

# iAPX286 Compiler Writer's Guide

By

Al Hartmann

Intel Corporation

Version #1

May 1983

© Copyright Intel Corporation, 1983

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

The following are trademarks of Intel Corporation and may only be used to identify Intel Products:

BXP, CREDIT, i, ICE, ICS, i<sub>m</sub>, iMMX, Insite, Intel, int<sub>el</sub>, Intelevison, Inteltec, iOSP, iRMX, iSBC, iSBX, Library Manager, MCS, Megachassis, Micromainframe, Micromap, Multimodule, Plug-A-Bubble, PROMPT, RMX/80, System 2000 and UPI.

## iAPX 286 Compiler Writer's Guide

### Table of Contents

1. iAPX 286 Architecture Sketch . . . . .	1-1
2. Program Addressing . . . . .	2-1
3. Data Addressing . . . . .	3-1
4. Compilation Techniques for Local Code Quality . . . . .	4-1
5. Exceptions, Errors, Debugging, and Interpretation . . . . .	5-1
6. The iAPX 286 Object Language . . . . .	6-1
7. Object Utilities . . . . .	7-1
References . . . . .	R-1

## INTRODUCTION

Intel's new iAPX-286 microprocessor is an extended version of the 8086 microprocessor (1,2) with a larger address space, virtual memory capability, multi-level protection mechanism, firmware support for multitasking, some additional instructions, and higher performance (2-3X). Other articles describe the 286 itself (3,4), operating system design for the 286 (5), and the software design rationale for the 86 family architecture (6). This ~~article~~ <sup>document</sup> describes programming language implementation for the 286, covering 286-specific compiler targeting issues, post-compilation object processing, and debugging. Section 1 sketches the 286 processor architecture from a language implementator's viewpoint, Section 2 covers program addressing, Section 3 covers data addressing, code generation is reviewed in Section 4, debugging and exception handling in Section 5, object module design in Section 6, and object utilities in Section 7.

### **\*\* 1. iAPX 286 ARCHITECTURE SKETCH \*\***

The programming language implementor views the 286 architecture through different eyes than those of an operating systems designer or of an applications programmer. The ring model protection structure and tasking support are not the driving factors here (as they might be to an operating systems implementor). Basic programming language issues are the addressing model (for programs and for data), the register set, operand forms, data types, and operator types.

### **\*\* ADDRESSING IN THE LARGE \*\***

The 286 uses a segmented, byte-addressable, virtual addressing model. The virtual address space is composed of an arbitrary number of segments each of which is of any size between one and 64 Kbytes. A segment is described by a 64-bit segment descriptor (Fig. 1.1) that tells its base address, offset limit, access rights, and also contains one word reserved for future usage (5).

Between one and 8K segment descriptors are grouped into a segment descriptor table, which is itself a segment. There may be any number of descriptor tables, since they are stored in memory as any other segment. The system virtual address space, then, may be of unlimited size, although it will be effectively limited by the number of separate processes, or tasks, in the system. The 286 processor provides each task with immediate access to one of two descriptor tables (Fig 1.2), a global descriptor table (GDT) shared by all tasks, and a local descriptor table (LDT) that is local to that particular task. Descriptor tables are protected segments modified only by the operating system. The operating system establishes the global table during system initialization, and the segment descriptor for the local table is part of the task state loaded by the processor during a context switch from one task to another. So the task virtual address space is defined by the system's GDT and by the task's own LDT, for a maximum of 8K local segments and 8K global segments (or conceivably a one-gigabyte address space).

limit (16 bits)
base (24 bits)
access (8 bits)
software reserved (16 bits)

Fig. 1.1: Segment descriptor format.

global descriptor table (GDT)

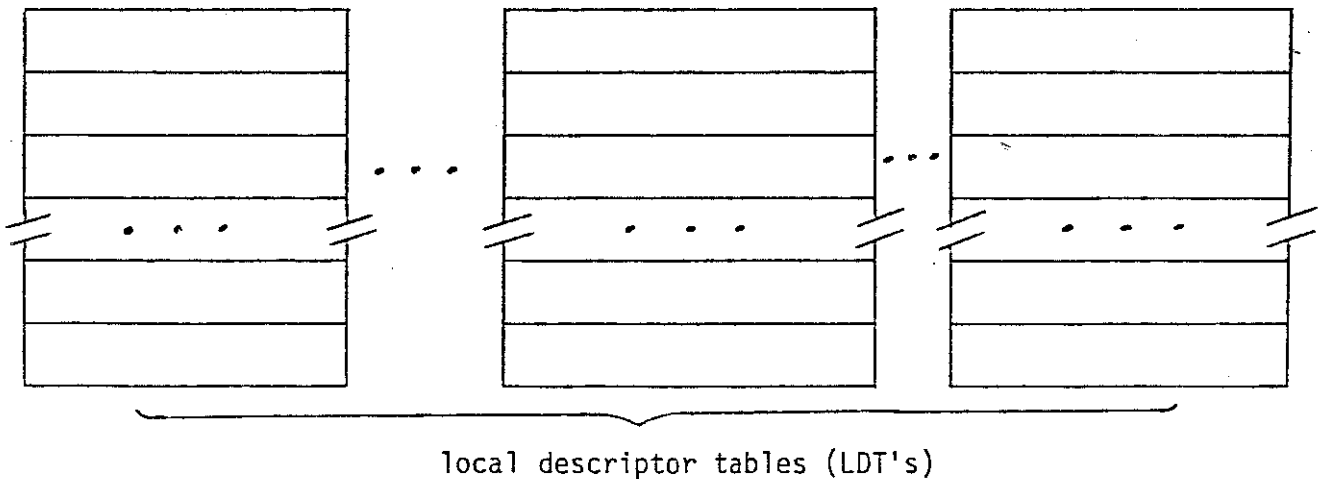
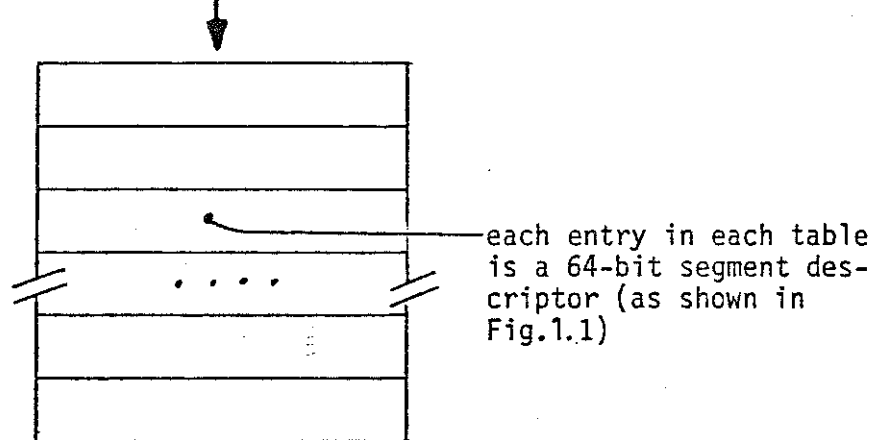


Fig. 1.2: Descriptor tables.

Since references over this enormous task address space are generally localized, the 286 processor contains four segment registers (Fig. 1.3) to hold four descriptors immediately on chip. The task itself is normally composed of program modules, each with a need to reference its code (via the code segment register, CS), its permanent data (via the data segment register, DS), its temporary data (via the stack segment register, SS), and to occasionally share other modules' data (via the extra data segment register, ES). The processor automatically loads the segment registers during context changes, or they may be explicitly loaded by program instructions. All references use one of the segment registers. Generally the appropriate segment register is implicitly known and selected by default (e.g. a branch instruction would use CS, an ordinary data reference would use DS, and a stack-relative data reference would use SS). If the default segment register choice is not the desired choice (e.g. for an inter-module reference using ES), then a one-byte segment register override prefix is added to the instruction.

Pointers or other indirect references are 32-bit virtual addresses in the form of a 16-bit segment selector and a 16-bit segment offset. The selector selects a segment descriptor from either the global or local descriptor tables for loading into a segment register. Since the iAPX 286 does not use indirect addressing, pointers must first be loaded into processor registers prior to usage (Fig. 1.4). The instructions load-data-segment-register (LDS) or load-extra-data-segment-register (LES) load a pointer into the processor registers. The pointer's selector half selects a descriptor to be loaded into one of the segment registers, and the pointer's offset half is loaded directly into a 16-bit base or index register. One bit in the selector chooses either the local or the global descriptor table, two other bits specify the ring privilege level (2), and the remaining 13 bits index among 8K possible descriptors in the table. Note that compilers generate code dealing just with selectors; only the operating system deals directly with descriptors.

Virtual-to-physical address translation (as performed in the processor chip itself) consists of adding the effective 16-bit offset generated by the instruction's addressing mode to the 24-bit base address contained in the segment register to produce a 24-bit physical address (16 Mbytes). Limit, privilege, and access rights checking are also performed (2).

## \*\* ADDRESSING MODES \*\*

The iAPX 286 shares the addressing modes of the 8086, which are extensively described elsewhere (1). Basically a multi-component address (Fig. 1.5) is used, consisting of a segment component and of base, index, or displacement components. The segment component, as already mentioned, selects one of four available segments. Figure 1.6 shows the decomposition of a software module into three segments for storage of program code, permanent data, and temporary data. The fourth segment access available is used for inter-module references or for temporary usage. A compiler processing a single module typically emits object code for the code segment, including compiled program code and constants. Initial values of global variables may be emitted for the data segment. No initial contents are generated for the stack segment. Emitted portions of all three segments from separate compilations are combined by the binder program (Sec. 7), according to object language rules (Sec. 6).

## 64-bit segment registers

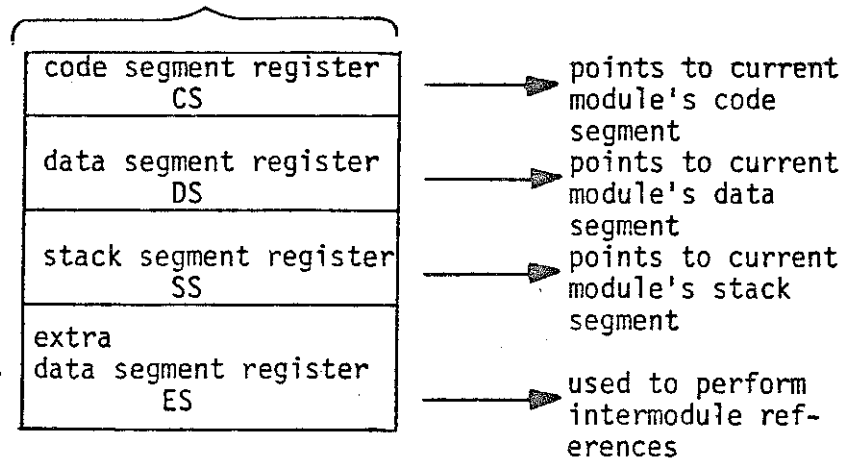


Fig. 1.3: The four segment registers.

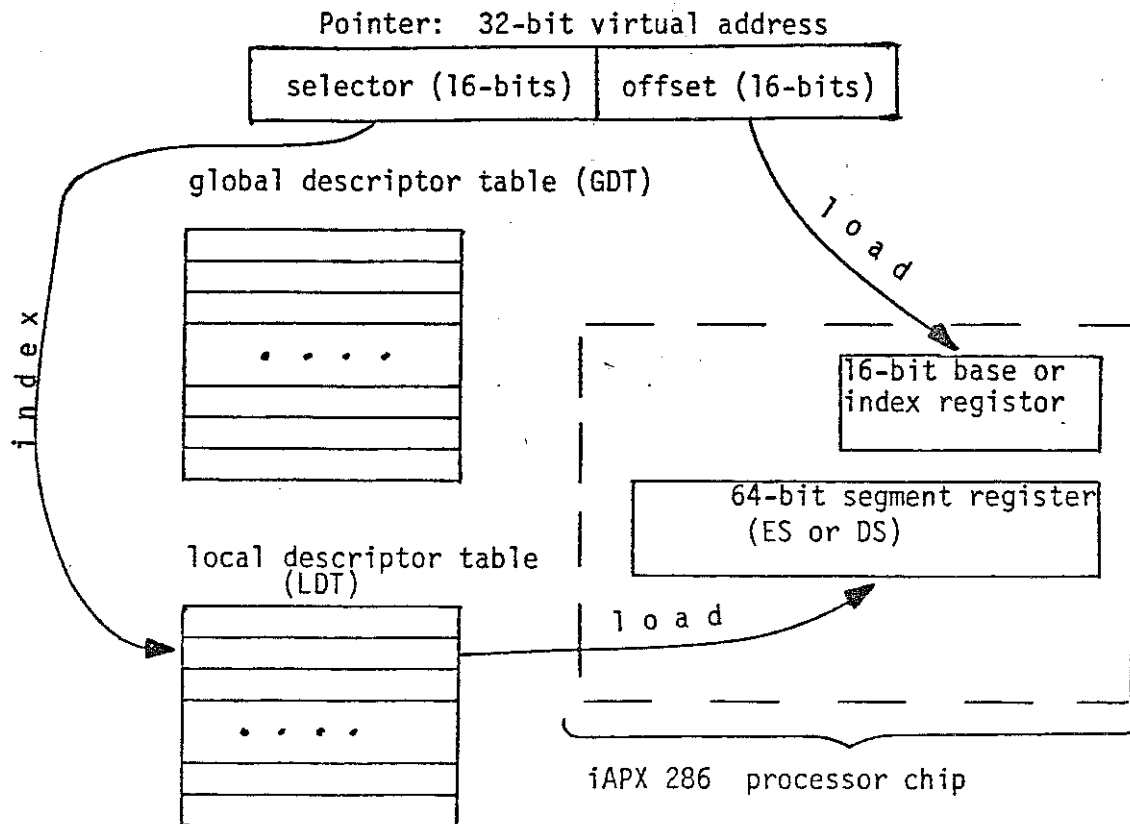


Figure 1.4: Pointer loading prior to usage.

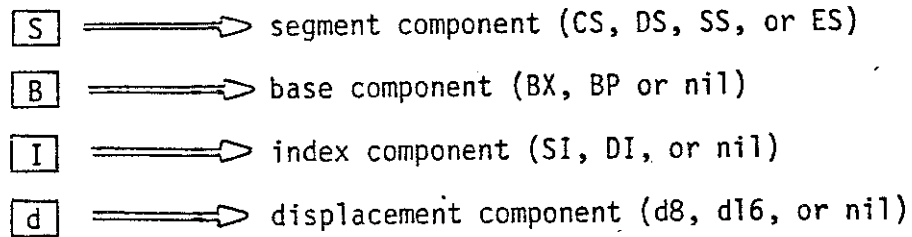


Figure 1.5: iAPX 286 multi-component addressing.

---

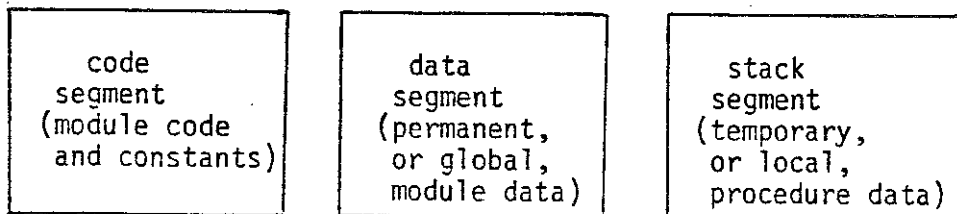


Figure 1.6: Division of a software module into segments.

---

Data references will operate in any of the segments using almost any combination of a base component, index component, and displacement component. Two sixteen-bit base registers (BX and BP) are provided, with BP usually treated as the local data base-register, since it implicitly selects the stack segment. Two sixteen-bit index registers (SI and DI) exist, and the displacement, if present, may be either eight or sixteen bits. Table 1.1 shows the address components typically selected for referencing static or dynamic scalar, vector, and record data items.

<u>Address Components</u>		
Data Type	Static Form	Dynamic Form
scalar	d	B
vector	d,I	B,I
record	d	B,d

Table 1.1: Typical address component usage.

## \*\* PROGRAM TRANSFER OVERVIEW \*\*

A transfer of control may be initiated by a jump, a call, a return, or by the occurrence of an exceptional condition (trap or interrupt). Jumps may be conditional or unconditional. Conditional jumps use an 8-bit self-relative displacement. Unconditional jumps use either an 8 or 16-bit self relative displacement, or a full 32-bit virtual address. Calls use either a 16 bit self-relative displacement or a full 32-bit virtual address. For both jumps and calls, the 16-bit or 32-bit target address may be contained in the transfer instruction itself (direct transfer) or be contained in data memory (indirect transfer) and be referenced via any addressing mode. Three return forms support 16-bit intrasegment returns, 32-bit inter-segment returns, or interrupt returns. Exceptional conditions may be invoked by external or internal events, or by soft interrupt, or trap, instructions.

Transfers of control are intra-procedural (as in local jumps), inter-procedural (as in ordinary calls), or inter-task (as in calling or jumping to a task entry point). The descriptor table entry for inter-segment transfers (selected by the 16-bit selector portion of the 32-bit virtual address) indicates whether the target is a task entry or not. An inter-segment transfer may cross protection level boundaries, and this is also indicated in the descriptor table entry selected by the virtual address. A special type of entry (called a gate) is used for protection level crossings. The topics of tasking and protection support in the iAPX 286 are treated more fully elsewhere (5).

## \*\* REGISTER SET AND OPERAND FORMS \*\*

The iAPX 286 possesses working registers, segment registers, and system and status registers. The working registers are shown in Figure 1.7, and consist of eight 16-bit registers. A companion math processing chip contains eight 80-bit floating-point registers, shown in the figure.

Instructions may take zero, one, or two operands. The two operand forms in general take one register or constant operand, and one register or memory operand, except for the string instructions, which take two memory operands. The addressing modes for referencing memory operands have already been discussed. The iAPX 286 can directly manipulate 8 or 16-bit signed or unsigned integers, while the companion math processor can operate on 16-bit, 32-bit, and 64-bit integers, and 32-bit, 64-bit, and 80-bit floating-point numbers.

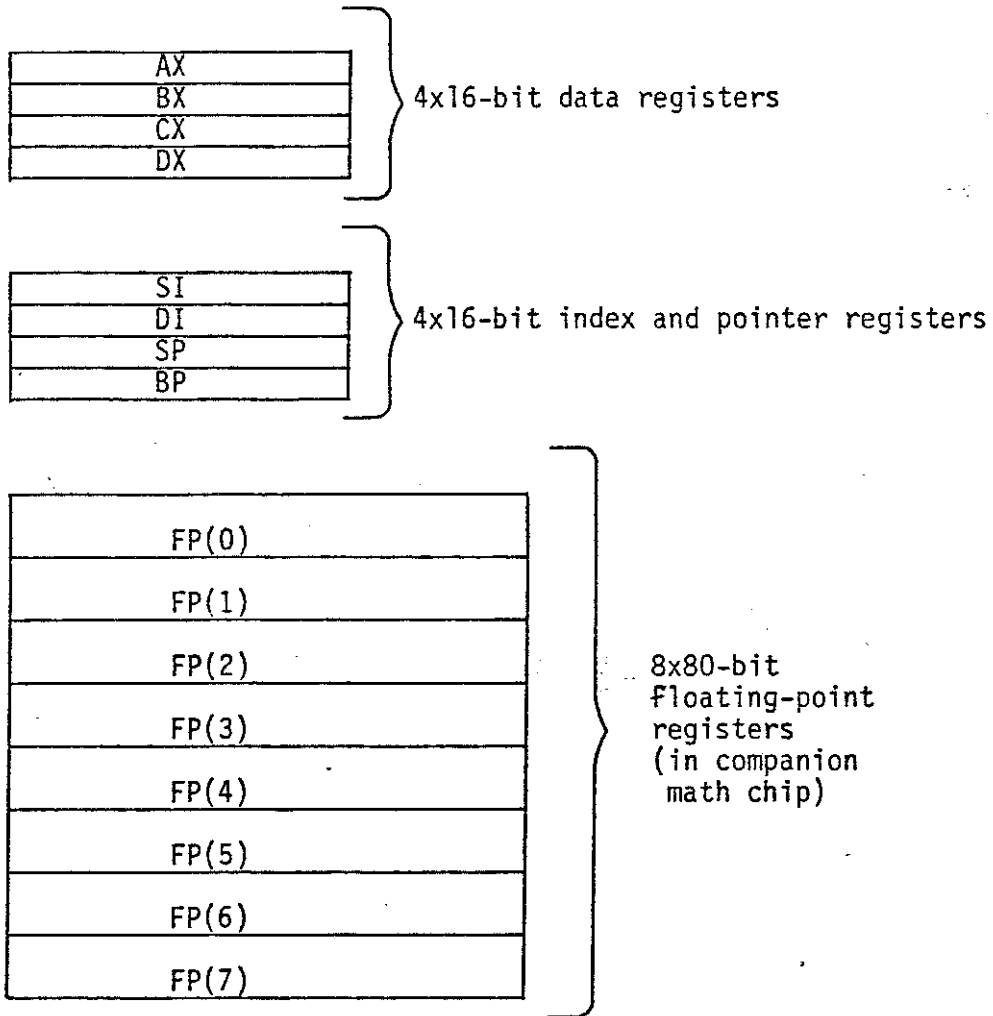


Fig. 1.7: iAPX 286 working register set and floating point register set.

## \*\* 2. PROGRAM ADDRESSING \*\*

As described in the overview, program addressing in the iAPX 286 follows the conventional pattern of conditional and unconditional branches for statements, and of call instructions for procedures and module entries. These instructions are extended by the architecture to also permit task invocation via branches or calls, rather than requiring special activation of tasks via traps or supervisor calls. Thus the processor architecture recognizes and supports inter-statement, inter-procedure, inter-module, and inter-task program transfers.

Examples of instructions generated for typical control flow statements are shown in Figures 2.1a (if-statement), 2.1b (case statement), and 2.2 (iterative loop). Note that conditional jumps use signed 8-bit displacements, and occasionally the 128-byte jump range may not be adequate. In this case the usual "jump around a jump" is employed wherein the sense of the jump condition is reversed and it jumps around a 16 or 32-bit displacement unconditional jump to the distant jump target. A simple code-generation technique for choosing between use of a normal short jump or use of a jump-around, in the case of forward branches where the displacement to the target is unknown, will be described later (Sec. 4).

Figure 2.1b shows the usual implementation of the case statement as an indexed branch. The case selector, *i*, is loaded into an index register, SI, scaled by 2, and used to index an indirect branch via the case table of addresses. Note that a single instruction, BOUND, performs bounds checking on the selector.

Figure 2.2 shows a typical iterative loop construction (this one from Ada) with a mid-loop conditional exit. Its implementation is straight forward and self-explanatory.

Figure 2.3a shows a procedure invocation with the usual argument push and call implementation. Figure 2.3b shows the procedure prologue/epilogue code, consisting of a single high-level ENTER instruction for the prologue, and a LEAVE, RETURN instruction pair for the epilogue. The PUSH instructions place the arguments on the stack, the CALL instruction deposits the return address on the stack, the ENTER instruction stores the old frame pointer (dynamic link) on the stack, copies the relevant portion of the caller's display (consisting of the base addresses of enclosing blocks' data frames and used for up-level addressing) into the local display area, adds the new frame pointer to the display, and allocates local storage for the new stack frame. The layout of the resultant stack appears in Fig. 2.4.

Where collections of data and procedures are combined into modules, as in Ada, storage for the module is allocated in one of the various program segments. This may affect the procedure prologue/epilogue for module entry procedures. Figure 2.5 shows several possibilities. In Fig. 2.5a the module is global to the entire program, so that its data is permanently allocated to the data segment for the module. So in the module entry procedure prologue the caller's DS value should be saved on the stack and the module's own DS value loaded. Then in the epilogue the caller's DS value should be restored from the stack prior to return.

Figure 2.5b shows the case of a local module whose declaration is nested within a procedure. For a small local module its data can be stored in the containing procedure's stack frame and the entry prologue/epilogue is unchanged. For a large local module, a new data segment needs to be dynamically allocated and its base address needs to be stored in a fixed location in the containing procedure's stack frame. This address then needs to be loaded into the ES register for reference to the data area prior to each such data area reference. Alternatively, the entry procedure prologue could insert the module data base address into the proper location in the local display.

