

Karma Computer

Tigran Koshkelian

Contents

1	Architecture	2
2	Karma processor description	3
2.1	Karma processor command set	3
2.2	Further specifications	9
2.2.1	Floating-point values	9
2.2.2	System calls	9
2.2.3	Flags	9
2.2.4	Labels	10
2.2.5	End directive	10
2.2.6	Comments	10
2.3	Notes	10
3	Karma executable file	11
4	Code samples	12
4.1	Calculate the square of a number without functions	12
4.2	Calculate the square of a number with functions	12
4.3	Calculate the factorial of a number using a loop	13
4.4	Calculate the factorial of a number using recursion	13

1 Architecture

Karma is a computer with a Von Neumann architecture with an address space of 2^{20} words, each of which takes up 32 bits.

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.

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

<code>r0-r12</code>	Free usage
<code>r13</code>	Call frame pointer
<code>r14</code>	Stack pointer
<code>r15</code>	Instruction pointer
<code>flags</code>	Comparison operation result

With respect to the operation code each command may be of one of the following formats:

- RM (register-memory):
 - 8 bits of operation code
 - 4 bits of register number (either source or receiver)
 - 20 bits of memory address (an unsigned number from 0 to $2^{20} - 1$)

Example: `load r0, 12323`
- RR (register-register):
 - 8 bits of operation code
 - 4 bits of receiver register number
 - 4 bits of source register number
 - 16 bits of source modifier aka immediate operand (a signed number from -2^{15} to $2^{15} - 1$)

Example: `mov r1, r2, -123`

- RI (register-immediate value):
 - 8 bits of operation code
 - 4 bits of receiver register number
 - 20 bits of immediate operand (a signed number from -2^{19} to $2^{19} - 1$)

Example: `ori r2, 64`

- J (jump):
 - 8 bits of operation code
 - 4 bits ignored
 - 20 bits of memory address (an unsigned number from 0 to $2^{20} - 1$)

Example: `calli 3121`

For the sake of not overcomplicating matters all arguments of any command are required. If one does not need the source modifier in a command of the RR format, the immediate operand should be specified as 0.

2 Karma processor description

2.1 Karma processor command set

Table 2: Karma processor commands description

Code	Name	Format	Description
0	halt	RI	Stop processor halt r1 0
1	syscall	RI	System call (see here for details) syscall r0, 100
2	add	RR	Registers addition The value in the source register modified by the immediate operand is added to the value in the receiver register add r1, r2, 3 In the example above r2+3 is added to r1
3	addi	RI	Immediate operand addition to register The immediate operand is added to the value in the receiver register addi r4, 10
4	sub	RR	Registers subtraction The value in the source register modified by the immediate operand is subtracted from the value in the receiver register sub r3, r5, 5 In the example above r5+5 is subtracted from r3
5	subi	RI	Immediate operand subtraction from register The immediate operand is subtracted from the value in the receiver register subi r4, 1
6	mul	RR	Registers multiplication The value in the receiver register is multiplied by the value in the source register modified by the immediate operand The result is placed in a pair of registers starting from the receiver so that the receiver contains the lower 32 bits of the result mul r3, r10, 2 In the example above r3 is multiplied by r10+2 and the result is placed in r3 (low 32 bits) and r4 (high 32 bits)
7	muli	RI	Register multiplication by immediate operand The value in the receiver register is multiplied by the immediate operand The result is placed in a pair of registers starting from the receiver so that the receiver contains the lower 32 bits of the result muli r5, 100 In the example above the low 32 bits of the result will be in r5 and the high 32 bits – in r6
8	div	RR	Registers division The low 32 bits of the dividend are located in the receiver register, the high 32 bits – in the next register. The divisor is located in the source register and is modified by the immediate operand The quotient is placed in the receiver register (if it does not fit, a ‘quotient overflow’ runtime error occurs) and the remainder – in the next register div r3, r10, 5 After execution of the example above r3 will contain the quotient of dividing (r3, r4) by r10+5 and r4 will contain the remainder
9	divi	RI	Register division by immediate operand The dividend and the result locations are analogous to div operation divi r3, 10 After execution of the example above r3 will contain the quotient of dividing (r3, r4) by 10 and r4 will contain the remainder

