
# ppci Documentation

***Release 0.5***

**Windel Bouwman**

August 06, 2016

Contents

CHAPTER 1

# Quickstart

## 1.1 Installation

Install ppci in a virtualenv environment:

```
$ virtualenv sandbox
$ source sandbox/bin/activate
(sandbox) $ pip install ppci
(sandbox) $ ppci-build.py -h
```

If ppci installed correcly, you will get a help message of the *ppci-build.py* commandline tool.

## 1.2 Example projects

Download and unzip `examples.zip` to checkout some demo projects that can be build using ppci.

**stm32f4 example**

To build the blinky project do the following:

```
$ cd examples/blinky
$ ppci-build.py
```

Flash the hexfile using your flashtool of choice on the stm32f4discovery board and enjoy the magic.

**arduino example**

To build and the arduino blink led example, follow the following commands:

```
$ cd examples/avr/arduino-blinky
$ ppci-build.py
$ avrdude -v -P /dev/ttyACM0 -c arduino -p m328p -U flash:w:blinky.hex
```

**Linux x86_64 example**

To build the hello world for 64-bit linux, go here:

```
$ cd examples/linux64/hello
$ ppci-build.py
$ ./hello
```

Or run the snake demo under linux:

```
$ cd examples/linux64/snake
$ ppci-build.py
$ ./snake
```

## 1.3 Next steps

If you have checked out the examples, head over to the api and reference sections to learn more!

CHAPTER 2

# Reference

## 2.1 Api

The ppci library provides provides an intuitive api to the compiler, assembler and other tools. For example to assemble, compile, link and objcopy the msp430 blinky example project, the api can be used as follows:

```
>>> from ppci.api import asm, c3c, link, objcopy
>>> march = "msp430"
>>> o1 = asm('examples/msp430/blinky/boot.asm', march)
>>> o2 = c3c(['examples/msp430/blinky/blinky.c3'], [], march)
>>> o3 = link([o2, o1], 'examples/msp430/blinky/msp430.mmap', march)
>>> objcopy(o3, 'flash', 'hex', 'blinky_msp430.hex')
```

Instead of using the api, a set of commandline tools are also prodived. The api module contains a set of handy functions to invoke compilation, linking and assembling.

ppci.api.**asm**(*source*, *march*, *debug=False*)

Assemble the given source for machine march.

**Parameters**

- **source** (*str*) – can be a filename or a file like object.
- **march** (*str*) – march can be a *ppci.arch.arch.Architecture* instance or a string indicating the machine architecture.
- **debug** – generate debugging information

**Returns** A ppci.binutils.objectfile.ObjectFile object

```
>>> import io
>>> from ppci.api import asm
>>> source_file = io.StringIO("db 0x77")
>>> obj = asm(source_file, 'arm')
>>> print(obj)
CodeObject of 1 bytes
```

ppci.api.**c3c**(*sources*, *includes*, *march*, *opt_level=0*, *reporter=None*, *debug=False*)

Compile a set of sources into binary format for the given target.

**Parameters**

- **sources** – a collection of sources that will be compiled.
- **includes** – a collection of sources that will be used for type and function information.
- **march** – the architecture for which to compile.
- **reporter** – reporter to write compilation report to
- **debug** – include debugging information

**Returns** An object file

```
>>> import io
>>> from ppci.api import c3c
>>> source_file = io.StringIO("module main; var int a;")
>>> obj = c3c([source_file], [], 'arm')
>>> print(obj)
CodeObject of 4 bytes
```

ppci.api.**link**(*objects*, *layout=None*, *use_runtime=False*, *partial_link=False*, *reporter=None*, *debug=False*)

Links the iterable of objects into one using the given layout.

**Parameters**

- **objects** – a collection of objects to be linked together.
- **use_runtime** (*bool*) – also link compiler runtime functions
- **debug** (*bool*) – when true, keep debug information. Otherwise remove this debug information from the result.

**Returns** The linked object file

```
>>> import io
>>> from ppci.api import asm, c3c, link
>>> asm_source = io.StringIO("db 0x77")
>>> obj1 = asm(asm_source, 'arm')
>>> c3_source = io.StringIO("module main; var int a;")
>>> obj2 = c3c([c3_source], [], 'arm')
>>> obj = link([obj1, obj2])
>>> print(obj)
CodeObject of 8 bytes
```

ppci.api.**objcopy**(*obj*, *image_name*, *fmt*, *output_filename*)

Copy some parts of an object file to an output

ppci.api.**bfcompile**(*source*, *target*, *reporter=None*)

Compile brainfuck source into binary format for the given target

**Parameters**

- **source** – a filename or a file like object.
- **march** – a architecture instance or a string indicating the target.

**Returns** A new object.

```
>>> import io
>>> from ppci.api import bfcompile
>>> source_file = io.StringIO(">>[-]<<[->>+<<]")
>>> obj = bfcompile(source_file, 'arm')
>>> print(obj)
CodeObject of ... bytes
```

ppci.api.**construct**(*buildfile*, *targets=()*)

Construct the given buildfile. Raise task error if something goes wrong.

ppci.api.**optimize**(*ir_module*, *level=0*, *reporter=None*, *debug_db=None*)

Run a bag of tricks against the ir-code.

This is an in-place operation!

**Parameters**

- **ir_module** (ppci.ir.Module) – The ir module to optimize.
- **level** – The optimization level, 0 is default. Can be 0,1,2 or s 0: No optimization 1: some optimization 2: more optimization s: optimize for size

ppci.api.**get_arch**(*arch*)
Try to return an architecture instance. arch can be a string in the form of arch:option1:option2

```
>>> from ppci.api import get_arch
>>> arch = get_arch('msp430')
>>> arch
msp430-arch
>>> type(arch)
<class 'ppci.arch.msp430.arch.Msp430Arch'>
```

## 2.2 Command line tools

This section describes the usage the commandline tools installed with ppci.

Take for example the stm32f4 blinky project. To build this project, run ppci-build.py in the project folder:

```
$ cd examples/blinky
$ ppci-build.py
```

This command is used to construct build files.

Or specify the buildfile a the command line:

```
$ ppci-build.py -f examples/blinky/build.xml
```

Instead of relying on a build system, the c3 compiler can also be activated stand alone.

```
$ ppci-c3c.py --machine arm examples/snake/game.c3
```

### 2.2.1 ppci-c3c.py

C3 compiler. Use this compiler to produce object files from c3 sources and c3 includes. C3 includes have the same format as c3 source files, but do not result in any code.

usage: ppci-c3c.py [-h] [–log log-level] [–report report-file] [–verbose] [–version] –machine {6500,arm,avr,example,msp430,riscv,x86_64} [–mtune option] –output output-file [-i include] [-g] [-O {0,1,2,s}] source [source ...]

**source**
source file

**-h, --help**
show this help message and exit

**--log** <log-level>
Log level (info,debug,warn)

**--report**
Specify a file to write the compile report to

**--verbose, -v**
Increase verbosity of the output

**--version, -V**
Display version and exit

**--machine, -m**
target architecture

**--mtune** <option>
architecture option

**--output, -o**
output file

**-i** <include>, **--include** <include>
include file

**-g**
create debug information

**-O** {0,1,2,s}
optimize code

### 2.2.2 ppci-build.py

Build utility. Use this to execute build files.

usage: ppci-build.py [-h] [–log log-level] [–report report-file] [–verbose] [–version] [-f build-file] [target [target ...]]

**target**

**-h, --help**
show this help message and exit

**--log** <log-level>
Log level (info,debug,warn)

**--report**
Specify a file to write the compile report to

**--verbose, -v**
Increase verbosity of the output

**--version, -V**
Display version and exit

**-f** <build-file>, **--buildfile** <build-file>
use buildfile, otherwise build.xml is the default

### 2.2.3 ppci-asm.py

Assembler utility.

usage: ppci-asm.py [-h] [–log log-level] [–report report-file] [–verbose] [–version] –machine {6500,arm,avr,example,msp430,riscv,x86_64} [–mtune option] –output output-file [-g] sourcefile

**sourcefile**
the source file to assemble

**-h, --help**
show this help message and exit

**--log** <log-level>
Log level (info,debug,warn)

**--report**
Specify a file to write the compile report to

**--verbose, -v**
Increase verbosity of the output

**--version, -V**
Display version and exit

**--machine, -m**
target architecture

**--mtune** <option>
architecture option

**--output, -o**
output file

**-g, --debug**
create debug information

### 2.2.4 ppci-ld.py

Linker. Use the linker to combine several object files and a memory layout to produce another resulting object file with images.

usage: ppci-ld.py [-h] [–log log-level] [–report report-file] [–verbose] [–version] –output output-file [–layout layout-file] [-g] obj [obj ...]

