`Karma` Computer

Tigran Koshkelian

Contents

1 Architecture

Karma is a computer with a Von Neumann architecture having an *address space* of 2^{20} *cells*, one 32-bit *machine word* each.

The processor has 16 one-word (32 bits each) *registers* r0-r15, as well as an additional flags register (also one-word). Their usage is described in Table 1.

Table 1: Usage of Karma processor registers

r0 – r12	Free usage
r13	Call frame pointer
r14	Stack head pointer
r15	Instruction pointer
flags	Comparison operation result

The usage of the r14 register is described in the Stack-related commands section, the r13 and the r15 registers are discussed in the Karma calling convention section, and the bits of the flags register are detailed in the Comparisons and jumps section.

2 Karma assembler standard

2.1 Command formats

Each command takes up *exactly one* word, 8 high bits of which specify the operation code and the use of the rest 24 bits is command-specific. With respect to the operation code each command may be of one of the following formats:

Table 2: Karma processor command formats

		8 bits	**4 bits**	**4 bits**	**16 bits**	**Syntax sample**
Register-memory	RM	command code	register	memory address		load r0, 12956
Register-register	RR		receiver register	source register	source modifier	mov r1, r2, -0xa21
Register-immediate	RI		register	immediate value		ori r2, 64
Jump	J		ignored	memory address		calli 01547

The *memory address* operand for RM and J commands is interpreted as an unsigned integer representing the number of a memory cell (in 0-indexing). The bit size of the operand (20 bits) allows it to represent any memory cell (there are 2^{20} of them).

For RM and J commands the *memory address* operand in the assembler code may be represented as:

- A decimal number (non-prefixed, not starting with 0)
- An octal number (with a 0 prefix)
- A hexadecimal number (with a 0x or a 0X prefix)

The same applies to the *source modifier* and *immediate value* operands for RR and RI commands respectively. If the operand is negative, in octal and hexadecimal representations the minus sign is placed before the prefix.

For the sake of not overcomplicating matters all arguments of any command are required. When using an RR command, if one does not wish to modify the value represented by the source modifier, the *source modifier* should be specified as 0.

2.2 Types

2.2.1 Basic types

There are three architecture-defined basic types in the Karma assembler:

- uint32 is a one-word unsigned integral type
- uint64 is a two-word unsigned integral type
- double is a two-word real-valued type

The two-word types (i.e. uint64 and double) are represented by two consecutive memory cells or registers. The low 32 bits are placed in the first memory cell/register and the high 32 bits – in the subsequent one. When referring to a two-word type value in commands, the specified memory cell/register is *always* considered to be the first one (i.e. the one containing the low 32 bits of the value).

2.2.2 Source modifier and immediate value: two's complement

The RR and RI commands both accept a signed integral operand, which cannot be straightforwardly represented using the basic types (because there are no signed integral basic types).

The signed integral values always have the two's complement representation (see Wiki for details). Such a representation is dependant on the bit size of the type that holds the value. Therefore, in the binary representation of a command, the same value may be represented differently if it is a source modifier versus an immediate value. That occurs iff the value is negative.

Before usage, both the source modifier and the immediate value are transformed to uint32 using the 32-bit two's complement representation for *the same value*. E.g. if a source modifier was -5 (written in 16-bit two's complement representation), it will be converted to *the same -5 value*, but in the 32-bit representation.

The two's complement representation of signed values is designed so that it produces intuitive additive operations results (see explanation). Therefore, when specifying the source modifier, one does not need to think about the signed integral values representation, and may simply assume that the *signed* source modifier value is added to the *unsigned* source register value in a purely mathematical sense of things, after which the resulting value is taken modulo 2^{32}.

Note, that the addition of the source modifier to the source register value always happens *before* the main operation (unless, of course, the command's documentation states that the source modifier is ignored). Therefore, all RI commands that do not ignore the source modifier effectively accept two uint32 operands.

2.3 Command set

2.3.1 System commands

Table 3: System commands

Code	Name	Format	Syntax sample	Description
0	halt	RI	`halt r1 0`	Stop processor
1	syscall	RI	`syscall r0 100`	System call

halt

The halt command sends an interruption signal to the CPU, which halts it until the next external signal is received, after which the execution is continued. The immediate value operand is ignored.

syscall

The syscall command's immediate operand specifies the call code. The semantics of those codes is described in Table 4:

Table 4: System call codes

Code	Name	Description	Register operand
0	EXIT	Finish execution with the code from the register	Source
100	SCANINT	Get a uint32 from stdin	Receiver
101	SCANDOUBLE	Get a double value from stdin	Low bits receiver
102	PRINTINT	Output a uint32 to stdout	Source
103	PRINTDOUBLE	Output double value to stdout	Low bits source
104	GETCHAR	Get a single ASCII character from stdin	Receiver
105	PUTCHAR	Output a single ASCII character from stdout	Source

Specifying a value for the syscall immediate operand not listed in the table above leads to an error during compilation or execution (depending on whether such a value was originally specified in an assembler code or in a handcrafted executable file).

If the source register for the PUTCHAR system call contains a value greater than 255 (that is, not representing an ASCII character), an execution error will occur.

