The TEX Programming Language

Formal Definition

Nils M Holm

Contents

3. Formal Syntax

The syntax of the T3X language is given in the following pages in a variant of the BNF notation [Backus1959] as used in the YACC metacompiler [Johnson1975]:

Each component (statement, expression, etc) that may appear in a program is defined in the following notation in the formal grammar:

name: component_1 component_2 $\cdots$

Such a definition is called a *rule* or *production*. Each component on the right side of the colon in a rule may be

- the name of another rule (also called a *non-terminal symbol*);
- a sequence of characters (called a *word* or *token*) that may appear in a program, like a plus sign or a symbol name.

Tokens are also called *terminal symbols*, because they do not consist of components and can only be matched or rejected while attempting to accept a sentence. In the grammar presented here tokens are either given in upper case characters (like CONST for the “const” keyword) or in single quotes (like ’:=’ for the assignment operator). Non-terminals (rule names) are in lower case. The rule

declaration: VAR var_list ’;’

means: a “declaration” is the keyword VAR followed by a “var_list” (which is defined elsewhere) and a semicolon. The colon is part of the meta language used to describe the T3X language.

Each sentence may have multiple forms. For example, the grammar

declaration: VAR var_list ’;’
declaration: CONST const_list ’;’

specifies two alternative forms for a “declaration”. This can be written more succinctly as

```
declaration:
    VAR var_list ';'
  | CONST const_list ';'
```

where the "|" operator serves as a logical "or".

Upper and lower case is not distinguished in keywords and symbol names of the T3X language, i.e. "VAR", "Var", and "var" all denote the same keyword. Any number of "whitespace" characters (like blanks, newlines, etc) are accepted between components of a sentence.

In addition to the YACC syntax, the following convention is used: when an ampersand (&) appears between two symbols, the symbols form a word in which no separating characters (blank, newline, etc) may appear. I.e. the rules

```
bits: bit | bit & bits
bit: '0' | '1'
```

would accept the sentence '10', but not the sentence '1 0', because the latter contains a blank between the digits.

The set of all productions that describes a *language* is called the *grammar* of that language.

A *sentence* of a language is a sequence of tokens that is accepted by the starting rule of the grammar of that language. In the T3X grammar, the non-terminal symbol program names the starting rule. Given the input

```
DO END
```

the program rule would descend into compound_stmt, which would match the above input directly, thereby accepting the sentence.

```
/*** The T3X/0 grammar begins here ***/

program:
    declaration_list compound_stmt
  | compound_stmt

declaration_list:
    declaration
  | declaration declaration_list
```

```
declaration:
      VAR var_list ';'
    | CONST const_list ';'
    | STRUCT symbol_name '=' struct_members ';'
    | DECL decl_list ';'
    | EXTERN decl_list ';'
    | INLINE inline_decl_list ';'
    | function_decl
    | module_decl
    | USE symbol_name ';'
    | USE symbol_name ':' symbol_name ';'

var_list:
      symbol_name
    | symbol_name '[' const_val ']'
    | symbol_name '::' const_val
    | var_list ',' symbol_name

const_list:
      const_decl
    | const_decl ',' const_list

const_decl:
    symbol_name '=' const_val

decl_list:
      fwd_or_extern_decl
    | fwd_or_extern_decl ',' decl_list

fwd_or_extern_decl:
    symbol_name '(' const_val ')'

inline_decl_list:
      inline_decl
    | inline_decl ',' inline_decl_list

inline_decl:
    symbol_name '(' const_val ')' '=' '[' integer_list ']'

integer_list:
      INTEGER
    | INTEGER ',' integer_list
```

4.2 The Tcode Machine

The *Tcode machine* is a *virtual* or *abstract machine.* It is abstract, because it is entirely defined in mathematical terms. It is virtual, because it is not implemented in hardware, although it maps conveniently to a wide range of existing computer systems.