**obj**
the object to link

**-h, --help**
show this help message and exit

**--log** <log-level>
Log level (info,debug,warn)

**--report**
Specify a file to write the compile report to

**--verbose, -v**
Increase verbosity of the output

**--version, -V**
Display version and exit

**--output, -o**
output file

**--layout** <layout-file>, **-L** <layout-file>
memory layout

**-g**
retain debug information

### 2.2.5 ppci-objcopy.py

Objcopy utility to manipulate object files.

usage: ppci-objcopy.py [-h] [–log log-level] [–report report-file] [–verbose] [–version] –segment SEGMENT [–output-format OUTPUT_FORMAT] input output

**input**
input file

**output**
output file

**-h, --help**
show this help message and exit

**--log** <log-level>
Log level (info,debug,warn)

**--report**
Specify a file to write the compile report to

**--verbose, -v**
Increase verbosity of the output

**--version, -V**
Display version and exit

**--segment** , **-S** 
segment to copy

**--output-format** <output_format>, **-O** <output_format>
output file format

### 2.2.6 ppci-objdump.py

Objdump utility to display the contents of object files.

usage: ppci-objdump.py [-h] [–log log-level] [–report report-file] [–verbose] [–version] [-d] obj

**obj**
object file

**-h, --help**
show this help message and exit

**--log** <log-level>
Log level (info,debug,warn)

**--report**
Specify a file to write the compile report to

**--verbose, -v**
Increase verbosity of the output

**--version, -V**
Display version and exit

**-d, --disassemble**
Disassemble contents

## 2.3 C3 language

### 2.3.1 Introduction

As an example language, the c3 language was created. As pointed out clearly in c2lang, the C language is widely used, but has some strange contraptions. These include the following:

- The include system. This results in lots of code duplication and file creation. Why would you need filenames in source code?
- The comma statement: x = a(), 2; assigns 2 to x, after calling function a.
- C is difficult to parse with a simple parser. The parser has to know what a symbol is when it is parsed. This is also referred to as the lexer hack.

In part for these reasons (and of course, for fun), C3 was created.

The hello world example in c3 is:

```
module hello;
import io;

function void main()
{
```

```
    io.println("Hello world");
}
```

### 2.3.2 Language reference

#### Modules

Modules in C3 live in file, and can be defined in multiple files. Modules can import each other by using the import statement.

For example:

*pkg1.c3*:

```
module pkg1;
import pkg2;
```

*pkg2.c3*:

```
module pkg2;
import pkg1;
```

#### Functions

Function can be defined by using the function keyword, followed by a type and the function name.

```
module example;

function void compute()
{
}

function void main()
{
    main();
}
```

#### Variables

Variables require the var keyword, and can be either global are function local.

```
module example;

var int global_var;

function void compute()
{
    var int x = global_var + 13;
    global_var = 200 - x;
}
```

#### Types

Types can be specified when a variable is declared, and also typedeffed.

```
module example;
var int number;
var int* ptr_num;
```

```
type int* ptr_num_t;
var ptr_num_t number2;
```

### If statement

The following code example demonstrates the if statement. The else part is optional.

```
module example;

function void compute(int a)
{
    var int b = 10;
    if (a > 100)
    {
        b += a;
    }

    if (b > 50)
    {
        b += 1000;
    }
    else
    {
        b = 2;
    }
}
```

### While statement

The while statement can be used as follows:

```
module example;

function void compute(int a)
{
    var int b = 10;
    while (b > a)
    {
        b -= 1;
    }
}
```

### For statement

The for statement works like in C. The first item is initialized before the loop. The second is the condition for the loop. The third part is executed when one run of the loop is done.

```
module example;

function void compute(int a)
{
    var int b = 0;
    for (b = 100; b > a; b -= 1)
    {
        // Do something here!
    }
}
```

## 2.4 Build system

It can be convenient to bundle a series of build steps into a script, for example a makefile. Instead of depending on make, yet another build tool was created. The build specification is specified in xml. Much like msbuild and Ant.

A project can contain a build.xml file which describes how the project should be build. The name of the file can be build.xml or another filename. This file can than be given to *ppci-build.py*.

An example build file:

```
<project name="Snake" default="snake">
    <import name="ppci.buildtasks" />

    <target name="snake">
        <assemble
            source="../startup.asm"
            arch="arm:thumb"
            output="startup.oj" />
        <compile
            arch="arm:thumb"
            sources="../../src/snake/*.c3;../bsp.c3;../../../librt/io.c3"
            output="rest.oj"
            report="snake_report.html"/>
        <link output="snake.oj"
            layout="../memlayout.mmap"
            objects="startup.oj;rest.oj" />
        <objcopy
            objectfile="snake.oj"
            imagename="flash"
            format="bin"
            output="snake.bin" />
    </target>

</project>
```

### 2.4.1 Projects

The root element of a build file is the project tag. This tag contains a name and optionally a default target attribute. When no target is given when building the project, the default target is selected.

### 2.4.2 Targets

Like make, targets can depend on eachother. Then one target is run, the build system makes sure to run depending targets first. Target elements contain a list of tasks to perform.

### 2.4.3 Tasks

The task elements are contained within target elements. Each task specifies a build action. For example the link task takes multiple object files and combines those into a merged object.

## 2.5 IR-code

The purpose of an intermediate representation (IR) of a program is to decouple the implementation of a front-end from the implementation of a back-end. That is, front ends generate IR-code, optimizers optimize this code and lastly backends transform it into machine code or something else.

A good IR has several characteristics:

- It should be simple enough for front-ends to generate code.
- It should be rich enough to exploit the target instructions best.

The IR in the ppci.ir module has the following properties:

- It is static single assignment form. Meaning a value can only be assigned once, and is then never changed. This has several advantages.
- It contains only basic types. Structures, arrays and void types are not represented.

### 2.5.1 Top level structure

The IR-code is implemented in the ir package.

**class** `ppci.ir.`**`Module`**(*name*)
Container unit for variables and functions.

**`add_variable`**(*variable*)
Add a variable to this module

**`functions`**
Get all functions of this module

**`variables`**
Get all variables of this module

**class** `ppci.ir.`**`Variable`**(*name*, *amount*)
Global variable, reserves room in the data area. Has name and size

**class** `ppci.ir.`**`SubRoutine`**(*name*)
Base class of function and procedure. These two differ in that a function returns a value, where as a procedure does not.

Design trade-off: In C, a void type is introduced to permit functions that return nothing (void). This seems somewhat artificial, but keeps things simple for the users. In pascal, the procedure and function types are explicit, and the void type is not needed. This is also the approach taken here.

So instead of a Function and Call types, we have Function, Procedure, FunctionCall and ProcedureCall types.

**`add_block`**(*block*)
Add a block to this function

**`remove_block`**(*block*)
Remove a block from this function

**class** `ppci.ir.`**`Procedure`**(*name*)
A procedure definition that does not return a value

**class** `ppci.ir.`**`Function`**(*name*, *return_ty*)
Represents a function.

**class** `ppci.ir.`**`Block`**(*name*)
Uninterrupted sequence of instructions. A block is properly terminated if its last instruction is a `FinalInstruction`.

**`add_instruction`**(*instruction*)
Add an instruction to the end of this block

**`is_closed`**
Determine whether this block is propert terminated

**`is_empty`**
Determines whether the block is empty or not

**`predecessors`**
Return all predecessing blocks

**remove_instruction**(*instruction*)
Remove instruction from block

**successors**
Get the direct successors of this block

### 2.5.2 Types

Only simple types are available.

ppci.ir.**ptr**
Pointer type

ppci.ir.**i64**
Signed 64-bit type

ppci.ir.**i32**
Signed 32-bit type

ppci.ir.**i16**
Signed 16-bit type

ppci.ir.**i8**
Signed 8-bit type

ppci.ir.**u64**
Unsigned 64-bit type

ppci.ir.**u32**
Unsigned 32-bit type

ppci.ir.**u16**
Unsigned 16-bit type

ppci.ir.**u8 = u8**
Unsigned 8-bit type

ppci.ir.**f64**
64-bit floating point type

ppci.ir.**f32**
32-bit floating point type

### 2.5.3 Instructions

The following instructions are available.

**Memory instructions**

**class** ppci.ir.**Load**(*address*, *name*, *ty*, *volatile=False*)
Load a value from memory

**class** ppci.ir.**Store**(*value*, *address*, *volatile=False*)
Store a value into memory

**class** ppci.ir.**Alloc**(*name*, *amount*)
Allocates space on the stack. The type of this value is a ptr

**Data instructions**

**class** ppci.ir.**Const**(*value*, *name*, *ty*)
Represents a constant value

**class** ppci.ir.**Binop**(*a*, *operation*, *b*, *name*, *ty*)
Generic binary operation

**class** ppci.ir.**Cast**(*value*, *name*, *ty*)
Base type conversion instruction

**Control flow instructions**

**class** ppci.ir.**ProcedureCall**(*function_name*, *arguments*)
Call a procedure with some arguments

**class** ppci.ir.**FunctionCall**(*function_name*, *arguments*, *name*, *ty*)
Call a function with some arguments and a return value

**class** ppci.ir.**Jump**(*target*)
Jump statement to another block within the same function

**class** ppci.ir.**CJump**(*a*, *cond*, *b*, *lab_yes*, *lab_no*)
Conditional jump to true or false labels.

**class** ppci.ir.**Return**(*result*)
This instruction returns a value and exits the function.

**class** ppci.ir.**Exit**
Instruction that exits the procedure.

**Other**

**class** ppci.ir.**Phi**(*name*, *ty*)
Imaginary phi instruction to make SSA possible.

**class** ppci.ir.**Undefined**(*name*, *ty*)
Undefined value, this value must never be used.

### 2.5.4 Abstract instruction classes

There are some abstract instructions, which cannot be used directly but serve as base classes for other instructions.