2.3.2 Integer arithmetic operators

Table 5: Integer arithmetic operators

Code	Name	Format	Syntax sample	Description
2	`add`	RR	`add  r1 r2 -3`	$\texttt{r1} \mathrel{+}= \texttt{r2} - \texttt{3}$
3	`addi`	RI	`addi r4 10`	$\texttt{r4} \mathrel{+}= \texttt{10}$
4	`sub`	RR	`sub  r3 r5 5`	$\texttt{r3} \mathrel{-}= \texttt{r5} + \texttt{5}$
5	`subi`	RI	`subi r4 1`	$\texttt{r4} \mathrel{-}= \texttt{1}$
6	`mul`	RR	`mul  r3 r7 -2`	$(\texttt{r4}, \texttt{r3}) = \texttt{r3} * (\texttt{r7} - \texttt{2})$
7	`muli`	RI	`muli r5 100`	$(\texttt{r6}, \texttt{r5}) = \texttt{r5} * \texttt{100}$
8	`div`	RR	`div  r3 r8 5`	$\texttt{tmp} = (\texttt{r4}, \texttt{r3})$ $\texttt{r3} = \texttt{tmp / } (\texttt{r8} + \texttt{5})$ $\texttt{r4} = \texttt{tmp \% } (\texttt{r8} + \texttt{5})$
9	`divi`	RI	`divi r6 7`	$\texttt{tmp} = (\texttt{r7}, \texttt{r6})$ $\texttt{r6} = \texttt{tmp / 7}$ $\texttt{r7} = \texttt{tmp \% 7}$

`add`, `addi`, `sub`, `subi`

These commands take two `uint32` values obtained from the specified operands (see the respective section for details) and produce a `uint32` value as a result.

In the mathematical sense of things, for the unsigned (as-is) interpretation of the operands the arithmetic is simply performed modulo 2^{32}. E.g. if `r0` contained 0, after `subi r0 1`, the value in `r0` will be $2^{32} - 1$.

At the same time, if both terms of each operation were to be interpreted as *signed* integral values in the 32-bit two's complement representation, the result, if interpreted as a *signed* integral value in the same representation, would be mathematically correct. That is provided by the two's complement representation properties (see explanations for addition and subtraction).

`mul`, `muli`

These commands multiply two `uint32` values obtained from the specified operands (see the respective section for details), produce a `uint64` value and store it into the two registers starting from the specified receiver register (see here for details).

Unlike additive operations, these operations would not produce the correct results if the factors were to interpreted as *signed* integral values in the 32-bit two's complement representation. That is because in order to produce valid results in such a case, the factors must first be *extended* to the size of the destination type, i.e. rewritten in the 64-bit two's complement representation (see explanation). However, if we were to perform such an extension, we would block the opportunity to multiply two big *unsigned* values (because we would have interpreted them as *signed* values and added non-zero bits to the right of the original representation when extending, thus producing an incorrect result for this case).

Overall, one must be cautious when specifying the source modifier and the immediate value in these operations, and bear in mind that if those produce negative factors, they will firstly be converted to `uint32` by being taken modulo 2^{32}.

`div`, `divi`

These commands accept a `uint64` dividend from the two registers starting from the specified receiver register (see here for storage details) and a `uint32` divisor obtained in the common manner and perform an integral division. The quotient is then placed into the receiver register and the remainder – into the next register.

If the quotient does not fit into a register, a *quotient overflow* execution error occurs.

Like the `mul` and `muli` commands, these command only work with *unsigned* operands and would not produce valid results if the operands were to be interpreted as *signed* integral values.

2.3.3 Bitwise operators

Table 6: Bitwise operators

Code	Name	Format	Syntax sample	Description
10	`not`	RI	`not  r1 0`	$\mathtt{r1} = \sim \mathtt{r1}$
11	`shl`	RR	`shl  r1 r2 1`	$\mathtt{r1} \ll= \mathtt{r2} + \mathtt{1}$
12	`shli`	RI	`shli r1 2`	$\mathtt{r1} \ll= \mathtt{2}$
13	`shr`	RR	`shr  r1 r2 -4`	$\mathtt{r1} \gg= \mathtt{r2} - \mathtt{4}$
14	`shri`	RI	`shri r1 2`	$\mathtt{r1} \gg= \mathtt{2}$
15	`and`	RR	`and  r4 r6 3`	$\mathtt{r4} \mathrel{\&}= \mathtt{r6} + \mathtt{3}$
16	`andi`	RI	`andi r5 2`	$\mathtt{r5} \mathrel{\&}= \mathtt{2}$
17	`or`	RR	`or   r3 r2 -2`	$\mathtt{r3} \mathrel{\vert}= \mathtt{r2} - \mathtt{2}$
18	`ori`	RI	`ori  r6 100`	$\mathtt{r6} \mathrel{\vert}= \mathtt{100}$
19	`xor`	RR	`xor  r1 r5 0`	$\mathtt{r1} \mathrel{\hat{}}= \mathtt{r5}$
20	`xori`	RI	`xori r1 127`	$\mathtt{r1} \mathrel{\hat{}}= \mathtt{127}$

`not`

The immediate value is ignored, but per our agreement must be present for simplicity.

Other bitwise operators

All other bitwise operators take two `uint32` values obtained from the specified operands in the common manner and produce a `uint32` value as a result. The right hand side operand must be less than the size of a machine word (i.e. no more than 31), otherwise an execution error occurs.

