The Design and Implementation of a Cross-Assembler for a Virtual Machine

Michel de Champlain
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada

November 9, 2020

Table of Contents

Chapter 1

Overview of the Cm Assembly Language

In this chapter, we present an overview of the **Cm** Assembly Language of a virtual machine (VM) to support programming languages intended for small footprint embedded systems. The instruction set of the VM is tailored to support a subset of the C programming language, called Cm[1], intended for a restrictive microcontroller environment such as 8-bit microcontrollers.

Upon completion of this chapter, you will be able to:

- Understand the format of a Cm assembly program.
- Define an assembly unit and line statements.
- Identify labels, comments, instructions, and tokens.
- Know how to assemble a Cm assembly program.

[1] **Cm** as a subset of the **C** programming language for **m**icrocontrollers. In music, Cm or C- means C minor. A C minor chord is a chord that has C as a root :)

At the lowest level of computation, the central processing unit of a computer directly executes instructions that are represented in binary (or hexadecimal) code. These instructions constitute a machine language, and are far removed from the natural expressiveness of higher level languages such as Java or C#. Therefore, to program in a machine language is tedious, time-consuming and prone to error.

In order to bridge the "language gap" between the human programmer and the computer itself, the labels, instructions, operands and offsets of a machine language are represented symbolically. An assembly language therefore is a symbolic, one-to-one representation of a corresponding machine language. For example, machine instructions for increment, decrement and test-for-equal may correspond to `0x11`, `0x12`, and `0x1A` respectively in hexadecimal notation. Using symbolic names (or mnemonics), these same instructions may be represented in an assembly language as `inc`, `dec`, and `teq`. Although an assembly language corresponds directly to the underlying machine code, it does provide one level of abstraction that vastly improves readability and usability. Consider the following segment expressed in the Cm assembly language.

```
        calls.i16   Fct

        ...

Fct     ldc.i3      3
        ldc.i3      1
        add
        ret
```

In this example, a call is made to the function `Fct` using its symbolic name. When the assembly code is translated into machine instructions using an assembler, the symbolic name `Fct` is replaced by the actual address of where the instruction `ldc.3 3` is loaded into memory as shown below. Furthermore, the instructions `ldc.i3 3`, `ldc.i3 1`, `add` and `ret` are replaced by `0x93`, `0x91`, `0x13` and `0x04` respectively.

```
Addr   Machine Code

0002   F4 0107

...    ...

0109   93
010A   91
010B   13
010C   04
```

Although the function name `Fct` may be substituted by an actual offset or memory address (such as `0x0109`), the practice of using a hardcoded address instead of a symbolic name is very prone to error. For that reason, numerical addresses or offsets for function calls are not permitted in the Cm assembly language.

This chapter offers a general overview of the Cm assembly language (**Cm ASM**).

1.1 Format of a Cm Assembly Program

The Cm assembly language is used to express the compiled version of a Cm program. But rather than represent the machine language of the underlying hardware, each line of the Cm assembly code directly corresponds to one instruction of the virtual machine. This instruction is then executed by the **Cm VM**.

Each line statement of a Cm assembly program (or assembly unit) is composed of three optional fields: a label, an instruction and a comment. The label identifier (when present) must begin in the first column and represents a destination address for branching and looping. The instruction field breaks down into two parts. The first part is a mnemonic that represents the operation code (or opcode) of an instruction. The second part is an optional operand. Only one operand is permitted in the Cm assembly language. Finally, the third field is a comment that begins with a semi-colon (`";"`) and documents the assembly code.

The following short program which calculates the sum of the values 0 to 9 illustrates the layout of a simple Cm assembly program. Note that the line numbers to the left are **not** part of the assembly code and have been inserted for easy reference.

```
1   ; Sample program
2
3            ldc.i3   0
4            dup
5            stv.u3   0     ; n = 0
6            stv.u3   1     ; sum = 0
7  Loop      ldv.u3   0     ; push n
8            ldc.i8   10    ; push 10
9            tlt            ; if n < 10 then Continue
10           brf.i5   Done  ; else Done
11 Continue  ldv.u3   1     ; push sum
12           ldv.u3   0     ; push n
13           add            ; add n to sum
14           stv.u3   1     ; store sum
15           incv.u8  0     ; n++
16           br.i5    Loop
17 Done
18           halt
```

The following sections provide a more complete syntax summary of the Cm assembly language. Details on the size of opcodes and instructions as well as the addresses of labels and operands are deferred to the next chapter.

1.2 Assembly Unit

An `AssemblyUnit` consists of a sequence of zero or more `LineStatement`s followed by an End-Of-File (`EOF`).

EBNF

```
AssemblyUnit  = { LineStatement } EOF .
```

Together, lines 1 through 18 above represent an assembly unit.

1.3 Line Statements

Each `LineStatement` is composed of three optional parts (a label, an instruction/directive and a comment) followed by an end-of-line (`EOL`).

EBNF

```
LineStatement = [ Label ] [ Instruction | Directive ] [ Comment ] EOL .
```

Notice that a line statement falls into one of two categories: instruction or directive. Whereas an instruction is intended for execution by a processor or virtual machine, a directive represents a pseudo-instruction for the assembler itself. For example, `.cstring` is a directive that instructs the assembler to allocate a C string which contains 8-bit ASCII characters ended by a null character.

In the code segment above, there are 18 line statements. Lines 1, 3 and 17 respectively consist of only the comment, instruction and label. Line 2, on the other hand, has omitted all three parts.

