DEVELOPMENT OF A "HIGHER-LEVEL" MICROPROGRAMMING LANGUAGE USING A COMPILER-COMPILER

John A. Lowry
ITT Defense Communications Division
San Diego, CA 92131

Michael J. Viehman
Linkabit Corporation
San Diego, CA 92121

Abstract
A non-traditional microprogramming language and its development using a compiler-compiler are described. An idea of the synergism that can develop between more user-oriented microprogramming tools, well-organized hardware architecture, and compiler-compiler syntax structure is presented. Considering the characteristics of the resulting product, the effort involved in this implementation methodology seems to have been well-spent.

Introduction
ITT Defense Communications Division has developed a low-power signal processing computer using bit-slice technology. The low-power constraints of the hardware specifications resulted in a machine architecture programmable at the microinstruction level only. A seemingly conflicting goal of providing a readable programming language to be used by the programming staff was also present. The approach taken in the language implementation effort was to use a compiler-compiler, a tool that takes as its input a BNF-like description of the language and provides as its output parse tables for the language specified. By using such a tool, it was found that it was possible to develop a language that is modularly constructed, easy to maintain, and reasonably high-level. It was also found that well-organized hardware architecture was reflected in the input to the compiler-compiler. Thus, by using modern language development tools and having a clean target machine for the language, it was possible to develop a higher-level "more modern" microprogramming language.

Microprogramming Languages
In the past, various constraints have caused conventional microprogramming languages, with the noted exception of those of the Burroughs Corporation [7], to look like computer design aids rather than user-oriented algorithmic descriptions of solutions to problems. Consider Figure 1, showing a microprogramming statement for the IBM 360/50 [3], as opposed to the program example in Figure 2 of the authors' higher-level microprogramming language.

Although there is a seemingly vertical structure in the statements presented in Figure 2, the syntax hides the complexity that can be expressed and precisely controlled with the language. The sources and destinations of the signal processor's three internal buses, as well as parallel condition testing and operand dependent operator selection within the arithmetic and logic unit (ALU), can be concisely expressed.

The machine's 16 registers are denoted R0-R15; the memory address and data registers are MAR and MDR, respectively; memory output gated to the INBUS is MI; the program counter is PC; and branches are accomplished by the keyword BR followed by an ALU condition code in parentheses.

Given these hints, reviewing the program in Figure 2 is instructive because, even without knowledge of the machine architecture (yet to be discussed), the semantics of the program statements are evident from the code. To be specific, the instruction at bcs6,

\[ r0 = mar = r0 + 1, \text{ br}(u) \text{ bcs10}; \]

increments register 0 while assigning that result to both register 0 and the memory address register, while at the same time an unconditional branch to label bcs10 is performed. Worth noting is the use of the equal sign for assignment, which seems much clearer than the syntax of a corresponding move instruction. The use of multiple equal signs in the statement is a construct reminiscent of the high-level "C" programming language [6].
Figure 1. 360/50 Microprogramming Statement
**Figure 2. Coding Example (adapted from B. Jarrett)**

```
bcsl0 r0 = mar-r0+1, br(u) bcsl10;
bcs8 mar = r36*r4 = m1;
mdr = r5 xor mi;
pc = r36*r3 = r3+2;
bcsl0 r2 = r5-236;
br (neg) bcsl2;
r5 = r5-r6;
bcsl2 r7 = r7+1;
r3 = r5+q, br(nz) bcsl2;
r2 = 0;
r0 = table;
a = r2+r0;

'branch to test for end of inner loop
'mar = s(j), r4 = m(ij)
's(j) = s(j) xor m(ij)
'indexed branch -- just a bit crude
'set up to test for
'end of inner loop
'if more branch to beginning
'begin set up for loop to fill inverse
'table
```

**Figure 3. Partial Processor Block Diagram**

**Figure 4. Output Format**

<table>
<thead>
<tr>
<th>0 0 000 H H 0 0 0 0 HHHH</th>
<th>address/data field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 000</td>
<td>outbus device</td>
</tr>
<tr>
<td>0 0 000</td>
<td>alu destination</td>
</tr>
<tr>
<td>0 0 000</td>
<td>inbus source</td>
</tr>
<tr>
<td>0 0 000</td>
<td>carry control</td>
</tr>
<tr>
<td>0 0 000</td>
<td>b register select</td>
</tr>
<tr>
<td>0 0 000</td>
<td>a register select</td>
</tr>
<tr>
<td>0 0 000</td>
<td>2901 op code</td>
</tr>
<tr>
<td>0 0 000</td>
<td>status change inhibit bit</td>
</tr>
<tr>
<td>0 0 000</td>
<td>op code</td>
</tr>
</tbody>
</table>

H: hex field
O: octal field
Hardware Structure

A low electrical power dedicated application signal processor with a fast instruction rate is achieved by eliminating all hardware support for any programming above the microprogramming level. A single level pipeline permits utilization of moderately slow program memory while only requiring programmers to consider branches as being effective at the end of the current instruction based on the results of the previous (or previously gated) instruction.