2.3.4 Real-valued operators

Table 7: Real-valued operators

Code	Name	Format	Syntax sample	Description
21	`itod`	RR	`itod r2 r5 5`	$(\mathtt{r3}, \mathtt{r2}) = \mathtt{double}(\mathtt{r5} + \mathtt{5})$
22	`dtoi`	RR	`dtoi r2 r5 0`	$\mathtt{r2} = \mathtt{uint32}((\mathtt{r6}, \mathtt{r5}))$
23	`addd`	RR	`addd r2 r5 0`	$(\mathtt{r3}, \mathtt{r2}) \mathrel{+}= (\mathtt{r6}, \mathtt{r5})$
24	`subd`	RR	`subd r1 r6 0`	$(\mathtt{r2}, \mathtt{r1}) \mathrel{-}= (\mathtt{r7}, \mathtt{r6})$
25	`muld`	RR	`muld r0 r2 0`	$(\mathtt{r1}, \mathtt{r0}) \mathrel{*}= (\mathtt{r3}, \mathtt{r2})$
26	`divd`	RR	`divd r1 r3 0`	$(\mathtt{r2}, \mathtt{r1}) \mathrel{/}= (\mathtt{r4}, \mathtt{r3})$

For the `double` values storage please refer to this section. For the double values memory representation please refer to Wiki, specifically to the Overview and Internal representation sections.

`itod`

The source modifier is added to the source register in the common manner, after which the resulting `uint32` value is converted to `double` and stored into the two registers starting from the specified receiver register.

`dtoi`

The source modifier is ignored, but per our agreement must be present for simplicity.

The `double` value obtained from the two registers starting from the source register is converted to `uint32` by discarding the fractional part and stored into the receiver register. If the resulting value does not fit into a register, an execution error occurs.

`Real-valued arithmetic operators`

The source modifier is ignored for all real-valued arithmetic operators, but per our agreement must be present for simplicity.

2.3.5 Comparisons and jumps

Table 8: Comparisons and jumps

Code	Name	Format	Syntax sample	Description
27	`cmp`	RR	`cmp   r0  r1  2`	`r0` <=> `r1` + 2
28	`cmpi`	RI	`cmpi  r0  0`	`r0` <=> 0
29	`cmpd`	RR	`cmpd  r1  r4  0`	(`r2`, `r1`) <=> (`r5`, `r4`)
30	`jmp`	J	`jmp   0xffa`	Unconditional jump
31	`jne`	J	`jne   0xd471`	Jump if not equal
32	`jeq`	J	`jeq   05637`	Jump if equal
33	`jle`	J	`jle   1234`	Jump if less or equal
34	`jl`	J	`jl    0X1e4b`	Jump if less
35	`jge`	J	`jge   29834`	Jump if greater or equal
36	`jg`	J	`jg    03457`	Jump if greater

Flags register

To allow for basic execution branching, an additional `flags` register is supported. It holds the result of the latest comparison. Only the lowest 6 bits of this register are used. The semantics of those bits is described in Table 9.

Table 9: `flags` register bits semantics (low-to-high, 0-indexed)

0	Equal
1	Not equal
2	Greater
3	Less
4	Greater or equal
5	Less or equal

Several bits may be simultaneously filled. For example, if the latest comparison resulted in equality, the value of the `flags` register will be 110001_2, because equality causes the 'less or equal' and the 'greater or equal' conditions to also be true.

`cmp`, `cmpi`, `cmpd`

These are comparison operators. They compare the two values provided by the operands and store the result to the `flags` register according to the `flags` register bits semantics described above.

The `cmp` and `cmpi` command compare the `uint32` value from the receiver register to the `uint32` value obtained from the operands in the common manner.

The `cmpd` command ignores the immediate value operand and compares the two `double` values specified by the receiver and the source register operands. For the `double` values storage please refer to this section.

These are the only commands that store values in the `flags` register.

Jump commands

These commands implement execution branching. All of them, except for the `jmp` command, do nothing unless the condition mentioned in the table above is satisfied, that is, if the respective bit (according to the `flags` register bits semantics) is filled in the `flags` register. The `jmp` command is performed unconditionally.

If one of these commands take effect, the next executed instruction will be the one stored in the provided memory address operand (because the `Karma` computer has a Von Neumann architecture, the instructions' binaries are stored in the common address space). That is achieved by storing the provided memory address in the `r15` register (see the `Karma` calling convention section for details).

These are the only commands that read the value stored in the `flags` register.

2.3.6 Stack-related commands

Table 10: Stack operators and function calls

Code	Name	Format	Syntax sample	Description
37	`push`	RI	`push r0 255`	`*(r14 − −) = r0 + 255`
38	`pop`	RI	`pop  r3 3`	`r3 = *(+ + r14) + 3`

These commands work directly with the stack, putting `uint32` values into it (`push`) and extracting them afterwards (`pop`).

For the `push` command, the immediate value operand is added to the value from the specified source register before it is pushed to the stack. Similarly, for the `pop` command, the immediate value operand is added to the value retrieved (popped) from the stack before it is stored in the specified receiver register. The additions happen according to the common rules.

The `r14` register is the stack head pointer (as stated here). It always points to the memory cell, in which the data will be put on `push` operation, that is, one memory cell ‘ahead’ of the latest pushed value. The stack grows *backwards*, i.e. the next pushed value is stored in the *previous* memory cell from the latest one.

