The Language PICL and its Implementation


1. Introduction

PICL is a small, experimental language for the PIC single-chip microcomputer. The class of computers which PIC represents is characterized by a wordlength of 8, a small set of simple instructions, a small memory of at most 1K cells for data and equally much for the program, and by integrated ports for input and output. They are typically used for small programs for control or data acquisition systems, also called embedded systems. Their programs are mostly permanent and do not change.

All these factors call for programming with utmost economy. The general belief is that therefore programming in a high-level language is out of the question. Engineers wish to be in total control of a program, and therefore shy away from complex languages and compiler generating code that often is unpredictable and/or obscure.

We much sympathize with this reservation and precaution, particularly in view of the overwhelming size and complexity of common languages and their compilers. We therefore decided to investigate, whether or not a language could be designed in such a way that the reservations would be unjustified, and the language would indeed be beneficial for programmers even of tiny systems.

We chose the PIC processor, because it is widely used, features the typical limitations of such single-chip systems, and seems to have been designed without consideration of high-level language application. The experiment therefore appeared as a challenge to both language design and implementation.

The requirements for such a language were that it should be small and regular, structured and not verbose, yet reflecting the characteristics of the underlying processor. In order to understand the challenge of bridging the gap between high-level abstractions and the concrete architecture, we must first obtain a picture of the processor, reduced to its essentials.

2. The Architecture of the PIC processor

The PIC processor is a typical Harvard architecture, i.e. a von Neumann machine with separate memories for program and data. In this experiment, we used the PIC 16C84, which uses an internal RAM for 64 bytes of data, and an EEPROM for 2k words of program. The first 12 bytes of data memory have special functions. They are the status register, a timer, input/output ports, etc. There is only one true register, the W-Register (not part of the RAM), which acts as an accumulator in the ALU, and on which data instructions operate. In the following diagram (see Fig. 1) we omit the “registers” with special functions.

There is a rather small instruction set with 4 formats for

1. Byte-oriented instructions consisting of opcode and operand address:
   
   MOV, ADD, SUB, AND, IOR, XOR,
   DEC, INC, DECFSZ, INCFSZ (increment/decrement and skip if result is zero)

2. Byte-oriented instructions consisting of opcode and literal operand:
   
   MOV, ADD, SUB, AND, IOR, XOR,
   GOTO, CALL, RETURN

3. Bit-oriented instructions consisting of opcode, operand address, and bit number:
   
   BFS, BFC (set/clear bit)
   BTFSC, BTFSS (bit test, skip if clear/set)
4. Jump instructions with an 11-bit absolute address.
Addresses are only 7 bits long, bit numbers range from 0 to 7 (see Fig. 2).

Fig. 1 The PIC architecture

<table>
<thead>
<tr>
<th>4</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>opcode</td>
<td>d address</td>
</tr>
<tr>
<td>11</td>
<td>opcode</td>
<td>literal</td>
</tr>
<tr>
<td>01</td>
<td>op</td>
<td>bit address</td>
</tr>
<tr>
<td>10x</td>
<td></td>
<td>address</td>
</tr>
</tbody>
</table>

byte-oriented data instr.
byte-oriented instr. with literal
bit-oriented data instr.
goto / call

Fig. 2. PIC instruction formats

3. The Language PICL

The language is concisely defined in a separate report. Here we merely point out its particular characteristics which distinguish it from conventional languages. Like conventional languages, however, it consist of constant, variable, and procedure declarations, followed by statements of various forms. The simplest forms, again like in conventional languages, are the assignment and the procedure call. Assignments consist of a destination variable and an expression. The latter is restricted to be a variable, a constant, or an operator and its operand pair. No concatenation of operations and no parentheses are provided. This is in due consideration of the PIC's simple facilities and ALU architecture. Examples can be found in the section on code patterns below.

