Ducky Documentation

Release 3.0

Milos Prchlik

Nov 15, 2017

Contents

Ducky is a simple virtual CPU/machine simulator, with modular design and interesting features.

CHAPTER 1

About

1.1 ducky

Ducky is a simple virtual CPU/machine simulator, with modular design and interesting features. Ducky was created for learning purposes, no bigger ambitions. The goal was to experiment with CPU and virtual machine simulation, different instruction sets, and later working FORTH kernel become one of the main goals.

1.1.1 Features

Ducky - as in "Ducky, the CPU" - is a 32-bit RISC CPU. Ducky, "the VM", is a simulator of Ducky CPU, adding few other modules to create the whole virtual machine, with CPUs, peripherals, storages and other components.

RISC instruction set

Instruction set was inspired by RISC CPUs, and sticks to LOAD/STORE aproach, with fixed-width instructions.

Memory model

Flat, paged, with linear addressing. Main memory consists of memory pages, each page supports simple access control - simple MMU is implemented.

Modular architecture

Virtual machine consists of several modules of different classes, and only few of them are necessary (e.g. CPU). Various peripherals are available, and it's extremely easy to develop your own and plug them in.

SMP support

Multiple CPUs with multiple cores per each CPU, with shared memory. Each core can be restricted to its own segment of memory.

Persistent storage

Modular persistent storages are available, and accessible by block IO operations, or by mmap-ing storages directly into memory.

Bytecode files

Compiled programs are stored in bytecode files that are inspired by ELF executable format. These files consist of common sections (`.text`, `.data`, ...), symbols, and their content. Assembler (`ducky-as`) translates assembly sources into object files, and these are then processed by a linker (`ducky-ld`) into the final executable. Both object and executable files use the same format and bytecode for instructions and data.

Snapshots

Virtual machine can be suspended, saved, and later restored. This is also useful for debugging purposes, every bit of memory and CPU registers can be investigated.

Debugging support

Basic support is included - break points, watch points, stack traces, stepping, snapshots, ...

Tools

- `as` for translating assembler sources to bytecode files
- `ld` for linking bytecode files into the final executable
- `objdump` for inspection of bytecode files
- `coredump` for inspection of snapshots
- `vm` for running virtual machine itself
- `img` for converting binaries to images

Planned features

- `FORTH` kernel - basic functionality but at least ANS compliant
- network support - it would be awesome to have a network stack available for running programs
- functioning C compiler, with simple C library
- and few others...

1.1.2 Need help?

The whole development is tracked on a GitHub page, including source codes and issue tracker.

1.2 Getting started

1.2.1 Installing

The easy way is to use package:

```
pip install ducky
```

Or, you can install Ducky by checking out the sources:

```
git clone https://github.com/happz/ducky.git
cd ducky
python setup.py
```

After this, you should have the `ducky` module on your path:

```
>>> import ducky
>>> ducky.__version__
'3.0'
```

1.2.2 Prerequisites

Ducky runs with both Python 2 *and* 3 - supported versions are 2.7, 3.3, 3.4 and 3.5. There are few other dependencies, installation process (or `setup.py`) should take care of them autmatically. PyPy is also supported, though only its implementation of Python2.

1.2.3 "Hello, world!" tutorial

Let's try out the "Hello, world!" example. It's a simple program that just prints out the well-known message.

Source code

Source is located in `examples/hello-world` directory. If you check it out, it's a plain and simple assembler:

```
#include <arch/tty.hs>

.data

.type stack, space, 64
.type message, string, "Hello, world!"


.text

main:
  la sp, stack
  add sp, 64

  la r0, message
  call writesn
  hlt 0x00

writesn:
```

```
  // > r0: string address
  // ...
  //   r0: port
  //   r1: current byte
  //   r2: string ptr
  push r1
  push r2
  mov r2, r0
  li r0, TTY_MMIO_ADDRESS
  add r0, TTY_MMIO_DATA

.__writesn_loop:
  lb r1, r2
  bz .__writesn_write_nl
  stb r0, r1
  inc r2
  j .__writesn_loop

.__writesn_write_nl:
  // \n
  li r1, 0x0000000A
  stb r0, r1
  // \r
  li r1, 0x0000000D
  stb r0, r1
  li r0, 0x00000000
  pop r2
  pop r1
  ret
```

It's a little bit more structured that necessary, just for educational purposes.

Binary

To run this code, we have to create a *binary* of it. Of course, there are tools for this very common goal:

```
ducky-as -i examples/hello-world/hello-world.s -o examples/hello-world/hello-world.o
```

This command will translate source code to an *object file* which contains instructions and other necessary resources for *machine* to run it. You can inspect the object file using `objdump` tool:

```
ducky-objdump -i examples/hello-world/hello-world.o -a
```

This should produce output similar to this one:

```
[INFO] Input file: examples/hello-world/hello-world.o
[INFO]
[INFO] === File header ===
[INFO]   Magic:    0xDEAD
[INFO]   Version:  1
[INFO]   Sections: 4
[INFO]
[INFO] === Sections ===
[INFO]
[INFO]   Index  Name      Type     Flags        Base        Items    Size    Offset
[INFO] -------  --------  -------  -----------  --------  -------  ------  --------
[INFO]       0  .data     DATA     RW-- (0x03)  0x000000       14      14       104
```

```
[INFO]       1  .text     TEXT     RWX- (0x07)  0x000100       24       96       118
[INFO]       2  .symtab   SYMBOLS  ---- (0x00)  0x000200        6      120       214
[INFO]       3  .strings  STRINGS  ---- (0x00)  0x000000        0      122       334
[INFO]
[INFO] === Symbols ===
[INFO]
[INFO] Name                    Section    Address    Type            Size  File
↪               Line    Content
[INFO] ----------------------  ---------  ---------  ------------  ------  -----------
↪-------------  ------  ---------------
[INFO] message                 .data      0x000000   string (2)        14  examples/
↪hello-world.asm  1       "Hello, world!"
[INFO] main                    .text      0x000100   function (3)       0  examples/
↪hello-world.asm  4
[INFO] outb                    .text      0x000110   function (3)       0  examples/
↪hello-world.asm  10
[INFO] writesn                 .text      0x000118   function (3)       0  examples/
↪hello-world.asm  16
[INFO] .__fn_writesn_loop      .text      0x00012C   function (3)       0  examples/
↪hello-world.asm  27
[INFO] .__fn_writesn_write_nl  .text      0x000140   function (3)       0  examples/
↪hello-world.asm  33
[INFO]
[INFO] === Disassemble ==
[INFO]
[INFO]   Section .text
[INFO]   0x000100 (0x00000004) li r0, 0x0000
[INFO]   0x000104 (0x0000800D) call 0x0010
[INFO]   0x000108 (0x00000004) li r0, 0x0000
[INFO]   0x00010C (0x0000000B) int 0x0000
[INFO]   0x000110 (0x000000E3) outb r0, r1
[INFO]   0x000114 (0x0000000E) ret
[INFO]   0x000118 (0x000000D4) push r1
[INFO]   0x00011C (0x00000154) push r2
[INFO]   0x000120 (0x00000054) push r0
[INFO]   0x000124 (0x00000095) pop r2
[INFO]   0x000128 (0x00040004) li r0, 0x0100
[INFO]   0x00012C (0x00000842) lb r1, r2
[INFO]   0x000130 (0x00006029) bz 0x000C
[INFO]   0x000134 (0x0FFEC00D) call -0x0028
[INFO]   0x000138 (0x00000096) inc r2
[INFO]   0x00013C (0x0FFF6026) j -0x0014
[INFO]   0x000140 (0x00002844) li r1, 0x000A
[INFO]   0x000144 (0x0FFE400D) call -0x0038
[INFO]   0x000148 (0x00003444) li r1, 0x000D
[INFO]   0x00014C (0x0FFE000D) call -0x0040
[INFO]   0x000150 (0x00000004) li r0, 0x0000
[INFO]   0x000154 (0x00000095) pop r2
[INFO]   0x000158 (0x00000055) pop r1
[INFO]   0x00015C (0x0000000E) ret
[INFO]
```

As you can see, object file contains instructions, some additional data, list of symbols, and some more, with labels replaced by dummy offsets. Offsets in jump instructions make no sense yet because object file is not the finalized binary - yet. For that, there's yet another tool:

```
ducky-ld -i examples/hello-world/hello-world.o -o examples/hello-world/hello-world
```

This command will take object file (or many of them), and produce one *binary* by merging code, data and other sections from all source object files, and updates addresses used by instructions to retrieve data and to perform jumps. You can inspect the resulting binary file using `objdump` tool as well:

```
ducky-objdump -i examples/hello-world/hello-world -a
```

This should produce output very similar to the one you've already seen - not much had changed, there was only one object file, only offsets used by `call` and `j` instructions are now non-zero, meaning they are now pointing to the correct locations.

Running

Virtual machine configuration can get quite complicated, so I try to avoid too many command line options, and opt for using configuration files. For this example, there's one already prepared. Go ahead and try it:

```
ducky-vm --machine-config=examples/hello-world/hello-world.conf --set-
↪option=bootloader:file=examples/hello-world/hello-world
```

There are two command-line options:

- `--machine-config` tells VM where to find its configuration file,
- `--set-option` modifies this configuration; this particular instance tells VM to set `file` option in section `bootloader` to path of our freshly built binary, `examples/hello-world/hello-world`. Since I run examples during testing process, their config files lack this option since it changes all the time.

You should get output similar to this:

```
1441740855.82 [INFO] Ducky VM, version 1.0
1441740855.82 [INFO] mm: 16384.0KiB, 16383.5KiB available
1441740855.82 [INFO] hid: basic keyboard controller on [0x0100] as device-1
1441740855.83 [INFO] hid: basic tty on [0x0200] as device-2
1441740855.83 [INFO] hid: basic terminal (device-1, device-2)
1441740855.83 [INFO] snapshot: storage ready, backed by file ducky-snapshot.bin
1441740855.83 [INFO] RTC: time 21:34:15, date: 08/09/15
1441740855.83 [INFO] irq: loading routines from file interrupts
1441740856.02 [INFO] binary: loading from from file examples/hello-world/hello-world
1441740856.02 [INFO] #0:#0: CPU core is up
1441740856.02 [INFO] #0:#0:   check-frames: yes
1441740856.02 [INFO] #0:#0:   coprocessor: math
1441740856.02 [INFO] #0: CPU is up
Hello, world!
1441740856.04 [INFO] #0:#0: CPU core halted
1441740856.05 [INFO] #0: CPU halted
1441740856.05 [INFO] snapshot: saved in file ducky-snapshot.bin
1441740856.05 [INFO] Halted.
1441740856.05 [INFO]
1441740856.05 [INFO] Exit codes
1441740856.05 [INFO] Core      Exit code
1441740856.06 [INFO] ------  -----------
1441740856.06 [INFO] #0:#0             0
1441740856.06 [INFO]
1441740856.06 [INFO] Instruction caches
1441740856.06 [INFO] Core      Reads    Inserts    Hits    Misses    Prunes
1441740856.06 [INFO] ------  -------  ---------  ------  --------  --------
1441740856.06 [INFO] #0:#0       133         34      99        34         0
1441740856.06 [INFO]
1441740856.06 [INFO] Core    Ticks
```

```
1441740856.06 [INFO] ------  -------
1441740856.06 [INFO] #0:#0   133
1441740856.06 [INFO]
1441740856.06 [INFO] Executed instructions: 133 0.028670 (4639.0223/sec)
1441740856.06 [INFO]
```

And there, on line 16, between all that funny nonsenses, it is! :) The rest of the output are just various notes about loaded binaries, CPU caches, nothing important right now.

And that's it.

1.3 Virtual hardware

1.3.1 CPU

Ducky VM can have multiple CPUs, each with multiple cores. Each core is a 32-bit microprocessor, with 32 32-bit registers, connected to main memory. It is equiped with MMU, and its own instruction cache.

CPU core can work in privileged and unprivileged modes, allowing use of several protected instructions in privileged mode.

Registers

- 32 32-bit registers - registers `r0` to `r29` are general purpose registers - `r31` is reserved, and used as a stack pointer register, `SP` - contains address of the last value pushed on stack - `r30` is reserved, and used as a frame pointer register, `FP` - contains content of `SP` in time of the last `call` or `int` instruction
- `flags` register

Flags register

`flags` register can be considered as read-only value, since it is not possible to modify it using common bit operations or arithmetic instructions. However, its content reflect outcomes of executed instructions, therefore it is possible e.g. to modify it content using comparison instructions. It is also possible to inspect and modify it in exception service routine, where pre-exception `flags` are stored on the stack (and loaded when ESR ends).

Mask	Flags	Usage
0x00	`privileged`	If set, CPU runs in privileged mode, and usage of protected instructions is allowed
0x01	`hwint_allowed`	If set, HW interrupts can be delivered to this core
0x04	`e`	Set if the last two compared registers were equal
0x08	`z`	Set if the last arithmetic operation produced zero
0x10	`o`	Set if the last arithmetic operation overflown
0x20	`s`	Set if the last arithmetic operation produced negative result

Instruction set

CPU supports multiple instruction sets. The default one, *ducky*, is the main workhorse, suited for general coding, but other instruction sets can exist, e.g. coprocessor may use its own instruction set for its operations.

Ducky instruction set

Design principles

- basic data unit is *a word* - 4 bytes, 32 bits. Other units are *short* - 2 bytes, 16 bits - and *byte* - 1 byte, 8 bits. Instructions often have variants for different data units, distinguished by a suffix (*w* for words, *s* for shorts, and *b* for single bytes)
- load and store operations are performed by dedicated instructions
- memory-register transfers work with addresses that are aligned to the size of their operands (1 byte alignment - so no alignment at all - for byte operands)
- unless said otherwise, destination operand is the first one
- when content of a register is changed by instruction, several flags can be modified subsequently. E.g. when new value of register is zero, `z` flag is set.

Notes on documentation

- `rN` refers to generaly any register, from `r0` up to `r29` - special registers are refered to by their common names (e.g. `SP`).
- `rA`, `rB` refer to the first and the second instruction operand respectively and stand for any register.
- `<value>` means immediate, absolute value. This covers both integers, specified as base 10 or base 16 integers, both positive and negative, and labels and addresses, specified as `&label`
- when instruction accepts more than one operand type, it is documented using `|` character, e.g. `(rA|<value>)` means either register or immediate value
- immediate values are encoded in the instructions, therefore such value cannot have full 32-bit width. Each instruction should indicate the maximal width of immediate value that can be safely encoded, should you require grater values, please see `li` and `liu` instructions

Stack frames

Several instructions transfer control to other parts of program, with possibility of returning back to the previous spot. It is done by creating a *stack frame*. When stack frame is created, CPU performs these steps:

- `IP` is pushed onto the stack
- `FP` is pushed onto the stack
- `FP` is loaded with value of `SP`

Destroying stack frame - reverting the steps above - effectively transfers control back to the point where the subroutine was called from.

Arithmetic

All arithmetic instructions take at least one operand, a register. In case of binary operations, the second operand can be a register, or an immediate value (15 bits wide, sign-extended to 32 bits). The result is always stored in the first operand.

```
add rA, (rB|<value>)
```

```
dec rA
inc rA
mul rA, (rB|<value>)
sub rA, (rB|<value>)
```

Bitwise operations

All bitwise operations - with exception of `not` - take two operands, a register, and either another register or an immediate value (15 bits wide, sign-extended to 32 bits). The result is always stored in the first operand.

```
and rA, (rB|<value>)
not rA
or rA, (rB|<value>)
shiftl rA, (rB|<value>)
shiftr rA, (rB|<value>)
shiftrs rA, (rB|<value>)
xor rA, (rB|<values>)
```

Branching instructions

Branching instructions come in form `<inst> (rA|<address>)`. If certain conditions are met, branching instruction will perform jump by adding value of the operand to the current value of `PC` (which, when instruction is being executed, points *to the next instruction* already). If the operand is an immediate address, it is encoded in the instruction as an immediate value (16 bit wide, sign-extended to 32 bits). This limits range of addresses that can be reached using this form of branching instructions.

Branching instructions do not create new stack frame.

Unconditional branching

```
j (rA|<value>)
```

Conditional branching

Instruction	Jump when ...
`be`	`e = 1`
`bne`	`e = 0`
`bs`	`s = 1`
`bns`	`s = 0`
`bz`	`z = 1`
`bnz`	`z = 0`
`bo`	`o = 1`
`bno`	`o = 0`
`bg`	`e = 0` and `s = 0`
`bge`	`e = 1` or `s = 0`
`bl`	`e = 0` and `s = 1`
`ble`	`e = 1` or `s = 1`

Conditional setting

All conditional setting instructions come in form `<inst> rA`. Depending on relevant flags, `rA` is set to `1` if condition is evaluated to be true, or to `0` otherwise.

For flags relevant for each instruction, see branching instruction with the same suffix (e.g. `setle` evaluates the same flags with the same result as `ble`).

Instruction
`sete`
`setne`
`setz`
`setnz`
`seto`
`setno`
`sets`
`setns`
`setg`
`setge`
`setl`
`setle`

Comparing

Two instructions are available for comparing of values. Compare their operands and sets corresponding flags. The second operand can be either a register or an immediate value (15 bits wide).

`cmp rA, (rB|<value>)` - immediate value is sign-extended to 32 bits.

`cmpu rA, (rB|<value>)` - treat operands as unsigned values, immediate value is zero-extended to 32 bits.

Interrupts

Delivery

If flag `hwint_allowed` is unset, no hardware IRQ can be accepted by CPU and stays queued. All queued IRQs will be delivered as soon as flag is set.

`cli` - clear `hwint` flag

`sti` - set `hwint` flag

In need of waiting for external events it is possible to suspend CPU until the next IRQ is delivered.

`idle` - wait until next IRQ

Invocation

Any interrupt service routine can be invoked by means of special instruction. When invoked several events take place:

- `SP` is saved in temporary space
- `IP` and `SP` are set to values that are stored in `EVT` in the corresponding entry
- important registers are pushed onto new stack (in this order): old `SP`, `flags`
- new stack frame is created
- privileged mode is enabled
- delivery of hardware interrupts is disabled

When routine ends (via `retint`), these steps are undone, and content of saved registers is restored.

```
int (rA|<index>)
```

`retint` - return from interrupt routine

Inter-processor interrupts (`IPI`) can be delivered to other processors, via dedicated instruction, similar to `int` but specifying CPUID of target core in the first operand.

```
ipi rA, (rB|<index>)
```

Routines

When routine is called, new stack frame is created, and CPU continues with instructions pointed to by the first operand. For its meaning (and limitations) see *Branching instructions*.

```
call (rA|<address>)
```

```
ret
```

Stack

```
pop rA
```

```
push (rA|<value>)
```

Miscellaneous

`nop` - do absolutely nothing

`hlt (rA|<value>)` - Halt CPU and set its exit code to specified value.

`rst` - reset CPU state. All flags cleared, `privileged = 1`, `hwint_allowed = 0`, all registers set to `0`

`mov rA, rB` - copy value of `rB` into `rA`

`swp rA, rB` - swap content of two registers

`sis <value>` - switch instruction set to a different one

Memory access

Address operand - `{address}` - can be specified in different ways:

- `rA` - address is stored in register
- `rA[<offset>]` - address is computed by addition of `rA` and `offset`. `offset` can be both positive and negative. `fp` and `sp` can be also used as `rA`. `<offset>` is an immediate value, 15 bits wide, sign-extended to 32 bits.

Read

`lw rA, {address}` - load word from memory

`ls rA, {address}` - load short from memory

`lb rA, {address}` - load byte from memory

Write

`stw {address}, rA`

`sts {address}, rA` - store lower 2 bytes of `rA`

`stb {addres}, rA` - store lower byte of `rA`

Constants

Instructions for filling registers with values known in compile time.

`li rA, <constant>` - load `constant` into register. `constant` is encoded into instruction as an immediate value (20 bits wide, sign-extended to 32 bits)

`liu rA, <constant>` - load `constant` into the upper half of register. `constant` is encoded into instruction as an immediate value (20 bits wide immediate, only lower 16 bits are used)

`la rA, <constant>` - load `constant` into the register. `constant` is an immediate value (20 bits wide, sign-extended to 32 bits), and is treated as an offset from the current value of `PC` - register is loaded with the result of `PC + constant`.

Compare-and-swap

`cas rA, rB, rC` - read word from address in register `rA`. Compare it with value in register `rB` - if both are equal, take content of `rC` and store it in memory on address `rA`, else store memory value in `rB`.

Math coprocessor instruction set

This page describes instruction set of math coprocessor.

Manipulating registers

Instructions for moving values between coprocessor stack and memory or CPU registers.

`itol`

`utol`

`ltoi`

`ltoii`

`iitol`

`dupl`

Arithmetic operations

`incl`

`decl`

`addl`

`subl`

`mull`

`divl`

`modl`

`symdivl`

`symmodl`

1.3.2 Control coprocessor

Control coprocessor (*CC*) is a coprocessor dedicated to control various runtime properties of CPU core. Using `ctr` and `ctw` instructions it is possible to inspect and modify these properties.

Control registers

CR0

`cr0` register contains so-called `CPUID` value which intendifies location of this CPU core in global CPU topology.

- upper 16 bits contain index number of CPU this core belongs to
- lower 16 bits contain index number of this core in the context of its parent CPU

There's always core with `CPUID 0x00000000`, since there's always at least one core present.

This register is read-only.

CR1

`cr1` register contains *Exception Vector Table* address for this core. Be aware that *any* address is accepted, no aligment or any other restrictions are applied.

CR2

`cr2` register contains page table address for this core. Be aware that *any* address is accepted, no aligment or any other restrictions are applied.

CR3

`cr3` register provides access to several flags that modify behavior of CPU core.