2.3.7 Data transfer commands

Table 11: Data transfer commands

Code	Name	Format	Syntax sample	Description
39	`lc`	RI	`lc    r7 123`	`r7 = 123`
40	`la`	RM	`la    r5 0xab1f`	`r5 = 0xab1f`
41	`mov`	RR	`mov   r0 r3 10`	`r0 = r3 + 10`
42	`load`	RM	`load    r3 0xffc2`	`r3 = *(0xffc2)`

43	`load2`	RM	`load2   r4 13224`	`r4 = *(13224)`
				`r5 = *(13225)`

44	`store`	RM	`store   r1 057123`	`*(057123) = r1`

45	`store2`	RM	`store2  r0 0X15fc`	`*(0X15fc) = r0`
				`*(0X15fd) = r1`

46	`loadr`	RR	`loadr   r8 r1 15`	`r8 = *(r1 + 15)`

47	`loadr2`	RR	`loadr2  r2 r0 12`	`r2 = *(r0 + 12)`
				`r3 = *(r0 + 13)`

48	`storer`	RR	`storer  r2 r3 3`	`*(r3 + 3) = r2`

49	`storer2`	RR	`storer2 r5 r2 13`	`*(r2 + 13) = r5`
				`*(r3 + 14) = r6`

`lc`, `la`, `mov`

These commands store the `uint32` value obtained from the operands in the common manner to the specified receiver register. The `la` command is similar to `lc`, but allows for labels usage (see Labels section for details).

Register-memory data transfer

All the other commands are used to transfer data between the registers and the memory.

Those of them which have the RR format treat the `uint32` value obtained from the operands in the common manner as a memory address. If this value does not represent a valid address, i.e. is greater or equal to 2^{20}, an execution error occurs. Note that the `storer` and the `storer2` commands treat the receiver register operand as the opposite, i.e. as the *source* for a value to be stored into the memory. These are the only two RR commands with such behaviour.

For the RM format commands the memory cell is specified by the 20-bit *unsigned* memory address operand. Since any 20-bit unsigned value represents a valid memory address, an execution error at this stage cannot occur in this case.

The commands with a ‘`2`’ suffix perform the same action as the respective commands without the suffix twice: once with the register and the memory cell specified by the operands and then with the next register and the next memory cell. Therefore, the specified register cannot be `r15` and the specified memory cell cannot be `0xfffff` (there are no next ones for them). If such a situation does occur, it results in an execution error.

2.4 Function calls

2.4.1 Introduction: calling conventions

This section defines the rules by which the execution of one part of the code (the *caller*) can be delayed until another part (the *callee*) is executed. The callee is often also referred to as a *subprogram* or a *function*, and the delay process is called a *function call*. The process of passing the execution from the callee back to the caller is often referred to as the *return from the function*.

For the functions to be meaningful they need to accept *arguments*, i.e. values passed from the caller to the callee for the latter to perform computations dependant on these values. The arguments of a function are often also referred to as *parameters*.

The functions often need to calculate some values for the caller to use later. In such cases these values are called the *return parameters* of the function and need to be somehow passed to the caller.

The function call procedure inevitably leads to a number of questions that need to be resolved, such as:

- Where the arguments are placed?
  Options include in the registers, on the stack, a mix of both, or in another memory structures
- In what order are the arguments passed?
  Options include first-to-last, last-to-first, or something more complex
- Whether and how metadata describing the arguments is passed?
  The metadata may include the number of arguments, the description of their position, or some other technical information
- How the functions that take a variable number of arguments (*variadic functions*) are handled?
  Options include passing the number of arguments as an argument itself placed in an obvious position, placing the variable arguments in an array in the memory and passing the pointer to it, or keeping track of the number of arguments in a separate register
- How return parameters are delivered from the callee to the caller?
  Options include in the registers, on the stack, a mix of both, or in another memory structures
- How the task of setting up for and cleaning up after a function call is divided between the caller and the callee?
  This question refers to the arguments passed to the caller, the caller's *local variables*, i.e. the values it places in the registers, on the stack and in the memory, and the additional values (if any) used internally for the function call itself
- Which registers are guaranteed to have the same value when the callee returns as they did when the callee was called?
  Such registers (if any) are called *saved* or *preserved* as opposed to the *volatile* registers, which are the rest of them. These registers usually have a utility purpose helping to resolve the other questions

The set of answers to these questions together with the description of the function call procedure is called the *calling convention*. There are a number of viable and widely used calling conventions. Each of them has its advantages and drawbacks, and the choice of the one most suitable for a specific case is often based on the computer architecture and the presumed programming process: whether the programs are written in the assembler language or some higher level languages to be later compiled into the assembler or the byte code.

Note that a calling convention is exactly that – a convention. Nothing on the architecture or the syntax level prevents one from not following it during their code development. However, that is considered a bad practice as it can lead to confusion and unexpected results for the code user. Besides, the assembler language often provides special commands that internally perform the utility actions defined by the convention. If one wishes to not follow the convention, all the actions must be performed manually for every function call.

Almost all calling conventions use the term *stack frame*, which is a part of the stack related to a specific function. The function's stack frame usually includes its arguments, the address of the caller instruction to be executed first after the return from the callee (the *return address)* and the function's local variables. It may or may not include additional utility purpose values.

There are two more terms used when describing a calling convention: the *prologue* and the *epilogue* of a function call.
The prologue is a set of internal technical instructions performed when calling a subprogram before the subprogram execution.
The epilogue is a set of such instructions performed when returning from a function before continuing the caller execution.