One advantage of the iAPX 286 architecture is its support for tasking. The processor recognizes task descriptors (referenced in the descriptor table through branch or call instructions) and loads new task states for execution via task state segments (TSS's) shown in Fig. 2.6. A task invocation (Fig. 2.7) saves the old task state in its TSS and loads the new task state from the new TSS as a single instruction action, making rapid, realtime context switching possible.

a) if B then S<sub>1</sub> else S<sub>2</sub>

```
-- evaluate B
JF S      -- jump false to start of S2 code
-- code for S1
JMP next  -- jump to next statment
-- code for S2
-- code for next statement
```

b) Case i of  
     1: S ;  
     2, 3: S  
end

```
MOV      SI, i -- load i into index reg
BOUND    SI, case_tbl -- check 1 ≤ i ≤ 3
SHL      SI, 1 -- scale i by 2
JMP      case_tbl+2[SI] -- indexed jump thru
                                case_tbl

case_tbl: DW      1 -- lower index bound
           DW      3 -- upper index bound
           DW      @S1 -- address of S1   code
           DW      @S2 -- address of S2   code
           DW      @S2 -- address of S2   code
           -- code for S1
           JMP next -- jump to next statement
           -- code for S2
           -- code for next statement
```

FIG. 2.1: a) code generated for if-statement branching  
 b) code generated for case-statement branching

```

For i in index'range loop
    S1;
    exit when B ;
    S2;
end loop;

```

---

```

loop:      MOV      SI, index'first -- initialize SI to first value in range
          CMP      SI, index'last -- check for terminators
          JG       next -- jump to next statement if terminated
          -- code for S1
          -- code to evaluate B;
          JT       next -- jump to next statement if true
          -- code for S2
          INC      SI -- increment loop parameter, SI
          JMP      loop -- and loop
next:

```

FIG. 2.2: Code generated for iterative loop statement with exit condition

---

```

a)      P(a,b);
        -- assume a,b scalar parameters
        PUSH a
        PUSH b
        CALL P

```

---

```

b)      procedure P(x,y); . . . end;
        -- procedure prologue code:
        ENTER local_size, lex_level -- mark stack, create display
                                     -- for this lexical level,
                                     -- and allocate local storage frame
        -- procedure body code here
        -- procedure epilogue code:
        LEAVE -- cut back stack to delete
              -- local frame and display
        RET arg-size -- return to caller and remove
                    -- arguments from stack

```

FIG. 2.3: a) Procedure invocation code example  
b) Procedure prologue/epilogue code example

---

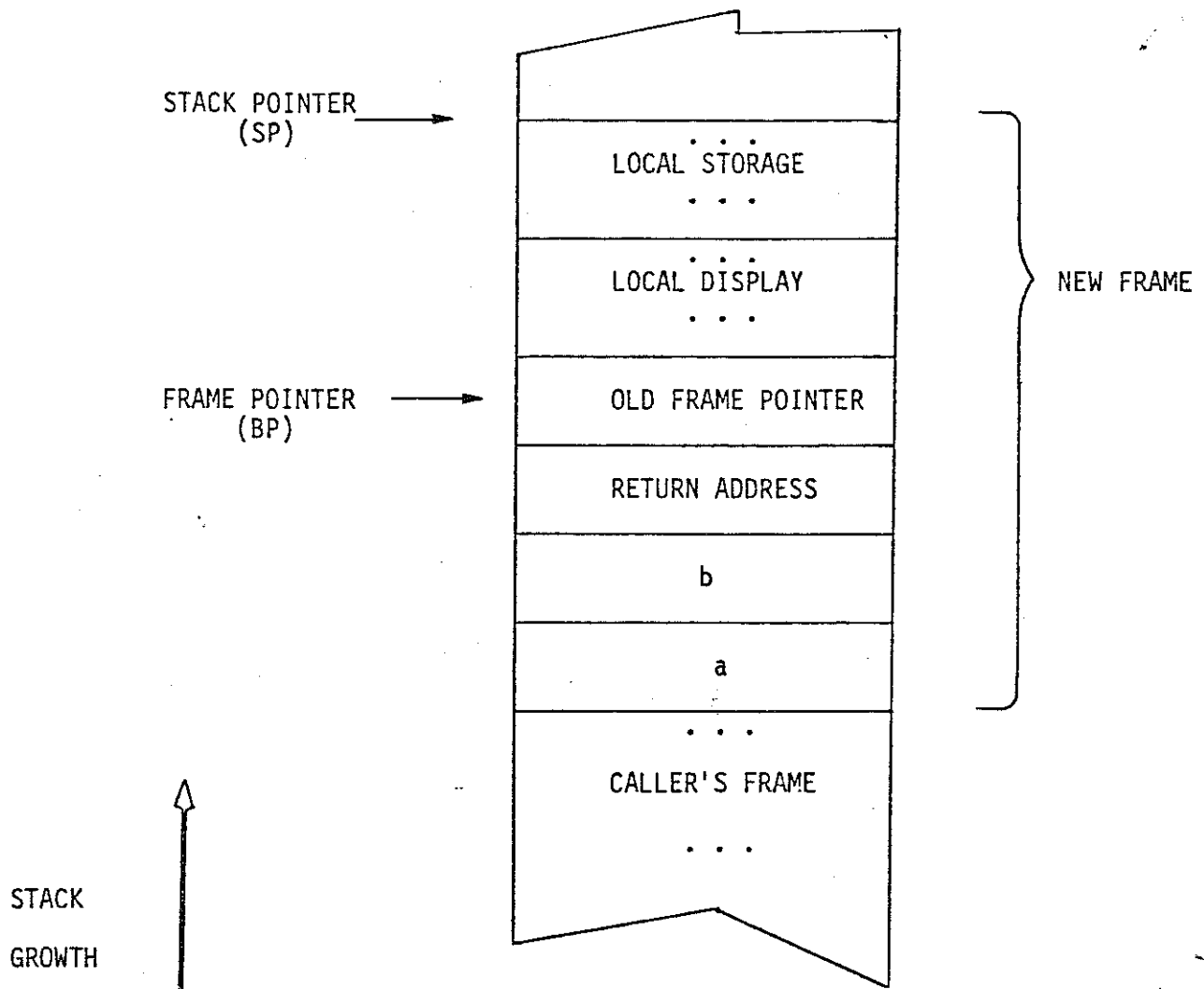


FIG. 2.4: Stack growth after procedure invocation and entry

a) global module:     package A is . . . . end;  
                           package body A is  
                               -- module data declarations  
                               . . . .  
                           end;

→ module data stored in data segment

---

b) local module:       procedure P is  
                           . . . .  
                           package B is . . . . end;  
                           package body B is  
                               -- module data declarations  
                               . . . .  
                           end;  
                           . . . .  
                           end;

→ for a small local module data area, include in stack frame storage for containing procedure;

→ for a large local module data area, create an extra segment and store pointer to it in containing procedure's stack frame.

FIG 2.5: Module data storage using Ada examples

---

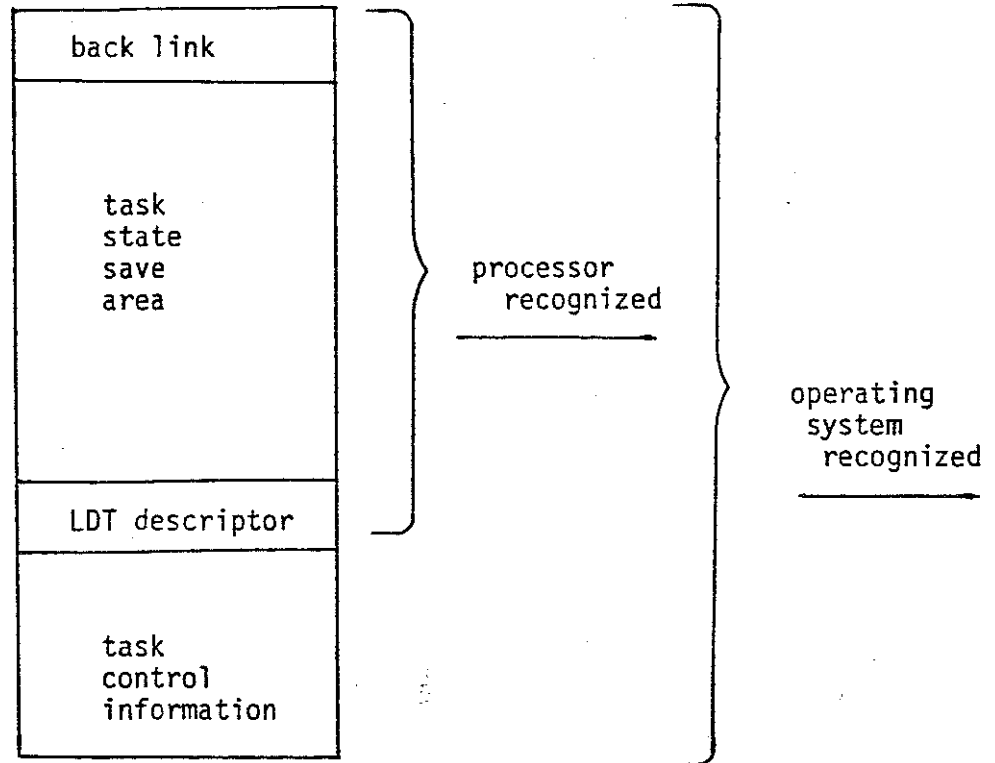


FIG 2.6: Task state segment (extended).

From call instruction

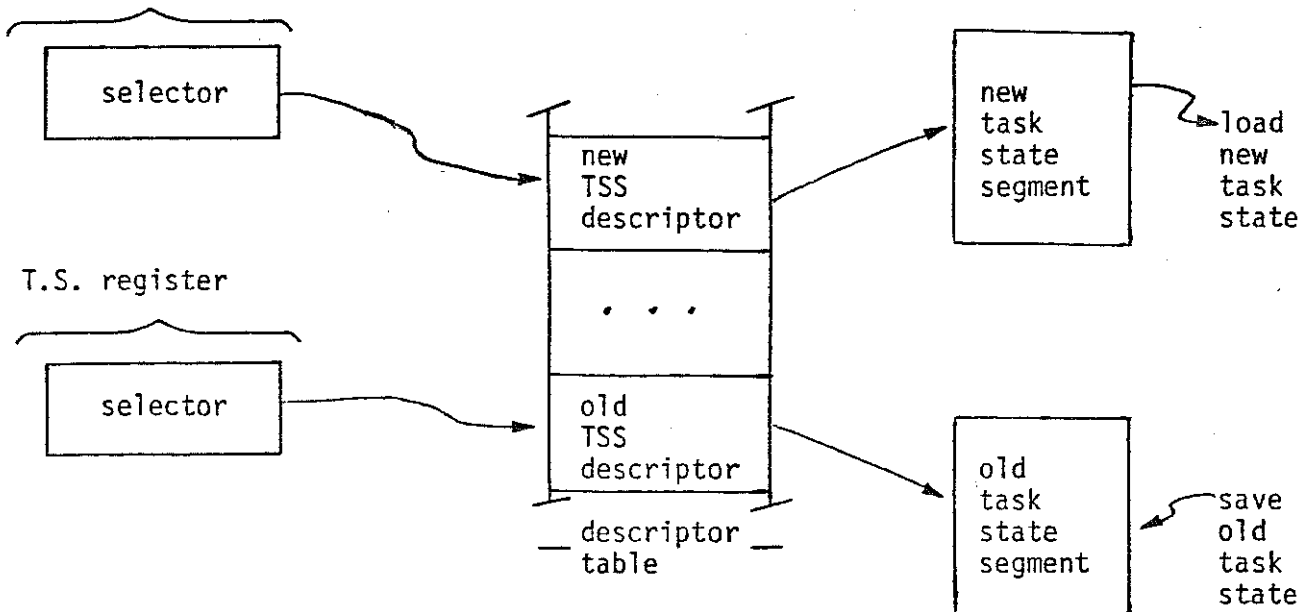


FIG 2.7: Task invocation actions

### \*\* 3. DATA ADDRESSING \*\*

As mentioned in the overview, the 286 addressing mechanism supports references to static or dynamic scalar, vector, and record data items. These utilize the addressing mode fields (1) contained in memory reference instructions to select among the various address components already shown in Figure 1.5. To give a reasonably complete set of examples of reference forms, let's study the code generated by a Pascal procedure call with a list of integer arguments passed by value as follows:

```
Proc (byte_constant, word_constant, static_scalar, dynamic_scalar,
     static_vector(i), dynamic_vector(j), static_record.field,
     dynamic_record.field, pointer^.field);
```

The two constant arguments would be passed as:

```
PUSH      byte_constant      -- two-byte instruction
PUSH      word_constant      -- three-byte instruction
```

The byte constant is contained as a single byte in the instruction, which sign-extends the constant to 16-bits before pushing it onto the stack. The vast majority of program constants fit in a byte, making automatic sign-extension worthwhile. The larger word constant is stored in two bytes, making for a three byte instruction. Instructions in the 286 are an arbitrary number of bytes in length, allowing for more compact instruction encoding than would be possible with word-aligned instruction set restrictions.

The two scalar arguments, the static(global) argument and the dynamic (local, stack-based) argument, are passed as:

```
PUSH      static_scalar      -- DS base + 16-bit offset
PUSH      dynamic_scalar[BP] -- SS base + BP frame ptr + 8 or
                             16-bit offset
```

The static scalar resides in the data segment and is reached with a 16-bit displacement, making for an instruction of four bytes. The dynamic scalar is assumed to reside in the current local data frame within the stack segment and is typically well within the range of an 8-bit displacement, making for a three-byte instruction. Since most data references in block structured languages such as Pascal, C, or Ada are generally to local variables and parameters, the bulk of these reference instructions are 3-byte instructions, again favoring the byte-aligned instruction set design.

If the dynamic scalar were stored in another data frame, belonging to an enclosing procedure, then the local display (created automatically by the ENTER instruction of the procedure prologue - Fig. 2.4) is used to indirectly reference the scalar, as follows:

```
MOV      BX, display_el      -- load other frame base into BX
PUSH     SS:dynamic_scalar[BX] -- SS base + BX frame + offset
```

Note that the appropriate frame base from the local display is loaded into the alternate base register, BX, in a single three-byte move instruction. This is more efficient than chasing down the so-called static chain (7) to find the enclosing frame base, as is done in many implementations. It also frees the registers from always containing the display, which is only infrequently referenced. The next instruction pushes the scalar on the stack using the BX base register and a stack segment override prefix (SS:) since the default would select the data segment for use with BX. The push instruction here would then typically be four bytes, or five bytes if a 16-bit displacement were needed.

The two vector arguments may either be vectors of scalars (1,2,4,8, or 10-bytes) or vectors of other data structures of arbitrary length. The 286 uses a special form of multiply instruction for indexing that disposes of the useless high-order half of the double-length product. Assuming an arbitrary vector element size, the reference code would be:

MOV	SI, i	-- 3-4 bytes
BOUND	SI, static_vector_bounds	-- " "
IMUL	SI, SI, static_vector_el_size	-- " "
PUSH	static_vector[SI]	-- " "
MOV	SI, j	-- " "
BOUND	SI, dynamic_vector_bounds	-- " "
IMUL	SI, SI, dynamic_vector_el_size	-- " "
PUSH	dynamic_vector[BP+SI]	-- " "

The instructions are either three or four bytes depending on the displacement length (8 or 16-bits). If bounds checking is disabled, the sequence can be simplified to:

IMUL	SI, i, static_vector_el_size	-- 4-6 bytes
PUSH	static_vector[SI]	-- 3-4 bytes
IMUL	SI, j, dynamic_vector_el_size	-- 4-6 bytes
PUSH	dynamic_vector[BP+SI]	-- 3-4 bytes

The multiply instructions are 4-6 bytes depending on the length of the displacements and element sizes (8 or 16-bits). Note the use of double indexing plus displacement in the last example.

The reference code for static (global) or dynamic (local) record references is identical to that for the corresponding scalar references, since the compiler will fold the field offset into the record displacement during compilation. The reference code for the record field accessed via a pointer into an extra data segment (or heap, as it is sometimes called) is:

LES	BX, pointer	-- load extra segment register/BX with pointer
PUSH	ES:field[BX]	-- reference field in record

The LES instruction is 3-4 bytes and the PUSH instruction is 4-5 bytes because of the ES override prefix.

#### \*\* 4. COMPILATION TECHNIQUES FOR LOCAL CODE QUALITY \*\*

Design of a good code generator for the 286 microprocessor can use many of the general techniques described in the numerous texts (e.g.7) and articles (8,9,10) on this subject. Also many of the usual code improvement techniques such as common sub-expression elimination, dead code elimination, constant expression folding, strength reduction, variable subsumption, code motion, etc. can be applied with any target machine architecture. The reader is assumed to be familiar with these generic aspects of code generation and improvement, so the following discussion directs itself to 286-specific aspects of good code generation. Since generation of optional code sequences is computationally impractical (8), some heuristic approaches that seem particularly suited for the 286 are described.

We assume the problem is to take as input a syntactically and semantically correct low-level intermediate representation of the source program, and convert it to 286 machine code in a particular object language format (e.g. Sec. 6). The sequence of intermediate form operators received as input is to be converted to an equivalent sequence of 286 machine operators. The mapping is in general many-to-many, since several intermediate operations may be performed in a single machine operation (e.g. a machine operation that performs complex address arithmetic for one of its operands), or conversely one complex intermediate operation may expand into several machine operations. Generation of a single machine instruction can be broken down into three steps:

- . choice of instruction form ,
- . allocation of registers,
- . emission of instruction.

This section will present heuristic, but systematic, techniques for doing these three tasks for the iAPX 286 instruction set.

#### \*\* CHOICE OF INSTRUCTION FORM \*\*

The most natural and systematic method for translating context-free input from one form to another is that of syntax-directed translation (7). This approach is developed into a code generation technique by Glanville (11) and is likely to become the method of choice in new code generator designs. The 286-specific description given here is taken largely from an earlier paper (12) describing the Pascal-86 (13) code generator.

Assume previous compiler stages have performed lexical, syntactic, and semantic analysis of the program, inserted necessary type conversions, performed code improvements, and that the program is now represented as an abstract attributed parse tree. The tree nodes' attributes are the usual semantic attributes such as variable addresses, constant values, and so on. The control constructs (if-then-else, do-while, etc.) may be represented in high-level form (i.e. not broken down into labels and jumps), but we assume expressions and assignments are represented in the input tree as low-level machine types and operators. Code improvements as mentioned earlier are assumed to be logically separate from code generation. In particular common sub-expressions are eliminated by inserting an assignment statement into the program tree and replacing redundant computations of the common sub-expression with references to the assignment target.

The method for instruction form selection to be described here will take as input one entire assignment statement or expression subtree at a time and convert it into an equivalent "code tree" which will later be traversed to emit the actual instruction sequence.

We build the code tree bottom-up by parsing the prefix representation of the expression tree with a bottom-up parser. (Either top-down or bottom-up approaches may be used, but bottom-up allows the first pass of register allocation to proceed as a co-routine to instruction form selection, as will subsequently be shown.)

Figure 4.1 shows an expression tree (a), its prefix representation (b), and the equivalent code tree (c). The translation is performed by following the rules contained in the translation grammar of the form:

$$\text{lhs} ::= \text{rhs} \Rightarrow \text{ins}$$

where      lhs is the left-hand side (typically a register non-terminal)  
              rhs is the right-hand side (typically a subexpression)  
              ins is the instruction to compute rhs into lhs, if any.

The translation grammar contains a rule for each instruction form, and non-terminals correspond to intermediate results stored in registers. The left-hand side non-terminal represents the location where the result of the right-hand side expression is computed. Non-terminals are represented in the rules by lower case names, while terminal symbols appear as upper-case names or quoted special symbols, for example:

$$\text{wordreg.0} ::= \text{WORD\_SUB wordreg.0 C.1} \Rightarrow \text{SUB R.0, C.1}$$

Application of this rule constructs a new register node in the code tree and associates it with the SUB instruction as an attribute. Other attributes associated with code tree nodes indicate register sets, constant values, and so forth. The .n suffix on grammar symbols indicate the actual associations among corresponding symbols in left-hand sides, right-hand sides, and instructions.

Another example is the rule:

$$\text{wordreg.0} ::= \text{WORD\_DEREF addr.1} \Rightarrow \text{MOV R.0, M.1}$$

which loads a register from memory. Attributes of addr.1 include the address components of base register set, index register set, and displacement. Since the left-hand side register set is not associated with a right-hand side register set, the register allocator (to be described later) will allocate a register set to the result. Register sets indicate the set of possible registers that may actually be used in the final generated code.

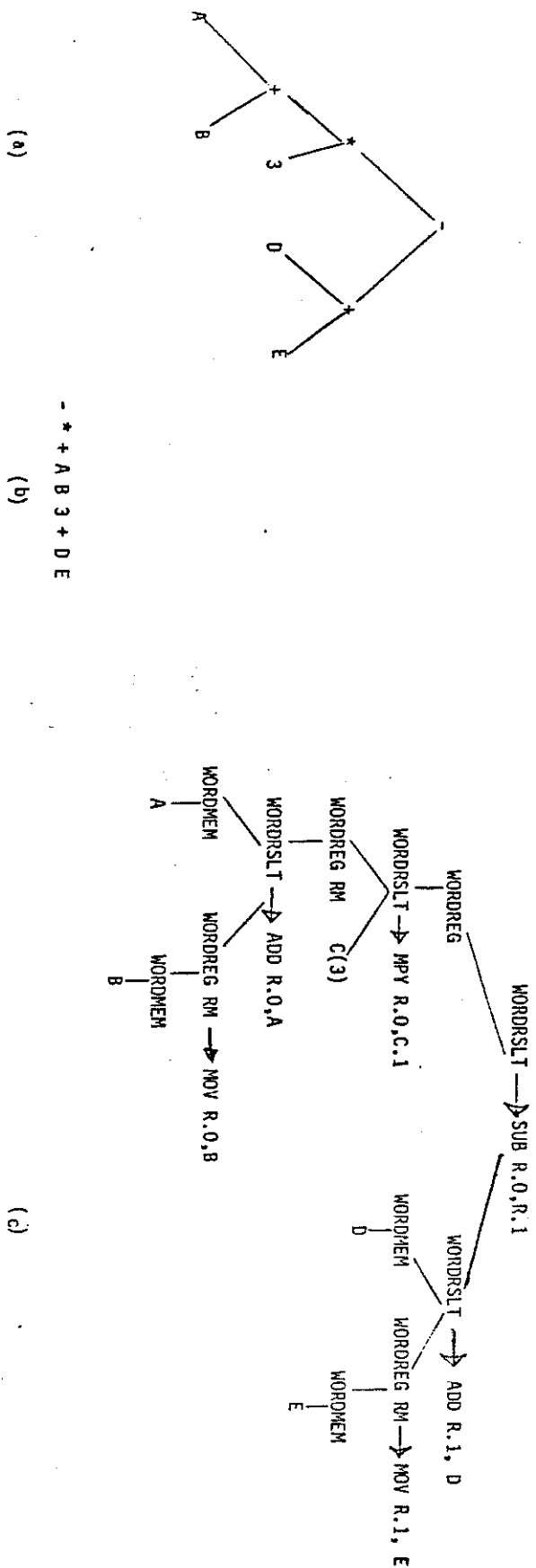


FIGURE 4.1 (a) AN EXPRESSION TREE  
(b) ITS PREFIX REPRESENTATION  
(c) ITS EQUIVALENT CODE TREE

Some things to keep in mind when designing the translation grammar are ambiguity, blocking, and cycles — the ABC's of code translation problems. Ambiguity tends to appear in the grammar because there is more than one instruction sequence to use for a given operation (e.g. 'SUB reg0, mem' or 'MOV reg1, mem; SUB reg0, reg1'). Blocking may occur if a necessary rule is missing from the grammar and the parser cannot proceed. Cycling may occur if a cycle of single productions (i.e. renamings) exist in the grammar (e.g. for coercing from one type to another). Occasionally a single-production introduced to cure a blocking problem may introduce a cycling problem.

Straightforward approaches in grammar design for code generation generally produce ambiguities. For example:

- (1) wordreg.0 ::= wordmem.1  $\Rightarrow$  MOV R.0, M.1
- (2) wordreg.0 ::= WORD\_SUB wordreg.0 wordreg.1  $\Rightarrow$  SUB R.0, R.1
- (3) wordreg.0 ::= WORD\_SUB wordreg.0 wordmem.1  $\Rightarrow$  SUB R.0, M.1

yields the ambiguity given in the preceeding paragraph where (1);(2) is equivalent to (3). It can be removed by creating a separate non-terminal to name only non-trivial (i.e. non-leaf) register values:

- (1) wordreg.0 ::= wordmem.1  $\Rightarrow$  MOV R.0, M.1
- (2) wordrslt.0 ::= WORD\_SUB wordreg.0 wordrslt.1  $\Rightarrow$  SUB R.0, R.1
- (3) wordrslt.0 ::= WORD\_SUB wordreg.0 wordmem.1  $\Rightarrow$  SUB R.0, M.1
- (4) wordreg.0 ::= wordrslt.0

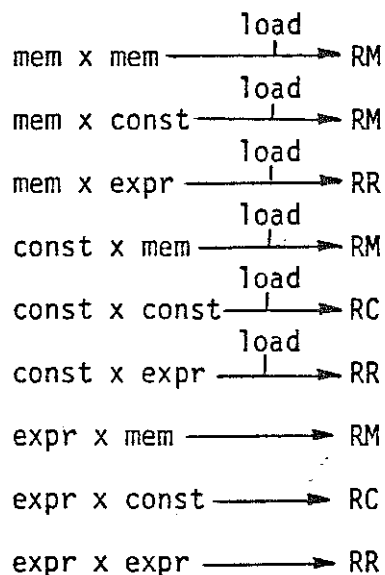
Here non-trivial register values are named "wordrslt" and may be coerced to "wordreg" values for use as lefthand operands by the single production in (4). Rule (2) is rewritten to accept only non-trivial register values for its right-hand operand. This removes the grammatical ambiguity in preference to the better (single instruction) sequence.

The full set of rules for binary operators in the 286 must take into account the nine terms in the cross-product

(mem, const, expr) X (mem, const, expr)

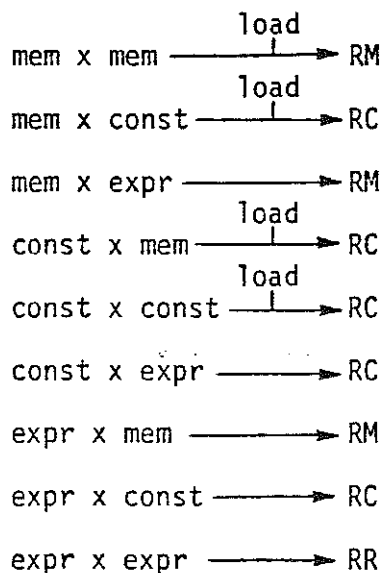
if it is to avoid blocking.

These nine terms are mapped onto three instruction forms: register-register (RR), register-memory (RM), and register-constant (RC). For non-commutative operators the left operand must be in a register, and the mapping is:



In transforming this mapping into a grammar, the first six products become loading rules leading to one of the last three products, which become computing rules.

For commutative operators the mapping is less restrictive:



The full set of rules for binary commutative and non-commutative operators ('cbop' and 'ncbop', respectively) are concisely summarized in five loading/renaming rules, three non-commutative binary computation rules, and five commutative binary computation rules, as shown in Table 4.1. The example in Figure 4.1(c) should now be fully comprehensible. Note that the rules in Table 4.1 will unambiguously: in the case of 'const x const' only load one of the constants; in a commutative 'const x mem' will load the memory operand; in a non-commutative 'const x mem' will load the constant operand; and so forth.