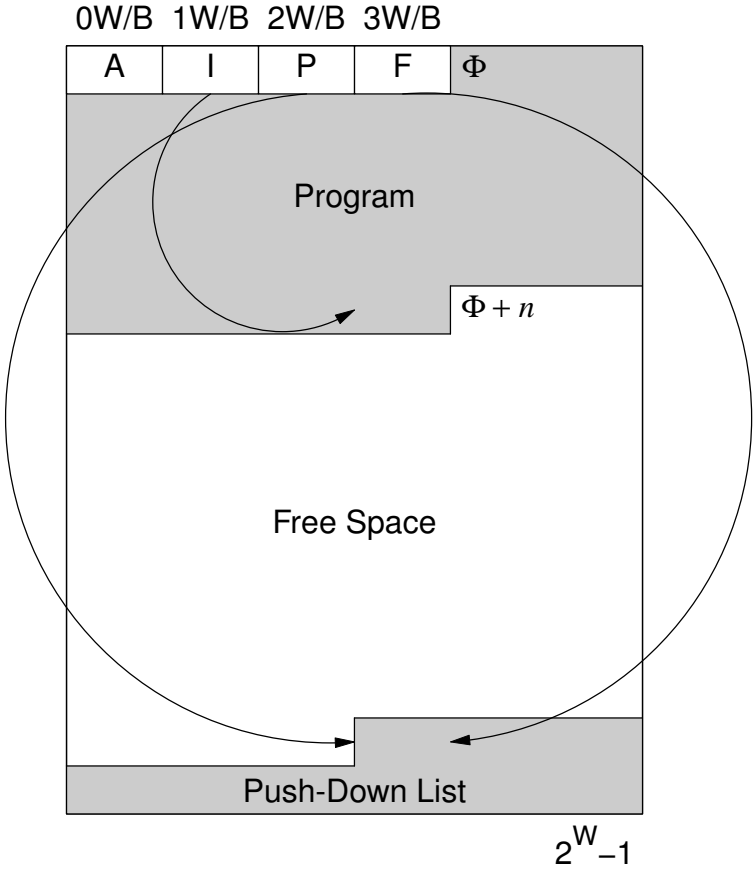

Figure 6: The State of the Tcode machine

The state S of the Tcode machine is depicted in figure 6. It consists of the four word-sized registers A, I, P, and F, a program Φ (of length n), and a push-down list (a.k.a. stack).

It might be tempting at this point to view the state S as the "memory" of the machine, but this would be an incomplete picture.

Each specific state S_i (more on this later) captures a point in a computation on the Tcode machine in its entirety. The machine could be stopped, the state saved, and when restoring the state on the machine, the computation would continue exactly where it was captured.

When mapping the Tcode machine to actual hardware, though, it would make sense to place the registers of the virtual machine in real registers, or a significant loss of performance would occur.

This leads to the dilemma that registers on the Tcode machine are part of the state and can hence be accessed through the memory locations $0W/B$ through $4W/B+1$. This is not desirable, because this mapping will most probably not be given when implementing the machine on hardware. Restricting access to the locations of the registers, for example by making $b(a)$ and $s(a,x)$ undefined for $a \leq 4W/B+1$, would not help, either, because this would restrict access to the registers even for functions that need it. Even limiting this kind of access only in functions that treat the state as memory will cause a contradiction because, again, this limitation would not be given on an actual CPU.

Finally, the state S could be defined as a set

$$S = \{A, I, P, F, M\}$$

where M (“memory”) plays the role of the orignal state S. This, however, would no longer allow to keep the entire state of the machine in a single integer number.

There does not appear to be any satisfactory solution to this dilemma, so it shall just be kept in mind when mapping the abstract Tcode machine to actual hardware.

4.2.1 Registers of the Tcode Machine

The first four words of the Tcode machine contain special portions of its state S that are called the *registers* of the machine. The values stored in these locations have special meanings that will be formally defined in the remainder of this chapter. In order to describe the *meaning* of some part of the Tcode machine, the *semantic brackets* ⟦ and ⟧ will be used. [Scott1971] The brackets will enclose the component of the Tcode machine to define —

mostly its instructions, but also complete programs. A formula of the form

$$[\![X]\!] \triangleq Y$$

will be used to define the meaning of X in terms of the formulae defined earlier in this chapter. For example, the formula

$$[\![CLEAR]\!] \triangleq A \xleftarrow{w} 0$$

defines the meaning of the CLEAR instruction as storing the value 0 in the A register. The A register is an ordinary word in the state S of the Tcode machine, so the $s(a, x)$ function, or its abbreviation $a \xleftarrow{w} x$, can be used to compose a new state with the new value in the A register.