Mask	Flag	Usage
0x00	`pt_enabled`	If set, MMU consults all memory accesses with page tables.
0x01	`jit`	If set, JIT optimizations are enabled.
0x02	`vmdebug`	If set, VM will produce huge amount of debugging logs.

Note: `jit` flag is read-only. It is controlled by options passed to Ducky when VM was created, and cannot be changed in runtime.

Note: `vmdebug` flag is shared between all existing cores. Changing it on one core affects immediately all other cores.

Note: `vmdebug` flag will not produce any debugging output if debug mode was disabled e.g. by not passing `-d` option to `ducky-vm` tool. If debug mode was allowed, changing this flag will control log level of VM.

1.3.3 Exception Vector Table

Exception vector table (*EVT*) is located in main memory, by default at address `0x00000000`, and provides core with routines that can help resolve some of exceptional states core can run into.

`EVT` address can be set per CPU core, see *CR1*.

`EVT` is 256 bytes - 1 memory page - long, providing enough space for 32 entries. Typically, lower 16 entries are reserved for hardware interrupts, provided by devices, and upper 16 entries lead to software routines that deal with exceptions, and provide additional functionality for running code in form of software interrupts.

Entry format

IP - 32 bits	SP - 32 bits

When CPU is interrupted - by hardware (device generates IRQ) or software (exception is detected, or program executes `int` instruction) interrupt - corresponding entry is located in `EVT`, using interrupt ID as an index.

1.3.4 Hardware Description Table

Hardware Description Table (*HDT*) is located in main memory, by default at address `0x00000100`, and hardware setup of the machine.

1.3.5 Memory

Memory model

- the full addressable memory is 4 GB, since the address bus is 32-bit wide, but it is quite unrealistic expectation. I usually stick to 24-bits for addresses, which leads to 16MB of main memory
- memory is organized into pages of 256 bytes - each page can be restricted for read, write and execute operations

Memory layout

Stack

- standard LIFO data structure
- grows from higher addresses to lower
- there is no pre-allocated stack, every bit of code needs to prepare its own if it intends to use instructions that operate with stack
- when push'ing value to stack, `SP` is decreased by 4 (size of general register), and then value is stored on this address
- each `EVT` provides its own stack pointer

1.4 Software

1.4.1 Calling convention

I use very simple calling convention in my code:

- all arguments are in registers
- first argument in `r0`, second in `r1`, ... You get the picture.
- if there's too many arguments, refactor your code or use a stack...
- return value is in `r0`
- callee is reponsible for save/restore of registers it's using, with exception of:
  - registers that were used for passing arguments - these are expected to have undefined value when callee returns

  - `r0` if callee returns value back to caller

All virtual interrupt routines, assembler code, any pieces of software I've written for this virtual machine follows this calling convention - unless stated otherwise...

1.4.2 Software interrupts

Software interrupts provide access to library of common functions, and - in case of virtual interrupts - to internal, complex and otherwise inaccessible resources of virtual machine itself.

For the list for existing interrupts and their numbers, see `ducky.irq.IRQList`. However, by the nature of invoking a software interrupt, this list is not carved into a stone. You may easily provide your own `EVT`, with entries leading to your own routines, and use e.g. the 33th entry, `HALT`, to sing a song.

All values are defined in files in `defs/` directory which you can - and should - include into your assembler sources.

BLOCKIO

`EVT` entry	`33`		
		Read mode	Write mode
Parameters	`r0`	device id	
	`r1`	bit #0: `0` for read, `1` for write bit #1: `0` for synchronous, `1` for asynchronous operation	
	`r2`	block id	src memory address
	`r3`	dst memory address	block id
	`r4`	number of blocks	
Returns	`r0`	`0` for success	

Perform block IO operation - transfer block between memory and storage device. Use the lowest bit of `r1` to specify direction:

- `0` - *read mode*, blocks are transfered from storage into memory
- `1` - *write mode*, blocks are tranfered from memory to storage

Current data segment is used for addressing memory locations.

If everything went fine, `0` is returned, any other value means error happened.

IO operation is a blocking action, interrupt will return back to the caller once the IO is finished. Non-blocking (*DMA*-like) mode is planned but not yet implemented.

This operation is implemented as a virtual interrupt, see `ducky.blockio.BlockIOInterrupt` for details.

VMDEBUG

`EVT` entry	`34`	
		`QUIET` mode
Parameters	`r0`	`0`
	`r1`	`0` for *quiet* mode, anything else for *full* mode
Returns	`r0`	`0` for success

`VM` interrupt allows control of VM debugging output. Currently, only two levels of verbosity that are available are *quiet* and *full* mode. In *quiet* mode, VM produces no logging output at all.

This interrupt can control level amount of debugging output in case when developer is interested only in debugging only a specific part of his code. Run VM with debugging turned on (`-d` option), turn the debugging off at the beggining of the code, and turn it on again at the beggining of the interesting part to get detailed output.

1.5 Tools

Ducky comes with basic toolchain necessary for development of complex programs. One piece missing is the C cross-compiler with simple C library but this issue will be resolved one day.

1.5.1 Common options

All tools accept few common options:

-q, --quiet

Lower verbosity level by one. By default, it is set to *info*, more quiet levels are *warnings, 'error* and *critical*.

-v, --verbose

Increase verbosity level by one. By default, it is set to *info*, more verbose levels is *debug*. *debug* level is not available for `ducky-vm` unless `-d` option is set.

-d, --debug

Set logging level to *debug* immediately. `ducky-vm` also requires this option to even provide any debugging output - it is not possible to emit *debug* output by setting `-v` enough times if `-d` was not specified on command-line.

When option takes an address as an argument, address can be specified either using decimal or hexadecimal base. Usually, the absolute address is necessary when option is not binary-aware, yet some options are tied closely to particular binary, such options can also accept name of a symbol. Option handling code will try to find corresponding address in binary's symbol table. This is valid for both command-line options and configuration files.

1.5.2 as

Assembler. Translates *assembler files* (`.asm`) to *object files* (`.o`) - files containing bytecode, symbol information, etc.

Options

-i FILE

Take assembly `FILE`, and create an object file from its content. It can be specified multiple times, each input file will be processed.

-o FILE

Write resulting object data into `FILE`.

This options is optional. If no `-o` is specified, `ducky-as` will then create output file for each input one by replacing its suffix by `.o`. If it is specified, number of `-o` options must match the number of `-i` options.

-f

When output file exists already, `ducky-as` will refuse to overwrite it, unless `-f` is set.

-D VAR

Define name, passed to processed assembly sources. User can check for its existence in source by `.ifdef`/`.ifndef` directives.

-I DIR

Add `DIR` to list of directories that are searched for files, when `.include` directive asks assembler to process additional source file.

-m, --mmapable-sections

Create object file with sections that can be loaded using `mmap()` syscall. This option can save time during VMstartup, when binaries can be simply mmapped into VM's memory space, but it also creates larger binary files because of the alignment of sections in file.

`-w, --writable-sections`

By default, `.text` and `.rodata` sections are read-only. This option lowers this restriction, allowing binary to e.g. modify its own code.

-b, --blob

Create object file wrapping a binary blob. Such file then contains the data from input file, encoded as ASCII data, with several symbols allowing other code to access this data.

-B BLOB_FLAGS, --blob-flags=BLOB_FLAGS

Flags of blob section. Syntax of assembly `.section` directive is accepted.

1.5.3 ld

Linker. Takes (one or multiple) *object files* (`.o`) and merges them into one, *binary*, which can be executed by VM.

Options

-i FILE

Take object `FILE`, and create binary file out of it. It can be specified multiple times, all input files will be processed into one binary file.

-o FILE

Output file.

-f

When output file exists already, `ducky-ld` will refuse to overwrite it, unless `-f` is set.

--section-base=SECTION=ADDRESS

Linker tries to merge all sections into a binary in a semi-random way - it can be influenced by order of sections in source and object files, and order of input files passed to linker. It is in fact implementation detail and can change in the future. If you need specific section to have its base set to known address, use this option. Be aware that linker may run out of space if you pass conflicting values, or force sections to create too small gaps between each other so other sections would not fit in.

1.5.4 coredump

Prints information stored in a saved VM snapshot.

1.5.5 objdump

Prints information about object and binary files.

1.5.6 profile

Prints information stored in profiling data, created by VM. Used for profiling running binaries.

1.5.7 vm

Stand-alone virtual machine - takes binary, configuration files, and other resources, and executes binaries.

VM Configuration file

Number of available options can easily get quite high, especially when different devices come into play, and setting all of them on command line is not very clear. To ease this part of VM processes, user can create a configuration file. Syntax is based on Python's `ConfigParser` (or Windows `.ini`) configuration files. It consists of sections, setting options for different subsystems (VM, CPUs, ...). You can find few configuration files in `examples/` directory.

When option takes an address as an argument, address can be specified either using decimal or hexadecimal base. Usually, the absolute address is necessary when option is not binary-aware, yet some options are tied closely to particular binary, such options can also accept name of a symbol. Option handling code will try to find corresponding address in binary's symbol table. This is valid for both command-line options and configuration files.

[machine]

cpus

Number of separate CPUs.

`int`, default `1`

cores

Number of cores per CPU.

`int`, default `1`

[memory]

size

Memory size in bytes.

`int`, default `0x1000000`

force-aligned-access

When set, unaligned memory access will lead to exception.

`bool`, default `yes`

[cpu]

math-coprocessor

When set, each CPU core will have its own math coprocessor.

`bool`, default `no`

control-coprocessor

When set, each CPU core will have its own control coprocessor.

`bool`, default `yes`

inst-cache

Number of slots in instruction cache.

`int`, default `256`

check-frame

When set, CPU cores will check if stack frames were cleaned properly when `ret` or `iret` is executed.

`bool`, default `yes`

evt-address

Address of interrupt vector table.

`int`, default `0x000000`

[bootloader]

file

Path to bootloader file.

`str`, required

[device-N]

Each section starting with `device-` tells VM to create virtual device, and provide it to running binaries. Device sections have few options, common for all kinds of devices, and a set of options, specific for each different device or driver.

klass

Device class - arbitrary string, describing family of devices. E.g. I use `input` for devices processing user input (e.g. keyboard controllers).

`str`, required

driver

Python class that *is* the device driver.

`str`, required

master

If set, `master` is superior device, with some responsibilities over its subordinates.

`str`, optional

Options

`--machine-config=PATH`

Specify `PATH` to VM configuration file. For its content, see *VM Configuration file*.

--set-option=SECTION:OPTION=VALUE

--add-option=SECTION:OPTION=VALUE

These two options allow user to modify the content of configuration file, by adding of new options or by changing the existing ones.

Lets have (incomplete) config file `vm.conf`:

```
[machine]
cpu = 1
core = 1

[binary-0]
```

You can use it to run different binaries without having separate config file for each of them, just by telling `ducky-vm` to load configuration file, and then change one option:

```
$ ducky-vm --machine-config=vm.conf --add-option=binary-0:file=<path to binary of␣
↪your choice>
```

Similarly, you can modify existing options. Lets have (incomplete) config file `vm.conf`:

```
[machine]
cpus = 1
cores = 1

[binary-0]
file = some/terminal/app

[device-1]
klass = input
driver = ducky.devices.keyboard.KeyboardController
master = device-3

[device-2]
klass = output
driver = ducky.devices.tty.TTY
master = device-3

[device-3]
klass = terminal
driver = ducky.devices.terminal.StandardIOTerminal
input = device-1
output = device-2
```

Your `app` will run using VM's standard IO streams for input and out. But you may want to start it with a different kind of terminal, e.g. PTY one, and attach to it using `screen`:

```
$ ducky-vm --machine-config=vm.conf --set-option=device-3:driver=ducky.devices.
↪terminal.StandalonePTYTerminal
```

--enable-device=DEVICE

--disable-device=DEVICE

Shortcuts for `--set-option=DEVICE:enabled=yes` and `--set-option=DEVICE:enabled=no` respectively.

--poke=ADDRESS:VALUE:LENGTH

`poke` option allows modification of VM's memory after all binaries and resources are loaded, just before the VM starts execution of binaries. It can be used for setting runtime-specific values, e.g. argument for a binary.

Consider a simple binary, running a loop for specified number of iterations:

By default, 10 iterations are hard-coded into binary. If you want to termporarily change number of iterations, it's not necessary to recompile binary. By default, this binary, being the only one running, would get segment `0x02`, it's `.data` section was mentioned first, therefore its base address will be `0x0000`, leading to `loops` having absolute address `0x020000`. Then:

```
$ ducky-vm --machine-config=vm.conf --poke=0x020000:100:2
```

will load binary, then modify its `.data` section by changing value at address `0x020000` to `100`, which is new number of iterations. Meta variable `LENGTH` specifies number of bytes to overwrite by `poke` value, and `poke` will change exactly `LENGTH` bytes - if `VALUE` cannot fit into available bits, exceeding bits of `VALUE` are masked out, and `VALUE` that can fit is zero-extended to use all `LENGTH` bytes.

--jit

Enable *JIT* - more dense implementation of Ducky instructions is used. Result is higher execution speed of each instruction, however it removes many debugging code. It may be difficult to debug instruction execution even with `-d` option enabled.

1.5.8 img

Converts binaries to binary images that can be loaded by boot loader.

Options

-i FILE

Take binary `FILE`, and create a binary image from its content.

-o FILE

Write resulting object data into `FILE`.

-f

When output file exists already, `ducky-img` will refuse to overwrite it, unless `-f` is set.

1.6 Examples

1.6.1 "Hello, world!"

1.6.2 "Hello, world!" using library

1.6.3 VGA

1.6.4 Clock

1.7 Glossary

binary Binary is the final piece in the process

bootloader *Bootloader*, for purposes of this documentation, means virtually any piece of bytecode. It gains its "bootloader-like" property by the fact that it's the first piece of code that's executed by CPU core.

EVT *Exception Vector Table*

HDT *Hardware Description Table*

linker *Linker* takes an *object file* (or more, or even an archive), and creates a *binary* by merging relevant sections and by replacing symbolic references with final offsets.

ld provides this functionality.

machine For a long time, Ducky existed only as an software simulator. But, since I got that great idea about getting me a simple FPGA and learn VHDL, this may no longer be true. So, when I write about *machine*, I mean both software simulator (*VM*) *and* hardware materialization of Ducky SoC.

object file *Object file* is a file containing compiled code in a form of distinct sections of instructions, data and other necessary resources. Despite sharing their format with *binary* file, object files are usualy *not* executable because pretty much no instructions that address memory contain correct offsets, and refer to the locations using symbols. Final offsets are calculated and fixed by a *linker*.

VM *Virtual Machine*. For a long time, the only existing Ducky implementation.

CHAPTER 2

Code Documentation

2.1 ducky.boot module

This file provides necessary code to allow boot up of a virtual machine with the correct program running. This code may provide slightly different environment when compared to real hardware process, since e.g. external files can be mmap-ed into VM's memory for writing.

ducky.boot.**DEFAULT_BOOTLOADER_ADDRESS = 131072**
By default, CPU starts executing instructions at this address after boot.

ducky.boot.**DEFAULT_HDT_ADDRESS = 256**
By default, Hardware Description Table starts at this address after boot.

class ducky.boot.**MMapArea**(*ptr*, *address*, *size*, *file_path*, *offset*, *pages_start*, *pages_cnt*, *flags*)
Bases: object

Objects of this class represent one mmaped memory area each, to track this information for later use.

Parameters

- **ptr** – mmap object, as returned by mmap.mmap() function.
- **address** (*u32_t*) – address of the first byte of an area in the memory.
- **size** (*u32_t*) – length of the area, in bytes.
- **file_path** – path to a source file.
- **offset** (*u32_t*) – offset of the first byte in the source file.
- **pages_start** (*int*) – first page of the area.
- **pages_cnt** (*int*) – number of pages in the area.
- **flags** (mm.binary.SectionFlags) – flags applied to this area.

load_state(*state*)

save_state(*parent*)

class ducky.boot.**MMapAreaState**
Bases: *ducky.snapshot.SnapshotNode*

class ducky.boot.**MMapMemoryPage**(*area*, **args*, ***kwargs*)
Bases: *ducky.mm.ExternalMemoryPage*

Memory page backed by an external file that is accessible via `mmap()` call. It's a part of one of *ducky.boot.MMapArea* instances, and if such area was opened as *shared*, every change in the content of its pages will reflect onto the content of an external file, and vice versa, every change of external file will be reflected in content of this page (if this page lies in affected area).

Parameters **area** (MMapArea) – area this page belongs to.

_get_py2(*offset*)
Read one byte from page.

Parameters **offset** (*int*) – offset of the requested byte.

Return type int

_get_py3(*offset*)
Read one byte from page.

Parameters **offset** (*int*) – offset of the requested byte.

Return type int

_put_py2(*offset*, *b*)
Write one byte to page.

Parameters

- **offset** (*int*) – offset of the modified byte.
- **b** (*int*) – new value of the modified byte.

_put_py3(*offset*, *b*)
Write one byte to page.

Parameters

- **offset** (*int*) – offset of the modified byte.
- **b** (*int*) – new value of the modified byte.

get(*offset*)
Read one byte from page.

This is an abstract method, `__init__` is expected to replace it with a method, tailored for the Python version used.

Parameters **offset** (*int*) – offset of the requested byte.

Return type int

put(*offset*, *b*)
Write one byte to page.

This is an abstract method, `__init__` is expected to replace it with a method, tailored for the Python version used.

Parameters

- **offset** (*int*) – offset of the modified byte.
- **b** (*int*) – new value of the modified byte.

class ducky.boot.**ROMLoader**(*machine*)
Bases: *ducky.interfaces.IMachineWorker*

This class provides methods for loading all necessary pieces into VM's memory. These methods are called in VM's *boot* phase.

_get_mmap_fileno(*file_path*)

_put_mmap_fileno(*file_path*)

boot()

halt()

mmap_area(*file_path*, *address*, *size*, *offset=0*, *flags=None*, *shared=False*)
Assign set of memory pages to mirror external file, mapped into memory.

Parameters

- **file_path** (*string*) – path of external file, whose content new area should reflect.
- **address** (*u24*) – address where new area should start.
- **size** (*u24*) – length of area, in bytes.
- **offset** (*int*) – starting point of the area in mmaped file.
- **flags** (ducky.mm.binary.SectionFlags) – specifies required flags for mmaped pages.
- **shared** (*bool*) – if `True`, content of external file is mmaped as shared, i.e. all changes are visible to all processes, not only to the current ducky virtual machine.

Returns newly created mmap area.

Return type ducky.mm.MMapArea

Raises ***ducky.errors.InvalidResourceError*** – when `size` is not multiply of *ducky.mm.PAGE_SIZE*, or when `address` is not multiply of *ducky.mm.PAGE_SIZE*, or when any of pages in the affected area is already allocated.

poke(*address*, *value*, *length*)

setup_bootloader(*filepath*, *base=None*)
Load *bootloader* into main memory.

In the world of a real hardware, bootloader binary would be transformed into an image, and then "burned" in some form into the memory - main, or some kind of ROM from which it'd be loaded into main memory at the very beginning of boot process.

Parameters

- **filepath** (*str*) – path to bootloader binary.
- **base** (*u32_t*) – address of the first byte of bootloader in memory. By default, *ducky.boot.DEFAULT_BOOTLOADER_ADDRESS* is used.

setup_debugging()

setup_hdt()
Initialize memory area containing *Hardware Description Table*.

If VM config file specifies `HDT` image file, it is loaded, otherwise HDT is constructed for the actual configuration, and then it's copied into memory.

Parameters

- **machine.hdt-address** (*u32_t*) – Base address of HDT in memory. If not set, *ducky.boot.DEFAULT_HDT_ADDRESS* is used.
- **machine.hdt-image** (*str*) – HDT image to load. If not set, HDT is constructed for the actual VM's configuration.

setup_mmaps()

unmmap_area(*mmap_area*)

2.2 ducky.config module

VM configuration management

class ducky.config.**MachineConfig**(**args*, ***kwargs*)
Bases: ConfigParser.ConfigParser

Contains configuration of the whole VM, and provides methods for parsing, inspection and extending this configuration.

_MachineConfig__count(*prefix*)

_MachineConfig__count_breakpoints()

_MachineConfig__count_devices()

_MachineConfig__count_mmaps()

_MachineConfig__sections_with_prefix(*prefix*)

_boolean_states = {'1': True, 'on': True, 'false': False, '0': False, 'off': False, 'yes': True, 'no': False, 'true': True}

add_breakpoint(*core*, *address*, *active=None*, *flip=None*, *ephemeral=None*, *countdown=None*)

add_device(*klass*, *driver*, ***kwargs*)
Add another device to the configuration.

Parameters

- **klass** (*string*) – class of the device (klass option).
- **driver** (*string*) – device driver - dot-separated path to class (driver option).
- **kwargs** – all keyword arguments will be added to the section as device options.

add_mmap(*filepath*, *address*, *size*, *offset=None*, *access=None*, *shared=None*)

add_storage(*driver*, *sid*, *filepath=None*)
Add another storage to the configuration.

Parameters

- **driver** (*string*) – storage's driver - dot-separated path to class (driver option).
- **sid** (*int*) – storage's SID (sid options).
- **filepath** (*string*) – path to backend file, if there's any (filepath option).

create_getters(*section*)

dumps()

get(*section*, *option*, *default=None*, ***kwargs*)
Get value for an option.

Parameters

- **section** (*string*) – config section.
- **option** (*string*) – option name,
- **default** – this value will be returned, if no such option exists.

Return type string

Returns value of config option.

getbool(*section*, *option*, *default=None*)

getfloat(*section*, *option*, *default=None*)

getint(*section*, *option*, *default=None*)

iter_breakpoints()

iter_devices()

iter_mmaps()

iter_storages()

read(**args*, ***kwargs*)

set(*section*, *option*, *value*, **args*, ***kwargs*)

ducky.config.**bool2option**(*b*)

Get config-file-usable string representation of boolean value.

