7. Byte Codes

The instruction formats described in this chapter are machine-independent byte codes. These byte codes have an unlimited addressing capability[1)]. As byte codes are well known in computer architecture, a briefly sketched basic example will be sufficient.

Encoded programs comprise byte strings, consisting of control bytes and numerical values. Control bytes contain a type field and a length field (fig. 7.1).

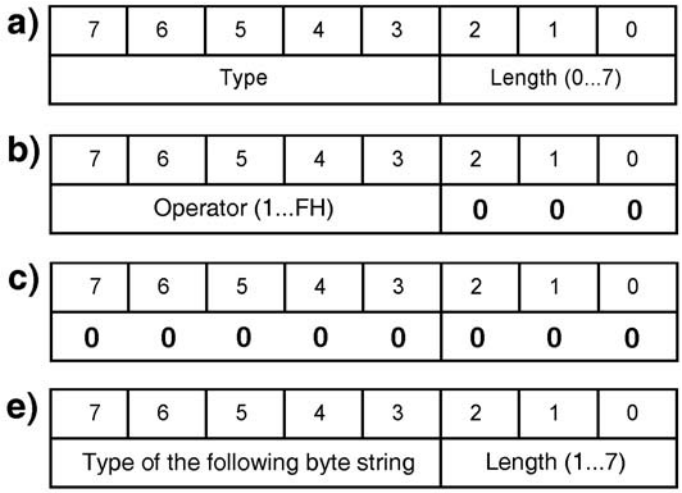

Fig. 7.1 The format of a control byte.

a) The general control byte format, featuring a type field of 5 bits and a length field of 3 bits.

b) Operator. Encoded by Length = 0H. The type field encodes the particular operator (table 7.1).

c) An all-zero control byte encodes an operator without any effect (No Operation (NOP)).

d) Control byte preceding a byte string, containing a numerical value. Its type is encoded within the type field (table 7.2). Encoded length values: 1 byte, 2 bytes, 3 bytes, 4 bytes, 6 bytes, 8 bytes. The remaining length value is reserved.

According to the tables 7.1 and 7.2, 11 types of numerical values and 14 types of operators have been encoded[2)]. The type field of 5 bits therefore provides a reserve of 21 or 18 code positions, respectively (in other words, 21 additional types of numerical data and 18 additional operators still could be encoded).

Numerical values in byte code strings can be ordinal numbers as well as addresses (it is only a question of interpretation). The tables 7.3 and 7.4 show typical operator strings.

---

1): Conventional byte codes have typically a limited addressing capability (example: the byte code of the Java Virtual Machine (JVM)).

2): To define the particular codes (for example, as hex numbers) has been deliberately left over to future work.

operator	function	operator	function
NOP	none	c	establish concatenations between the selected resources
s	select resource out of the resource pool	d	disconnect concatenation
s_a	select resource and assign number or address (select & assign)	l	move data between the selected resources (link)
p	fetch parameter out of system memory and move it into the selected resource (parameter passing)	r	return resources to the resource pool
p_imm	load immediate value into the selected resource (parameter passing)	h	additional information (hints) for supporting compilation and acceleration (for example, speculative memory reads)
y	initiate the information processing operation of the selected resource (yield)	m	additional meta-language information
a	store result from the selected resource into system memory (assign result)	u	utility functions. Here all machine-specific codes are assigned that are provided for supporting other operators (for example, loading platform registers)

Table 7.2 Operator codes.

encoded types of numerical values	corresponding operator
resource type	s
ordinal number of resource (in resource pool)	s
number of resources	s
type of variable	p, a
ordinal number of variable	p, a
bit string (immediate value)	p
source resource	a, l, c, d
source parameter	a, l, c, d
destination resource	p, l, c, d, y
destination parameter	p, l, c, d
function code	h, m, u

Table 7.3 Types of numerical values.

operator	syntax of the numerical data
s	number of resources – resource type
s_a	resource type – resource number – destination resource
p	type of variable – variable number – destination resource – destination parameter
p_imm	immediate value – destination resource – destination parameter
y	destination resource
a	source resource – source parameter – type of variable – variable number
l	source resource – source parameter – destination resource – destination parameter
c	source resource – source parameter – destination resource – destination parameter
d	source resource – source parameter – destination resource – destination parameter
r	destination resource

Table 7.4 Operator strings containing ordinal numbers or addresses supporting a split resource address space, respectively.

operator	syntax of numerical data
s	resource type address
s_a	resource type address – resource address
p	variable address – destination resource address
p_imm	immediate value – destination resource address
y	destination resource address
a	source resource address – variable address
l	source resource address – destination resource address
c	source resource address – destination resource address
d	source resource address – destination resource address
r	destination resource address

Table 7.5 Operator strings containing addresses supporting a flat resource address space.

