6

ABSTRACT CODE

This chapter describes an instruction set for an ideal computer tailored to the programming language Edison. The compiler translates an Edison program into *Edison code* consisting of a sequence of these instructions.

On existing computers with different instruction sets the Edison code can be interpreted by a *kernel* written in microcode or assembly language. The kernel contains a small code piece for every Edison instruction.

This approach to code generation and execution has several advantages:

(1) It enables compiler writers to ignore the strange properties of existing computers and make code generation simple and systematic.

(2) It enables programmers to move Edison software to different computers by rewriting the kernel.

(3) It provides hardware designers with a precise specification of a new computer that can execute Edison code efficiently.

The usefulness of *portable code* has been demonstrated successfully by earlier software systems written in the languages Pascal and Concurrent Pascal.

In the following the Edison code is explained in detail.

6.1 VARIABLES

During the execution of an Edison program the code and variables used are kept in a store. The *store* named *st* is an array of words with consecutive addresses. In addition, a set of *index registers* called *b*, *s*, *t*, and *p* are used. These entities can be declared as follows in Edison:

```
array store [...] (int)
var st: store; b, s, t, p: int
```

During the execution of a procedure, a block of storage known as a *call instance* of the procedure is used to hold the following local entities (Fig. 6.1):

(1) A function variable (if the procedure is a function)

(2) The parameters of the procedure (if any)

(3) A context link described below

(4) A set of values to be assigned to the index registers when the procedure has been executed

(5) The variables of the procedure and of all modules declared within it

(6) Temporary results used during the execution of the procedure

The address of the context link is called the *base address* of the call instance.

In this figure (and in the subsequent ones) low and high addresses are at the top and bottom of the figure, respectively.

Within a call instance the parameters and variables of the procedure are

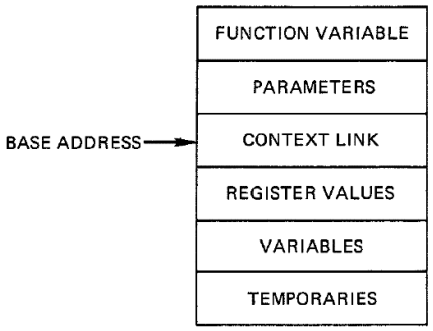

Fig. 6.1 Call instance

assigned store locations in their order of declaration from low toward high addresses.

With the exception of variable parameters, every *variable* occupies a fixed block of words that holds the current value of the variable. The address of the variable is the address of its first word.

A *variable parameter* occupies a single word that holds the address of the variable argument to which it is bound.

The store holds a *variable stack* for every process that is currently being executed. When a process calls a procedure, a call instance of that procedure is added on top of the variable stack of the process. And when the process has executed the procedure, the call instance is removed from the variable stack.

Consider, for example, a process described by the following program:

```
proc P1
var x1
  proc P2
  var x2
    proc P3
    var x3
    begin . . . end

    proc P4
    begin . . . P3 . . . end
  begin . . . P4 . . . end
begin . . . P2 . . . end
```

When the process has called the procedures P1, P2, P4, and P3 in that order and is executing P3, its variable stack contains a corresponding sequence of call instances (Fig. 6.2).

The *base address* of the most recent call instance created by a process (in this case, an instance of procedure P3) is kept in an index register named *b*.

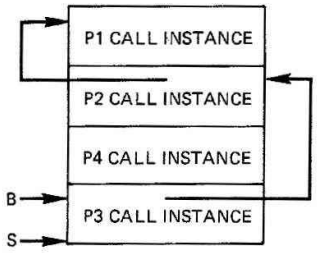

Fig. 6.2 Variable stack

The current top address of the variable stack of the process is held in another register named *s* (the *stack top*).

At any given moment a process can operate only on a subset of the call instances in its variable stack. This subset is called the *current context* of the process.

When a process is executing the procedure P3 in the example above, its context consists of a call instance of P3 and a call instance of each of the procedures P2 and P1 which surround P3 in the program text.

The call instance of the procedure P3 that is currently being executed is addressed by the base register b. This register points to a word, called the *context link*, which holds the base address of the call instance of the immediately surrounding procedure P2, which in turn points to the call instance of the next enclosing procedure P1.

In short, the call instances that form the current context of a process are linked together in their order of nesting from the currently executed procedure toward the procedures that surround it in the program text.

The dynamic creation and deletion of call instances (and the context links contained in them) are explained in Section 6.5. In the following we discuss the addressing of variables within a given context such as the one shown in Fig. 6.2.

A process selects a *whole variable* in the current context by executing two instructions named instance and variable:

```
instance(steps)
variable(displ)
```

Each *instruction* consists of an operation code followed by a single argument.

The execution of the instance instruction locates the call instance that includes the given variable by following the chain of context links a certain number of steps.

During the execution of procedure P3 in the previous example, a process would execute the instruction

```
instance(0)
```

to select a local variable x3, whereas the instruction

```
instance(1)
```

would be needed to locate a global variable x2 declared in the immediately surrounding procedure P2, and so on.

The execution of the instruction named variable completes the selection of a variable by adding the displacement of the variable to the base address of the call instance.

The operation codes and their arguments occupy one word each. This code representation is well suited to efficient interpretation by means of threaded code [Bell, 1973].

The address of the operation code that is currently being executed by a process is kept in a register named *p* (the *program index*) (Fig. 6.3).

The effect of executing an instruction will be described by an Edison procedure that has one parameter for each argument of the instruction, for example:

```
proc instance(steps: int)
var link, m: int
begin link := b;  m := steps;
  while m > 0 do
    link := st[link];  m := m - 1
  end;
  s := s + 1;  st[s] := link;  p := p + 2
end
```

An instance instruction computes the base address of a call instance in the current context and pushes it on the variable stack.

To obtain the address of a whole variable the instruction named variable adds a displacement to the base address on top of the stack:

```
proc variable(displ: int)
begin st[s] := st[s] + displ;  p := p + 2 end
```

A whole variable may be declared as a variable parameter. In that case, the execution of the instance and variable instructions will only compute the address of the parameter location. To obtain the address of the corresponding variable argument, an instruction named value is executed to replace the address of the parameter location by its value in the top of the stack. The algorithm for the value instruction will be described later.