**class** ppci.ir.**Instruction**
Base class for all instructions that go into a basic block

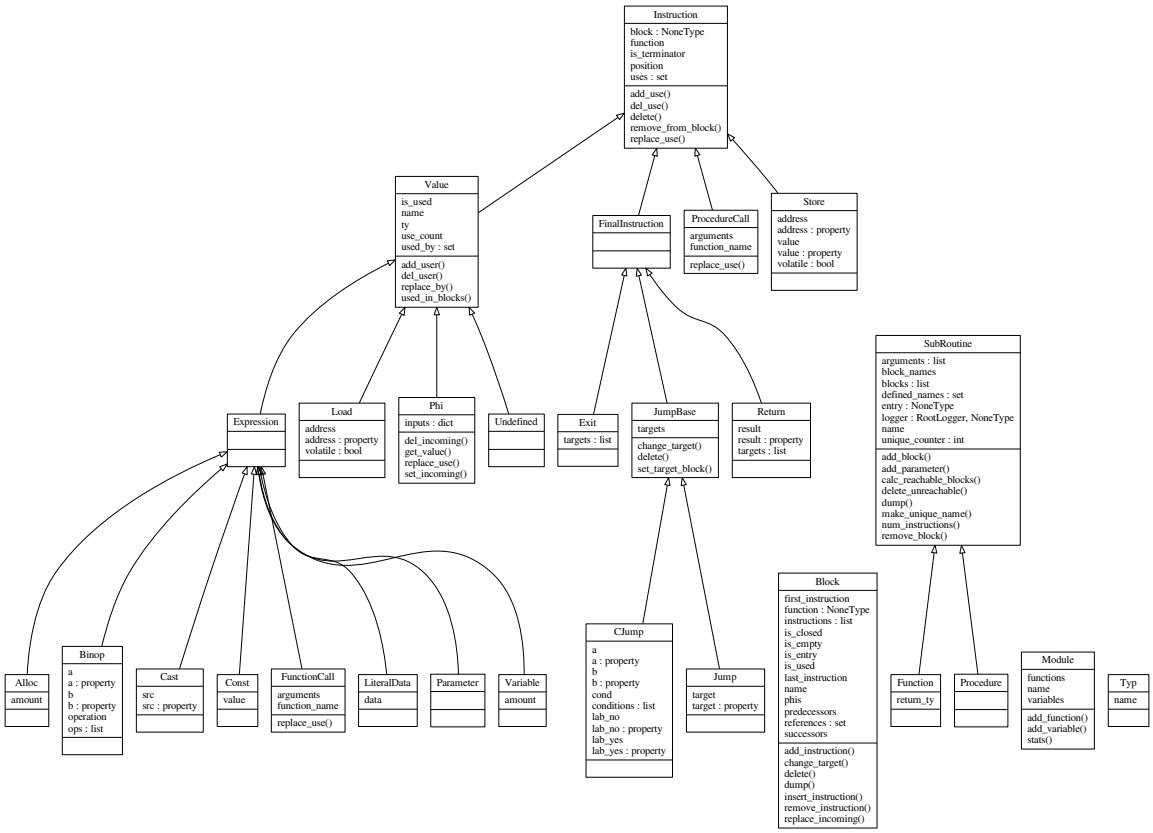
**function**
Return the function this instruction is part of

**class** ppci.ir.**Value**(*name*, *ty*)
An instruction that results in a value has a type and a name

**is_used**
Determine whether this value is used anywhere

**class** ppci.ir.**FinalInstruction**
Final instruction in a basic block

### 2.5.5 Uml




## 2.6 Debug

When an application is build, it often has to be debugged. This section describes the peculiarities of debugging.

### 2.6.1 Debugger

The debugger class is the main piece of the debugger. This is created for a specific architecture and is given a driver to communicate with the target hardware.

**class** `ppci.binutils.dbg.`**`Debugger`**(*arch*, *driver*)

Main interface to the debugger. Give it a target architecture for which it must debug and driver plugin to connect to hardware.

**`clear_breakpoint`**(*filename*, *row*)

Remove a breakpoint

**`read_mem`**(*address*, *size*)

Read binary data from memory location

**`run`**()

Run the program

**`set_breakpoint`**(*filename*, *row*)

Set a breakpoint

**`step`**()

Single step the debugged program

**`stop`**()

Interrupt the currently running program

**write_mem**(*address*, *data*)
Write binary data to memory location

One of the classes that uses the debugger is the debug command line interface.

**class** ppci.binutils.dbg_cli.**DebugCli**(*debugger*)
Implement a console-based debugger interface.

To connect to your favorite hardware, subclass the DebugDriver class.

**class** ppci.binutils.dbg.**DebugDriver**
Inherit this class to expose a target interface. This class implements primitives for a given hardware target.

The following class can be used to connect to a gdb server:

**class** ppci.binutils.dbg_gdb_client.**GdbDebugDriver**(*arch*, *port=1234*)
Implement debugging via the GDB remote interface.

GDB servers can communicate via the RSP protocol.

Helpfull resources:

**http://www.embecosm.com/appnotes/ean4/** embecosm-howto-rsp-server-ean4-issue-2.html

### 2.6.2 Debug info file formats

Debug information is of a complex nature. Various file formats exist to store this information. This section gives a short overview of the different formats.

#### pdb format

This is the microsoft debug format.

https://en.wikipedia.org/wiki/Program_database

#### Dwarf format

How a linked list is stored in dwarf format.

```
struct ll {
  int a;
  struct ll *next;
};
```

```
<1><57>: Abbrev Number: 3 (DW_TAG_base_type)
    <58>   DW_AT_byte_size   : 4
    <59>   DW_AT_encoding    : 5    (signed)
    <5a>   DW_AT_name        : int
 <1><5e>: Abbrev Number: 2 (DW_TAG_base_type)
    <5f>   DW_AT_byte_size   : 8
    <60>   DW_AT_encoding    : 5    (signed)
    <61>   DW_AT_name        : (indirect string, offset: 0x65): long int
 <1><65>: Abbrev Number: 2 (DW_TAG_base_type)
    <66>   DW_AT_byte_size   : 8
    <67>   DW_AT_encoding    : 7    (unsigned)
    <68>   DW_AT_name        : (indirect string, offset: 0xf6): sizetype
 <1><6c>: Abbrev Number: 2 (DW_TAG_base_type)
    <6d>   DW_AT_byte_size   : 1
    <6e>   DW_AT_encoding    : 6    (signed char)
    <6f>   DW_AT_name        : (indirect string, offset: 0x109): char
 <1><73>: Abbrev Number: 4 (DW_TAG_structure_type)
    <74>   DW_AT_name        : ll
    <77>   DW_AT_byte_size   : 16
```

```
    <78>   DW_AT_decl_file   : 1
    <79>   DW_AT_decl_line   : 4
    <7a>   DW_AT_sibling     : <0x95>
 <2><7e>: Abbrev Number: 5 (DW_TAG_member)
    <7f>   DW_AT_name        : a
    <81>   DW_AT_decl_file   : 1
    <82>   DW_AT_decl_line   : 5
    <83>   DW_AT_type        : <0x57>
    <87>   DW_AT_data_member_location: 0
 <2><88>: Abbrev Number: 6 (DW_TAG_member)
    <89>   DW_AT_name        : (indirect string, offset: 0xf1): next
    <8d>   DW_AT_decl_file   : 1
    <8e>   DW_AT_decl_line   : 6
    <8f>   DW_AT_type        : <0x95>
    <93>   DW_AT_data_member_location: 8
 <2><94>: Abbrev Number: 0
 <1><95>: Abbrev Number: 7 (DW_TAG_pointer_type)
    <96>   DW_AT_byte_size   : 8
    <97>   DW_AT_type        : <0x73>
```

## 2.7 Architecture

The arch contains processor architecture descriptions.

**class** ppci.arch.arch.**Architecture**(*options=None*, *register_classes=()*)

Base class for all targets

**between_blocks**(*frame*)

Generate any instructions here if needed between two blocks

**determine_arg_locations**(*arg_types*)

Determine argument location for a given function

**determine_rv_location**(*ret_type*)

Determine the location of a return value of a function given the type of return value

**epilogue**(*frame*)

Generate instructions for the epilogue of a frame

**gen_call**(*value*)

Generate a sequence of instructions for a call to a label. The actual call instruction is not yet used until the end of the code generation at which time the live variables are known.

**gen_copy_rv**(*res_type*, *res_var*)

Generate a sequence of instructions for copying the result of a function to the correct variable. Override this function when needed. The basic implementation simply moves the result.

