specification of the carry condition code settings in the SUB and CMP(B) instructions presume particular implementation schemes (see Incomplete and Imprecise Specification).

Tests recommended:

- Trap priority—special hardware is probably required to carry out this test.
- Handling of multiple trap conditions caused by a single instruction.
- Test the V bit setting of the SBC and SBCB instructions.
- Test the C bit setting of SUB and CMP(B) instructions.

** For the purpose of this study, FIS, FIN, and memory management floating point instructions are not considered part of the basic PDP-II architecture.

Interdependent side-effects

In the PDP-II, many instructions having interdependent side-effects are also inherently ambiguous because the architecture specification often does not specify the orders of execution for multiple side-effects.

Multiple, explicit operations on the same register

A double operand instruction is inherently ambiguous if (i) its source addressing mode uses Rn and its destination addressing mode is one of (Rn)+, @(Rn)+, -(Rn), and @(Rn); or if (ii) its source mode is one of (Rn)+, @(Rn)+, -(Rn), and @-(Rn) and its destination mode uses Rn. For example:

- OPR Rn,(Rn)+—if the second operand is fetched from memory (the first operand needs no fetching) and autoincrement is performed before carrying out the operation, the incremented Rn will be used as the source operand. But if the operands (including register operands) are first stored into temporary registers as they are fetched, the original value of Rn would be used as the source operand. The latter is more intuitive, but the processor handbook makes no statement about this ambiguity.

Test recommended:

- For all double operand instructions, test all the 64 combinations of addressing modes that are ambiguous. Of course, we would need to define how the combinations should behave before we can test them.

Multiple, explicit and implicit operations on the same register

PC (register 7) is automatically incremented each time it is used to fetch a word from memory. It is used implicitly in some addressing modes while SP (register 6) and some memory locations (notably those reserved for trap vectors) are used implicitly by several instructions. Instructions that operate on these registers or memory locations explicitly as well as use them implicitly at the same time deserve special attention.

- A double operand instruction that uses the PC is ambiguous if (i) its source is PC and its destination is one of (PC)+, @(PC)+, -(PC), @(PC), X(R), and @X(R): or if (ii) its source is one of the list just given and the destination is PC.

Tests recommended:

- Test all the 12 ambiguous combinations of PC addressing modes.
Modification and decision on the same operand

If an instruction both modifies its operand(s) and uses it for a decision (branch, setting of condition codes etc.), then the relative order of the modification and the decision becomes critical.

Tests recommended:
- Test JMP (Rn)+ and JSR Rm,(Rn)+ which are both ambiguous.

Asymmetry/nonuniformity

Fundamental asymmetries/nonuniformities
- Registers are not byte addressable while all the memory locations are.
- Highest page of main memory is reserved for the I/O page.
- Some memory locations are special (e.g. processor status word, stack limit etc.).
- Some memory locations are not writable (e.g. some status registers).
- Some instructions implicitly use special memory locations (EMT, TRAP, BPT, and IOT).

Tests recommended:
- Make sure that the I/O page is in the right place.
- Make sure that all the special memory locations are there. For instructions that use special memory locations, make sure that they access the correct special locations when executed.

Other asymmetries/nonuniformities
- Logical instructions COM(B), BIT(B), BIC(B), and XOR have different condition code setting conventions. In COM(B), the V bit is cleared but the C bit is not changed.
- The autoincrement deferred addressing mode @(Rn)+ always increments Rn by 2, even for byte instructions, whereas (Rn)+ increments Rn only by 1 for byte instructions. Similarly, @-(Rn) always decrements Rn by 2, even for byte instructions, whereas -(Rn) decrements Rn only by 1 for byte instructions.
- Automatic sign extension in MOVB - ,Rn (see Asymmetry/Nonuniformity).
- MUL & DIV instructions store low order part of result into R v 1 (see Asymmetry/Nonuniformity).

Tests recommended:
- Test COM(B) for the correct setting of the C bit.
- Test @(Rn)+ and @-(Rn) for incrementing/decrementing Rn by two.
- Test sign extension in MOVB - ,Rn.

Logical complexity

There are not many complex instructions or features on the basic PDP-11. The MARK instruction and trace trap are probably the most complex features and the trap instructions (EMT, TRAP, BPT, IOT, RTI, RTT), SOB, JSR, and RTS instructions deserve some extra testing effort.

Tests recommended:
- Test the previous instructions/features to make sure that the right sequences of operations are performed when they are invoked.

Boundary values

It is straightforward to figure out the boundary values for logical instructions—just test all four combinations for each bit position. It is often the case for arithmetic instructions, however, that there are too many boundary values and subsets must be chosen among them. Without going into detailed arguments, we assert that testing just a few key points on a boundary is almost as good as testing all the points on the boundary. This approach is illustrated below through a "boundary value analysis" of the ADD instruction.

