Karma Computer

Tigran Koshkelian

Contents

1 Architecture

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

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

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

2 **Karma** assembler standard

2.1 Command formats

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	Example:
Register-memory	RM	command	register	memory address		`load r0, 12956`
Register-register	RR	command	receiver register	source register	source modifier	`mov r1, r2, -0xa21`
Register-immediate	RI	command	register	immediate operand		`ori r2, 64`
Jump	J	command	ignored	memory address		`calli 01547`

For RM and J format commands the address operand in the code may be:

- 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 immediate operand for RR and RI format commands.

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.

2.2 `Karma` processor command set

Table 3: `Karma` processor commands description

Code	Name	Format	Description
0	`halt`	RI	Stop processor `halt r1 0`
1	`syscall`	RI	System call See dedicated section for details `syscall r0, 100`
2	`add`	RR	`add r1, r2, 3` $\Longleftrightarrow$ `r1 += r2 + 3`
3	`addi`	RI	`addi r4, 10` $\Longleftrightarrow$ `r1 += 10`
4	`sub`	RR	`sub r3, r5, 5` $\Longleftrightarrow$ `r3 -= r5 + 5`
5	`subi`	RI	`subi r4, 1` $\Longleftrightarrow$ `r4 += 1`
6	`mul`	RR	`mul r3, r10, 2` $\Longleftrightarrow$ `r3 *= r10 + 2` 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 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	`muli r5, 100` $\Longleftrightarrow$ `r5 *= 100` 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 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	`lc r7, 123` $\Longleftrightarrow$ `r7 = 123` Storing immediate operand to the register Supplemental bitwise shift and addition commands are required for storing constants greater than $2^{20} - 1$
13	`shl`	RR	`shl r1, r2, 1` $\Longleftrightarrow$ `r1` $\ll$= `r2 + 1`
14	`shli`	RI	`shli r1, 2` $\Longleftrightarrow$ `r1` $\ll$= `2`
15	`shr`	RR	`shr r1, r2, -4` $\Longleftrightarrow$ `r1` $\gg$= `r2 - 4`
16	`shri`	RI	`shri r1, 2` $\Longleftrightarrow$ `r1` $\gg$= `2`
17	`and`	RR	`and r4, r6, 3` $\Longleftrightarrow$ `r4 &= r6 + 3`
18	`andi`	RI	`andi r5, 2` $\Longleftrightarrow$ `r5 &= 2`
19	`or`	RR	`or r3, r2, -2` $\Longleftrightarrow$ `r3 |= r2 - 2`
20	`ori`	RI	`ori r6, 100` $\Longleftrightarrow$ `r6 |= 100`
21	`xor`	RR	`xor r1, r5, 0` $\Longleftrightarrow$ `r1 ^= r5`
22	`xori`	RI	`xori r1, 127` $\Longleftrightarrow$ `r1 ^= 127`
23	`not`	RI	`not r1, 0` $\Longleftrightarrow$ `r1 = ~r1` The immediate value is ignored, but per our agreement must be present for the simplicity of the compiler
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	Real-valued registers addition The floating-point values are stored in two registers, the provided registers contain the lower bits of the values (see here for details) 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 `addd r2, r5, 0` In the example above a floating-point value stored in (`r5`, `r6`) is added to the one stored in (`r2`, `r3`)
33	`subd`	RR	Real-valued registers subtraction For real value storage and immediate operand comments see `addd` `subd r1, r6, 0` In the example above a floating-point value stored in (`r6`, `r7`) is subtracted from the one stored in (`r1`, `r2`)
34	`muld`	RR	Real-valued registers multiplication For real value storage and immediate operand comments see `addd` `muld r0, r2, 0` In the example above a floating-point value stored in (`r0`, `r1`) is multiplied by the one stored in (`r2`, `r3`)
35	`divd`	RR	Real-valued registers division For real value storage and immediate operand comments see `addd` `divd r1, r3, 0` In the example above a floating-point value stored in (`r1`, `r2`) is divided by the one stored in (`r3`, `r4`)
36	`itod`	RR	Integer to floating-point transformation 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 `itod r2, r5, 5` 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
37	`dtoi`	RR	Floating-point to integer transformation The real value specified by the source register is rounded down to the closest integer For real value storage and immediate operand comments see `addd` If the resulting value does not fit a register, an error occurs `dtoi r2, r5, 0` In the example above the floating-point value stored in `(r5, r6)` is rounded down and stored in `r2`

38	`push`	RI	Push the value from the source register modified by the immediate operand to the stack (and then move the stack pointer) `push r0, 255` 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
39	`pop`	RI	Pop the value from the stack and store it in the receiver register after modifying by the immediate operand (after moving the stack pointer) `pop r3, 3` 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`
40	`call`	RR	Call the function, the address of which can be acquired by modifying the source register by the immediate operand The address of the command following the current one is stored in the receiver register `call r0, r5, 2` 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`
41	`calli`	J	Call the function, the address of which is specified by the immediate operand `calli 13323` 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`)
42	`ret`	J	Return from function to caller The address of the next executed instruction is popped from the stack 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) `ret 3` 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`
43	`cmp`	RR	Registers comparison 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 `cmp r0, r1, 2` In the example above `r0` is compared to `r1+2`
44	`cmpi`	RI	Comparison with immediate operand The value in the specified register is compared to the immediate operand and the result is stored to the `flags` register `cmpi r0, 0`
45	`cmpd`	RR	Real-valued registers comparison For real value storage and immediate operand comments `addd` 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 `cmpd r1, r4, 0` In the example above the floating-point value stored in `(r1,r2)` is compared to the one stored in `(r4,r5)`

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

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

2.3 Further specifications

2.3.1 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

2.3.2 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 4: System call codes

Code	Name	Description	Register operand
0	`EXIT`	Finish execution without error	–
100	`SCANINT`	Get an integer value from `stdin`	Receiver
101	`SCANDOUBLE`	Get a floating-point value from `stdin`	Low-bits receiver
102	`PRINTINT`	Output an integer value to `stdout`	Source
103	`PRINTDOUBLE`	Output a floating-point 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

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

2.3.3 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 5: `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.

2.3.4 Labels

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

Syntax:

- A label must consist only of lowercase latin letters and/or digits and 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 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 in 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)

2.3.5 `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).

2.3.6 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.

2.4 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 6: `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:                     ; a function calculating the square with one argument on the stack
    loadr r0, r14, 1     ; 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:                  ; a function printing its argument and '\n'
    load r0, r14, 1      ; 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 that 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
```