**gen_fill_arguments**(*arg_types*, *args*, *live*)

Generate a sequence of instructions that puts the arguments of a function to the right place.

**get_compiler_rt_lib**()

Gets the runtime for the compiler. Returns an object with the compiler runtime for this architecture

**get_reg_class**(*bitsize=None*, *ty=None*)

Look for a register class

**get_reloc**(*name*)

Retrieve a relocation identified by a name

**get_size**(*typ*)

Get type of ir type

**has_option**(*name*)

Check for an option setting selected

**make_call**(*frame*, *vcall*)
Actual call instruction implementation

**make_id_str**()
Return a string uniquely identifying this machine

**move**(*dst*, *src*)
Generate a move from src to dst

**new_frame**(*frame_name*, *function*)
Create a new frame with name frame_name for an ir-function

**prologue**(*frame*)
Generate instructions for the epilogue of a frame

**runtime**
Gets the runtime for the compiler. Returns an object with the compiler runtime for this architecture

**value_classes**
Get a mapping from ir-type to the proper register class

**class** ppci.arch.arch.**Frame**(*name*, *arg_locs*, *live_in*, *rv*, *live_out*)
Activation record abstraction. This class contains a flattened function. Instructions are selected and scheduled at this stage. Frames differ per machine. The only thing left to do for a frame is register allocation.

**add_constant**(*value*)
Add constant literal to constant pool

**alloc**(*size*)
Allocate space on the stack frame and return the offset

**emit**(*ins*)
Append an abstract instruction to the end of this frame

**insert_code_after**(*instruction*, *code*)
Insert a code sequence after an instruction

**insert_code_before**(*instruction*, *code*)
Insert a code sequence before an instruction

**live_ranges**(*vreg*)
Determine the live range of some register

**live_regs_over**(*instruction*)
Determine what registers are live along an instruction. Useful to determine if registers must be saved when making a call

**new_label**()
Generate a unique new label

**new_name**(*salt*)
Generate a new unique name

**new_reg**(*cls*, *twain=''*)
Retrieve a new virtual register

**class** ppci.arch.isa.**Isa**
Container type for an instruction set. Contains a list of instructions, mappings from intermediate code to instructions.

Isa's can be merged into new isa's which can be used to define target. For example the arm without FPU can be combined with the FPU isa to expand the supported functions.

**add_instruction**(*i*)
Register an instruction into this ISA

**pattern**(*non_term*, *tree*, *condition=None*, *size=1*, *cycles=1*, *energy=1*)
Decorator function that adds a pattern.

**peephole**(*function*)
Add a peephole optimization function

**register_pattern**(*pattern*)
Add a pattern to this isa

**register_relocation**(*relocation*)
Register a relocation into this isa

**class** ppci.arch.isa.**Register**(*name*, *num=None*, *aliases=()*)
Baseclass of all registers types

**is_colored**
Determine whether the register is colored

**class** ppci.arch.isa.**Instruction**(**args*, ***kwargs*)
Base instruction class. Instructions are automatically added to an isa object. Instructions are created in the following ways:

- From python code, by using the instruction directly: self.stream.emit(Mov(r1, r2))
- By the assembler. This is done via a generated parser.
- By the instruction selector. This is done via pattern matching rules

Instructions can then be emitted to output streams.

An instruction can be colored or not. When all its used registers are colored, the instruction is also colored.

**classmethod decode**(*data*)
Decode data into an instruction of this class

**defined_registers**
Return a set of all defined registers

**encode**()
Encode the instruction into binary form, returns bytes for this instruction.

**is_colored**
Determine whether all registers of this instruction are colored

**reads_register**(*register*)
Check if this instruction reads the given register

**registers**
Determine all registers used by this instruction

**replace_register**(*old*, *new*)
Replace a register usage with another register

**set_all_patterns**()
Look for all patterns and apply them to the tokens

**used_registers**
Return a set of all registers used by this instruction

**writes_register**(*register*)
Check if this instruction writes the given register

## 2.8 Backends

This page lists the available backends.

Status matrix:

| feature | 6500 | arm | avr | msp430 | riscv | stm8 | x86_64 |
|---|---|---|---|---|---|---|---|
| Samples build | | yes | yes | yes | yes | | yes |
| Samples run | | yes | | | | | yes |
| gdb remote client | | | yes | | yes | | |
| percentage complete | 1% | 70% | 50% | 20% | 70% | 1% | 60% |

### 2.8.1 6500

**class** ppci.arch.mos6500.**Mos6500Arch**(*options=None*)

### 2.8.2 arm

Arm machine specifics. The arm target has several options:

- thumb: enable thumb mode, emits thumb code

**class** ppci.arch.arm.**ArmArch**(*options=None*)
Arm machine class.

### 2.8.3 avr

The is the avr backend.

**class** ppci.arch.avr.**AvrArch**(*options=None*)
Avr architecture description.

**class** ppci.arch.avr.registers.**AvrRegister**(*name*, *num=None*, *aliases=()*)

**class** ppci.arch.avr.registers.**AvrWordRegister**(*name*, *num=None*, *aliases=()*)
Register covering two 8 bit registers

See also:

https://gcc.gnu.org/wiki/avr-gcc

### 2.8.4 msp430

To flash the msp430 board, the following program can be used:

http://www.ti.com/tool/msp430-flasher

**class** ppci.arch.msp430.**Msp430Arch**(*options=None*)
Texas Instruments msp430 target architecture

### 2.8.5 risc-v

See also: http://riscv.org

Contributed by Michael.

**class** ppci.arch.riscv.**RiscvArch**(*options=None*)

### 2.8.6 stm8

STM8 is an 8-bit processor, see also: http://www.st.com/stm8

### 2.8.7 x86_64

For a good list of op codes, checkout:

http://ref.x86asm.net/coder64.html

For an online assembler, checkout:

https://defuse.ca/online-x86-assembler.htm

#### Linux

For a good list of linux system calls, refer:

http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64

**class** ppci.arch.x86_64.**X86_64Arch**(*options=None*)
x86_64 architecture

## 2.9 How to write a new backend

This section describes how to add a new backend. The best thing to do is to take a look at existing backends, like the backends for ARM and X86_64.

A backend consists of the following parts:

1. register descriptions
2. instruction descriptions
3. template descriptions
4. architecture description

### 2.9.1 Register description

A backend must describe what kinds of registers are available. To do this define for each register class a subclass of *ppci.arch.isa.Register*.

There may be several register classes, for example 8-bit and 32-bit registers. It is also possible that these classes overlap.

```
from ppci.arch.isa import Register

class X86Register(Register):
    bitsize=32

class LowX86Register(Register):
    bitsize=8

AL = LowX86Register('al', num=0)
AH = LowX86Register('ah', num=4)
EAX = X86Register('eax', num=0, aliases(AL, AH))
```

### 2.9.2 Architecture description

## 2.10 Hexfile manipulation

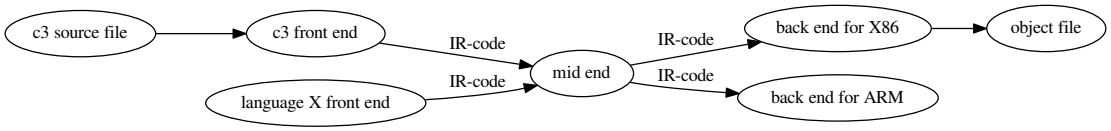
**class** ppci.utils.hexfile.**HexFile**
Represents an intel hexfile

```
>>> from ppci.utils.hexfile import HexFile
>>> h = HexFile()
>>> h.dump()
Hexfile containing 0 bytes
>>> h.add_region(0, bytes([1,2,3]))
>>> h
Hexfile containing 3 bytes
```

## 2.11 Compiler design

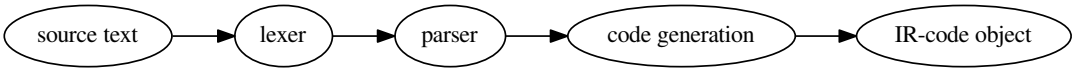
This chapter describes the design of the compiler. The compiler consists a frontend, mid-end and back-end. The frontend deals with source file parsing and semantics checking. The mid-end performs optimizations. This is optional. The back-end generates machine code. The front-end produces intermediate code. This is a simple representation of the source. The back-end can accept this kind of representation.

The compiler is greatly influenced by the LLVM design.




### 2.11.1 C3 Front-end

This is the c3 language front end. For the front-end a recursive descent parser is created.




**class** ppci.lang.c3.**C3Builder**(*diag*, *arch*)
Generates IR-code from c3 source. Reports errors to the diagnostics system.

**class** ppci.lang.c3.**Lexer**(*diag*)
Generates a sequence of token from an input stream

**class** ppci.lang.c3.**Parser**(*diag*)
Parses sourcecode into an abstract syntax tree (AST)

**class** ppci.lang.c3.**CodeGenerator**(*diag*, *debug_db*)
Generates intermediate (IR) code from a package. The entry function is 'genModule'. The main task of this part is to rewrite complex control structures, such as while and for loops into simple conditional jump statements. Also complex conditional statements are simplified. Such as 'and' and 'or' statements are rewritten in conditional jumps. And structured datatypes are rewritten.