Conditional and repetitive statements are given the modern forms suggested by E. W. Dijkstra. They may appear as somewhat cryptic. However, given the small sizes of programs, this seemed to be appropriate.
Conditional statements have the form shown at the left and explained in terms of conventional notation to the right.

\[
\begin{align*}
&[\text{cond} \rightarrow \text{StatSeq}] & \text{IF cond THEN Statseq END} \\
&[\text{cond} \rightarrow \text{StatSeq0} * \text{StatSeq1}] & \text{IF cond THEN Statseq0 ELSE StatSeq1 END} \\
&[\text{cond0} \rightarrow \text{StatSeq0} | \text{cond1} \rightarrow \text{StatSeq1}] & \text{IF cond0 THEN Statseq0 ELSIF cond1 THEN StatSeq1END}
\end{align*}
\]

Repetitive statements have the form:

\[
\begin{align*}
&\{\text{cond} \rightarrow \text{StatSeq}\} \text{ WHILE cond DO Statseq END} \\
&\{\text{cond0} \rightarrow \text{StatSeq0} | \text{cond1} \rightarrow \text{StatSeq1}\} \text{ WHILE cond0 DO Statseq0 ELSIF cond1 DO StatSeq1END}
\end{align*}
\]

There is also the special case mirroring a restricted form of for statement. Details will be explained in the section on code patterns below.

\[
\{\text{ident, xpression} \rightarrow \text{StatSeq}\}
\]

Procedures can have at most a single (value) parameter. They can be functions with a result that can be assigned to a variable. Recursion is not allowed, and the depth of calls can be at most 8. These restrictions are a direct consequence of architectural limitations and our effort to do without complicated, hidden mechanisms, such as a call stack, local variables, etc. Whereas the syntax of PICL is to provide the conveniences of high-level languages, its semantics are to mirror the facilities and limitations of the processor clearly and honestly.

4. The PICL Compiler

The compiler consists of two modules, the scanner, and the parser and code generator. The scanner recognizes symbols in the source text. The parser uses the straight-forward method of syntax analysis by recursive descent. It maintains a linear list of declared identifiers for constants, variables, and procedures.

5. Code Patterns

In order to exhibit the correspondence between language constructs and assembler code, a sequence of short samples is listed, followed by the code generated by the compiler.

**MODULE Assignments;**

\[
\begin{align*}
&\text{CONST N = 10;} \\
&\text{INT x, z;} \\
&\text{BEGIN z := x; z := N; z := 0;} \\
&\text{END Assignments.}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000080C</td>
<td>MOVFW 0</td>
<td>12</td>
<td>move x to W</td>
</tr>
<tr>
<td>1 0000080D</td>
<td>MOVWF 1</td>
<td>13</td>
<td>move W to z</td>
</tr>
<tr>
<td>2 0000300A</td>
<td>MOVLW 10</td>
<td>move 10 to W</td>
<td></td>
</tr>
<tr>
<td>3 0000080D</td>
<td>MOVWF 1</td>
<td>13</td>
<td>move W to z</td>
</tr>
<tr>
<td>4 0000018D</td>
<td>CLRF 1</td>
<td>13</td>
<td>z := 0</td>
</tr>
</tbody>
</table>

**MODULE Operators;**

\[
\begin{align*}
&\text{BOOL b; SET s; INT x;} \\
&\text{BEGIN !b; !~s.3;} \\
&\text{INC x; DEC x; ROL x; ROR x} \\
&\text{END Operators.}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000140C</td>
<td>BSF 0</td>
<td>12</td>
<td>!b set b.0</td>
</tr>
<tr>
<td>0 00001018</td>
<td>BCF 3</td>
<td>13</td>
<td>!~s.3 clear s.3</td>
</tr>
<tr>
<td>2 00000A8E</td>
<td>INCF 1</td>
<td>14</td>
<td>INC x</td>
</tr>
<tr>
<td>3 0000038E</td>
<td>DECF 1</td>
<td>14</td>
<td>DEC z</td>
</tr>
<tr>
<td>4 00000D8E</td>
<td>RLF 1</td>
<td>14</td>
<td>rotate x left via carry (S.0)</td>
</tr>
<tr>
<td>5 00000C8E</td>
<td>RRF 1</td>
<td>14</td>
<td>rotate x right via carry (S.0)</td>
</tr>
</tbody>
</table>

Statements operating on a single operand are called *operators*. They are denoted by an exclamation mark, and correspond to a single instruction.