2.4.2 Karma calling convention

By the Karma calling convention, the arguments are passed to a function via the stack last-to-first, that is, the semantically first argument is the last one to be pushed to the stack. No additional arguments metadata is used. The variadic functions are handled using an additional utility register r13 (see below for details). The return parameters are delivered via the registers, starting from r0 first-to-last, that is, the first return parameter is placed in r0, the second one – in r1, etc.

The setup for and the cleanup after a function call is mainly performed in its prologue and epilogue, i.e. a programmer does not need to explicitly write their code. However, some actions still must be performed explicitly by the assembler code, that is an inevitable part of any calling convention.

The preserved registers are r13, r14 and r15. The semantics of the r14 register is described in the Stack-related commands section, and the usage of the r13 and r15 registers is described below.

r13 register

The r13 register typically contains the stack pointer (r14) value before the local variables of the currently executing subprogram. During the function calling process for a short time it changes its purpose and contains the stack pointer value before the arguments of the callee. At the start of execution it is initialised with the initial stack pointer value (defined in the executable file, see the respective section for details), which is the stack pointer value before the local variables of the main (initial) subprogram defined by the end directive.

r15 register

The r15 register *always* contains the address of the next instruction to be executed. When executing the binary code, the Karma executor consecutively checks the r15 register and executes the command, whose address is stored there. If at any point the r15 register is corrupted and does not store a valid memory address (i.e. if the value stored in the r15 register is greater or equal to 2^{20}), an execution error occurs.

2.4.3 Function call procedure

Before pushing the function arguments to the stack, the following preparatory actions must be performed:

- Push the current value of the r13 register (i.e. the stack pointer before the caller's local variables) to the stack
  As if with a `push r13 0`
- Set the r13 register to the current stack pointer value (i.e. the value before the function arguments)
  As if with a `mov r13 r14 0`

After that, the code must push the arguments for the function to be called to the stack last-to-first as described above.

The prologue of a function call does the following

- Pushes the return address (i.e. the address of the next caller instruction, which is stored in the r15 register) to the stack
  As if with a `push r15 0`
- Pushes the current value of the r13 register (i.e. the value explicitly stored by the assembler code, which is the stack pointer value before the subprogram arguments) to the stack
  As if with a `push r13 0`
- Sets the r13 register to the current stack pointer value (i.e. the value before the function local variables)
  As if with a `mov r13 r14 0`

2.4.4 Return from a function procedure

Neither the callee nor the caller need to perform any explicit actions on a return from a function. Everything is done automatically by the epilogue. The epilogue undoes the actions performed during the function call procedure. Specifically, it does the following:

- Sets the stack pointer to the current value of the r13 register (i.e. removes the local variables of the callee from the stack)
  As if with a `mov r14 r13 0`
- Pops a value from the stack to the r13 register (that is the value of the stack pointer before the function arguments)
  As if with a `pop r13 0`
- Pops a value from the stack to the r15 register (that is the address of the next caller's instruction to be executed)
  As if with a `pop r15 0`
- Sets the stack pointer to the current value of the r13 register (i.e. removes the function arguments from the stack)
  As if with a `mov r14 r13 0`
- Pops a value from the stack to the r13 register (that is the value of the stack pointer before the caller's local variables)
  As if with a `pop r13 0`

2.4.5 Function call commands

Table 12: Function call comands

Code	Name	Format	Syntax sample	Description
50	`prc`	J	`prc 0`	Prepare for a function call
50	`call`	RR	`call r0 r5 2`	Call the function located at the address `r5` + `2`
51	`calli`	J	`calli 21913`	Call the function located at the address `21913`
52	`ret`	J	`ret 0`	Return from the callee

These commands implement the `Karma` calling convention and encourage the convention usage.

They do not provide any functionality that cannot be achieved using other instructions (see the Function call procedure and the Return from a function procedure sections above for the explicit sets of commands that perform the same actions).

Therefore, as described in the Introduction: calling conventions section, the `Karma` calling convention does not *enforce* anything. One may choose to follow a different convention in their code, but then they will be unable to use these commands and must perform all the utility actions required by their convention manually.

`prc`

This command must be *explicitly* executed before pushing the function arguments to the stack. It performs the preparatory actions described above.

This is the only additional (to the pushing the function arguments to the stack) explicit action required for the function call. The rest of utility actions are performed automatically by the `call`/`calli` and the `ret` commands.

After this command and before the subsequent `call` or `calli` command all values pushed to the stack are considered the function arguments.

The memory address operand is ignored, but per our convention must be present for simplicity.

`call`, `calli`

These commands perform the transfer of control from the caller to the callee by executing the function call prologue and storing the address of the next instruction to be executed (i.e. the first instruction of the callee) to `r15`. The `call` command additionally stores a copy of the return address into the provided receiver register.

For the `call` command the address of the next instruction to be executed is obtained from the specified operands in the common manner. If this value does not represent a valid address, i.e. is greater or equal to 2^{20}, an execution error occurs.

For the `calli` command the address of the next instruction to be executed is specified by the 20-bit *unsigned* memory address operand. Since any 20-bit unsigned value represents a valid memory address, no execution error can occur in this case.

Note that since the function call prologue pushes the values from the `r15` and the `r13` registers to the stack, after the prologue execution, the arguments start from the third latest value in the stack. For example the first argument can be stored to the `r0` register via a `loadr r0 r14 3` command (note the non-zero source modifier operand).