1.3.1 Labels

A `Label` simply consists of a name (identifier) and always starts at the first column of a line statement. For example, `Loop`, `Continue` and `Done` on lines 7, 11 and 17 represent labels.

1.3.2 Instructions

An `Instruction` is composed of an opcode `Mnemonic` and an optional `Operand`. All opcodes are presented in Tables 1.2 and 1.3.

EBNF

```
Instruction = Mnemonic [ Operand ] .
```

On lines 9, 13 and 18, the instruction consists of the mnemonic only. All other instructions, when present in the code segment above, have both a mnemonic and operand. An `Operand` itself may be a `Label`, an `Address` or an `Offset`.

EBNF

```
Operand = Label | Address | Offset.
```

1.3.3 Directives

A `Directive` is a pseudo-instruction to the assembler and has a format similar to that of an instruction except that all mnemonics begin with a period (.).

EBNF

```
Directive = CString .
```

The only possible directive is presented in Table 1.1.

Directive	**Mnemonic**	**Description**
CString	`.cstring` *StringOperand*	8-bit ASCII characters

Table 1.1: All directive mnemonics

1.3.4 Comments

A `Comment` starts with a "`;`" character, and terminates with an `EOL`. For example, all lines except 2, 3, 4, 16, 17 and 18 are commented.

Mnemonic	**Description**
add	Add
addv	Add Value to an Local Variable
and	Bitwise And
br	Branch at Offset
brf	Branch If False at Offset
call	Call Function at Offset
dec	Decrement
decv	Decrement Variable
div	Divide
dup	Duplicate
enter	Set up Frame on Function Entry
halt	Stop
inc	Increment
incv	Increment Variable

Table 1.2: **Cm VM** opcodes represented by their mnemonics (in alphabetic order)

Mnemonic	**Description**
ldc	Load Constant
ldv	Load Variable
mul	Multiply
neg	Negate
not	Bitwise One's Complement
or	Bitwise Or
pop	Remove Top of Stack
rem	Remainder
ret	Clean up Frame and Return from Function
shl	Shift Left
shr	Shift Right
sub	Substract
stv	Store Variable
teq	Test for Equality
tge	Test for Greater or Equal
tgt	Test for Greater Than
tle	Test for Less Than or Equal
tlt	Test for Less Than
tne	Test for Non Equality
trap	Trap Exception to occur
xor	Bitwise Exclusive Or

Table 1.3: **Cm VM** opcodes represented by their mnemonics (in alphabetic order)

1.4 Tokens

Each Cm assembly program consists of five basic building blocks or tokens:

1. mnemonic names,
2. labels,
3. addresses (represented by unsigned integers),
4. offsets (represented by signed integers), and
5. comments.

The syntax of each token is used by the lexical analyzer of the Cm assembler to translate Cm assembly programs into virtual machine code. The complete syntax of the Cm assembly language therefore is summarized below.

EBNF

```
AssemblyUnit  = { LineStatement } EOF .
LineStatement = [ Label ] [ Instruction | Directive ] [ Comment ] EOL .

Label         = IDENTIFIER .

Instruction   = Mnemonic [ Operand ] .
Mnemonic      = "add" | "and" | ...
Operand       = Label | Address | Offset .
Address       = NUMBER .
Offset        = NUMBER .

Directive     = CString .
CString       = ".cstring"   StringOperand .

Comment       = ";" AnyCharExceptEOL .

AnyChar       = 0 .. 255 (@). unicode (< 0x00C0)
Digit         = "0" .. "9".
Letter        = "a" .. "z" | "A" .. "Z".
IDENTIFIER    = Letter { Letter | Digit } .
NUMBER        = Digit { Digit } .
EOL           = "\n" | "\r" | "\r\n".
EOF           = <control-Z>.
```

1.5 Assembling a Cm Assembly Program

To assemble a Cm assembly program named `Pgm.asm`, the following command is entered:

```
C:\>cma [-h] [-l] [-v] Pgm.asm
```

This command invokes the Cm assembler and generates one or two files:

- a binary virtual code file `pgm.exe` (for the target)
- a complete listing file `pgm.lst` of all virtual code along with a label table.

Chapter 2

The Cm Virtual Machine Instruction Set

In this chapter, we discuss the instruction set of a virtual machine (VM) to support programming languages intended for small footprint embedded systems. The instruction set of the VM is tailored to support a subset of the C programming language, called Cm[1], intended for a restrictive microcontroller environment such as an ATmega368P 8-bit microcontroller with 32K bytes Flash and 2K bytes SRAM used in the Arduino Nano.

We carefully cover instruction formats, addressing modes and type representation as well as introduce the entire instruction set with practical examples.

[1] **Cm** as a subset of the **C** programming language for **m**icrocontrollers. In music, Cm or C- means C minor. A C minor chord is a chord that has C as a root :)

Before moving on, we present in Table 2.1, some naming conventions used to express the size and range of fields within operation codes and operands.

Symbol	Meaning and Size	Range Value
<i>	signed number	<i3> or <i4> or <i8> or <i16> or <i32>
<u>	unsigned number	<u3> or <u4> or <u8> or <u16> or <u32>
<v>	value	<i> or <u>
<n>	number	<i> or <u>
<a>	address	<u>
<o>	offset	<i>
<u3>	3-bit unsigned	0..7
<i3>	3-bit signed	-4..3
<u4>	4-bit unsigned nibble	0..15
<i4>	4-bit signed nibble	-8..7
<u5>	5-bit unsigned	0..31
<i5>	5-bit signed	-16..15
<u8>	8-bit unsigned	0..255
<i8>	8-bit signed	-128..127
<u16>	16-bit unsigned	0..65535
<i16>	16-bit signed	-32768..32767
<u32>	32-bit unsigned	0..4294967295
<i32>	32-bit signed	-2147483648..2147483647

Table 2.1: Instruction Format Naming Conventions.