This discussion of variable selection shows that the compiler may generate different instruction sequences for different instances of the same syntactic form of the Edison language.

The class of instruction sequences that corresponds to a syntactic form of the Edison language will be described by syntactic rules called *code rules*. The code rules are very similar to the syntactic rules of the language itself.

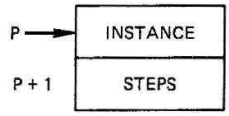

Fig. 6.3 An instruction

The basic symbols of the code rules are *instruction names* printed in boldface type.

The following rule describes the code generated for the selection of a whole variable discussed above:

Whole variable symbol:
instance variable [value]

This rule is to be understood as follows:

The code for a whole variable symbol consists of an instance instruction followed by a variable instruction and (possibly) followed by a value instruction.

The code that selects a whole variable may be followed by instructions that either select field variables or execute expressions and select indexed variables within the whole variable:

Variable symbol:
Whole variable symbol [**field** # Expression **index**]*

A field instruction adds the displacement of a field to the address of a record variable held on top of the variable stack:

```
proc field(displ: int)
begin st[s] := st[s] + displ; p := p + 2 end
```

An index instruction removes the value of an index expression from the top of the variable stack and computes the displacement of an element with a given length within an array value. The displacement is then added to the address of an array variable selected prior to the indexing:

```
proc index(lower, upper, length, lineno: int)
var i: int
begin i := st[s]; s := s - 1;
  if (i < lower) or (i > upper) do
    rangeerror(lineno)
  end;
  st[s] := st[s] + (i - lower) * length;
  p := p + 5
end
```

(Throughout this chapter the lengths of variables and temporaries are given in words.)

If the index value is outside the index range lower:upper, the program execution fails after reporting a range error.

The value of a selected variable is retrieved by executing an instruction named value:

Variable retrieval:
Variable symbol **value**

```
proc value(length: int)
var y, i: int
begin y := st[s]; i := 0;
  while i < length do
    st[s + i] := st[y + i]; i := i + 1
  end;
  s := s + length - 1; p := p + 2
end
```

A value instruction removes the address of a variable from the top of the variable stack and replaces it by the value of the variable.

6.2 CONSTANTS AND CONSTRUCTORS

Data values are stored as binary numerals known as stored values.

The stored value of an elementary value x is the ordinal value of x held in a single word.

An instruction named constant pushes a fixed elementary value on the variable stack:

```
proc constant(value: int)
begin s := s + 1; st[s] := value; p := p + 2 end
```

The evaluation of an elementary constructor involves evaluating an expression that leaves an elementary value on top of the variable stack:

Elementary constructor:
Expression

A stored record value is held in a block of words. The value consists of the fields stored in their order of declaration.

Record constructor:
Expression list

Expression list:
Expression [Expression] *

The evaluation of a record constructor involves evaluating a list of expressions to obtain the fields and leave a record value on the variable stack.

A stored array value is held in a block of words. The value consists of the elements stored in their order of indexing.

Array constructor:
Expression list [**blank**]

The evaluation of an array constructor involves evaluating a list of expressions to obtain the elements and leave an array value on the variable stack.

An abbreviated constructor for a string value includes an instruction named blank which adds the necessary number of spaces to the string value:

```
proc blank(number: int)
var i: int
begin i := 0;
  while i < number do
    i := i + 1; st[s + i] := int(' ')
  end;
  s := s + number;  p := p + 2
end
```

A stored set value is held in a block of fixed length (called the set length). Within this block the bits are numbered from 0 to an upper bound called the set limit. (The present compiler assumes a set length of 8 words and a set limit of 127.)

Each possible member x of a set value is represented by a single bit: bit number int (x). The bit has the value one if x is a member of the set value, and zero if it is not.

Set constructor:
[Expression list] **construct**

The evaluation of a set constructor involves evaluating a (possibly empty) list of expressions to obtain the members and then executing an instruction named construct to obtain a set value with the given members (if any).

```
set settype (int)

proc construct(number, lineno: int)
var member, i: int; new: settype
begin i := 0; new := settype;
  while i < number do
    member := st[s]; s := s - 1;
    if (member < 0) or (member > setlimit) do
      rangeerror(lineno)
    end;
    new := new + settype(member); i := i + 1
  end;
  storeset(s + 1, new);
  s := s + setlength; p := p + 3
end
```

The new set value is initially empty. The number of members given by the instruction are removed one at a time from the variable stack and are included in the set value, which is then finally pushed on the variable stack.

The copying of the new set value from a variable named new to a temporary location on top of the variable stack is performed by calling a procedure named storeset.

If a member is outside the base range 0:setlimit, the program execution fails after calling a procedure named range error that reports the failure and halts.

The following code rule summarizes the possible forms of constructors:

Constructor:
Elementary constructor # Record constructor #
Array constructor # Set constructor

6.3 EXPRESSIONS

The evaluation of an expression leaves a value on top of the variable stack according to the rules described in the following.

Factor:
constant # Constructor # Variable retrieval #
Function call # Expression # Factor **not**

A factor is evaluated either by executing a constant instruction, a variable retrieval, or a function call, or by evaluating a constructor, an expres-

sion, or another factor followed by a not instruction:

```
proc notx
begin st[s] := int(not bool(st[s])); p := p + 1 end
```

Term:
Factor # Term Factor Multiplying instruction
Multiplying instruction:
multiply # divide # modulo # and # intersection

A term is evaluated either by evaluating a factor or by applying a multiplying instruction to the values of another term and a factor:

```
proc multiply(lineno: int)
begin s := s - 1; st[s] := st[s] * st[s + 1];
  if overflow do rangeerror(lineno) end;
  p := p + 2
end
```

A boolean function named overflow determines whether the result of an arithmetic operation is within the range of integers; if not, the program execution fails after reporting a range error.