Type checking is done in one run with code generation.

### 2.11.2 Brainfuck frontend

The compiler has a front-end for the brainfuck language.

**class** ppci.lang.bf.**BrainFuckGenerator**(*arch*)
Brainfuck is a language that is so simple, the entire front-end can be implemented in one pass.

### 2.11.3 IR-code

The intermediate representation (IR) of a program de-couples the front end from the backend of the compiler.

See IR-code for details about all the available instructions.

### 2.11.4 Optimization

The IR-code generated by the front-end can be optimized in many ways. The compiler does not have the best way to optimize code, but instead has a bag of tricks it can use.

**class** ppci.opt.**Mem2RegPromotor**(*debug_db*)
Tries to find alloc instructions only used by load and store instructions and replace them with values and phi nodes

**class** ppci.opt.**LoadAfterStorePass**(*debug_db*)
Remove load after store to the same location.

```
[x] = a
b = [x]
c = b + 2
```

transforms into:

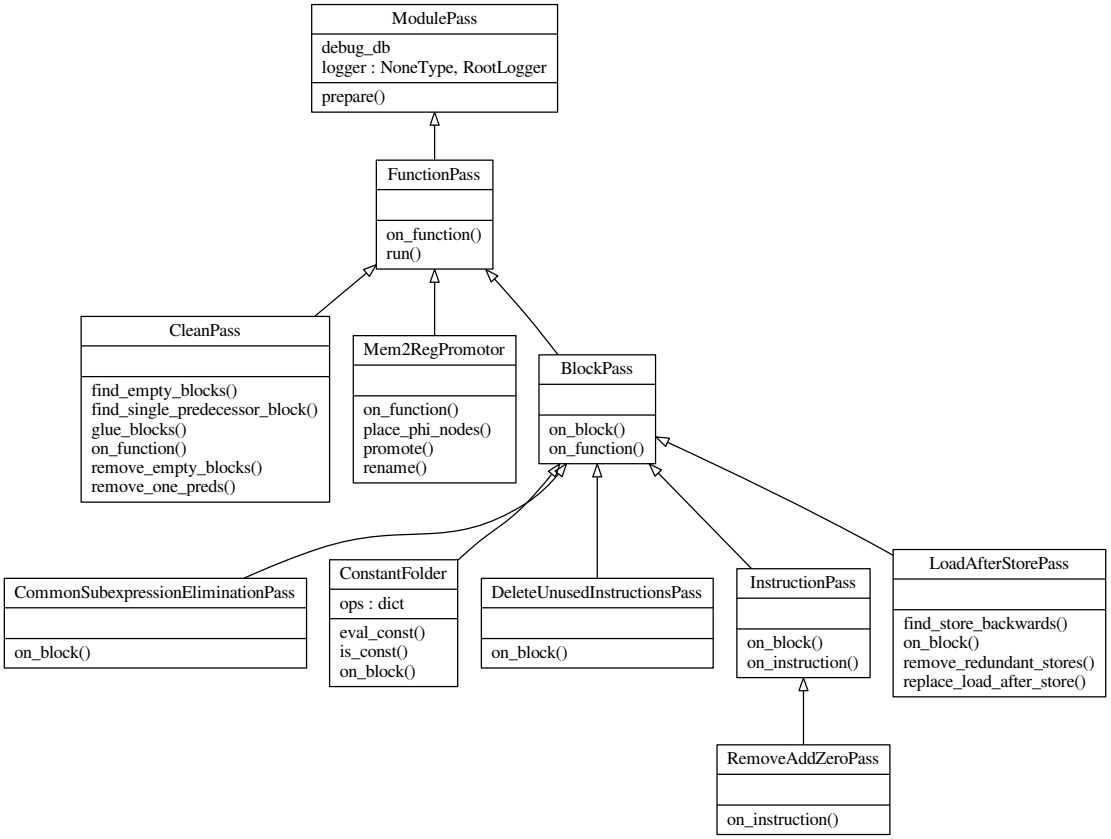
```
[x] = a
c = a + 2
```

**class** ppci.opt.**DeleteUnusedInstructionsPass**(*debug_db*)
Remove unused variables from a block

**class** ppci.opt.**RemoveAddZeroPass**(*debug_db*)
Replace additions with zero with the value itself. Replace multiplication by 1 with value itself.

**class** ppci.opt.**CommonSubexpressionEliminationPass**(*debug_db*)
Replace common sub expressions (cse) with the previously defined one.

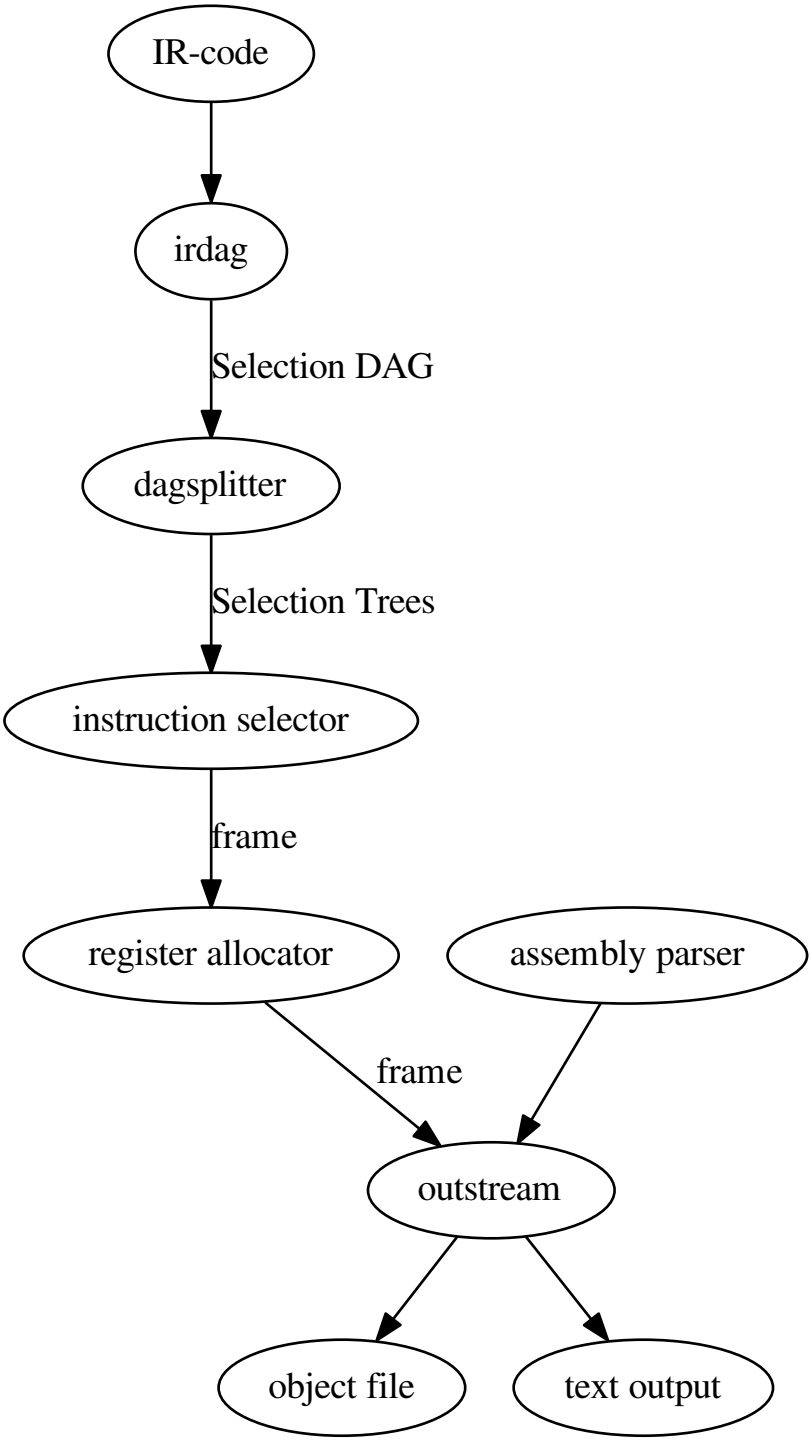
### Uml




## 2.11.5 Back-end

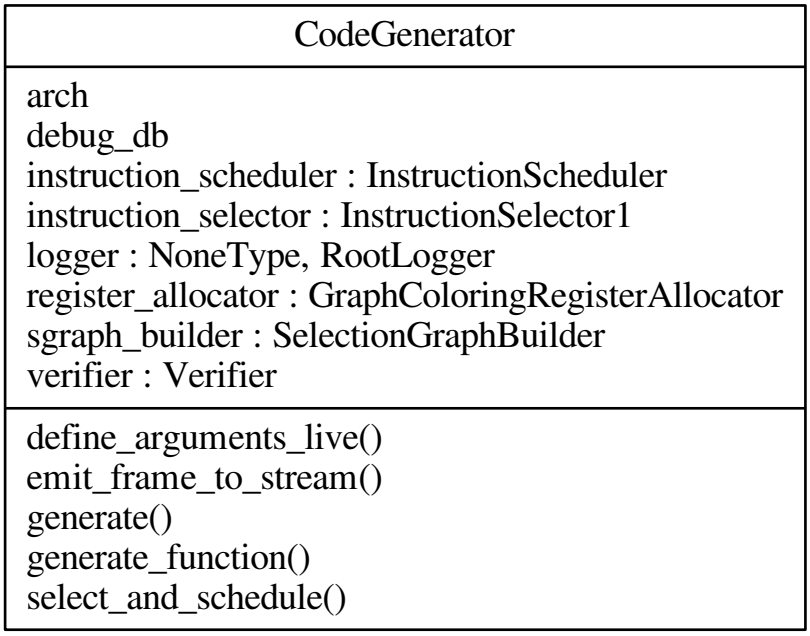
The back-end is more complicated. There are several steps to be taken here.

