py_register_machine2 Documentation

Release 0.1.0

Daniel Knuettel

December 09, 2016

CONTENTS

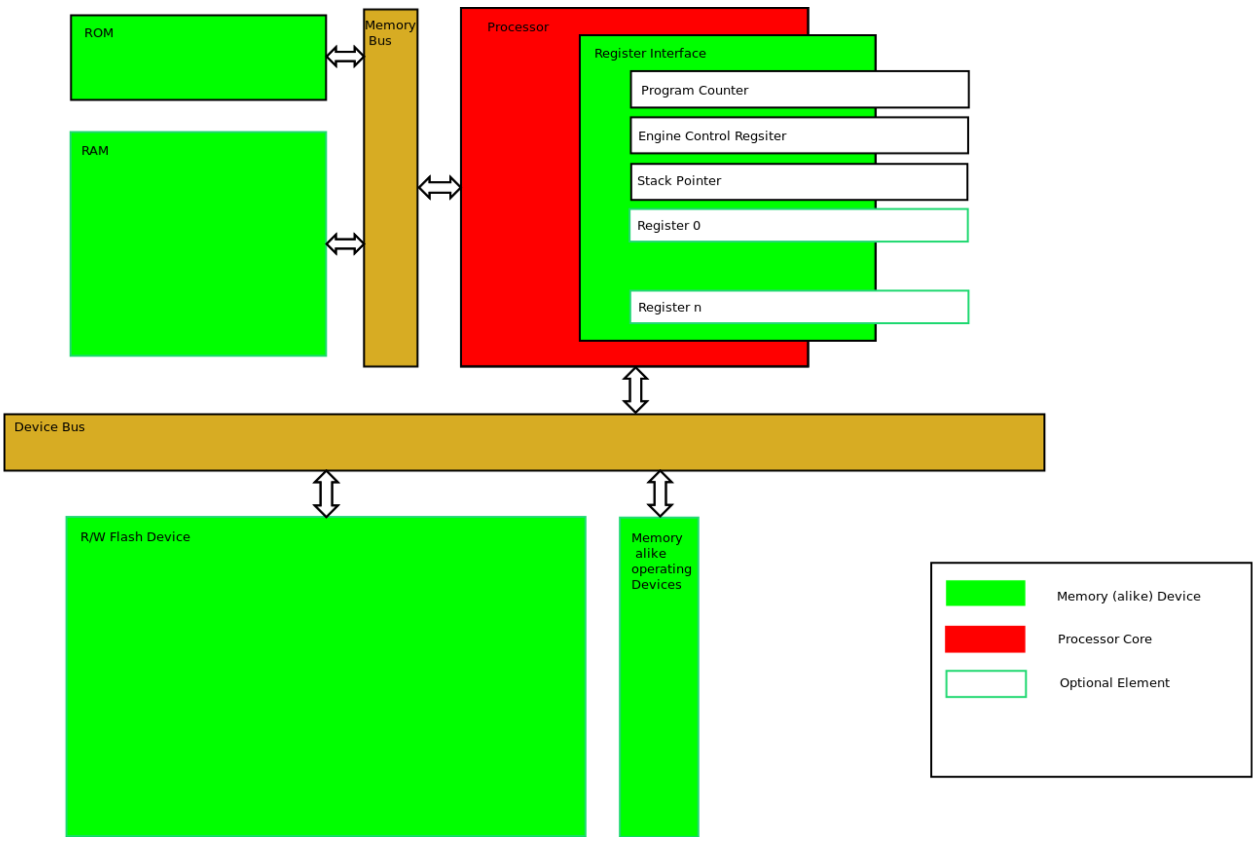
Contents:

