The design of a Pascal based microprogramming language is described. It is a Pascal dialect adapted to the task of microprogramming a highly pipelined, horizontal micro-architecture. The nature of high level microprogramming languages, data and control structures, and the effect of a microprogramming environment on language and translator development are discussed. Certain complexities of horizontal micro-architectures are outlined.

Introduction

CHAMIL is a microprogramming language designed at Sperry Univac Computer Operations (MCO) to improve target machine development and firmware environments. The goal of CHAMIL was to improve upon currently available microprogramming languages and develop a procedure-oriented high level language capable of generating horizontal code for a set of similar micro-architectures. The project was approached in phases. The first was to investigate existing and hypothetical languages. The second phase included documentation of a suggested syntax, and specification of semantics. The next phase consisted of the implementation of the language. Final phases will include extension of the language to other similar architectures and generalization to accommodate translation of code destined for dissimilar architectures.

Microprogrammability

In order to evaluate existing or hypothetical languages one must establish a system of measure. The following criteria were used as the measure of usefulness of a microprogramming language:

1) ease of algorithmic expression
2) efficiency of generated code
3) maintainability and readability
4) functionality

Such a measure is termed programmability. Applied to microprogramming, it would be microprogrammability.

Traditionally, microprogramming languages measure relatively low in areas such as: (1) algorithmic expression, maintainability, and functionality. Most existing microprogramming languages offer the microprogrammer the ability to quite clearly express which field gets what value at some particular address. The intent, the algorithm to be performed, is usually quite successfully hidden. Significant improvements can, and should, be made to areas (1), (3) and (4) without degrading performance.

There are basically six broad categories of microprogramming languages [9]:

1) bit stuffer
2) assembler
3) register transfer
4) macro register transfer
5) procedure-oriented, machine dependent
6) procedure-oriented, machine independent

Bit stuffers simply provide the ability to insert bit strings into appropriate fields within a micro-instruction. Assemblers allow mnemonic insertion of bit strings as discussed above. Register transfer languages provide a higher level assignment capability, but strictly speaking add nothing more than syntactic frills. Macro register transfer languages allow generalized building or definition facilities, often including parametric substitution. Procedure oriented languages provide additional constructs such as flow of control, compound statements, and structural data definition. The question of machine dependence/independence is central to current research.
Procedure oriented languages contain those attributes traditionally characteristic of a high level language. Such languages represent a consistently high level of microprogrammability across all categories. CHAMIL is a procedure-oriented, and in several respects, machine independent language. Its translator is organized such that the inevitable machine dependencies existing between similar architectures can be specified using tables which are selected during compilation of a particular translator. A similar approach is suggested by Dasgupta [1], wherein a translator's semantics are not complete until a machine dependent portion has been specified.

Pascal Model

Pascal was used as a model for development of CHAMIL. Pascal is a highly structured, procedure-oriented language [10]. It provides a sophisticated data definition facility and a standard set of high level control constructs. CHAMIL contains most of the Pascal control constructs, including 'WHILE', 'IF-THEN-ELSE', 'REPEAT', and so on. It retained a subset of the data definition facilities, including the 'RECORD' facility and the associated 'WITH' construct. Certain concessions were made regarding the 'GOTO' statement; in the interests of performance, some Pascal 'GOTO' restrictions were lifted. The elemental data type for CHAMIL is the unsigned 'INTEGER', which represents the basic data element of the micro-architecture in question. CHAMIL also includes a string substitution macro facility. Refer to Dasgupta [1], for a similar description of a Pascal-like data definition which is built from a 'BIT' data type.

Micro-architectures

A micro-instruction is a bit string which occupies one control store memory cell. A micro-operation is a task capable of being performed by a processor during one machine clock cycle. Encoding of a micro-instruction refers to the possible combinations of directives which can occupy the same field within a micro-instruction. Vertical architectures are characterized as having one micro-operation per micro-instruction. Horizontal architectures are characterized by being able to perform many micro-operations per micro-instruction.

Vertical architectures typically resemble macro-architectures. Compilation techniques including optimization are readily applicable to generation of code for this class of machine. Horizontal architectures introduce the possibility of concurrency or parallelism. Typically, such machines are highly encoded. CHAMIL was developed to generate code for a class of highly encoded, horizontal, multilevel pipelined machines.

Contrasting Environments

The criteria for measuring microprogrammability can also be the criteria used in developing some measure of usefulness for any programming language. The set of microprogramming languages are, after all, simply a subset of the set of programming languages. The distinction occurs when considering environmental factors or the ramifications of the micro-architecture upon the class of language. What is interesting, is the environment confronting a microprogramming language as opposed to a programming language. Admittedly, some of these contrasts apply only to the translation of a particular language class.