1. Tree creation
2. Instruction selection
3. Register allocation
4. Peep hole optimization




### Code generator

Machine code generator. The architecture is provided when the generator is created.




### Canonicalize

During this phase, the IR-code is made simpler. Also unsupported operations are rewritten into function calls. For example soft floating point is introduced here.

### Tree building

From IR-code a tree is generated which can be used to select instructions. The process of instruction selection is preceeded by the creation of a selection dag (directed acyclic graph). The dagger take ir-code as input and produces such a dag for instruction selection.

A DAG represents the logic (computation) of a single basic block.

### Instruction selection

The instruction selection phase takes care of scheduling and instruction selection. The output of this phase is a one frame per function with a flat list of abstract machine instructions.

To select instruction, a tree rewrite system is used. This is also called bottom up rewrite generator (BURG). See pyburg.

### Register allocation

Selected instructions use virtual registers and physical registers. Register allocation is the process of assigning a register to the virtual registers.

Some key concepts in the domain of register allocation are:

- **virtual register**: A value which must be mapped to a physical register.

- **physical register**: A real register
- **interference graph**: A graph in which each node represents a register. An edge indicates that the two registers cannot have the same register assigned.
- **pre-colored register**: A register that is already assigned a specific physical register.
- **coalescing**: The process of merging two nodes in an interference graph which do not interfere and are connected by a move instruction.
- **spilling**: Spilling is the process when no physical register can be found for a certain virtual register. Then this value will be placed in memory.
- **register class**: Most CPU's contain registers grouped into classes. For example, there may be registers for floating point, and registers for integer operations.
- **register alias**: Some registers may alias to registers in another class. A good example are the x86 registers rax, eax, ax, al and ah.

**Interference graph**

Each instruction in the instruction list may use or define certain registers. A register is live between a use and define of a register. Registers that are live at the same point in the program interfere with each other. An interference graph is a graph in which each node represents a register and each edge represents interference between those two registers.

**Graph coloring**

In 1981 Chaitin presented the idea to use graph coloring for register allocation.

In a graph a node can be colored if it has less neighbours than possible colors. This is true because when each neighbour has a different color, there is still a valid color left for the node itself.

Given a graph, if a node can be colored, remove this node from the graph and put it on a stack. When added back to the graph, it can be given a color. Now repeat this process recursively with the remaining graph. When the graph is empty, place all nodes back from the stack one by one and assign each node a color when placed. Remember that when adding back, a color can be found, because this was the criteria during removal.

See: https://en.wikipedia.org/wiki/Chaitin%27s_algorithm

*[Chaitin1982]*

**Coalescing**

Coalescing is the process of merging two nodes in an interference graph. This means that two temporaries will be assigned to the same register. This is especially useful if the temporaries are used in move instructions, since when the source and the destination of a move instruction are the same register, the move can be deleted.

Coalescing a move instruction is easy when an interference graph is present. Two nodes that are used by a move instruction can be coalesced when they do not interfere.

However, if we coalesc all moves, the graph can become incolorable, and spilling has to be done. To prevent spilling, coalescing must be done conservatively.

A conservative approach is the following: if the merged node has fewer than K nodes of significant degree, then the nodes can be coalesced. The prove for this is that when all nodes that can be colored are removed and the merged node and its non-colorable neighbours remain, the merged node can be colored. This ensures that the coalescing of the node does not have a negative effect on the colorability of the graph.

*[Briggs1994]*

**Spilling**

**Iterated register coalescing**

Iterated register coalescing (IRC) is a combination of graph coloring, coalescing and spilling. The process consists of the following steps:

- build an interference graph from the instruction list
- remove trivial colorable nodes.

- (optional) coalesc registers to remove redundant moves
- (optional) spill registers
- select registers

See: https://en.wikipedia.org/wiki/Register_allocation

*[George1996]*

**Graph coloring with more register classes**

Most instruction sets are not ideal, and hence simple graph coloring cannot be used. The algorithm can be modified to work with several register classes that possibly interfere.

*[Runeson2003] [Smith2004]*

**Implementations**

The following class can be used to perform register allocation.

**class** ppci.codegen.registerallocator.**GraphColoringRegisterAllocator**(*arch: ppci.arch.arch.Architecture*, *instruction_selector*, *debug_db*)

Target independent register allocator. Algorithm is iterated register coalescing by Appel and George. Also the pq-test algorithm for more register classes is added.

**alloc_frame**(*frame*)

Do iterated register allocation for a single frame.

This is the entry function for the register allocator and drives through all stages of the algorithm.

**Parameters** **frame** – The frame to perform register allocation on.

**coalesc**()

Coalesc moves conservative.

This means, merge the variables of a move into one variable, and delete the move. But do this only when no spill will occur.

**freeze**()

Give up coalescing on some node, move it to the simplify list and freeze all moves associated with it.

**is_colorable**(*node*)

Helper function to determine whether a node is trivially colorable. This means: no matter the colors of the nodes neighbours, the node can be given a color.

In case of one register class, this is: n.degree < self.K

In case of more than one register class, somehow the worst case damage by all neighbours must be determined.

We do this now with the pq-test.

**simplify**()

Remove nodes from the graph

### code emission

Code is emitted using the outputstream class. The assembler and compiler use this class to emit instructions to. The stream can output to object file or to a logger.

**class** ppci.binutils.outstream.**OutputStream**

Interface to generate code with. Contains the emit function to output instruction to the stream

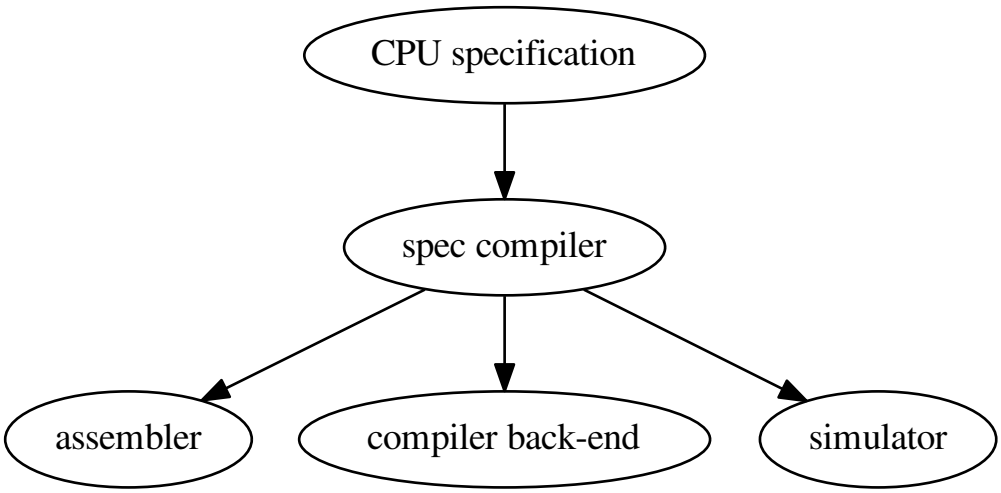
## 2.12 Specification languages

### 2.12.1 Introduction

*DRY*

Do not repeat yourself (DRY). This is perhaps the most important idea to keep in mind when writing tools like assemblers, disassemblers, linkers, debuggers and compiler code generators. Writing these tools can be a repetitive and error prone task.

One way to achieve this is to to write a specification file for a specific processor and generate from this file the different tools. The goal of a machine description file is to describe a file and generate tools like assemblers, disassemblers, linkers, debuggers and simulators.




### 2.12.2 Background

There are several existing languages to describe machines in a Domain Specific Language (DSL). Examples of these are:

- Tablegen (llvm)
- cgen (gnu)
- LISA (Aachen)
- nML (Berlin)
- SLED (Specifying representations of machine instructions (norman ramsey and Mary F. Fernandez)) *[sled]*

Concepts to use in this language:

- Single stream of instructions
- State stored in memory
- Pipelining
- Instruction semantics

Optionally a description in terms of compiler code generation can be attached to this. But perhaps this clutters the description too much and we need to put it elsewhere.