CHAPTER

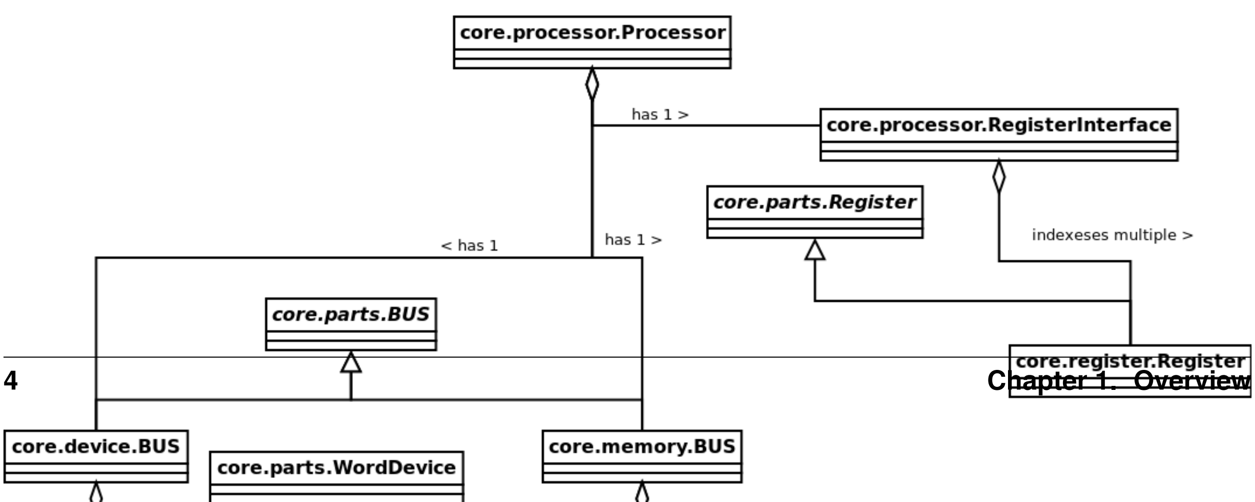
ONE

OVERVIEW

Block Diagram of the Architecture



Class Diagram of the Memory Layout



(UML Diagram)

CHAPTER TWO

QUICKSTART

Jumping into PyRegisterMachine2

The first thing you might want to do is to assemble some code and run it on the engine. To do so you have to set up a `Processor` and the `Assembler`:

```
from py_register_macine2.machines.small import small_register_machine
from py_register_machine2.tools.assembler.assembler import Assembler
processor, rom, ram, flash = small_register_machine()
processor.setup_done()
```

`small_register_machine` returns a configured engine with 50 words ROM, 200 words RAM and 500 words Flash. The size of a word is 64 bit.

Now you need a program, in this case we will use simple `Hello, World` program:

```
from io import StringIO
asm = '''\
ldi 'H' out0
ldi 'e' out0
ldi 'l' out0
ldi 'l' out0
ldi 'o' out0
ldi '!' out0
ldi 0xa out0
ldi 0b1 ECR'''
stream = StringIO(asm)
```

This will print `Hello!\n` to `sys.stdout` and stop the engine. The assembly language is KASM2.

The assembler will convert the assembly into machine code:

```
assembler = Assembler(processor, stream)
code = assembler.assemble()
```

The machine code is just a `list` of `int` objects that can be programmed to the ROM or Flash device:

```
rom.program(code)
```

And then the Processor will execute the program:

```
processor.run()
```

Putting it all together:

```
from py_register_macine2.machines.small import small_register_machine
from py_register_machine2.tools.assembler.assembler import Assembler
processor, rom, ram, flash = small_register_machine()
processor.setup_done()
from io import StringIO
asm = '''\
ldi 'H' out0
ldi 'e' out0
ldi 'l' out0
ldi 'l' out0
ldi 'o' out0
ldi '!' out0
ldi 0xa out0
ldi 0b1 ECR'''
stream = StringIO(asm)
assembler = Assembler(processor, stream)
code = assembler.assemble()
rom.program(code)
processor.run()
```

A simple Code Loader

Usually the ROM will be too small to hold the programs you want to execute, so those will be stored in the Flash.

To execute programs stored in the Flash one has to load them into RAM, this is done by the Code Loader(*CL*) or Boot Code Loader (*BCL*) stored in the ROM.

We will use the same setup like above:

```
from py_register_macine2.machines.small import small_register_machine
from py_register_machine2.tools.assembler.assembler import Assembler
from py_register_machine2.tools.assembler.directives import ConvertingDirective
from io import StringIO

processor, rom, ram, flash = small_register_machine()
processor.setup_done()

asm = '''\
.set flash_sec_size flash_sec_end
ldi 'H' out0
ldi 'e' out0
ldi 'l' out0
ldi 'l' out0
ldi 'o' out0
ldi '!' out0
ldi 0xa out0
ldi 0b1 ECR
flash_sec_end:
'''

stream = StringIO(asm)

# This directive will allow you to set one word to a special value
# used to get the size of the section
set_directive = ConvertingDirective(".set", lambda x: x)
```

```
assembler = Assembler(processor, stream, directives = [set_directive])
code = assembler.assemble()
```

The only new part is the size of the program, stored in the first word, now we will store the program in the Flash:

```
flash.program(code)
```

So now we need the *CL*:

```
cl = '''\
ldi 0 r0
in r0 r1
ldi RAMEND_LOW r2

loop:
inc r0
in r0 r3
pst r3 r2
inc r2
dec r1
jne r1 loop

sjmp RAMEND_LOW
'''
```

What does that code snippet?