Parameters **b** (*bool*) – value to convert.

Return type string

Returns `yes` if input is `True`, `no` otherwise.

2.3 ducky.console module

class ducky.console.**ConsoleConnection**(*cid*, *master*, *stream_in*, *stream_out*)

Bases: `object`

boot()

die(*exc*)

execute(*cmd*)

halt()

log(*logger*, *msg*, **args*)

prompt()

read_input()

table(*table*, ***kwargs*)

write(*buff*, **args*)

writeln(*buff*, **args*)

class ducky.console.**ConsoleMaster**(*machine*)

Bases: `object`

boot()

connect(*slave*)

console_id = **0**

halt()

is_registered_command(*name*)

register_command(*name*, *callback*, **args*, ***kwargs*)

register_commands(*commands*, **args*, ***kwargs*)

unregister_command(*name*)

class ducky.console.**TerminalConsoleConnection**(*cid*, *master*)
Bases: *ducky.console.ConsoleConnection*

halt()

ducky.console.**cmd_help**(*console*, *cmd*)
List all available command and their descriptions

2.4 ducky.cpu package

2.4.1 Subpackages

ducky.cpu.coprocessor package

Submodules

ducky.cpu.coprocessor.control module

class ducky.cpu.coprocessor.control.**ControlCoprocessor**(*core*)
Bases: *ducky.interfaces.ISnapshotable*, *ducky.cpu.coprocessor.Coprocessor*

read(*r*)

read_cr0()

read_cr1()

read_cr2()

read_cr3()

write(*r*, *value*)

write_cr1(*address*)

write_cr2(*address*)

write_cr3(*value*)

class ducky.cpu.coprocessor.control.**ControlRegisters**
Bases: enum.IntEnum

CR0 = **0**

CR1 = **1**

CR2 = **2**

CR3 = **3**

_member_map_ = OrderedDict([('CR0', <ControlRegisters.CR0: 0>), ('CR1', <ControlRegisters.CR1: 1>), ('CR2', <Co

_member_names_ = ['CR0', 'CR1', 'CR2', 'CR3']

_member_type_
alias of `int`

_value2member_map_ = {0: <ControlRegisters.CR0: 0>, 1: <ControlRegisters.CR1: 1>, 2: <ControlRegisters.CR2: 2>

class ducky.cpu.coprocessor.control.**CoreFlags**
Bases: *ducky.util.Flags*

_flags = ['pt_enabled', 'jit', 'vmdebug']

_labels = 'PJV'

ducky.cpu.coprocessor.math_copro module

Stack-based coprocessor, providing several arithmetic operations with "long" numbers.

Coprocessor's instructions operates on a stack of (by default) 8 slots. Operations to move values between math stack and registers/data stack are also available.

In the following documentation several different data types are used:

- `int` - standard *word*, 32-bit wide integer
- `long` - long integer, 64-bit wide

Unless said otherwise, instruction takes its arguments from the stack, removing the values in the process, and pushes the result - if any - back on the stack.

class ducky.cpu.coprocessor.math_copro.**ADDL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'addl'

opcode = 32

operands = ['']

class ducky.cpu.coprocessor.math_copro.**DECL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Decrement top of the stack by one.

static execute(*core*, *inst*)

mnemonic = 'decl'

opcode = 31

operands = ['']

class ducky.cpu.coprocessor.math_copro.**DIVL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Divide the value below the top of the math stack by the topmost value.

static execute(*core*, *inst*)

mnemonic = 'divl'

opcode = 11

operands = ['']

class ducky.cpu.coprocessor.math_copro.**DROP**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'drop'

opcode = 23

operands = ['']

class ducky.cpu.coprocessor.math_copro.**DUP**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'dup'

opcode = 20

operands = ['']

class ducky.cpu.coprocessor.math_copro.**DUP2**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'dup2'

opcode = 21

operands = ['']

class ducky.cpu.coprocessor.math_copro.**Descriptor_MATH**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_R*

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

operands = ''

exception ducky.cpu.coprocessor.math_copro.**EmptyMathStackError**(**args*, ***kwargs*)
Bases: *ducky.errors.CoprocessorError*

Raised when operation expects at least one value on math stack but stack is empty.

exception ducky.cpu.coprocessor.math_copro.**FullMathStackError**(**args*, ***kwargs*)
Bases: *ducky.errors.CoprocessorError*

Raised when operation tries to put value on math stack but there is no empty spot available.

class ducky.cpu.coprocessor.math_copro.**INCL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Increment top of the stack by one.

static execute(*core*, *inst*)

mnemonic = 'incl'

opcode = 30

operands = ['']

class ducky.cpu.coprocessor.math_copro.**LOAD**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Merge two registers together, and make the result new TOS.

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

static execute(*core*, *inst*)

mnemonic = 'load'

opcode = 9

operands = ['r', 'r']

class ducky.cpu.coprocessor.math_copro.**LOADUW**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Take a value from register, extend it to `long`, and make the result TOS.

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'loaduw'

opcode = 5

operands = ['r']

class ducky.cpu.coprocessor.math_copro.**LOADW**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Take a value from register, extend it to `long`, and make the result TOS.

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'loadw'

opcode = 4

operands = ['r']

class ducky.cpu.coprocessor.math_copro.**MODL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'modl'

opcode = 12

operands = ['']

class ducky.cpu.coprocessor.math_copro.**MULL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Multiply two top-most numbers on the stack.

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'mull'

opcode = 10

operands = ['']

class ducky.cpu.coprocessor.math_copro.**MathCoprocessor**(*core*, **args*, ***kwargs*)
Bases: *ducky.interfaces.ISnapshotable*, *ducky.cpu.coprocessor.Coprocessor*

Coprocessor itself, includes its register set ("math stack").

Parameters **core** (ducky.cpu.CPUCore) – CPU core coprocessor belongs to.

dump_stack()
Log content of the stack using parent's DEBUG method.

extend_with_push(*u32*)

load_state(*state*)

save_state(*parent*)

sign_extend_with_push(*i32*)

class ducky.cpu.coprocessor.math_copro.**MathCoprocessorInstructionSet**
Bases: *ducky.cpu.instructions.InstructionSet*

Math coprocessor's instruction set.

instruction_set_id = 1

instructions = [<ducky.cpu.coprocessor.math_copro.ADDL object>, <ducky.cpu.coprocessor.math_copro.INCL objec

opcode_desc_map = {<MathCoprocessorOpcodes.POPW: 0>: <ducky.cpu.coprocessor.math_copro.POPW object>, <M

opcode_encoding_map = {<MathCoprocessorOpcodes.POPW: 0>: <class 'ducky.cpu.instructions.EncodingR'>, <Mat

opcodes
alias of *MathCoprocessorOpcodes*

class ducky.cpu.coprocessor.math_copro.**MathCoprocessorOpcodes**
Bases: enum.IntEnum

Math coprocessor's instruction opcodes.

ADDL = 32

DECL = 31

DIVL = 11

DROP = 23

DUP = 20

DUP2 = 21

INCL = 30

LOAD = 9

LOADUW = 5

LOADW = 4

MODL = 12

MULL = 10

POPL = 6

POPUW = 1

POPW = 0

PUSHL = 8

PUSHW = 2

SAVE = 7

SAVEW = 3

SIS = 63

SWP = 22

SYMDIVL = 13

SYMMODL = 14

UDIVL = 15

UMODL = 16

_member_map_ = OrderedDict([('POPW', <MathCoprocessorOpcodes.POPW: 0>), ('POPUW', <MathCoprocessorOpc

_member_names_ = ['POPW', 'POPUW', 'PUSHW', 'SAVEW', 'LOADW', 'LOADUW', 'POPL', 'SAVE', 'PUSHL', 'L

_member_type_
alias of `int`

_value2member_map_ = {0: <MathCoprocessorOpcodes.POPW: 0>, 1: <MathCoprocessorOpcodes.POPUW: 1>, 2: <M

class ducky.cpu.coprocessor.math_copro.**MathCoprocessorState**
Bases: *ducky.snapshot.SnapshotNode*

Snapshot node holding the state of math coprocessor.

class ducky.cpu.coprocessor.math_copro.**POPL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Pop the `long` from data stack, and make it new TOS.

static execute(*core*, *inst*)

mnemonic = 'popl'

opcode = 6

operands = ['']

class ducky.cpu.coprocessor.math_copro.**POPUW**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Pop the `int``from data stack, extend it to ``long`, and make the result TOS.

static execute(*core*, *inst*)

mnemonic = 'popuw'

opcode = 1

operands = ['']

class ducky.cpu.coprocessor.math_copro.**POPW**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Pop the `int` from data stack, extend it to `long`, and make the result TOS.

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'popw'

opcode = 0

operands = ['']

class ducky.cpu.coprocessor.math_copro.**PUSHL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Push the TOS on the data stack.

static execute(*core*, *inst*)

mnemonic = 'pushl'

opcode = 8

operands = ['']

class ducky.cpu.coprocessor.math_copro.**PUSHW**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Downsize TOS to `int`, and push the result on the data stack.

static execute(*core*, *inst*)

mnemonic = 'pushw'

opcode = 2

operands = ['']

class ducky.cpu.coprocessor.math_copro.**RegisterSet**(*core*)
Bases: *ducky.interfaces.ISnapshotable*

Math stack wrapping class. Provides basic push/pop access, and direct access to a top of the stack.

Parameters **core** (ducky.cpu.CPUCore) – CPU core registers belong to.

load_state(*state*)

pop()
Pop the top value from stack and return it.

Raises ***ducky.cpu.coprocessor.math_copro.EmptyMathStackError*** – if there are no values on the stack.

push(*v*)
Push new value on top of the stack.

Raises ***ducky.cpu.coprocessor.math_copro.FullMathStackError*** – if there is no space available on the stack.

save_state(*parent*)

tos()
Return the top of the stack, without removing it from a stack.

Raises *ducky.cpu.coprocessor.math_copro.EmptyMathStackError* – if there are no values on the stack.

tos1()
Return the item below the top of the stack, without removing it from a stack.

Raises *ducky.cpu.coprocessor.math_copro.EmptyMathStackError* – if there are no values on the stack.

class ducky.cpu.coprocessor.math_copro.**SAVE**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Store TOS in two registers.

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'save'

opcode = 7

operands = ['r', 'r']

class ducky.cpu.coprocessor.math_copro.**SAVEW**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Downsize TOS to `int`, and store the result in register.

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'savew'

opcode = 3

operands = ['r']

ducky.cpu.coprocessor.math_copro.**STACK_DEPTH = 8**
Number of available spots on the math stack.

class ducky.cpu.coprocessor.math_copro.**SWP**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'swpl'

opcode = 22

operands = ['']

class ducky.cpu.coprocessor.math_copro.**SYMDIVL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

The same operation like `DIVL` but provides symmetric results.

static execute(*core*, *inst*)

mnemonic = 'symdivl'

opcode = 13

operands = ['']

class ducky.cpu.coprocessor.math_copro.**SYMMODL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'symmodl'

opcode = 14

operands = ['']

class ducky.cpu.coprocessor.math_copro.**UDIVL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

Divide the value below the top of the math stack by the topmost value.

static execute(*core*, *inst*)

mnemonic = 'udivl'

opcode = 15

operands = ['']

class ducky.cpu.coprocessor.math_copro.**UMODL**(*instruction_set*)
Bases: *ducky.cpu.coprocessor.math_copro.Descriptor_MATH*

static execute(*core*, *inst*)

mnemonic = 'umodl'

opcode = 16

operands = ['']

Module contents

Coprocessors are intended to extend operations of CPUs. They are optional, and can cover wide range of operations, e.g. floating point arithmetic, encryption, or graphics. They are always attached to a CPU core, and may contain and use internal resources, e.g. their very own register sets, machine's memory, or their parent's caches.

class ducky.cpu.coprocessor.**Coprocessor**(*core*)
Bases: object

Base class for per-core coprocessors.

2.4.2 Submodules

ducky.cpu.instructions module

class ducky.cpu.instructions.**ADD**(*instruction_set*)
Bases: *ducky.cpu.instructions._BINOP*

static jit(*core*, *inst*)

mnemonic = 'add'

opcode = 28

operands = ['r', 'ri']

class ducky.cpu.instructions.**AND**(*instruction_set*)
Bases: *ducky.cpu.instructions._BITOP*

static jit(*core*, *inst*)

mnemonic = 'and'

opcode = 34

operands = ['r', 'ri']

class ducky.cpu.instructions.**BE**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'be'

operands = ['ri']

class ducky.cpu.instructions.**BG**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bg'

operands = ['ri']

class ducky.cpu.instructions.**BGE**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bge'

operands = ['ri']

class ducky.cpu.instructions.**BL**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bl'

operands = ['ri']

class ducky.cpu.instructions.**BLE**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'ble'

operands = ['ri']

class ducky.cpu.instructions.**BNE**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bne'

operands = ['ri']

class ducky.cpu.instructions.**BNO**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bno'

operands = ['ri']

class ducky.cpu.instructions.**BNS**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bns'

operands = ['ri']

class ducky.cpu.instructions.**BNZ**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bnz'

operands = ['ri']

class ducky.cpu.instructions.**BO**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bo'

operands = ['ri']

class ducky.cpu.instructions.**BS**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bs'

operands = ['ri']

class ducky.cpu.instructions.**BZ**(*instruction_set*)
Bases: *ducky.cpu.instructions._BRANCH*

mnemonic = 'bz'

operands = ['ri']

class ducky.cpu.instructions.**CALL**(*instruction_set*)
Bases: *ducky.cpu.instructions._JUMP*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'call'

opcode = 15

operands = ['ri']

class ducky.cpu.instructions.**CAS**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

encoding
alias of *EncodingA*

static execute(*core*, *inst*)

mnemonic = 'cas'

opcode = 7

operands = ['r', 'r', 'r']

class ducky.cpu.instructions.**CLI**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'cli'

opcode = 17

operands = []

class ducky.cpu.instructions.**CMP**(*instruction_set*)
Bases: *ducky.cpu.instructions._CMP*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'cmp'

opcode = 47

operands = ['r', 'ri']

class ducky.cpu.instructions.**CMPU**(*instruction_set*)
Bases: *ducky.cpu.instructions._CMP*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'cmpu'

opcode = 48

operands = ['r', 'ri']

class ducky.cpu.instructions.**CTR**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_R*

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'ctr'

opcode = 60

operands = ['r', 'r']

class ducky.cpu.instructions.**CTW**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_R*

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'ctw'

opcode = 61

operands = ['r', 'r']

class ducky.cpu.instructions.**DEC**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R*

static execute(*core*, *inst*)

static jit (*core*, *inst*)

mnemonic = 'dec'

opcode = 27

operands = ['r']

class ducky.cpu.instructions.**DIV** (*instruction_set*)
Bases: *ducky.cpu.instructions._BINOP*

static jit (*core*, *inst*)

mnemonic = 'div'

opcode = 31

operands = ['r', 'ri']

class ducky.cpu.instructions.**Descriptor** (*instruction_set*)
Bases: object

_expand_operands ()

static _match_operand_type (*allowed*, *operand*)

static assemble_operands (*ctx*, *inst*, *operands*)

classmethod disassemble_mnemonic (*inst*)

static disassemble_operands (*logger*, *inst*)

static emit_instruction (*ctx*, *desc*, *operands*)

encoding
alias of *EncodingR*

static execute (*core*, *inst*)

static fill_reloc_slot (*inst*, *slot*)

inst_aligned = False

static jit (*core*, *inst*)

mnemonic = None

opcode = None

operands = None

relative_address = False

class ducky.cpu.instructions.**Descriptor_R** (*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands (*ctx*, *inst*, *operands*)

static disassemble_operands (*logger*, *inst*)

encoding
alias of *EncodingR*

operands = 'r'

class ducky.cpu.instructions.**Descriptor_RI** (*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands (*ctx*, *inst*, *operands*)

static disassemble_operands (*logger*, *inst*)

encoding
alias of *EncodingI*

operands = 'ri'

class ducky.cpu.instructions.**Descriptor_R_I** (*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands (*ctx*, *inst*, *operands*)

static disassemble_operands (*logger*, *inst*)

encoding
alias of *EncodingI*

operands = 'r,i'

class ducky.cpu.instructions.**Descriptor_R_R** (*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands (*ctx*, *inst*, *operands*)

static disassemble_operands (*logger*, *inst*)

encoding
alias of *EncodingR*

operands = 'r,r'

class ducky.cpu.instructions.**Descriptor_R_RI** (*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands (*ctx*, *inst*, *operands*)

static disassemble_operands (*logger*, *inst*)

encoding
alias of *EncodingR*

operands = 'r,ri'

class ducky.cpu.instructions.**DuckyInstructionSet**
Bases: *ducky.cpu.instructions.InstructionSet*

instruction_set_id = 0

instructions = [<ducky.cpu.instructions.NOP object>, <ducky.cpu.instructions.INT object>, <ducky.cpu.instructions.

opcode_desc_map = {<DuckyOpcodes.NOP: 0>: <ducky.cpu.instructions.NOP object>, <DuckyOpcodes.LW: 1>: <duc

opcode_encoding_map = {<DuckyOpcodes.NOP: 0>: <class 'ducky.cpu.instructions.EncodingI'>, <DuckyOpcodes.LW

opcodes
alias of *DuckyOpcodes*

class ducky.cpu.instructions.**DuckyOpcodes**
Bases: enum.IntEnum

ADD = 28

AND = 34

BRANCH = 50

CALL = 15

CAS = 7

CLI = 17

CMP = 47

CMPU = 48

CTR = 60

CTW = 61

DEC = 27

DIV = 31

FPTC = 62

HLT = 20

IDLE = 21

INC = 26

INT = 13

IPI = 23

J = 46

LA = 8

LB = 3

LI = 9

LIU = 10

LPM = 22

LS = 2

LW = 1

MOD = 33

MOV = 11

MUL = 30

NOP = 0

NOT = 37

OR = 35

POP = 25

PUSH = 24

RET = 16

RETINT = 14

RST = 19

SELECT = 51

SET = 49

SHL = 38

SHR = 39

SHRS = 40

SIS = 63

STB = 6

STI = 18

STS = 5

STW = 4

SUB = 29

SWP = 12

UDIV = 32

XOR = 36

_member_map_ = OrderedDict([('NOP', <DuckyOpcodes.NOP: 0>), ('LW', <DuckyOpcodes.LW: 1>), ('LS', <DuckyOpc

_member_names_ = ['NOP', 'LW', 'LS', 'LB', 'STW', 'STS', 'STB', 'CAS', 'LA', 'LI', 'LIU', 'MOV', 'SWP', 'INT', 'R

_member_type_
alias of int

_value2member_map_ = {0: <DuckyOpcodes.NOP: 0>, 1: <DuckyOpcodes.LW: 1>, 2: <DuckyOpcodes.LS: 2>, 3: <Du

class ducky.cpu.instructions.**Encoding**
Bases: _ctypes.Structure

static repr(*inst*, *fields*)

static sign_extend_immediate(*logger*, *inst*, *sign_mask*, *ext_mask*)

class ducky.cpu.instructions.**EncodingA**
Bases: _ctypes.Structure

fields = [('opcode', <class 'ctypes.c_uint'>, 6), ('reg1', <class 'ctypes.c_uint'>, 5), ('reg2', <class 'ctypes.c_uint'>, 5),

pack = 0

opcode
Structure/Union member

reg1
Structure/Union member

reg2
Structure/Union member

reg3
Structure/Union member

class ducky.cpu.instructions.**EncodingC**
Bases: _ctypes.Structure

fields = [('opcode', <class 'ctypes.c_uint'>, 6), ('reg', <class 'ctypes.c_uint'>, 5), ('flag', <class 'ctypes.c_uint'>, 3), ('

pack = 0

static fill_reloc_slot(*inst*, *slot*)

flag
Structure/Union member

immediate
Structure/Union member

immediate_flag
Structure/Union member

opcode
Structure/Union member

reg
Structure/Union member

static sign_extend_immediate(*logger*, *inst*)

value
Structure/Union member

class ducky.cpu.instructions.**EncodingContext**(*logger*)
Bases: *ducky.util.LoggingCapable*, object

_u32_to_encoding_pypy(*u*, *encoding*)

