1 P-instructions

1.1 Expressions and Assignments

instruction	meaning	condition	result	comments	references
add N	STORE[SP - 1] := STORE[SP - 1] $+_N$ STORE[SP]; SP := SP - 1	(N, N)	(N)		tab. 2.1, p. 11
sub N	STORE[SP - 1] := STORE[SP - 1] $-_N$ STORE[SP]; SP := SP - 1	(N, N)	(N)		tab. 2.1, p. 11
mul N	STORE[SP - 1] := STORE[SP - 1] $*_N$ STORE[SP]; SP := SP - 1	(N, N)	(N)		tab. 2.1, p. 11
div N	STORE[SP - 1] := STORE[SP - 1] $/_N$ STORE[SP]; SP := SP - 1	(N, N)	(N)		tab. 2.1, p. 11
neg N	STORE[SP] := $-_N$ STORE[SP];	(N)	(N)		tab. 2.1, p. 11
and	STORE[SP - 1] := STORE[SP - 1] and STORE[SP]; SP := SP - 1	(b, b)	(b)		tab. 2.1, p. 11
or	STORE[SP - 1] := STORE[SP - 1] or STORE[SP]; SP := SP - 1	(b, b)	(b)		tab. 2.1, p. 11
not	STORE[SP] := note STORE[SP];	(b)	(b)		tab. 2.1, p. 11
equ T	STORE[SP - 1] := STORE[SP - 1] $=_T$ STORE[SP]; SP := SP - 1	(T, T)	(b)		tab. 2.1, p. 11
geq T	STORE[SP - 1] := STORE[SP - 1] $\geq_T$ STORE[SP]; SP := SP - 1	(T, T)	(b)		tab. 2.1, p. 11
leq T	STORE[SP - 1] := STORE[SP - 1] $\leq_T$ STORE[SP]; SP := SP - 1	(T, T)	(b)		tab. 2.1, p. 11
les T	STORE[SP - 1] := STORE[SP - 1] $<_T$ STORE[SP]; SP := SP - 1	(T, T)	(b)		tab. 2.1, p. 11
grt T	STORE[SP - 1] := STORE[SP - 1] $>_T$ STORE[SP]; SP := SP - 1	(T, T)	(b)		tab. 2.1, p. 11

instruction	meaning	condition	result	comments	references
neq T	STORE[SP - 1] := STORE[SP - 1] $\neq_T$ STORE[SP]; SP := SP - 1	(T, T)	(b)		tab. 2.1, p. 11

1.2 Store and Load Instructions

instruction	meaning	condition	result	comments	references
ldo T q	SP := SP + 1; STORE[SP] := STORE[Q]	$q \in [0, maxstr]$	(T)	load location given by absolute address on top of stack	tab. 2.2, p. 12
ldc T q	SP := SP + 1; STORE[SP] := q	type(q) = T	(T)	load constant q on top of stack	tab. 2.2, p. 12
ind T	STORE[SP] := STORE[STORE[SP]]	(a)	(T)	load indirectly using highest stack location	tab. 2.2, p. 12
sro T q	STORE[q] := STORE[SP]; SP := SP - 1	(T) $q \in [0, maxstr]$		stores in location addressed by absolute address	tab. 2.2, p. 12
sto T	STORE[STORE[SP - 1]] := STORE[SP]; SP := SP - 2	(a, T)		stores in location addressed by 2^{nd} highest stack location	tab. 2.2, p. 12

1.3 Conditional and Iterative Statements

instruction	meaning	condition	result	comments	references
ujp q	PC := q	$q \in [0, codemax]$		unconditional branch	tab. 2.4, p. 14
fjp q	if STORE[SP] = false then PC := q fi SP := SP - 1	(b) $q \in [0, codemax]$		conditional branch	tab. 2.4, p. 14

instruction	meaning	condition	result	comments	references
ixj q	PC := STORE[SP] + q; SP := SP - 1		(i)	indexed branch (switch)	tab. 2.5, p. 17
ixa q	STORE[SP - 1] := STORE[SP - 1] + STORE[SP] $* q$; SP := SP -1	(a, i)	(a)	indexed address computation: start address in STORE[SP - 1], index of selected subarray in STORE[SP], $q = g \cdot d^{(j)}$ subarray size	tab. 2.6, p. 22
inc T	STORE[SP] := STORE[SP] + q	(T) and type(q) = i	(T)		tab. 2.7, p. 23
dec T	STORE[SP] := STORE[SP] - q	(T) and type(q) = i	(T)		tab. 2.7, p. 23