At first it reads the size of the code section in the Flash device:

```
ldi 0 r0
in r0 r1
```

and sets up a pointer to the RAM word to write into:

```
ldi RAMEND_LOW r2
```

After that it copies the content from the Flash to the RAM using a do-while-loop. Finally it jumps to the first word in RAM:

```
sjmp RAMEND_LOW
```

The constant `RAMEND_LOW` is a processor constant and points to the first word of the RAM. You are able to generate jump marks using `<name>:` (i.e. `loop:`).

Now you need to assemble and store the Code Loader:

```
stream = StringIO(cl)
assembler = Assembler(processor, stream)
code = assembler.assemble()
rom.program(code)
```

And run it:

```
processor.run()
```

Putting it all together:

```
from py_register_macine2.machines.small import small_register_machine
from py_register_machine2.tools.assembler.assembler import Assembler
from py_register_machine2.tools.assembler.directives import ConvertingDirective
from io import StringIO
```

```
processor, rom, ram, flash = small_register_machine()
processor.setup_done()

asm = '''\
.set flash_sec_size flash_sec_end
ldi 'H' out0
ldi 'e' out0
ldi 'l' out0
ldi 'l' out0
ldi 'o' out0
ldi '!' out0
ldi 0xa out0
ldi 0b1 ECR
flash_sec_end:
'''

stream = StringIO(asm)

# This directive will allow you to set one word to a special value
# used to get the size of the section
set_directive = ConvertingDirective(".set", lambda x: x)

assembler = Assembler(processor, stream, directives = [set_directive])
code = assembler.assemble()

flash.program(code)

cl = '''\
ldi 0 r0
in r0 r1
ldi RAMEND_LOW r2

loop:
inc r0
in r0 r3
pst r3 r2
inc r2
dec r1
jne r1 loop

sjmp RAMEND_LOW
'''

stream = StringIO(cl)
assembler = Assembler(processor, stream)
code = assembler.assemble()
rom.program(code)
processor.run()
```

CHAPTER

THREE

PY_REGISTER_MACHINE2.CORE

Parts

py_register_machine2.core.parts: Basic parts of the register machine

exception `py_register_machine2.core.parts.`**`AddressError`**`(*args)`

raised by a device if the requested offset exceeds the size of the device

class `py_register_machine2.core.parts.`**`BUS`**`(width=64, debug=0)`

The BUS object

A BUS object connects the Processor with one or more Memory alike operating devices (*WordDevice*).

Before the BUS starts working, devices can be registered, the address spaces of the devices are organized incremental:

```
d1 = WordDevice(4)
d2 = WordDevice(5)
d3 = WordDevice(19)

b = BUS()
addr1 = b.register_device(d1)
addr2 = b.register_device(d2)
addr3 = b.register_device(d3)

print( (addr1, addr2, addr3))
# (0, 4, 9)
```

Once the BUS started working (a read/write operation has been used) BUS.register_device will raise a BUSSetupError. If the addresspace of the BUS is too small do hold a new device, BUS.register_device will raise a BUSSetupError.

The number of read/write actions can be observed by accessing the variables `reads` and `writes`

`read_word``(offset)`

Read one word from a device. The offset is `device_addr + device_offset`, e.g.:

```
offset = 3 # third word of the device
offset += addr2
b.read_word(offset)
# reads third word of d2.
```

Truncates the value according to `width`.

May raise *BUSError*, if the offset exceeds the address space.

register_device(*word_device*)
Register the *WordDevice* `word_device` in the bus returns the start address of the device.

raises: *BUSSetupError*, if the device cannot be registered.

write_word(*offset*, *word*)
Writes one word from a device, see *read_word*.

exception `py_register_machine2.core.parts.`**BUSError**(**args*)

raised by a BUS if an operation failed.

exception `py_register_machine2.core.parts.`**BUSSetupError**(**args*)

raised by a BUS if the setup failed.

class `py_register_machine2.core.parts.`**Integer**(*value=0*, *width=64*)
The register machine may have a special width. This is handled by the Integer objects.

Automatically truncates the value to the defined width.

Use *setvalue* and *getvalue* or *setuvalue* and *getuvalue* to access the value.

Uses a bitset internally.

getuvalue()
Get the unsigned value of the Integer, truncate it and handle Overflows.

getvalue()
Get the signed value of the Integer, truncate it and handle Overflows.

setuvalue(*value*)
Set the unsigned value of the Integer.

setvalue(*value*)
Set the signed value of the Integer.

exception `py_register_machine2.core.parts.`**ReadOnlyError**(**args*)

raised by a device if it is read-only

class `py_register_machine2.core.parts.`**Register**(*name*, *width=64*)
Basically hold one value and permitt read/write operations. There may be several subclasses, like Input/Output Register.