Statements testing a bit and waiting until the bit is set or reset are called *queries*. They are denoted by a question mark, and they are applied to elements of input ports A and B.
```plaintext
MODULE Queries;
BEGIN ?A; ?~B.3
END Queries.

0 00001C05 BTFSS 0 5 ?A wait until A.0 true
1 00002800 GOTO 0
2 00001986 BTFSC 3 6 ?~B.3 wait until B.3 false
3 00002802 GOTO 2

Expressions have the simple form \( x \ op \ y \). Both operands must be of the same type. The type determines the operation. For example, + for integers denoted addition, + for sets denotes logical or. If the result is assigned to the first operand, the compiler makes use of the possibility that the result of an instruction may be written to the operand instead of the W-register. This saves one instruction.

```plaintext
MODULE Expressions;
INT x, y, z; SET u, v, w;
BEGIN z := x+3; z := y-3; z := x+y; x := x+y; z := 15-x;
w := u + $07; w := u * $0F; w := u - v; u := u - v
END Expressions.

```plaintext
0 00003003 MOVLW 3
1 0000070C ADDWF 0 12
2 0000008E MOVWF 1 14 z := x+3
3 00003003 MOVLW 3
4 0000020D SUBWF 0 13
5 0000008E MOVWF 1 14 z := y-3
6 0000080D MOVF W 0 13
7 0000070C ADDWF 0 12
8 0000008E MOVWF 1 14 z := x+y
9 0000080D MOVF W 0 13
10 0000078C ADDWF 1 12 x := x+y
11 0000080C MOVFW 0 12
12 000003C0F SUBLW 15
13 00000008E MOVWF 1 14 z := 15-x
14 000003007 MOVLW 7
15 00000040F IORWF 0 15
16 00000091 MOVWF 1 17 w := u + $07
17 00000300F MOVLW 15
18 00000050F ANDWF 0 15
19 00000091 MOVWF 1 17 w := u * $0F
20 000000810 MOVFW 0 16
21 0000060F XORWF 0 15
22 00000091 MOVWF 1 17 w := u - v
23 00000810 MOVFW 0 16
24 0000068F XORWF 1 15 u := u - v

Conditions yield a truth value. They consist of comparisons and bit tests concatenated by either logical disjunctions (or), or by conjunctions (and). Here, the conditions are part of if statements of the form \( \text{IF } \text{cond} \ \text{THEN } \text{statement} \ \text{END} \).

```plaintext
MODULE Conditions;
INT x, y, z, w; SET s; BOOL b;
BEGIN
IF x = y THEN z := 0 END;
IF x = y & z & y # z THEN z := 0 END;
IF x < y OR y <= z OR z > w THEN z := 0 END;
END Conditions.

```plaintext
0 0000080D MOVFW 0 13 y
1 0000020C SUBWF 0 12 x - y
2 00001D03 BTFSS 2 3 = 07 (test S.3)
3 00002805 GOTO 5
4 0000018E CLRF 1 14 z := 0
5 0000080D MOVFW 0 13
6 0000020C SUBWF 0 12 x - y
```
Statements preceded by an if clause are called **guarded** statements. They are executed only if the guard is true.

**MODULE IfStatements**;

```plaintext
INT x, BOOL p, q;
BEGIN
  IF p THEN x := 0-x END;
  IF p THEN x := 1 ELSIF q THEN x := 2 END;
  IF p THEN x := 3 ELSIF q THEN x := 4 ELSE x := 5 END
END IfStatements.
```

While statements are sequences of guarded statements separated by "|" and enclosed in braces.

**MODULE WhileStatements**;
INT x, y, z; BOOL b;
BEGIN
  WHILE x ≠ 0 DO z := z + y; x := x - 1 END;
  WHILE x = y & ~b DO !b END;
  WHILE x ≥ y OR b DO !~b END;
END WhileStatements.

Repeat statement have their test for termination at the end and are therefore executed at least once. There is only one goto instruction jumping backward to the beginning of the repeat statement.