Here are the formal definitions of the registers of the Tcode machine.

$$A = 0W/B \tag{f5.1}$$

$$I = 1W/B \tag{f5.2}$$

$$P = 2W/B \tag{f5.3}$$

$$F = 3W/B \tag{f5.4}$$

Each register corresponds to a word in the state S, so their meaning is the address of the word they occupy. Because register names are just integer values, there is no need to use semantic brackets. They would only make semantic forms unnecessarily heavyweight. Compare, for instance,

$$[\![PUSH]\!] \triangleq [\![P]\!] \xleftarrow{w} w([\![P]\!]) - W/B \;\; ; \;\; w([\![P]\!]) \xleftarrow{w} w([\![A]\!])$$

to

$$[\![PUSH]\!] \triangleq P \xleftarrow{w} w(P) - W/B \;\; ; \;\; w(P) \xleftarrow{w} w(A)$$

P corresponds to $2W/B$, and W/B is the number of bytes per word, so $2B/W$ is the address of the first byte of the second machine word in S. Thus formulae can be simplified as follows to retrieve the meaning of an instruction:

$$[\![CLEAR]\!] \triangleq A \xleftarrow{w} 0$$

simplifies to

$$[\![CLEAR]\!] \triangleq 0W/B \xleftarrow{w} 0$$

and further to

$$[\![CLEAR]\!] \;\hat{=}\; 0 \xleftarrow{w} 0$$

I.e., the meaning of the CLEAR instruction is to create a transition from a state S to a state S' that stores a word-sized value 0 at address 0.

4.2.2 The Programs of the Tcode Machine

From this point on it will be necessary to differentiate between a *unspecific* or *general state* S and a *specific* or *concrete state* S_i. A general state S is essentially any number $0 \le n \le 2^{B'+W}$ that represents some unspecific state of the Tcode machine. Functions like $b(a)$, $w(a)$, and $s(a, x)$ can be applied to any such unspecific state S and will always deliver a valid value, i.e. they are total functions of the set of possible states that are commonly referred to as S. Formally, this set $\mathbf{D}$ (the domain of b, w, etc) would be

$$\mathbf{D} \;=\; \{x \mid 0 \le x < 2^{B'+W}\}$$

A specific state is a state S_i in which the values of the registers, the program, and the push-down list are known. The *program* of a specific state consists of the values stored in a continuous range of addresses Φ through $\Phi + n - 1$, where n is the length of the program.

The push-down list consists of a continuous range of words stored in the addresses $w(P)$ through $2^W - W/B$, i.e. it allocates all addresses between (and including) $w(P)$ and the highest possible word address in the state S_i. If $w(P) = 0$, the PDL is empty.

The *initial state* S_0 of the Tcode machine is equal to the empty state $S_\varepsilon = 0$ with a program added to the addresses $i = a$ through $a + n - 1$ by storing its bytes in the state using $s(a_i, x_j)$ where each x_j with $0 \le j < n$ is an instruction (or part of an instruction) of the program. Furthermore, $s(I, a)$ initially, i.e. the I register contains the address of the first instruction in the program. The P register is 0, so the PDL is empty. This will be defined more rigorously in the next section.

The function $\tau(S)$ gives meaning to the program stored in a concrete state S_i by performing the computation that corresponds

to the instruction stored at $b(I)$. E.g.:

$$\tau(S) \;=\; \begin{cases} \cdots \\ A \xleftarrow{w} 0, \qquad \textit{if } b(I) = \mathit{CLEAR} \\ \cdots \\ \textit{undefined}, \;\; \text{otherwise} \end{cases} \tag{f6.1a}$$

I.e. $\tau(S)$ implements the "corresponds to" sign in the formula

$[\![\mathit{CLEAR}]\!] \;\hat{=}\; A \xleftarrow{w} 0$

but it does so for all instructions of the Tcode machine.

There are 72 Tcode instructions, so enumerating the entire $\tau(S)$ function would be a tedious exercise. Instead the semantics of the individual Tcode instructions will be given in the next section. The $\tau(S)$ function is a partial function on the set of possible states S, because not all bytes-sized values are valid instructions.

Note that there is one condition in which the Tcode machine enters an undefined state which results in $\tau(S)$ being undefined also. This happens when the PDL pointer P takes on a value below $\Phi + n$, i.e. when the PDL grows so much that it overlaps with the program Φ. This is formalized as follows:

$$\tau(S) \;=\; \begin{cases} \textit{undefined}, \;\; \text{if } w(P) < \Phi + n \\ \cdots \\ A \xleftarrow{w} 0, \qquad \textit{if } b(I) = \mathit{CLEAR} \\ \cdots \\ \textit{undefined}, \;\; \text{otherwise} \end{cases} \tag{f6.1}$$