The name will be used by the assembler.

read()
Return the content of the Register, may execute a function

write(*value*)
Set the content of the Register, may execute a function

class `py_register_machine2.core.parts.`**WordDevice**(*size*, *width=64*, *mode=3*, *debug=0*)
Base Device for the register machine. The words have the width `width` and are stored in an *Integer* object.

Values are accessed by *read* and *write*

read(*offset*)
Returns the value of the memory word at `offset`.

Might raise *WriteOnlyError*, if the device is write-only. Might raise *AddressError*, if the offset exceeds the size of the device.

write (*offset*, *value*)
Writes the memory word at `offset` to `value`.

Might raise *ReadOnlyError*, if the device is read-only. Might raise *AddressError*, if the offset exceeds the size of the device.

exception `py_register_machine2.core.parts.`**WriteOnlyError** (**args*)

raised by a device if it is write-only

Memory

class `py_register_machine2.core.memory.`**BUS** (*width=64*, *debug=0*)
The processor's memory BUS, its devices are ROM and RAM.

class `py_register_machine2.core.memory.`**RAM** (*size*, *width=64*, *debug=0*)
The Random Access Memory Device

By default the RAM device is filled with zeros. After poweron the Bootcode in the ROM might perform read/write operations on the RAM. If the register machine is to execute programs from the Flash device, this code has to be copied into the RAM.

class `py_register_machine2.core.memory.`**ROM** (*size*, *width=64*, *debug=0*)
The Read Only Memory Device

of the register machine stores either the boot code (for big programs) or the complete program, if the program is really small.

The ROM is attached to the same BUS as the RAM and **always** includes the `offset 0`. The Program Counter (*PC*) of the Processor points to this word on powerup.

Because RAM and ROM are in the same address space the following formula defines the size of RAM and ROM:

```
addr_space = 2 ** memorybus.width
ram.size + rom.size < addr_space
```

A `write` call will raise *ReadOnlyError*.

program (*prog*, *offset=0*)
Write the content of the iterable `prog` starting with the optional offset `offset` to the device.

Invokes *program_word*.

program_word (*offset*, *word*)
Write the word `word` to the memory at offset `offset`. Used to write the boot code.

Might raise *AddressError*, if the offset exceeds the address space.

Devices

py_register_machine2.core.device: Device BUS and attached devices

class `py_register_machine2.core.device.`**BUS** (*width=64*, *debug=0*)
The processor's device BUS, usually the Flash is attached to this BUS, but there might be more devices.

class `py_register_machine2.core.device.`**`Flash`**(*size*, *width=64*, *debug=0*)

The Program Flash

If the size of the program exceeds the size of the ROM the program has to be written into the Flash. The Flash is a Read/Write device and contains

- The Program
- Constants
- Static Variables

The Flash is a WordDevice and attached to the `device.BUS` .

`program`(*prog*, *offset=0*)

Write the content of the iterable `prog` starting with the optional offset `offset` to the device.

Invokes `program_word`.

`program_word`(*offset*, *word*)

Program one word of the Flash device. Might raise *AddressError*.

Register

py_register_machine2.core.register: Registers for the register machine

class `py_register_machine2.core.register.`**`BStreamIORegister`**(*name*, *open_stream_in*, *open_stream_out*, *width=64*)

Works like *StreamIORegister*, but `open_stream_in` and `open_stream_out` are byte streams (like `open("fname", "rb")`).

- A `read` operation will read `width // 8` bytes and convert them to one `int`.
- A `write` operation will write `width // 8` bytes

`read`()

Reads enough bytes from `open_stream_in` to fill the `width` (if available) and converts them to an `int`. Returns this `int`.

`write`(*word*)

Converts the `int word` to a `bytes` object and writes them to `open_stream_out`.

See `int_to_bytes`.

class `py_register_machine2.core.register.`**`OutputRegister`**(*name*, *open_stream*, *width=64*)

Used to print data to the user. The `write` call will convert `word` using `chr` and write the resulting `str` to `open_stream`.

The `read` call will return the last written `word`.

`open_stream` might be a `file` (like `sys.stdout`) or an `io.StringIO` object.

`write`(*word*)

Write the `chr` representation of `word` to the `open_stream`.

If `chr(word)` fails due `OverflowError`, a `"?"` will be written.

class `py_register_machine2.core.register.`**`Register`**(*name*, *width=64*)

The basic standard register. Permitts read and write, does not execute any callbacks on read/write.

See also: *Register*

class `py_register_machine2.core.register.`**`StreamIORegister`**(*name*, *open_stream_in*, *open_stream_out*, *width=64*)

Input/Output Register via streams.