2.1 Instruction Formats

instruction formats

Instruction formats determine the layout and size for each instruction of a virtual machine. Not surprisingly, the choice of instruction format is a fundamental design decision and involves several factors.

The first factor to consider is the instruction size itself. Making instructions short is especially important for embedded systems where memory is a limited resource. But keeping the size of an instruction very small can make it harder to decode in order to execute it. In general though, an instruction consists of an **operation code** (opcode) immediately followed by operands (or instruction parameters).

The **Cm VM** instruction formats are quite straightforward and come in one of three main formats. The **inherent** format has no operands and is self-contained in one byte, including immediate operands and displacements. The **byte-parameter** format has a single one-byte operand and requires two bytes of memory. And the **word-parameter** format has a single two-byte operand and requires three bytes of memory. All opcodes and most instructions of **Cm VM** are in inherent format and therefore require only a single byte. Many instructions, too, result in data transfer to and from the operand (32-bit) stack.

All formats are shown in Figure 2.1 below.

```
Format:
          +---------+
Inherent  |  opcode |
          +---------+

byte or   +---------+---------+
8-bit     |  opcode | operand |  operand = <i8> or <u8>
Operand   +---------+---------+

          +---------+-------------------+
16-bit    |  opcode |      operand      |  operand = <i16> or <u16>
Operand   +---------+-------------------+

          +---------+---------------------------------------+
32-bit    |  opcode |                operand                | operand = <i32> or <u32>
Operand   +---------+---------------------------------------+

Bits      |<---8--->|<---8--->|<---8--->|<---8--->|<---8--->|
```

Figure 2.1: Instruction Formats for **Cm VM**.

A second factor to consider ensures that there is sufficient space in the instruction format to express all operations required.

The third factor to consider is the number of bits in an address field. In our case, making the 8-bit byte as the basic unit of memory was the most realistic option for 8-bit microcontrollers. The maximum addressable memory of the **Cm VM** is 64K.

The fourth factor is concerned with the usage of relative addresses. Relative addressing allows position-independent code meaning that the virtual machine code can be loaded anywhere in memory. The generation of position-independent code follows one important rule of never using absolute addressing. This is achieved by using the instruction pointer (`ip`) as the base register for a relative offset. **Cm VM** mainly uses relative offset for flow control (branching and calling). Very short branches are optimized by embedding an immediate 5-bit offset in a one byte opcode where the range is limited to +15 or -16 bytes from the following opcode. For short branches, the byte following the branch opcode is treated as an 8-bit offset to be used to calculate the effective address of the next instruction. Finally, long branches require 16-bit offsets. Because instructions are three bytes, long branches are expensive in terms of space.

2.2 Addressing Modes

Addressing modes specify where operands are to be retrieved, either from memory, registers, accumulators, stacks and so on. Bearing in mind the tiny nature of our embedded systems, two general methods may be used to reduce the addressing size of operands within instructions:

addressing modes

- Move the operand into a register when it is used several times.
- Use a single specification to select operands.

The above methods work well for simple operations, but are a nightmare when several intermediate results are needed. By exploiting the stack machine architecture and using the operand stack for our instruction set, we can eliminate a number of non-applicable addressing modes such as direct, register, register indirect, and so on. Consequently, only the four addressing modes below are efficiently supported by **Cm VM**:

1. Stack (or inherent),
2. Immediate, and
3. Relative.

For **stack** or **inherent addressing**, otherwise known as zero-address instructions, both source and destination operands are implicitly retrieved from the operand stack. This makes virtual machine instructions as short as possible by reducing address lengths to zero. Hence, inherent instructions have no operands and are self-contained in a single byte. Table 2.2 illustrates all inherent instructions sorted by opcode.

inherent format

Hex	Binary	Mnemonic	Operand	Description	Operation
00	000 00000	halt		Stop virtual machine	
01	000 00001	pop		Remove top of stack	[... = v
02	000 00010	dup		Duplicate top of stack	[r r = v
03	000 00011	exit		Return from function with parameters	
04	000 00100	ret		Return from function	
05	000 00101	—		Reserved for future used	
06	000 00110	—		Reserved for future used	
07	000 00111	—		Reserved for future used	
08	000 01000	—		Reserved for future used	
09	000 01001	—		Reserved for future used	
0A	000 01010	—		Reserved for future used	
0B	000 01011	—		Reserved for future used	
0C	000 01100	not		Bitwise one's complement	[r = ~v
0D	000 01101	and		Bitwise AND	[r = v1 & v2
0E	000 01110	or		Bitwise OR	[r = v1 | v2
0F	000 01111	xor		Bitwise exclusive OR	[r = v1 ^ v2
10	000 10000	neg		Negate	[r = -v
11	000 10001	inc		Increment	[r = ++v
12	000 10010	dec		Decrement	[r = --v
13	000 10011	add		Addition	[r = v1 + v2
14	000 10100	sub		Subtraction	[r = v1 - v2
15	000 10101	mul		Multiplication	[r = v1 * v2
16	000 10110	div		Division	[r = v1 / v2
17	000 10111	rem		Remainder, modulo	[r = v1 % v2
18	000 11000	shl		Shift left	[r = v1 << v2
19	000 11001	shr		Shift right	[r = v1 >> v2
1A	000 11010	teq		Test for equal	[r = v1 == v2
1B	000 11011	tne		Test for not equal	[r = v1 != v2
1C	000 11100	tlt		Test for less than	[r = v1 < v2
1D	000 11101	tgt		Test for greater than	[r = v1 > v2
1E	000 11110	tle		Test for less or equal	[r = v1 <= v2
1F	000 11111	tge		Test for greater or equal	[r = v1 >= v2

Table 2.2: Inherent (one byte, no operand) Instructions.

immediate format

For **immediate addressing**, the operand is included as part of the opcode itself and is automatically fetched in one byte. Hence, immediate instructions are also self-contained in a single byte. Although 8 bits is obviously limited, it is handy for specifying small integer literals. Within the immediate addressing mode, the format specifies one or more additional fields with different ranges (<i3>, <u3>, or <i5>) and subdivides this mode into further instruction groupings as shown in Table 2.3.