_u32_to_encoding_python(*u*, *encoding*)

decode(*instr_set*, *inst*, *core=None*)

encode(*inst*, *field*, *size*, *value*, *raise_on_large_value=False*)

class ducky.cpu.instructions.**EncodingI**
Bases: _ctypes.Structure

fields = [('opcode', <class 'ctypes.c_uint'>, 6), ('reg', <class 'ctypes.c_uint'>, 5), ('immediate_flag', <class 'ctypes.c_

pack = 0

static fill_reloc_slot(*inst*, *slot*)

immediate
Structure/Union member

immediate_flag
Structure/Union member

opcode
Structure/Union member

reg
Structure/Union member

static sign_extend_immediate(*logger*, *inst*)

class ducky.cpu.instructions.**EncodingR**
Bases: _ctypes.Structure

fields = [('opcode', <class 'ctypes.c_uint'>, 6), ('reg1', <class 'ctypes.c_uint'>, 5), ('reg2', <class 'ctypes.c_uint'>, 5),

pack = 0

static fill_reloc_slot(*inst*, *slot*)

immediate
Structure/Union member

immediate_flag
Structure/Union member

opcode
Structure/Union member

reg1
Structure/Union member

reg2
Structure/Union member

static sign_extend_immediate(*logger*, *inst*)

class ducky.cpu.instructions.**EncodingS**
Bases: _ctypes.Structure

fields = [('opcode', <class 'ctypes.c_uint'>, 6), ('reg1', <class 'ctypes.c_uint'>, 5), ('reg2', <class 'ctypes.c_uint'>, 5),

pack = 0

static fill_reloc_slot(*inst*, *slot*)

flag
Structure/Union member

immediate
Structure/Union member

immediate_flag
Structure/Union member

opcode
Structure/Union member

reg1
Structure/Union member

reg2
Structure/Union member

static sign_extend_immediate(*logger*, *inst*)

value
Structure/Union member

class ducky.cpu.instructions.**FPTC**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'fptc'

opcode = 62

operands = []

class ducky.cpu.instructions.**HLT**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_RI*

static execute(*core*, *inst*)

mnemonic = 'hlt'

opcode = 20

operands = ['ri']

class ducky.cpu.instructions.**IDLE**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

mnemonic = 'idle'

opcode = 21

operands = []

ducky.cpu.instructions.**IE_FLAG**(*n*)

ducky.cpu.instructions.**IE_IMM**(*n*, *l*)

ducky.cpu.instructions.**IE_OPCODE**()

ducky.cpu.instructions.**IE_REG**(*n*)

class ducky.cpu.instructions.**INC**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'inc'

opcode = 26

operands = ['r']

class ducky.cpu.instructions.**INT**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_RI*

static execute(*core*, *inst*)

mnemonic = 'int'

opcode = 13

operands = ['ri']

class ducky.cpu.instructions.**IPI**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_RI*

static execute(*core*, *inst*)

mnemonic = 'ipi'

opcode = 23

operands = ['r', 'ri']

class ducky.cpu.instructions.**InstructionSet**
Bases: object

classmethod disassemble_instruction(*logger*, *inst*)

classmethod init()

instruction_set_id = None

instructions = []

opcodes = None

class ducky.cpu.instructions.**InstructionSetMetaclass**(*name*, *bases*, *dict*)
Bases: type

class ducky.cpu.instructions.**J**(*instruction_set*)
Bases: *ducky.cpu.instructions._JUMP*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'j'

opcode = 46

operands = ['ri']

ducky.cpu.instructions.**JUMP**(*core*, *inst*, *reg*)

class ducky.cpu.instructions.**LA**(*instruction_set*)
Bases: *ducky.cpu.instructions._LOAD_IMM*

static jit(*core*, *inst*)

classmethod load(*core*, *inst*)

mnemonic = 'la'

opcode = 8

operands = ['r', 'i']

relative_address = True

class ducky.cpu.instructions.**LB**(*instruction_set*)
Bases: *ducky.cpu.instructions._LOAD*

mnemonic = 'lb'

opcode = 3

operands = ['r', 'a']

class ducky.cpu.instructions.**LI**(*instruction_set*)
Bases: *ducky.cpu.instructions._LOAD_IMM*

static jit(*core*, *inst*)

classmethod load(*core*, *inst*)

mnemonic = 'li'

opcode = 9

operands = ['r', 'i']

class ducky.cpu.instructions.**LIU**(*instruction_set*)
Bases: *ducky.cpu.instructions._LOAD_IMM*

static jit(*core*, *inst*)

classmethod load(*core*, *inst*)

mnemonic = 'liu'

opcode = 10

operands = ['r', 'i']

class ducky.cpu.instructions.**LPM**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

mnemonic = 'lpm'

opcode = 22

operands = []

class ducky.cpu.instructions.**LS**(*instruction_set*)
Bases: *ducky.cpu.instructions._LOAD*

mnemonic = 'ls'

opcode = 2

operands = ['r', 'a']

class ducky.cpu.instructions.**LW**(*instruction_set*)
Bases: *ducky.cpu.instructions._LOAD*

mnemonic = 'lw'

opcode = 1

operands = ['r', 'a']

class ducky.cpu.instructions.**MOD**(*instruction_set*)
Bases: *ducky.cpu.instructions._BINOP*

static jit(*core*, *inst*)

mnemonic = 'mod'

opcode = 33

operands = ['r', 'ri']

class ducky.cpu.instructions.**MOV**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_R*

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'mov'

opcode = 11

operands = ['r', 'r']

class ducky.cpu.instructions.**MUL**(*instruction_set*)
Bases: *ducky.cpu.instructions._BINOP*

static jit(*core*, *inst*)

mnemonic = 'mul'

opcode = 30

operands = ['r', 'ri']

class ducky.cpu.instructions.**NOP**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

mnemonic = 'nop'

opcode = 0

operands = []

class ducky.cpu.instructions.**NOT**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R*

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

mnemonic = 'not'

opcode = 37

operands = ['r']

class ducky.cpu.instructions.**OR**(*instruction_set*)
Bases: *ducky.cpu.instructions._BITOP*

static jit(*core*, *inst*)

mnemonic = 'or'

opcode = 35

operands = ['r', 'ri']

class ducky.cpu.instructions.**POP**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'pop'

opcode = 25

operands = ['r']

class ducky.cpu.instructions.**PUSH**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_RI*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'push'

opcode = 24

operands = ['ri']

class ducky.cpu.instructions.**RET**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'ret'

opcode = 16

operands = []

class ducky.cpu.instructions.**RETINT**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'retint'

opcode = 14

operands = []

ducky.cpu.instructions.**RI_ADDR**(*core*, *inst*, *reg*)

ducky.cpu.instructions.**RI_VAL**(*core*, *inst*, *reg*, *sign_extend=True*)

class ducky.cpu.instructions.**RST**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

mnemonic = 'rst'

opcode = 19

operands = []

class ducky.cpu.instructions.**SELE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'sele'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELG**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selg'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELGE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selge'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELL**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'sell'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELLE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selle'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELNE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selne'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELNO**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selno'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELNS**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selns'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELNZ**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selnz'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELO**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selo'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELS**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'sels'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SELZ**(*instruction_set*)
Bases: *ducky.cpu.instructions._SELECT*

mnemonic = 'selz'

operands = ['r', 'ri']

class ducky.cpu.instructions.**SETE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'sete'

operands = ['r']

class ducky.cpu.instructions.**SETG**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setg'

operands = ['r']

class ducky.cpu.instructions.**SETGE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setge'

operands = ['r']

class ducky.cpu.instructions.**SETL**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setl'

operands = ['r']

class ducky.cpu.instructions.**SETLE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setle'

operands = ['r']

class ducky.cpu.instructions.**SETNE**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setne'

operands = ['r']

class ducky.cpu.instructions.**SETNO**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setno'

operands = ['r']

class ducky.cpu.instructions.**SETNS**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setns'

operands = ['r']

class ducky.cpu.instructions.**SETNZ**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setnz'

operands = ['r']

class ducky.cpu.instructions.**SETO**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'seto'

operands = ['r']

class ducky.cpu.instructions.**SETS**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'sets'

operands = ['r']

class ducky.cpu.instructions.**SETZ**(*instruction_set*)
Bases: *ducky.cpu.instructions._SET*

mnemonic = 'setz'

operands = ['r']

class ducky.cpu.instructions.**SHL**(*instruction_set*)
Bases: *ducky.cpu.instructions._BITOP*

static jit(*core*, *inst*)

mnemonic = 'shiftl'

opcode = 38

operands = ['r', 'ri']

class ducky.cpu.instructions.**SHR**(*instruction_set*)
Bases: *ducky.cpu.instructions._BITOP*

static jit(*core*, *inst*)

mnemonic = 'shiftr'

opcode = 39

operands = ['r', 'ri']

class ducky.cpu.instructions.**SHRS**(*instruction_set*)
Bases: *ducky.cpu.instructions._BITOP*

static jit(*core*, *inst*)

mnemonic = 'shiftrs'

opcode = 40

operands = ['r', 'ri']

class ducky.cpu.instructions.**SIS**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_RI*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'sis'

opcode = 63

operands = ['ri']

class ducky.cpu.instructions.**STB**(*instruction_set*)
Bases: *ducky.cpu.instructions._STORE*

mnemonic = 'stb'

opcode = 6

operands = ['a', 'r']

class ducky.cpu.instructions.**STI**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

encoding
alias of *EncodingI*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'sti'

opcode = 18

operands = []

class ducky.cpu.instructions.**STS**(*instruction_set*)
Bases: *ducky.cpu.instructions._STORE*

mnemonic = 'sts'

opcode = 5

operands = ['a', 'r']

class ducky.cpu.instructions.**STW**(*instruction_set*)
Bases: *ducky.cpu.instructions._STORE*

mnemonic = 'stw'

opcode = 4

operands = ['a', 'r']

class ducky.cpu.instructions.**SUB**(*instruction_set*)
Bases: *ducky.cpu.instructions._BINOP*

static jit(*core*, *inst*)

mnemonic = 'sub'

opcode = 29

operands = ['r', 'ri']

class ducky.cpu.instructions.**SWP**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_R*

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

mnemonic = 'swp'

opcode = 12

operands = ['r', 'r']

class ducky.cpu.instructions.**UDIV**(*instruction_set*)
Bases: *ducky.cpu.instructions._BINOP*

static jit(*core*, *inst*)

mnemonic = 'udiv'

opcode = 32

operands = ['r', 'ri']

ducky.cpu.instructions.**UINT20_FMT**(*i*)

class ducky.cpu.instructions.**XOR**(*instruction_set*)
Bases: *ducky.cpu.instructions._BITOP*

static jit(*core*, *inst*)

mnemonic = 'xor'

opcode = 36

operands = ['r', 'ri']

class ducky.cpu.instructions.**_BINOP**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_RI*

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

class ducky.cpu.instructions.**_BITOP**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_RI*

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

class ducky.cpu.instructions.**_BRANCH**(*instruction_set*)
Bases: *ducky.cpu.instructions._COND*

classmethod assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_mnemonic(*inst*)

static disassemble_operands(*logger*, *inst*)

encoding
alias of *EncodingC*

static execute(*core*, *inst*)

static fill_reloc_slot(*inst*, *slot*)

inst_aligned = True

static jit(*core*, *inst*)

opcode = 50

operands = 'ri'

relative_address = True

class ducky.cpu.instructions.**_CMP**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_RI*

encoding
alias of *EncodingR*

static evaluate(*core*, *x*, *y*, *signed=True*)
Compare two numbers, and update relevant flags. Signed comparison is used unless `signed` is `False`. All arithmetic flags are set to zero before the relevant ones are set.

`O` flag is reset like the others, therefore caller has to take care of it's setting if it's required to set it.

Parameters

- **x** (*u32*) – left hand number
- **y** (*u32*) – right hand number
- **signed** (*bool*) – use signed, defaults to `True`

class ducky.cpu.instructions.**_COND**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

FLAGS = ['arith_equal', 'arith_zero', 'arith_overflow', 'arith_sign', 'l', 'g']

GFLAGS = [0, 1, 2, 3]

MNEMONICS = ['e', 'z', 'o', 's', 'g', 'l']

static evaluate(*core*, *inst*)

static set_condition(*ctx*, *inst*, *flag*, *value*)

class ducky.cpu.instructions.**_JUMP**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

encoding
alias of *EncodingI*

inst_aligned = True

operands = 'ri'

relative_address = True

class ducky.cpu.instructions.**_LOAD**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

operands = 'r,a'

class ducky.cpu.instructions.**_LOAD_IMM**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor_R_I*

classmethod execute(*core*, *inst*)

classmethod load(*core*, *inst*)

class ducky.cpu.instructions.**_SELECT**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

classmethod assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_mnemonic(*inst*)

static disassemble_operands(*logger*, *inst*)

encoding
alias of *EncodingS*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

opcode = 51

operands = 'r,ri'

class ducky.cpu.instructions.**_SET**(*instruction_set*)
Bases: *ducky.cpu.instructions._COND*

classmethod assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_mnemonic(*inst*)

static disassemble_operands(*logger*, *inst*)

encoding
alias of *EncodingS*

static execute(*core*, *inst*)

opcode = 49

operands = 'r'

class ducky.cpu.instructions.**_STORE**(*instruction_set*)
Bases: *ducky.cpu.instructions.Descriptor*

static assemble_operands(*ctx*, *inst*, *operands*)

static disassemble_operands(*logger*, *inst*)

encoding
alias of *EncodingR*

static execute(*core*, *inst*)

static jit(*core*, *inst*)

operands = 'a,r'

ducky.cpu.instructions.**encoding_to_u32**(*inst*)

ducky.cpu.instructions.**get_instruction_set**(*i*, *exc=None*)

ducky.cpu.instructions.**update_arith_flags**(*core*, *reg*)
Set relevant arithmetic flags according to content of registers. Flags are set to zero at the beginning, then content of each register is examined, and `S` and `Z` flags are set.

`E` flag is not touched, `O` flag is set to zero.

Parameters **reg** (*u32_t*) – register

ducky.cpu.registers module

class ducky.cpu.registers.**RegisterSet**
Bases: `list`

class ducky.cpu.registers.**Registers**
Bases: `enum.IntEnum`

CNT = 33

FP = 30

IP = 32

R00 = 0

R01 = 1

R02 = 2

R03 = 3

R04 = 4

R05 = 5

R06 = 6

R07 = 7

R08 = 8

R09 = 9

R10 = 10

R11 = 11

R12 = 12

R13 = 13

R14 = 14

R15 = 15

R16 = 16

R17 = 17

R18 = 18

R19 = 19

R20 = 20

R21 = 21

R22 = 22

R23 = 23

R24 = 24

R25 = 25

R26 = 26

R27 = 27

R28 = 28

R29 = 29

REGISTER_COUNT = 34

REGISTER_SPECIAL = 30

SP = 31

_member_map_ = OrderedDict([('R00', <Registers.R00: 0>), ('R01', <Registers.R01: 1>), ('R02', <Registers.R02: 2>), ('

_member_names_ = ['R00', 'R01', 'R02', 'R03', 'R04', 'R05', 'R06', 'R07', 'R08', 'R09', 'R10', 'R11', 'R12', 'R13', 'R1

_member_type_
alias of int

_value2member_map_ = {0: <Registers.R00: 0>, 1: <Registers.R01: 1>, 2: <Registers.R02: 2>, 3: <Registers.R03: 3>,

2.4.3 Module contents

class ducky.cpu.**CPU**(*machine*, *cpuid*, *memory_controller*, *cores=1*)
Bases: *ducky.interfaces.ISnapshotable*, *ducky.interfaces.IMachineWorker*

boot()

die(*exc*)

halt()

load_state(*state*)

on_core_alive(*core*)
Triggered when one of cores goes alive.

on_core_halted(*core*)
Signal CPU that one of cores is no longer alive.

on_core_running(*core*)
Signal CPU that one of cores is now running.

on_core_suspended(*core*)
Signal CPU that one of cores is now suspended.

save_state(*parent*)

suspend()

wake_up()

class ducky.cpu.**CPUCore**(*coreid*, *cpu*, *memory_controller*)
Bases: *ducky.interfaces.ISnapshotable*, *ducky.interfaces.IMachineWorker*

This class represents the main workhorse, one of CPU cores. Reads instructions, executes them, has registers, caches, handles interrupts, ...

Parameters

- **coreid** (*int*) – id of this core. Usually, it's its serial number but it has no special meaning.
- **cpu** (ducky.cpu.CPU) – CPU that owns this core.
- **memory_controller** (ducky.mm.MemoryController) – use this controller to access main memory.

FP()

IP()

REG(*reg*)

SP()

_CPUCore__get_flags()

_CPUCore__set_flags(*flags*)

_enter_exception(*index*, **args*)

Prepare CPU for handling exception routine. CPU core loads new `IP` and `SP` from proper entry of EVT. Old `SP` and `FLAGS` are saved on the exception stack, and new call frame is created. Privileged mode flag is set, hardware interrupt flag is cleared.

Then, if exception provides its routine with some arguments, these arguments are pushed on the stack.

Exception stack layout then looks like this (original stack is left untouched):

```
+---------+ <= EVT SP
|   SP    |
+---------+
|  FLAGS  |
+---------+
|   IP    |
+---------+
|   FP    |
+---------+ <= FP
|   arg1  |
+---------+
|   ...   |
+---------+
|   argN  |
+---------+ <= SP
|   ...   |
+---------+
```

Parameters

- **index** (*int*) – exception ID - EVT index.
- **args** (*u32_t*) – if present, these values will be pushed onto the stack.

_exit_exception()

Restore CPU state after running an exception routine. Call frame is destroyed, registers are restored. Clearing routine arguments is responsibility of the routine.

_get_instruction_set()

_handle_exception(*exc*, *index*, **args*)

This method provides CPU exception classes with a simple recipe on how to deal with the exception:

- tell processor to start exception dance,
- if the exception is raised again, tell processor to plan double fault routine,
- and if yet another exception is raised, halt the core.

_handle_python_exception(*exc*)

_raw_pop()

Pop value from stack. 4 byte number is read from address in `SP`, then `SP` is incremented by four.

Returns popped value

Return type `u32`

_raw_push(*val*)

Push value on stack. `SP` is decremented by four, and value is written at this new address.

Parameters **val** (*u32*) – value to be pushed

_set_instruction_set(*instr_set*)

boot()

change_runnable_state(*alive=None*, *running=None*, *idle=None*)

check_protected_ins()
Raise `AccessViolationError` if core is not running in privileged mode.

This method should be used by instruction handlers that require privileged mode, e.g. protected instructions.

Raises *ducky.errors.PrivilegedInstructionError* – if the core is not in privileged mode

create_frame()
Creates new call stack frame, by performing the following operations:

- push `IP`
- push `FP`
- set `FP` to `SP` (i.e. `SP` before this method + 2 pushes)

Stack layout then looks like this:

```
+---------+
|   IPx   |
+---------+
|   FPx   |
+---------+ <= FPy
|   ...   |
+---------+ <= original SP
|   IPy   |
+---------+
|   FPy   |
+---------+ <= SP, FP
|   ...   |
+---------+
```

`FP` then points to the newly created frame, to the saved `FP` in particular, and this saved `FP` points to its predecesor, thus forming a chain.

destroy_frame()
Destroys current call stack frame by popping values from the stack, reversing the list of operations performed by *ducky.cpu.CPUCore.create_frame()*:

- pop `FP`
- pop `IP`

After this, `FP` points to the frame from which the instruction that created the currently destroyed frame was executed, and restored `IP` points to the next instruction.

Raises *InvalidFrameError* – if frame checking is enabled, current `SP` is compared with saved `FP` to see, if the stack was clean before leaving the frame. This error indicated there is some value left on stack when `ret` or `retint` were executed. Usually, this signals missing `pop` to complement one of previous ‘‘push‘‘es.

die(*exc*)

do_step(*ip*, *regset*)

flags

halt()

has_coprocessor(*name*)

init_debug_set()

instruction_set

irq(*index*)
This is a wrapper for _enter_exception, for device drivers to call when hardware interrupt arrives.

Parameters **index** (*int*) – exception ID - EVT index

load_state(*state*)

pop(**regs*)

pop_flags()

pop_frame()

push(**regs*)

push_flags()

reset(*new_ip=0*)
Reset core's state. All registers are set to zero, all flags are set to zero, except HWINT flag which is set to one, and IP is set to requested value.

Parameters **new_ip** (*u32_t*) – new IP value, defaults to zero

run()

save_state(*parent*)

step()
Perform one "step" - fetch next instruction, increment IP, and execute instruction's code (see inst_* methods)

suspend()

wake_up()

class ducky.cpu.**CPUCoreState**
Bases: *ducky.snapshot.SnapshotNode*

class ducky.cpu.**CPUState**(**fields*)
Bases: *ducky.snapshot.SnapshotNode*

get_core_state_by_id(*coreid*)

get_core_states()

class ducky.cpu.**CoreFlags**
Bases: *ducky.util.Flags*

_flags = ['privileged', 'hwint_allowed', 'equal', 'zero', 'overflow', 'sign']

_labels = 'PHEZOS'

ducky.cpu.**DEFAULT_CORE_INST_CACHE_SIZE = 256**
Default size of core instruction cache, in instructions.

ducky.cpu.**DEFAULT_EVT_ADDRESS = 0**
Default EVT address

ducky.cpu.**DEFAULT_PT_ADDRESS = 65536**
Default PT address

class ducky.cpu.**InstructionCache_Base**(*mmu*, **args*, ***kwargs*)
Bases: *ducky.util.LoggingCapable*, dict

Simple instruction cache class, based on a dictionary, with a limited size.

Parameters **core** (ducky.cpu.CPUCore) – CPU core that owns this cache.

class ducky.cpu.**InstructionCache_Full**(*mmu*, **args*, ***kwargs*)
Bases: *ducky.util.LoggingCapable*, list

Simple instruction cache class, based on a list, with unlimited size.

Parameters **core** (ducky.cpu.CPUCore) – CPU core that owns this cache.

clear()

class ducky.cpu.**InterruptVector**(*ip=0*, *sp=0*)
Bases: object

Interrupt vector table entry.

SIZE = 8

static load(*core*, *addr*)

class ducky.cpu.**MMU**(*core*, *memory_controller*)
Bases: *ducky.interfaces.ISnapshotable*

Memory management unit (aka MMU) provides a single point handling all core's memory operations. All memory reads and writes must go through this unit, which is then responsible for all translations, access control, and caching.

Parameters

- **core** (ducky.cpu.CPUCore) – parent core.
- **memory_controller** (ducky.mm.MemoryController) – memory controller that provides access to the main memory.
- **memory.force-aligned-access** (*bool*) – if set, MMU will disallow unaligned reads and writes. False by default.
- **cpu.pt-address** (*int*) – base address of page table. *ducky.cpu.DEFAULT_PT_ADDRESS* by default.
- **cpu.pt-enabled** (*bool*) – if set, CPU core will start with page table enabled. False by default.

_check_access(*access*, *addr*, *align=None*)
Check attempted access against several criteria:

- PT is enabled - disabled PT implies different set of read/write methods that don't use this method to check access
- access alignment if correct alignment is required
- privileged access implies granted access
- corresponding PTE settings

Parameters

- **access** – read, write or execute.
- **addr** (*int*) – memory address.
- **align** (*int*) – if set, operation is expected to be aligned to this boundary.