The `open_stream_in` has to be readable, `open_stream_out` writeable.

`read`()

Read a `str` from `open_stream_in` and convert it to an integer using `ord`. The result will be truncated according to *Integer*.

`write`(*word*)

Works like *SOwrite*.

Processor

class `py_register_machine2.core.processor.`**`EnigneControlBits`**

Container for the static engine controll bits. Used by the *Processor* to handle his ECR.

class `py_register_machine2.core.processor.`**`Processor`**(*f_cpu=None*, *width=64*, *interrupts=False*, *clock_barrier=None*, *debug=0*)

Fetches Opcodes from the ROM or RAM, decodes them and executes the commands. Phases in one operation cycle:

Fetch Phase The Processor fetches the Opcode (one word) from the ROM or RAM device according to the program counter and increases the program counter.

Decode Phase The Processor looks up the Command to execute

Fetch Operands Phase (optional) If requested the processor fetches the operands and increases the program counter.

Execute Phase The Processor executes the Command.

Write Back Phase (optional) If there is a result this result is written to a register or the RAM or a device.

Special Register

0.The first Register (index 0) is the Program Counter(PC).

1.The second Register (index 1) is the Engine Control Register (ECR) take a look at *EnigneControlBits*.

2.The third Register (index 2) is the Stack Pointer (SP) and may be used for `call`, `ret`, `push` and `pop`

Internal Constants

Constants used by the Assembler, should be set using *setup_done*

`ROMEND_LOW` First word of the *ROM* (always `0`)

`ROMEND_HIGH` Last word of the *ROM*

`RAMEND_LOW` First word of the *RAM*, (`ROMEND_HIGH + rom.size`)

`RAMEND_HIGH` Last word of the *RAM*

`FLASH_START` First word of the Flash_(always `0`)

`FLASH_END` Last word of the *Flash*

Interrupt Name Address of the interrupt (set by invoking `add_interrupt`)

Cycles

The number of cycles can be observed by acessing the `cycles` variable.

add_interrupt (*interrupt*)
Adds the interrupt to the internal interrupt storage `self.interrupts` and registers the interrupt address in the internal constants.

add_register (*register*)
Adds a new register in the *RegisterInterface*.

Invokes *add_register*.

do_cycle ()
Run one clock cycle of the *Processor*, works according to *processor_phases*.

Then all `on_cycle_callbacks` are executed and the internal Registers are updated.

If `f_cpu` is set and the execution took not long enough, `do_cycle` will wait until the right time for the next cycle.

If `clock_barrier` is set, `do_cycle` will perform the `clock_barrier.wait()`.

Might raise *SIGSEGV*, if there is an invalid opcode.

en_dis_able_interrupts (*mask*)
This callback might be used by a Register to enable/disable Interrupts.

`mask` is an `int`, the Interrupts are bits in this mask, the first registered interrupt has the bit `(1 << 0)`, the n-th Interrupt the bit `(1 << (n - 1))`. If the bit is cleared (`0`) the Interrupt will be disabled.

interrupt (*address*)
Interrupts the Processor and forces him to jump to `address`. If `push_pc` is enabled this will push the PC to the stack.

register_command (*command*)
Register a Command in the Processor, the Command can now be executed by the Processor.

register_device (*device*)
Registers a device in the device *BUS*.

Invokes *register_device*.

register_memory_device (*device*)
Registers a device in the memory *BUS*.

Invokes *register_device*.

register_on_cycle_callback (*callback*)
A on cycle callback is executed in every clock cycle of the Processor. No on cycle callback modifies the state of the Processor directly, but it might cause an Interrupt.

The on cycle callback is a function without arguments:

```
def on_cycle_callback():
        print("One cycle done")
```

The return value of a callback is ignored and the callback must not raise Exceptions, but fatal Errors may stop the engine.

reset ()
Resets the control registers of the Processor_(*PC*, *ECR* and *SP*)

run()
Runs *do_cycle*, until either a stop bit in the *ECR* is set (see *EnigneControlBits*), or if an Exception in *do_cycle* occurs.

setup_done()
Finish the setup of the Processor.

This should be the last call before the Processor is used. Sets the *internal constants* (used by the assembler) and sets the Stack Pointer to RAMEND_HIGH, if there is a RAM attached. If there is no RAM attached, SP will stay `0`.

If there is a RAM attached `push_pc` is set.

Might raise *SetupError*.

class `py_register_machine2.core.processor.`**RegisterInterface**(*registers=[]*, *debug=0*, *width=64*)
Used by the Processor to perform read/write operations on the registers.

add_register(*register*)
Adds the *Register* `register` to the interface.