1.4 Array Indexation

instruction	meaning	condition	result	comments	references
chk p q	if (STORE[SP] $<p$) **or** (STORE[SP] $>q$) then error('value out of range') fi	(i,i)	(i)	check array boundaries	tab. 2.8, p. 23
dpl T	SP := SP + 1; STORE[SP] := STORE[SP - 1]	(T)	(T, T)	copies highest stack entry	tab. 2.9, p. 27 tab. 2.9, p. 27
ldd T	SP := SP + 1; STORE[SP] := STORE[STORE[SP - 3] + q]	(a, T_1, T_2)	(a, T_1, T_2, i)	indirect access to descriptor arrays	tab. 2.9, p. 27 tab. 2.9, p. 27
sli T_2	STORE[SP - 1] := STORE[SP]; SP := SP - 1	(T_1, T_2)	(T_2)	move highest stack entry in 2^{nd} highest position	tab. 2.9, p. 27

1.5 Dynamic Memory Allocation

instruction	meaning	condition	result	comments	references
new	if NP - STORE[SP] $\leq$ EP then error('store overflow') else NP := NP - STORE[SP]; STORE[STORE[SP - 1]] := NP; SP := SP - 2; fi	(a, i)		 object size at top of stack address of pointer just below ptr $\rightarrow$ object start address	tab. 2.10, p. 29 fig. 2.7, p. 30

1.6 Procedure Calls and Frames on the Stack

1.6.1 Loading and Storing Bound and Free Variables

instruction	meaning	condition	result	comments	references
lod T p q	SP := SP + 1; STORE[SP] := STORE[$base(p, MP) + q$]		(T)	load a value of type T nesting depth p, relative address q	tab. 2.11, p. 42
lda p q	SP := SP + 1; STORE[SP] := $base(p, MP) + q$		(a)	loads addresses nesting depth p, relative address q	tab. 2.11, p. 42
str T p q	STORE[$base(p, MP) + q$] := STORE[SP]; SP := SP - 1		(T)	storing values nesting depth p, relative address q	tab. 2.11, p. 42
!!!	$base(p, a) =$ **if** $p = 0$ **then** a **else** $base(p-1, STORE[a+1])$ **fi**: e.g. $base(d, MP)$ follows SL chain d times to return MP of the predecessor frame (Fig. 2.12, p. 42)				**!!!**

1.6.2 Instructions for Calling and Entering Procedures (Caller)

instruction	meaning	condition	result	comments	references
mst p	STORE[SP + 2] := $base(p, MP)$; STORE[SP + 3] := MP; STORE[SP + 4] := EP; SP := SP + 5;		stack marked with organisational block	set SL to point to static predecessor's MP p nrof times to follow SL-chain set DL to pint to start of caller's frame save EP location for return address reserved; params can now be evaluated from STORE[SP + 1]	tab. 2.12, p. 47 Fig. 2.12, p. 42 Fig. 2.11, p. 40
cup p q	MP := SP $-(p+4)$; STORE[MP + 4] := PC; PC := q;		call user procedure	p: params storage requirement save return address proc. init. routine start address q in CODE	tab. 2.12, p. 47
!!!	$base(p, a) =$ **if** $p = 0$ **then** a **else** $base(p-1, STORE[a+1])$ **fi**: e.g. $base(d, MP)$ follows SL chain d times (Fig. 2.12, p. 42)				**!!!**

1.6.3 Instructions for Returning

instruction	meaning	condition	result	comments	references
retf	SP := MP; PC := STORE[MP + 4]; EP := STORE[MP + 3]; **if**$EP \geq NP$ **then**$error('storeoverflow')$**fi** MP := STORE[MP +2]		leaves result at top of stack	function result in local stack return branch restore EP release current stack frame revert via DL ptr	tab. 2.13, p. 48
retp	SP := MP - 1; PC := STORE[MP + 4]; EP := STORE[MP + 3]; **if**$EP \geq NP$ **then**$error('storeoverflow')$**fi** MP := STORE[MP +2]			procedure w/o results return branch restore EP release current stack frame revert via DL ptr	tab. 2.13, p. 48