Hex	Mnemonic	Operand	Description	Operation
30..4F	br.i5	Label (<*i5*>)	Branch always	pc += <i5>
50..6F	brf.15	Label (<*i5*>)	Branch if v != 1	if (TOS != 1) pc += <i5>
70..8F	enter.u5	FctInfo (<*i5*>)	Set up frame	See instruction section
90..97	ldc.i3	<*i3*>	Load constant	r = <i3>
98..9F	addv.u3	<*u3*>	Add TOS to variable	bp[<u3>] += TOS
A0..A7	ldv.u3	<*u3*>	Load variable	r = bp[<u3>]
A8..AF	stv.u3	<*u3*>	Store variable	bp[<u3>] = r

Table 2.3: Immediate (one byte) Instructions.

relative format

Finally, for **relative addressing**, the opcode is followed by either a one byte (8-bit) or two byte (16-bit) operand. Immediate addressing, in this sense, is the optimized version of relative addressing. The operand represents an offset (<i8>, <u8>, or <u16>) or index (<i8>, <u8>, or <u16>), and is used to reference local variables and arguments. Within the relative addressing mode, the format also specifies fields of different ranges (<i8> or <i16>), and subdivides this mode into further instruction groupings as shown in Table 2.4.

Hex	Mnemonic	Operand	Description	Operation
B0	addv.u8	<*u8*>	Add TOS to variable	bp[<u8>] += TOS
B1	ldv.u8	<*u8*>	Load variable	r = bp[<u8>]
B2	stv.u8	<*u8*>	Store variable	bp[<u8>] = r
B3	incv.u8	<*u8*>	Increment variable	++bp[<u8>]
B4	decv.u8	<*u8*>	Decrement variable	--bp[<u8>]
BF	enter.u8	<*u8*>	Set up frame on function entry	See instruction section
D5	lda.i16	<*i16*>	Load address	See instruction section
D9	ldc.i8	<*i8*>	Load an 8-bit constant	[r = <i8>
DA	ldc.i16	<*i16*>	Load a 16-bit constant	[r = <i16>
DB	ldc.i32	<*i32*>	Load a 32-bit constant	[r = <i32>
E0	br.i8	Label (<*i8*>)	Branch relative always	pc += <i8>
E1	br.i16	Label (<*i16*>)	Branch relative always	pc += <i16>
E3	brf.i8	Label (<*i8*>)	Branch relative if false	if (!r) pc += <i8>
E7	call.i16	Label (<*i16*>)	Call relative	
FF	trap	<*u8*>	Trap to vector	pc = vt[<u8>]

Table 2.4: Relative (two, three, or four byte) Instructions.

2.3 Instruction Set

The following section provides an alphabetized listing of the entire **Cm VM** instruction set. A detailed description of each instruction makes up the bulk of this section (and chapter), and serves as a reference. Descriptions are presented in alphabetical order using the following format:

- The assembler syntax.
- A concise description of how it works.
- An ANSI C description of its corresponding operation. The description is designed for readability and not optimization. On the other hand, the target **Cm VM** also written in ANSI C, is optimized for maximum performance.
- The layout of the stack before the operation.
- The layout of the stack after the operation.
- One or more examples.

add — Addition

Assembler Syntax: `add`

Description: The `add` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1` + `v2` and is pushed back onto the operand stack.

Operation:

```
void add() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 + v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
                ; [...
ldc     2
                ; [2, ...
ldc     -3
                ; [2, -3, ...
add
                ; [-1, ...
```

addv Add Value to a Local Variable

Assembler Syntax: `addv <u3>`

Description: Adds a value to the content of the specified object local variable. The `addv` pops the `value` from the operand stack. A function parameter is also considered as a local variable (see the ordering and layout on the operand stack in the `enter`/`ret` instructions). This instruction has one operand `<u3>` which indicates the local variable number (offset in the current frame pointer `fp`) in the object specified to add.

Operation:

```
void addv(u3 localVarNumber) {
    i32 value = pop();

    fp[localVarNumber] += value;
}
```

Stack Before: `[value, ...`

Stack After: `[...`

Example:

```
public void fct() {
    int n;          // local variable 0

    // ...

    n += 3;

          ldc     3
                        ; [3, ...
          addv    0
                        ; [...
}
```

and

Bitwise And

Assembler Syntax: `and`

Description: The `and` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 & v2` and is pushed back onto the operand stack.

Operation:

```
void and() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 & v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
                ; [...
      ldc     0b0101
                ; [5, ...
      ldc     0b0110
                ; [5, 4, ...
      and
                ; [4, ...
```

br Branch at Address

Assembler Syntax: `br`

Description: Unconditional branch to relative or absolute address. The `br` adds the `offset` (if relative) or sets the `addr` (if absolute) to the instruction pointer `ip`.