The operators must be supplemented by numerical data. There are two variants:

1. Postfix notation (fig. 7.2). First the numerical values are provided followed by the operator. The interpreting system has some kind of state buffer that receives the actual values. When passing from one operator to the next, only the information that has changed, respectively, must be entered anew.
2. Prefix notation (fig. 7.3). First the operator is provided followed by the numerical values. Their number must correspond to the respective operator syntax. The interpreting system must have an acceptor automaton that recognizes valid byte sequences. If such a sequence has been detected, the respective function is initiated.

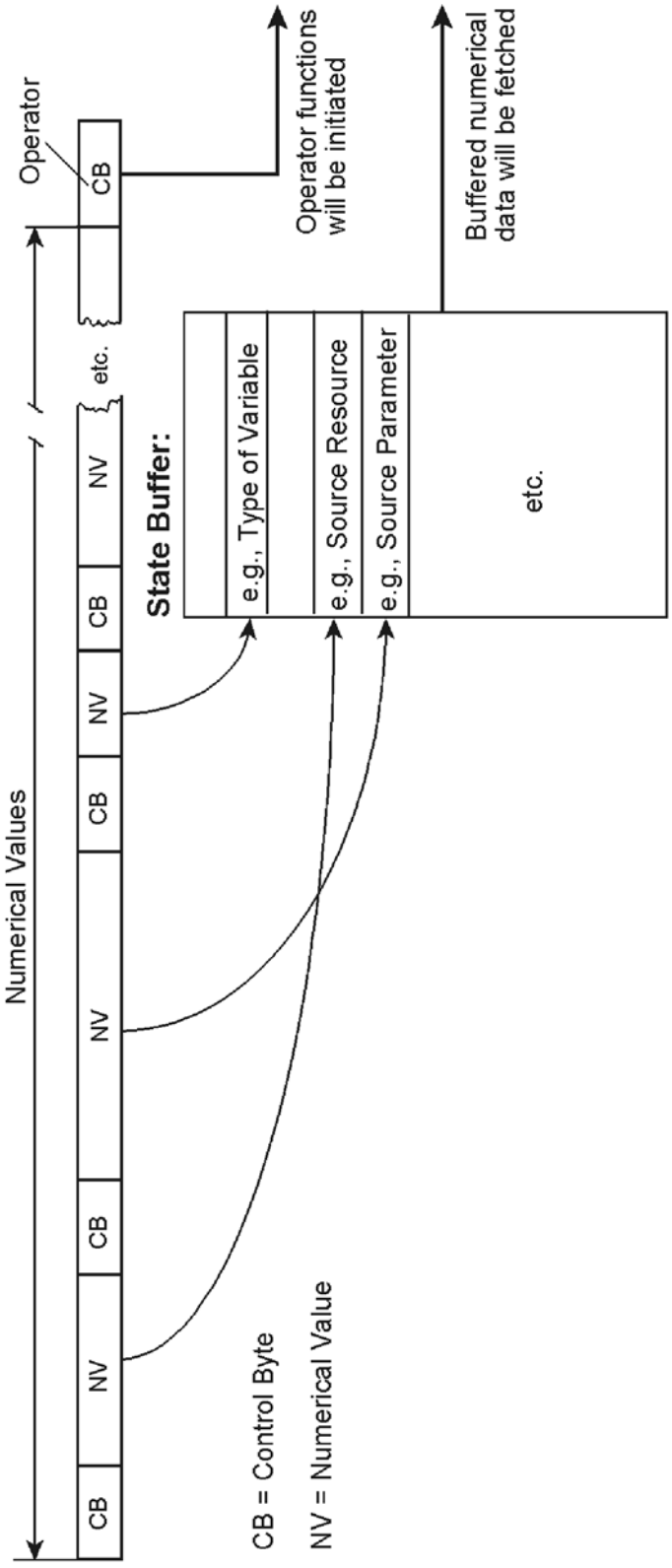

Fig. 7.2 Byte code in postfix notation.