`ret`

This command performs the transfer of control from the callee back to the caller by executing the function call epilogue, after which the `r15` register contains the return address.

The memory address operand is ignored, but per our convention must be present for simplicity.

2.4.6 Notes and other calling conventions

The suggested calling convention has two drawbacks.

The main one is that a programmer must explicitly use the prc command before pushing the function's arguments to the stack (see above for details). However, the fact that *some* instructions must be performed explicitly is an inherent part of any calling convention, and the fact that the Karma calling convention has but one necessary additional instruction is a good quality.

Another one is that the r13 register does not have one consistent semantics. Instead, it has two semantics (see above for details), one of which is only used during the function call procedure. This also leads to the need to push its value to the stack twice during the function call procedure (see above for details).

These drawbacks, however, allow for a major advantage of the Karma calling convention. The advantage is that there is no need for the programmer to manually clear the stack from the function arguments or local variables. It is done automatically by the epilogue (see above for details). Neither is there a need for any argument metadata, which optimises the stack usage.

Other calling conventions may manage the removal of the function arguments and local variables from the stack in one of the following manners:

- Passing the responsibility of removing the function arguments to the caller, which must clear the stack after the return from the function by explicitly moving the stack pointer. In this case the r13 register is usually solely used for automatic removal of the local variables.
- Interpreting the memory address operand of the ret command as the number of arguments to be removed from the stack. Like the previous case, in this one the r13 register is also usually solely used for automatic removal of the local variables. However, this approach has a problem with variadic functions, because when writing a variadic function one does not know the exact number of arguments that will be passed to it to pass this number as the operand to the ret command. One of the possible solutions of this issue may be to pass the number of arguments as the first parameter of all variadic functions and to introduce a special value for the ret operand (e.g. -1) to be interpreted as an instruction to treat the first argument as the number of arguments and increment the stack pointer by the respective value.
- Passing the responsibility of removing the function local variables to the callee, which must clear the stack before the return by explicitly moving the stack pointer. In this case the r13 register is usually solely used for automatic removal of the function arguments.

These other calling conventions may sometimes optimise the stack usage or the function call procedure performance, but are mainly meant for the cases when the programmer writes the code in some higher level languages to be then compiled to assembler. In such a case, the compiler can substitute the correct value to increment the stack pointer by knowing at compile time the number of function arguments and local variables. In our case, however, when the code is written directly in assembler, such an approach is bugprone because a single invalid stack pointer increment will lead to the stack corruption and unexpected behaviour.

2.5 Constants

One can define a constant of any basic type as well as of two additional types: `char` and `string`.

Syntax sample: `uint32 123`

For consistent disassembling purposes any constant's memory representation is preceded with a one-word unique type identifier.

`uint32`, `uint64`

These values may be represented in all the same ways as command operands.

The negative values are processed in the common manner, i.e. using the two's complement representation for the bit size of the respective type. The values of `uint32` type constants which are greater or equal to 2^{32}, but are less than 2^{64}, are taken modulo 2^{32}. If the specified value does not fit into `uint64`, a compile error occurs.

For storage details see the respective section.

`double`

A `double` constant may be represented either in a decimal or a hexadecimal formats. Both these formats are described with the std::stod function of the C++ standard.

For storage details see the respective section.

`char`

A `char` constant value must be surrounded with single quotes.

There are a number of *escape sequences* representing certain special `ASCII` characters. One of those is the '\#' escape sequence, used to represent a single '#' character (the *hash sign*, `ASCII` code 35). This escape sequence allows to resolve the confusion between a hash sign at the start of a comment (see Comments section for details). The rest of the supported escape sequences are listed in the Simple escape sequences table of the C++ standard.

A `char` constant value is stored as a `uint32` from 0 to 255 inclusively in `ASCII` encoding.

`string`

A `string` constant value must be surrounded with double quotes. Such a value represents a sequence of `char` values. In particular, that means that all the same escape sequences are used.

A `string` value is represented in memory as the respective `char` values stored in consecutive memory cells followed by an additional *zero character* (`ASCII` code 0). The zero character indicates the end of a `string`.

2.6 Labels

Before a command or a constant either on the same or on a separate line one may place a *label*, which can be used later on in the assembler code to indicate the memory address of the command or constant it is placed before.

Syntax:

- A label must consist only of lowercase latin letters, digits, underscores and dots
- A label must not start with a digit
- A label must be followed by a colon
- A label must be the first word in its line (it may be the only word of the line)
- A label must not conflict with predefined words (i.e. command names, directives, etc.)
- The next word after the label must be a command name or a constant type
  In particular, that means that there cannot be two consecutive labels pointing to the same command
- The labels must be unique (i.e. label redefinition is not allowed)

Usage:

- A label usage may precede its definition
- A label must be defined somewhere in the code to be used, there are no predefined labels
- A label may only be used as a memory address, i.e. only as an operand of a command of either RM or J type
  Note: this means that, when used, a label is always the last word in its line (see Command formats section)

2.7 Directives

2.7.1 include directive

An assembler file may have several include directives at the beginning. An include directive must be followed by an unquoted string representing a relative (to the original file) path to another assembler file. All include directives must appear before the code. If an include directive is encountered after the first command (or label, or constant, whichever occurs first) of the file or if the relative path specified by the directive is invalid for any reason (no such file exists, the file cannot be opened, etc.), a compilation error occurs.