Will raise a *SetupError* if the interface is locked (because it is running) or if there is already a Register with the name of the new Register or if the number of Registers would exceed the size of the interface.

Returns the index of the new Register

read(*name_or_index*)
Read a word from the Register with the name `name_or_index` or with the index `name_or_index`. `name_or_index` hat to be either `str` or `int`. If the type of `name_or_index` is wrong an AttributeError will be raised.

If there is no Register with the specified name or index, a NameError will be raised.

write(*name_or_index*, *word*)
Write a word in the Register with the name `name_or_index` or with the index `name_or_index`. `name_or_index` hat to be either `str` or `int`. If the type of `name_or_index` is wrong an AttributeError will be raised.

If there is no Register with the specified name or index, a NameError will be raised.

exception `py_register_machine2.core.processor.`**SIGSEGV**(**args*)
Raised if an invalid memory command or opcode occurs.

exception `py_register_machine2.core.processor.`**SetupError**(**args*)
Raised if the setup is invalid.

Commands

py_register_machine2.core.commands: Abstract Commands

class `py_register_machine2.core.commands.`**ArithmeticCommand**(*mnemonic*, *opcode*, *function*)
Used for calculation commands, `numargs` is always 2, both arguments are Registers.

Example: The `add` command:

```
add_function = lambda a,b: a+b
add_command = ArithmeticCommand("add", 2, add_function)
```

exec(*operand1*, *operand2*)
Uses two operands and performs a function on their content.:

```
operand1 = function(operand1, operand2)
```

class py_register_machine2.core.commands.**BaseCommand**(*mnemonic*, *opcode*, *numargs*, *argtypes*)
The base class for Commands.

Every Command has to be derived from BaseCommand and provide the following functions:

- *exec*
- *numargs*
- *mnemonic*
- *opcode*
- *argtypes*

argtypes()
Return a list of strings defining the argument types, i.e.:

```
["register", "register"]
```

exec(**args*)
Exec will execute the Action of the Command. The method will be provided with *numargs* arguments and might read/write data via the attributes `register_interface`, `membus` and `devbus` provided once the Command is registered in the *Processor* using *register_command*.

mnemonic()
Returns the mnemonic of the command (`str`). Used by the Assembler and Disassembler.

numargs()
Returns the number of needed arguments. Used in the `fetch-operands-phase`.

opcode()
Returns the opcode of the command (`int`). Used by the Assembler and in the `decode-command-phase`.

class py_register_machine2.core.commands.**FunctionCommand**(*mnemonic*, *opcode*, *numargs*, *function*, *argtypes*)
Provides a basic handle to create Commands.

The argument `function` is a function with at least three arguments:

1. `register_interface`
2. `memory_BUS`
3. `device_BUS`

The function will be able to access the Processor's *RegisterInterface* and *BUS* es through this arguments.

If the function needs any operands the number of additional arguments have to be in `numargs`

For arithmetic commands (like `add`, `mul`,...) see *ArithmeticCommand*.

Example: `ld` Command:

```
def ld_function(register_interface, memory_BUS, device_BUS, addr_from, to):
        from_ = register_interface.read(addr_from)

        word = memory_BUS.read_word(from_)
```

```
        register_interface.write(to, word)

ld_command = FunctionCommand("ld", 34, 2, ld_function, ["const", "register"])
```

Example: `nop` Command:

```
def nop_function(register_interface, memory_BUS, device_BUS):
        return
nop_command = FunctionCommand("nop", 36, 0, nop_function, [])
```

Interrupts

py_register_machine2.core.interrupts: Basic module to provide Interrupts.

class `py_register_machine2.core.interrupts.`**`Autoreset`**(*name*, *processor*, *overflow_size*)
A really rude form of the Watchdog. This Interrupt will force the Processor to jump to offset `0`.

class `py_register_machine2.core.interrupts.`**`Counter`**(*address*, *name*, *processor*, *overflow_size*)
A Counter/Timer implementation.

The `__init__` method will inject an `on_cycle_callback` into the Processor. This callback will increment the internal counter variable by one. If the internal counter reaches a predefined value the `interrupt` method will be invoked.

class `py_register_machine2.core.interrupts.`**`Interrupt`**(*address*, *name*, *processor*)
The Base Class for Interrupts. If `Interrupt.interrupt` is invoked this will invoke `Processor.interrupt` and provide the address of the Interrupt.

This will allow one to place an ISR (Interrupt Service Routine) at this address.

`interrupt`()
Will interrupt the Processor, if the Interrupt is enabled.