The input domain is partitioned into different decision regions for each of the four condition codes N,Z,V, and C (which stand for Negative, Zero, Overflow, and Carry respectively). For example, one region would consist of all input values that generate results which are less than zero and will therefore set the N bit while the complement of this region would consist of those input values that generate results which are not less than zero. There are well-defined boundaries between the partitions (see Figure 1). For example, the boundary values lying on the two sides of the longest N bit boundary are given by \( y + z = 1 \) and \( y + x = 0 \), respectively (Figure 2). Note that the partitioning is symmetrical with respect to the line \( y = x \) because ADD is a commutative operation and that the partitions for different condition codes often have common boundary values. Besides correct setting of condition codes, one may also want to test for the following: the correct operation of each output bit†, correct generation and propagation of carry from each

---

Error-oriented Architecture Testing 571

From the collection of the Computer History Museum (www.computerhistory.org)
Region in which the C (Carry) bit should be set

Region in which the N (Negative) bit should be set

Region in which the V (Overflow) bit should be set

Boundaries:

N: \( y + x = 0 \) & \( y + x = -1 \); \( y + x = 2^{15} - 1 \) & \( y + x = 2^{15} \); \( y + x = -2^{15} \) & \( y + x = -(2^{15} + 1) \)  
Z: \( y + x = 0 \) & \( y + x = -1 \) & \( y + x = 1 \)  
V: \( y + x = -2^{15} \) & \( y + x = -(2^{15} + 1) \); \( y + x = 2^{15} - 1 \) & \( y + x = 2^{15} \)  
C: \( y + x = 0 \) & \( y + x = -1 \); \( y \geq 0 \) & \( y = 0 \); \( x \geq 0 \) & \( x = -1 \), \( y \geq 0 \) & \( x = 0 \), \( y \geq 0 \)

Number of points on the boundaries:

N: \( 2^{18} \)  
Z: \( 3 \times 2^{15} \)  
V: \( 2^{17} \)  
C: \( 2^{18} \)

Figure 1—Boundary values for condition code settings of the ADD instruction.
Each of these have their own set of boundary values that need to be tested. In general, one first determines the things (actions) that one wants to check out, then partitions the input domain into decision regions for each particular action and finally picks out the boundary values as test data.

Ideally all boundary values of an instruction should be included as test points in testing the instruction. If one cannot afford to test all of them (e.g. in a 32 bit machine), a subset of "key points" could be chosen for testing. The minimum set of test points recommended are those values that are either at the "corners" of boundaries or at extreme points of the input domain. The existence of a boundary can be tested by crossing it from one boundary value to a neighboring value on the other side of the boundary. The process of selecting the key points is illustrated in Figure 3.

**Test recommended:**

- Verify the operation (condition code settings, etc.) of each logical and arithmetical instruction for all its boundary values. If this is not practical, pick a subset of "key points." For logical instructions, the simplest test is to assume independence among different bits in a word and apply the input pairs of 000000 & 000000, 000000 & 177777, 177777 & 000000, and 177777 & 177777.§

**Counter-intuitive and unusual features**

- Complement (COM), a logical instruction, sets the C bit instead of leaving it untouched.
- The increment (INC) and decrement (DEC) instructions do not affect the carry.

**Tests recommended:**

- Test the C bit setting of the COM(B) instruction.
- Test that the carry bit is truly unaffected by INC(B) and DEC(B) instructions.

**Inconsistencies in nomenclature and style**

- The usage of the "contents of" notation, ( ... ), is inconsistent.
- The notation used to represent a register is also inconsistent.

**Test recommended:**

- None. In this case none of the inconsistencies is very serious. If one is writing a complete test program, however, then it would be worthwhile to pay special attention to even minor inconsistencies.

---

§ A better test is to apply some algorithmic patterns (like walking 0's and 1's) to systematically check for cross-coupling between bits.
Missing features

A good test program should try to test for the existence (not necessarily the complete correctness) of as many features as possible.

Tests recommended:

- The existence of all major features should always be checked out.
- As our resources permit, test for the existence of as many minor features as possible.

RESULTS AND CONCLUSIONS

The eight error categories are conjectures established based on their potential of causing implementation errors. To examine how well they capture actual errors, the tests recommended in the previous section were calibrated against two error histories. The first one is a list of incompatibilities among models of the PDP-11 and the second one is a list of errors found in an ISP "implementation." In both cases, only errors in implementing the basic architecture* are considered.