Operation:

```
void brI8(i8 offset) { ip += offset; } // relative offset

void brU16(u16 addr) { ip  = addr; } // absolute address
```

Stack Before: `[...`

Stack After: `[...`

Example:

```
While

    ; ...

    br      While
```

brf

Branch If False at Address

Assembler Syntax: `brf`

Description: Conditional branch if the top of the operand stack is false. The `brf` pops the value `v` from the operand stack and adds the `offset` (if relative) or sets the `addr` (if absolute) to the instruction pointer `ip` if `v` is false. Otherwise, if `v` is true then one (if relative) or two (if absolute) is added to `ip`.

Operation:

```
void brfI8(i8 offset) { // relative offset
    bool v = (bool)pop();

    ip += v ? 1 : offset;
}

void brfU16(u16 addr) { // absolute address
    bool v = (bool)pop();

    ip  = v ? ip+2 : addr;
}
```

Stack Before: `[r, ...`

Stack After: `[...`

Example:

```
            ; [...
     ldc  3
            ; [3, ...
     ldc  2
            ; [3, 2, ...

If   tlt    ; if ( 3 < 2 )

            ; [0, ...
     brf  Else

; ...

Else
```

call Call Function at Address

Assembler Syntax: call <u8> or <u16>

Description: The call pushes the return address ra onto the operand stack. The call with an <i8> operand adds the relative offset to the instruction pointer ip. The call with an <u16> operand replaces the the instruction pointer ip by the absolute address <u16>.

Operation:

```
void callI8(i8 offset) { // using a relative offset
    push(ip+1);
    ip += offset;
}

void callU16(u16 addr) { // to absolute address
    push(ip+2);
    ip = addr;
}
```

Stack Before: [ra, ...

Stack After: [...

Example:

```
    call    Fct
RA  ldc     0      ; label RA corresponds to the return address pushed

    ; ...

Fct
```

dec

Decrement

Assembler Syntax: `dec`

Description: Decrements the top of the operand stack.

Operation:

```
void dec() {
    --stack[sp];
}
```

Stack Before: `[v, ...`

Stack After: `[v, ...`

Example:

```
                ; [...
ldc     10
                ; [10, ...
dec
                ; [9, ...
```

decv

Decrement Variable

Assembler Syntax: `decv <u3>`

Description: Decrements the content of the specified object local variable. A function parameter is also considered as a local variable (see the ordering and layout on the operand stack in the `enter/ret` instructions). This instruction has one operand `<u3>` which indicates the local variable number in the object specified to decrement.

Operation:

```
void decv(u3 localVarNumber) {
    --stack[localVarNumber];
}
```

Stack Before: `[...`

Stack After: `[...`

Example:

```
public void fct() {
    int n;          // local variable 0

    // ...

    --n;

          addv    0
                         ; [...
}
```

div Divide

Assembler Syntax: `div`

Description: The `div` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 / v2` and is pushed back onto the operand stack.

Operation:

```
void div() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 / v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
              ; [...
ldc     3
              ; [3, ...
ldc     2
              ; [3, 2, ...
div
              ; [1, ...
```

dup — Duplicate

Assembler Syntax: `dup`

Description: Duplicates the top item on the operand stack.

Operation:

```
void dup() {
    i32 v = pop();

    stack[++sp] = (i32)v;
    stack[++sp] = (i32)v;
}
```

Stack Before: `[v, ...`

Stack After: `[v, v, ...`

Example:

```
                ; [...
    ldc     3
                ; [3, ...
    dup
                ; [3, 3, ...
```

enter Set up Frame on Function Entry

Assembler Syntax: `enter <u5>` or `<u8>`

Description: The `enter` instruction is the first instruction of a function. It saves the frame context of its caller, and sets up the context of the current function. The `enter <u5>` instruction takes only one byte and has an immediate operand `u5` in the opcode. On the other hand, the `enter <u8>` instruction has a one-byte operand `u8`. Each operand represents important information about the frame context of the current function. This information is used by the instruction `ret` to clean up the operand stack. As such, the operand is divided into three fields containing a flag `v` if the function returns a value or not (`void`), the number of parameter(s) passed to a function (`np`), and the number of local variables to be allocated within the function (`nl`).

The instruction `enter <u5>` is optimized for functions up to a maximum of 3 parameters and 3 local variables. The instruction `enter <u8>` takes two bytes but permits up to 7 parameters and 7 local variables. To access local variables (including parameters) on the operand stack via the related instructions (`ldv`, `stv`, and so on), the following function is used:

```
int getFrameOffset(int v, int np, int nl) { return 2 + np + nl + v; }
```

Operation:

```
 7 6 5 4 3 2 1 0                    7 6 5 4 3 2 1 0
+-----+-+---+---+                  +-+-+-----+-----+
|0 1 1|v|np |nl |                  |x|v| np  | nl  |
+-----+-+---+---+                  +-+-+-----+-----+
   opcode.<u5>             opcode        <u8>

   (0x60..0x7F)             0xF5         <u8>



 7 6 5 4 3 2 1 0
+-+-+-----+-----+
|x|v| np  | nl  |
+-+-+-----+-----+
function info (fi)
```

```
void enter(int u5) {
    int   fi.v  = (u5 >> 4) & 0x01;
    int   fi.np = (u5 >> 2) & 0x03;
    int   fi.nl =  u5       & 0x03;
```

```
//  int   fi = (v << 6) | (np << 3) | nl;

    retAddr = stack[sp--];  // pop (save) caller's return address
    sp += nl;               // allocate space for local variables
    stack[++sp] = fi;       // push function info
    stack[++sp] = bp;       // push (save) caller's bp (context)
    bp = sp;                // set frame context for the current function
    stack[++sp] = retAddr;  // push back the caller's return address
}

void enter(int u8) {
    int   np = (u8 >> 4) & 0x0F;
    int   nl =  u8       & 0x0F;

    stack[++sp] = bp;  // push (save) caller's bp (context)
    bp = sp;           // set frame context for the current function
    sp += nl;          // allocate space for local variables
}
```