1.6.4 Instructions for Calling and Entering Procedures (Callee)

instruction	meaning	condition	result	comments	references
ssp p	SP := MP + p - 1			set stack pointer p size of static part data area	tab. 2.12, p. 47
sep p	EP := SP + p; **if**$EP \geq NP$ **then** error('store overflow') **fi**;			set extreme pointer p max. depth of local stack collision check stack, heap	tab. 2.12, p. 47
!!!	$base(p, a) =$ **if** $p = 0$ **then** a **else** $base(p-1, STORE[a+1])$ **fi**: e.g. $base(d, MP)$ follows SL chain d times (Fig. 2.12, p. 42)				**!!!**

1.6.5 Block Copy Instructions

instruction	meaning	condition	result	comments	references
movs q	**for** $i := q-1$ **down to** 0 **do** STORE[SP + i] := STORE[STORE[SP] + i] **od**; SP := SP + q - 1	(a)		STORE[SP] holds begin addresss structured type (backward copying; begin address TOS overwritten)	tab. 2.14, p. 50
movd q	**for** $i := 1$ **to** $STORE[MP+q+1]$ **do** STORE[SP + i] := STORE[STORE[MP + q] + STORE[MP + q + 2] + i - 1] **od**; SP := SP + STORE[MP + q + 1]				tab. 2.14, p. 50

2 Generating P-code Instruction Sequences: Compilation Schemes

2.1 Schemata for Expression Evaluation, Assignment and Statement Sequences

	Function	($\rho : Var \mapsto \mathbb{N}_0$; maps variable to relative address)	Condition	References
1	$code_R(e_1 = e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **equ** T	type(e_1) = type(e_2) = T	p. 13
2	$code_R(e_1 \neq e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **neq** T	type(e_1) = type(e_2) = T	p. 13
*				tab. 2.1, p. 11
3	$code_R(e_1 + e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **add** N	type(e_1) = type(e_2) = N	p. 13
4	$code_R(e_1 - e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **sub** N	type(e_1) = type(e_2) = N	p. 13
5	$code_R(e_1 * e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **mul** N	type(e_1) = type(e_2) = N	p. 13
6	$code_R(e_1 / e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **div** N	type(e_1) = type(e_2) = N	p. 13
7	$code_R(-e)\ \rho$	$= code_R\ e\ \rho;$ **neg** N	type(e) = N	p. 13
8	$code_R(x)\ \rho$	$= code_L\ x\ \rho;$ **ind** T	x variable identifier of type T	p. 13
9	$code_R(c)\ \rho$	= **ldc** T c	c constant of type T	p. 13
10	$code(x := e)\ \rho$	$= code_L\ x\ \rho; code_R\ e\ \rho;$ **sto** T	x variable identifier	p. 13
11	$code_L\ x\ \rho$	= **ldc** a $\rho(x)$	x variable identifier	p. 13
12	$code$(**if** e **then** st_1 **else** st_2 **fi**) ρ	$= code_R\ e\ \rho;$ **fjp** $l_1; code\ st_1\ \rho;$ **ujp** $l_2; l_1{:}code\ st_2\ \rho; l_2{:}$	e boolean expression	p. 14
13	$code$(**if** e **then** st **fi**) ρ	$= code_R\ e\ \rho;$ **fjp** $l; code\ st\ \rho; l{:}$	e boolean expression	p. 14
14	$code$(**while** e **do** st **od**) ρ	$= l_1{:}code_R\ e\ \rho;$ **fjp** $l_2; code\ st\ \rho;$ **ujp** $l_1; l_2{:}$	e boolean expression	p. 14
15	$code$(**repeat** st **until** e) ρ	$= l{:}code\ st\ \rho; code_R\ e\ \rho;$ **fjp** l	e boolean expression	p. 14
16	$code(st_1; st_2)\ \rho$	$= code\ st_1\ \rho; code\ st_2\ \rho$		p. 15