The divide and modulo instructions are similar to the multiply instruction, but use the operators **div** and **mod** instead of *.

```
proc andx
begin s := s - 1;
  st[s] := int(bool(st[s]) and bool(st[s + 1]));
  p := p + 1
end
```

The intersection instruction uses two operations named loadset and storeset to copy a set value from the stack to a local variable (and vice versa).

```
proc intersection
var x, y: settype
begin s := s - setlength; loadset(s + 1, y);
  loadset(s - setlength + 1, x);
  x := x * y;
  storeset(s - setlength + 1, x); p := p + 1
end
```

Simple expression:
Term [**minus**] #
Simple expression Term Adding instruction

Adding instruction:
add # subtract # or # union # difference

A simple expression is evaluated either by evaluating a term possibly followed by a minus instruction or by applying an adding instruction to the values of another simple expression and a term.

```
proc minus(lineno: int)
begin st[s] := - st[s];
  if overflow do rangeerror(lineno) end;
  p := p + 2
end
```

The add and subtract instructions are similar to the multiply instruction, but use the arithmetic operators + and - instead of *.

The or instruction is similar to the and instruction, but uses the operator **or** instead of **and.**

The union and difference instructions are similar to the intersection instruction, but use the set operators + and - instead of *.

Expression:
Simple expression
[Simple expression Relational instruction]
Relational instruction:
equal # notequal # less # notless #
greater # notgreater # in

An expression is evaluated either by evaluating a simple expression or by applying a relational instruction to the values of two simple expressions.

```
proc equal(length: int)
var y, i: int
begin y := s - length + 1;
  s := y - length;  i := 0;
  while (i < length - 1) and
    (st[s + i] = st[y + i]) do
      i := i + 1
  end;
  st[s] := int(st[s + i] = st[y + i]);
  p := p + 2
end
```

The instruction notequal is similar but uses the operator <> instead of = in the assignment to st[s].

```
proc less
begin s := s - 1;  st[s] := int(st[s] < st[s + 1]);
  p := p + 1
end
```

The instructions notless, greater, and notgreater are similar, but use the operators >=, >, and <= instead of <.

```
proc inx(lineno: int)
var x: int;  y: settype
begin s := s - setlength;  loadset(s + 1, y);
  x := st[s];
  if (x < 0) or (x > setlimit) do
    rangeerror(lineno)
  end;
  st[s] := int(x in y);  p := p + 2
end
```

6.4 SEQUENTIAL STATEMENTS

The code rules for statement lists and statements are:

Statement list:
    Statement [Statement] *
Statement:
    Empty # Assignment statement # Procedure call #
    If statement # While statement # When statement #
    Concurrent statement

An assignment involves the selection of a variable and the evaluation of an expression followed by the execution of an instruction named assign:

Assignment statement:
    Variable symbol Expression **assign**

```
proc assign(length: int)
var x, y, i: int
begin s := s - length - 1;  x := st[s + 1];
  y := s + 2;  i := 0;
  while i < length do
    st[x + i] := st[y + i];  i := i + 1
  end;
  p := p + 2
end
```

If statement:
    Conditional statement list
Conditional statement list:
    Conditional statement [Conditional statement]*
Conditional statement:
    Expression **do** Statement list **else**

The execution of an if statement involves executing a list of one or more conditional statements.

The execution of a conditional statement involves evaluating a boolean expression and executing an instruction named do: If the expression yields the value true, the statement list and an instruction named else are then executed; otherwise, the execution continues after the else instruction.

The do instruction removes a boolean value from the variable stack and jumps to a program address if the value is false. The program address is given by a displacement relative to the do instruction:

```
proc dox(displ: int)
begin
  if bool(st[s]) do p := p + 2
  else true do p := p + displ end;
  s := s - 1
end
```

The else instruction jumps to a program address given by a displacement:

```
proc elsex(displ: int)
begin p := p + displ end
```

The code of an if statement

```
if . . .
else B do SL
    . . .
end
```

includes the following program addresses L and M:

```
    . . .
    B do(L) SL else(M) L:
    . . .
M:
```

The execution of a while statement involves executing a conditional

statement list:

While statement:
Conditional statement list

The code of a while statement

```
while . . .
else B do SL
    . . .
end
```

includes the following program addresses L and M:

```
M: . . .
   B do(L) SL else(M) L:
   . . .
```

Procedure calls, when statement, and concurrent statements are described elsewhere in this chapter.

6.5 PROCEDURES AND MODULES

The addressing of variables in a given context was discussed in Section 6.1. The following describes how the context of a process changes during the execution of procedure calls.

Consider again the program example used in Section 6.1:

```
proc P1
  proc P2
    proc P3
    begin . . . end

    proc P4
    begin . . . P3 . . . end
  begin . . . P4 . . . end
begin . . . P2 . . . end
```

When a process has called the procedures P1, P2, and P4 in that order and is executing P4, the variable stack of the process contains a call instance of each of these procedures as shown in Fig. 6.4a.

The effect of calling procedure P3 within P4 is to add a call instance of P3 on top of the stack and link it to the call instances of the procedures P2 and P1 which surround P3 in the program text (Fig. 6.4b).

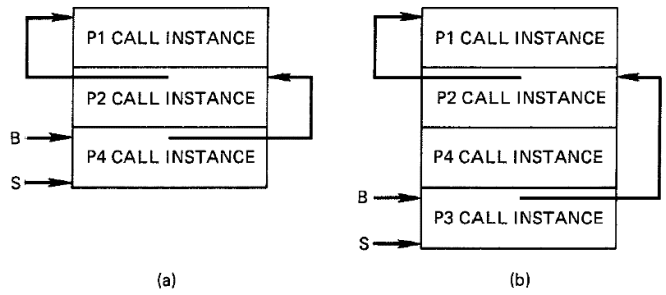

Fig. 6.4 Dynamic context change

When the execution of P3 terminates, the execution of the given process continues in the old context shown in Fig. 6.4a.