12	lc	RI	Storing immediate operand to the register Supplemental bitwise shift and addition commands are required for storing constants greater than $2^{20} - 1$ lc r7, 123
13	shl	RR	Bitwise left-shift of the value in the receiver register by the value in the source register modified by the immediate operand shl r1, r2, 1
14	shli	RI	Bitwise left-shift of the value in the receiver register by the immediate operand shli r1, 2
15	shr	RR	Bitwise right-shift of the value in the receiver register by the value in the source register modified by the immediate operand shr r1, r2, 1
16	shri	RI	Bitwise right-shift of the value in the receiver register by the immediate operand shri r1, 2
17	and	RR	Bitwise and of the value in the receiver register by the value in the source register modified by the immediate operand and r4, r6, 5
18	andi	RI	Bitwise and of the value in the receiver register by the immediate operand andi r5, 2
19	or	RR	Bitwise or of the value in the receiver register by the value in the source register modified by the immediate operand or r3, r2, 2
20	ori	RI	Bitwise or of the value in the receiver register by the immediate operand ori r6, 100
21	xor	RR	Bitwise xor of the value in the receiver register by the value in the source register modified by the immediate operand xor r1, r5, 0
22	xori	RI	Bitwise xor of the value in the receiver register by the immediate operand xori r1, 127
23	not	RI	Bitwise not of the value in the receiver register The immediate value is ignored, but per our agreement must be present for the simplicity of the compiler not r1, 0
24	mov	RR	Forwarding the value in the source register modified by the immediate operand to the receiver register mov r0, r3, 10 After the execution of the example above r3+10 is stored in r0

32	addd	RR	<p>Real-valued registers addition</p> <p>The floating-point values are stored in two registers, the provided registers contain the lower bits of the values (see here for details)</p> <p>The immediate operand modifies the lower bits of the floating-point representation of the source value, not the value itself, so it should be used with much caution</p> <pre>addd r2, r5, 0</pre> <p>In the example above a floating-point value stored in (r5, r6) is added to the one stored in (r2, r3)</p>
33	subd	RR	<p>Real-valued registers subtraction</p> <p>For real value storage and immediate operand comments see add</p> <pre>subd r1, r6, 0</pre> <p>In the example above a floating-point value stored in (r6, r7) is subtracted from the one stored in (r1, r2)</p>
34	muld	RR	<p>Real-valued registers multiplication</p> <p>For real value storage and immediate operand comments see add</p> <pre>muld r0, r2, 0</pre> <p>In the example above a floating-point value stored in (r0, r1) is multiplied by the one stored in (r2, r3)</p>
35	divd	RR	<p>Real-valued registers division</p> <p>For real value storage and immediate operand comments see add</p> <pre>divd r1, r3, 0</pre> <p>In the example above a floating-point value stored in (r1, r2) is divided by the one stored in (r3, r4)</p>
36	itod	RR	<p>Integer to floating-point transformation</p> <p>The value in the source register modified by the immediate operand is interpreted as an integer, transformed to a floating-point value and stored in two registers starting from the receiver so that the receiver contains the lower bits of the result</p> <pre>itod r2, r5, 5</pre> <p>In the example above the floating-point representation of value r5+5 is stored to (r2, r3) with r2 containing the low 32 bits of the result</p>
37	dtoid	RR	<p>Floating-point to integer transformation</p> <p>The real value specified by the source register is rounded down to the closest integer</p> <p>For real value storage and immediate operand comments see add</p> <p>If the resulting value does not fit a register, an error occurs</p> <pre>dtoid r2, r5, 0</pre> <p>In the example above the floating-point value stored in (r5, r6) is rounded down and stored in r2</p>

