

Back end table for the Intel 8080 micro-processor

Gerard Buskermolen

ABSTRACT

A back end is a part of the Amsterdam Compiler Kit (ACK). It translates EM, a family of intermediate languages, into the assembly language of some target machine, here the Intel 8080 and Intel 8085 microprocessors.

April 1985

Back end table for the Intel 8080 micro-processor

Gerard Buskermolen

INTRODUCTION

To simplify the task of producing portable (cross) compilers and interpreters, the Vrije Universiteit designed an integrated collection of programs, the Amsterdam Compiler Kit (ACK). It is based on the old UNCOL-idea ([4]) which attempts to solve the problem of making a compiler for each of N languages on M different machines without having to write $N * M$ programs.

The UNCOL approach is to write N "front ends", each of which translates one source language into a common intermediate language, UNCOL (UNiversal Computer Oriented Language), and M "back ends", each of which translates programs in UNCOL into a specific machine language. Under these conditions, only $N + M$ programs should be written to provide all N languages on all M machines, instead of $N * M$ programs.

The intermediate language for the Amsterdam Compiler Kit is the machine language for a simple stack machine called EM (Encoding Machine). So a back end for the Intel 8080 micro translates EM code into 8080 assembly language.

The back end is a single program that is driven by a machine dependent driving table. This driving table, or back end table, defines the mapping from EM code to the machine's assembly language.

1. THE 8080 MICRO PROCESSOR

This back end table can be used without modification for the Intel 8085 processor. Except for two additional instructions, the 8085 instruction set is identical and fully compatible with the 8080 instruction set. So everywhere in this document '8080' can be read as '8080 and 8085'.

1.1. Registers

The 8080 processor has an 8 bit accumulator, six general purpose 8-bit registers, a 16 bit program-counter and a 16 bit stackpointer. Assembler programs can refer the accumulator by A and the general purpose registers by B, C, D, E, H and L. (*) Several instructions address registers in groups of two, thus creating 16 bit registers:

Registers referenced:	Symbolic reference:
B and C	B
D and E	D
H and L	H

The first named register, contains the high order byte (H and L stand for High and Low).

The instruction determines how the processor interprets the reference. For example, ADD B is an 8 bit operation, adding the contents of register B to accumulator A. By contrast PUSH B is a 16 bit operation pushing B and C onto the stack.

There are no index registers.

* In this document 8080 registers and mnemonics are referenced by capitals, for the sake of clarity. Nevertheless the assembler expects small letters.

1.2. Flip-flops

The 8080 microprocessor provides five flip-flops used as condition flags (S, Z, P, C, AC) and one interrupt enable flip-flop IE.

The sign bit S is set (cleared) by certain instructions when the most significant bit of the result of an operation equals one (zero). The zero bit Z is set (cleared) by certain operations when the 8-bit result of an operation equals (does not equal) zero. The parity bit P is set (cleared) if the 8-bit result of an operation includes an even (odd) number of ones. C is the normal carry bit. AC is an auxiliary carry that indicates whether there has been a carry out of bit 3 of the accumulator. This auxiliary carry is used only by the DAA instruction, which adjusts the 8-bit value in the accumulator to form two 4-bit binary coded decimal digits. Needless to say this instruction is not used in the back-end.

The interrupt enable flip-flop IE is set and cleared under program control using the instructions EI (Enable Interrupt) and DI (Disable Interrupt). It is automatically cleared when the CPU is reset and when an interrupt occurs, disabling further interrupts until IE = 1 again.

1.3. Addressing modes

1.3.1. Implied addressing

The addressing mode of some instructions is implied by the instruction itself. For example, the RAL (rotate accumulator left) instruction deals only with the accumulator, and PCHL loads the program counter with the contents of register-pair HL.

1.3.2. Register addressing

With each instruction using register addressing, only one register is specified (except for the MOV instruction), although in many of them the accumulator is implied as second operand. Examples are CMP E, which compares register E with the accumulator, and DCR B, which decrements register B. A few instructions deal with 16 bit register-pairs: examples are DCX B, which decrements register-pair BC and the PUSH and POP instructions.

