

CALM
Common Assembly Language for Microprocessors

Article of presentation

Table des matières

1 Introduction.....	3
1.1 What is CALM ?.....	3
1.2 Advantages of a common assembly language.....	3
1.3 Programming in assembler.....	3
1.4 The origins of CALM.....	4
2 Instruction.....	4
2.1 Composition of an instruction.....	4
2.1.1 Data transfer instructions.....	4
2.1.2 Arithmetic instructions.....	5
2.1.3 Logical instructions.....	5
2.1.4 Shift instructions.....	5
2.1.5 Test instructions.....	6
2.1.6 Program flow instructions.....	6
2.1.7 Special instructions.....	6
2.2 Operation code.....	6
2.3 Operands.....	6
2.4 Condition code.....	7
2.4.1 General:.....	7
2.4.2 After unsigned (logical) compare:.....	7
2.4.3 After arithmetic (2's complement) compare:.....	7
2.4.4 Special:.....	7
2.5 Data specifier.....	8
2.6 Address specifier.....	8
3 Notation of the addressing.....	9
3.1 Register and memory cells.....	9
3.2 Address spaces.....	10
3.3 Direct addressing.....	10
3.4 Immediate addressing.....	10
3.5 Indirect addressing.....	11
3.6 Combined addressing modes.....	11
3.7 Special addressing.....	11
4 Pseudo-instructions.....	12
4.1 Listing.....	12
4.2 Assembling.....	12
4.3 File insertion.....	12
4.4 Code-generation.....	12
4.5 Conditional instructions and macros.....	13
5 Use of CALM - not so easy.....	13
5.1 Documentation.....	13
5.2 CALM notation - many possible instruction codes.....	13
5.3 Assembler.....	13
5.4 Object format.....	13
6 Examples.....	14
6.1 APX86.....	14
6.2 M68000.....	14
6.3 NS32000.....	15
7 Conclusions.....	16
8 References.....	16

1 Introduction

Microprocessors have existed now for over ten years. But each microprocessor has its own assembly language, which differs from others in many ways. This fact is still strange because all microprocessors resemble functionally each other.

But a consistent assembly language could not be defined until some years ago, because new possibilities (operations, addressing modes) are offered with the development of new microprocessors. This evolution slowed down in the 16/32 bit era, because it has become more clear of what one expects from a modern microprocessor on operations and addressing possibilities. Hence the actual 16/32 bit microprocessors differ only slightly. One observes an increasing symmetry of the addressing modes, the data lengths and the data types in an instruction.

One of these universal assembly languages is presented here. Its name is CALM, which is the acronym for Common Assembly Language for Microprocessors.

1.1 What is CALM ?

CALM is not a new programming language, but another notation of the instructions. Many years of experience have shown, that the defined instructions and their notation is sufficient to describe about 95% of the instructions of a microprocessor.

The problem is rather simple: One finds a notation, which on the one side expresses precisely what the hardware does and, on the other side, is not too complicated. CALM fulfills these requirements in almost all cases. The exceptions are often unusual operations or addressing modes, which do not justify their own notation and therefore are represented with a simplified notation. An additional meta language is not necessary.

1.2 Advantages of a common assembly language

Today assembly programming is relatively complicated and time-consuming. Besides the increased requirement to program in assembler, artificial difficulties are added: Each manufacturer uses other terms and defines its own notation of the instructions. This different terminology makes the user unsure, prevents comparisons of processors and renders more difficult processor changes.

CALM defines a common syntax for instructions and pseudo-instructions. Thus CALM forms a common base for training and communication. For instance, an assembly program can also be understood by those which do not know the processor, for which the instructions have been written for.

In addition a common notation allows an objective comparison of micro-processors. Comparison tests are more efficient and easier to do, as the learning of the specific assembly language is no longer necessary.

Also processor changes are easier, as the notation does not change. Therefore the training cost can be substantially lowered. This is especially interesting with "upward-compatible" microprocessors.

Furthermore a common syntax simplifies the assembler. One can construct an assembler, which consists of a processor independent main part and a processor specific part. For the main part, the following functions are integrated: handling of symbols and pseudo-instructions, evaluation of expressions, file handling and general functions for the text analysis. The syntax and the semantics of the instructions and the rules for the object generation are all determined in the processor part.

1.3 Programming in assembler

The question is whether the assembly programming is still useful today. More and more powerful hardware and software push the assembly programming continuously back. But the assembly programming is, in many cases, irreplaceable. One may think of hardware interfaces, optimizations of any kind, single-chip microcomputers, microprogrammed units, etc. However these tasks are increasingly realized by specialists and offer a subroutine call on the higher programming level. But wouldn't it be an advantage, even in these cases, to be able to write these few programs in a common assembly language?

1.4 The origins of CALM

The existence of CALM is indirectly the product of the 8080. Many people found that the correspondent assembly language, defined by Intel, was too primitive. Many alternative solutions have shot up like mushrooms.

