Karma Computer

Tigran Koshkelian

Contents

1 Architecture

`Karma` is a computer with a Von Neumann architecture with an address space of 2^{20} cells, one *machine word* (32 bits) 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 pointer
`r15`	Instruction pointer
`flags`	Comparison operation result

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 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

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`	`r1 += r2 - 3`
3	`addi`	RI	`addi r4 10`	`r4 += 10`
4	`sub`	RR	`sub r3 r5 5`	`r3 -= r5 + 5`
5	`subi`	RI	`subi r4 1`	`r4 -= 1`
6	`mul`	RR	`mul r3 r7 -2`	`(r4, r3) = r3 * (r7 - 2)`
7	`muli`	RI	`muli r5 100`	`(r6, r5) = r5 * 100`

8	`div`	RR	`div r3 r8 5`	`tmp = (r4, r3)`
				`r3 = tmp / (r8 + 5)`
				`r4 = tmp % (r8 + 5)`

9	`divi`	RI	`divi r6 7`	`tmp = (r7, r6)`
				`r6 = tmp / 7`
				`r7 = 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 of the signed operands (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 in 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 and perform an integral division. The quotient is then placed in the receiver register and the remainder – in 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`	r1 = ∼r1
11	shl	RR	`shl r1 r2 1`	r1 ≪= r2 + 1
12	shli	RI	`shli r1 2`	r1 ≪= 2
13	shr	RR	`shr r1 r2 -4`	r1 ≫= r2 - 4
14	shri	RI	`shri r1 2`	r1 ≫= 2
15	and	RR	`and r4 r6 3`	r4 &= r6 + 3
16	andi	RI	`andi r5 2`	r5 &= 2
17	or	RR	`or r3 r2 -2`	r3 |= r2 - 2
18	ori	RI	`ori r6 100`	r6 |= 100
19	xor	RR	`xor r1 r5 0`	r1 ^= r5
20	xori	RI	`xori r1 127`	r1 ^= 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 (see the respective section for details) 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`	`(r3, r2) = double(r5+5)`
22	dtoi	RR	`dtoi r2 r5 0`	`r2 = uint32((r6, r5))`
23	addd	RR	`addd r2 r5 0`	`(r3, r2) += (r6, r5)`
24	subd	RR	`subd r1 r6 0`	`(r2, r1) -= (r7, r6)`
25	muld	RR	`muld r0 r2 0`	`(r1, r0) *= (r3, r2)`
26	divd	RR	`divd r1 r3 0`	`(r2, r1) /= (r4, r3)`

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

`itod`

The source modifier is added to the source register in a common manner (see the respective section for details), after which the resulting `uint32` value is converted to `double` and stored in 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` and stored in the receiver register. If the value obtained after conversion does not fit into `uint32`, 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 Stack operators and function calls

Table 8: Stack operators and function calls

Code	Name	Format	Syntax sample	Description
38	`push`	RI	`push r0 255`	fff
39	`pop`	RI	`pop  r3 3`	fff
40	`call`	RR	`call r0 r5 2`	fff
41	`calli`	J	`calli 21913`	fff
42	`ret`	J	`ret 3`	fff

2.3.6 Comparisons and jumps

Table 9: Comparisons and jumps

Code	Name	Format	Syntax sample	Description
43	`cmp`	RR	`cmp r0 r1 2`	`r0 <=> r1 + 2`
44	`cmpi`	RI	`cmpi r0 0`	`r0 <=> 0`
45	`cmpd`	RR	`cmpd r1 r4 0`	Register real-valued comparison
46	`jmp`	J	`jmp 0xffa`	Unconditional jump
47	`jne`	J	`jne 0xd471`	Jump if not equal
48	`jeq`	J	`jeq 05637`	Jump if equal
49	`jle`	J	`jle 1234`	Jump if less or equal
50	`jl`	J	`jl 0X1e4b`	Jump if less
51	`jge`	J	`jge 29834`	Jump if greater or equal
52	`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 11.

Table 10: `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.

2.3.7 Data transfer commands

Table 11: Data transfer commands

Code	Name	Format	Syntax sample	Description
64	`load`	RM	`load r0, 12345`	
65	`store`	RM	`store r0, 12344`	
66	`load2`	RM	`load2 r0, 12345`	
67	`store2`	RM	`store2 r0, 12344`	
68	`loadr`	RR	`loadr r0, r1, 15`	
69	`storer`	RR	`storer r0, r11, 3`	
70	`loadr2`	RR	`loadr2 r0, r10, 12`	
71	`storer2`	RR	`storer2 r0, r3, 10`	

2.4 Further specifications

2.4.1 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)

2.4.2 `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).

2.4.3 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.

2.5 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 12: `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
```