1.3.3. Register indirect addressing

Each instruction that may refer to an 8 bit register, may refer also to a memory location. In this case the letter M (for Memory) has to be used instead of a register. It indicates the memory location pointed to by H and L, so ADD M adds the contents of the memory location specified by H and L to the contents of the accumulator.

The register-pairs BC and DE can also be used for indirect addressing, but only to load or store the accumulator. For example, STAX B stores the contents of the accumulator into the memory location addressed by register-pair BC.

1.3.4. Immediate addressing

The immediate value can be an 8 bit value, as in ADI 10 which adds 10 to the accumulator, or a 16 bit value, as in LXI H,1000, which loads 1000 in the register-pair HL.

1.3.5. Direct addressing

Jump instructions include a 16 bit address as part of the instruction. The instruction SHLD 1234 stores the contents of register pair HL on memory locations 1234 and 1235. The high order byte is stored at the highest address.

2. THE 8080 BACK END TABLE

The back end table is designed as described in [5]. For an overall design of a back end table I refer to this document.

This section deals with problems encountered in writing the 8080 back-end table. Some remarks are made about particular parts of the table that might not seem clear at first sight.

2.1. Constant definitions

Word size (EM_WSIZE) and pointer size (EM_PSIZE) are both defined as two bytes. The hole between AB and LB (EM_BSIZE) is four bytes: only the return address and the local base are saved.

2.2. Registers and their properties

All properties have the default size of two bytes, because one-byte registers also cover two bytes when put on the real stack.

The next considerations led to the choice of register-pair BC as local base. Though saving the local base in memory would leave one more register-pair available as scratch register, it would slow down instructions as 'lol' and 'stl' too much. So a register-pair should be sacrificed as local base. Because a back-end without a free register-pair HL is completely broken-winged, the only reasonable choices are BC and DE. Though the choice between them might seem arbitrary at first sight, there is a difference between register-pairs BC and DE: the instruction XCHG exchanges the contents of register-pairs DE and HL. When DE and HL are both heavily used on the fake-stack, this instruction is very useful. Since it won't be useful too often to exchange HL with the local base and since an instruction exchanging BC and HL does not exist, BC is chosen as local base.

Many of the register properties are never mentioned in the PATTERNS part of the table. They are only needed to define the INSTRUCTIONS correctly.

The properties really used in the PATTERNS part are:

areg:	the accumulator only
reg:	any of the registers A, D, E, H or L. Of course the registers B and C which are used as local base don't possess this property. When there is a single register on the fake-stack, its value is always considered non-negative.
dereg:	register-pair DE only
hlreg:	register-pair HL only
hl_or_de:	register-pairs HL and DE both have this property
local	used only once (i.e. in the EM-instruction 'str 0')

The stackpointer SP and the processor status word PSW have to be defined explicitly because they are needed in some instructions (i.e. SP in LXI, DCX and INX and PSW in PUSH and POP). It doesn't matter that the processor status word is not just register A but includes the condition flags.

2.3. Tokens

The tokens 'm' and 'const1' are used in the INSTRUCTIONS- and MOVES parts only. They will never be on the fake-stack.

The token 'label' reflects addresses known at assembly time. It is used to take full profit of the instructions LHLD (Load HL Direct) and SHLD (Store HL Direct).

Compared with many other back-end tables, there are only a small number of different tokens (four). Reasons are the limited addressing modes of the 8080 microprocessor, no index registers etc. For example to translate the EM-instruction

lol 10

the next 8080 instructions are generated:

```
LXI H,10    /* load registers pair HL with value 10 */
DAD B      /* add local base (BC) to HL      */
MOV E,M    /* load E with byte pointed to by HL */
INX H     /* increment HL                    */
MOV D,M    /* load D with next byte                */
```

Of course, instead of emitting code immediately, it could be postponed by placing something like a {LOCAL,10} on the fake-stack, but some day the above mentioned code will have to be generated, so a LOCAL-token is hardly useful. See also the comment on the load instructions.