Raises

- ***ducky.errors.UnalignedAccessError*** – when unaligned access is not allowed, but requested.
- ***ducky.errors.MemoryAccessError*** – when access is denied.

_debug_wrapper_read(*reader*, **args*, ***kwargs*)

_debug_wrapper_write(*writer*, **args*, ***kwargs*)

_fetch_instr(*addr*)

Read instruction from memory. This method is responsible for the real job of fetching instructions and filling the cache.

Parameters **addr** (*u24*) – absolute address to read from

Returns instruction

Return type InstBinaryFormat_Master

_fetch_instr_jit(*addr*)

Read instruction from memory. This method is responsible for the real job of fetching instructions and filling the cache.

Parameters **addr** (*u24*) – absolute address to read from

Returns instruction

Return type InstBinaryFormat_Master

_get_pg_ops_dict(*address*)

_get_pg_ops_list(*address*)

_get_pt_enabled()

_get_pte(*addr*)

Find out PTE for particular physical address. If PTE is not in internal PTE cache, it is fetched from PTE table.

Parameters **addr** (*int*) – memory address.

_nopt_read_u16(*addr*)

_nopt_read_u32(*addr*, *not_execute=True*)

_nopt_read_u8(*addr*)

_nopt_write_u16(*addr*, *value*)

_nopt_write_u32(*addr*, *value*)

_nopt_write_u8(*addr*, *value*)

_pt_read_u16(*addr*)

_pt_read_u32(*addr*, *not_execute=True*)

_pt_read_u8(*addr*)

_pt_write_u16(*addr*, *value*)

_pt_write_u32(*addr*, *value*)

_pt_write_u8(*addr*, *value*)

_set_access_methods()
Set parent core's memory-access methods to proper shortcuts. Methods named `MEM_{IN,OUT}{8,16,32}` will be set to corresponding MMU methods.

_set_pt_enabled(*value*)

halt()

pt_enabled

release_ptes()
Clear internal PTE cache.

reset()
Reset MMU. PT will be disabled, and all internal caches will be flushed.

class ducky.cpu.**StackFrame**(*sp*, *ip*)
Bases: object

ducky.cpu.**do_log_cpu_core_state**(*core*, *logger=None*, *disassemble=True*, *inst_set=None*)
Log state of a CPU core. Content of its registers, and other interesting or useful internal variables are logged.

Parameters

- **core** (ducky.cpu.CPUCore) – core whose state should be logged.
- **logger** – called for each line of output to actualy log it. By default, core's ducky.cpu.CPUCore.DEBUG() method is used.

ducky.cpu.**log_cpu_core_state**(**args*, ***kwargs*)
This is a wrapper for ducky.cpu.do_log_cpu_core_state function. Its main purpose is to be removed when debug mode is not set, therefore all debug calls of ducky.cpu.do_log_cpu_core_state will disappear from code, making such code effectively "quiet".

2.5 ducky.debugging module

Virtual machine debugging tools - break points, watch points, etc.

Create "point" that's triggered when a condition is satisfied (e.g. processor executes instruction on specified address, memory at specified address was modified, etc. Then, create "action" (e.g. suspend core), and bind both pieces together - when point gets triggered, execute list of actions.

class ducky.debugging.**Action**(*logger*)
Bases: object

Base class of all debugging actions.

Parameters **logger** (*logging.Logger*) – logger instance used for logging.

act(*core*, *point*)
This method is called when "action" is executed. Implement it in child classes to give child actions a functionality.

Parameters

- **core** (ducky.cpu.CPUCore) – CPU core where point was triggered.
- **point** (ducky.debugging.Point) – point that was triggered.

class ducky.debugging.**BreakPoint**(*debugging_set*, *ip*, **args*, ***kwargs*)
Bases: *ducky.debugging.Point*

static create_from_config(*debugging_set*, *config*, *section*)

is_triggered(*core*)

class ducky.debugging.**DebuggingSet**(*core*)
Bases: object

_DebuggingSet__check_chain(*stage*, *chain*, *clean_triggered=False*, **args*, ***kwargs*)

add_point(*p*, *chain*)

post_memory(*address=None*, *read=None*)

post_step()

pre_memory(*address=None*, *read=None*)

pre_step()

remove_point(*p*, *chain*)

class ducky.debugging.**LogMemoryContentAction**(*logger*, *address*, *size*)
Bases: *ducky.debugging.LogValueAction*

When triggered, logs content of a specified location in memory.

Parameters

- **logger** (*logging.Logger*) – logger instance used for logging.
- **address** (*u32_t*) – memory location.
- **size** (*int*) – size of logged number, in bytes.

static create_from_config(*debugging_set*, *config*, *section*)

get_message(*core*, *point*)

get_values(*core*, *point*)

class ducky.debugging.**LogRegisterContentAction**(*logger*, *registers*)
Bases: *ducky.debugging.LogValueAction*

When triggered, logs content of a specified register.

Parameters

- **logger** (*logging.Logger*) – logger instance used for logging.
- **registers** (*list*) – list of register names.

static create_from_config(*debugging_set*, *config*, *section*)

get_message(*core*, *point*)

get_values(*core*, *point*)

class ducky.debugging.**LogValueAction**(*logger*, *size*)
Bases: *ducky.debugging.Action*

This is the base class for actions that log a numerical values.

Parameters

- **logger** (*logging.Logger*) – logger instance used for logging.
- **size** (*int*) – size of logged number, in bytes.

act(*core*, *point*)

get_message(*core*, *point*)
Return message that, formatted with output of get_values(), will be shown to user.

Parameters

- **core** (ducky.cpu.CPUCore) – core point was triggered on.
- **point** (ducky.debugging.Point) – triggered point.

Return type string

Returns information message.

get_values(*core*, *point*)
Prepare dictionary with values for message that will be shown to the user.

Parameters

- **core** (ducky.cpu.CPUCore) – core point was triggered on.
- **point** (ducky.debugging.Point) – triggered point.

Return type dict

Returns dictionary that will be passed to message `format()` method.

class ducky.debugging.**MemoryWatchPoint**(*debugging_set*, *address*, *read*, **args*, ***kwargs*)
Bases: *ducky.debugging.Point*

static create_from_config(*debugging_set*, *config*, *section*)

is_triggered(*core*, *address=None*, *read=None*)

class ducky.debugging.**Point**(*debugging_set*, *active=True*, *countdown=0*)
Bases: object

Base class of all debugging points.

Parameters

- **debugging_set** (ducky.debugging.DebuggingSet) – debugging set this point belongs to.
- **active** (*bool*) – if not `True`, point is not active and will not trigger.
- **countdown** (*int*) – if greater than zero, point has to trigger `countdown` times before its actions are executed for the first time.

is_triggered(*core*, **args*, ***kwargs*)
Test point's condition.

Parameters **core** (ducky.cpu.CPUCore) – core requesting the test.

Return type bool

Returns `True` if condition is satisfied.

class ducky.debugging.**SuspendCoreAction**(*logger*)
Bases: *ducky.debugging.Action*

If executed, this action will suspend the CPU core that triggered its parent point.

act(*core*, *point*)

static create_from_config(*debugging_set*, *config*, *section*)

ducky.debugging.**cmd_bp_active**(*console*, *cmd*)
Toggle "active" flag for a breakpoint: bp-active <id>

ducky.debugging.**cmd_bp_add_breakpoint**(*console*, *cmd*)
Create new breakpoint: bp-break <#cpuid:#coreid> <address> [active] [countdown]

ducky.debugging.**cmd_bp_add_memory_watchpoint**(*console*, *cmd*)
Create new memory watchpoint: bp-mwatch <#cpuid:#coreid> <address> [rw] [active] [countdown]'

ducky.debugging.**cmd_bp_list**(*console*, *cmd*)
List existing breakpoints

ducky.debugging.**cmd_bp_remove**(*console*, *cmd*)
Remove breakpoint: bp-remove <id>

2.6 ducky.devices package

2.6.1 Submodules

ducky.devices.keyboard module

Keyboard controller - provides events for pressed and released keys.

class ducky.devices.keyboard.**Backend**(*machine*, *name*, *mmio_address=None*, *irq=None*)
Bases: *ducky.devices.DeviceBackend*

_process_input_events()

_read_char()

boot()

static create_from_config(*machine*, *config*, *section*)

static create_hdt_entries(*logger*, *config*, *section*)

halt()

class ducky.devices.keyboard.**ControlMessages**
Bases: enum.IntEnum

CONTROL_MESSAGE_FIRST = 1024

HALT = 1025

_member_map_ = OrderedDict([('CONTROL_MESSAGE_FIRST', <ControlMessages.CONTROL_MESSAGE_FIRST:

_member_names_ = ['CONTROL_MESSAGE_FIRST', 'HALT']

_member_type_
alias of int

_value2member_map_ = {1024: <ControlMessages.CONTROL_MESSAGE_FIRST: 1024>, 1025: <ControlMessages.H

class ducky.devices.keyboard.**Frontend**(*machine*, *name*)
Bases: *ducky.devices.DeviceFrontend*

_close_input()

_handle_input_error()

_handle_raw_input()

_open_input()

boot()

static create_from_config(*machine*, *config*, *section*)

enqueue_stream(*stream*)

halt()

class ducky.devices.keyboard.**HDTEntry_Keyboard**(*logger*, *config*, *section*)
Bases: *ducky.hdt.HDTEntry_Device*

fields = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>), ('name_length', <class 'ctypes.c_ub

flags
Structure/Union member

ident
Structure/Union member

length
Structure/Union member

mmio_address
Structure/Union member

name
Structure/Union member

name_length
Structure/Union member

type
Structure/Union member

class ducky.devices.keyboard.**KeyboardMMIOMemoryPage**(*device*, **args*, ***kwargs*)
Bases: *ducky.devices.MMIOMemoryPage*

read_u8(*offset*)

class ducky.devices.keyboard.**KeyboardPorts**
Bases: enum.IntEnum

DATA = 1

LAST = 1

STATUS = 0

_member_map_ = OrderedDict([('STATUS', <KeyboardPorts.STATUS: 0>), ('DATA', <KeyboardPorts.DATA: 1>), ('LA

_member_names_ = ['STATUS', 'DATA']

_member_type_
alias of int

_value2member_map_ = {0: <KeyboardPorts.STATUS: 0>, 1: <KeyboardPorts.DATA: 1>}

ducky.devices.rtc module

class ducky.devices.rtc.**HDTEntry_RTC**(*logger*, *config*, *section*)
Bases: *ducky.hdt.HDTEntry_Device*

fields = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>), ('name_length', <class 'ctypes.c_ub

flags
Structure/Union member

ident
Structure/Union member

length
Structure/Union member

mmio_address
Structure/Union member

name
Structure/Union member

name_length
Structure/Union member

type
Structure/Union member

class ducky.devices.rtc.**RTC**(*machine*, *name*, *frequency=None*, *mmio_address=None*, *irq=None*, **args*, ***kwargs*)
Bases: *ducky.devices.Device*

boot()

static create_from_config(*machine*, *config*, *section*)

static create_hdt_entries(*logger*, *config*, *section*)

frequency

halt()

class ducky.devices.rtc.**RTCMMIOMemoryPage**(*device*, **args*, ***kwargs*)
Bases: *ducky.devices.MMIOMemoryPage*

read_u8(*offset*)

write_u8(*offset*, *value*)

class ducky.devices.rtc.**RTCPorts**
Bases: enum.IntEnum

DAY = 5

FREQUENCY = 0

HOUR = 4

MINUTE = 2

MONTH = 6

SECOND = 1

YEAR = 6

_member_map_ = OrderedDict([('FREQUENCY', <RTCPorts.FREQUENCY: 0>), ('SECOND', <RTCPorts.SECOND:

_member_names_ = ['FREQUENCY', 'SECOND', 'MINUTE', 'HOUR', 'DAY', 'MONTH']

_member_type_
alias of int

_value2member_map_ = {0: <RTCPorts.FREQUENCY: 0>, 1: <RTCPorts.SECOND: 1>, 2: <RTCPorts.MINUTE: 2>,

class ducky.devices.rtc.**RTCTask**(*machine*, *rtc*)
Bases: *ducky.reactor.RunInIntervalTask*

on_tick(*task*)

update_tick()

ducky.devices.snapshot module

class ducky.devices.snapshot.**DefaultFileSnapshotStorage**(*machine*, *name*, *filepath=None*, **args*, ***kwargs*)

Bases: *ducky.devices.snapshot.FileSnapshotStorage*

static create_from_config(*machine*, *config*, *section*)

class ducky.devices.snapshot.**FileSnapshotStorage**(*machine*, *name*, *filepath=None*, **args*, ***kwargs*)

Bases: *ducky.devices.snapshot.SnapshotStorage*

boot()

static create_from_config(*machine*, *config*, *section*)

halt()

save_snapshot(*snapshot*)

class ducky.devices.snapshot.**SnapshotStorage**(*machine*, *name*, **args*, ***kwargs*)

Bases: *ducky.devices.Device*

static create_from_config(*machine*, *config*, *section*)

halt()

save_snapshot(*snapshot*)

ducky.devices.storage module

Persistent storage support.

Several different persistent storages can be attached to a virtual machine, each with its own id. This module provides methods for manipulating their content. Storages operate with blocks of constant, standard size, though this is not a mandatory requirement - storage with different block size, or even with variable block size can be implemented.

Block IO subsystem transfers blocks between storages and VM,

ducky.devices.storage.**BIO_BUSY = 16**
Data transfer in progress.

ducky.devices.storage.**BIO_DMA = 32**
Request direct memory access - data will be transfered directly between storage and RAM..

ducky.devices.storage.**BIO_ERR = 2**
Error happened while performing the operation.

ducky.devices.storage.**BIO_RDY = 1**
Operation is completed, user can access data and/or request another operation

ducky.devices.storage.**BIO_READ = 4**
Request data read - transfer data from storage to memory.

ducky.devices.storage.**BIO_SRST = 64**
Reset BIO.

ducky.devices.storage.**BIO_USER = 108**
Flags that user can set - others are read-only.

ducky.devices.storage.**BIO_WRITE = 8**
Request data write - transfer data from memory to storage.

ducky.devices.storage.**BLOCK_SIZE = 1024**
Size of block, in bytes.

class ducky.devices.storage.**BlockIO**(*machine*, *name*, *mmio_address=None*, *irq=None*, **args*, ***kwargs*)
Bases: *ducky.devices.Device*

_flag_busy()
Signals BIO is running an operation: *BIO_BUSY* is set, and *BIO_RDY* is cleared.

_flag_error()
Signals BIO request failed: *BIO_ERR* is set, and both *BIO_RDY* and *BIO_BUSY* are cleared.

_flag_finished()
Signals BIO is ready to accept new request: *BIO_RDY* is set, and *BIO_BUSY* is cleared.

If there was an request running, it is finished now. User can queue another request, or access data in case read by the last request.

boot()

buff_to_memory(*addr*, *buff*)

static create_from_config(*machine*, *config*, *section*)

halt()

memory_to_buff(*addr*, *length*)

read_data()

reset()

select_storage(*sid*)

status_write(*value*)
Handles writes to *STATUS* register. Starts the IO requested when *BIO_READ* or *BIO_WRITE* were set.

write_data(*value*)

class ducky.devices.storage.**BlockIOMMIOMemoryPage**(*device*, **args*, ***kwargs*)
Bases: *ducky.devices.MMIOMemoryPage*

read_u32(*offset*)

write_u32(*offset*, *value*)

class ducky.devices.storage.**BlockIOPorts**
Bases: enum.IntEnum

MMIO ports, in form of offsets from a base MMIO address.

ADDR = 16
Address of a memory buffer

BLOCK = 8
Block ID

COUNT = 12
Number of blocks

DATA = 20
Data port, for non-DMA access

SID = 4
ID of selected storage device

STATUS = 0
Status port - query BIO status, and submit commands by setting flags

_member_map_ = OrderedDict([('STATUS', <BlockIOPorts.STATUS: 0>), ('SID', <BlockIOPorts.SID: 4>), ('BLOCK', <

_member_names_ = ['STATUS', 'SID', 'BLOCK', 'COUNT', 'ADDR', 'DATA']

_member_type_
alias of `int`

_value2member_map_ = {0: <BlockIOPorts.STATUS: 0>, 4: <BlockIOPorts.SID: 4>, 8: <BlockIOPorts.BLOCK: 8>, 1

class ducky.devices.storage.**FileBackedStorage**(*machine*, *name*, *filepath=None*, **args*, ***kwargs*)
Bases: *ducky.devices.storage.Storage*

Storage that saves its content into a regular file.

_read(*cnt*)

_write(*buff*)

boot()

static create_from_config(*machine*, *config*, *section*)

do_read_blocks(*start*, *cnt*)

do_write_blocks(*start*, *cnt*, *buff*)

halt()

class ducky.devices.storage.**Storage**(*machine*, *name*, *sid=None*, *size=None*, **args*, ***kwargs*)
Bases: *ducky.devices.Device*

Base class for all block storages.

Parameters

- **machine** (ducky.machine.Machine) – machine storage is attached to.
- **sid** (*int*) – id of storage.
- **size** (*int*) – size of storage, in bytes.

do_read_blocks(*start*, *cnt*)
Read one or more blocks from device to internal buffer.

Child classes are supposed to reimplement this particular method.

Parameters

- **start** (*u32_t*) – index of the first requested block.
- **cnt** (*u32_t*) – number of blocks to read.

do_write_blocks(*start*, *cnt*, *buff*)
Write one or more blocks from internal buffer to device.

Child classes are supposed to reimplement this particular method.

Parameters

- **start** (*u32_t*) – index of the first requested block.
- **cnt** (*u32_t*) – number of blocks to write.

read_blocks(*start*, *cnt*)
Read one or more blocks from device to internal buffer.

Child classes should not reimplement this method, as it provides checks common for (probably) all child classes.

Parameters

- **start** (*u32_t*) – index of the first requested block.
- **cnt** (*u32_t*) – number of blocks to read.

write_blocks(*start*, *cnt*, *buff*)
Write one or more blocks from internal buffer to device.

Child classes should not reimplement this method, as it provides checks common for (probably) all child classes.

Parameters

- **start** (*u32_t*) – index of the first requested block.
- **cnt** (*u32_t*) – number of blocks to write.

exception ducky.devices.storage.**StorageAccessError**
Bases: exceptions.Exception

Base class for storage-related exceptions.

ducky.devices.terminal module

Terminal is a device that groups together character two input and output devices, thus forming a simple channel for bidirectional communication between VM and user.

Terminal has two slave frontends:

- *input*, usually a keyboard
- *output*, from simple TTY to more powerful devices

Terminal then manages input and input streams, passing them to its slave devices, which then transports events between streams and VM's comm channel.

class ducky.devices.terminal.**StandalonePTYTerminal**(**args*, ***kwargs*)
Bases: *ducky.devices.terminal.StreamIOTerminal*

boot()

static create_from_config(*machine*, *config*, *section*)

halt()

class ducky.devices.terminal.**StandardIOTerminal**(*machine*, *name*, *input_device=None*, *output_device=None*, **args*, ***kwargs*)
Bases: *ducky.devices.terminal.StreamIOTerminal*

static create_from_config(*machine*, *config*, *section*)

class ducky.devices.terminal.**StreamIOTerminal**(*machine*, *name*, *input_device=None*, *output_device=None*, **args*, ***kwargs*)
Bases: *ducky.devices.terminal.Terminal*

boot()

static create_from_config(*machine*, *config*, *section*)

enqueue_input_stream(*stream*)

enqueue_streams(*streams_in=None*, *stream_out=None*)

halt()

class ducky.devices.terminal.**Terminal**(*machine*, *name*, *echo=False*, **args*, ***kwargs*)
Bases: *ducky.devices.DeviceFrontend*

_input_read_u8_echo(**args*, ***kwargs*)

_patch_echo(*restore=False*)

boot()

halt()

ducky.devices.terminal.**get_slave_devices**(*machine*, *config*, *section*)

ducky.devices.terminal.**parse_io_streams**(*machine*, *config*, *section*)

ducky.devices.tty module

Very simple character device that just "prints" characters on the screen. It does not care about dimensions of the display, it kknow only how to "print" characters. Suited for the most basic output possible - just "print" chars by writing to this device, and you'll get this written into a stream attached to the frontend (`stdout`, file, ...).

class ducky.devices.tty.**Backend**(*machine*, *name*, *stream=None*, *mmio_address=None*, **args*, ***kwargs*)
Bases: *ducky.devices.DeviceBackend*

boot()

static create_from_config(*machine*, *config*, *section*)

static create_hdt_entries(*logger*, *config*, *section*)

halt()

tenh(*s*, **args*)

tenh_close_stream()

tenh_enable()

tenh_flush_stream()

class ducky.devices.tty.**Frontend**(*machine*, *name*)
Bases: *ducky.devices.DeviceFrontend*

boot()

close(*allow=False*)

static create_from_config(*machine*, *config*, *section*)

flush()

halt()

set_output(*stream*)

sleep_flush()

tenh_enable()

wakeup_flush()

class ducky.devices.tty.**FrontendFlushTask**(*frontend*, *queue*, *stream*)
Bases: *ducky.interfaces.IReactorTask*

run()

set_output(*stream*)

class ducky.devices.tty.**HDTEntry_TTY**(*logger*, *config*, *section*)
Bases: *ducky.hdt.HDTEntry_Device*

fields = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>), ('name_length', <class 'ctypes.c_ub

flags
Structure/Union member

ident
Structure/Union member

length
Structure/Union member

mmio_address
Structure/Union member

name
Structure/Union member

name_length
Structure/Union member

type
Structure/Union member

class ducky.devices.tty.**TTYMMIOMemoryPage**(*device*, **args*, ***kwargs*)
Bases: *ducky.devices.MMIOMemoryPage*

write_u8(*offset*, *value*)

class ducky.devices.tty.**TTYPorts**
Bases: enum.IntEnum

DATA = 0

_member_map_ = OrderedDict([('DATA', <TTYPorts.DATA: 0>)])

_member_names_ = ['DATA']

_member_type_
alias of int

_value2member_map_ = {0: <TTYPorts.DATA: 0>}

ducky.devices.svga module

SimpleVGA is very basic implementation of VGA-like device, with text and graphic modes.

class ducky.devices.svga.**Char**
Bases: _ctypes.Structure

fields = [('codepoint', <class 'ctypes.c_ushort'>, 7), ('unused', <class 'ctypes.c_ushort'>, 1), ('fg', <class 'ctypes.c_us

pack = 0

bg
Structure/Union member

blink
Structure/Union member

codepoint
Structure/Union member

fg
Structure/Union member

static from_u16(*u*)

static from_u8(*l*, *h*)

to_u8()

unused
Structure/Union member

ducky.devices.svga.**DEFAULT_BOOT_MODE = (t, 80, 25, 1)**
Default boot mode

ducky.devices.svga.**DEFAULT_MEMORY_BANKS = 8**
Default number of memory banks

ducky.devices.svga.**DEFAULT_MEMORY_SIZE = 65536**
Default memory size, in bytes

ducky.devices.svga.**DEFAULT_MMIO_ADDRESS = 33024**
Default MMIO address

ducky.devices.svga.**DEFAULT_MODES = [(g, 320, 200, 1), (t, 80, 25, 2), (t, 80, 25, 1)]**
Default list of available modes

class ducky.devices.svga.**Display**(*machine*, *name*, *gpu=None*, *stream_out=None*, **args*, ***kwargs*)
Bases: *ducky.devices.Device*

boot()

static create_from_config(*machine*, *config*, *section*)

static get_slave_gpu(*machine*, *config*, *section*)

halt()

class ducky.devices.svga.**DisplayRefreshTask**(*display*)
Bases: *ducky.reactor.RunInIntervalTask*

on_tick(*task*)

class ducky.devices.svga.**Mode**(*_type*, *width*, *height*, *depth*)
Bases: object

classmethod from_string(*s*)
Create Mode object from its string representation. It's a comma-separated list of for items:

	type	width	height	depth
Text mode	t	chars per line	lines on screen	bytes per char
Graphic mode	g	pixels per line	pixel lines on screen	bites per pixel

to_pretty_string()

to_string()

class ducky.devices.svga.**SimpleVGA**(*machine*, *name*, *memory_size=None*, *mmio_address=None*, *memory_address=None*, *memory_banks=None*, *modes=None*, *boot_mode=None*, **args*, ***kwargs*)

Bases: *ducky.devices.Device*

SimpleVGA is very basic implementation of VGA-like device, with text and graphic modes.