The operator $\mathop{\mathbf{R}}_{i=x}^{x} f(S_i)$ performs the transition from a state S_i to a state S_{i+1} by applying the function f to the *current state* S_i. I.e.,

$$\mathop{\mathbf{R}}_{i=x}^{x} f(S_i) \;=\; f(S_i) \;=\; S_{i+1}$$

By widening the range of i, multiple transitions are performed:

$$\mathop{\mathbf{R}}_{i=x}^{x+1} f(S_i) \;=\; f(f(S_i)) \;=\; S_{i+2}$$
$$\cdots$$
$$\mathop{\mathbf{R}}_{i=x}^{x+n} f(S_i) \;=\; f^n(S_i) \;=\; S_{i+n}$$

5. Mapping T3X to Tcode

5.1 Symbol Tables

A *symbol table* is a set Y of tuples (n, p, v), where n is the *name* of a symbol, p a number describing the properties (or "flags") of the symbol, and v another number denoting a value associated with that symbol. The name of a symbol is any sequence of letters, digits, and the underscore character "_", where the first character may not be a digit. Its encoding does not matter. The value of a symbol is, for example, the address at which a variable is stored, or the value of a constant. The *properties* of a symbol are encoded in the base-2 number p as described in figure 7.

`0000000000000001`	global
`0000000000000010`	constant
`0000000000000100`	vector
`0000000000001000`	forward declaration
`0000000000010000`	function
`0000000000100000`	module
`0000000001000000`	public symbol
`0000000010000000`	alternative module name (alias)
`0000000100000000`	extern function
`xxxxxx0000000000`	function arity

Figure 7: Encoding of symbol properties (flags)

A set of properties can be created by adding the desired flags. For example a public (and hence global) constant would have the flags

$$1_2 + 10_2 + 1000000_2 = 1000011_2,$$

and a "function of three arguments" would have the following flags (all functions are global):

$$1_2 + 10000_2 + 110000000000_2 = 110000010001_2$$

A global constant named “foo” with value 25 would be represented by the tuple $(\text{foo}, 11_2, 25)$.

The properties of symbols are denoted symbolically as follows

$$\begin{aligned} GLOB &= 1 \\ CNST &= 2 \\ VECT &= 4 \\ FORW &= 8 \\ FUNC &= 16 \\ MODU &= 32 \\ PUBL &= 64 \\ ALIA &= 128 \\ EXTN &= 256 \end{aligned} \tag{f8.1}$$

The function $\rho(p, f)$ returns 1, if the word p has the given property flag f set and 0 otherwise:

$$\rho(p, f) \;=\; \frac{p}{f} \; mod \; 2 \tag{f8.2}$$

The function $\sigma(p)$ returns the *arity* (number of formal arguments) of the function having the properties y:

$$\sigma(y) \;=\; \frac{p}{1024} \; mod \; 64 \tag{f8.3}$$

Due to this encoding a function cannot have more than 63 formal arguments.

A symbol y is contained in the symbol table Y, if there is a tuple $t \in Y$ such that $t = (y, p, v)$, or, more briefly, if $(y, p, v) \in Y$:

$$y \dot{\in} Y, \text{ if } (y, p, v) \in Y \text{ for any } p \text{ and } v \tag{f8.4}$$

A symbol is added to the symbol table by forming the union of the existing table and a singleton set containing the new symbol. This is what the $D(Y, (n, p, v))$ function does. Note that the T3X symbol table is a true set, because there is no shadowing, i.e., $(a, p, v) \in Y$ and $(b, q, w) \in Y$ implies $a \neq b$.

$$Y_i = D(Y_{i-1}, (n, p, v)) \;=\; \begin{cases} undefined, & \text{if } n \dot{\in} Y_{i-1} \\ Y_{i-1} \cup \{(n, p, v)\}, & \text{otherwise} \end{cases} \tag{f8.5}$$

Symbols are removed from a symbol table by forming the set difference of the symbol table and a singleton set containig the

tuple to remove:

$$Y_i = Y_{i-1} \setminus \{(n, p, v)\}$$

Alternatively, multiple symbols can be removed by referring to a previous version Y_j (where $j < i$) of the symbol table. [1] When referring to "the" symbol table Y, Y_i shall be implied unless stated otherwise.

5.2 Formal Semantics

In the following subsections the semantics of the individual T3X statements will be described formally by mapping them to sequences of Tcode instructions. The following definitions are used. The form $w[x]$ will denote a machine word with the value x and the form $b[x]$ will denote a byte with the value x. For instance,

$$[\![RJUMP\ b[W/B]]\!]$$

denotes a "relative jump" over W/B bytes. The notation a: will associate the variable a with the address of the byte following after the a:. For example,

$$[\![JUMP\ w[a]\ w[0]\ a:]\!]$$

denotes a jump over a machine word $w[0]$.