The directed graph in which the nodes represent the files, and the directed edges represent the inclusions, drawn from the file in which the include directive occurred to the file specified by said directive, is called the *inclusion tree* of the program. The program is then said to *consist* of the files represented by the nodes of its inclusion tree.

A *post-order traversal* of a graph is such a traversal of its nodes in which the parent node is processed only after all the child nodes have been processed. Encountering a node is called *entering* it, if a node has been encountered, it is said to be *entered*. After all its children have been processed and the algorithm returned to its parent, the node is said to be *exited*.

The Karma compiler uses a *single-entering policy*, which means that during the traversal each node is only entered once, i.e. if the node was already encountered and at the same time is listed as the child node of the currently entered, but not exited node, it is simply skipped. Such a situation can occur either if the *root* (i.e. the node from which the traversal started) is in a cycle or if a node has several *parents* (i.e. nodes from which there is an edge leading to the current node). Both these cases are successfully resolved (i.e. prevented from causing an endless loop) by the single-entering policy.

The *order of inclusion* is the order in which the files would be visited in a post-order traversal of the inclusion tree with the single-entering policy starting from the root Karma assembler file (i.e. the one passed to the compiler).

If include directives are present in a program, the files specified by them are compiled with the code from the root file as if their contents were copied to the beginning of the root file in the order of inclusion. In particular, that means that the code from the root file may use the labels defined in the included files. Therefore, the labels must be unique throughout all the included files as well as the original file.

Note that due to the single-entering policy described above, a valid Karma assembler program's inclusion tree may not actually be a tree in the graph theory sense, i.e. it may contain cycles or a file may be included from several other files. However, if all includes are *substantial* (i.e. if a file is included only if some labels from it are used in the current file), the cycles are impossible.

2.7.2 end directive

An assembler program must have *exactly one* end directive.

It has one operand which indicates the address of the first instruction (or a label).

2.8 Comments

Each line may contain a hash sign (#). If it does, everything after the hash sign is considered a comment and is not compiled into the executable file. Multiline comments are not allowed.

Note that if hash sign is preceded with a backslash (\) it is not considered a comment start. This allows for using a hash sign in string or char constants (see Constants section for details).

2.9 Notes

- The Karma processor has a RISC architecture, which means that there is no way to operate directly on memory cells, all data has to be loaded to the registers before modifications and the results have to be explicitly stored back to the memory if necessary (see Data transfer commands for details)
- A Von Neumann architecture of the Karma computer implies that both the machine code of the program and the stack are inside the global address space. The former is placed at its beginning, and the latter starts at its end and grows backwards
- By default, the stack is *unbounded*, i.e. it does not have any size limits besides the address space size
- By default, our system allows to write data to any memory cells, including the ones occupied by the machine code itself. Therefore, theoretically, a program might overwrite itself during runtime, although such behaviour is considered a bad practice
- The same applies to the registers: by default, any register (except for the flags register) may be passed as an operand to any command (that accepts a register operand). That includes the r13–r15 registers, which are meant for utility purposes. Manually modifying the values of these registers, while allowed, is considered a bad practice and can lead to unexpected results

3 **Karma** executable file

To run a program on a Karma computer one needs to generate an executable file which contains meta-information about the machine code and the code itself. The executable file is stored in a remote storage (e.g. a hard drive or an SSD) as a byte sequence. The header of the executable file takes up exactly 512 bytes. The format of the executable file is described in Table 5.

Table 13: Karma executable file format

Bytes	Contents
0..15	ASCII string “ThisIsKarmaExec”
16..19	Program code size
20..23	Program constants size
24..27	Address of the first instruction
28..31	Initial stack pointer value
32..35	ID of the target processor
512..	Code segment
	Constants segment

Notes:

- The ASCII string at the beginning of the executable file contains 15 explicit characters and an implicit ‘\0’ at the end
- The code, constants and data segments sizes are denoted in bytes (as opposed to in machine words)
- The execution of the program starts from the instruction whose address is specified in the executable file header
  This is the initial value of the r15 register
- The header also specifies the initial stack head address
  This is the initial value of the r14 register
- A Karma executable file can run on only one processor – that with ID 239
- The code, constants and data segments are loaded into the Karma computer consecutively, starting from the very first memory cell (i.e. the one with the 0 address)

4 Execution configuration

A Karma computer as a whole as well as any specific execution has a configuration.

The configuration allows to:

- block the access to some registers
- block the access to the code segment of the memory
- block the access to the constants segment of the memory
- *bound* the stack, i.e. specify the maximum stack size, so that if the stack grows beyond it, a *stack overflow* error occurs

The access to both the registers and the code and constants segments of the memory can be blocked either for writing (*write block*) or for both reading and writing (*read-write block*). The former raises an execution error in case a program tries to *manually* modify the contents of the blocked register/data segment, but allows for the program to read the values from the register/data segment. The latter raises an execution error in case a program tries to *manually* either modify or read the contents of the blocked register/data segment.

Note that regardless of whether such blocks are put in place, the r13–r15 registers will still be available for *internal usage*, that is, via the jumps, the stack-related and the function call commands. The r15 register will also be available for the Karma computer internal machinery to maintain the program execution (see the r15 register section for details).

The configuration for a specific execution is the *strictest* combination of its own configuration and the common Karma computer configuration. That is, if a register or a memory segment is specified to be blocked in either one of them, it is blocked for the execution. Similarly, if the stack is specified to be bounded in either one of them, it is bounded for the execution. If it is specified to be bounded by both configurations, its resulting maximum size is the minimal one of the specified sizes.