#### loading/renaming

```
wordreg.0 ::= wordrslt.0
wordreg.0 ::= C1  $\Rightarrow$  MOV R.0, C.1
wordreg.0 ::= wordmem.1  $\Rightarrow$  MOV R.0, M.1
wordreg RM.0 ::= wordrslt.0
wordreg RM.0 ::= wordmem.1  $\Rightarrow$  MOV R.0, M.1
```

#### non-commutative binary computation

```
wordrslt.0 ::= ncbop.1 wordreg.0 C.2  $\Rightarrow$  OP.1 R.0, C.2
wordrslt.0 ::= ncbop.1 wordreg.0 wordmem.2  $\Rightarrow$  OP.1 R.0, M.2
wordrslt.0 ::= ncbop.1 wordreg.0 wordrslt.2  $\Rightarrow$  OP.1 R.0, R.2
```

#### commutative binary computation

```
wordrslt.0 ::= cbop.1 wordrslt.0 wordmem.2  $\Rightarrow$  OP.1 R.0, M.2
wordrslt.0 ::= cbop.1 wordrslt.0 wordrslt.2  $\Rightarrow$  OP.1 R.0, R.2
wordrslt.0 ::= cbop.1 wordmem.2 wordregRM.0  $\Rightarrow$  OP.1 R.0, M.2
wordrslt.0 ::= cbop.1 wordregRM.0 C.2  $\Rightarrow$  OP.1 R.0, C.2
wordrslt.0 ::= cbop.1 C.1 wordreg.0  $\Rightarrow$  OP.1 R.0, C.1
```

TABLE 4.1 BINARY OPERATION TRANSLATION RULES

\*\* ALLOCATION OF REGISTERS \*\*

Allocation of registers is inter-related with choice of instruction forms, and logically these two operations are viewed as co-routines. One easy method to decouple the two is to assume an unlimited number of registers of each type, during instruction form selection, and then to insert register loads, stores, or moves where necessary, during register allocation and assignment. For our purposes we will allocate registers to a single code tree at a time, and actually make three passes over the code tree: (1) bottom-up as the tree (see Fig. 4.1c) is being built by the parser-driver instruction form selector, (2) top-down in a subsequent pass, and (3) bottom-up while emitting instructions. The second and third passes over the tree will actually occur in a single tree traversal utilizing both a prefix (i.e. top-down) and a postfix (i.e. bottom-up) visit to each node.

Figure 4.2 shows the register set for the iAPX 286, and clearly many of these registers have fixed purposes and are unavailable for allocation. In particular, BP and SP among the generic data registers denote the local stack frame and the stacktop, respectively. The other generic data registers may be allocated, however their usage is not fully general, and some care is required in their allocation. The usual simple schemes (e.g. 7) applicable to machines with general register organizations are inappropriate to the 286, while the more powerful graph coloring approach (14) provides a useful model although not a useful algorithm in this case.

Figure 4.3 shows a more realistic view of the allocatable registers within the 286, showing them in tree form as a nested hierarchy of register classes. The G (generic) class contains the W (working) and A (address) classes. The W class consists of the AX, CX, and DX registers, while the A class consists of the BX register and the I (index) class, the latter containing the SI and DI registers. For example a register-to-register add instruction requires two G-registers, a string operation needs the two I-registers, a doubly-indexed reference needs the BX-register and one I-register, byte operations use the W-registers, and so forth. The representation in Figure 4.3 is not optimal (BX should be either a W- or A-register, for instance), but it is practical, as will be shown.

Now, in the textbook allocation algorithm (7) for a general register machine, the code tree nodes would each contain a single integer attribute (sometimes called the Sethi-Ullman number(15)) specifying the number of registers needed to evaluate the node. Clearly this is a synthesized (i.e. derived bottom-up) attribute set on the first pass (up) the code tree. We do the analogous function, but instead of using a single integer, we use an integer vector that represents the number of registers required from each class, including the singleton classes composed of one specific register each. We call this a class vector, and it represents the number of registers needed in each class (but not more specific classes, i.e. it is non-cumulative). An arbitrary order for the elements of the vector might be [AX, CX, DX, BX, SI, DI, W, I, A, G]. Then the vector [1,0,0,0,0,0,0,0,1,1], for example, indicates three registers: AX, one address register, and one other generic register. The components of a given vector, X, are referenced as X.AX, X.BX, X.I, etc. and in our example X.AX, X.A, and X.G are all one, while all other components are zero. A more readable way of writing the vector [1,0,0,0,0,0,0,0,1,1] is as [AX,A(1), G(1)], and we will follow this convention. Note that our representation is not unique, since, for example, [BX,I(2)], [BX,SI,I(1)] and [A(3)] are three equivalent representations of the three address registers. This will not affect the allocation.

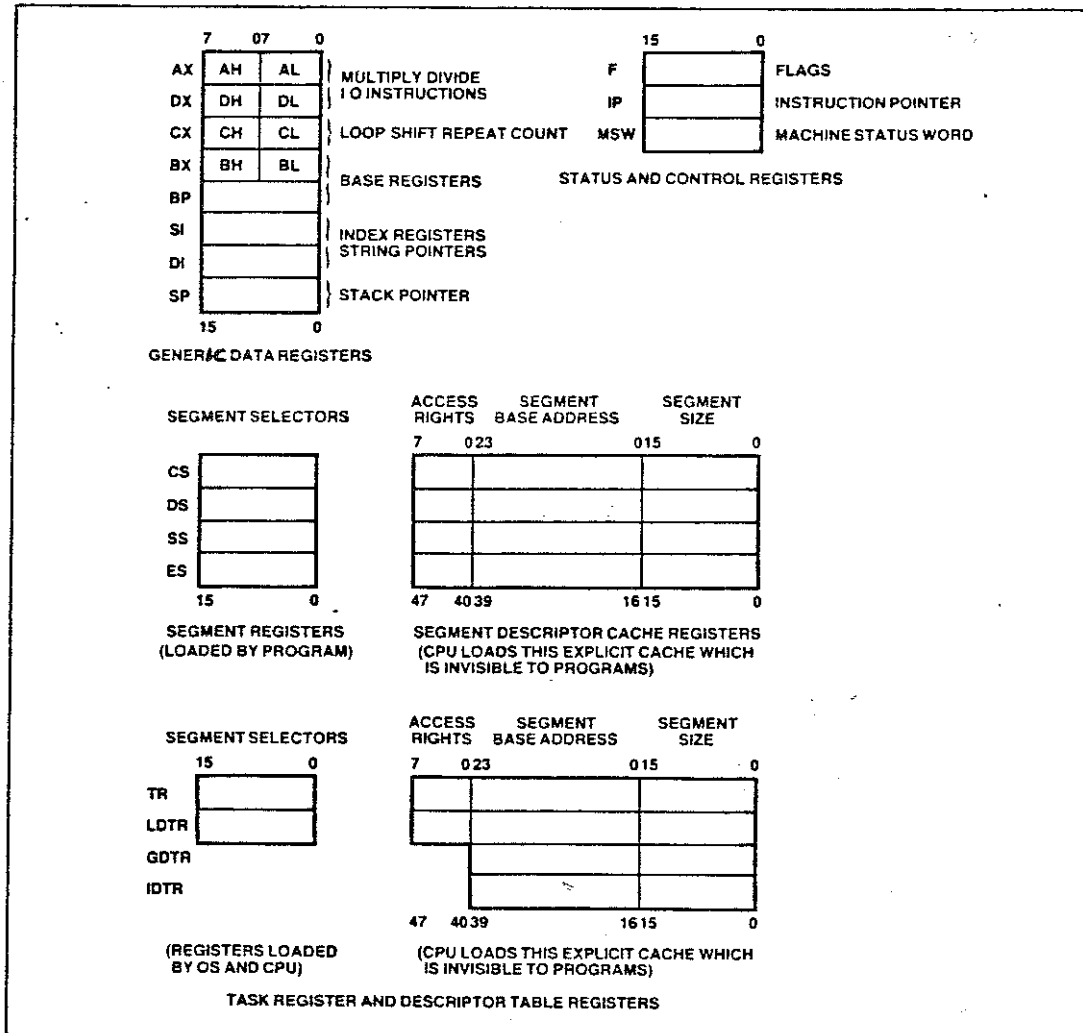


Figure 4.2: iAPX 286 register set

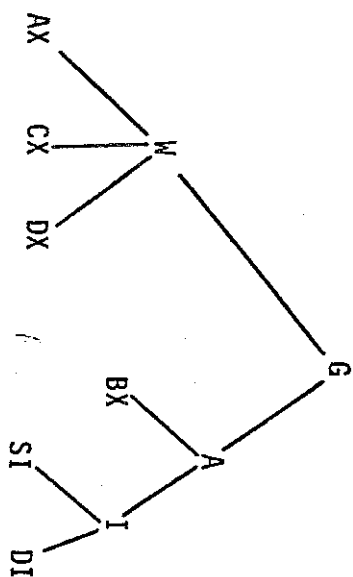


FIGURE 4.3 iAPX 286 REGISTER CLASS HIERARCHY

For any given class vectors, X and Y define the operations:

- i)  $|X|$  : the cardinality of X, equal to the sum of the elements of X;  
 ii)  $X \vee Y$ : the "minimax" join of X and Y, defined as the class vector Z with minimum cardinality but maximum generality (i.e. registers placed at highest possible levels in the hierarchy), such that either X or Y could be allocated from Z, examples:

$$\begin{aligned} [AX, W(1)] \vee [CX, W(1)] &= [AX, CX] \\ [W(1)] \vee [A(1)] &= [W(1), A(1)] \\ [AX] \vee [W(1), G(1)] &= [AX, G(1)] \end{aligned}$$

algorithm:

$$\begin{aligned} Z.r &:= \max(X.r, Y.r), \text{ for } r \in [AX, CX, DX, BX, SI, DI] \\ Z.W &:= \max(0, \max(X.W+X_w, Y.W+Y_w) - Z_w); \\ Z.I &:= \max(0, \max(X.I+X_i, Y.I+Y_i) - Z_i); \\ Z.A &:= \max(0, \max(X.A+X_a, Y.A+Y_a) - Z_a); \\ Z.G &:= \max(0, \max(X.G+X_g, Y.G+Y_g) - Z_g); \end{aligned}$$

where for R any of the X, Y, or Z we define

$$\begin{aligned} R_w &= R.AX + R.CX + R.DX; \\ R_i &= R.SI + R.DI; \\ R_a &= R.BX + R_i + R.I; \\ R_g &= R_w + R.W + R_a + R.A; \end{aligned}$$

- iii)  $X ? Y$ : the non-interference predicate, true if registers can be assigned disjointly to X and Y, false otherwise;  
 examples:

$$\begin{aligned} [AX] ? [W(2)] &= \text{true} \\ [W(2)] ? [W(2)] &= \text{false} \\ [G(4)] ? [W(3)] &= \text{false} \end{aligned}$$

$$\begin{aligned} \text{algorithm: true if } & X.r + Y.r \neq 2 \text{ for } r \in [AX, CX, DX, BX, SI, DI] \\ & \text{and } X.W + X_w + Y.W + Y_w \leq 3 \\ & \text{and } X.I + X_i + Y.I + Y_i \leq 2 \\ & \text{and } X.A + X_a + Y.A + Y_a \leq 3 \\ & \text{and } X.G + X_g + Y.G + Y_g \leq 6 \end{aligned}$$

- iv)  $X + Y$  the vector sum of X and Y, defined only if  $X ? Y$  holds, and equal to the class vector Z defined simply as the element-wise sum of class vectors X and Y;  
 examples:

$$\begin{aligned} [AX] + [W(2)] &= [AX, W(2)] \\ [W(1), I(1)] + [W(1), I(1)] &= [W(2), I(2)] \\ [AX] + [AX] &= \text{undefined, due to interference in AX.} \end{aligned}$$

- v)  $X / Y$ : the restriction of  $X$  to  $Y$ , being a class vector,  $X'$ , such that each register represented in  $X$  is also represented in  $X'$  and in  $Y$  at as general a level as  $X$  and  $Y$  will permit; in general, more than one restriction is possible and the implemented algorithm arbitrarily defines a choice among them; in our usage we will consider the restriction to have failed if  $|X| > |X'|$ ;

examples:

```

[AX]/[BX] = [ ] (failed)
[AX]/[W(2)] = [AX]
[BX]/[AX,CX,BX] = [BX]
[G(1)]/[AX,CX,BX] = [AX] (or [CX] or [BX]);
[A(2)]/[G(1),I(1)] = [A(1),I(1)]

```

algorithm: create a temporary class vector called 'balance' (which will contain the "bank balance", either positive or negative number of registers, at each level of the hierarchy); treat each class vector as though it were the tree of Fig. 4.3

```

match(G);
match(s); adj(s,G) for s ∈ (W,A)
match(r); adj(r,W), adj(r,G) for r ∈ (AX,CX,DX)
match(s); adj(s,A), adj(s,G) for s ∈ (BX,I)
match(n); adj(r,I), adj(r,A), adj(r,G) for
           r ∈ (SI,DI)

```

where:

match(s) is:

```

| X.s := min(X.s, Y.s);
| balance.s := Y.s - X.s;

```

and

adj(s,s') is (in guarded command form);

```

do
  balance.s < 0 and
  balance.s' > 0 ⇒
    increment (balance.s);
    decrement (balance.s');
    increment (X'.s);
  || balance.s > 0 and
  balance.s' < 0 ⇒
    decrement (balance.s);
    increment (balance.s');
    increment (X'.s);
od

```

Note:

the order of execution of the algorithm for vector elements at the same hierarchy level arbitrarily chooses among the family of possible results.

Given these operations it is now possible to define register allocation and register assignment as algorithms employing sequences of these fundamental operations. As already mentioned, three passes over the code tree (build by the parser driven instruction form selector) are performed. The first, bottom-up, pass occurs as the tree is built and basically computes the class vectors as synthesized attributes of the tree. The second, top-down, pass sets the target path (16) as an inherited attribute from higher-level to lower-level nodes on the way down the code tree. The third, bottom-up, pass will then emit instructions (using a template mechanism, to be described). This can be expressed (in guarded command form) as:

```

do
    →end of program ⇒ build code tree and compute
                        class vectors
                        set target paths and emit
                        instructions

```

```
od
```

Each of these operations will now be explained:

#### "BUILD CODE TREE AND COMPUTE CLASS VECTORS"

This operation involves the parser driven instruction form selector and the register allocator as coroutines. A code tree is composed of the operators and operands for a single expression. This first pass can be expressed as:

```

do
    →end of expression ⇒ build node (root, left, right)
                        match targets to preferences
                        choose evaluation order
                        choose target path

```

```
od
```

The 'build node' routine is performed by the instruction form selector which selects a machine operator and installs it as the 'root' node for the operation, with associated 'left' and 'right' operand subtrees. Instruction form selection has already been described and the tree building is a simple utility operation.

Once the new node is entered in the code tree, it is necessary to match the possible target registers of its operand subtrees with the operand location preferences of the new operator. For example the result of a sub-computation may need to be placed in an index register in order to be used in an indexed address.

The algorithm is:

```

"Match targets to preferences"

for operand:= left, right =>
  if root expects operand in a register =>
    operand.target:= operand.target/root.operand.preference;
    if |operand.target| = 0 "failed" =>
      insert reg-reg move (operand, root)
    fi
  fi
rof

```

Simply stated, the subtree target registers are restricted to the root's operand location preferences. Failing this, a register-to-register move is inserted from the subtree target vector to the root preference vector. For example a multiply instruction could not restrict its target to the I class, but would have to use a move instruction from the AX register target to the I class register preference.

Choosing an evaluation order is important to efficient expression evaluation since it may determine whether or not registers must be "spilled" to memory during the course of evaluation. The general approach is to evaluate the more complex (register consuming) subtree first, so that it would have the most registers available to it. Hopefully then the less complex subtree can be entirely computed in the remaining registers not occupied by the first subtree's result. The algorithm to choose an evaluation order has several stages. The first is:

```

"choose evaluation order - Stage 1"

left.usage := left.subuse v (left.target + left.work);
right.usage := right.subuse v (right.target + right.work);
left.may_be_first := left.target ? right.usage;
right.may_be_first := right.target ? left.usage;

```

This simple sequence computes each operand's 'usage' vector as the join of its subtree's usage vector, 'subuse', and the sum of its 'target' and 'work' vectors. A work vector consists of working registers used in complex macro instructions or of registers clobbered as a side effect of the computation (e.g. DX in multiply or divide). Of course target and work vectors do not interfere. An operand 'may be first' in evaluation order if its target vector does not interfere with its companion's usage vector.

"choose evaluation order - stage 2"

```

if left.may_be_first and right.may_be_first =>
  if [left.subuse v (left.target+(left.work v right.usage)) | <=
      |right.subuse v (right.target+(right.work v left.usage))]| =>
    root.order := left_first
  [] otherwise root.order := right_first
fi
[] left.may_be_first and ¬right.may_be_first => root.order:= left_first
[] ¬left.may_be_first and right.may_be_first => root.order:= right_first
[] ¬left.may_be_first and ¬right.may_be_first => "must spill"
  root.order := right_first; spill(right)

```

fi

This simply checks the cases: 1) both orders possible implies use the order with the lowest register usage; 2) either order possible implies use that order; 3) neither order possible implies spill an operand to memory. This is the classic textbook algorithm (7).

"choose evaluation order - stage 3"

```

if root.order=left_first =>
  root.subuse := left.subuse v (left.target + (left.work v right.usage));
  left.interference := left.interference v (right.usage v root.work);
  right.interference := right.interference v root.work
[] root.order=right_first =>
  root.subuse := right.subuse v (right.target + (right.work v
    left.usage));
  right.interference := right.interference v (left.usage v root.work);
  left.interference := left.interference v root.work

```

fi

This establishes the root subusage depending on the order of evaluation and additionally defines an 'interference' vector for the first operand that is the join of the usage vector of the second operand with the work vector of the root. Notice that the first operand target vector need not be included in the second operand interference vector since these registers will already be marked as assigned when it is time to assign registers to the second target vector. Interference vectors are used by register assignment (in the later pass) in actually making assignments from the target vector while avoiding registers in the interference vector. For example the first operand may target [W(1)] while the second operand may use [AX]. Register assignment will assign a work register other than AX.

[Digression: The graph coloring approach to register allocation (14) actually constructs an interference graph. This is a graph in which nodes represent expression operands and arcs represent interferences, i.e. the two operands joined by an arc may not occupy the same register. Register assignment is a "coloring" of the graph where each color represents one of the machine registers and no two nodes connected by an arc may be the same color. Figure 4.4 gives a code tree and its corresponding interference graph. The basic rules are that (a) sibling targets interfere, (b) first operand targets interfere with second operand usages, and (c) non-commutative operation results interfere with their second operand target. The interference vectors constructed in the first pass handle interference types (a) and (b), while first pass target path selection handles type (c) interferences in our approach. End of digression.]

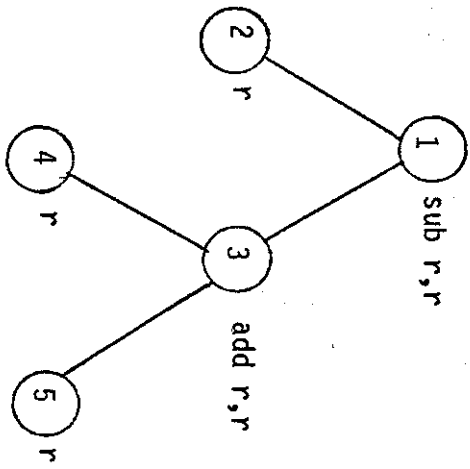
The only remaining operation in the first pass, assuming the root lies on the target path, is to choose a target path, i.e. which of the operand result targets should become the root result target. In the case of a commutative operator, the operand target with the largest cardinality is chosen for the root target:

"choose target path"

```

if root.path_continues  $\Rightarrow$ 
  if root.commutative and  $|left.target| < |right.target| \Rightarrow$ 
    exchange (left, right, root)
  fi
  root.target := left.target; root.interference := left.interference
fi

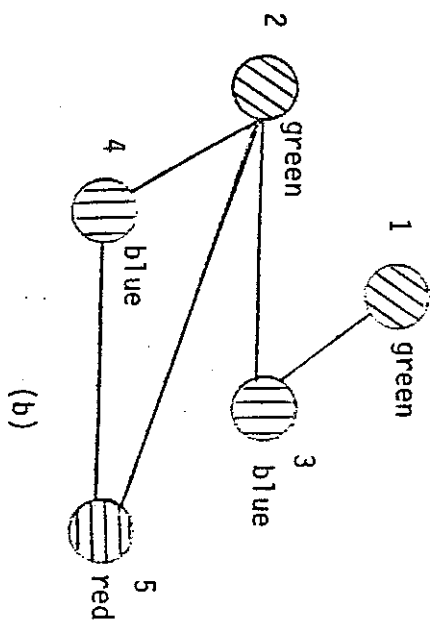
```



(a)

"red-R0, blue-R1, green-R2"  
 add R1, R0  
 sub R2, R1

(c)



(b)

FIGURE 4.4 a) CODE TREE  
 b) CORRESPONDING INTERFERENCE GRAPH  
 AND A VALID COLORING  
 c) AN ASSOCIATION OF REGISTERS TO COLORS

# \*\* THE SECOND PASS \*\*

The second, top-down, pass is a good deal simpler than the first pass. In the first pass the bottom-up matching of target and preference vectors had generally refined the target vectors to more specific registers. This information (target and interference vectors) must now be passed back down the code tree along the target paths to the leaves, where the registers actually get loaded. Not all machine operations continue the target path (e.g. comparisons or moves), so the 'path continues' Boolean attribute is specified for each machine operation. The inheritance of target paths and emission of instructions is a top-down traversal of the code tree, passing down the target and interference vectors on the preorder visit to each interior node, and emitting the instruction on the postorder visit. It is defined recursively as follows:

"set target paths and emit instructions"

set\_and\_emit (code\_tree.root)

where we define it as:

"set and emit (n: code tree node)"

```

if      n.leaf ⇒ emit_instruction(n)
[] → n.leaf ⇒ if n.path_continues ⇒ n.left.target := n.target;
                                   n.left.interference :=
                                   n.interference
fi

if n.order = left_first ⇒ set_and_emit (n.left);
                        set_and_emit (n.right)
[] n.order = right_first ⇒ set_and_emit (n.right);
                          set_and_emit (n.left)
fi
emit_instruction(n)
  
```

The actual instruction emission is defined as:

"emit instruction (n: code tree node)"

```

if → n.leaf and n.path_continues ⇒ n.target := n.left.target;
                                   assign_regs (n.work,
                                   n.interference);
                                   free_regs(n.right.target)
[] → n.leaf and → n.path_continues ⇒ assign_regs(n.work,
                                   n.interference);
                                   free_regs(n.left.target);
                                   free_regs(n.right.target);
                                   assign_regs(n.target,
                                   n.interference)
[] n.leaf ⇒ assign_regs(n.target, n.interference);
              assign_regs(n.work,
              n.interference)
fi
expand_template(n)
free_regs(n.work)
  
```

This algorithm treats three separate cases. In the first case, for an interior node that continues the target path, the target vector is passed up from the left (target path) operand, and the right operand target is freed. In the second case, for an interior node that breaks the target path, work registers are assigned, both operand target vectors are freed and the new target assigned. Finally, for the leaf node case, the new target is simply assigned. Following this the code pattern is emitted using a template expansion technique (17), and the work registers are freed.

Assigning and freeing registers uses the same basic class vector operations as the other register selection routines. The set of free registers, 'free set', is maintained as a class vector all of whose elements are specific (singleton class) registers. Note that the previous register allocation operations have allocated possibly non-specific registers to the various target sets and verified that they do not necessarily conflict with the corresponding interference sets. Therefore register assignment is guaranteed to find a specific target vector such that:

- (i)  $| \text{target/free set} | = | \text{target} |$
- (ii)  $\text{target} ? \text{interference holds.}$

This means there is absolutely no "on the fly" spilling of registers during register assignment; all essential spills were performed earlier, during the choice of an evaluation order for each operator's operands. This is called a "pre-planning" strategy in the literature (9).

It would be nice if we could just set target to be target/free set, but since there is more than one possible restriction of the target vector to the free set (some of which may conflict with the interference set) we must plod through, trying a register at a time:

"assign regs (target, interference)"

```

assigned:=0; free-temp:= free_set;
do    |assigned| < |target| =>
    reg:=pick_one(free-temp); try_target := target v reg;
    if |try_target| = |target| and try_target ? interference =>
        target := try_target; assigned := assigned+1
    fi
    free_temp := free_temp - reg;
od
free_set := free_set - target

```

The class vector difference operator, -, used here is of course element-wise subtraction. This algorithm just tries the free registers one at a time and joins any that satisfy the criteria into the target set, thus making that register entry specific. (The bottom-up character of the join operator guarantees that once a specific register has been assigned to the target vector, we will never have to back up and try another assignment.) The companion routine to free registers is of course:

"free regs (target)"

```
free-set := free_set + target
```

**\*\* INSTRUCTION EMISSION \*\***

A table-driven, code template mechanism (17) is best used with the 286 to organize the mass of detail associated with actually emitting the instruction once the machine operator and registers have been chosen. It would handle such details as:

- the size of the displacement used to reference a memory operand;
- the size of any immediate operands;
- special instruction forms that may be useable because of the register involved (often the case with AX);
- management of the current stack frame size and its high water mark;
- management of the current code segment size;
- association of compiler-generated labels with code segment offsets;
- association of the current condition code setting with the responsible instruction;
- management of jump and call displacements (i.e. span-dependant instructions);
- assembly of the instruction itself into the appropriate bit pattern with opcode, register, mode, displacement, etc. fields.

Each machine operator would have an associated table entry that would contain fields that are interpreted by the "template expander" to manage the myriad details listed above and emit the actual instruction patterns.