Comparison with published incompatibilities among various models

This error history is published by DEC itself and it lists known differences among five implementations of the PDP-11 architecture (see Appendix B). These are usually obscure errors that have slipped through the conventional validation tests. Among the 14 implementation errors, eight would definitely be caught by the recommended tests, three would probably be caught and the remaining three would likely slip through. All, however, fall into five of the eight categories and hence would likely be caught by more detailed tests (which requires a more detailed analysis of the PDP-11 architecture than what was done).

Comparison with errors found in the PDP-11 ISP

The ISP description of a computer architecture can be considered an implementation of that architecture through emulation on the "Register Transfer Machine." A PDP-11 ISP description was recently written and debugged.** A rather complete history of the errors that have been discovered in the description has been kept. The comparison (see Appendix C) revealed that eight of the 13 errors would definitely be caught by the recommended tests. Two would probably be caught and the remaining three would likely slip through. All, however, fall into the error categories and would conceivably be detected by more detailed tests.

Discussion

The above comparisons suggest the following:

1. An error-oriented architecture testing program written with the proposed categories as primary test targets can augment existing test schemes. In the first case, the recommended tests caught a significant percentage of obscure errors that have slipped through conventional tests. More encouraging is that all the implementation errors in both cases fall into the eight categories. Hence if someone devotes enough time (perhaps months, a reasonable investment for a major computer family) to develop an architecture testing program using the proposed methodology, most of these errors can conceivably be caught by the detailed tests.

2. Exercises like the "likely error analysis" presented can help architects to improve their specifications and reduce implementation errors caused by problems in the specification. Such exercises are also useful in spotting error-prone areas, which often cause difficulty in implementation as well as programming, in the architecture itself.

In retrospect, other than a few categories like boundary values which can conceivably be rigorously defined, most of the proposed categories have rather "soft" criteria and require human judgment in applying them. A lot more work is still needed to make them more specific and more amenable to automation. In addition, difficult areas such as floating point instructions have not been dealt with. We nevertheless hope that the proposed categories can serve as helpful guidelines for those who are going to write architecture testing programs. Analyzing an architecture specification to identify potential errors is a very time-consuming process, however, and the usefulness of any testing methodology ultimately depends on automation. Toward this end, research should continue on analysis techniques and the automatic generation of test data and test programs.

ACKNOWLEDGMENT

The author would like to thank Dan Siewiorek for initiating the study and helping along the way. Thanks are also due to Mario Barbacci, Len Shustek, Bob Sproull, Richard Swan, and the referees for their valuable suggestions.

REFERENCES

A brief description of PDP-II addressing modes and instructions

Single operand instructions have the format OPR destination and usually perform \( d \leftarrow op \, d \).

Double operand instructions have the format OPR source, destination and usually perform \( d \leftarrow s \, op \, d \). Each operand can be accessed using one of eight addressing modes, giving a cross-product of 64 ways of addressing operands in a double operand instruction.

### Addressing Modes

<table>
<thead>
<tr>
<th>Modes</th>
<th>Symbolic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>(R) is operand</td>
</tr>
<tr>
<td>1</td>
<td>(R)</td>
<td>(R) is address</td>
</tr>
<tr>
<td>2</td>
<td>(R)+</td>
<td>(R) is addr., increment R after fetching</td>
</tr>
<tr>
<td>3</td>
<td>@((R)+)</td>
<td>(R) is addr. of addr., incr. R after fetching</td>
</tr>
<tr>
<td>4</td>
<td>-(R)</td>
<td>decrement R before fetching, (R) is addr.</td>
</tr>
<tr>
<td>5</td>
<td>@(R)</td>
<td>decrement R before fetching, (R) is addr. of addr.</td>
</tr>
<tr>
<td>6</td>
<td>X(R)</td>
<td>indexing, (R)+X is addr.</td>
</tr>
<tr>
<td>7</td>
<td>@(X(R))</td>
<td>(R)+X is addr. of addr.</td>
</tr>
</tbody>
</table>

### APPENDIX B

#### Architectural incompatibilities among five implementations of the PDP-11

The incompatibilities listed below are compiled from a list in Microcomputer Handbook published by Digital Equipment Corporation in 1976. The list, entitled "LSI-11, PDP-11 Programming/Hardware Difference List," listed the known differences among five implementations of the PDP-11 architecture: LSI-11, PDP-11 05/10, 15/20, 35/40, 45. In compiling the following, differences among features that are not part of the basic PDP-11 architecture are not considered. Next to each of the architectural incompatibilities is listed the error category that the incompatibility belongs to and an indication of whether it would have been caught by the tests that we have recommended.