When a negative value $-x$ appears in $w[-x]$, then its conversion to two's complement is implied:

$$w[-x] = w[c(-x)]$$

In the beginning, formal definitions will be accompanied by informal explanations. When the formal definitions become more obvious, the additional prose will become more sparse.

5.2.1 Concatenation

Tcode is a *concatenative language* in which instructions form a map from state to state. Therefore programs can be combined or *composed* by placing them at subsequent addresses in the state S. For example, the Tcode program

[1] but forming the set difference is more mathematically sound

LDVAL 2 *MUL*

multiples a value by two and

LDVAL 5 *ADD*

adds five to a value. Because Tcode is concatenative, the program

LDVAL 2 *MUL LDVAL* 5 *ADD*

computes $x \cdot 2 + 5$. Other concatenative languages include, for example, FORTH [Moore1970] and Joy [vonThun1995].

The translation from T3X to Tcode makes use of the concatenative property of Tcode by emitting fragments of code to subsequent addresses. Let Φ be the address at which the emitted Tcode program will be located. Then the first partial program (of length m) will allocate the addresses Φ through $\Phi + m - 1$, the second fragment (of length n) will allocate the addresses $\Phi + m$ through $\Phi + m + n - 1$, etc.

The concrete address Φ does not matter, but it makes sense to set Φ to the lowest possible address. On the Tcode machine that would be $4W$, i.e. the first byte after the registers of the machine.

5.2.2 Global VAR Declarations

A **var** statement declaring n variables is equal to n **var** statements declaring one variable each.

$$[\![\textbf{var}\ x_1\textbf{,}\ \cdots\textbf{,}\ x_n\textbf{;}]\!] \ \hat{=}\ [\![\textbf{var}\ x_1\textbf{;}\ \cdots\ \textbf{var}\ x_n\textbf{;}]\!] \qquad \text{(var.1)}$$

The declaration of a scalar variable y creates a jump over the word allocated to the variable. The word has an initial value of 0. A new symbol table entry is added for the name y with properties indicating a global symbol and a value that is the address of the allocated machine word.

$$[\![\textbf{var}\ y\textbf{;}]\!] \ \hat{=}\ [\![RJUMP\ b[W/B]\ a\colon w[0]]\!] \qquad \text{(var.2)}$$
$$Y_i = D(Y_{i-1}, (y, GLOB, a))$$

The declaration of a global vector (of machine words) creates a global scalar variable that is being initialized with the address of a vector that is allocated on the push down list.

6. Mapping Tcode to CPUs

The *Tcode machine* can be implemented on any computer that provides

- byte-addressed memory
- two's-complement arithmetic
- at least one register

One additional register can be beneficial in the implementation of code fragments, but is not necessary and can easily be replaced with a word in memory.

Tcode is mapped to the instruction set of a CPU one Tcode instruction at a time. Each code fragment that implements a Tcode instruction expects the machine in a state S_i and, after executing, leaves the machine in a state S_{i+1}. In other words, Tcode is *concatenative*: programs are formed by concatenating Tcode instructions. This property is not necessarily given on all CPUs. For instance, a conditional jump usually follows a comparison operation on machines with a flags register.

The fragments implementing Tcode instructions must be closed in the sense that all information transferred from a Tcode instruction to the next is visible in the state S (which a flags register, for example, would not be).

Of course this prerequisite limits the way in which code can be generated and precludes many optimizations. The benefit of this approach is a highly portable compiler, which can be retargetted to a new platform by translating isolated small snippets of machine code. Optimization still is possible, but not covered in this text.

The following pages will list program fragments resembling Tcode instructions for the Z80, 8086, and ARMv6 architectures.

In the following tables, W denotes a machine word, H and L the offset of the "high" and "low" byte of a machine word, R a relative address, and A an absolute address (label).

Tcode	Z80	8086	ARMv6
PUSH	`push hl`	`push ax`	`push {r0}`
CLEAR	`ld hl,0`	`xor ax,ax`	`eor r0,r0,r0`
DROP	`pop de`	`pop bx`	`pop {r1}`

LDVAL	`ld hl,W`	`mov ax,W`	`ldr r0,[pc,#0]`
			`b .+8`
			`.long W`

LDADDR	`ld hl,A`	`mov ax,A`	`ldr r0,[pc,#0]`
			`b .+8`
			`.long A`

LDLREF	`ld hl,W`
	`push ix`
	`pop de`
	`add hl,de`
		`lea ax,[bp+W]`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long W`
			`add r0,r11,r1`