We will now describe the sequence of instructions executed when a process calls procedure P3:

(1) If the procedure is a function, its call instance must include a function variable that will hold the result of the call (see Fig. 6.1). The space for this variable is allocated on top of the variable stack by executing an instruction named valspace:

```
proc valspace(length: int)
begin s := s + length;  p := p + 2 end
```

(2) The parameters of the procedure called are created by executing the code for the arguments of the procedure call.

The code for an argument is the code for an expression, a variable symbol, or a procedure argument.

The evaluation of an expression allocates space for a value parameter on top of the stack and assigns the given value to it.

The evaluation of a variable symbol allocates space for a variable parameter on top of the stack and assigns the address of the given variable to it.

The evaluation of a procedure argument will be described later.

(3) The procedure P3 must be executed in the context of the procedures P2 and P1 which surround it in the program text. The execution of the instruction

```
instance(1)
```

locates the call instance of P2 in the old context which is valid before the procedure P3 is called (Fig. 6.4a). This instruction pushes the context link to P2 on top of the parameters in the variable stack.

(4) The jump to the code of the procedure P3 is performed by executing an instruction

proccall(displ)

When the process returns from the procedure, the index registers b, s, and t must be assigned the values they had prior to the procedure call. Figure 6.4a shows the return values of the registers b and s. The purpose of the t register will be described in Section 6.6. The return value of the program index p is the address of the instruction that follows the procedure call.

The proccall instruction pushes the return values of the index registers b, t, and p (but not s) on top of the stack, and assigns new values to b and s as shown in Fig. 6.5. Following this the program index is assigned the address of the procedure code. The code address is given by its displacement relative to the proccall instruction to make the code relocatable in the store.

```
proc proccall(displ: int)
begin st[s + 1] := b;  st[s + 3] := t;
  st[s + 4] := p + 2;  b := s;
  s := s + 4;  p := p + displ
end
```

(The return value of s is placed in the call instance during the execution of an instruction named procedure which will be described shortly.)

What has been described so far is the code generated for a procedure call (but not for a procedure body). This code is summarized by the fol-

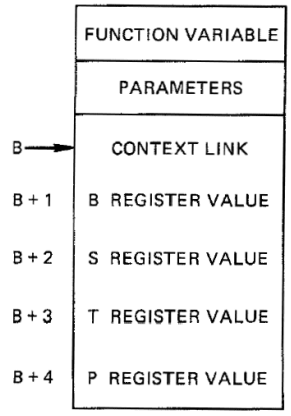

Fig. 6.5 Partial call instance

lowing rules:

Function call:
    valspace Procedure call
Procedure call:
    [Argument list] Procedure jump
Argument list:
    Argument [Argument]*
Argument:
    Expression # Variable symbol # Procedure argument
Procedure jump:
    instance proccall

The execution of the procedure body takes place as follows:
(5) The first instruction executed in the body is named procedure:

```
proc procedure(paramlength, varlength, templength,
  lineno: int)
begin st[b + 2] := b - paramlength - 1;
  s := s + varlength;
  if s + templength > stackmax do
    variablelimit(lineno)
  end;
  p := p + 5
end
```

The procedure instruction completes the call link with a return value of the stack top s that will cause the parameters to be removed from the variable stack when the procedure has been executed. Following this, the call instance is extended with space for the variables of the procedure, and it is checked that there is room in the variable stack for the temporaries used during execution of the procedure body.

The upper limit of the variable stack is given by a global value named stackmax. If this limit is exceeded, the program execution fails after calling a procedure named variable limit that reports the failure and halts.

(6) The modules (if any) declared within the procedure are initialized one at a time by executing the code of their initial operations.

(7) The statements of the procedure itself are executed one at a time.

(8) The last instruction executed in the procedure is named endproc:

```
proc endproc
begin p := st[b + 4]; t := st[b + 3];
  s := st[b + 2]; b := st[b + 1]
end
```

This instruction assigns the return values stored in the call instance of the procedure to the index registers. The effect of this is to remove the call instance (except for a possible function value) from the variable stack and jump to the code that follows the procedure call. This completes the execution of the procedure call.

The procedure code described above is summarized by the following rules:

Complete procedure declaration:
procedure [Declaration] * Statement list **endproc**
Declaration:
Procedure declaration # Module declaration # Empty
Module declaration:
[Declaration] * Statement list

A syntactic form of the language, such as a constant declaration, that is not mentioned in the code rules has an *empty code* sequence.

Several details have been ignored in the general discussion of procedure calls and procedure declarations:

(1) *Standard procedure calls* are compiled into system-dependent instructions that will not be described here. We will merely change the code rule for procedure calls as follows:

Procedure call:
[Argument list] Procedure jump #
Standard procedure call

(2) A *procedure parameter* is a parameter that is bound to another procedure, such as Q5 in the following program example:

```
proc Q1
  proc Q2
  begin . . . end

  proc Q3
    proc Q4(proc Q5)
    begin . . . Q5 . . . end
  begin . . . Q4(Q2) . . . end
begin . . . Q3 . . . end
```

When a process has called procedures Q1 and Q3 in that order and is executing Q3, its variable stack looks as shown in Fig. 6.6a.

Later, when the process has called procedures Q4 and Q5 (which is bound to procedure Q2), the stack will have changed as shown in Fig. 6.6b.

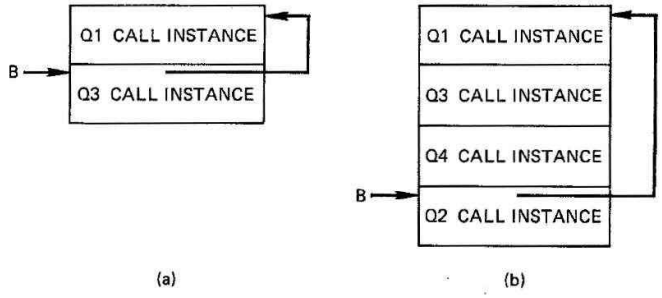

Fig. 6.6 Procedure parameter call

The procedure parameter Q5 is now executed in the context of the procedure Q1 which surrounds the argument Q2 in the program text.