38	push	RI	<p>Push the value from the source register modified by the immediate operand to the stack (and then move the stack pointer)</p> <p>push r0, 255</p> <p>In the example above the value r0+255 is stored to the address from r14, after which the stack pointer (r14) is decremented by 1</p>
39	pop	RI	<p>Pop the value from the stack and store it in the receiver register after modifying by the immediate operand (after moving the stack pointer)</p> <p>pop r3, 3</p> <p>In the example above the stack pointer (r14) is incremented by 1, after which the value stored by the address from r14 is incremented by 3 and stored in r3</p>
40	call	RR	<p>Call the function, the address of which can be acquired by modifying the source register by the immediate operand</p> <p>The address of the command following the current one is stored in the receiver register</p> <p>call r0, r5, 2</p> <p>In the example above the function stored by the address r5+2 is called and the address of the command following the current one is both pushed to the stack (i.e. stored by the address from r14 with a consequent decrement of r14) and stored in r0</p>
41	calli	J	<p>Call the function, the address of which is specified by the immediate operand</p> <p>calli 13323</p> <p>In the example above the function stored by the address 13323 is called and the address of the command following the current one is pushed to the stack (i.e. stored by the address from r14 with a consequent decrement of r14)</p>
42	ret	J	<p>Return from function to caller</p> <p>The address of the next executed instruction is popped from the stack</p> <p>The immediate operand specifies the number of additional words that should be ejected from the stack (by simply incrementing the r14 pointer) before popping the next executed instruction address (it must equal the number of the function arguments)</p> <p>ret 3</p> <p>In the example above the pointer from r14 is incremented by $3 + 1 = 4$, after which the next executed instruction address is acquired by the address from r14</p>
43	cmp	RR	<p>Registers comparison</p> <p>The value in the receiver register is compared to the value in the source pointer modified by the immediate operand and the result is stored to the flags register</p> <p>cmp r0, r1, 2</p> <p>In the example above r0 is compared to r1+2</p>
44	cmpi	RI	<p>Comparison with immediate operand</p> <p>The value in the specified register is compared to the immediate operand and the result is stored to the flags register</p> <p>cmpi r0, 0</p>
45	cmpd	RR	<p>Real-valued registers comparison</p> <p>For real value storage and immediate operand comments addd</p> <p>The floating-point value specified by the receiver register is compared to the one specified by the source register and the result is stored to the flags register</p> <p>cmpd r1, r4, 0</p> <p>In the example above the floating-point value stored in (r1, r2) is compared to the one stored in (r4, r5)</p>

46	jmp	J	<p>Unconditional jump</p> <p>The address of the next executed instruction is specified by the immediate operand</p> <p>jmp 2212</p>
47	jne	J	<p>Jump if not equal</p> <p>The jump only occurs if the <code>flags</code> register contains the ‘not equal’ condition, else the execution continues (see here for details)</p> <p>The address of the next executed instruction in case the actual jump occurs is specified by the immediate operand</p> <p>jne 2212</p>
48	jeq	J	<p>Jump if equal</p> <p>The jump only occurs if the <code>flags</code> register contains the ‘equal’ condition, else the execution continues (see here for details)</p> <p>The address of the next executed instruction in case the actual jump occurs is specified by the immediate operand</p> <p>jeq 2212</p>
49	jle	J	<p>Jump if less or equal</p> <p>The jump only occurs if the <code>flags</code> register contains the ‘less or equal’ condition, else the execution continues (see here for details)</p> <p>The address of the next executed instruction in case the actual jump occurs is specified by the immediate operand</p> <p>jle 2212</p>
50	j1	J	<p>Jump if less</p> <p>The jump only occurs if the <code>flags</code> register contains the ‘less’ condition, else the execution continues (see here for details)</p> <p>The address of the next executed instruction in case the actual jump occurs is specified by the immediate operand</p> <p>j1 2212</p>
51	jge	J	<p>Jump if greater or equal</p> <p>The jump only occurs if the <code>flags</code> register contains the ‘greater or equal’ condition, else the execution continues (see here for details)</p> <p>The address of the next executed instruction in case the actual jump occurs is specified by the immediate operand</p> <p>jge 2212</p>
52	jg	J	<p>Jump if greater</p> <p>The jump only occurs if the <code>flags</code> register contains the ‘greater’ condition, else the execution continues (see here for details)</p> <p>The address of the next executed instruction in case the actual jump occurs is specified by the immediate operand</p> <p>jg 2212</p>