LDGLOB	`ld hl,(A)`	`mov ax,[A]`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long A`
			`ldr r0,[r1]`

LDLOCL †	`ld h,(ix+H)`
	`ld l,(ix+L)`
		`mov ax,[bp+W]`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long W`
			`add r0,r11,r1`
			`ldr r0,[r0]`

STGLOB	`ld (A),hl`	`mov [A],ax`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long A`
			`str r0,[r1]`

STLOCL †	`ld (ix+H),h`
	`ld (ix+L),l`
		`mov [bp+W],ax`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long W`
			`add r1,r11,r1`
			`str r0,[r1]`

† Only up to 126 bytes can be allocated to local storage in **do** blocks on the Z80.

Tcode	Z80	8086	ARMv6

STINDR	`ex de,hl`
	`pop hl`
	`ld (hl),e`
	`inc hl`
	`ld (hl),d`
		`pop bx`
		`mov [bx],ax`
			`pop {r1}`
			`str r0,[r1]`

STINDB	`ex de,hl`
	`pop hl`
	`ld (hl),e`
		`pop bx`
		`mov [bx],al`
			`pop {r1}`
			`strb r0,[r1]`

INCGLOB	`ld hl,A`
	`inc (hl)`
	`jrnz +1`
	`inc (hl)`
		`inc [A]`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long A`
			`ldr r0,[r1]`
			`add r0,r0,#1`
			`str r0,[r1]`

INCLOCL †	`inc (ix+L)`
	`jrnz +3`
	`inc (ix+H)`
		`inc [bp+W]`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long W`
			`add r1,r11,r1`
			`ldr r0,[r1]`
			`add r0,r0,#1`
			`str r0,[r1]`

INCR	`ld de,W`
	`add hl,de`
		`add ax,W`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long W`
			`add r0,r0,r1`

STACK	`ld hl,W`
	`add hl,sp`
	`ld sp,hl`
		`add sp,W`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long W`
			`add sp,sp,r1`

UNSTACK	`ex de,hl`
	`ld hl,W`
	`add hl,sp`
	`ld sp,hl`
	`ex de,hl`
		`add sp,W`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long W`
			`add sp,sp,r1`

† see previous page

Tcode	Z80	8086	ARMv6
LOCLVEC	`push hl`	`mov ax,sp` `push ax`	`mov r1,sp` `push {r1}`

GLOBVEC	`ld (A),hl`	`mov [A],sp`	`ldr r1,[pc,#0]`
			`b .+8`
			`.long A`
			`str sp,[r1]`

INDEX	`add hl,hl`
	`pop de`
	`add hl,de`
		`shl ax,1`
		`pop bx`
		`add ax,bx`
			`pop {r1}`
			`add r0,r1,r0,`
			`lsl #2`

DEREF	`ld a,(hl)`
	`inc hl`
	`ld h,(hl)`
	`ld l,a`
		`mov bx,ax`
		`mov ax,[bx]`
			`ldr r0,[r0]`

INDXB	`pop de` `add hl,de`	`pop bx` `add ax,bx`	`pop {r1}` `add r0,r1,r0`

DREFB	`ld l,(hl)`
	`ld h,0`
		`mov bx,ax`
		`xor ax,ax`
		`mov al,[bx]`
			`ldrb r0,[r0]`

CALL	`call A`	`call R`	`bl A`
CALR	`call IND` †	`call ax`	`blx r0`
JUMP	`jp A`	`jmp R`	`b A`
RJUMP	`jr R`	`jmps R`	`b A`

JMPFALSE	`ld a,h`
	`or l`
	`jpz A`
		`or ax,ax`
		`jne +3`
		`jmp R`
			`cmp r0,#0`
			`beq A`

JMPTRUE	`ld a,h`
	`or l`
	`jpnz A`
		`or ax,ax`
		`je +3`
		`jmp R`
			`cmp r0,#0`
			`bne A`

FOR	`pop de`
	`ex de,hl`
	`call CMP`
	`jpnc A`
		`pop bx`
		`cmp bx,ax`
		`jl +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`bge A`

† `IND: jp (hl)`

Tcode	Z80	8086	ARMv6

FORDOWN	`pop de`
	`call CMP`
	`jpnc A`
		`pop bx`
		`cmp bx,ax`
		`jg +3`
		`jmp R`
			`pop {r1}`
			`cmp r0,r1`
			`bge A`

MKFRAME	`push ix`
	`ld ix,0`
	`add ix,sp`
		`push bp`
		`mov bp,sp`
			`push {r11}`
			`mov r11,sp`