This is achieved as follows: When procedure Q4 is called with procedure Q2 as an argument, the following instructions are executed in the old context shown in Fig. 6.6a:

```
instance(1)
procarg(displ)
```

The instance instruction computes the base address of the call instance of the procedure Q1 which surrounds the procedure argument Q2 and pushes it on the stack.

The instruction named procarg then pushes the code address of the procedure argument Q2 on top of the base address.

```
proc procarg(displ: int)
begin s := s + 1; st[s] := p + displ; p := p + 2 end
```

The code address is given by its displacement relative to the procarg instruction.

The execution of these two instructions creates the procedure parameter Q5 in the call instance of procedure Q4. The procedure parameter consists of two words defining the context link and code address to be used when the procedure parameter Q5 is called within Q4 (Fig. 6.7).

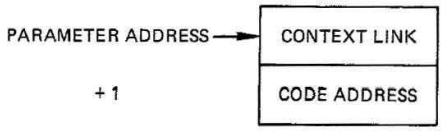

Fig. 6.7 Procedure parameter

When procedure Q4 calls its procedure parameter Q5 (bound to Q2), the following instructions are executed:

```
instance(0)
paramcall(displ)
```

The instance instruction locates the call instance of Q4, which includes the procedure parameter Q5. The displacement of the parameter within the call instance is given by the paramcall instruction. The latter pushes the context link held in the procedure parameter on top of the stack and creates a partial call instance of Q2 as shown in Fig. 6.6b. The call then proceeds as described earlier for the proccall instruction except that the code address of the procedure Q2 is also retrieved from the procedure parameter Q5:

```
proc paramcall(displ: int)
var addr: int
begin addr := st[s] + displ;
   st[s] := st[addr]; st[s + 1] := b;
   st[s + 3] := t; st[s + 4] := p + 2;
   b := s; s := s + 4; p := st[addr + 1]
end
```

To summarize the above we need a new rule describing the code for a procedure used as an argument in a procedure call:

Procedure argument:
instance procarg

and a previous rule must be extended to include the call of a procedure used as a parameter of another procedure:

Procedure jump:
instance proccall # instance paramcall

As the following example shows, a procedure Q3 may call another procedure Q1 and pass one of its own procedure parameters Q4 as an argument to Q1:

```
proc Q1(proc Q2)
begin . . . Q2 . . . end

proc Q3(proc Q4)
begin . . . Q1(Q4) . . . end
```

This use of a procedure parameter Q4 as a procedure argument in a call causes the following instructions to be executed:

```
instance(0)
paramarg(displ)
```

The instance instruction and the displacement of the paramarg instruction serve to locate the procedure parameter Q4 in the call instance of Q3.

The paramarg instruction creates the new procedure parameter Q2 by copying the context link and code address from the original parameter Q4 onto the top of the variable stack.

```
proc paramarg(displ: int)
var addr: int
begin addr := st[s] + displ;
   st[s] := st[addr];
   st[s + 1] := st[addr + 1];
   s := s + 1;  p := p + 2
end
```

Since a procedure used as an argument in a call can refer to either a procedure declaration or a procedure parameter, the code rule for procedure arguments must be extended as follows:

Procedure argument:
instance procarg # instance paramarg

(3) When the procedure instruction of a procedure P has been executed, the code of the local declarations of P is executed to initialize the modules declared within P.

During this initialization, the code of the local procedures of P must be bypassed. To achieve this, every procedure in a program (except the outermost one) is preceded by a goto instruction that causes a jump around the entire procedure code.

```
proc goto(displ: int)
begin p := p + displ end
```

Like all other jumps, this one is relative to the location of the jump instruction.

The goto instruction makes it necessary to revise the previous code rule for complete procedure declarations:

Complete procedure declaration:
[**goto**] **procedure** [Declaration] *
Statement list **endproc**

(4) The possible code sequences for procedure declarations are summarized as follows:

Procedure declaration:
Complete procedure declaration #
Library procedure declaration # Empty

Library procedure declarations are described in Section 6.6. Predeclarations of *split procedures* generate no code. The corresponding post declarations are compiled as complete procedure declarations.

6.6 PROGRAMS

In addition to the variable stack, the store holds a *program stack* for each process that is currently being executed.

Initially, the program stack contains the code of a single Edison program (called the operating system). When a process calls a library procedure, the code of the procedure is retrieved from a program library and is added to the program stack. And when the process has executed the procedure body, the corresponding library code is removed from the program stack (Fig. 6.8).

An index register named t holds the address of the current top of the program stack.

The code of an Edison program consists of a single word that defines the length of the compiled program followed by the code of a complete procedure plus a final instruction named endcode:

Program:
Program length Complete procedure declaration **endcode**

The execution of the program normally consists of executing the code of the procedure declaration. (The endcode instruction will be described later.)

When a program calls a library procedure declaration of the form

lib proc Procedure name (Parameter list)
[Expression]

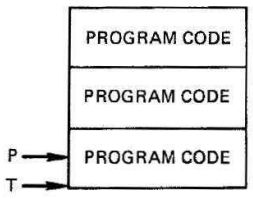

Fig. 6.8 Program stack

an expression is evaluated and used to locate a compiled program in a library and execute it.

The conventions for program loading are system dependent. The following assumes that a software system written in Edison includes a function named load that uses the name of a program to retrieve its code:

proc load(title: name): program

The data types used are declared as follows:

array name [1:namelength] (char)
array code [1:maxcode] (int)
record program (progname: name; progcode: code)

The program name is stored together with the code so that it can be displayed if the execution fails.

Using this convention, the expression in a library procedure declaration is simply a call of the load function, for example:

load(name('edit'))

The advantage of programming the load function in Edison is that the kernel only needs to know the structure of the program code but makes no assumption about the structure of the program library.

A library procedure declaration generates code of the following form:

Library procedure declaration:
goto libproc Expression **endlib**

The purpose of the goto instruction is to skip the library procedure declaration initially as explained earlier (Section 6.5).