Techniques (18,19) exist for generating near optimal code for span-dependant instructions (such as jumps or calls), since this is in general an NP-complete problem. One simple technique is to just use a buffer capable of holding the equivalent of 127 bytes of generated 286 code. This can be used to resolve forward jumps into either short or long form: into short form if the jump target is in the buffer, into long form otherwise. This technique associates worst-case displacements (assuming all long jumps and calls) with the compiler-generated labels, so it may sometimes use a long form where a short form could be used. This is generally not significant, but one or more optimizing passes could be added to iteratively reduce the program to its optimal form, and usually a second iteration is sufficient.

**\* EXAMPLES \***

Let's tackle some examples now that show most of the register allocation mechanism in action. We'll use the assignment:

$A[I] := (B+C*D) - (E/F+G[J*K])$  (Example 1)

for our first example, and walk through the first (bottom-up) pass that builds the code tree. This code tree building is directed by the parser-driven instruction form selector described earlier. As it specifies each Node, the register allocation attributes are computed according to the previously given algorithms, which are summarized in terser notation in Figure 4.5. We will use this notation as we step through the construction of the code tree as driven by the bottom-up parser.

NOTATION:

n. . . . the new (parent) node  
 c. . . . any of its child nodes  
 t. . . . target  
 s. . . . subuse  
 w. . . . work  
 i. . . . interference  
 p. . . . preference for child c \*\*  
 cl. . . . the left child  
 cr. . . . the right child  
 c1. . . . the child to be evaluated first \*  
 c2. . . . the child to be evaluated second \*  
 o. . . . evaluation order  
 path. . . path continues predicate

\*Note: Evaluation order (c1, c2) is orthogonal to conceptual operand order (cl, cr). For commutative operators, operand order may be reversed if  $|cl.t| > |cr.t|$ , changing the target path.

\*\*Note: The c subscript may be omitted when the child is understood.

ATTRIBUTE COMPUTATION:


```

Vc (c.t := c.t/n.pc; c.u := c.s v (c.t + c.w))
n.o := "compare |cl.s v (cl.t + (cl.w v cr.u))| with
        |cr.s v (cr.t + (cr.w v cl.u))|
        and take best or only order or spill"
n.s := cl.s v (cl.t + (cl.w v c2.u));
cl.i := cl.i v (c2.u + n.w);
c2.i := c2.i v n.w;
"path selection, set l and r for commutative op on
        target path to choose biggest target"

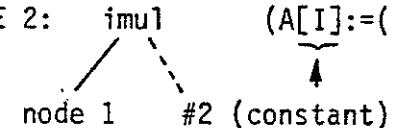
path ⇒ n.t := cl.t; n.i := cl.i
  
```

Figure 4.5: First-pass register allocation attribute computation.


(Initial attribute values are a function of the instruction operator (a constant table lookup), and computed attribute values are the result of the computation of Figure 4.5.)

NODE 1: load  $(A[I] := (B+C*D) - (E/F+G[J*K]))$   


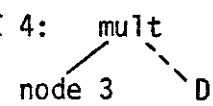
initially:  $n.t = [G(1)]; n.w = [ ]; n.i = [ ]$   
 compute:  $n.s := [ ];$

NODE 2: imul  $(A[I] := (B+C*D) - (E/F+G[J*K]))$   


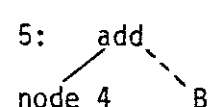
initially:  $n.t = [G(1)]; n.p = [G(1)]; n.w = [ ]; n.i = [ ]$   
 compute:  $c.t := [G(1)]; c.u := [G(1)];$   
 $n.s := [G(1)]; c.i := [ ];$   
 $n.t := [G(1)]; n.i := [ ];$

NODE 3: load  $(A[I] := (B+C*D) - E/F+G[J*K])$   



initially:  $n.t = [G(1)]; n.w = n.i = [ ]$   
 compute:  $n.s := [ ];$

NODE 4: mult  $(A[I] := (B+C*D) - (E/F+G[J*K]))$   


initially:  $n.t = [AX]; n.w = [DX]; n.p = [AX]; n.i = [ ]$   
 compute:  $c.t := [AX]; c.u := [AX];$   
 $n.s := [AX]; c.i := [DX];$   
 $n.t := [AX]; n.i := [DX];$

NODE 5: add  $(A[I] := (B+C*D) - E/F+G[J*K])$   


initially:  $n.t = n.p = [G(1)]; n.w = n.i = [ ]$   
 compute:  $c.t := [AX]; c.u := [AX, DX];$   
 $n.s := [AX, DX]; c.i := [DX];$   
 $n.t := [AX]; n.i := [DX];$

NODE 6: load  $(A[I] := (B+C*D) - (E/F+G[J*K]))$   


initially:  $n.t = [G(1)]; n.w = n.i = [ ]$   
 compute:  $n.s := [ ];$

NODE 7: div  
 node 6      F

initially:  
 compute:

$(A[I] := (B+C*D) - (E/F+G[J*K]))$



n.t=n.p=[AX]; n.w=[DX]; n.i=[ ]  
 c.t := [AX]; c.u := [AX];  
 n.s := [AX]; c.i := [DX];  
 n.t := [AX]; n.i := [DX];

NODE 8: load  
 J

initially:  
 compute:

$(A[I] := (B+C*D) - (E/F+G[J*K]))$



n.t=[G(1)]; n.w=n.i=[ ]  
 n.s := [ ];

NODE 9: mult  
 node 8      K

initially:  
 compute:

$(A[I] := (B+C*D) - (E/F+G[J*K]))$



n.p=n.t=[AX]; n.w=[DX]; n.i=[ ]  
 c.t := [AX]; c.u := [AX];  
 n.s := [AX]; c.i := [DX];  
 n.t := [AX]; n.i := [DX];

NODE 10: imul  
 node 9      #2

initially:  
 compute:

$(A[I] := (B+C*D) - E/F+G[J*K])$



n.p=n.t=[G(1)]; n.w=n.i=[ ]  
 c.t := [AX]; c.u := [AX,DX];  
 n.s := [AX,DX]; c.i := [DX];  
 n.t := [AX]; n.i := [DX];

NODE 11: load  
 node 10      @G

initially:  
 compute:

$(A[I] := (B+C*D) - (E/F+G[J*K]))$



n.p=[A(1)]; n.t=[G(1)]; n.w=n.i=[ ]  
 c.t := [ ];

\*\*\* restriction exception:

target and preference incompatible -- insert move \*\*\*

NODE 11' : move  
 node 10

initially:  
 compute:

n.p=n.t=[G(1)]; n.w=n.i=[ ]  
 c.t := [AX]; c.u := [AX,DX];  
 n.s := [AX,DX]; c.i := [DX];

NODE 11: load  
(cont'd) /  
node 11' @G

compute:  $c.t := [A(1)]; c.u := [A(1), AX, DX];$   
 $n.s := [A(1), AX, DX]; c.i := [ ];$

NODE 12: add  $(A[1] := (B+C*D) - (E/F+G[J*K]))$   
 node 7 node 11

initially:  $n.p_l = n.p_r = n.t = [G(1)]; n.w = n.i = [ ]$   
 compute:  $cl.t := [AX]; cl.u := [AX, DX];$   
 $cr.t := [G(1)]; cr.u := [A(1), AX, DX];$   
 $n.o := \text{right first};$   
 $n.s := [A(1), AX, DX];$   
 $cl.i := [AX, DX]; c2.i := [DX];$   
 $|cr.t| > |cl.t| \Rightarrow \text{swap}(l, r);$   
 $n.t := [G(1)]; n.i := [ ];$

NODE 13: sub  $(A[I] := (B+C*D) - (E/F+G[J*K]))$   
 node 5 node 12

initially:  $n.p_l = n.p_r = n.t = [G(1)]; n.w = n.i = [ ]$   
 compute:  $cl.t := [AX]; cl.u := [AX, DX];$   
 $cr.t := [G(1)]; cr.u := [A(1), AX, DX];$   
 $n.o := \text{right first};$   
 $n.s := [A(1), AX, DX];$   
 $cl.i := [AX, DX]; c2.i := [DX];$   
 $n.t := [AX]; n.i := [DX];$

NODE 14: store  $(A[I] := (B+C*D) - (E/F+G[J*K]))$   
 node 2 @A node 13

initially:  $n.p_l = [A(1)]; n.p_r = [G(1)]; n.t = n.w = n.i = [ ]$   
 compute:  $cl.t := [A(1)]; cl.u := [A(1)];$   
 $cr.t := [AX]; cr.u := [A(1), AX, DX];$   
 $n.o := \text{right first};$   
 $n.s := [A(1), AX, DX];$   
 $cl.i := [A(1), DX]; c2.i := [ ];$

Figure 4.6 shows the code tree at completion of pass 1 as just performed, with target and interference vectors specified, order of evaluation shown, and target paths marked. Note that order of evaluation is orthogonal to target path selection, that a move was inserted between the "J\*K" subscript computation and the loading of 'G[J\*K]', and that no spills were necessary.

In the second, top-down, pass we merely pass target and interference vectors down target paths. In Figure 4.6, only the leftmost target path inheritance results in a change, with the 'load (I)' node changing its target vector from '[G(1)]' to '[A(1)]'. Finally on the third, bottom-up, pass we assign specific registers based on the previous allocation and call the template expander. The sequence of templates based on Figure 4.6 and the given register assignment algorithm would be (again with target operands on the left):

	load	AX,J
	mult	AX,K
	imul	AX,#2
	move	BX,AX
(1)	load	BX,G[BX]
(2)	load	AX,E
(3)	div	AX,F
	add	BX,AX
	load	AX,C
	mult	AX,D
	add	AX,B
	sub	AX,BX
(4)	load	BX,I
	imul	BX,#2
	store	A[BX],AX

The entire assignment ( $A[I] := (B+C*D) - (E/F + G[J*K])$ ) is thus computed using only three registers (AX, DX, BX). Note that DX does not appear above, as it is a work register for the double length dividend or product in divides or multiplies. In particular, the template expander will need to propagate the sign of the dividend through DX prior to the division of lines (2) and (3). This may result in DX actually being loaded in line (2) and then getting shifted arithmetically into AX prior to the division in line (3); this is one of the nuisance details handled by the code template mechanism. Note also that in lines (1) and (4) we have arbitrarily chosen BX as the target register, when our target vector was actually more general. This is only to indicate that BX could be chosen if the other registers were needed, and that this assignment could be restrained to three registers. In general, however, the assignment will be "loose" in the sense that if there is not much "pressure" on the registers, an arbitrary selection will be made that may harmlessly employ more registers than the minimum possible. A simple heuristic to select the most recently used registers could be employed if global register minimization is desired, but this is not worth the effort, as a rule, on the iAPX 286. One heuristic that is valuable is to track current register contents inside the register assignment procedure and eliminate register loads wherever possible, or at least convert them into register moves.

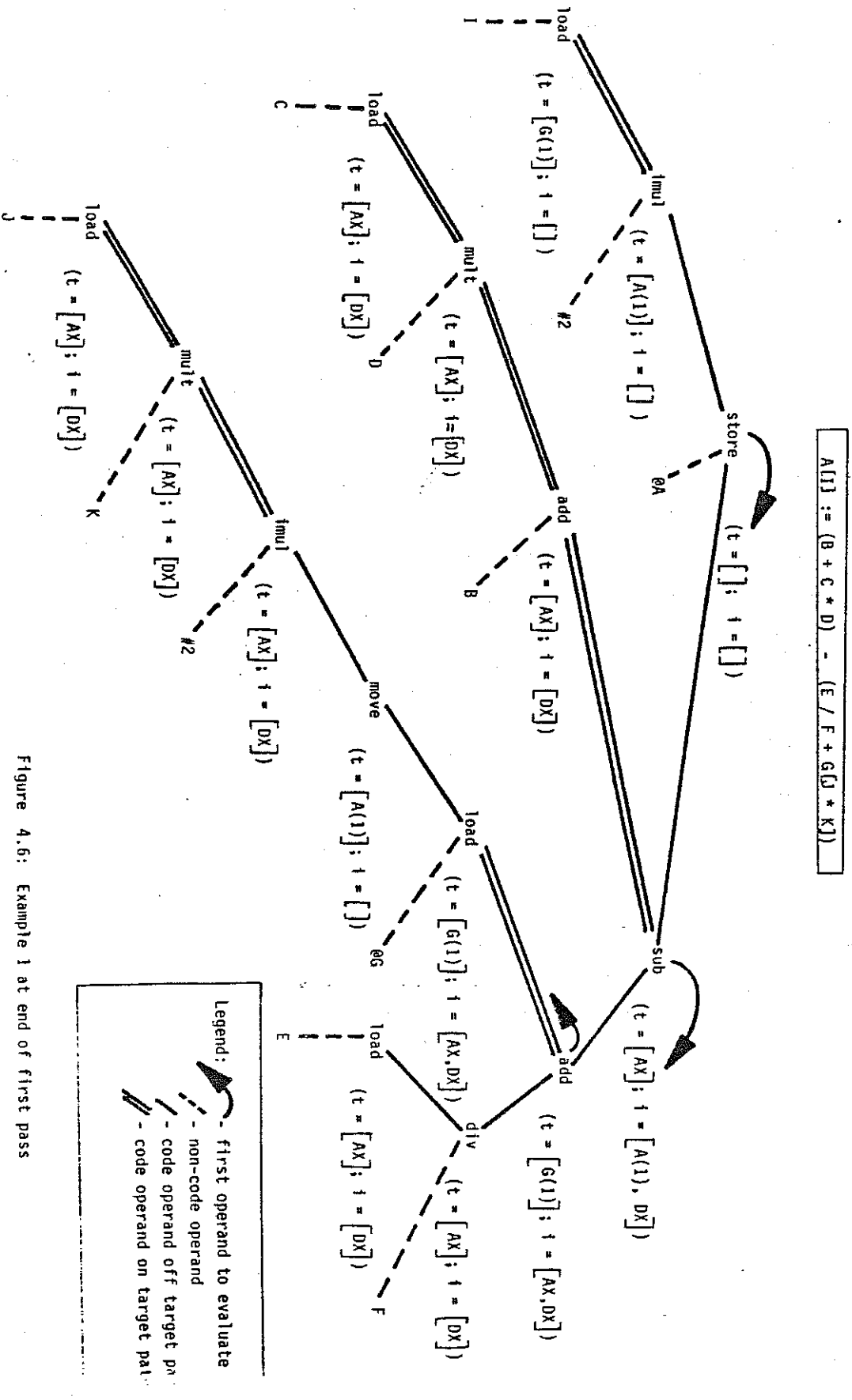
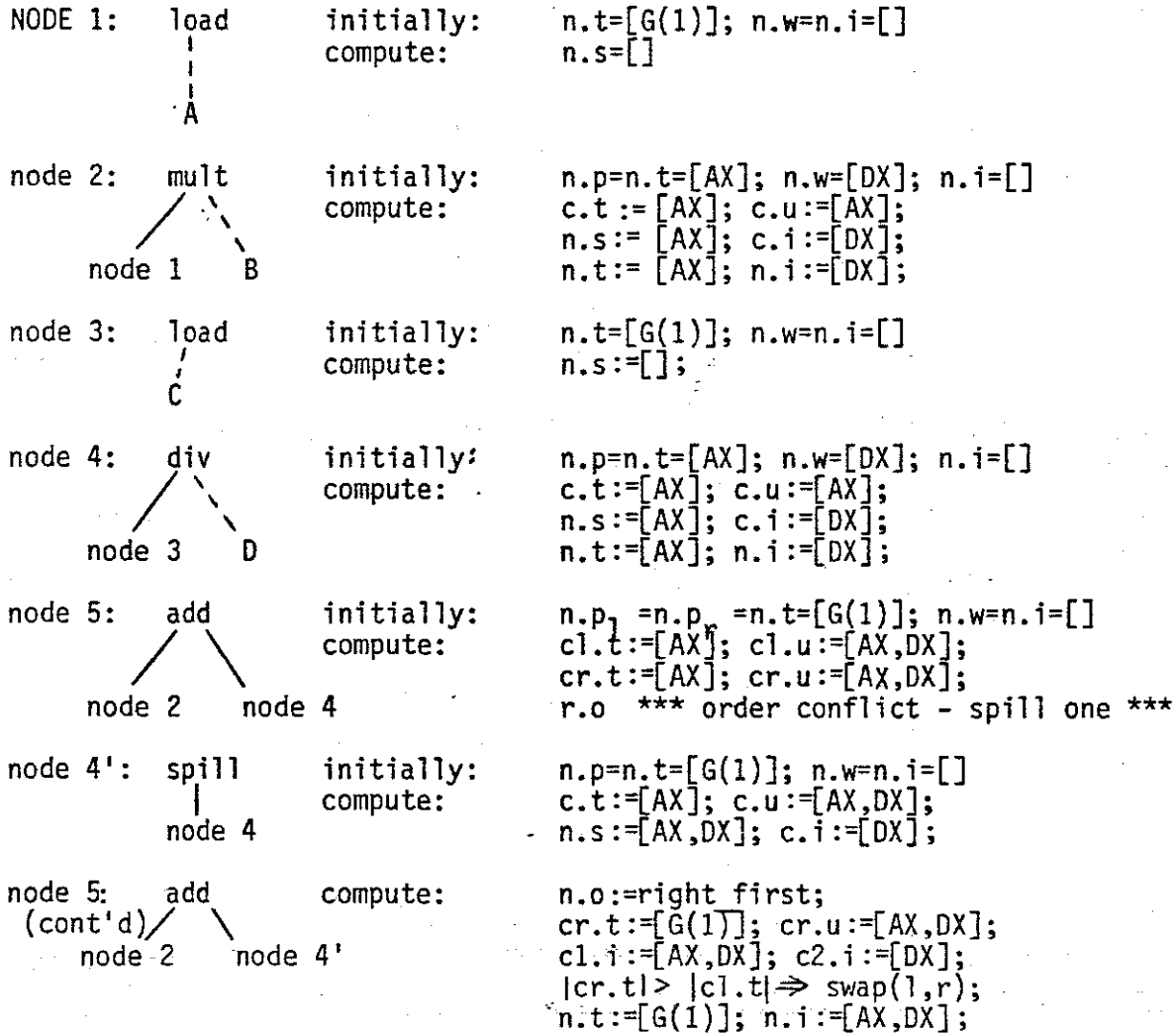


Figure 4.6: Example 1 at end of first pass

## \*\* A SECOND EXAMPLE \*\*

Since our first example did not generate any spills, we now give a simple example of the spill mechanism, using the expression:

$$A*B + C/D$$



Now the spill macro that is inserted as node 4' may be handled in two ways, as:

- i) a move from a register in the target vector to another register in the preference vector, if this would remove the order conflict (i.e. (n.p<sub>1</sub>?cr.u) or (n.p<sub>2</sub>?cl.u) holds); or as
- ii) a spill to a temporary storage location (e.g. the stack) and a change of the parent node to reference the operand in memory, if possible, or to reload the operand, if it must be in a register;

In our example the first alternative is preferable and leads to:

```

load    AX,C
mult    AX,D
move    CX,AX
load    AX,A
mult    AX,B
add     AX,CX

```

The second alternative would be less desirable, as

```

load    AX,C
mult    AX,D
push    AX
load    AX,A
mult    AX,B
add     AX,temp - or - pop DX
                        add AX,DX

```

#### \* A TECHNIQUE FOR LARGE DATA STRUCTURES \*

The segmentation facilities of the iAPX 286 are useful in structuring large address spaces into conceptually manageable units. The four segment registers (CS, DS, SS, ES) treat the current code, data, stack, and external areas as logically separate segments of memory whose relative position in a linear physical memory is immaterial. Whenever possible it is desirable to modularize data structures so that they can be conveniently referenced by this segmentation scheme. Occasionally, though, a natural decomposition of a large (> 64 KBytes) data structure does not exist, and it must be treated in its entirety. Examples might be large tables or matrices such as used in some control or numeric applications. This section discusses a compilation approach to large data structures that cannot entirely reside in a single segment.

Figure 4.7 shows the situation for a multi-segment data structure, with a contiguous portion of a segment table (GDT or LDT) containing the segment descriptors for the segments composing the structure. For arrays, the segments should always contain an integral number of array elements (i.e. not split an element across segment boundaries). For efficiency in manipulating arrays, the number of elements in a segment could be a power of two, so that shifts instead of divides could be used; this is not mandatory. For example, an 80,000 element array of two-byte integers would appear as two segments of  $2^{15} = 32,768$  elements, followed by a segment of 14,464 elements. As another example, an 80,000 element array of three-byte entries would appear as four segments of  $2^{14} = 16,384$  elements, followed by a segment of 14,464 elements; the first four segments are each 49,152 bytes in length. The largest power of two is used that will still yield an array slice that fits into 64 Kbytes.

Segment  
table:

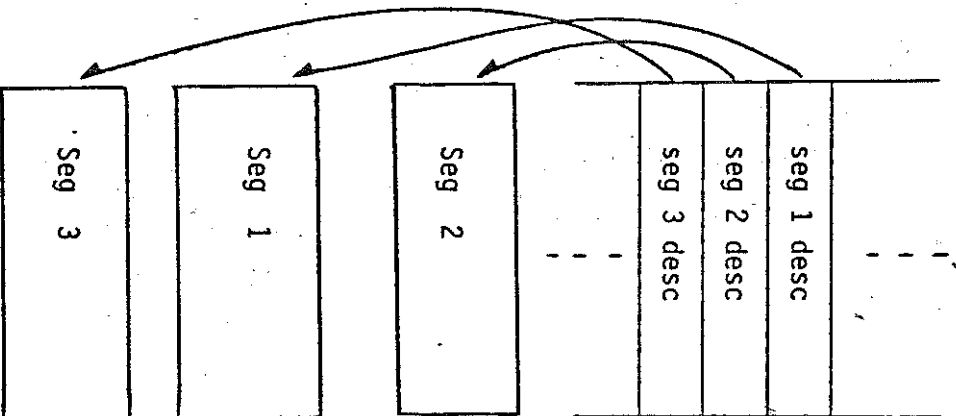


Figure 4.7: Multi-segment data structure

Figure 4.8 shows the general case address computation for a multi-segment data structure. A linear 32-bit element index is viewed at the top of the figure as a relative segment number, *sn*, followed by a relative element number, *en*, within the segment. The significant bits (up to 13) of *sn* are shifted left by 3 bit positions (to clear the L/G and RPL fields of the selector), added to the base selector value, and taken as the selector portion of the array element virtual address. The *en* value is multiplied by the element size and used as the offset portion of the array element virtual address. Assuming an index computation has left the element index in *DX:AX*, we have:

```
-- DX:AX contains element index

mov     BX,AX
and     BX,en_mask    } BX := en * el_size
imul    BX,el_size
shr     AX,i          }
shl     DX,16-i       } DX := sn
add     DX,AX
shl     DX,3          }
add     DX,bs         } ES := sn*23 + bs
mov     ES,DX

-- array element virtual address in ES:BX
```

where *i* is the number of bits within *en*. In the special case where *i*=16, such as for a byte string or byte array, the above reduces to:

```
mov     BX,AX
shl     DX,3
add     DX,bs
mov     ES,DX
```

---

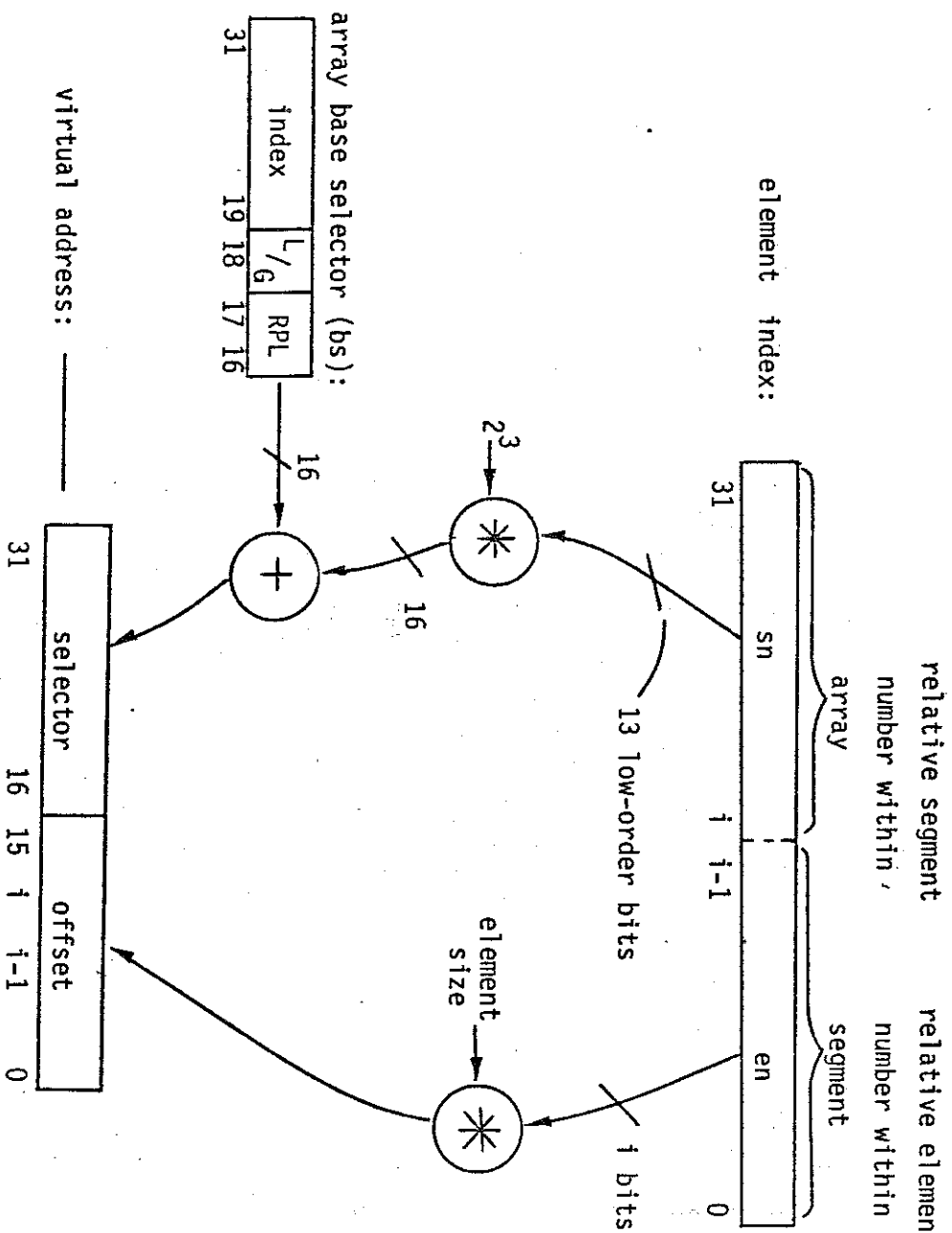


Figure 4.8: Multi-segment address computation (general case)

## \*\* 5. EXCEPTIONS, ERRORS, DEBUGGING, AND INTERPRETATION \*\*

Handling of exceptional conditions and errors, and debugging of programs are important topics in programming language implementation. In the iAPX 286 all exceptional conditions are intercepted by vectoring through the interrupt descriptor table (IDT) to a procedure or task that handles the exception. The IDT entries are either interrupt trap gates (for procedure handlers) or task gates (for task handlers). A procedure handler executes as a simple interrupt procedure within the current task and is often a part of the operating system portion of the virtual address space. A task handler is an entirely separate task for servicing of interrupts that can use its own local address space. Most exceptions are maskable, in that they will only be honored by the processor if the interrupt enable bit (IF) in the flags register is set. Interrupts are automatically disabled by transfer to a procedure handler, but transfer to a task handler loads the flags register from the new task state segment, setting IF as determined in the new TSS.