Figure 3 is a block diagram of a portion of the signal processor. The three buses in the places where are the INBUS, the direct data inputs to the 2901 ALU [1]; the OUTBUS, the destinations for the memory or ALU outputs; and the ALU OUTBUS, the link between the ALU and the address or data registers or the output bus.

Figure 4 summarizes the object code formats for the signal processor. From these, it is possible to see some of the synergism that developed between the hardware architecture and the syntax of the programming language. During any given machine cycle, a source may be specified to be placed on the INBUS. This may be some literal value from microprogram memory (if this instruction is not a branch), some input device such as an analog to digital converter, or one of the multiplier output registers. During the same cycle, the output provided by the 2901 ALU can be clocked into an output device, the input registers of the multiplier, the program counter, or the input bus.

Language Design

Figure 5 is a BNF-like description of part of the syntax of the microprogramming language, taken from input to the Yacc [4] compiler-compiler. Yacc generates parsing tables from an LR1 grammar specification [2]. Words in capital letters represent terminal symbols; those in lower case nonterminal symbols. A lexical analyzer passes tokens to the parser representing the terminal symbols as it recognizes them in the input stream.

If the sequence of tokens passed by the lexical analyzer matches a sequence found in the grammar, control is passed to that point in the grammar where code generation routines (not shown) can be invoked. For example, under "bstat", the register recognized as the third token will be assigned to the "a select" field of the micro-instruction word. The "b select" field is open for one other register, which will be recognized in "stat", if it is used.

While the use of a compiler-compiler provides the language designer very good tools for specifying the syntax of a language, no tools for implementing code generation are directly provided. Hence, only a non-code generating syntax checker can be developed quickly, which was in fact the course taken. The syntax checker was used by the programmers on the order of a month after language design commenced; code generation was completed in concert with the hardware development.

The use of the compiler-compiler provided two other benefits which were side effects of the fact that the language was specified by a grammar and the somewhat enhanced parser we were using (courtesy of K. Davis). First, syntax error messages are a list of the tokens recognized as part of an acceptable token sequence, followed by the offending, unrecognized token in brackets. Second, for debugging purposes, it is possible to tell the microassembler to list every reduction in the parsing process. This is very useful for debugging the code generation routines, since it is immediately clear what routines were called to generate code for a particular instruction. Figure 6 shows a slightly edited expansion for the second statement of Figure 2. The terminal symbols passed to the parser are capitalized or in quotation marks; the reduced nonterminals are in lower case.

Language Description

The grammar fragment presented in Figure 5 defines permissible constructs for one line of microcode. Other parts of the grammar define constructs common to most assembly language: BSECTS, DSECTS, entry statements, and how the lines of code are glued together. They are not included as they are not considered relevant to the issue of language/machine synergism.

"Label stat" is defined in terms of "xstat", which reflects the fact that the status change inhibit bit may be set independently of the rest of the hardware. Hence, one may precede any statement with "INHIBIT".

Since it is possible to set one of the ALU bus destinations (for example, the MAR) to the contents of the A-register independently of the ALU operations, the microprogrammer may insert phrases of the
Figure 5. Grammar Fragment -- Permissible Constructs for One Line of Microcode
mar = r3 & r4 = mi 

Figure 6. Code Line and Expanded Parser Output
form "MAR = R10 &" before the rest of the statement, as shown in the expansion of "bstat". The separate ALUBUS makes this independent assignment possible. This feature of the hardware caused a fair amount of trouble in the language design. In the rest of the language, independent data transfers are separated by commas. The ampersand was used in the "bstat" definition because a comma there did not result in an LR1 grammar. Yacc provides tools for dealing with ambiguous grammars, but their nature can be, in the words of the Yacc reference manual, a "black art" [5]. Similarly, it is evident from the grammar fragment for "astat" that one may assign memory to an outbus destination or specify a branch or call statement, but not both. This reflects the fact that the data field of the microprogram word is used for both these purposes.

Finally, "stat" reflects the operations of the 2901 ALU and its interactions with the devices on the OUTBUS and the ALU outbus. The expansion of "alu exp" is rather lengthy and not included to conserve space. Suffice it to say that "alu exp" defines what can be done with the 2901 bit slices at their lowest level; i.e., up to two registers or the direct data input may be manipulated arithmetically or logically and gated out or back into the 2901 (or both) according to the specifications of the 2901. Figure 2 shows several possibilities.

References


