SADL

A SYMBOLIC ARCHITECTURE DESCRIPTION LANGUAGE

A Thesis Presented in Partial Fulfilment

of the requirements for the Degree of

Master of Arts in Computer Science

at Massey University

Thomas William Livingstone

1985

Abstract

This thesis develops a new language capable of specifying computer architecture at the symbolic, or assembly language level.

The thesis first provides a representative sample of current, or proposed, computer description languages and discusses four of the languages and their merits with regard to the symbolic approach. Next, a model is proposed of computer architecture at the level which is visible to an executing sequence of instructions. This model is based on the assembly language level of computer architecture. Next, the Symbolic Architecture Description Language (SADL) is described. Finally, Build, a LISP program which takes SADL architecture descriptions and generates functions and data structures for use in simulating architectures, is described.

Acknowledgements

I would like to thank the following for their support and assistance during this thesis:

Nola Simpson, for being my supervisor;

Paul Lyons, for his excellent criticisms of the manuscript;

The staff of the Massey University Computer Centre for their cooperation.

Table of Contents

1 Introduction

This thesis proposes a language for symbolically specifying the execution environment of assembly language programs. The assembly language level of description was chosen as it is the most abstract level which is still capable of specifying the instruction set functionality of a computer. Higher level abstractions, such as compilers and interpreters, no longer allow explicit access to the physical machine state, while lower level descriptions have little meaning to the software engineer.

Computer Design, once an area of individual artistic expression, is becoming the result of systematic cooperation between the members of a team, often a large team, frequently aided by automated design tools. Members of the design team must be able to communicate with each other, and with their design tools, without ambiguity, and to this end a number of formal languages have been developed for the description of computer systems.

It has become a truism that a computer system consists of a number of layers, each describable in terms of a particular model. In this thesis, we shall find the level described by the ISP (Instruction Set Processor) model [Bell71] to be the most useful. A computer architecture defined in terms of this model would comprise:

(i) a set of registers,

(ii) a memory which contains the encoded instructions,

(iii) a set of functions which

  (a) produce the effective address for obtaining and storing the operands and

  (b) specify the actions required to implement the instructions.

(iv) a finite state machine which defines the loading, interpretation and execution of instructions defined for the architecture.

There are two approaches to modelling an architecture at the ISP level. The traditional method (adopted in the specification language ISPS [Barb81]) is a mechanical view: the architecture is viewed as a structure consisting of registers and decoding functions which operate on the machine code of the architecture.

The second approach is a symbolic view: it is derived from the Assembly Language model of architecture. It ignores the mechanics of encoding and decoding - the instruction is only ever represented in symbolic form - and models the decode and execute cycle as a language interpretation cycle.

Why use the symbolic approach ?

1. It is the natural tool for software engineers.

   A software engineer who programs an architecture directly (as opposed to using a high level language) makes use of the symbolic level and an Assembler. The costs of programming in machine code versus assembly language and the functional equivalence of the two means that machine code programming has been superceded by assembly language programming, except possibly for some extremely specialised applications.

2. It is a natural pedagogic tool.
   Because people are familiar with the symbolic approach to architecture, it is easier to comprehend architectures when expressed symbolically. This is important when attempting to learn new architectures, when comparing two architectures or when evaluating an architecture.

3. It allows direct simulation of the symbolic program.
   The normal process when simulating the execution of programs on a particular architecture is to write the programs (normally in assembly language), translate them into the machine code for the target architecture and run them on a simulator which emulates the instruction and register sets of the target machine.

   Having the architecture specified symbolically bypasses the translation phase as the assembly language program may be executed directly by the simulator. This saves programmer time and therefore saves money. Balanced against this is the increased cost in processor time of executing an interpreted program rather than a compiled program. Also, the symbolic tracing of instruction execution is simplified and protection mechanisms against faulty programs are easier to install; for instance it would be impossible for a running program to try executing data, an occurrence common in out-of-control machine code programs.

4. It can fully specify the register set of an architecture, and external lines may also be modelled indirectly as registers. The symbolic approach allows the register set of an architecture to be specified to the same detail as the mechanical approach to ISP specification. Thus there is no expressive capability lost when using the symbolic approach over the mechanical approach.

5. Fundamental to the symbolic approach is the fact that each machine instruction has one equivalent symbolic instruction and that the functionality of both is the same. This is a widely recognised view of pure assembly language (as opposed to macro-assembly language).