CHAPTER

FOUR

PY_REGISTER_MACHINE2.COMMANDS

Basic Commands

py_register_machine2.commands.basic_commands: The most important commands

mnemonic	opcode	Description
mov a b	0x01	copy content from register a to register b [1]
pld a b	0x02	load from address in register a to register b [1]
pst a b	0x03	store register a to address in register b [1]
ld a b	0x04	load from a into register b [1]
st a b	0x05	store from register a to address b [1]
add a b	0x06	b = a + b
sub a b	0x07	b = a - b
mul a b	0x08	b = a * b
div a b	0x09	b = a / b (integer division)
jmp a	0x0a	pc = pc + a - 2 [3]
in a b	0x0b	read from address in register a to register b [2]
out a b	0x0c	write register a to address in register b [2]
inc a	0x0d	increase register a
dec a	0x0f	decrease register a
jne a b	0x10	if a != 0: pc += b - 3
jeq a b	0x11	if a == 0: pc += b - 3
jle a b	0x12	if a <= 0: pc += b - 3
jlt a b	0x13	if a < 0: pc += b - 3
jge a b	0x14	if a >= 0: pc += b - 3
jgt a b	0x15	if a > 0: pc += b - 3
ldi a b	0x16	Load immediate a into Register b
sjmp a	0x17	pc = a - 2 [3]

A list of all commands is `basic_commands`. **py_register_machine2.commands.stack_based**: A bunch of stack based commands

[1] stores and loads to/from the memory BUS

[3] both the fetch opcode and fetch arguments increases the PC, so we need to undo this.

[2] reads and writes from/to the device BUS

mnemonic	opcode	Description
push a	0x18	*(SP–) = a
pop a	0x19	a = *(++SP)
call a	0x1a	*(SP–) = PC; PC += a
scall a	0x1b	*(SP–) = PC; PC = a
ret	0x1c	PC = *(++SP)

all stackbased commands are available in the list `stack_based_commands`.

CHAPTER FIVE

PY_REGISTER_MACHINE2.MACHINES

A collection of register machines in different states of configuration.

Small Machines

py_register_machine2.machines.small: a collection of small ready to use register machines

About Register Machine Definitions

All Register Machine Definitions are functions that take at least `0` arguments and return a `tuple` with length `4`: `(Processor, ROM, RAM, Flash)`

Example: a simple Register Machine:

```
from py_register_machine2.core import memory, processor, register, device
from py_register_machine2.commands.basic_commands import basic_commands
def simple_register_machine():
        rom = memory.ROM(60)
        ram = memory.RAM(70)
        flash = device.Flash(300)

        proc = processor.Processor()
        proc.register_memory_device(rom)
        proc.register_memory_device(ram)
        proc.register_device(flash)

        r0 = register.Register("r0")
        r1 = register.Register("r1")
        r2 = register.Register("r2")
        r3 = register.Register("r3")
        r4 = register.Register("r4")
        r5 = register.Register("r5")

        for reg in (r0, r1, r2, r3, r4, r5):
                proc.add_register(reg)
        for command in basic_commands:
                proc.register_command(command)

        return (proc, rom, ram, flash)
```

`py_register_machine2.machines.small.`**`small_register_machine`**`(`*rom_size=50, ram_size=200, flash_size=500*`)`

An unprogrammend Register Machine with

- one OutputRegister to `sys.stdout` (`out0`)
- 15 General Purpose Register (`r0 - r14`)

returns : `(Processor, ROM, RAM, Flash)`

CHAPTER SIX

TOOLS

Basic Assembler

py_register_machine2.tools.assembler.assember: The Basic Assembler

Just replaces mnemonics with opcodes and handles references.

exception `py_register_machine2.tools.assembler.assembler.`**`ArgumentError`**(**args*)
Raised if an argument does not fit the requirements.

exception `py_register_machine2.tools.assembler.assembler.`**`AssembleError`**(**args*)
Rasied if the assemler terminates without success.

class `py_register_machine2.tools.assembler.assembler.`**`Assembler`**(*processor, open_stream, directives=[], commentstarts=[';']*)
Reads assembly code from `open_stream` and converts it to a `list` of integers that can be programmed to the ROM or the Flash.

Stages:

Split Run Reads the complete file and converts it to a list of tuples: `[(lineno, "command", <command>, arguments), ...]`` or `[(lineno, "data", <data description>, data), ...]``

Argument Run Checks and converts the arguments. Unconvertable `str` objects are interpreted as addresses.

Dereference Run Handles references.

Program Run Generates one iterable of integers

`add_ref`(*wordlist*)
Adds a reference.

`argument_run`(*sp_r*)
Converts Arguments according to `to_int`

`assemble`()
Chains *split_run*, *argument_run*, *dereference_run* and *program_run*.

`checkargs`(*lineno, command, args*)
Check if the arguments fit the requirements of the command.