Three classes of events may cause an exception handler to be invoked:

- . external interrupts that assert a signal on the processor's interrupt pins,
- . traps caused by the software interrupt instructions, and
- . faults detected before or during instruction execution.

External interrupts normally proceed from an 8259A interrupt controller that resolves priorities among interrupt sources and provides an interrupt address. Traps are caused explicitly by the INT(#) instruction that uses the number (#) as an index into the interrupt descriptor table. A one-byte form of the instruction implicitly uses IDT index 3 and can be used by debuggers for setting breakpoints. Faults detected before or during instruction execution cause automatic transfer of control through the IDT and in general are not masked (i.e. do not depend on the setting of IF). Figure 5.1 shows the pre-assigned and reserved IDT entries. An asterisk indicates a non-maskable fault. Note that entry 1, the single-step trap, is listed as non-maskable because it is controlled by the trap flag, TF, rather than the interrupt enable flag, IF. Also note that entry 2, the non-maskable interrupt, is triggered by a signal on the processor's NMI pin. Non-maskable faults have a higher priority than external interrupts and are handled first.

The interrupt mechanism for protection violations will store an interrupt error word on the stack of the interrupt error handler - this is described in detail in the appropriate 286 manual (2). Note that a debugger would typically want to field protection violations during a debug session and the underlying operating system needs to support this requirement.

Processor extensions, or coprocessors, provide their own error codes through status inquiry instructions of the coprocessor. The 286 merely provides a generic trap (#16) for these conditions.

<u>IDT no.</u>	<u>Exception</u>
0*	divide error
1*	single-step
2*	non-maskable interrupt
3	breakpoint trap
4*	INT0-detected integer overflow
5*	BOUND-detected range error
6*	invalid opcode
7*	processor extension (coprocessor) not present
8*	double protection fault
9*	Processor extension (coprocessor) segment overrun
10*	invalid task segment format
11*	segment not present
12*	stack under/over flow
13*	general protection violation
14-15	reserved
16	processor extension (coprocessor) detected fault
17-31	reserved

\*- non-maskable

Figure 5.1: IDT pre-assigned and reserved entries.

---

\* DEBUGGERS \*

Debuggers are generally of two varieties - system debuggers and application debuggers. System debuggers are either low-level monitor-style debuggers at the machine level, or intelligent debuggers that understand and recognize operating system data structures and descriptor tables. System debuggers are in some ways simpler than application debuggers, since they can assume they control the entire system during a debug session, and in other ways more complex, since they must deal with operating system data structures, descriptor tables, and multi-tasking.

Application debuggers also come in two varieties - assembly level, or high-level language level. Either way an application debugger must assume the existence of an underlying operating system and of other applications programs, none of which may have their integrity violated. Indeed multiple instances of an application debugger may be in use simultaneously. Whether an application debugger needs to support debugging of multitasking applications is determined by the situation, but typically they do not.

Debuggers in the iAPX 286 may insert breakpoints using the one-byte form of the INT instruction. Single-stepping is also possible by setting the TF flag, but it is recommended only for machine-level debugging, as typically a high-level language programmer views the statement or even the procedure as a single "step". With a table of statement and procedure entry offsets, breakpoints may be inserted to give the high-level language equivalent of single stepping. Typically such a table is established as a "breakpoint table" with entries such as:

ENTRY:

code seg offset	breakpoint byte
-----------------	-----------------

Initially a breakpoint table is established for each code segment where breakpoints will be set, and the initial value of the breakpoint byte is set to the one-byte INT opcode. The following instructions would set/clear all the breakpoints in a breakpoint table (such as might be done for single statement stepping if the table is of statement offsets):

```

-- let bpt=breakpoint table description: (table    table    code)
--                                       (address, length, segment)
-- offset=code segment offset field of entry
-- bpop=breakpoint opcode (immediate value)
-- bb=breakpoint byte field of entry

bb_swap:  mov     SI,bpt
          mov     CX,bpt+2      -- number of entries
          mov     ES,bpt+4      -- segment base
1:        mov     DI,[SI]       -- DI ← offset
          mov     AX,2[SI]      -- AX ← bb
          xchg    AX,ES:[DI]    -- AX ↔ user code byte
          mov     2[SI],AX      -- bb ← user code byte
          add     SI,2          -- advance to next entry
          loopnz  1             -- and loop

```

Note that the user code segment wherein the breakpoints are set is treated as an external data segment in the breakpoint routine, and hence must use a data segment descriptor on a protected mode 286. The user program of course references its code segments through the CS register as executable segments. This means that the debugger needs an alias descriptor for each code segment in the user program that describes the code segment as a data segment. (Note that aliasing of segments requires a certain amount of bookkeeping work in the OS, especially if demand swapping of segments is supported.) The underlying operating system needs to support these data aliases for use by the debugger, but not for use by the user program. This means the debugger either operates at a higher privilege level or in a different task entirely from the user program. The debugger itself, of course is also protected from the program under debug by this same mechanism. In general it may be simpler to develop the debugger as a separate task since then it need not be bound with the user program, it has an entirely separate address and descriptor space, and it may operate at the appropriate privilege level(s) as established by the system. (Note that a debugger within the same task as the faulting or breakpointing program must either be "conforming" or be at the same or higher level of privilege since interrupt-induced level transitions must go to a more privileged (inward) level).

The debugger task needs an entry point for handling breakpoint exceptions. The 'breakpoint occurrence' entry point receives control when the user program executes the breakpoint instruction, INT (implicit #3). It should do the following:

```
breakpoint occurrence:  retrieve user CS and IP values from stack;
                        set 'current breakpoint' to user CS, IP values;
                        lookup user CS value in 'code segs' table and find
                          corresponding 'bpt';
                        set alias ES value from bpt description, lookup
                          user IP value in bpt table and get
                          corresponding bb value, then swap bb value with
                          the breakpoint byte;
                        go to user prompt routine.
```

The two table lookups above (for CS and IP) may use techniques such as hashing or binary search to speed search time. Once the user is prompted he may do any debugger operation. If or when the user elects to resume execution from the breakpoint, control is transferred back to the user program via the 'current breakpoint' address.

#### \* INTERPRETATION TECHNIQUES \*

Very often interpretation techniques are used to speed the edit-compile-debug cycle, to simplify breakpointing, tracing, single stepping, patching, profiling, etc., and to improve debugger portability. Klint (20) gives an excellent tutorial on interpretation techniques, and the following is adapted to the 286 from his paper.

Four techniques will be given

- i) classical interpreter with opcode table;
- ii) direct threaded code;
- iii) indirect threaded code; and
- iv) partial interpretation via software interrupt.

Before describing the techniques, let's first describe the segmentation view of our interpretation schemes. Figure 5.2 shows the various existing segments under an interpreter model. We assume the interpreter is small enough that all its code and read-only data can reside in a single segment (referenced via CS). The interpreter stack uses SS, and scratch data uses the registers or the stack. We assume most read-write data is stored in the interpreted program's data segments, but a small amount of read-write data private to the interpreter could be stored at the base of the stack and be referenced via SS. The user code is the most frequently referenced data for the interpreter, and is referenced via DS, while all types of user data are referenced via ES. This segmentation model is necessary to efficiently utilize the machines four active segment registers. We will use BX as the interpreted program's instruction pointer and maintain it always at the next byte to be read.

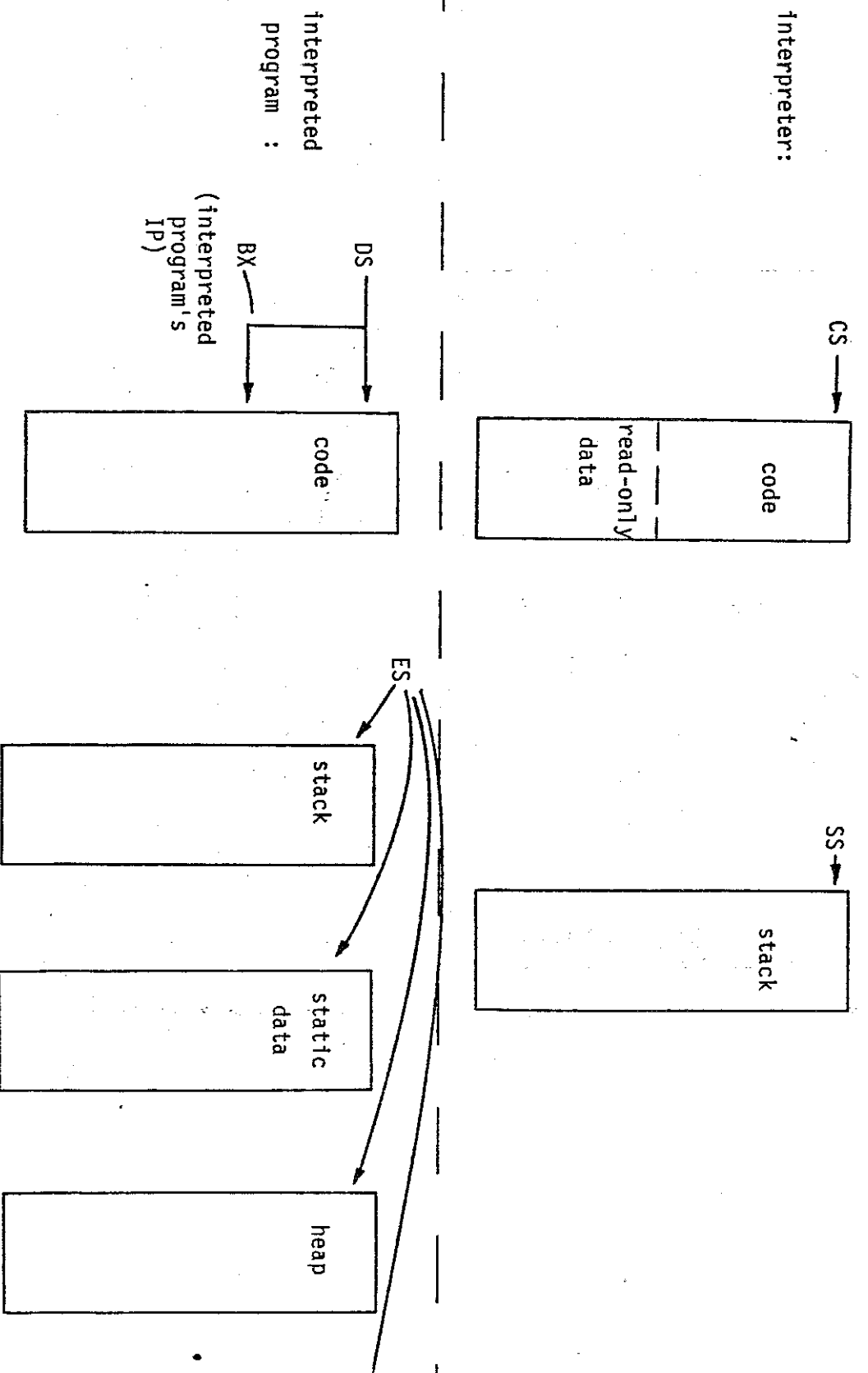


Figure 5.2: Segmentation view of an interpreted system

## \* CLASSICAL INTERPRETER \*

Classic interpretation uses an opcode table (indexed by the opcodes in the interpreted program) to vector to the appropriate interpretation routine:

1. pick up user opcode (8-bits, typically) from location DS:[BX];
2. increment user PC (BX) appropriately;
3. execute routine at location opcode\_table[user opcode].

The interpreter loop is:

			<u>BYTES</u>	<u>CYCLES</u>
interpret:	xor	SI,SI	2	2
	movbyte	SI,[BX]	2	5
	inc	BX	1	2
	jmp	opcode_table[SI]	3-4	13-15
.....	....			
(at end of routine)	jmp	interpret	<u>3n</u> (n routines)	<u>9</u>
			8-9+3n bytes	31-33 cycles

A typical instruction would load arguments into registers, execute the instruction, and repeat the interpretation loop:

```

instruction:      mov      reg, [BX]
                  inc      BX  -- or add BX,#2
                  .
                  .          (pick up other operands)
                  .
                  .
                  .          (execute instruction)
                  .
                  jmp      interpret

```

Classical interpretation has the advantage that modifications to the interpreter need not affect the interpreted code. In particular, the addresses of interpreter routines may be altered (without affecting user programs) simply by changing the opcode table. This can be useful in a debugger, for example, for enabling or disabling procedure tracing by altering the procedure entry routine address in the opcode table.

\*\* DIRECT THREADED CODE \*\*

The basic scheme for direct threaded code is:

1. Pick up the user opcode (16-bits, now) from location DS:[BX];
2. Increment BX appropriately;
3. Execute the routine at location user opcode.

The interpreter loop is:

		<u>bytes</u>	<u>cycles</u>
interpret:	mov SI,[BX]	2	5
	add BX, #2	3	3
	jmp SI	3	10-12
.....	.....		
(at end routine)	jmp interpret	<u>3n</u>	<u>9</u>
		8+3n bytes	27-29 cycles

If the BX incrementing is moved to the head of the interpretation routine, this can be changed to:

interpret:	jmp [BX]	2	13-15
(at start routine)	add BX,#2	3n	3
(at end routine)	jmp interpret	<u>3n</u>	<u>9</u>
		2+6n bytes	25-27 cycles

Opcodes are interpretation routine displacements here and are now 16-bits, taking twice the opcode space in the interpreted program. Changes to the interpreter now require recompilation of interpreted programs, to change the opcodes. Direct threaded code is slightly faster than classical interpretation. Tracing or break-pointing now requires testing of flags, since routines may not be changed without recompilation of user programs.

## \* INDIRECT THREADED CODE \*

The basic scheme for indirect threaded code is:

1. Pick up the user opcode (typically 16-bits) from location DS:[BX];
2. Increment BX appropriately;
3. Execute routine whose address is stored at the location addressed by user opcode.

The interpreter loop is:

		<u>bytes</u>	<u>cycles</u>
interpret:	mov SI,[BX]	2	5
	add BX,#2	3	3
	jmp [SI]	2	13-15
(at end routine)	jmp interpret	3n	9
		<u>7+3n bytes</u>	<u>30-32 cycles</u>

Indirect threaded code here shares most of the properties of direct threaded code, with the exception that changes in the interpreter do not require changes in the user opcodes, since they are only indirect (rather than direct) addresses.

## \* PARTIAL INTERPRETATIONS \*

When most user code is actually 286 machine code and only some instructions need to be interpreted, the software interrupt can be used for these interpreted instructions. The interpreted instruction's user opcode is actually a 286 INT (#n) instruction trap that vectors to a trap handler to actually process the interpreted instruction. Use of a software interrupt has the advantage that the instruction interpretation routines need not be bound to the user program. Alternatively a CALL instruction to an out-of-line routine may be used for the user opcode. This is done for complex operations in most high-level languages, since the language's runtime library is always bound to the user program prior to execution. Interpretation routines implemented as trap handlers are normally "conforming", i.e. they execute at the user's privilege level.

## 6.0 THE iAPX 286 OBJECT LANGUAGE

Programming language translators accept a source language file as input and generate an object language file as output. All iAPX 286 compilers and assemblers provided by Intel (and many provided by independent software suppliers) use a single, well-defined, and powerful object language that permits multi-lingual programming. The object language files (hereafter called simply object files) produced by translators are subsequently manipulated by various object utilities such as binders, system builders, program librarians, loaders, and debuggers (described in Section 7). This section describes the iAPX 286 object language in its generality. Since it does support most conceivable source languages, it may appear overly complex for any single language. However, a programming language implementor ought to choose the appropriate subset of capabilities from this object language rather than define a new incompatible language. An abstract procedural interface to the language is provided, so that the details of encoding may be ignored in actual usage. The description follows, and is abstracted from the pertinent technical specifications (21, 22). Numbers above the boxes indicate field length in bytes (or \* for variable length and 'b' for bit length).

### \* OBJECT FILES \*

Four types of object files exist in the language:

- linkable files - produced by translators or incremental linkers for subsequent binding into loadable files;
- loadable files - produced by binders or system builders for subsequent loading into processor memory;
- bootable files - produced by system builders for subsequent loading into processor memory as its initial contents;
- library files - produced by libraries for subsequent usage during the binding process.

An object file appears structurally as:

```

      1          *
! file header ! contents !

```

where 'header' gives the one-byte file type and 'contents' is peculiar to a given type. For a linkable file the contents are a sequence of linkable modules:

```

      *
! linkable module ! . . .

```

For a loadable file the contents are a single loadable module:

```

      *
! loadable module !

```

and similarly for a bootloadable file's contents:

\*  
! bootloadable module !

while a library file's contents is an object library:

\*  
! object library !

Primitive operations for object files are:

put\_obj\_file (type): opens for output and initiates a new object file of the desired type;

end\_put\_file: terminates and closes an object file opened for output;

get\_obj\_file: opens an existing object file and returns the file type;

end\_get\_file: closes an input object file.

We now continue our description of the object language, traversing down through the various levels.

#### \* LINKABLE MODULES \*

A linkable module corresponds to the unit of separate compilation in the particular programming language. Languages or compilers permitting batching of separate compilations (e.g. Fortran) will generate a sequence of linkable modules, otherwise a single linkable module is usually the entire contents of a linkable file.

A linkable module consists of a 'linkable module header', followed by a sequence of 'sections' and a final checksum:

175                      \*                      \*                      1  
! linkable module header ! section ! ... ! section ! checksum !

A linkable module header is:

4                      2                      2                      2  
! total length ! # of segments ! # of gates ! # of publics !

2                      1                      8                      8                      41                      41  
! # of externals ! linked ! date ! time ! module name ! module creator !

56                      8  
! table of contents ! reserved !

where most fields are self-explanatory. The 'linked' field is a boolean value; false for linkable modules produced by translators (indicating presence of type information, public symbol directory, and external symbol directory sections in the module), and true for linkable modules produced by incremental linkers (indicating type information and public and external symbol directories have been processed into debug segments). The table of contents is:

4	4	4	4
! seg def location !	length !	gate def location !	length !
4	4	4	4
! type def location !	length !	pub def location !	length !
4	4	4	4
! ext def location !	length !	text fix location !	length !
4	4		
! reg int location !	length !		

where the various fields will become clear after the linkable module sections are explained. The location fields are byte offsets into the modules.

Operations on linkable modules are:

`put_link_module (linked, date, time, mod_name, creator_name):`  
 initiates a linkable module on the output file; this then becomes the current module of the file;

`end_put_link_module:` terminates the current module of the file and updates its module header (including table of contents);

`get_link_module:` finds next module on the input file and returns the values of linked, date, time, module name, creator name, and found (true if a next module exists, false otherwise).

#### \* LOADABLE MODULES \*

Loadable modules are quite similar to linkable modules:

106	*	*	1
! loadable module header !	section !	... ! section !	checksum !

with the key exception that the sections must be in a fixed order to facilitate simple and rapid loading (namely: descriptors, loadable text, loadable fixups, descriptor names, debug text, debug fixups, and O.S. information, in that order). A loadable module header has the following format:

4	2	1	8	8	41
! total space ! # of descriptors ! built ! date ! time ! module creator !					
2	32	8			
! TSS selector ! table of contents ! reserved !					

where table of contents is:

4	4	4			
! descriptors location ! loadable text location ! loadable fixups location !					
4	4	4			
! descriptor names location ! debug text location ! debug fixups location !					
4	4				
! O.S. info location ! last location !					

'Built' is a boolean value, false for single-LDT modules and true for multi-LDT modules (created by the system builder), in which case the header's TSS selector field is a selector into the GDT for the initial task state segment.

Operations on loadable modules are:

`put_load_module` (total length, # of descriptors, built, date, time, creator name, TSS selector): initiates a loadable object module on the output file; if built is false then the eventual loader must construct the single LDT; if built is true then the module explicitly defines the multiple LDTs;

`get_load_module`: obtains the module from the object file and returns the values of the parameters described above, as well as a boolean 'found' value.

#### \* BOOTLOADABLE MODULES \*

Bootloadable modules are constructed by the system builder as the initial memory image for a system. It has only two sections:

95	*	*	1
! bootloadable module header ! section ! section ! checksum !			

The first section is absolute text and the second section is debug text. The bootloadable module header is:

4	8	8	41	2	4
! total space ! date ! time ! module creator ! GDT limit ! GDT base !					
2	4	2	4		
! IDT limit ! IDT base ! TSS selector ! abs text location !					
4	4	8			
! debug text location ! last location ! reserved !					

The only required section is absolute text. The TSS selector is a selector into the GDT for the initial task state segment, which should have requested privilege level (RPL) of zero.

Operations on bootloadable modules include:

`put_boot_module` (total\_length, date, time, creator\_name, GDT\_limit, GDT\_base, IDT\_limit, IDT\_base, TSS\_selector):  
initiates a bootloadable object module on the output file;

`get_boot_module`: returns true if the bootloadable module is found, and obtains the information produced by `put_boot_module`.

### \* OBJECT LIBRARIES \*

Linkable object modules may be placed in an object library by a utility called the librarian. These object modules may then be selectively copied from the library during the binding process whenever the binder is able to satisfy an external reference from one of the contained modules. An object library has the following format:

101	*	*	3
! library header ! library section ! ... ! library section ! checksum !			

The library header is:

4	2	2	2
! total length ! # of modules ! # of publics ! # of deleted modules !			
2	8	8	8
! # of library sections ! date created ! time created ! date last modified !			
8	41	2	
! time last modified ! library name ! version # !			
4		4	
! first library section location ! last library section location !			
6			
! reserved !			

Each library section appears in the format:

18	*	*
! table of contents ! module name directory ! public symbol directory !		
*		
! linkable module ! ... !		

where the table of contents appears as:

4	2
<u>! next library section location ! length of directories !</u>	
4	2
<u>! module name directory location ! length !</u>	
4	2
<u>! public symbol directory location ! length !</u>	

Each location field is a four-byte integer giving the byte offset from the start of the library file. 'Next library section location' is zero if this is the last library section. 'Length of directories' refers to this section's directories only.

Operations on object libraries are:

`begin_put_lib_section` (# of publics, length of publics, # of module names, length of module names):

starts a new library section in the output file;

`put_lib_dir` (module name or publics switch):

initiates a directory in the library file (to be followed by individual calls of `put_lib_sym` or `put_mod_name`);

`put_mod_name` (name, location):

writes one entry in the module name directory of the current library section;

`put_pub_sym` (name, module index, location):

writes one entry in the public symbol directory of the current library section; note that 'location' refers to the containing module (not to the public symbol);

`put_module_header` (header, module length):

puts the header into the output file;

`put_module_contents` (contents, length):

used after 'put\_module\_header' to copy the contents of a module to the output file;

`end_put_lib_section`:

terminates a library section;

`begin_update_lib_dir`:

initiates updating of all the directories in the input library file; followed by calls to 'update\_next\_section';

`update_next_section`:

seeks to the next section in the library so that it can be updated using 'update\_lib\_dir', 'update\_mod\_name', and 'update\_pub\_sym' procedures; returns false if no more sections in the library;

`update_lib_dir (mod_name_dir_or_pub_sym_dir_switch):`  
 initiates the update of the designated directory in the current library section; to be followed by calls to 'update\_mod\_name' or 'update\_pub\_sym';

`update_mod_name (name, location, deleted):`  
 overwrites the current module name directory entry with the given entry (`deleted = false`) or simply deletes the current entry (`deleted = true`); called once for each entry in the directory;

`update_pub_sym (name, module_index, module_location, deleted):`  
 overwrites the current public symbol directory entry (`deleted = false`) with the given entry or simply deletes the current entry (`deleted = true`); called once for each entry in the directory;

`update_lib_header (creation_date, creation_time, modified_date, modified_time, library name, version #):`  
 updates the header of the output library file;

`end_update_lib_dir:`  
 terminates library updating; between call to 'begin\_update\_lib\_dir' and 'end\_update\_lib\_dir', all entries of all directories of the library file must be updated.

## \* SECTIONS \*

Library sections have just been described; the other sections are absolute text, debug loadable fixups, debug loadable text, descriptors, descriptor names, external definition, gate definition, loadable fixup, loadable text, public definition, register initialization, segment definition, text and fixup, and type definition sections. Within sections it is convenient to refer to segments, gates, external symbols, GDT-selectors, descriptors, and types by internal (one-word) names. Internal names are local to the module which contains them, and are encoded in 16-bits as follows:

<u>Internal Name Type</u>	<u>Encoded Representation</u>
segment	<u>3b 13b</u> <u>! 1 ! index !</u>
gate	<u>3b 13b</u> <u>! 2 ! index !</u>
external symbol	<u>3b 13b</u> <u>! 3 ! index !</u>
GDT selector	<u>3b 13b</u> <u>! 0 ! index !</u>

descriptor	<u>16b</u> <u>! index !</u>
type	<u>16b</u> <u>! index !</u>

Descriptors and type internal names are distinguished by context, while the others are distinguished by their 3-bit type field. Several special descriptor internal names are defined for the GDT itself, the IDT, the initial TSS selector, and a constant in a public definition or a fixup record.