2.2 Array-Related Schemata

	Function	($\rho : Var \mapsto \mathbb{N}_0$; maps variable to relative address)	Condition	References
17	$code_L\ c[i_1,\ldots,i_k]\ \rho$	$= \mathbf{ldc}\ a\ \rho(c); code_I\ [i_1,\ldots,i_k]\ g\ \rho$	array components of type T; $g = \text{size}(T)$ $\text{type}(i_1) = \ldots = \text{type}(i_k) = N$	p. 23
18a	$code_I\ [i_1,\ldots,i_k]\ g\ \rho$	$= code_R\ i_1\ \rho; \mathbf{ixa}\ g \cdot d^{(1)};$ $code_R\ i_2\ \rho; \mathbf{ixa}\ g \cdot d^{(2)};$ $\vdots$ $code_R\ i_k\ \rho; \mathbf{ixa}\ g \cdot d^{(k)};$ $\mathbf{dec}\ a\ g \cdot d$	array components of type T; $g = \text{size}(T)$ $\text{type}(i_1) = \ldots = \text{type}(i_k) = N$ $d = \sum_{j=1}^{k} u_j \cdot d^{(j)}$; $d^{(j)} = \prod_{l=j+1}^{k} d_l$ d_l ranges of array dimensions	pp. 22–23
18b	$code_I\ [i_1,\ldots,i_k]\ arr\ \rho$	$= code_R\ i_1\ \rho; \mathbf{chk}\ u_1\ o_1; \mathbf{ixa}\ g \cdot d^{(1)};$ $code_R\ i_2\ \rho; \mathbf{chk}\ u_2\ o_2; \mathbf{ixa}\ g \cdot d^{(2)};$ $\vdots$ $code_R\ i_k\ \rho; \mathbf{chk}\ u_k\ o_k; \mathbf{ixa}\ g \cdot d^{(k)};$ $\mathbf{dec}\ a\ g \cdot d$	array components of type T; $g = \text{size}(T)$ $\text{type}(i_1) = \ldots = \text{type}(i_k) = N$ $d = \sum_{j=1}^{k} u_j \cdot d^{(j)}$; $d^{(j)} = \prod_{l=j+1}^{k} d_l$ d_l ranges of array dimensions $arr = (g; u_1, o_1, \ldots, u_k, o_k)$	pp. 22–23
19	$code_{Ld}\ b[i_1,\ldots,i_k]\ \rho$	$= \mathbf{ldc}\ a\ \rho(b);$ $code_{Id}\ [i_1,\ldots,i_k]\ g\ \rho$	load descriptor address (static) component size g	p. 26
20	$code_{Id}\ [i_1,\ldots,i_k]\ g\ \rho$	$= \mathbf{dpl}\ i;$ $\mathbf{ind}\ i;$ $\mathbf{ldc}\ i\ 0;$ $code_R\ i_1\ \rho; \mathbf{add}\ i; \mathbf{ldd}\ 2k+3; \mathbf{mul}\ i;$ $code_R\ i_2\ \rho; \mathbf{add}\ i; \mathbf{ldd}\ 2k+4; \mathbf{mul}\ i;$ $\vdots$ $code_R\ i_{k-1}\ \rho; \mathbf{add}\ i; \mathbf{ldd}\ 3k+1; \mathbf{mul}\ i;$ $code_R\ i_k\ \rho; \mathbf{add}\ i;$ $\mathbf{ixa}\ g;$ $\mathbf{sli}\ a;$	duplicate highest stack entry (descriptor address) load fictitious start address via upper duplicate Horner scheme; retrieve and account for d_2 Horner scheme; retrieve and account for d_3 Horner scheme; retrieve and account for d_{k-1} Horner scheme; account for last term indexed address: add offset to fictitious address pop redundant address value (2^{nd} highest stack entry)	p. 26 p. 25 p. 25 p. 25 p. 25 p. 25

2.3 Record- and Pointer-Related Schemata

	Function	($\rho : Var \mapsto \mathbb{N}_0$; maps variable to relative address)	Condition	References
21	$code_L\ c_i\ v\ \rho$	$= \mathbf{ldc}\ a\ \rho(v); \mathbf{inc}\ a\ \rho(c_i)$	$\text{type}(c_i) = \text{T}$; c_i component in record v; $\rho(c_i) = \sum_{j=1}^{i-1} size(t_j)$ for sizes	p. 28
22	$code(\mathbf{new}(x))\ \rho$	$= \mathbf{ldc}\ a\ \rho(x); \mathbf{ldc}\ i\ size(t); \mathbf{new}$	if x is a variable of type $\uparrow t$ *cf.* P-instruction **new** for upper 2 stack entries	p. 30
23	$code_L(xr)\ \rho$	$= \mathbf{ldc}\ a\ \rho(x); code_M(r)\ \rho$	for name x (variable reference)	p. 31
24	$code_M(.xr)\ \rho$	$= \mathbf{inc}\ a\ \rho(x); code_M(r)\ \rho$	for name x (record field section)	p. 31
25	$code_M(\uparrow r)\ \rho$	$= \mathbf{ind}\ a; code_M(r)\ \rho$	(pointer dereferencing)	p. 31
26	$code_M([i]r)\ \rho$	$= code_{I(d)}\ [i]\ g\ \rho; code_M(r)\ \rho$	component size g of array (array indexing)	p. 31
27	$code_M(\epsilon)$	$= \epsilon$	(empty statement)	p. 31