2.4. Sets

Only 'src1or2' is used in the PATTERNS.

2.5. Instructions

Each instruction indicates whether or not the condition flags are affected, but this information will never have any influence because there are no tests in the PATTERNS part of the table.

For each instruction a cost vector indicates the number of bytes the instruction occupies and the number of time periods it takes to execute the instruction. The length of a time period depends on the clock frequency and may range from 480 nanoseconds to 2 microseconds on a 8080 system and from 320 nanoseconds to 2 microseconds on a 8085 system.

In the TOKENS-part the cost of token 'm' is defined as (0,3). In fact it usually takes 3 extra time periods when this register indirect mode is used instead of register mode, but since the costs are not completely orthogonal this results in small deficiencies for the DCR, INR and MOV instructions. Although it is not particularly useful these deficiencies are corrected in the INSTRUCTIONS part, by treating the register indirect mode separately.

The costs of the conditional call and return instructions really depend on whether or not the call resp. return is actually made. However, this is not important to the behaviour of the back end.

Instructions not used in this table have been commented out. Of course many of them are used in the library routines.

2.6. Moves

This section is supposed to be straight-forward.

2.7. Tests

The TESTS section is only included to refrain **cgg** from complaining.

2.8. Stacking rules

When, for example, the token {const2,10} has to be stacked while no free register-pair is available, the next code is generated:

```
PUSH H
LXI H,10
XTHL
```

The last instruction exchanges the contents of HL with the value on top of the stack, giving HL its original value again.

2.9. Coercions

The coercion to unstack register A, is somewhat tricky, but unfortunately just popping PSW leaves the high-order byte in the accumulator.

The cheapest way to coerce HL to DE (or DE to HL) is by using the XCHG instruction, but it is not possible to explain **cgg** this instruction in fact exchanges the contents of these register-pairs. Before the coercion is carried out other appearances of DE and HL on the fake-stack will be moved to the real stack, because in the INSTRUCTION-part is told that XCHG destroys the contents of both DE and HL. The coercion transposing one register-pair to another one by emitting two MOV-instructions, will be used only if one of the register-pairs is the local base.

2.10. Patterns

As a general habit I have allocated (uses ...) all registers that should be free to generate the code, although it is not always necessary. For example in the code rule

```
pat loe
uses hlreg
gen lhld {label,$1}          yields hl
```

the 'uses'-clause could have been omitted because **cgg** knows that LHLD destroys register-pair HL.

Since there is only one register with property 'hlreg', there is no difference between 'uses hlreg' (allocate a register with property 'hlreg') and 'kills hlreg' (remove all registers with property 'hlreg' from the fake-stack). The same applies for the property 'dereg'.

Consequently 'kills' is rarely used in this back-end table.

2.10.1. Group 1: Load instructions

When a local variable must be squared, there will probably be EM-code like:

```
lol 10
lol 10
mli 2
```

When the code for the first 'lol 10' has been executed, DE contains the wanted value. To refrain **cgg** from emitting the code for 'lol 10' again, an extra pattern is included in the table for cases like this. The same applies for two consecutive 'loe'-s or 'lil'-s.

A bit tricky is 'lof'. It expects either DE or HL on the fake-stack, moves {const2,\$1} into the other one, and eventually adds them. The 'kills' part is necessary here because if DE was on the fake-stack, **cgg** doesn't see that the contents of DE is destroyed by the code (in fact 'kills dereg' would have been sufficient: because of the DAD instruction **cgg** knows that HL is destroyed).

By lookahead, **cgg** can make a clever choice between the first and second code rule of 'loi 4'. The same applies for several other instructions.

2.10.2. Group 2: Store instructions

A similar idea as with the two consecutive identical load instructions in Group 1, applies for a store instruction followed by a corresponding load instruction.