The first version of CALM was defined in 1974 at the Swiss Institute of Technology in Lausanne and has proven its efficiency by consistency and usefulness in the practical use. Other 8 bit processors have been expressed in CALM successfully.

With the 16/32 bit era, CALM has been extended and improved. CALM defines especially a consistent notation for the addressing modes and the data specifiers. The CALM version, presented in this article, has been defined in 1983 with DIN [1].

Under the worldwide efforts to define a common assembly language, the working group IEEE P694 should be mentioned. The group began its work in 1977 and published in 1984 the 18th version. Temporarily an intensive experience exchange took place. However P694 was too fixed to 4 and 8 bit microprocessors architectures and particularly underestimated the problems of an explicit notation for addressing modes.

In addition many programmers still believe that an assembly language should consist of names as short as possible and with every instruction only the absolute minimum should be noted.

2 Instruction

An instruction performs a precise action, which has been defined by the manufacturer of the microprocessor. The following text shows, how an instruction is decomposed in different parts and how these are named and which notation is used.

2.1 Composition of an instruction

An instruction is composed of an operation code and operands. In addition, depending on the complexity, a condition code, an address specifier and a data specifier may also be added.

This modular composition of an instruction and the independence of the meaning of every element, allow any extension. However this concept requires, that all details are necessary also in those instructions with limited possibilities.

CALM defines 49 main instructions (fig. 1). With these instructions, about 95% of the instructions of a microprocessor can be expressed. The rest are special instructions, where the defined operation code of the manufacturer is taken, but the CALM notation is used for the operands, the condition code, the address and data specifiers.

The instruction MOVE also fixes the order of the operands: source to destination (left to right). This order must also be kept for other instructions like SUB.

2.1.1 Data transfer instructions

MOVE	source,destination	move source operand to destination.
PUSH	source	push on stack (= MOVE source,{-SP}).
POP	destination	pop from stack (= MOVE {SP+},destination).
CONV	source,destination	perform a data type conversion.
CLR	destination	clear the destination.operand (= MOVE #0,destination).
SET	destination	set the destination.operand (= MOVE #-1,destination).
EX	operand1,operand2	exchange two locations.
SWAP	source,destination	the two halves of the operand are swapped.

2.1.2 Arithmetic instructions

ADD	source,sourcedestination source1,source2,destination	Add two operands.
ADDC	same as ADD	add two operands and the carry bit C.
SUB	same as ADD	subtract the first operand from the second.
SUBC	same as ADD	sub. the 1st operand and C from the 2nd operand
ACOC	same as ADD	add the 1's compl. of the 1st operand, 1 and C to the 2nd operand. Like SUBC but flags.
NEG	sourcedestination source,destination	negate the source operand (2's complement or subtract from zero).
NEGC	same as NEG	negate with carry, that means subtract the operand and the carry bit C from zero.
MUL	source,sourcedestination source1,source2,destination	multiply the 2 source operand. If the dest. operand is not expressly Mentioned with its size, there will be an ambiguity in the size of the destination. (double/single).
DIV	divisor,dividendquotient	divide the 2nd operand by the first divisor,dividend,quotient one. The destination is either the divisor,dividend,quotient,rest second or the third operand. Remainder may be an additional last op.
INC	sourcedestination	increment by one the operand.
DEC	sourcedestination	decrement by one the operand.
COMP	source1,source2	compare the second op. to the first one. Same operation as SUB, but no result is generated (only the flags are updated).
CHECK	lowerbound,upperbound,source	check a value against 2 bounds.

2.1.3 Logical instructions

AND	source,sourcedestination source1,source2,destination	perform a bitwise logical AND of the the source operands.
OR	same as AND	perform a bitwise logical OR.
XOR	same as AND	perform a bitwise logical XOR.
NOT	sourcedestination source,destination	perform a bitwise complement (inversion) of the source operand (1's complement).

2.1.4 Shift instructions

SR	sourcedestination amplitude,sourcedestination amplitude,source,destination	shift the source operand to the right. The MSB is replaced by zero, the amplitude,source,destination carry bit C is replaced by LSB.
ASR	same as SR	shift the source operand with sign to the right. MSB: unchanged. Carry bit C = LSB.
SL	same as SR	shift the source operand to the left. LSB = 0. Carry bit C = MSB.
ASL	same as SR	like SL but overflow bit V: has to be set, if the sign of the result is changed.
RR	same as SR	shift the source operand to the right. The MSB is replaced by the LSB. Carry bit = LSB.
RRC	same as SR	shift the source operand with C to the right. MSB = C. Carry bit C = LSB.
RL	same as SR	same as RR in the other direction.
RLC	same as SR	same as RRC in the other direction.

2.1.5 Test instructions

TEST	source source:bitaddress	test the sign and zero value of the source operand. With bitaddressing only the zero bit Z is modified (value of the bit).
TCLR	sourcedestination sourcedestination:bitaddress	test and clear the operand (bit(s)).
TSET	same as TCLR	test and set the operand (bit(s)).
TNOT	same as TCLR	test and complement the operand (bit(s)).