DELFRAME	`pop ix`	`pop bp`	`pop {r11}`
RET	`ret`	`ret`	`pop {pc}`

HALT †	`jp 0`	`mov ax,W`
		`mov ah,4Ch`
		`int 21h`
			`ldr r0,[pc,#0]`
			`b .+8`
			`.long W`
			`bl _exit`

NEG	`ld de,0`
	`ex de,hl`
	`xor a`
	`sbc hl,de`
		`neg ax`	`rsb r0,r0,#0`

INV	`ld de,ffffh`
	`ex de,hl`
	`xor a`
	`sbc hl,de`
		`not ax`	`mvn r0,r0`

LOGNOT	`ex de,hl`
	`ld hl,ffffh`
	`ld a,d`
	`or e`
	`jrnz +1`
	`inc hl`
		`neg ax`
		`sbb ax,ax`
		`not ax`
			`cmp r0,#0`
			`eor r0,r0,r0`
			`subeq r0,r0,#1`

ADD	`pop de` `add hl,de`	`pop bx` `add ax,bx`	`pop {r1}` `add r0,r1,r0`

SUB	`pop de`
	`ex de,hl`
	`xor a`
	`sbc hl,de`
		`mov bx,ax`
		`pop ax`
		`sub ax,bx`
			`pop {r1}`
			`sub r0,r1,r0`

† Assuming CP/M on the Z80, DOS on the 8086, and Unix on the ARMv6 processor.

Tcode	Z80	8086	ARMv6
MUL	`pop de` `call MUL`	`pop cx` `imul cx`	`pop {r1}` `mul r0,r1,r0`

DIV	`pop de`
	`call DIV`
		`mov cx,ax`
		`pop ax`
		`cwd`
		`idiv cx`
			`mov r1,r0`
			`pop {r0}`
			`bl SDIV`

MOD	`pop de`
	`call MOD`
		`mov cx,ax`
		`pop ax`
		`xor dx,dx`
		`div cx`
		`mov ax,dx`
			`mov r1,r0`
			`pop {r0}`
			`bl UDIV`
			`mov r0,r1`

AND	`pop de`
	`ld a,h`
	`and d`
	`ld h,a`
	`ld a,l`
	`and e`
	`ld l,a`
		`pop bx`
		`and ax,bx`
			`pop {r1}`
			`and r0,r1,r0`

OR	`pop de`
	`ld a,h`
	`or d`
	`ld h,a`
	`ld a,l`
	`or e`
	`ld l,a`
		`pop bx`
		`or ax,bx`
			`pop {r1}`
			`orr r0,r1,r0`

XOR	`pop de`
	`ld a,h`
	`xor d`
	`ld h,a`
	`ld a,l`
	`xor e`
	`ld l,a`
		`pop bx`
		`xor ax,bx`
			`pop {r1}`
			`eor r0,r1,r0`

Tcode	Z80	8086	ARMv6

SHL	`pop de`
	`ex de,hl`
	`ld b,e`
	`add hl,hl`
	`djnz -3`
		`mov cx,ax`
		`pop ax`
		`shl ax,cl`
			`pop {r1}`
			`lsl r0,r1,r0`

SHR	`pop de`
	`ex de,hl`
	`ld b,e`
	`srl h`
	`rr l`
	`djnz -6`
		`mov cx,ax`
		`pop ax`
		`shr ax,cl`
			`pop {r1}`
			`lsr r0,r1,r0`

EQ	`pop de`
	`xor a`
	`sbc hl,de`
	`ld hl,ffffh`
	`jrz +1`
	`inc hl`
		`call EQ`	`pop {r1}`
			`cmp r0,r1`
			`eor r0,r0,r0`
			`subeq r0,r0,#1`

NE	`pop de`
	`xor a`
	`sbc hl,de`
	`ld hl,ffffh`
	`jrnz +1`
	`inc hl`
		`call NE`	`pop {r1}`
			`cmp r0,r1`
			`eor r0,r0,r0`
			`subne r0,r0,#1`

LT	`pop de`
	`ex de,hl`
	`call CMP`
	`ld hl,ffffh`
	`jrc +1`
	`inc hl`
		`call LT`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`sublt r0,r0,#1`

GT	`pop de`
	`call CMP`
	`ld hl,ffffh`
	`jrc +1`
	`inc hl`
		`call GT`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`subgt r0,r0,#1`