A high level programming language (HLL) typically deals with data objects which are homogeneous. Data reside in cells of memory, all of which share the same characteristics, and all of which are operated on in alike manner. Another characteristic of this homogeneous environment is its virtually unlimited size. A high level microprogramming language (HLMPL) typically operates within a non-homogeneous environment. Data reside in registers. Registers can have differing bit lengths and specialized characteristics. No one standard memory bus is guaranteed to exist connecting all elements. Access is often restricted because of path or timing considerations. Frequently, a micro-architecture will include a set of scratchpad registers. Functionally, these registers can be used as homogeneous memory. Data structures may be allocated to these areas by a translator. In contrast to the massive memory available to a programming language, the scratchpad area is indeed quite small. There is no guarantee that a path exists between scratchpad registers and the ALU (arithmetic logic unit). These environmental differences are apparent when contrasting CHAMIL with Pascal. Pascal has a generalized assignment and expression evaluation scheme. CHAMIL has a complex assignment mechanism and restricted expression evaluation. In CHAMIL, only those registers which have access to the ALU are connected to the output of the ALU, can appear in an expression. Non-symmetry of the ALU chip can influence translation, but it can easily be made transparent to the microprogrammer.
It is currently an open question to what extent the microprogrammer should be able to control his environment. The language purists would argue that the translator should manage all resources. The realist, for reasons usually more pragmatic than theoretical, desires mechanisms in the language whereby the microprogrammer is given resource visibility. Other reasons for the latter approach should become more clear later.

Control store addressing mechanisms vary more widely at the micro-architecture level. The two most frequently encountered are:

1) \[ \text{<next instruction address>} := \text{<current instruction address>} + 1 \]
2) \[ \text{<next instruction address>} := F (\text{<current instruction>}) \]

where \( F \) is some function on the contents of an instruction. Paging of control store, in the sense of standard size subdivision, is common. This results in restricted transfer of control operations. Address alignment requirements pose further problems when trying to develop a translator capable of code generation for a group of these architectures.

Assignment Statement Enhancement

An extension to the underlying structure of the Pascal assignment statement was developed for CHAMIL to aid in specification of data path usage. It involves creation of a graph, in tabular form, which represents the internal structure of the micro-architecture. Each non-homogeneous register and internal bus is given a name and predeclared at a lexical level preceding the first available to the microprogrammer. Associated with each predefined register or bus is a list of attributes. One of the attributes is a list of possible sources from which the register or bus may be loaded. Implicit in the organization of the register/bus graph is the restriction that no path exists which could take more than one machine clock cycle to traverse. An algorithm has been devised which traverses the graph in an attempt to locate a path from one register to another. At each intermediate node, data is collected from that node's attribute list such that the appropriate code can be generated after a legal path is found. CHAMIL considers failure to locate a path a syntax error. Refer to figure 1(b); for example, the microprogrammer writes the following:

\[ \text{A\_REGISTER} := \text{B\_REGISTER}; \]

CHAMIL will break it up into its constituent micro-operations, e.g.

\[ \text{Z\_BUS} := \text{B\_REGISTER}; \]
\[ \text{ALU} := \text{Z\_BUS}; \]
\[ \text{Y\_BUS} := \text{ALU}; \]
\[ \text{A\_REGISTER} := \text{Y\_BUS}; \]

Then, because it was able to locate a path, it will generate the appropriate code to realize the four micro-operations.

Encoding Resolution

CHAMIL is responsible for insuring the integrity of the microcode it generates. In an encoded horizontal environment, the combinations of field use within the micro-instruction are quite complex. The translator for such an environment necessarily reflects this complexity. If a misuse of fields goes unnoticed the possibility of driving more than one value onto the same bus exists. This situation can result in destruction.
CHAML is written in Pascal. It uses the Pascal 'SET' facility to manage integrity. Every logical field of the microword is defined as a set of bits. They are also associated with an opcode set representing the group of operation codes upon which the field is meaningful. As values are placed in the various fields, set intersections are performed. If the intersection of the field's definition set and a set of currently unused bits is not identically the field's definition set, duplicate use of some set of bits has occurred.