2.4 Schemata for Procedure and Function Calls

	Function	Condition	References
29	$code\ p(e_1, \ldots, e_k)\ \rho\ st =$ **mst** $st - st'$; $code_A\ e_1\ \rho\ st$; $\vdots$ $code_A\ e_k\ \rho\ st$; **cup** $s\ l$;	st = nesting depth procedure call st' = nesting depth procedure declaration $\rho(p) = (l, st')$ s = storage requirements actual params l = CODE address of **ssp** instr. rule 28/30	p. 49 steps p. 46
28	$code$ (**proc** $p(specs);\ vdecls;\ pdecls;\ body)\ \rho\ st =$ **ssp** n_a''; $code_P\ specs\ \rho'\ st$; $code_P\ vdecls\ \rho''\ st$; **sep** k; **ujp** l; $proc\_code$; $l:\ code\ body\ \rho'''\ st$; **ret**[**f**|**p**]	st = nesting depth procedure declaration storage requirements static part storage requirements dynamic part create and initialise variables k max. depth local stack l = procedure start CODE address code for local procedures code for procedure body $(\rho', n\_a') =\quad elab\_specs\ specs\ \rho\ 5\ st$ $(\rho'', n\_a'') =\quad elab\_vdecls\ vdecls\ \rho'\ n\_a'\ st$ $(\rho''', proc\_code) =\quad elab\_pdecls\ pdecls\ \rho''\ st$	p. 49 steps p. 46
30	$code$ (**program** $vdecls;\ pdecls;\ stats)\ \rho_0 =$ **ssp** n_a; $code_P\ vdecls\ \rho\ 1$; **sep** k; **ujp** l; $proc\_code$; $l:\ code\ stats\ \rho'\ st$; **stp**	ρ_0: all registers initialised to 0; SP to -1 SP after organisational block generate code to fill array descriptors k max. depth local stack l = procedure start CODE address code for local procedures $st = 1$ nesting depth stop P-machine $(\rho, n\_a) =\quad elab\_vdecls\ vdecls\ \rho_0\ 5\ 1$ $(\rho', proc\_code) =\quad elab\_pdecls\ pdecls\ \rho\ 1$	p. 56 steps p. 46
37	$code$ (**return** $[e\|x])\ \rho\ st =$ $code_R\ [e\|x]\ \rho\ st$; **str** $T\ 0\ 0$;	T is type of expression e or variable/parameter x	

2.5 Schemata for Parameter Passing

	Function	Condition	References
31	$code_A\ x\ \rho\ st =$ $code_L\ x\ \rho\ st$;	parameter corresponding to x is **var/reference** parameter	p. 49
32	$code_A\ e\ \rho\ st =$ $code_R\ e\ \rho\ st$;	parameter corresponding to e is **value** parameter of scalar type	p. 50
33	$code_A\ x\ \rho\ st =$ $code_L\ x\ \rho\ st$; **movs** s;	parameter corresponding to e is of **structured** type t (record, descriptor) with static $size(t) = s$	p. 50
34	$code_P$(**value** x : **array**$[u_1..o_1, \ldots, u_k, .., o_k]$ **of** $t)\ \rho\ st =$ **movd** ra	copy the array	p. 52

2.6 Schemata for Access to Variables and Formal Parameters

	Function	Condition	References
35	$code_L(x\ r)\ \rho\ st =$ **lda** $d\ ra$; $code_M\ r\ \rho\ st$	$\rho(x) = (ra, st')$, $d = st - st'$ is difference in nesting depths of applied and defining occurrences **local variable** or **formal value parameter** x	p. 53 steps p. 46
36	$code_L(x\ r)\ \rho\ st =$ **lod** $a\ d\ ra$; $code_M\ r\ \rho\ st$	$\rho(x) = (ra, st')$, $d = st - st'$ is difference in nesting depths of applied and defining occurrences **formal var parameter** x (by reference)	p. 53 steps p. 46
!!!	r is a word describing indexing, selection or dereferencing: *cf.* **23 − 27**		**!!!**