When a program calls a library procedure, an instruction named libproc is executed immediately after the procedure jump:

```
proc libproc(paramlength, templength, lineno: int)
begin st[b + 2] := b - paramlength - 1;
  if s + templength > stackmax do
    variablelimit(lineno)
  end;
  p := p + 4
end
```

The instruction checks that there is sufficient space in the variable stack for the temporaries needed to evaluate the expression and completes a call link in the top of the variable stack with a return value of the stack top s.

The expression (assumed now to be a call of the load function) is then evaluated to place the name and code of a library procedure on top of the variable stack.

The instruction named endlib moves the library procedure from the variable stack to the program stack and changes the stack top s and the program top t accordingly:

```
proc endlib
begin move_program;  p := start_address end
```

Following this, the execution continues with the first instruction of the program.

As described earlier, a program is compiled as any other procedure beginning with an instruction named procedure and ending with an instruction named endproc (Section 6.5).

When a program P calls a library procedure, a call instance is created in the variable stack. The return value of the program index p is the address of the instruction that follows the call. The return value of the program top t is the value that this register has when the library procedure is being called but has not yet been loaded on the program stack. Consequently, when the library procedure eventually executes an endproc instruction, and thereby assigns the return values to the index registers, the library procedure is automatically removed from the program stack, and the execution continues in the previous program P following the call.

When the computer is started, a fixed Edison program is placed in the program stack and is executed. As all other programs, this operating system will eventually execute an endproc instruction. Since the operating system is called by the kernel (and not by another program), the following ad hoc rule is used: The variable stack is initialized with a call link that holds a dummy return address named none:

```
b := stackmin;  s := b + 4;  st[s] := none
```

The endproc instruction is then modified as follows:

```
proc endproc
begin
  if st[b + 4] <> none do as before
  else true do p := p + 1 end
end
```

If the execution of the operating system reaches the final endproc instruction, the execution proceeds with the following instruction, which is named endcode:

```
proc endcode(lineno: int)
```

This is a system-dependent instruction that reports the termination of the operating system and halts. Although all programs include this instruction, it can be reached only by the operating system.

6.7 CONCURRENT STATEMENTS

Initially, an Edison program is executed as a single process. When this initial process reaches a concurrent statement, it jumps to an instruction named cobegin:

Concurrent statement:
    goto Process statement list **cobegin**
Process statement list:
    Process statement [Process statement] *
Process statement:
    process Statement list **also**

The cobegin instruction creates a *tree-structured variable stack* with a branch for every process statement. Following this the process statements are executed simultaneously.

The stack of the initial process is the *root* of the stack tree. It holds the *common variables* of the processes. When a process calls one or more procedures, its stack branch is extended with local variables that are inaccessible to other processes. The context of a process consists of those call instances in the stack branch and root that are chained together by context links (Fig. 6.9).

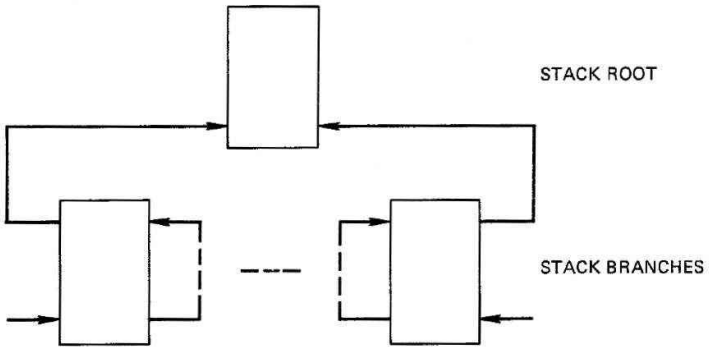

Fig. 6.9 Tree structured variable stack

The execution of a process statement takes place in three steps:

(1) An instruction named process checks that the stack branch of a process has sufficient space for the temporaries needed to execute a statement list.

(2) The statement list is executed.

(3) An instruction named also is executed. This removes a stack branch and terminates the corresponding process with one exception: The last process to reach an also instruction continues the execution after the cobegin instruction in the context of the stack root.

The code of a concurrent statement

```
cobegin . . .
also ci do SLi
    . . .
end
```

includes the following program addresses L1, . . . , Lm, M, N:

```
    goto(M) . . .
Li: process( ) SLi also(N)
    . . .
M:  cobegin(m, n, c1, L1, . . , cm, Lm)
N:
```

where the constant n is a line number.

A when statement generates the following code:

When statement:
when Synchronized statement **wait endwhen**
Synchronized statement:
Conditional statement list

The execution of the instruction named when delays a process until no other process is executing a synchronized statement.

If all the boolean expressions in the synchronized statement yield the value false, the instruction named wait is executed; otherwise, the instruction named endwhen is executed.

The wait instruction enables another process (if any) to execute a synchronized statement and jumps back to the when instruction.

The endwhen instruction enables another process (if any) to execute a synchronized statement.

The code of a when statement

```
when . . .
else B do SL
    . . .
end
```

includes the following program addresses L, M, and N:

```
M: when . . .
   B do(L) SL else(N) L:
   . . .
   wait(M)
N: endwhen
```

The instructions described above are system dependent and are explained further in the following section on Single Processor Systems.

6.8 SINGLE-PROCESSOR SYSTEMS

Initially, an Edison program is executed as a sequential process with a single variable stack and a single program stack.

Figure 6.10 shows the store of a single-processor system which holds a kernel and the two stacks. The stacks grow toward one another in the free space between them.

In this case it is most convenient to let the index register *t* hold the address of the first free word above the program stack. The value of this register can then also serve as the value *stackmax* that defines the maximum extent of the variable stack.

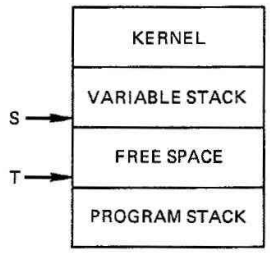

Fig. 6.10 Single-processor store

We will assume that a single-processor system does not permit concurrent processes to call library procedures. Consequently, the program stack does not change during the execution of a concurrent statement.

