ppci Documentation

Release 0.4.0

Windel Bouwman

April 27, 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.

Download and unzip the examples bundle project here `examples.zip`.

1.2 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.

1.3 arduino example

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

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

1.4 x86_64 example

1.4.1 Linux

Instead of for a board you can compile into a native linux binary:

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

Or run the snake demo under linux natively instead of on qemu:

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

1.4.2 Windows

TODO

1.4.3 Mac

TODO

1.5 msp430 example

Flash program:

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

CHAPTER 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
```

2.1 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] 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
optimize code

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

2.3 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 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

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] [–layout layout-file] –output output-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

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

--output, -o
output file

-g
retain debug information

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

2.6 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] 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

2.7 ppci-hexutil.py

hexfile manipulation tool by Windel Bouwman

usage: ppci-hexutil.py [-h] {info,new,merge} ...

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

2.7.1 ppci-hexutil.py info

usage: ppci-hexutil.py [-h] {info,new,merge} ...

hexfile

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

2.7.2 ppci-hexutil.py merge

usage: ppci-hexutil.py [-h] {info,new,merge} ...

hexfile1
hexfile 1

hexfile2
hexfile 2

rhexfile
resulting hexfile

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

2.7.3 ppci-hexutil.py new

usage: ppci-hexutil.py [-h] {info,new,merge} ...

hexfile

address
hex address of the data

datafile
binary file to add

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

CHAPTER 3

Api

Instead of using the commandline, it is also possible to use ppci api. For example to assemble, compile, link and objcopy the msp430 blinky example project:

```
>>> 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')
```

3.1 api module

This module contains a set of handy functions to invoke compilation, linking and assembling.

ppci.api.**asm**(*source*, *march*)

Assemble the given source for machine march.

source can be a filename or a file like object. march can be a machine instance or a string indicating the target.

For example:

```
>>> 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*, *reporter=None*, *debug=False*)

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

For example:

```
>>> 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*, *arch=None*, *use_runtime=False*, *partial_link=False*, *reporter=None*, *debug=False*)
Links the iterable of objects into one using the given layout.

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

source can be a filename or a file like object. march can be a machine instance or a string indicating the target.

For example:

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

ppci.api.**construct**(*buildfile*, *targets=()*)
Construct the given buildfile. Raise task error if something goes wrong.

ppci.api.**optimize**(*ir_module*, *reporter=None*, *debug_db=None*)
Run a bag of tricks against the ir-code. This is an in-place operation!

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

CHAPTER 4

Reference

4.1 C3 language

4.1.1 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:

- 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.
- Etc...

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

The hello world example in c3:

```
module hello;
import io;

function void main()
{
    io.println("Hello world");
}
```

4.1.2 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!
    }
}
```

4.2 IR-code

Front ends generate this IR-code. Backends transform it into machine code.

4.2.1 Top level structure

The IR-code is implemented in the ir package.

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

class ppci.ir.**Function**(*name*, *module=None*)
Represents a function.

class ppci.ir.**Block**(*name*, *function=None*)
Uninterrupted sequence of instructions with a label at the start.

4.2.2 Statements

A block contains a sequence of statements.

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

class ppci.ir.**Binop**(*a*, *operation*, *b*, *name*, *ty*, *loc=None*)
Generic binary operation

class ppci.ir.**Call**(*function_name*, *arguments*, *name*, *ty*, *loc=None*)
Call a function with some arguments

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.

4.3 Debug

4.4 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="lm3s6965/startup.asm"
            target="arm:thumb"
            output="startup.o" />
        <compile
            target="arm:thumb"
            sources="snake/main.c3;lm3s6965/bsp.c3;../librt/io.c3;snake/game.c3"
            output="snake.o"
            report="snake_report.html"/>
        <link output="snake.elf"
            layout="lm3s6965/memlayout.mmap"
            target="arm:thumb"
            objects="startup.o;snake.o" />
        <objcopy
            objectfile="snake.elf"
            imagename="flash"
            format="bin"
            output="snake.bin" />
    </target>