As stated above, by default, no registers or memory segments are blocked, and the stack is unbounded.

5 Code samples

5.1 Hello world!

```
hello_world:                    # define the string to print
    string “Hello, world!\n”

main:
    la r0 hello_world           # load the address of the string to r0
    loop:                       # a while loop
        cmpi r1 0               # check if the current character is ‘\0’
        jeq out                 # if we have encountered the ‘\0’ character, break the cycle
        syscall r1 105          # print the current character from r0 to stdout
        addi r0 1               # proceed to the next character
        jmp loop                # continue the loop
    out:                        # out of the while loop
        lc r0 0                 # set return code to 0
        syscall r0 0            # exit the program with code 0 from r0
end main                        # start execution from label main
```

5.2 Calculate the square of a number without functions

```
main:
    syscall r0 100              # read an integer from stdin to r0
    mov r2 r0 0                 # copy from r0 to r2
    mul r0 r2 0                 # a pair of registers (r0,r1) contains the square
    syscall r0 102              # print from r0 to stdout (i.e. the lower bits)
    lc r0 10                    # store the constant 10 (‘\n’) to r0
    syscall r0 105              # print ‘\n’ from r0 to stdout
        lc r0 0                 # set return code to 0
        syscall r0 0            # exit the program with code 0 from r0
end main                        # start execution from label main
```

5.3 Calculate the square of a number with functions

```
sqr:                            # a function calculating the square with one argument on the stack
    loadr r0 r14 3              # load the first (and only) argument to r0
    mov r2 r0 0                 # copy from r0 to r2
    mul r0 r2 0                 # a pair of registers (r0,r1) contains the square
    ret 0                       # return from function and remove the argument from the stack

intout:                         # a function printing its argument and ‘\n’
    loadr r0 r14 3              # load the first (and only) argument to r0
    syscall r0 102              # print r0 to stdout
    lc r0 10                    # store the constant 10 (‘\n’) to r0
    syscall r0 105              # print ‘\n’ from r0 to stdout
    ret 0                       # return from function and remove the argument from the stack

main:
    syscall r0 100              # read an integer from stdin to r0
    prc 0                       # prepare for the function call
    push r0 0                   # put r0+0 to the stack as the sqr argument
    calli sqr                   # call sqr, the function will put the result to r0
    push r0 0                   # prepare the result of sqr to be passed to intout
    calli intout                # call intout with the prepared argument
    lc r0 0                     # set return code to 0
    syscall r0 0                # exit the program with code 0 from r0
end main                        # start execution from label main
```

5.4 Calculate the factorial of a number using a loop

```
fact:                          # a non-recursive function calculating the factorial of its argument
    loadr r0 r14 3             # load the first (and only) argument to r0
    mov r2 r0 0                # copy the argument to r2
    lc r0 1                    # initialise the result with 1
    loop:                      # a while loop
        cmpi r2 1              # compare r2 to 1
        jle out                # if the next factor is less or equal to 1, break the cycle
        mul r0 r2 0            # multiply the current result by the next factor
        subi r2 1              # decrement r2 by 1
        jmp loop               # continue the loop
    out:                       # out of the while loop
        ret 0                  # return from function and remove the argument from the stack

main:
    syscall r0 100             # read an integer from stdin to r0
    prc 0                      # prepare for the function call
    push r0 0                  # put r0+0 to the stack as the fact argument
    calli fact                 # call fact, the function will put the result to r0
    syscall r0 102             # print r0 to stdout
    lc r0 10                   # store the constant 10 (‘\n’) to r0
    syscall r0 105             # print ‘\n’ from r0 to stdout
    lc r0 0                    # set return code to 0
    syscall r0 0               # exit the program with code 0 from r0
end main                       # start execution from label main
```

5.5 Calculate the factorial of a number using recursion

```
fact:                          # a recursive function calculating the factorial of its argument
    loadr r0 r14 3             # load the first (and only) argument to r0
    cmpi r0 1                  # compare r0 to 1
    jg skip0                   # if the argument is greater that 1, recurse
    lc r0 1                    # else store 1 (the result for this case, 1! = 1) to r0
    ret 0                      # return from function and remove the argument from the stack

skip0:                         # a supplemental function providing recursion
    push r0 0                  # push the current value to the stack (⋆)
    subi r0 1                  # decrement the current value by 1
    prc 0                      # prepare for the function call
    push r0 0                  # push the decremented value to stack as the fact argument
    calli fact                 # r0 contains the result for the decremented value
    pop r2 0                   # pop the value stored during the (⋆) push to r2
    mul r0 r2 0                # multiply the result for the decremented value by the current value
    ret 0                      # return from function and remove the argument from the stack

main:
    syscall r0 100             # read an integer from stdin to r0
    prc 0                      # prepare for the function call
    push r0 0                  # put r0+0 to the stack as the fact argument
    calli fact                 # call fact, the function will put the result to r0
    syscall r0 102             # print r0 to stdout
    lc r0 10                   # store the constant 10 (‘\n’) to r0
    syscall r0 105             # print ‘\n’ from r0 to stdout
    lc r0 0                    # set return code to 0
    syscall r0 0               # exit the program with code 0 from r0
end main                       # start execution from label main
```