It has its own graphic memory ("buffer"), split into several banks of the same size. Always only one bank can be directly accessed, by having it mapped into CPU's address space.

Parameters

- **machine** (*ducky.machine.Machine*) – machine this device belongs to.
- **name** (*string*) – name of this device.
- **memory_size** (*int*) – size of graphic memory.
- **mmio_address** (*u32_t*) – base oddress of MMIO ports.
- **memory_address** (*u24*) – address of graphic memory - to this address is graphic buffer mapped. Must be specified, there is no default value.
- **memory_banks** (*int*) – number of memory banks.
- **modes** (*list*) – list of *ducky.devices.svga.Mode* objects, list of supported modes.
- **boot_mode** (*tuple*) – this mode will be set when device boots up.

boot()

static create_from_config(*machine*, *config*, *section*)

halt()

reset()

set_mode(*mode*)

class ducky.devices.svga.**SimpleVGACommands**

Bases: enum.IntEnum

COLS = 33

DEPTH = 35

GRAPHIC = 32

MEMORY_BANK_ID = 48

REFRESH = 2

RESET = 1

ROWS = 34

_member_map_ = OrderedDict([('RESET', <SimpleVGACommands.RESET: 1>), ('REFRESH', <SimpleVGACommand

_member_names_ = ['RESET', 'REFRESH', 'GRAPHIC', 'COLS', 'ROWS', 'DEPTH', 'MEMORY_BANK_ID']

_member_type_

alias of int

_value2member_map_ = {32: <SimpleVGACommands.GRAPHIC: 32>, 1: <SimpleVGACommands.RESET: 1>, 2: <S

class ducky.devices.svga.**SimpleVGAMMIOMemoryPage**(*device*, **args*, ***kwargs*)

Bases: *ducky.devices.MMIOMemoryPage*

read_u16(*offset*)

write_u16(*offset*, *value*)

class ducky.devices.svga.**SimpleVGAMemoryPage**(*dev*, **args*, ***kwargs*)
Bases: *ducky.mm.ExternalMemoryPage*

Memory page handling MMIO of sVGA device.

Parameters **dev** (ducky.devices.svga.SimpleVGA) – sVGA device this page belongs to.

get(*offset*)

put(*offset*, *b*)

class ducky.devices.svga.**SimpleVGAPorts**
Bases: enum.IntEnum

CONTROL = 0

DATA = 2

_member_map_ = OrderedDict([('CONTROL', <SimpleVGAPorts.CONTROL: 0>), ('DATA', <SimpleVGAPorts.DATA:

_member_names_ = ['CONTROL', 'DATA']

_member_type_
alias of int

_value2member_map_ = {0: <SimpleVGAPorts.CONTROL: 0>, 2: <SimpleVGAPorts.DATA: 2>}

2.6.2 Module contents

Emulating different virtual devices like keyboard, display, disks...

class ducky.devices.**Device**(*machine*, *klass*, *name*)
Bases: *ducky.interfaces.IMachineWorker*

Base class for all devices. Serves more like an API description.

Parameters

- **machine** (ducky.machine.Machine) – VM this device belongs to.
- **klass** (*str*) – device family (input, output, snapshot, ...)
- **name** (*str*) – device name. Maps directly to a section of config file that hosts setup for this device.

boot()

static create_from_config(*machine*, *config*, *section*)
Create new instance, configured exactly as requested by configuration file.

Parameters

- **machine** (ducky.machine.Machine) – VM this device belongs to.
- **config** (ducky.config.MachineConfig) – configuration file.
- **section** (*str*) – name of config section with this device's setup.

static create_hdt_entries(*logger*, *config*, *section*)
Create HDT entries for this device, based on configuration file.

Parameters

- **logger** (*logging.Logger*) – logger to use for logging.

- **config** (ducky.config.MachineConfig) – configuration file.
- **section** (*str*) – name of config section with this device's setup.

Return type list of *ducky.hdt.HDTEntry*

Returns list of HDT entries. This list is then appended to entries of other devices.

get_master()

halt()

is_slave()

class ducky.devices.**DeviceBackend**(*machine*, *klass*, *name*)
Bases: *ducky.devices.Device*

Backend is a special component of a device, which is connected to internal queues, and processes events in the queue, originating from user via its frontend counterpart.

set_frontend(*device*)
Set frontend counterpart.

Parameters **device** (*ducky.device.DeviceFrontend*) – frontend part.

class ducky.devices.**DeviceFrontend**(*machine*, *klass*, *name*)
Bases: *ducky.devices.Device*

Frontend is a special component of a device, that interfaces communication channels from user and internal queues.

set_backend(*device*)
Set backend counterpart.

Parameters **device** (ducky.devices.DeviceFrontend) – backend part.

class ducky.devices.**MMIOMemoryPage**(*device*, **args*, ***kwargs*)
Bases: *ducky.mm.VirtualMemoryPage*

Memory page, suited for memory-mapped IO, supported by a device driver.

Parameters **device** (ducky.devices.Device) – device instance, backing this memory page.

ducky.devices.**get_driver**(*driver_class*)
Get Python class, implementing device driver, by its name.

Parameters **driver_class** (*str*) – path to a class, e.g. ducky.devices.rtc.RTC.

Returns driver class.

2.7 ducky.errors module

exception ducky.errors.**AccessViolationError**(*message=None*)
Bases: *ducky.errors.Error*

Raised when an operation was requested without having adequate permission to do so. message attribute will provide better idea about the fault.

exception ducky.errors.**AssemblerError**(*location=None*, *message=None*, *line=None*, *info=None*)
Bases: *ducky.errors.Error*

Base class for all assembler-related exceptions. Provides common properties, helping to locate related input in the source file.

Parameters

- **location** (*ducky.cpu.assembly.SourceLocation*) – if set, points to the location in the source file that was processed when the exception occured.
- **message** (*str*) – more detailed description.
- **line** (*str*) – input source line.
- **info** – additional details of the exception. This value is usually part of the `message`, but is stored as well.

create_text()

log(*logger*)

exception ducky.errors.**AssemblyIllegalCharError**(*c=None*, ***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**AssemblyParseError**(*token=None*, ***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**BadLinkerScriptError**(*script*, *exc*, **args*, ***kwargs*)
Bases: *ducky.errors.LinkerError*

exception ducky.errors.**ConflictingNamesError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**CoprocessorError**(*msg*, **args*, ***kwargs*)
Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Base class for coprocessor errors. Raised when coprocessors needs to signal its own exception, when none of alread yavailable exceptions would do.

EXCEPTION_INDEX = 25

exception ducky.errors.**DisassembleMismatchError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**DivideByZeroError**(**args*, ***kwargs*)
Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when divisor in a mathematical operation was equal to zero.

EXCEPTION_INDEX = 18

exception ducky.errors.**EncodingLargeValueError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**Error**(*message=None*)
Bases: `exceptions.Exception`

Base class for all Ducky exceptions.

Parameters **message** (*str*) – optional description.

log(*logger*)

class ducky.errors.**ExceptionList**
Bases: `enum.IntEnum`

List of exception IDs (`EVT` indices).

COUNT = 32

CoprocessorError = 25

DivideByZero = 18

DoubleFault = 21

FIRST_HW = 0

FIRST_SW = 16

InvalidException = 24

InvalidInstSet = 17

InvalidOpcode = 16

LAST_HW = 15

LAST_SW = 31

MemoryAccess = 22

PrivilegedInstr = 20

RegisterAccess = 23

UnalignedAccess = 19

_member_map_ = OrderedDict([('FIRST_HW', <ExceptionList.FIRST_HW: 0>), ('LAST_HW', <ExceptionList.LAST_

_member_names_ = ['FIRST_HW', 'LAST_HW', 'InvalidOpcode', 'InvalidInstSet', 'DivideByZero', 'UnalignedAccess'

_member_type_
alias of `int`

_value2member_map_ = {0: <ExceptionList.FIRST_HW: 0>, 32: <ExceptionList.COUNT: 32>, 15: <ExceptionList.LA

exception ducky.errors.**ExecutionException**(*msg*, *core=None*, *ip=None*)
Bases: exceptions.Exception

Base class for all execution-related exceptions, i.e. exceptions raised as a direct result of requests made by running code. Runnign code can then take care of handling these exceptions, usually by preparing service routines and setting up the `EVT`.

Unless said otherwise, the exception is always raised *before* making any changes in the VM state.

Parameters

- **msg** (*string*) – message describing exceptional state.
- **core** (ducky.cpu.CPUCore) – CPU core that raised exception, if any.
- **ip** (*u32_t*) – address of an instruction that caused exception, if any.

runtime_handle()
This method is called by CPU code, to find out if it is possible for runtime to handle the exception, and possibly recover from it. If it is possible, this method should make necessary arrangements, and then return `True`. Many exceptions, e.g. when division by zero was requested, will tell CPU to run exception service routine, and by returning `True` signal that it's not necessary to take care of such exception anymore.

Return type bool

Returns `True` when it's no longer necessary for VM code to take care of this exception.

class ducky.errors.**ExecutionException__SimpleESR**
Bases: object

Helper mixin class - as one of parents, it brings to its children very simle - and most of the time sufficient - implementation of *runtime_handle* method. Such exceptions will tell CPU to run exception service routine with a secific index, specified by class variable `EXCEPTION_INDEX`.

The address of the offensive instruction - or the value of `IP` when exception was raised, since there may be exceptions not raised in response to the executed instruction - is passed to CPU as the first argument of ESR.

runtime_handle()

exception ducky.errors.**IncompatibleSectionFlagsError**(*dst_section*, *src_section*, **args*, ***kwargs*)

Bases: *ducky.errors.LinkerError*

exception ducky.errors.**IncompleteDirectiveError**(***kwargs*)

Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**InvalidExceptionError**(*exc_index*, **args*, ***kwargs*)

Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when requested exception index is invalid (out of bounds).

EXCEPTION_INDEX = 24

exception ducky.errors.**InvalidFrameError**(*saved_sp*, *current_sp*, **args*, ***kwargs*)

Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when VM checks each call stack frame to be cleaned before it's left, and there are some values on stack when `ret` or `retint` are executed. In such case, the actual `SP` values does not equal to a saved one, and exception is raised.

Parameters

- **saved_sp** (*u32_t*) – `SP` as saved at the moment the frame was created.
- **current_sp** (*u32_t*) – current `SP`

runtime_handle()

exception ducky.errors.**InvalidInstructionSetError**(*inst_set*, **args*, ***kwargs*)

Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when switch to unknown or invalid instruction set was requested.

Parameters **inst_set** (*int*) – instruction set id.

EXCEPTION_INDEX = 17

exception ducky.errors.**InvalidOpcodeError**(*opcode*, **args*, ***kwargs*)

Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when unknown or invalid opcode is found in instruction.

Parameters **opcode** (*int*) – wrong opcode.

EXCEPTION_INDEX = 16

exception ducky.errors.**InvalidResourceError**(*message=None*)

Bases: *ducky.errors.Error*

Raised when an operation was requested on somehow invalid resource. `message` attribute will provide better idea about the fault.

exception ducky.errors.**LinkerError**(*message=None*)
Bases: *ducky.errors.Error*

exception ducky.errors.**MemoryAccessError**(*access*, *address*, *pte*, **args*, ***kwargs*)
Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when MMU decides the requested memory operation is now allowed, e.g. when page tables are enabled, and corresponding `PTE` denies write access.

Parameters

- **access** (*str*) – `read`, `write` or `execute`.
- **address** (*u32_t*) – address where memory access shuld have happen.
- **pte** (*ducky.mm.PageTableEntry*) – `PTE` guarding this particular memory location.

EXCEPTION_INDEX = 22

exception ducky.errors.**OperandMismatchError**(*instr*, *expected*, *actual*)
Bases: *ducky.errors.Error*

exception ducky.errors.**PatchTooLargeError**(*message=None*)
Bases: *ducky.errors.LinkerError*

exception ducky.errors.**PrivilegedInstructionError**(**args*, ***kwargs*)
Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when privileged instruction was about to be executed in non-privileged mode.

EXCEPTION_INDEX = 20

exception ducky.errors.**RegisterAccessError**(*access*, *reg*, **args*, ***kwargs*)
Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when instruction tries to access a registerall which is not available for requested operation, e.g. writing into read-only control register will raise this exception.

Parameters

- **access** (*str*) – `read` or `write`.
- **reg** (*str*) – register name.

EXCEPTION_INDEX = 23

exception ducky.errors.**TooManyLabelsError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**UnalignedAccessError**(**args*, ***kwargs*)
Bases: *ducky.errors.ExecutionException__SimpleESR*, *ducky.errors.ExecutionException*

Raised when only properly aligned memory access is allowed, and running code attempts to access memory without honoring this restriction (e.g. `LW` reading from byte-aligned address).

EXCEPTION_INDEX = 19

exception ducky.errors.**UnalignedJumpTargetError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**UnknownDestinationSectionError**(*src_section*, **args*, ***kwargs*)
Bases: *ducky.errors.LinkerError*

exception ducky.errors.**UnknownFileError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**UnknownInstructionError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**UnknownPatternError**(***kwargs*)
Bases: *ducky.errors.AssemblerError*

exception ducky.errors.**UnknownSymbolError**(*message=None*)
Bases: *ducky.errors.LinkerError*

2.8 ducky.hdt module

Hardware Description Table structures.

class ducky.hdt.**HDT**(*logger*, *config=None*)
Bases: object

Root of HDT. Provides methods for creating HDT for a given machine configuration.

Parameters

- **logger** (*logging.Logger*) – logger instance used for logging.
- **config** (ducky.config.MachineConfig) – configuration file HDT should reflect.

create()
Create HDT header and entries from config file.

klasses = [<class 'ducky.hdt.HDTEntry_Memory'>, <class 'ducky.hdt.HDTEntry_CPU'>, <class 'ducky.hdt.HDTEntry
These HDT entries are added automatically.

class ducky.hdt.**HDTEntry**(*entry_type*, *length*)
Bases: *ducky.hdt.HDTStructure*

Base class of all HDT entries.

Each entry has at least two fields, *type* and *length*. Other fields depend on type of entry, and follow immediately after *length* field.

Parameters

- **type** (*u16_t*) – type of entry. See *ducky.hdt.HDTEntryTypes*.
- **length** (*u16_t*) – length of entry, in bytes.

ENTRY_HEADER = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>)]

classmethod create(**args*, ***kwargs*)

class ducky.hdt.**HDTEntryTypes**
Bases: enum.IntEnum

Types of different HDT entries.

ARGUMENT = 3

CPU = 1

DEVICE = 4

MEMORY = 2

UNDEFINED = 0

_member_map_ = OrderedDict([('UNDEFINED', <HDTEntryTypes.UNDEFINED: 0>), ('CPU', <HDTEntryTypes.CPU

_member_names_ = ['UNDEFINED', 'CPU', 'MEMORY', 'ARGUMENT', 'DEVICE']

_member_type_
alias of `int`

_value2member_map_ = {0: <HDTEntryTypes.UNDEFINED: 0>, 1: <HDTEntryTypes.CPU: 1>, 2: <HDTEntryTypes

class ducky.hdt.**HDTEntry_Argument**(*arg_name*, *arg_type*, *arg_value*)
Bases: *ducky.hdt.HDTEntry*

MAX_NAME_LENGTH = 13

fields = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>), ('name_length', <class 'ctypes.c_ub

classmethod create(*logger*, *config*)

length
Structure/Union member

name
Structure/Union member

name_length
Structure/Union member

type
Structure/Union member

value
Structure/Union member

value_length
Structure/Union member

class ducky.hdt.**HDTEntry_CPU**(*logger*, *config*)
Bases: *ducky.hdt.HDTEntry*

HDT entry describing CPU configuration.

Parameters

- **nr_cpus** (*u16_t*) – number of CPUs.
- **nr_cores** (*u16_t*) – number of cores per CPU.

fields = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>), ('nr_cpus', <class 'ctypes.c_ushort'

length
Structure/Union member

nr_cores
Structure/Union member

nr_cpus
Structure/Union member

type
Structure/Union member

class ducky.hdt.**HDTEntry_Device**(*logger*, *name*, *ident*)
Bases: *ducky.hdt.HDTEntry*

ENTRY_HEADER = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>), ('name_length', <class 'ctypes

MAX_IDENT_LENGTH = 32

MAX_NAME_LENGTH = 10

class ducky.hdt.**HDTEntry_Memory**(*logger*, *config*)
Bases: *ducky.hdt.HDTEntry*

HDT entry describing memory configuration.

Parameters **size** (*u32_t*) – size of memory, in bytes.

fields = [('type', <class 'ctypes.c_ushort'>), ('length', <class 'ctypes.c_ushort'>), ('size', <class 'ctypes.c_uint'>)]

length
Structure/Union member

size
Structure/Union member

type
Structure/Union member

class ducky.hdt.**HDTHeader**
Bases: *ducky.hdt.HDTStructure*

HDT header. Contains magic number, number of HDT entries that immediately follow header.

fields = [('magic', <class 'ctypes.c_uint'>), ('entries', <class 'ctypes.c_uint'>), ('length', <class 'ctypes.c_uint'>)]

entries
Structure/Union member

length
Structure/Union member

magic
Structure/Union member

class ducky.hdt.**HDTStructure**
Bases: _ctypes.Structure

Base class of all HDT structures.

pack = 0

ducky.hdt.**HDT_MAGIC = 1298034544**
Magic number present in HDT header

ducky.hdt.**encode_string**(*struct*, *field*, *s*, *max_length*)
Store string in a structure's field, and set the corresponding length field properly.

Parameters

- **struct** (HDTStructure) – structure to modify.
- **field** (*str*) – name of field the string should be stored in.
- **s** (*str*) – string to encode.
- **max_length** (*int*) – maximal number of bytes that can fit into the field.

2.9 ducky.interfaces module

class ducky.interfaces.**IMachineWorker**
Bases: object

Base class for objects that provide pluggable service to others.

boot(**args*)
Prepare for providing the service. After this call, it may be requested by others.

die(*exc*)
Exceptional state requires immediate termination of service. Probably no object will ever have need to call others' `die` method, it's intended for internal use only.

halt()
Terminate service. It will never be requested again, object can destroy its internal state, and free allocated resources.

run()
Called by reactor's loop when this object is enqueued as a reactor task.

suspend()
Suspend service. Object should somehow conserve its internal state, its service will not be used until the next call of `wake_up` method.

wake_up()
Wake up service. In this method, object should restore its internal state, and after this call its service can be requested by others again.

class ducky.interfaces.**IReactorTask**
Bases: `object`

Base class for all reactor tasks.

run()
This method is called by reactor to perform task's actions.

class ducky.interfaces.**ISnapshotable**
Bases: `object`

Base class for objects that can be saved into a snapshot.

load_state(*state*)
Restore state of the object.

Parameters **state** (ducky.snapshot.SnapshotNode) – snapshot node containing saved state.

save_state(*parent*)
Create state of the object, and attach it to a parent snapshot node.