The following section descriptions will also refer to so-called generalized addresses which are 32-bit addresses represented as:

<u>16b</u>	<u>16b</u>
<u>! offset !</u>	<u>! internal name !</u>

If the internal name is the special constant designator, then the offset field contains a 16-bit constant value. In addition, three special internal names are defined for text sections of loadable modules: -1 for the global descriptor table (GDT), -2 for the interrupt descriptor table (IDT), and -3 for the initial task state segment (TSS).

Operations on sections in general include:

get\_section\_leng (type):

finds the corresponding section type entry in the module table of contents and returns its length (32-bits); for linkable modules only;

next\_def\_type: returns the type of the physically next definition (section) in the current module of the input file (0=no next section);

skip\_section: skips the remaining part of the current section in the input linkable module;

section\_exists (type):

checks if a given section exists in the loadable module, where type is one of descriptor, loadable text, loadable fixup, descriptor name, debug text, or debug fixup;

decode\_int\_name (internal\_name):

returns the internal name type.

#### \* SEGMENT DEFINITION SECTION \*

The segment definition section appears in all linkable modules which contain segments. The section appears as:

<u>*</u>
<u>! segment definition ! ...</u>

The index field of the segment's internal name refers to its order in the above sequence of segment definitions (1 = first, 2 = second, etc.). A segment definition is constructed as:

2	2	2	2	*
! attributes !	slimit !	dlength !	LDT position !	combine name !

Attributes can be further described as:

3b	2b	4b	1b	1b	2b	1b
! access !	DPL !	reserved !	external !	conforming !	fixed !	expandable !

3b
! combine type !

where

- access: one of read-only, read-write, execute-only, execute-read;
- DPL: the segment's (descriptor) privilege level, 0-3; normally set to 3 by translators and potentially alterable by the system builder;
- external: true if the segment contains only externally defined symbols or if its size is unknown at translation time; in this case all fields except 'combine name' are ignored;
- conforming: true for conforming segments (executable segments that "conform" to their caller's privilege level); should be set false by translators and only be altered by the appropriate utility;
- fixed: true if this segment's descriptor must occupy a fixed location in the LDT;
- expandable: true if the segment may be expanded at load time (e.g. stack, heap);
- combine type--one of:

- |                   |  |
|-------------------|--|
| <u>no combine</u> | - this segment may not be merged with any other segments;  |
| <u>normal</u>     | - this segment is merged during binding with other segments of the same name and combine type, or with the same name and combine types of stack or dsc; when two normal segments are combined, the new segment length (slength) is the sum of the old lengths; offsets are appropriately fixed up; |
| <u>stack</u>      | - this segment is merged during binding with other segments of the same name and a combine type of normal, stack, or dsc; when two stack segments are combined, the new segment length is the sum of the old lengths, but no relocation of stack-relative offsets is done;                         |

- dsc - when a stack segment and a normal or dsc segment are combined, a dsc (data and stack combined) segment is created consisting of a normal portion (in high memory) and a stack portion (in low memory); the dlength field indicates the length of the normal (data) portion of a dsc segment; the normal portions of dsc segments combine just as normal segments; the stack portions of dsc segments combine just as stack segments; translators do not create dsc segments, only object utilities such as the binder do;
- common - a segment produced for Fortran named common; two common segments with the same name are combined--their lengths must be equal (Fortran rule); the new length is the same as the old and no relocation of offsets is performed;
- blank common - a segment produced for Fortran blank common; blank common segments are combined and the new length is the maximum of the old lengths; and no relocation of offsets is performed;
- debug - treated as a normal segment up to the point of final binding; content of debug segments is agreed to between translator and debugger designers.

Combine types are used to develop a particular translator's "model of segmentation" of programs and data. By controlling assignment of segment names and combine types, a translator can merge independent logical segments into the same physical segment, if desired for greater addressing efficiency. For example, a compilation switch could instruct the compiler to treat all the code from separately compiled modules as one physical code segment of less than 64K bytes. The compiler would then give the code segment in each compilation the same name and the normal combine type and the binder would do the rest. The compiler, being aware of this plan, would also use 16-bit inter-module calls and jumps, rather than the larger and slower 32-bit forms used for inter-segment transfers. Given this relation between combine types and 286 instruction forms, models of segmentation must be supported both within the translators and within the object utilities; they cannot be implemented in just one and not the other. Popular models of segmentation and their compiler control names include:

<u>Model of Segmentation</u>	<u>Design</u>
"small"	one normal code segment for all code; one dsc segment for all data;
"compact"	one normal code segment for all code; one normal data segment for all non-stack data; one stack segment for all stack data;
"medium"	one normal code segment per module; one dsc segment for all data;
"large"	one normal code segment per module; one normal data segment per module; one stack segment for all stack data.

Since there are conceivably an unlimited number of segmentation models, a more general (but cumbersome) scheme is to permit the user to decide an arbitrary combination strategy for his or her program by explicitly naming the segments to be combined (e.g. MODULE A.CODE + MODULE B.CODE, MODULE B.DATA + MODULE C.DATA). A compromise solution adopted by Intel for some of its translators is to allow specification of 'small' or 'compact' "subsystems" (e.g. SMALL (MODULE E, MODULE F)). This permits small programs to take advantage of the efficiencies of combined segments (fewer segment register loads, for instance) while still allowing them to make inter-segment references to operating system service routines outside the subsystem.

After the 'attributes' field, the remaining fields in a segment definition are:

- slimit            - the length of the segment minus one, in bytes; zero length non-external segments are prohibited;
- dlength          - for dsc segments, the number of data (i.e. non-stack) bytes in the segment; field ignored otherwise;
- LDT position    - (fixed = 1) → position this segment's descriptor must occupy in the LDT;  
                  - (fixed = 0) → ignore this field;
- combine name    - a character string containing the name for the segment; segments with the same name are candidates for combination, as described above, and hence the term "combine name".

Segment combination according to the already described rules generally finds all other attributes equal for the segments being combined. However, if this is not the case and some differences exist, the object utilities will use the following rules in combining two segments, referred to as 1 and 2, into a new segment, "combined":

- a) combination not permitted if 1.DPL = 2.DPL;
- b) combined.fixed ← 1.fixed or 2.fixed;
- c) combination not permitted if: 1.fixed and 2.fixed and 1.LDT position = 2.LDT position;
- d) combined.expandable ← 1.expandable or 2.expandable;
- e) combined.conforming ← 1.conforming or 2.conforming;
- f) combined.access set according to following table:

		2			
		RO	RW	EO	ER
1	RO	RO	RW	ER	ER
	RW	RW	RW	*	*
	EO	ER	*	EO	ER
	ER	ER	*	ER	ER

\* = error

- g) it is an error to try to combine two segments if the combined slength will exceed 2<sup>16</sup>, except for debug segments (where a warning is issued and the debug segments are left uncombined).

```
emit_segment (combine_name, combine_type, access, expandable, external,
slength):      creates a segment definition and returns its internal name;
               this procedure is generally used by translators to define a
               segment in the current module of an object file (the more
               elaborate 'put_link_segment' is used by object utilities);
               the internal name is used for subsequent references to the
               segment within the current module, either in subsequent
               procedure calls or in generalized addresses;
```

```
put_link_segment (combine_name, combine_type, access, expandable,
conforming, fixed, DPL, external, slength, dlength, LDT_position):
    identical to 'emit_segment', with some additions for usage
    by object utilities; 'conforming' may be set true, fixed
    and LDT_position may be set, and a dsc combine type with a
    given dlength may be specified;
```

! public definition ! ...

4	2	1	*
! generalized addresss	! type internal name	! word count	! symbolic name !

generalized address - address of the public symbol, as a generalized address; typically the internal name portion of the address names a segment, gate, GDT selector, or the special 'constant' internal name (e.g. for I/O port numbers);

type internal name - the internal name of the type of the public symbol;  
supports inter-module type checking by object  
utilities;

- word count           - for a public procedure entry point, the total number of words of stacked parameters required by the procedure, otherwise 255; this permits object utilities to construct gates for public procedures when necessary; debuggers may also use it to format stacks and create calling sequences; note that a word count greater than 31 is ungatable;
- symbolic name       - a character string name of the public symbol, for matching against external symbol names from other linkable modules.

Operations related to the public definition section are:

`put_public (symbolic_name, type_internal_name, generalized_address, word_count):` defines a public symbol in the current module of the output file;

`emit_public_constant (symbolic_name, type_internal_name, constant_value):` defines a public constant (special internal name) in the current module of the output file;

`get_public` returns `found`, `symbolic_name`, `generalized_address`, `type_internal_name`, `word_count`: finds the next public symbol definition (`found = true`) in the current module of the input file and returns its characteristics.

#### \* EXTERNAL DEFINITION SECTION \*

This section occurs in a linkable module if the module has unresolved external references. It lists all the module's external symbols, the ordering of which then provides internal names for subsequent references to the external symbols. The external definition section has the predictable format

```

      *
! external definition ! ...

```

where an external definition is defined as:

```

      2           2           *           *
! segment internal name ! type internal name ! module name ! symbolic name !

```

with the fields being:

segment internal name - if known, a reference to the segment assumed to contain the matching public symbol, else zero;

type internal name   - refers to the type of the external symbol; supports inter-module type checking by object utilities;

- module name           - if known, the character string specifying the name of the module containing the matching public symbol, else null;
- symbolic name         - a character string name of the external symbol, for matching against the corresponding public symbol's name.

Operations for external definition sections are:

`put_external (symbolic_name, module_name, segment_internal_name, type_internal_name):`  
     defines an external symbol within the current module of the output file and returns an internal name for the external symbol;

`get_internal_returns symbolic_name, module_name, segment_internal_name, type_internal_name, external_symbol_internal_name:`  
     finds the next external symbol definition in the current module of the input file and returns its characteristics and internal name.

#### \* GATE DEFINITION SECTION \*

The gate definition section may appear in linkable modules. Typically it is placed there by an object utility (and not by a translator), since the beauty of gates is their transparency to programming languages. The gate definition section has an entry for each gate in the module, with the order providing an internal name for subsequent reference to the gate. Names and word counts for gates appear in the public definition section, whereas the gate definition itself provides additional characteristics of the gate, as shown below:

1	1	4
<u>! privilege ! gate type ! gate entry generalized address !</u>		

where

- privilege   - the gate's DPL;
- gate type   - one of call gate, task gate, interrupt gate, trap gate, or illegal;
- gate entry   - the generalized address representing the gated entry point.

Operations for the gate definition section are:

`put_gate (privilege):`  
     defines a gate in the current module of the output object file and returns an internal name for the gate; the name of the gate is defined using a public symbol definition; other fields in the gate definition are supplied by text definitions directed to the gate internal name;

get\_gate returns privilege, internal name:

finds the next gate definition in the current module of the input object file and returns its descriptor privilege level and internal name.

### \* TEXT AND FIXUPS SECTION \*

Actually this is the real body of a translator-produced object file. This section appears in linkable modules and consists of intermixed object text (i.e. code or constant data) and of fixups (i.e. address relocation requests) to that text. There is an ordering constraint that fixups not precede the text to which they apply.

A block of text appears in the section as:

1	4	2	*
<u>! 0 (text code) ! generalized address ! length ! content !</u>			

where

generalized address - specifies the desired address of the first byte of text; successive bytes of text are placed at successively increasing addresses; the internal name portion for translators always specifies a segment, but object utilities may also specify a gate;

length - length of the text content, in bytes;

content - a sequence of bytes.

A block of fixups appears as:

1	2	2	*
<u>! 1 (fixup code) ! where internal name ! length ! fixup ! ...</u>			

where we have

where - an internal name specifying the segment or gate to which the following fixups should be applied;

length - the length in bytes of the following fixups.

Fixups specify that a generalized address "what" be placed at a generalized address "where" in some format. They appear in several forms, as shown below:

#### general fixup

1	2	2	2
<u>! kind ! where offset ! what offset ! what internal names !</u>			

#### intra-segment fixup

1	2	2
<u>! kind ! where offset ! what offset !</u>		

call fixup

1	2	2
! kind	! where	! what internal name !

where we have

kind

3b	3b	2b
! reserved	! form	! type !

with

type - one of above fixup types (general, intra-segment, or call);

form - the format of the address to be placed into the text, one of selector only, offset only, self-relative offset, full virtual address, byte constant, or special debug address.

The general fixup specifies that the generalized address ('what' internal name--offset pair) should be placed into the program at the generalized address ('where' internal name--offset pair). The 'what' internal name may specify a segment, gate, external symbol, GDT selector, or constant. Translators will use typically segments or external symbol internal names, even if the form indicates self-relative; the object utilities will make the necessary adjustments in the self-relative case. The intra-segment fixup is equivalent to a general fixup with the assumption 'what' internal name equals 'where' internal name. Similarly, the call fixup is equivalent to a general fixup with 'what' offset of zero. Typically call fixups are used for calling external procedures.

Byte constant fixups use a 'form' of byte constant and the low-order 'what' offset byte, whereas word constant fixups use a constant 'what' and an offset form fixup.

Note that fixups are quite general and permit translators the luxury of avoiding most of the work typically involved in code assembly, such as building label tables and taking two passes to resolve forward references, etc. Fixups also permit delayed binding of constants, transfer vectors, and even opcodes (e.g. as Intel translators do for 8087 or 8087 emulator opcodes). The object language as a whole provides considerable flexibility for translator or system designers.

Operations applying to the text and fixup section include:

put\_text\_header (text\_generalized\_addresses, text\_length):  
initiates a text block on the output file; must be followed by one or more calls to 'put\_text\_contents';

put\_text\_contents (text\_pointer, length):  
used after 'put\_text\_header' to output a text item; the sum of the lengths in successive calls must equal the length in the header;

(NOTE: Translators use a more primitive interface shown next.

`emit_start (text_generalized_address):`  
 initiates a text block (length to be determined); followed by calls to 'emit\_byte', 'emit\_word', and 'emit\_text';

`emit_byte:` emits the byte of text stored at a known global location (to avoid parameter overhead);

`emit_word:` emits the word of text stored at a known global location;

`emit_text:` emits the byte sequence of text whose address and length are stored at a known global location);

End NOTE.)

`get_text_or_fixup:`  
 finds the next text or fixup block in the current module of the input file and returns the type of the block (text or fixup);

`get_fixup (what_generalized_address, where_generalized_address, form):`  
 reads the next fixup definition in the current module of the indicated file and returns its fields;

`get_text_header (text_generalized_address, length):`  
 reads the next text definition in the current module of the input file and returns its header fields; should be followed by one or more calls to 'get\_text\_contents';

`get_text_contents (text_pointer, length):`  
 used after 'get\_text\_header' to obtain the contents of a text definition; long text definitions may be processed using repeated calls.

#### \* REGISTER INITIALIZATION SECTION \*

This section is present in the linkable module associated with the main program only and provides some of the information required to produce an initial task state segment. Its format it:

```

    4      2      2
    ! CS-IP ! SS ! DS !
  
```

where

- CS-IP - a generalized address for the initial values of CS and IP;
- SS - an internal name for the initial value of SS;
- DS - an internal name for the initial value of DS.

All internal names in this section must be segment internal names, while a value of zero may be used to signify no initialization.

Associated operations are:

put\_register (CS\_IP\_generalized\_address, SS\_internal\_name,  
DS\_internal\_name):

defines initial register contents; ES is initialized to DS;  
SP and BP are initialized pointing to the top of the stack;

get\_register returns CS\_IP\_generalized\_address, SS\_internal\_name,  
DS\_internal\_name:

returns the contents of the register initialization section  
for the current module of the input file.

### \*TYPE DEFINITION SECTION \*

A type definition section may appear in linkable modules and provides descriptions of the types of the symbols found in the public and the external definition sections. This supports inter-module type checking by object utilities. Type internal names reflect the order of type definitions in the section. The section itself is:

\*

! type definition ! ...

while a type definition is:

1            2            \*

! linkage ! length ! type string !

where

linkage - true for type definitions referenced in the public or external sections, false otherwise (debug types);

length - the length in bytes of the following type string;

and type string is defined as:

\*

! leaf ! ...

where the leaves are a function of the particular type and are fixed in the interface design among the translator, debugger, and other language implementators. Leaf specifications must be set compatibly for all languages that might participate in a multi-language application or share a debugger.

The type definition scheme is so encompassing that it can properly be called a type "sublanguage" of the object language. From a translator viewpoint the type language is extensible, since it can be extended to support new types for new or existing languages, or new attributes for existing types. From an object utilities viewpoint it is semantically transparent since object utilities need not know the relationship between types and leaves, and in fact extensions to the type system supported by the translators do not affect the object utilities or existing object files--type checking is not affected!

More importantly, though, the system is flexible enough to provide inter-language type checking, even between strongly and weakly typed languages. Moreover, strongly typed languages may control the degree to which they permit type compatibility with weakly typed languages. Arbitrarily complex or recursive structured types are also supported by the mechanisms. The design of the sublanguage was both a technical and an economic breakthrough--technical because of its elegance and flexibility, and economic because it made the object utilities and debuggers language-independent. Its design is by Jim Stein (23).

As stated previously, a type string is a sequence of "leaves". These are the nodes of a "type tree", that in turn is composed of "branches", and branches are composed of leaves. Leaves are just primitive byte strings representing type information. Branches are conceptually infinite sequences of leaves, where a finite leaf sequence is considered to be extended by an infinite sequence of null leaves. The type checking performed by object utilities, then is just a relatively simple-minded tree matching algorithm, without any understanding of the underlying type semantics. Since all branches are the same length (i.e. infinite), strongly typed languages can supply additional type attributes (as leaves) that may (or may not) be matched against implicit null leaves in corresponding branches generated for weakly typed languages. For example, a one-byte boolean variable in Pascal may (or may not) be allowed to match a one-byte untyped variable in an assembly language program (since the Pascal branch will have the extra attribute leaf 'boolean').

Leaf types: Primitive leaves are of four types:

- a numeric leaf - an unsigned byte value;
- a string leaf - a (possibly empty) sequence of ASCII characters;
- an index leaf - an index between one and 32767 used to refer to another type via its internal name;
- a null leaf - a null valued leaf used to represent "don't care" attribute fields or to infinitely extend branches; null leaves may act as wild cards for matching purposes (called "easy" nulls) or may be restricted to matching only other null leaves ("nice" nulls).

The leaves are formatted as:

<u>easy null</u>	$\frac{1}{! 128 !}$
<u>nice null</u>	$\frac{1}{! 129 !}$
<u>string leaf</u>	$\frac{1 \quad 1}{! 130 ! \text{ character} ! \dots}$
<u>index leaf</u>	$\frac{1 \quad 2}{! 131 ! \text{ type internal name} !}$

$$\begin{array}{l}
 \text{numeric leaves} \quad \frac{1}{! m !} \quad , 20 \leq m < 27 \\
 \\
 \frac{1 \quad 2}{! 133 ! n !} \quad , 0 \leq n < 216 \\
 \\
 \frac{1 \quad 4}{! 134 ! n !} \quad , 0 \leq n < 232 \\
 \\
 \frac{1 \quad 8}{! 135 ! n !} \quad , 0 \leq n < 264 \\
 \\
 \frac{1 \quad 1}{! 136 ! s !} \quad , -27 \leq n < 27 \\
 \\
 \frac{1 \quad 2}{! 137 ! s !} \quad , -215 \leq n < 215 \\
 \\
 \frac{1 \quad 4}{! 138 ! s !} \quad , -231 \leq n < 231 \\
 \\
 \frac{1 \quad 8}{! 139 ! s !} \quad , -263 \leq n < 263
 \end{array}$$

The matching rules for leaves are:

- a) two numeric leaves match if their values are equal (even though their formats may differ);
- b) two string leaves match if the strings are identical;
- c) two index leaves match if the types referenced by them match;
- d) two null leaves match (nice, easy, or mixed);
- e) an easy null leaf matches any other kind of leaf;
- f) no other leaves match.

Certain numeric leaves are predefined with special meanings for type purposes, but to object utilities they are just numeric leaves:

127	list	113	const
126	real	112	integer
125	signed integer	111	char
124	unsigned integer	110	false
123	scalar	109	true
122	pointer	108	boolean
121	structure	107	chameleon
120	array	106	set
119	dimension	105	unpacked
118	parameter	104	packed
117	procedure	103	file
116	short	102	interrupt
115	long	101	selector
114	label		

For readability in the following discussion they will be represented as names

(e.g. ! scalar !) rather than numbers, but to a type checker they are just numbers without meaning. Index leaves will be represented as

3  
! @ type !

These are the primitive leaf types and are sufficient for our purposes. Stein's original proposal (23) provided for structured leaf types (conjunctions and disjunctions of leaves) and for general ordered comparisons for numeric leaves (<, >, =, etc.) that together verged almost on the expressive power of programming language expressions. These features have not been exploited to date.

A branch is just an infinite sequence of leaves, only a finite number of which are not easy nulls. Two branches match if all their corresponding (in linear order) leaves match. One branch refers to another by means of an index leaf, and in this manner a type tree is formed. Since some languages permit cyclic or recursive type definitions (e.g. Pascal), in some cases a general type graph will actually be represented rather than a tree, and so type checkers must take care not to cycle through infinite comparison loops.

High-Level Language Types: Figure 6.1 shows the correspondences that can be drawn among types in four popular programming languages implemented for the iAPX 286 (24). These correspondences are used to guide the design of type representations in the type sublanguage so that the indicated linkages are possible among the various languages without obtaining error or warning messages from an inter-module type checker. We'll now describe the representations:

Scalar Type Representations: The general pattern for representing scalar types is the following type branch:

```

      1      *      *      *
      ! scalar ! (length) ! (machine type) ! (type name string) !
      *      *      *
      ! (language type) ! (lower bound) ! (upper bound) !

```

where the length is given in bits. Since more than one language type may map onto the same machine type (for strongly typed languages), both a language type field and a machine type field are present. A type name field is also present since some languages (e.g. Pascal) require identical type names as well. We now describe the various scalar types as implemented in the four languages.

Signed or unsigned integers (8, 16, or 32-bits) are represented as:

```

      1      1      1
PL/M      ! scalar ! (length) ! signed/unsigned integer !
      1      1      1
Pascal:   ! scalar ! (length) ! signed/unsigned integer !
      *      1 -or- 3
      ! (type name string) ! integer-or-@ enumeration-list !
      *      *
      ! (lower bound) ! (upper bound) !
      1      1      1
Fortran:  ! scalar ! (length) ! signed integer !

```

Cobol: 1 1 1 1  
! scalar ! (length) ! signed/unsigned integer ! easy null !  
1 \* \*  
! integer ! (lower bound) ! (upper bound) !

Note that an enumerated type definition in Pascal will be represented as an unsigned integer branch with an index leaf pointing to an enumerated list branch where the enumerated constant names are listed, as in:

enumeration list: 1 \*  
! LIST ! (constant name string) ! ...

Floating point types (32, 64, or 80-bits) are represented as:

PL/M: 1 1 1  
! scalar ! 32 ! real !

Pascal: 1 1 1 \*  
! scalar ! (length) ! real ! (type name string) ! real !  
1  
! nice null !

Fortran: 1 1 1  
! scalar ! (length) ! real !

Cobol: (no counterpart)

where the nice null in the Pascal specification's range portion indicates that its real variables would be incompatible with real variables carrying a non-null range specification.

A character scalar appears as:

PL/M: 1 1 1  
! scalar ! 8 ! unsigned integer !

Pascal,  
Cobol 1 1 1 \*  
! scalar ! 8 ! unsigned integer ! (type name string) ! char !  
1  
! nice null !

Fortran: 1 1 1 1 1  
! scalar ! 8 ! unsigned integer ! easy null ! char !

where the nice null for Pascal prevents matches with character subrange types, and the nice null for Cobol prevents matches with ASCII numeric types.

A boolean scalar appears as:

PL/M: 1 1 1  
! scalar ! 8 ! unsigned integer !

Pascal: 1 1 1 \* 1  
! scalar ! 8 ! unsigned integer ! (type name string) ! boolean !

Fortran: 1 1 1 1 1  
! scalar ! 8 ! unsigned integer ! easy null ! logical !

Cobol: 1 1 1 1 1  
! scalar ! 8 ! unsigned integer ! easy null ! char ! boolean !  
1  
! nice null !

where we note Pascal's boolean type is not compatible with the other languages' comparable types.

Structured Types: Arrays appear as:

PL/M: 1 \* 3  
! array ! (length) ! @ base type !

Pascal: 1 \* 3 3  
! array ! (length) ! @ base type ! @ index type !  
\* 1  
! (type name string) ! (un)packed !

Fortran: 1 \* 3 3  
! array ! (length) ! @ base type ! @ index type-or-easy null !

Cobol: 1 \* 3 3  
! array ! (length) ! @ base type ! @ index type !

while structures are:

PL/M, Cobol: 1 \* \* 3 1  
! structure ! (length) ! (# of fields) ! @ field list ! list !  
3  
! @ field type ! ...

Pascal: 1 \* \* 3  
! structure ! (length) ! (# of fields) ! @ field list !  
\* 1 1  
! (type name string) ! (un)packed ! nice null !

Fortran: (no counterpart)

Note that Pascal structure representation will be more involved than this for variant records (24). A Pascal set will be:

Pascal: 1 \* 1 1  
! scalar ! (length) ! unsigned integer ! set !  
\* 1 3  
! (type name string) ! (un)packed ! @ member type !

thus permitting short Pascal sets to match unsigned integers in more weakly typed languages.