The description language can help to expand these descriptions by expanding the permutations.

### 2.12.3 Example specifications

For a complete overview of ADL (Architecture Description Language) see *[overview]*.

#### llvm

```
def IMUL64rr : RI<0xAF, MRMSrcReg, (outs GR64:$dst),
                                   (ins GR64:$src1, GR64:$src2),
                    "imul{q}\t{$src2, $dst|$dst, $src2}",
                    [(set GR64:$dst, EFLAGS,
                        (X86smul_flag GR64:$src1, GR64:$src2))],
                    IIC_IMUL64_RR>,
                TB;
```

#### LISA

```
<insn> BC
{
  <decode>
  {
    %ID: {0x7495, 0x0483}
    %cond_code: { %OPCODE1 & 0x7F }
    %dest_address: { %OPCODE2 }
  }
  <schedule>
  {
    BC1(PF, w:ebus_addr, w:pc) |
    BC2(PF, w:pc), BC3(IF) |
    BC4(ID) |
    <if> (condition[cond_code])
    {
      BC5(AC) |
      BC6(PF), BC7(ID), BC8(RE) |
      BC9(EX)
    }
    <else>
    {
      k:NOP(IF), BC10(AC, w:pc) |
      BC11(PF), BC12(ID), BC13(RE) |
      k:NOP(ID), BC14(EX) |
      k:NOP(ID), k:NOP(AC) |
      k:NOP(AC), k:NOP(RE) |
      k:NOP(RE), k:NOP(EX) |
      k:NOP(EX)
    }
  }
  <operate>
  {
    BC1.control: { ebus_addr = pc++; }
    BC2.control: { ir = mem[ebus_addr]; pc++ }
    BC10.control: { pc = (%OPCODE2) }
  }
}
```

### SLED

```
patterns
  nullary is any of [ HALT NEG COM SHL SHR READ WRT NEWL NOOP TRA NOTR ],
    which is op = 0 & adr = { 0 to 10 }
constructors
  IMULb        Eaddr            is      (grp3.Eb;    Eaddr) & IMUL.AL.eAX
```

### nML

```
type word = card(16)
type absa = card(9)
type disp = int(4)
type off = int(6)
mem PC[1,word]
mem R[16,word]
mem M[65536,word]
var L1[1,word]
var L2[1,word]
var L3[1,word]
mode register(i:card(4)) = R[i]
  syntax = format("R%s", i)
  image = format("%4b", i)
mode memory = ind | post | abs
mode ind(r:register, d:disp) = M[r+d]
  update = {}
  syntax = format("@%s(%d)", r.syntax, d)
  image = format("0%4b%4b0", r.image, d)
mode post(r:register, d:disp) = M[r+d]
  update = { r = r + 1; }
  syntax = format("@%s++(%d)", r.syntax, d)
  image = format("0%4b%4b1", r.image, d)
mode abs(a : absa) = M[a]
  update = {}
  syntax = format("%d", a)
  image = format("1%9b", a)
op instruction( i : instr )
  syntax = i.syntax
  image = i.image
  action = {
    PC = PC + 1;
    i.action;
  }
op instr = move | alu | jump
op move(lore:card(1), r:register, m:memory)
  syntax = format("MOVE%d %s %s", lore, r.syntax, m.syntax)
  image = format("0%1b%4b%10b", lore, r.image, m.image)
  action = {
    if ( lore ) then r = m;
    else m = r;
    endif;
    m.update;
  }
op alu(s1:register, s2:register, d:reg, a:aluop)
  syntax = format("%s %s %s %s", a.syntax, s1.syntax, s2.syntax, d.syntax)
  image = format("10%4b%4b%4b%2b", s1.image, s2.image, d.image, a.image)
  action = {
    L1 = s1; L2 = s2; a.action; d = L3;
  }
op jump(s1:register, s2:register, o:off)
  syntax = format("JUMP %s %s %d", s1.syntax, s2.syntax, o)
```

```
  image = format("11%4b%4b%6b", s1.image, s2.image, o)
  action = {
   if ( s1 >= S2 ) then PC = PC + o;
   endif;
  }
op aluop = and | add | sub | shift;
op and() syntax = "and" image = "00" action = { L3 = L1 & L2; }
op add() syntax = "add" image = "10" action = { L3 = L1 + L2; }
op sub() syntax = "sub" image = "01" action = { L3 = L1 - L2; }
```

### 2.12.4 Design

The following information must be captured in the specification file:

- Assembly textual representation
- Binary representation
- Link relocations
- Mapping from compiler back-end
- Effects of instruction (semantics)

## 2.13 Links

### 2.13.1 Classical compilers

The following list gives a summary of some compilers that exist in the open source land.

- *LLVM*

  A relatively new and popular compiler. LLVM stands for low level virtual machine, a compiler written in C++. This is a big inspiration place for ppci! http://llvm.org

- *GCC*

  The gnu compiler. The famous open source compiler. https://gcc.gnu.org/

- *ACK*

  The amsterdam compiler kit. http://tack.sourceforge.net/

- *lcc*

  A retargetable C compiler. https://github.com/drh/lcc

### 2.13.2 Other compilers written in python

- *zxbasic*

  Is a freebasic compiler written in python. http://www.boriel.com/wiki/en/index.php/ZXBasic

- *python-msp430-tools*

  A msp430 tools project in python. https://launchpad.net/python-msp430-tools

### 2.13.3 Citations

CHAPTER 3

# Contributing

This section is intended for contributors of ppci.

## 3.1 Support

Ppci is still in development, and all help is welcome!

### 3.1.1 How to help

If you want to add some code to the project, the best way is to create a fork of the project, make your changes and create a pull request.

For a list of tasks, refer to the todo page. For help on getting started with development see the development page.

### 3.1.2 Report an issue

If you find a bug, you can file it here:

https://bitbucket.org/windel/ppci/issues

### 3.1.3 Donate

If you use this project, and want to donate money, you can do this through this pledgie campaign:

https://pledgie.com/campaigns/31115

## 3.2 Development

This chapter descibes how to develop on ppci.

### 3.2.1 Communication

Join the #ppci irc channel on freenode!

Or visit the forum:

- https://groups.google.com/d/forum/ppci-dev

### 3.2.2 Source code

The sourcecode repository of the project is located at these locations:

- https://bitbucket.org/windel/ppci
- https://pikacode.com/windel/ppci/
- https://mercurial.tuxfamily.org/ppci/ppci

To check out the latest code and work use the development version use these commands to checkout the source code and setup ppci such that you can use it without having to setup your python path.

```
$ mkdir HG
$ cd HG
$ hg clone https://bitbucket.org/windel/ppci
$ cd ppci
$ sudo python setup.py develop
```

### 3.2.3 Running the testsuite

To run the unit tests with the compiler, use pytest:

```
$ python -m pytest -v test/
```

Or use the unittest module:

```
$ python -m unittest discover -s test
```

In order to test ppci versus different versions of python, tox is used. To run tox, simply run in the root directory:

```
$ tox
```

### 3.2.4 Building the docs

The docs can be build locally by using sphinx. Sphinx can be invoked directly:

```
$ cd docs
$ sphinx-build -b html . build
```

Alternatively the tox docs environment can be used:

```
$ tox -e docs
```

### 3.2.5 Release procedure

1. Determine the version numbers of this release and the next.
2. Switch to the release branch and merge the default branch into the release branch.

```
        $ hg update release
        $ hg merge default
        $ hg commit
```

3. Check the version number in ppci/__init__.py
4. Make sure all tests pass and fix them if not.

```
        $ tox
```

5. Tag this release with the intended version number and update to this tag.

```
$ hg tag x.y.z
$ hg update x.y.z
```

6. Package and upload the python package. The following command creates a tar gz archive as well as a wheel package.

```
$ python setup.py sdist bdist_wheel upload
```

7. Switch back to the default branch and merge the release branch into the default branch.

```
$ hg update default
$ hg merge release
$ hg commit
```

8. Increase the version number in ppci/__init__.py.

### 3.2.6 Continuous integration

The compiler is tested for linux:

- https://drone.io/bitbucket.org/windel/ppci

and for windows:

- https://ci.appveyor.com/project/WindelBouwman/ppci-786

### 3.2.7 Code metrics

Code coverage is reported to the codecov service:

- https://codecov.io/bitbucket/windel/ppci?branch=default

Other code metrics are listed at openhub:

- https://www.openhub.net/p/ppci

## 3.3 Todo

Below is a list of features / tasks that need to be done.

- Improve the debugger.
  - Add support for local variables.
- Implement the disassembler further.
- Implement Mac OSX support and add a mac 64-bit example project.
- Add Windows support and add a windows 64-bit example project.
- Improve the fuzzer tool that can generate random source code to stress the compiler.
- Implement a fortran frontend. The ppci.lang.fortran module contains a start.
- Implement a C frontend, The ppci.lang.c module contains an attempt.
- Add a pascal frontend.
- Add floating point support.
- Add a peephole optimizer.
- Add spilling to the register allocator.
- Add a front-end for LLVM IR-code, this way, the front-end of LLVM can be used and the backend of ppci.