Parameters **parent** (ducky.interfaces.ISnapshotable) – parent snapshot node.

class ducky.interfaces.**IVirtualInterrupt**(*machine*)
Bases: `object`

run(*core*)

2.10 ducky.log module

class ducky.log.**ColorizedLogFormatter**(*format=None*)
Bases: *ducky.log.LogFormatter*

_default_format = '{color_start}[{level}]{color_end} {message}'

_get_vars(*record*)

blue(*s*)

colorize(*s*, *fore='\x1b[39m\x1b[49m\x1b[0m'*, *back='\x1b[39m\x1b[49m\x1b[0m'*)

green(*s*)

red(*s*)

white(*s*)

class ducky.log.**LogFormatter**(*format=None*)
Bases: logging.Formatter

_default_format = '[{level}] {message}'

_get_vars(*record*)

blue(*s*)

colorize(*s*, **args*, ***kwargs*)

double_fault(*exc*, *record*)

format(*record*)

green(*s*)

red(*s*)

white(*s*)

class ducky.log.**StreamHandler**(*formatter=None*, **args*, ***kwargs*)
Bases: logging.StreamHandler

ducky.log.**create_logger**(*name=None*, *handler=None*, *level=20*)

2.11 ducky.machine module

ducky.machine.Machine is *the* virtual machine. Each instance represents self-contained virtual machine, with all its devices, memory, CPUs and other necessary properties.

class ducky.machine.**CommChannel**(*machine*)
Bases: object

create_queue(*name*)

get_queue(*name*)

unregister_queue(*name*)

class ducky.machine.**CommQueue**(*channel*)
Bases: object

is_empty_in()

is_empty_out()

read_in()

read_out()

write_in(*o*)

write_out(*o*)

class ducky.machine.**EventBus**(*machine*)
Bases: object

add_listener(*event*, *callback*, **args*, ***kwargs*)

remove_listener(*event*, *callback*)

trigger(*event*, **args*, ***kwargs*)

class ducky.machine.**HaltMachineTask**(*machine*)
Bases: *ducky.interfaces.IReactorTask*

run()

class ducky.machine.**IRQRouterTask**(*machine*)
Bases: *ducky.interfaces.IReactorTask*

This task is responsible for distributing triggered IRQs between CPU cores. When IRQ is triggered, IRQ source (i.e. device that requires attention) is appended to this tasks queue (ducky.machine.IRQRouterTask.qeueu). As long as this queue is not empty, this task pops IRQ sources, selects free CPU core, and by calling its *ducky.cpu.CPUCore.irq()* method core takes reponsibility for executing interrupt routine.

Parameters **machine** (ducky.machine.Machine) – machine this task belongs to.

run()

class ducky.machine.**Machine**(*logger=None*, *stdin=None*, *stdout=None*, *stderr=None*)
Bases: *ducky.interfaces.ISnapshotable*, *ducky.interfaces.IMachineWorker*

Virtual machine itself.

_do_tenh(*printer*, *s*, **args*)

boot()

capture_state(*suspend=False*)
Capture current state of the VM, and store it in it's *last_state* attribute.

Parameters **suspend** (*bool*) – if *True*, suspend VM before taking snapshot.

core(*cid*)
Find CPU core by its string id.

Parameters **cid** (*string*) – id of searched CPU core, in the form *#<cpuid>:#<coreid>*.

Return type *ducky.cpu.CPUCore*

Returns found core

Raises ***ducky.errors.InvalidResourceError*** – when no such core exists.

cores
Get list of all cores in the machine.

Return type list

Returns *list* of *ducky.cpu.CPUCore* instances

die(*exc*)

exit_code

get_device_by_name(*name*, *klass=None*)
Get device by its name and class.

Parameters

- **name** (*string*) – name of the device.

- **klass** (*string*) – if set, search only devices with this class.

Return type *ducky.devices.Device*

Returns found device

Raises ***ducky.errors.InvalidResourceError*** – when no such device exists

get_storage_by_id(*sid*)
Get storage by its id.

Parameters **sid** (*int*) – id of storage caller is looking for.

Return type *ducky.devices.Device*

Returns found device.

Raises ***ducky.errors.InvalidResourceError*** – when no such storage exists.

halt()

hw_setup(*machine_config*)

load_state(*state*)

on_core_alive(*core*)
Signal machine that one of CPU cores is now alive.

on_core_halted(*core*)
Signal machine that one of CPU cores is no longer alive.

run()

save_state(*parent*)

setup_devices()

suspend()

tenh(*s*, **args*)

trigger_irq(*handler*)

wake_up()

class ducky.machine.**MachineState**
Bases: *ducky.snapshot.SnapshotNode*

get_cpu_state_by_id(*cpuid*)

get_cpu_states()

ducky.machine.**cmd_boot**(*console*, *cmd*)
Setup HW, load binaries, init everything

ducky.machine.**cmd_halt**(*console*, *cmd*)
Halt execution

ducky.machine.**cmd_run**(*console*, *cmd*)
Start execution of loaded binaries

ducky.machine.**cmd_snapshot**(*console*, *cmd*)
Create snapshot

2.12 ducky.mm package

2.12.1 Submodules

ducky.mm.binary module

class ducky.mm.binary.**File**(*logger*, *stream*)
Bases: *ducky.util.BinaryFile*

MAGIC = 57005

VERSION = 3

_create_header()

_read(**args*, ***kwargs*)

_read_header()

_write(**args*, ***kwargs*)

_write_header(*header*)

create_section(*name=None*)

fix_offsets()

get_section_by_index(*index*)

get_section_by_name(*name*, *dont_create=False*)

get_section_by_type(*typ*)

get_strings_section()

header

static open(**args*, ***kwargs*)

save()

sections

setup()

string_table

class ducky.mm.binary.**FileFlags**
Bases: *ducky.util.Flags*

_encoding
alias of *FileFlagsEncoding*

_flags = ['reserved']

_labels = 'M'

field = ('reserved', <class 'ctypes.c_ushort'>, 1)

class ducky.mm.binary.**FileFlagsEncoding**
Bases: _ctypes.Structure

fields = [('reserved', <class 'ctypes.c_ushort'>, 1)]

reserved
Structure/Union member

class ducky.mm.binary.**FileHeader**
Bases: _ctypes.Structure

fields = [('magic', <class 'ctypes.c_ushort'>), ('version', <class 'ctypes.c_ushort'>), ('flags', <class 'ducky.mm.binary

pack = 0

flags
Structure/Union member

magic
Structure/Union member

sections
Structure/Union member

version
Structure/Union member

class ducky.mm.binary.**RelocEntry**
Bases: _ctypes.Structure

fields = [('flags', <class 'ducky.mm.binary.RelocFlagsEncoding'>), ('name', <class 'ctypes.c_uint'>), ('patch_add', <

pack = 0

flags
Structure/Union member

name
Structure/Union member

patch_add
Structure/Union member

patch_address
Structure/Union member

patch_offset
Structure/Union member

patch_section
Structure/Union member

patch_size
Structure/Union member

class ducky.mm.binary.**RelocFlags**
Bases: *ducky.util.Flags*

_encoding
alias of *RelocFlagsEncoding*

_flags = ['relative', 'inst_aligned']

_labels = 'RI'

field = ('inst_aligned', <class 'ctypes.c_ushort'>, 1)

class ducky.mm.binary.**RelocFlagsEncoding**
Bases: _ctypes.Structure

fields = [('relative', <class 'ctypes.c_ushort'>, 1), ('inst_aligned', <class 'ctypes.c_ushort'>, 1)]

inst_aligned
Structure/Union member

relative
Structure/Union member

class ducky.mm.binary.**Section**(*parent*, *index*, *name=None*, *header=None*, *payload=None*)
Bases: object

_create_header()

_create_payload()

_get_payload()

_read_header()

_read_payload()

_set_payload(*payload*)

_write_header()

_write_payload()

header

name

payload

prepare_write()

write()

class ducky.mm.binary.**SectionFlags**
Bases: *ducky.util.Flags*

_encoding
alias of *SectionFlagsEncoding*

_flags = ['readable', 'writable', 'executable', 'loadable', 'bss', 'globally_visible']

_labels = 'RWXLBG'

field = ('globally_visible', <class 'ctypes.c_ubyte'>, 1)

class ducky.mm.binary.**SectionFlagsEncoding**
Bases: _ctypes.Structure

fields = [('readable', <class 'ctypes.c_ubyte'>, 1), ('writable', <class 'ctypes.c_ubyte'>, 1), ('executable', <class 'ctyp

bss
Structure/Union member

executable
Structure/Union member

globally_visible
Structure/Union member

loadable
Structure/Union member

readable
Structure/Union member

writable
Structure/Union member

class ducky.mm.binary.**SectionHeader**
Bases: _ctypes.Structure

fields = [('index', <class 'ctypes.c_ubyte'>), ('name', <class 'ctypes.c_uint'>), ('type', <class 'ctypes.c_ubyte'>), ('fla

pack = 0

base
Structure/Union member

data_size
Structure/Union member

file_size
Structure/Union member

flags
Structure/Union member

index
Structure/Union member

name
Structure/Union member

offset
Structure/Union member

padding
Structure/Union member

type
Structure/Union member

class ducky.mm.binary.**SectionTypes**
Bases: enum.IntEnum

PROGBITS = 1

RELOC = 4

STRINGS = 3

SYMBOLS = 2

UNKNOWN = 0

_member_map_ = OrderedDict([('UNKNOWN', <SectionTypes.UNKNOWN: 0>), ('PROGBITS', <SectionTypes.PROGI

_member_names_ = ['UNKNOWN', 'PROGBITS', 'SYMBOLS', 'STRINGS', 'RELOC']

_member_type_
alias of int

_value2member_map_ = {0: <SectionTypes.UNKNOWN: 0>, 1: <SectionTypes.PROGBITS: 1>, 2: <SectionTypes.SYM

class ducky.mm.binary.**SymbolDataTypes**
Bases: enum.IntEnum

ASCII = 5

BYTE = 3

CHAR = 2

FUNCTION = 6

INT = 0

SHORT = 1

STRING = 4

UNKNOWN = 7

_member_map_ = OrderedDict([('INT', <SymbolDataTypes.INT: 0>), ('SHORT', <SymbolDataTypes.SHORT: 1>), ('CH

_member_names_ = ['INT', 'SHORT', 'CHAR', 'BYTE', 'STRING', 'ASCII', 'FUNCTION', 'UNKNOWN']

_member_type_
alias of `int`

_value2member_map_ = {0: <SymbolDataTypes.INT: 0>, 1: <SymbolDataTypes.SHORT: 1>, 2: <SymbolDataTypes.CI

class ducky.mm.binary.**SymbolEntry**
Bases: `_ctypes.Structure`

fields = [('flags', <class 'ducky.mm.binary.SymbolFlagsEncoding'>), ('name', <class 'ctypes.c_uint'>), ('address', <c

pack = 0

address
Structure/Union member

filename
Structure/Union member

flags
Structure/Union member

lineno
Structure/Union member

name
Structure/Union member

section
Structure/Union member

size
Structure/Union member

type
Structure/Union member

class ducky.mm.binary.**SymbolFlags**
Bases: *ducky.util.Flags*

_encoding
alias of *SymbolFlagsEncoding*

_flags = ['globally_visible']

_labels = 'G'

field = ('globally_visible', <class 'ctypes.c_ushort'>, 1)

class ducky.mm.binary.**SymbolFlagsEncoding**
Bases: `_ctypes.Structure`

fields = [('globally_visible', <class 'ctypes.c_ushort'>, 1)]

globally_visible
Structure/Union member

2.12.2 Module contents

class ducky.mm.**AnonymousMemoryPage**(*controller*, *index*)
Bases: *ducky.mm.MemoryPage*

"Anonymous" memory page - this page is just a plain array of bytes, and is not backed by any storage. Its content lives only in the memory.

Page is created with all bytes set to zero.

clear()

read_u16(*offset*)

read_u32(*offset*)

read_u8(*offset*)

write_u16(*offset*, *value*)

write_u32(*offset*, *value*)

write_u8(*offset*, *value*)

class ducky.mm.**ExternalMemoryPage**(*controller*, *index*, *data*, *offset=0*)
Bases: *ducky.mm.MemoryPage*

Memory page backed by an external source. Source is an array of bytes, and can be provided by device driver, mmaped file, or by any other mean.

clear()

get(*offset*)
Get one byte from page. Override this method in case you need a different offset of requested byte.

Parameters **offset** (*int*) – offset of the requested byte.

Return type int

Returns byte at position in page.

put(*offset*, *b*)
Put one byte into page. Override this method in case you need a different offset of requested byte.

Parameters

- **offset** (*int*) – offset of modified byte.
- **b** (*int*) – new value.

read_u16(*offset*)

read_u32(*offset*)

read_u8(*offset*)

save_state(*parent*)

write_u16(*offset*, *value*)

write_u32(*offset*, *value*)

write_u8(*offset*, *value*)

class ducky.mm.**MMOperationList**
Bases: enum.IntEnum

ALLOC = 3

FREE = 4

MMAP = 6

UNMMAP = 7

UNUSED = 5

_member_map_ = OrderedDict([('ALLOC', <MMOperationList.ALLOC: 3>), ('FREE', <MMOperationList.FREE: 4>)

_member_names_ = ['ALLOC', 'FREE', 'UNUSED', 'MMAP', 'UNMMAP']

_member_type_
alias of `int`

_value2member_map_ = {3: <MMOperationList.ALLOC: 3>, 4: <MMOperationList.FREE: 4>, 5: <MMOperationLis

exception ducky.mm.**MalformedBinaryError**
Bases: exceptions.Exception

class ducky.mm.**MemoryController**(*machine*, *size=16777216*)
Bases: object

Memory controller handles all operations regarding main memory.

Parameters

- **machine** (ducky.machine.Machine) – virtual machine that owns this controller.
- **size** (*int*) – size of memory, in bytes.

Raises ***ducky.errors.InvalidResourceError*** – when memory size is not multiple of *ducky.mm.PAGE_SIZE*.

_MemoryController__alloc_page(*index*)
Allocate new anonymous page for usage. The first available index is used.

Be aware that this method does NOT check if page is already allocated. If it is, it is just overwritten by new anonymous page.

Parameters **index** (*int*) – index of requested page.

Returns newly reserved page.

Return type *ducky.mm.AnonymousMemoryPage*

_MemoryController__remove_page(*pg*)
Removes page object for a specific memory page.

Parameters **pg** (ducky.mm.MemoryPage) – page to be removed

_MemoryController__set_page(*pg*)
Install page object for a specific memory page.

Parameters **pg** (ducky.mm.MemoryPage) – page to be installed

Returns installed page

Return type *ducky.mm.MemoryPage*

alloc_page(*base=None*)
Allocate new anonymous page for usage. The first available index is used.

Parameters **base** (*int*) – if set, start searching pages from this address.

Returns newly reserved page.

Return type *ducky.mm.AnonymousMemoryPage*

Raises ***ducky.errors.InvalidResourceError*** – when there is no available page.

alloc_pages(*base=None*, *count=1*)
Allocate continuous sequence of anonymous pages.

Parameters

- **base** (*u24*) – if set, start searching pages from this address.
- **count** (*int*) – number of requested pages.

Returns list of newly allocated pages.

Return type list of *ducky.mm.AnonymousMemoryPage*

Raises ***ducky.errors.InvalidResourceError*** – when there is no available sequence of pages.

alloc_specific_page(*index*)
Allocate new anonymous page with specific index for usage.

Parameters **index** (*int*) – allocate page with this particular index.

Returns newly reserved page.

Return type *ducky.mm.AnonymousMemoryPage*

Raises ***ducky.errors.AccessViolationError*** – when page is already allocated.

boot()
Prepare memory controller for immediate usage by other components.

free_page(*page*)
Free memory page when it's no longer needed.

Parameters **page** (ducky.mm.MemoryPage) – page to be freed.

free_pages(*page*, *count=1*)
Free a continuous sequence of pages when they are no longer needed.

Parameters

- **page** (ducky.mm.MemoryPage) – first page in series.
- **count** (*int*) – number of pages.

get_page(*index*)
Return memory page, specified by its index from the beginning of memory.

Parameters **index** (*int*) – index of requested page.

Return type *ducky.mm.MemoryPage*

Raises ***ducky.errors.AccessViolationError*** – when requested page is not allocated.

get_pages(*pages_start=0*, *pages_cnt=None*, *ignore_missing=False*)
Return list of memory pages.

Parameters

- **pages_start** (*int*) – index of the first page, 0 by default.
- **pages_cnt** (*int*) – number of pages to get, number of all memory pages by default.
- **ignore_missing** (*bool*) – if True, ignore missing pages, False by default.

Raises ***ducky.errors.AccessViolationError*** – when `ignore_missing == False` and there's a missing page in requested range, this exception is rised.

Returns list of pages in area

Return type *list* of *ducky.mm.MemoryPage*

halt()

load_state(*state*)

pages_in_area(*address=0*, *size=None*, *ignore_missing=False*)
Return list of memory pages.

Parameters

- **address** (*u24*) – beggining address of the area, by default 0.
- **size** (*u24*) – size of the area, by default the whole memory size.
- **ignore_missing** (*bool*) – if `True`, ignore missing pages, `False` by default.

Raises ***ducky.errors.AccessViolationError*** – when `ignore_missing == False` and there's a missing page in requested range, this exception is rised.

Returns list of pages in area

Return type *list* of *ducky.mm.MemoryPage*

read_u16(*addr*)

read_u32(*addr*)

read_u8(*addr*)

register_page(*pg*)
Install page object for a specific memory page. This method is intended for external objects, e.g. device drivers to install their memory page objects to handle memory-mapped IO.

Parameters **pg** (`ducky.mm.MemoryPage`) – page to be installed

Returns installed page

Return type *ducky.mm.AnonymousMemoryPage*

Raises ***ducky.errors.AccessViolationError*** – when there is already allocated page

save_state(*parent*)

unregister_page(*pg*)
Remove page object for a specific memory page. This method is intende for external objects, e.g. device drivers to remove their memory page objects handling memory-mapped IO.

Parameters **pg** (`ducky.mm.MemoryPage`) – page to be removed

Raises ***ducky.errors.AccessViolationError*** – when there is no allocated page

write_u16(*addr*, *value*)

write_u32(*addr*, *value*)

write_u8(*addr*, *value*)

class `ducky.mm.`**MemoryPage**(*controller*, *index*)
Bases: `object`

Base class for all memory pages of any kinds.

Memory page has a set of boolean flags that determine access to and behavior of the page.

Flag	Meaning	Default
`read`	page is readable by executed instructions	`False`
`write`	page is writable by executed instructions	`False`
`execute`	content of the page can be used as executable instructions	`False`
`dirty`	there have been write access to this page, its content has changed	`False`

Parameters

- **controller** (ducky.mm.MemoryController) – Controller that owns this page.
- **index** (*int*) – Serial number of this page.

clear()
Clear page.

This operation is implemented by child classes.

load_state(*state*)
Restore page from a snapshot.

read_u16(*offset*)
Read word.

This operation is implemented by child classes.

Parameters **offset** (*int*) – offset of requested word.

Return type int

read_u32(*offset*)
Read longword.

This operation is implemented by child classes.

Parameters **offset** (*int*) – offset of requested longword.

Return type int

read_u8(*offset*)
Read byte.

This operation is implemented by child classes.

Parameters **offset** (*int*) – offset of requested byte.

Return type int

save_state(*parent*)
Create state of this page, and attach it to snapshot tree.

Parameters **parent** (ducky.snapshot.SnapshotNode) – Parent snapshot node.

write_u16(*offset*, *value*)
Write word.

This operation is implemented by child classes.

Parameters

- **offset** (*int*) – offset of requested word.
- **value** (*int*) – value to write into memory.

write_u32(*offset*, *value*)
Write longword.

This operation is implemented by child classes.

Parameters

- **offset** (*int*) – offset of requested longword.
- **value** (*int*) – value to write into memory.

write_u8(*offset*, *value*)
Write byte.

This operation is implemented by child classes.

Parameters

- **offset** (*int*) – offset of requested byte.
- **value** (*int*) – value to write into memory.

class ducky.mm.**MemoryPageState**(**args*, ***kwargs*)
Bases: *ducky.snapshot.SnapshotNode*

class ducky.mm.**MemoryRegion**(*mc*, *name*, *address*, *size*, *flags*)
Bases: *ducky.interfaces.ISnapshotable*, object

load_state(*state*)

region_id = 0

save_state(*parent*)

class ducky.mm.**MemoryRegionState**
Bases: *ducky.snapshot.SnapshotNode*

class ducky.mm.**MemoryState**
Bases: *ducky.snapshot.SnapshotNode*

get_page_states()

ducky.mm.**OFFSET_FMT**(*offset*)

ducky.mm.**PAGE_SIZE = 256**
Size of memory page, in bytes.

class ducky.mm.**PageTableEntry**
Bases: *ducky.util.Flags*

DIRTY = 8

EXECUTE = 4

READ = 1

WRITE = 2

_flags = ['read', 'write', 'execute', 'dirty']

_labels = 'RWXD'

ducky.mm.**SIZE_FMT**(*size*)

class ducky.mm.**VirtualMemoryPage**(*controller*, *index*)
Bases: *ducky.mm.MemoryPage*

Memory page without any real storage backend.

save_state(*parent*)

ducky.mm.**addr_to_offset**(*addr*)

ducky.mm.**addr_to_page**(*addr*)

ducky.mm.**area_to_pages**(*addr*, *size*)

2.13 ducky.patch module

class ducky.patch.**Importer**
Bases: object

find_module(*fullname*, *path=None*)

loader_for_path(*directory*, *fullname*)

class ducky.patch.**ModuleLoader**(*fullpath*)
Bases: object

get_code(*fullname*)

get_source(*path*)

load_module(*fullname*)

class ducky.patch.**RemoveLoggingVisitor**
Bases: ast.NodeTransformer

visit_Expr(*node*)

visit_For(*node*)

visit_If(*node*)

ducky.patch.**debug**(*s*)

ducky.patch.**exec_f**(*object_*, *globals_=None*, *locals_=None*)

2.14 ducky.profiler module

This module provides support for profiling the virtual machine (*machine* profilers) and running programs (*code* profilers). Wrappers for python's deterministic profilers, and simple statistical profiler of running code are available for usage thoughout the ducky sources.

cProfile is the prefered choice of machine profiling backend, with profile is used as a backup option.