Pointer types (16 or 32-bits) are represented as:

PL/M:  $\frac{1}{! \text{ scalar } !} \frac{*}{(\text{length}) !} \frac{3}{@ \text{ pointer type } !} \longrightarrow \frac{1}{! \text{ pointer } !}$

Pascal:  $\frac{1}{! \text{ scalar } !} \frac{*}{(\text{length}) !} \frac{3}{@ \text{ pointer type } !} \frac{*}{(\text{type name string}) !}$   
 $\downarrow$   
 $\longrightarrow ! \text{ pointer } ! @ \text{ pointed-to type } !$

Fortran, Cobol: (same as Pascal, sans type name string).

A label is represented as:

PL/M, Pascal, Cobol:  $\frac{1}{! \text{ label } !} \frac{1}{\text{easy null } !} \frac{1}{\text{short/long } !}$

while a constant is:

Pascal:  $\frac{1}{! \text{ constant } !} \frac{3}{@ \text{ constant type } !} \frac{*}{(\text{value}) !}$

Procedures appear as:

PL/M, Pascal, Fortran, Cobol:

$\frac{1}{! \text{ procedure } !} \frac{1}{\text{easy null } !} \frac{3}{@ \text{ function type } !} \frac{1}{\text{short/long } !}$

$\frac{*}{! (\# \text{ of parms}) !} \frac{3}{@ \text{ parameter list } !}$

$\frac{1}{! \text{ list } !} \frac{3}{@ \text{ parameter } !} \dots$

$\frac{1}{! \text{ parameter } !} \frac{3}{@ \text{ parameter type } !}$

and files as:

Pascal:  $\frac{1}{! \text{ file } !} \frac{3}{@ \text{ file type } !}$

Cobol:  $\frac{1}{! \text{ file } !} \frac{1}{\text{nice null } !}$

indicating Cobol files are incompatible with Pascal files.

The following operations apply to the type definition section:

`put_type_header (linkage, length):`  
initiates a type definition on the output file; followed by one or more calls to 'put\_type\_def';

`put_type_def (type string, length):`  
used after 'put\_type\_header' to output a type definition or portion thereof;

`get_type_header` returns linkage, length, type internal name:  
finds the next type definition in the current module of the input file and returns its characteristics; the internal name will be zero if no type definition is found; followed by one or more calls to 'get\_type\_def';

`get_type_def (length)` returns type string:  
used after 'get\_type\_header' to obtain the (partial) contents of a type definition.

NOTE: The following sections may appear only in loadable object files.

#### \* DESCRIPTOR SECTION \*

This section appears in loadable modules, and contains all of the descriptors for the module which contains it:

8  
! descriptor ! ...

where a descriptor is:

6                      2  
! hardware portion ! software word !

with 'software word' defined as:

14b              1b              1b  
! reserved ! absolute ! expandable !

where

- absolute - true if the base address field in the hardware portion of the descriptor contains the base address of the segment;  
false if it must be defined at load time;
- expandable - true if the length of the segment may be expanded at load time; false otherwise.

The format of the hardware portion of a descriptor was defined in Section 1.

If the descriptor section was not built by the system builder (built = false in module header), then this section will become the module's sole LDT after filling in base addresses, expandable limits, and software words. If the descriptor section was built by the system builder (built = true), then this section represents the module's sole GDT and the module's various LDTs are represented explicitly by object table descriptors in the GDT and text items in the loadable text section.

Operations applying to the descriptor section are:

`put_descriptor (hardware_portion, expandable, absolute):`  
 outputs a descriptor definition to the current module of the output file and returns an internal name for subsequent reference;

`get_descriptor` returns `hardware_portion`, `expandable`, `absolute`, and `internal_name`:  
 returns the next descriptor on the input file.

#### \* DESCRIPTOR NAME SECTION \*

This section may appear in loadable modules and provides symbolic names for descriptors in the object module (e.g. a segment's symbolic name will also be its combine name). This information plays no direct role in making a module executable, but may be used by operating system routines for job control, by the system builder for user reference, or by debuggers. It appears as:

\*  
! descriptor name ! ...

where a descriptor name is:

2                      \*  
! internal name ! name string !

Operations are:

`put_desc_name (internal_name, name_string):`  
 outputs a descriptor name to a loadable file;

`get_desc_name` returns `internal_name`, `name_string`, `found`:  
 returns the contents of the next descriptor name item in the input file (`found = true`), or else `found = false`.

#### \* LOADABLE TEXT SECTION \*

This section appears in every loadable module and contains the program text (code and constant data) to be loaded. It appears as:

\*  
! text item ! ...

where each text item is:

2	2	2	*
!	offset	!	descriptor internal name
!	length	!	text
!		!	*

and where

- offset - an offset into the segment (or object table, etc.) designated by the internal name;
- descriptor internal name - internal name as previously described;
- length - the length of the following text in bytes;
- text - a sequence of bytes.

In non-'built' modules (i.e. single LDT, single task), the first text item must be the initial task state block, and all subsequent text items must have segment internal names only. In modules produced by the system builder, the first text item must also be the initial task state block; however, following text items may use any type of internal name.

Operations are 'put\_text\_header' and 'put\_text\_contents' as described previously for text sections.

#### \* LOADABLE FIXUP SECTION \*

This section optionally appears in loadable modules and is used by sophisticated loaders which perform load-time relocation of selector slots in the LDT, such as might be done if several separate tasks were bound to the same LDT by a dynamic loader. Generally this is not done unless a "task family" concept is supported by the system.

The section appears as:

*
!
selector locator
!
...

where a selector locator is:

2	2	2
!	descriptor internal name	!
!	length	!
!	offset	!
!	...	!

with

- descriptor internal name - specifies the internal name for a segment;
- length - length of the trailing offsets in bytes;
- offset - a byte offset in the specified segment that contains a selector (for any descriptor).

Operations are:

`put_lod_selector_fixup (generalized address):`  
     outputs the internal name and offset of a segment locator;

`get_lod_selector_fixup returns generalized address, found:`  
     gets the next selector fixup item from the section.

### \* DEBUG TEXT SECTION \*

This section optionally occurs in loadable modules. In linkable modules, debug information is represented by means of ordinary text items and ordinary fixups scattered among the non-debug text and fixups. Debug text and fixups are distinguished in linkable modules only by being directed to segments with the debug attribute (see description of segment definition section). Object utilities that create loadable modules separate text and fixups directed to debug segments from the ordinary text so that non-debug loaders may ignore it. Debug segments are not listed in the descriptor section.

This section appears as:

```

      *           *
! table of contents ! text !

```

where table of contents is:

```

      2           2           4
! # of debug segments ! length ! location ! ...

```

with

length - the length of the 'location' tail in bytes;

location - the location on disk of a debug segment, specified as an offset in bytes from the start of the file.

Internal names for debug segments refer to the order in the table of contents. The number of debug segments, their order of occurrence, their functions, and their formats are determined by mutual agreement between translator and debugger designers. Intel has adopted the following conventions.

There are six debug segments, named 'modules', 'types', 'symbols', 'lines', 'publics', and 'externals'. Translators generate special text for modules, symbols, and lines, whereas the object utilities will convert ordinary type, public, and external definition sections in linkable modules into text for the other three segments.

The debug 'modules' segment contains of the following for each linked module:

2	4	
<u>! LDT selector ! code segment generalized address !</u>		
4		4
<u>! types generalized address ! symbols generalized address !</u>		
4		4
<u>! lines generalized address ! public's generalized address !</u>		
4		2
<u>! externals generalized address ! first line number !</u>		
1	4	*
<u>! translator identifier ! version number ! module name !</u>		

where

- |                       |   |   |                      |      |                          |        |                             |         |                           |
|-----------------------|---|---|----------------------|------|--------------------------|--------|-----------------------------|---------|---------------------------|
| LDT selector          | - used only for multi-LDT modules produced by system builders, otherwise zero; selector into the GDT for an LDT that contains the segments in this module; LDT references in the debug segment will be evaluated with respect to this LDT;  |   |                      |      |                          |        |                             |         |                           |
| generalized address   | - the generalized address of the corresponding segment; since more than one linkable module will be represented in each debug segment, the offset portion of the address indicates the offset into the appropriate debug segment for the current module; debug form fixups must be issued for these generalized addresses (see text and fixup section); |   |                      |      |                          |        |                             |         |                           |
| first line number     | - the first line number assigned to lines in this module (used to convert module offsets back into line numbers via the 'lines' segment);   |   |                      |      |                          |        |                             |         |                           |
| translator identifier | - an unsigned byte value in the range 0..255; <table border="0" style="margin-left: 40px;"> <tr> <td>0</td> <td>- translator unknown</td> </tr> <tr> <td>1-49</td> <td>- reserved for Intel use</td> </tr> <tr> <td>50-199</td> <td>- reserved for customer use</td> </tr> <tr> <td>200-255</td> <td>- reserved for future use</td> </tr> </table>      | 0 | - translator unknown | 1-49 | - reserved for Intel use | 50-199 | - reserved for customer use | 200-255 | - reserved for future use |
| 0                     | - translator unknown  |   |                      |      |                          |        |                             |         |                           |
| 1-49                  | - reserved for Intel use  |   |                      |      |                          |        |                             |         |                           |
| 50-199                | - reserved for customer use   |   |                      |      |                          |        |                             |         |                           |
| 200-255               | - reserved for future use   |   |                      |      |                          |        |                             |         |                           |
| version number        | - four ASCII characters, typically Vd.d for a released translator or Xddd for an unreleased (experimental) translator.  |   |                      |      |                          |        |                             |         |                           |

Lines Segment: The 'lines' segment, emitted by translators as text for the 'lines' segment (for usage by debuggers) appears as a sequence of line offsets:

2
<u>! line offset ! ...</u>

within the code segment, representing the corresponding code segment offset values for this module's line numbers, beginning with 'first line number'. No fixups are needed for this segment, since these offsets will be added to the code segment virtual address (in the 'modules' segment) by the debugger to obtain the line's virtual address (and there is a fixup for the 'code segment generalized address' of the 'modules' segment). This is important since a long sequence of lines offset fixups would make debug object files unacceptably large.

Symbols Segment: Entries for the 'symbols' segment are produced by translators as text items for the 'symbols' debug segment. It contains one entry for each symbol used in the module. Note that pointer fixups are required for the generalized addresses in symbol base entries. The formats of the various entries are:

block start:       $\begin{array}{cccc} 1 & & 2 & & 2 & & * \\ \hline ! 0 ! & \text{code segment offset} & ! & \text{block length} & ! & \text{block name} & ! \end{array}$

procedure start:       $\begin{array}{ccc} 1 & & 2 \\ \hline ! 1 ! & \text{code segment offset} & ! \text{type index} ! \end{array}$

$\begin{array}{cc} 1 & 2 \\ \hline ! \text{near/far/interrupt} ! & \text{offset of return adr (from BP)} ! \end{array}$

$\begin{array}{cc} 2 & * \\ \hline ! \text{procedure length} ! & \text{procedure name} ! \end{array}$

block or procedure end:       $\begin{array}{c} 1 \\ \hline ! 2 ! \end{array}$

symbol base:       $\begin{array}{cc} 1 & 4 \\ \hline ! 3 ! & \text{generalized address} ! \end{array}$

stack (BP) relative base:       $\begin{array}{c} 1 \\ \hline ! 4 ! \end{array}$

symbol:       $\begin{array}{cccc} 1 & 2 & 2 & * \\ \hline ! \text{kind} ! & \text{offset} ! & \text{type index} ! & \text{symbol name} ! \end{array}$

kind:      5--symbol, 6--offset based symbol, 7--pointer based symbol, 8--selector based symbol.

Symbols appearing in a particular block or procedure should have entries defined between the start entry and the matching end. These start entries are optional but, when used, indicate symbol scoping to the debuggers. Offsets in symbol entries are relative to the base specified by the nearest preceding symbol base entry or stack relative base entry. Offset based symbols are referenced indirectly by debuggers (e.g. call by reference parameters), with the (symbol base, offset) pair giving the memory location of the symbol address' offset portion (while the symbol base of course gives the base portion). A full 32-bit indirect address is associated with pointer based symbols, with the indirect address appearing in memory at the location (symbol base, offset).

Types Segment: Type definitions for debug symbols will be output by translators as part of the type definition section. During binding of linkable modules, the binding object utility will convert the type definition section into text for the 'types' debug segment. Entries have the same format as type definitions in the type definition section, namely:

```

      1      2      *
    ! linkage ! length ! type string !

```

Publics Segment: The public definition section emitted by translators will be converted by the binding object utility into text for the 'publics' debug segment. Entries have the same format as public definitions in the public definition section, namely:

```

      4      2
    ! generalized address ! type internal name !

```

```

      1      *
    ! word count ! symbolic name string !

```

Externals Segment: The binding object utility will convert external definitions in the external definition section into identically formatted text in the 'externals' debug segment, namely:

```

      2      2
    ! segment internal name ! type internal name !

```

```

      *      *
    ! module name ! symbol name !

```

Operations include:

`put_debug_segment (name):`

used by object utilities to output a debug segment entry to the debug text table of contents; returns an internal name for subsequent reference by 'put\_debug\_text';

`put_debug_text (internal_name):`

initiates debug segment text on the output file; followed by one or more calls to 'put\_debug\_text\_contents';

`put_debug_text_contents (text, length):`

outputs the contents of the debug segment specified in a previous call to 'put\_debug\_text';

`get_debug_segment` returns name, length, internal\_name:

finds the next debug segment entry in the debug text table of contents and returns its characteristics (internal\_name ≠ 0);

`get_debug_text (internal_name, disk_offset):`  
 positions the disk file pointer to the indicated offset within the referenced debug segment; followed by one or more calls to 'get\_debug\_text\_contents';

`get_debug_text_contents (length)` returns text:  
 used after 'get\_debug\_text' to obtain the contents of a debug item.

### \* DEBUG FIXUP SECTION \*

The debug fixup section optionally appears in loadable modules (and performs the same function for the debug text section that the loadable fixup section performs for the loadable text section). It is

\*  
! selector locator ! ...

where a selector locator is:

$\begin{array}{ccc} 2 & 2 & 2 \\ \hline ! \text{ debug segment internal name ! length ! offset ! } \dots \end{array}$

where

- debug segment internal name - identifies the debug segment;
- length - byte length of the offset tail portion of the selector locator;
- offset - a byte offset within the segment referenced by the internal name; a selector should appear in the debug segment at location (internal name, offset).

Operations are:

`put_deb_selector_fixups (generalized_address):`  
 outputs the internal name and offset of a debug segment selector locator in the debug fixup section;

`get_debug_fixups` returns found:  
 returns true if any debug segment fixups are present;

`get_deb_selector_fixup` returns generalized\_address, found:  
 gets the next selector fixup item for the debug fixup section (found = true).

### \* ABSOLUTE TEXT SECTION \*

This section appears only within boot loadable modules and specifies initial main memory contents. It is represented as:

```

      *
! absolute text item ! ...

```

where an absolute text item is:

```

      3      2      *
! real address ! length ! text !

```

with

real address - the starting main memory address of the following text;  
length - the length of the following text, in bytes;  
text - text to be loaded at the indicated real address.

Operations are:

put\_abs\_text\_header (real\_address, length):  
initiates an absolute text item; followed by one or more calls to 'put\_text\_contents' (previously described for text section);

get\_abs\_text\_header returns real\_address, length, found:  
returns the base address and byte length for an absolute text item; followed by one or more calls to 'get\_text\_contents' (as previously described).

### \* AN EXAMPLE \*

Figure 6.2 shows a simple two-module example program written in PL/M for the iAPX 286. Module A declares three global variables: 'AGlob1', a public (exported) variable; 'BGlob1', an external (imported) variable; and 'AGlob2', a private module variable. Module A also contains a private procedure, 'AProc', and references an external procedure, 'BProc'. We compile Module A and Module B in separate compilations using the PL/M-286 compiler, examine the actual linkable object modules produced, bind them together, and examine the loadable module.

Following, in nested record notation, is the linkable file produced for Module A (with numbers in hexadecimal form and all null fields (zero, false, etc.) ignored):

Module A linkable file:

linkable file (file header(file type = linkable file),

linkable module(linkable module header (total length = X'2B4' bytes,  
# of segments = 7, # of publics = 1, # of externals = 2,  
date = 10/14/82, module name = MODULEA,

```

module creator = PL/M-286 COMPILER X034, table of contents (
  seg def location = X'AF', length = X'6A',
  type def location = X'119', length = X'35', pub def location = X'14E',
  length = X'E',
  ext def location = X'156', length = X'17', text fix location = X'173',
  length = X'138',
  regint location = X'2AB', length = X'8')),

segment definition section (
  segment definition1 (attributes(access = execute-read, DPL = 3,
    combine type = normal),
    slimit = X'2B', combine name = CODE),
  segment definition2 (attributes(access = read-write, DPL = 3,
    combine type = normal),
    slimit = X'5', combine name = DATA),
  segment definition3 (attributes(access = read-write, DPL = 3,
    combine type = stack),
    slimit = X'9', combine name = STACK),
  segment definition4 (attributes(access = read-only, DPL = 3,
    combine type = debug),
    slimit = X'28', combine name = MODULES:),
  segment definition5 (attributes(access = read-only, DPL = 3,
    combine type = debug),
    slimit = X'53', combine name = SYMBOLS:),
  segment definition6 (attributes(access = read-only, DPL = 3,
    external = true,
    combine type = debug),
    combine name = SYMBOLS:),
  segment definition7(attributes(access = read-only, DPL = 3,
    combine type = debug),
    slimit = X'F', combine name = LINES:)),

type definition section (
  type definition1 (linkage = true, length = 1, type string (easy null)),
  type definition2 (linkage = true, length = 3, type string (scalar,
    16-bits, unsigned integer)),
  type definition3 (linkage = true, length = 4, type string (parameter,
    @2)),
  type definition4 (linkage = true, length = 4, type string (list, @3)),
  type definition5 (linkage = true, length = 8, type string (procedure,
    easy null, easy null, short, 1-parm, @4)),
  type definition6 (length = 4, type string (list, @3)),
  type definition7 (length = 8, type string (procedure, easy null,
    easy null, short, 1-parm, @6))),

public definition section (
  public definition1 (generalized address(internal name = seg def2,
    type internal name = type def2, symbolic name = AGLOB1)),

external definition section (
  external definition1 (segment internal name = seg def2,
    type internal name = type def2, symbolic name = BGLOB1),
  external definition 2 (segment internal name = seg def1,
    type internal name = type def5, symbolic name = BPROC)),

```

```

text and fixup section (
  text1 (generalized address(internal name = seg def5), length = X'54',
    block start (block length = X'08, block name = MODULEA),
    symbol base (generalized address (0)),
    symbol (offset = 2, type index = type def2, symbol name = AGLOB2),
    block start (code segment offset = X'08', block length = X'24',
      block name = APROC),
    procedure start (code segment offset = X'08', type index = type def7,
      near offset of return adr = X'02', procedure length = X'24',
      procedure name = APROC),
    stack relative base,
    symbol (offset = X'04', type index = type def2, symbol name = ARG),
    symbol base (generalized address (0)),
    symbol (offset = X'04', type index = type def2, symbol name = ALOC),
    block or procedure end, block or procedure end),
  text2 (generalized address(internal name = seg def1, offset = X'08'),
    length = 3, content = X'55 8B EC'),
  text3 (generalized address(internal name = seg def7),
    length = 4, content = X'08 00 0B 00'),
  text4 (generalized address(internal name = seg def1, offset = X'0B'),
    length = 7, content = X'8B 46 04 3B 06 04 00'),
  fixup1 (where internal name = seg def5, length = X'0A',
    call fixup (kind(form = full virtual address), where offset = X'03',
      what internal name = seg def2),
    call fixup (kind(form = full virtual address), where offset = X'44',
      what internal name = seg def2)),
  text5 (generalized address(internal name = seg def1, offset = X'12'),
    length = X'1A', content = X'77 03 E9 09 00 68 01 00
E8 00 00 E9 08 00 8B 06 00 00 89 06 02 00
5D C2 02 00'),
  text6 (generalized address(internal name = seg def7, offset = X'04'),
    length = 6, content = X'0B 00 17 00 20 00'),
  text7 (generalized address(internal name = seg def1),
    length = 8, content = X'8B EC 68 05 00 E8 00 00'),
  text8 (generalized address(internal name = seg def7, offset = X'0A'),
    length = 6, content = X'28 00 00 00 08 00'),
  text9 (generalized address(internal name = seg def4),
    length = X'29', 'MODULES:' entry (first line number = 8,
      translator id = PL/M-286, version number = X034,
      module name = MODULEA)),
  fixup2 (where internal name = seg def1, length = X'18'
    general fixup (kind(form = offset), where offset = X'10,
      what offset = X'04', what internal name = seg def2),
    call fixup (kind(form = self-relative offset), where offset = X'1B',
      what internal name = ext def2),
    call fixup (kind(form = offset), where offset = X'22',
      what internal name = ext def1),
    general fixup (kind(form = offset), where offset = X'26',
      what offset = X'02', what internal name = seg def2)),
  fixup3 (where internal name = seg def4, length = X'0F',
    call fixup (kind(form = full virtual address), where offset = X'02',
      what internal name = seg def1),
    call fixup (kind(form = full virtual address), where offset = X'0A',
      what internal name = seg def5),
    call fixup (kind(form = full virtual address), where offset = X'0E',
      what internal name = seg def7)),

```

register initialization section (CS = seg def<sub>1</sub>, IP = 0, SS = seg def<sub>3</sub>, DS = seg def<sub>2</sub>)).

While the function of most of the object records can be readily deduced from the previous discussion, it would be wise to more closely examine the main body of the object module, namely the text and fixup section. We will ignore the debug text and fixups, since they play the obvious role of passing translator symbol table and source reference information to the debuggers, and focus our attention instead on the actual program text and fixups.

First note that the text for the code segment, seg def<sub>1</sub>, is emitted piecemeal as object records text<sub>2</sub>, text<sub>4</sub>, text<sub>5</sub>, and text<sub>7</sub>, and that text<sub>7</sub> is not even in logical order (but contains the main program body). As described previously, the only ordering constraint on the intermixed text and fixup records in this section is that fixups not precede the text to which they apply. This simplifies the task of object generation for compilers, since they need not manage large buffer areas for object text; in fact buffer management is done in the described procedural interface and text may be dribbled out in units as small as a single byte, for the convenience of translators.

Figure 6.3 shows the dis-assembled object text for the code segment, for reference in discussing the fixup records. Examining the figure, there should be a fixup in fixup<sub>2</sub> for each relocatable address appearing in the code segment (seg def<sub>1</sub>). The internal reference to 'Aloc' in the instruction at offset 'DE' is first, and is handled by the general offset fixup to offset '20'. Second is the 'BProc' external reference in the instruction at offset '1A' that is handled by the call fixup to offset '1B'. Third is the 'BGlob1' external reference in the instruction at offset '20' that is handled by the call fixup to offset '22'. (Note that a call fixup is used since it is equivalent to a general fixup with 'what offset = 0'.) And fourth is the 'AGlob2' internal reference in the instruction at offset '24' that is handled by the general offset fixup to offset '26'.

The other sections and records in the linkable object module for 'ModuleA' are easily understood, and the linkable object file for 'ModuleB' is analogous to that for 'ModuleA' and will not be given here. We'll now study the loadable file produced by the binder utility from the two object files produced by the separate compilations:

loadable file (file header(file type = loadable file),

loadable module (loadable module header(total space = X'4B',  
# of descriptors = 5, date = 10/14/82,  
module creator = iAPX 286 BINDER, X040,  
table of contents (descriptors location = X'6B',  
loadable text location = X'93',  
descriptor names location = X'FE', debug text location = X'11B',  
last location = X'2A0'))),

descriptor section (  
descriptor<sub>1</sub> (0),  
descriptor<sub>2</sub> (hardware portion(slimit = X'27', present = true, DPL = 0,  
segment = true, type = read-write)),  
descriptor<sub>3</sub> (hardware portion(slimit = X'33', present = true, DPL = 3,  
segment = true, type = execute-read)),

```

descriptor4 (hardware portion(slimit = X'08', present = true, DPL = 3,
    segment = true, type = read-write)),
descriptor5 (hardware portion(slimit = X'FFF1'(-15), present = true,
    DPL = 3, segment = true, type = read-write stack))),

```

```
loadable text section (
```

```

    text item (descriptor internal name = -3(initial TSS), length = X'2C',
        text = X'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
            00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
            00 00 1F 00 17 00 27 00 1F 00 00 00'),
    text item (descriptor internal name = 3(CODE), length = X'33',
        text = X'8B EC 68 05 00 E8 00 00 55 8B EC 8B 46 04 3B 06
            04 00 77 03 E9 09 00 68 01 00 E8 0F 00 E9 08 00
            8B 06 06 00 89 06 02 00 5D C2 02 00 55 8B EC 5D
            C2 02 00')),

```

```
descriptor name section (
```

```

    descriptor name (internal name = 2, name string = LDT?),
    descriptor name (internal name = 3, name string = CODE),
    descriptor name (internal name = 4, name string = DATA),
    descriptor name (internal name = 5, name string = STACK)),

```

```
debug text section (
```

```

    table of contents (# of debug segment = 6, length = X'18',
        location "of modules" = X'137', location "of types" = X'189',
        location "of symbols" = X'1E1', location "of lines" = X'25B',
        location "of publics" = X'277', location "of externals" = 0,
    modules (
        module (code segment generalized address(internal name = X'17',
            first line number = 8, translator identifier = PL/M-286,
            version number = X034, module name = MODULEA)
        module (code segment generalized address(internal name = X'17',
            offset = X'26'),
            types generalized address (offset = X'35'), symbols generalized
            address (offset = X'54'), lines generalized address
            (offset = X'10'),
            publics generalized address (offset = X'0E'), first line number = 4,
            translator identifier = PL/M-286, version number = X034,
            module name = MODULEB))),

```

```
types (
```

```

    "type1 thru type7 identical to MODULEA types",
    "type8 same as type1", "type9 same as type2",
    "type10 same as type3",
    "type11 same as type4", "type12 same as type5")

```

```
symbols (
```

```

    block start (code segment offset = 0, block length = X'08',
        block name = MODULEA),
    symbol base (generalized address(internal name = X'1F')),
    symbol (offset = X'02', type index = 2, symbol name = AGLOB2),
    block start (code segment offset = X'08', block length = X'24',
        block name = APROC),
    procedure start (code segment offset = X'08', type index = 7,
        offset of return adr = X'02', procedure length = X'24',
        procedure name = APROC),
    stack relative base,

```

```

symbol (offset = X'04', type index = 2, symbol name = ARG),
symbol base (generalized address(internal name = X'1F')),
symbol (offset = X'04', type index = 2, symbol name = ALOC,
block or procedure end, block or procedure end,
block start (code segment offset = 0, block length = X'07',
block name = BPROC),
procedure start (code segment offset = 0, type index = 5,
offset of return adr = 2,
procedure length = X'07', procedure name = BPROC),
stack relative base,
symbol (offset = X'04', type index = 2, symbol name = ARG),
block or procedure end),
lines (X'0008 0008 0008 0017 0020 0028 0000 0008 0000 0003
0003 0003 0003 0000'),
publics (
public (generalized address(internal name = X'1F'),
type internal name = 2, symbolic name = AGLOB1),
public (generalized address(internal name = X'1F', offset = 6),
type internal name = 2, symbolic name = BGLOB1),
public (generalized address(internal name = X'17', offset = X'2C'),
type internal name = 5, symbolic name = BPROC))).