Stack Before:

```
sp -> | retAddr |
      |   pn-1  |
      ~   ...   ~
      |   p1    |
      |   p0    |
      +---------+

      [p0, p1, ..., pn-1, retAddr, ...
```

Stack After:

```
int getFrameOffset(int v, int np, int nl) { return 2 + np + nl + v; }
```

```
sp->bp->|caller bp| bp + 0
        | retAddr | bp - 1
        | fctInfo | bp - 2
        |   ln-1  | bp - 3
        ~   ...   ~   ...
        |   l1    |
        |   l0    |
        |   pn-1  |
        ~   ...   ~   ...
        |   p1    |
        |   p0    |
        +---------+

        [p0, p1, ..., pn-1, l0, l1, ..., ln-1, <u5>, ra, bp, ...
```

Example: The following is the stack state after the execution of the enter (*):

```
bp -> |caller bp| bp + 0
      | retAddr | bp - 1
      |   <u5>  | bp - 2
      |   j     | bp - 3
      |   i     | bp - 4
      |   p     | bp - 5
      +---------+

      [p, retAddr, bp, i, j, ...


void fct(int p) {     // function with one parameter and two local variables
    int i, j;         // where v = 0, np = 0x01, and nl = 0x02

                      // enter  6  ; (0x01 << 2)|0x02
                      // (*)
                      //         ;                                bp - getFrameOffset(v,np,nl)
    j = i = p;        // ldv    0 ; load from stack[bp-2] or stack[bp - getFrameOffset(0, 1, 2)]
                      // dup
                      // stv    1 ; store to  stack[bp+1] or stack[bp - getFrameOffset(1, 1, 2)]
                      // stv    2 ; store to  stack[bp+2] or stack[bp - getFrameOffset(2, 1, 2)]
    //...
}
```

Note: `enter 0` is useless since it means to set up a frame with parameters and no local variables. In such a case, the instruction can be removed for optimization purposes. Hence, the Cm compiler is removes all `enter 0` when it generates the code.

halt

Stop Virtual Machine

Assembler Syntax: `halt`

Description: Stops the virtual machine. This instruction is also used to set breakpoints in the **CDotM**.

Operation:

```
void halt() {
    // Stop the virtual machine.
}
```

Stack Before: `[...`

Stack After: `[...`

Example:

```
                ; [...
ldc     3
                ; [3, ...
halt
```

inc — Increment

Assembler Syntax: `inc`

Description: Increments the top of the operand stack.

Operation:

```
void inc() {
    ++stack[sp];
}
```

Stack Before: `[...`

Stack After: `[...`

Example:

```
                ; [...
ldc     2
                ; [2, ...
inc
                ; [3, ...
```

incv Increment Variable

Assembler Syntax: `incv <u3>`

Description: Increments the content of the specified object local variable. A function parameter is also considered as a local variable (see the ordering and layout on the operand stack in the `enter`/`ret` instructions). This instruction has one operand `<u3>` which indicates the local variable number in the function specified to increment.

Operation:

```
void incv(u3 localVarNumber) {
    ++stack[localVarNumber];
}
```

Stack Before: `[...`

Stack After: `[...`

Examples:

```
public void fct() {
    int n;          // local variable 0

    // ...

    ++n;

          incv    0
                       ; [...
}
```

```
module Counter {
    public int count;
    public void fct(ref Counter this, int p, Counter c) {
            enter  ?? ; offsets ==> this = 0; p = 1; c = 2; v = 3
        int v;

        v++;
            ldv    0  ; push this
            incv   3  ; this.v++

        p++;
            ldv    0  ; push this
            incv   1  ; this.p++

        c.count++;
            ldv    2  ; push this
            incf   0  ; this.count++

            ret
    }
}
```

ldc Load Constant

Assembler Syntax: `ldc <i3> or <i8> or <i16>`

Description: Loads a constant onto the operand stack. The `ldc` pushes the integer `<i>` onto the operand stack.

Operation:

```
void ldc(I3  i3)  { stack[++sp] = i3;  }  //     [-4..3]
void ldc(I8  i8)  { stack[++sp] = i8;  }  //   [-128..127]
void ldc(I16 i16) { stack[++sp] = i16; }  // [-32768..32767]
```

Stack Before: `[...`

Stack After: `[<i>, ...`

Where `<i>` represents `<i3>`, `<i8>`, or `<i16>`.

Example:

```
ldc      1
ldc      -9
ldc      130
              ; [1, -9, 130, ...
```

ldv Load from Local Variable

Assembler Syntax: `ldv <u3> or <u8>`

Description: Retrieves a value or a reference from a local variable and pushes it onto the operand stack. A function parameter is also considered as a local variable (see ordering in the `enter`/`ret` instructions). This instruction has one operand, `u3` or `u8`, which indicates the variable number in the current stack frame to push.