2.1.6 Program flow instructions

JUMP	jumpaddress	jump to the given address.
JUMP	,c, jumpaddress	jump to address, when condition true.
DJ	,c sourcedestination, jumpaddress	decrement 1st op. and jump to the address given by the 2nd op., when the condition c is satisfied.
SKIP	,c	skip next instr., when condition is true.
CALL	same as JUMP	subroutine call (save return add. on stack and jump to subroutine at given address).
CALL	,c same as JUMP	return from subroutine (jump to the address popped from the stack).
RET		return from subroutine (jump to the address popped from the stack).
RET	,c	special subroutine call.
TRAP	expression	special subroutine call.
TRAP	,c	
WAIT		wait for interrupt.
HALT		halt processor until an electrical reset signal restarts it.
RESET		reset peripherals.
NOP		no operation.
ION		interrupt on.
IOFF		interrupt off.

2.1.7 Special instructions

All special instructions like DAA, LINK, LEAVE, FFS.

2.2 Operation code

The operation code expresses the operation which is performed. The operation code is a name, which is derived from an english action verb. In some cases abbreviations and their combinations are used.

The defined CALM operation codes can also be found in the manufacturer assembly language of a microprocessor. The only differences concern those operation codes where the manufacturer anticipates an addressing mode in their names (for example: BRANCH, LEA, XLAT, ADDI). However some little differences in the orthography are often COMP instead of CMP, MOVE and MOV, etc.

2.3 Operands

An operand indicates who (register, memory location, etc.) participates in the operation and how this information is addressed. The operand gives this information with a defined notation. This notation must be sufficient to express all possible (and useful) addressing modes. Refer to chapter 3 for this notation.

This general notation is still unique. As proof, one can compare the notations of the manufacturers for addressing modes: a given microprocessor has only a limited number of addressing possibilities. Often this leads to a short notation, which simplifies the work of the assembler, but a concept for addressing modes is not presented.

A precise notation is also necessary, as it is no longer possible to include the kind of addressing in the operation code. Also the times, where the accumulator is the middle of a microprocessor, have definitively passed. More and more powerful addressing modes need a precise notation, because they may be combined with every operation.

2.4 Condition code

A condition code expresses a condition bit state or a combination of condition bit states.

Condition bits are changed by instructions. If the instructions on their side depend on condition bits, a condition code separated by a comma is added to the operation code. Condition codes are names which consist usually of 2 letters (fig. 2).

2.4.1 General:

EQ	Equal	NE	Not Equal
BS	Bit Set	BC	Bit Clear
CS	Carry Set	CC	Carry Clear
VS	oVerflow Set	VC	oVerflow Clear
MI	Minus	PL	PLus

2.4.2 After unsigned (logical) compare:

LO	Lower	LS	Lower or Same
HI	Higher	HS	Higher or Same

2.4.3 After arithmetic (2's complement) compare:

LT	Lower Than	LE	Lower or Equal
GT	Greater Than	GE	Greater or Equal

2.4.4 Special:

PE	Parity Even	PO	Parity Odd
NV	NeVer	AL	Always
NMO	Not Minus One		

Some condition codes are equivalent: EQ=BC, NE=BS, CS=LO, CC=HS.

Condition bits are usually stored in the flag register F. Additional condition bits (I, S, T) can exist and are often stored in a special status register S

letter/function	use
C	carry binary adder and binary shifter
H	half-carry 4 bit carry (only valid for 8 bit processors)
L	link only if never used as a carry bit
N	sign set: MSB set (negative number)
V	overflow set: overflow with arithmetic numbers
Z	zero set: result is equal to zero
I	interrupt set: interrupts are allowed
S	supervisor set: supervisor mode
T	trace set: trace active

The names of the condition codes are frequently identical to those of the manufacturers. The only way that they are appended to the operation code, distinguishes both notations: In the manufacturer's notation, the condition code is directly appended to the operation code name (often only one letter). The CALM notation is more consistent, as any operation code may be combined with any condition code. So both pieces of information are clearly separated

```
JUMP ,NE jumpaddress
RET, LO
CALL ,CS jumpaddress
DJ.16 ,NE CX,jumpaddress
SKIP ,LO DJ.16,NMO D0,jumpaddress
```

2.5 Data specifier

Data specifiers became only necessary with the 16/32 bit microprocessors. With these microprocessors one must specify the length and the type of the transferred data (fig. 5).