The compiler recognizes the special case, where the statement ends by decrementing a variable and then testing it for zero, as is shown by the second statement in the preceding example. In this case, subtraction, and test with skip are contractable into a single instruction DECFSZ (decrement and skip if zero). This case is recognized, however, only if decrementing is done by the DEC operator.

Procedures may have a single parameter, which is passed via the W-register, and they may have a result, which is also passed via the W-register.
PROCEDURE NofBits(INT x): INT;
    INT cnt, n;
    BEGIN cnt := 0; n := 8;
    REPEAT
        IF x.0 THEN INC cnt END ;
        ROR x; DEC n
    UNTIL n = 0;
    RETURN cnt
    END NofBits;

PROCEDURE Swap;
    INT z;
    BEGIN z := x; x := y; y := z
    END Swap;

PROCEDURE P(INT a);
    BEGIN
        x := a + 10
    END P;

BEGIN Swap; P(y); x := NofBits(y)
END Procedures.

6. Applications

The following two procedures show how to use PIC facilities to implement multiplication and division (of 8-bit non-negative integers).

PROCEDURE Multiply:
    INT x, y, z, n;
    BEGIN z := 0; n := 8;
    REPEAT
        IF x.0 THEN z := z+y END ;
        ROR z; ROR x; DEC n
    UNTIL n = 0;
    RETURN z
    END Multiply;
UNTIL n = 0
END Multiply.

zh,z := x*y  16-bit product double length register

 PROCEDURE Divide;
  INT r, q, d, n;
  BEGIN r := 0; n := 8;
    REPEAT ROL q; ROL r;
      IF r >= d THEN r := r - d; INC q END
    DEC n
    UNTIL n = 0
  END Divide.

q := r DIV d; r := r MOD d;  r,q form a double length register

The following procedures serve for sending and receiving a byte. Transmission occurs over a 3-wire connection, using the conventional hand-shake protocol. Port A.3 is an output. It serves for signaling a request to receive a bit. Port B.6 is an input and serves for transmit tthe data. B.7 is usually in the receiving mode and switched to output only when a byte is to be sent. In the idle state, both request and acknowledge signals are high (1).

PROCEDURE Send(INT x);
  INT n;
  BEGIN ?B.6; wait for ack = 1
    !S.5; !~B.7; !~S.5; n := 8; switch B.7 to output
    REPEAT
      IF x.0 -> !B.7 ELSE !~B.7 END ; apply data
Another version of the same procedures also uses three lines. But it is asymmetric: There is a master and a slave. The clock is always delivered by the master on B.6 independent of the direction of the data transmission on A3 and B7.

When sending, the data is applied to A.3, when receiving, the data is on B.7. The advantage of this scheme is that no line ever switches its direction, the disadvantage is its dependence on the relative speeds of the two partners. The clock must be sufficiently slow so that the slave may follow. There is no acknowledgement.

7. Conclusions

The motivation behind this experiment in language design and implementation had been the question: Are high-level languages truly inappropriate for very small computers? The answer is: Not really, if the language is designed in consideration of the stringent limitations. I justify my answer out of the experience made in using the language for some small sample programs. The corresponding assembler code is rather long, and it is not readily understandable. Convincing
oneself of its correctness is rather tedious (and itself error-prone). In the new notation, it is not
easy either, but definitely easier due to the structure of the text.

In order to let the regularity of this notation stand out as its main characteristic, completeness was
sacrificed, that is, a few of the PIC's facilities were left out. For example, indirect addressing, or
adding multiple-byte values (adding with carry). Corresponding constructs can easily be added.

One might complain that this notation is rather cryptic too, almost like assembler code. However,
the command (!) and query (?) facilities are compact and useful, not just cryptic. Programs for
computers with 64 bytes of data and 2K of program storage are inherently short; their
descriptions should therefore not be longwinded. After my initial doubts, the new notation appears
as a definite improvement over conventional assembler code.

The compiler was written in the language Oberon. It consists of a scanner and a parser module of
2 and 4 pages of source code respectively (including the routines for loading and verifying the
generated code into the PIC's ROM). The parser uses the time-honored principle of top-down,
recursive descent. Parsing and code generation occur in a single pass.