Section 1.2 of chapter 1 examines four languages which are used, or have been proposed for use in describing the instruction set processor level. Two of the languages, LISP and VDL, deal with instruction set processors at the symbolic level while the other two languages, Pascal and ISPS, deal with the machine code level.

Chapter 2 proposes a model of computer architecture which is centred on the view of an executing program within a machine. The model is based upon the stored program concept with a single execution unit and single instruction and data streams; this excludes architectures based upon array and vector processing as well as systolic architectures.

Chapter 3 defines both the syntax and semantics of the Symbolic Architecture Description Language (SADL) and shows the capabilities and restrictions of the current version of the language.

Chapter 4 describes software which processes a description in SADL and produces a set of data structures and functions which may be used to simulate the architecture when provided with an assembly language program. It is an application intended to test the validity of SADL.

1.1 Multi-level Architectures and Virtual Machines

One of the major concepts that has evolved in computing in the last fifteen years has been the view of a computer system as a layered hierarchy of abstract machines. At the top of the hierarchy are user applications and at the bottom is the physical specification of the electronic components which combine to form the hardware.

Each level may be viewed (more or less) as a complete architecture independent of those levels in the hierarchy either above or below it. This view is invaluable in simplifying the task of designing or analysing computer systems.

There are differing views as to what constitutes each layer, but Siewiorek, Bell and Newell [Bell71,Siewiorek82] have proposed a layering that suits the author's purposes and is quite widely recognised. I shall refer to this as the Bell model.

In the Bell model there are four main levels which are subdivided into sublevels. The main levels are: Circuit level, Logic level, Program level, PMS level.

The only level of relevance to the software engineer is the program level, because this level is broken down into the ISP (Instruction Set Processor) sublevel, and the High Level Language sublevel which is itself broken down into Operating System, Run-time System, Application Routines and Applications Systems sublevels.

Example 1.1

```
-----------------------------------------------------------|
PMS                                                        |
-----------------------------------------------------------|
Program       | High Level        | Applications Systems   |
              | Language          |------------------------|
              |                   | Applications Routines  |
              |                   |------------------------|
              |                   | Run-time System        |
              |                   |------------------------|
              |                   | Operating System       |
              |-------------------|------------------------|
              | Instruction                                |
              | Set Processor                              |
-----------------------------------------------------------|
Logic                                                      |
-----------------------------------------------------------|
Circuit                                                    |
-----------------------------------------------------------|
```

The Assembly Language sublevel fits into the hierarchical view just above the ISP sublevel and below the Operating System sublevel (although Tanenbaum [Tanenbaum76] views the assembler level as being above the operating system level).

The reasons for placing Assembly Language at this point in the hierarchy are these:

(i) In the abstraction process, information is hidden or lost. Anything that may be specified by an Assembly Language program may be specified in greater detail at the ISP sublevel; this indicates that the Assembly level is an abstraction of the ISP sublevel.

(ii) Similarly, an Operating System is a composition of concepts expressible in Assembly Language. Its component subroutines, coroutines, and programs are built up from assembler-level instructions, either directly or (as in the case of UNIX and Burroughs' MCP which are written in high level languages) indirectly.

Where do compilers, which bypass the assembler level and directly produce code at the ISP level, fit into the model? Their mapping from a particular level in the hierarchy of abstract machines to another, lower level may bypass one or more levels. However the number of levels which a compiler bypasses does not invalidate the hierarchical structuring of abstract machines.

1.2 Current Architecture Description Languages

There currently exist a considerable number of languages for describing computer architectures at various levels. Most of these straddle the Register Transfer and the ISP levels. There seem to be almost no generally recognised languages which approach the ISP level from the language (or symbolic) direction.

Subrata Dasgupta [Dasgupta82] surveys a group of languages which he calls Computer Design and Description Languages (or CDDLs). The survey concentrates on ISPS, S*A and the CONLAN extensible language system.

Two points made by Dasgupta are significant. The first is that at the time of writing (1982) CDDLs had not been generally accepted by the computer design community. The second point is that the majority of CDDLs that have been proposed have fallen into the Register Transfer level of description. This is partly true of most of the languages described here although they all have applicability at the ISP level. Only LISP and VDL have the ability to specify architecture at the symbolic level.