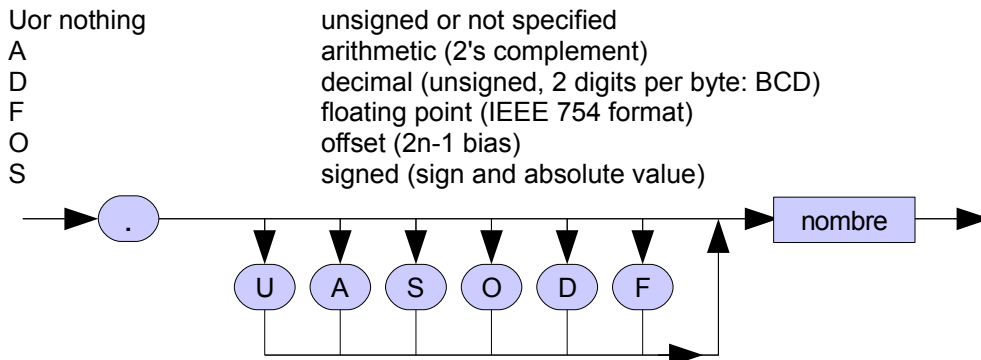


Fig. 5. Data specifiers

The data length indicates how many bits take part in an operation. Depending on the instruction these specifiers are given either with the operation code or with every operand individually. The data length is directly indicated in bits. A dot is used as a separation sign.

The data type says something about the processed data. The specifier consists of a letter, which is inserted between the dot and the data length. If there is no data specifier, it is assumed that the data doesn't need to be processed. Therefore a data type specifier is necessary when a certain unit, like BCD or FPU adder, is concerned (fig 6).

```

MOVE.32 R6,R1
ADD.D32 R5,R0
CONV R1.A16,R2.F64
    
```

Fig. 6. Examples with data specifiers

Also the manufacturer's notation of the instructions knows data specifiers. However only the data length is indicated independently.

The data length is indicated by a letter, which is either directly or separated by a dot appended to the operation code. The use of letters is still limited: On the one side there does not only exist data lengths of 8, 16 and 32 bits, and on the other side there will never be an agreement on their assignment.

For the different data types the manufacturers define different operation codes, for example IMUL for MUL.A16, ABCD for ADDX.D8 and NEGL for NEG.F64.

2.6 Address specifier

Address specifiers have become more and more necessary with the powerful addressing modes. Increasingly, an address is constituted by subaddresses, which do not have the full address length. Here, one must specify which length these subaddresses have and how they are extended to the full address length.

The address specifiers give all this information. The programmer can reconstruct on the paper what is happening in the microprocessor, or in other words, how the address is built. He can, therefore, clearly indicate to the assembler, which addressing mode is to be chosen.

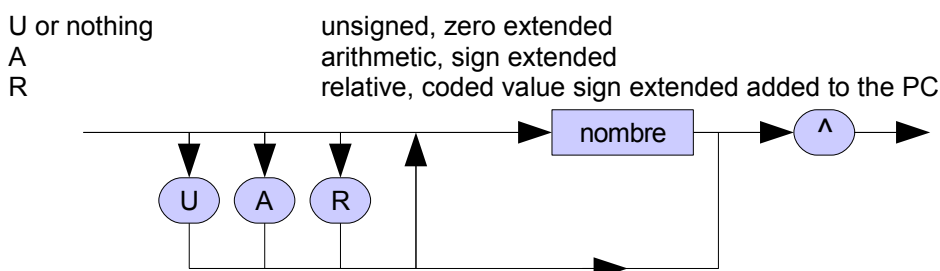


Fig. 7. Address specifiers

An address specifier includes an address length and an address type (fig. 7). The address length gives the number of bits which are delivered by a subaddress. The address length is also indicated in bits. The circumflex is used as a separation sign. An address specifier is placed before a subaddress in an operand (fig. 8).

```
JUMP 32^jumpaddress
CALL A16^jumpaddress
JUMP,NE R8^jumpaddress
MOVE.32 32^{A0}+A16^{D0}+A8^offset,D1
MOVE.16 32^{A5}+R8^label,D4
```

Fig. 8. Examples with address specifiers

The address type indicates how the address is interpreted. It distinguishes absolute and relative addressing. If an address does not deliver the full length, then the kind of extension to the full address length is indicated. With the absolute addressing this is possible with zero or sign extension. With the relative addressing, the program counter is added to the sign extended offset value.

Address specifiers exist also in the assembly languages of the manufacturers. However they can often not be recognized because of their nonuniform syntax. Different address lengths are specified by pre- or postfixes (for example: LBRA, BRA.S, disp(A0,D3.L), BNElabel:W). Address types are distinguished by different operation codes or special signs (for example: BSR, label(PC), @label). Fortunately, only for jump instructions, many manufacturers are still using two different operation codes (BRANCH and JUMP) to distinguish relative and absolute addressing. However this complicates and obstructs ultimately any extensions.

3 Notation of the addressing

The preceding text clarifies, that consistent rules must be formulated to unambiguously specify the type of addressing. For this, the programmer uses a simplified programming model, that generally contains the register structure and the general architecture of the memory cells.

3.1 Register and memory cells

Register and memory locations consist of a number of bits. In micro-processors, multiples of 8 bits (= 1 byte) are usual. Hence register lengths of 8, 16 and 32 bits are constructed.