</project>
```

4.4.1 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.

4.4.2 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.

4.4.3 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.

4.5 Backends

4.5.1 6500

Status: 1%

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

4.5.2 arm

Status: 70% 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.

4.5.3 avr

Status: 20%

class ppci.arch.avr.**AvrArch**(*options=None*)
Check this site for good info: - https://gcc.gnu.org/wiki/avr-gcc

4.5.4 msp430

Status: 20%

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

4.5.5 risc-v

See also: http://riscv.org

Status: 30%

Contributed by Michael.

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

4.5.6 stm8

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

Status: 0%

4.5.7 x86_64

Status: 60%

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

4.6 Hexfile manipulation

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

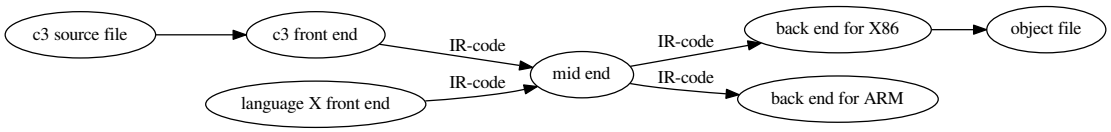
```
>>> from ppci.utils.hexfile import HexFile
>>> h = HexFile()
>>> h.dump()
Hexfile containing 0 bytes
>>> h.add_region(0, bytes([1,2,3]))
>>> h
Hexfile containing 3 bytes
```

CHAPTER 5

Compiler design

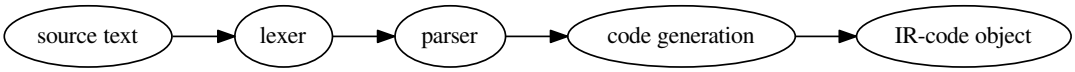
This chapter describes the design of the compiler. The compiler consists a frontend, mid-end and back-end. The frontend deals with source file parsing and semantics checking. The mid-end performs optimizations. This is optional. The back-end generates machine code. The front-end produces intermediate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



5.1 C3 Front-end

For the front-end a recursive descent parser is created for the c3 language. This is a subset of the C language with some additional features.



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.

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

5.2 Brainfuck frontend

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

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

5.3 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.

5.4 Optimalization

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.transform.**ModulePass**(*debug_db*)
Base class of all optimizing passes. Subclass this class to implement your own optimization pass

class ppci.opt.mem2reg.**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.transform.**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.transform.**DeleteUnusedInstructionsPass**(*debug_db*)
Remove unused variables from a block

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

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

5.5 Back-end

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

1. Canonicalization

2. Tree creation
3. Instruction selection
4. register allocation
5. Instruction emission
6. TODO: Peep hole optimization?

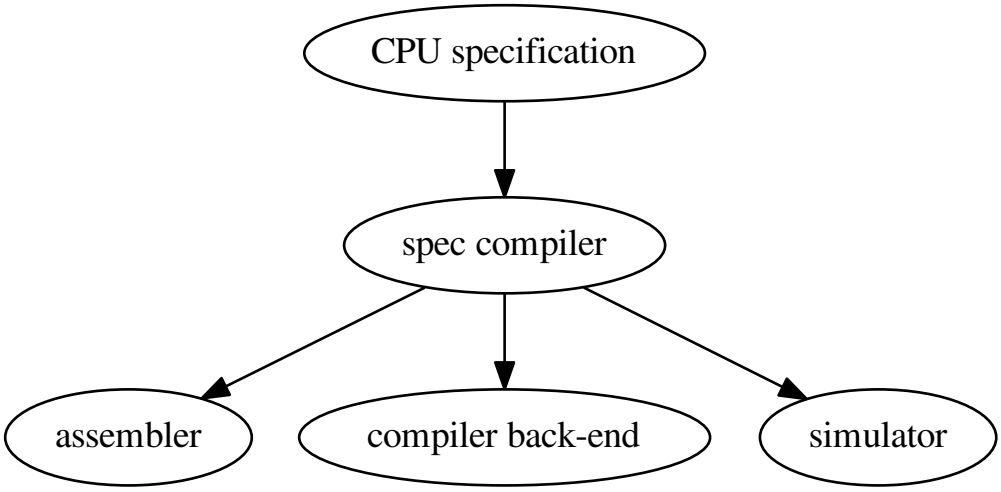
5.5.1 Specification languages

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.



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.

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; }
```

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)

5.5.2 Code generator

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

5.5.3 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.

5.5.4 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.

5.5.5 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.

5.5.6 Register allocation

The selected instructions are used to select correct registers.

class ppci.codegen.registerallocator.**RegisterAllocator**(*arch*, *debug_db*)

Target independent register allocator.

Algorithm is iterated register coalescing by Appel and George.

Chaitin's algorithm: remove all nodes with less than K neighbours. These nodes can be colored when added back.

The process consists of the following steps:

- •build interference graph from the instruction list
- •remove low degree non move related nodes.
- •(optional) coalesc registers to remove redundant moves
- •(optional) spill registers
- •select registers

TODO: Implement different register classes

5.5.7 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

5.6 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.

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

class ppci.binutils.dbg.**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.

CHAPTER 6

Development

This chapter descibes how to develop on ppci.

6.1 Communication

Join the #ppci irc channel on freenode!

6.2 Source code

The sourcecode of the project is located at these repositories:

- https://bitbucket.org/windel/ppci
- https://pikacode.com/windel/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
```

6.3 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

6.4 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

6.5 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
```

6.6 Building the docs

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

```
$ export PYTHONPATH=your_ppci_root
$ cd docs
$ sphinx-build -b html . build
```

Alternatively the tox docs environment can be used:

```
$ tox -e docs
```

6.7 Release procedure

Make sure all tests pass before a release.