2.10.3. Groups 3 and 4: Signed and unsigned integer arithmetic

Since the 8080 instruction set doesn't provide multiply and divide instructions, special routines are made to accomplish these tasks.

Instead of providing four slightly differing routines for 16 bit signed or unsigned division, yielding the quotient or the remainder, the routines are merged. This saves space and assembly time when several variants are used in a particular program, at the cost of a little speed. When the routine is called, bit 7 of register A indicates whether the operands should be considered as signed or as unsigned integers, and bit 0 of register A indicates whether the quotient or the remainder has to be delivered.

The same applies for 32 bit division.

The routine doing the 16 bit unsigned multiplication could have been used for 16 bit signed multiplication too. Nevertheless a special 16 bit signed multiplication routine is provided, because this one will usually be much faster.

2.10.4. Group 5: Floating point arithmetic

Floating point is not implemented. Whenever an EM-instruction involving floating points is offered to the code-generator, it calls the corresponding library routine with the proper parameters. Each floating point library routine calls 'eunimpl', trapping with trap number 63. Some of the Pascal and C library routines output floating point EM-instructions, so code has to be generated for them. Of course this does not imply the code will ever be executed.

2.10.5. Group 12: Compare instructions

The code for 'cmu 2', with its 4 labels, is terrible. But it is the best I could find.

2.10.6. Group 9: Logical instructions

I have tried to merge both variants of the instructions 'and 2', 'ior 2' and 'xor 2', as in

```
pat and $1==2
with hl_or_de hl_or_de
uses reusing %1, reusing %2, hl_or_de, areg
gen mov a,%1.2
  ana %2.2
  mov %a.2,a
  mov a,%1.1
  ana %2.1
  mov %a.1,a          yields %a
```

but the current version of **egg** doesn't approve this. In any case **egg** chooses either DE or HL to store the result, using lookahead.

2.10.7. Group 14: Procedure call instructions

There is an 8 bytes function return area, called '.fra'. If only 2 bytes have to be returned, register-pair DE is used.

3. LIBRARY ROUTINES

Most of the library routines start with saving the return address and the local base, so that the parameters are on the top of the stack and the registers B and C are available as scratch registers. Since register-pair HL is needed to accomplish these tasks, and also to restore everything just before the routine returns, it is not possible to transfer data between the routines and the surrounding world through register H or L. Only registers A, D and E can be used for this.

When a routine returns 2 bytes, they are usually returned in registers-pair DE. When it returns more than 2 bytes they are pushed onto the stack.

It would have been possible to let the 32 bit arithmetic routines return 2 bytes in DE and the remaining 2 bytes on the stack (this often would have saved some space and execution time), but I don't consider that as well-structured programming.

4. TRAPS

Whenever a trap, for example trying to divide by zero, occurs in a program that originally was written in C or Pascal, a special trap handler is called. This trap handler wants to write an appropriate error message on the monitor. It tries to read the message from a file (e.g. etc/pc_rt_errors in the EM home directory for Pascal programs), but since the 8080 back-end doesn't know about files, we are in trouble. This problem is solved, as far as possible, by including the 'open'-monitor call in the mon-routine. It returns with file descriptor -1. The trap handler reacts by generating another trap, with the original trap

number. But this time, instead of calling the C- or Pascal trap handler again, the next message is printed on the monitor:

```
trap number <TN>
line <LN> of file <FN>
```

where <TN> is the trap number (decimal)
<LN> is the line number (decimal)
<FN> is the filename of the original program

Trap numbers are subdivided as follows:

1-27: EM-machine error, as described in [3]
63: an unimplemented EM-instruction is used
64-127: generated by compilers, runtime systems, etc.
128-252: generated by user programs

5. IMPLEMENTATION

It will not be possible to run the entire Amsterdam Compiler Kit on a 8080-based computer system. One has to write a program on another system, a system where the compiler kit runs on. This program may be a mixture of high-level languages, such as C or Pascal, EM and 8080 assembly code. The program should be compiled using the compiler kit, producing 8080 machine code. This code should come available to the 8080 machine for example by downloading or by storing it in ROM (Read Only Memory).