64	load	RM	<p>Load from memory to register</p> <p>The value stored by the address specified by the immediate operand is copied to the receiver register</p> <p>load r0, 12345</p>
65	store	RM	<p>Store from register to memory</p> <p>The value stored in the source register is copied to the address specified by the immediate operand</p> <p>store r0, 12344</p>
66	load2	RM	<p>Load two words from memory to registers</p> <p>The value stored by the address specified by the immediate operand and the next memory cell is copied to the receiver register and the next register respectively</p> <p>load2 r0, 12345</p> <p>In the example above the values from the memory cells 12345 and 12346 are copied to registers r0 and r1 respectively</p>
67	store2	RM	<p>Store two words from registers to memory</p> <p>The value stored in the source register and the next register are copied to the address specified by the immediate operand and the next memory cell respectively</p> <p>store2 r0, 12344</p> <p>In the example above the values from registers r0 and r1 are copied to the memory cells 12344 and 12345 respectively</p>
68	loadr	RR	<p>Load from memory to register</p> <p>The value stored by the address which can be acquired by modifying the source register by the immediate operand is copied to the receiver register</p> <p>loadr r0, r1, 15</p> <p>In the example above the value from the memory cell r1+15 is copied to r0</p>
69	storer	RR	<p>Store from register to memory</p> <p>The value stored in the receiver register is copied to the address which can be acquired by modifying the source register by the immediate operand</p> <p>The naming of the argument registers for this command is counter-intuitive: the value is copied <i>from the receiver register</i></p> <p>storer r0, r11, 3</p> <p>In the example above the value from r0 is copied to the memory cell r11+3</p>
70	loadr2	RR	<p>Load two words from memory to registers</p> <p>The value stored by the address which can be acquired by modifying the source register by the immediate operand and the next memory cell is copied to the receiver register and the next register respectively</p> <p>loadr2 r0, r10, 12</p> <p>In the example above the values from the memory cells r10+12 and r10+13 are copied to registers r0 and r1 respectively</p>
71	storer2	RR	<p>Store two words from registers to memory</p> <p>The value stored in the receiver register and the next register are copied to the address which can be acquired by modifying the source register by the immediate operand and the next memory cell respectively</p> <p>The naming of the argument registers for this command is counter-intuitive: the value is copied <i>from the receiver register</i></p> <p>storer2 r0, r3, 10</p> <p>In the example above the values from registers r0 and r1 are copied to the memory cells r3+10 and r3+11 respectively</p>

2.2 Further specifications

2.2.1 Floating-point values

Floating-point values are represented in base-2 scientific notation, i.e. in the form $m \cdot 2^n$, where $m \in [1, 2)$ is called a *mantissa* and $n \in \mathbb{Z}$ – an *exponent*.

In memory they have a 64-bit representation. The meaning of those bits (high to low) is as follows:

- 1 bit – sign (0 means +, 1 means –)
- 11 bits – the exponent incremented by 1023
- 52 bits – the fractional part of the mantissa

2.2.2 System calls

The `syscall` operation has an immediate operand which specifies the call code. The semantics of those codes is described in [Table 3](#).