Consecutive memory locations are numbered and that number is called an address. If a 16 bit word is placed in two consecutive locations, two different byte orders exist (fig. 9).

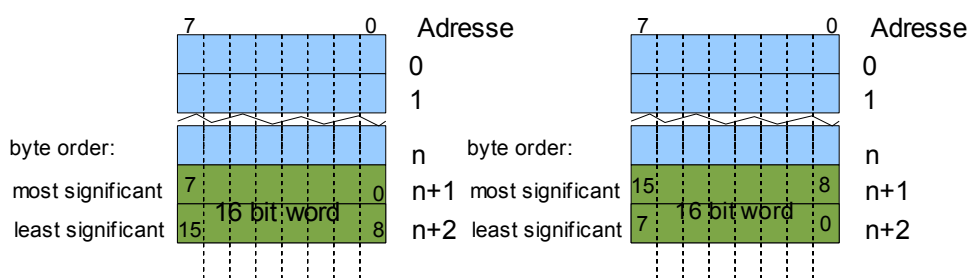


Fig. 9. Byte numbering

Furthermore, if a 16 bit word is bit addressed, the byte order, with the most significant byte first, is a disadvantage, because the bits are not numbered in the same way. In addition, the bit numbering in registers, and consecutive memory locations with the most significant byte first, is different.

3.2 Address spaces

Register and memory locations can be seen as part of a register and a memory address space. Often, the register address space contains only a few memory locations, but is directly placed in the microprocessor and has therefore a small access time. As there are only a few locations, and because of their technical privileged position, they are named registers. In addition each register has its own reserved name. If possible, CALM uses the register names defined by the manufacturer. Furthermore, only the listed symbols in fig. 10 are reserved in CALM.

PC	Program Counter at execution time
SP	Stack Pointer
F	arithmetic Flag register
S	Status register
APC	value of the Assembler PC at the beginning of the data instruction where APC appears
TRUE -1	(all bits ones)

Fig. 10. Reserved and predefined symbols

On the other hand the memory address space is very big and placed externally. Memory locations are numbered and can obtain user defined symbols.

On some microprocessors, additional address spaces exist: Input/Output and data. In order to distinguish them from the memory address space, a dollar sign is placed before the data address expression with the I/O address space and a percent sign with the address space (fig. 11).

MOVE B,A	register -> register
MOVE ADMEMORY,A	memory address space -> register
MOVE \$ADINOUT,A	input/output address space -> register
MOVE %ADDATA	A data address space -> register

Fig. 11. Examples: address spaces

3.3 Direct addressing

As mentioned before, registers and memory locations consist of a number of bits. These bit states represent a binary word, which defines the content of the register or the memory location.

It is common in assembly languages to indicate only the name of a content, instead of the content itself. One says "Increment D", instead of "Increment the content of the register D".

This implicit reference is always valid in CALM and is valid for all four mentioned address spaces. The reserved name of the register or the address of the memory location is directly given. This is named direct addressing (fig. 12).

ADD B,A	content(B) + content(A) -> content(A)
ADD ADMEMORY,A	content(ADMEMORY) + content(A) -> content(A)
ADD 16'1234,A	content(16'1234) + content(A) -> content(A)

The 2nd example is identical with the 3rd, when ADMEMORY = 16'1234.

Fig. 12. Examples: direct addressing

3.4 Immediate addressing

A number sign before a constant, a symbol or a complex expression cancels the implicit reference and gives the immediate built address (and not their content). This is named immediate addressing (fig. 13). The number sign corresponds to the pointer sign (^, @) in high-level languages.

ADD #16'1234,A	16'1234 + content(A) -> content(A)
ADD #ADMEMORY+1,A	ADMEMORY + 1 + content(A) -> content(A)
JUMP ADMEMORY	ADMEMORY -> content(PC)
MOVE #ADMEMORY,PC	ADMEMORY -> content(PC)

Fig. 13. Examples: immediate addressing

3.5 Indirect addressing

If the content of any address is again used as an address, braces enclose the corresponding address expression. This is named indirect addressing (fig. 14).

```
ADD {HL},A          content(content(HL)) + content(A) -> content(A)
MOVE #16'1234,{HL}  16'1234 -> content(content(HL))
MOVE #16'1234,{ADMEMORY}  16'1234 -> content(content(ADMEMORY))
```

Fig. 14. Examples: indirect addressing

There is no functional difference, if a register or a memory location is used for the indirect addressing. However many processors allow only an indirect addressing with registers.