When a concurrent statement is reached, the free space is divided evenly among the processes. The variable stack is now tree structured as described earlier.

The processor now executes the processes in cyclical order. Each process runs until it reaches an also or a wait instruction. The register values of the other processes are stored in a queue declared as follows:

```
record procstate (bx, sx, tx, px: int)
array queue [1:maxproc] (procstate)
var q: queue; this, tasks: int
```

The queue index of the process that is currently being executed is named this, while the number of concurrent processes is named tasks.

The processor preempts and resumes the execution of a process as follows:

```
proc preempt
begin q[this] := procstate(b, s, t, p) end

proc resume
begin b := q[this].bx; s := q[this].sx;
  t := q[this].tx; p := q[this].px
end
```

When the processor is started, there is only one process:

```
this := 1; tasks := 1
```

A cobegin instruction defines the number of processes to be executed and includes a process constant ci and a program displacement Li for every process:

cobegin(m, n, c1, L1, . . . , cm, Lm)

The cobegin instruction saves the current values of the stack top s and the program top t in two kernel variables:

```
var stacktop, progtop: int
```

Following this, the queue is assigned the initial states of the m processes, and the first of these is then executed:

```
proc cobeginx(no, lineno: int; arg: arglist)
var length, i: int
begin tasks := no;
  if tasks > maxproc do processlimit(lineno) end;
  stacktop := s; progtop := t;
  length := (t - s) div tasks; i := 0;
  while i < tasks do
    i := i + 1; t := s + length;
    q[i] := procstate(b, s, t, p + arg[i].displ);
    s := t
  end;
  this := 1; resume
end
```

The first instruction executed by a process checks the size of the local stack space:

```
proc process(templength, lineno: int)
begin
  if s + templength > t do
    variablelimit(lineno)
  end;
  p := p + 3
end
```

The last instruction executed by a process does one of two things:

(1) If some other processes are not terminated yet, the given process is removed from the queue by compacting the remaining entries and resuming the execution of another process.

(2) If all other processes have terminated, the execution continues after the concurrent statement using the original stack top and program top.

```
proc alsox(displ: int)
begin
  if tasks > 1 do
    while this < tasks do
      q[this] := q[this + 1]; this := this + 1
    end;
    tasks := tasks - 1; this := 1; resume
  else true do
    s := stacktop; t := progtop; p := p + displ
  end
end
```

Since each process runs to completion or delay, mutual exclusion is automatically ensured during the execution of a synchronized statement. So the instructions named when and endwhen have no effect other than incrementing the program index p by one.

A wait instruction serves to preempt the current process and resume another process (if any). Later, when the preempted process is resumed again, it will jump back to a when instruction given by a program displacement:

```
proc wait(displ: int)
begin p := p + displ; preempt;
  this := this mod tasks + 1; resume
end
```

On a PDP 11 computer the Edison kernel occupies 1800 words in a store of 28 K words:

Error reporting	400 words
Program loading	200 words
Code interpreter	1200 words
Kernel	1800 words

The kernel may pass procedures written in assembly language as parameters to the initial Edison program, which may then pass them on to other programs. The reader is referred to the kernel listing for details (Chapter 8).

6.9 CODE SUMMARY

The following is a list of the code rules of the Edison language. Standard procedure calls are not described here since they are system dependent.

Complete procedure declaration:
    [**goto**] **procedure** [Declaration] *
        Statement list **endproc**
Library procedure declaration:
    goto libproc Expression **endlib**
Procedure declaration:
    Complete procedure declaration #
    Library procedure declaration # Empty
Module declaration:
    [Declaration] * Statement list
Declaration:
    Procedure declaration # Module declaration # Empty
Variable symbol:
    instance variable [**value**] [**field** # Expression **index**] *
Constructor:
    [Expression] * [**blank** # **construct**]
Factor:
    constant # Constructor # Variable symbol **value** #
    valspace Procedure call # Expression # Factor **not**
Multiplying instruction:
    multiply # **divide** # **modulo** # **and** # **intersection**
Term:
    Factor [Factor Multiplying instruction] *
Adding instruction:
    add # **subtract** # **or** # **union** # **difference**
Simple expression:
    Term [**minus**] [Term Adding instruction] *
Relational instruction:
    equal # **notequal** # **less** # **notless** #
    greater # **notgreater** # **in**
Expression:
    Simple expression
        [Simple expression Relational instruction]
Procedure argument:
    instance procarg # **instance paramarg**
Argument:
    Expression # Variable symbol # Procedure argument

Argument list:
    Argument [Argument] *
Procedure jump:
    instance proccall # instance paramcall
Procedure call:
    Standard procedure call #
    [Argument list] Procedure jump
Conditional statement:
    Expression **do** Statement list **else**
Conditional statement list:
    Conditional statement [Conditional statement] *
If statement:
    Conditional statement list
While statement:
    Conditional statement list
When statement:
    when Conditional statement list **wait endwhen**
Process statement:
    process Statement list **also**
Concurrent statement:
    goto Process statement [Process statement] * **cobegin**
Statement:
    Variable symbol Expression **assign** # Procedure call #
    If statement # While statement # When statement #
    Concurrent statement # Empty
Statement list:
    Statement [Statement] *
Program:
    Program length Complete procedure declaration **endcode**

6.10 CODE OPTIMIZATION

The code described so far is called *standard Edison code*. It has the great advantage of corresponding closely to the syntax of the programming language. This makes the code generation simple and systematic.

But in many cases the standard code is too large to fit into microcomputers with limited address spaces. The approximately 50 standard instructions are therefore supplemented with 15 *extra instructions* to reduce the code size.

The choice of extra instructions was guided by a series of experiments using pass 3 of the compiler as an example of a large Edison program.

Initially, the 1500 lines of pass 3 were compiled into 16200 words of standard code. By comparison, an earlier version of pass 3 written in Pascal was compiled into 9000 words of Pascal code.

The standard Edison code of pass 3 was scanned by a program to identify the instruction that made the single largest contribution to the code size.

The code was then studied in detail to find a special case of that instruction which occurred frequently enough to make it worthwhile to replace it with a new, shorter instruction.