Table 3: System call codes

Code	Name	Description	Register operand
0	EXIT	Finish execution without error	–
100	SCANINT	Get an integer value from <code>stdin</code>	Receiver
101	SCANDOUBLE	Get a floating-point value from <code>stdin</code>	Low-bits receiver
102	PRINTINT	Output an integer value to <code>stdout</code>	Source
103	PRINTDOUBLE	Output a floating-point value to <code>stdout</code>	Low-bits source
104	GETCHAR	Get a single ASCII character from <code>stdin</code>	Receiver
105	PUTCHAR	Output a single ASCII character from <code>stdout</code>	Source

Notes:

- The floating-point value storage convention is the same as for the `addd` command, i.e. the specified register holds the lower bits of the value, and the next register holds the higher bits.
- If the register provided for the `PUTCHAR` system call holds a value greater than 255, an error occurs
- If the `syscall` command receives an unknown code, an error occurs

2.2.3 Flags

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 4](#).

Table 4: `flags` register bits semantics (counting from the lowest, 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/greater or equal’ conditions to also be true.

2.2.4 Labels

Either before a command or on a separate line one may place a *label*, i.e. a word consisting of latin letters and/or numbers and not starting with a number. It can be used later on in the assembler code to indicate the address of the command it is placed before.

Notes:

- A label must be followed by a colon
- A use of an undefined label causes an error making it impossible to create an executable file from the assembler script
- One can use a label defined later on in the code
- The labels must be unique, i.e. they must not repeat or conflict with predefined words

2.2.5 End directive

An assembler program must have *exactly one* end directive, which must be in *the last* line of the program. It has one operand which indicates the address of the first instruction (or a label).

2.2.6 Comments

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

2.3 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
- A function call does not include the function arguments. They can be passed either via the stack or via the registers (by a programmer-defined convention). However, if a function is directly or indirectly recursive, the best practice is to pass the arguments via the stack
- 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 machine code is placed at its beginning, while the stack starts at its end and grows ‘backwards’
- The stack does not have any size limits besides the address space size
- 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 not considered a good practice

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 5: Karma executable file format

Bytes	Contents
0..15	ASCII string “ThisIsKarmaExec”
16..19	Program code size
20..23	Program constants size
24..27	Program data size
28..31	Address of the first instruction
32..35	Initial stack pointer value
36..39	ID of the target processor
512..	Code segment
	Constants segment
	Data 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
- The code, constants and data segments are loaded into the virtual Karma computer starting from the first memory cell
- The execution of the program starts from the instruction the address of which is specified in the executable file header
- The header also specifies the initial stack head address

4 Code samples

4.1 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             ; clear r0
    syscall r0, 0        ; exit the program with code 0
    end main            ; start execution from label main
```

4.2 Calculate the square of a number with functions

```
sqr:
    loadr r0, r14, 1      ; a function calculating the square with one argument on the stack
                        ; 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 1                ; return from function and remove the argument from the stack

intout:
    load r0, r14, 1      ; a function printing its argument and '\n'
                        ; 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 1                ; return from function and remove the argument from the stack

main:
    syscall r0, 100      ; read an integer from stdin to r0
    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             ; clear r0
    syscall r0, 0        ; exit the program with code 0
    end main            ; start execution from label main
```

4.3 Calculate the factorial of a number using a loop

```
fact:                                ; a non-recursive function calculating the factorial of its argument
    loadr r0, r14, 1                ; 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 1                          ; return from function and remove the argument from the stack

main:
    syscall r0, 100                 ; read an integer from stdin to r0
    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                        ; clear r0
    syscall r0, 0                   ; exit the program with code 0
    end main                        ; start execution from label main
```

4.4 Calculate the factorial of a number using recursion

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

main:
    syscall r0, 100                 ; read an integer from stdin to r0
    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                        ; clear r0
    syscall r0, 0                   ; exit the program with code 0
    end main                        ; start execution from label main
```