3.6 Combined addressing modes

With the direct, immediate and indirect addressing we have already the base elements to build any complex address. The possible operations between the individual address terms are the addition, the subtraction and the multiplication (fig. 15).

```
MOVE #16'FE,{A0}+8    254 -> content(content(A0)+8)
MOVE #10'98,{A0}-8    98 -> content(content(A0)-8)
MOVE #8'177,{A0}*8    127 -> content(content(A0)*8)
MOVE #2'1011,{A0}*    11 -> content(content(A0)* datalength)
MOVE R0,{{SB}+5}+3    cont.(R0) -> content(content(content(SB)+5)+3)
MOVE {A0}+{D0}+10,A1 cont.(content(A0)+content(D0)+10) -> cont.(A1)£
MOVE #{A0}+{D0}+10,A1 content(A0)+content(D0)+10 -> content(A1)
```

Fig. 15. Examples: combined addressing modes

Note that the indirect addressing is only an explicit notation of the direct addressing, but may be applied as many times as desired. The implicit reference (= content of a built address) is only applied to the whole address expression. If a number sign is before the whole address expression, this implicit reference is suspended.

3.7 Special addressing

The relative addressing is in reality only a shorter notation of a combined addressing mode as shown in fig. 16.

notation	description	designation
MOVE R^ADDRESS,D0	MOVE {PC}+OFFSET,D0	relative addressing
MOVE {A0+},D0	MOVE {A0},D0	automodification (incrementation)
MOVE D0,{-A0}	ADD #data length,A0 SUB #data length,A0	automodification (decrementation)
CLR D0:#1	MOVE D0,{A0}	bit addressing
PUSH R1..R3 R5	bit 1 in D0 is cleared	register list
	PUSH R1	
	PUSH R2	
	PUSH R3	
	PUSH R5	
MOVE R0,R1:#1..#4	transfers bits 0 to 3 from bit list R0 to R1 (bits 1 to 4)	

Fig. 16. Examples: special addressing modes

Some additional addressing modes have attained a certain importance and therefore became a unique notation.

In the bit addressing, the first expression gives the byte address. The expression after the colon is the bit address. The bit address zero corresponds to the bit 0 in the addressed byte. If the byte address points to the memory address space, the bit address may exceed the byte (positive or negative).

4 Pseudo-instructions

Pseudo-instructions are commands to the assembler which control the code generation, the conditional assembly and listing, macros, etc. Every pseudo-instruction begins with a dot (fig. 17). Only with the pseudo-instructions `.ASCII`, `.ASCIZ`, `.BLK`, `.FILL`, `.nand` and `.STRING` are preceding labels allowed. The byte order with the pseudo-instructions `.16`, `.32`, etc. is determined by the processor (`.PROC`).

4.1 Listing

<code>.TITLE</code>	character string start a new page with the specified title.
<code>.CHAP</code>	character string add the spec. text in the subtitle header.
<code>.END</code>	terminate the assembler program.
<code>.TEXT</code>	until an <code>.ENDTEXT</code> is encountered, the foll. lines are ign. by the ass. (listing copy).
<code>.ENDTEXT</code>	terminate <code>.TEXT</code> .
<code>.LIST expression</code>	list the subsequent instr. only if the expression is true. <code>.LIST</code> may be nested.
<code>.ENDLIST</code>	end of a conditional list segment.
<code>.LAYOUT</code>	define the layout param. for the listing.

4.2 Assembling

<code>.BASE expression</code>	define the new default base.
<code>.START expression</code>	define the starting address.
<code>.EXPORT symbol1, symbol2, ...</code>	define the exported symbols.
<code>.IMPORT symbol1, symbol2, ...</code>	define the imported symbols.
Program counter	
<code>.APC expression</code>	select one of the ass. program counters.
<code>.LOC expression</code>	current ass. program counter (APC) = value.
<code>.ALIGN expression</code>	align APC to the next multiple of value.
<code>.EVEN</code>	align the APC value to the next even value.
<code>.ODD</code>	align the APC value to the next odd value.
<code>.BLK.8 expression (number)</code>	add to the value of the APC the product
<code>.BLK.8.16.32 expression</code>	of the data size times the nb. of items.

4.3 File insertion

<code>.INS filename</code>	insert the mentioned source file.
<code>.REF filename</code>	insert the mentioned symbol table.
<code>.PROC filename</code>	insert the mentioned processor description.

4.4 Code-generation

<code>.n expression, expression, ...</code>	insert the given values in the
<code>.8.32 exp8, exp32, exp8, ...</code>	object.
<code>.FILL.n expression (length), expression</code>	fill length with value.
<code>.ASCII "ascii text"</code>	insert the ASCII codes in the object.
<code>.ASCIZ "ascii text"</code>	as <code>.ASCII</code> , with ASCII code null at the end.
<code>.STRING "ascii text"</code>	as <code>.ASCII</code> , with the len. (8 bits) at begin.

4.5 Conditional instructions and macros

.IF expression	the sub. instr. up to the corresp. .ELSE or .ENDIF are ass. if exp.true. May be nested.
.ELSE	the sub. instr. up to the corresp. .ENDIF are ass., if the exp. with .IF was false.
.ENDIF	end an .IF or .ELSE section.
.MACRO name,parameter1,...	begin of a macro definition with the macro name and the optional parameter list.
.ENDMACRO	end of a macro definition.
.LOCALMACRO symbol1,...	list of local symbols in a macro.