```

Note that this loadable module's descriptor segment contains five descriptors: (1) a reserved (all zeros) descriptor for operating system usage; (2) a descriptor for the module's LDT; (3) a code segment descriptor; (4) a data segment descriptor; and (5) a stack segment descriptor. Note also that the first text item contains a skeletal initial task state segment for the task represented by this module (internal name of -3). Finally note that the internal names used in the public definition section are actual segment selector values (i.e. contain three low-order local/global and requested privilege level bits), rather than ordinal index values to the descriptor section. The descriptor index is obtained from the selector value by a right shift of three bits.

FIGURE 6.1: HIGH-LEVEL LANGUAGE TYPE CORRESPONDENCES

PL/M	PASCAL	FORTRAN	COBOL
NUMERIC SCALAR TYPES			
--	-128..127	integer*1	pic S9(2) comp
integer	integer	integer*2	pic S9(4) comp
--	longint	integer*4	pic S9(9) comp
--	--	--	pic S9(18) comp
real	real	real	--
--	longreal	double precision	--
--	tempreal	tempreal	--
byte	0..255	--	pic 9(2) comp
address/word	0..65535	--	pic 9(4) comp
--	0..4294967295	--	pic 9(9) comp
--	--	--	pic 9(18) comp
--	--	--	pic S9(n)
--	--	--	pic 9(n)
CHARACTER SCALAR TYPES (n > 1)			
byte	char	character*1	pic x
--	--	character*n	pic x(n)
BOOLEAN SCALAR TYPES			
--	boolean	--	--
byte	--	logical*1	--
--	--	logical*2	--
--	--	logical*4	--
--	--	--	pic 1(n)
STRUCTURED DATA			
(n)	array n **	(n)	occurs n times
structure	record**	***	(any group item)
OTHER TYPES			
--	set	--	--
pointer	pointer	--	--
--	bytes(n)	--	filler pic x(n)
label	label	--	paragraph-name
--	constant	--	--
procedure	procedure/function	subroutine/function	procedure div.
--	Pascal file	--	--
--	--	--	COBOL file

\*\*unpacked, or packed when no length differences result

! \*\*\*FORTRAN's COMMON could be described using structure TYPDEF's  
but object utilities support type checking with PUBLICS only

FIGURE 6.2: TWO-MODULE EXAMPLE PROGRAM WRITTEN IN PL/M-286

```
ModuleA : DO;  
  Declare AGlob1 Word Public;  
  Declare BGlob1 Word External;  
  Declare AGlob2 Word;  
  
  BProc : Procedure(Arg) External;  
    Declare Arg Word;  
  End BProc;  
  
  Aproc : Procedure(Arg);  
    Declare(Arg, Aloc) Word;  
  
  IF Arg > ALoc THEN Call BProc(1);  
    ELSE AGlob2 = BGlob1;  
  End AProc;  
  
  Call AProc(5);  
End ModuleA;
```

MODULE A

```
ModuleB : Do;  
  Declare AGlob1 Word External;  
  Declare BGlob1 Word Public;  
  
  BProc : Procedure(Arg) Public;  
    Declare Arg Word;  
  
    Do; /* Something */ End;  
  End BProc;  
  
End ModuleB;
```

MODULE B

FIGURE 6.3: DIS-ASSEMBLED CODE SEGMENT TEXT

	<u>Offset</u>	<u>Object Code</u>	<u>Dis-assembly</u>
text <sub>7</sub>	00	8B EC	mov BP, SP ; initialize BP to stack base
	02	68 05 00	push #5 ; push argument
	05	E8 00 00	call 0 ; AProc at relative location zero
text <sub>2</sub>	08	55	push BP ; AProc entry
	09	8B EC	mov BP, SP ; point to new stack frame
text <sub>4</sub>	0B	8B 46 04	mov AX, 4 [BP] ; Arg reference
	0E	3B 06 04 00	cmp AX, 4 ; ALoc reference
text <sub>5</sub>	12	77 03	ja 03 ; to offset '17' if Arg > ALoc
	14	E9 09 00	jmp 09 ; to offset '20' if not
	17	68 01 00	push #1 ; push argument
	1A	E8 00 00	call 0 ; BProc reference
	1D	E9 08 00	jmp 08 ; to offset '28'
	20	8B 06 00 00	mov AX, 0 ; BGlob1 reference
	24	89 06 02 00	mov 2, AX ; store into AGlob2
	28	5D	pop BP ; point to caller's stack frame
	29	C2 02 00	ret #2 ; pop 2 bytes of parameters

## 7.0 OBJECT UTILITIES

Object modules produced by programming language translators are combined into usable programs, libraries, and systems by a set of iAPX 286 object utilities. These utilities manipulate object modules expressed in the object language of Section 6, and they consist of the following utility programs:

- **Binder** - combines linkable modules into either another linkable module (for incremental or hierarchical binding of large programs) or into a loadable module (for loading and execution); the input linkable modules to the binder can come from the following sources:
  - (i) linkable modules produced by iAPX 286 translators,
  - (ii) linkable modules contained in object libraries,
  - and (iii) linkable modules produced previously by the binder; the binder is the most frequently used object utility since it is employed in every compile-link-execute cycle;
- **Builder** - combines linkable, and/or loadable, modules into "systems", where we define system to be either: (i) a bootloadable module (typically the resident portion of the operating system), or (ii) a loadable module for a multi-task application; typically the binder is first employed to generate a loadable module for each task in the system, followed by the builder to combine the loadable modules into a system by creation of the iAPX 286 processor recognized tables (GDT, IDT, LDTs) with the appropriately established entries; the builder is principally used by system developers, and not by typical applications programmers;
- **Librarian** - creates, modifies, and examines libraries of linkable modules; linkable modules may be produced either by translators or by the binder; the librarian is a generally useful tool typically employed by the project librarian in a programming team;
- **Miscellaneous** - assorted other utilities exist or are planned such as:
  - mapper--a tool to produce readable segment, gate, and symbol maps and cross-references;
  - converter--a tool to convert 86 family object modules to iAPX 286 format as automatically as possible;
  - overlay builder--a builder to support (the archaic notion of) overlays;
  - edobj and objed--tools to manually patch object modules (and intended only for skilled object surgeons).

Figure 7.1 shows an overview of the possible object module flows among the three tools--binder, builder, and librarian. Generally the downward and right pointing arrows show the usual direct flow from translation to execution, while the upward and left pointing arrows show recycling of modules for libraries or for incremental (hierarchical) creation of programs and systems. The straight diagonal, from source modules to execute, is the normal compile-link-execute path. System developers are free to use as much or as little of the object utilities' flexibility as is appropriate to their task.

Figure 7.1: Possible object module flows

In the remainder of this section we will explore the functional capabilities of the object utilities without all the details. The relevant literature (25, 26) can be consulted for specifics.

#### \* BINDER\*

Figure 7.2 shows the module inputs and outputs for the binder utility. The binder accepts linkable modules from any combination of translators, libraries, and previous bindings (as well as builder-produced export modules, to be discussed later), and produces exactly one output module (that is either linkable or is loadable). Additionally a link map is produced that describes the contents of the output module. The functions performed by the binder include:

- creates a loadable module or a linkable module by combining input linkable modules;
- automatically retrieves and combines required library modules referred to by external references;
- resolves symbolic inter-module references among the input modules and performs inter-module type checking;
- combines mergeable segments in input modules according to the rules given in Section 6;
- fixes or optionally filters debug information provided by translators; and
- produces output module statistics, error messages, and maps.

Ordinarily the binder is employed to directly combine the input linkable modules into a loadable module. When it does so, it:

- a) combines all segment definitions with the same combine name and creates a descriptor section containing descriptors for all non-external, non-debug segments and all gate definitions, if any, from the input modules; order and location of descriptors will be set arbitrarily by the binder except for those segments that specify fixed LDT positions;
- b) creates a descriptor name section for the loadable module;
- c) gathers all debug information and creates a debug text section;
- d) creates a loadable text section with the program text from all text and fixup sections in the input modules; computes offsets for general and intra-segment fixups; replaces segment internal names by their corresponding descriptor internal names;
- e) creates loadable fixup and debug fixup sections, containing information about the locations of selectors (referring to LDTs) in the corresponding text sections (loadable text and debug text sections);

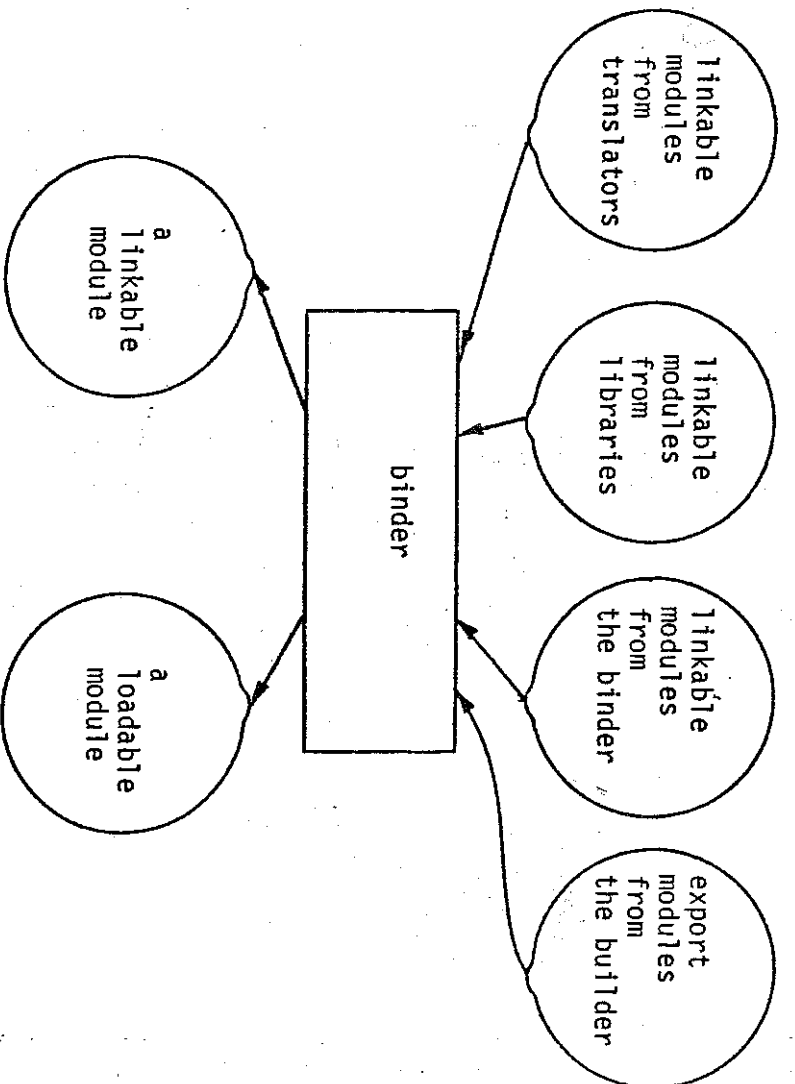


Figure 7.2: The binder - its module inputs and outputs

- f) creates a task state segment (TSS) as the first text item in the loadable text section (see Section 6 example) using information from the register initialization section found among the input modules; in the TSS, ES will be set equal to DS, and BP equal to SP; the I/O privilege level will be set to zero, the link field to zero, and all other flags and registers to zero; the LDT selector field will be set to zero (at load time the loader will convert the descriptor section into an LDT and install a selector for this LDT into this field);
- g) creates a writeable descriptor alias for the LDT and places it in LDT slot number 1 with the name 'LDT?';
- h) issues a warning message if any unresolved external symbols are in the output module.

The binder may, at the user's option, create a linkable module instead of a loadable module. This can be used for incremental linking or for system building (via the builder). To support hierarchic linking schemes, the binder gives the user the facility for selectively or entirely purging public symbols in the output module. Note that in the case of a linkable output module that descriptor and descriptor name sections are not produced, but appropriate segment combining and offset adjusting will be done for sections (e.g. public definition or text and fixup sections) using segment internal names.

#### \* BUILDER \*

The iAPX 286 system builder primarily is used to construct the processor recognized descriptor tables (GDT, IDT, and LDTs) for collections of linkable and loadable modules. In the process it can construct bound memory images (bootloadable modules) for iAPX 286 systems and partially bound (loadable) modules for multi-task applications. Another important use of the builder, that relates to its role as a system table constructor, is to produce a special abbreviated form linkable module called an export module (containing exported gate, public, module, or segment definitions). An export module is used generally by applications as a "system interface library" to describe the programmatic interface to the operating system or application system, and so supports the iAPX 286 architecture's gate mechanism. Export modules are used during the binding phase of application program construction, as input modules to the binder (see Figure 7.3).

The binder performs the following functions as requested:

- sets segment limit, access rights, and privilege level, and can bind segments to physical addresses;
- creates gates for controlled transfers (task or call gates for GDT/LDTs, task, trap, or interrupt gates for the IDT);
- creates descriptor tables: one GDT, one IDT, and zero or more LDTs;
- creates task state segments (TSSs) and task gates for multi-task applications;

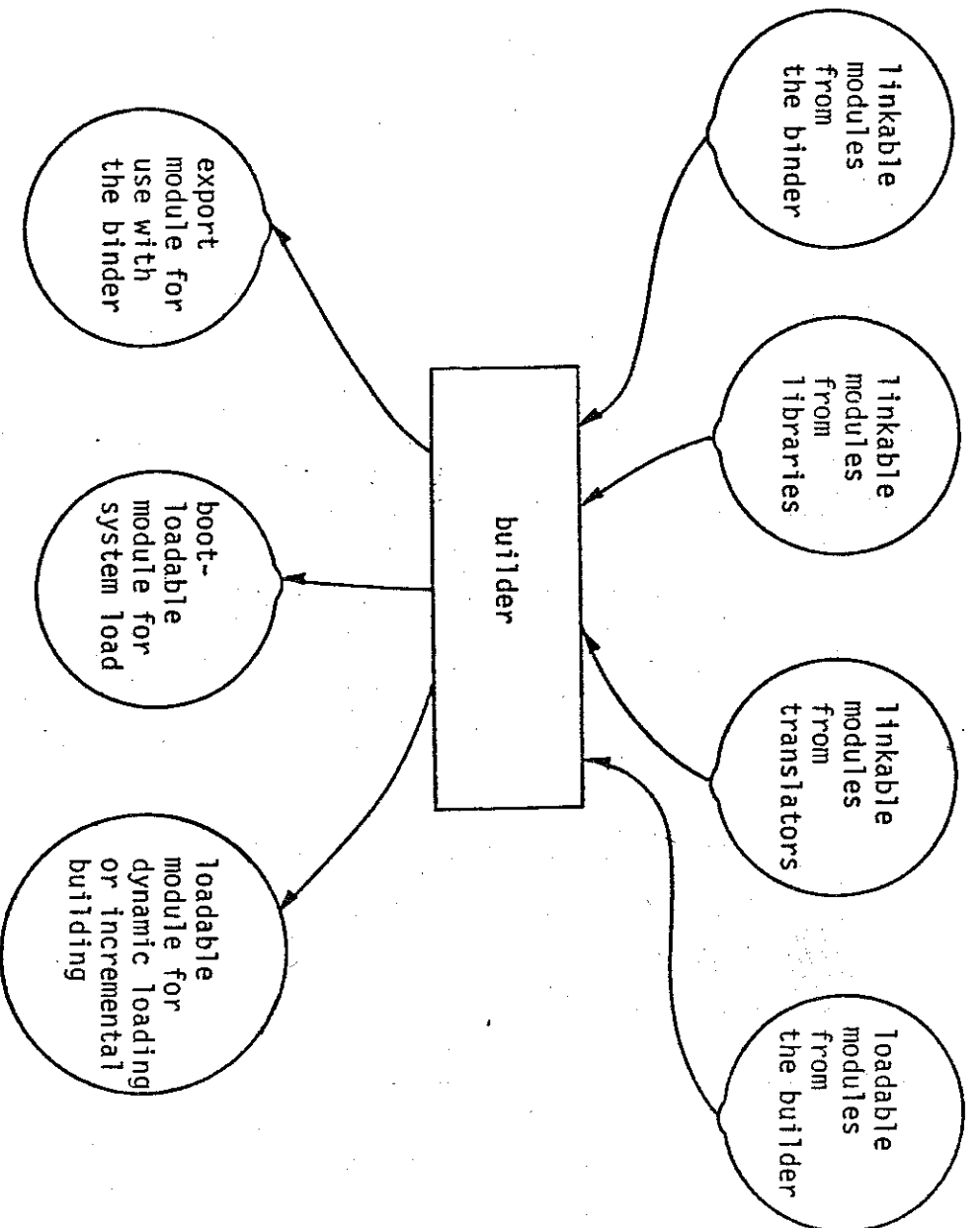


Figure 7.3: The builder - its module inputs and outputs

- resolves symbolic inter-module references and performs inter-module type checking (especially that type checking that cannot be performed by the binder, such as for gates);
- creates loadable modules from input linkable modules;
- creates export modules as requested, containing all exported symbols and conforming segments;
- optionally filters debug information from modules;
- selects required modules from specified libraries to resolve symbolic references;
- allows creation of segment aliases (e.g. to support a data segment alias for an LDT for OS use) and descriptor refinement (e.g. to create a read-only access rights version of a read-write access rights descriptor);
- allows incremental building (e.g. to build the OS kernel, executive, and data management systems in a hierarchical fashion);
- allows configuration of the address space into writeable, read-only, and reserved areas;
- produces a build file listing that contains a summary of the results of segment, gate, and public symbol processing, and an initial state of the constructed system; and
- detects and reports errors.

Of the numerous builder functions outlined above, the four chief ones are: segment descriptor binding, gate creation, descriptor table creation, and task state segment creation.

#### \* SEGMENT DESCRIPTOR BINDING \*

The builder does create some new segments (such as data segments for descriptor tables and for task blocks), but its primary function is to complete the binding of attributes to segment descriptors (that was begun by the translators and continued by the binder). The attributes bound by the builder are:

- limit                    - the input modules' segment definition records contain segment size limits, but these may be overridden in the builder under user control (the chief usage of this feature being for stacks); a segment may also be specified as load-time expandable (except in bootloadable modules);
- base address           - the builder invocation may specify an explicit segment base address, and the builder will warn of overlapping explicitly specified addresses;

- **privilege level** - typically translators set all segment privilege levels to 3 (least privileged) and multi-level system developers assign different privilege levels (on a module or segment basis) in the builder;
- **access rights** - an execute-only segment may be converted into an execute/read segment and/or a conforming segment; a read-only segment may be made writeable and/or expand/down (a stack); descriptors may be marked present or not.

#### \* GATE CREATION \*

Gates are created for installation in the GDT, IDT, or LDTs. They may be either call, interrupt, trap, or task gates, with the default being a call gate. The builder will bind the following attributes to the gate descriptor as requested:

- **name** - a symbolic name being either an existing public name (from an input module) or a new name;
- **type** - any of call, interrupt, trap, or task;
- **entry point** - a symbolic name for the entry point to be associated with the gate, in the default case being the public procedure name that is the same as the gate name, or otherwise being any existing procedure or task name;
- **word count** - necessary only for call gates; indicates the number of parameter words to be moved from the current stack to the target (more privileged) stack;
- **privilege level** - the gate privilege (visibility) level, between zero and three; must be at least as privileged as the entry point privilege level; and
- **present** - present or not, as for segment descriptors.

#### \* DESCRIPTOR TABLE CREATION \*

The builder may create one GDT, one IDT, and as many LDTs as there are tasks in the system. The GDT and IDT have exactly those names, while the LDT names are supplied to the builder by the user. Descriptors are installed in tables by listing their names, being either segment names from input modules, or previously defined gate, task, or LDT names. Table 7.1 shows the descriptor types that may be installed in the three forms of descriptor table--GDT, IDT, and LDT. By "LDT descriptor", we mean a builder-created data segment descriptor for an LDT.

Descriptor type	Table type		
	GDT	IDT	LDT
segment desc.	✓		✓
call gate	✓		✓
interrupt gate		✓	
trap gate		✓	
task gate	✓	✓	✓
LDT desc.	✓		

Table 7.1: Descriptor table entry types

For bootloadable output, a table base limit, privilege level, and present bit may be specified, as discussed for segment descriptors. By default, of course, the table limit will be the number of listed descriptors. For loadable output the table attributes just mentioned may be given only for LDTs (since a GDT and IDT are not generated).

#### \* TASK STATE SEGMENT CREATION \*

The builder may create task state segments statically (or the program may create them dynamically, during execution). If created by the builder, the specification includes the following:

- |                               |  |
|-------------------------------|--|
| TSS name                      | - user-supplied name for the task; generally the same as the task gate name;   |
| stacks                        | - at most one stack per privilege level may be specified by referring to existing writeable data segment descriptors (with the corresponding privilege levels); and  |
| task LDT and initial segments | - initial values for CS, DS, SS, and ES and for the task LDT may be specified by referring to an input module containing the initial specifications or explicitly; tasks may share an LDT, if desired, by installing the same LDT in different TSSs (but this places them in the same local address space, thus removing inter-task protection). |

#### \* THE LIBRARIAN \*

The librarian supports the creation, modification, and examination of object module libraries. These libraries contain linkable or loadable modules produced by translators or object utilities such as the binder or the builder. The librarian performs the following functions:

- creates object libraries and maintains library directories;
- adds, deletes, copies, and replaces object modules from or to object libraries;
- lists public symbols or module names within the library; and
- compacts libraries upon request.

The librarian-produced object libraries can be used as input sources (via external symbol resolution) by the binder or the builder.

REFERENCES

1. iAPX 86, 88 User's Manual, Intel Corporation, Order. No. 210201-001, July 1981.
2. Introduction to the iAPX 286, Intel Corporation, Order No. 210308-001, February 1982.
3. "Memory protection moves onto 16-bit microprocessor chip", P. Heller, R. Childs, J. Slager, Electronics, February 24, 1982.
4. "Integrating memory management into the CPU", G. Alexy, R. Childs, J. Crawford, Electronic Products, October 25, 1982.
5. iAPX 286 Operating System Writer's Guide, Intel Corporation, Order No. 121960-001, 1983.
6. "A VLSI architecture for software structure: The Intel 8086", A. Hartmann, S. Fehr, IEEE Micro, May 1981 (or Intel Corporation Order No. 210673-001, August 1982).
7. Principles of Compiler Design, A. Aho, J. Ullman, Addison Wesley, 1977.
8. "Code generation for expressions with common subexpressions", A. Aho, S. Johnson, J. Ullman, JACM, January 1977.
9. "Table-driven code generation", S. Graham, IEEE Computer, August 1980.
10. "Register allocation via usage counts", R. Frieburghouse, CACM, November 1974.
11. "A new method for compiler code generation ", S. Graham, R. Glanville, Conf. Record Fifth ACM Symposium Principles of Programming Languages, January 1978.
12. "Engineering a production code generator", J. Crawford, Proc. ACM SIGPLAN 1982 Symposium Compiler Construction, June 1982.
13. Pascal-86 User's Guide, Intel Corporation, Order No. 121539, 1981.
14. "Register allocation via coloring", G. Chaitin, M. Auslander, J. Cocke, A. Chandras, M. Hopkins, P. Monkstein, Computer Languages, Vol. 6, 1981.
15. "The generation of optimal code for arithmetic expressions", R. Sethi, J. Ullman, JACM, October 1970.
16. The Design of an Optimizing Compiler, W. Wulf, R. Johnsson, C. Wienstock, S. Hobbs, C. Geschke, American Elsevier, 1975.
17. Generating Machine Code For High-Level Programming Languages, T. Wilcox, Tech-Report 71-103, Dept. Computer Science, Cornell University, 1971.
18. "Code generation and storage allocation for machines with span-dependent instructions", E. Robertson, ACM TOPLAS, July 1979.

19. "Assembling code for machines with span-dependent instructions", T. Szymanski, CACM, April 1978.
20. "Interpretation techniques", P. Klint, Software Practice and Experience, Vol. II, 963-973, 1981.
21. The Concrete Representation of 80286 Object Modules, D. Albert, D. Nguyen, B. Prabhala, W. Van Riel, Intel Corporation Technical Specification, September 1982.
22. Procedural Interface to 80286 Object Module Formats, W. Van Riel, Intel Corporation Technical Specification, September 1982.
23. "Type checking", J. Stein, Intel Corporation Internal Memorandum, February 1978.
24. "8086 OMF TYPEDEFS For HLLs", A. Schlitt, Intel Corporation Internal Memorandum, July 1980.
25. iAPX 286 Utilities User's Guide, Intel Corporation, Order No. 121934, 1982.
26. iAPX 286 System Builder User's Guide, Intel Corporation, Order No. 121935, 1982.