After extending the compiler with the new instruction, the process of compilation, and code analysis was repeated to discover another instruction that would reduce the code size significantly.

These experiments were continued until the code reduction became marginal.

Originally, the standard code included a newline instruction of the form

```
newline(lineno)
```

to facilitate error reporting.

An initial analysis showed that no less than 17% of the code for pass 3 consisted of newline instructions. On the other hand, the only instructions that needed *line numbers* to report program failure accounted for less than 5% of the code size:

Procedure instructions	2.0%
Index instructions	1.4%
Construct instructions	0.9%
Arithmetic instructions	0.1%
Other instructions	0.2%

The newline instruction was therefore removed from the standard code and instead line numbers were included in the instructions above as described earlier in this chapter.

The encoding of line numbers in instructions eliminated 16% of the original code.

The next step was to replace a pair of instructions

```
instance(0) variable(displ)
```

which selects a *local variable* by a single, new instruction

```
localvar(displ)
```

Since the new instruction replaces a sequence of other instructions, it can be described by a (parameterized) syntactic rule:

```
Localvar(displ):
    instance(0) variable(displ)
```

The extra instructions can therefore be considered as syntactic forms recognized in the standard code.

The instruction named localvar is defined as follows:

```
proc localvar(displ: int)
begin s := s + 1; st[s] := b + displ; p := p + 2 end
```

Another frequent case is the retrieval of the elementary value of a local variable:

```
Localvalue(displ):
    Localvar(displ) value(1)
```

```
proc localvalue(displ: int)
begin s := s + 1; st[s] := st[b + displ]; p := p + 2 end
```

A conditional statement of the form

v = c **do** SL

where v is a local variable and c is a constant of the same elementary type generates the code

localcase(v, c, L) SL else( ) L:

The instruction named localcase is defined as follows:

```
Localcase(vardispl, value, progdispl):
    Localvalue(vardispl) constant(value)
        equal(1) do(progdispl)
```

```
proc localcase(vardispl, value, progdispl: int)
begin
  if st[b + vardispl] = value do p := p + 4
  else true do p := p + progdispl end
end
```

The instructions above refer to variables that are local to the procedure P that is currently being executed. A similar set of instructions are introduced for variables declared in the procedure that immediately surrounds P. These instructions, called outervar, outervalue, and outercase, are obtained from the previous ones by replacing the address expression b + displ by st[b] + displ.

When a procedure contains two local procedures P and Q, then Q may call P by executing an instruction named outercall:

Outercall(displ):
    instance(1) proccall(displ)

```
proc outercall(displ: int)
begin s := s + 1;  st[s] := st[b];
  st[s + 1] := b;  st[s + 3] := t;
  st[s + 4] := p + 2;  b := s;
  s := s + 4;  p := p + displ
end
```

If a call refers to a procedure parameter declared in the immediately surrounding procedure, it is compiled as a similar instruction named outerparam:

Outerparam(displ):
    instance(1) paramcall(displ)

Some other obvious instructions are:

Elemvalue:
    value(1)
Elemassign:
    assign(1)

The remaining instructions are described below:

Localset(displ):
    Localvar(displ) value(setlength)

```
proc localset(displ: int)
var x: settype
begin loadset(b + displ, x);  storeset(s + 1, x);
  s := s + setlength;  p := p + 2
end
```

Another instruction named outerset is similar.

Stringconst(n, value1, . . . , valuen):
constant(value1) . . . constant(valuen)

```
proc stringconst(no: int; value: valuelist)
var i: int
begin i := 0;
  while i < no do
    i := i + 1; st[s + i] := value[i]
  end;
  s := s + no; p := p + no + 2
end
```

Setconst(n, value1, . . . , valuen):
Stringconst(n, value1, . . . , valuen)
construct(n, lineno)

```
proc setconst(no: int; value: valuelist)
var i: int; new: settype
begin new := settype; i := 0;
  while i < no do
    i := i + 1; new := new + settype(value[i])
  end;
  storeset(s + 1, new);
  s := s + setlength; p := p + no + 2
end
```

Singleton(value):
constant(value) construct(1, lineno)

```
proc singleton(value: int)
var new: settype
begin new := settype(value);
  storeset(s + 1, new);
  s := s + setlength; p := p + 2
end
```

In a variable symbol of the form

v.f1.f2 . . . fn

the displacements of the fields f1, f2, . . . , fn are added to the displacement of the variable named v during compilation ("field encoding").

A conditional statement of the form

true **do** SL

generates code only for the statement list SL.

The last statement list of an if statement is not followed by an else instruction ("else elimination").

The combined result of these optimizations was a reduction of the code of pass 3 from 16200 to 6800 words. This makes the Edison code 24% shorter than the corresponding Pascal code.

The reduction of the original Edison code by 58% can be attributed to the various optimizations as follows:

Line number encoding	16%
Localvalue instruction	11%
Localset instruction	6%
Outercall instruction	5%
Outercase instruction	4%
Localvar instruction	3%
Localcase instruction	2%
Outervalue instruction	2%
Outerset instruction	1%
Outervar instruction	1%
Elemassign instruction	1%
Stringconst instruction	1%
Setconst instruction	1%
Else elimination	1%
True do elimination	1%
Field encoding	1%
Singleton instruction	0.5%
Elemvalue instruction	0.5%
Outerparam instruction	0.4%
Code size reduction	58%

In the Edison kernel for the PDP 11 computer, the code pieces that execute the extra instructions occupy only 150 (out of 1800) words.

The generated code consists of 40% standard instructions and 60% extra instructions.

This completes the description of the Edison code.

The process of code generation has been explained as a two-pass algorithm in which both passes are syntax directed:

First the program text is scanned and translated into standard code that corresponds closely to the syntactic rules of the language.

Then the standard code is scanned and some code sequences are replaced by extra instructions described by another set of syntactic rules.

The ability to explain a complicated data transformation as the result of two straightforward transformations is crucial. It would be much harder to explain the final code without using the standard code as a preliminary, pedagogical concept.