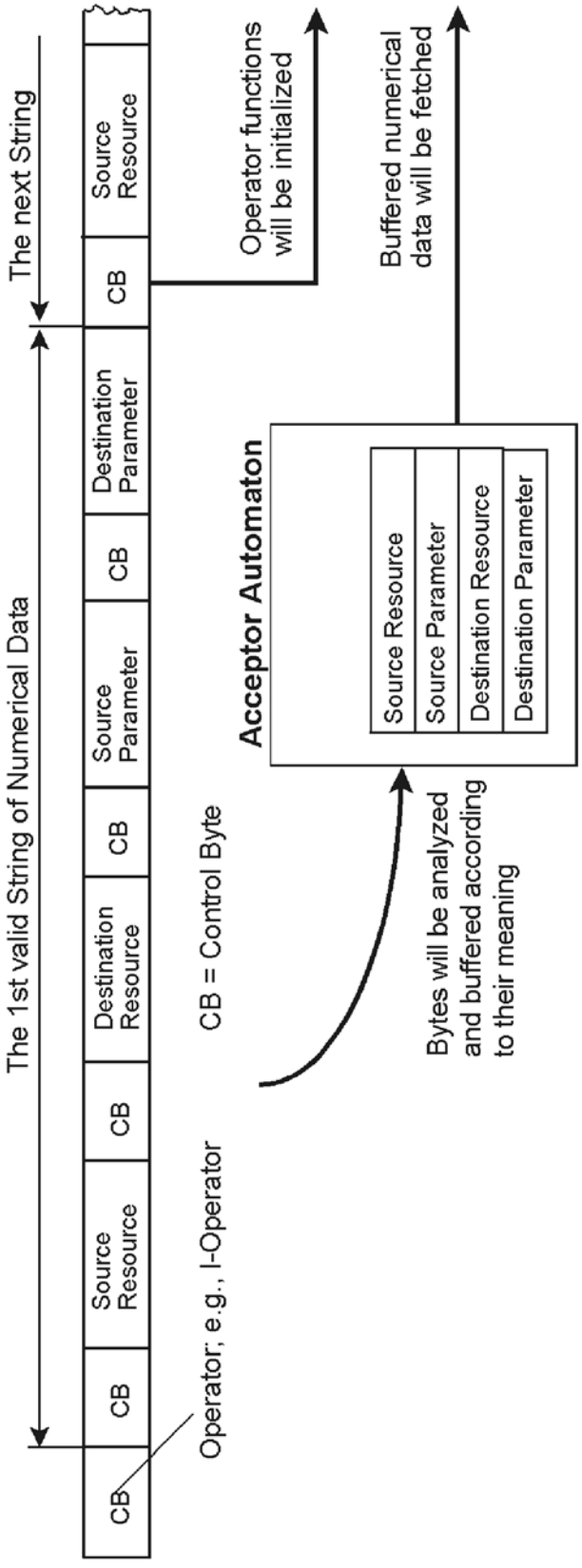

Fig. 7.3 Byte code in prefix notation.

Fig. 7.2 illustrates a byte code in postfix notation. According to the control bytes the numerical values are entered into the respective positions of the state buffer (this is, for example, an area in the system memory or a register file). When an operator arrives, the corresponding function is initiated. The required values are fetched from the state buffer. In the illustrated example, the state buffer has 11 entries, one for each type of numerical values according to table 7.2. When, for example, parameters having up to 8 bytes are allowed, each entry must be able to receive eight data bytes.

In a first implementation, the respective actual length (out of the control byte) is stored additionally so that later on (at the time of execution of the operations), the actual length of the parameter can be determined. In an alternative implementation, all parameters in the state buffer have, for example, a length of 8 bytes. Shorter values are entered right aligned. A parameter value of a length of one byte is entered at bit positions 7...0, a value of two byte length is entered at the bit positions 15...0 and so on. In a further modification, the state buffer is a stack. Numerical values are pushed onto the stack, operators fetch their parameters from the stack. Control bytes with subsequent numerical values essentially represent push instructions; control bytes that encode operators represent operation instructions (wherein their execution causes the operands to be removed from the stack).

Fig. 7.3 illustrates a byte code in prefix notation. The acceptor automaton[1)] analyzes the byte stream. Each operator is characterized by a valid string of numerical values (like shown in the tables 7.3 and 7.42). The received numerical values are buffered. After detecting a complete valid string of bytes, the corresponding function is initiated.

1): The principles of acceptor automata are basic knowledge in computer science and must therefore not be explained in detail.