Operation:

```
void ldv(u8 localVarNumber) {
    push(stack[localVarNumber]);
}
```

Stack Before: `[...`

Stack After: `[v, ...`

Example:

```
public void fct() {
    int n;          // local variable 0

    // ...

          ldv     0
                         ; [v, ...
          trap    0
                         ; [...
}
```

mul

Multiply

Assembler Syntax: `mul`

Description: The `mul` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 * v2` and is pushed back onto the operand stack.

Operation:

```
void mul() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 * v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
                ; [...
      ldc   2
                ; [2, ...
      ldc   -3
                ; [2, -3, ...
      mul
                ; [-6, ...
```

neg

Negate

Assembler Syntax: `neg`

Description: The `neg` pops the value `v` from the operand stack. The result `r` is `-v`, the bitwise two's complement of `v`, and is pushed back onto the operand stack.

Operation:

```
void neg() {
    stack[sp] = (i32)-stack[sp];
}
```

Stack Before: `[v, ...`

Stack After: `[-v, ...`

Example:

```
                ; [...
ldc     9
                ; [9, ...
neg
                ; [-9, ...
```

not Bitwise One's Complement

Assembler Syntax: `not`

Description: The `not` pops the value `v` from the operand stack. The result `r` is `~v`, the bitwise one's complement of `v`,and is pushed back onto the operand stack.

Operation:

```
void not() {
    stack[sp] = (i32)~stack[sp];
}
```

Stack Before: `[...`

Stack After: `[...`

Example:

```
                ; [...
ldc     0xAA55
                ; [0xAA55, ...      or [0b1010101001010101, ...
not
                ; [0x55AA, ...      or [0b0101010110101010, ...
```

or — Bitwise Or

Assembler Syntax: `or`

Description: The `or` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 | v2` and is pushed back onto the operand stack.

Operation:

```
void or() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 | v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
               ; [...
ldc     0b0101
               ; [5, ...
ldc     0b0110
               ; [5, 4, ...
or
               ; [7, ...
```

pop Remove Top of Stack

Assembler Syntax: `pop`

Description: Discards the top of stack.

Operation:

```
void pop() { --sp; }
```

Stack Before: `[v, ...`

Stack After: `[...`

Example:

```
                ; [...
    ldc    5
                ; [5, ...
    pop
                ; [...
```

rem Remainder

Assembler Syntax: `rem`

Description: The `rem` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 % v2` and is pushed back onto the operand stack.

Operation:

```
void rem() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 % v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
                ; [...
ldc     3
                ; [3, ...
ldc     2
                ; [3, 2, ...
rem
                ; [1, ...
```

ret — Clean up Frame and Return

Assembler Syntax: `ret`

Description: The `ret` instruction is the last instruction of a function. It returns and restores the frame context of its caller, and sets up the context of the current function. The operand `u4` is divided into two fields of values containing a flag `v` if the function returns a value or not (`void`), and the number of local variables that has been allocated within the function (`nl`).

Operation:

```
 3 2 1 0
+-+-----+
|v|  nl |  if v = 0 means the operand stack contains no return value (void)
+-+-----+     v = 1 means the operand stack contains a value to be returned
    u4
```

```
void ret(int u4) {
    int   v  = (u4 >> 3) & 0x01;
    int   nl =  u4       & 0x07;
    int   (*retAddr)();

    if (v) v = stack[sp--]; // save the return value in v (if any)

    sp -= nl;               // deallocate space for local variables
    bp = stack[sp--];       // pop (restore) caller's bp (context)
    retAddr = stack[sp--];  // pop (save) caller's return address
    bp = sp;                // set frame context for the current function
    sp += nl;               // allocate space for local variables
}

void ret() {
    int   u5 = stack[bp-2];
    int   v  = (u5 >> 4) & 0x01;
    int   np = (u5 >> 2) & 0x03;
    int   nl =  u5       & 0x03;

    int   (*retAddr)();
    int   retVal;

    if (v) retVal = stack[sp--]; // save the return value in v (if any)
    bp = stack[sp--];            // pop (restore) caller's bp (context)
    retAddr = stack[sp--];       // pop (save) caller's return address
    sp -= (np+nl+1);             // deallocate space for parameters, local variables, and <u5>
    if (v) stack[++sp] = retVal; // push back the return value (if any)
    stack[++sp] = retAddr;       // push back the caller's return address
}
```

Stack Before:

```
  sp->|  retVal | (if any)
  bp->|caller bp| bp + 0
      | retAddr | bp - 1
      |   <u5>  | bp - 2
      |   ln-1  | bp - 3
      ~   ...   ~   ...
      |   l1    |
      |   l0    |
      |   pn-1  |
      ~   ...   ~   ...
      |   p1    |
      |   p0    |
      +---------+

      [p0, p1, ..., pn-1, l0, l1, ..., ln-1, <u5>, ra, bp, ...

sp -> |  retVal | (if any)
      |   ln-1  |
      ~   ...   ~   ...
      |   l1    | bp + 2
      |   l0    | bp + 1
bp -> |caller bp| bp + 0
      | retAddr | bp - 1
      |   pn-1  | bp - 2
      ~   ...   ~   ...
      |   p1    |
      |   p0    |
      +---------+

      [p0, p1, ..., pn-1, ra, bp, l0, l1, ..., ln-1, ...
```

Stack After:

```
  sp->| retAddr |
  sp->|  retVal | (if any)
      +---------+

      [p0, p1, ..., pn-1, l0, l1, ..., ln-1, <u5>, ra, bp, ...


sp -> |   ln-1  |
      ~   ...   ~   ...
      |   l1    | bp + 2
      |   l0    | bp + 1
bp -> |caller bp| bp + 0
      | retAddr | bp - 1
      |   pn-1  | bp - 2
      ~   ...   ~   ...
      |   p1    |
      |   p0    |
      +---------+

      [p0, p1, ..., pn-1, ra, bp, l0, l1, ..., ln-1, ...
```