Tcode	Z80	8086	ARMv6

LE	`pop de`
	`call CMP`
	`ld hl,0`
	`jrc +1`
	`dec hl`
		`call LE`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`suble r0,r0,#1`

GE	`pop de`
	`ex de,hl`
	`call CMP`
	`ld hl,0`
	`jrc +1`
	`dec hl`
		`call GE`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`subge r0,r0,#1`

UMUL	`pop de` `call UMUL`	`pop cx` `mul cx`	`pop {r1}` `mul r0,r1,r0`

UDIV	`pop de`
	`ex de,hl`
	`call UDIV`
		`mov cx,ax`
		`pop ax`
		`xor dx,dx`
		`div cx`
			`mov r1,r0`
			`pop {r0}`
			`bl UDIV`

ULT	`pop de`
	`ex de,hl`
	`call UCMP`
	`ld hl,ffffh`
	`jrc +1`
	`inc hl`
		`call ULT`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`sublo r0,r0,#1`

UGT	`pop de`
	`call UCMP`
	`ld hl,ffffh`
	`jrc +1`
	`inc hl`
		`call UGT`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`subhi r0,r0,#1`

ULE	`pop de`
	`call UCMP`
	`ld hl,0`
	`jrc +1`
	`dec hl`
		`call ULE`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`subls r0,r0,#1`

Tcode	Z80	8086	ARMv6

UGE	`pop de`
	`ex de,hl`
	`call UCMP`
	`ld hl,0`
	`jrc +1`
	`dec hl`
		`call UGE`	`pop {r1}`
			`cmp r1,r0`
			`eor r0,r0,r0`
			`subhs r0,r0,#1`

JMPEQ	`pop de`
	`xor a`
	`sbc hl,de`
	`jpz A`
		`pop bx`
		`cmp bx,ax`
		`jne +3`
		`jmp R`
			`pop {r1}`
			`cmp r0,r1`
			`beq A`

JMPNE	`pop de`
	`xor a`
	`sbc hl,de`
	`jpnz A`
		`pop bx`
		`cmp bx,ax`
		`je +3`
		`jmp R`
			`pop {r1}`
			`cmp r0,r1`
			`bne A`

JMPLT	`pop de`
	`ex de,hl`
	`call CMP`
	`jpc A`
		`pop bx`
		`cmp bx,ax`
		`jge +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`blt A`

JMPGT	`pop de`
	`call CMP`
	`jpc A`
		`pop bx`
		`cmp bx,ax`
		`jle +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`bgt A`

JMPLE	`pop de`
	`call CMP`
	`jpnc A`
		`pop bx`
		`cmp bx,ax`
		`jg +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`ble A`

JMPGE	`pop de`
	`ex de,hl`
	`call CMP`
	`jpnc A`
		`pop bx`
		`cmp bx,ax`
		`jl +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`bge A`

JMPULT	`pop de`
	`ex de,hl`
	`call UCMP`
	`jpc A`
		`pop bx`
		`cmp bx,ax`
		`jae +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`blo A`

Tcode	Z80	8086	ARMv6

JMPUGT	`pop de`
	`call UCMP`
	`jpc A`
		`pop bx`
		`cmp bx,ax`
		`jbe +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`bhi A`

JMPULE	`pop de`
	`call UCMP`
	`jpnc A`
		`pop bx`
		`cmp bx,ax`
		`ja +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`bls A`

JMPUGE	`pop de`
	`ex de,hl`
	`call UCMP`
	`jpnc A`
		`pop bx`
		`cmp bx,ax`
		`jb +3`
		`jmp R`
			`pop {r1}`
			`cmp r1,r0`
			`bhs A`

SKIP	`jp A`	`jmp R`	`b A`

CALN
†
			`ldr r0,[sp,#0]`
			`ldr r1,[sp,#4]`
			`ldr r2,[sp,#8]`
			`ldr r3,[sp,#12]`
			`ldr r4,[sp,#16]`
			`ldr r5,[sp,#20]`
			`push {r5}`
			`push {r4}`
			`bl t3x_N`
			`add sp,#8`

LDNAM
†
			`ldr r0,[pc,#0]`
			`b .+8`
			`.long t3x_N`

ENTER ‡			`push {lr}`

† The *CALN* and *LDNAM* instructions are used to refer to external procedures whose names are given in symbolic form. Their semantics resemble those of *CALL* and *LDADDR*. On the ARMv6 *CALN* also loads registers with arguments, because this is how the ARMv6 ABI works.

‡ An *ENTER* instruction is generated at the beginning of procedures on the ARMv6 in order to save the link register.