Depending on the characteristics of the particular 8080 based system, some adaptations have to be made:

- 1) In 'head_em': the base address, which is the address where the first 8080 instruction will be stored, and the initial value of the stackpointer are set to 0x1000 and 0x8000 respectively. Other systems require other values.
- 2) In 'head_em': before calling "__m_a_i_n", the environment pointer, argument vector and argument count will have to be pushed onto the stack. Since this back-end is tested on a system without any knowledge of these things, dummies are pushed now.
- 3) In 'tail_em': proper routines "putchar" and "getchar" should be provided. They should write resp. read a character on/from the monitor. Maybe some conversions will have to be made.
- 4) In 'head_em': an application program returns control to the monitor by jumping to address 0xFB52. This may have to be changed for different systems.
- 5) In 'tail_em': the current version of the 8080 back-end has very limited I/O capabilities, because it was tested on a system that had no knowledge of files. So the implementation of the EM-instruction 'mon' is very simple; it can only do the following things:

Monitor call 1:	exit
Monitor call 3:	read, always reads from the monitor. echos the read character. ignores file descriptor.
Monitor call 4:	write, always writes on the monitor. ignores file descriptor.
Monitor call 5:	open file, returns file descriptor -1. (compare chapter about TRAPS)
Monitor call 6:	close file, returns error code = 0.
Monitor call 54:	io-control, returns error code = 0.

If the system should do file-handling the routine ".mon" should be extended thoroughly.

6. INTEL 8080 VERSUS ZILOG Z80 AND INTEL 8086

6.1. Introduction

At about the same time I developed the back end for the Intel 8080 and Intel 8085, Frans van Haarlem did the same job for the Zilog z80 microprocessor. Since the z80 processor is an extension of the 8080, any machine code offered to a 8080 processor can be offered to a z80 too. The assembly languages are quite different however.

During the developments of the back ends we have used two micro-computers, both equipped with a z80 microprocessor. Of course the output of the 8080 back end is assembled by an 8080 assembler. This should assure I have never used any of the features that are potentially available in the z80 processor, but are not part of a true 8080 processor.

As a final job, I have investigated the differences between the 8080 and z80 processors and their influence on the back ends. I have tried to measure this influence by examining the length of the generated code. I have also involved the 8086 micro-processor in this measurements.

6.2. Differences between the 8080 and z80 processors

Except for some features that are less important concerning back ends, there are two points where the z80 improves upon the 8080:

First, the z80 has two additional index registers, IX and IY. They are used as in

```
LD B,(IX+10)
```

The offset, here 10, should fit in one byte.

Second, the z80 has several additional instructions. The most important ones are:

- 1) The 8080 can only load or store register-pair HL direct (using LHL or SHLD). The z80 can handle BC, DE and SP too.
- 2) Instructions are included to ease block movements.
- 3) There is a 16 bit subtract instruction.
- 4) While the 8080 can only rotate the accumulator, the z80 can rotate and shift each 8 bit register.
- 5) Special routines are included to jump to near locations, saving 1 byte.

6.3. Consequences for the 8080 and z80 back end

The most striking difference between the 8080 and z80 back ends is the choice of the local base. The writer of the z80 back end chose index register IY as local base, because this results in the cheapest coding of EM-instructions like 'lol' and 'stl'. The z80 instructions that load local 10, for example

```
LD E,(IY+10)
LD D,(IY+11)
```

occupy 6 bytes and take 38 time periods to execute. The five corresponding 8080 instructions loading a local occupy 7 bytes and take 41 time periods. Although the profit of the z80 might be not world-shocking, it should be noted that as a side effect it may save some pushing and popping since register pair HL is not used.

The choice of IY as local base has its drawbacks too. The root of the problem is that it is not possible to add IY to HL. For the EM-instruction

```
lal 20
```

the z80 back end generates code like

```
LD BC,20
PUSH IY
POP HL
ADD HL,BC
```

leaving the wanted address in HL.