Example: The following is the stack state after the execution of the enter (∗):

```
sp -> |    j     | bp + 2
      |    i     | bp + 1
bp -> |caller bp| bp + 0
      | retAddr | bp - 1
      |    p     | bp - 2
      +---------+

      [p, retAddr, bp, i, j, ...


void fct(int p) {     // function with one parameter and two local variables
    int i, j;         // where np = 0x01 and nl = 0x02

                      // enter  6  ; (0x01 << 2)|0x02

                      // (*)

                      //          ;                                bp - getFrameOffset(v,np)
    j = i = p;        // ldv    0 ; load from stack[bp-2] or stack[bp - getFrameOffset(0, 1)]
                      // dup
                      // stv    1 ; store to  stack[bp+1] or stack[bp - getFrameOffset(1, 1)]
                      // stv    2 ; store to  stack[bp+2] or stack[bp - getFramrOffset(2, 1)]

    //...
                      // ret
}
```

shl

Shift Left

Assembler Syntax: `shl`

Description: The `shl` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 << v2` and is pushed back onto the operand stack.

Operation:

```
void shl() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 << v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
                ; [...
    ldc     0b0110
                ; [6, ...
    ldc     1
                ; [6, 1, ...
    shl
                ; [12, ...
```

shr Shift Right

Assembler Syntax: `shr`

Description: The `shr` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 >> v2` and is pushed back onto the operand stack.

Operation:

```
void shr() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 >> v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
                ; [...
    ldc     0b0110
                ; [6, ...
    ldc     1
                ; [6, 1, ...
    shr
                ; [3, ...
```

sub

Substract

Assembler Syntax: `sub`

Description: The `sub` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 - v2` and is pushed back onto the operand stack.

Operation:

```
void sub() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 - v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

stv Store into Local Variable

Assembler Syntax: `stv <u3> or <u8>`

Description: Pops a value or a reference from the operand stack and stores it in a parameter or a local variable. A function parameter is also considered as a local variable (see ordering in the `enter`/`ret` instructions). This instruction has one operand, `u3` or `u8`, which indicates the variable number in the current stack frame to push.

Operation:

```
void stv(u8 localVarNumber) {
    i32 value = pop();

    stack[localVarNumber] = value;
}
```

Stack Before: `[v, ...`

Stack After: `[...`

Example:

```
public void fct() {
    int n;          // local variable 0

    // ...

    n = 3;

          ldc     3
                       ; [3, ...
          stv     0
                       ; [...
}
```

teq

Test for Equality

Assembler Syntax: `teq`

Description: The `teq` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 == v2` and is pushed back onto the operand stack.

Operation:

```
void teq() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 == v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

tge — Test for Greater or Equal

Assembler Syntax: `tge`

Description: The `tge` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 >= v2` and is pushed back onto the operand stack.

Operation:

```
void tge() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 >= v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

tgt Test for Greater Than

Assembler Syntax: `tgt`

Description: The `tgt` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 > v2` and is pushed back onto the operand stack.

Operation:

```
void tgt() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 > v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

tle Test for Less Than or Equal

Assembler Syntax: `tle`

Description: The `tle` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 <= v2` and is pushed back onto the operand stack.

Operation:

```
void tle() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 <= v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

tlt Test for Less Than

Assembler Syntax: `add`

Description: The `tlt` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 < v2` and is pushed back onto the operand stack.

Operation:

```
void tlt() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 < v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

tne Test for Non Equality

Assembler Syntax: `tne`

Description: The `tne` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 != v2` and is pushed back onto the operand stack.

Operation:

```
void tne() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 != v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

trap **Trap**

Assembler Syntax: `trap <u8>`

Description: The `trap` instruction provides customized services for developers. In other words, its behavior can be defined for the need of the embedded target application. This instruction has one operand `u8` which indicates the service number requested.

The current **Cm VM** makes 8 services available for console output services (debugging purpose). In our case, the behavior of the `trap` pops the value `v` from the operand stack and prints the value on the console output.

The complete implementation of the `trap` instructions below are isolated in the `system.h` and `system.c` files with the source code of the **Cm VM**. Developers can replace these services with their own implementations.

Operation:

```
trap 0x82 (PutI)  - Print a signed integer (int) on console output.
trap 0x83 (PutU)  - Print an unsigned integer (uint) on console output.
trap 0x81 (PutC)  - Print a character (char) on console output.
trap 0x80 (PutB)  - Print a boolean (bool) on console output.
trap 0x86 (PutX)  - Print a byte (u8) on console output. The byte
                    is converted to two hexadecimal digits.
trap 0x85 (PutS)  - Print a C string on console output.
trap 0x87 (PutN)  - Print a newline on console output.
```

Stack Before: `[v, ...`

Stack After: `[...`

Example:

xor Bitwise Exclusive Or

Assembler Syntax: `xor`

Description: The `xor` pops the values `v1` and `v2` from the operand stack. The result `r` is `v1 ^ v2` and is pushed back onto the operand stack.

Operation:

```
void xor() {
    i32 v2 = pop();
    i32 v1 = stack[sp];

    stack[sp] = (i32)(v1 ^ v2);
}
```

Stack Before: `[v1, v2, ...`

Stack After: `[r, ...`

Example:

```
                ; [...
ldc     0b0101
                ; [5, ...
ldc     0b0110
                ; [5, 4, ...
xor
                ; [3, ...
```

Index