Raises *ArgumentError*, if an argument does not fit.

`convert_args`(*command, args*)
Converts `str -> int` or `register -> int`.

dereference_run(*arg_r*)
Converts the commands to opcodes and inserts the (relative or static) references.

getdirective(*name*)
Returns the directive with the name `name`.

handle_directive(*words*)
handles directives: adds the reference and allocates space for the content

isdirective(*words*)
Check if the line `words` is a directive.

program_run(*der_r*)
Generates an iterable that can be programmed onto the register machine.

split_run()
Splits the assembly code into

- commands
- directives
- jump marks

`py_register_machine2.tools.assembler.assembler.`**isreference**(*wordlist*)
if the line is a reference (jump mark), return true

Assembly Directives

class `py_register_machine2.tools.assembler.directives.`**BaseDirective**(*name*)
Every Directive has to provide the following Attributes/Methods:

- `name` (like `.set`)
- `get_words(line)`: return the data to store
- `get_word_count(line)`: return the number of words to store
- `isstatic()` returns True, if the reference should be static

class `py_register_machine2.tools.assembler.directives.`**ConvertingDirective**(*name*, *function*)
The function `function` will have to take the rest of the line (as a list) and convert it to an iterable of `int` objects

Example: The `.string` directive:

```
# usage: .string name string
# ie.: .string foo this is a test

def string_function(line):
        line = " ".join(line)
        res = []
        for char in line:
                res.append(ord(char))
        return res
```

class `py_register_machine2.tools.assembler.directives.`**Padding**(*name='.padd'*)
Usage:

```
.padd name n v
```

Fills the next n words with v.

class `py_register_machine2.tools.assembler.directives.`**`Zeros`**(*name='.zeros'*)
Usage:

```
.zeros name n
```

Fills the next n words with zeros.

CHAPTER

SEVEN

DEVELOPER'S GUIDE

Contributing

Because PyRegisterMachine2 is GPL'ed everybody is able to modify the code and redistribute the (modified) code, according to the GNU GPL v3.

If you want to share your changes on the main branch, just send a pull request (via GitHub).

Engine Tools

Conversions

A collection of conversion functions/generators.

`py_register_machine2.engine_tools.conversions.`**`bytes_to_int`**`(`*bytes_, width=None*`)`

Converts the `bytes` object `bytes_` to an `int`. If `width` is none, `width = len(byte_) * 8` is choosen.

See also: *int_to_bytes*

Example

```
>>> from py_register_machine2.engine_tools.conversions import *
>>> i = 4012
>>> int_to_bytes(i)
b'¬'
>>> bytes_to_int(int_to_bytes(i)) == i
True
```

`py_register_machine2.engine_tools.conversions.`**`int_to_bytes`**`(`*int_, width=None*`)`

Converts the `int` `int_` to a `bytes` object. `len(result) == width`.

If `width` is None, a number of bytes that is able to hold the number is choosen, depending on `int_.bit_length()`.

See also: *bytes_to_int*

`py_register_machine2.engine_tools.conversions.`**`to_int`**`(`*argument*`)`

Converts the `str` argument to an integer:

```
>>> from py_register_machine2.engine_tools.conversions import *
>>> to_int("0x04")
4
```

```
>>> to_int("'a'")
97
```

Operations

Operations used by the engine.

py_register_machine2.engine_tools.operations.**bitsetxor**(*b1*, *b2*)
If b1 and b2 would be `int` s this would be `b1 ^ b2` :

```
>>> from py_register_machine2.engine_tools.operations import bitsetxor
>>> b1 = [1, 1, 1, 1]
>>> b2 = [1, 1, 0, 1]
>>> bitsetxor(b1, b2)
[0, 0, 1, 0]
>>> bin(0b1111 ^ 0b1101)
'0b10'
```

CHAPTER

EIGHT

INSTALLING PYREGISTERMACHINE2

From Source

PyRegisterMachine2 is a simple python3-package, so the only thing one has to do is to place the folder in the `$PYTHONPATH`. One can get the `$PYTHONPATH` in the following ways:

```
echo $PYTHONPATH
python3 -c "import sys; print(sys.path)"
```

- Local Installation

Usually the local path is `/home/<username>/.local/lib/python3.5/site-packages`, so you are able to install the package via git:

```
cd /home/daniel/.local/lib/python3.5/site-packages
git clone https://github.com/daknuett/py_register_machine2
```

- Global Installation:

```
    cd /usr/local/lib/python3.5/dist-packages
    git clone https://github.com/daknuett/py_register_machine2
```

Using PyPi

Use

```
python3 -m pip install py_register_machine2
```

To install py_register_machine2 using PyPi.

p