Fig. 17. CALM pseudo-instructions

5 Use of CALM - not so easy

A certain amount of time is needed for any programming language before it is accepted. But in opposition to high-level languages, a common assembly language depends naturally on the microprocessor which the language tries to describe.

5.1 Documentation

The CALM notation of the instructions of a processor is documented in so-called reference cards. In these reference cards, all the instructions of a processor are listed in a compact form.

But additional information like instruction codes, execution times, particularities, etc. of a microprocessor can only be found in the documentation of the manufacturer. So the user is forced to know temporarily both notations. Therefore the CALM reference cards also give the operation code names of the manufacturer.

5.2 CALM notation - many possible instruction codes

There are many instructions which are identical on modern 16/32 bit microprocessors. The CALM notation shows this clearly. The question is now: which instruction code should the assembler generate?

Normally the assembler chooses the most compact and fastest instruction code. The distinguishing features are given by the operands. The resulting instruction code depends on the size and the type of the operand and on the additional address and data specifiers.

5.3 Assembler

Despite the advantages of CALM, programming in CALM obviously has no sense if the correspondent utilities (assembler, linker, etc.) do not exist. Actually, CALM assemblers generating machine code (without linker) are available from the author for all current microprocessors and for the operating systems PC/MS-DOS 2.11, Atari ST, and Smaky.

5.4 Object format

The machine code generation still remains very complicated because each processor has been defined a special object format. This diversity makes it quite impossible to generalize the assembler. Nevertheless a common object format [2] has been defined.

6 Examples

The following examples compare instructions in the CALM notation to the manufacturer's notation. These are not complete programs, but a selection of twenty instructions from the microprocessors iAPX86, M68000 and NS32000.

Additional details of the assembly notation depends strongly on the used assembler. Here the information from the manufacturer assembler has been used.

6.1 APX86

The iAPX86 from Intel is one of the most complex microprocessors because of its segmentation. The assembler ASM-86 needs a lot of information about these segment registers (ASSUME, NEAR, declaration of the variables, etc.). Therefore the following examples are not complete because this information is not given here.

Because of the exceptional architecture, a common assembly language can not be simple for this processor (fig. 18). CALM indicates the used segment register with every instruction. In addition the jump range (in or outside of the current segment) is determined by a data specifier appended to the jump instruction. Also a simplified notation is introduced for the automatic shifting of the segment register ([CS] for {CS}*16).

CALM		Intel	
MOVE .16	#16'1000, BX	MOV	BX, 1000H
MOVE .16	AX, \${DX}	OUT	DX, AX
MOVE .16	# [ES]+NEXT, DX	LEA	DX, ES:NEXT
MOVE .8	DL, BH	MOV	BH, DL
MOVE .8	[DS]+DATA, AL	MOV	AL, DATA
MOVE .16	[CS]+{SI}, AX	MOV	AX, CS:[SI]
MOVE .8	AH, F	SAHF	
PUSH .16	SF	PUSHF	
MOVE .8	[DS]+{BX}+{AL}, AL	XLAT	CONV_TAB
MOVE .32	[DS]+TARGET, DSBX	LDS	BX, TARGET
CONV.A8 .16	AX	CBW	
INC .8	AL	INC	AL
DIV.A16	CX, DX AX, AX, DX	IDIV	CX
OR .16	#ERROR, [DS]+STATUS	OR	STATUS, ERROR
RL .8	AL	ROL	AL, 1
TEST .16	#MASK, AX	TEST	AX, MASK
JUMP, LO	R^LOW	JB	LOW
CALL .32	20^ROUTINE	JSR	FAR ROUTINE
DJ .16, NE	CX, ADDRESS	LOOP	ADDRESS
TRAP .32, VS	INTO		

Fig. 18. Examples: iAPX86

6.2 M68000

Motorola's notation of the instructions for the M68000 differs slightly from CALM (fig. 19).

This processor shows that upward-compatibility is still valid for machine code (however only partly), but not for the assembly language. The additional addressing modes of the M68020 forced Motorola to change the notation of some addressing modes.

CALM		Motorola	
MOVE .32	#16'1000, D1	MOVE .L	#\$1000, D1
MOVE .32	#A8^WERT, D0	MOVEQ	#WERT, D0
MOVE .32	# {A6}+100, A3	LEA	100 (A6), A3
MOVE .8	D4, D6	MOVE .B	D4, D6
MOVE .8	{A4}+4, D2	MOVE .B	4 (A4), D2
MOVE .16	{A0}+A16^{D4}+2, {A2}+32^{A3}+4	MOVE	2 (A0, D4), 4 (A2, A3.L)
MOVE .16	D0, F	MOVE	D0, CCR
PUSH .16	SF	MOVE	SR, -(A7)
PUSH .32	#TABLE	PEA	TABLE