- Add a backend for LLVM IR-code.
- Add better support for harvard architecture cpu's like avr, 8051 and PIC.

CHAPTER 4

# Faq

*Why? WHY?!*

There are several reasons:

- it is possible!
- this compiler is very portable due to python being portable.
- writing a compiler is a very challenging task

*Is this compiler slower than compilers written in C/C++?*

Yes. Although a comparison is not yet done, this will be the case, due to the overhead and slower execution of python code.

*Cool project, I want to contribute to this project, what can I do?*

Great! If you want to add some code to the project, the best way is perhaps to send me a message, and create a fork of the project on bitbucket. If you are not sure where to begin, please contact me fist. For a list of tasks, refer to the todo page. For hints on development see the development page.

CHAPTER 5

# Changelog

## 5.1 Release 1.0 (Planned)

- platform.python_compiler() returns 'ppci 1.0'

## 5.2 Release 0.6 (Planned)

## 5.3 Release 0.5 (Upcoming)

- Debug type information stored in better format
- Expression evaluation in debugger
- Global variables can be viewed
- Improved support for different register classes

## 5.4 Release 0.4.0 (Apr 27, 2016)

- Start with debugger and disassembler

## 5.5 Release 0.3.0 (Feb 23, 2016)

- Added risc v architecture
- Moved thumb into arm arch
- msp430 improvements

## 5.6 Release 0.2.0 (Jan 23, 2016)

- Added linker (ppci-ld.py) command
- Rename *buildfunctions* to *api*
- Rename *target* to *arch*

## 5.7 Release 0.1.0 (Dec 29, 2015)

- Added x86_64 target.
- Added msp430 target.

## 5.8 Release 0.0.5 (Mar 21, 2015)

- Remove st-link and hence pyusb dependency.
- Support for pypy3.

## 5.9 Release 0.0.4 (Feb 24, 2015)

## 5.10 Release 0.0.3 (Feb 17, 2015)

## 5.11 Release 0.0.2 (Nov 9, 2014)

## 5.12 Release 0.0.1 (Oct 10, 2014)

- Initial release.

[sled] http://www.cs.tufts.edu/~nr/toolkit/

[overview] http://esl.cise.ufl.edu/Publications/iee05.pdf

[Smith2004] "A generalized algorithm for graph-coloring register allocation", 2004, Michael D. Smith and Norman Ramsey and Glenn Holloway.

[Runeson2003] "Retargetable Graph-Coloring Register Allocation for Irregular Architectures", 2003, Johan Runeson and Sven-Olof Nystrom. http://user.it.uu.se/~svenolof/wpo/AllocSCOPES2003.pdf

[George1996] "Iterated Register Coalescing", 1996, Lal George and Andrew W. Appel.

[Briggs1994] "Improvements to graph coloring register allocation", 1994, Preston Briggs, Keith D. Cooper and Linda Torczon

[Chaitin1982] "Register Allocation and Spilling via Graph Coloring", 1982, G. J. Chaitin.

## p

## Symbols


–layout <layout-file>, -L <layout-file>
ppci-ld.py command line option, 7
–log <log-level>
ppci-asm.py command line option, 6
ppci-build.py command line option, 6
ppci-c3c.py command line option, 5
ppci-ld.py command line option, 7
ppci-objcopy.py command line option, 7
ppci-objdump.py command line option, 8
–machine, -m
ppci-asm.py command line option, 6
ppci-c3c.py command line option, 5
–mtune <option>
ppci-asm.py command line option, 6
ppci-c3c.py command line option, 5
–output, -o
ppci-asm.py command line option, 7
ppci-c3c.py command line option, 5
ppci-ld.py command line option, 7
–output-format <output_format>, -O <output_format>
ppci-objcopy.py command line option, 8
–report
ppci-asm.py command line option, 6
ppci-build.py command line option, 6
ppci-c3c.py command line option, 5
ppci-ld.py command line option, 7
ppci-objcopy.py command line option, 7
ppci-objdump.py command line option, 8
–segment , -S 
ppci-objcopy.py command line option, 8
–verbose, -v
ppci-asm.py command line option, 6
ppci-build.py command line option, 6
ppci-c3c.py command line option, 5
ppci-ld.py command line option, 7
ppci-objcopy.py command line option, 7
ppci-objdump.py command line option, 8
–version, -V
ppci-asm.py command line option, 6
ppci-build.py command line option, 6
ppci-c3c.py command line option, 5
ppci-ld.py command line option, 7
ppci-objcopy.py command line option, 8
ppci-objdump.py command line option, 8
-O {0,1,2,s}
ppci-c3c.py command line option, 6
-d, –disassemble
ppci-objdump.py command line option, 8
-f <build-file>, –buildfile <build-file>
ppci-build.py command line option, 6
-g
ppci-c3c.py command line option, 6
ppci-ld.py command line option, 7
-g, –debug
ppci-asm.py command line option, 7
-h, –help
ppci-asm.py command line option, 6
ppci-build.py command line option, 6
ppci-c3c.py command line option, 5
ppci-ld.py command line option, 7
ppci-objcopy.py command line option, 7
ppci-objdump.py command line option, 8
-i <include>, –include <include>
ppci-c3c.py command line option, 6


## A



## B

new_name() (ppci.arch.arch.Frame method), 18
new_reg() (ppci.arch.arch.Frame method), 18

## O

obj
    ppci-ld.py command line option, 7
    ppci-objdump.py command line option, 8
objcopy() (in module ppci.api), 4
optimize() (in module ppci.api), 4
output
    ppci-objcopy.py command line option, 7
OutputStream (class in ppci.binutils.outstream), 28

## P

Parser (class in ppci.lang.c3), 22
pattern() (ppci.arch.isa.Isa method), 18
peephole() (ppci.arch.isa.Isa method), 19
Phi (class in ppci.ir), 14
ppci-asm.py command line option
    –log <log-level>, 6
    –machine, -m, 6
    –mtune <option>, 6
    –output, -o, 7
    –report, 6
    –verbose, -v, 6
    –version, -V, 6
    -g, –debug, 7
    -h, –help, 6
    sourcefile, 6
ppci-build.py command line option
    –log <log-level>, 6
    –report, 6
    –verbose, -v, 6
    –version, -V, 6
    -f <build-file>, –buildfile <build-file>, 6
    -h, –help, 6
    target, 6
ppci-c3c.py command line option
    –log <log-level>, 5
    –machine, -m, 5
    –mtune <option>, 5
    –output, -o, 5
    –report, 5
    –verbose, -v, 5
    –version, -V, 5
    -O {0,1,2,s}, 6
    -g, 6
    -h, –help, 5
    -i <include>, –include <include>, 6
    source, 5
ppci-ld.py command line option
    –layout <layout-file>, -L <layout-file>, 7
    –log <log-level>, 7
    –output, -o, 7
    –report, 7
    –verbose, -v, 7
    –version, -V, 7
    -g, 7
    -h, –help, 7
    obj, 7
ppci-objcopy.py command line option
    –log <log-level>, 7
    –output-format <output_format>, -O <output_format>, 8
    –report, 7
    –segment , -S , 8
    –verbose, -v, 7
    –version, -V, 8
    -h, –help, 7
    input, 7
    output, 7
ppci-objdump.py command line option
    –log <log-level>, 8
    –report, 8
    –verbose, -v, 8
    –version, -V, 8
    -d, –disassemble, 8
    -h, –help, 8
    obj, 8
ppci.api (module), 3
ppci.arch (module), 17
ppci.arch.arm (module), 20
ppci.arch.avr (module), 20
ppci.arch.mos6500 (module), 20
ppci.arch.msp430 (module), 20
ppci.arch.riscv (module), 20
ppci.arch.stm8 (module), 20
ppci.arch.x86_64 (module), 21
ppci.codegen.codegen (module), 25
ppci.codegen.irdag (module), 26
ppci.codegen.registerallocator (module), 26
ppci.lang.c3 (module), 22
predecessors (ppci.ir.Block attribute), 12
Procedure (class in ppci.ir), 12
ProcedureCall (class in ppci.ir), 14
prologue() (ppci.arch.arch.Architecture method), 18
ptr (in module ppci.ir), 13

## R

read_mem() (ppci.binutils.dbg.Debugger method), 15
reads_register() (ppci.arch.isa.Instruction method), 19
Register (class in ppci.arch.isa), 19
register_pattern() (ppci.arch.isa.Isa method), 19
register_relocation() (ppci.arch.isa.Isa method), 19
registers (ppci.arch.isa.Instruction attribute), 19
remove_block() (ppci.ir.SubRoutine method), 12
remove_instruction() (ppci.ir.Block method), 12
RemoveAddZeroPass (class in ppci.opt), 23
replace_register() (ppci.arch.isa.Instruction method), 19
Return (class in ppci.ir), 14
RiscvArch (class in ppci.arch.riscv), 20
run() (ppci.binutils.dbg.Debugger method), 15
runtime (ppci.arch.arch.Architecture attribute), 18

## S



## T



## U



## V



## W



## X