<table>
<thead>
<tr>
<th>Incompatibility in</th>
<th>Category</th>
<th>Ambiguity in Sequence</th>
<th>Would It Be Caught With the Tests?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. OPR R(R)+ or OPR R,−(R)</td>
<td>ditto</td>
<td>ditto</td>
<td>yes</td>
</tr>
<tr>
<td>2. OPR R, (R)+ or OPR R,−(R)</td>
<td>ditto</td>
<td>ditto</td>
<td>yes</td>
</tr>
<tr>
<td>3. OPRPC,X(R) or OPRPC, (X(R)</td>
<td>ditto</td>
<td>ditto</td>
<td>yes</td>
</tr>
<tr>
<td>4. JMP(R)+, JSR Rm.(Rn)+</td>
<td>Nonuniformity</td>
<td>Nonuniformity</td>
<td>yes</td>
</tr>
<tr>
<td>6. SWAB does not change V in some models</td>
<td>Missing Functions</td>
<td>Missing Functions</td>
<td>yes</td>
</tr>
<tr>
<td>7. Bus addresses of the registers are special</td>
<td>Nonuniformity</td>
<td>Nonuniformity</td>
<td>yes</td>
</tr>
</tbody>
</table>
8. Power fail trap has different priority w.r.t. RESET instruction
   Incomplete Spec. no

9. RTT instruction not implemented
   <deliberate omission—doesn’t count>
   Logical complexity probably

10. RTI behaves differently with the T bit set
    Priority between trace trap and interrupt is different
    Incomplete spec. yes

12. Trace trap will sequence out of WAIT instruction on some models
ditto probably

13. Direct access to Program Status Register (a special memory location) can change the T bit in some models
    Nonuniformity yes

14. Odd address/non-existent traps using the stack pointer
    Incomplete spec. no

15. Guaranteed execution of the first instruction in an interrupt routine
    Incomplete spec. no

16. Odd address trap not implemented on the LSI-11
    <deliberate omission>

17. Effect of bus errors on PC/register modification
    <part of error handling, whether this is part of the architecture is arguable>

<table>
<thead>
<tr>
<th>Error</th>
<th>Category</th>
<th>Would It Be Caught?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. DEC did not set V bit when destination was 100000.</td>
<td>boundary value</td>
<td>yes</td>
</tr>
<tr>
<td>2. MOV B did not sign extend byte moved to register.</td>
<td>asymmetry</td>
<td>yes</td>
</tr>
<tr>
<td>3. SBC did not set the V condition code</td>
<td>missing fn.</td>
<td>yes</td>
</tr>
<tr>
<td>4. SWAB did not assign the result of the SWAB back to memory</td>
<td>missing fn.</td>
<td>yes</td>
</tr>
<tr>
<td>5. Byte instruction using indexed deferred addressing mode didn’t work correctly.</td>
<td>nonuniformity</td>
<td>no</td>
</tr>
<tr>
<td>6. MARK instruction reset the stack pointer wrong</td>
<td>logical complexity</td>
<td>yes</td>
</tr>
<tr>
<td>7. PSW did not have bus address 177776.</td>
<td>nonuniformity or missing fn.</td>
<td>yes</td>
</tr>
<tr>
<td>8. SOB dropped highest bit of offset</td>
<td>asymmetry</td>
<td>no</td>
</tr>
<tr>
<td>9. ASRB instruction operates on wrong field that is off by one bit.</td>
<td>missing fn.</td>
<td>yes</td>
</tr>
<tr>
<td>10. ASHC &amp; ASH: C and V bits are not set correctly (spec. not clear).</td>
<td>incomplete spec.</td>
<td>no</td>
</tr>
<tr>
<td>11. MUL stores intermediate result in a 17 bit register instead of a 16 bit register</td>
<td>peculiar to ISP, would probably be caught by boundary value tests.</td>
<td></td>
</tr>
<tr>
<td>12. DIV: when source register addr. is odd garbage is produced.</td>
<td>incomplete spec./inconsistency in style</td>
<td>probably</td>
</tr>
<tr>
<td>13. DIV: divide by zero did not set condition codes and abort.</td>
<td>missing fn.</td>
<td>yes</td>
</tr>
</tbody>
</table>

APPENDIX C

Errors found in the PDP-11 ISP

The errors to be listed are compiled from a collection of memos documenting the errors that have been found in the PDP-11 ISP after it has been released by its author. The original PDP-11 ISP was written by Dan Siewiorek at Carnegie-Mellon University and has been maintained by Alan Parker of NRL. For the purpose of this paper, we have left out errors concerning user/supervisor/kernel modes, memory management, floating point instructions, and those errors peculiar to ISP as a programming language. Next to each of the errors is listed the category that the error belongs to and an indication of whether it would have been caught by one of the tests that we have recommended.