PUSH.32	D0..D3 D5	MOVEM.L	D0-D3/D5, -(A7)
CONV.A8	16 D4	EXT.W	D4
INC.8	D1	ADDQ.B	#1, D1
DIV.A16	#10, D6	DIVS	#10, D6
OR.16	#MASK, {A4}+A16^{D4}-4	OR	#MASK, -4 (A4, D4.W)
RL.32	#8, D4	ROL.L	#8, D4
TCLR.8	{A0+}:D4	BCLR	D4, (A0)+
JUMP, LO	R^LOW	BLO	LOW
CALL	U^ROUTINE	JSR	ROUTINE
DJ.16, NMO	D0, ADDRESS	DBRA	D0, ADDRESS
TRAP, VS		TRAPV	

Fig. 19. Exemples: M68000

6.3 NS32000

In reality this is a family, called "NS32000 series" by National Semiconductor, which actually consists of three 100% identical members (NS32032, NS32016, NS32008). The only difference is the external bus wide.

The NS32000 series has a complete symmetry on the addressing possibilities for all instructions (fig. 20). The relative addressing is allowed anywhere and is coded by the assembler automatically.

CALM		NS	
MOVE.32	#16'1000, R1	MOVD	H'1000, R1
MOVE.32	#A4^WERT, R0	MOVQD	WERT, R0
MOVE.32	#{R6}+100, {SB}-20	ADDR	100 (R6), -20 (SB)
MOVE.8	R4, R6	MOVB	R4, R6
MOVE.8	{SB}+4, R2	MOVB	4 (SB), R2
MOVE.16	{{FP}+2}+4, {SB}+{R0}*1+2	MOVW	4 (2 (FP)), 2 (SB) [R0:B]
MOVE.8	R0, F	LPRB	R0, UPSR
PUSH.16	SF	SPRW	PSR, TOS
PUSH.32	#R^TABLE	ADDR	TABLE, TOS
PUSH.32	R0..R3 R5	SAVE	[R0, R1, R2, R3, R5]
CONV	R4.8, R2.32	MOVZBD	R4, R2
INC.8	R1	ADDQB	1, R1
DIV.A32	#10, R6	QUOD	10, R6
OR.16	#MASK, {{SB}-2}+{R5}*8+4	ORW	MASK, 4 (-2 (SB)) [R5:Q]
RL.32	#-10, R4	ROTD	-10, R4
TCLR	{R0}+20:{{SB}+10}+4.A32	CBITD	4 (10 (SB)), 20 (R0)
JUMP, LO	R^LOW	BLO	LOW
CALL	U^ROUTINE	JSR	@ROUTINE
AJ.32, NE	#2, {R0}+4, ADDRESS	ACBD	2, 4 (R0), ADDRESS
TRAP, VS		FLAG	

Fig. 20. Exemples: NS32000

7 Conclusions

CALM is a common assembly language, which is adapted for almost all microprocessors, but also for miniprocessors, mainframes and microprogrammed units. The CALM notation is oriented towards the future, as the notation can be expanded purposefully and functionally.

The clear separation of the individual parts of an instruction allows a specific adaptation to any processor. In addition, one understands an instruction in the CALM notation better than a long description.

Twenty-four processors have been checked and it was proven that the CALM notation is sufficient. CALM reference cards have been established for the following processors: Z80, 65x02, 680x, 6805, 6809, 8048, 8051, 808x, iAPXx86, NS32000, 680xx, etc.

This short article tried to show in some terms the main characteristics of CALM. The assembly programming becomes not simpler in the CALM notation, but it is more comprehensible for the user, because CALM is constructed by logical rules.

8 References

- [1] DIN 66283, Allgem. Assembler-Sprache für Mikroproz. CALM Norme nat. all., Beuth Verlag GmbH, CP 1145, D-1000 Berlin 30
- [2] "The Microprocessor Universal Format for Object Modules", Prop. Stand.: IEEE P695 Working Group. IEEE Micro, 8.1983, p. 48-66.
- [3] Nicoud, J.D. and Fäh, P. Common Assembly Language for Micro-processors CALM, Document interne. LAMI-EPFL, INF-Ecublens, CH-1015 Lausanne, Dec. 1986. English version of DIN 66283.
- [4] Nicoud, J.D. and Fäh, P. Explanations and Comments, Related to the Common Assembly Language for Microprocessors. LAMI-EPFL.
- [5] Zeltwanger, H. "Genormte Assemblersprache für Ps", ELEKTRONIK, 35ème année (1986), no. 8, p. 66-71.
- [6] Nicoud, J.D. Calculatrices, Volume XIV du Traité d'Electricité, Lausanne: Presses polytechniques romandes, 1983.
- [7] Nicoud, J.D. and Wagner, F. Major Microprocessors, A unified approach using CALM. North Holland, 1987.
- [8] Strohmeier, A. Le matériel informatique, concepts et principes Lausanne: Presses polytechniques romandes, 1986.
- [9] Fäh, P. "Die (Un-)Logik von Assemblersprachen", Elektroniker, 26ème année (1987), no. 5, p. 97-100.