Using Pascal notation, suppose:

\[
\text{ADDRESS_FIELD := [BIT-23 .. BIT-34]; }
\text{LITERAL_FIELD := [BIT-29 .. BIT-44]; }
\text{UNUSED_BIT_SET := [BIT-1 .. BIT-80]; }
\]

Suppose the following CHAML instructions were parsed:

\[
\text{A_REGISTER := 12; }
\text{GOTO THAT_LABEL; }
\]

The intersection of the LITERAL FIELD and UNUSED_BIT_SET is identically the LITERAL_FIELD; no problem. The UNUSED_BIT_SET is then updated to reflect that Bits-29 thru 44 are now in use. The next instruction will cause a value to be placed in the ADDRESS FIELD. The intersection of the ADDRESS FIELD with the updated UNUSED_BIT_SET is not identically the ADDRESS FIELD. When this situation occurs, a warning is issued by CHAML.

If the intersection of the field's opcode set and a set representing the remaining operation code possibilities for the current instruction is empty, an improper field has been used. In the previous example, if the LITERAL FIELD was not meaningful in the context of a 'branch' operation CHAML would respond with a warning.

**Parallelism**

Horizontal microarchitectures allow many micro-operations per micro-instruction. At the microprogramming language level, this introduces the need for additional language constructs which will delineate groupings of micro-operations. Dasgupta [1] suggests use of the 'COBEGIN'- 'COEND' pair. This method can lead to confusion if one of the micro-operations is a conditional test. The alternative taken by CHAML was to introduce the keyword 'CLK'. It has the effect of a 'super-semicolon' indicating the end of the current micro-instruction. The following is an example of its use:

\[
\text{CLK; }
\text{THIS_REGISTER := THAT_REGISTER; }
\text{ITS_SUM := ONE_SUM + ANOTHER_SUM; }
\text{IF CARRY_SET THEN }
\text{BEGIN }
\text{CLK; }
\text{DO_THIS; }
\text{DO_THAT; }
\text{END; }
\]

The underlying problem is, of course, that code is written and presumably read in a serial, top down fashion whereas, the horizontal microcode is executed in a parallel, chunk at a time manner.

**Multi-level Pipelines**

Pipelining is a technique used by many microarchitectures to improve performance. Several types of pipelines exist. Single level pipelines occur most frequently. In a single level pipeline, one instruction is being executed while the next one is being fetched. If a condition is met in the instruction being executed, and a branch must take place, this type of pipeline will waste a cycle. There is no impact at the language level. Multiple level pipelines include a variety of designs. One is basically the single level approach except that the cycle is not wasted. Other designs allow for intermediate levels in which certain specific interpretations of the micro-instruction take place. Both of these influence language development. Consider the simple example:

\[
(1) \text{GOTO LABEL1; }
(2) \text{GOTO LABEL2; }
\]

A single level pipeline would never execute statement (2). In all probability, statement (2) would be optimised out of existence. A two level pipeline would execute statement (1), execute the first statement at LABEL1, then it would be abruptly jerked back, by the pipeline, to execute statement (2). CHAML provides a prefacing keyword 'P' (e.g., P GOTO), which, if applicable, indicates that the microprogrammer intends to make use of the pipeline. It would be the translators responsibility to inject a 'No-op' between the two statements, in the example, should the keyword be absent, so
that the code would execute as visually interpreted.

Results

CHAMIL exists and is functional. It represents significant improvements in the areas of algorithmic expression, readability, and functionality. Indications are that maintainability will be much improved over earlier methods. Performance requirements were maintained, although sometimes at the expense of established structured programming practices. CHAMIL has been received favorably by the internal user community and is being considered by other processor development groups as a possible tool.

Future enhancements include extensions to the data definition facility and the control repertoire in an attempt to generalize the language. Generalization of data definition implies a more sophisticated type checking mechanism. Reorganization of translator functions and tables are being examined to provide a more efficient interface for specification of a wider variety of micro-architectures.

Conclusions

Target machine development has become a highly refined art. Reported research results in this area pervade the literature. The topic of microprogramming languages, the tools used in building those target machines, remains inadequately discussed in the open literature. As late as the Micro-11 microprogramming workshop (Nov. 1978), explicit treatment of actual results in the microprogramming language domain was noticeably absent; exceptions being the papers by Dasgupta [1] and Dembinski and Budkowski [2]. Existing, structured high level languages, such as Pascal, are candidates for examination in terms of microprogramming applicability. Non-symbiotic relationships between programming and microprogramming environments, and intricacies involving horizontal micro-architectures pose particular problems in language design and translator development. CHAMIL was an attempt to raise the local level of microprogrammability by addressing each of these topics. In many respects it was a feasibility study, producing a functional translator which successfully met its design criteria.

References and Bibliography