This annoying push and pop instructions are also needed in some other instructions, for instance in 'lol' when the offset doesn't fit in one byte.

Beside the choice of the local base, I think there is no fundamental difference between the 8080 and z80 back ends, except of course that the z80 back end has register pair BC and, less important, index register IX available as scratch registers.

Most of the PATTERNS in the 8080 and z80 tables are more or less a direct translation of each other.

6.4. What did I do?

To get an idea of the quality of the code generated by the 8080, z80 and 8086 back ends I have gathered some C programs and some Pascal programs. Then I produced 8080, z80 and 8086 code for them. Investigating the assembler listing I found the lengths of the different parts of the generated code. I have checked two areas:

- 1) the entire text part
- 2) the text part without any library routine, so only the plain user program

I have to admit that neither one of them is really honest. When the entire text part is checked, the result is disturbed because not always the same library routines are loaded. And when only the user program itself is considered, the result is disturbed too. For example the 8086 has a multiply instruction, so the EM-instruction 'mli 2' is translated in the main program, but the 8080 and z80 call a library routine that is not counted. Also the 8080 uses library routines at some places where the z80 does not.

But nevertheless I think the measurements will give an idea about the code produced by the three back ends.

6.5. The results

The table below should be read as follows. For all programs I have computed the ratio of the code-lengths of the 8080, z80 and 8086. The averages of all Pascal/C programs are listed in the table, standardized to '100' for the 8080. So the listed '107' indicates that the lengths of the text parts of the z80 programs that originally were Pascal programs, averaged 7 percent larger than in the corresponding 8080 programs.

	8080	z80	8086
C, text part	100	103	65
Pascal, text part	100	107	55
C, user program	100	110	71
Pascal, user program	100	118	67

The most striking thing in this table is that the z80 back end appears to produce larger code than the 8080 back end. The reason is that the current z80 back end table is not very sophisticated yet. For instance it doesn't look for any EM-pattern longer than one. So the table shows that the preparations in the 8080 back end table to produce faster code (like recognizing special EM-patterns and permitting one byte registers on the fake-stack) was not just for fun, but really improved the generated code significantly.

The table shows that the 8080 table is relatively better when only the plain user program is considered

instead of the entire text part. This is not very surprising since the 8080 back end sometimes uses library routines where the z80 and especially the 8086 don't.

The difference between the 8080 and z80 on the one hand and the 8086 on the other is very big. But of course it was not equal game: the 8086 is a 16 bit processor that is much more advanced than the 8080 or z80 and the 8086 back end is known to produce very good code.

REFERENCES

- [1] 8080/8085 Assembly Language Programming Manual,
Intel Corporation (1977,1978)
- [2] Andrew S. Tanenbaum, Hans van Staveren, E.G. Keizer and Johan W. Stevenson,
A practical tool kit for making portable compilers,
Informatica report 74, Vrije Universiteit, Amsterdam, 1983.
- An overview on the Amsterdam Compiler Kit.
- [3] Tanenbaum, A.S., Stevenson, J.W., Keizer, E.G., and van Staveren, H.
Description of an experimental machine architecture for use with block structured languages,
Informatica report 81, Vrije Universiteit, Amsterdam, 1983.
- The defining document for EM.
- [4] Steel, T.B., Jr.
UNCOL: The myth and the Fact. in Ann. Rev. Auto. Prog.
Goodman, R. (ed.), vol. 2, (1960), p325-344.
- An introduction to the UNCOL idea by its originator.
- [5] van Staveren, Hans
The table driven code generator from the Amsterdam Compiler Kit (Second Revised Edition),
Vrije Universiteit, Amsterdam.
- The defining document for writing a back end table.
- [6] Voors, Jan
A back end for the Zilog z8000 micro,
Vrije Universiteit, Amsterdam.
- A document like this one, but for the z8000.