Beware, each thread needs its own profiler instance. These instances can be acquired from ProfilerStore class which handles saving their data when virtual machine exits.

To simplify the usage of profilers in ducky sources in case when user does not need profiling, I chose to provide classes with the same API but these classes does not capture any data. These are called *dummy* profilers, as opposed to the *real* ones. Both kinds mimic API of profile.Profile - the *real* machine profiler **is** profile.Profile object.

class ducky.profiler.**DummyCPUCoreProfiler**(*core*, *frequency=17*)
Bases: object

Dummy code profiler class. Base class for all code profilers.

Parameters

- **core** (ducky.cpu.CPUCore) – core this profiler captures data from.
- **frequency** (*int*) – sampling frequency, given as an instruction count.

create_stats()
Preprocess collected data before they can be printed, searched or saved.

disable()
Disable collection of profiling data.

dump_stats(*filename*)
Save collected data into file.

Parameters **filename** (*string*) – path to file.

enable()
Enable collection of profiling data.

take_sample()
Take a sample of current state of CPU core, and store any necessary data.

class ducky.profiler.**DummyMachineProfiler**(**args*, ***kwargs*)
Bases: object

Dummy machine profiler. Does absolutely nothing.

create_stats()
Preprocess collected data before they can be printed, searched or saved.

disable()
Stop collecting profiling data.

dump_stats(*filename*)
Save collected data into file.

Parameters **filename** (*string*) – path to file.

enable()
Start collecting profiling data.

class ducky.profiler.**ProfileRecord**
Bases: object

merge(*other*)

class ducky.profiler.**ProfilerStore**
Bases: object

This class manages profiler instances. When in need of a profiler (e.g. in a new thread) get one by calling proper method of ProfilerStore object.

enable_cpu()
Each newly created code profiler will be the real one.

enable_machine()
Each newly created virtual machine profiler will be the real one.

get_core_profiler(*core*)
Create new code profiler.

Return type *DummyCPUCoreProfiler*

get_machine_profiler()
Create and return new machine profiler.

Returns new machine profiler.

is_cpu_enabled()
Returns True when code profiling is enabled.

Return type bool

is_machine_enabled()
Returns `True` when virtual machine profiling is enabled.

Return type bool

save(*logger*, *directory*)
Save all captured data to files. Each created profiler stores its data in separate file.

Parameters **directory** (*string*) – directory where all files are stored.

class ducky.profiler.**RealCPUCoreProfiler**(*core*)
Bases: *ducky.profiler.DummyCPUCoreProfiler*

Real code profiler. This class actually does collect data.

dump_stats(*filename*)

take_sample()

ducky.profiler.**STORE = <ducky.profiler.ProfilerStore object>**
Main profiler store

2.15 ducky.reactor module

This module provides simple reactor core that runs each of registered tasks at least once during one iteration of its internal loop.

There are two different kinds of objects that reactor manages:

- task - it's called periodicaly, at least once in each reactor loop iteration
- event - asynchronous events are queued and executed before running any tasks. If there are no runnable tasks, reactor loop waits for incomming events.

class ducky.reactor.**CallInReactorTask**(*fn*, **args*, ***kwargs*)
Bases: *ducky.interfaces.IReactorTask*

This task request running particular function during the reactor loop. Useful for planning future work, and for running tasks in reactor's thread.

Parameters

- **fn** – callback to fire.
- **args** – arguments for callback.
- **kwargs** – keyword arguments for callback.

run()

class ducky.reactor.**FDCallbacks**(*on_read*, *on_write*, *on_error*)
Bases: `tuple`

_asdict()
Return a new OrderedDict which maps field names to their values

_fields = ('on_read', 'on_write', 'on_error')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new FDCallbacks object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new FDCallbacks object replacing specified fields with new values

on_error
Alias for field number 2

on_read
Alias for field number 0

on_write
Alias for field number 1

class ducky.reactor.**Reactor**(*machine*)
Bases: object

Main reactor class.

add_call(*fn*, **args*, ***kwargs*)
Enqueue function call. Function will be called in reactor loop.

add_event(*event*)
Enqueue asynchronous event.

add_fd(*fd*, *on_read=None*, *on_write=None*, *on_error=None*)
Register file descriptor with reactor. File descriptor will be checked for read/write/error posibilities, and appropriate callbacks will be fired.

No arguments are passed to callbacks.

Parameters

- **fd** (*int*) – file descriptor.
- **on_read** – function that will be called when file descriptor is available for reading.
- **on_write** – function that will be called when file descriptor is available for write.
- **on_error** – function that will be called when error state raised on file descriptor.

add_task(*task*)
Register task with reactor's main loop.

remove_fd(*fd*)
Unregister file descriptor. It will no longer be checked by its main loop.

Parameters **fd** (*int*) – previously registered file descriptor.

remove_task(*task*)
Unregister task, it will never be ran again.

run()
Starts reactor loop. Enters endless loop, calling runnable tasks and events, and - in case there are no runnable tasks - waits for new events.

When there are no tasks managed by reactor, loop quits.

task_runnable(*task*)
If not yet marked as such, task is marked as runnable, and its `run()` method will be called every iteration of reactor's main loop.

task_suspended(*task*)
If runnable, task is marked as suspended, not runnable, and it will no longer be ran by reactor. It's still registered, so reactor's main loop will not quit, and task can be later easily re-enabled by calling *ducky.reactor.Reactor.task_runnable()*.

class ducky.reactor.**RunInIntervalTask**(*ticks*, *fn*, **args*, ***kwargs*)
Bases: *ducky.interfaces.IReactorTask*

This task will run its callback every `ticks` iterations of reactor's main loop.

Parameters

- **ticks** (*int*) – number of main loop iterations between two callback calls.
- **args** – arguments for callback.
- **kwargs** – keyword arguments for callback.

run()

class ducky.reactor.**SelectTask**(*machine*, *fds*, **args*, ***kwargs*)
Bases: *ducky.interfaces.IReactorTask*

Private task, serving as a single point where `select` syscall is being executed. When a subsystem is interested in IO on a file descriptor, such file descriptor should be set as non-blocking, and then registered with reactor - it's not viable to place `select` calls everywhere in different drivers. This task takes list of registered file descriptors, checks for possible IO oportunities, and fires callbacks accordingly.

Parameters

- **machine** (ducky.machine.Machine) – VM this task (and reactor) belongs to.
- **fds** (*dict*) – dictionary, where keys are descriptors, and values are lists of their callbacks.

add_fd(*fd*, *on_read=None*, *on_write=None*, *on_error=None*)
Register file descriptor with reactor. File descriptor will be checked for read/write/error posibilities, and appropriate callbacks will be fired.

No arguments are passed to callbacks.

Parameters

- **fd** (*int*) – file descriptor.
- **on_read** – function that will be called when file descriptor is available for reading.
- **on_write** – function that will be called when file descriptor is available for write.
- **on_error** – function that will be called when error state raised on file descriptor.

remove_fd(*fd*)
Unregister file descriptor. It will no longer be checked by its main loop.

Parameters **fd** (*int*) – previously registered file descriptor.

run()

2.16 ducky.snapshot module

class ducky.snapshot.**CoreDumpFile**(*logger*, *stream*)
Bases: *ducky.util.BinaryFile*

load()

static open(**args*, ***kwargs*)

save(*state*)

class ducky.snapshot.**SnapshotNode**(**fields*)
Bases: object

add_child(*name*, *child*)

get_child(*name*)

get_children()

print_node(*level=0*)

class ducky.snapshot.**VMState**(*logger*)
Bases: *ducky.snapshot.SnapshotNode*

static capture_vm_state(*machine*, *suspend=True*)

static load_vm_state(*logger*, *filename*)

save(*filename*)

2.17 ducky.streams module

Streams represent basic IO objects, used by devices for reading or writing (streams) of data.

`Stream` object encapsulates an actual IO object - `file`-like stream, raw file descriptor, or even something completely different. `Stream` classes then provide basic IO methods for moving data to and from stream, shielding user from implementation details, like Python2/Python3 differencies.

class ducky.streams.**FDInputStream**(*machine*, *fd*, ***kwargs*)
Bases: *ducky.streams.InputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**FDOutputStream**(*machine*, *fd*, ***kwargs*)
Bases: *ducky.streams.OutputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**FileInputStream**(*machine*, *f*, ***kwargs*)
Bases: *ducky.streams.InputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**FileOutputStream**(*machine*, *f*, ***kwargs*)
Bases: *ducky.streams.OutputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**InputStream**(*machine*, *desc*, *stream=None*, *fd=None*, *close=True*, *allow_close=True*)
Bases: *ducky.streams.Stream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

static create(*machine*, *desc*)

read(*size=None*)

write(*b*)

class ducky.streams.**MethodInputStream**(*machine*, *desc*, ***kwargs*)
Bases: *ducky.streams.InputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**MethodOutputStream**(*machine*, *desc*, ***kwargs*)
Bases: *ducky.streams.OutputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**OutputStream**(*machine*, *desc*, *stream=None*, *fd=None*, *close=True*, *allow_close=True*)
Bases: *ducky.streams.Stream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

static create(*machine*, *desc*)

flush()

read(*size=None*)

write(*buff*)

class ducky.streams.**StderrStream**(*machine*)
Bases: *ducky.streams.OutputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**StdinStream**(*machine*, ***kwargs*)
Bases: *ducky.streams.InputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

close()

get_selectee()

class ducky.streams.**StdoutStream**(*machine*)
Bases: *ducky.streams.OutputStream*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

class ducky.streams.**Stream**(*machine*, *desc*, *stream=None*, *fd=None*, *close=True*, *allow_close=True*)
Bases: object

Abstract base class of all streams.

Parameters

- **machine** – parent :py:class‘ducky.machine.Machine‘ object.
- **desc** – description of stream. This is a short, string representation of the stream.
- **stream** – file-like stream that provides IO method (read() or write). If it is set, it is preferred IO object.
- **fd** (*int*) – raw file descriptor. stream takes precedence, otherwise this file descriptor is used.
- **close** (*bool*) – if True, and if stream has a close() method, stream will provide close() method that will close the underlaying file-like object. True by default.
- **allow_close** (*bool*) – if not True, stream's close() method will *not* close underlying IO resource. True by default.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

_raw_read_fd(*size=None*)

_raw_read_stream(*size=None*)

_raw_write_fd(*data*)

_raw_write_stream(*data*)

close()
This method will close the stream. If `allow_close` flag is not set to `True`, nothing will happen. If the stream wasn't created with `close` set to `True`, nothing will happen. If the wrapped IO resource does not have `close()` method, nothing will happen.

has_fd()
Check if stream has raw file descriptor. File descriptors can be checked for IO availability by reactor's polling task.

Return type bool

Returns `True` when stream has file descriptor.

has_poll_support()
Streams that can polled for data should return `True`.

Return type bool

read(*size=None*)
Read data from stream.

Parameters **size** (*int*) – if set, read at maximum `size` bytes.

Return type `bytearray` (Python2), `bytes` (Python3)

Returns read data, of maximum lenght of `size`, `None` when there are no available data, or empty string in case of EOF.

register_with_reactor(*reactor*, ***kwargs*)
Called by owner to register the stream with reactor's polling service.

See *ducky.reactor.Reactor.add_fd()* for keyword arguments.

Parameters **reactor** (`ducky.reactor.Reactor`) – reactor instance to register with.

unregister_with_reactor(*reactor*)
Called by owner to unregister the stream with reactor's polling service, e.g. when stream is about to be closed.

Parameters **reactor** (`ducky.reactor.Reactor`) – reactor instance to unregister from.

write(*buff*)
Write data to stream.

Parameters **buff** (*bytearray*) – data to write. `bytearray` (Python2), `bytes` (Python3)

ducky.streams.**fd_blocking**(*fd*, *block=None*)
Query or set blocking mode of file descriptor.

Return type bool

Returns if `block` is `None`, current setting of blocking mode is returned - `True` for blocking, `False` for non-blocking. Othwerwise, function returns nothing.

2.18 ducky.tools package

2.18.1 Submodules

ducky.tools.as module

ducky.tools.as.**encode_blob**(*logger*, *file_in*, *options*)

ducky.tools.as.**get_assembler_process**(*logger*, *buffer*, *file_in*, *options*)

ducky.tools.as.**main**()

ducky.tools.as.**save_object_file**(*logger*, *sections*, *file_out*, *options*)

ducky.tools.coredump module

ducky.tools.coredump.**__load_forth_symbols**(*logger*)

ducky.tools.coredump.**__read**(*state*, *cnt*, *address*)

ducky.tools.coredump.**__show_forth_word**(*state*, *symbols*, *base_address*, *ending_addresses*)

ducky.tools.coredump.**main**()

ducky.tools.coredump.**show_cores**(*logger*, *state*)

ducky.tools.coredump.**show_dump**(*state*, *dumps*)

ducky.tools.coredump.**show_forth_dict**(*logger*, *state*, *last*)

ducky.tools.coredump.**show_forth_word**(*logger*, *state*, *base_address*)

ducky.tools.coredump.**show_header**(*logger*, *state*)

ducky.tools.coredump.**show_memory**(*logger*, *state*)

ducky.tools.coredump.**show_pages**(*logger*, *state*, *empty_pages=False*)

ducky.tools.defs module

ducky.tools.defs.**main**()

ducky.tools.defs.**parse_template**(*file_in*, *file_out*)

ducky.tools.img module

ducky.tools.img.**align_file**(*f_out*, *boundary*)

ducky.tools.img.**create_binary_image**(*f_in*, *f_out*, *bio=False*)

ducky.tools.img.**create_hdt_image**(*file_in*, *f_out*, *options*)

ducky.tools.img.**main**()

ducky.tools.ld module

class ducky.tools.ld.**LinkerInfo**(*linker_script*)
Bases: object

class ducky.tools.ld.**LinkerScript**(*filepath=None*)
Bases: object

section_ordering()

where_to_base(*section*)

where_to_merge(*src_section*)

class ducky.tools.ld.**RelocationPatcher**(*re*, *se*, *symbol_name*, *section*, *original_section=None*, *section_offset=0*)

Bases: object

_apply_patch(*value*)

_create_patch()

patch()

ducky.tools.ld.**archive_files**(*logger*, *files_in*, *file_out*)

ducky.tools.ld.**fix_section_bases**(*info*, *f_out*)

ducky.tools.ld.**link_files**(*info*, *files_in*, *file_out*)

ducky.tools.ld.**main**()

ducky.tools.ld.**merge_object_into**(*info*, *f_dst*, *f_src*)

ducky.tools.ld.**resolve_relocations**(*info*, *f_out*, *f_ins*)

ducky.tools.ld.**resolve_symbols**(*info*, *f_out*, *f_ins*)

ducky.tools.objdump module

ducky.tools.objdump.**main**()

ducky.tools.objdump.**show_disassemble**(*f*)

ducky.tools.objdump.**show_file_header**(*f*)

ducky.tools.objdump.**show_reloc**(*f*)

ducky.tools.objdump.**show_sections**(*options*, *f*)

ducky.tools.objdump.**show_symbols**(*options*, *f*)

ducky.tools.profile module

ducky.tools.profile.**main**()

ducky.tools.profile.**read_profiling_data**(*logger*, *files_in*)

ducky.tools.vm module

class ducky.tools.vm.**DuckyProtocol**(*logger*, *options*, *config*)

Bases: autobahn.twisted.websocket.WebSocketServerProtocol

Protocol handling communication between VM and remote terminal emulator.

Parameters

- **logger** (*logging.Logger*) – Logger instanceto use for logging.
- **options** – command-line options, as returned by option parser.
- **config** (ducky.config.MachineConfig) – VM configuration.

_machines = []

onClose(*wasClean*, *code*, *reason*)

Called when connection ends. Tell VM to halt, and wait for its thread to finish. Then, print some VM stats.

onMessage(*payload*, *isBinary*)
Called when new data arrived from client. Feeds the data to VM's input stream.

See `autobahn.twisted.websocket.WebSocketServerProtocol.onMessage()`.

onOpen(**args*, ***kwargs*)
Called when new client connects to the server.

This callback will setup and start new VM, connecting it to remote terminal by wrapping this protocol instance in two streams (input/output), and spawn a fresh new thread for it.

run_machine()
Wraps VM's `run()` method by `try/except` clause, and call protocols `sendClose()` method when VM finishes.

This is the target of VM's thread.

class `ducky.tools.vm.`**DuckySocketServerFactory**(*logger*, *options*, *config*, **args*, ***kwargs*)
Bases: `autobahn.twisted.websocket.WebSocketServerFactory`

buildProtocol(**args*, ***kwargs*)

class `ducky.tools.vm.`**WSInputStream**(*protocol*, **args*, ***kwargs*)
Bases: *ducky.streams.InputStream*

Websocket input stream, receiving bytes from opened websocket, and pushing them to keyboard frontend device.

Stream has an internal LIFO buffer that is being fed by protocol, and consumed by VM frontend device.

Parameters **protocol** (`DuckyProtocol`) – protocol instance with opened websocket.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 39

_abc_registry = <_weakrefset.WeakSet object>

enqueue(*buff*)
Called by protocol instance, to add newly received data to stream's buffer.

Parameters **buff** (*bytearray*) – receirved bytes.

has_poll_support()
See *ducky.streams.Stream.has_poll_support()*.

read(*size=None*)
See *ducky.streams.Stream.read()*.

register_with_reactor(*reactor*, *on_read=None*, *on_error=None*)
See *ducky.streams.Stream.register_with_reactor()*.

unregister_with_reactor(*reactor*)
See *ducky.streams.Stream.unregister_with_reactor()*.

class `ducky.tools.vm.`**WSOutputStream**(*protocol*, **args*, ***kwargs*)
Bases: *ducky.streams.OutputStream*

Websocket output stream, receiving bytes from TTY frontend, and pushing them to protocol's socket.

Parameters **protocol** (`DuckyProtocol`) – protocol instance with opened websocket.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 39

_abc_registry = <_weakrefset.WeakSet object>

write(*buff*)
Write buffer into the socket.

Called by device from machine thread, therefore this method hands buffer over to the reactor thread.

Parameters **buff** (*bytearray*) – bytes to send to client.

ducky.tools.vm.**main**()

ducky.tools.vm.**print_machine_stats**(*logger*, *M*)

ducky.tools.vm.**process_config_options**(*logger*, *options*, *config_file=None*, *set_options=None*, *add_options=None*, *enable_devices=None*, *disable_devices=None*)
Load VM config file, and apply all necessary changes, as requested by command-line options.

Parameters

- **logger** (*logging.Logger*) – Logger instance to use for logging.
- **options** – command-line options, as returned by option parser.
- **config_file** (*string*) – path to configuration file.

Return type *ducky.config.MachineConfig*

Returns VM configuration.

2.18.2 Module contents

ducky.tools.**add_common_options**(*parser*)

ducky.tools.**parse_options**(*parser*, *default_loglevel=20*, *stream=None*)

ducky.tools.**setup_logger**(*stream=None*, *debug=False*, *quiet=None*, *verbose=None*, *default_loglevel=20*, *colorize=None*)

2.19 ducky.util module

class ducky.util.**BinaryFile**(*logger*, *stream*)
Bases: object

Base class of all classes that represent "binary" files - binaries, core dumps. It provides basic methods for reading and writing structures.

static do_open(*logger*, *path*, *mode='rb'*, *klass=None*)

static open(**args*, ***kwargs*)

read_struct(*st_class*)
Read structure from current position in file.

Returns instance of class st_class with content read from file

Return type st_class

setup()

write_struct(*st*)
Write structure into file at the current position.

Parameters **st** (*class*) – ctype-based structure

class ducky.util.**Flags**
Bases: object

_encoding = []

_flags = []

_labels = ''

classmethod create(***kwargs*)

classmethod encoding()

classmethod from_encoding(*encoding*)

classmethod from_int(*u*)

classmethod from_string(*s*)

load_encoding(*encoding*)

load_int(*u*)

load_string(*s*)

save_encoding(*encoding*)

to_encoding()

to_int()

to_string()

class ducky.util.**Formatter**
Bases: string.Formatter

format_field(*value*, *format_spec*)

format_int(*format_spec*, *value*)

class ducky.util.**LoggingCapable**(*logger*, **args*, ***kwargs*)
Bases: object

class ducky.util.**StringTable**(*buff=None*)
Bases: object

Simple string table, used by many classes operating with files (core, binaries, ...). String can be inserted into table and read, each has its starting offset and its end is marked with null byte ().

Thsi is a helper class - it makes working with string, e.g. section and symbol names, much easier.

buff
Serialize internal string table to a stream of bytes.

get_string(*offset*)
Read string from table.

Parameters **offset** (*int*) – offset of the first character from the beginning of the table

Returns string

Return type string

put_string(*s*)
Insert new string into table. String is appended at the end of internal buffer.

Returns offset of inserted string

Return type int

class ducky.util.**SymbolTable**(*binary*)
Bases: dict

get_symbol(*name*)

ducky.util.**align**(*boundary*, *n*)

ducky.util.**isfile**(*o*)

ducky.util.**sizeof_fmt**(*n*, *suffix='B'*, *max_unit='Zi'*)

ducky.util.**str2int**(*s*)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y