Package and upload the python package with:

```
$ hg update release
$ hg merge default
# Check version number
$ tox
$ hg tag x.y.z
$ hg update x.y.z
$ python setup.py sdist upload
$ hg update default
$ hg merge release
```

Increase the version number.

CHAPTER 7

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 8

Todo

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

- Improve the debugger.
- Implement the disassembler further.
- 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.

CHAPTER 9

Changelog

9.1 Release 0.4.0 (Upcoming)

- Start with debugger and disassembler

9.2 Release 0.3.0 (Feb 23, 2016)

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

9.3 Release 0.2.0 (Jan 23, 2016)

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

9.4 Release 0.1.0 (Dec 29, 2015)

- Added x86_64 target.
- Added msp430 target.

9.5 Release 0.0.5 (Mar 21, 2015)

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

9.6 Release 0.0.4 (Feb 24, 2015)

9.7 Release 0.0.3 (Feb 17, 2015)

9.8 Release 0.0.2 (Nov 9, 2014)

9.9 Release 0.0.1 (Oct 10, 2014)

- Initial release.

CHAPTER 10

Links

10.1 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++. http://llvm.org

- *GCC*

  The gnu compiler. The workhorse for many a year. https://gcc.gnu.org/

- *ACK*

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

- *lcc*

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

10.2 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

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

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

p

Symbols

–layout <layout-file>, -L <layout-file>
ppci-ld.py command line option, 5
–log <log-level>
ppci-asm.py command line option, 5
ppci-build.py command line option, 4
ppci-c3c.py command line option, 3
ppci-ld.py command line option, 5
ppci-objcopy.py command line option, 6
ppci-objdump.py command line option, 6
–machine, -m
ppci-asm.py command line option, 5
ppci-c3c.py command line option, 4
–mtune <option>
ppci-asm.py command line option, 5
ppci-c3c.py command line option, 4
–output, -o
ppci-asm.py command line option, 5
ppci-c3c.py command line option, 4
ppci-ld.py command line option, 5
–output-format <output_format>, -O <output_format>
ppci-objcopy.py command line option, 6
–report
ppci-asm.py command line option, 5
ppci-build.py command line option, 4
ppci-c3c.py command line option, 3
ppci-ld.py command line option, 5
ppci-objcopy.py command line option, 6
ppci-objdump.py command line option, 6
–segment , -S
ppci-objcopy.py command line option, 6
–verbose, -v
ppci-asm.py command line option, 5
ppci-build.py command line option, 4
ppci-c3c.py command line option, 3
ppci-ld.py command line option, 5
ppci-objcopy.py command line option, 6
ppci-objdump.py command line option, 6
–version, -V
ppci-asm.py command line option, 5
ppci-build.py command line option, 4
ppci-c3c.py command line option, 3
ppci-ld.py command line option, 5
ppci-objcopy.py command line option, 6
ppci-objdump.py command line option, 6
-O
ppci-c3c.py command line option, 4
-f <build-file>, –buildfile <build-file>
ppci-build.py command line option, 4
-g
ppci-c3c.py command line option, 4
ppci-ld.py command line option, 5
-h, –help
ppci-asm.py command line option, 4
ppci-build.py command line option, 4
ppci-c3c.py command line option, 3
ppci-hexutil.py command line option, 7
ppci-hexutil.py-info command line option, 7
ppci-hexutil.py-merge command line option, 7
ppci-hexutil.py-new command line option, 7
ppci-ld.py command line option, 5
ppci-objcopy.py command line option, 6
ppci-objdump.py command line option, 6
-i <include>, –include <include>
ppci-c3c.py command line option, 4

A

B

hexfile, 7
ppci-hexutil.py-merge command line option
-h, –help, 7
hexfile1, 7
hexfile2, 7
rhexfile, 7
ppci-hexutil.py-new command line option
-h, –help, 7
address, 7
datafile, 7
hexfile, 7
ppci-ld.py command line option
–layout <layout-file>, -L <layout-file>, 5
–log <log-level>, 5
–output, -o, 5
–report, 5
–verbose, -v, 5
–version, -V, 5
-g, 5
-h, –help, 5
obj, 5
ppci-objcopy.py command line option
–log <log-level>, 6
–output-format <output_format>, -O <output_format>, 6
–report, 6
–segment , -S , 6
–verbose, -v, 6
–version, -V, 6
-h, –help, 6
input, 6
output, 6
ppci-objdump.py command line option
–log <log-level>, 6
–report, 6
–verbose, -v, 6
–version, -V, 6
-h, –help, 6
obj, 6
ppci.api (module), 9
ppci.arch.arm (module), 16
ppci.codegen.codegen (module), 24
ppci.codegen.irdag (module), 25

R

S

T

X